Estructuras de Datos y Algoritmos

Tipos Abstractos De Datos

Introducción

Tipo de Datos(TD): conjunto de valores que una variable puede tomar.
Ej: un número entero es de tipo $entero$).

- Representación interna: agrupaciones de bits.
- Necesidad de usar TDs:
  - Elegir representación interna óptima.
  - Aprovechar las características de un TD.
    Ej: operaciones aritméticas.
- TD simples, elementales o primitivos: enteros, reales, booleanos y caracteres.
- **Operaciones** asociadas simples.
  Ej:multiplicación.

Introducción (II)

Trabajar con matrices de números $\longrightarrow$ definir un TD matriz y operaciones asociadas (Ej: multiplicación).

Tipo Abstracto de Datos(TAD): modelo matemático con una serie de operaciones definidas (procedimientos).

- TAD = generalización de los TD simples.
- Procedimientos = generalizaciones de operaciones primitivas.
- La **encapsulación** o **abstracción** permite:
  - Localizar la definición del TAD y sus operaciones en un lugar determinado.
    $\Rightarrow$ Librerías específicas para un TAD.
  - Cambiar la implementación según la adecuación al problema.
  - Facilidad en la depuración de código.
  - Mayor orden y estructuración en los programas.

Introducción (III)

Ej: TAD matriz $\longrightarrow$ implementación tradicional o para matrices dispersas.

- **Implementación** de un TAD: traducir en instrucciones de un lenguaje de programación la declaración de ese tipo.
  Un procedimiento por cada operación definida en el TAD.
- Una implementación elige una **estructura de datos** (ED) para el TAD.
- ED construída a partir de TDs simples + dispositivos de estructuración del lenguaje de programación
  (vectores o registros).

Ejemplo

Definir TAD Matriz de Enteros:

- MATRIZ DE ENTEROS≡ *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 de la matriz se identifica por su fila y columna.*

Representación lógica:

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

$M[1,2] = 4$.

Ejemplo (II)

- Operaciones:
  - `Suma_Mat(A,B:` MATRIZ`)` devuelve `C:` MATRIZ
    ≡ 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 . . . .
  - `Inversa_Mat(A:` MATRIZ`)` devuelve `C:` MATRIZ
    ≡ en el caso de que la matriz `A` posea inversa, ésta se calcula . . . .

Ejemplo: Implementación

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;


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];
}
```

Estructuras de Datos y Algoritmos

Gestión Dinámica de Memoria. Punteros

De esa manera son válidas las siguientes operaciones:

En el código	En ejecución
`x=5;`	`x` 5
`x=x+2;`	`x` 7

Para estructuras más complejas la situación es la misma:

En el código	En ejecución	Instrucción válida
`int v[3];`	`v`	`v[0]=1;`
`struct {float r,i;} im;`	`im`	`im.r=5.0;` `im.i=3.0;`

Gestión dinámica de memoria. Punteros

Definición y creación de una variable en tiempo de compilación

Durante la codificación y compilación se define el tipo de la variable y su tamaño.

Ejemplo

La definición

```
int x;
```

hace que cuando comienza la ejecución se reserve un espacio de memoria que puede almacenar un entero y al cual se puede acceder con la etiqueta `x`.

`x` □

Punteros

La definición

```
int *px;
```

define 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.

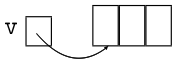
- El operador `&` permite recuperar la dirección de una variable que ya existe en memoria.
- Un puntero puede tomar un valor especial `NULL` para indicar que no apunta a ningún lugar accesible.

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

La definición de un vector

`int v[3];`

puede interpretarse de la siguiente manera:



La variable v se puede ver como un puntero a una o más posiciones de memoria consecutivas. De esa manera son posibles operaciones como las siguientes:

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

Punteros y vectores

Una característica importante de los punteros en C es que pueden apuntar a cualquier posición en una estructura más compleja como un vector o una `struct`. De esa manera se permiten operaciones como las siguientes:

En el código	En ejecución	Comentario
`int *px, v[3];`	px □ v □□□	px y los elementos de v están indefinidos.
`px=&v[0];`	px → v □□□	px apunta a la primera posición del vector.
`*px=1;`	px → v 1 □□	La primera posición del vector se modifica a través de px.
`px=px+1;`	px → v 1 □□	px apunta a la segunda posición del vector.
`v[1]=v[0]+*(px-1);`	px → v 1 2 □	?

Definición y creación de una variable en tiempo de ejecución

Durante la codificación y compilación se define el tipo de la variable. Durante el tiempo de ejecución se reserva la cantidad de memoria necesaria.

⇒ Reserva de memoria: las funciones `malloc` y `calloc` permiten reservar memoria dinámicamente. La función `malloc` no inicializa la memoria reservada, mientras que `calloc` sí.

⇒ Liberación de memoria: la función `free` permite liberar memoria previamente reservada.

⇒ Talla de un tipo: `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);
```

`malloc` recibe como parámetro el número de bytes que queremos reservar y devuelve un puntero sin tipo.
`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.

La sintaxis de `free` es:

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

Recibe como parámetro un puntero a una zona válida de memoria.

La sintaxis de `sizeof` es:

```
sizeof(type);
```

Recibe como parámetro un tipo.

Sea la siguiente definición:

```
int *v;
```

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

Sean las siguientes definiciones:

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

Los siguientes segmentos de código permiten definir una matriz cuadrada.

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

```
v=(int **)malloc(n*sizeof(int *)); /* Definimos un vector de punteros. */
pv=(int *)malloc(n*n*sizeof(int)); /* Definimos un vector de enteros. */
for (i=0; i<n*n; i++)
  pv[i]=0;                         /* Inicializamos la matriz. */
for (i=0; i<n; i++)   /* Cada componente de v apunta a n posiciones */
  v[i]=&pv[i*n];                   /* consecutivas de pv. */
```

Sean las siguientes definiciones:

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

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

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

En el código	Comentario
`tt=(tupla**)malloc(n*sizeof(tupla*));`	Reservamos un vector de punteros a tupla.
`t=(tupla*)malloc(n*sizeof(tupla));`	Reservamos un vector de tuplas.

`for (i=0;i<n;i++) {`
`  tt[i]=&t[i];`
`  tt[i]->a=0;`
`  tt[i]->b=0;`
`}`
	Después enlazamos cada elemento de `tt` con un elemento de `t`.

Paso de vectores a funciones

- Hay que pasarlos mediante un puntero a un vector.
- Realizar una copia local del vector cada vez es demasiado costoso.
- El vector se pasa por referencia.

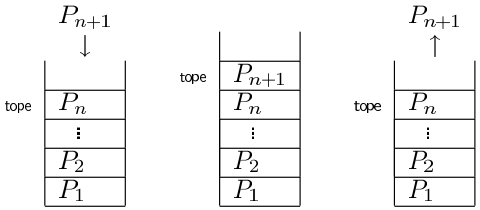
```
void f(int *v) { ... }
...
int main() {
  int p[n];
  ...
  f(p);
  ...
}
```

Estructuras de Datos y Algoritmos

Estructuras de Datos Lineales

Pilas. Representación vectorial y enlazada

Definición. Una pila es 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* (LIFO). Se accede a los elementos únicamente por el *tope* de la pila.

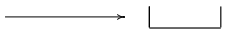


Aplicaciones

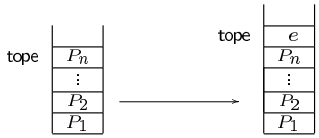
- Evaluación de expresiones aritméticas.
- Gestión de la recursión.

Operaciones sobre pilas

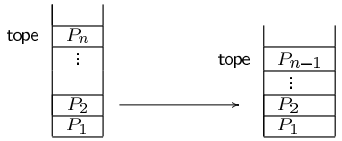
- `crearp()`: crea una pila vacía.



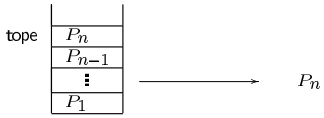
- `apilar(p,e)`: añade el elemento e a la pila p. (*push* en inglés)



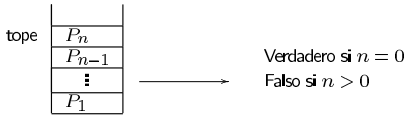
- `desapilar(p)`: elimina el elemento del tope de la pila p. (*pop* en inglés)



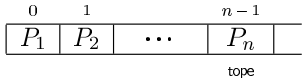
- `tope(p)`: consulta el tope de la pila p. (*top* en inglés)



- `vaciap(p)`: consulta si la pila p está vacía.



Representación vectorial de pilas

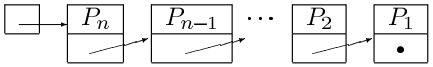


Definición de tipo

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

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

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

  p = (pila *) malloc(sizeof(pila)); /* Reservamos memoria para la pila. */
  p->tope = -1;                  /* Inicializamos el marcador 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. */
    tratarPilaLlena(); /* Si no cabe hacemos un tratamiento de error. */
  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 marcador al tope. */
  return(p);          /* Devolvemos un puntero a la pila modificada. */
}
```

Representación enlazada de pilas con variable dinámica



Definición de tipo

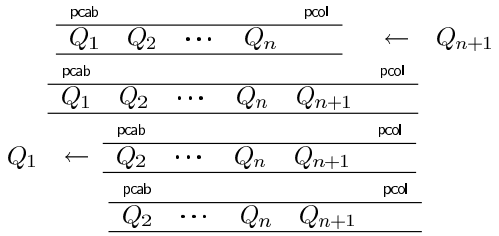
```
typedef struct _pnodo {
  int e;        /* Variable para almacenar un elemento de la pila. */
  struct _pnodo *sig; /* Puntero al siguiente nodo que contiene un elemento. */
} pnodo;     /* Tipo nodo. Cada nodo contiene un elemento de la pila. */
typedef pnodo pila;


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 */
}
```

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

```
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) {
  pnodo *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. */
```

Colas. Representación vectorial y enlazada

Definición. Una cola es 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* (FIFO). Los elementos se introducen por la *cola* y se extraen por la *cabeza*.

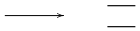


Aplicaciones

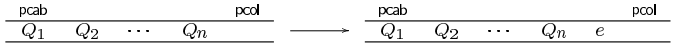
- Cola de procesos de acceso a un recurso.
- Cola de impresión.

Operaciones sobre colas

- `crearq()`: crea una cola vacía.

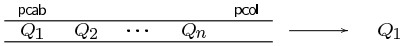


- `encolar(q,e)`: añade el elemento e a la cola q.

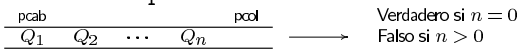


- `desencolar(q)`: elimina el elemento de la cabeza de q.

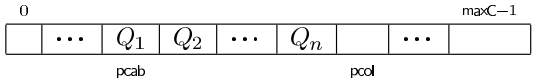
- `cabeza(q)`: consulta el primer elemento de la cola q.



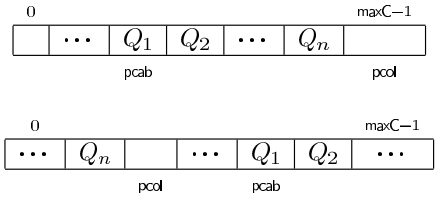
- `vaciaq(q)`: consulta si la cola q está vacía.

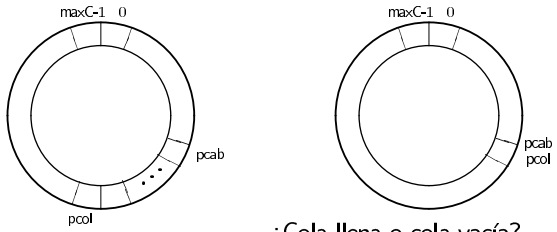


Representación vectorial de colas



Posibles situaciones





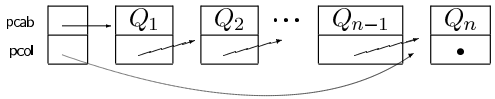
¿Cola llena o cola vacía?

Definición de tipo

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

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

```
cola *crearq() {
  cola *q;

  q = (cola *) malloc(sizeof(cola)); /* Reservamos memoria para la cola. */
  q->pcab = 0;              /* Inicializamos el marcador a la cabeza. */
  q->pcol = 0;                /* Inicializamos el marcador a la cola. */
  q->talla = 0;                          /* Inicializamos la talla. */
  return(q);              /* Devolvemos un puntero a la cola creada. */
}

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

Representación enlazada de colas con variable dinámica



Definición de tipo

```
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;
```

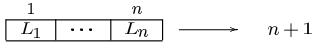
```
cola *desencolar(cola *q) {
  q->pcab = (q->pcab + 1) %  maxC; /* Avanzamos el marcador de cabeza. */
  q->talla = q->talla - 1;               /* Decrementamos la talla. */
  return(q);          /* Devolvemos un puntero a la cola modificada. */
}

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.*/
}
```

```
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 *qaux;

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

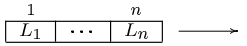
```
cola *desencolar(cola *q) {
  cnodo *qaux;

  qaux = 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(qaux);           /* 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. */
```

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

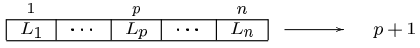
- `fin(l)`: devuelve la posición que sigue a la última en la lista `l`.



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



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

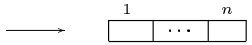


Listas. Representación vectorial y enlazada

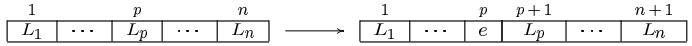
Definición. Una lista és una estructura de datos formada por una secuencia de objetos. Cada objeto se referencia mediante la posición que ocupa en la secuencia.

Operaciones sobre listas

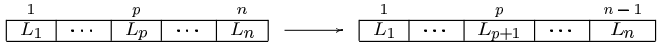
- `crearl()`: crea una lista vacía.



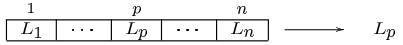
- `insertar(l,e,p)`: inserta `e` en la posición `p` de la lista `l`. Los elementos que están a partir de la posición `p` se desplazan una posición.



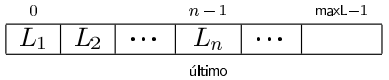
- `borrar(l,p)`: borra el elemento de la posición `p` de la lista `l`.



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



Representación vectorial de listas



Definición de tipo

```
#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;
```

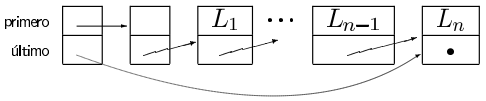
```
lista *crearl() {
  lista *l

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

```
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. */
}
```

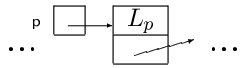
```
lista *borrar(lista *l, posicion p) {
  posicion i;

  for (i=p; i<l->ultimo; i++) /* Desplazamos los elementos del vector. */
    l->v[i] = l->v[i+1];
  l->ultimo = l->ultimo - 1;  /* Decrementamos el marcador al ultimo. */
  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. */
}
```

Representación enlazada de listas con variable dinámica

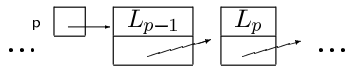
Usamos un nodo centinela al principio de la lista para optimizar las operaciones de actualización.



- Primera posibilidad: dada una posición p, el elemento L_p está en el nodo apuntado por p.



- Segunda posiblidad: dada una posición p, el elemento L_p está en el nodo apuntado por p->sig.



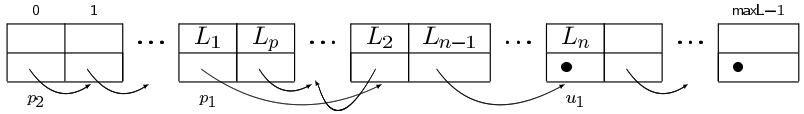
Definición de tipo

```
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;


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. */
}
```

```
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. */
}
```

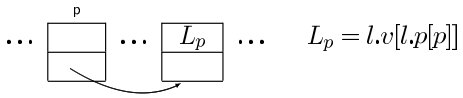
```
lista *insertar(lista *l, int e, posicion p) {
  posicion q;

  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. */

  if (p == l->ultimo) /* Si el nodo insertado ha pasaso a ser el 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. */
}
```

Representación enlazada de listas con variable estática



Definición de tipo

```
#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 execucion. */
  posicion p1,                  /* Marcador al principio de la lista. */
           u1,                        /* Marcador al fin de la lista. */
           p2;  /* Marcador al principio de la lista de nodos vacios. */
} lista;
```



$L_p = l.v[l.p[p]]$

```
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. */
  for (i=1; i<maxL-1; i++) /* Construimos la lista de nodos vacios. */
    l->p[i] = i+1;
  l->p[maxL-1] = -1; /* El ultimo nodo vacio no senyala a ningun lugar. */
  return(l);         /* Devolvemos un puntero a lista construida. */
}
```

```
lista *insertar(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 marcador 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. */
    if (p == l->u1) /* Si el nodo que hemos insertado pasa a ser el ultimo, */
      l->u1 = q;                          /* actualizamos el marcador u1. */
  return(l);          /* Devolvemos un puntero a la lista modificada. */
}
```

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

  if (l->p[p] == l->u1)       /* Si el nodo que borramos es el ultimo, */
    l->u1 = p;                                    /* actualizamos u1. */
  q = l->p[p];                /* Dejamos q senyalando al nodo a borrar. */
  l->p[p] = l->p[q];    /* El sucesor del nodo senyalado 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);
}
```

Estructuras de Datos y Algoritmos

Divide y vencerás

Esquema general de "Divide y Vencerás"

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

- Esquema recursivo.
- Coste en la parte de división del problema en subproblemas
  $+$
  Coste en la parte de combinación de resultados.
- Esquema eficiente cuando los *subproblemas* tienen una talla lo más parecida posible.

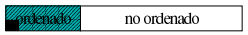
Algoritmos de ordenación

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

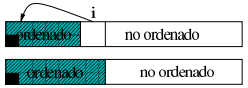
Ordenación por Inserción directa (InsertSort)

Estrategia:

1. 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 la posición que le corresponda por orden de la parte ordenada de tal forma que se mantenga la ordenación en dicha parte.

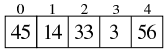


Algoritmo:	Inserción directa
Argumentos:	A: vector $A[l,\ldots,r]$,
	l: índice de la primera posición del vector,
	r: índice de la última posición del vector

```
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;
 }
}
```

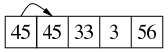
llamada a la función: insercion_directa(A,0,4)

vector inicial A:

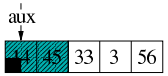


primera iteración (i=1,aux=14)

$j = 1$

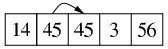


$j = 0$

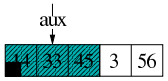


segunda iteración (i=2,aux=33)

$j = 2$

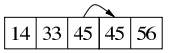


$j = 1$

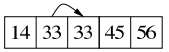


tercera iteración (i=3,aux=3)

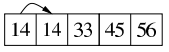
$j = 3$



$j = 2$



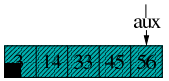
$j = 1$



$j = 0$

cuarta iteración (i=4,aux=56)

$j = 4$



Análisis de la eficiencia

- Caso peor

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

- Caso mejor

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

- Caso medio

insertar en pos i: $\quad c_i = \frac{1}{i}\left(2(i-1) + \sum_{k=1}^{i-2} k\right) = \frac{(i-1)(i+2)}{2i} = \frac{i+1}{2} - \frac{1}{i}$

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

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

Coste temporal del algoritmo "inserción directa" $\Longrightarrow \Omega(n)\ O(n^2)$

Ordenación por Selección directa (SelectSort)

Funcionamiento:

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

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.

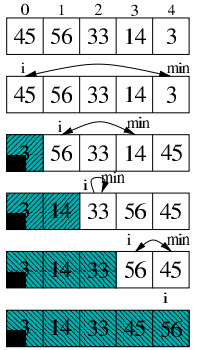
2. A continuación, se *selecciona* el valor mínimo almacenado en la *subsecuencia* $A[2, \ldots, N]$ y se coloca en la segunda posición; es decir, asignamos el 2-ésimo menor elemento a la posición 2.

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.

Algoritmo:	Selección directa
Argumentos:	A: vector $A[l,\ldots,r]$,
	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 de funcionamiento:



Análisis de la eficiencia

Número de comparaciones:

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

Coste temporal del algoritmo "selección directa" $\Longrightarrow \Theta(n^2)$

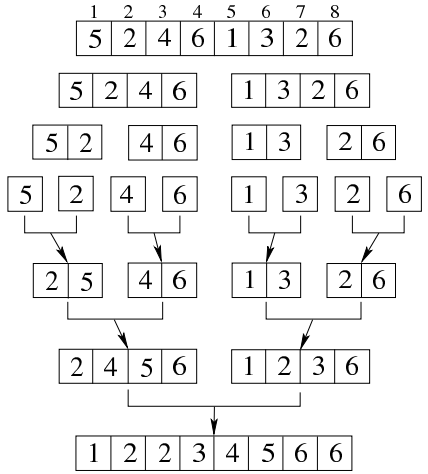
Ordenación por mezcla o fusión: Mergesort

Estrategia:

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

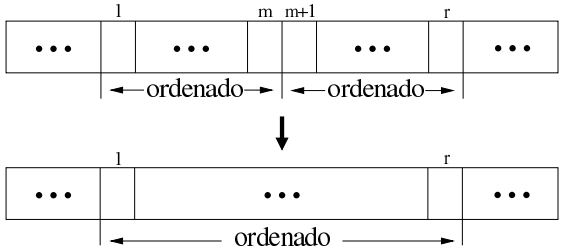
1. Se divide el vector en dos *subvectores*: $A[1,\ldots,\frac{N}{2}]$ $A[\frac{N}{2}+1,\ldots,N]$.
2. Se ordenan indepedientemente cada uno de los *subvectores*.
3. Se *mezclan* los dos *subvectores* ordenados de manera que se obtenga un nuevo vector ordenado de talla N .

Ejemplo



Algoritmo de mezcla o fusión

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



Algoritmo:	Merge
Argumentos:	A: vector $A[l, \ldots, r]$,
	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 = (int *) malloc((r-l+1) * sizeof(int));
 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);
}
```

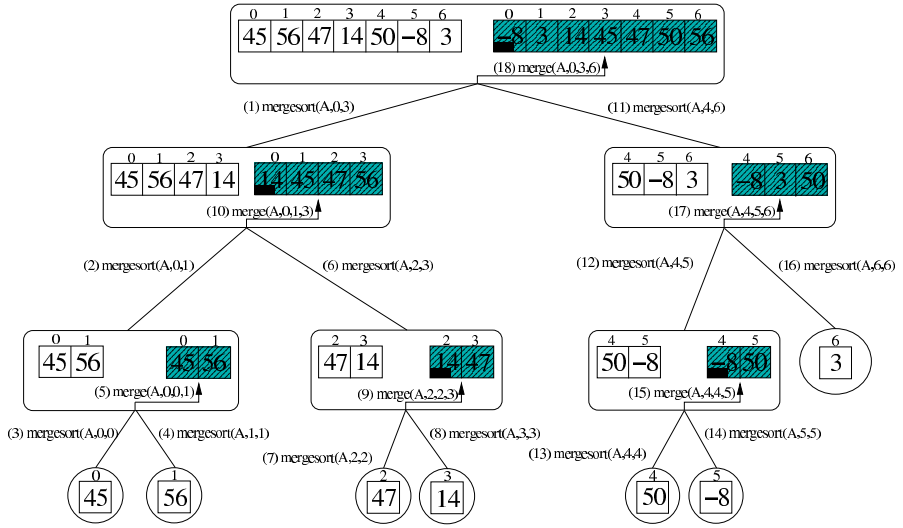
Coste temporal del algoritmo de mezcla $\Longrightarrow \Theta(n)$.

Algoritmo *Mergesort*

Algoritmo:	Mergesort
Argumentos:	A: vector $A[l,\ldots,r]$,
	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);
 }
}
```

Ejemplo del funcionamiento del algoritmo



Análisis de la eficiencia

Supondremos por simplicidad que n es una potencia de 2:

$$T(n) = \begin{cases} c_1 & n = 1 \\ 2T(\frac{n}{2}) + c_2 n + c_3 & n > 1 \end{cases}$$

Resolviendo por sustitución:

$$\begin{aligned} T(n) &= 2T(\frac{n}{2}) + c_2 n + c_3 \\ &= 2(2T(\frac{n}{2^2}) + c_2\frac{n}{2} + c_3) + c_2 n + c_3 \\ &= 2^2 T(\frac{n}{2^2}) + 2c_2 n + c_3(2+1) \\ &= 2^3 T(\frac{n}{2^3}) + 3c_2 n + c_3(2^2+2+1) \\ &= p\ iteraciones \ldots \\ &= 2^p T(\frac{n}{2^p}) + pc_2 n + c_3(2^{p-1} + \ldots + 2^2 + 2 + 1) \\ (p = log_2\ n) &= nc_1 + c_2 n\ log_2\ n + c_3(n-1) \in \Theta(n\ log n) \end{aligned}$$

Coste temporal del algoritmo "mergesort" $\Longrightarrow \Theta(n\ log n)$

Problema del balanceo

Si se descompone el problema original, de talla n, en un *subproblema* de talla $n-1$ y en otro de talla 1.

$$T(n) = \begin{cases} c_1 & n = 1 \\ T(n-1) + c_2 n + c_3 & n > 1 \end{cases}$$

Resolviendo por sustitución:

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

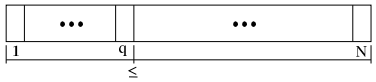
Posibles mejoras del algoritmo

- 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.

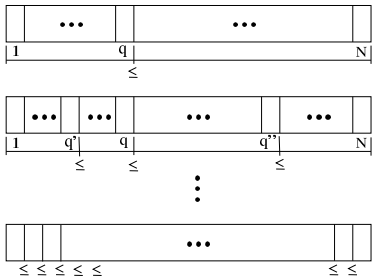
- Se puede evitar el tiempo que se dedica en el algoritmo "*merge*" a copiar el vector auxiliar (resultado) en el vector original, realizando las llamadas recursivas de forma que se alternen, en cada nivel, el vector auxiliar y el original.

Ordenación por partición: Quicksort

1. Elegir como *pivote* uno de los elementos del vector. Permutar los elementos: los que son menores que el pivote quedan en $A[1,\ldots,q]$, y los que son mayores en $A[q+1,\ldots,N]$ (**partición**).



2. Aplicar el mismo método repetidamente a cada uno de los *subvectores* hasta obtener *subvectores* de talla 1, así se obtendrá la ordenación del vector original.



Algoritmo de partición

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

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

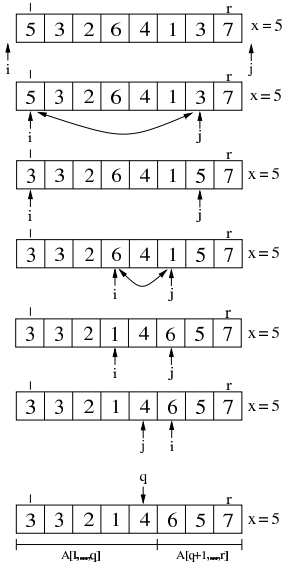
1. Elegir el primer elemento del vector como pivote $x = A[1]$.
2. Usando un índice i, recorrer incrementalmente desde la posición 1 una región $A[1,\ldots,i]$ hasta encontrar un elemento $A[i] \geq x$.
3. Usando un índice j, recorrer decrementalmente desde la posición N una región $A[j,\ldots,N]$ hasta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

Algoritmo de partición (II)

```
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);

}
```

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

Ejemplo del algoritmo de partición

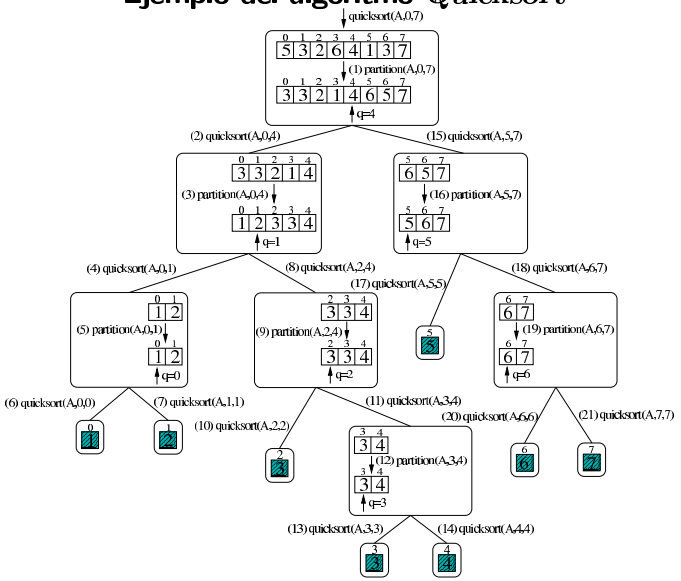


Algoritmo *Quicksort*

Algoritmo:	Quicksort
Argumentos:	A: vector $A[l,\ldots,r]$,
	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 algoritmo *Quicksort*



Análisis de la eficiencia

Caso peor

$$T(n) = \begin{cases} c_1 & n = 1 \\ T(n-1) + c_2 n + c_3 & n > 1 \end{cases}$$

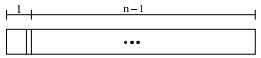
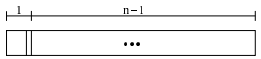
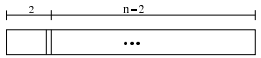
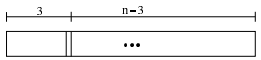
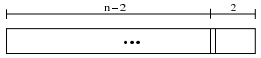
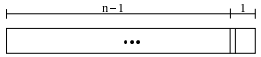
$\Longrightarrow T(n) \in \Theta(n^2)$.

Caso mejor

$$T(n) = \begin{cases} c_1 & n = 1 \\ 2T(\frac{n}{2}) + c_2 n + c_3 & n > 1 \end{cases}$$

$\Longrightarrow T(n) \in \Theta(n\ log n)$.

Caso medio

A	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	$\frac{1}{n}$

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

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

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

podemos reescribir $T(n)$ como:

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

Para $k=1,2,\ldots,n-1$ cada término $T(k)$ ocurre una vez como $T(q)$ y otra como $T(n-q)$, por lo tanto:

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

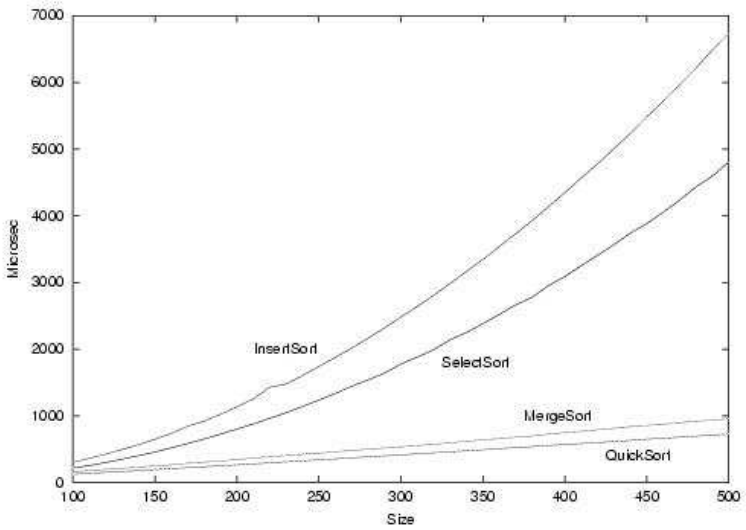
$$T(n) = \begin{cases} c_1 & n \le 1 \\ \frac{2}{n}\sum_{k=1}^{n-1}T(k)+c_2 n & n>1 \end{cases}$$

Por inducción $\forall\ n>1,\ T(n) \le c_3\, n log_2 n$, donde $c_3 = 2c_2 + c_1$, y por tanto:
$\Longrightarrow T(n) \in O(n\ logn)$

Posibles mejoras del algoritmo

- Resolviendo los *subproblemas* por debajo de cierta talla con un método de ordenación bueno para tallas pequeñas (por ejemplo, los métodos directos) evitaremos realizar cierto número de llamadas recursivas.
- 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* $A[l,\ldots,r]$ seleccionado al azar.
- Para solucionar el problema de usar la mediana como pivote, 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* $A[l,\ldots,r]$, seleccionar tres elementos: $A[l]$, $A[\frac{l+r}{2}]$, $A[r]$.

Comparación empírica de los algoritmos

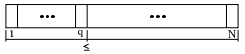


Búsqueda del k-ésimo menor elemento

Problema: Dado un conjunto de n enteros almacenados en un vector $A[1,\ldots,N]$, encontrar el k-ésimo menor elemento.

Estrategia:

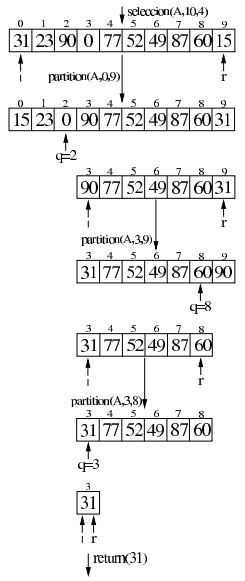
1. Se utiliza el algoritmo de partición (*partition*) utilizado por "*quicksort*" para dividir el vector en dos partes, de manera que todo elemento de la primera parte sea menor o igual que todo elemento de la segunda.



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* $A[1,\ldots,q]$; mientras que si, por el contrario, $q < k \leq N$ únicamente será necesario ordenar el *subvector* $A[q+1,\ldots,N]$.

Algoritmo:	Selección
Argumentos:	A: vector $A[0,\ldots,n-1]$,
	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]);
}
```



Análisis de la eficiencia

- Caso peor:

$$T(n) = \begin{cases} c_1 & n = 1 \\ T(n-1) + c_2 n + c_3 & n > 1 \end{cases}$$

$\Longrightarrow T(n) \in \Theta(n^2)$

- Caso mejor:

$$T(n) = \begin{cases} c_1 & n = 1 \\ T(\frac{n}{2}) + c_2 n + c_3 & n > 1 \end{cases}$$

Resolviendo por sustitución:

$$\begin{aligned} T(n) &= T(\frac{n}{2}) + c_2 n + c_3 \\ &= T(\frac{n}{2^2}) + (n + \frac{n}{2})c_2 + 2c_3 \\ &= T(\frac{n}{2^3}) + c_2 n(1 + \frac{1}{2} + \frac{1}{2^2}) + 3c_3 \\ &= p \; iteraciones \ldots \\ &= T(\frac{n}{2^p}) + c_2 n(1 + \frac{1}{2} + \frac{1}{2^2} + \ldots + \frac{1}{2^{p-1}}) + pc_3 \\ &= T(\frac{n}{2^p}) + c_2 n \frac{2(2^p - 1)}{2^p} + pc_3 \\ (p = log_2\ n) &= c_1 + 2c_2(n-1) + c_3 log_2 n \in O(n) \end{aligned}$$

- Caso medio: $\approx$ caso mejor $\Longrightarrow T(n) \in O(n)$.

Búsqueda binaria

Problema: Dado un vector A ordenado por orden no decreciente de n enteros, buscar la posición correspondiente a un elemento x.

Estrategia:

Dado un vector A[1. . . n] y un elemento x:

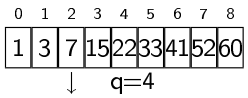
- Tomar el elemento de la mitad del vector A[q], con q=$n/2$.
- Si x=A[q] entonces la posición buscada es q sino:
  - Si x <A[q] entonces aplicar el mismo criterio de búsqueda en el vector A[1. . . q-1].
  - Si x >A[q] entonces aplicar el mismo criterio de búsqueda en el vector A[q+1. . . n].

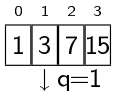
Algoritmo:	Búsqueda_binaria
Argumentos:	A: vector $A[l, \ldots, r]$,
	l: índice de la primera posición del vector,
	r: índice de la última posición del vector,
	x: elemento del vector a buscar

```
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));
 }
}
```

Ejemplo de búsqueda binaria

x=1 **llamada inicial: busqueda_binaria(A,0,8,1);**





A[0]=1 $\Longrightarrow$ return(0)

Análisis de la eficiencia

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

Resolviendo por sustitución:

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

$\Longrightarrow O(\log n)$

Cálculo de la potencia de un número

Problema: $b^n, n \geq 0$

Algoritmo trivial

- Descomposición trivial: $b^n = \underbrace{b \cdot b \cdot b \cdots b \cdot b}_{n}$

```
int potencia(int b, int n){
 int res, i;
 res = 1;
 for (i=1; i<=n; i++)
  res = res * b;
 return(res);
}
```

$\Longrightarrow$ Coste del algoritmo $O(n)$.

Cálculo de la potencia de un número: Algoritmo "divide y vencerás"

- Descomposición divide y vencerás: $b^n = \begin{cases} b^{\frac{n}{2}} b^{\frac{n}{2}} & \text{si } n \text{ es par} \\ b^{\frac{n}{2}} b^{\frac{n}{2}} b & \text{si } n \text{ es impar} \end{cases}$

```
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);
}
```

Análisis de la eficiencia

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

$\Longrightarrow T(n) \in O(log\ n)$.

Otros problemas mediante DyV

- 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.

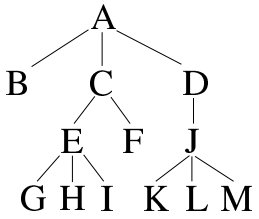
Estructuras de Datos y Algoritmos

Árboles

Definiciones

- *Árbol* de tipo base T:

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*.
   - n es *padre* de las raíces de los subárboles.
   - Las raíces de los subárboles son *hijos* de n.



Definiciones (II)

- **Camino** de n_1 a n_k: sucesión de nodos de un árbol $n_1, n_2, \ldots, n_k$, en la que n_i es padre de n_{i+1} para $1 \leq i < k$.
- **Longitud de un camino**: número de nodos del camino menos 1.
  Hay un camino de longitud cero de cualquier nodo a sí mismo.
- 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.
- **Antecesor propio** o **Descendiente propio**: un antecesor o un descendiente de un nodo que no sea él mismo.
- En un camino $n_1, n_2, \ldots, n_k$, n_i es **antecesor directo** de n_{i+1}, y n_{i+1} es **descendiente directo** de n_i, para $1 \leq i < k$.

Definiciones (III)

- **Hoja** o **nodo terminal**: nodo sin descendientes propios.
- **Nodo interior**: nodo con descendientes propios.
- **Grado de un nodo**: número de descendientes directos que tiene.
- **Grado de un árbol**: máximo de los grados de todos los nodos del árbol.
- **Nivel**:

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 $i+1$.

- **Altura de un nodo**: longitud del camino más largo desde ese nodo hasta una hoja.
- **Altura del árbol**: altura del nodo raíz.
- **Profundidad de un nodo**: longitud del camino único desde la raíz hasta ese nodo.

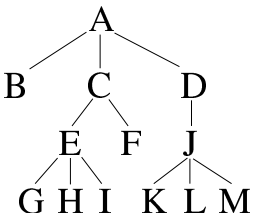
Recorrido de árboles

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

Dependiendo del tipo de recorrido:

- si A es vacío $\Rightarrow$ no realizar acción P;
- si A contiene un único nodo $\Rightarrow$ realizar acción P sobre ese nodo;
- si A tiene raíz n y subárboles $A_1, A_2, \ldots, A_k \Rightarrow$
  - ***Preorden***: realizar acción P sobre n, y, después, recorrer en *preorden* los subárboles $A_1, A_2, \ldots, A_k$ de izquierda a derecha.
  - ***Inorden***: recorrer en *inorden* el subárbol A_1, después realizar acción P sobre n, y, después, recorrer en *inorden* los subárboles $A_2, \ldots, A_k$ de izquierda a derecha.
  - ***Postorden***: recorrer en *postorden* los subárboles $A_1, A_2, \ldots, A_k$ de izquierda a derecha, y después realizar acción P sobre n.

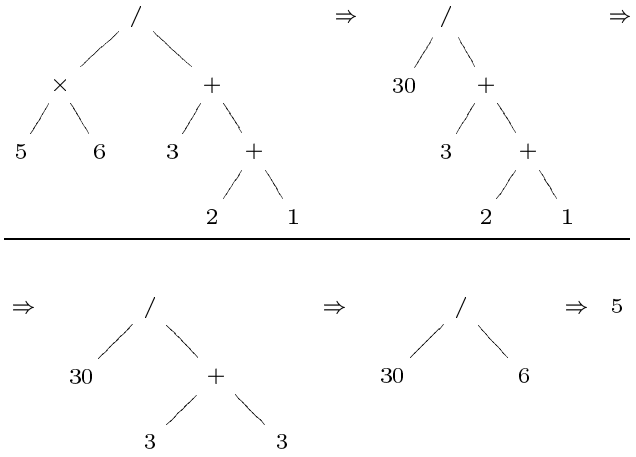
Ejemplo:



- *Preorden*: acción P se realizaría en este orden: A, B, C, E, G, H, I, F, D, J, K, L, M.
- *Inorden*: acción P se realizaría en este orden: B, A, G, E, H, I, C, F, K, J, L, M, D.
- *Postorden*: acción P se realizaría en este orden: B, G, H, I, E, F, C, K, L, M, J, D, A.

Ejemplo:

Evaluar expresión artimética con precedencia de operadores representada mediante un árbol $\Rightarrow$ evaluar primero los operandos de cada operador (recorrido postorden).



Representación de árboles

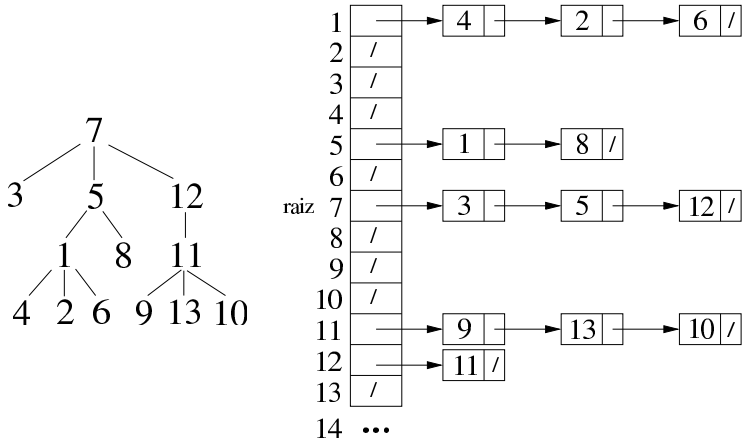
Representación mediante listas de hijos

$\Rightarrow$ formar una lista de los hijos de cada nodo.

Estructura de datos:

1. Vector v con 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.
3. Un índice a la raíz.

Representación mediante listas de hijos: Ejemplo



Representación mediante listas de hijos (III)

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.

Representación mediante listas de hijos (IV)

Ejercicio: función recursiva que imprime los índices de los nodos en *preorden*.

```
#define N ...
typedef struct snodo{
 int e;
 struct snodo *sig;
} nodo;

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

```
void preorden(arbol *T, int n){
 nodo *aux;
 aux = T->v[n];
 printf("%d ",n);
 while (aux != NULL){
  preorden(T,aux->e);
  aux = aux->sig;
 }
}
```

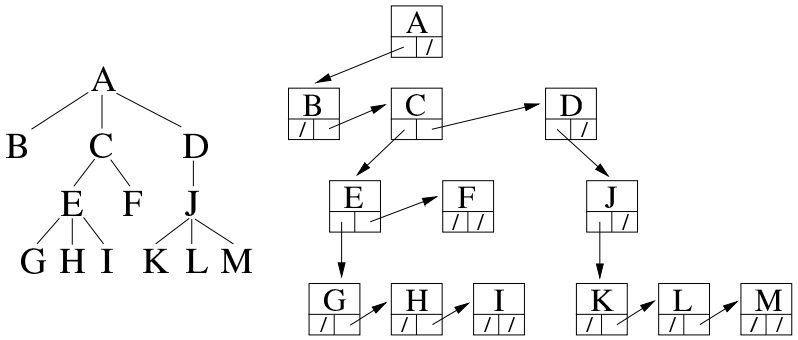
Representación de árboles (II)

Representación "hijo más a la izquierda – hermano derecho"

Para cada nodo, guardar 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.

Representación "hijo más a la izquierda – hermano derecho": Ejemplo

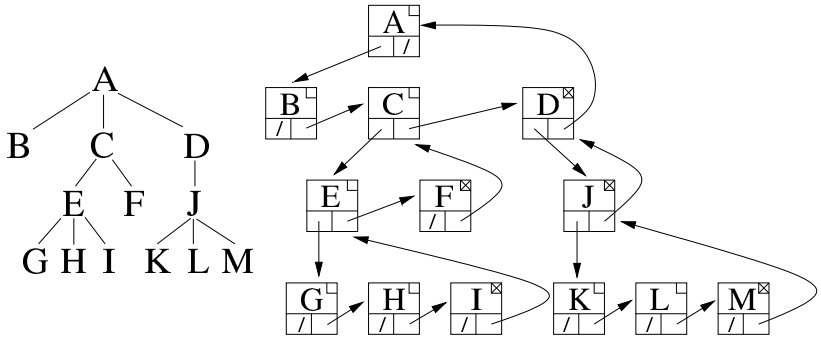


Representación "hijo más a la izquierda – hermano derecho" (III)

- Variante que facilita el acceso al padre desde un nodo hijo: enlazar el nodo hijo que ocupa la posición más a la derecha con su nodo padre.

Especificar, en cada nodo, si el puntero al hermano derecho apunta a un hermano o al padre.

Ejemplo:



Representación "hijo más a la izquierda – hermano derecho" (IV)

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.

Representación "hijo más a la izquierda – hermano derecho" (V)

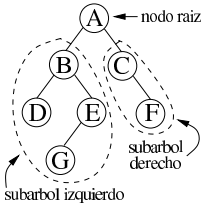
Ejercicio: función que calcula de forma recursiva la altura de un árbol.

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

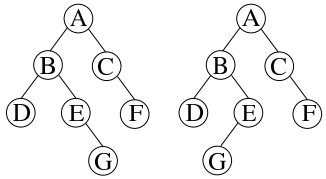
```
int altura(arbol *T){
 arbol *aux;
 int maxhsub=0, hsub;
 if (T == NULL)
  return(0);
 else if (T->hizq == NULL)
  return(0);
 else{
  aux = T->hizq;
  while ( aux != NULL){
   hsub = altura(aux);
   if (hsub > maxhsub)
    maxhsub = hsub;
   aux = aux->der;
  }
  return(maxhsub + 1);
 }
}
```

Árboles binarios

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



Ejemplo de árboles binarios distintos:



Representación de árboles binarios

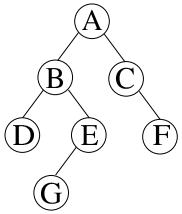
Representación mediante vectores

Estructura de datos:

- Índice al nodo raíz.
- Vector v para almacenar la información de cada nodo.
- Cada elemento del vector (nodo), será una estructura con:

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.

Representación mediante vectores: Ejemplo



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

Definición de tipos en C:

```
#define N ...
typedef ... tipo_baseT;
typedef struct{
 tipo_baseT e;
 int hizq, hder;
} nodo;
typedef struct{
 int raiz;
 nodo v[N];
} arbol;
```

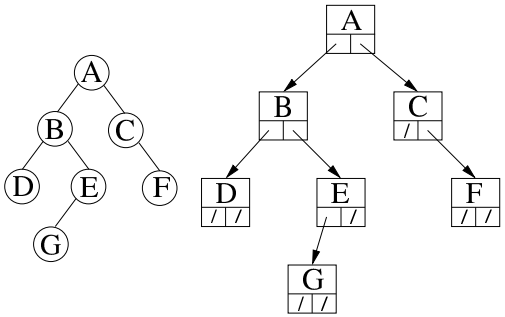
Representación de árboles binarios (II)

Representación mediante variables dinámicas

Para cada nodo, guardar 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.

Representación mediante variables dinámicas: Ejemplo

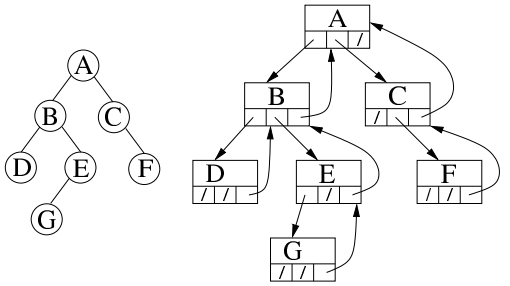


Definición de tipos en C:

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

Representación mediante variables dinámicas (III)

⇒ Variante que facilita el acceso al padre desde un hijo: en cada nodo, un puntero al padre.



Definición de tipos en C:

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

Recorrido de árboles binarios

Al igual que para cualquier tipo de árbol, hay tres formas:

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

Preorden(x)
si $x \neq$ VACIO *entonces*
AccionP(x)
Preorden(hizquierdo(x))
Preorden(hderecho(x))

Inorden(x)
si $x \neq$ VACIO *entonces*
Inorden(hizquierdo(x))
AccionP(x)
Inorden(hderecho(x))

Postorden(x)
si $x \neq$ VACIO *entonces*
Postorden(hizquierdo(x))
Postorden(hderecho(x))
AccionP(x)

Coste $\in \Theta(n)$, siendo n el número de nodos del árbol.

Recorrido de árboles binarios: Implementación

⇒ Representación mediante variables dinámicas.
Acción $P \equiv$ escribir la *clave* del nodo.

Definición de tipos en C:

```
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);
 }
}
```

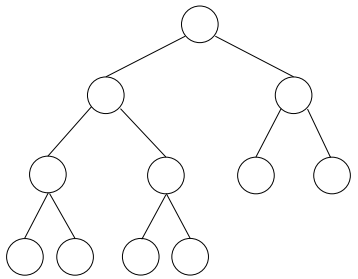
Árboles binarios: Ejercicio

Función en C que elimina todas las hojas del árbol binario, dejando exclusivamente los nodos del árbol que no lo son.

```
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);
 }
}
```

Árbol binario completo

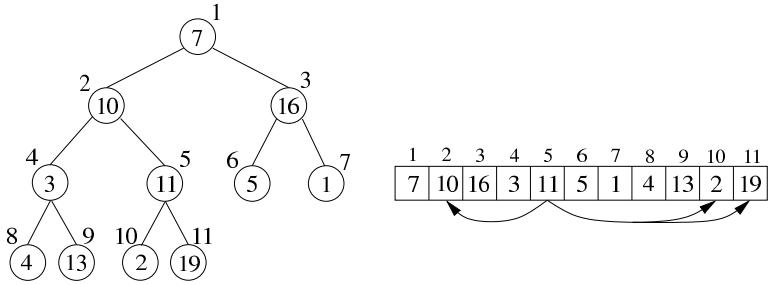
Árbol binario completo: árbol binario en el cual todos los niveles tienen el máximo número posible de nodos excepto, puede ser, el último.
En ese caso, las hojas del último nivel están tan a la izquierda como sea posible.



Árbol binario completo: Representación

Los árboles binarios completos pueden ser representados mediante un vector:

- 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 $i > 1$.



Árbol binario completo: Ejercicio

Tres funciones en C que, dado un nodo de un árbol binario completo representado mediante un vector, calculan la posición del vector en la que se encuentra el nodo padre, el hijo izquierdo y el hijo derecho:

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

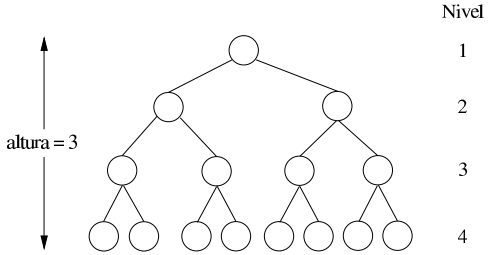
```
int hder(int i)
{
 return((2*i)+1);
}
```

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

Propiedades 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 $\lfloor log_2 n \rfloor$.

Propiedades de los árboles binarios: Ejemplo



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

$n = 15$

$\lfloor log_2 n \rfloor = \lfloor log_2 (15) \rfloor = 3$

Estructuras de Datos y Algoritmos

Conjuntos

Conceptos generales

Conjunto: colección de elementos distintos; cada elemento puede ser un conjunto, o un átomo.

Multiconjunto $\equiv$ conjunto con elementos repetidos.

Representación de conjuntos:

- Representación explícita.
$$C = \{1, 4, 7, 11\}$$
- Representación mediante propiedades.
$$C = \{x \in \mathbb{N} \mid x\ es\ par\}$$

Notación

- Relación fundamental: pertenencia ($\in$)
  - $x \in A$, si x es un miembro del conjunto A.
  - $x \notin A$, si x no es un miembro de A.
- Cardinalidad o talla: número de elementos que contiene un conjunto.
- Conjunto vacío o nulo: no tiene miembros, $\emptyset$.
- $A \subseteq B$ o $B \supseteq A$, si todo miembro de A también es miembro de B.
- Dos conjuntos son iguales sii $A \subseteq B$ y $B \subseteq A$.
- Subconjunto propio: $A \neq B$ y $A \subseteq B$.

Operaciones elementales sobre conjuntos

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

Conjuntos dinámicos

Conjunto dinámico: sus elementos pueden variar a lo largo del tiempo.

Representación de conjuntos dinámicos

Sus elementos tendrán:

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

Puede existir una relación de orden total entre las claves.
Ej: las claves son números enteros, reales, palabras (orden alfabético).

Si $\exists$ un orden total $\rightarrow$ definir mínimo y máximo, o predecesor o sucesor de un elemento.

Operaciones sobre conjuntos dinámicos

Dados S y x tal que $clave(x) = k$. Existen dos tipos de operaciones:

- Consultoras:
  - Buscar(S,k) $\rightarrow x \in S$ tal que $clave(x) = k$.
    $\rightarrow$ nulo si $x \notin S$.
  - Vacío(S): si S es vacío o no.

  Operaciones posibles si en S existe una relación de orden total entre las claves:
  - Mínimo(S): $\rightarrow x$ con clave k más pequeña del S.
  - Máximo(S): $\rightarrow x$ con clave k más grande del S.
  - Predecesor(S,x): $\rightarrow$ elemento de clave inmediatamente inferior a la de x.
  - Sucesor(S,x): $\rightarrow$ elemento de clave inmediatamente superior a la de x.
- Modificadoras:
  - Insertar(S,x): Añade x a S.
  - Borrar(S,x): Elimina x de S.
  - Crear(S): Crea S vacío.

Tablas de dispersión o tablas $Hash$

Diccionario: conjunto que permite, principalmente, las operaciones insertar un elemento, borrar un elemento y determinar la pertenencia de un elemento.

Tablas de direccionamiento directo

Direccionamiento directo:

- Aconsejable cuando el universo U de elementos posibles sea pequeño.
- Cada elemento se identifica mediante una clave única.
- Representación usando un vector de tamaño igual a la talla del universo $|U|$.
  $T[0, \ldots, \mid U \mid -1]$: cada posición k del vector referencia al elemento x con clave k.

Representación de Tablas de direccionamiento directo

Vector de punteros a estructuras donde se guarda la información de los elementos.

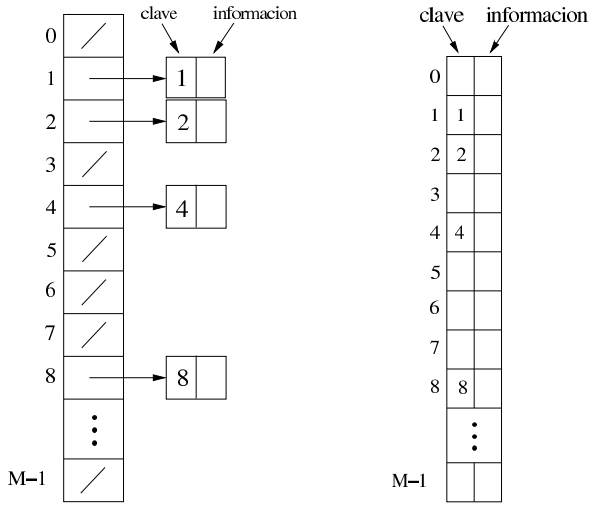
Operaciones triviales de coste $O(1)$:

- insertar(T,x).
- buscar(T,k).
- borrar(T,x).

Variante: almacenar la información en el propio vector.

- La posición del elemento indica cuál es su clave.
- Necesidad de mecanismo para distinguir posiciones vacías u ocupadas.

Representación de Tablas de direccionamiento directo (II)



Tablas de dispersión o tablas *Hash*

Usando direccionamiento directo, si el universo U es grande:

- Imposibilidad de almacenar el vector.
- Número de elementos suele ser pequeño en comparación con la talla del universo $U \rightarrow$ espacio desaprovechado.

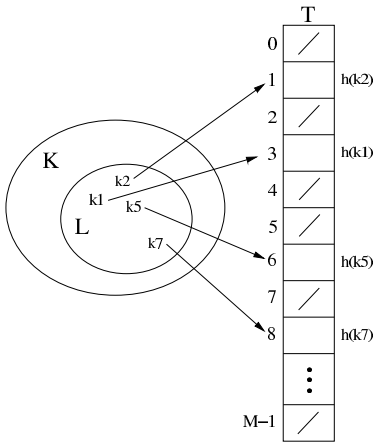
$\Rightarrow$ limitar tamaño del vector $T[0, \ldots, M-1]$.

Tabla de dispersión (tabla hash): la posición de x con clave k, se obtiene al aplicar una *función de dispersion* o *hashing* h sobre la clave k: $h(k)$.

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

Cubeta: cada posición del vector.
x con clave k *se dispersa* en la cubeta $h(k)$.

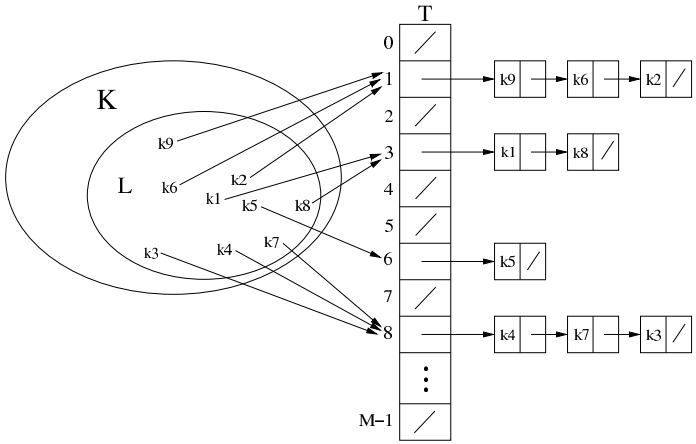
Tablas de dispersión o tablas *Hash* (II)



Colisión: dos o más claves en la misma cubeta.

Resolución de colisiones *por encadenamiento*

Estrategia: organizar los elementos de una cubeta mediante una lista enlazada.



Resolución de colisiones *por encadenamiento* (II)

Dado T y x, tal que $clave(x) = k$, las operaciones son:

- 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

Dada T con m cubetas y que almacena n elementos $\Rightarrow$ **factor de carga**: $\alpha = \frac{n}{m}$.

Asunción: $h(k)$ puede ser calculado en $O(1)$.

- Insertar $\rightarrow O(1)$.
- Buscar $\left.\begin{array}{l} \rightarrow O(n). \\ \rightarrow \Theta(1+\alpha) \end{array}\right\}$ si tenemos que $n = O(m) \Rightarrow O(1)$
- Borrar $\rightarrow \Theta(1+\alpha) \Rightarrow O(1)$

Funciones de dispersión

- Función "ideal" $\rightarrow$ satisface la **dispersión uniforme simple**: cada elemento con igual probabilidad de dispersarse en las m cubetas.
- Dificultad de encontrar función con esta condición.
- Usamos funciones que dispersen de forma aceptable los elementos entre las cubetas.
- Universo de claves $\equiv$ conjunto de números naturales $\mathbb{N} = \{0, 1, 2, \dots\}$.
- Representación de claves como números naturales.
  Ej: cadena de caracteres $\rightarrow$ combinar la representación ASCII de sus caracteres.

Funciones de dispersión: método de la división

- La clave k se transforma en un valor entre 0 y $m-1$:

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

  Ejemplo:
  Si $k = 100$ y $m = 12 \rightarrow h(100) = 100 \ mod \ 12 = 4 \rightarrow$ cubeta 4.
  Si $k = 16$ y $m = 12 \rightarrow h(16) = 16 \ mod \ 12 = 4 \rightarrow$ cubeta 4.

- **Aspecto crítico**: elección de m.
  Buen comportamiento $\rightarrow$ m primo y no próximo a una potencia de 2.
  Ejemplo:
  Almacenar 2000 cadenas con $\alpha \leq 3$.
  Mínimo número de cubetas necesario: $2000/3 = 666,\hat{6}$.
  m próximo a 666, primo y no cercano a una potencia de 2: 701.

Funciones de dispersión: método de la multiplicación

- La clave k se transforma en un valor entre 0 y $m-1$ en dos pasos:

1. Multiplicar k por una constante en el rango $0 < A < 1$ y extraer únicamente la parte decimal.

$$(k \cdot A)\ mod\ 1$$

2. Multiplicar el valor anterior por m, y truncar el resultado al entero más próximo por debajo.

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

- El valor de m no es crítico.
- Tomar m como potencia de 2 para facilitar cómputo: $m = 2^p$, p entero.

Funciones de dispersión: Ejemplos sobre cadenas de caracteres

- Realizar conversión de la cadena a un numero natural.
- Cadena x de n caracteres ($x = x_0x_1x_2 \ldots x_{n-1}$).

Ejemplos de funciones basadas en el método de la división.

- **Función** 1: sumar los códigos *ASCII* de cada caracter. (aprovecha toda la clave)

$$h(x) \quad = \quad (\sum_{i=0}^{n-1} x_i)\ mod\ m$$

Inconvenientes:

- $m \uparrow\uparrow$, mala distribución de las claves.
  Ej: $m = 10007$, cadenas de longitud $\leq 10 \rightarrow$ máximo valor de x: $255 \cdot 10 = 2550$.
  Cubetas desde 2551 a 10006 vacías.
- El orden de los caracteres no se tiene en cuenta. h("cosa")=h("saco").

Ejemplos de funciones basadas en el método de la división. (II)

- **Función** 2: usar tres primeros caracteres como números en una determinada base (256).

$$h(x) \quad = \quad (\sum_{i=0}^{2} x_i\ 256^i)\ mod\ m$$

Inconvenientes:

- Cadenas con tres primeros caracteres iguales $\rightarrow$ misma cubeta.
  h("clase")=h("clarinete")=h("clan").

- **Función** 3: Similar a función 2, considerando toda la cadena.

$$h(x) \quad = \quad (\sum_{i=0}^{n-1} x_i\ 256^{((n-1)-i)})\ mod\ m$$

Inconvenientes:

- Cálculo de h costoso.

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

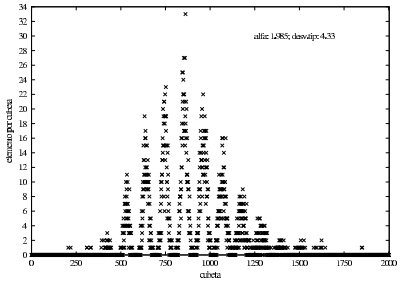
- **Funcion** 4: sumar los códigos *ASCII* de cada caracter considerando que son un número en base 2.

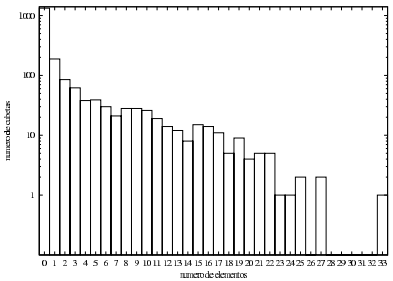
$$h(x) \quad = \quad \lfloor m\ (((\sum_{i=0}^{n-1} x_i\ 2^{((n-1)-i)}) \cdot A)\ mod\ 1)\ \rfloor$$

Ejemplos de funciones sobre cadenas de caracteres: Evaluación empírica

Número de elementos $n = 3975$. Número de cubetas $m = 2003$.

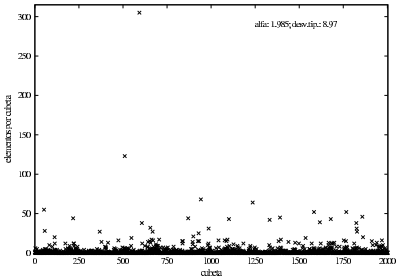
Función 1:

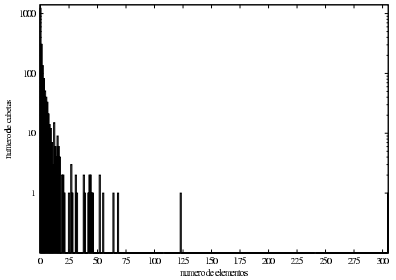




Ejemplos de funciones sobre cadenas de caracteres: Evaluación empírica (II)

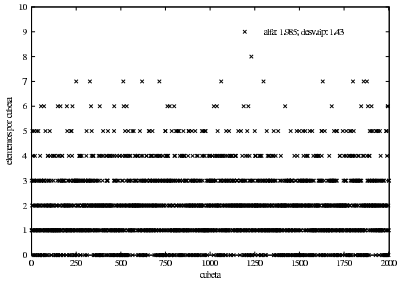
Función 2:

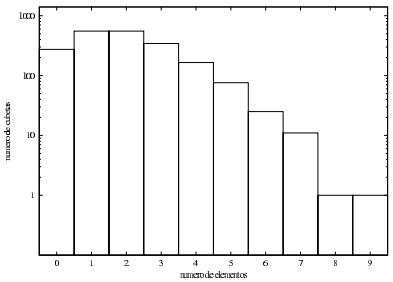




Ejemplos de funciones sobre cadenas de caracteres: Evaluación empírica (III)

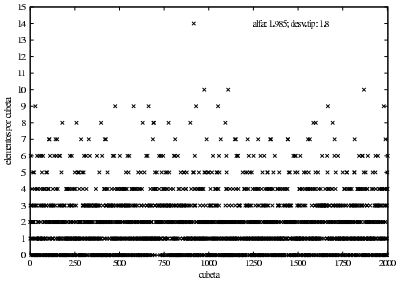
Función 3:

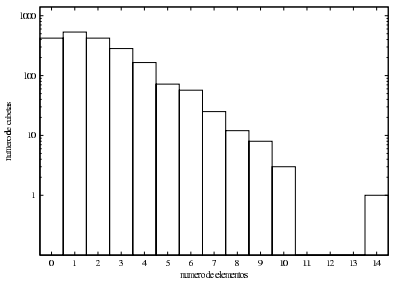




Ejemplos de funciones sobre cadenas de caracteres: Evaluación empírica (IV)

Función 4:

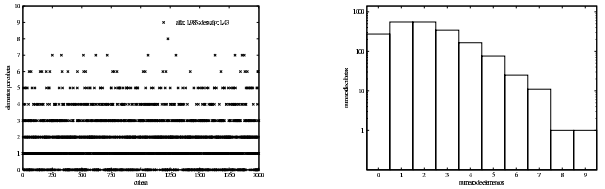




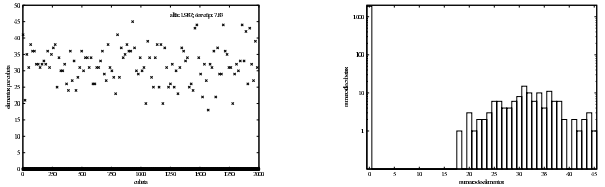
Ejemplos de funciones sobre cadenas de caracteres: Variación del n^o de cubetas

Función 3:

$m = 2003$



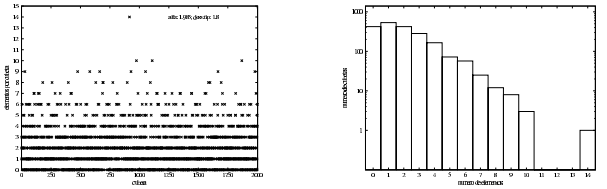
$m = 2000$



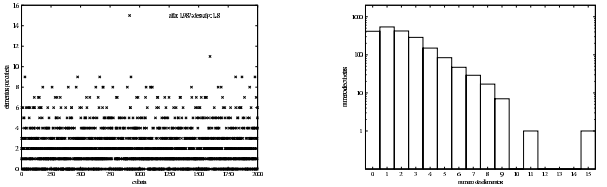
Ejemplos de funciones sobre cadenas de caracteres: Variación del n^o de cubetas (II)

Función 4:

$m = 2003$



$m = 2000$



Tablas de dispersión: Ejercicio

Definiciones:

```
#define NCUB ...
typedef struct snodo{
 char *pal;
 struct snodo *sig;
}nodo;
typedef nodo *Tabla[NCUB];
Tabla T1, T2, T3;

/* Inicializa una tabla vacía */
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)
```

Tablas de dispersión: Ejercicio (II)

Función que crea una tabla $T3$ con los elementos pertenecientes a la intersección de los conjuntos de 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;
  }
 }
}
```

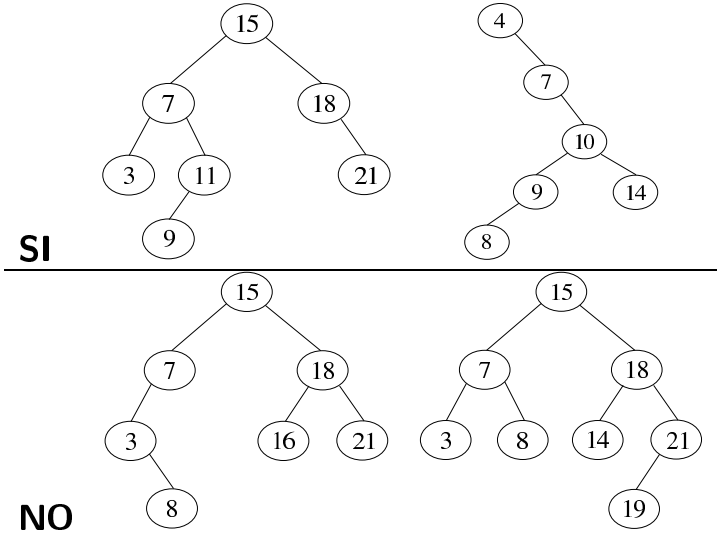
Árboles binarios de búsqueda

Árbol binario de búsqueda: árbol binario que puede estar vacío, o si no está vacío cumple:

- Cada nodo contiene una clave única.
- Para cada nodo n, si su subárbol izquierdo no es vacío, todas las claves almacenadas en su subárbol izquierdo son menores que la clave almacenada en n.
- Para cada nodo n, si su subárbol derecho no es vacío, todas las claves almacenadas en su subárbol derecho son mayores que la clave almacenada en n.

Usado para representar diccionarios, colas de prioridad, etc.

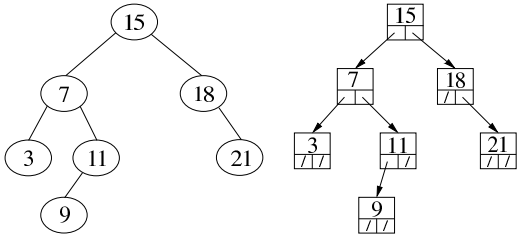
Árboles binarios de búsqueda: Ejemplos



Representación de árboles binarios de búsqueda

Usando variables dinámicas, cada nodo es una estructura:

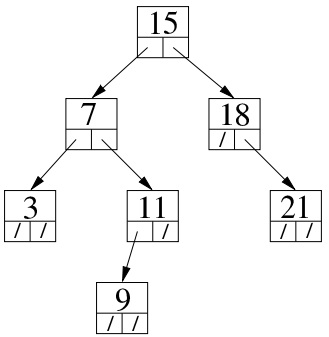
- clave: clave del nodo.
- hijo izquierdo: puntero a la estructura del nodo que es hijo izquierdo.
- hijo derecho: puntero a la estructura del nodo que es hijo derecho.



Representación de árboles binarios de búsqueda (II)

Definición de tipos en C:

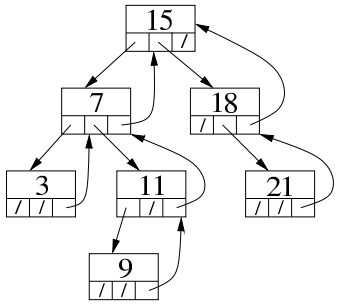
```
typedef ... tipo_baseT;
typedef struct snodo{
 tipo_baseT clave;
 struct snodo *hizq, *hder;
}abb;
```



Representación de árboles binarios de búsqueda (III)

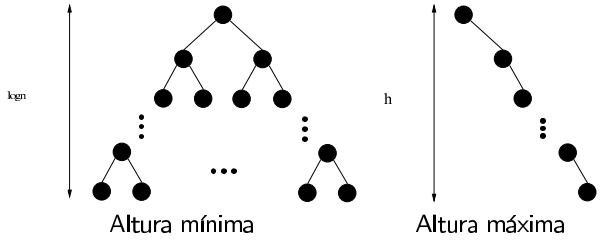
Variante: almacenar puntero al nodo padre.

```
typedef ... tipo_baseT;
typedef struct snodo{
 tipo_baseT clave;
 struct snodo *hizq, *hder, *padre;
}abb;
```



Altura máxima y mínima de un abb

- Altura mínima: $O(log n)$.
- Altura máxima: $O(n)$.



- **Notación**: Altura del árbol $= h$.
- Normalmente, altura de un abb aleatorio $\in O(log n)$

Recorrido de árboles binarios de búsqueda

- 3 recorridos: *preorden*, *inorden* y *postorden*.
- *Inorden*: si la acción es imprimir la clave del nodo, las claves de un abb se imprimirán de forma ordenada (menor a mayor).

```
void inorden(abb *T){
 if (T != NULL){
  inorden(T->hizq);
  printf("%d ",T->clave);
  inorden(T->hder);
 }
}
```

Ejercicio: realizar traza de "inorden" para el árbol ejemplo.

3, 7, 9, 11, 15, 18, 21

Búsqueda de un elemento en un abb

- Operación más común.

```
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);
}
```

- Coste: $O(h)$.

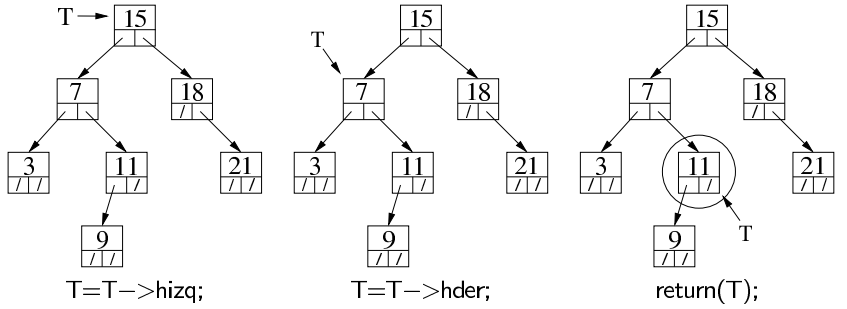
Búsqueda de un elemento en un abb (II)

Ejercicio: versión recursiva de la anterior función de búsqueda en un abb.

```
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);
}
```

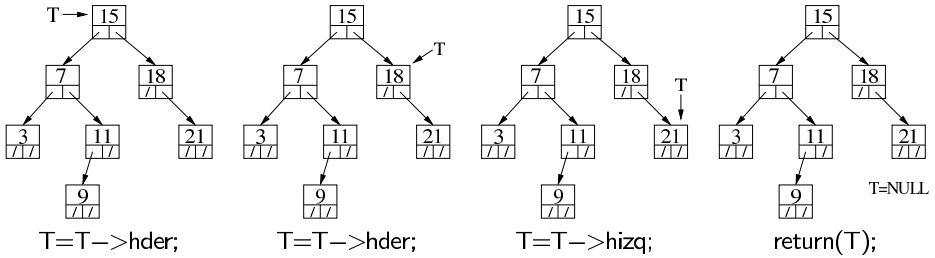
Búsqueda de un elemento en un abb: Ejemplo 1

Buscar $x = 11 \rightarrow$ `nodo = abb_buscar(T,11);`



Búsqueda de un elemento en un abb: Ejemplo 2

Buscar $x = 19 \rightarrow$ `nodo = abb_buscar(T,19);`



Búsqueda del elemento mínimo y del elemento máximo

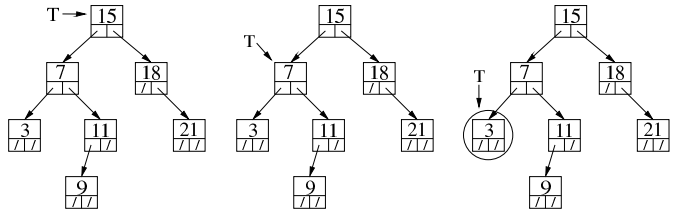
```
abb *minimo(abb *T){
 if (T != NULL)
  while(T->hizq != NULL)
   T=T->hizq;
 return(T);
}
```

```
abb *maximo(abb *T){
 if (T != NULL)
  while(T->hder != NULL)
   T=T->hder;
 return(T);
}
```

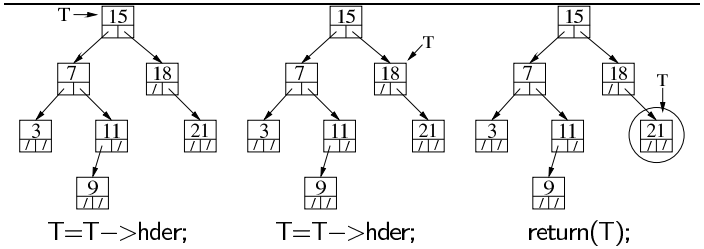
- Coste: $O(h)$.

Búsqueda del mínimo y del máximo: Ejemplo

Mínimo:



Máximo:



Inserción de un elemento en un abb

→ conservar la condición de abb.

Estrategia:

1. Si el abb está vacío → insertar nuevo nodo como raíz.
2. Si el abb no está vacío:
   a) Buscar la posición que le corresponde en el abb al nuevo nodo. Para ello recorrer desde la raíz del árbol actuando como en la operación de búsqueda.
   b) Una vez encontrada su posición, insertar en el árbol enlazándolo correctamente con el nodo que debe ser su padre.

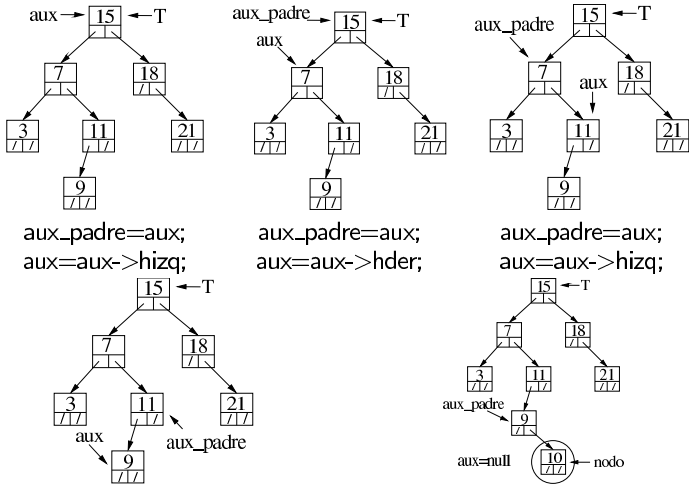
- Coste: $O(h)$.

Inserción de un elemento en un abb (II)

```
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);
 }
}
```

Inserción de un elemento en un abb: Ejemplo

Insertar un nodo de clave 10.



Borrado de un elemento en un abb

Estrategia:

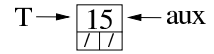
1. Si el elemento a borrar **no tiene hijos (es un hoja)**, se elimina.
2. Si el elemento a borrar es un **nodo interior que tiene un hijo**, se elimina, y su posición es ocupada por el hijo que tiene.
3. Si el elemento a borrar es un **nodo interior que tiene dos hijos**, su lugar es ocupado por el nodo de clave mínima del subárbol derecho (o por el de clave máxima del subárbol izquierdo).

- Coste: $O(h)$.

Ejercicio: Escribir en C una función que realice el borrado de un elemento x en un abb.

Borrado de un elemento en un abb: Hoja

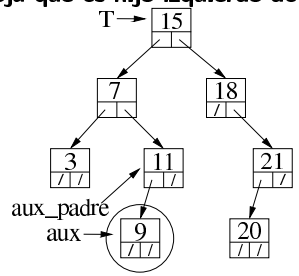
Borrar $x = 15$. **Hoja que no tiene padre.**



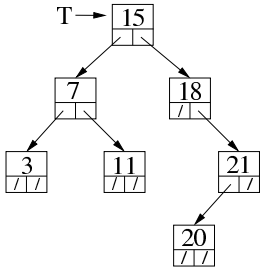
aux_padre=null

T=NULL; free(aux); return(T);

Borrar $x = 9$. **Hoja que es hijo izquierdo de un nodo.**



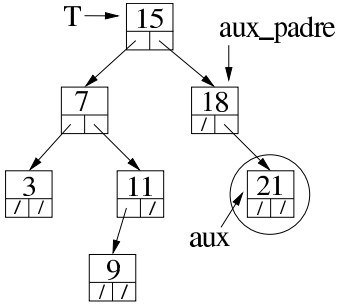
aux_padre->hizq=NULL; free(aux);



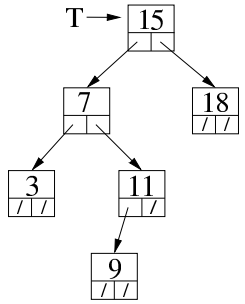
return(T);

Borrado de un elemento en un abb: Hoja (II)

Borrar $x = 21$. **Hoja que es hijo derecho de un nodo.**



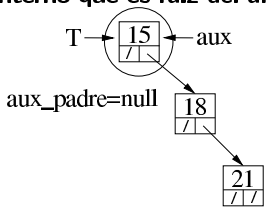
aux_padre->hder = NULL; free(aux);



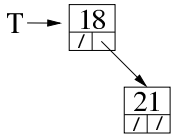
return(T);

Borrado de un elemento en un abb: Nodo interno con un solo hijo

Borrar $x = 15$. **Nodo interno que es raíz del árbol y tiene un único hijo derecho.**

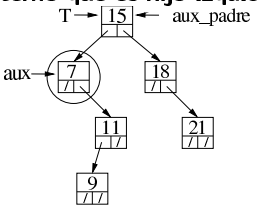


T=aux->hder; free(aux);

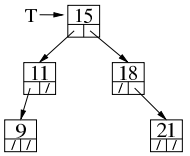


return(T);

Borrar $x = 7$. **Nodo interno que es hijo izquierdo y tiene un único hijo derecho.**



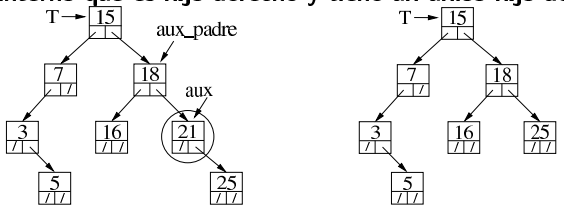
aux_padre->hizq=aux->hder; free(aux);



return(T);

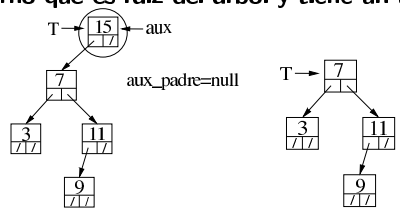
Borrado de un elemento en un abb: Nodo interno con un solo hijo (II)

Borrar $x = 21$. **Nodo interno que es hijo derecho y tiene un único hijo derecho.**



aux_padre->hder=aux->hder; free(aux); return(T);

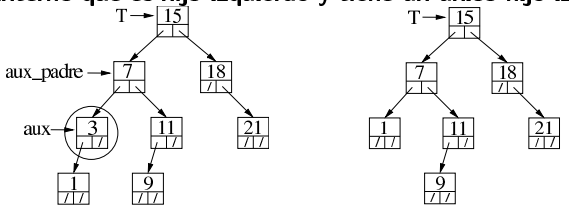
Borrar $x = 15$. **Nodo interno que es raíz del árbol y tiene un único hijo izquierdo.**



T=aux->hizq; free(aux); return(T);

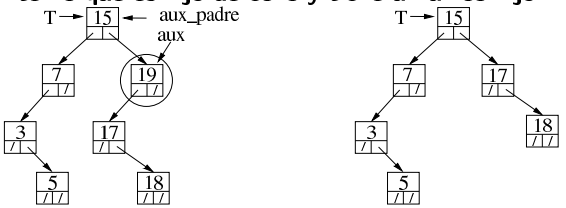
Borrado de un elemento en un abb: Nodo interno con un solo hijo (III)

Borrar $x = 3$. **Nodo interno que es hijo izquierdo y tiene un único hijo izquierdo.**



aux_padre->hizq=aux->hizq; free(aux); return(T);

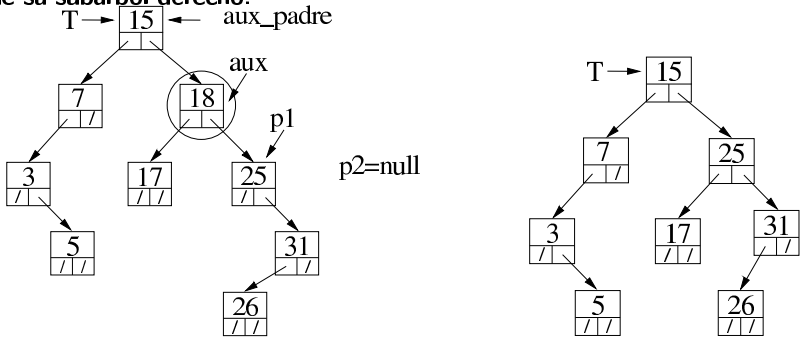
Borrar $x = 19$. **Nodo interno que es hijo derecho y tiene un único hijo izquierdo.**



aux_padre->hder=aux->hizq; free(aux); return(T);

Borrado de un elemento en un abb: Nodo interno con dos hijos

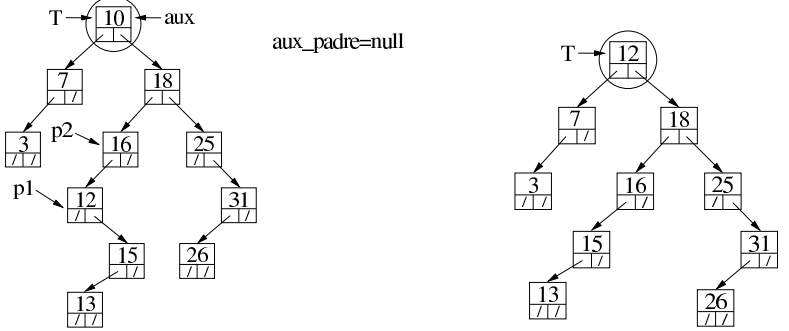
Borrar $x = 18$. **Nodo interno que tiene dos hijos y su hijo derecho es el nodo de clave mínima de su subárbol derecho.**



aux->clave=p1->clave; aux->hder=p1->hder; free(p1); return(T);

Borrado de un elemento en un abb: Nodo interno con dos hijos (II)

Borrar $x = 10$. **Nodo interno que tiene dos hijos y el nodo de clave mínima de su subárbol derecho no es su hijo derecho.**



aux->clave=p1->clave; p2->hizq=p1->hder; free(p1); return(T);

Árboles binarios de búsqueda: Ejercicio

Función recursiva que imprime por pantalla las claves de un árbol menores que una clave k.

```
typedef ... tipo_baseT;
typedef struct snodo{
 tipo_baseT clave;
 struct snodo *hizq, *hder;
}abb;
abb *T;
```

```
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);
 }
}
```

→ Coste: $O(n)$.

Montículos (*Heaps*). Colas de prioridad.

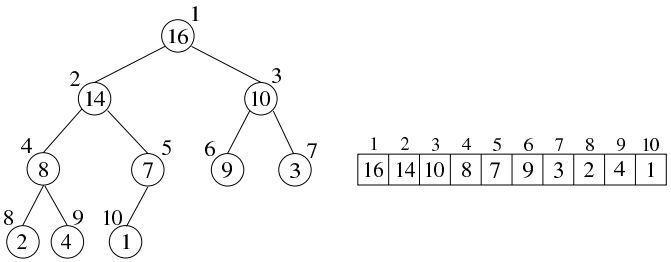
- **Árbol binario completo**: árbol binario en el que todos los niveles tienen el máximo número posible de nodos excepto, puede ser, el último. En ese caso, las hojas del último nivel están tan a la izquierda como sea posible.
- **Montículo o *heap***: conjunto de n elementos representados en un árbol binario completo en el que se cumple:
  para todo nodo i, excepto el raíz, la clave del nodo i es menor o igual que la clave del padre de i (**propiedad de montículo**).

Representación de montículos

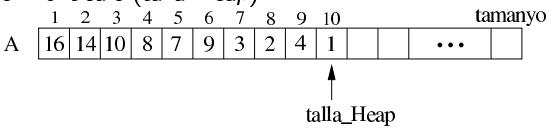
→ usando representación vectorial de los árboles binarios completos:

- Posición 1 del vector → nodo raíz del árbol.
- Nodo en la posición i del vector:
  - Posición $2i$ → nodo que es su hijo izquierdo.
  - Posición $2i+1$ → nodo que es su hijo derecho.
  - Posición $\lfloor i/2 \rfloor$ → nodo padre si $i > 1$.
- Desde posición $\lfloor n/2 \rfloor + 1$ hasta posición n → claves de los nodos que son hojas.
  (n es el número de nodos del árbol binario completo)

Representación de montículos: Ejemplo



Número de nodos del montículo (*talla_Heap*):



Montículos: Propiedades

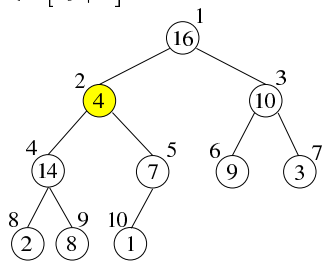
- Usando representación vectorial de un árbol binario completo:

$$A[\lfloor i/2 \rfloor] \geq A[i],\ 1 < i \leq n$$

- Nodo de clave máxima → raíz del árbol.
- Cada rama está ordenada de mayor a menor desde la raíz hasta las hojas.
- Altura $\in \Theta(log\, n)$.

Manteniendo la propiedad de montículo

Suponer $A[i] < A[2i]$ y/o $A[i] < A[2i+1]$



→ función *heapify*: mantiene propiedad de montículo.

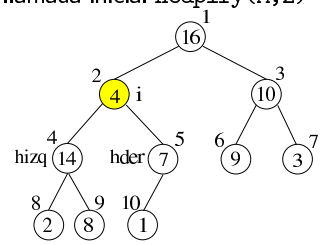
Transformar subárbol de un montículo. Subárboles izquierdo y derecho de i son montículos.

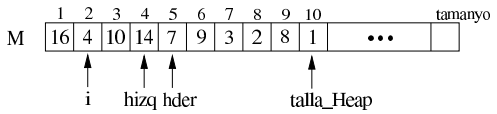
Manteniendo la propiedad de montículo: Heapify

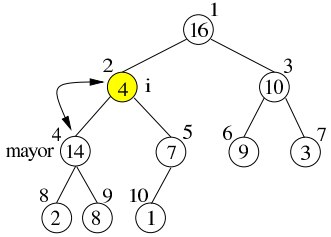
```
void heapify(tipo_baseT *M, int i){
 tipo_baseT aux;
 int hizq, hder, mayor;
 hizq = 2*i;
 hder = 2*i+1;
 if ( (hizq <= talla_Heap) && (M[hizq] > M[i]) )
  mayor = hizq;
 else
  mayor = i;
 if ( (hder <= talla_Heap) && (M[hder] > M[mayor]) )
  mayor = hder;
 if (mayor != i){
  aux = M[i];
  M[i] = M[mayor];
  M[mayor] = aux;
  heapify(M,mayor);
 }
}
```

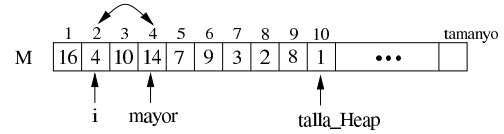
Manteniendo la propiedad de montículo: Ejemplo

→ llamada inicial `heapify(M,2)`

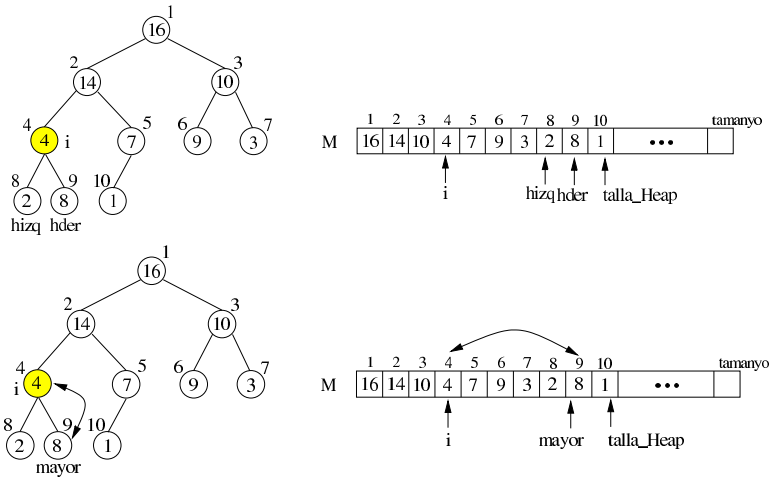




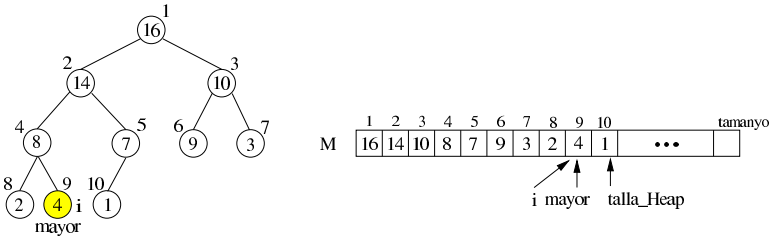




Manteniendo la propiedad de montículo: Ejemplo(II)



Manteniendo la propiedad de montículo: Ejemplo (III)



Coste temporal de *heapify*:

- Caso mejor: $O(1)$.
- Caso peor: $O(h) = O(logn)$.
- Para un nodo i: $O(h)$, h es la altura de i.

Construir un montículo: build_Heap

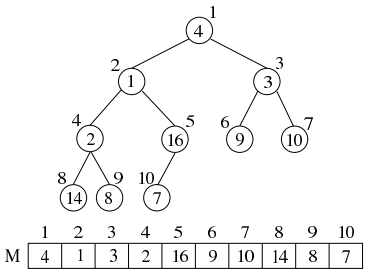
Problema: Dado un conjunto de n elementos representado en un árbol binario completo mediante un vector M, transformar el vector a un montículo.

Estrategia:

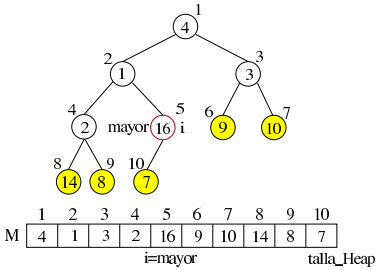
1. Las hojas ya son montículos.
2. Aplicar *heapify* sobre el resto de nodos, comenzando por el de mayor índice que no es hoja ($M[\lfloor n/2 \rfloor]$). Descender hasta el raíz ($M[1]$).

```
void build_Heap(tipo_baseT *M, int n){
 int i;
 talla_Heap = n;
 for(i=n/2; i>0; i--)
  heapify(M,i);
}
```

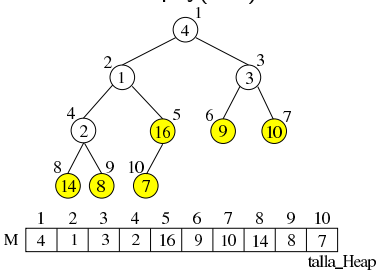
Construir un montículo: Ejemplo



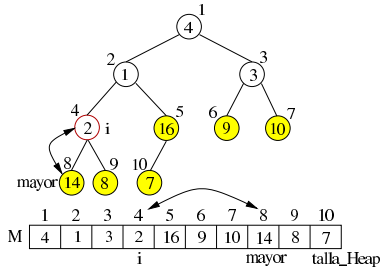
build_Heap(M,10);



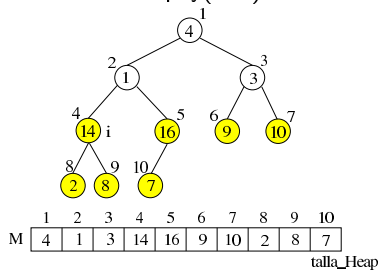
Heapify(M,5);



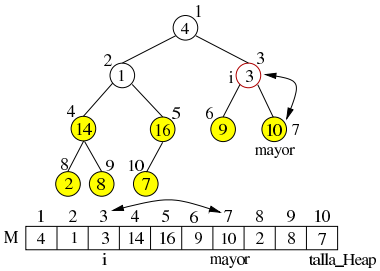
Heapify(M,5); //



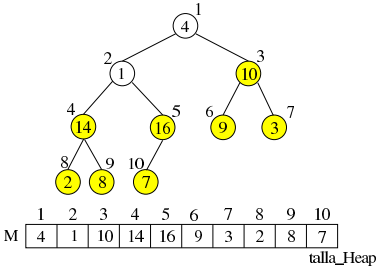
Heapify(M,4);



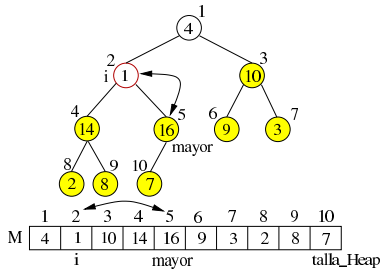
Heapify(M,4); //



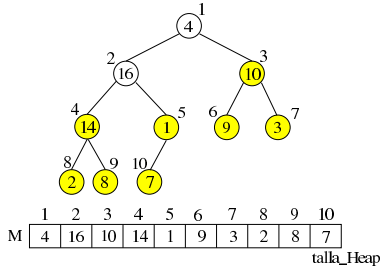
Heapify(M,3);



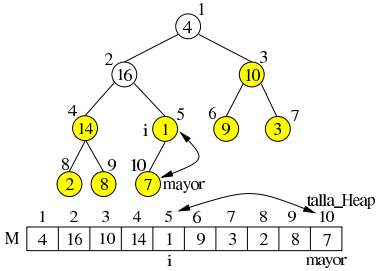
Heapify(M,3); //



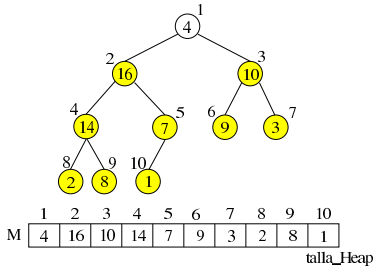
Heapify(M,2);



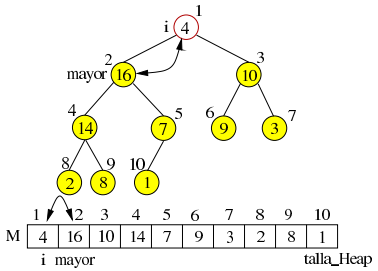
heapify(M,2); → heapify(M,5);



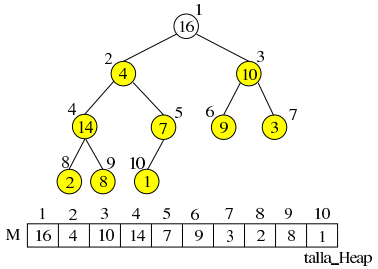
heapify(M,5);



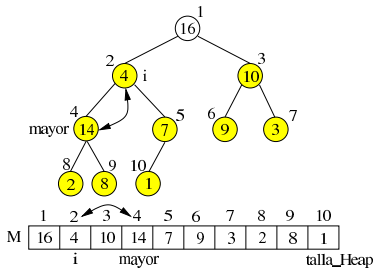
heapify(M,5); // → heapify(M,2); //



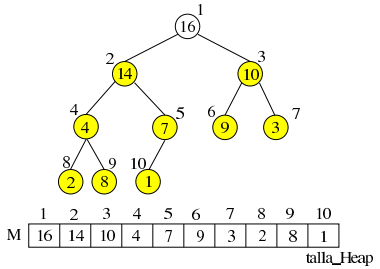
heapify(M,1);



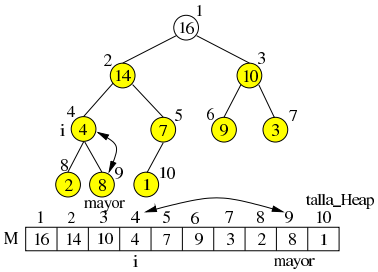
heapify(M,1); → Heapify(M,2);



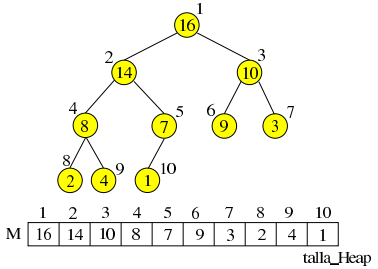
heapify(M,2);



heapify(M,2); → Heapify(M,4);



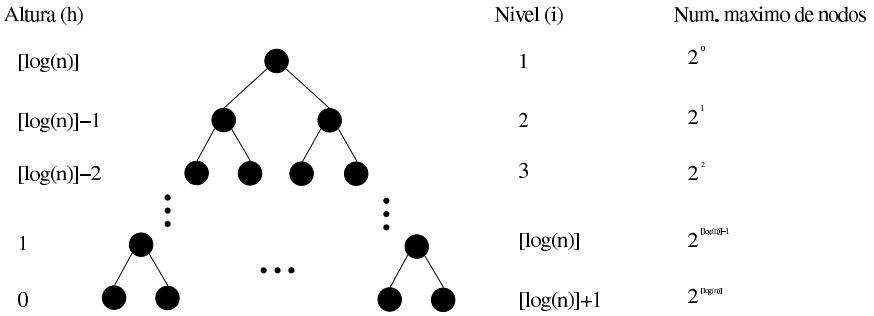
heapify(M,4);



heapify(M,4); // → heapify(M,2); // → heapify(M,1); //

Coste temporal de *build_Heap*

- 1^a aproximación: como *heapify* $\in O(logn) \Rightarrow$ *build_heap* $\in O(n\,logn)$
- 2^a aproximación: *heapify* sobre un nodo $O(h) \rightarrow$ altura h: $N_h O(h)$



$$N_h \leq 2^{\lfloor log_2 n \rfloor - h} = \frac{2^{\lfloor log_2 n \rfloor}}{2^h}$$

Coste temporal de *build_Heap* (II)

Para cada altura, *heapify* sobre cada nodo.

$$T(n) = \sum_{h=0}^{\lfloor log_2 n \rfloor} N_h O(h) \leq \sum_{h=0}^{\lfloor log_2 n \rfloor} \frac{2^{\lfloor log_2 n \rfloor}}{2^h}\, h \leq \sum_{h=0}^{log_2 n} 2^{log_2 n} \frac{h}{2^h} = n \sum_{h=0}^{log_2 n} h \left(\frac{1}{2}\right)^h$$

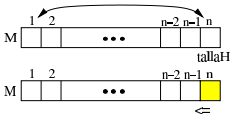
Como $\sum_{h=0}^{\infty} h x^h = x/(1-x)^2$, si $0 < x < 1$, $\Rightarrow$

$$\begin{aligned} n \sum_{h=0}^{log_2 n} h \left(\frac{1}{2}\right)^h &\leq \frac{1/2}{(1-1/2)^2}\, n \\ &= 2n \in O(n) \end{aligned}$$

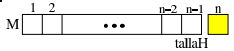
Algoritmo de ordenación *Heapsort*

Estrategia:

1. *build_Heap*: transformar el vector M en un montículo.
2. Clave mayor siempre en la raíz. $\rightarrow$ intercambiar raíz con el último nodo del montículo $\rightarrow$ el elemento mayor queda en su posición final.

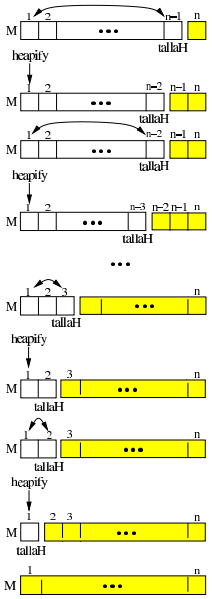


3. Reducir talla del montículo en 1.



4. La raíz puede violar la propiedad de montículo $\rightarrow$ *heapify* sobre la raíz.

5. Repetir (2), (3) y (4) hasta la ordenación total del vector.

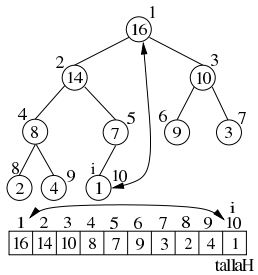


Algoritmo de ordenación *Heapsort* (II)

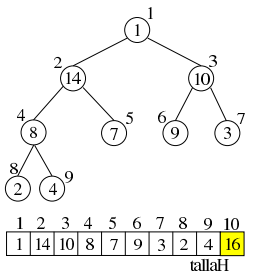
Función que, dado un vector M que contiene n elementos $M[1,\ldots,n]$, ordena el vector M:

```
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);
 }
}
```

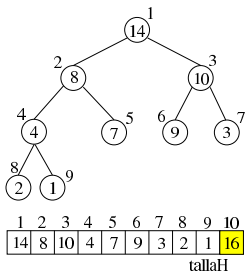
Algoritmo de ordenación *Heapsort*: Ejemplo



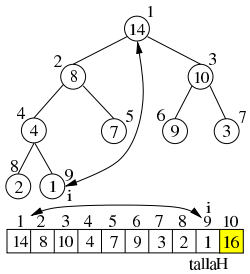
M[1]↔ M[i];



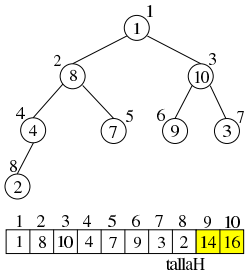
heapify(M,1);



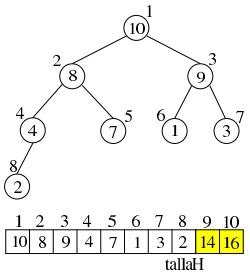
fin iteración $i = 10$;



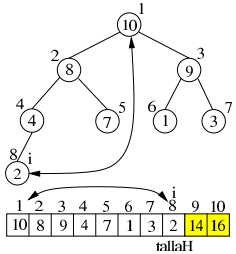
M[1]↔ M[i];



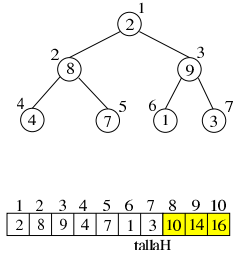
heapify(M,1);



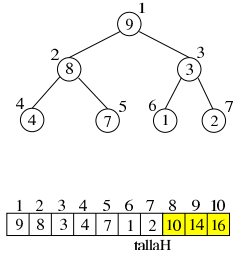
fin iteración $i = 9$;



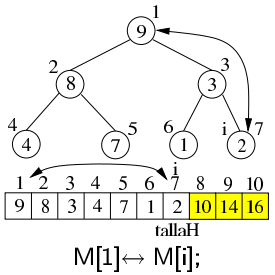
M[1]↔ M[i];



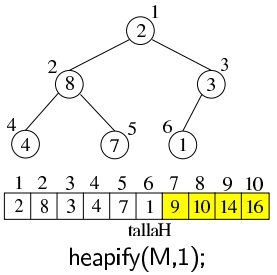
heapify(M,1);



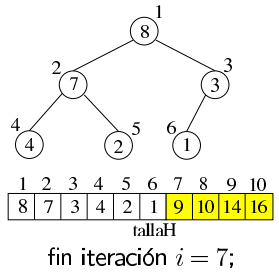
fin iteración $i = 8$;



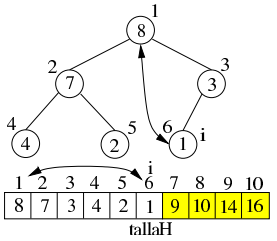
M[1]↔ M[i];



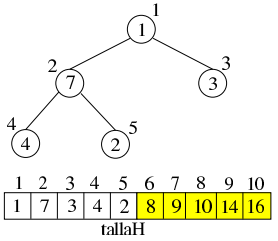
heapify(M,1);



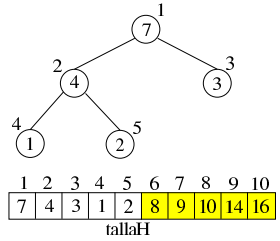
fin iteración $i = 7$;



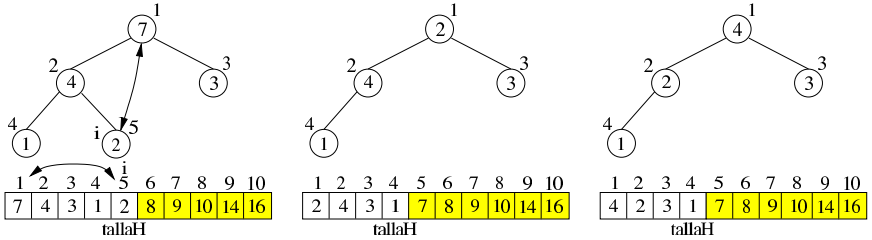
M[1]↔ M[i];

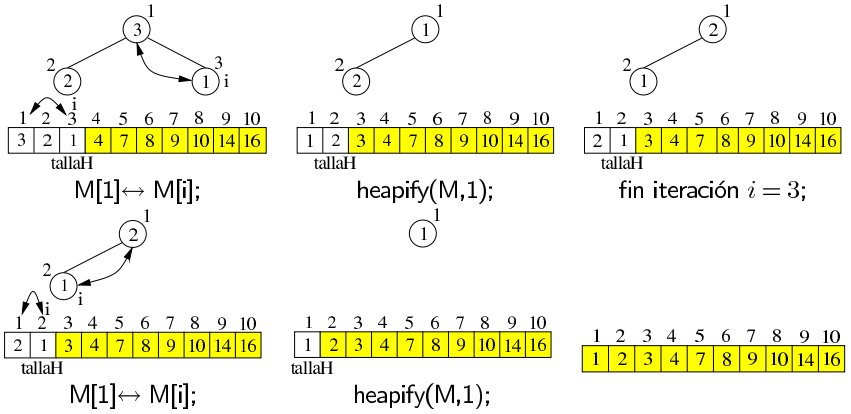


heapify(M,1);



fin iteración $i = 6$;





Coste temporal de *heapsort*

- Construir montículo: $O(n)$.
- $n-1$ llamadas a *heapify* (de coste $O(logn)$).

$\Rightarrow$ *heapsort* $\in O(n\ logn)$.

Únicamente cuando todos los elementos fueran iguales el coste de *heapsort* sería $O(n)$.

Colas de prioridad

- Aplicación usual de montículos.
- **Cola de prioridad** conjunto S de elementos, cada uno de los cuales tiene asociado una *clave* (prioridad).
- Operaciones asociadas:
  - Insert(S,x): inserta x en el conjunto S.
  - Extract_Max(S): borra y devuelve el elemento de S con clave mayor.
  - Maximum(S): devuelve el elemento de S con clave mayor.

$\Rightarrow$ planificación de procesos en un sistema compartido.

Insertar un elemento en un montículo

Estrategia:

1. Expandir el tamaño del montículo en 1 ($talla\_Heap + 1$).

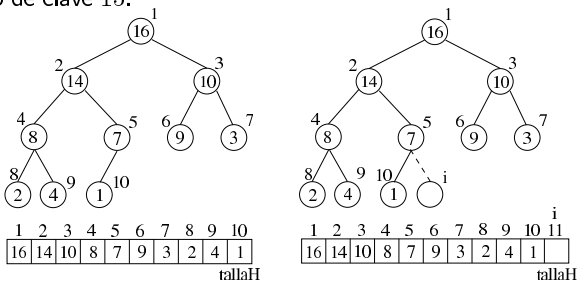
2. Insertar el nuevo elemento en esa posición del vector (hoja más a la izquierda posible).

3. Comparar clave del nuevo elemento con su padre:
   Si la clave del nuevo es mayor → el padre no cumple la propiedad de montículo:

   a) Intercambiar las claves.
   b) si el que ha pasado a ser padre del nuevo elemento no cumple la propiedad de montículo → repetir intercambio de claves hasta que el padre del nuevo elemento sea mayor o igual o hasta que el nuevo sea la raíz.

Insertar un elemento en un montículo: Ejemplo

insertar elemento de clave 15.

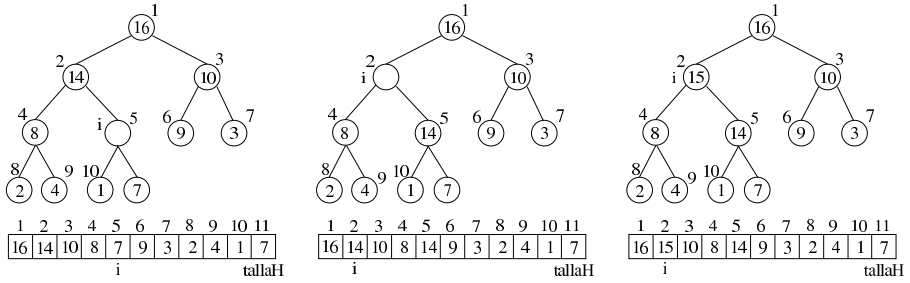


árbol inicial talla_Heap++; i=talla_Heap;

Insertar un elemento en un montículo (II)

Función que inserta un nuevo elemento de clave x (antes de llamarla, debe comprobarse si el tamaño del vector permite la inserción):

```
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;
}
```



M[i]=M[i/2]; i=i/2; M[i]=M[i/2]; i=i/2; M[i]=x;

Coste temporal de insertar en un montículo

- Coste: número de comparaciones a realizar hasta encontrar la posición del nuevo elemento.
- Montículo de n elementos:
  - **caso mejor**: se inserta inicialmente en la posición correspondiente: $O(1)$.
  - **caso peor**: el nuevo elemento es mayor que cualquier otro elemento → llegar a la raíz: $O(logn)$.

Extraer el máximo de un montículo

→ será la raíz

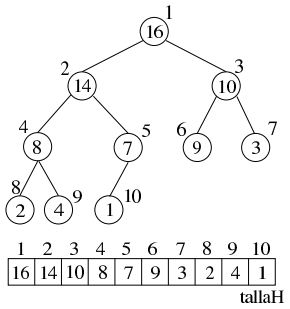
Estrategia:

1. obtener el valor de la raíz ($M[1]$).
2. borrar la raíz sustituyéndola por el valor de la última posición del montículo ($M[1]$=M[talla_Heap]).
   Reducir talla del montículo en 1.
3. Aplicar *heapify* sobre la raíz para restablecer propiedad de montículo.

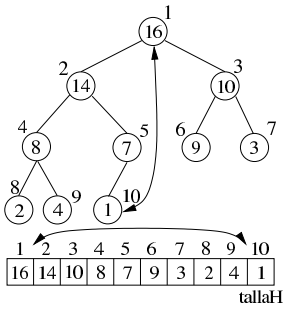
Extraer el máximo de un montículo (II)

```
tipo_baseT extract_max(tipo_baseT *M){
 tipo_baseT max;
 if (talla_Heap == 0){
  fprintf(stderr,"Montículo vacío");
  exit(-1);
 }
 max = M[1];
 M[1] = M[talla_Heap];
 talla_Heap--;
 heapify(M,1);
 return(max);
}
```

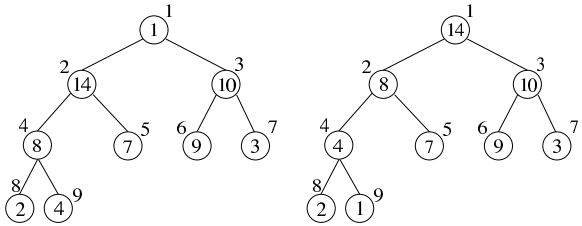
Extraer el máximo de un montículo: Ejemplo



max=M[1];



M[1]↔ M[talla_Heap];



talla_Heap−−; heapify(M,1); return(max);

Coste temporal de extraer el máximo de un montículo

- Realizar *heapify* sobre la raíz: $O(logn)$.
- *Extract_Max* $\in O(logn)$.

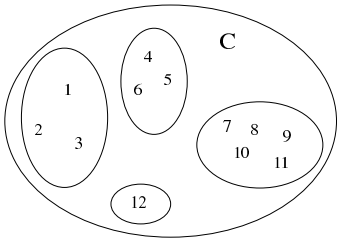
$\Rightarrow$ montículo que representa una cola de prioridad: operaciones $\in O(logn)$.

Estructura de datos para conjuntos disjuntos: MF-set

- **Clase de equivalencia de** a: subconjunto que contiene todos los elementos relacionados con a.
- Clases de equivalencia $\rightarrow$ partición de C
- Todo miembro del conjunto aparece en una clase de equivalencia.
- Para saber si a tiene una relación de equivalencia con $b \Rightarrow$ comprobar si a y b están en la misma clase de equivalencia.

MF-set (II)

- **MF-set (Merge-Find set)**: estructura de n elementos (fijos).
  No se pueden borrar ni añadir elementos.
  Elementos organizados en *clases de equivalencia*.
- Subconjuntos identificados por **representante**:
  - elemento menor del subconjunto.
  - no importa el elemento.
- Si un subconjunto no es modificado $\rightarrow$ mismo representante.



MF-set (III)

Operaciones:

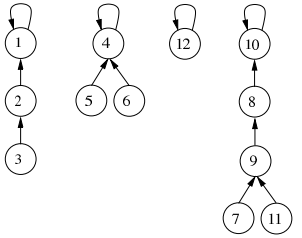
- Union(x,y) (*Merge*): $x \in S_x$ y $y \in S_y \rightarrow$ unión de S_x con S_y.
  El representante del nuevo subconjunto creado es alguno de sus miembros, normalmente se escoge al representante de S_x o al representante de S_y.
  Se eliminan los subconjuntos S_x y S_y.

- Buscar(x) (*Find*): devuelve el representante de la *clase de equivalencia* a la que pertenece x.

Aplicaciones: inferencia de gramáticas, equivalencias de autómatas finitos, cálculo del árbol de expansión de coste mínimo en un grafo no dirigido, etc.

Representación de MF-sets

- Cada subconjunto $\rightarrow$ un árbol:
  - Nodo: información del elemento.
  - La raíz es el representante.
- *Representación de árboles mediante apuntadores al padre*: el que apunte a sí mismo será raíz.
- MF-set: colección de árboles (bosque).
- Cada elemento $\rightarrow$ número de 1 a n + vector M: posición $i \equiv$ índice del padre de i.



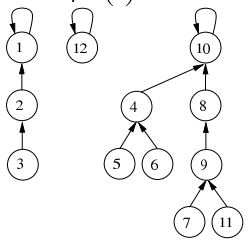
	1	2	3	4	5	6	7	8	9	10	11	12
M	1	1	2	4	4	4	9	10	8	10	9	12

Operaciones sobre MF-sets

Operación Unión *Merge*

Union(x,y): hacer que la raíz de un árbol apunte al nodo raíz del otro.

Suponiendo que x e y son raíces (representantes) $\Rightarrow$ modificar puntero al padre de uno de los representantes, $O(1)$.



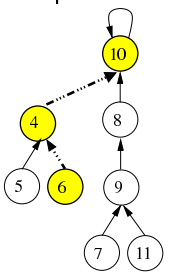
	1	2	3	4	5	6	7	8	9	10	11	12
M	1	1	2	10	4	4	9	10	8	10	9	12

Operaciones sobre MF-sets

Operación Buscar *Find*

Buscar(x): usando el puntero al padre, recorrer el árbol desde x hasta la raíz.
Los nodos visitados constituyen el **camino de búsqueda**.

Coste proporcional a la profundidad del nodo, $O(n)$.



	1	2	3	4	5	6	7	8	9	10	11	12
M	1	1	2	10	4	4	9	10	8	10	9	12

Análisis del coste temporal

MF-Set inicial: n subconjuntos de un elemento $\Rightarrow$ peor secuencia de operaciones:

1. realizar $n-1$ operaciones *Unión* $\rightarrow$ único conjunto de n elementos, y
2. realizar m operaciones *Buscar*.

Coste temporal:

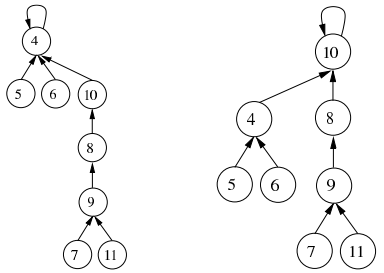
- $n-1$ operaciones *Unión* $\in O(n) \rightarrow m$ operaciones *Buscar* $\in O(mn)$.
- Coste determinado por cómo se hace la *Unión*, tras k operaciones *Unión* puede obtenerse un árbol de altura k.

$\Rightarrow$ Técnicas heurísticas para mejorar el coste (reduciendo altura del árbol).

Unión por altura o rango

Estrategia: unir de manera que la raíz del árbol más bajo apunte a la raíz del más alto.

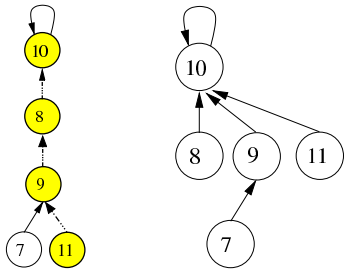
- Altura del árbol resultante: $max(h_1, h_2)$.
- Necesario mantener altura de cada nodo.
- La altura de un árbol de n elementos $\leq \lfloor logn \rfloor$.
- Coste de m operaciones de búsqueda: $O(m\ logn)$.



Compresión de caminos

Estrategia: Cuando se busca un elemento, hacer que todos los nodos del camino de búsqueda se enlacen directamente con la raíz.

- Combinando ambos heurísticos, coste de m operaciones de búsqueda: $O(m\alpha(m,n))$, con $\alpha(m,n) \equiv$ inversa de la función de Ackerman, (crecimiento lento). Normalmente, $\alpha(m,n) \leq 4$.
- En la práctica, con ambos heurísticos $\Rightarrow$ hacer m operaciones de búsqueda tiene un coste casi lineal en m.

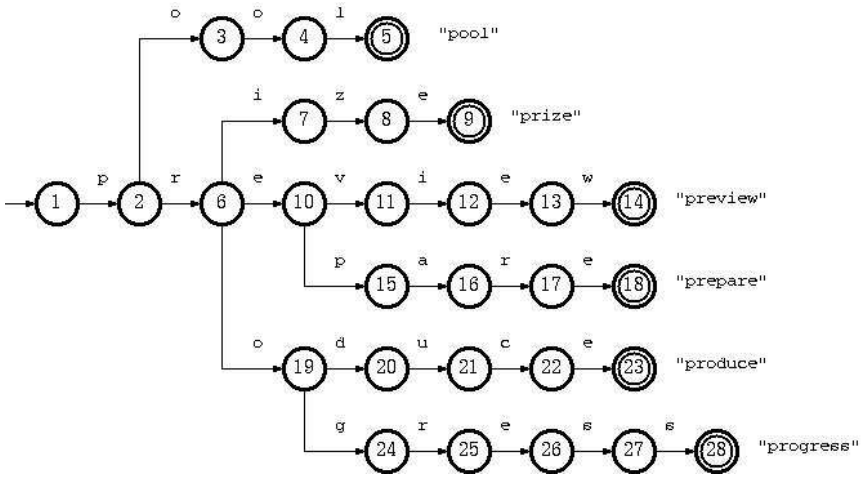


Tries

- Tipo de árbol de búsqueda.
- Aplicación: diccionarios.
- Palabras con prefijos comunes usan la misma memoria para dichos prefijos.
- *Compactación* de prefijos comunes $\Rightarrow$ ahorro de espacio.

Tries (II)

Ej: conjunto $\{pool, prize, preview, prepare, produce, progress\}$



Tries: Búsqueda de un elemento

- Se comienza en el nodo raíz.
- Desde el principio al final de la palabra, se toma caracter a caracter.
- Elegir la arista etiquetada con el mismo caracter.
- Cada paso consume un caracter y desciende un nivel.
- Si se agota la palabra y se ha alcanzado una hoja $\Rightarrow$ encontrado.
- Si en algún momento no existe ninguna arista con el caracter actual o se ha agotado la palabra y estamos en un nodo interno $\Rightarrow$ palabra no reconocida.

Tiempo de búsqueda proporcional a la longitud de la palabra $\longrightarrow$ estructura de datos muy eficiente.

Tries: Representación

Trie $\equiv$ un tipo de autómata finito determinista (AFD).

- Representación: matriz de transición.
  - Filas = estados.
  - Columnas = etiquetas.
- Cada posición de la matriz almacena el siguiente estado al que transicionar.
- Coste temporal muy eficiente.

La mayoría de nodos tendrán pocas aristas $\Longrightarrow$ gran cantidad de memoria desperdiciada.

Árboles Balanceados

- Estructuras de datos para almacenar elementos.
- Permiten una búsqueda eficiente.

Árbol balanceado: árbol donde ninguna hoja está más alejada de la raíz que cualquier otra hoja.

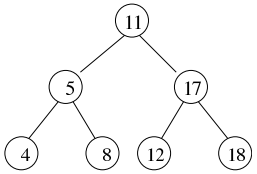
- Varios esquemas de balanceo (definición diferente de *más lejos*).
- Diferentes algoritmos para actualizar el árbol.

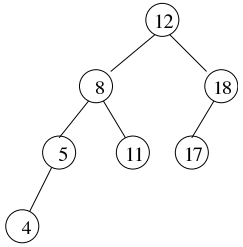
Árboles AVL

Árbol AVL: árbol binario de búsqueda que cumple

1. Las alturas de los subárboles de cada nodo difieren como mucho en 1.
2. Cada subárbol es un árbol AVL.

- Inventores: Adelsson, Velskii y Landis.
- No están balanceados perfectamente.
- Buscar, insertar y borrar un elemento $\in O(\log n)$.





Árboles 2-3

Árbol 2-3: árbol vacío (si tiene 0 nodos) o un nodo simple (si tiene un único nodo) o un árbol con múltiples nodos que cumplen:

1. Cada nodo interior tiene 2 o tres hijos.
2. Cada camino desde la raíz a una hoja tiene la misma longitud.

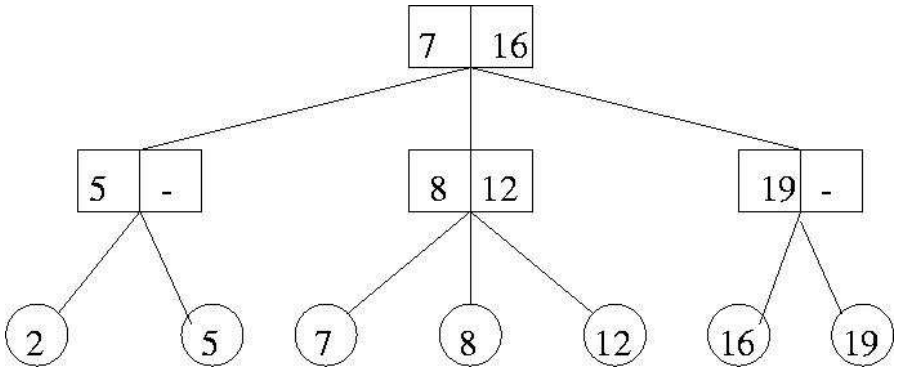
- Nodos internos:

p_1	k_1	p_2	k_2	p_3

  - 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 descendiente del segundo hijo.
  - k_2: clave más pequeña descendiente del tercer hijo.
- Hojas: información de la clave correspondiente.

Árboles 2-3 (II)

Ejemplo:



Árboles 2-3: Búsqueda

- Los valores en los nodos internos guían la búsqueda.
- Empezar en la raíz: k_1 y k_2 son los dos valores almacenados en la raíz.
  - Si $x < k_1$, seguir la búsqueda por el primer hijo.
  - 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 esquema a cada nodo que forme parte del camino de búsqueda.
- Fin: se llega a una hoja con
  - la clave $x \Longrightarrow$ elemento encontrado.
  - una clave diferente a $x \Longrightarrow$ elemento no encontrado.

B-árboles

- Generalización de los árboles 2-3.
- Aplicaciones:
  - Almacenamiento externo de datos.
  - Organización de índices en sistemas de bases de datos.
  - Permite minimizar los accesos a disco para consultas a bases de datos.

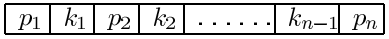
B-árboles (II)

Árbol B de orden n: árbol de búsqueda n-ario que cumple

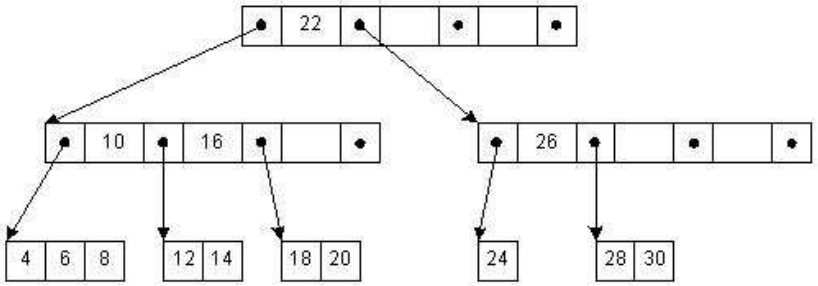
- La raíz es una hoja o tiene por lo menos dos hijos.
- Cada nodo, excepto el raíz y las hojas. tiene entre $\lceil \frac{n}{2} \rceil$ y n hijos.
- Cada camino desde la raíz a una hoja tiene la misma longitud.
- Cada nodo interno tiene hasta $(n-1)$ valores de claves y hasta m punteros a sus hijos.
- Los elementos se almacenan en los nodos internos y en las hojas.
- Un B-árbol puede verse como un índice jerárquico. La raíz sería el primer nivel de indizado.

B-árboles (III)

- Nodos internos:



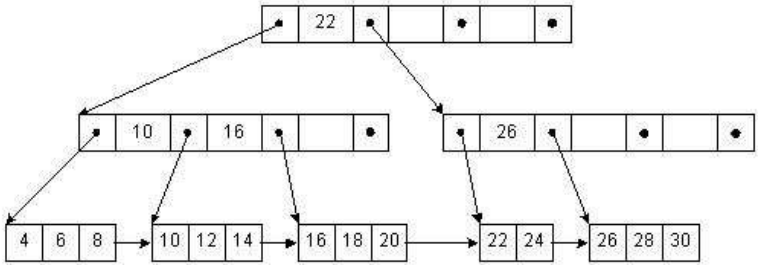
  - p_i es un puntero al i-ésimo hijo, $1 \leq i \leq n$.
  - k_i son los valores de las claves, ($k_1 < k_2 < \ldots < k_{n-1}$) de manera que:
    - todas las claves en el subárbol p_1 son menores que k_1.
    - Para $2 \leq i \leq n-1$, todas las claves en el subárbol p_i son mayores o iguales que k_{i-1} y menores que k_i.
    - Todas las claves en el subárbol p_n son mayores o iguales que k_{n-1}.



B-árboles +

Árbol B+: árbol B en el que las claves almacenadas en los nodos internos no son útiles (sólo se usan para búsquedas).

- Todas las claves de nodos internos están duplicadas en las hojas.
- Ventaja: las hojas están enlazadas secuencialmente y se puede acceder a la información de los elementos sin visitar nodos internos.
- Mejora la eficiencia de determinados métodos de búsqueda.

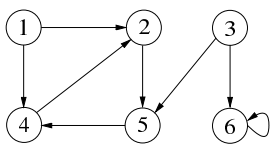


Estructuras de Datos y Algoritmos

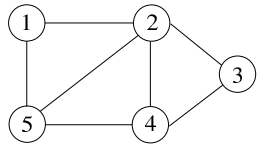
Grafos

Definiciones

- Grafo→ modelo para representar relaciones entre elementos de un conjunto.
- **Grafo**: (V,E), V es un conjunto de *vértices* o *nodos*, con una relación entre ellos; E es un conjunto de pares (u,v), $u,v \in V$, llamados *aristas* o *arcos*.
- **Grafo dirigido**: la relación sobre V no es simétrica. Arista $\equiv$ par ordenado (u,v).
- **Grafo no dirigido** la relación sobre V sí es simétrica. Arista $\equiv$ par no ordenado $\{u,v\}$, $u,v \in V$ y $u \neq v$



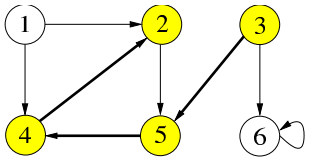
Grafo dirigido $G(V, E)$.
$V = \{1,2,3,4,5,6\}$
$E = \{(1,2),(1,4),(2,5),(3,5),(3,6),(4,2),$
$(5,4),(6,6)\}$



Grafo no dirigido $G(V, E)$.
$V = \{1,2,3,4,5\}$
$E = \{\{1,2\},\{1,5\},\{2,3\},\{2,4\},\{2,5\},$
$\{3,4\},\{4,5\}\}$

Definiciones (II)

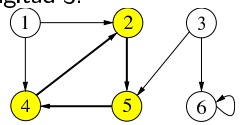
- **Camino** desde $u \in V$ a $v \in V$: secuencia $v_1, v_2, \ldots, v_k$ tal que $u = v_1$, $v = v_k$, y $(v_{i-1},v_i) \in E$, para $i = 2,\ldots,k$.
  Ej: camino desde 3 a 2 →<3,5,4,2>.



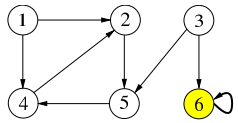
- **Longitud de un camino**: número de arcos del camino.
- **Camino simple**: camino en el que todos sus vértices, excepto, tal vez, el primero y el último, son distintos.

Definiciones (III)

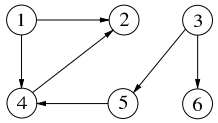
- **Ciclo**: camino simple $v_1, v_2, \ldots, v_k$ tal que $v_1 = v_k$.
  Ej: <2,5,4,2> es un ciclo de longitud 3.



- **Bucle**: ciclo de longitud 1.



- **Grafo acíclico**: grafo sin ciclos.

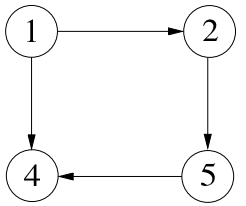


Definiciones (IV)

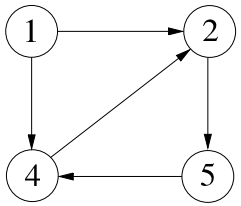
- v es **adyacente** a u si existe una arista $(u,v) \in E$.
- En un grafo no dirigido, $(u,v) \in E$ **incide** en los nodos u, v.
- En un grafo dirigido, $(u,v) \in E$ **incide** en v, y **parte** de u.
- **Grado** de un nodo: número de arcos que inciden en él.
- En grafos dirigidos existen el **grado de salida** y el **grado de entrada**. El grado del vértice será la suma de los grados de entrada y de salida.
- **Grado de un grafo**: máximo grado de sus vértices.

Definiciones (V)

- $G' = (V', E')$ es un **subgrafo** de $G = (V, E)$ si $V' \subseteq V$ y $E' \subseteq E$.
- **Subgrafo inducido** por $V' \subseteq V$: $G' = (V', E')$ tal que $E' = \{(u,v) \in E \mid u,v \in V'\}$. Ejemplos de subgrafos del grafo de la transparencia 1:



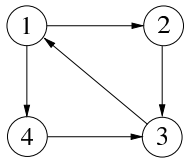
$V' = \{1,2,4,5\}$
$E' = \{(1,2),(1,4),(2,5),(5,4)\}$



Subgrafo inducido por
$V' = \{1,2,4,5\}$

Definiciones (VI)

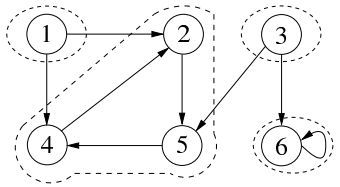
- v es **alcanzable desde** u, si existe un camino de u a v.
- Un grafo no dirigido es **conexo** si existe un camino desde cualquier vértice a cualquier otro.
- Un grafo dirigido con esta propiedad se denomina **fuertemente conexo**:



- Si un grafo dirigido no es fuertemente conexo, pero el grafo subyacente (sin sentido en los arcos) es conexo, el grafo es **débilmente conexo**.

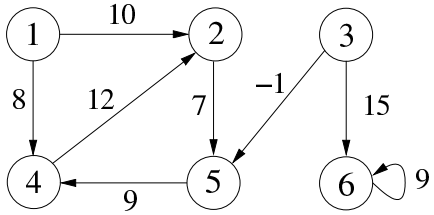
Definiciones (VII)

- En un grafo no dirigido, las **componentes conexas** son las clases de equivalencia según la relación "ser alcanzable desde".
- Un grafo no dirigido es **no conexo** si está formado por varias componentes conexas.
- En un grafo dirigido, las **componentes fuertemente conexas**, son las clases de equivalencia según la relación "ser mutuamente alcanzable".
- Un grafo dirigido es **no fuertemente conexo** si está formado por varias componentes fuertemente conexas.



Definiciones (VIII)

- **Grafo ponderado o etiquetado**: cada arco, o cada vértice, o los dos, tienen asociada una etiqueta.



Introducción a la teoría de grafos

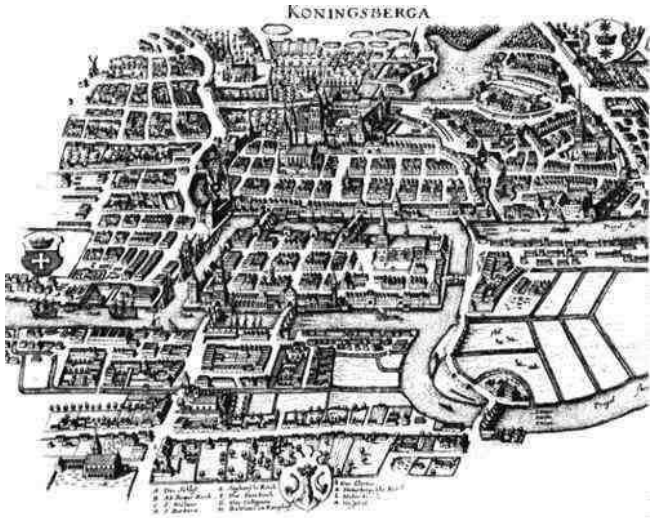
problema → representación con grafos → algoritmo → computador

Los puentes de Königsberg

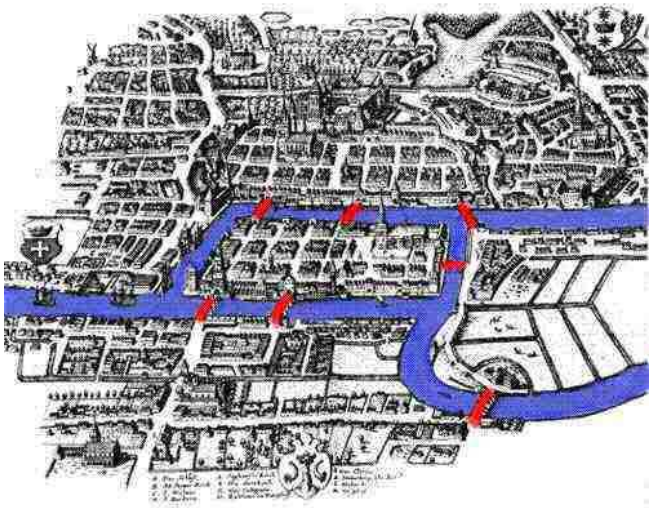
- Una isla en el centro del río.
- Siete puentes que enlazan distintos barrios.

Problema: planificar paseo de forma que saliendo de un punto se pueda volver a él, habiendo pasado cada uno de los puentes una vez y sólo una.

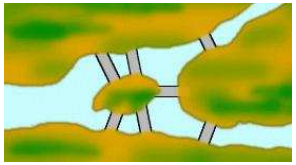
Los puentes de Königsberg (II)



Los puentes de Königsberg (III)

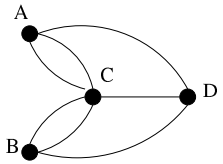


Los puentes de Königsberg (IV)



Traducción a grafos:

- representar islas y orillas con puntos.
- transformar puentes en líneas que enlacen los puntos.



Nuevo problema: ¿se puede dibujar la figura a partir de un punto y volver a él, sin repasar líneas y sin levantar el lápiz?

Los puentes de Königsberg (V)

Euler descubrió las siguientes reglas:

1. Un grafo compuesto de vértices solo de grado par se puede recorrer en una sola pasada, partiendo de un vértice y regresando al mismo.
2. Un grafo con sólo dos vértices de grado impar puede recorrerse en una sola pasada, pero sin volver al punto de partida.
3. Un grafo con un número de vértices de grado impar superior a 2 no se puede recorrer en una sola pasada.

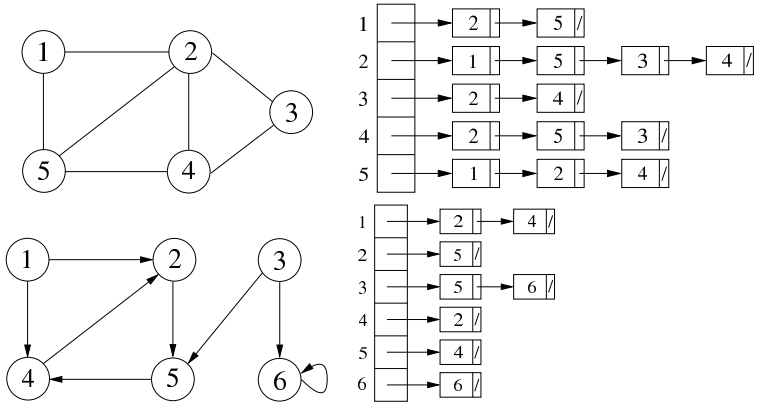
Los puentes de Königsberg:

VÉRTICE	GRADO
A	3
B	3
C	5
D	3

$\Rightarrow$ ¡el problema no tiene solución!

Representación de grafos: Listas de adyacencia

- $G = (V,E)$: vector de tamaño $|V|$.
- Posición $i \rightarrow$ puntero a una lista enlazada de elementos (*lista de adyacencia*). Los elementos de la lista son los vértices adyacentes a i

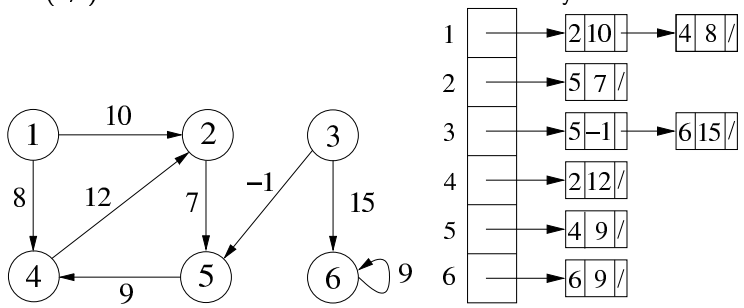


Listas de adyacencia (II)

- Si G es dirigido, la suma de las longitudes de las listas de adyacencia será $|E|$.
- Si G es no dirigido, la suma de las longitudes de las listas de adyacencia será $2|E|$.
- Coste espacial, sea dirigido o no: $O(|V| + |E|)$.
- Representación apropiada para grafos con $|E|$ menor que $|V|^2$.
- **Desventaja**: si se quiere comprobar si una arista (u,v) pertenece a $E \Rightarrow$ buscar v en la lista de adyacencia de u.
  Coste $O(Grado(G)) \subseteq O(|V|)$.

Listas de adyacencia (III)

- Representación extensible a grafos ponderados.
  El peso de (u,v) se almacena en el nodo de v de la lista de adyacencia de u.



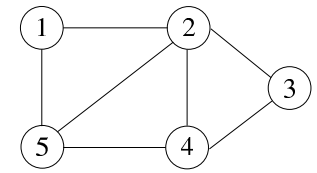
Listas de adyacencia (IV)

- Definición de tipos en C (grafos ponderados):

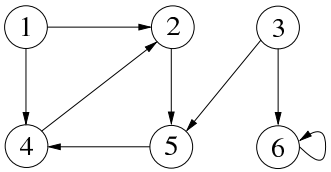
```
#define MAXVERT ...
typedef struct vertice{
 int nodo, peso;
 struct vertice *sig;
}vert_ady;
typedef struct{
 int talla;
 vert_ady *ady[MAXVERT];
}grafo;
```

Representación de grafos: Matriz de adyacencia

- $G = (V,E)$: matriz A de dimensión $|V| \times |V|$.
- 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

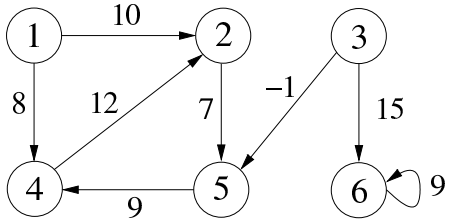
Matriz de adyacencia (II)

- Coste espacial: $O(|V|^2)$.
- Representación es útil para grafos con número de vértices pequeño, o grafos densos ($|E| \approx |V| \times |V|$).
- Comprobar si una arista (u,v) pertenece a $E \rightarrow$ consultar posición $A[u][v]$.
  Coste $O(1)$.

Matriz de adyacencia (III)

- Representación grafos ponderados:
  El peso de (i,j) se almacena en $A[i,j]$.

$$a_{ij} = \begin{cases} w(i,j) & \text{si } (i,j) \in E \\ 0 \text{ o } \infty & \text{en cualquier otro caso} \end{cases}$$



	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

Matriz de adyacencia (IV)

- Definición de tipos en C:

```
#define MAXVERT ...
typedef struct{
 int talla;
 int A[MAXVERT][MAXVERT];
}grafo;
```

Recorrido de grafos: recorrido primero en profundidad

→ Generalización del orden previo (preorden) de un árbol.

Estrategia:

- Partir de un vértice determinado v.
- Cuando se visita un nuevo vértice, explorar cada camino que salga de él.
  Hasta que no se ha finalizado de explorar uno de los caminos no se comienza con el siguiente.
- Un camino deja de explorarse cuando lleva a un vértice ya visitado.
- Si existían vértices no alcanzables desde v el recorrido queda incompleto:
  seleccionar alguno como nuevo vértice de partida, y repetir el proceso.

Recorrido primero en profundidad (II)

Esquema recursivo: dado $G = (V, E)$

1. Marcar todos los vértices como *no visitados*.
2. Escoger vértice u como punto de partida.
3. Marcar u como visitado.
4. $\forall v$ adyacente a u, $(u,v) \in E$, si v no ha sido visitado, repetir recursivamente (3) y (4) para v.
   - Finalizar cuando se hayan visitado todos los nodos alcanzables desde u.
   - Si desde u no fueran alcanzables todos los nodos del grafo: volver a (2), escoger un nuevo vértice de partida v no visitado, y repetir el proceso hasta que se hayan recorrido todos los vértices.

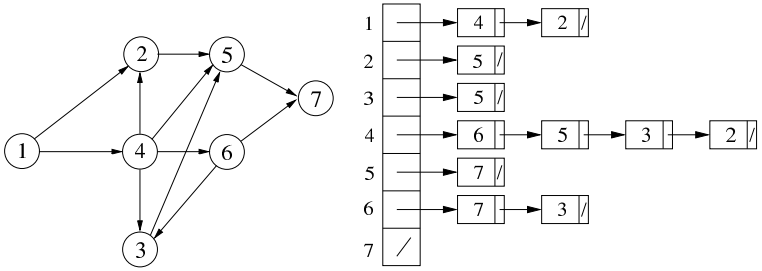
Recorrido primero en profundidad (III)

→ usar un vector *color* (talla $|V|$) para indicar si u ha sido visitado (color[u]=AMARILLO) o no (color[u]=BLANCO):

Algoritmo Recorrido_en_profundidad(G){
 para cada vértice $u \in V$
  color[u] = BLANCO
 fin_para
 para cada vértice $u \in V$
  si (color[u] = BLANCO) Visita_nodo(u)
 fin_para
}

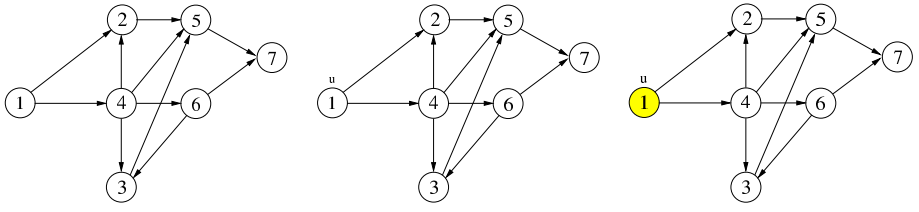
Algoritmo Visita_nodo(u){
 color[u] = AMARILLO
 para cada vértice $v \in V$ adyacente a u
  si (color[v] = BLANCO) Visita_nodo(v)
 fin_para
}

Recorrido primero en profundidad: Ejemplo

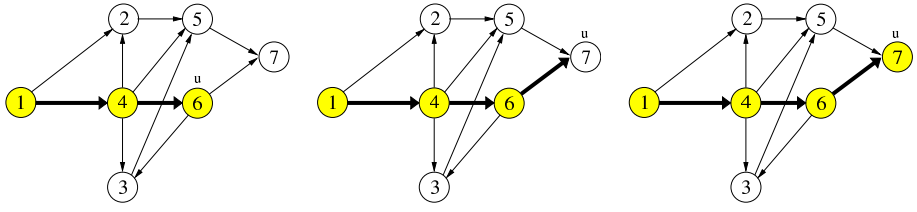


¡OJO!: el recorrido depende del orden en que aparecen los vértices en las listas de adyacencia.

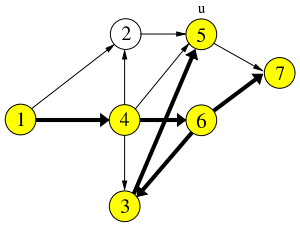
Recorrido primero en profundidad: Ejemplo (II)



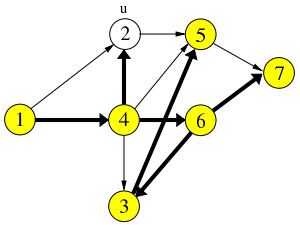
Recorrido primero en profundidad: Ejemplo (III)



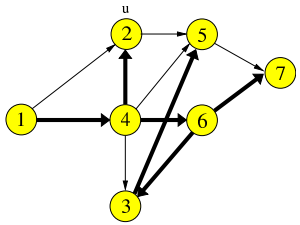
Recorrido primero en profundidad: Ejemplo (IV)



color[5]=AMARILLO



Visita_nodo(2)



color[2]=AMARILLO

Recorrido primero en profundidad: coste temporal

- $G = (V, E)$ se representa mediante listas de adyacencia.
- *Visita_nodo* se aplica únicamente sobre vértices no visitados → sólo una vez sobre cada vértice.
- *Visita_nodo* depende del número de vértices adyacentes que tenga u (longitud de la lista de adyacencia).
- coste de todas las llamadas a *Visita_nodo*:

$$\sum_{v \in V} |\text{ady}(v)| = \Theta(|E|)$$

- Añadir coste asociado a los bucles de *Recorrido_en_profundidad*: $O(|V|)$.

⇒ coste del recorrido en profundidad es $O(|V| + |E|)$.

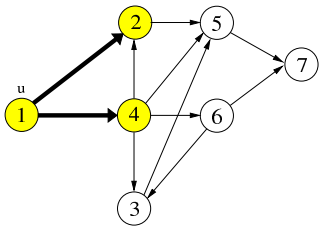
Recorrido de grafos: recorrido primero en anchura

→ Generalización del recorrido por niveles de un árbol.

Estrategia:

- Partir de algún vértice u, visitar u y, después, visitar cada uno de los vértices adyacentes a u.
- Repetir el proceso para cada nodo adyacente a u, siguiendo el orden en que fueron visitados.

Coste: $O(|V| + |E|)$.



Ordenación topológica

- Aplicación inmediata del recorrido en profundidad.
- $G = (V, E)$ dirigido y acíclico:
  la **ordenación topológica** es una permutación $v_1, v_2, v_3, \ldots, v_{|V|}$ de los vértices, tal que si $(v_i, v_j) \in E$, $v_i \neq v_j$, entonces v_i precede a v_j en la permutación.
- Ordenación no posible si G es cíclico.
- La ordenación topológica no es única.
- Una ordenación topológica es como una ordenación de los vértices a lo largo de una línea horizontal, con los arcos de izquierda a derecha.
- Algoritmo como el recorrido primero en profundidad. Utiliza una pila P en la que se almacena el orden topológico de los vértices.

Ordenación topológica (II)

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)
}

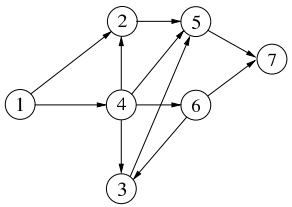
Algoritmo Visita_nodo(u){
 color[u] = AMARILLO
 para cada vértice $v \in V$ adyacente a u
  si (color[v] = BLANCO) Visita_nodo(v)
 fin_para
 apilar(P,u)
}

Ordenación topológica (III)

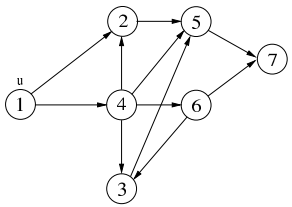
Finaliza Visita_nodo(u)
↓
Se han visitado los vértices alcanzables desde u
(los vértices que suceden a u en el orden topológico).
↓
Antes de acabar *Visita_nodo(u)* apilamos u
↓
El vértice se apila después de apilar todos los alcanzables desde él.
↓
Al final en P estarán los vértices ordenados topológicamente.

Coste: equivalente a recorrido primero en profundidad, $O(|V| + |E|)$.

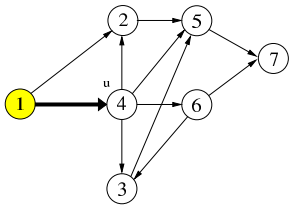
Ordenación topológica: Ejemplo



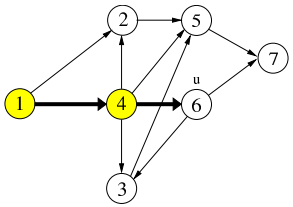
Orden_topologico(G)



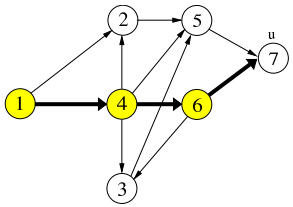
Visita_nodo(1)
$P = \{\}$



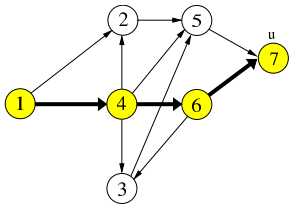
Visita_nodo(4)
$P = \{\}$



Visita_nodo(6)
$P = \{\}$

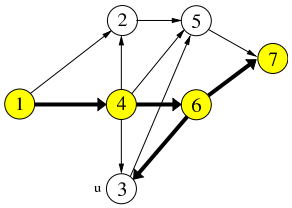


Visita_nodo(7)
$P = \{\}$

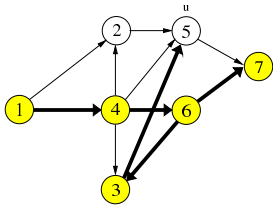


apilar(P,7)
$P = \{7\}$

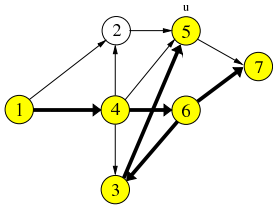
Ordenación topológica: Ejemplo (II)



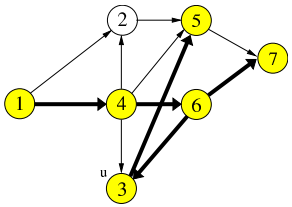
Visita_nodo(3)
$P = \{7\}$



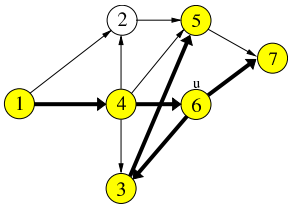
Visita_nodo(5)
$P = \{7\}$



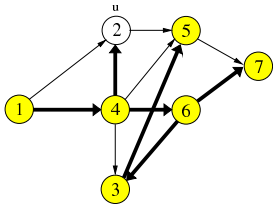
apilar(P,5)
$P = \{5,7\}$



apilar(P,3)
$P = \{3,5,7\}$

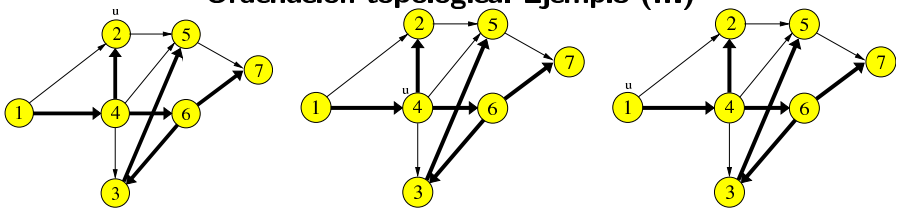


apilar(P,6)
$P = \{6,3,5,7\}$



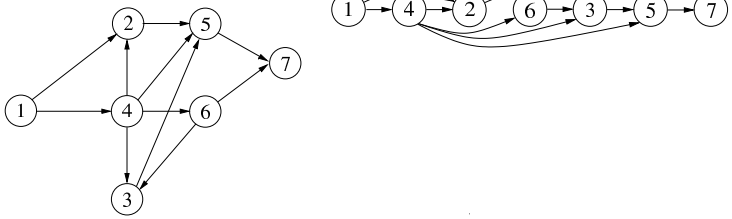
Visita_nodo(2)
$P = \{6,3,5,7\}$

Ordenación topológica: Ejemplo (III)



apilar(P,2) $P = \{2,6,3,5,7\}$ apilar(P,4) $P = \{4,2,6,3,5,7\}$ apilar(P,1) $P = \{1,4,2,6,3,5,7\}$

Ordenación de los vértices en una línea horizontal, con todos los arcos de izquierda a derecha:



Caminos de mínimo peso

$G = (V, E)$ dirigido y ponderado con el peso de cada arista $w(u,v)$.

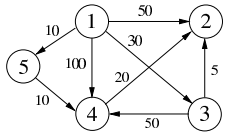
- **Peso de un camino** $p =< v_0, v_1, \ldots, v_k >$: la suma de los pesos de las aristas que lo forman:

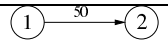
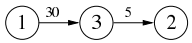
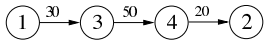
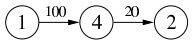
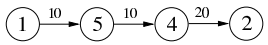
$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

- **Camino de mínimo peso** desde u a v: camino que tiene un peso menor entre todos los caminos de u a v, o ∞ si no existe camino de u a v.
- **Longitud** de un camino de u a v: peso de un camino de u a v.
- **Camino más corto** de u a v: camino de mínimo peso de u a v.

Caminos de mínimo peso (II)

Caminos desde 1 a 2:



Camino	Longitud (peso o coste)
	50
	35
	100
	120
	40

Caminos de mínimo peso (III)

- Usamos un grafo dirigido y ponderado para representar las comunicaciones entre ciudades:
  - vértice = ciudad.
  - arista (u,v) = carretera de u a v;
    el peso asociado a la arista es la distancia.
- Camino más corto = ruta más rápida.
- Variantes del cálculo de caminos de menor longitud:
  - 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.

Algoritmo de Dijkstra

Problema: $G = (V, E)$ grafo dirigido y ponderado con pesos no negativos; dado un vértice origen s, obtener los caminos más cortos al resto de vértices de V.

- Si existen aristas con pesos negativos, la solución podría ser errónea.
- Otros algoritmos permiten pesos negativos, siempre que no existan ciclos de peso negativo.
- Idea: explotar la propiedad de que el camino más corto entre dos vértices contiene caminos más cortos entre los vértices que forman el camino.

Algoritmo de Dijkstra (II)

El algoritmo de Dijkstra mantiene estos 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 $Q = V - S$ que mantiene, para cada vértice, la distancia más corta desde el origen pasando a través de vértices que pertenecen a S (Distancia provisional). Para guardar las distancias provisionales usaremos un vector $D[1..|V|]$, donde $D[i]$ indicará la distancia provisional desde el origen s al vértice i.
  Inicialmente, $D[u] = \infty \; \forall u \in V - \{s\}$ y $D[s] = 0$.

Además:

- Un vector $P[1..|V|]$ para recuperar los caminos mínimos calculados.
  $P[i]$ almacena el índice del vértice que precede al vértice i en el camino más corto desde s hasta i.

Algoritmo de Dijkstra (III)

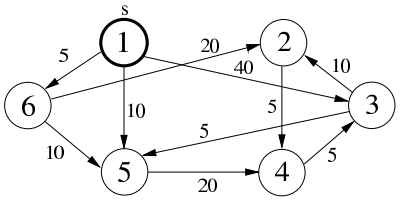
Estrategia:

1. Extraer de Q el vértice u cuya distancia provisional $D[u]$ sea menor.
   $\rightarrow$ esta distancia es la menor posible entre el vértice origen s y u.
   La distancia provisional se correspondía con el camino más corto utilizando vértices de S.
   $\Rightarrow$ ya no es posible encontrar un camino más corto desde s hasta u utilizando algún otro vértice del grafo
2. Insertar u, para el que se ha calculado el camino más corto desde s, en S ($S = S \cup \{u\}$).
   Actualizar las distancias provisionales de los vértices de Q adyacentes a u que mejoren usando el nuevo camino.
3. Repetir 1 y 2 hasta que Q quede vacío
   $\Rightarrow$ en D se tendrá, para cada vértice, la distancia más corta desde el origen.

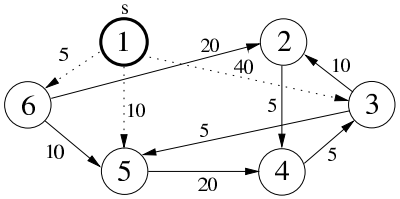
Algoritmo Dijkstra(G, w, s)
{
 para cada vértice $v \in V$ **hacer**
  $D[v] = \infty$;
  $P[v] = NULO$;
 fin_para
 $D[s] = 0$; $S = \emptyset$; $Q = V$;
 mientras $Q \neq \emptyset$ **hacer**
  u = extract_min(Q); /* según D */
  $S = S \cup \{u\}$;
  para cada vértice $v \in V$ adyacente a u **hacer**
   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
}

Algoritmo de Dijkstra: Ejemplo

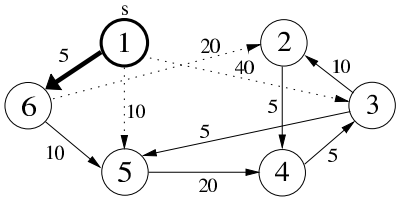
- Vértice origen 1.
- Aristas con trazo discontínuo = caminos provisionales desde el origen a los vértices.
- Aristas con trazo grueso = caminos mínimos ya calculados.
- Resto de aristas con trazo fino.



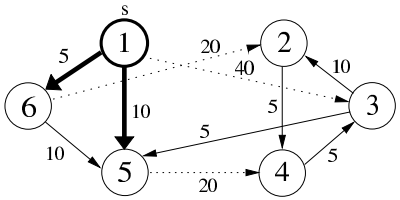
S	Q	u		1	2	3	4	5	6
{}	{1,2,3,4,5,6}	–	D	0	∞	∞	∞	∞	∞
			P	*NULO*	*NULO*	*NULO*	*NULO*	*NULO*	*NULO*



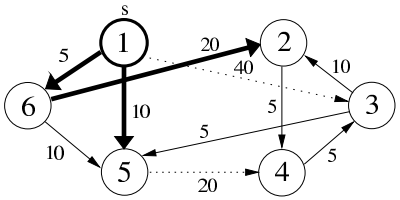
S	Q	u		1	2	3	4	5	6
{1}	{2,3,4,5,6}	1	D	0	∞	40	∞	10	5
			P	*NULO*	*NULO*	1	*NULO*	1	1



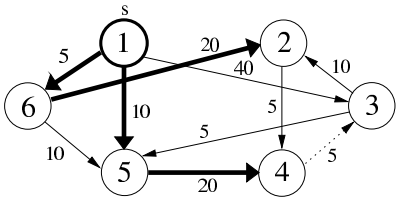
S	Q	u		1	2	3	4	5	6
{1,6}	{2,3,4,5}	6	D	0	25	40	∞	10	5
			P	*NULO*	6	1	*NULO*	1	1



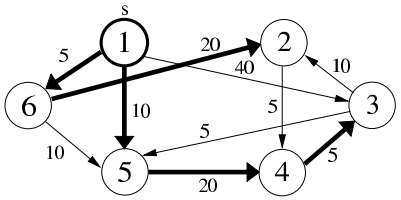
S	Q	u		1	2	3	4	5	6
{1,6,5}	{2,3,4}	5	D	0	25	40	30	10	5
			P	*NULO*	6	1	5	1	1



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

- Consideración: representación mediante listas de adyacencia.
- El bucle *mientras* se repite hasta que el conjunto Q queda vacío.
  Inicialmente Q tiene todos los vértices y en cada iteración se extrae uno de Q
  $\rightarrow |V|$ iteraciones.
- El bucle *para* interno al bucle *mientras* se repite tanto como vértices adyacentes tenga el vértice extraído de Q.
  El número total de iteraciones del bucle *para* será el número de vértices adyacentes de los vértices del grafo.
  $\rightarrow$ El número de arcos del grafo: $|E|$.
- La operación *extract_min* $\in O(|V|)$ (conjunto Q = vector).
  extract_min se realiza $|V|$ veces.
  Coste total de *extract_min* $\in O(|V|^2)$.

Sumando costes del bucle *para* y *extract_min*
$\Rightarrow$ Coste Dijkstra $\in O(|V|^2 + |E|) = O(|V|^2)$

Coste temporal del algoritmo (II)

Optimización: *extract_min* sería más eficiente si Q fuese un montículo:

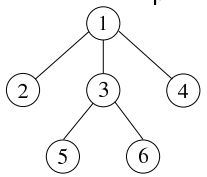
- Construir el montículo $\in O(|V|)$.
- La operación *extract_min* $\in O(\log |V|)$
  extract_min se realiza $|V|$ veces.
  Coste total de *extract_min* $\in O(|V| \log |V|)$.
- El bucle *para* supone modificar la prioridad (distancia provisional) del vértice en el montículo y reorganizar el montículo $\in O(\log |V|)$.
  $\rightarrow$ cada iteración del bucle *para* $\in O(\log |V|)$.
  El bucle *para* se realiza $|E|$ veces $\rightarrow O(|E| \log |V|)$.

$\Rightarrow$ Coste Dijkstra $O(|V| + |V| \log |V| + |E| \log |V|) = O((|V| + |E|) \log |V|)$.

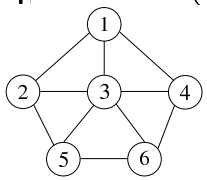
Árbol de expansión de coste mínimo

- **Árbol libre**: grafo no dirigido, acíclico y conexo. Propiedades:

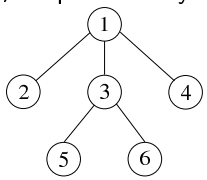
1. Un árbol libre con $n \geq 1$ nodos tiene $n - 1$ arcos.
2. Si se añade una nueva arista, se crea un ciclo.
3. Cualquier par de de vértices están conectados por un único camino.



- **Árbol de expansión** de $G = (V, E)$: árbol libre $T = (V', E')$, tal que $V' = V$ y $E' \subseteq E$.



$\rightarrow$ Árbol de expansión $\rightarrow$

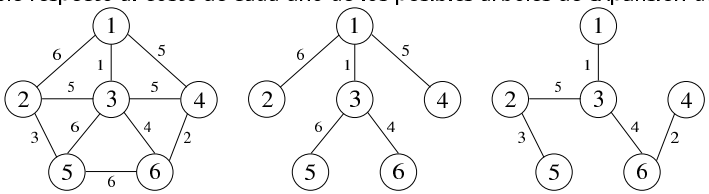


Árbol de expansión de coste mínimo (II)

- **Coste de un árbol de expansión** T: 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 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 del grafo.



Coste 22

Coste mínimo: 15

Problema: Sea $G = (V, E)$ un grafo no dirigido, conexo y ponderado; obtener $G' = (V,E')$ donde $E' \subseteq E$ tal que G' sea un árbol de expansión de coste mínimo de G.

Algoritmo de Kruskal

El algoritmo mantiene estas estructuras:

- Un conjunto A que mantiene las aristas que ya han sido seleccionadas como pertenecientes al árbol de expansión de coste mínimo. Al final A contendrá el subconjunto de aristas que forman el árbol de expansión decoste mínimo.
- Un MF-set que se usa 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:
  - Conforme se añaden aristas al conjunto A, se unen vértices entre sí formando árboles.
  - Estos árboles se enlazarán entre sí hasta obtener un único árbol que será el árbol de expansión de coste mínimo.

Algoritmo de Kruskal (II)

Estrategia:

1. Inicialmente:
   - $A = \emptyset$
   - el MF-set tendrá $|V|$ subconjuntos cada uno de los cuales contendrá uno de los vértices.
2. De entre todas las aristas (u,v), seleccionar como candidata para el AECM aquélla no seleccionada todavía y con menor peso:
   - Si u y v no están conectados entre sí en el AECM (provocarán incremento de coste mínimo) $\rightarrow$ unir los vértices conectados con v con los vértices conectados con u y añadir (u,v) a A.
   - Si u y v ya están conectados entre sí, no se realiza ninguna acción.
3. Repetir (2) una vez para cada arista (u,v).

Algoritmo de Kruskal (III)

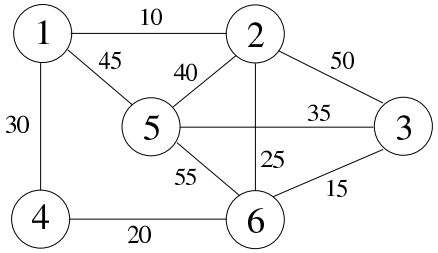
Dado $G = (V, E)$ no dirigido, conexo y ponderado:

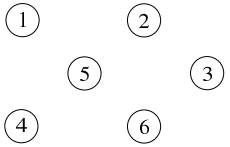
Algoritmo Kruskal(G)
{
 $A = \emptyset$
 para cada vértice $v \in V$ **hacer**
  Crear_Subconjunto(v)
 fin_para
 ordenar las aristas pertenecientes a E, según su peso, en orden no decreciente
 para cada arista $(u,v) \in E$, siguiendo el orden no decreciente **hacer**
  si Buscar(u) $\neq$ Buscar(v) **entonces**
   $A = A \cup \{(u,v)\}$
   Union(Buscar(u),Buscar(v))
  fin_si
 fin_para
 devuelve(A)
}

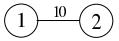
Algoritmo de Kruskal: Ejemplo

Para cada iteración se muestran:

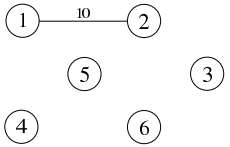
- El estado del conjunto A y del MF-set.
- El bosque que se va obteniendo durante la construcción del AECM.
- Qué arista ha sido seleccionada para formar parte del árbol y las acciones llevadas a cabo.

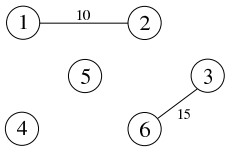


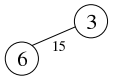
A	MF-set (componentes conexas)	Árbol de coste mínimo
{}	{{1},{2},{3},{4},{5},{6}}	



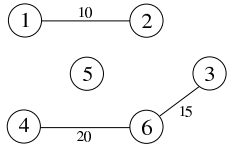
Buscar(1) $\neq$ Buscar(2) $\longrightarrow$ $A = A \cup \{(1,2)\}$; Union(Buscar(1),Buscar(2))

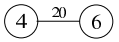
A	MF-set (componentes conexas)	Árbol de coste mínimo
{(1,2)}	{{1,2},{3},{4},{5},{6}}	

A	MF-set (componentes conexas)	Árbol de coste mínimo
{(1,2),(6,3)}	{{1,2},{6,3},{4},{5}}	



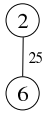
Buscar(6) $\neq$ Buscar(3) $\longrightarrow$ $A = A \cup \{(6,3)\}$; Union(Buscar(6),Buscar(3))

A	MF-set (componentes conexas)	Árbol de coste mínimo
{(1,2),(6,3),(4,6)}	{{1,2},{4,6,3},{5}}	

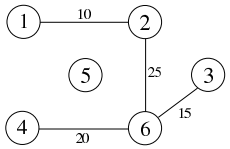


Buscar(4) $\neq$ Buscar(6) $\longrightarrow$ $A = A \cup \{(4,6)\}$; Union(Buscar(4),Buscar(6))

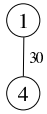
A	MF-set (componentes conexas)	Árbol de coste mínimo



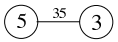
Buscar(2) $\neq$ Buscar(6) $\longrightarrow A = A \cup \{(2,6)\}$; Union(Buscar(2),Buscar(6))

$\{(1,2),(6,3),(4,6),(2,6)\}$	$\{\{1,2,4,6,3\},\{5\}\}$	

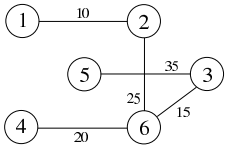
A	MF-set (componentes conexas)	Árbol de coste mínimo



Buscar(1) = Buscar(4)



Buscar(5) $\neq$ Buscar(3) $\longrightarrow A = A \cup \{(5,3)\}$; Union(Buscar(5),Buscar(3))

$\{(1,2),(6,3),(4,6),(2,6),(5,3)\}$	$\{\{5,1,2,4,6,3\}\}$	

Coste temporal del algoritmo

Asunción: Las operaciones *Unión* y *Buscar* aplican *unión por rango* y *compresión de caminos*.

- Construcción de $|V|$ subconjuntos disjuntos $\in O(|V|)$.
- *Unión* y *Buscar* para cada arista del grafo $\in O(|E|)$.
- Selección de aristas en orden no decreciente.
  Organizar aristas mediante un montículo:
  - Extraer arista de mínimo peso $\in O(\log |E|)$.
  - $|E|$ operaciones de extracción $\rightarrow O(|E| \log |E|)$.
  - Construcción del montículo $\in O(|E|)$.

$\Rightarrow$ Coste Kruskal $\in O(|V| + |E| + |E| \log |E| + |E|) = O(|E| \log |E|)$.

Algoritmo de Prim

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.

Algoritmo de Prim (II)

Idea:

- Toma como raíz un vértice cualquiera, u, y a partir de él extenderse por todos los vértices.
- En cada paso seleccionar el arco de menor peso que une un vértice presente en S con uno de V que no lo esté.

Estrategia: empezando por cualquier vértice, construir incrementalmente un árbol seleccionando en cada paso una arista $(u,v) \in A$ tal que:

- Si se añade (u,v) al conjunto A obtenido hasta ahora no se cree ningún ciclo.
- Produzca el menor incremento de peso posible.
- Los arcos del árbol de expansión parcial formen un único árbol.

Algoritmo de Prim (III)

Dado $G = (V, E)$ no dirigido, conexo y ponderado:

Algoritmo Prim(G)
{
$A = \emptyset$
$u =$ elemento arbitrario de V
$S = \{u\}$
mientras $S \neq V$ **hacer**
$(u,v) = \text{argmin}_{(x,y) \in S \times (V-S)}\, \text{peso}(x,y)$
$A = A \cup \{(u,v)\}$
$S = S \cup \{v\}$
fin_mientras
devuelve(A)
}

Algoritmo de Prim (IV)

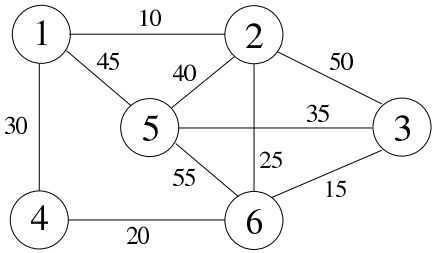
Donde la operación de búsqueda de la arista factible de menor peso, $(u,v) = \text{argmin}_{(x,y) \in S \times (V-S)}\, \text{peso}(x,y)$ se calcula:

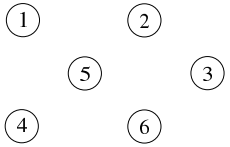
$m = +\infty$
para $x \in S$ **hacer**
  para $y \in V - S$ **hacer**
    si peso$(x,y) < m$ **entonces**
      $m =$ peso(x,y)
      $(u,v) = (x,y)$
    fin_si
  fin_para
fin_para

Algoritmo de Prim: Ejemplo

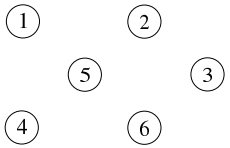
Para cada iteración se muestran:

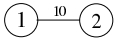
- El estado del conjunto A y del conjunto S.
- Qué arista ha sido la seleccionada para formar parte del árbol y las acciones llevadas a cabo.



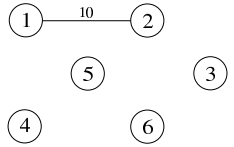
A	S	Árbol de coste mínimo
{}	{}	

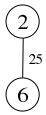
$A = \emptyset$; $u =$ elemento arbitrario de V (Ej: u=1); $S = \{1\}$;

{}	{1}	

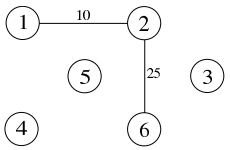
A	S	Árbol de coste mínimo
		

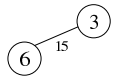
argmin $\longrightarrow (u,v) = (1,2)$; $A = A \cup \{(1,2)\}$; $S = S \cup \{2\}$;

{(1,2)}	{1,2}	

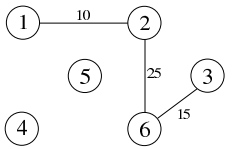
A	S	Árbol de coste mínimo
		

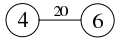
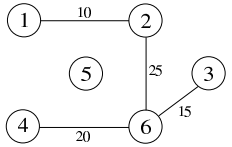
argmin $\longrightarrow (u,v) = (2,6)$; $A = A \cup \{(2,6)\}$; $S = S \cup \{6\}$;

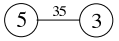
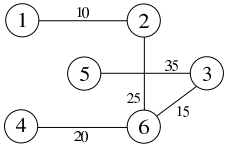
{(1,2),(2,6)}	{1,2,6}	

A	S	Árbol de coste mínimo
		

argmin $\longrightarrow (u,v) = (6,3)$; $A = A \cup \{(6,3)\}$; $S = S \cup \{3\}$;

{(1,2),(2,6),(6,3)}	{1,2,6,3}	

A	S	Árbol de coste mínimo
		
argmin $\longrightarrow$ $(u,v) =$ (6,4); $A = A \cup \{(6,4)\}$; $S = S \cup \{4\}$;		
{(1,2),(2,6),(6,3),(6,4)}	{1,2,6,3,4}	

A	S	Árbol de coste mínimo
		
argmin $\longrightarrow$ $(u,v) =$ (3,5); $A = A \cup \{(3,5)\}$; $S = S \cup \{5\}$;		
{(1,2),(2,6),(6,3),(6,4),(3,5)}	{1,2,6,3,4,5}	

Coste temporal del algoritmo

- Coste de $(u,v) =$ argmin$_{(x,y) \in S \times (V-S)}$ peso(x,y):
  - Dos bucles **para** anidados que recorren los subconjuntos de vértices que crean $\in O(|V|^2)$.
- bucle **mientras** que engloba el cálculo de la arista factible de menor peso. $|V| - 1$ iteraciones.

$\Rightarrow$ Coste $\in O(|V|^3)$.

- $\exists$ optimizaciones para $(u,v) =$ argmin$_{(x,y) \in S \times (V-S)}$ peso(x,y) con un coste $\in O(|V|)$

$\Rightarrow$ Coste $\in O(|V|^2)$.

Estructuras de Datos y Algoritmos

Algoritmos Voraces

Introducción

Objetivos:

- Estudiar el diseño de algoritmos voraces.
- Estudiar algunos problemas clásicos.

Aplicación:

- Problemas de optimización $\longrightarrow$ búsqueda del valor óptimo de una función objetivo.

Introducción (II)

- Solución = secuencia de decisiones.
- Toma de decisiones:
  - En base a una medida local de optimización.
  - Irreversibles.
- Solución subóptima.
- Coste computacional bajo.
- Estrategia voraz: realizar la elección que es mejor en ese momento.
  Criterio de manejo de datos.

Ejemplo: Cajero automático

Problema: suministrar cantidad de dinero solicitada usando sólo los tipos de billetes especificados, de manera que el número de billetes sea mínimo.

Ejemplo:

- Cantidad solicitada $M = 1100$ Euros.
- Billetes disponibles: $\{100, 200, 500\}$.
- Posibles soluciones:
  - 11×100 (11 billetes).
  - $5 \times 200 + 1 \times 100$ (6 billetes).
  - $2 \times 500 + 1 \times 100$ (3 billetes).

Ejemplo: Cajero automático (II)

Estrategia voraz: seleccionar siempre el billete de mayor valor, si quedan billetes de ese tipo y al añadirlo no nos pasamos de la cantidad solicitada.

Posible casos:

- No hay solución. Ej: $M = 300$ y no hay billetes de 100.
- La solución no se encuentra. Ej: $M = 1100$ y no hay billetes de 100.
  - Algoritmo $\longrightarrow 2 \times 500 \longrightarrow$ STOP.
  - Solución: $1 \times 500 + 3 \times 200$.
- Encuentra una solución no óptima.
  Ej: disponemos de billetes $\{10, 50, 110, 500\}$ y $M = 650$.
  - Algoritmo $\longrightarrow 1 \times 500 + 1 \times 110 + 4 \times 10$ (6 billetes).
  - Solución óptima: $1 \times 500 + 3 \times 50$ (4 billetes).

Esquema general Voraz

Notación:

C : Conjunto de elementos o candidatos a elegir.

S : Conjunto de elementos de la solución en curso.

solución(S) : indica si S es solución o no.

factible(S) : indica si S es factible o completable.

selecciona(C) : selecciona un candidato de C conforme al criterio definido.

f : función objetivo a optimizar.

Esquema general Voraz (II)

Buscar subconjunto de C que optimice f:

1. Seleccionar en cada instante un candidato.
2. Añadir candidato a la solución en curso si es completable, sino rechazarlo.
3. Repetir 1 y 2 hasta obtener la solución.

   NOTA IMPORTANTE:

   - no siempre se encuentra la solución óptima.
   - decisiones irreversibles.

Esquema general Voraz (III)

Algoritmo Voraz(C) {
 $S = \emptyset$; /* conjunto vacio */
 while ((!solución(S)) && ($C \neq \emptyset$)) {
  $x =$ selecciona(C);
  $C = C$ - $\{x\}$
  if (factible($S \cup \{x\}$)) {
   $S = S \cup \{x\}$;
  }
 }
 if (solución(S)) **return**(S);
 else return(No hay solución);
}

Ejemplo: cajero automático:

- $C \equiv$ conjunto de billetes disponibles, {24 de 100 Euros, 32 de 200 Euros, etc.}.
- **factible(S)** calcula si el nuevo billete se pasa de la cantidad total solicitada.
- etc.

El problema de la compresión de ficheros

Fichero con 100.000 caracteres (a, b, c, d, e, f).

Frecuencias de aparición:

caracter	a	b	c	d	e	f
Frecuencia $\times 10^3$	45	13	12	16	9	5

Codificación fija:

caracter	a	b	c	d	e	f
Código Fijo	000	001	010	011	100	101

Tamaño del fichero $\longrightarrow$ 300 Kbits.

Codificación variable:

caracter	a	b	c	d	e	f
Código Variable	0	101	100	111	1101	1100

Tamaño del fichero $\longrightarrow$ 224 Kbits.

Compresión de ficheros (II)

Algoritmo de compresión:

1. Partir de la secuencia de bits original.
2. Obtener secuencia de caracteres asociada.
3. Codificar con códigos variables.

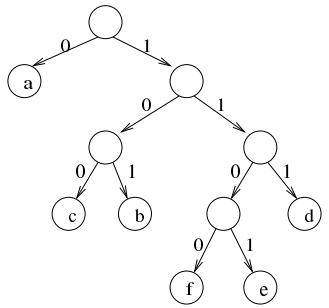
Ejemplo:

$$000001010 \overset{fijo}{\longrightarrow} abc \overset{variable}{\longrightarrow} 0101100$$

- La codificación variable cumple la propiedad de prefijo: ningún código es prefijo de otro.
- Cada caracter queda determinado por su secuencia de bits asociada.

Compresión de ficheros (III)

- **Código de Huffman:** cumple propiedad de prefijo.
- Usado para compresión de ficheros.
- Pretende eliminar redundancia en la codificación de un mensaje.
- Código generado a partir del fichero a comprimir.
- Algoritmo voraz para crear el código de Huffman para un fichero.



Compresión de ficheros (IV)

Algoritmo para construir el árbol binario (código de Huffman):

- $C =$ conjunto de símbolos.
- $F =$ conjunto de frecuencias.
- $Q =$ montículo que guardará el conjunto de frecuencias (MINHEAP).

Estrategia:

1. Extraer las dos frecuencias menores del Q y colgarlas como hijo izquierdo e hijo derecho de un nuevo nodo.
2. Asociar como frecuencia al nuevo nodo creado la suma de las frecuencias de sus hijos.
3. Insertar la frecuencia del nuevo nodo en Q.
4. Repetir 1, 2 y 3 hasta que no queden más nodos que juntar.

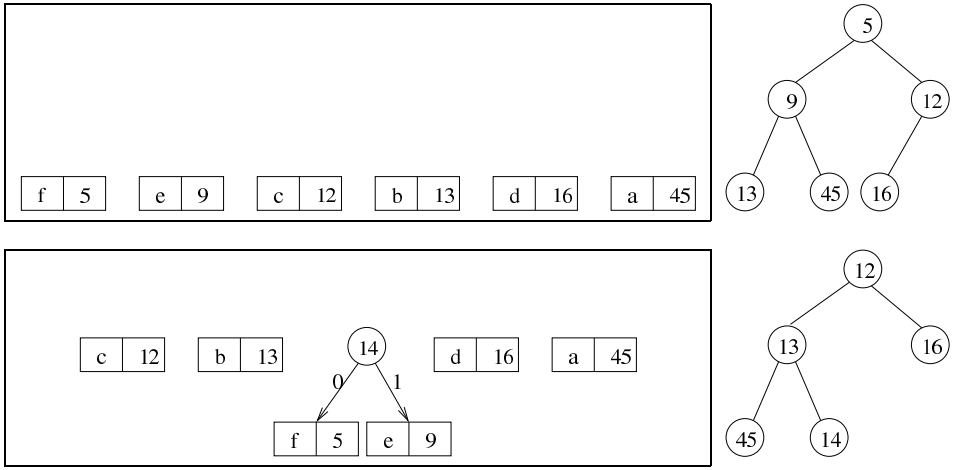
Finalizará cuando el árbol construido incluya a todos los símbolos.

```
Algoritmo Huffman(C, F) {
 for (x ∈ C, f_x ∈ F) {
  crear_hoja(x,f_x);
 }
 Q ⟵ F;
 build_heap(Q);
 for (i=1;i< |C|;i++) {
  z = crear_nodo();
  z.hizq = minimo(Q);
  extract_min(Q);
  z.hder = minimo(Q);
  extract_min(Q);
  z.frec = z.hizq.frec + z.hder.frec;
  insertar(Q, z);
 }
 return(minimo(Q));
}
```

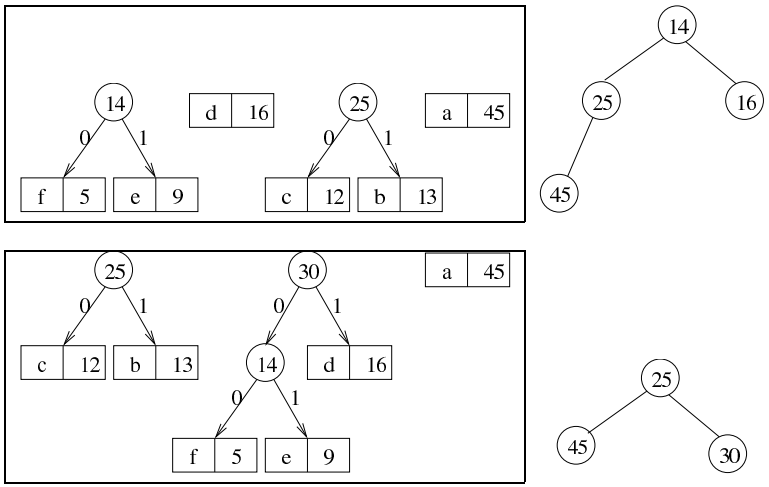
Coste $\in O(n \log n)$, $n \equiv$ talla del vocabulario.

Compresión de ficheros: Ejemplo

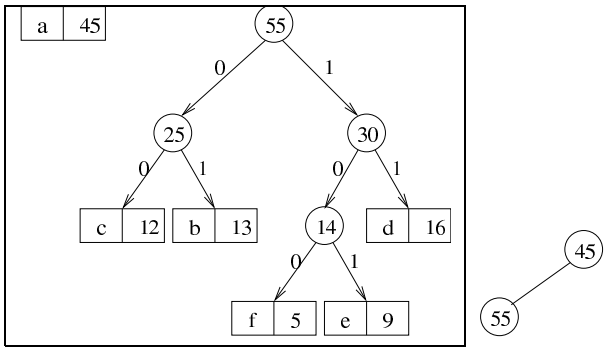
Traza para el ejemplo del fichero anterior:



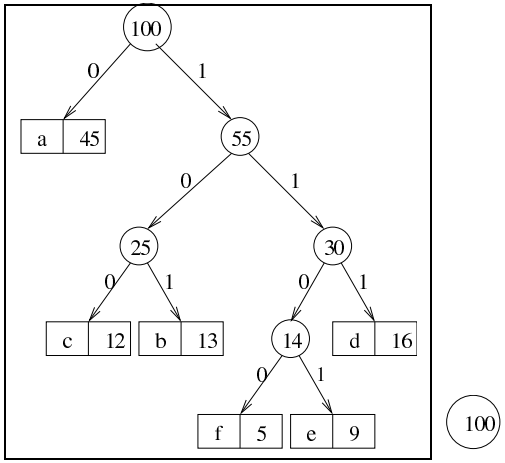
Compresión de ficheros: Ejemplo (II)



Compresión de ficheros: Ejemplo (III)



Compresión de ficheros: Ejemplo (IV)



El problema de la mochila con fraccionamiento

- Tenemos una mochila de capacidad M (en peso), y N objetos para incluir en ella.
- Cada objeto tiene un peso p_i y un beneficio b_i, $1 \leq i \leq N$.
- Los objetos se pueden fraccionar.
- No se puede sobrepasar la capacidad de la mochila.

Problema: ¿Cómo llenar la mochila de forma que el beneficio total sea máximo?

Planteamiento del problema:

- Buscamos secuencia de N valores $(x_1, \ldots, x_N)$, donde $0 \leq x_i \leq 1$, para $1 \leq i \leq N$. ($x_i \equiv$ parte fraccional tomada del objeto i).
- **Objetivo:** maximizar el beneficio $\sum_{i=1}^{N} b_i x_i$ con la restricción de que los objetos quepan, es decir, $\sum_{i=1}^{N} p_i x_i \leq M$

El problema de la mochila con fraccionamiento (II)

Ejemplo: $M = 20$, pesos de los objetos = (18, 15, 10), beneficios asociados = (25, 24, 15).

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

- La solución óptima deberá llenar la mochila al completo.
- Estrategia voraz: aplicamos criterio de optimización local.
- Podemos definir tres criterios de selección de objetos:
  - Escoger el elemento de mayor beneficio*.
  - Escoger el elemento de menor peso**.
  - Escoger el elemento de mayor relación beneficio/peso***.

El problema de la mochila con fraccionamiento (III)

Estrategia: coger elementos completos mientras no sobrepasemos el peso M de la mochila. Cuando no quepan elementos completos, del siguiente objeto que cumpla el criterio de selección, coger la parte fraccional correspondiente para llenar la mochila.

- $p =$ vector de pesos.
- $b =$ vector de beneficios.
- $M =$ tamaño de la mochila.
- $N =$ número de objetos.
- $C =$ conjunto que representa los objetos para elegir asignándoles identificadores.
- `solucion` = vector que almacena la solución.

```
Algoritmo Mochila-fraccional(p, b, M, N) {
  C = {1, 2, . . . , N};
  for (i = 1; i <= N; i++) solucion[i] = 0;
  while ((C ≠ ∅) && (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] ≤ M) {
      solucion[i] = 1;
      M = M - p[i];
    } else {
      solucion[i] = M/p[i];
      M = 0;
    }
  }
  return(solucion);
}
```

El problema de la mochila con fraccionamiento: Ejemplo

- $M = 50$.
- $p = (30, 18, 15, 10)$.
- $b = (25, 13, 12, 10)$.

1. Criterio: seleccionar objeto de mayor beneficio b_i.

Iteración	M	C	Solución			
	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

Beneficio $= 25 + 13 + (2/15) \cdot 12 = 39.6$

2. Criterio: seleccionar objeto de menor peso p_i.

Iteración	M	C	Solución			
	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

Beneficio $= (7/30) \cdot 25 + 13 + 12 + 10 = 40.83$

3. Criterio: seleccionar objeto de mayor beneficio unitario b_i/p_i = (5/30, 13/18, 12/15, 10/10).

Iteración	M	C	Solución			
	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

Beneficio $= 25 + (10/15) \cdot 12 + 10 =$ $\boxed{42.99}$

Coste temporal de Mochila-fraccional

- Bucle **for** que inicializa el vector $solucion \in O(N)$.
- Bucle **mientras** selecciona uno de los N objetos a cada iteración $\rightarrow$ num. de iteraciones $\in O(N)$.
  - Selección del elemento de C (representado en un vector) $\in O(N)$.
- Total bucle **mientras** $\in O(N^2)$.

$\Rightarrow$ Coste de Mochila-fraccional $\in O(N + N^2) \in O(N^2)$.

- *Implementación más eficiente*:
  Ordenar previamente los objetos conforme a la relación b/p:
  Selección $\in O(1)$ + Ordenación $\in O(N \log N)$.

$\Rightarrow$ Coste de Mochila-fraccional $\in O(N + N + N \log N) \in O(N \log N)$.