Estructura de Datos y Algoritmos

- Bibliografía:

Data Structures and Algorithms

Aho, Hopcroft, Ullman

Algorithms + Data Structures = Programs

N. Wirth

Handbook of Algorithms and Data Structures

Gonnet

Estructura de Datos y Algoritmos

- Bibliografía:

    The Art of Computer programming

        Knuth

    Structured Programming

        Dijkstra, Hoare, Dahl

Temas A.H.U. (por estructuras)

- Complejidad
- Listas : Stack, Colas, Mapeo
- Arboles: Generales, Binarios
- Conjuntos :Diccionarios, Hashing, Colas de Prioridad
- Arboles de Búsqueda

Temas A.H.U. (por estructuras)

- Grafos
- Ordenamiento
- Técnicas de Algoritmos
- Estructura de Archivos
- Manejo Memoria

Temas Gonnet (por algoritmos)

- Buscar :

  Búsqueda Secuencial: Arreglos, listas

  Búsqueda en arreglo ordenado

  Hashing : Arreglos, listas

  Búsqueda en estructuras recursivas :

  - Arbol Binario
  - Arbol B-Tree
  - Archivo indexado
  - Tries

Temas Gonnet (por algoritmos)

- Ordenar :

  Arreglos

  Listas

  Merging : Listas, arreglos

  Merging externo (archivos)

Temas Gonnet (por algoritmos)

- Seleccionar :

  Colas de Prioridad
  Listas
  Arboles de prioridad
  Heaps
  Pagodas
  Arboles Binarios
  Colas binarias de prioridad

Diseño y Análisis de Algoritmos Cap. 1 AHU

- Algoritmo :

Secuencia finita de operaciones, cada una de las cuales tiene un significado preciso y puede ser realizada en un lapso finito de tiempo y con un esfuerzo finito.

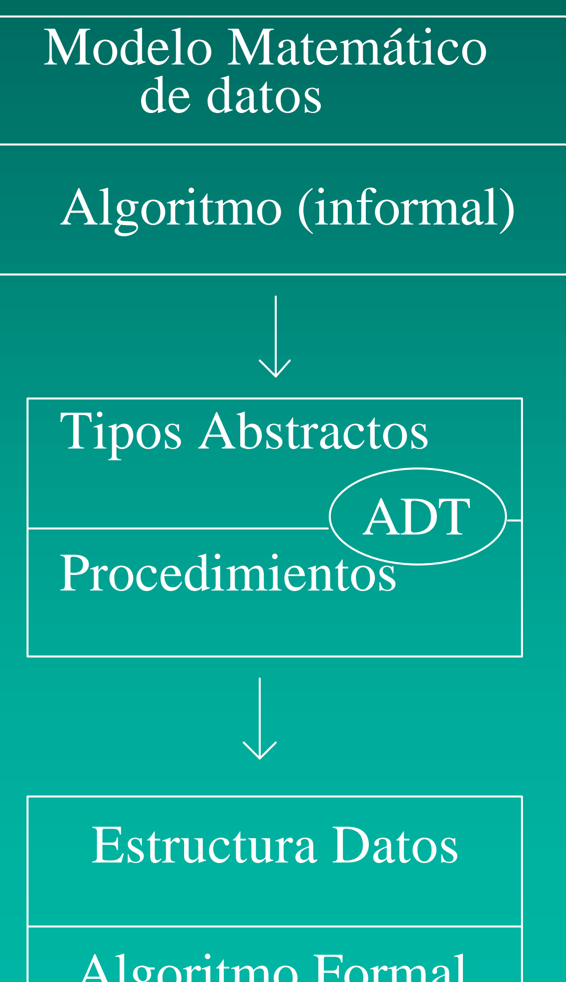
Diseño y Análisis de Algoritmos Cap. 1 AHU

- Heurística:

Un algoritmo que produce soluciones razonablemente rápidas y buenas, pero no necesariamente la solución óptima.

Diseño y Análisis de Algoritmos Cap. 1 AHU

- Método refinaciones sucesivas :



- **ADT** : Tipo Abstracto de Datos

  Modelo matemático con una colección de operaciones definidas sobre el modelo.

- **Implementación ADT** (en un lenguaje):

  - Declarando Tipos
  - Procedimientos de Manipulación

- Estructura de Datos :

Colección de Variables, posiblemente de diferentes tipos de datos, conectadas de algún modo. La estructura debe tener un bloque básico y una especificación de construcción o estructuración.

→ Mecanismos de Estructuración:
(Definición o axiomatización)

Agrupamiento (de bloques)
Relación (entre bloques)(punteros)

→ Manipulación:

Definición de Operadores

→ Representación:
En un computador (virtual o físico)

Punteros en Pascal

- Definición de Tipo:

  Type Tp= ↑ T

  -Tp tipo puntero a objeto de tipo T

- Declaración de puntero:

  var p : Tp

  - Espacio para p creado en momento de compilación

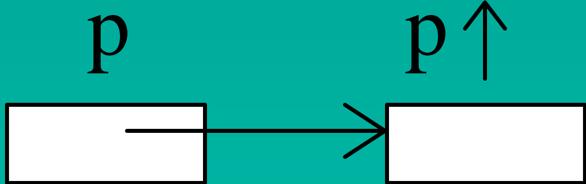
Punteros en Pascal

- Creación espacio dinámico:

new(p)

- En ejecución crea en memoria objeto de tipo T ; llamado p*

- Escribe en p la dirección del objeto



Punteros en Pascal

- dispose(p)

  - Libera el espacio dinámico, para ser reusado

- Fijación de Puntero:

  p:=NIL

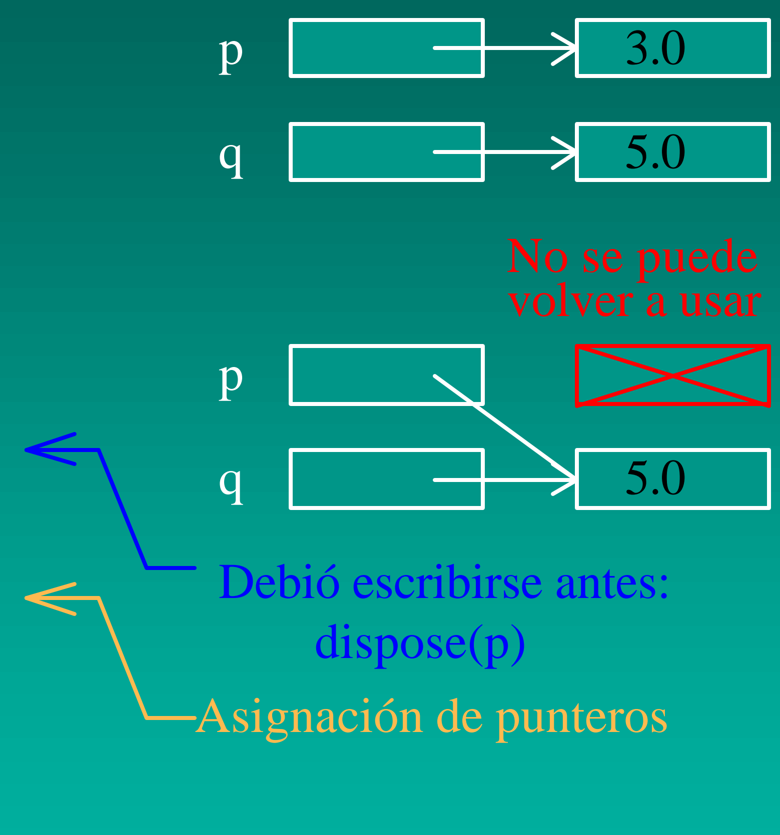
  - NIL es cte. estándar. Se inicia puntero apuntando a nada.

p

Ejemplos :

- Con :

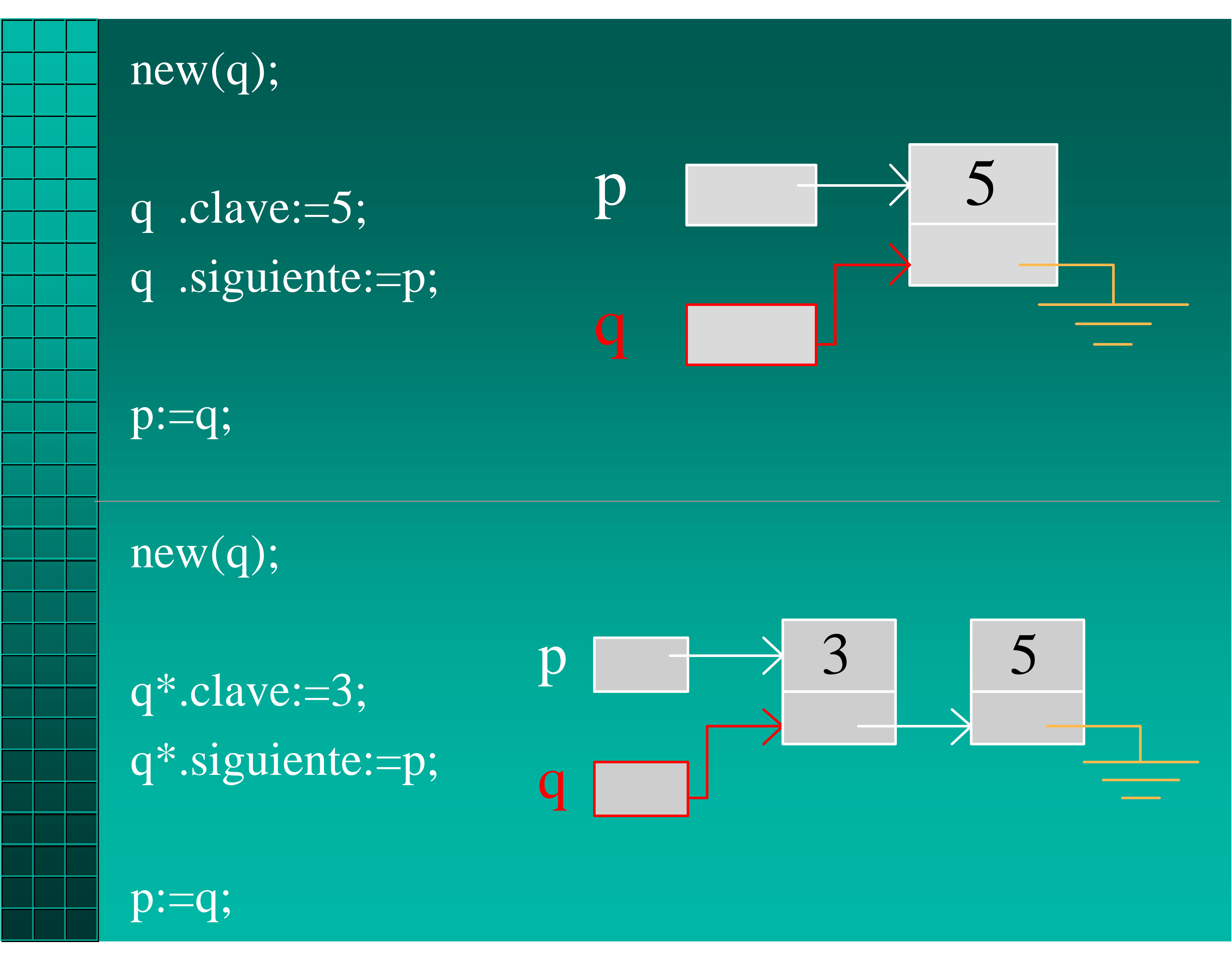
```
Type   Tp=*Real;
VAR    p,q:Tp;
...
new(p); p*:=3.0;
new(q); q*:=5.0;
....
p:=q ;
...
```



- Obs. También pueden asignarse valores : p ↑ :=q ↑

```
Type    T=RECORD
        clave: INTEGER
        siguiente: *T;
        ...
        END

VAR
        p,q:*T;

p:=NIL;
```

{q: puntero auxiliar}


new(q);
q .clave:=5;
q .siguiente:=p;
p:=q;
p
5
q
new(q);
q*.clave:=3;
q*.siguiente:=p;
p:=q;
p
3
5
q

Eficiencia. Complejidad Temporal

- Intro.: AHU pág. 16 y 17; LSB pág. 250 y 251

- Entrada :

  - Tamaño de los datos de entrada a procesar
  - La unidad es la entidad sometida al proceso básico y característico del programa.
  - Se dice que la entrada es de tamaño n, si se tienen n datos de tipo unidad.

T(n) mide la complejidad.

Ejemplo :

- Si

$$T(n) = (n+1)^2$$

- Puede demostrarse que :

$$(n+1)^2 \leq 4n^2 \quad ; \quad \text{para} \quad n \geq 1$$

- Entonces :

$$T(n) \leq 4n^2 \quad (\text{es de orden} \quad n^2)$$

- Lo que interesa es encontrar una cota superior para el crecimiento de n.

- Se dice que T(n) es de orden n^{i}

$$\text{Si} \quad \exists \; c \text{ y } k \quad \ni \quad T(k) \leq ck^{i}$$

- Notación O : (big oh)(orden)

$$\text{Si} \quad \exists \; c \text{ y } k \quad \ni \quad T(n) \leq cf(n) \;\; \text{con } n \geq k$$

- Se dice que: T(n) es O(f(n))

- La complejidad temporal es : f(n)

- **Ejemplo**: Demostrar que

Con : $1 \leq n$

Se tiene : $n^2 \leq n^3$

Multiplicando por 2 ; $2n^2 \leq 2n^3$

Sumando $3n^3$: $3n^3 + 2n^2 \leq 5n^3$ para $n > 0$

$\therefore \quad T(n) \leq 5$ para $n \geq 1 \Rightarrow c = 5, k = 1$

Complejidad n^3

Regla de Concatenación de Programas

- Regla de Suma:

P { P1 / P2 }

T1(n) es O(f(n))

T2(n) es O(g(n))

P es O(max(f(n),g(n)))

- Demo. (por definición de O) :

$$T1(n) \leq c1\, f(n) \qquad n \geq n1$$

$$T2(n) \leq c2\, g(n) \qquad n \geq n2$$

- Sea : no=max(n1,n2)

- Para

$$n \geq no: \quad T1(n) + T2(n) \leq c1\,f(n) + c2\,g(n)$$

- Caso 1: f(n)>g(n)

$$c1\,f(n) + c2\,g(n) = (c1 + c2)\,f(n) - c2(f(n) - g(n))$$

positivo

$$\therefore c1\,f(n) + c2\,g(n) \leq (c1 + c2)\,f(n)$$

- Caso 2 : ....

  ....

- Entonces:

$$T1(n)+T2(n) \leq (c1+c2) max(f(n), g(n)) \qquad n \geq max(n1.n2)$$

- Por definición de O :

  T = T1 + T2 es O(max(f(n),g(n)))

- Corolario :

  Si f > g entonces : **O(f+g) = O(f)**

- **Ejemplo : O(n2+n)=O(n2) !!!!**

- Regla de Productos:

T(n) = T1(n)*T2(n)   es   O(f(n)*g(n))

- Ejemplo:

$$O\left(\frac{n^2}{2}\right) = O(n^2)$$

En Programas :

a) If c then a1 else a2

En peor caso se toma la acción de mayor orden. Se aplica regla de sumas para la condición y la acción.

b) for c:=1 to n do a

Se suma n veces el tiempo de a. Si a es $O(1)$, entonces : $nO(1) = O(n)$

En Programas :

c) Acciones primitivas se consideran O(1)

Asignación, read, write, etc.

d) Procedimientos recursivos

Debe encontrarse relación de recurrencia para T(n). Ej. Pág. 24 AHU

- Ejemplo : for c=n downto i+1 do a ; a es O(1)

Entonces O(1)*(n-i), por lo tanto O(n-i)

Ejemplo:

```
procedure  m(n:integer);
var i,j,k: integer;
begin
  for i=1 to n-1 do
      for j=i+1 to n do
         for k=1 to j do
           {acciones O(1)}
end;
```

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = O(n^2)$$

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} = O(n^3)$$

Solución:

- Primer For:

$$\sum_{k=1}^{j} O(1) = j = O(j)$$

- Segundo For:

$$\sum_{j=i+1}^{n} j = (i+1) + (i+2) + \ldots + n = \sum_{j=1}^{n} j - \sum_{j=1}^{i} j$$

$$= \frac{n(n+1)}{2} - \frac{i(i+1)}{2}$$

- Tercer For :

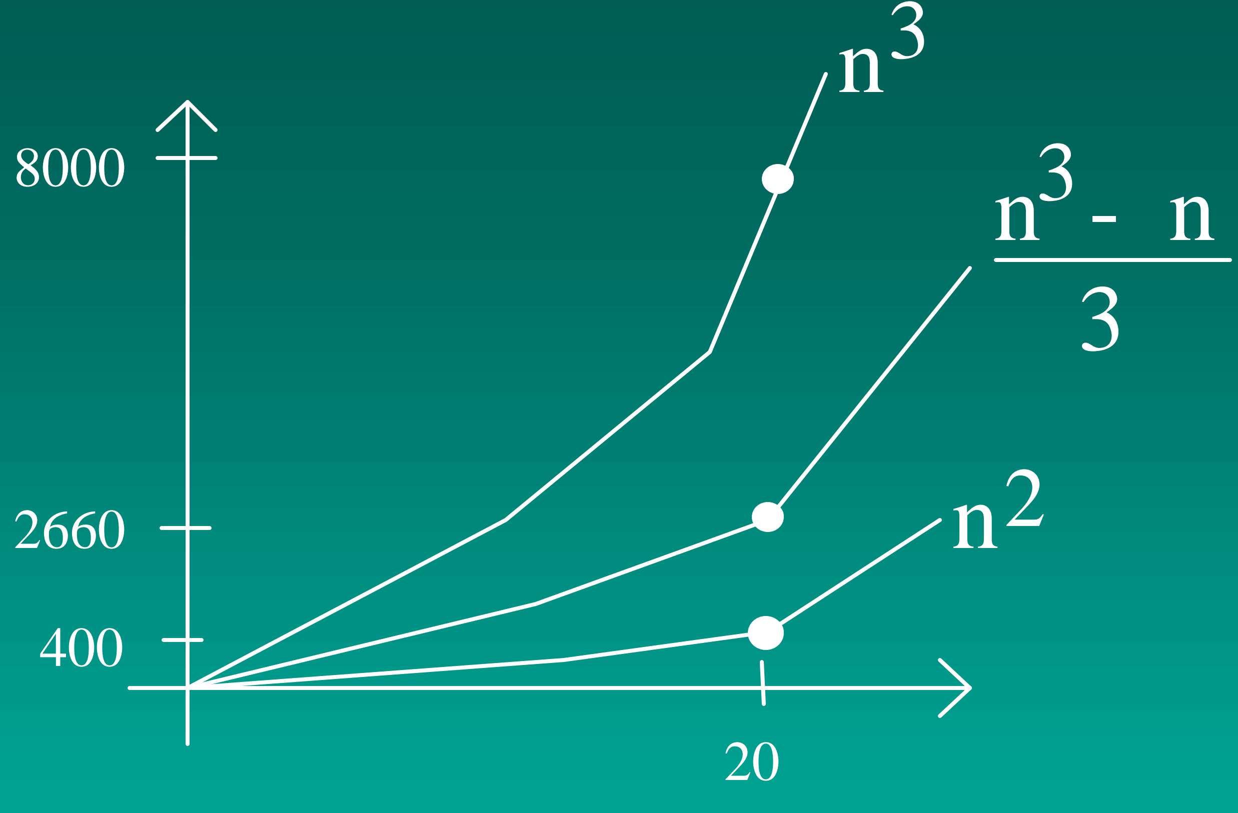
$$\sum_{i=1}^{n-1}\left(\frac{n(n+1)}{2}-\frac{i(i+1)}{2}\right)=$$

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

$$=\frac{(n-1)n(n+1)}{3}$$

- Finalmente :

$$T(n)=\frac{n^3-n}{3}=O(n^3)$$



n^3
$\frac{n^3 - n}{3}$
n^2
8000
2660
400
20

Listas

Estructura dinámica que puede crecer y acortarse bajo demanda.

Los elementos pueden ser accesados, insertados y descartados en cualquier posición dentro de la lista.

Tipos de Listas : (en memoria primaria)

- Stack : Inserciones y descartes en un solo extremo.
- Colas : Inserciones en un extremo; descartes en el otro.
- Circulares
- Mapping (arreglo asociativo)
- Doblemente enlazadas
- Strings

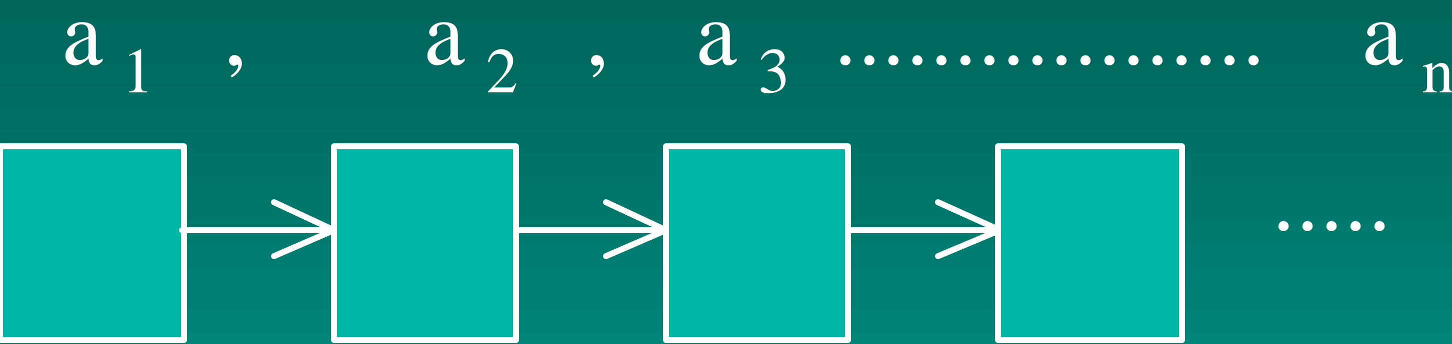
(Hoare define un ADT más general. La secuencia que incluye secuencias en discos, cintas , etc. pag. 130)

Definición Matemática.

- Secuencia de cero (lista vacía) o más elementos de un tipo.

- Los elementos quedan ordenados (linealmente) por su posición en la secuencia.

- Representación :



- Se requiere solo un enlace entre un elemento y su sucesor.

- Operaciones : Buscar elemento, insertar, descartar, concatenar, partir.

Operaciones Primitivas:

- Se definen en forma abstracta, las siguientes:

Sean :	L	una lista
	x	un objeto en la lista
	p	posición del objeto en la lista

- **makenull(L)** :crea lista vacía; retorna posición end(L).
- **end(L)** : retorna posición sucesora del último elemento.

Operaciones Primitivas:

- insert(x,p,L) resultado indefinido si posición p no existe.
- locate(x,L) posición de primera ocurrencia de x en L; si no está retorna end(L)
- retrieve(p,L) Valor de la componente en la posición p. Falla si p>=end(L)

Operaciones Primitivas:

- **delete(p,L)** Descarta elemento en posición p.Falla si p>=end(L)
- **next(p,L)** Sucesor. Falla si p>=end(L)
- **previous(p,L)** Antecesor, Falla si p>end(L) o si p=first(L)
- **first(L)** Primera posición; si vacía retorna end(L)

Observaciones :

- Las definiciones no están asociadas a la

implementación.

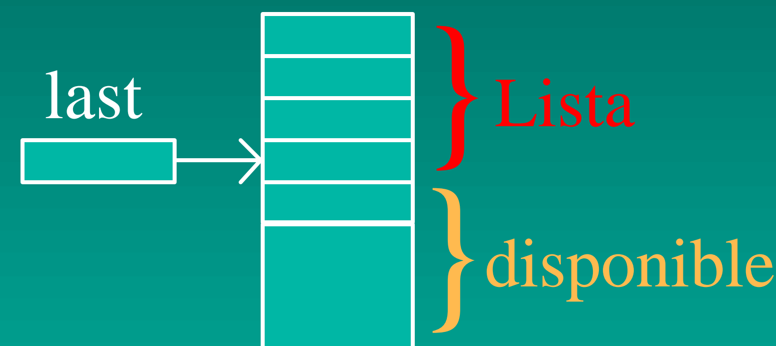
- Pueden haber diferentes implementaciones
- Pueden crearse operaciones más complejas, con estas primitivas. Ver ejemplo Purge, pag 40 2.1 AHU

Conviene escribir programas en términos de operaciones. Si se desea cambiar la operación, el código de éstas está perfectamente localizado; y no es necesario cambiar código a través del programa.(pág. 42).

Confección de Bibliotecas de ADT.

- Implementación de Listas

a) En base a arreglos



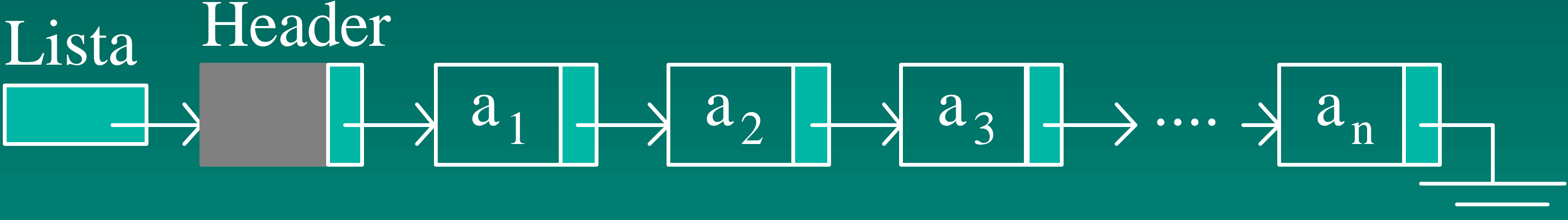
Op. fáciles (de bajo costo)

- Agregar al final
- Previo, sucesor, fin de lista

Op. costosas: Descartar, insertar (requieren reescribir zonas)

- La posición está asociada al índice.
- Se requiere reservar espacio máximo.

b) En base a punteros



- **Posición i** : Puntero que contiene ai (para i=1,...n)

- **Posición 1**: puntero a header (implica, se requiere header)

- **Posición end(L)** : puntero a última celda de L.

Observaciones :

- Las definiciones no están asociadas a la implementación.
- Pueden haber diferentes implementaciones
- Pueden crearse operaciones más complejas, con estas primitivas. Ver ejemplo Purge, pag 40 2.1 AHU

Pascal

```
Type
   celda=record
     elemento:telem;
     next: ↑celda
   end;
lista= ↑celda;
posición= ↑celda;
```

C

```
typedef struct tcelda{
    telem elemento;
    struct tcelda *next;
    } celda, *pcelda;

typedef pcelda lista;
typedef pcelda
posición;
```

Pascal

```
function end(l: lista):
   posición;
var
   q: posicion
begin
   q:=l;
   while q↑.next<>nil do
      q:=q↑.next;
      end:=q
end;
```

C

```
posicion end(l)
lista l;

{ posicion q=l;
  while(q->next!=NULL_P)
    q=q->next;
  return(q);
}
```

- Para evitar "warning" en la compilación:

```
#define NULL_P (posicion ) ∅
```

- Ya que en stdio: NULL es (char *) ∅

Petición de espacio dinámico:

```
posicion getcel()
{ posición q;
  char *malloc();
  q=(posicion) malloc(sizeof(celda));
  if (q==NULL_P) return (NULL_P);
  q->next=NULL_P;
  return(q);
}
```

```
position makenull()
  { posicion q, getcel();
    q=getcel();
    return(q);
  }

lista l;
....
l=makenull();
```

```
lista makenull()
{ posicion q, getcel();
   q=getcel();
   return(q);
}
```

Insertar

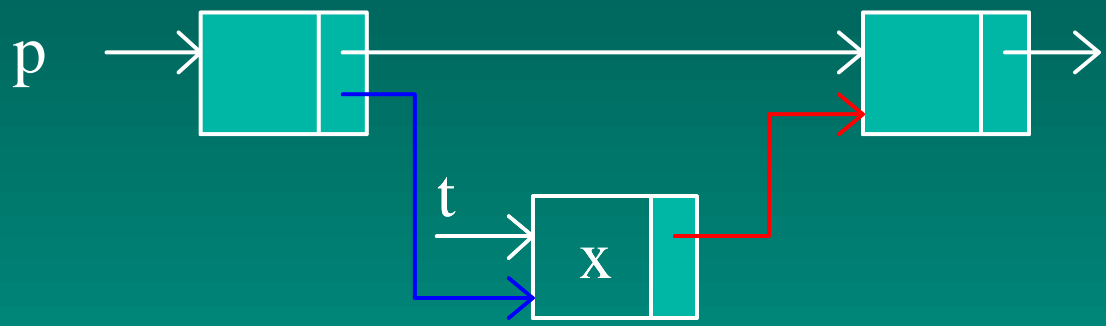
```
posicion insert(x,p)
          telem     x;
          posicion p;
          {         posicion t, getcel();
                    t=getcel();                          /* espacio */
                    if (t==NULL_P) return(NULL_P);
                    t->next=p->next;                     /* cola */
                    t->elemento=x;                       /* valor */
                    p->next=t;                           /* cabeza */
                    return(p->next);
          }
```

p

t

x

Insertar



- Retorna puntero al recién insertado.
- Si es NULL_P, falla la inserción

Borrar

```
int delete(x,p)
        posicion p;
        {       posicion t;
                t=p->next;                          /* marca */
                if (t==NULL_P)
                        return(1);
                else
                        {p->next=p->next->next;   /* liga */
                        free((char *) t);
                        return(∅);
                        }
        }
```

p

t

- Si no existe header, debe preguntarse si el que se desea borrar es el primero. Además debe pasarse la lista :

```
if (p==l)                    /* ¿ es el primero ?  */
        {       l=p->next;
                free((char *) p);
                return(*);
        }
else
        ...
```

Posición

```
posicion locate(x,l)
        telem   x;
        lista   l;
        {       posicion p=l;
                while(p->next!=NULL_P)
                        if (p->next->elemento==x)
                                return(p);
                        else
                                p=p->next;
                return(NULL_P);
        }
```

Locate falla si retorna NULL_P.

- En el último retorno, puede ponerse p. En este caso se detecta falla según :

```
q=locate(x,l);
if      (q->next==NULL_P)
        /* falla */
else...
```

- En nuestro caso:

```
if  ((q=locate(x,l))==NULL_P) /*not found */
{.......
```

c) Listas en base a cursores :

Un cursor es un índice de un arreglo, que simula un puntero.

	elemento	cursor
1		
2		

- En este caso, el cursor contiene el índice del siguiente en la lista.

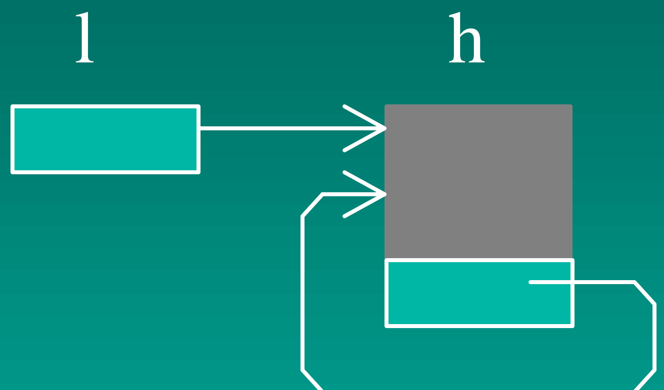
- Las listas terminan en cursor=0

- Ver implementación pág. 48 AHU

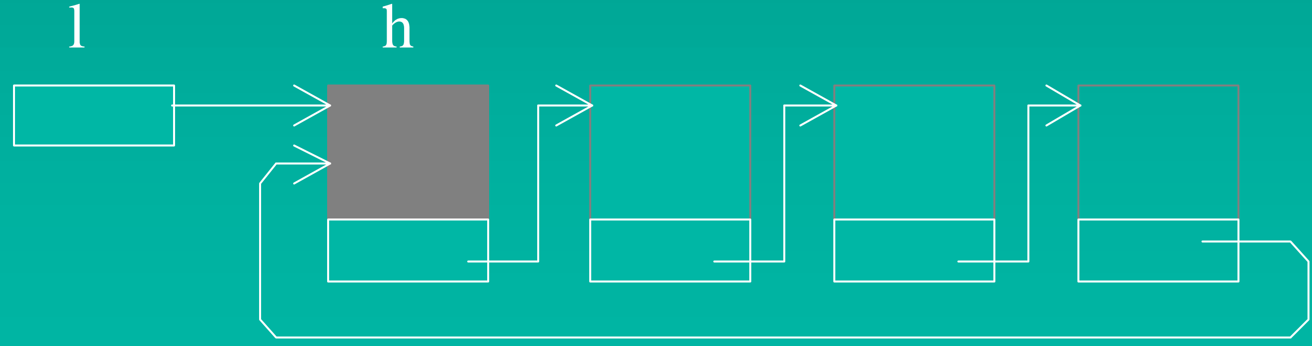
- **Obs** : Pueden guardarse varias listas en el espacio; pero se requiere tener una lista con los espacios disponibles

Lista Circular :

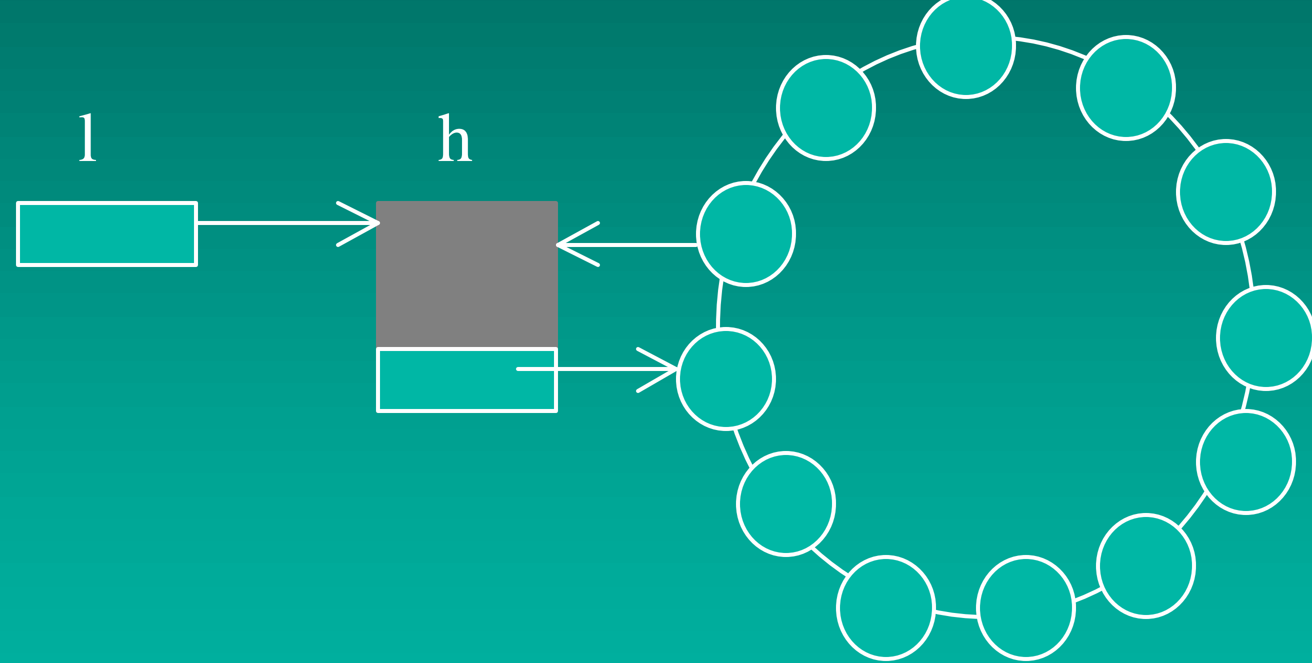
- Lista Vacía :



- Lista con elementos :

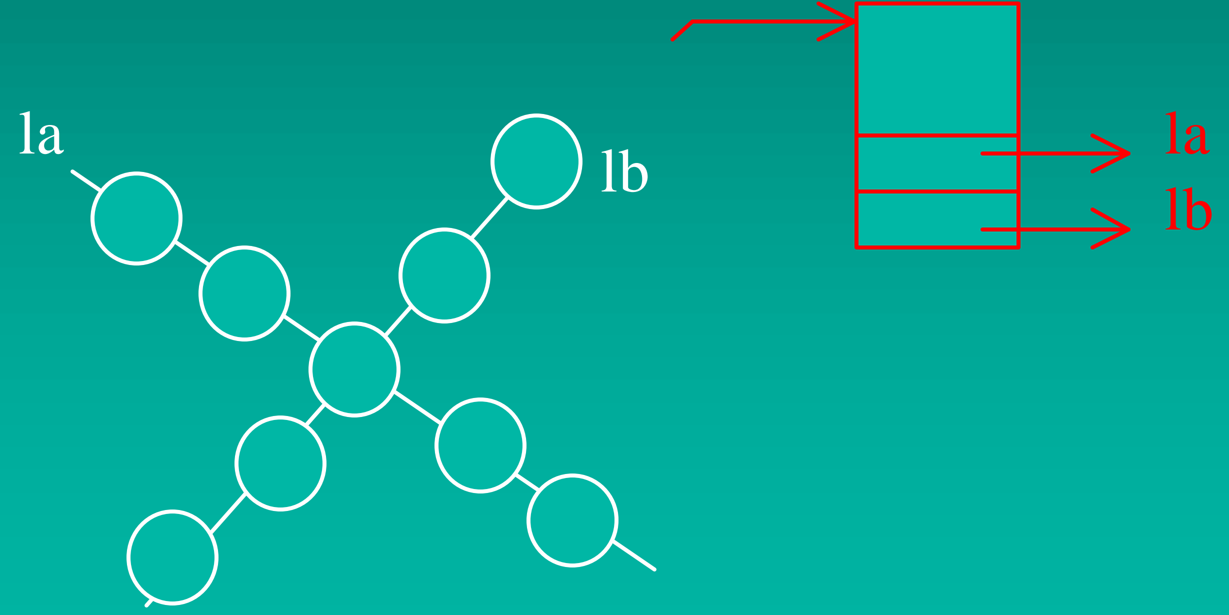


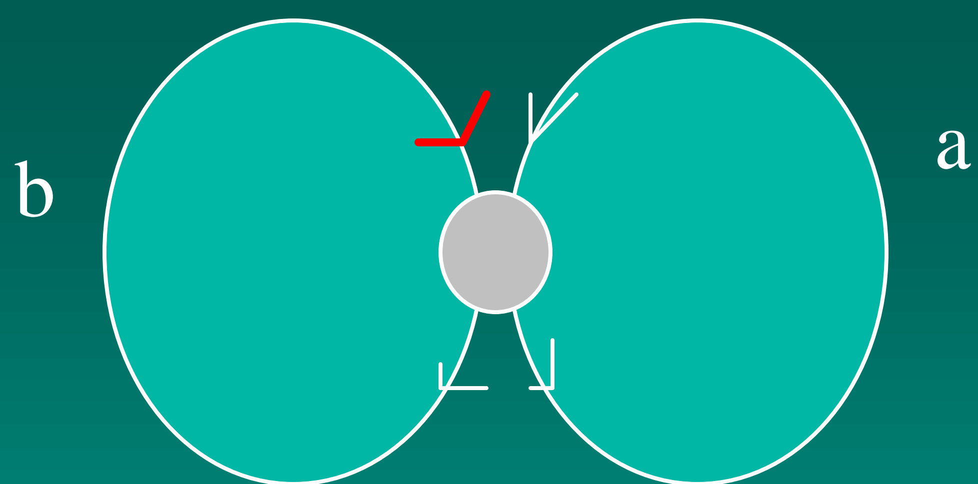
- Cualquiera puede ser el primero, (se puede cambiar la referencia)



- Otro uso de la lista circular :

Listas que se interceptan. Si se borra, en una, un elemento común; la otra queda desligada :





- Se marca en lista a

- Se recorre b hasta encontrar puntero ->

- Se borra en ambas listas; y se liga punteros

Listas doblemente enlazadas.

- En la lista simple, las operaciones *previous* y *end* son de complejidad O(n).

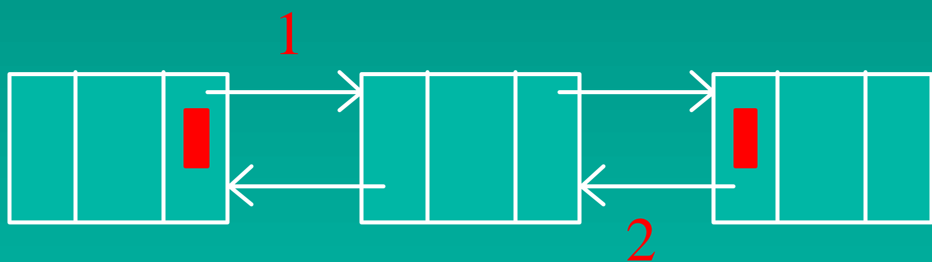
- Con la lista doblemente enlazada, dichas operaciones son de tiempo constante. Esto se logra con más espacio de punteros y con operaciones más complejas.

Características :

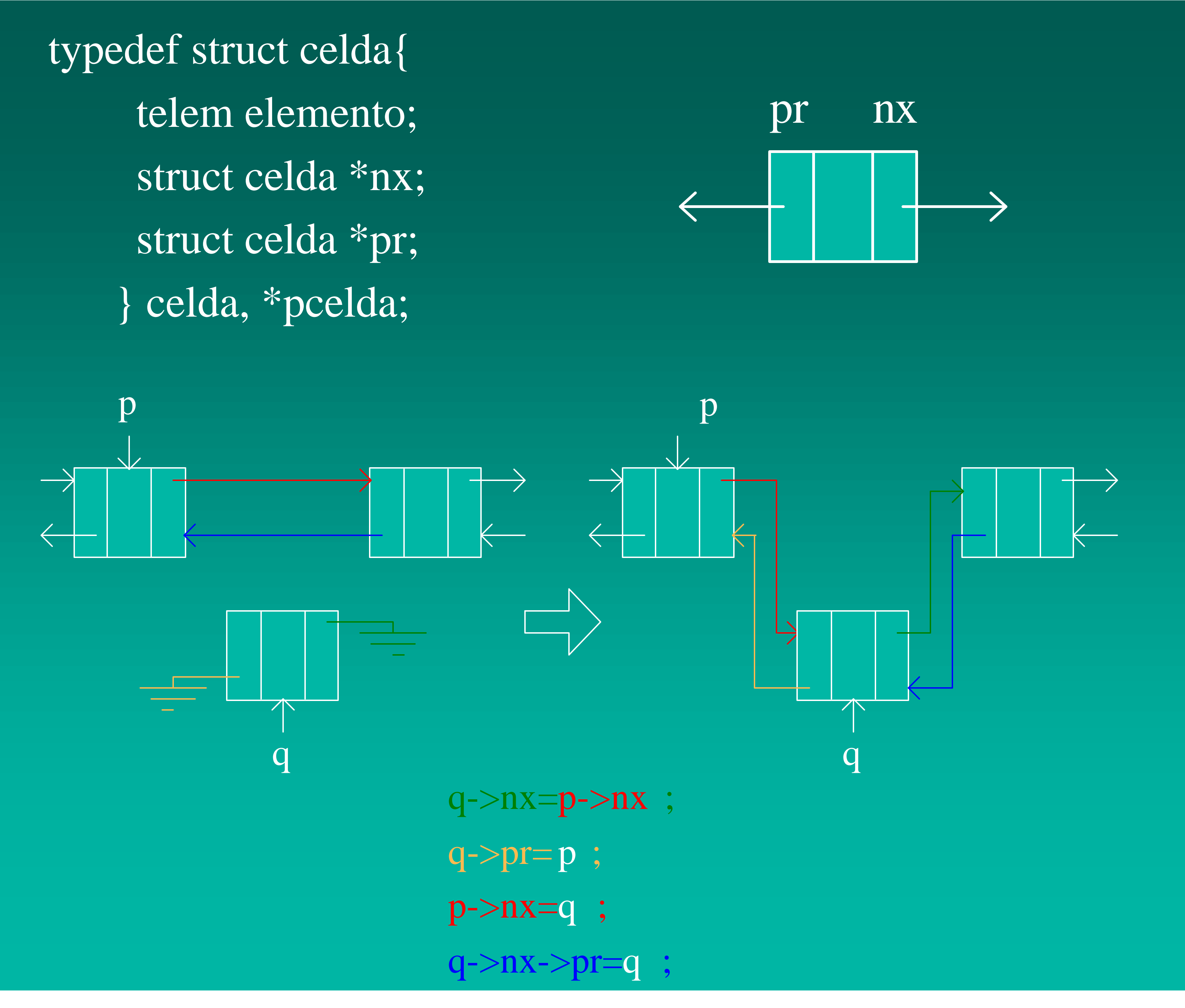
- Permite recorrido bidirecional

- Introduce redundancia

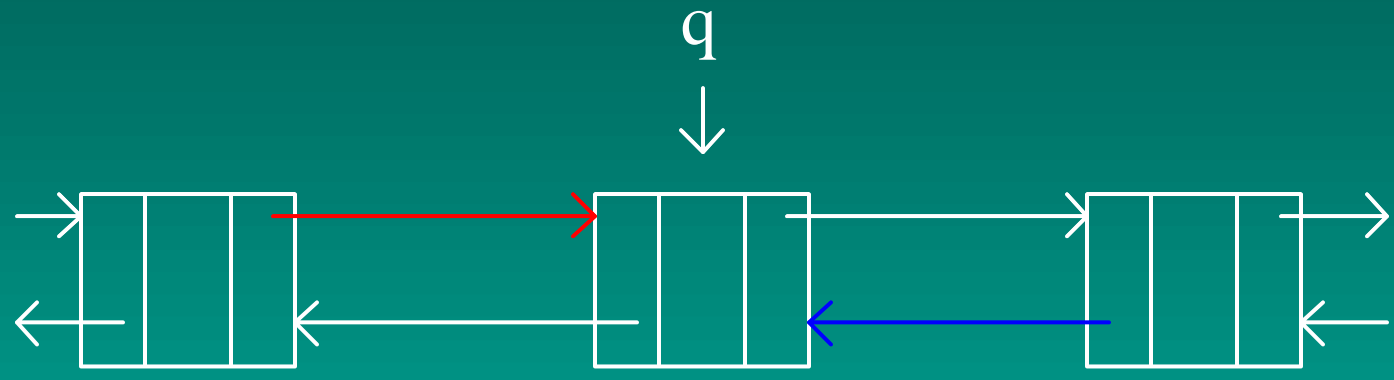
Si por error se borrara puntero 1 ó 2; pueden recuperarse :



- Pueden representarse: colas, stacks y dobles colas.

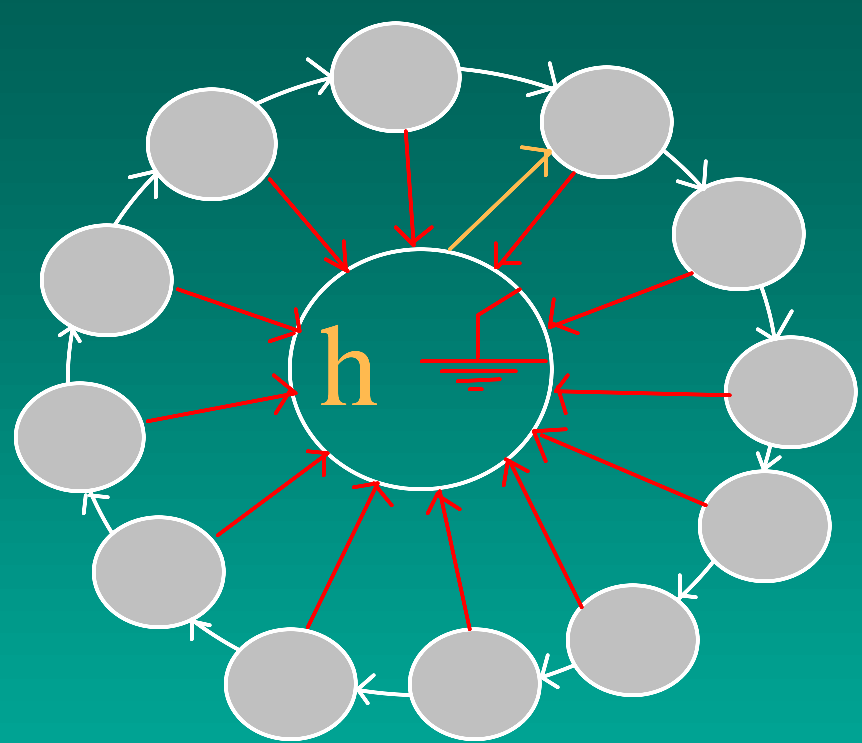

typedef struct celda{
telem elemento;
struct celda *nx;
struct celda *pr;
} celda, *pcelda;
pr
nx
p
q
p
q
q->nx=p->nx ;
q->pr=p ;
p->nx=q ;
q->nx->pr=q ;

sacar(q)



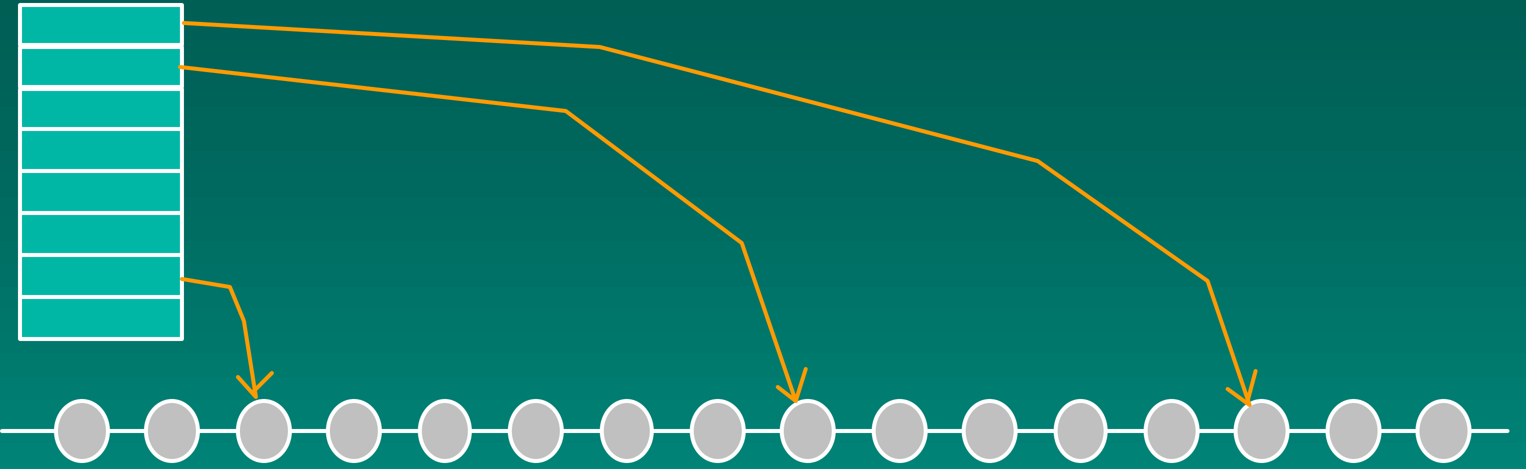

```
q->pr->nx= q->nx ;

q->nx->pr= q->pr ;
```

- Anillo con puntero a la cabecera :



- Basta referencia a un nodo.
- No se requiere pasar la lista
- Se tiene acceso al siguiente y al header

- Lista Indexada :



- Listas autoorganizadas :
  - Mover al frente (Gonnet 25. Wirth 174)
  - Transponer (Gonnet 29)
- Búsqueda secuencial
- Búsqueda en listas ordenadas (Wirth 176 a 182)

- Listas ordenadas :
  - Merge sort de listas ordenadas (Gonnet 134)
  - Quicksort en listas (Gonnet 136)
  - Empleadas como árboles de prioridad (Gonnet 164)

- Ordenamiento topológico (Wirth 182)

- Insertar antes (truco : Wirth 172)

Administrador de Memoria Dinámica

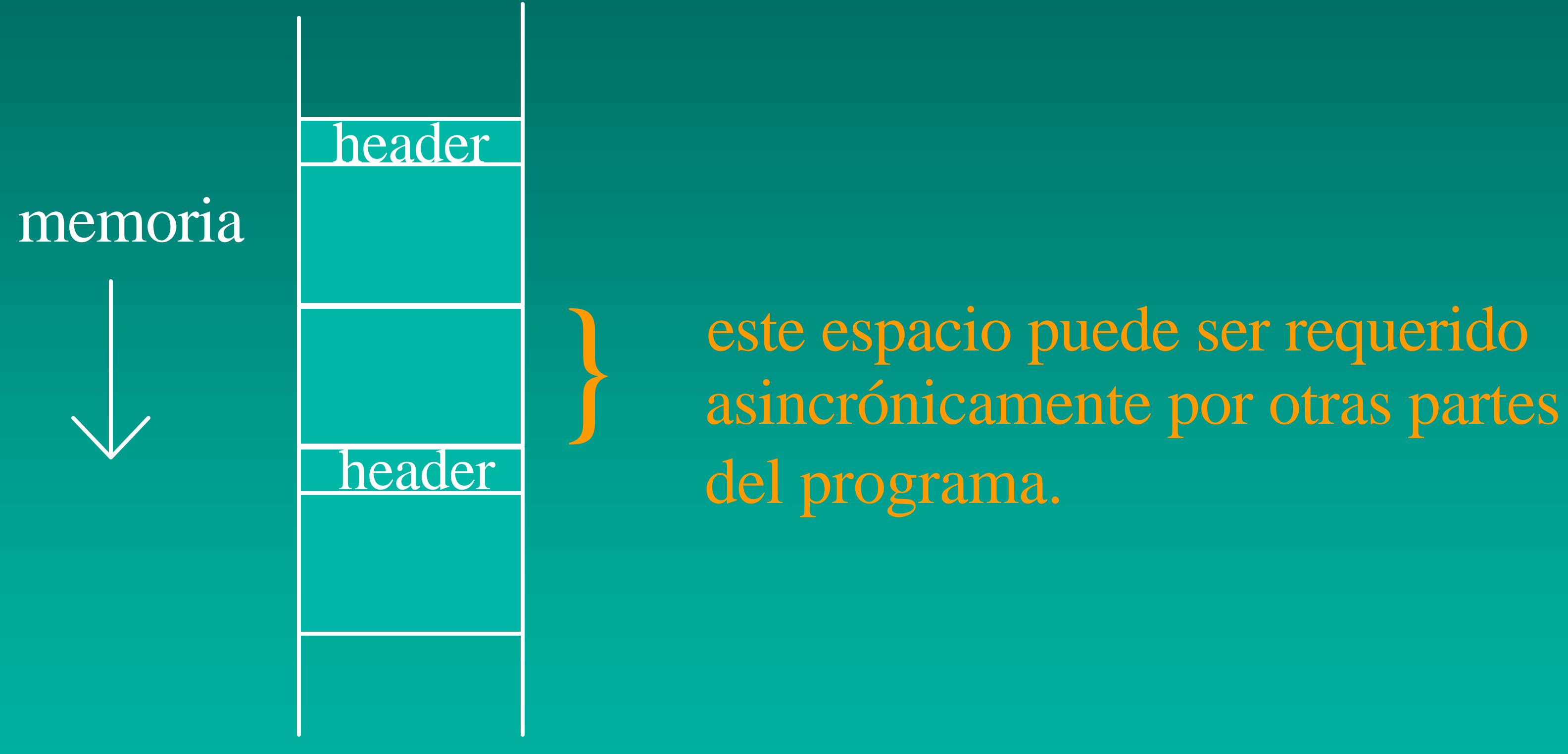
char *alloc(nbytes) unsigned nbytes;	void free(ap) char *ap;	K. R. 8.7

- Se mantiene lista circular de bloques libres.
- Cada bloque contiene :

header
espacio

{ tamaño
puntero al siguiente bloque } espacio de administración

- Se emplea lista circular, pues el espacio administrado no necesariamente es contiguo en la memoria.



- La lista se mantiene ordenada en orden creciente de direcciones (*).

Para tomar espacio :

- Se recorre el anillo hasta encontrar bloque con espacio suficiente.

- Si hay espacio :

  - se rebaja el tamaño del espacio libre.
  - el resto es pasado al programa (se saca de los libres).

- Si no hay espacio :

  - Se obtiene del sistema operativo y se encadena a la lista de libres. Se vuelve a pedir espacio.

Para soltar espacio :

- Se recorre lista para insertar bloque libre.
- Si es adyacente con alguno de la lista; se agranda el tamaño del bloque (se los pega), para evitar fragmentación (*).

```
#define NALLOC 128 /*#units to allocate at once*/

static HEADER *morecore(nu) /*ask system for memory */
unsigned nu;
{           char *sbrk();
            register char *cp;
            register HEADER *up;
            register int rnu;

            rnu= NALLOC*((nu+NALLOC - 1)/NALLOC);
            cp=sbrk(rnu*sizeof(HEADER));
            if ((int) cp==-1)   /* no space at all */
                        return(NULL);
            up=(HEADER *)cp;
            up->s.size=rnu;
            free((char *)(up+1));
            return(allocp);
}
```

```
free(ap)  /* put block ap in free list */
char *ap;
{          register  HEADER *p,*q;
           p=(HEADER *)ap -1 ; /* point to header */
           for (q=allocp;!(p>q&&p<q->s.prt);q=q->sprt)
                      if (q>=q->s.prt && (p>q||p<q->s.prt))
                                 break; /* at one end of other */
           if (p+p->s.size == q->s.prt)/* join to upper nbr */
                      { p->s.size += q->s.prt->s.size;
                        p->s.prt = q->s.prt->s.prt;}
           else    {  p->s.prt = q->s.prt;}
           if (q+q->s.size == p )/* join to lower nbr */
                      {  q->s.size += p->s.size;
                         q->s.prt =p->s.prt;}
           else
                      q->s.prt = p;
           allocp 0 q;
}
```

```
typedef int ALIGN ; /* forces alignment on PDP-11 */
union header { /* free block header */

        struct {
                union header *prt; /* next free block */
                unsigned size; /* size of this free block */
        } s;
        ALIGN x; /* forces alignment of blocks */

};

typedef union header HEADER;
```

```
static HEADER base; /* empty list to get started */
static HEADER *allocp = NULL; /* last allocated block */

char *alloc(nbytes) /*general-purpose storage allocator*/
unsigned nbytes;
{
        HEADER *morecore();
        register HEADER *p, *q;
        register units in nunits;

        nunits=1+(nbytes+sizeof(HEADER)-1)/sizeof(HEADER);
        if ((q=allocp) == NULL) /* no free list yet */
                { base.s.prt=allocp=q=&base;
                 base.s.size=0;
                }
```

continua..

continuación

```
for (p=q->s.prt; ; q=p, p=p->s.prt)
        {  if (p->s.size >=nunits)   /* big enough */
                {   if (p->s.size==nunits) /* exactly */
                                q->s.ptr=p->s.ptr;
                        else                    /* allocate tail end */
                            {p->s.size -= nunits;
                                p += p->s.size;
                                p->s.size = nunits;
                                }
                        allocp= q;
                        return((char *)(p+1));
                }
                if (p==allocp)/*wrapped around free list */
                if ((p=morecore(nunits))==NULL)
                                return(NULL); /* none left*/
        }
}
```

- Alineamiento :

- Se define un tamaño unitario independiente de la máquina.

- El encabezado se mantiene alineado (mediante uniones) [union con el tipo más restrictivo]

- El tamaño del bloque es múltiplo del encabezado (y por lo tanto alineado).

- Redondeos y detalles :

  - En el tamaño se considera el encabezado.

  - El puntero retornado apunta al espacio libre (no al encabezado)

  - El tamaño pedido en caracteres se redondea hacia arriba en unidades de header.

- Consideremos :

$$k = \frac{n + s - 1}{s}$$

- Donde :

n : número de bytes de espacio a reservar.

s : tamaño en bytes del header

- La división es con truncamiento

- Si : $s=1 \Rightarrow k=n$

$$s=2 \Rightarrow k=\frac{n+1}{2} \quad ; \quad n\leq 2 \Rightarrow k=1$$

$$2<n\leq 4 \Rightarrow k=2$$

$$4<n\leq 6 \Rightarrow k=3$$

.....

$$s=3 \Rightarrow k=\frac{n+2}{3} \quad ; \quad n\leq 3 \Rightarrow k=1$$

$$3<n\leq 6 \Rightarrow k=2$$

$$6<n\leq 9 \Rightarrow k=3$$

.....

- Entonces :

k es redondeado (hacia arriba) y corresponde al número de unidades de s contenidas en n, por lo tanto :

$$k = \frac{n + s - 1}{s}$$

$$\boxed{nunits \quad = 1 + k}$$

Variables estáticas :

- Al inicio :

allocp

base

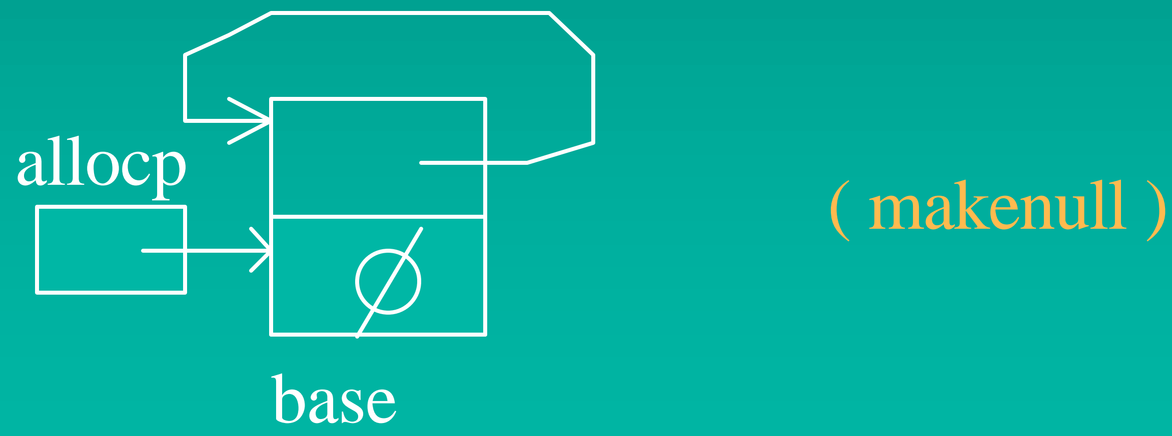
size

?

lista

header de la lista

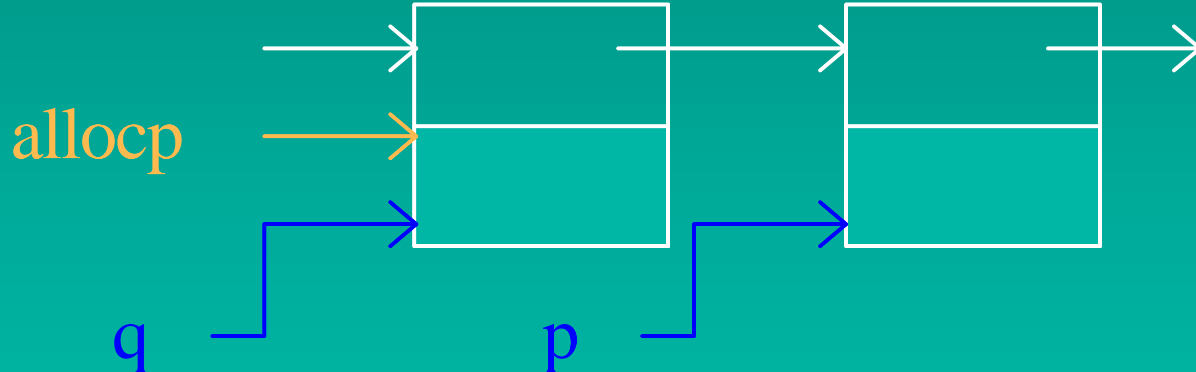
- Creación de la lista : (se realiza sólo una vez)



- Recorrido lista en petición (alloc):

  - Se emplean punteros :   p : siguiente
    q : referencia

  - Se inicia con :

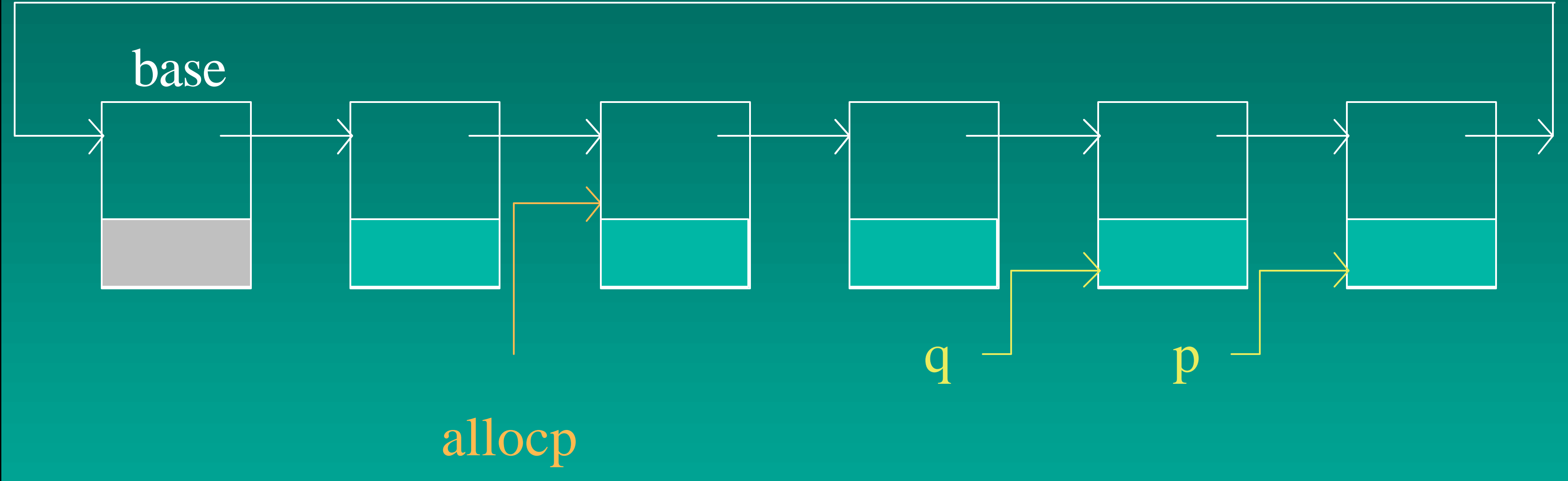


- *s* es la estructura dentro de la union. Por lo tanto los miembros de s son : s.ptr,   s.size.

- Por razones de homogeneidad (y rapidez):

  - Se comienza buscando espacio (en el siguiente bloque libre del cual se obtuvo espacio (queda marcado con allocp)) en el apuntado por p.

- Si no hay espacio suficiente y si no se ha recorrido toda la lista, se pasa al siguiente :

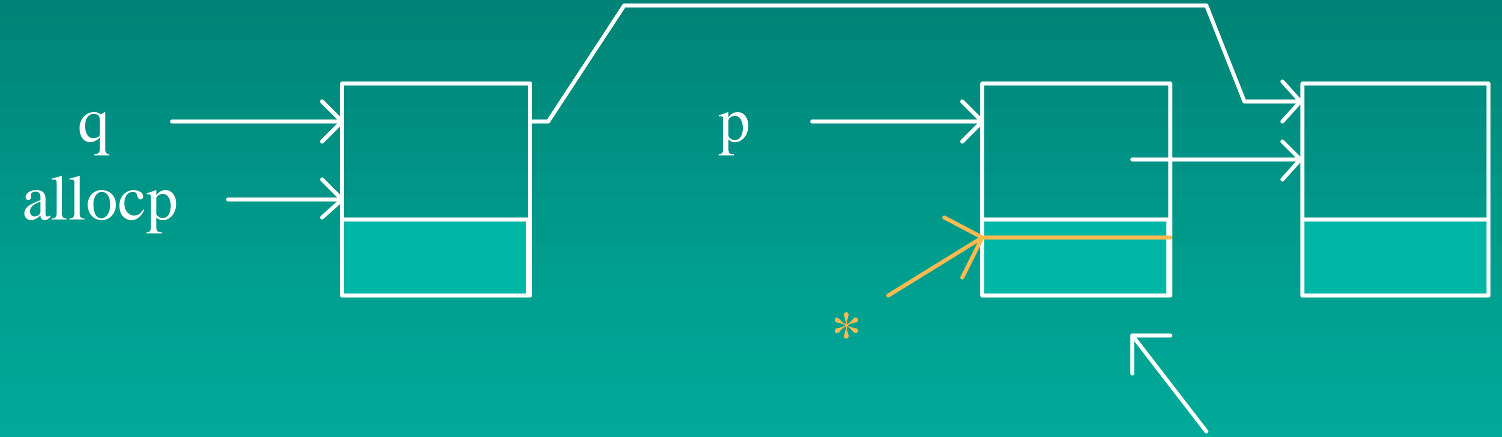


- Op. coma (de izquierda a derecha)

q = p , p = p -> s.ptr

- Si hay espacio exacto :

  - Se desconecta de la lista
  - Se fija allocp (apunta al anterior usado)



ya no está en la lista de los libres

- Retorna puntero al primer char del espacio (*)

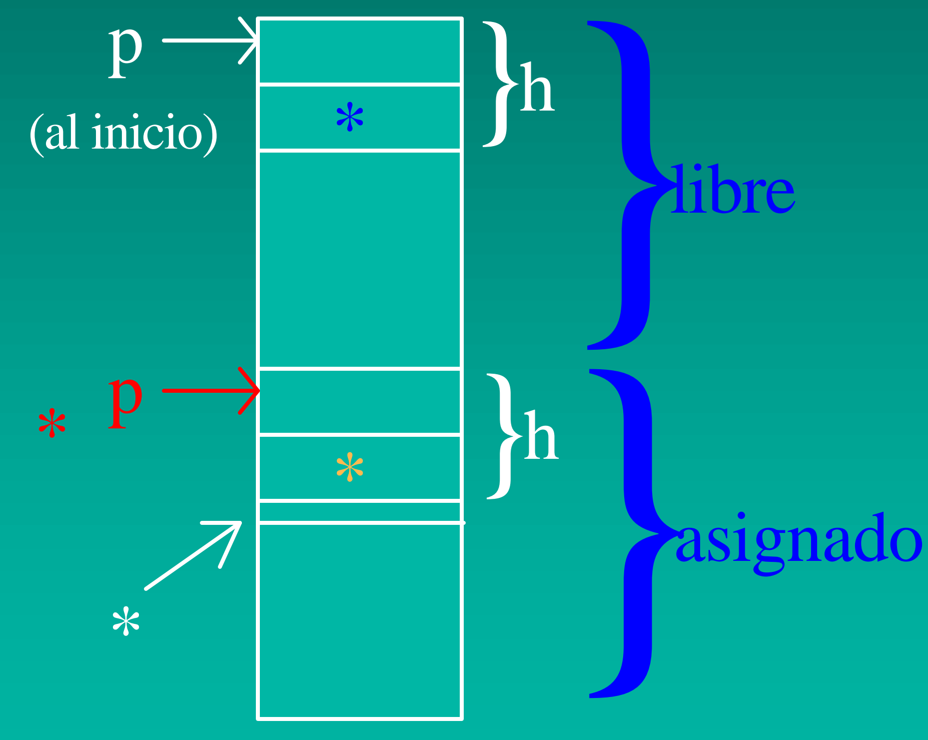
Si el espacio del bloque libre es mayor que el solicitado :

1.- Se descuenta espacio *

2.- Se apunta al bloque a asignar *

3.- Se escribe espacio en header del bloque asignado (para poder devolverlo) *

4.- Se fija allocp (igual que antes). Retorna primer puntero a caracter del bloque asignado. *



- Si se recorre la toda la lista y no hay espacio :

  - Se llama a morecore

- rnu : es el número de unidades de NALLOC contenidas en nu, por NALLOC.

- NALLOC : debe ser grande (respecto a las peticiones corrientes); no se desea pedir de a una unidad; para evitar llamadas al sistema :

  - efectos de buffer

  - Tiempo de cambio de contexto (pasar a modo kernel)

Se pide rnu veces el tamaño de asignación:

- up se emplea para cambiar tipo al puntero cp.

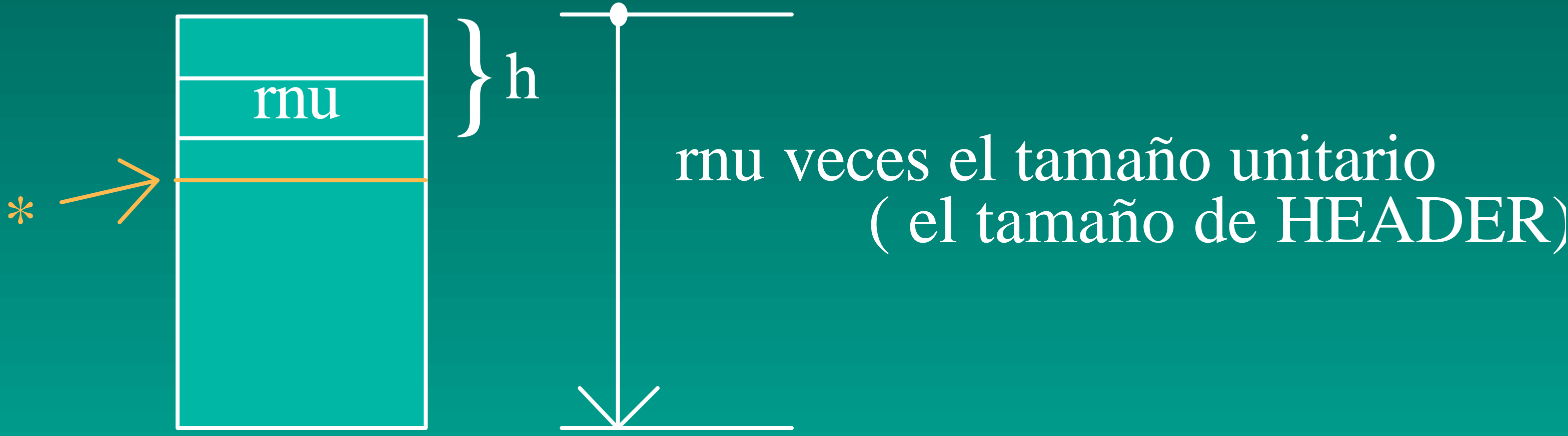
cp → ← up

unidades de char

unidades de header

- char *sbrk() retorna -1 (no NULL)
- Por lo tanto debe usarse cast con int, para comparar.

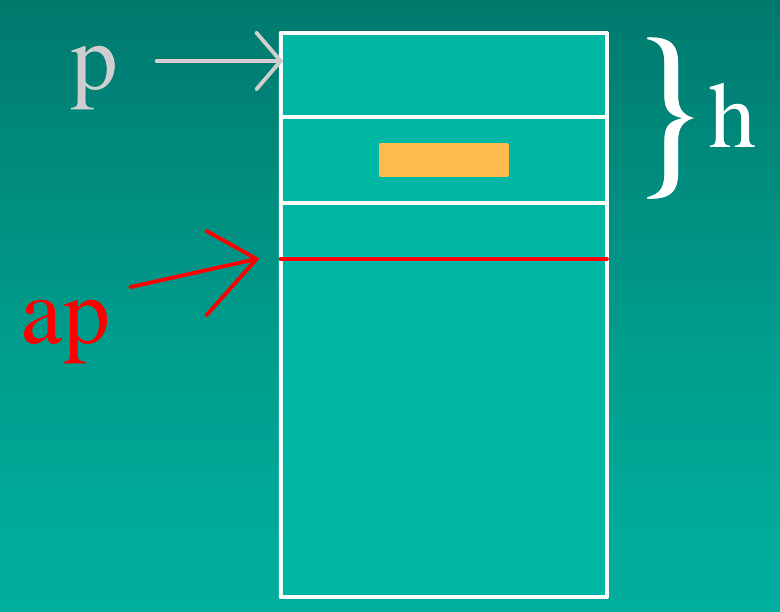
- El tamaño considera el header :



- * Se pasa a free un puntero a char, que apunta al espacio libre.

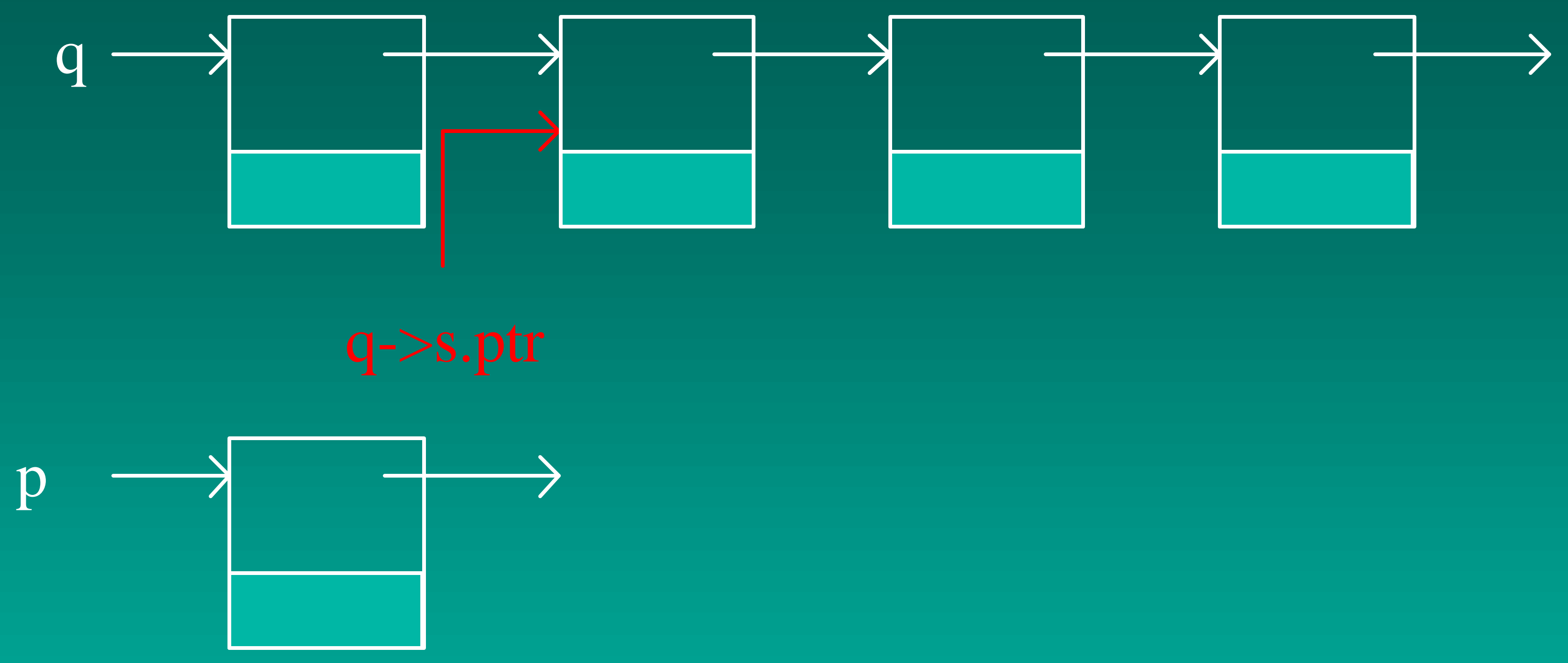
- **free** :

  Recorre lista de bloques libres,comenzando en allocp, buscando el lugar para insertar el bloque.



- p->s.size debe tener el valor del tamaño a ser liberado
- ap apunta al primer char del espacio del bloque
- p apunta al bloque
- q se inicia con allocp (apunta al bloque anterior al que se ha descontado espacio, más recientemente)

- El for ubica la posición para insertar :



- Si posición de memoria de p está entre q y el bloque siguiente : Debe insertarse p después de q.
- Es decir, si : p > q && p < q->s.ptr

- Si lo anterior no se cumple, se avanza al bloque siguiente. Salvo que se llegue al final:

q >= q->s.prt El anillo se mantiene ordenado según las direcciones de memoria

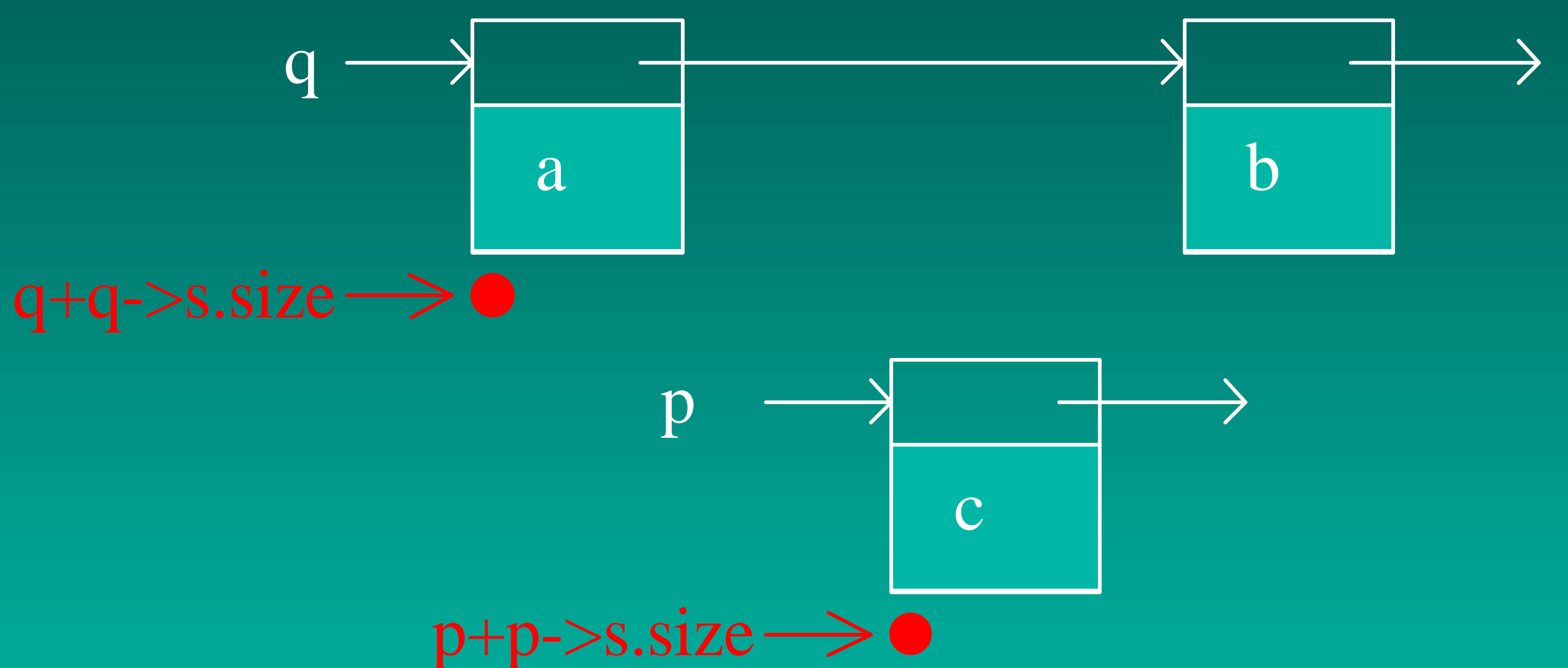
- y que sea mayor que el mayor :

p > q

- o menor que el menor :

p < q->s.ptr

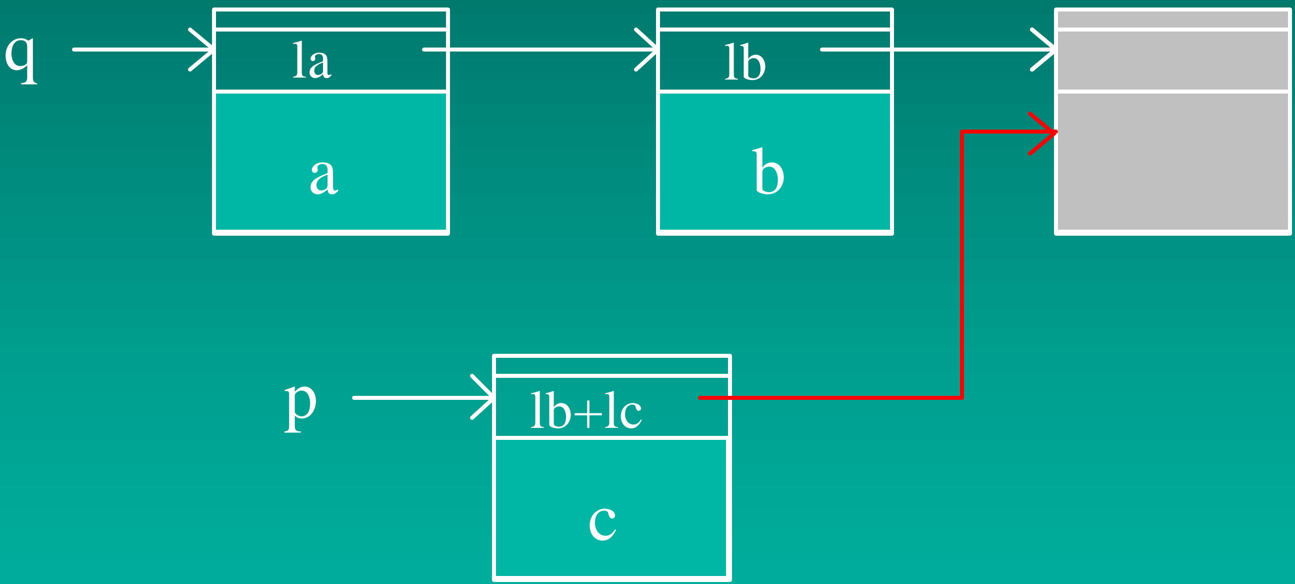
- Al salir del for, el bloque apuntado por p, debe insertarse después del bloque apuntado por q :



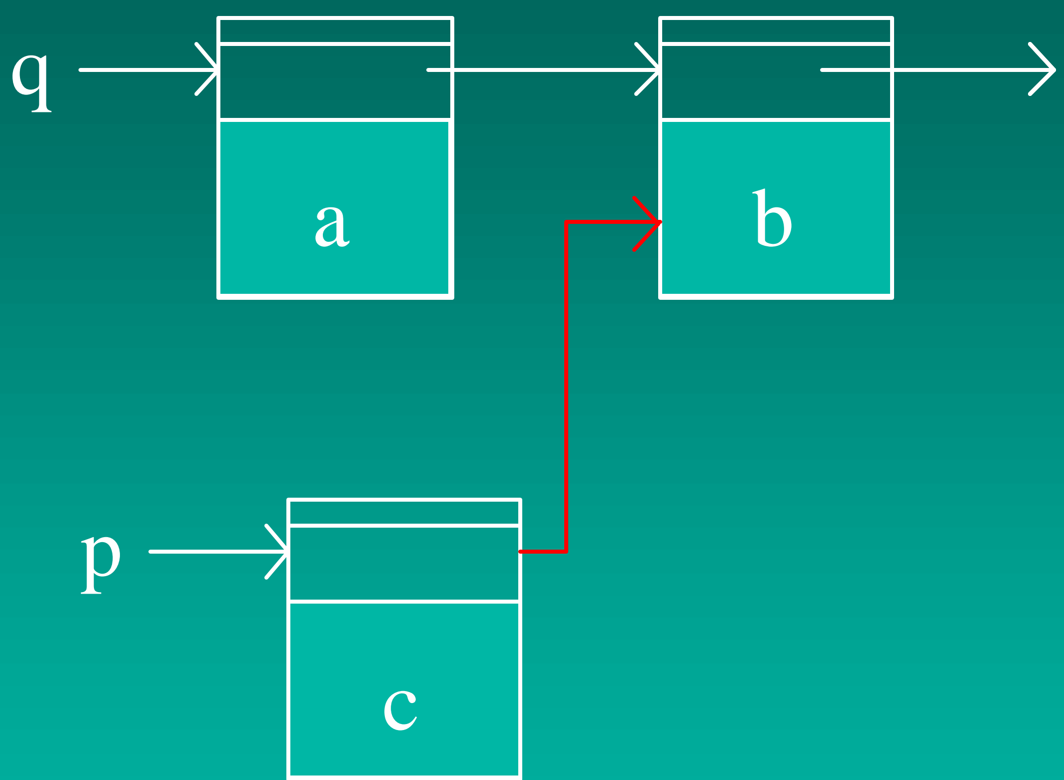
- c podría ser adyacente por arriba con b; y/o por abajo con a.

- El primer if, fija el puntero hacia la derecha de p:

  a) Si c es adyacente con b [then]

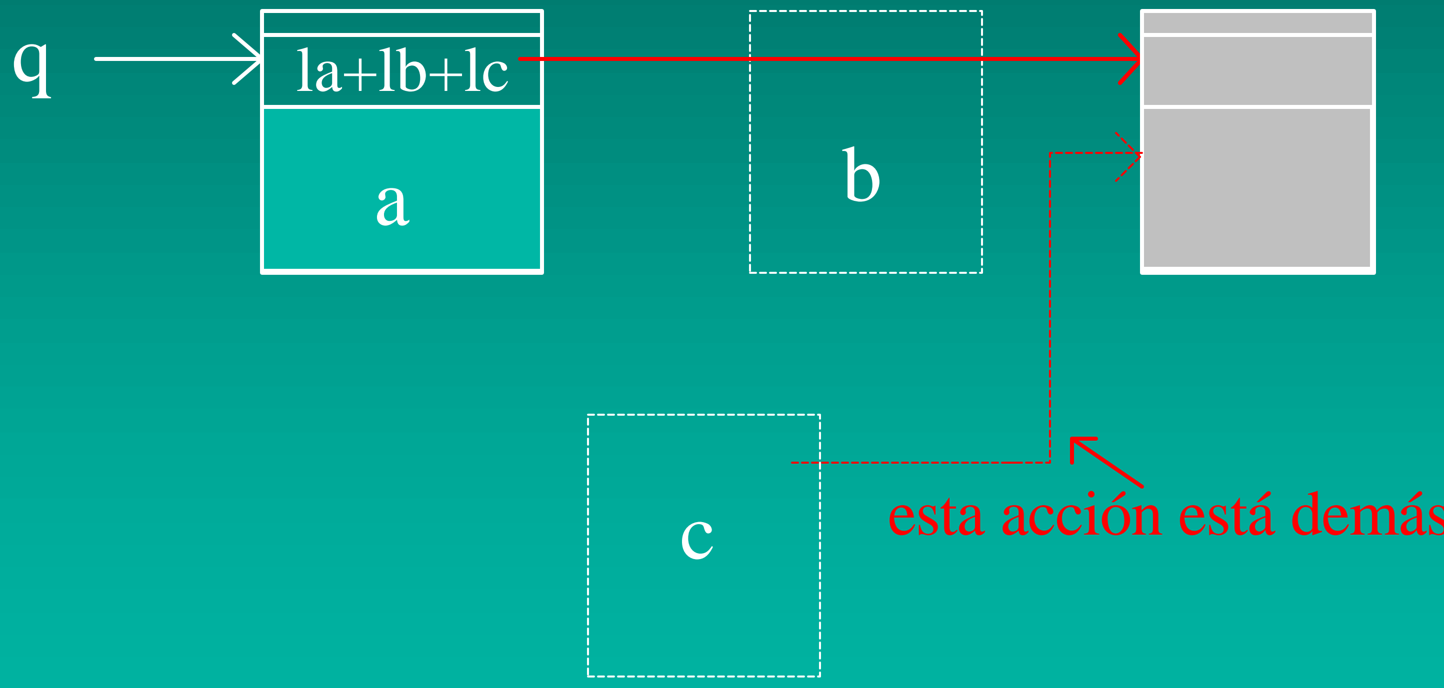


b) Si c no es adyacente con b [else]

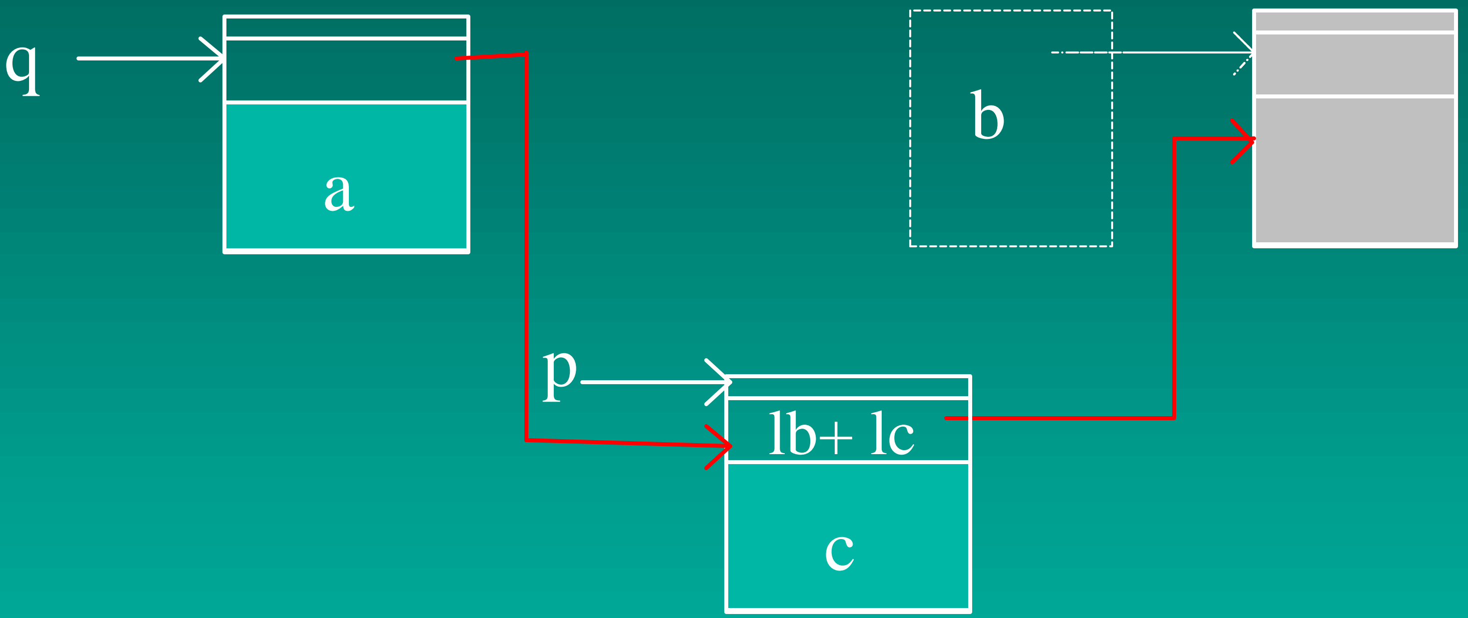


- El segundo if, fija puntero hacia bloque c (desde la izquierda de p) :

a1) Si c es adyacente con a y b [then]

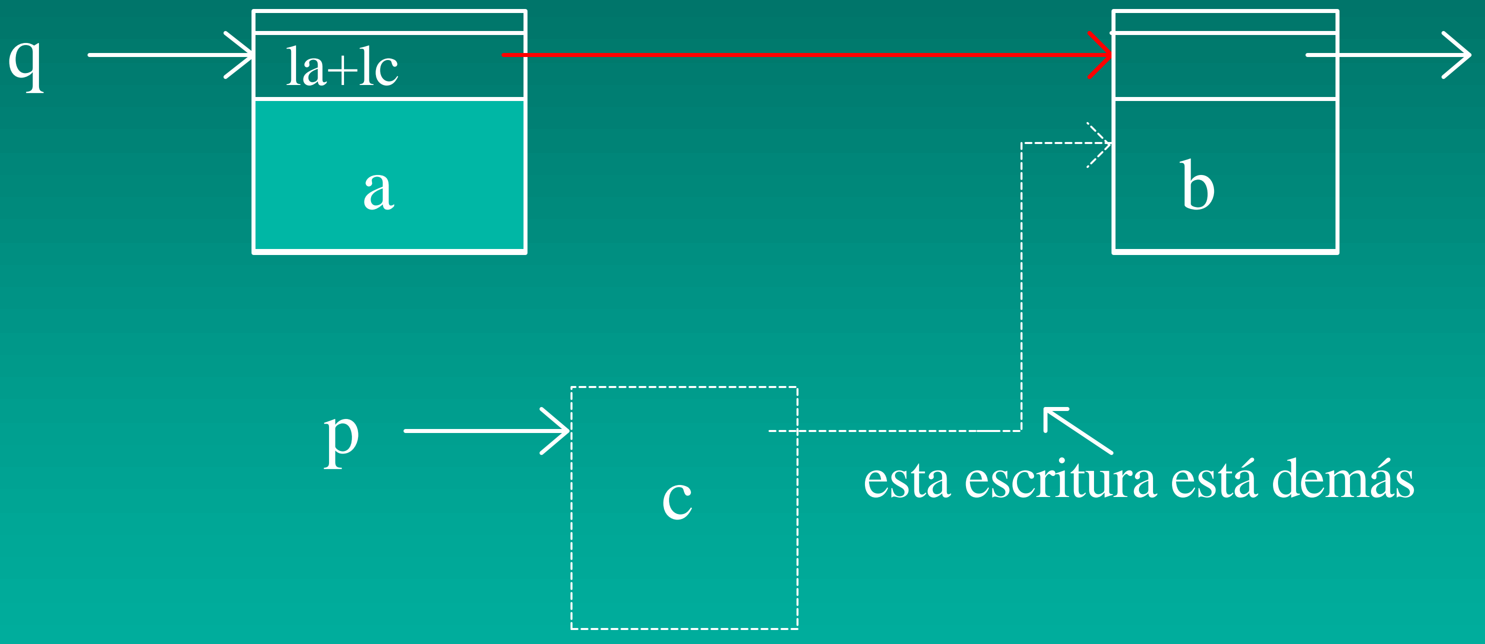


- a2) Si c no es adyacente con a; pero sí con b : [else]

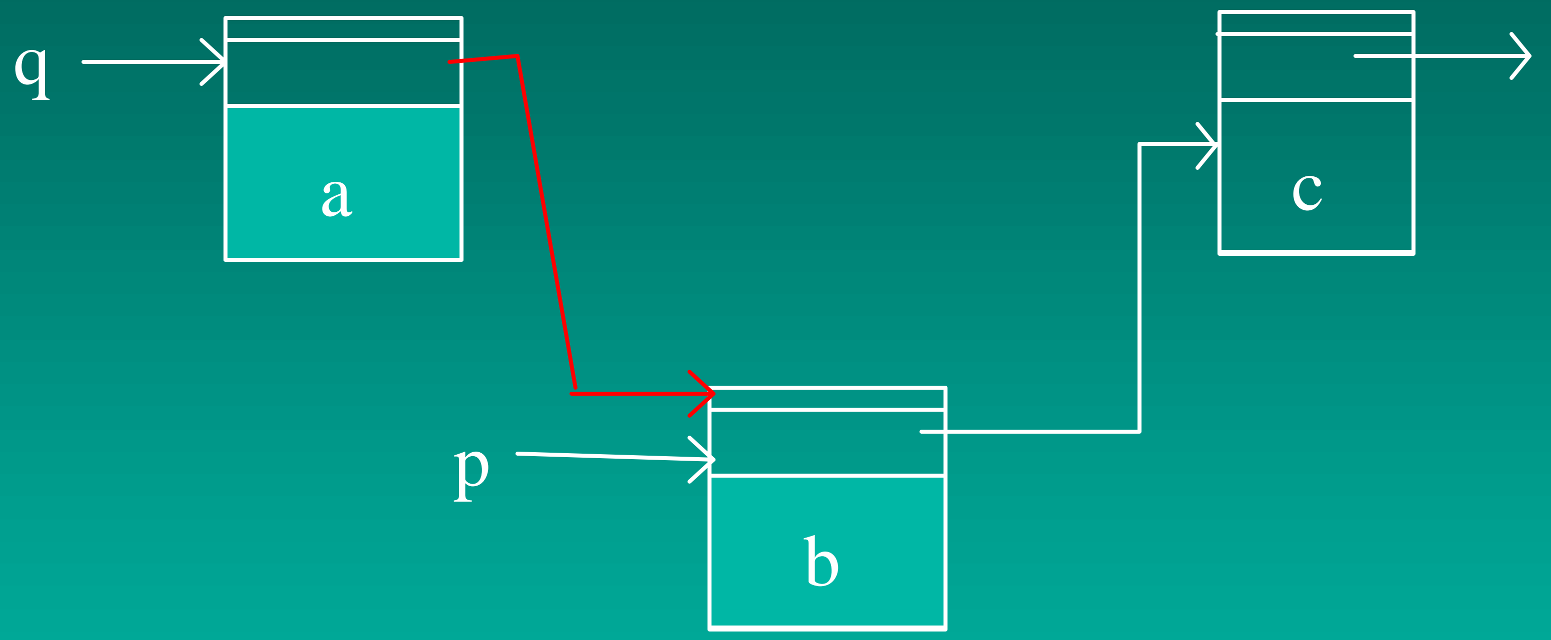


b1) Si c es adyacente con a , pero no con b

[then]



b2) Si c no es adyacente con a ni b [else]:



- Salida de free :

Escribe en la estática allocp el valor de q. Por lo tanto fija allocp en el que tiene nuevo espacio (a1,b1) cuyo siguiente tiene el nuevo espacio (a2,b2)

- En alloc :

a) Si se pide más espacio libre (morecore), se retorna en p un puntero al bloque con nuevo espacio (a1,b1) o al anterior (a2,b2) al que tiene nuevo espacio.

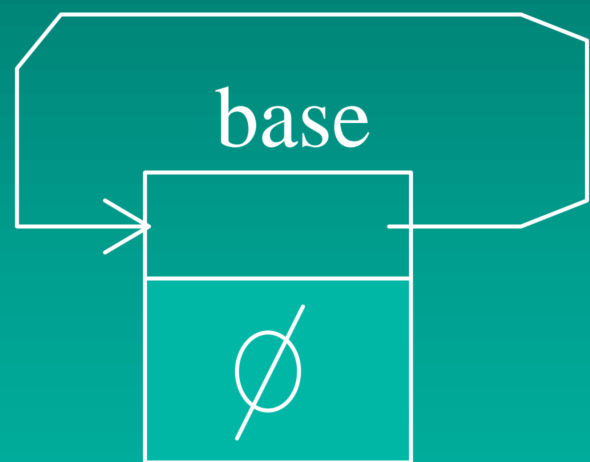
Por lo tanto en casos a1 y b1, debe recorrersetoda la lista nuevamente.(ésto no es eficiente). En free, puede registrarse la posición anterior a q y mejorar el código.

b) Si había espacio se fija allocp en el anterior al que se le descuenta espacio. Por lo tanto en la próxima vez intentará consumir del mismo bloque.

Iniciación de lista de bloques libres

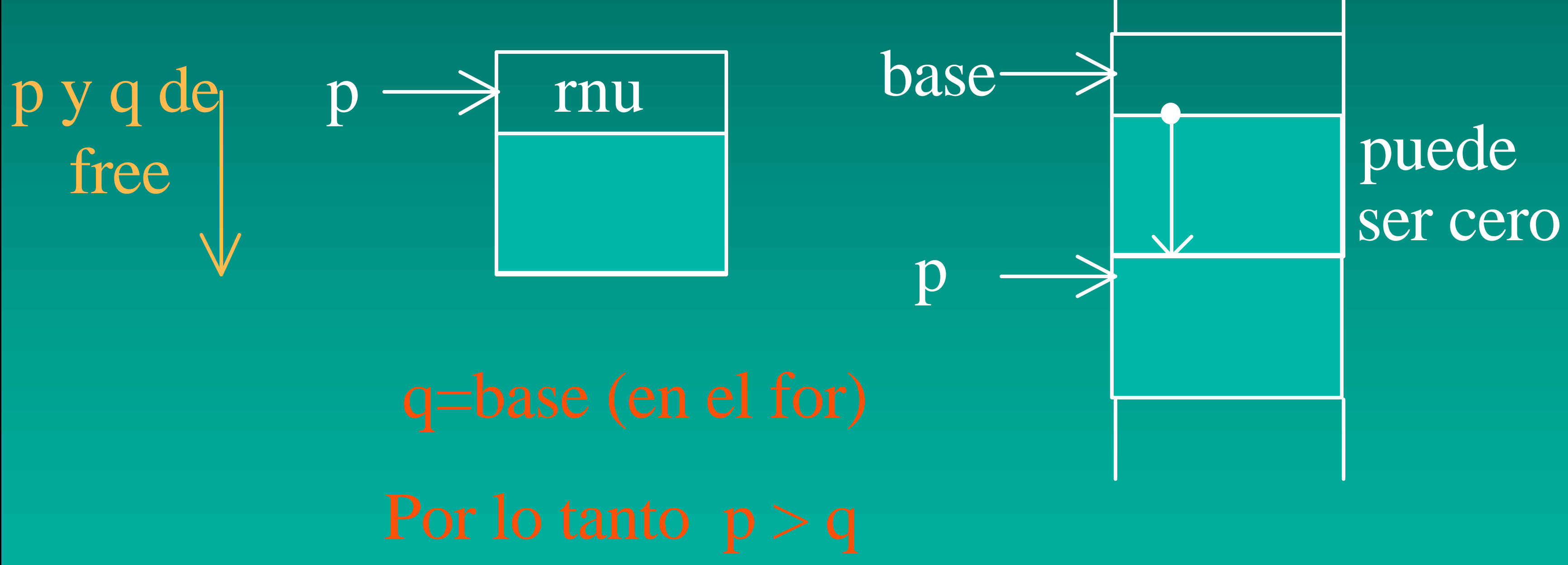
- Primer llamado a alloc.

  El primer if produce :

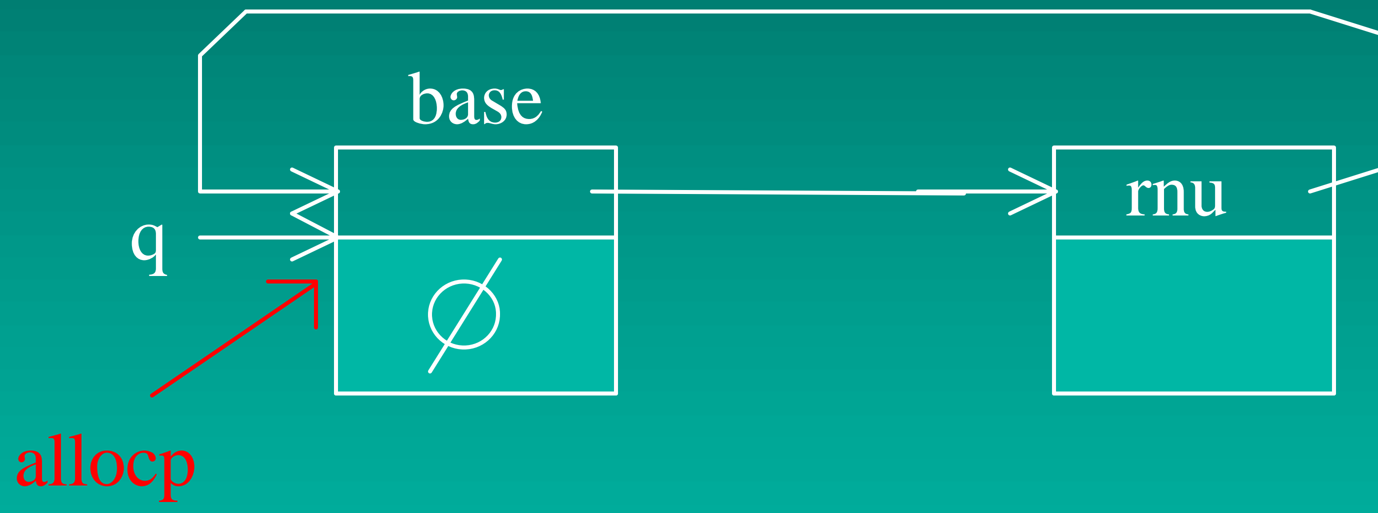


- Al comenzar el for, p = a Haep y se llama a morecore

- More core, llama a free:



- Se ejecuta el break
- Del primer if, se ejecuta al else
- Del segundo if, se efectúa el else (porque se inicia en $\emptyset$)
- Queda:



- Por lo tanto, No se usa el header
- No se emplea el header llamado base; que separa posiciones altas de las más bajas

Stack (Pila, LIFO, pushdown list)

- Inserciones y descartes en un extremo (denominado Tope).
- no tiene significado la variable posición (siempre será 1).

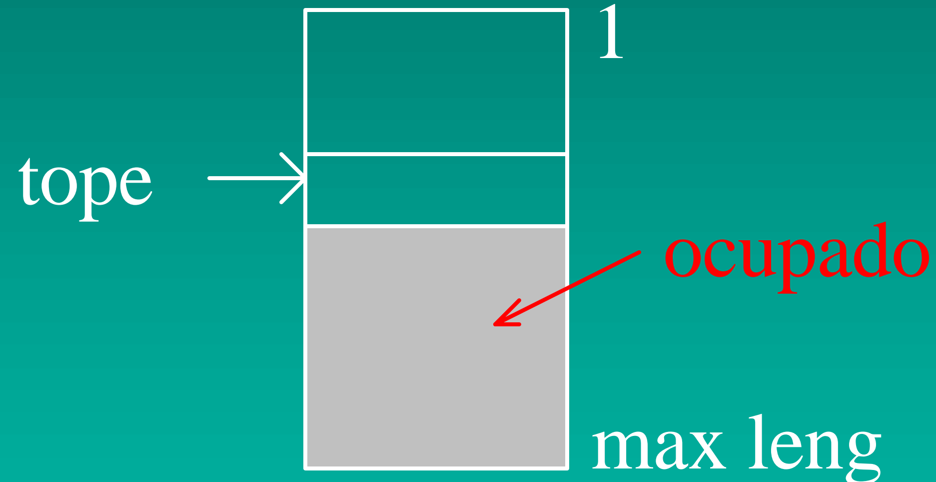
Operaciones

• makenull (s)	crea stack vacío
• top (s)	retrieve (first(s),s)
• pop (s)	delete (first(s),s)
• push (x, s)	insert (x, first (s),s)
• empty (s)	verdadera si S es stack vacío

- Operación costosa: revertir stack (o leer desde el fondo)

Implementaciones:

- Similares a listas
- Conviene modificar la implementación de lista mediante arreglo.



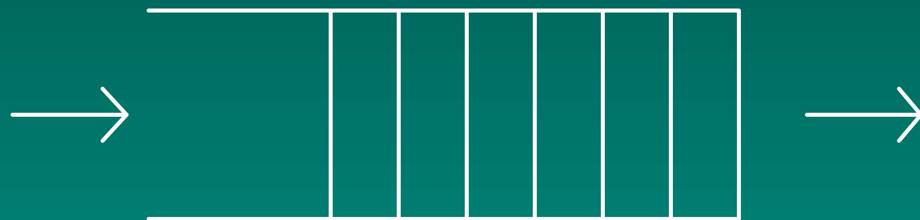
Obs fig. 2.18

- Procedure PUSH (x: element type; var S:stack);

Usos

- Reconocer expresión regular.
- Implementar recursión
- Compilación expresiones infix
  - Conversión exp. de un tipo en otro (ej. polaca -> infix)
  - Polaca o Código.
- Máquina de stack

Colas (FIFO, First In First Out)



in
end
rear

out
first
front

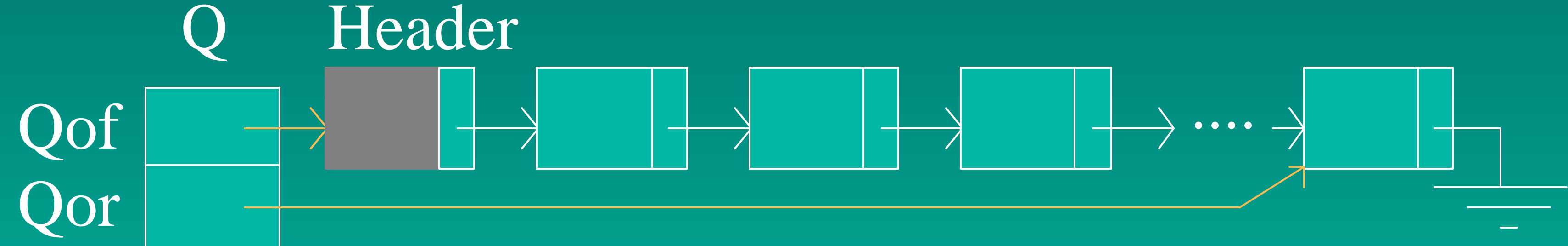
- Inserciones en parte trasera.
- Descartes en parte frontal.

Operaciones:

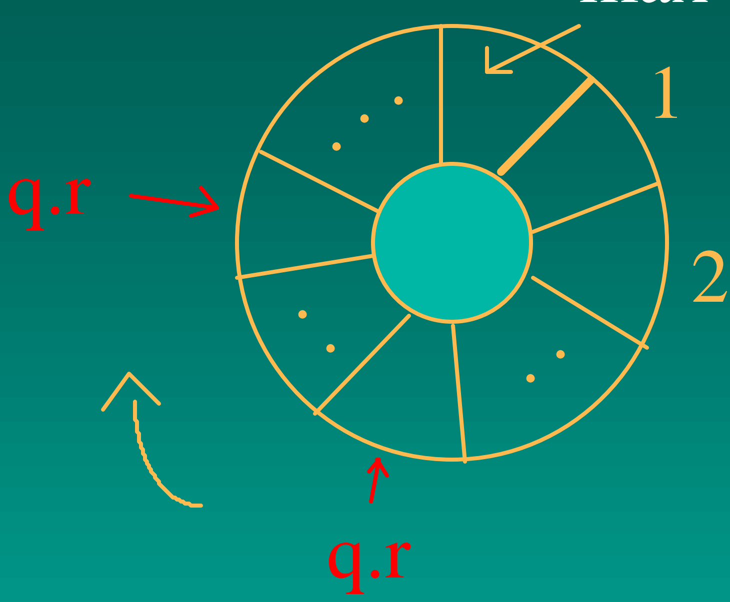
• makenull (Q)	crea cola vacía
• front (Q)	retrieve (first(Q),Q)
• enqueue (X,Q)	insert (x, end(Q),Q)
• dequeue (Q)	delete (first(Q),Q)
• empty (Q)	verdadera si la cola Q está vacía

Implementaciones

a).- Con punteros

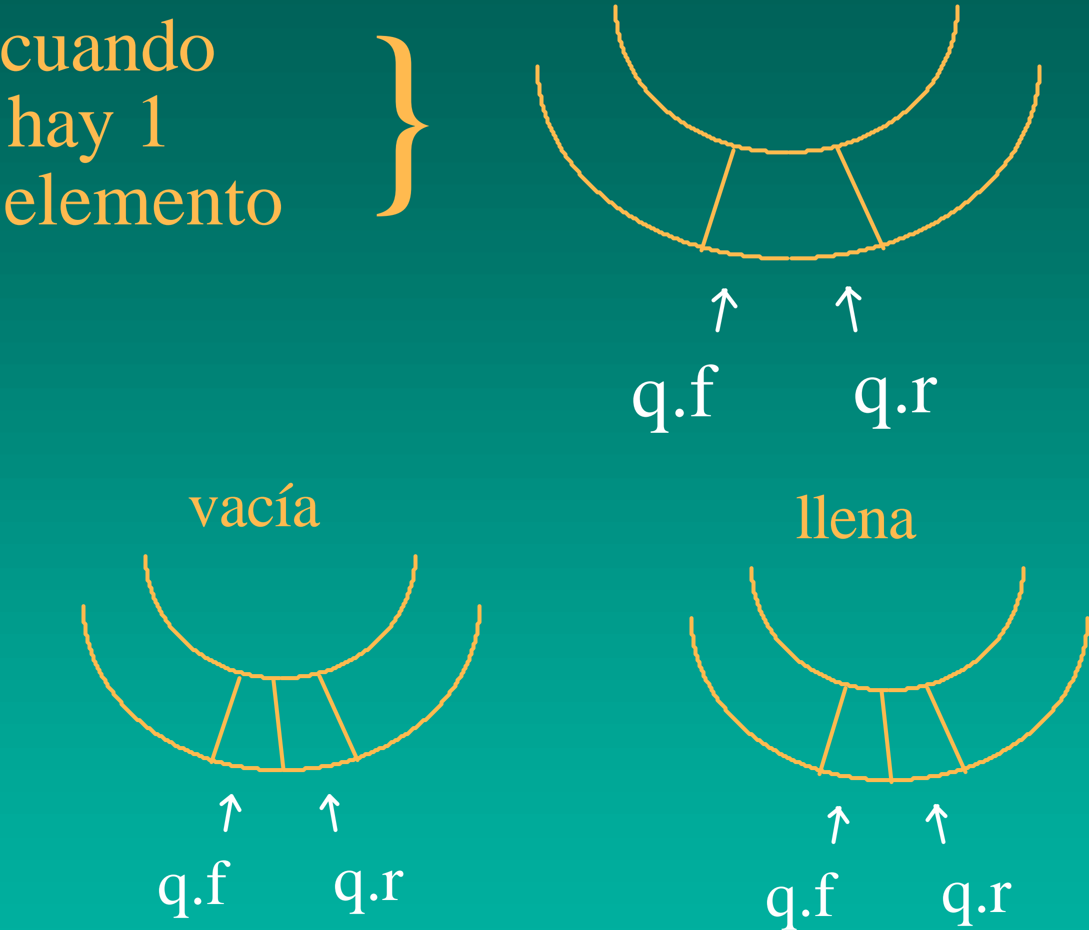


b).- Con arreglos (Buffer Circular)



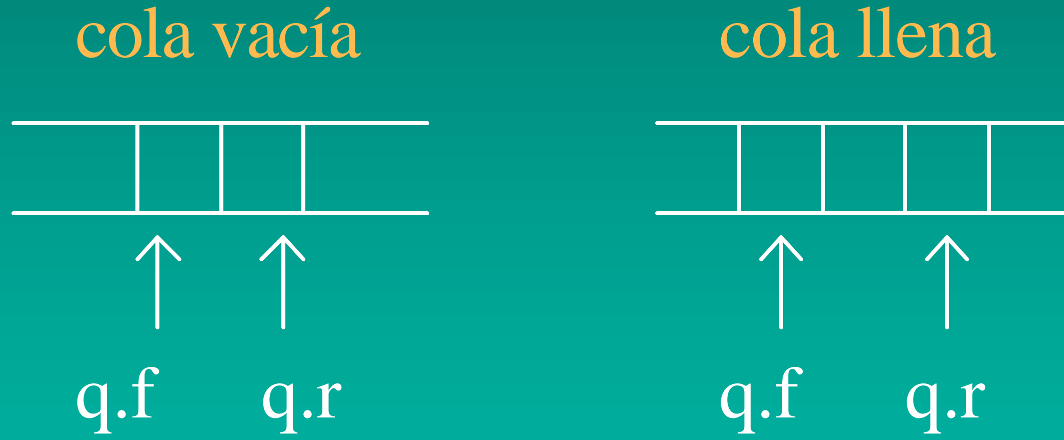
- Las posiciones deben considerarse en aritmética módulo max.
- La cola está formada por los elementos desde q.f hasta q . r (según reloj)
- A medida que se agregan y descartan (producen y consumen) elementos, la cola se mueve según reloj.

- Si q.r apunta al último ocupado, y q.f al que toca consumir



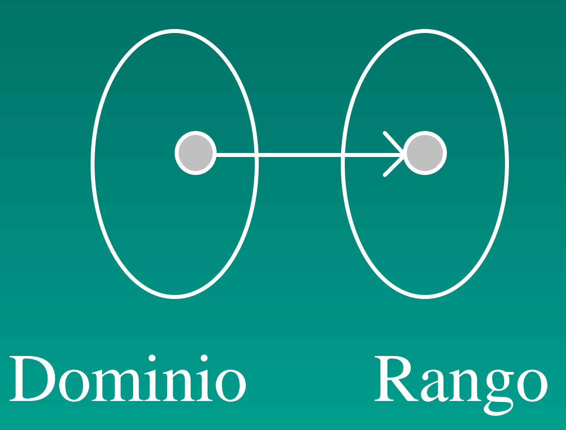
- Por lo tanto, no puede distinguirse entre cola vacía y cola llena

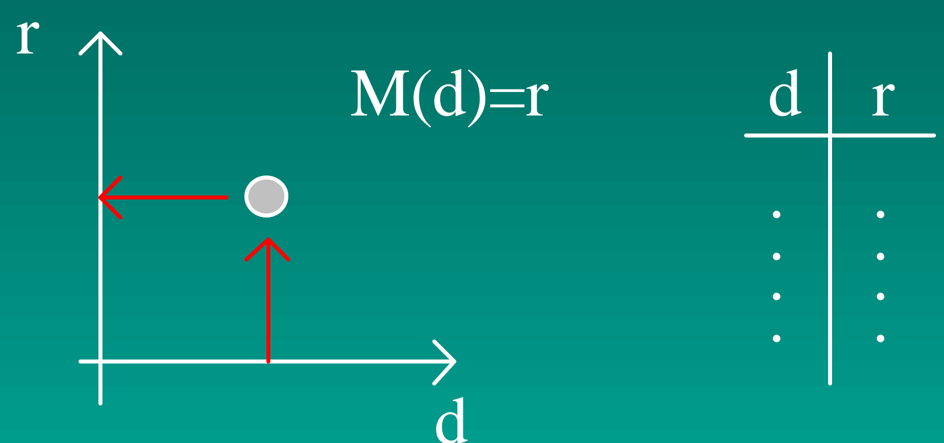
- Soluciones:
  - -Mantiene bit de estado para cola vacía
  - - Contador de posiciones ocupados.
  - - Permitir que la cola crezca en más de max - 1
- En este caso:



(A menudo se emplea q.r para marcar siguiente disponible.)

Mapping (Arreglo Asociativo)





- Si no hay función sencilla que permita expresar M(d), en términos de r, deberá almacenarse la tabla.

Operaciones

- makenull (M) — crea tabla nula
- assign (M,d,r) — define entrada (esté o no definida previamente)
- compute (M,d,r) — - retorna verdadero si M(d) está definida, con r = M(d).
  - retorna falso si M(d) no está definida.

Implementaciones

1.- Con arreglos

2.- Lista de pares

Obs: Lotus y awk, tienen como parte del lenguaje, el manejo de tablas.

Procedimientos Recursivos y Stack

- El conjunto de estructuras de datos que se usan para representar los valores de las variables durante la ejecución se denomina. "organización de tiempo de ejecución".

- Cuando se soporta recursividad, la unidad básica es el "registro de activación", registra variables de un procedimiento activo (y direcciones de retorno,...)

- Cuando se invoca a P, se coloca su registro de activación en el stack (de la máquina virtual).
- Cuando se sale de P, se efectúa el pop del registro de activación
- El esquema anterior soporta naturalmente la recursividad, siempre el registro activo está en el tope del stack

- Se puede eliminar recursividad en el algoritmo, mediante un stack definido por el usuario

- Particularmente se puede eliminar la recursividad por el fondo (o la cola). Si la última acción es el llamado recursivo, este puede eliminarse

```
P(x);
begin .....
        P(y);
end
```

```
P(x);
label 1;
1: begin

x: = y; goto 1;
end;
```

Arboles

Conceptos y Operaciones Básicas (Cap 3 AHU)

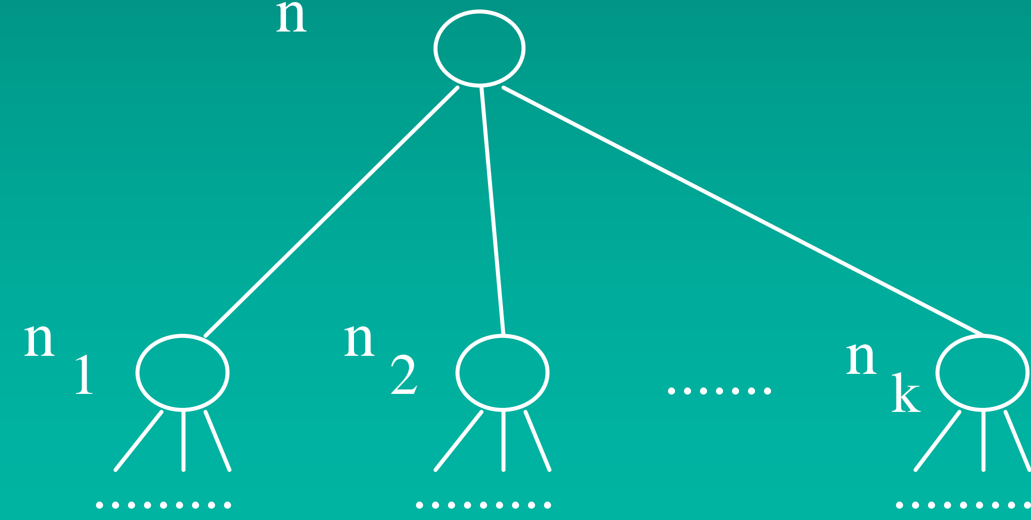
- Estructura jerárquica de componentes
- Un árbol es una colección de nodos con una relación de jerarquía entre ellos.
- Un solo nodo se denomina raíz
- Definición recursiva.
- Un nodo simple es por si mismo un árbol.(en este caso, también es la raíz)

- Sean los árboles Ti

n n_1 n_2 .... n_k

T_0 T_1 T_2 T_k

- Entonces:



Es también un árbol.

- Se dice que:

  - T1, T2, ... Tk son sub-árboles

  - n es la raíz

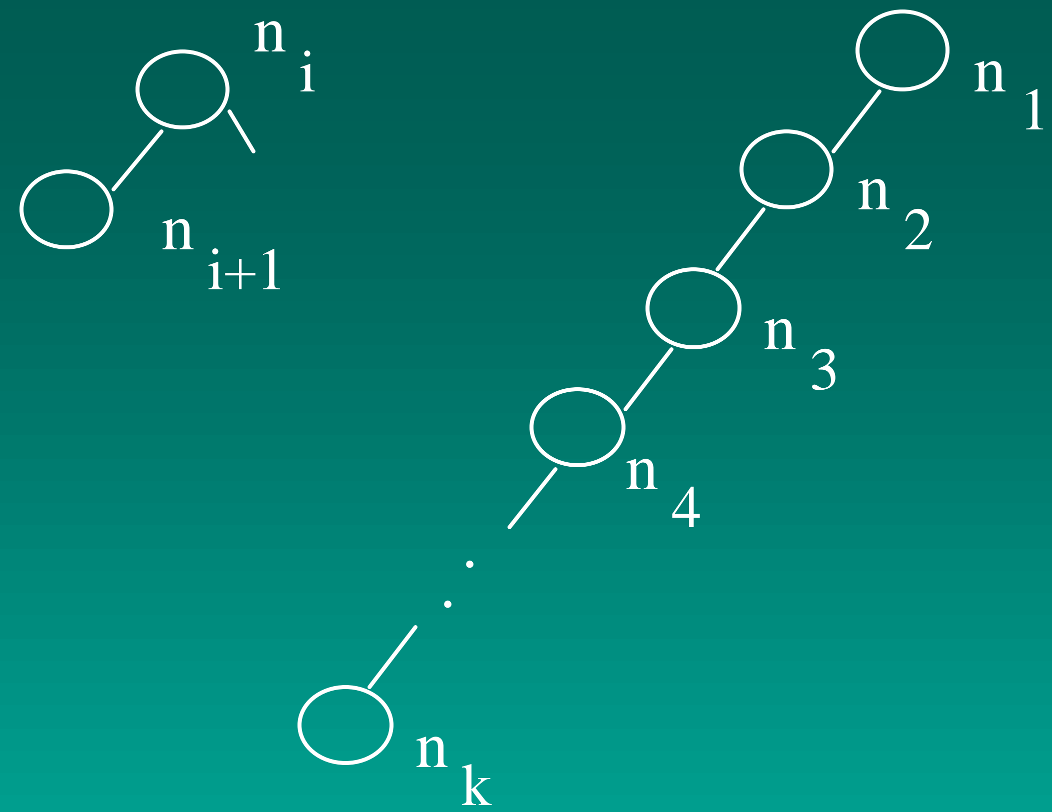
  - n1, n2, ... nk son hijos de n

- Arbol nulo: sin nodos. Suele anotarse Λ .

Definiciones

- **Trayectoria**: secuencia de nodos, tal que ni es el padre de ni + 1

- **Largo** : nº de nodos - 1



- ni es el ancestro de ni + 1
- ni + 1 es un descendiente de ni
- ni es el ancestro y descendiente de ni
- ni + 1 es el ancestro propio

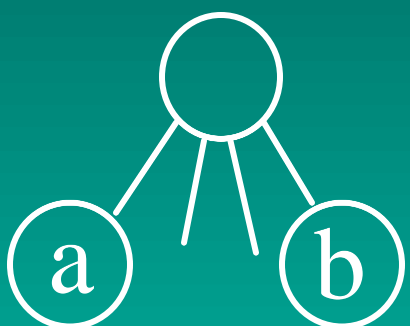
- La raíz no tiene ancestros propios
- Un subárbol es un nodo con todos sus descendientes
- Un nodo sin descendientes propios es una hoja

- **Alto de un nodo** = Largo de la trayectoria más larga de ese nodo a una hoja

- **Alto del árbol** = alto de la raíz

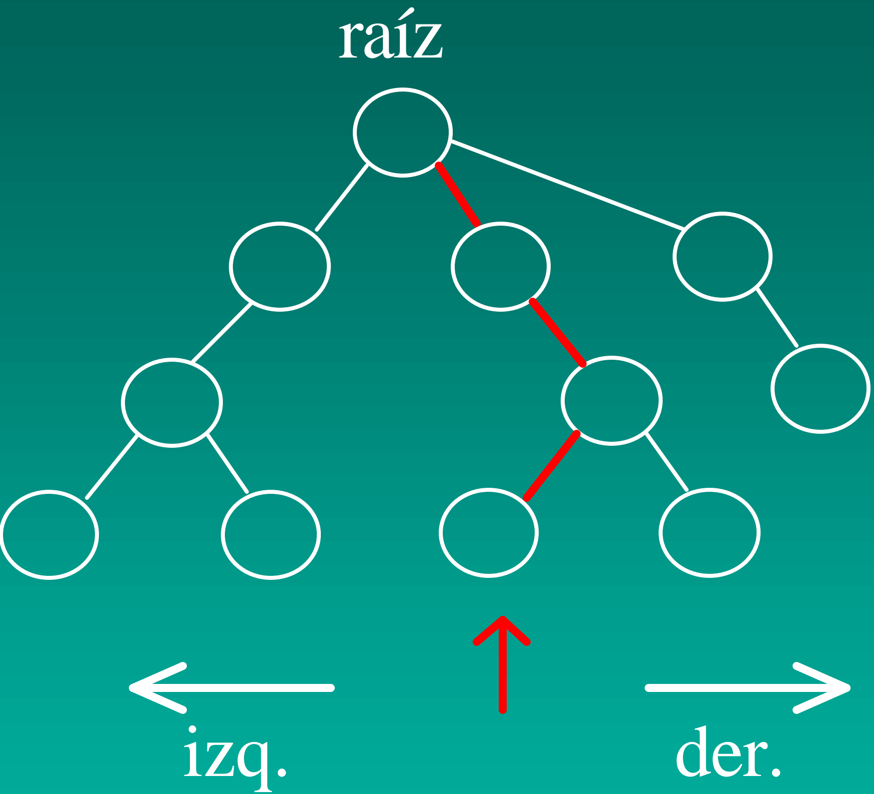
- **Profundidad de un nodo** = largo de única trayectoria de la raíz al nodo

- **Orden**: normalmente de izq. a derecha, en un multiárbol (establece la jerarquía de el árbol)



- Si a está a la izquierda de b; todos los descendientes de a están a la izquierda de todos los descendientes de b

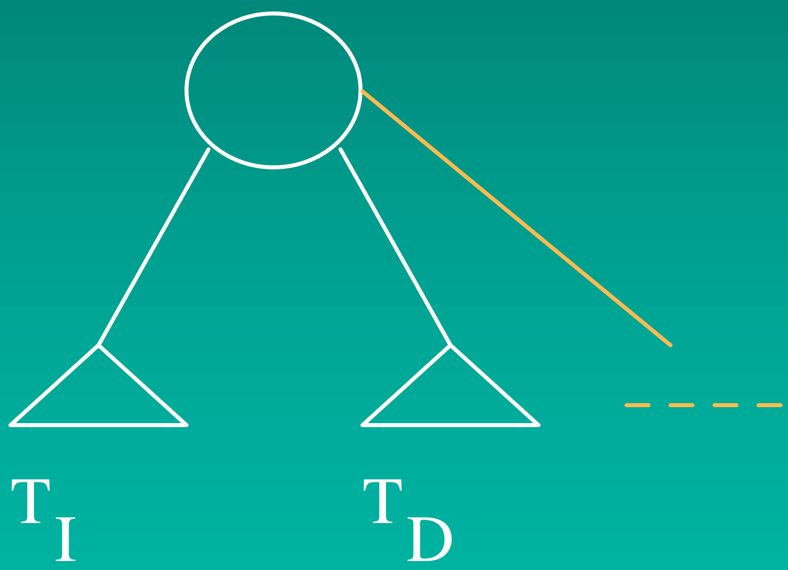
- Si se dibuja la trayectoria de la raíz a un nodo



- Puede visualizarse los nodos ubicados a la izq. y derecha respectivamente

Ordenes Sistemáticos (recursivos)

Pre orden	En orden	Post orden
r TI TD..	TI r TD..	TI TD.. r



Ver Fig. 3.6 Ej. 3.3 pag. 79 AHU.

Rótulo (label)

- **Valor del nodo** (almacenado en el nodo)

- Notación para expresiones

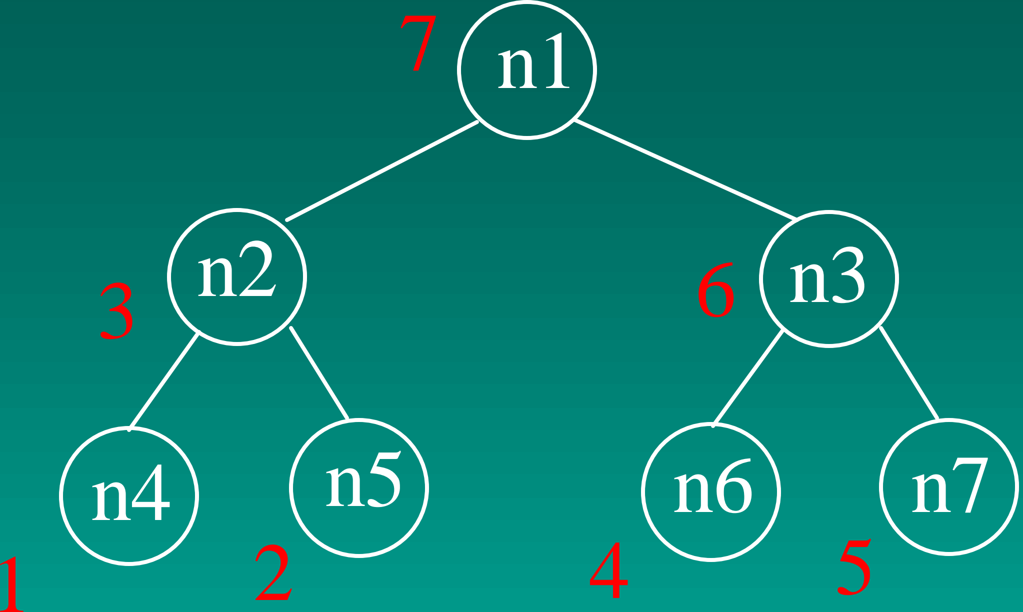
  Si se lista los rótulos, y estos son operadores y operandos en un árbol de expresiones, en preorden, inorden, y post orden, se obtienen las notaciones: prefijo, infix, post fix respectivamente.

Funciones utiles en árboles

Algunas de ellas:

- **post orden (n)** : posición de nodo n en una lista post orden de los nodos de un árbol.
- **desc (n)**: número de descendientes propios de n

Ej:



Lista post orden	:	n4	n5	n2	n6	n7	n3	n1
post orden (n)	:	1	2	3	4	5	6	7
desc (n)	:	0	0	2	0	0	2	6

- En un subárbol de raíz n:

Los nodos quedan numerados desde: postorden (n) - desc (n) hasta postorden (n)

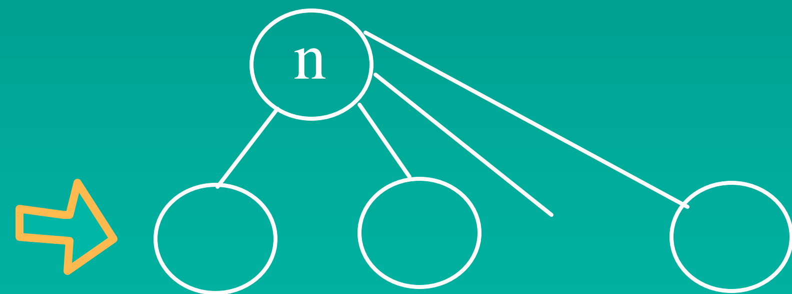
Ej: n2 numerados desde (3-2) a 3; es decir: 1 a 3.

Por lo tanto para ver si x es descendiente de y, debe cumplirse:

$$postorden\ (y) - desc\ (y) \leq postorden\ (x) \leq postorden\ (y)$$

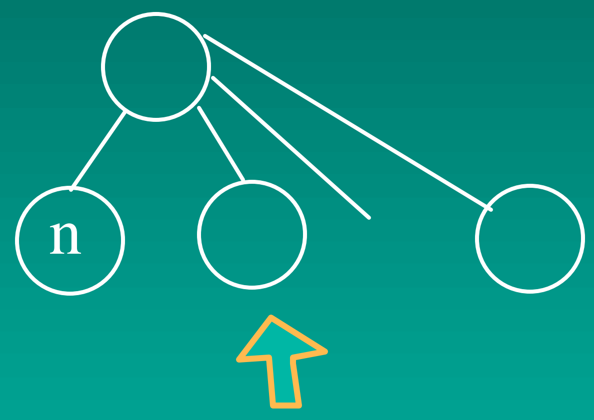
Operaciones

- parent (n, T) retorna: padre de n en T
  Λ si es la raíz.
- leftmost_child (n,T) retorna: hijo más izquierdista de n en T
  Λ si es hoja



Operaciones

- right_sibling (n,T) — retorna: hermano derecho de n en T



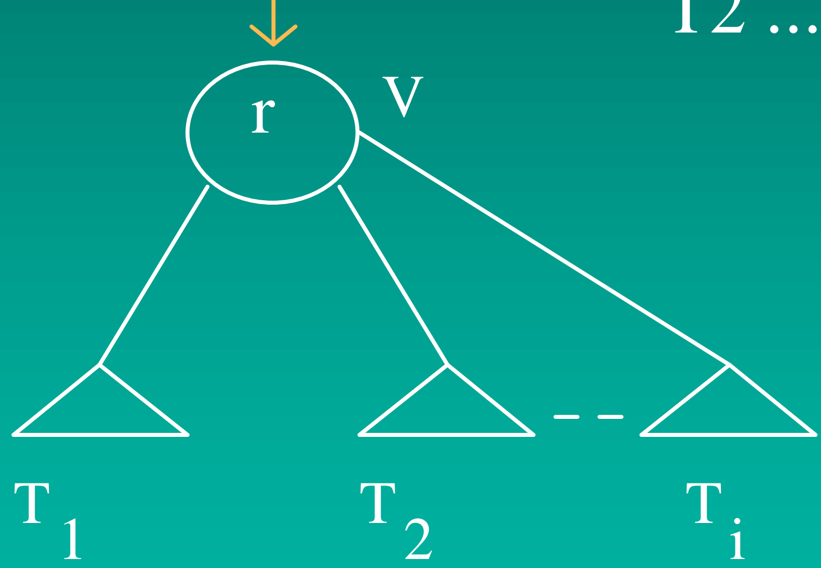
Λ si no tiene

- label (n,T) — retorna valor de rótulo de n en T

Operaciones

- label (n,T) retorna valor de rótulo de n en T

- Createi (v, T1 T2 ...Ti) Crea un árbol con raíz de valor v, y subárboles: T1 T2 ... Ti



Obs create ∅ (v) produce

r
v

Operaciones

- root (T) retorna nodo raíz; si T es nulo
- makenull (T) crea árbol nulo

Ver fig. 3.8 y 3.9 Ej. 3.5 AHU

Implementaciones multiárboles

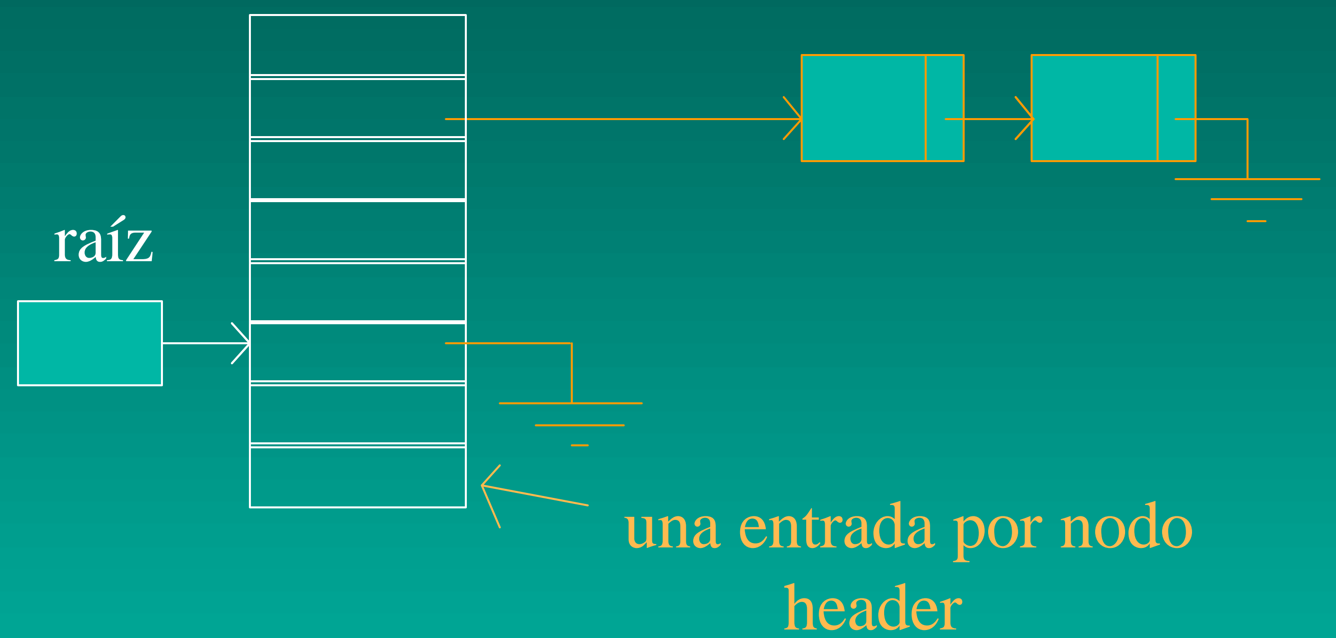
1.- Arreglo de padres

Ver fig 3.10 AHU

- El padre se obtiene en tiempo constante
- No es fácil obtener hijos (no queda definido orden de hijos)

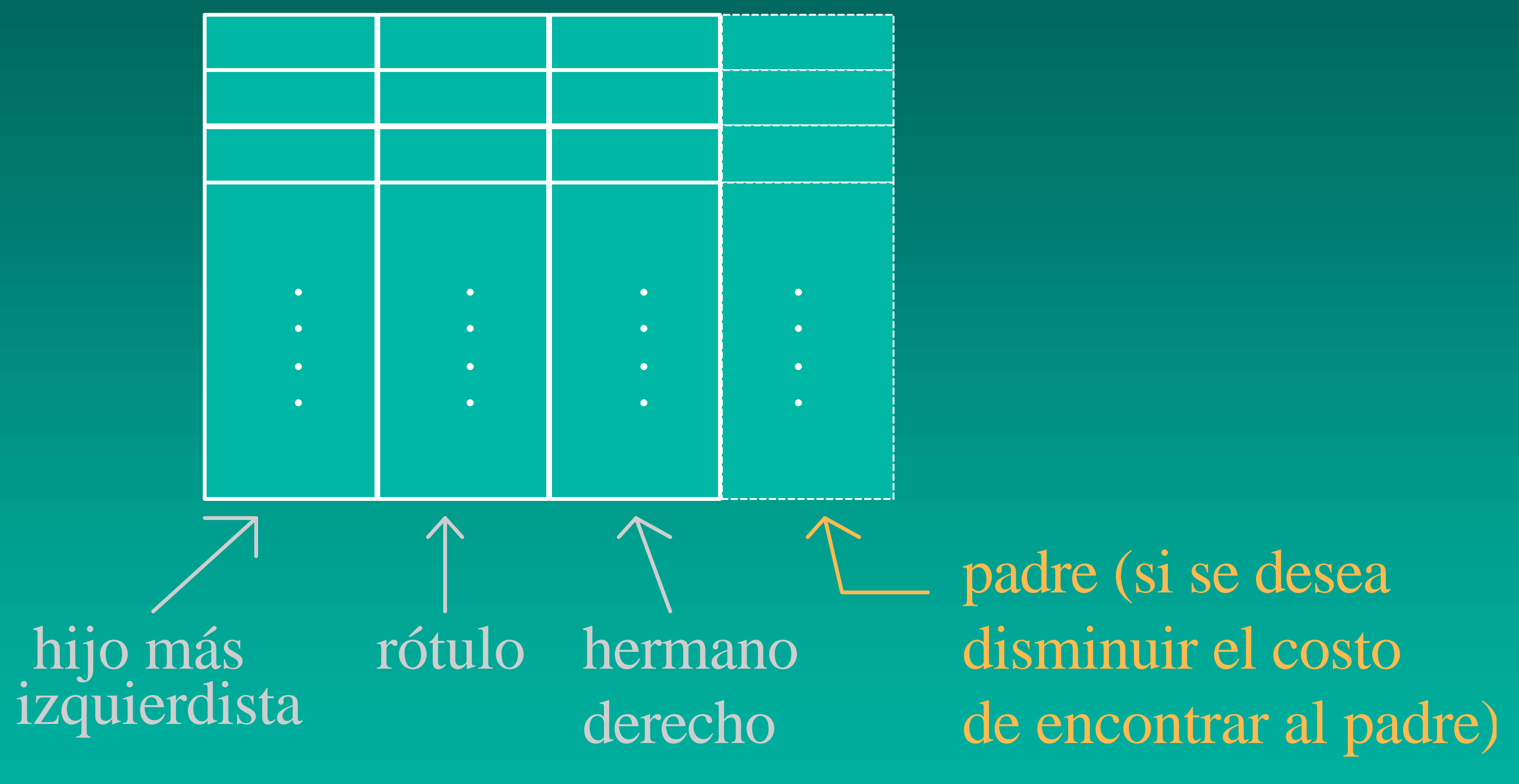
-Se puede imponer orden artificial: Numerando hijos (de izq. a der) después de enumerar al padre.

2.- Lista de hijos



- A cada nodo se asocia una lista de hijos
- Es difícil obtener el padre

3.- Hijo más izquierdista, hermano derecho



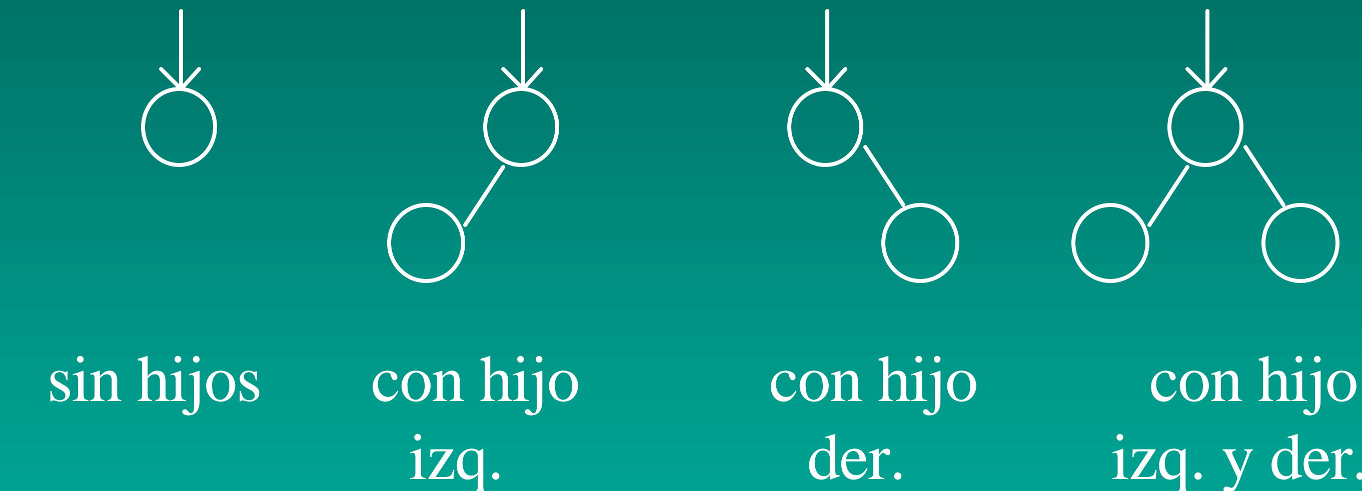
4.- Hay otras formas (árboles 2-3; Btrie)

- Tarea: Leer implementaciones en AHU 3.3. Comparando costos de operaciones,según implementación.

Arboles binarios:

- El multiárbol visto antes es una estructura ordenada y orientada.
- **Ordenada**: porque los hijos de un nodo están ordenados de izquierda a derecha.
- **Orientado**: Porque hay camino único desde un nodo hacia sus descendientes.
- En árboles binarios:

- Def: Un árbol binario puede ser:
  vacío o árbol en el cual cada nodo es



- Se pueden obtener listas en preorden, en orden y postorden

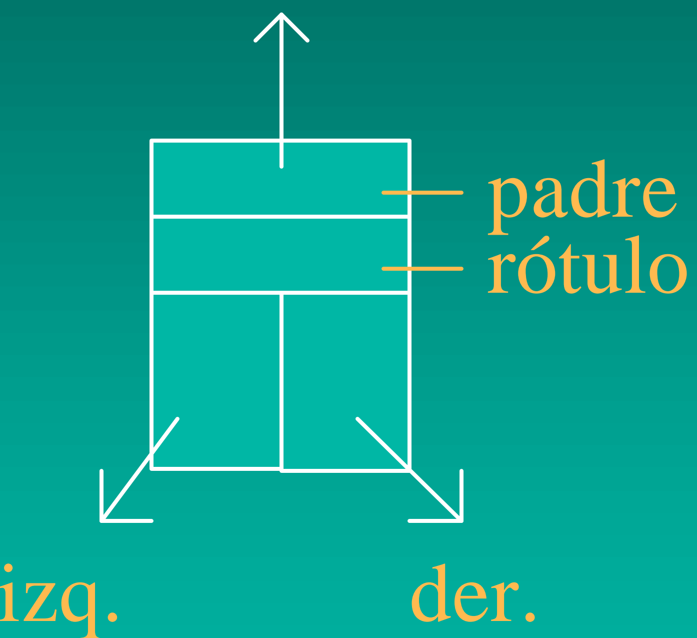
Representaciones

1.- Arreglo de registros

izq.	der.	rótulo	padre
⋮	⋮	⋮	⋮

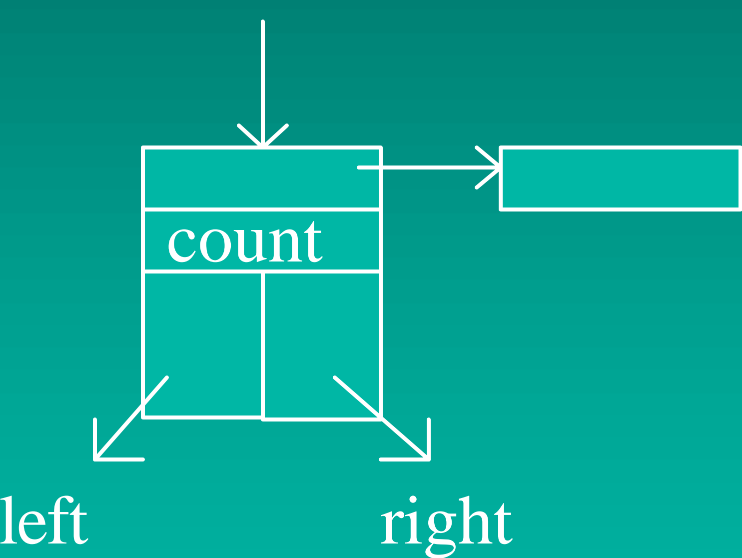
Representaciones

2.- Punteros



- Ejemplo en C, KR 6.5 (pag 130 - 134)

```
typedef      struct tnode {
             char   *word;
             int    count;
             struct tnode  *left;
             struct tnode  *right:
             } nodo;
```



- Estructura permite guardar palabras con un entero asociado.

- Impresión en orden:

```
treeprint (p)
nodo  *p;
{
        if (p!= NULL_T )
        {
        treeprint (p->left);
        printf ("%4d %s\n", p->count, p->word)
        treeprint (p->right);
        }
}
```

```
preorder(p)
...
    printf...
    preorder (p->left);
    preorder (p-> right);
    ...
________________________________
postorden(p)
...
  postorden (p->left);
  postorden (p->right);
  printf...
```

"Las operaciones sobre estructuras de datos definidas recursivamente , son más fácilmente implementados con algoritmos recursivos"

- Tareas: Completar estudio rutinas K.R. 6.5
  Código Huffman . AHU 3.4

Conjuntos

- Def: Colección de miembros (diferentes)
- No importa el orden: {a, b } igual a {b, a }
- No hay elementos repetidos {a, b, a} <- no es
- Pertenecer a un conjunto, se anota $x \in C$
- $\emptyset$ es un conjunto vacío.

Conjuntos

- $A \subseteq B$ A está incluido en B

  A es subconjunto de B

  B es superconjunto de A

- $A \cup B$ en A o B
- $A \cap B$ en A y B
- A - B están en A, pero no en B

Operaciones

- unión (A,B,C) — $C = A \cup B$
- intersección (A,B,C) — $C = A \cap B$
- diferencia (A,B,C) — $C = A - B$
- merge (A,B,C) — unión conjuntos disyuntos
  No está definida si $A \cap B \neq \emptyset$
- member (x,A) — retorna verdadero si $x \in A$
- makenull(A) — $A = \emptyset$

- insert(x,A) — A= A ∪ B{X} Si antes de insert x ∈ A, A no cambia.
- delete (x,A) — A= A - {X} Si antes x ∉ A, A no cambia
- assign (A,B) — A= B
- min (A) — retorna menor valor de A. A debe tener una relación de orden en sus elementos.
- equal (A,B) — retorna verdadero si A y B tienen los mismos elementos.
- find(x) — Existen conjuntos diyuntos, findo retorna el único nombre del conjunto, del cual x es miembro.

Implementaciones

1.- **Vector de bits**

```
const  N= ... ;
type set = packed array [1.. N] of boolean;
var A:set
```

A[i] es true si i $\in$ A

- Ventajas de esta implementación:

  - member, insert, delete se realizan en tiempo constante.

  - unión, intersección, diferencia son O(n)

- Si se desean rótulos (asociados a los enteros) puede usarse otro arreglo.

  Ver Union, fig 4.4 AHU.

Obs

- El set de Pascal es eficiente en universos pequeños
- Si N coincide con el largo de la palabra, se pueden emplear instrucciones (de máquina) lógicas para las operaciones.

2. Lista Ligada

Conviene que la lista esté ordenada. Con esto no es necesario recorrer toda la lista para determinar si un elemento está en la lista.

(Lo anterior puede hacerse si el conjunto universal cumple relación de orden lineal)

Ej: Análisis de op. intersección

a) Lista no ordenadas

Debe determinarse si un elemento está en ambas listas. Es decir, debe buscarse cada elemento de L1 en L2, esta operación es $O(n^2)$

Sean L1= {c, b , a, f } L2 {f, d, c}

Se busca c en L2 (3)

Se busca b en L2 (3)

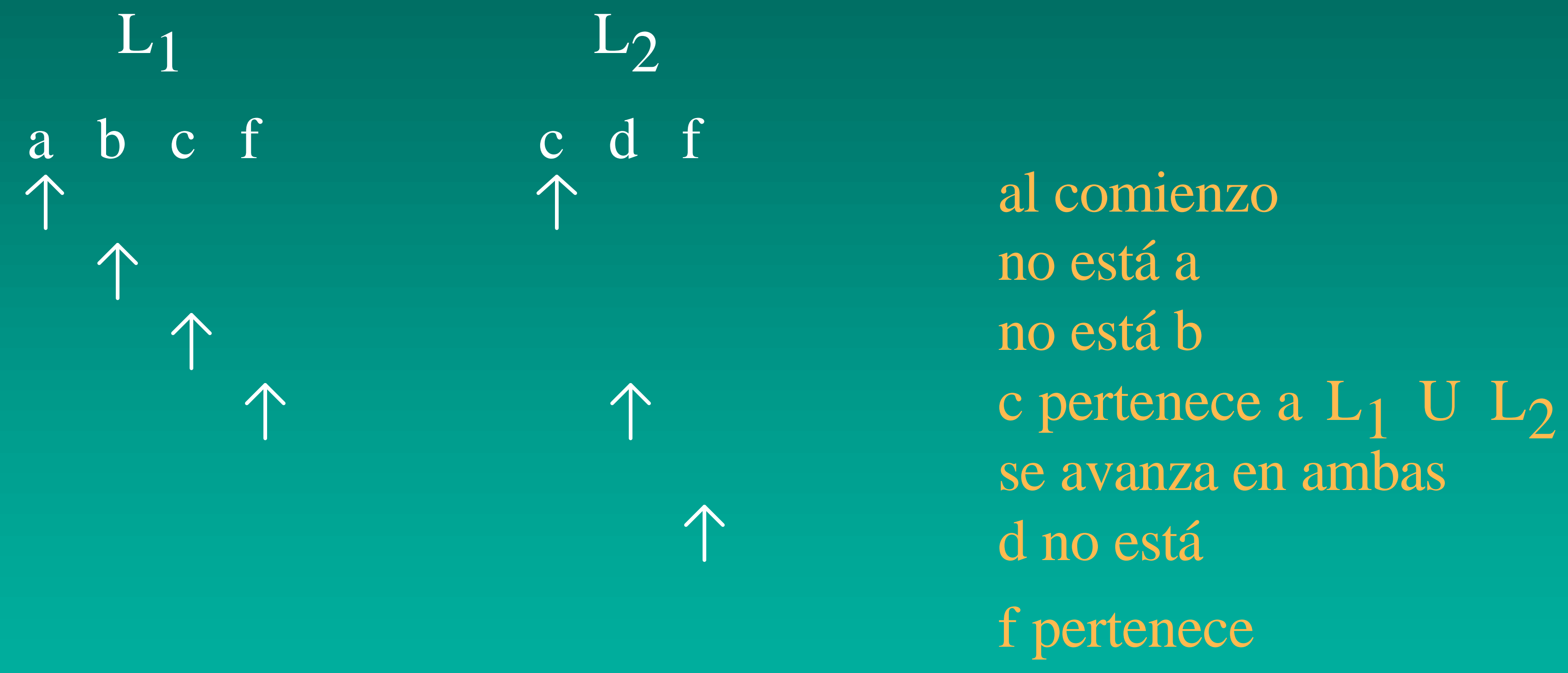
Se busca a en L2 (3)

Se busca f en L2 (1)

- Si L2 tiene n componentes; si no está, se requieren n op.
- Si el elemento está, en promedio, se requiere n/2
- Si L1 es de largo m, se requieren m x n en peor caso.

b) listas ordenadas

Se barren ambas listas, sólo una vez. Si se avanza en lista con elemento menor



- La op. en este caso es O(n)

  Ver fig. 4.5 AHU

Diccionario

- Conjunto con operaciones: insert, delete, member
- Un caso frecuente es con operaciones: instalar, buscar (tabla símbolo)
- En puro diccionario sólo se implementa: buscar.

Implementaciones

a) con arreglos

Ver fig 4.9 AHU

Aplicación fig. 4.8 AHU

Dificultades: descarte lento. Ocupación ineficiente del espacio, si los elementos son de largo variable.

Operaciones son O(n)

b) Listas ligadas ordenadas

Operaciones son O(n)

c) Tabla de Hash

En promedio las operaciones requieren tiempo constante, en pero caso son O(n).

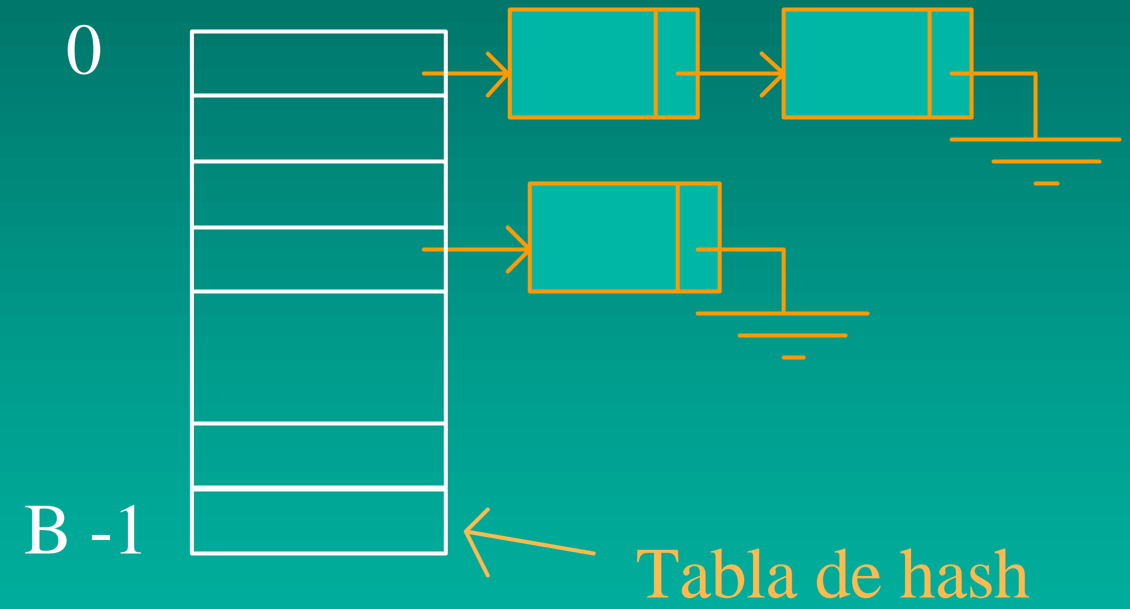
Hash= desmenuzado, picado, molido

- Cada item del conjunto debe caracterizarse por un valor de la clave (sobre la cual existe una relación de orden)
- Se desea encontrar un item, con una clave con el menor esfuerzo posible.
- Para todo x(clave) perteneciente al Universo, debe estar definida h(x) (función de Hash), con valores enteros, entre 0 y B -1

- h debe distribuir las claves, lo más equitativamente posible
- Si h(x1) = h (x2) = Bi   se dice que hay colisión
- Si la función es buena, habrán pocas colisiones.Si hay c colisiones en promedio, las operaciones resultan O(c), de tiempo constante; prácticamente independiente de n.
- Si todas colisionan (peor caso), se tendrán operaciones O(n).

Hash abierto o externo

- Las colisiones se resuelven en una lista



(Direct chaining hashing)

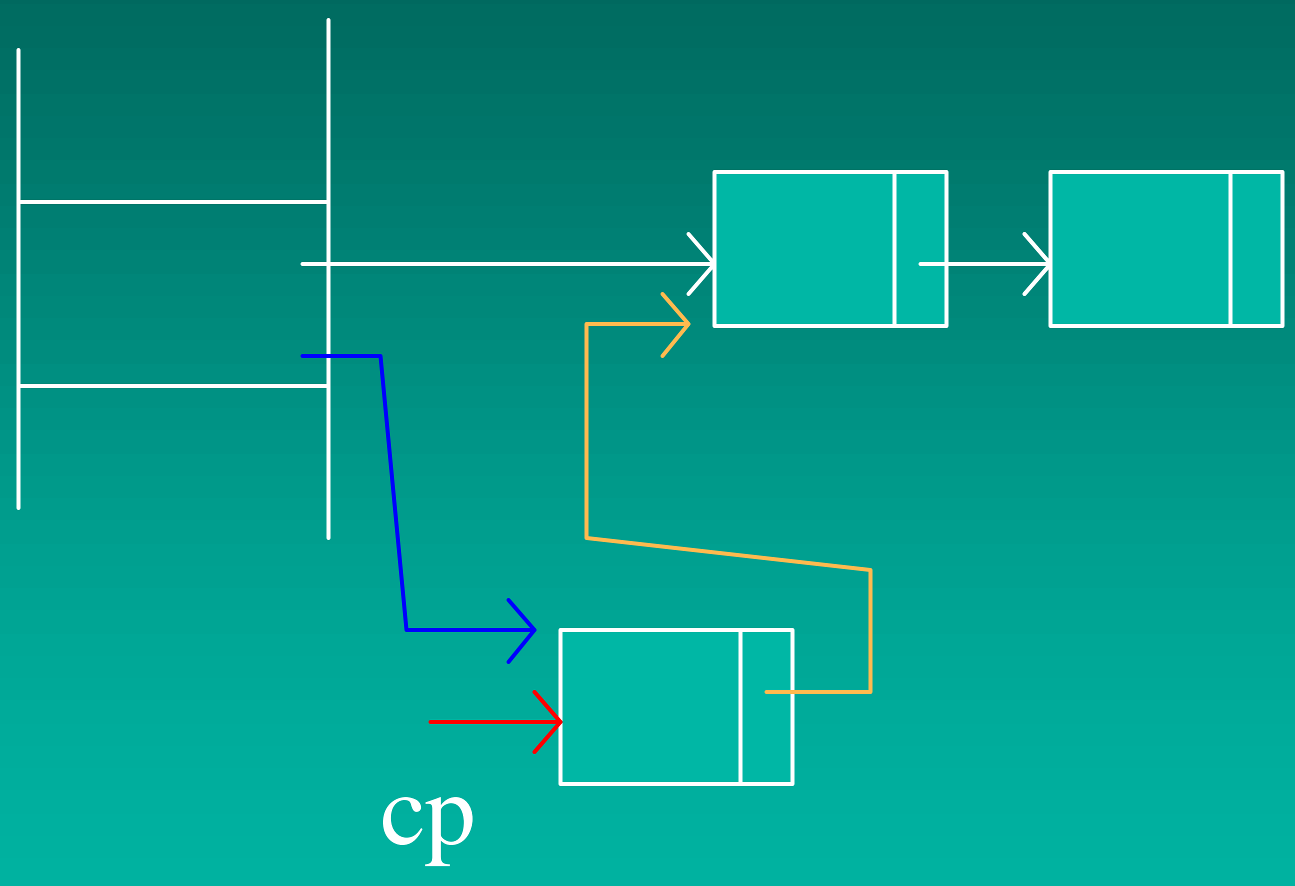
Creación espacio dinámico

```
typedef  struct    celda {
                   char      *nombre;
                   struct    celda     *next;
                   tcelda,   *pcelda;

#define  B 100;    /*        100       celda     */
Static   pcelda    Hashtab[B];  /*tabla punteros */
#define  NULLP  (pcelda)NULL
makenull( )
   {   int i;
       for ( i = 0; i < B; i + +)  hashtab [i] = NULLP;
   }
h(s)                         /* función simple de hash */
char *s;
   { int hval;
   for (hval =0; *s!='\0';)  hval + = *s++;
   return (hval % B);
   }
```

```
pcelda member (s)
   char *s
   {
         pcelda cp;                    /* current pointer */
         for  (cp = hashtab[h(s)]; cp!= NULLP; cp = cp ->next)
                if (strcmp(s, cp ->nombre ) = = 0)
                      return (cp);               /* lo encontró */
         return (NULLP);
}
```

```
pcelda   insert(s)
         char *s;
   {                pcelda  cp, member ( );
                    char  *strsave ( ), * alloc ( );
                    int  hval;

         if (( cp = member (s)) = = NULLP) {
                    cp =  (pcelda ) alloc (sizeof ( tcelda ));
                    if (cp = = NULLP) return (NULLP);
                    if (( cp-> nombre = strsave (s)) = = NULLP )
                                        return (NULLP);
                    hval = h (cp -> nombre);
                    cp -> next = hastab [hval];
                    hashtab [hval] = cp;
}
return (cp);
}
```

- Insert



- Crea espacio y guarda string, retorna puntero al lugar depositado.

```
char   *strsave(s)            /* K.R. pág 103 */
char *s;
{
        char *p, *alloc ( );
         if (( p =alloc (strlen (s) +1)) ! = NULL)
                 strcpy(p,s);
         return
}
```

```
pcelda    delete (s)
    char  *s;
    {
            pcelda  q, cp;
            cp =hastab [h(s)];
            if (cp ! = NULLP ) {
                if (strcmp (s, cp -> nombre ) == 0)      /* primero de lista */
                        hastab [h(s) ] = cp * next;
            else
               for (cp = cp ->next; cp! = NULLP; q = cp, cp = cp ->next )
                        if (strcmp (s, cp * nombre ) = = 0) {
                                q ->next = cp ->next
                                break
                        }
               if (cp! =NULLP ) {
                        free (cp -> nombre);
                        free ( ( char * ) cp );
            }
```

Eficiencia (hashing abierto)

a).- Caso ideal h produce distribución uniforme

Con tabla de B entradas

Con n elementos en el sistema

Factor de carga = α = n/B

- Resultan listas de largo n/B en promedio
- Por lo tanto las operaciones demanda, en promedio:

$O(1 + n/B)$

buscar en lista (n/B)

encontrar balde (1)

- Con B $\sim$ n resultan O (cte)

- La construcción de la tabla demanda :

$$O\,(n\,(\underbrace{1 + n/B}_{\text{busqueda}}))$$

b).- Distribución Binomial (Gonet 57 pag)

$$P\left[\text{lista largo i}\quad\right] = \left(\frac{n}{i}\right)\left(\frac{1}{B}\right)^{i}\left(1-\frac{i}{B}\right)^{n-i}$$

Gráfico en "Martin. Computer Date Base Org." pag 382

Hashing Cerrado

- Los elementos se guardan en la misma tabla (sin listas)
- Debe conocerse si un elemento de la tabla está vacío, ocupado o descartado.
- Las colisiones implican una estrategia de rehash.
- Si h(x) está ocupado, debe buscarse en hi (i = 1, 2,....).
- Si todas las localizaciones están ocupadas, se dice que la tabla está llena.

Hash lineal

$$h_i(x) = (h(x) + i) \bmod B$$

- **La búsqueda**: Se detiene cuando se encuentra vacío; debe seguir si se encuentra descartado.
- **En inserción**: Se coloca item en primer vacío o descartado.

```
typedef  enum (vacío, ocupado, descartado ) estado;

typedef    struct celda{
               char id [12];
               estado     state;
               } tcelda;
static tcelda hastab [B];    /* tabla hash cerrado */
```

- Existen B! trayectorias posibles dentro de una tabla.
- Primera celda de la trayectoria, puede escogerse de B formas.
- La segunda celda puede escogerse de B -1 formas

Hash cerrado lineal

- Búsqueda: Gonnet 3.3.3 pag. 43

```
i = h(s);
last = ( i +n -1 ) % B;
while (i ! = last   &&   hashtab [i] . state ! vacío
                    &&   atremp (s, hastab [i] . id ) ! = 0)
      i = (i + 1 ) % B;    /* lineal */
if (strcmp (s, hastab [i] . id ) = = 0 ) return (  i  );
else
      return ( -1 );
```

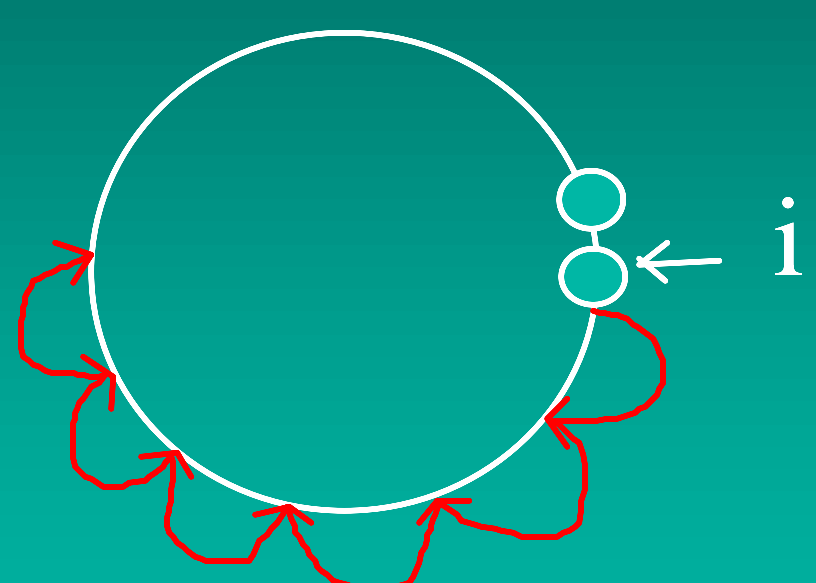
- Se resuelven las colisiones probando en la siguiente localización de la tabla.

- Se usa una trayectoria circular. Para detener la búsqueda en la tabla, debe alargarse el recorrido circular. Para esto se emplea la variable last.

- Nótese que (i -1) % B (o mejor (i-1 + B) % B)) es el final de un recorrido circular, al que se llega después de 8 intentos (se cuenta el intento en i)

- Si hay n ocupados, debe detenerse la búsqueda en:

$$(i - 1 + n) \% B$$



Inserción

```
i = h (s);
last = ( i - 1 + B ) %;
while ( i ! = last          && hashtab [i].state ! vacío
                            && hashtab [i]. state ! = descartado
                            && strcmp (s, hashtab [i].id ) ! = 0 )
        i = (i + 1 ) % B;
if ( hashtab [i].state==vacío || hashtab [i].state==descartado)
   {    strcpy (hashtab [i].id,s);
        hashtab [i].state = ocupado;
        n + +;
   }
else
{       /* Error: tabla llena o ya estaba en la tabla */
```

- Se produce apilamiento

- Mientras más larga es una secuencia, más probable son las colisiones con ella, cuando se desea agregar nuevos elementos a la tabla.

- **Además**: Una secuencia larga tiene gran probabilidad de colisionar con otras secuencias ya existentes. Lo cual tiende a dispersar la secuencia.

Otro métodos:

- Uniforme:

  - Se prueba en trayectoria definida por una permutación de los números entre 0 y B - 1.Para cada entrada hay una permutación

- Random

  Función random que genera un entero entre 0 y B-1, permite elegir el siguiente intento.

- Doble

  - Se usa otra funsión de hash, para calcular un incremento (número entre 0 y B -1); mediante el cual recorrer la tabla.

- Cuadrático

  Se prueba en :

  ```
  i = ( i + inc + 1 ) % B;
  inc + = 2;
  ```

- Etc.

Análisis complejidad en hash cerrado N.W. 272

Se asume que todas las claves posibles son igualmente probables, y que h las distribuye uniformemente en los baldes.

- Inserción

Se desea insertar clave en tabla de n posiciones, vuando hay k posiciones ocupadas

- La probabilidad de encontrar un balde disponible en el primer intento de inserción es:

$$P_1 = \frac{\text{disponibles primera vez}}{\text{total primera vez}} = \frac{n - k}{n}$$

- La probabilidad de que exactamente se requieran dos intentos, es el producto de tener colision la 1ª vez y la de encontrar posición libre en el segundo intento:

$$P_2 = \text{colisión } 1^{a} \text{ vez} * \frac{\text{disponible } 2^{a} \text{ vez}}{\text{total } 2^{a} \text{ vez}}$$

$$= \frac{k}{n} * \frac{(n-1)-(k-1)}{(n-1)} = \frac{k}{n} * \frac{n-k}{n-1}$$

- La probabilidad de colisión en 2º intento es :

$$\frac{k - 1}{n - 1}$$

- Encontrar balde disponible en 3º intento:

$$\frac{n - k}{n - 2}$$

- Entonces:

$$P_3 = \frac{k}{n} \cdot \frac{k - 1}{n - 1} \cdot \frac{n - k}{n - 2}$$

- Nótese que en la última fracción el numerador siempre será : n - k

- Finalmente, inserción después de i intentos:

$$P_i = \frac{k}{n} * \frac{k-1}{n-1} * \frac{k-2}{n-2} \cdots \frac{k-i+2}{n-i+2} * \frac{n-k}{n-i+1}$$

- El número esperado de intentos requeridos para insertar una clave, cuando ya hay k en una tabla de n posiciones es :

$$E_{k+1} = \sum_{i=1}^{k+1} i * p_i$$

También es el número de intentos esperados, para insertar la clave (k + 1).

- Nótese que :

$$p_{k+1} = \frac{k}{n} * \frac{k-1}{n-1} * \frac{k-2}{n-2} \cdots \frac{1}{n-k+1}$$

Observaciones :

- Poner el primero cuando la tabla está vacía

$$E_0 + 1 = 1 \qquad\qquad (k = 0)$$

- Poner el segundo, cuando hay uno en la tabla

$$(k = 1)$$

$$E_{1+1} = 1 * \frac{n-1}{n} + 2 * \frac{1}{n} * \frac{n-1}{n-1} = \frac{n+1}{n-1+1}$$

- Poner el tercero, cuando ya hay dos

(k = 2)

$$E_{2+1} = 1*\frac{n-2}{n} + 2*\frac{2}{n}*\frac{n-2}{n-1} + 3*\frac{2}{n}*\frac{2-1}{n-1}*\frac{n-2}{n-2} =$$

$$= \frac{n+1}{n-2+1}$$

- En general, el resultado de la sumatoria es:

$$E_{k+1} = \frac{n+1}{n-k+1}$$

Búsqueda

- Se desea calcular E, el número de intentos necesarios para accesar una clave cualquiera en una tabla en la que hay m elementos :

" El número de intentos para buscar un item, es igual al número de intentos para insertarlo "

- E = nº promedio de intentos para meter m elementos en la tabla.

Para el primero se requieren E1 intentos

Para el segundo se requieren E2 intentos

....

Para el m avo se requieren Em intentos

- En promedio :

$$E = \frac{E_1 + E_2 \dots E_m}{m}$$

$$E = \frac{1}{m} \sum_{k=1}^{m} E_k = \frac{n+1}{m} \sum_{k=1}^{m} \frac{1}{n-k+2}$$

- Para evaluar la sumatoria se emplea la función armónica

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$$

- Ya que para ella existe una aproximación

$$H_n \approx \ln(n) + g$$

$g = 0,\ 577216\ldots$ constante de Euler

- Entonces

$$\sum_{k=1}^{m}\frac{1}{n-k+2}=\frac{1}{n+1}+\frac{1}{n}+\frac{1}{n-1}+\frac{1}{n-2}+\ldots+\frac{1}{n-m+1}$$

- Si se suma en ambos lados:

$$\frac{1}{n-m+1}+\frac{1}{n-m}+\ldots+\frac{1}{3}+\frac{1}{2}+1$$

- Se logra: (Gonnet 3.3.1 pag 40)

$$E = \frac{n+1}{m}(H_{n+1} - H_{n-m+1})$$

$$= \frac{n+1}{m}(\ln(n+1) - \ln(n-m+1))$$

$$= \frac{n+1}{m}\ln(\frac{n+1}{n-m+1}) = \frac{1}{(\frac{m}{n+1})}\ln(\frac{1}{(1-\frac{m}{n+1})})$$

$$E = -\frac{\ln(1-a)}{a}$$

- Se tiene :

$$\boldsymbol{a} = \frac{m}{n+1}$$

Tabla vacía $\boldsymbol{a} = 0$

Tabla llena $\boldsymbol{a} = \frac{\mathrm{n}}{\mathrm{n}+1} \approx 1$

- Factor de carga : $\frac{m}{n} \approx \alpha$

- Los resultados numéricos explican el comportamiento excepcionalmente bueno del método de hashing

" En promedio se requieren de 3 ó 4 intentos, con tabla casi llena"

- Para rehash lineal : $E = \frac{1 - \alpha/2}{1 - \alpha}$ Gonnet 3.3.2 pag 42

- Obs. Hash es mejor que árboles de búsqueda en inserción y búsqueda.

- Pero tiene limitaciones

  - Es un método estático (el tamaño de la tabla debe ser estimado a priori)

  - Complejidad aumenta si se implementa operación descarte.

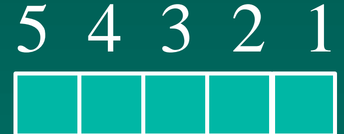
  Si el volumen cambia dinámicamente (creciendo hasta límites no previsibles) (o disminuyendo) se suelen emplear árboles de búsqueda.

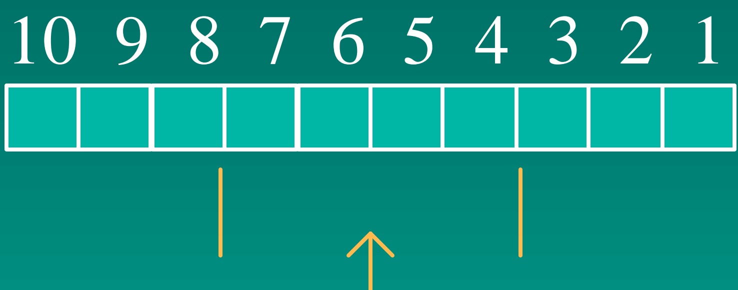
Funciones de hash

- Para claves alfanuméricas suele usarse la suma de los carácteres. Esto por su simplicidad y buen comportamiento experimental (ver ej 4.5 en AHU)

- A continuación se estudiarán otros métodos.

1.- Elevar el cuadrado y sacar el medio





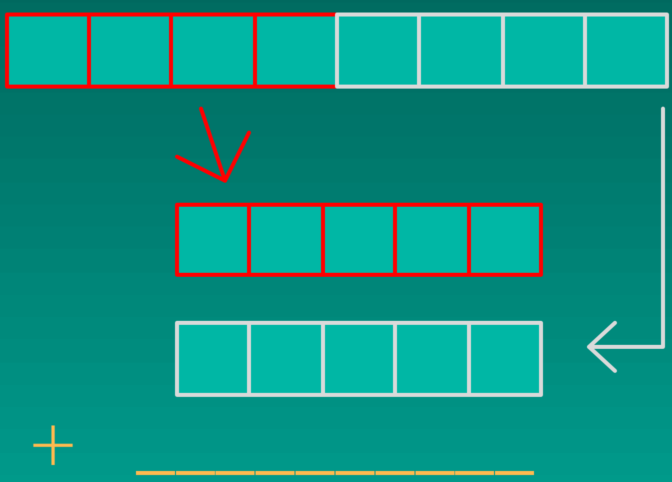
Esta parte depende más fuertemente de todos los dígitos

2.-División:

$$\left(\frac{clave}{n^{\circ}\ \ \text{primo}}\right) \text{mod} \quad B$$; n primo cercano a B

3.- Corrimientos

Corrimientos son eficientes con instrucciones de máquina



4.- Un ejemplo

```
hash (s)
char  *s
{ int c, n ;
  for ( n= 0; c= *s; s + + )
  n + = ( c*n + c << ( n% 4 )):
  return (n);
}
```

+ ___________

dirección del balde

Colas de Prioridad Seleccionar

- Def.: Conjunto con operaciones :
  - insert
  - deletmin (busca y descarta el menor)

(prioridad $\cong$ anterioridad. El más importante).

- Ver Ej. fig. 4.18 AHU Prioridad de procesos.

Implementaciones

- En tabla de hash cuesta encontrar mínimo
- Listas

  - **Ordenada**: Fácil búsqueda mínimo (O(1))
    Inserción (O(n))
    Peor caso: recorrer toda la lista : n
    Optimo: a al primera
    Promedio: n/2
  - **Desordenada**: Inserción: Tiempo constante
    Búsqueda: O (n)

Ver fig. 4.19 AHU

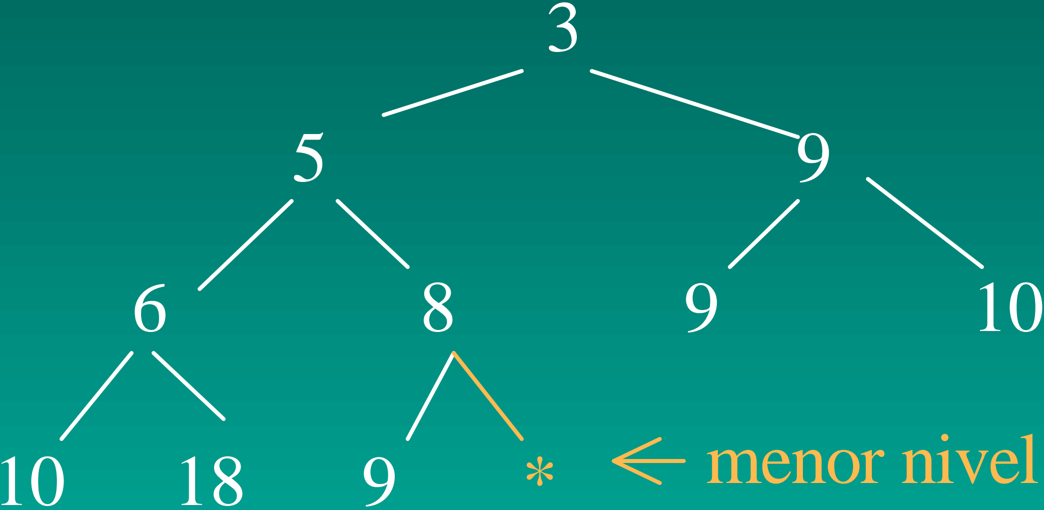
- Arbol parcialmente ordenado. = O(log(n)) en inserción y selección

a) Arbol binario lo más balanceado posible

En el nivel más bajo (cerca de hojas) las hojas que faltan están a la derecha de las presentes

b) Parcialmente ordenado

valor (hijo) >= valor (padre)



Obs

* Posición para insertar

- Pueden existir repeticiones

- El mínimo está en la raíz

Op. Descarte mínimo

- Se saca la raíz. Se corta el árbol
- Hoja más derechista, del menor nivel, se coloca de raíz.
- Empujar, por intercambio con hijos, desde la raíz. Para mantener el orden parcial.
- Complejidad : trayectoria de raíz a hojas, por el camino más largo.

Esto es O (lg2 (n))

Op. Inserción

- Se coloca en menor nivel, lo más a la izquierda posible
- Se asciende, por intercambio con padre. Manteniendo el orden
- Complejidad: O (lg2 (n))

Fig 4.20, 4.21, 4.22 AHU

Arboles completos Binarios

	Nodos	Niveles	Altura
	3	1	2
	7	2	3
	15	3	4
. . . . . .	n	m	h
	$n = 2^h - 1$	$h = m + 1$	$h = lg2 (n + 1)$

Nodos a revisar desde la raíz a los hijos

Arboles con un nivel de desbalance

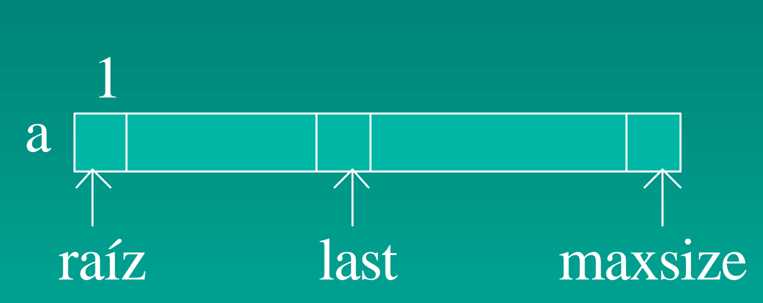
	Niveles	Altura
4 5 6	2	3(peor caso)
De 8 a 14	3	4
$2^{h-1} \leq n \leq 2^{h} - 1$		h

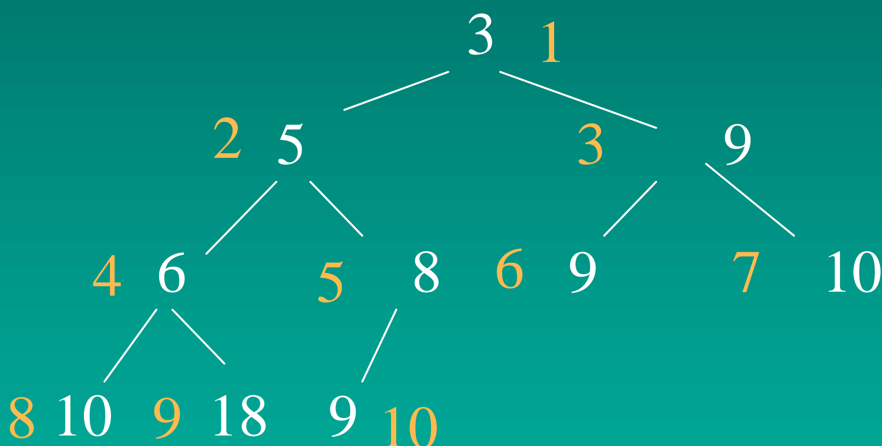
- Trayectoria más larga de raíz a hoja:

  Parte entera (log 2 (n) + 1) = O (lg2 (n))

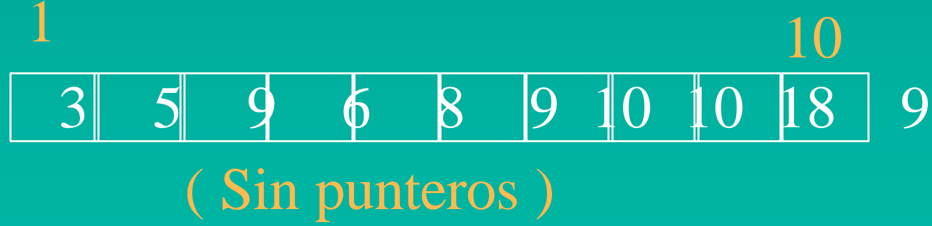
Heap (montón, pila)

- Implementación en arreglo del árbol binario desbalanceado con ordenamiento parcial.





Indices : de izq. a der.



- Padre de a[i] es a[i div 2]   i>1

    - Si existe hijo izq. de a[i] es a[2i]

    - Si existe hijo derecho de a[i] es a[2i +1]

- Para ver si existe un hijo:

  Sea i el nodo actual, last el último

    - Si j <= last; j = 2*i es el hijo izq. de i

    - Si j < last hay hijo derecho de i

  (last div 2) es el padre del último

  (last div 4) es el abuelo del último.

- En selección del mínimo:

  - Se toma el valor de la raíz: a[1]

  - a[last] reemplaza a la raiz last--

  - Se reordena. (descendiendo)

- En inserción :

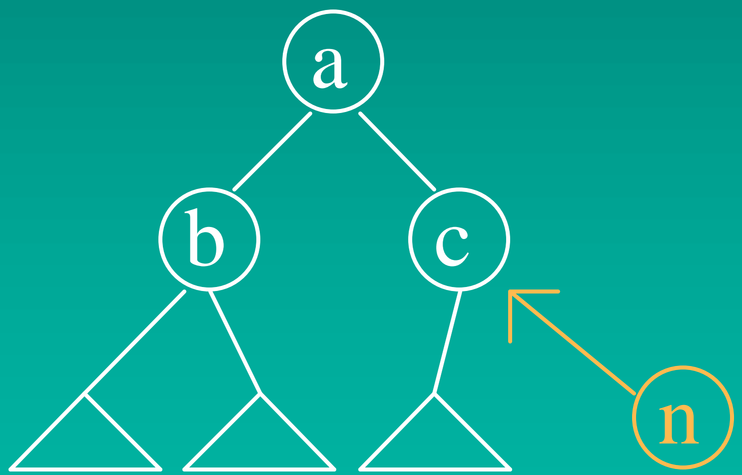
  - Se inserta en a[last + 1]

  - last + +

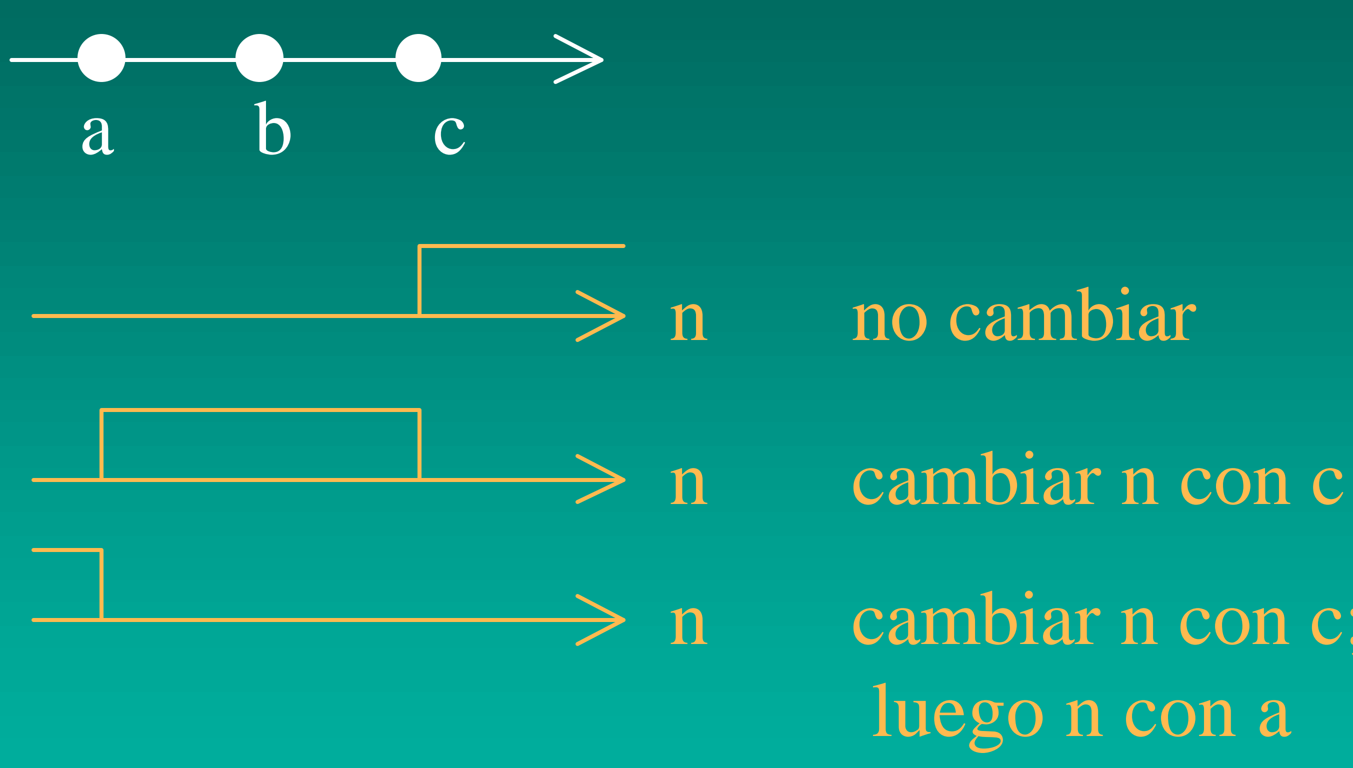
  - Se reordena (ascendiendo)

Análisis inserción en heap. (ascenso)

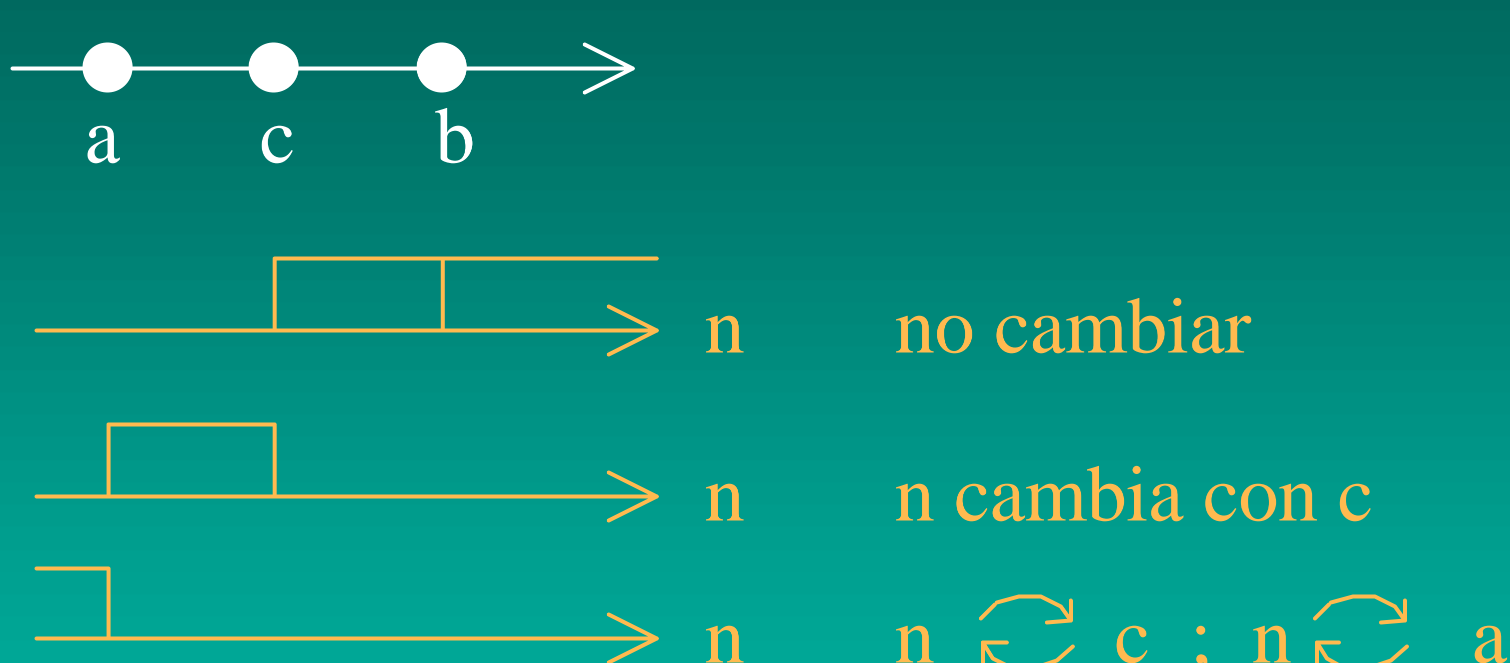
- Sea un heap que tiene elementos a, b y c.
- Se desea introducir n :



- Caso i)



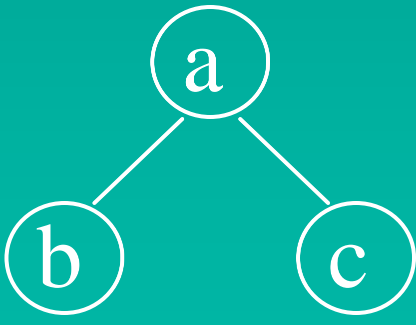
- Caso ii)

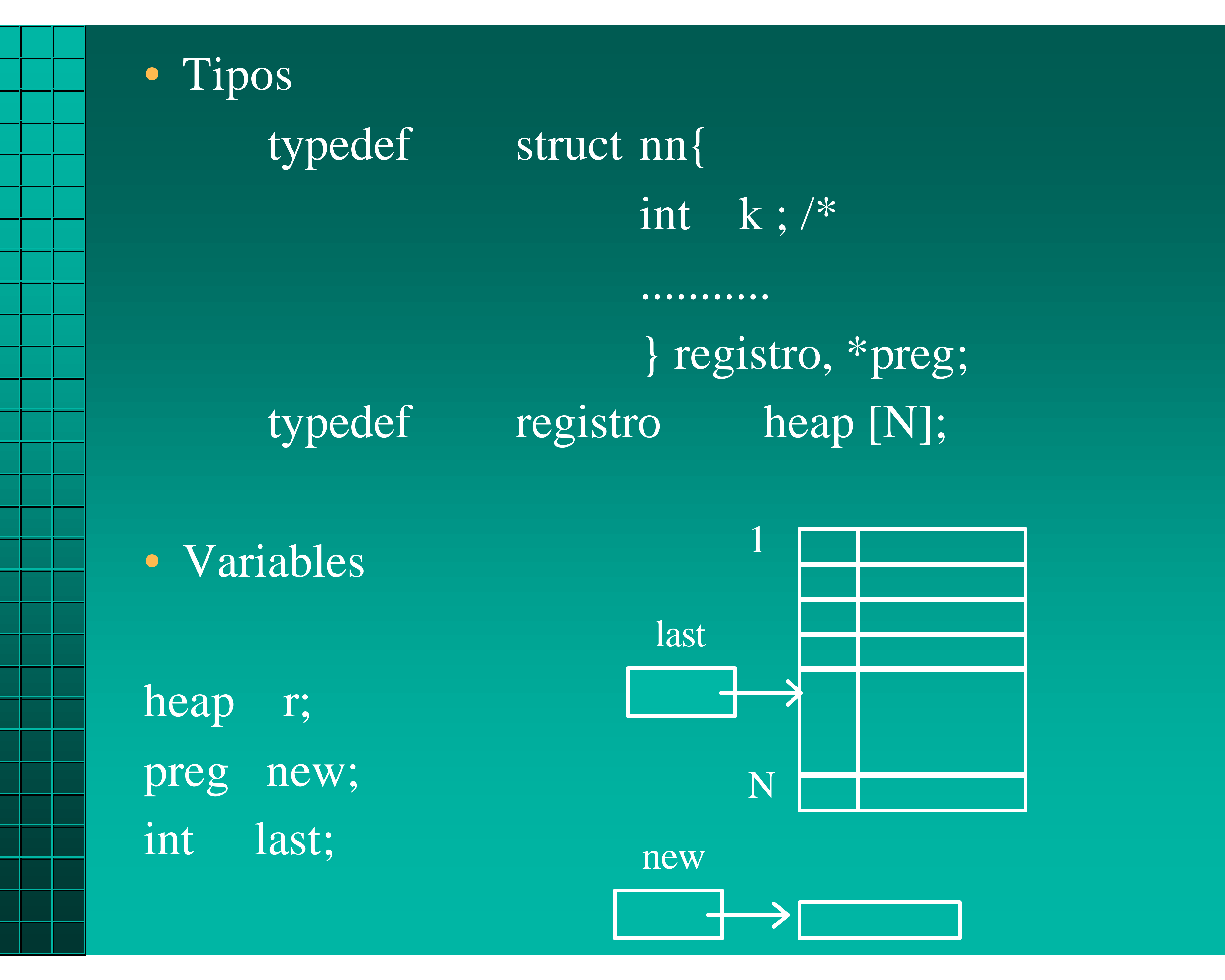


Análisis descenso en heap

- Caso i) a <= b y a <= c   no cambia
- Caso ii) a > b o a > c   empujar a

```
if (c> = b)
        if (a>b || a > c )  swap (a,b);
else
        if (a > b || a > c ) swap (a,c);
```





Tipos
typedef struct nn{
int k ; /*
....
} registro, *preg;
typedef registro heap [N];
Variables
heap r;
preg new;
int last;
1
last
N
new

Inserción en heap (Williams) Comm ACM 7 Nº6 <u>in situ</u> Alg 232. 1964

```
insert ( new, r)
preg new;
heap  r;
{
        int   i, j;
        extern  int  last;
        last + +;
        for ( j = last; j> 1; j = i ) {
                i = j / 2;              /* i es cursor del padre */
                if ( r[i] . k < new -> k ) break;
                        copyreg (j, i )             /* r[j] = r[i] */
                }
                copy r (j, new );                   /* r[j] = *new */
  }
```

Descenso en heap (Floyd) Comm ACM Nº 8 Slg in situ 113, 243 1962

```
siftup (r, i, n )
heap r ;
int i, n;
{
    int j
    registro temp;
    while (( j = 2 * i) < = n ) {
            if (j < n && r [j] . k > r [j+1]. k ) j + +;  /*hay hijo der. */
            if ( r[i].k > r[j].k) {
            swap ( r, i, j );          /* temp = r[j], r[j] = r [i], r [i] = temp */
            i = j;                            /* nueva raíz en el descenso */
            }
            else break
    }
}
```

```
delete (r)
heap r;
{
   extern int last;
   if ( last > 1)   error ("Intento extracción en heap vacío");
   else{                                              ... /* salvar raíz */
          copyreg (1, last );                         /* r [1] = r [last] */
          siftup ( r, 1, -- last );
}
```

Ordenar (según N.W.) cap 2

- Tipos

  - Interno : de arreglos (de registros)

  - Externos : de archivos (de registros)

- Se ordena según la clave del registro

- Medidas de eficiencia:

  - C(n) número de comparaciones de claves

  - M(n) número de movimientos de datos

- Métodos directos $O(n^2)$

  - Fácil de entender
  - Son la base de métodos más poderosos

  $O(n \lg_2 n)$

  - Funcionan bien para n pequeños

- In situ

  Se ordena en el mismo arreglo. (no requieren espacio adicional).

- Clases:

1.- Inserción

Ordenar una mano de naipes.

- Inserción directa N.W. 2.2.1 Gonnet 4.12

AHU 85

- Inserción binaria N.W. 2.2.1

En la etapa i ésima se inserta [i] en su lugar correcto entre a[1], a[2], ... a[i-1] que están previamente en orden.

2.- Selección

Directa (algoritmo simple) N.W. 2.2.2

```
for i:= 1  to n - 1 do
        begin
        Encuentre el menor entre a[i]  ....  a[n]
        Intercámbielo con a[i]
end
```

3.- Intercambio

Directo.

Se comparan e intercambian pares adyacentes de items, hasta que todos los items esténordenados

Bubble sort   shakesort   N.W. 2.2.3

- Existen demasiados métodos. Se estudian, con cierto detalle, dos de los más eficientes.
- **Heapsort** O (n lg(n)) en peor caso. (refinamiento de selección)
- **Quicksort** O (n lg(n)) en promedio. $O(n^2)$ en peor caso

- De selección directa a heapsort.

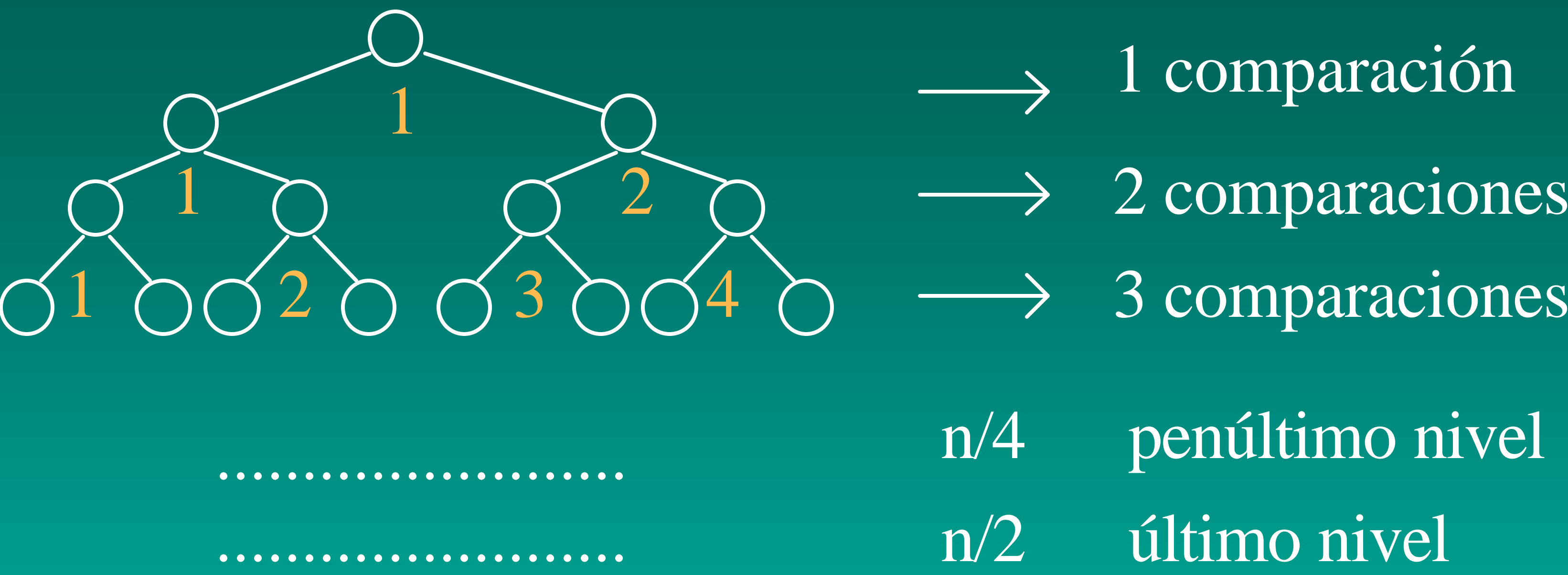
  - Encontrar menor clave entre n items implica n - 1 comparaciones
  - Encontrar menor clave entre (n - 1) items implica n - 2 comparaciones

- En total:

$$(n-1)+(n-2)+\ldots+1=\sum_{i=1}^{n} i - n =$$

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

- Si se forma un árbol de selección : (N.W. 2.2.5)



- Para formar el árbol, se requieren:

$$\frac{n}{2} + \frac{n}{4} + ... + 4 + 2 + 1 = 2^0 + 2^1 + 2^2 + ... + \frac{2^i}{2} = n - 1$$

- Si se elimina la raíz, y se vuelve a comparar el resto, se requieren lg(n) comparaciones en cada caso.

- En total:

$$n + n\lg(n)$$

↑

para formar el árbol

(Si se divide en $\sqrt{n}$ grupos de $\sqrt{n}$ items cada uno, resulta O (n $\sqrt{n}$). (Knuth pag 141 5.2.3)

Heapsort

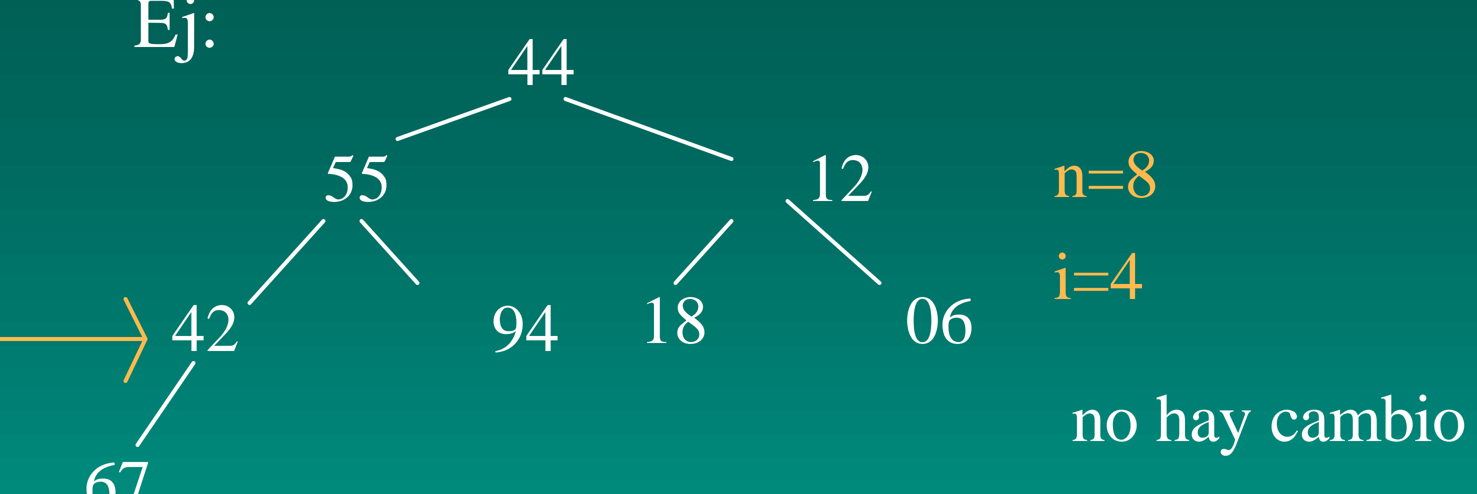
- Se construye cola de prioridad, repetidamente se extrae la raíz, hasta que la cola quede vacía.

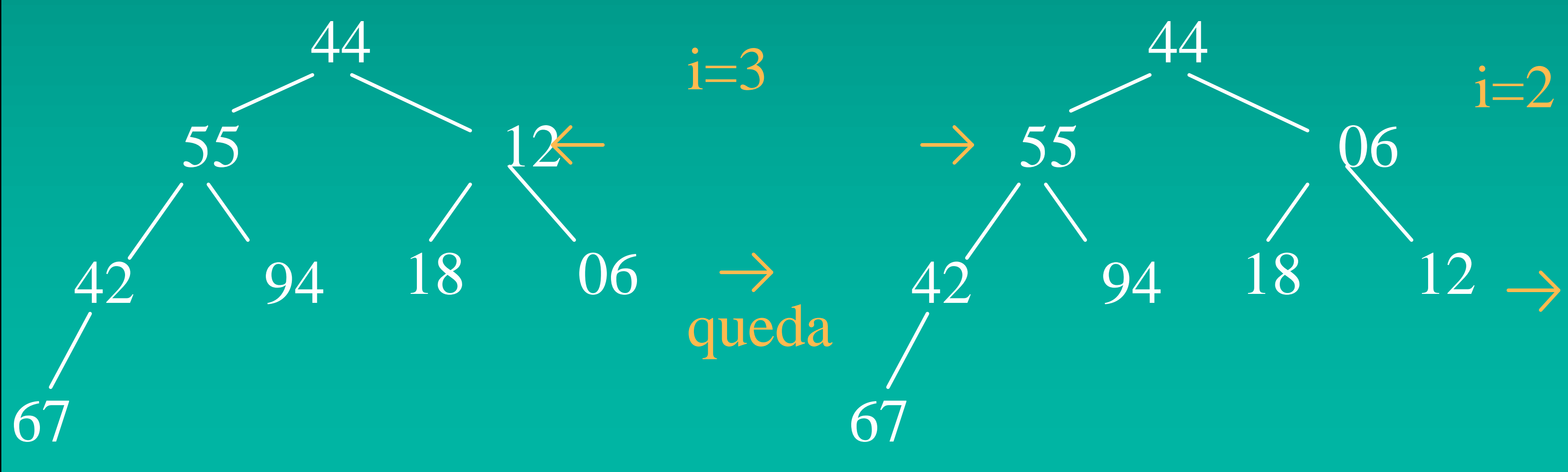
- Formación del heap, a partir arreglo desordenado

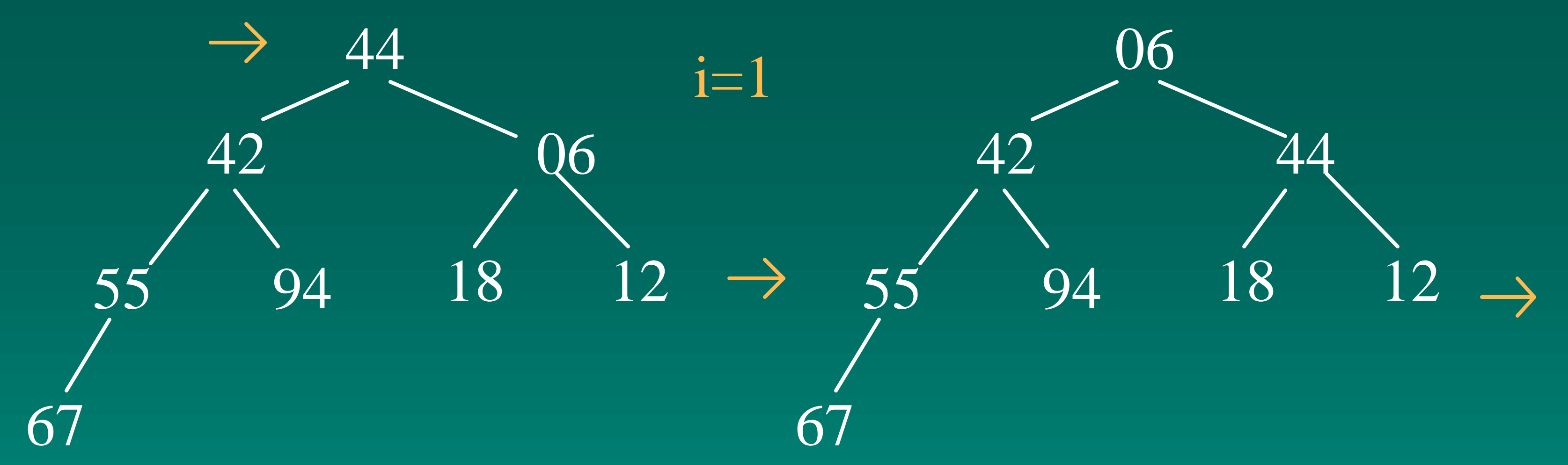
```
for ( i = n/2; i> 0; i - - ) siftup ( r, i, n );
```

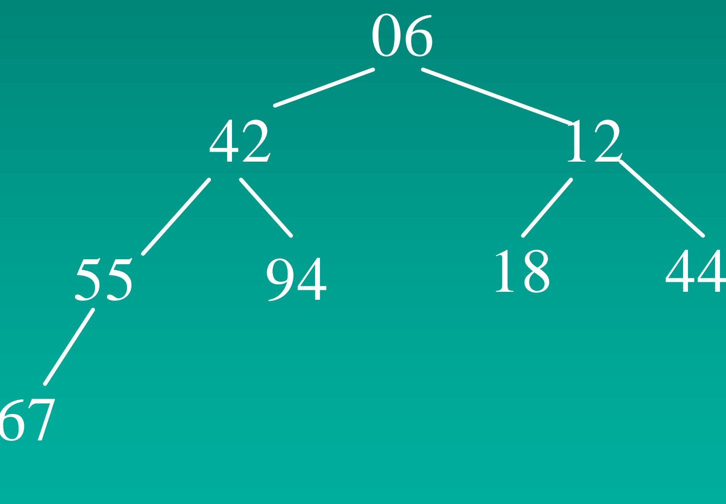
- Formación del heap in situ :

Ej:









- Complejidad

$$\frac{n}{2} * \lg(n) \Rightarrow O(n\lg(n))$$

Es heap

- heapsort:

```
for (i=n/2;i>0;i--)  siftup(r,n,i); /*forma heap*/
for (i=n;i>1;i++) { /* ordena */
            swap(r,1,i);
            siftup(r,1,i-1)
      }
```

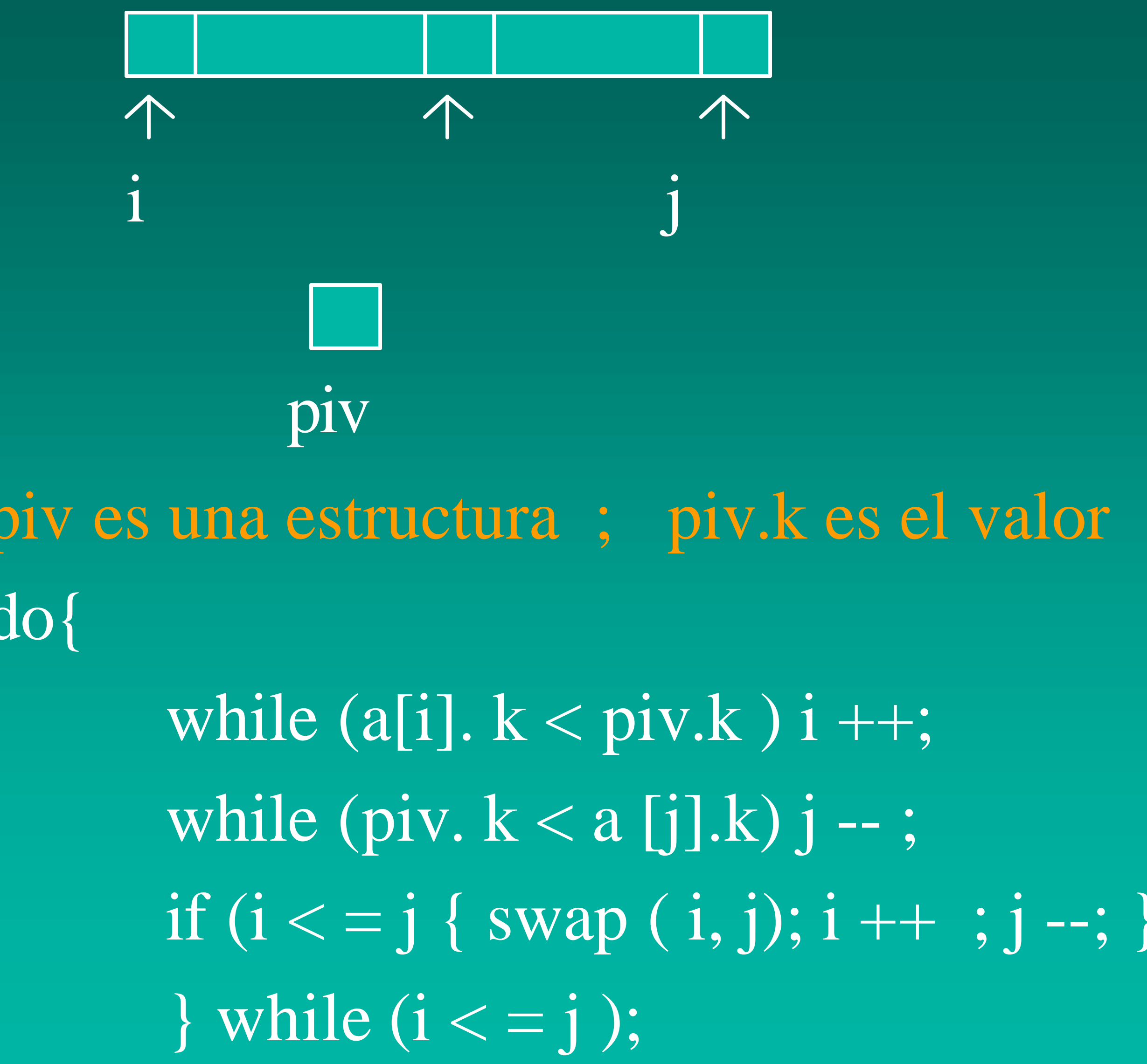
Quicksort Hoare 1962
En promedio nlg(n)

- Se parte el arreglo a ordenar en dos subarreglos. Si los subarreglos no están ordenados, se los vuelve a partir, hasta lograr ordenamiento
- Es un algoritmo del tipo "divide para vencer".
- Es un tipo de sort por intercambio, En éstos se compara e intercambia items hasta ordenar.

- El método bubblesort (burbuja) es del tipo orden por intercambio y requiere n^2 comparaciones entre items adyacentes

- En quicksort los intercambios se efectúan sobre distancias mayores y sobre un pivote

- Ordenamiento en partición



- Ejemplo :

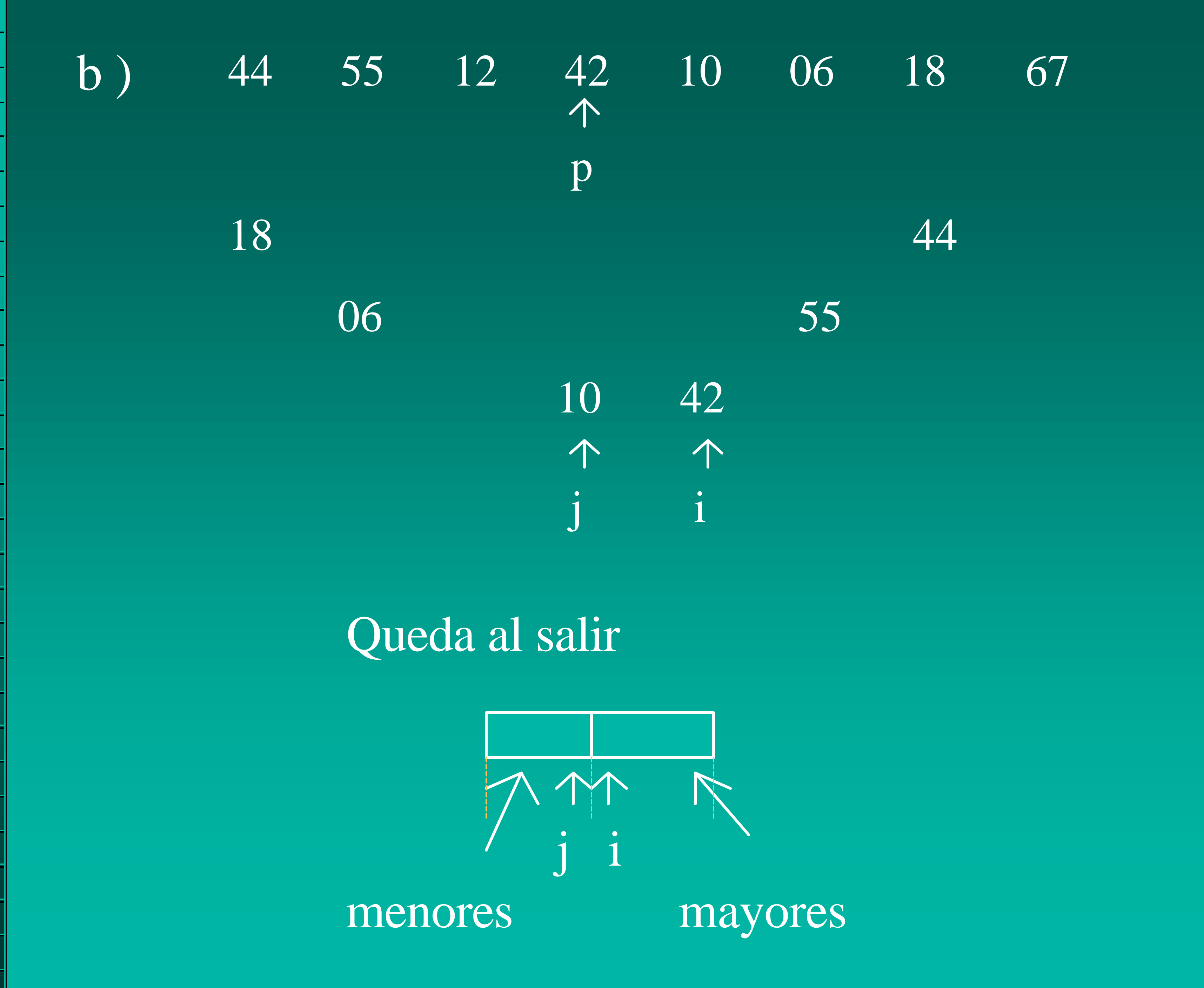
a)

44	55	12	42	94	06	18	67
			↑				
			p				
18						44	
	06				55		
			42				
		↑		↑			
		j		i			

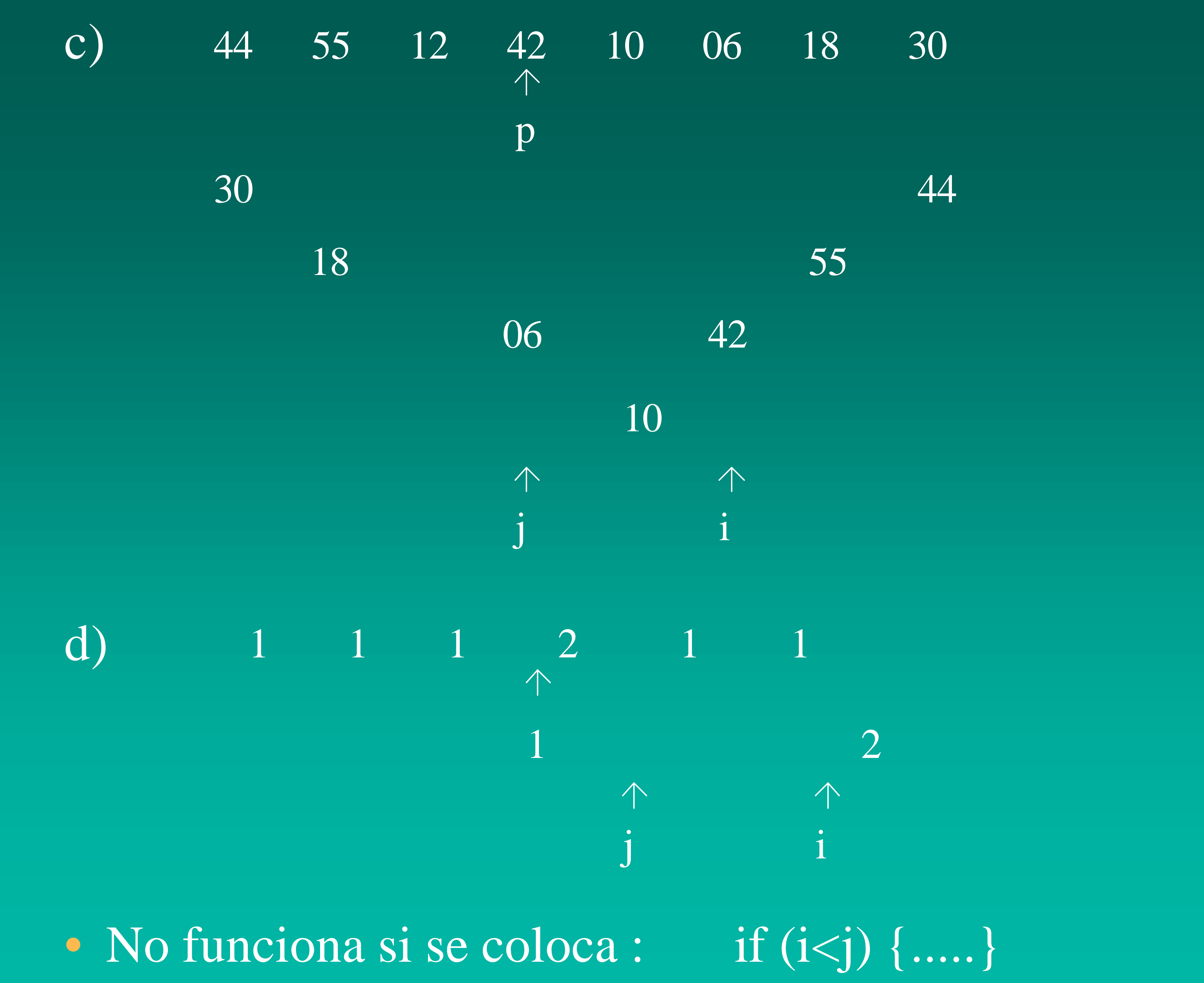
- Al salir (i > j):

  - Valores mayores que el pivote, no están ordenados

  - Valores menores que el pivote, no están ordenados



b)
44 55 12 42 10 06 18 67
p
18 44
06 55
10 42
j i
Queda al salir
j i
menores
mayores


c) 44 55 12 42 10 06 18 30
p
30 44
18 55
06 42
10
j i
d) 1 1 1 2 1 1
1 2
j i
No funciona si se coloca : if (i<j) {.....}

- Al salir (i > j):

  - Valores mayores que el pivote, quedan a la derecha del pivote.

  - Valores menores quedan a la izquierda.

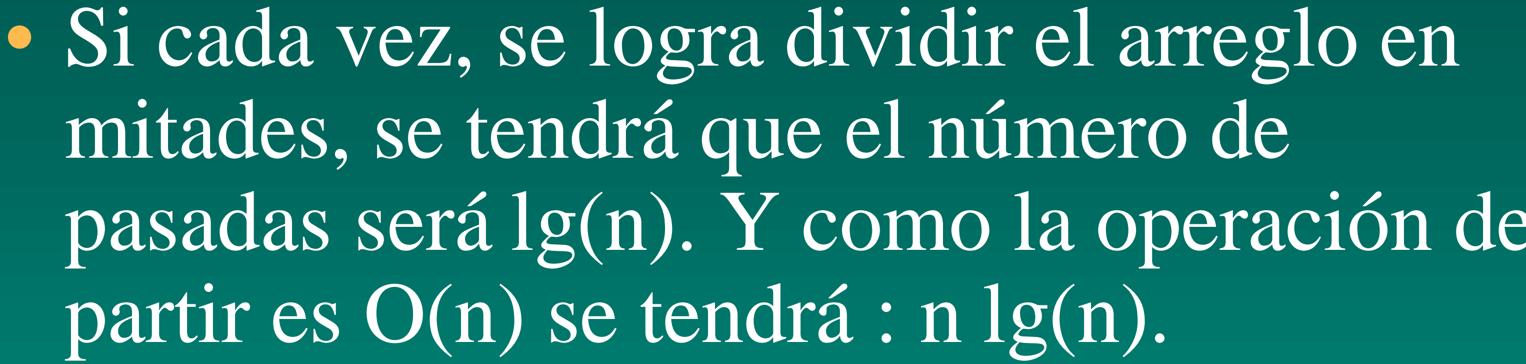
- Otra versión: Gonnet

```
temp : = a [i];
while (i < j )
 {
   while ( piv . k < a [j].k ) j - - ; a[i] = a[j];
   while ( i < j && (a[i] k < = piv. k ) i + +; a[j] = a [i];
 }; a[i] = piv
```

```
void   qsort (l,r)   N.W.    /* qsort, (1, n)  ordena el */
int, l,r                                /* arreglo */
{
   int, i, j;
   .. piv;
   i = l; j = r; piv = a[(l + r ) / 2 ];
         do {
         .....
         } while ( i < = j  );
   if ( l < j ) qsort ( l, j );
   if ( i < r ) qsort (i, r);
}
```

Gonnet

```
void   qsort (l,r)
  int l,r;
  {     int i, j;
        ...  piv;
        while (r > k ) {
        i = l; j = r; piv = a [ l]
        while ( i < j ) {
                ....
                }
        a[i] = piv;
        qsort ( l, i - - );
        l = i + +;
  }
```



- Si cada vez, se logra dividir el arreglo en mitades, se tendrá que el número de pasadas será lg(n). Y como la operación de partir es O(n) se tendrá : n lg(n).

- Obs :El peor caso es n^2

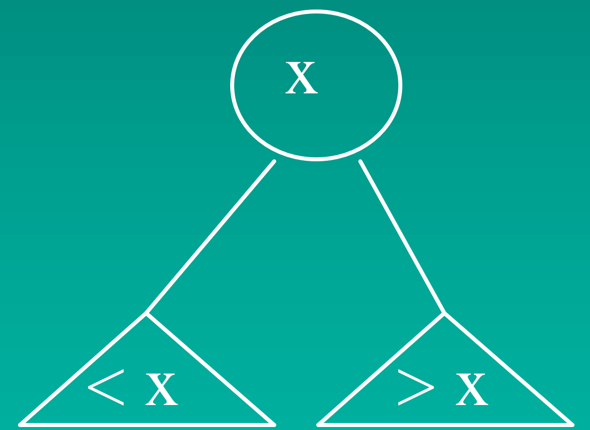
- Si siempre el valor mayor se toma como pivote, se tendrán n pasadas de n cada una.

  Es decir : $O(n^2)$

Arbol de Búsqueda Binario
AHU Cap 5

- ADT

  Cuando hay gran cantidad de insercionesy descartes; y cuando n es grande (conjuntos grandes).



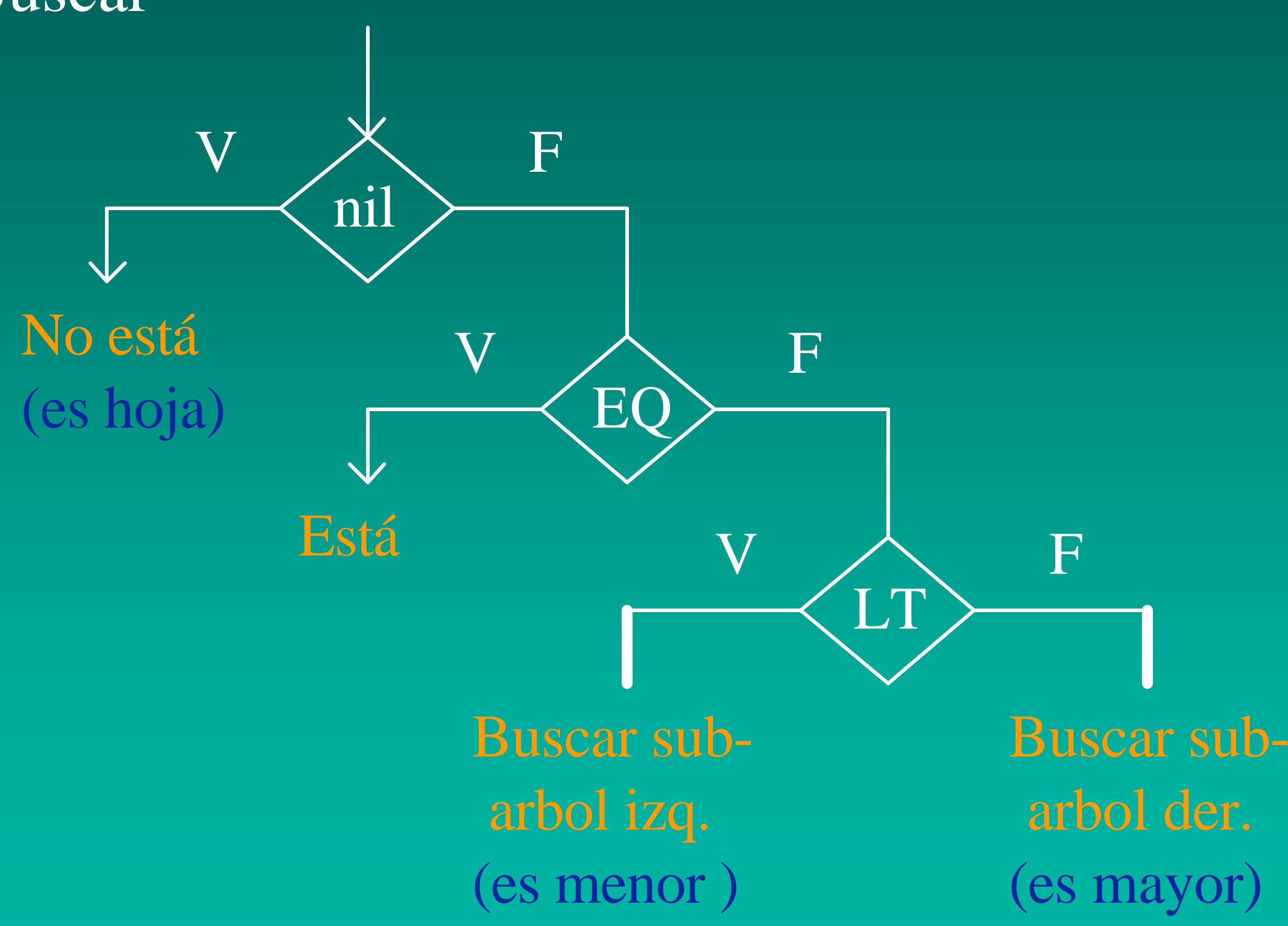
- Insertar, descartar, buscar, mínimo: O(lg n) en promedio

- Propiedad:
  - Miembros subárbol izq. menores que la raíz
  - Miembros subárbol der. mayores que la raíz
- Obs: Un listado in orden, implica recorrido ordenado. (sort)
- Debe disponerse de operadores de comparación:

  Por ej. L T (a, b) ; E Q (a, b)

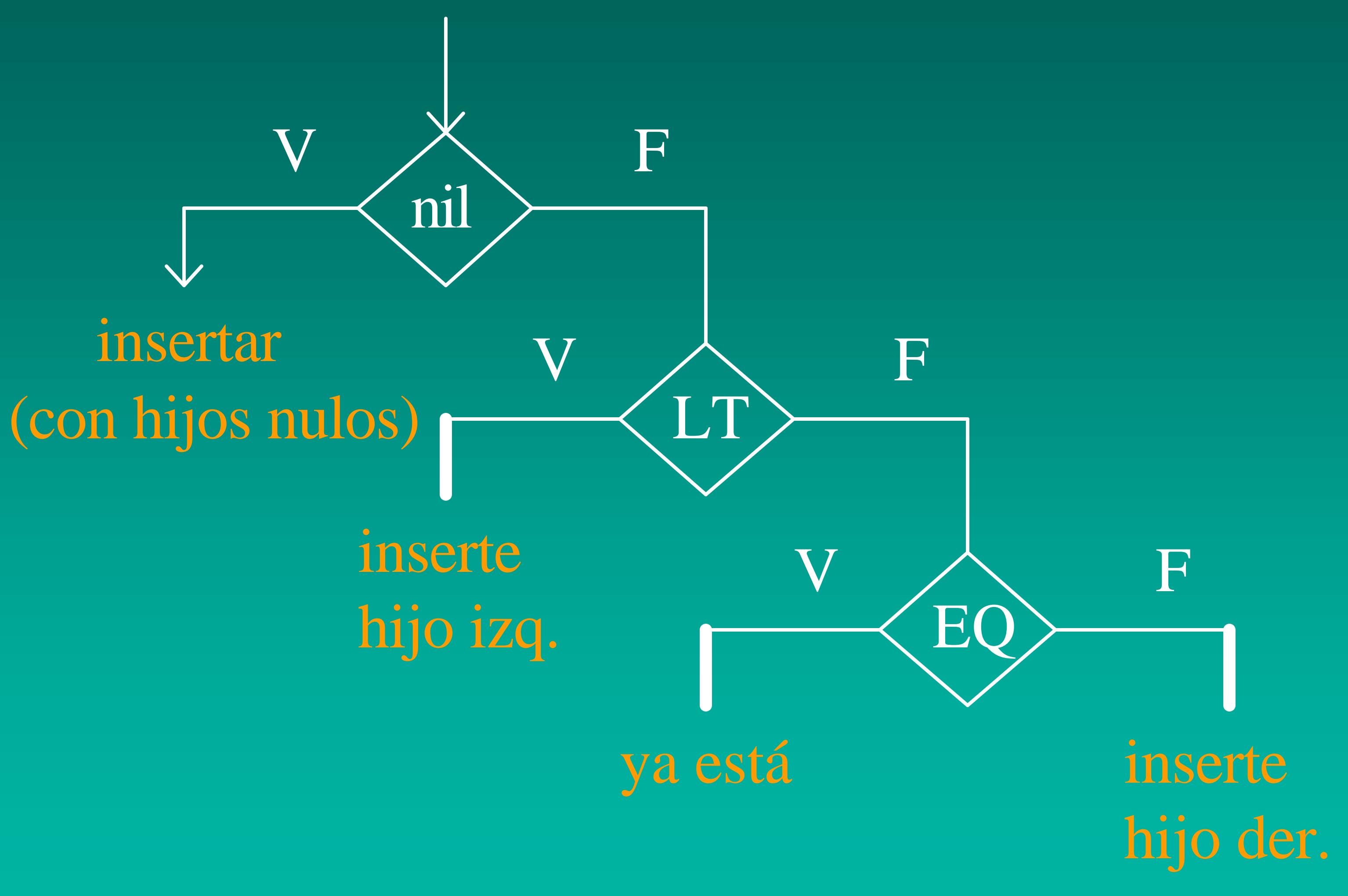
Operaciones (Recursivas)

- Buscar



- Insertar

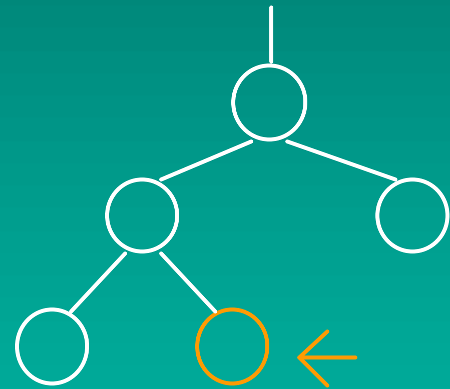
Se busca, si no está se inserta (N.W. pag 204)



Descartar (es más compleja)

- El nodo que se desea eliminar, puede ser

1.- Hoja : El descarte es trivial



colocar puntero nulo

2.- Nodo interno

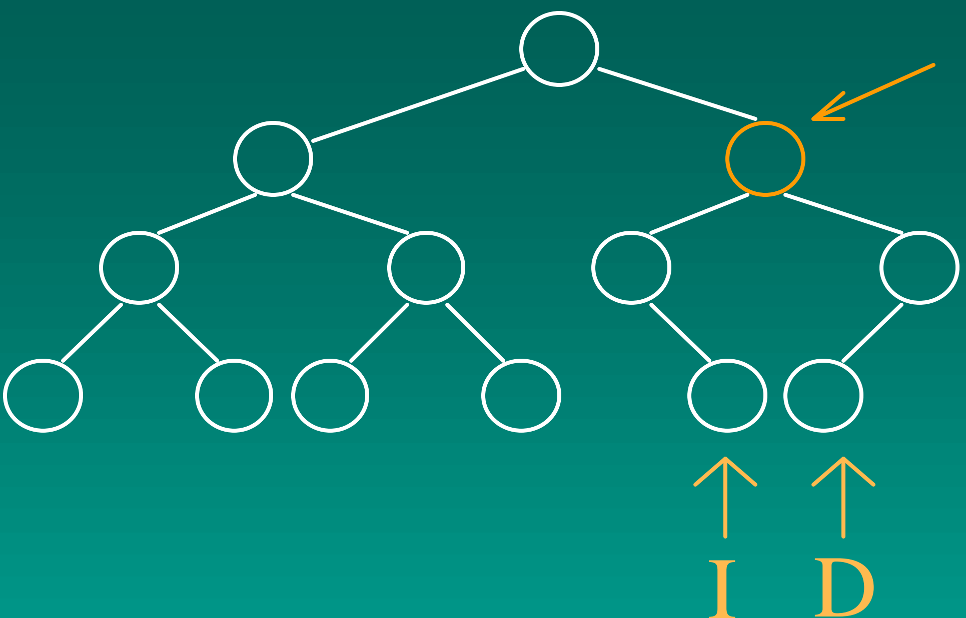
2.1 Con un hijo

a.- Izq.

b.Derecho

El padre debe apuntar al nieto

2.2 Con dos hijos



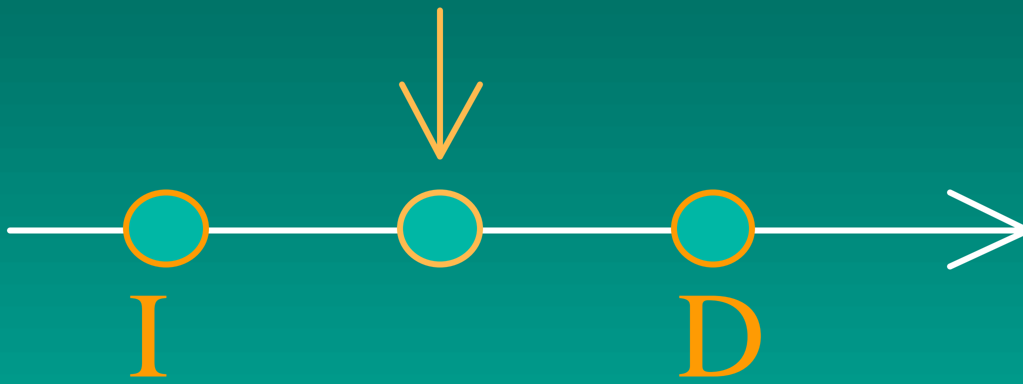
- Dos soluciones:

  - Buscar menor nodo descendiente de hijo derecho D

  - Buscar mayor descendiente hijo izq. I

Reemplazar hoja obtenida por el que se descarta

- Dibujando Valores en eje :



(No hay otros nodos entre medio)

```
typedef  struct  tnode {
            typekey k;
            struct tnodo *left, * right;
            } nodo, * pnodo;
```

```
pnodo buscar (x, t )
   typekey x;
   pnodo t;
   {
          if ( t = = NULL_T) return (NULL_T );  /* no está */
          else {
                   if ( t -> k = = x ) return (t);          /* lo encontró */
                            else {
                            if ( t -> k > x ) t = buscar ( x, t -> left);
                            else      t = buscar ( x. t -> right);
                            }
          }
          return ( t );               /* ! */
   }
```

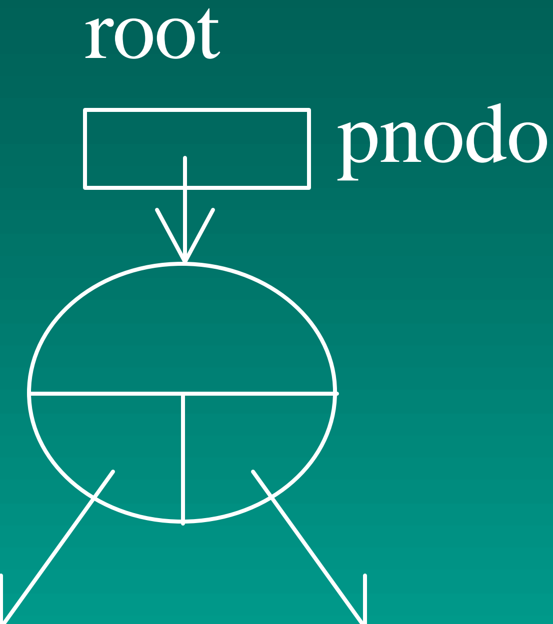
```
pnodo  buscar (x, t)                    /* no recursivo */
   typekey x;
   pnodo t;
   {
         while ( t ! = NULL_T )
               if ( t -> k = = x )   return (t);
               else{if (t-> k < x)  t = t -> right;
                     else t = t -> left
                     }
         return (t);               /* NULL_T */
   }
```

Diseño recursivo

Simulando procedimiento pascal

```
static int  found:
void     buscar (x, pp)
         typekey x;
         pnodo  * pp
         {
         extern int found;
         if ( *pp  =  = NULL_T)  found = 0:
         else if ( *pp -> k = = x )    found =  1;
           else if (( *pp -> k > x )  buscar (x, &((*pp) -> left))
           else    buscar ( x, & (( *pp -> right ));
         }
```

- Si el árbol comienza en root;



- Se procede a buscar según:

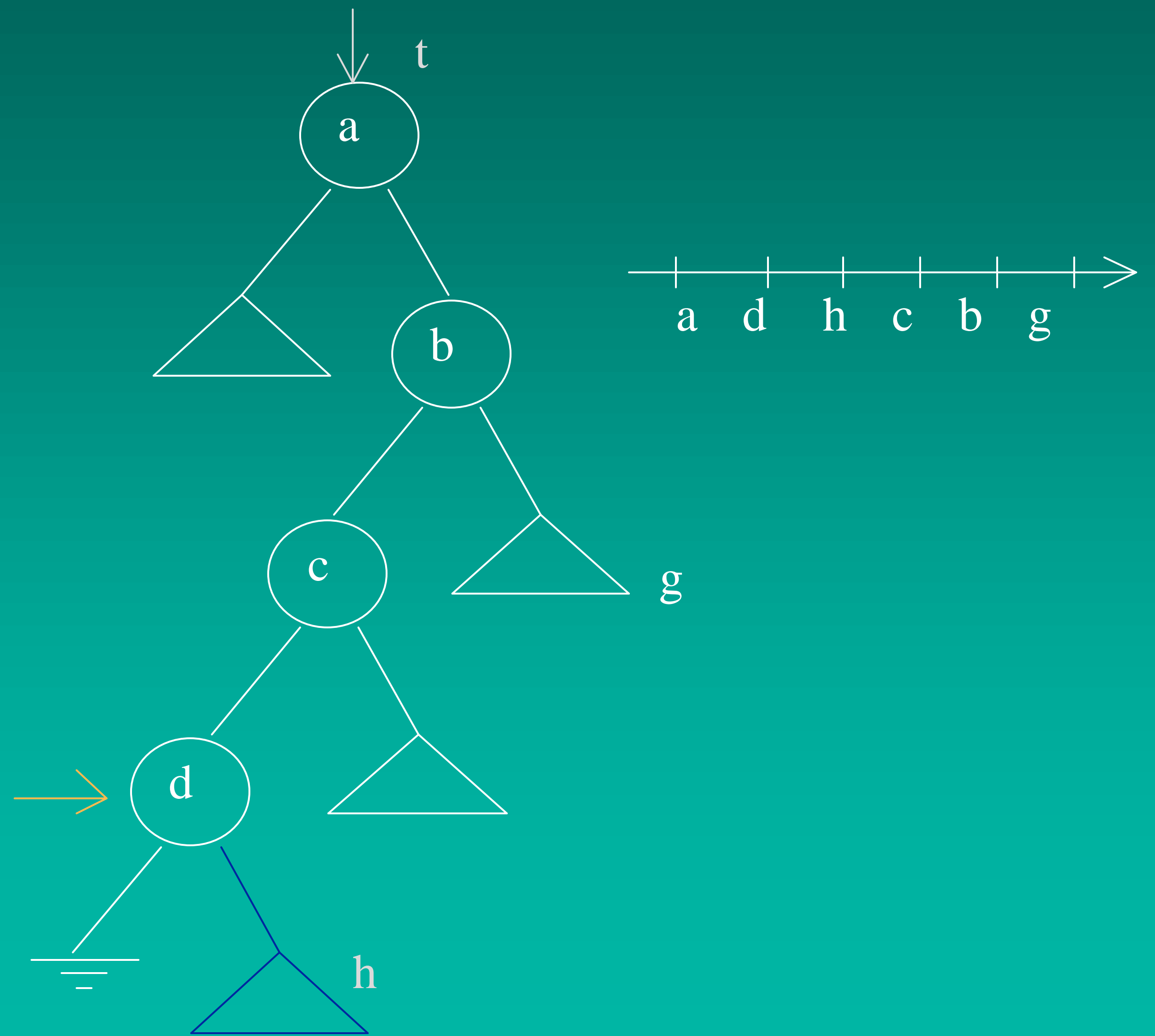
  buscar (x, & root);

  If (found)…….

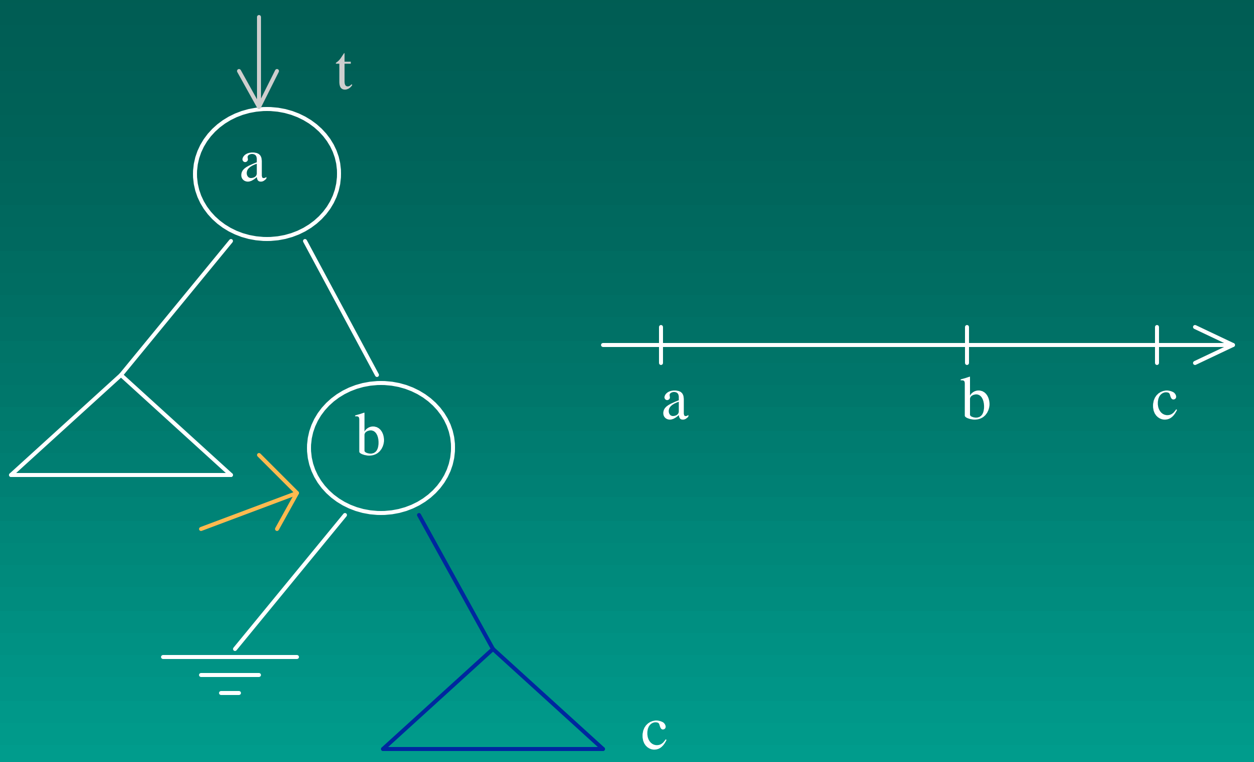
  ……

- Buscar menor descendiente del hijo derecho D
- Dos casos

a.-



b.-



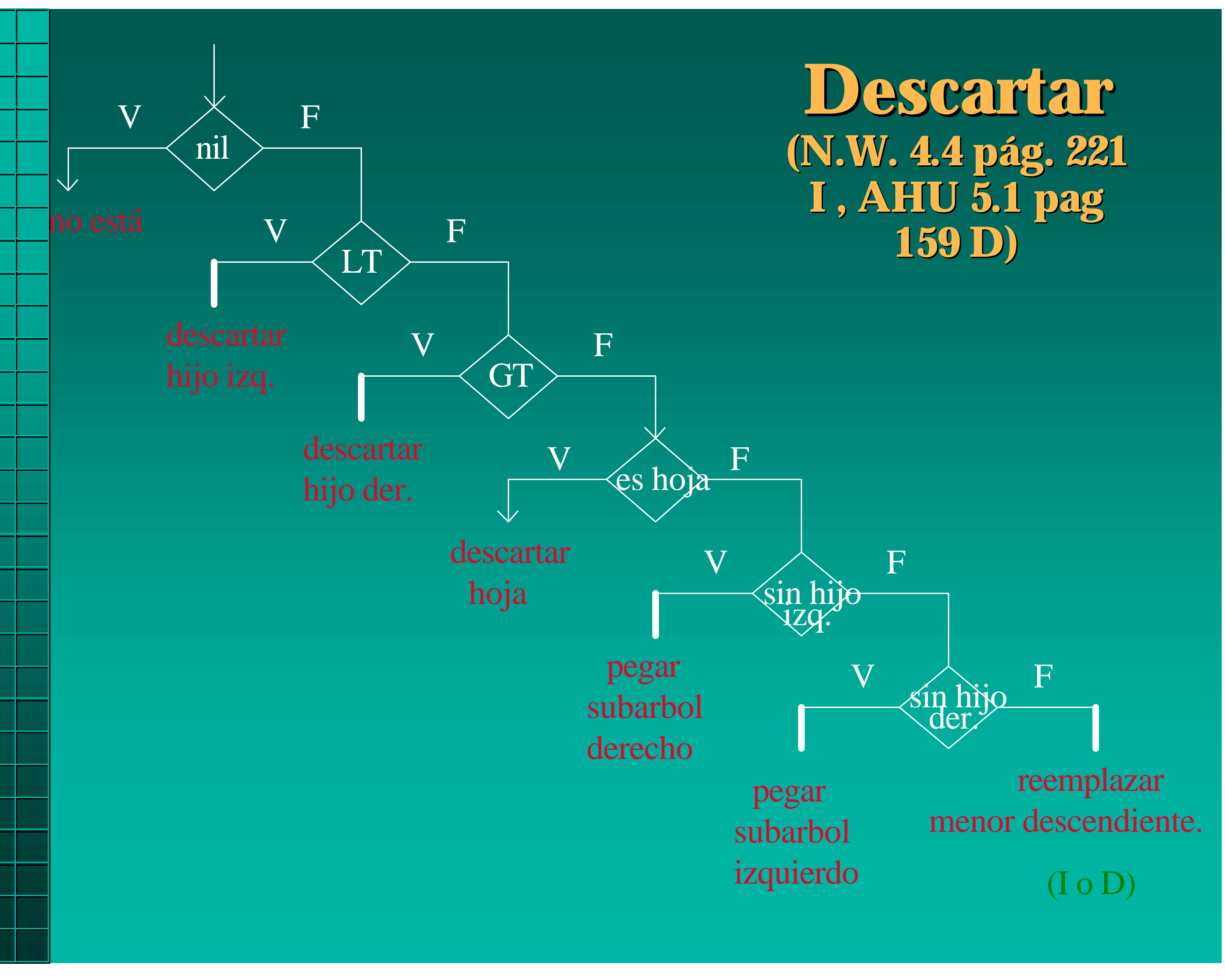
mientras (se tenga hijo izq.)

descender por la izq.;

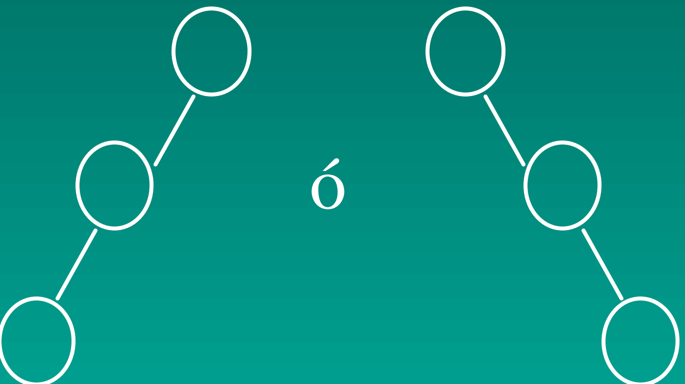
marcar menor descendiente;

pegar hijo derecho del menor descendiente;

```
pnodo  menordhd (t)/*Conviene proceder iterativamente */
   pnodo t;
   {      pnodo p;
          p = t -> right                              /* caso b */
          if p -> left = = NULL_T ){
                        t -> rigth = p -> right;
                        return (p);
                        }
          while ( p -> left ! = NULL_T) {       /* caso a */
                   t = p;
                   p = p -> left;
                   }
   t -> left = p -> right;
   return (p);
   }
```


Descartar
(N.W. 4.4 pág. 221 I , AHU 5.1 pag 159 D)
V
nil
F
no está
V
LT
F
descartar hijo izq.
V
GT
F
descartar hijo der.
V
es hoja
F
descartar hoja
V
sin hijo izq.
F
pegar subarbol derecho
V
sin hijo der.
F
pegar subarbol izquierdo
reemplazar menor descendiente.
(I o D)

- Si no se restringe desbalance, en peor caso se tiene O(n)
- árboles degenerados (son listas): requieren n/2 operaciones en promedio



$$\frac{\text{altura árbol desbalanceado promedio}}{\text{altura perfectamente balanceado}} = 1,\ 386$$

(N. W pag 214)

- El 39 % es bastante bajo; en la mayoría de los casos puede aceptarse el incremento de la complejidad, sin complicar el algoritmo, tratando de mantener balanceado el árbol.

- Existen esquemas de reorganización del arbol, para mantener más o menos balanceado el árbol, sin incurrir en el costo del perfectamente balanceado

N.W. AVL Sección 4.4.6 pag. 215

AHU 2-3 Sección 5.4 pag. 169

Arboles AVL

Adelson- Velskii - Landis

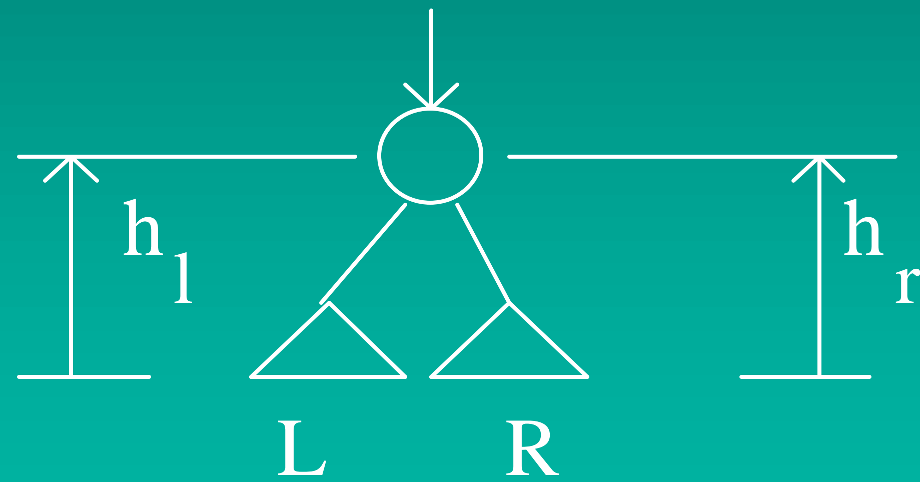
- La reorganización del árbol, después de inserciones no tiene que llevar al perfecto balance, pues de hacerlo la operación sería complicada

- AVL demuestran que si y solamente si para cada nodo, los altos de los subárboles difieren a lo más en uno :

$$h_{pb} = \lg(n+1) \leq h_{AVL} \leq 1,4404 \lg(n+2) - 0,328$$

- Es decir, en peor caso, un 45 % de alargue de trayectoria, respecto al perfectamente balanceado

- Factor de balance = hr - hl para cada nodo.
- En A.V.L., el factor de balance puede ser: - 1, 0, +1



Operación Inserción

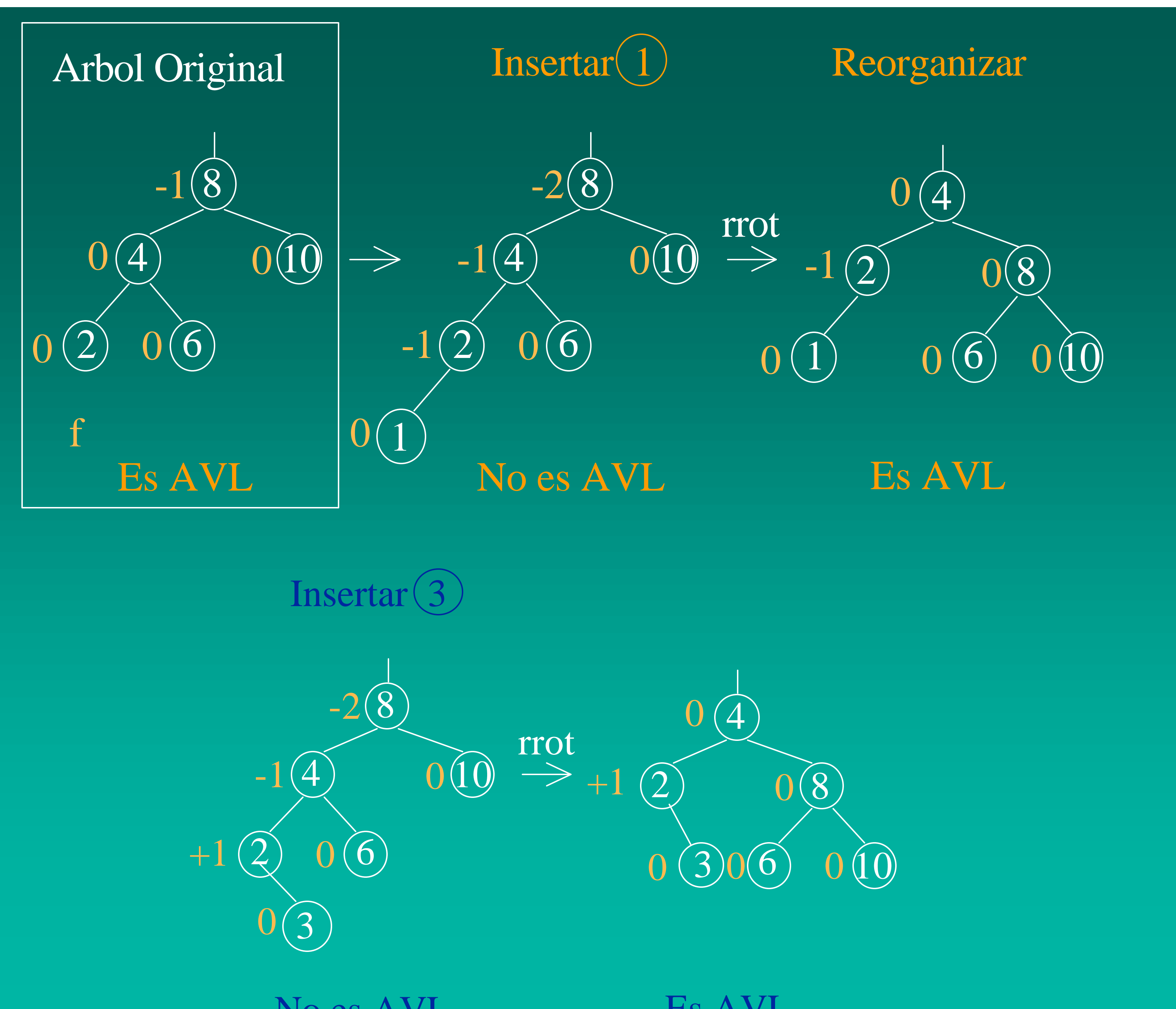
- En subárbol izq.

  - si antes hl = hr, después queda AVL

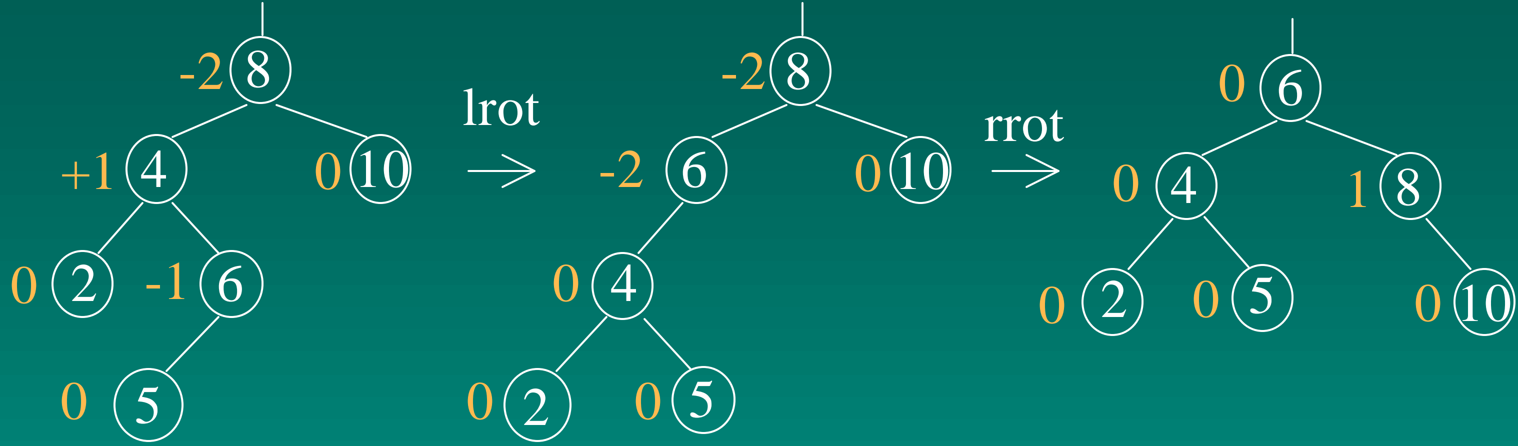
  - si antes hl < hr, después mejora balance

  - si antes hl > hr, no queda AVL, y debe reoganizarse el árbol

- Ejemplo : Se analizan 4 casos de inserción en subárbol izq del árbol original


Arbol Original
-1 8
0 4
0 10
0 2
0 6
f
Es AVL
Insertar 1
-2 8
-1 4
0 10
-1 2
0 6
0 1
No es AVL
rrot
Reorganizar
0 4
-1 2
0 8
0 1
0 6
0 10
Es AVL
Insertar 3
-2 8
-1 4
0 10
+1 2
0 6
0 3
No es AVL
rrot
0 4
+1 2
0 8
0 3
0 6
0 10
Es AVL

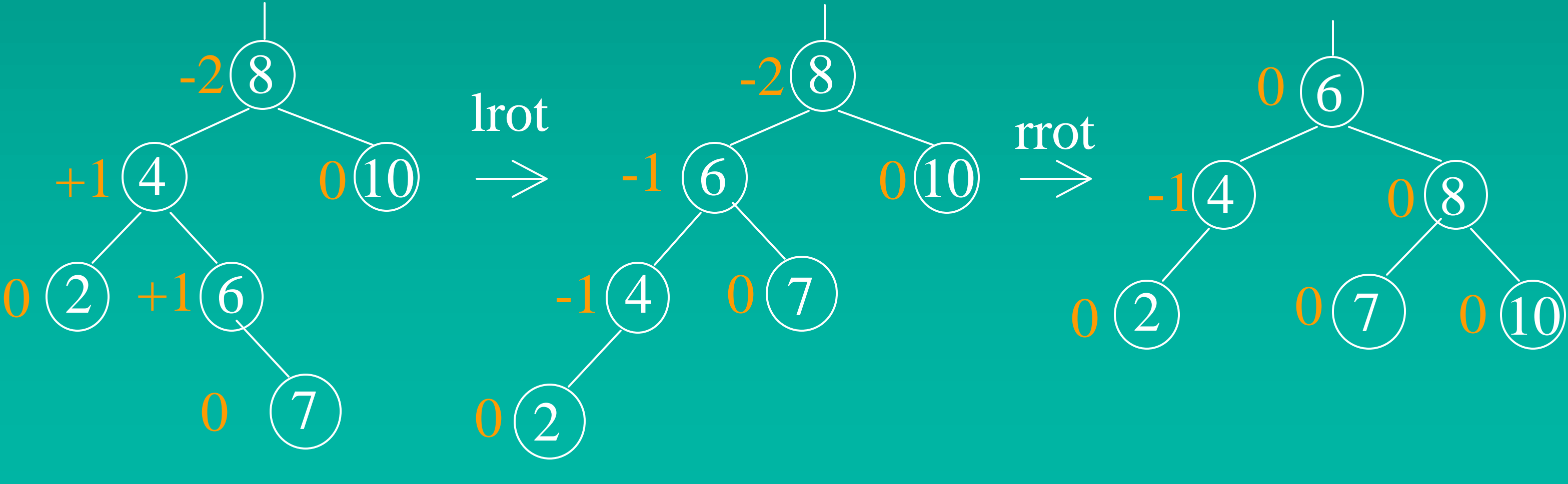
Insertar 5



No es AVL

No es AVL

Es AVL



No es AVL

No es AVL

Es AVL

Rotaciones simples

a) lrot a<A<b<B<c

```
pnodo        lrot (t)
pnodo t;
{       p nodo temp;
        temp = t;
        t = t -> right;
        temp -> right = t -> left;
        t -> left = temp;
/*      Factor correción  AVL :     */
        t -> left -> bal = 1- t -> bal;
        t -> bal=0;
return (t)
}
```

p
A
a
B
b
c
p = lrot(p)
p
B
A
c
a
b

b.- rrot

```
pnodo  rrot (t);
    pnodo t;
    {
    pnodo temp;
    temp = t ;
    t = t -> left
    temp -> left = t -> right
    t  -> right = temp
/*          Factor correción  AVL :      */
    t -> right-> bal = -1- t -> bal;
    t -> bal=0;
    return (t)
    }
```

Rotaciones dobles

a)

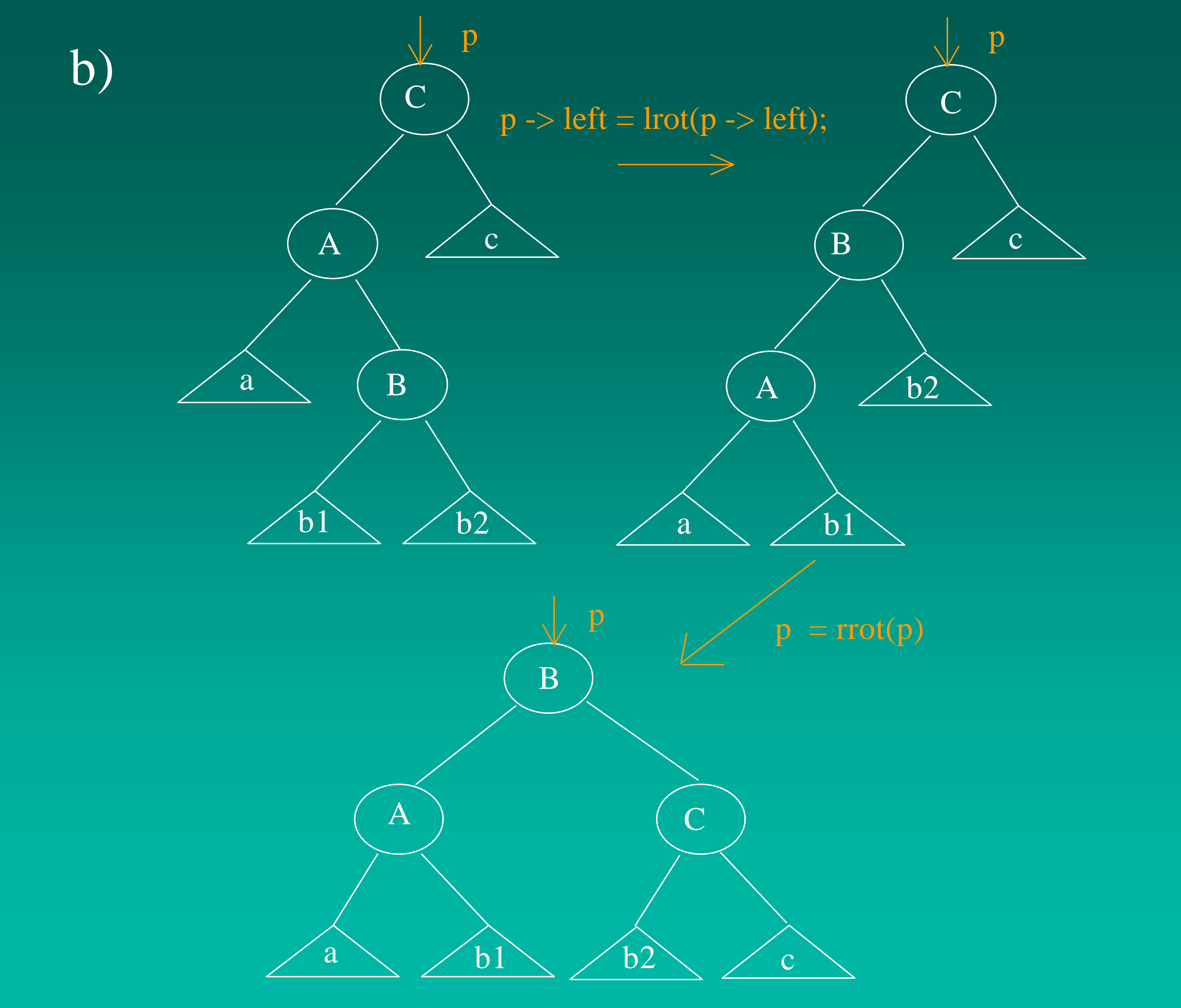
p -> right = rrot(p -> right);

p = lrot(p)

A, a, C, B, c, b1, b2

p, A, a, B, b1, C, b2, c

p, B, A, C, a, b1, b2, c



b)
p
C
A
c
a
B
b1
b2
p -> left = lrot(p -> left);
p
C
B
c
A
b2
a
b1
p
p = rrot(p)
B
A
C
a
b1
b2
c

- Factor correción en general:

- lrot :

newbal(A) = oldbal(A) -1 - max (oldbal(B),0);

newbal(B) = min(oldbal(A) - 2 , oldbal(A) + old(BAL) - 2,
oldbal(B) - 1)

- rrot:

newbal(A) = oldbal(A) +1 - min (oldbal(B),0);

newbal(B) = max(oldbal(A) + 2 , oldbal(A) + old(BAL) + 2,
oldbal(B) + 1)

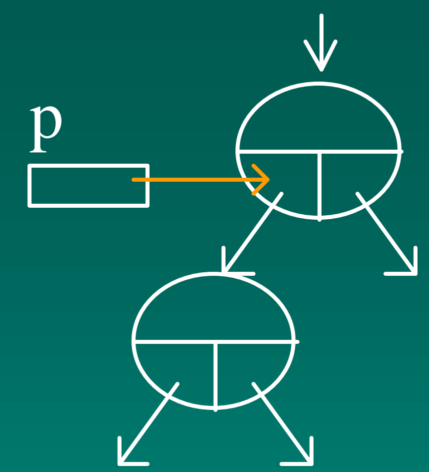
```
pnode rrot (r)              /* rutina hace rotación a der */
pnode t;
{  pnode  temp = t;
   int marca, aux;
   t = t - > left;
   temp -> left = t -> right;
   t -> right = temp;
   marca = temp -> bal;
   temp -> bal += 1 - (( t -> bal >0) ? t -> bal:0); /* corrige  factor
   de bal. de temp */
   aux = ( marca)(marca + t -> bal ) ? ( marca + 2) : marca +t ->
   bal +2));
   t->bal = (aux > t ->bal +1) ? aux : t ->  bal + 1 ); /* corrige
   factor de balance de t */
   return  (t);
}
```

```
pnode lrot (t)    /* rutina hace rotación a izq. */
pnode t;
{  p node temp = t;
   int    marca, aux;
   t = t -> right;
   temp -> right = t ->left;
   t->left= temp;
   marca = temp -> bal;
   temp -> bal += -1 -(( t-> bal > 0) ? t -> bal:0); /* corrige
   factor de bal. de temp */
   aux = ( marca < (t -> bal - 1 ) ? aux : t -> bal -1 ) ; /* corrige
   factor de bal. de t */
   t->bal = (aux < t ->bal +1) ? aux : t ->  bal - 1 ); /* corrige
   factor de balance de t */
   return (t);
}
```

```
treeprint(p)  /*imprime el árbol recursivamente de ref. cruzadas */
pnode p;         /* imprime el dato numérico almacenado */
{  static int      level=0; /* mantiene nivel de profundidad del
                                                  árbol */
   int    i;
   if (p! = NULL_P)    { /* si no es una rama del árbol */
          level++;
          treeprint(p -> right);    /* llama a treeprint por izq. */
          for (i=0; i < level; i++)           /* da forma al árbol */
                printf("           ");
          printf(" %3d\n",p -> key); /* imprime key */;
          treeprint(p -> left);      /* llama a treeprint por izq. */
          level--;
   }
}
```

Gonnet 3.4.1.3

pág. 74

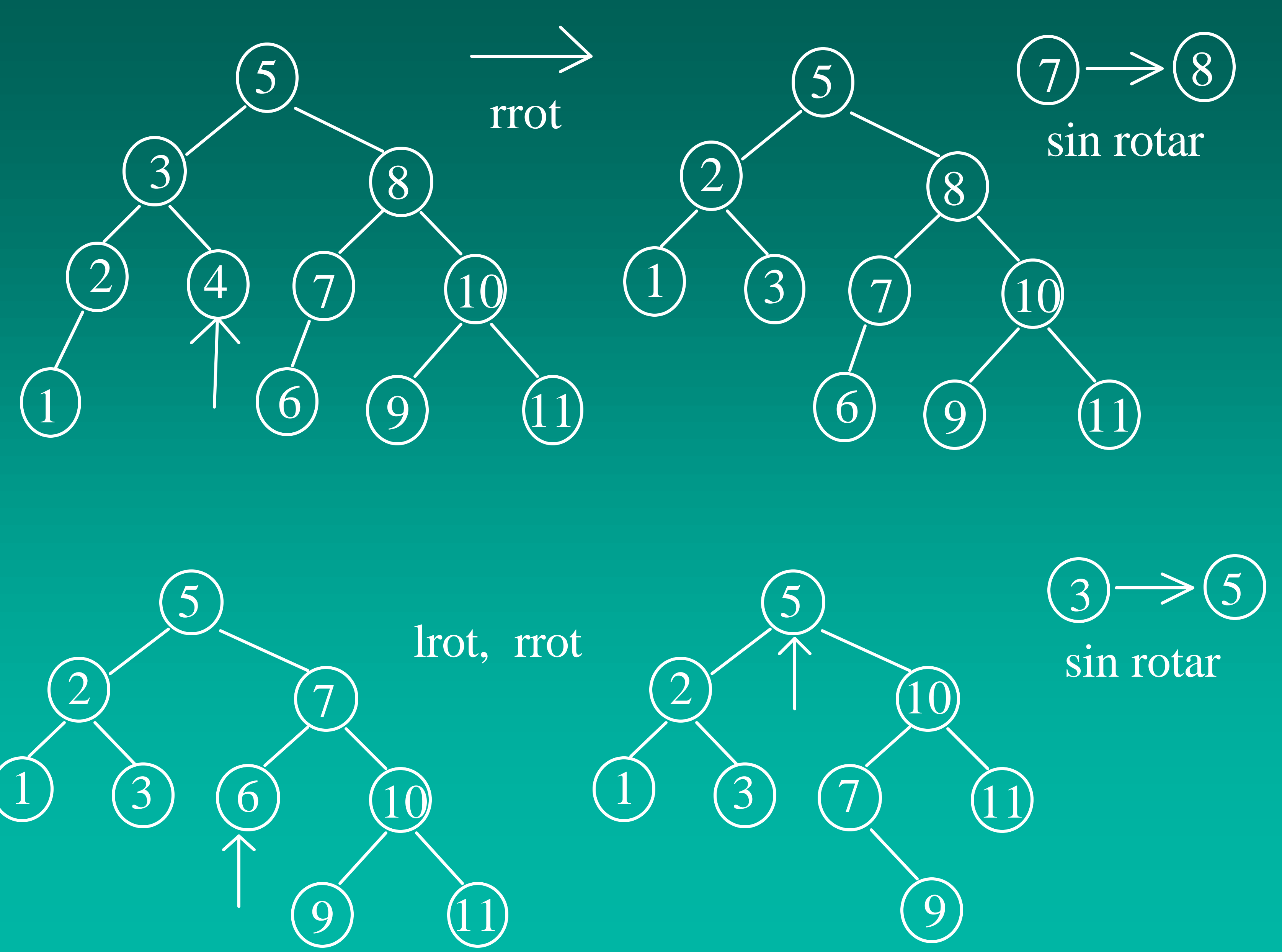


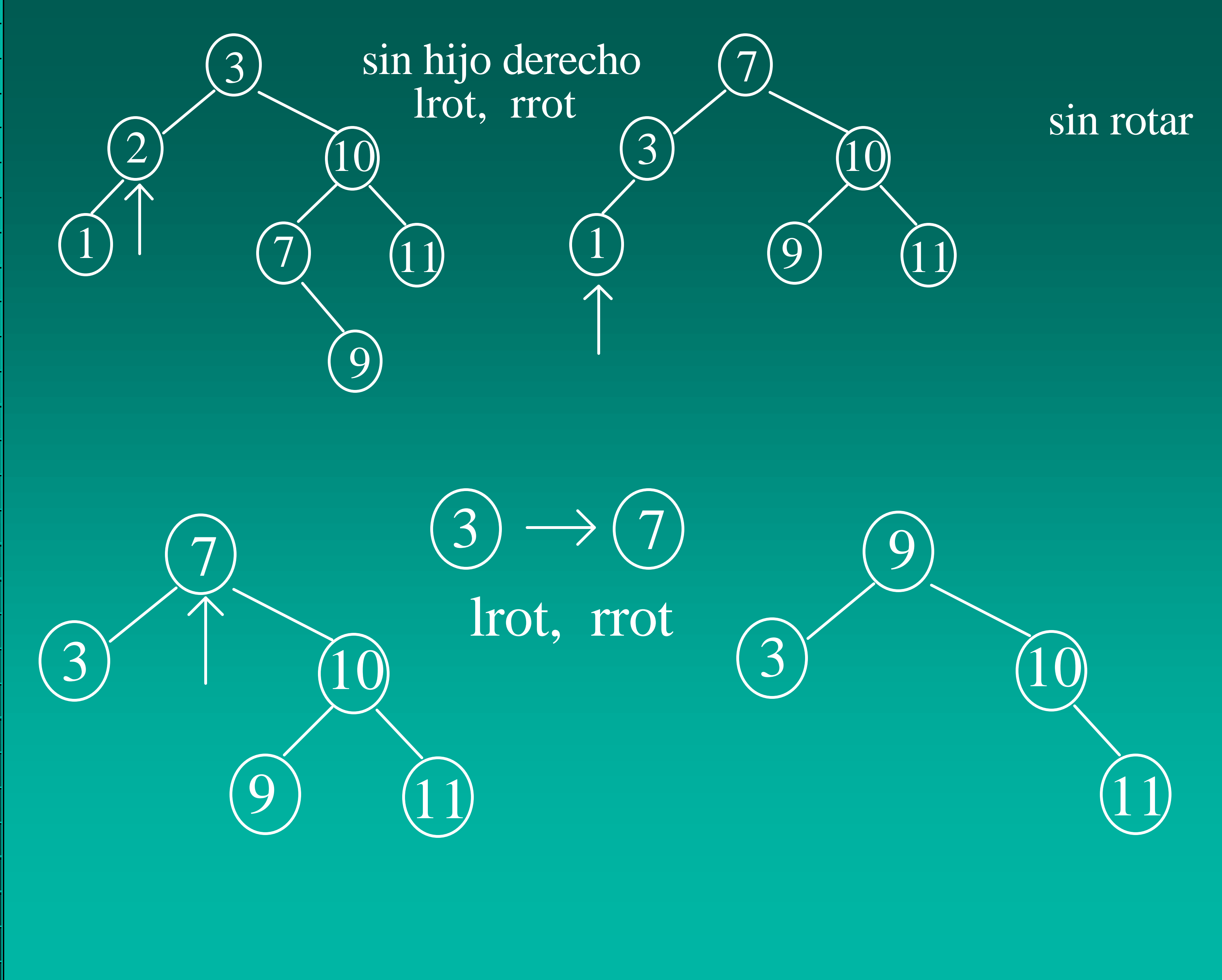

```
int insert (x,p)
int      x;
pnode *p;
{
   int   inc;
   pnode lrot(), rrot();
   pnode       make_nodo();
```

```
/*  código para insertar en el arbol */
   if (*p == NULL_P ) { /* llega a insertar como hoja */
         ( *p ) = make_nodo(); /* crea nodo */
         ( *p ) -> key = x ; /* almacena valor en nodo */
         return (1); /* devuelve 1 , árbol creció */  }

   if (x =( *p ) -> key )          { /* x ya estaba */
         (*p) -> count += 1 ; /* incrementa contador */
         return(0); /* retorna 0, árbol no creció */}

   if  ( x > ( *p ) -> key ) /* recursión por la derecha */
         inc = insert(x,&((*p)-> right)); /* si árbol crece por la
                                            der. inc es positivo*/
   else
         inc = -insert(x,&((*p)-> left)); /* si árbol crece por la
                                                izq. inc es negativo*/
```

```
/* código para mantener el árbol balanceado */
    (*p) -> bal += inc;
    if ( inc != 0 && (*p) -> bal!=0 ) {/* si árbol creció y  (*p) -> bal!=0 */
            if ( (*p) -> bal>1 ) { /* si árbol cargado para derecha
                                    y no balanceado */
                                    if ( (*p) -> right -> bal > 0) /* rot simple, bal>0 */
                                            (*p) = lrot(*p);
                                    else { /* rot. doble */
                                    (*p)->right=rrot( (*p) -> right);/*bal<=0*/}
            else if ( (*p) -> bal< -1 ) { /* si árbol cargado para
                                                        izq.y no balanceado */
                                    if ( (*p) -> left -> bal < 0) /* rot simple */
                                            (*p) = rrot(*p);
                                    else { /* rot. doble */
                                            (*p)-> left = lrot ( (*p) -> left );}
            }
            else  /* aun es AVL */
                        return (1); /* devuelve 1 para llamadas precedentes*/
    }                               /* si aca no se corrigio, pero alla a lo mejor si */
    return(0); /* si no creció o bal = 0*/
}
```

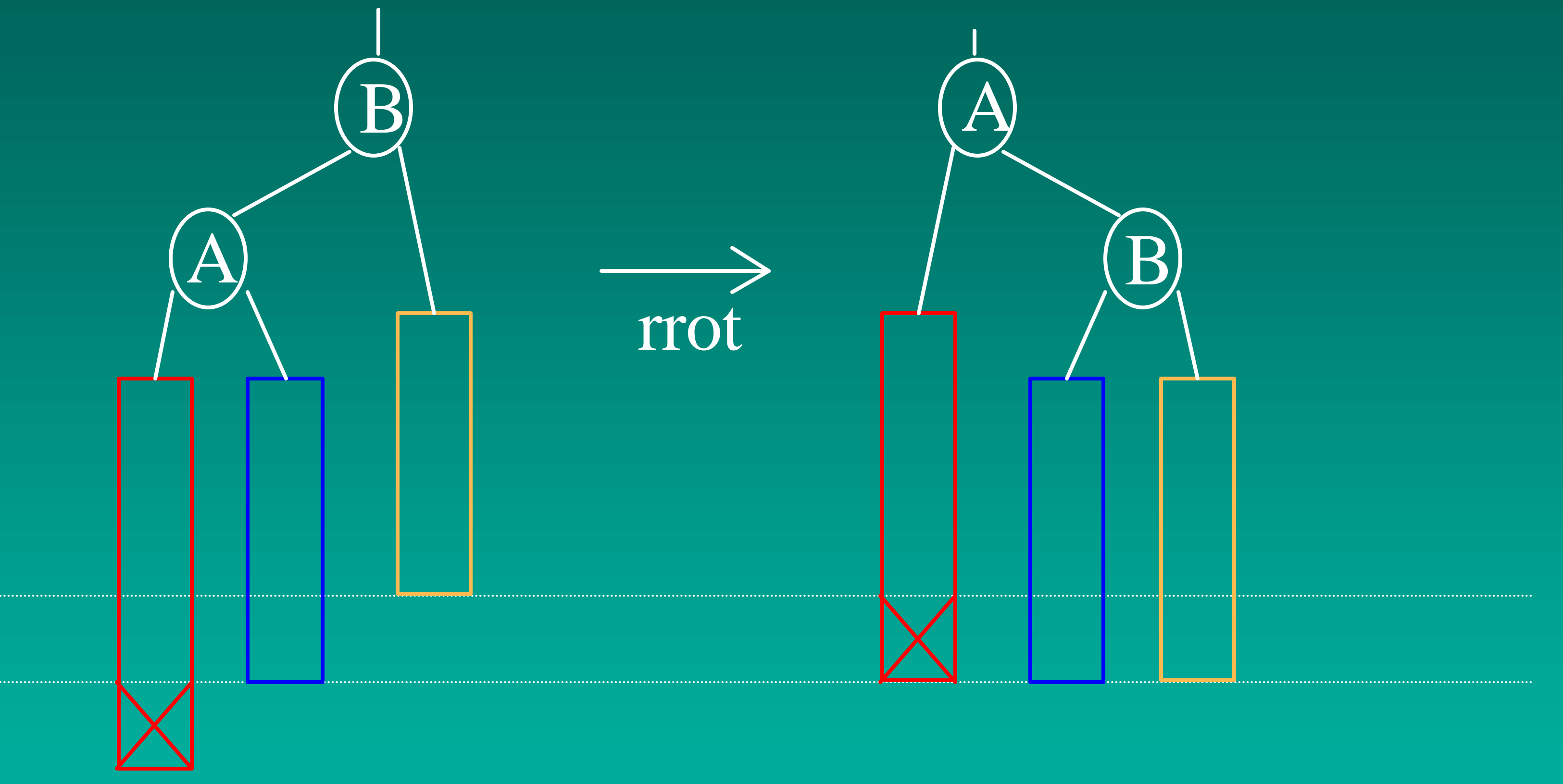
Descartar



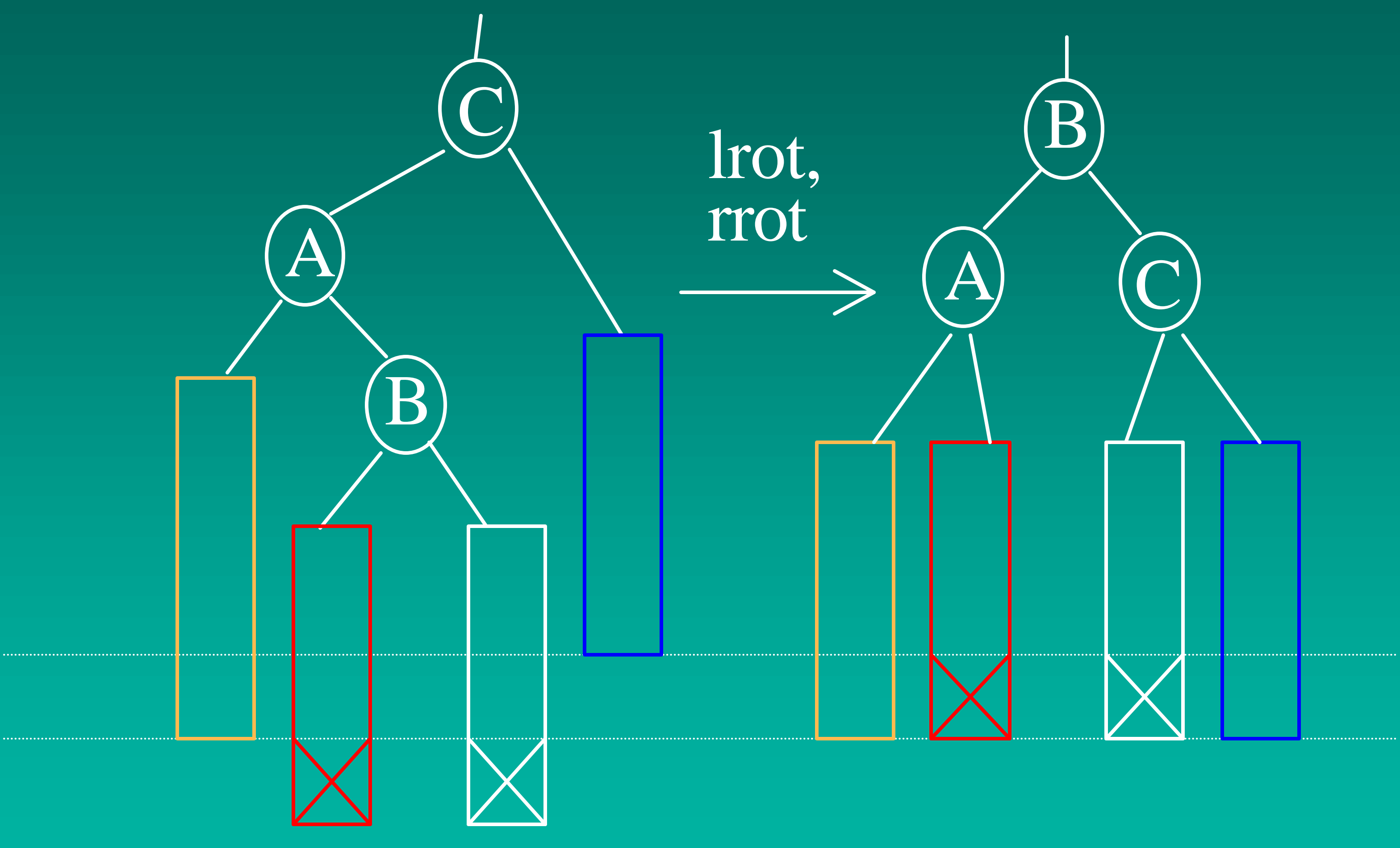


3
sin hijo derecho
lrot, rrot
7
2
10
3
10
sin rotar
1
7
11
1
9
11
9
3
7
7
lrot, rrot
9
3
10
3
10
9
11
11

- Caso 1



- Caso 2

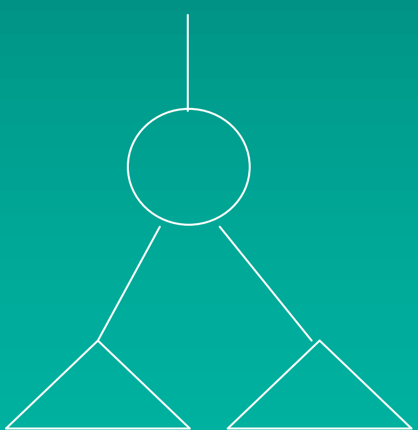


Gonnet 3.4.1.9 pag. 89.

```
int delete(x,p)  /* borra un elemento  del árbol manteniendolo
balanceado */
int     x;
pnode     *p;
{
pnode     aux;
pnode     lrot ( ), rrot ( )

if (*p= = NULL_P) /* llegamos a una hoja y elemento no estaba */
    printf( *\n\t Elemento a eliminar (%d) ausente \n", x);

else if ((*p) -> key (x)) /* x es mayor, se sigue a la derecha */
    delete  (x & (( *p) -> right ));

else if (( *p ) -> key ) x) */ x es menor se sigue por izq. */
    delete (x, & (( *p ) -> left ));
/* se encontró elemento */
```

```
else if ((*p ) -> left==NULL_P } {/* no hay hijo izq., se reemplaza nodo */
/* por subárbol derecho */

    aux = (*p);
    (*p) = (*p) -> right;
    free (( pchar laux );
}

else if (( *p ) -> right = = NULL_P ) {/* no hay hijo der., se
reemplaza nodo */
/* por subárbol derecho */

    aux = ( *p );
    ( *p ) = ( *p ) -> left;
    free ((pchar) aux );
}
```

```
else if (l_tree (( *p ) -> left) > l_tree ((  *p ) -> right ))
          /* si  p-> bal < 0        TIENE DOS HIJOS*/
   if ( l_tree (( *p ) -> left -> left)< l_tree (( *p )->left->right))
                              /* si p->left->bal<0*/
          ( *p )->left = lrot (( *p ) -> left);
   ( *p ) = rrot ( *p );
   delete (x, & (( *p )->right )); /* llamada recursiva */
```



izquierdo más liviano, hay que rotar, lrot, rrot.

```
else {              subárbol derecho más liviano, rotar, rrot, lrot.
        if (l_tree (( *p )->right ->left) >l_tree (( *p ) -> right ->right ))
        /* si p->right->bal<0*/
                ( *p )->right = rrot (( *p )->right);
        ( *p )= lrot ( *p );
        delete (x,& (( *p )->left));  /* llamada recursiva */
        }

if( *p ) = NULL_P
        /* si nodo no nulo, se calcula su factor de balance */
        ( *p )->bal=l_tree (( *p )->right ) -ltree(( *p )->left);
}
}/*  Fin algoritmo */
```

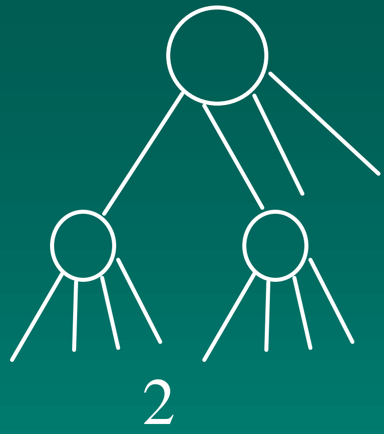
```
l_tree(t)                    /* iterativo */
pnodo t;
{       int = 0;
        while ( t! = NULL_P){
          if(t->bal>0) { h + + ; t = t -> right;}
          else { h + + ; t = t -> left }
        }
  return (h);
}
```

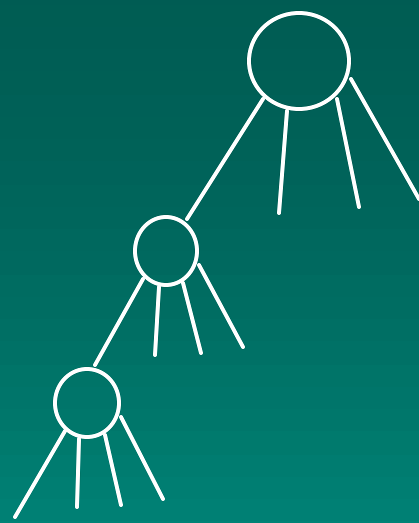
Multiárboles Varios Hijos

- Si aumentan los hijos implica disminuye la altura.

- Uso

  - Los punteros son direcciones del disco.
  - Si h es hijo implica pocos accesos a disco.
  - Se requiere controlar crecimiento para mantener balanceado





h = 1

N = m $\quad$ $m^2 + m$ $\quad$ $m^3 + m^2 + m$

$$N = m^h + m^{h-1} + \ldots\ldots + m =$$

$$= \frac{m}{m-1}(m^h - 1)$$

$$\lg_m N \approx h$$

- Comparación con binario

Para igual N:

$$m^{h_m} \approx \mathrm{N} \approx 2^{\mathrm{h_b}} \Rightarrow \frac{\mathrm{h_m}}{\mathrm{h_b}} = \frac{\lg 2}{\lg \mathrm{m}}$$

B-tree

N.W. 4.5.2 pag 245, Gonnet 34.2 pag 90

- ADT En un B-tree de orden n:

  -En cada nodo hay a lo menos n claves y 2n claves a lo más.

  - La raíz puede contener menos de n claves.

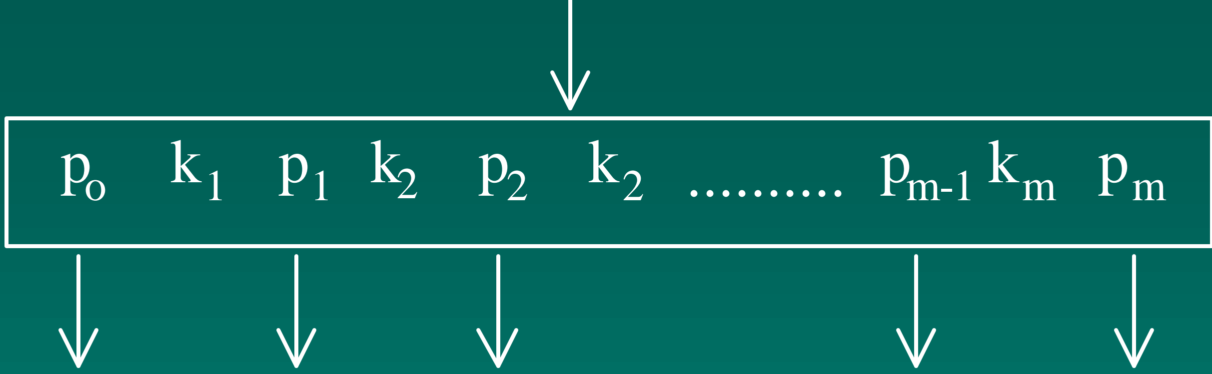
  - Las hojas tienen punteros nulos y deben tener igual altura.

- Si hay m claves, se tendrán (m + 1) punteros que apuntan a nodos descendientes.
- Si las claves se almacenan en forma ascendente:

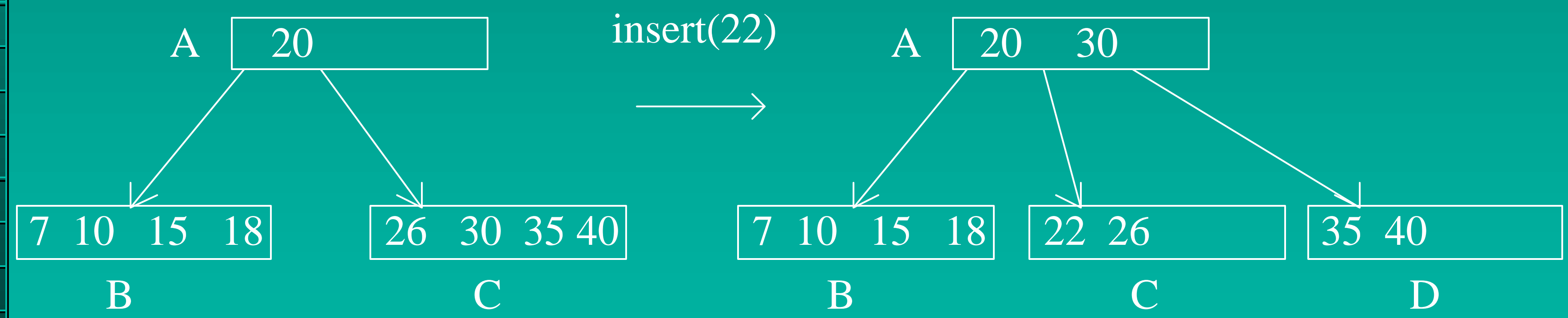
$$k_1 \langle k_2 \langle k_3 \langle \ldots \langle k_m \; ; \; n \leq m \leq 2n$$

- En el peor caso, hay n en cada página, con N item en B-tree se tiene:

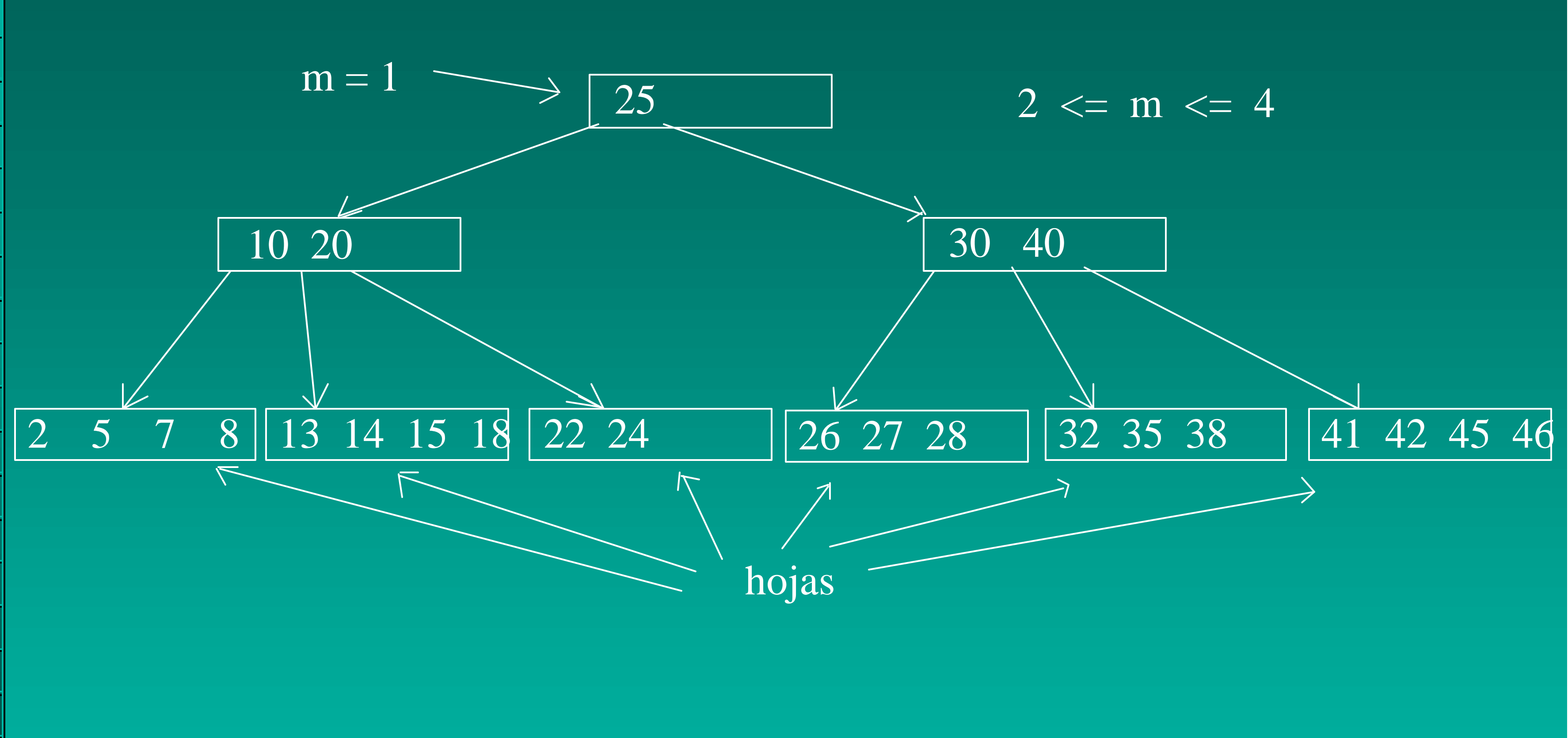
$$h = \lg_n N$$



- Si hay m claves : $n \leq m \leq 2n$
- Ejemplo, Para insertar 22 se parte C en C y D :



- Ejemplo :



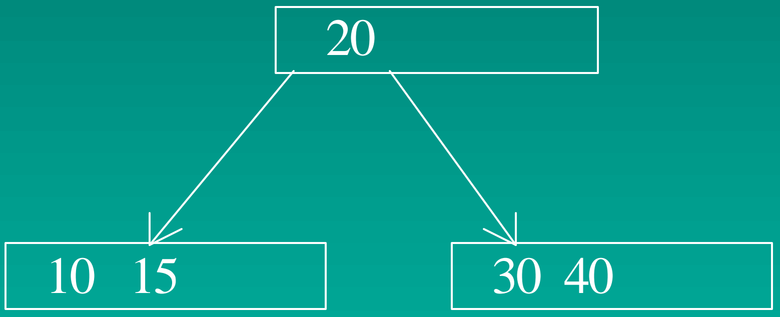
a) Arbol original 20

a1) 40 ; 20 40

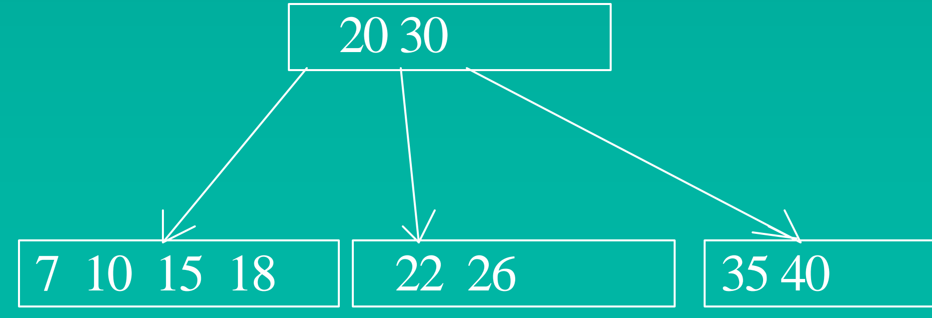
a2) 10 ; 10 20 40

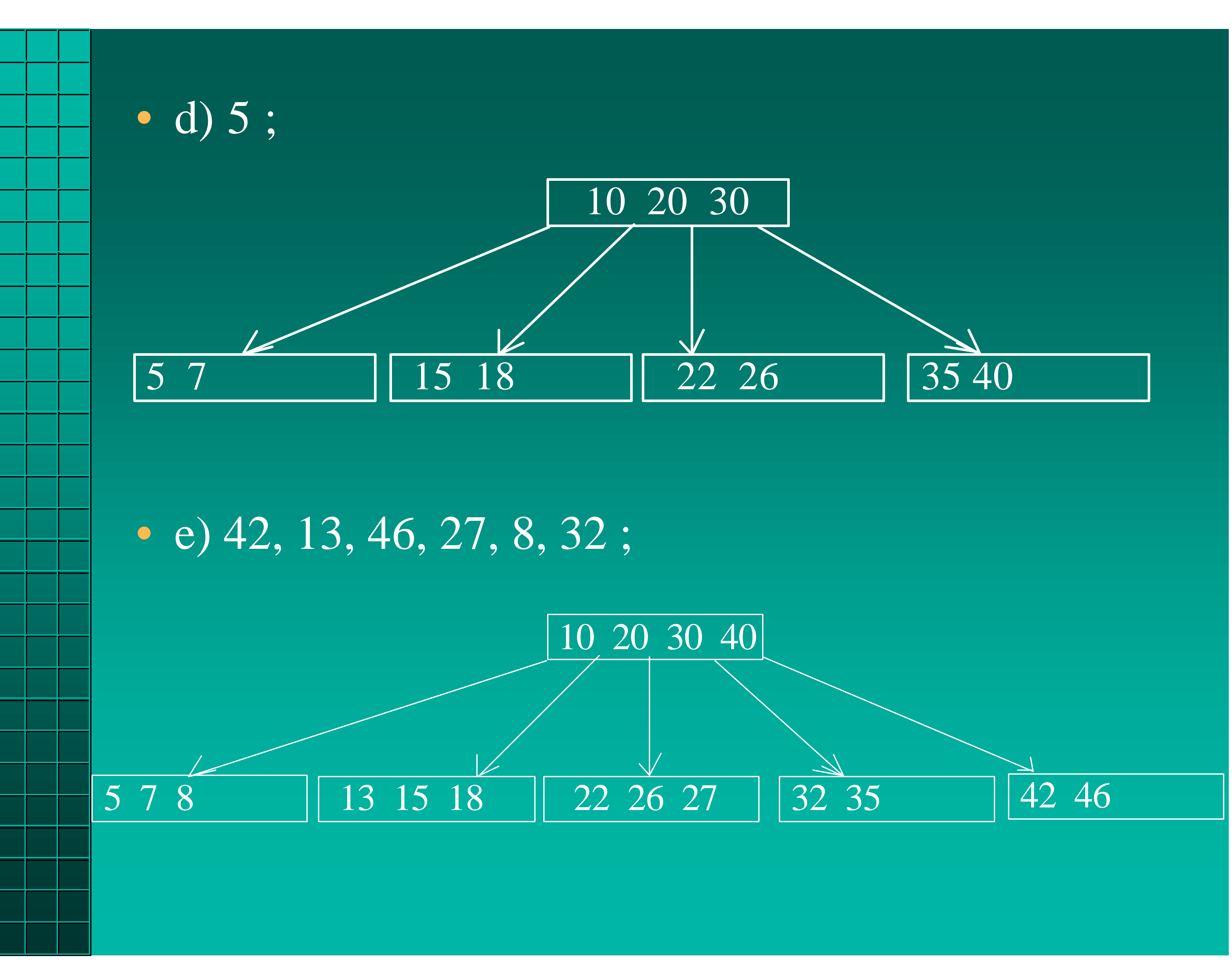
a3) 30 ; 10 20 30 40

b) 15 ;



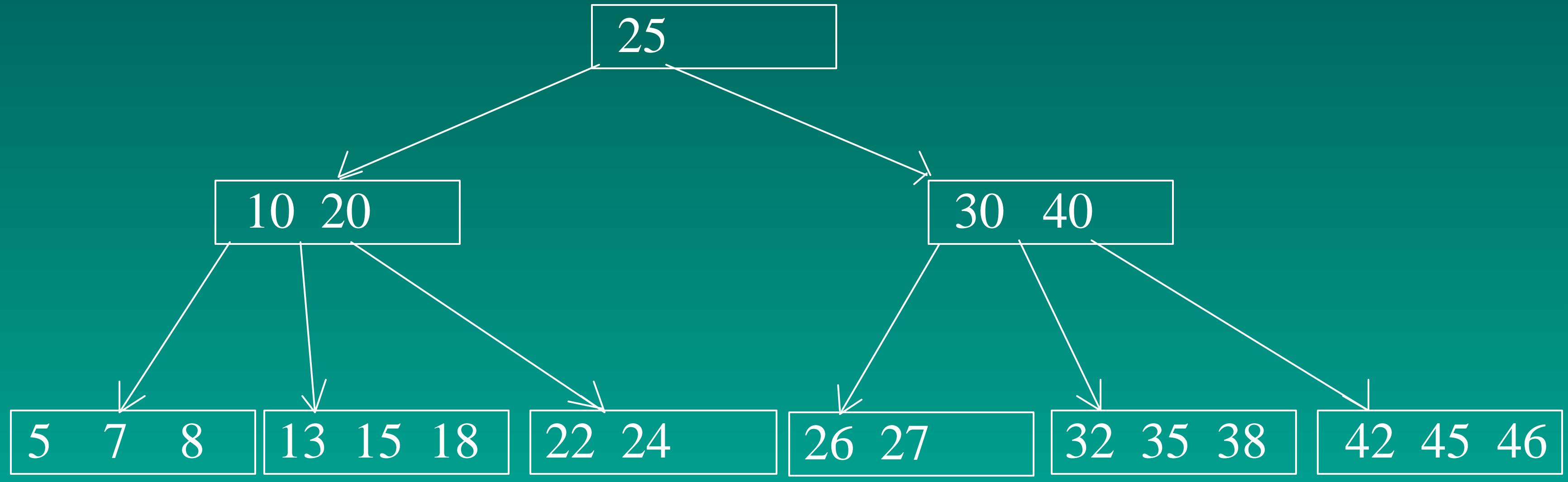
c) 35, 7, 26, 18, 22 ;





d) 5 ;
10 20 30
5 7
15 18
22 26
35 40
e) 42, 13, 46, 27, 8, 32 ;
10 20 30 40
5 7 8
13 15 18
22 26 27
32 35
42 46

- f) 38, 24, 45, 25



Operación : Buscar

- Si hay m claves :

$x < k_i$ buscar en descendiente apuntado por p_0

$x = k_i$ se encontró en página actual.

$k_i < x < k_{i-1}$ $(1 \leq i \leq m-1)$ buscar en página.

$k_m < x$ búsqueda continua en página apuntada por p_m

Operación : Insertar

- Si ya hay items en la página

a) Con m < 2n se inserta en página actual.

b) Con m = 2n se agrega una nueva página se redistribuyen los 2n + 1 items, n en una página, el central se pasa a la página superior, el resto a la nueva página.

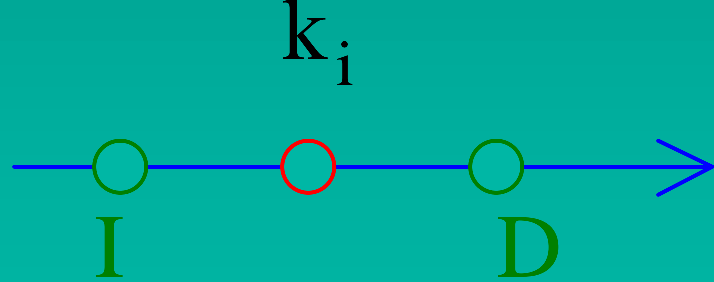
- La inserción en la página superior, puede requerir agregar una nueva págian. Esto puede propagarse hasta la raíz. Esta es la única forma de aumentar la altura del B-tree.

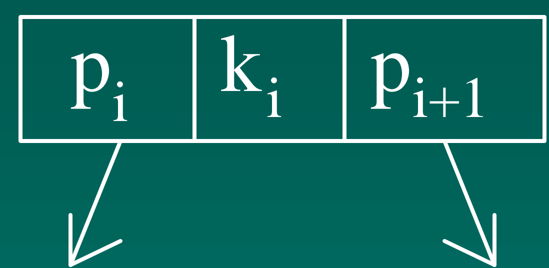
Operación : Descartar

- Si el item está en una hoja, su remoción es directa.

- Si el item no está en una hoja:

  Debe buscarse un adyacente, éste se encuentra en una hoja :





a) Para buscar D, se baja por Pi + 1 , y luego por el puntero más izquierdista del nivel inferior, hasta llegar a la hoja. Se escoge el más izquierdista de la hoja, se le copia en el lugar de ki , se lo elimina de la hoja.

b) Para buscar I, se baja por pi , y luego por el puntero más derechista del nivel inferior, hasta llegar a la hoja. Se escoge el más derechista, se lo elimina de la hoja y se copia su valor en ki

- Pero si la hoja disminuye de n items, se debe pedir prestado un item a la hoja adyacente (Esto para mantener criterio B-tree).

- Se puede escoger la hoja a la izquierda o a la derecha. En cualquier alternativa hay tratamiento especial en un borde.

- Sean las hojas P (de n - 1 items, después de eliminar) y Q (adyacente a P).

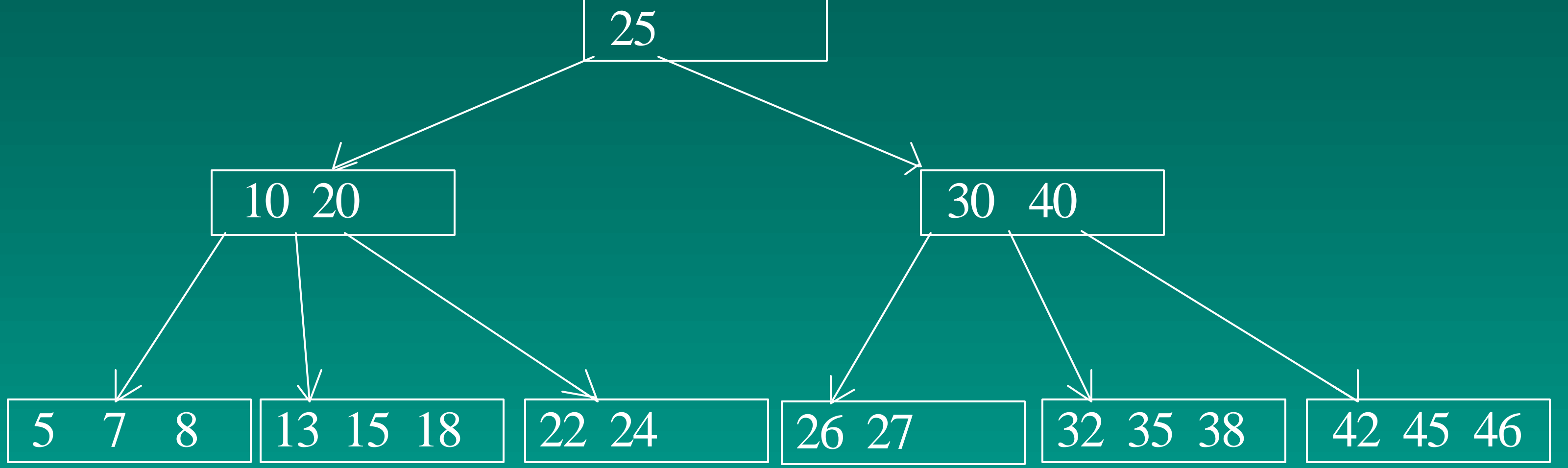
  - Si items de Q>n, se redistribuyen lo más parejo posible entre Q y P, empleando también el padre (esta operación se llama balancear).

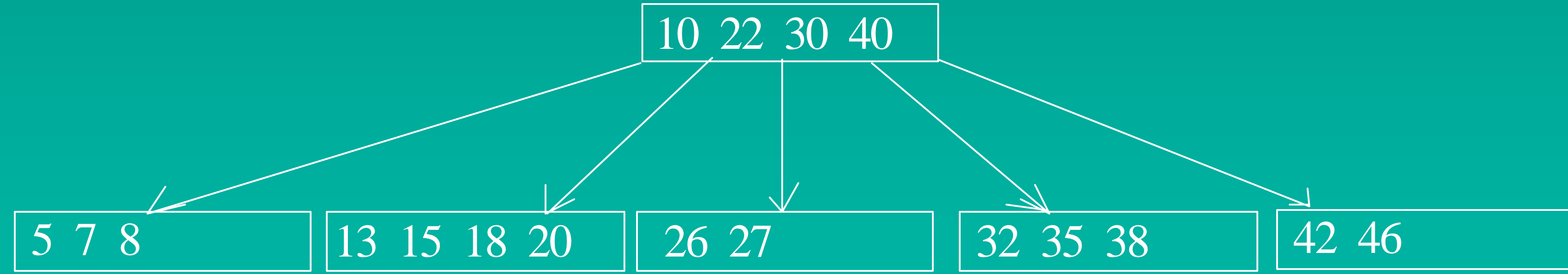
  - Si items de Q= n, el padre de P y Q, los (n-1) de P, y los n de Q (2n en total) se juntan en una hoja y se descarta la otra. (esta operación se llama mezclar).

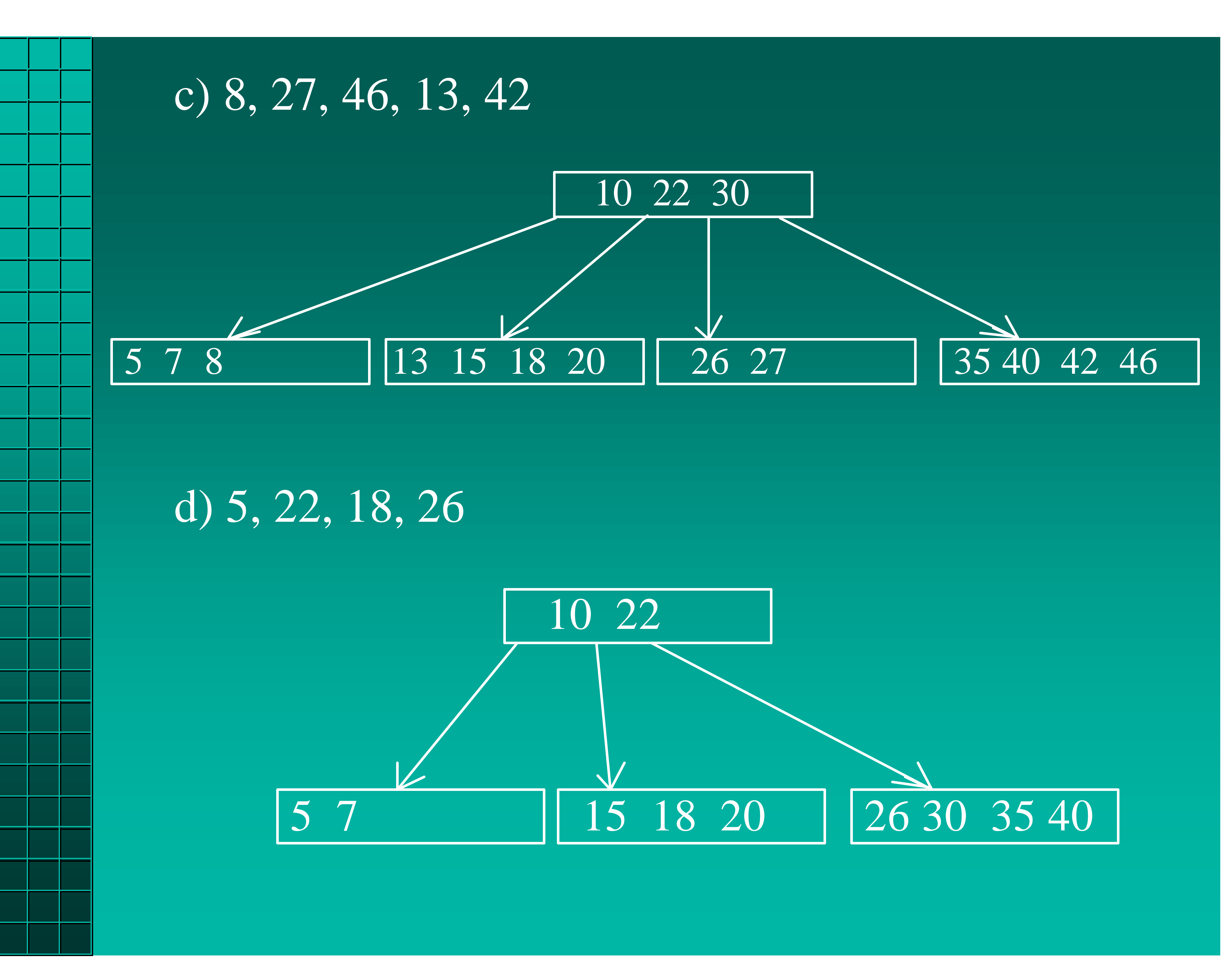
- Si padre de P y Q, baja de n items, se repite balancear o mezclar, en el nivel superior.
- Esta propagación puede llegar hasta la raíz.
- Si la raíz queda en cero items, se descarta.
- Esta es la única forma de disminuir la altura de B-tree.

- Ejemplo de Descartes :

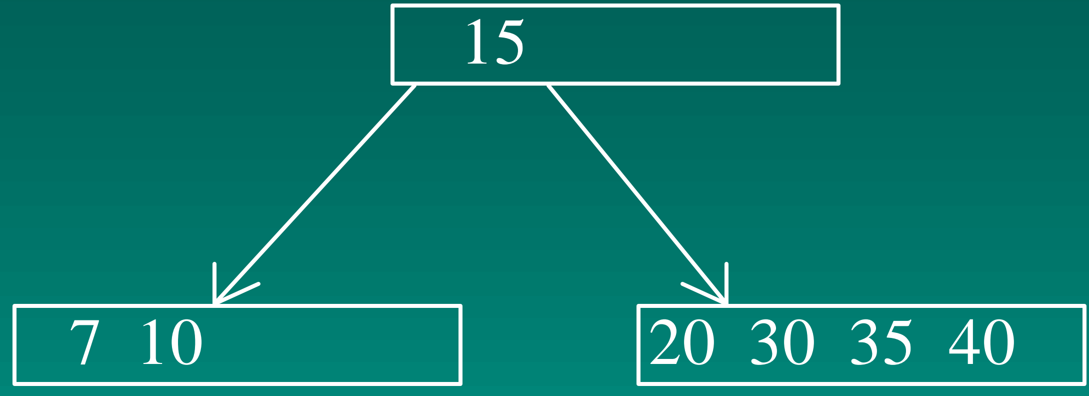
a) 25, 45, 24






c) 8, 27, 46, 13, 42
10 22 30
5 7 8
13 15 18 20
26 27
35 40 42 46
d) 5, 22, 18, 26
10 22
5 7
15 18 20
26 30 35 40

e) 7, 35, 15



f)

10 20 30 40

Program 4.7 B-Tree Search, Insertion, and Deletion

```
program Btree (input, output);
{B-tree search, insertion and deletion }
const  n = 2; nn = 4; {page size }
type ref = ↑page;
    item = record key; integer;
                p: ref;
                count: integer;
          end;
page =  record m: 0 . . nn; { no. of items}
                p0: ref;
                e: array  {1 .. nn }  of item;
          end;
```