

Escuela Politécnica Superior de Alcoy

---

**Ingeniería Técnica en Informática de  
Gestión**

**Estructuras de Datos y Algoritmos**

**Apuntes Teoría**

# Índice general

<b>1. Tipos Abstractos de Datos</b>	<b>5</b>
1.1. Introducción . . . . .	5
1.2. Ejemplo: . . . . .	6
<b>2. Gestión Dinámica de Memoria. Punteros</b>	<b>8</b>
2.1. Gestión dinámica de memoria. . . . .	8
2.1.1. Variables estáticas . . . . .	8
2.2. Punteros . . . . .	9
2.2.1. Punteros y vectores . . . . .	11
2.3. Variables dinámicas . . . . .	13
2.4. Ejercicios . . . . .	19
<b>3. Estructuras de Datos Lineales</b>	<b>20</b>
3.1. Introducción . . . . .	20
3.2. Pilas . . . . .	20
3.2.1. Operaciones sobre pilas . . . . .	21
3.2.2. Representación vectorial de pilas . . . . .	22
3.2.3. Representación enlazada de pilas con variable dinámica . . . . .	24
3.3. Colas . . . . .	26
3.3.1. Operaciones sobre colas . . . . .	27
3.3.2. Representación vectorial de colas . . . . .	28
3.3.3. Representación enlazada de colas con variable dinámica . . . . .	32
3.4. Listas . . . . .	34
3.4.1. Operaciones sobre listas . . . . .	34
3.4.2. Representación vectorial de listas . . . . .	35
3.4.3. Representación enlazada de listas con variable dinámica . . . . .	38
3.4.4. Representación enlazada de listas con variable estática . . . . .	42
3.5. Ejercicios . . . . .	46

<b>4. Divide y vencerás</b>	<b>62</b>
4.1. Esquema general de “Divide y Vencerás”	62
4.2. Algoritmos de ordenación	63
4.2.1. Inserción directa	63
4.2.2. Selección directa	67
4.2.3. Ordenación por mezcla o fusión: Mergesort	69
4.2.4. Algoritmo por partición: Quicksort	77
4.2.5. Comparación empírica de los algoritmos	88
4.3. Búsqueda del $k$ -ésimo menor elemento	89
4.3.1. Análisis de la eficiencia	90
4.4. Búsqueda binaria	93
4.4.1. Eficiencia del algoritmo	94
4.5. Cálculo de la potencia de un número	96
4.5.1. Algoritmo trivial	96
4.5.2. Algoritmo “divide y vencerás”	97
4.6. Otros problemas	98
4.7. Ejercicios	99
<b>5. Árboles</b>	<b>109</b>
5.1. Definiciones	109
5.2. Recorrido de árboles	111
5.3. Representación de árboles	112
5.3.1. Representación mediante listas de hijos	112
5.3.2. Representación “hijo más a la izquierda — hermano derecho”	114
5.4. Árboles binarios	117
5.4.1. Representación de árboles binarios	117
5.4.2. Recorrido de árboles binarios	120
5.4.3. Árbol binario completo. Representación.	123
5.4.4. Propiedades de los árboles binarios	124
5.5. Ejercicios	126
<b>6. Representación de Conjuntos</b>	<b>133</b>
6.1. Conceptos generales	133
6.1.1. Representación de conjuntos	133
6.1.2. Notación	133
6.1.3. Operaciones elementales sobre conjuntos	134
6.1.4. Conjuntos dinámicos	134
6.2. Tablas de dispersión o tablas <i>Hash</i>	136
6.2.1. Tablas de direccionamiento directo	136
6.2.2. Tablas de dispersión o tablas <i>Hash</i>	137

6.3.	Árboles binarios de búsqueda . . . . .	151
6.3.1.	Representación de árboles binarios de búsqueda . . . . .	152
6.3.2.	Altura máxima y mínima de un árbol binario de búsqueda . . . . .	154
6.3.3.	Recorrido de árboles binarios de búsqueda . . . . .	154
6.3.4.	Búsqueda de un elemento en un árbol binario de búsqueda . . . . .	155
6.3.5.	Búsqueda del elemento mínimo y del elemento máximo . . . . .	158
6.3.6.	Inserción de un elemento en un árbol binario de búsqueda . . . . .	160
6.3.7.	Borrado de un elemento en un árbol binario de búsqueda . . . . .	163
6.4.	Montículos ( <i>Heaps</i> ). Colas de prioridad. . . . .	173
6.4.1.	Manteniendo la propiedad de montículo . . . . .	174
6.4.2.	Construir un montículo . . . . .	179
6.4.3.	Algoritmo de ordenación <i>Heapsort</i> . . . . .	185
6.4.4.	Colas de prioridad . . . . .	191
6.5.	Estructura de datos para conjuntos disjuntos: MF-set . . . . .	196
6.5.1.	Representación de MF-sets . . . . .	197
6.5.2.	Operaciones sobre MF-sets . . . . .	197
6.6.	Otras Estructuras de Datos para Conjuntos . . . . .	202
6.6.1.	Tries . . . . .	202
6.6.2.	Árboles Balanceados . . . . .	204
6.7.	Ejercicios . . . . .	209
6.7.1.	Tablas de dispersión . . . . .	209
6.7.2.	Árboles binarios de búsqueda . . . . .	214
6.7.3.	Montículos (Heaps) . . . . .	219
6.7.4.	MF-sets . . . . .	224
<b>7.</b>	<b>Grafos</b> . . . . .	<b>226</b>
7.1.	Definiciones . . . . .	226
7.2.	Introducción a la teoría de grafos . . . . .	230
7.3.	Representación de grafos . . . . .	233
7.3.1.	Listas de adyacencia . . . . .	233
7.3.2.	Matriz de adyacencia . . . . .	235
7.4.	Recorrido de grafos . . . . .	237
7.4.1.	Recorrido primero en profundidad . . . . .	237
7.4.2.	Recorrido primero en anchura . . . . .	241
7.4.3.	Ordenación topológica . . . . .	242
7.5.	Caminos de mínimo peso: algoritmo de Dijkstra . . . . .	245
7.5.1.	Caminos de mínimo peso . . . . .	245
7.5.2.	Algoritmo de Dijkstra . . . . .	246
7.6.	Árbol de expansión de coste mínimo . . . . .	252
7.6.1.	Árbol de expansión . . . . .	252

7.6.2.	Algoritmo de Kruskal . . . . .	253
7.6.3.	Algoritmo de Prim . . . . .	258
7.7.	Ejercicios . . . . .	263
<b>8.</b>	<b>Algoritmos Voraces</b>	<b>271</b>
8.1.	Introducción . . . . .	271
8.1.1.	Ejemplo: Cajero automático . . . . .	272
8.2.	Esquema general Voraz . . . . .	272
8.3.	El problema de la compresión de ficheros . . . . .	274
8.4.	El problema de la mochila con fraccionamiento . . . . .	279
8.5.	Ejercicios . . . . .	283
.	<b>Exámenes de la asignatura</b>	<b>286</b>

# Tema 1

## Tipos Abstractos de Datos

### 1.1. Introducción

Entendemos como **Tipo de Datos** (TD) el conjunto de valores que una variable puede tomar. En otras palabras podríamos decir que un TD es la clase a la que pertenece una variable (p.e. un número que pertenezca a los enteros es de tipo *entero*).

Aunque a nivel lógico los TDs se representen de diferentes maneras (números enteros, caracteres, etc) a nivel interno el computador los tratará como agrupaciones de bits.

La necesidad de utilizar TDs viene fundamentada por dos razones:

- Que el compilador pueda elegir la representación interna óptima para una variable.
- Poder aprovechar las características de un TD, por ejemplo, las operaciones aritméticas de los números enteros.

Los lenguajes de programación suelen admitir cuatro tipos distintos de TD: enteros, reales, booleanos y caracteres. A estos TDs se les conoce como TD simples, elementales o primitivos. Cada uno de estos tipos tiene asociadas **operaciones** simples, como puede ser la multiplicación de enteros, etc.

Pero, ¿qué ocurriría si en uno de nuestros programas quisieramos trabajar con matrices de números? Como no son un TD simple, esto nos induce a pensar que necesitaríamos definir un TD matriz y también cómo se llevan a cabo las operaciones asociadas a ese tipo, como la multiplicación de matrices, por ejemplo.

Con esto llegamos al concepto de **Tipo Abstracto de Datos** (TAD o TDA<sup>1</sup>). Un TAD es un modelo matemático con una serie de operaciones definidas en ese modelo (se suelen conocer como los **procedimientos** u operaciones de ese TAD).

---

<sup>1</sup>de Tipo de Datos Abstracto, como se encuentra a veces en la literatura.

Un TAD es una generalización de los TD simples y los procedimientos son generalizaciones de operaciones primitivas.

Uno de los aspectos más importantes de los TADs es su capacidad de **encapsulación** o **abstracción** que nos permite localizar la definición del TAD y sus operaciones asociadas en un lugar determinado del programa. Pudiendo definir librerías específicas para un determinado TAD. Esta característica permite cambiar la implementación según la adecuación al problema, así como más facilidades en la depuración de código y mayor orden y estructuración en los programas. Por ejemplo, podríamos definir un TAD matriz, pero posteriormente elegir entre una implementación tradicional de matrices o utilizar estructuras especiales para matrices dispersas, en el caso de que fueran más útiles para nuestro problema.

Realizar la **implementación** de un TAD significa traducir en instrucciones de un lenguaje de programación la declaración que define a una variable como perteneciente a ese tipo, además de un procedimiento en ese lenguaje por cada operación definida para el TAD.

Una implementación de un TAD elige una **estructura de datos** (ED) para representar el TAD; cada ED se construye a partir de TDs simples del lenguaje, usando los dispositivos de estructuración disponibles en el lenguaje de programación utilizado como pueden ser los vectores o registros.

## 1.2. Ejemplo:

Vemos como podría definirse un TAD Matriz de Enteros:

- En primer lugar estaría la definición del tipo:  
*MATRIZ DE ENTEROS*  $\equiv$  es una agrupación de números enteros ordenados por filas y por columnas. El número de elementos en cada fila es el mismo. También el número de elementos en cada columna es el mismo. Un número entero que forme parte como elemento de la matriz se identifica por su número de fila y número de columna.

Representación lógica:

$$M = \begin{pmatrix} 3 & 7 & 9 \\ 4 & 5 & 6 \\ 2 & 3 & 5 \end{pmatrix}$$

y tenemos que  $M[2, 1] = 4$ .

- La operaciones definidas para el TAD matriz son:
  - $\text{Suma\_Mat}(A, B: \text{MATRIZ})$  devuelve  $C: \text{MATRIZ}$   
 $\equiv$  suma dos matrices que tienen igual número de filas y de columnas.

La suma se realiza sumando elemento a elemento cada número entero de la matriz A con el correspondiente número entero de la matriz B que tenga igual número de fila e igual número de columna.

- `Multiplicar_Mat(A,B: MATRIZ)` devuelve `C: MATRIZ`  
≡ multiplica dos matrices que cumplen la condición de que el número de columnas de la matriz A coincide con el número de filas de la matriz B. La multiplicación se lleva a cabo ....
- `Inversa_Mat(A: MATRIZ)` devuelve `C: MATRIZ`  
≡ en el caso de que la matriz A posea inversa, ésta se calcula ....

Con esta definición el TAD Matriz de Enteros se podría implementar de esta manera en lenguaje C:

```
#define NUM_FILAS 10
#define NUM_COLUMNAS 10

/* Definimos el TAD */
typedef int t_matriz[NUM_FILAS][NUM_COLUMNAS];

/* Definimos una variable */
t_matriz M;
```

y la operación `Suma_Mat()` sería:

```
void Suma_Mat(t_matriz A, t_matriz B, t_matriz C) {
    int i,j;

    for (i=0;i<NUM_FILAS;i++)
        for (j=0;j<NUM_COLUMNAS;j++)
            C[i][j] = A[i][j] + B[i][j];
}
```



## Tema 2

# Gestión Dinámica de Memoria. Punteros

### 2.1. Gestión dinámica de memoria.

Para poder comenzar este tema, se espera que el alumno conozca los distintos tipos básicos de variables que permiten definir la mayoría de los lenguajes de programación.

Durante este tema veremos la diferencia entre memoria estática y memoria dinámica, además discutiremos la necesidad de utilizar ésta última y como gestionarla. Veremos también varios ejemplos para explicar cómo se define memoria dinámica y como se maneja con el lenguaje de programación C.

#### 2.1.1. Variables estáticas

Supongamos que tenemos un programa en el cual hemos definido varias variables. Cuando ejecutamos el programa, éste se carga en memoria y reserva espacio para las variables que tiene definidas. En ese espacio se guardarán los valores que se asignen a las variables, además ese espacio de memoria que ha reservado es el mismo desde que se lanza a ejecución el programa hasta que se termina.

Cuando en la codificación de un programa declaramos una variable, definimos a su vez de qué tipo es. El proceso de compilación determinará después el tamaño de memoria que va a ocupar esa variable en el programa ejecutable.

Por ejemplo, si tenemos la definición:

```
int x;
```

Cuando comience la ejecución se reservará un espacio de memoria que podrá almacenar un entero y al cual se podrá acceder con la etiqueta x.

x

A la memoria consumida de esta manera, esto es, para variables declaradas en tiempo de compilación, la llamaremos memoria estática. A las variables que consuman memoria estática las llamaremos variables estáticas.

Así pues, usando la definición del ejemplo anterior, son válidas las siguientes operaciones:

En el código	En ejecución
<code>x=5 ;</code>	x <span style="border: 1px solid black; display: inline-block; width: 15px; height: 15px; text-align: center; vertical-align: middle;">5</span>
<code>x=x+2 ;</code>	x <span style="border: 1px solid black; display: inline-block; width: 15px; height: 15px; text-align: center; vertical-align: middle;">7</span>

Para variables que constituyan estructuras más complejas, como los registros o vectores, la situación es la misma:

En el código	En ejecución	Instrucción válida
<code><i>int</i> v[3];</code>	v <span style="border: 1px solid black; display: inline-block; width: 30px; height: 15px; vertical-align: middle;"></span>	<code>v[0]=1 ;</code>
<code>struct {<i>float</i> r,i;} im;</code>	im <span style="border: 1px solid black; display: inline-block; width: 15px; height: 20px; vertical-align: middle;"></span>	<code>im.r=5.0 ;</code> <code>im.i=3.0 ;</code>

## 2.2. Punteros

Un mecanismo que muchos lenguajes de programación ofrecen es poder acceder al espacio de memoria ocupado por una variable por medio de su dirección de memoria y no por el nombre de la variable (su identificador o etiqueta), así, por ejemplo, también podríamos modificar variables en memoria sin hacer referencia a la variable en sí.

Este mecanismo suele implementarse en los lenguajes de programación por medio de lo que se conoce como punteros. Vamos a comprender el concepto de puntero mediante un ejemplo.

Si tenemos la definición:

`int *px ;`

Estamos definiendo la variable `px` como un puntero a entero y **no** como entero. `px` puede contener una dirección a otra zona de memoria que sí que puede contener un entero. Básicamente, `px` es una variable que puede guardar la dirección de memoria de otra variable.

Para definir un puntero en lenguaje C, debe utilizarse la siguiente sintaxis:

```
tipo_datos *nombre_variable_puntero;
```

Donde el operador `*` indica que la variable `nombre_variable_puntero` es un puntero a otra variable de tipo `tipo_datos`.

Un puntero puede tomar un valor especial `NULL` para indicar que no apunta a ningún lugar accesible.

Así pues, `px` es una variable, pero no podrá tener valores como la `x`, es decir, no podrá contener un 2 o un 5, sino que contendrá sólo direcciones de memoria. Normalmente, a nivel gráfico representaremos una dirección de memoria con una flecha que apunta a una zona de memoria determinada o con un símbolo que represente que el puntero tiene un valor `NULL`.

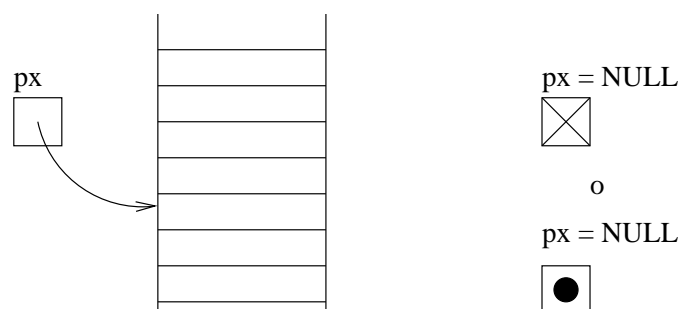














Figura 2.1: Representación gráfica de punteros.

Si queremos acceder al contenido de la posición de memoria a la que señala `px`, tenemos que hacerlo utilizando la expresión `*px`, además, hay que tener en cuenta que se accede a una variable del tipo para la cual se definió el puntero, en este caso un entero.

Además del mecanismo `*px` para acceder a la zona de memoria donde apunta `px`, en lenguaje C se dispone del operador `&` que permite recuperar la dirección de una variable que ya existe en memoria. Este nuevo operador se utilizará para poder obtener la dirección de memoria de una variable y asignarla a un puntero.

Vemos a continuación algunos ejemplos de como definir y trabajar con punteros en lenguaje C:

En el código	En ejecución	Comentario
<code>int *px, x=1;</code>	px  x 	px está indefinido; x contiene un 1.
<code>px=&amp;x;</code>	px   x	En px guardamos la dirección de x.
<code>*px=2;</code>	px   x	Modificamos x a través de px.
<code>px=1;</code>	px   x	ERROR: no se puede asignar un entero a px.
<code>px=NULL;</code>	px   x	px toma valor NULL.
<code>*px=1;</code>	px   x	ERROR: px no apunta a un lugar accesible.


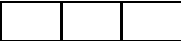

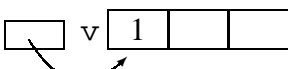
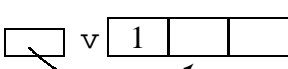
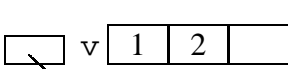
### 2.2.1. Punteros y vectores

¿Y si hacemos que un puntero apunte a una variable de tipo vector en vez de a una variable simple como un entero? Entonces obtenemos una característica interesante...

Una característica importante de los punteros en C es que pueden apuntar a variables con una estructura más compleja, como un vector o un registro (`struct`).

Además un puntero permite apuntar dentro de esas estructuras a cualquier posición cuyo tipo de datos se corresponda con el tipo al que apunta el puntero.

De esa manera si definimos una variable puntero a entero, `px`, y un vector de enteros, `v[3]`, son posibles operaciones como las siguientes:

En el código	En ejecución	Comentario
<code>int *px, v[3];</code>	px  v 	px y los elementos de v están indefinidos.
<code>px=&amp;v[0];</code>	px 	px apunta a la primera posición del vector.
<code>*px=1;</code>	px 	La primera posición del vector se modifica a través de px.
<code>px=px+1;</code>	px 	px apunta a la segunda posición del vector.
<code>v[1]=v[0]+*(px-1);</code>	px 	?

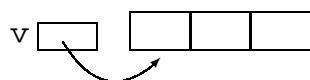
El resultado de la última instrucción `v[1]=v[0]+*(px-1);` es asignar un 2 a la segunda posición del vector `v` debido a que asignamos a `v[1]` el valor resultante de la expresión `v[0]+*(px-1)`. Interpretando esta última expresión, tendremos que `v[0]` tiene un valor 1, mientras que `*(px-1)` se refiere al contenido de la posición de memoria señalada por `(px-1)`, que al ser `px` un puntero a un entero indica la posición del vector de enteros anterior a la posición a la que apunta `px`, esto es, el contenido de la posición 0 del vector; así pues, `v[1]=1+1;`.

Profundizamos ahora en la relación entre punteros y vectores en lenguaje C que se va a traducir en poder manejar las mismas estructuras de memoria mediante vectores o punteros obteniendo los mismos resultados y utilizando un mecanismo u otro según convenga para la resolución del problema.


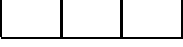

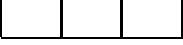


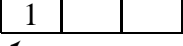

Si definimos un vector:

```
int v[3];
```

en lenguaje C podemos *interpretar* esa definición de la siguiente manera:



La variable `v` se puede ver como un puntero a una o más posiciones de memoria consecutivas. Aunque de aquí en adelante podamos pensar en los vectores de esta manera, seguiremos representando gráficamente los vectores como antes, y los punteros con variables que apuntan a otras variables. De esa manera son posibles operaciones como las siguientes:

En el código	En ejecución	Comentario
<code>int *px, v[3]; px</code>  <code>v</code> 		px y los elementos de v están indefinidos. Por comodidad representamos v como antes.
<code>px=v;</code>	<code>px</code>  <code>v</code>  	px apunta a la primera posición del vector.
<code>px[0]=1;</code>	<code>px</code>  <code>v</code>  	La primera posición de v se modifica a través de px.

La forma de acceso al vector utilizando la expresión `px[0] = 1;` es equivalente a usar la expresión `*px = 1;`.

## 2.3. Variables dinámicas

Ahora bien, si en un programa tuviéramos que crear temporalmente una gran cantidad de información para realizar ciertas operaciones con ella y no volverla a usar durante el resto del programa, ¿dónde almacenaríamos esa información? ¿qué pasaría si ni tan siquiera conociéramos la cantidad de memoria que se necesita para guardar esa información? ¿cómo almacenaríamos esa información?

Parece interesante pensar en un mecanismo de uso de memoria diferente al de declarar variables estáticas, ya que estas variables deberían ser lo suficientemente grandes para poder almacenar toda la información que podamos necesitar y además ese espacio de memoria reservada tan grande sólo se usaría cuando se opere con esos datos, pero durante el resto de la ejecución del programa estaría consumiendo memoria inútilmente.

Los lenguajes de programación han de tener una facilidad para reservar espacio en memoria y liberarlo, por ejemplo, para poder almacenar grandes cantidades de información temporales.

A la memoria que se reserva y libera durante la ejecución de un programa la llamamos memoria dinámica. A las variables que utilizaremos para acceder a esas zonas de memoria dinámica las llamaremos variables dinámicas. Durante la codificación y compilación se define el tipo de una variable dinámica. Durante la ejecución del programa se reserva la cantidad de memoria necesaria.

Vemos ahora cómo crear y gestionar memoria dinámica, es decir, reservar espacio en memoria durante la ejecución de un programa y utilizar ese espacio.

Para ello, será necesario usar las siguientes funciones del lenguaje C:

- Para reserva de memoria: las funciones `malloc` y `calloc` permiten reservar memoria dinámicamente. La función `malloc` no inicializa la memoria reservada, mientras que `calloc` la inicializa a 0 (introduce el valor 0 en los bytes de memoria reservada).
- Para liberación de memoria: la función `free` permite liberar memoria previamente reservada.
- Para calcular la talla de un tipo de datos: `sizeof` devuelve el número de bytes que necesita un tipo.

La sintaxis de `malloc` y `calloc` es:

```
void *malloc(size_t size);  
void *calloc(size_t nmemb, size_t size);
```

La función `malloc` recibe como parámetro el número de bytes de memoria que queremos reservar y devuelve un puntero sin tipo que apunta al principio de la zona de memoria reservada..

La función `calloc` recibe como parámetros el número de elementos que queremos reservar y el número de bytes de cada elemento y devuelve un puntero sin tipo que apunta al principio de la zona de memoria reservada.

La sintaxis para la función de liberación de memoria reservada, `free`, es:

```
void free(void *ptr);
```

Esta función recibe como parámetro un puntero al principio de una zona válida de memoria que habrá sido reservada previamente, y libera dicha memoria para que pueda ser utilizada por otras reservas de memoria que se quieran realizar, etc.

La sintaxis de la función `sizeof` es:

```
sizeof(type);
```

La función recibe como parámetro un tipo de datos, tanto de los tipos originales del lenguaje C (`int`, `float`, etc.), como de los tipos de datos definidos por el usuario.

A continuación vemos varios ejemplos de la definición y uso de memoria dinámica utilizando las funciones descritas.

Si suponemos una definición como la siguiente:

```
int *v;
```

Podríamos realizar las siguientes instrucciones:

En el código	Comentario
<code>v=(<i>int</i> *)malloc(n*sizeof(<i>int</i>));</code>	Reservamos un vector de $n$ enteros. Forzamos a la función malloc a devolver un puntero a entero que se guarda en v.
<code>v[0]=1;</code>	Indexamos el vector de la manera habitual.
<code>free(v);</code>	Liberamos la memoria previamente reservada. La información almacenada es a partir de ahora inaccesible.
<code>v[0]=1;</code>	ERROR: v ya no apunta a una zona válida.

Como se puede observar en el ejemplo anterior, en la llamada a la función malloc para reservar memoria para estructuras vectoriales, se utilizará como parámetro normalmente el resultado de una multiplicación donde los términos serán por un lado el número de elementos del vector y por otro el tamaño en bytes de cada uno de los elementos del vector. Sin embargo, **no** se debe confundir el operador \* de multiplicación con el operador \* de indirección.

Si sabemos que `int *x;` permite definir tanto un puntero a entero como un vector de enteros, entonces, ¿qué permitirá definir `int **x;`? Esta expresión se usará para definir un vector de punteros a enteros, con lo que también podrá verse como un vector de vectores de enteros, o lo que es lo mismo, una matriz de enteros.

Si tenemos las siguientes definiciones:

```
int **v,*pv,i,j; /* v es un puntero a uno o mas */
                /* punteros a enteros. */
```

El siguiente segmento de código permitiría definir una matriz cuadrada de enteros:

```
/* Definimos un vector de punteros. */
v=(int **)malloc(n*sizeof(int *));
for (i=0; i<n; i++) {
    /* Definimos un vector de enteros. */
    v[i]=(int *)malloc(n*sizeof(int));
    for (j=0; j<n; j++) /* Inicializamos cada fila. */
        v[i][j]=0;
}
```

El siguiente segmento de código tendría el mismo efecto que el anterior, es decir, definiría una matriz cuadrada de enteros:



```

/* Definimos un vector de punteros. */
v=(int **)malloc(n*sizeof(int *));
/* Definimos un vector de enteros. */
pv=(int *)malloc(n*n*sizeof(int));
for (i=0; i<n*n; i++)
    pv[i]=0;
for (i=0; i<n; i++)
    v[i]=&pv[i*n];

```

¿Qué haríamos para definir una matriz no cuadrada, de tamaño  $n \times m$ ? La solución a este problema sería cambiar una de las  $n$ 's de las reservas de memoria y bucles vistos arriba.

Vemos más ejemplos de manejo de punteros y memoria dinámica. Sabiendo que para definir nuevos tipos de datos en lenguaje C utilizamos la construcción

```
typedef definicion nombre_tipo;
```

Podemos considerar las siguientes definiciones:

```
typedef struct {
    int a, b;
} tupla;
```

```

tupla *t, **tt; /* tt es un vector de apuntadores a tupla. */
int *p,i;

```

Hay que recordar que en lenguaje C cuando se accede a un elemento de una struct mediante un puntero se tiene que utilizar el operador  $\rightarrow$  en vez del  $.$  para acceder a los diferentes campos del registro.

Entonces podemos ejecutar las siguientes instrucciones:

En el código	Comentario
<code>t=(tupla *)malloc(sizeof(tupla));</code>	Reservamos un registro. Forzamos a la función malloc a devolver un puntero al tipo deseado.
<code>t-&gt;a=0;</code>	Modificamos un campo del registro.
<code>p=&amp;t-&gt;b;</code>	Asignamos a p la dirección de un campo del registro, que es una zona válida de memoria.
<code>*p=0;</code>	Modificamos un campo del registro.
<code>free(t);</code>	Liberamos la memoria previamente reservada.
<code>*p=0;</code>	ERROR: p ya no apunta a una zona válida.

Si consideramos las mismas definiciones de tipos y variables que para el ejemplo anterior, vamos a construir e inicializar una estructura de datos un poco más compleja con las siguientes instrucciones:

En el código	Comentario
<code>tt=(tupla**)malloc(n*sizeof(tupla*));</code>	Reservamos un vector de punteros a tupla.
<code>t=(tupla*)malloc(n*sizeof(tupla));</code>	Reservamos un vector de tuplas.
<code><b>for</b> (i=0;i&lt;n;i++) {</code>	Después enlazamos cada elemento de tt con un elemento de t.
<code>    tt[i]=&amp;t[i];</code>	
<code>    tt[i]-&gt;a=0;</code>	
<code>    tt[i]-&gt;b=0;</code>	
<code>}</code>	

La estructura que se crea e inicializa con las anteriores instrucciones se muestra en la figura [2.2](#).

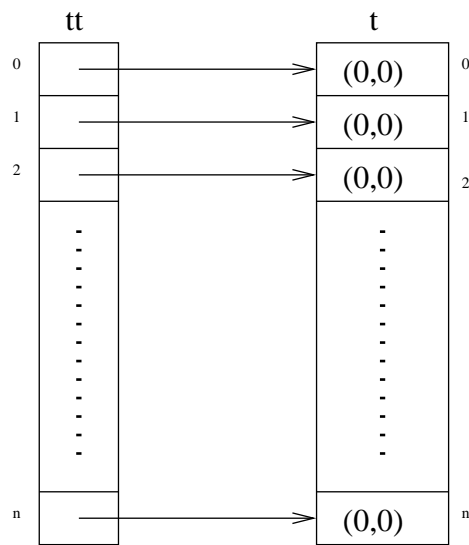


Figura 2.2: Representación gráfica de la estructura definida en el ejemplo.

### Paso de vectores a funciones

Si deseamos pasar como argumento un vector a una función o procedimiento en lenguaje C, entonces tendremos que pasarlo mediante un puntero a un vector, debido a que realizar una copia local del vector cada vez que se llame a la función resulta demasiado costoso. A nivel esquemático, podemos ver en el siguiente pedazo de código cómo se pasaría un vector a un procedimiento:

```
void f(int *v) { ... }
...
int main() {
    int p[n];
    ...
    f(p);
    ...
}
```

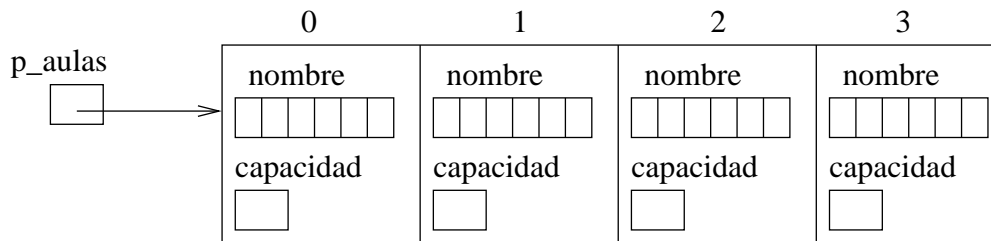
Esta obligación de pasar los vectores a las funciones o procedimientos mediante un puntero implicará que el vector se pasa por referencia, esto es, que cualquier modificación que se haga en el vector quedará reflejada en él aún después de acabar la función. Debe prestarse atención a las manipulaciones de vectores en las funciones. Si no queremos que las modificaciones dejen alterado el vector original una vez haya terminado la función, deberemos crear una copia local del vector en la función y trabajar con la copia mientras ejecutemos la función.

## 2.4. Ejercicios

### Ejercicio 1:

-Construye un pequeño programa en lenguaje C de manera que definas un tipo de datos aula que sea un registro que contenga una cadena de 6 caracteres llamada nombre y un entero llamado capacidad. En el programa principal define un puntero p\_aulas a este tipo de datos, después construye un vector de 4 elementos aula con la reserva de memoria adecuada y haz que se pueda acceder a este vector mediante el puntero p\_aulas.

La situación de memoria que debes obtener es la siguiente:



Ahora asigna al aula que está en la posición 2 del vector de aulas una capacidad de 100 y el nombre "F2A2".

### Solución:

En el ejercicio se solicita el siguiente programa en C:

```
#include <stdio.h>

typedef struct {
    char nombre[6]; /* nombre del aula */
    int capacidad; /* capacidad del aula */
} aula;

void main() {
    aula *p_aulas; /* Puntero a aula, sera el vector de aulas */

    /* Reservamos memoria del vector de aulas */
    p_aulas = (aula *) malloc(4 * sizeof(aula));
    /* Realizamos modificaciones */
    p_aulas[2].capacidad = 100;
    strcpy(p_aulas[2].nombre, "F2A2");
}
```

## Tema 3

# Estructuras de Datos Lineales

### 3.1. Introducción

En este tema introduciremos nuevos tipos abstractos de datos conocidos como pilas, colas y listas. A estos tads se les suele denominar *lineales* porque se utilizan para representar algún tipo de *ordenación espacial lineal* entre los datos que almacenan y además se construyen mediante estructuras que mantienen un *esquema lineal* de los datos.

Para cada tad estudiaremos su definición y cuáles son sus operaciones más comunes, después veremos diferentes estructuras de datos que se pueden aplicar a cada tad según la representación que se desee utilizar, y para cada representación veremos la implementación de las operaciones en lenguaje C.

### 3.2. Pilas

Definimos una **pila** como una estructura de datos para almacenar objetos que se caracteriza por la manera de acceder a los datos: *el último que entra es el primero en salir* (sigue una estructura LIFO<sup>1</sup>). Con esto queremos expresar que el último elemento que se haya insertado en la pila (el más recientemente insertado) será el primero que se extraiga de la pila, en el caso en que queramos extraer algún elemento de ésta.

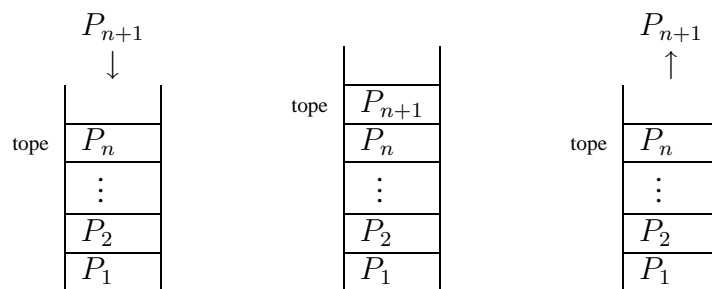
Junto con la pila que almacena los datos, existe un *marcador* llamado **tope** que indica cuál es el último elemento insertado y por tanto cuál es el elemento que se puede extraer ahora. Se accede a los elementos únicamente por el *tope* de la pila, es decir, cuando queramos extraer un elemento de una pila, extraeremos el elemento que esté en el tope de la pila (si no está vacía), y si queremos insertar un

---

<sup>1</sup>Del inglés, *Last In First Out*.

elemento en la pila, lo pondremos encima del elemento que estuviera en el tope y cambiaremos el tope de manera que señale al elemento recién insertado.

Una representación gráfica de la estructura de datos pila es la siguiente, donde se observa cómo se inserta y se extrae un elemento mientras `tope` siempre señala la cima de la pila:



Posibles aplicaciones de la estructura de datos pila son:

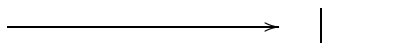
- La evaluación de expresiones aritméticas  $\equiv$  para calcular el resultado de una expresión aritmética es necesario ir calculando valores de las subexpresiones conforme a la prioridad de operadores, los elementos de estas subexpresiones se apilan hasta que se han leído todos los operandos, entonces se calcula el valor de la subexpresión y se apila. Se continúa calculando los valores de las subexpresiones hasta que se calcule el de la expresión aritmética global.
- Gestión de la recursión  $\equiv$  la *pila de recursión* debe almacenar el *estado de una función* (variables locales, etc.) antes de realizar una nueva llamada recursiva. El mantener el estado almacenado permite que después de la llamada recursiva se continúe la ejecución con los mismos valores de estado con los que se estaba ejecutando.

### 3.2.1. Operaciones sobre pilas

Para cada operación que definamos, veremos una representación gráfica del funcionamiento de la función. En algunas operaciones se verán el estado de la pila antes de la ejecución y el estado después.

Las operaciones más comunes que se realizan sobre una estructura de datos pila son las siguientes:

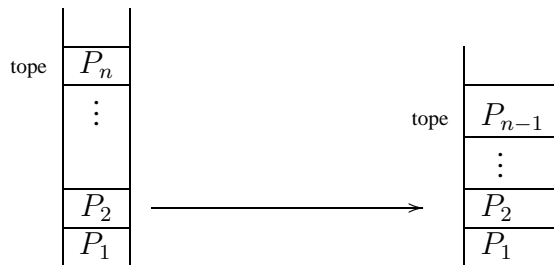
- `crearp()`: crea una pila vacía.



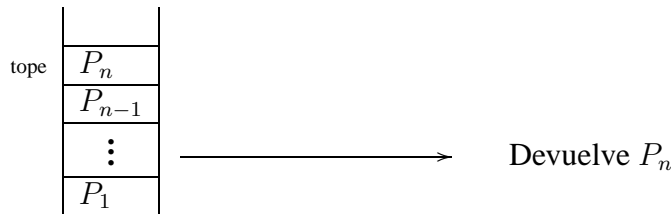
- $\text{apilar}(p, e)$ : añade el elemento  $e$  a la pila  $p$ . (*push* en inglés)



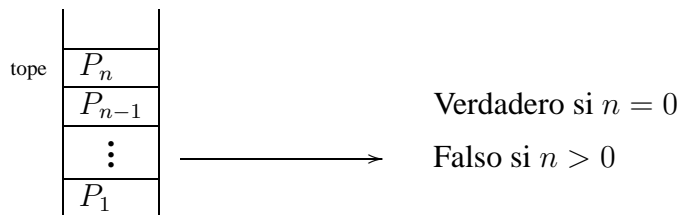
- $\text{desapilar}(p)$ : elimina el elemento del tope de la pila  $p$ . (*pop* en inglés)



- $\text{tope}(p)$ : consulta el tope de la pila  $p$ . (*top* en inglés)

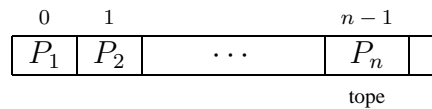


- $\text{vacio}(p)$ : consulta si la pila  $p$  está vacía.



### 3.2.2. Representación vectorial de pilas

Una posible representación para pilas sería almacenar los elementos en posiciones consecutivas de un vector a partir de la primera posición del vector, y utilizar una variable entera,  $\text{tope}$ , que indique dónde se encuentra el tope de la pila. De manera gráfica:



Un inconveniente de esta representación es que el número de elementos que puede almacenar la pila queda determinado por el tamaño físico del vector, con lo que el código que implemente las operaciones de la pila para esta representación deberá realizar las comprobaciones necesarias para no acceder a una posición fuera del vector y añadir elementos de más.

La definición del tipo de datos en lenguaje C sería:

```
#define maxP ... /* Talla maxima de vector. */

typedef struct {
    int v[maxP]; /* Vector definido en tiempo de compilacion. */
                /* Trabajamos con pilas de enteros. */
    int tope;    /* Puntero al tope de la pila. */
} pila;
```

La implementación de las operaciones definidas anteriormente se muestra a continuación:

```
pila *crearp() {
    pila *p;

    /* Reservamos memoria para la pila. */
    p = (pila *) malloc(sizeof(pila));
    p->tope = -1; /* Inicializamos el puntero al tope. */
    return(p);   /* Devolvemos un puntero a la pila creada. */
}

pila *apilar(pila *p, int e) {
    if (p->tope + 1 == maxP) /* Comprobamos si cabe el elemento. */
        /* Si no cabe hacemos un tratamiento de error. */
        tratarPilaLlena();
    else { /* Si cabe, entonces */
        p->tope = p->tope + 1; /* actualizamos el tope e */
        p->v[p->tope] = e;    /* insertamos el elemento. */
    }
    return(p); /* Devolvemos un puntero a la pila modificada. */
}
```



```

pila *desapilar(pila *p) {
    p->tope = p->tope - 1;  /* Decrementamos el puntero al tope. */
    return(p);             /* Devolvemos un puntero a la pila modificada. */
}

int tope(pila *p) {
    return(p->v[p->tope]);
    /* Devolvemos el elemento apuntado por tope. */
}

int vaciap(pila *p) {
    return(p->tope < 0);
    /* Devolvemos 0 (falso) si la pila no esta vacia, */
    /* y 1 (cierto) en caso contrario. */
}

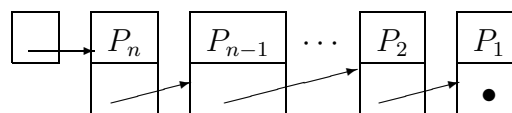
```

### 3.2.3. Representación enlazada de pilas con variable dinámica

Otra representación posible para pilas sería construyendo dinámicamente un registro para cada elemento que quiera insertarse en la pila y mantener estos registros enlazados mediante punteros desde un registro a otro, de manera que los punteros indicarán qué elemento está almacenado detrás de qué elemento, y por lo tanto podremos simular el orden de almacenamiento de los elementos en la pila.

En cierto modo lo que estamos haciendo es simular un vector, excepto que en este caso, las posiciones de los elementos se corresponden con punteros, y en el registro correspondiente a cada posición se guarda también la dirección de memoria donde se guarda el siguiente elemento.

Además se necesitará una variable de tipo puntero que apuntará al registro que sea el *primer* registro de la pila (el tope); y a partir de ese elemento estarán enlazados todos los demás conforme al orden que deban seguir en la pila. Una representación gráfica de esta estructura de datos para pilas sería:



Cada uno de los registros que forman la pila enlazada se denomina nodo.

Con este tipo de estructura, cuando se vaya a insertar un nuevo elemento en la pila, se creará un nuevo nodo (con una función de reserva de memoria) para el nuevo elemento y se enlazará con los demás nodos de la pila ocupando él la

primera *posición*. Cuando se quiera extraer el elemento del tope de la pila, bastará con liberar la memoria que ocupa el registro que forme el tope de la pila y actualizar el puntero que indica dónde está el tope.

Usando esta representación, el número máximo de elementos que puede almacenar una pila queda limitado por el número de nodos que se puedan crear, es decir, el número de registros que se puedan reservar sin agotar la memoria del computador. Así pues, esta representación de pilas no tiene el problema de tener limitado el número de elementos a almacenar, como ocurría con la representación vectorial de listas.

La definición del tipo de datos en lenguaje C sería:

```
typedef struct _pnodo {  
    int e;      /* Variable para almacenar un elemento de la pila. */  
    struct _pnodo *sig;  
                /* Puntero al siguiente nodo que contiene un elemento. */  
} pnodo;  
                /* Tipo nodo. Cada nodo contiene un elemento de la pila. */  
  
typedef pnodo pila;
```

La implementación de las operaciones definidas anteriormente se muestra a continuación:

```
pila *crearp() {  
    return(NULL); /* Devolvemos un valor NULL para inicializar */  
}                /* el puntero de acceso a la pila. */  
  
int tope(pila *p) {  
    return(p->e); /* Devolvemos el elemento apuntado por p */  
}  
  
pila *apilar(pila *p, int e) {  
    pnodo *paux;  
  
    paux = (pnodo *) malloc(sizeof(pnodo)); /* Creamos un nodo. */  
    paux->e = e;                            /* Almacenamos el elemento e. */  
    paux->sig = p; /* El nuevo nodo pasa a ser tope de la pila. */  
    return(paux); /* Devolvemos un puntero al nuevo tope. */  
}
```

```

pila *desapilar(pila *p) {
    pnode *paux;

    paux = p;          /* Guardamos un puntero al nodo a borrar. */
    p = p->sig;
    /* El nuevo tope sera el nodo apuntado por el tope actual. */
    free(paux);
    /* Liberamos la memoria ocupada por el tope actual. */
    return(p);          /* Devolvemos un puntero al nuevo tope. */
}

int vaciap(pila *p) {
    return(p == NULL);
    /* Devolvemos 0 (falso) si la pila no esta vacia, */
    /* y 1 (cierto) en caso contrario. */
}

```

### 3.3. Colas

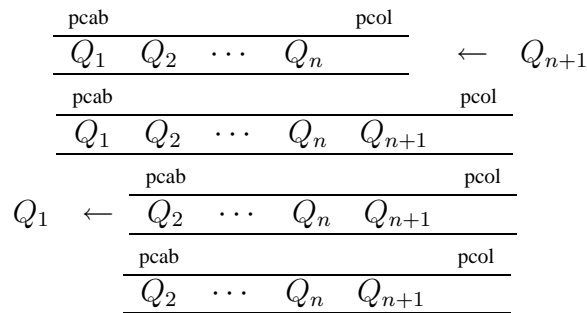
Definimos una **cola** como una estructura de datos para almacenar objetos que se caracteriza por la manera de acceder a los datos: *el primero que entra es el primero en salir* (sigue una estructura FIFO<sup>2</sup>). Con esto queremos expresar que el primer elemento que se insertó en la cola (el más antiguo temporalmente hablando) será el primero que se obtenga si se realiza una extracción. Podemos decir que los elementos se introducen por el *final* y se extraen por la *cabeza*.

Así, necesitaremos una estructura que guarde los elementos de la cola indicando el orden en el que fueron insertados, y necesitaremos un marcador para la cabeza de la cola, y así saber qué elemento se puede extraer en un determinado momento, y otro marcador al final de la cola, para saber por donde se pueden insertar nuevos elementos. A estos dos marcadores los denominaremos respectivamente *pcab* y *pcol*.

Una representación gráfica de la estructura de datos cola es la siguiente, donde se observa como se encolan los elementos al insertarlos y como se extraen elementos por la cabeza de la cola:

---

<sup>2</sup>Del inglés, *First In First Out*.



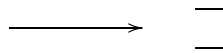
Una de las aplicaciones más comunes de la estructura de datos cola es la gestión de la cola de procesos para acceder a un recurso común, como puede ser la cola de impresión de una determinada impresora.

### 3.3.1. Operaciones sobre colas

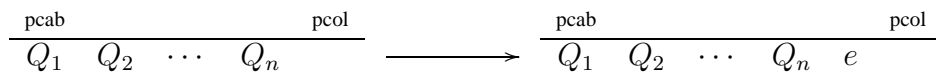
Para cada operación que definamos, veremos una representación gráfica del resultado de la función, mostrando el resultado de la operación o el estado de la cola antes y después de la ejecución.

Las operaciones más comunes que se realizan sobre una estructura de datos cola son las siguientes:

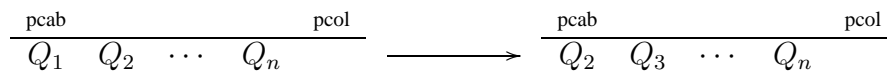
- `crearq()`: crea una cola vacía.



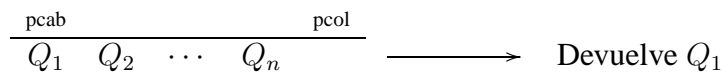
- `encolar(q, e)`: añade el elemento  $e$  a la cola  $q$ .



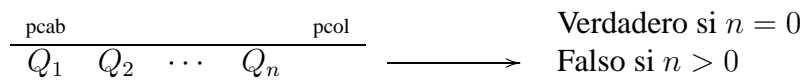
- `desencolar(q)`: elimina el elemento de la cabeza de  $q$ .



- `cabeza(q)`: consulta el primer elemento de la cola  $q$ .



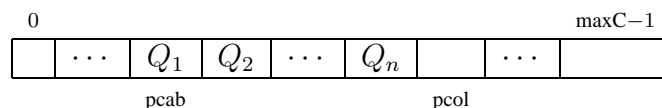
- `vaciao(q)`: consulta si la cola  $q$  está vacía.



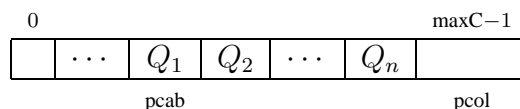
### 3.3.2. Representación vectorial de colas

Una posible representación para colas sería almacenar los elementos de la cola en posiciones consecutivas de un vector entre dos posiciones determinadas y utilizar dos variables enteras `pcab` y `pcol` para indicar esas dos posiciones que marcarán la cabeza y el final de la cola, es decir, las posiciones de donde se extraen e insertan elementos respectivamente.

Cada vez que insertemos un elemento en la cola, tendremos que incrementar `pcol`, y cada vez que extraigamos un elemento de la cola tendremos que incrementar `pcab`. Con lo que si se han producido varias inserciones y varios borrados, una situación general de una cola usada por un programa podría ser:

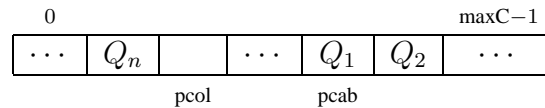


Ahora bien, si realizamos más inserciones en la cola que posiciones tiene el vector donde se almacenan los datos, entonces el marcador `pcol` se saldría del vector, con lo que tendremos que comprobar que no nos saldremos de las posiciones del vector físico. Además podría ocurrir que mientras tanto se hayan realizado algunas extracciones de elementos, con lo que el marcador `pcab` no señale la primera posición del vector, sino que haya posiciones libres al principio del vector. Esa situación sería la siguiente:

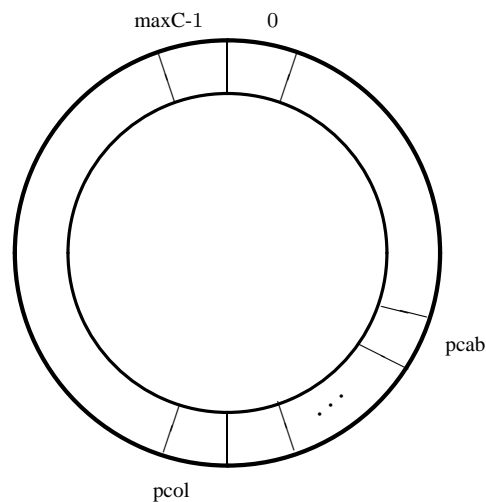


Entonces estaríamos diciendo que no podemos insertar más elementos en la cola, pero sí que quedarían posiciones libres para insertar elementos en el principio del vector. Para poder aprovechar ese espacio, haremos que `pcol` vuelva al principio del vector e inserte los posibles nuevos elementos ahí. Con esto hemos convertido el vector en lo que se conoce como cola **circular**, donde tanto los marcadores `pcol` y `pcab` pueden pasar de la última posición del vector a la primera, y podrán alcanzarse en el caso en que la cola se quede vacía o llena de elementos, pero nunca cruzarse. Para conseguir que `pcol` y `pcab` vuelvan al principio del vector utilizaremos la operación **modulo** (también conocido como **resto**), que nos da el resto de la división entera de dos números; si dividimos un número positivo cualquiera entre  $\text{maxC}$ , el resto de esta división será un número entre 0 y  $\text{maxC}-1$  y además si se incrementa en 1 el dividendo, el resto se incrementa en 1, con lo que estaremos identificando posiciones correlativas del vector. Así, cuando los marcadores `pcol` y `pcab` se incrementen, se dividirán entre  $\text{maxC}$  y su nuevo

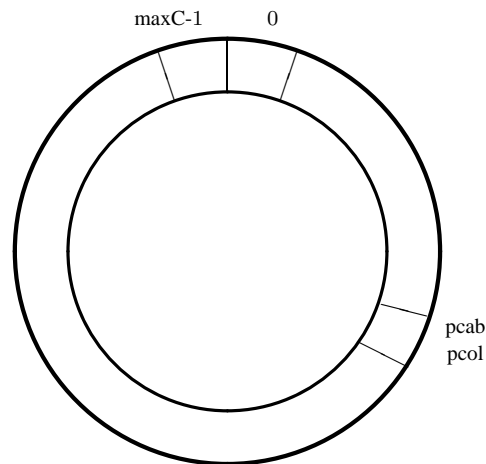
valor será el resto de esa división. Una visión genérica de una cola circular es la siguiente:



Ahora podemos plantear otro problema que puede ocurrir con la representación adoptada de cola circular, en una situación genérica en la que se hayan producido varias inserciones y extracciones de elementos podemos representar la cola de esta manera:



Ahora bien, imaginemos que llegamos a un punto en el que se hayan producido primero 20 inserciones de elementos y posteriormente se hayan producido 20 extracciones, con lo que la cola estará vacía y por ello los marcadores  $\text{pcol}$  y  $\text{pcab}$  tendrán el mismo valor (señalarán a la misma posición). Y ahora imaginemos otra situación en la que hemos insertado elementos en la cola hasta llenarla, ¿a donde señalarán los marcadores  $\text{pcol}$  y  $\text{pcab}$ ? Estarán señalando a la misma posición igualmente, puesto que  $\text{pcol}$  habrá dado una vuelta completa al vector hasta alcanzar a  $\text{pcab}$ . Gráficamente las dos situaciones se representarán de esta manera:



Con lo que necesitaremos algún mecanismo adicional para distinguir cuándo una cola está llena o está vacía, esto es, distinguir entre el caso en que `pcol` y `pcab` señalen a la misma posición. Para ello, la cola tendrá asociada una variable entera llamada `talla` que indicará el número de elementos que tiene la cola en un determinado momento. Cuando `pcol` y `pcab` señalen a la misma posición solo tendremos que comprobar la variable `talla` para comprobar si la cola está vacía o no.

Con la representación vectorial de colas tendremos el mismo problema que ocurría con la representación vectorial de pilas, el número máximo de elementos estará limitado por el tamaño físico del vector.

La definición del tipo de datos en lenguaje C para una cola con representación vectorial sería:

```
#define maxC ... /* Talla maxima del vector. */

typedef struct {
    int v[maxC]; /* Vector definido en tiempo de compilacion. */
    int pcab, pcol; /* Puntero a la cabeza y a la cola. */
    int talla; /* Numero de elementos. */
} cola;
```

La implementación de las operaciones definidas anteriormente se muestra a continuación:

```

cola *encolar(cola *q, int e) {
    if (q->talla == maxC)      /* Comprobamos si cabe el elemento. */
        /* Si no cabe hacemos un tratamiento de error. */
        tratarColaLlena();
    else {                      /* Si cabe, entonces */
        q->v[q->pcol] = e;      /* guardamos el elemento, */
        q->pcol = (q->pcol + 1)%maxC; /* incrementamos puntero de cola, */
        q->talla = q->talla + 1; /* e incrementamos la talla. */
    }
    return(q);      /* Devolvemos un puntero a la cola modificada. */
}

cola *encolar(cola *q, int e) {
    if (q->talla == maxC)      /* Comprobamos si cabe el elemento. */
        /* Si no cabe hacemos un tratamiento de error. */
        tratarColaLlena();
    else {                      /* Si cabe, entonces */
        q->v[q->pcol] = e;      /* guardamos el elemento, */
        q->pcol = (q->pcol + 1)%maxC; /* incrementamos puntero de cola, */
        q->talla = q->talla + 1; /* e incrementamos la talla. */
    }
    return(q);      /* Devolvemos un puntero a la cola modificada. */
}

cola *desencolar(cola *q) {
    /* Avanzamos el puntero de cabeza. */
    q->pcab = (q->pcab + 1) % maxC;
    q->talla = q->talla - 1;      /* Decrementamos la talla. */
    return(q);      /* Devolvemos un puntero a la cola modificada. */
}

int cabeza(cola *q) {
    return(q->v[q->pcab]);
    /* Devolvemos el elemento que hay en cabeza. */
}

int vaciaq(cola *p) {
    return(q->talla == 0); /* Devolvemos 0 (falso) si la cola */
    /* no esta vacia, y 1 (cierto) en caso contrario.*/
}

```

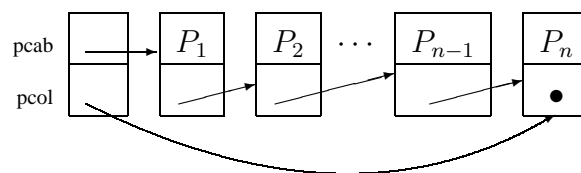


### 3.3.3. Representación enlazada de colas con variable dinámica

Al igual que con las pilas, almacenaremos cada elemento en un registro que se crea (se reserva memoria) cuando se inserta el elemento. Cada registro contendrá también, un puntero al siguiente registro correspondiente en la secuencia de elementos.

En este caso necesitaremos dos punteros, a los que llamaremos `pcab` y `pcol`, que apuntarán al primer registro de la cola (la cabeza de la cola), y al último registro de la cola (el final de la cola).

Una representación gráfica de esta estructura para colas es:



A cada registro que forme parte de la secuencia enlazada lo llamaremos nodo, al igual que para la representación enlazada de pilas.

Con esta representación dinámica para colas, no se sufrirá la limitación del número máximo de elementos que puede almacenar la cola, como ocurría con la representación vectorial. En este caso, el número máximo de elementos que puede almacenar la cola queda limitado por el número máximo de nodos que se puedan reservar en memoria.

La definición del tipo de datos en lenguaje C sería:

```
typedef struct _cnodo {  
    int e;      /* Variable para almacenar un elemento de la cola. */  
    struct _cnodo *sig;  
                /* Puntero al siguiente nodo que contiene un elemento. */  
} cnodo;  
                /* Tipo nodo. Cada nodo contiene un elemento de la cola. */  
  
typedef struct {  
    cnodo *pcab, *pcol;      /* Punteros a la cabeza y la cola. */  
} cola;
```

La implementación de las operaciones definidas anteriormente se muestra a continuación:

```

cola *crearq() {
    cola *q;

    q = (cola*) malloc(sizeof(cola));    /* Creamos una cola. */
    q->pcab = NULL;                      /* Inicializamos a NULL los punteros. */
    q->pcol = NULL;
    return(q);                          /* Devolvemos un puntero a la cola creada. */
}

cola *encolar(cola *q, int e) {
    cnodo *gaux;

    gaux = (cnodo *) malloc(sizeof(cnodo)); /* Creamos un nodo. */
    gaux->e = e;                             /* Almacenamos el elemento e. */
    gaux->sig = NULL;
    if (q->pcab == NULL) /* Si no hay ningun elemento, entonces */
        q->pcab = gaux; /* pcab apunta al nuevo nodo creado, */
    else /* y sino, */
        q->pcol->seg = gaux;
        /* el nodo nuevo va despues del que apunta pcol. */
    q->pcol = gaux;
    /* El nuevo nodo pasa a estar apuntado por pcol. */
    return(q); /* Devolvemos un puntero a la cola modificada. */
}

cola *desencolar(cola *q) {
    cnodo *gaux;

    gaux = q->pcab; /* Guardamos un puntero al nodo a borrar. */
    q->pcab = q->pcab->sig; /* Actualizamos pcab. */
    if (q->pcab == NULL) /* Si la cola se queda vacia, entonces */
        q->pcol = NULL; /* actualizamos pcol. */
    free(gaux); /* Liberamos la memoria ocupada por el nodo. */
    return(q); /* Devolvemos un puntero a la cola modificada. */
}

int cabeza(cola *q) {
    return(q->pcab->e);
    /* Devolvemos el elemento que hay en la cabeza. */
}

```

```

int vaciaq(cola *q) {
    return(q->pcab == NULL);
    /* Devolvemos 0 (falso) si la cola */
    /* no esta vacia, y 1 (cierto) en caso contrario. */
}

```

## 3.4. Listas

Definimos una **lista** como una estructura de datos formada por una secuencia de objetos. Cada objeto se referencia mediante la posición que ocupa en la secuencia.

Un ejemplo claro de aplicación de esta estructura es la representación de una lista de alumnos.

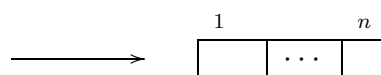
Si tenemos una lista con  $n$  elementos, las posiciones de la lista irán de la 1 a la  $n$  (no se debe contar a partir de la posición 0).

### 3.4.1. Operaciones sobre listas

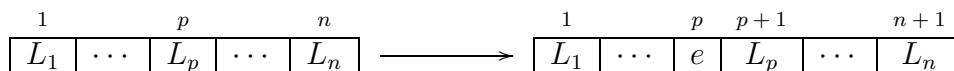
Para cada operación que definamos, veremos una representación gráfica del resultado de la función, mostrando el resultado de la operación o el estado de la lista antes y después de la ejecución.

Las operaciones más comunes que se realizan sobre una estructura de datos lista son las siguientes:

- `crearl()`: crea una lista vacía.



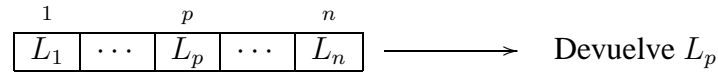
- `insertar(l, e, p)`: inserta  $e$  en la posición  $p$  de la lista  $l$ . Los elementos que están a partir de la posición  $p$  se desplazan una posición.



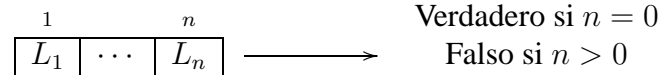
- `borrar(l, p)`: borra el elemento de la posición  $p$  de la lista  $l$ .



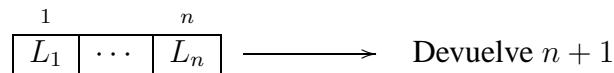
- `recuperar(l, p)`: devuelve el elemento de la posición  $p$  de la lista  $l$ .



- `vacial(l)`: consulta si la lista  $l$  está vacía.



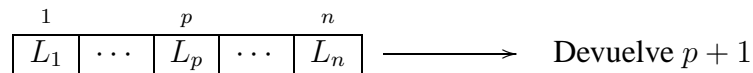
- `fin(l)`: devuelve la posición que sigue a la última en la lista  $l$ .



- `principio(l)`: devuelve la primera posición de la lista  $l$ .



- `siguiente(l, p)`: devuelve la posición siguiente a  $p$  de la lista  $l$ .

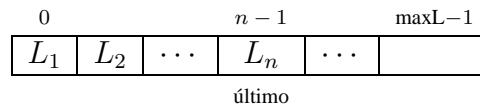


### 3.4.2. Representación vectorial de listas

Como con las estructuras de datos anteriores, para almacenar una colección de datos podemos utilizar un vector, además, en el caso de las listas donde cada elemento se accede por su posición en la lista resultará muy cómodo asignar a cada elemento una posición en el vector que se corresponda con su posición en la lista. Al trabajar con lenguaje C, donde los vectores comienzan desde la posición 0, la primera posición del vector será la posición 0, que se corresponderá con la posición 1 de la lista (el primer elemento), la posición 1 del vector se corresponderá con la posición 2 de la lista (el segundo elemento) y así sucesivamente.

Es conveniente también tener delimitadas las posiciones de la lista, es decir dónde empieza y acaba la lista, o, lo que es lo mismo, cuál es la primera posición y cuál es la última. Con esta representación vectorial la primera posición de la lista queda marcada por la posición 0 del vector, pero será necesario utilizar una variable `ultimo` de tipo entero que señale cuál es la última posición ocupada del vector.

Una representación gráfica de una lista almacenada mediante un vector es la siguiente:



De la misma manera que con las representaciones vectoriales vistas para pilas y colas, esta representación tendrá limitado el número máximo de elementos que se pueden almacenar en la lista al tamaño del vector.

La definición del tipo de datos en lenguaje C sería:

```
#define maxL ... /* Talla maxima del vector. */

typedef int posicion;
/* Cada posicion se referencia con un entero. */

typedef struct {
    int v[maxL]; /* Vector definido en tiempo de compilacion. */
    posicion ultimo; /* Posicion del ultimo elemento. */
} lista;
```

La implementación de las operaciones definidas anteriormente se muestra a continuación:

```
lista *crearl() {
    lista *l

    l = (lista *) malloc(sizeof(lista)); /* Creamos la lista. */
    l->ultimo = -1; /* Inicializamos el puntero al ultimo. */
    return(l); /* Devolvemos un puntero a la lista creada. */
}
```

```

lista *insertar(lista *l, int e, posicion p) {
    posicion i;

    if (l->ultimo == maxL-1) /* Comprobamos si cabe el elemento. */
        /* Si no cabe hacemos un tratamiento de error. */
        tratarListaLlena();
    else {
        /* Si cabe, entonces */
        /* hacemos un vacio en la posicion p, */
        for (i=l->ultimo; i>=p; i--)
            l->v[i+1] = l->v[i];
        l->v[p] = e; /* guardamos el elemento, */
        /* e incrementamos el puntero al ultimo. */
        l->ultimo = l->ultimo + 1;
        return(l);
        /* Devolvemos un puntero a la lista modificada. */
    }
}

lista *borrar(lista *l, posicion p) {

    /* Desplazamos los elementos del vector. */
    for (i=p; i<l->ultimo; i++)
        l->v[i] = l->v[i+1];
    /* Decrementamos el puntero al ultimo. */
    l->ultimo = l->ultimo - 1;
    return(l); /* Devolvemos un puntero a la lista modificada. */
}

int recuperar(lista *l, posicion p) {
    return(l->v[p]);
    /* Devolvemos el elemento que hay en la posicion p. */
}

int vacial(lista *l) {
    return(l->ultimo < 0);
    /* Devolvemos 0 (falso) si la lista */
    /* no esta vacia, y 1 (cierto) en caso contrario. */
}

```

```

posicion fin(lista *l) {
    return(l->ultimo + 1);
    /* Devolvemos la posicion siguiente a la ultima. */
}

posicion principio(lista *l) {
    return(0);          /* Devolvemos la primera posicion. */
}

posicion siguiente(lista *l, posicion p) {
    return(p+1);
    /* Devolvemos la posicion siguiente a la posicion p. */
}

```

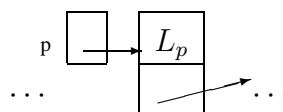
### 3.4.3. Representación enlazada de listas con variable dinámica

La representación enlazada para listas usando variables dinámicas va a ser muy similar a la técnica usada para las representaciones enlazadas dinámicas de pilas y colas. Cada elemento estará almacenado en un registro junto con un puntero que apuntará al registro que guarde el siguiente elemento de la lista, o dicho de otra manera, apuntará a la siguiente posición de la lista.

En esta representación las posiciones de los elementos se corresponderán con punteros, esto es, una posición de la lista es un puntero.

Ahora bien, ante este tipo de representación tenemos dos posibilidades a la hora de almacenar un elemento en su posición correspondiente:

1. Dada una posición  $p$ , el elemento  $L_p$  está en el nodo apuntado por  $p$ , es decir:



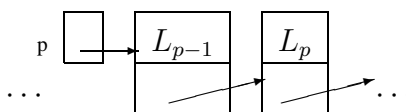
Si suponemos que el campo del nodo donde se almacena el valor de los elementos de la lista se denomina  $e$ , entonces para acceder al valor del elemento  $L_p$  utilizaremos la expresión  $p \rightarrow e$ .

Sin embargo esta opción tiene un inconveniente para las operaciones de inserción y borrado de elementos, ya que el coste de estas operaciones será de  $O(n)$ , siendo  $n$  el número de elementos de la lista. Esto es debido a que para insertar o borrar de una determinada posición (puntero) en la lista, es necesario conocer el puntero del nodo previo al nodo a insertar o borrar, para

ello habrá que hacer un recorrido por los nodos de la lista desde el principio de ésta hasta que encontremos la posición anterior a la que vamos a insertar o borrar, de aquí el coste temporal lineal.

La otra opción posible para almacenar los elementos de una lista es:

2. Dada una posición  $p$ , el elemento  $L_p$  está en el nodo apuntado por  $p \rightarrow \text{sig}$ , es decir:

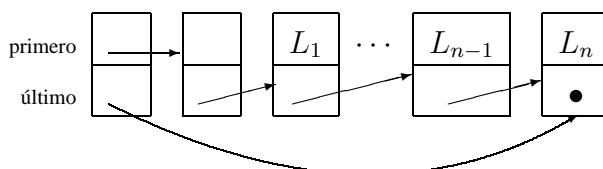


Con esta opción, la manera de acceder al valor del elemento  $L_p$  correspondiente a la posición  $p$  es con la expresión  $p \rightarrow \text{sig} \rightarrow e$ .

Además, esta forma de referenciar los elementos nos permite realizar las operaciones de inserción y borrado de elementos con un coste  $O(1)$ , puesto que en este caso sí que es conocido el puntero al nodo previo al nodo a insertar o borrar. Usaremos pues, esta representación para las listas enlazadas mediante variable dinámica; ello determinará los pasos a seguir en las operaciones de consulta, inserción, borrado, etc.

Al usar la segunda opción de las vistas anteriormente para listas enlazadas con variable dinámica, donde dada una posición  $p$ , el elemento  $L_p$  está en  $p \rightarrow \text{sig}$ , el primer elemento de la lista también necesitará un nodo previo al nodo donde él esté almacenado. Este nodo será un nodo que no contendrá ningún valor y al que llamaremos nodo centinela. El nodo centinela permitirá optimizar las operaciones de actualización.

Además, al igual que para la representación vectorial de listas, necesitaremos conocer donde está la primera y última posición de la lista (en este caso serán dos punteros), para lo que tendremos dos variables `primero` y `ultimo` que guardarán dicha información. Representando gráficamente una lista enlazada usando variables dinámicas:



Al igual que ocurría con la representación enlazada con variables dinámicas para pilas y colas, aquí el número máximo de elementos que puede almacenar una lista viene determinado por el número máximo de nodos que se puedan crear en memoria.



La definición del tipo de datos en lenguaje C sería:

```
typedef struct _lnodo {  
    int e;    /* Variable para almacenar un elemento de la lista. */  
    struct _lnodo *sig;  
        /* Puntero al siguiente nodo que contiene un elemento. */  
} lnodo  
  
typedef lnodo *posicion;  
        /* Cada posicion se referencia con un puntero. */  
  
typedef struct {    /* Definimos el tipo lista con un puntero */  
    posicion primero, ultimo;    /* al primero y ultimo nodos. */  
} lista;
```

La implementación de las operaciones definidas anteriormente se muestra a continuación:

```
lista *crearl() {  
    lista *l;  
  
    l = (lista *) malloc(sizeof(lista));    /* Creamos una lista. */  
    l->primero = (lnodo *) malloc(sizeof(lnodo));  
        /* Creamos el centinela */  
    l->primero->sig = NULL;  
    l->ultimo = l->primero;  
    return(l);    /* Devolvemos un puntero a la lista creada. */  
}
```

```

lista *insertar(lista *l, int e, posicion p) {
    posicion q;

    q = p->sig;    /* Dejamos q apuntando al nodo que se desplaza. */
    p->sig = (lnodo *) malloc(sizeof(lnodo));
                                   /* Creamos un nodo. */
    p->sig->e = e;                /* Guardamos el elemento. */
    p->sig->sig = q;
    /* El sucesor del nuevo nodo esta apuntado por q. */

    /* Si el nodo insertado ha pasaso a ser el ultimo, */
    if (p == l->ultimo)
        l->ultimo = p->sig;      /* actualizamos ultimo. */
    return(l);
    /* Devolvemos un puntero a la lista modificada. */
}

lista *borrar(lista *l, posicion p) {
    posicion q;

    if (p->sig == l->ultimo)
        /* Si el nodo que borramos es el ultimo, */
        l->ultimo = p;          /* actualizamos ultimo. */
    q = p->sig;    /* Dejamos q apuntando al nodo a borrar. */
    p->sig = p->sig->sig; /* p->sig apuntara a su sucesor. */
    free(q);
    /* Liberamos la memoria ocupada por el nodo a borrar. */
    return(l);
    /* Devolvemos un puntero a la lista modificada. */
}

int recuperar(lista *l, posicion p) {
    return(p->sig->e);
    /* Devolvemos el elemento que hay en la posicion p. */
}

int vacial(lista *l) {
    return(l->primero->sig == NULL);
    /* Devolvemos 0 (falso) si la lista */
    /* no esta vacia, y 1 (cierto) en caso contrario. */
}

```

```

posicion fin(lista *l) {
    return (l->ultimo);    /* Devolvemos la ultima posicion. */
}

posicion principio(lista *l) {
    return (l->primero);    /* Devolvemos la primera posicion. */
}

posicion siguiente(lista *l, posicion p) {
    return (p->sig);
    /* Devolvemos la posicion siguiente a la posicion p. */
}

```

#### 3.4.4. Representación enlazada de listas con variable estática

Hasta ahora sólo hemos visto representaciones enlazadas para las estructuras de datos usando variables dinámicas. Ahora vamos a ver una representación enlazada de listas utilizando variables estáticas que van a ser en primer lugar un vector para almacenar los valores de los elementos (hasta ahora nada nuevo) y después otro vector adicional en el que vamos a almacenar el orden de los elementos, es decir, las posiciones. En este vector de posiciones se indicará que elemento va detrás de otro en la lista mediante índices que indican posiciones físicas del vector. Así pues, en esta representación para listas las posiciones serán números enteros.

Así pues, la estructura de datos estará compuesta por 2 vectores *v* y *p* de igual tamaño, uno para los valores de los elementos y otro para los índices que *enlazan* los diferentes elementos de la lista en el orden adecuado.

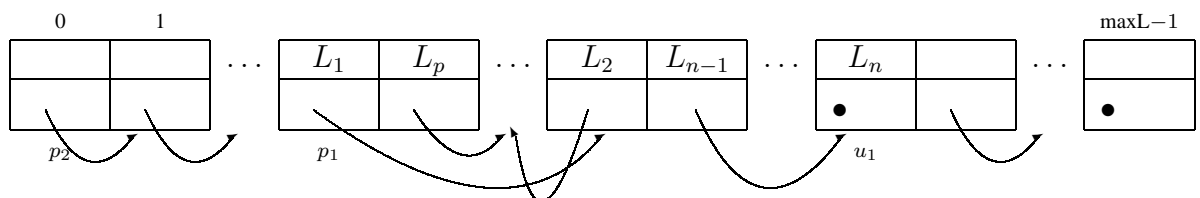
Como para las otras representaciones de listas necesitaremos conocer donde empieza la lista (la primera posición de la lista) y donde termina (la última posición de la lista), para ello tendremos dos variables enteras *p1* y *u1* que señalarán el principio y final de la lista respectivamente.

A cada par formado por un valor del vector *v* y su correspondiente del vector *p* lo llamaremos nodo, al igual que para el resto de estructuras de datos enlazadas.

Ahora bien, cuando queramos insertar un nuevo elemento en la lista, necesitaremos tomar una de las posiciones de los vectores *v* y *p* que no estén ocupadas ya por la lista y convertirla en un nuevo nodo que pertenezca a la lista enlazándolo correctamente con el resto de nodos. Para conocer qué posiciones del vector están libres para insertar nuevos elementos vamos a aprovechar las posiciones del vector *p* para enlazarlas en una lista simple. Es decir, a partir de uno de los nodos vacíos podemos conocer el resto de nodos vacíos siguiendo los enlaces que los unen for-

mando una lista simple. El principio de esta lista de nodos vacíos lo guardaremos en una variable entera  $p_2$  que señalará al primer nodo de la lista de nodos vacíos.

Una representación gráfica de todo lo expuesto es la siguiente:



Al trabajar con una representación con variable estática existirá nuevamente una limitación del número máximo de elementos que se pueden almacenar en la lista, determinada por el tamaño físico del vector.

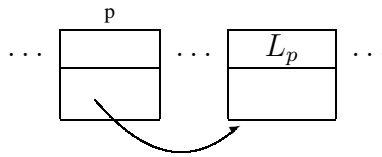
La definición del tipo de datos en lenguaje C sería:

```
#define maxL ... /* Talla maxima del vector. */

typedef posicion int;
/* El tipo posicion se define como un entero. */

typedef struct {
    int v[maxL]; /* Vector definido en tiempo de compilacion. */
    posicion p[maxL];
    /* Vector de posiciones creado en tiempo de ejecucion. */
    posicion p1, /* Puntero al principio de la lista. */
        u1, /* Puntero al fin de la lista. */
        p2;
    /* Puntero al principio de la lista de nodos vacios. */
} lista;
```

Como trabajamos con una representación enlazada de listas, tendríamos dos opciones a la hora de identificar el lugar donde se almacena un elemento con la posición que le corresponde. Una opción es aquella donde una posición  $p$  señala a la posición del vector donde está almacenado su elemento correspondiente  $L_p$ . Otra opción es aquella donde una posición  $p$  señala la posición del vector *previa* a la posición del vector donde se almacena el elemento correspondiente  $L_p$ ; por *previa* entenderemos que es la posición anterior en la lista, pero no necesariamente la anterior posición física del vector. Gráficamente podemos representarlo de esta manera:



La forma de acceder al valor del elemento  $L_p$  es mediante la expresión  $L = l.v[l.p[p]]$ .

Escogeremos esta forma de acceder a los elementos de la lista puesto que optimizará las actualizaciones (inserciones, borrados, etc). Debemos tener en cuenta que al usar esta representación, existirá un nodo *centinela* al principio de la lista de elementos.

A continuación se muestra tan solo la implementación de las operaciones `crearl`, `insertarl` y `borrar`:

```
lista *crearl() {
    lista *l;
    int i;

    l = (lista *) malloc(sizeof(lista)); /* Creamos la lista. */
    l->p1 = 0;                          /* El nodo 0 es el centinela. */
    l->u1 = 0;
    l->p[0] = -1;
    l->p2 = 1;

    /* La lista de nodos vacios comienza en el node 1. */
    /* Construimos la lista de nodos vacios. */
    for (i=1; i<maxL-1; i++)
        l->p[i] = i+1;
    l->p[maxL-1] = -1;

    /* El ultimo nodo vacio no apunta a ningun lugar. */
    return(l); /* Devolvemos un puntero a lista construida. */
}
```

```

lista *insertarl(lista *l, int e, posicion p) {
    posicion q;

    if (l->p2 == -1) /* Si no quedan nodos vacios, */
        tratarListaLlena(); /* hacemos un tratamiento de error. */
    else {
        q = l->p2; /* Dejamos un puntero al primer nodo vacio. */
        l->p2 = l->p[q];
        /* El primer nodo vacio sera el sucesor de q. */
        l->v[q] = e;
        /* Guardamos el elemento en el nodo reservado. */
        l->p[q] = l->p[p];
        /* Su sucesor pasa a ser el de la pos. p. */
        l->p[p] = q;
        /* El sucesor del nodo apuntado por p pasa a ser q. */
        /* Si el nodo que hemos insertado pasa a ser el ultimo, */
        if (p == l->ul)
            l->ul = q; /* actualizamos el puntero ul. */
        return(l); /* Devolvemos un puntero a la lista modificada. */
    }
}

lista *borrar(lista *l, posicion p) {
    posicion q;

    if (l->p[p] == l->ul) /* Si el nodo que borramos es el ultimo, */
        l->ul = p; /* actualizamos ul. */
    q = l->p[p]; /* Dejamos q apuntando al nodo a borrar. */
    l->p[p] = l->p[q];
    /* El sucesor del nodo apuntado por p pasa a */
    /* ser el sucesor del nodo apuntado por q. */
    l->p[q] = l->p2;
    /* El nodo que borramos sera el primero de los vacios. */
    l->p2 = q;
    /* El principio de la lista de nodos vacios comienza en q. */
    return(l);
}

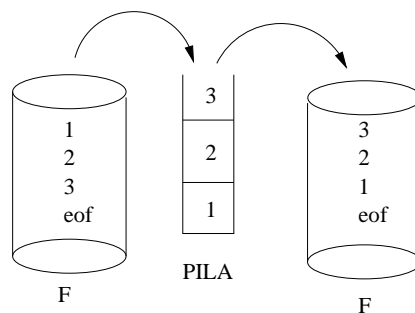
```

## 3.5. Ejercicios

### Ejercicio 1:

-Realizar un programa que invierta un pequeño fichero de datos F utilizando una estructura dinámica como soporte auxiliar (no se deben usar otros ficheros). Ofrecer la solución al problema en lenguaje C. Deben utilizarse los nombres de las operaciones vistas en clase. Los elementos almacenados en el fichero son caracteres, es decir, es un fichero de texto.

**Nota:** Recordar que en una pila los elementos estarán apilados en el orden inverso al de entrada en la pila, con lo que podemos considerar la PILA como la estructura típica de inversión.



### Solución:

La estructura típica de inversión es la pila. Una pila es toda aquella estructura sobre la cual el último elemento en entrar es el primero en salir, y por consiguiente, el primero que entró será el último en salir; los elementos estarán en orden inverso al de entrada.

Una pila se puede implementar de muchas formas, pero cada una de ellas tiene unas limitaciones. En este caso la implementación más correcta sería una implementación mediante variables dinámicas debido a que no se conoce a priori el número de elementos que forman el fichero.

```

void invierte(char *fich) {
    FILE *f;
    pila *p;
    char e;

    /* leemos el fichero y guardamos sus elementos en la pila */
    f = fopen(fich, "r");
    p = crearp();
    while ((e=fgetc(f)) != NULL)
        apilar(p, e);
    fclose(f);

    /* escribimos los elementos en el fichero a la inversa */
    f = fopen(fich, "w");
    while (!vaciap(p)) {
        e = tope(p);
        desapilar(p);
        fputc(e, f);
    }
    fclose(f);
}

```

---



### Ejercicio 2:

-Diseñar en lenguaje C un algoritmo que dada una pila de enteros, calcule la suma y el producto de los elementos de la pila.

### Solución:

```
void operaciones_pila (pila *p) {
    int suma, producto, e;

    /* el elemento neutro de la suma es el 0 */
    suma = 0;
    /* el elemento neutro del producto es el 1 */
    producto = 1;
    /* leemos la pila y acumulamos la suma y el producto */
    while (!vaciap(p)) {
        e = tope(p);
        desapilar(p);
        suma = suma + e;
        producto = producto * e;
    }
    printf("la suma es %d\n", suma);
    printf("el producto es %d\n", producto);
}
```

---

### Ejercicio 3:

-Dada cierta pila p de números reales y un valor real x, se desea realizar una operación `multiplicaPila(p, x)` que multiplique todos los elementos de la pila p por dicho valor x. Dicha operación deberá realizarse sin hacer uso de ninguna otra estructura de datos auxiliar, aparte de la propia p inicial y utilizando exclusivamente las operaciones de la clase pila.

#### Ejemplo:

```
// Inicialmente : p = 2.3 | 1.2 | -1.16 | 168
multiplicaPila(p, 3.0);
// Finalmente : p = 6.9 | 3.6 | -3.48 | 504
```

Se pide implementar **recursivamente** la operación `multiplicaPila(p, x)` utilizando solo las operaciones de la clase pila y sin hacer uso de ninguna otra estructura de datos auxiliar.

**Nota:** Al emplear un esquema recursivo, las variables locales de la función pueden servir para guardar el estado de la pila antes de cada llamada recursiva y poder reconstruir la pila de la manera deseada.

### **Solución:**

La estrategia a seguir es la misma que se utiliza en los algoritmos de inversión recursiva de una cola o una lista a partir de una posición determinada; es decir, para lograr una reconstrucción in-situ de la estructura original pero invertida, utilizamos la estructura local (registro de activación) que soporta cada llamada recursiva, en particular la variable local *e* para memorizar los estados de la pila en cada llamada antes de la inversión. Utilizando este mismo mecanismo para modificar una pila, al realizarse el acceso siempre por su tope, una vez realizada la modificación de todos sus elementos (todas las llamadas recursivas) podemos obtener una reconstrucción in-situ de la pila original pero con sus elementos multiplicados por *x*.

```
void multiplicaPila(pila *p, float x)
    float e;

    if (!vaciap(p)) {
        e = x * tope(p);
        desapilar(p);
        multiplicaPila(p, x);
        apilar(p, e);
    }
}
```

---

#### Ejercicio 4:

-Dada la especificación del TAD lista vista en clase, diseñar una función `longitud(l)` que devuelva el número de elementos que tiene una lista `l` dada. Utilizar únicamente las operaciones de la especificación.

#### Solución:

```
int longitud(lista *l) {
    tipo_posicion pos;
    int i;

    if (vacial(l))
        /* la lista tiene 0 elementos */
        return(0);
    else {
        /* recorreremos toda la lista y contaremos cuantos */
        /* elementos tiene */
        pos = principio(l);
        i=1;
        while (pos != fin(l)) {
            pos = siguiente(l, pos);
            if (pos != fin(l)) i++;
        }
        return(i);
    }
}
```

---

### Ejercicio 5:

-Suponer que queremos añadir las siguientes funciones para manipular el TAD lista:

- `localiza (l, x)` : devuelve la posición de `x` en la lista `l`. Si `x` aparece más de una vez en `l`, devuelve la posición de la primera aparición. Si `x` no se encuentra en la lista devuelve `fin(l)` ;.
- `borra (l, x)` : borra el elemento `x` de la lista `l`. Si `x` aparece más de una vez en `l`, borra la primera aparición. Si `x` no se encuentra en la lista, no hace nada.

Implementar en lenguaje C dichas operaciones a partir de una representación vectorial y una representación enlazada con variables dinámicas como las vistas en clase. Suponer que trabajamos con listas de enteros.

### Solución:

- REPRESENTACIÓN VECTORIAL:

```
posicion localiza(lista *l, int x)
{
    posicion pos;

    pos = 0;
    while ((pos != l->ultimo) && (l->v[pos] != x))
        pos++;
    return(pos);
}

lista *borra(lista *l, int x)
{
    posicion pos, i;

    pos = 0;
    while ((pos != l->ultimo) && (l->v[pos] != x)) pos++;
    for (i=pos; i<l->ultimo; i++)
        l->v[i] = l->v[i+1];
    l->ultimo = l->ultimo - 1;
    return(l);
}
```

■ REPRESENTACIÓN ENLAZADA CON VARIABLES DINÁMICAS:

```
posicion localiza(lista *l, int x)
{
    posicion pos;

    pos = l->primero;
    while ((pos != l->ultimo) && (pos->seg->e != x))
        pos = pos->seg;
    return(pos);
}
```

```
lista *borra(lista *l, int x)
{
    posicion pos, q;

    pos = l->primero;
    while ((pos != l->ultimo) && (pos->seg->e != x))
        pos = pos->seg;
    if (pos->seg == l->ultimo)
        l->ultimo = pos;
    q = pos->seg;
    pos->seg = pos->seg->seg;
    free(q);
    return(l);
}
```

---

### Ejercicio 6:

-Diseñar un algoritmo que tome como argumento dos listas de enteros l1 y l2, y dé como resultado otra lista l3 solo con aquellos elementos de l1 que no estén en l2 y los de l2 que no estén en l1. Sólo se permite utilizar las operaciones del TAD lista visto en clase y las vistas en el ejercicio num. 5.

### Solución:

```
lista * diferencia_de_listas(lista *l1, lista *l2) {
    lista *l3;
    posicion pos;
    elemento e;

    /* recorremos la lista l1 e insertamos en l3 todos */
    /* sus elementos que no esten en l2 */
    pos = principio(l1);
    while (pos != fin(l1)) {
        e = recuperar(l1, pos);
        if (localiza(l2, e) == fin(l2))
            insertar(l3, e, 1);
        pos = siguiente(l1, pos);
    }

    /* recorremos la lista l2 e insertamos en l3 todos */
    /* sus elementos que no esten en l1 */
    pos = principio(l2);
    while (pos != fin(l2)) {
        e = recuperar(l2, pos);
        if (localiza(l1, e) == fin(l1))
            insertar(l3, e, 1);
        pos = siguiente(l2, pos);
    }

    return(l3);
}
```

---

### Ejercicio 7:

-Diseñar una función **recursiva** que, dada una cola, la invierte. Sólo se permite utilizar las operaciones del TAD cola visto en clase, no se conoce la longitud de la cola y no hay que utilizar ninguna estructura auxiliar.

**Nota:** Utilizar las variables locales de la función recursiva de manera que la pila de llamadas recursivas pueda servirnos para invertir la cola.

### Solución:

Vamos a intentar comprender el esquema recursivo de inversión: supongamos que tenemos una cola, si nos guardamos el elemento que está en la cabeza de la cola y lo desencolamos, entonces el resto de la cola formaría una "nueva cola", si conseguimos invertir esta "nueva cola", obtener la inversión de la cola original consistirá en encolar el elemento que habíamos extraído de la cabeza y el problema estaría resuelto. Así pues, nuestro problema ahora es obtener la inversión de la "nueva cola", que resulta ser el mismo problema que estamos tratando desde el principio, con lo que hemos encontrado nuestro esquema recursivo.

```
cola * invierte_cola (cola *c) {
    elemento e;

    /* comprobamos si la cola tiene mas de un elemento */
    if (cabeza(c) != cola(c)) {
        /* guardamos la cabeza y la desencolamos */
        e = cabeza(c);
        desencolar(c);
        /* invertimos el resto de la cola */
        c = invierte_cola(c);
        /* encolamos al final de la cola el elemento */
        /* que era el primero */
        c = encolar(c, e);
    }
    return(c);
}
```

---

### Ejercicio 8:

-El recorrido de los autobuses de una línea interurbana que recorre varios pueblos se puede interpretar mediante una cola, donde cada nodo es una parada.

La información contenida en cada nodo es:

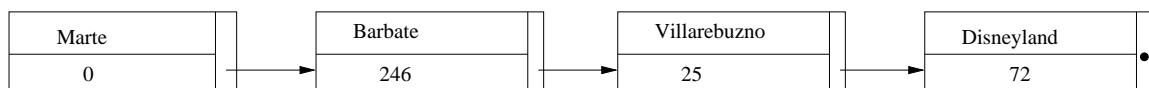
- Nombre del pueblo (String de 15 caracteres).
- Distancia en Km al anterior pueblo en la cola del recorrido del autobús.
- Puntero a la siguiente parada (es decir, al nodo del siguiente pueblo).

Una persona desea saber cuantos Kilometros va a recorrer en autobús entre dos pueblos del recorrido. Entre estos dos pueblos puede haber varios pueblos más.

Se pide:

- Escribir la definición del tipo de datos en lenguaje C. Debe utilizarse una representación **enlazada** de colas puesto que a priori no se sabe el número de paradas que va a tener una línea de autobús.
- Implementar en lenguaje C una función para esa representación enlazada de colas que reciba como parámetros la cola de las paradas del autobús, y dos nombres de dos de las paradas en la cola y que devuelva la distancia en Km entre esas dos paradas. (Suponemos que las paradas se proporcionan a la función en el orden de aparición en el camino, es decir, la primera parada que se especifica aparece antes en la cola que la segunda parada especificada).

Un ejemplo de cola de paradas sería:



y una llamada a la función sería:

```
kilometros = cuenta_km(cola_paradas, "Barbate", "Disneyland");
```

donde la variable `kilometros` acabaría valiendo  $97=25+72$ .



### Solución:

```
/* definicion del tipo de datos */
typedef struct _cnode {
    char nombre[15];      /* nombre de la parada */
    int distancia;        /* distancia a la anterior parada */
    struct _cnode *siguiente;
                          /* puntero a la siguiente parada */
} cnode;

typedef struct {
    cnode *pcab, *pcola;  /* punteros a la cabeza y la cola */
} cola;

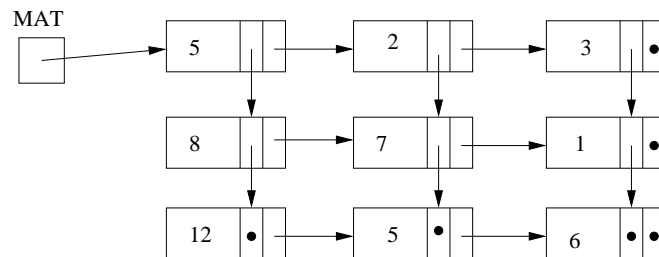
/* funcion para hallar la distancia entre 2 estaciones */
int cuenta_km(col a *c, char *ciudad1, char *ciudad2)
{
    int dist;             /* para acumular la distancia */
    cnode *aux;           /* para recorrer la cola */

    aux=c->pcab;
    /* buscamos la primera ciudad */
    while ((aux != c->pcola) && (strcmp(aux->nombre,ciudad1)!=0))
        aux = aux->siguiente;
    /* sumamos las distancias hasta la segunda ciudad */
    dist = 0;
    while ((aux != c->pcola) && (strcmp(aux->nombre,ciudad2)!=0)) {
        /* OJO!, movemos el puntero antes de */
        /* acumular distancias */
        aux = aux->siguiente;
        dist = dist + aux->distancia;
    }
    return(dist);
}
```

---

### Ejercicio 9:

-Tenemos dos matrices **CUADRADAS** de enteros y del mismo orden (mismo tamaño). La estructura de estas dos matrices es dinámica y puede verse un esquema en la siguiente figura:



Se pide realizar las siguientes cuestiones en lenguaje C:

- Definir el tipo de datos (la estructura) matriz, basándose en la figura de arriba.
- Diseñar una función que devuelva una matriz que sea el resultado de sumar dos matrices que recibe como parámetros. Utilizar la estructura dinámica elegida para las matrices.

**Nota:** Para definir la estructura sólo hay que mirar el dibujo para ver que todos los punteros apuntan a la misma cosa (al mismo tipo de objeto).

No conocemos a priori el orden de la matriz, esto es, si es de 3x3 o 4x4, etc, el enunciado sólo nos asegura que es cuadrada, y que las dos matrices con las que trabaja el programa principal son del mismo orden.

Otro aspecto a considerar es que para recorrer una matriz "tradicional" o "normal" se necesitaban dos variables índices de fila y columna; aquí es exactamente igual, con la diferencia de que estas variables serán punteros.

### Solución:

```
/* definicion de tipos */
#typedef struct _mat_node {
    int elemento;      /* el entero de este nodo */
    struct _mat_mode *horiz;  /* puntero horizontal */
    struct _mat_mode *vert;   /* puntero vertical */
} mat_node;

#typedef mat_node matriz;      /* tipo de datos matriz */

/* funcion que devuelve la suma de dos matrices */
matriz *suma_mat(matriz *mat1, *mat2)
{
    mat_node *fila1, *fila2;
    mat_node *col1, *col2;

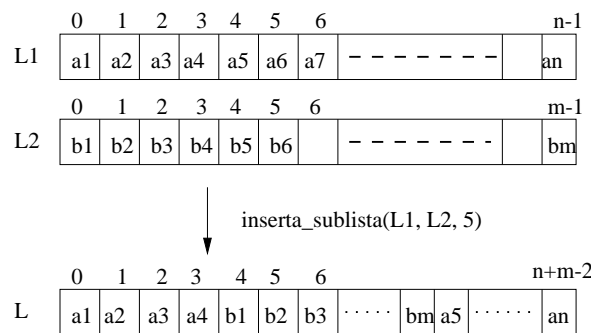
    fila1 = mat1;
    fila2 = mat2;
    /* Recorremos las 2 matrices, solo es necesario */
    /* comprobar que hemos recorrido completamente la */
    /* matriz 1 puesto que son del mismo tamaño. */
    /* Acumulamos el resultado sobre la matriz 1 */
    while (fila1 != NULL) {
        col1 = fila1;
        col2 = fila2;
        while (col1 != NULL) {
            col1->elemento = col1->elemento + col2->elemento;
            col1 = col1->horiz;
            col2 = col2->horiz;
        }
        fila1 = fila1->vert;
        fila2 = fila2->vert;
    }
    return(mat1);
}
```

---

### Ejercicio 10:

-Dada dos listas de números enteros L1 y L2, se pide implementar una función que inserte la lista L2 como una sublista de L1 en la posición de L1 que se indique. Esto es, tenemos la lista L1, por otro lado tenemos la lista L2, suponiendo una posición de L1 indicada, la lista L resultante contendrá por el mismo orden en que estaban almacenados anteriormente todos los elementos de L1 hasta la posición indicada, a partir de ahí se habrán insertado todos los elementos de L2 y por último estarán el resto de elementos de L1 que quedaban.

Un esquema de la operación se puede observar en la figura:



Se pide realizar lo siguiente:

- Implementar la función anterior utilizando para ello tan solo las operaciones del TAD lista vistas en clase. No debe utilizarse ninguna representación interna específica puesto que no es necesario.
- Implementar en lenguaje C la función anterior como si fuera una operación del TAD lista. En este caso se debe realizar accediendo a la estructura interna de la lista y manejándola adecuadamente. Deberá utilizarse la representación vectorial de listas vista en clase. Realmente esta operación será una generalización de la operación `insertar(l, e, p)` para listas vista en clase, donde en vez de insertar un solo entero se inserta una lista de enteros.

### Solución:

-Implementación de la función usando operaciones del TAD:

```
lista * inserta_sublista(lista *L1, lista *L2,
                        posicion pos) {
    lista *L;
    posicion pos1, pos2, posL;
    int e;

    /* creamos la nueva lista */
    L = crearl();
    posL = principio(L);
    pos1 = principio(L1);
    /* insertaremos elementos de L1 hasta que lleguemos a */
    /* la posicion pos, y entonces insertaremos toda la */
    /* lista L2 y despues seguiremos insertando de L1 */
    while (pos1 != fin(L1)) {
        if (pos1==pos) {
            pos2 = principio(L2);
            while (pos2 != fin(L2)) {
                e = recuperar(L2, pos2);
                L = insertar (L, e, posL);
                pos2 = siguiente(L2, pos2);
                posL = siguiente(L, posL);
            }
        }
        else {
            e = recuperar(L1, pos1);
            L = insertar (L, e, posL);
            posL = siguiente(L, posL);
        }
        pos1 = siguiente(L1, pos1);
    }
    return (L);
}
```

-Implementación en lenguaje C de la operación con una representación vectorial de listas:

```
/* funcion para insertar una lista dentro de otra en */
/* una posicion determinada */
lista *insertar_sublista(lista *L1, *L2, posicion pos)
{
    lista *L;          /* nueva lista a crear */
    posicion pos1, pos2, posL;
                        /* apuntadores de las posiciones de las */
                        /* diferentes listas a manejar */

    /* creamos la nueva lista */
    L = (lista *) malloc(sizeof(lista));
    posL = 0;
    pos1 = 0;
    /* insertamos en la nueva lista */
    while (pos1 != L1->ultimo) {
        /* si hemos llegado a la posicion pos, se insertan */
        /* todos los elementos de L2 seguidos */
        if (pos1 == pos) {
            pos2 = 0;
            while (pos2 != L2->ultimo) {
                L->v[posL] = L2->v[pos2];
                pos2++;
                posL++;
            }
        }
        /* sino insertamos un elemento mas de L1 */
        else {
            L->v[posL] = L1->v[pos1];
            posL++;
        }
        pos1++;
    }
    /* actualizamos el tamaño de la lista creada */
    L->ultimo = posL;
    return(L);
}
```

## Tema 4

# Divide y vencerás

### 4.1. Esquema general de “Divide y Vencerás”

La técnica de diseño de algoritmos llamada *divide y vencerás* consiste en, dado un problema a resolver de talla  $n$ :

1. Dividir el problema en problemas de talla más pequeña (*subproblemas*),
2. Resolver independientemente los *subproblemas* (de manera recursiva),
3. Combinar las soluciones de los *subproblemas* para obtener la solución del problema original.

#### Características

- Es un esquema claramente recursivo.
- Determinados algoritmos tienen un elevado coste temporal en la parte de división del problema en subproblemas, mientras que otros presentan un elevado coste temporal en la parte de combinación de resultados.
- El esquema “divide y vencerás” permite obtener **soluciones eficientes** cuando los *subproblemas* tienen una talla lo más parecida posible.

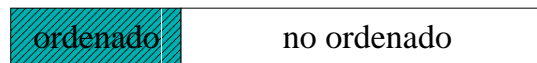
## 4.2. Algoritmos de ordenación

**Problema:** Ordenar de manera no decreciente un conjunto de  $n$  enteros almacenado en un vector  $A$ .

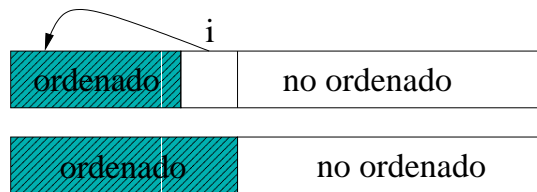
### 4.2.1. Inserción directa

**Estrategia:**

1. Se asume que existen dos partes en el vector: una ordenada y otra sin ordenar.



2. Se selecciona el elemento que ocupa la primera posición en la parte sin ordenar, y se **inserta** en una posición de la parte ordenada de tal forma que se mantenga la ordenación en dicha parte.



**Funcionamiento:**

Inicialmente y debido a que, en general, el vector  $A$  no presentará ningún tipo de ordenación, se considerará que la parte ordenada estará formada únicamente por el primer elemento del vector; siguiendo la estrategia del algoritmo, en los pasos sucesivos la ordenación en la parte ordenada siempre se mantendrá hasta que se consiga la ordenación de todo el vector (ver figura 4.1).



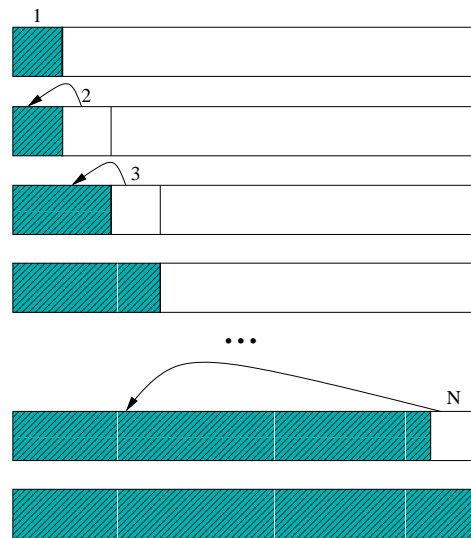


Figura 4.1: Estrategia que sigue el algoritmo de ordenación “inserción directa”. La parte sombreada representa la parte ordenada en el vector.

A continuación, se presenta una versión del algoritmo “inserción directa”<sup>1</sup>:

---

<b>Algoritmo:</b>	Inserción directa
<b>Argumentos:</b>	$A$ : vector $A[l, \dots, r]$ , $l$ : índice de la primera posición del vector, $r$ : índice de la última posición del vector

---

```

void insercion_directa(int *A, int l, int r) {
    int i, j, aux;

    for (i=l+1; i<=r; i++) {
        aux=A[i];
        j=i;
        while ((j>l) && (A[j-1]>aux)) {
            A[j]=A[j-1];
            j--;
        }
        A[j]=aux;
    }
}

```

---

<sup>1</sup>Todos los algoritmos serán presentados utilizando el lenguaje de programación C

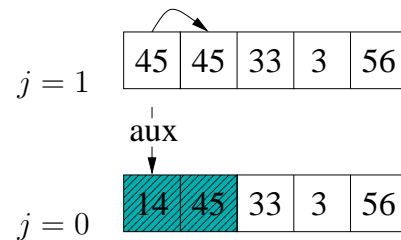
## Ejemplo del funcionamiento del algoritmo

llamada a la función: `insercion_directa(A,0,4)`

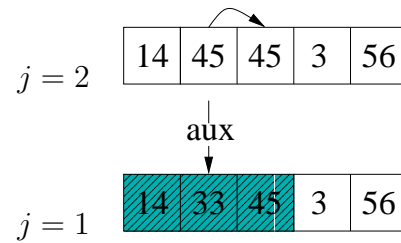
vector inicial A:

0	1	2	3	4
45	14	33	3	56

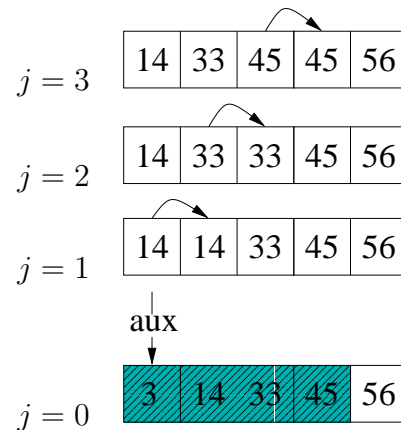
primera iteración ( $i=1, aux=14$ )



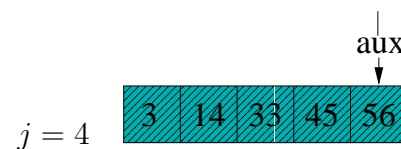
segunda iteración ( $i=2, aux=33$ )



tercera iteración ( $i=3, aux=3$ )



cuarta iteración ( $i=4, aux=56$ )



## Análisis de la eficiencia

El coste temporal del algoritmo dependerá, en gran medida, del número de veces que se repitan las comparaciones del bucle más interno (bucle *while*); es decir, de lo que cueste realizar la inserción de cada elemento en la parte ordenada.

### Caso peor

Dado un valor  $x$  y el vector  $A[1, \dots, n]$ , tal que  $A[i] = x$ ; el coste de la inserción de  $x$  en la parte ordenada del vector ( $A[1, \dots, i-1]$ ) será máximo cuando  $x$  sea menor que  $A[j]$  para todo  $j$  entre 1 e  $i-1$ ; es decir, cuando  $x$  sea menor que todos los valores almacenados en la parte ordenada. En este caso, se realizarán el número máximo de comparaciones:  $i-1$ . Si el vector  $A$  estuviera inicialmente ordenado en orden decreciente (sin repeticiones), nos encontraríamos ante el peor caso posible ya que para todo valor de  $i$  entre 2 y  $n$  el coste de la inserción sería el máximo ( $i-1$ ). Por lo tanto, el coste en el caso peor vendrá determinado por la expresión:

$$\sum_{i=2}^n (i-1) = \frac{n(n-1)}{2} \in \Theta(n^2)$$

### Caso mejor

Siguiendo con el mismo razonamiento que hemos utilizado en la estimación del caso peor, dado un valor  $x$  y el vector  $A[1, \dots, n]$ , tal que  $A[i] = x$ ; el coste de la inserción de  $x$  en la parte ordenada del vector ( $A[1, \dots, i-1]$ ) será mínima cuando  $x$  sea mayor o igual que  $A[i-1]$ ; es decir, cuando  $x$  sea mayor o igual que el máximo de los valores almacenados en la parte ordenada. Si el vector  $A$  estuviera inicialmente ordenado en orden creciente, nos encontraríamos ante el mejor caso ya que para todo valor de  $i$  entre 2 y  $n$  únicamente será necesaria realizar una comparación para realizar la inserción. Por lo tanto, el coste en el caso mejor será:

$$\sum_{i=2}^n 1 = n-1 \in \Theta(n)$$

### Caso medio

Para determinar el coste medio del algoritmo, supondremos que los  $n$  elementos a ordenar son distintos. En este caso, dado un valor de  $i$ , tal que  $A[i] = x$ ,

$x$  puede situarse con igual probabilidad ( $1/i$ ), en cualquier posición de la parte que quedará ordenada tras su inserción ( $A[1, \dots, i]$ ); hay que hacer la salvedad de que la probabilidad de que se realicen  $i - 1$  comparaciones será  $2/i$ , ya que esto sucede tanto si  $x < A[1]$  como si  $A[1] \leq x < A[2]$ . Considerando, pues, que el coste medio de la inserción de cada elemento  $i$  en la parte ordenada, es:

$$c_i = \frac{1}{i} \left( 2(i-1) + \sum_{k=1}^{i-2} k \right) = \frac{(i-1)(i+2)}{2i} = \frac{i+1}{2} - \frac{1}{i}$$

El coste medio del algoritmo para ordenar  $n$  elementos vendrá determinado por la expresión:

$$\sum_{i=2}^n c_i = \sum_{i=2}^n \left( \frac{i+1}{2} - \frac{1}{i} \right) = \frac{n^2 + 3n}{4} - H_n \in \Theta(n^2)$$

donde  $H_n = \sum_{i=1}^n \frac{1}{i} \in \Theta(\log n)$

El coste medio del algoritmo pertenece a  $\Theta(n^2)$  ya que el término  $H_n$  es despreciable con respecto al término dominante  $\frac{n^2}{4}$ . El coste medio del algoritmo es, pues, la mitad de su coste en el peor de los casos, pero su coste sigue perteneciendo a  $\Theta(n^2)$ .

Una vez analizados cada uno de los casos podemos concluir que el **coste temporal del algoritmo “inserción directa”** es:

$$\Omega(n) \ O(n^2)$$

Se puede obtener una información más detallada sobre el algoritmo “inserción directa” en los libros [Ferri, Brassard, Sedgewick].

### 4.2.2. Selección directa

**Estrategia:** Asignar a cada posición  $i$  del vector, el  $i$ -ésimo menor elemento almacenado en el vector.

**Funcionamiento:**

Dado un vector  $A[1, \dots, N]$

1. Inicialmente se *selecciona* el valor mínimo almacenado en el vector y se coloca en la primera posición; es decir, asignamos el 1-ésimo menor elemento a la posición 1.

2. A continuación, se *selecciona* el valor mínimo almacenado en la *subsecuencia*  $A[2, \dots, N]$  y se coloca en la segunda posición; es decir, asignamos el 2-ésimo menor elemento a la posición 2.
3. Siguiendo este procedimiento se recorren todas las posiciones  $i$  del vector, asignando a cada una de ellas el elemento que le corresponde: el  $i$ -ésimo menor.

A continuación se presenta una versión del algoritmo “selección directa”:

---

<b>Algoritmo:</b>	Selección directa
<b>Argumentos:</b>	$A$ : vector $A[l, \dots, r]$ , $l$ : índice de la primera posición del vector, $r$ : índice de la última posición del vector

---

```
void seleccion_directa(int *A, int l, int r) {
    int i, j, min, aux;

    for (i=l; i<r; i++) {
        min=i;
        for (j=i+1; j<=r; j++)
            if (A[j]<A[min])
                min=j;
        aux=A[i];
        A[i]=A[min];
        A[min]=aux;
    }
}
```

## Ejemplo del funcionamiento del algoritmo

### Análisis de la eficiencia

Dado un vector de talla  $n$  ( $A[1, \dots, n]$ ), el coste temporal del algoritmo vendrá determinado por el número de veces que se realice la comparación del bucle interno:  $n - i$ . Debido a que esta instrucción se encuentra en el interior de dos bucles *for*, no habrá que distinguir entre caso mejor y peor, ya que el número de comparaciones será, en cualquier caso (independientemente del estado original del vector):

$$\sum_{i=1}^{n-1} n - i = \frac{n(n-1)}{2} \in \Theta(n^2)$$

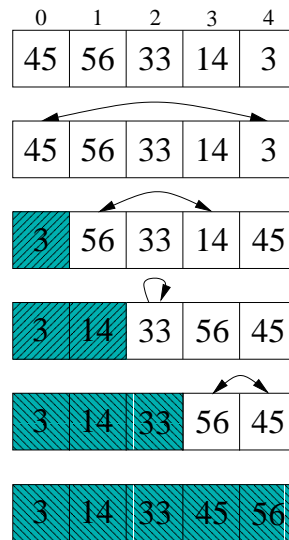


Figura 4.2: Ejemplo del funcionamiento del algoritmo *selección directa*. La parte sombreada representa la parte ordenada en el vector.

Por lo tanto, el **coste temporal** del algoritmo “selección directa” es:

$$\Theta(n^2)$$

Se puede obtener una información más detallada sobre el algoritmo “selección directa” en los libros [Ferri, Brassard, Sedgewick].

### 4.2.3. Ordenación por mezcla o fusión: Mergesort

#### Estrategia:

Dado un vector  $A[1, \dots, N]$

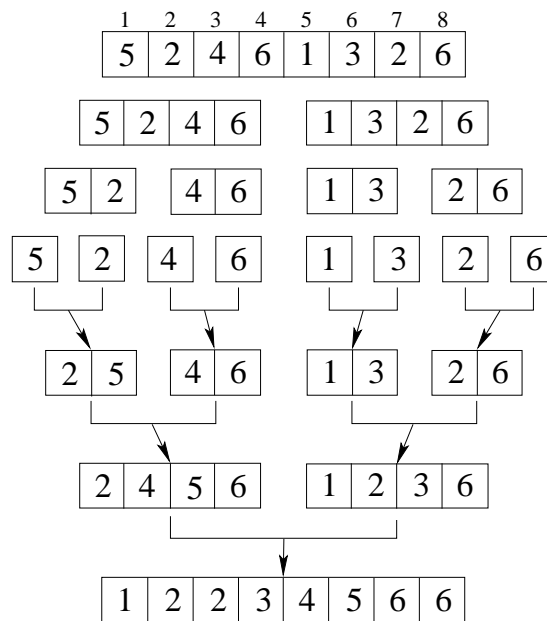
1. Se divide el vector en dos *subvectores*:  $A[1, \dots, \frac{N}{2}]$   $A[\frac{N}{2} + 1, \dots, N]$ .
2. Se ordenan independientemente cada uno de los *subvectores*.
3. Se *mezclan* los dos *subvectores* ordenados de manera que se obtenga un nuevo vector ordenado de talla  $N$ .

#### Funcionamiento:

El planteamiento del algoritmo sigue un esquema recursivo “divide y vencerás”. Para mostrar claramente el funcionamiento del algoritmo, supondremos que la talla del vector  $N$  es una potencia de 2.

1. Dividimos repetidamente el vector de talla  $N$  en *subvectores* de talla la mitad hasta obtener *subvectores* de talla 1 ( $v_1, v_2, \dots, v_N$ ).
2. Mezclamos los *subvectores* de talla 1 por pares ( $v_1, v_2$ ), ( $v_3, v_4$ ),  $\dots$ , ( $v_{n-1}, v_N$ ) obteniendo *subvectores* ordenados de talla 2 (el doble). Repetimos este proceso, obteniendo cada vez *subvectores* ordenados de talla el doble que la iteración anterior, hasta que se obtiene el vector original ordenado.

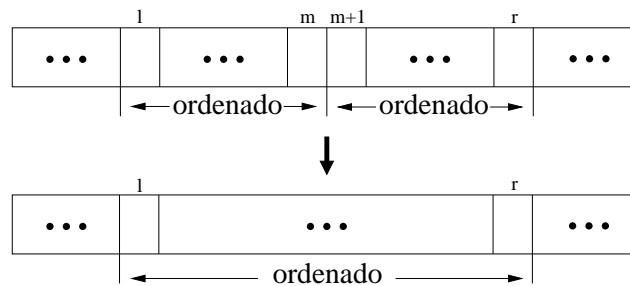
### Ejemplo:



Siguiendo el esquema “divide y vencerás”, el algoritmo primero divide el problema en *subproblemas* de talla menor y, posteriormente, combina las soluciones de los *subproblemas* para obtener la solución al problema original. La combinación de cada una de las soluciones de los *subproblemas*, se llevará a cabo mediante un algoritmo de mezcla o fusión que veremos a continuación.

### Algoritmo de mezcla o fusión

**Problema:** Mezclar ordenadamente dos vectores (*subsecuencias*) ordenados.



A continuación se muestra una versión del algoritmo de mezcla; en esta versión se utiliza la función *crear\_vector* que reserva el espacio de memoria necesario para el vector auxiliar *B*. El código para esta función de reserva de memoria es:

```
int *crear_vector(int n) {
    int *aux;

    aux = (int *) malloc(n*sizeof(int));
    return(aux);
}
```



---

**Algoritmo:** Merge  
**Argumentos:**  $A$ : vector  $A[l, \dots, r]$ ,  
 $l$ : índice de la primera posición del vector,  
 $m$ : índice de la posición que separa las dos partes ordenadas  
 $r$ : índice de la última posición del vector

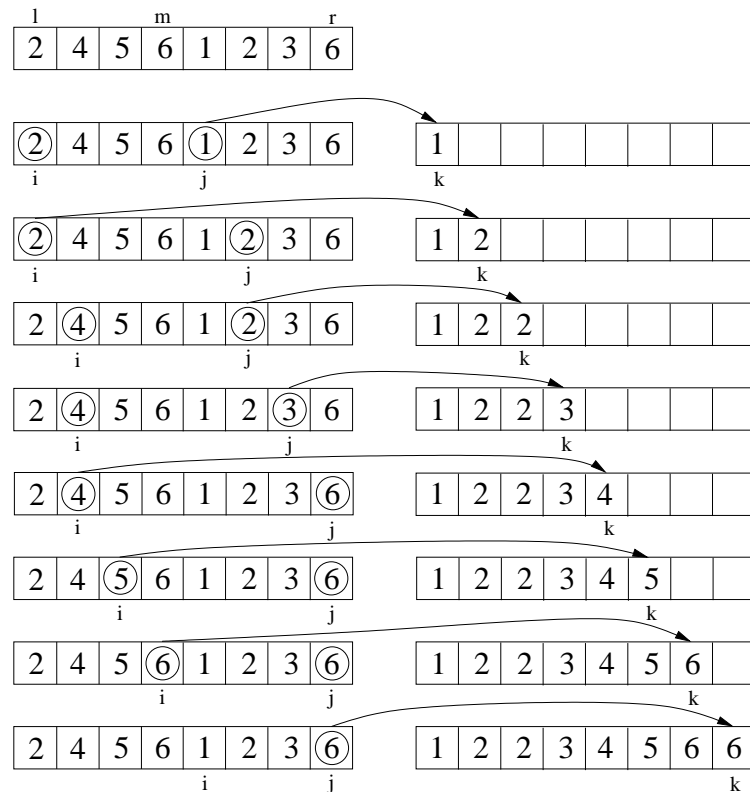
---

```
void merge(int *A, int l, int m, int r) {  
    int i, j, k, *B;  
  
    B = crea_vector(r-l+1);  
    i = l;    j = m + 1;    k = 0;  
  
    while ( (i<=m) && (j<=r) ) {  
        if (A[i] <= A[j]) {  
            B[k] = A[i];  
            i++;  
        } else {  
            B[k] = A[j];  
            j++;  
        }  
        k++;  
    }  
  
    while (i<=m) {  
        B[k] = A[i];  
        i++;  
        k++;  
    }  
  
    while (j<=r) {  
        B[k] = A[j];  
        j++;  
        k++;  
    }  
  
    for (i=l; i<=r; i++)  
        A[i]=B[i-l];  
    free(B);  
}
```

La estrategia que sigue el algoritmo es la de ir colocando sucesivamente en un vector auxiliar, el menor elemento de los que quedan por mezclar. Para ello, el

algoritmo utiliza un índice  $i$  que recorre secuencialmente la primera *subsecuencia* ordenada ( $A[l, \dots, m]$ ), y un índice  $j$  que recorre la segunda ( $A[m + 1, \dots, r]$ ); además utiliza un índice  $k$  para ir *mezclando* ordenadamente los valores en el vector auxiliar  $B$ . Inicialmente (primer bucle *while*), mientras quedan valores de la primera y segunda *subsecuencia* por mezclar, va recorriendo cada *subsecuencia*, de tal forma, que para cada nueva posición ( $i, j$ ), compara los valores  $A[i], A[j]$ , colocando en el vector auxiliar  $B$  el menor de los dos, o en caso de igualdad, el que se encuentra en la posición  $i$ ; a continuación, actualiza el índice correspondiente ( $i = i + 1$  o  $j = j + 1$ ) para que indique la posición del próximo valor a comparar. Cuando ya ha recorrido totalmente una de las dos *subsecuencias*, copia secuencialmente en  $B$  los elementos de la *subsecuencia* que faltan por mezclar (segundo o tercer bucle *while*). Finalmente, copia el resultado de la mezcla (el vector auxiliar  $B$ ) en el vector original  $A$  (bucle *for*).

### Ejemplo del funcionamiento del algoritmo de mezcla



El algoritmo de mezcla necesita un espacio de memoria adicional (de talla  $n = r - l + 1$ ) para almacenar el resultado en un vector auxiliar. Existen otras versiones del algoritmo que no necesitan el vector auxiliar, sino que realizan todos

los cambios sobre el vector original  $A$ ; sin embargo, el número de intercambios necesarios es tan elevado que no resultan muy apropiadas.

### Eficiencia del algoritmo de mezcla

El algoritmo de mezcla tiene un coste temporal proporcional a la suma de las tallas de las dos *subsecuencias*. Por lo tanto, si  $n = r - l + 1$  el **coste temporal del algoritmo de mezcla**  $\in \Theta(n)$ .

### Algoritmo *Mergesort*

A continuación se presenta el algoritmo “Mergesort”:

---

<b>Algoritmo:</b>	Mergesort
<b>Argumentos:</b>	$A$ : vector $A[l, \dots, r]$ , $l$ : índice de la primera posición del vector, $r$ : índice de la última posición del vector

---

```
void mergesort(int *A, int l, int r) {
    int m;

    if (l < r) {
        m = (int) ((l+r)/2);
        mergesort(A, l, m);
        mergesort(A, m+1, r);
        merge(A, l, m, r);
    }
}
```

En la figura 4.3 se muestra un ejemplo del funcionamiento del algoritmo utilizando una estructura arbórea, que ayuda a visualizar las distintas llamadas recursivas que realiza el algoritmo; además, se muestra cómo se combinan los resultados obtenidos en cada una de ellas para obtener la solución al problema original.

Cada nodo del árbol representa una llamada a la función *mergesort*. En cada uno de estos nodos se representa el vector original que recibe el algoritmo, y el vector resultado que produce (sombreado). También se indica, entre paréntesis, el orden en que se ejecutan cada una de las llamadas y los parámetros con que se invocan.

## Ejemplo de funcionamiento del algoritmo

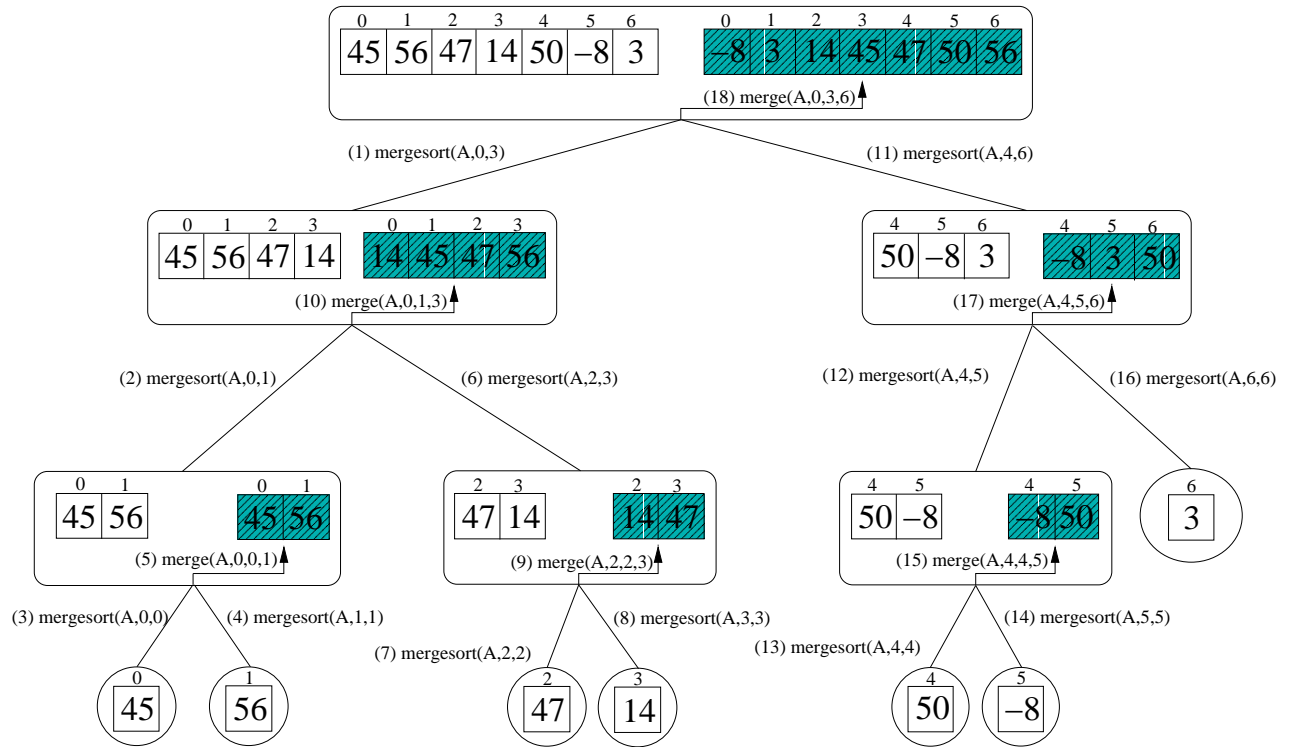


Figura 4.3: Ejemplo del funcionamiento del algoritmo *mergesort*. La parte sombreada representa el vector ordenado que se obtiene en cada llamada.

Respecto al funcionamiento del algoritmo, cabe destacar que la talla de la pila de recursividad tendrá una altura máxima logarítmica.

## Análisis de la eficiencia

El comportamiento del algoritmo será independiente del orden inicial en el que se encuentren los datos en el vector. Por lo tanto, el coste temporal puede aproximarse con la siguiente relación de recurrencia (supondremos por simplicidad que la talla del problema  $n$ , es una potencia de 2):

$$T(n) = \begin{cases} c_1 & n = 1 \\ 2T(\frac{n}{2}) + c_2n + c_3 & n > 1 \end{cases}$$

Resolviendo por sustitución:

$$\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + c_2n + c_3 \\
&= 2\left(2T\left(\frac{n}{2^2}\right) + c_2\frac{n}{2} + c_3\right) + c_2n + c_3 \\
&= 2^2T\left(\frac{n}{2^2}\right) + 2c_2n + c_3(2 + 1) \\
&= 2^3T\left(\frac{n}{2^3}\right) + 3c_2n + c_3(2^2 + 2 + 1) \\
&= p \text{ iteraciones} \dots \\
&= 2^pT\left(\frac{n}{2^p}\right) + pc_2n + c_3(2^{p-1} + \dots + 2^2 + 2 + 1) \\
(p = \log_2 n) &= nc_1 + c_2n \log_2 n + c_3(n - 1) \in \Theta(n \log n)
\end{aligned}$$

Por lo tanto, el **coste temporal del algoritmo “mergesort”** es:

$$\Theta(n \log n)$$

### Problema del balanceo

La descomposición del problema original en *subproblemas* de, aproximadamente, el mismo tamaño (balanceo), es fundamental (como ya se dijo) para la eficiencia de los algoritmos que utilizan las técnicas “divide y vencerás”. Para comprender por qué, veamos qué sucede si se descompone el problema original, de talla  $n$ , en un *subproblema* de talla  $n - 1$  y en otro de talla 1. En este caso, el coste temporal del algoritmo respondería a la siguiente relación de recurrencia:

$$T(n) = \begin{cases} c_1 & n = 1 \\ T(n - 1) + c_2n + c_3 & n > 1 \end{cases}$$

Resolviendo por sustitución:

$$\begin{aligned}
T(n) &= T(n - 1) + c_2n + c_3 \\
&= T(n - 2) + c_2n + c_2(n - 1) + 2c_3 \\
&= T(n - 3) + c_2n + c_2(n - 1) + c_2(n - 2) + 3c_3 \\
&= \dots \\
&= T(1) + c_2n + c_2(n - 1) + \dots + 2c_2 + (n - 1)c_3 \\
&= c_1 + (n - 1)c_3 + c_2 \sum_{i=2}^n i \\
&= c_1 + (n - 1)c_3 + c_2(n + 2) \frac{(n - 1)}{2} \in \Theta(n^2)
\end{aligned}$$

Como se puede observar, el algoritmo *mergesort* dejaría de ser eficiente si no realizara las divisiones de la manera más equilibrada posible; pasaría a tener un coste cuadrático, al igual que los métodos de ordenación directos.

### Posibles mejoras del algoritmo

- Tratando los *subproblemas* que estén por debajo de cierta talla, con un método de ordenación que se comporte bien para tallas pequeñas (por ejemplo, los métodos directos: “*inserción*”, “*selección*”), evitaremos realizar cierto número de llamadas recursivas consiguiendo mejorar el comportamiento temporal del algoritmo.
- Se puede evitar el tiempo que se dedica en el algoritmo “*merge*” a copiar el vector auxiliar (resultado) en el vector original, realizando las llamadas recursivas de forma que se alternen, en cada nivel, el vector auxiliar y el original.

Se puede obtener una información más detallada sobre el algoritmo “Merge-sort” en los libros [Ferri, Brassard, Sedgewick, Horowitz].

### 4.2.4. Algoritmo por partición: Quicksort

El algoritmo *Quicksort* fue propuesto por C. A. R. Hoare en el año 1960, y ha resultado ser el algoritmo de ordenación más eficiente (en el caso medio) que se conoce. Veamos en qué consiste:

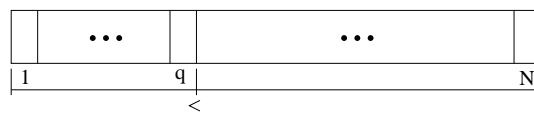
#### Estrategia:

Dado un vector  $A[1, \dots, N]$

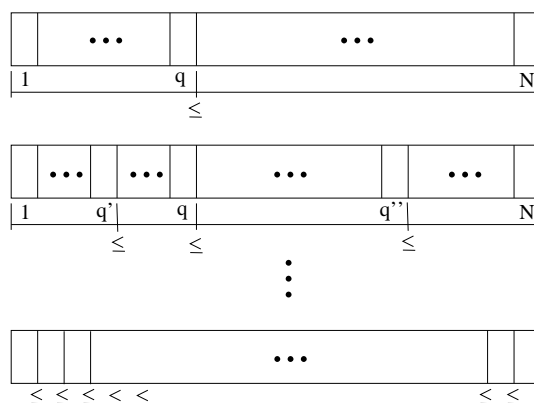
1. Se *divide* el vector  $A$  en dos *subvectores* no vacíos:  $A[1, \dots, q]$  y  $A[q + 1, \dots, N]$ , tal que todo elemento de  $A[1, \dots, q]$  sea menor o igual que todo elemento de  $A[q + 1, \dots, N]$ .
2. Se ordenan independientemente cada uno de los dos *subvectores* obteniendo el vector original ordenado.

#### Funcionamiento:

1. Se elige como *pivote* uno de los elementos del vector  $x$ : se permutan los elementos del mismo de manera que los que son menores que el pivote quedan en la parte izquierda ( $A[1, \dots, q]$ ), y los que son mayores en la parte derecha ( $A[q + 1, \dots, N]$ ) (**partición**). Una vez realizada la partición se cumplirá que todos los elementos del *subvector*  $A[1, \dots, q]$  serán menores o iguales que todos los elementos de  $A[q + 1, \dots, N]$ .



2. Aplicamos el mismo procedimiento repetidamente a cada uno de los *sub-vectores* hasta obtener *subvectores* de talla 1, momento en el cual se habrá obtenido la ordenación del vector original (ver figura en la siguiente página).



El funcionamiento del algoritmo *Quicksort* se basa en cómo se realizan las subdivisiones del problema original en *subproblemas* de talla menor. Tanto es así, que únicamente realizando subdivisiones sobre el vector original obtenemos, sin necesidad de combinar las soluciones de los *subproblemas*, su ordenación. Las subdivisiones sucesivas que realiza el algoritmo sobre el vector serán llevadas a cabo por un algoritmo de partición que veremos a continuación.

### Algoritmo de partición

Como ya se vio con el algoritmo *Mergesort*, los algoritmos que utilizan la técnica “divide y vencerás” basan su eficiencia en subdividir el problema original en *subproblemas* de tallas equilibradas (similares). Por esta razón, la eficiencia del algoritmo *Quicksort* dependerá totalmente del modo en que se realicen las subdivisiones en el vector: será importantísimo que el algoritmo de partición realice las subdivisiones de la forma más equilibrada posible.

El algoritmo de partición, como ya se adelantó, realizará la división de un vector en dos *subvectores* utilizando, para ello, un valor  $x$  denominado “pivote”. De esta forma, todo elemento contenido en el primer *subvector* será menor o igual que  $x$ , y todo elemento contenido en el segundo será mayor o igual que  $x$ . Por lo tanto, es obvio, que la talla de ambos *subvectores* estará más o menos equilibrada dependiendo del valor escogido como pivote; es decir, si  $x$  es el menor o mayor

elemento almacenado en el vector nos encontraremos ante la partición más desequilibrada posible: un *subvector* contendrá a  $x$  y el otro al resto, mientras que si  $x$  es la mediana nos encontraremos ante la partición más equilibrada posible.

Debido a que un algoritmo que encuentre la mediana es demasiado costoso, una alternativa simple es elegir un elemento cualquiera del vector con el consiguiente riesgo de que los *subvectores* resultantes no estén equilibrados.

**Problema:** Dividir un vector en dos partes de manera que todos los elementos de la primera parte sean menores o iguales que los de la segunda.

**Funcionamiento:** Dado un vector  $A[1, \dots, N]$

1. Se elige arbitrariamente el primer elemento del vector como pivote  $x = A[1]$ .
2. Utilizando un índice  $i$ , se recorre incrementalmente desde la posición 1 una región  $A[1, \dots, i]$  hasta encontrar un elemento  $A[i] \geq x$ .
3. Utilizando un índice  $j$ , se recorre decrementalmente desde la posición  $N$  una región  $A[j, \dots, N]$  hasta encontrar un elemento  $A[j] \leq x$ .
4. Se intercambian los valores  $A[i]$  y  $A[j]$ .
5. Mientras  $i < j$ , repetimos los pasos (2,3,4) iniciando el recorrido incremental y decremental desde las últimas posiciones  $i$  y  $j$ , respectivamente.
6. El índice  $j$ , donde  $1 \leq j < N$ , indicará la posición que separa las dos partes del vector  $A[1, \dots, j]$  y  $A[j+1, \dots, N]$ , tal que todo elemento de  $A[1, \dots, j]$  será menor o igual que todo elemento de  $A[j+1, \dots, N]$ .

A continuación, se presenta el algoritmo:



---

<b>Algoritmo:</b>	Partition
<b>Argumentos:</b>	$A$ : vector $A[l, \dots, r]$ , $l$ : índice de la primera posición del vector, $r$ : índice de la última posición del vector

---

```

int partition(int *A, int l, int r) {
    int i,j,x,aux;

    i=l-1;  j=r+1;  x=A[l];
    while(i<j) {

        do {
            j--;
        } while (A[j]>x);

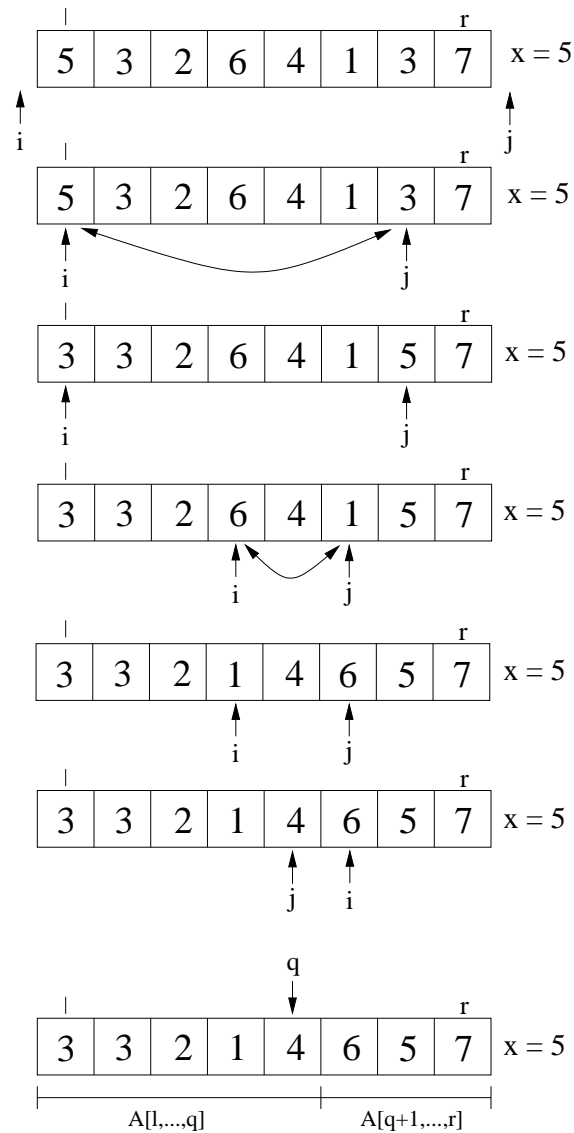
        do {
            i++;
        } while (A[i]<x);

        if (i<j) {
            aux=A[i];
            A[i]=A[j];
            A[j]=aux;
        }

    }
    return(j);
}

```

### Ejemplo de funcionamiento del algoritmo de partición



### Eficiencia del algoritmo de partición

El coste temporal del algoritmo de partición es  $\Theta(n)$ , donde  $n = r - l + 1$ .

### Algoritmo *Quicksort*

A continuación, se presenta el algoritmo *Quicksort*:

---

<b>Algoritmo:</b>	Quicksort
<b>Argumentos:</b>	$A$ : vector $A[l, \dots, r]$ , $l$ : índice de la primera posición del vector, $r$ : índice de la última posición del vector

---

```
void quicksort(int *A, int l, int r) {  
    int q;  
  
    if (l < r) {  
        q = partition(A, l, r);  
        quicksort(A, l, q);  
        quicksort(A, q+1, r);  
    }  
}
```

### Ejemplo del funcionamiento del algoritmo

En la figura 4.4 se muestra un ejemplo del funcionamiento del algoritmo *Quicksort*. Entre paréntesis se indica el orden en que se realizan las llamadas, tanto al algoritmo de partición como las recursivas.

En la figura 4.5 se puede observar una traza del funcionamiento del algoritmo *Quicksort*. Es una manera alternativa a la de la figura 4.4 de analizar la secuencia de operaciones de un algoritmo para un caso determinado.

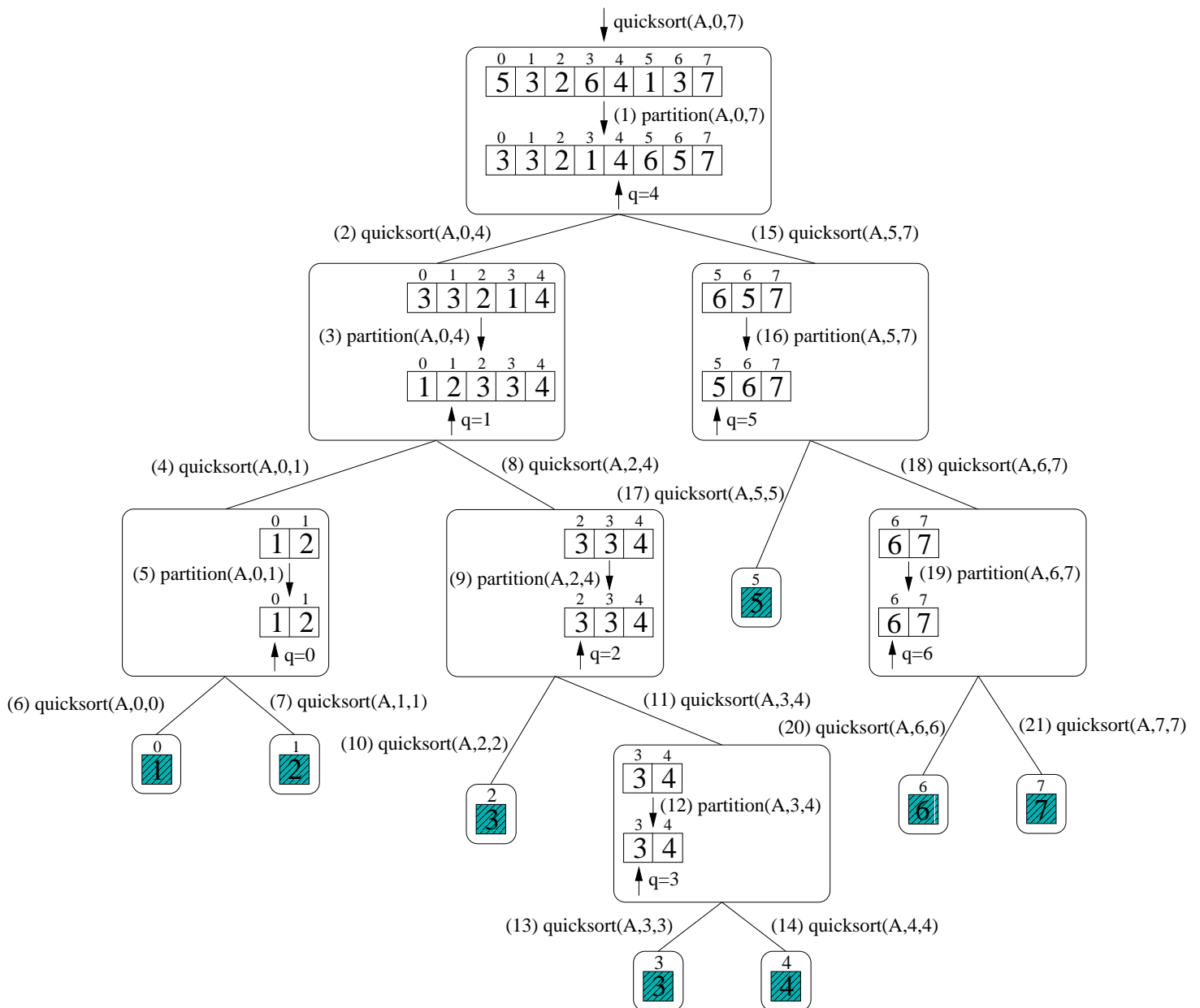


Figura 4.4: Ejemplo del funcionamiento del algoritmo *quicksort*.

### Análisis de la eficiencia

El coste del algoritmo dependerá directamente de que las tallas de los *subvec-*  
*tores* que se generen en cada partición sean equilibradas o no lo sean. Este hecho  
 dependerá del elemento elegido como “pivote” en cada una de las particiones. Por  
 lo tanto, para evaluar el coste del algoritmo habrá que diferenciar los siguientes

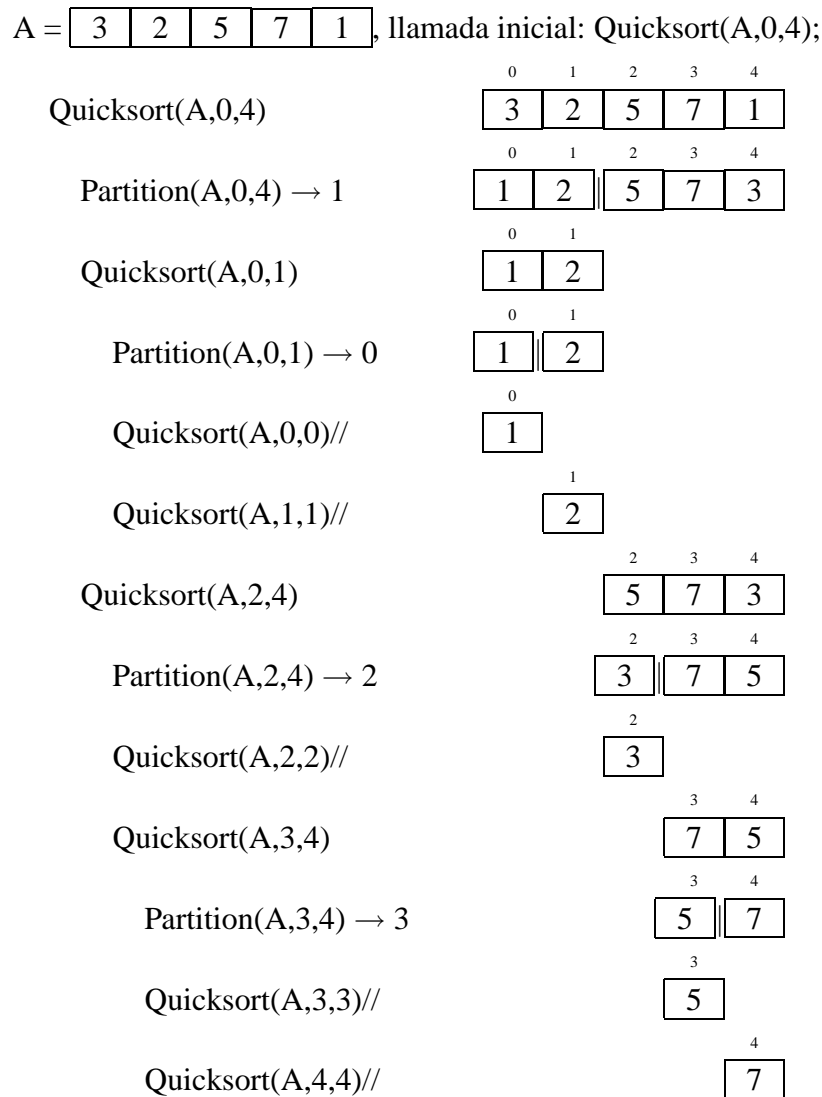


Figura 4.5: Traza ejemplo del algoritmo *quicksort*.

casos:

### Caso peor

La partición más desequilibrada posible se producirá cuando se genere un *sub-vector* de talla 1 y otro de talla  $N - 1$ . Por lo tanto, cuando todas las particiones que realice el algoritmo sean de esta índole, nos encontraremos ante el peor caso. Este caso ocurre, por ejemplo, cuando inicialmente el vector presenta una ordenación creciente (sin elementos repetidos), ya que, para todo partición, el valor

elegido como “pivote” será el menor. En este caso, el comportamiento del algoritmo responderá a la siguiente relación de recurrencia:

$$T(n) = \begin{cases} c_1 & n = 1 \\ T(n-1) + c_2n + c_3 & n > 1 \end{cases}$$

Por lo tanto,  $T(n) \in \Theta(n^2)$  (ver demostración en la sección 4.2.3).

### Caso mejor

El caso mejor se producirá cuando, en todas las particiones que realice el algoritmo, se generen dos *subvectores* de talla la mitad. En este caso, el comportamiento del algoritmo responderá a la siguiente relación de recurrencia:

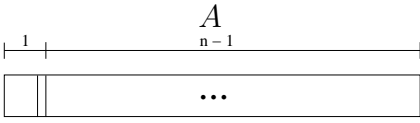
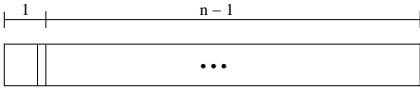
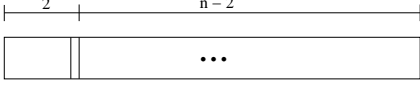
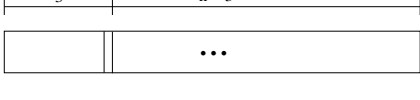
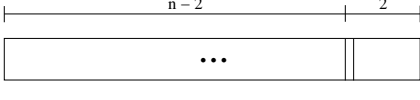
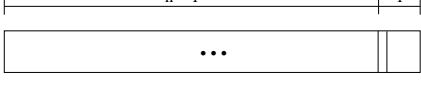
$$T(n) = \begin{cases} c_1 & n = 1 \\ 2T(\frac{n}{2}) + c_2n + c_3 & n > 1 \end{cases}$$

Por lo tanto,  $T(n) \in \Theta(n \log n)$  (ver demostración en la sección 4.2.3).

### Caso medio

Para evaluar el coste medio, asumiremos que todos los elementos del vector  $A[1, \dots, n]$  son distintos. Si no fuera el caso, el coste del algoritmo no variaría, pero el análisis sería más complicado del que se presenta a continuación:

Como ya se ha insistido anteriormente, las tallas de los *subvectores* que se generan en las diferentes particiones, serán más o menos equilibradas dependiendo del valor elegido como “pivote”. Debido a que, en general, desconocemos el valor que se va a elegir como pivote en cada una de las particiones, y, por lo tanto, las tallas de los *subproblemas* originados en cada partición, tendremos que asumir que con igual probabilidad ( $\frac{1}{n}$ ) se puede dar una de todas las posibles particiones:

	Talla subproblemas	Probabilidad
	1 $n - 1$	$\frac{1}{n}$
	1 $n - 1$	$\frac{1}{n}$
	2 $n - 2$	$\frac{1}{n}$
	3 $n - 3$	$\frac{1}{n}$
...	...	...
	$n - 2$ 2	$\frac{1}{n}$
	$n - 1$ 1	$\frac{1}{n}$

Únicamente, la probabilidad de que la partición generada sea  $A[1]$  y  $A[2, \dots, N]$ , es doblemente probable, ya que esto se cumple tanto si el valor elegido como “pivot” es el menor del vector, como si es el segundo menor.

Para evaluar el coste medio, debemos evaluar el coste que supondría ordenar cada uno de los posibles *subproblemas*, y ponderar el coste obtenido, para cada *subproblema*, por la probabilidad de que ocurra. Siguiendo este razonamiento, el coste medio de ordenación de un vector de talla  $n$  puede ser determinado por la siguiente expresión:

$$T(n) = \frac{1}{n} \left( T(1) + T(n-1) + \sum_{q=1}^{n-1} (T(q) + T(n-q)) \right) + \Theta(n) \quad (4.1)$$

Dado que  $T(1) = \Theta(1)$  y, en el caso peor,  $T(n-1) = O(n^2)$ , podemos deducir que:

$$\begin{aligned} \frac{1}{n} (T(1) + T(n-1)) &= \frac{1}{n} (\Theta(1) + O(n^2)) \\ &= O(n) \end{aligned}$$

Por lo tanto, y dado que, en la expresión 4.1, el término  $\Theta(n)$  puede absorber la expresión  $\frac{1}{n}(T(1) + T(n-1))$ , la expresión 4.1 puede reescribirse como:

$$T(n) = \frac{1}{n} \sum_{q=1}^{n-1} (T(q) + T(n-q)) + \Theta(n) \quad (4.2)$$

Obsérvese que para  $k = 1, 2, \dots, n-1$  cada término  $T(k)$  ocurre, en el sumatorio, una vez como  $T(q)$  y otra como  $T(n-q)$ . Por lo tanto, la expresión 4.2 puede reescribirse como:

$$T(n) = \frac{2}{n} \sum_{k=1}^{n-1} T(k) + \Theta(n)$$

Una vez hemos obtenido la expresión que estima el coste medio del algoritmo, podemos concluir que el comportamiento medio del algoritmo responderá a la siguiente relación de recurrencia:

$$T(n) = \begin{cases} c_1 & n \leq 1 \\ \frac{2}{n} \sum_{k=1}^{n-1} T(k) + c_2 n & n > 1 \end{cases}$$

Se puede demostrar por inducción que  $\forall n > 1, T(n) \leq c_3 n \log_2 n$ , donde  $c_3 = 2c_2 + c_1$ , y por lo tanto  $T(n) \in O(n \log n)$  [Cormen, Brassard].

### Posibles mejoras del algoritmo

- Tratando los *subproblemas* que estén por debajo de cierta talla, con un método de ordenación que se comporte bien para tallas pequeñas (por ejemplo, los métodos directos: “*inserción*”, “*selección*”), evitaremos realizar cierto número de llamadas recursivas consiguiendo mejorar el comportamiento temporal del algoritmo.
- Elegir el valor utilizado como “pivote” de forma aleatoria entre todos los valores almacenados en el vector. Para ello, antes de realizar cada partición, se intercambiará el elemento utilizado como pivote  $A[l]$  con un elemento del *subvector*  $A[l, \dots, r]$  seleccionado al azar. De esta forma, cualquier elemento del vector tendrá la misma probabilidad de ser el “pivote” y, por ejemplo, se evitará que, en el caso de que el vector ya esté ordenado de forma creciente (sin repeticiones), se elija siempre como “pivote” el valor mínimo del vector, lo que ocasiona un mal comportamiento del algoritmo.



- Como ya ha sido indicado, para realizar particiones equilibradas, lo ideal sería utilizar como “pivote” la mediana del vector; sin embargo, obtener la mediana de un vector es demasiado costoso. Para solucionar este problema, se podría utilizar como “pivote” una *pseudomediana* que se obtendría de la siguiente forma: se seleccionan varios elementos al azar y se calcula la mediana de ellos. Una posibilidad sería, dado un *subvector*  $A[l, \dots, r]$ , seleccionar tres elementos:  $A[l]$ ,  $A[\frac{l+r}{2}]$ ,  $A[r]$ .

Se puede obtener una información más detallada sobre el algoritmo “quicksort” en los libros [Ferri, Cormen, Brassard, Sedgewick, Horowitz].

#### 4.2.5. Comparación empírica de los algoritmos

En la gráfica de la figura 4.6 se muestra el comportamiento, ante muestras reales, de cada uno de los algoritmos analizados. El eje de abscisas representa el tamaño de los vectores (talla de los problemas), y el eje de ordenadas indica el tiempo (en microsegundos) empleado en la ordenación de los vectores. Como puede observarse, en la práctica, el algoritmo *quicksort* es el que presenta un mejor comportamiento.

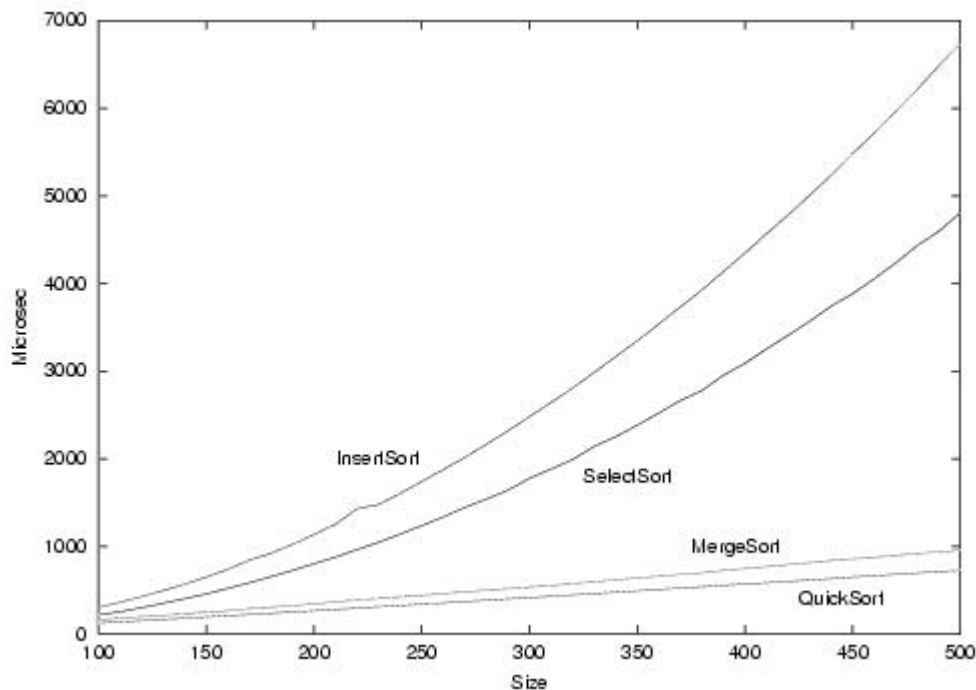


Figura 4.6: Gráfica comparativa de los diferentes algoritmos de ordenación.

### 4.3. Búsqueda del $k$ -ésimo menor elemento

Para la definición del problema que se presenta a continuación, se asumirá que los elementos del vector son distintos entre sí; aunque conceptualmente todo lo indicado se pueda extender al caso en que existan elementos repetidos.

**Problema:** El problema a resolver se denomina “*problema de la selección*”, y consiste en:

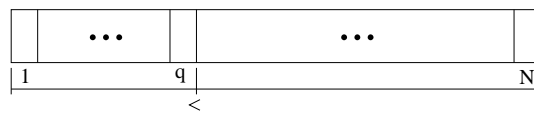
- Dado un conjunto de  $n$  enteros almacenados en un vector  $A[1, \dots, N]$ , encontrar el  $k$ -ésimo menor elemento; es decir, el elemento  $x \in A$  que es mayor que exactamente  $k - 1$  elementos.

Una primera solución a este problema se obtendría ordenando el vector y, posteriormente, seleccionando el elemento que ocupara la posición  $k$ . De esta forma, si se realizara la ordenación del vector, por ejemplo, mediante el algoritmo “*mergesort*”, se resolvería el problema con un coste temporal de  $\Theta(n \log n)$ . Sin embargo, este problema puede ser resuelto de una manera más eficiente utilizando la técnica de “divide y vencerás”.

**Estrategia:**

Dado un vector  $A[1, \dots, N]$

1. Se utiliza el algoritmo de partición (*partition*) utilizado por “*quicksort*” para dividir el vector en dos partes, de manera que todo elemento de la primera parte sea menor o igual que todo elemento de la segunda.



2. Debido a que, una vez ordenado el vector, el  $k$ -ésimo menor elemento ocupará la posición  $k$ , si se cumple que  $1 \leq k \leq q$  únicamente será necesario ordenar el *subvector*  $A[1, \dots, q]$ ; mientras que si, por el contrario,  $q < k \leq N$  únicamente será necesario ordenar el *subvector*  $A[q+1, \dots, N]$ .

El siguiente algoritmo resuelve el *problema de la selección* siguiendo la estrategia indicada:

---

<b>Algoritmo:</b>	Selección
<b>Argumentos:</b>	$A$ : vector $A[0, \dots, n-1]$ , $n$ : talla del vector, $k$ : valor de $k$

---

```

int seleccion(int *A, int n, int k) {
    int l, r, q;

    l = 0;    r = n-1;    k = k-1;
    while (l < r) {
        q = partition(A, l, r);
        if (k <= q)
            r = q;
        else
            l = q+1;
    }
    return (A[l]);
}

```

## Ejemplo de funcionamiento del algoritmo

En la figura 4.7 se muestra un ejemplo de cómo funciona el algoritmo de selección, dado un vector  $A = \{31, 23, 90, 0, 77, 52, 49, 87, 60, 15\}$ , y obteniendo el elemento 4-ésimo menor.

### 4.3.1. Análisis de la eficiencia

El coste temporal del algoritmo dependerá de la talla que tengan los sucesivos *subvectores*, en los que se va reduciendo el espacio de búsqueda después de cada partición.

#### Caso peor

En el caso en que, después de cada partición, el espacio de búsqueda se reduzca únicamente en un elemento, nos encontraremos en el peor caso. Este caso ocurrirá, por ejemplo, cuando los elementos del vector estén ordenados crecientemente (sin repeticiones), y el elemento buscado sea el  $n$ -ésimo menor.

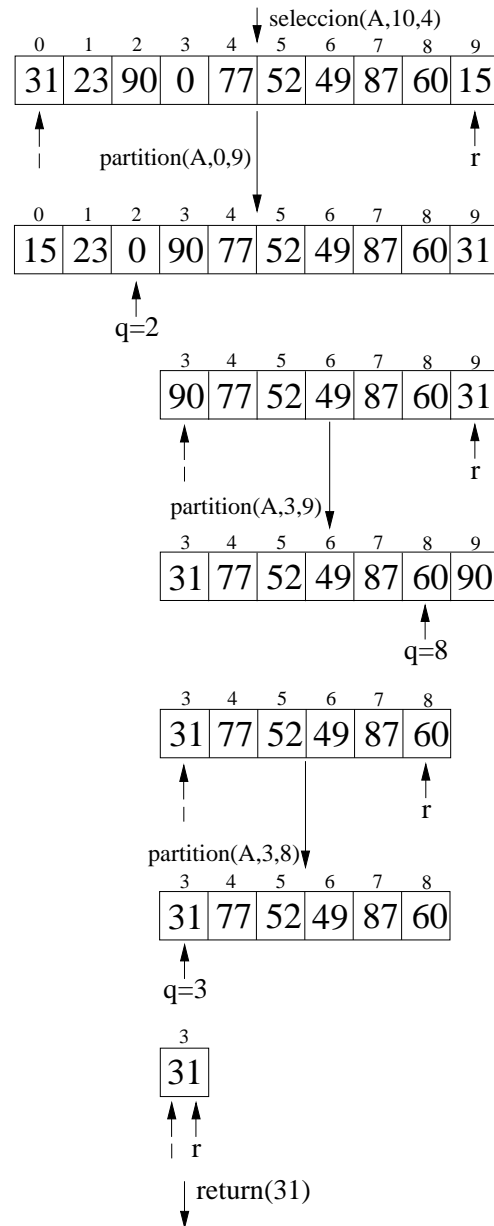


Figura 4.7: Ejemplo del funcionamiento del algoritmo “selección”.

En este caso, el comportamiento temporal del algoritmo responderá a la siguiente relación de recurrencia:

$$T(n) = \begin{cases} c_1 & n = 1 \\ T(n-1) + c_2n + c_3 & n > 1 \end{cases}$$

Por lo tanto,  $T(n) \in \Theta(n^2)$  (ver demostración en la sección 4.2.3).

## Caso mejor

El caso mejor se producirá cuando el algoritmo de partición divida siempre el vector en dos partes iguales hasta encontrar al elemento buscado. En este caso, el comportamiento temporal del algoritmo responderá a la siguiente relación de recurrencia:

$$T(n) = \begin{cases} c_1 & n = 1 \\ T(\frac{n}{2}) + c_2n + c_3 & n > 1 \end{cases}$$

Resolviendo por sustitución (supondremos por simplicidad que la talla del problema  $n$ , es una potencia de 2):

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + c_2n + c_3 \\ &= T\left(\frac{n}{2^2}\right) + c_2\frac{n}{2} + c_2n + 2c_3 \\ &= T\left(\frac{n}{2^2}\right) + \left(n + \frac{n}{2}\right)c_2 + 2c_3 \\ &= T\left(\frac{n}{2^3}\right) + c_2\frac{n}{2^2} + c_3 + \left(n + \frac{n}{2}\right)c_2 + 2c_3 \\ &= T\left(\frac{n}{2^3}\right) + c_2n\left(1 + \frac{1}{2} + \frac{1}{2^2}\right) + 3c_3 \\ &= p \text{ iteraciones} \dots \\ &= T\left(\frac{n}{2^p}\right) + c_2n\left(1 + \frac{1}{2} + \frac{1}{2^2} + \dots + \frac{1}{2^{p-1}}\right) + pc_3 \\ &= T\left(\frac{n}{2^p}\right) + c_2n\frac{2(2^p - 1)}{2^p} + pc_3 \\ (p = \log_2 n) \quad &= T(1) + c_2n\frac{2(2^{\log_2 n} - 1)}{2^{\log_2 n}} + \log_2 n \, c_3 \\ &= c_1 + 2c_2(n - 1) + c_3 \log_2 n \in O(n) \end{aligned}$$

## Caso medio

Como ya vimos al analizar el algoritmo “*quicksort*”, el algoritmo de partición, en el caso medio, genera *subvectores* más o menos equilibrados, lo que ocasiona que el comportamiento del algoritmo, en el caso medio, se aproxime al comportamiento en el caso mejor. Debido a que el algoritmo de selección basa su funcionamiento en el algoritmo de partición utilizado por *quicksort*, análogamente,

podemos concluir que el comportamiento del algoritmo de selección se aproximará, en término medio, a su coste en el caso mejor, es decir,  $O(n)$  [Cormen].

Se puede obtener una información más detallada sobre el algoritmo “selección” en los libros [Cormen, Sedgewick, Ferri, Horowitz].

## 4.4. Búsqueda binaria

El problema de la búsqueda binaria (o dicotómica) consiste básicamente en buscar la posición de un determinado elemento en un conjunto ordenado de elementos. Por ejemplo, si tenemos el vector

$$A = \begin{array}{cccc} 0 & 1 & 2 & 3 \\ \boxed{3} & \boxed{5} & \boxed{7} & \boxed{9} \end{array},$$

el resultado de llamar a una función `busqueda_binaria(A, 5)` nos devolvería la posición 1 puesto que el elemento 5 está en la segunda posición.

Podemos formular el problema de esta manera:

**Problema:** Dado un vector  $A$  ordenado por orden no decreciente de  $n$  enteros, buscar la posición correspondiente a un elemento  $x$ . Esto es, buscar un  $i$  tal que  $A[i]=x$ .

Podemos adoptar una solución directa que consistiría en recorrer el vector desde la primera posición hasta que encontremos el elemento buscado y devolver la posición de éste. Este es el clásico algoritmo de búsqueda secuencial, que como se puede observar claramente tiene un coste lineal con el tamaño del vector, esto es  $O(n)$ .

Vamos a proponer un esquema de búsqueda basado en divide y vencerás, después analizaremos su coste temporal y lo compararemos con la solución directa anterior.

La idea es que podemos aprovechar que el vector está ordenado para aplicar el esquema divide y vencerás. Si tomamos el valor del elemento que esté en la mitad del vector, primero podremos compararlo con el elemento que estamos buscando,  $x$ , si es igual, entonces ya hemos encontrado la posición resultado que será la mitad del vector. En el caso de que  $x$  sea menor que el elemento de la mitad del vector, quiere decir que  $x$  se encontrará en el subvector formado por la primera mitad del vector completo, entonces podemos aplicar el mismo criterio de búsqueda en ese subvector. En el caso de que  $x$  sea mayor que el elemento de la mitad del vector, quiere decir que  $x$  se encontrará en el subvector formado por la segunda mitad del vector completo, entonces podemos aplicar el mismo criterio de búsqueda en ese subvector. Así podemos ir buscando en subvectores cada vez más pequeños hasta que encontremos el elemento  $x$  en la mitad del vector o lleguemos a un subvector de tamaño 1 que necesariamente estará formado por  $x$  y podremos

devolver la posición correspondiente. En el caso de que el elemento  $x$  no estuviera en el vector  $A$ , deberíamos devolver un código de control que lo especifique, por ejemplo, devolver la posición -1.

El algoritmo para la búsqueda binaria en un vector mediante Divide y Vencerás es el siguiente:

---

<b>Algoritmo:</b>	Búsqueda_binaria
<b>Argumentos:</b>	$A$ : vector $A[l, \dots, r]$ , $l$ : índice de la primera posición del vector, $r$ : índice de la última posición del vector, $x$ : elemento del vector a buscar

---

```
int busqueda_binaria(int *A, int l, int r, int x) {
    int q;

    if (l==r)
        if (x==A[l]) return(l);
        else return(-1);
    else {
        q = (int) (l+r)/2;
        if (x == A[q]) return(q);
        else
            if (x < A[q]) return(busqueda_binaria(A,l,q-1,x));
            else return(busqueda_binaria(A,q+1,r,x));
    }
}
```

Un aspecto a tener en cuenta es que ésta es una codificación recursiva del algoritmo, pero también es sencillo escribir un algoritmo que aplique la metodología Divide y Vencerás descrita en el párrafo anterior con una codificación iterativa. La realización de este algoritmo queda propuesta para el alumno.

En la figura 4.8 podemos observar los sucesivos subvectores que emplea el algoritmo de búsqueda binaria para buscar la posición del elemento  $x=3$ .

#### 4.4.1. Eficiencia del algoritmo

Se puede comprobar que este algoritmo tiene un coste de  $O(\log n)$  dado un vector de talla  $n$ , puesto que en cada posible llamada recursiva, la talla del vector se reduce a la mitad,  $n/2$ .

Más detalladamente, si tenemos un vector de talla 1, ya tendremos la solución y la habremos obtenido con un coste constante. Si tenemos un vector de talla

**x=3**      llamada inicial: `busqueda_binaria(A,0,8,3);`

0	1	2	3	4	5	6	7	8
1	3	7	15	22	33	41	52	60

↓      q=4

0	1	2	3
1	3	7	15

↓      q=2

0	1
1	3

↓      q=0

1
3

q=1       $\implies$       A[1]=3       $\implies$       return(1)

Figura 4.8: Ejemplo de funcionamiento del algoritmo *busqueda\_binaria* para encontrar el elemento  $x=3$  en el vector  $A=\{1,3,7,15,22,33,41,52,60\}$ .

mayor que 1, lo dividiremos por la mitad y buscaremos o en la primera mitad o en la segunda mitad, pero no en las dos, con lo que se nos plantea la siguiente relación de recurrencia:

$$T(n) = \begin{cases} c_1 & n = 1 \\ T(\frac{n}{2}) + c_2 & n > 1 \end{cases}$$

Resolviendo por sustitución:



$$\begin{aligned}
T(n) &= T\left(\frac{n}{2}\right) + c_2 \\
&= T\left(\frac{n}{2^2}\right) + 2c_2 \\
&= T\left(\frac{n}{2^3}\right) + 3c_2 \\
&= p \text{ iteraciones} \dots \\
&= T\left(\frac{n}{2^p}\right) + pc_2 \\
(p > \log_2 n) &= c_1 + c_2 \log_2 n \in O(\log n)
\end{aligned}$$

## 4.5. Cálculo de la potencia de un número

**Problema:** Se quiere calcular:

$$N = b^n, n \geq 0$$

### 4.5.1. Algoritmo trivial

Una aproximación trivial a la resolución del problema, sería descomponerlo de la siguiente forma:

$$N = \underbrace{b \cdot b \cdot b \dots b \cdot b}_{n \text{ veces}}$$

El siguiente algoritmo resuelve el problema siguiendo esta estrategia trivial:

---

<b>Algoritmo:</b>	Potenciación
<b>Argumentos:</b>	$b$ : base, $n$ : exponente

---

```

int potencia(int b, int n) {
    int res, i;

    res = 1;
    for (i=1; i<=n; i++)
        res = res * b;
    return(res);
}

```

El coste temporal de este algoritmo es lineal en función del exponente y, por lo tanto, es  $O(n)$ .

### 4.5.2. Algoritmo “divide y vencerás”

Siguiendo la estrategia “divide y vencerás” el problema podría resolverse de la siguiente forma:

$$b^n = \begin{cases} b^{\frac{n}{2}} b^{\frac{n}{2}} & \text{si } n \text{ es par} \\ b^{\frac{n}{2}} b^{\frac{n}{2}} b & \text{si } n \text{ es impar} \end{cases}$$

El siguiente algoritmo resuelve el problema siguiendo esta estrategia:

---

<b>Algoritmo:</b>	Potenciación
<b>Argumentos:</b>	$b$ : base, $n$ : exponente

---

```
int potencia(int b, int n) {  
    int res, aux;  
  
    if (n == 0)  
        res = 1;  
    else {  
        aux = potencia(b, n/2);  
        if ((n % 2) == 0)  
            res = aux * aux;  
        else  
            res = aux * aux * b;  
    }  
    return(res);  
}
```

#### Eficiencia del algoritmo

La relación de recurrencia que define el comportamiento de este algoritmo es:

$$T(n) = \begin{cases} c_1 & n < 1 \\ T(\frac{n}{2}) + c_2 & n \geq 1 \end{cases}$$

Como se demostró en la sección [4.4.1](#), esta relación induce un coste para el algoritmo de  $O(\log n)$ .

## 4.6. Otros problemas

Otros problemas clásicos que tienen una solución más eficiente mediante Divide y Vencerás que con algoritmos directos o clásicos son los siguientes:

- Búsqueda binaria en 2 dimensiones (en una matriz en vez de en un vector).
- Búsqueda de una subcadena en cadenas de caracteres.
- Hallar el mínimo de una función cóncava.
- Búsqueda del mínimo y del máximo en un vector de enteros.
- Producto de enteros grandes.
- Multiplicación de matrices.

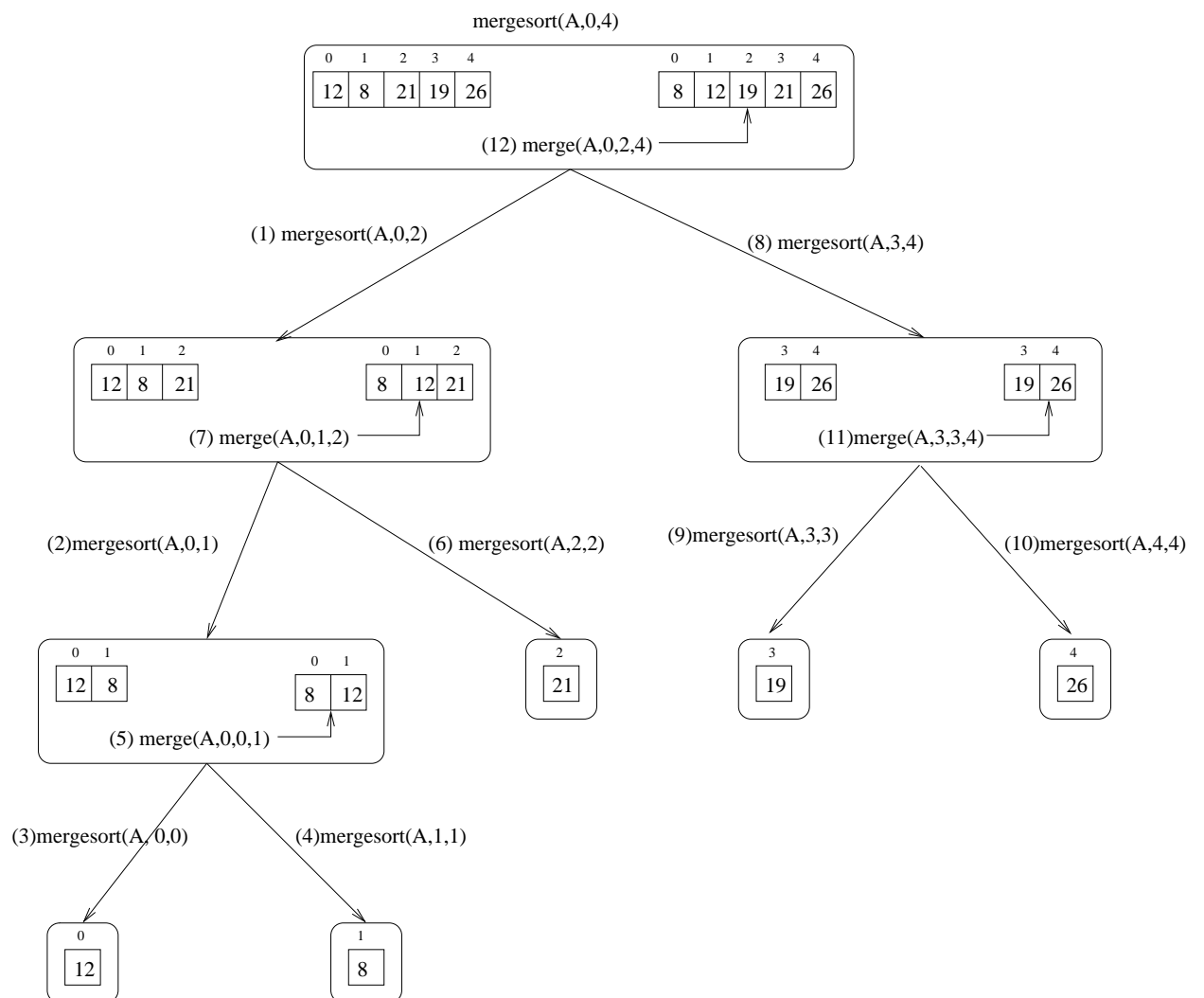
## 4.7. Ejercicios

### Ejercicio 1:

-Realiza una traza del algoritmo *mergesort* al aplicarlo sobre el vector  $A = \{12, 8, 21, 19, 26\}$ .

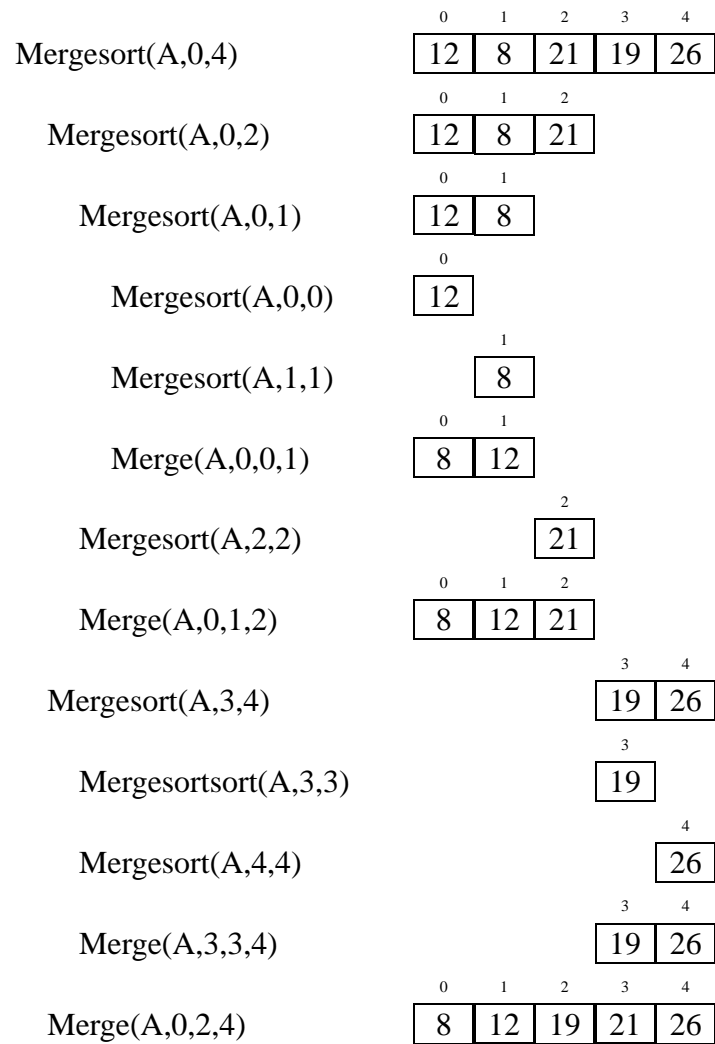
### Solución:

La traza del algoritmo *mergesort* sobre el vector  $A = \boxed{12} \boxed{8} \boxed{21} \boxed{19} \boxed{26}$  podría representarse de esta manera, donde los números entre paréntesis indican el orden de llamada a las funciones:



o bien de esta otra manera:

**llamada inicial: Mergesort(A,0,4);**



## Ejercicio 2:

-Realiza una versión del algoritmo *mergesort* que divida el vector en tres partes, para ello, deberás escribir otro algoritmo de mezcla *merge()* que actúe sobre tres *subsecuencias ordenadas*.

OJO! ten cuidado al obtener los puntos de corte, pues la fórmula no es exactamente igual que cuando se divide por la mitad. Ponte unos cuantos ejemplos para verlo. Ah! y ten cuidado porque a lo mejor hay más de un caso base, ya que antes el caso base era cuando quedaba un subvector de un solo elemento y ya estaba ordenado, ponte algún ejemplo y verás que pueden ocurrir otros casos.

## Solución:

Es fácil modificar la versión del algoritmo *mergesort* vista en clase. Solo hay que tener cuidado al obtener los "puntos de corte" para que queden subproblemas balanceados. Después habrá que modificar el algoritmo *merge* (de fusión) para que fusione 3 subsecuencias ordenadas, con lo que tendremos que hacer unas cuantas comprobaciones más. Los dos algoritmos son:

```
void mergesort(int *A, int l, int r) {
    int m1, m2;
    int aux;

    if (l < r) /* si tiene tamaño > 1, lo ordenamos. */
        if (l == (r-1)) { /* subvector de tamaño 2. */
            if (A[l] > A[r]) {
                aux = A[l];
                A[l] = A[r];
                A[r] = aux;
            }
        } else {
            m1 = (int) (((r-l)/3)+1);
            m2 = (int) (((2*(r-l))/3)+1);
            mergesort(A, l, m1);
            mergesort(A, m1 + 1, m2);
            mergesort(A, m2 + 1, r);
            merge(A, l, m1, m2, r);
        }
}
```

```

void merge(int *A, int l, int m1, int m2, int r) {
    int i,j,k,q;
    int *B;

    B = (int *) malloc((r-l+1)*sizeof(int));
    i=l;  j=m1 + 1;  k=m2 + 1;  q=0;

    while ((i<=m1) && (j<=m2) && (k<=r)) {
        if ((A[i] <= A[j]) && (A[i] <= A[k])) {
            B[q] = A[i];
            i++;
        }
        else if ((A[j] <= A[i]) && (A[j] <= A[k])) {
            B[q] = A[j];
            j++;
        }
        else {
            B[q] = A[k];
            k++;
        }
        q++;
    }

    while ((i<=m1) && (j<=m2)) {
        if (A[i] <= A[j]) {
            B[q] = A[i];
            i++;
        }
        else {
            B[q] = A[j];
            j++;
        }
        q++;
    }
}

```

```

while ((i<=m1) && (k<=r)) {
    if (A[i] <= A[k]) {
        B[q] = A[i];
        i++;
    }
    else {
        B[q] = A[k];
        k++;
    }
    q++;
}

while ((j<=m2) && (k<=r)) {
    if (A[j] <= A[k]) {
        B[q] = A[j];
        j++;
    }
    else {
        B[q] = A[k];
        k++;
    }
    q++;
}

while (i<=m1) {
    B[q] = A[i];
    i++;    q++;
}
while (j<=m2) {
    B[q] = A[j];
    j++;    q++;
}
while (k<=r) {
    B[q] = A[k];
    k++;    q++;
}
/* copiamos al vector original */
for (i=1; i<=r; i++) A[i] = B[i];
free(B);
}

```

---



### Ejercicio 3:

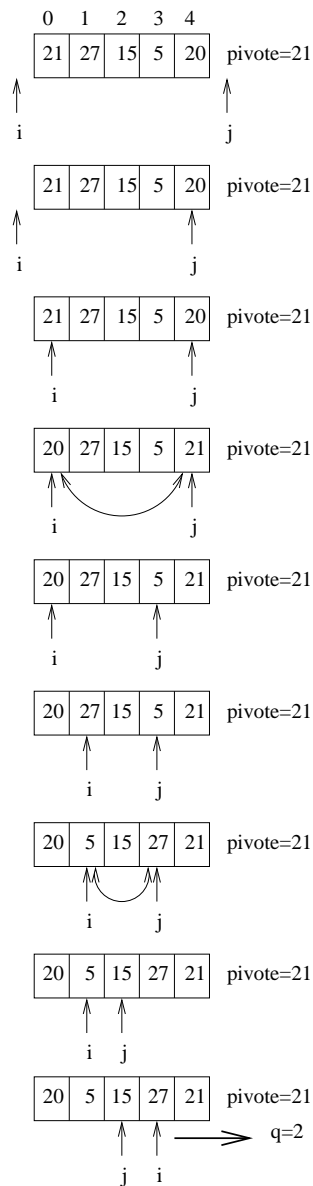
-Realiza una traza del algoritmo de partición que utiliza *Quicksort*, al aplicarlo sobre el vector  $A=\{21, 27, 15, 5, 20\}$ .

### Solución:

La traza del algoritmo *partition* (o de partición), sobre el vector  $A=$ 

21	27	15	5	20
----	----	----	---	----

 es la siguiente:

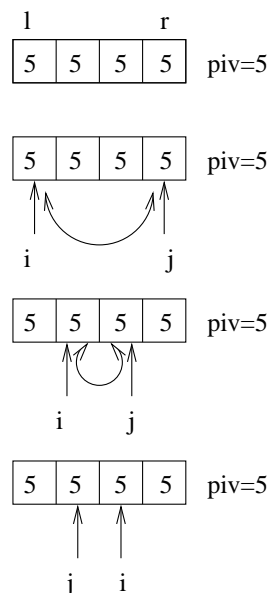


#### Ejercicio 4:

-¿Qué valor de  $q$  devuelve el algoritmo de *partición* cuando todos los elementos del vector tienen el mismo valor? ¿Y cuando el elemento escogido como *pivote* es el segundo menor del vector, si suponemos que no hay elementos repetidos en el vector?

#### Solución:

Para responder la primera pregunta podemos ver antes un ejemplo donde todos los elementos del vector sean iguales y aplicar el algoritmo de *partición*:



Vemos que en el ejemplo devuelve la posición  $j$  de la mitad del vector. Aunque fácilmente podemos ver que la solución será siempre la posición  $q = (\text{int})(l + r)/2$ , o sea, la posición de la mitad del vector. Además se habrán intercambiado todos los elementos del vector. El que se realicen todos estos intercambios es debido a que todos los elementos que comprueba el bucle de la  $j$  (del algoritmo *partición*) son menores o iguales (realmente son iguales) que el pivote y por tanto tiene que moverlos, y a su vez, todos los elementos que comprueba el bucle de la  $i$  son mayores o iguales (son iguales) que el pivote y por tanto tiene que moverlos.

La segunda pregunta fue respondida en clase cuando se realizó el análisis de casos para el coste medio de *Quicksort*. Si suponemos que no hay elementos repetidos, nos devolverá un valor  $q = 1$ , o lo que es lo mismo, un subvector formado por el elemento  $A[1]$  y otro subvector formado por los elementos  $A[2] \dots A[n]$ .

Esto es debido a que desde el principio del vector hasta la posición  $q$  sólo podrán quedar elementos menores o iguales que el pivote, como no hay elementos repetidos en nuestro caso solo podrá haber dos elementos como máximo, el mínimo del vector y el pivote, ya que el pivote es el segundo menor del vector. Sin embargo, el pivote tampoco estará en ese subvector porque siempre se mueve a la parte de la derecha de  $q$  (por cómo se definió el algoritmo partición), con lo que sólo nos queda el elemento mínimo del vector desde el inicio del vector hasta  $q$ , por tanto  $q = 1$ .

---

### Ejercicio 5:

-Dado un entero  $n$ ,  $n > 0$  se define la raíz cuadrada entera de  $n$  como  $\lfloor \sqrt{n} \rfloor$ . Dicho de otro modo, es la parte entera del número real que representa la  $\sqrt{n}$ . Podemos definir la raíz cuadrada entera de esta manera:

$$\lfloor \sqrt{n} \rfloor = z \mid z \text{ es entero y } z^2 \leq n < (z+1)^2$$

Siguiendo el esquema Divide y Vencerás, escribe una función que calcule eficientemente  $\lfloor \sqrt{n} \rfloor$ . A esta función la podemos llamar `raíz_entera`.

**Nota:** La idea es que busquemos un número  $z$  que cumpla  $z^2 \leq n < (z+1)^2$ . Hay que fijarse que la raíz entera estará incluida en un intervalo  $[x, y]$ , donde los números  $x$  e  $y$  son tales que  $1 \leq x \leq y \leq n$ . Así pues, nuestra idea Divide y Vencerás va a ser aplicar una reducción eficiente en ese intervalo. Es decir, hacer cada vez más pequeño el intervalo  $[x, y]$  hasta que demos con la solución.

El esquema del algoritmo va a ser, dado un número  $n$ , comenzaremos considerando el intervalo  $[1, n]$ , es decir,  $x = 1$  e  $y = n$ , a partir de ahí:

- Comprobar si  $x = y$ , entonces estaremos buscando en un intervalo con un solo número y habremos dado con la solución, devolveremos  $z = x$ . En cualquier otro caso:
- intentaremos recortar el intervalo de búsqueda de la solución, para ello, lo partimos por la mitad y vemos en cual de las dos mitades debemos seguir buscando la solución. Hacemos  $z' = \frac{x+y}{2}$  y ahora hay que saber si debemos seguir buscando en  $[x, z']$  o en  $[z' + 1, y]$ , para ello:
  - como buscamos un  $z$  que cumpla  $z^2 \leq n$ , entonces si  $(z')^2 > n \implies$  es que buscamos un número más bajo que  $z'$ , así pues, la  $z$  a buscar estará en  $[x, z']$ , con lo que aplicaremos la función `raíz_entera` al intervalo  $[x, z']$  y la solución sería la que nos devuelva esta función.
  - como buscamos un  $z$  que cumpla  $n < (z+1)^2$ , entonces si  $(z' + 1)^2 \leq n \implies$  es que buscamos un número más alto que  $z'$ , así pues, la  $z$  a buscar estará en  $[z' + 1, y]$ , con lo que aplicaremos la función `raíz_entera` al intervalo  $[z' + 1, y]$  y la solución sería la que nos devuelva esta función.
  - en cualquier otro caso es que estábamos buscando la  $z'$ , así pues devolvemos  $z'$ .

OJO! parece claro que dos de los parámetros de nuestra función serán los límites del intervalo donde estemos buscando, pero el algoritmo también necesitará conocer el valor de  $n$  para poder comparar con él. ¿qué te parece si lo incluimos como otro parámetro?

### Solución:

Dado un número entero  $n > 0$ , la llamada inicial a la función que calcula su raíz cuadrada entera mediante el anterior esquema divide y vencerás sería `raiz_entera(1,n,n)`. Y la función en lenguaje C sería:

```
int raiz_entera(int x, y, n){
    int z, z_cuadrado, z_mas_1_cuadrado;

    if (x==y) return(x);    /* la raiz esta en un intervalo */
                           /* de un solo numero */

    else {
        z = (int) (x+y)/2;
                /* calculo la mitad del intervalo [x,y] */

        /* Ahora hay que averiguar si toca buscar la raiz */
        /* de n en el intervalo [x,z'] o en [z'+1,y] */

        z_cuadrado = z*z;                /* z_cuadrado = z^2 */
        z_mas_1_cuadrado = (z+1)*(z+1);
                /* z_mas_1_cuadrado = (z+1)^2 */

        if (z_cuadrado > n) return( raiz_entera(x,z,n) );
        if (z_mas_1_cuadrado <= n)
            return( raiz_entera(z+1,y,n) );

        return(z);
    }
}
```

Si nos fijamos podemos observar que el coste de este algoritmo sería  $O(\log n)$ , ya que el análisis del coste sería igual que en el caso de la búsqueda binaria. Vamos partiendo un intervalo de tamaño  $n$  por la mitad hasta que encontramos la solución.

# Tema 5

## Árboles

### 5.1. Definiciones

- Un *árbol* de tipo base  $T$  puede ser definido como [Wirth]:
  1. Un conjunto vacío es un árbol,
  2. Un nodo  $n$  de tipo  $T$ , junto con un número finito de conjuntos disjuntos de elementos de tipo base  $T$ , llamados *subárboles*, es un árbol. El nodo  $n$  se denomina *raíz*.

Se dice que el nodo  $n$  es *padre* de las raíces de los subárboles, y que las raíces de los subárboles son *hijos* de  $n$ .

Por ejemplo, sea el tipo de base  $T$  el conjunto de las letras del alfabeto; un árbol puede ser representado mediante un grafo de la siguiente forma:

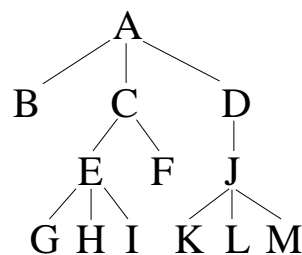


Figura 5.1: Representación de un árbol

- Se denomina **camino** de  $n_1$  a  $n_k$ , a una sucesión de nodos de un árbol  $n_1, n_2, \dots, n_k$ , en la que  $n_i$  es padre de  $n_{i+1}$  para  $1 \leq i < k$ . Por ejemplo, en el árbol de la figura 5.1, el camino de  $A$  hasta  $I$  es:  $A C E I$ .

- La **longitud de un camino** es el número de nodos del camino menos 1. Por lo tanto, hay un camino de longitud cero de cualquier nodo a sí mismo. Por ejemplo en el árbol de la figura 5.1, la longitud del camino de  $A$  hasta  $I$  es: 3.
- Si existe un camino de un nodo  $n_i$  a otro  $n_j$ , entonces  $n_i$  es un **antecesor** de  $n_j$ , y  $n_j$  es un **descendiente o sucesor** de  $n_i$ . Por lo tanto, en un camino  $n_1, n_2, \dots, n_k$  se dice que  $n_1, n_2, \dots, n_i, 1 \leq i \leq k$  son **antecesores** de  $n_i$ ; análogamente,  $n_i, n_{i+1}, \dots, n_k, 1 \leq i \leq k$  son **descendientes** de  $n_i$ . Obsérvese que cada nodo es a la vez un antecesor y un descendiente de sí mismo. Por ejemplo, en el árbol de la figura 5.1, los antecesores de  $E$  son:  $\langle A, C, E \rangle$  y los descendientes:  $\langle E, G, H, I \rangle$ .
- Un antecesor o un descendiente de un nodo que no sea él mismo recibe el nombre de **antecesor propio** o **descendiente propio**, respectivamente. En un árbol, la raíz será el único nodo que no tiene antecesores propios. Por ejemplo, en el árbol de la figura 5.1, los antecesores propios de  $E$  son:  $\langle A, C \rangle$  y los descendientes propios:  $\langle G, H, I \rangle$ .
- En un camino  $n_1, n_2, \dots, n_k, n_i$  es **antecesor directo** de  $n_{i+1}$ , y  $n_{i+1}$  es **descendiente directo** de  $n_i$ , para  $1 \leq i < k$ . Por ejemplo, en el árbol de la figura 5.1, el antecesor directo de  $E$  es:  $\langle C \rangle$  y los descendientes directos:  $\langle G, H, I \rangle$ .
- Una **hoja** o **nodo terminal** es un nodo sin descendientes propios. Por ejemplo, en el árbol de la figura 5.1, los nodos terminales son:  $\langle B, G, H, I, F, K, L, M \rangle$ .
- Un **nodo interior** es un nodo con descendientes propios. Por ejemplo, en el árbol de la figura 5.1, los nodos interiores son:  $\langle A, C, D, E, J \rangle$ .
- Se denomina **grado de un nodo** al número de descendientes directos que tiene. Por ejemplo, en el árbol de la figura 5.1, el grado del nodo  $E$  es: 3.
- Se denomina **grado de un árbol** al máximo de los grados de todos los nodos que pertenecen a él. Por ejemplo, el árbol de la figura 5.1 es de grado 3.
- Definimos recursivamente **nivel** como:
  1. La raíz del árbol está a nivel 1.
  2. Si un nodo está en el nivel  $i$ , entonces sus descendientes directos están al nivel  $i + 1$ .

- Se denomina **altura de un nodo**, a la longitud del camino más largo desde ese nodo hasta una hoja. La altura del nodo raíz es la **altura del árbol**. Por ejemplo, en el árbol de la figura 5.1, la altura del árbol es: 3, y la altura del nodo  $E$ : 1.
- Se denomina **profundidad de un nodo** a la longitud del camino único desde la raíz hasta ese nodo. Por ejemplo, en el árbol de la figura 5.1, la profundidad del nodo  $E$  es: 2.

## 5.2. Recorrido de árboles

Una tarea muy habitual cuando utilizamos un árbol, es realizar una determinada operación  $P$  con cada uno de los elementos del árbol. Para ello es necesario recorrer cada uno de los nodos del árbol. Las tres formas más importantes de recorrer un árbol son:

- recorrido en orden previo (**preorden**),
- recorrido en orden simétrico (**inorden**),
- recorrido en orden posterior (**postorden**).

Dependiendo del tipo de recorrido que se realice, la acción  $P$  se llevará a cabo en cada nodo de la siguiente forma:

- si un árbol  $A$  es vacío, entonces no se realiza la acción  $P$ ;
- si  $A$  contiene un único nodo, entonces se realiza la acción  $P$  sobre ese nodo;
- si  $A$  es un árbol con raíz  $n$  y subárboles  $A_1, A_2, \dots, A_k$  de izquierda a derecha, entonces:
  - En el recorrido en *preorden* se realiza la acción  $P$  sobre el nodo raíz  $n$ , y, a continuación, se recorren en *preorden* los subárboles  $A_1, A_2, \dots, A_k$  de izquierda a derecha.
  - En el recorrido en *inorden* se recorre en *inorden* el subárbol  $A_1$ , después se realiza la acción  $P$  sobre el nodo raíz  $n$ , y, a continuación, se recorren en *inorden* los subárboles  $A_2, \dots, A_k$  de izquierda a derecha.
  - En el recorrido en *postorden* se recorren en *postorden* los subárboles  $A_1, A_2, \dots, A_k$  de izquierda a derecha, y después se realiza la acción  $P$  sobre el nodo  $n$ .



### Ejemplo:

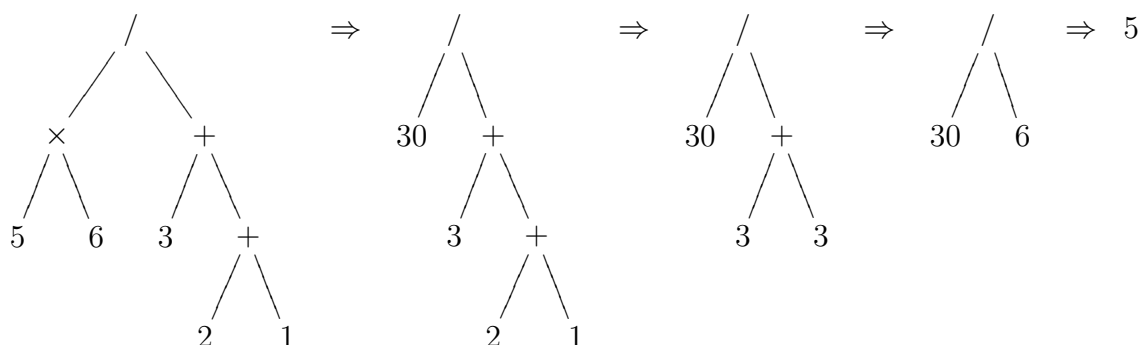
Si se recorriera el árbol de la figura 5.1 en *preorden*, la acción  $P$  sobre cada nodo se realizaría en el siguiente orden:  $A, B, C, E, G, H, I, F, D, J, K, L, M$ .

Si se recorriera el árbol de la figura 5.1 en *inorden*, la acción  $P$  sobre cada nodo se realizaría en el siguiente orden:  $B, A, G, E, H, I, C, F, K, J, L, M, D$ .

Si se recorriera el árbol de la figura 5.1 en *postorden*, la acción  $P$  sobre cada nodo se realizaría en el siguiente orden:  $B, G, H, I, E, F, C, K, L, M, J, D, A$ .

### Ejemplo:

Un ejemplo del uso de recorrido de árboles sería ver cómo se puede evaluar una expresión aritmética con la clásica precedencia de operadores representada mediante un árbol. Para ello necesitaremos realizar un recorrido en **postorden** que para cada nodo se encargará de obtener primero los valores de sus nodos hijos, esto es, los valores de los operandos evaluando recursivamente cada uno de ellos. Por último se llevará a cabo la operación asociada a ese nodo:



## 5.3. Representación de árboles

### 5.3.1. Representación mediante listas de hijos

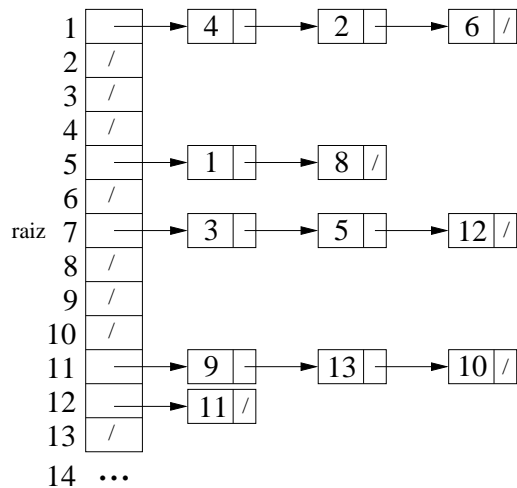
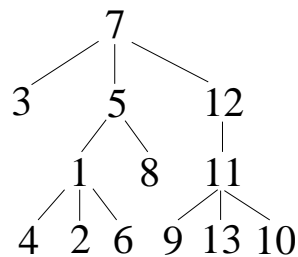
Una manera importante y útil de representar árboles consiste en formar una lista de los hijos de cada nodo. Debido a que el número de hijos que puede tener cada nodo es variable, las listas enlazadas resultarán ser la representación más apropiada para representar las listas de hijos.

La estructura de datos utilizada para este tipo de representación, consistirá en:

1. Un vector  $v$  para almacenar la información de cada nodo;
2. cada elemento (nodo) del vector apunta a una lista enlazada de elementos (nodos) que indican cuáles son sus nodos hijos. Los elementos de la lista encabezada por  $v[i]$  serán los hijos del nodo  $i$ .

Es conveniente tener una variable que indique cual es el nodo raíz, aunque no sería necesaria, puesto que para obtener la raíz se podría buscar aquel nodo que no tiene padre. Sin embargo, conocer la raíz evitaría el coste de esa búsqueda.

### Ejemplo de representación de un árbol mediante listas de hijos



### Ventajas de esta representación

- Es una representación simple.
- Facilita las operaciones de acceso a los hijos.

### Inconvenientes de esta representación

- Se desaprovecha memoria.
- Las operaciones para acceder al padre de un nodo son costosas.

### Ejercicio

Sea la siguiente definición de tipos en C:

```

#define N ...

typedef struct snodo {
    int e;
    struct snodo *sig;
} nodo;

typedef struct {
    int raiz;
    nodo *v[N];
} arbol;

```

Escribir una función recursiva que imprima en *preorden* los índices de los nodos de un árbol  $T$  que use esta representación.

```

void preorden(arbol *T, int n) {
    nodo *aux;

    aux = T->v[n];
    printf("%d ", n); /* Accion sobre nodo */
    while (aux != NULL) {
        preorden(T, aux->e);
        aux = aux->sig;
    }
}

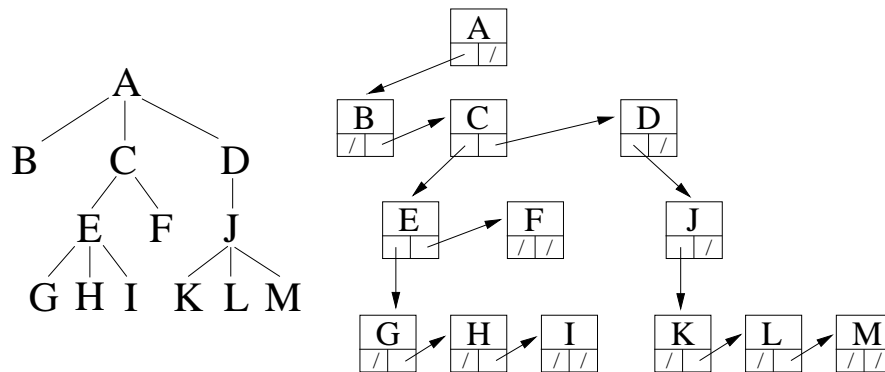
```

### 5.3.2. Representación “hijo más a la izquierda – hermano derecho”

Otra posible representación de un árbol es tener, para cada nodo, una estructura en la que se guarda la siguiente información:

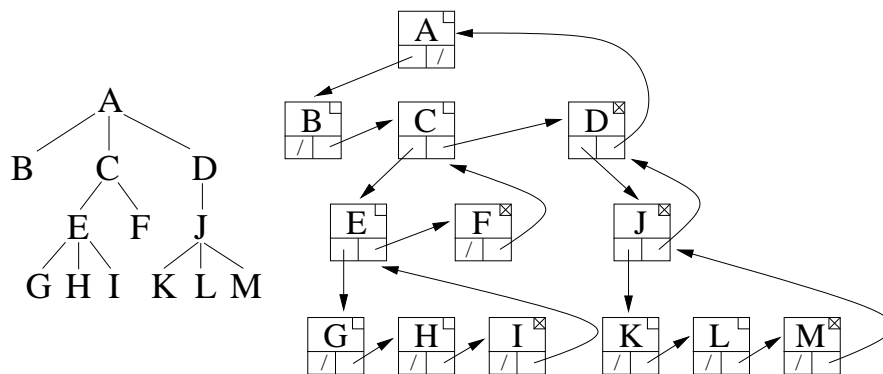
1. clave: valor de tipo base  $T$  almacenado en el nodo.
2. hijo izquierdo: hijo más a la izquierda del nodo.
3. hermano derecho: hermano derecho del nodo.

### Ejemplo de representación de un árbol mediante esta estructura



Una posible variante de esta representación que facilita el acceso al padre desde un nodo hijo, es enlazar el nodo hijo que ocupa la posición más a la derecha con su nodo padre. Para ello será necesario especificar, en cada nodo, si el puntero al hermano derecho apunta a un hermano o al padre.

### Ejemplo de representación de un árbol mediante esta variante



### Ventajas de esta representación

- Facilita las operaciones de acceso a los hijos y al padre de un nodo.
- Uso eficiente de la memoria.

## Inconvenientes de esta representación

- El mantenimiento de la estructura es complejo.

### Ejercicio

Sea la siguiente definición de tipos en C:

```
typedef struct snodo {  
    char clave[2];  
    struct snodo *hizq;  
    struct snodo *der;  
} arbol;
```

Escribir una función que calcule de forma recursiva la altura de un árbol T.

Como buscamos una estrategia recursiva, la idea va a ser calcular la altura de cada uno de los hijos del nodo raíz y sumarle 1 a la altura máxima que hayamos obtenido. El esquema recursivo se encuentra en el cálculo de la altura de los nodos hijos del nodo actual.

```
int altura(arbol *T) {  
    arbol *aux;  
    int maxhsub=0, hsub;  
  
    if (T == NULL)  
        return(0);  
    else if (T->hizq == NULL)  
        return(0);  
    else {  
        aux = T->hizq;  
        while ( aux != NULL) {  
            hsub = altura(aux);  
            if (hsub > maxhsub)  
                maxhsub = hsub;  
            aux = aux->der;  
        }  
        return(maxhsub + 1);  
    }  
}
```

## 5.4. Árboles binarios

**Definición:** Un árbol binario es un conjunto finito de nodos tal que, o está vacío, o consiste en un nodo especial llamado *raíz*, y el resto de nodos se agrupan en dos árboles binarios disjuntos llamados *subárbol izquierdo* y *subárbol derecho*.

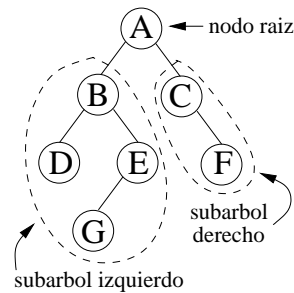


Figura 5.2: Ejemplo de árbol binario

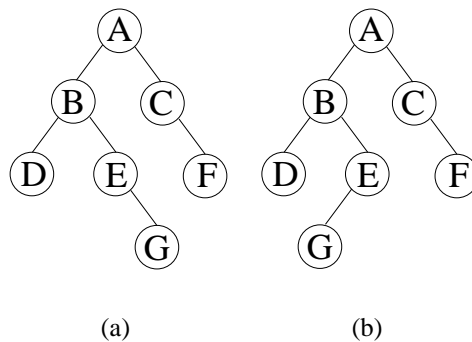


Figura 5.3: Ejemplo de dos árboles binarios distintos. El nodo *E* del árbol binario (a) no tiene hijo izquierdo y sí que tiene hijo derecho, mientras que en el árbol binario (b), el nodo *E* sí que tiene hijo izquierdo y no tiene hijo derecho.

### 5.4.1. Representación de árboles binarios

A continuación se muestran diferentes estructuras de datos que sirven para representar árboles binarios.

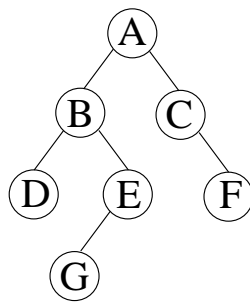
#### Representación mediante vectores

La estructura de datos utilizada para este tipo de representación, consistirá en:

- Un vector  $v$  para almacenar la información de cada nodo;
- cada elemento del vector (nodo), será una estructura que almacenará la siguiente información:
  1. clave: valor de tipo base  $T$  almacenado en el nodo.
  2. hijo izquierdo: índice del nodo que es hijo izquierdo.
  3. hijo derecho: índice del nodo que es hijo derecho.

Es conveniente también guardar en una variable el índice del nodo raíz para saber dónde empieza el árbol.

### Ejemplo de representación de un árbol mediante esta estructura



/	/	/	0
/	F	/	1
/	/	/	2
5	A	6	3 raíz
/	D	/	4
4	B	8	5
/	C	1	6
/	G	/	7
7	E	/	8
⋮			
/	/	/	N-1

La siguiente definición de tipos en C, nos serviría para definir una estructura de datos que se ajuste a esta representación.

```
#define N ...
```

```
typedef ... tipo_baseT;
```

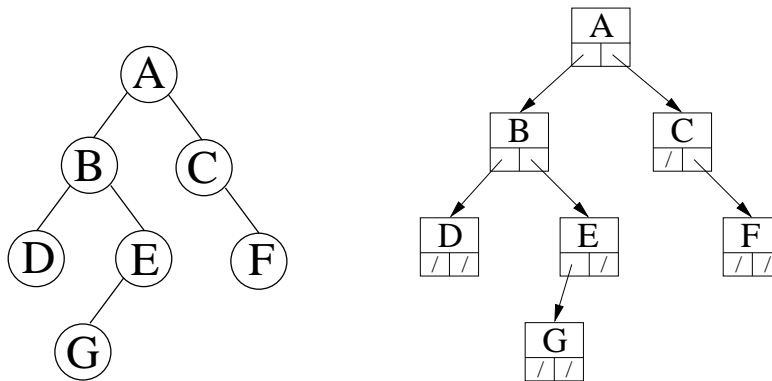
```
typedef struct {
    tipo_baseT e;
    int hizq, hder;
} nodo;
```

```
typedef struct {
    int raiz;
    nodo v[N];
} arbol;
```

## Representación mediante variables dinámicas

Otra posible representación de un árbol binario es tener, para cada nodo, una estructura en la que se guarda la siguiente información:

1. clave: valor de tipo base  $T$  almacenado en el nodo.
2. hijo izquierdo: puntero al hijo izquierdo.
3. hijo derecho: puntero al hijo derecho.

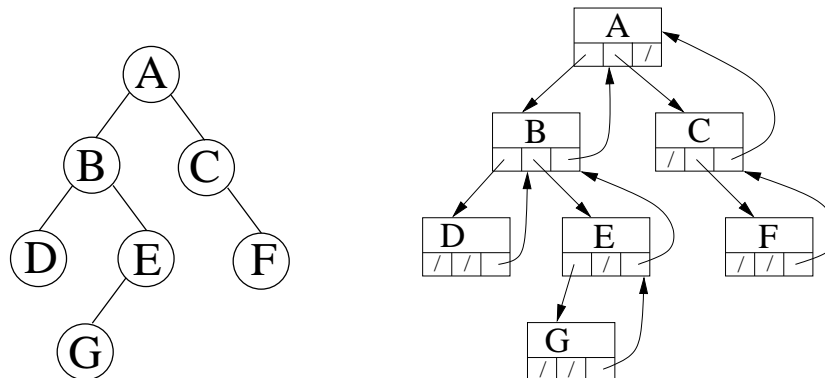


La siguiente definición de tipos en C, nos serviría para definir una estructura de datos que se ajuste a esta representación.

```
typedef ... tipo_baseT;  
  
typedef struct snodo {  
    tipo_baseT e;  
    struct snodo *hizq, *hder;  
} arbol;
```

Una posible variante a este tipo de representación, que facilitaría el acceso al nodo padre desde un nodo hijo, sería almacenar también, en cada nodo, un puntero al nodo padre. La siguiente figura muestra este tipo de representación.





La siguiente definición de tipos en C, nos serviría para definir una estructura de datos que se ajuste a esta representación.

```
typedef ... tipo_baseT;

typedef struct snodo {
    tipo_baseT e;
    struct snodo *hizq, *hder, *padre;
} arbol;
```

### 5.4.2. Recorrido de árboles binarios

El recorrido de árboles binarios podrá hacerse, al igual que ocurría para cualquier tipo de árbol, de tres formas distintas:

- recorrido en orden previo (*preorden*)
- recorrido en orden simétrico (*inorden*)
- recorrido en orden posterior (*postorden*)

Suponiendo que se tuviera que realizar una misma acción  $P$  sobre cada nodo del árbol, el procedimiento a seguir para cada uno de los distintos recorridos sería:

```
Preorden( $x$ )
    si  $x \neq \text{VACIO}$  entonces
        AccionP( $x$ )
        Preorden(hizquierdo( $x$ ))
        Preorden(hderecho( $x$ ))
```

```

Inorden( $x$ )
    si  $x \neq \text{VACIO}$  entonces
        Inorden(hizquierdo( $x$ ))
        AccionP( $x$ )
        Inorden(hderecho( $x$ ))

Postorden( $x$ )
    si  $x \neq \text{VACIO}$  entonces
        Postorden(hizquierdo( $x$ ))
        Postorden(hderecho( $x$ ))
        AccionP( $x$ )

```

El coste de todos los algoritmos es  $\Theta(n)$ , siendo  $n$  el número de nodos del árbol.

### Ejercicio

Suponiendo que la representación del árbol binario se realice mediante variables dinámicas, escribir una función en C para cada tipo de recorrido, de forma que la acción  $P$  sea escribir la *clave* del nodo.

```

typedef int tipo_baseT;

typedef struct snodo {
    tipo_baseT e;
    struct snodo *hizq, *hder;
} arbol;

void preorden(arbol *a) {
    if (a != NULL) {
        printf("%d ", a->e);
        preorden(a->hizq);
        preorden(a->hder);
    }
}

void inorden(arbol *a) {
    if (a != NULL) {
        inorden(a->hizq);
        printf("%d ", a->e);
        inorden(a->hder);
    }
}

```

```

void postorden(arbol *a) {
    if (a != NULL) {
        postorden(a->hizq);
        postorden(a->hder);
        printf( "%d ", a->e);
    }
}

```

### Ejercicio

Dada la siguiente definición de tipos para especificar la estructura de un árbol binario de enteros:

```

typedef int tipo_baseT;

typedef struct _nodo {
    tipo_baseT info;
    struct _nodo *hizq, *hder;
} arbol;

```

y dado un árbol binario  $T$ , declarado como:

```
arbol *T;
```

Escribir una función en C que elimine todas las hojas del árbol binario, dejando exclusivamente los nodos del árbol que no lo son. Considerar que eliminar las hojas de un árbol vacío deja el árbol vacío.

```

arbol *elimina_hojas(arbol *T) {
    if (T == NULL)
        return(NULL);
    else if ( (T->hizq == NULL) && (T->hder == NULL) ) {
        free(T);
        return(NULL);
    } else {
        T->hizq = elimina_hojas(T->hizq);
        T->hder = elimina_hojas(T->hder);
        return(T);
    }
}

```

### 5.4.3. Árbol binario completo. Representación.

**Definición:** Un árbol binario completo es un árbol binario en el cual todos los niveles tienen el máximo número posible de nodos excepto, puede ser, el último. En ese caso, las hojas del último nivel están tan a la izquierda como sea posible.

En la figura 5.4 se muestra un ejemplo de árbol binario completo.

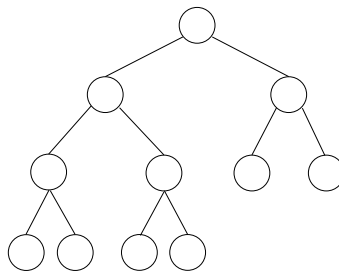


Figura 5.4: Ejemplo de árbol binario completo

Los árboles binarios completos pueden ser representados mediante un vector de la siguiente forma:

- En la posición 1 se encuentra el nodo raíz del árbol.
- Dado un nodo que ocupa la posición  $i$  en el vector:
  - En la posición  $2i$  se encuentra el nodo que es su hijo izquierdo.
  - En la posición  $2i + 1$  se encuentra el nodo que es su hijo derecho.
  - En la posición  $\lfloor i/2 \rfloor$  se encuentra el nodo padre si  $i > 1$ .

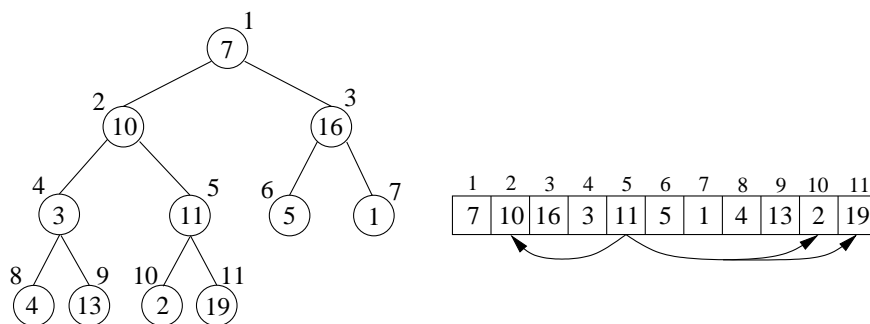


Figura 5.5: Representación de un árbol binario completo mediante un vector.

## Ejercicio

Escribe tres funciones en C que, dado un nodo de un árbol binario completo representado mediante un vector, calculen la posición del vector en la que se encuentra el nodo padre, el hijo izquierdo y el hijo derecho, respectivamente.

```
int hizq(int i)
{
    return (2*i);
}
```

```
int hder(int i)
{
    return ((2*i)+1);
}
```

```
int padre(int i)
{
    return (i/2);
}
```

### 5.4.4. Propiedades de los árboles binarios

Por último, veamos cuáles son algunas de las propiedades más importantes de los árboles binarios:

- El máximo número de nodos en el nivel  $i$  es  $2^{i-1}$ ,  $i \geq 1$ .
- En un árbol de  $i$  niveles hay como máximo  $2^i - 1$  nodos,  $i \geq 1$ .
- En un árbol binario no vacío, si  $n_0$  es el número de hojas y  $n_2$  es el número de nodos de grado 2, se cumple que  $n_0 = n_2 + 1$ .
- La altura de un árbol binario completo que contiene  $n$  nodos es  $\lfloor \log_2 n \rfloor$ .

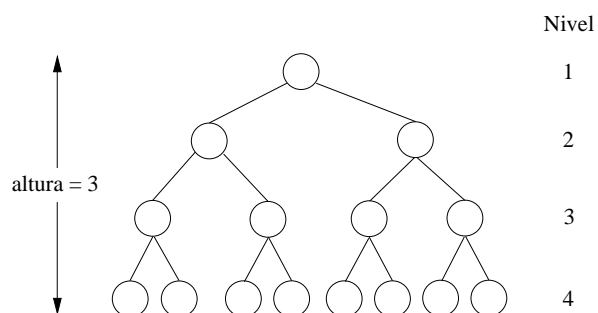


Figura 5.6: Ejemplo de un árbol binario completo con el máximo numero de nodos.

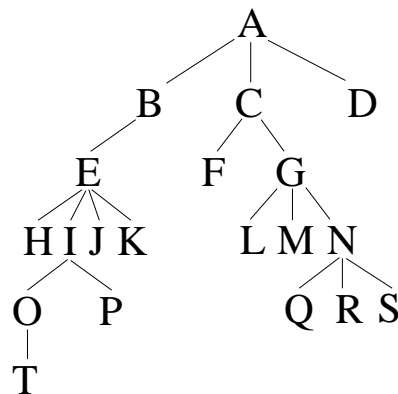
**Ejercicio:** Comprueba sobre el árbol de la figura 5.6 que se cumple cada una de las propiedades citadas.

Máximo nodos por nivel	
Nivel 1	$2^0 = 1$
Nivel 2	$2^1 = 2$
Nivel 3	$2^2 = 4$
Nivel 4	$2^3 = 8$
Máximo nodos en el árbol	
$2^4 - 1 = 15$	
Número de hojas ( $n_0$ )	
$n_2 = 7$	
$n_0 = n_2 + 1 = 7 + 1 = 8$	
Altura del árbol binario completo	
$n = 15$	
$\lfloor \log_2 n \rfloor = \lfloor \log_2 (15) \rfloor = 3$	

## 5.5. Ejercicios

### Ejercicio 1:

-Si la acción  $P$  fuera escribir la etiqueta del nodo, indica en qué orden se escribirían cada una de las etiquetas del siguiente árbol, para cada uno de los recorridos: *preorden*, *inorden* y *postorden*.



¿Cuál es la altura del árbol? ¿Y el grado?

### Solución:

**Preorden:** A, B, E, H, I, O, T, P, J, K, C, F, G, L, M, N, Q, R, S, D

**Inorden:** H, E, T, O, I, P, J, K, B, A, F, C, L, G, M, Q, N, R, S, D

**Postorden:** H, T, O, P, I, J, K, E, B, F, L, M, Q, R, S, N, G, C, D, A

El árbol tiene altura 5 y grado 4.

---

## Ejercicio 2:

-Para adoptar la representación de árboles binarios mediante vectores, se propone la siguiente definición de tipos, constantes y variables en lenguaje C:

```
#define N 100

typedef int tipo_baseT;

typedef struct{
    tipo_baseT e;
    int hizq, hder;
} nodo;

typedef struct{
    int raiz;
    nodo v[N];
} arbol;

arbol *T;
```

a) Escribir una función recursiva con el perfil:

```
void preorden(arbol *T, int m)
```

que imprima las claves de los nodos en *preorden*, cuando la llamada inicial es `preorden(T, T->raiz)` ;.

b) Escribir una función recursiva con el perfil:

```
int altura(arbol *T, int m)
```

que determine la altura de un árbol binario T, cuando la llamada inicial es `H=altura(T, T->raiz)` ;. ¿Cuál será el coste temporal del algoritmo?

c) Escribir una función recursiva con el perfil:

```
void espejo(arbol *T, int m)
```

que intercambie los hijos izquierdo y derecho de cada nodo, cuando la llamada inicial es `espejo(T, T->raiz)` ;. ¿Cuál será el coste temporal del algoritmo?



### Solución:

- a) Debemos aplicar un recorrido en preorden como el comentado en clase, pero debemos tener en cuenta que en este caso se especifica el nodo en que nos encontramos en cada llamada que hacemos a la función `preorden`:

```
void preorden(arbol *T, int m) {
    if (T!=NULL)
        if (m!=-1) {
            printf("%d, ", T->v[m].e);
            preorden(T, T->v[m].hizq);
            preorden(T, T->v[m].hder);
        }
}
```

- b) El esquema recursivo de la función vendrá dado porque la altura de un nodo determinado será uno más el máximo de las alturas de sus dos hijos:

```
int altura(arbol *T, int m) {
    int alt_hizq, alt_hder;

    if (T!=NULL)
        if (m!=-1) {
            if (T->v[m].hizq != -1)
                alt_hizq = altura(T, T->v[m].hizq);
            else alt_hizq = 0;
            if (T->v[m].hder != -1)
                alt_hder = altura(T, T->v[m].hder);
            else alt_hder = 0;

            if (alt_hizq>alt_hder) return(1+alt_hizq);
            else return(1+alt_hder);
        }
    else return(-1);
}
```

El coste del algoritmo es  $O(n)$ , siendo  $n$  el número de nodos del árbol, puesto que tendremos que ir calculando la altura de todos los nodos del árbol para conocer finalmente la altura del árbol, y por ello tendremos que, en cierta manera, *recorrer* el árbol.

- c) El esquema recursivo del procedimiento espejo es debido a que una vez hayamos dado la vuelta a los hijos del nodo actual, quedaría por dar la vuelta a todos los hijos del subárbol derecho y el subárbol izquierdo del

nodo actual, con lo que podríamos usar el mismo procedimiento aplicándolo al hijo izquierdo y al hijo derecho respectivamente:

```
void espejo(arbol *T, int m) {  
    int aux;  
  
    if (T!=NULL)  
        if (m!=-1) {  
            aux = T->v[m].hizq;  
            T->v[m].hizq = T->v[m].hder;  
            T->v[m].hder = aux;  
            espejo(T, T->v[m].hizq);  
            espejo(T, T->v[m].hder);  
        }  
}
```

El coste del algoritmo es  $O(n)$ , siendo  $n$  el número de nodos del árbol, puesto que tendremos que dar la vuelta a todos los hijos de todos los nodos internos del árbol. Al igual que en el apartado anterior, tendremos que *recorrer* todo el árbol.

---

### Ejercicio 3:

-Para adoptar la representación de árboles binarios mediante variables dinámicas, se propone la siguiente definición de tipos y variables en lenguaje C:

```
typedef int tipo_baseT;
```

```
typedef struct _nodo{  
    tipo_baseT info;  
    struct _nodo *hizq, *hder;  
} arbol;
```

```
arbol *T1, *T2;
```

Se solicita:

- a) Escribir una función recursiva con el perfil:

```
arbol *buscar(arbol *T, tipo_baseT m)
```

que devuelva un puntero al nodo de clave m del árbol binario T. Si el nodo de clave m no se encuentra en el árbol, la función devolverá NULL. Esta función puede verse como una función que devuelve un puntero al subárbol cuya raíz es el nodo m. ¿Cuál será el coste temporal del algoritmo?

- b) Suponiendo que ya hemos obtenido la función buscar de la cuestión anterior, utilizarla para escribir un procedimiento recursivo con el perfil:

```
void interseccion(arbol *T1, arbol *T2)
```

que imprima las claves de los nodos que son el resultado de obtener la intersección de los árboles binarios T1 y T2; es decir, las claves de los nodos que pertenecen a la vez a T1 y T2. ¿Cuál será el coste temporal del algoritmo?

- c) Suponiendo que ya hemos obtenido la función buscar de la cuestión a), utilizarla para escribir un procedimiento recursivo con el perfil:

```
void diferencia(arbol *T1, arbol T2)
```

que imprima las claves de los nodos que son el resultado de obtener la diferencia T1-T2; es decir, las claves de los nodos que pertenecen a T1 y no pertenecen a T2. ¿Cuál será el coste temporal del algoritmo?

### Solución:

- a) Habrá que hacer un recorrido por el árbol hasta que se encuentre el nodo buscado, la única diferencia con un recorrido tradicional es que una vez se haya encontrado, debe devolverse el puntero correspondiente y no seguir explorando nuevos nodos. A partir de un esquema similar al del recorrido en preorden podemos obtener esta función:

```
arbol *buscar(arbol *T, tipo_baseT m) {
    arbol *T_aux;

    if (T!=NULL) {
        if (T->info == m) return(T);
        else {
            T_aux = buscar(T->hizq, m);
            if (T_aux != NULL) return(T_aux);
            else {
                T_aux = buscar(T->hder, m);
                return(T_aux);
            }
        }
    }
    else return(T);
}
```

El coste del algoritmo será  $O(n)$ , siendo  $n$  el número de nodos en el árbol. Esto es debido a que en el caso peor (que el nodo buscado fuera el último visitado durante el recorrido del árbol) podríamos tener que recorrer todos los nodos del árbol.

- b) El procedimiento debe imprimir aquellos nodos de T1 cuya clave también sea clave de un nodo de T2. Para ello, recorreremos todos los nodos de T1, y para cada uno de ellos comprobaremos con la función buscar si existe un nodo en T2 con la misma clave, en el caso en que se encuentre en T2, lo imprimiremos:

```
void interseccion(arbol *T1, arbol *T2) {
    if (T1!=NULL){
        if (buscar(T2,T1->info)!= NULL)
            printf("%d, ",T1->info);
        interseccion(T1->hizq, T2);
        interseccion(T1->hder, T2);
    }
}
```

El coste del algoritmo será  $O(n_1 \times n_2)$ , siendo  $n_1$  el número de nodos del árbol T1 y  $n_2$  el número de nodos del árbol T2. Esto es debido a que hay que recorrer todos los nodos del árbol T1 y, para cada uno de esos nodos, ejecutar una operación buscar sobre T2, que tiene un coste  $O(n_2)$ .

- c) Es sencillo, el procedimiento debe imprimir aquellos nodos de T1 cuya clave no sea la clave de ningún nodo de T2. Para ello, recorreremos todos los nodos de T1, y para cada uno de ellos comprobaremos con la función `buscar` si existe un nodo en T2 con la misma clave, en el caso en que no se encuentre en T2, lo imprimiremos:

```
void diferencia(arbol *T1, arbol *T2) {
    if (T1 != NULL) {
        if (buscar(T2,T1->info) == NULL)
            printf("%d, ",T1->info);
        diferencia(T1->hizq, T2);
        diferencia(T1->hder, T2);
    }
}
```

El coste del algoritmo será  $O(n_1 \times n_2)$ , siendo  $n_1$  el número de nodos del árbol T1 y  $n_2$  el número de nodos del árbol T2. Esto es debido a que hay que recorrer todos los nodos del árbol T1 y, para cada uno de esos nodos, ejecutar una operación buscar sobre T2, que tiene un coste  $O(n_2)$ .

# Tema 6

## Representación de Conjuntos

### 6.1. Conceptos generales

**Definición:** Un conjunto es una colección de *miembros* o elementos distintos; cada miembro de un conjunto puede ser , a su vez, un conjunto, o un átomo o *ítem*. Un conjunto que contiene elementos repetidos se denomina *multiconjunto*.

#### 6.1.1. Representación de conjuntos

Un conjunto puede representarse de distintas formas:

- Representación explícita: se enumeran todos los elementos del conjunto encerrándolos entre llaves.

$$C = \{1, 4, 7, 11\}$$

- Representación mediante propiedades: se especifica alguna propiedad que caracterice a los elementos del conjunto.

$$C = \{x \in \mathbb{N} \mid x \text{ es par}\}$$

#### 6.1.2. Notación

La relación fundamental en teoría de conjuntos es la de pertenencia, que se indica por el símbolo “ $\in$ ”. Se dice que un elemento  $x$  *pertenece al conjunto*  $A$ ,  $x \in A$ , cuando  $x$  es un miembro del conjunto  $A$ ;  $x$  puede ser un ítem u otro conjunto, pero  $A$  no puede ser un *ítem*. Se utiliza  $x \notin A$  para señalar que  $x$  *no es un miembro de*  $A$ .

Al número de elementos que contiene un conjunto se le denomina *cardinalidad* o *talla* de un conjunto.

Existe un conjunto especial, llamado *conjunto vacío o nulo*, que no tiene miembros (de cardinalidad 0), y se simboliza con  $\emptyset$ .

Se dice que un conjunto  $A$  está incluido (o contenido) en un conjunto  $B$ , y se escribe  $A \subseteq B$  o  $B \supseteq A$ , si todo miembro de  $A$  también es miembro de  $B$ . En este caso se dice que  $A$  es un *subconjunto* de  $B$ . Todo conjunto es un *subconjunto* de sí mismo, y el conjunto vacío es un *subconjunto* de todo conjunto.

Dos conjuntos son iguales si cada uno de ellos está incluido en el otro,  $A \subseteq B$  y  $B \subseteq A$ ; es decir, si sus miembros son los mismos.

Un conjunto  $A$  es un *subconjunto propio* de otro  $B$ , si  $A \neq B$  y  $A \subseteq B$ .

### 6.1.3. Operaciones elementales sobre conjuntos

Las operaciones elementales con conjuntos son: unión, intersección y diferencia.

- *Unión de dos conjuntos*: Si  $A$  y  $B$  son conjuntos, entonces la unión de  $A$  y  $B$ ,  $A \cup B$ , es el conjunto de los elementos que son miembros de  $A$ , de  $B$ , o de ambos.
- *Intersección de dos conjuntos*: Si  $A$  y  $B$  son conjuntos, la intersección de  $A$  y  $B$ ,  $A \cap B$ , es el conjunto de los elementos que pertenecen, a la vez, tanto a  $A$  como a  $B$ .
- *Diferencia de dos conjuntos*: Si  $A$  y  $B$  son conjuntos, la diferencia,  $A - B$ , es el conjunto de los elementos que pertenecen a  $A$  y no pertenecen a  $B$ .

### 6.1.4. Conjuntos dinámicos

Un conjunto en el que sus *miembros* pueden variar a lo largo del tiempo, se denomina *conjunto dinámico*. Típicamente, un algoritmo que gestione un conjunto dinámico, representará cada uno de sus elementos mediante un objeto que contendrá la siguiente información:

- *clave*: valor que identifica al elemento.
- *información satélite*: información asociada al elemento.

En algunos conjuntos dinámicos puede establecerse una relación de orden total entre las claves; por ejemplo, si las claves son números enteros, reales, palabras (orden alfabético), etc. La existencia de un orden total, permitirá definir el mínimo y el máximo elemento de un conjunto, o hablar del predecesor o sucesor de un elemento dado.

## Operaciones sobre conjuntos dinámicos

Las operaciones que se pueden realizar sobre conjuntos dinámicos pueden ser agrupadas en dos categorías:

- *Consultoras(queries)*: La operación simplemente devuelve algún tipo de información sobre el conjunto.
- *Modificadoras*: Modifican el conjunto.

A continuación, se enumeran las operaciones más usuales que se realizan sobre conjuntos dinámicos:

Dado un conjunto  $S$ , y un elemento  $x$  tal que  $clave(x) = k$ .

### 1. Consultoras:

- $Buscar(S, k)$ : Devuelve el elemento  $x \in S$  tal que  $clave(x) = k$ ; si  $x \notin S$  devuelve un valor especial (valor nulo).
- $Vacío(S)$ : Indica si el conjunto  $S$  es vacío o no.  
Las siguientes operaciones se pueden realizar si en el conjunto  $S$  existe una relación de orden total entre las claves de sus *miembros*.
- $Mínimo(S)$ : Devuelve el elemento  $x$  con la clave  $k$  más pequeña del conjunto  $S$ .
- $Máximo(S)$ : Devuelve el elemento  $x$  con la clave  $k$  más grande del conjunto  $S$ .
- $Predecesor(S, x)$ : Devuelve el elemento de clave inmediatamente inferior a la de  $x$ , o un valor especial si  $x$  es el elemento de clave mínima.
- $Sucesor(S, x)$ : Devuelve el elemento de clave inmediatamente superior a la de  $x$ , o un valor especial si  $x$  es el elemento de clave máxima.

### 2. Modificadoras:

- $Insertar(S, x)$ : Añade al conjunto  $S$  el elemento  $x$ .
- $Borrar(S, x)$ : Elimina del conjunto  $S$  el elemento  $x$ .
- $Crear(S)$ : Crea el conjunto  $S$  vacío.



## 6.2. Tablas de dispersión o tablas *Hash*

Muchas aplicaciones requieren la utilización de un conjunto sobre el que realizar, frecuentemente, las siguientes operaciones: insertar un elemento en el conjunto, borrar un elemento del conjunto, y determinar la pertenencia de un elemento al conjunto. Un conjunto que permita, principalmente, estas operaciones, se denomina **diccionario**. A continuación veremos cómo es posible representar eficientemente un *diccionario*.

### 6.2.1. Tablas de direccionamiento directo

El *direccionamiento directo* es una técnica que funcionará bien cuando el universo  $U$  de elementos que puede contener el conjunto, sea razonablemente pequeño. Supongamos que cada *miembro* del conjunto se identifica mediante una clave que es única; es decir, no existen dos *miembros* del conjunto que se identifiquen por la misma clave. Mediante *direccionamiento directo*, el conjunto se representa utilizando un vector de tamaño igual a la talla del universo  $|U|$   $T[0, \dots, |U|-1]$ , en el que cada posición  $k$  del vector referencia al miembro  $x$  con clave  $k$ .

Una posible representación será, para todo elemento  $x$  de clave  $k$  perteneciente al conjunto, almacenar en la posición  $k$  del vector un puntero a una estructura donde se guarda la información del elemento  $x$  (ver figura 6.1(a)).

Las operaciones sobre el diccionario, en esta implementación, son triviales; si  $T$  es el vector,  $x$  un elemento y la operación  $clave(x)$  proporciona la clave  $k$  del elemento  $x$ ; entonces:

- insertar( $T, x$ ): insertar  $x$  en  $T[clave(x)]$ . La inserción supondrá crear un nuevo nodo con la información del elemento  $x$  (clave, información asociada), y almacenar en  $T[clave(x)]$  la dirección de memoria del nuevo nodo creado.
- buscar( $T, k$ ): devolver el valor almacenado en  $T[k]$ . Si  $x$  no pertenece al conjunto el valor almacenado en  $T[k]$  será el valor *NULO* (*NULL*); en caso contrario, se devolverá la dirección de memoria de la estructura donde se guarda la información de  $x$ .
- borrar( $T, x$ ): borrar  $x$  y asignar a  $T[clave(x)]$  el valor *NULO* (*NULL*). El borrado de  $x$  se realizará liberando el espacio de memoria ocupado por la estructura que almacena la información de  $x$ .

Cada una de estas operaciones tiene un coste  $O(1)$ .

Una variante sobre esta representación, sería almacenar la información del elemento en el propio vector (ver figura 6.1(b)). Por lo tanto, en cada posición del vector ya no se almacenaría un puntero a una estructura donde se guarda la

información del elemento, sino que esta información se tendría directamente en la posición del vector. En esta representación, puede resultar innecesario almacenar en la estructura cuál es la clave del elemento, ya que la posición que ocupa el elemento en el vector indica cuál es su clave. Si las claves no son almacenadas, se tendrá que habilitar un mecanismo que permita distinguir cuándo la posición del vector está vacía u ocupada por un elemento del conjunto.

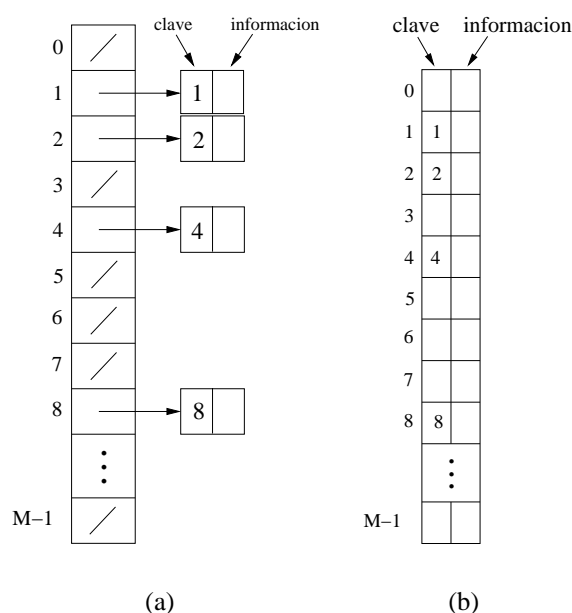


Figura 6.1: Representación de un diccionario utilizando direccionamiento directo. En la variante (a) la información del elemento se almacena en una estructura externa al vector; mientras que en la variante (b) la información del elemento se almacena en el propio vector

### 6.2.2. Tablas de dispersión o tablas *Hash*

Como acabamos de ver, en el direccionamiento directo, cada posición del vector almacena la información de un único elemento del universo  $U$ , por lo que, si el universo  $U$  es demasiado grande o infinito, será imposible almacenar en memoria un vector de tamaño  $|U|$ . Otro inconveniente añadido es que, normalmente, el número de elementos que, en un momento dado, sean *miembros* del conjunto, será relativamente pequeño en comparación con el número total de elementos del universo  $U$  ( $|U|$ ), por lo que gran cantidad de espacio ocupado por el vector será desaprovechado.

La utilización de una tabla de dispersión o tabla *hash* solventará en gran medida estos problemas, ya que el tamaño del vector estará limitado a un tamaño determinado  $M: T[0, \dots, M - 1]$ . Para todo elemento  $x$  perteneciente al universo  $U$ , la posición que ocupa el elemento en el vector se obtendrá tras aplicar sobre su clave  $k$  una *función de dispersión* o *hashing*, que convertirá el valor de  $k$  en un valor entre 0 y  $M - 1$ .

**Definición:** Una *tabla de dispersión* o *tabla hash* es una representación de conjuntos dinámicos en la que, la posición que ocupa en el vector un elemento  $x$  con clave  $k$ , se obtiene tras aplicar una *función de dispersión* o *hashing*  $h$  sobre la clave  $k: h(k)$ . De esta forma, la función de dispersión  $h$  transforma el universo de claves  $K$  en un valor comprendido entre 0 y  $M - 1$  (ver figura 6.2):

$$h : K \longrightarrow \{0, 1, \dots, M - 1\}$$

A cada posición del vector le denominaremos cubeta; y diremos que un elemento  $x$  con clave  $k$  *se dispersa* en la cubeta  $h(k)$  de la *tabla de dispersión*.

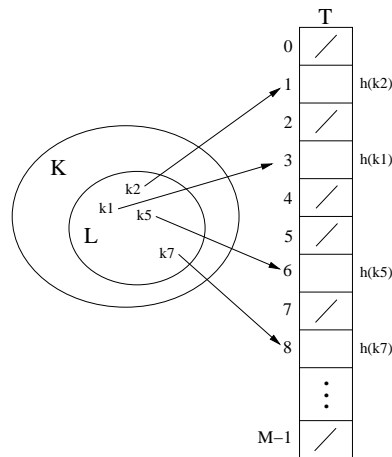


Figura 6.2: Ejemplo de cómo la función de dispersión  $h$  transforma las claves en posiciones (cubetas) de la tabla  $T$ . El conjunto  $K$  representa el universo de claves; y el conjunto  $L$  las claves de los elementos que, en un momento dado, pertenecen al conjunto.

Lógicamente, debido a que la talla del universo de claves  $K$  será mayor que la talla  $M$  de la tabla, podrá ocurrir que dos o más claves se dispersen en una misma cubeta, lo que provocará una *colisión* (ver figura 6.3). Dos métodos que resolverán eficientemente las colisiones son: por *encadenamiento*, por *direccionamiento abierto* [Cormen]. Nosotros estudiaremos uno de los métodos: resolución de colisiones *por encadenamiento*.

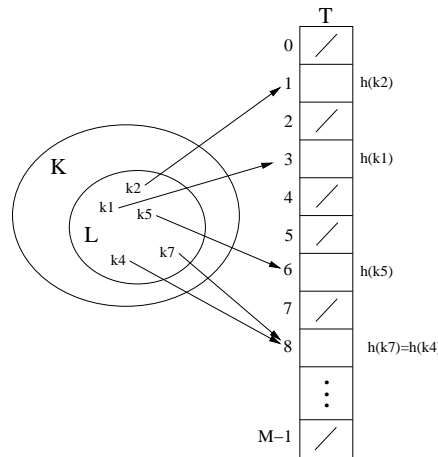


Figura 6.3: Ejemplo de colisión. Las claves  $k_7$  y  $k_4$  se dispersan en la misma cubeta debido a que  $h(k_7) = h(k_4) = 8$ .

### Resolución de colisiones *por encadenamiento*

**Estrategia:** Los elementos que se dispersen en una misma cubeta se organizan mediante una lista enlazada. Cada cubeta  $j$  de la tabla contiene un puntero a la cabeza de una lista enlazada, que contiene los elementos que han sido dispersados en la cubeta  $j$ ; si la cubeta no tiene elementos contendrá el valor *NULO* (*NULL*) (ver figura 6.4).

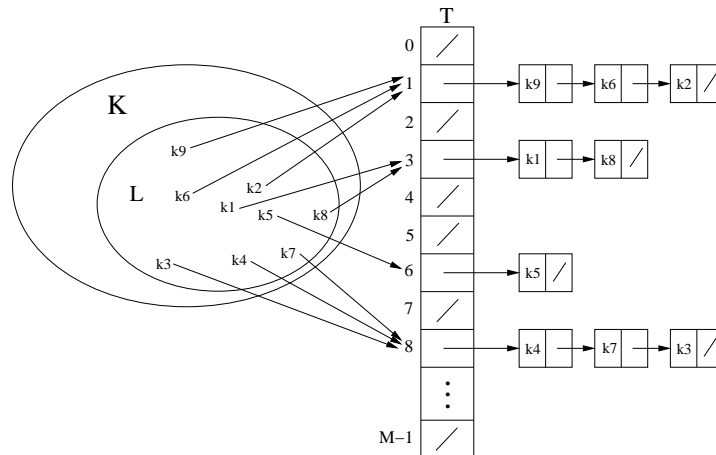


Figura 6.4: Ejemplo en el que se muestra cómo se resuelven las colisiones *por encadenamiento*. Por ejemplo,  $h(k_9) = h(k_6) = h(k_2) = 1$ .

Las operaciones sobre un *diccionario* representado mediante una tabla de dispersión  $T$ , son fáciles de realizar cuando las colisiones se resuelven *por encade-*

*namiento.*

Dado una tabla  $T$ , y un elemento  $x$ , tal que  $clave(x) = k$ :

- Insertar( $T, x$ ): inserta el elemento  $x$  en la cabeza de la lista apuntada por  $T[h(clave(x))]$ .
- Buscar( $T, k$ ): busca un elemento con clave  $k$  en la lista  $T[h(k)]$ .
- Borrar( $T, x$ ): borra  $x$  de la lista encabezada por  $T[h(clave(x))]$ .

### **Análisis del coste de las operaciones**

Para analizar el coste de cada una de las operaciones, será necesario introducir previamente el concepto de *factor de carga*. Dada una tabla de dispersión  $T$  con  $m$  cubetas y que almacena  $n$  elementos, se define como *factor de carga*  $\alpha$  al valor  $n/m$ ; es decir, al número medio de elementos almacenados en cada cubeta de la tabla. Además asumiremos que, dada un clave  $k$ , el valor de dispersión  $h(k)$  puede ser calculado en un tiempo  $O(1)$ .

#### **Insertar**

La inserción de un elemento consiste en obtener la cubeta que le corresponde:  $O(1)$ , e insertarlo en la cabeza de la lista:  $O(1)$ . Por lo tanto, el coste de la inserción de un elemento en la tabla es  $O(1)$ .

#### **Buscar**

El coste temporal de la búsqueda de un elemento en la tabla, vendrá determinado por el número de elementos que será necesario examinar, hasta encontrar o no, el elemento buscado. O lo que es lo mismo, por la longitud de la lista que encabeza la cubeta en la que se dispersa el elemento buscado.

El peor caso, por tanto, que puede producirse, ocurrirá cuando los  $n$  elementos que contiene la tabla hayan sido dispersados en la misma cubeta. En este caso, una cubeta de la tabla contendrá una lista enlazada de  $n$  elementos, y el resto de cubetas estarán vacías. Por lo tanto, en el peor caso, el coste de buscar un elemento en la tabla, será igual al coste de buscar un elemento en una lista de  $n$  elementos:  $O(n)$ .

Como se puede observar, el coste de buscar un elemento en la tabla dependerá de cómo distribuya la función de dispersión  $h$  los elementos entre las cubetas. En general, asumiremos que cualquier elemento tiene la misma probabilidad de dispersarse en cualquiera de las  $m$  cubetas; a esta asunción se le denomina *dispersión uniforme simple*.

En la estimación del caso medio deberemos considerar dos casos: a) el elemento buscado no se encuentra en la tabla; b) el elemento buscado sí que es encontrado.

**Teorema:** En una tabla de dispersión en la que las colisiones se resuelven por encadenamiento, una búsqueda sin éxito tiene una complejidad temporal  $\Theta(1+\alpha)$ , en promedio, bajo la asunción de *dispersión uniforme simple* [Cormen].

**Demostración:**

Si el elemento no se encuentra en la tabla, habrá que examinar todos los elementos que contenga la lista de la cubeta en la que se dispersa el elemento buscado. Bajo la asunción de *dispersión uniforme simple*, la longitud media de cada lista vendrá determinada por el factor de carga  $\alpha = n/m$ . Por lo tanto, el tiempo total requerido será  $\Theta(1 + \alpha)$  (incluyendo el tiempo para calcular  $h(k) = O(1)$ ).

**Teorema:** En una tabla de dispersión en la que las colisiones se resuelven por encadenamiento, una búsqueda con éxito tiene una complejidad temporal  $\Theta(1 + \alpha)$ , en promedio, bajo la asunción de *dispersión uniforme simple* [Cormen].

**Demostración:**

Para la demostración, asumiremos que cada vez que se inserta un nuevo elemento en la tabla, se coloca al final de la lista de la cubeta en la que se dispersa (el coste que vamos a estimar será el mismo si la inserción se realizara en la cabeza de la lista). Para estimar el coste medio, deberemos estimar el coste que supone buscar cada uno de los  $n$  elementos que contiene la tabla, y hallar la media  $\frac{1}{n}$ . El coste de la búsqueda de un elemento dependerá de la posición que ocupa en la lista de la cubeta en la que se dispersa. A su vez, la posición que ocupa en la lista, dependerá del factor de carga que existía en la tabla cuando fue insertado; por ejemplo, si el elemento buscado fue insertado cuando existían  $i - 1$  elementos en la tabla, su posición en la lista será:  $\frac{i-1}{m} + 1$  (debido a que asumimos que se inserta al final de la lista). Por lo tanto, encontrar el elemento que fue el  $i$  elemento insertado en la tabla costará:  $c_i = \frac{i-1}{m} + 1$ . De esta forma, cuando en la tabla se han insertado  $n$  elementos, el número medio de elementos que será necesario examinar en una búsqueda con éxito será:

$$\begin{aligned}
\frac{1}{n} \sum_{i=1}^n c_i &= \frac{1}{n} \sum_{i=1}^n \left( \frac{i-1}{m} + 1 \right) \\
&= \frac{1}{n} \left( \sum_{i=1}^n \frac{i-1}{m} + \sum_{i=1}^n 1 \right) \\
&= \frac{1}{n} \left( \sum_{i=1}^n \frac{i-1}{m} + n \right) \\
&= 1 + \frac{1}{n} \left( \sum_{i=1}^n \frac{i-1}{m} \right) \\
&= 1 + \frac{1}{nm} \sum_{i=1}^n (i-1) \\
&= 1 + \left( \frac{1}{nm} \right) \left( \frac{(n-1)n}{2} \right) \\
&= 1 + \frac{n-1}{2m} \\
&= 1 + \frac{n}{2m} - \frac{1}{2m} \\
&= 1 + \frac{\alpha}{2} - \frac{1}{2m}
\end{aligned}$$

Por lo tanto, el tiempo total requerido para encontrar con éxito un elemento en la tabla (incluyendo el tiempo para calcular la cubeta en la que se dispersa  $h(k) = O(1)$ ) es  $\Theta(2 + \alpha/2 - 1/2m) = \Theta(1 + \alpha)$ .

Esto significa que el tiempo total que se requiere para buscar en promedio un elemento en la tabla, dependerá del factor de carga  $\alpha$ . Por lo tanto, si el número de cubetas es, al menos, proporcional al número de elementos de la tabla, tendremos que  $n = O(m)$  y, en consecuencia,  $\alpha = n/m = O(m)/m = O(1)$ . En definitiva, podemos concluir que el tiempo de búsqueda de un elemento en una tabla de dispersión es, en promedio, constante.

## Borrar

La eliminación de un elemento de la tabla supone que previamente debe ser buscado. Por lo tanto, el coste temporal de la operación será esencialmente el mismo que el de la búsqueda:  $\Theta(1 + \alpha)$ .

Como se ha demostrado, si se utiliza una tabla de dispersión para representar un diccionario, todas las operaciones que necesitamos efectuar sobre él (inserción,

búsqueda y borrado) pueden realizarse, en promedio, con un **coste temporal constante**  $O(1)$ .

## Funciones de dispersión

Una función de dispersión “ideal”, debería satisfacer la asunción de *dispersión uniforme simple*: cada elemento tiene la misma probabilidad de dispersarse en cualquiera de las  $m$  cubetas. En la práctica, será difícil encontrar alguna función de dispersión que cumpla esta asunción. De cualquier forma, podremos utilizar funciones de dispersión que dispersen de forma aceptable los elementos entre las cubetas.

Asumiremos que el universo de claves es el conjunto de números naturales  $\mathbb{N} = \{0, 1, 2, \dots\}$ . Si la clave que identifica al elemento no es un número natural, habrá que encontrar una forma de representarla como un número natural. Por ejemplo, una clave que sea una cadena de caracteres, puede ser representada como un número natural, combinando de algún modo la representación ASCII (numérica) de sus caracteres.

## El método de la división

En este método, para obtener una función de dispersión  $h$ , la clave<sup>1</sup>  $k$  se puede transformar en un valor entre 0 y  $m - 1$ , tomando el resto de la división de  $k$  dividido por  $m$ , donde  $m$  es el número de cubetas:

$$h(k) = k \bmod m$$

**Ejemplo:** Si la clave  $k = 100$  y la tabla de dispersión es de talla  $m = 12$ ; entonces:  $h(100) = 100 \bmod 12 = 4$ . El elemento con clave 100 se dispersaría en la cubeta 4 de la tabla. Obsérvese, por ejemplo, que el elemento con clave 16:  $h(16) = 16 \bmod 12 = 4$  se dispersaría en la misma cubeta.

**Aspecto crítico:** La elección del valor  $m$ . Para que la función de dispersión tenga un buen comportamiento, es decir, disperse de forma aceptable los elementos entre las cubetas, será recomendable escoger un valor de  $m$  que sea primo y no esté próximo a una potencia de 2.

**Ejemplo:** Supongamos que queremos almacenar en la tabla de dispersión 2000 cadenas de caracteres, y queremos que el factor de carga no sea mayor que 3. En este caso, el mínimo número de cubetas necesario sería  $2000/3 = 666,6$ . Buscamos un número  $m$  próximo a 666 que sea primo y no esté cercano a una potencia de 2: 701. Por lo tanto la función de dispersión sería:  $h(k) = k \bmod 701$ .

---

<sup>1</sup>A partir de ahora siempre consideraremos que la clave es un número natural:  $k \in \mathbb{N}$ .



## El método de la multiplicación

En este método, para obtener una función de dispersión  $h$ , la clave  $k$  se puede transformar en un valor entre 0 y  $m - 1$ , en dos pasos:

1. Se multiplica la clave  $k$  por una constante en el rango  $0 < A < 1$  y se extrae del resultado únicamente la parte decimal. Esta operación puede expresarse como:

$$(k \cdot A) \bmod 1 \quad (6.1)$$

2. Se multiplica el valor obtenido en (6.1) por el valor de  $m$ , y se extrae el valor entero más próximo por debajo al número obtenido. Por lo tanto, la función de dispersión  $h$  quedaría definida como:

$$h(k) = \lfloor m ((k \cdot A) \bmod 1) \rfloor$$

En las funciones de dispersión basadas en el método de la multiplicación, el valor de  $m$  no es crítico. Se suele tomar como valor de  $m$  una potencia de 2 para facilitar el cómputo de la función:  $m = 2^p$ , para algún valor entero  $p$ .

Aunque el método sirve para cualquier constante  $A$ , en algunos trabajos se sugiere utilizar el valor de  $A \approx (\sqrt{5} - 1)/2 = 0,618033988750$ .

## Ejemplos de funciones de dispersión sobre cadenas de caracteres

Como se ha comentado anteriormente, en el caso de que las claves sean cadenas de caracteres, deberá realizarse una conversión de la cadena en un número natural. Para ello, dada una cadena  $x$  de  $n$  caracteres,  $x_i$  simbolizará el código *ASCII* del carácter que ocupa la posición  $i$  en la cadena,  $0 \leq i \leq n - 1$ , ( $x = x_0x_1x_2 \dots x_{n-1}$ ). A continuación, veremos algunos ejemplos de funciones de dispersión aplicadas sobre cadenas de caracteres.

1. Ejemplos de funciones de dispersión basadas en el método de la división.
  - Función 1: Se transforma la clave alfabética en una clave numérica sumando los códigos *ASCII* de cada carácter. Se caracteriza porque aprovecha toda la clave alfanumérica para realizar la transformación.

$$h(x) = \left( \sum_{i=0}^{n-1} x_i \right) \bmod m$$

**Inconvenientes:**

- Si el tamaño de  $T$  es grande, por ejemplo  $m = 10007$ , no se distribuirán bien las claves, ya que si todas las cadenas son de longitud menor o igual que, por ejemplo, 10, el máximo valor que se podría obtener en una transformación de la cadena en un número natural sería:  $255 \cdot 10 = 2550$  (suponiendo que la cadena de longitud 10, estuviera formada por el mismo carácter cuya representación ASCII fuera el valor máximo: 255). Por lo tanto, los valores de la función  $h(x)$  tomarán valores entre 0 y máximo 2550. Este hecho provocará que las cubetas desde la posición 2551 hasta 10006 estén vacías.
  - El orden en que aparecen los caracteres en la cadena no se tiene en cuenta, por lo que, por ejemplo  $h(\text{"cosa"})=h(\text{"saco"})$ . Esto significa que claves que estén formadas por los mismos caracteres serán dispersadas en las mismas cubetas.
- **Función 2:** Se transforma la clave alfabética en una clave numérica, utilizando, únicamente, los tres primeros caracteres de la clave, pero considerando que son un valor numérico en una determinada base (256 en este caso).

$$h(x) = \left( \sum_{i=0}^2 x_i 256^i \right) \bmod m$$

**Inconvenientes:**

- Al utilizar únicamente los tres primeros caracteres de la cadena, todas las cadenas cuyos primeros tres caracteres coincidan serán dispersadas en las mismas cubetas.  
Por ejemplo,  $h(\text{"clase"})=h(\text{"clarinete"})=h(\text{"clan"})$ .
- **Función 3:** Similar a la función 2 pero considerando toda la cadena.

$$h(x) = \left( \sum_{i=0}^{n-1} x_i 256^{((n-1)-i)} \right) \bmod m$$

**Inconvenientes:**

- El cálculo de la función  $h$  es costoso.

2. Ejemplos de funciones de dispersión basadas en el método de la multiplicación.

- Funcion 4: Se transforma la clave alfabética en una clave numérica, sumando los códigos *ASCII* de cada caracter considerando que son un valor numérico en base 2.

$$h(x) = \lfloor m \left( \left( \sum_{i=0}^{n-1} x_i 2^{((n-1)-i)} \right) \cdot A \right) \bmod 1 \rfloor$$

### Evaluación empírica

En la figura 6.5 se muestra una comparativa empírica del comportamiento de cada una de las funciones, para un número de elementos  $n = 3975$  y un número de cubetas  $m = 2003$ . Para cada función, se muestran dos tipos de gráficas:

1. En las gráficas de la primera columna se representa, para cada cubeta, el número de elementos que contiene. Puede apreciarse que la función 1 y 2 distribuyen los elementos deficientemente, ya que existen cubetas con una gran concentración de elementos. Obsérvese que las funciones 3 y 4 consiguen distribuir los elementos de una manera más uniforme entre las cubetas: la función 3 no contiene más de 9 elementos por cubeta con una desviación típica de 1,43; mientras que la función 4, únicamente presenta una cubeta con 14 elementos, mientras que el resto no supera los 10 elementos por cubeta (desviación típica: 1,8).
2. En cada una de las gráficas de la segunda columna se representa, el número de cubetas (eje  $y$ ) que tienen exáctamente un número determinado de elementos (eje  $x$ ). Por ejemplo, puede observarse que tanto la función 1 como la función 2, presentan un número elevado de cubetas que contienen más de, por ejemplo, 10 elementos; mientras que para la función 3, la gran mayoría de cubetas contienen un número menor de 5 elementos y ninguna más de 9, y para la función 4, se cumple que la gran mayoría de cubetas contienen un número menor de 7 elementos, y sólo una cubeta contiene un número máximo de 14.

A la vista de los resultados obtenidos, se aprecia claramente que la función 3 y 4 presentan un mejor comportamiento. Esto se debe a que ambas funciones aprovechan mejor la información de la cadena (clave). Las funciones 3 y 4 utilizan todos los caracteres de la cadena y, además, consideran la posición que ocupan; sin embargo, la función 1, aunque utiliza todos los caracteres de la cadena no tiene en cuenta su posición; y la función 2 únicamente utiliza la información de los tres primeros caracteres.

En las figuras 6.6 y 6.7 se muestra de qué forma afecta, al buen comportamiento de una función de dispersión basada en el método de la división, la modificación

del número de cubetas  $m$ : el comportamiento de la función de dispersión 3 (basada en el método de la división) empeora al variar el valor de  $m$  (de 2003 a 2000), mientras que a la función 4 (basada en el método de la multiplicación) no le afecta esta variación en el número de cubetas manteniendo el buen comportamiento.

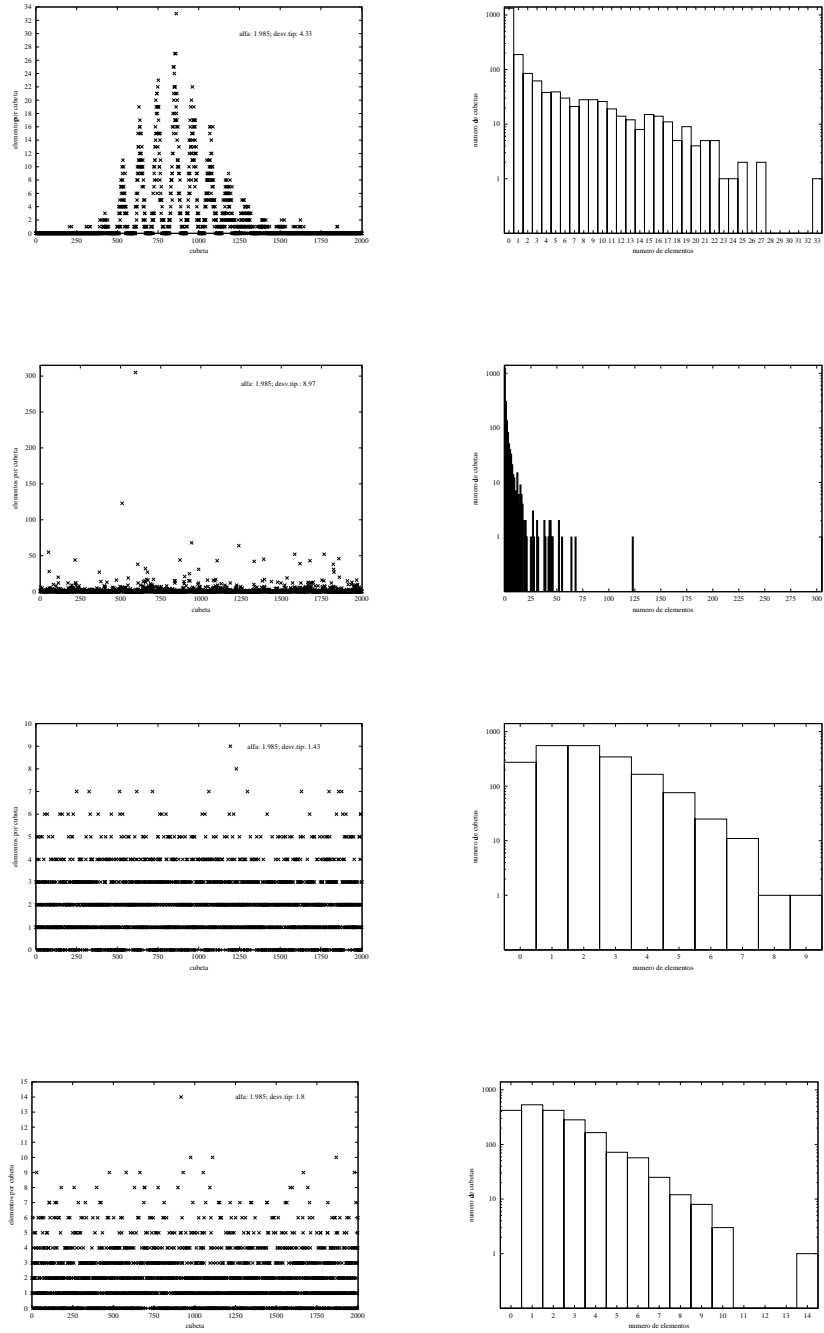


Figura 6.5: Distribución de los elementos entre las cubetas para cada una de las funciones de dispersión 1, 2, 3 y 4, respectivamente, para un número de cubetas  $m = 2003$  y un número de elementos  $n = 3975$ . Las gráficas de la izquierda muestran, para cada cubeta, el número de elementos que contiene. Las gráficas de la derecha muestran, en el eje  $y$ , el número de cubetas (en escala logarítmica) que contienen exactamente el número de elementos que se representa en el eje  $x$ .

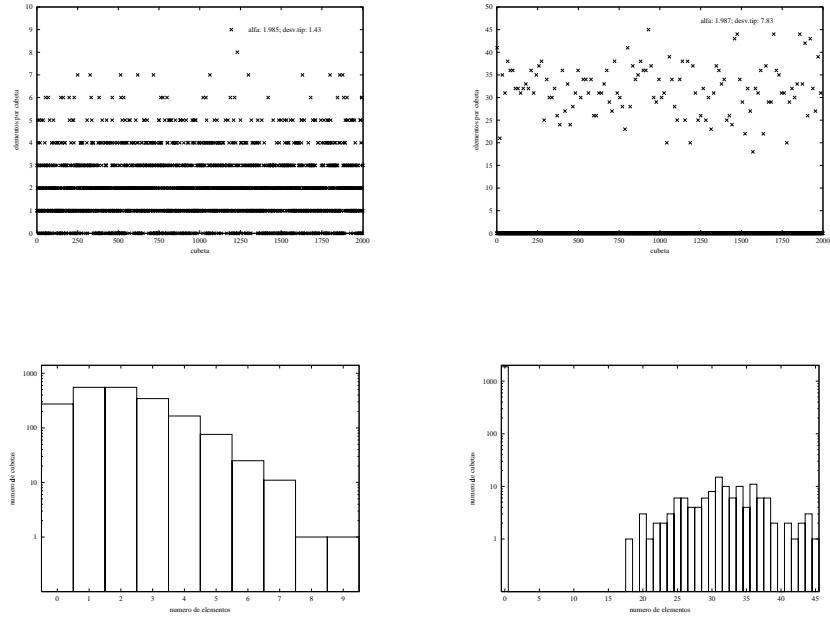


Figura 6.6: Comparativa en la que se muestra cómo afecta la elección del valor de  $m$  a la función 3 basada en el método de la división;  $m = 2003$  (izquierda),  $m = 2000$  (derecha).

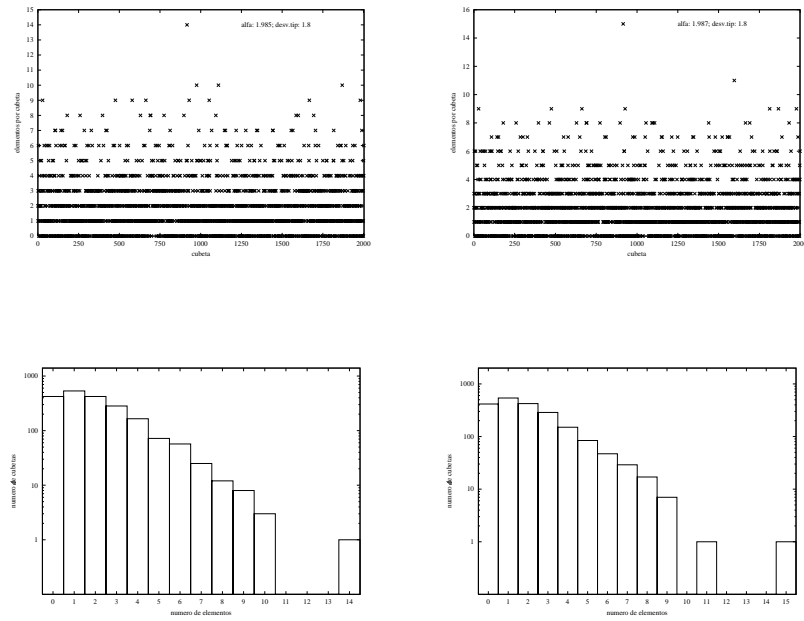


Figura 6.7: Comparativa en la que se muestra que la elección del valor de  $m$  no afecta al comportamiento de la función 4 basada en el método de la multiplicación;  $m = 2003$  (izquierda),  $m = 2000$  (derecha).

**Ejercicio:** Se disponen de las siguientes definiciones de tipos, constantes y variables, para declarar y manipular tablas de dispersión:

```
#define NCUB ...
```

```
typedef struct snodo {
    char *pal;
    struct snodo *sig;
} nodo;
typedef nodo *Tabla[NCUB];
```

```
Tabla T1, T2, T3;
```

Además, las siguientes funciones están disponibles y pueden utilizarse cuando se considere oportuno.

```
/* Inicializa una tabla vac\{'\i}a */
void crea_Tabla(Tabla T)
```

```
/* Inserta la palabra w en la tabla T */
void inserta(Tabla T, char *pal)
```

```
/* Devuelve un puntero al nodo que almacena la palabra */
/* pal, o NULL si no la encuentra */
nodo *buscar(Tabla T, char *pal)
```

Escribir una función que cree una tabla *T3* con los elementos pertenecientes a la intersección de los conjuntos almacenados en las tablas *T1* y *T2*.

```
void interseccion(Tabla T1, Tabla T2, Tabla T3) {
    int i;
    nodo *aux;

    crea_Tabla(T3);
    for(i=0; i<NCUB; i++) {
        aux = T1[i];
        while (aux != NULL) {
            if ( buscar(T2,aux->pal) != NULL )
                inserta(T3,aux->pal);
            aux = aux->sig;
        }
    }
}
```

## 6.3. Árboles binarios de búsqueda

**Definición:** Un árbol binario de búsqueda es un árbol binario que puede estar vacío, o si no está vacío cumple las siguientes propiedades:

- Cada nodo contiene una clave y no existen dos nodos con la misma clave.
- Para cada nodo  $n$  de un árbol binario de búsqueda, se cumple que, si su subárbol izquierdo no es vacío, todas las claves almacenadas en los nodos de su subárbol izquierdo son menores que la clave almacenada en el nodo  $n$ .
- Para cada nodo  $n$  de un árbol binario de búsqueda, se cumple que, si su subárbol derecho no es vacío, todas las claves almacenadas en los nodos de su subárbol derecho son mayores que la clave almacenada en el nodo  $n$ .

Los árboles binarios de búsqueda, son una estructura de datos que soportan muchas de las operaciones que se pueden realizar sobre conjuntos dinámicos; como son: inserción, borrado, búsqueda, mínimo, máximo, predecesor y sucesor. Un árbol binario de búsqueda puede ser utilizado para representar un diccionario o una cola de prioridad. En la figura 6.8 se muestran dos ejemplos de árboles binarios de búsqueda; mientras que en la figura 6.9 se muestran dos árboles binarios que no cumplen las propiedades para ser considerados como árboles binarios de búsqueda.

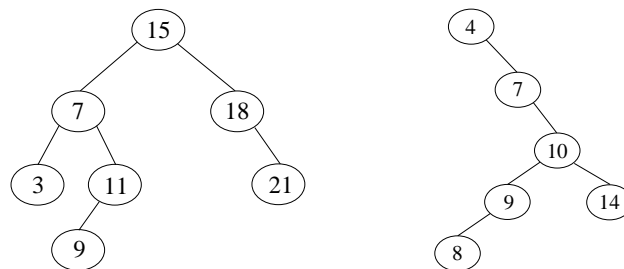


Figura 6.8: Ejemplos de árboles binarios de búsqueda. Para cualquier nodo  $n$  de clave  $x$ , las claves de los nodos del subárbol izquierdo de  $n$  son menores que  $x$ , y las claves de los nodos del subárbol derecho de  $n$  son mayores que  $x$ .



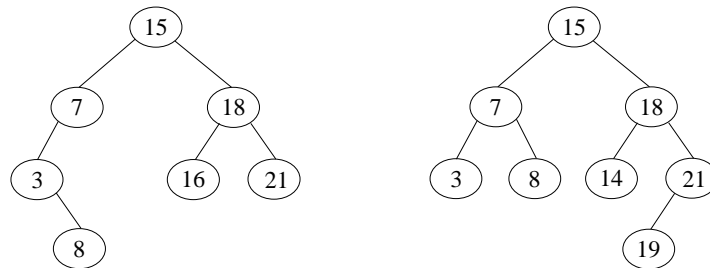


Figura 6.9: Ejemplos de árboles binarios que no son de búsqueda. En el primer árbol, el nodo de clave 8 pertenece al subárbol izquierdo del nodo de clave 7. En el segundo árbol, el nodo de clave 14 pertenece al subárbol derecho del nodo de clave 15.

### 6.3.1. Representación de árboles binarios de búsqueda

Los árboles binarios de búsqueda suelen representarse mediante variables dinámicas, de tal forma que cada nodo del árbol es una estructura con los siguientes campos:

- clave: clave del nodo.
- hijo izquierdo: puntero a la estructura que representa al nodo que es hijo izquierdo.
- hijo derecho: puntero a la estructura que representa al nodo que es hijo derecho.

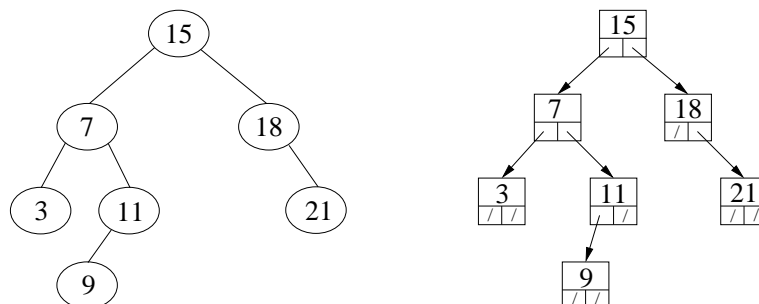


Figura 6.10: Representación de un árbol binario de búsqueda mediante una estructura enlazada de variables dinámicas.

La siguiente definición de tipos en C, nos serviría para definir una estructura de datos que se ajuste a esta representación.

```
typedef ... tipo_baseT;

typedef struct snodo {
    tipo_baseT clave;
    struct snodo *hizq, *hder;
} abb;
```

Una posible variante a este tipo de representación, como ya se indicó en el tema 2 cuando se vio la representación de árboles binarios, sería almacenar en otro campo de la estructura, un puntero al nodo padre (ver figura 6.11).

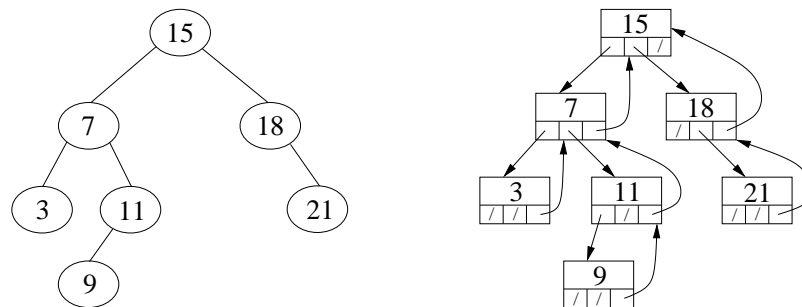


Figura 6.11: Representación de un árbol binario de búsqueda mediante una estructura enlazada de variables dinámicas, en la que también se mantiene un puntero al nodo padre de cada nodo.

La siguiente definición de tipos en C, nos serviría para definir una estructura de datos que se ajuste a esta representación.

```
typedef ... tipo_baseT;

typedef struct snodo {
    tipo_baseT clave;
    struct snodo *hizq, *hder, *padre;
} abb;
```

### 6.3.2. Altura máxima y mínima de un árbol binario de búsqueda

Dado cualquier árbol binario de búsqueda, su altura será la mínima posible, en el caso de que todos los niveles del árbol contengan el máximo número de nodos posibles, excepto, si cabe, el último nivel. En ese caso, si el árbol contiene  $n$  nodos, su altura será  $\lfloor \log_2 n \rfloor$ . Por lo tanto, la altura mínima de un árbol binario de búsqueda es  $O(\log n)$ .

De la misma forma, dado cualquier árbol binario de búsqueda, su altura será la máxima posible, en el caso de que todos los niveles del árbol contengan un único nodo. En ese caso su altura será  $n - 1$ . Por lo tanto, la altura máxima de un árbol binario de búsqueda es  $O(n)$ .

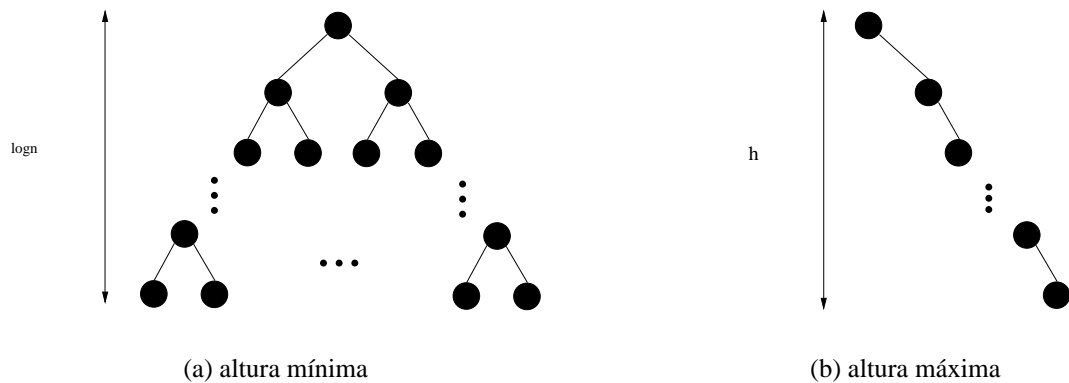


Figura 6.12: Altura mínima (a) y máxima (b) de un árbol binario de búsqueda.

Cuando analicemos cada una de las operaciones que se pueden realizar sobre un árbol binario de búsqueda, veremos que el coste temporal de cada una de ellas es  $O(h)$ , donde  $h$  es la altura del árbol. Por lo tanto, si la altura del árbol es mínima, el coste temporal de cada operación será  $O(\log n)$ ; por el contrario, si la altura del árbol es máxima, cada una de las operaciones tendrá un coste temporal de  $O(n)$ . En general, la altura de un árbol binario de búsqueda construido aleatoriamente es  $O(\log n)$ , por lo que cada operación podrá realizarse con un coste temporal de  $O(\log n)$ .

### 6.3.3. Recorrido de árboles binarios de búsqueda

Tal como se vio en el tema 2, una acción  $P$  puede ser realizada sobre todos los nodos de un árbol, siguiendo tres tipos de recorrido: recorrido en orden previo (*preorden*), recorrido en orden simétrico (*inorden*) y recorrido en orden posterior

(*postorden*). Por lo tanto, un árbol binario de búsqueda podrá recorrerse igualmente de cualquiera de estas tres formas.

Sin embargo, vamos a centrarnos en el recorrido en orden simétrico (*inorden*), ya que, debido a las características de un árbol binario de búsqueda, si la acción *P* fuera imprimir la clave del nodo, *las claves de un árbol binario de búsqueda se imprimirán de forma ordenada (de menor a mayor) siguiendo el recorrido en inorden*. Esto es así, debido a que en el recorrido en *inorden*, la operación *P* sobre un nodo *n* se realiza después de recorrer en *inorden* su subárbol izquierdo y antes de recorrer en *inorden* su subárbol derecho. Por lo tanto, como siempre se cumple que, para todo nodo, su clave es mayor que todas las claves del subárbol izquierdo, y menor que todas las claves del subárbol derecho, la clave de cada uno de los nodos se imprime justamente después de imprimir todas las que son menores y antes de imprimir todas las que son mayores.

La siguiente función realiza un recorrido en *inorden* de un árbol binario de búsqueda:

```
void inorden(abb *T) {
    if (T != NULL) {
        inorden(T->hizq);
        printf("%d ", T->clave);
        inorden(T->hder);
    }
}
```

### Ejercicio

Realizar la traza del algoritmo “*inorden*” para el árbol de la figura 6.10:

La solución es: <3, 7, 9, 11, 15, 18, 21>

### 6.3.4. Búsqueda de un elemento en un árbol binario de búsqueda

La operación más común que se realiza sobre un árbol binario de búsqueda es buscar una clave almacenada en el árbol. A continuación se presenta una función que dado un puntero *T* al nodo raíz de un árbol binario de búsqueda y una clave *x* a buscar, devuelve un puntero al nodo con clave *x* si se encuentra, o NULL en caso contrario.

```

abb *abb_buscar(abb *T, tipo_baseT x) {
    while ( (T != NULL) && (x != T->clave) )
        if (x < T->clave)
            T = T->hizq;
        else
            T = T->hder;
    return(T);
}

```

La función comienza la búsqueda desde el nodo raíz y va trazando un camino descendente a través del árbol, de forma que cada vez que se sitúa en un nuevo nodo  $n$ , compara la clave del nuevo nodo con la clave buscada; si las claves son iguales la búsqueda finaliza con éxito, mientras que si son diferentes, si la clave buscada es menor, la búsqueda continúa por el subárbol izquierdo de  $n$ , ya que la característica de los árboles binarios de búsqueda implica que el elemento buscado no puede encontrarse en el subárbol derecho. De forma análoga, si la clave buscada es mayor que la clave del nodo  $n$ , la búsqueda continúa por el subárbol derecho. Si el elemento buscado no es encontrado, la búsqueda finaliza cuando se llega a un subárbol vacío ( $T = NULL$ ).

### Ejercicio

Realizar una versión recursiva de la anterior función de búsqueda de una clave  $x$  en un árbol binario de búsqueda:

```

abb *abb_buscar(abb *T, tipo_baseT x) {
    if (T != NULL)
        if (x == T->clave)
            return(T);
        else if (x < T->clave) return(abb_buscar(T->hizq,x));
        else return(abb_buscar(T->hder,x));
    return(T);
}

```

El número de nodos que es necesario examinar hasta encontrar o no el elemento buscado, determinará el coste del algoritmo. Debido a que el camino de búsqueda que se va trazando en el árbol, examina un nodo por nivel, como máximo será necesario examinar todos los niveles del árbol hasta encontrar o no el elemento buscado. Por lo tanto, **el coste temporal de la búsqueda de un elemento en un árbol binario de búsqueda será  $O(h)$ , donde  $h$  es la altura del árbol.**

### Ejemplo

En la figura 6.13, se muestra un ejemplo del funcionamiento del algoritmo de búsqueda para la llamada inicial:  $nodo = abb\_buscar(T, x)$ , donde  $x$  es un elemento con clave 11. En este ejemplo, el elemento buscado sí que se encuentra en el árbol.

Por otra parte, en la figura 6.14 se muestra un ejemplo en el que el elemento buscado no se encuentra en el árbol. En este caso, la llamada inicial es:  $nodo = abb\_buscar(T, x)$  donde  $x$  es un elemento con clave 19.

Debajo de las figuras se muestra qué instrucción se lleva a cabo dada la situación que se muestra en la figura.

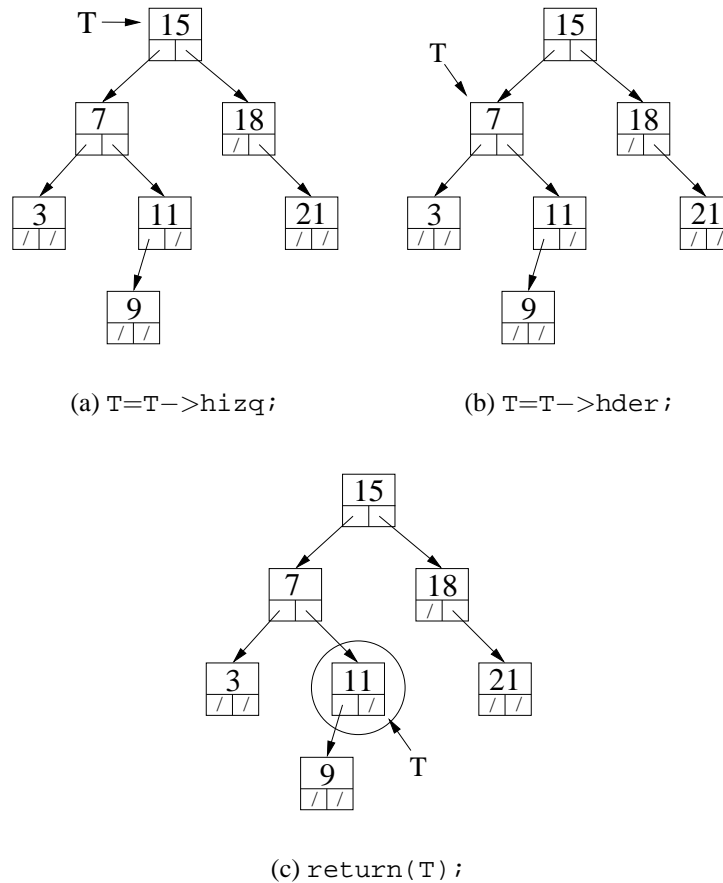


Figura 6.13: Ejemplo de cómo se desplaza el puntero  $T$  a través de la estructura del árbol hasta encontrar el elemento buscado ( $x = 11$ ).

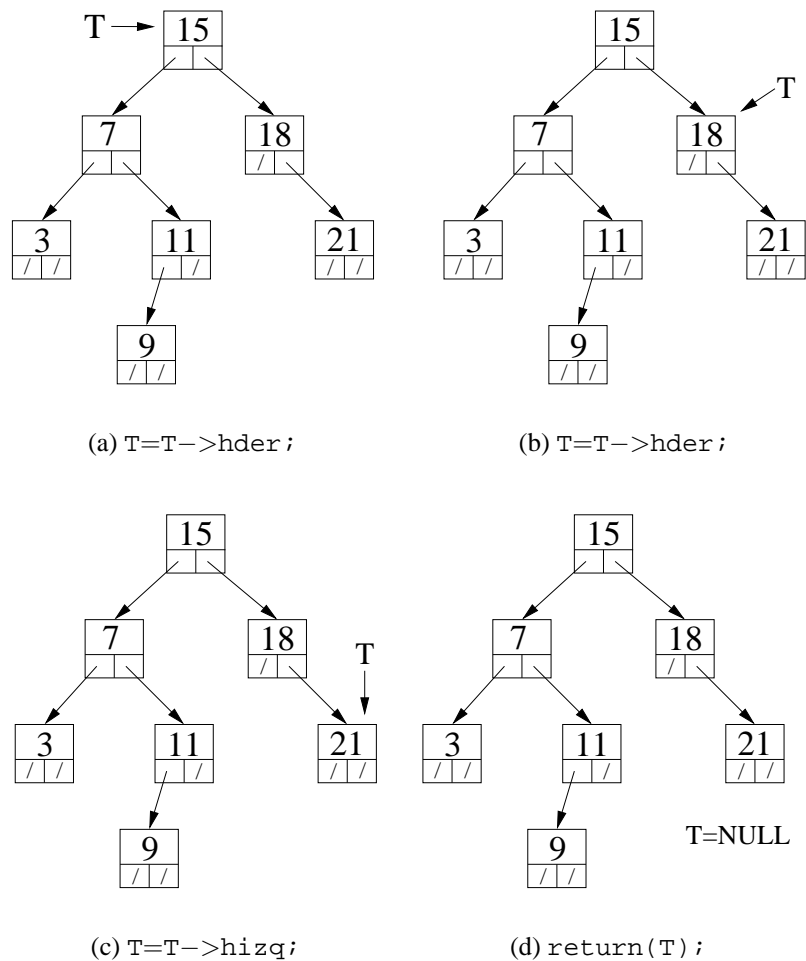


Figura 6.14: Ejemplo de cómo se desplaza el puntero  $T$  a través de la estructura del árbol sin encontrar el elemento buscado  $x = 19$ .

### 6.3.5. Búsqueda del elemento mínimo y del elemento máximo

En un árbol binario de búsqueda, el elemento cuya clave es mínima puede ser fácilmente encontrado, simplemente realizando un recorrido descendente desde la raíz a través de los hijos izquierdos de cada nodo, hasta encontrar un nodo que no tenga hijo izquierdo. Ese nodo será el nodo cuya clave es mínima.

La siguiente función devuelve un puntero al nodo cuya clave es la clave mínima del árbol apuntado por  $T$ .

```

abb *minimo(abb *T) {
    if (T != NULL)
        while(T->hizq != NULL) T=T->hizq;
    return (T);
}

```

La función basa su funcionamiento en el aspecto de que, en los árboles binarios de búsqueda, el hecho de que un nodo tenga subárbol izquierdo no vacío, implica necesariamente que existen en el árbol nodos con claves menores que la clave de ese nodo. Por lo tanto, en general, para cualquier nodo  $n$  que tenga hijo izquierdo, el nodo  $n$  no podrá ser el elemento de clave mínima del subárbol de raíz  $n$ , y el elemento de clave mínima habrá que buscarlo siempre en su subárbol izquierdo. Por lo tanto, aplicando este razonamiento, si el árbol binario de búsqueda no es vacío y nos situamos inicialmente en el nodo raíz, la búsqueda del elemento de clave mínima supone ir descendiendo siempre a través de los hijos izquierdo de cada nodo hasta encontrar un nodo  $x$  que carezca de subárbol izquierdo, lo que implicará necesariamente que no existen en el árbol claves menores que la clave del nodo  $x$ .

Para obtener el nodo cuya clave es máxima, el razonamiento será simétrico al expuesto en el caso de la búsqueda del mínimo; es decir, se realizará un recorrido, comenzando desde el nodo raíz, descendiendo siempre a través de los hijos derecho de cada nodo. Cuando se encuentre un nodo que no tenga subárbol derecho se habrá localizado al nodo de clave máxima. A continuación, se presenta una función, que devuelve un puntero al nodo cuya clave es la clave máxima del árbol apuntado por  $T$ .

```

abb *maximo(abb *T) {
    if (T != NULL)
        while(T->hder != NULL) T=T->hder;
    return (T);
}

```

El número de nodos que es necesario examinar hasta encontrar el elemento de clave mínima o máxima, determinará el coste de los algoritmos. Debido a que el camino de búsqueda que se va trazando en el árbol, examina un nodo por cada nivel, como máximo será necesario examinar todos los niveles del árbol hasta encontrar el elemento buscado. Por lo tanto, **el coste temporal de obtener el mínimo o el máximo de un árbol binario de búsqueda será  $O(h)$ , donde  $h$  es la altura del árbol.**

En las figuras 6.15 y 6.16 se muestran ejemplos del funcionamiento de ambos algoritmos. Debajo de las figuras se muestra qué instrucción se lleva a cabo dada la situación que se muestra en la figura.



### Ejemplo

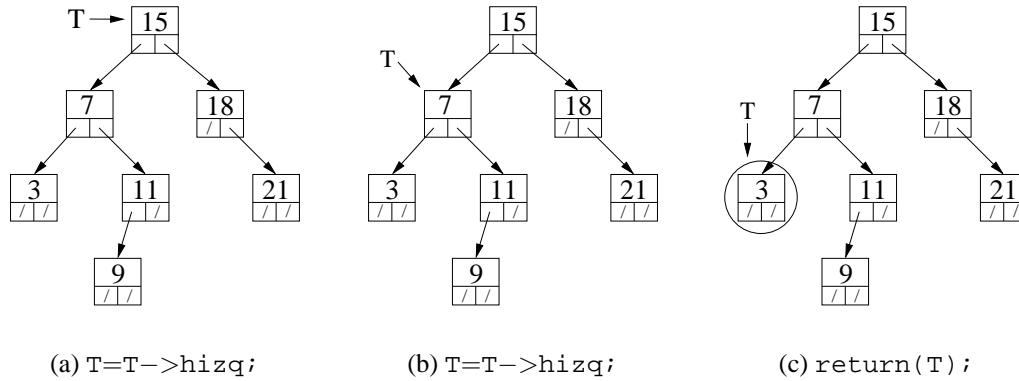


Figura 6.15: Ejemplo de cómo se desplaza el puntero  $T$  a través de la estructura del árbol hasta encontrar el nodo de clave mínima.

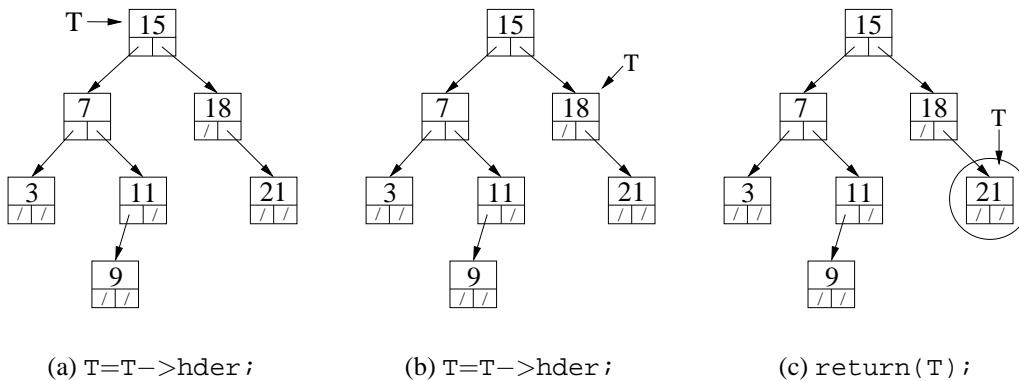


Figura 6.16: Ejemplo de cómo se desplaza el puntero  $T$  a través de la estructura del árbol hasta encontrar el nodo de clave máxima.

### 6.3.6. Inserción de un elemento en un árbol binario de búsqueda

La inserción de un nuevo elemento en un árbol binario de búsqueda debe realizarse de tal forma, que tras la inserción, se siga cumpliendo que, para todo nodo del árbol, las claves de los nodos de su subárbol izquierdo son menores, y las claves de los nodos de su subárbol derecho son mayores. Para conseguirlo, la inserción se realizará de la siguiente forma:

1. Si el árbol binario de búsqueda está vacío, se inserta el nuevo nodo como raíz del árbol.
2. Si el árbol binario de búsqueda no está vacío, la inserción se realizará en dos pasos:
  - a) Se busca la posición que le corresponde en el árbol binario de búsqueda al nuevo nodo. Para obtener la posición correcta donde debe ubicarse el nuevo nodo, se realiza un recorrido desde la raíz del árbol siguiendo el mismo criterio que se seguía en la operación de búsqueda.
  - b) Una vez encontrada la posición que le corresponde, el nuevo elemento es insertado en el árbol enlazándolo correctamente con el nodo que debe ser su padre.

A continuación se presenta una función que, dado un puntero  $T$  al nodo raíz de un árbol binario de búsqueda, y un puntero  $nodo$  a un elemento, realiza la inserción del elemento apuntado por  $nodo$  en el árbol apuntado por  $T$ , y como resultado devuelve un puntero al nodo raíz del nuevo árbol que se obtiene tras la inserción.

```
abb *abb_insertar(abb *T, abb *nodo) {
    abb *aux, *aux_padre=NULL;

    aux = T;
    while (aux != NULL) {
        aux_padre = aux;
        if (nodo->clave < aux->clave)
            aux = aux->hizq;
        else aux = aux->hder;
    }
    if (aux_padre == NULL) return(nodo);
    else {
        if (nodo->clave < aux_padre->clave)
            aux_padre->hizq = nodo;
        else aux_padre->hder = nodo;
        return(T);
    }
}
```

La función comienza la búsqueda de la posición que le corresponde al nuevo nodo desde el nodo raíz, y va trazando un camino descendente a través del árbol, de forma que cada vez que se sitúa en un nuevo nodo  $n$ , compara la clave

del nodo  $n$  con la clave del nuevo elemento a insertar; si la clave del nuevo elemento es menor, el recorrido continúa por el subárbol izquierdo de  $n$ , ya que la característica de los árboles binarios de búsqueda implica que el nuevo elemento debería insertarse en el subárbol izquierdo. De forma análoga, si la clave del nuevo elemento es mayor que la clave del nodo  $n$ , el recorrido continúa por el subárbol derecho. El recorrido finaliza cuando se llega a un subárbol vacío, momento en el cual se ha encontrado la posición que le corresponde al nuevo nodo en el árbol. La inserción se realiza enlazando el nuevo nodo con el último nodo  $n$  que se visitó en el recorrido. Para ello, si la clave del nuevo nodo es menor que la clave del último nodo  $n$ , el nuevo elemento se inserta como hijo izquierdo de  $n$  o, en caso contrario, como hijo derecho.

La inserción de un nuevo elemento en el árbol binario de búsqueda supone buscar previamente la posición que le corresponde en el árbol. Por lo tanto, el número de nodos que es necesario examinar hasta encontrar la posición determinará el coste del algoritmo. Al igual que ocurría con el resto de operaciones, el camino de búsqueda que se va trazando en el árbol, examina un nodo por cada nivel; por lo tanto, como máximo, será necesario examinar todos los niveles del árbol hasta encontrar la posición correcta. Esto implica que **el coste temporal de insertar un elemento en un árbol binario de búsqueda será  $O(h)$ , donde  $h$  es la altura del árbol.**

A continuación se muestra un ejemplo de inserción en un árbol binario de búsqueda; se inserta un nodo de clave 10. Debajo de las figuras se indica qué instrucciones se llevan a cabo dada la situación que se muestra en la figura.

## Ejemplo

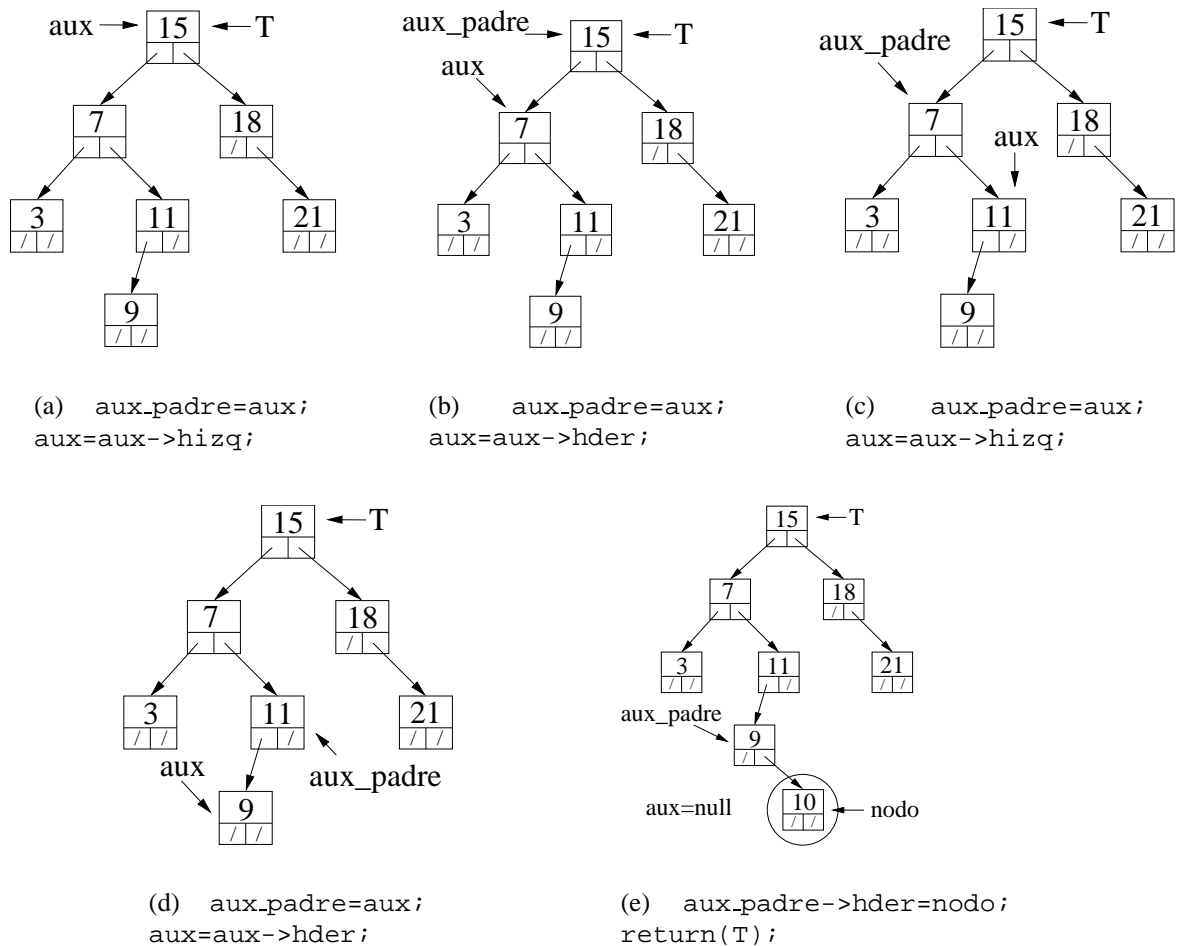


Figura 6.17: Ejemplo de cómo se desplazan los punteros *aux* y *aux\_padre* a través de la estructura del árbol hasta encontrar la posición que le corresponde al nuevo nodo a insertar (10). Una vez localizada la posición del nuevo nodo, se inserta enlazándolo correctamente al padre (que es apuntado por *aux\_padre*).

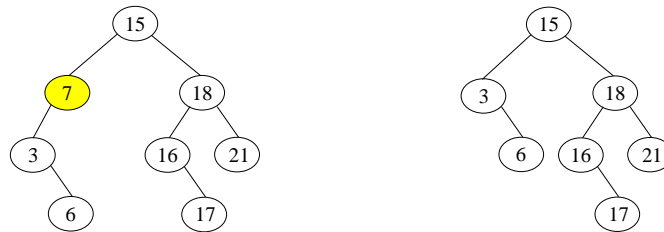
### 6.3.7. Borrado de un elemento en un árbol binario de búsqueda

El borrado de un elemento en un árbol binario de búsqueda se realiza teniendo en cuenta los siguientes casos:

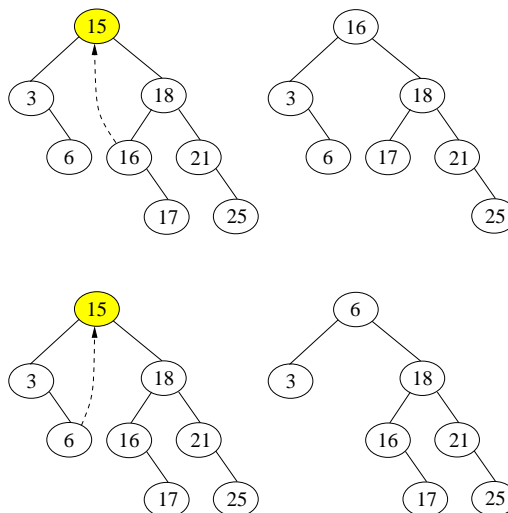
1. Si el elemento a borrar **no tiene hijos (es un hoja)**, se elimina.



2. Si el elemento a borrar es un **nodo interior que tiene un hijo**, se elimina, y su posición es ocupada por el hijo que tiene.



3. Si el elemento a borrar es un **nodo interior que tiene dos hijos**, su lugar es ocupado por el nodo de clave mínima del subárbol derecho (o por el de clave máxima del subárbol izquierdo).



A continuación se presenta una función en C que, siguiendo el esquema descrito, borra el nodo de clave  $x$  (si se encuentra) del árbol binario de búsqueda a cuya raíz apunta  $T$ .

```

abb *abb_borrar(abb *T, tipo_baseT x){
    abb *aux, *aux_padre=NULL, *p1, *p2=NULL;
    aux = T;
    /* buscamos el nodo a borrar (clave x) */
    while ( (aux != NULL) && (x != aux->clave) ){
        aux_padre = aux;
        if (x < aux->clave) aux = aux->hizq;
        else aux = aux->hder;
    }

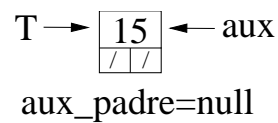
    if (aux != NULL){ /* se encuentra el nodo a borrar */
        /* Caso 1: El nodo a borrar es una hoja */
        if ( (aux->hizq == NULL) && (aux->hder == NULL) ){
            /* El árbol sólo tenía un nodo -> retorna árbol vacío (Fig.6.18) */
            if (aux_padre == NULL) T = NULL;
            else{ /* El nodo a borrar tiene padre */
                if (aux_padre->hizq == aux) /* El nodo era hijo izquierdo (Fig.6.19) */
                    aux_padre->hizq = NULL;
                else /* El nodo era hijo derecho (Fig.6.20) */
                    aux_padre->hder = NULL;
            }
            free(aux); /* Eliminamos el nodo */
        } /* Caso 2: El nodo a borrar es un nodo interior con un hijo */
        else if (aux->hizq == NULL){ /* Sólo tiene hijo derecho */
            if (aux_padre == NULL) /* El nodo a borrar es la raíz (Fig.6.21) */
                T = aux->hder;
            else{ /* El padre hereda el hijo derecho del nodo borrado */
                if (aux_padre->hizq == aux) aux_padre->hizq = aux->hder; /* (Fig.6.22) */
                else aux_padre->hder = aux->hder; /* (Fig.6.23) */
            }
            free(aux); /* Eliminamos el nodo */
        } else if (aux->hder == NULL){ /* Sólo tiene hijo izquierdo */
            if (aux_padre == NULL) /* El nodo a borrar es la raíz (Fig.6.24) */
                T = aux->hizq;
            else{ /* El padre hereda el hijo izquierdo del nodo borrado */
                if (aux_padre->hizq == aux) aux_padre->hizq = aux->hizq; /* (Fig.6.25) */
                else aux_padre->hder = aux->hizq; /* (Fig.6.26) */
            }
            free(aux); /* Eliminamos el nodo */
        } /* Caso 3: El nodo a borrar es un nodo interior con dos hijos */
        else{ /* buscamos el nodo de clave mínima del subárbol derecho */
            p1 = aux->hder;
            while (p1->hizq != NULL){ p2 = p1; p1=p1->hizq;}
            /* el nodo de clave mínima (p1) era el hijo derecho del nodo a borrar */
            if (p2 == NULL){aux->clave = p1->clave;aux->hder = p1->hder;} /* (Fig.6.27) */
            /* el nodo de clave mínima (p1) estaba en el subárbol */
            /* izquierdo del hijo derecho */
            else{aux->clave = p1->clave; p2->hizq = p1->hder;} /* (Fig.6.28) */
            free(p1); /* Eliminamos el nodo de clave mínima */
        }
    }
}

return(T);
}

```

A continuación se representan cada una de las situaciones que pueden suceder en el borrado de un nodo en un árbol binario de búsqueda. También, se muestra el árbol resultante después de realizar cada borrado. Debajo de las figuras se indican las instrucciones que se llevan a cabo, dada la situación que se muestra en la figura.

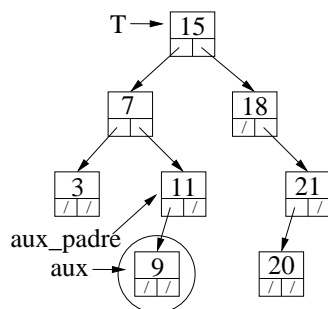
En las figuras 6.18, 6.19 y 6.20 se muestran los distintos casos que pueden darse al **borrar un nodo que es hoja** (Caso 1).



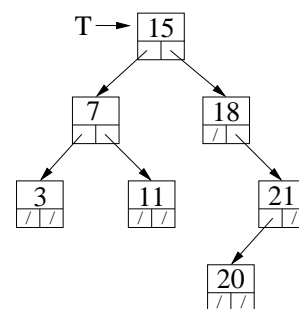
(i)

`T=NULL; free(aux); return(T);`

Figura 6.18: Borrado  $x = 15$ . Caso: **hoja que no tiene padre** ( $aux\_padre = NULL$ ). En este caso, el nodo a borrar es el único nodo del árbol, por lo que el borrado devuelve el árbol vacío ( $T = NULL$ ).



(a) `aux_padre->hizq=NULL; free(aux);`



(b) `return(T);`

Figura 6.19: Borrado  $x = 9$ . Caso: **hoja que es hijo izquierdo de un nodo**. Para realizar el borrado se actualiza a *NULL* el puntero al hijo izquierdo del padre (apuntado por *aux\_padre*), y se libera la memoria del nodo.



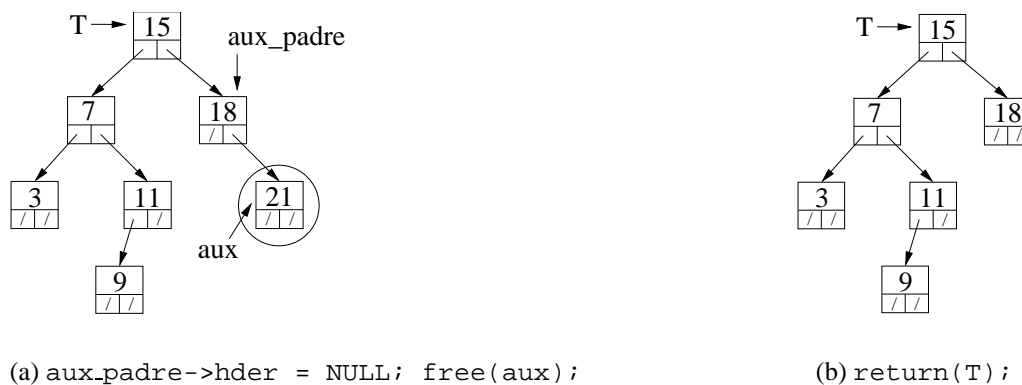


Figura 6.20: Borrado  $x = 21$ . Caso: **hoja que es hijo derecho de un nodo**. Para realizar el borrado se actualiza a *NULL* el puntero al hijo derecho del padre (apuntado por *aux\_padre*), y se libera la memoria del nodo.

En las figuras 6.21, 6.22 y 6.23, 6.24, 6.25 y 6.26 se muestran los distintos casos que pueden darse al **borrar un nodo interno que tiene un único hijo** (Caso 2).

En las figuras 6.21, 6.22 y 6.23, el nodo a borrar tiene un único hijo derecho.

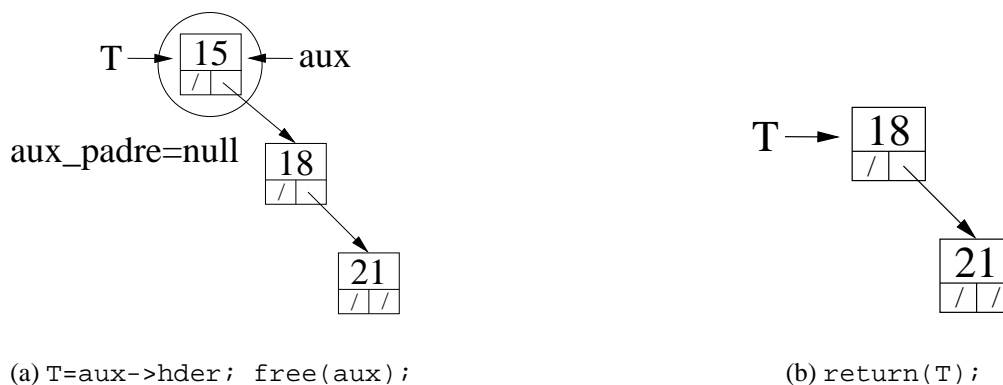
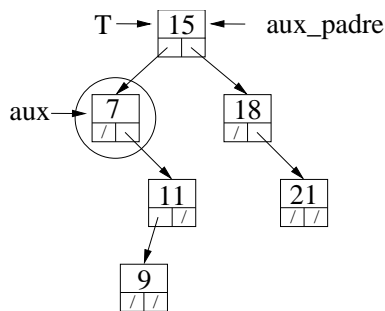
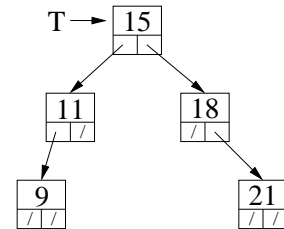


Figura 6.21: Borrado  $x = 15$ . Caso: **nodo interno que es raíz del árbol y tiene un único hijo derecho**. Para realizar el borrado se actualiza el puntero *T* para que apunte al hijo derecho, y se realiza el borrado liberando la memoria del nodo (apuntado por *aux*).

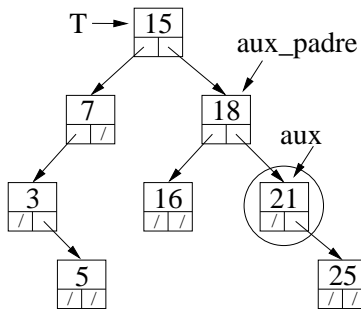


(a) `aux_padre->hizq=aux->hder; free(aux);`

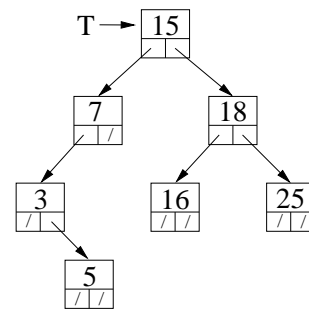


(b) `return(T);`

Figura 6.22: Borrado  $x = 7$ . Caso: **nodo interno que es hijo izquierdo de un nodo y tiene un único hijo derecho**. Antes de borrar el nodo, el hijo debe ser enlazado correctamente con el padre del nodo a borrar; para ello, se actualiza el puntero “hijo izquierdo” del padre (apuntado por *aux\_padre*) para que apunte al hijo derecho del nodo a borrar (apuntado por *aux*); finalmente, se realiza el borrado liberando la memoria del nodo.



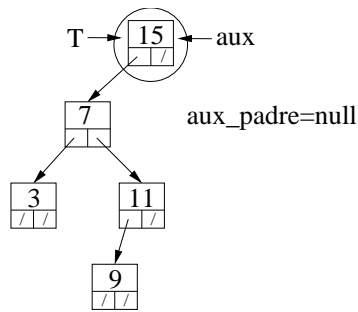
(a) `aux_padre->hder=aux->hder; free(aux);`



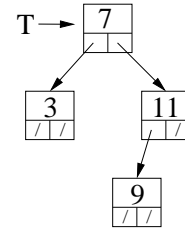
(b) `return(T);`

Figura 6.23: Borrado  $x = 21$ . Caso: **nodo interno que es hijo derecho de un nodo y tiene un único hijo derecho**. Antes de borrar el nodo, el hijo debe ser enlazado correctamente con el padre del nodo a borrar; para ello, se actualiza el puntero “hijo derecho” del padre (apuntado por *aux\_padre*) para que apunte al hijo derecho del nodo a borrar (apuntado por *aux*); finalmente, se realiza el borrado liberando la memoria del nodo.

En las figuras 6.24, 6.25 y 6.26, el nodo a borrar tiene un único hijo izquierdo.

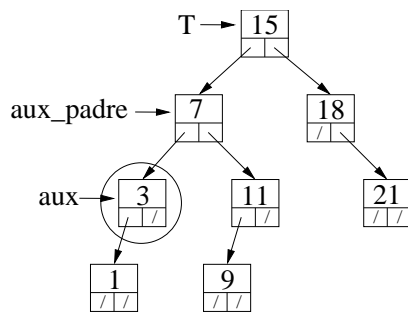


(a) `T=aux->hizq; free(aux);`

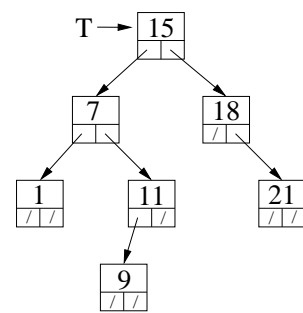


(b) `return(T);`

Figura 6.24: Borrado  $x = 15$ . Caso: **nodo interno que es raíz del árbol y tiene un único hijo izquierdo**. Para realizar el borrado se actualiza el puntero  $T$  para que apunte al hijo izquierdo, y se realiza el borrado liberando la memoria del nodo (apuntado por  $aux$ ).

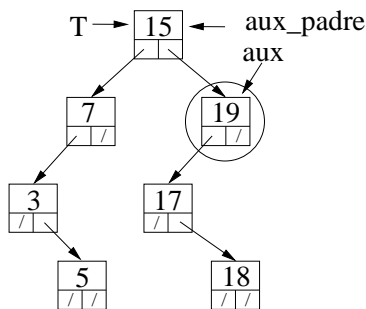


(a) `aux_padre->hizq=aux->hizq; free(aux);`

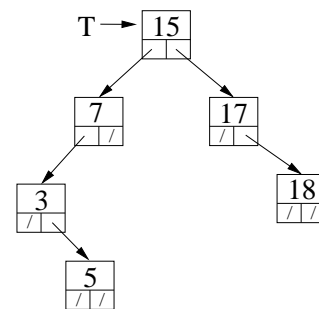


(b) `return(T);`

Figura 6.25: Borrado  $x = 3$ . Caso: **nodo interno que es hijo izquierdo de un nodo y tiene un único hijo izquierdo**. Antes de borrar el nodo, el hijo debe ser enlazado correctamente con el padre del nodo a borrar; para ello, se actualiza el puntero “hijo izquierdo” del padre (apuntado por  $aux\_padre$ ) para que apunte al hijo izquierdo del nodo a borrar (apuntado por  $aux$ ); finalmente, se realiza el borrado liberando la memoria del nodo.



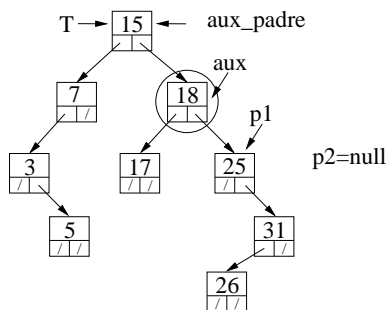
(a) `aux_padre->hder=aux->hizq; free(aux);`



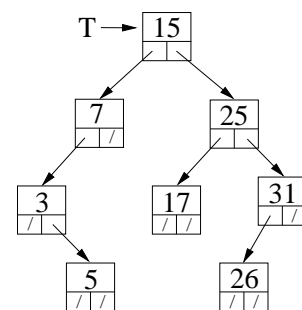
(b) `return(T);`

Figura 6.26: Borrado  $x = 19$ . Caso: **nodo interno que es hijo derecho de un nodo y tiene un único hijo izquierdo**. Antes de borrar el nodo, el hijo debe ser enlazado correctamente con el padre del nodo a borrar; para ello, se actualiza el puntero “hijo derecho” del padre (apuntado por *aux\_padre*) para que apunte al hijo izquierdo del nodo a borrar (apuntado por *aux*); finalmente, se realiza el borrado liberando la memoria del nodo.

En las figuras 6.27 y 6.28 se muestran los casos que pueden darse al borrar un **nodo interno que tiene dos hijos** (Caso 3).

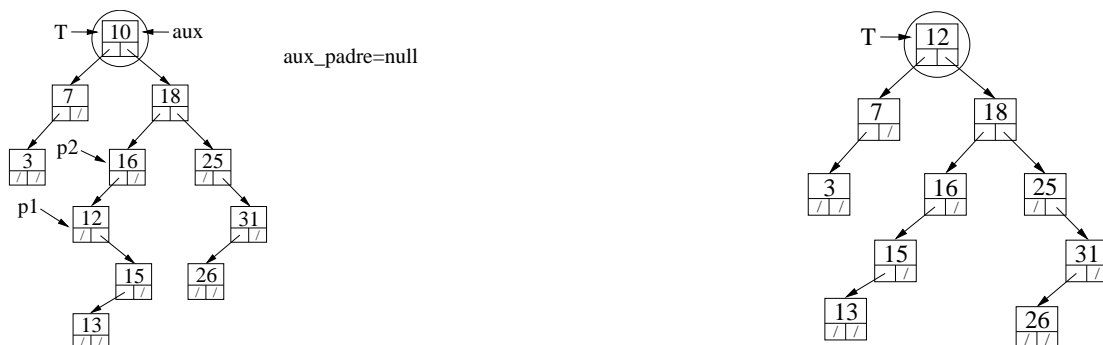


(a) `aux->clave=p1->clave; aux->hder=p1->hder; free(p1)`



(b) `return(T);`

Figura 6.27: Borrado  $x = 18$ . Caso: **nodo interno que tiene dos hijos y su hijo derecho es el nodo de clave mínima de su subárbol derecho**. La clave del nodo a borrar se sustituye por la clave mínima de su subárbol derecho. Esta modificación implica que el nodo de clave mínima ha pasado a ocupar la posición de su *padre*, se enlazará su hijo derecho (único hijo), como hijo derecho del nuevo nodo que ocupa. Finalmente, se libera el nodo que ocupaba, la clave mínima del subárbol derecho del nodo a borrar.



(a) `aux->clave=p1->clave; p2->hizq=p1->hder; free(p1);` (b) `return(T);`

Figura 6.28: Borrado  $x = 10$ . Caso: **nodo interno que tiene dos hijos y el nodo de clave mínima de su subárbol derecho no es su hijo derecho**. La clave del nodo a borrar se sustituye por la clave mínima de su subárbol derecho; como el *padre* del nodo de clave mínima no ha pasado a ocupar la clave mínima, su hijo derecho (único hijo) se enlaza con el nodo que era su padre antes del cambio. Así, el hijo derecho del nodo de clave mínima pasa a ser hijo izquierdo del nodo padre (apuntado por *p2*). Finalmente, se libera el nodo que ocupaba la clave mínima del subárbol derecho del nodo a borrar.

**El coste temporal de borrar un elemento en un árbol binario de búsqueda será  $O(h)$ , donde  $h$  es la altura del árbol.**

### Ejercicio:

Las siguientes definiciones de tipos y variables sirven para representar árboles binarios de búsqueda:

```
typedef ... tipo_baseT;

typedef struct snodo {
    tipo_baseT clave;
    struct snodo *hizq, *hder;
} abb;

abb *T;
```

Escribir una función recursiva que imprima por pantalla las claves almacenadas en un árbol  $T$  que sean menores que una clave dada  $k$ . En el caso peor, el coste del algoritmo debe ser  $O(n)$ , siendo  $n$  el número de nodos del árbol. Sugerencia: Una posible solución la puede proporcionar una modificación adecuada de algún algoritmo ya conocido de recorrido de árboles.

```

void menores(abb *T, tipo_baseT k) {
    if (T != NULL) {
        if (T->clave < k) {
            printf("%d ", T->clave);
            menores(T->hizq, k);
            menores(T->hder, k);
        }
        else
            menores(T->hizq, k);
    }
}

```

## 6.4. Montículos (*Heaps*). Colas de prioridad.

Antes de definir la estructura de datos *montículo*, recordaremos brevemente qué es un árbol binario completo: un árbol binario completo es un árbol binario en el que todos los niveles tienen el máximo número posible de nodos excepto, puede ser, el último. En ese caso, las hojas del último nivel están tan a la izquierda como sea posible <sup>2</sup>.

Un montículo o *heap* es un conjunto de  $n$  elementos representados en un árbol binario completo en el que se cumple la siguiente propiedad: para todo nodo  $i$ , excepto el nodo raíz, la clave del nodo  $i$  es menor o igual que la clave del nodo padre de  $i$ . A esta propiedad le denominaremos **propiedad de montículo**.

La representación de un montículo se realiza utilizando la representación vectorial de los árboles binarios completos; es decir:

- En la posición 1 del vector se encuentra el nodo raíz del árbol.
- Dado un nodo que ocupa la posición  $i$  en el vector:
  - En la posición  $2i$  se encuentra el nodo que es su hijo izquierdo.
  - En la posición  $2i + 1$  se encuentra el nodo que es su hijo derecho.
  - En la posición  $\lfloor i/2 \rfloor$  se encuentra el nodo padre si  $i > 1$ .
- Desde la posición  $\lfloor n/2 \rfloor + 1$  hasta la posición  $n$ , donde  $n$  es el número de nodos del árbol binario completo, se encuentran las claves de los nodos que son hojas.

---

<sup>2</sup>En el apartado 4,3 del tema de árboles se introdujo el concepto de árbol binario completo

Por lo tanto, en un montículo representado mediante un vector  $A$  se cumplirá que:  
 $A[\lfloor i/2 \rfloor] \geq A[i], 1 < i \leq n$ .

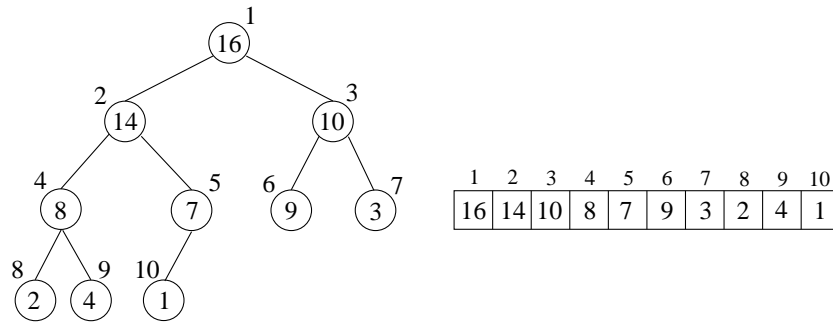
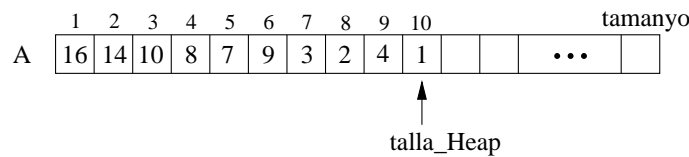


Figura 6.29: Representación de un montículo como árbol binario completo y como vector.

Dado que, posiblemente, el vector  $A$  que representa un montículo tendrá un tamaño superior al número de elementos almacenados en el montículo, será necesario mantener un índice que indique el número de nodos que almacena el montículo; a este índice le denominaremos *talla\_Heap*.



Debido a la propiedad que satisfacen todos los nodos de un montículo, en un montículo se cumple que:

- El nodo de clave máxima se encuentra en la raíz del árbol.
- Cada rama del árbol está ordenada de mayor a menor desde la raíz hasta las hojas.

Además, debido a que un montículo se representa en un árbol binario completo, su altura será  $\Theta(\log n)$ , donde  $n$  es el número de elementos que contiene el montículo.

### 6.4.1. Manteniendo la propiedad de montículo

Supongamos que tenemos un árbol binario completo donde existe un nodo  $i$  que tiene un subárbol izquierdo y un subárbol derecho que son montículos. Supongamos también, que la clave del nodo  $i$  es menor que alguna de las claves

de los nodos que son sus hijos ( $A[i] < A[2i]$  y/o  $A[i] < A[2i + 1]$ ) (ver figura 6.30). En este caso, el subárbol que parte del nodo  $i$  no cumpliría la propiedad de montículo. Una función muy importante en el manejo de montículos es aquella que, dado un nodo que viola la propiedad de montículo, transforma el subárbol que parte de ese nodo en un montículo, manteniendo así la propiedad de montículo.

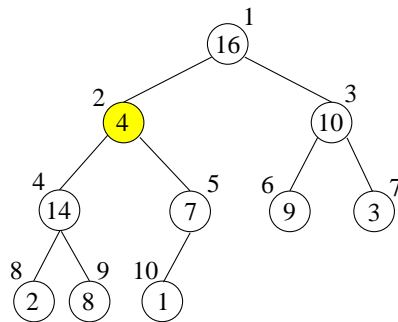


Figura 6.30: El nodo 2 no cumple la propiedad de montículo. Obsérvese que los subárboles izquierdo y derecho del nodo 2 sí que son montículos.

La función, denominada *heapify*, transforma el subárbol que parte de un nodo  $i$  en un montículo. La función asume que los subárboles izquierdo y derecho del nodo  $i$  son montículos. A continuación se presenta una versión recursiva de la función *heapify*. La función recibe un vector  $M$  que representa un árbol binario completo, y un índice  $i$  que indica el nodo raíz del subárbol que se desea que sea un montículo.



```

void heapify(tipo_baseT *M, int i) {
    tipo_baseT aux;
    int hizq, hder, mayor;

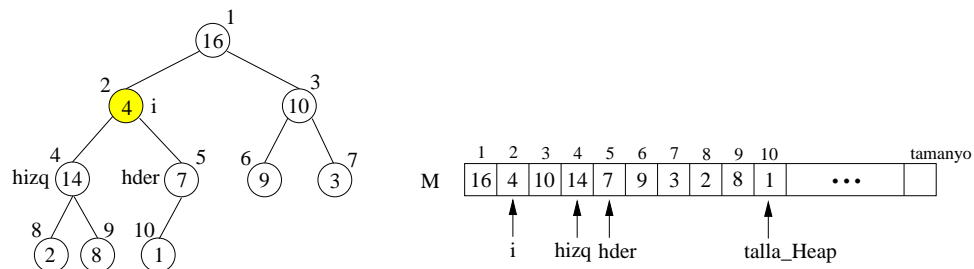
    hizq = 2*i;
    hder = 2*i+1;
    if ((hizq<=talla_Heap) && (M[hizq]>M[i]))
        mayor = hizq;
    else
        mayor = i;
    if ((hder<=talla_Heap) && (M[hder]>M[mayor]))
        mayor = hder;
    if (mayor != i) {
        aux = M[i];
        M[i] = M[mayor];
        M[mayor] = aux;
        heapify(M,mayor);
    }
}

```

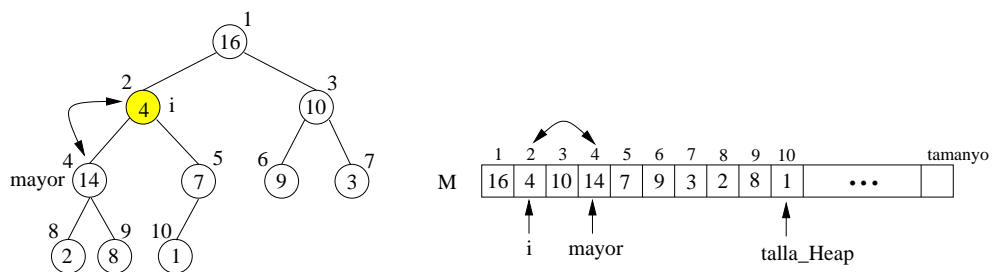
La función *heapify* compara inicialmente la clave del nodo  $i$  ( $M[i]$ ) con la de su hijo izquierdo ( $M[2i]$ ), si lo tiene, y almacena en la variable *mayor* el índice del nodo que tiene la clave mayor de los dos; a continuación compara la clave del nodo obtenido ( $M[mayor]$ ) con la clave del hijo derecho del nodo  $i$  ( $M[2i+1]$ ), si lo tiene, almacenando nuevamente en la variable *mayor* el índice del nodo mayor. Una vez ha obtenido qué nodo de los tres (nodo  $i$ , hijo izquierdo, hijo derecho) tenía la clave mayor, si el nodo de clave mayor era el nodo  $i$  la función finaliza ya que, para el subárbol que parte del nodo  $i$ , se cumplirá la propiedad de montículo. En caso contrario, uno de los dos hijos es el nodo mayor, en ese caso, la claves del nodo  $i$  y del nodo mayor se intercambian, consiguiendo que el nodo  $i$  y sus hijos cumplan la propiedad de montículo. Debido a que el cambio puede suponer que ahora la propiedad de montículo no se cumpla en el subárbol que partía del nodo de clave mayor, se deberá aplicar recursivamente el algoritmo *heapify* sobre ese nodo.

#### **Ejemplo del funcionamiento de *heapify***

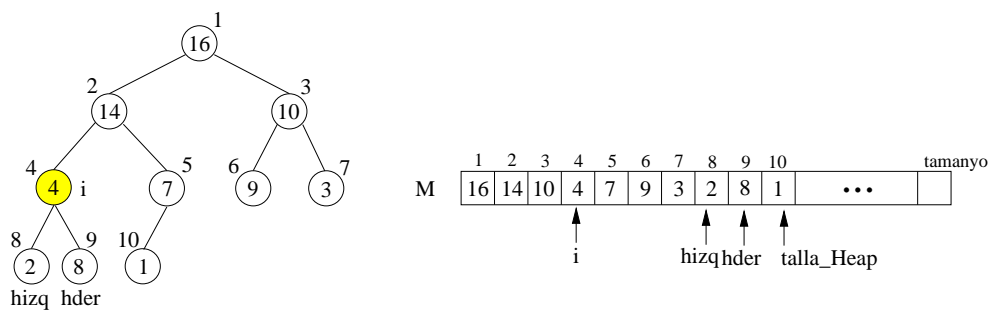
A continuación se muestra un ejemplo del funcionamiento de *heapify* para una llamada inicial *heapify*( $M, 2$ ), donde  $M$  es el vector que se muestra en la figura y  $talla\_Heap = 10$ .



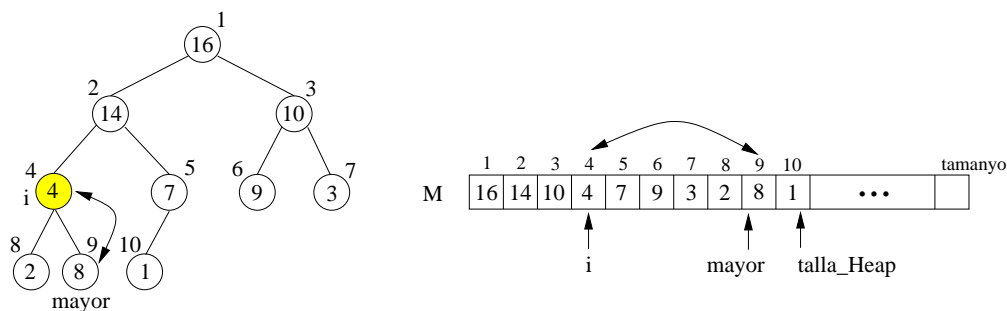
(a) El nodo 2 no cumple la propiedad de montículo ya que no es mayor que sus hijos.



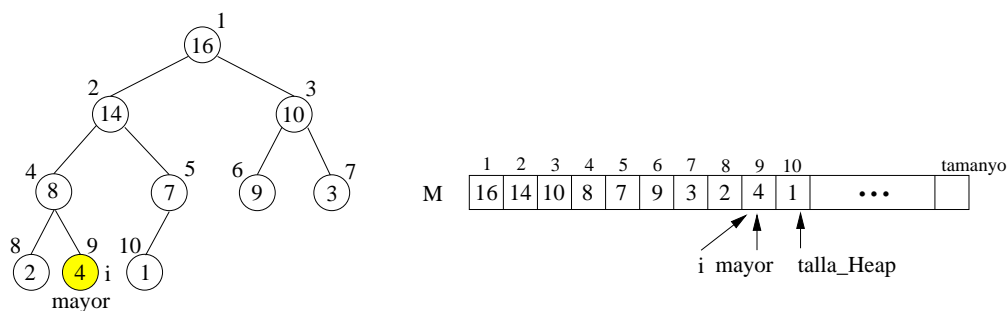
(b) El nodo de clave mayor es el hijo izquierdo del nodo  $i$  por lo que se intercambian sus claves. A continuación se realiza la llamada recursiva `heapify(M, 4)`



(c) La modificación de la clave del nodo 4 en la llamada anterior, ha provocado que ahora sea este nodo el que no cumple la propiedad de montículo.



(d) El nodo de clave mayor es el hijo derecho del nodo  $i$  por lo que se intercambian sus claves. A continuación se realiza la llamada recursiva  $\text{heapify}(M, 9)$ .



(e) Por último, debido a que el nodo 9 no tiene hijos, el nodo de clave mayor es el nodo  $i$ , por lo que las llamadas recursivas finalizan obteniendo el árbol resultante. Obsérvese que, ahora, el subárbol que tiene como raíz el nodo 2, nodo sobre el que se aplicó inicialmente la acción *heapify*, sí que es un montículo.

### Coste temporal de *heapify*

El coste temporal del algoritmo vendrá determinado por el coste que supone obtener la clave del nodo mayor, entre el nodo  $i$ , su hijo izquierdo y su hijo derecho, más el coste que supone aplicar la función *heapify* sobre un subárbol cuya raíz es uno de los hijos del nodo  $i$ .

Por lo tanto, el **caso mejor** ocurrirá cuando la clave del nodo  $i$  sea mayor que las claves de sus hijos. En este caso, el coste temporal será únicamente el coste de obtener el nodo de clave mayor:  $O(1)$ .

El **caso peor** se producirá cuando la acción *heapify* se tenga que realizar sobre el máximo número posible de nodos. Este caso ocurrirá cuando la llamada inicial a la función *heapify* se realice sobre el nodo raíz del árbol, y para cada nodo sobre el que se aplique *heapify*, se cumpla que la clave del nodo es menor que alguna de las claves de sus hijos. En este caso, dado que la acción *heapify* se aplica sobre

un nodo en cada nivel del árbol, el coste vendrá determinado por  $O(h)$ , donde  $h$  es la altura del árbol. Dado que la altura de un árbol binario completo es  $\lfloor \log_2 n \rfloor$ , donde  $n$  es el número de nodos que contienen el árbol, el coste de la función *heapify* en el caso peor es  $O(\log n)$ . En general, para un nodo  $i$ , el coste de aplicar *heapify* sobre ese nodo, en el caso peor, será  $O(h)$  donde  $h$  es la altura del nodo  $i$ .

## 6.4.2. Construir un montículo

**Problema:** Dado un conjunto de  $n$  elementos representados en un árbol binario completo mediante un vector  $M$ , se desea transformar el vector para que sea un montículo.

**Estrategia:** Dado que los nodos que son hojas ya cumplen la propiedad de montículo, tras aplicar la función *heapify* sobre cada uno del resto de nodos, comenzando por el nodo de mayor índice que no es hoja ( $M[\lfloor n/2 \rfloor]$ ) y recorriendo decrecientemente hasta el nodo raíz del árbol ( $M[1]$ ), se conseguirá que todos los nodos cumplan la propiedad de montículo, por lo que el árbol binario completo resultante será un montículo. El orden en que se aplica sobre cada uno de los nodos la función *heapify*, garantiza que cuando se aplica la acción *heapify* sobre un nodo siempre se cumple que sus subárboles son montículos.

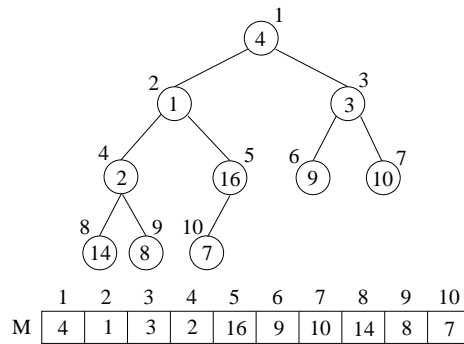
A continuación se presenta una función en C que, dado un vector  $M$  que contiene  $n$  elementos  $M[1, \dots, n]$ , transforma el vector en un montículo siguiendo la estrategia indicada.

```
void build_Heap(tipo_baseT *M, int n) {
    int i;

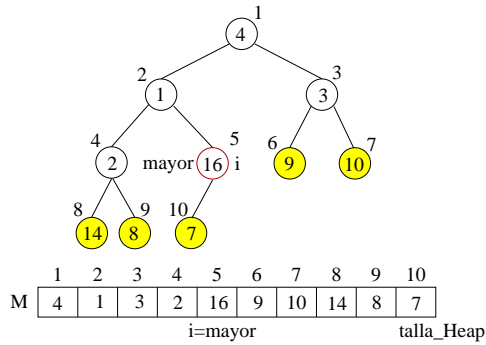
    talla_Heap = n;
    for (i=n/2; i>0; i--) heapify(M,i);
}
```

### Ejemplo del funcionamiento de `build_Heap`

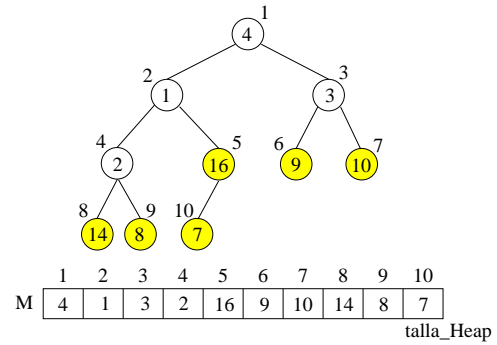
A continuación se muestra un ejemplo del funcionamiento de la función `build_Heap` al aplicarlo sobre el vector  $M$  que se muestra en la figura; la llamada inicial es: `build_Heap(M, 10)`. Los nodos sombreados representan los nodos del árbol sobre los que se ha aplicado la acción *heapify*; a excepción de las hojas sobre las que no es necesario aplicar *heapify* ya que inicialmente son montículos.



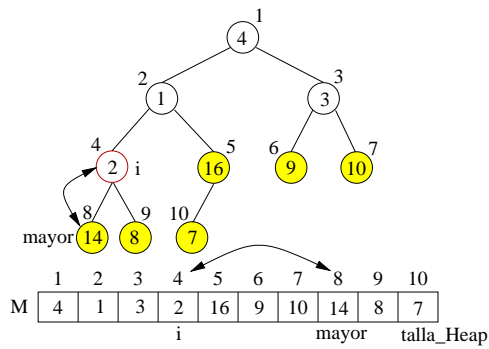
(a) `build_Heap(M, 10)`



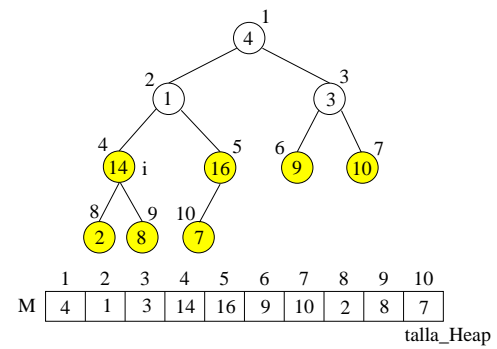
(b)  $\text{Heapify}(M, 5)$ : En este caso, el nodo 5 ya cumple la propiedad de montículo.



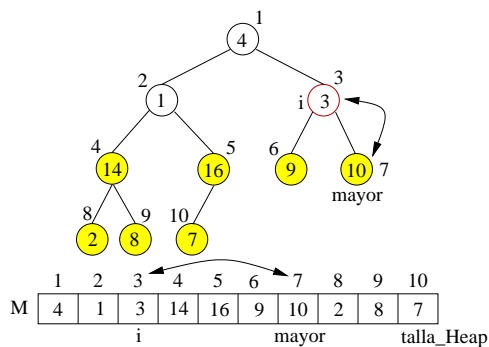
(c)  $\text{Heapify}(M, 5)$ : La acción de heapify sobre el nodo 5 no realiza ninguna modificación sobre el árbol.



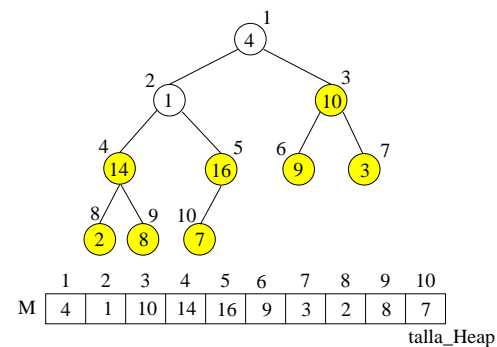
(d)  $\text{Heapify}(M, 4)$ : El nodo 4 no cumple la propiedad de montículo. Se intercambia su clave con la clave mayor de sus hijos.



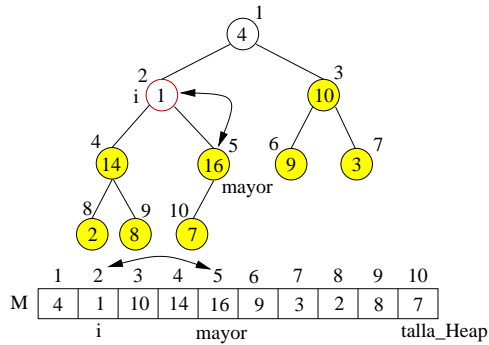
(e)  $\text{Heapify}(M, 4)$ : La acción de heapify sobre el nodo 4 transforma el subárbol de raíz en el nodo 4 en un montículo.



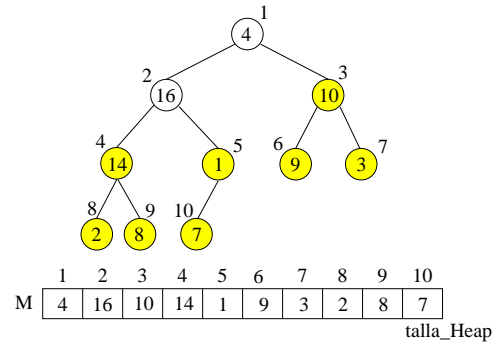
(f)  $\text{Heapify}(M, 3)$ : El nodo 3 no cumple la propiedad de montículo. Se intercambia su clave con la clave mayor de sus hijos.



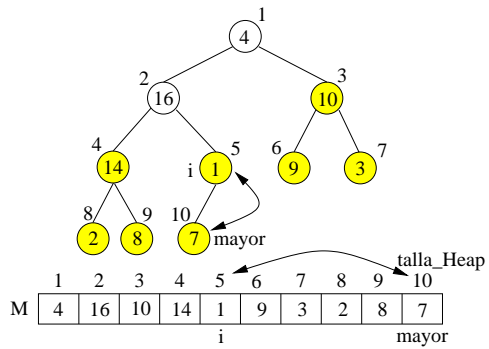
(g)  $\text{Heapify}(M, 3)$ : La acción de heapify sobre el nodo 3 transforma el subárbol de raíz en el nodo 3 en un montículo.



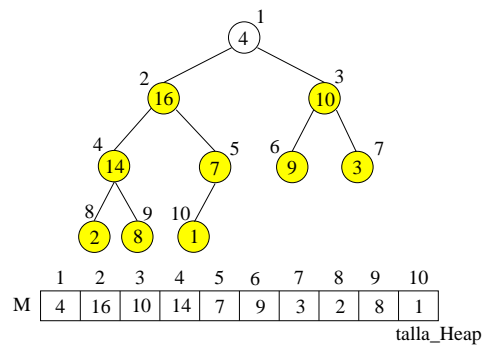
(h)  $\text{Heapify}(M, 2)$ : El nodo 2 no cumple la propiedad de montículo. Se intercambia su clave con la clave mayor de sus hijos.



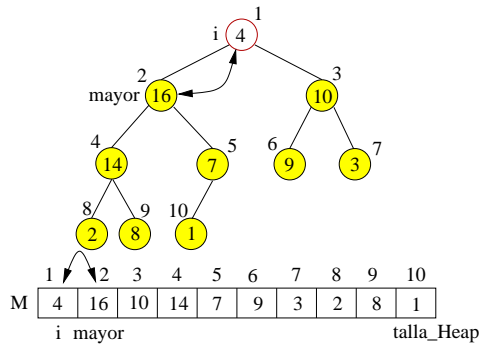
(i)  $\text{heapify}(M, 2)$ : La modificación de la clave del nodo 5 provoca una llamada recursiva a  $\text{Heapify}(M, 5)$ .



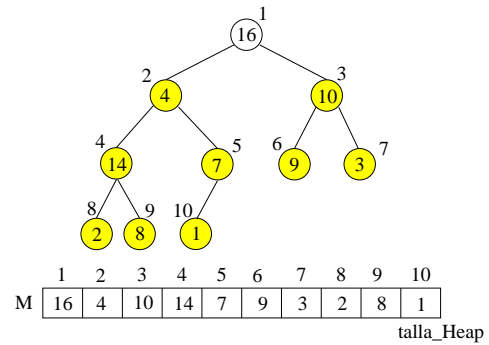
(j)  $\text{heapify}(M, 5)$ : El nodo 5 no cumple la propiedad de montículo. Se intercambia su clave con la clave mayor de sus hijos.



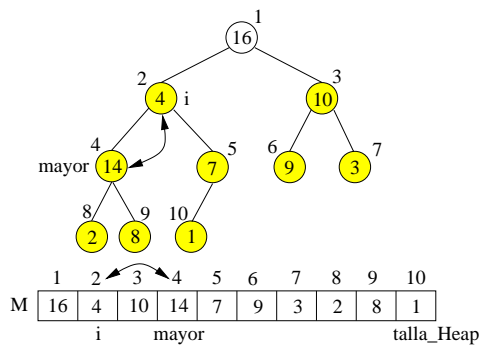
(k)  $\text{heapify}(M, 5)$ : La acción *heapify* sobre el nodo 5 transforma el subárbol de raíz en este nodo nuevamente en un montículo; al ser la última de las modificaciones iniciadas en  $\text{heapify}(M, 2)$  se cumple también que el subárbol de raíz en el nodo 2 es un montículo.



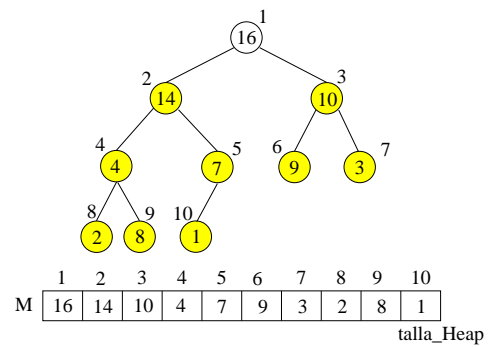
(l)  $\text{heapify}(M, 1)$ : El nodo 1 no cumple la propiedad de montículo. Se intercambia su clave con la clave mayor de sus hijos.



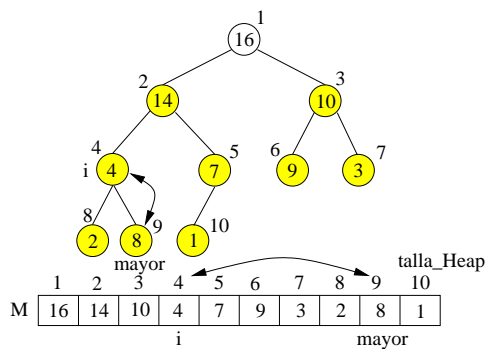
(m)  $\text{heapify}(M, 1)$ : La modificación de la clave del nodo 2 provoca una llamada recursiva a  $\text{Heapify}(M, 2)$ .



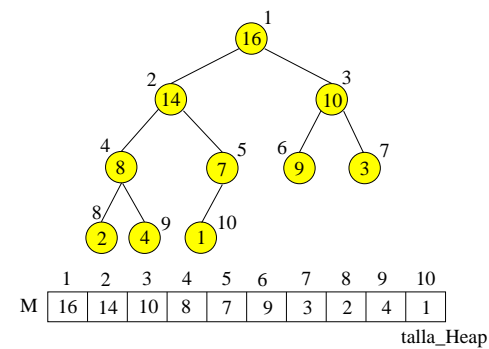
(n)  $\text{Heapify}(M, 2)$ : El nodo 2 no cumple la propiedad de montículo. Se intercambia su clave con la clave mayor de sus hijos.



(ñ)  $\text{heapify}(M, 2)$ : La modificación de la clave del nodo 4 provoca una llamada recursiva a  $\text{Heapify}(M, 4)$ .



(o)  $\text{Heapify}(M, 4)$ : El nodo 4 no cumple la propiedad de montículo. Se intercambia su clave con la clave mayor de sus hijos.



(p)  $\text{heapify}(M, 4)$ : La acción *heapify* sobre el nodo 4 transforma el subárbol de raíz en este nodo nuevamente en un montículo; al ser la última de las modificaciones iniciadas en  $\text{heapify}(M, 1)$  se cumple también que el subárbol de raíz en el nodo 1 es un montículo, es decir, se ha transformado el vector inicial en un montículo.

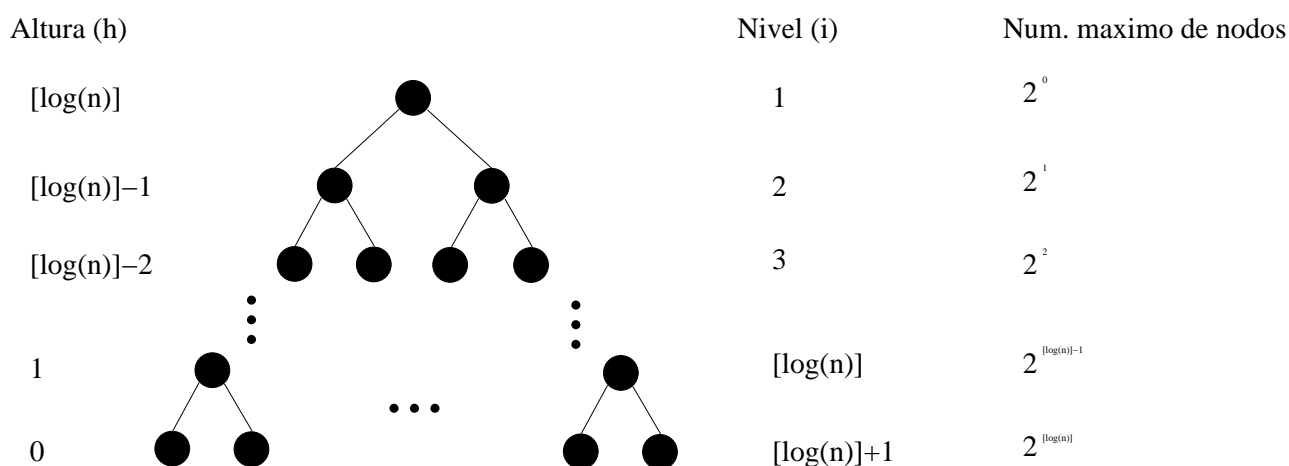


### Coste temporal de *build Heap*

Sabiendo que el coste temporal de *heapify* es  $O(\log n)$ , una forma de obtener fácilmente una cota superior para el coste temporal del algoritmo *build Heap*, sería asumir que cada llamada a la función *heapify* tiene un coste  $O(\log n)$ . De esta forma, podríamos decir que, como máximo, el coste temporal de transformar un vector de talla  $n$  en un montículo es  $O(n \log n)$ . Sin embargo, aún siendo esta cota superior correcta, no es exáctamente la más cercana a la cota real.

Dado que aplicar la acción *heapify* sobre un nodo tiene un coste temporal de  $O(h)$ , donde  $h$  es la altura del nodo, el coste temporal de *build heap* dependerá de la altura a la que se encuentren cada uno de los nodos sobre los que se aplica *heapify* durante la construcción del montículo. Por lo tanto, para cada altura  $h$ , tendremos un coste de  $N_h O(h)$ , donde  $N_h$  es el número de nodos que tienen altura  $h$  en el árbol.

Para determinar el coste temporal, será necesario establecer una relación entre la altura de un nodo y el número máximo de nodos que pueden tener esa altura. Dado que en un árbol binario completo que contiene  $n$  nodos se cumple que su altura es  $\lfloor \log_2 n \rfloor$ , y que para cada nivel  $i$  el número máximo de nodos es  $2^{i-1}$ ; se puede establecer una relación entre la altura de un nodo y el número máximo de nodos que puede contener el árbol a esa altura, de la siguiente forma:



Observando la figura puede verse que para cada altura  $h$  se cumple que el número de nodos  $N_h$  es:

$$N_h \leq 2^{\lfloor \log_2 n \rfloor - h} = \frac{2^{\lfloor \log_2 n \rfloor}}{2^h}$$

Utilizando esta relación podemos obtener una cota superior del coste temporal del algoritmo *build\_Heap*. Para ello, para cada altura del árbol, evaluaremos el coste que supone aplicar *heapify* sobre cada uno de sus nodos. La siguiente expresión realiza este cálculo:

$$\begin{aligned}
 T(n) &= \sum_{h=0}^{\lfloor \log_2 n \rfloor} N_h O(h) \\
 &\leq \sum_{h=0}^{\lfloor \log_2 n \rfloor} \frac{2^{\lfloor \log_2 n \rfloor}}{2^h} h \\
 &\leq \sum_{h=0}^{\log_2 n} 2^{\log_2 n} \frac{h}{2^h} \\
 &= n \sum_{h=0}^{\log_2 n} h \left(\frac{1}{2}\right)^h
 \end{aligned}$$

Dado que  $\sum_{h=0}^{\infty} hx^h = x/(1-x)^2$ , si  $0 < x < 1$ , entonces:

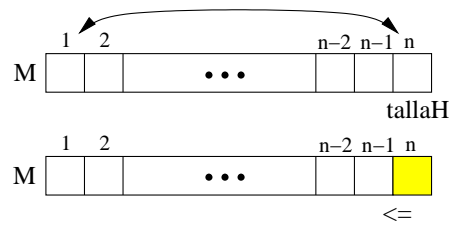
$$\begin{aligned}
 n \sum_{h=0}^{\log_2 n} h \left(\frac{1}{2}\right)^h &\leq \frac{1/2}{(1-1/2)^2} n \\
 &= 2n \in O(n)
 \end{aligned}$$

Como se ha visto el **coste temporal del algoritmo *build\_Heap* es  $O(n)$** . Por lo tanto, podemos concluir que construir un montículo partiendo de un vector tiene un coste temporal lineal.

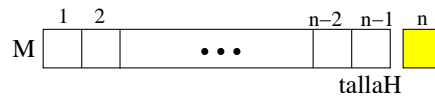
### 6.4.3. Algoritmo de ordenación *Heapsort*

El algoritmo *heapsort* ordena de manera no decreciente los elementos almacenados en un vector. Dado un vector  $M$  de  $n$  elementos  $M[1, \dots, n]$ , el algoritmo realiza la siguiente estrategia:

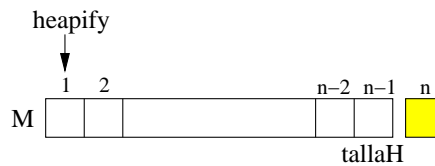
1. utiliza la función *build\_Heap* para transformar el vector  $M$  en un montículo;
2. El algoritmo *heapsort* explota el hecho de que la clave mayor de un montículo siempre se encuentra en el nodo raíz del montículo. Por ello, intercambia la clave del nodo raíz (valor máximo) con la clave del último nodo almacenado en el montículo. De esta forma, el elemento mayor del montículo se ubica en la posición que debe ocupar tras la ordenación del vector;



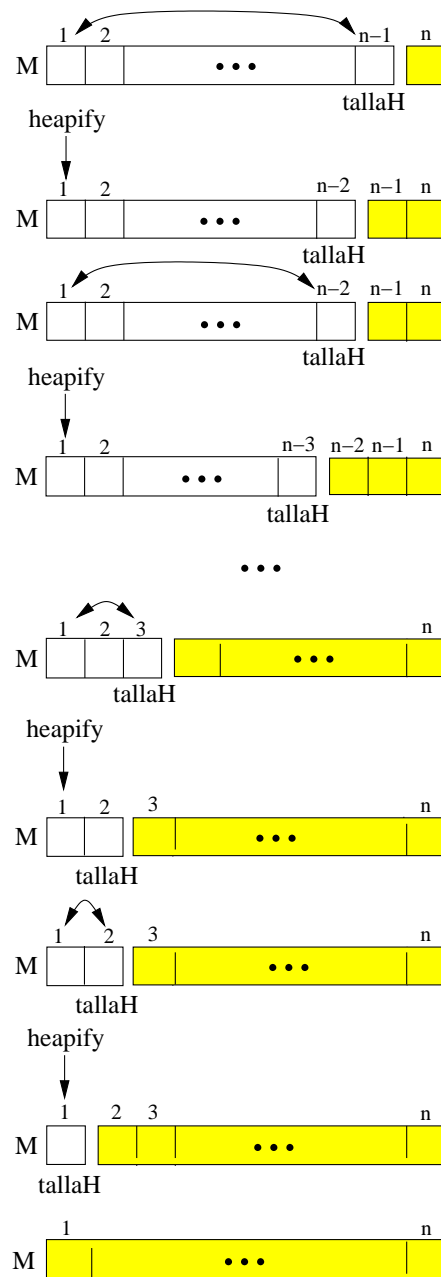
3. dado que uno de los elementos ya ha sido colocado en la posición que le corresponde (al final del montículo), se reduce la talla del montículo en 1, consiguiendo, así, excluir este elemento de las siguientes acciones a realizar en la ordenación;



4. debido a que, en el intercambio, el valor que ha sido colocado en la raíz puede violar la propiedad de montículo, se aplica *heapify* sobre el nodo raíz para que el vector  $M$  vuelva a ser un montículo;



5. tras restablecer la propiedad de montículo en el vector  $M$ , se repiten los pasos (2), (3) y (4) hasta conseguir la ordenación total del vector.



A continuación, se presenta una función que, dado un vector  $M$  que contiene  $n$  elementos  $M[1, \dots, n]$ , ordena el vector  $M$  siguiendo la estrategia del algoritmo *heapsort*:

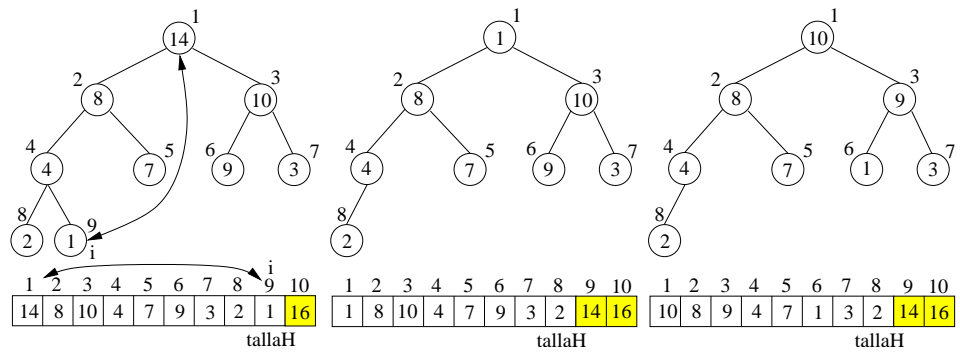
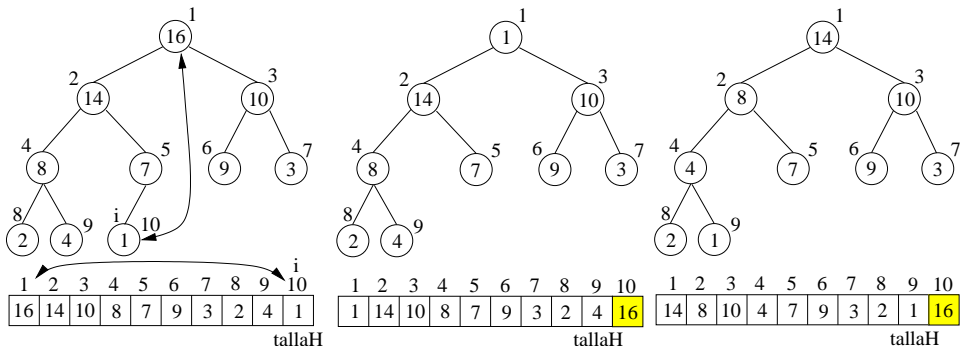
```

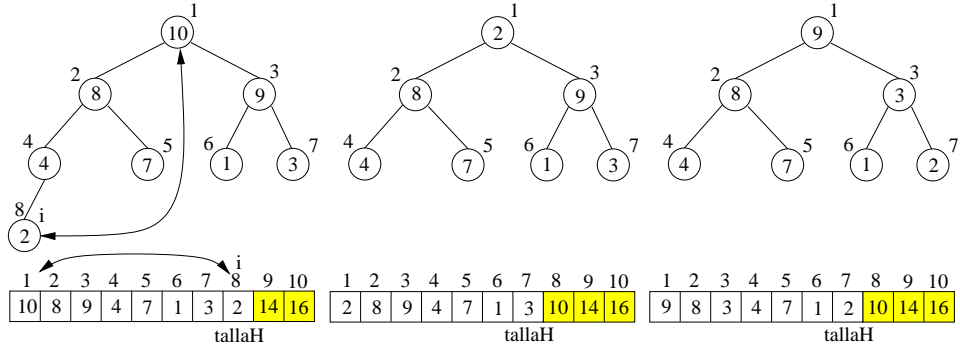
void heapsort(tipo_baseT *M, int n) {
    int i;
    tipo_baseT aux;

    build_Heap(M,n);
    for (i=n; i>1; i--) {
        aux = M[i];
        M[i] = M[1];
        M[1] = aux;
        talla_Heap--;
        heapify(M,1);
    }
}

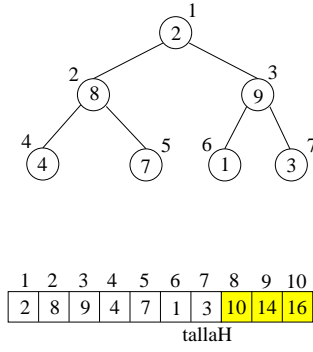
```

### Ejemplo del funcionamiento del algoritmo *heapsort*

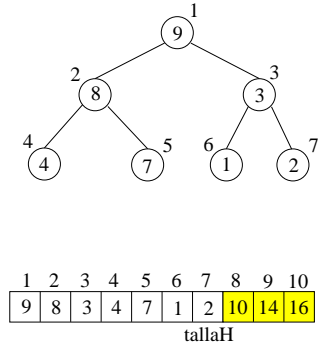




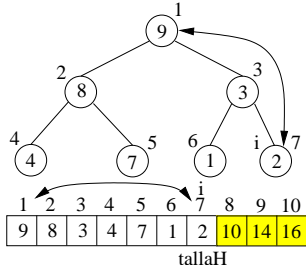
(g)  $M[1] \leftrightarrow M[i]$



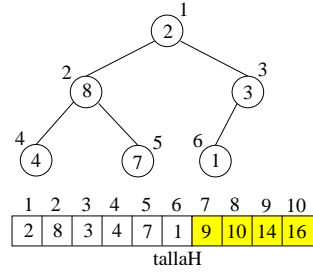
(h)  $\text{heapify}(M, 1)$



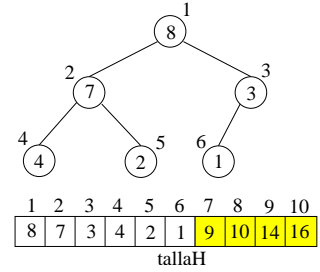
(i) fin iteraci3n  $i = 8$



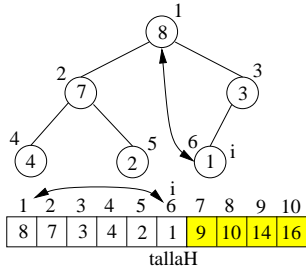
(j)  $M[1] \leftrightarrow M[i]$



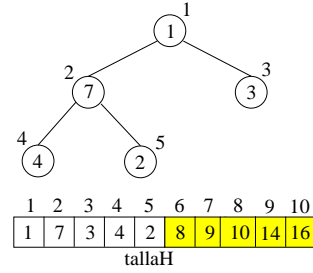
(k)  $\text{heapify}(M, 1)$



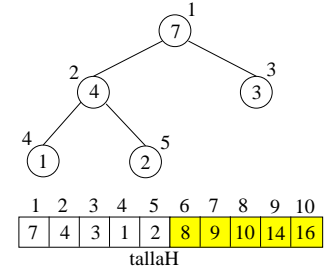
(l) fin iteraci3n  $i = 7$



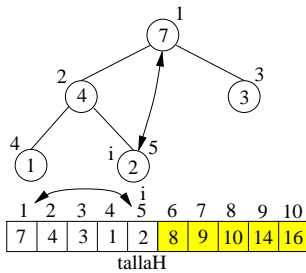
(m)  $M[1] \leftrightarrow M[i]$



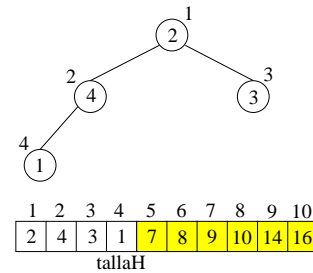
(n)  $\text{heapify}(M, 1)$



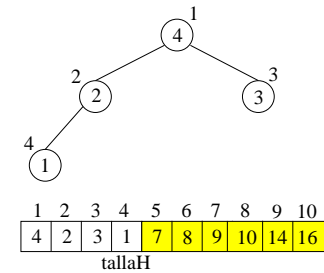
(ñ) fin iteraci3n  $i = 6$



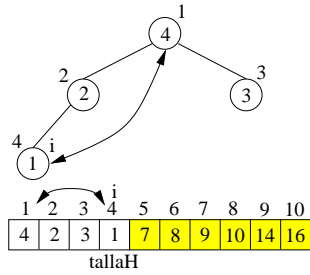
(o)  $M[1] \leftrightarrow M[i]$



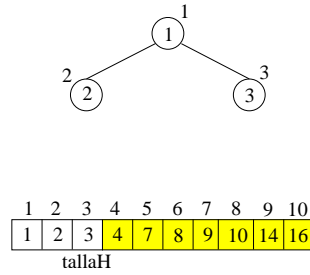
(p)  $\text{heapify}(M, 1)$



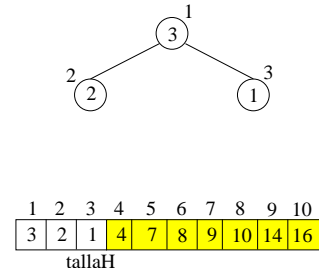
(q) fin iteraci3n  $i = 5$



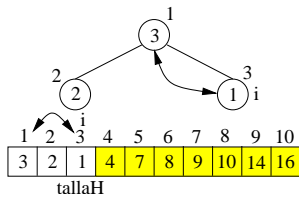
(r)  $M[1] \leftrightarrow M[i]$



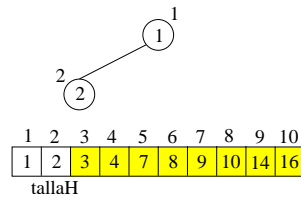
(s)  $\text{heapify}(M, 1)$



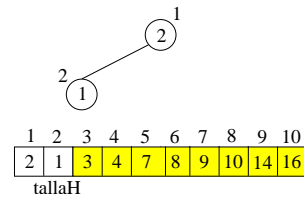
(t) fin iteración  $i = 4$



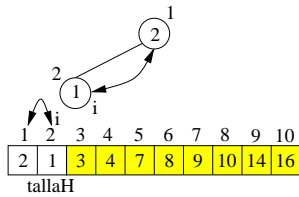
(u)  $M[1] \leftrightarrow M[i]$



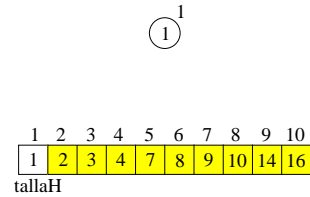
(v)  $\text{heapify}(M, 1)$



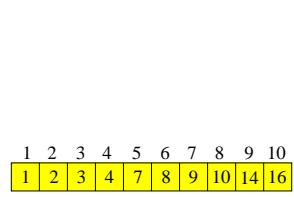
(w) fin iteración  $i = 3$



(x)  $M[1] \leftrightarrow M[i]$



(y)  $\text{heapify}(M, 1)$



## Coste temporal de *heapsort*

Debido a que el coste temporal de construir un montículo es  $O(n)$ , y que cada una de las  $n - 1$  llamadas que realiza el algoritmo a la función *heapify* tienen un coste temporal  $O(\log n)$ , podemos concluir que **el algoritmo *heapsort* tiene un coste temporal  $O(n \log n)$** . Únicamente en el caso de que todos los elementos del vector fueran iguales el coste temporal del algoritmo *heapsort* sería  $O(n)$ .

#### 6.4.4. Colas de prioridad

Una de las aplicaciones más usuales de un montículo es su utilización como una cola de prioridad.

Una **cola de prioridad** es una estructura de datos que mantiene un conjunto  $S$  de elementos, cada uno de los cuales tiene asociado un valor llamado *clave* (prioridad). Una cola de prioridad soporta las siguientes operaciones:

- $\text{Insert}(S,x)$ : inserta el elemento  $x$  en el conjunto  $S$ .
- $\text{Extract\_Max}(S)$ : borra y devuelve el elemento de  $S$  con la clave mayor.
- $\text{Maximum}(S)$ : devuelve el elemento de  $S$  con la clave mayor.

Una de las aplicaciones de las colas de prioridad es la planificación de procesos en un sistema compartido. La cola de prioridad mantiene la información de los procesos con sus prioridades asociadas. Cuando un proceso finaliza o es interrumpido, el proceso de mayor prioridad se selecciona para entrar en ejecución utilizando la operación *Extract-Max*. Nuevos procesos pueden ser añadidos a la cola en cualquier momento utilizando la operación *Insertar*.

##### Insertar un elemento en un montículo

La inserción de un nuevo elemento en un montículo debe hacerse de tal forma que el nuevo conjunto continúe siendo un montículo. Para conseguirlo, el algoritmo de inserción realiza la siguiente estrategia:

1. expande el tamaño del montículo en 1 ( $talla\_Heap + 1$ ), e inserta el nuevo elemento en esa posición del vector. De esta forma, el nuevo elemento se añade como una hoja que ocupa la posición más a la izquierda posible del último nivel del árbol;
2. a continuación, compara la clave del nuevo elemento con la clave de su nodo padre; si la clave del nuevo elemento es mayor, la inserción ha provocado que el nodo padre no cumpla la propiedad de montículo, por lo que se intercambian las claves. Debido a que el intercambio de claves puede provocar que nuevamente no se cumpla la propiedad de montículo, esta vez para el nodo que tras el intercambio ha pasado a ser padre del nuevo elemento, se repite el intercambio de claves hasta que se cumpla que la clave del nodo padre del nuevo elemento es mayor o igual, o hasta que el nuevo elemento ocupa la raíz del montículo.



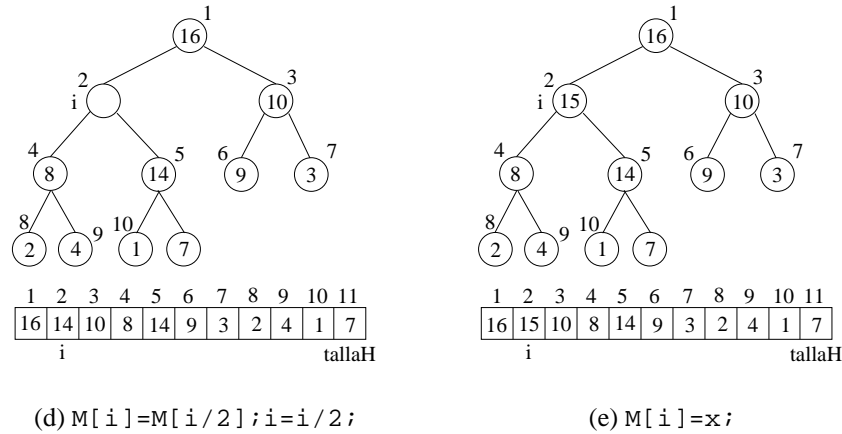
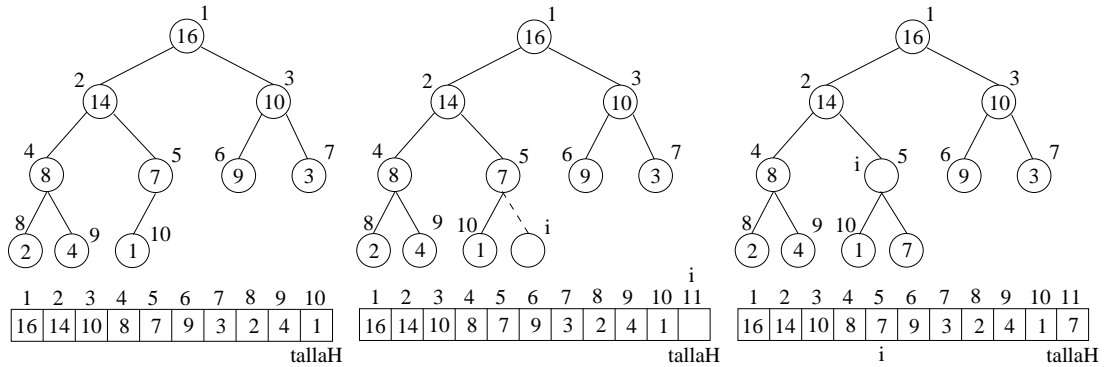
A continuación se presenta una función que, dado un montículo  $M$  inserta un nuevo elemento de clave  $x$ . Antes de realizarse la llamada a la función, debería comprobarse si el tamaño físico del vector permite la inserción de un nuevo elemento.

```
void insert(tipo_baseT *M, tipo_baseT x) {
    int i;

    talla_Heap++;
    i = talla_Heap;
    while ( (i > 1) && (M[i/2] < x) ) {
        M[i] = M[i/2];
        i = i/2;
    }
    M[i] = x;
}
```

### **Ejemplo del funcionamiento del algoritmo *insert***

A continuación se muestra un ejemplo de cómo se realiza la inserción del elemento de clave 15 en el árbol de la figura. Debajo de cada figura se indican las instrucciones que se han llevado a cabo para obtener el árbol que se muestra.



### Coste temporal del algoritmo

El coste temporal de la inserción depende del número de comparaciones que es necesario realizar, hasta encontrar la posición que le corresponde al nuevo elemento en el montículo. Por lo tanto, en un montículo que contenga  $n$  elementos, el **caso mejor** ocurrirá cuando el elemento se inserte inicialmente en la posición que le corresponde, en cuyo caso únicamente será necesario realizar una comparación:  $O(1)$ . El **caso peor** se producirá cuando el elemento insertado sea mayor que cualquier elemento del montículo; en este caso será necesario ubicar el nuevo elemento como raíz del montículo, por lo que el número de comparaciones necesarias será:  $O(\log n)$ .

### Extraer el máximo de un montículo

La extracción del máximo de un montículo supone obtener la clave almacenada en la raíz del montículo (valor máximo) y, a continuación, eliminar esta clave del montículo de tal forma que el nuevo conjunto siga siendo un montículo.

La extracción y borrado del máximo de un montículo se realiza de la siguiente forma; dado un vector  $M$  que representa un montículo:

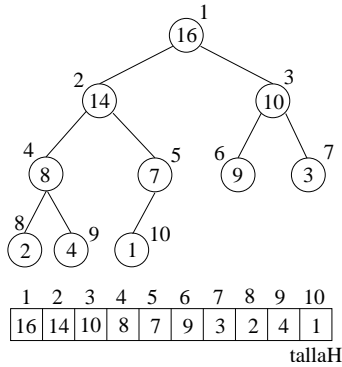
1. se obtiene el valor máximo almacenado en el montículo, que se corresponde con la clave almacenada en el nodo raíz del mismo ( $M[1]$ ).
2. se borra la clave obtenida sustituyéndola por el valor almacenado en la última posición del montículo ( $M[1]=M[talla\_Heap]$ ); y, a continuación, se reduce el tamaño del montículo en 1 ( $talla\_Heap-1$ ).
3. Debido a que el intercambio puede provocar que el nodo raíz no cumpla la propiedad de montículo, se aplica la acción *heapify* sobre el nodo raíz para restablecer la propiedad de montículo.

A continuación se presenta el algoritmo *Extract\_Max*:

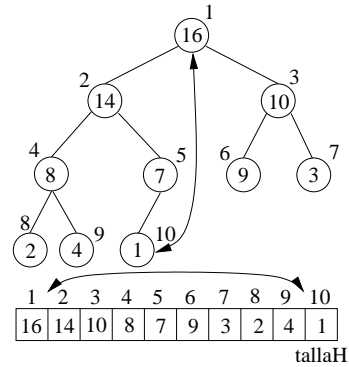
```
tipo_baseT extract_max(tipo_baseT *M) {
    tipo_baseT max;

    if (talla_Heap == 0) {
        fprintf(stderr, "Monticulo vacio");
        exit(-1);
    }
    max = M[1];
    M[1] = M[talla_Heap];
    talla_Heap--;
    heapify(M, 1);
    return (max);
}
```

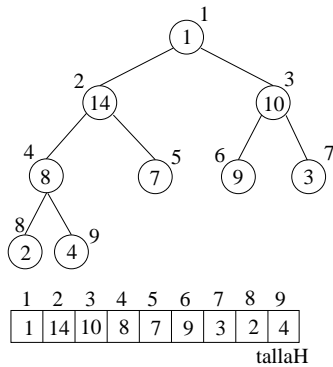
## Ejemplo del funcionamiento del algoritmo *Extract\_Max*



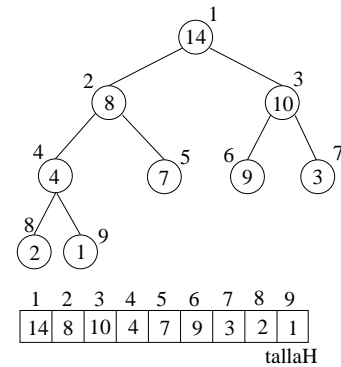
(a)  $\text{max} = M[1]$  ;



(b)  $M[1] \leftrightarrow M[\text{talla\_Heap}]$



(c)  $\text{talla\_Heap}--$ ;  $\text{heapify}(M, 1)$



(d)  $\text{return}(\text{max})$  ;

## Coste temporal del algoritmo

El **coste temporal del algoritmo *Extract\_Max***, para un montículo que contiene  $n$  elementos, es  $O(\log n)$ , ya que depende del tiempo que supone realizar la acción *heapify* sobre el nodo raíz del montículo:  $O(\log n)$ .

En resumen, **un montículo que representa un conjunto de  $n$  elementos, soporta las operaciones de una cola de prioridad con un coste temporal  $O(\log n)$ .**

## 6.5. Estructura de datos para conjuntos disjuntos: MF-set

Dado un conjunto  $C$ , la *clase de equivalencia* de un elemento  $a \in C$  es el subconjunto de  $C$  que contiene todos los elementos relacionados con  $a$ . Obsérvese que las clases de equivalencia forman una partición de  $C$ : todo miembro de  $C$  aparece en exáctamente una clase de equivalencia. Para saber si un elemento  $a$  tiene una relación de equivalencia con otro elemento  $b$ , únicamente necesitamos verificar si  $a$  y  $b$  están en la misma clase de equivalencia.

Un MF-set (Merge-Find set) es una estructura en la que el número  $n$  de elementos es fijo, no se pueden borrar ni añadir elementos, y los elementos se organizan en una colección  $S = \{S_1, S_2, \dots, S_k\}$  de subconjuntos disjuntos o *clases de equivalencia*.

Cada subconjunto se identifica mediante un **representante**, que debe ser uno de los miembros del subconjunto. En algunas aplicaciones se mantiene la norma de que el representante de cada subconjunto es siempre el elemento menor del subconjunto (asumiendo que puede establecerse una relación de orden total entre los elementos); en otras, no importa el elemento que se elija como representante. En cualquier caso debe cumplirse que siempre que se consulte el valor del representante de un subconjunto, si el subconjunto no ha sido modificado entre consulta y consulta, el resultado sea el mismo.

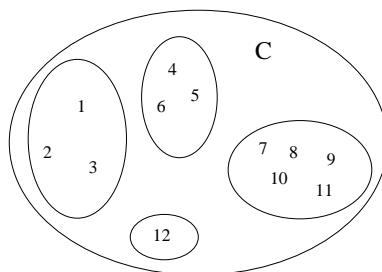


Figura 6.31: Subconjuntos disjuntos o *clases de equivalencia* en las que se agrupan los elementos del conjunto  $C$ . Cada subconjunto se identifica mediante alguno de sus miembros: en la figura cada representante aparece en negrita.

Sobre un MF-Set únicamente pueden realizarse dos operaciones:

- Union( $x, y$ ) (*Merge*): Sean  $S_x$  el subconjunto que contiene a  $x$ , y  $S_y$  el subconjunto que contiene a  $y$ , se crea un nuevo subconjunto resultado de la unión de  $S_x$  con  $S_y$ . El representante del nuevo subconjunto creado es alguno de sus miembros, aunque normalmente se escoge como representante al elemento que representaba a  $S_x$  o al que representaba a  $S_y$ . Debido a que

en todo momento se debe cumplir que la colección de subconjuntos sean disjuntos, se eliminan los subconjuntos  $S_x$  y  $S_y$ .

- **Buscar(x) (Find):** devuelve el nombre del subconjunto o *clase de equivalencia* a la que pertenece el elemento  $x$ . El nombre del subconjunto equivale al elemento que representa al subconjunto.

Este tipo de representación se utiliza en numerosas aplicaciones como, por ejemplo: inferencia de gramáticas, equivalencias de autómatas finitos, cálculo del árbol de expansión de coste mínimo en un grafo no dirigido, compiladores que procesan declaraciones (o tipos) de equivalencia, etc.

### 6.5.1. Representación de MF-sets

Existen distintas maneras de representación para MF-sets, pero nos centraremos en analizar la más eficiente.

Cada uno de los subconjuntos disjuntos se representa mediante un árbol, donde cada nodo contiene la información de un elemento, y el nodo raíz del árbol es el representante del subconjunto. Para representar cada árbol, utilizaremos un tipo de representación denominada *representación de árboles mediante apuntadores al padre*. En este tipo de representación, para cada nodo del árbol, sólo se mantiene un puntero al nodo padre. Además, dado un árbol que representa al subconjunto  $S_i$ , el nodo que se apunte a sí mismo será el nodo raíz del árbol y, por lo tanto, el representante del subconjunto  $S_i$ .

Por lo tanto, dado que cada subconjunto disjunto se representa mediante un árbol, un MF-set será una colección de árboles, o lo que es lo mismo, un bosque.

Debido a que el número  $n$  de elementos del conjunto es fijo, podemos identificar cada elemento mediante un número desde 1 hasta  $n$ . Este conjunto es fácilmente representable en un vector  $M$ , en el que en cada posición  $i$  se almacena el índice del elemento que es padre del elemento  $i$ . Obsérvese que si se cumple que  $M[i] = i$ , el elemento  $i$  será la raíz de un árbol y, por lo tanto, el representante de un subconjunto (ver figura 6.32).

### 6.5.2. Operaciones sobre MF-sets

A continuación analizaremos cómo pueden llevarse a cabo cada una de las operaciones sobre MF-sets utilizando la representación indicada en el apartado anterior.

#### **Operación Unión Merge**

**Union(x,y):** hacer que la raíz de un árbol apunte al nodo raíz del otro.

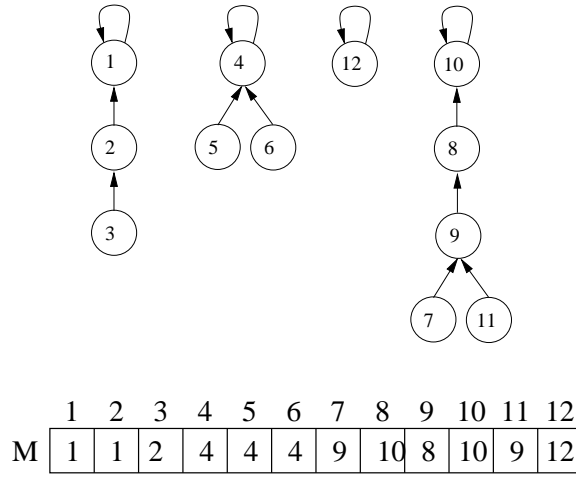


Figura 6.32: Ejemplo de representación de MF-Set. Cada uno de los conjuntos disjuntos mostrados en la figura 6.31 se representa mediante un árbol. Cada posición  $i$  del vector contiene el índice del elemento que es padre de  $i$ .

Suponiendo que  $x$  e  $y$  son raíces (representantes), la operación únicamente implica modificar el puntero al padre de uno de los representantes, por lo tanto el coste será  $O(1)$  (ver figura 6.33).

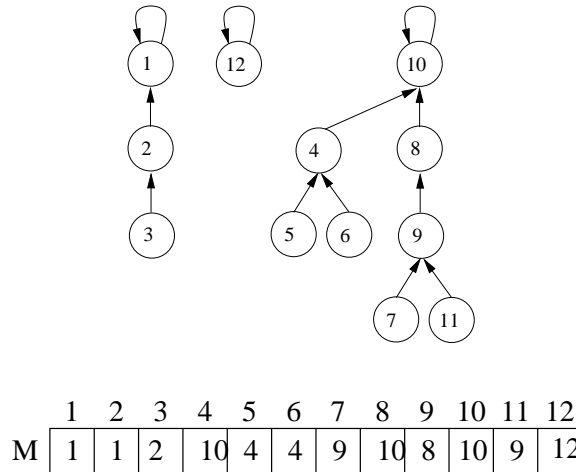


Figura 6.33: MF-set resultante tras realizar la unión de los subconjuntos disjuntos 5 y 8 que se muestran en la figura 6.32. Obsérvese que la unión se realiza modificando únicamente un valor del vector: el padre del representante de la clase de equivalencia del 5 pasa a ser el representante de la clase de equivalencia del 8.

## Operación Buscar Find

Buscar( $x$ ): utilizando el puntero al padre recorrer el árbol desde el nodo  $x$  hasta encontrar la raíz del árbol. Los nodos visitados en el camino hasta la raíz constituyen el **camino de búsqueda**.

El coste temporal será proporcional a la profundidad a la que se encuentre el nodo. El caso peor ocurrirá cuando los  $n$  elementos estén en un único conjunto y el árbol que lo representa sea una lista enlazada de  $n$  nodos; en este caso el coste es  $O(n)$ .

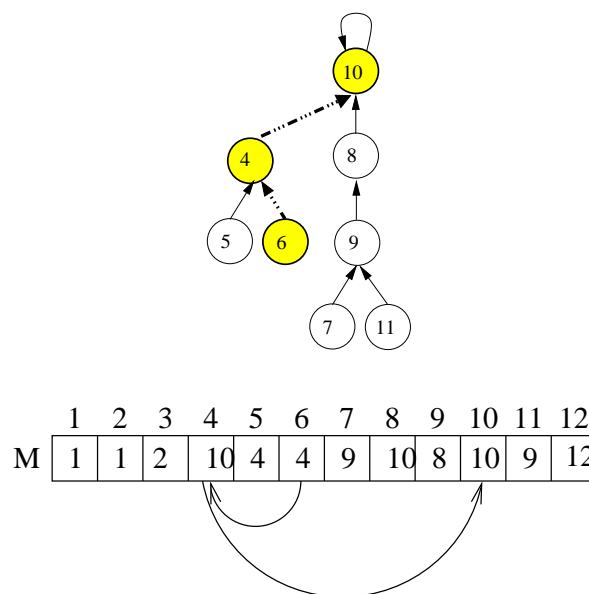


Figura 6.34: Ejemplo de búsqueda del elemento 6: el subconjunto al que pertenece el elemento nos lo dice el *representante*, que se corresponde con la raíz del árbol. La búsqueda consiste en trazar un camino a través del árbol, comenzando desde el nodo sobre el que se aplica la operación *Buscar* hasta llegar a la raíz.

## Análisis del coste temporal

Inicialmente, en un MF-Set, existen  $n$  subconjuntos disjuntos que contienen, cada uno, un elemento. Esta representación indica que, inicialmente, no existen relaciones de equivalencia entre los elementos del conjunto.

Dada esta situación inicial, la peor secuencia de operaciones que puede producirse es: a) realizar el máximo número posible de operaciones *Unión*:  $n - 1$ , con lo que se obtendrá un único conjunto de  $n$  elementos, y b) realizar  $m$  operaciones *Búsqueda*. Dada esta situación, analizaremos el coste temporal.



Dado que cada operación *Unión* tiene un coste temporal  $O(1)$ ,  $n - 1$  operaciones *Unión* tendrán un coste temporal  $O(n)$ . Debido a que tras realizar todas las operaciones de unión, tendremos un único árbol que contendrá los  $n$  elementos, el coste temporal de realizar  $m$  operaciones *Buscar* será  $O(mn)$ .

El coste obtenido viene determinado por el hecho de que, tal como se realiza la operación *Unión*, tras realizar  $k$  operaciones de *Unión* podría obtenerse un árbol de altura  $k$ . Es decir, un árbol en el que todos sus nodos formen una lista enlazada.

Utilizando dos técnicas heurísticas podemos mejorar el coste temporal de las operaciones a base de reducir la altura del árbol.

### Unión por altura o rango

**Estrategia:** la unión de dos conjuntos se realiza de tal forma que la raíz del árbol con menos altura apunta a la raíz del árbol con más altura. De esta forma, la altura del árbol que se obtiene tras unir un árbol de altura  $h_1$  con un árbol de altura  $h_2$ , será  $\max(h_1, h_2)$  si  $h_1 \neq h_2$ , o bien  $h_1 + 1$  si  $h_1 = h_2$  (ver figura 6.35).

Para llevar a cabo este heurístico será necesario mantener, para cada nodo  $n_i$ , la altura a la que se encuentra el nodo  $n_i$ .

Puede demostrarse que si se realiza la unión por rango, la altura de un árbol de  $n$  elementos nunca será superior a  $\lfloor \log n \rfloor$ . Por lo tanto, el coste de realizar la operación *Buscar*, en el peor caso, utilizando en la operación *Unión* el heurístico “unión por rango” es de  $O(\log n)$  y, por lo tanto, el coste de realizar  $m$  operaciones de búsqueda será  $O(m \log n)$ .

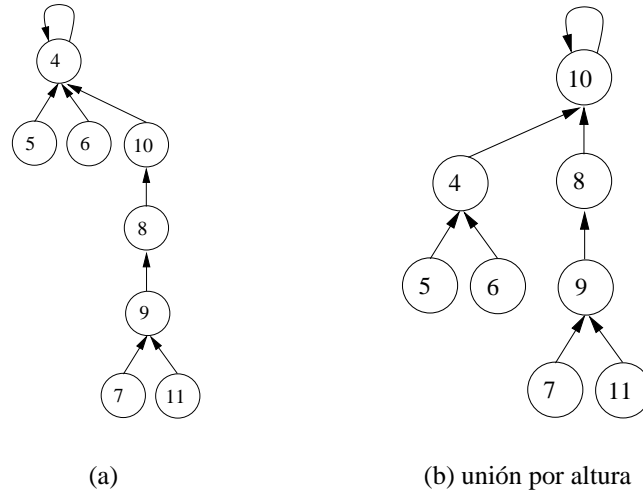


Figura 6.35: En ambas figuras se muestra un árbol que representa la unión de los subconjuntos 12 y 8 de la figura 6.32. En la figura (a), la unión de ambos subconjuntos se ha realizado enlazando arbitrariamente una raíz de un árbol con la raíz del otro, lo que, desaconsejablemente, ha causado que el árbol resultante crezca en altura. En la figura (b), al aplicar el heurístico *unión por rango o altura*, el árbol resultante de la unión sigue teniendo la misma altura que el árbol de mayor altura de los que se unieron; por lo tanto se ha evitado que el árbol obtenido tras realizar la unión crezca en altura.

### Compresión de caminos

**Estrategia:** Cuando se busca un elemento se aprovecha la búsqueda, de tal manera, que todos los nodos que forman parte del camino de búsqueda se enlazan directamente con la raíz del árbol; de esta forma se consigue reducir considerablemente la altura del árbol (ver figura 6.36).

Combinando ambos heurísticos, el tiempo requerido para realizar  $m$  operaciones de búsqueda, en el peor caso, es  $O(m\alpha(m, n))$ , donde  $\alpha(m, n)$  es una inversa de la función de Ackerman, la cual crece muy lentamente. Normalmente, en cualquier aplicación que se utilice un MF-Set,  $\alpha(m, n) \leq 4$ .

En definitiva, puede decirse que, en la práctica, aplicando ambos heurísticos, realizar  $m$  operaciones de búsqueda tiene un coste temporal casi lineal en  $m$ .

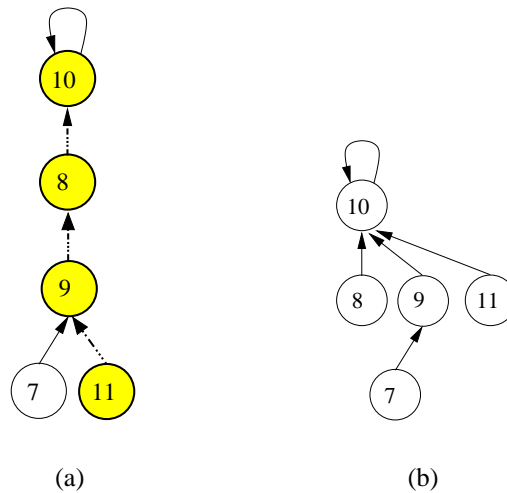


Figura 6.36: En la figura (b) se muestra como quedaría el árbol de la figura (a), tras realizar sobre él una operación búsqueda del elemento 11 y aplicar el heurístico *compresión de caminos*.

## 6.6. Otras Estructuras de Datos para Conjuntos

En la siguiente sección se pretende ofrecer una visión muy general de otras estructuras de datos clásicas y de gran interés práctico.

Aunque no se realizará una descripción completa de las estructuras, sí que se pretende mostrar en cierta manera un pequeño compendio de estructuras de datos para completar la lista de las estructuras vistas a lo largo de este tema.

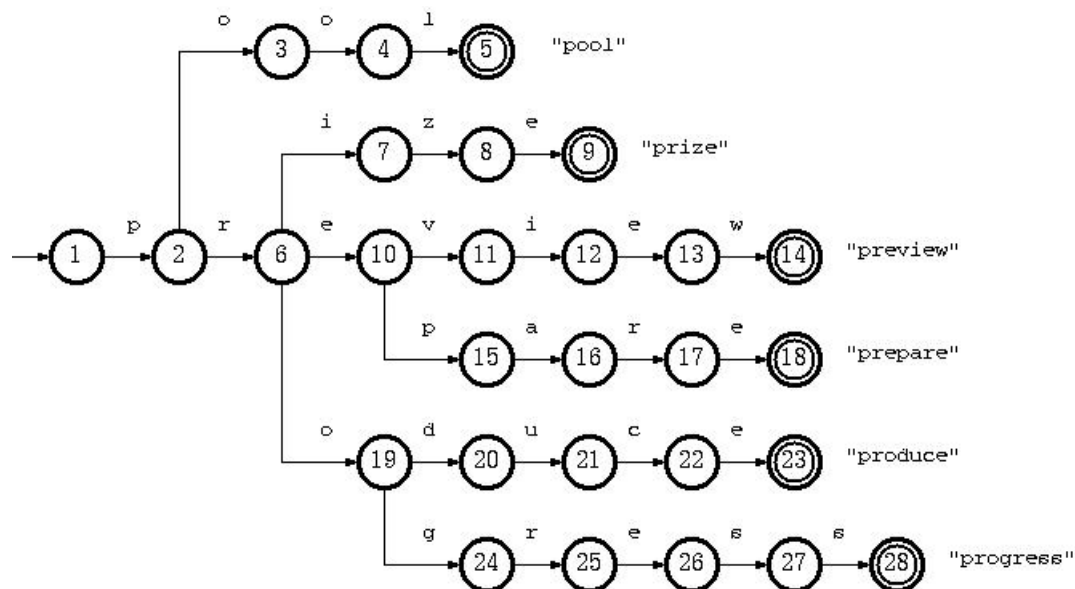
Cada estructura tiene unas aplicaciones específicas, las cuales comentaremos brevemente en cada apartado.

### 6.6.1. Tries

Un trie es un tipo de árbol de búsqueda. El término *trie* proviene de abreviar la palabra inglesa *Retrieval*, que se refiere a la operación de acceder a la información guardada en la memoria del computador.

Tradicionalmente, los tries se han utilizado para guardar diccionarios de manera que las palabras que tienen prefijos comunes (secuencias iniciales de símbolos iguales) utilizan la misma memoria para guardar dichos prefijos. Esta estructura ahorra espacio respecto a un almacenamiento simple de las cadenas debido a la *compactación* de los prefijos comunes.

Si suponemos un conjunto de palabras formado por las palabras:  $\{pool, prize, preview, prepare, produce, progress\}$ , y las almacenamos en un trie, obtendremos:



Cuando buscamos una palabra determinada, la búsqueda comienza en el nodo raíz. Después desde el principio al final de la palabra, se toma carácter a carácter para determinar el siguiente nodo al que ir. Se elige la arista etiquetada con el mismo carácter. Cada paso de esta búsqueda consume un carácter de la palabra y desciende un nivel en el árbol. Si se agota toda la palabra y se ha alcanzado un nodo hoja, entonces habremos encontrado la información correspondiente a esa palabra. Si nos quedamos parados en un nodo, tanto porque no existe ninguna arista etiquetada con el carácter actual, tanto porque se ha agotado la palabra en un nodo interno, entonces esto indica que la palabra no es reconocida por el trie.

El tiempo necesario para llegar desde el nodo raíz al nodo hoja (proceso de búsqueda de una palabra) no depende del tamaño del trie, sino que es proporcional a la longitud de la palabra, lo cual convierte al trie en una estructura de datos muy eficiente.

Un trie puede entenderse como un tipo de autómata finito determinista (AFD), donde cada nodo se corresponde con un estado del AFD y cada arista del árbol se corresponde con una transición del AFD.

En general un AFD se representa con una matriz de transición, en la que las filas se corresponden con los estados y las columnas se corresponden con las etiquetas o símbolos para realizar una transición. En cada posición de la matriz se almacena el siguiente estado al que transicionar para un estado cuando la entrada es equivalente a la etiqueta correspondiente. Esta representación mediante

una matriz bidimensional puede resultar muy eficiente respecto al coste temporal, pero un poco extraña vista desde el coste espacial, puesto que la mayoría de nodos tendrán pocas aristas, dejando la mayoría de las posiciones de la matriz vacías. Existen representaciones más eficientes espacialmente, pero nos las veremos aquí.

### 6.6.2. Árboles Balanceados

Otras estructuras de datos muy utilizadas para almacenar elementos de manera que se permita una búsqueda eficiente de estos una vez construida las estructuras, son los árboles balanceados.

Un árbol balanceado es un árbol donde ninguna hoja está más alejada de la raíz que cualquier otra hoja.

Se pueden definir varios esquemas de balanceo, con lo que se permite una definición diferente de *más lejos* y diferentes algoritmos para mantener el árbol balanceado conforme a las operaciones de actualización del árbol.

En la tres siguientes secciones veremos algunos de los tipos de árboles balanceados más importantes según el criterio de balanceo elegido.

#### Árboles AVL

Un árbol AVL es un árbol binario de búsqueda balanceado. Su nombre viene de sus inventores Adelson, Velskii y Landis. No están balanceados perfectamente como veremos en algún ejemplo.

Un árbol AVL es un árbol binario de búsqueda que cumple las siguientes propiedades:

- Las alturas de los subárboles de cada nodo difieren como mucho en 1.
- Cada subárbol es un árbol AVL.

Como hemos comentado anteriormente, debe tenerse cuidado con esta definición ya que permite árboles aparentemente no balanceados. En la figura 6.37 podemos ver algunos ejemplos de árboles AVL.

Las operaciones de búsqueda de un elemento, inserción y borrado tienen un coste  $O(\log n)$ , siendo  $n$  el número de elementos.

#### Árboles 2-3

Un árbol 2-3 se define como un árbol vacío (cuando tiene 0 nodos) o un nodo simple (cuando tiene un único nodo) o un árbol con múltiples nodos que cumplen las siguientes propiedades:

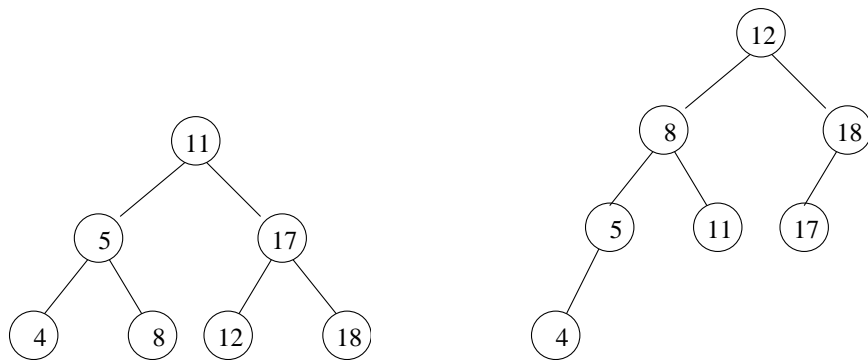


Figura 6.37: Ejemplos de árboles AVL.

- Cada nodo interior tiene 2 o tres hijos.
- Cada camino desde la raíz a una hoja tiene la misma longitud.

Diferenciamos los nodos internos de las hojas creando unos nodos internos con la siguiente información:

$p_1$	$k_1$	$p_2$	$k_2$	$p_3$
-------	-------	-------	-------	-------

Donde cada campo es:

- $p_1$ : puntero al primer hijo.
- $p_2$ : puntero al segundo hijo.
- $p_3$ : puntero al tercer hijo (si existe).
- $k_1$ : clave más pequeña que sea un descendiente del segundo hijo.
- $k_2$ : clave más pequeña que sea un descendiente del tercer hijo.

Los nodos hoja tienen solo la información referente a la clave correspondiente. En la figura 6.38 puede observarse un ejemplo de árbol 2-3.

Los valores almacenados en los nodos internos se usan para guiar el proceso de búsqueda. Para buscar un elemento con clave  $x$ , empezamos en la raíz y suponemos que  $k_1$  y  $k_2$  son los dos valores almacenados aquí. A partir de aquí realizamos las siguientes comprobaciones:

- Si  $x < k_1$ , seguir la búsqueda por el primer hijo.

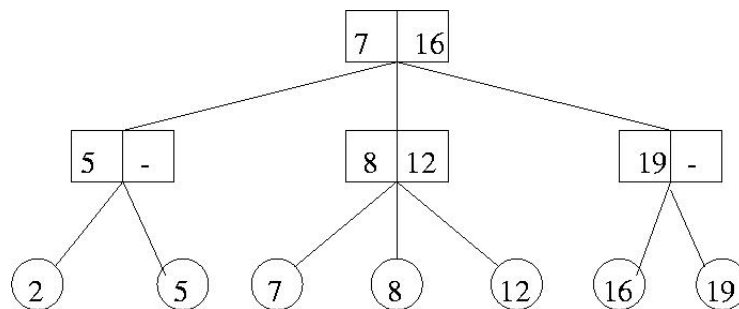


Figura 6.38: Ejemplo de árbol 2-3.

- Si  $x \geq k_1$  y el nodo tiene solo 2 hijos, seguir la búsqueda por el segundo hijo.
- Si  $x \geq k_1$  y el nodo tiene 3 hijos, seguir la búsqueda por el segundo hijo si  $x < k_2$  y por el tercer hijo si  $x \geq k_2$ .

Aplicar el mismo esquema a cada uno de los nodos que vayan formando parte del camino de búsqueda. El proceso acaba cuando se encuentra una hoja con la clave  $x$  (elemento encontrado) o se llega a una hoja con una clave diferente a  $x$  (elemento no perteneciente al árbol 2-3).

## B-árboles

Los B-árboles o árboles B son una generalización de los árboles 2-3. Esta estructura de datos es eficiente para almacenamiento externo de datos y suele aplicarse de manera estándar para la organización de los índices en sistemas de bases de datos. Además, proporciona la minimización de los accesos a disco para las aplicaciones que manejan bases de datos.

Un árbol B de orden  $n$  es un árbol de búsqueda  $n$ -ario con las siguientes propiedades:

- La raíz es una hoja o tiene por lo menos dos hijos.
- Cada nodo, excepto el raíz y las hojas, tiene entre  $\lceil \frac{n}{2} \rceil$  y  $n$  hijos.
- Cada camino desde la raíz a una hoja tiene la misma longitud.
- Cada nodo interno tiene hasta  $(n - 1)$  valores de claves y hasta  $m$  punteros a sus hijos.

- Como se ha comentado en el punto anterior, los elementos se almacenan típicamente repartidos entre los nodos internos y las hojas. Aunque en algunas variantes de implementación sólo se almacenan en las hojas, como veremos un poco más adelante.
- Un B-árbol puede verse como un índice jerárquico en el cual la raíz es el primer nivel de indizado.
- Cada nodo interno es de la forma:

$p_1$	$k_1$	$p_2$	$k_2$	.....	$k_{n-1}$	$p_n$
-------	-------	-------	-------	-------	-----------	-------

donde:

- $p_i$  es un puntero al  $i$ -ésimo hijo,  $1 \leq i \leq n$ .
- $k_i$  son los valores de las claves, las cuales guardan un orden  $k_1 < k_2 < \dots < k_{n-1}$  de manera que:
  - todas las claves en el subárbol apuntado por  $p_1$  son menores que  $k_1$ .
  - Para  $2 \leq i \leq n - 1$ , todas las claves en el subárbol apuntado por  $p_i$  son mayores o iguales que  $k_{i-1}$  y menores que  $k_i$ .
  - Todas las claves en el subárbol apuntado por  $p_n$  son mayores o iguales que  $k_{n-1}$ .

En la figura 6.39 puede observarse un ejemplo de árbol B.

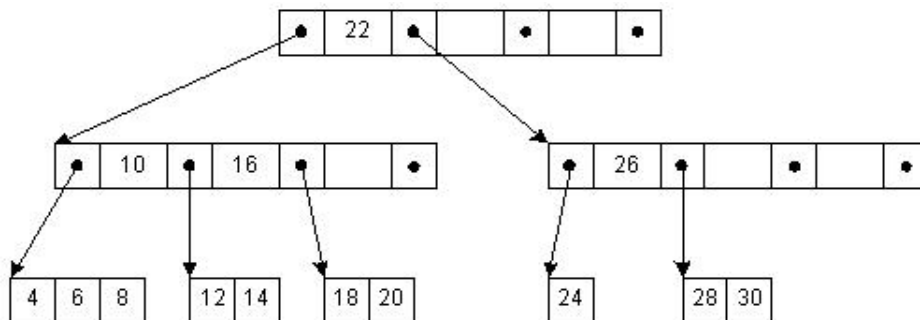


Figura 6.39: Ejemplo de árbol B.

Un árbol B+ es un árbol B en el que se considera que las claves almacenadas en los nodos internos no son útiles como claves (sólo se utilizan para operaciones de búsqueda). Por lo tanto, todas esas claves de los nodos internos están duplicadas



en las hojas. Esto tiene la ventaja de que todas las hojas están enlazadas secuencialmente y se podría acceder a toda la información de los elementos guardados en el árbol sin necesidad de visitar nodos internos. En la figura 6.40 se muestra un ejemplo de árbol B+. Esta modificación de los B-árboles consigue una mejor utilización del espacio en el árbol (a nivel de implementación) y mejora la eficiencia de determinados métodos de búsqueda.

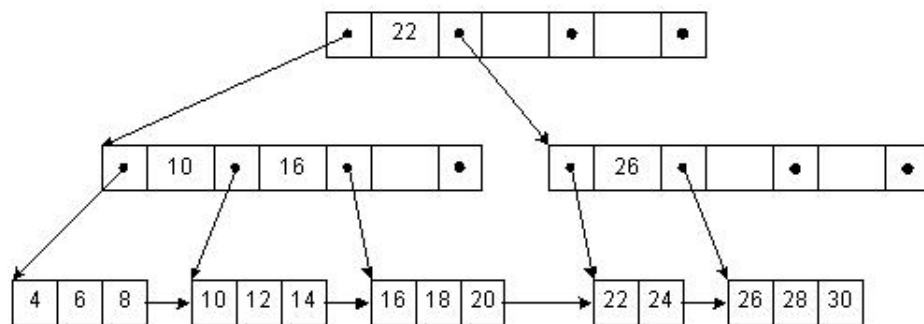


Figura 6.40: Ejemplo de árbol B+.

## 6.7. Ejercicios

### 6.7.1. Tablas de dispersión

#### Ejercicio 1:

-Dadas las siguientes definiciones de tipos, constantes y variables, para declarar y manipular tablas de dispersión:

```
#define NCUBETAS ...
typedef struct snodo {
    char *pal;
    struct snodo *sig;
} nodo;
typedef nodo * Tabla[NCUBETAS];
```

```
Tabla T1, T2, T3;
```

```
int ncubetas1, ncubetas2, ncubetas3;
```

Donde se considera que *ncubetas1*, *ncubetas2* y *ncubetas3* son variables que indican el número de cubetas de *T1*, *T2* y *T3*, respectivamente. Además, las siguientes funciones están disponibles y pueden utilizarse cuando se considere oportuno:

```
/* Inicializa una tabla vacia */
```

```
void crea_Tabla(Tabla T);
```

```
/* Inserta la palabra pal en la tabla T */
```

```
void inserta(Tabla T, char *pal);
```

```
/* Devuelve un puntero al nodo que almacena la palabra */
```

```
/* pal, o NULL si no la encuentra */
```

```
nodo *buscar(Tabla T, char *pal);
```

Se pide:

- Escribir una función que cree una tabla *T3* con los elementos pertenecientes a la unión de los conjuntos almacenados en las tablas *T1* y *T2*. Estudia cuál es el coste temporal del algoritmo.
- Escribir una función que cree una tabla *T3* con los elementos pertenecientes a la diferencia *T2 - T1*; es decir, aquellos elementos que están en *T2* y no están en *T1*. Estudia cuál es el coste temporal del algoritmo.

**Solución:**

a) Función que realiza la unión de dos tablas hash:

```
void union(Tabla T1, Tabla T2, Tabla T3)
{
    int i;
    nodo *aux;

    crea_Tabla(T3);

    /* Metemos en T3 los elementos de T1 */
    for (i=0; i<ncubetas1; i++){
        aux = T1[i];
        while (aux != NULL){
            inserta(T3, aux->pal);
            aux = aux->sig;
        }
    }
    /* Metemos en T3 los elementos de T2 */
    for (i=0; i<ncubetas2; i++){
        aux = T2[i];
        while (aux != NULL){
            inserta(T3, aux->pal);
            aux = aux->sig;
        }
    }
}
```

El coste del algoritmo es  $O(n_1 + n_2)$ , siendo  $n_1$  el número de elementos almacenados en T1 y  $n_2$  el número de elementos almacenados en T2. Esto es debido a que habrá que recorrer todos los elementos de T1 e insertarlos en T3 y recorrer todos los elementos de T2 e insertarlos en T3 igualmente. Como la operación de inserción se considera que tiene un coste constante, entonces el coste de esta función viene dado por el número de elementos que habrá que insertar en T3, que son los de T1 y T2.

b) Función que realiza la diferencia de dos tablas hash:

```
void diferencia(Tabla T1, Tabla T2, Tabla T3)
{
    int i;
    nodo *aux;

    crea_Tabla(T3);

    for (i=0; i<ncubetas2; i++){
        aux = T2[i];
        while (aux != NULL){
            if (buscar(T1, aux->pal) == NULL)
                inserta(T3, aux->pal);
            aux = aux->sig;
        }
    }
}
```

El coste del algoritmo es  $O(n_2)$ , siendo  $n_2$  el número de elementos almacenados en T2. Esto es debido a que habrá que recorrer todos los elementos de T2, para cada uno de ellos buscar si está en T1 y si no lo está, entonces insertarlo en T3. Como las operaciones de búsqueda e inserción se considera que tienen un coste constante, entonces el coste de esta función vendrá dado por el número de elementos que hay que recorrer en T2.

---

### Ejercicio 2:

-Dadas las siguientes definiciones de tipos y constantes para declarar y manipular tablas de dispersión que almacenan números enteros:

```
#define NCUBETAS ...  
typedef struct snodo {  
    int numero;  
    struct snodo *sig;  
} nodo;  
typedef nodo * Tabla[NCUBETAS];
```

Se pide escribir una función:

```
int minimo(Tabla T, int ncubetas)
```

que obtenga el mínimo de un conjunto de números enteros representado como una tabla de dispersión. Analiza cuál es el coste temporal del algoritmo.

### Solución:

Función para calcular el mínimo de una tabla hash de números enteros:

```
int minimo(Tabla T, int ncubetas)
{
    int i,j;
    nodo *aux;
    int min;

    /* Buscamos el primer elemento que haya en la tabla */
    /* y lo ponemos como el minimo. */
    i=0;
    while ((T[i]==NULL) && (i<ncubetas)) i++;
    min = T[i]->numero;

    /* Buscamos en el resto de la tabla si hay otro */
    /* elemento menor. */
    for (j=i; j<ncubetas; j++){
        aux = T[j];
        while (aux != NULL){
            if (aux->numero < min) min = aux->numero;
            aux = aux->sig;
        }
    }
    return(min);
}
```

El coste del algoritmo es  $O(n)$ , siendo  $n$  el número de elementos del conjunto, esto es, la cantidad de enteros que están almacenados en la tabla de dispersión. Esto es debido a que habrá que recorrer todos los elementos para saber cuál es el mínimo.

---

## 6.7.2. Árboles binarios de búsqueda

### Ejercicio 3:

- Dadas las siguientes definiciones de tipos y variables para representar árboles binarios de búsqueda:

```
typedef ... tipo_baseT;  
typedef struct snodo {  
    tipo_baseT clave;  
    struct snodo *hizq, *hder;  
} abb;
```

abb \*T;

Se pide:

- a) Escribe una versión recursiva del algoritmo visto en clase de teoría que obtiene el máximo de un árbol binario de búsqueda.

```
abb *maximo(abb *T)
```

- b) Escribe un algoritmo recursivo que, dado un árbol binario indique si es de búsqueda o no.

```
int es_abb(abb *T)
```

### Solución:

- a) El esquema para la función que obtiene el máximo de un árbol binario de búsqueda de manera recursiva es sencillo. Para buscar el máximo siempre debíamos buscar por el hijo derecho del nodo actual en el caso de que tuviera, si no tenía hijo derecho es porque él era el máximo. El esquema recursivo consistirá en si un nodo tiene hijo derecho, el problema se reduce a buscar el máximo de su subárbol derecho, en el caso de que no tenga hijo derecho, él es el máximo:

```
abb *maximo(abb *T) {  
    abb *max=NULL;  
  
    if (T!=NULL)  
        if (T->hder!=NULL) max = maximo(T->hder);  
        else max = T;  
    return (max);  
}
```

- b) La función debe devolver 0 (falso) si el árbol binario no es de búsqueda y 1 (verdadero) en caso contrario. Para comprobar si un árbol binario es de búsqueda podemos comprobar si para cada nodo del árbol, se cumple la propiedad de árbol binario de búsqueda a nivel local, esto es, si para el nodo actual que estemos consultando, se cumple que su hijo izquierdo tiene una clave menor que la suya y su hijo derecho tiene una clave mayor que la suya, entonces solo quedará por comprobar si su subárbol izquierdo es o no un árbol binario de búsqueda y si su subárbol derecho es o no un árbol binario de búsqueda. Aquí es donde tenemos el esquema recursivo del problema.

Si nos fijamos, veremos que realmente lo que hay que hacer es un recorrido del árbol, donde la acción a realizar para cada nodo es comprobar si su clave es mayor que la de su hijo izquierdo y menor que la de su hijo derecho. La función que indica si un árbol binario es de búsqueda o no es la siguiente:

```
int es_abb(abb *T) {
    int es=1;

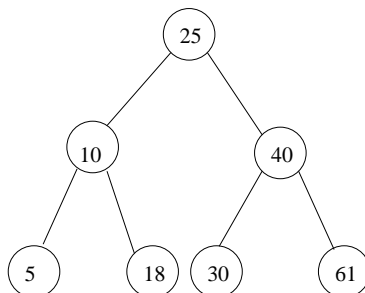
    /* Comprobamos si el hijo izq cumple condicion */
    /* de abb y si el subarbol izq es abb. */
    if (T->hizq!=NULL) {
        if (T->clave < T->hizq->clave) return(0);
        else es = es_abb(T->hizq);
    }
    if (es)
        /* Comprobamos si el hijo der cumple */
        /* condicion de abb y si el subarbol der */
        /* es abb. */
        if (T->hder!=NULL) {
            if (T->clave > T->hder->clave) return(0);
            else es = es_abb(T->hder);
        }
    return(es);
}
```

---



#### Ejercicio 4:

-Sea  $A$  un árbol binario de búsqueda con  $n$  elementos y todos sus niveles completos. Por ejemplo, sea el árbol de enteros, con  $n = 7$  de esta figura:



Escribir un algoritmo "Divide y Vencerás" en lenguaje C,

```
int select_kmenor(arbol *A, int k, int n)
```

que encuentre con un coste  $O(\log n)$  el elemento  $k$ -ésimo en la secuencia ordenada de los elementos de  $A$  sin ordenar dichos elementos (esto es, debe buscar el  $k$ -ésimo menor elemento), con  $1 \leq k \leq n$ . Por ejemplo, en el árbol de la figura anterior, si  $k = 5$ , debe devolverse el elemento 30.

NOTA: Para intentar hallar la función correspondiente que resuelva este problema, hay que pensar cómo quedarían los elementos del árbol si estuvieran ordenados en un vector.

Como los árboles binarios especificados deben cumplir que son completos y por la propiedad de ser árbol binario de búsqueda, el elemento que estuviera almacenado en la raíz del árbol quedaría justo en la posición central de la secuencia de elementos ya ordenados. Además esa posición sería la posición  $\lfloor n/2 \rfloor$ . Además, todos los elementos de su subárbol izquierdo estarán antes que la posición  $\lfloor n/2 \rfloor$  de la secuencia ordenada y todos los elementos de su subárbol derecho estarán en las posiciones posteriores a  $\lfloor n/2 \rfloor$ . Así pues, como estamos buscando el elemento que ocupe la posición  $k$  en la secuencia final, si  $k$  es menor que  $\lfloor n/2 \rfloor$  entonces el elemento a buscar estará en el subárbol izquierdo del raíz y podemos centrar la búsqueda en ese subárbol, pero si  $k$  es mayor que  $\lfloor n/2 \rfloor$  entonces el elemento a buscar estará en el subárbol derecho del raíz y debemos centrar la búsqueda en ese subárbol; en el caso en que  $k = \lfloor n/2 \rfloor$  entonces ya hemos encontrado el elemento buscado.

Cuando comencemos la búsqueda en el subárbol correspondiente (el izquierdo o el derecho) debemos aplicar la misma estrategia, por ello, podemos escribir la función siguiendo un esquema recursivo o iterativo, según se desee.

OJO!: ¿crees que habrá que cambiar la  $k$  según busquemos por el subárbol izquierdo o el derecho?

### Solución:

Los parámetros de la función serán el árbol binario de búsqueda, la posición  $k$  que buscamos y el número de nodos que forman el árbol binario de búsqueda actual en el que estamos buscando.

Un aspecto a tener en cuenta es que si buscamos por el subárbol derecho entonces la  $k$  debe cambiar, esto es debido a que ahora la posición que buscamos dentro de esa subsecuencia no se corresponde con la posición  $k$  puesto que el primer elemento de la secuencia ordenada de los elementos del subárbol derecho no está en la posición 0.

- Solución recursiva:

```
int select_kmenor(arbol *A, int k, int n)
{
    int n_subarb;

    n_subarb = (n-1)/2;
    if (k <= n_subarb)
        return(select_kmenor(A->hizq, k, n_subarb));
    else
        if (k > n_subarb + 1)
            return(select_kmenor(A->hder, k-(n_subarb+1), n_subarb));
        else
            return(A->clave);
}
```

- Solución iterativa:

```
int select_kmenor(arbol *A, int k, int n)
{
    int n_subarb;
    arbol *aux;

    aux = A;
    do {
        n_subarb = (n-1)/2;
        if (k <= n_subarb)
            aux = aux->hizq;
        else if (k > n_subarb + 1){
            aux = aux->hder;
            k = k - (n_subarb + 1);
        }
        n = n_subarb;
    }while( k != n + 1 );
    return(aux->clave);
}
```

---

### 6.7.3. Montículos (Heaps)

#### Ejercicio 5:

-Escribe un algoritmo recursivo que, dada una clave  $x$ , obtenga, de la forma más eficiente posible, la posición que ocupa en un montículo (la posición del vector que representa el montículo). Si la clave no se encuentra, el algoritmo debe devolver el valor -1. Suponer que existe una variable global `talla_Heap` que indica el número de nodos que forman el Heap.

¿Cuál sería el coste temporal del algoritmo en el peor caso?

#### Solución:

Puede aparentar que en un montículo no exista ninguna propiedad que pueda permitirnos facilitar la búsqueda de un elemento, por lo que tendríamos que aplicar un recorrido tradicional de árboles (preorden, postorden o inorden) para llevar a cabo la búsqueda. Sin embargo, sí que podemos optimizar un poco el proceso de búsqueda utilizando la propiedad de montículo que dice que para un determinado nodo la clave de éste ha de ser mayor o igual que la de sus dos posibles hijos. La idea es que si llegamos a un nodo cuya clave es menor que la clave  $x$  que estamos buscando, entonces no seguiremos buscando por sus subárboles, ya que estos contendrán nodos cuyas claves es segura que van a ser menores que  $x$  y por tanto no podrán ser iguales. Estaremos evitando así el tener que hacer una exploración completa del árbol.

La función que realiza la búsqueda es:

```

int busca(int *M, int act, int x) {
    int pos;

    if (M[act] == x) return(act);
    else
        if (M[act] < x) return(-1);
        else {
            pos = -1
            /* Si tiene hijo izquierdo, buscamos */
            /* por subarbol izquierdo. */
            if ((2*act)<=talla_Heap) pos = busca(M,2*act,x);
            /* Si tiene hijo derecho, buscamos */
            /* por subarbol derecho. */
            if ((pos == -1) && ((2*act)+1)<=talla_Heap))
                pos = busca(M,(2*act)+1,x);
            return(pos);
        }
}

```

La llamada inicial sería `pos=busca(M,n,1,x)`.

El peor caso será cuando estemos buscando una clave  $x$  que sea menor que todas las claves que estén almacenadas en el montículo y además no se encuentre en él. En este caso habrá que explorar todas las ramas del montículo puesto que nunca se cumplirá la condición de que  $x$  sea menor que ninguna clave del montículo, además no podremos saber que  $x$  no está en el montículo hasta que no hayamos recorrido todos los nodos. Así pues, para el peor caso el coste será  $O(n)$ , siendo  $n$  el número de nodos del montículo.

---

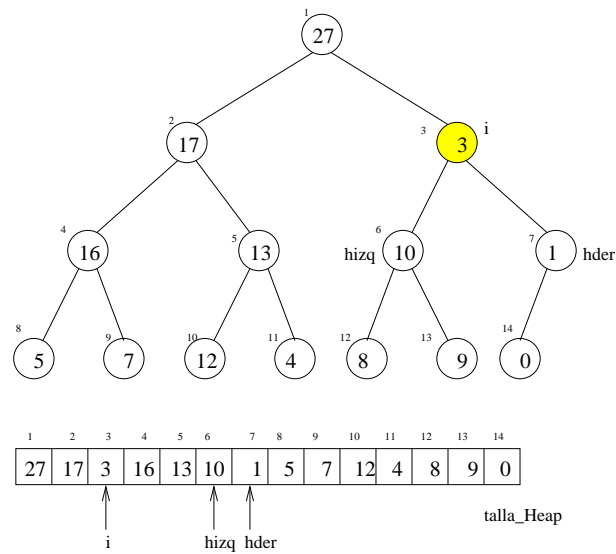
### Ejercicio 6:

-Realiza una traza de la llamada  $\text{heapify}(M, 3)$  para el vector:

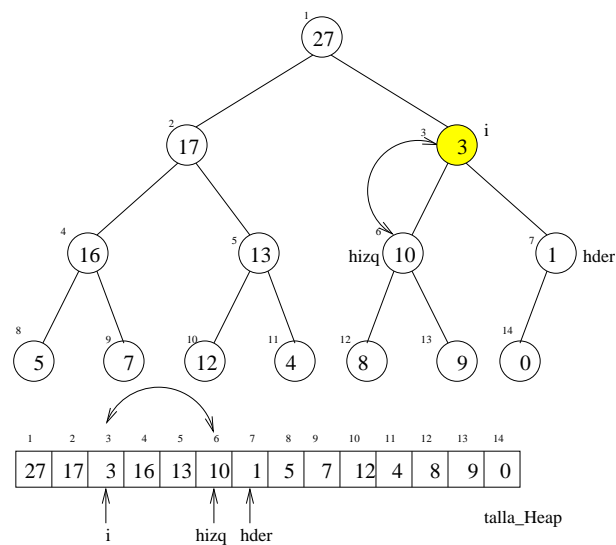
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
M =	27	17	3	16	13	10	1	5	7	12	4	8	9	0

### Solución:

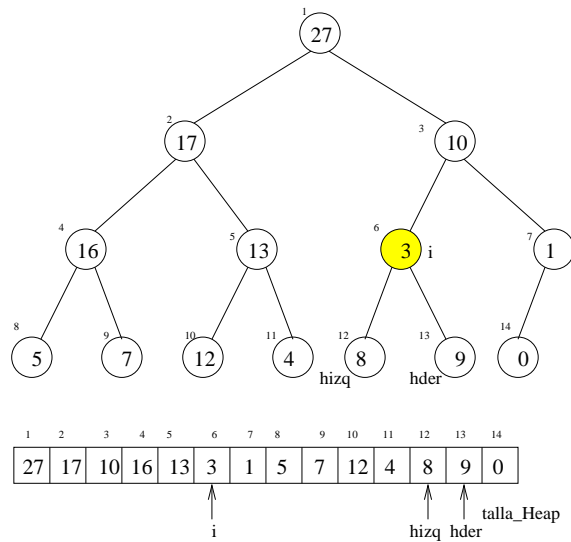
- El nodo 3 no cumple la propiedad de montículo:



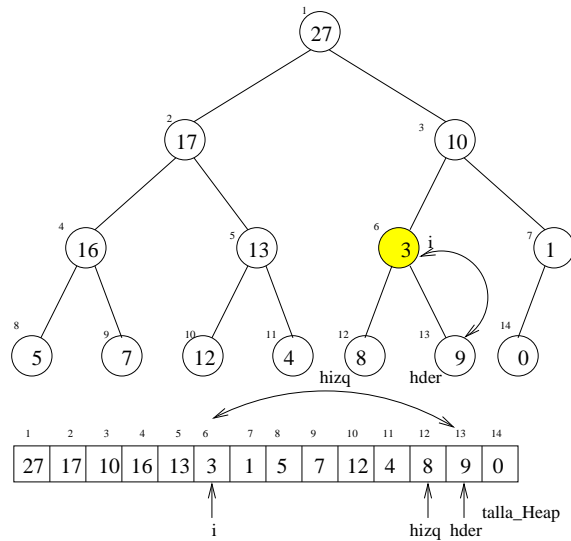
- El nodo de clave mayor es el hijo izquierdo de i, se intercambian las claves y se hace una llamada recursiva a  $\text{heapify}(M, 6)$ :



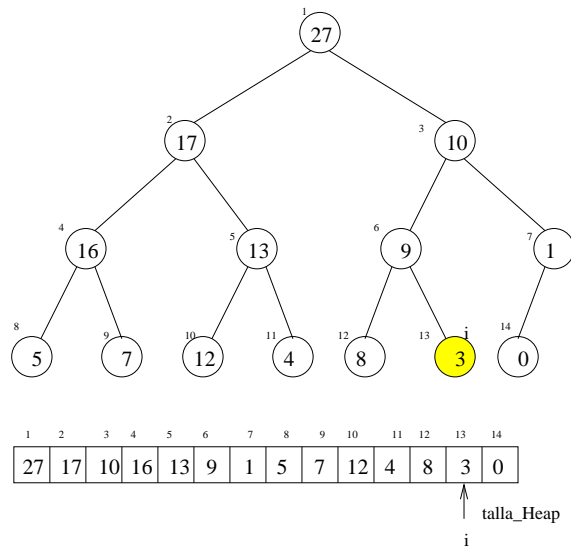
- El nodo 6 no cumple la propiedad de montículo:



- El nodo de clave mayor es el hijo derecho de  $i$ , se intercambian las claves y se hace una llamada recursiva a  $\text{heapify}(M, 13)$ :



- El nodo 13 cumple la propiedad de montículo, termina la llamada a  $\text{heapify}(M, 13)$ , termina la llamada a  $\text{heapify}(M, 6)$  y por último termina la llamada a  $\text{heapify}(M, 3)$ . Obsérvese que ahora el subárbol que comienza en el nodo 3 sí es un montículo:



### Ejercicio 7:

-¿Cuál es el resultado de realizar una llamada a la función `heapify(M, i)` para  $i > \text{talla\_Heap}/2$ ?

¿Y si  $i < (\text{talla\_Heap}/2)+1$  y la clave del nodo  $i$  es mayor que la de sus hijos?

### Solución:

Para el caso en que  $i > \text{talla\_Heap}/2$  la función `heapify` no hace nada puesto que detecta que el nodo  $i$  es una hoja y por tanto no puede incumplir la propiedad de montículo.

Si  $i < (\text{talla\_Heap}/2)+1$  y la clave del nodo  $i$  es mayor que la de sus hijos la función `heapify` también deja inalterado el montículo puesto que detecta que el nodo  $i$ , a pesar de no ser una hoja, sí que cumple la propiedad de montículo.



### 6.7.4. MF-sets

#### Ejercicio 8:

-Dado el siguiente MF-set:

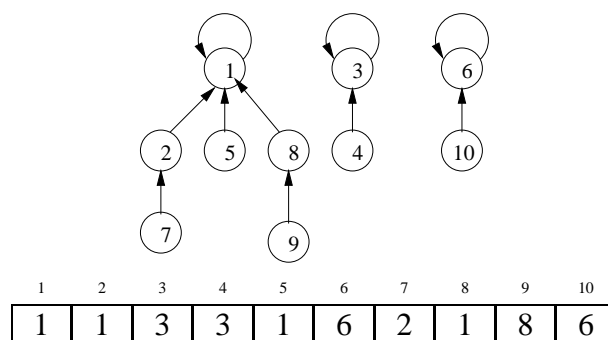
$$M = \begin{array}{c|c|c|c|c|c|c|c|c|c} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ \hline 1 & 1 & 3 & 3 & 1 & 6 & 2 & 1 & 8 & 6 \end{array}$$

Se pide:

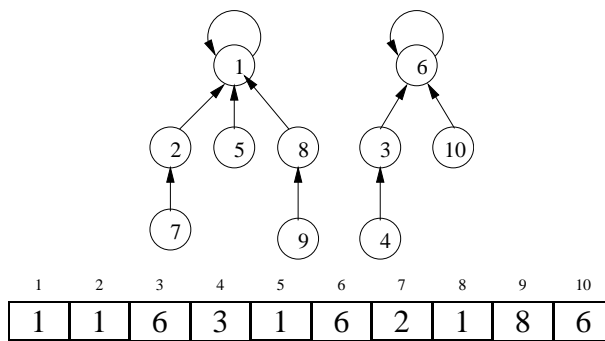
- Dibuja cada uno de los árboles que representan los subconjuntos disjuntos que contiene el MF-set.
  - Muestra cómo se van transformando los árboles, después de ejecutar cada una de las operaciones de la secuencia indicada. Utiliza los heurísticos *unión por rango* y *compresión de caminos*. En la unión, en caso de que la altura de los árboles a unir sea idéntica, la raíz del primer árbol debe pasar a ser hijo de la raíz del segundo árbol.
- Union(3,6)
  - Buscar(4)
  - Union(1,6)
  - Buscar(10)

#### Solución:

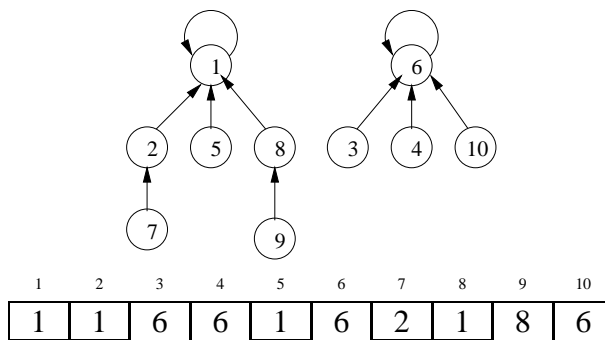
- El MF-set se representa de esta forma:



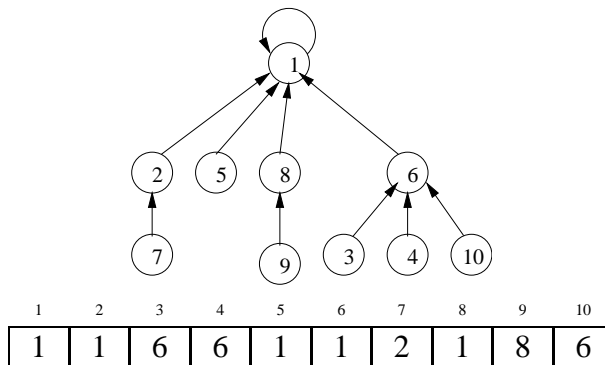
1. Union(3,6)



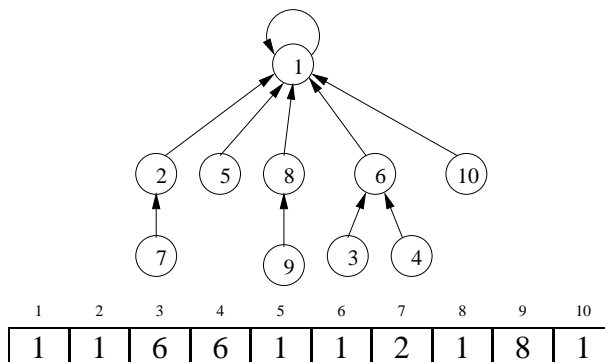
2. Buscar(4)



3. Union(1,6)



4. Buscar(10)



# Tema 7

## Grafos

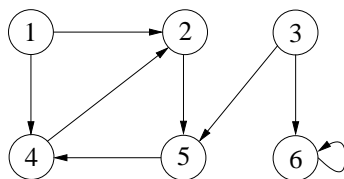
### 7.1. Definiciones

Los grafos son modelos que permiten representar relaciones entre elementos de un conjunto. A continuación veremos una serie de definiciones básicas:

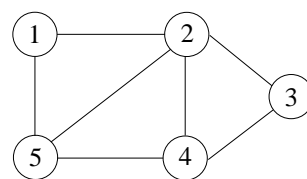
Un **grafo** es un par  $(V, E)$ , donde  $V$  es un conjunto de elementos denominados *vértices* o *nodos*, sobre los que se ha definido una relación; y  $E$  es un conjunto de pares  $(u, v)$ ,  $u, v \in V$ , denominados *aristas* o *arcos*, que indica que  $u$  está relacionado con  $v$  de acuerdo a la relación definida sobre  $V$ .

Un **grafo dirigido** es aquél en el que la relación definida sobre  $V$  no es simétrica. Cada arista en  $E$  es un par ordenado de vértices  $(u, v)$ ; es decir, la arista  $(u, v)$  y la arista  $(v, u)$  son aristas distintas.

Un **grafo no dirigido** es aquél en el que la relación definida sobre  $V$  sí es simétrica. Cada arista en  $E$  es un par no ordenado de vértices  $\{u, v\}$  donde  $u, v \in V$  y  $u \neq v$ ; es decir, la arista  $(u, v)$  y  $(v, u)$  son la misma arista, y no puede existir una arista de un vértice sobre sí mismo.



(a) Grafo dirigido  $G(V, E)$ .  $V = \{1, 2, 3, 4, 5, 6\}$   
 $E = \{(1, 2), (1, 4), (2, 5), (3, 5), (3, 6), (4, 2), (5, 4), (6, 6)\}$



(b) Grafo no dirigido  $G(V, E)$ .  
 $V = \{1, 2, 3, 4, 5\}$   
 $E = \{\{1, 2\}, \{1, 5\}, \{2, 3\}, \{2, 4\}, \{2, 5\}, \{3, 4\}, \{4, 5\}\}$

Figura 7.1: Ejemplo de grafos.

Un **camino** desde un vértice  $u \in V$  a un vértice  $v \in V$ , es una secuencia de vértices  $v_1, v_2, \dots, v_k$  tal que  $u = v_1, v = v_k$ , y  $(v_{i-1}, v_i) \in E$ , para  $i = 2, \dots, k$ .

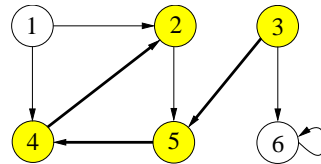


Figura 7.2: Ejemplo de camino desde el vértice 3 al vértice 2:  $\langle 3, 5, 4, 2 \rangle$ .

La **longitud de un camino** es el número de arcos del camino. El camino que se muestra en la figura 7.2 es de longitud 3.

Un **camino simple** es un camino en el que todos sus vértices, excepto, tal vez, el primero y el último, son distintos. El camino que se muestra en la figura 7.2 es un camino simple; sin embargo, el camino  $\langle 3, 5, 4, 2, 5 \rangle$  no es simple.

Un **ciclo** es un camino simple  $v_1, v_2, \dots, v_k$  tal que  $v_1 = v_k$ .

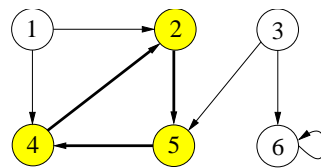


Figura 7.3: El camino  $\langle 2, 5, 4, 2 \rangle$  es un ciclo de longitud 3.

Un **bucle** es un ciclo de longitud 1.

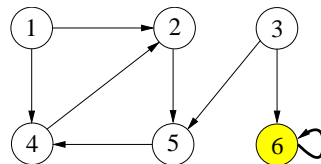


Figura 7.4: Ejemplo de bucle.

Un **grafo acíclico** es un grafo sin ciclos.

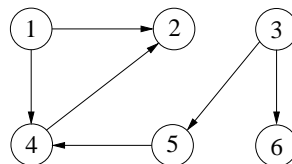


Figura 7.5: Ejemplo de grafo acíclico.

Se dice que un nodo  $v$  es **adyacente** al nodo  $u$  si existe una arista  $(u,v) \in E$  que va desde  $u$  hasta  $v$ . Por ejemplo, en el grafo que se muestra en la figura 7.5, los vértices 2 y 4 son adyacentes al vértice 1.

En un grafo no dirigido, se dice que un arco  $(u,v) \in E$  **incide** en los nodos  $u, v$ ; mientras que en un grafo dirigido un arco  $(u,v) \in E$  **incide** en el nodo  $v$ , y **parte** del nodo  $u$ . Por ejemplo, en el grafo dirigido de la figura 7.1, la arista  $(1,2)$  incide en el vértice 2 y parte del vértice 1, mientras que en el grafo no dirigido que se muestra en la misma figura, la arista  $\{1,2\}$  incide en ambos vértices: 1 y 2.

Se denomina **grado** de un nodo al número de arcos que inciden en él. En los grafos dirigidos se puede distinguir entre **grado de salida** de un nodo (número de arcos que parten de él), y **grado de entrada** (número de arcos que inciden en él). En este caso, el grado del vértice será la suma de los grados de entrada y de salida. El **grado de un grafo** es el máximo grado de sus vértices. Por ejemplo, el grafo dirigido de la figura 7.1 es de grado 3, y el vértice 2 como tiene grado de salida 1 y grado de entrada 2, es de grado 3.

Un grafo  $G' = (V', E')$  es un **subgrafo** de  $G = (V, E)$  si  $V' \subseteq V$  y  $E' \subseteq E$ . Un **subgrafo inducido** por  $V' \subseteq V$  es un subgrafo  $G' = (V', E')$  tal que  $E' = \{(u,v) \in E \mid u,v \in V'\}$ .

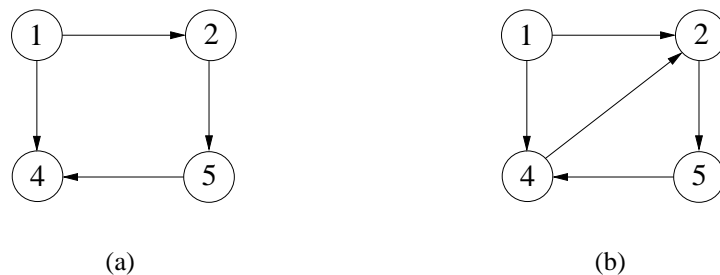


Figura 7.6: Ejemplo de subgrafos del grafo dirigido de la figura 7.1: en la figura (a) se muestra un subgrafo  $G' = (V', E')$ , tal que  $V' = \{1,2,4,5\}$  y  $E' = \{(1,2), (1,4), (2,5), (5,4)\}$ ; en la figura (b) se muestra el subgrafo  $G' = (V', E')$  inducido por  $V' = \{1,2,4,5\}$

Se dice que un vértice  $v$  es **alcanzable desde un vértice**  $u$ , si existe un camino de  $u$  a  $v$ .

Un grafo no dirigido es **conexo** si existe un camino desde cualquier vértice a cualquier otro. Un grafo dirigido con esta propiedad se denomina **fuertemente conexo**. Si un grafo dirigido no es fuertemente conexo, pero el grafo subyacente (sin sentido en los arcos) es conexo, se dice que el grafo es **débilmente conexo**. El grafo no dirigido de la figura 7.1 es conexo; mientras que el grafo dirigido de la

misma figura es débilmente conexo. A continuación se muestra un grafo dirigido fuertemente conexo:

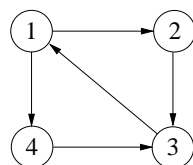


Figura 7.7: Ejemplo de grafo fuertemente conexo.

En un grafo no dirigido, las **componentes conexas** son las clases de equivalencia de vértices según la relación “ser alcanzable desde”. Por lo tanto, todos los vértices que formen parte de una componente conexa serán alcanzables entre sí.

Un grafo no dirigido es **no conexo** si está formado por varias componentes conexas. Por lo tanto, un grafo no dirigido es conexo, si tiene una única componente conexa. El grafo no dirigido que se muestra en la figura 7.1 tiene una única componente conexa ya que todos los vértices son alcanzables entre sí; por lo tanto es conexo.

En un grafo dirigido, las **componentes fuertemente conexas**, son las clases de equivalencia de vértices según la relación “ser mutuamente alcanzable”. Por lo tanto, para todo par de vértices  $u, v$  que pertenezcan a la misma componente fuertemente conexa, existirá un camino desde  $u$  hasta  $v$  y otro desde  $v$  hasta  $u$ .

Un grafo dirigido es **no fuertemente conexo** si está formado por varias componentes fuertemente conexas. Por lo tanto, un grafo dirigido es fuertemente conexo si tiene una única componente fuertemente conexa. El grafo que se muestra en la figura 7.7 tiene una única componente fuertemente conexa por lo que es fuertemente conexo. El grafo dirigido que se muestra en la figura 7.1 tiene cuatro componentes fuertemente conexas, como se representa en la figura 7.8, por lo que es un grafo dirigido no fuertemente conexo.

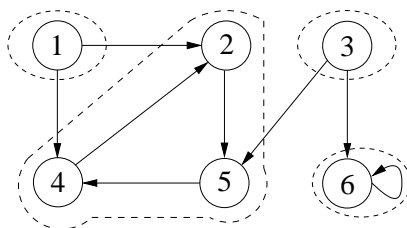


Figura 7.8: Componentes fuertemente conexas del grafo dirigido de la figura 7.1.

Un grafo **ponderado o etiquetado** es aquel grafo en el que cada arco, o cada vértice, o los dos, tienen asociada una etiqueta, que puede ser un nombre, un peso, etc.

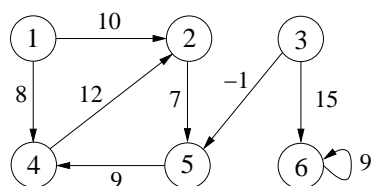


Figura 7.9: Ejemplo de grafo ponderado: cada arista tiene asociada un peso.

## 7.2. Introducción a la teoría de grafos

¿Por qué trabajar con grafos? Existen una multitud de problemas que pueden reducirse a una representación basada en grafos, a partir de esta representación el problema se hace abordable desde un punto de vista formal y es susceptible de ser resuelto mediante un algoritmo y por tanto tratable mediante un computador. Vamos a ver un problema clásico que motivó el nacimiento de la teoría de grafos: el problema de *los puentes de Königsberg*.

En siglos pasados, Königsberg fue una rica ciudad de la zona oriental de Prusia; hoy día su nombre es Kaliningrad y pertenece a Rusia. Se encuentra a orillas del mar Báltico y a unos 50 Km de la frontera con Polonia. Königsberg es cruzada por un río, el Pregel, que forma una isla en el centro del río. En el siglo XVIII, el río estaba atravesado por siete puentes (ya no, pues la ciudad fue parcialmente destruida durante la segunda Guerra Mundial), situados como en la figura 7.10, que pueden verse resaltados en color en la figura 7.11. y que permitían enlazar distintos barrios. Königsberg fue la ciudad natal de Kant, famoso filósofo alemán. La disposición topográfica de Königsberg dió lugar, precisamente en la época de Kant a un juego que concentró la atención de los matemáticos del momento. El juego consiste en lo siguiente: como es habitual en los pueblos alemanes, también en Königsberg sus habitantes solían pasear los domingos por las calles, pero ¿era posible planificar tal paseo de forma que saliendo de casa se pudiera regresar a ella, tras haber atravesado cada uno de los puentes una vez, pero sólo una?



Figura 7.10: Mapa de la ciudad de Königsberg en tiempos de Euler.

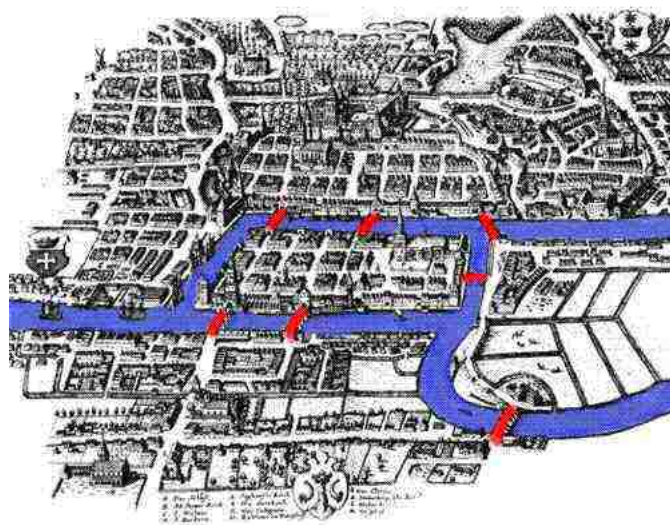


Figura 7.11: Mapa de la ciudad de Königsberg con el río y los puentes coloreados.

La resolución del problema puede obtenerse mediante *fuerza bruta*, probando todas las combinaciones posibles de paseos, pero fue Leonhard Euler (1707-1783), matemático suizo quien analizó el problema y le dio una solución formal. Tras prolongados estudios, Euler dio una respuesta segura: ¡no es posible planificar el recorrido para lograr atravesar cada uno de los puentes y una sola vez! Sus investigaciones sentaron las bases de una nueva rama de las matemáticas y de la



geometría que recibe el nombre de **teoría de grafos**. Desde entonces, esta teoría ha tenido aplicación práctica no sólo en las matemáticas, sino también en otras ramas. Ya en el siglo XIX los grafos se emplearon en la teoría de los circuitos eléctricos y en las teorías de los diagramas moleculares. Actualmente, además de constituir un instrumento de análisis en las matemáticas puras, la teoría de grafos se emplea para solucionar múltiples problemas de orden práctico: de transporte, por ejemplo, y, en general, en problemas de programación.

Para resolver el problema, Euler se comportó como un científico moderno: trató de traducir el problema a una fórmula más general y simplificada. Para ello trazó sobre un papel un esquema de Königsberg semejante al de la figura 7.12, que posteriormente tradujo a un esquema todavía más simplificado como el de la figura 7.13.

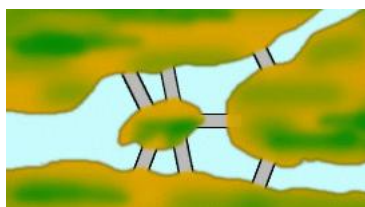


Figura 7.12: Mapa esquemático de la ciudad de Königsberg.

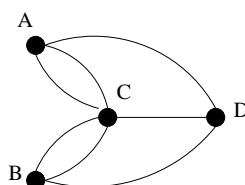


Figura 7.13: Grafo que representa la ciudad de Königsberg y sus puentes.

Representó las islas y las orillas del río con puntos y transformó los puentes en otras tantas líneas que enlazaban los diferentes puntos. Así el problema quedaba reducido a este otro: ¿es posible, partiendo de un punto cualquiera A, B, C, D, dibujar la figura volviendo siempre al mismo punto de partida, sin repasar ninguna línea ya trazada y sin levantar el lápiz del papel?

Para lograr una comprensión clara de la solución propuesta por Euler, tenemos que aclarar un nuevo concepto: el de la practicabilidad. Un grafo es practicable (en el sentido de un camino) cuando se pasa una sola vez por cada arista, mientras que por los vértices se puede pasar cuantas veces sea necesario. Corresponde a Euler el mérito de haber descubierto las siguientes reglas:

1. Si un grafo está compuesto de vértices solo de grado par, se puede entonces recorrer en una sola pasada, partiendo de un determinado vértice y regresando al mismo.
2. Si un grafo contiene sólo dos vértices de grado impar, también puede recorrerse en una sola pasada, pero sin volver al punto de partida.
3. Si un grafo contiene un número de vértices de grado impar superior a 2, entonces el problema no tiene solución: no se puede recorrer en una sola pasada.

Volvemos ahora sobre nuestro problema de los puentes de Königsberg y analizamos cual es el grado de cada vértice:

VÉRTICE	GRADO
A	3
B	3
C	5
D	3

Hay 4 vértices de grado impar y ninguno de grado par, por consiguiente, ¡el problema no tiene solución!

## 7.3. Representación de grafos

Típicamente un grafo  $G = (V, E)$  suele representarse utilizando dos posibles representaciones: representación mediante *listas de adyacencia* o representación mediante una *matriz de adyacencia*.

En ambas representaciones, para poder referenciar los vértices de un grafo, a cada uno de ellos, se le asigna un número que lo identifica.

### 7.3.1. Listas de adyacencia

Esta es la representación más escogida a la hora de representar un grafo.

Un grafo  $G = (V, E)$  se representa mediante un vector de tamaño igual al número de vértices que contiene el grafo  $|V|$ , de tal forma, que cada posición  $i$  del vector, contiene un puntero a una lista enlazada de elementos, denominada *lista de adyacencia*; cada elemento de la lista de adyacencia representa a cada uno de los vértices adyacentes al vértice  $i$ ; es decir, todos los vértices  $v$  tales que existe un arco  $(i, v) \in E$ .

Los vértices en cada lista de adyacencia son, normalmente, almacenados en un orden arbitrario.

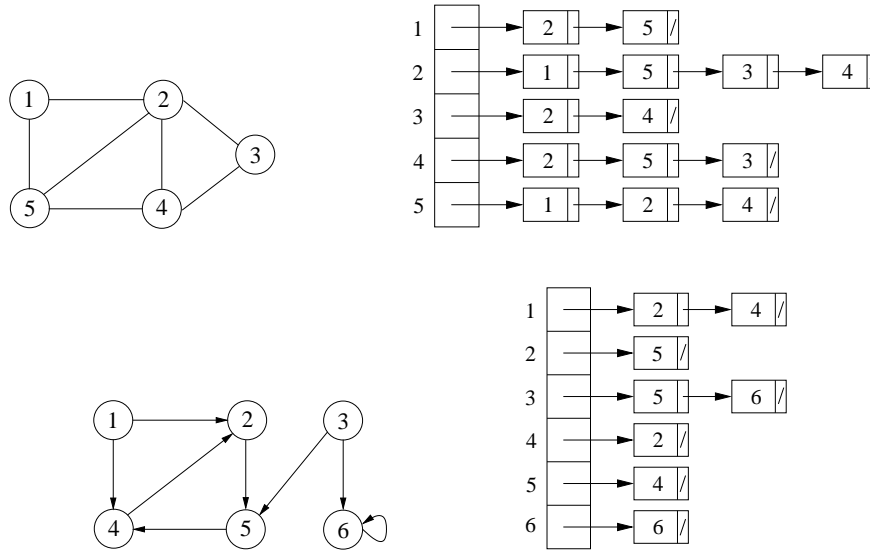


Figura 7.14: Ejemplo de representación de un grafo no dirigido, y de otro dirigido, mediante listas de adyacencia.

Si  $G = (V, E)$  es un grafo dirigido, la suma de las longitudes de todas las listas de adyacencia será  $|E|$ ; ya que cada arista del grafo  $(u, v) \in E$  implica tener un elemento en una lista de adyacencia ( $v$  en la lista de adyacencia de  $u$ ). Si  $G$  es un grafo no dirigido, la suma de las longitudes de todas las listas de adyacencia será  $2|E|$ ; ya que cada arista del grafo  $(u, v) \in E$  implica tener dos elementos en dos listas de adyacencia ( $u$  en la lista de adyacencia de  $v$ , y  $v$  en la lista de adyacencia de  $u$ ).

Por lo tanto, el tamaño máximo de memoria requerido (coste espacial) para representar un grafo, sea dirigido o no lo sea, es  $O(|V| + |E|)$ ; donde  $|V|$  es el número de vértices del grafo (tamaño del vector), y  $|E|$  es el número de aristas del grafo.

Esta representación será apropiada para grafos en los que  $|E|$  sea considerablemente menor que  $|V|^2$ .

Esta representación es fácilmente extensible a su utilización con grafos ponderados, es decir, grafos en los que cada arista tiene asociada un peso. El peso  $w$  de una arista  $(u, v) \in E$  se puede almacenar fácilmente en el elemento que representa a  $v$  en la lista de adyacencia de  $u$ .

Una potencial desventaja de la representación mediante listas de adyacencia es, que si se quiere comprobar si una arista  $(u, v) \in E$  es necesario buscar el vértice  $v$  en la lista de adyacencia de  $u$ . El coste de esta operación será  $O(\text{Grado}(G)) \subseteq O(|V|)$ .

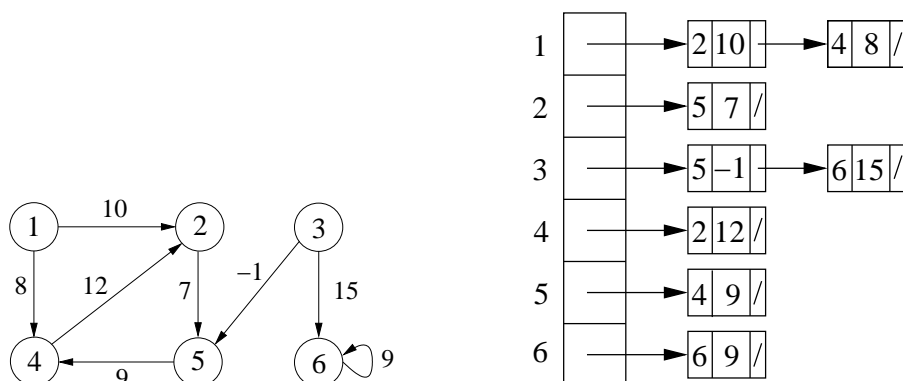


Figura 7.15: Ejemplo de representación de un grafo ponderado mediante listas de adyacencia.

Un posible definición de tipos en C para esta representación (considerando que el grafo puede ser ponderado) sería:

```
#define MAXVERT ...

typedef struct vertice {
    int nodo, peso;
    struct vertice *sig;
} vert_ady;

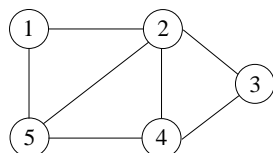
typedef struct {
    int talla;
    vert_ady *ady[MAXVERT];
} grafo;
```

### 7.3.2. Matriz de adyacencia

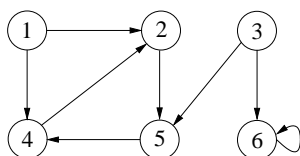
Una posible solución para evitar el inconveniente que presentan las listas de adyacencia es utilizar una matriz de adyacencia para la representación del grafo, asumiendo, eso sí, la necesidad de disponer de más memoria para este tipo de representación.

La representación de un grafo  $G = (V, E)$  mediante una matriz de adyacencia, consiste en tener una matriz  $A$  de dimensiones  $|V| \times |V|$ , tal que, para cada valor  $a_{ij}$  de la matriz:

$$a_{ij} = \begin{cases} 1 & \text{si } (i, j) \in E \\ 0 & \text{en cualquier otro caso} \end{cases}$$



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



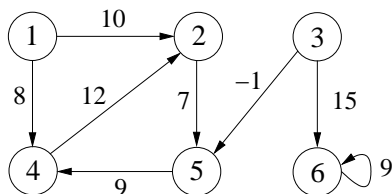
	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Figura 7.16: Ejemplo de representación de un grafo no dirigido, y de otro dirigido, mediante una matriz de adyacencia.

Independientemente del número de arcos del grafo, el coste espacial es  $O(|V|^2)$ .

En el caso de que el grafo fuese ponderado, para cada arco  $(i, j) \in E$ , su peso se almacena en la posición  $(i, j)$  de la matriz (ver figura 7.17):

$$a_{ij} = \begin{cases} w(i, j) & \text{si } (i, j) \in E \\ 0 \text{ o } \infty & \text{en cualquier otro caso} \end{cases}$$



	1	2	3	4	5	6
1	0	10	0	8	0	0
2	0	0	0	0	7	0
3	0	0	0	0	-1	15
4	0	12	0	0	0	0
5	0	0	0	9	0	0
6	0	0	0	0	0	9

Figura 7.17: Ejemplo de representación de un grafo ponderado mediante una matriz de adyacencia.

Este tipo de representación es útil cuando los grafos tienen un número de vértices razonablemente pequeño, o cuando se trata de grafos densos, es decir, donde el número de aristas  $|E|$  es cercano a las dimensiones de la matriz  $|V| \times |V|$ .

En este tipo de representación se puede comprobar si una arista  $(u,v) \in E$  con un coste temporal  $O(1)$ , con tan solo consultar la posición  $A[u][v]$  de la matriz.

Un posible definición de tipos en C para esta representación sería:

```
#define MAXVERT ...

typedef struct {
    int talla;
    int A[MAXVERT][MAXVERT];
} grafo;
```

## 7.4. Recorrido de grafos

Para resolver con eficiencia muchos problemas relacionados con grafos, es necesario recorrer los vértices y los arcos del grafo de manera sistemática. Existen dos formas típicas de recorrer un grafo: *recorrido primero en profundidad* y *recorrido primero en anchura*.

### 7.4.1. Recorrido primero en profundidad

El recorrido primero en profundidad de un grafo, es una generalización del recorrido en orden previo de un árbol. Iniciando en algún vértice  $v$  el recorrido, se procesa  $v$  y, a continuación, recursivamente, se recorren en profundidad todos los vértices adyacentes a él que queden por recorrer. Esta técnica se conoce como recorrido primero en profundidad porque recorre el grafo en la dirección hacia adelante (más profunda) mientras sea posible.

La estrategia consiste en ir recorriendo el grafo, partiendo de un vértice determinado  $v$ , de forma que cuando se visita un nuevo vértice, se exploran cada uno de los caminos que parten de ese vértice, de tal manera que hasta que no se ha finalizado de explorar uno de los caminos no comienza a explorarse el siguiente. Es importante resaltar que un camino deja de explorarse en el momento que lleva a un vértice que ya ha sido visitado en el recorrido con anterioridad. Una vez que se han visitado todos los vértices alcanzables desde  $v$ , el recorrido del grafo puede quedar incompleto si existían vértices en el grafo que no eran alcanzables desde  $v$ ; en este caso, se selecciona alguno de ellos como nuevo vértice de partida, y se repite el mismo proceso hasta que todos los vértices del grafo han sido visitados.

Dado un grafo  $G = (V, E)$ , el funcionamiento del algoritmo se ajusta al siguiente esquema recursivo:

1. Inicialmente se marcan todos los vértices del grafo  $G$  como *no visitados*;
2. Se escoge un vértice  $u \in V$  como punto de partida;
3.  $u$  se marca como visitado;
4. para cada vértice  $v$  adyacente a  $u$ ,  $(u, v) \in E$ , si  $v$  no ha sido visitado, se repiten recursivamente los pasos (3) y (4) para el vértice  $v$ . Este proceso finaliza cuando se visitan todos los nodos alcanzables desde el vértice escogido como de partida en el paso 2. Debido a que el recorrido del grafo puede quedar incompleto si desde el vértice de partida no fueran alcanzables todos los nodos del grafo, se vuelve al paso (2) escogiendo un nuevo vértice  $v$ , que no haya sido visitado, como de partida, y se repite el mismo proceso hasta que se han recorrido todos los vértices del grafo.

A continuación se presenta el algoritmo (en *pseudo-código*) que, dado un grafo  $G = (V, E)$ , realiza un recorrido del grafo  $G$  siguiendo la estrategia *primero en profundidad*. El algoritmo utiliza un vector denominado *color* de talla  $|V|$  para indicar si un vértice  $u \in V$  ha sido visitado ( $\text{color}[u]=\text{AMARILLO}$ ) o no ( $\text{color}[u]=\text{BLANCO}$ ).

```

Algoritmo Recorrido_en_profundidad( $G$ ){
    para cada vértice  $u \in V$ 
         $\text{color}[u] = \text{BLANCO}$ 
    fin_para
    para cada vértice  $u \in V$ 
        si ( $\text{color}[u] = \text{BLANCO}$ ) Visita_nodo( $u$ )
    fin_para
}

Algoritmo Visita_nodo( $u$ ){
     $\text{color}[u] = \text{AMARILLO}$ 
    para cada vértice  $v \in V$  adyacente a  $u$ 
        si ( $\text{color}[v] = \text{BLANCO}$ ) Visita_nodo( $v$ )
    fin_para
}

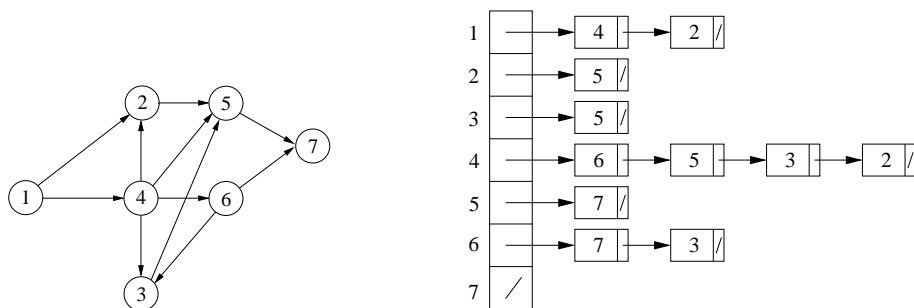
```

El algoritmo *Recorrido\_en\_profundidad* comienza marcando todos los vértices como *no visitado* ( $\forall u \in V \text{ color}[u]=\text{BLANCO}$ ). A continuación, para cada vértice  $u$  que quede sin visitar ( $\text{color}[u]=\text{BLANCO}$ ), se comienza el recorrido llamando a la función *Visita\_nodo*( $u$ ). Inicialmente, como todos los vértices están sin visitar, se iniciará el recorrido arbitrariamente por uno de ellos. Hay que tener en cuenta que cuando se finalice el recorrido iniciado desde un vértice  $u$ , se habrán visitado todos los nodos que eran alcanzables desde  $u$ , por lo que la siguiente llamada a *Visita\_nodo* desde *Recorrido\_en\_profundidad* se realizará sobre un vértice que no era alcanzable desde  $u$  (si existiera alguno en el grafo).

Cada llamada a la función *Visita\_nodo* se realiza sobre un vértice *no visitado*. Debido a que lo primero que realiza la función es marcar el vértice  $u$ , sobre el que se aplica, como visitado ( $\text{color}[u]=\text{AMARILLO}$ ), nunca más se volverá a aplicar la función *Visita\_nodo* sobre el nodo  $u$ . A continuación, se recorren en profundidad todos los vértices adyacentes a  $u$  que no han sido visitados llamando recursivamente a la función *Visita\_nodo*.

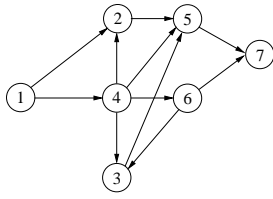
### Ejemplo de funcionamiento del algoritmo

En la figura 7.18 se muestra un ejemplo del funcionamiento del algoritmo para el siguiente grafo:

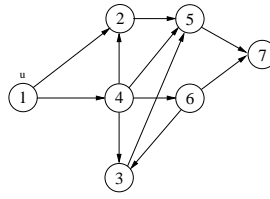


Obsérvese en el ejemplo que el recorrido a través del grafo depende del orden en que aparecen en las listas de adyacencia los vértices. Es decir, cuando se aplica la función *Visita\_nodo* sobre un vértice  $u$ , la misma función se aplica recursivamente sobre los vértices adyacentes a  $u$  no visitados siguiendo el orden en que aparecen en la lista de adyacencia de  $u$ .

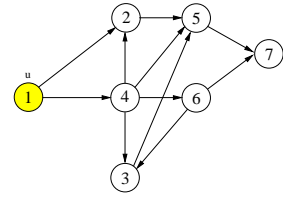




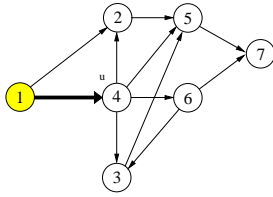
(a) Recorrido\_en\_profundidad(G)



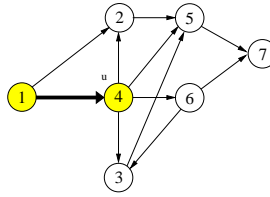
(b) Visita\_nodo(1)



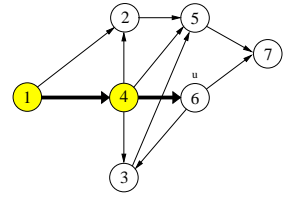
(c) color[1]=AMARILLO



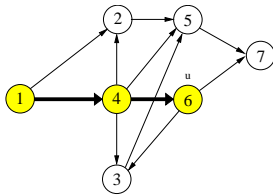
(d) Visita\_nodo(4)



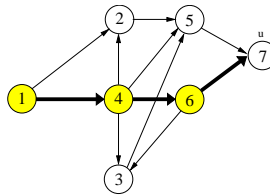
(e) color[4]=AMARILLO



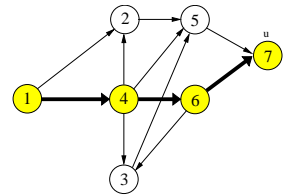
(f) Visita\_nodo(6)



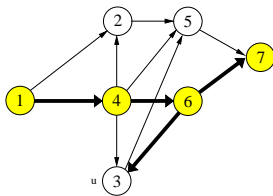
(g) color[6]=AMARILLO



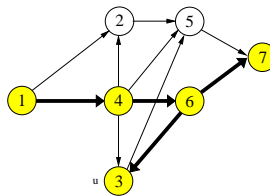
(h) Visita\_nodo(7)



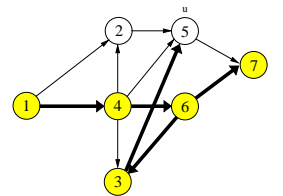
(i) color[7]=AMARILLO



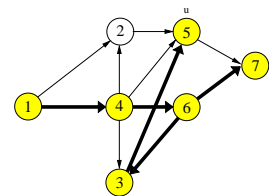
(j) Visita\_nodo(3)



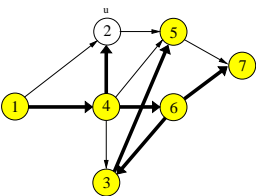
(k) color[3]=AMARILLO



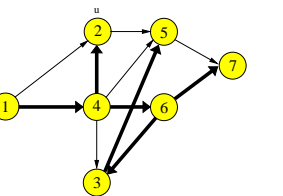
(l) Visita\_nodo(5)



(m) color[5]=AMARILLO



(n) Visita\_nodo(2)



(ñ) color[2]=AMARILLO

Figura 7.18: Ejemplo de cómo se recorre un grafo siguiendo la estrategia primero en profundidad.

### Coste temporal del algoritmo

Para evaluar el coste temporal del algoritmo consideraremos que el grafo  $G = (V, E)$  se representa mediante listas de adyacencia.

Obsérvese que el algoritmo *Visita\_nodo* se aplica exáctamente una vez sobre cada vértice del grafo. Esto es debido a que el algoritmo *Visita\_nodo* se aplica únicamente sobre vértices  $u \in V$  no visitados ( $\text{color}[u]=\text{BLANCO}$ ), y lo primero que hace el propio algoritmo es marcar como *visitado* al vértice sobre el que se aplica

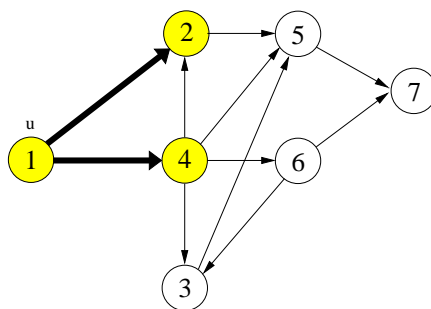
( $\text{color}[u]=\text{AMARILLO}$ ), por lo que nunca más volverá a realizarse una llamada al algoritmo *Visita\_nodo* sobre  $u$ . Debido a que el coste temporal del algoritmo *Visita\_nodo* depende del número de vértices adyacentes que tiene el nodo  $u$  sobre el que se aplica (bucle *para*), es decir, de la longitud de la lista de adyacencia del vértice  $u$ ; el coste temporal de realizar todas las llamadas al algoritmo *Visita\_nodo* será:

$$\sum_{v \in V} |\text{ady}(v)| = \Theta(|E|)$$

Si, además, añadimos el coste temporal asociado a los bucles que se realizan en *Recorrido\_en\_profundidad*:  $O(|V|)$ , podemos concluir que el coste temporal del algoritmo de recorrido en profundidad es  $O(|V| + |E|)$ .

#### 7.4.2. Recorrido primero en anchura

El recorrido primero en anchura de un grafo, es una generalización del recorrido por niveles de un árbol. Iniciando en algún vértice  $u$  el recorrido, se visita  $u$  y, a continuación, se visitan cada uno de los vértices adyacentes a  $u$ . El proceso se repite para cada uno de los nodos adyacentes a  $u$ , siguiendo el orden en que fueron visitados. El coste temporal del algoritmo es el mismo que para el recorrido en profundidad:  $O(|V| + |E|)$ .



### 7.4.3. Ordenación topológica

Una aplicación inmediata del recorrido en profundidad es su utilización para obtener una ordenación topológica de los vértices de un grafo dirigido y acíclico.

Sea  $G = (V, E)$  un grafo dirigido y acíclico; la **ordenación topológica** es una permutación  $v_1, v_2, v_3, \dots, v_{|V|}$  de los vértices del grafo, tal que si  $(v_i, v_j) \in E$ ,  $v_i \neq v_j$ , entonces  $v_i$  aparece antes que  $v_j$  en la permutación.

La ordenación no es posible si el grafo es cíclico. Además, la permutación válida, obtenida como resultado de la ordenación topológica, no es única; es decir, podrían existir distintas permutaciones de los vértices que dieran como resultado una ordenación topológica de los mismos.

Una ordenación topológica de un grafo puede ser vista como una ordenación de los vértices a lo largo de una línea horizontal, de forma que todos los arcos van de izquierda a derecha. Los grafos dirigidos y acíclicos se utilizan en numerosas aplicaciones en las que se necesita representar el orden de ejecución de diferentes tareas relacionadas entre sí.

A continuación se presenta el algoritmo que, dado un grafo  $G = (V, E)$  dirigido y acíclico, obtiene una ordenación topológica de sus vértices. El algoritmo es prácticamente el mismo que el utilizado en el recorrido primero en profundidad, únicamente varía en la utilización de una pila  $P$  en la que se va almacenando el orden topológico de los vértices del grafo.

```
Algoritmo Ordenación_topológica( $G$ ) {  
    para cada vértice  $u \in V$   
        color[ $u$ ] = BLANCO  
    fin_para  
     $P = \emptyset$   
    para cada vértice  $u \in V$   
        si (color[ $u$ ] = BLANCO) Visita_nodo( $u$ )  
    fin_para  
    devolver( $P$ )  
}
```

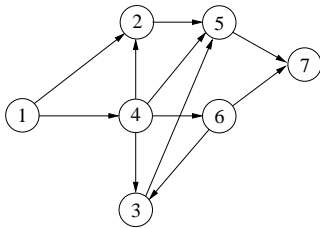
```
Algoritmo Visita_nodo( $u$ ) {  
    color[ $u$ ] = AMARILLO  
    para cada vértice  $v \in V$  adyacente a  $u$   
        si (color[ $v$ ] = BLANCO) Visita_nodo( $v$ )  
    fin_para  
    apilar( $P, u$ )  
}
```

El algoritmo explota el hecho de que, en el recorrido primero en profundidad, cuando finaliza el recorrido partiendo desde un nodo  $u$  ( $\text{Visita\_nodo}(u)$ ), se habrán visitado todos aquellos vértices que son alcanzables desde  $u$ ; es decir, todos aquellos vértices que deben suceder a  $u$  en el orden topológico. De esta forma, si justo antes de finalizar la función  $\text{Visita\_nodo}$  apilamos el vértice sobre el que se acaba de aplicar la función, el vértice en cuestión se apilará justamente después de haber apilado todos aquellos vértices que eran alcanzables desde él. Por lo tanto, cuando el algoritmo  $\text{Ordenación\_topológica}$  finalice, en la pila  $P$  estarán los vértices del grafo ordenados topológicamente.

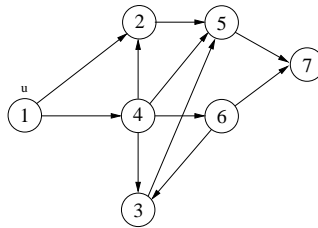
Debido a que el tiempo de inserción de cada vértice en la pila tiene un coste constante  $O(1)$ , obtener el orden topológico de los vértices de un grafo dirigido y acíclico tiene un **coste temporal** equivalente a recorrer el grafo siguiendo la estrategia primero en profundidad:  $O(|V| + |E|)$ .

### Ejemplo de funcionamiento del algoritmo

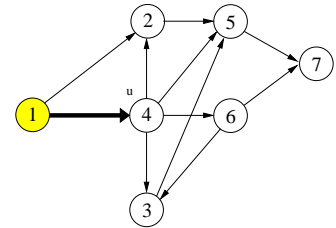
A continuación se muestra cómo el algoritmo obtiene el orden topológico de los vértices de un grafo dirigido y acíclico. Debajo de cada figura se muestra qué acción se lleva a cabo, dada la situación que se muestra en la figura. Además, en todo momento, se muestra el contenido de la pila  $P$  y en qué instante se insertan cada uno de los vértices en la pila.



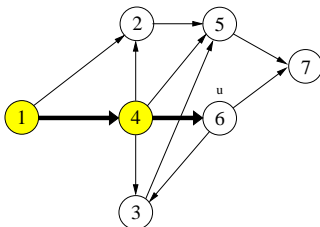
(a)  $\text{Orden\_topologico}(G)$



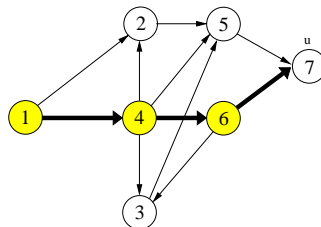
(b)  $\text{Visita\_nodo}(1)$   
 $P = \{\}$



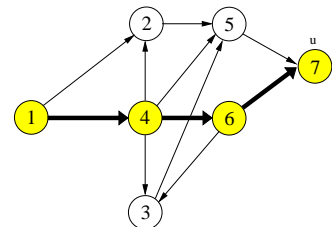
(c)  $\text{Visita\_nodo}(4)$   
 $P = \{\}$



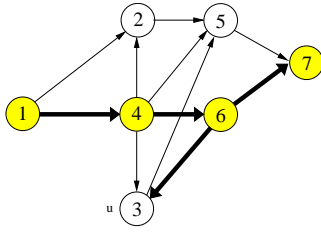
(d)  $\text{Visita\_nodo}(6)$   
 $P = \{\}$



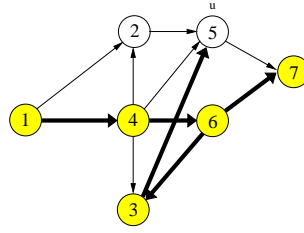
(e)  $\text{Visita\_nodo}(7)$   
 $P = \{\}$



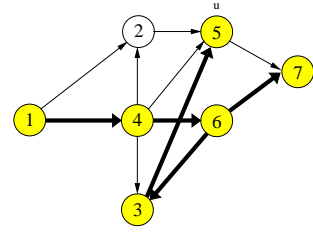
(f)  $\text{apilar}(P, 7)$   
 $P = \{7\}$



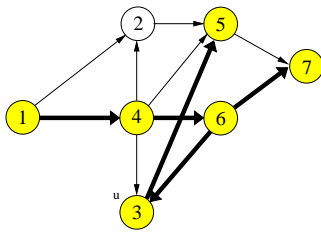
(g) Visita\_nodo(3)  
 $P = \{7\}$



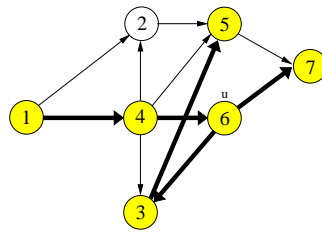
(h) Visita\_nodo(5)  
 $P = \{7\}$



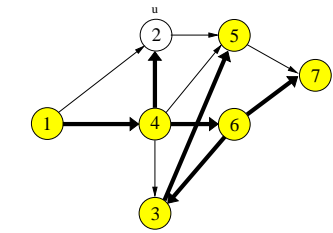
(i) apilar(P, 5)  
 $P = \{5, 7\}$



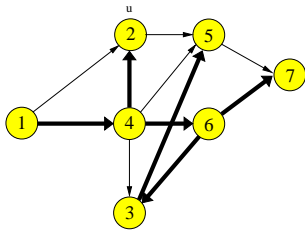
(j) apilar(P, 3)  
 $P = \{3, 5, 7\}$



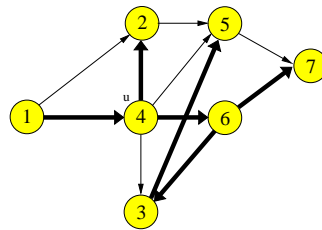
(k) apilar(P, 6)  
 $P = \{6, 3, 5, 7\}$



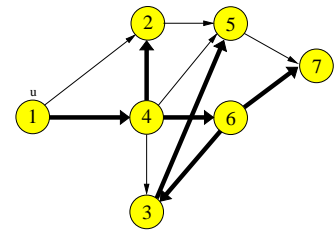
(l) Visita\_nodo(2)  
 $P = \{6, 3, 5, 7\}$



(m) apilar(P, 2)  
 $P = \{2, 6, 3, 5, 7\}$

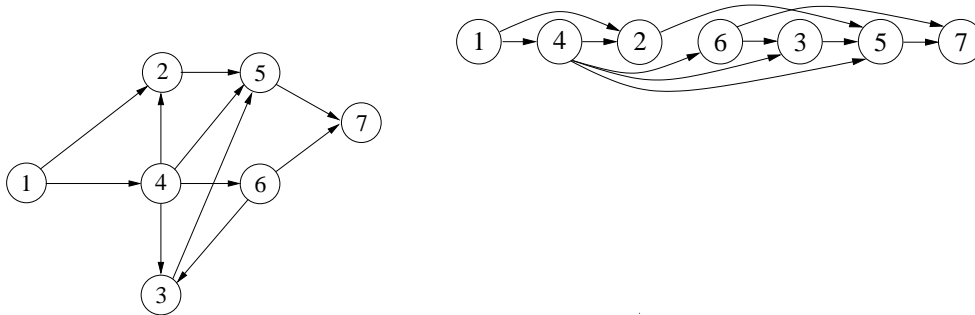


(n) apilar(P, 4)  
 $P = \{4, 2, 6, 3, 5, 7\}$



(ñ) apilar(P, 1)  
 $P = \{1, 4, 2, 6, 3, 5, 7\}$

Tal como se indicó anteriormente, una ordenación topológica de un grafo puede ser vista como una ordenación de los vértices a lo largo de una línea horizontal, de forma que todos los arcos van de izquierda a derecha. En la siguiente figura se muestra cómo puede representarse el grafo, utilizado en el ejemplo, siguiendo el orden topológico de sus vertices.



## 7.5. Caminos de mínimo peso: algoritmo de Dijkstra

### 7.5.1. Caminos de mínimo peso

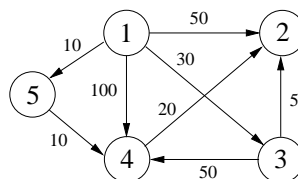
Un problema común que se presenta en numerosas aplicaciones es la búsqueda de caminos de mínimo peso en grafos dirigidos y ponderados. Veamos qué es un camino de mínimo peso:

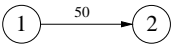
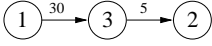
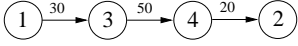
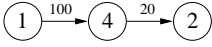
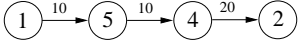
Dado un grafo  $G = (V, E)$  dirigido y ponderado, donde cada arista  $(u, v)$  tiene asociada un peso  $w(u, v)$ , se define el **peso de un camino**  $p = \langle v_0, v_1, \dots, v_k \rangle$  como la suma de los pesos de las aristas que lo forman:

$$w(p) = \sum_{i=1}^k w(v_{i-1}, v_i)$$

Ahora podemos definir el **camino de mínimo peso** desde un vértice  $u$  a  $v$ , como el camino que tenga un peso menor entre todos los caminos de  $u$  a  $v$ , o  $\infty$  si no existe camino de  $u$  a  $v$ . En lo sucesivo, para referirnos al peso de un camino de  $u$  a  $v$  hablaremos de la *longitud* de un camino de  $u$  a  $v$ , y para referirnos a un camino de mínimo peso de  $u$  a  $v$  diremos *camino más corto* de  $u$  a  $v$ .

Dado el grafo dirigido y ponderado que se muestra en la figura, se indican cuáles son las longitudes de todos los caminos desde el vértice 1 al vértice 2. Como puede observarse el camino  $\langle 1, 3, 2 \rangle$  es el camino de menor longitud.



Camino	Longitud (peso o coste)
	50
	35
	100
	120
	40

Por ejemplo, supóngase que utilizamos un grafo dirigido y ponderado para representar las comunicaciones entre aeropuertos: cada vértice representa una ciudad, y cada arista  $(u,v)$  una ruta aérea de la ciudad  $u$  a la ciudad  $v$ ; el peso asociado a la arista podría ser el tiempo que se requiere para volar de  $u$  a  $v$ . La obtención del camino más corto entre un par de ciudades nos indicaría cuál es la ruta más rápida entre ellas.

Dentro del cálculo de caminos de menor longitud en grafos dirigidos y ponderados existen distintas variantes:

- Obtención de los caminos más cortos desde un vértice origen a todos los demás.
- Obtención de los caminos más cortos desde todos los vértices a uno destino.
- Obtención del camino más corto de un vértice  $u$  a un vértice  $v$ .
- Obtención de los caminos más cortos entre todos los pares de vértices.

El algoritmo de Dijkstra, que veremos a continuación, obtiene los caminos más cortos desde un vértice origen a todos los demás. El resto de variantes indicadas pueden resolverse aplicando igualmente el algoritmo de Dijkstra, si bien, la última de las variantes puede resolverse de manera más eficiente [Cormen].

### 7.5.2. Algoritmo de Dijkstra

**Problema:** Sea  $G = (V, E)$  un grafo dirigido y ponderado con pesos no negativos en las aristas; dado un vértice origen  $s \in V$ , se desea obtener cada uno de los caminos más cortos de  $s$  al resto de vértices  $v \in V$ .

El algoritmo de Dijkstra resuelve eficientemente este problema, si bien, hay que hacer hincapié en que si existen aristas ponderadas con pesos negativos en el grafo, la solución obtenida podría ser errónea. Otros algoritmos, como el de Bellman-Ford [Cormen], permiten pesos negativos en las aristas, siempre y cuando, no existan ciclos de peso negativo alcanzables desde el vértice origen  $s \in V$ .

Los algoritmos de búsqueda de caminos más cortos explotan la propiedad de que el camino más corto entre dos vértices contiene, a su vez, caminos más cortos entre los vértices que forman el camino.

El algoritmo de Dijkstra mantiene los siguientes conjuntos:

- Un conjunto de vértices  $S$  que contiene los vértices para los que la distancia más corta desde el origen ya es conocida. Inicialmente  $S = \emptyset$ .
- Un conjunto de vértices  $Q = V - S$  en el que se mantiene, para cada vértice, la distancia más corta desde el origen pasando a través de vértices que pertenecen a  $S$ ; es decir, para cada vértice  $u \in Q$  se mantiene la distancia más corta desde el origen utilizando uno de los caminos más cortos ya calculados. A esta distancia la denominaremos *distancia provisional*. Para mantener las distancias provisionales utilizaremos un vector  $D$  de talla  $|V|$ , en el cual, para cada posición  $i$ ,  $D[i]$  indicará la distancia provisional desde el vértice origen  $s$  al vértice  $i$ . Inicialmente,  $D[u] = \infty \forall u \in V - \{s\}$  y  $D[s] = 0$ .

La estrategia que sigue el algoritmo es la siguiente:

1. Se extrae del conjunto  $Q$  el vértice  $u$  cuya distancia provisional  $D[u]$  es la menor. Se puede afirmar que esta distancia es la menor posible entre el vértice origen  $s$  y  $u$ . La razón de esta aseveración radica en que, debido a que los pesos de las aristas son no negativos, no será posible encontrar otro camino más corto desde  $s$  hasta  $u$  pasando a través de algún otro vértice perteneciente a  $Q$ , ya que sus distancias provisionales eran mayores o iguales que la distancia provisional a  $u$ . Al mismo tiempo, la distancia provisional se correspondía con el camino más corto posible pasando a través de algún camino más corto ya calculado; o lo que es lo mismo, con el camino más corto posible utilizando vértices de  $S$ . Por este motivo, ya no es posible encontrar un camino más corto desde  $s$  hasta  $u$  utilizando algún otro vértice del grafo, ya que  $V = S \cup Q$ , por lo que podemos concluir que hemos hallado el camino más corto de  $s$  a  $u$ .
2. A continuación, se inserta el nuevo vértice  $u$ , para el que se ha calculado el camino más corto desde  $s$ , en el conjunto  $S$  ( $S = S \cup \{u\}$ ). Al añadir  $u$  al conjunto  $S$  será posible acceder a los vértices  $v$  adyacentes a  $u$  a través



del nuevo camino más corto calculado desde  $s$  hasta  $u$ . Por lo tanto, podría ocurrir que las distancias provisionales de los vértices  $v \in Q$  adyacentes a  $u$  fueran mejoradas utilizando el nuevo camino, por lo que, si esto ocurre, se actualiza la distancia provisional de estos vértices al nuevo valor calculado.

3. Los pasos 1 y 2 se repiten hasta que el conjunto  $Q$  queda vacío, momento en el cual se habrán estimado los caminos más cortos desde el vértice origen  $s$  al resto de vértices del grafo. En el vector  $D$  se tendrá, para cada vértice, la distancia más corta desde el origen.

A continuación se presenta el algoritmo de Dijkstra (en *pseudo-código*). El algoritmo utiliza un vector  $P$  de talla  $|V|$  que permitirá recuperar la secuencia de vértices que forman cada uno de los caminos mínimos calculados; en cada posición  $i$  del vector  $P$ , se almacena el índice del vértice que precede al vértice  $i$  en el camino más corto desde el origen  $s$  hasta  $i$ .

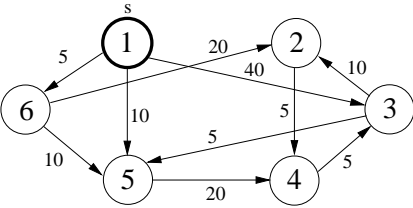
Dado un grafo  $G = (V, E)$  dirigido y ponderado, una función de ponderación  $w$  y un vértice origen  $s$ :

```

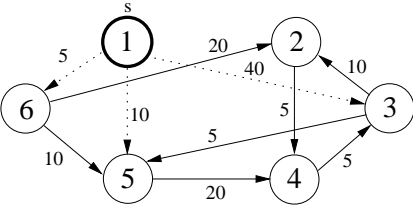
Algoritmo Dijkstra( $G, w, s$ ){
  para cada vértice  $v \in V$  hacer
     $D[v] = \infty$ 
     $P[v] = NULO$ 
  fin_para
   $D[s] = 0$ 
   $S = \emptyset$ 
   $Q = V$ 
  mientras  $Q \neq \emptyset$  hacer
     $u = \text{extract\_min}(Q)$  /* según  $D$  */
     $S = S \cup \{u\}$ 
    para cada vértice  $v \in V$  adyacente a  $u$  hacer
      si  $D[v] > D[u] + w(u,v)$  entonces
         $D[v] = D[u] + w(u,v)$ 
         $P[v] = u$ 
      fin_si
    fin_para
  fin_mientras
}
```

**Ejemplo del funcionamiento del algoritmo**

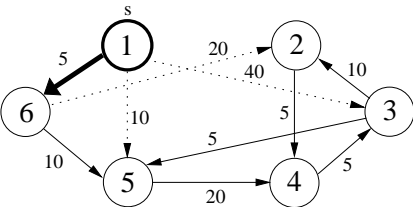
A continuación se muestra un ejemplo de cómo el algoritmo de Dijkstra obtiene los caminos más cortos desde el vértice origen 1 a todos los demás, en el siguiente grafo. Las aristas con trazo discontinuo indican caminos provisionales desde el origen a vértices; las aristas con trazo grueso indican caminos mínimos ya calculados; el resto de aristas se dibujan con trazo fino. Debajo de cada figura se muestra el estado de las estructuras que mantiene el algoritmo dada la situación mostrada en la figura.



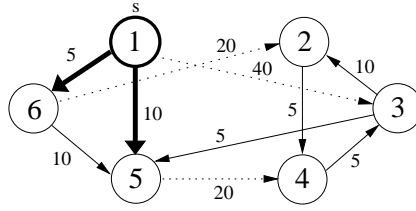
<i>S</i>	<i>Q</i>	<i>u</i>	1	2	3	4	5	6
{}	{1,2,3,4,5,6}	—	<i>D</i>	0	∞	∞	∞	∞
			<i>P</i>	NULO	NULO	NULO	NULO	NULO



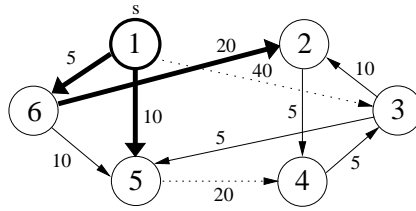
<i>S</i>	<i>Q</i>	<i>u</i>	1	2	3	4	5	6
{1}	{2,3,4,5,6}	1	<i>D</i>	0	∞	40	∞	10
			<i>P</i>	NULO	NULO	1	NULO	1



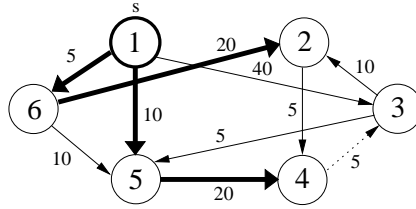
<i>S</i>	<i>Q</i>	<i>u</i>	1	2	3	4	5	6
{1,6}	{2,3,4,5}	6	<i>D</i>	0	25	40	∞	10
			<i>P</i>	NULO	6	1	NULO	1



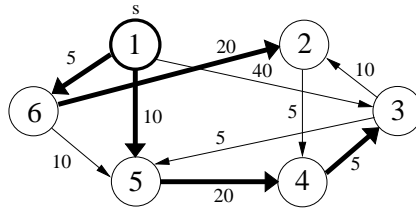
$S$	$Q$	$u$	1	2	3	4	5	6	
$\{1,6,5\}$	$\{2,3,4\}$	5	$D$	0	25	40	30	10	5
			$P$	$NULO$	6	1	5	1	1



$S$	$Q$	$u$	1	2	3	4	5	6	
$\{1,6,5,2\}$	$\{3,4\}$	2	$D$	0	25	40	30	10	5
			$P$	$NULO$	6	1	5	1	1



$S$	$Q$	$u$	1	2	3	4	5	6	
$\{1,6,5,2,4\}$	$\{3\}$	4	$D$	0	25	35	30	10	5
			$P$	$NULO$	6	4	5	1	1



$S$	$Q$	$u$	1	2	3	4	5	6	
$\{1,6,5,2,4,3\}$	$\{\}$	3	$D$	0	25	35	30	10	5
			$P$	$NULO$	6	4	5	1	1

## Coste temporal del algoritmo

Para evaluar el coste temporal del algoritmo asumiremos que la representación del grafo se realiza mediante listas de adyacencia.

El coste temporal del algoritmo viene determinado por el coste que supone realizar todas las iteraciones del bucle *mientras*. Debido a que el bucle *mientras* se repite hasta que el conjunto  $Q$  queda vacío, si tenemos en cuenta que inicialmente el conjunto  $Q$  contiene todos los vértices del grafo, y que, en cada iteración del bucle se extrae un elemento de  $Q$ , podemos concluir que el bucle *mientras* se realiza  $|V|$  veces. Así pues, la operación *extract\_min* se llevará a cabo  $|V|$  veces. Por otro lado, el bucle *para*, que contiene el bucle *mientras*, se realizará igualmente  $|V|$  veces (una vez para cada vértice del grafo). El bucle *para* se repite tantas veces como vértices adyacentes tenga el vértice extraído de  $Q$ , por lo que el número total de iteraciones de este bucle coincidirá con el número de vértices adyacentes que tengan los vértices del grafo, o lo que es lo mismo, con el número de arcos del grafo:  $|E|$ .

Para obtener finalmente el coste temporal del algoritmo, nos falta estimar el coste temporal asociado a la operación *extract\_min*. Este coste dependerá de la forma en que se obtenga el vértice de  $Q$  cuya distancia desde el origen sea la mínima respecto a todos los vértices de  $Q$ . Como una primera aproximación consideraremos que los elementos del conjunto  $Q$  se organizan en un vector sin ningún tipo de ordenación, por lo que la operación *extract\_min* tendrá un coste  $O(|V|)$ . En este caso, y debido a que tal como dijimos anteriormente, la operación *extract\_min* se realiza  $|V|$  veces, el coste temporal total de *extract\_min* es  $O(|V|^2)$ . Si a este coste le sumamos el coste que supone realizar el bucle *para*, podemos concluir que el coste temporal del algoritmo de Dijkstra, para esta primera aproximación, es  $O(|V|^2 + |E|) = O(|V|^2)$ .

La operación *extract\_min* podría realizarse de forma más eficiente, si los elementos del conjunto  $Q$  estuvieran organizados mediante un montículo<sup>1</sup> en una cola de prioridad. En este caso, la operación *extract\_min* tendría un coste temporal  $O(\log |V|)$  y, como se realiza  $|V|$  veces, el coste temporal total de *extract\_min* sería  $O(|V| \log |V|)$ . Sin embargo, construir el montículo tendría un coste  $O(|V|)$ .

Por otro lado, actualizar una distancia provisional (en el bucle *para*) supondría modificar la prioridad (distancia provisional) del vértice en el montículo y, consecuentemente, reorganizar el montículo con un coste temporal  $O(\log |V|)$ ; por lo tanto, el coste de cada iteración del bucle *para* sería, en este caso,  $O(\log |V|)$ . Debido a que el bucle *para* se realiza  $|E|$  veces, el coste total de este bucle será  $O(|E| \log |V|)$ .

En definitiva, si organizáramos los vértices del conjunto  $Q$  en una cola de prioridad, el coste temporal del algoritmo sería:  $O(|V| + |V| \log |V| + |E| \log |V|) =$

---

<sup>1</sup>El mínimo estaría en la raíz y, para todo nodo excepto el raíz, su clave sería mayor o igual que la del padre.

$O((|V| + |E|) \log |V|)$ .

## 7.6. Árbol de expansión de coste mínimo

### 7.6.1. Árbol de expansión

A continuación introduciremos algunos conceptos necesarios para la exposición del problema que se pretende resolver.

Un grafo no dirigido, acíclico y conexo se denomina **árbol libre**. Un árbol libre cumple las siguientes propiedades:

1. Un árbol libre con  $n \geq 1$  nodos contiene exactamente  $n - 1$  arcos.
2. Si se añade una nueva arista a un árbol libre, se crea un ciclo.
3. Cualquier par de de vértices están conectados por un único camino.

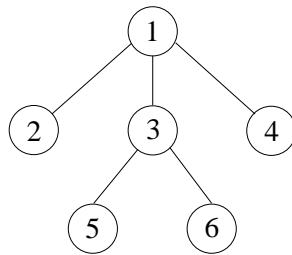


Figura 7.19: Ejemplo de árbol libre

Un **árbol de expansión de un grafo no dirigido**  $G = (V, E)$ , es un árbol libre  $T = (V', E')$ , tal que  $V' = V$  y  $E' \subseteq E$ ; es decir  $T$  contiene todos los vértices de  $G$ , y las aristas de  $T$  son aristas de  $G$ .

Dado un grafo no dirigido y ponderado mediante una función de ponderación  $w$ , el **coste de un árbol de expansión**  $T$  es la suma de los costes de todos los arcos del árbol.

$$w(T) = \sum_{(u,v) \in T} w(u, v)$$

Un **árbol de expansión de un grafo será de coste mínimo**, si se cumple que su coste es el menor posible respecto al coste de cada uno de los posibles árboles de expansión que contiene el grafo.

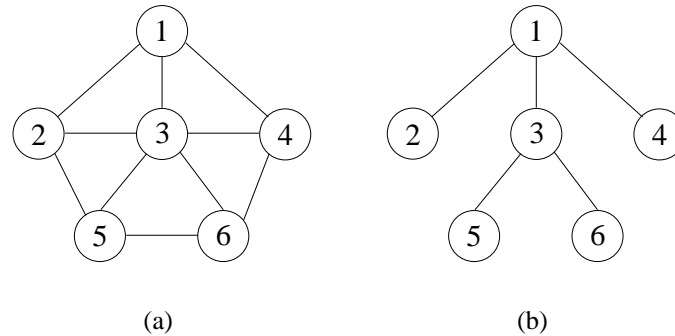


Figura 7.20: En la figura (b) se muestra un árbol de expansión del grafo que se muestra en la figura (a). Obsérvese que el árbol de expansión es un árbol libre que contiene todos los vértices y un subconjunto de aristas del grafo mostrado en (a).

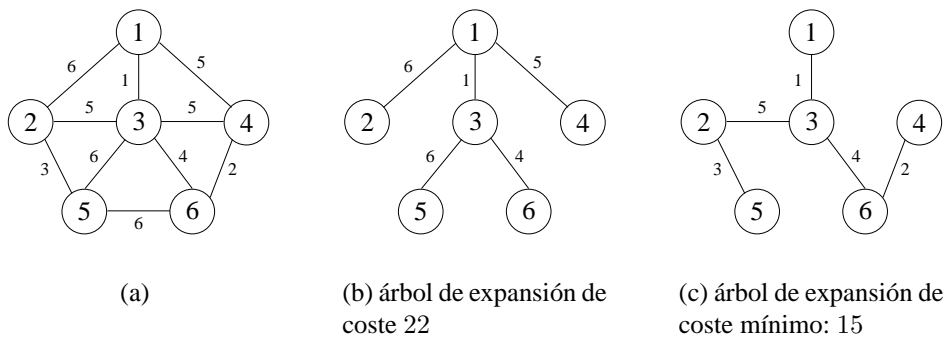


Figura 7.21: En la figura (b) se muestra un árbol de expansión del grafo ponderado que se muestra en la figura (a), y en la figura (c) se muestra un árbol de expansión de coste mínimo para el grafo de la figura (a).

Así podemos plantear el problema de obtener el árbol de expansión de coste mínimo para un grafo dado de esta forma:

**Problema:** Sea  $G = (V, E)$  un grafo no dirigido, conexo y ponderado mediante pesos asociados a los arcos; se desea obtener un nuevo grafo  $G' = (V, E')$  donde  $E' \subseteq E$  tal que  $G'$  sea un árbol de expansión de coste mínimo de  $G$ .

Existen dos algoritmos clásicos que resuelven este problema: el algoritmo de Kruskal y el algoritmo de Prim.

### 7.6.2. Algoritmo de Kruskal

El algoritmo de Kruskal resuelve eficientemente el problema de obtención de un árbol de expansión de coste mínimo para un grafo. Veamos en qué consiste:

El algoritmo mantiene las siguientes estructuras:

- Un conjunto  $A$  en el que se mantiene, en todo momento, aquellas aristas que ya han sido seleccionadas como pertenecientes al árbol de expansión de coste mínimo. Por lo tanto, cuando el algoritmo finalice, el conjunto  $A$  contendrá el subconjunto de aristas  $E' \subseteq E$  que forman parte del árbol de expansión de coste mínimo.
- Un MF-set que se utiliza para saber qué vértices están unidos entre sí en el bosque (conjunto de árboles) que se va obteniendo durante el proceso de construcción del árbol de expansión de coste mínimo. Hay que tener en cuenta que, a medida que se van añadiendo aristas que deben formar parte del árbol de expansión de coste mínimo al conjunto  $A$ , se irán uniendo vértices entre sí formando árboles, por lo que, en un momento dado del proceso de construcción, existirán distintos árboles (bosque) que, poco a poco, irán enlazándose entre sí hasta obtener un único árbol que se corresponderá con el árbol de expansión de coste mínimo. Cada subconjunto disjunto del MF-set contiene aquellos vértices que ya están unidos entre sí en el árbol de expansión de coste mínimo que se está obteniendo; por lo tanto, si dos vértices  $u, v$  pertenecen al mismo subconjunto disjunto indicará que existe un camino que los une en el árbol (dentro del bosque) en que se encuentren.

La estrategia que sigue el algoritmo es la siguiente:

1. Inicialmente el conjunto  $A$  está vacío  $A = \emptyset$ ; es decir, al principio, no existe ninguna arista que pertenezca al árbol de expansión de coste mínimo. Por lo tanto, inicialmente, el MF-set estará formado por  $|V|$  subconjuntos disjuntos que contendrán, cada uno de ellos, uno de los vértices del grafo  $G$ ; esto es así, ya que, como todavía no existen aristas que pertenezcan al árbol de expansión de coste mínimo, no pueden existir vértices conectados entre sí.
2. De entre todas las aristas  $(u,v) \in E$ , se selecciona, como candidata para formar parte del árbol de expansión de coste mínimo, aquélla que no se ha seleccionado todavía y tenga asociada un menor peso; es decir, la que, si se añade, provoque un incremento menor del peso del árbol de expansión de coste mínimo. Si el vértice  $u$  y  $v$  no están conectados entre sí en el árbol de expansión de coste mínimo ( $\text{Buscar}(u) \neq \text{Buscar}(v)$ ), al añadir esta nueva arista al árbol de expansión de coste mínimo, se unirán con el menor coste posible, todos los vértices conectados con  $v$  con todos los vértices conectados con  $u$  ( $\text{Union}(\text{Buscar}(u), \text{Buscar}(v))$ ), por lo que la arista  $(u,v)$  se añade al conjunto  $A$  para formar parte del árbol de expansión de coste mínimo. Si,

por el contrario, el vértice  $u$  y  $v$  ya estaban conectados entre sí, no se realiza ninguna acción ya que si se añadiera la arista al árbol de expansión de coste mínimo, éste dejaría de cumplir las propiedades de los árboles libres.

3. El paso (2) se repite una vez para cada arista  $(u,v) \in E$ .

A continuación se presenta el algoritmo. Dado un grafo  $G = (V, E)$  no dirigido, conexo y ponderado:

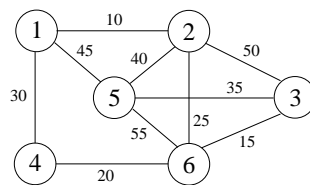
```

Algoritmo Kruskal( $G$ ){
   $A = \emptyset$ 
  para cada vértice  $v \in V$  hacer
    Crear_Subconjunto( $v$ )
  fin_para
  ordenar las aristas pertenecientes a  $E$ , según su peso, en orden no decreciente
  para cada arista  $(u,v) \in E$ , siguiendo el orden no decreciente hacer
    si Buscar( $u$ )  $\neq$  Buscar( $v$ ) entonces
       $A = A \cup \{(u,v)\}$ 
      Union(Buscar( $u$ ),Buscar( $v$ ))
    fin_si
  fin_para
  devuelve( $A$ )
}

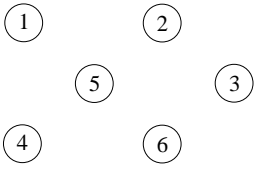
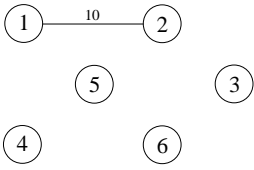
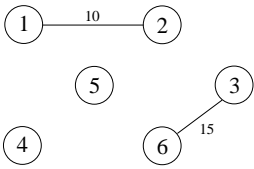
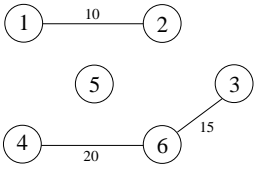
```

### Ejemplo de funcionamiento del algoritmo

A continuación se muestra un ejemplo de cómo el algoritmo de Kruskal obtiene el árbol de expansión de coste mínimo del grafo de la siguiente figura. Para cada iteración que realiza el algoritmo se muestra el estado del conjunto  $A$  y del MF-set; así como también el bosque que se va obteniendo durante el proceso de construcción del árbol de expansión de coste mínimo. También, para cada iteración, se muestra qué arista ha sido la seleccionada para formar parte del árbol, y, dependiendo de la arista escogida, las acciones que lleva a cabo el algoritmo.





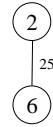
$A$	MF-set (componentes conexas)	Árbol de coste mínimo
$\{\}$	$\{\{1\},\{2\},\{3\},\{4\},\{5\},\{6\}\}$	
$\{(1,2)\}$	$\{\{1,2\},\{3\},\{4\},\{5\},\{6\}\}$	
$\{(1,2),(6,3)\}$	$\{\{1,2\},\{6,3\},\{4\},\{5\}\}$	
$\{(1,2),(6,3),(4,6)\}$	$\{\{1,2\},\{4,6,3\},\{5\}\}$	

---

A

MF-set (componentes conexas)    Árbol de coste mínimo

---

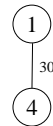
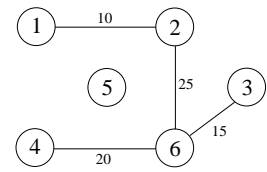


$\text{Buscar}(2) \neq \text{Buscar}(6) \longrightarrow A = A \cup \{(2,6)\}; \text{Union}(\text{Buscar}(2), \text{Buscar}(6))$

---

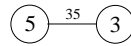
$\{(1,2), (6,3), (4,6), (2,6)\}$

$\{\{1,2,4,6,3\}, \{5\}\}$



$\text{Buscar}(1) = \text{Buscar}(4)$

---

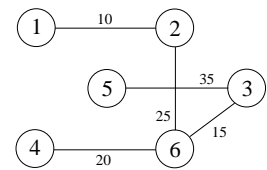


$\text{Buscar}(5) \neq \text{Buscar}(3) \longrightarrow A = A \cup \{(5,3)\}; \text{Union}(\text{Buscar}(5), \text{Buscar}(3))$

---

$\{(1,2), (6,3), (4,6), (2,6), (5,3)\}$

$\{\{5,1,2,4,6,3\}\}$



## Coste temporal del algoritmo

Para analizar el coste temporal del algoritmo asumiremos que en las operaciones sobre el MF-set *Unión* y *Buscar*, se aplican los heurísticos *unión por rango* y *compresión de caminos*, respectivamente. Por lo tanto, tal como se estudió en el tema 3, en ese caso, estas operaciones tendrán un coste prácticamente constante. Teniendo en cuenta que estas operaciones se realizan para cada arista del grafo, el

coste temporal total de realizar las operaciones sobre el MF-set será  $O(|E|)$ . Por otra parte, la creación de un subconjunto disjunto (operación *Crear\_Subconjunto*) tiene un coste constante, por lo que, la construcción inicial de  $|V|$  subconjuntos disjuntos tendrá un coste  $O(|V|)$ .

Otra operación que realiza el algoritmo es la de ir seleccionando las aristas en orden no decreciente. Si la lista de aristas se organizara mediante un montículo<sup>2</sup> en una cola de prioridad, el coste temporal de extraer la arista de mínimo peso sería  $O(\log |E|)$ . Debido a que se extraen todas las aristas de la cola de prioridad, se realizarían  $|E|$  operaciones de extracción, por lo que el coste temporal total sería  $O(|E| \log |E|)$ . Además, habría que añadir el coste temporal de construir el montículo  $O(|E|)$ .

En definitiva, podemos concluir que el coste temporal del algoritmo de Kruskal es  $O(|V| + |E| + |E| \log |E| + |E|) = O(|E| \log |E|)$ .

### 7.6.3. Algoritmo de Prim

El algoritmo de Prim resuelve el mismo problema que el de Kruskal (obtener el árbol de expansión de coste mínimo), pero aplicando otra estrategia.

El algoritmo de Prim utiliza las siguientes estructuras durante la ejecución del algoritmo:

- Un conjunto  $A$  en el que se guardan aquellas aristas que ya forman parte del árbol de expansión de coste mínimo.
- Un conjunto  $S$  inicialmente vacío, y al que se irán añadiendo los vértices de  $V$  conforme se vayan recorriendo para formar el árbol de expansión de coste mínimo.

La estrategia del algoritmo de Prim es: empezando por cualquier vértice, construir incrementalmente un árbol de recubrimiento, seleccionando en cada paso una arista  $(u,v) \in A$  tal que:

- Si se añade  $(u,v)$  al conjunto de aristas  $A$  obtenido hasta el momento no se cree ningún ciclo.
- Produzca el menor incremento de peso posible.
- Los arcos del árbol de expansión parcial forman un único árbol.
- El árbol toma como raíz un vértice cualquiera del grafo,  $u \in V$  y a partir de éste se extiende por todos los vértices del grafo  $G = (V, E)$ .

---

<sup>2</sup>El mínimo estaría en la raíz y, para todo nodo excepto el raíz, su clave (peso de la arista) sería mayor o igual que la del padre.

- En cada paso se selecciona el arco de menor peso que une un vértice presente en  $S$  con uno de  $V$  que no lo esté.

A continuación se presenta el algoritmo. Dado un grafo  $G = (V, E)$  no dirigido, conexo y ponderado:

```

Algoritmo Prim( $G$ ){
     $A = \emptyset$ 
     $u$  = elemento arbitrario de  $V$ 
     $S = \{u\}$ 
    mientras  $S \neq V$  hacer
         $(u,v) = \operatorname{argmin}_{(x,y) \in S \times (V-S)} \text{peso}(x, y)$ 
         $A = A \cup \{(u,v)\}$ 
         $S = S \cup \{v\}$ 
    fin_mientras
    devuelve( $A$ )
}

```

Donde la operación de búsqueda de la arista factible de menor peso, que hemos denotado como  $(u,v) = \operatorname{argmin}_{(x,y) \in S \times (V-S)} \text{peso}(x, y)$  se calcularía de la siguiente forma:

```

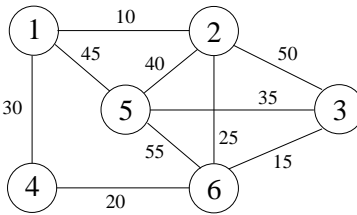
 $m = +\infty$ 
para  $x \in S$  hacer
    para  $y \in V - S$  hacer
        si  $\text{peso}(x, y) < m$  entonces
             $m = \text{peso}(x, y)$ 
             $(u,v) = (x,y)$ 
        fin_si
    fin_para
fin_para

```

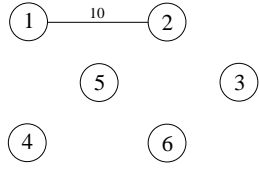
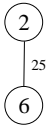
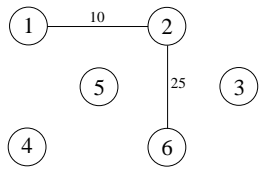
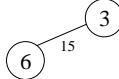
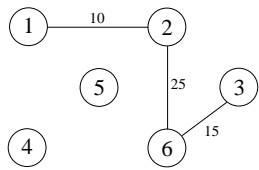
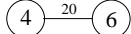
### Ejemplo del funcionamiento del algoritmo

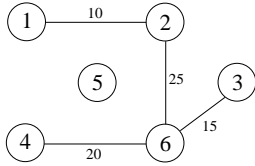
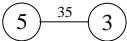
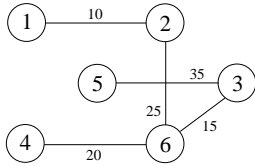
A continuación se muestra un ejemplo de cómo el algoritmo de Prim obtiene el árbol de expansión de coste mínimo del grafo de la siguiente figura (es el mismo

ejemplo que el presentado para el algoritmo de Kruskal). Para cada iteración que realiza el algoritmo se muestra el estado del conjunto  $A$  y del conjunto  $S$ . También , para cada iteración, se muestra qué arista ha sido la seleccionada para formar parte del árbol, y, dependiendo de la arista escogida, las acciones que lleva a cabo el algoritmo.



$A$	$S$	Árbol de coste mínimo
		<div> <div> <div>1</div> <div>2</div> </div> <div> <div>5</div> <div>3</div> </div> <div> <div>4</div> <div>6</div> </div> </div>
$\{\}$	$\{\}$	
<hr/>		
$A = \emptyset$ ; $u$ = elemento arbitrario de $V$ (Ej: $u=1$ );	$S = \{1\}$ ;	<div> <div>1</div> </div>
<hr/>		
		<div> <div>1</div> <div>2</div> </div> <div> <div>5</div> <div>3</div> </div> <div> <div>4</div> <div>6</div> </div>

$A$	$S$	Árbol de coste mínimo
$\{(1,2)\}$	$\{1,2\}$	
<hr/>		
		
$\text{argmin} \longrightarrow (u,v) = (2,6); A = A \cup \{(2,6)\}; S = S \cup \{6\};$		
<hr/>		
$\{(1,2),(2,6)\}$	$\{1,2,6\}$	
<hr/>		
		
$\text{argmin} \longrightarrow (u,v) = (6,3); A = A \cup \{(6,3)\}; S = S \cup \{3\};$		
<hr/>		
$\{(1,2),(2,6),(6,3)\}$	$\{1,2,6,3\}$	
<hr/>		
		
$\text{argmin} \longrightarrow (u,v) = (6,4); A = A \cup \{(6,4)\}; S = S \cup \{4\};$		
<hr/>		

$A$	$S$	Árbol de coste mínimo
$\{(1,2),(2,6),(6,3),(6,4)\}$	$\{1,2,6,3,4\}$	
<hr/> <div style="text-align: center;">  </div> $\text{argmin} \longrightarrow (u,v) = (3,5); A = A \cup \{(3,5)\}; S = S \cup \{5\};$ <hr/>		
$\{(1,2),(2,6),(6,3),(6,4),(3,5)\}$	$\{1,2,6,3,4,5\}$	

## Coste temporal del algoritmo

Analizando en primer lugar el coste del cálculo de  $(u,v) = \text{argmin}_{(x,y) \in S \times (V-S)} \text{peso}(x,y)$  vemos que por los dos bucles **para** anidados que tiene esta operación y que recorren los subconjuntos de vértices que se van creando, su coste temporal es  $O(|V|^2)$ .

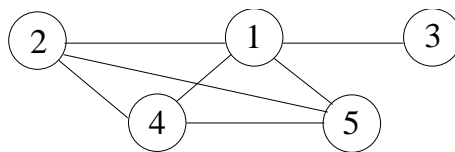
El algoritmo de Prim tiene un bucle principal **mientras** que realizará  $|V| - 1$  iteraciones, debido a que en la primera iteración el conjunto  $S$  solo tiene un vértice y a cada iteración sólo se añade un nuevo vértice de  $V$ . La operación principal que engloba este bucle es la de cálculo de la arista factible de menor peso, que como ya hemos analizado tiene un coste  $O(|V|^2)$ . Por todo esto el algoritmo de Prim tiene un coste  $O(|V|^3)$ .

Sin embargo existen algunas optimizaciones para el algoritmo del cálculo de  $(u,v) = \text{argmin}_{(x,y) \in S \times (V-S)} \text{peso}(x,y)$  que consiguen rebajarlo a un coste  $O(|V|)$  y por tanto se puede construir una versión optimizada del algoritmo de Prim con un coste  $O(|V|^2)$ , aunque no veremos aquí dicha optimización.

## 7.7. Ejercicios

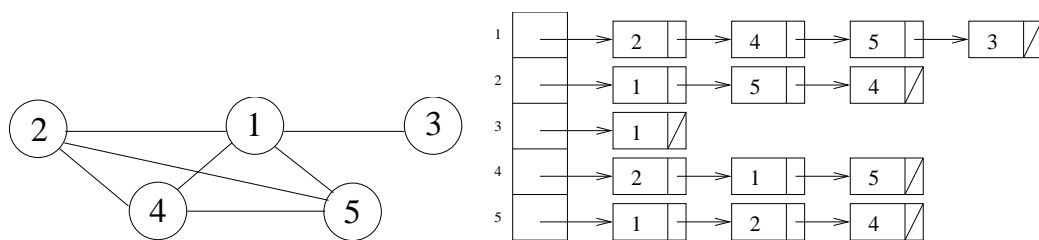
### Ejercicio 1:

-Haz una traza del algoritmo de recorrido de un grafo primero en profundidad para el grafo que se muestra a continuación. Para hacer la traza se recomienda previamente, representar la estructura del grafo mediante listas de adyacencia. El recorrido se inicia desde el vértice 1.



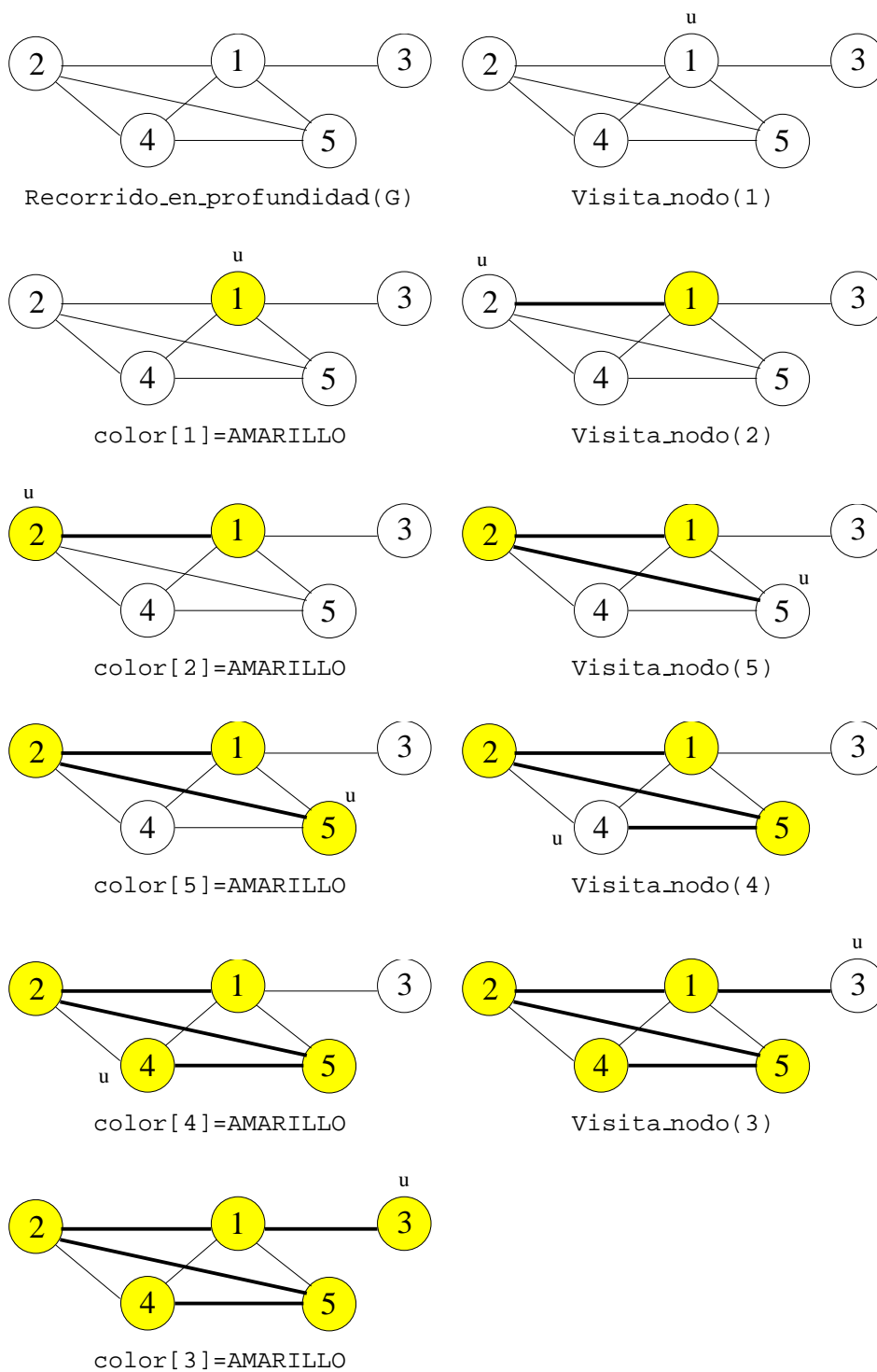
### Solución:

En primer lugar representamos el grafo mediante listas de adyacencia; el orden de los nodos que se ha elegido para las listas es arbitrario, aunque una vez se haya escogido un orden, debe respetarse para la traza del recorrido en profundidad.



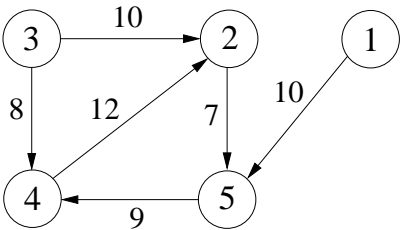
El vértice de inicio se ha especificado que es el 1, con lo que el recorrido en profundidad se realizaría de la siguiente manera:





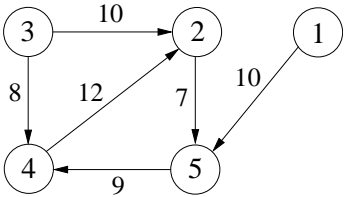
**Ejercicio 2:**

-Haz una traza del algoritmo de Dijkstra para el siguiente grafo, tomando como vértice origen el 1. Una vez obtenida la solución, recupera la secuencia de vértices que forman cada uno de los caminos más cortos, utilizando, para ello, el vector  $P$  que usa el algoritmo para almacenar los predecesores.

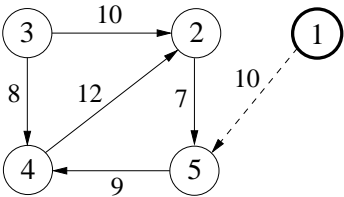


**Solución:**

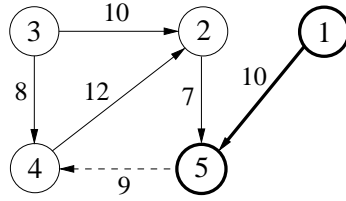
En el problema se ha especificado que el vértice de inicio es el 1. Así pues, la traza de Dijkstra para el grafo de la figura es:



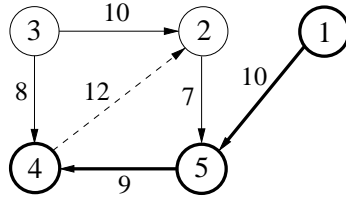
$S$	$Q$	$u$	1	2	3	4	5
$\{\}$	$\{1,2,3,4,5\}$	—	$D$ 0	$\infty$	$\infty$	$\infty$	$\infty$
			$P$ NULO	NULO	NULO	NULO	NULO



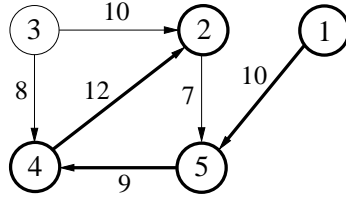
$S$	$Q$	$u$	1	2	3	4	5
$\{1\}$	$\{2,3,4,5\}$	1	$D$ 0	$\infty$	$\infty$	$\infty$	10
			$P$ NULO	NULO	NULO	NULO	1



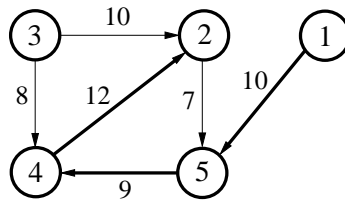
$S$	$Q$	$u$	1	2	3	4	5	
$\{1,5\}$	$\{2,3,4\}$	5	$D$	0	$\infty$	$\infty$	19	10
			$P$	$NULO$	$NULO$	$NULO$	5	1



$S$	$Q$	$u$	1	2	3	4	5	
$\{1,5,4\}$	$\{2,3\}$	4	$D$	0	31	$\infty$	19	10
			$P$	$NULO$	4	$NULO$	5	1



$S$	$Q$	$u$	1	2	3	4	5	
$\{1,5,4,2\}$	$\{3\}$	2	$D$	0	31	$\infty$	19	10
			$P$	$NULO$	4	$NULO$	5	1



$S$	$Q$	$u$	1	2	3	4	5	
$\{1,5,4,2,3\}$	$\{\}$	3	$D$	0	31	$\infty$	19	10
			$P$	$NULO$	4	$NULO$	5	1

Lo que ocurriría ahora sería que el algoritmo acabaría, pero con una solución un tanto extraña. La operación de extraer el mínimo de  $Q$  nos devolvería el vértice 3, puesto que no queda otro.

Ahora el bucle:

**mientras  $Q \neq \emptyset$  hacer**

del algoritmo de Dijkstra terminaría ya que el conjunto  $Q$  se queda vacío.

Sin embargo, el resultado nos dice que si comenzamos por el vértice 1, el vértice 3 es inalcanzable y por eso el camino de coste mínimo de 1 a 3 es infinito, o podemos decir que no existe camino de 1 a 3 ya que  $P[3] = \text{NULO}$ .

**Ejercicio 3:**

-Utilizando el algoritmo de Kruskal:

- a) ¿cómo se podría determinar si un grafo es conexo?
- b) ¿cómo se podrían obtener los vértices que forman cada una de las componente conexas?

No hace falta que escribas los algoritmos modificados, sólo comenta la estrategia a seguir y la modificación que se haría al algoritmo.

**Solución:**

- a) Si aplicamos el algoritmo de Kruskal, al acabar éste obtendremos como un subproducto un MF-SET que indica las componentes conexas que hay en el grafo, esto es, indica qué vértices están unidos entre sí al terminar el algoritmo.

Si el MF-SET tiene un solo subconjunto, esto es, una sola componente conexa, entonces el grafo es conexo.

- b) A partir del mismo MF-SET comentado en el apartado a). Este MF-SET contiene los vértices que están conectados entre sí en el grafo y los tiene agrupados en subconjuntos. Cada subconjunto disjunto del MF-SET se corresponde con una componente conexa, y los elementos de ese subconjunto son los vértices de esa componente conexa.
-

#### Ejercicio 4:

-Dados dos grafos  $G_1 = (V_1, E_1)$  y  $G_2 = (V_2, E_2)$ , con  $V_1 = V_2$ , se desea obtener un nuevo grafo  $G = (V, E)$  que sea la **diferencia simétrica** de  $G_1$  y  $G_2$ , tal que se cumpla que  $V = V_1 = V_2$ , y el conjunto de aristas  $E$  contenga las aristas de  $E_1$  que no estén en  $E_2$ , junto con las de  $E_2$  que no estén en  $E_1$ .

Escribe una función en lenguaje C con la cabecera:

```
grafo *diferencia_simetrica(grafo *G1, grafo *G2)
```

que construya el grafo resultante  $G$ , suponiendo que los grafos están representados mediante matrices de adyacencia con la definición de tipos vista en clase:

```
#define MAXVERT 1000
```

```
typedef struct{
    int talla;
    int A[MAXVERT][MAXVERT];
} grafo;
```

Analiza también el coste temporal del algoritmo.

#### Solución:

La función tendrá que crear una nueva estructura grafo (reserva de memoria), e indicar que el nuevo grafo creado tiene el mismo número de vértices que  $G_1$  o  $G_2$ , puesto que  $V = V_1 = V_2$ . La estrategia a seguir para obtener las aristas correspondientes al grafo resultado  $G$  aprovecha que la representación de los grafos se realiza mediante una matriz de adyacencias para obtener dichas aristas con un solo recorrido conjunto de las matrices de adyacencia de  $G_1$  y  $G_2$ .

La idea es que en  $G$  habrá una arista del vértice  $i$  al  $j$  (es decir,  $G[i][j] = 1$ ) si:

- en  $G_1$  hay una arista del vértice  $i$  al  $j$  y no la hay en  $G_2$  (es decir,  $G_1[i][j] == 1$  y  $G_2[i][j] == 0$ ), o bien
- en  $G_2$  hay una arista del vértice  $i$  al  $j$  y no la hay en  $G_1$  (es decir,  $G_1[i][j] == 0$  y  $G_2[i][j] == 1$ ).

Como se puede observar, los dos casos en los que habrá una arista en  $G$  del vértice  $i$  al  $j$  será cuando las posiciones  $(i, j)$  de las matrices de adyacencias de  $G_1$  y  $G_2$  tengan un contenido diferente, con lo que los dos casos se pueden reducir a comparar el contenido de las posiciones correspondientes en las matrices.

```

grafo *diferencia_simetrica(grafo *G1, grafo *G2) {
    grafo *G;
    int i,j;
    int t;

    G = (grafo *) malloc(sizeof(grafo));
    /* Sabemos que el numero de vertices de G1 y G2 es */
    /* el mismo --> sus matrices tienen la misma talla. */
    t = G1->talla;
    /* Recorremos las matrices que representan las aristas */
    /* de G1 y G2 y pondremos como aristas de G aquellas */
    /* de G1 y G2 que sean diferentes. */
    for (i=1;i<=t;i++)
        for (j=1;j<=t;j++)
            if (G1->A[i][j] != G2->A[i][j])
                G->A[i][j] = 1;
            else G->A[i][j] = 0;
    return (G) ;
}

```

Como se ve claramente, el coste del algoritmo será  $O(|V|^2)$ , esto es, el cuadrado del número de vértices del grafo resultado (que es el mismo número de vértices que tienen el grafo  $G_1$  y  $G_2$ ) y que viene dado por los dos bucles `for` anidados, cada uno de los cuales se ejecuta  $|V|$  veces.

# Tema 8

## Algoritmos Voraces

### 8.1. Introducción

Los objetivos básicos de este tema son: estudiar la técnica de diseño de algoritmos voraces y estudiar algunos problemas clásicos, como el problema de la mochila con fraccionamiento.

Normalmente esta técnica de diseño de algoritmos se aplica a **problemas de optimización**, esto es, a la búsqueda del valor óptimo (máximo o mínimo) de una cierta función objetivo en un dominio determinado. Más adelante veremos algunos ejemplos como el problema del Cajero automático y el problema de la mochila con fraccionamiento.

La solución a un problema mediante un algoritmo voraz se puede presentar como una secuencia de decisiones:

- Las decisiones se toman en base a una medida local de optimización. Esta medida local de optimización puede entenderse como un criterio de manejo de datos.
- Las decisiones que se toman para buscar una solución determinada son irreversibles.

Un algoritmo voraz no siempre encuentra la solución óptima, pero en ocasiones permite encontrar una solución aproximada con un coste computacional bajo.

La estrategia voraz que ha de seguir un algoritmo voraz consiste en que en cada paso del algoritmo se debe realizar una selección entre las opciones existentes. La estrategia voraz aconseja realizar la elección que es mejor en ese momento. Tal estrategia no garantiza, de forma general, que se encuentre la solución óptima al problema.



### 8.1.1. Ejemplo: Cajero automático

Vamos a ver cómo se puede resolver un sencillo problema mediante una estrategia voraz. Supongamos que tenemos que programar el control de un cajero automático.

El problema a resolver es suministrar la cantidad de dinero solicitada en billetes usando sólo los tipos de billetes especificados y de manera que el número total de billetes sea mínimo.

Por ejemplo supongamos que se solicita una cantidad  $M = 1100$  Euros, disponiendo de billetes de cantidad  $\{100, 200, 500\}$ . Algunas de las posibles soluciones que podemos dar para este problema serían:

- $11 \times 100$  (11 billetes).
- $5 \times 200 + 1 \times 100$  (6 billetes).
- $2 \times 500 + 1 \times 100$  (3 billetes).

Una posible estrategia voraz consistiría en ir seleccionando siempre el billete de mayor valor, siempre y cuando queden billetes de ese tipo y al añadir uno más de esos billetes no nos pasemos de la cantidad total de dinero solicitada.

Respecto a esta estrategia podemos observar que:

- a veces no hay solución, por ejemplo, si  $M = 300$  y no hay billetes de 100.
- a veces la solución no se encuentra, por ejemplo, si  $M = 1100$  y no hay billetes de 100 (Nuestro algoritmo cogería  $2 \times 500$  y no podría seguir mientras que sí que existiría una solución:  $1 \times 500 + 3 \times 200$ ).
- a veces encuentra una solución factible, pero no óptima; por ejemplo, si los tipos de billetes de que disponemos son  $\{10, 50, 110, 500\}$  y  $M = 650$ , la solución que se obtendría sería:  $1 \times 500 + 1 \times 110 + 4 \times 10$  (6 billetes) mientras que la solución óptima es:  $1 \times 500 + 3 \times 50$  (4 billetes).

## 8.2. Esquema general Voraz

Vamos a describir a nivel general el esquema que ha de seguir un algoritmo que pretenda utilizar una estrategia voraz para la resolución de un problema. Para ello usaremos la siguiente notación:

$C$  : Conjunto de elementos o candidatos a elegir.

$S$  : Conjunto de elementos de la solución en curso (secuencia de decisiones tomadas).

**solución**( $S$ ) : función que nos indica si  $S$  es solución o no.

**factible**( $S$ ) : función que nos indica si  $S$  es factible o completable.

**selecciona**( $C$ ) : función que selecciona un candidato o elemento de  $C$  conforme al criterio definido.

$f$  : función objetivo a optimizar.

Básicamente, el algoritmo voraz obtiene un subconjunto de  $C$  que optimiza la función objetivo. Para ello, los pasos a seguir son:

1. Seleccionar en cada instante un candidato.
2. Añadir el candidato a la solución en curso si es completable, sino rechazarlo.
3. Repetir 1 y 2 hasta obtener la solución.

NOTA IMPORTANTE:

- Un algoritmo voraz no siempre proporciona la solución óptima.
- las decisiones son irreversibles.

El esquema de diseño voraz tiene esta forma:

```
Algoritmo Voraz( $C$ ) {  
     $S = \emptyset$ ; /* conjunto vacío */  
    while ((!solución( $S$ )) && ( $C \neq \emptyset$ )) {  
         $x = \text{selecciona}(C)$ ;  
         $C = C - \{x\}$   
        if (factible( $S \cup \{x\}$ )) {  
             $S = S \cup \{x\}$ ;  
        }  
    }  
    if (solución( $S$ )) return( $S$ );  
    else return(No hay solución);  
}
```

Por ejemplo, aplicando el esquema anterior al problema del cajero automático, el conjunto  $C$  sería el conjunto de billetes que tiene disponibles el cajero: {24 billetes de 100 Euros, 32 billetes de 200 Euros, etc. }; la función **factible**( $S$ )

calcularía si al añadir el nuevo billete candidato nos pasamos de la cantidad total solicitada, etc.

En las próximas secciones vamos a exponer dos problemas clásicos resueltos mediante algoritmos voraces: el problema de la compresión de ficheros usando códigos de Huffman y el problema de la mochila con fraccionamiento.

### 8.3. El problema de la compresión de ficheros

Vamos a abordar el problema de la compresión de ficheros de textos (formados por secuencias de caracteres) mediante el uso de los códigos de Huffman.

Supongamos que tenemos un fichero con 100.000 caracteres (solamente aparecen los caracteres  $a, b, c, d, e, f$ ) cuyas frecuencias de aparición son las siguientes:

caracter	$a$	$b$	$c$	$d$	$e$	$f$
Frecuencia $\times 10^3$	45	13	12	16	9	5

Para codificar esas letras en el computador podríamos usar en un principio el clásico código ASCII de 8 bits para cada caracter. Sin embargo, como sólo estamos trabajando con 6 letras, podemos codificar cada una de ellas con 3 bits. Así establecemos los siguientes códigos para las letras (codificación fija):

caracter	$a$	$b$	$c$	$d$	$e$	$f$
Código Fijo	000	001	010	011	100	101

Con este sistema de codificación, el espacio de memoria requerido para cada caracter es el mismo, 3 bits, por lo que el tamaño del fichero de 100.000 caracteres será 300 Kbits.

Ahora bien, como se puede comprobar en la tabla de frecuencias de aparición de cada caracter en el fichero, existen unos caracteres que aparecen muchas más veces que otros. Así pues, si consiguiéramos codificar con menos bits los caracteres muy frecuentes conseguiremos reducir considerablemente el tamaño del fichero, para esto veremos más adelante que será necesario codificar los caracteres menos frecuentes con más bits, pero aún así se conseguirá una reducción del número total de bits del fichero. Proponemos una nueva codificación de bits para los caracteres del fichero (codificación variable):

caracter	$a$	$b$	$c$	$d$	$e$	$f$
Código Variable	0	101	100	111	1101	1100

Con esta nueva codificación el fichero de 100.000 caracteres ocupará 224 Kbits, con lo que hemos conseguido comprimir el fichero original.

Con este tipo de codificaciones podemos definir un algoritmo de compresión. La compresión será a partir de la secuencia de bits del fichero original, si los interpretamos como los códigos de caracteres correspondientes a la codificación fija vista anteriormente, obtendremos una secuencia de caracteres, si ahora codificamos esos caracteres con el código variable también visto anteriormente, obtendremos una secuencia de bits, más corta que la original, pero que la identifica plenamente. Como ejemplo:

$$000001010 \xleftrightarrow{\text{fijo}} abc \xleftrightarrow{\text{variable}} 0101100$$

Hemos reducido la secuencia original de 9 bits a 7 bits.

El que esta codificación variable sirva como esquema de compresión se debe a que la codificación variable que hemos usado cumple la propiedad de prefijo: ningún código es prefijo de otro. Con lo que cada caracter quedará plenamente determinado por su secuencia de bits asociada, sea de la longitud que sea, y a la hora de interpretar una secuencia de bits del fichero comprimido, no existirá ambigüedad, esto es, no habrá confusión en la interpretación, sino que identificará a una secuencia de caracteres y solo a una. Un caso particular de código que cumple esta propiedad es el código de Huffman, utilizado para compresión de ficheros. Estos códigos se generan a partir del fichero a comprimir. Para ello existe un algoritmo voraz que crea un árbol binario que se corresponde con el código de Huffman para ese fichero. En ese árbol binario las aristas están etiquetadas con un 0 o con un 1, en las hojas tendremos los caracteres que forman nuestro vocabulario, y el código de Huffman para cada caracter se obtendrá con la secuencia de etiquetas de las aristas que forman el camino desde la raíz hasta la hoja correspondiente al caracter. En la figura 8.1 se puede observar el árbol binario correspondiente a la codificación propuesta para el ejemplo anterior.

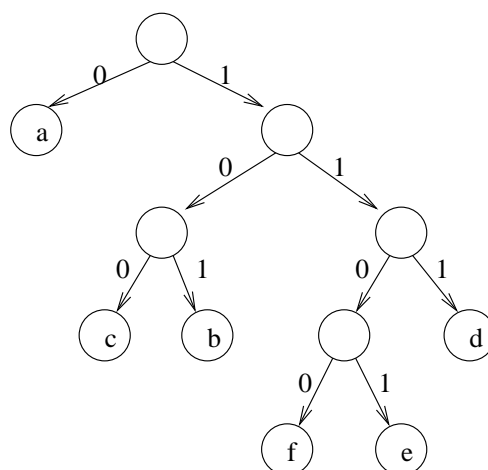


Figura 8.1: Árbol binario con la codificación Huffman para el ejemplo anterior

Lo que pretende el código de Huffman es eliminar la redundancia en la codificación de un mensaje.

Para construir un árbol como el de la figura 8.1 necesitaremos tener determinado el conjunto de símbolos  $C$  y el conjunto de frecuencias  $F$ . Además, necesitaremos un montículo  $Q$ , en el que guardaremos el conjunto de frecuencias formando un MINHEAP<sup>1</sup>. A partir de ahí, extraeremos cada vez las dos menores frecuencias del montículo  $Q$  y las colgaremos como hijo izquierdo e hijo derecho respectivamente de un nuevo nodo que crearemos, el nuevo nodo creado tendrá asociada como frecuencia la suma de las frecuencias de sus hijos y se insertará la frecuencia del nuevo nodo en el montículo  $Q$ . Así hasta que no queden más nodos que juntar. Esto indicará que el árbol que hemos ido construyendo ya incluye a todos los símbolos. Cuando busquemos un determinado símbolo en ese árbol, al recorrer el camino desde la raíz al nodo correspondiente al símbolo, si descendemos por un hijo izquierdo, esto supondrá un bit 0 y si descendemos por un hijo derecho, supondrá un bit 1.

El algoritmo que, a partir de un conjunto de símbolos (que constituye el vocabulario) y un conjunto de frecuencias asociadas a cada uno de los símbolos, crea este árbol, es el siguiente:

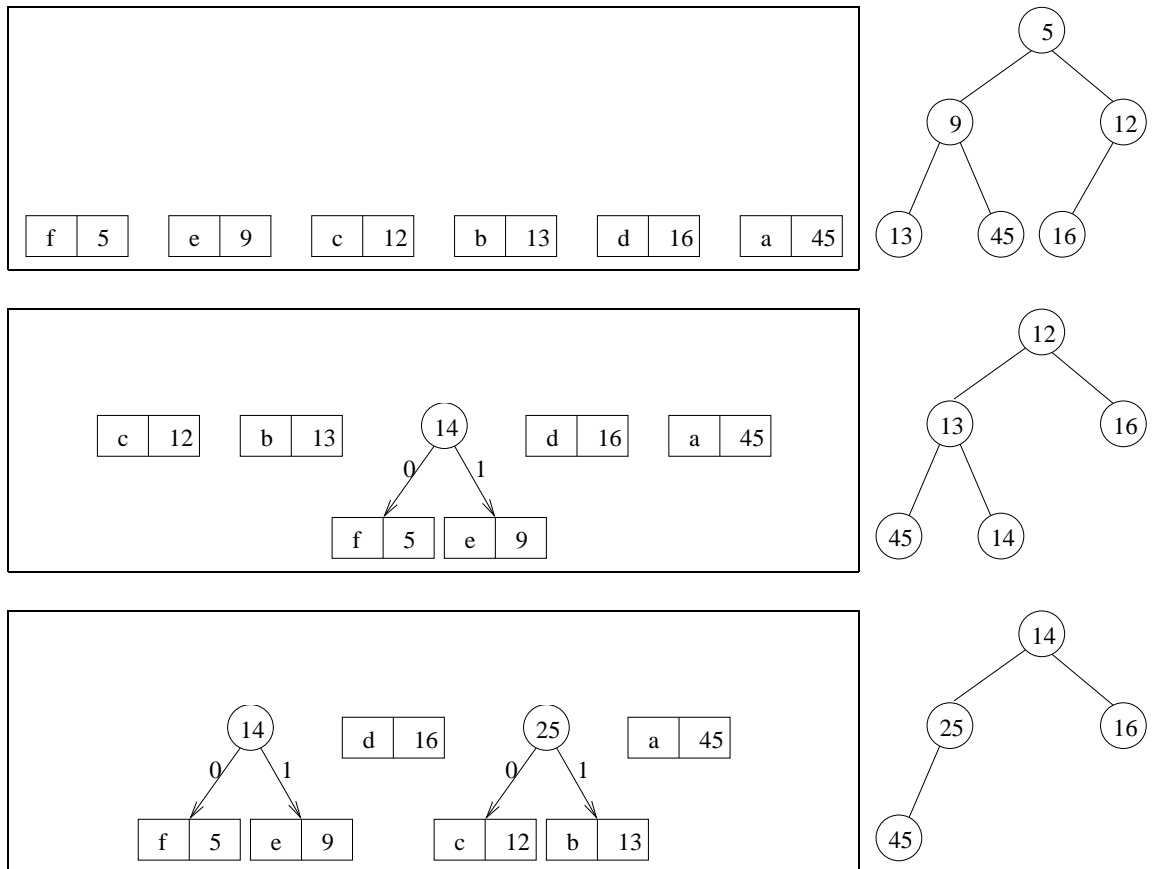
```
Algoritmo Huffman( $C, F$ ) {  
  for ( $x \in C, f_x \in F$ ) {  
    crear_hoja( $x, f_x$ );  
  }  
   $Q \leftarrow F$ ;  
  build_heap( $Q$ );  
  for ( $i=1; i < |C|; i++$ ) {  
     $z = \text{crear\_nodo}()$ ;  
     $z.\text{hizq} = \text{minimo}(Q)$ ;  
    extract_min( $Q$ );  
     $z.\text{hder} = \text{minimo}(Q)$ ;  
    extract_min( $Q$ );  
     $z.\text{frec} = z.\text{hizq}.\text{frec} + z.\text{hder}.\text{frec}$ ;  
    insertar( $Q, z$ );  
  }  
  return(minimo( $Q$ ));  
}
```

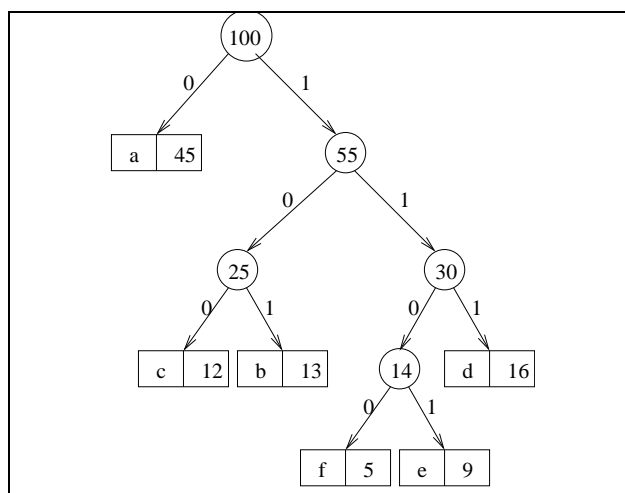
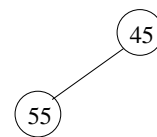
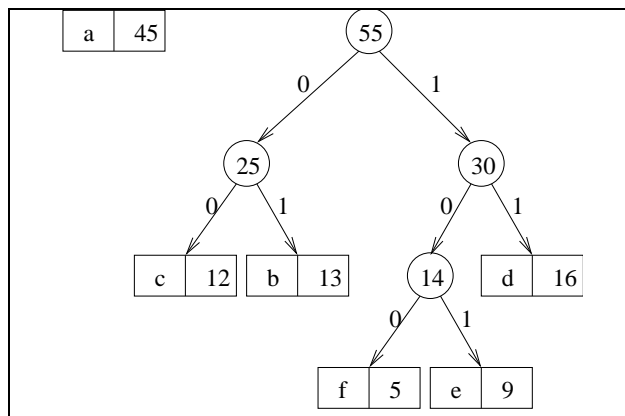
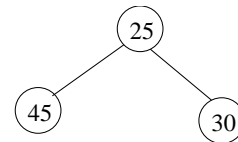
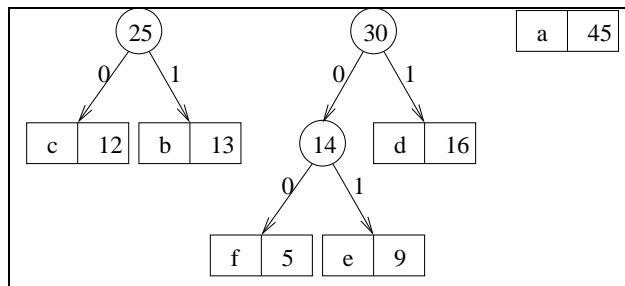
---

<sup>1</sup>Ver práctica de montículos

El coste del algoritmo es  $O(n \log n)$ , siendo  $n$  la talla del vocabulario, es decir, el número de caracteres diferentes que hay.

Vemos ahora la traza de cómo obtiene el algoritmo el árbol binario propuesto para el ejemplo anterior. Partimos del conjunto de símbolos con sus frecuencias asociadas y el montículo de las frecuencias:





Que se corresponde con el árbol de la figura 8.1.

## 8.4. El problema de la mochila con fraccionamiento

Supongamos que tenemos una mochila con una capacidad  $M$  (en peso), y tenemos también  $N$  objetos para incluir en ella. Cada objeto tiene un peso  $p_i$  y un beneficio  $b_i$ ,  $1 \leq i \leq N$ .

**Problema:** ¿Cómo llenar la mochila de forma que el beneficio total de los elementos que contenga sea máximo? Suponemos que los objetos se pueden fraccionar, esto es, se puede introducir en la mochila  $1/3$  de un objeto, con el peso y beneficio correspondientes a ese  $1/3$ . Tampoco se puede sobrepasar la capacidad de la mochila.

**Planteamiento del problema:**

- Los datos del problema son: la capacidad  $M$  y los  $N$  objetos con sus pesos asociados  $(p_1, \dots, p_N)$  y sus beneficios asociados  $(b_1, \dots, b_N)$ .
- El resultado que queremos obtener es una secuencia de  $N$  valores  $(x_1, \dots, x_N)$ , donde  $x_i$  cumple  $0 \leq x_i \leq 1$ , para  $1 \leq i \leq N$ . Esto es,  $x_i$  indica la parte fraccional que se toma del objeto  $i$ .
- Se desea maximizar el beneficio  $\sum_{i=1}^N b_i x_i$  (la suma de los beneficios correspondientes a la parte fraccional de cada objeto incluido) con la restricción de que los objetos quepan en la mochila  $\sum_{i=1}^N p_i x_i \leq M$

Algunos ejemplos de soluciones factibles para el caso en que la capacidad de la mochila sea  $M = 20$ , los pesos de los objetos sean  $(18, 15, 10)$  y los beneficios asociados sean  $(25, 24, 15)$ , son:

Solución	Peso total	Beneficio total
$(1/2, 1/3, 1/4)$	16.5	24.25
$(1, 2/15, 0)^*$	20.0	28.20
$(0, 2/3, 1)^{**}$	20.0	31.00
$(0, 1, 1/2)^{***}$	20.0	31.50

Como es evidente, la solución óptima deberá ser una que llene la mochila al completo.

Nuestro objetivo final es maximizar el beneficio total de los elementos de la mochila. Para ello, aplicando una estrategia voraz, aplicaremos un criterio de optimización local con la idea de que este criterio obtenga una solución global lo más óptima posible. Para nuestro problema podemos definir tres criterios de selección de objetos:

- Escoger a cada iteración el elemento de mayor beneficio\*.



- Escoger a cada iteración el elemento de menor peso\*\*.
- Escoger a cada iteración el elemento de mayor relación beneficio/peso\*\*\*.

La idea es que cogemos elementos completos (es decir,  $x_i=1$ ) mientras no sobrepasemos el peso  $M$  permitido en la mochila. Cuando no podamos coger más elementos completos, cogemos del siguiente objeto que cumpla el criterio de selección, la parte fraccional correspondiente para llenar la mochila hasta el tope.

El algoritmo recibe el vector de pesos,  $p$ , el vector de beneficios,  $b$ , el tamaño de la mochila,  $M$  y el número de objetos,  $N$ . Además, crea un conjunto  $C$  que representa los objetos que hay para elegir asignándole un número identificador a cada uno y genera el vector `solucion`. El algoritmo para resolver el problema de la mochila fraccional es:

```

Algoritmo Mochila-fraccional( $p, b, M, N$ ) {
     $C = \{1, 2, \dots, N\}$ ;
    for ( $i = 1; i \leq N; i++$ )  $solucion[i] = 0$ ;
    while ( $(C \neq \emptyset) \ \&\& \ (M > 0)$ ) {
        /*  $i$  = elemento de  $C$  con máximo beneficio  $b[i]$ ; */
        /*  $i$  = elemento de  $C$  con mínimo peso  $p[i]$ ; */
         $i$  = elemento de  $C$  con máxima relación  $b[i]/p[i]$ ;
         $C = C - \{i\}$ ;
        if ( $p[i] \leq M$ ) {
             $solucion[i] = 1$ ;
             $M = M - p[i]$ ;
        } else {
             $solucion[i] = M/p[i]$ ;
             $M = 0$ ;
        }
    }
    return( $solucion$ );
}

```

Se puede demostrar que si los objetos se seleccionan por orden decreciente del criterio beneficio/peso, el algoritmo Mochila-fraccional encuentra una solución óptima al problema (ver [Brassard]).

**Ejemplo:** Observa como se obtiene la solución con cada uno de los tres criterios de selección de elementos propuestos para un problema donde  $M = 50$ , los pesos son  $p = (30, 18, 15, 10)$  y los beneficios son  $b = (25, 13, 12, 10)$ :

1. Criterio: seleccionar objeto de mayor beneficio  $b_i$ .

Iteración	$M$	$C$	Solución				Beneficio
	50	{1,2,3,4}	0	0	0	0	
1	20	{2,3,4}	1	0	0	0	
2	2	{3,4}	1	1	0	0	
3	0	{4}	1	1	2/15	0	$25 + 13 + (2/15) \cdot 12 = 39.6$

2. Criterio: seleccionar objeto de menor peso  $p_i$ .

Iteración	$M$	$C$	Solución				Beneficio
	50	{1,2,3,4}	0	0	0	0	
1	40	{1,2,3}	0	0	0	1	
2	25	{1,2}	0	0	1	1	
3	7	{1}	0	1	1	1	
4	0	$\emptyset$	7/30	1	1	1	$(7/30) \cdot 25 + 13 + 12 + 10 = 40.83$

3. Criterio: seleccionar objeto de mayor beneficio unitario  $b_i/p_i = (5/30, 13/18, 12/15, 10/10)$ .

Iteración	$M$	$C$	Solución				Beneficio
	50	{1,2,3,4}	0	0	0	0	
1	40	{1,2,3}	0	0	0	1	
2	10	{2,3}	1	0	0	1	
3	0	{2}	1	0	10/15	1	$25 + (10/15) \cdot 12 + 10 = \boxed{42.99}$

### Coste temporal del algoritmo de la Mochila fraccional

El bucle **for** que inicializa el vector de *solucion* en el algoritmo de Mochila-fraccional tiene un coste de  $O(N)$ .

Por otro lado el bucle **mientras** selecciona uno de los  $N$  posibles objetos a cada iteración y como mucho realizará tantas iteraciones como objetos existan, es decir realizará  $O(N)$  iteraciones como máximo.

La operación de seleccionar el elemento de  $C$  según el criterio elegido, en el caso en que  $C$  estuviera representado en un vector, tendrá un coste  $O(N)$ . Como esta operación se encuentra dentro del bucle **mientras**, el coste total de este bucle será  $O(N^2)$ .

Sumando el coste de inicializar el vector solución y el del bucle **mientras** obtenemos que el coste total de Mochila-fraccional es:  $O(N + N^2) \in O(N^2)$ .

Sin embargo, podemos realizar una implementación más eficiente del algoritmo, si ordenamos previamente los objetos conforme a la relación  $b/p$  (o el criterio escogido), el coste de la selección es  $O(1)$  y ordenar los objetos tendrá un coste  $O(N \log N)$ , con lo que el coste total del algoritmo es:  $O(N + N + N \log N) \in O(N \log N)$ .

## 8.5. Ejercicios

### Ejercicio 1:

-Se tiene un disco de tamaño  $T$  y  $N$  ficheros que se desean grabar en el disco. El tamaño total de los ficheros puede ser mayor que  $T$  y por tanto puede que no quepan todos. Se desearía un algoritmo voraz que decidiera qué ficheros almacenar en el disco, suponiendo que:

- Se dispone de un vector  $t[1 \dots N]$  con los tamaños de los  $N$  ficheros.
- La solución se da como un vector  $s[1 \dots N]$ , en el que  $s[i] = 1$  indica que el fichero  $i$ -ésimo se almacena en el disco, y  $s[i] = 0$  en caso contrario.
- Se desea maximizar el coeficiente de ocupación del disco definido como:

$$\frac{s[1]t[1] + s[2]t[2] + \dots + s[N]t[N]}{T}$$

Para ello, se propone el siguiente algoritmo: “Ordenar los ficheros por orden decreciente de tamaño. Suponiendo los ficheros numerados de acuerdo con dicho orden, desde el primero hasta el último, se van grabando mientras quede espacio en el disco. Si un determinado fichero no cabe, se descarta y se pasa al siguiente.”

Se pide:

- a) Discutir el coste temporal del algoritmo propuesto.
- b) Dar un contraejemplo para demostrar que este algoritmo no siempre obtiene la solución óptima.

### Solución:

- a) El coste temporal será el mismo que para el algoritmo de la mochila (en su versión no fraccional, donde los objetos no se pueden fraccionar), esto es debido a que el problema y el algoritmo de resolución son extremadamente similares.

La ordenación del vector  $t$  de los tamaños de los ficheros por orden decreciente de tamaño tendrá un coste de  $O(N \log N)$  (por ejemplo, usando quicksort).

Después habrá que realizar un bucle que vaya recorriendo el vector  $t$  hasta que o no queden ficheros por comprobar (se han grabado todos los ficheros disponibles o ninguno de los que quedan por grabar cabe en el espacio restante en disco) o se haya llegado a la ocupación máxima del disco (se

haya llenado todo el disco). Este bucle tendrá un coste máximo correspondiente al tamaño del vector  $t$ , esto es,  $O(N)$ .

Sumando todo, el coste final es  $O(N + N \log N) \in O(N \log N)$ .

**b)** Un posible contraejemplo podría ser éste:

Supongamos que tenemos un disco con capacidad  $T = 100$  Mb. y tenemos ficheros de tamaño (42, 40, 33, 25, 12, 10). En primer lugar ordenamos los ficheros de mayor a menor, con lo que obtenemos el siguiente vector de tamaños ordenado:

1	2	3	4	5	6
42	40	33	25	12	10

Y ahora vamos recorriendo el vector anterior y grabando ficheros en el disco conforme a la estrategia dictada anteriormente, para ello usamos un bucle `for (i=1; i<=6; i++)`:

i=1

1	2	3	4	5	6
s= 1	0	0	0	0	0

$T = 100 - 42 = 58$

---

i=2

1	2	3	4	5	6
s= 1	1	0	0	0	0

$T = 58 - 40 = 18$

---

i=3

1	2	3	4	5	6
s= 1	1	0	0	0	0

como  $t[3] = 33$ , no cabe en el disco

---

i=4

1	2	3	4	5	6
s= 1	1	0	0	0	0

como  $t[4] = 25$ , no cabe en el disco

---

---

i=5

$s =$ 

1	2	3	4	5	6
1	1	0	0	1	0

$T = 18 - 12 = 6$

---

i=6

$s =$ 

1	2	3	4	5	6
1	1	0	0	1	0

como  $t[6] = 10$ , no cabe en el disco

---

NO QUEDAN MÁS FICHEROS

El coeficiente de ocupación que obtenemos con esta solución es:

$$\text{Coeficiente de ocupación} = \frac{42 + 40 + 12}{100} = \frac{94}{100} = 0,94$$

Sin embargo la solución óptima sería la siguiente:

$s =$ 

1	2	3	4	5	6
1	0	1	1	0	0

cuyo coeficiente de ocupación es:

$$\text{Coeficiente de ocupación} = \frac{42 + 33 + 25}{100} = \frac{100}{100} = 1$$

que es el máximo posible.

## **Exámenes de la asignatura**

# Estructuras de Datos y Algoritmos

**Febrero 2002**

## **Examen de Teoría: SOLUCIONES**

Departamento de Sistemas Informáticos y Computación  
Escuela Politécnica Superior de Alcoy

### **Ejercicio: (1 punto)**

-Supón que tienes que implementar la gestión de la cola de clientes de un taller mecánico. Lo normal sería atender a los clientes en el orden en el que van llegando al taller. Sin embargo, el dueño del taller quiere tener en consideración la prisa que un cliente pueda tener por recoger su auto. Con lo que habrá que permitir asignar una prioridad para cada cliente según la prisa que tenga, por ejemplo, un comercial necesita urgentemente el coche y no puede esperar mucho porque si no puede viajar, pierde ventas, mientras que un jubilado que sólo lo usa para pasear al perro, puede esperar mucho más tiempo.

- a) ¿qué estructura de datos (TAD) propones utilizar para poder gestionar esta cola de clientes. Razónalo.(0.5 puntos)
- b) Nombra dos operaciones (las que quieras) asociadas al TAD que has definido en el apartado anterior y explica brevemente en qué consisten. (0.5 puntos)

### **Solución:**

- a) La estructura más adecuada sería una cola con prioridades, refiriéndonos con esto a una estructura de datos cola tradicional, donde los elementos se insertan por el final de la cola, pero pueden avanzar tanto hacia la cabeza como su prioridad les permita. Para ello, cada uno de los elementos de la cola deberá estar caracterizado por su prioridad. La única operación distinta del TAD cola clásico es la de insertar, pues debe comparar con los elementos de la cola empezando desde el final hasta llegar a la posición de la cola donde su prioridad indique que debe insertarse. Los elementos se extraen como siempre por la cabeza de la cola, de hecho, si nos fijamos, tal y como se ha definido el TAD, el elemento que se encuentre en la cabeza de la cola será el que tenga más prioridad de toda la cola.



- b) A gusto del consumidor: las operaciones posibles asociadas a este TAD serían las mismas que para el TAD cola visto en clase, excepto la operación de insertar, que incluiría la prioridad además del elemento y tendría que insertar en la posición que dicha prioridad le indicase. Por lo demás, el resto de operaciones sería igual.
- 

### Ejercicio: (4 puntos)

-Supongamos que estamos desarrollando el software de la agencia matrimonial Crazy Casanova S.A.". Para ello tenemos que mantener una lista con las fichas de los clientes. Las fichas de los clientes van a tener los siguientes datos:

- Una cadena con el nombre y apellidos del cliente.
- Sexo del cliente (refiriéndonos a su género, claro está).
- Edad.
- Una cadena describiendo sus hobbies.

Ejemplos de fichas pueden ser estas:

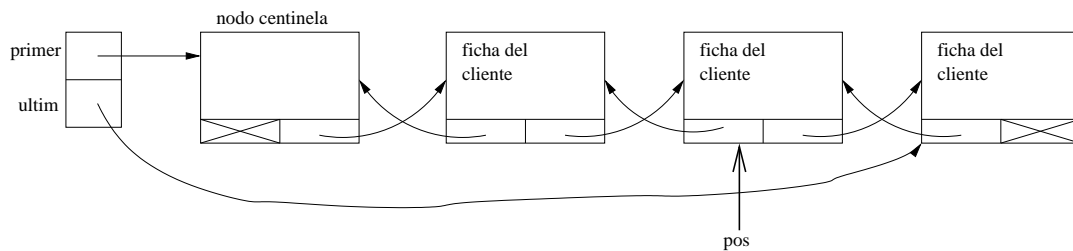
Nombre: Marieta Piruleta	Nombre: Marianico Cocoliso
Sexo: Mujer	Sexo: Hombre
Edad: 27	Edad: 35
Hobbies: Cine romántico	Hobbies: Cazar saltamontes

Nuestro cometido va a consistir en solucionar problemas bastante sencillos. Supongamos que dada la lista de clientes tenemos una posición `pos` que indica en cierto modo el nodo "activo" en un determinado momento, esto es, señala la ficha de un cliente. Existe una interfaz a la lista de clientes, que muestra por la pantalla el "cliente activo" (señalado por `pos`) en el momento actual.

Queremos conseguir que la/el secretaria/o de la agencia pueda cambiar la ficha de cliente mostrada por la pantalla a la ficha siguiente o a la anterior de la lista pulsando `AvPag` o `RePag` respectivamente.

Para ello, necesitamos definir una nueva estructura de datos lista similar a la vista en clase pero a la cual podamos aplicarle un recorrido hacia adelante o hacia atrás.

Un esquema gráfico de la nueva estructura lista utilizando memoria dinámica y variables enlazadas sería:



donde al igual que en la representación dinámica enlazada vista en clase, dada una posición  $p$ , el elemento correspondiente a dicha posición, está en el siguiente nodo al que apunta  $p$ , o sea, en  $p \rightarrow \text{sig}$ . Además, también existe un nodo centinela al principio de la lista. Esta estructura se conoce normalmente con el nombre de lista doblemente enlazada utilizando memoria dinámica.

Se pide:

- Dar la definición del tipo de datos lista necesario para montar este tipo de lista doblemente enlazada usando memoria dinámica en lenguaje C. (0.50 puntos)
- Con la definición de la estructura lista doblemente enlazada del apartado a). Escribir en lenguaje C la operación

```
posicion siguiente(lista *l, posicion pos)
```

que devuelve la posición siguiente a la posición  $pos$  en la lista  $l$ . (0.25 puntos)

- Con la definición de la estructura lista doblemente enlazada del apartado a). Escribir en lenguaje C la operación

```
posicion previa(lista *l, posicion pos)
```

que devuelve la posición previa a la posición  $pos$  en la lista  $l$ . (0.25 puntos)

- Utilizando las operaciones de los apartados b) y c), escribir una función

```
posicion situa_ficha(lista *l, posicion pos, int tecla)
```

que recibirá en el parámetro entero  $tecla$  un 0 si se ha pulsado AvPag o un 1 si se ha pulsado RePag. Basándose en este código de control, la función debe devolver la posición  $p$  siguiente a la posición  $pos$  de la lista si se

ha pulsado AvPag o devolver la posición previa si se ha pulsado RePag. (1 punto)

e) Suponer ahora que tenemos las siguientes operaciones definidas para nuestra estructura de datos lista doblemente enlazada:

- `lista *crearl()`
- `lista *insertar(lista *l, ficha *f, posicion pos)`
- `lista *borrar(lista *l, posicion pos)`
- `ficha *recuperar(lista *l, posicion pos)`
- *int* `vacial(lista *l)`
- `posicion fin(lista *l)`
- `posicion principio(lista *l)`
- `posicion siguiente(lista *l, posicion pos)`
- `posicion previa(lista *l, posicion pos)`
- *int* `compatible(lista *l, posicion pos1, posicion pos2)`  
→ esta operación devuelve verdadero si los clientes cuyas fichas son las correspondientes a las posiciones pos1 y pos2 de la lista son compatibles para emparejarlos. Falso en caso contrario.

Se pide escribir una función

```
lista *encuentra_compatibles(lista *l, posicion pos)
```

que dada una lista l y una posición pos de la lista, crea y devuelve una nueva lista donde están incluidas todas las fichas de aquellas posiciones de la lista l compatibles con la de la posición pos. (2 puntos)

**Solución:**

```
a) /* definimos por un lado la estructura ficha */
/* para las fichas de clientes */
typedef struct _ficha {
    char nombre[60];    /* cadena del nombre */
    char sexo[10];      /* cadena del sexo */
    int edad;           /* edad en años */
    char hobbies[100]; /* cadena de hobbies */
} ficha;

/* por otro lado la estructura para la lista */
/* doblemente enlazada de clientes */
typedef struct _lnode {
    ficha f;    /* la ficha del cliente */
    struct _lnode *siguiente;
                /* puntero al siguiente nodo */
    struct _lnode *previo;
                /* puntero al nodo previo */
} lnode;

typedef lnode *posicion;
                /* las posiciones son punteros */

typedef struct {
    posicion primero,ultimo;
    /* punteros a los nodos primero y ultimo */
} lista;
```

- b) La implementación de esta operación es idéntica a la vista en clase para una estructura lista simple.

```
posicion siguiente(lista *l, posicion pos) {
    return(pos->siguiente);
    /* devolvemos la posicion siguiente a pos */
}
```

- c) Esta operación consistirá en consultar y devolver el puntero al nodo previo al nodo al que está apuntando pos.

```

posicion previa(lista *l, posicion pos) {
    return(pos->previo);
    /* devolvemos la posicion previa a pos */
}

```

- d) En la función basta comprobar cual es el valor de la variable *tecla* y devolver la posición correspondiente.

```

posicion situa_ficha(lista *l, posicion pos, int tecla) {
    if (tecla==0) return(siguiente(l,pos));
    else if (tecla==1) return(previa(l,pos));
    else printf( "ERROR\n" );
    return(NULL);
}

```

La llamada desde el programa principal sería de esta manera:

```
pos = situa_ficha(l, pos, tecla);
```

- e) Consistirá en recorrer la lista de clientes e ir creando otra lista nueva con los clientes que sean compatibles con el de la posición *pos*:

```

lista *encuentra_compatibles(lista *l, posicion pos) {
    lista *nueva_l;
    posicion act;
    posicion new;
    ficha *fich;

    nueva_l = crearl();
    act = principio(l);
    new = principio(nueva_l);
    while (act != fin(l)) {
        if (compatible(l, pos, act) && (pos!=act)) {
            fich = recuperar(l, act);
            nueva_l = insertar(nueva_l, fich, new);
            new = siguiente(nueva_l, new);
        }
        act = siguiente(l, act);
    }
    return(nueva_l);
}

```

---

**Ejercicio: (1 punto)**

-Supón que eres el/la profesor/a de la asignatura, debido a ello tienes disponible una lista de fichas de alumnos ordenada alfabéticamente por apellidos. Las fichas contienen otros campos, como por ejemplo la nota de la asignatura, la cual ya está puesta. Resulta que ahora queremos saber cuál es el alumno que ha obtenido la 3ª mejor nota. ¿qué algoritmo puedes aplicar para saber cuál es el alumno que ha obtenido la 3ª mejor nota (no implementes el algoritmo, solo dí cual sería)? Razona tu respuesta. (1 punto)

**Solución:**

Como es una lista, podemos considerarla como un vector donde se guardarían en una determinada posición cada uno de los alumnos ordenados por su apellido. Podríamos optar, pues, por una representación vectorial de la lista y ahora el algoritmo más correcto (o adecuado) sería el desarrollado en clase para la búsqueda del  $k$ -ésimo menor elemento mediante Divide y Vencerás, pero en este caso aplicado a la búsqueda del  $k$ -ésimo mayor elemento. De hecho, este algoritmo debería aplicarse al campo de nota de cada ficha, puesto que buscamos a un alumno por su nota. Para esto se debe cambiar la implementación del algoritmo `partition` visto en clase de manera que en el subvector izquierdo queden los elementos mayores o iguales que el pivote y en el subvector derecho queden los elementos menores o iguales que el pivote: esto se conseguiría cambiando las siguientes líneas del algoritmo `partition`

```
    } while (A[j]>x)
```

por las siguientes líneas donde se han cambiado los signos de comparación:

```
    } while (A[i]<x)
```

del algoritmo de clase por

```
    } while (A[j]<x)
```

y

```
    } while (A[i]>x)
```

---

**Ejercicio: (1 punto)**

-Dado el algoritmo Quicksort estudiado en clase, del cual una codificación posible es:

```
void quicksort(int *A, int l, int r) {  
    int q;  
  
    if (l<r) {  
        q = partition(A, l, r);  
        quicksort(A, l, q);  
        quicksort(A, q+1, r);  
    }  
}
```

y considerando también el algoritmo de Inserción directa visto en clase, cuyo perfil era:

```
void insercion_directa(int *A, int l, int r)
```

Se pide: implementar la optimización propuesta en clase para el Quicksort mediante la cual se trataban los problemas inferiores a una cierta talla  $t$ , con un método de ordenación directo, en este caso deberá utilizarse el método de inserción directa. (1 punto)

**Solución:**

Hay que tener en cuenta que un nuevo parámetro del algoritmo sería la talla  $t$  a partir de la cual se ha de aplicar inserción directa. El algoritmo Quicksort quedaría de esta manera:

```
void quicksort(int *A, int l, int r, int t) {  
    int q;  
  
    if ((r-l+1)<t) insercion_directa(A,l,r);  
    else {  
        q = partition(A, l, r);  
        quicksort(A, l, q, t);  
        quicksort(A, q+1, r, t);  
    }  
}
```

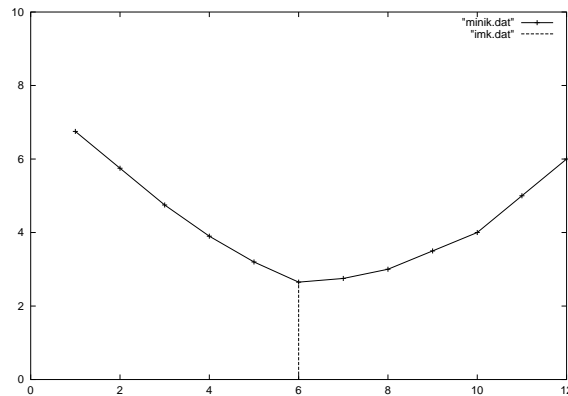
---

### Ejercicio: (3 puntos)

-Tenemos un vector  $A$  de  $n$  enteros positivos que se ajusta al perfil de una curva cóncava y se desea encontrar el mínimo de esta curva en este intervalo mediante un algoritmo divide y vencerás con coste logarítmico, esto es,  $O(\log n)$  (con esto se mejorará el algoritmo directo de búsqueda del mínimo de coste  $O(n)$ ).

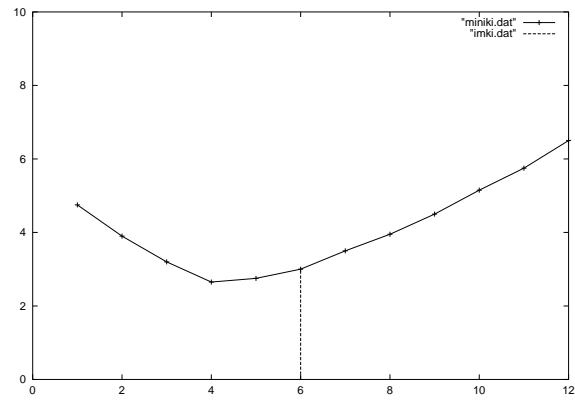
Para ello vamos a aprovechar el perfil de la curva. En las siguientes gráficas podemos observar la estrategia a seguir para realizar el algoritmo de búsqueda:

Como nuestro algoritmo va a ser divide y vencerás y estamos buscando dentro de un intervalo  $[l,r]$  del vector que representa la curva, pues vamos a obtener el punto medio  $k$  del intervalo, comprobaremos el valor de la curva para ese punto y a partir de ahí decidiremos si hay que seguir buscando en la parte del intervalo a la izquierda del punto medio o en la parte derecha. Fijándonos podemos distinguir 3 casos en la búsqueda:

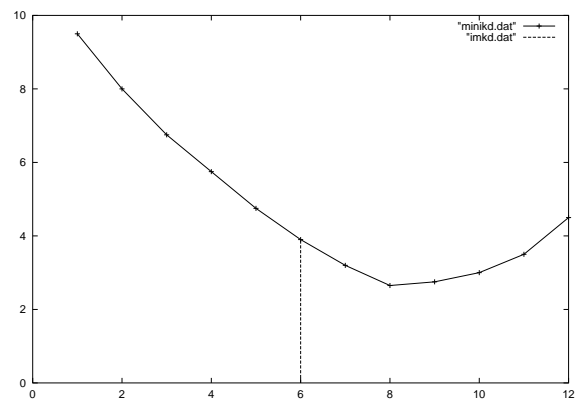


$A[k-1] > A[k] < A[k+1] \Rightarrow$  encontrado mínimo.





$A[k-1] < A[k] < A[k+1] \Rightarrow$  buscar mínimo en  $[l, k-1]$



$A[k-1] > A[k] > A[k+1] \Rightarrow$  buscar mínimo en  $[k+1, r]$

Se pide: escribir la codificación en lenguaje C del algoritmo. (3 puntos)

### Solución:

La codificación de este algoritmo Divide y Vencerás de manera recursiva sería:

```
int min_concava(int *A, int l,r) {
    int k;

    if (l<r) {
        k = (int) (l+r)/2;
        if ((A[k-1]>A[k]) && (A[k]<A[k+1])) min=A[k];
                                                    /* el minimo en k */
        else if ((A[k-1]<A[k]) && (A[k]<A[k+1]))
            min = min_concava(A,l,k-1);
                                                    /* buscamos en la parte izquierda */
        else min = min_concava(A,k+1,r);
                                                    /* buscamos en la parte derecha */
    }
    else min=A[l];
    return(min);
}
```

Una versión iterativa del algoritmo equivalente a la anterior sería:

```
int min_concava(int *A, int l,r) {
    int k;

    while (l<r) {
        k = (int) (l+r)/2;
        if ((A[k-1]>A[k]) && (A[k]<A[k+1])) return(A[k]);
                                                    /* el minimo en k */
        else if ((A[k-1]<A[k]) && (A[k]<A[k+1])) r = k-1;
                                                    /* buscamos en la parte izquierda */
        else l = k+1;
                                                    /* buscamos en la parte derecha */
    }
    return(A[l]);
}
```

# Estructuras de Datos y Algoritmos

**Junio 2002**

## **Examen de Teoría: SOLUCIONES**

Departamento de Sistemas Informáticos y Computación  
Escuela Politécnica Superior de Alcoy

### **Ejercicio : (1.5 puntos)**

- Dadas las siguientes definiciones de tipos y variables para representar árboles binarios de búsqueda:

```
typedef ... tipo_baseT;  
typedef struct snodo {  
    tipo_baseT clave;  
    struct snodo *hizq, *hder;  
} abb;
```

abb \*T;

Se pide escribir una versión recursiva del algoritmo visto en clase de teoría que obtiene el mínimo de un árbol binario de búsqueda.

```
abb *minimo(abb *T)
```

### **Solución:**

El esquema para la función que obtiene el mínimo de un árbol binario de búsqueda de manera recursiva es sencillo. Para buscar el mínimo siempre debíamos buscar por el hijo izquierdo del nodo actual en el caso de que lo tuviera, si no tenía hijo izquierdo es porque él era el mínimo.

El esquema recursivo consistirá en: si un nodo tiene hijo izquierdo, el problema se reduce a buscar el mínimo de su subárbol izquierdo, en el caso de que no tenga hijo izquierdo, él es el mínimo. El código de la función es:

```
abb *minimo(abb *T) {  
    abb *min=NULL;  
  
    if (T!=NULL)  
        if (T->hizq != NULL)  
            min = minimo(T->hizq);  
        else min = T;  
    return(min);  
}
```

---

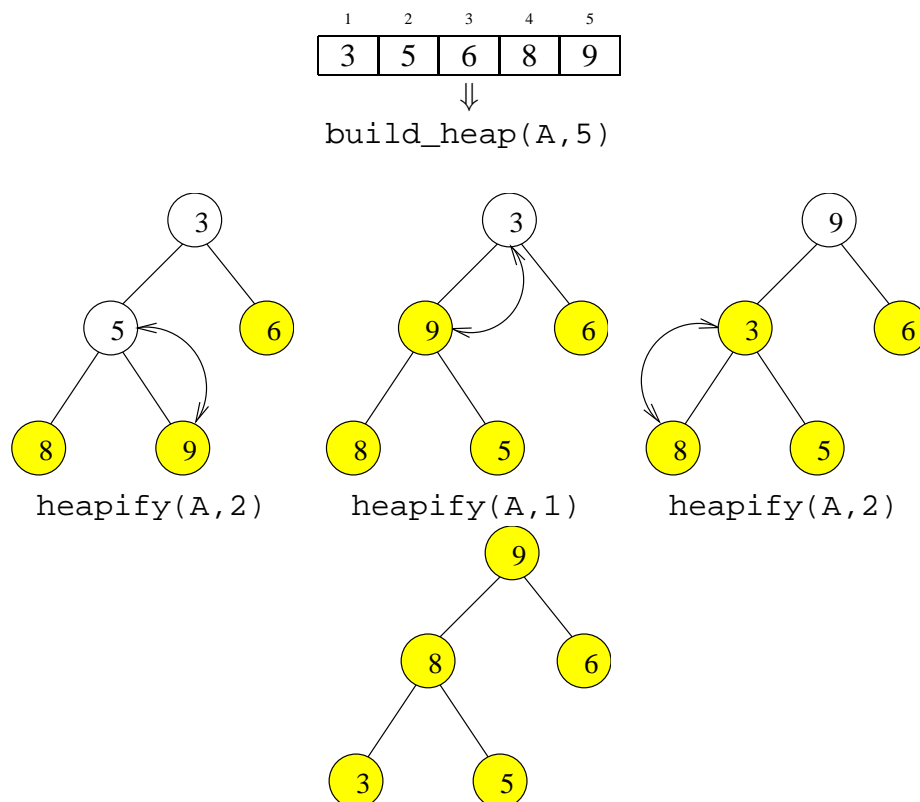
### Ejercicio : (2 puntos)

-Resolver las siguientes cuestiones relacionadas con el algoritmo `build_heap()`:

- a) ¿Cuál será el coste temporal del algoritmo `build_heap()` al aplicarlo sobre un vector  $A$  de talla  $n$ , si los elementos del vector están ordenados en orden creciente? Pon un pequeño ejemplo. (1 punto).
- b) ¿Y si estuvieran ordenados en orden decreciente? Pon un pequeño ejemplo. (1 punto).

### Solución:

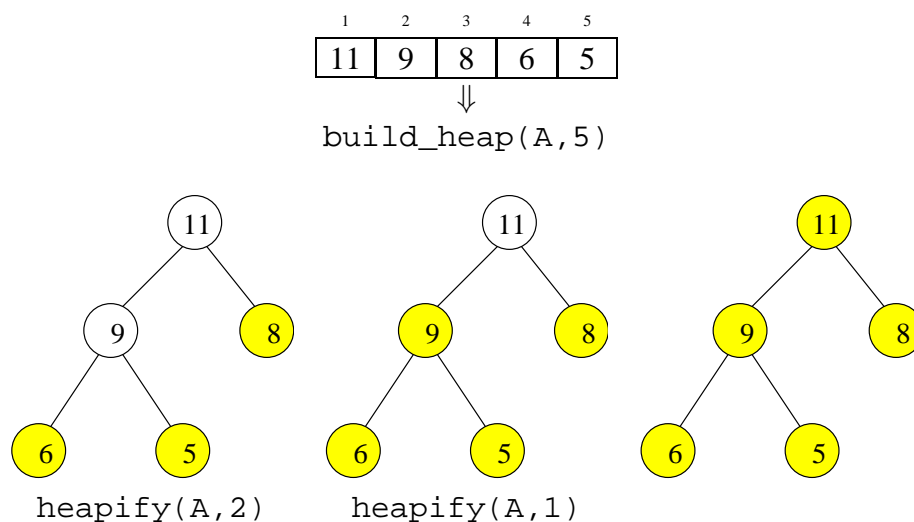
- a) Vemos primero un pequeño ejemplo de cómo se aplicaría `build_heap()` sobre un vector cuyos elementos están ordenados en orden creciente.



Como podemos ver en el ejemplo (y a partir de él, generalizar), construir un montículo a partir de un vector ordenado crecientemente tendrá un coste  $O(n)$ . Vamos a analizarlo con un poco más de detenimiento:

Como el vector inicial estaba ordenado crecientemente, a partir de él obtenemos directamente un árbol binario completo donde para cada nodo interno, la clave del nodo va a ser menor que la claves de todos los nodos que estén en sus subárboles. Por ello, cada vez que apliquemos `heapify()` sobre un nodo interno  $i$ , el coste de `heapify()` va a ser el máximo posible para ese nodo, esto es,  $O(h_i)$ , siendo  $h_i$  la altura del nodo  $i$ . Con esta situación llegamos al mismo análisis de costes visto en clase de teoría para `build_heap()`, por lo que aplicando el mismo desarrollo de clase llegamos a que el coste de `build_heap()` cuando el vector inicial está ordenado crecientemente es  $O(n)$ . Siendo  $n$  el número de elementos del vector y por tanto el número de nodos del montículo.

- b) Vemos ahora un ejemplo de aplicación de `build_heap()` sobre un vector cuyos elementos están ordenados en orden decreciente.



Como se ve en el ejemplo, y se intuye que va a ocurrir en cualquier caso en el que el vector esté ordenado decrecientemente, el coste temporal de `build_heap()` va a ser igual que para el caso anterior  $O(n)$ . Esto es debido a que aunque cualquier vector ordenado decrecientemente va a ser un montículo, tal y como se ha definido el algoritmo `build_heap()` (ver apuntes teoría) se ejecuta un bucle que recorre desde la posición  $\lfloor \frac{n}{2} \rfloor$  del vector hasta la posición 1 (la raíz) del montículo) aplicando `heapify()` a cada posición. Si nos fijamos, en un vector ordenado decrecientemente para cualquier elemento en una posición  $i$ , tendremos que los elementos en las posiciones  $2i$  y  $2i + 1$  serán menores que él, por lo que `heapify()` detectará que la condición de montículo se cumple y por ello cada llamada a

`heapify()` tendrá un coste  $O(1)$ . Así pues, tendremos que `build_heap()` realiza  $\frac{n}{2}$  llamadas de coste  $O(1)$ , con lo que el coste total de `build_heap()` será  $O(n)$ .

---

### Ejercicio : (1.5 puntos)

-Considerando un MF-set con una *representación de árboles mediante apuntes al padre*, y, suponiendo que utilizamos una estructura como la vista en clase de teoría donde el MF-set se representa con un vector en el que en cada posición  $i$  se almacena el índice del elemento que es padre del elemento  $i$ .

- a) ¿qué estrategia propones para hallar el número de clases de equivalencia (subconjuntos) existentes en el MF-set? Describe el algoritmo que aplicarías. (0.75 puntos)
- b) ¿Y qué algoritmo aplicarías si además quisieras conocer cuáles son los representantes de las clases de equivalencia del MF-set? Pon un pequeño ejemplo de MF-set representado mediante un vector e indica cuántas clases de equivalencia tiene y cuáles son sus representantes. (0.75 puntos).

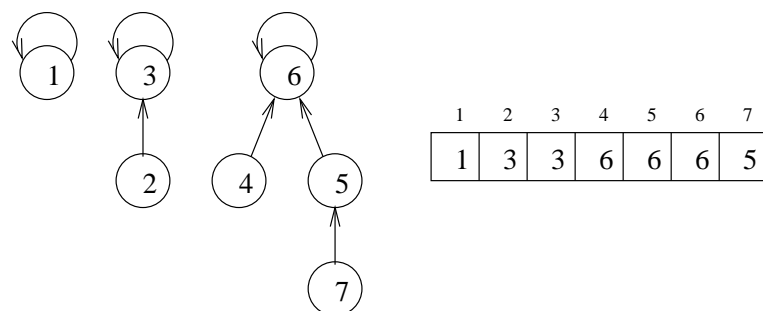
### Solución:

- a) La estrategia a seguir sería usar una variable como contador del número de subconjuntos que hay. Inicializamos esta variable a 0.

Y ahora hacemos un recorrido a lo largo del vector que representa al MF-set y comprobamos para cada componente cual es su padre, es decir, comprobamos el valor de cada componente. En el caso de que el padre de un elemento sea él mismo (esto es,  $V[i] == i$ ) entonces quiere decir que es un representante de una clase de equivalencia, con lo que habrá una clase de equivalencia más que las que llevamos contadas, así pues, incrementaremos en 1 el contador del número de subconjuntos y seguiremos recorriendo el resto del vector.

- b) El algoritmo a aplicar sería casi idéntico al descrito en el apartado a), realizar un recorrido sobre el vector que representa al MF-set y comprobar quien es el padre de cada elemento. En el caso de que el padre de un elemento sea él mismo (esto es,  $V[i] == i$ ) entonces quiere decir que es un representante de una clase de equivalencia y podríamos imprimirlo o guardarlo donde se necesite.

Un ejemplo de MF-set representado mediante un vector sería:



Que tiene 3 clases de equivalencia cuyos representantes son: 1, 3 y 6.

---

### Ejercicio : (3 puntos)

-Sea  $G = (V, E)$  un grafo no dirigido con  $n > 0$  nodos. Dado que la relación "ser dos nodos mutuamente conectados" es reflexiva, simétrica y transitiva, se pueden calcular las componentes conexas de un grafo no dirigido utilizando un MF-set de la siguiente forma:

1. Se inicializa el MF-set con  $n$  subconjuntos disjuntos: cada uno de los subconjuntos contiene un vértice del grafo. Esto indicará que, inicialmente, los vértices del grafo son inalcanzables entre sí.
2. Para cada arco  $(u, v) \in E$ , si  $u$  y  $v$  pertenecen a subconjuntos disjuntos, se unen los subconjuntos disjuntos a los que pertenecen  $u$  y  $v$ .

Cuando el algoritmo finalice, cada una de las componentes conexas que contenga el grafo  $G$  estará representada mediante un subconjunto disjunto. Y cada subconjunto disjunto contendrá aquellos vértices que pertenecen a la misma componente conexas.

a) Se pide escribir un algoritmo en lenguaje C

```
int conexo_grafo(grafo *G);
```

que siguiendo esta estrategia determine si un grafo no dirigido es conexo. La función debe devolver un 1 si el grafo es conexo y 0 si no lo es.

Supondremos que el grafo se representa mediante una matriz de adyacencia utilizando esta definición:



```
#define MAXVERT 1000

typedef struct{
    int talla;
    int A[MAXVERT][MAXVERT];
} grafo;
```

(2 puntos)

- b) Se pide también analizar cuál sería el coste temporal del algoritmo si las operaciones

```
void union(int i, int j, int *mfset);
```

y

```
int buscar(int i, int *mfset);
```

del MF-set se realizan aplicando los heurísticos *unión por rango* y *compresión de caminos*. (1 punto)

### Solución:

- a) Necesitaremos definir un vector de enteros que utilizaremos para representar el MF-set. Debido a que la representación usada para el grafo es mediante matriz de adyacencia, haremos un recorrido por toda la matriz, y para cada arista que conecte dos vértices  $i$  y  $j$  (es decir  $G \rightarrow A[i][j] == 1$ ) comprobaremos si ya están unidas las componentes conexas que los contienen, y si no es así, las uniremos en el MF-set. Por último, una vez el MF-set contiene los vértices agrupados en subconjuntos que representan las diferentes componentes conexas, debemos realizar un recorrido por el MF-set contando cuantas componentes conexas existen, en el caso en que haya un número de componentes diferente a 1, el grafo no será conexo.

Siguiendo la estrategia propuesta en el enunciado podemos escribir la siguiente función en C:

```

/* La funcion devuelve 1 si el grafo es conexo y */
/* 0 si no lo es. */

int conexo_grafo(grafo *G) {
    int mfset[MAXVERT];    /* El MFset. */
    int i,j;    /* Para recorrer las aristas del grafo. */
    int cont;    /* Contador de componentes conexas. */

    /* Inicializar MFset. */
    for (i=1;i<=G->talla;i++) mfset[i]=i;

    /* Comprobamos que vertices unen todas las aristas. */
    for (i=1;i<=G->talla;i++)
        for (j=1;j<=G->talla;j++)
            if (G->A[i][j] == 1) /* Hay arista entre i y j. */
                if (buscar(i,mfset) != buscar(j,mfset))
                    union(i,j,mfset);

    /* Comprobamos cuantas componentes conexas hay. */
    cont = 0;
    for (i=1;i<=G->talla;i++)
        if (mfset[i] == i) cont++;
    if (cont != 1) return(0); /* NO ES CONEXO. */
    else return(1); /* ES CONEXO. */
}

```

b) Considerando que las operaciones `buscar(i, mfset)` y `union(i, j, mfset)` tienen un coste constante debido a los heurísticos empleados con el MF-set, como el grafo  $G$  tiene  $n$  vértices, el coste de las diferentes operaciones que se realizan en el algoritmo es:

- Inicializar el MF-set tiene un coste  $O(n)$  (un bucle for).
- Comprobar a qué componente conexas (clase de equivalencia) pertenece cada vértice tiene un coste  $O(n^2)$  (dos bucles for anidados).
- Contar cuantas componentes conexas tiene el grafo tiene un coste  $O(n)$  (un bucle for).

Por tanto el coste es:  $O(n + n^2 + n) \in O(n^2)$ .

### Ejercicio : (2 puntos)

-Dados una cinta magnética de longitud  $L$  y  $n$  ficheros  $f_1, f_2, \dots, f_n$  de longitudes  $l_1, l_2, \dots, l_n$ , ¿Cuál sería la estrategia voraz de almacenamiento de ficheros que minimiza el tiempo medio de acceso?

Asumimos:

- Todos los ficheros caben dentro de la cinta magnética.
- Cuando se va a realizar cualquier acceso, la cinta está siempre al principio.
- Si el orden de almacenamiento de ficheros es  $i_1, i_2, \dots, i_n$ , el tiempo necesario para acceder al programa  $i_j$  será  $t_j = \sum_{k=1}^{j-1} l_{i_k}$ , siendo  $l_{i_k}$  la longitud del fichero  $i_k$ ,  $1 \leq k < j$ .
- Si la probabilidad de acceder a cualquier programa es la misma, el tiempo medio de acceso se define como  $TMA = \frac{1}{n} \sum_{j=1}^n t_j$ .

### Solución:

Observando la expresión para el tiempo medio de acceso:

$$TMA = \frac{1}{n} \sum_{j=1}^n t_j$$

con esto, minimizar el  $TMA$  significa que hay que minimizar  $t_j$ . La expresión para  $t_j$  es:

$$t_j = \sum_{k=1}^{j-1} l_{i_k}$$

si el orden de almacenamiento de ficheros es  $i_1, i_2, \dots, i_n$ . Si todos los ficheros  $i_k$ ,  $1 \leq k < j$ , que están almacenados delante del fichero  $i_j$  tienen un tamaño menor que  $i_j$ , entonces el tiempo de acceso al fichero  $i_j$  será el menor posible, esto es,  $t_j$  será el menor posible. Así pues, la estrategia voraz para almacenar los ficheros será almacenarlos en orden no decreciente de tamaño. Para ello, los ordenaremos de menor a mayor por tamaño y los almacenaremos en ese mismo orden, con esto conseguiremos reducir el tiempo de acceso  $t_j$  para cada fichero  $i_j$ ,  $1 \leq j \leq n$ , con lo que se reducirá el  $TMA$ .

# Estructuras de Datos y Algoritmos

Septiembre 2002

## Examen de Teoría: SOLUCIONES

Departamento de Sistemas Informáticos y Computación  
Escuela Politécnica Superior de Alcoy

### SEMESTRE A

#### Ejercicio : (2 puntos)

-Suponiendo que trabajamos con una lista de enteros con una representación enlazada de listas con variable dinámica especificada mediante la siguiente definición de tipos:

```
typedef struct _lnodo {  
    int e;                                /* Un elemento. */  
    struct _lnodo *sig;                  /* Puntero al siguiente nodo. */  
} lnodo  
  
typedef lnodo *posicion;  
                /* Cada posicion es un puntero. */  
  
typedef struct {  
    posicion primero, ultimo;          /* primer y ultimo nodos. */  
} lista;
```

Escribe una función en lenguaje C con el perfil

```
lista *intercambia(lista *L, posicion pos1, posicion pos2)
```

que devuelva la lista L modificada de manera que haya intercambiado los elementos correspondientes a las posiciones pos1 y pos2.

**Solución:**

```
lista *intercambia(lista *L, posicion pos1, posicion pos2) {  
    int aux;  
  
    aux = pos1->e;  
    pos1->e = pos2->e;  
    pos2->e = aux;  
    return (L);  
}
```

---

**Ejercicio : (5 puntos)**

-En el programa de radio *La Gramola*, disponen de una gramola gigantesca que almacena centenares de CDs de música. Los CDs almacenados se encuentran organizados en una lista ordenada, donde cada posición de la lista identifica a un CD.

Existe además un sistema que permite seleccionar un determinado CD por la posición que ocupa en la lista y que acciona un brazo automático que carga el CD seleccionado para ser escuchado por antena.

Sin embargo, la dirección del programa quiere optimizar el tiempo de selección de un CD, esto es, el tiempo que tarda el brazo en buscar un CD y cargarlo en el lector. Para ello se desea hacer más accesibles para el brazo los CDs más solicitados, es decir, ponerlos más cerca de la unidad de lectura de CDs. Por esto, nos han pedido que contemos el número de veces que se solicita cada CD y reorganizar la lista en consecuencia.

Se pide:

a) Escribe la definición de tipos necesaria para usar una estructura de datos lista con representación vectorial, de manera que esta estructura permita llevar un contador para cada elemento de la lista y así poder conocer el número de veces que cada CD ha sido solicitado. El identificador del CD es un entero simple, dentro del cual se encuentra codificada la posición física de éste en la gramola. (1.25 puntos)

b) Reescribe la función de selección de un elemento de la lista (CD)

```
int recuperar(lista *l, posicion p)
```

de forma que quede reflejado que el CD de la posición p ha sido solicitado una vez más. (1.25 puntos)

c) Escribe una función con el siguiente perfil

```
lista *reordena(lista *l)
```

que reordene la lista de CDs basándose en los contadores de solicitud para cada CD, de manera que la lista quede ordenada con los CDs más solicitados en las primeras posiciones y los menos solicitados en las últimas posiciones. (2.5 puntos)

### Solución:

a) Bastará con incluir dentro de la estructura para la lista un vector de contadores, de esta manera:

```
#define maxL ... /* Talla maxima del vector. */

typedef int posicion;
/* Cada posicion se referencia con un entero. */

typedef struct {
    int v[maxL]; /* Vector para elementos. */
    int cont[maxL]; /* Vector para contadores. */
    posicion ultimo; /* Posicion del ultimo elemento. */
} lista;
```

b) Tan sólo hay que incrementar el contador de solicitudes del CD de la posición p.

```
int recuperar(lista *l, posicion p) {
    l->cont[p]++;
    /* Incrementamos el contador de este elemento. */
    return(l->v[p]);
    /* Devolvemos el elemento que hay en la posicion p. */
}
```

c) Aplicamos un método de ordenación sobre el vector de contadores l->cont. Por ejemplo, aplicamos el de inserción directa. Hay que tener en cuenta que

cuando se cambie de posición algún contador de solicitudes, también debe cambiarse el identificador del CD correspondiente.

```

lista *reordena(lista *l){
    int i, j, aux_cont, aux_v;

    for (i=1; i<=l->ultimo; i++) {
        aux_cont = l->cont[i];
        aux_v = l->v[i];
        j = i;
        while ((j>0) && (l->cont[j-1]<aux_cont)) {
            l->cont[j] = l->cont[j-1];
            l->v[j] = l->v[j-1];
            j--;
        }
        l->cont[j] = aux_cont;
        l->v[j] = aux_v;
    }
    return (l);
}

```

---

### Ejercicio : (3 puntos)

-Sea  $A[1..r]$  un vector **ordenado** de menor a mayor de enteros diferentes (positivos).

Diseña un algoritmo Divide y Vencerás que encuentre un  $i$  tal que  $l \leq i \leq r$  y  $A[i]=i$ , siempre que este  $i$  exista; en caso contrario, debe devolver -1. El perfil de la función a diseñar es:

```
int busca_igual(int *A, int l, int r)
```

Por ejemplo, para el siguiente vector:

1	l+1	...	8	...	r-1	r
2	4	...	8	...	11	14

La función debería devolver 8.

El algoritmo debe tener un coste  $O(\log n)$ .

NOTA: puedes basarte en el algoritmo de búsqueda binaria (dicotómica) en un vector ordenado.

### Solución:

La idea básica es ir cercando el subvector de búsqueda donde puede estar ese posible elemento tal que  $i=A[i]$ . Buscaremos el elemento en la posición media del vector actual, si no está allí, entonces habrá que seguir buscando en la mitad izquierda del vector o en la derecha. Para saber en qué parte debemos buscar, podemos fijarnos que si  $i>A[i]$ , entonces el encontrar un  $i=A[i]$  sólo podrá ocurrir en la mitad izquierda del vector, del mismo modo si  $i<A[i]$ , deberemos buscar sólo en la mitad derecha del vector. Aplicando este razonamiento de manera sucesiva, si existe un  $i$  tal que  $i=A[i]$ , lo encontraremos, en caso contrario, devolveremos -1.

La solución de manera recursiva sería:

```
int busca_igual(int *A, int l, int r) {
    int i;

    if (l == r)
        if (l == A[l]) return(l);
        else return(-1);
    else {
        i = (int) (l+r)/2;
        if (i == A[i]) return(i);
        else if (i < A[i]) return(busca_igual(A,l,i-1));
        else return(busca_igual(A,i+1,r));
    }
}
```

La solución de manera iterativa sería:

```
int busca_igual(int *A, int l, int r) {
    int i;

    while (l<r) {
        i = (int) (l+r)/2;
        if (i == A[i]) return(i);
        else if (i < A[i]) l = i;
        else r = i;
    }

    if (l == A[l]) return(l)
    else return(-1);
}
```

---



## SEMESTRE B

### Ejercicio : (3 puntos)

-Dada la siguiente definición de tipos:

```
typedef struct snodo {  
    int info;  
    struct snodo *izq, *der;  
} tnodo;
```

```
typedef tnodo* arbol;
```

Escribid una función con el perfil

```
int esDeBusqueda(arbol T)
```

de orden lineal que devuelva 1 si T es un árbol binario de búsqueda y 0 en caso contrario.

### Solución:

```
int esDeBusqueda(arbol T) {  
    int nodoIzqEsMenor, nodoDerEsMayor;  
  
    if (T!=NULL) {  
        if ((T->izq != NULL) && (T->izq->info > T->info))  
            nodoIzqEsMenor = 0;  
        else nodoIzqEsMenor = 1;  
        if ((T->der != NULL) && (T->der->info < T->info))  
            nodoDerEsMayor = 0;  
        else nodoDerEsMayor = 1;  
  
        if (nodoIzqEsMenor && nodoDerEsMayor &&  
            esDeBusqueda(T->izq) && esDeBusqueda(T->der))  
            return(1);  
        else return(0);  
    }  
    else return(1);  
}
```

---

**Ejercicio : (4 puntos)**

-Escribe una función recursiva tal que dado un *max-heap* de enteros de talla  $n$ , ( $n > 0$ ) y un entero  $k$ , escriba por pantalla los enteros del heap mayores que  $k$ . El perfil de la función tiene que ser

```
int escribe(int *heap, int nodo, int n, int k)
```

donde *nodo* es el nodo que se procesa en cada llamada recursiva. Si hay  $m$  valores mayores que  $k$ , ( $m \leq n$ ), entonces el algoritmo tiene que tener un coste temporal  $O(m)$ .

Especifica cual tendría que ser la llamada inicial a la función desde el programa principal.

**Solución:**

```
int escribe(int *heap, int nodo, int n, int k) {  
    if (heap[nodo] > k) {  
        printf("%d\n", heap[nodo]);  
        if ((2*nodo) <= n)  
            escribe(heap, 2*nodo, n, k);  
        if ((2*nodo+1) <= n)  
            escribe(heap, 2*nodo+1, n, k);  
    }  
}
```

La llamada inicial a la función sería:

```
escribe(heap, 1, n, k)
```

---

### Ejercicio : (3 puntos)

-Considerando la siguiente definición de grafo representado con listas de adyacencia:

```
#define N ...

typedef struct snodo {
    int nodo;
    struct snodo *sig;
} tnode;

typedef struct snodo *lista;

lista G[n];
```

Suponiendo que  $G$  representa un grafo dirigido, y que los nodos de  $G$  están numerados de 0 a  $N - 1$ .

Escribe una función

```
int alcanzable(lista *G, int a, int b)
```

basada en el recorrido en profundidad del grafo y en la representación dada, tal que dados una par de nodos  $a, b$ ,  $a \neq b$ , indique si hay un camino en el grafo  $G$  que vaya de  $a$  a  $b$ .

Más exactamente, la función tendrá que devolver 0 en el caso que  $b$  no sea alcanzable desde  $a$  por ningún camino, y 1 en el caso que sí lo sea.

### Solución:

Supondremos una variable global vector de enteros `visitado`, indexado por los nodos del grafo, cuyas componentes se habrán inicializado a 0. La función `alcanzable(G, a, b)` es un recorrido en profundidad del grafo que devuelve los siguientes valores:

1. si  $b$  es adyacente a  $a$ , o recursivamente es alcanzable desde alguno adyacente a  $a$ , devuelve 1.
2. si  $a$  no tiene adyacentes, o  $b$  no es alcanzable recursivamente desde ningún adyacente a  $a$ , devuelve 0.

```

int alcanzable(lista *G, int a, int b) {
    lista aux;
    int nodoaux, r;

    visitado[a] = 1;
    aux = G[a];
    while (aux!=NULL) {
        nodoaux = aux->nodo;
        if (nodoaux==b) return(1);    /* CASO 1 */
        else if (visitado[nodoaux]==0) {
            r = alcanzable(G,nodoaux,b);
            if (r==1) return(1);    /* CASO 1 */
            else aux = aux->sig
        }
    }
    return(0);    /* aux==NULL, CASO 2 */
}

```

# Bibliografía

- [Aho] A.V. Aho, J.E. Hopcroft, J.D. Ullman, "*Estructuras de datos y algoritmos*", Addison-Wesley, 1988.
- [Aoe] J. Aoe, "*An efficient digital search algorithm by using a double-array structure*", IEEE Transactions on Software Engineering, Vol. 15, 9, pp. 1066-1077, 1989.
- [Brassard] G. Brassard, P. Bratley, "*Algorítmica. Concepción y análisis*", Masson, 1990.
- [Cormen] T. H. Cormen, C. E. Leiserson, R. L. Rivest, "*Introduction to algorithms*", MIT Press, 1990.
- [Ferri] F. Ferri, J. Albert, G. Martin, "*Introducció a l'anàlisi i disseny d'algorismes*". Universitat de València, 1998.
- [Horowitz] E. Horowitz, S. Sahni, "*Computer algorithms*" Computer Science Press, 1997.
- [Kernighan] B.W. Kernighan, D. M. Ritchie, "*El lenguaje de programación C*", Prentice-Hall, 1991.
- [Sedgewick] R. Sedgewick, "*Algorithms in C. Fundamentals. Data structures. Sorting. Searching*", Third edition. Addison-Wesley, 1998.
- [Weiss] M. A. Weiss, "*Estructuras de datos y algoritmos*", Addison-Wesley, 1995.
- [Wirth] N. Wirth., "*Algoritmos + Estructuras de Datos = Programas*", Ed. Castillo, 1980.