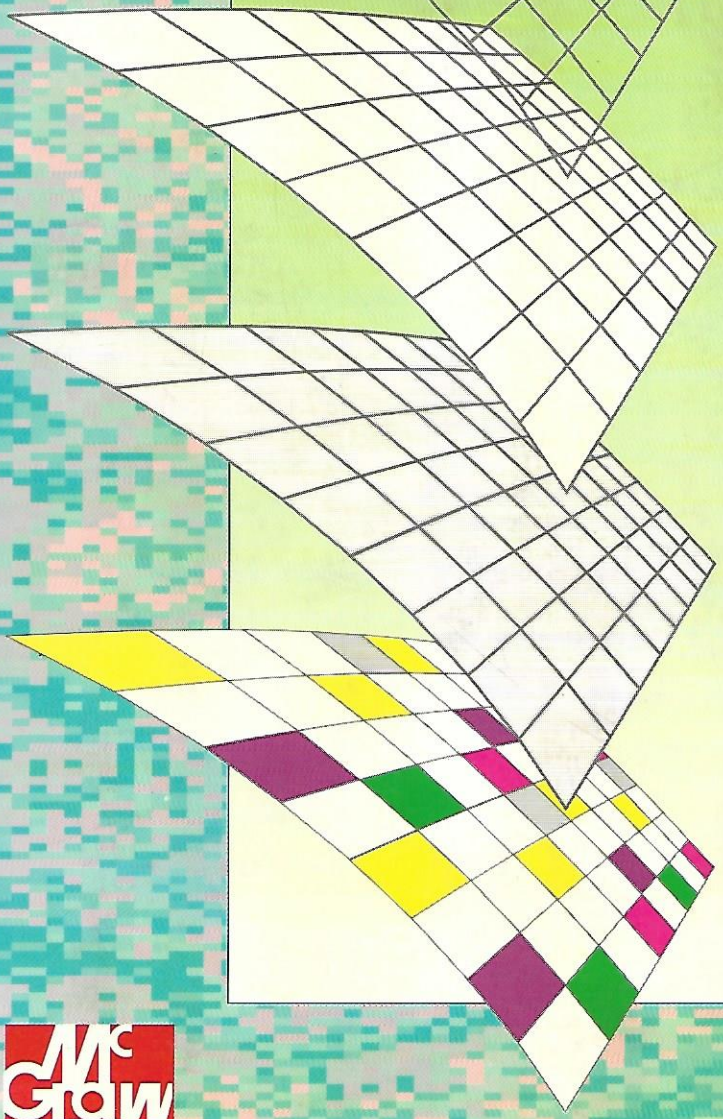
Diseño y manejo de estructuras de datos en C / Jorge A. Villalobos S

Diseño y Manejo de Estructuras de Datos en C



Jorge A. Villalobos S.

Mc Graw Hill

Prefacio

Este libro está dirigido a estudiantes de ingeniería de sistemas, con conocimientos básicos de programación en algún lenguaje de alto nivel, de preferencia C. En la Universidad de los Andes está planteado como el texto del tercer curso del ciclo de formación básica en informática, y supone que el estudiante maneja con cierta habilidad los conceptos básicos de la programación de computadores.

El objetivo del libro es servir como guía para un curso en diseño y manejo de estructuras de datos en C. Al final, el estudiante será capaz de diseñar las estructuras de datos, en memoria principal, más adecuadas para un problema específico, y desarrollar los algoritmos para el manejo de éstas. El libro utiliza metodologías de Tipos Abstractos de Datos y soporta todo el proceso de diseño en sólidas bases teóricas. Brinda al estudiante herramientas para la evaluación de soluciones, como la complejidad de algoritmos, de manera que cuente con criterios concretos de decisión. El libro no se queda en consideraciones teóricas, sino que muestra la dimensión práctica de las metodologías de diseño propuestas y la manera de aplicarlas para mejorar la calidad del *software* obtenido.

El lenguaje escogido para el libro es C, dada su enorme difusión en el medio informático, su eficiencia, su estandarización y porque ha sido seleccionado como el lenguaje de base por muchas universidades del mundo. Además, el contenido del libro se ajusta muy bien para ser utilizado en desarrollo de *software* en C++. Cada algoritmo que trae el libro tiene una especificación formal, una explicación de su funcionamiento y el cálculo de su complejidad. Todos los programas del libro han sido desarrollados y probados en ANSI C, sobre diferentes plataformas computacionales.

En cada capítulo, el estudiante encuentra un conjunto de ejemplos y ejercicios propuestos, tanto a nivel de diseño como a nivel de implementación. Esto permite al estudiante ver la aplicación de las metodologías propuestas en problemas reales y practicar para obtener destreza en su utilización. Los ejercicios están organizados por temas y clasificados por nivel de complejidad. En cada capítulo se dan algunas referencias bibliográficas a través de las cuales es posible profundizar en los temas allí tratados. Cada capítulo presenta un tipo de estructura de datos, un TAD que la administra y ejemplos de la parte algorítmica.

El libro viene apoyado por un disquete, que incluye la implementación y los programas de prueba de todos los algoritmos que son presentados en la parte teórica, lo mismo que la solución de algunos de los ejercicios propuestos. Esto permite al estudiante ver el funcionamiento de cada una de las rutinas planteadas, lo mismo que reutilizar el *software* para el desarrollo de sus propios proyectos. Existe también una guía completa del profesor, en la cual se hacen varias recomendaciones metodológicas para la presentación en clase del material, lo mismo que la solución de otros de los ejercicios propuestos a lo largo del libro.

Este libro es el producto de más de 8 años de trabajo en la Universidad de los Andes, en el curso Estructuras de Datos. Materiales previos han sido utilizados por más de 3000 estudiantes en ésta y otras universidades del país.

En el capítulo 0 se presentan algunos conceptos básicos de programación y las nociones elementales de análisis de algoritmos. En el capítulo 1 se dan las pautas generales de la metodología de diseño de

estructuras de datos. Su utilización se ilustra a lo largo de todo el libro a través del estudio de casos. Los capítulos 2 y 3 presentan las estructuras lineales de datos, tales como listas, pilas y colas. Los capítulos 4 y 5 estudian las estructuras recursivas de datos como los árboles binarios y los árboles n-arios. El capítulo 6 hace un estudio de estructuras de datos no lineales, como grafos dirigidos. Por último, el capítulo 7 trata el tema de estructuras de acceso directo, en particular, tablas de *hashing*.

Al interior del libro se utilizan las siguientes convenciones:

	Algoritmo implementado en el disquete
	Ejercicio medianamente complejo
	Ejercicio con alto grado de dificultad
	Ejercicio de programación
	Marca de final de ejemplo
times, sin tilde	Nombre de rutinas en el texto
negrilla	Definición de un nuevo concepto
itálica	Palabra en inglés que, por razones de claridad, no ha sido traducida al español

Quiero agradecer los comentarios sobre las versiones preliminares del libro, que recibí de parte de las siguientes personas: Mario Daniel Ramírez (Instituto Tecnológico de Costa Rica), Javier Cañas (Universidad Federico Santa María - Chile), Claudia Jiménez (Universidad de los Andes - Bogotá), Roberto Ojeda (Universidad Nacional de Colombia - Bogotá), Yadran Eterovic (Pontificia Universidad Católica de Chile), Silvia Takahashi (Universidad de los Andes - Bogotá), Diego Andrade (Pontificia Universidad Católica del Ecuador), Jorge Elías Morales (Fundación Universitaria San Martín - Bogotá), Carlos Figueira (Universidad Simón Bolivar - Venezuela), Ariel Ortiz Ramírez (Instituto Tecnológico de Monterrey - México), John A. Atkinson (Universidad Federico Santa María - Chile), Demetrio Ovalle (Universidad Nacional de Colombia - Medellín), Hernán Dario Toro (Universidad EAFIT - Medellín), Emilio Insfran (Universidad Católica de Asunción - Paraguay) y Gustavo Rossi (Universidad de La Plata - Argentina).

Por último, quiero agradecer de una manera muy especial a mi colega y amigo Alejandro Quintero, con quien comenzamos hace casi 9 años este proyecto, y sin cuyo apoyo habría sido imposible llevarlo a cabo.

J.V.

Enero de 1996

A Vicky,
por lo que ella significa para mí...

Contenido

CAPITULO 0 - CONCEPTOS BÁSICOS

CAPITULO 1 - DISEÑO DE *SOFTWARE* Y TIPOS ABSTRACTOS

CAPITULO 2 - ESTRUCTURAS LINEALES: LISTAS

CAPITULO 3 - ESTRUCTURAS LINEALES: PILAS Y COLAS

CAPITULO 4 - ESTRUCTURAS RECURSIVAS: ARBOLES BINARIOS

CAPITULO 5 - ESTRUCTURAS RECURSIVAS: ARBOLES N-ARIOS

CAPITULO 6 - ESTRUCTURAS NO LINEALES: GRAFOS DIRIGIDOS

CAPITULO 7 - ESTRUCTURAS DE ACCESO DIRECTO: TABLAS DE *HASHING*

CAPITULO 0

CONCEPTOS BÁSICOS

Este capítulo presenta algunos conceptos fundamentales sobre desarrollo y análisis de algoritmos. Se muestra una metodología de programación, basada en aserciones, utilizada a lo largo de todo el libro. Se estudia la manera de calcular la complejidad de un algoritmo, como una herramienta para comparar varias soluciones a un mismo problema, sin necesidad de implementarlas.

0.1. Diseño y Documentación de Algoritmos

Aunque suene contradictorio, programar es mucho más que escribir un programa. No es suficiente con escribir código en un lenguaje para resolver un problema y que éste funcione correctamente. El programa resultante debe ser también claro, eficiente y fácil de modificar. Eso implica una disciplina de programación y una metodología, que impongan un estilo de desarrollo que garantice la calidad del producto. Esto es más importante entre más grande sea el programa que se va a desarrollar, y se vuelve un factor crítico en grandes sistemas de información.

El objetivo de esta sección, más que mostrar a fondo la metodología de desarrollo formal de algoritmos, es ilustrar los elementos básicos que la componen, de tal manera que, aunque el estudiante no la utilice para programar, sea capaz de entender los elementos de especificación y documentación de un programa, que se usan a lo largo del libro.

Esta sección sólo supone que el estudiante tiene habilidad de programación en algún lenguaje, de preferencia C, y que maneja los fundamentos de las técnicas de solución de problemas. Para profundizar en cualquiera de esos temas se recomienda consultar la bibliografía que se sugiere al final del capítulo.

0.1.1. Los Conceptos de Estado y Aserción

Cuando se programa, se busca modelar a través de datos los elementos que intervienen en el problema, y describir los procesos, en términos de estos datos, para resolverlo. Un dato es un nombre que se le da a un valor de algún tipo para representar una característica del mundo. Al comenzar el programa, algunos datos tienen ya un valor (datos de entrada) y representan la situación exacta que se debe resolver. Al terminar, unos datos tienen los resultados esperados (datos de salida), que se interpretan de acuerdo al modelaje que se hizo del mundo.

Se define el **estado** de la ejecución de un programa, como el conjunto de los valores de todos sus datos. Eso quiere decir que el estado inicial lo determinan los datos de entrada y el estado final, los datos de salida.

Ejemplo 0.1:

Suponga que existe el problema de calcular el salario de un empleado de una empresa. Los datos de entrada contienen toda la información necesaria para resolver el problema exacto, como podrían ser el cargo, el sueldo básico, las primas extralegales, las horas extra trabajadas, la antigüedad, etc. Los datos de salida se pueden manejar como un solo dato: el salario.

El programa debe comenzar en un estado en el cual están definidas todas las características del empleado, y terminar en un estado en el cual el dato llamado salario contenga el sueldo que se le debe pagar. El proceso exacto de cálculo de este valor depende de la empresa, y se describe en un lenguaje de programación.

El estado del programa en cualquier etapa de la solución, está definido por los valores de los datos de entrada, los datos de salida y los datos temporales o de trabajo. En este caso, el estado estaría compuesto por 6 valores, asociados con 6 variables.

⏎

Una **aserción** es una afirmación que se hace en un punto de un programa sobre el estado vigente de la ejecución. Esta afirmación se refiere al valor de las variables del programa en ese instante y puede expresarse bajo cualquier formalismo (cálculo de predicados, un dibujo, etc.) o, incluso, en lenguaje natural.

Cuando el control del programa pasa por el sitio donde se encuentra una aserción, el estado de la ejecución debe ser tal que dicha afirmación sea verdadera.

Durante la especificación, una aserción se coloca entre los símbolos { }. En el programa, se coloca entre comentarios una breve descripción de la aserción en lenguaje natural. Una aserción está compuesta por un conjunto de afirmaciones separadas por coma. En algunos casos es conveniente asociarle un nombre, que se coloca al comienzo, para permitir referirse a ellas sin ambigüedad. Al igual que en el estándar definido en el lenguaje C, se utilizan minúsculas para variables y mayúsculas para constantes.

Ejemplo 0.2:

Si en un programa se encuentran declaradas 3 variables enteras var1, var2, var3 y una constante MAX, es posible tener las siguientes aserciones:

{ A1: var1 = var2, var3 > MAX }

{ var1 = 2 $\vee$ var1 $\geq$ MAX }

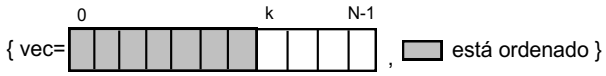
{ A3: TRUE }

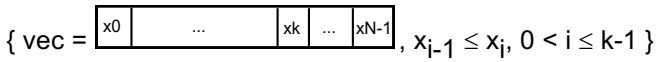
La aserción A1 expresa dos condiciones que debe cumplir el programa al pasar por el punto en el que ésta se encuentre situada, si el funcionamiento del programa es correcto. En este ejemplo, se usan los operadores relacionales igual (=) y mayor (>) para expresar la noción de validez. La segunda aserción utiliza el operador lógico *OR* ($\vee$), el cual hace que la aserción sea cierta si cualquiera de las dos condiciones se cumple. La tercera aserción siempre es cierta. Se coloca para expresar que con cualquier estado que pase por allí el programa, el funcionamiento es correcto.

⏎

Ejemplo 0.3:

Existen muchas formas distintas de expresar la misma aserción. Si en un punto de un programa se tiene un vector vec de tamaño N, ordenado en sus k primeras casillas, tres maneras posibles de construir una aserción que establezca este hecho son:



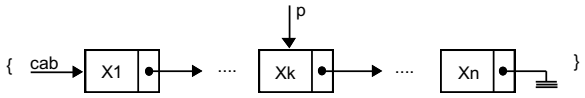


{ vec[0 → k-1] está ordenado, vec [k → N-1] no está ordenado }

⏎

Ejemplo 0.4:

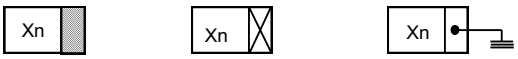
Para indicar que existe un apuntador p en la k-ésima posición de una lista encadenada apuntada por cab, una aserción posible es:



Para este tipo de estructuras, se va a suponer a través de los ejemplos del capítulo, que existe la siguiente declaración:

```
struct Nodo
{   int info;
    struct Nodo *sig;
};
```

Fíjese en el símbolo que se utiliza al final de la lista encadenada para representar el valor NULL. A lo largo del libro se utilizan indistintamente los siguientes símbolos para indicar este hecho:

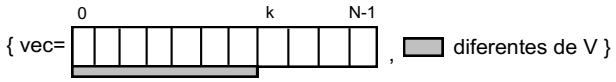


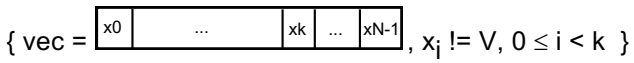
En la aserción del ejemplo, además de situar el apuntador p en la secuencia, se le da un nombre a cada uno de los elementos que la componen. La misma aserción podría referirse a x_k como el valor apuntado por p (v.g. $x_k > 0$).

⏎

Ejemplo 0.5:

El lenguaje gráfico puede ser de gran ayuda para expresar alguna condición sobre las estructuras de datos de un programa. En el momento de incluirlo, como parte del código del programa, se debe remplazar por lenguaje natural. Por ejemplo, con el fin de mostrar que los k primeros elementos de un vector vec son distintos a un valor V dado, se pueden construir las siguientes aserciones:





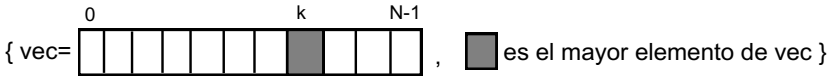
/* vec[i] != V, $0 \leq i < k \leq N-1$ */

/* Los k primeros elementos del vector vec son distintos de V */

⏎

Ejemplo 0.6:

Para afirmar que el mayor elemento de un vector vec se encuentra en la posición k, es posible utilizar cualquiera de las aserciones que se dan a continuación:



/* vec[k] ≥ vec[i], ∀i | 0 ≤ i < N */

/* vec[k] es el mayor elemento del vector vec */

Una aserción debe ser muy precisa con respecto a lo que pretende afirmar. En las aserciones anteriores, por ejemplo, no es claro en qué posición queda k si existen varios elementos iguales al mayor. Si este hecho es importante, se debe completar la aserción con una afirmación que elimine dicha ambigüedad, y diga, por ejemplo, que es la primera ocurrencia de dicho valor.

⏎

Ejemplo 0.7:

Para indicar que en un punto de un programa todos los elementos de una lista encadenada apuntada por cab son diferentes entre sí, se puede colocar la siguiente aserción:

/* cab $\rightarrow x_1 \rightarrow ... \rightarrow x_N$, x_k != x_i, ∀i, k | 0 ≤ i < N, 0 ≤ k < N, i != k */

⏎

0.1.2. La Especificación de un Programa

Un **programa** es una secuencia de instrucciones que hace que se transforme un estado inicial en un estado final, donde el problema se encuentra resuelto. Esto permite especificar un programa mediante dos aserciones: una, que describa el estado inicial (condiciones de los datos de entrada) y otra, el estado final (condiciones de los datos de salida). Estas dos aserciones asociadas con un programa se denominan respectivamente la **precondición** y la **postcondición**.

Básicamente, la precondición define las condiciones del estado inicial para que el programa pueda comenzar a resolver el problema. La postcondición, por su parte, establece las características de los datos de salida, en términos de los datos de entrada.

La notación:

{ pre: <aserción> } <programa> { post: <aserción> }

significa que si la ejecución del programa se inicia en un estado que satisface la precondición, se garantiza su terminación en un estado que satisface la postcondición.

Ejemplo 0.8:

Un programa que calcula el factorial de un número num y deja el resultado en una variable fact tiene la siguiente especificación:

{ pre: num ≥ 0 }

{ post: fact = num! }

Las aserciones establecen que si el valor de la variable num es no negativo, al ejecutar el programa se obtendrá el factorial de dicho valor en la variable fact. La postcondición no explica la manera de obtener un resultado, sino únicamente afirma la relación que debe existir al final del programa entre los datos de salida y los datos de entrada.

⏎

Ejemplo 0.9:

Es muy importante que la postcondición sea suficiente para describir el estado final de la ejecución. Para un programa que ordena un vector vec de N posiciones, sin elementos repetidos, dada la precondición:

{ pre: vec = [x_0 ... x_{N-1}], todos los x_i son diferentes }

No es suficiente la postcondición:

{ post: vec = [y_0 ... y_{N-1}], $y_i \leq y_{i+1}$, todos los y_i son diferentes }

Puesto que todo vector ordenado cumple la postcondición, sin importar el estado inicial. Note que la postcondición sólo exige que el vector resultante tenga sus elementos ordenados, pero no indica su relación con los valores iniciales. Lo correcto sería:

{ post: vec = [y_0 ... y_{N-1}], $y_i \leq y_{i+1}$, todos los y_i son diferentes, $\forall i \; \exists k \mid y_i = x_k$ }

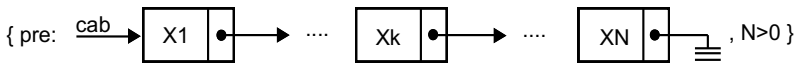
Incluso, es suficiente con una aserción como la siguiente:

/* vec ha sido ordenado ascendentemente */

⏎

Ejemplo 0.10:

Un programa que encuentre el mayor elemento de una lista encadenada apuntada por cab, se puede especificar mediante la siguiente pareja de aserciones:



{ post: mayor = x_k, $x_k \geq x_i \; \forall i \mid 1 \leq i \leq N$ }

Es importante que la precondición dé suficiente notación para referirse a los datos de entrada, de tal manera que sea posible expresar el valor de los datos de salida en función de éstos. En el ejemplo, no es suficiente con decir que cab apunta al comienzo de una lista sencillamente encadenada, sino que es necesario darle un nombre a cada uno de los elementos que la componen, para expresar, sin ambigüedad, el valor que tiene la variable mayor al final de la ejecución.

⏎

Ejemplo 0.11:

La postcondición de un programa puede considerar diferentes casos. Por ejemplo, calcular el valor absoluto de un valor num, dejándolo en esa misma variable, se puede especificar así:

{ pre: num = N }

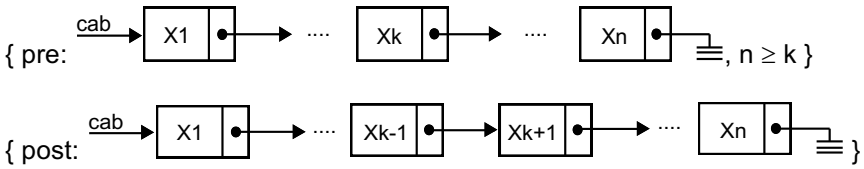
{ post: ($N \leq 0$, num = -N) $\vee$ (N > 0, num = N) }

Note la necesidad de utilizar en la precondición una constante adicional N para darle un nombre al valor inicial de la variable num, ya que la postcondición debe referirse a dicho valor. Esto se debe hacer siempre que un dato de entrada sea a la vez un dato de salida, y este valor final sea una función del valor inicial.

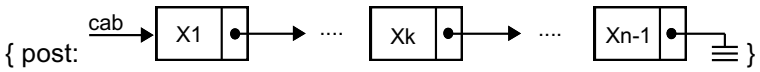
⏎

Ejemplo 0.12:

Una rutina que elimina el k-ésimo elemento de una lista encadenada se puede especificar con las siguientes aserciones:



En este caso x_k es un nombre para el valor que se encuentra inicialmente en la posición k de la lista. Es incorrecto, por tanto, colocar la siguiente postcondición:



Porque, aunque ahora sólo hay n-1 elementos, se deben respetar los nombres dados a los datos de entrada. En la última aserción se afirma que se ha eliminado el último elemento de los que aparecían en la lista inicial.

⏎

Ejercicios Propuestos:

Especifique formalmente los siguientes problemas:

0.1. Calcular la suma de todos los elementos de un vector vec.

0.2. Indicar si un valor num se encuentra presente en un vector vec.

0.3. Calcular el número de veces que aparece un valor num en un vector vec.

0.4. Calcular el elemento que aparece un mayor número de veces en una lista encadenada apuntada por cab.

0.5. Invertir una lista encadenada apuntada por cab.

0.6. Remplazar en un vector vec todas las ocurrencias del valor num1 por el valor num2.

0.7. Imprimir los elementos del vector vec.

0.8. Indicar si dos listas encadenadas tienen los mismos elementos, aunque sea en diferente orden.

0.9. Calcular el número de elementos de una lista encadenada apuntada por cab.

0.10. Invertir los elementos de un vector vec de N posiciones, intercambiando el primero con el último, el segundo con el penúltimo y así sucesivamente hasta terminar.

0.11. Informar la posición donde aparece por última vez un valor val en una lista encadenada apuntada por cab.

0.12. Informar si un valor val aparece entre los elementos de una lista encadenada apuntada por cab.

0.13. Retornar el elemento que se encuentra en la i-ésima posición de una lista encadenada apuntada por cab.

0.14. Adicionar un elemento val al final de una lista encadenada apuntada por cab.

0.15. 🕒 Calcular el número de valores diferentes que se encuentran en un vector vec.

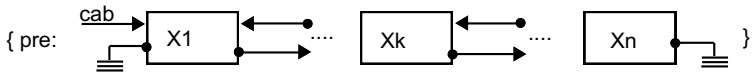
0.1.3. Dividir y Conquistar

Una de las técnicas más comunes de solución de problemas consiste en dividir el problema en dos o más subproblemas, más sencillos, de manera que la solución global sea la unión de las subsoluciones. Para esto, se colocan una o más aserciones intermedias entre la precondición y la postcondición, mostrando un estado en el cual una parte del problema ya ha sido resuelto.

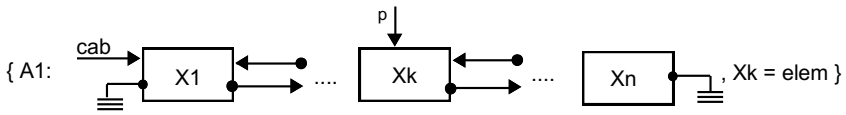
{ pre: <aserción> } <subproblema-1> { A1: <aserción>} <subproblema-2> { post: <aserción> }

Ejemplo 0.13:

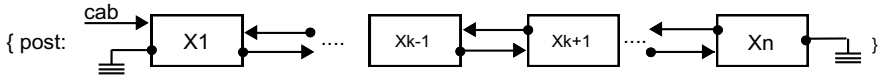
Para eliminar el valor elem presente en una lista doblemente encadenada apuntada por cab, se puede dividir el problema en dos partes: localizar el elemento y desencadenarlo.



<subproblema-1: localizar el elemento >



<subproblema-2: desencadenar el elemento >

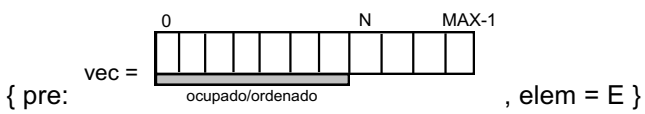


En este ejemplo, el problema de eliminar un elemento de una lista doblemente encadenada se reduce a dos problemas más sencillos: localizar un elemento en una lista doblemente encadenada y eliminar un nodo dado un apuntador a él. Cada subproblema se resuelve con un programa independiente, con su propia especificación, en la cual su precondición y/o su precondición corresponden a una aserción intermedia.

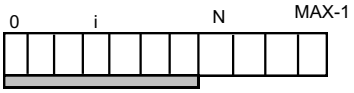
⏎

Ejemplo 0.14:

Agregar ordenadamente un valor elem a un vector vec de MAX posiciones cuyas N primeras están ocupadas. Un posible planteamiento es el siguiente, que incluye dos aserciones intermedias, y divide el problema en 3 subproblemas más sencillos:

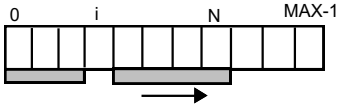


<subproblema-1: localizar el punto de inserción >

{ A1: vec = 

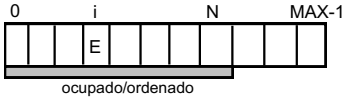
, i es la posición que debe ocupar E }

<subproblema-2: desplazar la información del vector >

{ A2: vec = 

}

<subproblema-3: colocar el nuevo valor >

{ post: vec = 

}

Para cada uno de los subproblemas encontrados, se busca una solución que puede consistir en volver a dividir el problema, en utilizar una de las técnicas que se muestran en las siguientes secciones o, si es posible hacerlo, en escribir una secuencia de asignaciones y llamadas de rutinas que lo resuelvan. Este último es el caso del subproblema-3, cuya solución es:

```
vec[ i ]= elem;
```

⏎

0.1.4. Consideración de Casos

Otra forma de resolver un problema es dividirlo en casos, cada uno con una solución más sencilla que el problema completo. De esta forma, un problema planteado con una precondición y una postcondición se resuelve con la siguiente estructura de algoritmo:

```
{ pre: <aserción> }

if (<condición>)
    { pre-caso-1: <aserción> }
    <subproblema -1>
else
    { pre-caso-2: <aserción> }
    <subproblema-2>

{ post: <aserción> }
```

Donde la <condición> define las características de la precondición que distinguen cada caso.

Ejemplo 0.15:

Calcular el valor absoluto de un número num, dejando el resultado en la variable abs. La estructura del algoritmo de solución es:

```
/* pre: TRUE */

if ( num >= 0 )
    /* pre1: num ≥ 0 */
    <subproblema-1: valor absoluto de un número no negativo>
else
    /* pre2: num < 0 */
    <subproblema-2: valor absoluto de un número negativo>

/* post: abs = | num | */
```

El problema global se divide en dos subproblemas, donde cada uno considera un caso distinto. La precondición de cada uno de ellos agrega condiciones a la precondición inicial, para distinguir la situación específica que debe tratar.

⏎

Ejemplo 0.16:

Calcular la longitud de una lista encadenada apuntada por cab. La estructura condicional para resolver este problema puede ser:

```
/* pre:  cab → [X1] → .... [Xk] → .... [XN] → ≡ , N ≥ 0 */

if ( cab == NULL )

    /* pre-1:  cab → ≡ , N = 0 */

    <subproblema-1: longitud de una lista vacía>

else
    /* pre-2:  cab → [X1] → .... [Xk] → .... [XN] → ≡ , N > 0 */

    <subproblema-2: longitud de una lista no vacía>

{ post: long = N }
```

⏎

0.1.5. Ciclos e Invariantes

La solución de un problema puede consistir en la repetición de un proceso, a través del cual se va transformando gradualmente el estado de la ejecución, para llevarlo de la precondición a la postcondición. La estructura del algoritmo, en ese caso, es:

```
{ pre: <aserción> }

{ inv: <invariante> }
while( <condición> )     <subproblema>

{ post: <aserción> }
```

La <condición> representa la situación en la cual el proceso iterativo ya ha llevado el estado del programa a cumplir la postcondición: cuando la <condición> es falsa, el problema ha sido resuelto. El <subproblema> es el proceso que se repite para avanzar en la solución del problema global. El <**invariante**> es una aserción que describe el ciclo. Se debe cumplir antes y después de cada iteración y su función es especificar el <subproblema>.

Ejemplo 0.17:

Calcular el factorial de un número no negativo num y dejar el resultado en la variable fact. La solución es:

```
/* pre: num ≥ 0 */

int i = 0;
int fact = 1;

/* inv: fact = i!, 0 ≤ i ≤ num */

while( i < num )
{   i++;
    fact *= i;
}
/* post: fact = num! */
```

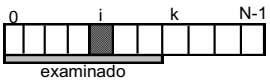
El invariante afirma que, después de cada iteración, en la variable fact se encuentra el valor i!. Esta es una variable de trabajo que cuenta el número de iteraciones, y varía entre 0 y num. En cada iteración, el código del ciclo debe garantizar que se sigue cumpliendo el invariante, aunque hayan avanzado las variables de trabajo. En este caso, se debe garantizar que al avanzar i, se modifique el valor de la variable fact.

El invariante se cumple antes de entrar la primera vez al ciclo y después de abandonarlo. Esa es la característica que garantiza la continuidad de la solución y, por lo tanto, su corrección. La condición de salida es (i == num), puesto que en ese instante se ha llegado a la postcondición del problema.

Ejemplo 0.18:

Encontrar el menor valor de un vector vec.

```
/* pre: vec es un vector de N posiciones */
int menor = vec[ 0 ];
```

/* inv: vec = 

, menor = vec[i], es el menor de la parte examinada */

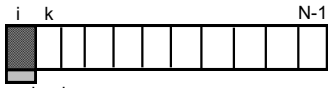
```
for( k = 1; k < N; k++ )
    if( menor > vec[ k ] )
        menor = vec[ k ];

/* post: menor contiene el menor elemento del vector vec */
```

El invariante afirma que para cualquier valor de la variable de trabajo k, que avanza de 1 hasta N, en la variable menor va a estar el mínimo elemento que existe en el vector en el rango de posiciones 0 a k-1. Cuando k = N, menor tiene la respuesta que se está buscando. En este caso, para conservar el invariante

después de cada ciclo, antes de avanzar la variable k es necesario constatar si se debe alterar el valor de la variable menor.

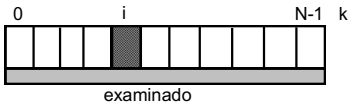
Es importante anotar que la variable i no se utiliza como parte del programa, sino únicamente como parte de la especificación, y se refiere a la posición del menor elemento dentro del vector hasta el lugar examinado por el proceso iterativo.

En la siguiente aserción se puede ver claramente que el invariante se cumple al entrar la primera vez al ciclo:

vec = 

{ A1: , menor = vec[0], es el menor elemento entre 0 y 0 }

Lo mismo ocurre al terminar:

vec = 

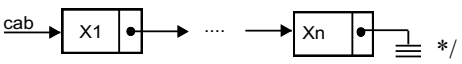
{ A2: , menor = vec[i], es el menor elemento entre 0 y N-1 }

Al escribir el invariante es importante verificar que estas dos situaciones límite se encuentren adecuadamente consideradas.

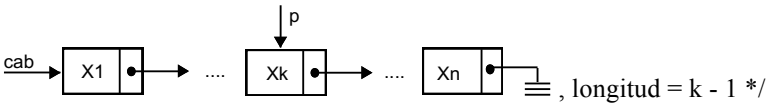
⏎

Ejemplo 0.19:

Una función para calcular y dejar en la variable longitud el número de elementos de una lista encadenada apuntada por cab:

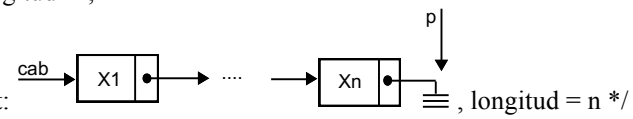
/* pre: 

*/

/* inv: 

, longitud = k - 1 */

```
for( p = cab, longitud = 0; p != NULL; p = p->sig )
    longitud++;
```

/* post: 

, longitud = n */

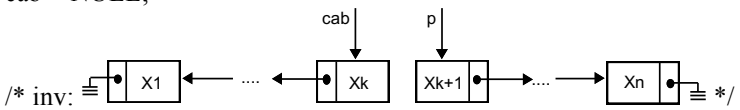
La función se limita a recorrer la estructura encadenada contando los elementos. El invariante relaciona la posición en la que va apuntando p con el valor de la variable longitud, afirmando que cuando p apunte al elemento x_k, el valor de la variable longitud debe ser k-1. Al final, cuando p haya llegado a NULL, la variable longitud tiene almacenada la longitud de la lista (n).

⏎

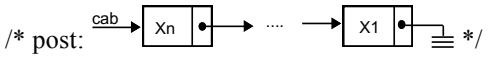
Ejemplo 0.20:

Invertir una lista encadenada, modificando el sentido de los apuntadores.

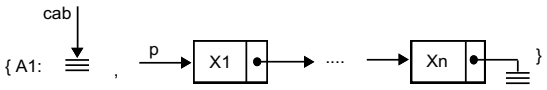

```
p = cab;
cab = NULL;
```



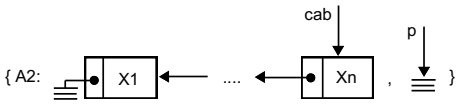

```
while( p != NULL )
{   q = p->sig;
    p->sig = cab;
    cab = p;
    p = q;
}
```



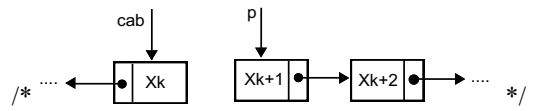
La idea de fondo es ir cambiando uno por uno el sentido de los apuntadores. Para esto se trabaja con dos listas: una, con los elementos que no han sido procesados (encabezada por p), y otra, con los elementos cuyo encadenamiento ya ha sido invertido (encabezada por cab). Al entrar la primera vez al ciclo se cumple un caso particular del invariante, en el cual todos los elementos están sin procesar:



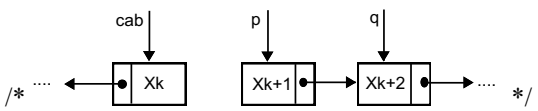
En el momento en el cual se terminan los elementos de la lista encabezada por p (elementos por procesar), ya no es interesante volver a entrar al proceso iterativo. Ese también es un caso particular del invariante:



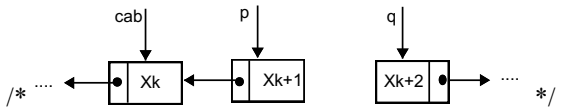
El cuerpo del ciclo debe resolver el problema de pasar de un estado que cumple el invariante, a otro que lo sigue cumpliendo, pero que se encuentra más cerca de la situación final esperada. En este ejemplo, debe resolver el problema de pasar un nodo de la lista de no procesados a la lista de procesados. La manera de hacerlo se ilustra con la siguiente secuencia de aserciones intermedias:



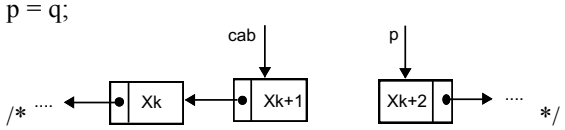

```
q = p->sig;
```



p->sig = cab;



cab = p;
p = q;



⏎

Ejercicios Propuestos:

Para los siguientes problemas, siga todo el proceso metodológico planteado para obtener y documentar la función que lo resuelve. Empiece por una especificación clara y, luego, utilice las técnicas de refinamiento antes enunciadas. Defina claramente el invariante de cada proceso iterativo y verifique que se cumpla antes de comezar el ciclo y después de terminar.

0.16. Invertir los elementos de un vector vec de N posiciones, intercambiando el primero con el último, el segundo con el penúltimo y así sucesivamente hasta terminar.

0.17. Sumar los elementos de un vector vec de N posiciones y retornar el resultado.

0.18. Determinar si dos vectores vec1 y vec2 de N posiciones son iguales.

0.19. Remplazar en un vector vec1 de N posiciones todas las ocurrencias del valor val1 por el valor val2.

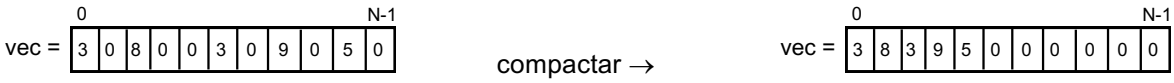
0.20. Imprimir todos los elementos de un vector vec de N posiciones, tales que son mayores que el valor val.

0.21. Agregar ordenadamente un valor elem a un vector vec de N posiciones, cuyas primeras M están ocupadas (ver ejemplo 0.14).

0.22. 🕒 Rotar M posiciones los elementos de un vector vec de N casillas. Rotar una posición significa desplazar todos los elementos una posición hacia la izquierda y pasar a la última posición el que antes se encontraba de primero.

0.23. 🕒 Decidir si dos vectores vec1 y vec2 de N posiciones tienen los mismos elementos, aunque estén en diferente orden.

0.24. 🕒 Compactar un vector vec de N posiciones. Esto significa que, al terminar, en la parte inicial del vector se deben encontrar todos los valores diferentes de cero. Por ejemplo:



0.25. Informar si un valor val aparece entre los elementos de una lista encadenada apuntada por cab.

0.26. 🖫 Calcular y retornar el elemento que aparece un mayor número de veces en una lista encadenada apuntada por cab.

0.27. Eliminar de una lista encadenada apuntada por cab los elementos que se encuentran en una posición par.

0.28. Retornar el elemento presente en la i-ésima posición de una lista encadenada apuntada por cab.

0.29. Eliminar de una lista encadenada apuntada por cab todas las apariciones de un elemento dado.

0.30. Informar la posición donde aparece por última vez un valor val en una lista encadenada apuntada por cab.

0.31. Invertir los elementos de una lista encadenada apuntada por cab, moviendo únicamente la información y no el encadenamiento.

0.32. Eliminar de una lista doblemente encadenada apuntada por cab todas las apariciones de un elemento.

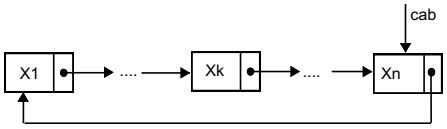
0.33. Invertir los elementos de una lista doblemente encadenada apuntada por cab, moviendo la información.

0.34. Rotar una posición una lista doblemente encadenada apuntada por cab. Esto implica pasar el primer elemento al final de la secuencia.

0.35. Leer del teclado una secuencia de valores enteros y construir una lista doblemente encadenada.

0.36. En un vector vec de N posiciones existe un elemento que no se encuentra ordenado con respecto a los demás. Localizar el elemento y colocarlo en el sitio adecuado.

Una lista circular tiene la siguiente estructura:



0.37. Calcular el número de elementos presentes en una lista circular apuntada por cab.

0.38. Adicionar un elemento val al final de una lista circular apuntada por cab.

0.39. Eliminar de una lista circular apuntada por cab todas las ocurrencias de un elemento val.

0.40. Concatenar dos listas circulares no vacías apuntadas por cab1 y cab2, dejando el resultado en la primera de ellas.

0.2. Recursión

En esta sección se presenta un breve resumen de una técnica muy utilizada en solución de problemas, denominada recursión. Este enfoque se utiliza con frecuencia en los capítulos de estructuras de datos recursivas, como árboles, en las cuales resulta mucho más sencillo plantear un algoritmo recursivo que uno iterativo. Para profundizar en el tema, se recomienda consultar la bibliografía dada al final del capítulo.

0.2.1. Conceptos Básicos

Un algoritmo es **recursivo** si se encuentra definido en términos de sí mismo. El ejemplo típico es el factorial de un número, que consiste en la multiplicación de dicho número por el factorial del número anterior (N! = N * (N- 1)!). Para indicar la forma de calcular el factorial de un número se utiliza la misma solución planteada, pero aplicada en otro valor. El código de dicha función recursiva es:

```
int factorial( int num )
{   if ( num == 0 )
        return 1;
    else
        return num * factorial( num - 1 );
}
```

En general, un algoritmo recursivo se plantea él mismo un problema con la misma estructura del inicial, pero de un tamaño menor. Luego, decide cómo extender esa solución para que incluya el problema completo.

Ejemplo 0.21:

Calcular la longitud de una lista encadenada. La función debe plantearse a sí misma un problema más sencillo, pero con la misma estructura. En este caso puede pedir que se calcule la longitud de la lista que comienza en la segunda posición. Tan pronto la función dé su respuesta, la manera de extender la solución al problema completo se reduce a sumarle 1 al valor obtenido.

```
/* pre: cab → x1 → x2 → ... → xn */
/* post: longLista = n */

int longLista( struct Nodo *cab )
{   if ( cab == NULL )
        return 0;
    else
        return 1 + longLista( cab->sig );
}
```

El planteamiento se puede resumir en el siguiente esquema recursivo:

longLista($x_1 \rightarrow x_2 \rightarrow ... \rightarrow x_n$) = longLista($x_2 \rightarrow ... \rightarrow x_n$) + 1

⏎

Esta técnica de solución de problemas tiene como ventajas la simplicidad de expresión (los algoritmos suelen ser muy sencillos) y lo natural que resulta utilizarlo en algunos problemas. Como desventajas están el espacio extra que ocupa en memoria por cada llamada de la rutina y el tiempo adicional de ejecución que todo esto implica.

Para dar una idea del funcionamiento de una rutina recursiva, la figura 0.1. muestra un resumen del proceso de ejecución de la función factorial, y de la manera como cada llamada va delegando la responsabilidad de resolver un problema más sencillo, para luego armar la respuesta a partir de la información que le retornan. fact(5) espera hasta que fact(4) se resuelva, lo cual sólo se lleva a cabo cuando fact(3) es calculado, y así sucesivamente hasta llegar a plantear el problema en términos de fact(0), que constituye un caso trivial.

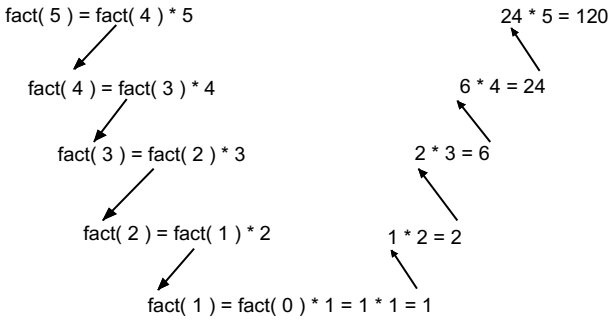


Fig. 0.1 - Ejecución de la función recursiva de factorial

La figura 0.2. muestra el proceso de ejecución de la rutina que calcula la longitud de una lista. De nuevo se va descomponiendo el problema en problemas más pequeños, a medida que avanza.

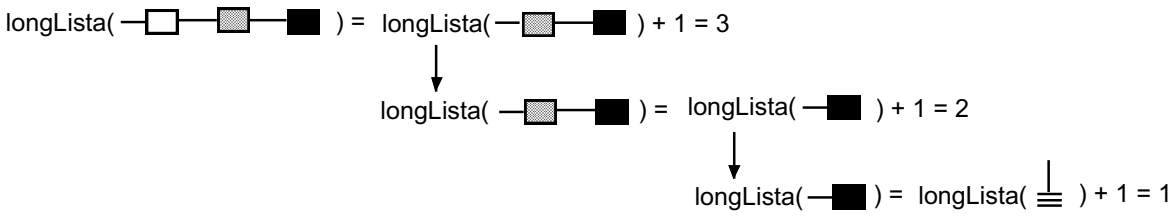


Fig. 0.2 - Ejecución de la función recursiva de longLista

0.2.2. Estructura de una Rutina Recursiva

En toda rutina recursiva se deben considerar dos conjuntos distintos de casos. Los primeros, son los llamados casos triviales, en los cuales se puede dar una respuesta sin necesidad de generar una llamada recursiva. Por ejemplo, para la función factorial, si el argumento vale 0, sin necesidad de una llamada recursiva se puede dar como respuesta el valor 1. En el caso de la función longLista, el caso trivial corresponde a la lista vacía, con respuesta 0. Estos casos se denominan las **salidas de la recursión**, y es necesario que exista por lo menos una de éstas dentro de cada rutina recursiva, para garantizar que el proceso termina.

Como segunda medida, se deben considerar los casos en los cuales, para poder dar una respuesta, es necesario esperar hasta obtener la respuesta de una llamada recursiva. Estos casos se conocen como los **avances de la recursión**, y también es necesaria la presencia de al menos uno de éstos en toda rutina recursiva.

La estructura general de una rutina recursiva, en la cual existen varias salidas y varios avances de la recursión, es la siguiente:

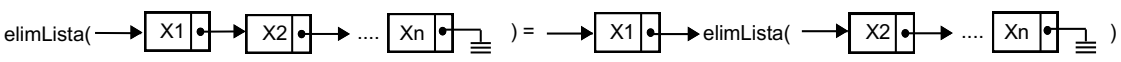
```
if (<condición-1>)
    <salida-1>
else if (<condición-2>)
    <salida-2>
else if (<condición-k>)
    <avance-1>
...
else <avance-n>
```

Ejemplo 0.22:

Eliminar un elemento de una lista encadenada ordenada ascendentemente. El planteamiento recursivo contempla tres salidas o casos triviales:

- La lista es vacía ⇒ la respuesta es la lista vacía
- El primer elemento de la lista es el que se quiere eliminar ⇒ la respuesta es la lista que comienza con el segundo elemento.
- El primer elemento de la lista es mayor que el que se quiere eliminar ⇒ la lista permanece sin modificación, puesto que el elemento no está presente.

Incluye un avance de la recursión, que se puede resumir en el siguiente esquema:



La rutina recursiva es la siguiente:

```
struct Nodo *elimLista( struct Nodo *cab, int elem )
{   if ( cab == NULL || cab->info > elem )          /* Salidas 1 y 3 */
        return cab;
    else if( cab->info == elem )                    /* Salida 2 */
    {   p = cab->sig;
        free( cab );
        return p;
    }
    else
    {   cab->sig = elimLista( cab->sig, elem );     /*Avance */
        return cab;
    }
}
```

Es importante resaltar que a partir del planteamiento recursivo, el código del programa se obtiene de manera natural.

⏎

Ejemplo 0.23:

Buscar un elemento en una lista encadenada. El planteamiento tiene 2 salidas de la recursión (la lista es vacía o el primer elemento es el buscado) y un avance (buscar desde el segundo elemento):

```
int estaLista( struct Nodo *cab, int valor )
{   if (cab == NULL )
        return 0;
    else if ( cab->info == valor )
        return 1;
    else
        return estaLista( cab->sig,valor );
}
```

⏎

0.2.3. Metodología de Desarrollo

Para el caso de una rutina recursiva, se debe abordar el desarrollo en tres etapas:

- Buscar las salidas de la recursión: en qué casos o bajo qué circunstancias se puede dar una respuesta inmediata al problema planteado.
- Determinar los avances de la recursión: en qué casos se debe plantear la respuesta en términos de una llamada recursiva. Para esto es conveniente: (1) identificar los casos posibles, (2) definir qué significa un problema con la misma estructura pero menor tamaño y plantear la llamada recursiva y (3) explicar en cada caso la manera de construir la respuesta a partir del retorno de dicha llamada.
- Escribir el algoritmo que implementa el planteamiento antes logrado.

Ejemplo 0.24:

Contar el número de veces que aparece un elemento en una lista encadenada.

- Prototipo de la función: int contar($x_1 \rightarrow ... \rightarrow x_n$, elem)

- Caso trivial: la lista es vacía ⇒ aparece 0 veces
- Avance-1:
  - El primer elemento de la lista es el que se busca (elem)
  - Llamada recursiva: contar($x_2 \rightarrow ... \rightarrow x_n$)
  - Construcción de la respuesta: 1 + contar($x_2 \rightarrow ... \rightarrow x_n$)
- Avance-2:
  - El primer elemento de la lista no es el que se busca (elem)
  - Llamada recursiva: contar($x_2 \rightarrow ... \rightarrow x_n$)
  - Construcción de la respuesta: contar($x_2 \rightarrow ... \rightarrow x_n$)

```
int contar( struct Nodo *cab )
{   if( cab == NULL )
        return 0;
    else if( cab->info == elem )
        return 1 + contar( cab->sig, elem );
    else
        return contar( cab->sig, elem );
}
```

Ejercicios Propuestos

Desarrolle rutinas recursivas para resolver cada uno de los siguientes problemas:

0.41. Imprimir los elementos de una lista encadenada.

0.42. Imprimir en orden contrario los elementos de una lista encadenada.

0.43. Determinar si dos vectores vec1 y vec2, de N posiciones, son iguales.

0.44. Calcular el número de ocurrencias del elemento elem en una lista encadenada.

0.45. Decidir si un elemento elem aparece en una lista encadenada ordenada ascendentemente.

0.46. Encontrar el menor valor de un vector.

0.47. Decidir si un elemento elem aparece en un vector.

0.48. Decidir si un vector se encuentra ordenado.

0.49. Calcular el número de valores diferentes que se encuentran en un vector.

0.50. Retornar la posición de la i-ésima ocurrencia de un elemento elem, en una lista encadenada.

0.51. Retornar el elemento que aparece un mayor número de veces en una lista encadenada.

0.52. Concatenar dos listas lst1 y lst2, dejando todos los elementos de lst2 al final de lst1.

0.53. Eliminar todas las ocurrencias del elemento elem en una lista encadenada.

0.54. Retornar el i-ésimo elemento de una lista encadenada.

0.55. Invertir una lista encadenada.

0.56. Ordenar ascendentemente un vector.

0.3. Análisis de Algoritmos

Una de las herramientas con que cuenta un ingeniero, para hacer la evaluación de un diseño, es el análisis de algoritmos. A través de éste, es posible establecer la calidad de un programa y compararlo con otros programas que se puedan escribir para resolver el mismo problema, sin necesidad de desarrollarlos. El análisis se basa en las características estructurales del algoritmo que respalda el programa y en la cantidad de memoria que éste utiliza para resolver un problema.

El análisis de algoritmos se utiliza también para evaluar el diseño de las estructuras de datos de un programa, midiendo la eficiencia con que los algoritmos del programa son capaces de resolver el problema planteado, si la información que se debe manipular se representa de una forma dada.

La presentación que se hace en esta sección del tema de análisis de algoritmos no es completa. Sólo se ven las bases para que el estudiante pueda evaluar los algoritmos que se presentan en el libro. Para una presentación más profunda se recomienda consultar la bibliografía que se sugiere al final del capítulo.

0.3.1. Definición del Problema

Suponga que existen dos programas P1 y P2 para resolver el mismo problema. Para decidir cuál de los dos es mejor, la solución más sencilla parece ser desarrollarlos y medir el tiempo que cada uno de ellos gasta para resolver el problema. Después, se podrían modificar los datos de entrada, de alguna manera preestablecida, y promediar al final su desempeño para establecer su comportamiento en el caso promedio.

La solución anterior tiene varios problemas. Primero, que pueden existir muchos algoritmos para resolver un mismo problema y resulta muy costoso, por no decir imposible, implementarlos todos para poder llevar a cabo la comparación. Segundo, modificar los datos de entrada para encontrar el tiempo promedio puede ser una labor sin sentido en muchos problemas, llevando a que la comparación pierda significado.

El objetivo del análisis de algoritmos es establecer una medida de la calidad de los algoritmos, que permita compararlos sin necesidad de implementarlos. Esto es, tratar de asociar con cada algoritmo una función matemática que mida su eficiencia, utilizando para este efecto únicamente las características estructurales del algoritmo. Así, se podrían llegar a comparar diversos algoritmos sin necesidad, siquiera, de tenerlos implementados. Como una extensión de esto, sería posible comparar diferentes estructuras de datos, tomando como factor de comparación el algoritmo más eficiente que se pueda escribir sobre ellas, para resolver un problema dado, tal como se sugiere en la figura 0.3.

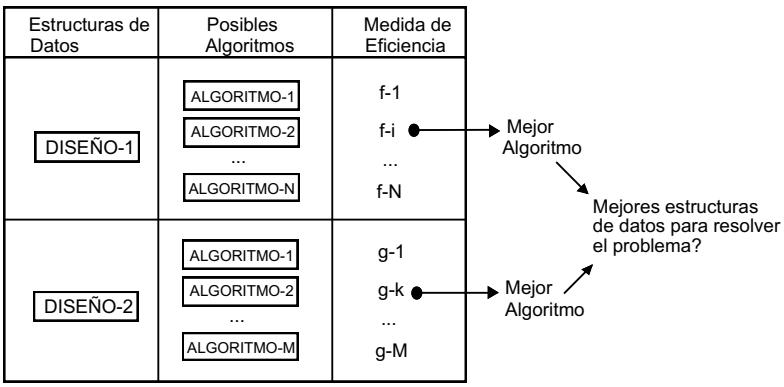


Fig. 0.3- Análisis de algoritmos como herramienta para el diseño de estructuras de datos

Adicional al tiempo de ejecución, existe otro factor que también se debe considerar al medir la eficiencia de un algoritmo: el espacio que ocupa en memoria. No tiene sentido escoger un algoritmo muy veloz, cuyas exigencias de memoria puedan impedir su uso en algunas situaciones. Esta es la segunda medida que se va a utilizar en el diseño de las estructuras de datos.

0.3.2. Tiempo de Ejecución de un Algoritmo

Para poder tener una medida del tiempo de ejecución de un programa, se debe pensar en los factores que tienen influencia en dicho valor. Inicialmente, se pueden citar los siguientes:

- La velocidad de operación del computador en el que se ejecuta. Es diferente ejecutar el programa en un micro 80386 que en un *Pentium* de 150 Mhz.
- El compilador utilizado (calidad del código generado). Cada compilador utiliza diferentes estrategias de optimización, siendo algunas más efectivas que otras.
- La estructura del algoritmo para resolver el problema.

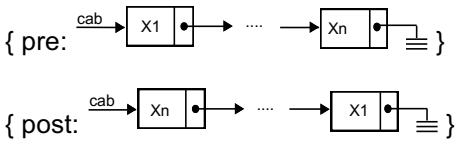
Con excepción del último, los factores mencionados no son inherentes a la solución, sino a su implementación, y por esta razón se pueden descartar durante el análisis.

Además de la estructura del algoritmo, se debe tener en cuenta que el número de datos con los cuales trabaja un programa también influye en su tiempo de ejecución. Por ejemplo, un programa para ordenar los elementos de un vector, se demora menos ordenando un vector de 100 posiciones que uno de 500. Eso significa que el tiempo de ejecución de un algoritmo debe medirse en función del **tamaño de los datos de entrada** que debe procesar. Esta medida se interpreta según el tipo de programa sobre el cual se esté trabajando.

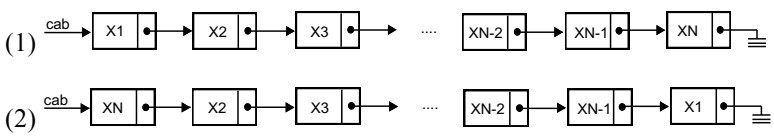
Se define $T_A(n)$ como el tiempo empleado por el algoritmo A en procesar una entrada de tamaño n y producir una solución al problema.

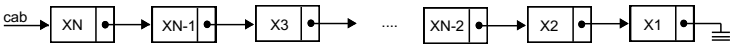
Ejemplo 0.25:

Considere dos rutinas que invierten una lista sencillamente encadenada de n elementos. Ambas cumplen la siguiente especificación:



El primero de los algoritmos en cuestión es el presentado en el ejemplo 0.20., que modifica los encadenamientos de la lista para invertirla. Hace solo una pasada sobre la estructura, haciendo en cada iteración el cambio de sentido de un apuntador. El segundo algoritmo es el que se desarrolla a continuación, que, para invertir la lista, mueve la información en lugar de alterar los encadenamientos. Para esto remplaza el primer elemento por el último, el segundo por el penúltimo, y así sucesivamente hasta terminar, como se muestra en la siguiente secuencia:



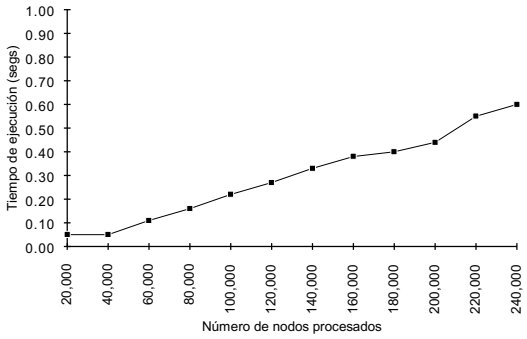
(3) 


```
struct nodo *inv2( struct nodo *cab )
{   int temp;
    struct nodo *p, *q, *r;
    for( q = cab; q->sig != NULL; q = q->sig );
    for( p = cab; p != q && q->sig != p; p = p->sig )
    {       temp = p->info;
            p->info = q->info;
            q->info = temp;
            for( r = p; r->sig != q; r = r->sig );
            q = r;
    }
    return cab;
}
```

El tamaño de los datos de entrada es el número de elementos de la lista encadenada. Los tiempos de ejecución de cada algoritmo se resumen en la siguiente tabla y dan una idea de la eficiencia de cada uno:

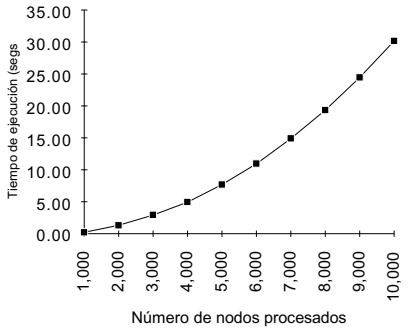
Algoritmo 1: tiempo de ejecución (tomado cada 20.000 nodos)

# nodos	T(n) segs
20,000	0.05
40,000	0.05
60,000	0.11
80,000	0.16
100,000	0.22
120,000	0.27
140,000	0.33
160,000	0.38
180,000	0.38
200,000	0.44
220,000	0.55
240,000	0.60



Algoritmo 2: tiempo de ejecución (tomado cada 1.000 nodos)

# Nodos	T(n) segs
1,000	0.22
2,000	1.32
3,000	2.91
4,000	4.95
5,000	7.69
6,000	10.93
7,000	14.89
8,000	19.34
9,000	24.45
10,000	30.16



Si se consideran los resultados obtenidos, lo que a primera vista podría parecer una pequeña ineficiencia del segundo algoritmo, consistente en la relocalización -en cada iteración- del apuntador q, lleva a desempeños muy diferentes: mientras el primer algoritmo alcanza a invertir 240.000 nodos en menos de 1 segundo, el segundo gasta más de 30 segundos en procesar sólo 10.000.

⏎

Lo ideal, al hacer la evaluación de la eficiencia de un algoritmo, sería encontrar una función matemática que describiera de manera exacta $T_A(n)$. Sin embargo, en muchos casos, el cálculo de esta función no se puede realizar, ya que depende de otro factor no considerado y que es, la mayoría de las veces, imposible de medir: el contenido o calidad de la entrada. Esto se ilustra en el siguiente ejemplo.

Ejemplo 0.26:

Considere el siguiente algoritmo, utilizado para decidir si el elemento elem se encuentra en un vector vec de N posiciones.

{ pre: vec = [X_0, ..., X_{N-1}] }

{ post: (X_i = elem, existe = TRUE) ∨ (X_k != elem, ∀k | 0 ≤ k < N, existe = FALSE) }

```
for( i = 0; i < N && vec[ i ] != elem; i++ );
existe = i < N;
```

Haciendo un análisis puramente teórico, se puede ver la influencia que tienen los datos específicos de la entrada (no sólamente su cantidad) en el tiempo de ejecución. Suponga que se fija el valor de N en 6 y que la evaluación de cada expresión del programa toma t microsegundos. Si los datos de entrada son:

vec = [5 6 7 8 9 10], elem = 5	i = 0;
	0 < 6 && vec[0] != 5
	existe = 0 < 6;
		t µseg
		t µseg
		t µseg

El algoritmo gasta en total 3t microsegundos, como se puede apreciar en el desarrollo anterior. Mientras que, para la siguiente entrada, el mismo algoritmo, con el mismo valor para N, toma 11t microsegundos.

vec = [5 6 7 8 9 10], elem = 9	i = 0;
	0 < 6 && vec[0] != 5; i++
	1 < 6 && vec[1] != 5; i++
	2 < 6 && vec[2] != 5; i++
	3 < 6 && vec[3] != 5; i++
	4 < 6 && vec[4] != 5;
	existe = 4 < 6;
		t µseg
		2t µseg
		2t µseg
		2t µseg
		2t µseg
		t µseg
		t µseg

Esto implica que, por más que se conozca el tamaño de los datos de entrada, es imposible -para muchos problemas- determinar el tiempo de ejecución para cada una de las posibles entradas.

⏎

Por la razón antes enunciada, se va a trabajar con el tiempo utilizado por el algoritmo en el peor de los casos, ya que es mucho más fácil definir cuál es el peor de los casos, que considerarlos todos, o incluso, que considerar el caso promedio. Se redefine la función de tiempo así:

$T_A(n)$ = tiempo que se demora el algoritmo A, en el peor de los casos, para encontrar una solución a un problema de tamaño n.

Así, se pueden comparar dos algoritmos cuando tratan de resolver el mismo problema en el peor de los casos. Para el ejemplo 0.26, el peor de los casos es cuando no encuentra el elemento en el vector, puesto que debe iterar N veces antes de darse cuenta de su inexistencia.

En algunos casos particulares de este libro, sobre todo en los capítulos 4 y 5, se utiliza el cálculo de la complejidad de un algoritmo en el caso promedio. Para eso se debe tener en cuenta la distribución probabilística de los datos que se manejan. Allí se ilustra este proceso.

0.3.3. El Concepto de Complejidad

La idea detrás del concepto de **complejidad** es tratar de encontrar una función f(n), fácil de calcular y conocida, que acote el crecimiento de la función de tiempo, para poder decir "$T_A(n)$ crece aproximadamente como f" o, más exactamente, "en ningún caso $T_A(n)$ se comporta peor que f al aumentar el tamaño del problema". En la figura 0.4. aparece la manera como crecen algunas de las funciones más utilizadas en el cálculo de la complejidad.

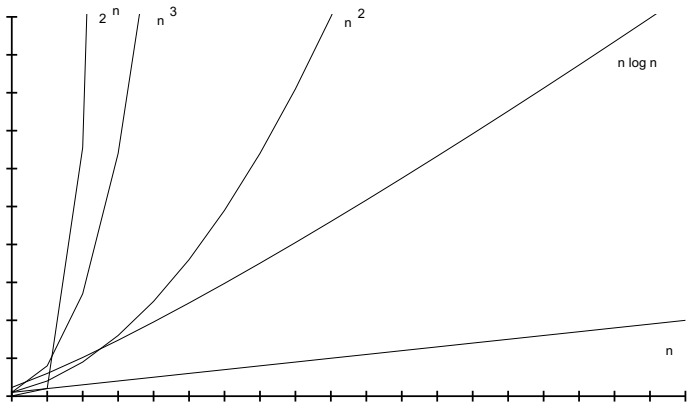


Fig. 0.4 - Crecimiento de las funciones típicas de complejidad de algoritmos

Al afirmar que un algoritmo es O(f (n)), se está diciendo que al aumentar el número de datos que debe procesar, el tiempo del algoritmo va a crecer como crece f en relación a n. En el ejemplo 0.25, una de las rutinas para invertir la lista es O(n), mientras que la otra es O(n^2), y esto se puede apreciar claramente en la forma de la gráfica de tiempos incluida en dicho ejemplo.

Ejemplo 0.27:

Suponga que se tienen 8 algoritmos distintos A1, ..., A8 para resolver un problema dado, cada uno con una complejidad diferente. Si a cada algoritmo le toma 1 microsegundo procesar 1 dato, en la siguiente tabla aparece el tamaño máximo del problema que puede resolver en una cierta unidad de tiempo [TAR91].

Allí se puede apreciar claramente cómo algunos algoritmos pueden resultar inaplicables para problemas de un cierto tamaño.

	Complejidad	1 seg	10^2 seg (1.7 min)	10^4 seg (2.7 horas)	10^6 seg (12 días)	10^8 seg (3 años)	10^{10} seg (3 siglos)
A1	$1000n$	10^3	10^5	10^7	10^9	10^{11}	10^{13}
A2	$1000n\log_2 n$	$1.4 * 10^2$	$7.7 * 10^3$	$5.2 * 10^5$	$3.9 * 10^7$	$3.1 * 10^9$	$2.6 * 10^{11}$
A3	$100n^2$	10^2	10^3	10^4	10^5	10^6	10^7
A4	$10n^3$	46	$2.1 * 10^2$	10^3	$4.6 * 10^3$	$2.1 * 10^4$	10^5
A5	$n \log 2n$	22	36	54	79	112	156
A6	$2^{n/3}$	59	79	99	119	139	159
A7	2^n	19	26	33	39	46	53
A8	3^n	12	16	20	25	29	33

Un problema se denomina **tratable** si existe un algoritmo de complejidad polinomial para resolverlo. En otro caso se denomina **intratable**. Esta clasificación es importante porque, cuando el tamaño del problema aumenta, los algoritmos de complejidad polinomial dejan de ser utilizables de manera gradual, como se puede apreciar en la figura 0.5. Por su parte, los algoritmos para resolver los problemas intratables explotan de un momento a otro, volviéndose completamente incapaces de llegar a una respuesta para el problema planteado. En la figura 0.5 se puede apreciar cómo un algoritmo de complejidad O(2^n) es capaz de resolver un problema de tamaño 20 en 1 segundo, pero ya es completamente inutilizable para problemas de tamaño 50, puesto que se demoraría 35 años buscando la solución.

El caso límite de los problemas intratables son los problemas **indecidibles**. Esos son problemas para los cuales no existe ningún algoritmo que los resuelva.

Complejidad	20	50	100	200	500	1000
$1000n$	0.02 seg	0.05 seg	0.1 seg	0.2 seg	0.5 seg	1 seg
$1000n\log_2 n$	0.09 seg	0.3 seg	0.6 seg	1.5 seg	4.5 seg	10 seg
$100n^2$	0.04 seg	0.25 seg	1 seg	4 seg	25 seg	2 min
$10n^3$	0.02 seg	1 seg	10 seg	1 min	21 min	2.7 horas
$n \log 2n$	0.4 seg	1.1 horas	220 días	125 siglos		
$2^{n/3}$	0.001 seg	0.1 seg	2.7 horas	$3*10^4$ siglos		
2^n	1 seg	35 años	$3*10^4$ siglos			
3^n	58 min	$2*10^9$ siglos				

Fig. 0.5 - Estimativos de tiempo para resolver un problema de tamaño N [TAR91]

En el momento de calcular la complejidad de un algoritmo, se debe encontrar la función que mejor se ajuste al crecimiento de $T_A(n)$, y no simplemente una cota cualquiera. En particular, todo algoritmo O(n) es a la vez O(n^2), y también es O(f(n)), para toda función f que crezca más rápido que f(n) = n.

Formalmente, se dice que:

$T_A(n)$ es O(f(n)) (la complejidad de A es f(n)) ssi $\exists$ c, $n_0 > 0$ | $\forall$ $n \geq n_0$, $T_A(n) \leq c$ f(n)

Esto implica que, para demostrar que un algoritmo tiene complejidad f(n), se debe buscar un punto n_0 sobre el eje del tamaño del problema, a partir del cual se pueda garantizar el acotamiento de $T_A(n)$ por la función f(n), ignorando los factores constantes de esta última (sólo interesa la forma de la función y no su valor exacto).

0.3.4. Aritmética en Notación O

Para facilitar el cálculo de la complejidad de un algoritmo es necesario desarrollar aritmética en **notación O**, de tal manera que sea posible dividir un algoritmo y, a partir del estudio de sus partes, establecer el cálculo global. Las siguientes demostraciones utilizan la definición formal de complejidad. Más importante que la demostración misma, es la interpretación intuitiva que se puede hacer de los resultados.

Teorema 0.1:

Si $T_A(n)$ es $O(k f(n)) \Rightarrow T_A(n)$ también es $O(f(n))$.

Este teorema expresa una de las bases del análisis de algoritmos: lo importante no es el valor exacto de la función que acota el tiempo, sino su forma. Esto permite eliminar todos los factores constantes de la función cota. Por ejemplo, un algoritmo que es $O(2n)$ también es $O(n)$, puesto que ambas funciones tienen la misma forma, aunque tienen diferente pendiente.

Demostración:

Si $T_A(n)$ es $O(k f(n)) \Rightarrow$ $\exists\ c, n_0 > 0 \mid \forall\ n \geq n_0\ ,\ T_A(n) \leq c.k.f(n)$

Al tomar $c_1 = c.k > 0$ se tiene que

$\exists\ c_1, n_0 > 0 \mid \forall\ n \geq n_0\ T_A(n) \leq c_1.f(n) \Rightarrow$

$T_A(n)$ es $O(f(n))$ ◆

Teorema 0.2:

Si A1 y A2 son algoritmos, tales que $T_{A1}(n)$ es $O(f1(n))$ y $T_{A2}(n)$ es $O(f2(n))$, el tiempo empleado en ejecutarse A1 seguido de A2 es $O(\max(f1(n), f2(n)))$.

Esto quiere decir que si se tienen dos bloques de código y se ejecuta uno después del otro, la complejidad del programa resultante es igual a la complejidad del bloque más costoso. Por esta razón, si hay una secuencia de comandos $O(1)$, también esta secuencia tendrá, en conjunto, complejidad constante. Pero si alguna de sus instrucciones es $O(n)$, todo el programa será $O(n)$.

Demostración:

Si $T_{A1}(n)$ es $O(f1(n)) \Rightarrow$ $\exists\ c_1, n_1 > 0 \mid \forall\ n \geq n_1, T_{A1}(n) \leq c_1.f_1(n)$

Si $T_{A2}(n)$ es $O(f2(n)) \Rightarrow$ $\exists\ c_2, n_2 > 0 \mid \forall\ n \geq n_2, T_{A2}(n) \leq c_2.f_2(n)$

$\Rightarrow T_A(n) = T_{A1}(n) + T_{A2}(n) \leq c_1.f_1(n) + c_2.f_2(n), \forall\ n \geq \max(n1, n2) \Rightarrow$

$\Rightarrow T_A(n) \leq\ (c_1 + c_2) * \max(f_1(n), f_2(n)), \forall\ n \geq \max(n1, n2) \Rightarrow$

Al tomar: $n_0 = \max(n_1, n_2) > 0$

$c_0 = (c_1 + c_2) > 0$, se tiene que:

$\exists\ c_o, n_0 > 0 \mid \forall\ n \geq n_0\ T_A(n) \leq c_o.\max(f_1(n), f_2(n))$

$\Rightarrow T_A(n)$ es $O(\max(f1(n), f2(n)))$ ◆

La demostración se basa en la idea de que la suma de las funciones de tiempo T1 y T2, acotadas por f1 y f2 respectivamente, se puede acotar con una función de la misma forma que la cota que crezca más rápido de las dos.

Teorema 0.3:

Sea A1 un algoritmo que se repite itera(n) veces dentro de un ciclo, tal que itera(n) es O(f2(n)) y T_{A1}(n) es O(f1(n)). El tiempo de ejecución del programa completo T_A(n) =T_{A1}(n) * itera(n) es O(f1(n) * f2(n)), suponiendo que el tiempo de evaluación de la condición se encuentra incluido en el tiempo de ejecución del algoritmo A1.

Este resultado permite que, al analizar un ciclo, se puedan estudiar primero el cuerpo y la condición, y finalmente acotar el número de iteraciones con una función conocida, de manera que la unión de estos resultados sea sencilla.

Demostración:

Si T_{A1}(n) es O(f1(n)) $\Rightarrow$ $\exists\ c_1, n_1 > 0 \mid \forall\ n \geq n_1\ T_{A1}(n) \leq c_1.f_1(n)$

Si itera(n) es O(f2(n)) $\Rightarrow$ $\exists\ c_2, n_2 > 0 \mid \forall\ n \geq n_2$ itera(n) $\leq c_2.f_2(n)$

$\Rightarrow T_A(n) = T_{A1}(n) * itera(n) \leq c_1.f_1(n) * c_2.f_2(n),\ \forall\ n \geq max(n1, n2) \Rightarrow$

$\Rightarrow T_A(n) \leq (c_1 * c_2) * f_1(n) * f_2(n) \Rightarrow$

Al tomar: $n_0 = max(n_1, n_2) > 0$

$c_0 = (c_1 * c_2) > 0$, se tiene que:

$\exists\ c_o, n_0 > 0 \mid \forall\ n \geq n_0\ T_A(n) \leq c_o.f_1(n).f_2(n)$

$\Rightarrow T_A(n)$ es O(f1(n) * f2(n)) ◆

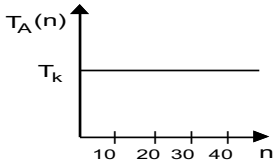
0.3.5. Ejemplos

En los siguientes ejemplos se ilustra la manera de calcular la complejidad de un algoritmo, utilizando los resultados obtenidos en la sección anterior. Los ejemplos van de lo elemental a lo complejo, y por esta razón es conveniente seguirlos en orden.

Ejemplo 0.28:

Calcular la complejidad de la asignación var = 5.

Este programa es O(1), porque una asignación que no tiene llamadas a funciones se ejecuta en un tiempo constante, sin depender del número de datos del problema. Si T_k es el tiempo que toma la asignación (expresado en cualquier unidad de tiempo), T_A(n) es O(T_k) puesto que se puede acotar con una función constante con ese mismo valor:



De acuerdo con el teorema 0.1., si T_A(n) es O(T_k) $\Rightarrow T_A$(n) también es O(1).

⏎

Ejemplo 0.29:

Calcular la complejidad del programa:

```
x = 1;
y = 2;
z = 3;
```

La complejidad de cada asignación es O(1), según se mostró en el ejemplo anterior. De acuerdo con el teorema 0.2., se puede concluir que $T_A(n)$ es O(max(1,1,1)), es decir O(1). Intuitivamente se puede establecer esta misma respuesta, al verificar que el número de asignaciones no depende del tamaño del problema que se quiere resolver.

⏎

Ejemplo 0.30:

Calcular la complejidad del programa x = abs(y), donde abs( ) es una función con el siguiente código:

```
float abs( float n )
{ if ( n < 0 )
        return -n;
  else
        return n;
}
```

Primero, se debe calcular la complejidad de la función, ya que la asignación va a tener la complejidad de la llamada.

El tiempo de ejecución de la instrucción if se puede acotar con el tiempo de evaluación de la condición más el tiempo de ejecución del subprograma más demorado de los dos asociados con la estructura condicional.

$T_{abs}(n) \leq T_{cond} + \max(T_{return}, T_{return}) \leq T_{cond} + T_{return}$

Ahora, la condición (n < 0) es O(1) porque toma un tiempo constante ejecutarla. Lo mismo sucede con la instrucción return. Entonces,

$T_{abs}(n)$ es O(max(1,1)) = O(1)

Esto hace que la asignación x = abs(y) sea a su vez O(1).

⏎

Ejemplo 0.31:

Calcular la complejidad del programa x = fact(n), si la función fact viene dada por el siguiente código:

```
int fact( int num )
{   int i, acum;
    i = 0;
    acum = 1;
    while ( i < num )
    {     i++;
          acum *= i;
    }
    return acum;
}
```

Para calcular la complejidad de un algoritmo se suele comenzar de arriba hacia abajo y de adentro hacia afuera (en el caso de ciclos).

El subprograma:

```
i = 0;
acum = 1;
```

es O(1) lo mismo que la instrucción:

```
return acum;
```

Para calcular la complejidad del ciclo se comienza por evaluar la complejidad del subprograma asociado, incluyendo la evaluación de la condición.

```
while ( i < num )
{   i++;
    acum *= i;
}
```

Tanto la comparación como las dos asignaciones son O(1). Ahora, se busca una función que acote el número de iteraciones del ciclo. En este caso, podemos escoger f(num) = num, puesto que éste nunca se va a ejecutar más de num veces.

Esto hace que:

T_{while}(num) sea O(num * 1) = O(num)

Y la complejidad de toda la función:

T_{fact}(num) es O(max(1, num, 1)) = O(num)

En todos los casos, la complejidad de una función debe quedar en términos de sus parámetros de entrada, puesto que son los que definen el tamaño del problema.

El programa `x = fact( n )` es O(n), porque ese es el costo de evaluar la parte derecha de la asignación.

⏎

Ejemplo 0.32:

Calcular la complejidad del siguiente programa:

```
for ( i = 0; i < 9; i++ )
    a[ i ] = 0;
```

La complejidad del ciclo es O(1), porque equivale a 9 asignaciones y siempre va a tomar un tiempo constante. Note que este programa se podría reescribir como:

```
a[ 0 ] = 0;
a[ 1 ] = 0;
a[ 2 ] = 0;
a[ 3 ] = 0;
a[ 4 ] = 0;
a[ 5 ] = 0;
a[ 6 ] = 0;
a[ 7 ] = 0;
a[ 8 ] = 0;
```

Y por lo visto en ejemplos anteriores es O(1).

⏎

Ejemplo 0.33:

Calcular la complejidad del siguiente procedimiento, que inicializa un vector de tamaño tam.

```
void inic( int a[ ], int tam )
{   int i;
    for ( i = 0; i < tam; i++ )
        a[ i ] = 0;
}
```

En este caso la rutina es O(tam), porque el tiempo de ejecución va a depender, de manera proporcional, del tamaño del vector. Es importante apreciar la diferencia entre este ejemplo y el anterior, que a primera vista pueden parecer semejantes. En este ejemplo el número de asignaciones no es fijo, y el tiempo que va a demorar en ejecutarse la rutina va a depender del tamaño del vector que se debe inicializar. Si el tamaño es N, se va a demorar T segundos, mientras que si es 2N se va a gastar 2T segundos.

⏎

Ejemplo 0.34:

Calcular la complejidad del siguiente programa, que suma dos matrices mat1 y mat2 de dimensión N*M y deja el resultado en una tercera matriz mat3 de las mismas dimensiones:

```
for ( i = 0; i < N; i++ )
    for ( k = 0; k < M; k++ )
        mat3[ i ][ k ] = mat1[ i ][ k ]+ mat2[ i ][ k ];
```

El ciclo interno es O(M), porque la asignación es O(1) y se repite M veces. Puesto que el ciclo exterior se repite N veces, el programa completo es O(N*M).

⏎

Ejemplo 0.35:

Establecer la complejidad de un procedimiento que calcula e imprime la longitud de una lista encadenada:

```
void impLongitud( struct Nodo* cab )
{   int cont;
    struct Nodo *p;
    cont = 0;
    for ( p = cab; p != NULL; p = p->sig )
        cont ++;
    printf( "%d", cont );
}
```

El tiempo de ejecución de este algoritmo es O(n), donde n es el número de elementos presentes en la lista encadenada. Note que, en este caso, la complejidad se da en función de un valor implícito en un parámetro de entrada. Para este cálculo, suponemos que todas las operaciones de entrada/salida se ejecutan en tiempo constante.

⏎

Ejemplo 0.36:

Calcular la complejidad de una rutina que ordena un vector de tamaño N. La rutina se encuentra apoyada por un función que retorna el menor elemento de un vector a partir de una posición dada:

```
void ordenar( int vec[  ] )
{   int i, temp, pos;
    for ( i = 0; i < N - 1; i++ )
    {       pos = posMenor( vec, i );
            temp = vec[ i ];
            vec[ i ] = vec[ pos ];
            vec[ pos ] = temp;
    }
}

int posMenor( int vec[  ], int desde )
{   int i, menor;
    menor = desde;
    for ( i = desde + 1; i < N; i++ )
            if ( vec[ i ] < vec[ menor ] )
                    menor = i;
    return menor;
}
```

$T_{posMenor}$ es O(N), porque en el peor de los casos el parámetro de entrada desde vale 0, y debe recorrer todo el vector buscando el menor elemento. $T_{ordenar}$, por su parte, es O(N^2), puesto que repite N veces la llamada de la otra función.

⏎

Ejemplo 0.37:

La **búsqueda binaria** es un proceso muy eficiente para localizar un elemento en un vector ordenado. En cada iteración, el algoritmo compara el valor que está buscando con el elemento que se encuentra en la mitad del vector, y, basado en si el elemento es menor o mayor, descarta la otra mitad de los valores, antes de continuar el proceso de búsqueda bajo el mismo esquema. El código de dicha rutina es el siguiente:

```
int busquedaBinaria( int vec[  ], int elem, int dim )
{   int desde = 0;
    int hasta = dim - 1;
    int mitad;
    while( desde <= hasta )
    {       if( vec[ mitad = ( desde + hasta + 1 ) / 2 ] == elem )
                    return TRUE;
            if( vec[ mitad ] > elem )
                    hasta = mitad - 1;
            else
                    desde = mitad + 1;
    }
    return FALSE;
}
```

Puesto que el cuerpo del ciclo es evidentemente O(1), el problema se reduce a encontrar una función que acote el número de iteraciones del ciclo. La primera posibilidad es utilizar la función f(dim) = dim (donde dim es el tamaño del vector), puesto que nunca va a entrar más de dim veces al ciclo. Pero, dado que en cada iteración se reduce a la mitad el tamaño del problema, es mejor, como cota del número de iteraciones, una función f(dim), que cumpla que $2^{f(\ dim\ )}$ = dim (v.g. 2 al número de iteraciones es igual al tamaño del vector). Despejando de allí la función, se obtiene que la complejidad de la búsqueda binaria es O($\log_2$dim).

Este algoritmo resulta tan eficiente, que encontrar un valor en un vector de 25.000 elementos requiere solamente 15 comparaciones.

⏎

0.3.6. Complejidad en Espacio

La misma idea que se utiliza para medir la complejidad en tiempo de un algoritmo se utiliza para medir su **complejidad en espacio**. Decir que un programa es O(N) en espacio significa que sus requerimientos de memoria aumentan proporcionalmente con el tamaño del problema. Esto es, si el problema se duplica, se necesita el doble de memoria. Del mismo modo, para un programa de complejidad O(N^2) en espacio, la cantidad de memoria que se necesita para almacenar los datos crece con el cuadrado del tamaño del problema: si el problema se duplica, se requiere cuatro veces más memoria. En general, el cálculo de la complejidad en espacio de un algoritmo es un proceso sencillo que se realiza mediante el estudio de las estructuras de datos y su relación con el tamaño del problema.

El problema de eficiencia de un programa se puede plantear como un compromiso entre el tiempo y el espacio utilizados. En general, al aumentar el espacio utilizado para almacenar la información, se puede conseguir un mejor desempeño, y, entre más compactas sean las estructuras de datos, menos veloces resultan los algoritmos. Lo mismo sucede con el tipo de estructura de datos que utilice un programa, puesto que cada una de ellas lleva implícitas unas limitaciones de eficiencia para sus operaciones básicas de administración. Por eso, la etapa de diseño es tan importante dentro del proceso de construcción de software, ya que va a determinar en muchos aspectos la calidad del producto obtenido.

0.3.7. Selección de un Algoritmo

La escogencia de un algoritmo para resolver un problema es un proceso en el que se deben tener en cuenta muchos factores, entre los cuales se pueden nombrar los siguientes:

- La complejidad en tiempo del algoritmo. Es una primera medida de la calidad de una rutina, y establece su comportamiento cuando el número de datos que debe procesar es muy grande. Es importante tenerla en cuenta, pero no es el único factor que se debe considerar.
- La complejidad en espacio del algoritmo. Es una medida de la cantidad de espacio que necesita la rutina para representar la información. Sólo cuando esta complejidad resulta razonable es posible utilizar este algoritmo con seguridad. Si las necesidades de memoria crecen desmesuradamente con respecto al tamaño del problema, el rango de utilidad del algoritmo es bajo y se debe descartar.
- La dificultad de implementar el algoritmo. En algunos casos el algoritmo óptimo puede resultar tan difícil de implementar, que no se justifique desarrollarlo para la aplicación que se le va a dar a la rutina. Si su uso es bajo o no es una operación crítica del programa que se está escribiendo, puede resultar mejor adoptar un algoritmo sencillo y fácil de implementar, aunque no sea el mejor de todos.
- El tamaño del problema que se va a resolver. Si se debe trabajar sobre un problema de tamaño pequeño (v.g. procesar 20 datos), da prácticamente lo mismo cualquier rutina y cualquier estructura de datos para representar la información. No vale la pena complicarse demasiado y es conveniente seleccionar el algoritmo más fácil de implementar o el que menos recursos utilice.
- El valor de la constante asociada con la función de complejidad. Si hay dos algoritmos A1 y A2 de complejidad O(f(n)), el estudio de la función cota debe hacerse de una manera más profunda y precisa en ambos casos, para tratar de establecer la que tenga una menor constante. Las diferencias en tiempo de ejecución de dos rutinas con la misma complejidad pueden ser muy grandes, como se muestra en la figura 0.6. Además, este es un factor que se puede ajustar en la implementación del algoritmo, lo cual hace que la calidad de la programación del algoritmo deba ser tenida en cuenta.

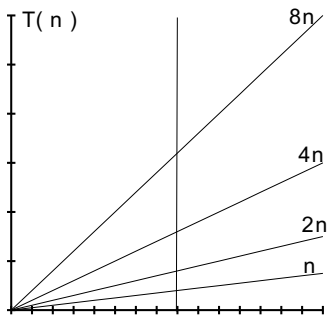


Fig. 0.6 - Funciones cota con diferentes constantes asociadas

- El rango de tamaños del problema en el cual debe trabajar eficientemente el algoritmo. Para cierto número de datos, un algoritmo de complejidad O(n^2) puede ser más eficiente que uno de complejidad O(n), o incluso que uno O(1), como se sugiere en la figura 0.7. Por eso se debe determinar el rango de datos para el cual se espera que el algoritmo sea eficiente.

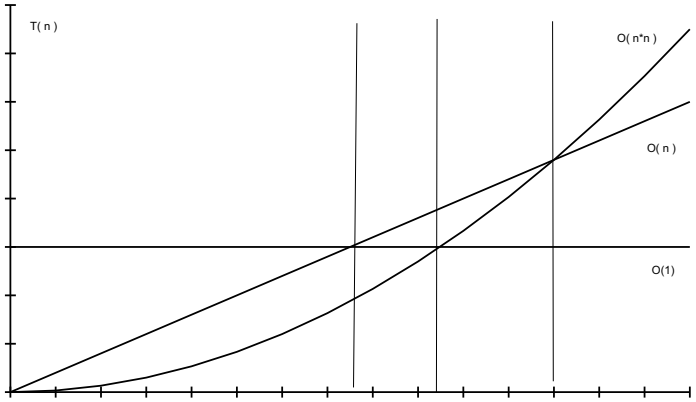


Fig. 0.7 - Comparación de varias funciones para valores pequeños de un problema

0.3.8. Complejidad de Rutinas Recursivas

Antes de comenzar esta sección, vale la pena advertir que el cálculo de la complejidad de una función recursiva puede resultar, en algunos casos, un problema matemático difícil de resolver. Para los problemas sencillos, como los presentados a través de ejemplos en esta parte, la solución matemática exacta es trivial. A lo largo del libro, cuando se haga el cálculo de la complejidad de una función recursiva cuya deducción no sea simple, se hará una presentación intuitiva del resultado, en lugar de una demostración formal.

Para las rutina iterativas, la solución planteada consistía básicamente en encontrar una función cota para el tiempo de ejecución T(n), mediante el estudio estructural del algoritmo. Ahora, el problema radica en que dicha función de tiempo se encuentra definida en términos de sí misma, y ya no es posible hacer una descomposición para estudiar el algoritmo, sino se hace necesaria la solución de una **ecuación de recurrencia**.

En los siguientes ejemplos se ilustra el proceso:

Ejemplo 0.38:

Calcular la complejidad de la función factorial:

```
int factorial( int num )
{   if ( num == 0 )
        return 1;
    else
        return num * factorial( num - 1 );
}
```

La función de tiempo de ejecución T(num), se puede plantear mediante la siguiente ecuación de recurrencia:

$$T(num) = \begin{cases} Tk, & num = 0 \\ Tk + T(num-1), & num > 0 \end{cases}$$

En ella aparece expresado que si num vale cero, la función toma un tiempo constante en calcular la respuesta. Si el parámetro num tiene un valor mayor que cero, el tiempo total viene definido como la suma del tiempo de calcular el factorial de num-1, más un tiempo constante, correspondiente a la multiplicación y al retorno de la respuesta.

La solución de dicha ecuación se puede hacer mediante la expansión simple de la recurrencia, como se muestra a continuación:

$T(num) = T_k + T(num-1)$

$= T_k + T_k + T(num-2)$

$= T_k + T_k + T_k + T(num-3)$

$= \ldots$

$= num * T_k + T(0)$

$= T_k * (num + 1)$

De allí se puede concluir que T(num) es O(num+1) $\Rightarrow$ T(num) es O(num).

⏎

Ejemplo 0.39:

Calcular la complejidad de una función recursiva que cuente el número de elementos que tienen en común dos listas encadenadas no ordenadas, sin elementos repetidos:

```
int num( struct Nodo *lst1, struct Nodo *lst2 )
{   if( lst1 == NULL )
        return 0;
    else if( esta( lst2, lst1->info )
        return 1 + num( lst1->sig, lst2 );
    else
        return num( lst1->sig, lst2 );
}
```

Esta función utiliza una segunda rutina recursiva que informa si un elemento se encuentra en una lista encadenada, cuyo código es:

```
int esta( struct Nodo *lst, int elem )
{   if( lst == NULL )
        return FALSE;
    else if( lst->info == elem )
        return TRUE;
    else
        return esta( lst->sig, elem );
}
```

Primero se calcula la complejidad de la segunda función. Para ésta, la ecuación de recurrencia es:

$$Testa(n) = \begin{cases} 1, & n = 0 \\ 1 + Testa(n-1), & n > 0 \end{cases}$$

Por las siguientes razones:

- El parámetro n corresponde al número de nodos de la lista, y define el tamaño del problema, de manera que la ecuación de recurrencia debe estar definida en términos suyos.
- El tiempo de ejecución que interesa medir es el del peor de los casos, y éste corresponde a la situación en la cual el elemento no aparece en la lista. Esto implica que sólo se utiliza la primera salida de la recursión.
- En lugar de la constante T_k se utiliza el valor 1, porque según se pudo apreciar en el ejemplo anterior, el valor de dicha constante es intrascendente para el resultado final.

La solución se obtiene por expansión simple de la recurrencia, como en el ejemplo anterior, y se llega a que $T_{esta}(n)$ es $O(n)$.

Para la función num, la ecuación de recurrencia para el tiempo de ejecución en el peor de los casos es:

$$Tnum(n1,n2) = \begin{cases} 1, & n1 = 0 \\ Testa(n2) + 1 + Tnum(n1-1,n2), & n1 > 0 \end{cases}$$

Donde n1 es el número de nodos de lst1 y n2 es el número de nodos de lst2. La solución de esta ecuación lleva a lo siguiente:

$T_{num}(n1, n2) = T_{esta}(n2) + 1 + T_{num}(n1-1, n2)$

$= T_{esta}(n2) + 1 + T_{esta}(n2) + 1 + T_{num}(n1-2, n2)$

$= \ldots$

$= n1 * (T_{esta}(n2) + 1) + T_{num}(0, n2)$

$= n1 * (T_{esta}(n2) + 1) + 1$

Puesto que $T_{esta}(n2)$ es $O(n2)$, se obtiene que $T_{num}(n1, n2)$ es $O(n1 * n2)$.

Ejemplo 0.40:

Calcular la complejidad de la implementación recursiva de la búsqueda binaria.

```
int busqueda( int vec[ ], int elem, int limiteInf, int limiteSup )
{   int medio;
    if ( limiteInf > limiteSup )
        return FALSE;
    else if ( vec[ medio = ( limiteInf + limiteSup + 1 ) / 2 ] == elem )
        return TRUE;
    else if ( elem < vec[ medio ] )
        return busqueda( vec, elem, limiteInf, medio-1 );
    else
        return busqueda( vec, elem, medio+1, limiteSup );
}
```

La ecuación de recurrencia resultante es:

$$T(n) = \begin{cases} 1, & n \le 1 \\ 1 + T(n/2), & n > 1 \end{cases}$$

Por las siguientes razones:

- El tamaño del problema corresponde al número de elementos entre las marcas de limiteInf y limiteSup. Dicho valor se denomina en este caso n. Cuando sólo queda un elemento en ese rango, o el rango sea vacío, utiliza la primera salida de la recursión.
- La segunda salida se puede ignorar, porque en el peor de los casos nunca la utiliza.
- Los dos avances de la recursión disminuyen a la mitad el tamaño del problema, y, sin importar cual de los dos utilice, va a gastar el mismo tiempo.

Al resolver la ecuación de recurrencia por simple expansión se obtiene:

$T(n) = 1 + T(n/2)$
$= 1 + 1 + T(n/4)$
$= 1 + 1 + 1 + T(n/8)$
$= \ldots$
$= \log_2 n * 1 + T(n/n)$ (suponiendo que n es potencia de 2)
$= \log_2 n + 1$

Por lo tanto, T(n) es O($\log_2 n$)

⏎

Ejercicios Propuestos

0.57. Desarrollar una rutina iterativa, de complejidad O(N), que lea una lista encadenada. N es el número de elementos leídos.

0.58. Calcular la complejidad de un programa que multiplique dos matrices cuadradas.

0.59. Calcular la complejidad del siguiente procedimiento, teniendo en cuenta que n es un entero positivo:

```
void proc( int n )
{       int i, k;
        i = 1;
        while ( i <= n )
        {       k = i;
                while ( k <= n )
                        k++;
                k = 1;
                while ( k <= i )
                        k++;
                i++;
        }
}
```

0.60. Calcular la complejidad del siguiente algoritmo, teniendo en cuenta que num es una potencia de 2 (v.g. 2,4,8,16,...):

```
void proc( int num )
{       int i;
        i = 2;
        while ( i < num )
                i *= 2;
}
```

0.61. Calcular la complejidad del siguiente algoritmo, sabiendo que val es un entero positivo:

```
void proc( int val )
{       int i, k, t;
        i = 1;
        while ( i <= val - 1)
        {       k = i + 1;
                while (k <= val)
                {       t = 1;
                        while (t <= k)
                                t++;
                        k++;
                }
                i++;
        }
}
```

0.62. Calcular la complejidad del siguiente procedimiento, sabiendo que n es un entero positivo:

```
void proc( int n )
{       int i = 1, k;
        while ( i <= n )
        {       k = n - i;
                while ( k >= 1)
                        k = k / 5;
                i++;
        }
}
```

0.63. Calcular la complejidad de los algoritmos que resuelven los ejercicios propuestos de todas las secciones anteriores.

0.64. Sea $P(n) = a_0 + a_1 n + a_2 n^2 + ... + a_m n^m$, un polinomio de grado m. Demostrar que si un algoritmo A tiene complejidad $O(P(n))$, entonces también es $O(n^m)$.

0.65. Implementar un algoritmo de multiplicación de matrices. Graficar la curva de tiempo de ejecución para matrices de diferentes tamaños. Comparar los resultados obtenidos con los teóricos.

0.66. Implementar el algoritmo de búsqueda binaria, de manera recursiva e iterativa. Construir una gráfica de tiempos de ejecución para el peor de los casos, en la cual se pueda apreciar la complejidad logarítmica. Utilizar esta gráfica para calcular el sobrecosto que tiene en términos de la constante, una rutina recursiva sobre una rutina iterativa.

0.67. Considere el siguiente problema: rotar k posiciones los elementos de un vector de N casillas. Rotar una posición significa desplazar todos los elementos una posición hacia la izquierda y pasar a la última posición el que antes se encontraba de primero. Desarrolle dos rutinas que lo resuelvan, de manera que una tenga complejidad O(N) y la otra O(N*k). Impleméntelas y grafique el tiempo de ejecución a medida que crecen N y k.

Bibliografía

Algoritmos: Metodología de desarrollo

- [CAR91] Cardoso, R., "Verificación y Desarrollo de Programas", Ediciones Uniandes, 1991.
- [DAL86] Dale, N., Lilly, S., "Pascal y Estructura de Datos", McGraw-Hill, 1986.
- [DIJ76] Dijkstra, E. W., "A Discipline of Programming", Prentice-Hall, 1976.
- [DRO82] Dromey, R.G., "How to Solve it by Computer", Prentice Hall, 1982.
- [GRI81] Gries, D., "The Science of Programming", Springer-Verlag, 1981.

Recursión:

- [FEL88] Feldman, M., "Data Structures with Modula-2", Prentice-Hall, 1988..
- [KRU87] Kruse, R., "Data Structures & Program Design", 1987.
- [MAR86] Martin, J., "Data Types and Data Structures", Prentice-Hall, 1986.
- [ROB86] Roberts, E., "Thinking Recursively", John Wiley & Sons, 1986.
- [WIR86] Wirth, N., "Algorithms & Data Structures", Prentice-Hall, 1986.

Análisis de Algoritmos:

- [AHO74] Aho, A., Hopcroft, J., Ullman, J., "The Design and Analysis of Computer Algorithms", Addison-Wesley, 1974.
- [AHO83] Aho, A., Hopcroft, J., Ullman, J., "Data Structures and Algorithms", Addison-Wesley, 1983.
- [FEL88] Feldman, M., "Data Structures with Modula-2", Prentice-Hall, 1988..
- [LIP87] Lipschutz, S., "Estructura de Datos", McGraw-Hill, 1987.
- [TAR91] Tarjan, R., "Data Structures and Network Algorithms", Society for Industrial and Applied Mathematics, 1991.

CAPITULO 1

DISEÑO DE *SOFTWARE* Y TIPOS ABSTRACTOS

El objetivo de este capítulo es enmarcar el diseño de estructuras de datos dentro del proceso completo de producción de *software*. Para esto se muestra una metodología de diseño a varios niveles: el primero, a nivel de Tipos Abstractos (TAD), en el cual se define la arquitectura global de un programa a partir del enunciado del problema, y el segundo, a nivel de diseño e implementación de estructuras de datos al interior de cada TAD. La metodología presentada incluye el análisis de algoritmos, visto en el capítulo anterior, como uno de los mecanismos de evaluación de diseños.

1.1. Ingeniería de *Software*

El propósito de la **ingeniería de *software*** es permitir al diseñador enfrentar el problema de construcción de *software* como un problema de ingeniería, con guías y principios concretos, al igual que con herramientas de evaluación y validación. Últimamente, se le ha dado especial importancia al estudio de este tema, dados los enormes costos de desarrollo de los sistemas informáticos y la forma vertiginosa como se multiplica su demanda.

1.1.1. Ciclo de Vida del *Software*

El **ciclo de vida** del *software* se suele representar mediante un **modelo de cascada** (Figura 1.1), en el cual, en cada etapa, se cumplen unos ciertos objetivos, para lograr obtener como producto un programa que satisfaga los requerimientos del usuario y cumpla con ciertos estándares de calidad.

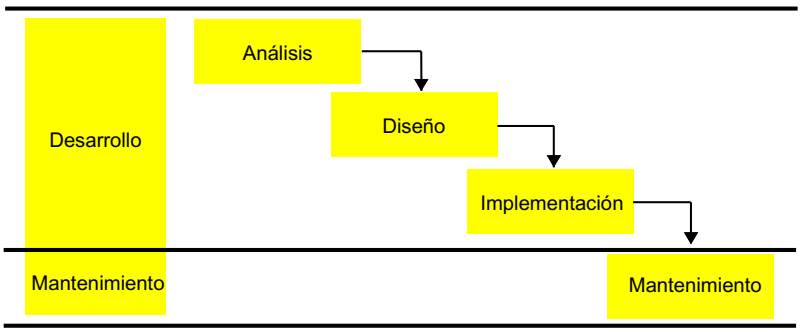


Fig. 1.1 - Ciclo de vida del *software*

El ciclo de vida de un programa se divide en dos partes: el **desarrollo**, que cubre las etapas de análisis, diseño e implementación, y el **mantenimiento**, en la cual se modifica el *software* para que continúe satisfaciendo los requerimientos del usuario durante la vida útil de una aplicación.

Las tres etapas de la fase de desarrollo, antes mencionadas, tienen el siguiente propósito:

- Análisis: el ingeniero debe entender a fondo el problema al cual se enfrenta, conseguir la información pertinente al mundo en el cual ocurre el problema, estructurar esta información, validarla y crear a partir de ella un modelo que refleje todo este conocimiento.
- Diseño: en esta etapa se estructura el programa que va a resolver el problema y se toman las decisiones de cómo representar la información, cómo dividir los procesos, cómo comunicarlos, etc.
- Implementación: en esta última etapa, se parte del diseño detallado del *software* y se escribe el programa correspondiente, expresando todos los elementos del diseño en un lenguaje de programación.

Al interior del ciclo de vida, la relación de costos entre las etapas de desarrollo y mantenimiento suele ser del orden de 30% - 70% (Figura 1.2), lo cual da una idea de por qué puede resultar tan costoso para un departamento de sistemas mantener en funcionamiento el *software* de una organización.

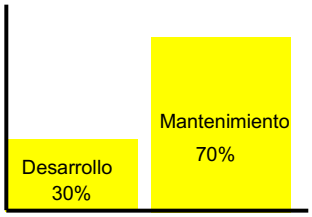


Fig. 1.2 - Costos comparativos de desarrollo y mantenimiento

1.1.2. *Software* de Alta Calidad

La **calidad del *software*** se mide a través de algunos factores que se pueden dividir en externos e internos. Los externos son los que ve el usuario final del *software*, como son la eficiencia, la corrección, la facilidad de uso, etc., indispensables en cualquier solución. Los factores internos de calidad son aquellos que vienen dados por la estructura misma del código, y sólo los ve el ingeniero de *software*. Entre otros, se pueden mencionar la documentación, la claridad del código, la modularidad, etc. Pero, considerando los grandes problemas y costos asociados con el mantenimiento, se puede afirmar que el principal factor de calidad interna de un programa es la facilidad que éste tiene para evolucionar, lo cual está fuertemente relacionado con su arquitectura interna. A esta característica del *software* se le denomina **extensibilidad**.

Los cambios en el *software* se pueden clasificar en tres grupos:

- Correcciones al programa: estas modificaciones son causadas por errores en el desarrollo, tanto a nivel de diseño como de implementación.
- Cambios en los requerimientos del usuario: a medida que la organización para la cual fue escrito el *software* evoluciona, las necesidades de los usuarios hacia los sistemas informáticos de apoyo van cambiando, y eso se debe reflejar en las posibilidades que ofrecen las aplicaciones.
- Evolución del mundo del problema: estas modificaciones responden a cambios en los elementos del mundo que participan en el problema, en sus relaciones, en las reglas de validez, etc. Son cambios muy frecuentes, con serias repercusiones sobre la estructura del *software*.

La misma vida útil de un programa está relacionada con su estructura y su extensibilidad. Si se define la **entropía** del *software* como el nivel de desorden interno que éste tiene, y se acepta que todo mantenimiento tiende a aumentarlo, en la figura 1.3 se ilustra cómo la vida útil de un programa está limitada por un umbral de entropía por encima del cual no es rentable seguir manteniendo el *software*, y que éste se alcanza más o menos cerca en el tiempo dependiendo de la entropía inicial.

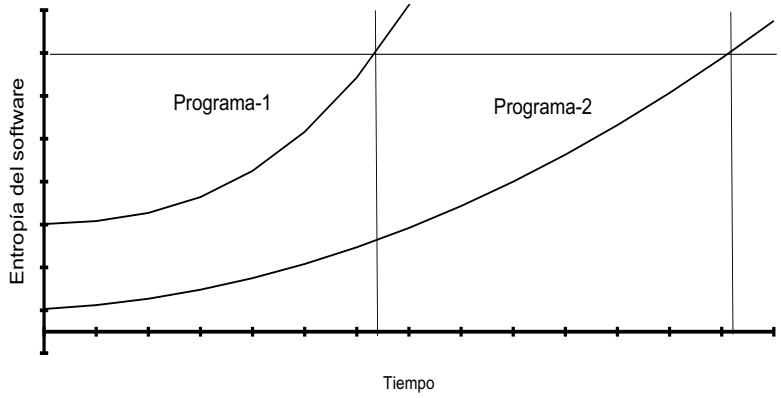


Fig. 1.3 - Vida útil del *software* en función de la estructura inicial

Por todas estas razones, las metodologías de diseño de *software* no se deben contentar con garantizar corrección o eficiencia en su producto, sino que, además de esto, deben generar un programa con una arquitectura interna que garantice una fácil evolución.

1.1.3. Arquitectura del *Software*

Una arquitectura que facilite la extensibilidad debe separar desde un comienzo toda la parte de requerimientos del usuario (interfaz), de la solución misma del problema (algorítmica de la aplicación), como se muestra en la figura 1.4.

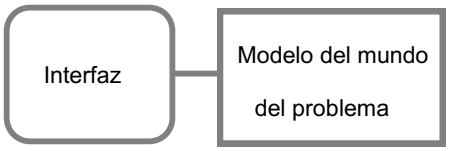


Fig. 1.4 - Separación entre la interfaz y el resto de la aplicación

La **interfaz** de un programa es la parte encargada de mantener el diálogo con el usuario, mientras va haciendo las llamadas adecuadas a las rutinas de la aplicación. Existen interfaces gráficas y alfanuméricas, según la manera como el usuario expresa sus comandos y como la aplicación le presenta la situación actual del problema que está resolviendo. Lo ideal es que el usuario pueda trabajar sobre elementos gráficos de interacción que pueda relacionar fácilmente con elementos de su mundo.

En el segundo módulo de la aplicación, denominado el **modelo del mundo**, está contenida toda la algorítmica del programa. La metodología de diseño de *software*, planteada en este capítulo, se restringe al diseño de esta parte de la aplicación, olvidando todo lo referente a la interfaz.

Para garantizar la extensibilidad del *software*, éste debe estar estructurado de tal manera que resulte sencillo localizar los puntos del programa que son afectados por un cambio en el mundo del problema. Además, debe asegurar que cualquier modificación que se haga tenga mínimas repercusiones sobre el resto del *software*. Esto hace que el diseño sea un proceso que exige una gran disciplina de parte del ingeniero, y en el cual, cualquier error que se cometa, comprometerá seriamente la calidad del producto obtenido. En el programa deben quedar situados y relacionados todos sus componentes de manera adecuada y sólida: diseñar es una labor de ingeniería y no un proceso puramente intuitivo.

Pensando en esto, las metodologías modernas de diseño de *software* incluyen nuevos elementos estructuradores, diferentes de los datos y los procesos, que van a convertirse en las unidades de diseño y mantenimiento de las aplicaciones. Cada unidad de estas se denomina un **Tipo Abstracto de Dato** (TAD). Así, la concepción de las estructuras de datos y el papel que juegan en la metodología de diseño se modifica radicalmente. Si se pretendiera reescribir la clásica ecuación de Wirth:

Programa = Datos + Algoritmos

Con este nuevo enfoque, se llegaría a algo del estilo:

Programa = { TAD } + Interfaz.

En la sección 1.2 se hace una presentación formal del concepto de tipo abstracto. Por ahora, solo se enuncian algunas ideas sobre la forma como un conjunto de TAD van a conformar el modelo del mundo:

- Un TAD es un ente cerrado y autosuficiente, que no requiere de un contexto específico para que pueda ser utilizado en un programa. Esto garantiza portabilidad y reutilización del *software*, y minimiza los efectos de borde que puede producir un cambio al interior de un TAD. Esta propiedad se denomina **encapsulamiento**.
- Cada elemento del mundo que participa en el problema va a tener un representante dentro del *software* que simule su operación. Cada uno de estos trozos de *software* se va a denominar un Tipo Abstracto de Dato, e, internamente, corresponde a una composición de datos y rutinas.

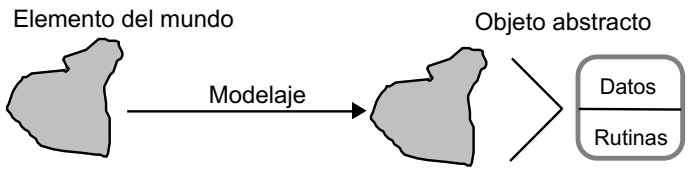


Fig. 1.5 - Relación elemento del mundo - objeto abstracto

- En el fondo, un programa va a estar compuesto por un conjunto de TAD, cada uno representando un tipo de elemento del mundo, estructurados y relacionados igual a como están estructurados y relacionados estos mismos elementos en el mundo del problema.
- Si el mundo cambia y el *software* debe evolucionar, la localización dentro del *software* del lugar que debe ser modificado resulta trivial, y el tamaño de la modificación, proporcional al cambio en el mundo. Se evitan efectos de borde indeseables y dependencias entre partes del programa.

Ejemplo 1.0:

Suponga que se va a desarrollar un programa para administrar la información de una biblioteca.

Puesto que allí hay elementos como ficheros, usuarios, libros, etc., que participan en el problema, en el *software* existirá un TAD que represente y simule la operación de cada uno de ellos: el TAD Fichero, el TAD Usuario y el TAD Libro. Estos TAD estarán relacionados dentro del programa, de la misma manera como los elementos que modelan están relacionados en la biblioteca: un elemento del TAD Usuario puede tener en préstamo un elemento del TAD Libro, los elementos del TAD Fichero tienen elementos del TAD Ficha, que representan libros de la biblioteca, etc.

No existirá un TAD Pared, puesto que no participa en el problema, así haga parte de la biblioteca (a menos, claro está, que se trate de un sistema de diseño arquitectónico, en el cual las paredes sean los elementos de base.

⏎

1.1.4. Reutilización de *Software*: Genericidad

Además de la facilidad de mantenimiento, es importante que la metodología simplifique el proceso de creación de *software*. Uno de los principales mecanismos con que se cuenta para esto es la reutilización de *software* mediante la **genericidad**, que se basa en patrones de *software* aplicables a distintos tipos de dato. Por ejemplo, el código de una rutina es igual para ordenar un vector de enteros, que para ordenar un vector de elementos gráficos, de manera que con un solo patrón de rutina es posible resolver ambos problemas.

Un **TAD genérico** es un patrón de Tipo Abstracto en el cual se parametrizan algunas de sus características, de tal forma que todos los elementos del mundo que se puedan modelar como variantes del mismo TAD, solo requieran un TAD de soporte, con el consecuente ahorro en diseño e implementación. El caso más común de TAD genérico es el **TAD contenedor**, cuya misión es agrupar y relacionar elementos de otros tipos. En esos casos, es posible definir un TAD contenedor en abstracto, sin necesidad de comprometerse con el tipo de los elementos que va a manejar. Por ejemplo, es posible diseñar el TAD Lista sin necesidad de restringir el tipo de los elementos que almacena, colocando dicho tipo como parámetro del tipo abstracto.

Esta forma de reutilización tiene múltiples ventajas. Es posible adquirir soluciones genéricas implementadas (existen grandes bibliotecas de TAD de soporte), y utilizarlas en cualquier problema que lo requiera. Esto ha demostrado que disminuye considerablemente el tiempo de desarrollo.

Los TAD genéricos y las rutinas genéricas tienen una implementación sencilla en lenguajes como C++ (*templates*), pero, en lenguajes como C, es necesario simularlos mediante algún mecanismo, que no siempre resulta sencillo.

1.2. Tipos Abstractos de Datos

A partir del contexto dado en la sección anterior, en esta parte se formaliza la noción de Tipo Abstracto (TAD) y se da notación para expresar un diseño.

1.2.1. Motivación y Definiciones

Informalmente, se puede decir que un TAD es un tipo de dato, que se agrega al lenguaje de programación, para representar un tipo de elemento involucrado en el problema que se quiere resolver. De esta forma se hace que el lenguaje se acerque al mundo del problema, manejando los elementos que allí se encuentran. Por ejemplo, si se va a desarrollar *software* para la administración de notas de una universidad, los TAD Curso, Estudiante, Nota, Lista, etc., van a permitir expresar la solución de cualquier problema planteado, en términos más sencillos, más fáciles de mantener y de probar.

Desde un punto de vista más formal, se define un TAD como una estructura algebraica compuesta por un conjunto de **objetos abstractos**, que modelan elementos del mundo, y un conjunto de operaciones para su manipulación, las cuales simulan el comportamiento que el elemento modelado tiene en el mundo del problema.

Se denomina un **cliente** de un TAD a toda rutina que utilice un objeto de dicho tipo. En particular, un TAD T1 es cliente de un TAD T2 si alguna operación de T1 es cliente de T2.

1.2.2. Representación de un Objeto Abstracto

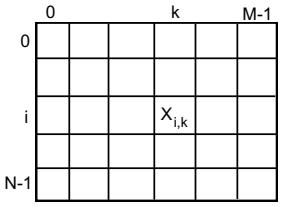
En el momento de comenzar el diseño de un TAD es necesario tener una representación abstracta del objeto sobre el cual se quiere trabajar, sin necesidad de establecer un compromiso con ninguna estructura de datos concreta, ni con ningún tipo de dato del lenguaje de programación seleccionado. Esto va a permitir expresar

las condiciones, relaciones, restricciones y operaciones de los elementos modelados, sin necesidad de restringirse a una representación interna concreta.

Para esto, lo primero que se hace es dar nombre y estructura a los elementos a través de los cuales se puede modelar el estado interno de un objeto abstracto, utilizando algún formalismo matemático o gráfico.

Ejemplo 1.1:

Para el TAD Matriz, una manera gráfica de representar el objeto abstracto sobre el cual se va a trabajar es la siguiente:



Con esta notación es posible hablar de cada uno de los componentes de una matriz, de sus dimensiones, de la noción de fila y columna, de la relación de vecindad entre elementos, etc., sin necesidad de establecer unas estructuras de datos concretas para manejarlas.

⏎

Ejemplo 1.2:

Para el TAD Diccionario, en el cual cada palabra tiene uno a más significados asociados, el objeto abstracto se puede representar mediante el siguiente formalismo:

$$< \text{elem-1} \ , ..., \text{elem-N} > \ \Big| \ \text{elem-i} \begin{cases} \text{palabra: String} \\ < \text{sig-1}, ..., \text{sig-k} > \end{cases}$$

Así, se define claramente su estructura general, dándole nombre a cada una de sus partes y relacionando las palabras con sus significados. En este caso, se utiliza la notación < ... > para expresar múltiples repeticiones y el símbolo de bifurcación para mostrar composición.

Otra manera de definir el mismo objeto abstracto podría ser la siguiente:

$$< [\ \text{palabra}_1, < s_{11}, ..., s_{1k} >] \ , ..., [\ \text{palabra}_N, < s_{N1}, ..., s_{Nk} >] >$$

Incluso, podría pensarse en la siguiente representación gráfica:

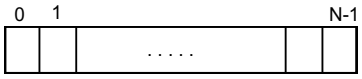
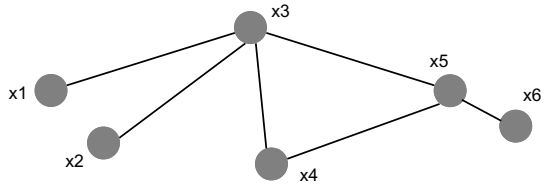
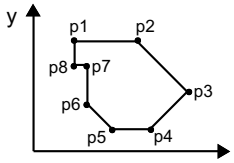
palabra-1	s-11, ..., s-1k
palabra-N	s-N1, ..., s-Nk

Lo importante en todos los casos, es que los componentes del objeto abstracto sean referenciables, y que su estructura global se haga explícita.

⏎

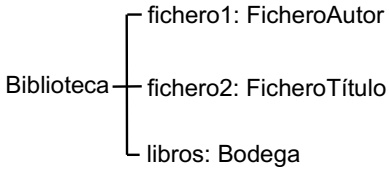
Ejemplo 1.3:

Existen objetos abstractos con una representación gráfica natural. Los siguientes son algunos ejemplos:

Conjunto	$\{ x_1, ..., x_N \}$
Cadena de caracteres	$"c_1c_2...c_N"$
Vector	
Polinomio	$c_o + c_1 x + c_2 x^2 + ... + c_N x^N$
Red	
Lista	$< x_1, x_2, ..., x_N >$
Polígono	

Ejemplo 1.4:

Algunos elementos del mundo se pueden modelar como una composición de **atributos**, los cuales representan las características importantes del objeto abstracto, en términos de otros elementos del mundo. En el caso de una biblioteca, se puede tener el siguiente formalismo:



Los atributos corresponden a objetos abstractos de los TAD FicheroAutor, FicheroTítulo y Bodega. En ese caso se dice que la Biblioteca es un cliente de dichos TAD. Por claridad en la notación, los nombres de los TAD se colocan en mayúsculas, mientras los nombres de los atributos tienen las características de cualquier variable.

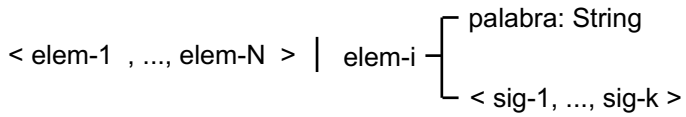
1.2.3. El Invariante de un TAD

El **invariante** de un TAD establece una noción de validez para cada uno de sus objetos abstractos, en términos de condiciones sobre su estructura interna y sus componentes. Esto es, indica en qué casos un objeto abstracto modela un elemento posible del mundo del problema. Por ejemplo, para el TAD Conjunto y la

notación { x_1, ..., x_N }, el invariante debe exigir que todos los x_i pertenezcan al mismo tipo de dato, y que sean diferentes entre sí, para que un objeto abstracto esté modelando realmente un conjunto. Estructuralmente, el invariante está compuesto por condiciones que restringen el dominio de los componentes internos y por relaciones entre ellos.

Ejemplo 1.5:

Para el TAD Diccionario, cuyo objeto abstracto tiene la siguiente estructura:



El invariante debe incluir tres condiciones, enunciadas a continuación en lenguaje natural y en lenguaje formal:

- Las palabras están ordenadas ascendentemente y no hay repetidas:

  $elem_i.palabra < elem_{i+1}.palabra$, $1 \leq i < N$

- Los significados están ordenados ascendentemente y no hay repetidos:

  $elem_i.sig_r < elem_i.sig_{r+1}$, $1 \leq i \leq N$, $1 \leq r < k$

- Toda palabra tiene asociado por lo menos un significado:

  $\forall\ elem_i = [\ palabra, < sig_1, ..., sig_k >\ ],\ k > 0$

Si un objeto del TAD Diccionario no cumple cualquiera de ellas, implica que no se encuentra modelando un diccionario real, de acuerdo con el modelaje que se ha hecho de ellos.

⏎

1.2.4. Especificación de un TAD

Un TAD se define con un nombre, un formalismo para expresar un objeto abstracto, un invariante y un conjunto de operaciones sobre este objeto. En este libro se usa el siguiente esquema:

TAD <nombre>
<Objeto abstracto>
<Invariante del TAD>
<Operaciones>

La especificación de las operaciones consta de dos partes: inicialmente, se coloca la funcionalidad de cada una de ellas (dominio y codominio de la operación), y, luego, su comportamiento. Esto último se hace mediante dos aserciones (precondición y postcondición) que indican la manera como se ve afectado el estado del objeto una vez se ha ejecutado la operación.

• <operación1>: <dominio> → <codominio>
• …
• <operaciónk>: <dominio> → <codominio>
<prototipo operación1>

```
/* Explicación de la operación */
{ pre: ... }
{ post: ... }
```

La precondición y la postcondición de una operación pueden referirse, únicamente, a los elementos que componen el objeto abstracto y a los argumentos que recibe. No puede incluir ningún otro elemento del contexto en el cual se va a ejecutar. En la especificación de las operaciones, se debe considerar implícito en la precondición y la postcondición, que el objeto abstracto sobre el cual se va a operar cumple el invariante. Eso quiere decir, que dichas aserciones sólo deben incluir condiciones adicionales a las de validez del objeto. Por claridad, si la precondición de una operación es *TRUE*, es decir no impone ninguna restricción al objeto abstracto ni a los argumentos, se omite de la especificación.

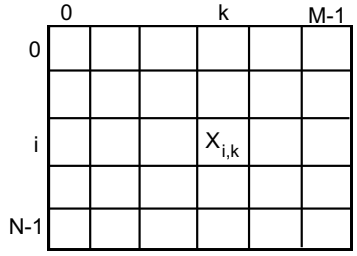
Es importante colocar una breve descripción de cada operación, de manera que el cliente pueda darse una rápida idea de los servicios que un TAD ofrece, sin necesidad de entrar a hacer una interpretación de su especificación formal. Esta última está dirigida sobre todo al programador.

Al seleccionar los nombres de las operaciones se debe tener en cuenta que no pueden existir dos operaciones con el mismo nombre en un programa, incluso si pertenecen a TAD diferentes. Por esta razón, es conveniente agregar un mismo sufijo a todas las operaciones de un TAD, de tal forma que las identifique. Es conveniente que este sufijo tenga por lo menos 3 caracteres.

Ejemplo 1.6:

Para definir el TAD Matriz de valores enteros, se puede utilizar la siguiente especificación:

TAD Matriz



{ inv: N > 0, M > 0 }

• crearMat:	int x int	→ Matriz
• asignarMat:	Matriz x int x int x int	→ Matriz
• infoMat:	Matriz x int x int	→ int
• filasMat:	Matriz	→ int
• columnasMat:	Matriz	→ int

```
Matriz crearMat( int fil, int col )
/* Construye y retorna una matriz de dimensión [ 0...fil-1, 0...col-1 ], inicializada en 0 */

{ pre: fil > 0, col > 0 }
{ post: crearMat es una matriz de dimensión [ 0...fil-1, 0...col-1 ], x_ik = 0 }
```

```
void asignarMat( Matriz mat, int fil, int col, int val )
```

```
/* Asigna a la casilla de coordenadas [ fil, col ] el valor val */

{ pre: 0 ≤ fil < N, 0 ≤ col < M }
{ post: X fil, col = val }
```

```
int infoMat( Matriz mat, int fil, int col )
/* Retorna el contenido de la casilla de coordenadas [ fil, col ]  */

{ pre:  0 ≤ fil < N, 0 ≤ col < M }
{ post: infoMat = X fil, col }
```

```
int filasMat( Matriz mat )
/* Retorna el número de filas de la matriz */

{ post: filasMat = N }
```

```
int columnasMat( Matriz mat )
/* Retorna el número de columnas de la matriz */

{ post: columnasMat = M }
```

En el caso del TAD Matriz, el invariante sólo establece una restricción para el número de filas y de columnas (coloca una limitante al dominio en el cual puede tomar valores). Cuenta con 5 operaciones para administrar un objeto del TAD: una para crearlo, una para asignar un valor a una casilla, otra para tomar el valor de una casilla, y dos para informar sus dimensiones. Con ese conjunto de operaciones, y sin necesidad de seleccionar unas estructuras de datos específicas, es posible resolver cualquier problema que involucre una matriz.

Es importante anotar que todo elemento utilizado como parte del formalismo de un objeto abstracto, puede utilizarse directamente como parte de la especificación de una operación. Ese es el caso de los valores N y M, utilizados como parte de la postcondición de las operaciones filasMat y columnasMat.

Ejemplo 1.7:

Para definir el TAD Diccionario, se puede utilizar la siguiente especificación, la cual supone que está ya diseñado el TAD String, al cual pertenecen las palabras y sus significados:

TAD Diccionario

$$< \text{elem-1} , ..., \text{elem-N} > \mid \text{elem-i} \begin{cases} \text{palabra: String} \\ < \text{sig-1}, ..., \text{sig-k} > \end{cases}$$

{ inv: $elem_i.palabra < elem_{i+1}.palabra$, $1 \le i < N$

$elem_i.sig_r < elem_i.sig_{r+1}$, $1 \le i \le N$, $1 \le r < k$

$elem_i = [palabra, < sig_1, ..., sig_k >], k > 0$ }

- crearDic: → Diccionario
- agregarPalabraDic: Diccionario x String x String → Diccionario

• eliminarPalabraDic:	Diccionario x String	→ Diccionario
• agregarSignifDic:	Diccionario x String x String	→ Diccionario
• numSignifDic:	Diccionario x String	→ int
• signifDic:	Diccionario x String x int	→ String

```
Diccionario crearDic( void )
/* Construye y retorna un diccionario sin palabras */

{ post: crearDic = < > }
```

```
void agregarPalabraDic( Diccionario dic, String pal, String sig )
/* Agrega la palabra pal al diccionario, suponiendo que no está, y le asocia como significado sig */

{ pre: dic = < elem_1, ..., elem_N >, ∀i elem_i.palabra != pal }
{ post: dic = < elem_1, ..., elem_i, [ pal, < sig > ], elem_i+1, ..., elem_N > }
```

```
void eliminarPalabraDic( Diccionario dic, String pal )
/* Elimina una palabra del diccionario, con todos los significados que tiene asociados */

{ pre: dic = < elem_1, ..., elem_N >, elem_i.palabra = pal }
{ post: dic = < elem_1, ..., elem_i-1, elem_i+1, ..., elem_N > }
```

```
void agregarSignifDic( Diccionario dic, String pal, String sig )
/* Agrega el significado sig a la palabra pal, presente en el diccionario */

{ pre: elem_i = [ pal, < sig_1, ..., sig_k > ] }
{ post: elem_i = [ pal, < sig_1, ..., sig, ..., sig_k > ] }
```

```
int numSignifDic(  Diccionario dic, String pal )
/* Retorna el número de significados de la palabra pal en el diccionario */

{ pre: elem_i = [ pal, < sig_1, ..., sig_k > ] }
{ post: numSignifDic = k }
```

```
String signifDic(  Diccionario dic, String pal, int numSig )
/* Retorna el significado numSig asociado con la palabra pal en el diccionario */

{ pre: elem_i = [ pal, < sig_1, ..., sig_k > ], 1 ≤ numSig ≤ k }
{ post: signifDic = sig_numSig }
```

1.2.5. Clasificación de las Operaciones

Las operaciones de un TAD se clasifican en 3 grupos, según su función sobre el objeto abstracto:

- Constructora: es la operación encargada de crear elementos del TAD. En el caso típico, es la encargada de crear el objeto abstracto más simple. Tiene la siguiente estructura:

  Clase <constructora> (<argumentos>)

{ pre: <condiciones de los argumentos> }

{ post: <condiciones del objeto inicial, adicionales al invariante> }

En los ejemplos anteriores, las operaciones crearMat y crearDic son las constructoras de los TAD Matriz y Diccionario respectivamente. Un TAD puede tener múltiples constructoras.

- Modificadora: es la operación que puede alterar el estado de un elemento del TAD. Su misión es simular una reacción del objeto. Su estructura típica es:

  void <modificadora> (<objetoAbstracto>, <argumentos>)

  { pre: <condiciones del objeto adicionales al invariante, condiciones de los argumentos> }

  { post: <condiciones del objeto adicionales al invariante> }

  En el ejemplo del TAD Matriz, la única modificadora es la operación asignarMat, que altera el contenido de una casilla de la matriz. Otra modificadora posible de ese TAD sería una que cambiara sus dimensiones. Al final de toda modificadora se tiene que seguir cumpliendo el invariante.

- Analizadora: es una operación que no altera el estado del objeto, sino que tiene como misión consultar su estado y retornar algún tipo de información. Su estructura es la siguiente:

  <tipo> <analizadora> (<objetoAbstracto>, <argumentos>)

  { pre: <condiciones del objeto adicionales al invariante, condiciones de los argumentos> }

  { post: <analizadora> = función (<estado del objetoAbstracto>) }

  En el TAD Matriz, las operaciones infoMat, filasMat y columnasMat son analizadoras. A partir de ellas, es posible consultar cualquier aspecto del objeto abstracto.

En la especificación del TAD es conveniente hacer explícito el tipo de operación al cual corresponde cada una de ellas, porque, en el momento de hacer el diseño de manejo de error, es necesario tomar decisiones diferentes. Existen además varias operaciones interesantes que se deben agregar a un TAD para aumentar su portabilidad. Son casos particulares de las operaciones ya vistas, pero, dada su importancia, merecen una atención especial. Entre estas operaciones se pueden nombrar las siguientes:

- Comparación: Es una analizadora que permite hacer calculable la noción de igualdad entre dos objetos del TAD.
- Copia: Es una modificadora que permite alterar el estado de un objeto del TAD copiándolo a partir de otro.
- Destrucción: Es una modificadora que se encarga de retornar el espacio de memoria dinámica ocupado por un objeto abstracto. Después de su ejecución el objeto abstracto deja de existir, y cualquier operación que se aplique sobre él va a generar un error. Sólo se debe llamar esta operación, cuando un objeto temporal del programa ha dejado de utilizarse.
- Salida a pantalla: Es una analizadora que le permite al cliente visualizar el estado de un elemento del TAD. Esta operación, que parece más asociada con la interfaz que con el modelo del mundo, puede resultar una excelente herramienta de depuración en la etapa de pruebas del TAD.
- Persistencia: Son operaciones que permiten salvar/leer el estado de un objeto abstracto de algún medio de almacenamiento en memoria secundaria. Esto permite a los elementos de un TAD sobrevivir a la ejecución del programa que los utiliza.

  En general, las aplicaciones están soportadas por manejadores de bases de datos que se encargan de resolver los problemas de persistencia, lo mismo que los problemas de concurrencia, coherencia, etc. Pero, para aplicaciones pequeñas, puede ser suficiente un esquema de persistencia sencillo, en el cual cada TAD sea responsable de su propia administración.

Ejemplo 1.8:

Con el fin de enriquecer el TAD Matriz con algunas de las operaciones interesantes mencionadas en la sección anterior, se tiene la siguiente especificación:

```
void copiarMat( Matriz mat1, Matriz mat2 )
/* Modificadora: Asigna a mat1 el contenido de mat2 */

{ pre: mat2 = MAT2 }
{ post: mat1.N = MAT2.N, mat1.M = MAT2.M, ∀ik, mat1.x_ik = MAT2.x_ik }
```

```
int igualMat( Matriz mat1, Matriz mat2 )
/* Analizadora: Informa si las dos matrices son iguales (dimensiones y contenido) */

{ post: igualMat = mat1.N = mat2.N, mat1.M = mat2.M, ∀ik, mat1.x_ik = mat2.x_ik }
```

```
void destruirMat( Matriz mat )
/* Modificadora: Destruye el objeto abstracto */

{ post: se ha destruido el objeto abstracto y se ha recuperado toda la memoria que ocupaba }
```

```
void imprimirMat( Matriz mat )
/* Analizadora: Presenta por pantalla el estado interno de la matriz */

{ post: se ha presentado por pantalla el contenido de la matriz }
```

```
Matriz cargarMat( FILE *fp )
/* Persistencia: Lee una matriz del archivo fp. Es un caso particular de constructora */

{ pre: archivo abierto listo para leer, información válida en el archivo  }
{ post: se ha construido una matriz con las dimensiones y la información del archivo }
```

```
void salvarMat( Matriz mat, FILE *fp )
/* Persistencia: Escribe una matriz en el archivo fp. Es un caso particular de analizadora */

{ pre: archivo abierto listo para escribir }
{ post: las dimensiones y el contenido de la matriz han sido escritos en el archivo, de tal manera que la
        operación cargarMat sea capaz de reconstruir la matriz }
```

⏎

1.2.6. Manejo de Error

Uno de los aspectos que se debe estudiar con especial cuidado en el momento de diseñar un TAD, es el manejo y recuperación de errores. Para esto existen tres aproximaciones básicas:

- Responsabilidad del usuario: La operación supone que el elemento del TAD sobre el cual se va a ejecutar la operación, lo mismo que los argumentos de la llamada, cumplen todos los requisitos planteados por la precondición de la operación. En caso de no cumplirlos, las consecuencias son problema del cliente y el comportamiento de la operación es indefinido.

- Informa el error: En este caso, cada operación -sobre todo las modificadoras- verifican que la operación haya tenido éxito. Lo usual es que retorne un código informándole al cliente el tipo de error detectado, o el éxito de la operación.

  La estructura típica de las modificadoras resulta la siguiente:

  int <modificadora> (<objetoAbstracto>, <argumentos>)

  { pre: <condiciones del objeto adicionales al invariante, condiciones de los argumentos> }

  { post: <condiciones del objeto adicionales al invariante>, <modificadora> = <código éxito>}

  { error: <caso de error>, <modificadora> = <código error> }

  Antes de diseñar cualquier operación es necesario hacer la lista de los errores posibles y asignarles constantes con un código de retorno. Esta forma de especificar el manejo de error tiene la ventaja de no oscurecer la especificación para situaciones normales.

- Responsabilidad de la operación: En este caso, la operación intenta recuperarse y si no lo consigue cancela la ejecución e informa la razón. Este esquema es utilizado por las operaciones que pueden verse afectadas por problemas de memoria, o de Entrada/Salida, para las cuales no tiene sentido continuar la ejecución si no se da la adecuada recuperación.

Ejemplo 1.9:

Para el TAD Diccionario, la operación que elimina una palabra tendría la siguiente especificación, si se utiliza el segundo tipo de manejo de error:

```
int eliminarPalabraDic( Diccionario dic, String pal )
/* Elimina una palabra del diccionario, con todos los significados que tiene asociados */

{ pre: dic = < elem1, ..., elemk >, elemi.palabra = pal, dic = DIC }

{ post: dic = < elem1, ..., elemi-1, elemi+1, ..., elemk >, eliminarPalabraDic = TRUE }

{ error: ∀i ei.palabra != pal, dic = DIC, eliminarPalabraDic = FALSE }
```

Una posible variante al segundo tipo de manejo de error, es retornar un mensaje con el error detectado, en lugar del código. Algo como lo sugerido en la siguiente especificación:

```
char *eliminarPalabraDic( Diccionario dic, String pal )
/* Elimina una palabra del diccionario, con todos los significados que tiene asociados */

{ pre: dic = < elem1, ..., elemk >, elemi.palabra = pal, dic = DIC }

{ post: dic = < elem1, ..., elemi-1, elemi+1, ..., elemk >, eliminarPalabraDic = NULL }

{ error: ∀i ei.palabra != pal, dic = DIC, eliminarPalabraDic = "ERROR: Palabra inexistente" }
```

Ejemplo 1.10:

Para el TAD Matriz, la operación que construye una matriz a partir del número de filas y columnas, tendría la siguiente especificación, si se utiliza el tercer enfoque de manejo de error:

```
Matriz crearMat( int fil, int col )
/* Construye y retorna una matriz de dimensión [ 0...fil-1, 0...col-1 ], inicializada en 0 */

{ pre: fil > 0, col > 0 }

{ post: crearMat es una matriz de dimensión [ 0...fil-1, 0...col-1 ], x_ik = 0 }

{ error: ( fil < 0 v col < 0 v no hay suficiente memoria ), mensaje + cancelación de la ejecución }
```

1.2.7. Metodología de Diseño de TAD

Para el diseño de un Tipo Abstracto se siguen los pasos mostrados en la figura 1.6, y explicados más adelante.

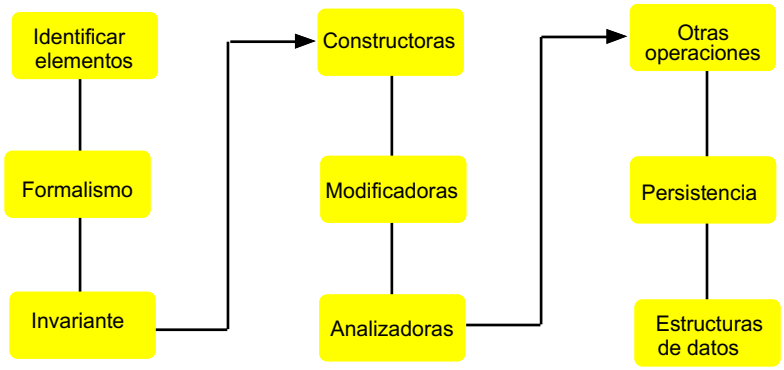


Fig. 1.6 - Pasos para el diseño de un Tipo Abstracto de Dato

- Identificar claramente los objetos del mundo que se quieren modelar, sus propiedades, relaciones, estructura, etc., y darles un nombre.
- Definir un formalismo para reflejar el estado de un objeto abstracto.
- Hacer explícitas, sobre el formalismo, mediante un invariante, las condiciones de validez de un objeto abstracto: restricciones y relaciones entre sus componentes.
- Diseñar las constructoras del TAD. Para esto, se debe pensar inicialmente en el objeto más elemental del TAD y la forma de generarlo mediante una operación. Luego, adicionar otras constructoras interesantes.
- Diseñar las modificadoras del TAD. Se debe pensar en todos los cambios que puede sufrir el elemento del mundo que se está modelando. Se comienza por las operaciones más elementales y se van agregando operaciones hasta que cualquier modificación imaginable se pueda expresar en términos de las operaciones ya incluidas. Luego se decide el tipo de manejo de error que se va a hacer y, por último, se hace una especificación de cada operación.

  Para decidir cuáles son los parámetros de cada modificadora, se establece cuál es la información necesaria para que el objeto pueda llevar a cabo el cambio de estado que debe producir la operación.

Durante esta etapa se debe intentar que el TAD asuma únicamente sus propias responsabilidades. Se debe evitar la tentación de delegar responsabilidades a los clientes o resolver un problema de un cliente como parte del TAD.

- Diseñar las analizadoras del TAD. Se deben colocar suficientes analizadoras para que cualquier cliente pueda obtener la información necesaria del objeto para su adecuado funcionamiento. La idea no es colocar una analizadora por atributo, sino buscar un conjunto de operaciones, independientes de las estructuras de datos concretas, que le permita a los clientes consultar el estado interno del objeto. Para cada una de estas operaciones se debe construir su especificación y, para aquellas que puedan fallar, definir el manejo que se va a hacer del posible error.
- Enriquecer el TAD con operaciones interesantes para los clientes. Casi siempre es conveniente ofrecer al cliente, además de las operaciones indispensables, un buen conjunto de operaciones de frecuente uso. Aunque estas operaciones se pueden escribir en términos de operaciones más sencillas del TAD, por eficiencia, satisfacción del cliente y extensibilidad del *software*, es conveniente agregarlas.
- Diseñar el manejo de la persistencia. Básicamente se deben agregar dos operaciones: una constructora que tome la información de memoria secundaria y cree un objeto del TAD, y otra que lleve el estado del objeto al disco en algún formato coherente con la primera operación. A lo largo de todo el libro se muestran ejemplos de cómo diseñar estas operaciones y de la forma de diseñar el esquema de persistencia de los objetos del TAD.
- Diseñar las estructuras de datos para representar el estado de un objeto (ver §1.3).

Ejemplo 1.11:

En este ejemplo se muestra el proceso completo de diseño de un TAD. Se va a utilizar como objeto abstracto un conjunto de valores naturales en un rango dado (i.e. enteros entre 25 y 100).

- Objeto abstracto: conjunto de números naturales en un rango dado, no vacío.
- Nombre: Conjunto (sufijo de las operaciones: Conj)
- Formalismo: inf : { x_1, x_2, ..., x_N } : sup
- Invariante: $inf \leq x_i \leq sup$ /* Todos los elementos están en el rango [inf...sup] */

  x_i != x_k, i != k /* No hay elementos repetidos */

  $1 \leq inf \leq sup$ /* El rango es válido */

- Constructoras: Únicamente se requiere una constructora, que permita crear conjuntos vacíos, dado un rango de enteros

  Conjunto crearConj(int infer, int super);
- Manejo de error: Retorno de un objeto inválido

  crearConj $\rightarrow$ { error: inf <1 v sup < inf, crearConj = NULL }

- Modificadoras: Son necesarias dos operaciones para alterar el estado de un conjunto. Una para agregar elementos y otra para eliminarlos. Estas dos operaciones son suficientes para simular cualquier modificación posible de un conjunto.

  int insertarConj(Conjunto conj, int elem);

  int eliminarConj(Conjunto conj, int elem);

- Manejo de error: Informe de fallas por código de retorno. Se seleccionan las siguientes constantes y códigos de error.

0	OK	Operación con éxito
1	RANGO	Elemento fuera de rango
2	INEXIS	Elemento inexistente
3	DUPLI	Elemento ya presente

insertarConj $\rightarrow$ { error: (elem < inf v elem > sup, insertarConj = RANGO) v

($\exists i \mid x_i$ = elem, insertarConj = DUPLI) }

eliminarConj $\rightarrow$ { error: elem != x_i $\forall i$, eliminarConj = INEXIS }

- Analizadoras: La operación básica de consulta a un conjunto es si un elemento pertenece a él. Adicionalmente, es necesario permitirle al cliente consultar los límites del rango de enteros que puede contener. Con estas tres operaciones es posible extraer toda la información del conjunto.

```
int estaConj( Conjunto conj, int elem );

int inferiorConj( Conjunto conj );

int superiorConj( Conjunto conj );
```

- Manejo de error: Ninguna analizadora puede fallar

- Operaciones interesantes: Se colocan operaciones para copiar, comparar, visualizar y destruir.

```
int igualConj( Conjunto c1, Conjunto c2 );          /* Informa si c1 == c2 */

int subConj( Conjunto c1, Conjunto c2 );            /* Informa si c1 ⊇ c2 */

void imprimirConj( Conjunto conj );                 /* Presenta los elementos del conjunto */

void destruirConj( Conjunto conj );                 /* Destruye un conjunto */
```

También agregan operaciones de amplio uso por parte de los clientes:

```
int cardinalidadConj( Conjunto conj );              /* Número de elementos de un conjunto */

void unirConjunto( Conjunto c1, Conjunto c2 );      /* c1 = c1 U c2 */
```

- Persistencia: Lo usual es colocar dos operaciones para la persistencia. Una, para salvar un objeto en disco y otra, para cargarlo de nuevo en memoria.

```
Conjunto cargarConj( FILE *fp );                    /* Lee un conjunto de disco */

void salvarConj( Conjunto conj, FILE *fp );         /* Salva un conjunto en disco */
```

El formato exacto de la persistencia en un archivo se define en el momento de diseñar las estructuras de datos. El cliente no necesita esta información.

- Especificación de las operaciones:

```
Conjunto crearConj( int infer, int super )
/* Crea un conjunto vacío con rango de valores potenciales [ infer .. super ] */

{ pre: 1 ≤ infer ≤ super }
{ post: crearConj = infer: { } :super }
```

```
int insertarConj( Conjunto conj, int elem )
/* Inserta al conjunto un elemento válido */
{ pre: conj = inf: { x1, x2, ..., xN } :sup, elem != xi ∀i, inf ≤ elem ≤ sup }
{ post: conj = inf: { x1, x2, ..., xN, elem } :sup }
```

```
int eliminarConj( Conjunto conj, int elem )
/* Elimina un elemento del conjunto */
{ pre: conj = inf: { x1, x2, ..., xN } :sup, xi = elem }
{ post: conj = inf: { x1, ..., xi-1, xi+1, ..., xN }: sup }
```

```
int estaConj( Conjunto conj, int elem )
/* Informa si un elemento se encuentra en el conjunto */
{ post: estaConjunto = ( ∃i | xi = elem ) }
```

```
int inferiorConj( Conjunto conj )
/* Retorna el límite inferior del rango de valores válidos del conjunto */
{ post: inferiorConj = inf }
```

```
int superiorConj( Conjunto conj )
/* Retorna el límite superior del rango de valores válidos del conjunto */
{ post: superiorConj = sup }
```

⏎

1.2.8. Uso de TAD en Solución de Problemas

Para resolver un problema utilizando como base un TAD, se debe expresar la solución en términos de las operaciones disponibles sobre los objetos abstractos. Un TAD puede ser visto, en el momento de utilizarlo, como un tipo de dato básico del lenguaje, con un conjunto restringido de operaciones.

Ejemplo 1.12:

Desarrollar una rutina que retorne un conjunto que corresponda a la intersección de dos conjuntos dados.

```
Conjunto interConj( Conjunto c1, Conjunto c2 )
/* pre: c1 = inf1 :{ x1, x2, ..., xN }: sup1, c2 = inf2 : { y1, y2, ..., yM }: sup2 */
/* post: interConj = min( inf1, inf2 ) : { z1, z2, ..., zk } : max( sup1, sup2 ) | zi ∈ c1 ∧ zi ∈ c2 */
{   int i;
    int infer = min( inferiorConj( c1 ), inferiorConj( c2 ) );
    int super = max( superiorConj( c1 ), superiorConj( c2 ) );
    Conjunto ct = crearConj( infer, super );
    for( i = infer; i <= super; i++ )
        if( estaConj( c1,i ) && estaConj( c2,i ) )
            insertarConj( ct, i );
    return ct;
}
```

⏎

Ejemplo 1.13:

Una rutina que sume a una matriz mat1 otra matriz mat2, viene dada por el siguiente código. Fíjese que no existe en el algoritmo ningún compromiso con estructuras de datos, sino solamente se plantea la solución para cualquier implementación que se haga de matrices.

```
void sumarMatriz( Matriz mat1, Matriz mat2 )
/* pre: mat1.N = mat2.N, mat1.M = mat2.M (v.g. tienen las mismas dimensiones) */
/* post: mat1 += mat2 */
{   int i, k;
    for( i = 0; i < filasMat( mat1 ); i++ )
        for( k = 0; k < columnasMat( mat1 ); k++ )
            asignarMat( mat1, i, k, infoMat( mat1, i, k ) + infoMat( mat2, i, k ) );
}
```

1.2.9. Genericidad: TAD Paramétricos

La idea de desarrollar un TAD contenedor parametrizado tiene la ventaja de que deja claro qué puntos del diseño son dependientes del tipo de elemento que maneja. Lo ideal es poder reutilizar todo el *software* de una aplicación a otra, y no solo el diseño, de tal forma que sea posible contar con librerías genéricas perfectamente portables, capaces de contener elementos de cualquier tipo. La sintaxis para especificar un TAD paramétrico se puede apreciar en el siguiente ejemplo.

Ejemplo 1.14:

Si se quiere definir un TAD Conjunto, para cualquier tipo de elemento, se puede utilizar un esquema como el siguiente:

TAD Conjunto[Tipo]		
	$\{ x_1, x_2, ..., x_N \}$	
{ inv: x_i != x_k, i != k, x_i pertenece al TAD Tipo }		
• crearConjunto:		→ Conjunto
• insertarConjunto:	Conjunto x Tipo	→ Conjunto
• eliminarConjunto:	Conjunto x Tipo	→ Conjunto
• estaConjunto:	Conjunto x Tipo	→ int
• vacioConjunto:	Conjunto	→ int

```
void insertarConjunto( Conjunto conj, Tipo elem )
{ pre: conj = { x1, x2, ..., xN }, elem != xi ∀i }
{ post: conj = { x1, x2, ..., xN, elem } }
```

Ejercicios Propuestos

1.1. Una lista es una estructura muy flexible, de longitud variable, en la cual se pueden agregar y eliminar elementos en cualquier posición. Diseñe el TAD Lista de enteros.

1.2. Una pila es una estructura lineal, en la cual únicamente es posible insertar y eliminar por uno de sus extremos. Diseñe el TAD Pila[Tipo].

1.3. Una fila es una estructura lineal, en la cual entran los elementos por un extremo y salen por el otro, una vez son atendidos. Diseñe el TAD Fila[Tipo]

1.4. En muchos lenguajes de programación (i.e. Pascal) existe un tipo de dato denominado *string*, que corresponde a una cadena de caracteres de longitud variable. En C el manejo de este tipo de objetos abstractos es limitado. Diseñe el TAD String.

1.5. Con el fin de hacer manipulación simbólica de polinomios, es conveniente tener definido un TAD Polinomio, a través del cual sea posible sumarlos, multiplicarlos, derivarlos, etc. Diseñe y especifique un TAD Polinomio.

1.6. En diversas aplicaciones es necesario manejar valores numéricos por fuera del rango representable con el tipo int de un lenguaje de programación. Piense, por ejemplo, en un número con 200.000 dígitos. Para trabajar con estos valores, es necesario contar con un TAD SuperEntero, que sea capaz de manejar números enteros positivos de cualquier longitud. Haga el diseño de este TAD.

1.7. Se define un vector en un espacio tridimensional como una magnitud (un escalar) y una dirección (dada por 3 coordenadas). Diseñe el TAD Vector3D. Debe incluir operaciones como producto punto y producto cruz.

1.8. 🕒 Se quiere modelar una calculadora como un objeto abstracto, con las siguientes características mínimas: 10 memorias, operaciones aritméticas básicas, borrar la pantalla, desplegar el valor actual en pantalla, leer un valor de teclado, etc. Haga el diseño del TAD Calculadora.

1.9. En un sistema de atención al público, el orden de entrada corresponde estrictamente al orden de llegada. Hay casos en los cuales este tipo de filas de espera deben ser más flexibles. Considere, por ejemplo, la fila de entrada a la unidad de urgencias de un hospital. Hay enfermos que por su estado merecen una mayor prioridad que otros y debe entrar antes que algunos de los que llegaron antes. Diseñe el TAD Fila con prioridades.

1.10. Una **tabla de asociación** es una estructura que permite asociar llaves con información. Por ejemplo, si se quiere almacenar la información de un conjunto de personas y se quiere permitir el acceso por la cédula (llave) se utiliza este tipo de contenedora. Diseñe el TAD genérico Tabla[Llave, Tipo].

1.11. Un texto es una secuencia de renglones de cualquier longitud, cada uno de los cuales corresponde a una cadena de caracteres. Diseñe y especifique el TAD Texto.

1.12. Un directorio es una estructura ordenada, en la cual se asocia un teléfono y una dirección con un nombre y un apellido. Esta pareja [nombre, apellido] no es única, de manera que la respuesta a una consulta puede ser una lista de teléfonos. Diseñe el TAD Directorio telefónico teniendo en cuenta las restricciones anteriores.

1.13. La pantalla del computador se puede modelar mediante el conjunto de textos y elementos gráficos (líneas y círculos) allí desplegados, cada uno de los cuales se encuentra en una posición dada. Diseñe el TAD Pantalla.

Para los siguientes enunciados abiertos, diseñe los TAD que considere necesarios para modelar todos los elementos involucrados:

1.14. En la universidad se maneja la información académica de cada uno de sus estudiantes. Allí se coloca el apellido, el nombre, la fecha de nacimiento, la facultad a la que pertenece y la lista de materias que ha cursado con la nota obtenida.

1.15. Un club de *squash* está formado por 9 canchas, disponibles a los socios en el horario 7 am - 7pm (turnos cada hora). Los socios son atendidos por teléfono, y así pueden reservar o cancelar turnos hasta con tres días de anticipación.

1.16. Un banco administra toda la información de sus clientes, en la cual aparece registrada cada consignación y retiro, lo mismo que el saldo actual. Los clientes pueden abrir o cerrar una cuenta, consultar el saldo, depositar o retirar, consultar las últimas transacciones y transferir dinero de una cuenta a otra del mismo banco.

1.17. En un almacén se debe manejar el inventario de los productos con que cuenta. Cada producto tiene un código, un nombre, un precio y una cantidad. Se necesita desarrollar un sistema de información para que el gerente administre toda la información referente al inventario.

1.18. 🕒 Un hospital está compuesto por un conjunto de cuartos en los cuales se sitúan los pacientes. Cada cuarto tiene una identificación y una capacidad. Cada paciente es atendido por un médico especializado en la enfermedad que éste sufre. La historia clínica de cada paciente incluye un apellido, un nombre, un sexo, una enfermedad y un número de días hospitalizado.

Por reglamento del hospital, todos los pacientes que se encuentran en una misma habitación deben tener la misma enfermedad, para evitar posibles contagios. Además, si llega un paciente y no existe un médico disponible para la enfermedad que éste sufre, o si no hay un lugar disponible en una habitación, el enfermo no se admite. Otra regla del hospital es que ningún médico puede tener a su cargo más de 10 pacientes. Eso garantiza la calidad del servicio. Cuando se da de alta un paciente, se le presenta una cuenta de cobro que corresponde al número de días que estuvo hospitalizado por $20.000, más un 14% de impuestos.

Usando los TAD definidos en los ejemplos y ejercicios de las secciones anteriores, desarrolle los algoritmos que resuelven los siguientes problemas:

1.19. Utilizando los TAD Diccionario y String, desarrolle un procedimiento que, dados una palabra y un significado, informe si en el diccionario se encuentran relacionados.

1.20. Utilizando los TAD Diccionario y String, desarrolle una rutina que, dadas dos palabras, informe si son sinónimas. Esto es, si comparten por lo menos un significado en el diccionario.

1.21. Utilizando las operaciones de los TAD Diccionario y String, desarrolle una rutina que, dadas dos palabras, retorne el número de significados que comparten en un diccionario. Debe considerar el caso en el cuál no comparten ninguno.

1.22. Utilizando el TAD Conjunto, desarrolle una función que calcule y retorne el conjunto diferencia de otros dos conjuntos.

1.23. 🕒 Utilizando el TAD Conjunto de enteros en un rango, desarrolle una función que retorne su mediana. Se define la **mediana** de un conjunto como el valor del conjunto que cumple que la mitad de los elementos presentes son mayores que él, y la otra mitad menores o iguales.

1.24. Utilizando las operaciones del TAD Matriz, desarrolle una función que calcule y retorne la transpuesta de una matriz.

1.25. Utilizando las operaciones del TAD Matriz, desarrolle una función que informe si una matriz es un cuadrado mágico. Una matriz es un **cuadrado mágico** si el resultado de sumar los elementos de cada una de las filas es el mismo e igual a la suma de los elementos de cada una de las columnas y a la

suma de los elementos de cada una de las diagonales principales. Por ejemplo, la matriz de la figura es un cuadrado mágico, puesto que tanto sus filas como sus columnas y sus diagonales suman 15.

8	1	6
3	5	7
4	9	2

1.3. Diseño de Estructuras de Datos

Para hacer operacional un TAD, es decir, para que funcione realmente sobre un lenguaje de programación, es necesario implementarlo y dejarlo disponible para que las otras partes del *software* lo utilicen. Esto es equivalente a aumentar el lenguaje para que maneje un nuevo tipo de dato, generando una estratificación conceptual como la mostrada en la figura 1.7.

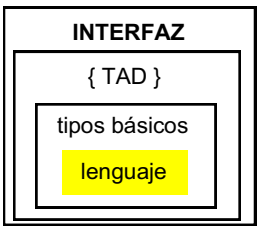


Fig. 1.7 - Estratificación conceptual

Para implementar un TAD se siguen dos pasos:

- Diseñar las estructuras de datos que van a representar cada objeto abstracto.
- Desarrollar una función, por cada operación del TAD, que simule el comportamiento del objeto abstracto, sobre las estructuras de datos seleccionadas.

En esta sección se estudia la etapa de diseño de las estructuras de datos para un TAD. En la siguiente, se aborda la parte de implementación de operaciones.

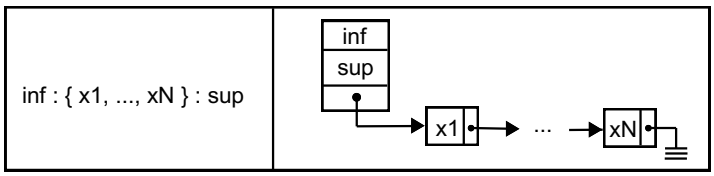
1.3.1. Relación Objeto Abstracto-Estructuras de Datos

El propósito de esta etapa es diseñar unas estructuras de datos que representen de manera adecuada el estado de un objeto abstracto. Para esto, es necesario verificar que toda la información sea almacenada convenientemente en dichas estructuras. El resultado es un **esquema de representación** y almacenamiento del objeto abstracto en estructuras concretas de datos. Este esquema corresponde a una función que explica la manera de interpretar y almacenar la información. Es necesario verificar que los casos especiales (lista vacía, conjunto vacío, pila llena, etc.) tengan una adecuada representación con las estructuras de datos diseñadas.

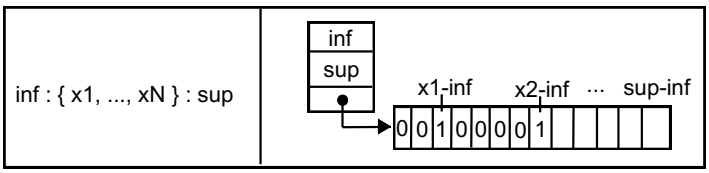
Ejemplo 1.15:

Para el TAD Conjunto de enteros en un rango, se pueden utilizar diversas estructuras de datos. El esquema de representación se puede expresar de manera gráfica, como se muestra a continuación, señalando la forma de almacenar cada uno de los elementos que componen el objeto abstracto. En este ejemplo se presentan dos esquemas diferentes, ambos válidos.

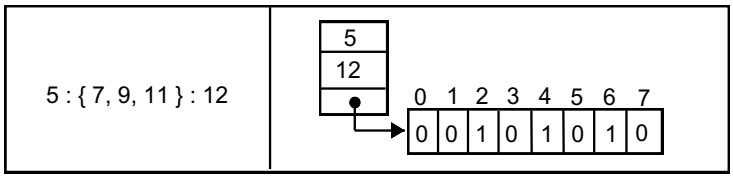
La primera posibilidad consiste en un registro con 3 campos: dos con los límites del conjunto y uno con un apuntador a una lista sencillamente encadenada con sus elementos. Gráficamente, el esquema se podría expresar de la siguiente manera:



El segundo diseño corresponde a un vector de tamaño dinámico de valores lógicos (*TRUE*, *FALSE*), con una posición por cada posible elemento del conjunto. Si el elemento está presente, en la casilla correspondiente hay un *TRUE*. En caso contrario, un *FALSE*.



Este segundo esquema se puede ilustrar mediante el siguiente caso:



Note la eficiencia con la cual se pueden hacer las operaciones de inserción y supresión de elementos sobre este segundo esquema.

⏎

1.3.2. Consideraciones Básicas

Para cualquier proceso de diseño, en particular para el diseño de estructuras de datos, es difícil hablar de un método único e infalible, que garantice la obtención de un buen producto. Una metodología se debe contentar con dar unas guías generales de trabajo, que incluyan todas las consideraciones teóricas del producto que se va a diseñar, y unos mecanismos de evaluación y medición de calidad (fórmulas, teoremas, ecuaciones, etc.). A partir de esto, el diseñador debe utilizar su experiencia y habilidad para lograr un buen diseño, tanto si es un ingeniero civil o mecánico, como si es un ingeniero de *software*.

En esta parte del capítulo, se presentan algunos lineamientos generales del diseño de estructuras de datos, y se ilustran, mediante el uso de ejemplos muy sencillos, algunas de las opciones y decisiones que tiene que tomar un diseñador. Estos ejemplos utilizan únicamente estructuras lineales de datos, de complejidad mínima. A lo largo de todo el libro, se muestra la manera de extender esta teoría a estructuras más complejas.

Básicamente, en la metodología de diseño se siguen las líneas de desarrollo expuestas a continuación:

- Construir un conjunto de diseños válidos, es decir, que satisfagan las condiciones establecidas. Esto se refiere a si una estructura de datos es capaz de almacenar y recuperar toda la información asociada con el objeto abstracto que quiere representar. Para cada uno de estos diseños, se debe comenzar por una

versión inicial (un borrador) e irlo refinando según consideraciones de eficiencia, facilidad de mantenimiento, etc., hasta llegar a una solución viable.

- Evaluar y comparar los diseños obtenidos y escoger uno de acuerdo con el problema específico que se quiera resolver. Este proceso se expone con detenimiento en una sección posterior.
- Hacer la declaración de las estructuras de datos, de manera que los clientes puedan comenzar a utilizar los objetos del TAD. Para esta parte se deben tener en cuenta las siguientes consideraciones, básicas para la portabilidad del producto:

  (1) Los objetos se van a manejar en memoria dinámica, de manera que un objeto siempre va a corresponder a un apuntador. La declaración del TAD <nombre> debe tener la siguiente estructura:

  typedef <estructuras de datos> T<nombre>, *<nombre>;

  (2) Debido al primer punto, los objetos siempre se pasan como parámetros por referencia a las operaciones que los manipulan.

  (3) Si el objeto tiene un conjunto de atributos asociados, se utiliza un registro para encapsularlos y un campo por cada uno de los atributos.

  (4) Si un atributo corresponde a otro objeto del mundo, se utiliza un elemento del TAD correspondiente como atributo.

  (5) Si ya se cuenta con otros TAD, se debe intentar reutilizar el código, o utilizar un TAD genérico como parte del modelaje.

Ejemplo 1.16:

La declaración de las estructuras de datos para los esquemas de representación sugeridos en el ejemplo anterior son:

```
/* Primer esquema */
typedef struct
{   int inf;                    /* Límite inferior */
    int sup;                    /* Límite superior */
    struct Nodo *lst;           /* Lista encadenada de elementos */
} TConjunto, *Conjunto;

/* Segundo esquema */
typedef struct
{   int inf;                    /* Límite inferior */
    int sup;                    /* Límite superior */
    int *vec;                   /* Vector de longitud dinámica, pedido en ejecución, de acuerdo */
                                /* con los valores de los atributos inf y sup */
} TConjunto, *Conjunto;
```

En cualquier caso, la declaración de una variable del TAD debe ser:

```
Conjunto conj;
```

Para hacer la evaluación de los diseños de las estructuras de datos válidas (v.g. que son capaces de representar todo el estado de un objeto), se tienen en cuenta tres factores principales:

- Complejidad de las operaciones del TAD bajo esa implementación. Se hace una tabla como se muestra en el siguiente ejemplo, para poder medir la eficiencia de las operaciones.

- Espacio ocupado en términos de los atributos que maneja. Se hace una columna adicional a la tabla del primer punto.
- Restricciones inherentes a la representación. Se coloca una columna en la que se hacen explícitas las restricciones que tienen los objetos abstractos, al representarse mediante esas estructuras de datos

Ejemplo 1.17:

En el ejemplo 1.15, se presentaron dos posibles diseños de estructuras de datos para el TAD Conjunto de enteros en un rango. La tabla comparativa para estos dos diseños es:

	crear	insertar	eliminar	está	inferior	superior	espacio	restricciones
1	O(1)	O(1)	O(N)	O(N)	O(1)	O(1)	O(N)	ninguna
2	O(M)	O(1)	O(1)	O(1)	O(1)	O(1)	O(M)	tamaño de M

donde: N = cardinalidad del conjunto
M = sup-inf+1

Este resultado confirma la relación tiempo-espacio mencionada en el capítulo anterior: casi siempre es posible mejorar el tiempo de respuesta de una operación a costa de espacio en memoria.

⏎

Para escoger uno de los diseños de estructuras de datos es necesario tener en cuenta lo siguiente, con respecto al uso que se le va a dar al TAD dentro del *software*:

- Operaciones críticas. Para el ejemplo anterior, si es muy dinámico el conjunto (se insertan y eliminan muy frecuentemente sus elementos), o si se necesita contestar muy rápidamente a la pregunta de si un elemento está en el conjunto, es mejor la segunda implementación. Todo eso lo hace en O(1).
- Restricciones de implementación inaceptables. En el ejemplo del TAD Conjunto, el gran problema de la segunda implementación es la cantidad de memoria que ocupa. Si alguien quiere manejar un conjunto con enteros en el rango [0...64K], las estructuras de datos van a ocupar 128K, suponiendo una representación de la máquina de 2 bytes por entero (se puede pensar en una representación a nivel de bits, pero siempre habrá un conjunto suficientemente disperso para que esa representación resulte inadecuada). Eso implica que, si no se conoce con certeza un límite razonable para el tamaño del vector, se debe descartar la segunda implementación.
- Restricciones de espacio. En algunos casos el espacio ocupado por las estructuras de datos descarta su posible utilización. El ejemplo es el mismo utilizado en el punto anterior.
- Dificultad de escribir los algoritmos vs. eficiencia esperada de los mismos. Si el TAD únicamente se va a utilizar en un proceso no crítico, no se justifica la dificultad de escribir un algoritmo sobre estructuras de datos complejas. En ese caso, es mejor escoger las estructuras de datos para las cuales los algoritmos resulten más fáciles de implementar y probar.
- Número de movimientos de la información. Cuando el tamaño de la información que se maneja es considerable (i.e. registros de 4K bytes), el costo de su movimiento comienza a ser otro factor que se debe tener en cuenta en el tiempo de ejecución de un algoritmo. No es lo mismo desplazar la información en un vector de enteros, que mover textos completos sobre el mismo tipo de estructura.

En las siguientes secciones aparecen algunas de las opciones que tiene un ingeniero para el diseño de estructuras de datos, ilustradas mediante ejemplos sobre la representación interna de una cadena de caracteres.

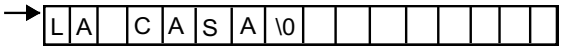
1.3.3. Representación de Longitud Variable

El primer punto se refiere a la representación de atributos que pueden aparecer una o varias veces al interior de un objeto. Se debe escoger una estructura de datos sobre la cual se pueda hacer una simulación de longitud variable. En general, cualquier TAD contenedor de los presentados a lo largo del libro puede utilizarse para esto. En el siguiente ejemplo se muestran los esquemas más sencillos que existen.

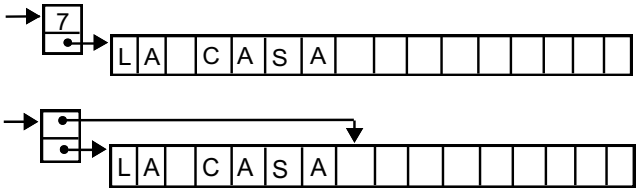
Ejemplo 1.18:

Para representar cadenas de caracteres de cualquier longitud, es posible utilizar alguno de los siguientes esquemas. Se ilustra la representación con la cadena "LA CASA":

- Vector con marca de final: la información se sitúa secuencialmente en un vector y se coloca una marca para señalar la posición final. Esta marca hace parte de la información, luego debe ser un elemento distinguido (el caracter *NULL* en este ejemplo). Así maneja internamente las cadenas de caracteres el lenguaje C.

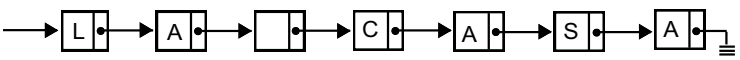


- Vector con campo adicional para marcar la longitud: la información se coloca secuencialmente en el vector y desde un segundo campo se indica la posición final. Este segundo campo puede ser un entero o un apuntador.



  También es posible colocar este campo como parte de la información, tal como hace el lenguaje Pascal para representar la longitud variable de un elemento de tipo *String*.

- Apuntadores: El medio ideal para manejar múltiples repeticiones de información es el encadenamiento de registros mediante apuntadores. La estructura más sencilla es la siguiente:



  Se puede pensar también en listas doblemente encadenadas, listas circulares, etc.

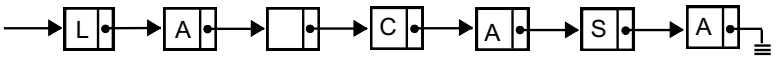
⏎

1.3.4. Manejo de Información Redundante

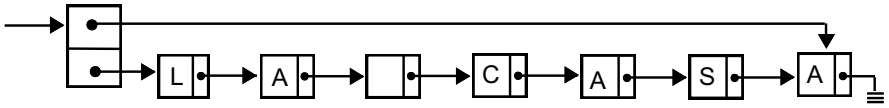
En algunos casos, aunque la información se pueda calcular a partir de los datos almacenados en las estructuras, es conveniente agregar información redundante. Esto puede mejorar el tiempo de ejecución de las operaciones. Para decidir si es conveniente agregar información, se debe calcular el costo marginal de ejecución de las modificadoras para mantener este nuevo dato actualizado, contra el tiempo que se gana al tenerlo.

Ejemplo 1.19:

Suponga que una operación del TAD String es anexarStr, que agrega un caracter al final de toda la cadena. Si las estructuras de datos seleccionadas son apuntadores, la complejidad será O(n), donde n es el número de caracteres de la cadena, porque debe recorrer toda la estructura antes de poder adicionar el elemento.



La posibilidad de adicionar un campo en las estructuras de datos, que indique la posición final de la cadena, para aumentar la eficiencia de la operación anexarStr, y hacerla en O(1), se debe medir contra el costo adicional que le implica a cada modificadora mantenerlo actualizado.



Suponga que el TAD tiene dos modificadoras: insertarStr y eliminarStr, que insertan y eliminan respectivamente un caracter, dada una posición en la cadena. Con las estructuras originales la complejidad de ambas es O(n), lo cual se sigue manteniendo igual aunque deban actualizar el nuevo campo. Esto hace que resulte una buena decisión adicionarlo a las estructuras de datos.

⏎

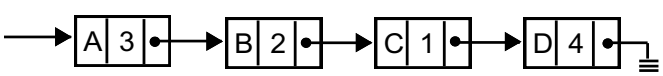
El caso límite de las representaciones redundantes es el de las **representaciones múltiples**. En ellas se decide tener duplicada y estructurada de dos formas distintas toda la información, cada una con el fin de responder eficientemente a algún conjunto de operaciones. Esto se suele aplicar comúnmente en el desarrollo de *software* gráfico, en el cual el tiempo de respuesta es un factor crítico, y las estructuras de datos que permiten realizar operaciones como intersección o unión de sólidos tridimensionales, suelen ser muy malas para hacer la visualización realista de los mismos. Esto obliga a que cada objeto tenga en sus estructuras de datos todas sus características geométricas duplicadas, bajo dos esquemas diferentes, orientados a operaciones distintas.

1.3.5. Representaciones Compactas vs. Exhaustivas

Las representaciones vistas hasta ahora corresponden a representaciones exhaustivas: se representa independientemente cada componente del objeto abstracto. En esta sección se estudia la posibilidad de utilizar una representación compacta, que disminuya el volumen de la información almacenada, a costa muchas veces de pérdida de eficiencia en las operaciones.

Ejemplo 1.20:

Una manera compacta de representar cadenas de caracteres se puede basar en la idea de que una secuencia de caracteres iguales se puede colocar en un solo nodo. En la figura se muestra la representación de la cadena "AAABBCDDDD":



⏎

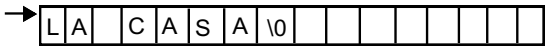
1.3.6. Ordenamiento Físico vs. Lógico

Cuando los componentes de un objeto abstracto deben guardar un orden relativo entre ellos, la representación interna escogida, en lugar de un ordenamiento físico, puede corresponder a un ordenamiento virtual. Esto es, aunque en las estructuras de datos los elementos no se encuentren físicamente consecutivos,

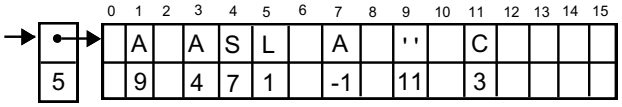
existe la forma de interpretarlos como si lo estuvieran. Esto tiene dos ventajas: la primera, que no es necesario mover la información en sí cada vez que se agrega un elemento, y la segunda, que se pueden manejar al mismo tiempo múltiples ordenamientos.

Ejemplo 1.21:

Si se escoge un vector con marca de final para manejar un elemento del TAD String:



El ordenamiento físico se podría remplazar por un ordenamiento lógico, en el cual, cada registro, informa cuál es el elemento que sigue en la secuencia. En este caso la solución no parece muy adecuada, pero si la información que se debe manejar no fuera un caracter sino un registro más complejo (i.e. un texto), este enfoque sería una buena solución.



En el registro de encabezado aparece un campo con el valor 5, que indica el punto de entrada en la estructura física. Cada elemento tiene la posición en el vector del elemento que lo sucede. El último elemento tiene una marca especial (-1 en este caso), para indicar el final del encadenamiento.

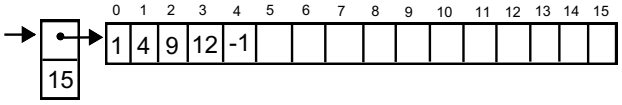
⏎

1.3.7. Representación Implícita vs. Explícita

La única manera de representar un componente del objeto abstracto no es almacenándolo en las estructuras de datos. Se puede pensar en una representación implícita, en la cual, si el elemento no se encuentra explícitamente como parte de la información, significa que tiene una cierta característica especial, o un valor específico.

Ejemplo 1.22:

Suponga que se quiere manejar un ***bitstring***: un elemento formado por una secuencia de tamaño variable de ceros y unos. Es posible representar de manera explícita únicamente los unos y de manera implícita los ceros. Por ejemplo, el *bitstring* "010010000100100", se representaría así:



Estas estructuras de datos simulan longitud variable marcando el final de la secuencia con un -1 y utilizan un campo para almacenar el tamaño del *bitstring*. En el vector se encuentran las posiciones del *bitstring* que tienen valor 1. Las que no se encuentran allí tienen valor 0. En este ejemplo, sólo se necesitan 5 posiciones para representar un *bitstring* de 15 elementos.

⏎

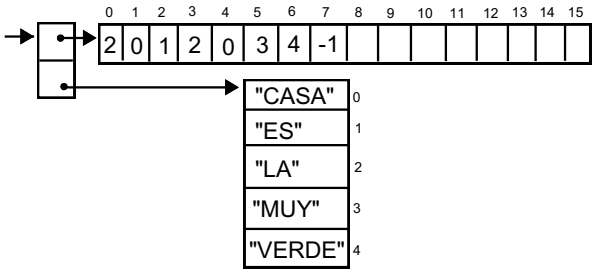
1.3.8. Incorporación de Restricciones del Problema

Algunas características específicas del problema que se está resolviendo pueden permitirle al ingeniero de *software* encontrar mejores representaciones internas, más compactas, o más eficientes. Se debe verificar

que sean características no volátiles del mundo (v.g. que difícilmente van a cambiar), porque de lo contrario se pueden generar problemas de mantenimiento.

Ejemplo 1.23:

Suponga que se quiere manejar una cadena de caracteres, en la cual sólo pueden aparecer palabras de un conjunto limitado y predefinido de ellas, separadas por un blanco. En lugar de hacer una representación general de las cadenas de caracteres, el problema permite diseñar unas estructuras de datos muy compactas, como las que se sugieren a continuación. En el ejemplo se representa la cadena "LA CASA ES LA CASA MUY VERDE".

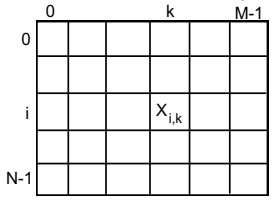


En las estructuras se coloca un diccionario con las palabras que pueden aparecer, y en el vector únicamente se hace referencia al orden en el cual éstas se encuentran en la cadena que se quiere representar.

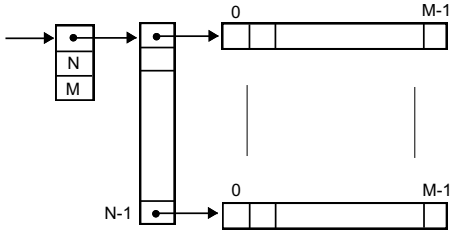
⏎

1.3.9. Estructuras de Datos para el TAD Matriz

Para las estructuras de datos del TAD Matriz se pueden hacer muchos diseños diferentes, cada uno con sus ventajas y desventajas, y cada uno orientado hacia un tipo específico de problema. A continuación se presentan seis de ellos, para ilustrar la gran gama de posibilidades que tiene un diseñador, incluso para un objeto abstracto tan sencillo como es una matriz. El formalismo para expresar una matriz es:



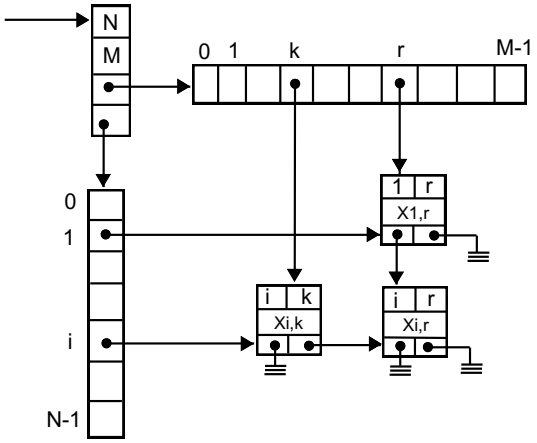
a-) **Matriz en memoria dinámica.** Es una de las implementaciones más sencillas y naturales. El espacio en memoria para los elementos de la matriz se maneja como un vector de apuntadores a apuntadores, y la constructora es la encargada de reservar en memoria dinámica el espacio necesario. Las estructuras de datos se pueden definir mediante el siguiente dibujo:



Este diseño tiene las siguientes declaraciones:

```
typedef struct
{     int N;              /* Número de filas */
      int M;              /* Número de columnas */
      int **mat;          /* Matriz dinámica de enteros, pedida en ejecución */
} TMatriz, *Matriz;
```

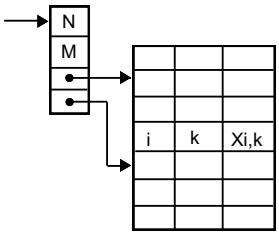
b-) **Matriz dispersa.** Esta implementación se utiliza en casos en los cuales la matriz es de tamaño considerable, y existen muy pocos valores diferentes de cero en su interior. La idea es representar de manera explícita los valores no nulos, y encadenarlos por las filas y las columnas, como se muestra en la siguiente figura:



Los algoritmos son mucho más complicados de implementar y más lentos que los del primer diseño, pero el ahorro en espacio justifica algunas veces este costo adicional. Este diseño tiene las siguientes declaraciones:

```
struct Nodo
{     int fil;                  /* Fila a la cual pertenece el elemento */
      int col;                  /* Columna a la cual pertenece el elemento */
      int info;                 /* Elemento almacenado en la posición [ fil, col ] */
      struct Nodo *sigFila;     /* Siguiente nodo en la misma fila */
      struct Nodo *sigColumna;  /* Siguiente nodo en la misma columna */
};

typedef struct
{     int N;                    /* Número de filas */
      int M;                    /* Número de columnas */
      struct Nodo **filas;      /* Vector de apuntadores a nodos de la matriz */
      struct Nodo **columnas;   /* Vector de apuntadores a nodos de la matriz */
} TMatriz, *Matriz;
```

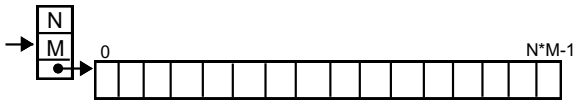
c-) **Vector de tripletas.** Es otra implementación usada para manejar matrices con múltiples valores nulos. Esta estructura de datos se basa en un vector de tripletas, en el cual, en cada posición, se almacena la fila, la columna y el valor respectivo. Se tiene además un apuntador a la última posición ocupada del vector, para simular así longitud variable. Todo elemento que no aparezca allí referenciado tiene valor cero. Los elementos no tienen ningún orden específico en la estructura de datos.



Este diseño tiene las siguientes declaraciones:

```
struct Nodo
{     int fil;              /* Fila a la cual pertenece el elemento */
      int col;              /* Columna a la cual pertenece el elemento */
      int info;             /* Elemento almacenado en la posición [ fil, col ] */
};

typedef struct
{     int N;                /* Número de filas */
      int M;                /* Número de columnas */
      struct Nodo *vector;  /* Vector dinámico de tripletas */
      struct Nodo *ultimo;  /* Última posición ocupada del vector de tripletas */
} TMatriz, *Matriz;
```

d-) **Vector en memoria dinámica.** Esta representación utiliza un vector de N * M posiciones para representar la matriz. Es muy parecida a la primera representación, pero se simplifica un poco el código de la constructora que pide la memoria para manejar los elementos. El elemento que se encuentra en la posición [fil, col] de la matriz, va a aparecer en la casilla (fil * M) + col del vector.



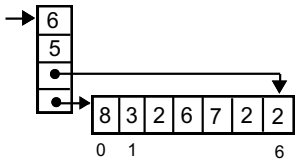
Este diseño tiene las siguientes declaraciones:

```
typedef struct
{     int N;
      int M;
      int *vector;
} TMatriz, *Matriz;
```

e-) **Vector de secuencias de valores.** Suponga que se quiere manejar una matriz de ceros y unos (i.e. imágenes en blanco y negro). En lugar de hacer una representación exhaustiva, es posible pensar en una representación compacta. Coloque en la posición 0 de un vector el número de ceros consecutivos de la matriz comenzando en la primera fila, y continuando con las filas siguientes. En la posición 1, el número de unos; en la posición 2, el número de ceros, y así consecutivamente hasta completar toda la información de la matriz. Por ejemplo, la matriz:

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	1	1	1	0
2	0	1	1	1	1	1
3	1	0	0	0	0	0
4	0	0	1	1	0	0

Se representaría con las siguientes estructuras de datos:

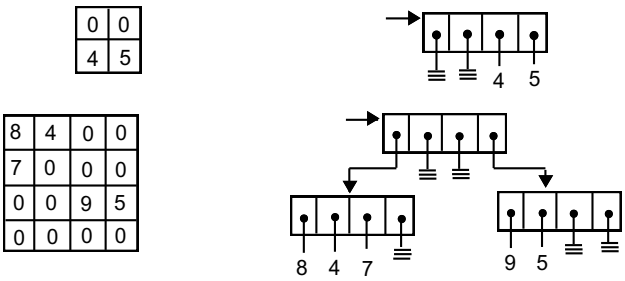


Esta representación es muy ineficiente para modificar el contenido de la matriz y para consultar su contenido, pero puede disminuir considerablemente el espacio ocupado en memoria.

Este diseño tiene las siguientes declaraciones:

```
typedef struct
{       int N;
        int M;
        int *ultimo;
        int *vector;
} TMatriz, *Matriz;
```

f-) ***Cuadtree.*** Un ***cuadtree*** es una estructura de datos que corresponde a una descomposición recursiva de una matriz, en bloques uniformes rectangulares. En la siguiente figura aparecen dos ejemplos que ilustran su uso. En el capítulo de estructuras recursivas de datos se profundiza en este tema. La idea es representar bloques de ceros de manera compacta. Esta representación se utiliza frecuentemente para representar imágenes digitalizadas.

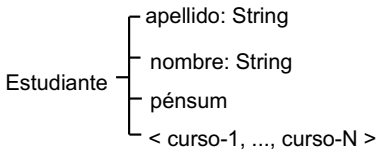


1.3.10. Un TAD como Estructura de Datos

Es usual que un objeto abstracto utilice como parte de sus estructuras de datos un elemento de otro TAD. Esta situación se presenta en dos casos: el primero, cuando en el modelaje se decide que parte del estado del objeto abstracto corresponde a otro elemento del mundo, más simple, pero con su propia estructura. El segundo caso se presenta cuando un TAD contenedor utiliza como estructura de datos otro TAD contenedor más general, como una manera de simplificar su desarrollo.

Ejemplo 1.24:

Considere el TAD Estudiante, definido mediante el siguiente formalismo:



Una de las características del objeto abstracto está dada por el pénsum de la carrera que se encuentra cursando el estudiante, y para el cual se debe hacer el diseño completo de un TAD. En este caso, las estructuras de datos incluirían uno de esos objetos:

```
typedef struct
{   String apellido;
    String nombre;
    Pensum pensum;              /* Objeto del TAD Pénsum */
    ListaCurso listaCursos;     /* Lista[ Curso ] listaCursos */
} TEstudiante, *Estudiante;
```

Las estructuras de datos también incluyen una lista de cursos (un TAD contenedor Lista y un TAD componente Curso) como parte del modelaje de un estudiante. Todas las operaciones del TAD Estudiante deben expresarse en términos de las operaciones de los TAD Pensum, ListaCurso y Curso. Por ejemplo, una operación que retorne el número de créditos que se encuentra cursando un estudiante, se reduce a utilizar una operación que retorne cada curso de la lista, y, a cada uno de ellos, invocarle la analizadora que informa el número de créditos que tiene. Algo del siguiente estilo:

```
int numCreditosEst( Estudiante est )
{   int i, acum;
    for( i = 1, acum = 0; i <= longLista( est->listaCursos ); i++ )
        acum += numCreditosCurso( infoLista ( est->listaCursos, i ) );
    return acum;
}
```

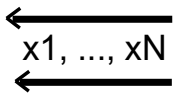
En esta rutina, las operaciones que utiliza de los otros TAD retornan los siguientes valores:

- longLista: número de elementos de la lista
- infoLista: i-ésimo elemento de la lista
- numCreditosCurso: número de créditos de un curso

⏎

Ejemplo 1.25:

Suponga que se quieren diseñar las estructuras de datos para el TAD Fila, una contenedora en la cual la entrada de nuevos elementos se hace por un extremo y la supresión, por el otro. El formalismo gráfico para representar una fila es:



Además de todos los diseños posibles de estructuras de datos, como vectores, apuntadores, etc., es posible utilizar un objeto de otro TAD contenedor, como sería el TAD Lista, para su representación interna. En este caso, el esquema de representación se debe definir entre los formalismos de los dos objetos abstractos. Esto es, explicar la manera de colocar cada uno de los componentes de un objeto del TAD Fila en un objeto del TAD Lista, permitiendo así almacenar convenientemente toda la información. El esquema se puede resumir gráficamente de la siguiente manera:

El formalismo explica que el elemento que se encuentra próximo a salir de la fila se debe colocar en la primera posición de la lista, y el último elemento en ingresar, debe colocarse al final de ella. Las

operaciones, por su parte, deben escribirse en términos de las operaciones del TAD de base, sin entrar en ningún momento a manipular directamente sus estructuras de datos. Esto implica, que la implementación del TAD Fila es completamente independiente de las estructuras de datos que utilice internamente una lista, teniendo la posibilidad de cambiarlas sin que haya que modificar su implementación.

1.3.11. Esquema de Persistencia

El **esquema de persistencia** de un TAD define la manera como se salva y lee de memoria secundaria la información asociada con un objeto abstracto: cada objeto es responsable de sobrevivir a la ejecución del programa que lo crea. La única diferencia con el esquema de representación interno, mediante estructuras de datos en memoria principal, es que se debe trabajar con todas las dificultades y restricciones que implica la administración de información en memoria secundaria, las cuales no se entran a detallar aquí.

Este tema de persistencia se trata únicamente de manera tangencial para mostrar la manera de relacionar el diseño de un tipo abstracto de dato con todo el problema de diseño de estructuras de datos en memoria secundaria (bases de datos, archivos, etc.). Los ejemplos que se utilizan a lo largo del libro corresponden a esquemas triviales, y no deben ser entendidos como la solución adecuada para *software* de alguna envergadura.

En la figura 1.8 se muestra la relación entre un objeto abstracto y sus estructuras de datos en memoria principal y en memoria secundaria. El esquema de persistencia debe cumplir la condición de validez con respecto al formalismo del objeto abstracto (v.g. ser capaz de representar completamente su estado interno), como lo hace el esquema de representación, y es de esta manera como se garantiza que se puedan implementar las operaciones de lectura y escritura de un objeto abstracto en memoria secundaria.

En algunos casos las operaciones de persistencia se pueden implementar sobre las demás operaciones del TAD, haciendo independientes ambos diseños, pero a veces, es más conveniente escribir directamente las operaciones de persistencia sobre las estructuras de datos en memoria principal.

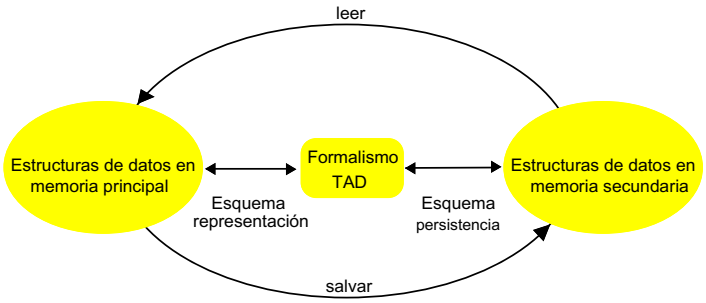
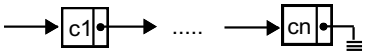


Fig. 1.8 - Esquema de persistencia en un TAD

Ejemplo 1.26:

Para el TAD String, un esquema posible de representación y de persistencia se puede resumir, de manera gráfica, de la siguiente manera:

Esquema de representación	Formalismo	Esquema de persistencia
	"c_1 c_2 ... c_n"	n c1 c2 .... cn

El esquema de persistencia indica la manera de colocar la información en un archivo. En este caso, se utiliza un archivo de texto, en cuyo primer renglón se coloca el número de caracteres que conforman la cadena, seguido de cada uno de los elementos. Cada caracter se coloca en un renglón aparte.

La implementación de las dos operaciones de persistencia del TAD sería:

```
String leerStr( FILE * fp )
{   int longit;
    char car;
    String st = inicStr( );
    fscanf( fp, "%d", &longit );                /* Lee del archivo el número de elementos */
    for( ; longit > 0; longit-- )
    {     fscanf( fp, "%c", &car );             /* Lee del archivo cada caracter de la cadena */
          anexarStr( st, car );                 /* Adiciona el nuevo caracter a las estructuras de datos */
    }
    return st;
}

void salvarStr( String st, FILE *fp )
{   int k, longit =  longitudStr( st );
    fprintf( fp, "%d\n", longit );              /* Escribe en el archivo el número de elementos */
    for( k = 1; k <= longit; k-- )
          fprintf( fp, "%c\n", infoStr( st, k ) );   /* Escribe el k-ésimo caracter en el archivo */
}
```

Si se quisiera escribir el código de estas rutinas, directamente sobre las estructuras de datos, en lugar de las llamadas a las operaciones anexarStr o longitudStr, se deberían manipular los apuntadores y los nodos que representan la cadena en memoria principal.

Ejercicios Propuestos

1.26. Diseñe diferentes esquemas de representación y persistencia para el TAD String y haga una tabla para compararlos.

1.27. Diseñe diferentes esquemas de representación y persistencia para el TAD Polinomio y haga una tabla para compararlos.

1.28. Diseñe diferentes esquemas de representación y persistencia para el TAD Polígono y haga una tabla para compararlos.

1.29. Diseñe diferentes esquemas de representación y persistencia para el TAD Lista y haga una tabla para compararlos.

1.30. Diseñe diferentes esquemas de representación y persistencia para el TAD Pila y haga una tabla para compararlos.

1.31. Diseñe diferentes esquemas de representación y persistencia para el TAD Fila y haga una tabla para compararlos.

1.32. Diseñe diferentes esquemas de representación y persistencia para el TAD SuperEntero y haga una tabla para compararlos.

1.33. Diseñe diferentes esquemas de representación y persistencia para el TAD ColaPrioridad y haga una tabla para compararlos.

1.34. Diseñe diferentes esquemas de representación y persistencia para el TAD Bicola y haga una tabla para compararlos.

1.35. Diseñe diferentes esquemas de representación y persistencia para el TAD Diccionario y haga una tabla para compararlos.

1.36. Diseñe diferentes esquemas de representación y persistencia para el TAD Texto y haga una tabla para compararlos.

1.37. Diseñe diferentes esquemas de representación y persistencia para el TAD TablaAsociacion y haga una tabla para compararlos.

1.38. Diseñe diferentes esquemas de representación y persistencia para el TAD Bolsa y haga una tabla para compararlos.

1.39. Diseñe diferentes esquemas de representación y persistencia para el TAD DirectorioTelefonico y haga una tabla para compararlos.

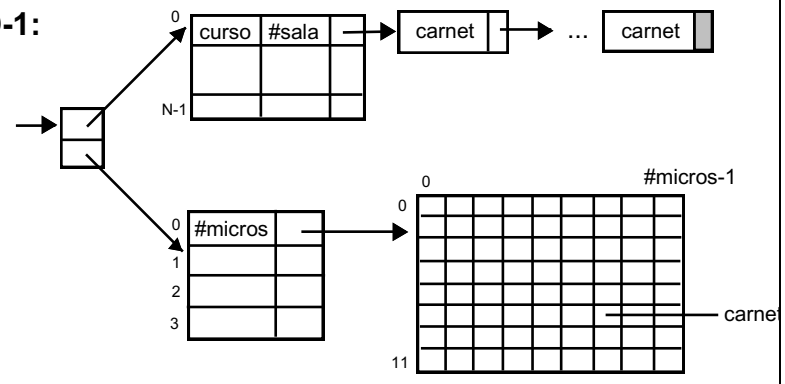
1.40. 🕒 En las salas de micros de la Facultad se piensa desarrollar un sistema informático para el manejo de los turnos. En la actualidad hay 4 salas, cada una con un número variable de micros. Las salas funcionan de 7 a.m. a 7 p.m. Los micros de cada sala se encuentran numerados de 0 en adelante. Cada curso tiene asignada una sala y sólo los estudiantes inscritos pueden trabajar en ellas. En el momento de reservar un turno, el sistema debe verificar que el carnet del estudiante haga parte de un curso válido. Un estudiante debe pedir turno el mismo día en que piensa trabajar.

Las operaciones importantes del sistema son:

- Reservar un turno para un estudiante
- Cancelar una reserva de un turno de un estudiante: dados el carnet del estudiante y la hora, el sistema anula dicho turno.
- Consultar un turno a una hora dada: dados el carnet de un estudiante y la hora, el sistema informa la sala y el micro en el cual el estudiante tiene turno.

Para el desarrollo del *software*, se contrataron 4 diseños de estructuras de datos, los cuales se denominan D-1 hasta D-4, y se presentan a continuación. Todos los diseños son válidos, en el sentido de que modelan toda la información importante del problema:

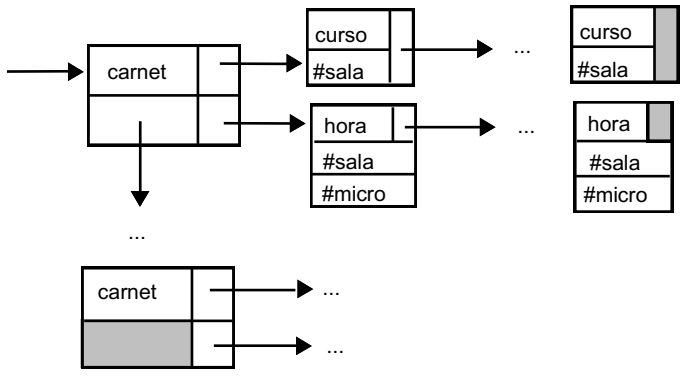
D-1:



Las salas están representadas por dos vectores: uno con los cursos, cuyo final viene marcado por un curso de código 0, y otro de 4 posiciones, con las planillas de turnos de cada una de las salas.

```
typedef struct
{     Curso cursos[ N ];        /* Información de cada uno de los cursos. Final marcado con código 0 */
      Sala salas[ 4 ];          /* Información de cada una de las salas */
} *Salas;

struct Nodo
{     int carnet;               /* Carnet del estudiante */
      struct Nodo *sig;         /* Siguiente nodo en el encadenamiento */
};

typedef struct
{     int codigo;               /* Código del curso */
      int Nsala;                /* Número de sala asignada al curso: 0..3 */
      struct Nodo *est;         /* Lista encadenada con los carnets de los estudiantes del curso */
} Curso;

typedef struct
{     int Nmicros;              /* Número de micros en la respectiva sala */
      int **turno;              /* Matriz de dimensión [ 0..11, 0.. Nmicros-1 ]. 0 = 7 am, 11= 7pm. */
                                /* Cada casilla tiene un carnet si el turno está asignado o 0 */
                                /* El espacio para la matriz lo pide la constructora según el número de micros */
                                /* Este campo se maneja como cualquier matriz: turno[ i ][ j ] */
} Sala;
```

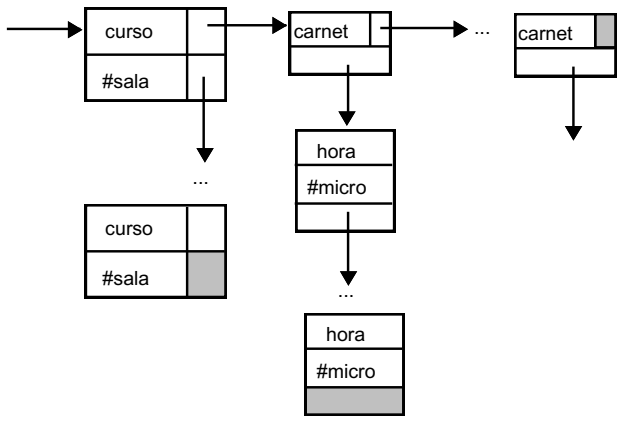
D-2:



La estructura de datos se basa en una lista ordenada, con los carnets de los estudiantes, en la cual se referencian los cursos que cada uno se encuentra cursando y los turnos que tiene reservados.

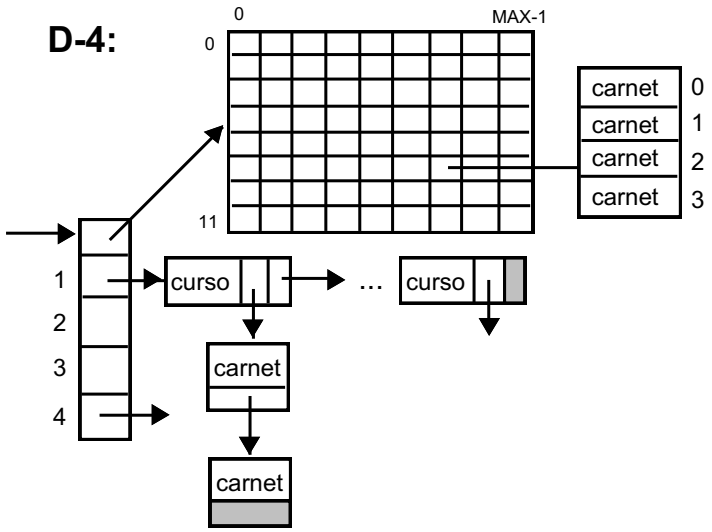
```
struct curso
{     int codigo;              /* Código del curso */
      int sala;                /* Sala asignada al curso: 0..3 */
      struct curso *sig;       /* Siguiente curso en la lista */
};

struct turno
{     int hora;                /* Hora del turno: 0..11→ 0 = 7 am,..., 11 = 7pm */
      int sala;                /* Sala en la cual tiene el turno reservado */
      int Nmicro:              /* Número del micro que tiene reservado */
      struct turno *sig;       /* Siguiente turno en la lista */
};

typedef struct Nodo
{     int carnet;              /* Carnet del estudiante. La lista está ordenada ascendentemente */
      struct curso *cursos:    /* Lista de cursos que está tomando el estudiante */
      struct turno *turnos;    /* Lista de turnos reservados por el estudiante */
      struct Nodo *sig;        /* Siguiente estudiante en la lista */
} *Salas;
```

D-3:



La estructura de datos se basa en una lista de cursos, cada uno de los cuales tiene los estudiantes en él inscritos. Por su parte, cada estudiante tiene los turnos pedidos para ese curso.

```
struct Turno
{     int hora;                /* Hora del turno: 0..11→ 0 = 7 am,..., 11 = 7 pm */
      int micro;               /* Número de micro asignado */
      struct Turno *sig;       /* Siguiente turno del estudiante */
};

struct Estud
{     int carnet;              /* Carnet del estudiante */
      struct Turno *turnos;    /* Turnos del estudiante */
      struct Estud *sig;       /* Siguiente estudiante en la lista */
};

typedef  struct Nodo
{     int codigo;              /* Código del curso */
      int Nsala;               /* Número de sala asignada al curso */
      struct Estud *est;       /* Lista de los carnets de los estudiantes que están inscritos en el curso */
      struct Nodo *sig;        /* Siguiente curso en la lista */
} *Salas;
```



Compuesta por una planilla de turnos para las 4 salas, y por 4 listas de cursos, una asociada con cada sala de micros. La planilla tiene una fila por cada hora y MAX columnas, que corresponde al número máximo de micros que puede haber en una sala. En cada casilla de la planilla hay 4 espacios, para colocar los carnets de los estudiantes que tienen turno en cada sala. Cuando una sala tiene menos de MAX micros, se marca con -1 en la planilla. Si el turno no ha sido asignado, se marca con 0.

```
typedef Turno int[ 4 ];          /* Carnet de los estudiantes que tienen turno en cada una de las salas */
                                 /* 0 = micro libre , -1 = micro inexistente */
struct Estud
{     int carnet;                /* Carnet del estudiante */
      struct Estud *sig;         /* Siguiente estudiante en la lista */
};

struct Nodo
{     int código;                /* Código del curso */
      struct Estud *est;         /* Lista de estudiantes en el curso */
      struct Nodo *sig;          /* Siguiente curso en la lista */
};

typedef  struct
{     Turno turnos[12][MAX];     /* Matriz de turnos */
      struct Nodo *sala1;        /* Lista de cursos asignados a la sala 1 */
      struct Nodo *sala2;        /* Lista de cursos asignados a la sala 2 */
      struct Nodo *sala3;        /* Lista de cursos asignados a la sala 3*/
      struct Nodo *sala4;        /* Lista de cursos asignados a la sala 4 */
} *Salas;
```

a-) Calcule y explique la complejidad de las operaciones reservar, cancelar y consultar, descritas anteriormente, para cada diseño propuesto de estructuras de datos.

b-) Para cada uno de los diseños, desarrolle la operación para dar un turno a un estudiante, dado el número de carnet (int), la hora (int: 7 h - 18 h) y el código del curso al cual pertenece (int). La función debe retornar un entero que indique si fue posible asignar el turno pedido de la siguiente manera: -1, indica que no fue posible darle un turno, cualquier otro número, indica el número de micro que le fue asignado para el turno pedido.

c-) Compare los diseños propuestos y escoja el mejor, justificando claramente su elección.

1.4. Implementación de las Operaciones de un TAD

En esta sección se estudia la segunda etapa de la implementación de un Tipo Abstracto. Cuando se llega a este punto, ya se han escogido las estructuras de datos más adecuadas y se va a hacer la implementación de las operaciones.

1.4.1. Esquema de Implementación en C

Para cada operación del TAD se debe escribir una función en C que simule su comportamiento sobre las estructuras de datos seleccionadas. Es necesario ceñirse a un estándar de implementación, de manera que en el momento de hacer el mantenimiento del *software*, se tenga una idea clara de los problemas que puede producir un cambio en el código.

Un TAD se implementa en dos archivos:

- <nombre>. h: Se denomina el **archivo de encabezado** del TAD, y contiene una descripción del objeto abstracto, la declaración de las estructuras de datos y el prototipo de cada una de las operaciones, ordenadas por tipo de operación. Incluye también las constantes con códigos de error. Este archivo lo debe incluir todo programa que utilice el TAD.

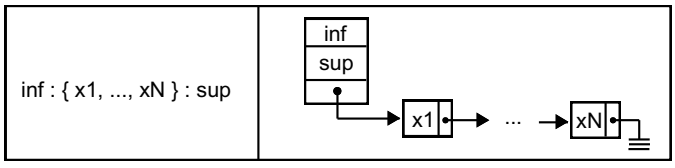
  Para evitar que el archivo de encabezado de un TAD sea incluido más de una vez durante una compilación, se debe asociar con él una constante, de tal manera que cuando se incluya el archivo por primera vez, quede definida, y evite que se vuelvan a incluir las declaraciones. El nombre de la constante se forma a partir del nombre del archivo de encabezado. Esto se hace utilizando las facilidades de compilación condicional que da C:

```
#ifndef __ARCH_H
#define __ARCH_H

<declaraciones del TAD>

#endif
```

- <nombre>.c: Contiene las rutinas que implementan las operaciones del TAD. Para aprovechar las facilidades de compilación independiente que tiene C, cada uno de estos archivos se debe incluir en el proyecto (*project* o *makefile*), para que el compilador y el encadenador lo consideren como parte del *software*.

Ejemplo 1.27:

Para implementar el TAD Conjunto de enteros en un rango, se construyen los archivos conj.h y conj.c, con todas las declaraciones del TAD y la implementación de las operaciones respectivamente. Las estructuras de datos seleccionadas en este caso corresponden a las ilustradas en el siguiente dibujo:



A continuación se presentan los dos archivos que implementan el TAD Conjunto. Como el objetivo del ejemplo es ilustrar la estructura de los mismos y no su contenido exacto, se eliminó toda la documentación de las operaciones, la cual se debe tomar del diseño (comentarios, precondición y postcondición, etc.)

Archivo: conj.h:

```
#ifndef __CONJ_H
#define __CONJ_H
/*-----------------------------------------------------------------------------------------------------*/
/* TAD: Conjunto                                                                                       */
/* Objeto abstracto: Conjunto de enteros en un rango definido                                          */
/* Estructuras de datos: Lista sencillamente encadenada de enteros                                     */
/* Manejo de error: Responsabilidad del cliente                                                        */
/*-----------------------------------------------------------------------------------------------------*/
/* ESTRUCTURAS DE DATOS                                                                                */
/*-----------------------------------------------------------------------------------------------------*/

struct NodoConj
{   int elem;                   /* Información en el nodo */
    struct NodoConj *sig;       /* Encadenamiento al siguiente */
};

typedef struct
{   int inf, sup;               /* Rango de valores del conjunto */
    struct NodoConj *primero;   /* Cabeza de la lista de valores */
} TConjunto, *Conjunto;

/*-----------------------------------------------------------------------------------------------------*/
/*  OPERACIONES DEL TAD                                                                                */
/*-----------------------------------------------------------------------------------------------------*/

/* CONSTRUCTORA */
Conjunto crearConj( int infer, int super );

/* MODIFICADORAS */
void insertarConj( Conjunto conj, int elem );
void eliminarConj( Conjunto conj, int elem );

/* ANALIZADORAS */
int estaConj( Conjunto conj, int elem );
int inferiorConj( Conjunto conj );
int superiorConj( Conjunto conj );
#endif
```

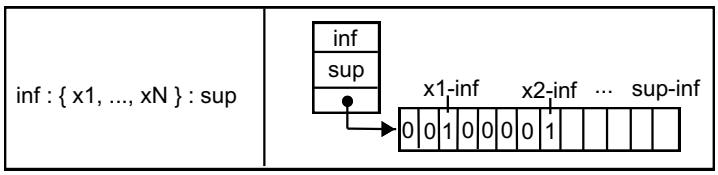
Archivo: conj.c:

```
#include "conj.h"
#define NULL 0

Conjunto crearConj( int infer, int super )
{   Conjunto pconj = ( Conjunto )malloc( sizeof( TConjunto ) );
    pconj->primero = NULL;
    pconj->inf = infer;
    pconj->sup = super;
    return pconj;
}
```

```
/*---------------------------------------------------------------------------------- */

void insertarConj( Conjunto conj, int elem )
{   struct NodoConj *p = ( struct NodoConj * )malloc( sizeof( struct NodoConj ) );
    p->elem = elem;
    p->sig = conj->primero;
    conj->primero = p;
}

/*---------------------------------------------------------------------------------- */

void eliminarConj( Conjunto conj, int elem )
{   struct NodoConj *q, *p = conj->primero;
    if( p->elem == elem )
    {     conj->primero = p->sig;
          free( p );
    }
    else
    {     for( ; p->sig != NULL && p->sig->elem != elem; p = p->sig );
          if( p->sig != NULL )
          {     q = p->sig;
                p->sig = q->sig;
                free( q );
          }
    }
}

/*---------------------------------------------------------------------------------- */

int estaConj( Conjunto conj, int elem )
{   struct NodoConj *p;
    for( p = conj->primero; p != NULL && p->elem  != elem; p = p->sig );
    return p != NULL;
}

/*---------------------------------------------------------------------------------- */

int inferiorConj( Conjunto conj )
{    return conj->inf;
}

/*---------------------------------------------------------------------------------- */

int superiorConj( Conjunto conj )
{    return conj->sup;
}
```

Ejemplo 1.28:

En este ejemplo se muestra otra implementación del TAD Conjunto. Fíjese que el archivo de encabezado es igual para ambas implementaciones, salvo por la sección de estructuras de datos. Esto hace que dos implementaciones de un mismo TAD se puedan intercambiar al interior de un programa sin ninguna dificultad. Si en algún momento, a causa de la evolución del *software*, se necesita pasar todo el manejo que se hacía con vectores a apuntadores, basta con remplazar los dos archivos del TAD respectivo y no habrá ningún problema con el funcionamiento del resto del programa, porque nadie podrá darse cuenta del cambio.



Archivo: conj.h:

```
#ifndef __CONJ_H
#define __CONJ_H

/*------------------------------------------------------------------------------------------------ */
/* TAD: Conjunto                                                                                   */
/* Objeto abstracto: Conjunto de enteros en un rango definido                                      */
/* Estructuras de datos: Vector de booleanos                                                       */
/* Manejo de error: Responsabilidad del cliente                                                    */
/*------------------------------------------------------------------------------------------------*/
/*                    ESTRUCTURAS DE DATOS                                                         */
/*------------------------------------------------------------------------------------------------*/

typedef struct
{   int inf, sup;                   /* Rango de valores del conjunto */
    int *info;                      /* Vector de booleanos */
} TConjunto, *Conjunto;

/*------------------------------------------------------------------------------------------------*/
/*                    OPERACIONES DEL TAD                                                          */
/*------------------------------------------------------------------------------------------------*/

/* CONSTRUCTORA */
Conjunto crear( int infer, int super );

/* MODIFICADORAS */
void insertar( Conjunto conj, int elem );
void eliminar( Conjunto conj, int elem );

/* ANALIZADORAS */
int esta( Conjunto conj, int elem );
int inferior( Conjunto conj );
int superior( Conjunto conj );
#endif
```

Archivo: conj.c:

```
#include "conj.h"

Conjunto crear( int infer, int super )
{    int i;
     Conjunto pconj = ( Conjunto )malloc( sizeof( TConjunto ) );
     pconj->info = ( int * )calloc( super - infer + 1,sizeof( int ) );
     for( i = 0; i < super - infer + 1; i++ )
             pconj->info[ i ] = 0;
     pconj->inf = infer;
      pconj->sup = super;
      return pconj;
}

/*-----------------------------------------------------------------------------------------*/

void insertar( Conjunto conj, int elem )
{    conj->info[ elem - conj->inf ] = 1;
}

/*-----------------------------------------------------------------------------------------*/

void eliminar( Conjunto conj, int elem )
{    conj->info[ elem - conj->inf ] = 0;
}

/*-----------------------------------------------------------------------------------------*/

int esta( Conjunto conj, int elem )
{    return conj->info[ elem - conj->inf ] == 1;
}

/*-----------------------------------------------------------------------------------------*/

int inferior( Conjunto conj )
{    return conj->inf;
}

/*-----------------------------------------------------------------------------------------*/

int superior( Conjunto conj )
{    return conj->sup;
}
```

⏎

1.4.2. Documentación

En el archivo de encabezado de un TAD es conveniente agregar la siguiente información:

- Información administrativa de desarrollo: Nombre, autor, fecha de creación, fecha de la última modificación, versión, etc.

- Descripción detallada del objeto abstracto que se maneja.
- Declaración de las estructuras de datos y la relación que guardan con el objeto abstracto.
- Restricciones de implementación del TAD.
- Prototipo de las operaciones, clasificadas por tipo: constructoras, modificadoras, analizadoras, persistencia, Entrada/Salida.
- Complejidad de cada una de las operaciones.

La indentación y la documentación de las rutinas dependen del estándar que utilice cada desarrollador. Sobre eso se debe consultar cualquier libro de programación en C.

1.4.3. Implementación de la Genericidad

Cuando se va a hacer la implementación de un TAD genérico, parametrizado por algún tipo de dato, existen varias opciones:

- Utilizar un lenguaje como C++ que maneja clases paramétricas. Es la solución ideal.
- El esquema más sencillo es definir el tipo de dato mediante un typedef, de manera que el TAD quede planteado en términos generales y solo en el momento de usarlo se define el tipo específico que se quiere manejar. Es una aproximación muy restrictiva, puesto que únicamente permite utilizar una vez en un programa un TAD.
- Seguir el esquema anterior, pero aprovechar el mecanismo de macros de C, de tal forma que sea posible generar un TAD distinto para cada tipo de dato que se quiera manejar.
- En lugar de manipular objetos de un TAD es posible manejar sus apuntadores. La idea es definir una contenedora que contenga apuntadores a void, de tal forma que, en últimas, pueda apuntar a objetos de cualquier TAD. Este enfoque es muy práctico, pero el código resultante es oscuro.

1.4.4. Probador Interactivo de un TAD

Durante el proceso de desarrollo y pruebas del *software*, es conveniente probar por separado cada uno de los TAD. Por eso, es necesario desarrollar un programa de prueba que presente un menú con todas las operaciones posibles, y a través del cual el programador pueda validar su implementación. Este programa debe quedar como un anexo del *software*, de tal forma que en la etapa de mantenimiento se cuente con él como una herramienta.

Ejemplo 1.29:

Programa de prueba para el TAD Conjunto:

```
#include "conj.h"

void main( void )
{   int com, elem, i;
    Conjunto conj;
    conj = crear( 1, 100 );
    while( 1 )
    {       /* Presenta el menú de opciones */
            printf( "0. Salir\n" );
            printf( "1. Adicionar un elemento\n" );
            printf( "2. Eliminar un elemento\n" );
```

```
        scanf( "%d", &com );
        switch( com )
        {   case 0:     /* Salida del programa */
                return;
            case 1:     /* Operación de inserción */
                printf( "Elemento: " );
                scanf( "%d", &elem );
                insertar( conj, elem );
                break;
            case 2:     /* Operación de supresión */
                printf( "Elemento: " );
                scanf( "%d", &elem );
                eliminar( conj, elem );
                break;
        }

    /* Presenta por pantalla el conjunto */
    printf( "\n{ " );
    for( i = inferior( conj ); i <= superior( conj ); i++ )
        if( esta( conj, i ) )
            printf( "%d ", i );
    printf( "}\n" );
    }
}
```

Bibliografía

- [BER94] Bergin, J., "Data Abstraction: The Object-Oriented Approach Using C++", McGraw-Hill, 1994.
- [CAR91] Cardoso, R., "Verificación y Desarrollo de Programas", Ediciones Uniandes, 1991.
- [ESA89] Esakov, J., "Data Structures: An Advanced Approack Using C", Prentice-Hall, 1989.
- [FEL88] Feldman, M., "Data Structures with Modula-2", Prentice-Hall, 1988.
- [GUT77] Guttag, J., "Abstract Data Types and the Development of Data Structures", Comm. ACM, Vol. 20, No. 6, Junio 1977.
- [GUT78] Guttag, J., "Abstract Data Types and Software Validation", Comm. ACM, vol. 21, No. 12, Diciembre 1978.
- [JAC93] Jacobson, I., "Object-Oriented Software Engineering", Addison-Wesley, 1993.
- [LIS86] Liskov, B., Guttag, J., "Abstraction and Specification in Program Development", The MIT Press, McGraw-Hill, 1986.
- [MAR93] Martin, J., "Principles of Object-Oriented Analysis and Design", Prentice-Hall, 1993.
- [MAR86] Martin, J., "Data Types and Data Structures", Prentice-Hall, 1986.
- [VIL83] Villalobos, J., "Análisis y Diseño Orientados por Objetos", Depto. de Ingeniería de Sistemas, Universidad de los Andes, 1993.

CAPITULO 2

ESTRUCTURAS LINEALES: LISTAS

Este capítulo trata el tema de listas, estructuras de datos de uso muy común en los programas que manejan agrupamientos de elementos. Presenta también una forma de administrar la noción de orden al interior de las listas, para poder, así, manejar secuencias ordenadas de valores. Por último, ilustra la forma de implementar otros TAD sobre el TAD Lista, con el fin de ganar un mayor nivel de independencia.

2.1. Definiciones y Conceptos Básicos

Se define una **lista** como una secuencia de cero o más elementos de un mismo tipo. El formalismo escogido para representar este tipo de objeto abstracto es:

$$< e_1, e_2, \ldots, e_n >$$

Cada e_i modela un elemento del agrupamiento. Así, e_1 es el **primero** de la lista, e_n es el **último** y la lista formada por los elementos $< e_2, e_3, \ldots, e_n >$ corresponde al **resto** de la lista inicial. La **longitud** de una lista se define como el número de elementos que la componen. Si la lista no tiene ningún elemento la lista se encuentra **vacía** y su longitud es 0. Esta estructura sin elementos se representa mediante la notación < >, y se considera, simplemente, como un caso particular de una lista con cero elementos.

La **posición** de un elemento dentro de una lista es el lugar ocupado por dicho elemento dentro de la secuencia de valores que componen la estructura. Es posible referirse sin riesgo de ambigüedad al elemento que ocupa la i-ésima posición dentro de la lista, y hacerlo explícito en la representación mediante la notación:

$$< e1^{1}, e2^{2}, ..., ei^{i}, ..., en^{n} >$$

Esta indica que e_i es el elemento que se encuentra en la posición i de la lista y que dicha lista consta de n elementos. Esta extensión del formalismo sólo se utiliza cuando se quiere hacer referencia a la relación entre un elemento y su posición. Para efectos prácticos, una posición es un entero positivo, menor o igual al número total de elementos de una lista.

El **sucesor** de un elemento dentro de una lista es aquél que ocupa la siguiente posición. Por esta razón, el único elemento de una lista que no tiene sucesor es el último. De la misma manera, cada elemento de una lista, con excepción del primero, tiene un **antecesor**, correspondiente al elemento que ocupa la posición anterior.

Un proceso muy importante en las estructuras contenedoras de datos es su **recorrido**. Este consiste en pasar exactamente una vez sobre cada uno de sus elementos, ya sea para encontrar algún elemento en particular o

para establecer alguna característica de la estructura (el número de elementos, por ejemplo). En las estructuras lineales este proceso suele hacerse moviéndose en orden de posición.

Ejemplo 2.1:

Para la lista representada mediante el objeto abstracto < 1, 3, 5, 7, 11, 13, 17, 19 >, se tiene que:

- 1 es el primer elemento, 19 es el último y la lista < 3, 5, 7, 11, 13, 17, 19 > es el resto
- La longitud de la lista es 8
- La posición del elemento 13 es 6
- El sucesor de 11 es 13 y el sucesor de 5 es 7
- El antecesor de 19 es 17

⏎

A continuación se dan algunas definiciones importantes sobre listas, las cuales se ilustran en el ejemplo 2.2. Otras definiciones se presentan como parte de los ejercicios propuestos.

- Dos listas lst1 y lst2 son **iguales** si ambas estructuras tienen el mismo número de componentes y, además, sus elementos son iguales uno a uno. En particular, dos listas vacías son iguales.

- Dos listas lst1 y lst2 son **semejantes** si tienen los mismos elementos aunque estén en diferente orden. Si existe un elemento repetido en lst1, debe aparecer el mismo número de veces en lst2.

- Una lista lst2 es una **sublista** de una lista lst1 si todos los elementos de lst2 se encuentran en lst1, consecutivos y en el mismo orden (también se puede decir que lst2 **ocurre** en lst1). En particular, una lista vacía ocurre en cualquier otra lista, y una lista es sublista de sí misma.

- Una lista lst2 está **contenida** en una lista lst1, si todos los elementos de lst2 están en lst1, aunque sea en diferente orden. Esta es una noción de contenencia parecida a la de conjuntos.

- Una lista lst es **ordenada** si los elementos contenidos respetan una relación de orden $\leq$ definida sobre ellos, de acuerdo con la posición que ocupan (i.e. $e_i \leq e_k \ \forall\ i \leq k$). Esto es equivalente a afirmar que la lista lst = < e_1, e_2, …, e_n > es ordenada, si $e_i \leq e_{i+1} \ \forall\ i \mid 1 \leq i < n$.

Ejemplo 2.2:

Para las listas representadas mediante los siguientes objetos abstractos:

lst1 = < 1, 3, 5, 7, 9 >

lst2 = < 1, 2, 3, 4, 5, 6, 7, 8, 9 >

lst3 = < 5, 6, 7 >

lst4 = < 5, 6, 7 >

lst5 = < 6, 5, 7 >

Se cumple que:

- lst3 es igual a lst4
- lst3 es semejante a lst4 y a lst5
- lst1, lst2, lst3 y lst4 son listas ordenadas, con la relación de orden $\leq$ definida sobre los números naturales.
- lst3 es sublista de lst2
- lst1 está contenida en lst2

⏎

2.2. El TAD Lista

Existen muchos diseños posibles de un TAD para manejar los objetos abstractos descritos en la sección anterior. Como parte de los ejercicios se sugieren otros conjuntos de operaciones diferentes al presentado en esta parte. Aquí, se aprovecha la oportunidad para ilustrar la manera de administrar la noción de estado en un TAD contenedor. Este diseño va a permitir un uso muy eficiente de las listas en los diferentes problemas en que se usen.

Se define la **ventana** de una lista como el lugar de la secuencia sobre el cual se van a realizar las operaciones que se apliquen al objeto abstracto. De cierta manera, se puede ver como el único punto de la lista visible al usuario. Esto implica que el TAD, además de las operaciones usuales, debe tener operaciones para mover la ventana, de tal manera que se pueda colocar sobre cualquier elemento de la lista y afectarlo de la manera deseada. El formalismo extendido es el siguiente:

$$< \overset{1}{e1}, \overset{2}{e2}, ..., \boxed{\overset{i}{ei}}, ..., \overset{n}{en} >$$

En él se hace explícito que la ventana se encuentra sobre un elemento dado de la lista (el i-ésimo). Un caso particular es la lista vacía, en la cual la ventana se encuentra **indefinida**, ya que no puede estar situada sobre ningún elemento. Este estado posible del objeto abstracto se representa así:

$$< > \square$$

Otra situación posible de la ventana es que no esté situada sobre ningún elemento, aunque la lista no esté vacía. También en este caso se dice que la ventana está **indefinida** y se denota de la siguiente manera:

$$< \overset{1}{e1}, \overset{2}{e2}, ..., \overset{i}{ei}, ..., \overset{n}{en} > \square$$

La definición del TAD Lista se presenta a continuación, siguiendo el esquema de especificación expuesto en el capítulo anterior, y dejando todo el manejo de error como responsabilidad del usuario:

TAD Lista[TipoL]		
$< \overset{1}{e1}, \overset{2}{e2}, ..., \boxed{\overset{i}{ei}}, ..., \overset{n}{en} >$		
{ inv: $n \geq 0$, e_i pertenece a TipoL }		
Constructoras:		
• inicLista:		→ Lista
Modificadoras:		
• anxLista:	Lista x TipoL	→ Lista
• insLista:	Lista x TipoL	→ Lista
• elimLista:	Lista	→ Lista
• primLista:	Lista	→ Lista
• ultLista:	Lista	→ Lista
• sigLista:	Lista	→ Lista
• posLista:	Lista x int	→ Lista

Analizadoras:

- infoLista: Lista → TipoL
- longLista: Lista → int
- finLista: Lista → int

```
Lista inicLista( void )
/* Crea y retorna una lista vacía */

{ post: inicLista = < >□ }
```

```
void anxLista( Lista lst, TipoL elem )
/* Agrega un elemento después de la ventana */

{ pre: lst = LST, lst = < >□ ∨ lst = < x1, ..., [xi] , ... , xn > }
{ post: ( LST = < >□, lst = < [elem] >) ∨ ( LST = < x1, ..., [xi] , ... , xn >, lst = < x1, ..., xi, [elem] , ... , xn > ) }
```

```
void insLista( Lista lst, TipoL elem )
/* Agrega un elemento antes de la ventana */

{ pre: lst = LST, lst = < >□ ∨ lst = < x1, ..., [xi] , ... , xn > }
{ post: ( LST = < >□, lst = < [elem] > ) ∨ ( LST = < x1, ..., [xi] , ... , xn >, lst = < x1, ...,xi-1, [elem] , xi, ... , xn > ) }
```

```
void elimLista( Lista lst )
/* Elimina el elemento que se encuentra en la ventana */

{ pre: lst = LST = < x1, ..., [xi] , ... , xn > }
{ post: ( LST = < [x1] >, lst = < >□ ) ∨ ( LST = < x1, ..., [xn] >, lst = < x1, ..., xn-1 >□ ) ∨
        ( LST = < x1, ...,[xi] , ... , xn >, lst = < x1, ...,xi-1, [xi+1] , ... , xn > ) }
```

```
void primLista( Lista lst )
/* Coloca la ventana sobre el primer elemento de la lista */

{ pre: lst = LST }
{ post: ( LST = < >□, lst = < >□ ) ∨ ( LST = < x1, ..., xn >, lst = < [x1] , ..., xn > ) }
```

```
void ultLista( Lista lst )
/* Coloca la ventana sobre el último elemento de la lista */

{ pre: lst = LST }
{ post: ( LST = < >□, lst = < >□ ) ∨ ( LST = < x1, ..., xn >, lst =< x1, ..., [xn] > ) }
```

```
void sigLista( Lista lst )
/* Avanza la ventana una posición */

{ pre: lst = LST = < x1, ...,[xi] , ... , xn > }
{ post: ( LST = < x1, ...,[xi] , ... , xn >, lst = < x1, ..., xi, [xi+1] , ... , xn > ) ∨
        ( LST = < x1, ...,[xn] >, lst = < x1, ..., xn >□ ) }
```

```
void posLista( Lista lst, int pos )
/* Coloca la ventana sobre el pos-ésimo elemento de la lista */

{ post: ( pos < 1 ∨ pos > n, lst = < x1, ..., xn >□ ) ∨ ( lst = < x1, ...,[xpos] , ... , xn > ) }
```

```
TipoL infoLista( Lista lst )
/* Retorna el elemento de la ventana */

{ pre: lst = < x1, ...,[xi] , ... , xn > }
{ post: infoLista = xi }
```

```
int longLista( Lista lst )
/* Retorna el número de elementos de la lista */

{ post: longLista = n }
```

```
int finLista( Lista lst )
/* Informa si la ventana está indefinida */

{ post: ( lst = < x1, ..., xn >□, finLista = TRUE ) ∨ ( lst = < x1, ...,[xi] , ... , xn >, finLista = FALSE ) }
```

2.3. Ejemplos de Utilización del TAD

En esta sección se muestran algunos ejemplos sencillos de cómo utilizar las operaciones del TAD Lista, para desarrollar algoritmos que resuelvan problemas sobre este tipo de estructuras.

Ejemplo 2.3:

Imprimir el contenido de una lista de enteros. La complejidad de esta rutina es O(n), donde n es la longitud de la lista.

```
/* post: se han impreso todos los elementos de la lista */

void imprimirLista( Lista lst )
{ for( primLista( lst ); !finLista( lst ); sigLista( lst ) )
        printf( "%d ", infoLista( lst ) );          /* En el caso general, en lugar del printf debe ir la llamada */
}                                                   /* de la rutina que imprime un elemento de tipo TipoL */
```

Al comienzo del ciclo, la rutina sitúa la ventana de la lista sobre el primer elemento y, luego, la va colocando secuencialmente sobre cada uno de los demás elementos, hasta que la ventana queda indefinida. Cada vez que un elemento queda en la ventana, se manda imprimir. El invariante del ciclo es:

```
{ inv: se han impreso los elementos anteriores a la ventana }
```

Esta es la estructura típica de todas las rutinas que deben recorrer secuencialmente una lista.

Ejemplo 2.4:

Hacer una copia de una lista. La complejidad de esta rutina es O(n), donde n es la longitud de la lista.

```
/* post: copiarLista es una copia de lst */

Lista copiarLista( Lista lst )
{   Lista resp = inicLista( );
    for( primLista( lst ); !finLista( lst ); sigLista( lst ) )
        anxLista( resp, infoLista( lst ) );
    return resp;
}
```

La rutina recorre secuencialmente la lista lst, y, por cada elemento que encuentra, llama la operación de anexar sobre la lista de respuesta. El invariante del ciclo asegura que cuando la ventana de la lista lst se encuentre situada sobre el elemento x_i, en la lista de respuesta resp habrá una copia de los elementos x_1... x_{i-1}. Además, afirma que la ventana estará sobre el último elemento de la lista de respuesta:

{ inv: lst = < x_1,…, [xi] ,…, x_n >, resp = < x_1,…, [xi-1] > }

Ejemplo 2.5:

Concatenar dos listas lst1 y lst2, dejando el resultado en la primera de ellas. La complejidad de esta rutina es O(m), donde m es la longitud de la segunda lista.

```
/* pre: lst1 = < x1, …, xn >, lst2 = LST2 = < y1, …, ym > */
/* post: lst1 = < x1, …, xn, y1, …, ym >, lst2 = LST2 */

void concatLista( Lista lst1, Lista lst2 )
{   ultLista( lst1 );
    for( primLista( lst2 ); !finLista( lst2 ); sigLista( lst2 ) )
        anxLista( lst1, infoLista( lst2 ) );
}
```

Antes de comenzar el recorrido de la lista lst2, la rutina coloca la ventana de la lista lst1 sobre su último elemento, de tal forma que sea posible agregarle los elementos de lst2 a medida que se avanza sobre ella. El invariante asegura que cuando la ventana de la segunda lista se encuentre sobre el elemento y_i, todos los anteriores elementos (y_1, ..., y_{i-1}) ya habrán sido agregados adecuadamente a lst1.

{ inv: lst1 = < x_1 ,…, x_n, y_1, ..., [yi-1] >, lst2 = < y_1, …, [yi] , …, y_m > }

Ejemplo 2.6:

Invertir una lista, destruyendo la lista original. La complejidad de esta rutina es O(n), donde n es la longitud de la lista.

```
/* pre: lst = < x1,…,xn > */
/* post: invLista = < xn, …, x1 >, lst = < >□ */

Lista invLista( Lista lst )
{   Lista resp = inicLista( );
    for( primLista( lst ); !finLista( lst ); elimLista( lst ) )
        insLista( resp, infoLista( lst ) );
    return resp;
}
```

Esta rutina es muy parecida a la de copiar una lista, con la diferencia de que, en lugar de anexar los elementos al final, los va insertando antes del primero. El invariante afirma que en el momento en el cual el elemento x_i se encuentre en la primera posición de la lista, todos los anteriores elementos (x_1, ..., x_{i-1}) ya se habrán colocado en sentido inverso en la lista de respuesta.

{ inv: lst = < xi , …, x_n >, resp = < xi-1 , …, x_1 > }

⏎

Ejemplo 2.7:

Localizar el elemento elem en la lista lst. Si hay varias ocurrencias del elemento, deja la ventana en la primera de ellas. La complejidad de esta rutina es O(n), donde n es la longitud de la lista.

```
/* pre: lst = < x1, …, xn > */
/* post: ( lst = < x1 ,…, xi , …, xn >, xi = elem, xk != elem, ∀k < i ) ∨ ( lst = < x1, …, xn >□, xi != elem, ∀i ) */

void buscarLista( Lista lst, TipoL elem )
{   for( primLista( lst ); !finLista( lst ) && infoLista( lst ) != elem; sigLista( lst ) );
}
```

El ciclo de la rutina hace un recorrido completo de la lista, y termina cuando aparece el elemento que se está buscando, o cuando la lista se termina. El invariante es el siguiente:

{ inv: lst = < x1, …, xi , …, xn >, xk != elem, ∀k < i }

⏎

Ejemplo 2.8:

Eliminar todas las ocurrencias de un elemento en una lista. La complejidad de esta rutina es O(n), donde n es la longitud de la lista.

```
/* post: se han eliminado todas las ocurrencias de elem en lst */

void elimTodosLista( Lista lst, TipoL elem )
{   primLista( lst );
    while( !finLista( lst ) )
        if ( infoLista(lst) == elem )
            elimLista( lst );
        else
            sigLista( lst );
}
```

La rutina recorre la lista con un ciclo que avanza, algunas veces, eliminando el elemento de la ventana, y otras, colocándola sobre su sucesor, según se trate del elemento que se quiere eliminar o de otro. El invariante del ciclo es:

```
{ inv: se han eliminado las ocurrencias de elem anteriores a la ventana }
```
⏎

Ejemplo 2.9:

Decidir si dos listas son iguales. La complejidad de esta rutina es O(n), donde n es el mayor valor entre las longitudes de las listas. La razón de esto es que, en el peor de los casos, las listas tienen la misma longitud, y debe recorrerlas hasta el final para informar que son iguales. Dicho valor se puede acotar siempre con la mayor de las longitudes de las listas.

```
/* post: igualesListas = ( lst1 es igual a lst2 ) */

int igualesListas( Lista lst1, Lista lst2 )
{   if ( longLista( lst1 ) != longLista( lst2 ) )
        return FALSE;
    else
    {   primLista( lst1 );
        primLista( lst2 );
        while( !finLista( lst1 ) && infoLista( lst1 ) == infoLista( lst2 ) )
        {   sigLista( lst1 );
            sigLista( lst2 );
        }
        return finLista( lst1 );
    }
}
```

El caso en el cual las dos listas tienen diferente longitud se trata como una situación aparte, evitando que la rutina haga comparaciones inútiles entre los elementos. El ciclo avanza hasta que termina la primera lista (como son de la misma longitud, se podría utilizar también como condición de salida la finalización de la lista lst2), o hasta que encuentra un elemento en la posición i de las dos listas, que sea diferente. El invariante asegura que cuando la ventana de cada una de las listas esté sobre su i-ésimo elemento, todos los elementos hasta este punto habrán sido iguales uno a uno.

{ inv: lst1 = < x_1, …, xi, …, x_n >, lst2 = < y_1, …, yi , …, y_n >, $x_k = y_k$, $1 \le k < i$ }

⏎

Ejemplo 2.10:

En este ejemplo se presenta otra solución al problema de decidir si dos listas lst1 y lst2 son iguales. La complejidad de esta rutina depende básicamente de la complejidad de la operación posLista, que en la mayoría de implementaciones va a ser O(n), donde n es el número de elementos de la lista. Esto hace que la complejidad de la rutina completa varíe entre O(n) y O(n^2), según la implementación escogida. Por esa razón, se considera mejor la solución presentada en el ejemplo anterior, cuya eficiencia es independiente de la implementación.

```
/* post: igualesListas = ( lst1 es igual a lst2 ) */

int igualesListas( Lista lst1, Lista lst2 )
{   int i;
    if ( longLista( lst1 ) != longLista( lst2 ) )
        return FALSE;
    else
    {   for( i = 1; i <= longLista( lst1 ); i++ )
        {   posLista( lst1, i );
            posLista( lst2, i );
            if ( infoLista( lst1 ) != infoLista( lst2 ) )
                return FALSE;
        }
        return TRUE;
    }
}
```

La estructura de la rutina es similar a la utilizada en el ejemplo anterior, pero se utiliza un mecanismo diferente para avanzar la ventana sobre los elementos de las listas. Mientras en este caso se localiza haciendo una llamada a la operación posLista, antes, se hacía un barrido secuencial utilizando la operación sigLista. El invariante es idéntico al del ejemplo anterior.

{ inv: lst1 = < x_1, …, xi, …, x_n >, lst2 = < y_1, …, yi , …, y_n >, $x_k = y_k$, $1 \le k < i$ }

Ejemplo 2.11:

Indicar si una lista se encuentra ordenada. La complejidad de esta rutina es O(n), donde n es la longitud de la lista.

```
/* pre: lst = < x1,…, xn > */
/* post: ordenadaLista = ( x i ≤ xi+1 ) */

int ordenadaLista( Lista lst )
{   TipoL anterior;
    if ( longLista( lst ) == 0 )   return TRUE;
    else
    {   primLista( lst );
        for( anterior = infoLista( lst ), sigLista( lst ); !finLista( lst ) && infoLista( lst ) >= anterior; sigLista( lst ) )
            anterior = infoLista( lst );
        return finLista( lst );
    }
}
```

La rutina se basa en la idea de que cada elemento de la lista debe ser mayor que su antecesor. Así, es suficiente con recorrer la lista y llevar en una variable auxiliar el elemento anterior al de la ventana. Si en algún caso dicho elemento es mayor que el actual, la lista no es ordenada. El avance del ciclo se hace actualizando el valor de la variable temporal que lleva el anterior y moviendo la ventana. El invariante del ciclo es el siguiente:

{ inv: lst = < x_1,…, xi ,…, x_n >, *lst* es ordenada hasta x_{i-1}, anterior = x_{i-1} }

2.4. Otras Operaciones Interesantes

El TAD definido anteriormente se puede enriquecer con operaciones de manejo de persistencia y destrucción, de acuerdo con la siguiente especificación:

TAD Lista[TipoL]		
Destructora:		
• destruirLista:	Lista	
Persistencia:		
• cargarLista:	FILE *	→ Lista
• salvarLista:	Lista x FILE *	

```
void destruirLista( Lista lst )
/* Destruye el objeto abstracto, retornando toda la memoria ocupada por éste */

{ post: la lista lst no tiene memoria reservada }
```

```
Lista cargarLista( FILE *fp )
/* Construye una lista a partir de la información de un archivo */

{ pre: el archivo está abierto y es estructuralmente correcto, de acuerdo con el esquema de persistencia }
{ post: se ha construido la lista que corresponde a la imagen de la información del archivo }
```

```
void salvarLista( Lista lst, FILE *fp )
/* Salva la lista en un archivo */

{ pre: el archivo está abierto }
{ post: se ha hecho persistir la lista en el archivo, la ventana de la lista está indefinida }
```

Ejemplo 2.12:

Traer de memoria secundaria una lista, modificar su contenido eliminando todas las ocurrencias de un valor dado y hacer persistir de nuevo la lista resultante. La complejidad es O(n), donde n es la longitud de la lista. Se debe tener en cuenta que la constante asociada con la función cota es muy alta, dado el elevado costo en tiempo que tiene el acceso a la información en memoria secundaria.

```
/* pre: el archivo nombre existe y corresponde a una lista que ha persistido en memoria secundaria */
/* post:  en el archivo nombre ha persistido la lista inicial, sin ninguna ocurrencia del valor val*/
```

```
void actualizarLista( char nombre[ ], TipoL val )
{   FILE *fp = fopen( nombre, "r" );
    Lista lst = cargarLista( fp );
    fclose( fp );
    elimTodosLista( lst, val );
    fp = fopen( nombre, "w" );
    salvarLista( lst, fp );
    fclose( fp );
    destruirLista( lst );
}
```

Para cada uno de los objetos abstractos temporales que se utilicen en cualquier función, es necesario llamar la respectiva operación de destrucción, de tal forma que se recupere la memoria que se reservó para su almacenamiento. Si no se hace esto, se estará desperdiciando espacio en memoria por cada llamada de la función. Esto sucede porque los objetos abstractos se localizan siempre en memoria dinámica, y ésta no se recupera automáticamente al terminar la rutina que la reservó.

⏎

Ejemplo 2.13:

Las operaciones de persistencia reciben como parámetro un archivo abierto, para permitir que más de un objeto abstracto se pueda salvar bajo el mismo nombre. Es válido, por ejemplo, lo que se muestra en las siguientes rutinas:

```
void salvarListas( char nombre[ ], Lista l1, Lista l2, Lista l3 )
{   FILE *fp = fopen( nombre, "w" );
    salvarLista( l1, fp );
    salvarLista( l2, fp );
    salvarLista( l3, fp );
    fclose( fp );
}

void cargarListas( char nombre[ ], Lista *l1, Lista *l2, Lista *l3 )
{   FILE *fp = fopen( nombre, "r" );
    *l1 = cargarLista( fp );
    *l2 = cargarLista( fp );
    *l3 = cargarLista( fp );
    fclose( fp );
}
```

La operación que lee una lista de un archivo toma únicamente la información asociada con la lista (según el esquema de persistencia del TAD), y deja el archivo listo para que el siguiente objeto abstracto allí almacenado se pueda recuperar.

⏎

Adicional a estas tres nuevas funciones del TAD, es conveniente agregar una operación que permita guardar en una variable auxiliar la ventana de una lista, de manera que sea posible restaurarla de manera eficiente, sin necesidad de utilizar la operación posLista, la cual puede tener una complejidad lineal en algunas implementaciones. Esta variable auxiliar pertenece a un tipo interno del TAD llamado Ventana. La siguiente es la especificación de la pareja de operaciones que permite guardar y recuperar la ventana de una lista. En el ejemplo 2.14 se ilustra el uso de esta nueva facilidad.

Analizadora:

- ventanaLista: Lista → Ventana

Modificadora:

- situarLista: Lista x Ventana → Lista

Ventana ventanaLista(Lista lst)
/* Retorna una marca sobre la ventana actual de la lista */

{ pre: lst = < x_1, ..., [xi], ... , x_n > }

ventanaLista

{ post: lst = < x_1, ..., [xi] , ... , x_n > }

Ventana situarLista(Lista lst, Ventana vent)
/* Coloca la ventana de una lista sobre la marca dada */

vent

{ pre: lst = < x_1, ..., xk , ... , [xi], ... , x_n >, vent es una marca sobre una ventana de la lista lst, no se han hecho modificaciones sobre la lista desde el momento en que se tomó la marca }

vent

{ post: lst = < x_1, ..., [xk] , ... , x_i , ... , x_n > }

Ejemplo 2.14:

Indicar si una lista es sublista de otra. La primera solución que se plantea utiliza la operación posLista para volver a situarse sobre la primera estructura. Tiene una complejidad de O(n^2) (en cualquier implementación sobre apuntadores), donde n es la longitud de la primera lista. La razón de esto es que la complejidad exacta de la rutina es O(n * max(n, m)), pero n ≥ m (para que entre al ciclo aunque sea una vez). Luego se puede afirmar que en el peor de los casos es O(n^2).

```
/* pre: l1 = < x1,..., xn >, l2 = < y1,..., ym > */
/* post: indica si l2 es una sublista de l1 */

int ocurreLista( Lista l1, Lista l2 )
{   int i;
    for( i = 1; i <= longLista( l1 ) - longLista( l2 ) + 1; i++ )
    {     posLista( l1, i );
          for( primLista( l2 ); !finLista( l2 ) && infoLista( l1 ) == infoLista( l2 ); sigLista( l1 ), sigLista( l2 ) );
          if( finLista( l2 ) )
                return TRUE;
    }
    return FALSE;
}
```

La segunda solución utiliza el mecanismo de dejar una marca sobre la ventana de la primera lista, de manera que pueda volver a situarse sobre ella en complejidad constante. La función es O(n * m) para cualquier implementación.

```
/* pre: lst1 = < x1,…, xn >, lst2 = < < y1,…, ym > */
/* post: indica si lst2 es una sublista de lst1 */

int ocurreLista( Lista l1, Lista l2 )
{   int i;
    Ventana v;
    for( primLista(l1 ), i = 1; i <= longLista( l1 ) - longLista( l2 ) + 1; i++, sigLista( l1 ) )
    {     v = ventanaLista( l1 );
          for( primLista( l2 ); !finLista( l2 ) && infoLista( l1 ) == infoLista( l2 ); sigLista( l1 ), sigLista( l2 ) );
          if( finLista( l2 ) )
                return TRUE;
          situarLista( l1, v );
    }
    return FALSE;
}
```

Ejercicios Propuestos

Especifique formalmente (precondición y postcondición) y desarrolle una rutina en C para resolver los siguientes problemas. Calcule la complejidad de la solución, suponiendo que todas las operaciones del TAD Lista son O(1).

2.1. void adicLista(Lista lst, TipoL elem)
/* Adiciona el elemento elem al final de lst */

2.2. void sustLista(Lista lst, TipoL elem)
/* Sustituye el contenido actual de la ventana por el valor elem */

2.3. int estaLista(Lista lst, TipoL elem)
/* Indica si el elemento elem aparece en la lista */

2.4. void imprimirLista(Lista lst)
/* Imprime los elementos de la lista, utilizando la operación posLista para avanzar */

2.5. void antLista(Lista lst)
/* Coloca la ventana en la posición anterior a la actual */

2.6. int posVentanaLista(Lista lst)
/* Retorna la posición de la ventana en lst */

2.7. void simplificarLista(Lista lst)
/* Deja en lst una sola ocurrencia de cada uno de los elementos presentes */

2.8. int numDiferentes(Lista lst)
/* Retorna el número total de elementos diferentes en lst */

2.9. int numOcurre(Lista lst, TipoL elem)
/* Calcula el número de veces que aparece elem en lst */

2.10. TipoL maxOcurre(Lista lst)
/* Retorna el elemento que aparece un mayor número de veces en la lista no vacía lst */

2.11. int ultOcurre(Lista lst, TipoL elem)
/* Retorna la posición de la última aparición de elem. Si no ocurre, retorna 0 */

2.12. int medioOcurre(Lista lst, TipoL elem)
/* Retorna la posición de la **aparición media** del valor elem, la cual cumple que existen tantas ocurrencias del elemento antes que él (incluyéndolo, si hay un número par de apariciones) que después de él. Por ejemplo, para la lista de valores enteros < 1, 4, 2, 5, 2, 6, 2, 1, 6, 2, 6 >, la posición de la aparición media del valor 2 es 5, la posición de la aparición media del valor 6 es 9, y la del valor 5 es 4. Si no ocurre, retorna 0 */

2.13. void podarLista(Lista lst)
/* Deja en lst una sola ocurrencia de cada uno de los elementos presentes, garantizando que sea la aparición media del elemento la que permanezca. Por ejemplo, después de podar la lista de valores enteros < 1, 4, 2, 5, 2, 6, 2, 1, 6, 2, 6 >, se obtiene la lista < 1, 4, 5, 2, 6 > */

2.14. int localizarLista(Lista lst, TipoL elem, int i)
/* Retorna la posición de la i-ésima ocurrencia del valor elem en la lista lst. Si hay menos de i ocurrencias, retorna el valor 0 */

2.15. void partirLista(Lista lst, Lista lst1, Lista lst2, TipoL elem)
/* Deja en lst1 todos los elementos de lst menores que elem, y en lst2 los mayores a dicho elemento. Suponga que las listas lst1 y lst2 llegan inicializadas y vacías a la rutina */

2.16. TipoL medianaLista(Lista lst)
/* La **mediana** de una lista se define como el elemento de dicha secuencia tal que la mitad de los elementos son menores que él y la otra mitad mayores o iguales. Suponiendo que la lista lst es no vacía, no ordenada y sin elementos repetidos, esta función calcula su mediana */

2.17. Lista binario(int num)
/* Retorna la lista de ceros y unos correspondiente a la representación binaria del entero positivo num. Por ejemplo, si num = 215, binario =< 1 1 0 1 0 1 1 1 > */

2.18. int esPalindrome(Lista lst)
/* Una lista de caracteres es un **palíndrome** si es igual leerla de izquierda a derecha que de derecha a izquierda. Por ejemplo, son palíndromes:

< A N I T A L A V A L A T I N A >
< D A B A L E A R R O Z A L A Z O R R A E L A B A D >

Esta función indica si la lista lst es un palíndrome */

2.19. int semejantesListas(Lista lst1, Lista lst2)
/* Informa si las listas lst1 y lst2 son semejantes */

2.20. TipoL mayorElemento(Lista lst)
/* Retorna el elemento de la lista lst cuyo valor es máximo, suponiendo la existencia de una relación de orden ≤ definida entre los elementos de la lista. La lista lst no es vacía */

2.21. void rotarLista(Lista lst, int n)
/* Esta rutina rota n posiciones los elementos de la lista lst. **Rotar** una posición significa pasar el primer elemento de la lista a la última posición y desplazar todos los demás elementos una posición hacia la izquierda. n puede ser mayor que la longitud de la lista. n es mayor o igual a cero */

2.22. int igualOcurre(Lista lst)
/* Indica si todos los elementos de la lista aparecen igual número de veces */

2.23. int mayorLista(Lista lst1, Lista lst2)
/* Informa si la lista lst1 es mayor que la lista lst2, utilizando el siguiente criterio para decidir si una lista es mayor que otra:
Sea lst1 = < $a_1, a_2, \ldots, a_n$ > y lst2 = < $b_1, b_2, \ldots, b_m$ >, se dice que lst1 es **mayor** que lst2 si:
1. ($a_i = b_i$ $1 \le i \le m$) y ($n > m$), o 2. $\exists j \le n, j \le m \mid (a_i = b_i$ $1 \le i < j$) y ($a_j > b_j$) */

2.24. int contieneLista(Lista lst1, Lista lst2)
/* informa si la lista lst2 está contenida en la lista lst1 */

2.25. int insertarLista(Lista lst1, Lista lst2, int pos)
/* Inserta la lista lst2 en la lista lst1 de manera que el primer elemento de lst2 quede en la posición pos de lst1 */

2.26. void eliminarLista(Lista lst, int p1, int p2)
/* Elimina de lst los elementos comprendidos entre la posición p1 y la posición p2 inclusive */

2.27. void elimOcurreLista(Lista lst1, Lista lst2)
/* Elimina todas las ocurrencias de la lista lst2 en la lista lst1 */

2.28. void ordenarLista(Lista lst)
/* Ordena ascendentemente la lista lst */

2.29. int intLista(Lista lst)
/* lst es una lista de dígitos. intLista es el entero que se puede crear a partir de esa lista. Por ejemplo, si lst = < 4, 2, 1, 4 >, intLista = 4214 */

2.30. void diferenciaLista(Lista lst1, Lista lst2)
/* Elimina de la lista lst1 todos los elementos que aparecen en la lista lst2 */

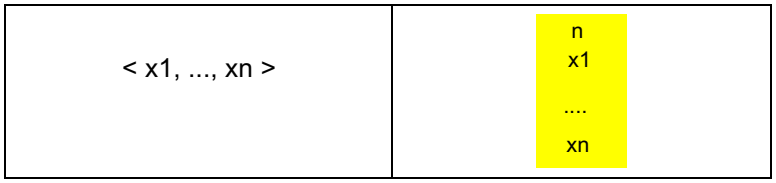
2.31. void sumarLista(Lista lst1, Lista lst2)
/* lst1 y lst2 son listas de dígitos que representan un número entero. Esta rutina suma a lst1 la lista lst2. Por ejemplo, si lst1 =< 4, 2, 1, 4 > y lst2 = < 3, 4 >, después de la llamada a la función sumarLista el valor de lst1 es < 4, 2, 4, 8 > */

2.32. void restarLista(Lista lst1, Lista lst2)
/* lst1 y lst2 son listas de dígitos que representan un número entero. Esta rutina resta de lst1 la lista lst2. Por ejemplo, si lst1 = < 4, 2, 1, 4 > y lst2 = < 3, 4 >, después de la llamada a la función restarLista el valor de lst1 es < 4, 1, 8, 0 > */

2.5. Esquema de Persistencia

Hacer persistir una lista, no es una labor muy complicada. Básicamente, se debe almacenar en memoria secundaria el número de elementos que la componen y la información asociada con cada uno de ellos. Las operaciones de persistencia deben suponer que cada uno de los elementos de la lista tiene una operación asociada para leer y escribir en el disco (leerTipo y salvarTipo). Si se trata de un tipo simple, es posible utilizar las rutinas fscanf y fprintf directamente.

Un posible esquema de persistencia para una lista se puede expresar gráficamente de la siguiente manera:



Las rutinas que implementan estas dos operaciones, expresadas en términos de las operaciones del TAD, son las siguientes:

```
Lista cargarLista( FILE *fp )
{   Lista lst = inicLista( );
    TipoL elem;
    int longit;
    fscanf( fp, "%d", &longit );          /* Lee el número de elementos */
    for( ; longit > 0; longit-- )
    {     elem = leerTipo( fp );          /* Lee cada elemento de la lista */
          anxLista( lst, elem );          /* Adiciona el nuevo elemento al final de la lista */
    }
    return lst;
}

void salvarLista( Lista lst, FILE *fp )
{   fprintf( fp, "%d\n", longLista( lst ) );          /* Escribe en el archivo el número de elementos */
    for( primLista( lst ); !finLista( lst ); sigLista( lst ) )
          salvarTipo( infoLista( lst ), fp );          /* Escribe el siguiente elemento en el archivo */
}
```

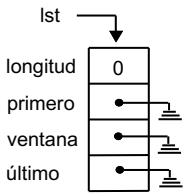
2.6. Algunas Implementaciones del TAD Lista

En esta parte se presentan varias implementaciones posibles para el TAD Lista, entre las cuales están: listas doblemente encadenadas, vectores, listas encadenadas con centinela y listas encadenadas con encabezado. Como parte de los ejercicios se proponen otras representaciones internas.

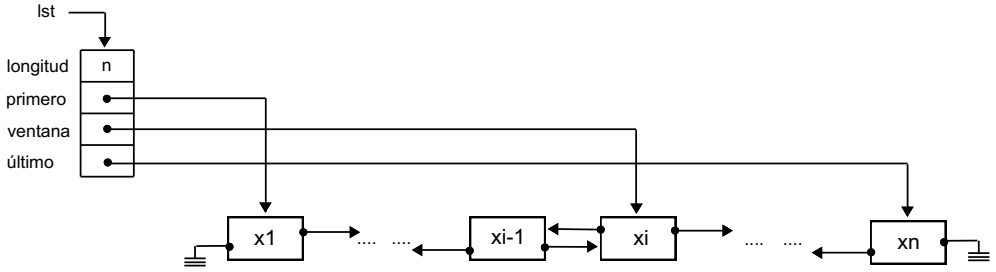
2.6.1. Estructura Doblemente Encadenada

En esta representación, la lista tiene un nodo con apuntadores al primer elemento, al último y al elemento de la ventana. Los nodos, por su parte, se encuentran doblemente encadenados entre sí, para permitir que las modificadoras se puedan implementar con algoritmos O(1). El esquema de representación se puede resumir en los siguientes tres casos:

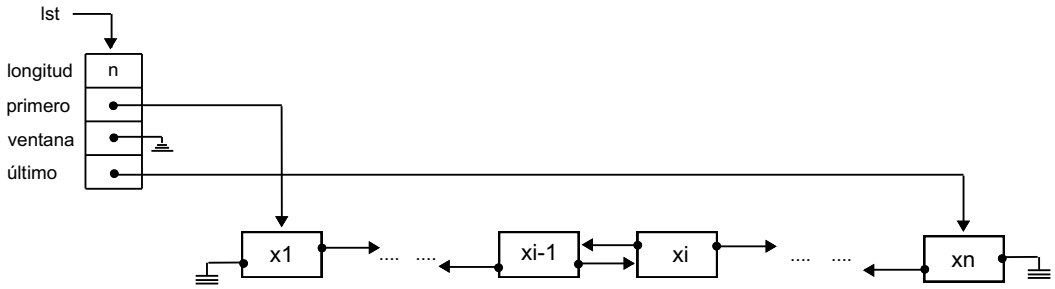
- La lista vacía (lst = < >□) se representa internamente como un apuntador a un registro, tal como se muestra en la figura:



- En el caso general, la lista lst =< x_1, ...,[xi] , ... , x_n > utiliza las siguientes estructuras de datos:



- Si la ventana está indefinida (lst =< x_1, ... , x_n >□), se coloca el valor NULL en el respectivo campo del encabezado.



Las estructuras de datos se declaran de la siguiente manera, definiendo una lista como un apuntador a su nodo de encabezado:

```
typedef struct ListaNodo
{   TipoL info;                     /* Elemento de la lista */
    struct ListaNodo *ant, *sig;    /* Encadenamientos hacia adelante y hacia atrás */
}   *pListaNodo;

typedef struct
{   pListaNodo primero,             /* Primer elemento de la lista */
               ultimo,              /* Ultimo elemento de la lista */
               ventana;             /* Elemento en la ventana de la lista */
    int longitud;
} TLista, *Lista;

typedef pListaNodo Ventana;         /* Marca para almacenar la ventana de la lista */
```

Algunas de las rutinas que implementan las operaciones del TAD se dan a continuación, con una pequeña explicación de su funcionamiento. El resto de operaciones aparecen desarrolladas en el disquete de apoyo.

Para crear una lista vacía basta con pedir a la memoria dinámica el nodo de encabezado, y llenarlo de acuerdo con el primer caso del esquema de representación:

```
Lista inicLista( void  )
{   Lista resp;
    resp = ( Lista )malloc( sizeof( TLista ) );
    resp->primero = resp->ultimo = resp->ventana = NULL;
    resp->longitud = 0;
    return resp;
}
```

La rutina que agrega un elemento después de la ventana debe considerar 3 casos: la lista es vacía, la ventana está sobre el último elemento o se debe adicionar un nodo intermedio.

```
void anxLista( Lista lst, TipoL elem )
{   pListaNodo nuevo = ( pListaNodo )malloc( sizeof( struct ListaNodo ) );
    nuevo->info = elem;
    nuevo->ant = nuevo->sig = NULL;
    if ( lst->longitud == 0 )
        lst->primero = lst->ultimo = nuevo;
```

```
    else if ( lst->ventana == lst->ultimo )
    {     lst->ventana->sig = lst->ultimo = nuevo;
          nuevo->ant = lst->ventana;
    }
    else
    {     nuevo->ant = lst->ventana;
          nuevo->sig = lst->ventana->sig;
          lst->ventana->sig->ant = nuevo;
          lst->ventana->sig = nuevo;
    }
    lst->ventana = nuevo;
    lst->longitud++;
}
```

Esta rutina elimina el elemento que se encuentra en la ventana. Aprovechando el doble encadenamiento de los nodos, es posible hacer esta operación en O(1).

```
void elimLista( Lista lst )
{   pListaNodo aux;
    if ( lst->ventana == lst->primero )
    {     if ( lst->ultimo == lst->primero )
                lst->ultimo = NULL;
          lst->primero = lst->primero->sig;
          free( lst->ventana );
          lst->ventana = lst->primero;
    }
    else
    {     if ( lst->ultimo == lst->ventana )
                lst->ultimo = lst->ultimo->ant;
          lst->ventana->ant->sig = lst->ventana->sig;
          if ( lst->ventana->sig ! =  NULL )
                lst->ventana->sig->ant = lst->ventana->ant;
          aux = lst->ventana;
          lst->ventana = lst->ventana->sig;
          free( aux );
    }
    lst->longitud--;
}
```

La operación de destrucción debe retornar toda la memoria asociada con la lista. Esto es, tanto los nodos de los elementos, como el nodo del encabezado.

```
void destruirLista( Lista lst )
{   pListaNodo p, q;
    for( p = lst->primero; p != NULL; )
    {     q = p;
          p = p->sig;
          free( q );
    }
    free( lst );
}
```

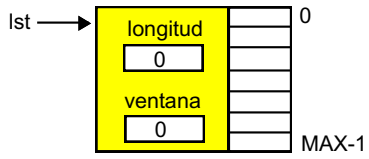
Algunas de las operaciones de manejo de la ventana se implementan de la siguiente manera:

```
void posLista( Lista lst,  int pos )
{   int i;
    for( lst->ventana = lst->primero, i = 1; i < pos; i++ )
          lst->ventana = lst->ventana->sig;
}

void situarLista( Lista lst, Ventana vent )
{   lst->ventana = vent;
}

Ventana ventanaLista( Lista lst )
{   return lst->ventana;
}
```

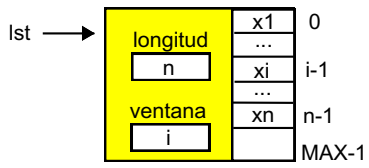
2.6.2. Vectores

En esta representación, los elementos de la lista se sitúan consecutivamente en un vector. Maneja además dos campos adicionales que indican la longitud actual y la posición de la ventana. El esquema de representación se resume en los siguientes puntos:

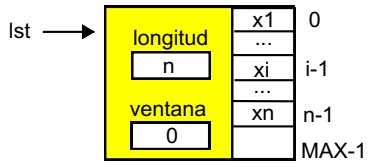
- La lista vacía (lst = < >) se representa internamente como:



- La lista lst = < x_1,..., xi ,..., x_n > utiliza las siguientes estructuras de datos:



- La lista con ventana indefinida (lst =< x_1, ... , x_n >) se define con el valor 0 en el campo respectivo.



MAX es una constante que define una restricción de implementación, puesto que nunca se va a poder manejar una lista con un mayor número de elementos. Este es uno de los puntos más débiles de esta manera de representar listas. Las estructuras de datos, para esta implementación, se declaran de la siguiente forma:

```
typedef struct
{   TipoL info[ MAX ];      /* Vector para almacenar los elementos de la lista */
    int longitud,           /* Número de elementos de la lista */
        ventana;            /* Posición de la ventana en la lista (1..longitud) */
} TLista, *Lista;

typedef int Ventana;        /* Marca para almacenar la ventana de la lista */
```

Algunas de las rutinas que implementan las operaciones del TAD se presentan a continuación.

- Agregar un elemento después de la ventana:

```
void anxLista( Lista lst, TipoL elem )
{   int i;
    if ( lst->longitud! = 0 )
        for ( i = lst->longitud-1; i > lst->ventana-1; i-- )
            lst->info[ i+1 ] = lst->info[ i ];
    lst->info[ lst->ventana ] = elem;
    lst->ventana++;
    lst->longitud++;
}
```

- Eliminar el elemento de la lista que se encuentra en la ventana:

```
void elimLista( Lista lst )
{   int i;
    for ( i = lst->ventana - 1; i < lst->longitud; i++ )
        lst->info[ i ] = lst->info[ i+1 ];
    lst->longitud--;
    if ( lst->ventana > lst->longitud )
        lst->ventana = 0;
}
```

- Avanzar la ventana una posición:

```
void sigLista( Lista lst )
{   if ( lst->ventana == lst->longitud )
        lst->ventana = 0;
    else
        lst->ventana++;
}
```

- Colocar la ventana en la primera posición de la lista:

```
void primLista( Lista lst )
{   lst->ventana = ( lst->longitud == 0 ) ? 0 : 1;
}
```

- Destruir la lista:

```
void destruirLista( Lista lst )
{   free( lst );
}
```

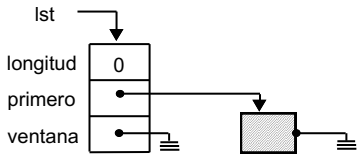
- Administrar la ventana:

```
void posLista( Lista lst,  int pos )
{   lst->ventana = pos;
}

void situarLista( Lista lst, Ventana vent )
{   lst->ventana = vent;
}

Ventana ventanaLista( Lista lst )
{   return lst->ventana;
}
```

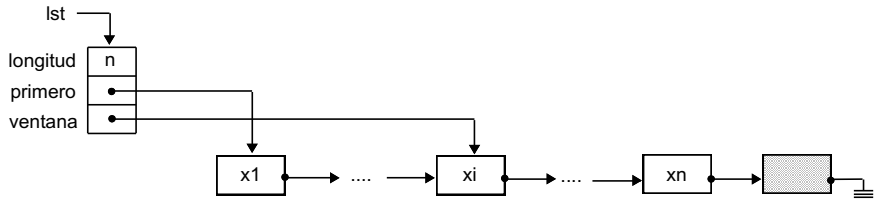
2.6.3. Encadenamiento Sencillo con Centinela

Con esta representación se hace un ahorro considerable de memoria, respecto a la primera implementación planteada, ya que no requiere un doble encadenamiento entre los nodos. Al igual que en ese caso, se maneja un registro de encabezado como parte de la estructura de datos. El **centinela** es un nodo que se agrega al final de toda la secuencia de elementos; no tiene información válida, pero permite realizar todas las modificadoras en O(1) sin necesidad del encadenamiento hacia atrás, como se verá más adelante. El esquema de representación se resume en los siguientes casos:

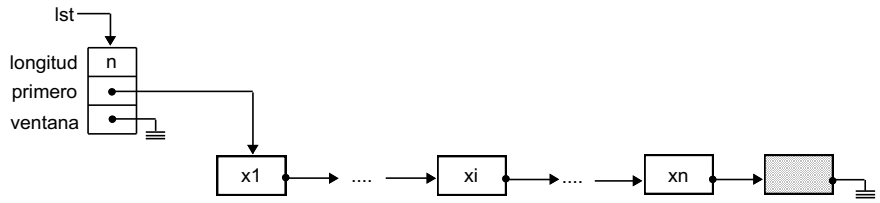
- La lista vacía (lst = <> □) se representa internamente como:



- La lista lst =< x_1, ...,[xi] , ... , x_n > utiliza las siguientes estructuras de datos:



- La lista con ventana indefinida (lst =< x_1, ... , x_n >□) se representa con el valor NULL en ese campo del encabezado.



Las estructuras de datos se declaran así:

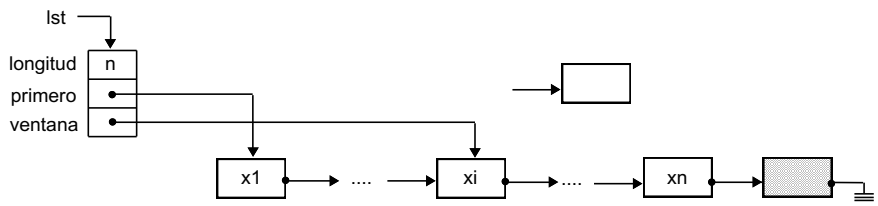
```
typedef struct ListaNodo
{   TipoL info;                 /*Información de cada elemento de la lista */
    struct ListaNodo *sig;      /* Encadenamiento de los nodos */
} *pListaNodo;

typedef struct
{   pListaNodo primero,         /* Apuntador al primer nodo de la lista */
               ventana;         /* Apuntador al nodo que se encuentra en la ventana */
    int longitud;               /* Número de elementos de la lista */
} TLista, *Lista;

typedef pListaNodo Ventana;     /* Marca para almacenar la ventana de la lista */
```

Algunas de las rutinas que implementan las operaciones del TAD son las siguientes.

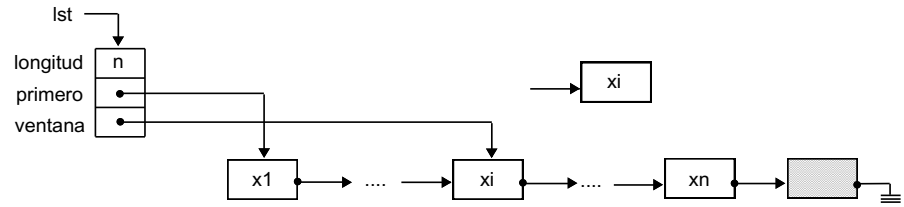
- Crear una lista vacía:

```
Lista inicLista( void )
{   Lista resp;
    pListaNodo centinela;
    resp = ( Lista )malloc( sizeof( TLista ) );
    centinela = ( pListaNodo )malloc( sizeof( struct ListaNodo ) );
    centinela->sig = NULL;
    resp->longitud = 0;
    resp->primero = centinela;
    resp->ventana = NULL;
    return resp;
}
```

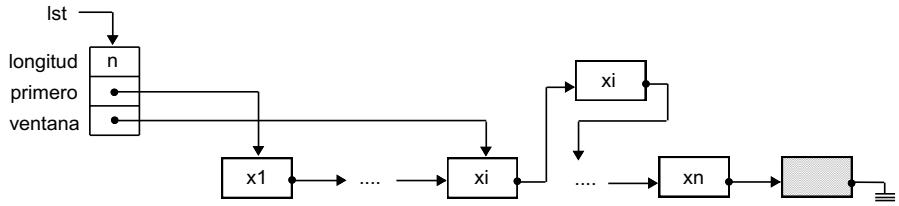
- Insertar un elemento en la lista: Ante la dificultad de alterar el campo de encadenamiento del antecesor de la ventana, para que incluya el nuevo nodo, se coloca en la ventana la información del nuevo elemento y se agrega después de éste un nodo con el valor que aparecía inicialmente en la ventana. El proceso se ilustra con la siguiente secuencia de gráficas:



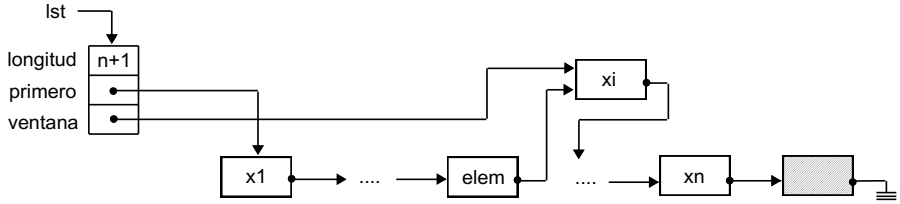
Paso 1: se pide el nuevo nodo



Paso 2: se coloca en el nuevo nodo el elemento de la ventana



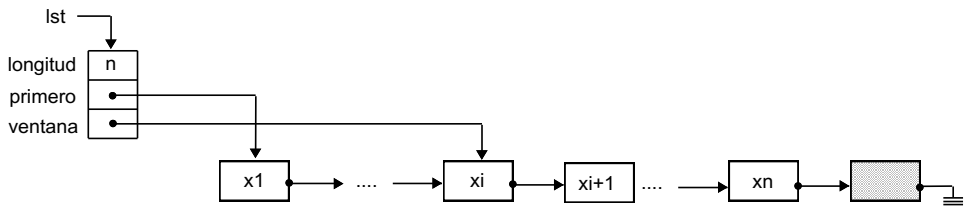
Paso 3: se encadena el nuevo nodo después de la ventana



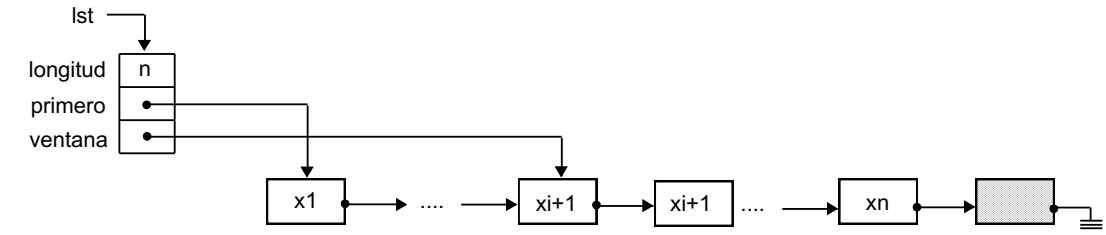
Paso 4: se coloca el nuevo elemento en el nodo viejo y se mueve la ventana

```
void insLista( Lista lst, TipoL elem )
{   pListaNodo p = lst->ventana;
    if( lst->longitud != 0 )
    {     TipoL temp = lst->ventana->info;
          lst->ventana->info = elem;
          anxLista( lst, temp );
          lst->ventana = p;
    }
    else
          anxLista( lst, elem );
}
```

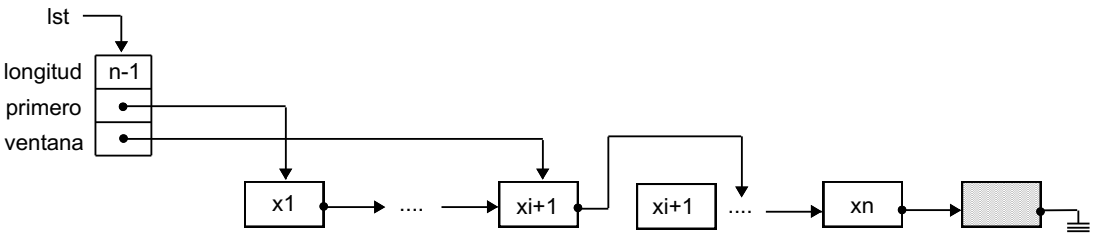
- Eliminar un elemento de la lista: Para esta rutina se presenta el mismo problema que existe en el caso anterior. La solución es colocar la información del sucesor sobre la ventana, y después eliminar el sucesor, un proceso sencillo de reencadenamiento. Para poder hacer esta operación en O(1), es que se coloca el centinela como parte de las estructuras de datos, ya que evita que se presente un caso especial con el último elemento, para el cual el proceso descrito anteriormente no sería aplicable ante la ausencia de un sucesor. El proceso se ilustra con la siguiente secuencia de gráficas:



Situación inicial



Paso 1: se coloca la información del sucesor en el nodo que se quiere eliminar



Paso 2: se elimina el sucesor

```
void elimLista( Lista lst )
{   pListaNodo aux;
    lst->ventana->info = lst->ventana->sig->info;
    aux = lst->ventana->sig;
    lst->ventana->sig = aux->sig;
    free( aux );
    if ( lst->ventana->sig == NULL )
          lst->ventana = NULL;
    lst->longitud--;
}
```

- Destruir la lista:

```
void destruirLista( Lista lst )
{   pListaNodo p, q;
    for( p = lst->primero; p != NULL; )
    {     q = p;
          p = p->sig;
          free( q );
    }
    free( lst );
}
```

- Administrar la ventana:

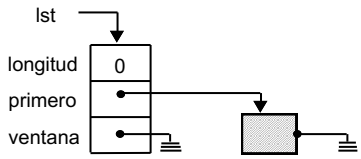
```
void posLista( Lista lst,  int pos )
{   int i;
    for( lst->ventana = lst->primero, i = 1;  i < pos;  i++ )
          lst->ventana = lst->ventana->sig;
}

void situarLista( Lista lst, Ventana vent )
{   lst->ventana = vent;
}
```

```
Ventana ventanaLista( Lista lst )
{   return lst->ventana;
}
```

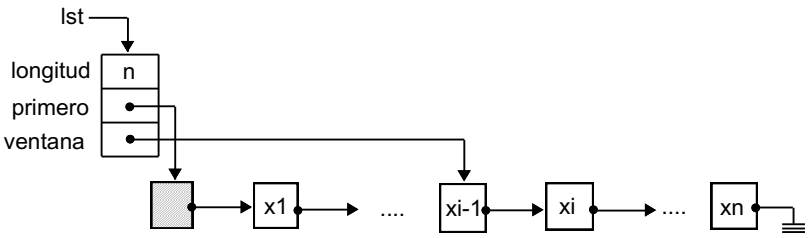
2.6.4. Encadenamiento Sencillo con Encabezado

Esta manera de representar una lista resuelve el problema de inserción (explicado en la sección anterior) llevando el apuntador de la ventana una posición retrasada con respecto al elemento que quiere indicar. Con el objeto de que el primer elemento no se convierta en un caso particular, se coloca un nodo adicional sin información válida al comienzo de la lista. Este esquema de representación se resume en los siguientes casos:

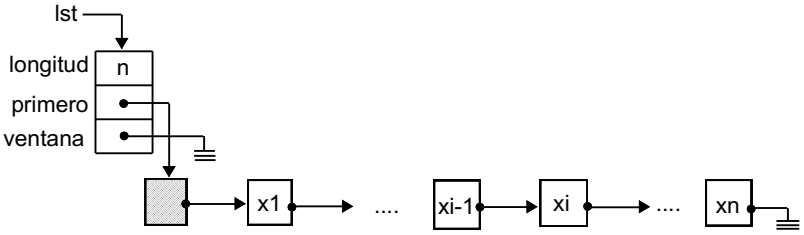
- La lista vacía (lst = < >□) se representa internamente como:



- La lista lst =< x_1, ...,[xi] , ... , x_n > utiliza las siguientes estructuras de datos:



- La lista con ventana indefinida (lst = < x_1, ... , x_n >□) se representa con el valor NULL en dicho campo del encabezado.



Las estructuras de datos se declaran así:

```
typedef struct ListaNodo
{   TipoL info;                 /* Información de cada elemento de la lista */
    struct ListaNodo *sig;      /* Encadenamiento de los nodos */
} *pListaNodo;

typedef struct
{   pListaNodo primero,         /* Apuntador al primer nodo de la lista */
               ventana;         /* Apuntador al nodo que se encuentra en la ventana */
    int longitud;               /* Número de elementos de la lista */
} TLista, *Lista;
```

```
typedef pListaNodo Ventana;        /* Marca para almacenar la ventana de la lista */
```

Las rutinas para implementar las operaciones del TAD bajo esta representación se proponen como ejercicio al lector.

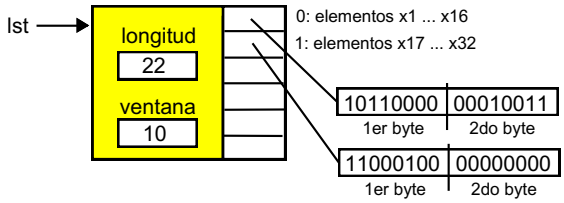
2.6.5. Representación a Nivel de *Bits*

Cuando los elementos de la lista corresponden a valores lógicos (verdadero y falso), es posible utilizar una representación compacta de la información, de tal manera que cada elemento utilice un solo *bit* en las estructuras de datos. La idea es usar un vector de enteros, en el cual, en cada posición, se almacenen sizeof(int)* 8 valores. Esto permite optimizar el espacio sin perder eficiencia en el acceso. Por ejemplo, en un computador que represente un valor entero con 2 *bytes*, es posible almacenar 16 elementos en cada casilla del vector, como se muestra en el ejemplo 2.15.

Para su implementación, las operaciones del TAD deben utilizar las facilidades de manejo de *bits* que ofrece el lenguaje C.

Ejemplo 2.15:

Utilizando unas estructuras de datos parecidas a las definidas en la implementación de vectores (§2.6.2), se tiene que la lista de valores booleanos lst = < 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1 > se representa como:



Note cómo una lista de 22 elementos utiliza únicamente dos enteros (4 *bytes*) para su representación, logrando así un ahorro importante en memoria.

⏎

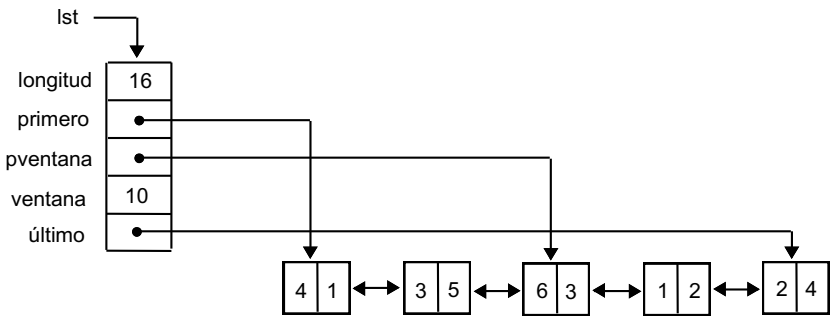
El esquema de representación, la declaración de las estructuras de datos y la implementación de las operaciones se dejan como ejercicio al lector.

2.6.6. Representación Compacta de Elementos Repetidos

Para representar una lista, en la cual aparecen secuencias de valores repetidos, se pueden diseñar unas estructuras de datos compactas, que tengan en cada nodo el valor y el número de repeticiones consecutivas. En el siguiente ejemplo se ilustra la idea. El ahorro en espacio puede ser considerable, pero a costa de una mayor dificultad de los algoritmos que implementan las operaciones del TAD.

Ejemplo 2.16:

La lista lst = < 1, 1, 1, 1, 5, 5, 5, 3, 3, 3, 3, 3, 3, 2, 4, 4 > se representa internamente mediante la siguiente estructura doblemente encadenada, que tiene, en cada nodo, el número de ocurrencias consecutivas de cada valor:



El esquema de representación, la declaración de las estructuras de datos y la implementación de las operaciones se dejan como ejercicio al lector.

2.6.7. Multirrepresentación

En los casos en los cuales es difícil predecir las características de la información que se quiere almacenar en una lista (número de elementos, dispersión de los datos, rango de valores, etc.), y es indispensable minimizar el espacio ocupado en memoria, es posible diseñar las estructuras de datos de tal manera que el mismo objeto abstracto decida -en cada caso y durante la ejecución- cómo hacer la representación. Por ejemplo, se puede intentar hacer una representación compacta, como la sugerida en la sección anterior, pero si el objeto detecta que está desperdiciando espacio puede cambiar a una representación que ocupe menos memoria, como podría ser una estructura sencillamente encadenada con centinela.

La ventaja de este enfoque es que cada objeto abstracto se acomoda al problema en el cual trabaja: cada lista decide cómo almacenar su información, según su estado específico. El problema es que se deben implementar todas las operaciones del TAD para cada estructura de datos posible.

Las estructuras de datos deben tener en su encabezado un indicador del tipo de representación que se está usando, y dos operaciones para hacer la traducción entre las dos.

2.6.8. Tabla Comparativa

La siguiente tabla contiene la complejidad de cada una de las operaciones del TAD, sobre las cuatro primeras estructuras de datos planteadas:

	1	2	3	4
inicLista	O(1)	O(1)	O(1)	O(1)
anxLista	O(1)	O(N)	O(1)	O(1)
insLista	O(1)	O(N)	O(1)	O(1)
elimLista	O(1)	O(N)	O(1)	O(1)
sigLista	O(1)	O(1)	O(1)	O(1)
primLista	O(1)	O(1)	O(1)	O(1)
ultLista	O(1)	O(1)	O(N)	O(N)
posLista	O(N)	O(1)	O(N)	O(N)
situarLista	O(1)	O(1)	O(1)	O(1)
infoLista	O(1)	O(1)	O(1)	O(1)
longLista	O(1)	O(1)	O(1)	O(1)

finLista	O(1)	O(1)	O(1)	O(1)
ventanaLista	O(1)	O(1)	O(1)	O(1)
destruirLista	O(N)	O(1)	O(N)	O(N)
cargarLista	O(N)	O(N)	O(N)	O(N)
salvarLista	O(N)	O(N)	O(N)	O(N)

Una comparación completa de las representaciones internas para una lista se sugiere más adelante como ejercicio.

Ejercicios Propuestos

Implemente las siguientes rutinas sobre las estructuras de datos de las primeras cuatro representaciones vistas. Calcule la complejidad de sus algoritmos.

2.33. int posVentanaLista(Lista lst)
/* Retorna la posición de la ventana en lst */

2.34. void antLista(Lista lst)
/* Coloca la ventana en la posición anterior a la actual */

2.35. int estaLista(Lista lst, TipoL elem)
/* Indica si el elemento elem se encuentra en la lista lst */

2.36. void adicLista(Lista lst, TipoL elem)
/* Adiciona al final de la lista lst el elemento elem */

2.37. void impLista(Lista lst)
/* Imprime los elementos de la lista lst */

2.38. void eliminarLista(Lista lst, int pos)
/* Elimina de la lista lst el pos-ésimo elemento */

Los siguientes ejercicios proponen la implementación del TAD Lista sobre otras estructuras de datos. Calcule la complejidad de los algoritmos que desarrolle.

2.39. 💻 Para la representación de encadenamiento sencillo con encabezado, implemente todas las operaciones del TAD. Utilice el probador interactivo del TAD para validar la implementación.

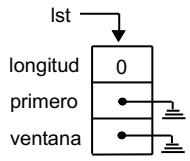
2.40. 💻 Para la representación a nivel de *bits*, especifique formalmente el esquema de representación, haga la declaración de las estructuras de datos e implemente las operaciones del TAD. Utilice el probador interactivo del TAD para validar la implementación.

2.41. 🕒 💻 Para la representación compacta de elementos repetidos, especifique formalmente el esquema de representación, haga la declaración de las estructuras de datos e implemente las operaciones del TAD. Utilice el probador interactivo del TAD para validar la implementación.

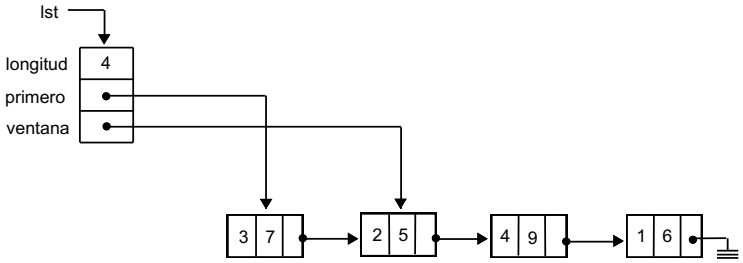
2.42. 🕒 💻 Implemente una multirrepresentación para el TAD Lista, utilizando la representación compacta de elementos repetidos y la representación con encadenamiento sencillo y encabezado. Defina las características bajo las cuales comienza a ser más eficiente un almacenamiento que otro. Implemente una rutina de uso local, que haga la traducción de una representación a otra. Utilice el probador interactivo del TAD para validar la implementación.

2.43. 💻 Una lista se puede representar internamente como una secuencia de parejas [posición, elemento], con un encabezado que indica el número de elementos presentes.

La lista vacía (lst = < >□) se representa como:

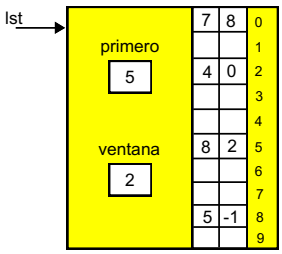


La lista lst =< 6,[5],7,9 >, por ejemplo, puede utilizar las siguientes estructuras de datos:

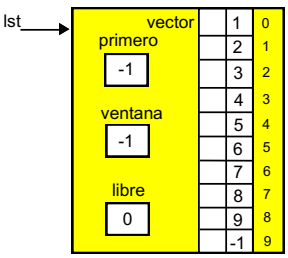


La ventana indefinida se representa con el apuntador respectivo en NULL. Fíjese que las parejas no van ordenadas por ningún concepto. Especifique formalmente el esquema de representación, haga la declaración de las estructuras de datos e implemente las operaciones del TAD. Utilice el probador interactivo del TAD para validar la implementación.

2.44. En lenguajes que no manejan apuntadores, es posible simular su comportamiento mediante el encadenamiento de vectores. Para esto, cada elemento del vector mantiene el índice en el cual se encuentra su sucesor. Por ejemplo, una posible representación de la lista lst = <8,[4],7,5> sería:

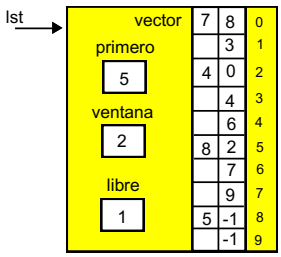


Un problema de esta representación es la administración del espacio libre: los elementos no ocupan posiciones consecutivas y puede haber registros libres intercalados con registros ocupados. Una solución al problema de saber dónde existe espacio libre dentro del vector, es encadenar todas las casillas no utilizadas. En ese caso se manejan dos listas sobre la misma estructura, una con los registros libres y otra con la secuencia que se está representando. Esto implica que una lista vacía se representa con todos los nodos libres encadenados, como se sugiere a continuación:



Cuando se requiere un nuevo registro, se toma de la lista de casillas libres. Al liberar espacio, se lleva a cabo el proceso contrario, agregándolo a dicha lista.

Con esta solución, la lista <8,[4],7,5> se puede representar con la siguientes estructuras de datos:



Como no se maneja doble encadenamiento ni centinela, la mayoría de operaciones son O(N), donde N es la longitud de la lista. Especifique formalmente el esquema de representación, haga la declaración de las estructuras de datos e implemente las operaciones del TAD. Utilice el probador interactivo del TAD para validar la implementación.

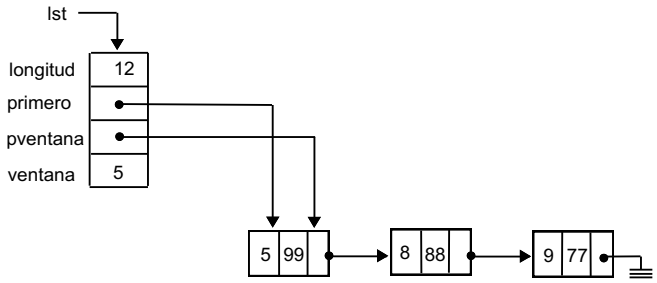
2.45. Mejore el diseño de las estructuras de datos del ejercicio anterior, para que disminuya la complejidad de las operaciones del TAD. Especifique el esquema de representación y explique la razón por la cual es más eficiente que el anterior diseño.

2.46. Una manera de evitar el desplazamiento de información en la implementación con vectores, es manejar 2 arreglos: uno con **cursores**, que indica el orden de los elementos dentro de la lista, y el otro con la información en sí. Esto quiere decir que el movimiento de elementos para insertar o eliminar información se reduce al desplazamiento de algunos cursores, sin necesidad de mover físicamente los elementos. Diseñe unas estructuras de datos para utilizar esta idea e implemente todas las operaciones sobre ellas. Utilice el probador interactivo del TAD para validar el desarrollo.

2.47. Para representar una lista, en la cual muchos de los valores contenidos son cero, es posible utilizar la misma idea de las matrices dispersas y únicamente representar de manera explícita los valores distintos de dicho valor. Se maneja, entonces, una secuencia encadenada con las posiciones y los valores de los elementos de la lista cuyo contenido es distinto de cero, ordenada de menor a mayor por posición. Por ejemplo, para la lista:

lst = < 0 0 0 0 [99] 0 0 88 77 0 0 0 >

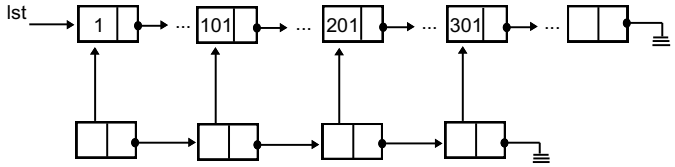
Se tiene la siguiente representación interna:



En caso de que la ventana esté sobre un elemento no representado de manera explícita, se coloca su posición en el campo ventana y el apuntador pventana en NULL. Si la ventana está indefinida, al campo ventana se le asigna 0. Especifique formalmente el esquema de representación, haga la declaración de las estructuras de datos e implemente las operaciones del TAD. Utilice el probador interactivo del TAD para validar la implementación.

2.48. Una manera eficiente de representar listas de tamaño considerable es mediante una secuencia encadenada de nodos con un directorio asociado, el cual tiene una entrada cada cierto número de

elementos de la lista. Por ejemplo, una lista de más de 300 elementos y menos de 400 se puede representar con una estructura parecida a la del dibujo:



Tiene la ventaja de que permite dar "saltos" más largos al moverse sobre los elementos de la secuencia. Especifique formalmente el esquema de representación, haga la declaración de las estructuras de datos e implemente las operaciones del TAD. Utilice el probador interactivo del TAD para validar la implementación.

2.49. Modifique las estructuras de datos utilizadas en la implementación con encadenamiento sencillo y centinela, para que la operación ultLista se pueda hacer en tiempo constante. ¿Qué operaciones del TAD aumentan de complejidad? ¿Es posible hacerlas todas en O(1)?

2.7. El TAD Lista Ordenada

De acuerdo con la definición presentada en la primera sección de este capítulo, una lista $< e_1,..., e_n >$ es ordenada si cumple que $e_i \leq e_{i+1}$, $\forall i \mid 1 \leq i < n$. Como una lista ordenada se puede ver como un objeto abstracto más especializado que una lista, se puede diseñar un TAD para su administración. Como primera medida, se escoge un nuevo formalismo para referirse a las listas ordenadas.

- La lista ordenada vacía se representa como $\subset \supset$.
- La lista ordenada con n elementos se representa como $\subset e_1, ..., e_n \supset$.

El TAD diseñado tiene las siguiente especificación:

TAD ListOrd[TipoLO]		
$\subset e_1, ..., e_n \supset$		
{ inv: $e_i \leq e_{i+1}$, $\forall i \mid 1 \leq i < n$ }		
Constructoras:		
inicListOrd:		→ ListOrd
Modificadoras:		
insListOrd:	ListOrd x TipoLO	→ ListOrd
elimListOrd:	ListOrd x TipoLO	→ ListOrd
Analizadoras:		
infoListOrd:	ListOrd x int	→ TipoLO
longListOrd:	ListOrd	→ int
estaListOrd	ListOrd x TipoLO	→ int

```
ListOrd inicListOrd( void )
/* Crea una lista ordenada vacía */

{ post: inicListOrd = ⊂ ⊃ }
```

```
void insListOrd( ListOrd lst, TipoLO elem )
/* Inserta un elemento a la lista ordenada */

{ pre: ∀i, ei != elem }
{ post: lst = ⊂ e1, ..., ei, elem, ei+1, ..., en ⊃ }
```

```
void elimListOrd( ListOrd lst, TipoLO elem )
/* Elimina un elemento de la lista ordenada */

{ pre: ∃i | ei = elem }
{ post: lst = ⊂ e1, ..., ei-1, ei+1, ..., en ⊃ }
```

```
TipoLO infoListOrd( ListOrd lst, int pos )
/* Retorna el pos-ésimo elemento de la lista ordenada */

{ pre: 1 ≤ pos ≤ n }
{ post: infoListOrd = epos}
```

```
int longListOrd( ListOrd lst )
/* Retorna la longitud de la lista ordenada */

{ post: longListOrd = n }
```

```
int estaListOrd( ListOrd lst, TipoLO elem )
/* Informa si un elemento se encuentra en la lista ordenada */

{ post: estaListOrd = ∃i | ( ei = elem ) }
```

2.8. Implementación del TAD Lista Ordenada

2.8.1. Sobre el TAD Lista

En esta sección se ilustra la manera de utilizar un objeto abstracto del TAD Lista como estructura de datos. El esquema de representación se resume en los siguientes puntos:

- La lista ordenada vacía $\subset \supset$, se representa internamente con la lista vacía < >.
- La lista ordenada con n elementos $\subset e_1,...,e_n \supset$ se representa con la lista $< e_1,...,e_n >$.

Las únicas declaraciones de estructuras de datos necesarias para esta implementación son:

```
typedef TLista TListaOrd;
typedef Lista ListOrd;
```

Ahora, basta con escribir las operaciones del nuevo TAD, en términos de las operaciones del TAD de base, como se muestra a continuación:

- Crear una lista ordenada vacía:

```
ListOrd inicListOrd( void )
{   return inicLista( );
}
```

- Insertar un elemento en una lista ordenada:

```
void insListOrd( ListOrd lst, TipoLO elem )
{   primLista( lst );
    while ( !finLista( lst ) && infoLista( lst ) < elem )
          sigLista( lst );
    if ( !finLista( lst ) )
          insLista( lst, elem );
    else
    {     ultLista( lst );
          anxLista( lst, elem );
    }
}
```

- Eliminar un elemento de una lista ordenada:

```
void elimListOrd( ListOrd lst, TipoLO elem )
{   primLista( lst );
    while ( infoLista( lst ) != elem )
          sigLista( lst );
    elimLista( lst );
}
```

- Decidir si un elemento se encuentra en una lista ordenada:

```
int estaListOrd( ListOrd lst, TipoLO elem )
{   primLista( lst );
    while ( !finLista( lst ) && infoLista( lst ) != elem )
          sigLista( lst );
    return !finLista( lst );
}
```

- Retornar la longitud de una lista ordenada:

```
int longListOrd( ListOrd lst )
{   return longLista( lst );
}
```

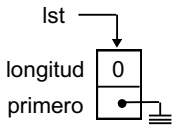
- Retornar el pos-ésimo elemento de una lista ordenada:

```
int infoListOrd( ListOrd lst, int pos )
{   posLista( lst, pos );
    return infoLista( lst );
}
```

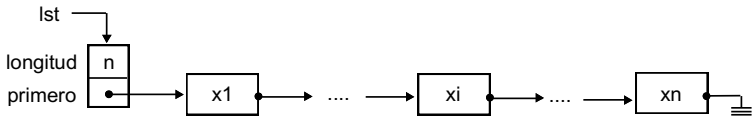
2.8.2. Estructura Sencillamente Encadenada

Una de las representaciones más elementales para una lista ordenada es una estructura sencillamente encadenada con apuntadores. El esquema de representación se resume en los siguientes puntos:

- La lista ordenada vacía $\subset \supset$, se representa como:



- La lista ordenada con n elementos $\subset x_1,..., x_n \supset$ se representa como:



Las declaraciones de estructuras de datos necesarias para esta implementación son:

```
typedef struct ListaNodOr
{   TipoLO info;                    /* Elemento de la lista */
    struct ListaNodOr *sig;         /* Encadenamientos hacia adelante */
}   *pListaNodOr;

typedef struct
{   pListaNodOr primero;            /* Primer elemento de la lista */
    int longitud;
} TListOrd, *ListOrd;
```

La implementación de algunas de las operaciones del TAD ListOrd sobre una lista sencillamente encadenada se presentan a continuación. Las demás se pueden consultar en el disquete de apoyo:

- Crear una lista ordenada vacía:

```
ListOrd inicListOrd( void )
{   ListOrd ls = ( ListOrd )malloc( sizeof( TListOrd ) );
    ls->longitud = 0;
    ls->primero = NULL;
    return ls;
}
```

- Insertar un elemento en una lista ordenada:

```
void insListOrd( ListOrd ls, TipoLO elem )
{   pListaNodOr p = ( pListaNodOr )malloc( sizeof( struct ListaNodOr ) );
    pListaNodOr q;
    p->info = elem;
    p->sig = NULL;
    if( ls->primero == NULL )
        ls->primero = p;
```

```
    else if( elem < ls->primero->info )
        {   p->sig = ls->primero;
            ls->primero = p;
        }
        else
        {   for( q = ls->primero; q->sig != NULL && elem > q->sig->info; q = q->sig );
            p->sig = q->sig;
            q->sig = p;
        }
        ls->longitud++;
    }
```

- Eliminar un elemento de una lista ordenada:

```
    void elimListOrd( ListOrd ls, TipoLO elem )
    {   pListaNodOr p, q;
        if( ls->primero->info == elem )
        {   p = ls->primero;
            ls->primero = ls->primero->sig;
            free( p );
        }
        else
        {   for( p = ls->primero; p->sig->info != elem; p = p->sig );
            q = p->sig->sig;
            free( p->sig );
            p->sig = q;
        }
        ls->longitud--;
    }
```

Ejercicios Propuestos

Especifique formalmente (precondición y postcondición) y desarrolle un algoritmo sobre cada una de las representaciones vistas, para resolver los siguientes problemas de listas ordenadas. Calcule la complejidad de su solución.

2.50. ListOrd mezclarListOrd(ListOrd lst1, ListOrd lst2)
/* Crea y retorna una lista ordenada con todos los elementos de lst1 y lst2 */

2.51. ListOrd interListOrd(ListOrd lst1, ListOrd lst2)
/* Crea y retorna una lista ordenada con la intersección de lst1 y lst2 */

2.52. ListOrd unionListOrd(ListOrd lst1, ListOrd lst2)
/* Crea y retorna una lista ordenada con la unión de lst1 y lst2, dejando una sola ocurrencia de cada elemento */

2.53. ListOrd diferListOrd(ListOrd lst1, ListOrd lst2)
/* Crea y retorna una lista ordenada con todos los elementos de lst1 que no están en lst2 */

2.54. ListOrd subListOrd(ListOrd lst, TipoLO elem)
/* Crea y retorna una lista ordenada con todos los elementos de lst mayores que elem */

Desarrolle las implementaciones del TAD ListOrd sugeridas en los siguientes ejercicios:

2.55. Implemente el TAD ListOrd sobre una representación de vectores. Calcule la complejidad de cada operación. Utilice el probador interactivo para validar el desarrollo.

2.56. Implemente el TAD ListOrd utilizando como representación interna una estructura doblemente encadenada. Calcule la complejidad de cada operación. Utilice el probador interactivo para validar el desarrollo.

2.57. Implemente el TAD ListOrd sobre una estructura sencillamente encadenada con centinela. Calcule la complejidad de cada operación. Utilice el probador interactivo para validar el desarrollo.

2.58. Haga una tabla completa para comparar la complejidad de las operaciones del TAD ListOrd. Decida cuál de todas las opciones es la más conveniente:

1: TAD ListOrd sobre TAD Lista (doble encadenamiento)
2: TAD ListOrd sobre TAD Lista (vector)
3: TAD ListOrd sobre TAD Lista (encadenamiento sencillo y centinela)
4: TAD ListOrd sobre TAD Lista (encadenamiento sencillo y encabezado)
5: TAD ListOrd con lista doblemente encadenada
6: TAD ListOrd con un vector
7: TAD ListOrd una lista sencillamente encadenada con centinela

	1	2	3	4	5	6	7
inicListOrd							
insListOrd							
elimListOrd							
estaListOrd							
longListOrd							
infoListOrd							

Haga también una tabla en la que considere las restricciones de implementación, el espacio ocupado y la dificultad de los algoritmos.

A continuación, se presentan algunos ejercicios en los cuales se pide implementar un TAD específico, representando internamente cada objeto abstracto con elementos del TAD Lista.

2.59. Un polinomio sobre una variable entera se puede ver como una lista de términos, en donde cada uno consta de un coeficiente y un exponente.

$$P(x) = T_0 + T_1 + \ldots + T_n \qquad T_i = c_i \; X^i$$

Un polinomio se puede representar como una lista de coeficientes, donde el valor del exponente va implícito por su posición dentro de la lista. Por ejemplo, el polinomio:

$$P(x) = 12x^5 + 3x^3 - x + 6$$

Se representa con la lista < 6, -1, 0, 3, 0, 12 > donde los términos no presentes se representan con coeficiente 0.

Del TAD Polinomio son interesantes las siguientes operaciones:

evalPolinomio:	Polinomio X int	→ int
sumarPolinomio:	Polinomio X Polinomio	→ Polinomio
multiplicarPolinomio:	Polinomio X Polinomio	→ Polinomio
derivarPolinomio:	Polinomio	→ Polinomio

a. Defina formalmente las operaciones presentadas del TAD Polinomio.

b. Implemente las operaciones del TAD Polinomio utilizando como representación interna listas.

2.60. Otra manera de representar polinomios como listas es utilizando, para cada término, una pareja de la forma (coeficiente, exponente), y, así, colocar explícitamente el valor del exponente de cada elemento. En este caso, el polinomio:

$$P(x) = 12x^5 + 3x^3 - x + 6$$

se representa con la lista < (12, 5), (3, 3), (-1, 1), (6, 0) >, en la cual no necesariamente existe un orden determinado entre sus componentes. Utilizando esta nueva representación resuelva el mismo punto anterior.

2.61. Implemente el TAD String (cadena de caracteres de longitud variable) sobre el TAD Lista. Defina y especifique inicialmente las operaciones; luego, la manera de representar un String como una lista y, por último, escriba un algoritmo que implemente cada operación.

2.62. Un diccionario es una estructura ordenada, en la cual cada palabra tiene asociada una lista no vacía de significados. Especifique el TAD Diccionario e impleméntelo sobre listas.

2.63. Para manejar grandes números (enteros positivos con cualquier cantidad de dígitos) en un programa, resulta muy conveniente poder contar con la definición de este TAD. Piense, por ejemplo, en el problema de calcular 50! que debe tener alrededor de 40 dígitos, o de obtener el resultado de 50^{50}. Especifique el TAD superEntero e impleméntelo sobre listas.

2.64. Para manejar valores de gran precisión, es posible definir el TAD superReal. Los elementos serían de la forma:

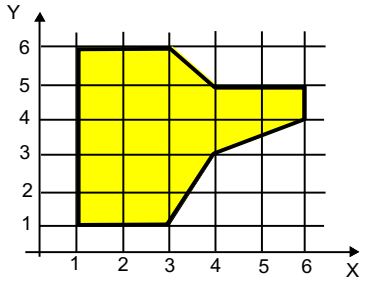
num = [e1 e2 ... en] . [d1 d2 ... dk]
parte entera — parte decimal

Especifique el TAD e impleméntelo sobre listas.

2.65. Se puede ver un texto como una lista de líneas, y una línea como una secuencia de caracteres de longitud variable (String). Aprovechando el TAD String definido en un ejercicio anterior, desarrolle el TAD Texto para manejar este tipo de información. Para desarrollarlo, puede basarse en los comandos de un editor de textos, los cuales dan una idea de las operaciones necesarias para poderlos manipular adecuadamente.

2.66. Un manejador de memoria dinámica debe mantener internamente una lista con las posiciones no utilizadas de la memoria. Esta lista se encuentra compuesta por parejas de la forma [dirección, número de *bytes*], e indican el punto de la memoria donde existe un espacio libre y su tamaño en *bytes* (esta lista está ordenada por tamaño). Desarrolle operaciones como malloc y free para manejar esta estructura

2.67. Un polígono es una secuencia de puntos en un mismo plano del espacio. Cada uno de estos puntos se denomina un vértice:



Para el polígono anterior, la secuencia de puntos es: <(1,1), (3,1), (4,3), (6,4), (6,5), (4,5), (3,6), (1,6) >. Teniendo una lista como representación interna, desarrolle las siguientes rutinas:

a. Calcular y retornar el min-max del polígono. El **min-max** se define como el mínimo rectángulo que incluye el polígono. Para el ejemplo, el min-max es el polígono <(1,1), (6,1), (6,6), (1,6) >.

b. Trasladar todo el polígono en el espacio un (deltaX, deltaY).

c. Dibujar el polígono. Suponga la existencia de una función dibujeLínea que dibuja una recta dados dos puntos del espacio.

2.68. Se puede representar internamente una matriz como una lista de registros, donde cada uno de ellos tiene el número de la fila, el número de la columna y el valor del elemento que se encuentra presente en esa posición. Los elementos no representados explícitamente en las estructuras de datos tienen un valor 0.

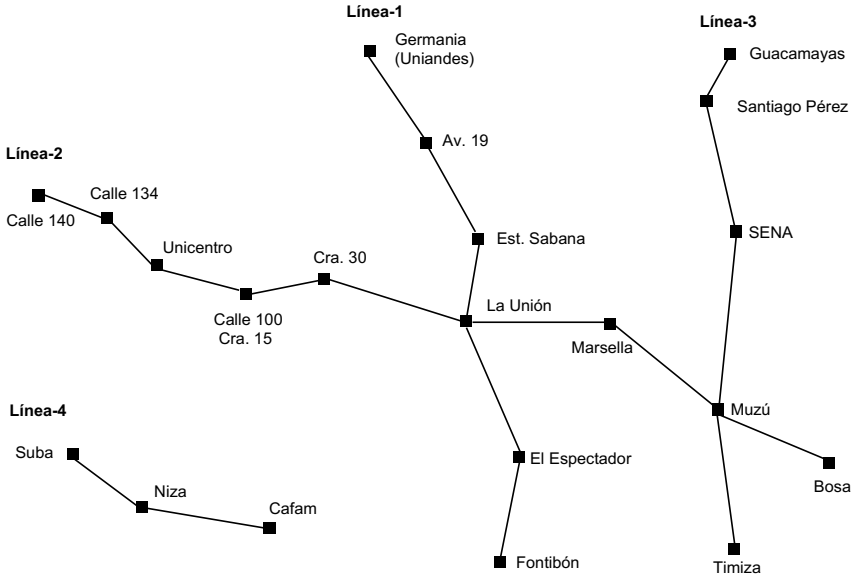
	1	2	3	4
1	4	0	7	0
2	9	0	0	0
3	12	0	1	0
4	0	0	0	2

Por ejemplo, la matriz del dibujo se representa con la lista:

<(1,1,4), (1,3,7), (2,1,9), (3,1,12), (3,3,1), (4,4,2) >

Del TAD Matriz interesa implementar las operaciones inicMatriz, asigneMatriz, infoMatriz y sumeMatriz.

2.69. 🕒🕒 El metro es uno de los sistemas de transporte más utilizado en las grandes ciudades. Está conformado por un conjunto de líneas y cada línea por una secuencia de estaciones.



a. Diseñe e implemente el TAD Línea

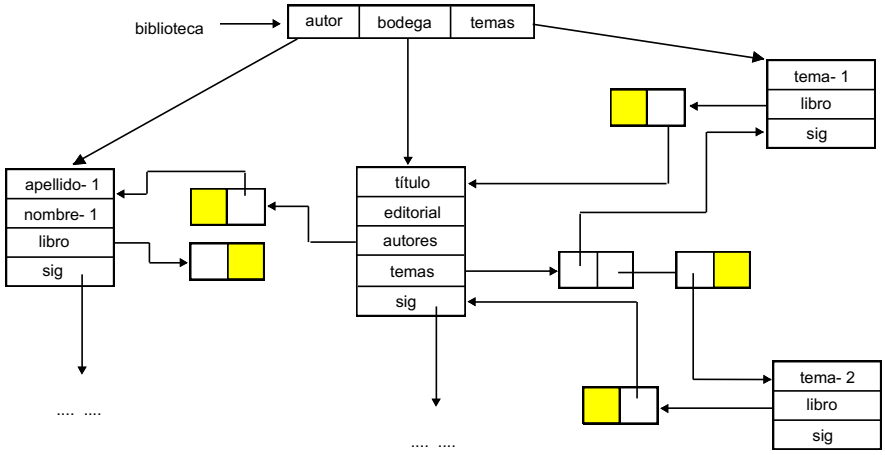
b. Diseñe e implemente el TAD Metro, basándose en el TAD Línea

c. Desarrolle una rutina que, dado un Metro y dos estaciones, retorne una lista con el trayecto que debe seguir un usuario para ir de una a la otra. Suponga que no existe ningún ciclo al interior del Metro.

2.70. El inventario de un almacén se puede ver como una secuencia de tripletas de la forma [nombre-producto, cantidad, precio]. Diseñe e implemente el TAD Almacén, que permite manejar su inventario, utilizando como representación interna el TAD Lista.

2.71. 🕒🕒 Una biblioteca tiene un conjunto de libros, cada uno de los cuales tiene un título, una editorial, uno o varios autores (apellido y nombre) y uno o varios descriptores. Maneja, además, 2 ficheros: uno ordenado por apellido de los autores y otro ordenado por descriptor, los cuales se deben mantener constantemente actualizados.

Diseñe e implemente los TAD FicheroAutor, FicheroTema y Bodega, y, sobre estos, el TAD Biblioteca. Utilice apuntadores como representación interna de todos los TAD. Las estructuras de datos completas deben ser de la siguiente manera:



Cada TAD debe manejar su parte de las estructuras de datos. Por ejemplo, la operación de adicionar un libro a la biblioteca se hace adicionando un libro a la bodega y, luego, autor por autor y tema por tema, utilizando las respectivas operaciones de los TAD Fichero.

2.72. 🕒 En muchas ocasiones es interesante mantener la secuencia de elementos de una lista ordenada por más de un concepto. Por ejemplo, es conveniente poder recorrer la lista de un curso en orden alfabético, en orden de carnet o en orden de nota, con algoritmos O(n).

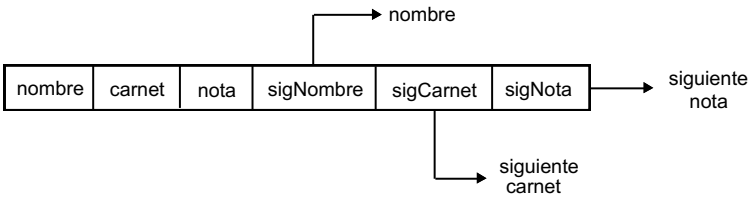
Sobre estas estructuras se quiere desarrollar 7 operaciones:

a. insertar un nuevo estudiante (nombre, carnet, nota)
b. eliminar un estudiante (carnet)
c. listar el curso por orden alfabético
d. listar el curso por orden de carnet
e. listar el curso ascendentemente por nota
f. dar la nota de un estudiante dado su nombre
g. dar la nota de un estudiante dado su carnet

Formalice la definición del TAD ListaCurso e impleméntelo sobre las estructuras de datos que se sugieren a continuación. Compare las 3 implementaciones propuestas.

1. Listas **replicadas**: 3 listas de registros (nombre, carnet, nota) cada una ordenada por un concepto.

2. Listas **multiencadenadas**: estructura con triple sucesor, uno por cada orden.



3. Listas **invertidas**: estructura doblemente encadenada de registros (nombre, carnet, nota) ordenada por nombre y dos listas de apuntadores hacia la estructura principal, ordenadas por carnet y nota respectivamente.

Bibliografía

- [AHO83] Aho, A., Hopcroft, J., Ullman, J., "Data Structures and Algorithms", Cap. 2, Addison-Wesley, 1983.
- [BER94] Bergin, J., "Data Abstraction: The Object-Oriented Approach Using C++", Cap. 6, McGraw-Hill, 1994.
- [DAL86] Dale, N., Lilly, S., "Pascal y Estructura de Datos", Cap. 5, McGraw-Hill, 1986.
- [DRO82] Dromey, R., "How to Solve it by Computer", Cap. 7, Prentice-Hall, 1982.
- [ESA89] Esakov, J., Weiss, T., "Data Structures: An Advanced Approack Using C", Cap. 4, Cap. 6, Prentice-Hall, 1989.
- [FEL88] Feldman, M., "Data Structures with Modula-2", Cap. 4, Prentice-Hall, 1988.
- [HOR83] Horowitz, E., "Fundamentals of Data Structures", Cap. 4, Computer Science Press, 1983.
- [KNU73] Knuth, D., "The Art of Computer Programming. Vol. 1 - Fundamental Algorithms", 2da edición, Addison-Wesley, 1973.
- [KRU87] Kruse, R., "Data Structures & Program Design", Cap.4, Prentice-Hall, 1987.
- [LIP87] Lipschutz, S., "Estructura de Datos", Cap. 5, McGraw-Hill, 1987.
- [MAR86] Martin, J., "Data Types and Data Structures", Cap. 6, Prentice-Hall, 1986.
- [STU85] Stubbs, D., Webre, W., "Data Structures with Abstract Data Types and Pascal", Brooks/Cole Publishing Company, 1985.
- [TEN93] Tenenbaum, A., Langsam, Y., "Estructuras de Datos en C", Cap. 4, Prentice Hall, 1993.
- [TRE76] Tremblay, J., Sorenson, P., "An Introduction to Data Structures with Applications", Cap. 3, McGraw-Hill, 1976.
- [WEL84] Welsh, J., Elder, J., "Sequential Program Structures", Prentice-Hall, 1984.
- [WIR76] Wirth, N., "Algorithms + Data Structures = Programs", Cap. 4, Prentice-Hall,1976.
- [WIR86] Wirth, N., "Algorithms & Data Structures", Cap. 4, Prentice-Hall, 1986.

CAPITULO 3

ESTRUCTURAS LINEALES: PILAS Y COLAS

En este capítulo se presentan unas estructuras lineales de datos, de comportamiento y uso más restringidos que las listas, pero de amplio uso en procesos de simulación, entre las que se encuentran las pilas, las colas, las colas de prioridad, las bicolas y las rondas.

3.1. Pilas: Definiciones y Conceptos Básicos

Una **pila** (*stack*) es una secuencia de cero o más elementos de un mismo tipo, que solamente puede crecer y decrecer por uno de sus extremos (fig. 3.1). Se puede ver como un caso particular de una lista, en el cual la ventana se mantiene estática en la primera posición y las operaciones posibles se restringen, permitiendo el acceso a la estructura únicamente por ese punto.

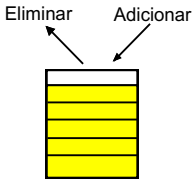


Fig. 3.1- Operaciones sobre una pila

La pilas se denominan también estructuras **LIFO** (*Last-In-First-Out*), porque su característica principal es que el último elemento en llegar es el primero en salir. Son muy utilizadas en programación, para evaluar expresiones, reconocer lenguajes, recorrer árboles y simular procesos recursivos. En todo momento, el único elemento visible de la estructura es el último que se colocó. Se define el **tope** de la pila como el punto donde se encuentra dicho elemento, y el **fondo**, como el punto donde se encuentra el primer elemento incluido en la estructura (fig. 3.2).

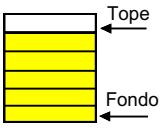


Fig. 3.2- Tope y fondo de una pila

El formalismo escogido para referirse al objeto abstracto pila se muestra a continuación. Allí se da un nombre a cada uno de los elementos que hacen parte de la estructura, y se marca claramente el tope y el fondo:

e1 e2 ... en

Si la pila no tiene ningún elemento se dice que se encuentra **vacía** y no tiene sentido referirse a su tope ni a su fondo. Una pila vacía se representa con el símbolo Ø. Por último, se define la **longitud** de una pila como el número de elementos que la conforman.

3.2. El TAD Pila

La administración de una pila se puede hacer con muy pocas operaciones: una constructora (permite crear pilas vacías), dos modificadoras (para agregar y eliminar elementos) y dos analizadoras (retornar el elemento del tope, e informar si la pila está vacía). Se incluye, además, una destructora para retornar el espacio ocupado por la pila. Por simplicidad, no se contemplan operaciones de persistencia.

TAD Pila[TipoP]		
	e1 e2 ... en	
{ inv: TRUE }		
Constructoras:		
• inicPila:		→ Pila
Modificadoras:		
• adicPila:	Pila x TipoP	→ Pila
• elimPila:	Pila	→ Pila
Analizadoras:		
• infoPila:	Pila	→ TipoP
• vaciaPila:	Pila	→ int
Destructora:		
• destruirPila:	Pila	

```
Pila inicPila( void )
/* Crea una pila vacía */

{ post: inicPila = ∅ }
```

```
void adicPila( Pila pil, TipoP elem )
/* Coloca sobre el tope de la pila el elemento elem */

{ post: pil = elem e1 e2 ... en }
```

```
void elimPila( Pila pil )
/* Elimina el elemento que se encuentra en el tope de la pila */

{ pre: pil = e1 e2 ... en, n > 0 }
{ post: pil = e2 ... en }
```

```
TipoP infoPila( Pila pil )
/* Retorna el elemento que se encuentra en el tope de la pila */

{ pre: n > 0 }
{ post: infoPila = e1 }
```

```
int vaciaPila( Pila pil )
/* Informa si la pila es vacía */

{ post: vaciaPila = ( pil = ∅ ) }
```

```
void destruirPila( Pila pil )
/* Destruye la pila retornando toda la memoria ocupada */

{ post: pil ha sido destruida }
```

3.3. Ejemplos de Utilización del TAD Pila

En esta sección se presentan algunos ejemplos de solución de problemas utilizando el TAD Pila. Los usos más importantes se verán en capítulos posteriores, en particular, como soporte al recorrido de estructuras de datos recursivas.

Ejemplo 3.1:

Invertir una lista utilizando una pila. La complejidad de esta rutina es O(n), donde n es el número de componentes de la lista.

```
/* pre: lst = < x1, ..., xN > */
/* post: lst = < xN, ..., x1 > */

void invLista( Lista lst )
{   Pila pil = inicPila( );
    for( primLista( lst ); !finLista( lst ); elimLista( lst ) )
        adicPila( pil, infoLista( lst ) );
    for( ; !vaciaPila( pil ); elimPila( pil ) )
        anxLista( lst, infoPila( pil ) );
    destruirPila( pil );
}
```

El primer ciclo hace un recorrido de la lista y va pasando los elementos a una pila, con el siguiente invariante:

{ inv_1: lst = < xk , ..., x_N >, pil = xk-1 ... x1 }

Al final de dicho proceso, en el tope se encuentra el último elemento y en el fondo el primero, de tal manera que si se sacan uno a uno los elementos de la pila y se van colocando de nuevo en la lista se obtiene, al final, la secuencia inicial invertida, como ilustra el invariante del segundo ciclo:

{ inv_2: lst = < x_N, ..., xk >, pil = xk-1 ... x1 }

Ejemplo 3.2:

Copiar una pila. Esta rutina debería ser una operación del TAD, puesto que su implementación sobre las operaciones existentes es un poco dispendiosa, aunque perfectamente posible. La complejidad de esta función es O(n), donde n es la longitud de la pila.

```
/* pre: pil = PIL = x1 ... xN */
/* post: copiarPila = x1 ... xN, pil = PIL */

Pila copiarPila( Pila pil )
{   Pila resp = inicPila( );
    Lista lst;
    for( lst = inicLista( ); !vaciaPila( pil ); elimPila( pil ) )
        insLista( lst, infoPila( pil ) );
    for( primLista( lst ); !finLista( lst ); elimLista( lst ) )
    {   adicPila( pil, infoLista( lst ) );
        adicPila( resp, infoLista( lst ) );
    }
    destruirLista( lst );
    return resp;
}
```

El primer ciclo, pasa a la lista temporal lst todos los elementos de la pila pil, dejándolos en el orden inverso, tal como aparece en la siguiente aserción:

{ A1: lst = < x_N, ..., x_1 >, pil = ∅ }

El invariante de este ciclo es:

{ inv1: lst = < xk , ... , x_1 >, pil = xk+1 ... xN }

El segundo ciclo, recorre la lista agregando sus elementos a dos pilas: la pila original pil, puesto que se debe reconstruir, y la pila de respuesta resp. El invariante de este proceso es:

{ inv2: lst = < xk , ... , x_1 >, pil = xk+1 ... xN, resp = xk+1 ... xN }

⏎

Ejemplo 3.3:

Invertir el contenido de una pila. La complejidad de esta rutina es O(n), donde n es el número de elementos de la estructura.

```
/* pre: pil = x1 ... xN */
/* post: pil = xN ... x1 */
```

```
void invPila( Pila pil )
{   Pila aux = copiarPila( pil );
    destruirPila( pil );
    for( pil = inicPila( ); !vaciaPila( aux ); elimPila( aux ) )
        adicPila( pil, infoPila( aux ) );
    destruirPila( aux );
}
```

El invariante del ciclo asegura que cuando se hayan sacado de la pila auxiliar los elementos x_1 a x_k, éstos ya estarán situados en la pila original en orden inverso:

```
{ inv:  aux = xk+1 ... xN , pil = xk ... x1 }
```
⏎

Ejemplo 3.4:

Decidir si dos pilas son iguales sin destruir su contenido. Esta rutina también debería hacer parte de las operaciones del TAD Pila. Su implementación resulta poco natural, pero se utiliza en este ejemplo para ilustrar el uso de las operaciones.

```
/* pre: pil1 = x1 ... xN , pil2 = y1 ... yM */
/* post: igualesPilas = ( N = M ∧ xk = yk ∀ k ≤ N */

int igualesPilas( Pila pil1, Pila pil2 )
{   Pila aux1 = copiarPila( pil1 );
    Pila aux2 = copiarPila( pil2 );
    while( !vaciaPila( aux1 ) && !vaciaPila( aux2 ) )
    {     if ( infoPila( aux1 ) != infoPila( aux2 ) )
          {     destruirPila( aux1 );
                destruirPila( aux2 );
                return FALSE;
          }
          elimPila( aux1 );
          elimPila( aux2 );
    }
    if ( vaciaPila( aux1 ) && vaciaPila( aux2 ) )
          return TRUE;
    else
    {     destruirPila( aux1 );
          destruirPila( aux2 );
          return FALSE;
    }
}
```

La rutina trabaja sobre una copia de las pilas, las cuales debe ir destruyendo a medida que verifica que son iguales. En el momento en el cual encuentra dos elementos que no corresponden, abandona la rutina después de destruir el resto de las pilas de trabajo. El invariante afirma que cuando en el tope de cada una de las pilas esté el k-ésimo elemento, todos los que estaban antes han sido iguales uno a uno.

```
{ inv: aux1 = xk ... xN , aux2 = yk ... yM , x1 = y1, ..., xk-1 = yk-1 }
```
⏎

Ejemplo 3.5:

Evaluar una expresión aritmética en **notación postfija**. En esta notación, se colocan los operadores después de los dos operandos. Tiene la ventaja de no ser ambigua, algo que no ocurre con la notación que se utiliza comúnmente para expresiones aritméticas (**notación infija**). Por ejemplo, si se tiene la expresión infija:

3 + 5 * 10

El resultado de su evaluación puede ser 80 ó 53, dependiendo del orden en el cuál los operadores tomen sus operandos:

(3 + 5) * 10 = 80

3 + (5 * 10) = 53

En postfijo, cada una de estas interpretaciones posibles tiene una única expresión:

3 5 + 10 * (es 80)

3 5 10 * + (es 53)

En este ejemplo, se va a suponer que la entrada que se va a procesar es una lista de cadenas de caracteres y que los únicos operadores presentes son +, -, *, /. Para el caso ilustrado anteriormente, la lista que representa la expresión es < "3" "5" "+" "10" "*" >.

El método de solución consiste en ir guardando en una pila de enteros los operandos, y, en el momento de encontrar un operador, aplicarlo sobre los dos valores que se encuentren más arriba de la pila, reemplazándolos por dicho resultado. Por ejemplo, para la expresión postfija:

< "12", "4", "-", "5", "3", "+", "*" >

El proceso de evaluación es el siguiente:

Expresión	Elemento por procesar	Pila
< "12", "4", "-", "5", "3", "+", "*" >	12	∅
< "4", "-", "5", "3", "+", "*" >	4	12
< "-", "5", "3", "+", "*" >	-	4 12
< "5", "3", "+", "*" >	5	8
< "3", "+", "*" >	3	5 8
< "+", "*" >	+	3 5 8
< "*" >	*	8 8
< >		64

El siguiente algoritmo realiza el proceso de evaluación descrito anteriormente.

/* pre: exp = < e_1, ..., e_n >, representa una expresión válida en notación postfija, e_k es de tipo (char *) */

/* post: eval es la evaluación de la expresión */

```
int eval( Lista exp )
{   char *elem;
    int op1, op2;
    Pila pil = inicPila(  );                          /* Pila de enteros para la evaluación */
    for( primLista( exp ); !finLista( exp ); sigLista( exp ) )
    {     elem = infoLista( exp );
          if ( isdigit( elem[ 0 ] ) )
                adicPila( pil, atoi( elem ) );        /* Agrega el valor después de convertirlo */
          else
          {     op1 = infoPila( pil );              /* Toma los dos operandos de la pila */
                elimPila( pil );
                op2 = infoPila( pil );
                elimPila( pil );
                switch ( elem[ 0 ] )
                { case '+' : adicPila( pil, op2 + op1 );
                             break;
                  case '-' : adicPila( pil, op2 - op1 );
                             break;
                  case '*' : adicPila( pil, op2 * op1 );
                             break;
                  case '/' :  adicPila( pil, op2 / op1 );
                             break;
                }
          }
    }
    return infoPila( pil );
}
```

Esta función utiliza las rutinas *isdigit* y *atoi* de la librería estándar *ctype* de C, para determinar si un caracter es un dígito, y para convertir una cadena de caracteres en un entero.

Ejercicios Propuestos

Especifique formalmente y desarrolle las siguientes rutinas:

3.1. void impPila(Pila pil)
/* Imprime el contenido de una pila, sin cambiar su contenido */

3.2. void fondoPila(Pila pil, TipoP elem)
/* Coloca en el fondo de la pila el elemento elem */

3.3. int longPila(Pila pil)
/* Calcula el número de elementos de la pila, sin modificar su contenido */

3.4. int sumePila(Pila pil)
/* Suma todos los elementos de la pila y retorna el resultado */

3.5. void elimTodosPila(Pila pil, Tipo elem)
/* Elimina de la pila todas las ocurrencias del elemento elem */

3.6. int palindromePila(Pila pil)
/* Indica si el contenido de la pila es un palíndrome */

3.7. void intercambioPila(Pila pil)
/* Intercambia los valores del tope y el fondo de la pila */

3.8. void duplicaPila(Pila pil)
/* Duplica el contenido de la pila, dejando el doble de elementos */

3.9. int sintaxisExp(Lista exp)
/* Informa si una expresión en notación postfija está bien construida */

3.10. void convertir1(Lista infija, Lista postfija)
/* Pasa una expresión aritmética de notación infija (con todos los paréntesis) a notación postfija */

3.11. void convertir2(Lista postfija, Lista infija)
/* Pasa una expresión aritmética de notación postfija a notación infija (con todos los paréntesis) */

3.12. Implemente un algoritmo para evaluar una expresión en notación infija, sin convertirla a notación postfija. Utilice dos pilas, una para operandos y otra para operadores.

3.13. Modifique el algoritmo de conversión de notación de infija a postfija (propuesto en el ejercicio 3.10), suponiendo que la expresión no trae paréntesis completos, pero sabiendo que la prioridad de los operadores está dada de mayor a menor por el siguiente orden: *, /, +, -.

3.14. Se define el lenguaje L como el conjunto de palabras obtenidas al aplicar las siguientes reglas sintácticas:

< palabra > ::= < letra >
| "(" < palabra > < palabra > ")"
< letra > ::= A | B | ... | Z

Según lo anterior, una palabra del lenguaje L puede ser, o una letra o una construcción entre paréntesis compuesta por otras dos palabras del lenguaje. Por ejemplo, las siguientes palabras pertenecen al lenguaje:

A
(A B)
((A B) (C D))
((A B) ((C D) (E F)))

Desarrolle un algoritmo que, dada una lista de caracteres, indique si la secuencia pertenece o no al lenguaje.

3.4. Implementación del TAD Pila

En esta sección se presentan tres implementaciones -muy sencillas- para el Tipo Abstracto Pila.

3.4.1. Listas

En esta implementación se utiliza un objeto abstracto del TAD Lista para representar una pila. El esquema de representación es el siguiente:

- La pila pil = $\overline{e1\ \ e2\ ...\ en}$ se representa con la lista < e_1, e_2 ... e_N >.
- La pila vacía (pil = ∅) se representa internamente como una lista sin elementos (pil = < >).

La declaración de las estructuras de datos es:

```
typedef TipoP TipoL;
typedef Lista Pila;
```

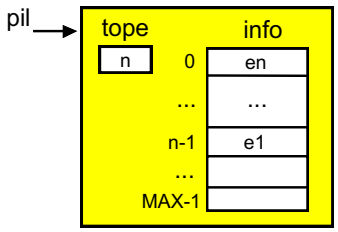
Ahora, es suficiente con expresar las operaciones del TAD Pila en términos de las operaciones del TAD Lista, como se hace a continuación:

```
Pila inicPila( void )
{   return inicLista( );
}

void adicPila( Pila pil, TipoP elem )
{   primLista( pil );
    insLista( pil, elem );
}

void elimPila( Pila pil )
{   primLista( pil );
    elimLista( pil );
}

TipoP infoPila( Pila pil )
{   primLista( pil );
    return infoLista( pil );
}

int vaciaPila( Pila pil )
{   return longLista( pil ) == 0;
}

void destruirPila( Pila pil )
{  destruirLista( pil );
}
```

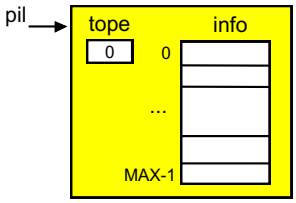
3.4.2. Vectores

Un vector es una buena manera de representar una pila, si se conoce con anterioridad el número máximo de elementos que va a contener. Sólo se necesita algún medio para marcar el tope de la pila, puesto que los elementos se colocan en casillas consecutivas a partir de la primera. El esquema de representación es el siguiente:

- La pila pil = e1 e2 ... en se representa con la estructura:



- La pila vacía pil = ∅ se representa con un cero en el campo que indica la posición del tope:



Las estructuras de datos se declaran de la siguiente manera:

```
typedef struct
{   int tope;
    TipoP info[ MAX ];
} TPila, *Pila;
```

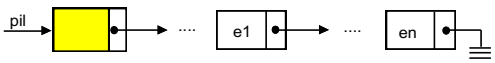
Las funciones que implementan las operaciones del TAD Pila bajo esta representación, son:

```
Pila inicPila( void )
{   Pila pil = ( Pila )malloc( sizeof( TPila ) );
    pil->tope = 0;
    return pil;
}

void adicPila( Pila pil, TipoP elem )
{   pil->info[ pil->tope++ ] = elem;
}

void elimPila( Pila pil )
{   pil->tope--;
}

TipoP infoPila( Pila pil )
{   return pil->info[ pil->tope - 1 ];
}

int vaciaPila( Pila pil )
{   return pil->tope == 0;
}

void destruirPila( Pila pil )
{   free( pil );
}
```

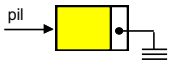
3.4.3. Estructura Sencillamente Encadenada

En esta representación se utilizan listas sencillamente encadenadas para almacenar la información de una pila, con el siguiente esquema:

- La pila pil = $\overline{\text{e1 e2 ... en}}$ se representa mediante una lista encadenada, con un nodo de encabezado:



- Una pila vacía pil = ∅ tiene únicamente el nodo de encabezado:



Las declaraciones de las estructuras de datos son de la siguiente forma:

```
typedef struct TNodoPila
{   TipoP info;
    struct TNodoPila *sig;
} TPila, *Pila;
```

Las rutinas que implementan las operaciones del TAD Pila bajo esta representación son las siguientes:

```
Pila inicPila( void )
{   Pila pil = ( Pila )malloc( sizeof( TPila ) );
    pil->sig = NULL;
    return pil;
}

void adicPila( Pila pil, TipoP elem )
{   Pila p = ( Pila )malloc( sizeof( TPila ) );
    p->info = elem;
    p->sig = pil->sig;
    pil->sig = p;
}

void elimPila( Pila pil )
{   Pila p = pil->sig;
    pil->sig = pil->sig->sig;
    free( p );
}

TipoP infoPila( Pila pil )
{   return pil->sig->info;
}

int vaciaPila( Pila pil )
{ return pil->sig == NULL;
}

void destruirPila( Pila pil )
{   Pila p, q;
    for( p = pil;  p != NULL;  )
    {       q = p;
            p = p->sig;
            free( q );
    }
}
```

Ejercicios Propuestos:

3.15. Haga una comparación detallada de las tres implementaciones del TAD Pila presentadas anteriormente.

3.16. ¿Por qué es necesario tener un encabezado en la implementación de apuntadores? Dé un ejemplo.

3.17. Implemente sobre las tres representaciones vistas una rutina que coloque un elemento en el fondo de la pila. Calcule la complejidad de la operación.

3.18. Implemente sobre las tres representaciones vistas una rutina que haga una copia de una pila. Calcule la complejidad de la operación.

3.19. Implemente sobre las tres representaciones vistas una rutina que informe si dos pilas son iguales. Calcule la complejidad de la operación.

3.20. 💻 Implemente sobre las tres representaciones vistas un esquema de persistencia parecido al que se utilizó en el capítulo de listas.

3.5. Colas: Definiciones y Conceptos Básicos

Una **cola** (*queue*) es una estructura lineal, en la cual los elementos sólo pueden ser adicionados por uno de sus extremos y eliminados o consultados por el otro. El ejemplo típico de una cola es la fila de espera que se hace mientras se espera atención en alguna parte. Al llegar un nuevo elemento se coloca al final (después del último) y espera a que atiendan y salgan de la fila todos los que llegaron antes. No existe manera de llegar a un puesto diferente del último, ni esperar ser atendido y salir de la fila mientras se esté en un puesto distinto del primero.

Fig. 3.3 - Operaciones sobre una cola

Las colas se utilizan mucho en los procesos de simulación, en los cuales se quiere determinar el comportamiento de un sistema que presta servicio a un conjunto de usuarios, quienes esperan mientras les toca el turno de ser atendidos. Como ejemplos de estos sistemas se pueden nombrar los bancos, los aeropuertos (los aviones hacen cola para despegar y aterrizar) y los procesos dentro de un computador. Las colas también se utilizan en muchos algoritmos de recorrido de árboles y grafos.

Este tipo de estructuras lineales se conoce en la literatura como estructuras **FIFO** (First-In-First-Out), indicando con su nombre el mecanismo básico utilizado para incluir y eliminar un elemento: el primero en llegar es el primero en salir. El único elemento visible en una cola es el primero y mientras éste no haya salido, no es posible tener acceso al siguiente.

El formalismo escogido para expresar el estado de un objeto abstracto Cola es:

$$\overline{x1\ x2\ \ldots\ xn}\leftarrow$$

La cola sin elementos se representa con el símbolo ∅.

Por último, se define la **longitud** de una cola como el número de elementos que la conforman. Si la longitud es cero (no tiene ningún elemento), se dice que la cola está **vacía**.

3.6. El TAD Cola

TAD Cola[TipoC]		
	$\overline{x1\ x2\ ...\ xn}\leftarrow$	
{ inv: TRUE }		
Constructoras:		
• inicCola:		→ Cola
Modificadoras:		
• adicCola:	Cola x TipoC	→ Cola
• elimCola:	Cola	→ Cola
Analizadoras:		
• infoCola:	Cola	→ TipoC
• vaciaCola:	Cola	→ int
Destructora:		
• destruirCola:	Cola	

```
Cola inicCola( void )
/* Crea una cola vacía */

{ post: inicCola = ∅ }
```

```
void adicCola( Cola col, TipoC elem )
/* Agrega el elemento elem al final de la cola */

{ post: col = x1 x2 ... xn elem← }
```

```
void elimCola( Cola col )
/* Elimina el primer elemento de la cola */

{ pre: n > 0 }
{ post: col = x2 ... xn← }
```

```
TipoC infoCola( Cola col )
/* Retorna el primer elemento de la cola */

{ pre: n > 0 }
{ post: infoCola = x1 }
```

```
int vaciaCola( Cola col )
/* Informa si la cola está vacía */

{ post:  vaciaCola = ( col = ∅ ) }
```

```
void destruirCola( Cola col )
/* Destruye la cola retornando toda la memoria ocupada */

{ post: la cola ha sido destruida }
```

3.7. Ejemplos de Utilización del TAD Cola

Ejemplo 3.6:

Hacer y retornar una copia de una cola. Esta operación debería colocarse como parte del TAD. La complejidad de esta rutina es O(n), donde n es la longitud de la cola.

```
/* pre: col = COL = x1 x2 ... xn ← */
/* post: col = COL, copiarCola = x1 x2 ... xn ← */

Cola copiarCola( Cola col )
{   Cola resp = inicCola( );
    Lista lst;
    for( lst = inicLista( ); !vaciaCola( col ); elimCola( col ) )
        anxLista( lst, infoCola( col ) );
    for( primLista( lst ); !finLista( lst ); sigLista( lst ) )
    {   adicCola( col,infoLista( lst ) );
        adicCola( resp,infoLista( lst ) );
    }
    destruirLista( lst );
    return resp;
}
```

La rutina utiliza dos ciclos. El primero, para pasar todos los elementos de la cola a una lista temporal. El otro, para hacer dos copias de la lista.

Ejemplo 3.7:

Calcular la longitud de una cola. La complejidad de esta rutina es O(n), donde n es el número de elementos.

```
/* pre: col = x1 x2 ... xn ← */
/* post: longCola = n */

int longCola( Cola col )
{   Cola aux = copiarCola( col );
    int longitud;
    for( longitud = 0; !vaciaCola( aux ); elimCola( aux ) )
        longitud++;
    destruirCola( aux );
    return longitud;
}
```

La rutina se limita a hacer una copia de la cola, y destruirla a medida que va contando sus elementos. El invariante del ciclo afirma, que cuando se hayan sacado k elementos de la cola auxiliar, la variable longitud valdrá k:

{ inv: aux = $\overline{xk \ldots xn}\leftarrow$, longitud = k-1 }

⏎

Ejemplo 3.8:

Concatenar dos colas, dejando el resultado en la primera de ellas. La complejidad de esta rutina es O(n), donde n es la longitud de la segunda cola.

{ pre: col1 = $\overline{x1\ x2 \ldots xn}\leftarrow$, col2 = COL2 = $\overline{y1\ y2 \ldots ym}\leftarrow$ }
{ post: col1 = $\overline{x1\ x2 \ldots xn\ y1\ y2 \ldots ym}\leftarrow$, col2 = COL2 }

```
void concatColas( Cola col1, Cola col2 )
{   Cola aux;
    for( aux = copiarCola( col2 ); !vaciaCola( aux ); elimCola( aux ) )
        adicCola( col1, infoCola( aux ) );
    destruirCola( aux );
}
```

El invariante asegura que cuando se hayan eliminado los k-1 primeros elementos de la cola auxiliar, éstos se habrán agregado al final de la cola col1:

{ inv: col1 = $\overline{x1\ x2 \ldots xn\ y1\ y2 \ldots yk\text{-}1}\leftarrow$, aux = $\overline{yk \ldots ym}\leftarrow$ }

⏎

Ejercicios Propuestos

3.21. void invCola(Cola col)
/* Invierte los elementos de la cola */

3.22. int existeElemento(Cola col, TipoC elem)
/* Informa si el elemento elem se encuentra presente en la cola col */

3.23. int igualesColas(Cola col1, Cola col2)
/* Informa si las colas col1 y col2 tienen los mismos elementos, en el mismo orden */

3.24. void colarElemento(Cola col, TipoC elem, int pos)
/* Agrega el elemento elem en la posición pos de la cola, desplazando todos los elementos siguientes una posición hacia el final */

3.25. void sacarElemento(Cola col, TipoC elem)
/* Saca el elemento elem de la cola col */

3.26. void partirCola(Cola col, Cola col1, Cola col2, TipoC elem)
/* Deja en la cola col1 todos los elementos de col menores que elem y en la cola col2 los mayores a dicho elemento */

3.27. void primeroCola(Cola col, TipoC elem)
/* Coloca el elemento elem de primero en la cola */

3.8. Implementación del TAD Cola

En esta parte se presentan tres maneras distintas de implementar el TAD Cola. La primera con una lista, la segunda, utilizando una estructura de datos llamada vector circular y la tercera, mediante apuntadores.

3.8.1. Listas

En esta primera implementación se representa una cola con un objeto abstracto del TAD Lista, con el esquema que se sugiere a continuación:

- La cola col = $\overline{\text{x1 x2 ... xn}}\leftarrow$ se representa con la lista de n elementos col = < x1 x2 ... xn >.
- La cola vacía col = ∅, se representa con una lista sin elementos col = < >.

Las estructuras de datos se declaran como:

```
typedef TipoC TipoL;
typedef Lista Cola;
```

Las operaciones del TAD se implementan con las siguientes rutinas:

```
Cola inicCola( void )
{   return inicLista( );
}

void adicCola( Cola col, TipoC elem )
{   ultLista( col );
    anxLista( col, elem );
}

void elimCola( Cola col )
{   primLista( col );
    elimLista( col );
}

TipoC infoCola( Cola col )
{   primLista( col );
    return infoLista( col );
}

int vaciaCola( Cola col )
{   return longLista( col ) == 0;
}

void destruirCola( Cola col )
{   destruirLista( col );
}
```

3.8.2. Vectores Circulares

Si se representa una cola con un vector, las rutinas que implementan las operaciones del TAD tienen la siguiente complejidad:

inicCola	O(1)
adicCola	O(1)
elimCola	O(N)
infoCola	O(1)
vaciaCola	O(1)

La ineficiencia en la operación que elimina un elemento (elimCola - O(n)) se debe a la necesidad de desplazar todos los elementos de la estructura, para ocupar el lugar liberado después de sacar el primero. Una posibilidad para evitar este movimiento es marcar los lugares dentro del arreglo donde comienza y termina la cola. Para esto se pueden colocar dos campos extra indicando las casillas en las cuales se encuentran los elementos primero y último, de tal forma que sólo el espacio comprendido entre estas dos marcas se halle ocupado por los elementos de la cola, como se muestra en la figura 3.4.

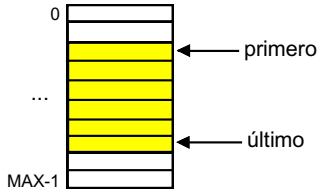


Fig. 3.4 - Marcas en un vector para simular una cola

Así, eliminar un elemento equivale a mover la marca de comienzo de la cola una posición, sin necesidad de hacer ningún desplazamiento, lo cual es O(1). El problema de esta solución es el espacio desperdiciado, puesto que las casillas del vector, anteriores a la marca de comienzo, nunca serían reutilizadas. Más aún, se puede dar el caso de no poder adicionar un elemento a la cola por no haber sitio al final (último = MAX-1), a pesar de tener las primeras posiciones del arreglo desocupadas (primero > 0). Para obviar estos inconvenientes es posible ver el vector como una estructura circular, en la cual, después de utilizar la última casilla del vector, se pueden reutilizar todas las que se encuentran libres al comienzo, como se sugiere en la figura 3.5.

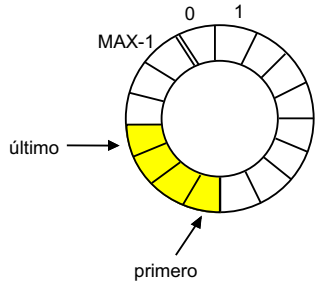
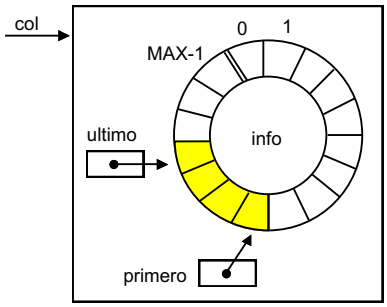


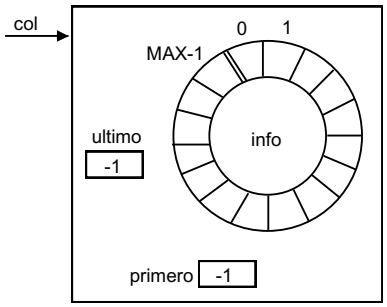
Fig. 3.5 - Estructura de un vector circular

En esta implementación, todas las casillas después de la marca de último y antes de la marca de primero, se encuentran libres y listas para ser utilizadas. El esquema de representación se resume en los siguientes puntos:

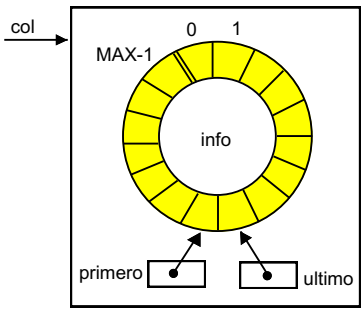
- La cola col = $\overleftarrow{x1\ x2\ ...\ xn}$ se representa internamente con el vector circular:



- La cola vacía col = ∅ se representa con valores especiales en las marcas de primero y ultimo:



- Una cola completamente llena tiene siempre la marca de primero una casilla después de la marca de ultimo. Esto es, ((ultimo + 1) % MAX) = primero, donde % representa el operador módulo:



En esta forma de implementar el TAD Cola se tiene el inconveniente de que se debe definir desde un comienzo el número máximo de elementos que puede contener la estructura. Esta es la principal restricción de esta forma de representación. Las estructuras de datos para vectores circulares se declaran así:

```
typedef struct
{   TipoC info[ MAX ]       /* Vector circular */
    int primero;            /* Posición del primer elemento */
    int ultimo;             /* Posición del último elemento */
} TCola, *Cola;
```

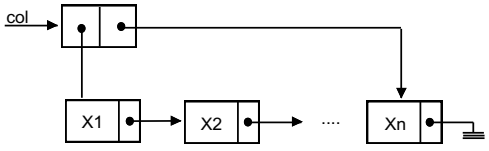
Las operaciones del TAD se implementan con las siguientes rutinas:

```
Cola inicCola( void )
{   Cola col = ( Cola )malloc( sizeof( TCola ) );
    col->primero = col->ultimo = -1;
    return col;
}

void adicCola( Cola col, TipoC elem )
{   if ( col->primero == -1 )
    {     col->info[ 0 ] = elem;
          col->primero = col->ultimo = 0;
    }
    else
    {     col->ultimo = ( col->ultimo + 1 ) % MAX;
          col->info[ col->ultimo ] = elem;
    }
}

void elimCola( Cola col )
{   if ( col->primero == col->ultimo )
          col->primero = col->ultimo = -1;
    else
          col->primero = ( col->primero + 1 ) % MAX;
}

TipoC infoCola( Cola col )
{   return col->info[ col->primero ];
}

int vaciaCola( Cola col )
{   return col->primero == -1 && col->ultimo == -1;
}

void destruirCola( Cola col )
{   free( col );
}
```

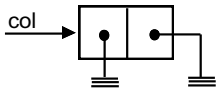
3.8.3. Estructura Sencillamente Encadenada

En esta implementación, se va a representar una cola como un registro con dos campos de tipo apuntador: uno al primer elemento de una estructura simplemente encadenada y, otro, al último de ellos. El esquema de representación se define en los siguientes puntos:

- La cola col = $\overline{x1\ x2\ ...\ xn}\leftarrow$ se representa internamente como:



- Una cola vacía col =∅, se representa con un registro con sus dos campos en NULL:



La declaración de las estructuras de datos para esta implementación es de la siguiente forma:

```
typedef struct TNodoCola
{   TipoC info;
    struct TNodoCola *sig;
} *pNodoCola;

typedef struct
{   pNodoCola primero, ultimo;
} TCola, *Cola;
```

Las operaciones del TAD se implementan con rutinas de complejidad constantes, tal como se muestra a continuación:

```
Cola inicCola( void )
{   Cola col = ( Cola )malloc( sizeof( TCola ) );
    col->primero = col->ultimo = NULL;
    return col;
}

void adicCola( Cola col, TipoC elem )
{   pNodoCola p = ( pNodoCola )malloc( sizeof( struct TNodoCola ) );
    p->info = elem;
    p->sig = NULL;
    if( col->primero == NULL )
        col->primero = col->ultimo = p;
    else
    {   col->ultimo->sig = p;
        col->ultimo = p;
    }
}

void elimCola( Cola col )
{   pNodoCola p;
    if( col->primero->sig == NULL )
    {   free( col->primero );
        col->primero = col->ultimo = NULL;
    }
    else
    {   p = col->primero;
        col->primero = p->sig;
        free( p );
    }
}

TipoC infoCola( Cola col )
{   return col->primero->info;
}
```

```
int vaciaCola( Cola col )
{ return col->primero == NULL;
}

void destruirCola( Cola col )
{   pNodoCola p, q;
    for( p = col->primero; p != NULL; p = p->sig, free( q ) )
        q = p;
    free( col );
}
```

Ejercicios Propuestos

3.28. Haga una comparación de las tres implementaciones del TAD Cola presentadas anteriormente.

3.29. 💻 Implemente el TAD Cola con una estructura circular simplemente encadenada. Calcule la complejidad de las operaciones.

3.30. 💻 Implemente el TAD Cola sobre el TAD Pila. Calcule la complejidad de las operaciones.

3.31. Desarrolle sobre las tres representaciones vistas una rutina que coloque un elemento como primero de la cola. Calcule la complejidad de la operación.

3.32. Desarrolle sobre las tres representaciones vistas una rutina que cuente el número de ocurrencias de un elemento en la cola. Calcule la complejidad de la operación.

3.33. Desarrolle sobre las tres representaciones vistas una rutina que elimine un elemento dado de la cola. Calcule la complejidad de la operación.

3.34. Desarrolle sobre las tres representaciones vistas una rutina que avance k posiciones en la cola un elemento dado. Calcule la complejidad de la operación.

3.35. Desarrolle sobre las tres representaciones vistas una rutina que calcule la posición de un elemento en la cola. Calcule la complejidad de la operación.

3.9. EL TAD Cola de Prioridad

Cuando un conjunto de procesos dentro de un computador hacen cola para utilizar algún recurso, no todos tienen la misma prioridad. Existen algunos más importantes que otros, que requieren ser atendidos con mayor prontitud. En ese caso, al entrar un nuevo elemento a la cola, debe saltarse todos aquellos cuya prioridad sea menor y se encuentren en la fila. Para sacar un elemento, lo mismo que para consultarlo, se toma el primero de la secuencia (el más antiguo de mayor prioridad). En el resto de operaciones, una cola de prioridades se comporta como una cola corriente.

En este tipo de estructuras, cada componente incluye un valor que representa su prioridad. Por esto, se extiende de la siguiente manera el formalismo propuesto en la sección anterior:

$$[\, x_1 \text{«} p_1] \;\; [\, x_2 \text{«} p_2] \;\ldots\; [\, x_n \text{«} p_n] \leftarrow$$

En éste, cada x_i representa un elemento y cada p_i su respectiva prioridad. De acuerdo con la definición de este tipo de colas, se debe cumplir necesariamente que $p_i \geq p_k$ cuando $i < k$. La cola de prioridades **vacía** se representa con el símbolo $\varnothing$

Ejemplo 3.9:

Para la cola de prioridades colP = [A«9] [B«5] [C«3] [D«1] ←, si llega el elemento E con prioridad 6, se obtiene la secuencia:

colP = [A«9] [E«6] [B«5] [C«3] [D«1] ←

Si llega el mismo elemento pero con prioridad 5, debe quedar antes que todos los elementos de menor prioridad, pero después de los que se encuentren presentes y tengan una prioridad mayor o igual:

colP = [A«9] [B«5] [E«5] [C«3] [D«1] ←

En cualquier caso, el siguiente elemento que va a ser atendido en la cola es el elemento A.
⏎

La especificación del TAD Cola de Prioridades es:

TAD ColaP[TipoCP]		
	$[x_1«p_1]\ [x_2«p_2] \ldots [x_n«p_n] \leftarrow$	
{ inv: $p_i \geq p_k$, $\forall$ i < k }		
Constructoras:		
• inicColaP:		→ ColaP
Modificadoras:		
• adicColaP:	ColaP x TipoCP x int	→ ColaP
• elimColaP:	ColaP	→ ColaP
Analizadoras:		
• infoColaP:	ColaP	→ TipoCP
• vaciaColaP:	ColaP	→ int
Destructora:		
• destruirColaP:	ColaP	

```
ColaP inicColaP( void )
/* Crea una cola de prioridades vacía */

{ post: inicColaP = ∅ }
```

```
void adicColaP( ColaP col, TipoCP elem, int pri )
/* Agrega el elemento elem con prioridad pri a la cola de prioridades */

{ post: col = [ x1«p1] [ x2«p2] ...[ xk«pk] [ elem«pri] ... [ xn«pn] ←, pk ≥ pri > pk+1 }
```

```
void elimColaP( ColaP col )
/* Elimina el primer elemento de la cola de prioridades */

{ post: col = [ x2«p2] ... [ xn«pn] ← }
```

```
TipoCP infoColaP( ColaP col )
/* Retorna el primer elemento de la cola de prioridades */

{ pre:  n > 0 }
{ post: infoColaP = x1 }
```

```
int vaciaColaP( ColaP col )
(* Informa si la cola de prioridades es vacía *)

{ post:  vaciaColaP = ( col = ∅ ) }
```

```
void destruirColaP( ColaP col )
/* Destruye la cola de prioridades retornando toda la memoria ocupada */

{ post: la cola de prioridades ha sido destruida }
```

3.10. Implementación del TAD Cola de Prioridades

Suponiendo que las prioridades que maneja la cola son valores enteros entre 0 y 9, se puede representar una cola de prioridades mediante un vector con 10 colas, cada una con los elementos de una misma prioridad. Por ejemplo, la cola de prioridades:

colP = [A«9] [B«4] [E«4] [C«2] [D«0] ←

estará representada internamente mediante el vector de 10 posiciones:

9	8	7	6	5	4	3	2	1	0
A ←	∅	∅	∅	∅	B E ←	∅	C ←	∅	D ←

La cola de prioridades vacía es un vector de diez colas sin elementos:

9	8	7	6	5	4	3	2	1	0
∅	∅	∅	∅	∅	∅	∅	∅	∅	∅

Con esta manera de representar una cola de prioridades se ilustra la forma de manejar agrupamientos de objetos abstractos para modelar un elemento de un TAD. Las estructuras de datos para esta implementación se declaran así:

```
typedef TipoCP TipoC;

typedef struct
{   int maxP;              /* Máxima prioridad presente en la cola */
    Cola info[ 10 ];       /* Vector de 10 colas */
} TColaP, *ColaP;
```

Las rutinas para implementar las operaciones del TAD son:

```
ColaP inicColaP( void )
{   int i;
    ColaP col = ( ColaP )malloc( sizeof( TColaP ) );
    for( i = 0; i < 10; i++ )
        col->info[ i ] = inicCola( );
    col->maxP = -1;
    return col;
}

void adicColaP( ColaP col, TipoCP elem, int pri )
{   adicCola( col->info[ pri ], elem );
    if( pri > col->maxP )
        col->maxP = pri;
}

void elimColaP( ColaP col )
{   elimCola( col->info[ col->maxP ] );
    for( ; col->maxP != -1 && vaciaCola( col->info[ col->maxP ] ); col->maxP-- );
}

TipoCP infoColaP( ColaP col )
{   return infoCola( col->info[ col->maxP ] );
}

int vaciaColaP( ColaP col )
{   return col->maxP == -1;
}

void destruirColaP( ColaP col )
{   int i;
    for( i = 0; i < 10; i++ )
        destruirCola( col->info[ i ] );
    free( col );
}
```

Ejercicios Propuestos

3.36. 💻 Suponiendo que la prioridad de los elementos viene dada por un valor entero no negativo, una cola de prioridades se puede implementar como una lista de colas, ordenada por prioridad. Por ejemplo, la cola de prioridades:

colP = [A«100] [B«100] [E«30] [C«30] [D«30] ←

Se representaría internamente con la lista de parejas [prioridad, cola]:

colP = < [100, A B ←], [30, E C D ←] >

Una cola de prioridades vacía estaría representada con una lista vacía. Defina claramente el esquema de representación e implemente las operaciones del TAD.

3.37. 💻 Suponiendo que la prioridad de los elementos viene dada por un valor entero no negativo, diseñe unas estructuras de datos basadas en apuntadores (multilistas) e implemente sobre ellas el TAD ColaP.

3.38. En las colas de prioridad, para evitar que algunos elementos de baja prioridad se queden sin atención es posible definir una política de servicio, en la cual, por cada elemento de una prioridad mayor que sale, se aumenta en 1 la prioridad de todos los elementos de menor importancia. Utilizando las estructuras de datos explicadas en la sección anterior, implemente la operación elimCola.

3.39. Una cola de prioridades se puede representar como una lista de parejas de la forma [prioridad, elemento]. Por ejemplo, la cola de prioridades:

colP = [A«100] [B«100] [E«30] [C«30] [D«30] ←

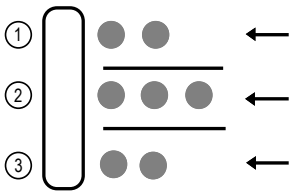
se puede representar con la lista de parejas:

colP = < [A, 100], [B, 100], [E, 30], [C, 30], [D, 30] >

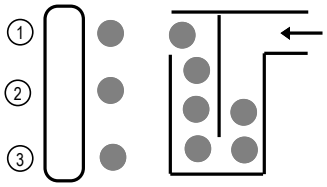
Defina claramente el esquema de representación e implemente las operaciones del TAD.

3.40. Diseñe unas estructuras de datos basadas en vectores e implemente sobre ellas el TAD ColaP.

3.41. En los bancos, el sistema de atención a los clientes se basa en una estructura líneal (una cola), atendida a la vez por N cajeros. Esta estructura se denomina una **Cola-MP** (multipunto), y remplaza las N tradicionales colas que se hacían al frente de cada punto de atención.



3 Colas para 3 cajeros



1 Cola-MP atendida po 3 cajeros

Al llegar un cliente a una Cola-MP, debe situarse detrás de todas las personas que ya se encuentran en fila, y esperar hasta que un cajero esté libre para que lo atienda. Cada cliente se identifica por un nombre y un número dado de transacciones, todas las cuales deben ser efectuadas por el cajero antes de atender a otro cliente. Todas las transacciones toman el mismo tiempo, y un cajero nunca descansa. Eso quiere decir que mientras un cajero atiende k clientes con una transacción, otro cajero puede atender un cliente con k transacciones. Los cajeros siempre están sincronizados, de manera que comienzan a atender una transacción de los clientes al mismo tiempo.

Son 6 las operaciones principales para administrar este tipo de estructuras: (1) crear una Cola-MP vacía con N cajeros, (2) agregar una persona de nombre nn y k transacciones a la Cola-MP, (3) atender una transacción por parte de todos los cajeros y actualizar el estado de la Cola-MP, (4) informar si la cola está vacía, (5) informar el nombre de la persona que está siendo atendida por el cajero i, (6) informar el número de cajeros del banco.

a-) Seleccione un formalismo para expresar el objeto abstracto

b-) Escriba el invariante del TAD

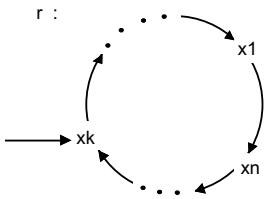
c-) Especifique las 6 operaciones del TAD

d-) Utilice el TAD para resolver el siguiente problema: Una persona de nombre nn, que se encuentra haciendo cola, quiere saber en cuanto tiempo va a salir del banco. Suponga que cada transacción toma 1 minuto. La operación modifica la cola durante el proceso.

e-) Haga un diseño de estructuras de datos, de manera que las 6 operaciones del TAD sean lo más eficientes posibles. Especifique claramente el esquema de representación. Implemente cada operación y calcule la complejidad.

3.11. El TAD Ronda

Una **ronda** es un objeto abstracto que se puede representar mediante el siguiente formalismo:



En ella, los elementos se encuentran organizados de manera circular y uno de ellos, denominado **punto de entrada**, es un elemento especial sobre el cual se aplican las operaciones. Una ronda vacía se representa con el símbolo $\rightarrow \varnothing$, y su punto de entrada es indefinido ($\perp$).

Para agregar un elemento a una ronda, existe la operación adicRonda cuyo efecto es el siguiente:

Existe también la operación elimineRonda, con el efecto inverso:

Para hacer girar una ronda, se tiene la operación rotarRonda, con el siguiente efecto:

Se cuenta además con dos analizadoras: una, para tomar el elemento que se encuentra en el punto de entrada (infoRonda), y otra, para establecer si una ronda está vacía (vaciaRonda). En los ejercicios que se proponen a continuación se trabaja sobre estas estructuras.

Ejercicios Propuestos

3.42. Haga una definición formal del TAD Ronda[TipoR]. Diseñe diferentes estructuras de datos y compárelas, teniendo en cuenta la complejidad de los algoritmos que implementan las operaciones. Finalmente, implemente el TAD Ronda sobre las mejores estructuras de datos encontradas.

3.43. El problema de Josefo.

Cuenta el historiador Flavio Josefo, que a principios de la era cristiana hubo un movimiento de rebelión judía contra la dominación de Roma. Los alzados fueron sitiados en una fortaleza inexpugnable. Estos hombres resistieron hasta que, faltos de comida, prefirieron matarse colectivamente antes que entregarse al enemigo. Para matarse, se dispusieron en círculo, y, a partir del jefe, contaban en redondo hasta N, y éste era asesinado por su vecino. Este procedimiento se repitió hasta que quedó un solo hombre. Este se debía suicidar, sin embargo prefirió entregarse a los romanos. De esa forma se conoció la historia.

a-) Desarrolle un algoritmo para resolver el problema de Josefo, en términos del TAD Ronda, el cual consiste en conocer el nombre del último soldado.

b-) Suponga que todas las operaciones del TAD Ronda son O(1). Calcule la complejidad de su algoritmo.

3.44. int longRonda(Ronda ron)
/* Calcula el número de elementos presentes en la ronda ron, suponiendo que no hay elementos repetidos */

3.45. void menorRonda(Ronda ron, TipoR elem)
/* Elimina de la ronda todos los elementos menores o iguales a elem. Supone que no hay elementos repetidos */

3.46. void invRonda(Ronda ron)
/* Invierte una ronda sin elementos repetidos, utilizando una pila como estructura auxiliar */

3.47. Ronda copiarRonda(Ronda ron)
/* Hace una copia de una ronda sin elementos repetidos */

3.48. Implemente el TAD Ronda sobre el TAD Lista. Calcule la complejidad de cada operación.

3.49. Implemente el TAD Ronda sobre una estructura circular sencillamente encadenada. Calcule la complejidad de cada operación.

3.50. Implemente el TAD Ronda sobre una estructura circular doblemente encadenada. Calcule la complejidad de cada operación.

3.12. El TAD Bicola

Una **bicola** es una estructura lineal en la cual los elementos sólo pueden ser adicionados, consultados y eliminados por cualquiera de sus dos extremos.

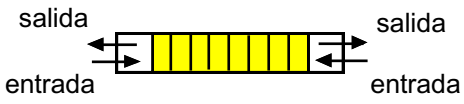


Fig. 3.6 - Operaciones sobre una bicola

El diseño, especificación e implementación de este nuevo TAD se propone como ejercicio.

Ejercicios Propuestos

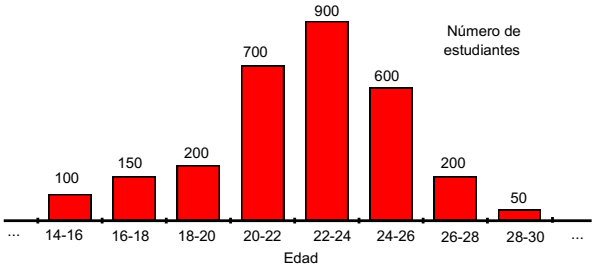
3.51. Diseñe y especifique formalmente el TAD Bicola. Piense en la posibilidad de tener una ventana sobre el extremo en el que se van a realizar las operaciones.

3.52. 💻 Implemente el TAD Bicola, diseñado en el punto anterior, sobre el TAD Lista.

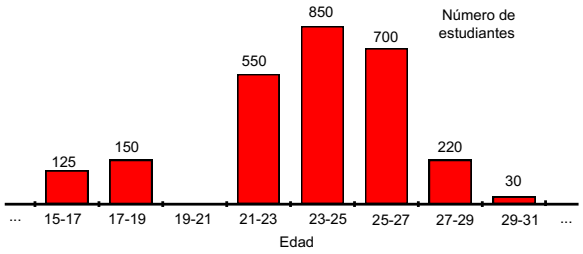
3.53. 💻 Implemente el TAD Bicola sobre una estructura doblemente encadenada.

3.54. 💻 Implemente el TAD Bicola, permitiendo la entrada de elementos por un solo lugar, pero con la posibilidad de eliminar por cualquier extremo.

3.55. 💻 🕒 Una **tabla de frecuencias** es una estructura lineal, que permite asociar con un rango de valores enteros positivos (menores que un valor dado) un número de ocurrencias. Por ejemplo, si se quiere mostrar la distribución de edades de los estudiantes de la Universidad, es posible utilizar una gráfica como la siguiente, que representa una tabla de frecuencias. Allí se tiene, por ejemplo, que son 100 los estudiantes matriculados que tienen 14 o más años, y menos de 16 ($14 \leq$ edad < 16).



Una característica de una tabla de frecuencias es su periodo, el cual indica el tamaño de cada rango. Para el ejemplo anterior, el periodo es 2. También es necesario fijar un valor de referencia, con respecto al cual se calculan los rangos. Para el ejemplo anterior puede ser cualquier número par menor o igual a 14. Un ejemplo de una tabla de frecuencias con el mismo periodo del anterior, pero con diferente valor de referencia (por ejemplo 1), es el siguiente:



a-) Diseñe y especifique el TAD TablaFrecuencia

b-) Diseñe y especifique un esquema de representación para el TAD TablaFrecuencia. Construya una tabla con la complejidad de las operaciones. Justifique su diseño utilizando argumentos como eficiencia, completitud, etc.

c-) Diseñe y especifique un esquema de persistencia para el TAD TablaFrecuencia

d-) Implemente y pruebe el TAD TablaFrecuencia

3.56. 💻 Una **bolsa** es una estructura lineal, que permite almacenar múltiples ocurrencias de elementos de un dominio específico.

a-) Seleccione un formalismo para representar los objetos abstractos.

b-) Diseñe y especifique el TAD Bolsa

c-) Diseñe y especifique un esquema de representación para el TAD Bolsa. Construya una tabla con la complejidad de las operaciones. Justifique su diseño utilizando argumentos como eficiencia, completitud, etc.

d-) Diseñe y especifique un esquema de persistencia para el TAD Bolsa

e-) Implemente y pruebe el TAD Bolsa

Bibliografía

- [AHO83] Aho, A., Hopcroft, J., Ullman, J., "Data Structures and Algorithms", Cap. 2, Addison-Wesley, 1983.
- [BER94] Bergin, J., "Data Abstraction: The Object-Oriented Approach Using C++", Cap. 7, McGraw-Hill, 1994.
- [DAL86] Dale, N., Lilly, S., "Pascal y Estructura de Datos", Cap. 3, McGraw-Hill, 1986.
- [DRO82] Dromey, R., "How to Solve it by Computer", Cap. 7, Prentice-Hall, 1982.
- [ESA89] Esakov, J., Weiss, T., "Data Structures: An Advanced Approack Using C", Cap. 5, Prentice-Hall, 1989.
- [FEL88] Feldman, M., "Data Structures with Modula-2", Cap. 5, Prentice-Hall, 1988.
- [HOR83] Horowitz, E., "Fundamentals of Data Structures", Cap. 3, Computer Science Press, 1983.
- [KNU73] Knuth, D., "The Art of Computer Programming. Vol. 1 - Fundamental Algorithms", 2da edición, Addison-Wesley, 1973.
- [KRU87] Kruse, R., "Data Structures & Program Design", Cap. 3, Prentice-Hall, 1987.
- [LIP87] Lipschutz, S., "Estructura de Datos", Cap. 6, McGraw-Hill, 1987.
- [MAR86] Martin, J., "Data Types and Data Structures", Cap. 6, Prentice-Hall, 1986.
- [STU85] Stubbs, D., Webre, W., "Data Structures with Abstract Data Types and Pascal", Brooks/Cole Publishing Company, 1985.
- [TEN93] Tenenbaum, A., Langsam, Y., "Estructuras de Datos en C", Cap. 4, Prentice Hall, 1993.
- [TRE76] Tremblay, J., Sorenson, P., "An Introduction to Data Structures with Applications", Cap. 3, McGraw-Hill, 1976.
- [WEL84] Welsh, J., Elder, J., "Sequential Program Structures", Prentice-Hall, 1984.

CAPITULO 4

ESTRUCTURAS RECURSIVAS:

ARBOLES BINARIOS

En este capítulo se presentan las estructuras de datos recursivas llamadas árboles binarios, utilizadas para representar relaciones de jerarquía entre elementos de un conjunto. Este tipo de organización se utiliza mucho para representar información ordenada, para mostrar relaciones estructurales entre elementos de un conjunto y, en general, para modelar situaciones que se puedan expresar en términos de jerarquías. Se estudian en particular los árboles ordenados, los árboles balanceados y los árboles de sintaxis.

4.1. Definiciones y Conceptos Básicos

Un **árbol binario** es una estructura recursiva, compuesta por un elemento, denominado la **raíz**, y por dos árboles binarios asociados, denominados **subárbol derecho** y **subárbol izquierdo**. El hecho de definir la estructura de datos en términos de sí misma es lo que hace que se denomine recursiva. El formalismo gráfico escogido para representar un árbol aparece en la figura 4.1. En él, se hace explícito que los dos subárboles tienen la misma composición estructural del árbol completo. El caso más sencillo de árbol binario es un árbol **vacío**, el cual no tiene elementos ni subárboles asociados. El símbolo escogido para representarlo es Δ.

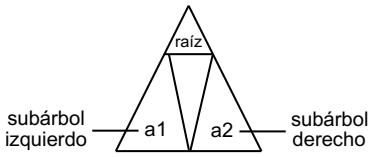


Fig. 4.1 - Formalismo para expresar un árbol binario

Otro formalismo posible para representar árboles binarios, cuando se quieren hacer explícitos todos los componentes de la estructura, utiliza un nombre para cada uno de los elementos del árbol y líneas para las relaciones de composición, como se muestra en la figura 4.2.

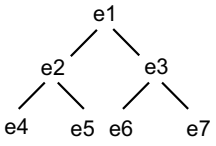
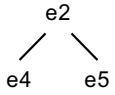
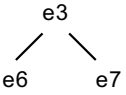


Fig. 4.2. - (a) Árbol binario



(b) Subárbol izquierdo



(c) Subárbol derecho

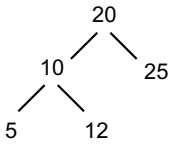
Un elemento e2 es **hijo** de un elemento e1, si e2 es la raíz de uno de los subárboles asociados con e1. En ese mismo caso, se dice que e1 es el **padre** de e2. Un elemento e2 es **hermano** de un elemento e3 si ambos tienen el mismo padre.

Un elemento de un árbol binario es una **hoja** si sus dos subárboles asociados son vacíos. En la figura 4.2, los elementos e4, e5, e6 y e7 son hojas. El formalismo gráfico para expresar que un árbol está compuesto solamente por una hoja aparece en la figura 4.3. Todo elemento de un árbol que no es una hoja se denomina un elemento **no terminal** o interior.

Fig. 4.3. Formalismo para un árbol compuesto por una hoja

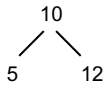
Ejemplo 4.1:

Para el árbol de la siguiente figura:



Se tiene que:

- La raíz es 20 y los dos subárboles asociados son:



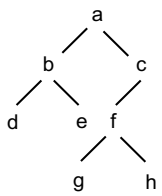
- Los elementos 5, 12 y 25 son hojas
- Los nodos interiores son 20 y 10.
- El padre de 5 es 10. El padre de 25 es 20. Los hijos de 10 son 5 y 12.
- Los elementos 5 y 12 son hermanos.

⏎

Un **camino** entre dos elementos e1 y e2 de un árbol binario es una secuencia <x_1, x_2,..., x_n >, que cumple que el primer elemento es e1, el último es e2, y cada elemento es padre de su sucesor. No siempre existe un camino entre dos elementos de un árbol, pero si existe, éste es único. La raíz de un árbol se caracteriza porque tiene un camino a cualquier elemento del árbol. La **longitud** de un camino <x_1, x_2,..., x_n > es n-1, o sea, el número de veces que se debe aplicar la relación padre→hijo durante el recorrido. Siempre existe un camino de longitud 0 que lleva de un elemento r a sí mismo y corresponde a la secuencia < r >. Por último, se tiene que un camino que parte de la raíz y termina en una hoja se conoce como una **rama**.

Ejemplo 4.2:

Para el árbol que se muestra en la siguiente figura:



Se cumple que:

- La longitud del camino < a, b, e > es 2. La longitud del camino < a > es 0.
- No existe un camino entre d y c.
- El único camino que lleva de c a h es < c, f, h >.
- El camino < a, c, f, g > es una rama.
- Desde la raíz existe un camino que lleva hasta cualquier otro elemento de la estructura.

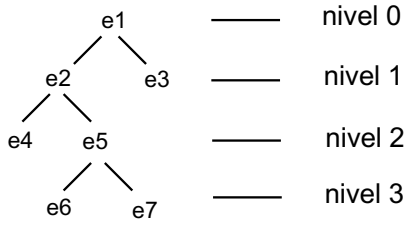
⏎

Un elemento e1 es **ancestro** de un elemento e2, si existe un camino entre e1 y e2. En ese mismo caso, se dice que e2 es **descendiente** de e1. El **nivel** de un elemento dentro de un árbol binario se define como la longitud del camino que parte de la raíz y llega hasta él. De esta forma, el nivel de la raíz es 0 y el de cualquier elemento es uno más que el de su padre. El nivel determina qué tan lejos de la raíz se encuentra un elemento. El **ancestro común** más próximo de dos elementos e1 y e2 es un elemento e3, que cumple que es ancestro de ambos y se encuentra a mayor nivel que cualquier otro ancestro que compartan.

La **altura** de un árbol es la longitud del camino más largo que parte de la raíz más 1. La altura de un árbol vacío se define como 0. El **peso** de un árbol es el número de elementos que contiene. Recursivamente se puede definir como la suma de los pesos de sus subárboles más 1. De acuerdo con la definición, el peso de un árbol vacío es 0.

Ejemplo 4.3:

Para el siguiente árbol:



Se tiene que:

- La altura es 4.
- El peso es 7.
- e1 es ancestro de todos los elementos del árbol.
- e7 es descendiente de e2.
- El ancestro común más próximo de e4 y e7 es e2.
- El ancestro común más próximo de e6 y e1 es e1.

⏎

Un árbol binario es **completo**, si todo elemento no terminal tiene asociados exactamente dos subárboles no vacíos. Eso equivale a decir que todo elemento de un árbol completo tiene los dos subárboles o no tiene ninguno. En la figura 4.4. aparece un ejemplo.

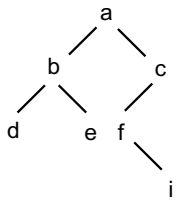
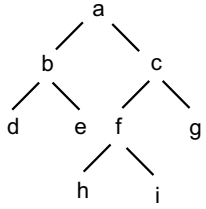
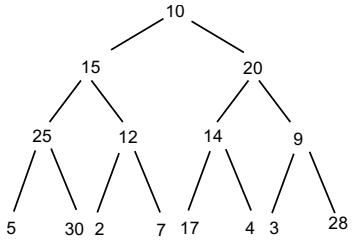


Fig. 4.4 -(a) árbol binario no completo.

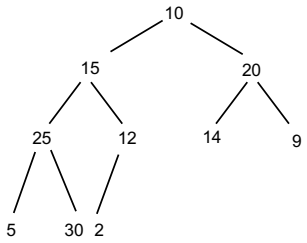


(b) árbol binario completo

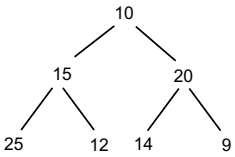
Un árbol binario está **lleno** si es completo y, además, todas las hojas están en el mismo nivel. Un árbol binario está **casi lleno** si está lleno hasta el penúltimo nivel y todas las hojas del siguiente nivel están tan a la izquierda como es posible. De acuerdo con la definición dada anteriormente, un árbol lleno es un caso particular de un árbol casi lleno. Estos dos conceptos se ilustran en la figura 4.5.



(a) Árbol binario lleno (y casi lleno)



(b) Árbol binario casi lleno



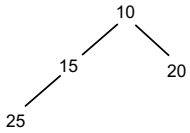


Fig. 4.5 - (c) Árbol binario lleno (y casi lleno) (d) Árbol binario casi lleno

Dos árboles binarios son **iguales** si ambos son vacíos, o si sus raíces son iguales, lo mismo que sus respectivos subárboles izquierdo y derecho. Dos árboles binarios son **isomorfos** si tienen la misma estructura, pero no necesariamente los mismos elementos. En la figura 4.6 aparece un ejemplo de dos árboles isomorfos.

Fig. 4.6 - Árboles binarios isomorfos

Dos árboles binarios son **semejantes** si contienen los mismos elementos, aunque no sean isomorfos. En ese caso, se dice que tienen el mismo contenido, pero no la misma estructura (figura 4.7).

Fig. 4.7 - Árboles binarios semejantes

Un árbol binario a1 **ocurre** en otro árbol binario a2, si a1 y a2 son iguales, o si a1 ocurre en alguno de los subárboles de a2. El caso general y un ejemplo de la noción de ocurrencia aparece en la figura 4.8.

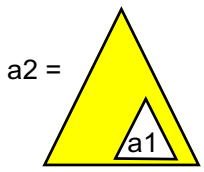
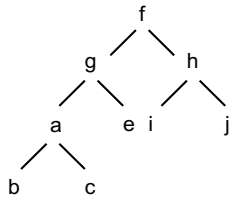
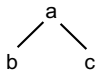


Fig. 4.8 - (a) Caso general



(b) árbol binario

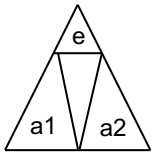


(c) árbol binario que ocurre en (b)

4.2. TAD Arbin: Analizadoras para Arboles Binarios

En esta sección se presenta un conjunto de operaciones analizadoras para el manejo de los árboles binarios. Esto permite mostrar la algorítmica que es común a este tipo de árboles. Las modificadoras y analizadoras particulares de cada clase específica de árbol (i.e. árboles ordenados, árboles de sintaxis, etc.) se presentan en la sección respectiva.

TAD Arbin[TipoA]



{ inv: a1 y a2 son disyuntos }

Analizadoras:

- izqArbin: Arbin → Arbin
- derArbin: Arbin → Arbin
- raizArbin: Arbin → TipoA
- vacioArbin: Arbin → int

```
Arbin izqArbin( Arbin a )
/* Retorna el subárbol izquierdo */

{ pre: a != Δ }

{ post: izqArbin = a1 }
```

```
Arbin derArbin( Arbin a )
/* Retorna el subárbol derecho */

{ pre: a != Δ }

{ post: derArbin = a2 }
```

```
TipoA raizArbin( Arbin a )
/* Retorna la raíz */

{ pre: a != Δ }
{ post: raízArbin = e }
```

```
int vacioArbin( Arbin a )
/*  Informa si un árbol binario es vacío */

{ post: vacioArbin = ( a = Δ ) }
```

4.3. Ejemplos de Utilización del TAD Arbin

En esta parte se muestran algunos ejemplos sencillos de utilización de las operaciones del TAD Arbin, como una manera de ilustrar la algorítmica básica de manejo de este tipo de estructuras. En una sección posterior se tratan otras técnicas más avanzadas de solución recursiva de problemas.

Ejemplo 4.4:

Calcular el peso de un árbol binario. La rutina tiene una salida de la recursión, que corresponde al caso trivial de un árbol sin elementos (peso = 0), y un avance, que calcula el peso de cada uno de los subárboles y le suma 1 para incluir en la cuenta el elemento de la raíz.

```
/* post: pesoArbin = número de elementos en el árbol binario a */

int pesoArbin( Arbin a )
{   if( vacioArbin( a ) )
        return 0;
    else
        return 1 + pesoArbin( izqArbin( a ) ) + pesoArbin( derArbin( a ) );
}
```

La complejidad de la rutina se puede calcular haciendo la expansión de la siguiente ecuación de recurrencia, donde n1 es el número de elementos del subárbol izquierdo, n2 el número de elementos del subárbol derecho y n el peso total del árbol:

$$T(n) = \begin{cases} 1, & n = 0 \\ 1 + T(n1) + T(n2), & n > 0 \end{cases}$$

Pero, si se tiene en cuenta que (n1 + n2) = n - 1, se puede reescribir la ecuación de la siguiente manera:

$$T(n) = \begin{cases} 1, & n = 0 \\ 1 + T(n-1), & n > 0 \end{cases}$$

De ahí se obtiene que la complejidad resultante para esta rutina es O(n), según se demostró en el capítulo 0.

Ejemplo 4.5:

Informar si un elemento se encuentra presente en un árbol binario. La rutina tiene dos salidas de la recursión y un avance:

- si el árbol es vacío debe retornar FALSE
- si el elemento es igual a la raíz, retorna TRUE
- en cualquier otro caso debe tratar de determinar si el elemento se encuentra en alguno de sus subárboles asociados.

```
/* post: estaArbin = ( elem ∈ a ) */
```

```
int estaArbin( Arbin a, TipoA elem )
{   if( vacioArbin( a ) )
        return FALSE;
    else if( elem == raizArbin( a ) )
        return TRUE;
    else
        return estaArbin( izqArbin( a ),elem ) || estaArbin( derArbin( a ),elem );
}
```

Siguiendo el mismo planteamiento del ejemplo anterior, se obtiene que la complejidad de la rutina es O(n), donde n es el peso del árbol.

⏎

Ejemplo 4.6:

Calcular el número de hojas de un árbol. La complejidad de esta rutina es O(n), donde n es el peso del árbol. La función tiene dos salidas (el árbol es vacío o el árbol es una hoja) y un avance (sumar las hojas de los dos subárboles asociados).

```
/* post: numHojas = número de hojas del árbol a */

int numHojas( Arbin a )
{   if( vacioArbin( a ) )
        return 0;
    else if( vacioArbin( izqArbin( a ) ) && vacioArbin( derArbin( a ) ) )
        return 1;
    else
        return numHojas( izqArbin( a ) ) + numHojas( derArbin( a ) );
}
```

⏎

Ejemplo 4.7:

Calcular el número de veces que aparece un elemento en un árbol binario. La complejidad de esta rutina es O(n), donde n es el peso del árbol. Tiene una salida de la recursión (el árbol es vacío) y dos avances, según si el elemento de la raíz es igual al valor buscado o diferente.

```
/* post: ocurre = número de apariciones del elemento elem en el árbol a */

int ocurre( Arbin a, TipoA elem )
{   if( vacioArbin( a ) )
        return 0;
    else if( raizArbin( a ) == elem )
        return 1 + ocurre( izqArbin( a ), elem ) + ocurre( derArbin( a ), elem );
    else
        return ocurre( izqArbin( a ), elem ) + ocurre( derArbin( a ), elem );
}
```

⏎

Ejemplo 4.8:

Decidir si existe un camino entre dos elementos e1 y e2 de un árbol binario. La complejidad de la rutina de este ejemplo es O(n), donde n es el peso del árbol. Tiene dos salidas de la recursión -una O(1) y la otra O(n)- y un avance que es O(n) (la misma ecuación de recurrencia de los ejemplos anteriores).

```
/* camino = existe un camino en el árbol a que parte de e1 y termina en e2 */

int camino( Arbin a, TipoA e1, TipoA e2 )
{   if( vacioArbin( a ) )
        return FALSE;
    else if( e1 == raizArbin( a ) )
        return estaArbin( a, e2 );
    else
        return camino( izqArbin( a ), e1, e2 ) || camino( derArbin( a ), e1, e2 );
}
```

Ejemplo 4.9:

Calcular el número de elementos que tiene un árbol binario en un nivel dado. La complejidad de esta rutina es O(n), donde n es el peso del árbol, puesto que en el peor de los casos el nivel pedido corresponde a la altura del árbol. El avance consiste en sumar los elementos del nivel num - 1 de cada uno de los subárboles asociados, ya que esos son los elementos del nivel num del árbol completo.

```
/* pre: num ≥ 0 */
/* post: contNivel = número de elementos en el nivel num del árbol a */

int contNivel( Arbin a, int num )
{   if( vacioArbin( a ) )
        return 0;
    else if( num == 0 )
        return 1;
    else
        return contNivel( izqArbin( a ), num - 1 ) + contNivel( derArbin( a ), num - 1 );
}
```

Ejemplo 4.10:

Informar si dos árboles binarios sin elementos repetidos son semejantes. Este problema implica dos rutinas: la primera función verifica que los dos árboles tengan el mismo peso, y que todos los elementos del primer árbol estén en el segundo.

```
/* pre: a1 y a2 no tienen elementos repetidos */
/* post: semejantes = a1 y a2 son semejantes */

int semejantes( Arbin a1, Arbin a2 )
{   return pesoArbin( a1 ) == pesoArbin( a2 ) && incluidoArbin( a1, a2 );
}
```

La segunda función verifica, para cada elemento de a1, si éste se encuentra en a2.

```
/* incluidoArbin = todo elemento de a1 está en a2 */

int incluidoArbin( Arbin a1, Arbin a2 )
{   if( vacioArbin( a1 ) )
        return TRUE;
    else
        return estaArbin( a2, raizArbin( a1 ) ) &&
            incluidoArbin( izqArbin( a1 ), a2 ) &&
            incluidoArbin( derArbin( a1 ), a2 );
}
```

La complejidad de la función incluidoArbin es O(n * m), donde n es el peso de a1 y m es el peso de a2. Esto se obtiene de la expansión de la siguiente ecuación de recurrencia:

$$T_{incluidoArbin}(n,m) = \begin{cases} 1, & n = 0 \\ 1 + T_{estaArbin}(m) + T_{incluidoArbin}(n-1), & n > 0 \end{cases}$$

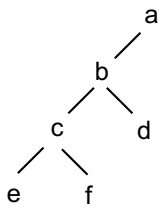
Esto hace que la complejidad de la rutina semejantes sea O(n * m), puesto que:

$O_{semejantes} = \max(O_{pesoArbin}(n), O_{pesoArbin}(m), O_{incluidoArbin}(n * m)) = O (n * m)$

⏎

Ejercicios Propuestos:

4.1. Para el siguiente árbol binario:



a-) ¿Cuáles son hojas?
b-) ¿Cuál es la altura?
c-) ¿Cuántas y cuáles son sus ramas?
d-) ¿Cuál es el ancestro común más próximo entre f y d?
e-) ¿Cuántos y cuáles son los hermanos de c?
f-) ¿Cuántos y cuáles subárboles tiene asociados el elemento a?
g-) ¿Cuál es su peso?

4.2. Dibuje dos árboles isomorfos y semejantes, que no sean iguales

4.3. Determine los siguientes valores para un árbol binario:
a-) Número mínimo y máximo de elementos en un árbol completo de N niveles.
b-) Número mínimo de niveles en un árbol de peso P.
c-) Número máximo de hojas en un árbol con N niveles.
d-) Número mínimo y máximo de elementos presentes en el nivel N de un árbol completo de altura H.
e-) Número de elementos en un árbol lleno de N niveles.
f-) Número mínimo de elementos de un árbol casi lleno de N niveles.

Especifique y desarrolle las siguientes rutinas utilizando las operaciones del TAD Arbin. Calcule la complejidad de cada una de ellas, resolviendo la ecuación de recurrencia respectiva.

4.4. int alturaArbin(Arbin a)
/* Calcula la altura de un árbol binario */

4.5. int igualesArbin(Arbin a1, Arbin a2)
/* Indica si dos árboles binarios son iguales */

4.6. int isomorfosArbin(Arbin a1, Arbin a2)
/* Informa si los árboles binarios a1 y a2 son isomorfos */

4.7. int semejantesArbin(Arbin a1, Arbin a2)
/* Informa si los árboles binarios a1 y a2 son semejantes, aunque existan elementos repetidos. Esto es, si un elemento aparece k veces en uno de los árboles, debe aparecer el mismo número de veces en el segundo */

4.8. int completoArbin(Arbin a)
/* Indica si un árbol binario es completo */

4.9. int llenoArbin(Arbin a)
/* Informa si un árbol binario está lleno */

4.10. int casiLlenoArbin(Arbin a)
/* Indica si un árbol binario está casi lleno */

4.11. int estableArbin(Arbin a)
/* Un árbol de valores enteros es **estable** si para todo elemento de la estructura su padre es mayor. Esta función indica si un árbol es estable */

4.12. Lista primosArbin(Arbin a, TipoA elem)
/* Retorna una lista con los primos del elemento elem dentro del árbol. Por **primos** se entienden los hijos del hermano del padre */

4.13. Lista busqueCaminoArbin(Arbin a, TipoA elem)
/* Retorna la lista de elementos del árbol correspondiente al camino que lleva desde la raíz del árbol hasta el elemento elem. Si el camino no existe retorna una lista vacía */

4.14. int ocurreArbin(Arbin a1, Arbin a2)
/* Indica si el árbol a2 ocurre en el árbol a1 */

4.15. TipoA ancestroLista(Arbin a, Lista lst)
/* Retorna el ancestro común más próximo de los elementos presentes en la lista lst, los cuales están en el árbol a */

4.16. int esMenorArbin(Arbin a1, Arbin a2)
/* Indica si el árbol binario a1 es menor que el árbol binario a2. Un árbol a1 es **menor** que otro a2, si todos los elementos de a1 son menores que todos los elementos de a2 */

4.17. Lista rutaMínimaArbin(Arbin a, TipoA e1, TipoA e2)
/* Se define la **ruta mínima** entre dos elementos cualesquiera e1 y e2 de un árbol binario sin elementos repetidos, como la secuencia de elementos $< x_1, x_2, ..., x_n >$ que cumple las siguientes condiciones:

- $x_1 = e1$, $x_n = e2$
- x_i es padre de x_{i+1}, o x_{i+1} es padre de x_i
- no existen elementos repetidos en la secuencia

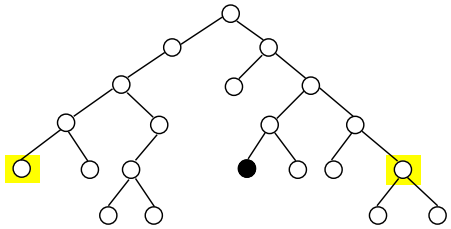
Dicha ruta existe entre todo par de elementos de un árbol binario, y es única. Esta función retorna una lista de elementos con la ruta mínima entre dos elementos dados */

4.18. 🕒 void impRutasMinimas(Arbin a, TipoA elem)

/* Imprime las rutas mínimas (ver ejercicio anterior) desde el elemento elem hasta todos los demás componentes del mismo nivel */

4.19. 🕒🕒 Lista mVecinos(Arbin a, TipoA elem, int m)

/* Retorna los vecinos del elemento elem que se encuentran a una distancia m de él. Por **vecino** se entiende un elemento del mismo nivel y por **distancia** el número de elementos que los separa. Por ejemplo, en la siguiente figura aparecen los 3-vecinos del elemento marcado:



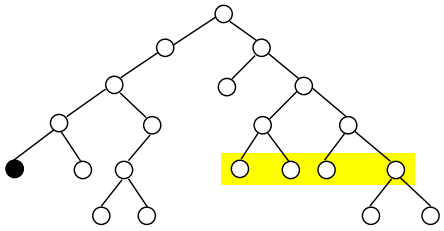
Suponga que no hay elementos repetidos en el árbol */

4.20. Lista listaNivelArbin(Arbin a, int n)

/* Retorna la lista con todos los elementos del nivel n del árbol binario */

4.21. 🕒🕒 Lista primosNLejanos(Arbin a, int n, TipoA elem)

/* Se definen los **primos n-lejanos** de un elemento de un árbol binario como aquellos elementos del mismo nivel cuyo ancestro común más próximo se encuentra exactamente n niveles por encima de ellos. Con esta definición, se puede afirmar que un hermano es un primo 1-lejano. Otro ejemplo es el del dibujo, que muestra los primos 4-lejanos del elemento marcado:



Desarrolle una función que dado un árbol binario sin elementos repetidos, un elemento presente en el árbol y un valor n, retorne una lista con todos los primos n-lejanos del elemento */

4.22. int esMovilArbin(Arbin a)

/* Se define el **contenido** de un árbol binario como la suma de los valores de todos sus elementos (suponiendo que son enteros). Se dice que un árbol binario es un **móvil** si la diferencia de contenido de los dos subárboles no difiere en más de uno, y éstos a su vez son móviles. En especial, un árbol vacío es móvil y tiene contenido 0. Esta función indica si un árbol binario es un móvil */

4.4. Recorrido de Arboles Binarios

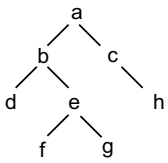
La operación de **recorrer** un árbol binario se puede hacer de diversas maneras. El orden en el que se visitan los elementos puede variar, puesto que lo único que debe garantizar la operación es que pase exactamente una vez por cada uno de los elementos del árbol. Las cuatro maneras más comunes de hacer este recorrido se denominan: preorden, inorden, postorden y niveles, y se definen de la siguiente manera:

- **preorden**: visite la raíz, recorra en preorden el subárbol izquierdo y, finalmente, recorra en preorden el subárbol derecho.

- **inorden**: recorra en inorden el subárbol izquierdo, luego visite el elemento de la raíz y, por último, recorra en inorden el subárbol derecho.
- **postorden**: recorra el subárbol izquierdo y el subárbol derecho en postorden y, luego, visite el elemento de la raíz.
- **niveles**: visite la raíz del árbol, los elementos de nivel 1 de izquierda a derecha, seguidos por los elementos de nivel 2, y así sucesivamente hasta terminar de visitar todos los elementos.

Ejemplo 4.11:

Para el árbol que se muestra en la siguiente figura:



Los recorridos principales son:

- preorden: a - b - d - e - f - g - c - h
- inorden: d - b - f - e - g - a - c - h
- postorden: d - f - g - e - b - h - c - a
- niveles: a - b - c - d - e - h - f - g

⏎

Los algoritmos recursivos para hacer los 3 primeros recorridos surgen de manera natural de su definición, tal como se muestra a continuación. La rutina visitar es la encargada de hacer la operación respectiva (i.e. imprimir) sobre cada elemento:

```
void preordenArbin( Arbin a )
{   if( !vacioArbin( a ) )
    {     visitar( raizArbin( a ) );
          preordenArbin( izqArbin( a ) );
          preordenArbin( derArbin( a ) );
    }
}

void inordenArbin( Arbin a )
{   if( !vacioArbin( a ) )
    {     inordenArbin( izqArbin( a ) );
          visitar( raizArbin( a ) );
          inordenArbin( derArbin( a ) );
    }
}

void postordenArbin( Arbin a )
{   if( !vacioArbin( a ) )
    {     postordenArbin( izqArbin( a ) );
          postordenArbin( derArbin( a ) );
          visitar( raizArbin( a ) );
    }
}
```

Cada uno de estos algoritmos es O(n), donde n es el peso del árbol, puesto que tienen que pasar una vez sobre cada elemento del árbol.

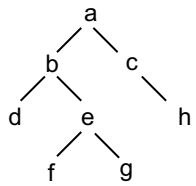
4.4.1. Algoritmo de Recorrido por Niveles

Dado que este recorrido no tiene un planteamiento recursivo, la rutina más sencilla que lo implementa es iterativa. Para esto, es necesario contar con alguna estructura auxiliar de datos que permita mantener la información relacionada con cada uno de los niveles, a medida que se avanza. En este caso, la estructura apropiada es una cola, que se maneja de la siguiente manera:

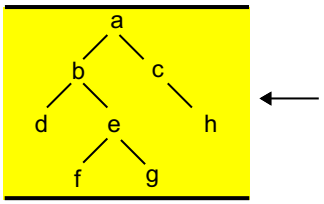
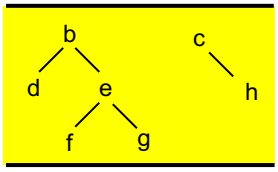
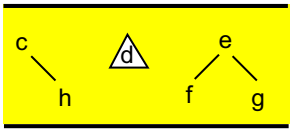
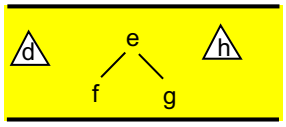
En un momento dado (inicialmente, por ejemplo), se encuentran en la cola todos los subárboles cuya raíz está en el nivel k (denominados los k-árboles). De esta cola se sacan uno por uno, se visita su raíz y se incluyen al final de la misma cola sus subárboles asociados. Al final de esto, cuando hayan sido sacados todos los k-árboles, en la cola estarán ordenados de izquierda a derecha todos los k+1-árboles, y se habrán visitado los elementos del nivel k. El proceso termina cuando la cola queda vacía.

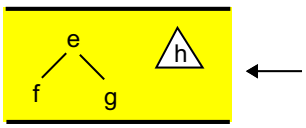
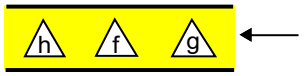
Ejemplo 4.12:

Para el árbol:



El proceso que sigue el algoritmo de recorrido por niveles se muestra en la siguiente tabla:

Cola	Acción
	El árbol completo está en la primera posición de la cola. Se saca el árbol de la cola, se visita el elemento "a", y se agregan a la cola los dos subárboles
	Se saca el primer árbol de la cola, se visita el elemento "b", y se agregan a la cola sus dos subárboles
	Se saca el primer árbol de la cola, se visita el elemento "c" y se agregan sus subárboles
	Se saca el primer árbol de la cola, se visita el elemento "d".

	Se saca el primer árbol de la cola, se visita el elemento "e", y se agregan sus subárboles
	Se visitan los elementos "h", "f" y "g", y termina el algoritmo al quedar la cola vacía

El algoritmo para realizar este recorrido es de complejidad O(n) (n es el peso del árbol), si todas las operaciones del TAD Cola son O(1).

```
void nivelesArbin( Arbin a )
{   Cola col;
    Arbin arb;
    if( !vacioArbin( a ) )
    {     col = inicCola(  );
          adicCola( col, a );
          while( !vaciaCola( col ) )
          {     arb = infoCola( col );
                elimCola( col );
                if( !vacioArbin( arb ) )
                {     visitar( raizArbin( arb ) );
                      adicCola( col, izqArbin( arb ) );
                      adicCola( col, derArbin( arb ) );
                }
          }
    }
}
```

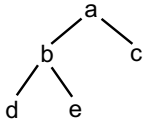
4.4.2. Algoritmo Iterativo de Recorrido de Árboles

En este tipo de estructuras recursivas, aunque la solución iterativa suele ser más eficiente, el algoritmo que la implementa es típicamente más complicado, y requiere estructuras de datos adicionales. En este caso, se necesita una pila, que le permita a la rutina volver a subir en la estructura, después de visitar los niveles más alejados de la raíz.

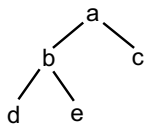
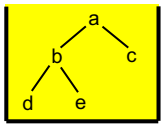
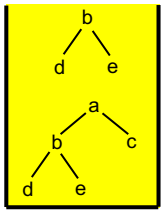
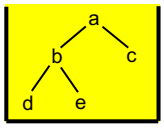
En esta pila, se guardan los árboles binarios por los cuales se ha bajado tratando de recorrer su subárbol izquierdo. Una vez se ha llegado a una hoja, se recupera y elimina el último árbol incluido en la pila, se visita su raíz y se comienza a bajar por el subárbol derecho.

Ejemplo 4.13:

Para el árbol:



El proceso que sigue la rutina se resume como:

Árbol de trabajo	Pila	Acción
		Va a comenzar el proceso. La pila se encuentra vacía, y el árbol de trabajo es el árbol inicial. Baja por el subárbol izquierdo, y guarda en la pila el árbol completo
		Baja por el subárbol izquierdo, y guarda el árbol de trabajo en la pila
		Visita el elemento "d", y, para subir, saca el primer árbol de la pila, visita su raíz ("b"), y coloca como árbol de trabajo el subárbol derecho.
		Visita el elemento "e", y, para subir, saca el primer árbol de la pila, visita su raíz ("a"), y coloca como árbol de trabajo el subárbol derecho. La pila queda vacía.
		Visita el elemento "c", y, al encontrar la pila vacía, termina el proceso

La rutina que hace el recorrido iterativo de un árbol binario se presenta a continuación. Si se supone que todas las operaciones del TAD Pila son O(1), se puede afirmar que la complejidad de esta rutina es O(n), donde n es el peso del árbol.

```
void inordenItera( Arbin a )
{   Pila pil;
    Arbin arb;
    if( !vacioArbin( a ) )
    {     arb = izqArbin( a );
          pil = inicPila(  );
          adicPila( pil, a );
          while( !vacioArbin( arb ) || !vaciaPila( pil ) )
                if( !vacioArbin( arb ) )
                {     adicPila( pil, arb );
                      arb = izqArbin( arb );
                }
                else
                {     arb = infoPila( pil );
                      elimPila( pil );
                      visitar( raizArbin( arb ) );
                      arb = derArbin( arb );
                }
    }
}
```

4.4.3. Reconstrucción de un Árbol a partir de sus Recorridos

Si un árbol binario no tiene elementos repetidos, es posible reconstruirlo a partir de la información que se obtiene de dos de sus recorridos (inorden y postorden o inorden y preorden). El proceso de reconstrucción del árbol binario, a partir de la secuencia de los elementos que va visitando en sus recorridos, se ilustra en el siguiente ejemplo. La implementación de la rutina que lo hace se muestra en una sección posterior, directamente sobre las estructuras de datos que representan el árbol.

Ejemplo 4.14:

Suponga que se quiere reconstruir el árbol binario sin elementos repetidos, cuyos recorridos en inorden y preorden vienen dados por las secuencias:

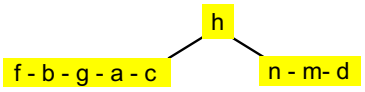
preorden: h - a - b - f - g - c - m - n - d

inorden: f - b - g - a - c - h - n - m - d

Paso 1: Encontrar la raíz y subdividir los recorridos. La raíz siempre es el primer elemento del preorden. Al localizar dicho elemento en el inorden, se obtienen los recorridos para los dos subárboles que se le deben asociar como izquierdo y derecho.

preorden : h a - b - f - g - c - m - n - d

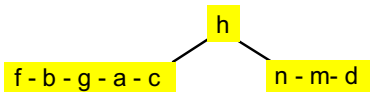
inorden :



Paso 2: Sabiendo el peso de cada uno de los subárboles (5 el izquierdo y 3 el derecho), es posible calcular el recorrido en preorden de cada uno de ellos.

preorden : h a - b - f - g - c m - n - d

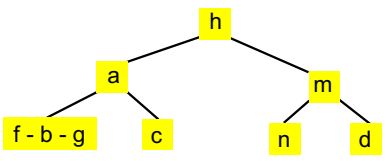
inorden:



Paso 3: Repetir el paso 1 con cada uno de los subárboles encontrados.

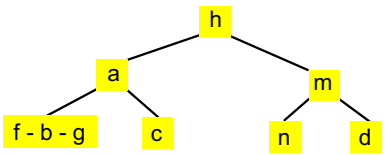
preorden : h a b - f - g - c m n - d

inorden:



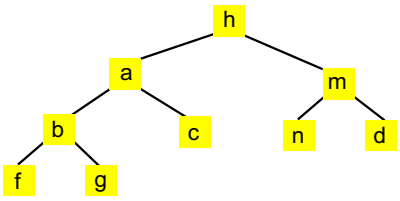
Paso 4: Repetir el paso 2 con cada uno de los subárboles encontrados.

preorden : h a b f - g c m n d



inorden:

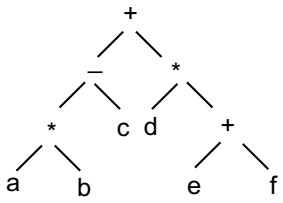
Paso 5: Repetir el mismo proceso descrito en los pasos 1 y 2, con el único subárbol que falta.



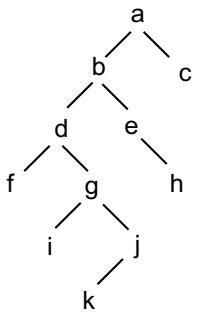
inorden:

Ejercicios Propuestos:

4.23. Para el siguiente árbol binario, dé sus 4 recorridos principales:



4.24. Para el siguiente árbol binario, dé sus 4 recorridos principales:



4.25. Reconstruya el árbol binario con los siguientes recorridos:

preorden: 10 - 20 - 30 - 50 - 60 - 40 - 70 - 80 - 90
inorden: 50 - 30 - 60 - 20 - 80 - 70 - 90 - 40 - 10

4.26. Reconstruya el árbol binario con los siguientes recorridos:

postorden: 60 - 30 - 80 - 70 - 40 - 20 - 50 - 90 -10
inorden: 30 - 60 - 20 - 80 - 70 - 40 -10 - 90 -50

Utilizando las operaciones del TAD Arbin, implemente las siguientes operaciones y calcule su complejidad:

4.27. void postordenArbin(Arbin a)
/* Hace un recorrido iterativo en postorden de un árbol binario */

4.28. void escribeRamas(Arbin a)
/* Imprime por pantalla todas las ramas del árbol a */

4.29. int ramaMinima(Arbin a)
/* Suponiendo que los elementos del árbol a son enteros, se define el **costo de un camino** como la suma de los componentes de dicha secuencia. Esta función retorna el costo de la rama más barata del árbol */

4.30. void niveles2Arbin(Arbin a)
/* Hace un recorrido por niveles de un árbol binario, pero en orden inverso. Esto es, recorre los elementos de abajo hacia arriba, y de izquierda a derecha. Comienza en la hoja que se encuentra más a la izquierda del árbol y termina en la raíz */

4.31. void gRecorrido(Arbin a)
/* Se define la **anchura** de un nivel de un árbol binario como el número de elementos presentes en dicho nivel. La **gordura** de un árbol binario corresponde al valor máximo de las anchuras de sus niveles. Un g+Recorrido (o recorrido por subárboles más gordos), de un árbol binario a, se define recursivamente de la siguiente manera:
- visitar la raíz de a,
- recorrer en g+ el subárbol más gordo de a,
- recorrer en g+ el otro subárbol de a.

Esta rutina realiza un g+ recorrido del árbol binario a */

4.32. void impArbin(Arbin a)
/* Imprime por pantalla un árbol binario, permitiendo apreciar su estructura */

4.5. Algorítmica de Manejo de Árboles

En esta sección se ilustra, a través de ejemplos, la gran gama de posibilidades que tiene un programador para resolver un problema que incluya árboles. Se ven desde soluciones iterativas, eficientes pero complicadas, hasta técnicas avanzadas de acumulación de parámetros para planteamientos recursivos. En cada ejemplo aparecen 3 ó 4 soluciones del mismo problema, con sus evidentes ventajas y desventajas, para que el lector se dé una idea de las opciones con que cuenta.

Ejemplo 4.15:

Calcular el nivel en el que aparece un elemento dado en un árbol binario sin elementos repetidos. Si el elemento no está presente, la rutina retorna el valor -1.

Solución No. 1: Planteamiento recursivo en el cual el algoritmo verifica la existencia del elemento en uno de los subárboles antes de hacer el avance de la recursión ("descenso controlado"). Es casi siempre la solución más sencilla, pero la menos eficiente (típicamente es O(n^2), puesto que establecer en cuál de los dos subárboles se encuentra un elemento es O(n)).

```
/* pre: a no tiene elementos repetidos */
/* post: nivelArbin1 = nivel en el que aparece el elemento elem en el árbol a, o -1 si no aparece */
```

```
int nivelArbin1( Arbin a, TipoA elem )
{   if( vacioArbin( a ) )
        return -1;
    else if( raizArbin( a ) == elem )
        return 0;
    else if( estaArbin( izqArbin( a ), elem ) )
        return nivelArbin1( izqArbin( a ), elem ) + 1;
    else if( estaArbin( derArbin( a ), elem ) )
        return nivelArbin1( derArbin( a ), elem ) + 1;
    else
        return -1;
}
```

Solución No. 2: Planteamiento recursivo en el cual el algoritmo hace el avance de la recursión sobre el subárbol izquierdo, y, sólo si la rutina informa que no tuvo éxito, intenta avanzar sobre el otro subárbol ("subir información en la recursión"). En este caso se utiliza el valor especial -1 para indicar que no encontró el elemento y no pudo calcular el nivel en el que se encuentra. La complejidad de la rutina es O(n), puesto que, en el peor de los casos, hace una pasada sobre cada elemento de la estructura.

```
/* pre: a no tiene elementos repetidos */
/* post: nivelArbin2 = nivel en el que aparece el elemento elem en el árbol a, o -1 si no aparece */

int nivelArbin2( Arbin a, TipoA elem )
{   int temp;
    if( vacioArbin( a ) )
        return -1;
    else if( raizArbin( a ) == elem )
        return 0;
    else if( ( temp = nivelArbin2( izqArbin( a ), elem ) ) != -1 )
        return temp + 1;
    else if( ( temp = nivelArbin2( derArbin( a ), elem ) ) != -1 )
        return temp + 1;
    else
        return -1;
}
```

Solución No. 3: Planteamiento recursivo en el cual el algoritmo va acumulando información en un parámetro adicional, y la va enviando hacia los niveles inferiores de la recursión, los cuales se encargan de dar la respuesta ("acumulación de parámetros"). Esta técnica se implementa a través de dos rutinas: la primera se encarga de crear el espacio del parámetro adicional, y de darle un valor inicial. La segunda va actualizando el valor de dicho parámetro a medida que avanza la recursión. La complejidad de este planteamiento puede ser lineal o cuadrática, según se haga un descenso controlado o se suba información en la recursión. A continuación se presentan las dos opciones:

```
/* pre: a no tiene elementos repetidos */
/* post: nivelArbin3 = nivel en el que aparece el elemento elem en el árbol a, o -1 si no aparece */

int nivelArbin3( Arbin a, TipoA elem )
{   return ( !estaArbin( a, elem ) ) ? -1 : nivelAux( a, elem, 0 );
}
```

Solución de la rutina nivelAux utilizando descenso controlado (O (n^2)):

```
/* pre: a no tiene elementos repetidos, elem está en a, niv = nivel de la raíz de a en el árbol completo */
/* post: nivelAux = nivel en el que aparece el elemento elem en el árbol completo */

int nivelAux( Arbin a, TipoA elem, int niv )
{   if( raizArbin( a ) == elem )
        return niv;
    else if( estaArbin( izqArbin( a ), elem ) )
        return nivelAux( izqArbin( a ), elem, niv + 1 );
    else
        return nivelAux( derArbin( a ), elem, niv + 1 );
}
```

Solución de la rutina nivelAux subiendo información en la recursión. Aquí no se puede exigir en la precondición que el elemento esté en el árbol, puesto que la función va a intentar bajar por cada subárbol sin antes verificar que allí se encuentre el elemento (O(n)):

```
/* pre: a no tiene elementos repetidos, niv = nivel de la raíz de a en el árbol completo */
/* post: nivelAux = nivel en el que aparece el elemento elem en el árbol completo, o -1 si no aparece */

int nivelAux( Arbin a, TipoA elem, int niv )
{   int temp;
    if( vacioArbin( a ) )
        return -1;
    else if( raizArbin( a ) == elem )
        return niv;
    else if( ( temp = nivelAux( izqArbin( a ), elem, niv + 1 ) ) != -1 )
        return temp;
    else if( ( temp = nivelAux( derArbin( a ), elem, niv + 1 ) ) != -1 )
        return temp;
    else
        return -1;
}
```

Solución No. 4: Planteamiento iterativo de complejidad O(n^2). El algoritmo necesita una pila como estructura auxiliar de datos para poder establecer si un elemento está presente en un árbol. Con esta rutina, se va controlando el descenso por la estructura, hasta encontrar el elemento buscado. Existe una solución iterativa de complejidad O(n), que no se presenta en este ejemplo.

```
/* pre: a no tiene elementos repetidos */
/* post: nivelArbin4 = nivel en el que aparece el elemento elem en el árbol a, o -1 si no aparece */

int nivelArbin4( Arbin a, TipoA elem )
{   int niv;
    if( !estaItera( a, elem ) )
        return -1;
    else
    {   for( niv = 0; !vacioArbin( a ) && raizArbin( a ) != elem; niv++ )
            a = ( estaItera( izqArbin( a ), elem ) ) ? izqArbin( a ) : derArbin( a );
        return ( !vacioArbin( a ) ) ? niv : -1;
    }
}
```

La rutina iterativa que establece si un elemento se encuentra en un árbol binario, hace un recorrido en preorden del árbol utilizando una pila, y tan pronto localiza el elemento destruye la pila y retorna la respuesta.

```
/* post: estaItera = elem está en a */

int estaItera( Arbin a, TipoA elem )
{   Pila pil = inicPila( );
    while( !vacioArbin( a ) || !vaciaPila( pil ) )
    {     if( !vacioArbin( a ) )
          {     if( raizArbin( a ) == elem )
                {     destruirPila( pil );
                      return TRUE;
                }
                adicPila( pil, derArbin( a ) );
                a = izqArbin( a );
          }
          else
          {     a = infoPila( pil );
                elimPila( pil );
          }
    }
    destruirPila( pil );
    return FALSE;
}
```

Ejemplo 4.16:

Encontrar el padre de un elemento en un árbol sin elementos repetidos, suponiendo que este elemento se encuentra presente en la estructura de datos y es distinto de la raíz.

Solución No. 1: Planteamiento recursivo con descenso controlado. Complejidad O(n^2).

```
/* pre: a no tiene elementos repetidos, elem está en a, elem es diferente de la raíz de a */
/* post: padreArbin1 = padre de elem en a */

TipoA padreArbin1( Arbin a, TipoA elem )
{   if( !vacioArbin( izqArbin( a ) ) && raizArbin( izqArbin( a ) ) == elem ||
        !vacioArbin( derArbin( a ) ) && raizArbin( derArbin( a ) ) == elem )
          return raizArbin( a );
    else if( estaArbin( izqArbin( a ), elem ) )
          return padreArbin1( izqArbin( a ), elem );
    else
          return padreArbin1( derArbin( a ), elem );
}
```

Solución No. 2: Planteamiento recursivo subiendo información durante el avance. Puesto que el elemento que se quiere encontrar es de TipoA, y no es posible definir para cualquier tipo de dato un valor distinguido que indique la falla del proceso, se utiliza un parámetro por referencia para subir la información del padre, y funcionalmente se informa si se encontró la solución. Para definir el espacio de este nuevo parámetro se utiliza una función auxiliar. La complejidad es O(n).

```
/* pre: a no tiene elementos repetidos, elem está en a, elem es diferente de la raíz de a */
/* post: padreArbin2 = padre de elem en a */

TipoA padreArbin2( Arbin a, TipoA elem )
{   TipoA aux;
    padreAux2( a, elem, &aux );
    return aux;
}

/* pre: a no tiene elementos repetidos, elem es diferente de la raíz de a */
/* post: padreArbin = se encontró el padre de elem, *padre = padre de elem */

int padreAux2( Arbin a, TipoA elem, TipoA *padre )
{   if( vacioArbin( a ) )
        return FALSE;
    else if( ( !vacioArbin( izqArbin( a ) ) && raizArbin( izqArbin( a ) ) == elem ) ||
             ( !vacioArbin( derArbin( a ) ) && raizArbin( derArbin( a ) ) == elem ) )
    {   *padre = raizArbin( a );
        return TRUE;
    }
    else if( padreAux2( izqArbin( a ), elem, padre ) )
        return TRUE;
    else
        return padreAux2( derArbin( a ), elem, padre );
}
```

Solución No. 3: Acumulación de parámetros con descenso controlado. Complejidad O(n^2).

```
/* pre: a no tiene elementos repetidos, elem está en a, elem es diferente de la raíz de a */
/* post: padreArbin3 = padre de elem en a */

TipoA padreArbin3( Arbin a, TipoA elem )
{   return padreAux3( a, elem, raizArbin( a ) );
}

/* pre: a no tiene elementos repetidos, elem está en a, padre es el elemento padre de la raíz actual */
/* post: padreAux3 = padre de elem en el árbol completo */

TipoA padreAux3( Arbin a, TipoA elem, TipoA padre )
{   if( raizArbin( a ) == elem )
        return padre;
    else if( estaArbin( izqArbin( a ), elem ) )
        return padreAux3( izqArbin( a ), elem, raizArbin( a ) );
    else
        return padreAux3( derArbin( a ), elem, raizArbin( a )  );
}
```

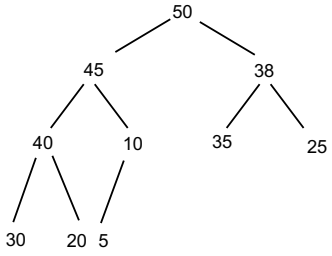
Solución No. 4: Planteamiento iterativo, en el cual se recorre el árbol en preorden, y se pregunta, para cada componente, si uno de sus hijos es el elemento buscado. Complejidad O(n).

```
/* pre: a no tiene elementos repetidos, elem está en a, elem es diferente de la raíz de a */
/* post: padreArbin4 = padre de elem en a */
```

```
TipoA padreArbin4( Arbin a, TipoA elem )
{   Pila pil = inicPila( );
    while( !vacioArbin( a ) || !vaciaPila( pil ) )
    {     if( !vacioArbin( a ) )
          {     if( ( !vacioArbin( izqArbin( a ) ) && raizArbin( izqArbin( a ) ) == elem ) ||
                    ( !vacioArbin( derArbin( a ) ) && raizArbin( derArbin( a ) ) == elem ) )
                {     destruirPila( pil );
                      return raizArbin( a );
                }
                adicPila( pil, derArbin( a ) );
                a = izqArbin( a );
          }
          else
          {     a = infoPila( pil );
                elimPila( pil );
          }
    }
}
```

Ejemplo 4.17:

Se define la **posición** de un elemento en un árbol binario como el orden en el cual se visita dicho elemento en un recorrido en inorden. De esta forma, el elemento de más a la izquierda del árbol ocupa la primera posición, y el elemento de más a la derecha, la última. Por ejemplo, para el árbol de la figura, la posición del elemento 40 es 2 y la posición del elemento 50 es 7:



Se va a desarrollar una rutina que, dados dos elementos presentes en un árbol binario, indique la diferencia de posición que hay entre ellos. Por ejemplo, para los valores 40 y 50, en el árbol de la figura anterior, la respuesta debe ser 5. Se supone que el árbol no tiene elementos repetidos.

Solución No. 1: Planteamiento recursivo basado en dos rutinas: una para calcular la posición de un elemento en un árbol binario (posArbin) y otra para restar las posiciones de los valores dados (difPosArbin1). La complejidad de esta solución es O(n).

```
/* pre: a no tiene elementos repetidos, e1 y e2 están en a */
/* post: difPosArbin1 = diferencia de posición entre e1 y e2 */

int difPosArbin1( Arbin a, TipoA e1, TipoA e2 )
{   int p1 = 0, p2 = 0;
    posArbin( a, e1, &p1 );
    posArbin( a, e2, &p2 );
    return abs( p1 - p2 );
}
```

La segunda función tiene la misma estructura de un recorrido recursivo en inorden, pero en lugar de visitar cada elemento, pregunta si es igual al valor buscado. En un parámetro adicional va llevando la posición actual, y funcionalmente informa si ha encontrado o no el elemento. En este caso se utiliza una misma variable por referencia para hacer la acumulación de parámetros y para retornar la respuesta.

```
/* pre: a no tiene elementos repetidos, *pos es la última posición visitada en el árbol completo */

/* post:  ( !posArbin, e no está en a, *pos es la última posición visitada en el árbol completo ) ∨
          ( posArbin = TRUE, e está en a, *pos es la posición del elemento e en el árbol completo, */

int posArbin( Arbin a, TipoA e, int *pos )
{   if( vacioArbin( a ) )
          return FALSE;
    else
    {     if( posArbin( izqArbin( a ), e, pos ) )
                return TRUE;
          ( *pos )++;
          if( raizArbin( a ) == e )
                return TRUE;
          else
                return posArbin( derArbin( a ), e, pos );
    }
}
```

Solución No. 2: En este segundo planteamiento se intenta realizar el mismo proceso de la solución anterior, pero sin necesidad de hacer dos recorridos sobre el árbol. Esto mantiene la complejidad igual, pero disminuye el valor de la constante asociada. La acumulación de parámetros se hace sobre una variable por referencia, y la respuesta se retorna en dos variables distintas.

```
/* pre: a no tiene elementos repetidos, e1 y e2 están en a */
/* post: difPosArbin2 = diferencia de posición entre e1 y e2 */

int difPosArbin2( Arbin a, TipoA e1, TipoA e2 )
{   int actual = 0, pos1 = -1, pos2 = -1;
    auxArbin2( a, e1, e2, &actual, &pos1, &pos2 );
    return abs( pos1 - pos2 );
}

/* pre:   a no tiene elementos repetidos, *actual es la última posición visitada en el árbol,
          si ya se ha encontrado el elemento e1 en el recorrido, su posición está en *pos1,
          si ya se ha encontrado el elemento e2 en el recorrido, su posición está en *pos2 */

/* post:  ( !auxArbin2, falta encontrar uno de los elementos, *actual es la última posición visitada en el árbol,
          si ya se ha encontrado el elemento e1 en el recorrido, su posición está en *pos1,
          si ya se ha encontrado el elemento e2 en el recorrido, su posición está en *pos2  ) ∨

          ( auxArbin2 = TRUE, *pos1 es la posición de e1, *pos2 es la posición de e2 ) */
```

```
int auxArbin2( Arbin a, TipoA e1, TipoA e2, int *actual, int *pos1, int *pos2 )
{   if( vacioArbin( a ) )
        return FALSE;
    else
    {   if( auxArbin2( izqArbin( a ), e1, e2, actual, pos1, pos2 ) )
            return TRUE;
        ( *actual )++;
        if( raizArbin( a ) == e1 )
            *pos1 = *actual;
        if( raizArbin( a ) == e2 )
            *pos2 = *actual;
        if( *pos1 != -1 && *pos2 != -1 )
            return TRUE;
        else
            return auxArbin2( derArbin( a ), e1, e2, actual, pos1, pos2 );
    }
}
```

Solución No. 3: Planteamiento iterativo que hace un recorrido en inorden y calcula en una sola pasada la diferencia de posición de los dos elementos. Complejidad O(n).

```
/* pre: a no tiene elementos repetidos, e1 y e2 están en a */
/* post: difPosArbin3 = diferencia de posición entre e1 y e2 */

int difPosArbin3( Arbin a, TipoA e1, TipoA e2 )
{   Pila pil;
    int actual = 0, pos1 = -1, pos2 = -1;
    adicPila( pil = inicPila( ), a );
    a = izqArbin( a );
    while ( !vacioArbin( a ) || !vaciaPila( pil ) )
        if( !vacioArbin( a ) )
        {   adicPila( pil, a );
            a = izqArbin( a );
        }
        else
        {   a = infoPila( pil );
            elimPila( pil );
            actual++;
            if( raizArbin( a ) == e1 )
                pos1 = actual;
            if( raizArbin( a ) == e2 )
                pos2 = actual;
            if( pos1 != -1 && pos2 != -1 )
                break;
            a = derArbin( a );
        }
    destruirPila( pil );
    return abs( pos1 - pos2 );
}
```

Ejercicios Propuestos:

4.33. TipoA posElemArbin1(Arbin a, int pos)
/* Utilizando la técnica de acumulación de parámetros, esta función retorna el elemento que se encuentra en la posición pos del árbol binario a */

4.34. TipoA posElemArbin2(Arbin a, int pos)
/* Utilizando un planteamiento iterativo, esta función retorna el elemento que se encuentra en la posición pos del árbol binario a */

4.35. Lista ramaMinima(Arbin a)
/* Suponiendo que los elementos del árbol a son enteros, se define el **costo de un camino** como la suma de los componentes de dicha secuencia. Esta función retorna la rama más barata del árbol */

4.36. Lista caminoArbin1(Arbin a, TipoA elem)
/* Utilizando la técnica de acumulación de parámetros, esta función retorna el camino que lleva de la raíz al elemento elem. Suponga que el árbol no tiene elementos repetidos */

4.37. Lista caminoArbin2(Arbin a, TipoA elem)
/* Utilizando un planteamiento iterativo, esta función retorna el camino que lleva de la raíz al elemento elem. Suponga que el árbol no tiene elementos repetidos */

4.38. TipoA mayorNivelArbin1(Arbin a, int niv)
/* Utilizando la técnica de acumulación de parámetros, esta función calcula y retorna el mayor elemento que se encuentra en el nivel niv */

4.39. TipoA mayorNivelArbin2(Arbin a, int niv)
/* Utilizando un planteamiento iterativo, esta función calcula y retorna el mayor elemento que se encuentra en el nivel niv */

4.40. TipoA mayorElemArbin1(Arbin a)
/* Utilizando un recorrido recursivo en inorden, esta función retorna el mayor elemento del árbol a */

4.41. TipoA mayorElemArbin2(Arbin a)
/* Utilizando un recorrido iterativo en inorden, esta función retorna el mayor elemento del árbol a */

4.42. Lista inordenArbin(Arbin a)
/* Utilizando un recorrido recursivo y la técnica de acumulación de parámetros, esta función retorna una lista con el recorrido en inorden del árbol binario a */

4.43. Lista primosArbin(Arbin a, TipoA elem)
/* Utilizando la técnica de acumulación de parámetros, esta función retorna una lista con los primos del elemento elem dentro del árbol. Por **primos** se entienden los hijos del hermano del padre */

4.44. Lista listaNivelArbin1(Arbin a, int n)
/* Utilizando la técnica de acumulación de parámetros, esta función retorna la lista con todos los elementos del nivel n del árbol binario a */

4.45. Lista listaNivelArbin2(Arbin a, int n)
/* Utilizando un planteamiento iterativo basado en un recorrido por niveles, esta función retorna la lista con todos los elementos del nivel n del árbol binario a */

4.46. TipoA sigInordenArbin1(Arbin a, TipoA elem)
/* Utilizando la técnica de acumulación de parámetros, esta función retorna el elemento que se encuentra después de elem en el recorrido en inorden del árbol a. Suponga que el árbol no tiene elementos repetidos */

4.47. TipoA sigInordenArbin2(Arbin a, TipoA elem)

/* Utilizando un planteamiento iterativo, esta función retorna el elemento que se encuentra después de elem en el recorrido en inorden del árbol a. Suponga que el árbol no tiene elementos repetidos */

4.48. TipoA antInordenArbin1(Arbin a, TipoA elem)

/* Utilizando la técnica de acumulación de parámetros, esta función retorna el elemento que se encuentra antes de elem en el recorrido en inorden del árbol a. Suponga que el árbol no tiene elementos repetidos */

4.49. TipoA antInordenArbin2(Arbin a, TipoA elem)

/* Utilizando un planteamiento iterativo, esta función retorna el elemento que se encuentra antes de elem en el recorrido en inorden del árbol a. Suponga que el árbol no tiene elementos repetidos */

4.50. int numElemArbin(Arbin a, TipoA elem)

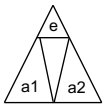
/* Utilizando un recorrido recursivo en inorden, esta función calcula el número de veces que aparece el elemento elem en el árbol a */

4.6. Implementación de Arboles Binarios

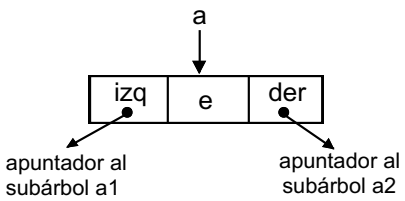
En esta sección se estudian algunas de las posibles estructuras de datos para manejar árboles binarios. Se presenta, en cada caso, un ejemplo de la manera de representar un árbol y un algoritmo que utilice dichas estructuras de datos. Para la implementación de las operaciones del TAD Arbin, es suficiente con la primera representación planteada (§4.6.1), pero para algunas modificadoras específicas, necesarias en cierto tipo de problemas, es necesario contar con estructuras de datos más complejas, como las que se muestran en las secciones siguientes.

4.6.1. Arboles Sencillamente Encadenados

Esta primera implementación es la más sencilla de todas las que se van a presentar en este capítulo, y una de las más usadas. La idea es representar un árbol binario a través de nodos encadenados, copiando la estructura del árbol con el siguiente esquema:

- El árbol a = 

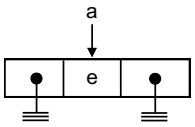
se representa con un apuntador a un nodo, el cual tiene un elemento (la raíz) y dos apuntadores a sus subárboles, como muestra la siguiente figura:



- El árbol vacío, a = Δ, se representa con un apuntador a NULL:

- El árbol a = 

, se representa con la siguiente estructura:



Ejemplo 4.18:

Para el árbol binario de la figura, las estructuras de datos que lo representan, con el esquema planteado anteriormente, son:

Como se puede apreciar en el ejemplo, esta manera de representar internamente un árbol binario resulta muy natural.

La declaración de las estructuras de datos es la siguiente:

```
typedef struct NodoArbin
{   TipoA info;
    struct NodoArbin *izq, *der;
} TArbin, *Arbin;
```

Las rutinas que implementan las operaciones del TAD Arbin resultan triviales bajo esta representación:

```
Arbin izqArbin( Arbin a )
{   return a->izq;
}

Arbin derArbin( Arbin a )
{   return a->der;
}

TipoA raizArbin( Arbin a )
{   return a->info;
}

int vacioArbin( Arbin a )
{   return a == NULL;
}
```

Ejemplo 4.19:

Reconstruir un árbol binario sin elementos repetidos, a partir de sus recorridos en inorden y preorden, suponiendo que el árbol está representado internamente con encadenamientos sencillos de apuntadores. El proceso que se sigue es el ilustrado en el ejemplo 4.14.

```
/* pre:   la lista in es el recorrido en inorden del árbol a, la lista pre es el recorrido en preorden del árbol a,
          a no tiene elementos repetidos */
/* post:  reconstruirArbin = a */

Arbin reconstruirArbin( Lista in, Lista pre )
{   return auxArbin( in, 1, longLista( in ), pre, 1 );
}
```

La rutina reconstruirArbin utiliza una rutina auxiliar (auxArbin) en la cual va descomponiendo recursivamente las listas de los recorridos, a medida que va armando y encadenando los subárboles correspondientes. Utiliza 3 parámetros adicionales.

```
/* pre:   a no tiene elementos repetidos, la lista in, en el rango d1..h1, es el recorrido en inorden del árbol a,
          la lista pre, comenzando en la posición d2, es el recorrido en preorden del árbol a */
/* post: auxArbin = a */

Arbin auxArbin( Lista in, int d1, int h1, Lista pre, int d2 )
{   TipoA elem;
    int pos;
    Arbin a;
    if( longLista( pre ) == 0 || h1 < d1 )
        return NULL;
    else
    {   posLista( pre, d2 );
        elem = infoLista( pre );
        if( h1 == d1 )
        {   a = ( Arbin )malloc( sizeof( TArbin ) );
            a->info = elem;
            a->izq = a->der = NULL;
        }
        else
        {   pos = localizarLista( in, elem, d1 );
            a = (Arbin )malloc( sizeof( TArbin ) );
            a->info = elem;
            a->izq = auxArbin( in, d1, pos - 1, pre, d2 + 1 );
            a->der = auxArbin( in, pos + 1, h1, pre, d2 + ( pos - d1 ) + 1 ) ;
        }
        return a;
    }
}
```

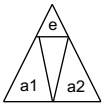
La rutina anterior utiliza una función auxiliar de manejo de listas, que localiza un elemento en una lista a partir de una posición dada:

```
/* pre: lst no tiene elementos repetidos, elem está en lst después de la posición desde */
/* post: localizarLista = posición de elem en la lista lst */

int localizarLista(Lista lst, TipoL elem, int desde )
{   for( posLista( lst, desde );  !finLista( lst ) && infoLista( lst ) != elem; sigLista( lst ), desde++ );
   return desde;
}
```

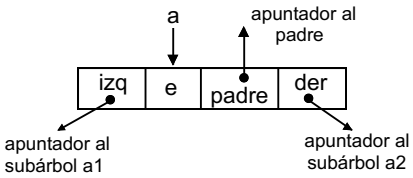
4.6.2. Arboles con Encadenamiento al Padre

Una variante de la implementación anterior, usada también con bastante frecuencia, consiste en que cada elemento del árbol mantiene un apuntador a su padre. Esto permite ascender con facilidad por la estructura jerárquica en busca de los antecesores, simplificando la expresión de algunos algoritmos. Para las operaciones analizadoras del TAD Arbin vistas hasta ahora, resulta totalmente transparente este nuevo campo, pero para algunas operaciones de recorrido y búsqueda, este campo permite la construcción de algoritmos más sencillos y eficientes, tal como se ilustra más adelante.

El esquema de representación se puede definir a través de los siguientes puntos:

- El árbol a = 

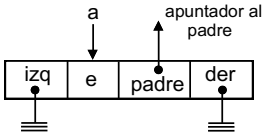
se representa con un apuntador a un nodo, el cual tiene un elemento (la raíz), dos apuntadores a sus subárboles, y un apuntador al padre del elemento e, tal como muestra la siguiente figura. Si el elemento e es la raíz del árbol completo, el apuntador al padre toma el valor NULL.



- El árbol vacío, a = Δ, se representa con un apuntador a NULL:

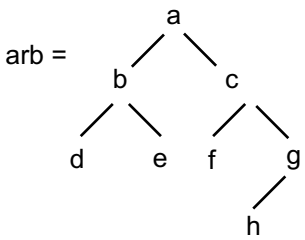
- El árbol a = 

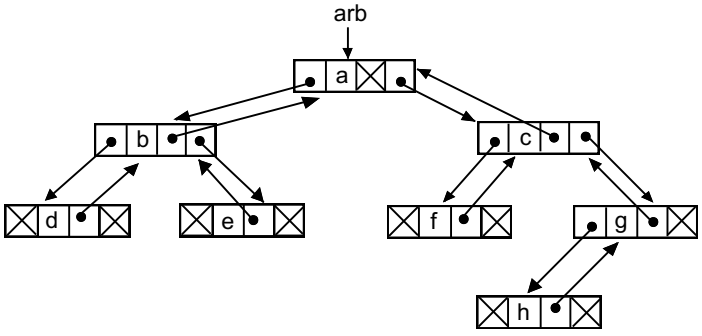
, se representa con la siguiente estructura:



Ejemplo 4.20:

Para el árbol binario de la figura, las estructuras de datos que lo representan con el esquema planteado anteriormente son:





⏎

La declaración de las estructuras de datos es la siguiente:

```
typedef struct NodoArbin
{   TipoA info;
    struct NodoArbin *izq, *der, *padre;
} TArbin, *Arbin;
```

Ejemplo 4.21:

Retornar el camino que lleva desde la raíz de un árbol hasta un elemento dado. Supone que el árbol no tiene elementos repetidos. Si no encuentra el elemento retorna una lista vacía.

Aprovecha el hecho de poder ascender por el árbol, para esperar hasta encontrar el elemento y, ahí sí, comenzar a construir el camino moviéndose por el encadenamiento del padre, hasta llegar a la raíz del árbol completo.

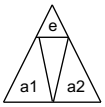
```
/* pre: a no tiene elementos repetidos */
/* post: ( caminoArbin = < >, si no existe camino de la raíz de a hasta elem ) ∨
         ( caminoArbin = camino de la raíz de a hasta elem ) */

Lista caminoArbin( Arbin a, TipoA elem )
{   Lista lst;
    if( a == NULL )
        return inicLista( );
    else if( a->info == elem )
    {   for( lst = inicLista( );  a != NULL; a = a->padre )
            insLista( lst, a->info );
        return lst;
    }
    else if( longLista( lst = caminoArbin( a->izq, elem ) )  != 0 )
        return lst;
    else if( longLista( lst = caminoArbin( a->der, elem ) )  != 0 )
        return lst;
    else
        return inicLista( );
}
```

4.6.3. Arboles Enhebrados por la Derecha

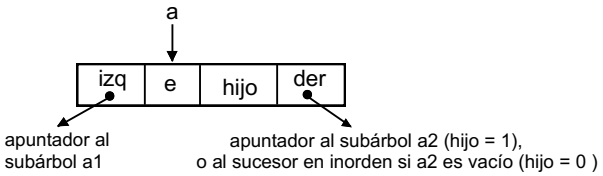
En las representaciones mostradas en las secciones anteriores, es considerable la cantidad de memoria que se desaprovecha al interior de los nodos para indicar que un subárbol asociado está vacío. La idea de esta implementación es aprovechar este espacio para mantener un encadenamiento adicional, que permita a algunas operaciones del TAD moverse con mayor facilidad al interior del árbol.

En los **árboles enhebrados por la derecha**, todas las hojas y los nodos sin subárbol derecho tienen un apuntador hacia su sucesor en el árbol en su recorrido en inorden. Para distinguir si el apuntador va a un hijo, o a su sucesor en inorden, se requiere un campo adicional en cada nodo, que indique este hecho. Este encadenamiento extra, por ejemplo, va a permitir realizar el recorrido en inorden con un algoritmo iterativo sin necesidad de utilizar una pila como estructura auxiliar.

El esquema de representación se puede definir a través de los siguientes puntos:

- El árbol a = 

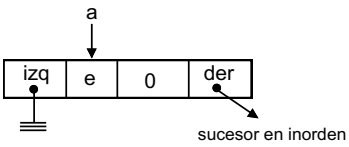
se representa con un apuntador a un nodo, el cual tiene un elemento (la raíz), dos apuntadores a subárboles y un campo que informa si el segundo apuntador tiene el subárbol derecho o el sucesor en inorden, tal como muestra la siguiente figura.



- El árbol vacío, a = Δ, se representa con un apuntador a NULL:

- El árbol a = 

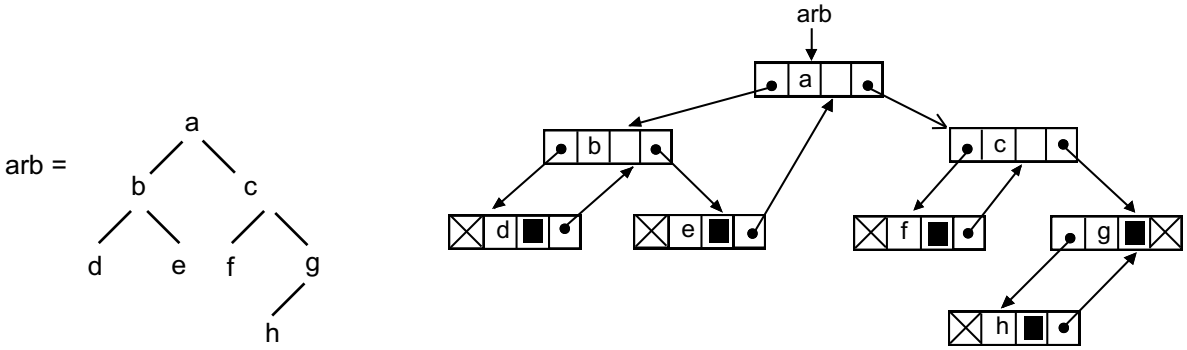
, se representa con la siguiente estructura:



Siguiendo la misma idea anterior, existen **árboles enhebrados por la izquierda** y **árboles completamente enhebrados**, según la forma como se utilicen los campos libres de los nodos para mantener encadenamientos adicionales.

Ejemplo 4.22:

Para el árbol binario de la figura, las estructuras de datos que lo representan, con el esquema de árbol enhebrado por la derecha son:



Fíjese en la necesidad del campo adicional en cada nodo y de la marca respectiva, para distinguir si el segundo apuntador está siendo utilizado para indicar el sucesor en inorden o el subárbol derecho.

⏎

La declaración de las estructuras de datos es la siguiente:

```
typedef struct NodoArbin
{   TipoA info;
    int hijo;                       /* TRUE: el campo der apunta al subárbol derecho */
                                    /* FALSE: el campo der apunta al sucesor en inorden */
    struct NodoArbin *izq, *der;
} TArbin, *Arbin;
```

La única analizadora básica del TAD Arbin que sufre cambio (con respecto a la primera representación vista) es la que retorna el subárbol derecho de un árbol binario, la cual debe verificar que el encadenamiento que va a utilizar apunte efectivamente al subárbol derecho.

```
Arbin derArbin( Arbin a )
{   return ( a->hijo ) ? a->der : NULL;
}
```

Ejemplo 4.23:

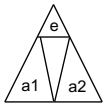
Recorrer en inorden un árbol binario enhebrado por la derecha, retornando la lista respectiva.

```
/* pre: a está enhebrado por la derecha */
/* post: inordenArbin = recorrido en inorden del árbol a */

Lista inordenArbin( Arbin a )
{   Lista lst = inicLista( );
    while( a != NULL )
    {     if( a->izq != NULL )
                a = a->izq;
          else
          {     anxLista( lst, a->info );
                if( a->hijo || a->der == NULL )
                      a = a->der;
                else
                {     a = a->der;
                      anxLista( lst, a->info );
                      a = a->der;
                }
          }
    }
    return lst;
}
```

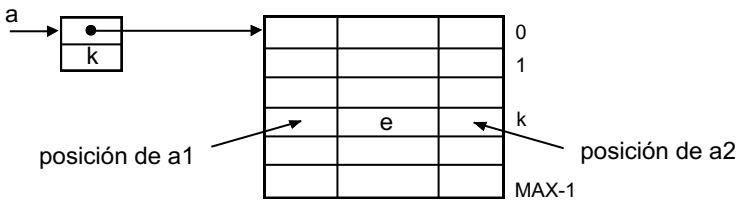
4.6.4. Cursores

En esta manera de representar un árbol binario se remplazan los apuntadores, que se manejan al interior de cada nodo, por índices en un vector. Este esquema de representación se utiliza con frecuencia para manejar estructuras arborescentes en memoria secundaria (§4.7.1). El mecanismo de encadenamiento es el mismo, pero los campos de cada nodo, que referencian los subárboles asociados, son enteros e indican su posición absoluta dentro del vector. El árbol vacío se representa con el índice -1. En caso de almacenar la información en memoria secundaria, los índices se refieren al número del registro en un archivo de acceso directo.

El esquema de representación se puede definir a través de los siguientes puntos:

- El árbol a = 

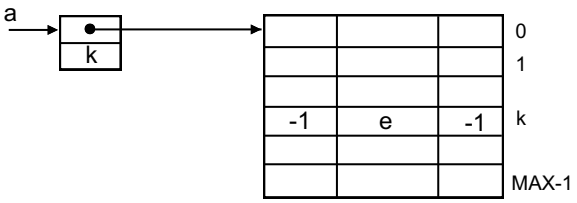
se representa con un apuntador a un nodo, el cual tiene la dirección del vector en memoria dinámica con el árbol completo, y el índice en el cual se encuentra el elemento e, tal como se muestra en la siguiente figura.



- El árbol vacío, a = Δ, se representa como:

- El árbol a = 

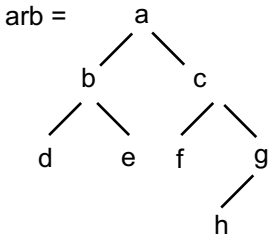
utiliza las siguientes estructuras de datos:

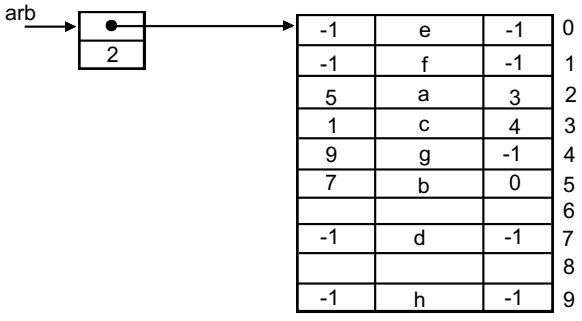


En este esquema de representación, es necesario manejar una marca especial que indique si una casilla del vector está siendo ocupada por el árbol, o si ésta se encuentra libre.

Ejemplo 4.24:

Para el árbol binario de la figura, las estructuras de datos que lo representan, con el esquema de cursores son:





La declaración de las estructuras de datos es la siguiente:

```
struct NodoArbin
{   TipoA info;
    int izq, der;
};
```

```
typedef struct
{   struct NodoArbin *vector;          /* Vector de MAX nodos en memoria dinámica */
    int arbol;                         /* Posición de la raíz en el vector */
} TArbin, *Arbin;
```

Las rutinas que implementan las operaciones del TAD Arbin son:

```
Arbin izqArbin( Arbin a )
{   if( a->vector[ a->arbol ].izq == -1 )
        return NULL;
    else
    {   Arbin aux = ( Arbin )malloc( sizeof( TArbin ) );
        aux->vector = a->vector;
        aux->arbol = a->vector[ a->arbol ].izq;
        return aux;
    }
}

Arbin derArbin( Arbin a )
{   if( a->vector[ a->arbol ].der == -1 )
        return NULL;
    else
    {   Arbin aux = ( Arbin )malloc( sizeof( TArbin ) );
        aux->vector = a->vector;
        aux->arbol = a->vector[ a->arbol ].der;
        return aux;
    }
}

TipoA raizArbin( Arbin a )
{   return a->vector[ a->arbol ].info;
}

int vacioArbin( Arbin a )
{   return a == NULL;
}
```

Ejemplo 4.25:

Agregar una operación al TAD Arbin que permita encontrar el padre de un elemento dado, en un árbol sin valores repetidos, suponiendo que este elemento se encuentra presente en la estructura de datos y es distinto de la raíz.

Puesto que se tiene acceso a las estructuras de datos, basta con hacer un recorrido por el vector, tratando de localizar un elemento que tenga un subárbol situado en la posición del elemento elem.

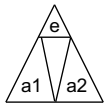
```
/* pre: a no tiene elementos repetidos, elem está en a, elem es diferente de la raíz de a */
/* post: padreArbin = padre de elem en a */
```

```
TipoA padreArbin( Arbin a, TipoA elem )
{   int i, k;
    for( k = 0; a->vector[ k ].info != elem; k++ );
    for( i = 0; a->vector[ i ].izq != k && a->vector[ i ].der != k; i++ );
    return a->vector[ i ].info;
}
```
⏎

4.6.5. Representación Secuencial

Para representar internamente un árbol binario, es posible utilizar como estructura de soporte una lista (o cualquier otra estructura lineal), con el siguiente esquema de representación:

- El árbol a = Δ se representa con la lista vacía a = < >.
- El árbol a = 

se representa con la lista a = < e >.
- El árbol a = 

se representa con la lista a = < x_1, ..., x_n >, donde:

  (1) Si el árbol a tiene altura k, $n \leq 2^k - 1$.

  (2) x_i es un elemento del árbol a o la marca especial $\otimes$. Dado que $\otimes$ debe ser un elemento válido del tipo TipoA, $\otimes$ debe seleccionarse del dominio de valores de dicho tipo.

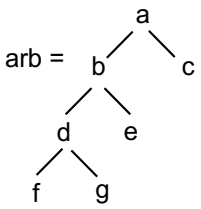
  (3) Cada elemento del árbol se sitúa en la lista en la posición que dicho elemento ocuparía en el recorrido por niveles del árbol binario lleno de altura igual a la del árbol a (ver ejemplo 4.26).

  (4) Todas las posiciones no ocupadas de la lista se completan con la marca $\otimes$.

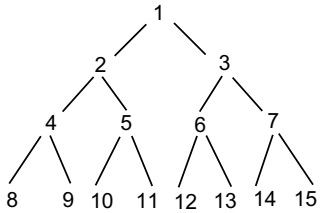
  (5) x_n != $\otimes$.

Ejemplo 4.26:

Suponga que se quiere hacer una representación secuencial del siguiente árbol:



Teniendo en cuenta que, cuando se hace el recorrido por niveles en un árbol binario lleno de altura 4, se visitan los elementos en el siguiente orden :



Las posiciones en que deben colocarse los elementos del árbol arb en la lista que lo representa son:

a = 1, b = 2, c = 3, d = 4, e = 5, f = 8, g = 9

Obteniendo la lista:

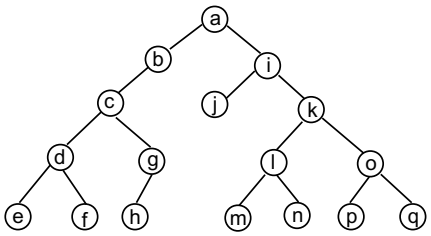
arb = < a, b, c, d, e, ⊗, ⊗, f, g >

⏎

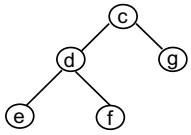
Un hecho importante en esta representación es que si un elemento se encuentra en la posición pos de la lista, su hijo izquierdo está en la posición (pos * 2) y su hijo derecho en la posición (pos * 2) + 1. De la misma manera, su padre se encuentra en la posición [pos / 2].

Ejemplo 4.27:

Otros ejemplos de representación secuencial se pueden apreciar en los siguientes árboles:



< a b i c ⊗ j k d g ⊗ ⊗ ⊗ ⊗ l o e f h ⊗ ⊗ ⊗ ⊗ ⊗ ⊗ ⊗ ⊗ ⊗ m n p q >



< c, d, g, e, f >

Note que no es necesario colocar en la lista todos los elementos del último nivel, sino únicamente hasta el último elemento presente.

⏎

La declaración de las estructuras de datos para esta representación es:

```
typedef TipoA TipoL;
typedef Lista Arbin;
```

Las rutinas que implementan las operaciones del TAD Arbin se dan a continuación:

- La rutina que calcula el subárbol izquierdo se divide en dos partes. La primera, crea la lista con el subárbol, y, la segunda, elimina del final de la lista todas las marcas. Para construir la lista, la rutina se sitúa en la segunda posición, y, comenzando desde allí, lleva a cabo el siguiente proceso: pasar a la lista de respuesta 1 elemento y avanzar 1 posición, pasar a la lista de respuesta 2 elementos y avanzar 2 posiciones, pasar a la lista de respuesta 4 elementos y avanzar 4 posiciones, etc. Utiliza el procedimiento auxiliar avanzarLista que se muestra más adelante.

```
Arbin izqArbin( Arbin a )
{   int i, paso = 1;
    Arbin a1 = inicLista( );
    if( longLista( a ) >= 2 )
    {     for( posLista( a, 2 ); !finLista( a ); avanzarLista( a, paso ), paso *= 2 )
                for( i = 1; !finLista( a ) && i <= paso; i++, sigLista( a ) )
                      anxLista( a1, infoLista( a ) );
          while( longLista( a1 ) != 0  && infoLista( a1 ) == MARCA )
          {     elimLista( a1 );
                ultLista( a1 );
          }
    }
    return a1;
}
```

- La rutina avanzarLista es un procedimiento auxiliar que permite mover la ventana hacia adelante un número dado de posiciones:

```
void avanzarLista( Lista lst, int num )
{   for( ; !finLista( lst ) && num-- > 0; sigLista( lst ) );
}
```

- La operación que calcula el subárbol derecho tiene una estructura parecida a la explicada para la rutina izqArbin, pero el proceso de construcción del árbol es: se sitúa en la posición 3, toma 1 elemento y avanza 2 pasos, toma 2 elementos y avanza 4 pasos, etc.

```
Arbin derArbin( Arbin a )
{   int i, paso = 1;
    Arbin a1 = inicLista( );
    if( longLista( a ) >= 3 )
    {     for( posLista( a, 3 ); !finLista( a ); avanzarLista( a, paso * 2 ), paso *= 2 )
                for( i = 1; !finLista( a ) && i <= paso; i++, sigLista( a ) )
                      anxLista( a1, infoLista( a ) );
          while( longLista( a1 ) != 0  && infoLista( a1 ) == MARCA )
          {     elimLista( a1 );
                ultLista( a1 );
          }
    }
    return a1;
}
```

- La raíz del árbol se encuentra en la primera posición de la lista. Puesto que la precondición de la operación garantiza que el árbol no es vacío, no se debe hacer ninguna validación.

```
TipoA raizArbin( Arbin a )
{   primLista( a );
    return infoLista( a );
}
```

- El árbol está vacío si la lista que lo representa está vacía, según se definió en el esquema de representación.

```
int vacioArbin( Arbin a )
{   return longLista( a ) == 0;
}
```

Ejemplo 4.28:

Agregar al TAD Arbin una operación que haga un recorrido en inorden de un árbol representado con una estructura lineal.

```
/* post:  se ha recorrido en inorden el árbol binario a */

void inordenArbin( Arbin a )
{   inorden2( a, 1 );
}

/* pre: pos es la posición en la lista a del subárbol que se va a comenzar a recorrer */
/* post: se ha recorrido en inorden el subárbol que comienza en la posición pos */

void inorden2( Arbin a, int pos )
{   Ventana v;
    if( pos <= longLista( a ) )
    {     posLista( a, pos );
          if( infoLista( a ) != MARCA )
          {     v = ventanaLista( a );
                inorden2( a, pos * 2 );
                situarLista( a, v );
                printf("%d ",infoLista( a ) );
                inorden2( a, ( pos * 2 ) + 1 );
          }
    }
}
```

4.7. Destrucción y Persistencia de Árboles Binarios

Para la persistencia de árboles binarios, se agregan dos operaciones al TAD Arbin. Una para leer un árbol de un archivo (cargarArbin) y otra para salvarlo (salvarArbin).

En esta sección se estudian dos esquemas diferentes de persistencia, basados en las representaciones de cursores (§4.6.4) y listas (§4.6.5).

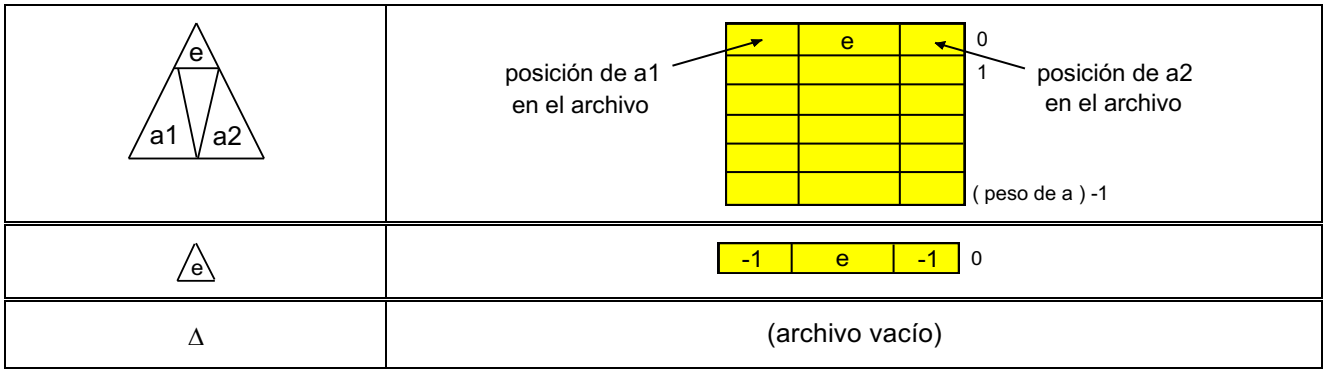
Persistencia:		
• cargarArbin:	FILE *	→ Arbin
• salvarArbin:	Arbin x FILE *	

```
Arbin cargarArbin( FILE *fp )
/* Construye un árbol binario a partir de la información de un archivo */

{ pre: el archivo está abierto y es estructuralmente correcto  }
{ post: se ha construido el árbol que corresponde a la imagen de la información del archivo }
```

```
void salvarArbin( Arbin a, FILE *fp )
/* Salva el árbol en un archivo */

{ pre: el archivo está abierto }
{ post: se ha hecho persistir el árbol en el archivo }
```

4.7.1. Persistencia con Cursores

Este esquema de persistencia utiliza un archivo de acceso directo, con registros de 3 campos: uno con la información del elemento, y los otros dos con la posición absoluta en el archivo de los subárboles izquierdo y derecho. La raíz del árbol completo se encuentra en el primer registro del archivo. El archivo tiene tantos registros como elementos tiene el árbol, de tal manera que un árbol vacío se representa con un archivo vacío. El esquema de persistencia se puede resumir en el siguiente dibujo:



Entre las diversas posibilidades que existen para implementar las dos operaciones de persistencia, se decidió agregar un campo a cada nodo del árbol, en el cual se incluye la posición en el archivo de dicho elemento. Por esta razón, la rutina hace dos pasadas sobre el árbol: en la primera asigna una posición en el archivo a cada nodo, y, en la segunda, salva los elementos siguiendo el esquema de persistencia antes planteado.

Para la implementación de apuntadores (§4.6.1), las estructuras de datos se modifican de la siguiente manera:

```
typedef struct NodoArbin
{   TipoA info;
    int posArch;                    /* posición en el archivo en la cual debe persistir el elemento info */
    struct NodoArbin *izq,*der;
} TArbin,*Arbin;
```

También es necesario declarar el tipo de registro que se va a almacenar en el archivo:

```
struct regArbin
{   TipoA info;                     /* elemento del árbol */
    int izq,der;                    /* posición en el archivo de los dos subárboles */
};
```

La rutina que lee un árbol de un archivo verifica que éste no se encuentre vacío, y, luego, hace un recorrido en preorden reconstruyendo el árbol. Puesto que la precondición garantiza que la estructura del archivo es correcta, la función no hace ningún tipo de validación con respecto a la información allí almacenada.

```
Arbin cargarArbin( FILE *fp )
{   return auxCargar( fp, 1 );
}

Arbin auxCargar( FILE *fp, int posArch )
{   struct regArbin reg;
    Arbin a = ( Arbin )malloc( sizeof( TArbin ) );
    fseek( fp, ( posArch - 1 ) * sizeof( struct regArbin ), SEEK_SET );
    if( fread(&reg,sizeof(struct regArbin),1,fp) == 0 )
        return NULL;
    a->info = reg.info;
    a->izq = ( reg.izq == -1 ) ? NULL : auxCargar( fp, reg.izq );
    a->der = ( reg.der == -1 ) ? NULL: auxCargar( fp, reg.der );
    return a;
}
```

Para salvar el árbol se utilizan dos rutinas auxiliares, encargadas de llenar el campo de posición en el archivo (marcarArbin) y de escribir la información a medida que se hace un recorrido en preorden (auxSalvar):

```
void salvarArbin( Arbin a, FILE *fp )
{   int posArch = 1;
    marcarArbin( a, &posArch );
    auxSalvar( a, fp );
}

void marcarArbin( Arbin a, int *actual )
{   if( a != NULL )
    {   a->posArch = ( *actual )++;
        marcarArbin( a->izq, actual );
        marcarArbin( a->der, actual );
    }
}

void auxSalvar( Arbin a, FILE *fp )
{   struct regArbin reg;
    if( a != NULL )
    {   reg.izq = ( a->izq == NULL ) ? -1 : a->izq->posArch;
        reg.der = ( a->der == NULL ) ? -1 : a->der->posArch;
        reg.info = a->info;
        fwrite( &reg, sizeof( struct regArbin ), 1, fp );
        auxSalvar( a->izq, fp );
        auxSalvar( a->der, fp );
    }
}
```

4.7.2. Persistencia con Representación Secuencial

En este esquema de persistencia se utiliza un archivo de texto, en el cual se coloca en cada posición (línea) el elemento que en la representación secuencial de árboles debe ocupar dicho espacio. Las rutinas se basan en un recorrido por niveles de los árboles. En la primera línea del archivo aparece el número de elementos que se salvaron en el archivo.

```
Arbin cargarArbin( FILE *fp )
{   Cola col;
    Arbin resp, a, a1, a2;
    int i, alt, numElem, elem;
    fscanf( fp, "%d", &alt );
    if( ( numElem = pow( 2, alt ) - 1 ) == 0 )
        return NULL;
    fscanf( fp, "%d", &elem );
    resp = crearNodo( elem );
    adicCola( col = inicCola( ), resp );
    for( i = 1; i < numElem; i += 2 )
    {   a = infoCola( col );
        elimCola( col );
        if ( a != NULL )
        {   fscanf( fp, "%d", &elem );
            a->izq = a1 = crearNodo( elem );
            adicCola( col, a1 );
            fscanf( fp, "%d", &elem );
            a->der = a2 = crearNodo( elem );
            adicCola( col, a2 );
        }
        else
        {   fscanf( fp, "%d", &elem );
            fscanf( fp, "%d", &elem );
            adicCola( col, NULL );
            adicCola( col, NULL );
        }
    }
    destruirCola( col );
    return resp;
}
```

La rutina crearNodo tiene la responsabilidad de crear un nuevo nodo en memoria dinámica e inicializar su campo de información con el parámetro que se le pase.

```
Arbin crearNodo( TipoA elem )
{   Arbin a = NULL;
    if( elem != MARCA )
    {   a = ( Arbin )malloc( sizeof( TArbin ) );
        a->info = elem;
        a->izq = a->der = NULL;
    }
    return a;
}
```

El proceso de salvar un árbol en un archivo se basa en un recorrido por niveles de la estructura. Se crea una lista con todos los elementos de cada nivel, pero se debe tener cuidado de completar las posiciones no ocupadas con la MARCA preestablecida. Al final de cada nivel, si existe algún elemento allí, se escriben los elementos en el archivo. En caso contrario se termina el proceso.

```
void salvarArbin( Arbin a, FILE *fp )
{   Lista lst = inicLista( );
    Cola col;
    int numElem = 1, i = 0, nivelVacio = TRUE;
    fprintf( fp, "%d\n", alturaArbin( a ) );
    if ( a != NULL )
    {     adicCola( col = inicCola( ) ,a );
          while( !vaciaCola( col ) )
          {     a = infoCola( col );
                elimCola( col );
                if (a != NULL )
                {     nivelVacio = FALSE;
                      anxLista( lst, a->info );
                      adicCola( col, a->izq );
                      adicCola( col, a->der );
                }
                else
                {     anxLista( lst, MARCA );
                      adicCola( col, NULL );
                      adicCola( col, NULL );
                }
                if( ++i == numElem )
                {     if( nivelVacio )
                      {     destruirCola( col );
                            destruirLista( lst );
                            return;
                      }
                      for( primLista( lst ); !finLista( lst ); sigLista( lst ) )
                            fprintf( fp, "%d ", infoLista( lst ) );
                      fprintf( fp, "\n" );
                      destruirLista( lst );
                      lst = inicLista( );
                      nivelVacio = TRUE;
                      numElem *= 2;
                      i = 0;
                }
          }
    }
}
```

4.7.3. Destructora del TAD Arbin

Para facilitar el proceso de destrucción de un árbol binario, se agrega al TAD Arbin una operación destructora, encargada de devolver el espacio ocupado en memoria dinámica por las estructuras de datos. Esta operación es diferente para cada representación interna.

Destructora:
• destruirArbin: Arbin

```
void destruirArbin( Arbin a )
/* Destruye un árbol binario, retornando toda la memoria ocupada en su representación */

{ post: se ha devuelto toda la memoria ocupada en la representación del árbol a. a es indefinido }
```

Para las representaciones de apuntadores (árboles sencillamente encadenados, árboles con encadenamiento al padre y árboles enhebrados por la derecha), la operación destructora se implementa con la siguiente rutina recursiva:

```
void destruirArbin( Arbin a )
{   if( a != NULL )
    {     destruirArbin( a->izq );
          destruirArbin( a->der );
          free( a );
    }
}
```

Ejercicios Propuestos:

Utilizando la representación de árboles sencillamente encadenados desarrolle las siguiente rutinas:

4.51. Arbin reflejarArbin(Arbin a)
/* Dado un árbol binario a, este procedimiento retorna su **reflejo**. Esto es, para cada elemento del árbol, intercambia sus subárboles asociados */

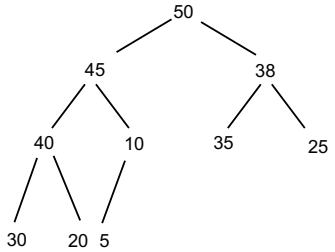
4.52. void reemplazarArbin(Arbin a, TipoA elem1, TipoA elem2)
/* Reemplaza en el árbol a todas las ocurrencias del elemento elem1 por el elemento elem2 */

4.53. Arbin podar1Arbin(Arbin a)
/* Elimina del árbol a todas sus hojas */

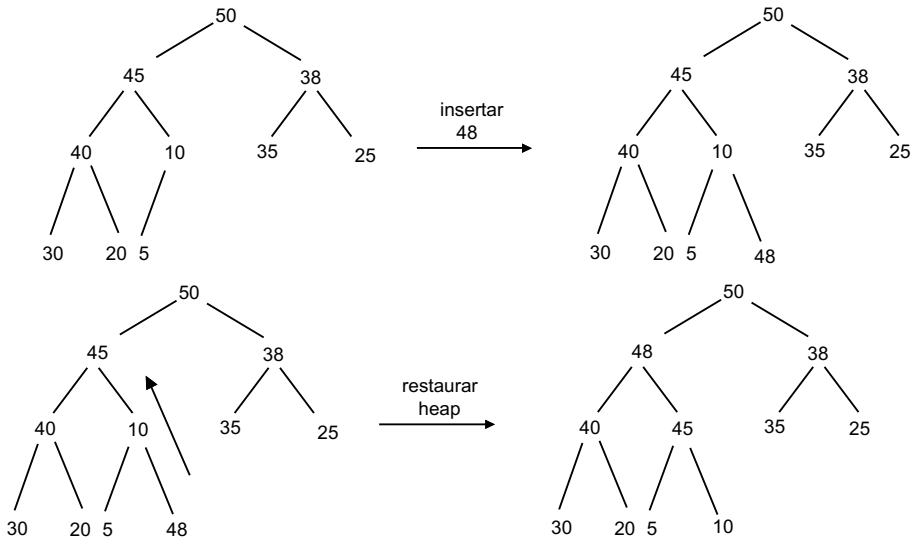
4.54. Arbin podar2Arbin(Arbin a, int niv)
/* Elimina del árbol a todos los elementos que se encuentran en un nivel superior o igual a niv */

4.55. 🕒 Arbin eliminarArbin(Arbin a, TipoA elem)
/* Elimina del árbol binario a el elemento elem, de tal manera que el árbol resultante conserve el mismo inorden, salvo por el elemento retirado */

4.56. 🕒 Arbin insHeap(Arbin a, TipoA elem)
/* Un ***heap*** es un árbol binario casi lleno, en el cual se cumple que todo elemento es mayor que sus hijos y los subárboles asociados son a su vez *heaps*. Un ejemplo de un *heap* es el siguiente:



La inserción de un elemento en un *heap* se hace colocándolo en la siguiente posición libre (para que siga siendo casi lleno), y, después, subiéndolo por la jerarquía hasta encontrar el punto adecuado según su valor. Por ejemplo, si al *heap* anterior se le agrega el elemento 48, el proceso que se sigue es:



Esta función agrega un elemento a un *heap* */

Utilizando la representación de árboles con encadenamiento al padre desarrolle las siguiente rutinas:

4.57. void encadenarPadre(Arbin a)
/* Suponiendo que el campo padre de cada nodo del árbol se encuentra en NULL, esta función hace el encadenamiento respectivo */

4.58. TipoA ancestroComún(Arbin a, TipoA elem1, TipoA elem2)
/* Busca y retorna el ancestro común más próximo de los elementos elem1 y elem2, aprovechando el encadenamiento al padre. Suponga que no hay elemento repetidos */

4.59. Arbin podar2(Arbin a, int niv)
/* Elimina del árbol a todos los elementos que se encuentran en un nivel superior o igual a niv */

4.60. int nivelArbin(Arbin a, TipoA elem)
/* Retorna el nivel en el que se encuentra el elemento elem en el árbol a, aprovechando el encadenamiento que cada nodo tiene a su padre */

4.61. 🕒 Arbin reconstruyeArbin(Lista postorden, Lista inorden)
/* Dados los recorridos en postorden e inorden de un árbol binario sin elementos repetidos, esta función reconstruye y devuelve el árbol que los generó */

Utilizando la representación de árboles enhebrados por la derecha desarrolle las siguientes rutinas:

4.62. Arbin eliminarArbin(Arbin a, TipoA elem)
/* Elimina del árbol binario a el elemento elem, de tal manera que el árbol resultante conserve el mismo inorden, salvo por el elemento retirado */

4.63. void enhebrarDerecha(Arbin a)
/* Suponiendo que el campo que hace el enhebramiento por la derecha en cada nodo se encuentra en NULL, este procedimiento hace dicho encadenamiento */

4.64. void enhebrarHojasDerecha(Arbin a)
/* Suponiendo que el campo que hace el enhebramiento por la derecha en cada nodo se encuentra en NULL, esta rutina lo aprovecha para encadenar las hojas del árbol de izquierda a derecha */

Utilizando la representación de cursores desarrolle las siguientes rutinas:

4.65. Arbin podarHojas(Arbin a, TipoA elem)
/* Elimina del árbol binario a todas las hojas cuyo valor es igual a elem */

4.66. void niveles(Arbin a)
/* Recorre por niveles el árbol binario a */

Utilizando la representación secuencial de un árbol binario, desarrolle las siguientes rutinas:

4.67. int esHoja(Arbin a, int pos)
/* Determina si el elemento que se encuentra en la posición pos es una hoja */

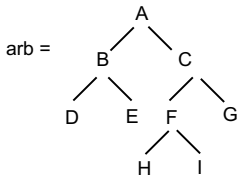
4.68. int estaArbin(Arbin a, TipoA elem)
/* Determina si el valor elem se encuentra en el árbol binario a. La función debe ser O(n), donde n es el peso del árbol */

4.69. Arbin podar2(Arbin a, int niv)
/* Elimina del árbol a todos los elementos que se encuentran en un nivel superior o igual a niv */

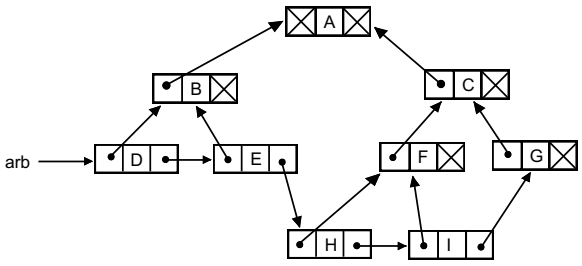
4.70. void nivelesArbin(Arbin a)
/* Hace el recorrido por niveles del árbol a */

4.71. Arbin consArbin(TipoA elem, Arbin a1, Arbin a2)
/* Retorna un árbol cuya raíz es elem, subárbol izquierdo es a1 y subárbol derecho es a2. No modifica los parámetros de entrada */

4.72. Con el fin de "ahorrar espacio" en la representación de árboles binarios, alguien decidió que en lugar de que el padre tenga apuntadores a los hijos, sean los hijos quienes guarden un apuntador al padre. Para resolver el problema de localizar las hojas del árbol (nada apunta a ellas, pues no tienen hijos), decidió encadenar las hojas una con otra, de izquierda a derecha. Bajo esta representación, el árbol binario:



Se representa como:



De acuerdo con este esquema de representación, un árbol binario está definido por un apuntador a la primera hoja, de tal manera que, a partir de este elemento, sea posible tener acceso a los demás componentes del árbol.

Las estructuras de datos de esta representación son las siguientes:

```
typedef struct Nodo
{     TipoA info;
      struct Nodo *padre, *sigHoja;
} TArbin, *Arbin;
```

Dado que los nodos de esta representación tienen la misma estructura interna (un campo de tipo TipoA y dos de tipo apuntador), desarrolle un algoritmo para pasar un árbol de esta representación a la representación estándar con apuntadores. El algoritmo debe reutilizar los nodos de la representación inicial, y retornar un apuntador a la raíz.

4.8. EL TAD Arbol Binario Ordenado

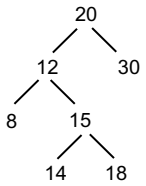
Una de las aplicaciones más frecuentes de árboles binarios es el almacenamiento de información de manera ordenada, mejorando la eficiencia de acceso a la información con respecto a las estructuras lineales. La idea es colocar todos los elementos menores que la raíz en el subárbol izquierdo, y todos los elementos mayores en el derecho, de manera que, cada vez que la operación de búsqueda pasa por un elemento del árbol, descarta todos los valores que se encuentran en uno de los dos subárboles asociados. Si se repite recursivamente este esquema para cada subárbol, se obtiene una estructura de datos llamada **árbol binario ordenado**.

Formalmente, se dice que un árbol binario es ordenado si todos los elementos del subárbol izquierdo son menores que la raíz, todos los del derecho mayores que la raíz y los dos subárboles asociados son, a su vez, árboles binarios ordenados.

Una propiedad interesante de este tipo de árbol binario es que, en el recorrido en inorden, se visitan siempre los elementos en orden ascendente de valor.

Ejemplo 4.29:

El siguiente es un ejemplo de un árbol binario ordenado:



El recorrido en inorden de este árbol es < 8 12 14 15 18 20 30 >, que corresponde a una secuencia ordenada ascendentemente.

⏎

Ejemplo 4.30:

Verificar si un árbol binario es ordenado. La primera solución que se presenta en este ejemplo es de complejidad O(n^2), y se basa en la definición recursiva de árbol binario ordenado. La segunda manera, es hacer un recorrido en inorden y verificar que los elementos se visiten en orden ascendente (O (n)).

Solución No. 1 - Utiliza dos funciones auxiliares que retornan el mayor y el menor elemento de un árbol binario ordenado.

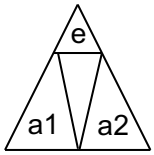
```
/* post: ordenadoArbin1 = el árbol a es ordenado */

int ordenadoArbin1( Arbin a )
{   if( vacioArbin( a ) || ( vacioArbin( izqArbin( a ) ) && vacioArbin( derArbin( a ) ) ) )
        return TRUE;
    else
        return ordenadoArbin1( izqArbin( a ) ) &&
            ordenadoArbin1( derArbin( a ) ) &&
            ( vacioArbin( izqArbin( a ) ) || mayorArbin( izqArbin( a ) ) < raizArbin( a ) ) &&
            ( vacioArbin( derArbin( a ) ) || menorArbin( derArbin( a ) ) > raizArbin( a ) );
}

/* pre: a es ordenado */
/* post: mayorArbin = mayor elemento de a */

int mayorArbin( Arbin a )
{   return ( vacioArbin( derArbin( a ) ) ) ? raizArbin( a ) : mayorArbin( derArbin( a ) );
}

/* pre: a es ordenado */
/* post: menorArbin = menor elemento de a */

int menorArbin( Arbin a )
{   return ( vacioArbin( izqArbin( a ) ) ) ? raizArbin( a ) : menorArbin( izqArbin( a ) );
}
```

Solución No. 2 - Hace un recorrido en inorden, y va llevando en un parámetro adicional por referencia el último valor visitado del árbol.

```
/* post: ordenadoArbin2 = el árbol a es ordenado */

int ordenadoArbin2( Arbin a )
{   int aux = VACIO;
    return esOrdenado( a, &aux );
}

/* pre: *ant es el último elemento del árbol completo que se ha visitado en el inorden */
/* post: esOrdenado = el árbol a es ordenado, *ant es el último elemento del árbol que se ha visitado en el inorden */

int esOrdenado( Arbin a, int *ant )
{   if( vacioArbin( a ) )
        return TRUE;
    else if( esOrdenado( izqArbin( a ), ant ) )
        if( *ant != VACIO && *ant > raizArbin( a ) )
            return FALSE;
        else
        {   *ant = raizArbin( a );
            return esOrdenado( derArbin( a ), ant );
        }
    else return FALSE;
}
```

El TAD que maneja árboles binarios ordenados (TAD ArbinOr) utiliza como formalismo para su objeto abstracto el mismo que utiliza el TAD Arbin. La única diferencia se encuentra en el invariante, que agrega condiciones especiales para este nuevo objeto. Fuera de las operaciones analizadoras, que ya vienen con el TAD Arbin, este nuevo TAD agrega una constructora, dos modificadoras y una analizadora.

TAD ArbinOr[TipoAO]



{ inv: a1 y a2 son disyuntos, todos los elementos de a1 son menores que e,
todos los elementos de a2 son mayores que e, a1 y a2 son ordenados }

Constructora:

- inicArbinOr: → ArbinOr

Modificadoras:

- insArbinOr: ArbinOr x TipoAO → ArbinOr
- elimArbinOr: ArbinOr x TipoAO → ArbinOr

Analizadora:

- estaArbinOr: ArbinOr x TipoAO → int

```
ArbinOr inicArbinOr( void )
/* Crea un árbol binario ordenado vacío */

{ post: inicArbinOr = Ø }
```

```
ArbinOr insArbinOr( ArbinOr a, TipoAO elem )
/* Agrega un elemento a un árbol binario ordenado */

{ pre: elem ∉ a, a = A }
{ post: insArbinOr = A ∪ { elem } }
```

```
ArbinOr elimArbinOr( ArbinOr a, TipoAO elem )
/* Elimina un elemento de un árbol binario ordenado */

{ pre: elem ∈ a, a = A}
{ post: elimArbinOr = A - { elem } }
```

```
int estaArbinOr( ArbinOr a, TipoAO elem )
/* Informa si un elemento se encuentra en un árbol binario ordenado */

{ post: estaArbinOr = ( elem ∈ a ) }
```

Se presenta a continuación la implementación de estas tres operaciones, sobre el esquema de representación de árboles sencillamente encadenados, como una manera de ilustrar el proceso que se debe seguir.

4.8.1. Proceso de Búsqueda

La operación de búsqueda aprovecha la estructura de un árbol binario ordenado, bajando únicamente por los subárboles en los cuales existe la posibilidad de encontrar el elemento. En el peor de los casos, el algoritmo es O(n), donde n es el peso del árbol, ya que su máxima altura puede ser n (un árbol **degenerado**) como se muestra en la figura 4.9. En el caso promedio, el algoritmo es O($\log_2$ n), donde n es el peso del árbol, ya que todos los valores son igualmente probables en la secuencia de inserción.

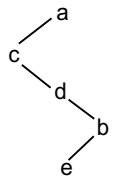


Fig. 4.9 - Árbol binario degenerado (peso = altura)

Cuando se trabaja sobre árboles ordenados, casi siempre resulta igual de sencilla la solución iterativa a la solución recursiva, puesto que no es necesario el uso de una pila para moverse por la estructura. La rutina recursiva que informa si un elemento está en un árbol ordenado es:

```
int estaArbinOr( ArbinOr a, TipoAO elem )
{   if(  a == NULL  )
        return FALSE;
    else if(  a->info == elem  )
        return TRUE;
    else if(  a->info > elem  )
        return estaArbinOr( a->izq, elem );
    else
        return estaArbinOr( a->der, elem );
}
```

La rutina iterativa para realizar esa misma operación es:

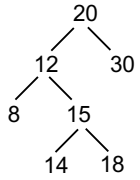
```
int estaArbinOr( ArbinOr a, TipoAO elem )
{   while( a != NULL && a->info != elem )
        a = ( elem < a->info ) ? a->izq : a->der;
    return a != NULL;
}
```

4.8.2. Proceso de Inserción

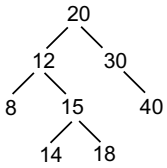
El proceso para agregar un elemento a un árbol binario ordenado tiene la misma estructura del proceso de búsqueda. La idea es seguir la misma manera de bajar por el árbol, y, en el momento de llegar al punto en el cual se reconoce que el elemento no está presente (no se puede bajar más), agregar allí el elemento.

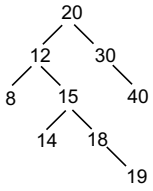
Ejemplo 4.31:

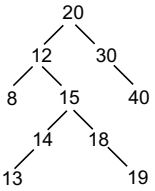
Si se tiene el árbol que se muestra en la figura:



Los árboles que se obtienen al insertarle la secuencia de valores 40, 19 y 13 son los que se presentan a continuación. Es importante anotar que siempre existe un punto en el árbol donde se puede agregar un nuevo elemento, de manera que sea una hoja. Esto facilita el proceso de inserción, puesto que agregar en un lugar interior podría significar cambios estructurales del árbol:







⏎

A continuación se presentan las implementaciones recursiva e iterativa de la operación de inserción, las cuales son O(n), en el peor de los casos, y O($\log_2$ n), en el caso promedio, donde n es el peso del árbol.

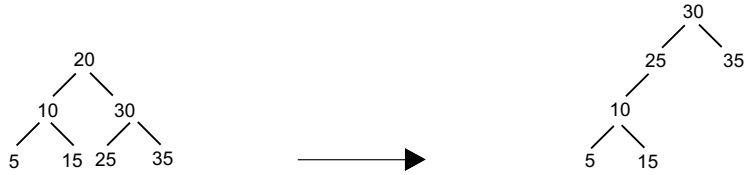
```
ArbinOr insArbinOr( ArbinOr a, TipoAO elem )
{   if( a == NULL )
    {     a = ( ArbinOr )malloc( sizeof( TArbinOr ) );
          a->info = elem;
          a->izq = a->der = NULL;
    }
    else if( a->info > elem )
          a->izq = insArbinOr( a->izq, elem );
    else
          a->der = insArbinOr( a->der, elem );
    return a;
}

ArbinOr insArbinOr( ArbinOr a, TipoAO elem )
{   ArbinOr resp = a, aux = ( ArbinOr )malloc( sizeof( TArbinOr ) );
    aux->info = elem;
    aux->izq = aux->der = NULL;
    if( a == NULL )
          return aux;
    while( ( elem < a->info && a->izq != NULL) || ( elem > a->info && a->der != NULL ) )
          a = ( elem < a->info ) ? a->izq : a->der;
    if( elem < a->info )
          a->izq = aux;
    else
          a->der = aux;
    return resp;
}
```

4.8.3. Proceso de Eliminación

El proceso para eliminar un elemento de un árbol binario es más complicado que los dos anteriores, porque al suprimir un valor se debe alterar la estructura del árbol. Existen varias maneras de hacerlo, las cuales se ilustran a continuación, mostrando el caso general y un ejemplo.

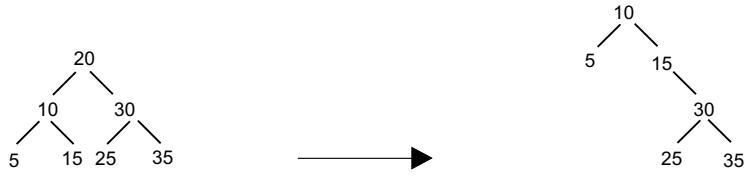
Opción 1: Para eliminar el elemento de la raíz, se puede colocar el subárbol izquierdo a la izquierda del menor elemento del subárbol derecho:

Por ejemplo, al eliminar el elemento 20 se tiene:



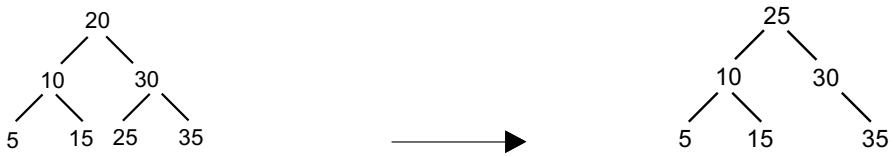
Opción 2: Para eliminar el elemento de la raíz, se puede colocar el subárbol derecho a la derecha del menor elemento del subárbol izquierdo:

Por ejemplo, al eliminar el elemento 20 se tiene:



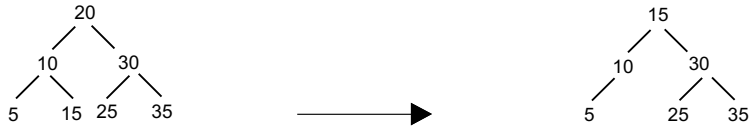
Opción 3: Para eliminar el elemento de la raíz, se puede reemplazar dicho elemento por el menor elemento del subárbol derecho:

Por ejemplo, al eliminar el elemento 20 se tiene:



Opción 4: Para eliminar el elemento de la raíz, se puede reemplazar el elemento por el mayor elemento del subárbol izquierdo:

Por ejemplo, al eliminar el elemento 20 se tiene:



El algoritmo para implementar la supresión de un elemento utilizando la cuarta opción, considera tres grandes casos: el elemento es la raíz, el elemento está en el subárbol izquierdo o el elemento está en el subárbol derecho. En el primer caso aplica la solución planteada anteriormente, utilizando una función que retorna el mayor elemento de un árbol binario. Los otros dos casos hacen avanzar la recursión, por el subárbol respectivo.

Cuando el elemento que se quiere eliminar es la raíz, considera tres casos: es una hoja (la elimina sin ningún problema), el árbol no tiene subárbol izquierdo (coloca el subárbol derecho en lugar de todo el árbol), o tiene ambos subárboles (busca el mayor elemento del subárbol izquierdo, lo coloca en la raíz, y hace una llamada recursiva para suprimirlo de dicho subárbol).

El algoritmo recursivo es el siguiente:

```
ArbinOr elimArbinOr( ArbinOr a, TipoAO elem )
{   ArbinOr p;
    TipoAO mayor;
    if(  a->info == elem  )
    {     if(  a->izq == NULL && a->der == NULL  )
          {     free( a );
                return NULL;
          }
          else if(  a->izq == NULL  )
          {     p = a->der;
                free( a );
                return p;
          }
```

```
            else
            {   mayor = mayorElemento( a->izq );
                a->info = mayor;
                a->izq = elimArbinOr( a->izq,mayor );
            }
        }
        else if(  a->info>elem  )
            a->izq = elimArbinOr( a->izq,elem );
        else
            a->der = elimArbinOr( a->der,elem );
        return a;
    }
```

La solución iterativa viene dada por el siguiente código en C:

```
    ArbinOr elimArbinOr(ArbinOr a, TipoAO elem)
    {   ArbinOr padre1 = NULL, padre2, a1 = a, a2;
        while( a1 != NULL )
        {   while( a1->info != elem )
            {   padre1 = a1;
                a1 = ( elem < a1->info ) ? a1->izq : a1->der;
            }
            if( a1->izq == NULL && a1->der == NULL )
            {   if( padre1 == NULL )
                    a = NULL;
                else if( padre1->izq == a1 )
                    padre1->izq = NULL;
                else
                    padre1->der = NULL;
                break;
            }
            else if( a1->izq == NULL )
            {   /* solo tiene hijo derecho */
                if( padre1 == NULL )
                    a = a->der;
                else if( padre1->izq == a1 )
                    padre1->izq = a1->der;
                else
                    padre1->der = a1->der;
                break;
            }
            else if( a1->der == NULL )
            {   /* solo tiene hijo izquierdo */
                if( padre1 == NULL )
                    a = a->izq;
                else if( padre1->izq == a1 )
                    padre1->izq = a1->izq;
                else
                    padre1->der = a1->izq;
                break;
            }
```

```
            else
            {     /* tiene ambos hijos */
                  for( a2=a1->der, padre2=a1; a2->izq != NULL; padre2 = a2, a2=a2->izq );
                  a1->info = a2->info;
                  elem = a2->info;
                  padre1 = padre2;
                  a1 = a2;
            }
      }
      free( a1 );
      return a;
}
```

Ejercicios Propuestos:

4.73. ArbinOr elimArbinOr(ArbinOr a, TipoAO elem)
/* Elimina el elemento elem del árbol binario ordenado a, utilizando la opción No. 1 explicada en la sección anterior */

4.74. TipoA casiArbinOr(Arbin a)
/* Se define un árbol **casi ordenado** como un árbol binario en el cual todos los elementos cumplen las condiciones de orden del árbol, salvo uno. Esta función detecta dicho elemento y lo retorna. La complejidad de la rutina es O(n), donde n es el peso del árbol */

4.75. ArbinOr unirArbinOr(ArbinOr a1, ArbinOr a2)
/* Une al árbol binario ordenado a1 el árbol binario ordenado a2 (unión de conjuntos) */

4.76. ArbinOr interArbinOr(ArbinOr a1, ArbinOr a2)
/* Elimina de a1 los elementos que no se encuentran en a2 */

4.77. int subArbinOr(ArbinOr a1, ArbinOr a2)
/* Indica si todos los elementos de a2 están en a1 */

4.78. Un **Generador de Referencias Cruzadas** es un programa que trabaja sobre un texto y produce una lista alfabética con las palabras allí contenidas, indicando, para cada palabra, los renglones en los que aparece y el número total de apariciones.

Para esto se debe manejar internamente un directorio de palabras encontradas y una lista para cada una de ellas, en la cual aparezcan los números de los renglones donde se encuentra presente la palabra.

Se puede definir el TAD Directorio con las siguientes operaciones, con el fin de dar soporte al generador de referencias:

TAD Directorio		
• inicDir:		→ Directorio
• adicPalDir:	Directorio x String x int	→ Directorio
• elimPalDir:	Directorio x String	→ Directorio
• renglonesDir	Directorio x String	→ Lista[int]
• listarDir:	Directorio	→ Lista[String]

a-) Especifique formalmente el TAD Directorio e impleméntelo sobre árboles binarios ordenados y listas.
b-) Escriba un Generador de Referencias Cruzadas utilizando el TAD Directorio desarrollado en el punto anterior.

4.79. Implemente el TAD ArbinOr utilizando una estructura con encadenamiento al padre.

4.80. Implemente el TAD ArbinOr utilizando una estructura de árbol enhebrado por la derecha.

4.81. Implemente el TAD ArbinOr utilizando cursores como representación interna.

4.82. Implemente el TAD ArbinOr utilizando una representación secuencial.

4.83. Implemente el TAD ArbinOr utilizando árboles completamente enhebrados.

4.84. Implemente el TAD ArbinOr utilizando **árboles encadenados por niveles**. En este caso, en cada nodo de las estructuras de datos se mantiene un apuntador al siguiente elemento en el recorrido por niveles.

4.9. Arboles Binarios Ordenados Balanceados

Un problema con el TAD ArbinOr es que no es suficiente con mantener el árbol ordenado para garantizar eficiencia en el acceso a la información, puesto que en el peor de los casos la complejidad de las operaciones sigue siendo O(n), aunque en el caso promedio sea O($\log_2 n$). Una posible solución es exigir que los dos subárboles asociados con cada elemento tengan aproximadamente el mismo número de componentes, garantizando de esta manera que se descarta la mitad de los elementos de la estructura, en cada paso de la búsqueda. Los árboles con dicha característica se denominan **balanceados**, y garantizan que la operación de búsqueda de un elemento tiene complejidad O($\log_2 n$), en el peor de los casos, a costa de algoritmos más complicados para implementar las modificadoras, puesto que deben alterar en cada inserción y supresión la estructura del árbol, para garantizar la condición de balanceo.

Existen básicamente dos tipos de árboles binarios balanceados: los árboles **AVL** (o balanceados por altura) y los árboles **perfectamente balanceados** (o balanceados por peso). Adelson - Velskii y Landis introdujeron en 1962 el concepto de árbol balanceado por altura, y de allí su nombre de árboles AVL. En este tipo de árboles, las alturas de los dos subárboles asociados con cada elemento no pueden diferir en más de 1, y los dos subárboles deben ser también AVL (figura 4.10). Por definición, un árbol binario vacío es AVL.

• avl(a) ssi abs(altura(izqArbin(a)) - altura(derArbin(a))) ≤ 1 ∧ avl(izqArbin(a)) ∧ avl(derArbin(a))

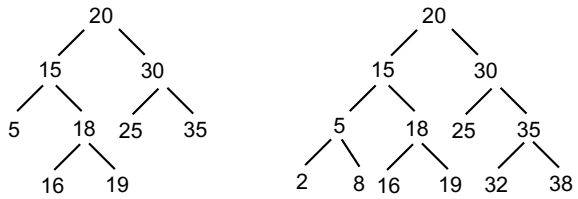


Fig. 4.10 - Ejemplos de árboles AVL

En los árboles perfectamente balanceados, la noción de equilibrio viene dada por el peso de los subárboles: el número de elementos en cada uno de los subárboles asociados no puede diferir en más de uno, y los dos subárboles asociados deben ser también perfectamente balanceados (figura 4.11). Por definición, un árbol binario vacío es perfectamente balanceado.

• bal(a) ssi abs(peso(izqArbin(a)) - peso(derArbin(a))) ≤ 1 ∧ bal(izqArbin(a)) ∧ bal(derArbin(a))

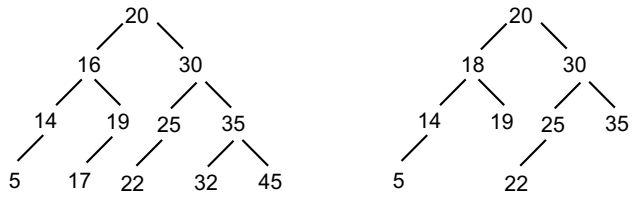
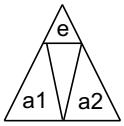


Fig. 4.11 - Ejemplos de árboles perfectamente balanceados

De acuerdo con las definiciones dadas, todo árbol balanceado por peso es en especial AVL, pero no todo árbol AVL es balanceado por peso. En este capítulo únicamente se trata la algorítmica de los árboles AVL. Se recomienda consultar la bibliografía para ver las rutinas que implementan las modificadoras de un árbol perfectamente balanceado.

4.9.1. El TAD AVL

En el TAD AVL sólo existen una constructora (crear un árbol AVL vacío) y dos modificadoras (insertar y eliminar un elemento), que garanticen las condiciones de balanceo que debe cumplir el árbol. La analizadora de búsqueda es igual a la de árboles binarios ordenados.

TAD AVL[TipoAVL]		
		
{ inv: a1 y a2 son disyuntos, todos los elementos de a1 son menores que e, todos los elementos de a2 son mayores que e, a1 y a2 son ordenados, | altura (a1) - altura (a2) | ≤ 1, a1 y a2 son AVL }		
Constructora:		
• inicAVL:		→ AVL
Modificadoras:		
• insAVL:	AVL x TipoAVL	→ AVL
• elimAVL:	AVL x TipoAVL	→ AVL

```
AVL inicAVL ( void )
/* Crea un árbol AVL vacío */

{ post: inicAVL = ∅ }
```

```
AVL insAVL ( AVL a, TipoAVL elem )
/* Adiciona un elemento a un árbol AVL */

{ pre: a = A, elem ∉ a }
{ post: insAVL = A ∪ { elem } }
```

AVL elimAVL (AVL a, TipoAVL elem)
/* Elimina un elemento de un árbol AVL */
{ pre: a = A, elem ∈ a }
{ post: elimAVL = A - { elem } }

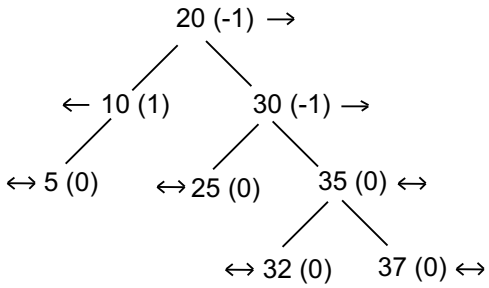
4.9.2. Estructuras de Datos

La implementación de las operaciones del TAD AVL se ilustra utilizando el esquema de representación de árboles sencillamente encadenados, extendido con un campo en cada nodo (balan) para incluir un indicador del estado de balanceo, que corresponde a la diferencia de altura de sus subárboles (izq - der). Este nuevo campo es utilizado por las modificadoras para dirigir la estrategia de rebalanceo de un árbol. Las convenciones gráficas son las siguientes:

- balan = 0 (↔): ambos subárboles tienen la misma altura
- balan = -1 (→): el subárbol derecho excede en 1 la altura del izquierdo
- balan = 1 (←): el subárbol izquierdo excede en 1 la altura del derecho

Ejemplo 4.32:

El siguiente árbol AVL tendría en sus estructuras de datos los factores de balance que se muestran en el dibujo:



⏎

La declaración de las estructuras de datos es:

```
typedef struct NodoAVL
{   TipoAVL info;
    struct NodoAVL *izq, *der;
    int balan;
} TAVL, *AVL;
```

Las constantes para representar los factores de desbalanceo en los algoritmos son:

```
#define IZQ   1
#define BAL  0
#define DER  -1
```

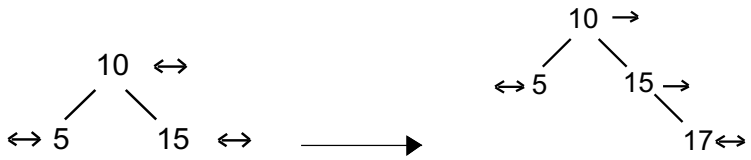
4.9.3. Algoritmo de Inserción

El proceso para insertar un nuevo elemento en un árbol AVL se divide en dos etapas: primero, se agrega el elemento como en un árbol binario ordenado cualquiera y, luego, se restaura el balanceo del árbol mediante ajustes de la estructura que se denominan **rotaciones**. Los casos de desbalanceo de un árbol y las rotaciones respectivas para balancearlo se presentan a continuación, puesto que son el fundamento de la estructura del algoritmo de inserción.

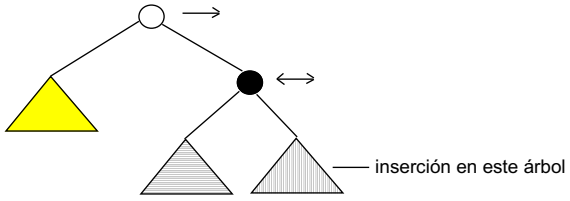
Caso 1: No se dañó el balanceo al insertar el elemento

Proceso: No se debe rebalancear. Sólo se debe actualizar el indicador de balanceo en los elementos afectados

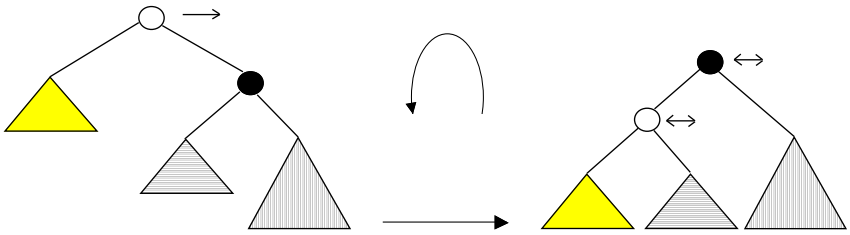
Ejemplo: Resultado de insertar el elemento 17



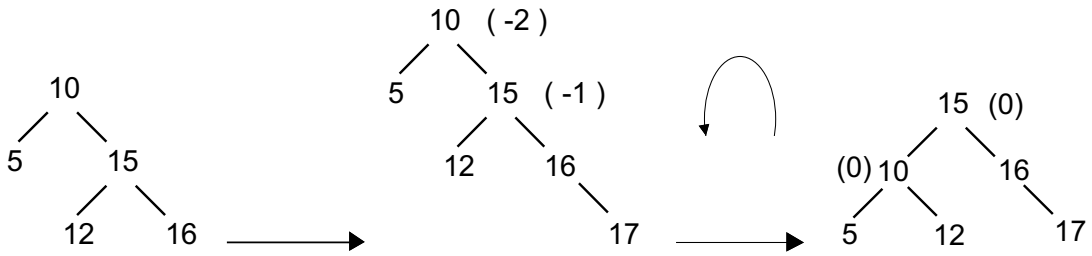
Caso 2: El árbol inicial es de la forma mostrada en la figura, y se inserta el nuevo elemento en el subárbol derecho.



Proceso: Rebalancear el árbol con una rotación a la izquierda, de la siguiente manera:



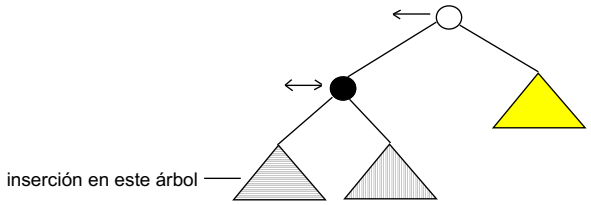
Ejemplo: Resultado de insertar el elemento 17



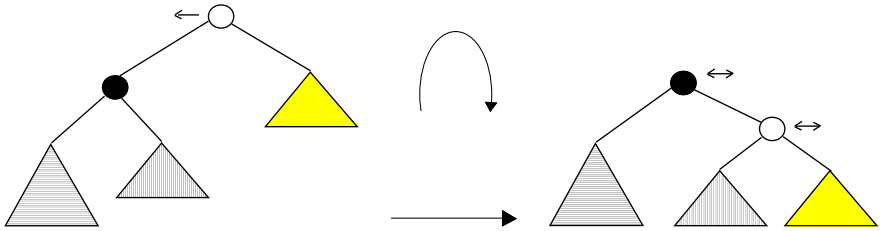
Rutina para la rotación a la izquierda (no recalcula el factor de balance):

```
AVL roteIzq( AVL a )
{    AVL temp = a->der;
     a->der = temp->izq;
     temp->izq = a;
     return temp;
}
```

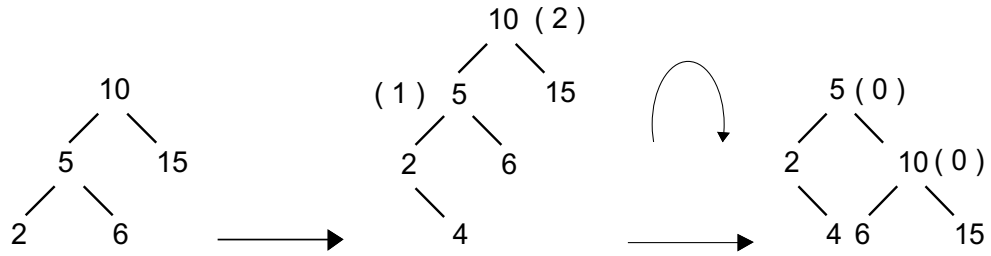
Caso 3: El árbol inicial es de la forma mostrada en la figura, y se inserta el nuevo elemento en el subárbol izquierdo. Es un caso simétrico al anterior.



Proceso: Rebalancear el árbol con una rotación a la derecha, de la siguiente manera:



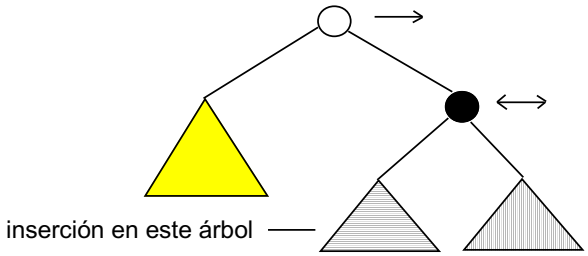
Ejemplo: Resultado de insertar el elemento 4



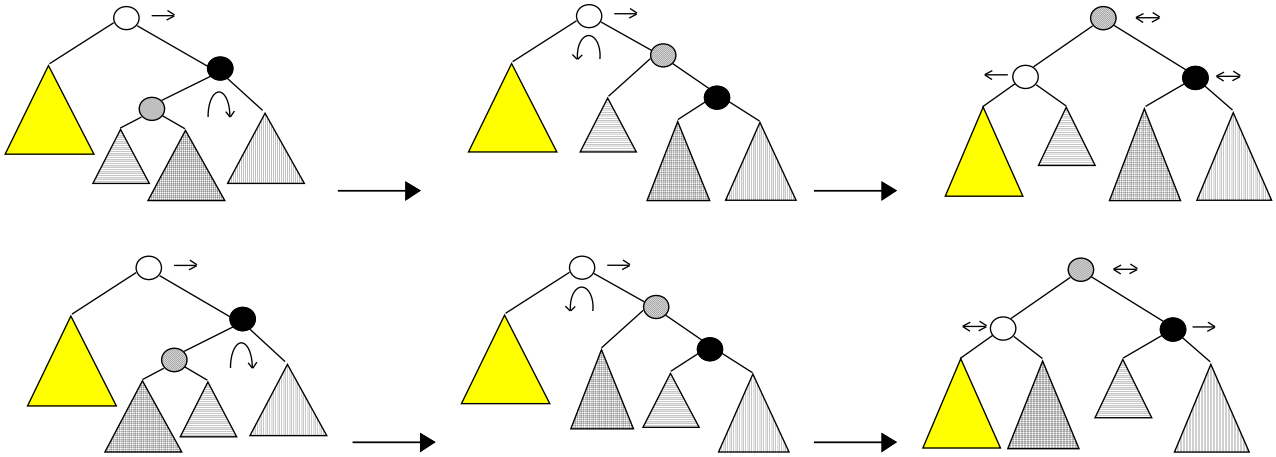
Rutina para la rotación a la derecha (no recalcula el factor de balance):

```
AVL roteDer( AVL a )
{    AVL temp = a->izq;
     a->izq = temp->der;
     temp->der = a;
     return temp;
}
```

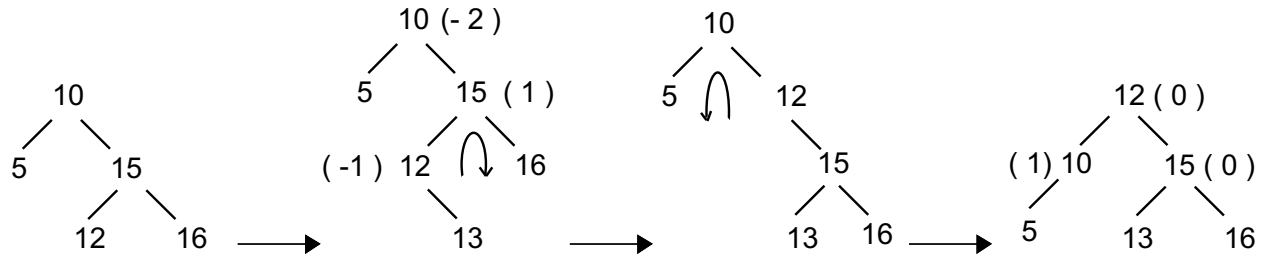
Caso 4: El árbol inicial es de la forma mostrada en la figura, y se inserta el nuevo elemento en un subárbol interno.



Proceso: Rebalancear el árbol con una doble rotación derecha-izquierda (subárbol derecho una rotación a la derecha, y, luego, el árbol completo una rotación a la izquierda). Pueden darse dos casos distintos, con respecto al subárbol del subárbol interno que se desbalancea, pero ambos se resuelven con la misma estrategia de rotación, de la siguiente manera:



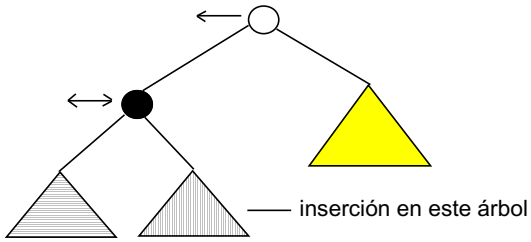
Ejemplo: Resultado de insertar el elemento 13



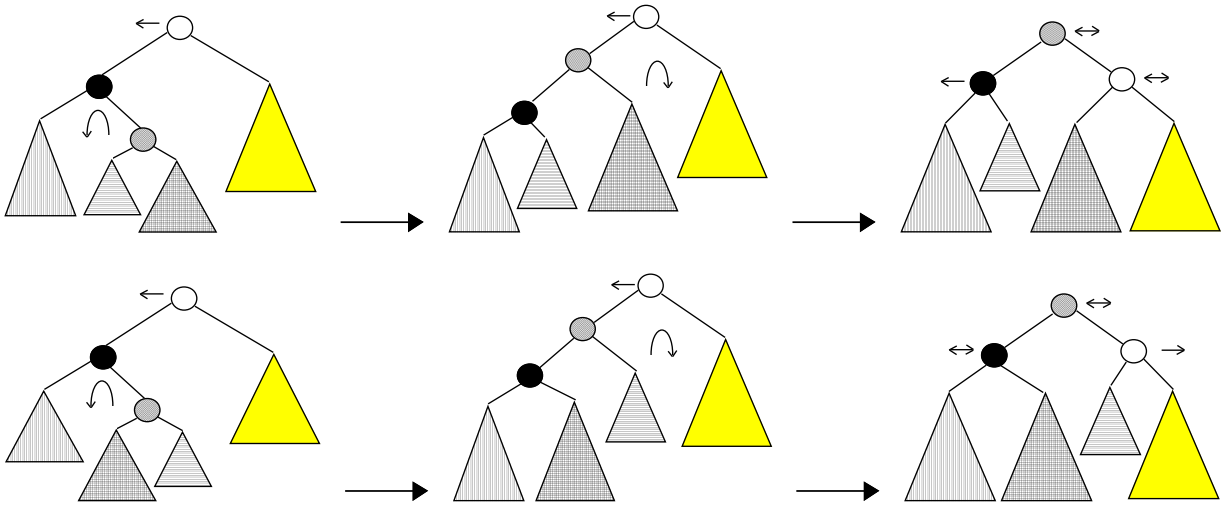
Rutina para la doble rotación derecha-izquierda (no recalcula el factor de balance):

```
AVL roteDerIzq( AVL a )
{     a->der = roteDer( a->der );
      return roteIzq( a );
}
```

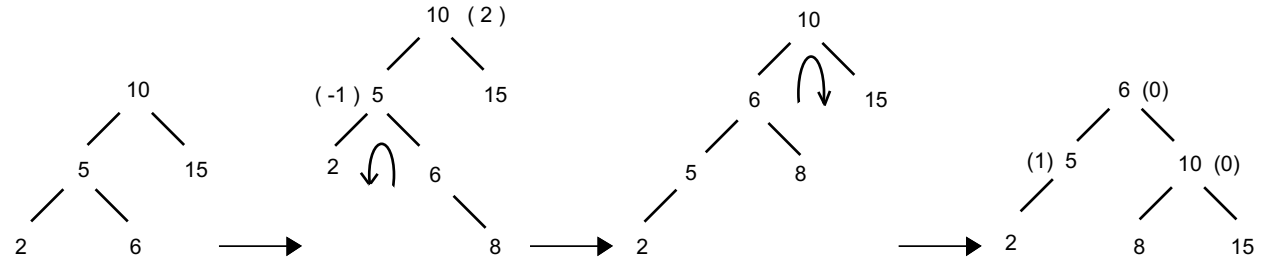
Caso 5: El árbol inicial es de la forma mostrada en la figura, y se inserta el nuevo elemento en un subárbol interno. Es un caso simétrico al anterior:



Proceso: Rebalancear el árbol con una doble rotación izquierda-derecha (subárbol izquierdo una rotación a la izquierda, y, luego, el árbol completo una rotación a la derecha). Pueden darse dos casos distintos, con respecto al subárbol del subárbol interno que se desbalancea, pero ambos se resuelven con la misma estrategia de rotación, de la siguiente manera:



Ejemplo: rebalanceo del árbol al insertar el 8



Rutina para la doble rotación izquierda-derecha (no recalcula el factor de balance):

```
static AVL roteIzqDer( AVL a )
{     a->izq = roteIzq( a->izq );
      return roteDer( a );
}
```

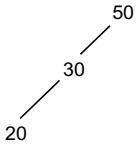
Ejemplo 4.33:

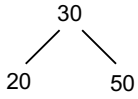
Mostrar el proceso de creación de un árbol AVL al insertar la siguiente secuencia de números: 50 - 30 -20 - 60 - 70 - 55 - 57 - 58. Los pasos se dan a continuación:

1. Insertar el 50 - Caso 1

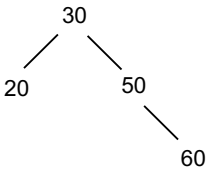
2. Insertar el 30 - Caso 1

3. Insertar el 20 - Caso 3 (rotación derecha)

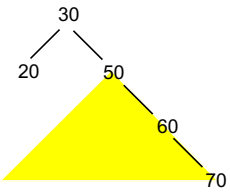




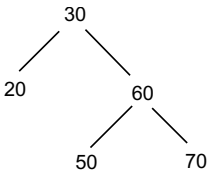
4. Insertar el 60 - Caso 1

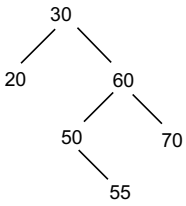


5. Insertar el 70 - Caso 2 (rotación izquierda del subárbol derecho, mínimo subárbol desbalanceado)

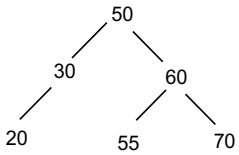


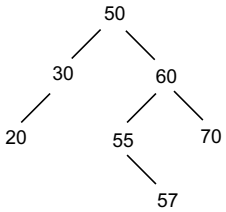
6. Insertar el 55 - Caso 4 (rotación derecha izquierda)





7. Insertar el 57 - Caso 1





8. Insertar el 58 - Caso 4 (rotación derecha-izquierda del mínimo subárbol desbalanceado)

La rutina completa, para balancear el subárbol derecho de un árbol AVL (el balanceo del izquierdo es equivalente), establece a cuál de los dos casos posibles de desbalanceo corresponde (2 ó 4) y llama la rutina respectiva, actualizando luego los indicadores que lo requieran.

```
/* pre: se ha insertado un elemento en el subárbol derecho de a y se ha desbalanceado */
/* post: retorna un árbol AVL con todos los elementos de a */

AVL balanceaDer ( AVL a )
{   if( a->der->balan == DER )
    {     /* Caso 2 */
          a->balan = a->der->balan = BAL;
          a = roteIzq( a );
    }
    else
    {     /* Caso 4 */
           switch( a->der->izq->balan )
          { case IZQ:     a->balan = BAL;
                          a->der->balan = DER;
                          break;
            case BAL:     a->balan = a->der->balan = BAL;
                          break;
            case DER:     a->balan = IZQ;
                          a->der->balan = BAL;
                          break;
          }
          a->der->izq->balan = BAL;
          a = roteDerIzq( a );
    }
    return a;
}
```

La operación de inserción en un árbol AVL se basa en varias rutinas, las cuales se presentan a continuación, y utilizan los algoritmos de rebalanceo descritos en cada caso. La primera rutina (insAVL) corresponde a la operación de inserción en un árbol AVL; crea el nodo que se va a adicionar y abre espacio para el manejo de

un parámetro adicional, que se va a manejar durante todo el proceso de inserción, para informar si su altura aumentó después de alguna modificación estructural del árbol.

```
AVL insAVL( AVL a, TipoAVL elem )
{   AVL p = ( AVL )malloc( sizeof( TAVL ) );
    int masAlto;
    p->izq = p->der = NULL;
    p->info = elem;
    p->balan = BAL;
    return insertar( a, p, &masAlto );
}
```

La rutina insertar recibe un árbol AVL y un nodo con el elemento que se quiere agregar, y retorna funcionalmente un árbol AVL con el nuevo nodo agregado a la estructura inicial. Informa, en un parámetro por referencia, si la altura del árbol resultante es mayor que la altura del árbol inicial. La rutina considera tres casos principales:

- el árbol es vacío: retorna el nodo apuntado por p
- el nuevo elemento es menor que la raíz: inserta el nodo apuntado por p en el subárbol izquierdo, y, si la altura del árbol aumenta, lo rebalancea según el factor de balance de la raíz
- el nuevo elemento es mayor que la raíz: inserta el nodo apuntado por p en el subárbol derecho, y, si la altura del árbol aumenta, lo rebalancea según el factor de balance de la raíz

```
/* pre: p->info ∉ a, a = A */
/* post:  insertar = A ∪ p, *masAlto = altura( a ) > altura( A ) */

AVL insertar( AVL a , Nodo AVL *p, int *masAlto )
{   if( a == NULL )
    {     *masAlto = TRUE;
          a = p;
    }
    else if( a->info > p->info )
    {     a->izq = insertar( a->izq, p, masAlto );
          if( *masAlto )
          switch( a->balan )
          { case IZQ:     *masAlto = FALSE;
                          a = balanceaIzq( a );
                          break;
            case BAL:     a->balan = IZQ;
                          break;
            case DER:     *masAlto = FALSE;
                          a->balan = BAL;
                          break;
          }
    }
```

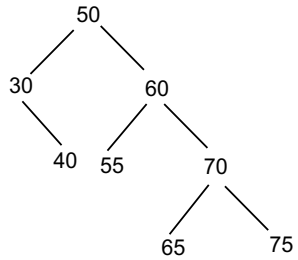
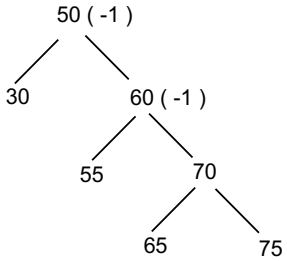
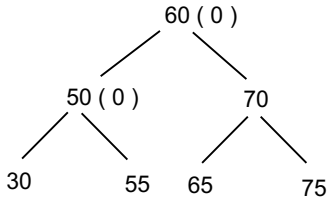
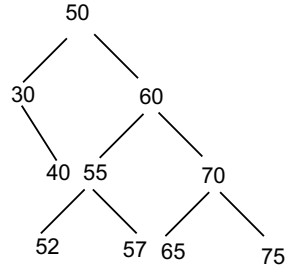
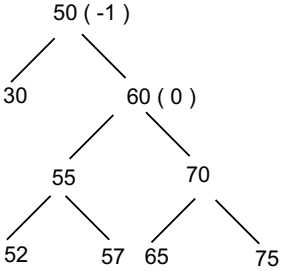
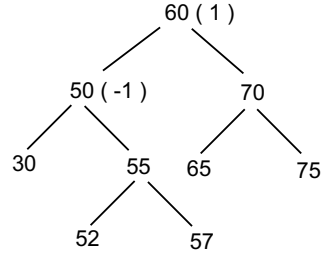
```
    else
    {     a->der = insertar( a->der, p, masAlto );
          if( *masAlto )
          switch( a->balan )
          { case IZQ:     *masAlto = FALSE;
                          a->balan = BAL;
                          break;
            case BAL:     a->balan = DER;
                          break;
            case DER:     *masAlto = FALSE;
                          a = balanceaDer( a );
          }
    }
    return a;
}
```

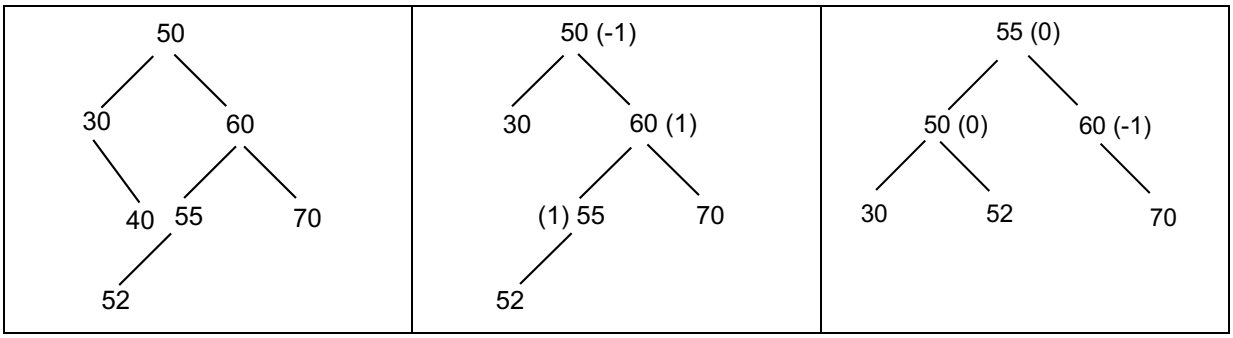
4.9.4. Algoritmo de Eliminación

El proceso que se sigue para eliminar un elemento de un árbol AVL se basa en los mismos esquemas de rotación mostrados en la sección anterior, como se puede apreciar en el siguiente ejemplo.

Ejemplo 4.34:

En cada uno de los siguientes tres casos se muestra el árbol original, el árbol después de eliminar el elemento 40 (sin actualizar los factores de balanceo) y el árbol después de aplicar una rotación izquierda.

Arbol antes de eliminar el 40	Arbol sin actualizar los factores de desbalance	Rotación izquierda
		
		



El algoritmo que implementa la operación de eliminación de un elemento (elimAVL) utiliza un conjunto de rutinas auxiliares, similares a las que utiliza el algoritmo de inserción, y las cuales se presentan a continuación. Se utiliza durante el proceso un parámetro auxiliar por referencia (menosAlto), que indica si el árbol ha perdido un nivel al haber sido retirado el elemento pedido.

```
AVL elimAVL( AVL a, TipoAVL elem )
{   int menosAlto;
    return eliminar( a, elem, &menosAlto );
}
```

La función eliminar sigue el mismo proceso descrito en una sección anterior para suprimir un elemento de un árbol binario ordenado: se busca una hoja del árbol que pueda remplazar la raíz, y, luego, se elimina dicha hoja de la estructura. La rutina debe incluir código para considerar los casos de desbalanceo a los que se puede llegar después de suprimir dicho elemento.

```
AVL eliminar(AVL a, TipoAVL elem, int *menosAlto)
{   AVL p;
    if( a->info == elem )
    {     if( a->izq == NULL && a->der == NULL )
          {     free(a);
                *menosAlto = TRUE;
                return NULL;
          }
          else if( a->izq == NULL )
          {     p = a->der;
                free( a );
                *menosAlto = TRUE;
                return p;
          }
          else
          {     a->izq = eliminar( a->izq, a->info = mayorArbinOr( a->izq ), menosAlto );
                if( *menosAlto )
                      a = balanDer( a, menosAlto );
          }
    }
    else if( a->info>elem )
    {     a->izq = eliminar(a->izq, elem, menosAlto );
          if( *menosAlto )
                a = balanDer( a, menosAlto );
    }
    else
```

```
    {     a->der = eliminar( a->der, elem, menosAlto );
          if( *menosAlto )
                a = balanIzq( a, menosAlto );
    }
    return a;
}
```

Las funciones balanDer y balanIzq tienen una estructura similar, y se encargan de reestablecer el balanceo de un árbol al cual se le ha eliminado un elemento de uno de los subárboles, y, por esta razón, el otro subárbol está generando un desbalanceo. Al entrar a la rutina, el parametro *menosAlto es TRUE. A continuación se presenta una de estas rutinas.

```
AVL balanIzq( AVL a, int *menosAlto )
{   switch( a->balan )
    { case IZQ:     if( a->izq->balan != DER )
                    {     a = roteDer( a );
                          if( a->balan == BAL )
                          {     a->balan = DER;
                                a->der->balan = IZQ;
                                *menosAlto = FALSE;
                          }
                          else
                                a->balan = a->der->balan = BAL;
                    }
                    else
                    {     a = roteIzqDer( a );
                          a->der->balan = ( a->balan == IZQ ) ? DER : BAL;
                          a->izq->balan = ( a->balan == DER ) ? IZQ : BAL;
                          a->balan = BAL;
                    }
                    break;
      case BAL:     a->balan = IZQ;
                    *menosAlto = FALSE;
                    break;
      case DER:     a->balan = BAL;
                    break;
    }
    return a;
}
```

Ejercicios Propuestos:

4.85. Muestre el proceso de creación de un árbol AVL para la siguiente secuencia de elementos: identifique el caso de desbalanceo al cual se llega después de cada inserción y aplique la solución correspondiente:

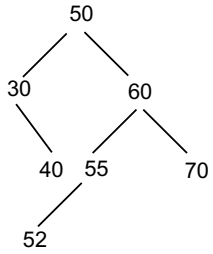
a-) 10 - 8 - 5 - 20 - 30 - 25

b-) 25 - 20 - 30 - 22 - 12 - 27 - 32 - 28 - 26 - 29

c-) 5 - 10 - 15 - 20 - 25 - 30 - 35 - 40 - 45

d-) 100 - 90 - 80 - 70 - 60 - 50 - 40 - 30 - 20 - 10

4.86. Muestre gráficamente el proceso que se sigue para eliminar los elementos del árbol AVL dado a continuación, en el siguiente orden: 60 - 40 - 50 - 70 - 55 - 52 - 30.



4.87. Desarrolle una rutina para determinar si un árbol binario ordenado es AVL.

4.88. Desarrolle una rutina para determinar si un árbol binario ordenado es perfectamente balanceado.

4.89. Desarrolle una rutina que dado un árbol AVL con el factor de desbalanceo no inicializado llene ese campo en todos los nodos. La solución debe ser tal, que no se calcule 2 veces la altura de ninguno de sus subárboles, porque eso lo haría muy ineficiente. ¿Cómo guardar las alturas que va calculando?

4.90. Desarrolle una función que, dada una lista ordenada de elementos, construya un árbol perfectamente balanceado con todos esos valores. Fíjese que la solución no es única.

4.10. El TAD Arbol de Sintaxis

Otra utilización común de los árboles binarios es la representación de relaciones de composición. Un ejemplo de esto son los árboles de sintaxis, cuyo propósito es representar expresiones matemáticas. Por ejemplo, si se tiene la expresión ((A+10) * 15) - (B / 10), ésta puede ser representada, sin necesidad de los paréntesis, con un árbol como el mostrado en la figura 4.12, donde la relación padre→hijo viene establecida por la relación operador→ operando, que es recursiva.

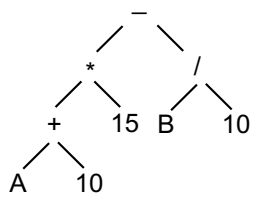


Fig. 4.12 - Arbol de sintaxis

4.10.1. Expresiones Aritméticas en Infijo

Toda expresión aritmética está constituida por operadores y operandos, y representa la manera de obtener un resultado, a partir del valor de sus componentes. Cada operador tiene asociados dos operandos, con los cuales debe trabajar para obtener un valor, que posiblemente será utilizado por otro operador para llegar a un resultado. Los operadores binarios considerados en esta parte son cuatro: suma (+), resta (-), multiplicación (*) y división (/). Los operandos, por su parte, pueden ser constantes o variables. Las constantes, para efectos prácticos, son secuencias de dígitos, mientras que las variables son cadenas de letras.

La estructura de una expresión en notación infija se puede resumir con la siguiente gramática (recursiva), que especifica que ésta puede estar formada por una variable, una constante, o dos expresiones entre paréntesis con un operador binario entre ellas. Los paréntesis son indispensables para evitar ambigüedades.

< expresión > ::= < variable > |
< constante > |
(< expresión > < operador > < expresión >)

El valor de una expresión depende del valor de sus componentes. Si todos los elementos son constantes, tiene un único valor, pero si hay variables, el valor depende del contenido de cada una de ellas. Los siguientes son ejemplos de expresiones aritméticas válidas en notación infija:

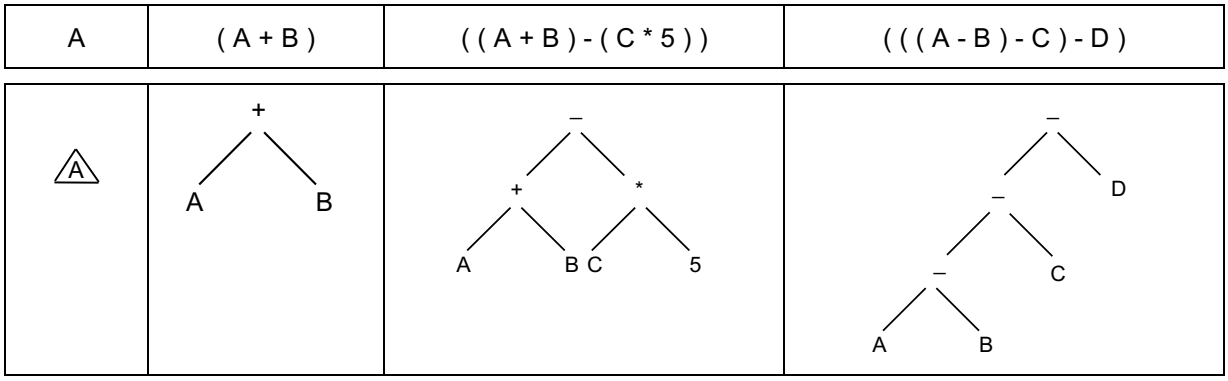
- A
- 46
- (A + B)
- ((A + B) - (C * 5))

4.10.2. Arboles de Sintaxis

Un **árbol de sintaxis** es un árbol binario completo, en el cual los nodos interiores son operadores y las hojas, operandos, y representa, sin ambigüedad, una expresión aritmética.

Ejemplo 4.35:

A continuación se presenta el árbol de sintaxis correspondiente a algunas expresiones aritméticas en notación infija:



4.10.3. La Tabla de Símbolos

Para manejar los valores asociados con las variables de una expresión se utiliza una estructura de datos denominada **tabla de símbolos**. Esta estructura está compuesta por parejas de la forma [variable, valor], en las cuales se mantiene -en todo momento- el último valor asignado a cada variable de un programa (su valor actual). Para tener un rápido acceso a la información, esta tabla se puede representar internamente como un árbol binario ordenado, de acuerdo con el nombre de la variable. Un ejemplo de una tabla de símbolos, estructurada como un árbol binario se muestra en la figura 4.13.

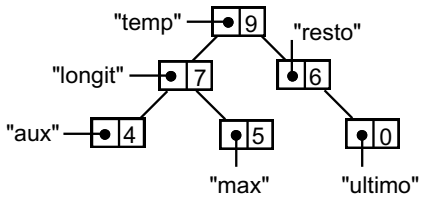


Fig. 4.13 - Tabla de símbolos

Para administrar una tabla de símbolos, el TAD TabSim cuenta con tres operaciones: una constructora (inicTabSim), una modificadora (asigneTabSim) y una analizadora (valorTabSim). Su implementación se propone como ejercicio al lector.

• inicTabSim:		→ TabSim
• asigneTabSim:	TabSim x char * x int	→ TabSim
• valorTabSim:	TabSim x char *	→ int

```
TabSim inicTabSim( void )
/* Crea una tabla de símbolos vacía */

{ post: inicTabsim = Δ }
```

```
TabSim asigneTabSim( TabSim TS, char * nom, int val )
/* Asigna el valor val a la variable de nombre nom */

{ pre: ts = TS }
{ post: ( [ nom, v ] ∈ TS, asigneTabsim = ( TS - [ nom, v ] ) + [ nom, val ] ) ∨
        ( [ nom, v ] ∉ TS, asigneTabsim = TS + [ nom, val ] ) }
```

```
int valorTabSim( TabSim TS, char * nom )
/* Retorna el valor asociado con la variable de nombre nom */

{ pre: [ nom, v ] ∈ TS }
{ post: valorTabSim = v }
```

4.10.4. El TAD Arsin

El TAD que maneja árboles de sintaxis tiene dos operaciones básicas. La primera operación (expresionArsin) construye un árbol a partir de una expresión aritmética en notación infija, que recibe como una lista de cadenas de caracteres. La segunda operación (evalArsin) hace la evaluación de un árbol de sintaxis, dada una tabla de símbolos de donde puede tomar los valores de las variables.

• expresionArsin:	Lista[char *]	→ Arsin
• evalArsin:	Arsin x TabSim	→ int

```
Arsin expresionArsin( Lista exp )
/* Crea un árbol de sintaxis a partir de una expresión en infijo representada en una lista de cadenas */

{ pre: exp es una expresión aritmética infija bien construida }
{ post: expresiónArsin es el árbol de sintaxis de exp }
```

```
int evalArsin( Arsin as, TabSim TS )
/* Retorna el resultado de evaluar el árbol de sintaxis con los valores de la tabla de símbolos */

{ pre: TS es una tabla de símbolos con todas las variables del árbol de sintaxis }
{ post: evalArsin es el resultado de evaluar la expresión }
```

Las estructuras de datos seleccionadas para representar el árbol de sintaxis se declaran como aparece a continuación. El campo tipo, de cada nodo, almacena un código para indicar la clase de elemento que contiene: 0 → suma (+), 1 → resta (-), 2 → multiplicación (*), 3 → división (/), 4 → constante (valor en el campo ival), 5 → variable (nombre de la variable en el campo pval).

```
typedef struct NodoArsin
{   int tipo;                       /* Clase de elemento que contiene */
    union
    {     int num;                  /* Valor numerico */
          char *nom;                /* Nombre de la variable */
    } val;
    struct NodoArsin *izq,*der;     /* Subárboles de sintaxis que representan los operandos */
} TArsin,*Arsin;
```

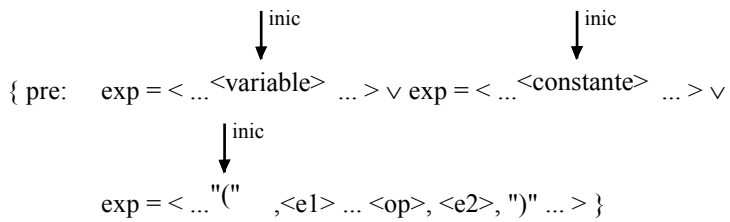
La función que implementa la operación de evaluación de un árbol de sintaxis corresponde a una rutina recursiva, que exige la evaluación de los operandos antes de aplicar el operador.

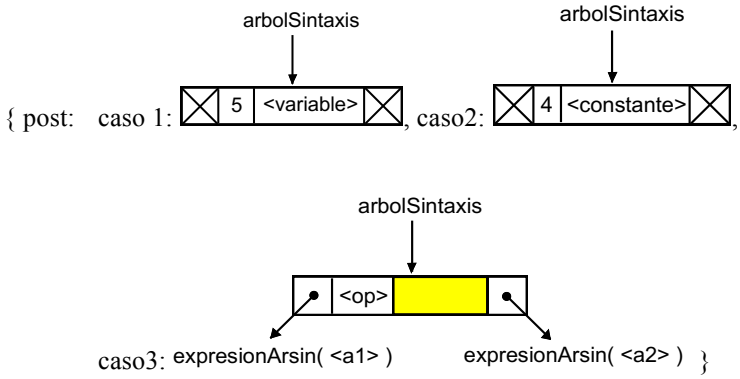
```
int evalArsin( Arsin as, Tabsim ts )
{   switch( as->tipo )
    {     case 0:   return evalArsin( as->izq, ts ) + evalArsin( as->der, ts );
          case 1:   return evalArsin( as->izq, ts ) - evalArsin( as->der, ts );
          case 2:   return evalArsin( as->izq, ts ) * evalArsin( as->der, ts );
          case 3:   return evalArsin( as->izq, ts ) / evalArsin( as->der, ts );
          case 4:   return as->val.num;
          case 5:   return valorTabsim( ts, as->val.nom );
    }
    return 0;
}
```

La implementación de la constructora requiere una rutina de apoyo (posOperador), que retorna la posición del operador principal, en la lista que representa la expresión. Con este valor, es posible hacer las llamadas recursivas correspondientes, para que construya los subárboles de sintaxis asociados. Se utiliza otra rutina (arbolSintaxis), con un parámetro adicional, para llevar la posición dentro de la lista donde comienza la expresión que se está considerando. En la llamada inicial dicho valor debe ser 1.

```
Arsin expresionArsin( Lista exp )
{   return arbolSintaxis( exp, 1 );
}
```

La especificación recursiva de la rutina arbolSintaxis es la siguiente:





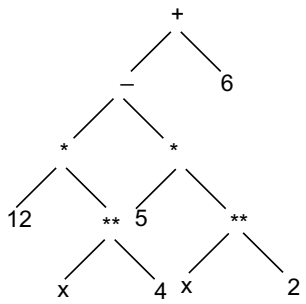

```
static Arsin arbolSintaxis( Lista exp, int inic )
{   int pos;
    Arsin as;
    posLista( exp, inic );
    if( infoLista( exp )[ 0 ] != '(' )
    {     /* casos 1 ó 2 */
          as = ( Arsin )malloc( sizeof( TArsin ) );
          as->izq = as->der = NULL;
          if( isdigit( infoLista( exp )[ 0 ] ) )                    /* Caso 2 */
          {     as->tipo = 4;
                as->val.num = atoi( infoLista( exp ) );
          }
          else
          {     as->tipo = 5;
                as->val.nom = (char *)malloc( strlen( infoLista( exp ) ) + 1 );   /* Caso 1 */
                strcpy( as->val.nom, infoLista( exp ) );
          }
    }
    else
    {     /* Caso 3 */
          posLista( exp, pos = posOperador( exp, inic ) );
          as = ( Arsin )malloc( sizeof( TArsin ) );
          switch( infoLista( exp )[ 0 ] )
          { case '+': as->tipo = 0;
                      break;
            case '-': as->tipo = 1;
                      break;
            case '*': as->tipo = 2;
                      break;
            case '/': as->tipo = 3;
                      break;
          }
                                                  inic  inic+1  pos  pos+1
          as->izq = arbolSintaxis( exp, inic+1 );   /* exp = < ..."("  , <e1>  ... <op>  , <e2>  , ")" ... > */
          as->der = arbolSintaxis( exp, pos+1 );
    }
    return as;
}
```

Para localizar el operador principal se va contando la diferencia entre el número de paréntesis abiertos y cerrados, y, sólo cuando esta diferencia sea 1, se puede tomar el operador principal. La precondición de la rutina asegura que en la posición inic de la lista aparece un paréntesis abierto.

```
int posOperador( Lista exp, int inic )
{   int numParentesis;
    posLista( exp, ++inic );
    if( infoLista( exp )[ 0 ] == '(' )
    {     for( numParentesis = 2, sigLista( exp ), inic++; numParentesis != 1; sigLista( exp ), inic++ )
              if( infoLista( exp )[ 0 ] == '(' )
                    numParentesis++;
              else if( infoLista( exp )[ 0 ] == ')' )
                    numParentesis--;
          return inic;
    }
    else
          return inic + 1;
}
```

Ejercicios Propuestos:

4.91. Un polinomio puede ser representado internamente como un árbol de sintaxis. Por ejemplo, el polinomio P(x) = $12x^4 - 5x^2 + 6$, puede tener la siguiente representación, donde el operador ** corresponde a la exponenciación:



Implemente el TAD Polinomio utilizando el esquema de representación sugerido.

4.92. Desarrolle una función para determinar si una expresión aritmética en notación infija se encuentra bien construida.

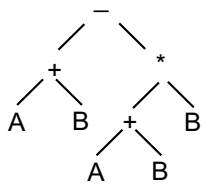
4.93. Escriba una rutina para crear un árbol de sintaxis de una expresión aritmética en notación postfija.

4.94. Desarrolle un algoritmo para simplificar una expresión representada mediante su árbol de sintaxis. Por simplificar se entiende calcular los valores constantes de la expresión (por ejemplo, remplazar (4 + 5) por 9), eliminar los términos para los cuales ya se conoce respuesta (por ejemplo, remplazar ((A + B) * 0) por 0, o ((A + B) + 0) por (A + B)), realizar simplificaciones aritméticas simbólicas (por ejemplo, remplazar ((A + B) - (A + B - C)) por C), etc.

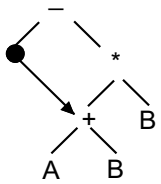
4.95. Escriba un programa que reciba por pantalla asignaciones de la forma x = < expresión >, evalúe la expresión y deje el resultado en la tabla de símbolos. También debe aceptar el comando print(x), que imprime por pantalla el valor de la variable x.

4.96. 🕒 Cuando se representa una expresión aritmética mediante su árbol de sintaxis, muchas veces se desperdicia espacio en la memoria en el momento de su implementación, puesto que si hay subexpresiones comunes, se utiliza espacio para representar cada una de sus ocurrencias. En ese

caso, se podría pensar en compartir la memoria que las representa, con el fin de ahorrar espacio. Por ejemplo, si se tiene la expresión ((A + B) - ((A + B) * B)), el árbol de sintaxis correspondiente es:



Pero, si comparte las subexpresiones comunes, se tendría algo del estilo:



Desarrolle una función que modifique la estructura de la representación interna de un árbol de sintaxis, utilizando la idea enunciada anteriormente.

Bibliografía

- [AHO83] Aho, A., Hopcroft, J., Ullman, J., "Data Structures and Algorithms", Cap. 3, 4, 5, Addison-Wesley, 1983.
- [BAE77] Baer, J., "A Comparison of Tree-Balancing Algorithms", Comm. ACM, Vol. 20, No. 5, 1977.
- [BER94] Bergin, J., "Data Abstraction: The Object-Oriented Approach Using C++", Cap. 9, 10, 12, McGraw-Hill, 1994.
- [BRU72] Bruno, J., Coffman, E., "Nearly Optimal Binary Search Trees", Proc. IFIP, North-Holland, 1972.
- [CHA84] Chang, H., Iyengar, S., "Efficient Algorithms to Globally Balance a Binary Search Tree", Comm. ACM, Vol. 27, 1984.
- [CUN87] Cunto, W., Gascon, J., "Improving Time and Space Efficiency in Generalized Binary Search Trees", Acta Informatica, Vol. 24, No 5, 1987.
- [DAL86] Dale, N., Lilly, S., "Pascal y Estructura de Datos", Cap. 8, 9, McGraw-Hill, 1986.
- [ER86] Er, M., "Efficient Generation of Binary Trees from Inorden-Postorden Sequences", Information Sciences, Vol. 40 , No. 2, 1986.
- [ESA89] Esakov, J., Weiss, T., "Data Structures: An Advanced Approack Using C", Cap. 7, Prentice-Hall, 1989.
- [FEL88] Feldman, M., "Data Structures with Modula-2", Cap. 7, Prentice-Hall, 1988.
- [FOS73] Foster, C., "A Generalization of AVL Trees", Comm. ACM, Vol. 16, No. 8, 1973.
- [GER88] Gerasch, T., "An Insertion Algorithm for Minimal Internal Path Length Binary Search Tree", Comm. ACM, Vol. 31, No. 15, 1988.
- [HOR83] Horowitz, E., "Fundamentals of Data Structures", Cap. 5, Computer Science Press, 1983.
- [KNU73] Knuth, D., "The Art of Computer Programming. Vol. 1 - Fundamental Algorithms", 2da edición, Addison-Wesley, 1973.
- [KRU87] Kruse, R., "Data Structures & Program Design", Cap.9, Prentice-Hall, 1987.

- [LIP87] Lipschutz, S., "Estructura de Datos", Cap. 7, McGraw-Hill, 1987.
- [MAR86] Martin, J., "Data Types and Data Structures", Cap. 7, Prentice-Hall, 1986.
- [TEN93] Tenenbaum, A., Langsam, Y., "Estructuras de Datos en C", Cap. 5, Prentice Hall, 1993.
- [TRE76] Tremblay, J., Sorenson, P., "An Introduction to Data Structures with Applications", Cap. 5, McGraw-Hill, 1976.
- [WIR76] Wirth, N., "Algorithms + Data Structures = Programs", Cap. 4, Prentice-Hall,1976.
- [WIR86] Wirth, N., "Algorithms & Data Structures", Cap. 4, Prentice-Hall, 1986.

CAPITULO 5

ESTRUCTURAS RECURSIVAS:

ARBOLES N-ARIOS

En este capítulo se estudian las estructuras recursivas de datos denominadas árboles n-arios, que corresponden a una generalización del concepto de árbol binario. La diferencia radica en que esta nueva estructura puede manejar múltiples subárboles asociados con cada elemento, y no solamente 2, como en el caso de los árboles binarios. Este tipo de estructura se utiliza para modelar jerarquías con una relación 1 a N entre un grupo de elementos. En particular, se estudian los árboles 2-3, *cuadtrees*, *tries*, árboles *AND-OR* y árboles de juego.

5.1. Motivación

Considere el caso de una fábrica, en la cual se quiere hacer el modelaje de un automóvil en términos de sus componentes: cada pieza debe estar relacionada con todos los elementos que la constituyen, como se sugiere en la figura 5.1. Fíjese que es insuficiente un árbol binario para manejar este tipo de estructuras, ya que el número de hijos no se puede restringir a dos.

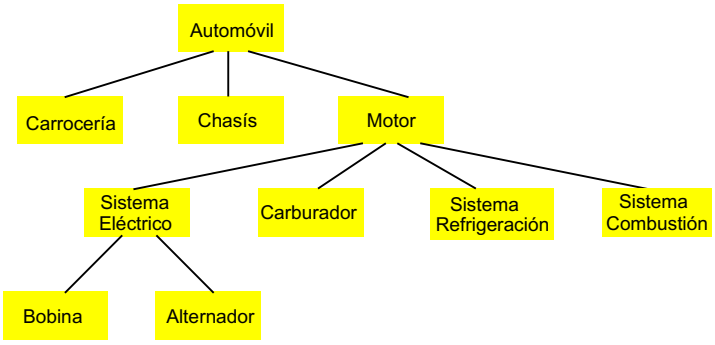


Fig. 5.1 Modelaje de la relación pieza - componente

Otro ejemplo interesante de uso de un árbol n-ario puede ser un árbol de juego. Cada elemento de la estructura corresponde a un estado posible del tablero y la relación padre → hijo modela la situación $estado_i \rightarrow estado_{i+1}$ de la partida, como se ilustra en la figura 5.2. para el caso del juego de triqui. Allí se tiene que la raíz del árbol es el tablero vacío (sin ninguna jugada), y que sus hijos son todas las posibles configuraciones a las cuales se puede llegar mediante una jugada de las *X*. (9 en total). Estos, a su vez, van a tener 8 hijos cada uno, que corresponden a los tableros que se obtienen con todas las respuestas posibles del jugador

adversario (las *O*). Las hojas de dicho árbol son los tableros en los cuales el juego ha terminado, ya sea por el triunfo de alguno de los participantes, o por un empate ante la ausencia de nuevas jugadas posibles.

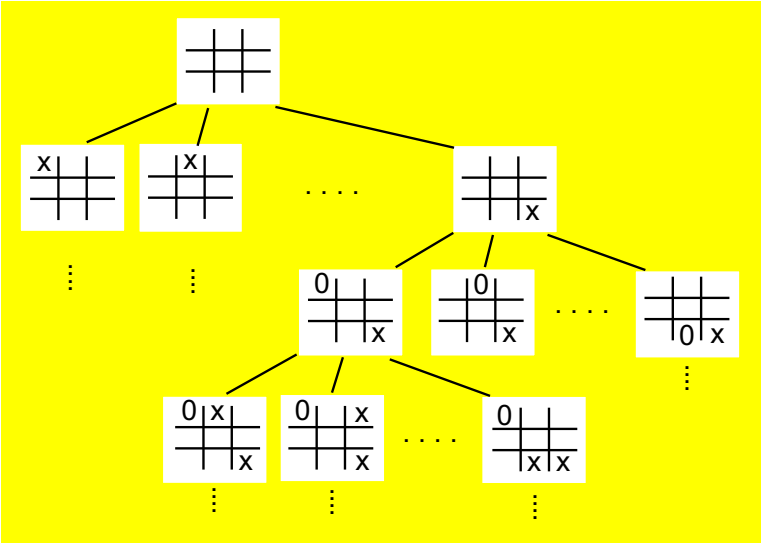


Fig. 5.2 - Árbol de juego para el triqui

5.2. Definiciones y Conceptos Básicos

Un **árbol n-ario** es una estructura recursiva, en la cual cada elemento tiene un número cualquiera de árboles n-arios asociados. En la figura 5.3 se muestra el formalismo gráfico escogido para representar este objeto abstracto. En él se hace explícita la raíz, y cada uno de los n subárboles que tiene asociados. Un árbol vacío se representa con el símbolo Δ.

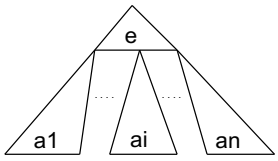
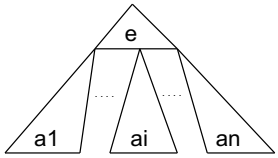


Fig. 5.3 - Formalismo para árboles n-arios

En general, los conceptos estudiados en el capítulo anterior, relacionados con árboles binarios, se pueden extender sin ningún problema a árboles n-arios. Tal es el caso de las definiciones de padre, hijo, hermano, nivel, altura, peso, camino, etc. Sólo vale la pena hacer algunas precisiones en el caso de los recorridos, sobre todo en lo que respecta al inorden, puesto que no es claro en qué punto del proceso se debe visitar la raíz.

Por definición, si un árbol n-ario a tiene la siguiente estructura:



Los **recorridos principales** son:

- inorden(a) = inorden(a_1), e, inorden(a_2), ..., inorden(a_n)

- preorden(a) = e, preorden(a_1), ..., preorden(a_n)
- postorden(a) = postorden(a_1), ..., postorden(a_n), e

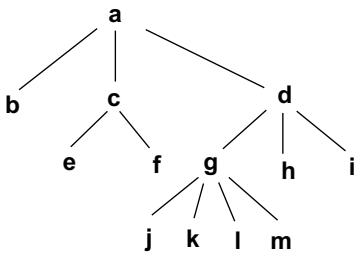
Fuera de los conceptos extendidos a partir de los árboles binarios, se tienen dos nuevas definiciones para los árboles n-arios:

- El **orden de un elemento** de un árbol n-ario es el número de subárboles que éste tiene asociados. En particular, una hoja es un elemento de orden 0.
- El **orden de un árbol** n-ario es el máximo orden de sus elementos. Eso hace que un árbol binario sea un árbol n-ario de orden 2.

Algunos conceptos de árboles binarios no se pueden extender directamente a árboles n-arios, dado que el número máximo de subárboles asociados con cada elemento es indeterminado. Tal es el caso de las definiciones de árbol binario lleno, casi lleno y completo, en las cuales se debe restringir la noción respectiva a un orden dado en el árbol n-ario. Por ejemplo, se puede hablar de un árbol 4-ario lleno, pero no de un árbol n-ario lleno.

Ejemplo 5.1:

Para el árbol n-ario de la figura:



- el orden del árbol es 4
- el orden del elemento a es 3
- preorden = a, b, c, e, f, d, g, j, k, l, m, h, i
- inorden = b, a, e, c, f, j, g, k, l, m, d, h, i
- postorden = b, e, f, c, j, k, l, m, g, h, i, d, a
- niveles = a, b, c, d, e, f, g, h, i, j, k, l, m
- altura = 4
- peso = 13
- los hijos de g son los elementos j, k, l, m
- el ancestro común más próximo de k y h es d

⏎

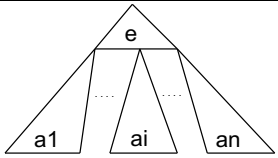
5.3. El TAD ArbolN: Analizadoras

Siguiendo la misma idea que se utilizó para el TAD Arbin, el TAD ArbolN sólo incluye un conjunto de analizadoras comunes a todas las clases posibles de árboles n-arios. Las modificadoras vienen asociadas

con el tipo exacto de árbol sobre el cual se esté trabajando (árboles ordenados, árboles de juego, organigramas, árboles de composición, árboles de sintaxis, etc.).

La diferencia básica en las operaciones de los TAD Arbin y ArbolN es que en lugar de dos operaciones analizadoras para retornar el subárbol izquierdo y el subárbol derecho (utilizadas en el TAD Arbin), se va a definir una única operación que retorna la lista de todos los subárboles asociados con un elemento. Por esta razón, los algoritmos de manejo de árboles n-arios deben utilizar las operaciones del TAD Lista para desplazarse sobre los hijos de cada uno de sus componentes.

TAD ArbolN[TipoAN]



{ inv: a_1,...., a_n son disyuntos }

Constructoras:

• inicArbolN:		→ ArbolN

Analizadoras:

• subArbolN:	ArbolN	→ Lista[ArbolN]
• raizArbolN:	ArbolN	→ TipoAN
• vacioArbolN:	ArbolN	→ int

```
ArbolN inicArbolN( void )
/* Crea un árbol n-ario vacío */

{ post: inicArbolN = Δ }
```

```
Lista[ ArbolN ] subArbolN( ArbolN a )
/* Retorna la lista de subárboles asociados */

{ pre: a no es vacío }
{ post: subArbolN = ⟨ a1, ...,an ⟩, ai es el i-ésimo subárbol de a }
```

```
TipoAN raizArbolN( ArbolN a )
/* Retorna la raíz del árbol */

{ pre: a no es vacío }
{ post: raizArbolN = e }
```

```
int vacioArbolN( ArbolN a )
/* Informa si el árbol es vacío */

{ post: vacioArbolN = ( a = Δ ) }
```

5.4. Ejemplos de Utilización

Los algoritmos para manejar árboles n-arios tienen una estructura similar a la utilizada para resolver el mismo problema sobre un árbol binario. La diferencia radica en que, en lugar de tener sólo dos avances viables en la recursión (subárbol izquierdo y subárbol derecho), va a tener n avances posibles, lo cual obliga a utilizar un ciclo para iterar sobre cada uno de ellos.

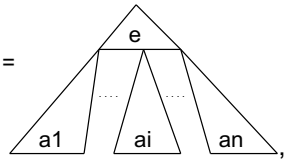
Los casos típicos de búsqueda, recorrido y verificación de alguna característica de un árbol n-ario se ilustran a continuación mediante ejemplos.

Ejemplo 5.2

Establecer si un elemento dado se encuentra presente en un árbol n-ario. Puesto que no existe ninguna relación de orden entre los elementos de la estructura, se debe buscar recursivamente en cada uno de los subárboles presentes. Esto hace que la complejidad de la operación sea O(n), donde n es el peso del árbol.

```
/* post: estaArbolN = elem está presente en el árbol a */

int estaArbolN( ArbolN a, TipoAN elem )
{   Lista lst;
    if( vacioArbolN( a ) )
        return FALSE;
    else if( raizArbolN( a ) == elem )
        return TRUE;
    else
    {   lst = subArbolN( a );
        for( primLista( lst ); !finLista( lst ); elimLista( lst ) )
            if( estaArbolN( infoLista( lst ), elem ) )
            {   destruirLista( lst );
                return TRUE;
            }
        return FALSE;
    }
}
```

Las dos salidas de la recursión corresponden a los mismos casos triviales considerados en el algoritmo equivalente en árboles binarios: el árbol es vacío o el elemento es igual a la raíz. La diferencia radica en la manera como avanza la recursión, puesto que debe intentar bajar por cada uno de los subárboles asociados, buscando el elemento pedido. Tan pronto localiza el elemento en uno de los subárboles, termina el proceso. El invariante de este ciclo de avance de la recursión es el siguiente:

a = 

{ inv: , elem != e, lst = < a_1, ..., [ai], ..., a_n >, elem no está en a_1, ..., a_{i-1} }

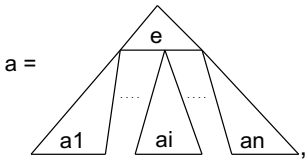
Allí se afirma que cuando la ventana de la lista esté sobre el i-ésimo subárbol, el elemento buscado no será la raíz, ni habrá aparecido en los árboles anteriores.

Ejemplo 5.3:

Calcular la altura de un árbol n-ario dado. Se debe hacer una llamada recursiva para calcular la altura de cada uno de los subárboles asociados, y retornar -finalmente- la mayor de ellas más 1. La complejidad del algoritmo es O(n), donde n es el peso del árbol.

```
/* post: alturaArbolN = altura del árbol a */

int alturaArbolN( ArbolN a )
{   Lista lst;
    int mayor, temp;
    if( vacioArbolN( a ) )
        return 0;
    else
    {   lst = subArbolN( a );
        for( mayor = 0, primLista( lst ); !finLista( lst ); elimLista( lst ) )
            if( ( temp = alturaArbolN( infoLista( lst ) ) ) > mayor )
                mayor = temp;
        return mayor + 1;
    }
}
```

La salida de la recursión corresponde al caso en el cual el árbol es vacío. Para avanzar se debe hacer una llamada recursiva sobre cada uno de los subárboles asociados y acumular la altura del más alto de ellos. El invariante de este ciclo es:



{ inv: , lst = < a_1, ..., ai, ..., a_n >, mayor = mayor altura de a_1, ..., a_{i-1} }

Allí se asegura que cuando la ventana de la lista esté sobre el i-ésimo subárbol, en la variable mayor se encontrará acumulada la mayor altura de los primeros i-1 subárboles.

⏎

Ejemplo 5.4:

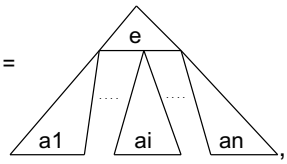
Recorrer en inorden un árbol n-ario dado. La estructura del algoritmo recursivo surge de la definición misma del recorrido. Su complejidad es O(n), donde n es el peso del árbol.

La rutina supone la existencia del procedimiento visitar, que procesa (imprime, en el caso más simple) cada uno de los elementos del árbol.

```
/* post: se ha recorrido en inorden el árbol a */
```

```
void inordenArbolN( ArbolN a )
{   Lista lst;
    if( !vacioArbolN( a ) )
    {     lst = subArbolN( a );
          if( longLista( lst ) == 0 )
                visitar( raizArbolN( a ) );
          else
          {     primLista( lst );
                inordenArbolN( infoLista( lst ) );
                visitar( raizArbolN( a ) );
                for( sigLista( lst ); !finLista( lst ); elimLista( lst ) )
                      inordenArbolN( infoLista( lst ) );
          }
    }
}
```

El invariante del ciclo de avance de la recursión afirma que cuando la ventana de la lista lst se encuentre sobre el i-ésimo subárbol, ya se habrán recorrido en inorden los árboles a_1, ... , a_{i-1}, y también la raíz, en caso de que i sea mayor que 1.

a = 

{ inv: , lst = < a_1, ..., ai, ..., a_n >, se ha recorrido a_1, e, a2, ..., a_{i-1} }

Ejemplo 5.5:

Recorrer por niveles un árbol n-ario dado. Al igual que en el recorrido por niveles de un árbol binario, se necesita una cola como estructura auxiliar, para ir incluyendo en ella los subárboles de izquierda a derecha. La complejidad de la rutina es O(n), donde n es el peso del árbol, si las operaciones del TAD Cola son O(1). Supone la existencia de la rutina visitar, encargada de procesar cada elemento del árbol.

```
void nivelesArbolN( ArbolN a )
{   Lista lst;
    Cola col;
    ArbolN a1;
    if( !vacioArbolN( a ) )
    {     col = inicCola( );
          adicCola( col, a );
          while( !vaciaCola( col ) )
          {     a1 = infoCola( col );
                elimCola( col );
                visitar( raizArbolN( a ) );
                lst = subArbolN( a1 );
                for( primLista( lst ); !finLista( lst ); elimLista( lst) )
                      adicCola( col, infoLista( lst ) );
          }
    }
}
```

El invariante del ciclo garantiza que cuando se hayan visitado por niveles los primeros k elementos del árbol, en la cola se van a encontrar todos los subárboles asociados con dichos elementos (los cuales no han sido recorridos todavía), ordenados por niveles:

{ inv: se han recorrido por niveles los elementos e_1, ..., e_k,

col = $\overline{a1... am}\leftarrow$, todos los elementos no visitados se encuentran en a_1, ..., a_m,

los árboles a_1, ..., a_m son disyuntos,

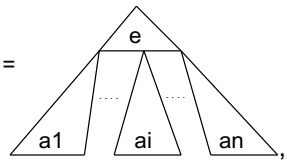
los siguientes elementos del recorrido por niveles son raizArbolN(a_1), ..., raizArbolN(a_m) }

⏎

Ejemplo 5.6:

Contar el número de hojas de un árbol n-ario dado. La complejidad del algoritmo es O(n), donde n es el peso del árbol. La estructura del algoritmo es similar a la utilizada para calcular la altura.

```
int hojasArbolN( ArbolN a )
{   Lista lst;
    int acum;
    if( vacioArbolN( a ) )
        return 0;
    else if( longLista( lst = subArbolN( a ) ) == 0 )
        return 1;
    else
    {   for( acum = 0, primLista( lst ); !finLista( lst ); elimLista( lst ) )
            acum += hojasArbolN( infoLista( lst ) );
        return acum;
    }
}
```

a = 

{ inv: , lst = < a_1, ..., ai, ..., a_n >, acum = número de hojas de a_1, ... , a_{i-1} }

⏎

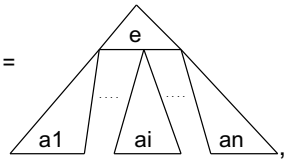
Ejemplo 5.7:

Calcular el nivel de un elemento en un árbol n-ario sin valores repetidos. La complejidad de la rutina es O(n), donde n es el peso del árbol. En este ejemplo se ilustra el uso de la técnica de acumulación de parámetros, para el caso de árboles n-arios:

```
/* pre: a no tiene elementos repetidos */
/* post: nivelArbolN = nivel del elemento elem en el árbol a ∨ nivelArbolN = -1 si el elemento no está presente */

int nivelArbolN( ArbolN a, TipoAN elem )
{   return nivel2ArbolN( a, elem, 0 );
}
```

```
/* pre: a no tiene elementos repetidos, nivel es el nivel del árbol a en el problema global */
/* post: ( nivel2ArbolN = -1, elem no está en el árbol a ) ∨ ( nivel2ArbolN = nivel de elem en el árbol a ) */

int nivel2ArbolN( ArbolN a, TipoAN elem, int nivel )
{   Lista lst;
    int temp;
    if( vacioArbolN( a ) )
        return -1;
    else if( raizArbolN( a ) == elem )
        return nivel;
    else
    {   lst = subArbolN( a );
        for( primLista( lst ); !finLista( lst ); elimLista( lst ) )
            if( ( temp = nivel2ArbolN( infoLista( lst ), elem, nivel + 1 ) ) != -1 )
            {   destruirLista( lst );
                return temp;
            }
        return -1;
    }
}
```

La primera rutina abre el espacio para la acumulación del parámetro nivel, y lo inicializa en 0. La segunda rutina baja por cada subárbol, indicando cada vez que avanza en la recursión que se encuentra un nivel más abajo en el árbol global. Si no lo encuentra, retorna -1. Si lo encuentra, informa el nivel que ha acumulado durante el descenso.

a = 

{ inv: , lst = < a_1, ..., ai, ..., a_n >, el nivel del elemento e es nivel, e != elem, elem no está en los subárboles a_1, ... , a_{i-1} }

⏎

Ejercicios Propuestos

5.1. int igualesN(ArbolN a1, ArbolN a2)
/* Indica si dos árboles n-arios son iguales */

5.2. int isomorfosN(ArbolN a1, ArbolN a2)
/* Informa si los árboles n-arios a1 y a2 son isomorfos */

5.3. int estableN(ArbolN a)
/* Un árbol de valores enteros es **estable** si para todo elemento su padre es mayor. Esta función indica si un árbol n-ario es estable */

5.4. Lista primosN(ArbolN a, TipoAN elem)
/* Retorna una lista con los primos del elemento elem dentro del árbol n-ario a. Por **primos** se refiere a todos los hijos de los hermanos del padre */

5.5. Lista busqueCaminoN(ArbolN a, TipoAN elem)
/* Retorna una lista de elementos del árbol n-ario a correspondiente al camino que lleva desde la raíz del árbol hasta el elemento elem. Si el camino no existe retorna una lista vacía. Utiliza la técnica de acumulación de parámetros */

5.6. int ocurreN(ArbolN a1, ArbolN a2)
/* Indica si el árbol n-ario a2 ocurre en el árbol n-ario a1 */

5.7. TipoAN ancestroListaN(ArbolN a, Lista lst)
/* Retorna el ancestro común más próximo de los elementos presentes en la lista lst, los cuales están presentes en el árbol a */

5.8. Lista ramaN(ArbolN a)
/* Retorna la rama más larga del árbol n-ario a */

5.9. int esMenorN(ArbolN a1, ArbolN a2)
/* Indica si el árbol n-ario a1 es menor que el árbol n-ario a2. Un árbol a1 es **menor** que otro a2, si todos los elementos de a1 son menores que todos los elementos de a2 */

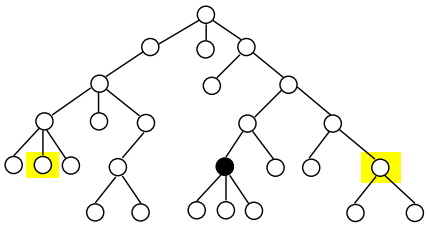
5.10. 🕒 💾 void postordenArbolN(ArbolN a)
/* Hace un recorrido iterativo en postorden de un árbol n-ario */

5.11. Lista rutaMinimaN(ArbolN a, TipoAN e1, TipoAN e2)
/* Se define la **ruta mínima** entre dos elementos cualesquiera e1 y e2 de un árbol n-ario sin elementos repetidos, como la secuencia de elementos < x_1, x_2, ..., x_n > que cumple las siguientes condiciones:

- x_1 = e1, x_n = e2
- x_i es padre de x_{i+1}, o x_{i+1} es padre de x_i
- no existen elementos repetidos en la secuencia

Dicha ruta existe entre todo par de elementos de un árbol n-ario, y es única. Esta función retorna una lista de elementos con la ruta mínima entre dos elementos dados */

5.12. 🕒 Lista mVecinosN(ArbolN a, TipoAN elem, int m)
/* Retorna los vecinos del elemento elem que se encuentran a una distancia menor que m de él. Por **vecino** se entiende un elemento del mismo nivel y por **distancia** el número de elementos que los separa. Por ejemplo, en el árbol de la figura aparecen los 3-vecinos del elemento marcado:



Suponga que no hay elementos repetidos en el árbol */

5.13. Lista listaNivelN(ArbolN a, int n)
/* Retorna una lista con todos los elementos del nivel n del árbol n-ario a */

5.14. int ramaMinimaN(ArbolN a)
/* Suponiendo que los elementos del árbol n-ario a son enteros, se define el **costo de un camino** como la suma de los componentes de dicha secuencia. Esta función retorna el costo de la rama más barata del árbol */

5.15. 🕒 int llenoN(ArbolN a, int ord)
/* Informa si el árbol n-ario a es lleno para un orden ord */

5.16. 🕒 int casiLlenoN(ArbolN a, int ord)
/* Informa si el árbol n-ario a es casi lleno para un orden ord */

5.17. int completoN(ArbolN a, int ord)
/* Informa si el árbol n-ario a es completo para un orden ord */

5.18. Lista descendientesN(ArbolN a, TipoAN elem)
/* Retorna una lista con todos los descendientes del elemento elem en el árbol a */

5.19. int piramideN(ArbolN a)
/* Un árbol n-ario es una **pirámide** si todo elemento cumple que es igual a la suma de sus hijos. Esta función informa si el árbol n-ario a es una pirámide */

5.20. int ordenN(ArbolN a)
/* Retorna el orden del árbol n-ario a */

5.21. int gorduraN(ArbolN a)
/* Se define la **anchura** de un nivel de un árbol n-ario como el número de elementos presentes en dicho nivel. La **gordura** de un árbol n-ario corresponde al valor máximo de las anchuras de sus niveles. Esta rutina calcula la gordura del árbol n-ario a */

5.22. int ordenadoArbolN(ArbolN a)
/* Se define un árbol n-ario **ordenado** como aquél en el cual su recorrido en inorden es una secuencia ordenada ascendentemente. Esta función indica si un árbol n-ario es ordenado */

5.23. int masProfundo(ArbolN a, TipoAN elem)
/* Retorna el nivel de la ocurrencia más profunda del elemento elem en el árbol a, suponiendo que en el árbol puede haber elementos repetidos. Si el elemento elem no aparece en el árbol retorna -1 */

5.24. TipoAN masOcurre(ArbolN a)
/* Retorna el elemento que más veces aparece en el árbol n-ario a, suponiendo que no es vacío */

5.25. Considere la siguiente gramática, la cual define expresiones aritméticas con cualquier número de operandos:

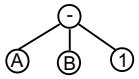
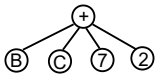
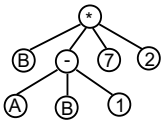
<expresión> ::= <constante> |
<variable> |
<operador> (<expresión> <expresión> ... <expresión>)

<constante> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<variable> ::= A | B | C | ... | X | Y | Z |

<operador> ::= + | - | * | /

La forma de interpretar una expresión, se ilustra con los siguientes ejemplos:

expresión	árbol n-ario de sintaxis	interpretación
-(A B 1)		(A - B) - 1
+(B C 7 2)		B + C + 7 + 2
*(B -(A B 1) 7 2)		B * ((A - B) - 1) * 7 * 2

a-) Desarrolle una operación que verifique que una expresión que viene dada como una lista de caracteres corresponde a la gramática. Por ejemplo, la primera expresión del ejemplo anterior vendría expresada mediante la siguiente lista:

< '-', '(', 'A', 'B', '1', ')' >

b-) Desarrolle una operación que construya un árbol n-ario de sintaxis a partir de una lista de caracteres con una expresión válida.

c-) Suponiendo que existe una tabla de símbolos, en la cual aparece un valor asociado con cada variable, desarrolle una función que evalúe un árbol n-ario que representa una expresión válida.

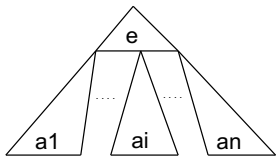
5.5. Implementación del TAD ArbolN

De todas las estructuras de datos posibles para representar internamente un árbol n-ario, se muestran en esta parte cinco de las más difundidas. El problema de la representación se limita a la forma de simular un número variable de subárboles: si dicho valor es fijo y conocido (v.g. un árbol 4-ario), se pueden utilizar muchas de las representaciones planteadas en el capítulo anterior, con mínimos ajustes.

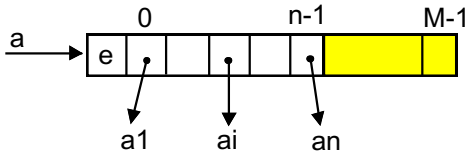
5.5.1. Vector de Apuntadores

Esta representación corresponde a una generalización de la implementación de apuntadores para árboles binarios. Puesto que se desconoce *a priori* el número de subárboles que va a tener cada elemento de la estructura, se debe reservar para cada uno de ellos un vector de apuntadores de un tamaño dado M y almacenar en cada posición la dirección de uno de sus subárboles. Si el orden del elemento es menor que el espacio reservado, se debe colocar el valor NULL en las casillas sobrantes. El esquema de representación se puede resumir en los siguientes casos:

- El árbol n-ario vacío a = Δ se representa con un apuntador a NULL:

- El árbol n-ario a = 

se representa con un apuntador a un nodo, que contiene la raíz y un vector de apuntadores a sus subárboles, como se muestra en la siguiente figura:



- El árbol n-ario a = 

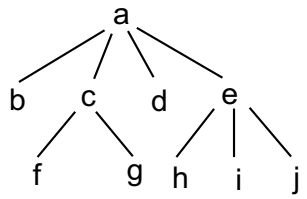
se representa con la siguiente estructura:

El valor M es una constante de la implementación. Las casillas del vector entre las posiciones n y M-1 apuntan a NULL, para indicar que no tiene más subárboles asociados. Esta implementación tiene la

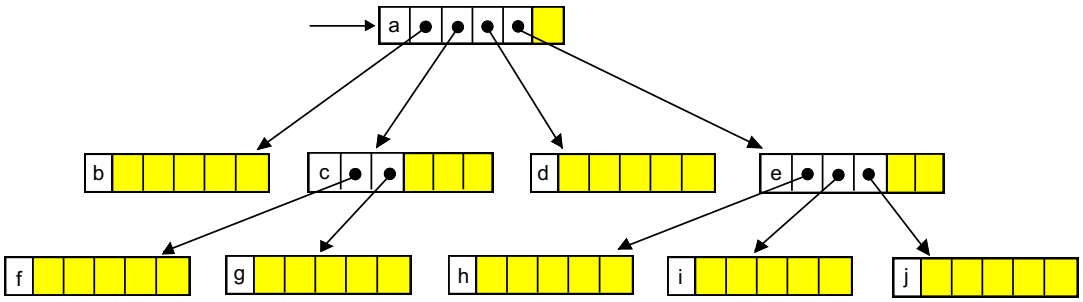
restricción de manejar un número máximo fijo de subárboles, con las consiguientes consecuencias de falta de flexibilidad y espacio en memoria desaprovechado. Se usa, sobre todo, cuando el orden de los elementos del árbol es fijo o aproximadamente conocido. Tiene la ventaja de que los algoritmos que implementan las operaciones son sencillos.

Ejemplo 5.8:

Para el árbol de la figura:



Las estructuras de datos para M = 5 son:



Para esta representación, la declaración de las estructuras de datos en C es:

```
#define M 5

typedef struct NodoArbolN
{   TipoAN info;                        /* Raíz del árbol n-ario */
    struct NodoArbolN *hijos[ M ];          /* Vector de apuntadores a los subárboles */
} TArbolN, *ArbolN;
```

Las rutinas que implementan las operaciones del TAD son:

- Para crear un árbol n-ario vacío basta con retornar el valor NULL, de acuerdo con el primer caso del esquema de representación:

```
ArbolN inicArbolN( void )
{   return NULL;
}
```

- La operación que retorna una lista con los subárboles debe recorrer el vector de apuntadores e irlos agregando a la lista de respuesta. Termina cuando ha recorrido todo el vector o cuando encuentra un subárbol vacío:

```
Lista subArbolN( ArbolN a )
{   int i;
    Lista lst = inicLista( );
    for( i = 0; i < M && a->hijos[ i ] != NULL; i++ )
        anxLista( lst, a->hijos[ i ] );
    return lst;
}
```

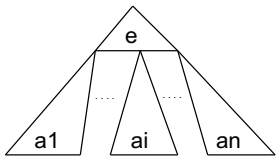
- Las otras dos operaciones del TAD tienen el siguiente código:

```
TipoAN raizArbolN( ArbolN a )
{   return a->info;
}

int vacioArbolN( ArbolN a )
{   return a == NULL;
}
```

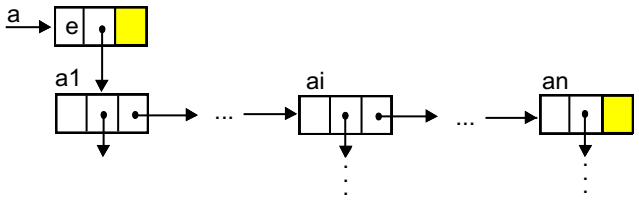
5.5.2. Hijo Izquierdo - Hermano Derecho

En esta implementación, cada elemento del árbol guarda únicamente información de su hijo izquierdo y de su hermano derecho. De esta forma, es posible tener acceso a toda la estructura. El esquema de representación se resume en los siguientes puntos:

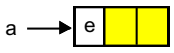
- El árbol n-ario vacío a = Δ se representa con un apuntador a NULL:

- El árbol n-ario a = 

se representa con un apuntador a un nodo que contiene la raíz y apuntadores a su hijo izquierdo y a su hermano derecho, como se muestra en la siguiente figura:

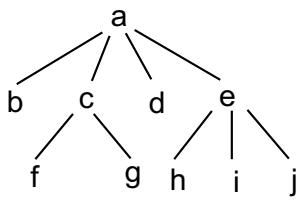


- El árbol n-ario a = - se representa con la siguiente estructura:

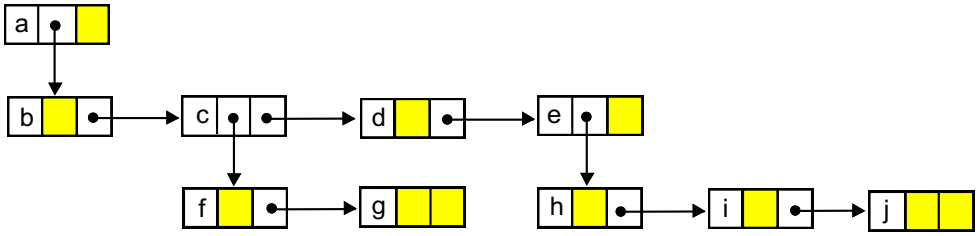


Ejemplo 5.9:

Para el árbol de la figura:



Las estructuras de datos que lo representan bajo el esquema hijo izquierdo - hermano derecho son:



Este esquema de representación es muy flexible, puesto que no restringe el número de subárboles que puede tener asociados un elemento. Las estructuras de datos se declaran así en C:

```
typedef struct NodoArbolN
{   TipoAN info;                    /* Raíz del árbol */
    struct NodoArbolN *hijo;        /* Encadenamiento al hijo izquierdo */
    struct NodoArbolN *hermano;     /* Encadenamiento al hermano derecho */
} TArbolN, *ArbolN;
```

- La rutina que crea un árbol n-ario vacío debe retornar un apuntador a NULL, de acuerdo con el esquema de representación planteado:

```
ArbolN inicArbolN( void )
{   return NULL;
}
```

- La operación que retorna una lista con los subárboles asociados debe bajar por el apuntador del hijo izquierdo, y hacer un recorrido siguiendo el encadenamiento del hermano derecho. Cada elemento encontrado se va adicionando a la lista de respuesta:

```
Lista subArbolN( ArbolN a )
{   ArbolN pa;
    Lista lst = inicLista( );
    for( pa = a ->hijo; pa != NULL; pa = pa->hermano )
        anxLista( lst, pa );
    return lst;
}
```

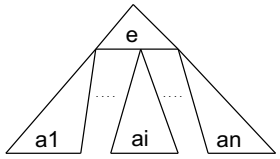
- Las otras dos operaciones del TAD tienen una implementación sencilla, como se presenta a continuación:

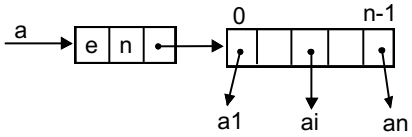
```
TipoAN raizArbolN( ArbolN a )
{   return a->info;
}

int vacioArbolN( ArbolN a )
{   return a == NULL;
}
```

5.5.3. Vectores Dinámicos

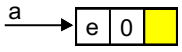
Las principales desventajas de la implementación de vectores de apuntadores (§5.5.1) son su rigidez y la gran cantidad de espacio que se desaprovecha. Si el árbol no es muy dinámico, es decir, si no está sometido a constantes alteraciones, es posible asociar con cada elemento del árbol un vector de longitud variable, con el espacio indispensable para mantener los apuntadores a sus hijos. De esta forma, utilizando la función realloc (de ANSI C), se puede ir reacomodando el espacio necesario en memoria. Sólo se requiere un campo adicional en cada nodo para almacenar su orden, como se muestra en el siguiente esquema de representación:

- El árbol n-ario vacío a = Δ se representa con un apuntador a NULL:

- El árbol n-ario a =  se representa con un apuntador a un nodo, que contiene la raíz y un vector de apuntadores a sus subárboles (de tamaño igual al número de subárboles asociados), como se muestra en la siguiente figura:

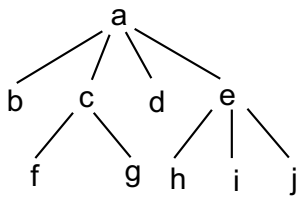


- El árbol n-ario a =  se representa con la siguiente estructura:

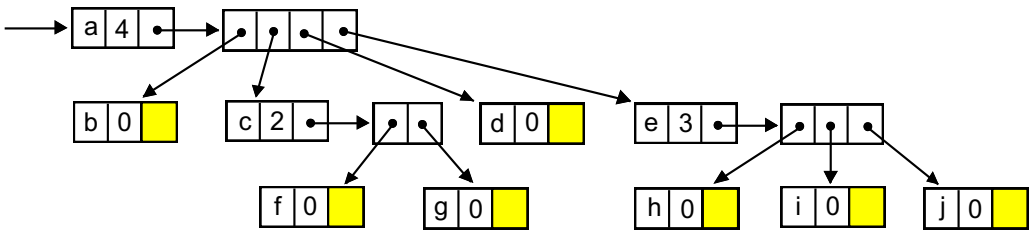


Ejemplo 5.10:

Para el árbol de la figura:



Las estructuras de datos que lo representan son:



Las estructuras de datos se declaran así:

```
typedef struct NodoArbolN
{   TipoAN info;                    /* Raíz del árbol n-ario */
    int orden;                      /* Orden del elemento */
    struct NodoArbolN **hijos;      /* Vector dinámico de apuntadores a los subárboles */
} TArbolN, *ArbolN;
```

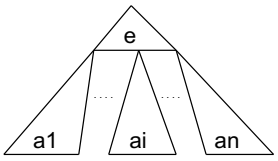
Las rutinas que implementan las operaciones del TAD ArbolN son muy similares a las de vectores de apuntadores:

```
ArbolN inicArbolN( void )
{   return NULL;
}

Lista subArbolN( ArbolN a )
{   int i;
    Lista lst = inicLista( );
    for( i = 0; i < a->orden; i++ )
        anxLista( lst, a->hijos[ i ] );
    return lst;
}

TipoAN raizArbolN( ArbolN a )
{   return a->info;
}

int vacioArbolN( ArbolN a )
{   return a == NULL;
}
```

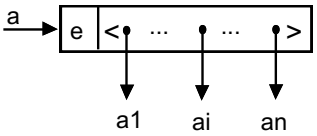
5.5.4. Lista de Hijos

Una implementación muy sencilla para árboles n-arios consiste en asociar con cada elemento del árbol la lista de sus subárboles, utilizando para ello un objeto abstracto del TAD Lista[ArbolN]. El esquema de representación es el siguiente:

- El árbol n-ario vacío a = Δ se representa con un apuntador a NULL:

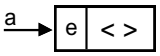
- El árbol n-ario a = 

se representa con un apuntador a un nodo, que contiene la raíz y un objeto del TAD Lista con los apuntadores a sus subárboles, como se muestra en la siguiente figura:



- El árbol n-ario a = 

se representa con la siguiente estructura:



Esta representación garantiza una mayor independencia entre la implementación del árbol y la forma de almacenar y manejar los subárboles asociados con cada elemento.

La declaración de las estructuras de datos para esta representación es la siguiente:

```
typedef struct NodoArbolN *TipoL;          /* Tipo de los elementos de la lista */

typedef struct NodoArbolN
{   TipoAN info;                           /* Raíz del árbol n-ario */
    Lista hijos;                           /* Lista de apuntadores a los subárboles */
} TArbolN, *ArbolN;
```

La implementación de las operaciones del TAD se hace con las rutinas que se dan a continuación:

```
ArbolN inicArbolN( void )
{   return NULL;
}

Lista subArbolN( ArbolN a )
{   Lista lst = inicLista( );
    for( primLista( a->hijos); !finLista( a->hijos ); sigLista( a->hijos ) )
        anxLista( lst, infoLista( a->hijos ) );
    return lst;
}

TipoAN raizArbolN( ArbolN a )
{   return a->info;
}

int vacioArbolN( ArbolN a )
{   return a == NULL;
}
```

5.5.5. Representaciones Implícitas

No todas las representaciones de un árbol deben ser explícitas, en el sentido de que hay casos en los cuales los hijos pueden ser calculados a partir de la información del padre, sin necesidad de tenerlos almacenados en estructuras de datos. Este es el caso, por ejemplo, del juego de triqui, en el cual se puede implementar la operación subArbolN partiendo únicamente del estado del tablero de juego en un momento dado (ver ejemplo 5.10).

Estas representaciones implícitas dependen del problema exacto que se quiere resolver, y no es posible hacer una generalización. Eso implica que, en cada caso, se debe hacer un estudio y un diseño nuevo para el esquema de representación, teniendo en cuenta las relaciones implícitas entre los elementos de la estructura jerárquica. En el siguiente ejemplo se ilustra el proceso para el caso del juego de triqui.

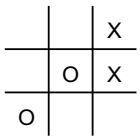
Ejemplo 5.11:

Suponga que se quiere representar el árbol de juego para el triqui, el cual tiene la estructura mostrada en la figura 5.2. La primera opción es utilizar cualquiera de los esquemas de representación estudiados en las secciones anteriores y situar explícitamente en las estructuras de datos las relaciones jerárquicas entre los tableros.

Otra opción es diseñar una representación implícita, que aproveche las reglas del juego para que dado un tablero, sea capaz de calcular sus hijos sin necesidad de tenerlos previamente en las estructuras de datos del árbol. Esto es, en lugar de crear una estructura con encadenamientos entre padre e hijos, es posible pensar en representar el árbol con una matriz de 3 x 3, que mantenga únicamente el estado del tablero en un momento dado. Los subárboles asociados van implícitos con el tablero, puesto que las reglas del juego permiten calcularlos sin necesidad de tenerlos todo el tiempo en memoria. Considere las siguientes declaraciones de estructuras de datos:

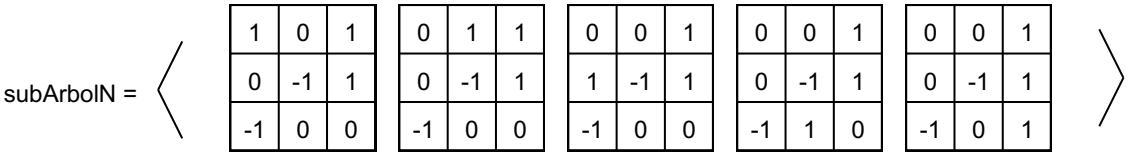
```
#define X 1
#define O -1
#define VACIO 0

typedef struct
{    int tablero[ 3 ][ 3 ];          /* Tablero de juego del triqui */
} TArbolN, *ArbolN;
```

Allí se representa un árbol n-ario con una matriz de enteros de 3 x 3, en donde una casilla vacía se representa con un 0, una *X* con un 1 y una *O* con un -1, como se muestra en el siguiente dibujo:



0	0	1
0	-1	1
-1	0	0

Para este árbol (una matriz de 3 x 3), la operación subArbolN es capaz de retornar la siguiente lista de árboles n-arios, sin necesidad de mantener ninguna otra estructura de datos auxiliar, sólo suponiendo que juegan las *X*:



La rutina que crea el árbol de juego completo tiene el siguiente código:

```
ArbolN crearArbolN( void )
{   int i, k;
    ArbolN a = (ArbolN ) malloc (sizeof( TArbolN ) );
    for( i = 0; i < 3; i++ )
        for( k = 0; k < 3; k++ )
            a->tablero[ i ][ k ] = VACIO;
    return a;
}
```

La operación que retorna la lista de los subárboles asociados debe identificar todos los estados posibles del juego al colocar una nueva *X*, aplicando las reglas del triqui, y crear los tableros respectivos:

```
Lista subArbolN( ArbolN a )
{   int i, k;
    ArbolN aux;
    Lista lst = inicLista( );
    for( i = 0; i < 3; i++ )
        for( k = 0; k < 3; k++ )
            if( a->tablero[ i ][ k ] == VACIO )
            {   aux = crearArbolN( );
                *aux = *a;
                aux->tablero[ i ][ k ] = X;
                anxLista( lst, aux );
            }
    return lst;
}
```

En los ejercicios propuestos se sigue trabajando sobre esta misma idea, lo mismo que en la sección de árboles de juego.

5.6. El TAD ArbolN: Algunas Modificadoras y Destructoras

En general, las operaciones modificadoras de un tipo abstracto para el manejo de árboles dependen de la clase específica de árbol que se quiera administrar, tal como se hizo en el capítulo anterior para árboles binarios. En esta parte se presentan e implementan una constructora, dos modificadoras básicas y una destructora, como una manera de ilustrar la forma de aumentar la funcionalidad del TAD dependiendo de la semántica específica del árbol. Estas nuevas operaciones van a permitir crear una hoja, asociar y desasociar un subárbol de un elemento y destruir un árbol.

Constructora:

• hojaArbolN:	TipoAN	→ ArbolN

Modificadoras:

• adicSubArbolN:	ArbolN X ArbolN	→ ArbolN
• elimSubArbolN:	ArbolN X int	→ ArbolN

Destructora:

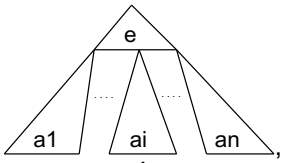
• destruirArbolN:	ArbolN

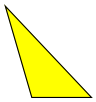
ArbolN hojaArbolN(TipoAN e)
/* Crea una hoja con raíz igual al elemento e */

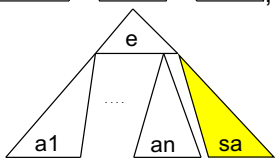
{ post: hojaArbolN = 

}

void adicSubArbolN(ArbolN a, ArbolN sa)
/* Agrega el árbol sa como último subárbol de a */

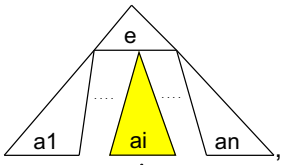
{ pre: a = 

, sa = 
}

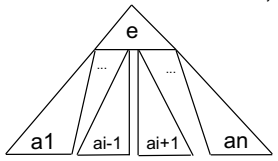
{ post: a = 

}

void elimSubArbolN(ArbolN a, int i)
/* Desasocia (pero no destruye) el i-ésimo subárbol de a */

{ pre: a = 

, 1 ≤ i ≤ n }

{ post: a = 

}

```
void destruirArbolN( ArbolN a )
/* Destruye un árbol n-ario, retornando toda la memoria ocupada en su representación */

{ post: se ha devuelto toda la memoria ocupada en la representación del árbol a. a es indefinido }
```

5.6.1. Implementación sobre Vector de Apuntadores

- Constructora: crea una hoja con raíz e. La rutina pide memoria, almacena la raíz e inicializa todas las casillas del vector de apuntadores en NULL. La complejidad es O(M), donde M es el tamaño del vector de apuntadores. En este caso M es constante pero mayor o igual que el máximo orden posible del árbol.

```
ArbolN hojaArbolN( TipoAN e )
{   int i;
    ArbolN a = ( ArbolN )malloc( sizeof( TArbolN ) );
    a->info = e;
    for( i = 0; i < M; i++ )
        a->hijos[ i ] = NULL;
    return a;
}
```

- Modificadora: agrega un subárbol. La rutina coloca el nuevo subárbol en la siguiente posición libre del vector de apuntadores. Puesto que el manejo de error es responsabilidad del cliente, no verifica la validez de la nueva posición. La complejidad es O(M), donde M es el tamaño del vector de apuntadores.

```
void adicSubArbolN( ArbolN a, ArbolN sa )
{   int i;
    for( i = 0; a->hijos[ i ] != NULL; i++ );
    a->hijos[ i ] = sa;
}
```

- Modificadora: elimina el i-ésimo subárbol. La rutina desplaza los subárboles (i+1... n) una posición a la izquierda en el vector de apuntadores, para ocupar la posición liberada por el i-ésimo subárbol. La complejidad es O(n), donde n es el orden de la raíz del árbol a.

```
void elimSubArbolN( ArbolN a, int i )
{   int k;
    for( k = i-1; k < M-1 && a->hijos[ k ] != NULL; k++ )
        a->hijos[ k ] = a->hijos[ k + 1 ];
    a->hijos[ k ] = NULL;
}
```

- Destructora. La rutina se llama recursivamente para cada uno de los subárboles asociados, y, finalmente, retorna la memoria ocupada por la raíz. La complejidad es O(n), donde n es el peso del árbol.

```
void destruirArbolN( ArbolN a )
{   int i;
    if( a != NULL )
    {     for( i = 0; i < M && a->hijos[ i ] != NULL; i++ )
                destruirArbolN( a->hijos[ i ] );
          free( a );
    }
}
```

5.6.2. Implementación sobre Apuntadores

- Constructora: crea una hoja con raíz e. La rutina pide memoria, almacena la raíz e inicializa los campos de hijo y hermano en NULL. La complejidad es O(1).

```
ArbolN hojaArbolN( TipoAN e )
{   ArbolN a = ( ArbolN )malloc( sizeof( TArbolN ) );
    a->info = e;
    a->hijo = a->hermano = NULL;
    return a;
}
```

- Modificadora: agrega un subárbol. La rutina recorre la lista de hijos utilizando el encadenamiento del hermano derecho, localiza el último de ellos y agrega allí el nuevo nodo. La complejidad es O(n), donde n es el orden de la raíz del árbol a.

```
void adicSubArbolN( ArbolN a, ArbolN sa )
{   ArbolN pa;
    if( a->hijo == NULL )
          a->hijo = sa;
    else
    {     for( pa = a->hijo; pa->hermano != NULL; pa = pa->hermano );
          pa->hermano = sa;
    }
}
```

- Modificadora: elimina el i-ésimo subárbol. La rutina localiza el i-ésimo subárbol utilizando el encadenamiento de los hermanos y lo saca de la secuencia. La complejidad es O(n), donde n es el orden de la raíz del árbol a.

```
void elimSubArbolN( ArbolN a, int i )
{   ArbolN pa, qa;
    if( i == 1 )
    {     qa = a->hijo;
          a->hijo = a->hijo->hermano;
    }
    else
    {     for( pa = a->hijo; i > 2; i--, pa = pa->hermano );
          qa = pa->hermano;
          pa->hermano = pa->hermano->hermano;
    }
    free( qa );
}
```

- Destructora. La rutina se llama recursivamente para cada uno de los subárboles asociados, y, finalmente, retorna la memoria ocupada por la raíz. La complejidad es O(n), donde n es el peso del árbol.

```
void destruirArbolN( ArbolN a )
{   ArbolN pa;
    if( a != NULL )
    {     for( pa = a ->hijo; pa != NULL; pa = pa->hermano )
                destruirArbolN( pa );
          free( a );
    }
}
```

5.6.3. Implementación sobre Vectores Dinámicos

- Constructora: crea una hoja con raíz e. La rutina pide memoria, almacena la raíz e inicializa el vector dinámico en vacío, informando que el orden del elemento es 0. La complejidad es O(1).

```
ArbolN hojaArbolN( TipoAN e )
{   ArbolN a = ( ArbolN )malloc( sizeof( TArbolN ) );
    a->info = e;
    a->orden = 0;
    a->hijos = NULL;
    return a;
}
```

- Modificadora: agrega un subárbol. La rutina aumenta el tamaño del vector dinámico, actualiza el orden del nodo y almacena en esta nueva posición el subárbol. La complejidad es O(n), donde n es el orden de la raíz del árbol a, puesto que la operación realloc debe copiar, en el peor de los casos, cada uno de los subárboles a una nueva zona de memoria.

```
void adicSubArbolN( ArbolN a, ArbolN sa )
{   a->hijos = ( struct NodoArbolN ** ) realloc ( a->hijos, ( ++( a->orden ) ) * sizeof( struct NodoArbolN * ) );
    a->hijos[ a->orden - 1 ] = sa;
}
```

- Modificadora: elimina el i-ésimo subárbol. La rutina compacta el vector para ocupar la casilla liberada por el i-ésimo subárbol y luego ajusta el espacio ocupado por el vector dinámico, disminuyendo el número de posiciones en 1. La complejidad es O(n), donde n es el orden de la raíz del árbol a

```
void elimSubArbolN( ArbolN a, int i )
{   int k;
    for( k = i-1; k < a->orden - 1; k++ )
        a->hijos[ k ] = a->hijos[ k + 1 ];
    a->hijos = (struct NodoArbolN **) realloc ( a->hijos, ( --( a->orden ) ) * sizeof( struct NodoArbolN * ) );
}
```

- Destructora. La rutina se llama recursivamente para cada uno de los subárboles asociados, y, finalmente, retorna la memoria ocupada por el vector dinámico y por la raíz. La complejidad es O(n), donde n es el peso del árbol.

```
void destruirArbolN( ArbolN a )
{   int i;
    if( a != NULL )
    {   for( i = 0; i < a->orden; i++ )
            destruirArbolN( a->hijos[ i ] );
        free( a->hijos );
        free( a );
    }
}
```

5.6.4. Implementación sobre Lista de Hijos

- Constructora: crea una hoja con raíz e. La rutina pide memoria, almacena la raíz y crea una lista vacía de subárboles. La complejidad de la rutina viene dada por la complejidad de la operación inicLista.

```
ArbolN hojaArbolN( TipoAN e )
{   ArbolN a = ( ArbolN )malloc( sizeof( TArbolN ) );
    a->info = e;
    a->hijos = inicLista( );
    return a;
}
```

- Modificadora: agrega un subárbol. La rutina sitúa la ventana de la lista de hijos sobre la última posición y anexa en ese punto el nuevo subárbol. La complejidad de la rutina depende de la complejidad de las operaciones ultLista y anxLista.

```
void adicSubArbolN( ArbolN a, ArbolN sa )
{   ultLista( a->hijos );
    anxLista( a->hijos, sa );
}
```

- Modificadora: elimina el i-ésimo subárbol. La rutina sitúa la ventana de la lista sobre el i-ésimo subárbol y procede a eliminarlo valiéndose de la respectiva operación del TAD Lista. La complejidad de este procedimiento depende de la complejidad de las operaciones posLista y elimLista.

```
void elimSubArbolN( ArbolN a, int i )
{   posLista( a->hijos, i );
    elimLista( a->hijos );
}
```

- Destructora. La rutina se llama recursivamente para cada uno de los subárboles asociados, y, finalmente, retorna la memoria ocupada por la lista de subárboles y por la raíz. La complejidad es O(n), donde n es el peso del árbol.

```
void destruirArbolN( ArbolN a )
{   if( a!= NULL )
    {     for( primLista( a->hijos ); !finLista( a->hijos ); sigLista( a->hijos ) )
                destruirArbolN( infoLista( a->hijos ) );
          destruirLista( a->hijos );
          free( a );
    }
}
```

Ejercicios Propuestos

Para cada una de las 4 representaciones vistas anteriormente, desarrolle las siguientes rutinas, de manera que trabajen directamente sobre las estructuras de datos. Calcule la complejidad de su solución:

5.26. void reemplazarArbolN(ArbolN a, TipoAN elem1, TipoAN elem2)
/* Reemplaza en el árbol a todas las ocurrencias del elemento elem1 por el elemento elem2 */

5.27. int nivelArbolN(ArbolN a, TipoAN elem)
/* Retorna el nivel en el que aparece el elemento elem en el árbol n-ario a. Supone que el árbol no tiene elementos repetidos. Si el elemento no aparece retorna -1 */

5.28. ArbolN podar1ArbolN(ArbolN a)
/* Elimina del árbol a todas sus hojas */

5.29. int alturaArbolN(ArbolN a)
/* Calcula la altura del árbol n-ario a */

5.30. ArbolN podar2ArbolN(ArbolN a, int niv)
/* Elimina del árbol a todos los elementos que se encuentran en un nivel superior o igual a niv */

5.31. int hojasArbolN(ArbolN a)
/* Cuenta el número de hojas del árbol n-ario a */

5.32. 🕒 ArbolN eliminarArbolN(ArbolN a, TipoAN elem)
/* Elimina del árbol n-ario a el elemento elem, de tal manera que el árbol resultante conserve el mismo inorden, salvo por el elemento retirado */

5.33. int estaArbolN(ArbolN a, TipoAN elem)
/* Informa si el elemento elem se encuentra presente en el árbol n-ario a */

5.34. void inordenArbolN(ArbolN a)
/* Hace un recorrido iterativo en inorden, utilizando como estructura auxiliar una pila */

5.35. Lista busqueCaminoN(ArbolN a, TipoAN elem)
/* Retorna una lista de elementos del árbol n-ario a correspondiente al camino que lleva desde la raíz del árbol hasta el elemento elem. Si el camino no existe retorna una lista vacía */

5.36. void insSubArbolN(ArbolN a, ArbolN sa, int i)
/* Agrega el árbol sa como i-ésimo subárbol de a. Supone que a tiene por lo menos i subárboles asociados */

5.37. Diseñe un esquema de persistencia para árboles n-arios. Implemente una operación que lea un árbol n-ario de un archivo (cargarArbolN) y otra que lo haga persistir (salvarArbolN). Pruebe sus rutinas con cada una de las representaciones planteadas en la sección anterior.

5.38. Diseñe un esquema de representación secuencial para árboles n-arios. Implemente todas las operaciones del TAD ArbolN (constructoras, analizadoras modificadoras y destructora) sobre la representación propuesta. Calcule la complejidad de cada rutina y haga una comparación con las representaciones estudiadas en la sección anterior.

5.39. Para la representación implícita sugerida en el ejemplo 5.11 (árbol de juego de triqui), desarrolle una rutina que determine en cuantas jugadas se encuentra la primera posición ganadora del juego, suponiendo que en ese momento tienen el turno las *X*.

5.40. Para la representación implícita sugerida en el ejemplo 5.11 (árbol de juego de triqui), desarrolle una rutina que determine cuál de los dos jugadores tiene mayor opción de triunfo. Esto es, quién tiene un mayor número de posiciones ganadoras como descendientes de la situación actual.

5.7. El TAD Arbol1-2-3: Un Árbol Triario Ordenado

Un **árbol 1-2-3** es un árbol n-ario ordenado de orden 3, que en cada nodo tiene 1 ó 2 elementos y 1, 2 ó 3 subárboles asociados. En la figura 5.4 aparece un ejemplo de uno de esos árboles.

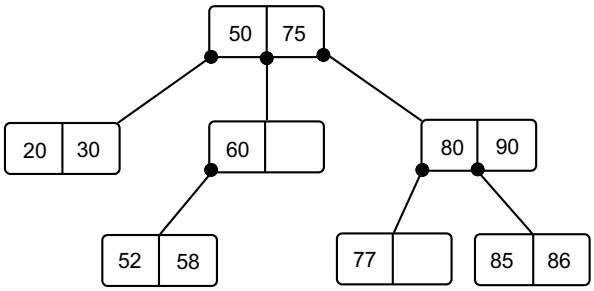


Fig. 5.4 - Árbol 1-2-3

Los árboles 1-2-3 satisfacen las siguientes condiciones:

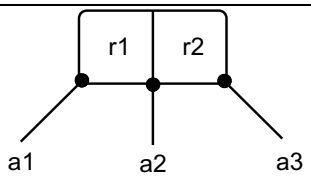
- No hay elementos repetidos en la estructura.
- El elemento de la izquierda de cada nodo, llamado **raíz izquierda**, es menor que el elemento de su derecha (**raíz derecha**).
- El primer subárbol es un árbol 1-2-3 que contiene elementos menores que la raíz izquierda.
- El segundo subárbol es un árbol 1-2-3 que contiene los elementos mayores que la raíz izquierda pero menores que la raíz derecha.
- El tercer subárbol es un árbol 1-2-3 que contiene los elementos mayores que la raíz derecha.
- La raíz derecha puede estar vacía, caso en el cual los subárboles segundo y tercero son vacíos.

Esta estructura se utiliza para almacenar conjuntos de elementos, para los cuales tiene sentido una relación de orden, permitiendo, por la manera como reparte los elementos, un acceso eficiente a la información. Los

árboles 1-2-3 se introducen en este libro, como una herramienta pedagógica para facilitar la presentación de los árboles 2-3, en la siguiente sección.

La especificación del TAD es la siguiente:

TAD Arbol123[Tipo123]



{ inv: a1, a2 y a3 son árboles 1-2-3,

((r1 < r2, (e ∈ a1 ⇒ e < r1), (e ∈ a2 ⇒ r1 < e < r2), (e ∈ a3 ⇒ e > r2)) ∨

(r2 = VACIO, (e ∈ a1 ⇒ e < r1), a2 = a3 = Δ)) }

Constructoras:

• inicArbol123:		→ Arbol123

Modificadoras:

• insArbol123:	Arbol123 x Tipo123	→ Arbol123
• elimArbol123:	Arbol123 x Tipo123	→ Arbol123

Analizadoras:

• estaArbol123:	Arbol123 x Tipo123	→ int

```
Arbol123 inicArbol123( void )
/* Crea un árbol 1-2-3 vacío */

{ post: inicArbol123 = Δ }
```

```
Arbol123 insArbol123( Arbol123 a, Tipo123 elem )
/* Agrega el elemento elem al árbol 1-2-3 */

{ pre: elem ∉ a, a = A }
{ post: insArbol123 = A + elem }
```

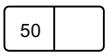
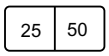
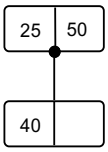
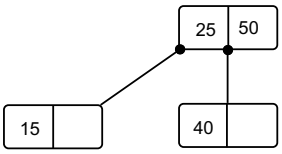
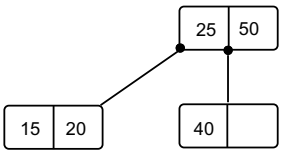
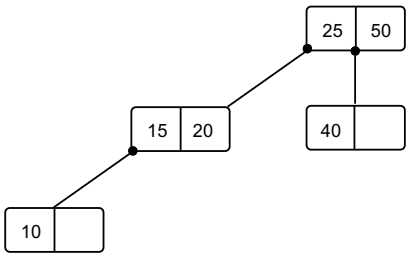
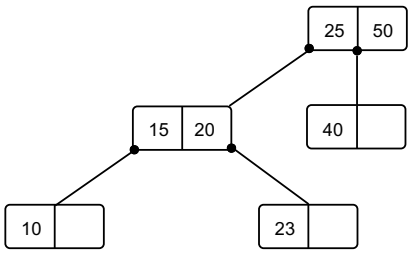
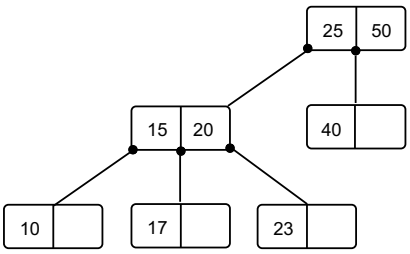
```
Arbol123 elimArbol123( Arbol123 a, Tipo123 elem )
/* Elimina el elemento elem del árbol 1-2-3 */

{ pre: elem ∈ a, a = A }
{ post: elimArbol123 = A - elem }
```

```
int estaArbol123( Arbol123 a, Tipo123 elem )
/* Informa si el elemento elem está en el árbol 1-2-3 */

{ post: estaArbol123 = ( elem ∈ a ) }
```

Ejemplo 5.12:

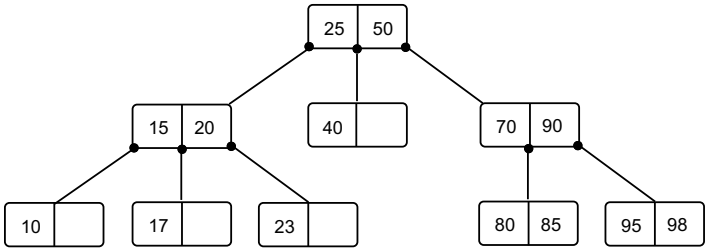
Este ejemplo ilustra el proceso de inserción de un elemento dado en un árbol 1-2-3, mostrando los estados intermedios alcanzados por el árbol cuando se inserta la siguiente secuencia de elementos:

50 - 25 - 40 - 15 - 20 - 10 - 23 - 17

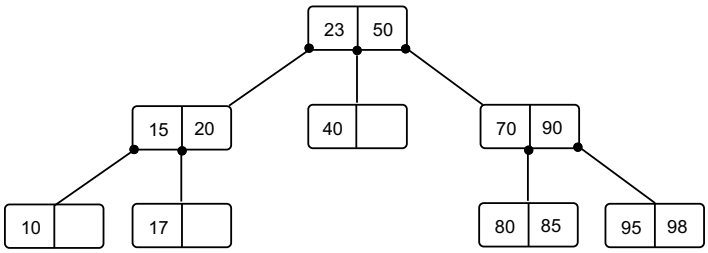
Paso No. 1: insertar 50	Paso No. 2: insertar 25
	
Paso No. 3: insertar 40	**Paso No. 4: insertar 15**
	
Paso No. 5: insertar 20	**Paso No. 6: insertar 10**
	
Paso No. 7: insertar 23	**Paso No. 8: insertar 17**
	

Ejemplo 5.13:

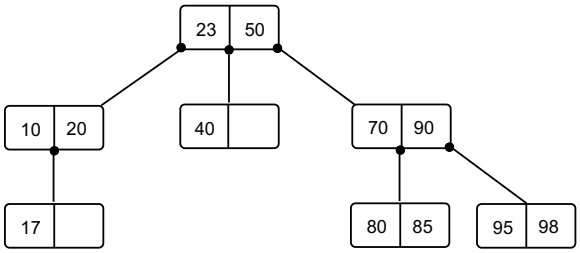
Para eliminar un elemento de un árbol 1-2-3, se sigue un proceso equivalente al utilizado en árboles binarios, viendo un árbol 1-2-3 como dos árboles binarios paralelos, y eliminando el elemento del lado respectivo. Por ejemplo, para el árbol siguiente:



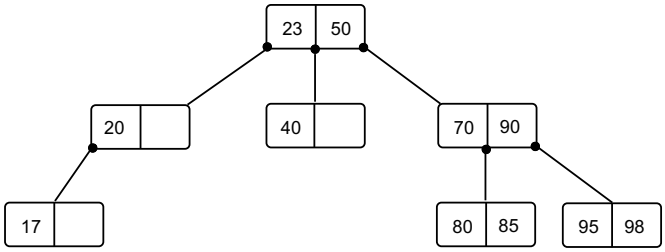
Si se quiere eliminar el elemento 25, siguiendo la técnica de remplazo por el mayor elemento del subárbol izquierdo (vista en el capítulo anterior), se obtiene el árbol:



Para este nuevo árbol, si se elimina el elemento 15, se llega a la siguiente estructura:



Repitiendo el mismo proceso para el elemento 10, se llega al siguiente árbol 1-2-3, en el cual se debe mover el valor 20 de la raíz derecha a la raíz izquierda, para seguir manteniendo las propiedades de este tipo de árboles:



La representación más sencilla para los árboles 1-2-3 utiliza la siguiente declaración de estructuras de datos:

```
typedef struct NodoArbol123
{   int raiz1, raiz2;                          /* Elementos almacenados en el nodo */
    struct NodoArbol123 *hijo1, *hijo2, *hijo3;  /* Apuntadores a los posibles 3 subárboles asociados */
} TArbol123, *Arbol123;
```

Sobre dicha representación, la implementación de la operación de búsqueda se hace con la siguiente rutina:

```
int estaArbol123( Arbol123 a, Tipo123 elem )
{   if( a == NULL )
        return FALSE;
    else if( a->raiz1 == elem || a->raiz2 == elem )
        return TRUE;
    else if( elem < a->raiz1 )
        return estaArbol123( a->hijo1, elem );
    else if( a->raiz2 == VACIO )
        return FALSE;
    else if ( elem < a->raiz2 )
        return estaArbol123( a->hijo2, elem );
    else
        return estaArbol123( a->hijo3, elem );
}
```

La implementación de las demás operaciones del TAD Arbol123 se propone más adelante como ejercicio, tanto sobre la representación antes planteada, como sobre otras representaciones interesantes.

Ejercicios Propuestos:

Utilizando la representación de apuntadores planteada en la sección anterior, desarrolle y pruebe las siguientes rutinas:

5.41. Arbol123 insArbol123(Arbol123 a, Tipo123 elem)
/* Agrega el elemento elem al árbol 1-2-3 */

5.42. Arbol123 elimArbol123(Arbol123 a, Tipo123 elem)
/* Elimina el elemento elem del árbol 1-2-3 */

5.43. void nivelesArbol123(Arbol123 a)
/* Presenta por niveles el árbol 1-2-3 en la pantalla */

5.44. int esArbol123(Arbol123 a)
/* Verifica que el árbol a cumpla todas las condiciones de un árbol 1-2-3 */

5.45. void inordenArbol123(Arbol123 a)
/* El recorrido en **inorden** de un árbol 1-2-3 visita los elementos de la estructura en orden ascendente. Esta rutina presenta por pantalla el recorrido en inorden de un árbol 1-2-3 */

Utilizando un esquema hijo izquierdo - hermano derecho como representación interna del TAD Arbol123 , desarrolle los siguientes ejercicios y pruebe las rutinas resultantes:

5.46. Especifique formalmente el esquema de representación y escriba la declaración de las estructuras de datos en C.

5.47. Arbol123 insArbol123(Arbol123 a, Tipo123 elem)
/* Agrega el elemento elem al árbol 1-2-3 */

5.48. Arbol123 elimArbol123(Arbol123 a, Tipo123 elem)
/* Elimina el elemento elem del árbol 1-2-3 */

5.49. int estaArbol123(Arbol123 a, Tipo123 elem)
/* Informa si el elemento elem está en el árbol 1-2-3 */

5.50. void nivelesArbol123(Arbol123 a)
/* Presenta por niveles el árbol 1-2-3 en la pantalla */

5.51. int esArbol123(Arbol123 a)
/* Verifica que el árbol a cumpla todas las condiciones de un árbol 1-2-3 */

5.52. void inordenArbol123(Arbol123 a)
/* El recorrido en **inorden** de un árbol 1-2-3 visita los elementos de la estructura en orden ascendente. Esta rutina presenta por pantalla el recorrido en inorden de un árbol 1-2-3 */

Suponiendo que se escoge para el TAD Arbol123 una representación secuencial similar a la presentada en el capítulo anterior para árboles binarios, desarrolle los siguientes ejercicios:

5.53. Especifique formalmente el esquema de representación y escriba la declaración de las estructuras de datos en C.

5.54. Arbol123 insArbol123(Arbol123 a, Tipo123 elem)
/* Agrega el elemento elem al árbol 1-2-3 */

5.55. Arbol123 elimArbol123(Arbol123 a, Tipo123 elem)
/* Elimina el elemento elem del árbol 1-2-3 */

5.56. int estaArbol123(Arbol123 a, Tipo123 elem)
/* Informa si el elemento elem está en el árbol 1-2-3 */

5.57. void nivelesArbol123(Arbol123 a)
/* Presenta por niveles el árbol 1-2-3 en la pantalla */

5.58. int esArbol123(Arbol123 a)
/* Verifica que el árbol a cumpla todas las condiciones de un árbol 1-2-3 */

5.59. void inordenArbol123(Arbol123 a)
/* El recorrido en **inorden** de un árbol 1-2-3 visita los elementos de la estructura en orden ascendente. Esta rutina presenta por pantalla el recorrido en inorden de un árbol 1-2-3 */

5.8. El TAD Arbol2-3: Un Árbol Triario Ordenado Balanceado

Una situación parecida a la que ocurre con los árboles binarios ordenados se tiene con los árboles 1-2-3: aunque en el caso promedio el acceso a la información es de complejidad logarítmica, en el peor de los casos sigue siendo lineal. Un árbol 2-3 resuelve este problema manteniendo balanceados sus subárboles, a costa de mayor dificultad de los algoritmos que implementan las operaciones de actualización.

Estos árboles fueron introducidos por R. Bayer y E. McCreight en 1972, con el principal objetivo de mejorar el tiempo de acceso en estructuras de datos manejadas en memoria secundaria, en las cuales el número de consultas a un registro influye de manera determinante en el tiempo de respuesta de la operación.

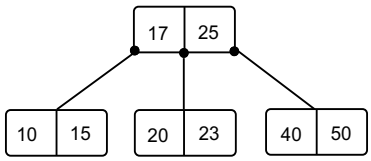
5.8.1. Definiciones y Conceptos Básicos

Un **árbol 2-3** es un árbol triario, con una estructura similar a la de un árbol 1-2-3, que cumple con las siguientes condiciones adicionales para garantizar su adecuado balanceo:

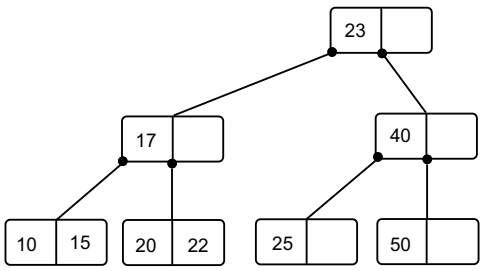
- Todas las hojas se encuentran en el mismo nivel
- Todos los nodos internos tienen por lo menos 2 subárboles asociados no vacíos, aunque la raíz derecha esté vacía

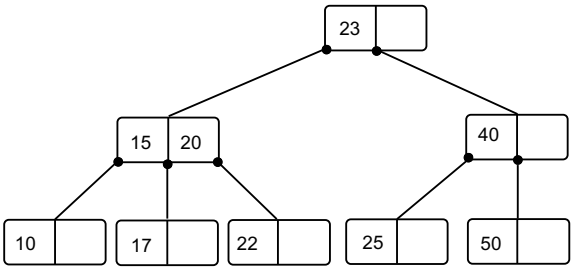
Ejemplo 5.14:

El siguiente es un árbol 2-3 con los mismos elementos que contiene el árbol 1-2-3 del ejemplo 5.12, pero cumpliendo las condiciones de balanceo antes enunciadas:



Si a este árbol se le inserta el elemento 22, debe alcanzar un nuevo estado de equilibrio, repartiendo sus elementos de manera que se conserven las propiedades de balanceo. Podría llegar, por ejemplo, a alguno de los siguientes estados válidos para un árbol 2-3, dependiendo del algoritmo de inserción:





Esto implica que las modificadoras del TAD deben tener en cuenta la situación completa del árbol para poder decidir el punto en el cual se agrega el nuevo valor, dificultando los algoritmos que llevan a cabo los procesos de actualización.

⏎

Un **árbol B** es un árbol n-ario ordenado y balanceado, que corresponde a una generalización de la estructura de un árbol 2-3. En este tipo de árboles, cada nodo tiene k elementos y $k+1$ subárboles B asociados, como se muestra en la figura 5.5. En particular, un árbol 2-3 es un árbol B de orden 3.

Un árbol B de orden k cumple las siguientes condiciones:

- Todas las hojas se encuentran en el mismo nivel
- Todos los nodos internos, excepto la raíz, tienen por lo menos (k + 1) / 2 subárboles asociados no vacíos
- El número de elementos de un nodo interno es uno menos que el número de subárboles asociados

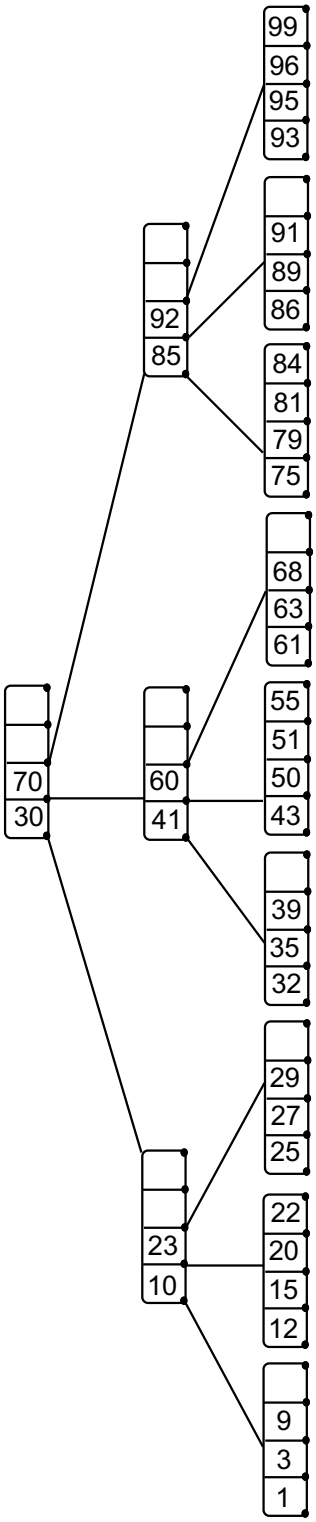
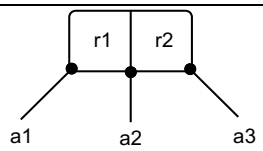


Fig. 5.5. - Ejemplo de un árbol B de orden 4

5.8.2. Especificación del TAD

TAD Arbol23[Tipo23]



{ inv: todas las hojas del árbol se encuentran en el mismo nivel,
a1, a2 y a3 son árboles 2-3,
r1 < r2 ∨ r2 = VACIO,
(es una hoja ∨
(a1 != Δ, a2 != Δ, a3 = Δ, r2 = VACIO, (e ∈ a1 ⇒ e < r1), (e ∈ a2 ⇒ r1 < e)) ∨
(a1 != Δ, a2 != Δ, a3 = Δ, r2 != VACIO, (e ∈ a1 ⇒ e < r1), (e ∈ a2 ⇒ r1 < e < r2)) ∨
(a1 != Δ, a2 != Δ, a3 != Δ, r2 != VACIO,
(e ∈ a1 ⇒ e < r1), (e ∈ a2 ⇒ r1 < e < r2), (e ∈ a3 ⇒ e > r2))) }

Constructoras:

• inicArbol23:		→ Arbol23

Modificadoras:

• insArbol23:	Arbol23 x Tipo23	→ Arbol23
• elimArbol23:	Arbol23 x Tipo23	→ Arbol23

Analizadoras:

• estaArbol23:	Arbol23 x Tipo23	→ int

```
Arbol23 inicArbol23( void )
/* Crea un árbol 2-3 vacío */

{ post: inicArbol23 = Δ }
```

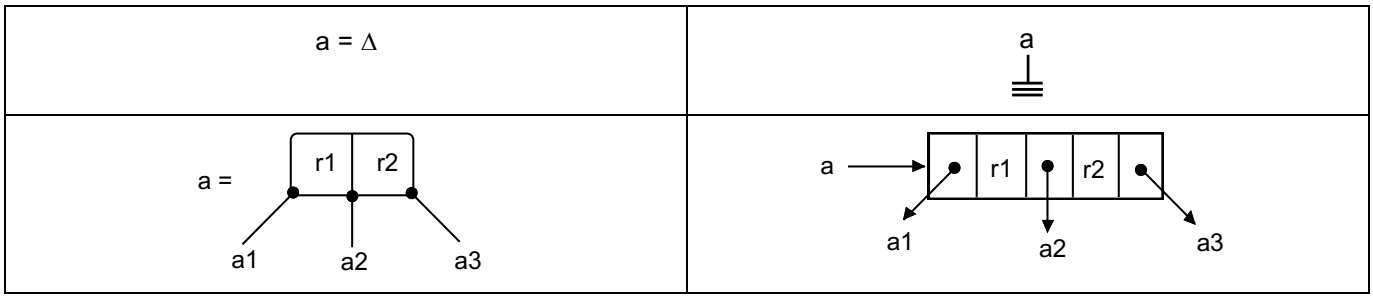
```
Arbol23 insArbol23( Arbol23 a, Tipo23 elem )
/* Agrega el elemento elem al árbol 2-3 */

{ pre: elem ∉ a, a = A }
{ post: insArbol23 = A + elem }
```

```
Arbol23 elimArbol23( Arbol23 a, Tipo23 elem )
/* Elimina el elemento elem del árbol 2-3 */

{ pre: elem ∈ a, a = A }
{ post: elimArbol23 = A - elem }
```

```
int estaArbol23( Arbol23 a, Tipo23 elem )
/* Informa si el elemento elem está en el árbol 2-3 */

{ post: estaArbol23 = ( elem ∈ a ) }
```

5.8.3. Estructuras de Datos

Existen muchas estructuras de datos posibles para implementar un árbol 2-3. Para ilustrar los algoritmos de actualización de este tipo de árbol se va a trabajar con una representación de apuntadores, basada en el siguiente esquema de representación:



La declaración de estas estructuras de datos se hace de la siguiente manera:

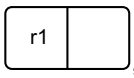
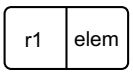
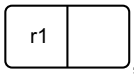
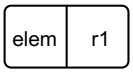
```
typedef struct NodoArbol23
{   Tipo23 raiz1, raiz2;                    /* Elementos almacenados en el nodo */
    struct NodoArbol23 *hijo1, *hijo2, *hijo3;   /* Apuntadores a los posibles 3 subárboles asociados */
} TArbol23, *Arbol23;
```

5.8.4. Algoritmo de Inserción

Contrario a lo que sucede con los árboles binarios ordenados y con los árboles 1-2-3, la estructura de un árbol 2-3 exige que el crecimiento no se haga a nivel de las hojas (aunque la inserción se sigue haciendo en las hojas), sino a nivel de la raíz, ya que todas las hojas se deben mantener siempre en el mismo nivel.

El proceso global de inserción comienza por localizar la hoja en la cual se debe agregar el elemento. En ese punto se pueden presentar varios casos:

- **Caso 1:** Existe espacio en el nodo. Se coloca allí adecuadamente el elemento y la estructura del árbol no se altera.

Situación inicial	Situación final
 , r1 < elem	
 , r1 > elem	

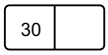
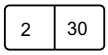
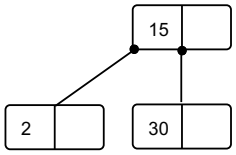
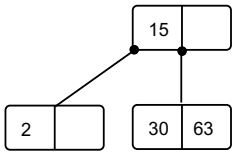
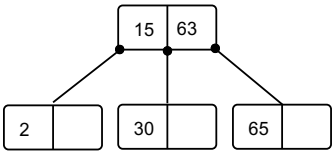
- **Caso 2:** El nodo está lleno. Este se debe partir en dos nodos del mismo nivel, repartiendo los tres elementos (dos elementos del nodo y el nuevo elemento) de la siguiente manera:

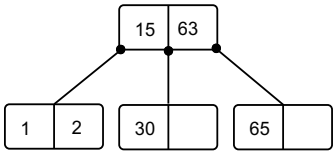
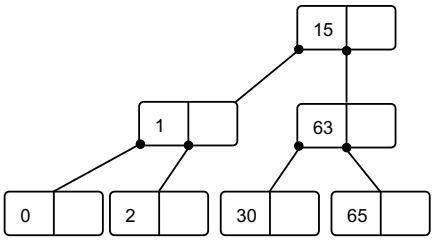
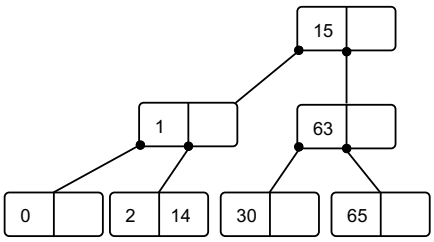
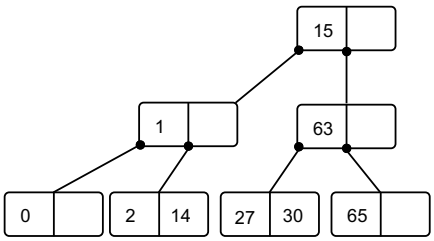
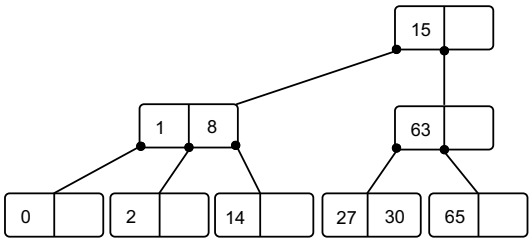
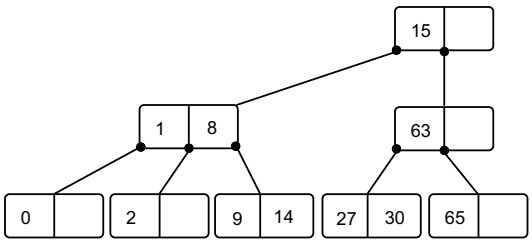
Situación inicial	Situación intermedia
r1 | r2, elem > r2	r1 , elem , sube r2
r1 | r2, elem < r1	elem , r2 , sube r1
r1 | r2, r1 < elem < r2	r1 , r2 , sube elem

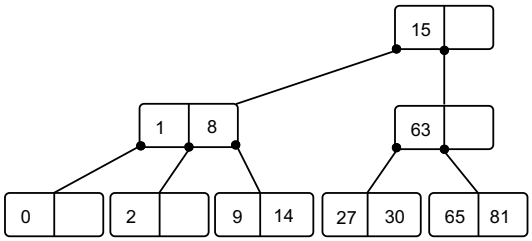
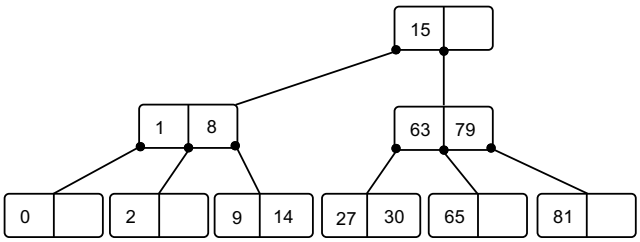
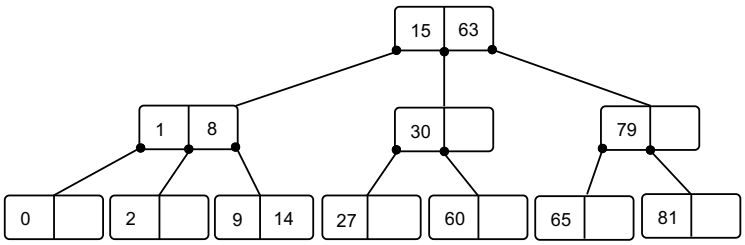
Luego, el elemento que no fue incluido en los dos nodos nuevos, se sube en la estructura y se inserta en su padre. El proceso se repite hacia arriba, ya que al partir en dos el nodo se está generando un nuevo subárbol que puede generar que los ancestros se tengan que partir a su vez para poderlo incluir, tal como se muestra en el siguiente ejemplo.

Ejemplo 5.15:

Considere la siguiente secuencia de inserciones en un árbol 2-3:

	Insertar el elemento 30: se crea una hoja y se coloca el elemento como raíz izquierda
	Insertar el elemento 2: corresponde al caso 1. Se mueve a la derecha la raíz izquierda para dar cabida al nuevo elemento.
	Insertar el elemento 15: corresponde al caso 2. Encuentra una hoja llena. La parte en dos nodos e inserta en el padre el elemento que se encuentre en la mitad de los tres (2 < 15 < 30). Como el padre es vacío, se crea un nuevo nivel, se coloca el elemento como la raíz izquierda del nodo, y se le asocian los dos nodos que se acaban de partir.
	Insertar el elemento 63: corresponde al caso 1
	Insertar el elemento 65: corresponde al caso 2. Se parte la hoja y sube al padre el elemento de la mitad (63). Al insertar dicho valor en el padre se trata como el caso 1, porque en el nodo hay espacio.

	Insertar el elemento 1: corresponde al caso 1.
	Insertar el elemento 0: corresponde al caso 2. Se parte la hoja [1, 2], se colocan allí los elementos 0 y 2, y sube el valor 1 a su padre. Como el nodo del padre [15, 63] está lleno también se debe partir, dejando en ese nivel los elementos 1 y 63, y subiendo el 15.
	Insertar el elemento 14: corresponde al caso 1.
	Insertar el elemento 27: corresponde al caso 1.
	Insertar el elemento 8: corresponde al caso 2. Se parte el nodo [2, 14] y sube el 8. Allí encuentra espacio y se coloca como raíz derecha.
	Insertar el elemento 9: corresponde al caso 1.

	Insertar el elemento 81: corresponde al caso 1.
	Insertar el elemento 79: corresponde al caso 2. Se parte el nodo [65, 81] y sube el elemento 79.
	Insertar el elemento 60: corresponde al caso 2. Se parte el nodo [27, 30], se incluye el 60 y sube el elemento 30. Como su padre está lleno se parte en los nodos [30] y [79], y sube el elemento 63. Este elemento se sitúa en la raíz derecha del árbol, donde hay espacio libre.

⏎

Las rutinas que implementan la operación de inserción en un árbol 2-3 se presentan a continuación:

- La operación de inserción en un árbol 2-3 considera dos casos: si el árbol inicial es vacío, crea un nodo nuevo y coloca en su raíz izquierda el nuevo elemento. Si no es vacío, llama la rutina auxiliar insertar, la cual agrega el elemento y le informa si debe aumentar un nivel hacia arriba el árbol. En este último caso, en los parámetros por referencia derecho e izquierdo llegan los dos subárboles que se acaban de partir, y en el parámetro elem, el elemento que viene subiendo.

```
Arbol23 insArbol23( Arbol23 a, Tipo23 elem )
{   Arbol23 derecho = NULL, izquierdo = NULL, nuevo;
    if( a == NULL)
    {     a = ( Arbol23 )malloc( sizeof ( TArbol23 ) );
          a->raiz1 = elem;
          a->raiz2 = VACIO;
          a->hijo1 = a->hijo2 = a->hijo3 = NULL;
          return a;
    }
    else if( insertar( a, &elem, &derecho, &izquierdo ) )
    {     nuevo = ( Arbol23 )malloc( sizeof ( TArbol23 ) );
          nuevo->raiz1 = elem;
          nuevo->raiz2 = VACIO;
          nuevo->hijo1 = izquierdo;
          nuevo->hijo2 = derecho;
          nuevo->hijo3 = NULL;
          return nuevo;
    }
    else
          return a;
}
```

- La rutina insertar lleva todo el control de la inserción. Es recursiva y considera cuatro casos: (1) el árbol es una hoja, (2) el elemento se debe adicionar en el primer subárbol, (3) el elemento se debe adicionar en el segundo subárbol, (4) el elemento se debe adicionar en el tercer subárbol. En cualquiera de los tres últimos casos debe verificar si el proceso de inserción trae en ascenso un elemento, situación en la cual debe hacer las modificaciones estructurales necesarias para continuar adecuadamente el proceso, llamando la rutina indicada:

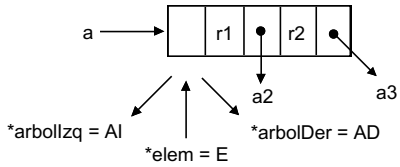
```
/* pre:   a es el árbol 2-3 original, *elem es el elemento que se va a insertar, *elem ∉ a */
/* post:  ( insertar = TRUE, se debe subir el elemento *elem, *arbolDer y *arbolIzq son los nodos partidos ) ∨
          ( insertar = FALSE, a incluye el elemento *elem */

static int insertar( Arbol23 a, Tipo23 *elem, Arbol23 *arbolDer, Arbol23 *arbolIzq )
{   if( a->hijo1 == NULL )
          return insHoja( a, elem, arbolDer, arbolIzq );
    else if( *elem < a->raiz1 )
          return insertar( a->hijo1, elem, arbolDer, arbolIzq ) ? subirInfo1( a, elem, arbolDer, arbolIzq ) : FALSE;
    else if( a->raiz2 == VACIO || *elem < a->raiz2 )
          return insertar( a->hijo2, elem, arbolDer, arbolIzq ) ? subirInfo2( a, elem, arbolDer, arbolIzq ) : FALSE;
    else
          return insertar( a->hijo3, elem, arbolDer, arbolIzq ) ? subirInfo3( a, elem, arbolDer, arbolIzq ) : FALSE;
}
```

- La rutina que inserta un elemento en una hoja considera los dos casos planteados antes en la teoría: si hay espacio para el elemento, reacomoda la información del nodo. Si no hay espacio, parte el nodo y comunica hacia arriba los dos árboles que se obtienen de dicho proceso, lo mismo que el elemento que debe subir al padre.

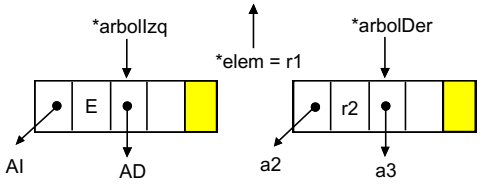
```
/* pre:   a es una hoja, *elem ∉ a */
/* post:  ( insHoja = TRUE, se debe subir el elemento *elem, *arbolDer y *arbolIzq son los nodos partidos ) ∨
          ( insHoja = FALSE, a incluye el elemento *elem */

static int insHoja( Arbol23 a, Tipo23 *elem, Arbol23 *arbolDer, Arbol23 *arbolIzq )
{   Arbol23 nuevo;
    if( a->raiz2 == VACIO )                    /* caso 1: hay espacio en el nodo */
    {     if( *elem < a->raiz1 )
          {     a->raiz2 = a->raiz1;
                a->raiz1 = *elem;
          }
          else
                a->raiz2 = *elem;
          return FALSE;
    }
    else                                       /*caso 2: no hay espacio en el nodo y se debe partir */
    {     nuevo =  ( Arbol23 )malloc( sizeof ( TArbol23 ) );
          nuevo->hijo1 = nuevo->hijo2 = nuevo->hijo3 = NULL;
          nuevo->raiz2 = VACIO;
          if( *elem < a->raiz1 )               /* sube la raíz 1 */
          {     nuevo->raiz1 = *elem;
                *elem = a->raiz1;
                a->raiz1 = a->raiz2;
                a->raiz2 = VACIO;
          }
          else if( *elem < a->raiz2 )          /* sube el elemento nuevo */
          {     nuevo->raiz1 = a->raiz1;
                a->raiz1 = a->raiz2;
                a->raiz2 = VACIO;
          }
          else                                 /* sube la raíz 2 */
          {     nuevo->raiz1 = a->raiz1;
                a->raiz1 = *elem;
                *elem = a->raiz2;
                a->raiz2 = VACIO;
          }
          *arbolIzq = nuevo;
          *arbolDer = a;
          return TRUE;
    }
}
```

- Las tres rutinas siguientes se encargan de realizar las transformaciones necesarias en el árbol cuando sube un elemento, teniendo en cuenta cuál fue el subárbol afectado en el proceso:

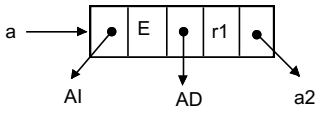
/* pre: a es el árbol original, se insertó un elemento en el subárbol 1,
se partió dicho subárbol en *arbolDer y *arbolIzq, viene subiendo el elemento *elem */



/* post: (subirInfo1 = TRUE, se debe subir el elemento *elem, *arbolDer y *arbolIzq son los nodos partidos) ∨



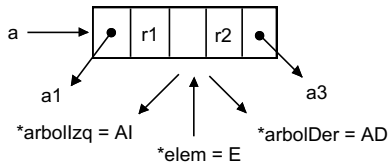
(subirInfo1 = FALSE, a ya incluye el elemento *elem */



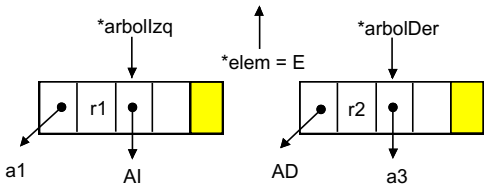

```
static int subirInfo1( Arbol23 a, Tipo23 *elem, Arbol23 *arbolDer, Arbol23 *arbolIzq )
{   int temp;
    Arbol23 nuevo;
    if( a->raiz2 == VACIO )                 /* hay campo en ese nodo: reorganizar */
    {   a->raiz2 = a->raiz1;
        a->hijo3 = a->hijo2;
        a->raiz1 = *elem;
        a->hijo1 = *arbolIzq;
        a->hijo2 = *arbolDer;
        return FALSE;
    }
    else                                    /* no hay campo en el nodo: partir y subir */
    {   nuevo =  ( Arbol23 )malloc( sizeof ( TArbol23 ) );
        nuevo->hijo3 = NULL;
        nuevo->raiz2 = VACIO;
        nuevo->raiz1 = a->raiz2;
        nuevo->hijo1 = a->hijo2;
        nuevo->hijo2 = a->hijo3;
        temp = *elem;
        *elem = a->raiz1;
        a->raiz1 = temp;
        a->raiz2 = VACIO;                   a->hijo1 = *arbolIzq;
        a->hijo2 = *arbolDer;               a->hijo3 =NULL;
        *arbolIzq = a;
        *arbolDer = nuevo;
        return TRUE;
    }
}
```

/* pre: a es el árbol original, se insertó un elemento en el subárbol 2,

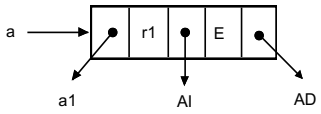
se partió dicho subárbol en *arbolDer y *arbolIzq, viene subiendo el elemento *elem */



/* post: (subirInfo2 = TRUE, se debe subir el elemento *elem, *arbolDer y *arbolIzq son los nodos partidos) ∨

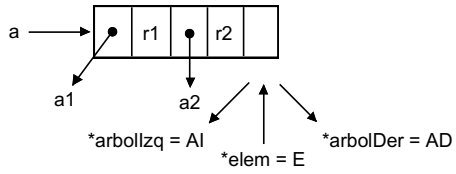


(subirInfo2 = FALSE, a ya incluye el elemento *elem */

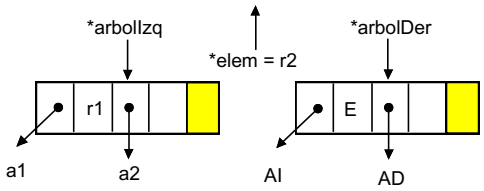



```
static int subirInfo2( Arbol23 a, Tipo23 *elem, Arbol23 *arbolDer, Arbol23 *arbolIzq )
{   Arbol23 nuevo;
    if( a->raiz2 == VACIO )                    /* hay campo en ese nodo: reorganizar */
    {     a->raiz2 = *elem;
          a->hijo2 = *arbolIzq;
          a->hijo3 = *arbolDer;
          return FALSE;
    }
    else                                       /* no hay campo en el nodo: partir y subir */
    {     nuevo =  ( Arbol23 )malloc( sizeof ( TArbol23 ) );
          nuevo->hijo3 = NULL;
          nuevo->raiz2 = VACIO;
          nuevo->raiz1 = a->raiz2;
          nuevo->hijo1 = *arbolDer;
          nuevo->hijo2 = a->hijo3;
          a->hijo2 = *arbolIzq;
          a->hijo3 =NULL;
          a->raiz2 = VACIO;
          *arbolIzq = a;
          *arbolDer = nuevo;
          return TRUE;
    }
}
```

```
/* pre:   a es el árbol original, se insertó un elemento en el subárbol 3,
          se partió dicho subárbol en *arbolDer y *arbolIzq, viene subiendo el elemento *elem */
```




```
/* post:  subirInfo3 = TRUE, se debe subir el elemento *elem, *arbolDer y *arbolIzq son los nodos partidos */
```

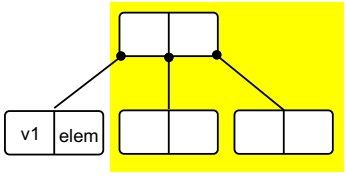
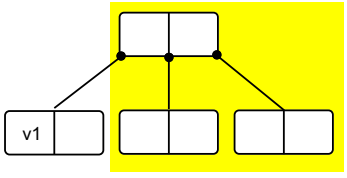


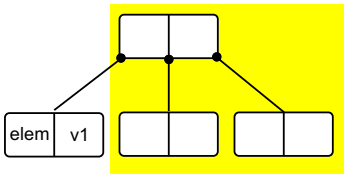
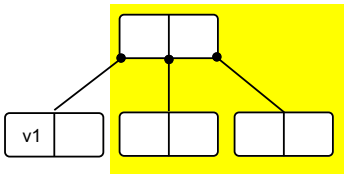
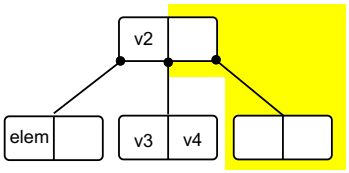
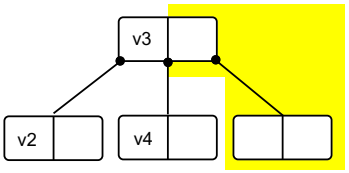
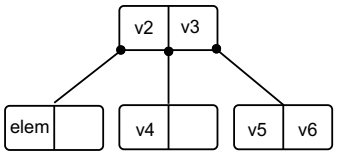
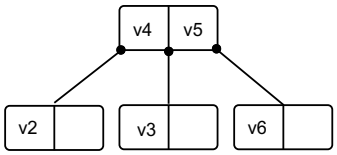
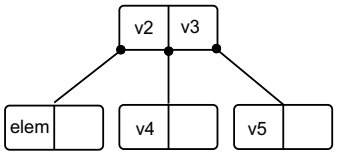
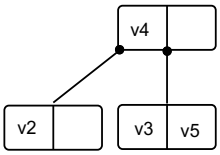
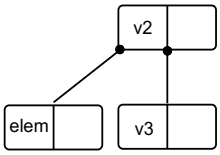
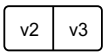
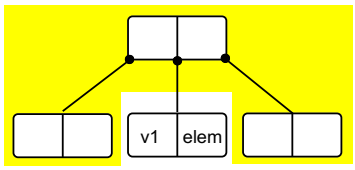
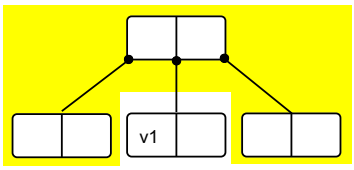
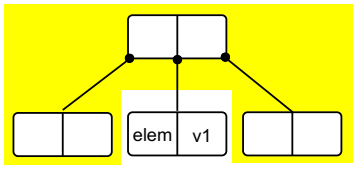
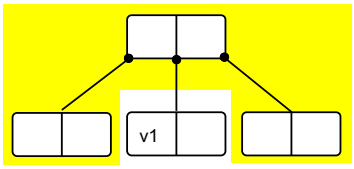
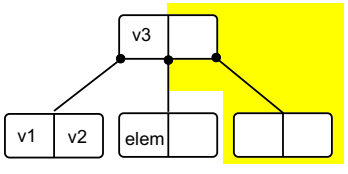
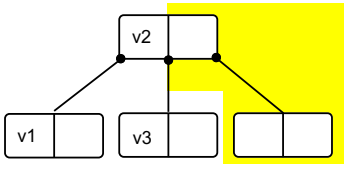

```
static int subirInfo3( Arbol23 a, Tipo23 *elem, Arbol23 *arbolDer, Arbol23 *arbolIzq )
{   Arbol23 nuevo;
    nuevo =  ( Arbol23 )malloc( sizeof ( TArbol23 ) );
    nuevo->hijo3 = NULL;
    nuevo->raiz2 = VACIO;
    /* No hay campo en el nodo: tiene que seguir subiendo */
    nuevo->raiz1 = *elem;
    nuevo->hijo1 = *arbolIzq;
    nuevo->hijo2 = *arbolDer;
    *elem = a->raiz2;
    a->raiz2 = VACIO;
    a->hijo3 = NULL;
    *arbolIzq = a;
    *arbolDer = nuevo;
    return TRUE;
}
```

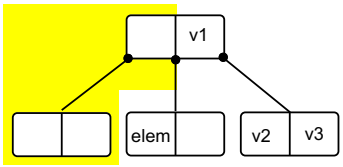
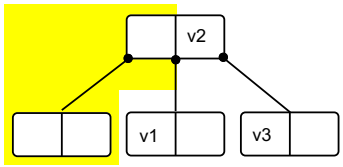
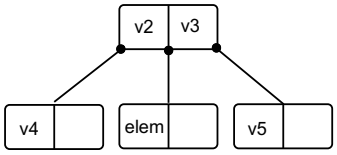
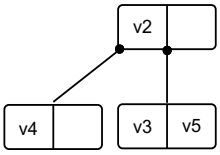
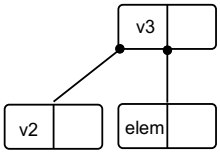
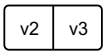
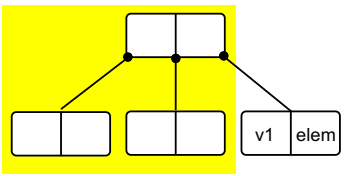
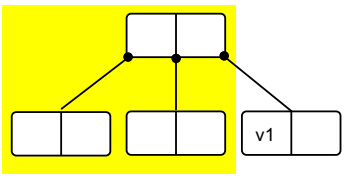
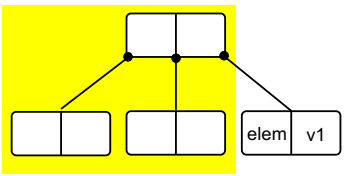
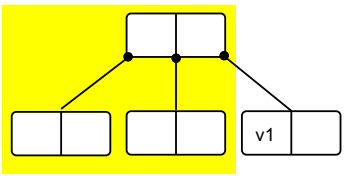
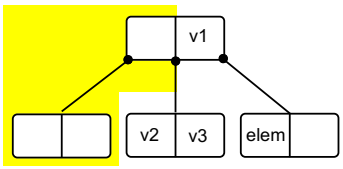
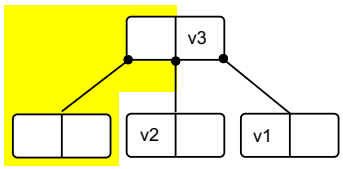
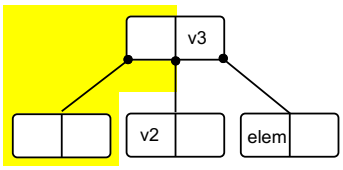
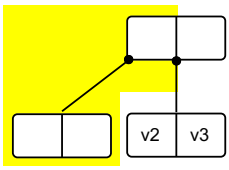
5.8.5. Algoritmo de Eliminación

El proceso de supresión de un elemento de un árbol 2-3 considera dos grandes casos, los cuales terminan siempre en el mismo proceso:

Caso 1: El elemento que se quiere eliminar está en una hoja. Allí se pueden presentar las siguientes situaciones, con la respectiva solución:

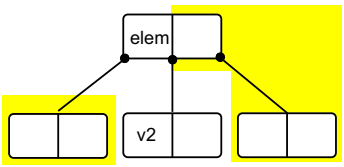
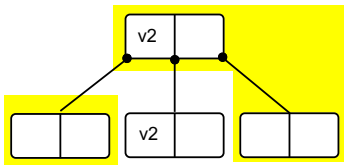
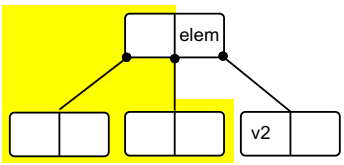
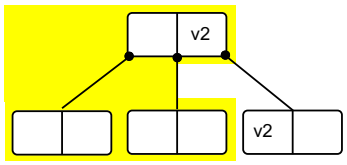
Caso	Situación inicial	Solución
A		

B		
C		
D		
E		
F		
G		
H		
I		

J		
K		
L		
M		
N		
O		
P		

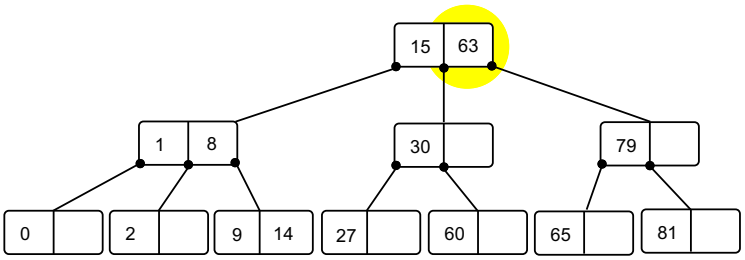
Sólo en las situaciones F y L el árbol pierde un nivel, caso en el cual se debe replicar recursivamente la política de préstamo entre hermanos, pero teniendo cuidado de mover adecuadamente los subárboles correspondientes, tal como se muestra en el ejemplo 5.16.

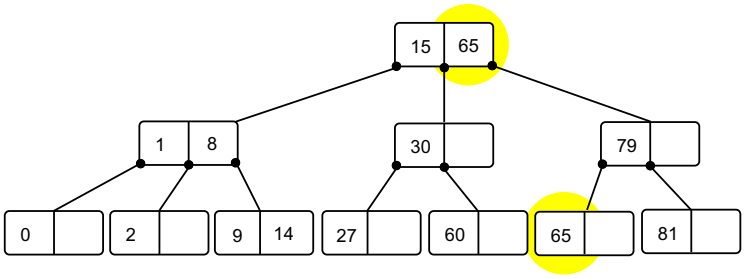
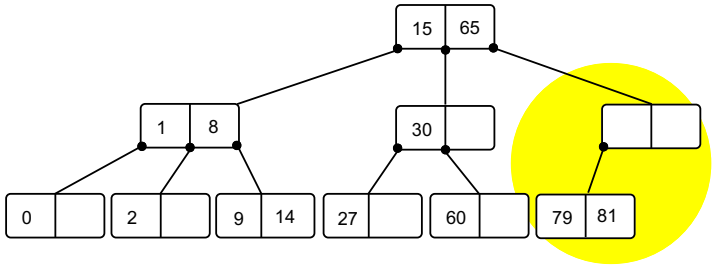
Caso 2: El elemento que se quiere eliminar no está en una hoja. En ese caso, se busca un valor que se encuentre en una hoja y que pueda remplazar el valor en cuestión (siguiendo un proceso equivalente al utilizado en árboles binarios ordenados, en el cual se localiza el menor valor del subárbol derecho), y luego se utiliza la solución planteada para el caso 1.

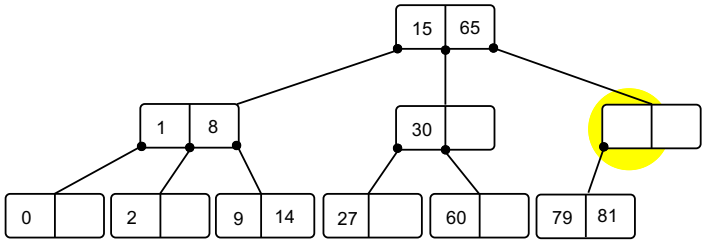
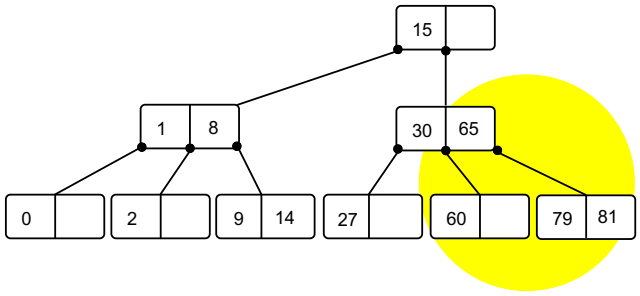
	Situación inicial	Situación intermedia: Eliminar v2 de una hoja (Caso 1)
Q		
R		

Ejemplo 5.16:

Considere el siguiente proceso para eliminar el elemento 63 del árbol 2-3 de la figura:



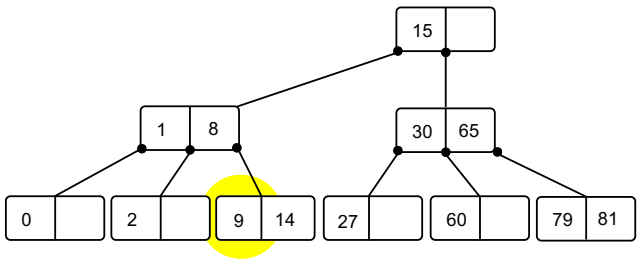
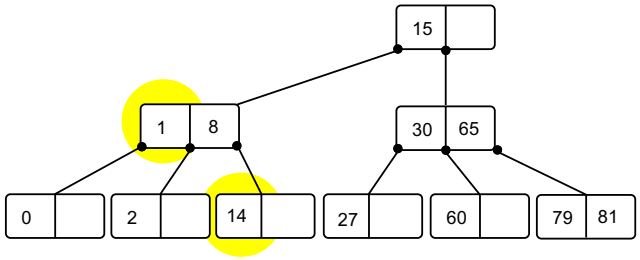
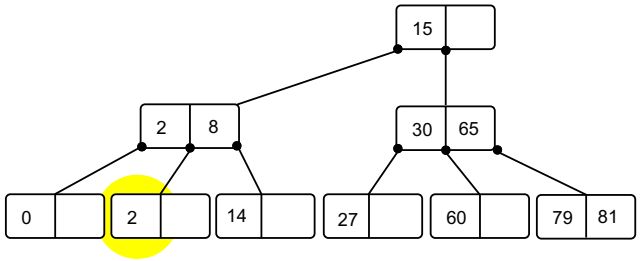
	(1) Se busca un valor en una hoja que pueda ocupar el espacio que va a liberar el 63. Se escoge para esto el elemento 65 (caso **2-R**), y se replantea el problema para eliminar este valor.
	(2) El tercer subárbol entra en el caso **1-F**, en el cual se pierde un nivel. Se llega al siguiente caso intermedio, porque el árbol debe decrecer por la raíz.

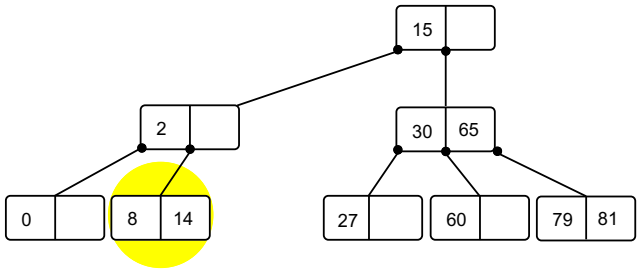
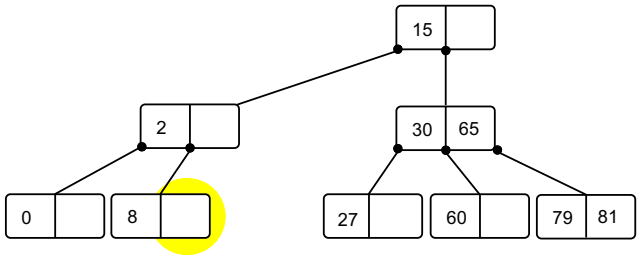
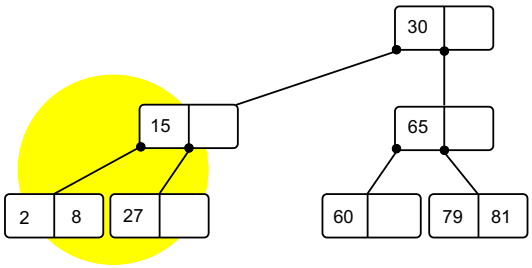
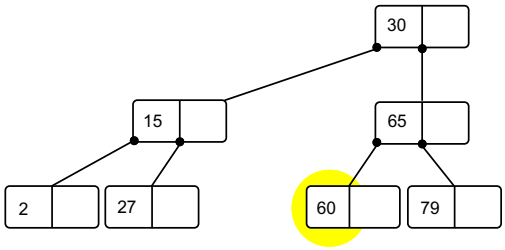
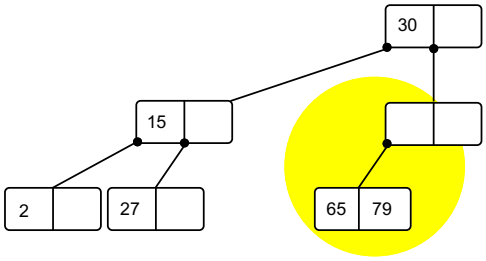
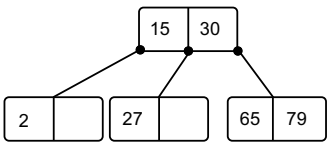
	(3) El árbol completo llega al caso **1-P**, en el cual el segundo subárbol no tiene raíz derecha para prestársela al tercer subárbol. Se debe aplicar la solución planteada para ese caso, pero teniendo en cuenta que el elemento no es una hoja, y que también deben moverse los subárboles asociados.
	(4) El proceso termina porque el árbol ya es estructuralmente correcto.

⏎

Ejemplo 5.17:

Considere la siguiente secuencia de eliminaciones del árbol 2-3 de la figura:

	(1) Eliminar el valor 9. Corresponde al caso **1-N**. La solución no modifica la estructura del árbol.
	(2) Eliminar el valor 1. Corresponde al caso **2-Q**. Se remplaza por el valor 2 y se elimina este nuevo valor.
	(3) Al tratar de eliminar el valor 2 de una hoja, se llega al caso **1-K**, cuya solución deja el árbol balanceado.

	(4) Eliminar el valor 14.
	(5) Eliminar el elemento 0. El primer subárbol pierde un nivel, y toma la raíz izquierda del hermano para compensar el nivel perdido. Al pasar la raíz izquierda al padre, debe reacomodar todos los subárboles.
	(6) Eliminar los elementos 81 y 8.
	(7) Eliminar el elemento 60.
	(8) El segundo subárbol pierde un nivel, y no puede tomar ningún otro elemento del árbol para compensarlo. Se llega a la siguiente situación intermedia:
	(9) El árbol completo perdió un nivel

La rutina que elimina un elemento de un árbol 2-3 llama una función auxiliar (eliminar), la cual suprime el elemento pedido e informa si el árbol completo debe perder un nivel. En este último caso, como el árbol resultante viene como primer subárbol, elimina el nodo de la raíz y deja como respuesta dicho subárbol.

```
/* pre: a = A, elem está en a */
/* post: a = A - elem */

Arbol23 elimArbol23( Arbol23 a, Tipo23 elem )
{   Arbol23 p;
    if( eliminar( a, elem ) )
    {     p = a;
          a = a->hijo1;
          free( p );
    }
    return a;
}
```

La función eliminar lleva todo el control del proceso de supresión de un valor. Primero considera el caso en el cual el elemento ya ha sido localizado (en una hoja o en un nodo interior), y luego plantea diferentes avances de la recursión según el valor del elemento. Utiliza 3 rutinas auxiliares (restaurar1, restaurar2, restaurar3) para alterar la estructura, de tal forma que se restauren las características del árbol, teniendo en cuenta el subárbol sobre el cual se ha hecho la operación. También se vale de una función auxiliar (menor) para obtener el menor elemento de un árbol 2-3.

Al final, la función retorna TRUE si el árbol perdió un nivel, y FALSE en caso contrario.

```
/* pre: a = A, elem está en a */
/* post: ( eliminar = TRUE, a = A - elem, el árbol perdió el nivel de la raíz ) ∨
         ( eliminar = FALSE, a = A - elem, el árbol no perdió ningún nivel ) */
```

```
int eliminar( Arbol23 a, Tipo23 elem )
{   Tipo23 temp;
    if( a->raiz1 == elem )
    {     if( a->hijo1 == NULL )          /* es una hoja */
          {     if( a->raiz2 == VACIO )
                      return TRUE;
                else
                      a->raiz1 = a->raiz2;
                      a->raiz2 = VACIO;
                      return FALSE;
          }
          else
          {     temp = menor( a->hijo2 );
                a->raiz1 = temp;
                return eliminar( a->hijo2, temp ) ? restaurar2( a ) : FALSE;
          }
    }
    else if( a->raiz2 == elem )
    {     if( a->hijo1 == NULL )          /* es una hoja */
          {     a->raiz2 = VACIO;
                return FALSE;
          }
          else
          {     temp = menor( a->hijo3 );
                a->raiz2 = temp;
                return eliminar( a->hijo3, temp ) ? restaurar3( a ) : FALSE;
          }
    }
    else if( elem < a->raiz1 )
          return eliminar( a->hijo1, elem ) ? restaurar1( a ) : FALSE;
    else if( a->raiz2 == VACIO || elem < a->raiz2 )
          return eliminar( a->hijo2, elem ) ? restaurar2( a ) : FALSE;
    else
          return eliminar( a->hijo3, elem ) ? restaurar3( a ) : FALSE;
}
```

La función que calcula el menor elemento de un árbol 2-3 se plantea de manera iterativa, avanzando siempre sobre el primer subárbol hasta que éste sea vacío:

```
/* pre: a no es vacío */
/* post: menor = menor elemento del árbol a */

Tipo23 menor( Arbol23 a )
{   while( a->hijo1 != NULL )
          a = a->hijo1;
    return a->raiz1;
}
```

La rutina restaurar1 toma la raíz izquierda del segundo subárbol para intentar suplir el vacío que tiene en su raíz izquierda. Si éste tiene raíz derecha, reacomoda este valor y los subárboles asociados. Si no tiene raíz derecha, hace una llamada a la función restaurar2, para que resuelva el problema de colocar un elemento en la raíz izquierda del segundo subárbol.

```
/* pre: a = A, el primer subárbol de a perdió el nivel de la raíz */
/* post: ( restaurar = TRUE, a y A tienen los mismos elementos, el árbol a perdió el nivel de la raíz ) ∨
         ( restaurar = FALSE, a y A tienen los mismos elementos, el árbol a tiene todos los niveles completos ) */

int restaurar1( Arbol23 a )
{   a->hijo1->raiz1 = a->raiz1;
    a->raiz1 = a->hijo2->raiz1;
    a->hijo1->hijo2 = a->hijo2->hijo1;
    a->hijo2->hijo1 = a->hijo2->hijo2;
    a->hijo2->hijo2 = NULL;
    if( a->hijo2->raiz2 != VACIO )
    {     a->hijo2->raiz1 = a->hijo2->raiz2;
          a->hijo2->raiz2 = VACIO;
          a->hijo2->hijo2 = a->hijo2->hijo3;
          a->hijo2->hijo3 = NULL;
          return FALSE;
    }
    return restaurar2( a );
}
```

La rutina restaurar2 intenta llenar el espacio de la raíz izquierda del segundo subárbol tomando un valor de alguno de sus hermanos. Si no puede reacomoda la información y pierde un nivel:

```
/* pre: a = A, el segundo subárbol de a perdió el nivel de la raíz */
/* post: ( restaurar = TRUE, a y A tienen los mismos elementos, el árbol a perdió el nivel de la raíz ) ∨
         ( restaurar = FALSE, a y A tienen los mismos elementos, el árbol a tiene todos los niveles completos ) */

int restaurar2( Arbol23 a )
{   if( a->raiz2 != VACIO )
    {     a->hijo2->raiz1 = a->raiz2;
          a->hijo2->hijo2 = a->hijo3->hijo1;
          a->raiz2 = a->hijo3->raiz1;
          a->hijo3->hijo1 = a->hijo3->hijo2;
          if( a->hijo3->raiz2 != VACIO )
          {     a->hijo3->raiz1 = a->hijo3->raiz2;
                a->hijo3->raiz2 = VACIO;
                a->hijo3->hijo2 = a->hijo3->hijo3;
                a->hijo3->hijo3 = NULL;
                return FALSE;
          }
          return restaurar3( a );
    }
    else if( a->hijo1->raiz2 != VACIO )
    {     a->hijo2->raiz1 = a->raiz1;
          a->raiz1 = a->hijo1->raiz2;
          a->hijo1->raiz2 = VACIO;
          a->hijo2->hijo2 = a->hijo2->hijo1;
          a->hijo2->hijo1 = a->hijo1->hijo3;
          a->hijo1->hijo3 = NULL;
          return FALSE;
    }
```

```
    else
    {   a->hijo1->raiz2 = a->raiz1;
        a->hijo1->hijo3 = a->hijo2->hijo1;
        free( a->hijo2 );
        return TRUE;
    }
}
```

La rutina restaurar3 realiza el proceso equivalente sobre el tercer subárbol:

```
/* pre: a = A, el tercer subárbol de a perdió el nivel de la raíz */
/* post: ( restaurar = TRUE, a y A tienen los mismos elementos, el árbol a perdió el nivel de la raíz ) ∨
         ( restaurar = FALSE, a y A tienen los mismos elementos, el árbol a tiene todos los niveles completos ) */

int restaurar3( Arbol23 a )
{   if( a->hijo2->raiz2 != VACIO )
    {   a->hijo3->raiz1 = a->raiz2;
        a->raiz2 = a->hijo2->raiz2;
        a->hijo2->raiz2 = VACIO;
        a->hijo3->hijo2 = a->hijo3->hijo1;
        a->hijo3->hijo1 = a->hijo2->hijo3;
        a->hijo2->hijo3 = NULL;
    }
    else
    {   a->hijo2->raiz2 = a->raiz2;
        a->hijo2->hijo3 = a->hijo3->hijo1;
        a->raiz2 = VACIO;
        free( a->hijo3 );
        a->hijo3 = NULL;
    }
    return FALSE;
}
```

Ejercicios Propuestos

5.60. Muestre los estados intermedios que alcanza un árbol 2-3 al insertar la siguiente secuencia de valores:
10 - 20 - 30 - 40 - 50 - 60 - 70 - 80 - 90 - 100 - 110 - 120 - 130 - 140 - 150 - 160 - 170 - 180 - 190 - 200

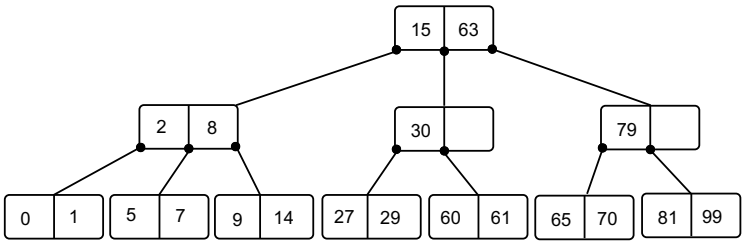
5.61. Muestre los estados intermedios que alcanza un árbol 2-3 al insertar la siguiente secuencia de valores:
75 - 90 - 23 - 12 - 1 - 56 - 32 - 45 - 19 - 43 - 7 - 14 - 86

5.62. Muestre los estados intermedios que alcanza un árbol 2-3 al insertar la siguiente secuencia de valores:
65 - 75 - 85 - 15 - 25 - 35 - 45 - 55 - 95 - 100 - 11 - 22 - 33 - 44 - 56 - 67 - 78 - 89

5.63. Muestre los estados intermedios que alcanza un árbol 2-3 al insertar la siguiente secuencia de valores:
11 - 56 - 67 - 33 - 65 - 44 - 89 - 75 - 45 - 55 - 95 - 78 - 22 - 85 - 15 - 25 - 35 - 100

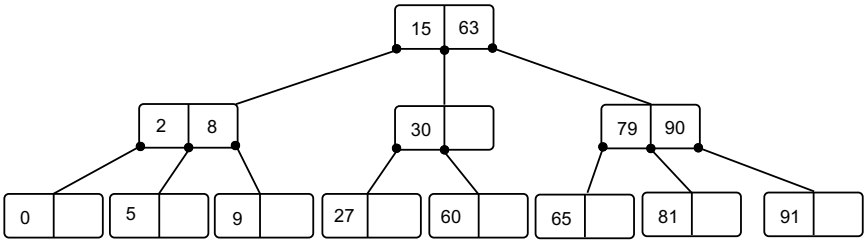
5.64. Para el árbol 2-3 de la figura, muestre los estados intermedios al eliminar la siguiente secuencia de valores:

15 - 63 - 14 - 79 - 27 - 29 - 0 - 1 - 81



5.65. Para el árbol 2-3 de la figura, muestre los estados intermedios al eliminar la siguiente secuencia de valores:

0 - 5 - 9 - 27 - 60 - 65 - 81 - 91 - 2 - 8 - 30 - 79 - 90 - 15 - 63



5.66. ¿Cuál es el número mínimo y máximo de elementos que puede contener un árbol 2-3 con k niveles?

5.67. ¿Cuál es el número mínimo y máximo de elementos que puede contener un árbol B de orden n con k niveles?

5.68. Calcule la complejidad de cada una de las operaciones del TAD Arbol23, considerando el peor de los casos.

5.69. 💾 int esArbol23(Arbol23 a)
/* Informa si un árbol 2-3 es estructuralmente correcto */

5.70. Diseñe unas estructuras de datos adecuadas para implementar el TAD ArbolB. Justifique su diseño.

5.71. 💻 Utilizando el resultado del ejercicio anterior, implemente la operación de búsqueda en un árbol B.

5.72. 💻 🕒 Utilizando el resultado del ejercicio 5.70, implemente la operación de inserción en un árbol B, generalizando el algoritmo presentado en la sección anterior.

5.73. 💻 🕒 Utilizando el resultado del ejercicio 5.70, implemente la operación de eliminación de un árbol B, generalizando el algoritmo presentado en la sección anterior.

5.74. 💻 🕒🕒 Un **árbol B*** es un árbol B, en el cual todos los nodos (excepto posiblemente la raíz) tienen por lo menos 2n/3 elementos, donde n es el orden del árbol. Esto obliga a que el algoritmo de inserción postergue la decisión de partir un nodo en dos, hasta cuando haya dos nodos contiguos completamente llenos. En caso contrario, los elementos de un nodo se desplazan horizontalmente para abrir el espacio necesario para acomodar el nuevo valor. Desarrolle el algoritmo de inserción para este tipo de árboles, utilizando la representación obtenida en el ejercicio 5.70.

5.75. Enuncie las ventajas y las desventajas de los árboles B* frente a los árboles B.

5.9. El TAD *Trie*: Conjunto de Palabras

Un ***trie*** es una estructura recursiva utilizada para representar de una manera compacta y eficiente un conjunto de palabras. Un *trie* es un árbol n-ario, en el cual cada elemento es un caracter y cada rama una palabra del conjunto, como se muestra en la figura 5.6.

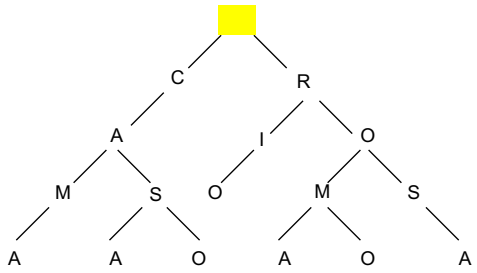


Fig. 5.6 - Ejemplo de un *trie*

En esta estructura todos lo prefijos comunes de las palabras comparten su representación. En el ejemplo de la figura 5.6, aparece representado el conjunto { "CAMA", "CASA", "CASO", "RIO", "ROMA", "ROMO", "ROSA" }, donde cada palabra comparte los prefijos comunes con las demás. En un *trie* es necesario agregar un caracter especial para los casos en los cuales un prefijo de una palabra, es una palabra en sí misma. Para agregar las palabras "CASAS" y "ROSAL" al *trie* de la figura 5.6, sin perder las palabras "CASA' y "ROSA" que son un prefijo de las nuevas, se debe incluir una marca especial para indicar que ambas ramas siguen siendo válidas, como se muestra en la figura 5.7.

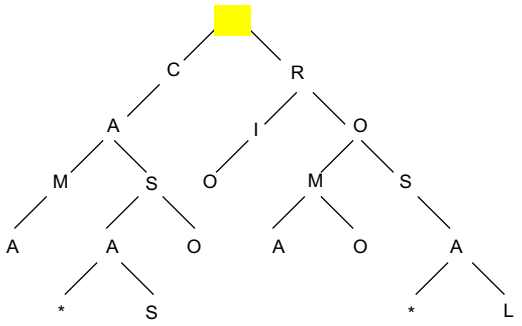
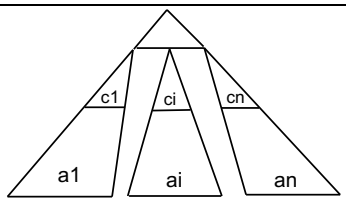


Fig 5.7 - Ejemplo de un *trie*

Para la administración de un *trie*, el TAD cuenta con 4 operaciones básicas: una constructora, dos modificadoras (adicionar y eliminar) y una analizadora (buscar), con la siguiente especificación:

TAD Trie	
	
{ inv:	todos los elementos son caracteres, $c_1 < ... < c_i < ... < c_n$, $a_1, a_2, ..., a_n$ son *tries*, solo las hojas pueden ser el caracter '*', la raíz de toda la estructura es el caracter ' ' }

Constructoras:

- inicTrie: → Trie

Modificadoras:

- insTrie: Trie x char * → Trie
- elimTrie: Trie x char * → Trie

Analizadoras:

- estaTrie: Trie x char * → int

```
Trie inicTrie( void )
/* Crea un trie vacío */

{ post: inicTrie = △ }
```

```
void insTrie( Trie t, char *elem )
/* Inserta una palabra al trie */

{ pre: elem ∉ t, t = T, strlen( elem ) > 0 }
{ post: t = T + elem }
```

```
void elimTrie( Trie t, char *elem )
/* Elimina una palabra del trie */

{ pre: elem ∈ t, t = T }
{ post: t = T - elem }
```

```
int estaTrie( Trie t, char *elem )
/* Informa si una palabra se encuentra en el trie */

{ post: estaTrie = ( elem ∈ t ) }
```

Para ilustrar la algorítmica de este tipo de estructuras, se escoge la representación hijo izquierdo - hermano derecho, con las siguientes declaraciones:

```
typedef struct NodoTRIE
{   char info;                  /* Raíz del árbol */
    struct NodoTRIE *hijo;      /* Encadenamiento al hijo izquierdo */
    struct NodoTRIE *hermano;   /* Encadenamiento al hermano derecho */
} TTrie, *Trie;
```

La rutina que se presenta a continuación implementa la operación de búsqueda en un *trie*. Es una rutina con un planteamiento iterativo sencillo, que aprovecha las facilidades de aritmética sobre apuntadores que da C, para disminuir el tamaño de la palabra buscada a medida que se van encontrando los caracteres iniciales.

```
int estaTrie( Trie t, char *elem )
{   for( t = t->hijo; t != NULL && strlen( elem ) != 0; elem++, t = t->hijo )
    {     for(; t != NULL && t->info < elem[ 0 ]; t = t->hermano );
          if( t == NULL || t->info > elem[ 0 ] )
                return FALSE;
    }
    return ( t == NULL && strlen( elem ) == 0 ) || t->info == '*';
}
```

Ejercicios Propuestos

5.76. 💻 Implemente y pruebe la operación que crea un *trie* vacío (inicTrie), utilizando como representación interna el esquema hijo izquierdo - hermano derecho.

5.77. 💻 Implemente y pruebe la operación que inserta una palabra en un *trie* (insTrie), utilizando como representación interna el esquema hijo izquierdo - hermano derecho.

5.78. 💻 Implemente y pruebe la operación que elimina una palabra de un *trie* (elimTrie), utilizando como representación interna el esquema hijo izquierdo - hermano derecho.

5.79. 💻 Implemente y pruebe una operación que verifique que un *trie* es estructuralmente correcto (esTrie), utilizando como representación interna el esquema hijo izquierdo - hermano derecho.

5.80. 💻 Implemente y pruebe la operación que crea un *trie* vacío (inicTrie), utilizando como representación interna un vector fijo de 27 posiciones, cada uno representando la respectiva letra.

5.81. 💻 Implemente y pruebe la operación que inserta una palabra en un *trie* (insTrie), utilizando como representación interna un vector fijo de 27 posiciones, cada uno representando la respectiva letra.

5.82. 💻 Implemente y pruebe la operación que elimina una palabra de un *trie* (elimTrie), utilizando como representación interna un vector fijo de 27 posiciones, cada uno representado la respectiva letra.

5.83. 💻 Implemente y pruebe la operación que busca una palabra en un *trie* (estaTrie), utilizando como representación interna un vector fijo de 27 posiciones, cada uno representando la respectiva letra.

5.84. 💻 Implemente y pruebe una operación que verifique que un *trie* es estructuralmente correcto (esTrie), utilizando como representación interna un vector fijo de 27 posiciones, cada uno representado la respectiva letra.

5.85. 💻 Implemente y pruebe la operación que crea un *trie* vacío (inicTrie), utilizando como representación interna un vector dinámico.

5.86. 💻 Implemente y pruebe la operación que inserta una palabra en un *trie* (insTrie), utilizando como representación interna un vector dinámico.

5.87. 💻 Implemente y pruebe la operación que elimina una palabra de un *trie* (elimTrie), utilizando como representación interna un vector dinámico.

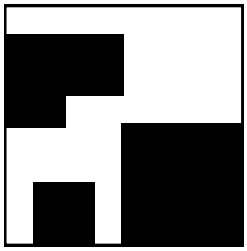
5.88. 💻 Implemente y pruebe la operación que busca una palabra en un *trie* (estaTrie), utilizando como representación interna un vector dinámico.

5.89. 💻 Implemente y pruebe una operación que verifique que un *trie* es estructuralmente correcto (esTrie), utilizando como representación interna un vector dinámico.

5.90. 💻 Diseñe e implemente un esquema de persistencia para un *trie*.

5.10. El TAD *Cuadtree*: Representación de Imágenes

Un ***cuadtree*** es una estructura arborescente utilizada con frecuencia para representar imágenes obtenidas de una cámara. Una **imagen** digitalizada en blanco y negro es un espacio rectangular compuesto de N x N ***pixels*** (donde N es una potencia de 2), cada uno representando un punto (blanco o negro) de la imagen. La representación usual de una imagen es una matriz de N x N, con ceros y unos mostrando los *pixels* blancos y negros, como se sugiere en la figura 5.8 para una imagen de 8 x 8.



0	0	0	0	0	0	0	0
1	1	1	1	0	0	0	0
1	1	1	1	0	0	0	0
1	1	0	0	0	0	0	0
0	0	0	0	1	1	1	1
0	0	0	0	1	1	1	1
0	1	1	0	1	1	1	1
0	1	1	0	1	1	1	1

Fig. 5.8 - Representación de una imagen con una matriz

Un *cuadtree* es un árbol 4-ario que permite representar de manera compacta una imagen, que en otro caso podría ocupar grandes cantidades de memoria. Un *cuadtree* tiene tres tipos de elementos: blancos, negros y grises, que representan un grupo de *pixels* de la imagen según el esquema recursivo de representación planteado en la figura 5.9.

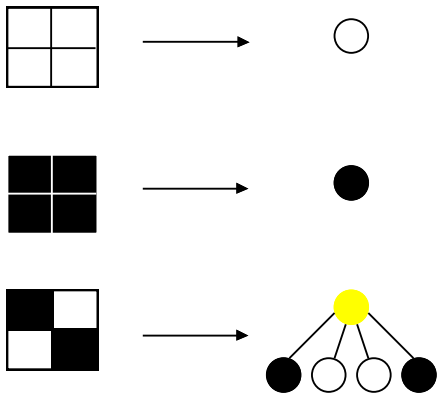
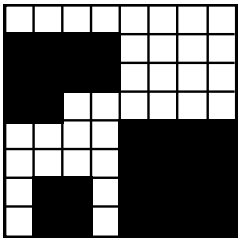


Fig. 5.9 - Tipos de nodos en un *cuadtree*

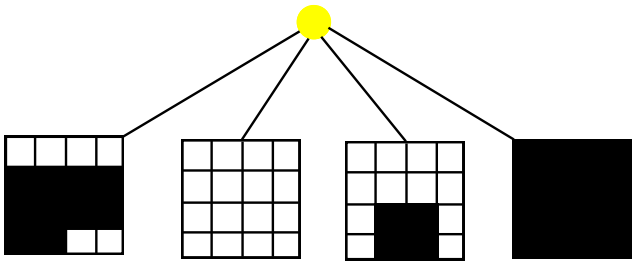
Esto es, la raíz del *cuadtree* representa la imagen completa: si la imagen es toda negra, el *cuadtree* tiene un solo elemento con valor NEGRO. Si la imagen es completamente blanca, tiene un solo elemento con valor BLANCO. En cualquier otro caso, la imagen se representa con un nodo GRIS, y se le asocian 4 subárboles, cada uno representando uno de los cuadrantes resultantes al dividir la imagen en 4 cuadros iguales (esto es posible gracias a que N es una potencia de 2). Al continuar este proceso recursivamente se obtiene la representación completa de la imagen por medio de una estructura arborescente compacta.

Ejemplo 5.18:

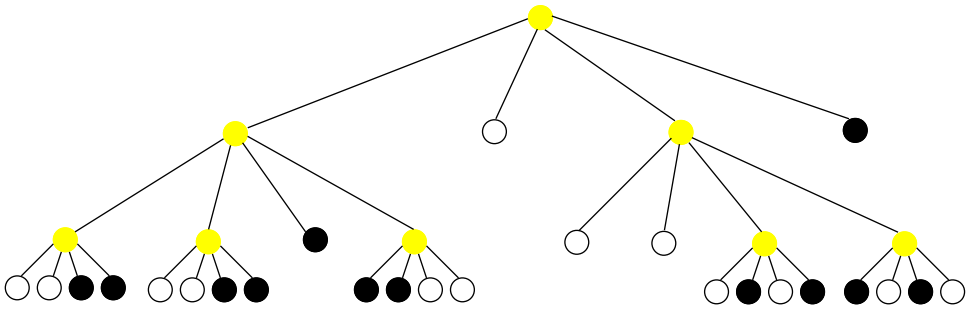
Considere la siguiente imagen de 8 x 8 *pixels*:



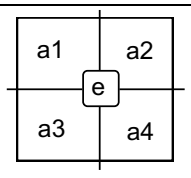
La raíz del *cuadtree* que la representa debe ser un nodo gris, para indicar que no todos los *pixels* de la imagen son del mismo color. Luego, se parte la imagen en 4 cuadrantes y se aplica recursivamente el mismo proceso, asociando los *cuadtrees* resultantes como subárboles:



Hasta llegar al siguiente árbol 4-ario, que representa la imagen completa:



El TAD Cuadtree tiene definidas cuatro operaciones básicas que permiten encender (colocar en blanco), apagar (colocar en negro), inicializar y preguntar por el valor de cualquier *pixel* de una imagen. Por simplicidad, el TAD presentado solo manipula imágenes de 512 x 512, pero sería perfectamente posible parametrizar dicho valor.

TAD Cuadtree

{ inv: (e = BLANCO $\Rightarrow a_1 = a_2 = a_3 = a_4 = \Delta$) (e = NEGRO $\Rightarrow a_1 = a_2 = a_3 = a_4 = \Delta$) (e = GRIS $\Rightarrow a_1 \neq \Delta, a_2 \neq \Delta, a_3 \neq \Delta, a_4 \neq \Delta$, los hijos no son todos BLANCOS, ni todos NEGROS) }

Constructoras:

• inicCuadtree:		→ Cuadtree

Analizadoras:

• valorPixel:	Cuadtree x int x int	→ int

Modificadoras:

• blancoPixel:	Cuadtree x int x int	→ Cuadtree
• negroPixel:	Cuadtree x int x int	→ Cuadtree

```
Cuadtree inicCuadtree( void )
/* Crea un cuadtree completamente blanco */

{ post: inicCuadtree =  BLANCO }
```

```
Cuadtree blancoPixel( Cuadtree c, int fil, int col )
/* Coloca en blanco el pixel de coordenadas [ fil, col ] */

{ pre: 0 ≤ fil ≤ 511, 0 ≤ col ≤ 511 }
{ post: blancoPixel[ fil, col ] = BLANCO, los demás pixels conservan su valor }
```

```
Cuadtree negroPixel( Cuadtree c, int fil, int col )
/* Coloca en negro el pixel de coordenadas[ fil, col ] */

{ pre: 0 ≤ fil ≤ 511, 0 ≤ col≤ 511 }
{ post: negroPixel[ fil, col ] = NEGRO, los demás pixels conservan su valor }
```

```
int valorPixel( Cuadtree c, int fil, int col )
/* Retorna el valor del pixel de coordenadas [ fil, col ] */

{ pre: 0 ≤ fil ≤ 511, 0 ≤ col ≤ 511 }
{ post: ( valorPixel = 0, c[ fil, col ] = NEGRO ) ∨ ( valorPixel = 1, c[ fil, col ] = BLANCO ) }
```

Para la implementación del TAD se van a utilizar las siguientes estructuras de datos:

```
#define BLANCO 1
#define NEGRO 0
#define GRIS -1

typedef struct NodoCuadtree
{   int info;                       /* BLANCO, NEGRO o GRIS */
    int infx, infy;                 /* Coordenada inferior del cuadrante */
    int supx, supy;                 /* Coordenada superior del cuadrante */
    struct NodoCuadtree *hijos[ 4 ];
} TCuadtree, *Cuadtree;
```

En la figura 5.10 aparece un ejemplo de la manera como serían las estructuras de datos:

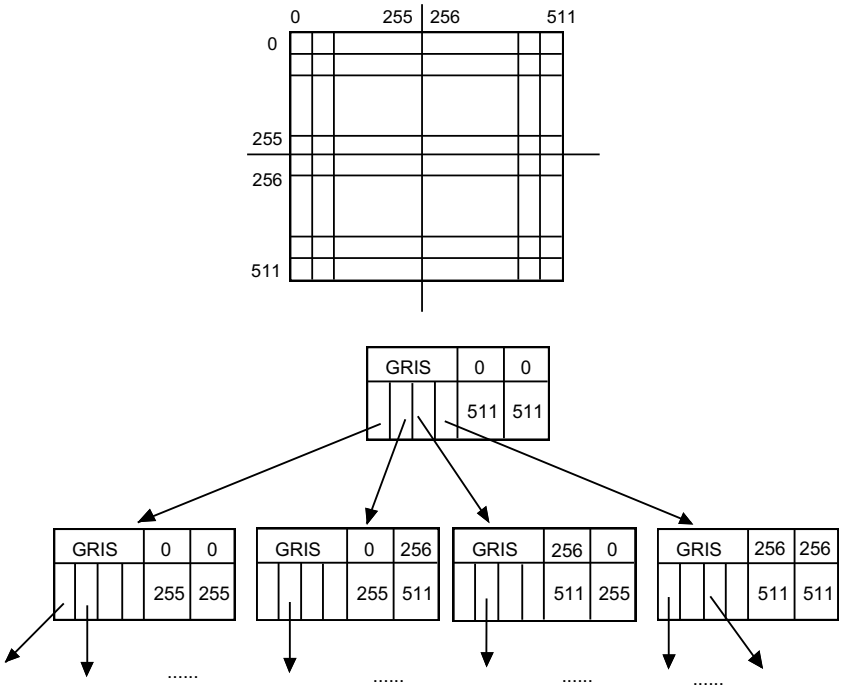


Fig. 5.10 - Estructuras de datos para un *cuadtree*

Se presenta a continuación la implementación de dos de las operaciones del TAD Cuadtree sobre las estructuras de datos antes sugeridas:

- La operación analizadora valorPixel considera dos casos: si la raíz tiene un color diferente a GRIS (BLANCO o NEGRO), el *pixel* tiene dicho valor. En caso contrario, localiza el cuadrante que contiene el *pixel* y desciende recursivamente por allí:

```
int valorPixel( Cuadtree c, int fil, int col )
{   int nc;
    if( c->info != GRIS )
        return c->info;                    /* NEGRO o BLANCO */
    else
    {   nc = cuadrante( c, fil, col );
        return valorPixel( c->hijos[ nc - 1 ], fil, col );
    }
}
```

- La función cuadrante es una rutina de utilidad, que retorna el cuadrante en el cual se encuentra el *pixel* [fil, col] dentro del *cuadtree* c. Para esto divide en dos el rango de valores en la coordenada x y el rango de valores en la coordenada y, y determina en cuál de los cuatro cuadrantes está el *pixel* buscado, como se muestra en la figura 5.11.

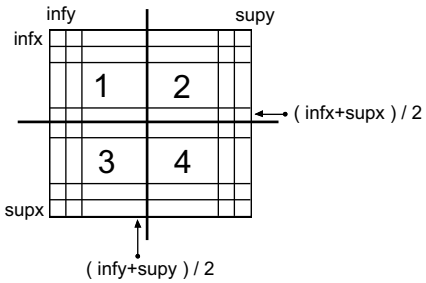


Fig. 5.11- División en 4 cuadrantes de un *cuadtree*

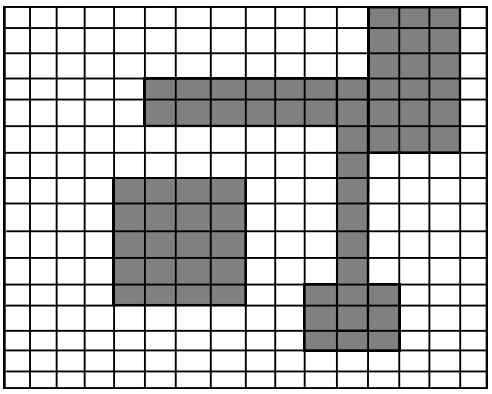
```
int cuadrante( Cuadtree c, int fil, int col )
{   if( fil <= ( c->infx + c->supx ) / 2 )                    /* cuadrante 1 ó 2 */
        return ( col <= ( c->infy + c->supy ) / 2 ) ? 1: 2;
    else
        return ( col <= ( c->infy + c->supy ) / 2 ) ? 3: 4;
}
```

- La rutina que coloca un *pixel* en blanco, considera tres casos: si la raíz es GRIS, localiza el cuadrante en el que se encuentra el *pixel* y hace una llamada recursiva sobre dicho subárbol. Si la dimensión del *cuadtree* es 1 X 1 (solo tiene un *pixel*), lo coloca en BLANCO. Si la raíz del *cuadtree* es de color NEGRO, la cambia por gris, crea los 4 subárboles correspondientes de color NEGRO, y finalmente hace una llamada recursiva sobre el subárbol que contiene el *pixel*. La rutina crear se encarga de construir un nodo de color NEGRO, con las coordenadas dadas en los parámetros.

```
Cuadtree blancoPixel( Cuadtree c, int fil, int col )
{   int nc, mx, my;
    if( c->info == GRIS )
    {   nc = cuadrante( c, fil, col );
        c->hijos[ nc-1 ] = blancoPixel( c->hijos[ nc-1 ], fil, col );
        if( c->hijos[ 0 ]->info == BLANCO && c->hijos[ 1 ]->info == BLANCO &&
            c->hijos[ 2 ]->info == BLANCO && c->hijos[ 3 ]->info == BLANCO )
        {   c->info = BLANCO;
            free( c->hijos[ 0 ] ); free( c->hijos[ 1 ] ); free( c->hijos[ 2 ] ); free( c->hijos[ 3 ] );
            c->hijos[ 0 ] = c->hijos[ 1 ] = c->hijos[ 2 ] = c->hijos[ 3 ] = NULL;
        }
    }
    else if( c->infx == c->supx )
        c->info = BLANCO;
    else if( c->info == NEGRO )
    {   c->info = GRIS;
        mx =( c->infx + c->supx ) / 2;
        my =( c->infy + c->supy) / 2;
        c->hijos[ 0 ] = crear( c->infx, c->infy, mx, my );
        c->hijos[ 1 ] = crear( c->infx, my+1, mx, c->supy );
        c->hijos[ 2 ] = crear( mx+1, c->infy, c->supx, my );
        c->hijos[ 3 ] = crear( mx+1, my+1, c->supx, c->supy );
        nc = cuadrante( c, fil, col );
        c->hijos[ nc-1 ] = blancoPixel( c->hijos[ nc-1 ], fil, col );
    }
    return c;
}
```

Ejercicios Propuestos

5.91. Dibuje el *cuadtree* correspondiente a la siguiente imagen de 16 x 16 *pixels*:



5.92. Implemente la operación del TAD Cuadtree que inicializa una imagen en BLANCO (inicCuadtree).

5.93. Implemente la operación del TAD Cuadtree que coloca un *pixel* de una imagen en NEGRO (negroPixel).

5.94. 🕒 Adicione al TAD Cuadtree una operación que coloque en color NEGRO una zona de la imagen, determinada por 2 coordenadas ([minx, miny], [maxx, maxy]). Utilice directamente las estructuras de datos. La complejidad de la operación debe ser lo mínima posible (v.g. no utilice la operación negroPixel para colocar en NEGRO cada *pixel* de la zona en cuestión).

5.95. 🕒 Adicione al TAD Cuadtree una operación que coloque en color BLANCO una zona de la imagen, determinada por 2 coordenadas ([minx, miny], [maxx, maxy]). Utilice directamente las estructuras de datos. La complejidad de la operación debe ser lo mínima posible (v.g. no utilice la operación blancoPixel para colocar en BLANCO cada *pixel* de la zona en cuestión).

5.11. El TAD Arbol *AND-OR*

Un árbol ***AND-OR*** es un árbol n-ario, utilizado para representar conocimiento sobre grupos de tareas que se deben ejecutar para lograr algún objetivo. En la figura 5.12 aparece un ejemplo de un árbol *AND-OR*, que explica la manera de hacer la tarea T1. Un árbol *AND-OR* tiene dos tipos de nodos: los nodos *AND* y los nodos *OR*. Por ejemplo, para el árbol de la figura 5.12, se tiene que para lograr T1, se deben realizar las tareas T2, T3, T4 (un nodo *AND*). Por su parte, para hacer la tarea T2, es suficiente con terminar T5 o T6 (un nodo *OR*). Las tareas T4, T5, T9, etc., que no están compuestas por subtareas, se conocen como tareas atómicas.

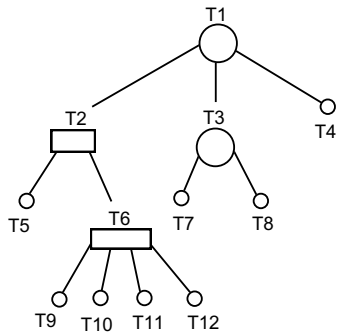


Fig. 5.12 - Ejemplo de árbol *AND-OR*

Una tarea atómica puede estar en dos estados: ejecutada o sin ejecutar. Por su parte, el estado de una tarea no atómica depende del estado de sus hijos: si es un nodo *AND*, está ejecutada si todos sus hijos están ejecutados. Un nodo *OR* está ejecutado si por lo menos uno de sus hijos ha sido realizado. Si se señala con una marca el hecho de haber sido ejecutado, algunos estados posibles para un árbol *AND-OR* aparecen en la figura 5.13.

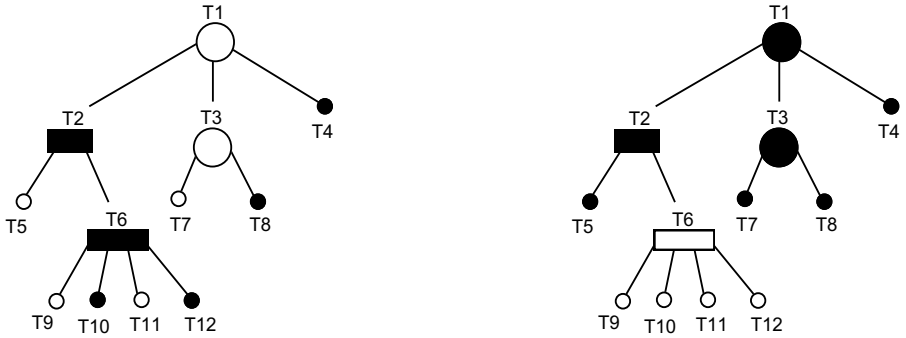


Fig. 5.13 - Arboles *AND-OR* con algunas tareas ejecutadas

En un árbol *AND-OR* cada tarea tiene un identificador, que es único al interior de la estructura, el cual, para efectos prácticos, se puede considerar una cadena de caracteres.

Ejercicios Propuestos

5.96. Diseñe y especifique el TAD ArbolAO.

5.97. Diseñe unas estructuras de datos para el TAD ArbolAO, especificado en el ejercicio anterior, y muestre claramente el esquema de representación propuesto.

5.98. Implemente y pruebe las operaciones del TAD ArbolAO sobre las estructuras de datos diseñadas en el ejercicio anterior.

Sobre las estructuras de datos diseñadas en el ejercicio 5.97, implemente las siguientes rutinas:

5.99. int contarAtomicas(ArbolAO a)
/* Calcula y retorna el número de tareas atómicas que aparecen como ejecutadas */

5.100. void hacerAtomica(ArbolAO a, char *nomTarea)
/* Coloca como realizada la tarea atómica de nombre nomTarea y actualiza todo el árbol. Supone que una tarea con ese nombre existe y que no está ejecutada */

5.101. void deshacerAtomica(ArbolAO a, char *nomTarea)
/* Coloca como no realizada la tarea atómica de nombre nomTarea y actualiza todo el árbol. Supone que una tarea con ese nombre existe y que está ejecutada */

5.102. int minimoAtomico(ArbolAO a)
/* Calcula y retorna el número mínimo de tareas atómicas que se deben hacer desde la situación actual para tener terminada la tarea global . */

5.103. Diseñe unas estructuras de datos para el TAD ArbolAO, de tal manera que si las operaciones críticas son las presentadas en los cuatro ejercicios anteriores, su implementación sea lo más eficiente posible.

5.12. Arboles de Juego

Otra aplicación de árboles n-arios es la representación de posibles movimientos para un juego, de tal manera que el algoritmo que se encarga de seleccionar una jugada se reduzca a buscar sobre dicha estructura un movimiento que maximice las posibilidades de ganar. La idea se ilustra en esta sección con el juego de *triqui*.

La raíz del árbol es el estado actual de la partida. Sus hijos son los estados que se pueden alcanzar en el juego, haciendo un movimiento. Los hijos de los hijos son los estados a los que se puede llegar después de hecha la jugada del contrincante. En la figura 5.14 se muestra un posible árbol de juego.

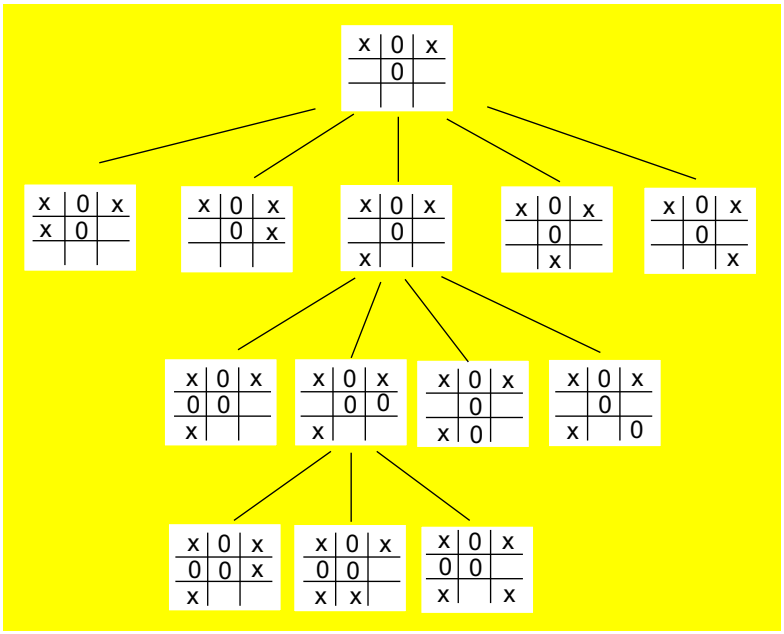


Fig. 5.14 - Árbol de juego para *triqui*

Para evaluar un movimiento de un jugador, se pueden examinar varias jugadas hacia adelante y ver qué tan buenas son las diferentes posiciones resultantes. Se define la **profundidad de análisis** como el número de movimientos futuros que van a ser considerados en la evolución de una jugada. Entre mayor profundidad se utilice, más certeza se tiene sobre la calidad de la jugada, pero más tiempo y memoria se requiere. Muchos juegos de ajedrez asocian el nivel de juego (principiante o experto) con la profundidad de análisis, y por eso hace mejores jugadas, pero les toma una mayor cantidad de tiempo.

Para seleccionar la mejor jugada, se debe intentar maximizar el valor de la jugada propia y minimizar el valor de la jugada del oponente. Esto es, hacer la mejor jugada posible, garantizando que la mejor respuesta del contrincante lo lleve a la peor situación posible.

Esto obliga a establecer una medida de calidad de una jugada, en términos del estado del tablero al cual se llega. El nivel de profundidad de análisis establece qué tan precisa es esta medida de calidad, puesto que considera una mayor o menor cantidad de información. El proceso de evaluación parte de la asignación de una calificación a cada hoja del árbol (el último nivel explorado). Este valor se calcula utilizando únicamente el estado del tablero en ese punto. Existen muchas formas de hacer esta evaluación, quedando en ella consignada una estrategia de juego.

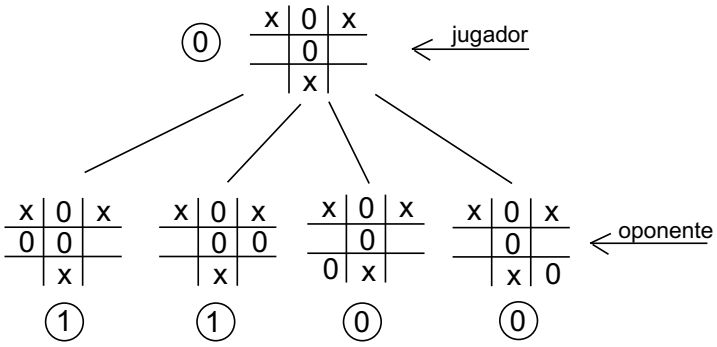
Una forma posible de hacer esta evaluación para el juego de *triqui* es sumar el número de filas, columnas y diagonales que permanecen abiertas para un jugador, y restar de allí el número de éstas que permanecen

abiertas a su oponente (el valor 9 indica triunfo y el valor -9 indica derrota). Entre mayor sea dicho valor, mejor es llegar a ese estado del juego. En la figura 5.15 aparece ilustrada esta función de evaluación para una hoja.

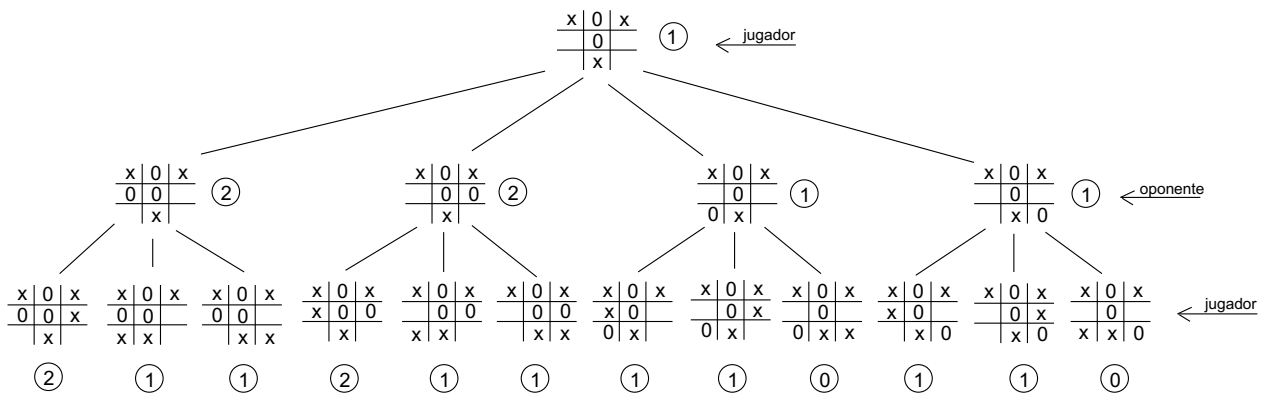
	jugador:	oponente:	calificación:
x 0 x 0 	filas: 1 columnas: 2 diagonales: 0	filas: 2 columnas: 1 diagonales: 0	3 - 3 = 0
x 0 x 0 	filas: 1 columnas: 1 diagonales: 2	filas: 2 columnas: 1 diagonales: 0	4 - 3 = 1
x 0 0 x 	filas: 2 columnas: 1 diagonales: 1	filas: 1 columnas: 1 diagonales: 0	4 - 2 = 2
x 0 0 x	filas: 1 columnas: 1 diagonales: 0	filas: 1 columnas: 1 diagonales: 1	2 - 3 = -1

Fig. 5.15- Función de evaluación para las hojas

Con esta información en las hojas, se debe realizar un proceso para subir ese conocimiento hacia la raíz, de manera que le pueda dar una medida de su calidad, basado en la calificación de sus hijos. La manera de subir esta información se ilustra en la figura 5.16.



Nivel de profundidad 1: Al nivel superior se sube el valor que lleva al jugador a una peor posición. Esto es, le da como valor a ese tablero la peor posición a la que pueda llevarlo el oponente con su mejor respuesta



Nivel de profundidad 2: Al nivel 1 del árbol sube el mayor valor posible de todos los hijos. Esto es, el valor de ese tablero es igual a la mejor respuesta que el jugador pueda dar a su oponente. Al nivel 0 sigue subiendo el mínimo de todos sus hijos, puesto que se supone que el contrario intentará hacer su mejor jugada. Fíjese que la calificación del tablero aumenta, con respecto al nivel de profundidad 1, puesto que se hace un estudio de las mejores respuestas que se le pueden dar al contrincante cuando hace su mejor jugada.

Fig. 5.16- Función de evaluación para un árbol con nivel de análisis 1

Ejercicios Propuestos

5.104. Utilizando las ideas dadas en la sección anterior diseñe el TAD ArbolTriqui para el juego de *triqui*.

5.105. Para el TAD ArbolTriqui diseñe las mejores estructuras de datos posibles y defina claramente el esquema de representación

5.106. 💻 🕒🕒 Implemente las operaciones del TAD ArbolTriqui sobre las estructuras de datos diseñadas en el ejercicio anterior.

5.107. 💻 🕒🕒 Desarrolle un programa que juegue *triqui*, basado en el TAD ArbolTriqui.

Bibliografía

- [AHO83] Aho, A., Hopcroft, J., Ullman, J., "Data Structures and Algorithms", Cap. 3, 4, 5, Addison-Wesley, 1983.
- [BER94] Bergin, J., "Data Abstraction: The Object-Oriented Approach Using C++", Cap. 9, McGraw-Hill, 1994.
- [COM79] Comer, D., "The Ubiquitous B-Tree", ACM Computing Surveys, Vol. 11, No. 2, 1979.
- [ESA89] Esakov, J., Weiss, T., "Data Structures: An Advanced Approack Using C", Cap. 7, Prentice-Hall, 1989.
- [FEL88] Feldman, M., "Data Structures with Modula-2", Cap. 7, Prentice-Hall, 1988.
- [FRE60] Fredkin, E., "Trie Memory", Comm. ACM, Vol. 3, 1960.
- [KNU73] Knuth, D., "The Art of Computer Programming. Vol. 1 - Fundamental Algorithms", 2da edición, Addison-Wesley, 1973.
- [KRU87] Kruse, R., "Data Structures & Program Design", Cap.10, Prentice-Hall, 1987.

- [LIP87] Lipschutz, S., "Estructura de Datos", Cap. 7, McGraw-Hill, 1987.
- [MAR86] Martin, J., "Data Types and Data Structures", Cap. 8, Prentice-Hall, 1986.
- [TEN93] Tenenbaum, A., Langsam, Y., "Estructuras de Datos en C", Cap. 5, Prentice Hall, 1993.
- [TRE76] Tremblay, J., Sorenson, P., "An Introduction to Data Structures with Applications", Cap. 5, McGraw-Hill, 1976.
- [WIR76] Wirth, N., "Algorithms + Data Structures = Programs", Cap. 4, Prentice-Hall,1976.
- [WIR86] Wirth, N., "Algorithms & Data Structures", Cap. 4, Prentice-Hall, 1986.

CAPITULO 6

ESTRUCTURAS NO LINEALES:

GRAFOS DIRIGIDOS

En este capítulo se presentan unas estructuras de datos más generales que los árboles, que permiten modelar relaciones no necesariamente jerárquicas entre elementos de un conjunto. Los grafos se utilizan para representar mapas de rutas, organización de procesos, espacios de búsqueda para juegos, circuitos lógicos, etc.

6.1. Motivación

En 1736, los habitantes de la ciudad de Koenigsberg plantearon a Euler el problema de determinar una manera de recorrer exactamente una vez cada uno de los siete puentes que atraviesan la ciudad, terminando en el mismo punto de partida. Los puentes se encuentran dispuestos sobre el río Pregal como se muestra en la figura 6.1.

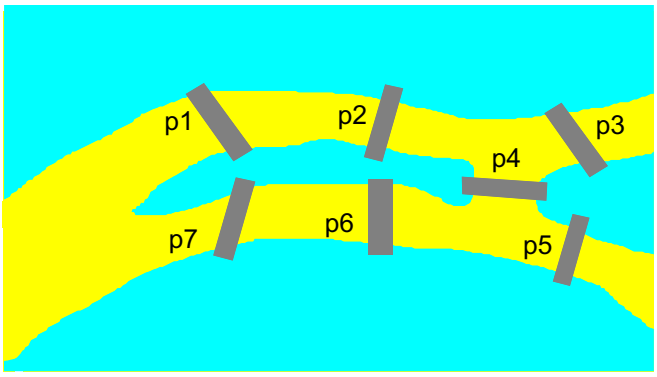


Fig. 6.1 - Problema de los puentes de Koenigsberg

Euler demostró que no era posible encontrar dicha ruta, debido al hecho de que a cada sector llega un número impar de puentes, y modeló el problema utilizando un grafo, en el cual los cuatro elementos del conjunto corresponden a las orillas, y, las relaciones, a los puentes, tal como se sugiere en la figura 6.2.

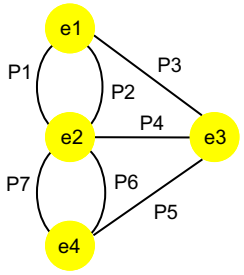


Fig. 6.2 - Modelaje del problema de los puentes de Koenigsberg a través de un grafo

Desde esa época, muchos problemas se plantean y resuelven utilizando un grafo como la estructura de modelaje. El caso típico es el de un camión repartidor, que debe visitar diferentes puntos de la ciudad para dejar su producto. Allí, la ciudad se representa con un grafo como el de la figura 6.3., y el algoritmo que busca la solución se mueve sobre sus componentes y relaciones, teniendo en cuenta el tiempo que toma ir de un lugar a otro.

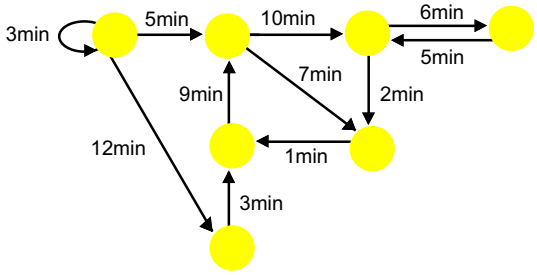


Fig. 6.3 - Problema del camión repartidor en una ciudad

En este capítulo se presenta el TAD Grafo, se estudia la algorítmica para resolver diversos problemas mediante la manipulación de este objeto abstracto, y, por último, se muestran algunas de las estructuras de datos más utilizadas para representar internamente un grafo dirigido.

6.2. Definiciones y Conceptos Básicos

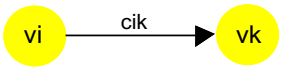
Un **grafo dirigido** es una estructura compuesta por un conjunto de elementos, denominados los **vértices**, y por un conjunto de relaciones entre dichos elementos, denominados los **arcos**. El formalismo escogido para representar un grafo es una pareja ordenada de conjuntos:

$$G = (V, A)$$

donde V es el conjunto de vértices y A el conjunto de arcos. Para efectos prácticos se dice que $v_1, ..., v_n$ son los elementos de V.

$$V = \{ v_1, ..., v_n \}$$

Un arco, por su parte, es una tripleta de la forma (v_i, v_k, c_{ik}), la cual establece una relación entre los vértices v_i y v_k de V, un **sentido** de la relación ($v_i \rightarrow v_k$), y un **valor** o **peso** asociado (c_{ik}). Gráficamente, una tripleta se suele representar de la siguiente forma:



De esta forma, se puede definir el conjunto de arcos como:

$$A = \{ (x, y, c) \mid x, y \in V, x \rightarrow y, \text{ x e y están relacionados con un valor c} \}$$

Se dice que v_k es **sucesor** de v_i, si $(v_i, v_k, c) \in A$ para algún c. En este mismo caso se dice que v_i es **predecesor** de v_k. Una representación gráfica de estos conceptos se da en la figura 6.4.

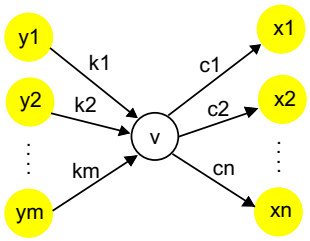


Fig. 6.4 - Sucesores y predecesores del vértice v

Cada vértice v_i del grafo tiene asociados dos valores: uno, corresponde a un identificador que lo distingue como elemento único de V, y, el otro, es alguna información asociada con el elemento. Como parte de la notación se utiliza v_i para hablar indistintamente del vértice y de su identificador, e $info(v_i)$ para referirse a la información asociada con el vértice v_i.

En un grafo dirigido existe la restricción de que no puede haber más de un arco entre cualquier par de vértices, en cada uno de los sentidos. En la figura 6.5 se ilustra un caso válido y otro inválido.

(a) Situación válida (b) Situación inválida

Fig. 6.5 - Dos arcos entre dos vértices

El **orden** de un grafo corresponde al número de sus elementos, es decir, a la cardinalidad del conjunto de vértices. Cuando un grafo es de orden cero, el grafo es **vacío** y se representa mediante la pareja (,). Un vértice de un grafo es una **fuente** si no tiene ningún predecesor. En ese caso, no existe en el conjunto de arcos ninguna tripleta cuyo segundo elemento sea dicho vértice. Un vértice de un grafo es un **sumidero**, si no tiene ningún sucesor en el grafo. Formalmente se tiene que:

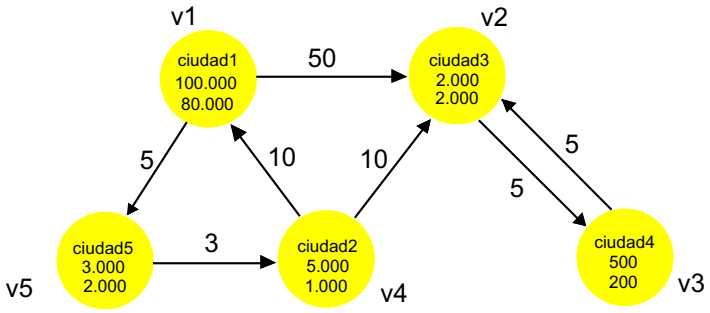
- $fuente(v) \underline{ssi} (w, v, c) \notin A, \forall w, c$
- $sumidero(v) \underline{ssi} (v, w, c) \notin A, \forall w, c$

Ejemplo 6.1:

Suponga que se quiere modelar una red de distribución de agua entre N ciudades de una región. Cada ciudad tiene un nombre, una capacidad máxima de almacenamiento y un estado actual. Cada tubo tiene una capacidad de transporte por minuto. El grafo G que modela esta situación es:

- $V = \{ v_1, ..., v_N \mid v_i \text{ es una ciudad} \}$
- $A = \{ (x, y, c) \mid \text{hay un tubo entre x e y, con capacidad de transporte por minuto c} \}$
- $info(v_i) = [nombre_i, capacidad_i, actual_i]$

Un ejemplo gráfico de uno de estos grafos es el siguiente:



Para dicho grafo se tiene la siguiente información asociada:

- V = { v_1, v_2, v_3, v_4, v_5 }
- A = { (v_1, v_2, 50), (v_2, v_3, 5), (v_3, v_2, 5), (v_4, v_2, 10), (v_4, v_1, 10), (v_5, v_4, 3), (v_1, v_5, 5) }
- info(v_1) = ["ciudad1", 100.000, 80.000]
- info(v_2) = ["ciudad3", 2.000, 2.000]
- info(v_3) = ["ciudad4", 500, 200]
- info(v_4) = ["ciudad2", 5.000, 1.000]
- info(v_5) = ["ciudad5", 3.000, 2.000]
- El orden del grafo es 5 y no tiene fuentes ni sumideros
- Los sucesores de v_1 son v_2 y v_5, y su único predecesor es v_4.

⏎

Un **camino**, entre un vértice v_1 y un vértice v_2 de un grafo G, es una secuencia de vértices < x_1, ..., x_n >, con las siguientes características:

- $x_i \in V, 1 \leq i \leq n$
- $x_1 = v_1, x_n = v_2$
- $x_i \rightarrow x_{i+1}, 1 \leq i \leq n\text{-}1$

Esto es, comienza en el vértice v_1 (**origen**), termina en el vértice v_2 (**destino**), y de cada elemento de la secuencia sale un arco hacia el siguiente. Se dice, entonces, que el camino **pasa** por los vértices x_2, ..., x_{n-1} y que su **longitud** es n-1. De acuerdo con esta definición, siempre existe un camino de longitud 0 que parte de cualquier vértice y termina en él mismo. Del mismo modo, pueden existir caminos infinitos, que repiten un ciclo indefinidamente.

Un camino es **simple** si todos los vértices por los cuales pasa son diferentes entre sí y diferentes del origen y del destino. Debe ser claro que un camino simple puede comenzar y terminar en el mismo vértice, pero no puede pasar dos veces por un mismo nodo.

- $\text{simple}(< x_1, ..., x_n >) \underline{\text{ssi}} \ (x_i \neq x_k, i \neq k, 1 \leq i \leq n\text{-}1, 1 \leq k \leq n\text{-}1) \wedge (x_i \neq x_k, i \neq k, 2 \leq i \leq n, 2 \leq k \leq n)$

Es importante resaltar la diferencia que existe entre el camino < v, v > de longitud 1, que parte y termina en el vértice v, y el camino < v > de longitud 0, sobre ese mismo elemento. Para que exista el primero, debe haber en el conjunto de arcos una tripleta (v, v, c), para algún valor de c, mientras que el segundo existe para cualquier vértice. En la figura 6.6. se muestra un vértice con ambos caminos.

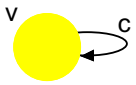


Fig. 6.6 - Vértice con caminos < v, v > y < v >

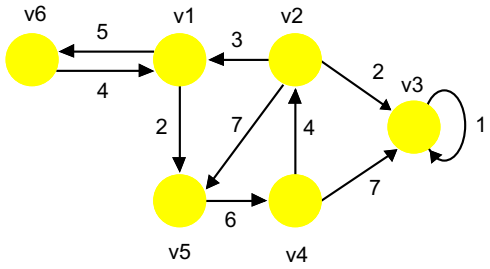
Se define el **costo** de una camino C, como la suma de los valores asociados con los arcos que lo componen. Sólo cuando todos los arcos tienen como valor asociado 1, el costo y la longitud de un camino coinciden.

- $\text{costo}\ (< x1, .., xn > \mid x1 \xrightarrow{c1} x2 \ldots \xrightarrow{cn-1} xn) = c1 + c2 + \ldots + cn - 1$
- $\text{longitud}\ (< x1, .., xn > \mid x1 \xrightarrow{c1} x2 \ldots \xrightarrow{cn-1} xn) = n - 1$

Un **ciclo simple** es un camino simple, de longitud mayor o igual a 1, que comienza y termina en el mismo vértice. En general, se define un **ciclo** como un camino de longitud diferente de 0, cuyo origen y destino son iguales. Se dice que un grafo es **acíclico** si no contiene ciclos.

Ejemplo 6.2:

Para el grafo de la figura:



Se tienen los siguientes ejemplos de caminos:

	camino	costo	longitud
• Caminos no simples	< v_1, v_6, v_1, v_5, v_4, v_3 >	24	5
	< v_3, v_3, v_3 >	2	2
• Caminos simples	< v_3 >	0	0
	< v_3, v_3 >	1	1
	< v_2, v_5, v_4, v_3 >	20	3
• Ciclos simples	< v_1, v_6, v_1 >	9	2
	< v_2, v_1, v_5, v_4, v_2 >	15	4
• Ciclos no simples	< v_3, v_3, v_3 >	2	2
	< v_2, v_1, v_6, v_1, v_5, v_4, v_2 >	24	6
• Camino más corto de v1 a v3	< v_1, v_5, v_4, v_3 >	15	3
• Camino más barato de v1 a v3	< v_1, v_5, v_4, v_2, v_3 >	14	4
• Camino simple más largo	< v_6, v_1, v_5, v_4, v_2, v_3 >	18	5

Se define un **camino hamiltoniano** como un camino que pasa exactamente una vez por cada uno de los vértices de un grafo. Este camino no siempre existe, y tampoco es único. De la misma forma, se define un **ciclo hamiltoniano** como un ciclo que pasa exactamente una vez por cada uno de los vértices de un grafo.

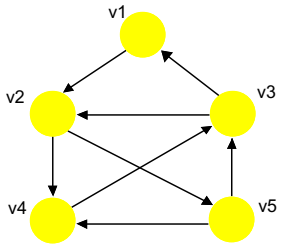
- $hamilton(< x_1, ..., x_n >)$ ssi $orden(g) = n \wedge x_i \neq x_k, i \neq k \wedge x_i \rightarrow x_{i+1}, 1 \leq i \leq n-1$
- $cicloHamilton(< x_1, ..., x_n >)$ ssi $hamilton(< x_1, ..., x_{n-1} >) \wedge x_1 = x_n$

Se define un **camino de euler** (o camino euleriano) como un camino que pasa exactamente una vez por cada uno de los arcos de un grafo. Así mismo, se define un **ciclo de euler** (o ciclo euleriano) como un ciclo que pasa exactamente una vez por cada uno de los arcos de un grafo.

- $euler(< x_1, ..., x_n >)$ ssi $card(A) = n-1 \wedge x_i \rightarrow x_{i+1}, 1 \leq i \leq n-1 \wedge si\ (v, w, c) \in A \Rightarrow \exists!\ i \mid x_i = v, x_{i+1} = w$
- $cicloEuler(< x_1, ..., x_n >)$ ssi $euler(< x_1, ..., x_n >) \wedge x_1 = x_n$

Ejemplo 6.3:

Para el grafo de la figura:



Se tienen los siguientes resultados:

- camino de euler: $< v_5, v_3, v_2, v_5, v_4, v_3, v_1, v_2, v_4 >$
- camino hamiltoniano: $< v_5, v_3, v_1, v_2, v_4 >$
- ciclo hamiltoniano: $< v_1, v_2, v_5, v_4, v_3, v_1 >$

⏎

Dos vértices v, w de un grafo G son **adyacentes** si existe en el grafo por lo menos un arco entre los dos, sin importar el sentido. Una **cadena** es una secuencia $< x_1, ..., x_n >$ de vértices de V, tal que cualesquiera dos vértices consecutivos son adyacentes. Para el ejemplo anterior una cadena posible es $< v_1, v_2, v_3, v_4, v_5 >$.

- $adyacente(v, w)$ ssi $v \rightarrow w \vee w \rightarrow v$
- $cadena(< x_1, ..., x_n >)$ ssi $adyacente(x_i, x_{i+1}), 1 \leq i \leq n-1$

Un grafo G es **completo** si dos vértices diferentes cualesquiera del conjunto V son adyacentes. En la figura 6.7 aparece un ejemplo de un grafo dirigido completo.

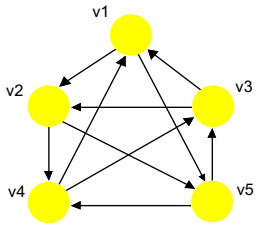


Fig. 6.7 - Grafo completo

Un grafo es **conexo** si para cualquier par de vértices v, w del conjunto V, existe una cadena que los une. Un grafo es **fuertemente conexo** si para cualquier par de vértices v, w del conjunto V, existe un camino que vaya de v a w. En la figura 6.8 aparecen algunos ejemplos.

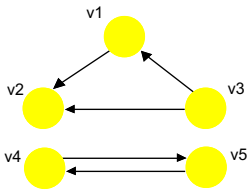
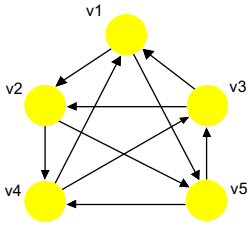


Fig. 6.8 - (a) Grafo no conexo



b) Grafo fuertemente conexo

Un grafo es **planar** si es posible dibujarlo en un plano sin que se crucen los arcos. Un problema típico en grafos planares es el de las 3 casas y los 3 servicios. Se trata de establecer si es posible llevar las tuberías de agua, gas y electricidad, sobre un mismo plano, a 3 casas vecinas. El problema se reduce a determinar si el grafo que representa la situación es planar. Cada casa y cada servicio están representados por un vértice y cada tubería por un arco. En este caso, se pueden dibujar sin cruces 8 de los arcos, como se muestra en la figura 6.9, pero está demostrado que es imposible colocar los 9.

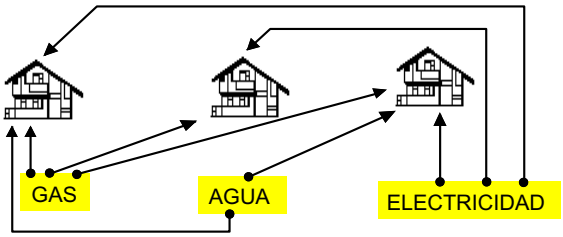


Fig. 6.9 - Problema de las 3 casas y los 3 servicios

Dos grafos G1 y G2 son **isoformos** si tienen la misma estructura, aunque tengan diferentes contenidos y/o identificadores en los vértices. En la figura 6.10 se muestra un ejemplo de grafos isoformos.

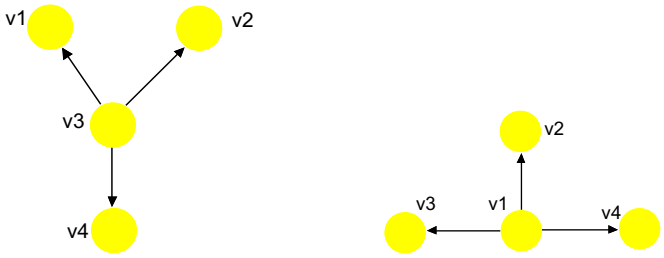


Fig. 6.10 - Grafos isomorfos

- isomorfos(g1, g2) ssi $g_1 = (V_1, A_1), g_2 = (V_2, A_2) \wedge$

  $f_1, f_2 \mid f_1: V_1 \rightarrow V_2$, biyectiva, $f_2: A_1 \rightarrow A_2$, biyectiva $\mid$

  $v, w \in V_1$, si $(v, w) \in A_1 \Rightarrow f_2((v, w)) = (f_1(v), f_1(w)) \in A_2$

6.3. El TAD Grafo

El TAD Grafo que se presenta en esta sección sólo está parametrizado por el tipo de información que se almacena en los vértices, puesto que se decidió, por simplicidad en el planteamiento, que los identificadores

de los vértices fueran valores enteros consecutivos entre 1 y el orden del grafo, y que el costo de un arco fuera un valor entero no negativo. Otra posibilidad habría sido adicionar como parámetros del TAD los tipos Vértice y Costo, de tal forma que no quedaran restringidos al tipo entero, como sucede en este caso. Aprovechando esta simplificación, algunas de las operaciones del TAD retornan el valor -1 para señalar casos especiales.

TAD Grafo[TipoG]

(V, A) → V = { v_1, ... ,v_n }, A = { (v, w, c) }

{ inv: ∀ (v, w, c) ∈ A ⇒ v, w ∈ V ∧ v → w con costo c ≥ 0,
si (v, w, c1) ∈ A ∧ (v, w, c2) ∈ A ⇒ c1 = c2 }

Constructoras:

• inicGrafo:		→ Grafo

Modificadoras:

• insVertice:	Grafo X TipoG	→ Grafo
• insArco:	Grafo X int X int X int	→ Grafo
• elimArco:	Grafo X int X int	→ Grafo

Analizadoras:

• costoArco:	Grafo X int X int	→ int
• sucesores:	Grafo X int	→ Lista[int]
• infoVertice:	Grafo X int	→ TipoG
• ordenGrafo:	Grafo	→ int

Destructora:

• destruirGrafo:	Grafo

Persistencia:

• cargarGrafo:	FILE *	→ Grafo
• salvarGrafo:	Grafo X FILE *	

```
Grafo inicGrafo( void )
/* Crea y retorna un grafo vacío */

{ post: inicGrafo = ( ,  ) }
```

```
void insVertice( Grafo g, TipoG elem )
/* Agrega un vértice al grafo con la información elem asociada */

{ pre: g = ( { v1, ... , vn }, A ) }
{ post: g = ( { v1, ... , vn, vn+1 }, A ), info( vn+1 ) = elem }
```

```
void insArco( Grafo g, int x1, int x2, int c )
/* Agrega al grafo el arco x1  x2 con costo c */

{ pre: V = { v1, ... , vn }, x1 = vi, x2 = vk, ¬∃k | ( x1, x2, k ) ∈ A, c ≥ 0 }
{ post: g = ( V, A U ( x1, x2, c ) }
```

```
void elimArco( Grafo g, int x1, int x2 )
```

```
/* Elimina del grafo el arco x1  x2 */

{ pre: ( x1, x2, c )  A }
{ post: g = ( V, A - ( x1, x2, c ) ) }
```

```
int costoArco( Grafo g, int x1, int x2 )
/* Retorna el costo del arco x1  x2 si éste existe. En caso contrario retorna -1 */

{ post: ( ( x1, x2, c )  A , costoArco = c )  ( ¬∃c | ( x1, x2, c ) ∈ A , costoArco = -1 ) }
```

```
Lista sucesores( Grafo g, int v )
/* Retorna una lista con los identificadores de los vértices sucesores de v */

{ pre: v  V }
{ post: sucesores = < x1, ... , xk > | v  xi }
```

```
TipoG infoVertice( Grafo g, int v )
/* Retorna la información asociada con el vértice v */

{ pre: v  V }
{ post: infoVertice = info( v ) }
```

```
int ordenGrafo( Grafo g )
/* Retorna el número de vértices del grafo */

{ post: ordenGrafo = n }
```

```
void destruirGrafo( Grafo g )
/* Destruye el objeto abstracto, retornando toda la memoria ocupada por éste */

{ post: el grafo g no tiene memoria reservada }
```

```
Grafo cargarGrafo( FILE *fp )
/* Construye un grafo a partir de la información de un archivo */

{ pre: el archivo está abierto y es estructuralmente correcto, de acuerdo con el esquema de persistencia }
{ post: se ha construido el grafo que corresponde a la imagen de la información del archivo }
```

```
void salvarGrafo( Grafo g, FILE *fp )
/* Salva el grafo en un archivo */

{ pre: el archivo está abierto }
{ post: se ha hecho persistir el grafo en el archivo, la ventana de la lista está indefinida }
```

Con las operaciones especificadas anteriormente, es posible desarrollar cualquier algoritmo de manejo de grafos. Se omitió intencionalmente la operación que elimina un vértice del grafo, por simplicidad en el planteamiento y posterior implementación del TAD.

La razón por la cual se va a extender el conjunto de operaciones del TAD es que muchos de los algoritmos de manejo de grafos se implementan recursivamente, de una forma muy natural, y surge, en ese caso, un problema de control de la recursión, debido a que los ciclos de un grafo pueden generar llamadas recursivas infinitas. Por ejemplo, si se busca un camino entre dos puntos, es necesario recordar de alguna manera los puntos por los cuales ya ha pasado el proceso de búsqueda, para evitar hacer de nuevo una llamada recursiva desde ese vértice. A la acción de colocar una señal en un vértice, para indicar que ya se ha visitado, se le denomina **marcar**.

Aunque no es indispensable que el TAD Grafo maneje las marcas, se van a adicionar en este caso las operaciones modificadoras y analizadoras necesarias, de manera que los algoritmos que trabajan sobre grafos se puedan implementar de una forma más clara. La otra aproximación posible es manejar alguna estructura de datos auxiliar donde se van colocando los vértices marcados.

El formalismo extendido para incluir la noción de vértice marcado agrega un nuevo conjunto de vértices llamado V', subconjunto de V, que incluye todos los vértices con marca:

$$G = (V, A, V'),\ V' \subseteq V,\ v \in V' \text{ si } v \text{ está marcado}$$

Gráficamente se va a denotar una marca con el vértice sombreado, como se muestra en la figura 6.11.

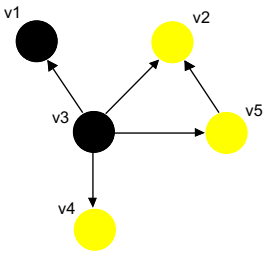


Fig. 6.11 - Formalismo para indicar vértices marcados

Las operaciones que se agregan al TAD Grafo para manejar las marcas son las siguientes:

{ inv: $V' \subseteq V$ }		
Modificadoras:		
• marcarVertice:	Grafo X int	→ Grafo
• desmarcarVertice:	Grafo X int	→ Grafo
• desmarcarGrafo:	Grafo	→ Grafo
Analizadoras:		
• marcadoVertice:	Grafo X int	→ int

```
void marcarVertice( Grafo g, int v )
/* Marca el vértice v */

{ pre: v ∈ V, V' = V'ini }
{ post: V' = V'ini + v }
```

```
void desmarcarVertice( Grafo g, int v )
/* Desmarca el vértice v */

{ pre: v  V', V' = V'ini }
{ post: V' = V'ini - v }
```

```
void desmarcarGrafo( Grafo g )
/* Desmarca todos los vértices del grafo */

{ post: V' =  }
```

```
int marcadoVertice( Grafo g, int v )
/* Informa si el vértice v está marcado */

{ pre: v  V }
{ post: marcadoVertice = ( v  V' ) }
```

6.4. Caminos en un Grafo

En esta sección se hace una presentación incremental del problema de buscar caminos en un grafo dirigido, utilizando un planteamiento recursivo. Se comienza por el problema de determinar si hay un camino entre dos puntos de un grafo acíclico y se llega hasta el problema de calcular el camino de costo mínimo entre dos vértices. La presentación se hace por medio de ejemplos, en los cuales se muestra el planteamiento de la solución y se explica el algoritmo.

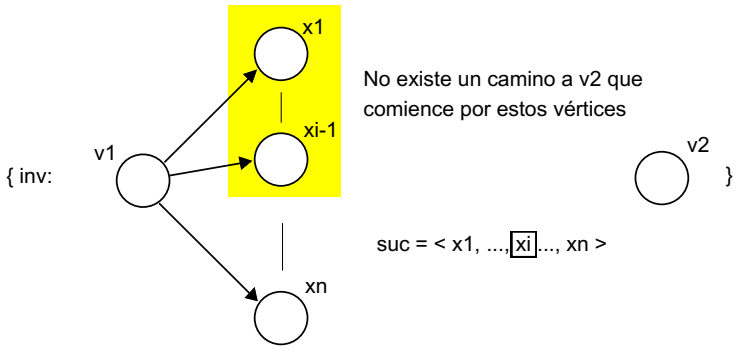
Ejemplo 6.4:

Determinar si existe un camino entre dos vértices v1 y v2 de un grafo acíclico. Puesto que el grafo no tiene ciclos, el algoritmo de solución puede trabajar sobre el grafo como si fuera un árbol n-ario, porque no hay riesgo de quedarse haciendo llamadas recursivas.

```
/* pre: g = ( V, A, V' ), g es acíclico, v1, v2  V */
/* post: hayCamino1 = existe un camino que va de v1 a v2 */

int hayCamino1( Grafo g, int v1, int v2 )
{   Lista suc;
    if ( v1 == v2 || costoArco( g, v1, v2 ) != -1 )
        return TRUE;
    else
    {   suc = sucesores( g, v1 );
        for ( primLista( suc ); !finLista( suc ); elimLista( suc ) )
            if ( hayCamino1( g, infoLista( suc ), v2 ) )
            {   destruirLista( suc );
                return TRUE;
            }
        return FALSE;
    }
}
```

La primera salida de la recursión corresponde al caso en el cual el vértice de origen v1 y el vértice de destino v2 coinciden, puesto que por definición allí existe un camino de longitud 0. El segundo caso de salida es cuando existe un arco que los conecta (v1 → v2). En el ciclo se intenta encontrar un camino que parta de alguno de los sucesores de v1 y termine en v2. Termina si encuentra dicho camino, o si se ha intentado con todos los sucesores y no ha encontrado una respuesta. El planteamiento recursivo en este caso es muy simple y se puede resumir de la siguiente manera:

- hayCamino1(v1, v2) ssi (v1 = v2) ∨ (v1 → v2) ∨ (∃w ∈ V | v1 → w ∧ hayCamino1(w, v2))



Considerando que el grafo del problema es acíclico (lejos del peor de los casos de un grafo), se puede afirmar que la complejidad de esta rutina es O(n), donde n es el orden del grafo, puesto que en ningún caso va a pasar más de una vez por cada uno de los vértices, independiente del número de arcos que los relacionen.

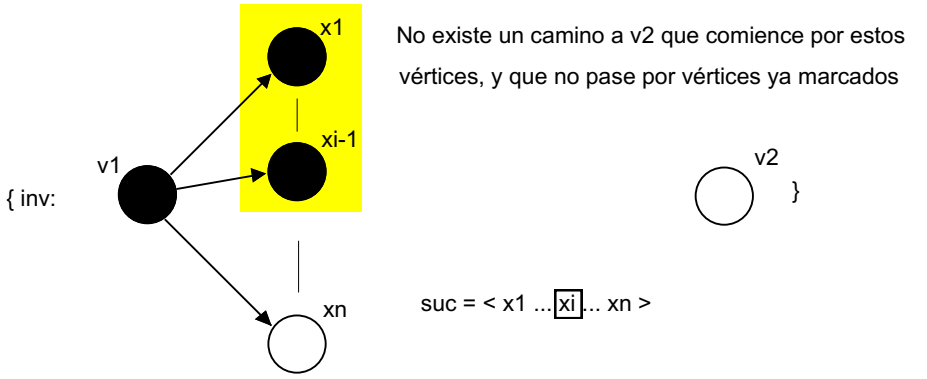
⏎

Ejemplo 6.5:

En este ejemplo se elimina la restricción de que el grafo sea acíclico. La estructura del algoritmo utilizado en el ejemplo anterior se conserva, pero se deben agregar las marcas respectivas en los vértices por los cuales se va pasando, para evitar que se hagan llamadas recursivas infinitas al entrar en un ciclo del grafo.

```
/* pre:  g = ( V, A, V' ), v1, v2  V, V' tiene los vértices a partir de los cuales se ha hecho una llamada recursiva */
/* post:  hayCamino2 = existe un camino de v1 a v2 */

int hayCamino2( Grafo g, int v1, int v2 )
{   Lista suc;
    if ( v1 == v2 || costoArco( g, v1, v2 ) != -1 )
        return TRUE;
    else
    {   marcarVertice( g, v1 );
        suc = sucesores( g, v1 );
        for ( primLista( suc ); !finLista( suc ); elimLista( suc ) )
            if ( !marcadoVertice( g, infoLista( suc ) ) && hayCamino2( g, infoLista( suc ), v2 ) )
            {   destruirLista( suc );
                return TRUE;
            }
        return FALSE;
    }
}
```

La única diferencia con el algoritmo del ejemplo anterior, es que antes de hacer las llamadas recursivas buscando un camino desde los sucesores de v1 hasta v2, verifica que éstos no estén marcados. Además, marca v1 para evitar que un camino desde un sucesor intente utilizarlo como parte de la solución. El invariante del ciclo es el siguiente:



Dado que el mecanismo de marca de vértices que se utiliza en la rutina evita que se haga más de una llamada recursiva desde cualquier punto, se puede concluir que su complejidad es O(n), donde n es el orden del grafo.

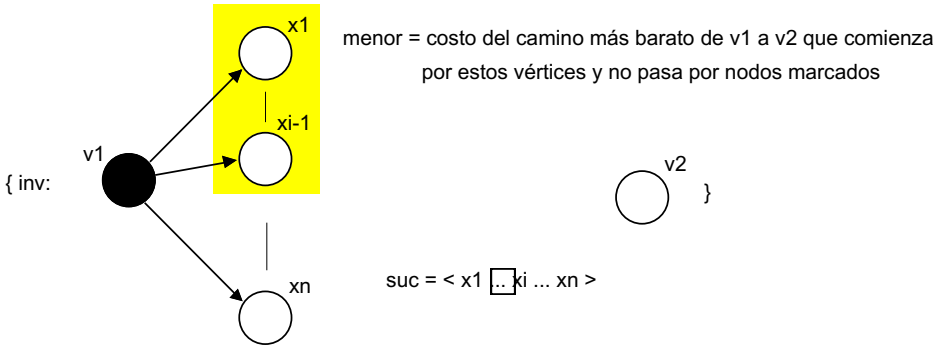
⏎

Ejemplo 6.6:

Calcular y retornar el costo del camino mínimo entre dos vértices v1 y v2 de un grafo dirigido, teniendo en cuenta que el grafo puede tener ciclos. Si el camino no existe debe retornar -1. El algoritmo conserva las líneas generales del ejemplo anterior, pero debe buscar todos los caminos posibles entre los dos vértices antes de retornar un valor.

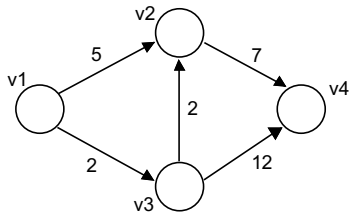
```
/* pre: g = ( V, A, V' ), v1, v2  V, V' tiene los marcados */
/* post: ( hayCamino3 = costo del camino mínimo de v1 a v2 ) || ( hayCamino3 = -1 si no existe camino ) */

int hayCamino3( Grafo g, int v1, int v2 )
{   int temp, menor;
    Lista suc;
    if ( v1 == v2 )
        return 0;
    else
    {   suc = sucesores( g, v1 );
        marcarVertice( g, v1 );
        for ( primLista( suc ), menor = -1; !finLista( suc ); elimLista( suc ) )
            if ( !marcadoVertice( g, infoLista( suc ) ) && ( temp = hayCamino3( g, infoLista( suc ), v2 ) ) != -1 )
                if( menor == -1 )
                    menor = temp + costoArco( g, v1, infoLista( suc ) );
                else
                    menor = min( menor, temp + costoArco( g, v1, infoLista( suc ) ) );
        desmarcarVertice( g, v1 );
        return menor;
    }
}
```

La única salida de la recursión corresponde al caso en el cual los vértices v1 y v2 son iguales. El otro caso trivial de los ejemplos anteriores se debe descartar, porque la existencia de un arco entre v1 y v2 no garantiza que éste sea el camino más corto que los une. En el avance de la recursión se debe manejar una variable temporal que vaya llevando el menor camino hasta el punto que se ha analizado de la lista de sucesores, como se muestra en el invariante:

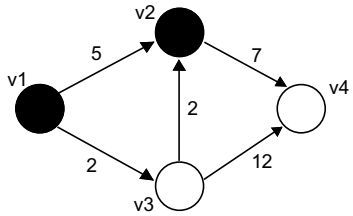


Dos modificaciones se deben resaltar con respecto a los ejemplos anteriores: la primera, que es necesario encontrar el costo de todos los caminos de v1 a v2 para poder seleccionar el menor. Esto obliga a hacer una llamada recursiva desde cada sucesor no marcado, y, según el costo que retorne cada llamada, seleccionar el mejor.

El segundo cambio consiste en la necesidad de desmarcar el nodo v1 antes de abandonar la función. En los casos anteriores, esto no era necesario puesto que sólo se quería determinar la existencia de un camino. Ahora, es indispensable desmarcarlos porque se puede dar el caso de que el camino buscado pase por el nodo v1 pero viniendo de otra llamada recursiva. Por ejemplo, considere el grafo de la figura, y suponga que se está buscando un camino de v1 a v4:



El primer camino que el algoritmo encuentra es < v1, v2, v4 > con costo 12, dejando marcados v1 y v2, como se muestra en la siguiente figura:



En el momento de buscar los caminos desde v1 que comiencen por el vértice v3, solo va a encontrar el camino < v1, v3, v4 > con costo 14, ignorando el camino de costo mínimo < v1, v3, v2, v4 > porque v2 estaba marcado. Esto hace que sea necesario desmarcar los nodos en todos los algoritmos que deban identificar todos los posibles caminos entre dos puntos.

La complejidad de esta rutina es O(n^2), en el peor de los casos, donde n es el orden del grafo. La razón de esto, es que el hecho de desmarcar el vértice de partida al salir de la rutina, permite que se hagan hasta n-1 llamadas recursivas desde dicho punto.

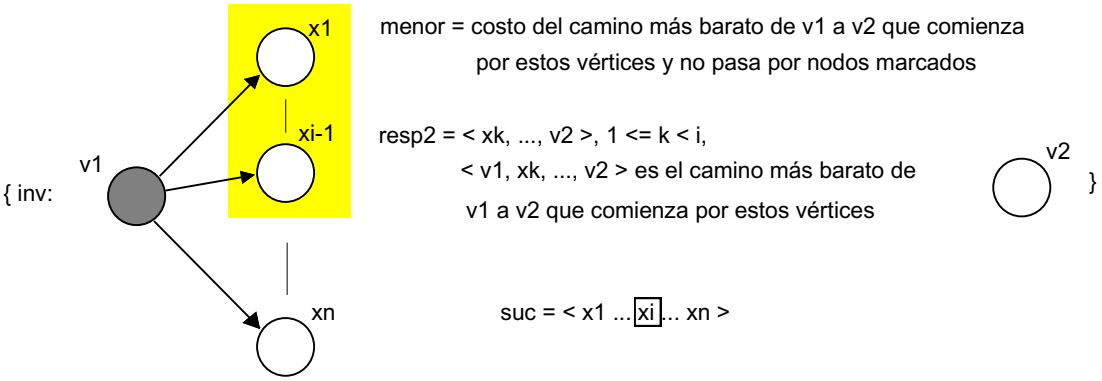
⏎

Ejemplo 6.7:

Calcular y retornar el camino de costo mínimo entre los vértices v1 y v2 de un grafo, teniendo en cuenta que el grafo puede tener ciclos. Al algoritmo del ejemplo anterior solo se le debe agregar un mecanismo que le permita almacenar los caminos que va encontrando. Por razones de eficiencia, se agrega un parámetro por referencia, en el cual la función deja el costo del camino retornado.

```
/* pre: g = ( V, A, V' ), v1, v2  V, V' tiene los marcados */
/* post: ( hayCamino4 = < v1, ... , v2 > camino de costo mínimo de v1 a v2, *costo = costo( < v1, ... ,v2 > ) ) v
        ( hayCamino4 = < > , *costo = -1, no hay camino de v1 a v2 ) */

Lista hayCamino4 ( Grafo g, int v1, int v2, int *costo )
{   int menor;
    Lista suc, resp1, resp2 = inicLista();
    if ( v1 == v2 )
    {     *costo = 0;
          insLista( resp2, v1 );
          return resp2;
    }
    else
    {     suc = sucesores( g, v1 );
          marcarVertice( g, v1 );
          for ( menor = -1, primLista( suc ); !finLista( suc ); elimLista( suc ) )
                if ( !marcadoVertice( g, infoLista( suc ) ) )
                {     resp1 = hayCamino4 ( g, infoLista( suc ), v2, costo );
                      if ( *costo != -1 )
                      {     *costo += costoArco( g, v1, infoLista( suc ) );
                            if ( menor == -1 || *costo < menor )
                            {     menor = *costo;
                                  destruirLista( resp2 );
                                  resp2 = resp1;
                            }
                            else
                                  destruirLista( resp1 );
                      }
                }
          desmarcarVertice( g, v1 );
          *costo = menor;
          if( *costo != -1 )
          {     primLista( resp2 );
                insLista( resp2, v1 );
          }
          return resp2;
    }
}
```

En la salida de la recursión se retorna un camino de longitud 0, puesto que el origen y el destino coinciden. En el avance, se pide el camino de costo mínimo que parte de cada uno de los sucesores de v1, y, en resp2, se va llevando el mejor camino encontrado hasta ese momento.



Al terminar el ciclo se debe desmarcar el vértice v1 e insertarlo en la lista de respuesta (si ésta no es vacía), ya que es el primero por el cuál va a pasar el camino resultado.

Por las mismas razones expuestas en el ejemplo anterior, la complejidad de esta rutina es O(n^2), en el peor de los casos, donde n es el orden del grafo.

Ejercicios Propuestos:

Para cada uno de los siguientes ejercicios desarrolle la rutina pedida, y calcule su complejidad.

6.1. int hayCiclo(Grafo g, int v)
/* Indica si existe algún ciclo que parta del vértice v */

6.2. Lista mayorCamino(Grafo g)
/* Retorna en una lista la secuencia de vértices del camino simple más largo del Grafo g, o sea, aquél que pasa por un mayor número de elementos */

6.3. void imprimeCaminos(Grafo g, int v1, int v2)
/* Imprime todos los caminos simples entre los vértices v1 y v2 del Grafo g, de menor a mayor de acuerdo con su costo */

6.4. Lista frontera1(Grafo g, int vert, int longit)
/* Retorna una lista con todos los vértices del grafo a partir de los cuales existe un camino simple de longitud longit que termina en el vértice vert */

6.5. Lista cicloMasLargo(Grafo g)
/* Retorna el ciclo más largo que hay en el Grafo g */

6.6. void caminoHamilton(Grafo g, int v)
/* Imprime, si existe, un camino hamiltoniano que arranque del vértice v */

6.7. void imprimeCiclos(Grafo g)
/* Imprime todos los ciclos simples del Grafo g */

6.8. void cicloEuler(Grafo g)
/* Imprime, si existe, un ciclo euleriano */

6.9. void cicloHamilton(Grafo g)
/* Imprime, si existe, un ciclo hamiltoniano */

6.10. void caminoEuler(Grafo g)
/* Imprime, si existe, un camino euleriano */

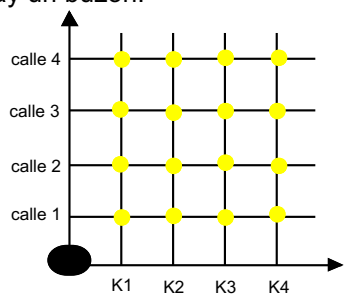
6.11. Lista frontera2(Grafo g, int vert, int longit)
/* Retorna una lista con todos los vértices del grafo a los cuales existe un camino simple de longitud menor o igual a longit que parte del vértice vert */

6.12. int existeMayor(Grafo g, int v1, int v2, int longit)
/* Informa si existe un camino que lleve de v1 a v2 con una longitud mayor que longit */

6.13. int numCaminos(Grafo g, int v1, int v2)
/* Retorna el número de caminos simples diferentes que hay entre v1 y v2 */

6.14. Un problema típico de grafos es la simulación de movimiento sobre su estructura. Suponga que en un grafo hay dos fichas, cada una ocupando un vértice. En un movimiento cada una de las fichas debe avanzar una posición utilizando un arco. El objetivo del juego es llegar a una posición final dada, con la restricción de que en ningún momento las dos fichas pueden ocupar el mismo vértice. Suponiendo que comienza jugando la ficha 1, desarrolle una función que determine el ganador del juego.

6.15. Una ciudad se puede modelar como un conjunto de calles y carreras (ver figura), y suponer que en cada uno de sus cruces hay un buzón.



La dirección de un buzón viene dada por el número de la calle y la carrera. La oficina de correos está situada en la dirección (0, 0). Desarrolle un algoritmo para organizar la distribución de la correspondencia, teniendo en cuenta las siguientes características del problema:

- El cartero comienza la jornada con N cartas, cada una de las cuales tiene una dirección de destino.
- El tiempo que gasta un cartero en ir de un buzón a otro (consecutivos) es de 1 minuto, más 1 segundo por cada carta que transporte, porque su peso lo hace moverse más despacio.

Modele el problema utilizando un grafo, y desarrolle una rutina que retorne la ruta que implique menor tiempo total de repartición.

6.5. Recorrido de Grafos

Muchos de los algoritmos que manejan grafos se basan en la idea de recorrerlos de una manera específica, con el fin de determinar alguna característica o encontrar alguna información especial. **Recorrer** un grafo consiste en pasar exactamente una vez por cada uno de los vértices de la estructura. Existen varias maneras de realizar este proceso, dependiendo del objetivo que se tenga en mente.

6.5.1. Recorrido Plano sobre el Conjunto de Vértices

La forma más sencilla de recorrer un grafo es visitar sus vértices haciendo un barrido secuencial de los elementos del conjunto V. De acuerdo con la simplificación hecha al TAD, este conjunto está constituido por los valores 1...n, donde n es el orden del grafo. De esta forma es posible localizar todos los vértices que cumplan alguna propiedad, pero sin tener en cuenta la topología del grafo (ignorando los arcos).

Ejemplo 6.8:

Retornar una lista con todos los sumideros de un grafo dirigido.

```
Lista sumideros( Grafo g )
{   int i, numVert = ordenGrafo( g );
    Lista lst, resp = inicLista( );
    for ( i = 1; i <= numVert; i++ )
    {     if ( longLista( lst = sucesores( g, i ) ) == 0 )
                anxLista( resp, i );
          destruirLista( lst );
    }
    return resp;
}
```

La rutina se reduce a evaluar la propiedad deseada en cada vértice del grafo, y agregarlo a la lista de respuesta si la cumple. Este tipo de recorrido es de complejidad O(n), donde n es el orden del grafo, si la propiedad que se quiere establecer en el vértice no tiene una complejidad mayor.

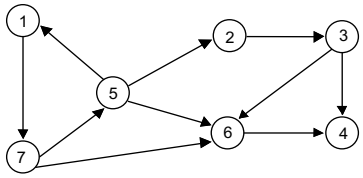
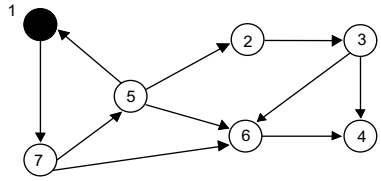
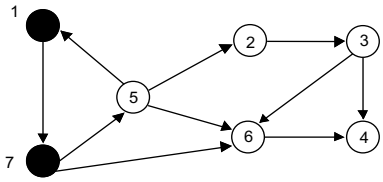
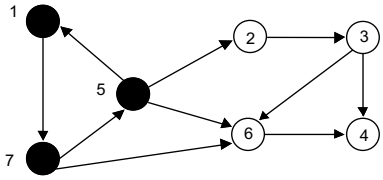
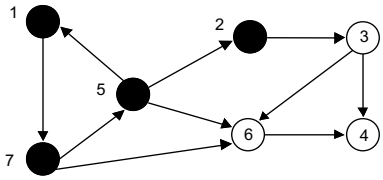
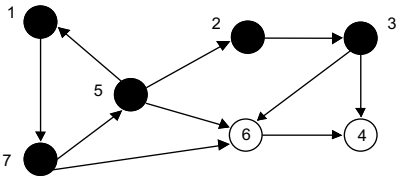
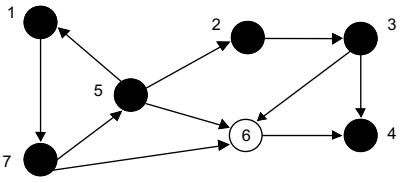
6.5.2. Recorrido en Profundidad

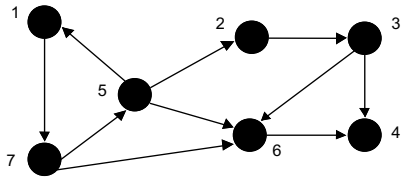
Otros problemas sobre grafos implican un recorrido sistemático basado en las relaciones de los elementos del grafo. En ese caso el avance no se hace sobre el orden de los vértices, sino siguiendo la relación definida por los arcos. Existen dos formas principales de hacer este recorrido, las cuales se ilustran en esta sección y en la siguiente. La primera forma se denomina recorrido **en profundidad** y es equivalente a un recorrido en preorden de un árbol n-ario, verificando en todo momento que el algoritmo no se quede en ciclos. La segunda manera de recorrer un grafo es **por niveles**, similar al proceso correspondiente en un árbol.

El algoritmo de recorrido en profundidad de un grafo consta de dos rutinas: la primera hace un recorrido en profundidad a partir de un vértice dado, marcando los puntos por los cuales va pasando. La segunda rutina busca vértices sin marcar y lanza el proceso antes mencionado, terminando cuando todos los vértices hayan sido visitados. El proceso se ilustra en el siguiente ejemplo:

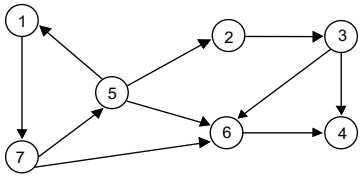
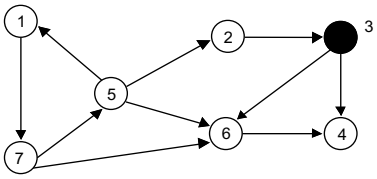
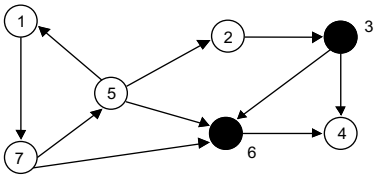
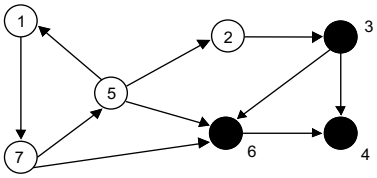
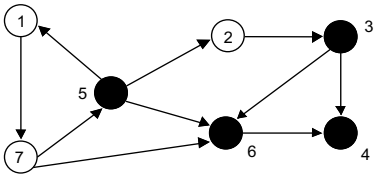
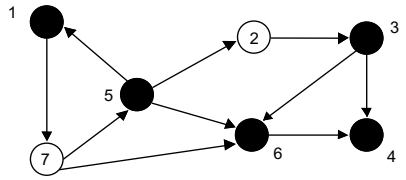
Ejemplo 6.9:

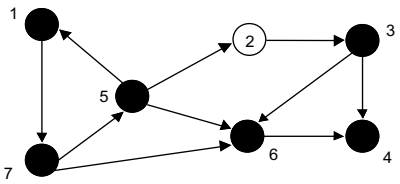
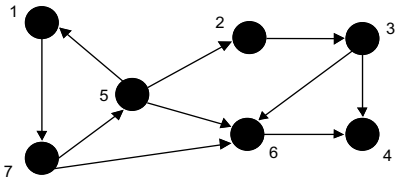
Considere el grafo dirigido de la figura, y suponga que se va a hacer un recorrido en profundidad partiendo del vértice 1.

	El primer paso es visitar el vértice 1, marcar dicho vértice, localizar sus sucesores, y hacer una llamada recursiva sobre cada uno de ellos, verificando que no estén marcados. Recorrido: 1
	Se debe hacer un recorrido en profundidad partiendo del vértice 7 (el único sucesor de 1), para lo cual se repite el mismo proceso del paso anterior. Recorrido: 1, 7
	Se debe hacer el recorrido en profundidad a partir del vértice 5, y, al terminar, comenzar el mismo proceso a partir del vértice 6. Recorrido: 1, 7, 5 Pendiente recorrido desde: 6
	Se repite recursivamente el proceso para los vértices 2 y 6 (sucesores de 5) Recorrido: 1, 7, 5, 2 Pendiente recorrido desde: 6, 6
	Se recorre en profundidad el grafo a partir del vértice 3 (sucesor de 2) Recorrido: 1, 7, 5, 2, 3 Pendiente recorrido desde: 6, 6
	Se repite recursivamente el proceso para los vértices 4 y 6 (sucesores de 3) Recorrido: 1, 7, 5, 2, 3, 4 Pendiente recorrido desde: 6, 6, 6
	Puesto que el vértice 4 no tiene sucesores, se hace el recorrido en profundidad desde el elemento 6 (el último que quedó pendiente en el proceso) Recorrido: 1, 7, 5, 2, 3, 4, 6 Pendiente recorrido desde: 6, 6

	Ya que todos los recorridos pendientes comienzan en vértices marcados (el 6), se da por terminado el proceso.

Debe ser claro que el orden de visita de los elementos, en el recorrido en profundidad, depende del vértice inicial escogido y del orden en el cual la respectiva operación del TAD retorne los sucesores de un elemento. Por ejemplo, el recorrido en profundidad del grafo anterior, partiendo del vértice 3, se puede resumir en la siguiente secuencia de figuras:

	Recorrido: 3
	Recorrido: 3, 6 Pendiente recorrido desde: 4
	Recorrido: 3, 6, 4
	Puesto que no hay sucesores, ni recorridos pendientes, se escoge cualquier vértice no marcado y se continúa desde allí el proceso. Recorrido: 3, 6, 4, 5
	Recorrido: 3, 6, 4, 5, 1 Pendiente recorrido desde: 2
	Recorrido: 3, 6, 4, 5, 1, 7 Pendiente recorrido desde: 2

	Recorrido: 3, 6, 4, 5, 1, 7, 2
	

Las rutinas que llevan a cabo el proceso antes descrito son las siguientes:

```
/* pre: g = ( V, A, V' ), v ∈ V */
/* post: se ha hecho un recorrido en profundidad a partir del vértice v */

void profundidadGrafo( Grafo g, int v )
{   int i, numVert = ordenGrafo( g );
    desmarcarGrafo( g );
    profundidadVertice( g, v );
    for( i = 1; i <= numVert; i++ )
        if ( !marcadoVertice( g, i ) )
            profundidadVertice( g, i );
}

/* pre: g = ( V, A, V' ), v ∈ V, V' tiene los vértices ya recorridos */
/* post: se ha hecho un recorrido en profundidad a partir del vértice v de los vértices alcanzables desde
        dicho punto y se han marcado a medida que se van visitando */

void profundidadVertice( Grafo g, int v )
{   Lista lst = sucesores( g, v );
    visitar( g, v );
    marcarVertice( g, v );
    for ( primLista(lst); !finLista(lst); elimLista(lst) )
        if ( !marcadoVertice( g, infoLista( lst ) ) )
            profundidadVertice( g, infoLista( lst) );
}
```

Es importante anotar en este punto que todos los algoritmos de búsqueda de caminos que se presentaron en una sección anterior, se basan en un recorrido por profundidad a partir de un nodo, hasta encontrar el vértice de destino.

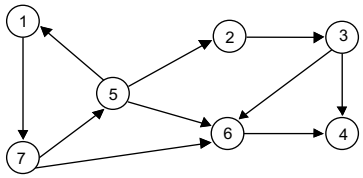
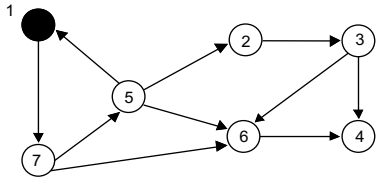
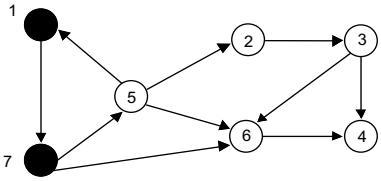
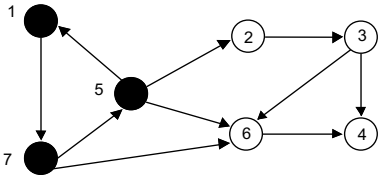
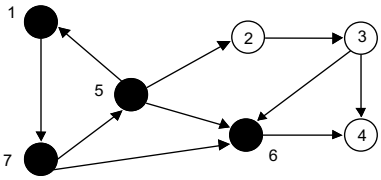
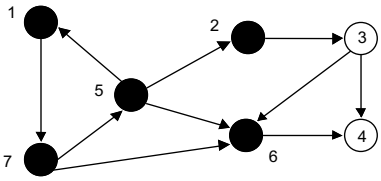
6.5.3. Recorrido por Niveles

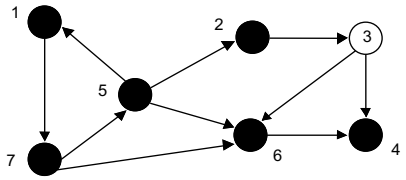
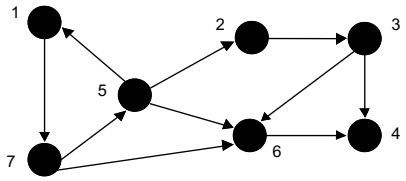
El algoritmo de recorrido por niveles de un grafo se basa también en dos rutinas, con la misma idea del recorrido en profundidad. Desde el punto de vista estructural, es semejante a un recorrido por niveles de un árbol n-ario. Se utiliza como estructura auxiliar una lista en lugar de una cola, para luego poder generalizar

este tipo de recorridos (Ver "Recorridos Heurísticos"). La lista de elementos preparados para ser recorridos se denomina el **frente de exploración**. El siguiente ejemplo ilustra el proceso:

Ejemplo 6.10:

Considere el grafo dirigido de la figura, y suponga que se va a hacer un recorrido por niveles partiendo del vértice 1.

	El primer paso es visitar el vértice 1, marcar dicho vértice, localizar sus sucesores, e incluirlos al final del frente de exploración. Recorrido: 1 Frente de exploración: 7
	Se toma el primer elemento del frente de exploración (7) y se procesa como se hizo en el paso anterior. Recorrido: 1, 7 Frente de exploración: 5, 6
	Recorrido: 1, 7, 5 Frente de exploración: 6, 2, 6
	Recorrido: 1, 7, 5, 6 Frente de exploración: 2, 6, 4
	Recorrido: 1, 7, 5, 6, 2 Frente de exploración: 6, 4, 3
	Recorrido: 1, 7, 5, 6, 2, 4 Frente de exploración: 3

	Recorrido: 1, 7, 5, 6, 2, 4, 3 Frente de exploración:
	

⏎

Las rutinas que implementan el recorrido por niveles de un grafo vienen dadas por el siguiente código:

```
/* pre: g = ( V, A, V' ), v ∈ V */
/* post: se ha hecho un recorrido por niveles a partir del vértice v */

void nivelesGrafo( Grafo g, int v )
{   int i, numVert = ordenGrafo( g );
    desmarcarGrafo( g );
    nivelesVertice( g, v );
    for ( i = 1; i <= numVert; i++ )
        if ( !marcadoVertice( g, i ) )
            nivelesVertice( g, i );
}

/* pre: g = ( V, A, V' ), v ∈ V, V' tiene los vértices ya recorridos */
/* post:  se ha hecho un recorrido por niveles a partir del vértice v de los vértices alcanzables desde
          dicho punto y se han marcado a medida que se van visitando */

void nivelesVertice( Grafo g, int v )
{   int x;
    Lista lst, frente = inicLista( );
    anxLista( frente, v );
    while( longLista( frente ) != 0 )
    {   primLista( frente );
        x = infoLista( frente );
        elimLista( frente );
        if( marcadoVertice( g, x ) )
            continue;
        visitar( g, x );
        marcarVertice( g, x );
        lst = sucesores( g, x );
        for ( ultLista( frente ), primLista( lst ); !finLista( lst ); elimLista( lst ) )
            if ( !marcadoVertice( g, infoLista( lst ) ) )
                anxLista( frente, infoLista( lst ) );
    }
}
```

Ejercicios Propuestos:

Los siguientes ejercicios se resuelven utilizando las ideas planteadas en la sección anterior sobre recorrido de grafos. Algunos de los algoritmos no son triviales y requieren idear una estrategia clara y efectiva para abordar el problema, antes de comenzar a escribir las rutinas.

6.16. Lista predecesores(Grafo g, int v)
/* Retorna la lista de predecesores del vértice v. Esta rutina debería ser una operación analizadora del TAD Grafo */

6.17. Lista fuentesGrafo(Grafo g)
/* Retorna la lista de todas las fuentes del grafo */

6.18. int conexoGrafo(Grafo g)
/* Indica si el grafo g es conexo */

6.19. int completoGrafo(Grafo g)
/* Indica si el grafo g es completo */

6.20. Lista raizGrafo(Grafo g)
/* Cuando a partir de un vértice de un grafo dirigido es posible encontrar un camino a todos los demás vértices, dicho vértice se denomina una **raíz** del grafo. Esta función retorna una lista de vértices con las raíces del grafo g */

6.21. int arbolN(Grafo g)
/* Un **árbol n-ario** se puede ver como un caso particular de un grafo dirigido, que cumple ciertas propiedades con respecto al número de arcos que llegan a cada uno de los elementos. Esta función informa si el grafo g tiene la estructura de un árbol n-ario */

6.22. int excentricidadGrafo(Grafo g, vértice v)
/* Dado el vértice v de un grafo g, se define la **excentricidad** de dicho vértice como:

$$\text{excentricidad}(v) = \max(\text{longitud}(\text{hayCamino3}(g, v, w))), \forall w \in V$$

Esta función retorna la excentricidad de un vértice */

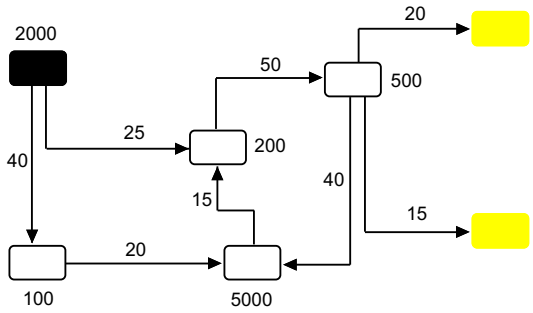
6.23. int centroGrafo(Grafo g)
/* El **centro** de un grafo es el vértice cuya excentricidad es mínima. Intuitivamente, se puede ver como el vértice más cercano de su vértice más lejano. Esta función retorna el centro del grafo g */

6.24. int isomorfosGrafo(Grafo g1, Grafo g2)
/* Informa si los grafos g1 y g2 son isomorfos */

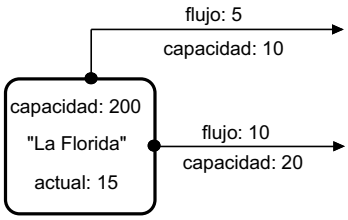
6.25. int numArcosGrafo(Grafo g)
/* Retorna el número de arcos en el grafo g */

6.26. int fuertementeConexo(Grafo g)
/* Informa si el grafo g es fuertemente conexo */

6.27. Problema de Flujo en Redes
Un sistema de alcantarillado se puede modelar como un grafo dirigido, en el cual cada nodo es un tanque y cada arco es un tubo que lleva agua de un tanque a otro. Existen nodos especiales denominados desagües, cuya capacidad es infinita, y otros que no tienen predecesores que se llaman fuentes:



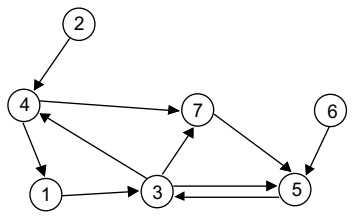
Cada tanque tiene un nombre y una capacidad máxima de almacenamiento, la cual no se puede sobrepasar en ningún momento. De la misma forma, cada tubo tiene un diámetro diferente, el cual le permite transportar una cierta cantidad de agua en una unidad de tiempo (i.e. un minuto). No existe ningún tipo de válvula en la red, de manera que en cada unidad de tiempo sale de cada tanque la capacidad de cada tubo. Si no hay suficiente agua en el tanque para satisfacer la capacidad de cada tubo, se envía por cada uno de ellos una cantidad proporcional a su capacidad. Por ejemplo:



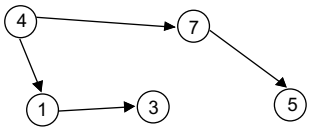
a-) Suponiendo la llegada de una cierta cantidad de agua por unidad de tiempo a una fuente dada del grafo, escriba una función que indique si se genera en algún momento de la siguiente hora un problema de desbordamiento en algún tanque. (Ayuda: haga una simulación del flujo y verifique en cada minuto si hay un problema de desbordamiento. Utilice dos grafos durante la simulación de flujo: uno para el estado de la red en el tiempo T, y otro para el estado en el tiempo $T + 1$)

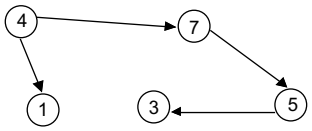
b-) Para limpiar los tubos de la red, periódicamente se envía por ellos una sonda que arrastra hasta los desagües todos los residuos que se van acumulando. Una sonda entra por una fuente y sigue cualquier camino que se le indique hasta llegar a un desagüe. Una sonda no puede pasar dos veces por el mismo tubo en un proceso de limpieza, porque desgasta el material del cual está hecho el tubo. Escriba una función que indique si es posible limpiar todos los tubos de una red con una sonda, con la restricción impuesta anteriormente.

6.28. Dado un grafo $g = (V, A)$, se define un **árbol parcial de recubrimiento** con raíz v ($v \in V$), como el grafo acíclico que incluye todos los vértices de g alcanzables desde v, y suficientes arcos de g para garantizar que existe un único camino desde v hasta cualquier otro vértice del árbol parcial de recubrimiento. Por ejemplo, para el grafo:



Dos posibles árboles parciales de recubrimiento con raíz 4 serían:





Escriba una función que retorne un árbol parcial de recubrimiento para una raíz dada.

6.29. 🕒🕒 Juegos de Guerra

Un general tiene un mapa con las ciudades donde se encuentran sus tropas y los trenes que le permiten llevarlas de un punto a otro. Cuenta además con la información del número total de soldados en cada ciudad y el número máximo de soldados que se pueden transportar en un día sobre cada una de las líneas del tren. El quiere determinar cuantos días necesita para concentrar todos sus soldados en una ciudad dada.

6.30. void elimCiclos(Grafo g)

/* Elimina todos los ciclos, suprimiendo la mínima cantidad posible de arcos */

6.31. 🕒 💻 int numeroCromatico(Grafo g)

/* El **número cromático** de un grafo es el número mínimo de colores con los cuales se pueden colorear los vértices del grafo, sin que dos vértices adyacentes tengan el mismo color. Esta función retorna el número cromático del grafo g */

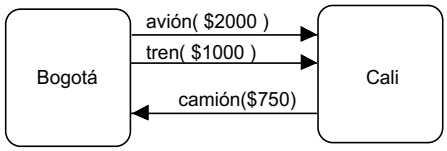
6.32. Grafo invertirGrafo(Grafo g)

/* Retorna un grafo con los mismos vértices de g, pero con el sentido contrario para cada uno de los arcos presentes */

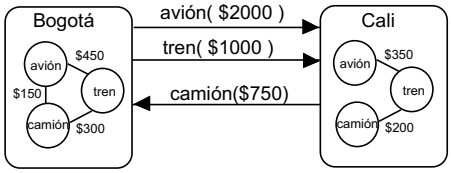
6.33. 🕒 Problema de Transporte

Una empresa de transporte está interesada en desarrollar *software* de apoyo a la toma de decisiones, para establecer la mejor manera de llevar mercancía de una ciudad a otra, según ciertos criterios de optimización.

Entre dos ciudades del país puede haber hasta tres medios diferentes de transporte de mercancía, cada uno con un costo asociado por kilo: camión, tren y avión, como se muestra en la siguiente figura:



Esto implica que por cada kilo que se quiera mover de Bogotá a Cali, se debe pagar $2.000 en avión y $1.000 en tren. Y para hacer el movimiento contrario en camión (la única forma de hacerlo) se debe pagar $750 por kilo. Adicional a este costo de transporte, existe un costo que corresponde al valor de cambiar la mercancía de un medio de transporte a otro, después de llegar a una ciudad. Estos costos son diferentes para cada ciudad, porque dependen de las facilidades que ofrezcan en las terminales de transporte y la distancia a la cual se encuentren los aeropuertos. Es un costo fijo y no depende del peso del material transportado. No en todas las ciudades se puede hacer el cambio de medio de transporte, pero, si esto es posible, cuesta lo mismo en ambos sentidos.



El ejemplo anterior se interpreta de la siguiente manera: si llega mercancía por avión a Bogotá (cualquier cantidad), cuesta $450 pasarla a tren y $150 a un camión. En Cali, por ejemplo, la empresa no hace el cambio de avión a camión, por convenios con el sindicato.

Escriba una rutina que, dadas dos ciudades, encuentre la ruta de costo mínimo para mover N Kg. de mercancía. El costo de esta ruta debe incluir el valor de los medios de transporte y los costos de cambiar de medio en las ciudades.

6.34. Detección de Conglomerados Financieros

Desarrolle un pequeño sistema de control, que ayude a una entidad administrativa del estado a detectar los conglomerados financieros. En este caso, el grafo que modela la información tiene dos tipos de vértices: personas naturales y personas jurídicas (compañías), identificadas mediante su nombre.

Existen varios tipos de relaciones (arcos) entre los vértices:

- Entre dos personas naturales: consanguinidad o afinidad (se conectan con arcos entre ellas)
- Entre dos personas naturales: testaferrato, que la primera persona p1 sea testaferro de la segunda persona p2 (se conectan mediante un arco de p2 a p1).
- Entre una persona natural y una compañía: participación accionaria de la persona en la compañía (se conectan mediante un arco de la persona a la compañía indicando el número de acciones poseídas por esta persona en esa compañía).
- Entre dos compañías: participación accionaria de la primera compañía c1 en la segunda compañía c2 (se conectan mediante un arco de c1 a c2 indicando el número de acciones poseídas por c1 en c2).

Resuelva los siguientes ejercicios:

a-) Dada una compañía, informar por pantalla quienes son sus dueños, en orden de participación accionaria.

b-) Dada una persona o una compañía, informar por pantalla todas las compañías de las cuales es dueña, es decir, posee directamente el 50% o más de las acciones en circulación.

c-) Dada una persona o una compañía, informar por pantalla cual es su conglomerado financiero, es decir todas las compañías de las cuales es directa o indirectamente dueña (de las cuales domina el 50% o más de las acciones en circulación. Este dominio se calcula como la suma de las acciones que posee directamente más las acciones que poseen las personas con las que tiene alguna relación de consanguinidad, afinidad o testaferrato, más las acciones de las compañías de su conglomerado).

6.5.4. Recorridos Heurísticos

Los recorridos estudiados hasta este punto se denominan **desinformados**, puesto que no tienen en cuenta ninguna característica del mundo en el cual ocurre el problema, para hacer más "inteligente" el proceso de avance en la búsqueda de una solución. Sencillamente, garantizan que si la solución existe en el grafo, el proceso de recorrido debe pasar sobre ella en algún momento, y será capaz de identificarla como tal.

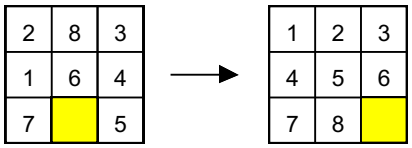
La idea de los **recorridos heurísticos** es darle al algoritmo de búsqueda una medida de qué tan cerca se encuentra de la solución, de tal forma que le dé prioridad en la exploración a los vértices que tengan más posibilidades de llevar a la respuesta. Esto puede disminuir notablemente el tiempo de ejecución, y hacer viable la representación de problemas complejos a través de grafos, aún cuando el espacio de búsqueda sea muy grande e incluso infinito. Este tipo de búsqueda se utiliza como una técnica de inteligencia artificial en juegos, planificación para robótica, deducción automática de teoremas, etc.

En un recorrido heurístico, el frente de búsqueda se encuentra ordenado de acuerdo con una función h(v), que tiene un menor valor a medida que los vértices se encuentran más cerca de la respuesta. Esta función se

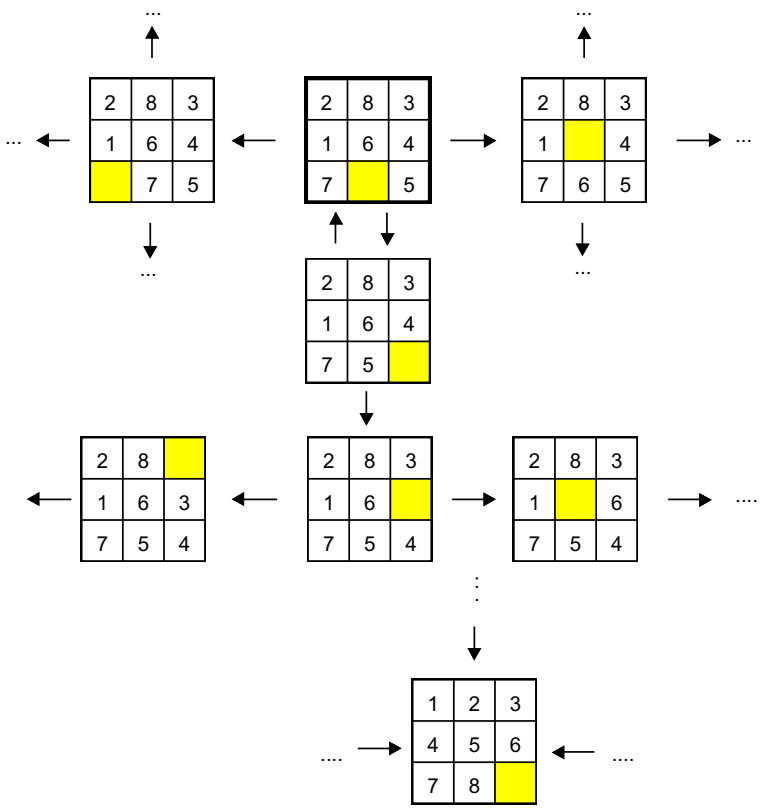
denomina una **heurística**, y su estructura depende del problema mismo sobre el cual se esté trabajando, tal como se ilustra en el siguiente ejemplo.

Ejemplo 6.11:

Dados un tablero de 3 x 3 y unas fichas numeradas de 1 a 8 situadas sobre él, se quiere encontrar una secuencia de jugadas para pasar de una posición inicial dada, a una posición final, en la cual las fichas estén ordenadas. El movimiento de una ficha sólo se puede hacer a la posición vacía del tablero, si ésta se encuentra a su lado:



Este planteamiento da lugar a un grafo de estados del tablero, como el sugerido en la figura que se presenta más adelante, en el cual, cada vértice es un estado de juego, y sus sucesores son los estados a los cuales se puede llegar mediante un movimiento. El problema se resuelve buscando un camino que lleve del nodo actual al nodo final, para lo cual se puede utilizar cualquiera de los recorridos antes estudiados.



Una heurística posible h(v) es sumar el número de movimientos que le falta a cada ficha del tablero, en el mejor de los casos, para llegar a la respuesta, como se ilustra en la siguiente figura:

2	8	3
1	6	4
7		5

=

ficha	movimientos
1	1
2	1
3	0
4	2
5	2
6	1
7	0
8	2
total:	9

6	4	5
7	2	3
8		1

=

ficha	movimientos
1	4
2	1
3	1
4	2
5	2
6	3
7	1
8	1
total:	15

Si ambos nodos estuvieran a punto de ser visitados, el algoritmo escogería el primero de ellos, porque, según la heurística, éste se puede encontrar más cerca de la posición buscada. En particular, de los tres sucesores de la posición inicial del ejemplo, el algoritmo escogería el siguiente movimiento:

2	8	3
1	6	4
7		5

→

2	8	3
1	6	4
7	5	

Debe ser claro que si se selecciona mal una heurística, en lugar de mejorar el desempeño de la búsqueda puede entorpecerla.

El algoritmo para hacer un recorrido heurístico de un grafo dirigido es una variante del recorrido por niveles, en el cual se inserta en el frente de exploración de acuerdo con una función h(v), dándole prioridad en el recorrido a algunos elementos sobre otros.

```
/* pre: g = ( V, A, V' ), v ∈ V, está definida una función h( v ) */
/* post: se ha hecho un recorrido heurístico a partir del vértice v */

void heuristicaGrafo( Grafo g, int v )
{   int i, numVert = ordenGrafo( g );
    desmarcarGrafo( g );
    heuristicaVertice( g, v );
    for ( i = 1; i <= numVert; i++ )
        if ( !marcadoVertice( g, i ) )
            heuristicaVertice( g, i );
    printf("\n");
}

/* pre: g = ( V, A, V' ), v ∈ V, V' tiene los vértices ya recorridos, está definida una función h( v ) */
/* post:  se ha hecho un recorrido heurístico a partir del vértice v de los vértices alcanzables desde
          dicho punto y se han marcado a medida que se van visitando */
```

```
void heuristicaVertice( Grafo g, int v )
{   int x;
    Lista lst, frente = inicLista( );
    anxLista( frente, v );
    while( longLista( frente ) != 0 )
    {     primLista( frente );
          x = infoLista( frente );
          elimLista( frente );
          if( marcadoVertice( g, x ) )
                continue;
          visitar( g, x );
          marcarVertice( g, x );
          lst = sucesores( g, x );
          for ( primLista( lst ); !finLista( lst ); elimLista( lst ) )
                if ( !marcadoVertice( g, infoLista( lst ) ) )
                      insHeuristica( frente, g, infoLista( lst ) );
    }
}

/* pre: frente es una lista ordenada de acuerdo con la función h( x ), v ∉ frente, frente = F */
/* post: frente = F + v, frente es una lista ordenada de acuerdo con la función h( x ) */

void insHeuristica( Lista frente, Grafo g, int v )
{   if( longLista( frente ) == 0 )
          insLista( frente, v );
    else
    {     for( primLista( frente ); !finLista( frente ); sigLista( frente ) )
                if( h( infoVertice( g, v ) ) < h( infoVertice( g, infoLista( frente ) ) ) )
                {     insLista( frente, v );
                      return;
                }
          ultLista( frente );
          anxLista( frente, v );
    }
}
```

El algoritmo de recorrido heurístico antes planteado se puede modificar, para que en el orden del frente de exploración se tenga en cuenta también el costo que ha tenido llegar hasta cada uno de los elementos que allí se encuentran. Esto es, ordenar el frente de acuerdo con una función $f(v) = g(v) + h(v)$, donde $g(v)$ representa el costo acumulado de la búsqueda hasta la posición actual. Esta variante se denomina **algoritmo A***, y, bajo ciertas condiciones de la función heurística, garantiza en un proceso de búsqueda de un elemento en un grafo que encuentra la solución óptima.

Ejercicios Propuestos

6.35. Implemente una rutina heurística que retorne la lista de jugadas para pasar de una posición inicial a una posición final en el juego planteado en la sección anterior. Diseñe otras heurísticas y compare el desempeño del programa.

6.36. Implemente el algoritmo A* para el juego planteado en la sección anterior, de manera que retorne la mejor solución (lista de jugadas) para pasar de la posición inicial a la posición final. La mejor solución, en este caso, es la de menor longitud.

6.37. 🕒 💻 Escriba una rutina que resuelva el problema de las **torres de Hanoi** utilizando búsqueda heurística sobre grafos. Compárela con la solución recursiva pura.

6.38. 🕒 💻 Considere el siguiente problema: "Un granjero con su lobo, cabra y un saco de trigo llegan al borde de un río que desean cruzar. Tienen un bote en el cual sólo hay capacidad para otro elemento diferente del granjero, quien obviamente debe remar. Este sabe que si en algún momento deja solo el lobo con la cabra, el lobo se la come. De la misma manera, sabe que si deja la cabra con el trigo, ésta lo consume. ¿Cómo puede el granjero pasar al otro lado del río sin perder ninguna de sus valiosas posesiones?"

a-) Haga un planteamiento del problema utilizando un grafo dirigido
b-) Resuelva el problema utilizando un recorrido en profundidad del grafo. Calcule el tiempo que toma la búsqueda y la calidad de la solución.
c-) Resuelva el problema utilizando un recorrido por niveles del grafo. Calcule el tiempo que toma la búsqueda y la calidad de la solución.
d-) Resuelva el problema utilizando un recorrido heurístico del grafo. Calcule el tiempo que toma la búsqueda y la calidad de la solución. Diseñe diferentes heurísticas y compare los resultados.

6.39. 🕒🕒 💻 Considere el siguiente problema: "Para atravesar un río, un grupo de N misioneros y N caníbales tienen una lancha para N personas, que debe manejar uno de los misioneros. Si en cualquier momento en la lancha o en cualquier orilla quedan más caníbales que misioneros los devoran. ¿Cómo pasar todos al otro lado del río, sin que haya un festín de misioneros?"

a-) Haga un planteamiento del problema utilizando un grafo dirigido
b-) Resuelva el problema utilizando un recorrido en profundidad del grafo. Calcule el tiempo que toma la búsqueda y la calidad de la solución. Evalúe la manera como se comporta la solución a medida que el valor de N crece.
c-) Resuelva el problema utilizando un recorrido por niveles del grafo. Calcule el tiempo que toma la búsqueda y la calidad de la solución. Evalúe la manera como se comporta la solución a medida que el valor de N crece.
d-) Resuelva el problema utilizando un recorrido heurístico del grafo. Calcule el tiempo que toma la búsqueda y la calidad de la solución. Diseñe diferentes heurísticas y compare los resultados. Evalúe la manera como se comporta la solución a medida que el valor de N crece.

6.40. 🕒🕒 💻 Desarrolle un programa que juegue **picas y famas**, utilizando búsqueda heurística sobre grafos.

6.6. Más Definiciones sobre Grafos

Un grafo sg es **subgrafo** de un grafo g, si incluye un subconjunto de sus vértices y todos los arcos que los relacionan en g. Por ejemplo, si un grafo representa las ciudades y las carreteras de un país, un subgrafo sería el grafo de las carreteras de un departamento. En la figura 6.12 aparecen tres ejemplos de subgrafos de un grafo dado.

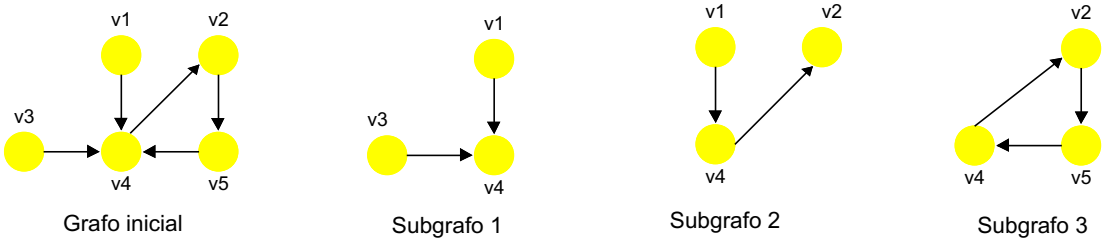


Fig. 6.12 - Subgrafos de un grafo

- subgrafo(g, sg) ssi $g = (V, A), sg = (V_{sg}, A_{sg}),$

$V_{sg} \subseteq V,$

$\forall x, y \in V_{sg}$, si $(x, y, c) \in A \Rightarrow (x, y, c) \in A_{sg}$

Un grafo gp es un **subgrafo parcial** de un grafo g, si se encuentra formado a partir de un subconjunto de arcos de g y el conjunto de vértices correspondiente. En la figura 6.13 aparecen tres ejemplos de subgrafos parciales de un grafo:

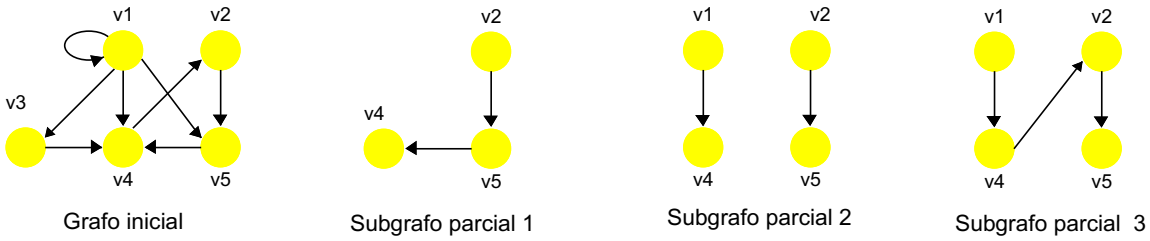


Fig. 6.13 - Subgrafos parciales de un grafo

- subgrafoP(g, gp) ssi $g = (V, A), gp = (Vp, Ap),$

$Ap \subseteq A,$

$\forall (x, y, c) \in Ap \Rightarrow x \in Vp, y \in Vp$

La diferencia entre un subgrafo y un subgrafo parcial, es que en el primero se parte de un subconjunto de vértices, y se construye el conjunto de arcos con todas las relaciones presentes entre los vértices seleccionados. Por su parte, en un subgrafo parcial, se parte de un conjunto de arcos y el conjunto de vértices se construye con todos los vértices referenciados en los arcos escogidos. Continuando con el ejemplo del mapa de ciudades y carreteras, un subgrafo parcial sería aquél que incluye sólo las carreteras principales de un país, y por ende, las ciudades por las cuales pasan.

De acuerdo con las dos definiciones dadas, se puede concluir que todo subgrafo es también un subgrafo parcial.

- subgrafo(g1, g2) $\Rightarrow$ subgrafoP(g1, g2)

Si un grafo g es no conexo, puede dividirse en subgrafos de tal forma que cada uno de ellos sea conexo. Estos subgrafos se denominan **componentes conexos** de g. En la figura 6.14 aparecen algunos componentes conexos de un grafo no conexo.

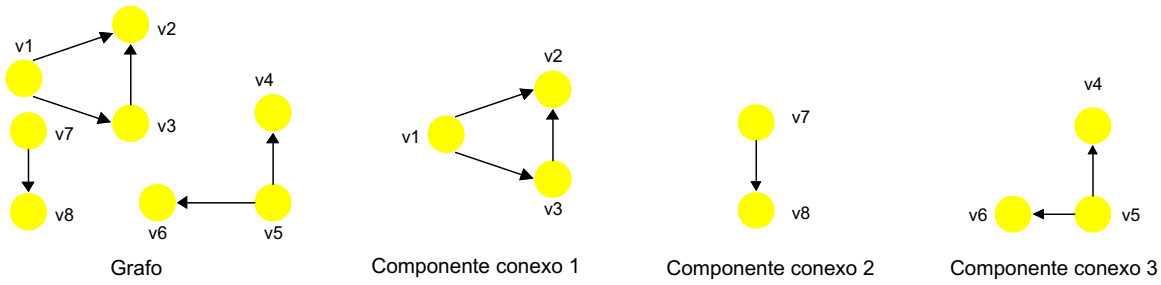


Fig. 6.14 - Componentes conexos de un grafo

Un **árbol de recubrimiento** para un grafo g es un subgrafo parcial de g, que estructuralmente cumple la condición de ser un árbol n-ario y además contiene todos los vértices del grafo original. Este tipo de árboles no siempre existe para un grafo y tampoco es único. La **raíz** del árbol de recubrimiento es el vértice del cual comienza el árbol. Se define el **peso** de un árbol de recubrimiento como la suma de los pesos de los arcos incluidos en él. Un árbol de recubrimiento para una raíz dada es **mínimo**, si tiene el peso mínimo de todos los que existen. Este árbol tampoco tiene que ser único. Un árbol de recubrimiento con n vértices, debe tener n-1 arcos. En la figura 6.15 aparecen algunos ejemplos de árboles de recubrimiento para un grafo.

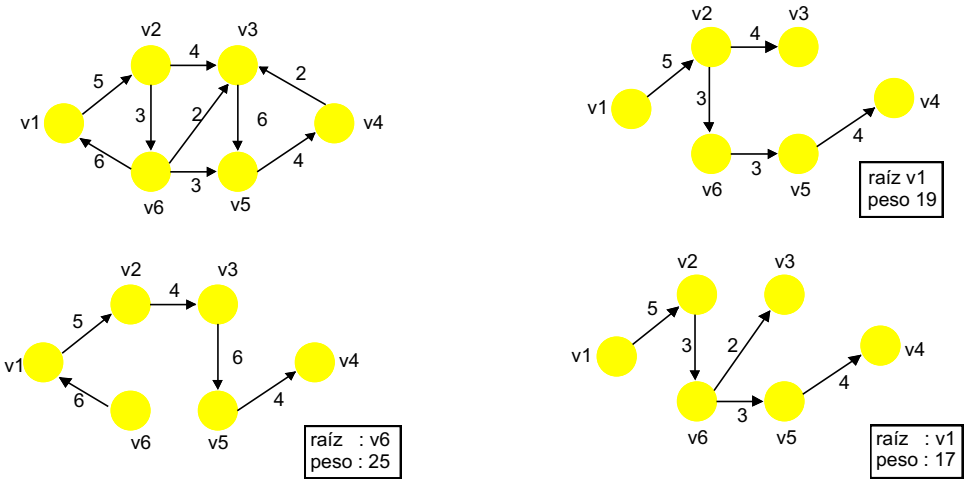


Fig. 6.15 - Arboles de recubrimiento de un grafo

La **clausura transitiva** de un grafo g, es un grafo con el mismo conjunto de vértices y un arco de un vértice a otro si existe un camino que los una. El costo de los arcos corresponde al costo del camino más barato que los une. En la figura 6.16 aparece un ejemplo de la clausura transitiva de un grafo.

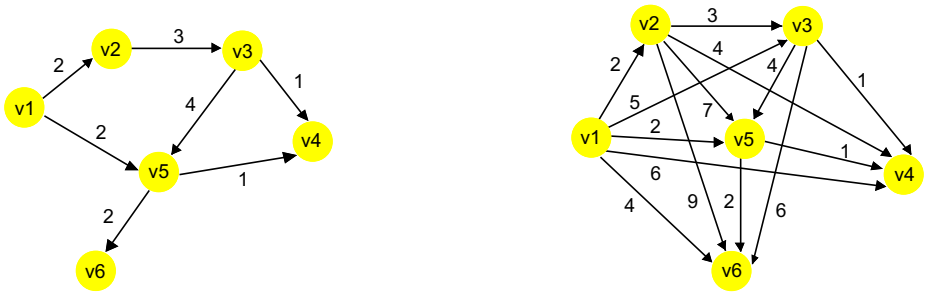


Fig. 6.16 - Clausura transitiva de un grafo

Ejercicios Propuestos

Resuelva los siguientes ejercicios y calcule la complejidad de su solución. Intente encontrar la rutina más eficiente posible para cada problema.

6.41. int subGrafo(Grafo g, Grafo sg)
/* Informa si sg es un subgrafo de g */

6.42. int subGrafoP(Grafo g, Grafo gp)
/* Informa si gp es un subgrafo parcial de g */

6.43. int componentesConexos(Grafo g)
/* Calcula el número de componentes conexos del grafo g */

6.44. Grafo arbolRecubrimiento(Grafo g, int v)
/* Retorna un árbol de recubrimiento para el grafo g con raíz v. Si no existe retorna un grafo vacío */

6.45. Grafo arbolRecubrimientoMinino(Grafo g, int v)
/* Retorna un árbol de recubrimiento mínimo para el grafo g con raíz v. Si no existe retorna un grafo vacío */

6.46. Lista conjuntoCorte(Grafo g)
/* Se define un **conjunto de corte** de un grafo g como el conjunto minimal de arcos, tal que si se suprimen, el grafo deja de ser conexo. Esta función retorna una lista de arcos (parejas de vértices) que corresponde a un conjunto de corte del grafo g */

6.47. Lista nucleoGrafo(Grafo g)
/* Se denomina un **núcleo** de un grafo al subconjunto minimal de vértices de g, tal que existe un camino hasta todos los vértices de g que comienza en un vértice del núcleo */

6.48. Grafo clausuraTransitiva(Grafo g)
/* Crea la clausura transitiva del grafo g */

6.49. Demuestre formalmente que todo subgrafo es también un subgrafo parcial.

6.7. El Algoritmo de Dijkstra

Los métodos de búsqueda de caminos han tenido, hasta esta sección, un planteamiento puramente recursivo, basado en un recorrido en profundidad del grafo. Esto hace que resulten ineficientes para resolver ciertos problemas. Dijkstra propuso un algoritmo iterativo, de complejidad cuadrática, para encontrar los caminos de costo mínimo que parten de un vértice dado y terminan en cada uno de los demás vértices del grafo. Este mismo problema resuelto con el algoritmo recursivo de búsqueda de caminos planteado en una sección anterior sería $O(n^3)$.

La misma idea de este algoritmo se puede aplicar para resolver muchos otros problemas sobre grafos, y por esta razón es importante su estudio.

La presentación del algoritmo de Dijkstra se hace en dos etapas. La primera etapa trata el problema de determinar el costo del camino mínimo que parte de un vértice y termina en cada uno de los demás elementos del grafo. La segunda etapa almacena además la secuencia de vértices que conforman dicho camino.

6.7.1. Costo de los Caminos Mínimos

La especificación de este primer problema es la siguiente:

int *dijkstra(Grafo g, int v)

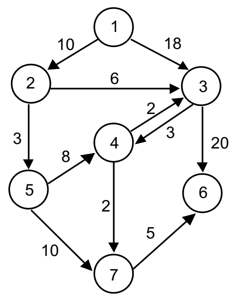
/* pre: g = (V, A), v ∈ V, card (V) = N */

/* post: dijkstra = 

, dijkstra [i] = costo del camino mínimo de v al vértice i+1 */

Ejemplo 6.12:

Para el grafo de la figura, al aplicar el algoritmo de Dijkstra para determinar el costo de los caminos mínimos que parten del vértice 1 y van a todos los demás vértices, la rutina debe retornar el vector que se da a continuación:

g = 

dijkstra(g, 1) =

0	1	2	3	4	5	6
0	10	16	19	13	26	21

Este vector contiene en la posición k el costo del camino mínimo que lleva del vértice 1 al vértice k+1. Por ejemplo, el costo del camino mínimo que lleva del vértice 1 al vértice 7 es 21, y se encuentra en la posición 6 del vector.

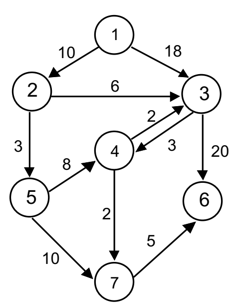
⏎

Se define un **camino especial** entre el vértice v1 y el vértice v2 de un grafo g, como una secuencia de vértices de V, < x_1 ... x_n >, que cumple:

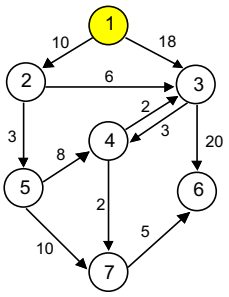
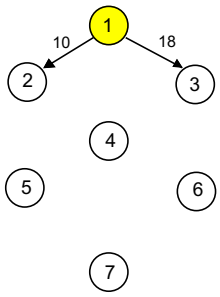
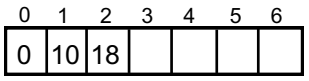
- < x_1 ... x_n > es un camino simple de v1 a v2 ($x_1 = v_1$, $x_n = v_2$)
- x_1,..., x_{n-1} ∈ V' (sólo incluye vértices marcados, exceptuando posiblemente a x_n)
- No existe ningún otro camino en g de v_1 a v_2 que, pasando sólo por vértices marcados, tenga un costo menor que el costo de < x_1 ... x_n >

Ejemplo 6.13:

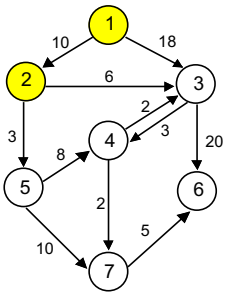
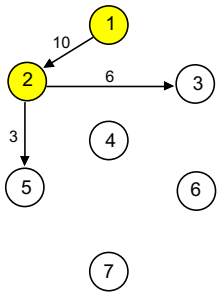
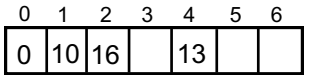
Este ejemplo ilustra y explica todos los pasos del algoritmo de Dijkstra. Considere el grafo de la figura, y suponga que se quiere determinar el costo del camino mínimo que lleva del vértice 1 a cada uno de los demás vértices del grafo:



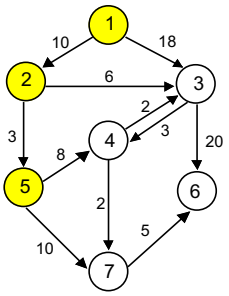
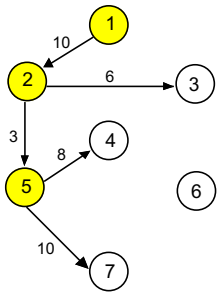
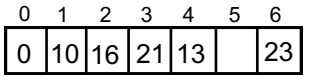
1. Marcar el vértice de origen y calcular el costo de todos los caminos especiales. Si no existe el camino especial a algún vértice se coloca -1 (vacío en el dibujo):

Grafo	Caminos especiales	Costos
		

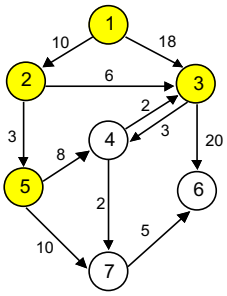
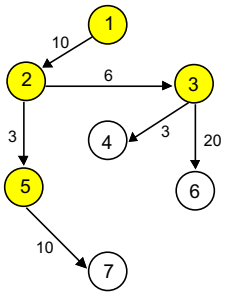
2. Seleccionar el vértice no marcado con costo de camino especial mínimo (vértice 2 con costo 10), marcarlo y recalcular todos los caminos especiales de la siguiente manera: para cada vértice no marcado se debe decidir si resulta mejor el camino antes calculado o si es preferible utilizar el camino especial hasta el vértice que se acaba de marcar, y luego utilizar el arco que los une. Por ejemplo, para el vértice 3 es mejor utilizar el camino especial que lleva al vértice 2 y luego utilizar el arco que los une (costo 16), que seguir el camino especial antes calculado (costo 18).

Grafo	Caminos especiales	Costos
		

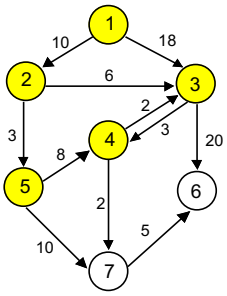
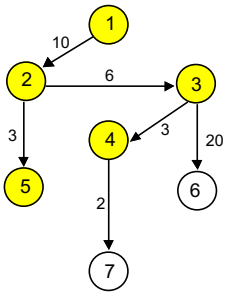
3. Repetir el proceso mientras haya vértices no marcados. Se escoge el vértice 5.

Grafo	Caminos especiales	Costos
		

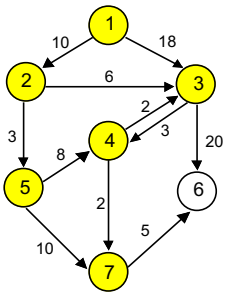
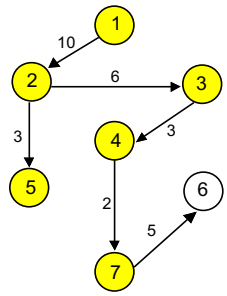
4. Repetir el proceso mientras haya vértices no marcados. Se escoge el vértice 3.

Grafo	Caminos especiales	Costos
		0 1 2 3 4 5 6 0 10 16 19 13 36 23

5. Repetir el proceso mientras haya vértices no marcados. Se escoge el vértice 4.

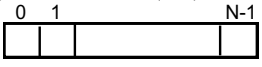
Grafo	Caminos especiales	Costos
		0 1 2 3 4 5 6 0 10 16 19 13 36 21

6. Repetir el proceso mientras haya vértices no marcados. Se escoge el vértice 7.

Grafo	Caminos especiales	Costos
		0 1 2 3 4 5 6 0 10 16 19 13 26 21

La implementación del algoritmo de Dijkstra es la siguiente:

/* pre: g = (V, A), v ∈ V, card (V) = N */

/* post: dijkstra = 

, dijkstra [i] = costo del camino mínimo de v al vértice i+1 */

```
int *dijkstra( Grafo g, int v )
{   int c, i, numElem = ordenGrafo( g );
    int *costo = calloc( numElem, sizeof( int ) );
    desmarcarGrafo( g ) ;
    for( i = 0; i < numElem ; i++ )
        costo [ i ] = costoArco( g, v, i + 1 ) ;
    marcarVertice( g, v );
    costo [ v - 1 ] = 0 ;
    while( ( v = siguienteVertice( g, costo, numElem ) ) != -1 )
    {   marcarVertice( g, v ) ;
        for( i = 0; i < numElem ; i++ )
            if( !marcadoVertice( g, i + 1 ) && ( c = costoArco( g, v, i + 1 ) ) != -1 )
                if( costo [ i ] == -1 )
                    costo [ i ] = costo [ v - 1 ] + c;
                else
                    costo [ i ] = min( costo [ i ], costo [ v - 1 ] + c );
    }
    return costo;
}
```

El algoritmo termina cuando todos los vértices están marcados, o cuando hasta ninguno de los no marcados hay un camino desde el vértice inicial. La rutina que retorna el siguiente vértice que se debe incluir en el conjunto de marcados es la siguiente:

```
/* pre: g = ( V, A ), costo[ i ] ≥ -1, costo[ i ] ≥ 0 si i+1 ∈ V' */
/* post:  ( siguienteVertice = v ∈ V,  v ∉ V', costo [ v-1 ] ≠ 1, costo [ v-1 ] es mínimo entre todos los v posibles ) ∨
          ( siguienteVertice  = -1, si no existe un v que cumpla las condiciones anteriores ) */

int siguienteVertice( Grafo g, int costo[ ], int numElem )
{   int i, menor = -1;
    for( i = 0 ; i < numElem ; i++ )
        if( !marcadoVertice( g, i + 1 ) && costo[ i ] != -1 )
            if( menor == -1 || costo[ menor ] > costo[ i ] )
                menor = i;
    return ( menor == -1 ) ? -1 : menor + 1;
}
```

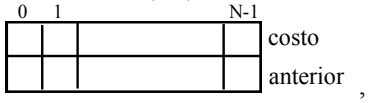
6.7.2. Caminos Mínimos

Una sencilla variante de este algoritmo permite calcular el camino de costo mínimo que parte de un nodo y lleva a cada uno de los demás nodos del grafo. El resultado de este algoritmo se puede utilizar para construir el árbol de recubrimiento mínimo con raíz v. La especificación de este segundo problema es la siguiente:

```
typedef struct
{   int costo;
    int anterior;
} Pareja;
```

Pareja *dijkstra(Grafo g, int v)

/* pre: g = (V, A), v ∈ V, card (V) = N */

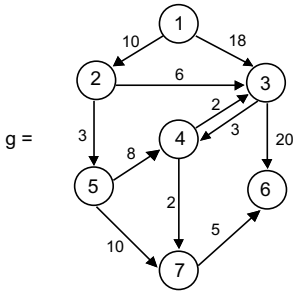
/* post: dijkstra = 

,

dijkstra[i].costo = costo del camino mínimo de v al vértice i+1
dijkstra[i].anterior = anterior elemento en el camino mínimo que lleva al vértice i+1*/

Ejemplo 6.14:

Para el grafo de la figura, al aplicar el algoritmo de Dijkstra para determinar los caminos mínimos que parten del vértice 1 y van a todos los demás vértices, la rutina debe retornar el vector que se da a continuación:

g = 

dijkstra(g, 1) =

0	1	2	3	4	5	6	← posición
0	10	16	19	13	26	21	
	1	2	3	2	7	4	
1	2	3	4	5	6	7	← vértice

Este vector contiene en la posición k el costo del camino mínimo que lleva del vértice 1 al vértice k+1, y el vértice anterior al k+1 en el camino mínimo que llega hasta ese punto. Los caminos mínimos resultantes de interpretar dicho vector son:

camino(1, 1) = < 1 >, costo = 0

camino(1, 2) = < 1, 2 >, costo = 10

camino(1, 3) = < 1, 2, 3 >, costo = 16

camino(1, 4) = < 1, 2, 3, 4 >, costo = 19

camino(1, 5) = < 1, 2, 5 >, costo = 13

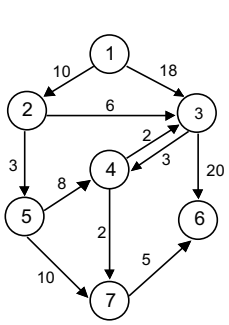
camino(1, 6) = < 1, 2, 3, 4, 7, 6 >, costo = 26

camino(1, 7) = < 1, 2, 3, 4, 7 >, costo = 21

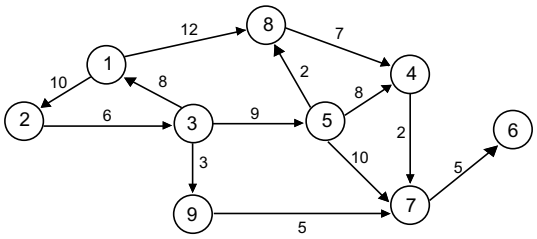
⏎

Ejercicios Propuestos

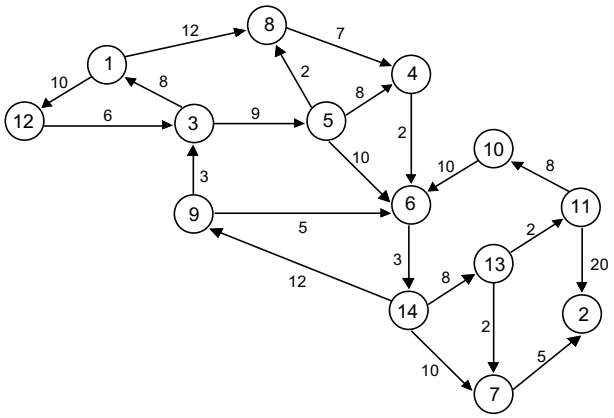
6.50. Para el grafo de la figura, calcule los caminos mínimos a partir del vértice 4, utilizando el algoritmo de Dijkstra:



6.51. Para el grafo de la figura, calcule los caminos mínimos a partir del vértice 3, utilizando el algoritmo de Dijkstra:



6.52. Para el grafo de la figura, calcule los caminos mínimos a partir del vértice 6, utilizando el algoritmo de Dijkstra:



6.53. 💻 Implemente la modificación en el algoritmo de Dijkstra, planteada en la sección anterior, para que retorne los caminos mínimos.

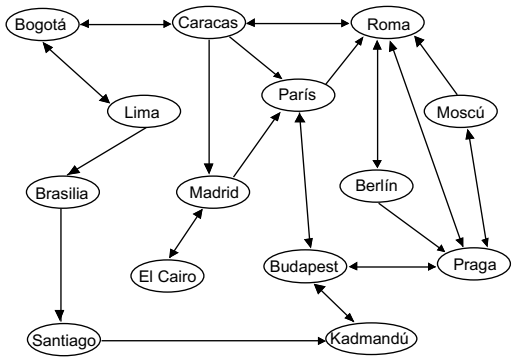
6.54. 💻 Desarrolle una rutina que imprima por pantalla los caminos mínimos desde un vértice a todos los demás, utilizando como entrada la salida del ejercicio anterior. Esto es, un procedimiento que tome un vector de parejas [costo, anterior] y despliegue el camino a cada uno de los vértices del grafo.

6.55. 💻 Modifique el algoritmo de Dijkstra para que retorne el árbol mínimo de recubrimiento de un grafo g, con raíz v.

6.56. 💻 Modifique el algoritmo de Dijkstra para que retorne la clausura transitiva de un grafo.

6.57. 💻 Desarrolle una rutina basada en el algoritmo de Dijkstra que retorne el camino de costo mínimo que lleva de un vértice a otro.

6.58. 🕒 💻 El objetivo de este ejercicio es desarrollar un programa que le permita a una agencia de viajes planear las excursiones de sus clientes. Para esto, cuenta con información de todos los vuelos que existen en el mundo.



Dada una ciudad de origen y una lista de ciudades que desea visitar el cliente, retorna un plan de viaje con el mínimo número de ciudades. Para el dibujo del ejemplo anterior, si la ciudad de origen es Bogotá y las ciudades que quiere visitar el cliente son Kadmandú y Moscú, un plan de viaje sería: Bogotá, Caracas, París, Budapest, Kadmandú, Budapest, Praga, Moscú, Roma, Caracas, Bogotá. Fíjese que el plan no es único. También sería posible seguir la siguiente ruta: Bogotá, Lima, Brasilia, Santiago, Kadmandú, Budapest, Praga, Moscú, Roma, Caracas, Bogotá. Ambas rutas incluyen 11 ciudades. Puede suponer que en la lista de ciudades no existen repetidas.

6.59. Se define un **ρ-grafo** como un grafo dirigido conexo sin ciclos, en el cual si se suprime cualquier arco, el grafo deja de ser conexo. Desarrolle una rutina que informe si un grafo dado es un ρ-grafo.

6.60. Lista ordenTopologico(Grafo g)
/* Un **ordenamiento topológico** de un grafo $g = (V, A)$ es una lista $< x_1, ..., x_n >$ que contiene los vértices de V, en un orden tal que si $(v, w, c) \in A$, entonces v se encuentra en esa lista antes que w. Esta función calcula el ordenamiento topológico de un grafo, suponiendo que es acíclico, condición suficiente para que exista el ordenamiento. */

6.8. Implementación del TAD Grafo

Existen muchas formas distintas de implementar el TAD Grafo. En esta sección se presentan varias de ellas, consideradas como clásicas, y se ilustra, mediante ejemplos, el diseño de otras estructuras de datos, para casos en los cuales las características mismas del problema lo exigen.

6.8.1. Matrices de Adyacencias

Esta es una de las representaciones más comunes para un grafo. Se define una **matriz de adyacencias** como una matriz de $N \times N$, donde N es el orden del grafo, en la cual se tiene en la posición $[i-1, j-1]$ el costo del arco que existe entre los vértices i y j del grafo. Si éste no existe, se coloca una marca especial para indicarlo. En este caso se va a utilizar el valor -1. Con una matriz de este tipo se representa el conjunto de arcos con el siguiente esquema:

- $(v, w, c) \in A$ ssi arcos[v-1][w-1] = c
- $(v, w, c) \notin A$ ssi arcos[v-1][w-1] = -1

Para incluir en las estructuras de datos toda la información asociada con los vértices, se maneja también un vector de N posiciones, en el cual, en la posición $i-1$ aparece el contenido del vértice i, y una indicación de si se encuentra o no marcado.

- info(i) = T ssi vertices[i-1]. info = T

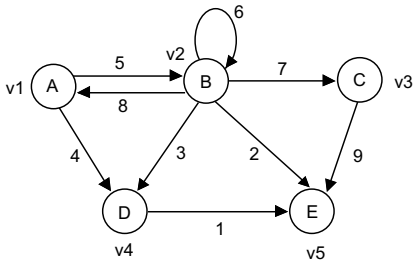
- i ∈ V' ssi vertices[i-1]. marca = 1
- i ∉ V' ssi vertices[i-1]. marca = 0

Lo único que falta en las estructuras de datos, es una manera de manejar un número variable de vértices en el grafo. Con este fin, se utiliza un campo adicional para indicar el número de vértices presentes.

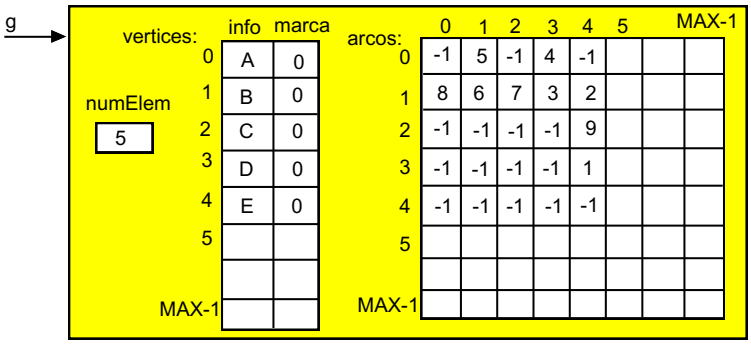
- orden(g) = N ssi numElem = N

Ejemplo 6.15:

Para el grafo de la figura:



Las estructuras de datos que lo representan, suponiendo que el tamaño máximo de la matriz de adyacencias es MAX, son:



Las estructuras de datos para esta representación se declaran de la siguiente manera:

```
typedef struct
{   TipoG info;          /* Información asociada con un vértice */
    int marca;           /* Marca del vértice */
} Vertice;

typedef struct
{   int **arcos;         /* Matriz de cualquier tamaño, pedida en ejecución */
    Vertice *vertices;   /* Vector de vértices de cualquier tamaño, pedido en ejecución */
    int numElem;         /* Número de vértices */
} TGrafo, *Grafo;
```

Las operaciones del TAD Grafo se implementan con las rutinas que se presentan a continuación:

- La rutina de inicialización debe reservar el espacio en memoria dinámica para el objeto abstracto, incluyendo una matriz de MAX x MAX enteros (inicializado en -1) y un vector para la información de los vértices. La complejidad de la operación es O(MAX), puesto que depende de esta constante de la implementación:

```
Grafo inicGrafo( void )
{   int i, k;
    Grafo g = ( Grafo )malloc( sizeof( TGrafo ) );
    g->arcos = ( int ** )calloc( MAX, sizeof( int * ) );
    for( i = 0 ; i < MAX ; i++ )
    {     g->arcos[ i ] = ( int * )calloc( MAX, sizeof( int ) ) ;
          for( k = 0; k < MAX; k++ )
                g->arcos[ i ][ k ] = -1;
    }
    g->vertices = ( Vertice * )calloc( MAX, sizeof( Vertice ) );
    g->numElem = 0;
    return g;
}
```

- Para insertar un nuevo vértice, la rutina toma la siguiente posición libre del vector vértices, y coloca allí la información. Sobre la matriz no es necesario hacer ninguna modificación, puesto que la operación de inicialización garantiza que la fila y la columna correspondientes a este nuevo vértice tienen el valor -1. Esta rutina es O(1).

```
void insVertice( Grafo g, TipoG elem )
{   g->vertices[ g->numElem ]. info = elem;
    g->vertices[ g->numElem ]. marca = 0;
    g->numElem++;
}
```

- Las operaciones de insertar y eliminar un arco se implementan con rutinas elementales, de complejidad constante, como se muestra a continuación:

```
void insArco( Grafo g, int v, int w, int c )
{   g->arcos[ v-1 ][ w-1 ] = c;
}

void elimArco( Grafo g, int v, int w )
{   g->arcos[ v-1 ][ w-1 ] = -1;
}
```

- La operación que calcula el costo de un arco debe tomar el valor correspondiente de la matriz y retornarlo. La complejidad es O(1).

```
int costoArco( Grafo g, int v, int w )
{   return g->arcos[ v-1 ][ w-1 ] ;
}
```

- La lista de sucesores de un vértice se obtiene recorriendo la fila correspondiente en la matriz de adyacencias. Cada arco con valor diferente a -1 indica un sucesor. La complejidad de la rutina es O(n), donde n es el orden del grafo.

```
Lista sucesores( Grafo g, int v )
{   int i;
    Lista lst = inicLista( );
    for( i = 0 ; i < g->numElem ; i++ )
        if( g->arcos[ v-1 ][ i ] != -1 )
            anxLista( lst, i+1 );
    return lst;
}
```

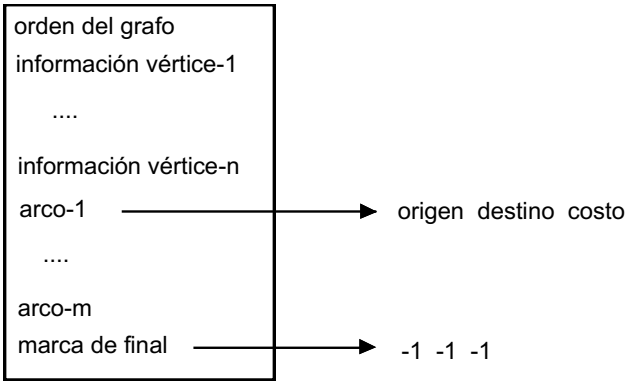
- Algunas de las demás operaciones del TAD Grafo se implementan de manera trivial sobre matrices de adyacencias, tal como se muestra a continuación. Todas son O(1), con excepción de la operación que desmarca todos los vértices del grafo, que tiene complejidad O(n), donde n es el orden del grafo.

```
TipoG infoVertice( Grafo g, int v )
{   return g->vertices[ v-1 ].info;
}

int ordenGrafo( Grafo g )
{   return g->numElem;
}

void marcarVertice( Grafo g, int v )
{   g->vertices[ v-1 ].marca = 1;
}

void desmarcarVertice( Grafo g, int v )
{   g->vertices[ v-1 ].marca = 0;
}

void desmarcarGrafo( Grafo g )
{   int i;
    for( i = 0 ; i < g->numElem ; i++ )
        g->vertices[ i ].marca = 0;
}

int marcadoVertice( Grafo g, int v )
{   return g->vertices[ v-1].marca;
}
```

- La operación de destrucción de un grafo, debe comenzar por retornar cada una de las filas de la matriz dinámica, y, luego, devolver los vectores y nodos pedidos en la inicialización. La complejidad de la rutina es O(MAX), donde MAX es una constante de la implementación.

```
void destruirGrafo( Grafo g )
{   int i;
    for( i = 0 ; i < MAX ; i++ )
        free( g->arcos[ i ] );
    free( g->arcos );
    free( g-> vertices );
    free( g );
}
```

- El esquema de persistencia seleccionado para esta implementación consiste en un archivo de texto, que almacena los elementos del grafo en el siguiente orden:



Las rutinas que cargan y salvan un grafo de un archivo con la estructura antes mencionada son:

```
Grafo cargarGrafo( FILE *fp )
{   int numVertices, v1, v2, costo;
    Grafo g = inicGrafo();
    fscanf( fp, "%d", &numVertices );
    for( ; numVertices > 0; numVertices-- )
    {     fscanf( fp, "%d", &v1 );
          insVertice( g, v1 );
    }
    fscanf( fp, "%d %d %d", &v1, &v2, &costo );
    while( v1 != -1 )
    {     insArco( g, v1, v2, costo );
          fscanf( fp, "%d %d %d", &v1, &v2, &costo );
    }
    return g;
}

void salvarGrafo( Grafo g, FILE *fp )
{   int i, k;
    fprintf( fp, "%d\n", g->numElem );
    for( i = 0 ; i < g->numElem; i++ )
          fprintf( fp, "%d\n", infoVertice( g, i+1 ) );
    for( i = 0 ; i < g->numElem; i++ )
          for( k = 0; k < g->numElem; k++ )
                if( g->arcos[ i ][ k ] != -1 )
                      fprintf( fp, "%d %d %d\n", i+1, k+1, g->arcos[ i ][ k ] );
    fprintf( fp, "-1 -1 -1\n" );
}
```

Esta implementación tiene la ventaja de que las operaciones de acceso a la información son de complejidad constante, haciéndolas independientes del tamaño del grafo. El inconveniente de su uso radica en el espacio ocupado, que es O(MAX²), donde MAX es el máximo número de vértices que puede manejar el grafo. Esto hace que las necesidades de memoria crezcan con el cuadrado del orden del grafo (n ≤ MAX), lo cual la convierte en una implementación costosa.

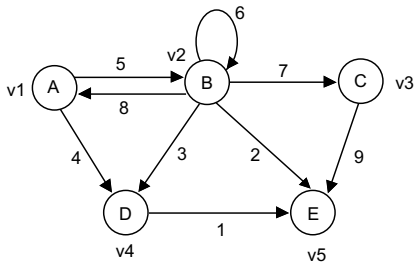
6.8.2. Listas de Sucesores

El objetivo de esta implementación es obtener una representación de un grafo con complejidad en memoria proporcional al número de arcos. Esto va a repercutir necesariamente en la eficiencia de algunas de las operaciones. La idea es tener un vector de vértices que incluya, además de toda la información, una lista con los arcos (parejas de la forma [sucesor, costo]). El esquema de representación es el siguiente:

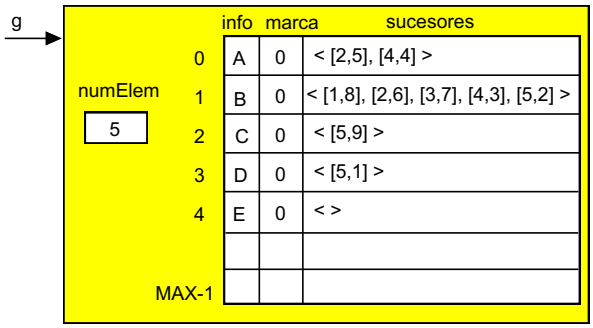
- info(i) = K ssi vertices[i - 1].info = K
- i ∈ V' ssi vertices[i - 1].marca = 1
- i ∉ V' ssi vertices[i - 1].marca = 0
- (v, w, c) ssi [w, c] ∈ vertices[v - 1].sucesores
- orden(g) = n ssi numElem = n

Ejemplo 6.16:

Para el grafo de la figura:



Las estructuras de datos que lo representan son:



En esta implementación de grafos hay un pequeño problema: es necesario manejar en las estructuras de datos una lista de parejas (Lista [vértice, costo]) y, en la implementación de la operación sucesores, una lista de vértices (Lista[vértice]). Esto obliga a que se manejen dos copias de la implementación del TAD Lista, y se deba modificar una de ellas para que pueda almacenar parejas de elementos. Con mínimas modificaciones del código se crea el TAD ListaP, que maneja elementos de tipo TipoLP, y en el cual los nombres de las operaciones se modifican agregando una P al final, con el fin de distinguirlas de las operaciones equivalentes en el TAD Lista. Teniendo en cuanta esto, las estructuras de datos se declaran de la siguiente manera:

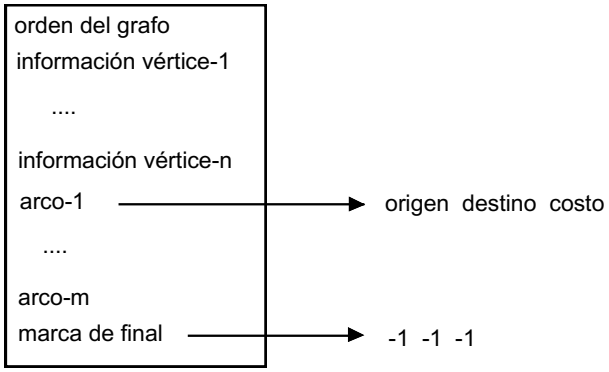
```
typedef struct
{   int vertice;            /* Vértice sucesor */
    int costo;              /* Costo del arco */
} TipoLP;

typedef struct
{   TipoG info;             /* Información asociada con el vértice */
    int marca;              /* Marca */
    ListaP sucesores;       /* Lista de parejas [ sucesor, costo ] */
} Vertice;

typedef struct
{   Vertice *vertices;      /* Vector de tamaño dinámico de vértices */
    int numElem;            /* Orden del grafo */
} TGrafo, *Grafo;
```

Las rutinas que implementan las operaciones del TAD Grafo sobre listas de sucesores son las siguentes.

```
Grafo inicGrafo( void )
{   Grafo g = ( Grafo )malloc( sizeof( TGrafo ) );
    g->vertices = ( Vertice * ) calloc( MAX, sizeof( Vertice ) );
    g->numElem = 0;
    return g;
}

void insVertice( Grafo g, TipoG elem )
{   g->vertices[ g->numElem ].info = elem;
    g->vertices[ g->numElem ].marca = 0;
    g->vertices[ g->numElem ].sucesores = inicListaP( );
    g->numElem++;
}

void insArco( Grafo g, int v, int w, int c )
{   TipoLP a;
    a.vertice = w;
    a.costo =  c;
    primListaP( g->vertices[ v-1 ].sucesores );
    insListaP( g->vertices[ v-1 ].sucesores, a ) ;
}

void elimArco( Grafo g, int v, int w )
{   ListaP lst = g->vertices[ v-1 ].sucesores;
    for( primListaP( lst ); !finListaP( lst ); sigListaP( lst ) )
        if( infoListaP( lst ).vertice == w )
        {     elimListaP( lst );
              break;
        }
}

void marcarVertice( Grafo g, int v )
{   g->vertices[ v - 1 ].marca = TRUE;
}
```

```
void desmarcarVertice( Grafo g, int v )
{   g->vertices[ v - 1 ].marca = FALSE;
}

void desmarcarGrafo( Grafo g )
{   int i;
    for( i = 0; i < g->numElem; i++ )
        g->vertices[ i ].marca = FALSE;
}

int costoArco( Grafo g, int v, int w )
{   ListaP lst = g->vertices [ v - 1 ].sucesores;
    for( primListaP( lst ); !finListaP( lst ); sigListaP( lst ) )
        if( infoListaP( lst ).vertice == w )
            return infoListaP( lst ).costo;
    return -1;
}

Lista sucesores( Grafo g, int v )
{   Lista resp = inicLista( );
    ListaP lst = g->vertices[ v-1 ].sucesores;
    for( primListaP( lst ); !finListaP( lst ); sigListaP( lst ) )
        anxLista( resp, infoListaP( lst ).vertice );
    return resp;
}

TipoG infoVertice( Grafo g, int v )
{   return g->vertices[ v - 1 ].info;
}

int ordenGrafo( Grafo g )
{   return g->numElem;
}

int marcadoVertice( Grafo g, int v )
{   return g->vertices[ v - 1 ].marca;
}

void destruirGrafo( Grafo g )
{   int i;
    for( i = 0; i < g->numElem; i++ )
        destruirListaP( g->vertices[ i ].sucesores );
    free( g->vertices );
    free( g );
}
```

Para la implementación de las operaciones de persistencia, se utiliza la misma estructura del archivo de texto planteada en la representación de matrices de adyacencias, y que se resume en la siguiente figura:




```
Grafo cargarGrafo( FILE *fp )
{   int numVertices, v1, v2, costo;
    Grafo g = inicGrafo();
    fscanf( fp, "%d", &numVertices );
    for( ; numVertices > 0; numVertices-- )
    {     fscanf( fp, "%d", &v1 );
          insVertice( g, v1 );
    }
    fscanf( fp, "%d %d %d", &v1, &v2, &costo );
    while( v1 != -1 )
    {     insArco( g, v1, v2, costo );
           fscanf( fp, "%d %d %d", &v1, &v2, &costo );
    }
    return g;
}

void salvarGrafo( Grafo g, FILE *fp )
{   int i;
    ListaP lst;
    fprintf( fp, "%d\n", g->numElem );
    for( i = 0; i < g->numElem; i++ )
          fprintf( fp, "%d\n", g->vertices[ i ].info );
    for( i = 0; i < g->numElem; i++ )
    {     lst = g->vertices[ i ].sucesores;
          for( primListaP( lst ); !finListaP( lst ); sigListaP( lst ) )
                fprintf( fp, "%d %d %d\n", i+1, infoListaP( lst ).vertice, infoListaP( lst ).costo );
    }
    fprintf( fp, "-1 -1 -1\n");
}
```

La siguiente tabla resume la complejidad de las operaciones del TAD Grafo, para la representación de listas de sucesores, donde n es el orden del grafo y m el número de arcos:

inicGrafo	O(1)
insVertice	O(1)
insArco	O(1)
elimArco	O(n)
costoArco	O(n)
sucesores	O(n)
infoVertice	O(1)

ordenGrafo	O(1)
marcarVertice	O(1)
desmarcarVertice	O(1)
desmarcarGrafo	O(n)
marcadoVertice	O(1)
destruirGrafo	O(n)
cargarGrafo	O(max(n, m))
salvarGrafo	O(max(n, m))

La ventaja de esta representación es que utiliza únicamente el espacio de memoria indispensable para representar el grafo. Esto es, el espacio depende únicamente del número de arcos presentes y no del número total de arcos posibles, como en la implementación anterior. La desventaja es que, para determinar si existe un arco entre dos vértices dados, se requiere un proceso de complejidad O(n) en el peor de los casos (donde n es el orden del grafo), pues esta es la máxima longitud posible de una lista de sucesores.

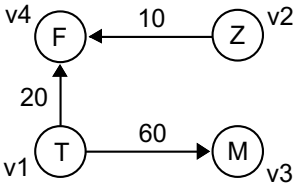
6.8.3. Listas Encadenadas de Adyacencias

Dos inconvenientes se presentan en las representaciones anteriores. El primero, es que se debe manejar y reservar una cantidad fija de memoria para representar el grafo (MAX posiciones) y, aunque puede mejorarse la implementación para que sea posible aumentar o disminuir este espacio en ejecución, esto resulta de todos modos un proceso lento. El segundo problema es el costo que tiene calcular, para la segunda representación vista, la lista de predecesores de un vértice. Esta operación es muy importante en algunos procesos y en ese caso resulta de complejidad cuadrática.

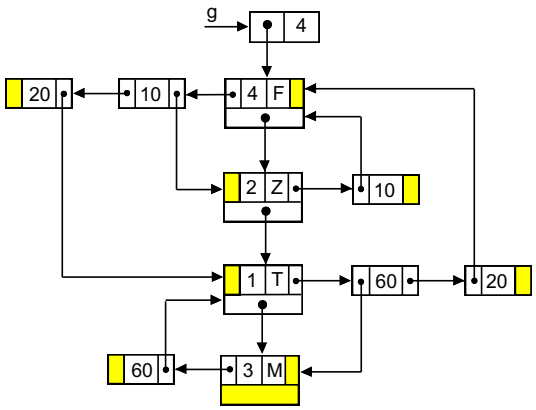
La estructura que se propone se ilustra mediante el siguiente ejemplo. La formalización del esquema de representación, la declaración de las estructuras de datos y la implementación de las operaciones se proponen más adelante como ejercicio.

Ejemplo 6.17:

Para el grafo de la figura:



Las estructuras de datos que lo representan son:



6.8.4. Listas de Arcos

Otra manera de representar internamente un grafo dirigido es mediante 3 listas: la primera con los vértices y sus contenidos, la segunda con los vértices marcados y la tercera con los arcos.

- vértices = < [v_1, $info_1$], ..., [v_n, $info_n$] >
- marcados = < v_i, v_k, ..., >
- arcos = < [v_k, v_i, c_{ki}], ..., [v_p, v_r, c_{pr}] >

La formalización del esquema de representación, la declaración de las estructuras de datos y la implementación de las operaciones se proponen más adelante como ejercicio.

6.8.5. Estructuras de Datos Implícitas

En algunos casos, la representación del grafo puede hacerse de manera implícita, si el problema tiene completamente definida su estructura. Suponga por ejemplo que el mundo en el cual ocurre un problema es un espacio bidimensional cerrado de 5 x 5, donde cada vértice está definido por sus coordenadas (x, y), tal como aparece en la figura 6.17.

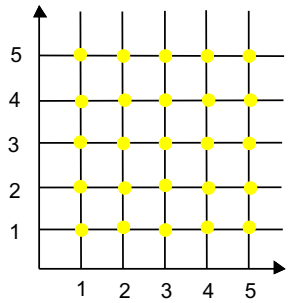


Fig. 6.17 - Espacio bidimensional discretizado

Aunque es perfectamente posible utilizar una representación como las vistas anteriormente, que da la estructura del grafo mostrada en el figura 6.18, también se pueden diseñar unas estructuras de datos especiales, mucho más compactas, que aprovechen la uniformidad topológica del grafo. Algunas operaciones del TAD como agregar o eliminar un arco pierden sentido, porque el grafo tiene una estructura predefinida y estática.

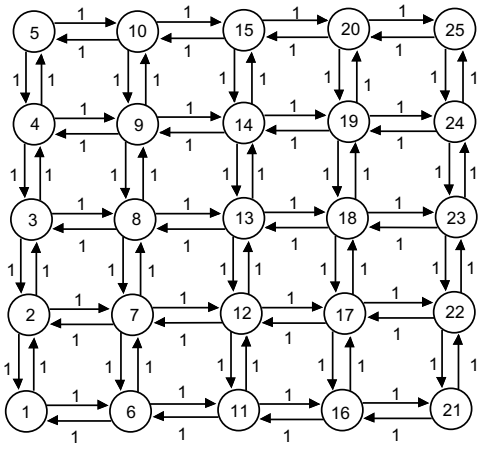


Fig. 6.18 - Grafo asociado con un espacio bidimensional discretizado de 5 x 5

En las estructuras de datos sólo se deben manejar explícitamente los vértices marcados. Las coordenadas de un vértice (su información asociada) y los arcos van implícitos en el número asignado al vértice (su identificador). La declaración de las estructuras de datos es la siguiente:

```
typedef struct
{    int marcado[ 25 ];
} TGrafo, *Grafo ;
```

Algunas de las rutinas que implementan las operaciones del TAD Grafo se presentan a continuación:

```
Grafo inicGrafo( void )
{   int i;
    Grafo g = ( Grafo )malloc( sizeof( TGrafo ) );
    for( i = 0 ; i < 25 ; i++ )
          g->marcado[ i ] = 0;
    return g;
}

int costoArco( Grafo g, int v, int w )
{   int x1 = ( (v - 1) / 5 ) + 1;          /* coordenada en x de v */
    int y1 = ( (v - 1) % 5 ) + 1;          /* coordenada en y de v */
    int x2 = ( (w - 1) / 5 ) + 1;          /* coordenada en x de w */
    int y2 = ( (w - 1) % 5 ) + 1;          /* coordenada en y de w */
    if( ( x1 - 1 == x2 && y1 == y2 ) ||
       ( x1 + 1 == x2 && y1 == y2 ) ||
       ( x1 == x2 && y1 - 1 == y2 ) ||
       ( x1 == x2 && y1 + 1 == y2 ) )
          return 1;
    else
          return -1;
}
```

```
Lista sucesores( Grafo g, int v )
{   int x = ( (v - 1) / 5 ) + 1;            /* coordenada en x de v */
    int y = ( (v - 1) % 5 ) + 1;            /* coordenada en y de v */
    Lista resp = inicLista( );
    if( x - 1 >= 1 ) anxLista( resp, ( ( x - 2 ) * 5 ) + y ) ;
    if( y - 1 >= 1 ) anxLista( resp, ( ( x - 1 ) * 5 ) + ( y - 1 ) ) ;
    if( x + 1 <= 5 ) anxLista( resp, ( ( x ) * 5 ) + y ) ;
    if( y + 1 <= 5 ) anxLista( resp, ( ( x - 1 ) * 5 ) + ( y + 1 ) ) ;
    return resp;

void marcarVertice( Grafo g, int v )
{   g->marcado[ v - 1 ] = 1 ;
}
```

Ejercicios Propuestos

6.61. Implemente y pruebe la operación que retorna una lista con los predecesores de un vértice en un grafo, sobre matrices de adyacencias.

6.62. Implemente y pruebe la operación que retorna una lista con los predecesores de un vértice en un grafo, sobre listas de sucesores.

6.63. Implemente todas las operaciones del TAD Grafo sobre listas de adyacencias. Calcule la complejidad de cada una de ellas. Haga una comparación con las dos primeras representaciones propuestas.

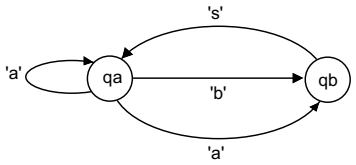
6.64. Implemente todas las operaciones del TAD Grafo sobre listas de arcos. Calcule la complejidad de cada una de ellas. Haga una comparación con las dos primeras representaciones propuestas.

6.65. Autómatas de Estados Finitos

Un **autómata de estados finitos** es una cuádrupla (Q, P, F, q_i), donde:

- Q es un conjunto de **estados**: { q_1,...,q_N }. Para efectos prácticos, se puede ver como el conjunto de enteros { 1, 2, ..., N }.

- P es un conjunto de tripletas: { (q, r, c) | q es un estado, r es un estado, c es un caracter }. Cada una de las tripletas de denomina una **transición**, y se denota como q $\xrightarrow{c}$ r. El caracter asociado con cada transición se llama su **etiqueta**, y corresponde a una letra: 'a' ... 'z'.

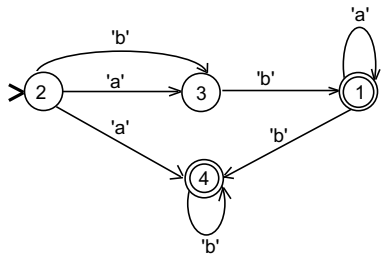


A diferencia de un grafo dirigido, entre dos estados de un autómata puede haber más de una transición en cualquiera de los dos sentidos. Además, pueden existir dos transiciones que salgan del mismo estado y tengan la misma etiqueta.

- F es un subconjunto de Q. Los elementos de F se denominan **estados finales** del autómata, y se marcan de la siguiente manera:

- q_i es un elemento de Q que se denomina estado inicial, y se marca como:

El siguiente es un ejemplo de un autómata de estados finitos:



Para trabajar con este tipo de objeto abstracto, se define el TAD Automata con el siguiente formalismo:

TAD Autómata	
(Q, P, F, q_i)	Q = { q_1,...,q_N }
	P = { (q, r, c) }

a-) Escriba el invariante del TAD Automata

Para la administración de un autómata, el TAD cuenta con las siguientes operaciones analizadoras:

sucesoresEstado	Automata × char × int	→ Lista[int]
primerEstado	Automata	→ int
esEstadoFinal	Automata × int	→ int

Lista sucesoresEstado(Autómata a, char c, int e)
/* Retorna una lista con los estados sucesores de e mediante transiciones etiquetadas con el caracter c */

{ pre: e ∈ Q }
{ post: sucesoresEstado = ⟨ x_1, ..., X_n ⟩ | e $\xrightarrow{c}$ x_i }

int primerEstado(Autómata a)
/* Retorna el estado inicial de a */

{ post: primerEstado = q_i }

int esEstadoFinal(Autómata a, int e)
/* Informa si e es un estado final */

{ pre: e ∈ Q}
{ post: esEstadoFinal = (e ∈ F) }

b-) Utilizando las tres operaciones analizadoras del TAD, desarrolle la siguiente función:

Dada una cadena de caracteres S = "$c_1c_2...c_n$", se dice que S lleva a un autómata de un estado x_0 a otro estado x_n, si existen estados x_1 .... x_{n-1} tales que $x_i \xrightarrow{ci} x_{i+1}$. Por ejemplo, para el autómata de la figura anterior, la cadena "abbbbbb" lleva del estado inicial a un estado final. Escriba una función que recibe una cadena de caracteres y retorna verdadero si lleva el autómata del estado inicial a un estado final.

c-) Diseñe unas estructuras de datos para el TAD Automata, que sean lo más eficientes posible en tiempo y espacio. Especifique el esquema de representación.

d-) Implemente las tres operaciones del TAD antes mencionadas sobre las estructuras de datos del punto anterior. Calcule la complejidad de su solución, y justifique su diseño.

Bibliografía

- [AHO83] Aho, A., Hopcroft, J., Ullman, J., "Data Structures and Algorithms", Cap. 6, 7, Addison-Wesley, 1983.
- [BER62] Berge, C., "Theory of Graphs and Its Applications", Methuen Press, 1962.
- [BER73] Berge, C., "Graphs and Hypergraphs", North-Holland, 1973.
- [BER94] Bergin, J., "Data Abstraction: The Object-Oriented Approach Using C++", Cap. 12, McGraw-Hill, 1994.
- [DAL86] Dale, N., Lilly, S., "Pascal y Estructura de Datos", Cap. 9, McGraw-Hill, 1986.
- [ESA89] Esakov, J., Weiss, T., "Data Structures: An Advanced Approack Using C", Cap. 7, Prentice-Hall, 1989.
- [EVE79] Even, S., "Graph Algorithms", Computer Science Press, 1979.
- [FEL88] Feldman, M., "Data Structures with Modula-2", Cap. 6, Prentice-Hall, 1988.
- [HOR83] Horowitz, E., "Fundamentals of Data Structures", Cap. 6, Computer Science Press, 1983.
- [KNU73] Knuth, D., "The Art of Computer Programming. Vol. 1 - Fundamental Algorithms", 2da edición, Addison-Wesley, 1973.
- [KRU87] Kruse, R., "Data Structures & Program Design", Cap.10, Prentice-Hall, 1987.
- [LIP87] Lipschutz, S., "Estructura de Datos", Cap. 8, McGraw-Hill, 1987.
- [MAR86] Martin, J., "Data Types and Data Structures", Cap. 9, Prentice-Hall, 1986.
- [REI77] Reingold, E., Nievergelt, N., "Combinatorial Algorithms: Theory and Practice", Prentice-Hall, 1977.
- [TAR83] Tarjan, R., "Data Structures and Network Algorithms", Society for Industrial and Applied Mathematics, 1983.
- [TEN93] Tenenbaum, A., Langsam, Y., "Estructuras de Datos en C", Cap. 8, Prentice Hall, 1993.
- [TRE76] Tremblay, J., Sorenson, P., "An Introduction to Data Structures with Applications", Cap. 5, McGraw-Hill, 1976.

CAPITULO 7

ESTRUCTURAS DE ACCESO DIRECTO: TABLAS DE *HASHING*

En este capítulo se presentan unas estructuras de datos que responden muy eficientemente a las operaciones de búsqueda en un conjunto de datos, aunque no tienen un adecuado desempeño para otro tipo de operaciones. Esta estructura se conoce como una tabla de *hashing*.

7.1. Motivación

Suponga que se quiere almacenar en una estructura de datos un conjunto de elementos cuyas llaves de acceso son valores entre 0 y 9.999. La representación más simple y eficiente se hace a través de un vector, en el cual aparece, en la posición k, el elemento con llave de acceso k. Esto permite un acceso a la información en O(1) cuando se da su llave.

Piense ahora en el mismo problema, pero donde los 10.000 elementos que se quieren almacenar tienen llaves de acceso con valores entre 0 y 99.999.999. Este es el caso de una empresa con N empleados con acceso por su número de cédula. Con las estructuras estudiadas hasta este momento es imposible tener acceso a un elemento de dicha estructura, dada su llave, en menos de O($\log_k N$).

El objetivo de una estructura de acceso directo es retomar la idea del vector y definir una manera rápida de proyectar el valor de una llave en una posición del vector, de manera que se tenga un acceso muy eficiente de la información, aunque otras operaciones como recorridos, etc. resulten ineficientes.

La idea de las tablas de *hashing* surgió a mediados de los años 50, en IBM, como un esquema de representación de información en memoria secundaria para hacer consultas rápidas. Knuth [KNU73] menciona los aportes de P. Luhn (IBM-1953), A. Dumey (1956), A. Ershov (Rusia-1958) y R. Morris (1968), como la base del desarrollo de este tipo de estructuras de datos. El término *hashing* fue popularizado por R. Morris, y se ha convertido, desde mediados de los años 60, en un nombre estándar para este tipo de estructuras de acceso directo. La traducción al español de dicho término no es evidente, puesto que surgió de la jerga informática del momento, tratando de explicar con el verbo *to hash* (cortar, picar) la acción de partir la llave y sacar de allí la información que permita localizar el elemento asociado.

7.2. Definiciones y Conceptos Básicos

Una **tabla de *hashing*** es una estructura de datos de acceso directo, en la cual cada elemento tiene asociada una llave por medio de la cual se consulta. El formalismo escogido para expresar el objeto abstracto es el siguiente:

$$< llave_1 \rightarrow elem_1, llave_2 \rightarrow elem_2, ..., llave_n \rightarrow elem_n >$$

Una tabla sin elementos se denomina **vacía**, y se expresa mediante el símbolo $\otimes$, para no utilizar el mismo formalismo usado en el capítulo 2 para las listas vacías. El número de elementos presentes en la tabla se denomina su **tamaño**.

Una **llave** es una cadena de caracteres alfanuméricos, con un significado especial en el mundo en el cual ocurre el problema. Las llaves son únicas al interior de la tabla y son el único medio para tener acceso a la información asociada. El conjunto de todas las llaves posibles se denomina el **espacio de llaves** de la tabla. Al interior de una tabla de *hashing* no existe una noción de orden, en el sentido de que no se puede hacer un recorrido secuencial de la información, no hay un sucesor ni un predecesor de un elemento, no hay un primero ni un último. No es posible ejecutar operaciones como traer la siguiente llave, o determinar cuántos elementos hay en un rango de llaves. El único acceso posible es por la llave y la gran cualidad de una tabla de *hashing* es la eficiencia con la cual es capaz de hacer acceso directo a la información bajo esa condición.

El espacio físico de representación de la información se conoce como el **área primaria** de la tabla. Allí se colocan los elementos de la estructura con una cierta disposición, de manera que las operaciones de búsqueda sean capaces de llegar rápidamente al elemento que se quiere localizar. Esta área tiene un tamaño fijo, denominado la **capacidad** de la tabla y se denota por M. En el formalismo, este valor se hace explícito de la siguiente manera:

$$< llave_1 \rightarrow elem_1, llave_2 \rightarrow elem_2, ..., llave_n \rightarrow elem_n > [\ M\ ]$$

La capacidad de la tabla es una decisión de diseño, que se debe determinar basados en el número esperado de elementos que debe almacenar. La **dirección** de un elemento en el área primaria es la posición que éste ocupa, y corresponde a un valor entre 0 y M-1. El **factor de carga** de una tabla de *hashing* se define como el tamaño de la tabla sobre su capacidad, y es una medida de lo saturada que ésta se encuentra.

Una **función de *hashing*** es una función h que proyecta un valor del espacio de llaves a una dirección del área primaria, como se sugiere en la figura 7.1. Esta función es la base del esquema de acceso a la información.

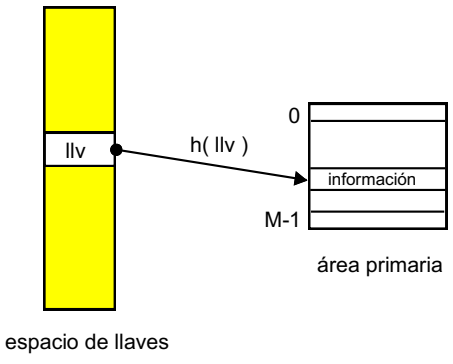


Fig. 7.1 - Función de *hashing*

Resulta conveniente utilizar una tabla de *hashing* cuando el espacio de llaves es mucho mayor que el área primaria, y las llaves que se deben almacenar en la estructura tienen una distribución altamente no uniforme.

Ejemplo 7.1:

El código que se le asigna a un estudiante de la Universidad está formado por la inicial del nombre, la inicial del apellido y el número de carnet, cuyos dígitos corresponden al año de entrada, el semestre y un consecutivo de 4 dígitos. El espacio de llaves en este caso tiene un tamaño de 1.458.000.000 códigos potenciales diferentes, calculado de la siguiente manera:

27 (inicial nombre) * 27 (inicial apellido) * 100 (año) * 2 (semestre)* 10.000 (consecutivo)

En la Facultad de Ingeniería hay aproximadamente 3.000 estudiantes, cuyos códigos están distribuidos de manera no uniforme sobre dicho espacio de llaves. El área primaria debería tener una capacidad superior a 3.000, y la función de *hashing* debería ser tal, que al código de un estudiante le asociara una dirección en el área primaria, donde estaría localizada su información.

Una posible función de *hashing*, para este caso, sería sumar todos los dígitos del carnet, y multiplicar dicho resultado por el código ASCII de las iniciales. Luego, a este valor se le aplicaría la función módulo M. Esta función siempre retorna un valor entre 0 y 2.999 (para M=3.000), direcciones válidas del área primaria. Por ejemplo:

- h("VD9113984") = 86 * 68 * 35 = 204.680 % 3000 = 680
- h("CM9113578") = 67 * 77 * 34 = 175.406 % 3000 = 1406

Una función de *hashing* no preserva el orden, de manera que si $llave_1 < llave_2$ no necesariamente se cumple que $h(llave_1) < h(llave_2)$. Esto implica que en el área primaria la información va a quedar en un orden indeterminado con respecto al orden de las llaves.

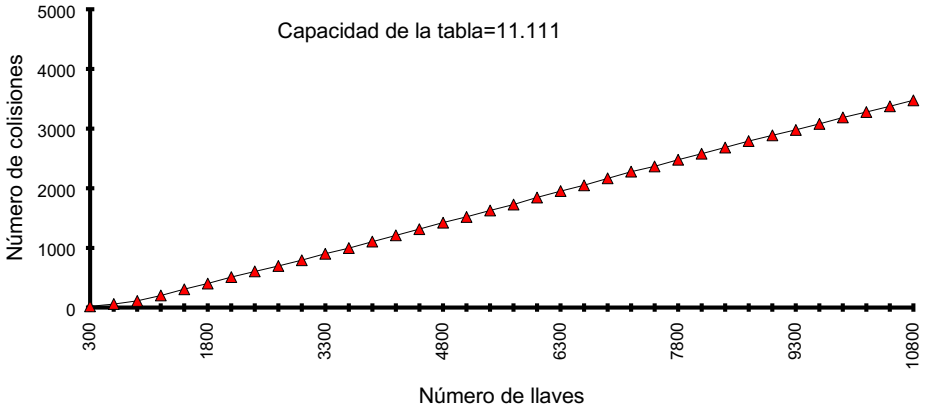
Cuando dos llaves distintas son proyectadas sobre la misma dirección del área primaria, se habla de una **colisión**. La eficiencia de una tabla de *hashing* radica básicamente en su estrategia para resolver estos conflictos. Existen varias formas de hacer que si dos llaves llegan a una misma dirección, se puedan almacenar los dos elementos asociados en la estructura, tal como se presenta en la sección de implementación del TAD. El planteamiento más sencillo es asociar una lista de elementos con cada dirección del área primaria.

• colisión(llv_1, llv_2) ssi $llv_1 \neq llv_2 \wedge h(llv_1) = h(llv_2)$

El desempeño de una tabla de *hashing* comienza a disminuir a medida que aumenta el factor de carga, puesto que crece el número de colisiones, y se debe recurrir a las estructuras auxiliares de datos sobre las cuales se hace una búsqueda más lenta (de orden lineal en estructuras lineales y de orden logarítmico en estructuras arborescentes). Por esta razón el área primaria se debe definir desde un comienzo con una capacidad extra aproximada del 20%, para asegurar un buen desempeño cuando el factor de carga aumenta.

Ejemplo 7.2:

En la gráfica se puede apreciar la manera como aumenta el número de colisiones a medida que el factor de carga crece. Esto repercute directamente en el tiempo de acceso a la información. Para obtener estos resultados se construyó un generador aleatorio de llaves, se utilizó como función de *hashing* la presentada en el ejemplo 7.1, y se trabajó sobre la segunda implementación propuesta para este tipo de estructuras (distribución en área primaria):



Note como el número de colisiones aumenta de manera aproximadamente lineal, permitiendo el acceso a la información a una velocidad razonable (en promedio, una de cada 3 llaves tiene colisión), como se muestra en la siguiente tabla:

# llaves insertadas	# total colisiones	# colisiones por llave insertada
1000	160	0.16
2000	481	0.32
3000	813	0.33
4000	1143	0.33
5000	1462	0.32
6000	1800	0.34
7000	2121	0.32
8000	2448	0.33
9000	2777	0.33
10000	3094	0.32
11000	3422	0.33

⏎

Otros dos elementos son decisivos en el comportamiento de la tabla: la función de *hashing* h y su capacidad M. La función no puede ser demasiado compleja, puesto que esto incide en el tiempo de ejecución, pero debe distribuir adecuadamente las llaves. No se espera que evite completamente las colisiones, pero si debe evitar los excesos. Para seleccionar el valor de M, es conveniente escoger un número primo. En la práctica se ha observado que es suficiente con seleccionar M de manera que no tenga divisores primos menores que 20 [KNU73].

Las tablas de *hashing* se utilizan sobre todo en memoria secundaria, donde el número de accesos a la estructura de datos resulta un factor crítico, haciendo que se descarten estructuras de datos como árboles AVL, en las cuales una búsqueda puede implicar múltiples accesos al disco.

Una tabla de *hashing* es una estructura con bajo desempeño en procesamiento secuencial de un rango de llaves, en recorridos ordenados por llave y en búsquedas con llaves incompletas. En esos casos es necesario utilizar varias estructuras de datos simultáneas, para responder eficientemente a todas las operaciones, como se propone más adelante en los ejercicios.

7.3. El TAD TablaH

TAD TablaH[TipoH]

< $llave_1 \rightarrow elem_1$, $llave_2 \rightarrow elem_2$, ..., $llave_n \rightarrow elem_n$ > [M]

{ inv: $llave_i = llave_k$ ssi i = k }

Constructoras:

•	inicTablaH:	int	→ TablaH

Modificadoras:

•	insTablaH:	TablaH X char * X TipoH	→ TablaH
•	elimTablaH:	TablaH X char *	→ TablaH

Analizadoras:

•	infoTablaH:	TablaH X char *	→ TipoH
•	estaTablaH:	TablaH X char *	→ int

Persistencia:

•	cargarTablaH:	FILE *	→ TablaH
•	salvarTablaH:	TablaH X FILE *	

Destructora:

•	destruirTablaH:	TablaH

```
TablaH inicTablaH( int cap )
/* Crea y retorna una tabla vacía con capacidad cap */

{ post: inicTablaH = ⊗ [cap] }
```

```
void insTablaH( TablaH t, char *llv, TipoH elem )
/* Inserta en la tabla el elemento elem con llave asociada llv */

{ pre: llv ∉ t }
{ post: t = < llave1→elem1, llave2→elem2, ..., llaven→elemn, llv→elem > [ M ] }
```

```
void elimTablaH( TablaH t, char *llv )
/* Elimina de la tabla el elemento con llave asociada llv */

{ pre: t = < llave1→elem1, llave2→elem2, ..., llaven→elemn > [ M ], llv = llavei }
{ post: t = < llave1→elem1, ..., llavei-1→elemi-1, llavei+1→elemi+1, ..., llaven→elemn > [ M ] }
```

```
TipoH infoTablaH( TablaH t, char *llv )
/* Retorna el elemento asociado con la llave llv */

{ pre: t = < llave1→elem1, llave2→elem2, ..., llaven→elemn > [ M ], llv = llavei }
{ post: infoTablaH = elemi }
```

```
int estaTablaH( TablaH t, char *llv )
/* Informa si un elemento con llave llv aparece en la tabla */

{ post: estaTablaH = ( llv ∈ t ) }
```

```
TablaH cargarTablaH( FILE *fp )
/* Construye una tabla a partir de la información de un archivo */

{ pre: el archivo está abierto y es estructuralmente correcto, de acuerdo con el esquema de persistencia }
{ post: se ha construido la tabla que corresponde a la imagen de la información del archivo }
```

```
void salvarTablaH( TablaH t, FILE *fp )
/* Salva la tabla en un archivo */

{ pre: el archivo está abierto }
{ post: se ha hecho persistir la tabla en el archivo de acuerdo con el esquema de persistencia }
```

```
void destruirTablaH( TablaH t )
/* Destruye el objeto abstracto, retornando toda la memoria ocupada por éste */

{ post: la tabla t no tiene memoria reservada }
```

7.4. Implementación del TAD TablaH

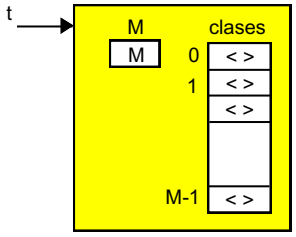
En esta parte se presentan varias maneras diferentes de resolver los problemas de implementación de una tabla de *hashing*. Soluciones a los problemas de eliminar un elemento de manera eficiente, resolver un conflicto, buscar una llave, etc., se ilustran a través de tres estructuras de datos distintas.

7.4.1. Listas de Clases de Equivalencia

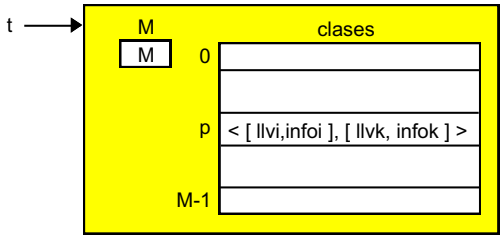
Una función de *hashing* puede ser vista como una relación de equivalencia, la cual divide el espacio de llaves en clases de equivalencia: llv1 *R* llv2 ssi h(llv1) = h(llv2). En esta representación, cada una de las posiciones del área primaria está constituida por una lista de parejas [llave, elemento], ordenada ascendentemente por llave, con los elementos que pertenecen a la clase de equivalencia respectiva. Cuando se presenta una colisión, ésta se resuelve incluyendo el nuevo elemento en la lista.

El esquema de representación es el siguiente:

- La tabla vacía t = ⊗ [M] se representa con un apuntador a una estructura que tiene un vector de M posiciones, en cada uno de los cuales hay una lista ordenada vacía:



- La tabla t = < llave$_1$→elem$_1$, llave$_2$→elem$_2$, ..., llave$_n$→elem$_n$ > [M] se representa con un apuntador a una estructura que tiene un vector de M posiciones, en cada una de los cuales hay una lista ordenada de parejas con todos los elementos que pertenecen a la clase de equivalencia respectiva:



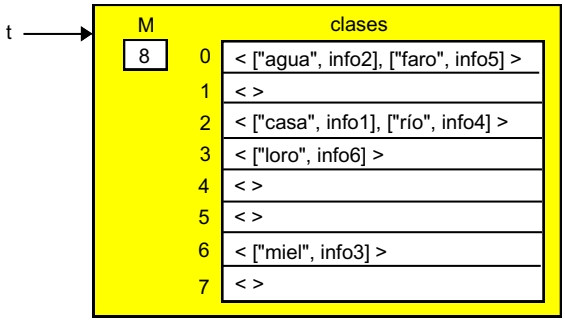
Dos elementos info$_i$, info$_k$ de la tabla se encuentran en la posición p del vector, de la manera como se muestra en la figura, si se cumple que h(llv$_i$) = h(llv$_k$) ∧ llv$_i$ < llv$_k$.

Ejemplo 7.3:

Para la tabla:

t = < "casa"→info1, "agua"→info2, "miel"→info3, "río"→info4, "faro"→info5, "loro"→info6 > [8]

Dada alguna función de *hashing*, las estructuras de datos que la representan son:



⏎

La declaración de las estructuras de datos para esta primera implementación de tablas de *hashing* es:

```
typedef struct Pareja
{   char *llave;          /* Llave de acceso */
    TipoH info;           /* Informacion asociada con la llave */
} TipoLO;

typedef struct
{   int M;                /* Capacidad de la tabla */
    ListOrd *clases;      /* Vector dinamico de listas ordenadas de parejas */
} TTablaH, *TablaH;
```

- La rutina de inicialización de una tabla de *hashing* pide memoria para ella y para el área primaria, de acuerdo con el parámetro que envía el usuario. Luego, hace un ciclo para inicializar las listas de clases de equivalencia en vacío.

```
TablaH inicTablaH( int cap )
{   int i;
    TablaH t = ( TablaH )malloc( sizeof( TTablaH ) );
    t->M = cap;
    t->clases = ( ListOrd * )calloc( cap, sizeof( ListOrd ) );
    for( i = 0; i < t->M; i++ )
        t->clases[ i ] = inicListOrd( );
    return t;
}
```

- En esta representación, para insertar un elemento en la tabla, se localiza primero la dirección que le corresponde a la llave en el área primaria de acuerdo con la función de *hashing*. Luego, se agrega a la respectiva lista ordenada de clases de equivalencia un nodo con la pareja [llave, elemento].

```
void insTablaH( TablaH t, char *llv, TipoH elem )
{   struct Pareja nodo;
    nodo.llave = ( char * )malloc( strlen( llv ) + 1 );
    strcpy( nodo.llave, llv );
    nodo.info = elem;
    insListOrd( t->clases[ h( t, llv ) ], nodo );
}
```

- Eliminar un elemento se reduce a suprimir, de la lista de equivalencia respectiva, el elemento que tiene asociada la llave.

```
void elimTablaH( TablaH t, char *llv )
{   elimListOrd( t->clases[ h( t, llv ) ], llv );
}
```

- Para localizar un elemento en la tabla, se toma su llave y, en la dirección del área primaria que se obtiene con la función de *hashing*, se hace una búsqueda secuencial sobre la lista ordenada que allí se encuentra almacenada. Una vez encuentra la pareja [llave, información], la rutina retorna el elemento asociado.

```
TipoH infoTablaH( TablaH t, char *llv )
{   int i;
    int pos = h( t, llv );
    for( i = 1; i <= longListOrd( t->clases[ pos ] ); i++ )
        if( strcmp( infoListOrd( t->clases[ pos ], i ).llave, llv ) == 0 )
            break;
    return infoListOrd( t->clases[ pos ], i ).info;
}
```

- Para decidir si una llave se encuentra presente en una tabla, se sigue un proceso parecido al anterior:

```
int estaTablaH( TablaH t, char *llv )
{   int i;
    int pos = h( t, llv );
    for( i = 1; i <= longListOrd( t->clases[ pos ] ); i++ )
        if( strcmp( infoListOrd( t->clases[ pos ], i ).llave, llv ) == 0 )
            return TRUE;
    return FALSE;
}
```

- La destructora recorre primero el área primaria destruyendo las listas de parejas y retorna, luego, toda la memoria reservada:

```
void destruirTablaH( TablaH t )
{   int i;
    for( i = 0; i < t->M; i++ )
        destruirListOrd( t->clases[ i ] );
    free( t->clases );
    free( t );
}
```

- Para las operaciones de persistencia se diseña el siguiente esquema, el cual utiliza un archivo de texto en el que aparecen secuencialmente las parejas [llave, información], con una marca especial al final para indicar el final del contenido válido:

$< llave_1 \rightarrow elem_1, llave_2 \rightarrow elem_2, ..., llave_n \rightarrow elem_n >$ [M]	M llave1 elem1 llave2 elem2 .... llaven elemn ****** 0

Las dos rutinas que cargan y salvan una tabla de *hashing* son:

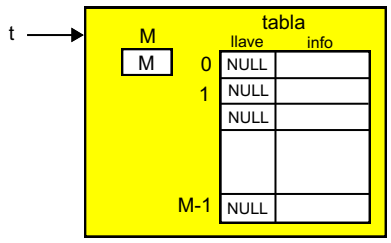
```
TablaH cargarTablaH( FILE *fp )
{   char llave[ 12 ];
    TablaH t;
    int valor, cap;
    fscanf( fp, "%d", &cap );
    t=inicTablaH( cap );
    fscanf( fp, "%s %d", llave, &valor );
    while( llave[ 0 ] != '*' )
    {   insTablaH( t, llave, valor );
        fscanf( fp, "%s %d", llave, &valor );
    }
    return t;
}

void salvarTablaH( TablaH t, FILE *fp )
{   int i;
    fprintf( fp, "%d\n", t->M );
    for( i = 0; i < t->M; i++ )
        salvarListOrd( t->clases[ i ], fp );
    fprintf( fp, "******* 0\n");
}
```

7.4.2. Distribución en Área Primaria

En esta implementación las colisiones se resuelven utilizando algún espacio libre en la misma área primaria, sin necesidad de manejar una estructura adicional. Existen diversas variantes que se nombran más adelante, pero, para efectos de la presentación, se escoge el sistema más simple.

El esquema de representación es el siguiente:

- La tabla vacía t = ⊗ [M] se representa con un apuntador a una estructura que tiene un vector de M posiciones, en cada uno de los cuales hay un elemento de la tabla. Para indicar una posición vacía se coloca el campo llave en NULL.



- La tabla t = < $llave_1 \rightarrow elem_1$, $llave_2 \rightarrow elem_2$, ..., $llave_n \rightarrow elem_n$ > [M], se representa con un apuntador a una estructura que tiene un vector de M posiciones. Para insertar un elemento para el cual no existe conflicto, se coloca la llave y la información asociada en la posición del vector definida por la función de *hashing*.

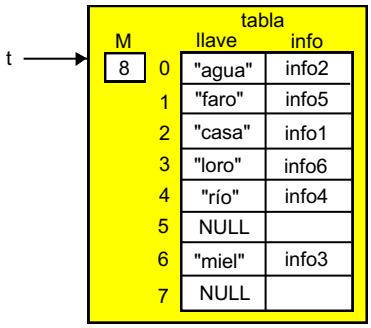
  Si en la tabla t se encuentra la llave llv_i y al tratar de agregar la llave llv_k aparece una colisión con la primera (i.e. h(llv_i) = h(llv_k)), la información de la segunda llave se coloca en la siguiente posición libre del área primaria, recorriéndola secuencialmente posición por posición.

Ejemplo 7.4:

Para la tabla:

t = < "casa"→info1, "agua"→info2, "miel"→info3, "río"→info4, "faro"→info5, "loro"→info6 > [8]

Dada la misma función de *hashing* del ejemplo anterior, las estructuras de datos que representan la tabla son:



⏎

La declaración de las estructuras de datos es:

```
typedef struct
{   int M;                  /* Capacidad de la tabla */
    struct Pareja
    {     char *llave;      /* Llave de acceso a la información */
          TipoH info;       /* Elemento que se guarda en la tabla */
    } *tabla;               /* Vector dinamico con parejas llave-información */
} TTablaH, *TablaH;
```

- Para crear una tabla vacía se pide la memoria correspondiente al área primaria, que corresponde a un vector de parejas [llave - información]. Luego, se coloca el valor NULL en el campo llave de cada casilla, para indicar que se encuentra vacía.

```
TablaH inicTablaH( int cap )
{   int i;
    TablaH t = ( TablaH )malloc( sizeof( TTablaH ) );
    t->M = cap;
    t->tabla = ( struct Pareja * )calloc( cap, sizeof( struct Pareja ) );
    for( i = 0; i < t->M; i++ )
          t->tabla[ i ].llave = NULL;
    return t;
}
```

- La rutina que agrega un elemento a la tabla sigue los siguientes pasos: primero, establece el punto en el cual debería encontrarse el elemento en el área primaria. Si esta posición se encuentra libre coloca allí la llave y el elemento asociado. Si encuentra un conflicto, hace una búsqueda secuencial a partir de ese punto, hasta que encuentra un lugar libre, en el cual puede situar el nuevo elemento. Esta rutina ve el área primaria como un vector circular.

```
void insTablaH( TablaH t,char *llv,TipoH elem )
{   int pos;
    for( pos = h( t, llv ); t->tabla[ pos ].llave != NULL; pos = ( pos == t->M - 1 ) ? 0 : pos + 1 );
    t->tabla[ pos ].llave = ( char * )malloc( strlen( llv ) + 1 );
    strcpy( t->tabla[ pos ].llave, llv );
    t->tabla[ pos ].info = elem;
}
```

- Para eliminar un elemento, se hace una búsqueda igual a la que se requiere en una consulta, y al encontrar la posición en la cual se encuentra el elemento, se marca esa casilla como vacía, colocando la llave en NULL.

```
void elimTablaH( TablaH t,char *llv )
{   int pos;
    for( pos = h( t, llv ); strcmp( t->tabla[ pos ].llave, llv ) !=0; pos = ( pos == t->M - 1 ) ? 0 : pos + 1 );
    free( t->tabla[ pos ].llave );
    t->tabla[ pos ].llave = NULL;
}
```

- La rutina que busca un elemento en la tabla sigue los siguientes pasos: primero, establece el punto en el cual debería encontrarse el elemento en el área primaria. Luego, si no se encuentra allí, hace una búsqueda secuencial desde esa posición hasta que lo encuentra. Si llega al final del área primaria, comienza de nuevo la búsqueda desde la dirección 0. Cuando localiza la llave, retorna la información que tiene asociada.

```
TipoH infoTablaH( TablaH t,char *llv )
{   int pos;
    for( pos = h( t, llv ); strcmp( t->tabla[ pos ].llave, llv) != 0; pos = ( pos == t->M - 1 ) ? 0 : pos + 1 );
    return( t->tabla[ pos ].info );
}
```

- La operación que informa si una llave se encuentra presente en una tabla utiliza un proceso parecido al anterior, pero verificando que no se haya quedado en un ciclo durante la búsqueda. Esta rutina tiene una complejidad de O(M), en el peor de los casos (la llave no está presente en la tabla), lo cual la hace muy ineficiente.

```
int estaTablaH( TablaH t, char *llv )
{   int primero;
    int pos = h( t, llv );
    if( strcmp( t->tabla[ pos ].llave, llv) == 0 )
        return TRUE;
    else
    {   for( primero = pos++; strcmp( t->tabla[ pos ].llave, llv) != 0 && primero != pos;
                                pos = ( pos == t->M - 1 ) ? 0 : pos + 1 );
        return primero != pos;
    }
}
```

- Para destruir la tabla se recorre el área primaria, retornando la memoria ocupada por las llaves. Luego, se devuelve el resto de la memoria ocupada:

```
void destruirTablaH( TablaH t )
{   int i;
    for( i = 0; i < t->M; i++ )
        if( t->tabla[ i ].llave != NULL )
            free( t->tabla[ i ].llave );
    free( t->tabla );
    free( t );
}
```

- Para el mismo esquema de persistencia planteado en la implementación anterior, las rutinas que salvan y cargan una tabla son:

```
TablaH cargarTablaH( FILE *fp )
{   char llave[ 12 ];
    TablaH t;
    int valor, cap;
    fscanf( fp, "%d", &cap );
    t=inicTablaH( cap );
    fscanf( fp, "%s %d", llave, &valor );
    while( llave[ 0 ] != '*' )
    {   insTablaH( t, llave, valor );
        fscanf( fp, "%s %d", llave, &valor );
    }
    return t;
}
```

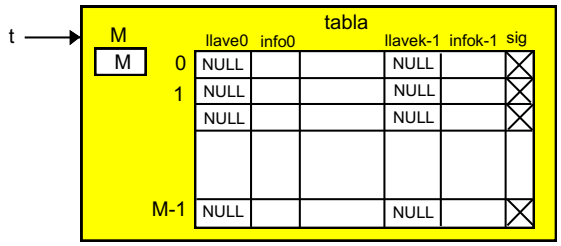
```
void salvarTablaH( TablaH t, FILE *fp )
{   int i;
    fprintf( fp, "%d\n", t->M );
    for( i = 0; i < t->M; i++ )
        if( t->tabla[ i ].llave != NULL )
            fprintf( fp, "%s %d\n", t->tabla[ i ].llave, t->tabla[ i ].info );
    fprintf( fp, "******* 0\n");
}
```

7.4.3. Bloques con Área de Desbordamiento

Esta representación sigue una idea utilizada en el manejo de estructuras de datos dinámicas en memoria secundaria. Esta consiste en reservar un espacio fijo asociado con cada dirección del área primaria, para colocar allí un número limitado de elementos que colisionan. Si en algún momento este espacio resulta insuficiente, se habla de un **desbordamiento** del bloque. Para resolverlo, se utiliza un bloque del mismo tamaño y se encadena con el bloque original. Al interior de los bloques los elementos no tienen un orden específico.

El esquema de representación es el siguiente:

- La tabla vacía t = ⊗ [M], con bloques de tamaño K, se representa con un apuntador a una estructura que tiene un vector de M posiciones, en cada uno de los cuales hay un bloque de K elementos de la tabla. Para indicar una posición vacía se coloca el campo llave del elemento en NULL. El indicador de área de desbordamiento de cada posición se inicializa también en NULL.



- La tabla t = < llave$_1$→elem$_1$, llave$_2$→elem$_2$, ..., llave$_n$→elem$_n$ > [M], con bloques de tamaño K, se representa con un apuntador a una estructura que tiene un vector de M bloques de K posiciones cada uno. Para insertar un elemento para el cual no existe conflicto, se coloca la llave y la información asociada en la primera posición del bloque respectivo.

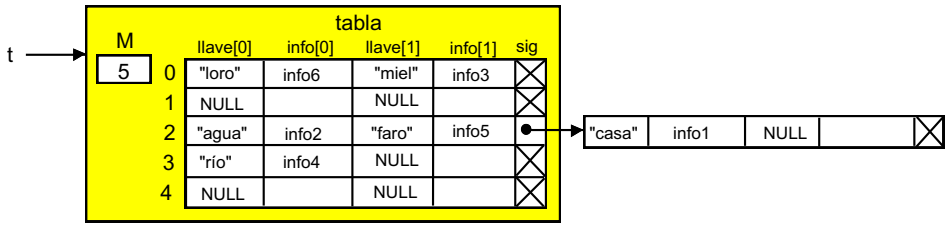
  En caso de aparecer un conflicto, se resuelve colocando el nuevo elemento y su llave en una posición libre del bloque señalado por la función de *hashing*. Si el bloque se encuentra lleno, se crea un nuevo bloque de K elementos y se encadena con el bloque original.

Ejemplo 7.5:

Para la tabla:

t = < "casa"→info1, "agua"→info2, "miel"→info3, "río"→info4, "faro"→info5, "loro"→info6 > [5]

Suponiendo alguna función de *hashing* dada, las estructuras de datos que la representan, con bloques de tamaño K = 2 son:



Cuando K es igual a 1, se trata de una lista sencillamente encadenada, implementación parecida a la primera presentada en este capítulo (listas de clases de equivalencia).

La declaración de las estructuras de datos es:

```
struct Pareja
{   char *llave;            /* Llave de acceso a la información */
    TipoH info;             /* Información asociada con la llave */
};

struct Bloque
{   struct Pareja elem[ K ];    /* Vector de K parejas [ llave - información ] */
    struct Bloque *sig;         /* Encadenamiento al siguiente bloque */
};

typedef struct
{   int M;                  /* Capacidad de la tabla */
    struct Bloque *tabla;   /* Vector dinámico para el área primaria */
} TTablaH, *TablaH;
```

- La rutina que inicializa la tabla debe colocar en NULL el campo llave de cada una de las K entradas de los bloques:

```
TablaH inicTablaH( int cap )
{   int i, j;
    TablaH t = ( TablaH )malloc( sizeof( TTablaH ) );
    t->M = cap;
    t->tabla = ( struct Bloque * )calloc( cap, sizeof( struct Bloque ) );
    for( i = 0; i < t->M; i++ )
    {       for( j = 0; j < K; j++ )
                t->tabla[ i ].elem[ j ].llave = NULL;
            t->tabla[ i ].sig = NULL;
    }
    return t;
}
```

- Para insertar un elemento en la tabla se deben seguir los siguientes pasos: (1) calcular la dirección que le corresponde en el área primaria, (2) buscar una entrada libre en el bloque principal, o en un bloque de la zona de desbordamiento asociada con la dirección calculada, (3) colocar en ese espacio la llave y el elemento asociado. En el paso 2 se debe tener en cuenta el caso en el cual se debe crear un nuevo bloque en la zona de desbordamiento.

```
void insTablaH( TablaH t, char *llv, TipoH elem )
{   int i;
    struct Bloque *pb, *pa = NULL;
    for( pb = &t->tabla[ h( t, llv ) ]; pb != NULL; pa = pb, pb = pb->sig )
        for( i = 0; i < K; i++ )
            if( pb->elem[ i ].llave == NULL )
            {   pb->elem[ i ].llave = ( char * )malloc( strlen( llv ) + 1 );
                strcpy( pb->elem[ i ].llave, llv );
                pb->elem[ i ].info = elem;
                return;
            }
    pb = ( struct Bloque * )calloc( K, sizeof( struct Bloque ) );
    pa->sig = pb;
    pb->sig = NULL;
    pb->elem[ 0 ].llave = ( char * )malloc( strlen( llv ) + 1 );
    strcpy( pb->elem[ 0 ].llave, llv );
    pb->elem[ 0 ].info = elem;
    for( i = 1; i < K; i++ )
        pb->elem[ i ].llave = NULL;
}
```

- Para eliminar un elemento se debe localizar su posición en la tabla, siguiendo el mismo esquema de una consulta. Luego, es suficiente con marcar esa entrada como vacía. No vale la pena destruir bloques que ya no se utilicen en la zona de desbordamiento, ni colocar las entradas realmente ocupadas al comienzo de los bloques, a menos que exista un problema crítico de espacio.

```
void elimTablaH( TablaH t, char *llv )
{   int i;
    struct Bloque *pb;
    for( pb = &t->tabla[ h( t, llv ) ]; 1; pb = pb->sig )
        for( i = 0; i < K; i++ )
            if( strcmp( pb->elem[ i ].llave, llv ) == 0 )
            {   free( pb->elem[ i ].llave );
                pb->elem[ i ].llave = NULL;
                return;
            }
}
```

- La búsqueda se limita a localizar la dirección que le corresponde al elemento en la tabla, a partir de su llave, y luego hacer un recorrido secuencial al interior de cada bloque hasta localizar la información buscada.

```
TipoH infoTablaH( TablaH t, char *llv )
{   int i;
    struct Bloque *pb;
    for( pb = &t->tabla[ h( t, llv ) ]; 1; pb = pb->sig )
        for( i = 0; i < K;i++ )
            if( strcmp( pb->elem[ i ].llave, llv ) == 0 )
                return pb->elem[ i ].info;
}
```

- El proceso que establece si una llave se encuentra presente en una tabla, localiza la entrada en el área primaria utilizando la función de *hashing*, y, luego, hace una búsqueda secuencial al interior del bloque y al interior de cada uno de los bloques que se encuentran encadenados a dicha entrada.

```
int estaTablaH( TablaH t, char *llv )
{   int i;
    struct Bloque *pb;
    for( pb = &t->tabla[ h( t, llv ) ]; pb != NULL; pb = pb->sig )
        for( i = 0; i < K;i++ )
            if( strcmp( pb->elem[ i ].llave, llv ) == 0 )
                return TRUE;
    return FALSE;
}
```

- Para destruir una tabla, primero, se debe entrar por cada dirección del área primaria y liberar los bloques encadenados. Luego, se libera el vector de bloques del área primaria, y, por último, el registro que representa la tabla completa.

```
void destruirTablaH( TablaH t )
{   int i, j, primero;
    struct Bloque *pb;
    for( i = 0; i < t->M; i++ )
        for( primero = TRUE, pb = &t->tabla[ i ]; pb != NULL; pb = pb->sig )
        {   for( j = 0; j < K; j++ )
                if( pb->elem[ j ].llave != NULL )
                    free( pb->elem[ j ].llave );
            if( !primero )
                free( pb );
            else
                primero = FALSE;
        }
    free( t->tabla );
    free( t );
}
```

- Las operaciones de salvar y cargar una tabla utilizan el esquema de persistencia definido en las implementaciones anteriores.

```
TablaH cargarTablaH( FILE *fp )
{   char llave[ 12 ];
    TablaH t;
    int valor, cap;
    fscanf( fp, "%d", &cap );
    t=inicTablaH( cap );
    fscanf( fp, "%s %d", llave, &valor );
    while( llave[ 0 ] != '*' )
    {   insTablaH( t, llave, valor );
        fscanf( fp, "%s %d", llave, &valor );
    }
    return t;
}
```

```
void salvarTablaH( TablaH t, FILE *fp )
{   int i, j;
    struct Bloque *pb;
    fprintf( fp, "%d\n", t->M );
    for( j = 0; j < t->M; j++ )
        for( pb = &t->tabla[ j ]; pb != NULL; pb = pb->sig )
            for( i = 0; i < K; i++ )
                if( pb->elem[ i ].llave != NULL )
                    fprintf( fp, "%s %d\n", pb->elem[ i ].llave, pb->elem[ i ].info );
    fprintf( fp, "******* 0\n");
}
```

Ejercicios Propuestos

7.1. Implemente el TAD TablaH utilizando una lista sencillamente encadenada con apuntadores para manejar las colisiones. Utilice el probador interactivo para validar las rutinas desarrolladas.

7.2. Implemente el TAD TablaH utilizando un árbol binario ordenado para manejar las colisiones. Utilice el probador interactivo para validar las rutinas desarrolladas.

7.3. Una variante de la segunda implementación mostrada para tablas de *hashing* es avanzar en pasos mayores que 1, para localizar una posición libre en caso de colisión. Implemente el TAD TablaH con un paso igual a una constante P mayor que 1. Estudie las ventajas de esta implementación. Utilice el probador interactivo para validar las rutinas desarrolladas.

7.4. Una variante de la segunda implementación mostrada para tablas de *hashing* es utilizar doble *hashing* para determinar el paso de avance en la búsqueda de una posición libre. La idea es tener una segunda función paso(llave), que indica para cada llave el número de espacios que debe avanzar. Implemente el TAD TablaH con esta variante. Estudie las ventajas de esta implementación. Utilice el probador interactivo para validar las rutinas desarrolladas.

7.5. Implemente sobre la primera representación planteada para tablas de *hashing* una rutina que imprima ordenadamente las llaves presentes. Calcule la complejidad. Modifique el probador interactivo del TAD para validar esta rutina.

7.6. Implemente sobre la segunda representación planteada para tablas de *hashing* una rutina que imprima ordenadamente las llaves presentes. Calcule la complejidad. Modifique el probador interactivo del TAD para validar esta rutina.

7.7. Implemente sobre la tercera representación planteada para tablas de *hashing* una rutina que imprima ordenadamente las llaves presentes. Calcule la complejidad. Modifique el probador interactivo del TAD para validar esta rutina.

7.8. Modifique el esquema con el que se elimina un elemento en la segunda representación planteada para tablas de *hashing*, de manera que si existe otra llave en conflicto con la llave que se va a eliminar, pase a ocupar el espacio que se libera. De esta forma, se simplifica la rutina que indica si un elemento se encuentra presente en la tabla, porque al encontrar la primera posición vacía se sabe que ya no puede aparecer.

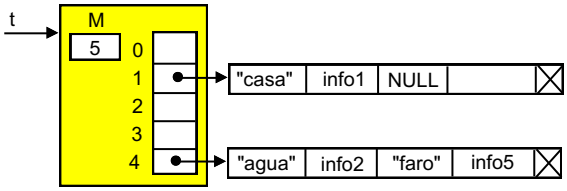
7.9. Modifique el esquema con el que se elimina un elemento en la tercera representación planteada para tablas de *hashing*, de manera que libere los bloques que ya no se utilizan en la zona de desbordamiento. Debe además garantizar que las posiciones ocupadas se encuentran a la izquierda de cada bloque. De esta forma, se simplifica la rutina que indica si un elemento se encuentra presente en la tabla, porque al encontrar la primera posición vacía se sabe que ya no puede aparecer.

7.10. Desarrolle una rutina sobre las 3 representaciones de tablas de *hashing* vistas en la sección anterior, que calcule y retorne el número de colisiones presentes. Esto puede dar una medida de la calidad de la función utilizada.

7.11. Desarrolle una rutina sobre las 3 representaciones de tablas de *hashing* vistas en la sección anterior, que aumente la capacidad de una tabla en un porcentaje dado. Fíjese que debe recalcular las direcciones en el área primaria de cada llave. Modifique el probador interactivo del TAD para validar esta rutina.

7.12. Una variante a la tercera representación de tablas de *hashing* consiste en reservar desde un principio un número fijo de bloques al final del área primaria para manejar allí la zona de desbordamiento. Esta aproximación es muy utilizada cuando la tabla se debe manejar en memoria secundaria. Especifique el esquema de representación, e implemente cada una de las operaciones del TAD. Utilice el probador interactivo para validar las rutinas desarrolladas.

7.13. Una variante a la tercera representación de tablas de *hashing*, que puede ahorrar espacio en memoria, consiste en manejar en el área primaria únicamente apuntadores a bloques, los cuales son pedidos por necesidad. El siguiente ejemplo ilustra la idea.



Implemente el TAD TablaH sobre la representación propuesta. Utilice el probador interactivo para validar las rutinas desarrolladas.

7.14. Diseñe un esquema de persistencia para una tabla de *hashing*, basado en archivos de acceso directo, e implemente las rutinas que cargan y salvan una tabla utilizando la primera representación vista. Utilice el probador interactivo para validar las rutinas desarrolladas.

7.15. Diseñe un esquema de persistencia para una tabla de *hashing*, basado en archivos de acceso directo, e implemente las rutinas que cargan y salvan una tabla utilizando la segunda representación vista. Utilice el probador interactivo para validar las rutinas desarrolladas.

7.16. Diseñe un esquema de persistencia para una tabla de *hashing*, basado en archivos de acceso directo, e implemente las rutinas que cargan y salvan una tabla utilizando la tercera representación vista. Utilice el probador interactivo para validar las rutinas desarrolladas.

7.17. Diseñe unas estructuras de datos para el TAD TablaH, de manera que sea posible reorganizar periódicamente las llaves, con el fin de que aquéllas que se consultan más frecuentemente queden en las primeras posiciones, al aplicar el respectivo mecanismo de solución de conflictos. Implemente sobre dichas estructuras de datos las operaciones del TAD y valide su funcionamiento con el probador interactivo.

7.18. Diseñe unas estructuras de datos para el TAD TablaH, en las cuales el mecanismo de solución de conflictos sea semejante al de distribución en área primaria, pero que utilice un campo adicional, de tipo apuntador, para encadenar los elementos con el mismo valor de la función de *hashing*.

7.5. Funciones de *Hashing*

Aunque para una tabla de *hashing* resulta fundamental la función que la soporta, no es conveniente perder demasiado tiempo en estudios teóricos para escogerla. Es mejor utilizar una función que distribuya

razonablemente las llaves y no tratar de encontrar un óptimo. Es más importante hacer un manejo eficiente de las colisiones, que de todas formas se van a presentar.

En la mayoría de los casos, la función de *hashing* se puede dividir en dos etapas: la primera, proyecta una llave sobre un espacio intermedio de dimensión superior a la que tiene el área primaria, y, la segunda, convierte un valor de dicho espacio en una dirección válida. Para esta última etapa están las funciones de división y de truncamiento.

7.5.1. Funciones de División

La función típica es la función módulo M, que retorna siempre un valor entero entre 0 y M-1. Esta función se encuentra definida como $H(x) = x \% M$, y convierte cualquier valor numérico en una dirección válida del área primaria. En el ejemplo 7.1 se utiliza una función de este tipo. En el siguiente ejemplo aparecen algunas otras funciones de división.

Ejemplo 7.6:

Sobre el valor 38.998.787, se pueden utilizar las siguientes funciones de división, suponiendo que la capacidad de la tabla es 4.096.

- Función módulo M: 38.998.787 % 4.096 = 771
- Dividir el número en dos cadenas del mismo número de dígitos y sumarlas, repitiendo el proceso hasta obtener una dirección válida. Para el ejemplo, el resultado es:

  38.998.787 → 3.899 + 8.787 = 12.686
  12.686 → 126 + 86 = 212

⏎

7.5.2. Funciones de Truncamiento

Una función de truncamiento toma un valor numérico, y, a partir de diferentes alteraciones de dicho valor, consistentes en la supresión de algunos de sus elementos (dígitos o *bits*), lo modifica hasta convertirlo en una dirección válida. En el siguiente ejemplo se presentan algunas funciones de truncamiento.

Ejemplo 7.7:

Sobre el valor 38.998.787.665, se pueden utilizar las siguientes funciones de truncamiento, suponiendo que la capacidad de la tabla es 4.096.

- Eliminar alternativamente el primero y el último dígitos, hasta obtener una dirección válida. Para el valor del ejemplo, el resultado es 878.
- Truncar suficientes dígitos en el medio de la cadena numérica, hasta obtener un valor menor que la capacidad de la tabla. Para el ejemplo, el resultado es 3865.

⏎

7.5.3. Funciones sobre un Espacio Intermedio

Las funciones de proyección sobre un espacio intermedio pertenecen a una gama muy amplia de funciones, cuyo fin es pasar de un espacio alfanumérico a un espacio puramente numérico o simplemente lograr una mejor distribución de las llaves. En la bibliografía aparecen referenciadas gran cantidad de estas funciones,

con reportes de su desempeño para cierto tipo específico de tareas. A continuación se presentan, a manera de ejemplo, algunas de estas funciones.

Ejemplo 7.8:

Sobre el valor 38.998, se pueden utilizar las siguientes funciones de proyección sobre un espacio intermedio, y sobre el resultado obtenido, utilizar una función de división o de truncamiento.

- La llave se multiplica por sí misma. El resultado es 1.520.844.004.
- Se multiplica cada uno de los dígitos de la llave, incluido el valor ASCII de los caracteres no numéricos. El resultado es 15.552.
- Se toma la llave como si se encontrara en otra base (v.g. 11), y se convierte a base 10. El resultado es $(38.998)_{11} = 8 + 9*11 + 9*11^2 + 8*11^3 + 3*11^4 = 55.767$.
- Se suman cada par de dígitos adyacentes módulo 10. El resultado es 188.
- Unir las cifras que representan los caracteres ASCII de la llave, si éstos son alfabéticos. Para la llave "CASA" el resultado de la función es 67.658.465.

⏎

Bibliografía

- [AHO83] Aho, A., Hopcroft, J., Ullman, J., "Data Structures and Algorithms", Cap. 4, Addison-Wesley, 1983.
- [BER94] Bergin, J., "Data Abstraction: The Object-Oriented Approach Using C++", Cap. 12, McGraw-Hill, 1994.
- [ESA89] Esakov, J., Weiss, T., "Data Structures: An Advanced Approack Using C", Cap. 8, Prentice-Hall, 1989.
- [FEL88] Feldman, M., "Data Structures with Modula-2", Cap. 8, Prentice-Hall, 1988.
- [HOR83] Horowitz, E., "Fundamentals of Data Structures", Cap. 9, Computer Science Press, 1983.
- [KNU73] Knuth, D., "The Art of Computer Programming", Vol. 3 - Sorting and Searching, (pp. 506 - 549), Addison-Wesley, 1973.
- [KRU87] Kruse, R., "Data Structures & Program Design", 2da. edición, Cap. 6, Prentice-Hall, 1987.
- [LIP87] Lipschutz, S., "Estructura de Datos", Cap. 9, McGraw-Hill, 1987.
- [MAR86] Martin, J., "Data Types and Data Structures", Cap. 9, Prentice-Hall, 1986.
- [TEN93] Tenenbaum, A., Langsam, Y., "Estructuras de Datos en C", Cap. 7, Prentice Hall, 1993.
- [TRE76] Tremblay, J., Sorenson, P., "An Introduction to Data Structures with Applications", Cap. 6, McGraw-Hill, 1976.
- [WIR86] Wirth, N., "Algorithms & Data Structures", Cap. 5, Prentice-Hall, 1986.

Anexo A:

Tabla ASCII

0	NULL	20		40	(	60	<
1		21		41	)	61	=
2		22		42	*	62	>
3		23		43	+	63	?
4		24		44	,	64	@
5		25		45	-	65	A
6		26		46	.	66	B
7	BELL	27	ESC	47	/	67	C
8	backspace	28		48	0	68	D
9		29		49	1	69	E
10		30		50	2	70	F
11		31		51	3	71	G
12		32	espacio	52	4	72	H
13	return	33	!	53	5	73	I
14		34	"	54	6	74	J
15		35	#	55	7	75	K
16		36	$	56	8	76	L
17		37	%	57	9	77	M
18		38	&	58	:	78	N
19		39	'	59	;	79	O
80	P	92	\	104	h	116	t
81	Q	93	]	105	i	117	u
82	R	94	^	106	j	118	v
83	S	95	_	107	k	119	w
84	T	96	`	108	l	120	x
85	U	97	a	109	m	121	y
86	V	98	b	110	n	122	z
87	W	99	c	111	o	123	{
88	X	100	d	112	p	124	|
89	Y	101	e	113	q	124	}
90	Z	102	f	114	r	126	~
91	[	103	g	115	s	127	DEL

Anexo B:

Contenido y Uso del Disquete de Apoyo

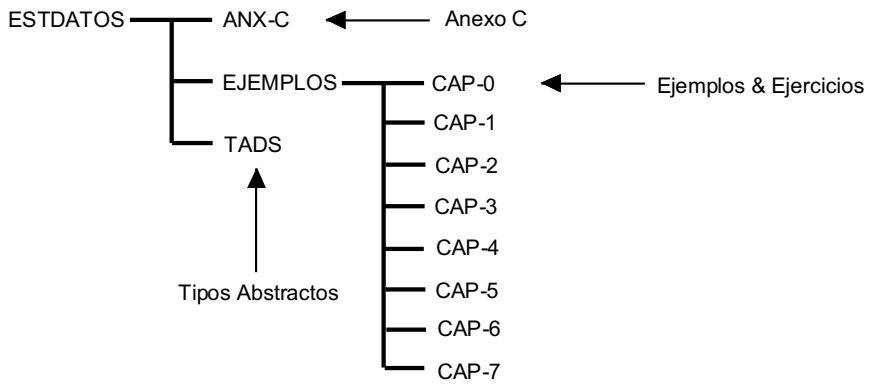
1. Instalación

En el disquete que acompaña el libro se puede encontrar una copia de todos los ejemplos y ejercicios, marcados como implementados, a lo largo de los diferentes capítulos. Dicho disquete viene en formato de *backup* de *Windows*™, y debe utilizarse dicho programa para restaurar su contenido en un disco duro (ocupa 2.5 *MBytes* aproximadamente). Si no conoce bien su funcionamiento, se le recomienda consultar un manual de *Windows*™ 3.1, o de una versión posterior. Luego de restaurar los archivos mencionados anteriormente, es necesario compilar de nuevo cada uno de los ejemplos y ejercicios. Para facilitar este proceso, se incluyen los *projects* de compilación para *Turbo C++* 3.1™, los cuales deben ser ajustados según la instalación que se tenga del compilador. Si no se cuenta con dicho compilador, es necesario crear el *makefile* correspondiente, asegurándose que se incluyen las implementaciones adecuadas de los TAD que se referencian en cada caso.

En el disquete de apoyo también vienen incluidas las implementaciones de los diferentes TAD del libro, lo mismo que una solución completa del caso estudiado en el Anexo C. En estos dos casos se incluyen de una vez los archivos ejecutables (archivos *.EXE*).

2. Estructura de Directorios

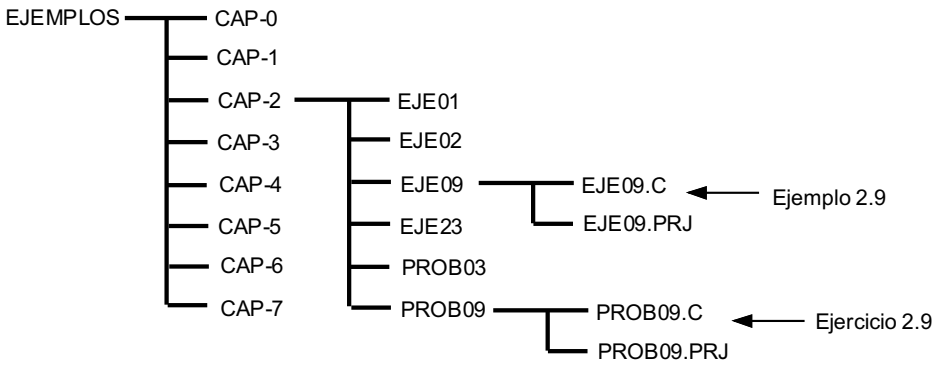
La siguiente es la estructura de directorios en la cual se encuentran distribuidos los archivos mencionados en la sección anterior:



(1) Los archivos del anexo C se encuentran en el directorio ANX-C.

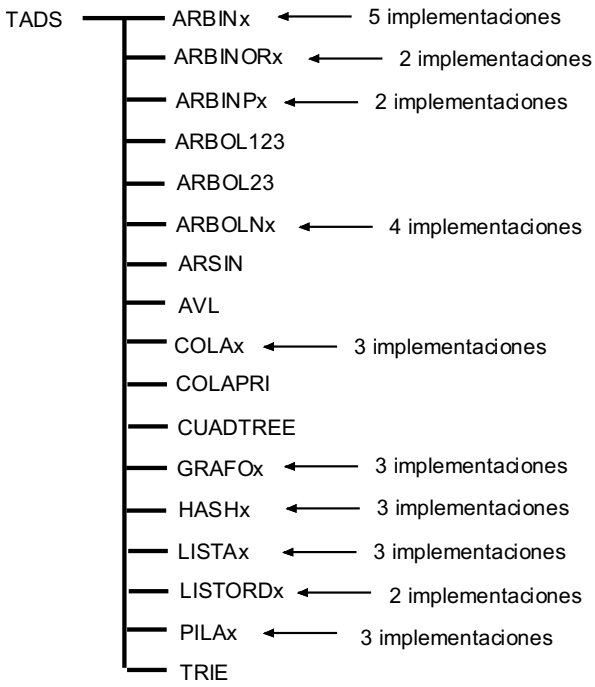
(2) Los ejemplos y ejercicios resueltos se encuentran en el directorio EJEMPLOS. Allí aparece un directorio por cada capítulo, y, en su interior, un subdirectorio distinto por cada ejemplo (archivos EJEXX) o ejercicio (archivos PROBXX).

Siguiendo esta convención, el ejemplo 9 del capítulo 2 se encuentra en el subdirectorio EJE09 del directorio CAP-2, mientras el ejercicio 9 del mismo capítulo, se encuentra en el subdirectorio PROB09 del mismo directorio CAP-2, como se ilustra en la siguiente figura:



(3) Cada una de las implementaciones de los TAD del libro aparece en un subdirectorio distinto del directorio TADS. El nombre de dicho subdirectorio corresponde al nombre del TAD, seguido de un índice, en los casos en los cuales exista más de una implementación.

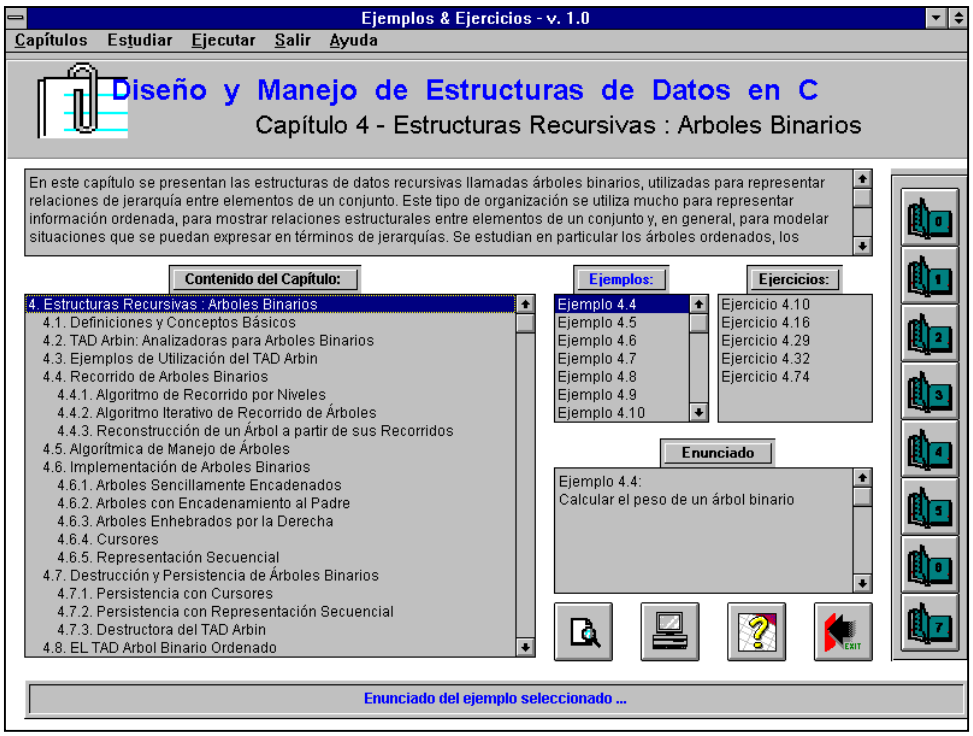
La siguiente figura resume los subdirectorios de los TADS:

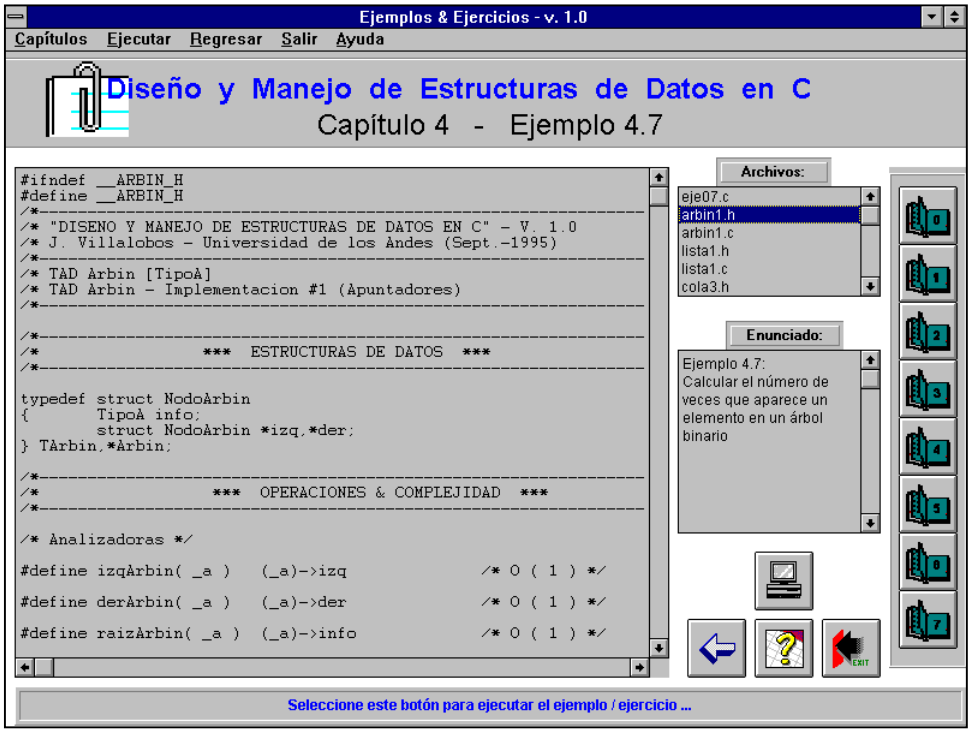


3. Otro Software de Apoyo

Adicional a las implementaciones antes mencionadas, existe un conjunto de herramientas computacionales de apoyo al curso, las cuales denominamos el **Laboratorio de Estructuras de Datos (LED)**. Dicho laboratorio se ejecuta sobre *Windows*™ 3.11 o sobre *Windows*™ 95, y permite al estudiante armar nuevos ejemplos y visualizar gráficamente su comportamiento. Este laboratorio consta de los siguientes módulos:

(1) *Browser* de ejemplos y ejercicios del libro. Permite al estudiante viajar de manera cómoda y transparente por los directorios que contienen los ejemplos, ejercicios e implementaciones de los TAD, etc., dándole facilidades de ejecución, compilación, edición y búsqueda. Incluye la solución de otros ejercicios propuestos del libro y la implementación completa de nuevos TAD.

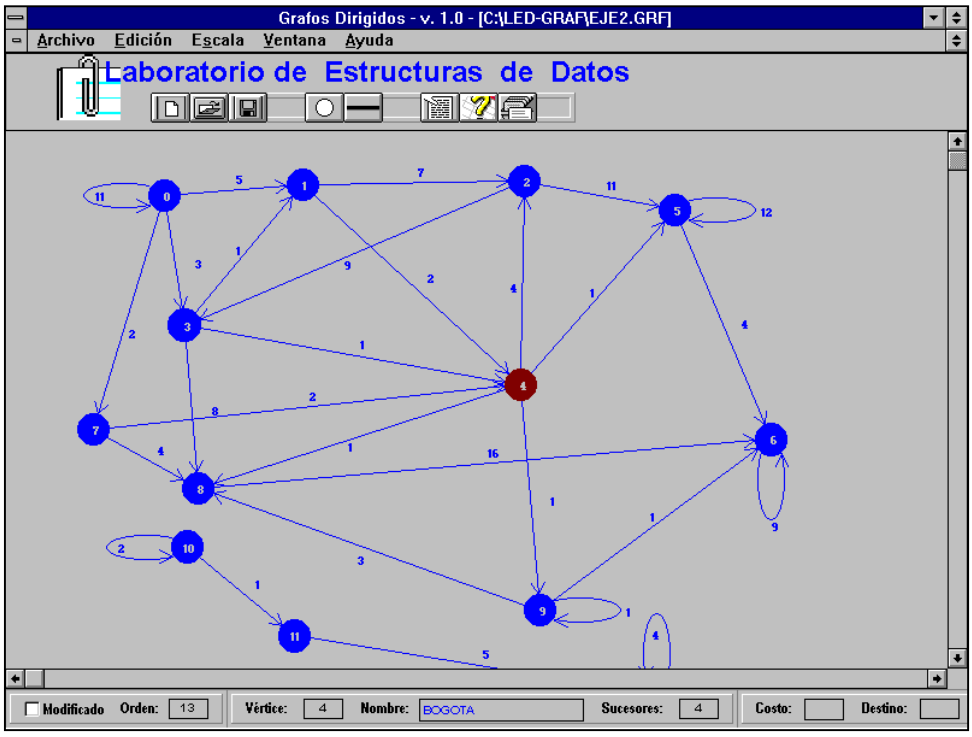

Ejemplos & Ejercicios - v. 1.0
Capítulos Estudiar Ejecutar Salir Ayuda
Diseño y Manejo de Estructuras de Datos en C
Capítulo 4 - Estructuras Recursivas : Arboles Binarios
En este capítulo se presentan las estructuras de datos recursivas llamadas árboles binarios, utilizadas para representar relaciones de jerarquía entre elementos de un conjunto. Este tipo de organización se utiliza mucho para representar información ordenada, para mostrar relaciones estructurales entre elementos de un conjunto y, en general, para modelar situaciones que se puedan expresar en términos de jerarquías. Se estudian en particular los árboles ordenados, los
Contenido del Capítulo:
4. Estructuras Recursivas : Arboles Binarios
4.1. Definiciones y Conceptos Básicos
4.2. TAD Arbin: Analizadoras para Arboles Binarios
4.3. Ejemplos de Utilización del TAD Arbin
4.4. Recorrido de Arboles Binarios
4.4.1. Algoritmo de Recorrido por Niveles
4.4.2. Algoritmo Iterativo de Recorrido de Árboles
4.4.3. Reconstrucción de un Árbol a partir de sus Recorridos
4.5. Algorítmica de Manejo de Árboles
4.6. Implementación de Arboles Binarios
4.6.1. Arboles Sencillamente Encadenados
4.6.2. Arboles con Encadenamiento al Padre
4.6.3. Arboles Enhebrados por la Derecha
4.6.4. Cursores
4.6.5. Representación Secuencial
4.7. Destrucción y Persistencia de Árboles Binarios
4.7.1. Persistencia con Cursores
4.7.2. Persistencia con Representación Secuencial
4.7.3. Destructora del TAD Arbin
4.8. EL TAD Arbol Binario Ordenado
Ejemplos:
Ejemplo 4.4
Ejemplo 4.5
Ejemplo 4.6
Ejemplo 4.7
Ejemplo 4.8
Ejemplo 4.9
Ejemplo 4.10
Ejercicios:
Ejercicio 4.10
Ejercicio 4.16
Ejercicio 4.29
Ejercicio 4.32
Ejercicio 4.74
Enunciado
Ejemplo 4.4:
Calcular el peso de un árbol binario
Enunciado del ejemplo seleccionado ...


Ejemplos & Ejercicios - v. 1.0
Capítulos Ejecutar Regresar Salir Ayuda
Diseño y Manejo de Estructuras de Datos en C
Capítulo 4 - Ejemplo 4.7
#ifndef __ARBIN_H
#define __ARBIN_H
/*--*/
/* "DISENO Y MANEJO DE ESTRUCTURAS DE DATOS EN C" - V. 1.0
/* J. Villalobos - Universidad de los Andes (Sept.-1995)
/*--*/
/* TAD Arbin [TipoA]
/* TAD Arbin - Implementacion #1 (Apuntadores)
/*--*/
/*--*/
/* *** ESTRUCTURAS DE DATOS ***
/*--*/
typedef struct NodoArbin
{ TipoA info;
struct NodoArbin *izq,*der;
} TArbin,*Arbin;
/*--*/
/* *** OPERACIONES & COMPLEJIDAD ***
/*--*/
/* Analizadoras */
#define izqArbin(_a) (_a)->izq /* O (1) */
#define derArbin(_a) (_a)->der /* O (1) */
#define raizArbin(_a) (_a)->info /* O (1) */
Archivos:
eje07.c
arbin1.h
arbin1.c
lista1.h
lista1.c
cola3.h
Enunciado:
Ejemplo 4.7:
Calcular el número de veces que aparece un elemento en un árbol binario
Seleccione este botón para ejecutar el ejemplo / ejercicio ...

(2) Entrenador gráfico de árboles. Permite al estudiante trabajar de manera gráfica e interactiva con árboles binarios ordenados, árboles AVL y árboles 2-3, facilitándole así la comprensión de toda la algorítmica de manipulación de dichas estructuras. De esta forma, por ejemplo, el estudiante puede seguir paso a paso las rotaciones de los árboles AVL, o visualizar las diferentes soluciones a las que se llega, siguiendo las estrategias de supresión de elementos para árboles binarios ordenados, sugeridas en el capítulo 4.

A través de esta herramienta, el estudiante puede ver en acción los algoritmos del libro, con la opción de ejecución paso a paso, facilitando en buena medida la comprensión de los mismos.

(3) Manejador gráfico de grafos. Permite al estudiante diseñar interactivamente los grafos dirigidos sobre los cuales quiere ver funcionando la algorítmica presentada en el libro. Esto permite animar el algoritmo de Dijkstra, la búsqueda de caminos mínimos, los recorridos de grafos, la búsqueda de caminos hamiltonianos, etc.



También se pueden utilizar otras herramientas computacionales de apoyo al aprendizaje de programación en C (**Laboratorio de Programación Básica**), que incluye módulos de interpretación gráfica de manejo de estructuras encadenadas, recursión simple, ordenamiento en memoria principal y manejo de archivos.

Anexo C:

Estudio de un Caso

Este anexo muestra una solución basada en tipos abstractos de datos, para un problema real. Se presenta la estructura global del *software*, la descripción de cada TAD y, en el disquete de apoyo, la implementación completa de la solución, con una explicación guiada de su desarrollo.

C.1. Enunciado del Problema

Se quiere desarrollar un sistema informático para la administración de la bodega de una biblioteca. Allí se encuentran almacenados libros, de los cuales es importante registrar su título, sus autores, su referencia bibliográfica (3 caracteres alfabéticos, seguidos de un guión y 4 dígitos), su posición en la bodega (número de estante, número de anaquel, posición en el anaquel) y su número de páginas. Para efectos del problema, se supone que el título es único, lo mismo que la referencia bibliográfica. Esto implica que hay un único ejemplar de cada libro en la bodega.

En la biblioteca hay N estantes, cada uno de los cuales tiene 5 anaqueles y en cada anaquel hay espacio para 4 libros. Los estantes, los anaqueles y las posiciones se numeran desde 1. Cuando llega un libro nuevo, el sistema lo asigna automáticamente al primer lugar libre del primer estante con espacio.

Las consultas de los usuarios de la bodega se hacen por referencia bibliográfica (el encargado busca así los libros) o por autor (el usuario debe tener a su disposición un fichero de autores, en donde aparezca, para cada autor, la lista de libros que ha escrito).

El sistema debe permitir además las siguientes opciones:

(1) Agregar un libro a la biblioteca

(2) Eliminar un libro de la biblioteca

(3) Presentar por pantalla el estado de la bodega

(4) Presentar por pantalla el índice de autores (ordenado ascendentemente)

C.2. Descripción de los Tipos Abstractos de Datos

En esta parte se muestra una breve descripción de cada uno de los tipos abstractos que intervienen en la solución del problema, se establece un formalismo, y se presenta una lista de las operaciones para su administración. En el disquete de apoyo se puede estudiar a fondo la solución planteada.

C.2.1. TAD Biblioteca

TAD Biblioteca

Biblioteca
- fichero1: FicheroLibro → contenedora de libros
- fichero2: FicheroAutor → contenedora de fichas de autor
- bodega: Bodega → estructura física de la bodega

{ inv: todos los libros del fichero1 tienen asociada una posición válida de la bodega,
no hay dos libros del fichero1 que ocupen la misma posición en la bodega,
únicamente para los autores de los libros del fichero1 hay una entrada en el fichero2,
todas las posiciones de la bodega que no tienen asociado un libro del fichero1 están marcadas como vacías,
no hay dos libros del fichero1 que tengan el mismo título, ni la misma referencia bibliográfica,
en el fichero2 hay una única entrada para cada autor }

Constructoras:

• inicBiblio:		→ Biblioteca

Modificadoras:

• insLibroBiblio:	Biblioteca x Libro	→ Biblioteca
• elimLibroBiblio:	Biblioteca x char *	→ Biblioteca

Analizadoras:

• consultarBiblio:	Biblioteca x char *	→ Libro
• librosAutorBiblio:	Biblioteca x Autor	→ ListRef
• impBodegaBiblio:	Biblioteca	→

Destructora:

• destruirBiblio:	Biblioteca	→

Persistencia:

• cargarBiblio:	FILE *	→ Biblioteca
• salvarBiblio:	Biblioteca x FILE *	→

Depuración:

• impBiblio:	Biblioteca	→

Esquema de persistencia:

Una biblioteca persiste en un archivo de texto, en el cual se salva la información de la bodega (ver esquema de persistencia de la bodega), seguida de la información de la tabla de *hashing*. Se toma la decisión de no hacer persistir el árbol 2-3 y, en lugar de ello, reconstruirlo cada vez que arranca la ejecución del software, a partir de la información de la tabla de libros.

Biblioteca - fichero1: FicheroLibro - fichero2: FicheroAutor - bodega: Bodega	bodega fichero1

C.2.2. TAD FicheroLibro = TAD TablaH[Libro]

Para modelar el fichero de libros, se escoge una tabla se *hashing* con acceso por referencia bibliográfica, implementada con distribución en área primaria. Se toma para esto la especificación del TAD TablaH (Cap. 7) y se parametriza con el TAD Libro.

Dado que la llave de acceso es de la forma AAA-DDDD, donde A es una letra y D un dígito, se decide utilizar como función de *hashing* la siguiente:

h("$A_1A_2A_3$-$D_1D_2D_3D_4$") = ((A_1 * A_2 * A_3) + D_1 + D_2 + D_3 + D_4) módulo M

Esquema de persistencia:

Se utiliza un esquema parecido al planteado en el capítulo 7 para tablas de *hashing*, pero utilizando el fin de archivo (EOF) como marca de final de la tabla.

< $llave_1 \rightarrow elem_1$, $llave_2 \rightarrow elem_2$, ..., $llave_n \rightarrow elem_n$ > [M]	M libro1 libro2 .... libron

Cada libro, por su parte, persiste de acuerdo con el esquema definido para dicho TAD.

C.2.3. TAD FicheroAutor = TAD Arbol23[FichAutor]

Para modelar el fichero de autores, se escoge un árbol 2-3 con acceso por apellido y nombre del autor. Se toma para esto la especificación del TAD Arbol23 (Cap. 5) y se parametriza con el TAD FichAutor, que representa la noción de ficha de un autor.

Se agregan dos operaciones al TAD, las cuales se encargan de imprimirlo y de destruirlo, respectivamente:

Analizadora:		
• impArbol23:	Arbol23	→
Destructora:		
• destruirArbol23:	Arbol23	→

Así mismo, se modifican las operaciones de inserción y supresión, de la siguiente forma:

- Al insertar una nueva ficha de autor, se verifica primero si el mismo autor tiene ya una ficha activa en el árbol, caso en el cual únicamente se agrega la nueva referencia bibliográfica a la ficha ya existente.

- Sólo se elimina una ficha, cuando la referencia bibliográfica que se quiere suprimir es la única que tiene asociada el autor. En otro caso, sólo se elimina la referencia respectiva, y el árbol no sufre ninguna modificación estructural.

C.2.4. TAD Bodega

La estructura y estado de la bodega de la biblioteca se modela con una matriz de enteros de 3 dimensiones (estantes, anaqueles y posiciones), que indica, en cada posición, si ese lugar se encuentra ocupado.

TAD Bodega		
	$[e_1, ..., e_N]$	→ N estantes
	$e_i = [a_{i1}, ..., a_{i5}]$	→ 5 anaqueles por estante
	$a_{ik} = [p_{ik1},..., p_{ik4}]$	→ 4 posiciones por anaquel
{ inv: $p_{ikr} = 1 \vee p_{ikr} = 0$ }		
Constructoras:		
• inicBodega:		→ Bodega
Modificadoras:		
• asignarBodega:	Bodega x Posición	→ Bodega
• liberarBodega:	Bodega x Posición	→ Bodega
Analizadoras:		
• posLibreBodega:	Bodega	→ Posición
Destructora:		
• destruirBodega:	Bodega	→
Persistencia:		
• cargarBodega:	FILE *	→ Bodega
• salvarBodega:	Bodega x FILE *	→

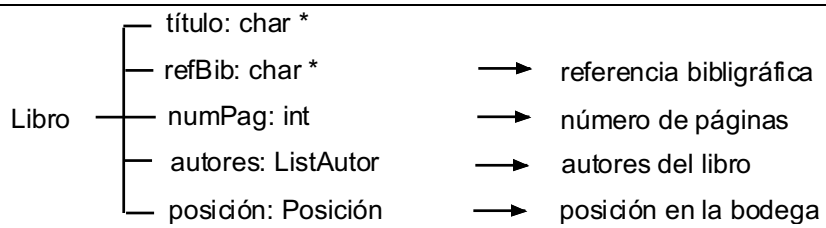
Esquema de persistencia:

El estado de la bodega se almacena de tal manera, que cada línea del archivo representa un estante, con todos sus anaqueles y posiciones consecutivos. Puesto que el número de estantes puede variar en el problema, se coloca dicho valor en la primera línea del archivo:

$[e_1, ..., e_N]$ $e_i = [a_{i1}, ..., a_{i5}]$ $a_{ik} = [p_{ik1},..., p_{ik4}]$	N p111 p112 p113 p114 p121 p122 ... p211 p212 p213 p214 p221 p222 ... . pN11 pN12 pN13 pN14 pN21 pN22 ...

C.2.5. TAD Libro

TAD Libro



{ inv: titulo != NULL, refBib != NULL, numPag > 0 }

Constructoras:

- inicLibro: char * x char * x int → Libro

Modificadoras:

- insAutorLibro: Libro x Autor → Libro
- elimAutorLibro: Libro x Autor → Libro
- asignarPosLibro: Libro x Posición → Libro

Analizadoras:

- refLibro: Libro → char *
- posLibro: Libro → Posición
- autoresLibro: Libro → ListAutor
- esRefLibro: Libro x char * → int
- esAutorLibro: Libro x char * → int
- impLibro: Libro →

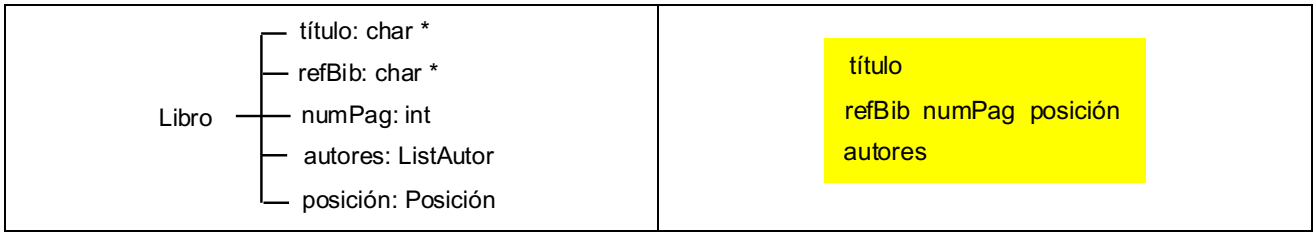
Destructora:

- destruirLibro: Libro →

Persistencia:

- cargarLibro: FILE * → Libro
- salvarLibro: Libro x FILE * →

Esquema de persistencia:



C.2.6. TAD ListAutor = TAD Lista[Autor]

Para modelar los autores de un libro se escoge un objeto abstracto contenedor del TAD Lista. Se toma para esto la especificación de dicho TAD (Cap. 2) y se parametriza con el TAD Autor.

Se utiliza la implementación de listas con doble encadenamiento. Para esto se modifica el nombre del TAD (Lista → ListAutor) y se agrega un sufijo a de cada una de sus operaciones, para evitar conflicto con otras utilizaciones del mismo TAD en otras partes de la solución.

Para la persistencia se utiliza el esquema planteado en el capítulo 2.

C.2.7. TAD FichAutor

Representa una ficha del fichero de autor. En ella se asocia, con un autor, la lista de las referencias bibliográficas de sus libros.

TAD FichAutor		
[autor, < ref_1, ..., ref_N >]		
{ inv: N > 0 }		
Constructoras:		
• inicFichAutor:	Autor x char *	→ FichAutor
Modificadoras:		
• adicRefFichAutor:	FichAutor x char *	→ FichAutor
• elimRefFichAutor:	FichAutor x char *	→ FichAutor
Analizadoras:		
• autorFichAutor:	FichAutor	→ Autor
• refFichAutor:	FichAutor	→ ListRef
• impFichAutor:	FichAutor	→
Destructora:		
• destruirFichAutor:	FichAutor	→

C.2.8. TAD Posición

Modela la noción de localización de un libro en la bodega. Una posición está compuesta por un número de estante, un número de anaquel y un consecutivo en dicho anaquel.

TAD Posición		
[nEstante, nAnaquel, nPosición]		
{ inv: 1 ≤ nEstante, 1 ≤ nAnaquel ≤ 5, 1 ≤ nPosición ≤ 4 }		
Constructoras:		
• inicPos:	int x int x int	→ Posición
Analizadoras:		
• estantePos:	Posición	→ int
• anaquelPos:	Posición	→ int
• posicionPos:	Posición	→ int
• impPos:	Posición	→
Destructora:		
• destruirPos:	Posición	→
Persistencia:		

• cargarPos:	FILE *	→ Posición
• salvarPos:	Posición x FILE *	→

Esquema de persistencia:

[nEstante, nAnaquel, nPosición]	nEstante nAnaquel nPosición

C.2.9. TAD Autor

Representa al autor de un libro. La única información relevante para el modelaje de este objeto abstracto es su apellido y su nombre.

TAD Autor		
[apellido, nombre]		
Constructoras:		
• inicAutor:	char * x char *	→ Autor
Analizadoras:		
• apellidoAutor:	Autor	→ char *
• nombreAutor:	Autor	→ char *
• esAutor:	Autor x Autor	→ int
• menorAutor:	Autor x Autor	→ int
• impAutor:	Autor	→
Destructora:		
• destruirAutor:	Autor	→
Persistencia:		
• cargarAutor:	FILE *	→ Autor
• salvarAutor:	Autorx FILE *	→

Esquema de persistencia:

[apellido, nombre]	apellido nombre

C.2.10. TAD ListRef = TAD Lista[char *]

Para modelar la lista de referencias bibliográficas asociadas con un autor, se escoge un elemento del TAD Lista, parametrizado con el tipo char *. Se sigue un proceso semejante al utilizado para las listas de autores.

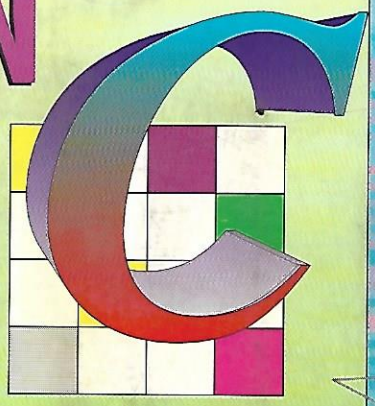
C.2.11. Interfaz del Sistema

Se utiliza una interfaz alfanumérica sencilla, que hace llamadas a las operaciones de los TAD antes planteados, para obedecer los comandos del usuario.

Indice

V

Diseño y Manejo de Estructuras de Datos en C



Este libro está diseñado para servir como texto guía en un curso en diseño y manejo de estructura de datos en C.

El texto utiliza metodologías de tipos abstractos de datos y soporta todo el proceso de diseño en sólidas bases teóricas, brindando al lector herramientas para la evaluación de soluciones, como la complejidad de algoritmos, de manera que el lector cuente con criterios concretos de decisión. El libro no se queda en consideraciones teóricas sino que muestra la dimensión práctica de las metodologías de diseño propuestas y la manera de aplicarlas para mejorar la calidad del software obtenido.

El texto viene apoyado por un disquete, que incluye la implementación y los programas de prueba de todos los algoritmos que son presentados en la parte teórica, lo mismo que la solución de algunos de los ejercicios propuestos; esto permite ver el funcionamiento de cada una de las rutinas planteadas, lo mismo que reutilizar el software para el desarrollo de sus propios proyectos.

ISBN 958-600-505-4

9 789586 005050

Mc Graw Hill