



CENTRO DE ESTUDIOS FINANCIEROS

VIRIATO, 52	28010 MADRID	914 44 49 20
PONZANO, 15	28010 MADRID	914 44 49 20
G. DE GRÀCIA, 171	08012 BARCELONA	934 15 09 88
ALBORAYA, 23	46010 VALENCIA	963 61 41 99

www.cef.es

info@cef.es

Índice Tema 9

1. Estructuras de datos.
 - 1.1. Tablas.
 - 1.2. Listas.
 - 1.2.1. Pilas.
 - 1.2.2. Colas.
 - 1.2.3. Listas doblemente enlazadas.
 - 1.3. Árboles.
 - 1.3.1. Árboles binarios.
 - 1.3.2. Árboles binarios de búsqueda (ABB).
 - 1.3.3. Árboles multicamino. B, B+, B*.
 - 1.4. Grafos.
2. Algoritmos: recursión, ordenación, búsqueda.
 - 2.1. Eficiencia de un algoritmo.
 - 2.2. Recursividad.
 - 2.3. Búsqueda.
 - 2.3.1. Búsqueda secuencial.
 - 2.3.2. Búsqueda binaria.
 - 2.3.3. Búsqueda por interpolación.
 - 2.3.4. Árboles binarios de búsqueda.
 - 2.4. Clasificación.
 - 2.4.1. Algoritmos simples de clasificación.
 - 2.4.2. Algoritmos rápidos de clasificación.

3. Organización de ficheros.

3.1. Organización secuencial.

3.2. Organización directa.

3.3. Variantes de la organización secuencial.

3.3.1. Organización secuencial indexada.

3.3.2. Organización secuencial encadenada.

3.3.3. Organización secuencial indexada encadenada.



CENTRO DE ESTUDIOS FINANCIEROS

VIRIATO, 52	28010 MADRID	914 44 49 20
PONZANO, 15	28010 MADRID	914 44 49 20
G. DE GRÀCIA, 171	08012 BARCELONA	934 15 09 88
ALBORAYA, 23	46010 VALENCIA	963 61 41 99

www.cef.es

info@cef.es

TEMA 9

Estructuras de datos: tablas, listas y árboles. Algoritmos: Ordenación, Búsqueda, Recursión, Grafos. Organizaciones de ficheros.

1. ESTRUCTURAS DE DATOS.

Hay que tener en cuenta que para estudiar la informática es fundamental el conocimiento de las estructuras de datos, ya que éstas se consideran la base de todo el proceso de información.

Lo que realmente se pretende es aclarar del modo más directo y sencillo todos los conceptos relacionados con las estructuras de datos, empezando desde lo más elemental, como puede ser la forma de organizar letras y números y finalizando con las estructuras que los aglutinan, listas, registros, ficheros, etc.

Por otra parte veremos de qué manera se almacenan los datos dentro y fuera del ordenador, y es aquí donde veremos claramente lo qué son y para qué sirven las estructuras de datos.

Los datos representan la información objeto del tratamiento automático de la información, así como el resultado del proceso, se almacenan codificados de acuerdo a unas reglas (estructuras de datos) en los soportes de la información y en la memoria principal, son trasladados a la memoria principal y de ésta a los soportes agrupados en estructuras a través de órdenes específicas en los programas.

Los datos se pueden clasificar en dos tipos:

- Dato elemental. Aquellos que pueden ser almacenados en cualquier soporte de información sin necesidad de ninguna estructura especial para su almacenamiento.

Los datos elementales se pueden clasificar en tres grupos:

1. Numéricos. Lo componen los guarismos (0, 1, 2, 3,... 9), se utilizan para representación de información de tipo numérico.

2. Alfabéticos: compuestos por los caracteres del alfabeto (a, b, c,... x, y, z), se utilizan para representar información no numérica.
 3. Especiales: utilizados con fines sintácticos en la mayoría de los casos o con fines aritméticos en otros. Formados por todos los caracteres que no constituyen información por sí solos (¿,?,=,),(/,&,% , etc.).
- Dato estructurado. Aquellos datos que necesitan estar formados o constituidos con una determinada forma, es decir, la información se almacena siguiendo unas reglas.

Los datos estructurados los podemos clasificar a su vez en dos tipos:

1. Simples. Mínima estructura de datos para el almacenamiento de la información, reciben el nombre de campos; la información se almacena a través de los datos elementales; dentro de estas estructuras destacamos unos campos que no contienen información sino donde se encuentra ésta y que reciben el nombre de puntero (campos que contienen la dirección de memoria de un campo).

Las reglas de almacenamiento de estas estructuras viene determinada por la longitud y el tipo de dato a almacenar.

Ejemplo de campos.

EDAD numérico de 2 caracteres.

NOMBRE alfabético de 15 caracteres.

DIRECCIÓN alfanumérico (letras + números + caracteres especiales) de 10 caracteres.

2. Complejas: estas estructuras se definen a partir de las estructuras simples especificando nuevas reglas de almacenamiento. Son estructuras complejas, listas, grafos, árboles, arrays, registros, ficheros.

1.1. TABLAS.

Colección de elementos homogéneos (de igual longitud y tipo) entre los que existe una relación lineal determinada por la posición del elemento dentro de la estructura, también se las conoce con el nombre de Array.

Las tablas son estructuras de datos complejas que se caracterizan porque sus unidades homogéneas de información se ubican en posiciones contiguas de memoria.

Una tabla viene definida por:

- La definición de sus elementos. Longitud y tipo de dato a almacenar en la estructura.
- Índice (o varios), variable numérica entera positiva que permite el acceso a los distintos elementos de la estructura según el valor de éste y por tanto según la posición en la que se encuentra el elemento dentro de la estructura.

EJEMPLO:

Nombres (5) alfanumérica de 15 caracteres.

I numérico entero.

I	NOMBRES(I)
0	César
1	Pablo
2	Lucía
3	Sandra
4	Tania

Cuando a los distintos elementos de la tabla se accede a través de dos índices, se dice que la tabla es bidimensional y también se la conoce con el nombre de Matriz; en esta circunstancia los elementos quedan estructurados en filas y columnas para su representación gráfica.

I \ J	0	1	2
0	6	7	77
1	14	45	85
2	19	45	85
3	44	1	14
4	1	2	3

Para acceder a cada elemento lo haremos con dos índices I para las filas y J para las columnas de tal forma que ventas (1, 2) vale 85.

Por extensión en la utilización de índices podemos tener tablas n-dimensionales, si bien los lenguajes de programación tienen un rango máximo de dimensiones para su implementación que no suele pasar de dos.

Otro aspecto importante a tener en cuenta será cuando se establece el número de elementos que forman la estructura, que siempre habrá que fijarlo, o bien en tiempo de compilación (estática) o bien en tiempo de ejecución, entonces diremos que la tabla es dinámica.

No todos los lenguajes de programación contemplan las tablas dinámicas; en el lenguaje C gracias a las funciones de asignación dinámica de memoria y a la flexibilidad de los punteros es posible.

En C++ se dispone de clases para crear objetos con estas características al igual que ocurre en Java con la clase Vector (para tablas estáticas) y la clase Array para tablas dinámicas.

A continuación se pone un ejemplo de cómo crear una tabla bidimensional dinámica en el lenguaje de programación C.

```
#include<stdio.h>
#include<stdlib.h>
main(){
int i,j,filas,columnas;
float **matriz;
printf("Introduzca número de filas: ");
scanf("%i",&filas);
printf("Introduzca número de columnas: ");
scanf("%i",&columnas);
//ASIGNACIÓN DINÁMICA
matriz=(float **)malloc(filas*sizeof(float *));
if (matriz==NULL)
exit(0);
for (i=0;i<filas;i++) {
matriz[i]=(float *)malloc(columnas*sizeof(float));
if (matriz[i] ==NULL)
exit(0);
}
}
```

1.2. LISTAS.

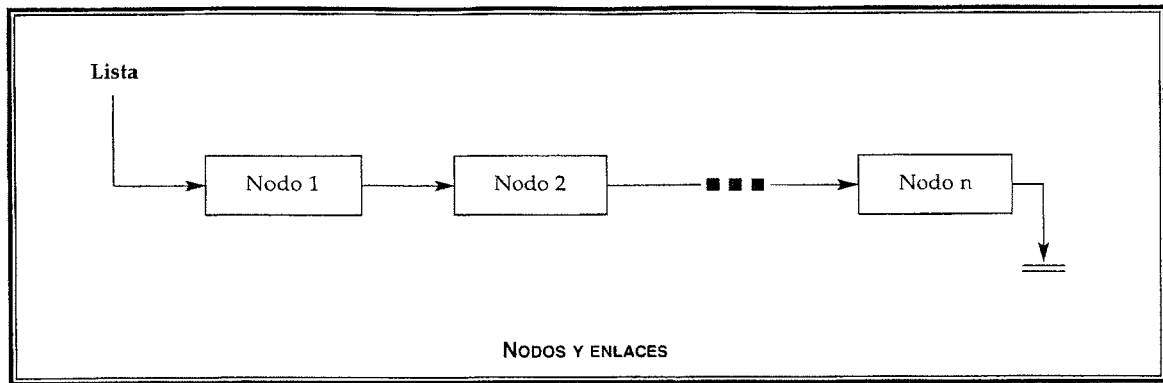
Es un conjunto lógico de elementos homogéneos (nodos) entre los que existe una relación lineal, considerados a cada uno de éstos como unidad básica de información dentro de la estructura. Cada elemento de la estructura puede estar formado por uno o varios campos, subcampos, conjuntos de subcampos. Las listas, a diferencia de los registros, son estructuras de datos que se utilizan en la mayoría de los casos para el almacenamiento de información dentro de la memoria interna del ordenador.

Descripción lógica.

Lista = colección de elementos homogéneos entre los que existe una relación lineal:

- Cada elemento de la lista, a excepción del primero, tiene un único predecesor.
- Cada elemento de la lista, a excepción del último, tiene un único sucesor.

Nodos y enlaces:

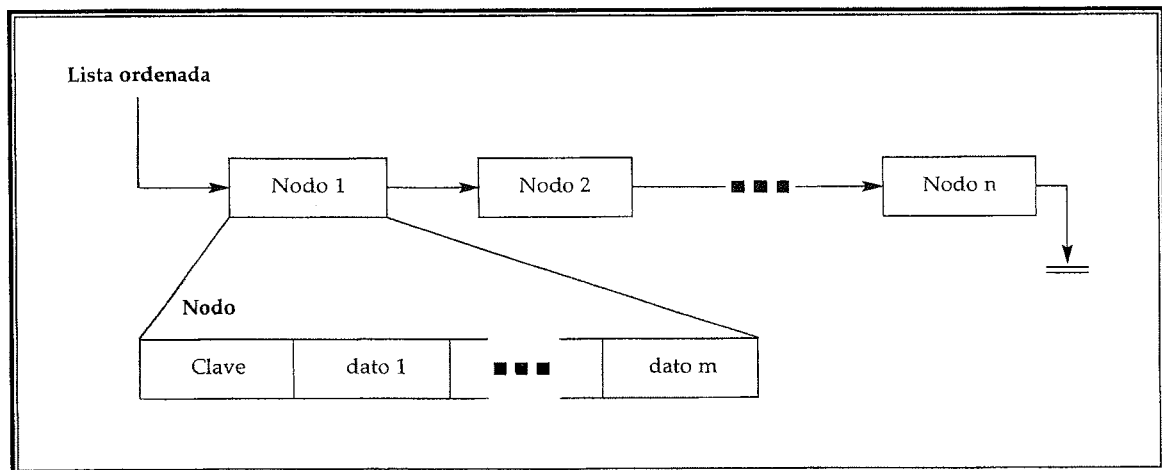


Definición

$\langle \text{lista} \rangle ::= \langle \text{comienzo} \rangle + \{ \langle \text{nodo} \rangle \}$
 $\langle \text{comienzo} \rangle ::= \langle \text{enlace} \rangle$
 $\langle \text{enlace} \rangle ::= (\langle \text{ReferenciaNodo} \rangle \mid \text{NULL})$
 $\langle \text{nodo} \rangle ::= \langle \text{informacion} \rangle + \langle \text{enlace} \rangle$
 $\langle \text{informacion} \rangle ::= \{ \langle \text{dato} \rangle \}$

El orden de nodos afecta a la función de acceso:

- Según orden de inserción.
- Según clave.



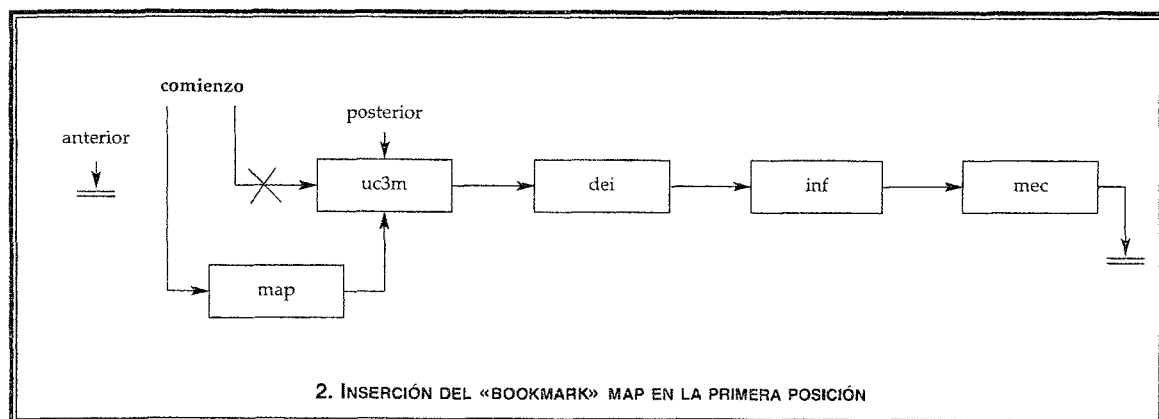
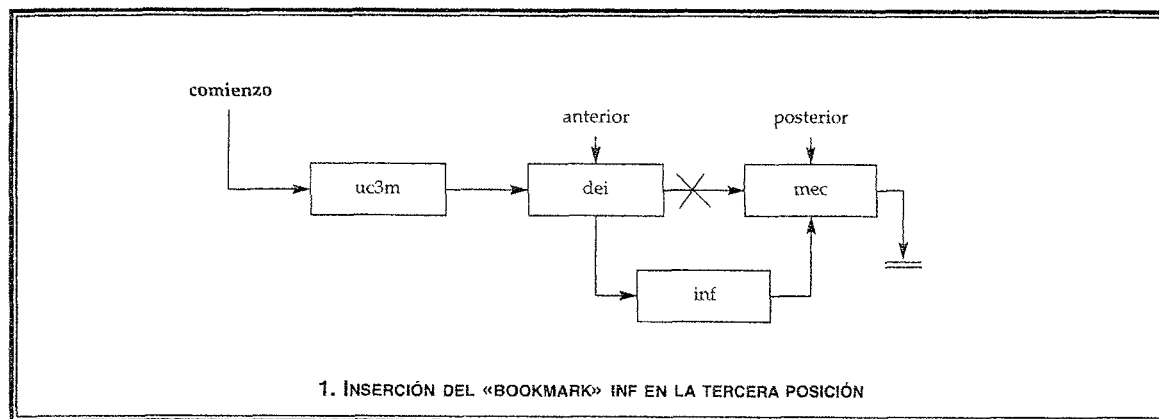
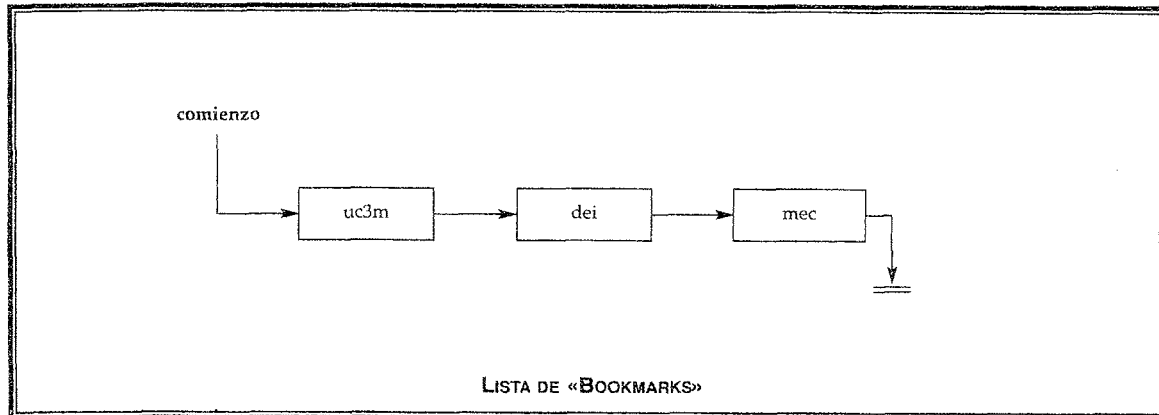
EJEMPLO:

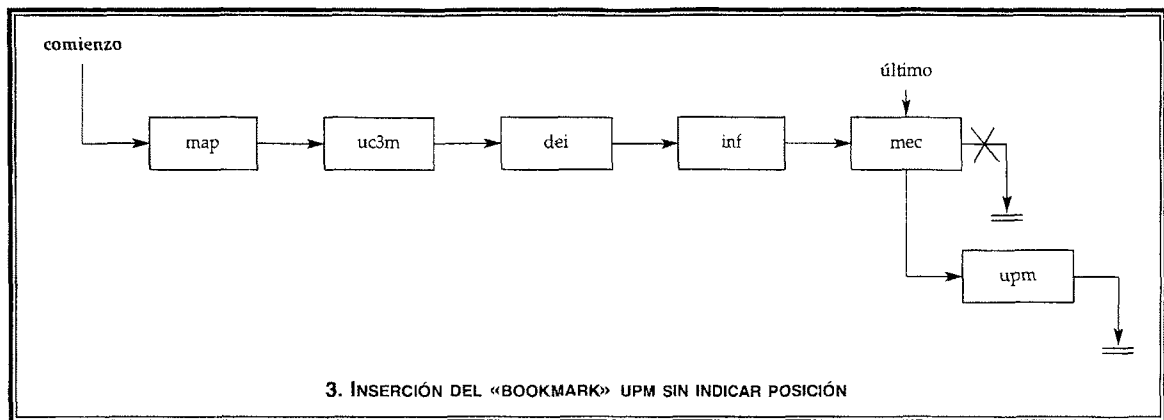
Lista de calificaciones $::= \langle \text{Alumno} \rangle + \{ \langle \text{Alumno} \rangle \}$

$\langle \text{Alumno} \rangle ::= \langle \text{DNI} \rangle + \langle \text{NIA} \rangle + \langle \text{Apellido1} \rangle + \langle \text{Apellido2} \rangle + \langle \text{Nombre} \rangle + \langle \text{Calificación} \rangle$

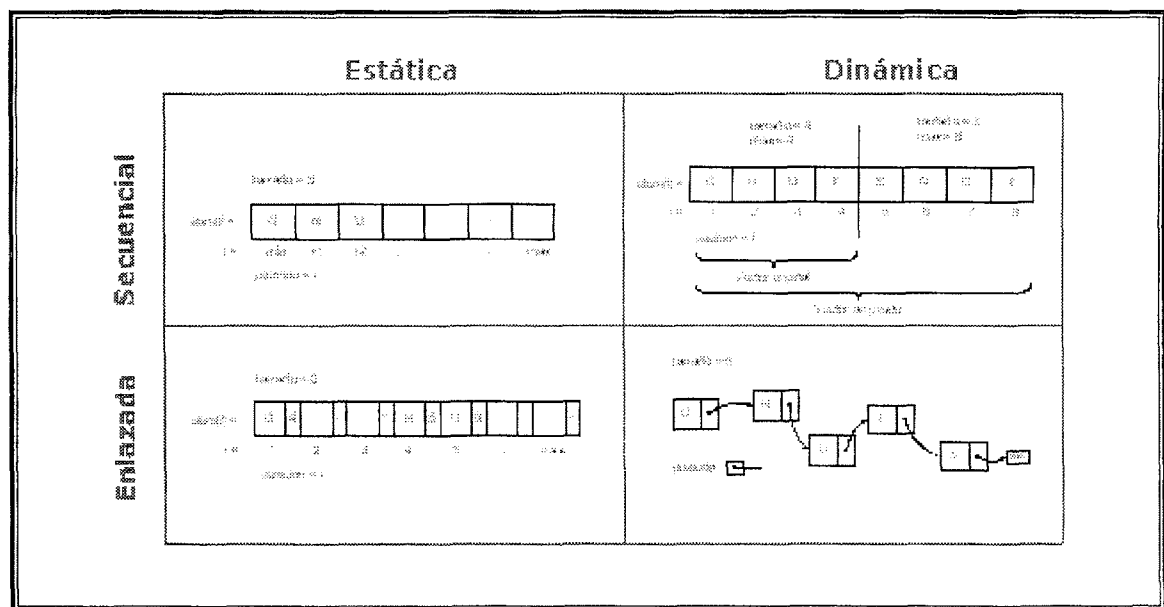
Operaciones sobre listas.

Entre las distintas operaciones a realizar sobre las listas, destacamos la operación de inserción por su mayor complejidad técnica, en la que se requerirá al igual que en las bajas tres casuísticas bien diferenciadas.





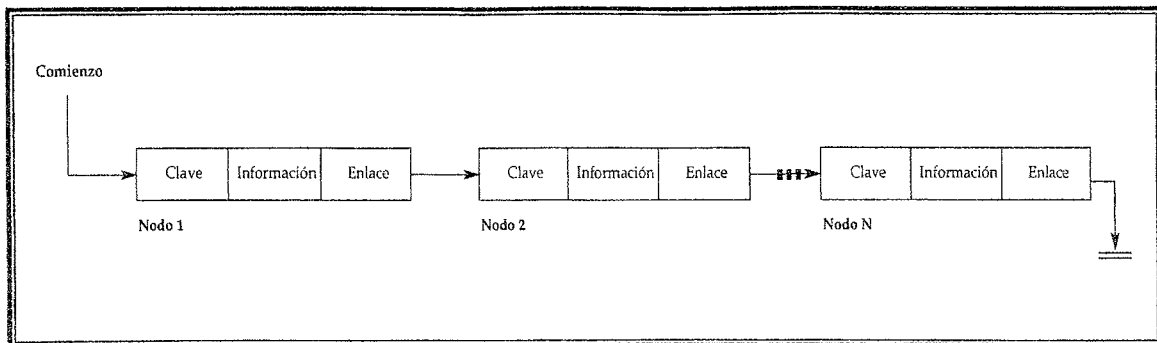
Posibles implementaciones de una lista:



Listas ordenadas:

La posición de cada nodo viene determinada por el valor de uno o más campos obligatorios de información del nodo denominados clave.

No se permite tener dos nodos con la misma clave.



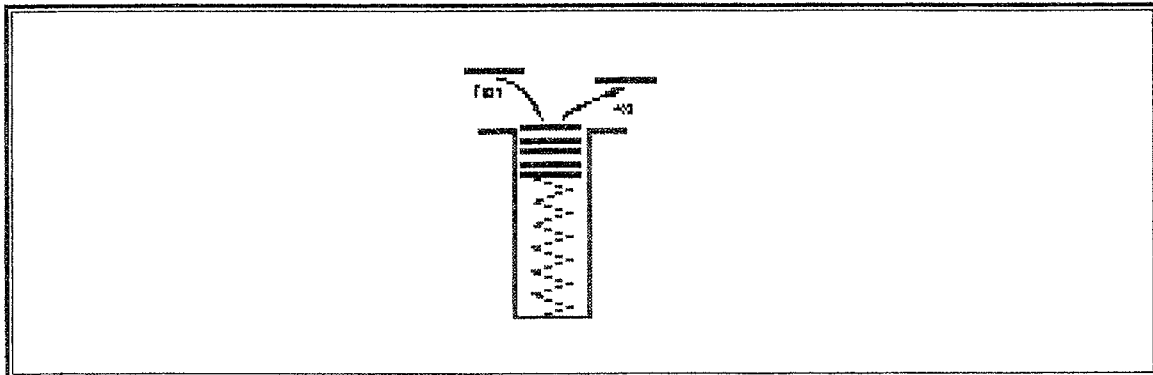
Definición:

```

<listaOrdenada> ::= <comienzo> + {<nodo>}
<comienzo> ::= <enlace>
<enlace> ::= (<<ReferenciaNodo>> | NULL)
<nodo> ::= <clave> + <informacion> + <enlace>
<clave> ::= <<dato>>{<<dato>>}
<informacion> ::= {<<dato>>}
  
```

1.2.1. Pilas.

Pila: colección de elementos homogéneos en la que sólo se pueden añadir y eliminar elementos por el principio de la misma (cabecera), filosofía LIFO.



Definición:

```

<pila> ::= <cabecera> + {<nodo>}
<cabecera> ::= <enlace>
<enlace> ::= (<<ReferenciaNodo>> | NULL)
<nodo> ::= <informacion> + <enlace>
<informacion> ::= <<dato>>{<<dato>>}
  
```

TAD Pila: ejemplos de aplicación:

- Vuelta atrás en un navegador web.
- Comando «undo» en un editor.
- Invocaciones a métodos.

```
main(){
    int i = 5;
    fno(i);
}
fno(intj) {
    int k;
    k = j+1;
    bar(k);
}
bar(int m){
    ...
}
```

bar
PC = 1
m = 6

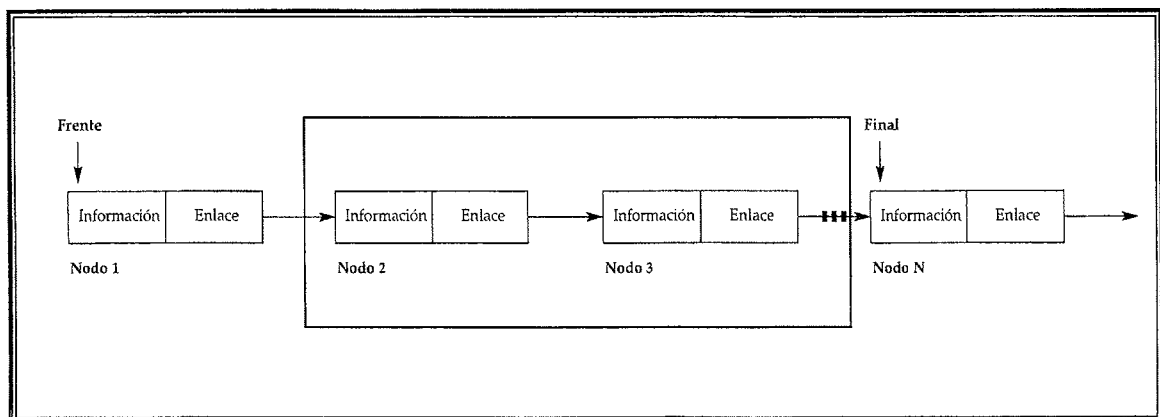
fno
PC = 3
j = 5
k = 6

main
PC = 2
i = 5

- Cuando se invoca un método, se inserta en la pila una nueva entrada.
- Por cada método se guardan:
Variables locales.
Valor devuelto.
Contador de programa.
- Cuando el método finaliza se saca ese elemento de la pila y se devuelve el control al método que esté en la cabecera.

1.2.2. Colas.

Cola: colección ordenada de elementos homogéneos en la que sólo se pueden añadir elementos por el final y se eliminan por el principio (frente), filosofía FIFO.



Definición:

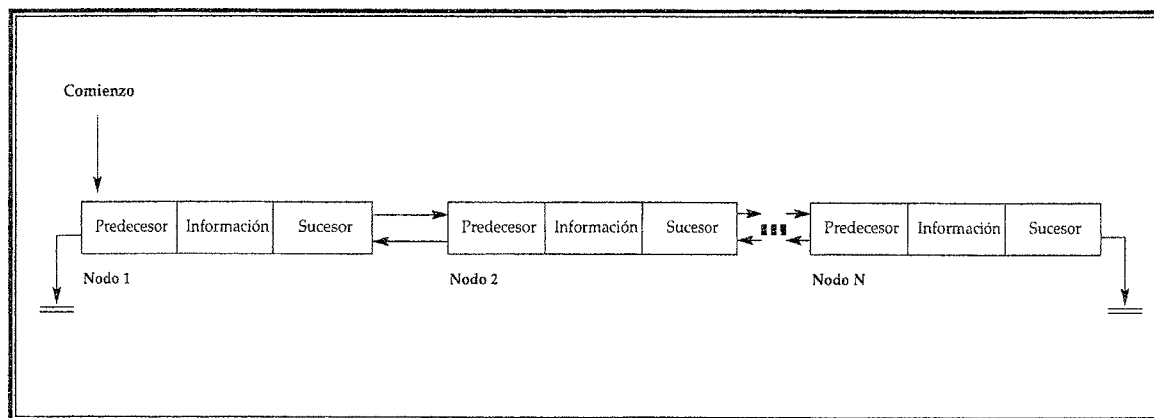
```
<cola> ::= <frente> + <final> + {<nodo>}  
<frente> ::= <enlace>  
<enlace> ::= (<<ReferenciaNodo>> | NULL)  
<final> ::= <enlace>  
<nodo> ::= <informacion> + <enlace>  
<informacion> ::= <<dato>>{<<dato>>}
```

EJEMPLOS DE APLICACIÓN:

- Listas de espera.
- Acceso a recursos compartidos dedicados (v. g., impresoras).
- Multiprogramación de la CPU.

1.2.3. Listas doblemente enlazadas.

- Relación lineal en ambos sentidos.
- Enlace a predecesor y antecesor en cada nodo.
- Recorrido puede ser en ambos sentidos.
- Pueden ser simples u ordenadas.



Definición:

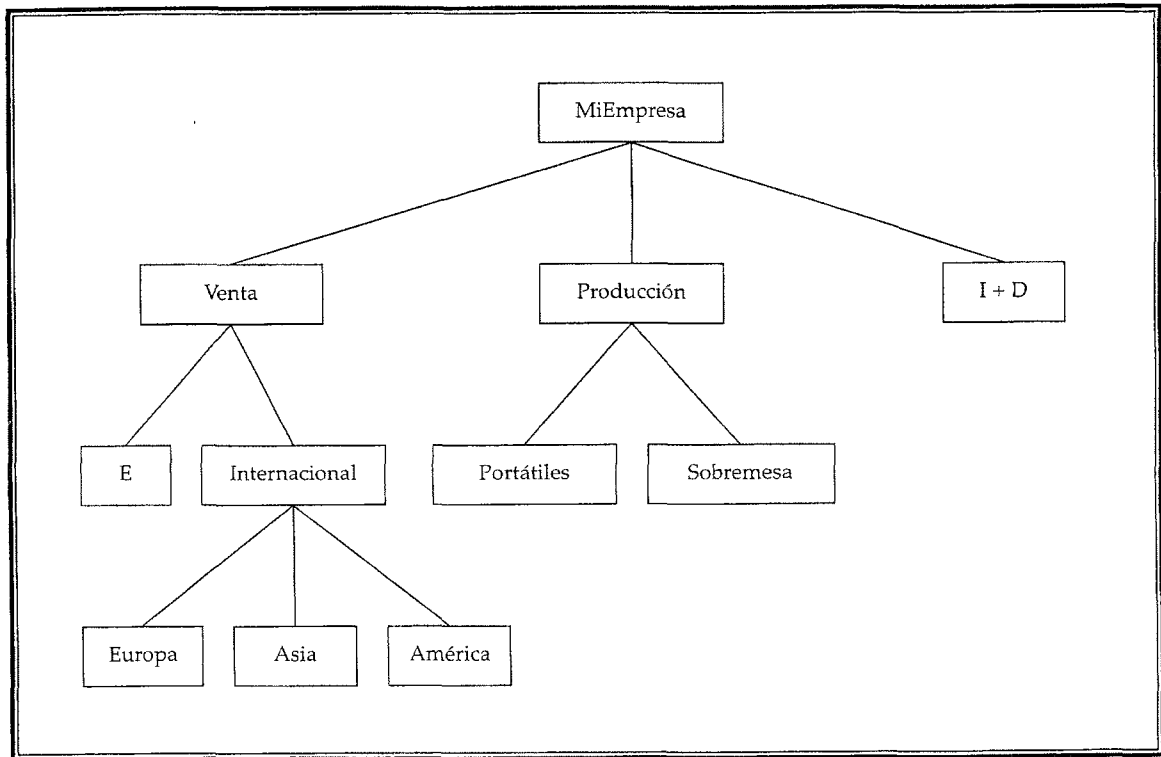
```
<lda> ::= <comienzo> + {<nodo>}  
<comienzo> ::= <enlace>  
<enlace> ::= (<<ReferenciaNodo>> | NULL)  
<nodo> ::= <informacion> + <predecesor> + <sucesor>
```

<predecesor> ::= <enlace>
<sucesor> ::= <enlace>
<informacion> ::= <<dato>>{<<dato>>}

1.3. ÁRBOLES.

Concepto de árbol.

Estructura jerárquica no lineal, Relaciones padre-hijo entre nodos:



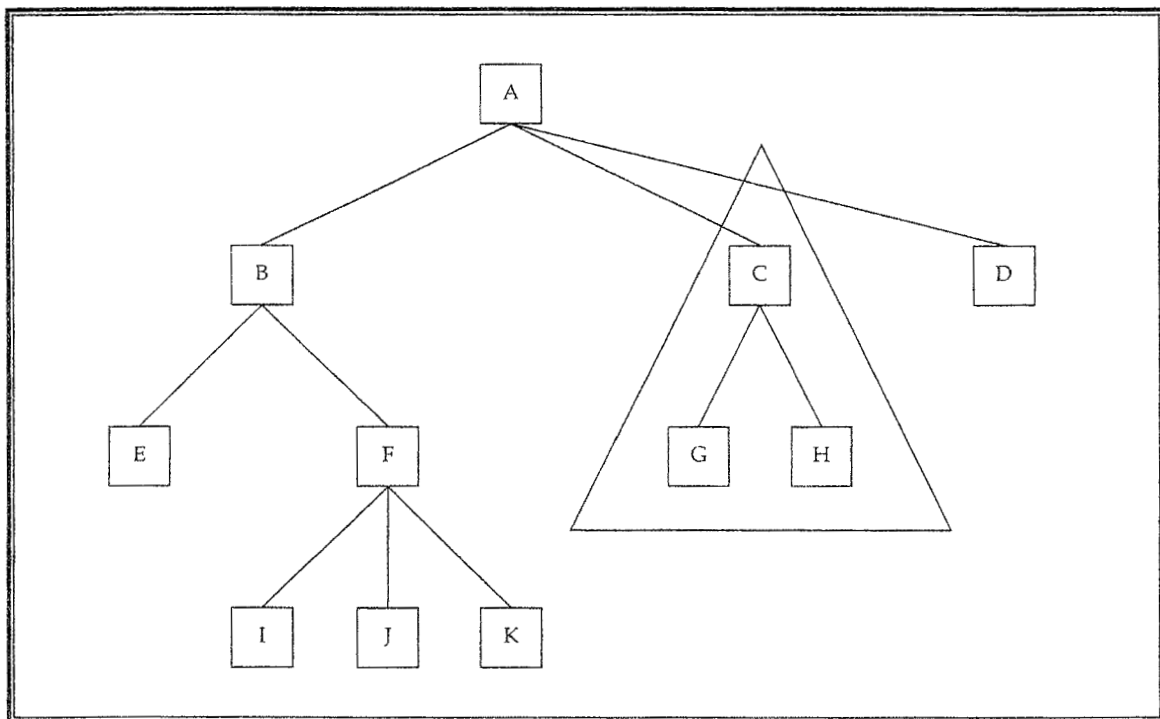
Un árbol se caracteriza por estar formado por una serie de nodos conectados por una serie de aristas que verifican que:

- Hay un único nodo raíz.
- Cada nodo, excepto la raíz, tiene un único padre.
- Hay un único camino (desde la raíz hasta cada nodo).

Terminología básica:

- Raíz: único nodo sin padre.
- Nodo interno: tiene al menos un hijo.

- Nodo hoja (externo): no tiene hijos.
- Descendiente directo: hijo.
- Descendientes: hijo, nieto...
- Subárbol: árbol formado por un nodo y sus descendientes.
- Grado de un nodo: número de descendientes directos.
- Grado del árbol: mayor grado de sus nodos.
- Árbol binario: árbol de grado 2. Cada nodo tiene como mucho dos descendientes directos.
- Árbol multicamino: árbol de grado mayor que 2. Cada nodo puede tener n descendientes directos.
- Lista = árbol degenerado de grado 1.



- Profundidad de un nodo: número de predecesores:

Profundidad (A) = 0

Profundidad (H) = 2

- Altura del árbol: profundidad máxima de cualquier nodo

Altura = 3

- Camino: existe un camino del nodo X al nodo Y, si existe una sucesión de nodos que permitan llegar desde X a Y. camino (A, K) = {A, B, F, K} camino (C, K) = {}

1.3.1. Árboles binarios.

Es un árbol de grado 2. Cada nodo tiene de 0 a 2 descendientes directos: el hijo izquierdo y el derecho.

Descripción Lógica:

```
<arbol> ::= nulo | <nodo>
<nodo> ::= <info> <izq> <der>
<izq> ::= <arbol>
<der> ::= <arbol>
```

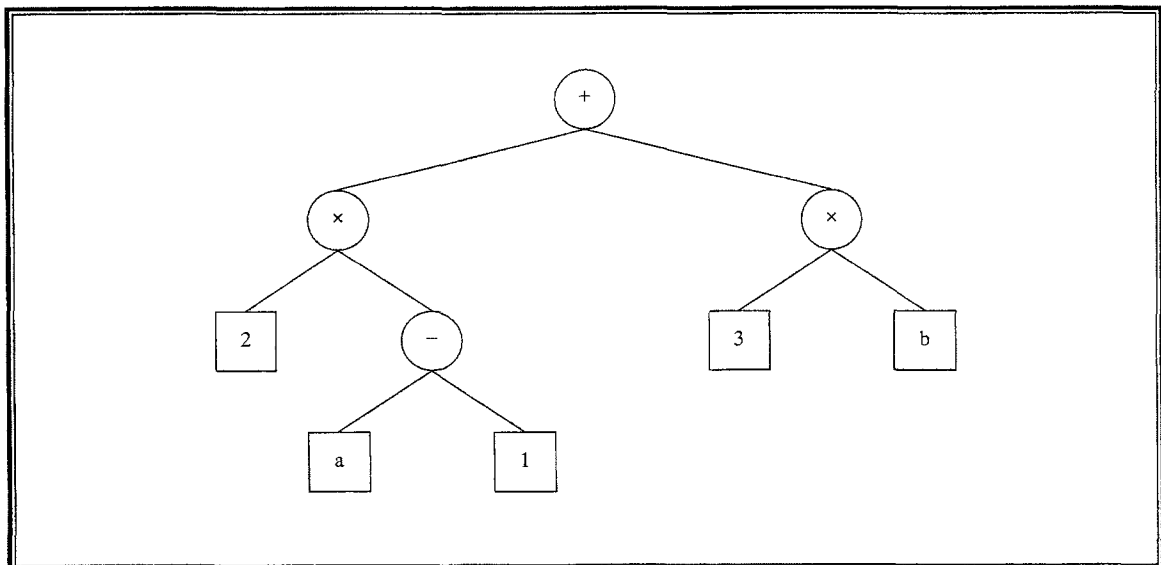
Aplicación: expresiones aritméticas, árboles de decisión, búsqueda (ABB).

En algunos casos se exige que el árbol sea completo, es decir, todo nodo interno tiene dos descendientes.

EJEMPLO: expresiones aritméticas: $2(a-1) + 3b$

Nodo interno: operadores.

Nodos hoja: operandos.

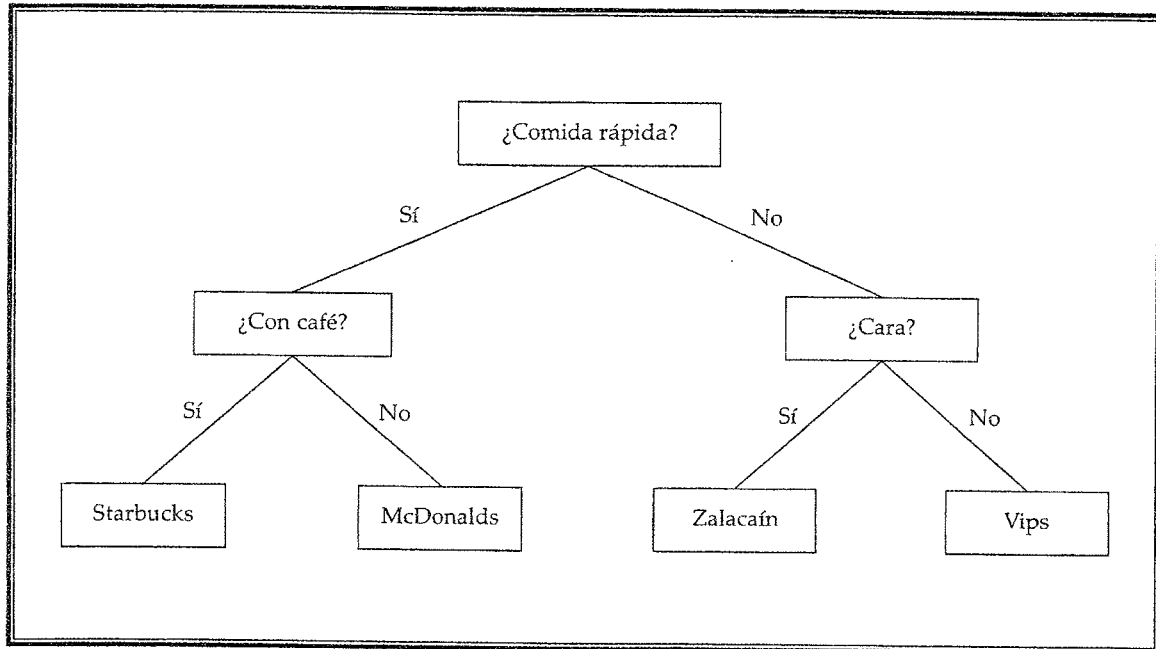


Ejemplo de aplicación: árboles de decisión.

Nodo interno: preguntas con respuesta sí/no.

Nodos hoja: decisiones.

¿Dónde cenamos?



Árboles binarios:

Notación:

n: número de nodos.

e: número de nodos hoja.

i: número de nodos internos.

h: altura del árbol.

Propiedades:

$$e \leq 2^h$$

$$h \leq i \leq 2^h - 1$$

$$\log_2(n+1) - 1 \leq h \leq (n-1)/2$$

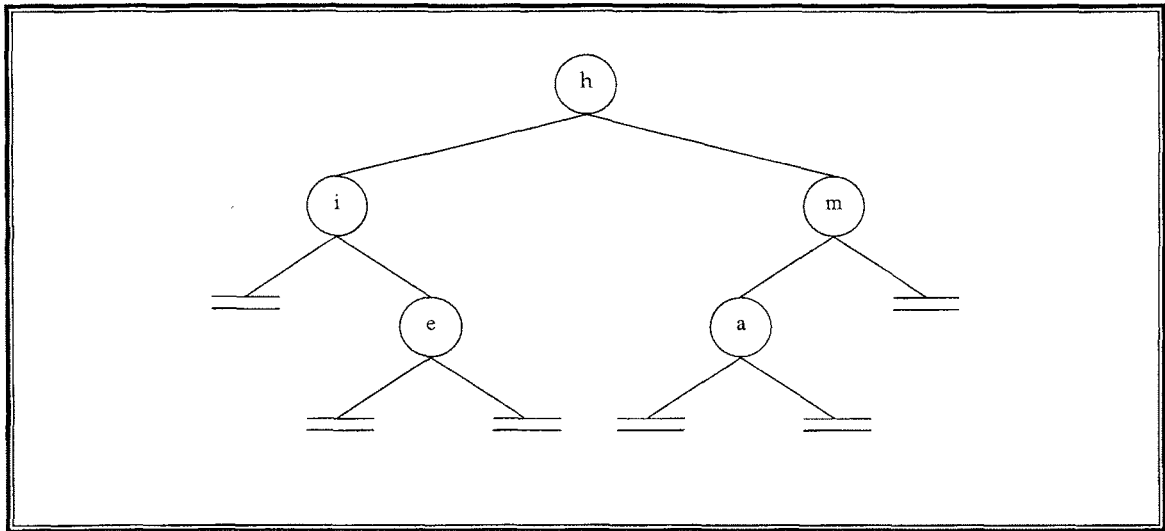
$$2h+1 \leq n \leq 2^{h+1} - 1$$

- Si es completo:

$$e \geq h+1$$

$$e = i+1$$

- Hay tres tipos de recorridos en profundidad en un árbol binario:



in-Orden:

Cada nodo se visita tras visitar su subárbol izquierdo y antes de visitar el derecho (izq, raíz, der):
 resultado del ejemplo: (i, e, h, a, m).

pre-Orden:

Primero se visita cada nodo, luego su subárbol izquierdo y finalmente el derecho (raíz, izq, der):
 resultado del ejemplo: (h, i, e, m, a).

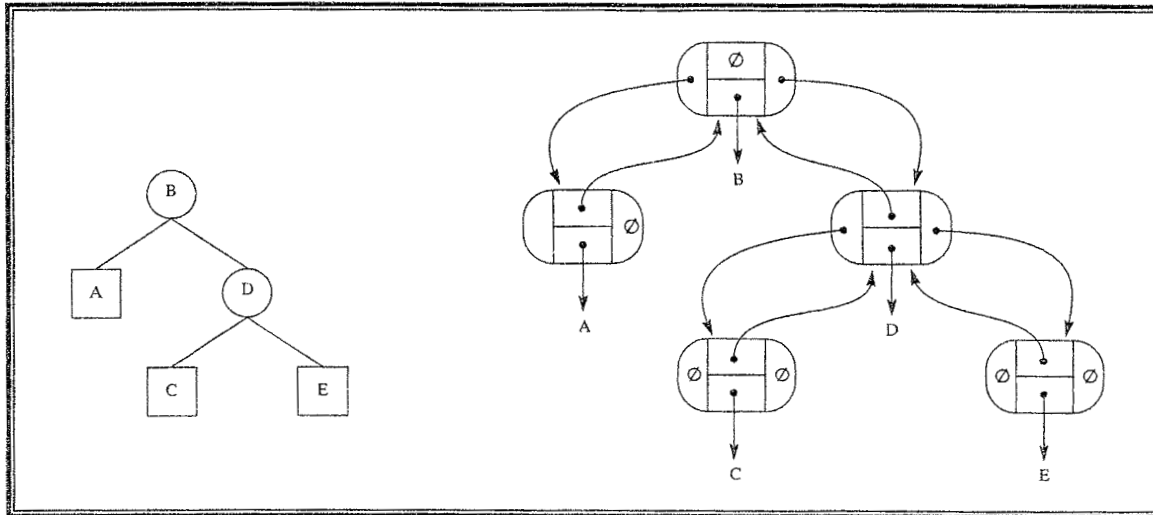
post-Orden:

Cada nodo se visita después de visitar su subárbol izquierdo y después de visitar el derecho (izq, der, raíz): resultado del ejemplo: (e, i, a, m, h).

Árbol como estructura dinámica.

Cada nodo contiene:

- Elemento.
- Nodo hijo izquierdo.
- Nodo hijo derecho.
- [Nodo padre].

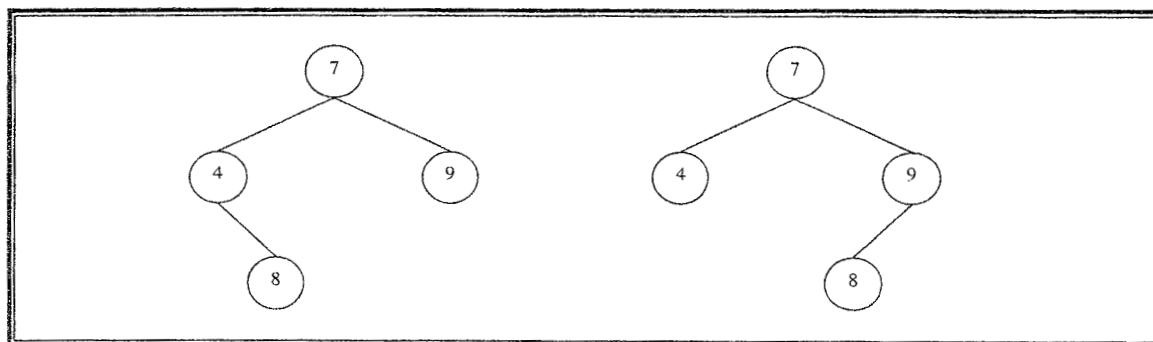


1.3.2. Árboles binarios de búsqueda (ABB).

Cada nodo tiene dos hijos:

- El subárbol izquierdo es el árbol vacío o es un subárbol que contiene nodos cuya clave es menor que la suya.
- El subárbol derecho es el árbol vacío o es un subárbol que contiene nodos cuya clave es mayor que la suya.

¿Cuál de estos árboles es un ABB?



Ventajas:

Almacenar estructuras lineales (que normalmente serían listas) mejorando la complejidad de las búsquedas:

- En el caso peor.
- En el caso medio.

Un ABB recorrido in orden permite obtener una lista de sus nodos ordenada (véanse algoritmos de búsqueda en ABB).

Inconvenientes:

- Los AB no ordenados son de poco interés. La falta de ordenación en un AB hace injustificable una estructura enlazada de árbol, prefiriéndose una lista.
- Un ABB puede llegar a degenerar en una lista. Solución: equilibrio en ABB.

El nodo se compone de clave, información y referencias a los subárboles izquierdo y derecho:

```
<arbol> ::= nulo | <nodo>
<nodo> ::= <clave> <info> <izq> <der>
<clave> ::= <dato> {<dato>}
<info> ::= {<dato>}
<izq> ::= <arbol>
<der> ::= <arbol>
```

El proceso de inserción es el encargado de garantizar que se cumple la condición de ABB.

INSERCIÓN DE NODOS EN ABB.

Los nodos se insertan siempre como nodos hoja.

El algoritmo de inserción garantiza para cada nodo del árbol que:

- Su subárbol izquierdo contiene claves menores.
- Su subárbol derecho contiene claves mayores.
- Si el árbol estuviera vacío, se inserta el nodo en la raíz.
- Si no, se va recorriendo el árbol:
 - En cada nodo se decide si hay que insertar a la derecha o la izquierda.
 - Si el subárbol en que hay que insertar es vacío, se inserta el nuevo elemento.
 - Si el subárbol en que hay que insertar no es vacío, hay que recorrerlo hasta encontrar el lugar que le corresponde al nodo en ese subárbol.

Es un algoritmo recursivo.

Búsqueda:

- Se va recorriendo el árbol.
- Si el nodo actual no es el buscado, se decide si hay que buscar por la derecha o la izquierda.

El algoritmo para encontrar el nodo o llegar al árbol vacío puede desarrollarse:

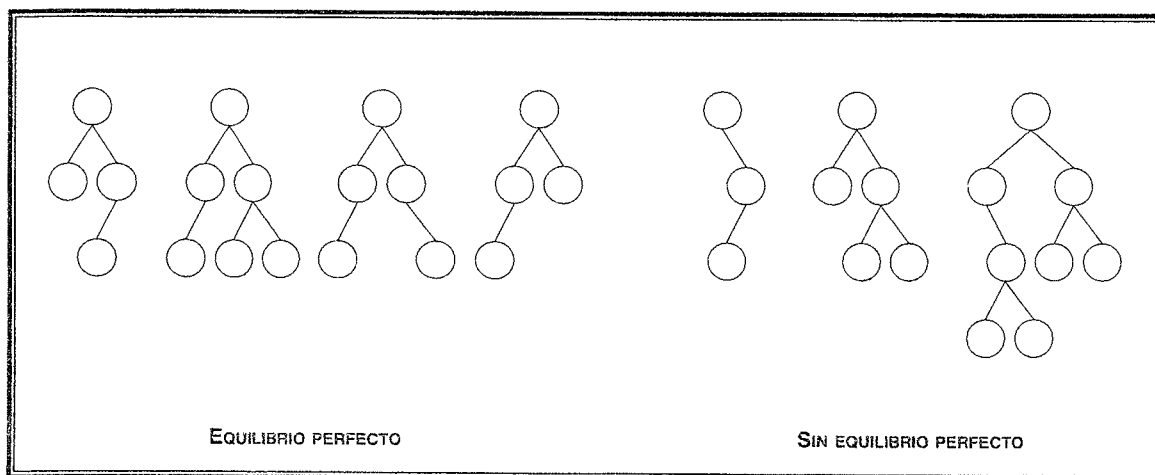
- Como algoritmo recursivo del nodo del árbol.
- Como algoritmo iterativo del árbol.

Borrado:

- Buscar el nodo a borrar.
- Si es un nodo hoja, basta con que su padre haga referencia al árbol vacío.
- Si no es nodo hoja, hay que sustituirlo por otro:
 1. El nodo a borrar sólo tiene un hijo: sustituirlo por su hijo.
 2. El nodo a borrar tiene dos hijos, sustituirlo por:
 - El mayor de su subárbol izquierdo, o
 - El menor de su subárbol derecho.
- Si el nodo a borrar es la raíz, hay que variarla aplicando el caso que corresponda.

Equilibrio perfecto.

Para cada nodo, el número de nodos del subárbol izquierdo y el número de nodos del subárbol derecho difieren como máximo en 1 unidad.



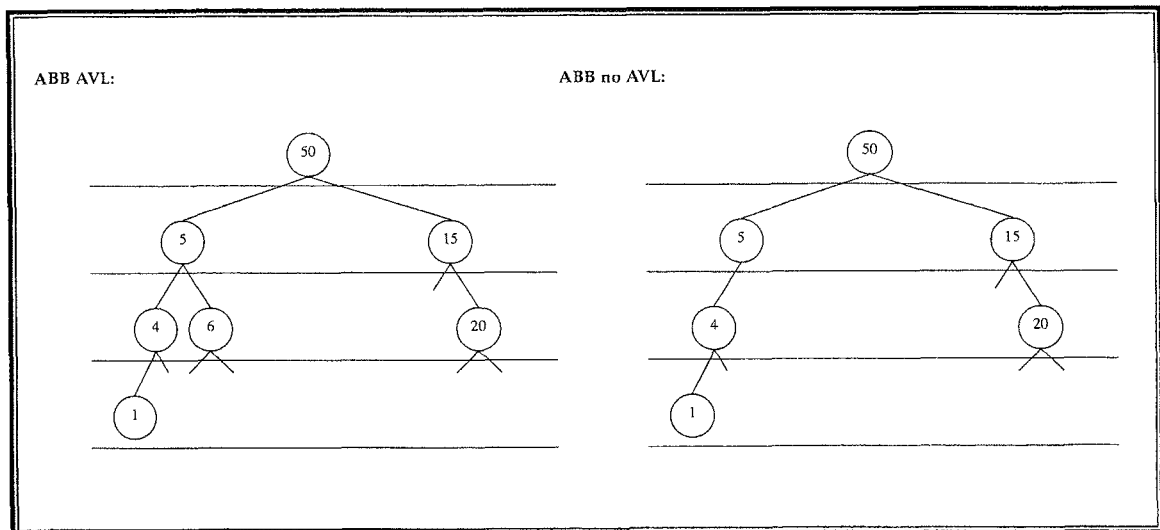
Ventajas e inconvenientes del equilibrio perfecto.

- Suponiendo que todas las claves se buscan con la misma probabilidad, cabe esperar una mejora en la longitud del camino medio de un 39 por 100 si el ABB está perfectamente equilibrado.
- La mejora puede ser mayor en el caso más desfavorable.
- Coste alto de mantener un ABB perfectamente equilibrado.
- La mejora no es suficientemente buena salvo si:
 - El caso más desfavorable se presenta con asiduidad.
 - Relación (núm. accesos)/(núm. inserciones) es muy grande.

AVL

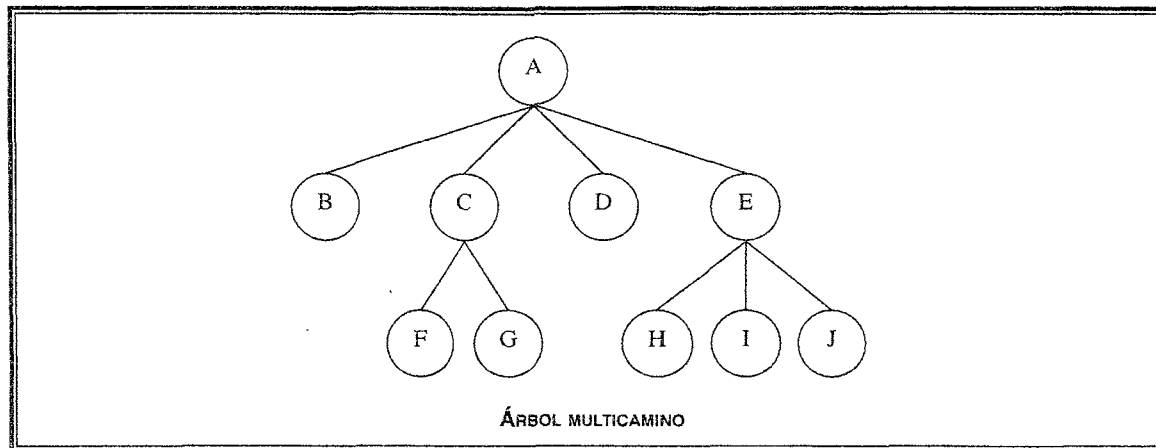
ABB equilibrado en altura (AVL).

Para cada uno de sus nodos, las alturas de sus subárboles izquierdo y derecho difieren como máximo en 1 unidad.



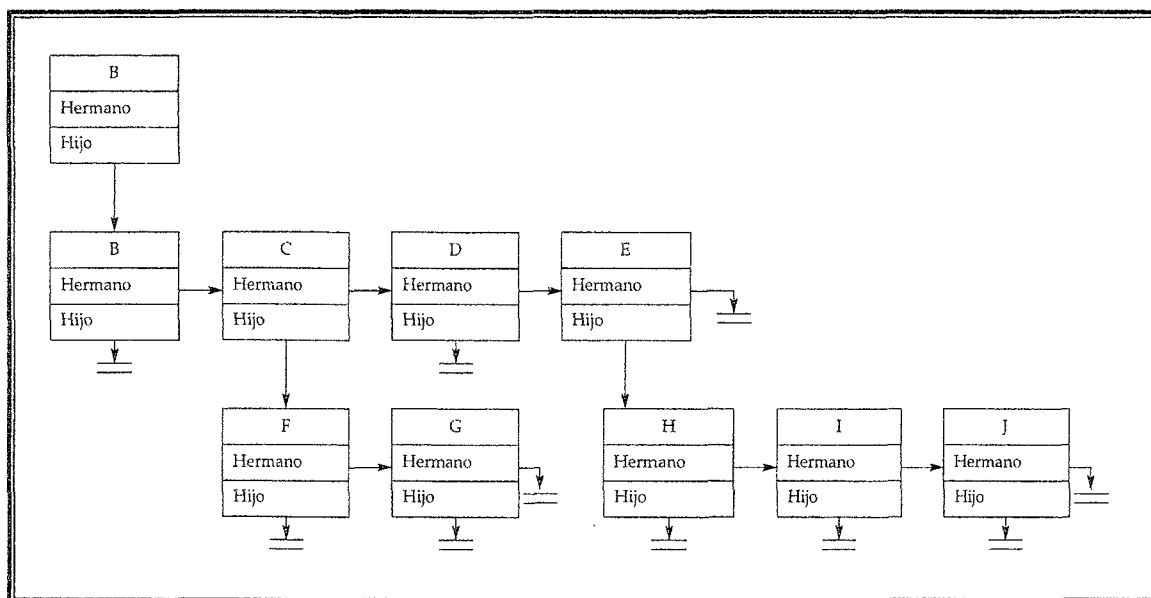
1.3.3. Árboles multicamino. B, B+, B*

Son aquellos cuyo grado es ≥ 2 .



Disponemos de dos formas para su implementación:

1. Diseñando los hijos como arrays de punteros. Tiene dos inconvenientes: desaprovecha memoria si el número de hijos es muy variable a no ser que se diseñe de manera dinámica y no debería usarse si el número de hijos es ilimitado.
2. Diseñando los hijos con punteros al hermano siguiente y a su hijo.



ÁRBOLES B:

Son árboles multicamino, de un orden determinado que se caracterizan por:

- Cada página tiene como máximo N nodos (orden del árbol).
- Cada página (excepto la raíz) tiene como mínimo $N/2$ nodos.

- Cada página es o bien una página hoja o bien tiene $m+1$ descendientes (m = núm. nodos que contiene).
- Todas las páginas hoja se encuentran al mismo nivel.

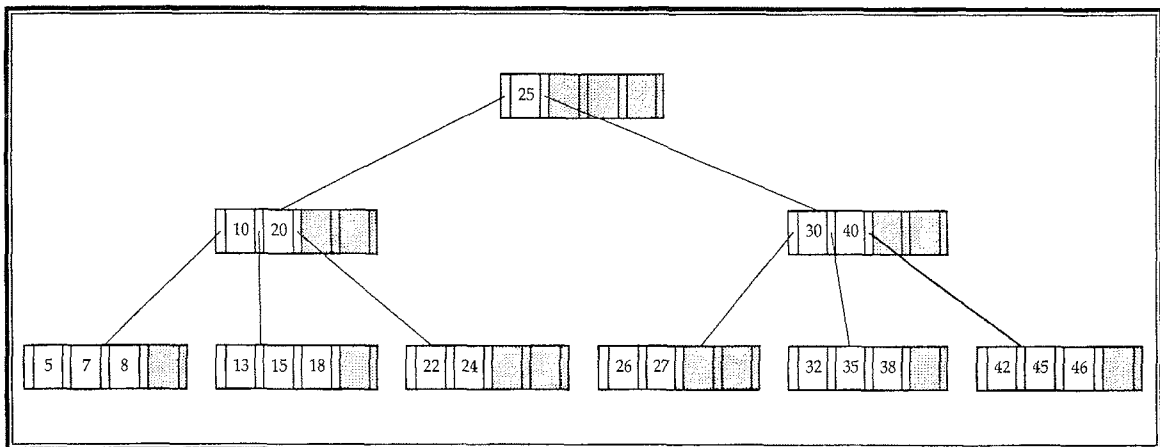
Estructura de una página con m claves:

p_0	k_1	p_1	k_2	p_2	...	k_m	p_m
-------	-------	-------	-------	-------	-----	-------	-------

$$k_i < k_{i+1} \quad \forall i = 1 \dots m, N/2 \leq m \leq N$$

$\forall i = 1 \dots m$, p_i apunta a una página cuyas claves son mayores o iguales que k_i y menores que k_{i+1} .

Ejemplo de árbol B de orden 4.



Definición formal.

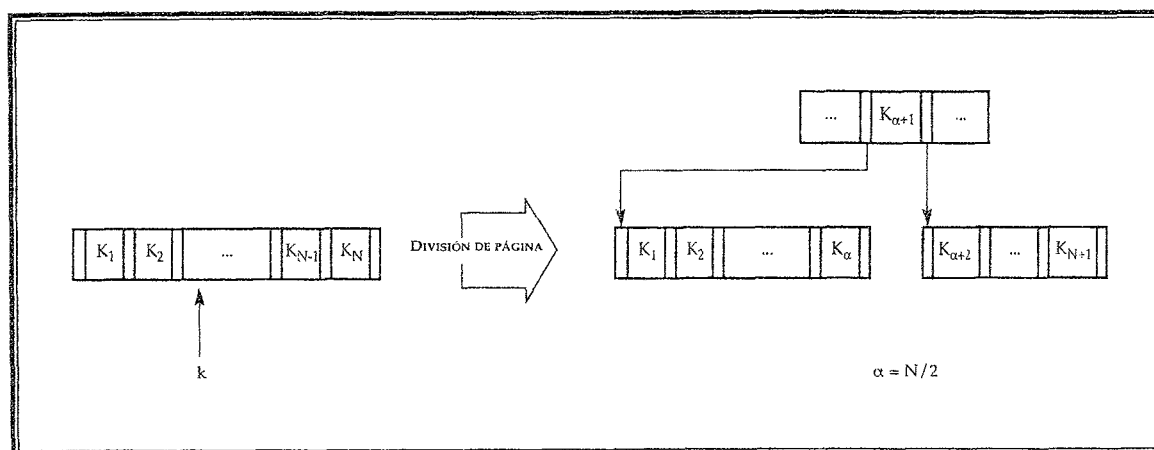
```

<arbolB> ::= <raiz> + {<pagina>}
<raiz> ::= <enlace>
<enlace> ::= (<<ReferenciaPagina>> | NULL)
<pagina> ::= <<numeroElementos>> + <antecesora> + <nodo>{<nodo>}
<antecesora> ::= <enlace>
<nodo> ::= <clave> + <informacion> + <siguiente>
<clave> ::= <<dato>>{<<dato>>}
<informacion> ::= {<<dato>>}
<siguiente> ::= <enlace>

```

Algoritmo de inserción.

1. Insertar una clave en el lugar que le corresponde.
2. Si se excede el número de claves permitidas, dividir la página en dos, promocionando hacia arriba la clave central.



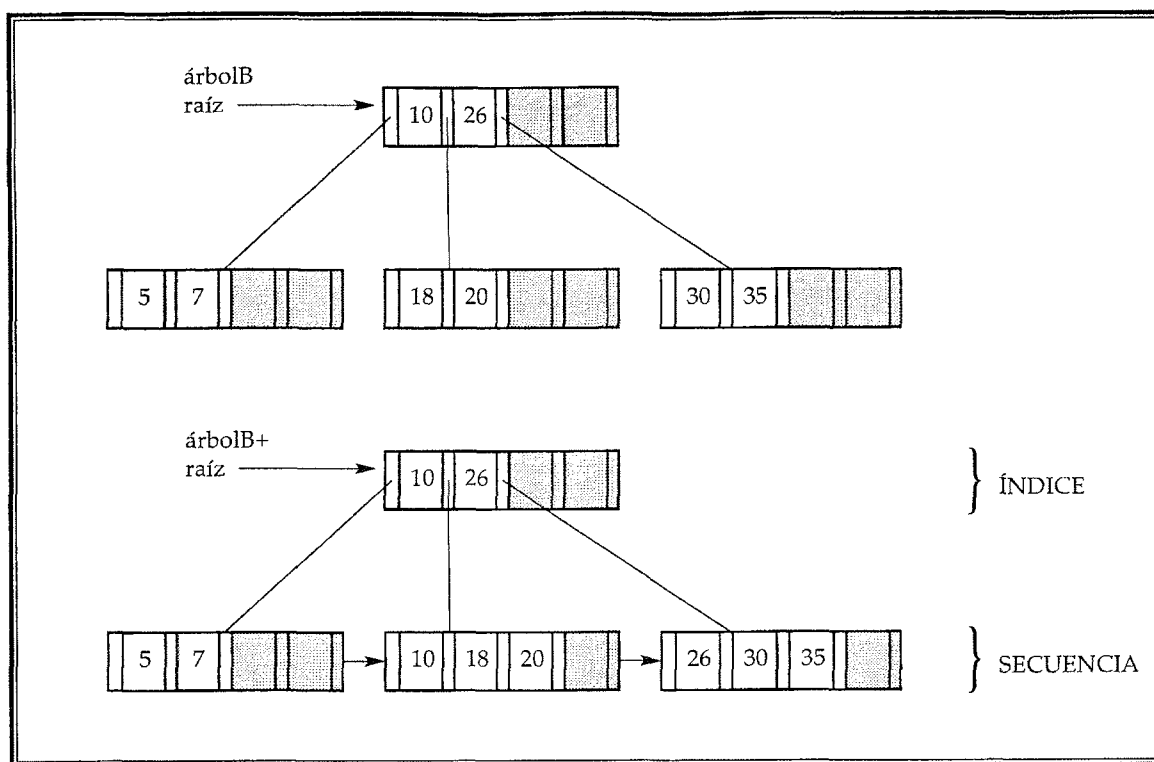
- Los árboles B crecen «hacia arriba» y decrecen «hacia abajo».
- La clave que sube puede dar lugar a una nueva división.
- Se asegura una ocupación en cada página 50 por 100.
- Las hojas siempre están al mismo nivel.

Variantes de los árboles B (B+, B*).

Árboles B+.

- Mismas características que árboles B, pero...
- Árboles formados por dos partes:
 - Índice: nodos interiores.
 - Secuencia: páginas hoja enlazadas secuencialmente en las que se repiten las claves interiores.
- Recorrido más fácil: basta con recorrer la secuencia.

EJEMPLO:



Árboles B*.

Características:

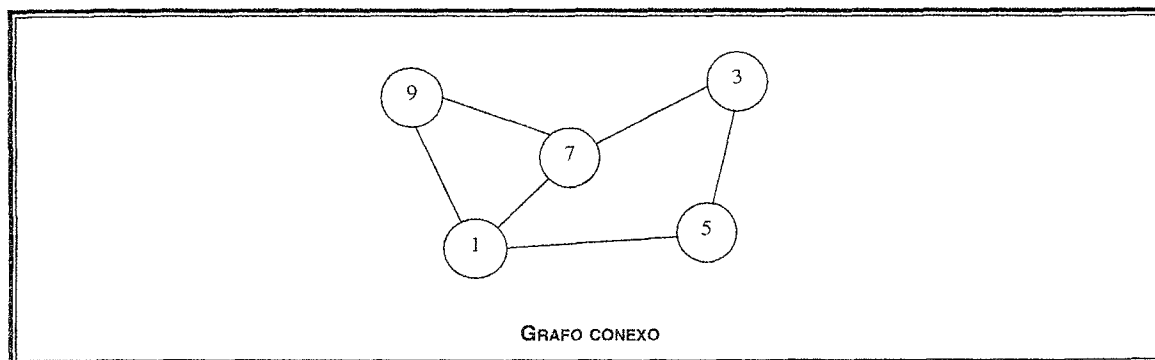
- Mejoran la eficiencia del acceso directo y la sobrecarga de reorganizar el árbol en la inserción y el borrado.
- Aseguran ocupación de páginas 66,6 por 100 (2/3).
- Cambia la división y fusión de páginas.

1.4. GRAFOS.

Fundamentos.

- Un grafo consiste en un conjunto de vértices o nodos y un conjunto de arcos. Se representa con el par $G = (V, A)$.
- Un arco o arista está formado por un par de nodos y se escribe (u, v) siendo u, v el par de nodos. Un grafo es dirigido si los pares de nodos que forman los arcos son ordenados, se representan $u \rightarrow v$. Un grafo no dirigido es aquel que los arcos están formados por pares de nodos no ordenados, se representa $u - v$.
- Si (u, v) es una arista en $A(G)$, entonces u y v se dice que son vértices adyacentes. Si (u, v) es una arista dirigida, entonces u se dice que es adyacente a v y v se dice que es adyacente de u .

- Un arco tiene, a veces, asociado un factor de peso, en cuyo caso se dice que es un grafo valorado.
- En un grafo no dirigido el grado de un nodo u , $\text{grado}(u)$, es el número de aristas que contienen a u .
- En un grafo dirigido se distingue entre grado de entrada y grado de salida; grado de entrada de un nodo u , $\text{gradent}(u)$, es el número de arcos que llegan a u , grado de salida de u , $\text{gradsal}(u)$, es el número de arcos que salen de u .
- Un camino P , en el grafo G , de longitud n desde un vértice u_0 a u_n es la secuencia de $n + 1$ vértices: $P = (u_0, u_1, u_2, \dots, u_n)$ tal que (u_i, u_{i+1}) son arcos de G para $0 \leq i \leq n$.
- Un camino $P = (u_0, u_1, u_2, \dots, u_n)$ es simple si todos los nodos que forman el camino son distintos, pudiendo ser iguales los extremos del camino.
- Un ciclo es un camino simple cerrado, $u_0 = u_n$, compuesto al menos por tres nodos.
- En algunos grafos se dan arcos desde un vértice a sí mismo (u, u) , el camino $u \rightarrow u$ se denomina bucle.
- Un grafo no dirigido es conexo si existe un camino entre cualquier par de nodos que forman el grafo.



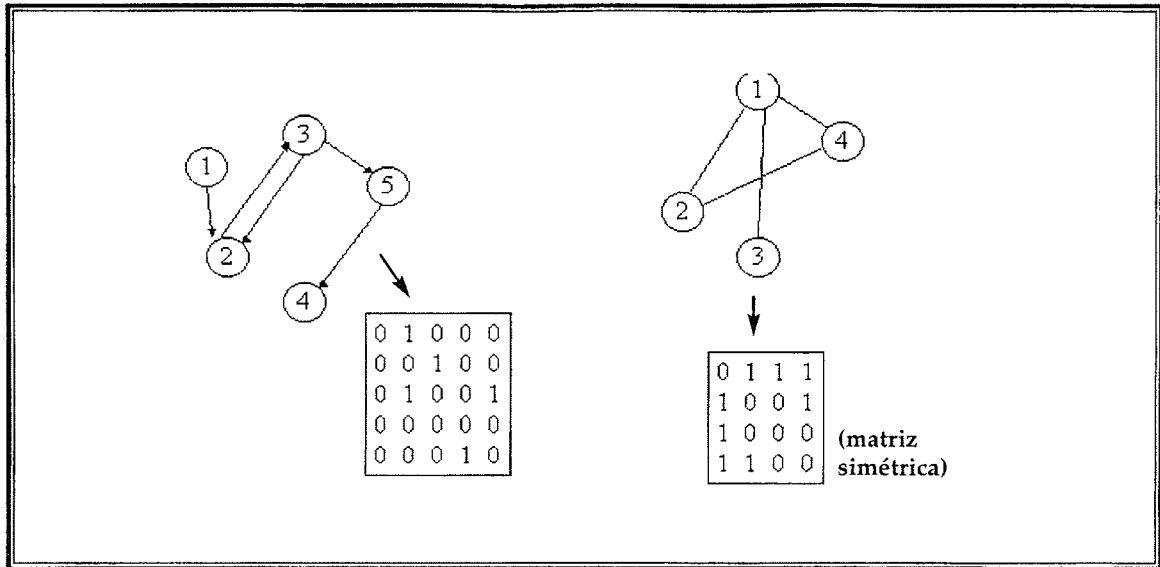
Representación de los grafos.

- Representación mediante arrays. Matrices de adyacencia.
- Representación mediante estructuras multienlazadas. Listas de adyacencia.

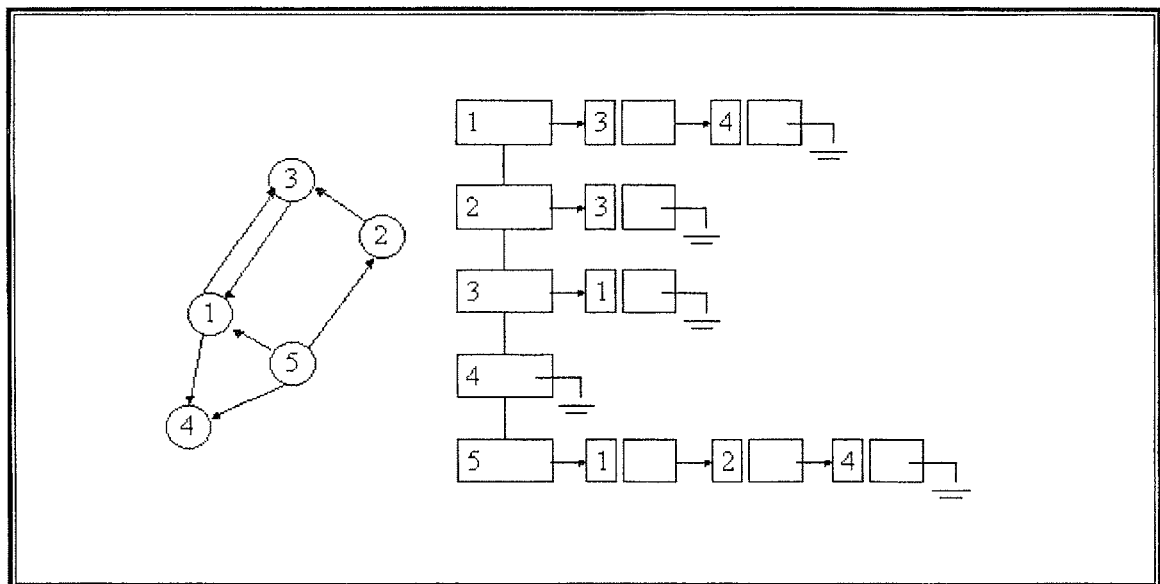
Matriz de adyacencia.

Sea $G = (V, A)$ un grafo de n nodos, suponemos que los nodos $V = \{u_1, \dots, u_n\}$ están ordenados y podemos representarlos por sus ordinales $\{1, 2, \dots, n\}$. La representación de los arcos se hace con una matriz A de $n \times n$ elementos a_{ij} definida:

$$a_{ij} = \begin{cases} 1 & \text{si hay arco } (u_i, u_j) \\ 0 & \text{si no hay arco } (u_i, u_j) \end{cases} \quad (\text{matriz de adyacencia})$$



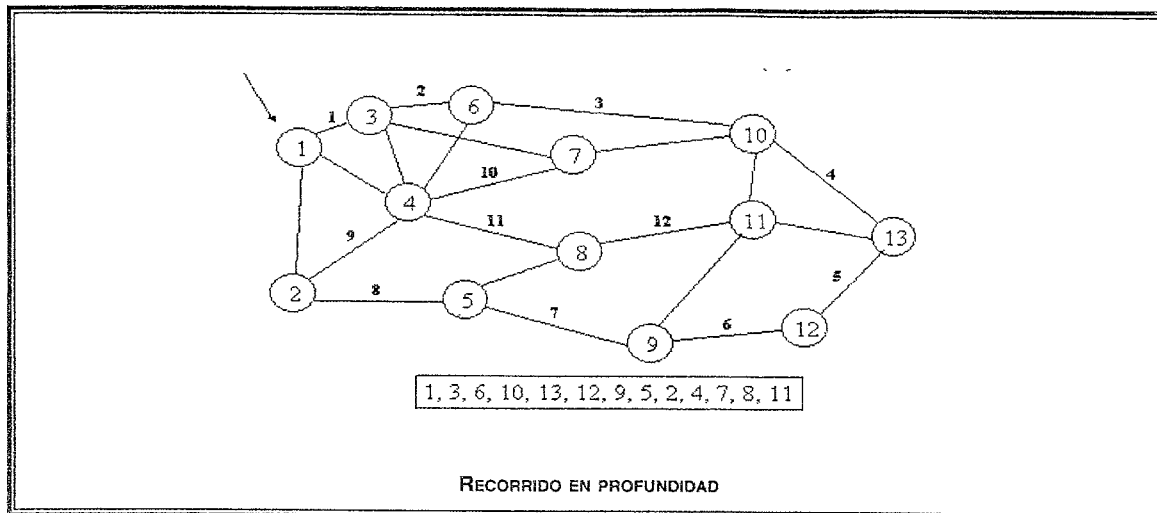
Listas de adyacencia.



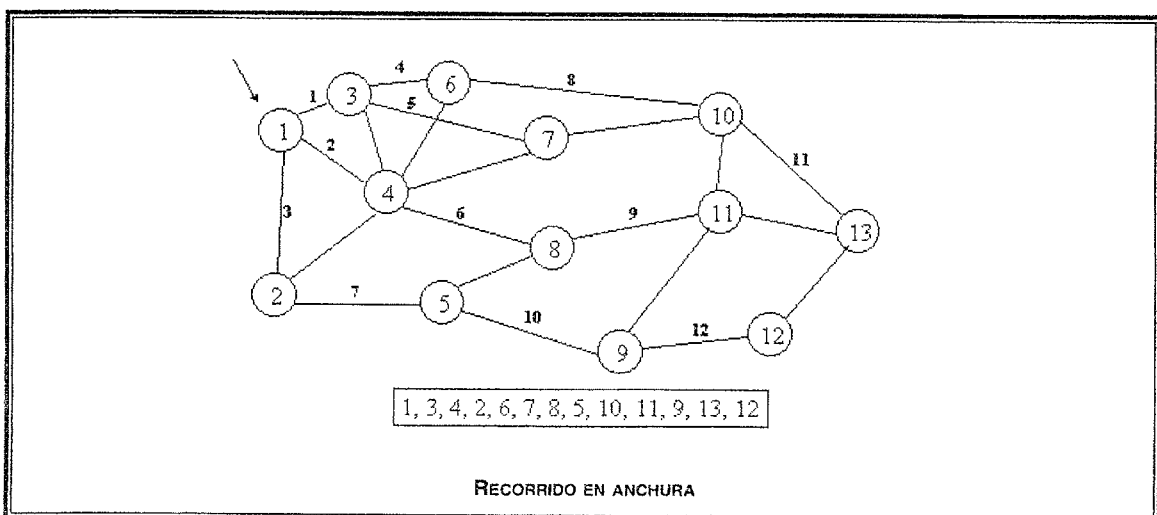
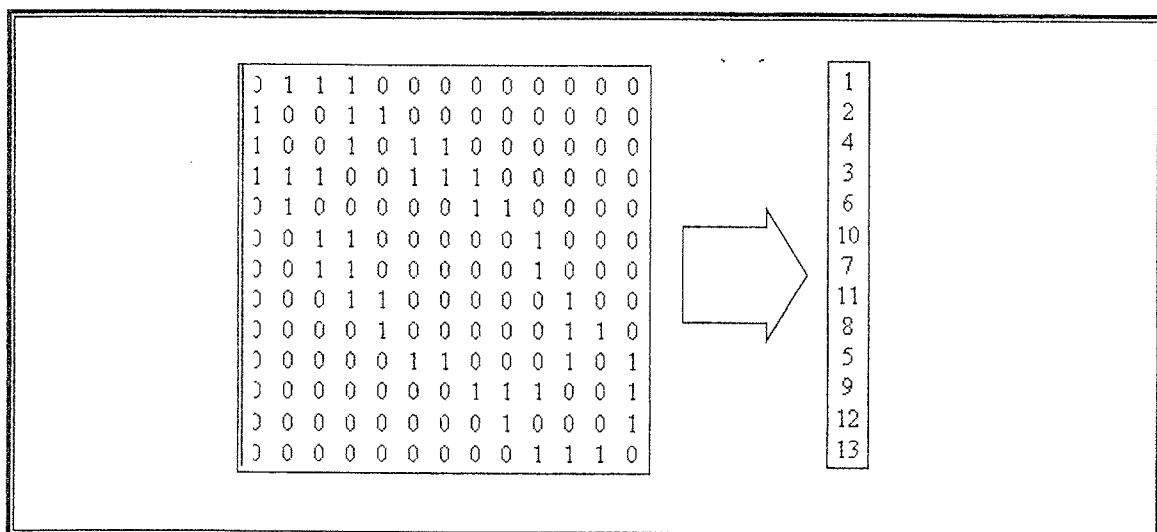
RECORRIDO DE UN GRAFO.

Recorrer un árbol consiste en visitar (procesar) cada uno de los nodos (aquellos alcanzables) a partir de un nodo dado.

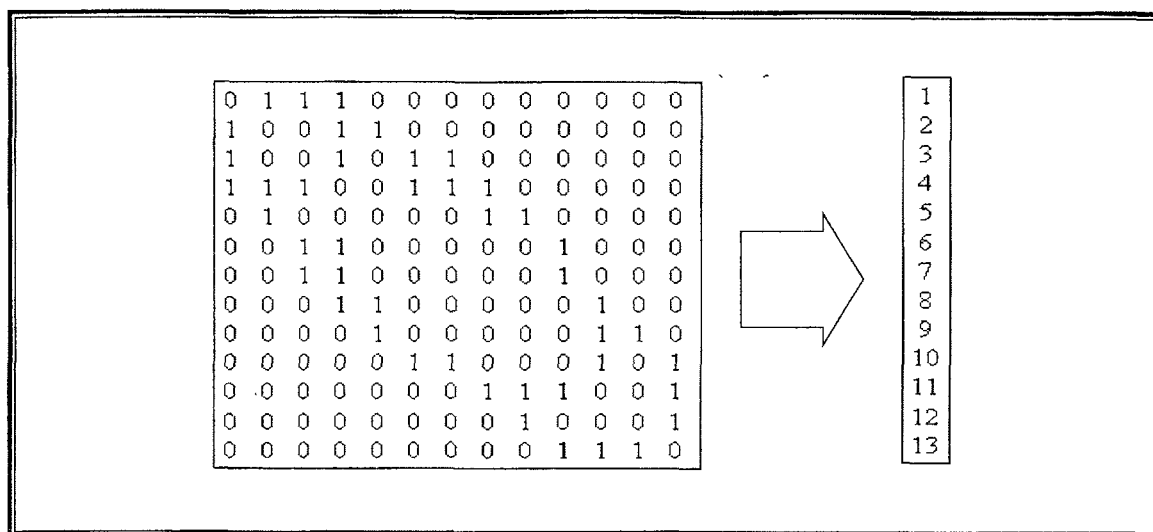
- Recorrido primero en anchura.
- Recorrido primero en profundidad.



Con la matriz de adyacencia.



Con la matriz de adyacencia



2. ALGORITMOS: RECURSIÓN, ORDENACIÓN, BÚSQUEDA.

2.1. EFICIENCIA DE UN ALGORITMO.

Cuando se nos propone realizar un programa para resolver un problema puede ser interesante plantearnos el diseño de varios algoritmos, y de entre éstos escoger el mejor. Pero rápidamente surge la pregunta: ¿cuál es el mejor? ¿con qué criterios se decide cuándo un algoritmo es mejor que otro?

Esta cuestión puede tener varias respuestas alternativas: aquel que consuma menos memoria, o el que finalice la ejecución más velozmente, aquel que funcione adecuadamente (que realice correctamente la misión para la cual ha sido diseñado) o incluso el que sea más fácil para un humano de leer, escribir o entender.

¿Cómo podemos realizar el estudio de la eficiencia de un algoritmo? Son dos las líneas básicas con las que podemos abordar dicho estudio:

- Empíricamente: programar los algoritmos y ejecutarlos varias veces con distintos datos de entrada.
- Teóricamente, lo que equivale a determinar matemáticamente la cantidad de recursos (tiempo de ejecución y memoria) requeridos por la implementación en función del tamaño de la entrada.

El resultado de un análisis teórico es genérico para cualquier tamaño de la entrada y depende exclusivamente de las instrucciones que componen el algoritmo y del citado tamaño. La salida de este análisis será una expresión matemática que indique cómo se produce el crecimiento del tiempo que tardaría en ejecutarse el algoritmo conforme aumente el tamaño de la entrada, cuestión que trataremos detalladamente.

El tiempo de ejecución de un algoritmo y su orden de eficiencia.

Una vez definido el tamaño de la entrada, resulta conveniente usar una función, $T(n)$, para representar el número de unidades de tiempo (segundos, milisegundos,...) que un algoritmo tardaría en ejecutarse con unos datos de entrada de tamaño n . Como el tiempo de ejecución de un programa depende claramente del ordenador que se utilice para medirlo y del traductor con el que se haya generado el código objeto, sería preferible que $T(n)$ no represente un tiempo, sino el número de instrucciones simples (asignaciones, comparaciones, operaciones aritméticas,...) que se ejecutan, o de forma equivalente, el tiempo de ejecución del algoritmo en un ordenador idealizado, donde cada una de las instrucciones simples consumen una unidad de tiempo. En general se suele dejar sin especificar las unidades empleadas en $T(n)$ y se asume que $n \geq 0$ y $T(n)$ es positivo.

Un concepto que nos ayudará a entender por qué podemos evitar el uso de unidades de tiempo en la función $T(n)$ es el que se conoce como el principio de invarianza, válido independientemente tanto del ordenador que estemos utilizando, como del compilador. El principio establece que dos implementaciones distintas de un mismo algoritmo, que toman $t_1(n)$ y $t_2(n)$ unidades de tiempo, respectivamente, para resolver un problema de tamaño n , no diferirán en eficiencia en más de una constante multiplicativa. Expresado matemáticamente, existe un $c > 0$ / $t_1(n) \leq ct_2(n)$.

Así, se podrá hacer que un programa vaya 10 ó 1.000 veces más rápido cambiando de máquina, pero sólo un cambio de algoritmo nos permitirá obtener una mejora mayor cuanto más crezca el tamaño de los ejemplos, lo que nos llevará a ignorar las constantes multiplicativas a todos los efectos.

Ahora bien, un mismo algoritmo puede ejecutarse con conjuntos de datos diferentes y su tiempo de ejecución puede ser distinto para cada uno de ellos: habrá datos de entrada con los que el algoritmo emplee más tiempo, y otros con los que finalice antes, por lo que es conveniente indicar si la expresión que se obtiene al realizar el análisis de la eficiencia corresponde al caso peor, al caso promedio, o al mejor caso. En el primero de ellos, estamos indicando un límite superior, el máximo valor, del tiempo de ejecución para cualquier entrada al algoritmo, al contrario que ocurre con el mejor caso, donde ofrecemos un límite inferior, indicando que el algoritmo no se ejecutará por debajo de ese tiempo. El caso promedio será una media ponderada de ambos casos, aunque los abundantes y a veces complicados cálculos para realizar un análisis de eficiencia del caso promedio complican notablemente su uso.

Aunque el peor caso suele ser bastante pesimista, y probablemente el comportamiento del algoritmo sea algo mejor que el obtenido por el análisis, se suele utilizar más a menudo que el mejor.

¿Cómo se calcula el tiempo de ejecución de un algoritmo? La respuesta es clara: sobre la base de las instrucciones que componen el algoritmo. Aunque posteriormente lo estudiaremos con detalle, esbozaremos seguidamente algunas ideas.

- La evaluación de una expresión tendrá como tiempo de ejecución lo que se tarde en realizar las operaciones que contenga.
- Una asignación a una variable simple de una expresión, al igual que una operación de escritura de una expresión, suele tardar el tiempo de evaluar dicha expresión más un tiempo constante relativo a gestión interna de la asignación o del proceso de escritura.
- Una lectura de una variable requiere un tiempo constante.
- Una secuencia de instrucciones, la suma de los tiempos de cada una de las instrucciones.

- En una sentencia condicional, su tiempo de ejecución será el de evaluar la condición más el máximo de los costes del bloque entonces y del bloque sino.
- Para un bucle, se evalúa el tiempo del cuerpo del bucle y se multiplica por el número de iteraciones más el tiempo de evaluar la condición del bucle.

Todas aquellas instrucciones cuyo tiempo de ejecución queda limitado superiormente por una constante, que sólo depende de la implementación, se denominarán operaciones elementales. Por tanto, al habernos desprendido de las constantes multiplicativas, para el análisis de la eficiencia de un algoritmo sólo será relevante el número de operaciones primitivas y no su duración.

Veamos un primer ejemplo para identificar intuitivamente estos conceptos. La siguiente función obtiene la posición donde se encuentra el mínimo valor de un vector de números enteros:

```
/* 1 */ int BuscarMinimo(int *Vector, int n)
/* 2 */ {
/* 3 */ int j, min;
/* 4 */ min= 0;
/* 5 */ for (j=1; j<n;j++)
/* 6 */ if (Vector[j] < Vector[min])
/* 7 */ min= j;
/* 8 */ return min;
/* 9 */ }
```

En el tiempo de ejecución de este programa no se consideran todas las instrucciones: las declaraciones de variables y la propia declaración de la función no intervienen, por lo que empezaremos por la línea 4 a hacer el cálculo del tiempo de ejecución.

- La línea 4 pertenece a una asignación, por lo que contabilizará una constante c_a .
- En la quinta línea se tendrán en cuenta dos constantes: una por el incremento del contador j , c_i , y otra por la evaluación de la condición booleana, c_e , que incrementarán el tiempo de ejecución tantas veces como se ejecute el bucle: $n-1$ veces, más dos veces adicionales correspondientes al caso en que $j = n+1$.
- El cuerpo del bucle podrá tener como coste c_e ó $c_e + c_a$, dependiendo de si la condición del if se evalúa verdadera o falsa, y se ejecuta o no el bloque then. Como hemos considerado trabajar siempre con el peor caso, tomaremos la suma del tiempo de evaluación de la condición (línea 6) más el de la asignación de la línea 7.
- Por último, nos queda la línea 8, que consumirá una constante c_r .

Agrupando todo lo anterior en una única expresión tendríamos:

$$T(n) = c_a + (c_i + c_e) (c_e + c_a) (n - 1) + c_r$$

Si consideramos que esas constantes son la unidad, por provenir de operaciones elementales, haciendo cálculos obtendremos $T(n) = 4 (n - 1) + 4 = 4n$.

En este momento ya estamos en condiciones de aclarar el concepto de eficiencia, el cual hace referencia a la forma en que el tiempo de ejecución (y en general cualquier recurso) necesario para procesar una entrada de tamaño n crece cuando se incrementa el valor de n . Es por esto, por lo que se especificará mediante una función de crecimiento. Como se puede observar, dicha eficiencia sólo depende del tamaño del problema, dejando a un lado cuestiones como la velocidad del ordenador y la eficiencia del compilador.

Se dice que un algoritmo necesita un tiempo de ejecución del orden de una función cualquiera $f(n)$, cuando existe una constante positiva c y una implementación del algoritmo que resuelve cada instancia del problema en un tiempo acotado superiormente por $cf(n)$, siendo la función $f(n)$ la que marca cómo crecerá dicho tiempo de ejecución cuando aumente n . Un algoritmo será, por tanto, más eficiente que otro si el tiempo de ejecución del peor caso tiene un orden de crecimiento menor que el segundo.

Notación asintótica O .

La notación O mayúscula, $O[f(n)]$, representa el conjunto de funciones g que crecen como mucho tan rápido como f , o lo que es lo mismo, las funciones g tales que f llega a ser en algún momento una cota superior para g . En definitiva se trata de buscar una función sencilla, $f(n)$, que acote superiormente el crecimiento de otra $g(n)$, en cuyo caso se notará como $g(n) \in O[f(n)]$ (g es del orden de f). La definición formal es la siguiente:

sea $f: \mathbb{N} \rightarrow \mathbb{R} + \cup \{0\}$ una función cualquiera,

el conjunto de las funciones del orden de $f(n)$, notado como $O[f(n)]$, se define:

$$O[f(n)] = \{ g \mid \exists c_0 \in \mathbb{R}^+ \text{ y } \exists n_0 \in \mathbb{N}, \forall n \geq n_0 \ g(n) \leq c_0 f(n) \}$$

Esta definición garantiza que si el tiempo de ejecución de una implementación de un algoritmo es $g(n)$, la cual es del orden de $f(n)$, el tiempo $g'(n)$ empleado por cualquier otra implementación que difiera de la primera en el lenguaje y en el compilador utilizado, o la propia máquina, también será del orden de $f(n)$.

Veamos algunos ejemplos:

$T(n) = 3n + 2 \in O(n)$, ya que existen dos constantes positivas $n_0 = 2$, y $c_0 = 4$, tal que $3n + 2 \leq 4n$.

$T(n) = 1.000n^2 + 100n - 6 \in O(n^2)$, porque existen $n_0 = 100$, y $c_0 = 1.000$ que hacen que se cumpla que $1.000n^2 + 100n - 6 \leq 1.001n^2$.

$T(n) = 6 \times 2^n + n^2 \in O(2^n)$ debido a que se pueden encontrar dos constantes $n_0 = 4$, y $c_0 = 7$ que hacen que $6 \times 2^n + n^2 \leq 7 \times 2^n$.

Funciones de complejidad en tiempo más usuales:

Las funciones de complejidad algorítmica ordenadas de mayor a menor eficacia son:

- $O(n) = 1$: constante, es la complejidad mas deseada.
- $O(n) = \log n$: logarítmica. Esta complejidad suele aparecer en determinados algoritmos con iteración o recursión no estructural (por ejemplo búsqueda binaria).
- $O(n) = n$: Lineal. En general esta complejidad es buena y bastante usual. Suele aparecer en la evaluación de un bucle simple cuando la complejidad de las operaciones interiores es constante o en algoritmos con recursión estructural.
- $O(n) = n \log n$: quasilineal. También aparece en algoritmos con recursión no estructurada.
- $O(n) = n^2$: cuadrática. Aparece en bucles o recursiones doblemente anidados.
- $O(n) = n^3$: cúbica, aparece en bucles o recursiones triples.
- $O(n) = n^3$: polinómica ($n > 3$). Si K crece, la complejidad del algoritmo es bastante mala.
- $O(n) = n^k$: exponencial. Debe evitarse en la medida de lo posible. Puede aparecer en un subprograma recursivo que contenga dos o más llamadas internas. En problemas donde aparece esta complejidad suele hablarse de explosión combinatoria.

2.2. RECURSIVIDAD.

La recursividad es una técnica de resolución de problemas muy potente ya que muchos problemas que a primera vista parecen poseer una solución difícil, son resueltos de manera sencilla y elegante, incluso inmediata de forma recursiva. Este método divide el problema original en varios más pequeños que son del mismo tipo que el inicial, procediendo seguidamente a encontrar la solución de estos subproblemas, soluciones en las que se basará la del problema inicialmente planteado. Para ello, realiza de nuevo las correspondientes divisiones en problemas más pequeños aún, hasta que éstos tengan un tamaño tan pequeño para que su solución se conozca de forma directa. Una vez conocidas estas soluciones, podremos ir resolviendo sucesivamente los problemas más grandes hasta llegar a la solución buscada del problema primeramente planteado. Es de vital importancia que existan dichas soluciones a los problemas más sencillos, porque nos permitirán, a partir de ellas, ir resolviendo los problemas más complejos.

De manera general, aquellos problemas que puedan ser resueltos de forma recursiva tendrán las siguientes características:

- Los problemas pueden ser redefinidos en términos de uno o más subproblemas, idénticos en naturaleza al problema original, pero de alguna forma menores en tamaño.
- Uno o más subproblemas tienen solución directa o conocida, no recursiva.
- Aplicando la redefinición del problema en términos de problemas más pequeños, dicho problema se reduce sucesivamente a los subproblemas cuyas soluciones se conocen directamente.
- La solución a los problemas más simples se utiliza para construir la solución al problema inicial.

Cuando plasmamos la solución de un problema de manera recursiva en un algoritmo diremos entonces que ha sido resuelto mediante un algoritmo recursivo. Este algoritmo, o en general cualquier módulo, tendrá que realizar una o varias llamadas a sí mismo.

Es muy importante que siempre se determine en qué momento se detendrán las llamadas recursivas del módulo, ya que si no se hace, se corre el riesgo de no llegar nunca a la solución del problema inicial por estar llamándose el módulo a sí mismo de forma infinita.

Una vez que hemos diseñado el algoritmo recursivo, el siguiente paso dentro del proceso de desarrollo del software es realizar su implementación en un lenguaje de programación, aunque no todos están preparados para ello. Lenguajes como BASIC, FORTRAN o COBOL no permiten la implementación de algoritmos recursivos, por lo que se descartará este método de resolución de problemas en caso de tener que utilizarlos, buscando vías alternativas. Otros como C, C++, Pascal o Java sí incluyen esta característica.

Si una función F contiene una llamada explícita a sí misma, entonces se dice que es recursiva de forma directa, pero si F posee una referencia a otra función Q, que a su vez contiene una llamada a F, entonces F es recursiva de forma indirecta.

La primera implementación de una función recursiva que vamos a tratar es el factorial de un número, la cual se obtiene de manera directa a partir de la propia definición del factorial (en este caso es una función recursiva directa):

```
long factorial (long n)
{
  if (n == 0) return 1;
  else return n * factorial(n-1);
}
```

En esta función podemos claramente determinar dos partes principales:

1. La llamada recursiva, que expresa el problema original en términos de otro más pequeño.
2. El valor para el cual se conoce una solución no recursiva. Esto es lo que se conoce como caso base: una instancia del problema cuya solución no requiere de llamadas recursivas.

A la hora de resolver recursivamente un problema, son cuatro las preguntas que nos debemos plantear:

- [P1] ¿Cómo se puede definir el problema en términos de uno o más problemas más pequeños del mismo tipo que el original?
- [P2] ¿Qué instancias del problema harán de caso base?
- [P3] Conforme el problema se reduce de tamaño ¿se alcanzará el caso base?
- [P4] ¿Cómo se usa la solución del caso base para construir una solución correcta al problema original? Apliquemos esta técnica de preguntas y respuestas para diseñar una función recursiva que obtenga el valor del n-ésimo término de la secuencia de Fibonacci. La secuencia es la siguiente: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

Se puede observar que el tercer término de la sucesión se obtiene sumando el segundo y el primero. El cuarto, a partir de la suma del tercero y el segundo. El problema es calcular el valor del n ésimo término de la solución, que se obtendrá sumando los términos $n-1$ y $n-2$.

Las respuestas a la preguntas anteriores serían:

[P1] $\text{fibonacci}(n) = \text{fibonacci}(n-1) + \text{fibonacci}(n-2)$.

[P2] En este caso hay que seleccionar como casos bases $\text{fibonacci}(1) = 1$ y $\text{fibonacci}(2) = 1$.

[P3] En cada llamada a la rutina `fibonacci` se reduce el tamaño del problema en uno o en dos, por lo que siempre se alcanzará uno de los dos casos bases.

[P4] $\text{fibonacci}(3) = \text{fibonacci}(2) + \text{fibonacci}(1) = 1 + 1$. Se construye la solución del problema $n = 2$ a partir de los dos casos bases.

Teniendo en cuenta estas respuestas estamos preparados para implementar la función `fibonacci` en C:

```
int fibonacci(int n)
{
    if ((n == 1) || (n == 2)) return 1;
    else return (fibonacci(n-2) + fibonacci(n-1));
}
```

Esta función posee dos peculiaridades: tiene más de un caso base y hay más de una llamada recursiva, denominándose a este tipo recursión no lineal, en contraposición a la recursión lineal producida cuando sólo hay una única llamada recursiva. Generalmente, la recursión no lineal requiere la identificación de más de un caso base.

La pila del ordenador y la recursividad.

Ahora vamos a estudiar cómo gestiona un ordenador las llamadas recursivas cuando éste ejecuta un módulo recursivo. Así, comprendiendo este funcionamiento, seremos conscientes de los problemas que se nos pueden plantear si el diseño del módulo recursivo no es correcto.

La memoria de un ordenador a la hora de ejecutar un programa queda dividida en dos partes: la zona donde se almacena el código del programa y la zona donde se guardan los datos, utilizada esta última fundamentalmente en las llamadas a rutinas y que se denomina pila (en inglés *stack*).

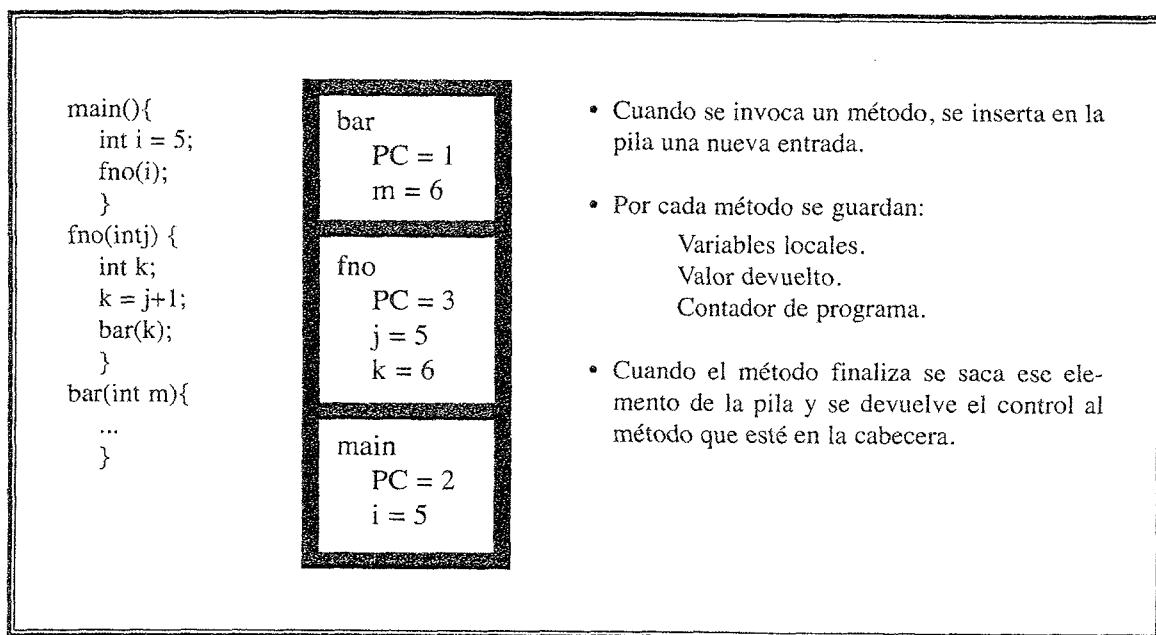
Cuando un programa principal llama a una rutina M , se crea en la pila lo que se denomina el registro de activación o entorno E , asociado al módulo M . El registro de activación almacena información como constantes y variables locales del módulo, así como sus parámetros formales y el contador del programa.

Conforme se van llamando de manera anidada a las rutinas, se van creando sucesivos registros de activación, almacenándose de forma apilada: si $M1$ llama a $M2$ y a su vez, invoca a $M3$, la pila estará formada por un primer registro de activación correspondiente a $M1$, $E1$, sobre el que se reservará es-

pacio para el de M2, E2, y posteriormente para el de M3, E3, conviviendo los tres entornos mientras se esté ejecutando M3.

Cuando finaliza la ejecución del último módulo invocado, se devuelve el control a M2, y el espacio ocupado por su entorno se libera, quedando ahora la pila compuesta sólo por dos registros de activación. Al acabar M2 se elimina de la pila el entorno E2, devolviéndose el control a M1 y quedando sólo el registro E1 correspondiente a ese último módulo, que desaparecerá, a su vez, cuando acabe su ejecución.

Veamos un ejemplo. aunque no se trata de un algoritmo recursivo la invocación entre métodos distintos tiene el mismo efecto en la pila que si éstos fueran recursivos.



Como se puede observar, este trozo de memoria recibe el nombre de pila por la forma en que se gestiona, ya que en él se encuentran los entornos apilados en el orden en que han sido llamados, de tal manera que el registro activo en cada momento es el que está situado en la cabecera de la pila, ocupando así el lugar más alto de esta hipotética pila. Esta forma de gestionar ese espacio de memoria permite que el lenguaje de programación correspondiente pueda utilizar la recursividad.

En la pila existirán tantos entornos de una misma función como llamadas recursivas se hayan efectuado, siendo todos ellos independientes y distintos entre sí. Así, llamamos profundidad de recursión de un módulo recursivo al número de entornos que están presentes en la pila en un momento dado.

El concepto de profundidad se suele usar como elemento de decisión para optar por una solución recursiva de un problema, ya que problemas que requieran profundidades muy grandes pueden originar que se llene la memoria del ordenador dedicada a la pila.

¿Recursividad o iteración?

La recursividad es una herramienta muy poderosa para resolver problemas, los cuales resueltos iterativamente podrían tener una resolución muy compleja, sobre todos aquellos cuya propia definición es recursiva, aunque esto no asegura que dichos algoritmos recursivos posean una eficiencia alta, ya que suelen consumir un mayor tiempo de cálculo.

En general, si un problema se puede describir en términos de versiones más pequeñas de él mismo, entonces la recursividad permite expresarlo, en la mayoría de los casos, y por tanto implementarlo más fácilmente. De igual forma, cuando la recursividad nos permita solucionar problemas cuyas soluciones iterativas sean difíciles de implementar, utilizaremos esta primera técnica.

A pesar de estas ideas expresadas en el párrafo anterior, hay dos aspectos que influyen en la ineficiencia de algunas soluciones recursivas, y que tras evaluar convenientemente debemos decidir si son inconvenientes sustanciales como para decidir buscar una solución iterativa al problema:

- a) El tiempo asociado con la llamada a las rutinas es una cuestión a tener en cuenta, ya que en cada llamada se aloja en la pila un nuevo entorno, con el consiguiente tiempo adicional consumido en esta operación. En una rutina iterativa la operación se realiza sólo una vez y se puede considerar a este tiempo irrelevante con respecto a la ejecución total de la rutina, pero una simple llamada recursiva inicial puede generar un gran número de llamadas posteriores, penalizando sustancialmente el tiempo final de ejecución con el tiempo total de creación de los entornos. En el momento de diseñar un algoritmo recursivo hay que decidir si la facilidad de elaboración del algoritmo merece la pena con respecto al costo en tiempo de ejecución que incrementará la gestión de la pila originado por el gran número de posibles llamadas recursivas.
- b) La ineficiencia inherente de algunos algoritmos recursivos. Por ejemplo, fijémonos en el módulo que resuelve la sucesión de Fibonacci. Si se trazan las llamadas recursivas de fibonacci (6) podremos apreciar cómo fibonacci (4) se calcula dos veces como problemas separados, fibonacci (3) se obtiene tres veces; fibonacci (2), cinco veces y fibonacci (1), tres. Es de suponer, según estas evidencias, que conforme aumente n , el número de cálculos repetidos que se realizarán será mucho mayor, presentando una gran cantidad de procesamiento duplicado que no es aprovechado para evitar llamadas recursivas posteriores.

Así, es de especial importancia que la rutina recursiva implemente algún tipo de técnica que evite realizar procesamiento duplicado en diferentes partes de la ejecución de la rutina. Una solución podría ser la utilización de algún tipo de estructura de datos adicional, como por ejemplo un vector o una lista, que mantuviera toda la información calculada hasta el momento, para evitar de esta forma, cálculos redundantes. Adicionalmente, y relacionado con la gestión de la pila en la recursividad, cabe destacar que otro problema puede ser el planteado por la profundidad de la recursión.

2.3. BÚSQUEDA.

Con el objeto de liberarnos de la necesidad de elegir una representación específica adoptamos la convención algorítmica de hacer referencia a la clave i -ésima como $k(i)$ y a la clave del registro apuntado por p como $k(p)$. De igual forma, hacemos referencia al registro correspondiente como $r(i)$ o $r(p)$. De esta manera podemos concentrar nuestra atención en los detalles de la técnica en lugar de los de la implementación.

2.3.1. Búsqueda secuencial.

La forma más simple de búsqueda es la búsqueda secuencial. Esta búsqueda es aplicable a una tabla organizada, ya sea como un vector o como una lista ligada. Supongamos que k es un vector de n claves, de $k(0)$ a $k(n-1)$ y r es un vector de registro de $r(0)$ a $r(n-1)$ de tal manera que $k(i)$ es la clave de $r(i)$. Supongamos también que key es un argumento de búsqueda. Queremos obtener el entero i más pequeño tal que $k(i)$ sea igual a key si existe tal i y -1 en caso contrario. El algoritmo para hacerlo es el siguiente.

```
for (i = 0; i < n; i++)
    if (key == k(i))
        return(i);
return(-1);
```

Supongamos que la tabla está organizada como una lista lineal ligada apuntada por $table$ y ligada mediante un campo puntero $next$. Entonces, suponiendo k , r , key y rec como antes, la búsqueda secuencial en una lista ligada puede escribirse de la siguiente manera:

```
q = null;
for(p = table; p != null && k(p) != key; p = next(p))
    q = p;
return (p);
```

Eficiencia de la búsqueda secuencial.

El número de comparaciones depende del lugar de la tabla donde aparece el registro que tienen la clave del argumento. Si el registro es el primero de la tabla, se realiza una sola comparación; si el registro es el último, se necesitan n comparaciones. Si es igualmente probable que el argumento aparezca en cualquier posición dada en la tabla, una búsqueda exitosa haría (en el promedio) $(n+1)/2$ comparaciones y, una infructuosa n comparaciones. En cualquier caso el número de comparaciones es $O(n)$.

Reordenamiento de una lista para alcanzar máxima eficiencia de búsqueda.

Así, sería de gran ayuda tener un algoritmo que reordenara de manera continua la tabla, de tal forma que los registros que se accedan con mayor frecuencia estuvieran al frente y los que se accedan con menor frecuencia al final.

Hay dos métodos de búsqueda para realizar lo anterior. Uno de ellos se conoce como método de moverse-al-frente y es eficiente en el caso de una tabla como una lista. En este método, siempre que una búsqueda es exitosa, el registro recuperado se elimina de su localización actual de la lista y se coloca a la cabeza de la misma.

El otro método se llama transposición, en el cual un registro recuperado se intercambia con el registro que lo precede de manera inmediata.

Búsqueda en una tabla ordenada.

Si la tabla se almacena en orden ascendente o descendente de las claves de los registros pueden usarse varias técnicas para mejorar la eficiencia de la búsqueda. Esto es cierto en especial si la tabla es de tamaño fijo. Una ventaja obvia de la búsqueda en un archivo ordenado se tiene cuando la clave del argumento no está presente en el archivo. En el caso de un archivo ordenado suponiendo que las claves argumentos están distribuidas de manera uniforme sobre el rango de claves del archivo, se necesitan (en promedio) sólo $n/2$ comparaciones. Esto ocurre porque sabemos que una clave está faltando en un archivo ordenado de manera ascendente tan pronto como encontremos una clave que sea mayor que el argumento.

Búsqueda secuencial indexada.

Hay otra técnica para perfeccionar la eficiencia de la búsqueda en un archivo ordenado, pero involucra un incremento en la cantidad de espacio requerido. Este método se llama método de búsqueda secuencial indexada. Se aparta una tabla auxiliar, llamada index además del propio archivo ordenado. Cada elemento en el index consta de una clave kindex y un puntero al registro del archivo que corresponde a kindex. Los elementos en el índice tanto como los elementos en el archivo tienen que estar ordenados de acuerdo con las claves (véase organización secuencial indexada de ficheros).

2.3.2. Búsqueda binaria.

El método de búsqueda más eficiente en una tabla secuencial sin usar índices o tablas auxiliares es el de búsqueda binaria. Básicamente, se compara el argumento con la clave del elemento medio de la tabla. Si son iguales, la búsqueda termina con éxito; en caso contrario se busca de manera similar en la mitad superior o inferior de la tabla.

La mejor manera de definir la búsqueda binaria es la forma recursiva. Sin embargo la recarga asociada a la recursividad puede hacerla inapropiada para su uso en situaciones prácticas en las cuales la eficiencia se considera primordial. En consecuencia, presentamos la siguiente versión no recursiva del algoritmo de búsqueda binaria:

```
low = 0;
hi = n - 1;
while (low <= hi) {
    mid = (low + hi)/2;
    if (key == k(mid))
        return(mid);
    if (key < k(mid))
        hi = mid - 1;
    else
        low = mid + 1;
} /* fin de while */
return (-1);
```

Cada comparación en la búsqueda binaria reduce el número de posibles candidatos en un factor de dos. Así, el número máximo de comparaciones de claves es de manera aproximada $\lg n$ (en realidad es $2 \cdot \lg n$ dado que en C, se hacen cada vez dos comparaciones de claves a través del ciclo: $\text{key} == k(\text{mid})$ y $\text{key} < k(\text{mid})$). Así, podemos decir que el algoritmo de búsqueda binaria es $O(\lg n)$.

Obsérvese que la búsqueda binaria se puede usar en conjunción con la organización secuencial indexada. En lugar de buscar en el índice de manera secuencial, se puede usar una búsqueda binaria. La búsqueda binaria también puede usarse al buscar en la tabla principal una vez que los dos registros frontera hayan sido identificados. Sin embargo, es probable que el tamaño de este segmento de tabla sea tan pequeño que la búsqueda binaria no sea más ventajosa que una búsqueda secuencial.

Por desgracia, el algoritmo de búsqueda binaria sólo puede usarse si la tabla está almacenada como un vector. Esto ocurre porque hace uso del hecho de que los índices de los elementos del vector son enteros consecutivos. Por esta razón, la búsqueda binaria es prácticamente inútil en situaciones donde se requieran muchas eliminaciones e inserciones.

2.3.3. Búsqueda por interpolación.

Otra técnica para buscar en un vector ordenado es la llamada búsqueda por interpolación. Si las claves están distribuidas de manera uniforme entre $k(0)$ y $k(n-1)$, el método puede ser aún más eficiente que la búsqueda binaria.

En principio, como en la búsqueda binaria, low se hace 0 y high se hace $n-1$, y a través del algoritmo, se sabe que la clave argumento key está entre $k(\text{low})$ y $k(\text{high})$. Suponiendo que las claves están distribuidas de manera uniforme entre esos dos valores, se esperaría que key estuviese de forma aproximada en la posición:

$$\text{mid} = \text{low} + (\text{high} - \text{low}) * [(\text{key} - k(\text{low})) / (k(\text{high}) - k(\text{low}))]$$

Si key es menor que $k(\text{mid})$ haga high igual a $\text{mid}-1$; si es mayor, haga low igual a $\text{mid}+1$.

Repetir el proceso hasta que la clave haya sido encontrada o $\text{low} > \text{high}$.

En efecto si las claves están distribuidas de manera uniforme a lo largo del vector la búsqueda por interpolación requiere un promedio de $\lg(\lg n)$ comparaciones y es raro que requiera muchas más, comparado con la búsqueda binaria que requiere $\lg n$. Sin embargo, si las claves no están distribuidas de manera uniforme, la búsqueda por interpolación puede tener un comportamiento promedio muy pobre. En el peor de los casos, el valor de mid puede ser igual a $\text{low}+1$ o $\text{high}-1$ en cuyo caso la búsqueda por interpolación degenera en búsqueda secuencial. En situaciones prácticas las claves tienden con frecuencia a aproximarse en torno a ciertos valores y no están distribuidas de forma uniforme. Por ejemplo hay más nombres que comiencen por s que por q. En situaciones tales la búsqueda binaria es muy superior a la interpolación.

Vamos a ver un ejemplo con números para que nos aclare todo esto. Si tenemos la siguiente secuencia de valores en un vector:

Posición	0	1	2	3	4	5	6	7	8	9
Valor	1	2	3	4	5	6	7	8	9	10

y queremos buscar un elemento en el, por ejemplo el valor de clave 3, tendríamos lo siguiente:

Valores iniciales de las variables:

low = 0

high = (10 - 1) = 9

key = 3

Calculamos mid con la fórmula anterior y obtenemos:

$$\text{mid} = 0 + (9 - 0) * [(3 - k(0)) / [k(9) - k(0)]]$$

Sustituimos los valores de las k, que son:

k(0) = 1

k(9) = 10

y obtenemos que $\text{mid} = 0 + 9 * 2 / 9 = 2$,

C comparamos el valor de k(2) con key (el buscado) y como son iguales acabamos.

2.3.4. Árboles binarios de búsqueda.

La búsqueda en árboles binarios es un método de búsqueda simple, dinámico y eficiente considerado como uno de los fundamentales. De forma que terminología sobre árboles, tan sólo recordar que la propiedad que define un árbol binario es que cada nodo tiene a lo más un hijo a la izquierda y otro a la derecha. Para construir los algoritmos consideremos que cada nodo contiene un registro con un valor clave a través del cual efectuamos las búsquedas.

Un árbol binario de búsqueda es un árbol binario con la propiedad de que todos los elementos almacenados en el subárbol izquierdo de cualquier nodo x (incluyendo la raíz) son menores que el elemento almacenado en x, y todos los elementos almacenados en el subárbol derecho de x son mayores que el elemento almacenado en x.

Obsérvese la interesante propiedad de si se listan los nodos del árbol binario de búsqueda en inorden nos da la lista de nodos ordenados. Esta propiedad define un método de ordenación similar al Quicksort, con el nodo raíz jugando un papel similar al del elemento partición del Quicksort aunque con los árboles binarios de búsqueda hay un gasto extra de memoria debido a los punteros.

La propiedad de árbol binario de búsqueda hace que sea muy simple diseñar un procedimiento para realizar la búsqueda. Para determinar si k está presente en el árbol la comparamos con la clave si-

tuada en la raíz, r. Si coincide la búsqueda finaliza con éxito, si $k < r$ es evidente que k, de estar presente, a de ser un descendiente del hijo izquierdo de la raíz, y si es mayor será un descendiente de hijo derecho. La función puede ser codificada fácilmente de la siguiente forma:

```
#define BINARIO_VACIO NULL
#define NODO_NULO NULL
typedef int TEtiqueta;

typedef struct tipo_celdaB{
    TEtiqueta etiqueta;
    struct tipo_celdaB * hizqda;
    struct tipo_celdaB * hdrcha;
    struct tipo_celdaB * padre;
} tipo_celdaB;
typedef tipo_celdaB * TNodeB;
typedef TNodeB TArbolB;

int pertenece(TEtiqueta e, TNodeB n ){
    if (n == NODO_NULO) return(0);
    else if (e == n ->etiqueta) return(1);
        else if (e < n -> etiqueta) return(pertenece(e, n->hizqda);
            else return(pertenece(e, n->hdrcha);
    }
}
```

El procedimiento de construcción de un árbol binario de búsqueda puede basarse en un procedimiento de inserción que vaya añadiendo elementos al árbol. Tal procedimiento inserta (e, n, T) comenzaría mirando si $T == \text{BINARIO_VACIO}$ y de ser así se crearía un nuevo árbol con un nodo para e y dejaría T apuntando hacia él. Si T no esta vacío se busca e como lo hace el procedimiento pertenece, sólo que al encontrar un puntero a NODO_NULO durante la búsqueda, se reemplaza por un puntero a un nodo nuevo que contenga e. El código podría ser el siguiente:

```
void inserta(TEtiqueta e, TNodeB n, TArbolB T)
{
    TNodeB nodo;
    if (T == BINARIO_VACIO) T= CrearB(e, NULL, NULL);
    else if (e < n->etiqueta)
    {
        if (n->hizqda == NULL)
        {
            nodo = (TNodeB) malloc (sizeof (tipo_celdaB));
            if (!nodo){printf ("error no hay memoria");exit (1);}
            nodo->padre=n;
            nodo->hizqda=NULL;
            nodo->hdrcha=NULL;
        }
    }
}
```

```

        nodo->etiqueta=e;
        n->hizqda = nodo;
    }
    else inserta(e, n->hizqda, T);
}
else if (e > n->etiqueta)
{
    if (n->hdrcha == NULL)
    {
        nodo = (TNodoB) malloc (sizeof (tipo_celdaB));
        if (!nodo){printf ("error no hay memoria");exit (1);}
        nodo->padre=n;
        nodo->hizqda=NULL;
        nodo->hdrcha=NULL;
        nodo->etiqueta=e;
        n->hdrcha = nodo;
    }
    else inserta(e, n->hdrcha, T);
}
}

```

2.4. CLASIFICACIÓN.

Los algoritmos más simples de ordenación requieren un tiempo $O(n^2)$ para clasificar n objetos. Uno de los algoritmos de clasificación más populares es la clasificación rápida (quicksort), que tiene un tiempo promedio de $O(n \lg n)$. El quicksort funciona muy bien para la mayor parte de las aplicaciones; sin embargo, en el peor de los casos tiene un tiempo de $O(n^2)$. Existen otros algoritmos, como la clasificación por montículos (heapsort) y la clasificación por intercalación (mergesort) que lleva un tiempo $O(n \lg n)$ en el peor de sus casos, aunque su comportamiento en el caso promedio es peor que el del quicksort. El mergesort se puede utilizar en la clasificación externa. Existen otros algoritmos de clasificación llamados por urnas o por cubetas, los cuales se pueden utilizar sólo con clases especiales de datos, requiriendo un tiempo de $O(n)$ en el peor de los casos.

2.4.1. Algoritmos simples de clasificación.

Clasificación por intercambio. Burbuja (Bubblesort).

Uno de los algoritmos de clasificación más simples puede ser el de la burbuja (bubblesort). La idea básica de éste es imaginar que los registros a ordenar están almacenados en un vector vertical. Se recorre varias veces el vector de abajo hacia arriba, y al hacer esto si hay dos elementos adyacentes que no están en orden se invierten. El efecto producido por esta operación es que en el primer recorrido es que el registro con valor clave menor, sube al primer valor del vector. En el segundo recorrido el valor de la clave menor siguiente, pasa a la segunda posición del vector, y así sucesivamente. Si el vector A es un tipo_registro $A[n]$, n es el número de registros, y el campo clave contiene el valor de la clave de cada registro, tendríamos el siguiente algoritmo:

```

PARA (I=1, I<N-1, I++)
PARA (J=N-1; J>=I; J--)
SI A[J].CLAVE<A[J-1].CLAVE)
    TEM=A[J]
    A[J]=A[J-1]
    A[J-1]=TEM
FIN SI
FIN PARA
FIN PARA

```

Clasificación por inserción.

En este método de clasificación se llama así, porque en el i -ésimo recorrido se inserta el i ésimo elemento $A[i]$ en el lugar correcto, entre $A[1]$, $A[2]$, ..., $A[i-1]$, los cuales fueron ordenados previamente. Después de hacer esta inserción, se encuentran clasificados los registros colocados en $A[0]$.. $A[i]$. Esto se resume es:

```

PARA (I=1, I<N-1, I++)
    TEM=A[I]
    K=I-1
    MIENTRAS ((A[I].CLAVE>TEM.CLAVE) Y K>0)
        A[K+1]=A[K]
        K=K-1
    FIN MIENTRAS
    SI A[K].CLAVE>TEM.CLAVE
        A[K+1]=A[K]
        A[K]=TEM
    SINO
        A[K+1]=TEM
    FIN SI
FIN PARA.

```

Clasificación por selección.

La idea es que en el i -ésimo recorrido se selecciona el registro con la clave más pequeña, entre $A[i]$, ..., $A[n-1]$, y se intercambia con $A[i]$. Como resultado, después de i pasadas, los i registros menores ocupan $A[0]$, ..., $A[i]$, en el orden clasificado. Esta clasificación la podemos describir de la siguiente forma:

```

PARA (K=0, K<N-2, K++)
    I=K
    TEM= A[K]
    PARA (J=K+1; J<N-1; J++)
        SI A[J].CLAVE<TEM.CLAVE)
            I=J
    TEM= A[I]

```

```

FIN SI
FIN PARA
    A[I]=A[K]
    A[K]=TEM
FIN PARA

```

Eficiencia en los algoritmos simples de clasificación.

Puede observarse que el método de selección directa aparece como el más eficiente de los tres en general, debido a que hace menos movimientos de componentes. En realidad podría haberse previsto este resultado, porque no parece eficiente realizar los movimientos masivos de componentes propios de los algoritmos de inserción directa y de burbuja. Estos dos métodos solo son aconsejables si se sabe que el vector está inicialmente casi ordenado. El método de inserción es mejor que el de la burbuja, por lo que apenas hay razones para usar este método (salvo su sugerente nombre y su sencillez de memorización).

Se debe tener presente que los algoritmos mencionados en esta sección tienen un tiempo de ejecución $O(n^2)$, tanto en el peor de los casos como para el caso promedio. Así para una n grande, ninguno de estos algoritmos se compara de modo favorable con los algoritmos $O(n \lg n)$ que se analizarán en la siguiente sección. Una regla razonable es que a menos que n sea aproximadamente 100, puede ser una pérdida de tiempo implantar un algoritmo más complicado que los estudiados en esta sección.

2.4.2. Algoritmos rápidos de clasificación.

La esencia del método consiste en clasificar un vector $A[0], \dots, A[n-1]$ tomando en el vector un valor clave v como elemento pivote, alrededor del cual reorganizar los elementos del vector. Es de esperar que el pivote esté cercano a la mediana de la clave del vector. Se permutan los elementos del vector con el fin de que para alguna j , todos los registros con claves menores que v aparezcan en $A[0], \dots, A[j]$, y todos aquellos con claves mayores aparezcan en $A[j+1], \dots, A[n-1]$. Después se aplica recursivamente el Quicksort a $A[0], \dots, A[j]$ y a $A[j+1], \dots, A[n-1]$ para clasificar ambos grupos de elementos. Dado que todas las claves del primer grupo preceden a todas las claves del segundo grupo, todo el vector quedará ordenado.

Eficiencia del Quicksort.

El algoritmo tiene en promedio un tiempo de $O(n \lg n)$ para ordenar n elementos, y en el peor de los casos tiene $O(n^2)$.

3. ORGANIZACIÓN DE FICHEROS.

Un fichero es un conjunto de registros organizados según unas reglas lógicas y almacenados en determinados soportes físicos. Este almacenamiento se realizará atendiendo unas determinadas reglas.

El porqué de utilizar registros es para poder acceder a unidades homogéneas de información dentro del fichero y de forma individualizada.

Al conjunto de reglas utilizadas para introducir información en forma de registros en los soportes de almacenamiento denominaremos organización de ficheros.

Básicamente los dos modos de organizar los ficheros son:

1. Organización Secuencial, en este tipo de organización se caracteriza fundamentalmente por almacenar los registros en posiciones físicas contiguas de memoria.
2. Organización directa, también llamada aleatoria, caracterizada porque el almacenamiento físico de los registros se hace a través de un identificativo o clave que indicará la posición del registro en el fichero, el valor de la posición de almacenamiento en el fichero puede obtenerse directamente del valor del identificativo o después de haber aplicado un algoritmo de transformación sobre éste.

Cualquiera que sea el tipo de organización utilizada, cada registro que forme parte del mismo fichero siempre tendrá un campo, subcampo, conjunto de campos o conjunto de subcampos que discriminen o identifiquen a ese registro frente a los demás del fichero. Este campo ha de ser común a todos los registros del fichero y se llamará identificativo o clave. No sería necesaria hacer ninguna diferenciación entre los distintos usos que se pueden hacer de un identificativo, aunque de todas formas es conveniente saber que, al menos, el identificativo se puede utilizar de dos formas. Se puede utilizar como:

- IDENTIFICATIVO DE ORDEN: utilizado para introducir los registros en el fichero.
- IDENTIFICATIVO DE BÚSQUEDA: utilizado para localizar registros dentro de un fichero, atendiendo a una determinada identidad o cualidad.

Consideraremos el identificativo de orden únicamente en el caso de creación de ficheros, es decir, en el caso de introducción de registros dentro del fichero. El de búsqueda lo utilizaremos solamente en aquellos casos de localizar registros dentro de un determinado fichero. Además, los identificativos que utilizaremos serán en la mayoría de los casos identificativos monocampo, es decir, identificativos o claves, formados por un solo campo del registro.

Independientemente de la organización que se utilice, hay que saber que cuando nos refiramos a secuencia lógica, hablaremos de qué forma se almacenan los registros atendiendo al valor de alguno de sus campos, es decir, dependiendo del contenido. Cuando nos refiramos a secuencia física, hablaremos de cuál es el orden que siguen los registros dentro del soporte, es decir, las posiciones en las que están almacenados.

3.1. ORGANIZACIÓN SECUENCIAL.

Si estuviésemos almacenando registros en un fichero con este tipo de organización, tendríamos que tener cuenta que el fichero, una vez grabado no tiene por qué quedar ordenado. Si el fichero quedase ordenado sería porque el propio usuario controló que la secuencia lógica de grabación del fichero se efectúe de forma ordenada. En los dos casos, el usuario no tiene por qué preocuparse de la secuencia física de almacenamiento ya que ésta se realiza en posiciones físicamente contiguas.

En este tipo de organización, el identificativo solamente indicará cuál será el campo clave por el que accederemos al registro una vez que deseemos consultarlo, pero no necesariamente indicará el orden del registro dentro del fichero. El orden de los registros, como el nombre de la organización indica, será secuencial, es decir, uno detrás de otro.

Este tipo de organización generará los registros en el mismo orden en que se grabaron y el acceso a ellos siempre será, siguiendo esa misma secuencia de grabación.

FICHERO

1	A	0
25	B	1
139	M	2
89	D	3
123	E	4
123	E	5
198	L	6
204	H	7
1	A	8

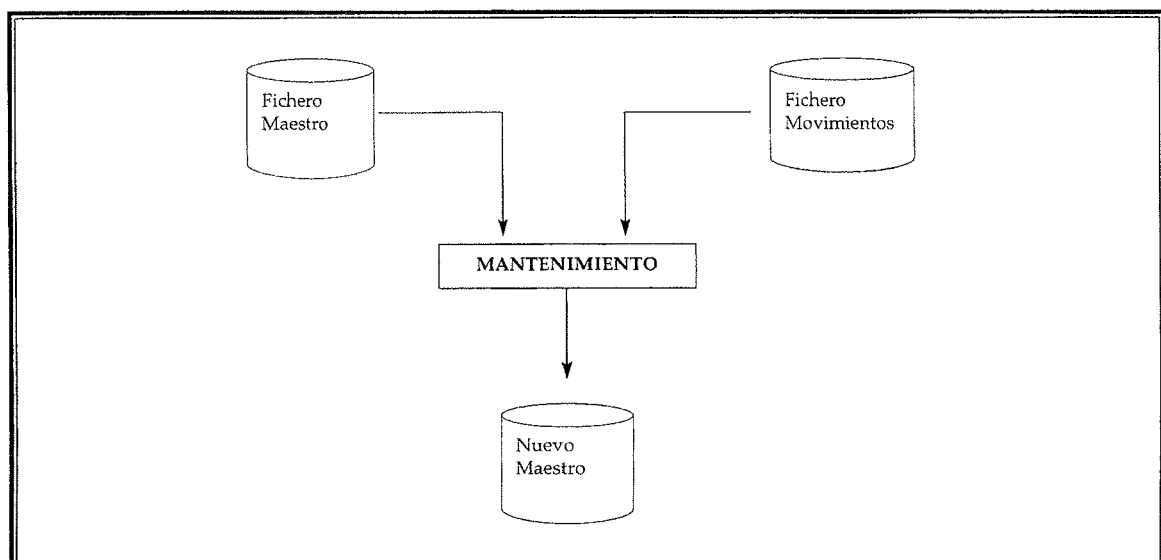
REGISTRO

Datos
Claves Direcciones

Suponiendo que queremos consultar una gran parte del fichero, o la mayoría de sus registros la organización secuencial presenta una ventaja y es la rapidez con la que se puede acceder a registros colocados en posiciones físicamente contiguas.

Por contra, si queremos consultar solamente unos pocos registros o un solo registro, este tipo de organización no es la más idónea, ya que habría que consultar secuencialmente todos los registros situados en posiciones anteriores al registro o registros buscados.

Hay otro problema que presenta la organización secuencial, y que se hace patente cuando surge la necesidad de insertar nuevos registros, o de modificar cualquiera ya existente. Esto implica que en cualquiera de los dos casos el fichero tendrá que ser creado de nuevo, ya que cuando se utiliza un fichero secuencial, se utiliza o bien para leer o bien para escribir, pero nunca para leer y escribir simultáneamente. Por ello, si queremos insertar un registro, la única solución es crear un fichero nuevo en el que se copien los registros antiguos, más los que queremos añadir. Si queremos modificar la solución será la de generar un nuevo fichero con los datos de los registros no modificados y con los datos de los registros modificados.



Actualmente existen formas de utilizar los ficheros con organización secuencial que permiten añadir registros, al final del mismo (APPEND en basic, EXTEND en cobol, fopen (fichero,"r+") en C. Estos modos solamente permiten añadir registros en cola de fichero, pero sin orden alguno, simplemente en orden de llegada. De la misma forma, estos modos no permiten la modificación de registros ya existentes.

En lo referente a la ocupación del soporte físico, este tipo de organización es la mejor, ya que no deja huecos pues los registros se graban unos a continuación de otros y nunca puede haber espacios intermedios entre los mismos.

Si tuviésemos que elegir un tipo de ficheros determinado para que sus registros estuviesen organizados secuencialmente, los únicos que parecen apropiados son los ficheros con gran frecuencia de utilización, es decir, ficheros en los que la mayoría de sus registros son consultados.

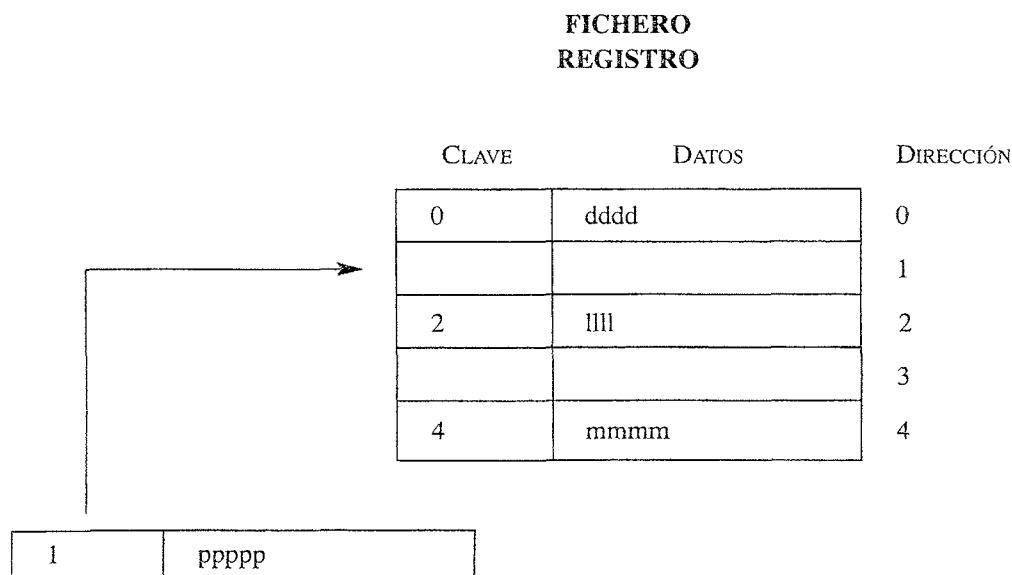
3.2. ORGANIZACIÓN DIRECTA.

En este tipo de organización, el concepto de identificativo o clave tiene sentido, ya que será éste el que indique la posición (DIRECCIÓN) física de almacenamiento. Esta dirección la obtendremos siempre del propio identificativo del registro, pero de dos posibles formas.

En primer lugar, supongamos que la dirección de almacenamiento del registro es indicada directamente por el valor del identificativo. Hay que tener en cuenta que en la mayoría de los casos, el valor del identificativo es numérico, ya que las direcciones en las que se ubican los registros son direcciones de tipo numérico. Si el identificativo fuese alfabético o alfanumérico, sería necesario aplicar un algoritmo de transformación.

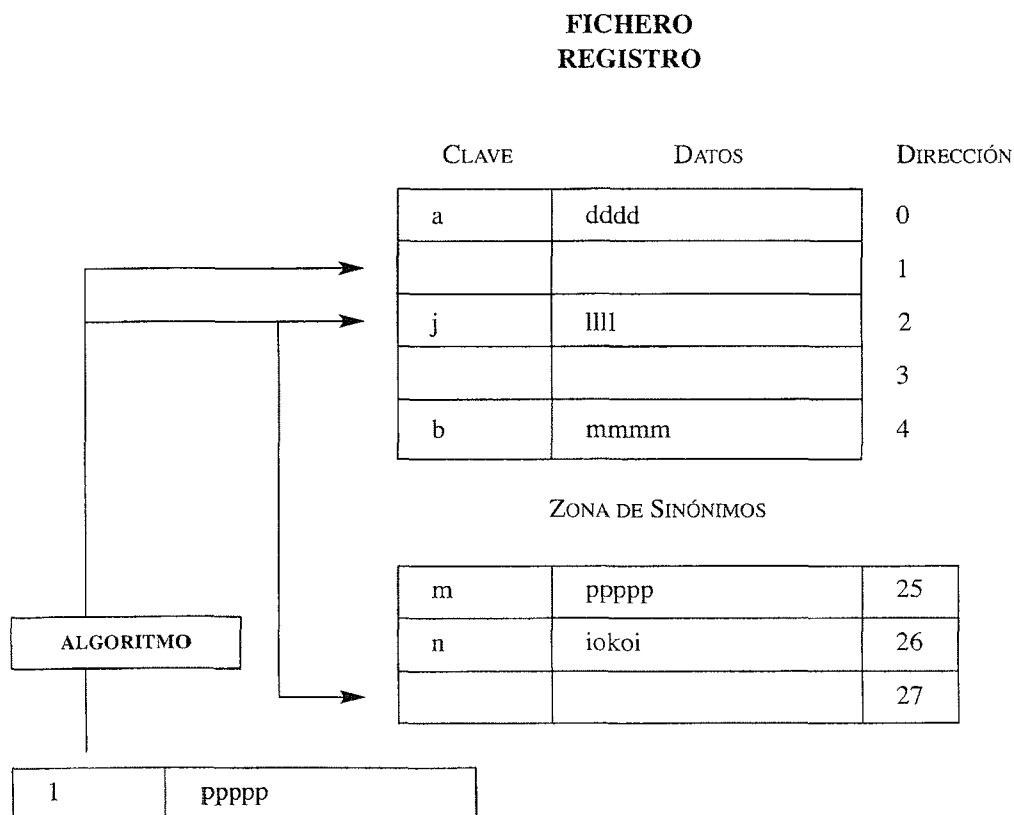
Si la dirección física de almacenamiento es obtenida directamente del contenido del identificativo, fácilmente podemos deducir que la secuencia lógica de almacenamiento de los registros, y la secuencia física coinciden, ya que por ejemplo un registro con valor de identificativo 1=0 se almacenará en la posición 0, uno con identificativo 1 = 4, en la posición 4, etc. Evidentemente, utilizando este tipo de direccionamiento, cada posición solamente puede ser ocupada por un registro. Es de suponer que si hay más de un registro con el mismo identificativo, estamos en un caso de error (registro repetido).

Por ejemplo: directo con direccionamiento directo.



En segundo lugar, supongamos que la dirección de almacenamiento se obtiene del identificativo, pero después de haber sufrido algún tipo de transformación. Este tipo de transformación se denomina algoritmo, y suele ser de tipo matemático. En este caso, la secuencia física y lógica de almacenamiento no coinciden.

Por ejemplo: directo con direccionamiento indirecto.



Cuando utilizamos el direccionamiento indirecto, puede ocurrir que registros con distinto identificativo obtengan la misma dirección de almacenamiento. Cuando esto es así, decimos que se ha producido un sinónimo o una colisión. Por supuesto, estos registros que obtienen la misma dirección de almacenamiento no tenemos por qué ignorarlos ya que son registros totalmente válidos.

Consideraremos que todas las direcciones para almacenar registros en ficheros con organización directa las obtendremos del identificativo, después de haberle sido aplicado un determinado algoritmo.

Cuando en ficheros con organización directa queremos acceder a un determinado registro del fichero, bastará con conocer el valor del identificativo, ya que con él obtendremos la dirección de almacenamiento, y por lo tanto el acceso al registro deseado será inmediato. Evidentemente, en estos registros anteriores al deseado, ya que directamente accederemos al solicitado sin más preámbulos.

Sabiendo que la localización de registros en ficheros con esta organización es inmediata, podemos deducir que las operaciones de inserción, eliminación y modificación de registros, se realizarán también de forma inmediata y sin presentar ningún problema adicional.

En general, un fichero con organización directa tiene las siguientes características:

- Acceso inmediato a los registros.
- No es necesario ordenar el fichero.
- No es necesario regenerar el fichero.

Es evidente que en un fichero con organización directa, podemos realizar operaciones de entrada/salida a la vez. Es por ello por lo que los ficheros con organización directa son los más rápidos cuando nos referimos a tratamientos de pocos registros. Más adelante veremos que los ficheros con organización secuencial indexada y secuencial indexada encadenada presentan algunas ventajas frente a los ficheros con organización directa.

Lo importante en ficheros con organización directa para realizar operaciones de entrada/salida es posicionarse sobre el registro a tratar. A continuación solamente hay que realizar la operación de entrada/salida deseada (inserción, eliminación o consulta).

Si quisiéramos consultar completamente el fichero o la mayoría de los registros contenidos en él, tendríamos un gran inconveniente. Lo que es evidente es que el usuario no puede saber en todo momento todos los identificativos de todos los registros que actualmente tiene almacenados en el fichero. Es por lo que para realizar este tipo de consultas tendremos que ir analizando posición a posición del fichero, desde la primera a la última. Puede ocurrir que algunas de las posiciones leídas estén vacías, implicando una considerable pérdida de tiempo. Por lo general, si queremos consultar casi completamente un fichero con organización directa, utilizaremos técnicas avanzadas de programación, que permiten leer secuencialmente el fichero desde el primer al último registro (Lectura siguiente = READ NEXT). De todas formas y aun así, este tipo de organización es la menos recomendada para realizar consultas completas del fichero.

Otro gran inconveniente que presenta este tipo de organización es la cantidad de huecos que deja dentro del fichero. Estos huecos surgen ya que los identificativos de los registros a almacenar, después de haber sufrido la aleatorización, indicarán posiciones de almacenamiento que lo más normal es que no sean contiguas. Esto implica un desaprovechamiento del soporte de almacenamiento respecto del número real de registros almacenados frente al número total de posiciones ocupadas.

Es importante considerar que el valor de la clave siempre estará en relación con la capacidad máxima del soporte físico utilizado para el almacenamiento. Evidentemente, nunca podremos almacenar registros cuya dirección de almacenamiento esté por encima de los límites máximos del fichero. Si ocurriese así, resultaría que el algoritmo utilizado para la transformación no sería el adecuado. Si no se utilizase algoritmo, y ocurriese esto, es decir, que sobrepasemos los límites del fichero con direccionamiento directo, necesariamente tendríamos que pensar en un nuevo algoritmo.

3.3. VARIANTES DE LA ORGANIZACIÓN SECUENCIAL.

Como resultado de estas limitaciones surgen nuevas organizaciones de ficheros, organizaciones en las que será necesario manejar dos nuevos conceptos: el puntero y el índice. Estos conceptos se utilizan exclusivamente en ficheros con organización secuencial y sirven para agilizar procesos de consulta y para indicar secuencias lógicas dentro de los propios ficheros. Puntero e índice podemos definirlos de la siguiente forma:

- **Puntero:** registro o célula de memoria que contiene la dirección de una información o conjunto de informaciones, dentro de un almacén de informaciones.
- **Índice:** tabla cuyos elementos contienen direcciones de datos a los que se puede acceder de forma secuencial. Contenido que sumado a un operando que indica una posición relativa resulta una dirección efectiva o absoluta.


3.3.1. Organización secuencial indexada.

Este tipo de organización aprovecha lo mejor de la organización secuencial (tratamiento de grandes volúmenes de información) y lo mejor de la organización directa (acceso rápido y directo a registros).

En este tipo de ficheros vamos a tener dos partes fundamentales: una, en la que se almacenen los datos como tal; y otra, destinada a almacenar las diferentes tablas de índices. Evidentemente, si vamos a utilizar una tabla de índices secuenciada y ordenada, los registros han de estar ordenados con arreglo a su identificativo. En este tipo de organización, el acceso a registros siempre se hará consultando previamente la tabla de índices, y posteriormente el fichero como tal.

La organización secuencial indexada es eficaz tanto en consultas esporádicas como en consultas completas del fichero.

TABLA DE ÍNDICES



68	0
138	3
205	6

FICHERO

CLAVES	DATOS	DIRECCIONES
12	A	0
25	B	1
68	C	2
89	D	3
123	E	4
138	F	5
198	G	6
204	H	7
205	I	8

Evidentemente, una vez creado el fichero, puede ser que queramos introducir nuevos registros. Esto plantea un problema, y es que siempre que creamos un fichero con este tipo de organización, reservamos un espacio para almacenar los registros que introdujimos al crear el fichero. Si posteriormente queremos insertar registros, tendremos que añadir al fichero una parte adicional, denominada zona de excedentes, de desbordamiento o de overflow. En esta zona almacenaremos todos los registros que se inserten, después de haber creado el fichero y haber llenado el espacio total destinado a almacenar los registros de la primera creación. Si recordamos cuando se hacían inserciones en ficheros con organización secuencial, los ficheros tenían que reorganizarse para que su secuencia lógica no se


rompiese. En la organización secuencial indexada, se crea una la zona de desbordamiento para almacenar los registros nuevos insertados. Obviamente la secuencia lógica de fichero en esta organización se romperá.

De cualquier manera, utilizar zona de excedentes depende directamente de las necesidades del usuario. No hay que olvidar que los registros en esta zona se almacenarán en orden de llegada, por lo que esta zona siempre que contenga registros, lo más probable es que no estén ordenados.

En el caso de eliminaciones, la eliminación del registro no es real, ya que el registro no es eliminado del fichero. Solamente es marcado para que en posteriores procesos sea ignorado. La tabla no sufre ninguna modificación, ya que las posiciones de entrada al fichero seguirán siendo las mismas. Si realizásemos una consulta del fichero, este registro será consultado, aunque será ignorado.

Si el fichero tuviese demasiadas eliminaciones, quedará bastante degradado teniendo que proceder a su reorganización. Esta reorganización, única y exclusivamente, consistirá en eliminar realmente los registros marcados en un proceso de baja, con el fin de eliminar espacios muertos dentro del fichero de datos.

TABLA DE ÍNDICES



68	0
138	3
205	6
overflow	
50	9
400	11

FICHERO

CLAVES	DATOS	DIRECCIONES
12	A	0
25	B	1
68	C	2
89	D	3
*	E	4
138	F	5
198	G	6
204	H	7
205	I	8
4	L	9
50	M	10
51	N	11
400	R	12

O
V
E
R
F
L
O
W

3.3.2. Organización secuencial encadenada.

Partimos de que, al igual que en los ficheros secuencial indexados, los ficheros secuenciales encadenados tienen la misma base que los secuenciales puros, pero sirviéndose de punteros con un fin muy concreto.

Ya sabemos que el puntero es un campo adicional que se incluye en el registro y que indica cuál es el siguiente o anterior registro en secuencia lógica. Los ficheros secuenciales puros que utilizan estos campos adicionales (punteros) se dice que tienen organización secuencial encadenada.

En este tipo de ficheros, las inserciones de los registros siempre se hacen al final. Esto implica que los registros, a medida que van llegando, se van situando en cola, pero sin ninguna secuencia lógica. Es por lo que empezamos a utilizar punteros cuya única función es la de encadenar unos registros con otros para que éstos tengan una secuencia lógica dentro del fichero.

Podemos, pues, decir que los registros se almacenan secuencialmente según el orden de llegada, por lo que la secuencia física y la secuencia lógica no coinciden; así, el último registro que llega es el que ocupa la primera posición vacía. Puede ocurrir que el último registro en secuencia física sea el primero en secuencia lógica. También hay que considerar que cada registro se almacenará con su puntero correspondiente.

Cuando realizamos eliminaciones de registros, el borrado se efectúa marcando el registro. Con esto, lo único que se consigue es que el registro se ignore en sucesivas consultas, ya que el puntero seguirá actuando como tal, pero sin que el contenido de ese registro sea válido. En los ficheros así organizados, las eliminaciones deterioran progresivamente el fichero, ya que el espacio de los registros borrados no es ocupado por otros registros nuevos.

Cuando queremos localizar registros, lo único que puede resultar más curioso es cómo localizar el primer registro en secuencia lógica. Como dijimos anteriormente, todos los ficheros con organización secuencial encadenada tienen al principio del mismo un campo adicional, llamado cabecera, con el único fin de señalar el primer registro en secuencia lógica dentro del fichero. Este campo se comporta igual que cualquier puntero, aunque solamente sirva como punto de partida de la secuencia lógica. Normalmente, es un campo situado al principio del fichero.

FICHERO REGISTROS

CLAVES	DATOS	PUNTERO	DIRECCIONES
		4	0
2	B	7	1
68	C	3	2
89	D	5	3
1	E	1	4
138	F	6	5
198	G	8	6
3	H	2	7
205	I	0	8

Obsérvese que el registro con clave 138 está dado de baja.

3.3.3. Organización secuencial indexada encadenada.

Como alternativa a estos dos tipos de organización (indexada y encadenada), surge la organización secuencial indexada encadenada aprovechando lo mejor de sus compañeras más directas. Destaca fundamentalmente por utilizar punteros e índices simultáneamente.

Claramente se deduce que si usamos punteros es para que la secuencia lógica no se pierda. Sabemos que si se utilizan índices, las inserciones van a parar a la zona de desbordamiento rompiendo la secuencia lógica. Es entonces cuando el puntero realiza la función de encadenar la zona real con la zona de desbordamiento. Este encadenamiento implica que la secuencia lógica no se rompa. En este tipo de ficheros, los índices tienen el mismo funcionamiento que en la secuencial indexada, es decir, permitir acceso casi directo (PSEUDODIRECTO) a registros situados en la zona real del fichero.

FICHERO

TABLA DE ÍNDICES

68	0
138	3
205	6
overflow	
50	9
400	11

CLAVES	DATOS	PUNTERO	DIRECCIONES
		9	0
25	B	10	1
68	C	3	2
89	D	5	3
*	E	5	4
138	F	6	5
198	G	7	6
204	H	8	7
205	I	12	8
4	L	21	9
50	M	11	10
51	N	2	11
400	R	0	12

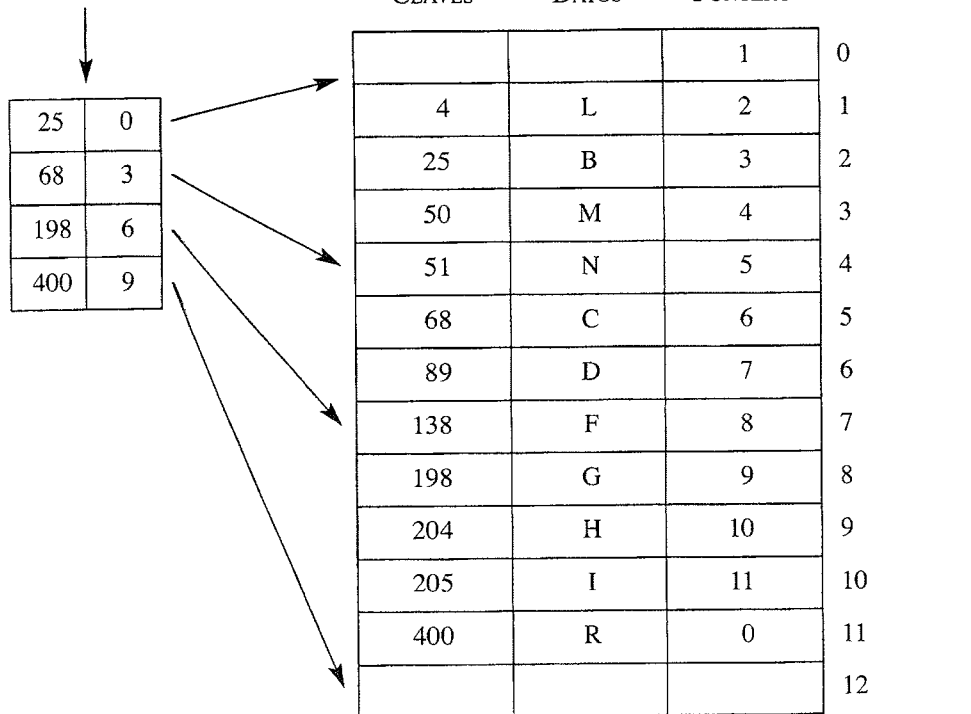
O
V
E
R
F
L
O
W

Reorganización de un fichero secuencial indexado encadenado.

Consiste en eliminar el área de desbordamiento, eliminar físicamente las bajas lógicas y dejar el fichero ordenado por el campo clave.

FICHERO

TABLA DE ÍNDICES



● ● ●

