

Estructuras de Datos y Algoritmos II

2 Curso - Grado en Ingeniería Informática

M José Polo Martín
mjpolo@usal.es

Universidad de Salamanca

curso 2023-2024

Contenido

- 1 Tema1. Árboles Generales y Binarios
- 2 Tema 2. Montículos Binarios
- 3 Tema 3. Conjuntos Disjuntos
- 4 Tema 4. Grafos
- 5 Tema 5. Árboles Binarios de Búsqueda
- 6 Tema 6. Organización de archivos
- 7 Tema 7. Organización de Índices

Contenido

1 Tema1. Árboles Generales y Binarios

- Definiciones y conceptos básicos
- Árboles Binarios
- Recorridos en Árboles Binarios
- Árboles Binarios Completos y Casi-Completos
- Ejercicios

1 Definiciones y conceptos básicos

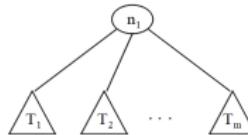
Árbol

Colección de elementos llamados nodos, uno de los cuales se distingue como **raíz**, con una relación entre ellos que impone una estructura **jerárquica**

Definición formal

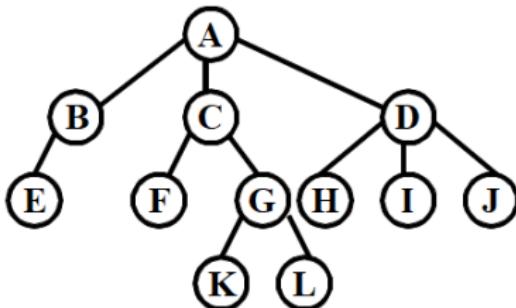
Un **árbol** T es un conjunto finito de cero o más nodos $\{n_1, n_2, \dots, n_i\}$ de tal forma que

- ① Hay un nodo especialmente designado llamado raíz ($n_1 = \text{raíz}(T)$)
- ② Los nodos restantes $\{n_2, n_3, \dots, n_i\}$ se dividen en **m** conjuntos disjuntos T_1, T_2, \dots, T_m ($m \geq 0$), llamados **subárboles** de la raíz, tales que cada T_j es, por sí mismo, un árbol



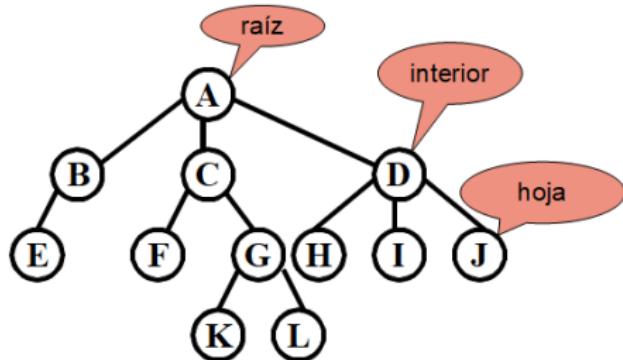
Características y propiedades

- **Recursión** característica inherente de los árboles
- Un árbol sin nodos es un **árbol vacío** o **nulo**
- Todo árbol que no es vacío tiene un **único** nodo **raíz**
- Un nodo X es **descendiente directo** de un nodo Y si existe una arista del nodo Y al nodo X (X “es hijo” de Y)
- Un nodo Z es **antecesor directo** de un nodo V si existe una arista del nodo Z al nodo V (Z “es padre” de V)
- Todos los nodos que son descendientes directos (hijos) de un mismo nodo (padre) se designan como **hermanos**
- Cada nodo puede tener un número arbitrario de descendientes directos, incluso cero



Definciones

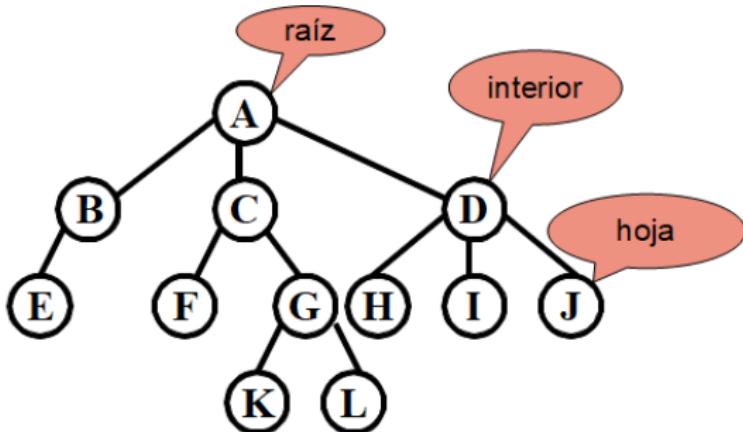
- Todos los nodos que no tienen descendientes directos se denominan nodos **hoja** o **terminales**
- Todo nodo con descendientes, es decir, que no es hoja, se designa como nodo **interior**
- **Grado de un nodo:** número de sus descendientes directos
- **Grado del árbol:** grado máximo de todos los nodos del árbol



Definiciones

- **Camino** de un nodo n_i a otro nodo n_k : secuencia de nodos n_i, n_{i+1}, \dots, n_k tal que n_j es el padre de n_{j+1} para $i \leq j < k$
 - **Longitud de un camino:** número de aristas que lo forman ($k-1$)
 - En un árbol hay exactamente un camino de la raíz a cada nodo
- Si existe un camino de un nodo n_i a otro nodo n_j , entonces n_j es **descendiente** de n_i y n_i es **antecesor** de n_j
- **Profundidad de un nodo:** número de aristas en el camino único que permite acceder a él desde la raíz. La raíz tiene profundidad cero
- **Altura de un nodo:** número de aristas en el camino más largo desde el nodo en cuestión hasta una hoja. Todos los nodos hoja tienen altura 0
- **Altura de un árbol** es igual a la altura de la raíz

Ejemplo



$$\text{grado}(A) = 3 \quad \text{profundidad}(A) = 0 \quad \text{altura}(A) = 3$$

$$\text{grado}(C) = 2 \quad \text{profundidad}(C) = 1 \quad \text{altura}(C) = 2$$

...

...

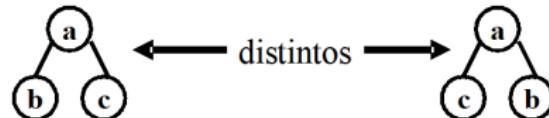
...

$$\text{grado}(K) = 0 \quad \text{profundidad}(K) = 3 \quad \text{altura}(K) = 0$$

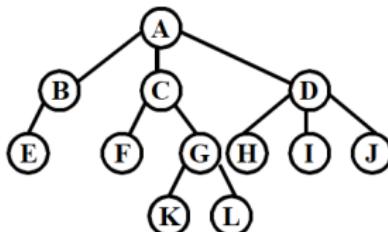
$$\boxed{\text{grado(árbol)} = 3 \quad \text{altura(árbol)} = 3}$$

Árboles ordenados

- Árboles en los que los hijos de cada nodo se ordenan de izquierda a derecha



- El orden de izquierda a derecha entre nodos hermanos puede extenderse para comparar dos nodos cualesquiera con el siguiente criterio: “ si **b** y **c** son hermanos y **b** está a la izquierda de **c**, entonces todos los descendientes de **b** están a la izquierda de todos los descendientes de **c**”.



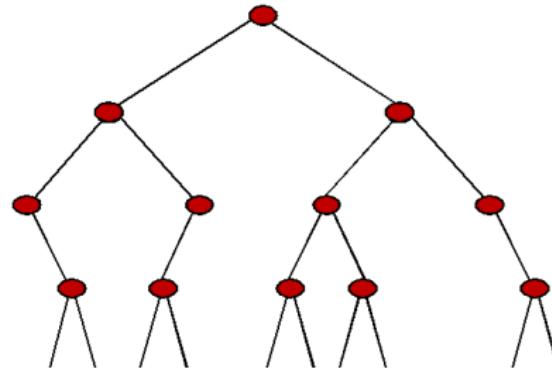
Contenido

1 Tema1. Árboles Generales y Binarios

- Definiciones y conceptos básicos
- Árboles Binarios
- Recorridos en Árboles Binarios
- Árboles Binarios Completos y Casi-Completos
- Ejercicios

2 Árboles Binarios

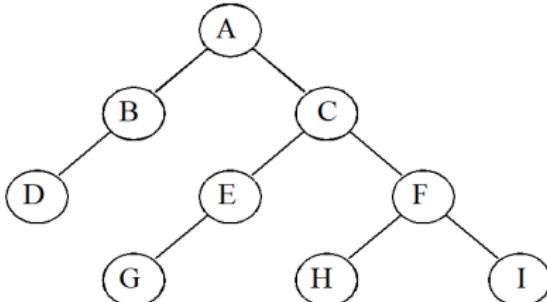
- **Definición 1:** conjunto finito de nodos que puede ser vacío o puede distribuirse en una raíz y un par de árboles binarios llamados subárboles izquierdo y derecho de la raíz, los cuales pueden también estar vacíos
- **Definición 2:** árbol ordenado de grado dos



2.1 Representación mediante Memoria Contigua

Matrices

- Los nodos se almacenan en las celdas contiguas de una matriz
- Cada celda de la matriz almacenará la información del nodo y dos enteros que indicarán los índices de la matriz donde se encuentran sus descendientes directos izquierdo y derecho (el valor 0 indicará ausencia de hijo)



1	A	2	3
2	B	4	0
3	C	5	6
4	D	0	0
5	E	7	0
6	F	8	9
7	G	0	0
8	H	0	0
9	I	0	0
10			
11			
...			
N-2			
N-1			
N			

Declaraciones Básicas

Consideraciones:

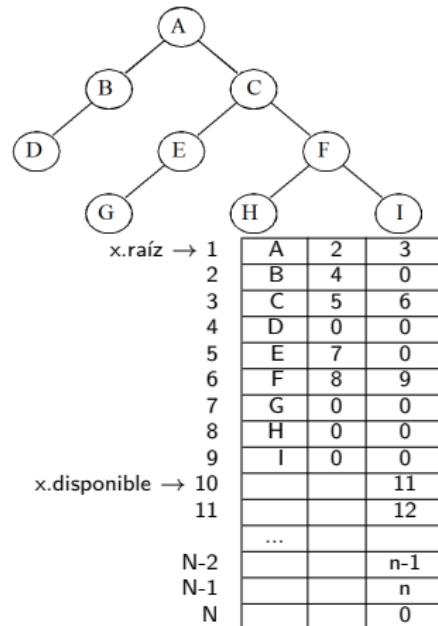
- Utilización de un matriz suficientemente grande para almacenar el número máximo de nodos que pueda tener el árbol
- Necesidad de una lista de disponibilidad para manejar el espacio en la matriz y un entero para indicar el índice de la matriz donde se encuentra la raíz

Algoritmo declaraciones básicas

```

1: constantes
2:   N = 100
3: tipos
4:   tipoNodo = registro
5:   información : tipolformación
6:   izq, der : entero
7: fin registro
8:   tipoÁrbol = registro
9:   nodos : matriz[1, ..., N] de tipoNodo
10:  raíz : entero
11:  disponible : entero
12: fin registro

```



Ejemplo variable x:tipoÁrbol

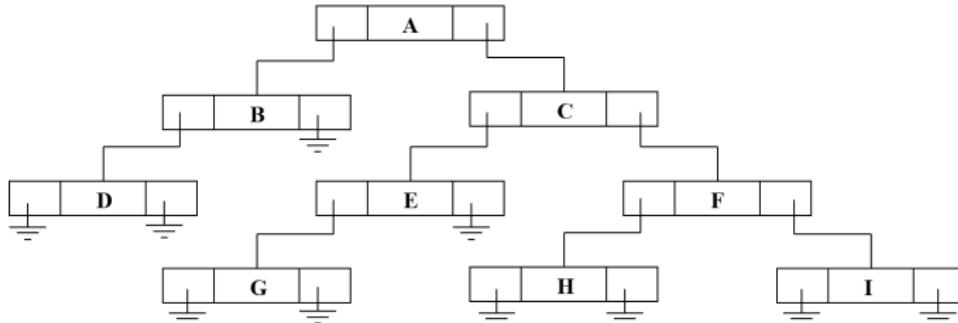
Para todo nodo situado en la celda i:

- Hijo izquierdo ⇒ x.nodos[i].izq
- Hijo derecho ⇒ x.nodos[i].der

2.2 Representación mediante Memoria Dispersa

Memoria dinámica (punteros)

- Cada celda se enlaza, a lo sumo, con otras dos siguiendo la estructura del árbol que representa
- Cada celda almacenará la información del nodo y dos apuntadores para enlazar con las raíces de los subárboles izquierdo y derecho del nodo
- Los nodos se almacenan en celdas que no tienen por qué ocupar posiciones consecutivas en memoria



Declaraciones Básicas

Consideraciones:

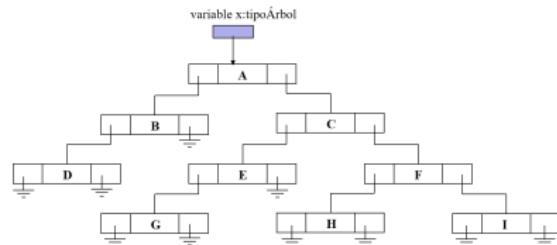
- Desde la raíz de un árbol puede llegarse al resto de los nodos
- Para acceder a un árbol solo se necesita la dirección de memoria donde se almacena la raíz ⇒ **El árbol puede representarse por un puntero a un nodo de árbol binario**

Algoritmo declaraciones básicas

```

1: tipos
2: tipoNodo = registro
3: izq :↑ tipoNodo
4: información : tipolformación
5: der :↑ tipoNodo
6: fin registro
7: tipoÁrbol :↑ tipoNodo
8: punteroNodo :↑ tipoNodo

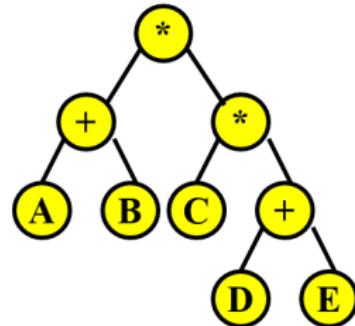
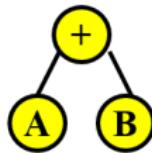
```



Este es el tipo de representación más común y la que utilizaremos en el resto del tema

Aplicaciones

- Numeración de capítulos y secciones de un libro
- Análisis de circuitos eléctricos
- Representación de expresiones aritméticas:



Prefija	Infija	Postfija
+AB	A+B	AB+
*+AB*C+DE	(A+B)*(C*(D+E))	AB+CDE+**

Ejemplo de aplicación: construcción de un árbol algebraico

Algoritmo generaÁrbol(expresión:cadena):tipoÁrbol

Entrada: expresión algebraica en notación Postfija

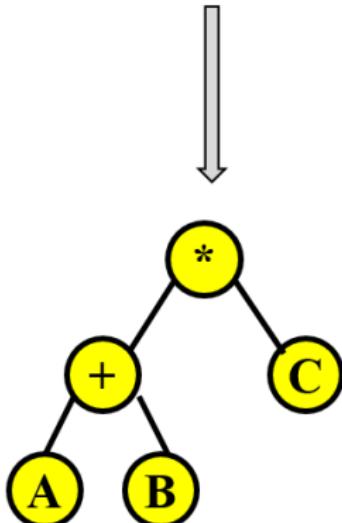
Salida: árbol binario que representa la expresión de entrada

```

1: a : tipoÁrbol
2: p : tipoPila
3: i : entero
4: símbolo : carácter
5: creaVacia(p)
6: símbolo  $\leftarrow$  expresión[1]
7: i  $\leftarrow$  1
8: mientras símbolo  $\neq$  FINEXPRESIÓN hacer
9:   casos símbolo en
10:    operando :
11:      a  $\leftarrow$  creaNodo(símbolo)
12:      inserta(a, p)
13:    operador:
14:      a  $\leftarrow$  creaNodo(símbolo)
15:      a  $\uparrow$ .der  $\leftarrow$  suprime(p)
16:      a  $\uparrow$ .izq  $\leftarrow$  suprime(p)
17:      inserta(a, p)
18:   fin casos
19:   i  $\leftarrow$  i + 1
20:   símbolo  $\leftarrow$  expresión[i]
21: fin mientras
22: devolver suprime(p)

```

A B + C *



Obsérvese la utilización del **TAD pila** en la implementación de este algoritmo

Especificación del TAD PILA utilizado en el ejemplo

- Pila cuyos elementos son punteros a nodos de árboles binarios
- Operaciones básicas:
 - **creaVacía(p)**: inicia o crea la pila **p** como una pila vacía, sin ningún elemento
 - **vacía(p)**: devuelve verdadero si la pila **p** está vacía, y falso en caso contrario
 - **inserta(x,p)**: añade el elemento **x** a la pila **p** convirtiéndolo en el nuevo tope o cima de la pila
 - **suprime(p)**: devuelve y elimina el elemento del tope o cima de la pila **p**

Contenido

1 Tema1. Árboles Generales y Binarios

- Definiciones y conceptos básicos
- Árboles Binarios
- Recorridos en Árboles Binarios
- Árboles Binarios Completos y Casi-Completos
- Ejercicios

3 Recorridos en Árboles Binarios

- Recorrer un árbol: visitar todos sus nodos de forma sistemática, de tal manera que cada nodo sea visitado una sola vez
- Un nodo es visitado cuando se encuentra en el recorrido y en ese momento se puede efectuar cualquier proceso sobre su contenido
- Categorías básicas de recorridos:
 - en PROFUNDIDAD: basados en las relaciones padre-hijo de los nodos
 - en AMPLITUD o por NIVELES: basados en la distancia de cada nodo a la raíz

Recorridos en PROFUNDIDAD

- Existen tres métodos diferentes de efectuar el recorrido en profundidad, todos ellos de naturaleza recursiva, imponiendo un orden secuencial y lineal a los nodos del árbol
- Las actividades básicas a realizar son:
 - visitar la raíz
 - recorrer el subárbol izquierdo
 - recorrer el subárbol derecho
- Los tres métodos se diferencian en el orden en que se visite la raíz, dando lugar a los tres recorridos:
 - EN-ORDEN
 - PRE-ORDEN
 - POST-ORDEN

Recorrido PRE-ORDEN

- ① Visitar la raíz
- ② Recorrer en PRE-ORDEN el subárbol izquierdo
- ③ Recorrer en PRE-ORDEN el subárbol derecho

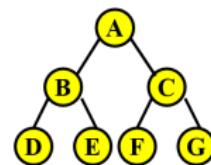
Algoritmo preOrden(a : tipoÁrbol)

Entrada: a dirección del nodo raíz del árbol

Salida: procesa todos los nodos en preorden

```

1: si a ≠ NULO entonces
2:   visitar(a ↑ .información)
3:   preOrden(a ↑ .izq)
4:   preOrden(a ↑ .der)
5: fin si
  
```



Recorrido EN-ORDEN

- ① Recorrer en En-ORDEN el subárbol izquierdo
- ② Visitar la raíz
- ③ Recorrer en EN-ORDEN el subárbol derecho

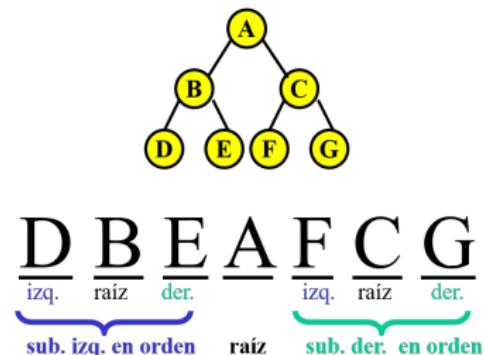
Algoritmo enOrden(a : tipoÁrbol)

Entrada: a dirección del nodo raíz del árbol

Salida: procesa todos los nodos en orden

```

1: si a ≠ NULO entonces
2:   enOrden(a ↑ .izq)
3:   visitar(a ↑ .información)
4:   enOrden(a ↑ .der)
5: fin si
  
```



Recorrido POST-ORDEN

- ① Recorrer en POST-ORDEN el subárbol izquierdo
- ② Recorrer en POST-ORDEN el subárbol derecho
- ③ Visitar la raíz

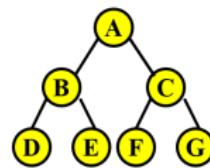
Algoritmo postOrden(a : tipoÁrbol)

Entrada: a dirección del nodo raíz del árbol

Salida: procesa todos los nodos en postorden

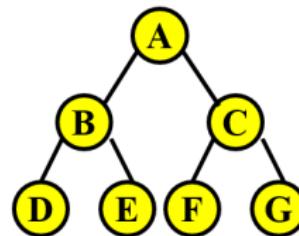
```

1: si a ≠ NULO entonces
2:   postOrden(a ↑ .izq)
3:   postOrden(a ↑ .der)
4:   visitar(a ↑ .información)
5: fin si
  
```



Recorrido en AMPLITUD o por NIVELES

- Consiste en visitar primero la raíz del árbol, después los nodos que se encuentran en siguiente nivel, etc.
- En este recorrido no importa tanto la estructura recursiva del árbol, si no la distribución de los nodos en los diferentes niveles
- Una posible implementación puede efectuarse utilizando una cola cuyos elementos son punteros a nodos del árbol



A BC DEF
nivel 0 nivel 1 nivel 2

Algoritmo de recorrido en AMPLITUD

Algoritmo niveles(*a* : tipoÁrbol)

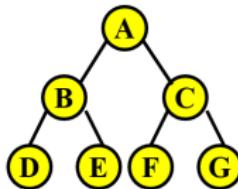
Entrada: *a* dirección del nodo raíz del árbol

Salida: procesa todos los nodos por niveles

```

1: nodo : punteroNodo
2: c : tipoCola
3: creaVacia(c)
4: si a ≠ NULO entonces
5:   inserta(a, c)
6: fin si
7: mientras NOT(vacia(c)) hacer
8:   nodo ← suprime(c)
9:   visitar(nodo ↑ .información)
10:  si nodo ↑ .izq ≠ NULO entonces
11:    inserta(nodo ↑ .izq, c)
12:  fin si
13:  si nodo ↑ .der ≠ NULO entonces
14:    inserta(nodo ↑ .der, c)
15:  fin si
16: fin mientras

```



A	BC	DEFG
nivel 0	nivel 1	nivel 2

Obsérvese la utilización del **TAD cola** en la implementación de este algoritmo

Especificación del TAD COLA utilizado en algoritmo amplitud

- Cola cuyos elementos son punteros a nodos de árboles binarios
- Operaciones básicas:
 - **creaVacía(c)**: inicia o crea la cola c como una cola vacía, sin ningún elemento
 - **vacía(c)**: devuelve verdadero si la cola c está vacía, y falso en caso contrario
 - **inserta(x,c)**: añade el elemento x a la cola c convirtiéndolo en el último elemento de cola
 - **suprime(c)**: devuelve y elimina el primer elemento la cola c

Contenido

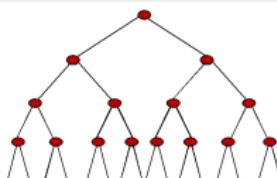
1 Tema1. Árboles Generales y Binarios

- Definiciones y conceptos básicos
- Árboles Binarios
- Recorridos en Árboles Binarios
- Árboles Binarios Completos y Casi-Completos
- Ejercicios

4 Árboles Binarios Completos y Casi-Completos

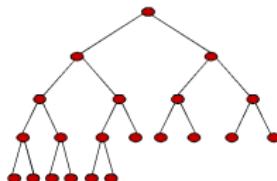
Árbol Binario Completo

Árbol binario que contiene el número máximo de nodos para su altura



Árbol Binario Casi-Completo

Árbol binario completamente lleno, con la posible excepción del nivel más bajo, el cual se llena de izquierda a derecha



Característica: tienen la altura mínima para su conjunto de nodos

Altura de un árbol binario de N nodos

- Altura máxima \Rightarrow lista de nodos

$$H_{\max} = N - 1$$

- Altura mínima \Rightarrow árbol binario casi-completo

$N = 1$	$H_{\min} = 0$	$N_{\min} = 1 = 2^0$	$N_{\max} = 1 = 2^1 - 1$
$2 \leq N \leq 3$	$H_{\min} = 1$	$N_{\min} = 2 = 2^1$	$N_{\max} = 3 = 2^2 - 1$
$4 \leq N \leq 7$	$H_{\min} = 2$	$N_{\min} = 4 = 2^2$	$N_{\max} = 7 = 2^3 - 1$
$8 \leq N \leq 15$	$H_{\min} = 3$	$N_{\min} = 8 = 2^3$	$N_{\max} = 15 = 2^4 - 1$
...
N	$H_{\min} = h$	$N_{\min} = 2^h$	$N_{\max} = 2^{h+1} - 1$

$$H_{\min} = h = \lceil \log N \rceil$$

- Número de nodos de un a.b. casi-completo de altura h

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

Más aplicaciones: Árboles de Búsqueda

- Los árboles binarios suelen utilizarse para estructurar una colección de datos de forma que faciliten la búsqueda de un elemento particular
- Conviene, por tanto, organizar los datos de forma que las longitudes de los caminos de búsqueda sean mínimas ⇒ Árboles binarios CASI-COMPLETOS y sus variantes
 - Árboles Binarios de Búsqueda y Balanceados, los estudiaremos en el tema 5
 - Árboles Multicamino: extensiones de los Árboles Balanceados que se utilizan para búsqueda de información en memoria secundaria, se estudiarán en el tema 7, dedicado a la Organización de Índices (Árboles B y Árboles B+)

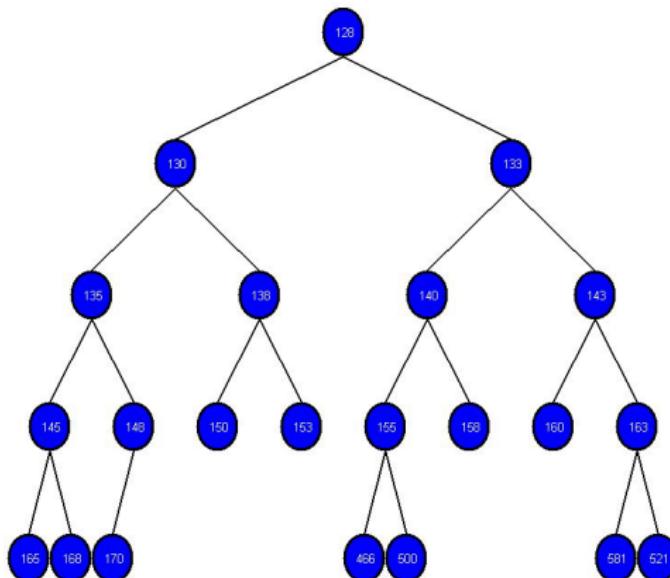
Contenido

1 Tema1. Árboles Generales y Binarios

- Definiciones y conceptos básicos
- Árboles Binarios
- Recorridos en Árboles Binarios
- Árboles Binarios Completos y Casi-Completos
- Ejercicios

Ejercicios sobre recorridos

1. Utilizando la aplicación RAED Representación de Algoritmos de Estructuras de Datos analizar el comportamiento de los algoritmos de recorridos sobre diferentes ejemplos. Se dejan dos ficheros creados con la aplicación raed que pueden descargarse para ser utilizados con la misma. Uno de ellos se corresponde con el árbol que muestra la siguiente figura y el otro es similar a los árboles presentados en las transparencias de recorridos.



Ejercicios sobre recorridos

2. Dibujar razonadamente un árbol binario sabiendo su recorrido en preorden y en orden:

- Preorden: 1234. En orden: 1342
- Preorden: 1234. En orden: 3421
- Preorden: 4321. En orden: 3124
- Preorden: 4321. En orden: 4213
- Preorden: ESTRUCTURAS. En orden: RTUSECUTARS
- Preorden: CRSETUTUARS. En orden: ESTRUCTURAS

Contenido

- 1 Tema1. Árboles Generales y Binarios
- 2 Tema 2. Montículos Binarios
- 3 Tema 3. Conjuntos Disjuntos
- 4 Tema 4. Grafos
- 5 Tema 5. Árboles Binarios de Búsqueda
- 6 Tema 6. Organización de archivos
- 7 Tema 7. Organización de Índices

Contenido

2 Tema 2. Montículos Binarios

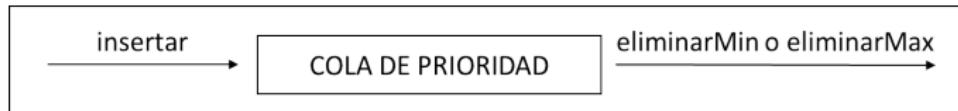
- Introducción
- Nivel abstracto o de definición
- Nivel de representación o implementación
- Ordenación por montículos
- Ejercicios

1 Introducción

Cola de PRIORIDAD

Tipo abstracto de datos que almacena una colección de elementos y que permite al menos las dos operaciones siguientes:

- Insertar un elemento
- Buscar, devolver y eliminar el elemento con valor mínimo|máximo en alguno de sus campos de información(clave)



Aplicaciones y posibles implementaciones

- Aplicaciones:

- Colas de impresión con prioridades
- Planificador sistema operativo en entorno multiusuario
- Nuevo Algoritmo de Ordenación
- Implantación eficiente de algunos algoritmos de grafos (tema 4)

- Posibles implementaciones:

- Listas enlazadas
- Listas enlazadas clasificadas
- Árbol binario de búsqueda
- **Montículo binario, montículo o *heap***

Contenido

2 Tema 2. Montículos Binarios

- Introducción
- Nivel abstracto o de definición
- Nivel de representación o implementación
- Ordenación por montículos
- Ejercicios

2 Nivel abstracto o de definición

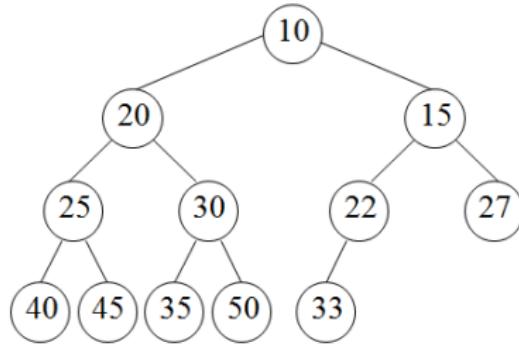
Montículo Binario

Árbol binario casi-completo (completo, con la posible excepción del nivel más bajo, el cual se llena de izquierda a derecha) en el cual, para todo nodo n se cumple la siguiente condición:

“la clave en el padre de n es menor (o igual) que la clave de n , con la excepción obvia de la raíz (que no tiene nodo padre)”

- Observaciones:
 - Regularidad de un a.b. casi-completo \Rightarrow se puede representar mediante una matriz sin necesidad de recurrir a punteros
 - El elemento con valor mínimo siempre se encuentra en la raíz
 - Puede redefinirse cambiando la condición de prioridad de mínimo a máximo

Ejemplo



?	10	20	15	25	30	22	27	40	45	35	50	33			
0	1	2	3	4	5	6	7	8	9	10	11	12			

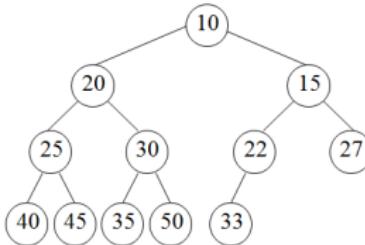
Propiedades

Propiedad de orden

El elemento con valor mínimo siempre se encuentra en la raíz

Propiedad de la estructura

Para cualquier elemento en la posición i de la matriz, el hijo izquierdo está en la posición $2i$, el hijo derecho está en la posición siguiente al hijo izquierdo ($2i + 1$) y el padre está en la posición $[i/2]$



?	10	20	15	25	30	22	27	40	45	35	50	33			
0	1	2	3	4	5	6	7	8	9	10	11	12			

Especificación de operaciones

- **inserta(x,m)**: Inserta un nuevo elemento **x** en el montículo **m**
- **eliminaMin(m)**: Elimina y devuelve el elemento del montículo **m** con valor mínimo en su campo clave
- **decrementaClave(p,x,m)**: Reduce el valor de la clave del elemento de la posición **p** del montículo **m** una cantidad positiva **x**
- **incrementaClave(p,x,m)**: Aumenta el valor de la clave del elemento de la posición **p** del montículo **m** una cantidad positiva **x**
- **construirMonticulo(m)**: Construye un montículo binario a partir de una colección de elementos

Contenido

2 Tema 2. Montículos Binarios

- Introducción
- Nivel abstracto o de definición
- Nivel de representacion o implementación
- Ordenación por montículos
- Ejercicios

3 Nivel de representación o implementación

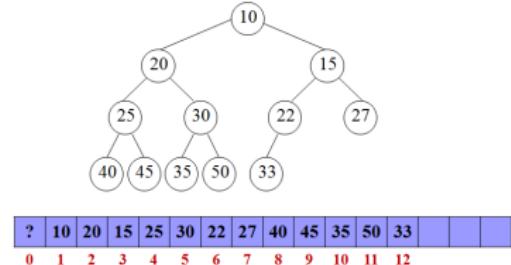
- Un montículo binario puede representarse mediante una matriz y un entero que indique el tamaño actual del montículo

Algoritmo declaraciones básicas

```

1: constantes
2: MAX = 100
3: tipos
4: tipoElemento = registro
5: clave : tipoClave
6: información : tipoInformación
7: fin registro
8: tipoMontículo = registro
9: elementos : matriz[0..MAX] de tipoElemento
10: tamaño : entero
11: fin registro

```



Inserción de un elemento en el montículo

- Añadir un elemento x al montículo manteniendo las propiedades de estructura y orden que los definen
- Proceso:
 - ① Añadir un nuevo nodo (hueco) en la siguiente posición disponible del a.b. casi-completo.
 - ② Distinguir los siguientes casos:
 - Si x se puede asignar al hueco manteniendo la propiedad de orden del montículo, se asigna y finaliza el proceso
 - En otro caso se desliza el elemento del nodo padre al hueco, subiendo el hueco hacia la raíz, hasta poder asignar x al hueco

Estrategia de Filtrado Ascendente

El nuevo elemento se filtra en el montículo hasta encontrar su posición correcta

Ejemplo filtrado ascendente

insertar un nuevo elemento con clave 5 en el montículo

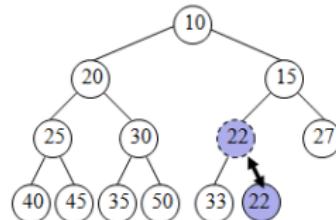
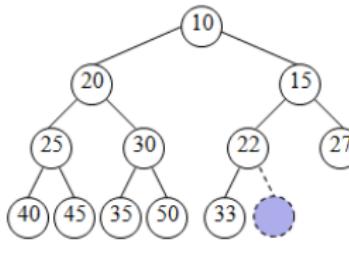
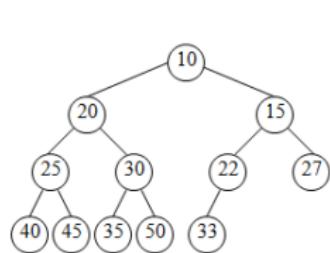
añadir hueco (m.tamaño++)

$\Rightarrow \Rightarrow$

$5 < 22$ (intercambio)

$\Rightarrow \Rightarrow$

$5 < 15$ (intercambio)

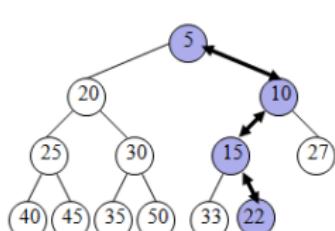
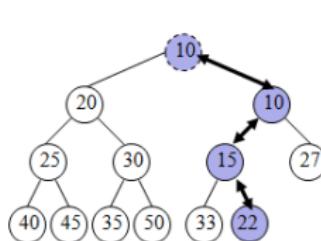
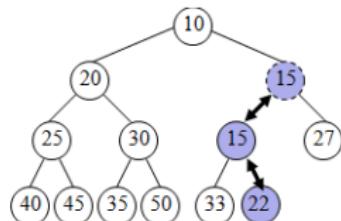


$\Rightarrow 5 < 10$ (intercambio)

$\Rightarrow \Rightarrow$

nuevo mínimo (¡¡¡ No siempre!!!)

\Rightarrow



Algoritmo de INSERCIÓN

Algoritmo inserta(*x:tipoElemento, referencia m:tipoMontículo*)

Entrada: *x* elemento a insertar**Salida:** el montículo *m* con el nuevo elemento *x*

```
1: hueco : entero
2: si m.tamaño  $\geq$  MAX entonces
3:   implementar según especificación y diseño
4: si no
5:   m.tamaño  $\leftarrow$  m.tamaño + 1
6:   hueco  $\leftarrow$  m.tamaño
7:   mientras m.elementos[hueco ÷ 2].clave > x.clave hacer
8:     m.elementos[hueco]  $\leftarrow$  m.elementos[hueco ÷ 2]
9:     hueco  $\leftarrow$  hueco  $\div$  2
10:  fin mientras
11:  m.elementos[hueco]  $\leftarrow$  x
12: fin si
```

->"montículo lleno"

Observaciones

- Si el elemento a insertar es el nuevo mínimo, el hueco debe subir hasta la raíz. En el momento que i tome el valor 1 habrá que romper el ciclo mientras
- Dos soluciones:
 - ① Comprobación explícita en la condición del bucle ($i \neq 1$ AND ...)
 - ② Asignar un valor (centinela) en la posición 0 del array (menor o igual que cualquier elemento del montículo) que asegure que el bucle termina
- La operación insertar es $O(\log n)$ ¿Puedes justificar esta afirmación?

Eliminación del elemento mínimo del montículo

- Eliminar del montículo el elemento que contiene la clave mínima manteniendo las propiedades de estructura y orden que los definen.
- Proceso:
 - ① Se elimina el valor del elemento con clave mínima creando un hueco en la raíz
 - ② Se reduce el tamaño del montículo en una unidad.
 - ③ El que era ultimo elemento debe moverse a otro lugar dentro del montículo conservando la propiedad de orden.
 - ④ Distinguir los siguientes casos:
 - Si puede asignarse al hueco manteniendo la propiedad de orden del montículo, se asigna y finaliza el proceso
 - En otro caso se intercambia el hueco con el menor de sus dos hijos, empujando el hueco hacia abajo un nivel, hasta poder asignar el último elemento al hueco

Estrategia de Filtrado Descendente

Se comparan la clave de un elemento con la del menor de sus dos hijos...

Ejemplo filtrado descendente

eliminar el elemento mínimo del montículo

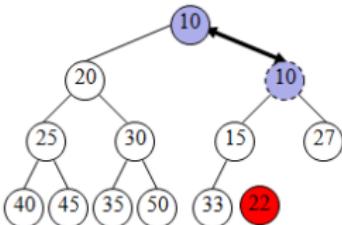
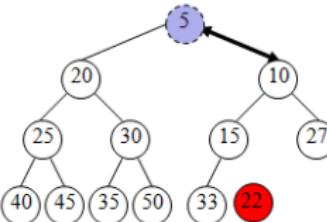
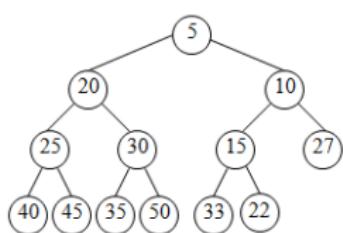
hueco en la raíz (m.tamaño→)

⇒ ⇒

recolocar último elemento

⇒ ⇒

$22 > 10$ (intercambio)

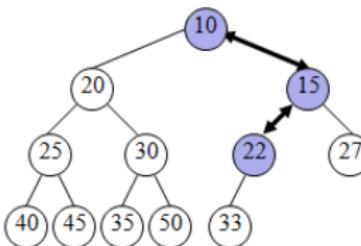
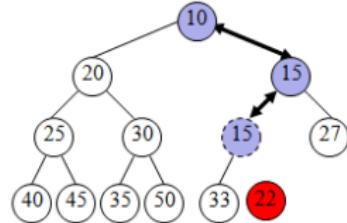


⇒

$22 > 15$ (intercambio)

⇒ ⇒

$22 < 33$ (iii Fin filtrado!!!)



Algoritmo de ELIMINACIÓN

Utiliza estrategia de filtrado descendente

Algoritmo eliminarMin(referencia m:tipoMontículo)

Entrada: el montículo *m*

Salida: el elemento mínimo y el montículo *m* actualizado

```
1: i, hijo : entero
2: mínimo : tipoElemento
3: si vacío(m) entonces
4:   implementar según especificación y diseño
      ->"montículo vacío"
5: si no
6:   mínimo ← m.elementos[1]
7:   último ← m.elementos[m.tamaño]
8:   m.tamaño ← m.tamaño – 1
9:   hueco ← 1
10:  finFiltrado ← FALSO
11:  mientras ( $2 * \text{hueco} \leq \text{m.tamaño}$  Y NO finFiltrado) hacer
12:    hijo ←  $2 * \text{hueco}$ 
13:    si hijo ≠ m.tamaño entonces
14:      si m.elementos[hijo + 1].clave < m.elementos[hijo].clave entonces
15:        hijo ← hijo + 1
16:      fin si
17:    fin si
18:    si m.elementos[hijo].clave < último.clave entonces
19:      m.elementos[hueco] ← m.elemento[hijo]
20:      hueco ← hijo
21:    si no
22:      finFiltrado ← VERDADERO
23:    fin si
24:  fin mientras
25:  m.elementos[hueco] ← último
26:  devolver mínimo
27: fin si
```

Observaciones

- En el filtrado descendente se compara la clave del último elemento con la del menor de los hijos del hueco. Debe tenerse en cuenta que el último nodo interno puede tener un único hijo
- ¿De qué orden es la operación **eliminarMin**?
- Las operaciones **decrementarClave** e **incrementarClave** necesitarán las estrategias de filtrado ascendente y descendente, respectivamente, para mantener la propiedad de orden si esta se pierde
- Dos soluciones para la operación **construirMontículo** que toma como entrada n claves y las coloca en un montículo vacío

Solución 1: Realizar n operaciones insertar sucesivas $\Rightarrow O(n \log n)$

Solución 2: Existe otra solución de $O(n)$ (Ver ejemplo)

Algoritmo Construir Montículo (solución 2)

- Colocar las n claves dentro del árbol en cualquier orden, manteniendo la propiedad de la estructura
- Realizar un filtrado descendente de todos los nodos internos para conseguir la propiedad de orden y obtener un montículo

Algoritmo construirMontículo(**referencia m:tipoMontículo**)

Entrada: montículo m con los n elementos asignados respectando la propiedad de la estructura

Salida: montículo m respetando tanto la propiedad de la estructura como la de orden

- 1: $i, n : \text{entero}$
 - 2: $n \leftarrow m.\text{tamaño}$
 - 3: **para** $i \leftarrow n \div 2$ **bajando hasta 1 hacer**
 - 4: **filtradoDescendente**(m, i)
 - 5: **fin para**
-

En la línea 4 se invoca a una función **genérica** que realiza el **filtrado descendente** del elemento de la posición i en el montículo m

Análisis del Algoritmo Construir Montículo

Solución 2

- Se realiza el filtrado descendente de todos los nodos internos del montículo binario
- En el peor de los casos, cada uno de estos nodos debe descender desde su nivel a un nivel hoja, es decir, como máximo su altura
- Un árbol binario **completo** de altura h tiene

<i>nodos</i>	<i>altura</i>	<i>patrón</i>
1	h	$n_0 = 1 = 2^{h-h}$
2	$h - 1$	$n_1 = 2 = 2^{h-(h-1)}$
2^2	$h - 2$	$n_2 = 4 = 2^{h-(h-2)}$
...	...	
2^{h-1}	1	$n_{(i-1)} = 2^{h-(1)}$
2^h	0	$n_i = 2^{h-(0)}$

- La suma de las alturas de todos los nodos internos de un árbol binario completo (peor de los casos) es

$$1 \cdot h + 2 \cdot (h - 1) + 4 \cdot (h - 2) + \dots + 2^{h-1} \cdot 1$$

- Esta suma indica el número de veces que se ejecuta la instrucción barómetro y, por tanto, el orden del algoritmo

Análisis del Algoritmo Construir Montículo

- Calculamos la suma multiplicando por dos la ecuación anterior y restando (algunos términos se cancelan)

$$\begin{array}{rcl} 2S & = & 2h + 4(h-1) + 8(h-2) + \dots + 2^h \\ S & = & h + 2(h-1) + 4(h-2) + 8(h-3) + \dots \\ \hline \end{array}$$

$$S = -h + 2 + 4 + 8 + \dots + 2^h$$

$$t(n) = -h + 2 + 4 + 8 + \dots + 2^h = -h + 2^{h+1} - 2$$

- Teniendo en cuenta que un árbol casi-completo tiene entre 2^h y 2^{h+1} nodos (transparencia 11, tema 1), la suma que obtenemos es de orden $O(n)$ siendo n el número de nodos

$$t(n) = 2^{h+1} - 2 - h = n - (2 + h) \in O(n)$$

Contenido

2 Tema 2. Montículos Binarios

- Introducción
- Nivel abstracto o de definición
- Nivel de representación o implementación
- Ordenación por montículos
- Ejercicios

4 Ordenación por montículos

- Los montículos se pueden utilizar como método de ordenación interna con un comportamiento $O(n \log n)$
- El algoritmo basado en esta idea se denomina ordenación por montículo (*heapsort*) y se basa en:
 - **Construir** un montículo binario de n elementos
 - Efectuar n operaciones **eliminarMin**: los elementos más pequeños dejan primero el montículo
 - Cada operación eliminarMin contrae el montículo en uno. Si se aprovecha la que era última celda del montículo para almacenar el elemento eliminado, al final de la operación las n celdas del array contienen los elementos en orden decreciente

Análisis de algoritmo de ordenación por montículos

- Crear montículo requiere un tiempo lineal, $O(n)$, y genera un montículo de altura $\lceil \log n \rceil$
- La operación eliminarMin se ejecutará $(n - 1)$ veces y filtra la raíz a lo largo de un camino de longitud máxima $\log n$, por tanto, en el peor caso requiere un tiempo que es $O(\log n)$
- Habrá también $n - 1$ operaciones de asignación que requieren un tiempo constante, $O(1)$
- Por tanto el tiempo, $t(n)$, necesario para ordenar los n elementos verifica

$$t(n) \in O(n) + (n - 1)\log(n) + (n - 1)O(1) \in \mathbf{O(n\log n)}$$

Observaciones

- Para obtener los elementos en orden creciente, basta con cambiar la propiedad de orden de forma que el padre tenga una clave mayor que los hijos y considerar la operación eliminarMax

Definición alternativa de Montículo Binario

Árbol binario casi-completo en el cual, para todo nodo n se cumple que, la clave en el padre de n es **mayor** (o igual) que la clave de n, con la excepción obvia de la raíz”

- Cambia la propiedad de orden del montículo \Rightarrow el elemento con valor **máximo** siempre se encuentra en la raíz

Contenido

2 Tema 2. Montículos Binarios

- Introducción
- Nivel abstracto o de definición
- Nivel de representación o implementación
- Ordenación por montículos
- Ejercicios

Ejercicios

Ejercicio 1: Analizar qué ocurre en el montículo de la figura 1 en los siguientes casos:

- ① Cuando se inserta un nuevo nodo con valor 9 para su campo clave
- ② Cuando se elimina el elemento mínimo
- ③ Cuando se incrementa en 2 la clave del elemento situado en la posición 10
- ④ Cuando se decremente en 40 la clave del elemento situado en la posición 23

Ejercicio 2: Dado el montículo binario de la figura 2 dibujar el montículo, explicando brevemente el proceso, después de

- ① Realizar una operación que decremente en 620 el valor del elemento de la posición 18.
- ② Realizar una operación que incremente en 400 el valor del elemento de la posición 17.

Figura 1

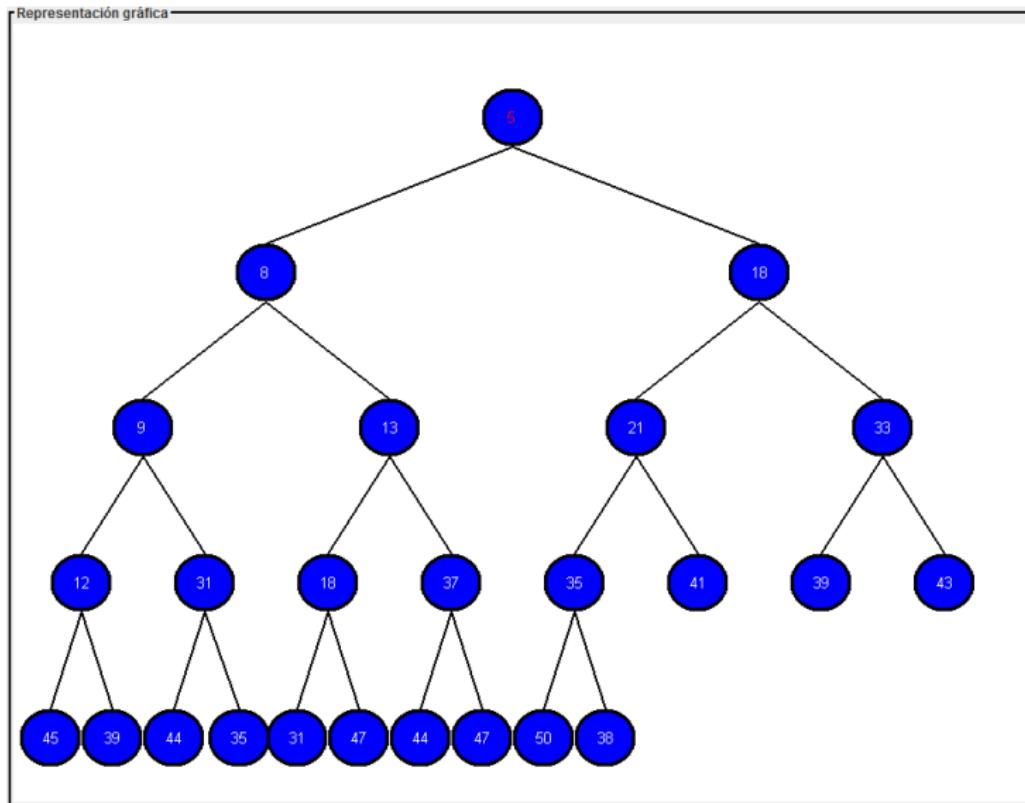
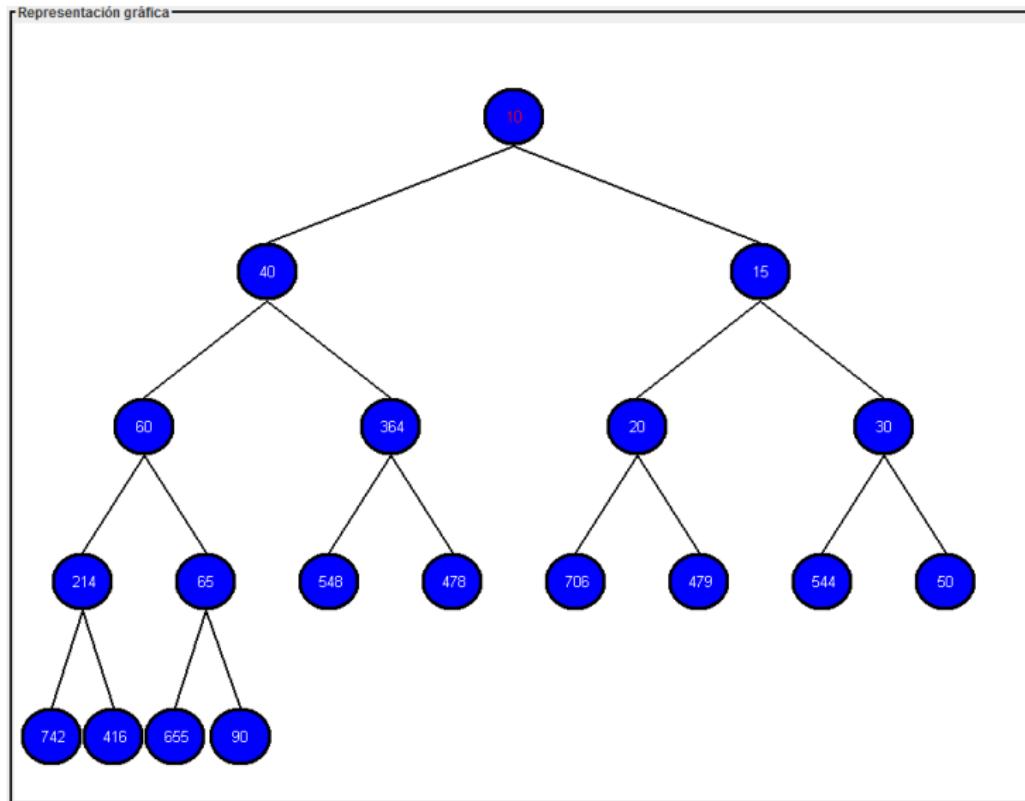


Figura 2



Ejercicio 3

Considerando una montículo binario en el que se han introducido aleatoriamente 10 claves manteniendo la propiedad de la estructura.

- ① ¿Cuántas veces es necesario aplicar el algoritmo de filtrado descendente para conseguir la propiedad de orden?
- ② Suponiendo que inicialmente el contenido del array que representa el montículo es el que muestra la figura, aplicar el algoritmo de filtrado descendente las veces necesarias para completar la tabla de forma que muestre el contenido del array después de cada filtrado.

Contenido inicial →

?	99	35	127	191	198	304	54	107	37	115
---	----	----	-----	-----	-----	-----	----	-----	----	-----

--	--	--	--	--	--	--	--	--	--	--

--	--	--	--	--	--	--	--	--	--	--

--	--	--	--	--	--	--	--	--	--	--

--	--	--	--	--	--	--	--	--	--	--

--	--	--	--	--	--	--	--	--	--	--

Ejercicio 4

Utilizando la aplicación RAED estudiar y analizar los procesos de filtrado ascendente y descendente aplicando las operaciones típicas de inserción y eliminación a diferentes ejemplos de montículos binarios.

Nota: En el apartado dedicado a la aplicación RAED se proporcionan dos ficheros con los montículos correspondientes a los ejercicios 1 y 2.

Contenido

- 1 Tema1. Árboles Generales y Binarios
- 2 Tema 2. Montículos Binarios
- 3 Tema 3. Conjuntos Disjuntos
- 4 Tema 4. Grafos
- 5 Tema 5. Árboles Binarios de Búsqueda
- 6 Tema 6. Organización de archivos
- 7 Tema 7. Organización de Índices

Contenido

3 Tema 3. Conjuntos Disjuntos

- Relación de Equivalencia
- Nivel abstracto o de definición
- Nivel de representación o implementación
- Compresión de caminos

1 Relación de Equivalencia

- Concepto matemático definido sobre un conjunto basado en la idea de representación de relaciones entre sus elementos (ciudades en una misma comunidad, colores en una imagen, ...)

Definición

Se define una **relación de equivalencia** R sobre un conjunto U si para todo par de elementos a, b pertenecientes a U , $a R b$ es verdadera o falsa

Propiedades

- 1 **Reflexiva.** $\forall a \in U, a R a$
- 2 **Simétrica.** $\forall a, b \in U, a R b \text{ si y sólo si } b R a$
- 3 **Transitiva.** $\forall a, b, c \in U, \text{ Si } a R b \text{ y } b R c \Rightarrow a R c$

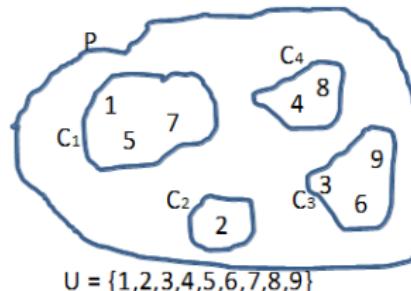
Clase de equivalencia de un elemento $a \in U$

Subconjunto de U que tiene todos los elementos relacionados con a .

Partición de U

Conjunto de todas las clases de equivalencia definidas sobre U según la relación R . Es decir, conjunto de **subconjuntos disjuntos** cuya unión es el conjunto U .

- Todo elemento de U aparece exactamente en una clase de equivalencia.
- $a R b \Rightarrow a$ y b están en la misma clase de equivalencia



Aplicaciones

- Procesamiento de imágenes digitales
 - Imagen en color o en escala de grises compuesta por una matriz de pixels
 - Clases de equivalencia: regiones continuas y del mismo color
 - Relación de equivalencia: dos pixels a y b están relacionados si tienen el mismo color y son adyacentes en la imagen
 - Operación rellenar una región de color:
 - Buscar la clase de equivalencia del punto sobre el que se aplica el relleno
 - Después de la operación puede haber cambio en las clases de equivalencia
- **Tema 4. Grafos**
 - Algoritmo de Kruskal
 - Estudio de la conectividad

Contenido

3 Tema 3. Conjuntos Disjuntos

- Relación de Equivalencia
- Nivel abstracto o de definición
- Nivel de representación o implementación
- Compresión de caminos

2 Nivel abstracto o de definición

Partición de U

Una estructura de datos **Partición** es la colección de conjuntos disjuntos (disjoint-set) entre sí que forman las clases de equivalencia, según alguna relación de equivalencia R, con los elementos de un cierto conjunto universal $U = \{x_1, x_2, \dots, x_n\}$

$$P = \{C_1, C_2, \dots, C_k\} \quad \forall i, j \in \{1, 2, \dots, k\} \mid i \neq j \Rightarrow C_i \cap C_j = \emptyset$$

Cada conjunto C_i representa una **clase de equivalencia** según alguna **relación R**

Operaciones

- Creación inicial de las clases de equivalencia (conjuntos C_1, C_2, \dots, C_n)
- Añadir una relación de equivalencia entre dos elementos no relacionados:
 - si x_i y x_j no están en la misma clase de equivalencia (pertenecen a conjuntos C_i y C_j diferentes) se combinan las dos clases de equivalencia en una clase de equivalencia nueva, preservando que todos los conjuntos de la partición son disjuntos
- Encontrar a que clase de equivalencia pertenece un elemento x_i de U

Observaciones

- No se realizan operaciones comparando los valores relativos de los elementos, solo se requiere conocer su localización, su clase de equivalencia
- La operación búsqueda devuelve el nombre de la clase de equivalencia o conjunto al que pertenece un elemento:
 - Este nombre es bastante arbitrario
 - Es importante que devuelva el mismo nombre para todos los elementos que pertenecen a la misma clase de equivalencia
 - Puede elegirse como representante un miembro cualquiera del conjunto
- La operación que crea una nueva relación de equivalencia a partir de otras dos es dinámica, pues durante el proceso los conjuntos cambian

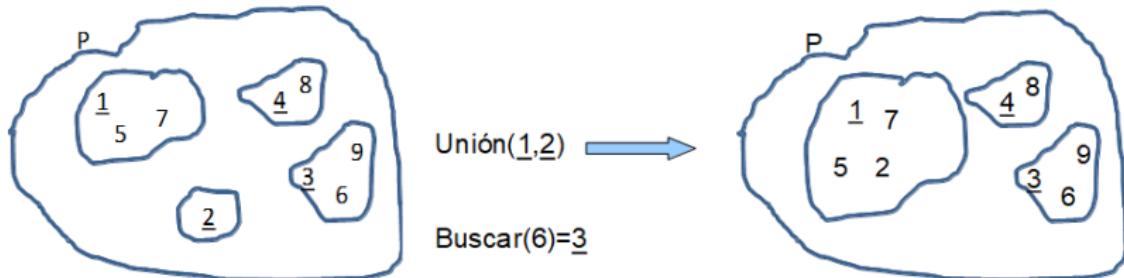
Situación inicial

Todas las relaciones son falsas, excepto las reflexivas. Colección de n conjuntos, cada uno con un elemento diferente

$$n \text{ conjuntos disjuntos: } C_i \cap C_j = \emptyset$$

Especificación de Operaciones

- **crear(x_i)**: Crea un nuevo conjunto cuyo único miembro es x_i . Se requiere que x_i no esté en ningún otro conjunto de la estructura
- **buscar(x_i)**: Devuelve la clase de equivalencia a la que pertenece x_i , es decir, el nombre del conjunto disjunto, que será uno cualquiera de sus elementos, el elegido como su representante
- **unión(x_i, x_j)**: Establece una relación de equivalencia entre los elementos x_i y x_j . Es necesaria la unión de las clases de equivalencia a que pertenecen x_i y x_j , sean C_i y C_j , formando una nueva. Se requiere la eliminación de los conjuntos C_i y C_j , para que la colección sea disjunta



Contenido

3 Tema 3. Conjuntos Disjuntos

- Relación de Equivalencia
- Nivel abstracto o de definición
- Nivel de representación o implementación
- Compresión de caminos

3 Nivel de representación o implementación

- Simplificamos el problema suponiendo que los elementos del conjunto U, sobre el que se definen las relaciones de equivalencia, están numerados de 1 a N. Si no es así habrá que definir una función biyectiva que realice la traducción entre U y el conjunto $\{1, 2, \dots, N\}$

$$U = \{1, 2, \dots, N\}$$

- **Nivel de Representación**

- ① Mediante matrices
- ② Mediante listas
- ③ Mediante árboles

3.1 Representación mediante MATRICES

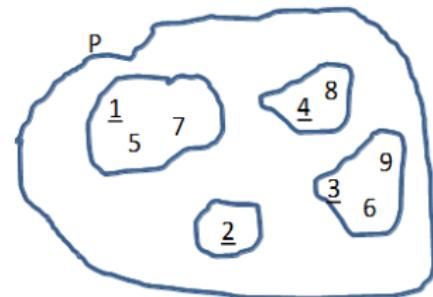
- La partición se representa con una matriz unidimensional de tamaño N donde:
 - los índices representan los elementos del conjunto U
 - cada celda almacena el nombre de la clase de equivalencia del elemento correspondiente

Algoritmo declaraciones básicas

```

1: constantes
2:    $N = 100$ 
3: tipos
4:   tipoElemento = entero
5:   tipoConjunto = entero
6:   tipoPartición = matriz[1, ..., N] de tipoConjunto
7: tipos

```



Variable P de tipo partición

1	2	3	4	5	6	7	8	9
1	2	3	4	1	3	1	4	3

Operaciones

- La operación **crear(x)** que inicia la estructura a una situación inicial donde todas las relaciones son falsas excepto las reflexivas toma un tiempo de $O(N)$, pero solo se aplicará una vez
- La operación **buscar(x)** devuelve el contenido de la celda **x** del array, está claramente en un $O(1)$
- La operación **unión(x,y)** es $O(N)$: si **x** está en clase C_i e **y** en clase C_j , se recorre la matriz cambiando todo i a j . Esta operación es previsible que aparezca con bastante frecuencia y el coste puede ser excesivo para muchas aplicaciones

La **implementación** de todas estas operaciones es **inmediata** y se deja como **ejercicio**

Análisis de una secuencia arbitraria de operaciones búsqueda/unión

- No sabemos el orden en que se van a presentar estas operaciones, pero no habrá más de $N - 1$ uniones, ya que entonces todos los objetos están en el mismo conjunto
- Una secuencia de $N - 1$ uniones puede requerir un tiempo en $O(N^2)$
- Si hay n operaciones de búsqueda requerirán un tiempo $O(n)$
- Si n y N son comparables (ocurre en muchas aplicaciones), la secuencia de operaciones requiere un tiempo que se encuentra en $O(N^2)$
- Procesamiento imágenes:
 - Se aplicará unión si un píxel tiene al lado otro del mismo color (bastante frecuente)
 - Aplicar unión para cada píxel de una imagen de $800 \times 600 \Rightarrow (800 * 600)^2 \approx 1.230$ mil millones de comparaciones!
- Coste excesivo para muchas aplicaciones

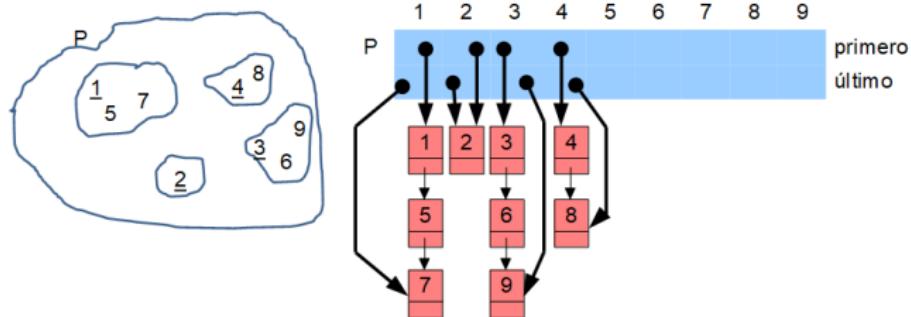
3.2 Representación mediante LISTAS

- Cada clase de equivalencia se representa mediante una lista que contiene sus elementos
- Para la unión necesitamos una estructura que permita concatenar dos listas de forma rápida
 - ① Mantener en la cabecera de las listas dos punteros, uno al primer elemento y otro al último
 - ② Listas circulares circulares y doblemente enlazadas

Algoritmo declaraciones básicas

```
1: constantes
2:    $N = 100$ 
3: tipos
4:   tipoElemento = entero
5:   tipoConjunto = entero
6:   tipoPartición = matriz[1, ...,  $N$ ] de tipoLista
7: fin tipos
```

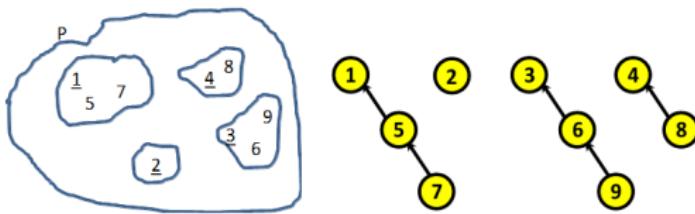
Ejemplo



- La operación unión puede conseguirse ahora en un tiempo constante
 - La operación buscar, sin embargo, debe recorrer todas las listas hasta encontrar el elemento en una de ellas. En el peor de los casos todos los elementos de todas las listas. En promedio la mitad de las listas, siendo de $O(N)$
 - Si este comportamiento no era bueno para la unión en la representación anterior, puede ser aún peor para la búsqueda, si se necesita frecuentemente
- La **implementación** de todas estas operaciones es **inmediata** y se deja como **ejercicio**

3.3 Representación mediante ÁRBOLES

- Las dos representaciones anteriores utilizan estructuras lineales dando lugar a tiempos lineales, bien para la búsqueda bien para la unión. Alternativa, utilizar estructuras no lineales, árboles
- **Cada clase de equivalencia se representa por un árbol utilizando su raíz para nombrar al conjunto**
- La unión puede conseguirse en un tiempo constante, colocando un árbol como subárbol del otro
- Para la búsqueda es necesario, dado un elemento del árbol, encontrar cual es la raíz: **representación de árboles con punteros al nodo padre**
- Se puede seguir utilizando la representación mediante una matriz, donde cada entrada almacena el padre del elemento i
- La estructura no almacena un sólo árbol, sino un conjunto de árboles: **bosque** de relaciones de equivalencia



Algoritmo declaraciones básicas

```

1: constantes
2: N = 100
3: tipos
4: tipoElemento = entero
5: tipoConjunto = entero
6: tipoPartición = matriz[1, ..., N] de tipoConjunto
7: tipos

```

Variable P de tipo partición

1	2	3	4	5	6	7	8	9
?	?	?	?	1	3	5	4	6

Convenio para la representación

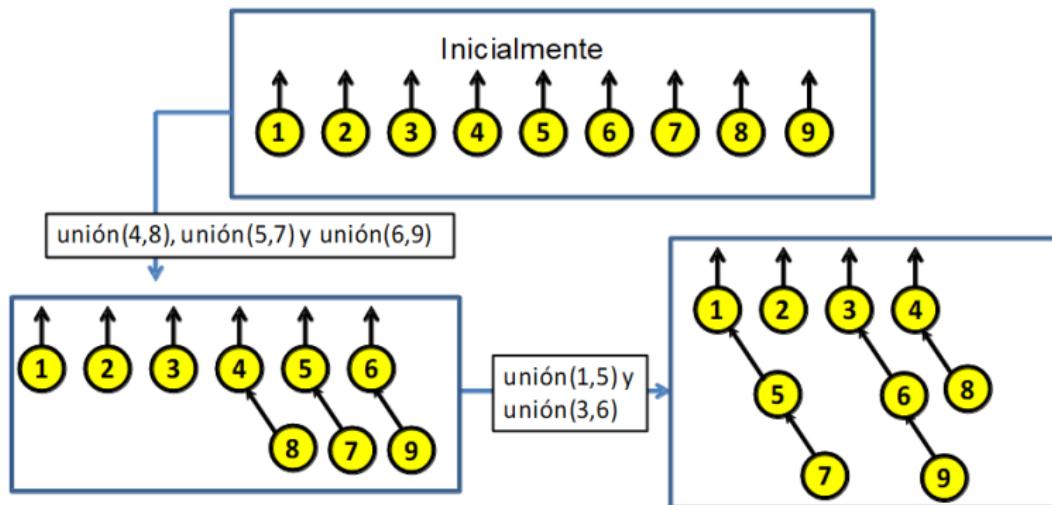
- Si $C[i]=0$ entonces i es a la vez el nombre del conjunto y su raíz
- Si $C[i]=j \neq 0$ entonces j es el padre de i en algún árbol

Variable P de tipo partición

1	2	3	4	5	6	7	8	9
0	0	0	0	1	3	5	4	6

Operación Unión

- Se combinan dos árboles haciendo que la raíz de uno apunte a la raíz del otro
- Convenio (arbitrario) en $\text{unión}(x,y)$ la nueva raíz es x



Operaciones

Algoritmo crear(**ref P**: tipoPartición)

Entrada: partición P con valores desconocidos (no representa nada)

Salida: partición P con valores que representan la situación inicial

```
1: i : tipoElemento  
2: para i  $\leftarrow 1$  hasta N hacer  
3:   P[i]  $\leftarrow 0$   
4: fin para
```

Algoritmo buscar(*x*:tipoElemento, *P*: tipoPartición):tipoConjunto

Entrada: partición P y elemento x

Salida: clase de equivalencia a la que pertenece x

```
1: si P[x] = 0 entonces  
2:   devolver x  
3: si no  
4:   devolver buscar(P[x], P)  
5: fin si
```

Algoritmo union(**raíz1,raíz2**:tipoConjunto, **ref P**: tipoPartición) **!!! ojo !!!**

Entrada: representantes de los elementos que se quieren relacionar y partición P que representa las clases de equivalencia actuales

Salida: partición P con las nuevas clases de equivalencia

```
1: P[raíz2]  $\leftarrow$  raíz1
```

Análisis de una secuencia arbitraria de operaciones búsqueda/unión

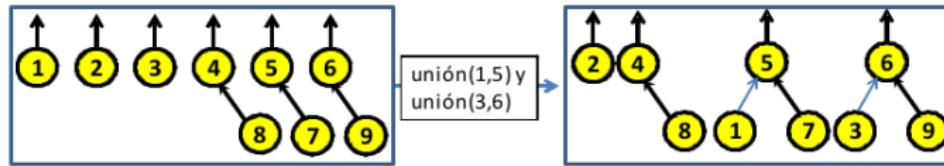
- La operación unión ahora toma un tiempo constante($O(1)$)
- La operación búsqueda devuelve la raíz del árbol del elemento buscado
 - El tiempo de ejecución es proporcional a la profundidad del nodo
 - Con la estrategia utilizada en la unión puede crearse un árbol de profundidad $N - 1$. Por tanto, en el peor de los casos el tiempo de búsqueda es $O(N)$
- Una secuencia arbitraria de n operaciones de búsqueda y $N - 1$ operaciones unión, en el caso peor, requerirá un tiempo en $O(nN)$, que será $O(N^2)$ si n es comparable a N
- ¡No hemos ganado nada con respecto a las representaciones anteriores!

Mejora

Eliminar la arbitrariedad de la operación unión haciendo siempre que el subárbol menor sea un subárbol del mayor

Mejoras en la operación unión

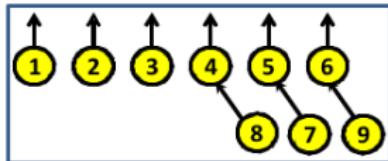
- Dos alternativas:
 - Unión por tamaño: el árbol de menor tamaño se hace subárbol del mayor
 - Unión por altura: el árbol de menor altura se hace subárbol del más alto



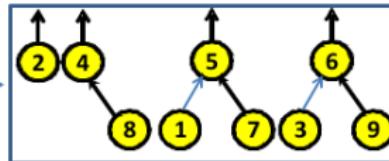
Unión por tamaño

Convenio para la representación

- Si i es la raíz del árbol $C[i]$ contiene el tamaño del árbol que representa en negativo
- Si $C[i]=j > 0$ entonces j es el padre de i en algún árbol



unión(1,5) y
unión(3,6)



Variable P de tipo partición

1	2	3	4	5	6	7	8	9
-1	-1	-1	-2	-2	-2	5	4	6

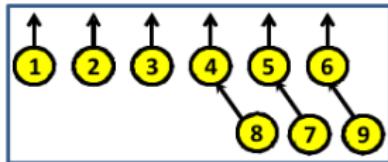
Variable P de tipo partición

1	2	3	4	5	6	7	8	9
5	-1	6	-2	-3	-3	5	4	6

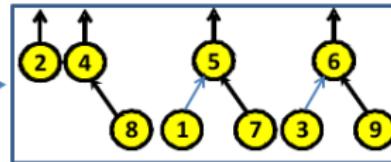
Unión por altura

Convenio para la representación

- Si i es la raíz del árbol $C[i]$ contiene la altura del del árbol que representa en negativo
- Si $C[i]=j > 0$ entonces j es el padre de i en algún árbol



unión(1,5) y
unión(3,6)



Variable P de tipo partición

1	2	3	4	5	6	7	8	9
0	0	0	-1	-1	-1	5	4	6

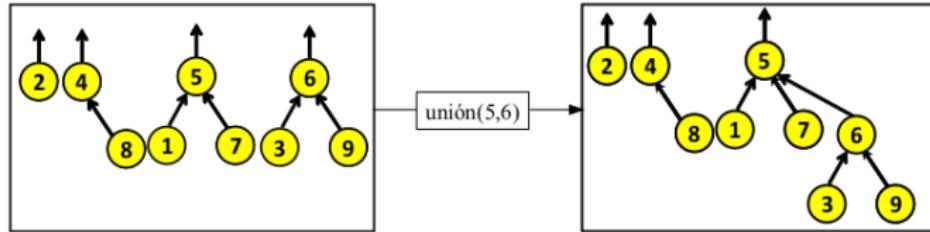
Variable P de tipo partición

1	2	3	4	5	6	7	8	9
5	0	6	-1	-1	-1	5	4	6

Análisis unión por altura

- Si se fusionan dos árboles de alturas respectivas h_1 y h_2 , la altura del árbol resultante será:

$$h = \begin{cases} \max(h_1, h_2) & \text{si } h_1 \neq h_2 \\ h_1 + 1 & \text{si } h_1 = h_2 \end{cases}$$



Teorema

Empleando la técnica de unión por altura, al cabo de una secuencia arbitraria de operaciones unión, que comienzan en la situación inicial, un árbol de k nodos tiene una altura que es como máximo $\lceil \lg k \rceil$

Demostración por inducción:

- ① Verdadero para el caso base $\Rightarrow k = 1 \Rightarrow h = 0 \leq \lceil \lg 1 \rceil$
- ② Hipótesis de inducción. Teorema cierto para todo m menor que k

$$m/1 \leq m < k \Rightarrow h \leq \lceil \lg m \rceil$$

- ③ Demostrar que es cierto para k . Un árbol de k nodos se obtiene de otros dos más pequeños con a y b nodos, donde:
 - sin pérdida de generalidad suponemos $a \leq b$
 - $a \geq 1$ (partimos de la situación inicial)
 - $k = a + b$

Demostración teorema

$$\left. \begin{array}{l} a \leq b \\ a + b = k \end{array} \right\} \Rightarrow a \leq k - a \Rightarrow a \leq \frac{k}{2} \Rightarrow \underbrace{a < k}_{k>1 \Rightarrow \frac{k}{2} < k} \xrightarrow{\text{Hipótesis de inducción}} h_a \leq [\lg a]$$

$$\left. \begin{array}{l} a \geq 1 \\ a + b = k \end{array} \right\} \Rightarrow b \leq k - 1 \Rightarrow \underbrace{b < k}_{k>1 \Rightarrow k-1 < k} \xrightarrow{\text{Hipótesis de inducción}} h_b \leq [\lg b]$$

Sea h_k la altura del árbol resultado de la unión $\Rightarrow h_k = \begin{cases} \max(h_a, h_b) & \text{si } h_a \neq h_b \\ h_a + 1 & \text{si } h_a = h_b \end{cases}$

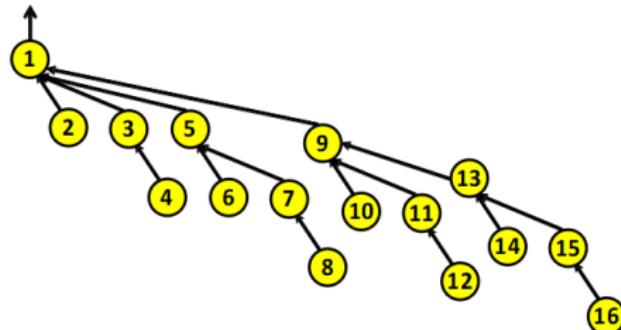
$h_k \leq [\lg k]?$

Se pueden dar dos casos

- ① Si $h_a \neq h_b \Rightarrow h_k = \max(h_a, h_b) \leq \max([\lg a], [\lg b]) \leq [\lg k]$
- ② Si $h_a = h_b \Rightarrow h_k = h_a + 1 \leq [\lg a] + 1 \leq [\lg \frac{k}{2}] + 1 \leq [\lg k] - 1 + 1 \leq [\lg k]$

Análisis

- Si cada consulta o modificación de un elemento de la matriz cuenta como operación elemental, la operación unión sigue tomando tiempo constante
- La operación búsqueda, ahora es $O(\lg N)$
- Una secuencia arbitraria de n operaciones de búsqueda y $N - 1$ operaciones unión, a partir de la situación inicial, en el caso peor requerirá un tiempo en $O(N + n \lg N)$, que será $O(n \lg n)$ si n es comparable a N
- Árbol del peor caso para $N = 16$



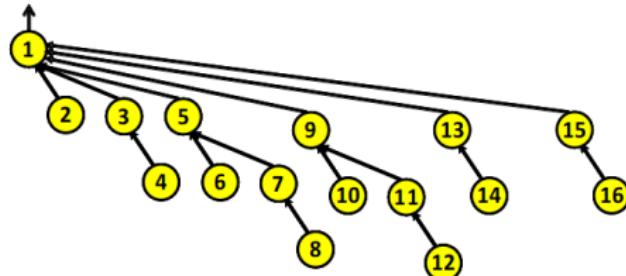
Contenido

3 Tema 3. Conjuntos Disjuntos

- Relación de Equivalencia
- Nivel abstracto o de definición
- Nivel de representación o implementación
- Compresión de caminos

4 COMPRESIÓN DE CAMINOS

- Técnica que se aplica a la operación de búsqueda para tratar de conseguir que las operaciones sean más rápidas
- Para determinar que conjunto contiene a cierto elemento x , la operación de búsqueda sube desde el nodo que contiene a x hasta la raíz del árbol
- La **compresión de caminos**, una vez que se conoce la raíz, vuelve a recorrer este camino, modificando el padre de cada nodo del camino, para que sea directamente la raíz
- Compresión de caminos después de **buscar(15)**



Estrategia

- Se ejecuta durante la operación **buscar(x)** y es independiente de la estrategia utilizada en la unión
- Todo nodo en el camino de **x** a la raíz cambia su padre por la raíz
- Accesos futuros más rápidos a cambio de algunos movimientos de punteros adicionales

Algoritmo **buscar(x:tipoElemento, ref P: tipoPartición):tipoConjunto**

Entrada: partición **P** y elemento **x**

Salida: clase de equivalencia a la que pertenece **x** modificada, el padre de cada nodo del camino en el retorno pasa a ser directamente la raíz

```
1: si P[x] <= 0 entonces
2:   devolver x
3: si no
4:   P[x] ← buscar(P[x], P)
5:   devolver P[x]
6: fin si
```

Observaciones

- La implantación de la compresión de caminos se hace con un cambio trivial en el algoritmo básico de búsqueda: asignación recursiva a $P[x]$ del valor que devuelve buscar
- La compresión de caminos tiende a reducir la altura del árbol y por tanto a conseguir que las operaciones buscar subsiguientes sean más rápidas
- La nueva operación de búsqueda, sin embargo, recorre dos veces el camino que va desde el nodo en cuestión hasta la raíz. Requiere aproximadamente el doble de tiempo
- Si pocas operaciones de búsqueda puede que no merezca la pena
- Si operaciones de búsqueda frecuentes, al cabo de un tiempo, todos los nodos implicados quedan asociados con su raíz, y las operaciones buscar subsiguientes tomarán un tiempo constante
- La unión perturbará ligeramente esta situación y no durante mucho tiempo
- La mayor parte del tiempo, tanto las operaciones de búsqueda como unión requerirán un tiempo constante

Observaciones

- La compresión de caminos es perfectamente compatible con la unión por tamaños, y así ambas rutinas pueden implantarse a la vez
- La compresión de caminos no es del todo compatible con la unión por altura, porque la compresión puede cambiar las alturas de los árboles:
 - No está muy claro como volver a calcularlas con eficiencia
 - Solución, no se calculan
 - Las alturas que se almacenan para cada árbol se convierten en alturas estimadas o **rangos**
 - La unión por altura se denomina entonces **unión por rango**
- El tiempo promedio de la operación buscar incluyendo compresión de caminos y unión por tamaño o altura, resulta difícil de calcular
- Intuitivamente puede observarse que será un tiempo mucho menor que logarítmico: será difícil obtener un árbol de profundidad 5 teniendo en cuenta que se requiere la unión de dos árboles de profundidad 4

Coste utilizando unión por rango y compresión de caminos

- Utilizando ambas heurísticas, el coste en el caso peor para una secuencia de $m(n + N - 1)$ operaciones es $O(m\alpha(m, N))$, donde $\alpha(m, N)$ es una función (inversa de la función de Ackerman) que crece muy despacio (*)
- Tan despacio que en cualquier situación práctica que podamos imaginar se tiene que $\alpha(m, n) \leq 4 \Rightarrow$ coste casi lineal
- La función de Ackerman, definida para enteros $i, j \geq 1$:

$$A(i, j) = 2^j \quad \text{para } j \geq 1$$

$$A(i, 1) = A(i - 1, 2) \quad \text{para } i \geq 2$$

$$A(i, j) = A(i - 1, A(i, j - 1)) \quad \text{para } i, j \geq 2$$

- La función inversa $\alpha(m, n)$:

$$\alpha(m, n) = \min\{i \geq 1 | A(i, [m/N]) > \log N\}$$

(*) Análisis muy complejo demostrado por Robert E. Tarjan en 1975
(versión más simple en 1983)

La función de Ackerman y su inversa $\alpha(m, n)$

- Algunos valores de la función de Ackerman

$$A(1,j) = 2^j, \text{ para } j \geq 1$$

$$A(i,1) = A(i-1,2), \text{ para } i \geq 2$$

$$A(i,j) = A(i-1, A(i,j-1)), \text{ para } i, j \geq 2$$

	$j = 1$	$j = 2$	$j = 3$	$j = 4$
$i = 1$	2^1	2^2	2^3	2^4
$i = 2$	2^2	2^{2^2}	$2^{2^{2^2}}$	$2^{2^{2^{2^2}}}$
$i = 3$	2^{2^2}	$2^{2^{2^2}}$	$2^{2^{2^{2^2}}}$	$2^{2^{2^{2^{2^2}}}}$

- Función “inversa” en cuanto a que crece tan despacio como deprisa lo hace la función de Ackerman

$$\alpha(m, N) = \min\{i \geq 1 \mid A(i, \lfloor m/N \rfloor) > \log N\}$$

$$m \geq N \Rightarrow \left[\frac{m}{N} \right] \geq 1 \Rightarrow A(i, \left[\frac{m}{N} \right]) \geq A(i, 1) \text{ para } i \geq 1$$

$$A(4, \left[m\middle/\cancel{N}\right]) \geq A(4,1) = A(3,2) = 2^{2^2} \Bigg\}^{16 \text{ veces}}$$

- Solo para valores enormes de N ocurrirá $A(4, 1) > \log N$. Por tanto, para valores razonables de m y N , $\alpha(m, n) \geq 4$

Contenido

- 1 Tema1. Árboles Generales y Binarios
- 2 Tema 2. Montículos Binarios
- 3 Tema 3. Conjuntos Disjuntos
- 4 Tema 4. Grafos
- 5 Tema 5. Árboles Binarios de Búsqueda
- 6 Tema 6. Organización de archivos
- 7 Tema 7. Organización de Índices

Contenido

4

Tema 4. Grafos

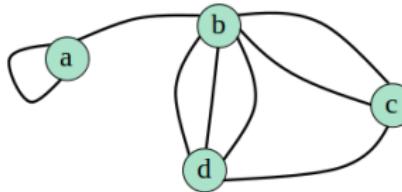
- Nivel Abstracto o de definición
- Nivel de representación o implementación
- Recorridos
- Ordenación Topológica
- Caminos Mínimos
- Árbol de expansión de coste mínimo
- Ejercicios

1 Nivel Abstracto o de definición

- Intuitivamente, un **grafo** es un conjunto de puntos (vértices o nodos) y un conjunto de líneas (aristas o arcos), cada una de las cuales une un punto con otro
- Un grafo está completamente definido por su conjunto de vértices, V ; y su conjunto de aristas, A

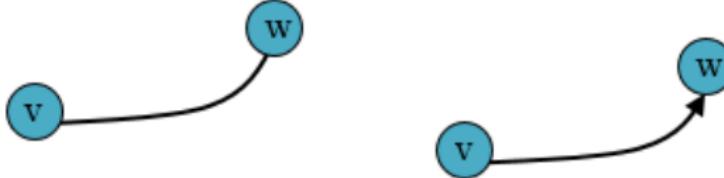
$$G = (V, A)$$

- **Orden** del grafo: número de vértices
- Ejemplo: $G = (V, A)$ donde
 $V = \{a, b, c, d\}$
 $A = \{(a, a), (a, b), (b, c), (b, c), (b, d), (b, d), (b, d), (c, d)\}$



Concepto de Adyacencia

- Cada **arista** es un par (v,w) donde $v, w \in V$
 - Si el par está ordenado \Rightarrow **grafo dirigido o digrafo**
 - Si las aristas tiene un tercer componente (peso o costo) \Rightarrow grafos con aristas ponderadas o **grafos ponderados**
 - Un vértice w es **adyacente** a otro vértice v si y sólo si la arista $(v,w) \in A$
 - En un grafo no dirigido con arista (v,w) , y por tanto, (w,v) , w es adyacente a v y v es adyacente a w
 - En un grafo dirigido con arista (v,w) , w es adyacente a v



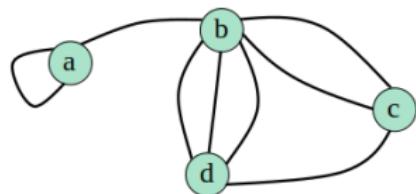
Grafos generales y Grafos dirigidos o Digrafos

• Grafos Generales

- Puede haber varios arcos conectando dos vértices
- Algunos pares de vértices pueden estar desconectados
- Algunos arcos pueden conectar un vértice a sí mismo

Ejemplo: $G = (V, A)$ donde

$$\begin{aligned} V &= \{ a, b, c, d \} \\ A &= \{ (a,a), (a,b), (b,c), (b,c), (b,d), (b,d), (b,d), (c,d) \} \end{aligned}$$

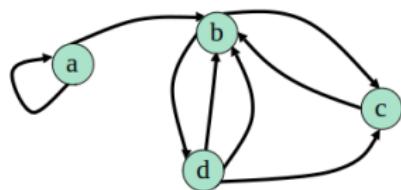


• Grafos Dirigidos o Digrafos

- Se impone un orden o dirección en las aristas del grafo

Ejemplo: $G = (V, A)$ donde

$$\begin{aligned} V &= \{ a, b, c, d \} \\ A &= \{ (a,a), (a,b), (b,c), (c,b), (b,d), (d,b), (d,b), (d,c) \} \end{aligned}$$



Caminos o trayectorias en grafos

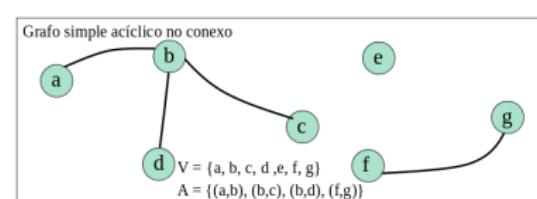
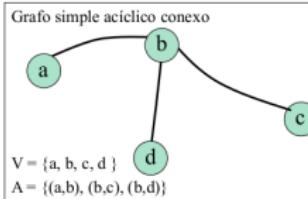
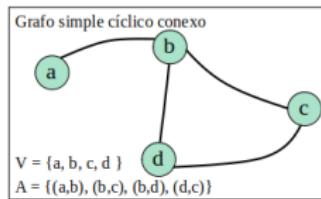
Definición

Un **camino** o **trayectoria** en un grafo es una secuencia de vértices $v_{i(1)}, v_{i(2)}, \dots, v_{i(n)}$ tal que la arista $(v_{i(x)}, v_{i(x+1)}) \in A$ para $1 \leq x < n$

- **Longitud** de camino: número de arcos que lo componen ($n-1$)
- En un grafo se permiten caminos de un vértice a sí mismo:
 - **Caso especial:** un camino de un vértice a sí mismo que no contiene aristas tiene longitud cero
 - Si el grafo contiene una arista (v,v) de un vértice a sí mismo \Rightarrow al camino v,v se le denomina **bucle**
- **Camino simple:** todos los vértices son distintos, excepto el primero y el último que pueden ser el mismo
- **Ciclo:** camino simple donde el vértice inicial y final coinciden

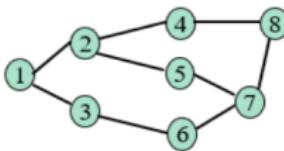
Grafos simples, acíclicos y conexos

- Un grafo $G = (V, A)$ se denomina **grafo simple** si:
 - ① No tiene bucles: no existe en A un arco de la forma (v, v) siendo $v \in V$
 - ② No hay más de una arista uniendo un par de vértices: no existe más de una arista en A de la forma (v_i, v_j) para cualquier par de elementos $v_i, v_j \in V$
- Un grafo G se denomina **acíclico** si no contiene ciclos
- Un grafo G se denomina **conexo** si para cualquier par de vértices (v_i, v_j) en G , existe al menos una trayectoria de v_i a v_j . No puede dividirse en dos sin eliminar uno de sus arcos.
- **Árbol:** grafo conexo, simple y acíclico

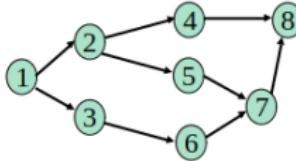


Grado de un vértice

- **Grafo no dirigido:** Número de aristas que confluyen en el vértice



- **Grafo dirigido:** Suma de sus grados interno y externo
 - **Grado interno o de entrada:** número de aristas que terminan en él
 - **Grado externo o de salida:** número de aristas que salen de él



Ejemplos de aplicación

- Aeropuertos
 - Vértices: ciudades
 - Aristas: vuelos aéreos de una ciudad a otra, distancia entre ciudades (aristas ponderadas), ...
- Flujo de tráfico
 - Vértices: intersección de calles
 - Aristas: conjunto de calles
 - Peso aristas: límite de velocidad, número de carriles, etc.
- Programas
 - Vértices: bloques básicos de un programa
 - Aristas: posibles transferencias de control de flujo
- Redes de ordenadores

Contenido

4

Tema 4. Grafos

- Nivel Abstracto o de definición
- Nivel de representación o implementación
- Recorridos
- Ordenación Topológica
- Caminos Mínimos
- Árbol de expansión de coste mínimo
- Ejercicios

2 Nivel de representación o implementación

- Normalmente se hace referencia a los vértices de los grafos por sus etiquetas o identificadores. En las aplicaciones reales, un vértice puede contener cualquier tipo de información, aunque en su estudio lo ignoremos
- Simplificamos el problema suponiendo que los identificadores de los vértices están numerados de 1 a n . Si no es así habrá que definir una función biyectiva que traduzca los identificadores al conjunto $\{1, 2, \dots, n\}$
- Dado un grafo $G = (V, A)$ de orden n , para $n \geq 1$, existen dos formas principales de representación:
 - Matriz de Adyacencia

El grafo se representa mediante una matriz lógica m , de dimensión $n \times n$, donde $m[i,j]$ es verdadero si y sólo si el arco (v_i, v_j) está en A

• Listas de Adyacencia

El grafo se representa mediante una matriz m de dimensión n , donde $m[i]$ es un puntero a una lista enlazada que contiene todos los vértices adyacentes a v_i

Declaraciones básicas Matriz de ADYACENCIA

- Grafos no ponderados

El grafo se representa mediante una matriz lógica \mathbf{m} , de dimensión $n \times n$, donde $\mathbf{m}[i,j]$ es verdadero si y sólo si el arco (v_i, v_j) está en A

Algoritmo declaraciones básicas grafos

- 1: **tipos**
- 2: $tipoGrafo = matriz[1..n, 1..n]$ de tipoLógico
- 3: **fin tipos**

- Grafos ponderados

La presencia de un arco se representa con su peso en la matriz

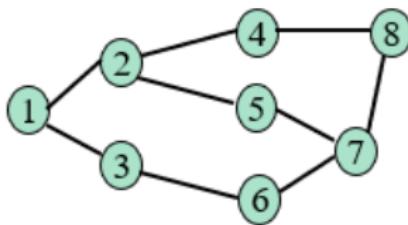
Algoritmo declaraciones básicas grafos

- 1: **tipos**
- 2: $tipoGrafo = matriz[1..n, 1..n]$ de tipoPeso
- 3: **fin tipos**

- Mismas declaraciones para grafos dirigidos y no dirigidos

Grafo no dirigido

Ejemplo



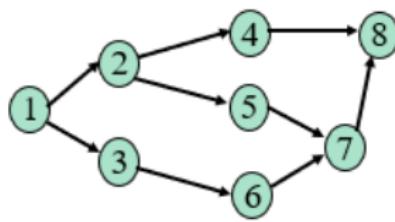
	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	0	1	1	0	0	0
3	1	0	0	0	0	1	0	0
4	0	1	0	0	0	0	0	1
5	0	1	0	0	0	0	1	0
6	0	0	1	0	0	0	1	0
7	0	0	0	0	1	1	0	1
8	0	0	0	1	0	0	1	0

$$\text{grado}(v_i) = \sum_{j=1}^n m[i, j] = \sum_{j=1}^n m[j, i] \Rightarrow \sum \text{filas} \text{ ó } \sum \text{columnas}$$

$$\text{grado}(7) = \sum_{j=1}^8 m[7, j] = \sum_{j=1}^8 m[j, 7] = 3$$

Grafo dirigido

Ejemplo



	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	0	0	0	1	1	0	0	0
3	0	0	0	0	0	1	0	0
4	0	0	0	0	0	0	0	1
5	0	0	0	0	0	0	1	0
6	0	0	0	0	0	0	1	0
7	0	0	0	0	0	0	0	1
8	0	0	0	0	0	0	0	0

$$gradoInterno(v_i) = \sum_{k=1}^n m[k, i] \Rightarrow \sum \text{filas}$$

$$gradoExterno(v_i) = \sum_{k=1}^n m[i, k] \Rightarrow \sum \text{columnas}$$

$$grado(v_i) = gradoInterno(v_i) + gradoExterno(v_i)$$

$$grado(7) = \sum_{k=1}^8 m[k, 7] + \sum_{k=1}^8 m[7, k] = 2 + 1 = 3$$

Declaraciones básicas Listas de ADYACENCIA

- Esta representación requiere dos estructuras de datos, una para representar los vértices y otra para representar los arcos:
 - **Directorio de vértices:** matriz que contiene una entrada para cada vértice del grafo, donde la entrada del vértice i apunta a una lista enlazada que contiene todos los vértices adyacentes a i
 - **Orden** del grafo: número de vértices
 - **Listas de adyacencia:** a cada vértice se le asocia una lista que contiene todos sus vértices adyacentes
- Si los arcos están ponderados habrá que reservar espacio en la estructura que representa los arcos para los pesos
- Mismas declaraciones para grafos dirigidos y no dirigidos (cambia el número de aristas que se representan)

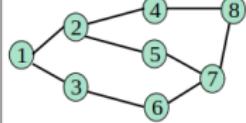
Declaraciones Básicas

Algoritmo declaraciones básicas

```
1: constants
2:    $N = 100$ 
3: tipos
4:    $tipoArco = \text{registro}$ 
5:    $\text{vértice} : tipoVértice$ 
6:    $\text{peso} : tipoPeso$  //aristas ponderadas
7:    $sig : \uparrow tipoArco$ 
8: fin registro
9:    $punteroArco = \uparrow tipoArco$ 
10:   $tipoGrafo = \text{registro}$ 
11:   $directorio : matriz[1..N]$  de punteroArco
12:   $orden : entero$ 
13: fin registro
14: fin tipos
```

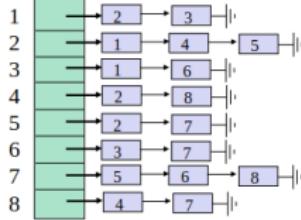
Ejemplos

No dirigido



$\text{grado}(v)$ = número de veces que v aparece en las listas de adyacencia de v

$\text{grado}(7) = 3$



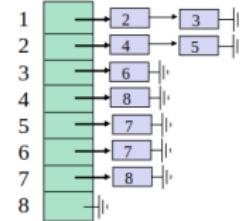
Dirigido



gradoEntrada(v): número de veces que v aparece en las listas de adyacencia del resto de los vértices

gradoSalida(v): número de elementos en la lista de adyacencia de v

$\text{grado}(7) = 2 + 1 = 3$



Observaciones

- La selección apropiada de la representación dependerá de las operaciones que se apliquen a los vértices y a los arcos del grafo
- Matriz de adyacencia
 - El tiempo de acceso requerido a un elemento es independiente del tamaño de V y A. Puede ser útil en aquellos algoritmos que necesiten saber si un arco determinado está presente en el grafo
 - La principal desventaja es que requiere un espacio proporcional a n^2 para representar todos los arcos posibles, aun cuando el grafo tenga menos de n^2 arcos (ocurre siempre en el caso de grafos dirigidos)
 - Examinar la matriz llevará un tiempo de $O(n^2)$
- Listas de adyacencia
 - Requieren un espacio proporcional a la suma del número vértices y de arcos, es útil cuando se quieren representar grafos que tienen un número de arcos mucho menor que n^2
 - Una desventaja potencial es, que determinar si existe un arco, puede llevar un tiempo del $O(n)$, ya que puede haber n vértices en lista de adyacencia
- **En el resto del tema utilizaremos la representación mediante listas de adyacencia**

Contenido

4

Tema 4. Grafos

- Nivel Abstracto o de definición
- Nivel de representación o implementación
- Recorridos
- Ordenación Topológica
- Caminos Mínimos
- Árbol de expansión de coste mínimo
- Ejercicios

3 Recorridos en Grafos

- Para resolver muchos problemas relacionados con grafos, es necesario visitar los vértices y los arcos de manera sistemática
- Debe elegirse un **vértice de partida** y desde él visitar el resto de forma que cada vértice se visite una sola vez
- Un vértice puede encontrarse varias veces en el recorrido, es necesario por tanto, marcar cada vértice a medida que se visita, para no volver a visitarlo ⇒ modificación en la declaración de las estructuras básicas
- Dos categorías básicas:
 - Recorrido en **Amplitud**
 - Recorrido en **Profundidad**

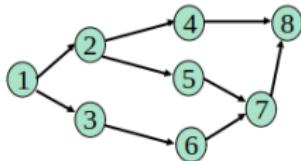
Recorridos y ejemplos

- Recorrido en **Amplitud**

- Se visita un vértice inicial, a continuación todos sus vértices adyacentes, después los adyacentes a estos últimos y así sucesivamente hasta que todos hayan sido visitados

- Recorrido en **Profundidad**

- Se visita un vértice inicial y se siguen visitando sus vértices en una trayectoria hasta el final de esa trayectoria, después se elige otra trayectoria y se visitan todos sus vértices hasta el final de la trayectoria y así sucesivamente hasta visitar todos los vértices



Recorrido Amplitud \Rightarrow 1 2 3 4 5 6 8 7
Recorrido Profundidad \Rightarrow 1 2 4 8 5 7 3 6

Declaraciones básicas

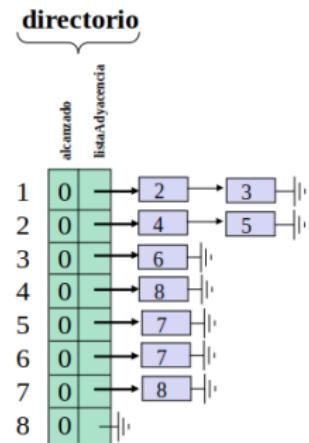
Ampliación y modificación

Algoritmo declaraciones básicas

```

1: tipos
2: tipoArco = registro
3: vértice : tipoVértice
4: peso : tipoPeso /*aristas ponderadas*/
5: sig :↑ tipoArco
6: fin registro
7: punteroArco =↑ tipoArco
8: tipoVértice = registro
9: alcanzado : tipoLógico
10: ...
11: listaAdyacencia : punteroArco
12: fin registro
13: tipoGrafo = registro
14: directorio : matriz[1..N] de tipoVértice
15: orden : entero
16: fin registro
17: fin tipos

```



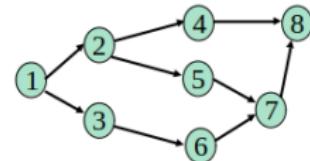
Algoritmo de recorrido en Profundidad

Algoritmo profundidad(vInicio:tipoldVértice, **ref** g:tipoGrafo)

```

1: w : tipoldVértice
2: p : punteroArco
3: visitar(vInicio)
4: g.directorio[vInicio].alcanzado ← VERDADERO
5: p ← g.directorio[vInicio].listaAdyacencia
6: mientras p ≠ NULO hacer
7:   w ← p ↑ .vertice
8:   si NOT(g.directorio[w].alcanzado) entonces
9:     profundidad(w, g)
10:    fin si
11:    p ← p ↑ .sig
12: fin mientras

```



directorio

alcanzado
listaAdyacencia

1	0	2	3	
2	0	4	5	
3	0	6		
4	0	8		
5	0	7		
6	0	7		
7	0	8		
8	0			

Recorrido Profundidad \Rightarrow 1 2 4 8 5 7 3 6

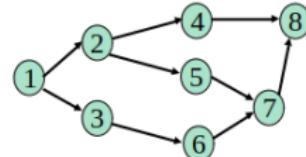
Algoritmo de recorrido en Amplitud

Algoritmo amplitud(vInicio:tipoldVértice, ref g:tipoGrafo)

```

1: w : tipoldVértice
2: p : punteroArco
3: c : Cola
4: creaVacia(c)
5: inserta(vInicio, c)
6: mientras NOT(vacia(c)) hacer
7:   w ← suprime(c)
8:   si NOTg.directorio[w].alcanzado entonces
9:     visitar(w)
10:    g.directorio[w].alcanzado ← VERDADERO
11:    p ← g.directorio[w].listaAdyacencia
12:    mientras p ≠ NULO hacer
13:      inserta(p ↑ .vertice, c)
14:      p ← p ↑ .sig
15:    fin mientras
16:  fin si
17: fin mientras
  
```

Recorrido Amplitud \Rightarrow 1 2 3 4 5 6 8 7



directorio

	alcancado	listaAdyacencia
1	0	2 → 3 →
2	0	4 → 5 →
3	0	6 →
4	0	8 →
5	0	7 →
6	0	7 →
7	0	8 →
8	0	

Observaciones

- El directorio de vértices tiene que estar correctamente inicializado antes de comenzar el recorrido: ningún vértice marcado
- Puede que algunos vértices no se visiten dependiendo del vértice de partida y del tipo de grafo:
 - En grafos dirigidos si existen vértices inalcanzables desde el vértice inicial
 - En grafos no conexos el recorrido solo visita los vértices conectados con el vértice inicial
- **Solución:** buscar los vértices no marcados y aplicarles de nuevo el recorrido hasta que no queden vértices sin marcar

Contenido

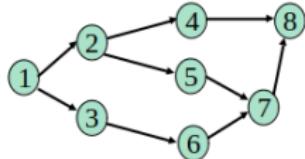
4

Tema 4. Grafos

- Nivel Abstracto o de definición
- Nivel de representación o implementación
- Recorridos
- Ordenación Topológica
- Caminos Mínimos
- Árbol de expansión de coste mínimo
- Ejercicios

4 Ordenación Topológica

- Consiste en la clasificación de los vértices de un **grafo dirigido acíclico** (gda) tal que si existe un camino de v a w , v aparece antes que w en la clasificación
- Observaciones:
 - No es posible la ordenación topológica si en grafos cíclicos: para dos vértices v y w en el ciclo, v precede a w y w precede a v
 - La ordenación topológica no es necesariamente única



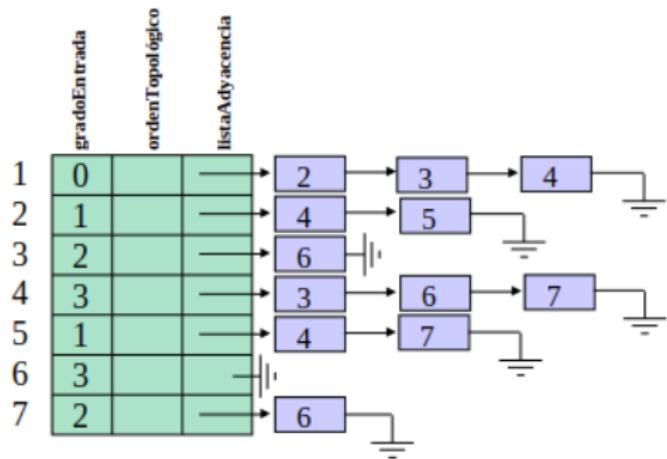
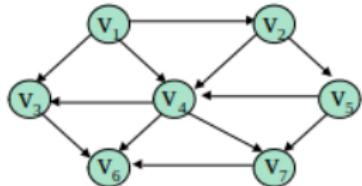
Orden topológico

- 1) 1 2 3 4 5 6 7 8
- 2) 1 3 2 6 4 5 7 8

Algoritmo sencillo para establecer la ordenación topológica

- ① Encontrar un vértice cualquiera, v , con grado de entrada cero (a ese vértice no llegan aristas) y asignarle el primer orden topológico
 - ② Decrementar el grado de entrada de todos los vértices adyacentes a v y aplicar la misma estrategia a aquellos vértices que todavía no tengan asignado un orden
El primer vértice cuyo grado de entrada se convierta en cero, será el siguiente vértice en orden topológico, ya que no podremos volver a acceder a él desde v
- Para formalizar el algoritmo es necesario calcular el grado de entrada de cada vértice del grafo y guardar esta información en la estructura que representa al grafo (ampliación declaraciones básicas)

Ejemplo



Vértice Ordenado	v_1	v_2	v_5	v_4	v_3	v_7	v_6
Orden Topológico	1	2	3	4	5	6	7
inicio							
1	0	1	1	0	1	1	0
2	1	0	2	0	2	0	2
3	2	1	1	1	0	5	0
4	3	2	1	0	4	4	0
5	1	5	0	0	3	0	5
6	3	6	3	3	0	3	0
7	2	7	2	1	0	6	7

Algoritmo de Ordenación Topológica

versión 1

Algoritmo ordenTopológico(ref g: tipoGrafo)

Entrada: g grafo

Salida: ordenación topológica de los vértices de g

```
1: orden : entero
2: p : punteroArco
3: v, w : tipoldVértice
4: iniciar(g)
5: para orden ← 1 hasta g.orden hacer
6:   v ← buscarVérticeGradoCeroNoOrdenTop(g)
7:   si v = -1 entonces
8:     error("grafo cíclio")
9:   si no
10:    g.directorio[v].ordenTopológico ← orden
11:    p ← g.directorio[v].listaAdyacencia
12:    mientras p ≠ NULO hacer
13:      w ← p ↑ .vertice
14:      g.directorio[w].gradoEntrada ← g.directorio[w].gradoEntrada – 1
15:      p ← p ↑ .sig
16:    fin mientras
17:  fin si
18: fin para
```

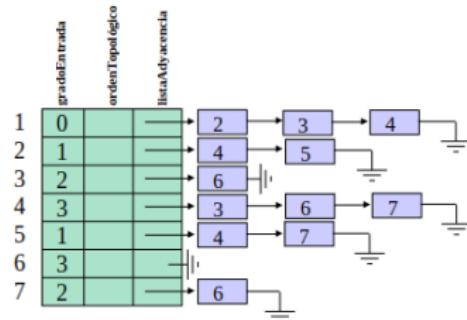
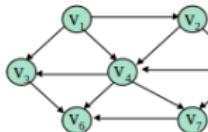
Observaciones:

- La función **buscarVérticeGradoCeroNoOrdenTop** recorre el directorio de vértices buscando un vértice **v**, con grado de entrada cero, al que todavía no se haya asignado un orden topológico
 - Devuelve el identificador de ese vértice, si existe; y -1 si no existe, indicando que hay un ciclo
- Cada llamada a esta función recorre secuencialmente el directorio de vértices y se llama n veces \Rightarrow tiempo de ejecución deficiente de $O(n^2)$
- **Mejora:** utilización de una cola en la que se vayan insertando todos los vértices con grado cero
 - Cada vez que se decrementan los grados de entrada de los vértices, estos se revisan y se insertan en la cola si su grado se convierte en cero

Proceso de ordenación topológica II

- ① Calcular del grado de entrada de todos los vértices
- ② Todos los vértices con grado de entrada cero se insertan en la cola (initialmente vacía)
- ③ Mientras la cola no esté vacía
 - ① Se suprime un vértice de la cola: v
 - ② Se decrementan los grados de todos los vértices adyacentes a v
 - ③ Si el grado de entrada de algún vértice se convierte en cero, se inserta en la cola
- ④ La ordenación topológica coincide con el orden en que los vértices van saliendo de la cola

Ejemplo



Vértice	Grado de entrada antes de desencolar							
v_1	0	0	0	0	0	0	0	0
v_2	$1 \Rightarrow$	0	0	0	0	0	0	0
v_3	$2 \Rightarrow$	1	1	$1 \Rightarrow$	0	0	0	0
v_4	$3 \Rightarrow$	$2 \Rightarrow$	1 \Rightarrow	0	0	0	0	0
v_5	1	$1 \Rightarrow$	0	0	0	0	0	0
v_6	3	3	3	$3 \Rightarrow$	$2 \Rightarrow$	$1 \Rightarrow$	0	0
v_7	2	2	$2 \Rightarrow$	$1 \Rightarrow$	0	0	0	0
En cola	v_1	v_2	v_5	v_4	v_3, v_7	v_7	v_6	
Desencolar	v_1	v_2	v_5	v_4	v_3	v_7	v_6	
Orden Topológico	1	2	3	4	5	6	7	

Algoritmo de Ordenación Topológica

versión 2

Algoritmo ordenTop(*ref g: tipoGrafo*)

```
1: v, w : tipoldVértice
2: c : Cola
3: inicia(g)
4: creaVacia(c)
5: para v  $\leftarrow$  1 hasta g.orden hacer
6:   si g.directorio[v].gradoEntrada = 0 entonces
7:     inserta(v, c)
8:   fin si
9: fin para
10: orden  $\leftarrow$  1
11: mientras NOT(vacia(c)) hacer
12:   v  $\leftarrow$  suprime(c)
13:   g.directorio[v].ordenTop  $\leftarrow$  orden
14:   orden ++
15:   p  $\leftarrow$  g.directorio[v].listaAdyacencia
16:   mientras p  $\neq$  NULO hacer
17:     w  $\leftarrow$  p  $\uparrow$ .vertice
18:     g.directorio[w].gradoEntrada  $\leftarrow$  g.directorio[w].gradoEntrada - 1
19:     si g.directorio[w].gradoEntrada = 0 entonces
20:       inserta(w, c)
21:     fin si
22:     p  $\leftarrow$  p  $\uparrow$ .sig
23:   fin mientras
24: fin mientras
```

Observaciones

- El tiempo de ejecución del nuevo algoritmo es $O(n + a)$
 - Las operaciones encola y desencola una vez por vértice $\Rightarrow O(n)$
 - El bucle mientras interior se ejecuta una vez por arista $\Rightarrow O(a)$
 - La asignación de valores iniciales toma un tiempo proporcional al tamaño del grafo
- Los algoritmos de ordenación topológica, con pequeñas modificaciones, pueden utilizarse para determinar si un grafo es cíclico

Contenido

4

Tema 4. Grafos

- Nivel Abstracto o de definición
- Nivel de representación o implementación
- Recorridos
- Ordenación Topológica
- **Caminos Mínimos**
- Árbol de expansión de coste mínimo
- Ejercicios

5 Algoritmos de Caminos Mínimos

Grafos dirigidos

- Un **camino** o **trayectoria** en un grafo es una secuencia de vértices $v_i(1), v_i(2), \dots, v_i(n)$ tal que la arista $(v_i(x), v_i(x+1)) \in A$ para $1 \leq x < n$
- **Longitud** de camino: número de arcos que lo componen ($n-1$)
- **Coste** de un camino:
 - Grafo dirigido **ponderado**: cada arco del grafo (v_i, v_j) tiene asociado un peso $p_{i,j} \geq 0$, de tal forma que el camino (v_1, v_2, \dots, v_k) tiene asociado un coste que viene dado por

$$\text{coste}(v_i, v_k) = \sum_{i=1}^{k-1} p_{i,i+1}$$

- Grafo dirigido **no ponderado**: el coste de un camino coincide con su longitud, es decir, con su número de arcos. Es un caso especial de grafo ponderado con coste igual a 1 en todos los arcos.

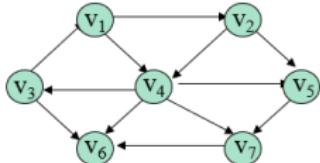
Planteamiento del Problema

- Determinar
 - ① El coste del camino mínimo entre un par de vértices
 - ② La trayectoria real que nos conduce con coste mínimo de un vértice a otro
- Los algoritmos que resuelven el problema proporcionan caminos mínimos y trayectorias entre un vértice inicial y el resto de los vértices del grafo
- En grafos no ponderados, el camino de menor coste entre un par de vértices será el de menor número de arcos, es decir, el de menor longitud
- En grafos ponderados puede no ser así, depende del coste del camino

Grafos no ponderados

- Tomando un vértice inicial, v , encontrar el camino más corto de v al resto de los vértices del grafo
- Grafo no ponderado: solo interesa el número de aristas que contiene el camino. Es un caso especial de grafo ponderado con coste igual a 1 en todos los arcos

- Ejemplo



Un único camino simple de v_3 a v_1
Varios caminos simples de v_3 a v_6

Trayectoria del camino	Coste o distancia
(v_3, v_1)	1
(v_3, v_6)	1
(v_3, v_1, v_4, v_6)	3
(v_3, v_1, v_4, v_7, v_6)	4
($v_3, v_1, v_4, v_5, v_7, v_6$)	5

Ampliación declaraciones básicas

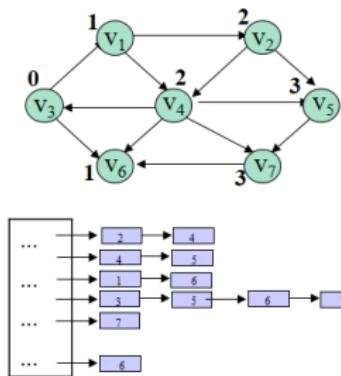
- Los algoritmos de caminos mínimos utilizan estrategias que necesitan ampliar la información para cada vértice en el directorio de vértices:
 - alcanzado:** tendrá valor VERDADERO si en el recorrido se ha pasado por ese vértice. Inicialmente con valor FALSO
 - distancia:** valor que indicará el número de arcos desde el vértice inicial al vértice representado (longitud de camino). Inicialmente con valor inalcanzable (“infinito”) excepto el vértice de partida que tendrá valor 0
 - anterior:** almacenará el último vértice desde el que se alcanza el vértice representado. Inicialmente a valor 0. Permitirá mostrar los caminos reales

Estado inicial directorio de vértices

v	alcanzado	distancia	anterior
v_1	0	∞	0
v_2	0	∞	0
v_3	0	∞	0
v_4	0	∞	0
v_5	0	∞	0
v_6	0	∞	0
v_7	0	∞	0

Ejemplo

vértice origen v_3



Inicio

v	alcanzado	distancia anterior
v_1	0	∞
v_2	0	∞
v_3	0	0
v_4	0	∞
v_5	0	∞
v_6	0	∞
v_7	0	∞

distanciaActual = 0 y
noAlcanzado(v_3)

v	alcanzado	distancia anterior
v_1	0	1
v_2	0	∞
v_3	1	0
v_4	0	∞
v_5	0	∞
v_6	0	1
v_7	0	∞

distanciaActual = 1 y
noAlcanzado(v_1 y v_6)

v	alcanzado	distancia anterior
v_1	1	1
v_2	0	2
v_3	1	0
v_4	0	2
v_5	0	∞
v_6	1	1
v_7	0	∞

distanciaActual = 2 y
noAlcanzado(v_2 y v_4)

v	alcanzado	distancia anterior
v_1	1	1
v_2	1	2
v_3	1	0
v_4	1	2
v_5	0	3
v_6	1	1
v_7	0	3

distanciaActual = 3 y
noAlcanzado(v_5 y v_7)

v	alcanzado	distancia anterior
v_1	1	1
v_2	1	2
v_3	1	0
v_4	1	2
v_5	1	3
v_6	1	1
v_7	1	3

Algoritmo de Camino de Longitud Mínima

versión 1

Algoritmo caminoM(vInicio:tipoldVértice, ref g: tipoGrafo)

```

1: p : punteroArco
2: v, w : tipoldVértice
3: distanciaActual : entero
4: inicia(g)
5: g.directorio[vInicio].distancia  $\leftarrow$  0
6: para distanciaActual  $\leftarrow$  0 hasta g.orden  $- 1$  hacer
7:   para v  $\leftarrow$  1 hasta g.orden hacer
8:     si (NOT(g.directorio[v].alcanzado) Y
      (g.directorio[v].distancia=distanciaActual) entonces
9:       g.directorio[v].alcanzado  $\leftarrow$  VERDADERO
10:      p  $\leftarrow$  g.directorio[v].listaAdyacencia
11:      mientras p  $\neq$  NULO hacer
12:        w  $\leftarrow$  p  $\uparrow$ .vertice
13:        si g.directorio[w].distancia = INFINITO entonces
14:          g.directorio[w].distancia  $\leftarrow$  g.directorio[v].distancia + 1
15:          g.directorio[w].anterior  $\leftarrow$  v
16:        fin si
17:        p  $\leftarrow$  p  $\uparrow$ .sig
18:      fin mientras
19:    fin si
20:  fin para
21: fin para

```

Observaciones

- Debe inicializarse correctamente el directorio de vértices: todos los vértices son inalcanzables excepto el vértice de partida, cuya longitud de camino es 0
- Cuando se procesa un vértice se tiene la garantía de que no se encontrará un camino más económico para alcanzarlo. Alcanzado toma el valor 1 y el procesamiento de ese vértice está completo
- Pueden existir vértices inalcanzables desde el vértice inicial
¿Qué ocurre si el vértice de inicio es v_6 ?
- Es posible mostrar el camino real de un vértice a otro regresando a través del campo anterior

Interpretando el resultado del algoritmo

Ejemplo

vértice	alcanzado	distancia	anterior
v_1	1	1	v_3
v_2	1	2	v_1
v_3	1	0	0
v_4	1	2	v_1
v_5	1	3	v_2
v_6	1	1	v_3
v_7	1	3	v_4

- **¿Existe un camino entre v_3 y v_7 ?**

Sí, existe un camino, que además es el de coste mínimo (distancia 3)

- **¿Qué trayectoria tiene ese camino?**

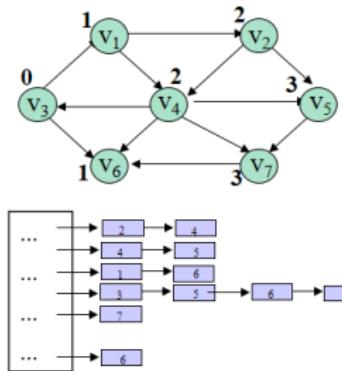
$v_3 \quad v_1 \quad v_4 \quad v_7$

Análisis

- Tiempo de ejecución del algoritmo bastante deficiente debido a los ciclos “para” doblemente anidados($O(n^2)$)
 - El ciclo externo continúa hasta $n-1$ aunque todos los vértices se hayan alcanzado mucho antes
- **Mejora**
- Utilizar una cola que inicialmente guardará el vértice de partida (longitud de camino 0), y a medida que va sacando los vértices de la cola almacena sus adyacentes (longitud de camino 1), y así sucesivamente
 - La cola garantiza que no se procesan vértices de longitud de camino **d+1** hasta que no se hayan procesado todos los vértices con longitud de camino **d**

Ejemplo

vértice origen v_3



Inicio

v	alc.	dist.	ant.
v_1	0	∞	0
v_2	0	∞	0
v_3	0	0	0
v_4	0	∞	0
v_5	0	∞	0
v_6	0	∞	0
v_7	0	∞	0
C		v_3	

v_3 desencolado

v	alc.	dist.	ant.
v_1	0	1	v_3
v_2	0	∞	0
v_3	1	0	0
v_4	0	∞	0
v_5	0	∞	0
v_6	0	1	v_3
v_7	0	∞	0
C		$v_1 v_6$	

v_1 desencolado

v	alc.	dist.	ant.
v_1	1	1	v_3
v_2	0	2	v_1
v_3	1	0	0
v_4	0	2	v_1
v_5	0	∞	0
v_6	1	1	v_3
v_7	0	∞	0
C		$v_6 v_2 v_4$	

v_6 desencolado

v	alc.	dist.	ant.
v_1	1	1	v_3
v_2	0	2	v_1
v_3	1	0	0
v_4	0	2	v_1
v_5	0	∞	0
v_6	1	1	v_3
v_7	0	∞	0
C		$v_2 v_4$	

v_2 desencolado

v	alc.	dist.	ant.
v_1	1	1	v_3
v_2	1	2	v_1
v_3	1	0	0
v_4	0	2	v_1
v_5	0	3	v_2
v_6	1	1	v_3
v_7	0	∞	0
C		$v_4 v_5$	

v_4 desencolado

v	alc.	dist.	ant.
v_1	1	1	v_3
v_2	1	2	v_1
v_3	1	0	0
v_4	1	2	v_1
v_5	0	3	v_2
v_6	1	1	v_3
v_7	0	3	v_4
C		$v_5 v_7$	

v_5 desencolado

v	alc.	dist.	ant.
v_1	1	1	v_3
v_2	1	2	v_1
v_3	1	0	0
v_4	1	2	v_1
v_5	1	3	v_2
v_6	1	1	v_3
v_7	0	3	v_4
C		v_7	

v_7 desencolado

v	alc.	dist.	ant.
v_1	1	1	v_3
v_2	1	2	v_1
v_3	1	0	0
v_4	1	2	v_1
v_5	1	3	v_2
v_6	1	1	v_3
v_7	1	3	v_4
C		vacia	

Algoritmo de Camino de Longitud Mínima

versión 2

Algoritmo caminoM(vInicio:tipoldVértice, **ref** g: tipoGrafo)

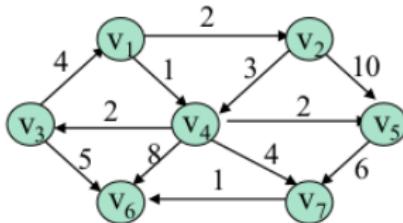
```
1: p : punteroArco
2: v, w : tipoldVértice
3: c : Cola
4: inicia(g)
5: g.directorio[vInicio].distancia  $\leftarrow$  0
6: creaVacia(c)
7: inserta(vInicio, c)
8: mientras NOT(vacia(c)) hacer
9:   v  $\leftarrow$  suprime(c)
10:  p  $\leftarrow$  g.directorio[v].listaAdyacencia
11:  mientras p  $\neq$  NULO hacer
12:    w  $\leftarrow$  p  $\uparrow$  .vertice
13:    si g.directorio[w].distancia = INFINITO entonces
14:      g.directorio[w].distancia  $\leftarrow$  g.directorio[v].distancia + 1
15:      g.directorio[w].anterior  $\leftarrow$  v
16:      inserta(w, c)
17:    fin si
18:    p  $\leftarrow$  p  $\uparrow$  .sig
19:  fin mientras
20: fin mientras
```

Observaciones

- Una vez procesado un vértice nunca puede volver a entrar en la cola, queda marcado implícitamente que no se debe volver a procesar (puede eliminarse la información alcanzado)
- Puede ocurrir que la cola se vacíe prematuramente si algunos vértices son inalcanzables desde el vértice origen, para estos vértices inalcanzables se obtendrá una distancia “infinita”, lo cual es perfectamente lógico
¿Qué ocurre si el vértice de inicio es v6?
- El tiempo de ejecución de este algoritmo es $O(n+a)$

Grafos Ponderados

- Tomando un vértice inicial, v , encontrar el camino de menor coste (puede que no sea el más corto) de v al resto de los vértices del grafo
- Ejemplo:



Varios caminos simples de v_1 a v_6

Trayectoria del camino	Coste o distancia
$(v_1, v_2, v_4, v_3, v_6)$	12
(v_1, v_2, v_4, v_6)	13
$(v_1, v_2, v_4, v_5, v_7, v_6)$	14
$(v_1, v_2, v_4, v_7, v_6)$	10
...	
(v_1, v_4, v_7, v_6)	6

Grafos PONDERADOS versus NO PONDERADOS

- Aplicando la misma lógica que en el caso no ponderado (versión 1):
 - se marca cada vértice como alcanzado o no alcanzado
 - se utiliza una **distancia provisional** por cada vértice que será la de menor coste desde el vértice inicial utilizando como intermediarios solo vértices alcanzados con la siguiente diferencia:

- **No ponderado:**

$$d_w = d_v + 1 \text{ si } d_w = \infty$$

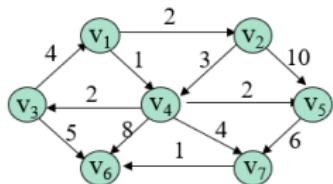
- **Ponderado:**

$$d_w = d_v + \text{peso}_{v,w} \text{ si } d_v + \text{peso}_{v,w} < d_w$$

Se actualiza d_w si el nuevo valor ofrece una mejoría sobre el anterior

Ejemplo

vértice origen v_1



Estado inicial

v	alc.	dist.	ant.
v_1	0	0	0
v_2	0	∞	0
v_3	0	∞	0
v_4	0	∞	0
v_5	0	∞	0
v_6	0	∞	0
v_7	0	∞	0

 v_1 alcanzado

v	alc.	dist.	ant.
v_1	1	0	0
v_2	0	2	v_1
v_3	0	∞	0
v_4	0	1	v_1
v_5	0	∞	0
v_6	0	∞	0
v_7	0	∞	0

 v_4 alcanzado

v	alc.	dist.	ant.
v_1	1	0	0
v_2	0	2	v_1
v_3	0	3	v_4
v_4	1	1	v_1
v_5	0	3	v_4
v_6	0	9	v_4
v_7	0	5	v_4

 v_2 alcanzado

v	alc.	dist.	ant.
v_1	1	0	0
v_2	1	2	v_1
v_3	0	3	v_4
v_4	1	1	v_1
v_5	0	3	v_4
v_6	0	9	v_4
v_7	0	5	v_4

 v_5 alcanzado

v	alc.	dist.	ant.
v_1	1	0	0
v_2	1	2	v_1
v_3	0	3	v_4
v_4	1	1	v_1
v_5	1	3	v_4
v_6	0	9	v_4
v_7	0	5	v_4

 v_3 alcanzado

v	alc.	dist.	ant.
v_1	1	0	0
v_2	1	2	v_1
v_3	1	3	v_4
v_4	1	1	v_1
v_5	1	3	v_4
v_6	0	8	v_3
v_7	0	5	v_4

 v_7 alcanzado

v	alc.	dist.	ant.
v_1	1	0	0
v_2	1	2	v_1
v_3	1	3	v_4
v_4	1	1	v_1
v_5	1	3	v_4
v_6	0	6	v_7
v_7	1	5	v_4

 v_6 alcanzado

v	alc.	dist.	ant.
v_1	1	0	0
v_2	1	2	v_1
v_3	1	3	v_4
v_4	1	1	v_1
v_5	1	3	v_4
v_6	1	6	v_7
v_7	1	5	v_4

Algoritmo de Dijkstra (versión 1)

Caminos mínimos en grafos ponderados

Algoritmo Dijkstra(vInicio:tipoldVértice, ref g: tipoGrafo)

```

1: p : punteroArco
2: v, w : tipoldVértice
3: distanciaActual : entero
4: i : entero
5: inicia(g)
6: g.directorio[vInicio].distancia  $\leftarrow$  0
7: para i  $\leftarrow$  1 hasta g.orden hacer
8:   v  $\leftarrow$  buscarVérticeDistanciaMínimaNoAlcanzado(g)
9:   g.directorio[v].alcanzado  $\leftarrow$  VERDADERO
10:  p  $\leftarrow$  g.directorio[v].listaAdyacencia
11:  mientras p  $\neq$  NULO hacer
12:    w  $\leftarrow$  p  $\uparrow$  .vertice
13:    si NOT(g.directorio[w].alcanzado) entonces
14:      si g.directorio[v].distancia + p  $\uparrow$  .peso  $<$  g.directorio[w].distancia entonces
15:        g.directorio[w].distancia  $\leftarrow$  g.directorio[v].distancia + p  $\uparrow$  .peso
16:        g.directorio[w].anterior  $\leftarrow$  v
17:      fin si
18:    fin si
19:    p  $\leftarrow$  p  $\uparrow$  .sig
20:  fin mientras
21: fin para
```

Análisis

- El tiempo de ejecución es $O(a + n^2)$
 - La función *buscarVérticeDistanciaMínimaNoAlcanzado* toma un tiempo $O(n)$ en recorrer el directorio. Se llama n veces por tanto consumirá en todo el algoritmo un tiempo de $O(n^2)$
 - El tiempo para actualizar d_w es constante y se ejecuta como mucho una vez por arista
- Si el grafo es denso ($a = n^2$) el algoritmo es sencillo y óptimo: se ejecuta en un tiempo lineal sobre el número de aristas
- Si el grafo es disperso ($a \ll n^2$) el algoritmo de Dijkstra es demasiado lento
- **Mejora:** Utilización de una cola de prioridad (montículo binario)

Estrategia

- Utilizar un montículo en el que se guarda el nuevo valor de la distancia cada vez que se ajusta un vértice (clave del montículo) y el identificador del vértice ajustado.
- Cada elemento del montículo contiene
 - **clave:** distancia conseguida
 - **información:** identificador del vértice
- Puede haber más de un representante de cada vértice en la cola de prioridad pero siempre se elimina primero la ocurrencia de distancia mínima (criterio de orden) y en ese momento se marca (no puede conseguirse una distancia mejor)
- La operación *buscarVérticeDistanciaMínimaNoAlcanzado* debe convertirse en un ciclo que ejecutará *eliminarMin* hasta que aparezca un vértice no alcanzado
 - El tamaño de la cola de prioridad puede llegar a coincidir con el número de aristas

Algoritmo de Dijkstra (versión 2)

Caminos mínimos en grafos ponderados

Algoritmo Dijkstra(vInicio:tipoDeVértice, **ref** g: tipoGrafo)

```

1: m : tipoMontículo
2: x : tipoElemento
3: ...
4: inicia(g)
5: g.directorio[vInicio].distancia ← 0
6: creaVacio(m)
7: x.clave ← 0
8: x.información ← vInicio
9: inserta(x, m)
10: mientras NOT(vacio(m)) hacer
11:   x ← eliminarMin(m)
12:   v ← x.información
13:   si NOT(g.directorio[v].alcanzado) entonces
14:     g.directorio[v].alcanzado ← VERDADERO
15:     p ← g.directorio[v].listaAdyacencia
16:     mientras p ≠ NULO hacer
17:       w ← p ↑ .vertice
18:       si NOT(g.directorio[w].alcanzado) entonces
19:         si g.directorio[v].distancia + p ↑ .peso < g.directorio[w].distancia
20:           entonces
21:             g.directorio[w].distancia ← g.directorio[v].distancia + p ↑ .peso
22:             g.directorio[w].anterior ← v
23:             x.clave ← g.directorio[w].distancia
24:             x.información ← w
25:             inserta(x, m)
26:           fin si
27:         fin si
28:         p ← p ↑ .sig
29:       fin mientras
30:     fin si
31:   fin mientras

```

Análisis

- La operación *buscarVérticeDistanciaMínimaNoAlcanzado* debe convertirse en un ciclo que ejecutará *eliminarMin* hasta que aparezca un vértice no alcanzado
 - La operación *eliminarMin* toma un tiempo en $O(\log t)$ siendo t el tamaño del montículo
 - El tamaño del montículo puede llegar a coincidir con el número de aristas $\Rightarrow O(a \log a)$
 - El tiempo de ejecución por tanto está en $O(a \log n)$ ya que para grafos dispersos $a < n^2$ y, por tanto, $\log a < 2 \log n$

Contenido

4

Tema 4. Grafos

- Nivel Abstracto o de definición
- Nivel de representación o implementación
- Recorridos
- Ordenación Topológica
- Caminos Mínimos
- Árbol de expansión de coste mínimo
- Ejercicios

6 Árbol de expansión de coste mínimo

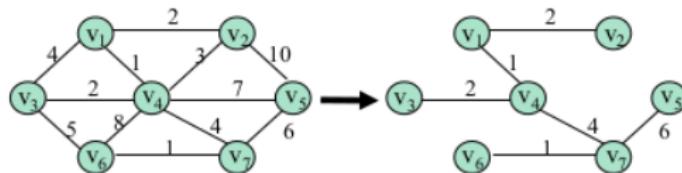
Grafos no dirigidos

Definición 1

Un árbol de expansión mínimo de un grafo no dirigido G es el árbol formado a partir de las aristas que conectan todos los vértices de G con un coste total mínimo

Definición 2

Un árbol de expansión de un grafo no dirigido $G=(V, A)$ y conexo es un subgrafo $G'=(V, A')$ no dirigido, conexo y sin ciclos. Si el grafo es ponderado, el coste del árbol de expansión será la suma de los costes de las aristas



Dos estrategias voraces

- Algoritmos básicos que resuelven el problema: Prim y Kruskal
 - Algoritmo de **Prim**
Selecciona un vértice y construye el árbol a partir de ese vértice, seleccionando en cada etapa la arista más corta que extienda el árbol
 - Algoritmo de **Kruskal**
Selecciona en cada paso la arista más corta que todavía no se haya considerado

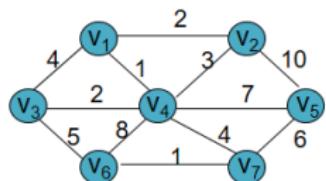
Estrategia del algoritmo de Prim

- Hacer crecer el árbol en etapas sucesivas, de forma que en cada etapa se agrega al árbol una arista (la de menor peso) y con ella su vértice asociado
- **Proceso:**
 - ① Elegir un vértice cualquiera **u** del grafo como nodo raíz
 - ② Repetir mientras queden vértices por añadir al árbol:
 - Seleccionar la arista (u, v) con menor peso entre todas las aristas tal que **u** está en el árbol y **v** no
 - Agregar **v** al árbol
- En cada etapa se agrega al árbol un vértice **v** si el peso de la arista (u, v) es el menor entre todas las aristas tal que **u** está en el árbol y **v** no
- Una vez elegido **v**, para cada vértice **w** no alcanzado adyacente a **v** se actualiza el peso y anterior, si se obtiene un peso menor

$$p_w = \min(p_w, \text{peso}_{w,v})$$

Ampliación declaraciones básicas

- Como en el algoritmo de Dijkstra se necesita mantener información sobre cada vértice :
 - alcanzado**: indica si el vértice ya se ha incluido en el árbol
 - peso**: peso del arco de menor coste que conecta ese vértice con un vértice alcanzado
 - anterior**: identificador del último vértice que ocasiona un cambio en peso

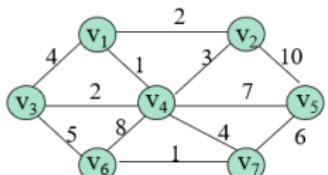


Estado inicial

v	alc.	peso	ant.	
V ₁	0	0	0	→ [2] 2 → [3] 4 → [4] 1
V ₂	0	∞	0	→ [1] 2 → [4] 3 → [5] 10
V ₃	0	∞	0	→ [1] 4 → [4] 2 → [6] 5
V ₄	0	∞	0	→ [1] 1 → [2] 3 → [3] 2 → [5] 7 → [6] 8 → [7] 4
V ₅	0	∞	0	→ [2] 10 → [4] 7 → [7] 6
V ₆	0	∞	0	→ [3] 5 → [4] 8 → [7] 1
V ₇	0	∞	0	→ [4] 4 → [5] 6 → [6] 1

Ejemplo

vértice inicial v_1



v	alc.	peso	ant.	
v_1	0	0	0	
v_2	0	∞	0	
v_3	0	∞	0	
v_4	0	∞	0	
v_5	0	∞	0	
v_6	0	∞	0	
v_7	0	∞	0	

Camino de v_1 a v_7 (peso total 14): $v_1 \rightarrow v_4 \rightarrow v_6 \rightarrow v_7$. Ruta: $2|2 \rightarrow 3|4 \rightarrow 4|1$.

v_1 alcanzado

v	A	P	ant.
v_1	1	0	0
v_2	0	2	v_1
v_3	0	4	v_1
v_4	0	1	v_1
v_5	0	∞	0
v_6	0	∞	0
v_7	0	∞	0

v_4 alcanzado

v	A	P	ant.
v_1	1	0	0
v_2	0	2	v_1
v_3	0	2	v_4
v_4	1	1	v_1
v_5	0	7	v_4
v_6	0	8	v_4
v_7	0	4	v_4

v_2 y v_3 alcanzados

v	A	P	ant.
v_1	1	0	0
v_2	1	2	v_1
v_3	1	2	v_4
v_4	1	1	v_1
v_5	0	7	v_4
v_6	0	5	v_3
v_7	0	4	v_4

v_7 alcanzado

v	A	P	ant.
v_1	1	0	0
v_2	1	2	v_1
v_3	1	2	v_4
v_4	1	1	v_1
v_5	0	6	v_7
v_6	0	1	v_7
v_7	1	4	v_4

v_6 y v_5 alcanzados

v	A	P	ant.
v_1	1	0	0
v_2	1	2	v_1
v_3	1	2	v_4
v_4	1	1	v_1
v_5	1	6	v_7
v_6	1	1	v_7
v_7	1	4	v_4

Algoritmo de Prim (versión 1)

Árbol de expansión mínimo

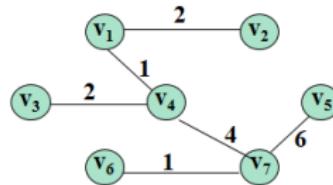
Algoritmo Prim(v_linicio:tipoldVértice, **ref** g: tipoGrafo)

```
1: p : punteroArco
2: v, w : tipoldVértice
3: i : entero
4: inicia(g)
5: g.directorio[vlinicio].peso ← 0
6: para i ← 1 hasta g.orden hacer
7:   v ← buscarVérticeCosteMínimoNoAlcanzado(g)
8:   g.directorio[v].alcanzado ← VERDADERO
9:   p ← g.directorio[v].listaAdyacencia
10:  mientras p ≠ NULO hacer
11:    w ← p↑.vertice
12:    si NOT(g.directorio[w].alcanzado) entonces
13:      si g.directorio[w].peso > p↑.peso entonces
14:        g.directorio[w].peso ← p↑.peso
15:        g.directorio[w].anterior ← v
16:      fin si
17:    fin si
18:    p ← p↑.sig
19:  fin mientras
20: fin para
```

Observaciones:

- La implantación completa del algoritmo es prácticamente idéntica al algoritmo de Dijkstra
- El algoritmo de Prim se ejecuta sobre grafos no dirigidos: todas las aristas deben aparecer en dos listas de adyacencia
- El algoritmo es independiente del vértice inicial elegido (puede eliminarse el primer parámetro)
- El tiempo de ejecución es $O(n^2)$ y puede mejorarse para grafos poco densos a $O(a \log n)$ utilizando montículos binarios
- Interpretación de resultado

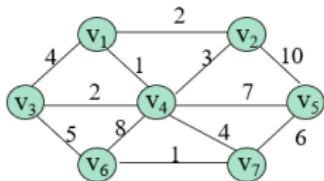
v	A	P	ant.
v_1	1	0	0
v_2	1	2	v_1
v_3	1	2	v_4
v_4	1	1	v_1
v_5	1	6	v_7
v_6	1	1	v_7
v_7	1	4	v_4



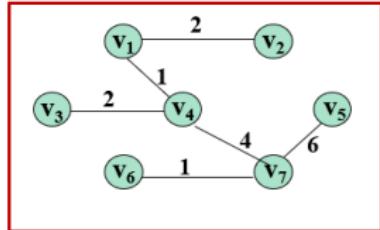
Estrategia del algoritmo de Kruskal

- Uso del TAD Conjuntos Disjuntos estudiado en el tema anterior
- Inicialmente se suponen n árboles de un solo nodo ($\text{crear}(B)$)
- Se van agregando arista combinando dos árboles en uno ($\text{union}(u,v,B)$)
- Al finalizar, se tiene un solo componente que forma el árbol de expansión mínimo del grafo
- **Proceso:**
 - ① Crear un bosque B (conjunto de árboles), donde cada vértice del grafo forma un árbol diferente
 - ② Crear un conjunto A que contenga todas las aristas del grafo
 - ③ Repetir mientras A tenga aristas
 - eliminar de A la arista de menor peso
 - si esa arista conecta dos árboles diferentes se añade al bosque, combinando los dos árboles en un solo árbol, en caso contrario se rechaza

Ejemplo



Crear un bosque B donde cada vértice del grafo forma un árbol diferente



Aceptar aristas si conecta dos árboles diferentes

A veces no hace falta revisar todas las aristas

Crear un conjunto A que contenga a todas las aristas del grafo

Arista	Peso	Aceptada
(v_1, v_4)	1	SI
(v_6, v_7)	1	SI
(v_1, v_2)	2	SI
(v_3, v_4)	2	SI
(v_2, v_4)	3	NO
(v_1, v_3)	4	NO
(v_4, v_7)	4	SI
(v_3, v_6)	5	NO
(v_5, v_7)	6	SI
(v_4, v_5)	7	NO
(v_4, v_6)	8	NO
(v_2, v_5)	10	NO

Utilización de conjuntos disjuntos

- En cualquier momento del proceso, dos vértices pertenecen al mismo conjunto si y sólo si, están conectados en el bosque de expansión actual
 - Relación de equivalencia: “**estar conectado**”
 - Inicialmente, cada vértice en su propio conjunto: ningún vértice conectado ($crea(B)$)
 - Según se aceptan aristas, se van conectando vértices. Cada vez que se conectan dos vértices ($union(u, v, B)$) quedan en el mismo conjunto
 - Solo se acepta una nueva arista (u, v) , si u y v no están en el mismo conjunto ($buscar(u, B) <> buscar(v, B)$)
 - Si los vértices que conecta esa arista están en el mismo conjunto, significa que ya están conectados, añadirla crearía un ciclo
 - Cada vez que se agrega una arista (u, v) al bosque los vértices del conjunto u quedan conectados con los vértices del conjunto v
 - Si x esta conectado con u y w esta conectado con v . Agregar la arista (u, v) significa que x y w pasan a estar conectados

Algoritmo de Kruskal

Árbol de expansión mínimo

Algoritmo Kruskal(**ref** g: tipoGrafo):tipoGrafo

- 1: A: tipoMontículo
- 2: numAristasAceptadas: entero
- 3: B: tipoPartición
- 4: conjuntoU, conjuntoV: tipoConjunto
- 5: x: tipoElemento
- 6: arbolExp: tipoGrafo
- 7: crear(B)
- 8: construirMontículoDeAristas(g,A)
- 9: numAristasAceptadas \leftarrow 0
- 10: **mientras** numAristasAceptadas $<$ g.orden -1 **hacer**
- 11: x \leftarrow eliminarMin(A)
- 12: conjuntoU \leftarrow buscar(x.informacion.u,B)
- 13: conjuntoV \leftarrow buscar(x.informacion.v,B)
- 14: **si** conjuntoU \neq conjuntoV **entonces**
- 15: unir(conjuntoU, conjuntoV, B)
- 16: numAristasAceptadas \leftarrow numAristasAceptadas + 1
- 17: aceptarArista(x,arbolExp)
- 18: **fin si**
- 19: **fin mientras**
- 20: **devolver** arbolExp

Análisis

- El algoritmo de Kruskal se puede implantar para que se ejecute en un tiempo $O(a \log n)$
 - Si el grafo tiene a aristas, construir el montículo de aristas lleva un tiempo en $O(a \log a)$, que puede mejorarse a $O(a)$
 - En el peor de los casos, que sea necesario revisar todas las aristas del grafo, las operaciones en la cola de prioridad llevan un tiempo $O(a \log a)$. Normalmente no es necesario revisarlas todas para obtener el árbol de expansión mínimo
 - El tiempo total requerido por las operaciones buscar/unir en la estructura partición, depende del método utilizado en su implantación, sabemos que hay métodos en $O(a \log a)$ y $O(a\alpha(a))$
 - El tiempo de ejecución por tanto está en $O(a \log n)$ ya que para grafos dispersos $a < n^2$ y por tanto $\log a < 2 \log n$

Contenido

4

Tema 4. Grafos

- Nivel Abstracto o de definición
- Nivel de representación o implementación
- Recorridos
- Ordenación Topológica
- Caminos Mínimos
- Árbol de expansión de coste mínimo
- Ejercicios

Ejercicio 1

Razonar brevemente que información nos aporta la siguiente tabla, teniendo en cuenta que representa parte del directorio de vértices de un grafo después de aplicar el algoritmo de **Dijkstra**. El razonamiento implica establecer los caminos mínimos obtenidos: el coste de cada camino y la secuencia de vértice que forman cada camino.

En la tabla sólo se representa la información del directorio de vértices relevante para el algoritmo de Dijkstra: identificador de vértice, distancia y anterior

vértice	distancia	anterior
1	0	0
2	5	4
3	3	1
4	4	3
5	6	4

Ejercicio 2

Dada la representación en memoria que se muestra en la siguiente figura y que se corresponde con un grafo después de aplicar el algoritmo de **Prim**, donde:

- sólo se representa la información del directorio de vértices relevante para el algoritmo de Prim: identificador de vértice (v), coste o peso (p) y anterior (a)
- las celdas de las listas de adyacencia tienen dos campos de información el primero sobre el vértice adyacente y el segundo sobre el peso del arco

Teniendo en cuenta el contenido de esta estructura de datos, se pide:

- Dibujar el grafo
- Dibujar el árbol de expansión

V	C	A	
1	2	4	$\rightarrow [2 \ 5] \rightarrow [3 \ 4] \rightarrow [4 \ 2]$
2	1	4	$\rightarrow [1 \ 5] \rightarrow [4 \ 1] \rightarrow [5 \ 17]$
3	3	4	$\rightarrow [1 \ 4] \rightarrow [4 \ 3] \rightarrow [6 \ 5]$
4	7	5	$\rightarrow [1 \ 2] \rightarrow [2 \ 1] \rightarrow [3 \ 3] \rightarrow [5 \ 7] \rightarrow [6 \ 6] \rightarrow [7 \ 1]$
5	0	0	$\rightarrow [2 \ 17] \rightarrow [4 \ 7] \rightarrow [7 \ 15]$
6	5	3	$\rightarrow [3 \ 5] \rightarrow [4 \ 6] \rightarrow [7 \ 10]$
7	1	4	$\rightarrow [4 \ 1] \rightarrow [5 \ 15] \rightarrow [6 \ 10]$

Ejercicio 3

- a) Proponer un algoritmo en pseudocódigo que determine si un grafo es conexo mediante clases de equivalencia

Algoritmo conexo(**ref g:** tipoGrafo):lógico

- b) Implementar el algoritmo en C

Idea básica

- Si dos vértices del grafo **están conectados** (relación de equivalencia) pertenecen a la misma clase de equivalencia, por tanto:
 - Partiendo de una situación inicial en que todos los vértices están desconectados (`crea(B)`) se recorre el grafo conectando (uniendo) vértices entre los que existe una arista
 - Si todos los vértices del grafo acaban en la misma clase de equivalencia (**están conectados**) el grafo es **conexo**
 - Si hay varias clases de equivalencia el grafo no es conexo

Aplicación RAED

La aplicación RAED permite estudiar y analizar los diferentes algoritmos aplicados a distintos grafos ejemplo

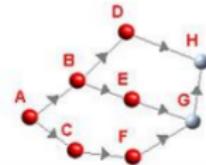
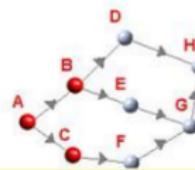
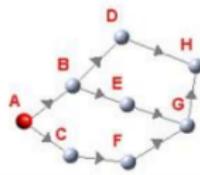


 Tabla de vértices :

VERTICE	ALCANZADO	DISTANCIA	ANTERIOR
A	true	0	-1
B	false	1	0
C	false	1	0
D	false	@	-1
E	false	@	-1
F	false	@	-1
H	false	@	-1
G	false	@	-1

 Tabla de vértices :

VERTICE	ALCANZADO	DISTANCIA	ANTERIOR
A	true	0	-1
B	true	1	0
C	true	1	0
D	false	2	1
E	false	2	1
F	false	2	2
H	false	@	-1
G	false	@	-1

 Tabla de vértices :

VERTICE	ALCANZADO	DISTANCIA	ANTERIOR
A	true	0	-1
B	true	1	0
C	true	1	0
D	true	2	1
E	true	2	1
F	true	2	2
H	false	3	3
G	false	3	4

Contenido

- 1 Tema1. Árboles Generales y Binarios
- 2 Tema 2. Montículos Binarios
- 3 Tema 3. Conjuntos Disjuntos
- 4 Tema 4. Grafos
- 5 Tema 5. Árboles Binarios de Búsqueda
- 6 Tema 6. Organización de archivos
- 7 Tema 7. Organización de Índices

Contenido

5 Tema 5. Árboles Binarios de Búsqueda

- Nivel abstracto o de definición
- Nivel de representación e implementación
- Árboles Balanceados
- Inserción y Equilibrio del árbol
- Eliminación y equilibrio del árbol
- Ejercicios

1 Nivel abstracto o de definición

- Aplicación común de árboles binarios: **Recuperación de información**
⇒ **Árboles Binarios de BÚSQUEDA**
- **Requisito:** los nodos del árbol deben estar ordenados según el valor de alguno de sus campos de información (clave)

Definición formal de Árbol Binario de Búsqueda

Árbol binario que o bien es nulo o cada nodo contiene una clave que satisface las siguientes condiciones:

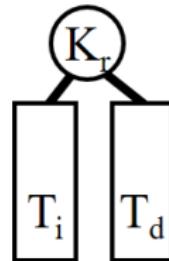
- ① Todas las claves, si las hay, en el subárbol izquierdo de la raíz preceden a la clave de la raíz

$$K_i < K_r \quad \forall K_i \in T_i$$

- ② La clave de la raíz precede a todas las claves, si las hay, que contiene el subárbol derecho

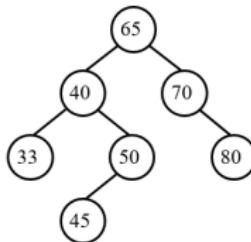
$$K_r < K_d \quad \forall K_d \in T_d$$

- ③ Los subárboles izquierdo y derecho de la raíz son también árboles binarios de búsqueda



(Esta definición puede modificarse para admitir claves duplicadas)

Características



- Existen diferentes formas de ordenar un conjunto de claves para conseguir un árbol binario de búsqueda
- Una vez decidida la clave que se inserta en primer lugar las propiedades del árbol determinan donde deben insertarse las siguientes
- La estructura de un árbol binario de búsqueda particular está determinada por el orden en que se insertan los nodos en el árbol
- Un nuevo nodo siempre se inserta como nodo hoja, a no ser que se permita reestructurar el árbol durante el proceso de inserción
- El recorrido **en orden** de un árbol binario de búsqueda da lugar a una clasificación ascendente de los nodos según el valor de su campo clave

Operaciones básicas sobre árboles binarios de búsqueda

- **búsqueda(k , A , n)**: busca un nodo con valor de clave k en el árbol binario de búsqueda A y devuelve la posición de ese nodo en el árbol si lo encuentra o nulo en otro caso
- **inserción(n , A)**: añade el nodo n al árbol binario de búsqueda A . Después de la inserción A continuará siendo un árbol binario de búsqueda
- **eliminación(k , A)**: suprime el nodo con valor k en su campo clave del árbol binario de búsqueda A si existe. Después de la eliminación A continuará siendo un árbol binario de búsqueda

Contenido

5 Tema 5. Árboles Binarios de Búsqueda

- Nivel abstracto o de definición
- Nivel de representación e implementación
- Árboles Balanceados
- Inserción y Equilibrio del árbol
- Eliminación y equilibrio del árbol
- Ejercicios

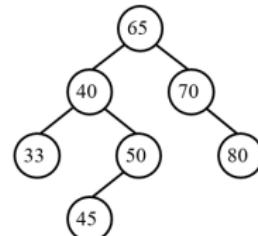
2 Nivel de representación e implementación

Algoritmo declaraciones básicas

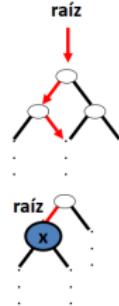
```

tipos
tipoNodo = registro
clave: tipoClave
información: tipoInformación
izq,der:↑ tipoNodo
fin registro
tipoÁrbol: ↑tipoNodo
punteroNodo: ↑tipoNodo

```



- Recuperamos las declaraciones básicas del tema 1 añadiendo el campo clave necesario para la clasificación
- La definición de a.b.b implica la existencia de procedimientos **recursivos**, partiendo de un puntero a la raíz del árbol, para
 - encontrar en el árbol un nodo con un valor de clave dado
 - encontrar la posición en la que debe insertarse un nuevo nodo
 - eliminar del árbol un nodo con un valor de clave dado



2.1 Búsqueda en árboles binarios de búsqueda

- Procedimiento para encontrar un nodo con valor k para la clave en un a.b.b. con raíz R y clave de la raíz k_R
 - Si el árbol está vacío la búsqueda termina sin éxito
 - Si $k = k_R$ la búsqueda termina satisfactoriamente. El nodo buscado es la raíz del árbol
 - Si $k < k_R$ se sigue la búsqueda en el subárbol izquierdo de la raíz
 - Si $k > k_R$ se sigue la búsqueda en el subárbol derecho de la raíz
- Procedimiento que puede implementarse de forma recursiva, donde inicialmente R apunta a la raíz del árbol

Algoritmo **búsqueda**(k:tipoClave, **raíz**:tipoÁrbol, ref nodo:tipoNodo)

Entrada: *raíz* dirección del nodo raíz y *k* clave de búsqueda

Salida: *nodo* dirección del nodo con clave *k* si existe y *NULO* en caso contrario

```
1: si raíz = NULO entonces
2:   nodo ← NULO
3: si no
4:   si k = raíz↑.clave entonces
5:     nodo ← raíz
6:   si no
7:     si k < raíz↑.clave entonces
8:       búsqueda(k, raíz↑.izq,nodo)
9:     si no
10:      búsqueda(k, raíz↑.der,nodo)
11:   fin si
12: fin si
13: fin si
```

Comportamiento del algoritmo de búsqueda

- Se analizan el número de comparaciones realizadas antes de terminar la búsqueda
- El comportamiento del algoritmo depende de la profundidad del nodo que contiene la clave buscada en el árbol. Cuanto más lejos se encuentre de la raíz peor será
- ¿Cómo se puede mejorar el comportamiento?
 - Organizando el árbol de forma que las claves buscadas con mayor frecuencia estén situadas tan cerca como sea posible de la raíz. Esto se consigue insertándolas en el árbol en el orden apropiado
 - Problemas:
 - deben conocerse las probabilidades de acceso
 - cambiarán a medida que se vayan insertando nuevos nodos
 - se deben tener en cuenta los efectos de una búsqueda sin éxito
- En general el número de comparaciones se reduce cuando la altura del árbol es mínima ⇒ **Árboles Balanceados**

2.2 Inserción en árboles binarios de búsqueda

- Procedimiento para insertar un nodo con valor k para la clave en un a.b.b. con raíz R y clave de la raíz k_R
 - Si el árbol está vacío el nodo con clave k será la nueva raíz
 - Si $k = k_R$ la inserción no puede hacerse (ya existe un nodo con clave k)
 - Si $k < k_R$ se recorre el subárbol izquierdo de la raíz hasta encontrar la posición adecuada para insertar el nuevo nodo
 - Si $k > k_R$ se recorre el subárbol derecho de la raíz hasta encontrar la posición adecuada para insertar el nuevo nodo
- Procedimiento, similar al de búsqueda, que puede implementarse de forma recursiva, donde inicialmente R apunta a la raíz del árbol

Algoritmo **insertar**(nuevo:tipoNodo; **ref** **raíz**:tipoÁrbol)

Entrada: nuevo nodo a insertar en árbol raíz

Salida: el árbol $raíz$ con el nodo nuevo insertado correctamente

```
1: si raíz = NULO entonces
2:   raíz ← nuevo
3: si no
4:   si nuevo↑.clave = raíz↑.clave entonces
5:     /* clave duplicada implementar según especificacion */
6:   si no
7:     si nuevo↑.clave < raíz↑.clave entonces
8:       insertar(nuevo, raíz↑.izq)
9:     si no
10:      insertar(nuevo, raíz↑.der)
11:    fin si
12:  fin si
13: fin si
```

Observaciones

- El parámetro *nuevo* es un puntero al nodo que ha de ser insertado en el a.b.b
- La condición que causa que el procedimiento de búsqueda termine con éxito es la misma que causa que el procedimiento de inserción termine sin éxito
- La inserción ordenada de claves en un a.b.b. produce un árbol largo sin ramificaciones (lista de nodos)
- El orden en que se insertan las claves influye en la altura del árbol y, por tanto, en el comportamiento del algoritmo de búsqueda
- Puede mejorarse este comportamiento reacomodando nodos en el proceso de inserción ⇒ **Árboles Balanceados**
- Podría permitirse la inserción de claves duplicadas

2.3 Eliminación en árboles binarios de búsqueda

- Eliminación de un nodo del árbol de forma que no viole los principios que lo definen: **el árbol resultante después de la eliminación será un a.b.b**
- Procedimiento para eliminar el nodo con valor k para la clave en un a.b.b. con raíz R y clave de la raíz k_R
 - ① Localizar el nodo que se desea eliminar siguiendo el mismo método que en el procedimiento de búsqueda
 - ② Distinguir los siguientes casos:
 - caso 1** Si el nodo a eliminar es hoja o terminal, simplemente se suprime
 - caso 2** Si el nodo a eliminar tiene un solo descendiente, se sustituye por ese descendiente y se suprime
 - caso 3** Si el nodo a eliminar tiene los dos descendientes, entonces se sustituye por el nodo más a la izquierda del subárbol derecho o por el **nodo más a la derecha del subárbol izquierdo**, eliminándose el nodo sustituido

Algoritmo de eliminación en a.b.b.

Primer bosquejo

Algoritmo **eliminar**(x:tipoClave; ref raíz:tipoÁrbol)

Entrada: x clave del nodo a eliminar del árbol $raíz$

Salida: el árbol $raíz$ con el nodo eliminado si existe

si $raíz = \text{NULL}$ entonces

/*no existe nodo con clave x : implementar según especificación*/

si no

si $x < raíz^.clave$ entonces

eliminar($x, raíz^.izq$)

si no

si $x > raíz^.clave$ entonces

eliminar($x, raíz^.der$)

si no

$aux \leftarrow raíz$

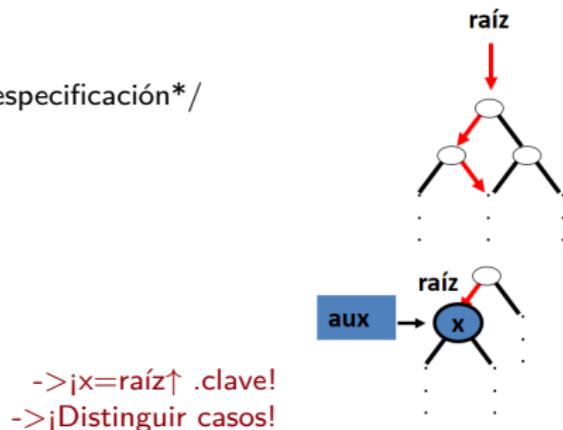
.....

eliminarNodo(aux)

fin si

fin si

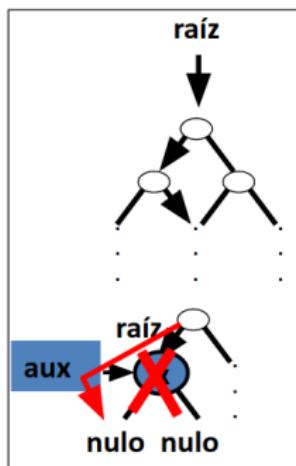
fin si



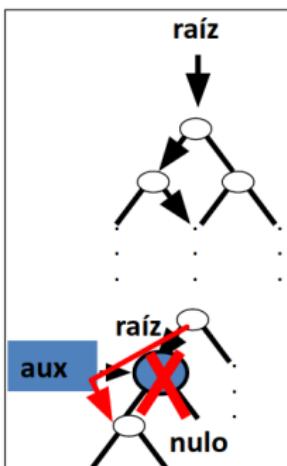
Distinción de casos: un descendiente o ninguno

Ningún hjo	Sólo hijo izquierdo	Sólo hijo derecho
$\text{aux} \uparrow.\text{izq} = \text{NULO}$	$\text{aux} \uparrow.\text{izq} \neq \text{NULO}$	$\text{aux} \uparrow.\text{izq} = \text{NULO}$
$\text{aux} \uparrow.\text{der} = \text{NULO}$	$\text{aux} \uparrow.\text{der} \neq \text{NULO}$	$\text{aux} \uparrow.\text{der} \neq \text{NULO}$

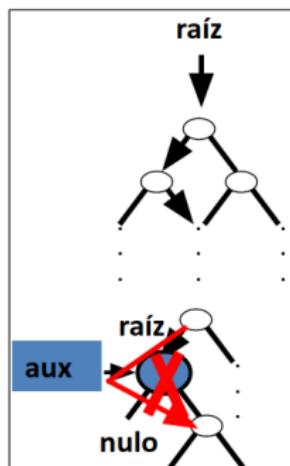
Ningún hijo



Sólo hijo Izquierdo



Sólo hijo Derecho


 $\text{raíz} \leftarrow \text{aux} \uparrow.\text{izq}$
 $\text{raíz} \leftarrow \text{aux} \uparrow.\text{izq}$
 $\text{raíz} \leftarrow \text{aux} \uparrow.\text{der}$

Distinción de casos: un descendiente o ninguno

Algoritmo **eliminar**(x:tipoClave; **ref** **raíz**:tipoÁrbol)

Entrada: x clave del nodo a eliminar del árbol raíz

Salida: el árbol raíz con el nodo eliminado si existe

si **raíz** = NULO **entonces**

 /*no existe nodo con clave x: implementar según especificación*/

si no

si x < **raíz**.clave **entonces**

eliminar(x,**raíz**.izq)

si no

si x > **raíz**.clave **entonces**

eliminar(x,**raíz**.der)

si no

 aux ← **raíz**

si aux.der = NULO **entonces**

 raíz ← aux.izq

->sólo hijo izquierdo o ningun descendiente

si no

si aux.izq = NULO **entonces**

 raíz ← aux.der

->sólo hijo derecho

si no

 ...

->dos hijos

fin si

fin si

eliminarNodo(aux)

fin si

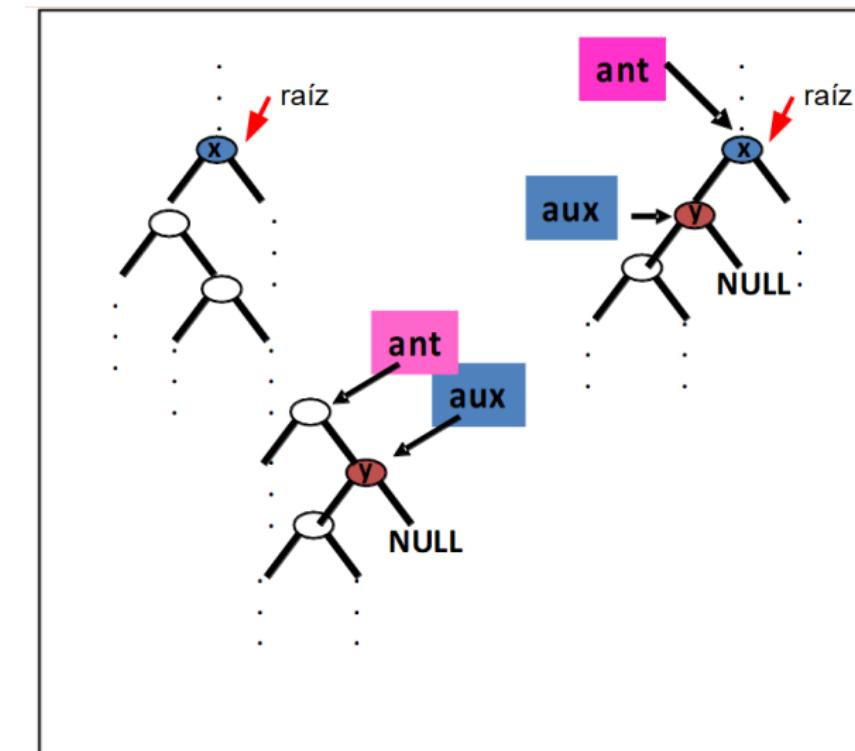
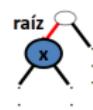
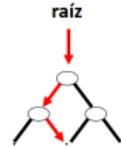
fin si

fin si

Distinción de casos: dos descendientes

buscar nodo más derecha de subárbol izquierdo

$\text{aux} \uparrow .\text{izq} \neq \text{NULL}$
 $\text{aux} \uparrow .\text{der} \neq \text{NULL}$

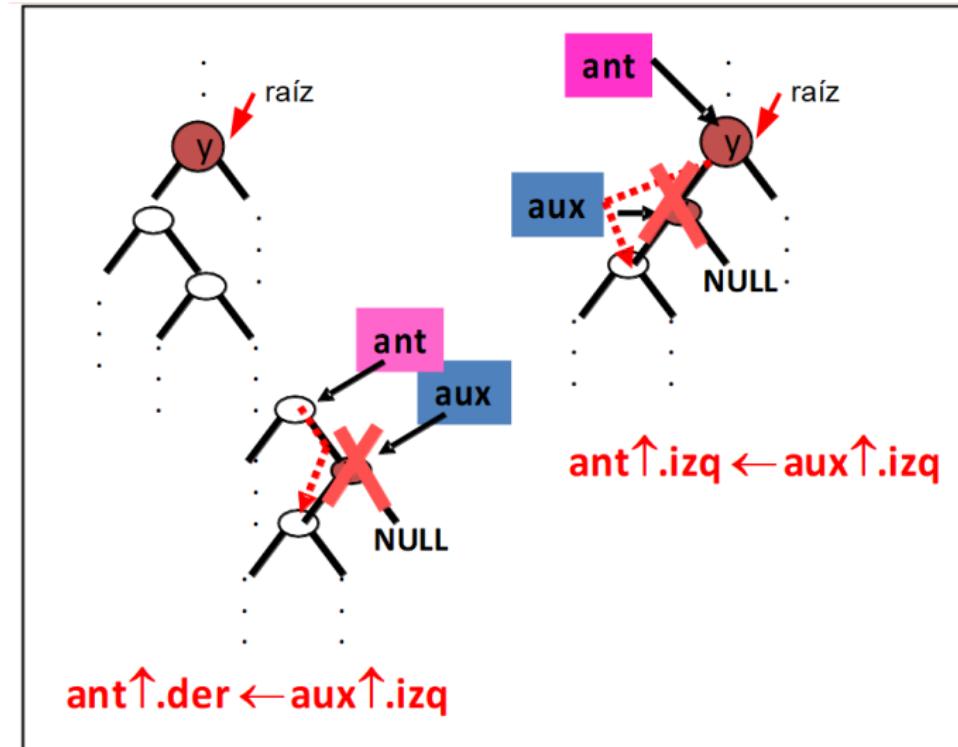
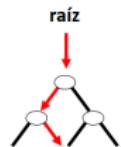


Distinción de casos: dos descendientes

sustituir por nodo más derecho de subárbol izquierdo

$\text{aux} \uparrow .\text{izq} \neq \text{NULL}$

$\text{aux} \uparrow .\text{der} \neq \text{NULL}$



Fragmento algoritmo eliminación: dos descendientes

Algoritmo eliminar(x:tipoClave, referencia raíz:tipoÁrbol)

```
aux ← raíz                                ->raíz↑.clave= x ¡nodo a eliminar!
...
si no
    ant ← aux                                ->dos hijos
    aux ← aux↑.izq                            ->buscar nodo más derecha sub.izquierdo
mientras aux↑.der ≠ NULO hacer
    ant ← aux
    aux ← aux↑.der
fin mientras
    raíz↑.clave ← aux↑.clave                ->sustituir información
    raíz↑.info ← aux↑.info
si ant = raíz entonces
    ant↑.izq ← aux↑.izq                    ->enlazar subárboles de aux antes de eliminar
si no
    ant↑.der ← aux↑.izq
fin si
...
eliminarNodo(aux)
```

Algoritmo eliminar(x:tipoClave; ref raíz:tipoÁrbol)

```

1: si raíz = NULO entonces
2:   /*no existe nodo con clave x: implementar según especificación*/
3: si no
4:   si x < raíz↑.clave entonces
5:     eliminar(x,raíz↑ .izq)
6:   si no
7:     si x > raíz↑.clave entonces
8:       eliminar(x,raíz↑.der)
9:     si no
10:    aux ← raíz
11:    si aux↑.der = NULO entonces
12:      raíz ← aux↑.izq
13:    si no
14:      si aux↑.izq = NULO entonces
15:        raíz ← aux↑.der
16:      si no
17:        ant ← aux
18:        aux ← aux↑.izq
19:        mientras aux↑.der ≠ NULO hacer
20:          ant ← aux
21:          aux ← aux↑.der
22:        fin mientras
23:        raíz↑.clave ← aux↑.clave
24:        raíz↑.info ← aux↑.info
25:        si ant = raíz entonces
26:          ant↑.izq ← aux↑.izq
27:        si no
28:          ant↑.der ← aux↑.izq
29:        fin si
30:      fin si
31:    fin si
32:    eliminarNodo(aux)
33:  fin si
34: fin si
35: fin si

```

->raíz↑.clave= x ¡nodo a eliminar!

->sólo hijo izquierdo o ningún hijo

->sólo hijo derecho

->dos hijos

->buscar nodo más derecho sub.izquierdo

->sustituir información

->enlazar subárboles de aux antes de eliminar

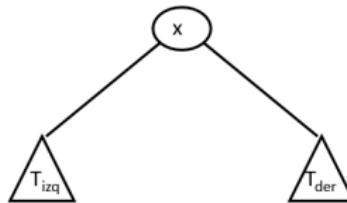
2.4 Análisis del caso promedio

- Puesto que el tiempo en descender un nivel en el árbol es constante, el tiempo de ejecución de las tres operaciones vistas será $O(d)$, siendo d la **profundidad** del nodo que contiene la clave de búsqueda
- Para cada nodo del árbol, el numero de comparaciones viene dado por su profundidad o distancia desde la raíz a ese nodo
- La suma de estas distancias para todos los nodos se denomina **longitud de camino interno** del árbol
- Dividiendo la longitud de camino interno por el número de nodos se obtendrá el **número medio de comparaciones** para una búsqueda con éxito
- Si todos los árboles son igualmente probables, la profundidad media en todos los nodos de un árbol es **$O(\log n)$** y por tanto también lo será el tiempo de ejecución de una operación de búsqueda

Longitud media de camino interno

- Un a.b.b. aleatorio de n nodos, para $0 \leq i < n$, consta de
 - Una raíz
 - Un subárbol izquierdo de i nodos.
 - Un subárbol derecho de $n - i - 1$ nodos
- Si $D(n)$ es la longitud de camino interno de un árbol de n nodos

$$D(n) = D(i) + D(n - i - 1) + (n - 1)$$



- El término $(n-1)$ tiene en cuenta el hecho de que la raíz contribuye con 1 a la longitud del camino para cada uno de los $n - 1$ nodos restantes del árbol
- Si todos los tamaños de subárboles son igualmente probables \Rightarrow el valor promedio de $D(i)$ y $D(n - i - 1)$ es

$$\frac{1}{n} \sum_{j=0}^{n-1} D(j)$$

Profundidad esperada de un nodo cualquiera

- Se obtiene, por tanto, la longitud media de camino interno de un árbol de n nodos

$$D(n) = \frac{2}{n} \sum_{j=0}^{n-1} D(j) + (n - 1)$$

- Misma recurrencia que aparece en el análisis del algoritmo de ordenación rápida (quicksort), donde se obtuvo que

$$D(n) \in O(n \log n)$$

- Por tanto la profundidad esperada de cualquier nodo es $O(\log n)$
- ¿Son todos los árboles igualmente probables?
 - La inserción ordenada de claves produce una lista de nodos
 - El algoritmo de eliminación favorece la creación de subárboles izquierdos
- Solución: añadir una condición estructural que evite profundidades excesivas en los nodos

Contenido

5 Tema 5. Árboles Binarios de Búsqueda

- Nivel abstracto o de definición
- Nivel de representación e implementación
- Árboles Balanceados
- Inserción y Equilibrio del árbol
- Eliminación y equilibrio del árbol
- Ejercicios

3 Árboles Balanceados

- Los árboles balanceados o árboles AVL (Adelson-Velskii, Laudis) tratan de mejorar el comportamiento del algoritmo de búsqueda en a.b.b. realizando “reacomodos” de nodos después de las inserciones y eliminaciones
- Evitan que el árbol pueda “crecer” o “decrecer” descontroladamente

Definición formal

Un árbol balanceado es un a.b.b. en el cual para todo nodo n_i se cumple la siguiente condición:

“La altura del subárbol izquierdo de n_i y la altura del subárbol derecho de n_i difieren como máximo en una unidad”

- Para determinar si un árbol está o no balanceado se necesita información relativa al equilibrio de cada nodo del árbol \Rightarrow factor de equilibrio.

Factor de equilibrio de un nodo

Diferencia entre las alturas de sus subárboles derecho e izquierdo

$$fe = h_d - h_i$$

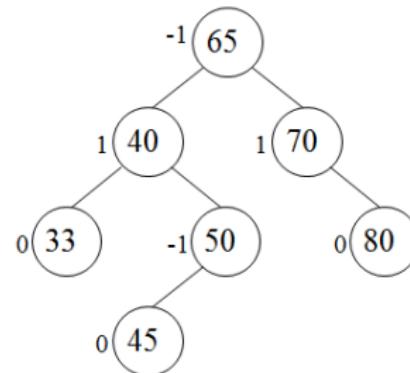
- Valores posibles en un árbol balanceado: **-1, 0, 1**
- Para evitar que el factor de equilibrio llegue a tomar valores -2 ó 2 el árbol deberá **reestructurarse**

Algoritmo declaraciones básicas

```

tipos
tipoNodo = registro
    clave: tipoClave
    información: tipoInformación
    fe: -1 .. 1
    izq, der: ↑tipoNodo
fin registro
tipoÁrbol: ↑tipoNodo
punteroNodo: ↑tipoNodo

```



Contenido

5 Tema 5. Árboles Binarios de Búsqueda

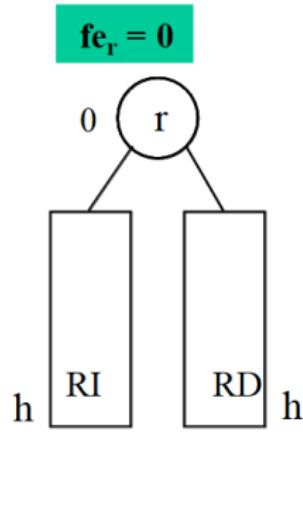
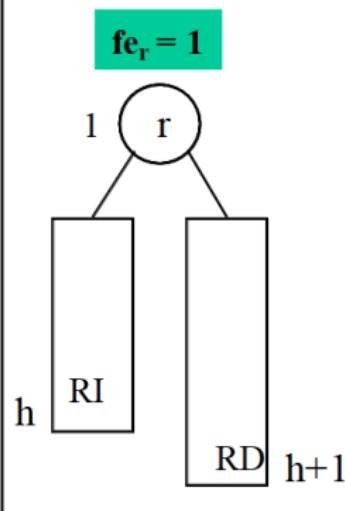
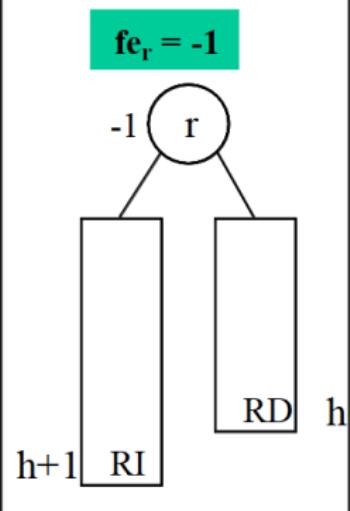
- Nivel abstracto o de definición
- Nivel de representación e implementación
- Árboles Balanceados
- **Inserción y Equilibrio del árbol**
- Eliminación y equilibrio del árbol
- Ejercicios

4 Inserción y equilibrio del árbol

- Siguiendo el algoritmo de búsqueda en a.b.b, se insertará el nodo en el subárbol izquierdo o derecho, según corresponda
- Casos a tener en cuenta:
 - El nuevo nodo se inserta sin modificar la altura del subárbol en que se inserta ⇒ ni la altura de la raíz ni el equilibrio del árbol se modifican
 - El nuevo nodo se inserta aumentando la altura del subárbol más corto ⇒ tampoco se perderá el equilibrio del árbol
 - El nuevo nodo se inserta aumentando la altura del subárbol más largo ⇒ el árbol perderá el equilibrio
- Para distinguir estos casos, partiremos de las diferentes situaciones antes de la inserción y estudiaremos todas las posibilidades que pueden presentarse ante una inserción de un nuevo nodo

Distinción de casos antes de la inserción

- Las ramas izquierda y derecha tienen la misma altura
- La altura de la rama izquierda es menor que la altura de la rama derecha
- La altura de la rama izquierda es mayor que la altura de la rama derecha

Caso 1**Caso 2****Caso 3**

Proceso de Inserción

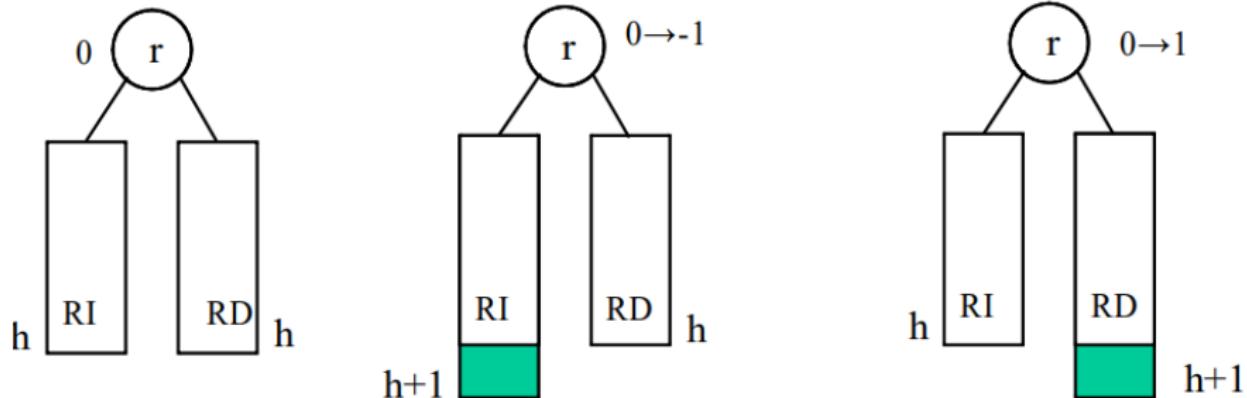
- ① Localizar la posición del árbol donde debe insertarse el nuevo nodo utilizando el mismo método que en la inserción del a.b.b, insertar el nodo y calcular su factor de equilibrio (lógicamente será cero)
 - ② **Regresar** por el camino de búsqueda recalculando el factor de equilibrio de **todos** los nodos, siempre que la altura de alguno de sus subárboles haya cambiado. Reestructurar el **subárbol** en aquellos casos en que sea necesario, evitando que el factor de equilibrio tome valores 2 ó -2
- Puede implementarse como un algoritmo recursivo que tiene como parámetros:
 - puntero al nuevo nodo
 - puntero que inicialmente señala a la raíz del árbol, que permitirá seguir el camino de búsqueda
 - un parámetro de tipo lógico que indicará si la altura del subárbol ha cambiado (aumentado) como consecuencia de la inserción

Algoritmo de inserción

Algoritmo insertar(nuevo:tipoNodo; ref cambiaH:lógico; ref nodo:tipoÁrbol)

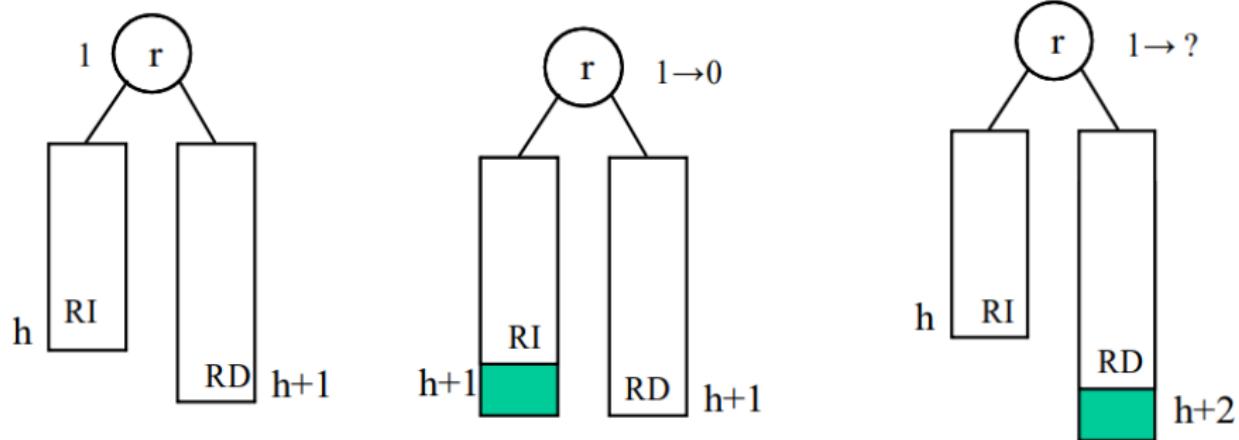
```
1: si nodo = NULO entonces
2:   nodo ← nuevo
3:   cambiaH ← VERDADERO
4: si no
5:   si nuevo↑.clave = nodo↑.clave entonces
6:     /* clave duplicada implementar según especificación */
7:   si no
8:     si nuevo↑.clave < nodo↑.clave entonces
9:       insertar(nuevo,cambiaH, nodo↑.izq)
10:      si cambiaH entonces
11:        equilibrarlzq(nodo,cambiaH)
12:      fin si
13:    si no
14:      insertar(nuevo,cambiaH,nodo↑.der)
15:      si cambiaH entonces
16:        equilibrarDer(nodo,cambiaH)
17:      fin si
18:    fin si
19:  fin si
20: fin si
```

Inserción de un nodo en el caso 1 ($fe = 0$)



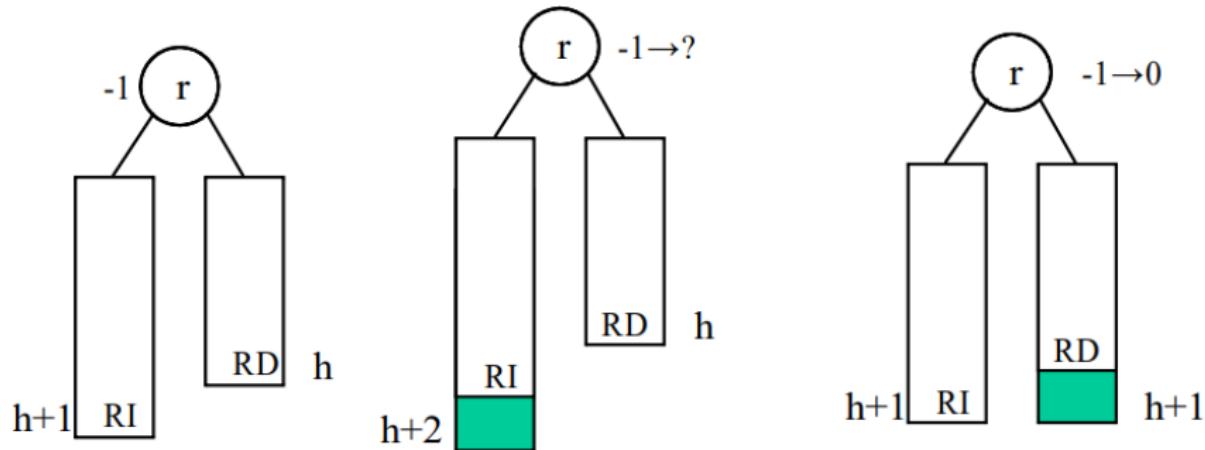
Antes Inserción		Después Inserción por rama Izquierda			Después Inserción por rama Derecha		
nodo [↑] .fe	altura	nodo [↑] .fe	altura	cambiaH	nodo [↑] .fe	altura	cambiaH
0	h+1	-1	h+2	verdadero	1	h+2	verdadero

Inserción de un nodo en el caso 2 ($fe = 1$)



Antes Inserción		Después Inserción por rama Izquierda			Después Inserción por rama Derecha		
nodo [↑] .fe	altura	nodo [↑] .fe	altura	cambiaH	nodo [↑] .fe	altura	cambiaH
0	h+1	-1	h+2	verdadero	1	h+2	verdadero
1	h+2	0	h+2	falso	Reestruct. Sub.Derecho		

Inserción de un nodo en el caso 3 ($fe = -1$)

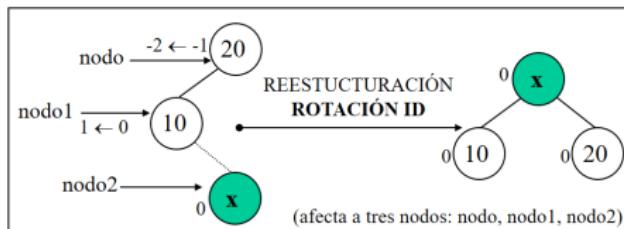
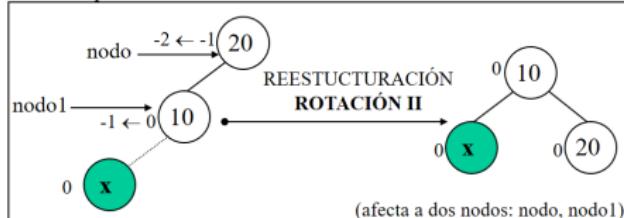


Antes Inserción		Después Inserción por rama Izquierda			Después Inserción por rama Derecha		
nodo↑.fe	altura	nodo↑.fe	altura	cambiaH	nodo↑.fe	altura	cambiaH
0	h+1	-1	h+2	verdadero	1	h+2	verdadero
1	h+2	0	h+2	falso	Reestruct.Sub.Derecho		
-1	h+2	Reestruct.Sub.Izquierdo			0	h+2	falso

4.1 Equilibrar rama izquierda

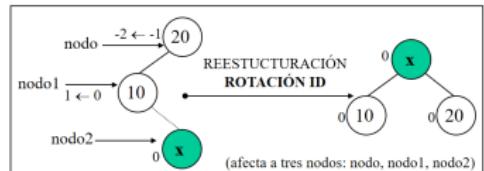
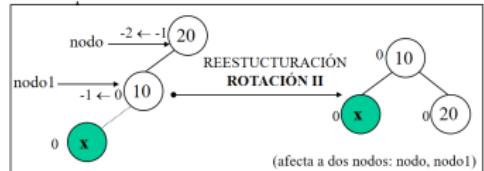
Ejemplo reestructuración subárbol Izquierdo (nodo \uparrow .fe = -1)

Estado después de la inserción:



Antes Inserción		Después Inserción por rama Izquierda			Después Inserción por rama Derecha		
nodo \uparrow .fe	altura	nodo \uparrow .fe	altura	cambiaH	nodo \uparrow .fe	altura	cambiaH
0	h+1	-1	h+2	verdadero	1	h+2	verdadero
1	h+2	0	h+2	falso	Restruct.Sub.Derecho		
-1	h+2	¿II o ID? Depende de FE nodo \uparrow .izq			0	h+2	falso

Antes Inserción		Después Inserción por rama Izquierda		
nodo↑.fe	altura	nodo↑.fe	altura	cambiaH
0	h+1	-1	h+2	verdadero
1	h+2	0	h+2	falso
-1	h+2	nodo1↑.fe = -1 Rotación II nodo1↑.fe = 1 Rotación ID		falso falso



Algoritmo equilibrarIzq(ref nodo: punteroNodo; ref cambiaH:lógico)

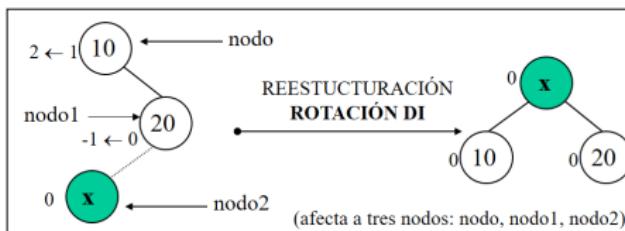
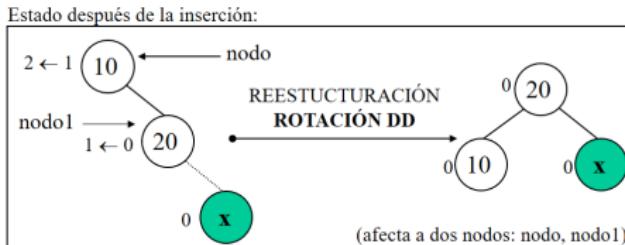
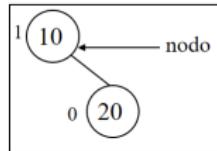
```

1: casos nodo↑.fe en
2:   0: nodo↑.fe ← -1
3:   1: nodo↑.fe ← 0
4:   cambiaH ← FALSO
5:   -1: nodo1 ← nodo↑.izq
6:     si nodo1↑.fe = -1 entonces
7:       rotaciónII(nodo,nodo1)
8:     si no
9:       nodo2 ← nodo1↑.der
10:      rotaciónID(nodo,nodo1,nodo2)
11:    fin si
12:    cambiaH ← FALSO
13: fin casos
  
```

4.2 Equilibrar rama derecha

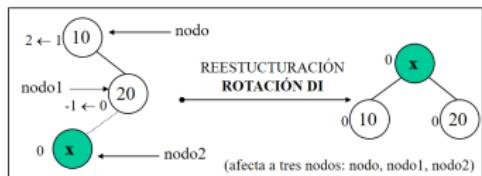
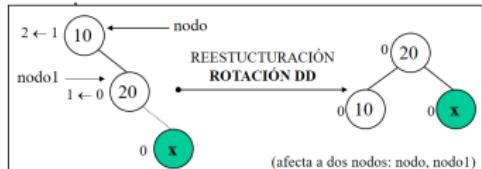
Reestructuración subárbol derecho (nodo \uparrow .fe = 1)

Estado antes de la inserción:



Antes Inserción		Después Inserción por rama Izquierda			Después Inserción por rama Derecha		
nodo \uparrow .fe	altura	nodo \uparrow .fe	altura	cambiaH	nodo \uparrow .fe	altura	cambiaH
0	h+1	-1	h+2	verdadero	1	h+2	verdadero
1	h+2	0	h+2	falso	¿DD o DI? Depende de FE nodo \uparrow .der		
-1	h+2	¿II o ID? Depende de FE nodo \uparrow .izq			0	h+2	falso

Antes Inserción		Después Inserción por rama Derecha		
nodo [↑] .fe	altura	nodo [↑] .fe	altura	cambiaH
0	h+1	1	h+2	verdadero
1	h+2	nodo1[↑].fe = 1 Rotación DD nodo1[↑].fe = -1 Rotación DI		falso falso
-1	h+2	0	h+2	falso



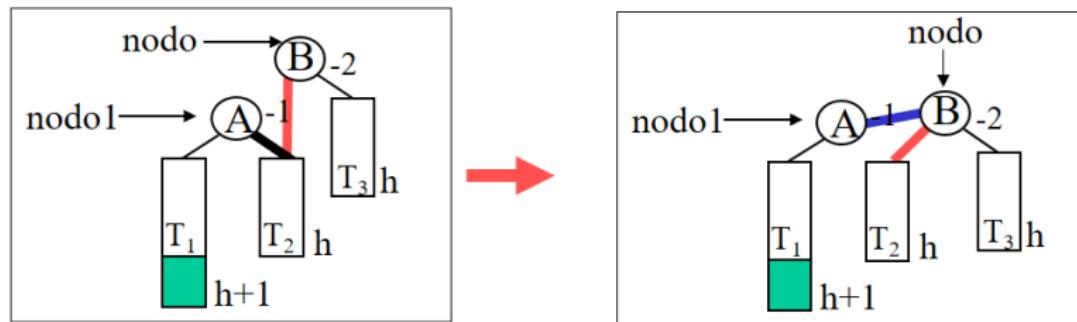
Algoritmo equilibrarDer(ref nodo: punteroNodo; ref cambiaH:lógico)

```

1: casos nodo↑.fe en
2:   0: nodo↑.fe ← 1
3:   1: nodo1 ← nodo↑.der
4:     si nodo1↑.fe = 1 entonces
5:       rotaciónDD(nodo,nodo1)
6:     si no
7:       nodo2 ← nodo1↑.izq
8:       rotaciónDI(nodo,nodo1,nodo2)
9:     fin si
10:    cambiaH ← FALSO
11:    -1: nodo↑.fe ← 0
12:    cambiaH ← FALSO
13: fin casos
  
```

4.3 Rotación Izquierda Izquierda

Movimiento de los nodos en la rotación y cambio factor equilibrio



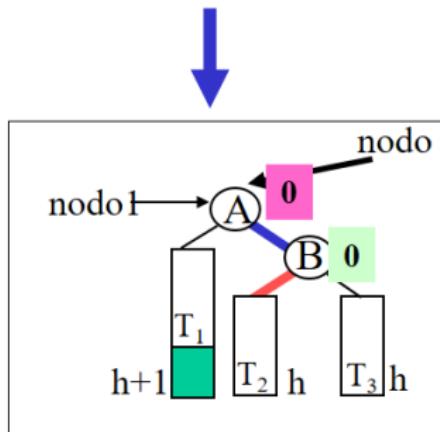
nodo[↑].izq ← nodo1[↑].der

nodo1[↑].der ← nodo

nodo[↑].fe ← 0

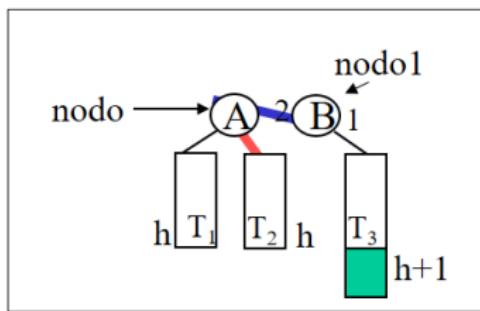
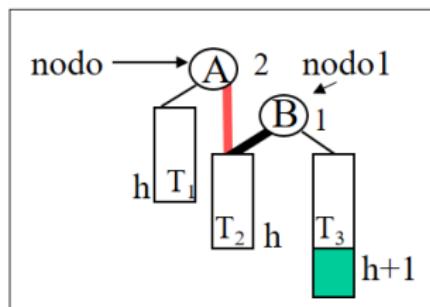
nodo1[↑].fe ← 0

nodo ← nodo1



4.4 Rotación Derecha Derecha

Movimiento de los nodos en la rotación y cambio factor equilibrio



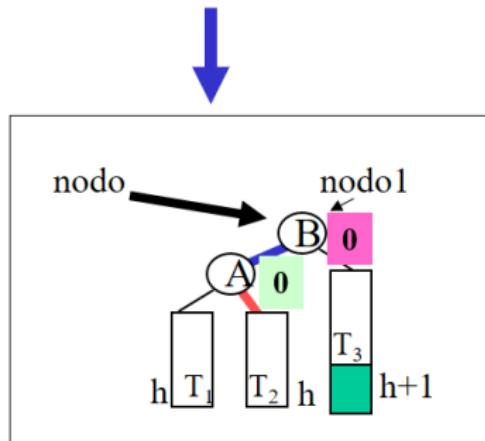
nodo[↑].der ← nodo1[↑].izq

nodo1[↑].izq ← nodo

nodo[↑].fe ← 0

nodo1[↑].fe ← 0

nodo ← nodo1



Algoritmos de rotaciones simples: II y DD

Sólo sirven para proceso de inserción

Algoritmo rotaciónII(**ref nodo: punteroNodo, nodo1: punteroNodo**)

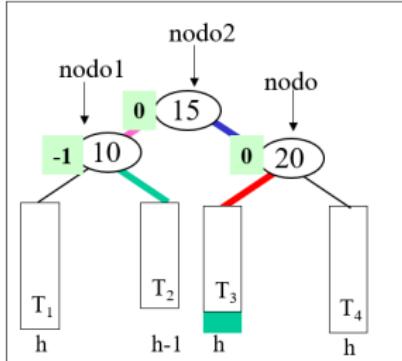
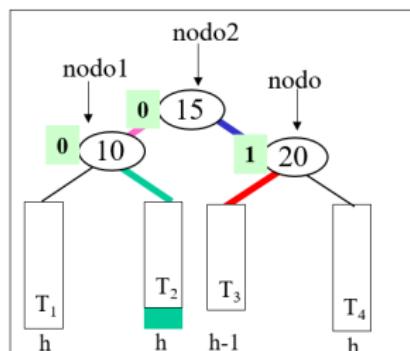
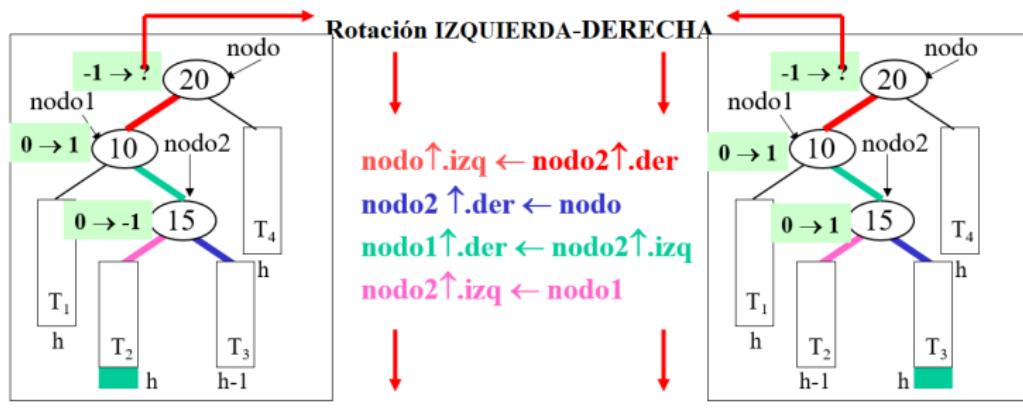
- 1: $nodo \uparrow .izq \leftarrow nodo1 \uparrow .der$
 - 2: $nodo1 \uparrow .der \leftarrow nodo$ ->Movimiento nodos en rotación
 - 3: $nodo \uparrow .fe \leftarrow 0$
 - 4: $nodo1 \uparrow .fe \leftarrow 0$ ->Cambio en los factores de equilibrio
 - 5: $nodo \leftarrow nodo1$ ->Nueva raíz del subárbol
-

Algoritmo rotaciónDD(**ref nodo: punteroNodo, nodo1: punteroNodo**)

- 1: $nodo \uparrow .der \leftarrow nodo1 \uparrow .izq$
 - 2: $nodo1 \uparrow .izq \leftarrow nodo$ ->Movimiento nodos en rotación
 - 3: $nodo \uparrow .fe \leftarrow 0$
 - 4: $nodo1 \uparrow .fe \leftarrow 0$ ->Cambio en los factores de equilibrio
 - 5: $nodo \leftarrow nodo1$ ->Nueva raíz del subárbol
-

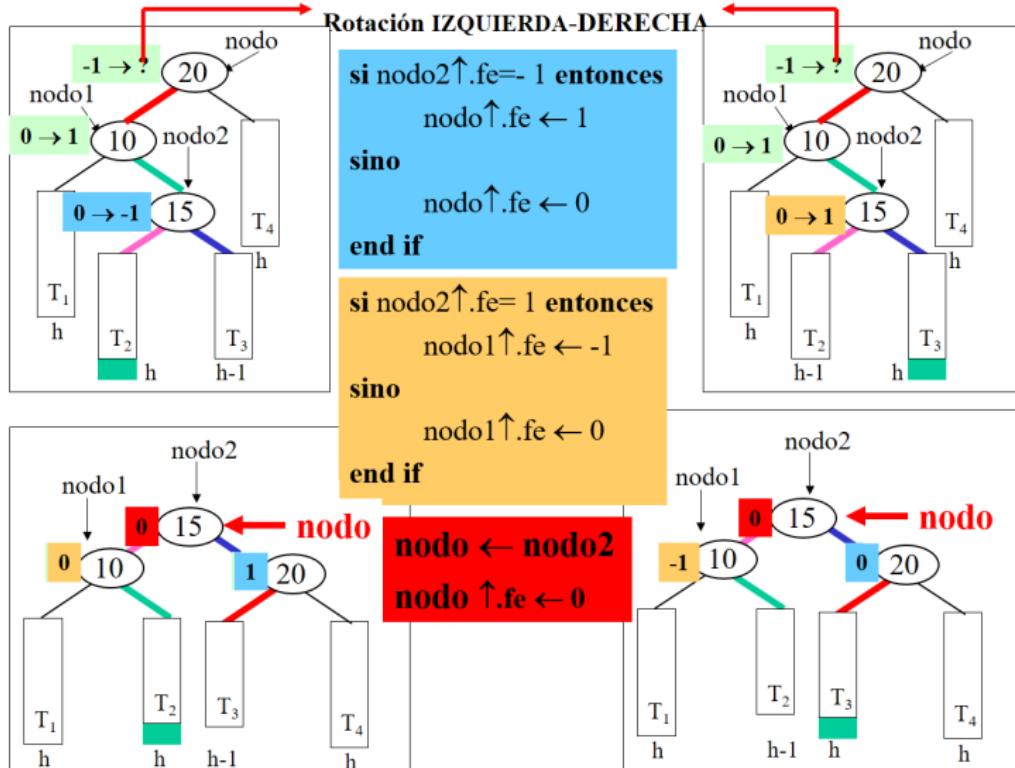
4.5 Rotación Izquierda Derecha

1. Movimiento de los nodos en la rotación



Rotación Izquierda Derecha

2.Cambio de factores de equilibrio



Algoritmo de rotación Izquierda Derecha

Algoritmo rotaciónID(**ref** nodo: punteroNodo, nodo1,nodo2: punteroNodo)

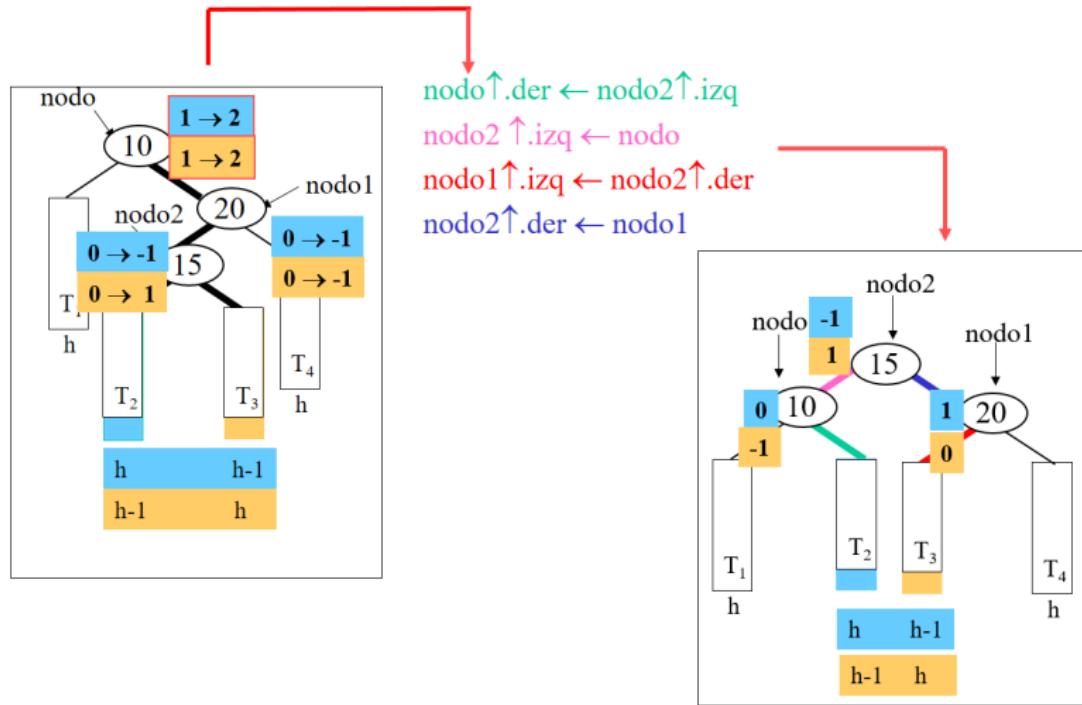
```
1: nodo↑.izq ← nodo2↑.der
2: nodo2↑.der ← nodo
3: nodo1↑.der ← nodo2↑.izq
4: nodo2↑.izq ← nodo1
5: si nodo2↑.fe = -1 entonces
6:     nodo↑.fe ← 1
7: si no
8:     nodo↑.fe ← 0
9: fin si
10: si nodo2↑.fe = 1 entonces
11:     nodo1↑.fe ← -1
12: si no
13:     nodo1↑.fe ← 0
14: fin si
15: nodo ← nodo2
16: nodo↑.fe ← 0
```

->Movimiento nodos en rotación

->Cambio en los factores de equilibrio
->Nueva raíz del subárbol

4.6 Rotación Derecha Izquierda

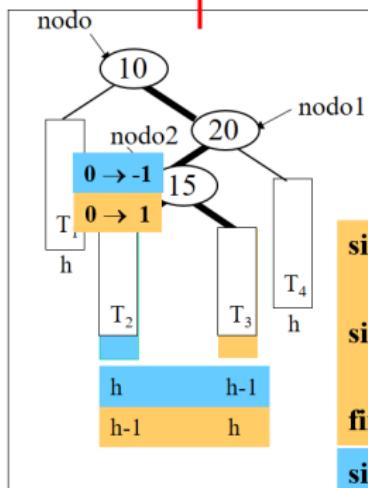
1. Movimiento de los nodos en la rotación



Rotación Derecha Izquierda

2. Cambio de factores de equilibrio

Rotación DERECHA-IZQUIERDA



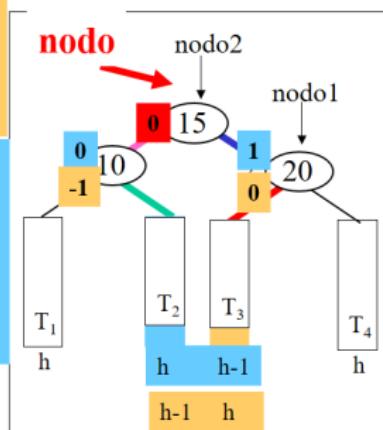
```

nodo↑.der ← nodo2↑.izq
nodo2↑.izq ← nodo
nodo1↑.izq ← nodo2↑.der
nodo2↑.der ← nodo1

si nodo2↑.fe= 1 entonces
    nodo↑.fe ← -1
sino
    nodo↑.fe ← 0
fin

si nodo2↑.fe= -1 entonces
    nodo1↑.fe ← 1
sino
    nodo1↑.fe ← 0
fin

nodo ← nodo2
nodo↑.fe ← 0
  
```



Algoritmo de rotación Derecha Izquierda

Algoritmo rotaciónDI(**ref** nodo: punteroNodo, nodo1,nodo2: punteroNodo)

```
1: nodo↑.der ← nodo2↑.izq
2: nodo2↑.izq ← nodo
3: nodo1↑.izq ← nodo2↑.der
4: nodo2↑.der ← nodo1
5: si nodo2↑.fe = 1 entonces
6:     nodo↑.fe ← -1
7: si no
8:     nodo↑.fe ← 0
9: fin si
10: si nodo2↑.fe = -1 entonces
11:     nodo1↑.fe ← 1
12: si no
13:     nodo1↑.fe ← 0
14: fin si
15: nodo← nodo2
16: nodo↑.fe ← 0
```

->Movimiento nodos en rotación

->Cambio en los factores de equilibrio
->Nueva raíz del subárbol

Contenido

5 Tema 5. Árboles Binarios de Búsqueda

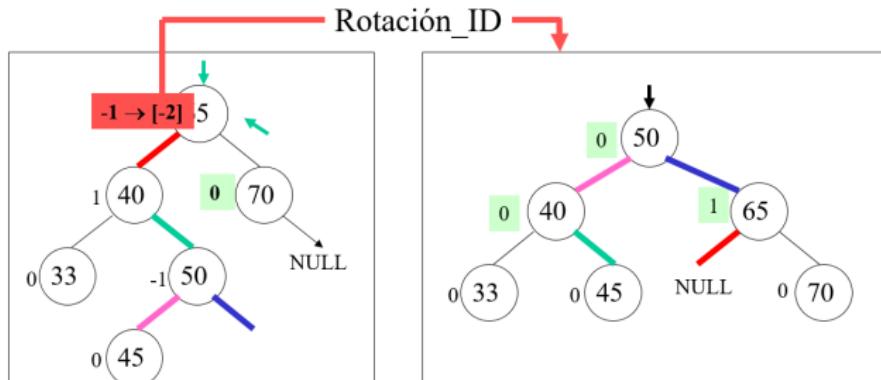
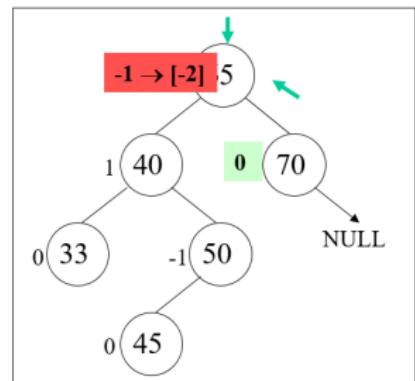
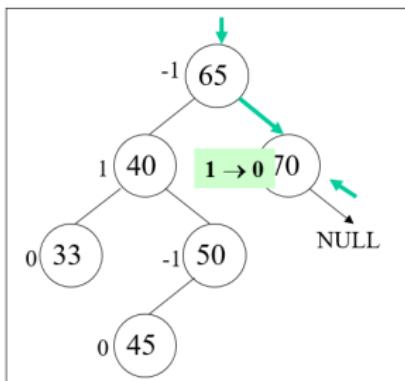
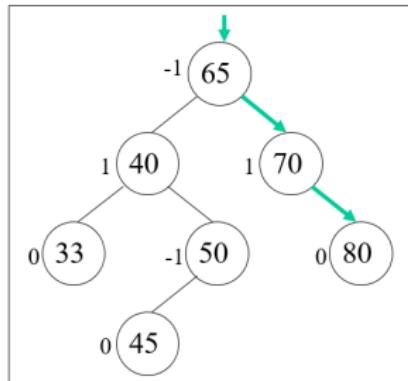
- Nivel abstracto o de definición
- Nivel de representación e implementación
- Árboles Balanceados
- Inserción y Equilibrio del árbol
- Eliminación y equilibrio del árbol
- Ejercicios

5 Eliminación y equilibrio del árbol

- El proceso sigue la misma lógica que el algoritmo de eliminación en a.b.b. añadiéndole las operaciones de reestructuración utilizadas en el algoritmo de inserción en árboles balanceados (rotaciones II, DD, ID y DI) \Rightarrow Resulta bastante más complejo
- En el proceso de inserción, los subárboles después de una rotación tienen la misma altura que antes de la inserción y la rotación. Por tanto, después de una rotación no se producirán más rotaciones.
- En eliminación, sin embargo, una vez efectuada una rotación el algoritmo puede continuar. Es decir, se puede producir más de una rotación en el retroceso realizado por el camino de búsqueda, pudiendo llegar hasta la raíz del árbol
- La rama que hay que equilibrar es la contraria a la rama por la que se hace la eliminación

Eliminación por Derecha \Rightarrow Equilibrar por Izquierda

Ejemplo: eliminación del nodo con clave 80



Proceso de Eliminación

- ① Localizar en el árbol la posición del nodo que se desea eliminar teniendo en cuenta los mismos casos que en el procedimiento de eliminación de un a.b.b. (el nodo a suprimir es hoja, tiene un único descendiente o tiene dos descendientes)
- ② **Regresar** por el camino de búsqueda recalculando el factor de equilibrio de los nodos visitados. Reestructurar el árbol en aquellos casos en que sea necesario (antes de que el factor de equilibrio tome valores 2 ó -2)
- Puede implementarse como un algoritmo recursivo que tiene como parámetros:
 - el valor de la clave del nodo que se desea eliminar
 - un puntero que inicialmente señala a la raíz del árbol que permitirá seguir el camino de búsqueda
 - un parámetro de tipo lógico que indicará si la altura del árbol ha cambiado (disminuido) bien por la eliminación de un nodo bien por la reestructuración

Algoritmo eliminar(x:tipoClave; ref cambiaH:lógico; ref nodo:tipoÁrbol)

```

1: si nodo = NULO entonces
2:   /*no existe nodo con clave x: implementar según especificación*/
3: si no
4:   si x < nodo↑.clave entonces
5:     eliminar(x,cambiaH,nodo↑.izq)
6:     si cambiaH entonces
7:       equilibrarDer(nodo,cambiaH)
8:     fin si
9:   si no
10:    si x > nodo↑.clave entonces
11:      eliminar(x,cambiaH,raíz↑.der)
12:      si cambiaH entonces
13:        equilibrarIzq(nodo,cambiaH)
14:      fin si
15:    si no
16:      aux ← nodo                                ->nodo↑.clave = x ¡nodo a eliminar!
17:      si aux↑.der = NULO entonces
18:        nodo← aux↑.izq                         ->sólo hijo izquierdo o ningún hijo
19:        eliminarNodo(aux)
20:        cambiaH← VERDADERO
21:      si no
22:        si aux↑.izq = NULO entonces
23:          nodo← aux↑.der                      ->sólo hijo derecho
24:          eliminarNodo(aux)
25:          cambiaH← VERDADERO
26:        si no
27:          borrar(nodo, aux↑.izq, aux, cambiaH)      ->dos hijos
28:          si cambiaH entonces
29:            equilibrarDer(nodo,cambiaH)
30:          fin si
31:        fin si
32:      fin si
33:    fin si
34:  fin si
35: fin si

```

Observaciones

- Al regresar por el camino de búsqueda se debe equilibrar, si es necesario, la rama opuesta a la eliminación
 - La rama derecha si se ha eliminado a la izquierda
 - La rama izquierda si se ha eliminado a la derecha
- Procedimiento **borrar**
 - elimina el nodo más a la derecha del subárbol izquierdo después de haber sustituido sus valores en el nodo que se pretendía eliminar
 - Regresa por el camino de búsqueda recalculando los factores de equilibrio de los nodos encontrados en el recorrido
 - Puede implementarse como un algoritmo recursivo con cuatro parámetros:
 - tres de tipo puntero nodo para buscar el nodo mas derecha subárbol izquierdo y distinguir si ese nodo es o no la raíz del subárbol izquierdo
 - un parámetro de tipo lógico para controlar el cambio de altura en el subárbol después de la eliminación

Algoritmo borrar

**Algoritmo borrar(ref nodo: punteroNodo; aux, ant: punteroNodo;
ref cambiaH:lógico)**

```
1: si aux↑.der ≠ NULO entonces
2:   borrar(nodo, aux↑.der, aux, cambiaH)
3:   si cambiaH entonces
4:     equilibrarIzq(nodo,cambiaH)
5:   fin si
6: si no
7:   nodo↑.clave ← aux↑.clave
8:   nodo↑.info ← aux↑.info
9:   si ant↑.izq = aux entonces
10:    ant↑.izq ← aux↑.izq
11:   si no
12:    ant↑.der ← aux↑.izq
13:   fin si
14:   liminarNodo(aux)
15:   cambiaH ← VERDADERO
16: fin si
```

5.1 Equilibrar Subárbol Izquierdo

- Este proceso se efectúa al volver por el camino de búsqueda, después de haber **eliminado** un nodo en el **subárbol derecho**
- En esta vuelta se van recalculando los factores de equilibrio de los nodos encontrados en el camino
- Antes de que el factor de equilibrio de un nodo (raíz del subárbol) llegue a tomar el valor -2 se debe reestructurar el **subárbol izquierdo** que será de mayor altura
- Este proceso se implementa mediante un algoritmo con dos parámetros:
 - puntero a la raíz del **subárbol**
 - parámetro de tipo lógico para controlar el cambio de altura del subárbol
- Como siempre, habrá que tener en cuenta tres casos, los tres posibles valores del factor de equilibrio de la raíz del subárbol (-1,0,1)

Casos al equilibrar subárbol IZQUIERDO (vuelta de eliminar por DERECHO)

Antes Eliminación			Después Eliminación			
nodo↑.fe	Subárbol	H _{árbol}	Subárbol	nodo↑.fe	H _{árbol}	CAMBIA_H
0 H _{RI} =H _{RD}	nodo → 0 T ₁ h T ₂ h	h+1	nodo → -1 T ₁ h T ₂ h-1	-1	h+1	FALSO
1 H _{RI} <H _{RD}	nodo → 1 T ₁ h T ₂ h+1	h+2	nodo → 0 T ₁ h T ₂ h	0	h+1	VERDADERO
-1 H _{RI} >H _{RD}	nodo → -1 T ₁ h+1 T ₂ h	h+2	nodo → -2 T ₁ h+1 T ₂ h-1	REESTRUCTURACIÓN SUBÁRBOL IZQUIERDO ¿Rotación? Depende nodo↑.izq↑.fe nodo1 ← nodo↑.izq		

Antes Eliminación		Después Eliminación por rama Izquierda			Después Eliminación por rama Derecha		
nodo↑.fe	altura	nodo↑.fe	altura	cambiaH	nodo↑.fe	altura	cambiaH
0	h+1				-1	h+1	falso
1	h+2				0	h+1	verdadero
-1	h+2				Reestructuración Subárbol Izquierdo		

Casos al equilibrar subárbol IZQUIERDO (continuación I)

Antes Rotación $h_{\text{árbol}} = h+2$	Después Rotación
<p>nodo1↑.fe = -1</p> <p>ROTACIÓN II</p>	<p>nodo1↑.fe ← 0 nodo↑.fe ← 0 $H_{\text{árbol}} = h+1$ \downarrow cambiaH ← VERDADERO</p>
<p>nodo1↑.fe = 0</p> <p>ROTACIÓN II</p>	<p>nodo1↑.fe ← 1 nodo↑.fe ← -1 $h_{\text{árbol}} = h+2$ \downarrow cambiaH ← FALSO</p>
<p>nodo1↑.fe = 1</p> <p>ROTACIÓN ID</p>	

Antes Eliminación		Después Eliminación por rama Derecha					
nodo↑.fe	altura	nodo↑.fe	altura	cambiaH	nodo↑.fe	altura	cambiaH
0	$h+1$	-1				$h+1$	falso
1	$h+2$	0				$h+1$	verdadero
-1	$h+2$	nodo1↑.fe		nodo↑.fe	nodo1↑.fe		
Reest.Sub.Izquierdo nodo1 ← nodo↑.izq ¿II o ID?		-1	RotaciónII(*)	0	0	$h+1$	verdadero
		0	RotaciónII(*)	-1	1	$h+1$	falso
		1	RotaciónID			$h+1$	verdadero

(*)Modificada para eliminación

Algoritmo rotación Izquierda Izquierda modificado para eliminación

Algoritmo rotaciónII(**ref nodo: punteroNodo**, nodo1: punteroNodo)

```

1: nodo↑.izq ← nodo1↑.der
2: nodo1↑.der ← nodo
3: si nodo1↑.fe = -1 entonces
4:     nodo↑.fe ← 0
5:     nodo1↑.fe ← 0
6: si no
7:     nodo↑.fe ← -1
8:     nodo1↑.fe ← 1
9: fin si
10: nodo ← nodo1
    
```

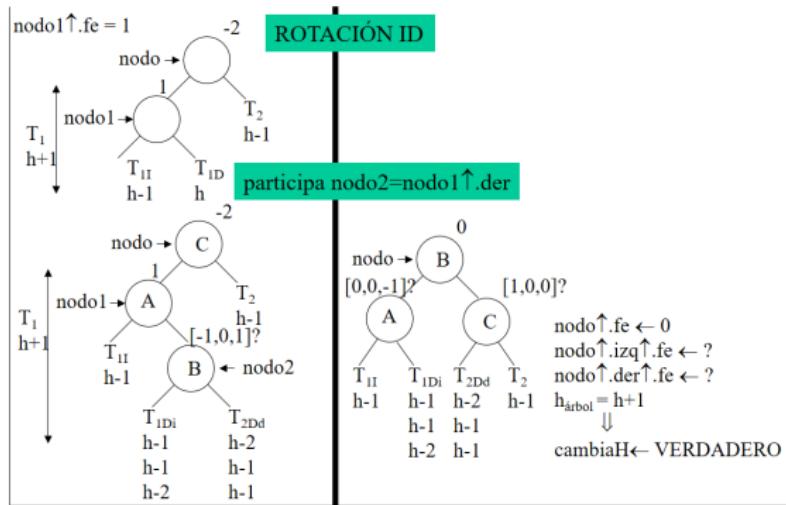
->Movimiento nodos en rotación

->Siempre si la rotación es por una inserción

->Nueva raíz del subárbol

Antes Eliminación		Después Eliminación por rama Derecha					
nodo [↑] .fe	altura	nodo [↑] .fe	altura	cambiaH	nodo [↑] .fe	altura	cambiaH
0	h+1	-1				h+1	falso
1	h+2	0				h+1	verdadero
-1	h+2	nodo1 [↑] .fe	nodo [↑] .fe	nodo1 [↑] .fe			
Reest.Sub.Izquierdo nodo1 ← nodo [↑] .izq ¿II o ID?		-1	RotaciónII(*)	0	0	h+1	verdadero
		0	RotaciónII(*)	-1	1	h+1	falso
		1	RotaciónID			h+1	verdadero

Casos al equilibrar subárbol IZQUIERDO (continuación II)



Equilibra Rama Izquierda (eliminación por derecha)

Antes Eliminación		Después Eliminación por rama Derecha					
nodo↑.fe	altura	nodo↑.fe	altura	cambiaH	nodo↑.fe	altura	cambiaH
0	h+1	-1				h+1	falso
1	h+2	0				h+1	verdadero
-1	h+2	nodo1↑.fe			nodo1↑.fe		
Reest.Sub.Izquierdo nodo1 ← nodo↑.izq ¿II o ID?		-1	RotaciónII(*)	0	0	h+1	verdadero
		0	RotaciónII(*)	-1	1	h+1	falso
		1	RotaciónID			h+1	verdadero

Algoritmo equilibrarIzq(ref nodo: punteroNodo; ref cambiaH:lógico)

```

1: casos nodo↑.fe en
2:   1: nodo↑.fe ← 0
3:   0: nodo↑.fe ← -1
4:   cambiaH ← FALSO
5:   -1: nodo1 ← nodo↑.izq
6:     si nodo1↑.fe ≤ 0 entonces
7:       rotaciónII(nodo,nodo1)
8:     si nodo1↑.fe = 0 entonces
9:       cambiaH ← FALSO
10:    si no
11:      si no
12:        nodo2← nodo1↑.der
13:        rotaciónID(nodo,nodo1,nodo2)
14:      fin si
15: fin casos

```

5.2 Equilibrar Subárbol Derecho

- Este proceso se efectúa al volver por el camino de búsqueda, después de haber **eliminado** un nodo en el **subárbol izquierdo**
- En esta vuelta se van recalculando los factores de equilibrio de los nodos encontrados en el camino
- Antes de que el factor de equilibrio de un nodo (raíz del subárbol) llegue a tomar el valor +2 se debe reestructurar el **subárbol derecho** que será de mayor altura
- Este proceso se implementa mediante un algoritmo con dos parámetros:
 - puntero a la raíz del **subárbol**
 - parámetro de tipo lógico para controlar el cambio de altura del subárbol
- De nuevo, habrá que tener en cuenta tres casos, los tres posibles valores del factor de equilibrio de la raíz del subárbol (-1,0,1)

Casos al equilibrar subárbol DERECHO (vuelta de eliminar por IZQUIERDO)

Antes Eliminación			Después Eliminación			
nodo↑.fe	Subárbol	Hárbol	Subárbol	nodo↑.fe	Hárbol	cambiaH
0 $H_{RI}=H_{RD}$	nodo → 0	$h+1$	nodo → 1	1	$h+1$	FALSO
-1 $H_{RI}>H_{RD}$	nodo → -1	$h+2$	nodo → 0	0	$h+1$	VERDADERO
1 $H_{RI}<H_{RD}$	nodo → 1	$h+2$	nodo → +2	REESTRUCTURACIÓN SUBÁRBOL DERECHO ¿Rotación? Depende nodo↑.der↑.fe $nodo1 \leftarrow nodo↑.der$		

Antes Eliminación		Después Eliminación por rama Izquierda			Después Eliminación por rama Derecha		
nodo↑.fe	altura	nodo↑.fe	altura	cambiaH	nodo↑.fe	altura	cambiaH
0	$h+1$	1	$h+1$	falso	-1	$h+1$	falso
1	$h+2$	Reestructuración Subárbol Derecho			0	$h+1$	verdadero
-1	$h+2$	0	$h+1$	verdadero	Reestructuración Subárbol Izquierdo		

Casos al equilibrar subárbol DERECHO (continuación I)

Antes Rotación $H_{árbol} = h+2$	Después Rotación
<p>nodo\uparrow.fe = 1 nodo → A A → B B → T₁ B → T_{2I} B → T_{2D} T₁ → T_{2I} T₁ → T_{2D}</p> <p>ROTACIÓN DD</p>	<p>nodo\uparrow.fe ← 0 nodo\uparrow.fe ← 0 $H_{árbol} = h+1$ cambiaH ← VERDADERO</p>
<p>nodo\uparrow.fe = 0 nodo → A A → B B → T₁ B → T_{2D} T_{2D} → T₁ T_{2D} → T_{2I}</p> <p>ROTACIÓN DD</p>	<p>nodo\uparrow.fe ← 1 nodo\uparrow.fe ← -1 $H_{árbol} = h+2$ cambiaH ← FALSO</p>
<p>nodo\uparrow.fe = -1</p> <p>ROTACIÓN DI</p>	

Antes Eliminación		Después Eliminación por rama Izquierda					
nodo \uparrow .fe	altura	nodo \uparrow .fe	altura	cambiaH	nodo \uparrow .fe	altura	cambiaH
0	$h+1$	1				$h+1$	falso
-1	$h+2$	0				$h+1$	verdadero
1	$h+2$	Reest.Sub.Derecho nodo \downarrow ← nodo \uparrow .der ¿DD o DI?		nodo \uparrow .fe	nodo \uparrow .fe	nodo \uparrow .fe	
				1	RotaciónDD(*)	0	verdadero
				0	RotaciónDD(*)	-1	falso
				1	RotaciónDI	$h+1$	verdadero

(*)Modificada para eliminación

Algoritmo rotación Derecha Derecha modificado para eliminación

Algoritmo rotaciónDD(**ref nodo: punteroNodo**, nodo1: punteroNodo)

```

1: nodo  $\uparrow$ .der  $\leftarrow$  nodo1  $\uparrow$ .izq
2: nodo1  $\uparrow$ .izq  $\leftarrow$  nodo
3: si nodo1  $\uparrow$ .fe = 1 entonces
4:   nodo  $\uparrow$ .fe  $\leftarrow$  0
5:   nodo1  $\uparrow$ .fe  $\leftarrow$  0
6: si no
7:   nodo  $\uparrow$ .fe  $\leftarrow$  1
8:   nodo1  $\uparrow$ .fe  $\leftarrow$  -11
9: fin si
10: nodo  $\leftarrow$  nodo1

```

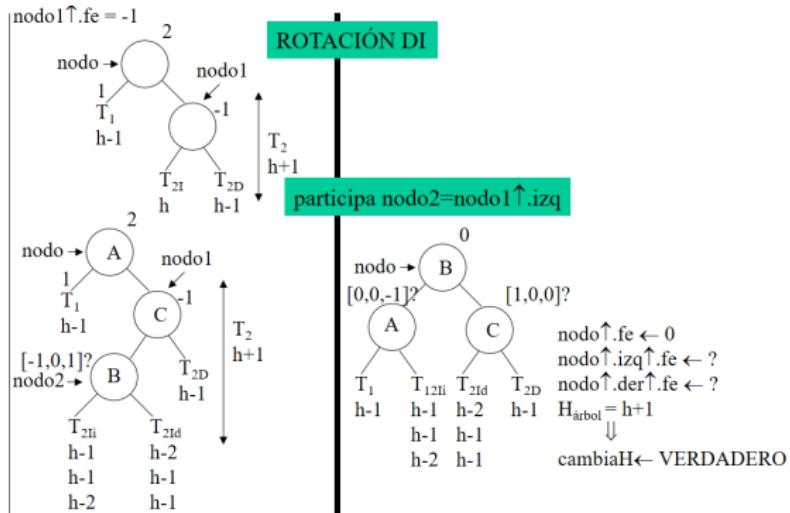
->Movimiento nodos en rotación

->Siempre si la rotación es por una inserción

->Nueva raíz del subárbol

Antes Eliminación		Después Eliminación por rama Izquierda					
nodo \uparrow . fe	altura	nodo \uparrow . fe	altura	cambiaH	nodo \uparrow . fe	altura	cambiaH
0	$h+1$	1				$h+1$	falso
-1	$h+2$	0				$h+1$	verdadero
1 Reest.Sub.Derecho <i>nodo1</i> \leftarrow <i>nodo</i> \uparrow . <i>der</i> ¿DD o DI?	$h+2$	<i>nodo1</i> \uparrow . fe	nodo \uparrow . fe	nodo1 \uparrow . fe			
		1	RotaciónDD(*)	0	0	$h+1$	verdadero
		0	RotaciónDD(*)	-1	1	$h+1$	falso
		1	RotaciónDI			$h+1$	verdadero

Casos al equilibrar subárbol DERECHO (continuación II)



Equilibra Rama Derecha (elimanción por izquierda)

Antes Eliminación		Después Eliminación por rama Izquierda					
nodo↑.fe	altura	nodo↑.fe	altura	cambiaH	nodo↑.fe	altura	cambiaH
0	h+1	1				h+1	falso
-1	h+2	0				h+1	verdadero
1	h+2	nodo1↑.fe			nodo1↑.fe		
Reest.Sub.Derecho nodo1 ← nodo↑.der ¿DD o DI?		1	RotaciónDD(*)	0	0	h+1	verdadero
		0	RotaciónDD(*)	-1	1	h+1	falso
		1	RotaciónDI			h+1	verdadero

Algoritmo equilibrarDer(ref nodo: punteroNodo; ref cambiaH:lógico)

```

1: casos nodo↑.fe en
2:   -1: nodo↑.fe ← 0
3:   0: nodo↑.fe ← 1
4:   cambiaH ← FALSO
5:   1: nodo1 ← nodo↑.der
6:     si nodo1↑.fe ≥ 0 entonces
7:       rotaciónDD(nodo,nodo1)
8:     si nodo1↑.fe = 0 entonces
9:       cambiaH ← FALSO
10:    si no
11:      si no
12:        nodo2← nodo1↑.izq
13:        rotaciónDI(nodo,nodo1,nodo2)
14:      fin si
15: fin casos

```

Contenido

5 Tema 5. Árboles Binarios de Búsqueda

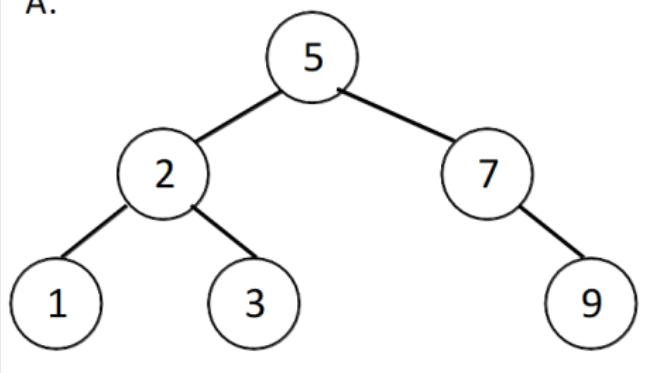
- Nivel abstracto o de definición
- Nivel de representación e implementación
- Árboles Balanceados
- Inserción y Equilibrio del árbol
- Eliminación y equilibrio del árbol
- Ejercicios

Ejercicio 1

Crear el árbol binario de búsqueda que resulta al insertar las siguientes claves:

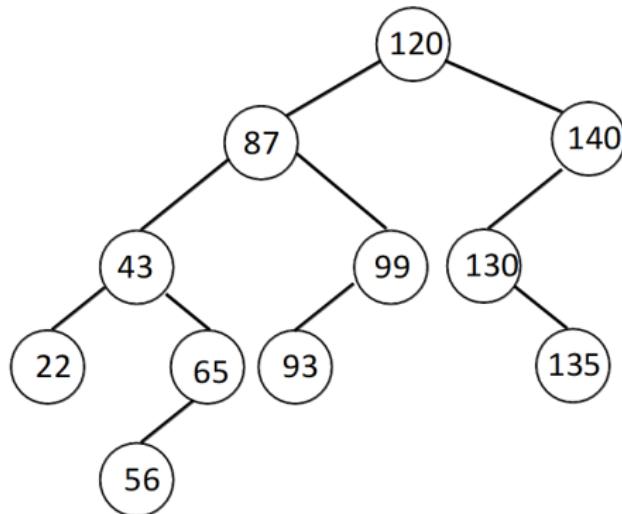
- A. 5,7,2,3,9,1
- B. 1,2,3,5,7,9
- C. 9,7,5,3,2,1
- D. En cualquier otro orden

A.



Ejercicio 2

Eliminar del a.b.búsqueda de la figura los nodos con las claves: 22, 99, 87, 120, 140, 135 y 56 (en ese orden)

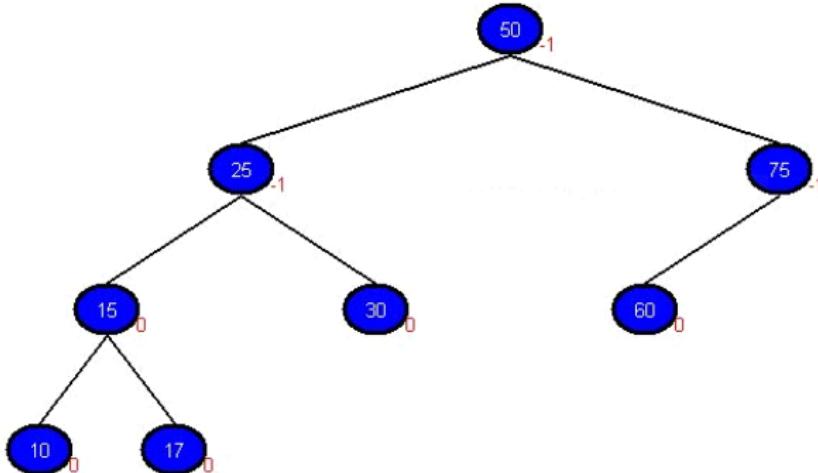


Ejercicio 3

Dado el árbol balanceado de la figura:

- a) Insertar un nodo con valor 12
- b) Insertar un nodo con valor 20

Indicar que tipo de rotación se produce y mostrar gráficamente el árbol resultante. ¿Qué nodos participan en la rotación (nodo, nodo1 y nodo2)?

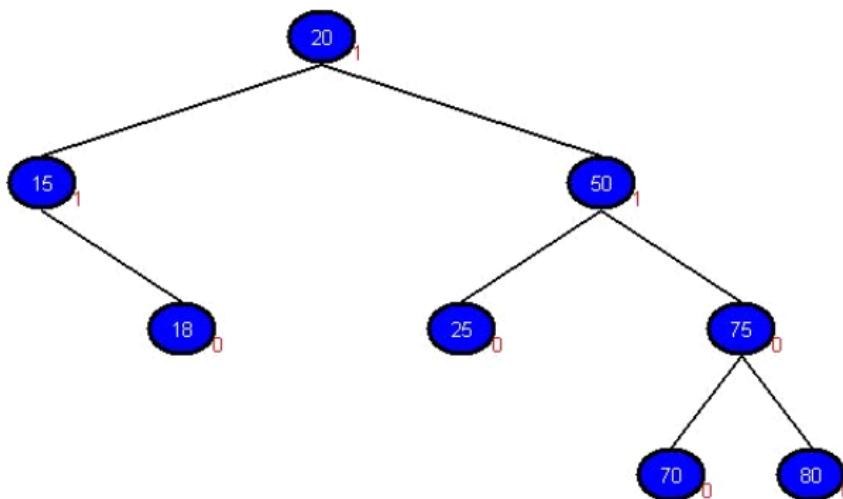


Ejercicio 4

Dado el árbol balanceado de la figura:

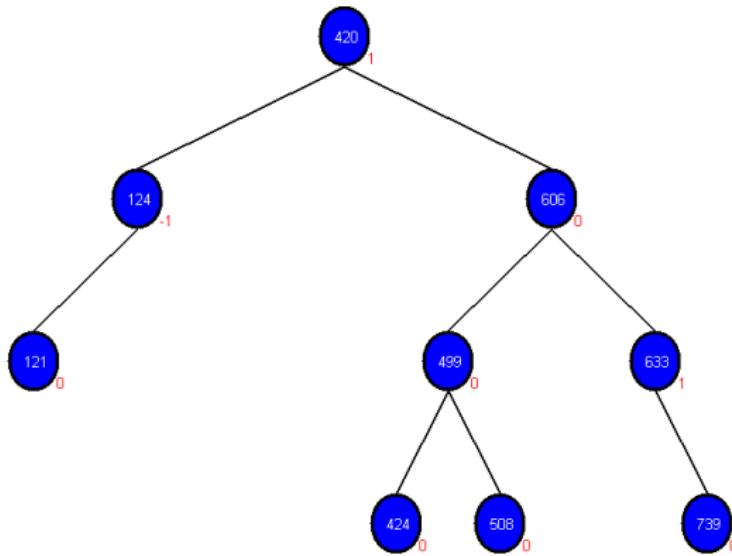
- a) Insertar un nodo con valor 90
- b) Insertar un nodo con valor 60

Indicar que tipo de rotación se produce y mostrar gráficamente el árbol resultante. ¿Qué nodos participan en la rotación (nodo, nodo1 y nodo2)?



Ejercicio 5

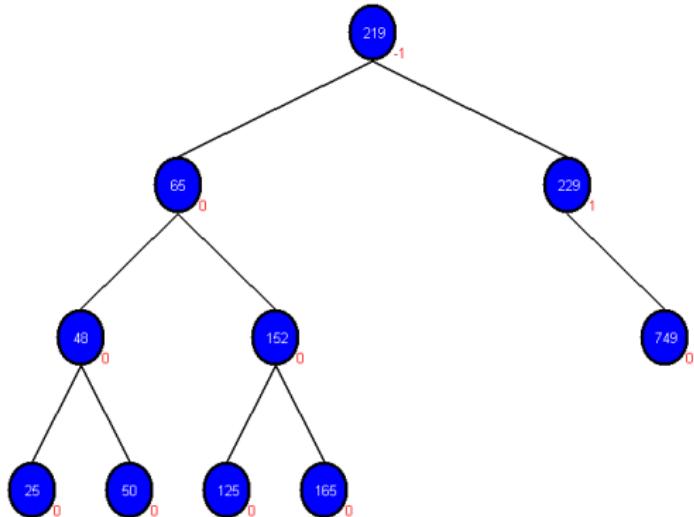
Dado el árbol balanceado de la figura y, teniendo en cuenta la tabla que muestra todos los casos de rotaciones que se pueden presentar ante una eliminación en un árbol balanceado, indicar qué tipo de rotación se produce y mostrar cómo queda el árbol después de eliminar el nodo con valor 121.



nodo↑.fe = 2	nodo↑.fe = 2		
nodo↑.der.fe	rotación	nodo↑.izq.fe	rotación
1	DD	-1	II
0	DD	0	II
-1	DI	1	ID

Ejercicio 6

Dado el árbol balanceado de la figura y, teniendo en cuenta la tabla que muestra todos los casos de rotaciones que se pueden presentar ante una eliminación en un árbol balanceado, indicar qué tipo de rotación se produce y mostrar cómo queda el árbol después de eliminar el nodo con valor 749



nodo↑.fe = 2		nodo↑.fe = 2	
nodo↑.der.fe	rotación	nodo↑.izq.fe	rotación
1	DD	-1	II
0	DD	0	II
-1	DI	1	ID

Contenido

- 1 Tema1. Árboles Generales y Binarios
- 2 Tema 2. Montículos Binarios
- 3 Tema 3. Conjuntos Disjuntos
- 4 Tema 4. Grafos
- 5 Tema 5. Árboles Binarios de Búsqueda
- 6 Tema 6. Organización de archivos
- 7 Tema 7. Organización de Índices

Contenido

6 Tema 6. Organización de archivos

- Introducción
- Organización Secuencial
- Organizaciones Indexadas
- Organización Directa. Dispersión
- Clasificación externa

1 Introducción

- Ventajas del almacenamiento de información en memoria principal:
 - rapidez de acceso
 - igual cantidad de tiempo para acceder a datos en diferentes posiciones
- No siempre es posible almacenar datos en memoria principal:
 - situaciones en que la cantidad de datos a procesar supera la capacidad de la memoria principal
 - situaciones en que se necesita guardar la información en almacenamiento estable
- Necesidad de utilizar dispositivos de almacenamiento externo cuyas características de acceso difieren bastante de las de la memoria principal
- Las estructuras de datos aplicadas a colecciones de datos en almacenamiento secundarios se denominan ARCHIVOS o FICHEROS

Conceptos básicos relacionados con la organización de archivos

Archivo, registro, campo

- Los lenguajes de programación suelen tener el tipo de datos archivo, destinado a representar datos en memoria secundaria
 - **Archivo.** Conjunto de datos estructurados en una colección de entidades elementales o básicas denominadas registros.
 - **Registro.** Colección de diferentes entidades de nivel inferior denominadas campos
 - **Campo.** Elemento de datos caracterizado por su nombre, tamaño y tipo de datos. Unidad mínima de información de un registro
- La organización de archivos define la forma en que los registros se disponen en el medio de almacenamiento
- Tres organizaciones fundamentales
 - Organización secuencial
 - Organizaciones indexadas
 - Organización directa o aleatoria

Conceptos básicos relacionados con el acceso a archivos

- El sistema operativo divide la memoria secundaria en bloques de igual tamaño y considera un archivo como una colección de bloques
- **Operación básica en archivos:** copiar un solo bloque del archivo a un área temporal de la memoria principal (buffer), cuyo tamaño es idéntico al de un bloque, de acuerdo con el orden en que el sistema operativo maneja los bloques
- El tiempo para encontrar un bloque y copiarlo de memoria secundaria a memoria principal es **muy grande** comparado con el tiempo de proceso de los datos
- Al evaluar el **tiempo de ejecución** de los algoritmos que operan con información almacenada en memoria secundaria es importante considerar el número de veces que se lee/escribe un bloque: número de accesos a bloque o **número de operaciones de entrada /salida**

Contenido

6 Tema 6. Organización de archivos

- Introducción
- **Organización Secuencial**
- Organizaciones Indexadas
- Organización Directa. Dispersión
- Clasificación externa

2 Organización Secuencial

- Forma básica de almacenar un conjunto de registros que constituyen un archivo
- Los registros quedan grabados consecutivamente cuando el archivo se crea y debe accederse a ellos de forma consecutiva cuando el archivo se procesa

Búsqueda SECUENCIAL

- Método de búsqueda o direccionamiento de un registro en un archivo secuencial que consiste en leer y comprobar uno tras otro los registros del archivo hasta encontrar el registro deseado
- Método muy lento, normalmente solo se emplea en medios de almacenamiento secuencial, pero sirve como punto de partida y comparación para medir las mejoras de otros métodos
- Búsqueda secuencial o búsqueda por bloque es $O(n)$
 - El número de operaciones de E/S es **proporcional** al número de registros y se incrementa en proporción directa al incremento del tamaño del archivo
- **Objetivo.** Encontrar métodos mejores para acceder a registros individuales

Archivos secuenciales ORDENADOS

- En muchos casos, los registros de un archivo secuencial se clasifican según el valor de algún campo o conjunto de campos de cada registro
- El campo elegido para ordenar, clasificar e identificar a cada registro se le denomina **clave** (no tiene ninguna relación con la posición física de los registros)
- **Dirección relativa o número relativo de registro (NRR)**: posición relativa de un registro respecto al primer registro del archivo

Búsqueda BINARIA

- Método de direccionamiento que requiere que el archivo esté clasificado según el valor de la clave que se emplea en la búsqueda
- Proceso para encontrar un registro con valor K_b para la clave:
 - ① Comparar K_b con la clave del registro que ocupa la posición central del fichero
 - ② Según el resultado de la comparación se repite el mismo proceso en el área del fichero que contenga el registro hasta encontrar el registro buscado, si existe

Número máximo de accesos búsqueda binaria

Ejemplo: búsqueda binaria en archivo secuencial ordenado. $K_b = \text{"Felix"}$

- En promedio el número de operaciones de E/S para un archivo de n registros es proporcional a $\lg n$:

- Mejora considerable frente a la búsqueda secuencial
- Incrementar el tamaño del archivo no supone aumentar directamente el número de accesos: duplicar el tamaño del archivo sólo implica una comparación más como máximo

- Búsqueda binaria más efectiva pero **solo funciona si los registros están ordenados** en términos de la clave que se está buscando

NRR	Clave	Otros campos
1	ADOLFO	
2	ALBERTO	
3	ALONSO	
4	ANA	
5	ANTONIO	
6	BEATRIZ	
7	BERTA	
8	BRAULIO	
9	CARLOS	
10	CARMEN	
11	CAROLINA	
12	CLARA	2º inspección: $K_b > K_{12} \Downarrow$
13	DANIEL	
14	DIANA	
15	DORA	
16	ELISA	
17	ELSA	3º inspección: $K_b > K_{17} \Downarrow$
18	EMILIO	
19	ESTEBAN	
20	FÉLIX	4º inspección: $K_b = K_{12}$ Éxito
21	FRANCISCO	
22	GONZALO	
23	IGNACIO	1º inspección: $K_b < K_{23} \Updownarrow$
24	JOAQUÍN	
25	LEONARDO	
26	MANUEL	
27	MARÍA	
28	MARIANO	
29	MIGUEL	
30	NATALIA	
31	OLGA	
32	PASCUAL	
33	PEDRO	
34	PILAR	
35	RAMIRO	
36	RAMÓN	
37	RAQUEL	
38	RAÚL	
39	RICARDO	
40	RUBÉN	
41	SANTIAGO	
42	SARA	
43	SUSANA	
44	TERESA	
45	VERÓNICA	
46	VICENTE	

Contenido

6 Tema 6. Organización de archivos

- Introducción
- Organización Secuencial
- **Organizaciones Indexadas**
- Organización Directa. Dispersión
- Clasificación externa

3 Organizaciones Indexadas

- Las organizaciones indexadas constan de dos tipo de archivos
 - Archivo de datos para almacenar los registros
 - Archivo índice para localizar los registros. Contiene la posición relativa de cada registro o grupo de registros en el archivo de datos
- **Método de direccionamiento directo:** acceso a un registro determinado sin necesidad de consultar los registros precedentes
- Archivo índice
 - tabla cuya entrada es la clave del registro buscado y da como salida la dirección relativa del registro en el fichero de datos o del área del fichero que contiene al registro buscado
- Organizaciones indexadas:
 - Organización **Secuencial Indexada:** fichero de datos **ordenado** según el valor de la clave de búsqueda
 - Organización **No Secuencial Indexada:** fichero de datos **no ordenado** según la clave de búsqueda

Organización Secuencial Indexada

- El archivo está clasificado secuencialmente según el valor de la clave de entrada utilizada en el índice
- No es necesaria una entrada por cada registro: el índice incluye referencias a bloques de registros, los cuales se explorarán después para finalizar la búsqueda

NRR	Registros
1	ADOLFO
2	ALBERTO
3	ALONSO
4	ANA
5	ANTONIO
6	BEATRIZ
7	BERTA
8	BRAULIO
9	CARLOS
10	CARMEN
11	CAROLINA
12	CLARA
13	DANIEL
14	DIANA
15	DORA
16	ELISA
17	ELSA
18	EMILIO
19	ESTEBAN
20	FELIX
21	FRANCISCO
22	GONZALO
23	IGNACIO
24	JOAQUÍN
25	LEONARDO
26	MANUEL
27	MARÍA
28	MARIANO
29	MIGUEL
30	NATALIA
31	OLGA
32	PASCUAL
33	PEDRO
34	PILAR
35	RAMIRO
36	RAMÓN
37	RAQUEL
38	RAÚL
39	RICARDO
40	RUBÉN
41	SANTIAGO

Organización NO secuencial indexada

- El archivo no está clasificado según el valor de la clave de entrada utilizada en el índice
- Se necesita una entrada por cada registro, ocupan más espacio y aumenta el tiempo para explorarlos
- Normalmente dan una **dirección simbólica** que se corresponde con el valor de la clave primaria

NRR	DATOS
1	ELISA MARTIN SANCHEZ
2	EMILIO
3	RICARDO
4	CLARA
5	RAQUEL
6	DANIEL
7	SUSANA
8	ADOLFO
9	ANTONIO
10	DIANA
11	PEDRO
12	RUBEN
13	CARLOS
14	Elsa
15	IGNACIO
16	SANTIAGO
17	FRANCISCO
18	CARMEN
19	JOAQUIN
20	GONZALO
21	RAMON
22	BRAULIO
23	MANUEL
24	OLGA
25	CAROLINA
26	MIGUEL
27	BERTA
28	PASCUAL
29	TERESA
30	ALONSO
31	LEONARDO
32	MARIA
33	MARIANO
34	RAMIRO
35	FELIX
36	NATALIA
37	DORA
38	PILAR
39	BEATRIZ
40	ALBERTO
41	ANA
42	VERONICA
43	SARA
44	ESTEBAN
45	VICENTE
46	RAUL

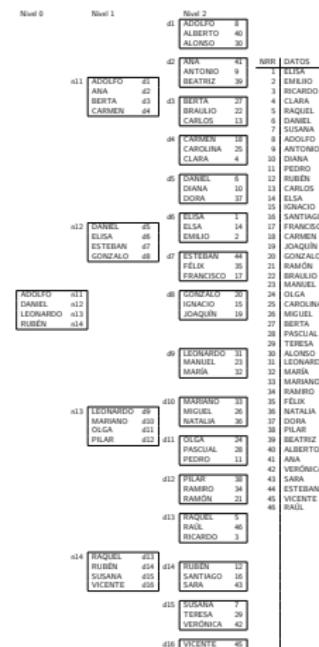
Índices primarios y secundarios

- En la mayoría de los ficheros se buscan registros según el valor de más de una clave. El fichero solo puede estar ordenado respecto a una de las claves
 - Se ordena según la clave de búsqueda más frecuente y se le asocia una organización secuencial indexada (índice primario)
 - Para el resto de las claves (no secuenciales) se asocia una organización no secuencial indexada (índices secundarios)
- Para evitar índices muy grandes, a veces, se recurre a índices de varios niveles:
 - El nivel más alto da la posición del siguiente nivel
 - El nivel más bajo da la posición del bloque de registros que contiene al registro buscado
 - El bloque se explora secuencialmente o mediante búsqueda binaria para encontrar el registro

Ejemplos con índices de varios niveles

O. Secuencial indexada (índice de dos niveles)

	Nivel 0	Nivel 1	NRR	Registros
D1	ADOLFO 1 ANTONIO 5 CARLOS 9 DANIEL 13		1	ADOLFO
			2	ALBERTO
			3	ALONSO
			4	ANA
			5	ANTONIO
			6	BEATRIZ
			7	BERTA
			8	BRAULIO
			9	CARLOS
			10	CARMEN
			11	CAROLINA
			12	CLARA
			13	DANIEL
			14	DIANA
			15	DORA
			16	ELISA
D2	ELSA 17 FRANCISCO 21 LEONARDO 25 MIGUEL 29		17	ELSA
			18	EMILIO
			19	ESTEBAN
			20	FELIX
			21	FRANCISCO
			22	GONZALO
			23	IGNACIO
			24	JOAQUÍN
			25	LEONARDO
			26	MANUEL
			27	MARÍA
			28	MARIANO
			29	MIGUEL
			30	NATALIA
			31	OLGA
			32	PASCUAL
			33	PEDRO
			34	PILAR
			35	RAMIRO
			36	RAMÓN
D4			37	RAQUEL
			38	RÁUL
			39	RICARDO
			40	RUBÉN
			41	SANTIAGO
			42	SARA
			43	SUSANA
			44	TERESA
			45	VERONICA
			46	VICENTE



Organización no secuencial con índice de tres niveles

Nivel 0	Nivel 1	Nivel 2	
		d1 ADOLFO 8 ALBERTO 40 ALONSO 30	
n11	ADOLFO d1 ANA d2 BERTA d3 CARMEN d4	d2 ANA 41 ANTONIO 9 BEATRIZ 39	NRR DATOS
		d3 BERTA 27 BRAULIO 22 CARLOS 13	1 ELISA 2 EMILIO 3 RICARDO 4 CLARA 5 RAQUEL 6 DANIEL 7 SUSANA 8 ADOLFO 9 ANTONIO 10 DIANA 11 PEDRO 12 RUBÉN 13 CARLOS 14 ELSA 15 IGNACIO 16 SANTIAGO 17 FRANCISCO 18 CARMEN 19 JOAQUÍN 20 GONZALO 21 RAMÓN 22 BRAULIO 23 MANUEL 24 OLGA 25 CAROLINA 26 MIGUEL 27 BERTA 28 PASCUAL 29 TERESA 30 ALONSO
n12	DANIEL d5 ELISA d6 ESTEBAN d7 GONZALO d8	d5 DANIEL 6 DIANA 10 DORA 37	
		d6 ELISA 1 ELSA 14 EMILIO 2	
		d7 ESTEBAN 44 FÉLIX 35 FRANCISCO 17	
		d8 GONZALO 20 IGNACIO 15 JOAQUÍN 19	
		d9 LEONARDO 31	

ADOLFO	n11
DANIEL	n12
LEONARDO	n13
RUBÉN	n14

Inserción de registros en Organización Secuencial Indexada

- La organización secuencial indexada ofrece como ventajas:
 - admite procesamiento secuencial y directo
 - el índice es de menor extensión
- Desventaja
 - la distribución secuencial del fichero de datos es más difícil de mantener al insertar nuevos registros
- Técnicas para tratar las inserciones:
 - **Área de desborde.** Se almacenan los nuevos registros en otras posiciones dejando punteros en el lugar del fichero donde debería almacenarse el registro ⇒ un acceso adicional cada vez que se busque uno de los nuevos registros
 - **Espacio libre distribuido.** Se dejan espacios libres en el fichero al cargarlo inicialmente
- En ambos casos serán necesarias operaciones básicas de mantenimiento

Ejemplo de inserción de registros con claves: SAMUEL, JUAN, JOSÉ y MARTA

Área de Desborde

Nivel 0	Nivel 1	NRR	Registros
D1	ADOLFO 1 ANTONIO 5 CARLOS 9 DANIEL 13	1	ADOLFO 2 ALBERTO 3 ALONSO 4 ANA 9 ANTONIO 6 BEATRIZ 7 BERTA 8 BRAULIO 9 CARLOS 10 CARMEN 11 CAROLINA 12 CLARA 13 DANIEL 14 FRANCA 15 DORA 16 ELISA
D2	ELSA 17 FRANCISCO 21 LEONARDO 25 MIGUEL 29	17	17 ELSA 18 EMILIO 19 ESTEBAN 20 FELIX
D3	ADOLFO D1 ELSA D2 PEDRO D3	33	21 FRANCISCO 22 GONZALO 23 IGNACIO 24 JOAQUÍN ... 102
D3	PEDRO 33 RAQUEL 37 SANTIAGO 41 VERÓNICA 45	37	25 LEONARDO 26 MANUEL 27 MARIA 28 MARIANO ... 103
D4	41	29 MIGUEL 30 NATALIA 31 OLGA 32 PASCUAL 33 PEDRO 34 PILAR 35 RAMIRO 36 RAMON 37 RAQUEL 38 RAÚL 39 RICARDO 40 RUBÉN ... 100
		42	41 SANTIAGO 42 SARA 43 SUSANA 44 TERESA 45 VERONICA 46 VICENTE
		100	100 SAMUEL 101 JUAN 102 JOSÉ ... 101 103 MARTA
		101	
		102	
		103	

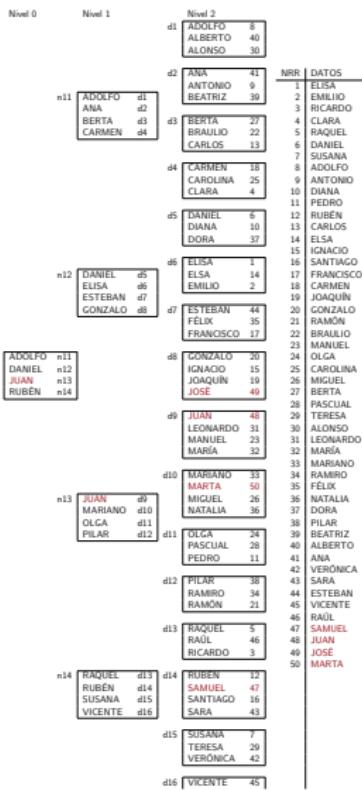
Espacio Libre Distribuido

Nivel 0	Nivel 1	NRR	Registros
D1	ADOLFO 1 ANNA 5 BERTA 9 CARMEN 13	1	1 ADOLFO 2 ALBERTO 3 ALONSO 4 ANA 5 ANTONIO 6 BEATRIZ 8 BERTA 9 BERTA 10 BRAULIO 11 CARLOS 12 CLARA 13 CARMEN 14 CAROLINA 15 CLARA 16 DIANA 17 DANIEL 18 DIANA 19 DORA 20 DORA
D2	DANIEL 17 ELISA 21 ESTEBAN 25 GONZALO 29	17	21 ELISA 22 ELSA 23 EMILIO 24 FELIX 25 ESTEBAN 26 FRANCISCO 27 FRANCISCO 28 GONZALO 29 GONZALO 30 IGNACIO 31 JOAQUIN 32 JUAN
D3	ADOLFO D1 DANIEL D2 LEONARDO D3 RAQUEL D4	33	33 JOSÉ!!! 34 LEONARDO 35 MANUEL 36 MARIA 37 MARIANO 38 MIGUEL 39 NATALIA 40 OLGA 41 PASCUAL 42 PEDRO 43 RICARDO 44 RUBÉN 45 PILAR 46 RAMIRO 47 RAMÓN 48 RAQUEL 49 RAÚL 50 RICARDO 51 SANTIAGO 52 SARA 53 SAMUEL 54 SANTIAGO 55 SARA 56 SUSANA 57 TERESA 58 VERÓNICA 60 VICENTE 61 VICENTE
D4	RAQUEL 49 RUBEN 53 SUSANA 57 VICENTE 61	49	
		50	
		51	
		52	
		53	
		54	
		55	
		56	
		57	
		58	
		59	
		60	
		61	

Inserción de registros en Organización No Secuencial Indexada

- La inserción de nuevos registros no plantea problemas en el archivo de datos: simplemente se añaden al final del fichero
- La organización del índice, sin embargo, se complica en las organizaciones indexadas. Realmente lo que se organiza es el archivo índice (organizaciones B y B+ que estudiaremos en el tema 7)

Ejemplo →



Contenido

6 Tema 6. Organización de archivos

- Introducción
- Organización Secuencial
- Organizaciones Indexadas
- **Organización Directa. Dispersión**
- Clasificación externa

4 Organización Directa. Dispersión

- Método de organización que opera basándose en una función de cálculo de dirección o función HASH que convierte la clave de un registro en una dirección dentro del fichero: $h : K \rightarrow D$
- Ventajas
 - Rapidez de acceso: permite el acceso directo a cualquier registro sin consultar índices ni leer registros anteriores
 - Aumenta la velocidad de búsqueda sin necesidad de tener los registros ordenados
 - El tiempo de búsqueda es independiente del número de registros del fichero
- Características de la función hash:
 - Simple de calcular
 - Transformación de claves en direcciones apropiadas equilibrando el espacio de almacenamiento y el tiempo de búsqueda
 - Distribución uniforme generando posiciones diferentes para claves diferentes

Método de DISPERSIÓN

- Consiste en dividir el área de almacenamiento en grupos denominados CUBOS que permiten almacenar un determinado número de registros cada uno
- Si el archivo se divide en N cubos, éstos se numeran de 0 a $N-1$ y se utiliza una función de dispersión de forma que a cada valor de la clave le hace corresponder algún entero en el rango $0..N-1$
- Algunos cubos pueden estar llenos mientras que otros pueden estar vacíos o poco ocupados \Rightarrow colisiones o desbordes

Búsqueda directa

Si k es el valor de la clave, $h(k)$ proporciona el número de cubo que contiene al registro con clave k , si ese registro existe

Colisión o desborde

Si la función de cálculo de dirección asigna un registro a un cubo que está lleno

Método de Dispersion. Ejemplo

El método de dispersión requiere la elección previa de:

- **Función de cálculo de dirección o función hash:** simple de calcular y que asigne direcciones de la forma más uniforme posible (módulo, cuadrado dígitos centrales, elegir dígitos, ...)

- **Método para resolver los desbordes** que genere posiciones alternativas para almacenar los registros que sean asignados a cubos llenos

CLAVE	conversión numérica de la clave	clave%29
ELISA	53921	10
EMILIO	5493997	5
RICARDO	1931147	8
CLARA	33111	22
RAQUEL	119453	2
DANIEL	415063	6
SUSANA	14251	6
ADOLFO	147367	18
ANTONIO	1537597	17
DIANA	49151	25
PEDRO	85417	12
RUBÉN	14295	16
CARLOS	311372	28
ELSA	5321	14
IGNACIO	9751397	2
SANTIAGO	21539177	7
FRANCISCO	611539237	26
CARMEN	311455	24
JOAQUÍN	1717945	27
GONZALO	7759137	13
RAMÓN	11475	20
BAUWIL	2114397	7
MANUEL	415453	28
OLGA	7371	5
CAROLINA	31173951	24
MIGUEL	497453	16
BERTA	25131	17
PASCUAL	8123413	20
TERESA	351521	12
ALONSO	137527	9
LEONARDO	35751147	5
MARÍA	41191	11
MARIANO	4119157	26
RAMIRO	114917	19
FÉLIX	65397	2
NATALIA	5131391	15
DORA	4711	13
PILAR	89311	20
BEATRIZ	2513199	1
ALBERTO	1325137	11
ANA	151	6
VERÓNICA	55175931	9
SARA	2111	23
ESTEBAN	5235215	19
VICENTE	5935535	18
RAÚL	1143	12

Espacio inicial de Almacenamiento
29 cubos de capacidad 2

CUBO	jdesbordes!	
0	SUSANA	BEATRIZ
1	RAQUEL	IGNACIO
2		
3		
4		
5	EMILIO	OLGA
6	DANIEL	ANA
7	SANTIAGO	BAUWIL
8	RICARDO	
9	ALONSO	VERÓNICA
10	ELISA	
11	MARÍA	ALBERTO
12	PEDRO	TERESA
13	GONZALO	DORA
14	ELSA	
15	NATALIA	
16	RUBÉN	MIGUEL
17	ANTONIO	BERTA
18	ADOLFO	VICENTE
19	RAMIRO	ESTEBAN
20	RAMÓN	PASCUAL
21		
22	CLARA	
23	SARA	
24	CARMEN	CAROLINA
25	DIANA	
26	FRANCISCO	MARIANO
27	JOAQUÍN	
28	CARLOS	MANUEL
29		

Desbordes y Nuevas Inserciones

- Algunos métodos utilizan un área especial, **área de Desborde**, para almacenar los registros desbordados, registrando la dirección del cubo de desborde en el cubo inicialmente asignado: **encadenamiento de desbordes**
- ¡¡No se necesita ninguna técnica adicional para tratar las nuevas inserciones!!!

CLAVE	conversión numérica de la clave	clave%29	Espacio inicial de Almacenamiento 29 cubos de capacidad 2
ELISA	53921	10	
EMILIO	5493997	5	
RICARDO	1931147	8	
CLARA	33111	22	
RAQUEL	119453	2	
DANIEL	415953	6	
cline5-SUSANA	242151	1	CUBO
ADOLFO	147367	18	0
ANTONIO	1537597	17	1 SUSANA BEATRIZ
DIANA	40151	25	2 RAQUEL IGNACIO
PEDRO	85417	12	3
RUBÉN	14255	16	4
CARLOS	311372	28	5 EMILIO OLGA
ELSA	5321	14	6 DANIEL ANA
IGNACIO	9751397	2	7 SANTIAGO BRAULIO
SANTIAGO	21539177	7	8 RICARDO
FRANCISCO	611539237	26	9 ALONSO VERÓNICA
CARMEN	311455	24	10 ELISA
JOAQUÍN	171949	27	11 MARIA ALBERTO
GONZALO	7759137	13	12 PEDRO TERESA
RAMÓN	11475	20	13 GONZALO DORA
BRAULIO	2114397	7	14 ELSA JOSE
MANUEL	415453	28	15 NATALIA
OLGA	7371	5	16 RUBEN MIGUEL
CAROLINA	31173951	24	17 ANTONIO BERTA
MIGUEL	407453	16	18 ADOLFO VICENTE
BERTA	25131	17	19 RAMIRO ESTEBAN
PASCUAL	8123413	10	20 RAMON PASCUAL
TERESA	351521	12	21
ALONSO	137527	9	22 CLARA
LEONARDO	35751147	5	23 SARA JUAN
MARÍA	41191	11	24 CARMEN CAROLINA
MARIANO	4119157	16	25 DIANA
RAMIRO	114917	19	26 FRANCISCO MARIANO
FÉLIX	65397	2	27 JOAQUIN SAMUEL
NATALIA	5131391	15	28 CARLOS MANUEL
DORA	4711	13	29 LEONARDO FELIX
PILAR	89311	20	30 PILAR RAUL
BEATRIZ	2513199	1	31 MARTA
ALBERTO	1325137	11	32
ANA	151	6	
VERÓNICA	55175931	9	
SARA	2111	23	
ESTEBAN	5235215	19	
VICENTE	5935535	18	
RAÚL	1143	12	
SAMUEL	214453	27	
JUAN	1415	23	
JOSE	1725	14	
MARTA	41131	9	

Cubos de desborde

Dispersión Dinámica

- Modificación del método de organización directa que permite variar el número de cubos asignados en función de su densidad de ocupación
- El número de cubos asignados inicialmente aumenta o disminuye a medida que estos se van llenando por nuevas inserciones o se van desocupando por eliminaciones
- Gran flexibilidad pues se incrementa o disminuye el espacio de almacenamiento según se necesita
- Las operaciones de expansión y reducción originan reasignación de registros, esto supone un alto coste por las lecturas y escrituras que deben realizarse
- La asignación dinámica de cubos se realiza mediante expansiones

Expansiones

- Expansión total: cuando se supera la densidad de ocupación el número de cubos se duplica

Inicialmente $\Rightarrow N$ cubos

Primera expansión $\Rightarrow 2N$ cubos

Segunda expansión $\Rightarrow 4N$ cubos

...

- Expansión parcial: se incrementa el número de cubos en un 50%, de forma que dos expansiones parciales equivalen a un expansión total

Inicialmente $\Rightarrow N$ cubos

Primera expansión $\Rightarrow N + N/2 = 3N/2$ cubos

Segunda expansión $\Rightarrow 3N/2 + N/2 = 2N$ cubos

Tercera expansión $\Rightarrow 2N + N = 3N$ cubos

Cuarta expansión $\Rightarrow 3N + N = 4N$ cubos

...

Ejemplo

N=2 C=2

Densidad máxima de ocupación 80%

K	KmodN (densidad)	KmodN (densidad)	KmodN (densidad)
42	0 (25%)	2	2
24	0 (50%)	0	0
15	1 (75%)	3	7
53	1 (100%)	1	5
21		1 (63%)	5
12		0 (75%)	4
14		2 (88%)	6
18			2 (50%)
49			1 (56%)
128			0 (63%)
22			6 (69%)
23			7 (75%)
67			3 (81%)

Densidad mínima de ocupación 40%

Eliminar	K	densidad	N=8
	53	75%	0 24 128
	24	69%	1 49
	12	63%	2 42 18
	18	56%	3 67
	22	50%	4 12
	128	44%	5 53 21
	23	37,5%	6 14 22
			7 15 23

	N=2	N=4	N=8
0	42 24	24 12	24 128
1	15 53	53 21	49
2		42 14	42 18
3		15	67

1^a expansión

2^a expansión ⇒

K	K mod N
49	1
42	2
67	3
21	1
14	2
15	3

N=4

Reducción->	0	1	2	3
		49 21		
		42 14		
			67 15	

Observaciones

- La idea básica de la dispersión consiste en trasformar las claves en direcciones de memoria mediante una función de cálculo de dirección
- Generalmente se aprovecha su potencial para la implementación en memoria secundaria pero ...
 - Estas direcciones pueden hacer referencia a memoria primaria, en cuyo caso tendríamos un vector o array
 - Todos los conceptos pueden utilizarse para construir el TAD conocido como tablas de dispersión (hashing) o tablas asociativas
 - Prácticamente todos los compiladores utilizan tablas asociativas para implementar la tabla de símbolos (la búsqueda de cualquier símbolo se realiza en un tiempo constante)

Contenido

6 Tema 6. Organización de archivos

- Introducción
- Organización Secuencial
- Organizaciones Indexadas
- Organización Directa. Dispersión
- Clasificación externa

5 Clasificación externa

- Clasificación de datos almacenados en memoria secundaria, es decir, de datos organizados en archivos
- Métodos:
 - ① Clasificación por intercalación
 - ② Clasificación por intercalación múltiple
 - ③ Clasificación polifase

5.1 Clasificación por Intercalación

Consiste en organizar un archivo en fragmentos o secuencias ordenadas cada vez más grandes

Fragmneto de longitud k

Secuencia de k registros clasificados según el valor de su clave
 $r_1, r_2, \dots, r_k \quad / \text{ clave}(r_i) \leq \text{clave}(r_{i+1}) \text{ para } 1 < i < k$

Archivo organizado en fragmentos

Un archivo de n registros está organizado en fragmentos de longitud k si para todo $i \geq 0$ tal que $k * i \leq N$ la secuencia $r_{k*(i-1)+1}, r_{k*(i-2)+1}, \dots, r_{k*i}$ es un fragmento de longitud k

Si n no es divisible por k ($n = p * k + q$) habrá una secuencia final de q registros ($q \leq k$) llamada cola, que será un fragmento de longitud q

Método de clasificación por intercalación

Clasifica los N registros utilizando cuatro archivos de la siguiente forma:

- Se dividen los N registros del archivo inicial en dos archivos F_1 y F_2 , lo más uniformemente. Inicialmente F_1 y F_2 serán archivos organizados en **fragmentos de longitud 1**
- Se combinan los fragmentos de F_1 y F_2 y se distribuyen en otros dos archivos G_1 y G_2 organizados en **fragmentos de longitud 2**
- Se combinan los fragmentos de G_1 y G_2 y se distribuyen en los archivos F_1 y F_2 organizados en **fragmentos de longitud 4**
- Se continua el proceso hasta que el archivo quede clasificado: todo los registros en un archivo con un único **fragmento de longitud N**

Ejemplo

Archivo Inicial	F1 (k=1)	F2 (K=1)	G1 (k=2)	G2 (K=2)	F1 (k=4)	F2 (K=4)	G1 (k=8)	G2 (K=8)	F1 (k=16)	F2 (K=16)	G1 (k=32)	G2 (K=32)
93	93	31	31	3	3	10	3	9	3	8	3	
3	3	5	93	5	5	28	5	13	5	8	5	
28	28	96	28	10	31	40	10	30	9	10	8	
10	10	40	96	40	93	96	28	39	10	10	8	
54	54	85	54	9	9	13	31	54	13	22	9	
65	65	9	85	65	54	30	40	65	28	69	10	
30	30	39	30	13	65	39	93	85	30	77	10	
90	90	13	39	90	85	90	96	90	31		10	
10	10	8	8	69	8	8	8		39		13	
69	69	77	10	77	10	10	8		40		22	
8	8	10	8	22	69	22	10		54		28	
22	22		10		77		10		65		30	
31							10		85		31	
5							22		69		39	
96							69		90		40	
40							77		93		54	
85											65	
9											69	
39											77	
13											85	
8											90	
77											93	
10											96	

Algoritmo de clasificación

Algoritmo clasificación(f:tipoArchivo)

```

1: f1,f2,g1,g2: tipoArchivo
2: k: tipoEntero
3: cambiaEntrada:tipoLógico
4: k ← 1
5: dividir(f, f1,f2)
6: cambiaEntrada ← FALSO
7: mientras k < númeroRegistros hacer
8:   si NO (cambiaEntrada) entonces
9:     intercalación(k,f1,f2,g1,g2)
10:    si no
11:      intercalación(k,g1,g2,f1,f2)
12:    fin si
13:    k ← k*2
14:  cambiaEntrada ← NO(cambiaEntrada)
15: fin mientras
16: si cambiaEntrada entonces
17:   copiar(f1,f)
18:   si no
19:   copiar(g1,f)
20: fin si

```

Algoritmo leerRegistro(f: tipoArchivo, k: tipoEntero, ref finFrag: tipoLógico, ref registro: tipoRegistro, ref númeroRegistros: tipoEntero)

```

1: si númeroRegistros = k O finFichero(f) entonces
2:   finFrag ← VERDADERO
3: si no
4:   leer(f,registro)
5:   númeroRegistros ← númeroRegistros + 1
6: fin si

```

Algoritmo de intercalación

Algoritmo intercalación(*k*: tipoEntero, *e1*, *e2*, *s1*, *s2*: tipoArchivo)

```
1: mientras NO(finFichero(e1)) O NO(finFichero(e2)) hacer
2:   nReg1 ← 0
3:   nReg2 ← 0
4:   finFrag1 ← FASLO
5:   finFrag2 ← FASLO
6:   leerRegistro(e1,k,finFrag1, actual1, nReg1)
7:   leerRegistro(e2,k,finFrag2, actual2, nReg2)
8:   mientras NO(finFrag1) O NO(finFrag2) hacer
9:     si finFrag1 entonces
10:       actual ← actual2
11:       leerRegistro(e2,k,finFrag2, actual2, nReg2)
12:     si no, si finFrag2 entonces
13:       actual ← actual1
14:       leerRegistro(e1,k,finFrag1, actual1, nReg1)
15:     si no, si actual1.clave < actual2.clave entonces
16:       actual ← actual1
17:       leerRegistro(e1,k,finFrag1, actual1, nReg1)
18:     si no
19:       actual ← actual2
20:       leerRegistro(e2,k,finFrag2, actual2, nReg2)
21:     fin si
22:     si cambiaSalida entonces
23:       escribe(actual, s1)
24:     si no
25:       escribe(actual, s2)
26:     fin si
27:   fin mientras
28:   cambiaSalida ← NO(cambiaSalida)
29: fin mientras
```

Análisis clasificación por Intercalación

Incialmente:

F_1	fragmentos $k = 1$	$\frac{n}{2}$ registros
F_2	fragmentos $k = 1$	$\frac{n}{2}$ registros

Intercalación:

	ENTRADA	SALIDA
1 ^a pasada	$F_1(k = 1)$	$G_1(k = 2)$
	$F_2(k = 1)$	$G_2(k = 2)$
2 ^a pasada	$G_1(k = 2)$	$F_1(k = 4)$
	$G_2(k = 2)$	$F_2(k = 4)$
3 ^a pasada	$F_1(k = 4)$	$G_1(k = 8)$
	$F_2(k = 4)$	$G_2(k = 8)$
...		...
pasada "i"	dos archivos (F 's o G 's) organizados organizados en fragmentos de longitud 2^i	

$$\text{Archivo CLASIFICADO} \Rightarrow 2^i \geq n$$

Comportamiento del algoritmo

- Archivo clasificado $\Rightarrow 2^i \geq n \Rightarrow i \geq \log n$
- Se necesitan $\log n$ pasadas de intercalación para clasificar un archivo con n registros:
 - Cada pasada lee/escribe n registros
 - El número de operaciones de entrada salida del proceso completo es $(n \log n / b)$, siendo b el número de registros por bloque
- Clasificación por intercalación es $O(n \log n)$
- Mejora del proceso: Partir de archivos organizados en fragmentos de longitud mayor que 1
 - Paso previo que lea grupos de k registros, los ordene en memoria principal según algún método de clasificación interna (quicksort, ...) y los devuelva a memoria secundaria como un fragmento de longitud k

Observaciones

- El tiempo empleado en leer/escribir un bloque entre M.P. Y M.S. es mayor que el tiempo de proceso de datos de M.P.
- Si solo hay un canal dedicado a la transferencia de datos llegará un momento en que los archivos estén organizados en fragmentos de longitud mayor que el tamaño de un bloque \Rightarrow intercalar dos fragmentos supone leer muchos bloques de cada archivo

5.2 Clasificación por Intercalación Múltiple

- Modificación del método de clasificación por intercalación que puede aplicarse cuando se dispone de más de un canal de transferencia de datos
- Suponiendo $2m$ unidades de disco, cada una con su propio canal de transferencia de datos:
 - Se consideran m archivos de entrada F_1, F_2, \dots, F_m organizados en **fragmentos de longitud** k y m archivos de salida G_1, G_2, \dots, G_m
 - Se leen m fragmentos (uno de cada archivo), se combinan en un fragmento de longitud mk y se van copiando en los archivos de salida
 - Se intercambian los fichero de entrada y de salida y se repite el proceso hasta que la longitud de los fragmentos sea mayor o igual que el número de registros

Clasificación por Intercalación Múltiple

Inicialmente	Pasada 1	Pasada 2	...	Pasada i (i par i impar)
$F_1(k)$	$G_1(mk)$	$F_1(m^2k)$		$F_1(m^i k) G_1(m^i k)$
$F_2(k)$	$G_2(mk)$	$F_2(m^2k)$		$F_2(m^i k) G_2(m^i k)$
...		
$F_m(k)$	$G_m(mk)$	$F_m(m^2k)$		$F_m(m^i k) G_m(m^i k)$

Archivo clasificado $\Rightarrow m^i k \geq n$

- Número de pasadas proporcional a $\log_m n \Rightarrow O(n \log_m n)$
- Aunque se utilizan m ficheros de entrada y m de salida, el proceso de intercalación múltiple en cada momento solo escribe en un fichero de salida
- ¡En cada pasada se desaprovechan $(m-1)$ canales!

5.3 Clasificación Polifase

- Alternativa a la intercalación múltiple utilizando en cada paso **todos** los canales transferencia de datos
- Considerando $m + 1$ canales E/S: m archivos de entrada y un solo archivo de salida
- Proceso para $m = 2$ (dos archivos de entrada y uno de salida): F_1 , F_2 y F_3
 - ① Inicialmente se dispone de dos archivos de entrada (F_1 y F_2) organizados en **fragmentos de longitud 1**
 - ② Se intercalan los registros de F_1 y F_2 en **fragmentos de longitud 2** en el archivo de salida F_3 , hasta que uno de los archivos de entrada quede vacío
 - ③ El archivo que quede vacío en el primer paso se utiliza como salida en el siguiente y en él se intercalan los fragmentos de longitud 2 de F_3 y los restantes de longitud 1 (F_1 y F_2) quedando organizado en **fragmentos de longitud 3**
 - ④ Se repite el paso 3 aumentando la longitud de los fragmentos hasta que el archivo quede clasificado

Ejemplo

F_1	F_2	F_3
Número de fragmentos (Longitud de fragmentos)	Número de fragmentos (Longitud de fragmentos)	Número de fragmentos (Longitud de fragmentos)
13(1)	21(1)	vacío
vacío	8(1)	13(2)
8(3)	vacío	5(2)
3(3)	5(5)	vacío
vacío	2(5)	3(8)
2(13)	vacío	1(8)
1(13)	1(21)	vacío
vacío	vacío	1(34)

- Las longitudes de los fragmentos que se obtienen en cada paso siguen la secuencia de Fibonacci 1, 1, 2, 3, 5, 8, 13, 21,
- Número de fragmentos en archivos iniciales dos números consecutivos de Fibonacci

Contenido

- 1 Tema1. Árboles Generales y Binarios
- 2 Tema 2. Montículos Binarios
- 3 Tema 3. Conjuntos Disjuntos
- 4 Tema 4. Grafos
- 5 Tema 5. Árboles Binarios de Búsqueda
- 6 Tema 6. Organización de archivos
- 7 Tema 7. Organización de Índices

Contenido

7 Tema 7. Organización de Índices

● Introducción

- Organización en ÁRBOLES B+
- Organización en ÁRBOLES B
- Ejercicios

1 Introducción

- Las técnicas más utilizadas en la organización de índices se basan en estructuras arbóreas que son generalizaciones de los árboles binarios de búsqueda y sus variantes
- Árboles binarios adecuados para trabajar en M.P. pero no muy eficientes cuando se trata de búsqueda externa en la que el tiempo de acceso a M.S. es un factor importante
 - Árbol binario de n nodos almacenado en disco $\Rightarrow \log n$ accesos para localizar un nodo
 - Teniendo en cuenta que cada vez que se accede a disco se lee un bloque o página de datos \Rightarrow organizar el árbol en páginas de forma que cada una contenga un determinado número de nodos
- **Idea básica:** dividir el árbol en subárboles (páginas) y representar los subárboles como unidades a las que se accede en bloques

Contenido

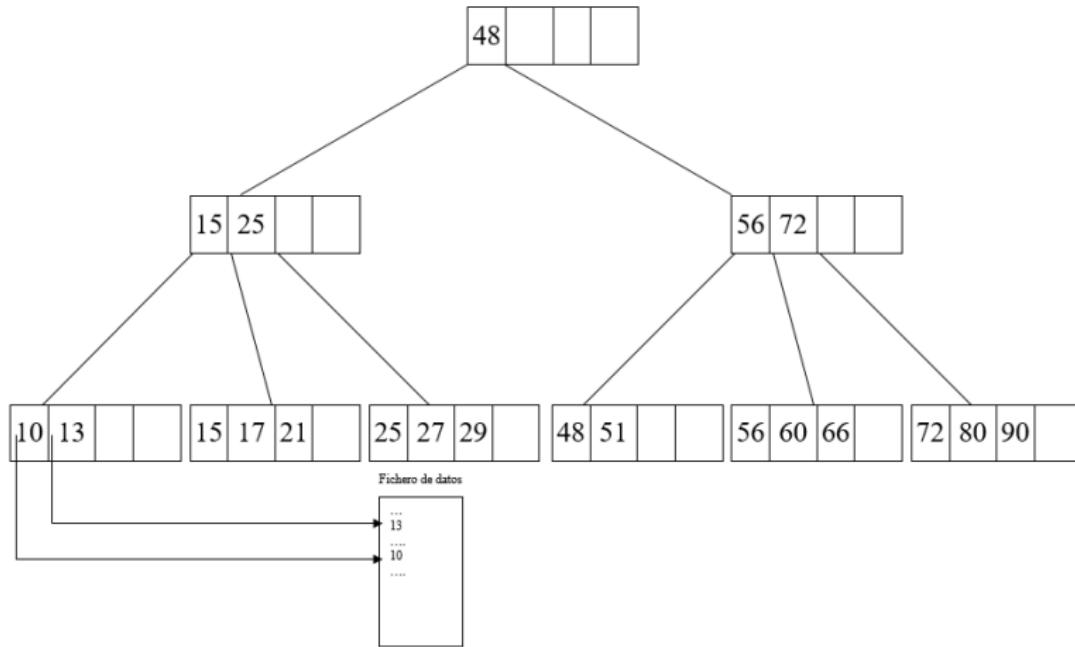
7 Tema 7. Organización de Índices

- Introducción
- Organización en ÁRBOLES B+
- Organización en ÁRBOLES B
- Ejercicios

2 Organización en ÁRBOLES B+

- Un fichero índice organizado en **árbol B+** tiene la estructura de un árbol equilibrado en el que cualquier camino desde la raíz del árbol hasta un nodo hoja tiene la misma longitud (mismo número de niveles)
- Características de un árbol B+ de orden **m**:
 - Cada nodo o página contiene como máximo **2m claves** clasificadas de izquierda a derecha
 - Cada nodo o página, excepto la raíz, tiene como mínimo **m claves**
 - Cada nodo interno tiene **n+1** descendientes, siendo **n** el número de claves del nodo
 - El descendiente i -ésimo de un nodo contiene las claves comprendidas entre la $(i-1)$ -ésima y la i -ésima clave del padre
 - Todas las claves se encuentran en los nodos hoja
 - Todos los nodos hoja se encuentran en el mismo nivel

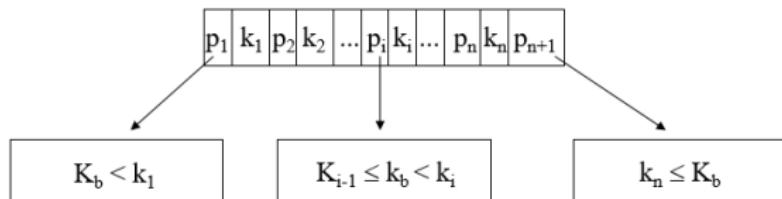
Ejemplo: árbol B+ de orden 2 con tres niveles



Estructura de los nodos internos

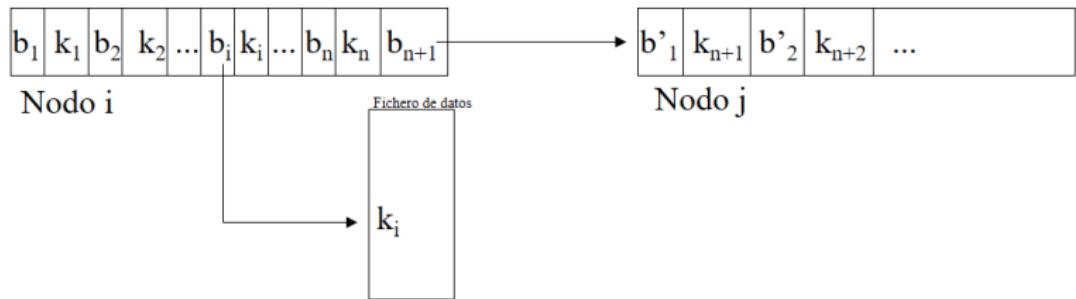
- Los nodos internos forman un índice de varios niveles de los nodos hoja
- Cada nodo interno contiene **n** valores de la clave de búsqueda k_1, k_2, \dots, k_n y (**n+1**) punteros $p_1, p_2, \dots, p_n, p_{n+1}$ tal que:
 - p_1 apunta al subárbol que contiene valores de la clave menores que k_1
 - p_i ($1 < i \leq n$) señala al subárbol que contiene valores de la clave menores que k_i y mayores o iguales que k_{i-1}
 - p_{n+1} apunta al subárbol que contiene valores de la clave mayores o iguales que k_n

$p_1 \rightarrow k_b < k_1$
$p_2 \rightarrow k_1 \leq k_b < k_2$
...
$p_i \rightarrow k_{i-1} \leq k_b < k_i$
...
$p_n \rightarrow k_{n-1} \leq k_b < k_n$
$p_{n+1} \rightarrow k_n \leq k_b$



Estructura de los nodos hoja

- Estructura similar a la de los nodos internos, con la excepción de que los punteros señalan a bloques de registros en el fichero de datos
- Cada nodo hoja contiene **n** valores de la clave de búsqueda k_1, k_2, \dots, k_n y **(n+1)** punteros $b_1, b_2, \dots, b_n, b_{n+1}$ tal que:
 - ① Si $i < j \Rightarrow K_i < k_j$
 - ② b_i ($1 < i \leq n$) señala al bloque de registros que contiene el valor k_i de la clave de búsqueda
 - ③ b_{n+1} se utiliza para encadenar los nodos hoja según el orden de la clave. Esto permite procesar eficientemente el fichero de forma secuencial.



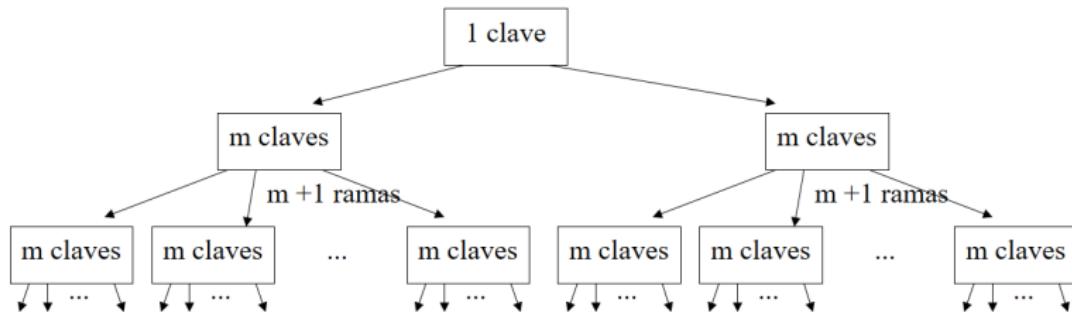
Operación de Búsqueda

Para recuperar un registro r con clave k_b se sigue el camino desde la raíz hasta el nodo hoja que contiene a k_b según el siguiente proceso:

- ① Se copia el nodo raíz de M.S. a M.P.
- ② Se busca la posición relativa de k_b respecto a las claves del nodo k_1, k_2, \dots, k_n y se copia el nodo correspondiente de M.S a M.P.:
 - si $k_b < k_1$ se copia el nodo al que apunta p_1
 - si $k_{i-1} \leq k_b < k_i$ se copia el nodo al que apunta p_i
 - si $k_n \leq k_b$ se copia el nodo al que apunta p_{n+1}
- ③ Se repite el paso 2 con el nodo copiado hasta llegar a un nodo hoja
- ④ El puntero del nodo hoja dará la dirección del bloque del archivo que contiene el registro buscado, si existe. Se copia el bloque correspondiente a M.P. y se explora

Análisis

- Al procesar una búsqueda se recorre el árbol desde la raíz hasta un nodo hoja \Rightarrow el número de accesos depende de la altura del árbol
- Peor caso: si cada nodo del árbol contiene el número mínimo de claves, estas se distribuyen generando un árbol de altura máxima
- Para un árbol B+ de orden m, en el peor de los casos, se tiene:



Altura del árbol en el peor de los casos

Número mínimo de descendientes en cada nivel

altura	número mínimo de descendientes
0	2
1	$(m + 1) + (m + 1) =$
2	$(m + 1)(m + 1) + (m + 1)(m + 1) =$
...	...
h	$2(m + 1)^h$

- Como todas las claves están en los nodos hoja, un árbol B+ que contenga n claves tendría $n+1$ descendientes desde su nivel hoja
- El número de descendientes no puede ser menor que el número de descendientes para el peor de los casos, por tanto:

$$n + 1 \geq 2(m + 1)^h \Rightarrow h \leq \lg_{m+1}((n + 1)/2))$$

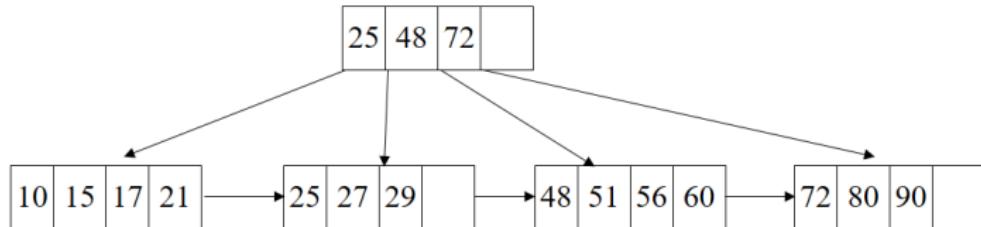
Inserción de claves en un árbol B+ de orden m

Empleando la misma técnica que para la búsqueda se localiza el nodo hoja en el que debe insertarse la clave y se tienen en cuenta los siguientes casos:

- ① Número de claves del nodo hoja menor que número máximo ($n < 2m$)
 - Se inserta el valor de la clave en el nodo hoja de forma que siga clasificado según la clave
 - Se añade el registro al fichero y se copia su dirección en el lugar correspondiente del nodo hoja
- ② El nodo hoja ya contiene el número máximo de claves ($n=2m$) y no queda espacio para insertar una nueva clave \Rightarrow **Partición de Nodo**
 - Crear un nuevo nodo hoja y pasar a él la mitad de las claves del nodo desbordado
 - Insertar el nuevo nodo en la estructura de árbol B+: almacenar valor más pequeño de la clave del nuevo nodo en el nodo padre teniendo en cuenta de nuevo los casos 1 y 2

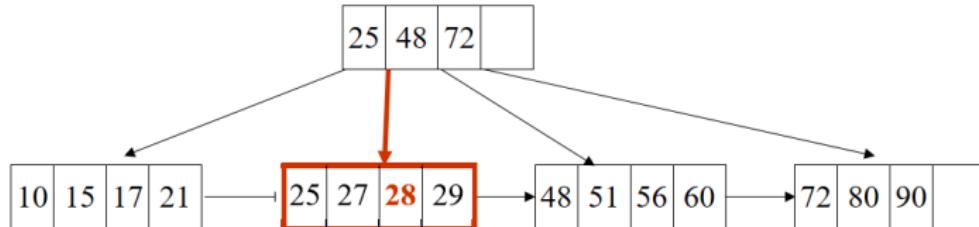
Si se desbordan todos los nodos a lo largo de la ruta hacia la raíz y es necesario dividir la raíz, el árbol adquiere un nivel más y aumenta su altura
La partición de un nodo interno no produce duplicidad de las claves

Ejemplo



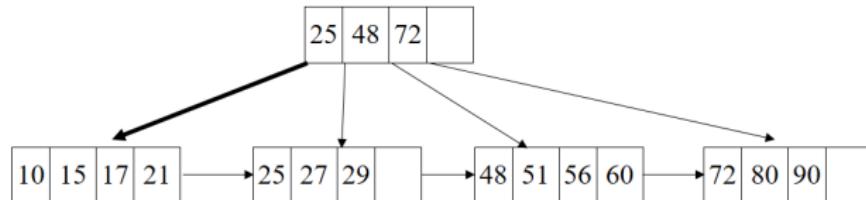
Insertar las claves: **28, 13 y 66**

28 ⇒ el nodo hoja correspondiente tiene espacio, se inserta



Inserción de la clave 13

13 ⇒ el nodo hoja correspondiente no tiene espacio ⇒ **PARTICIÓN DE NODO**

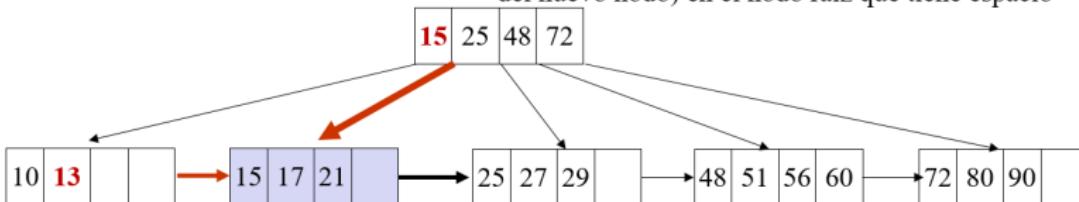


10 **13** 15 17 21



nuevo nodo que hay que insertar en la estructura

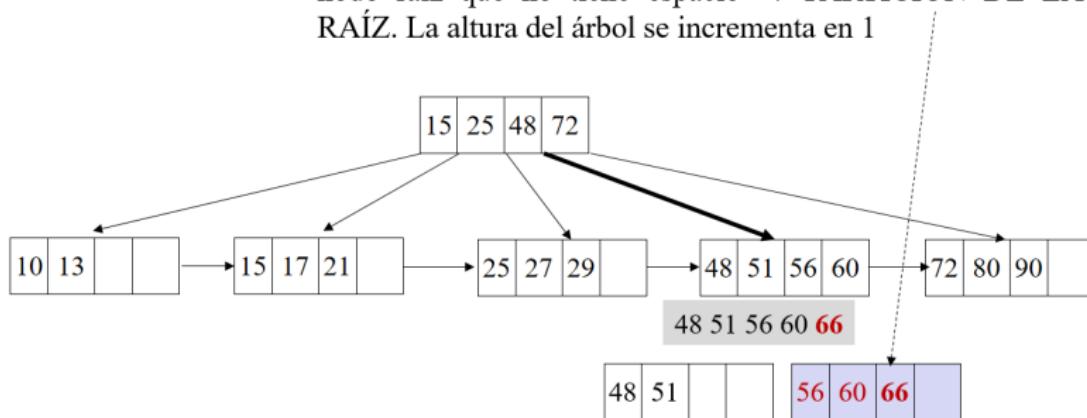
de árbol B+: insertar la clave 15 (y la dirección del nuevo nodo) en el nodo raíz que tiene espacio



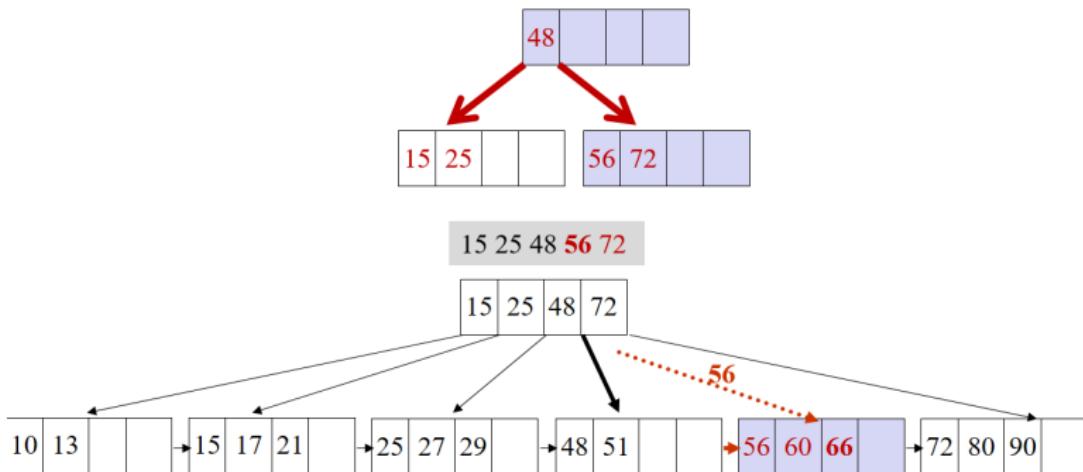
Inserción de la clave 66

66 ⇒ el nodo hoja correspondiente no tiene espacio ⇒ PARTICIÓN DE NODO

nuevo nodo que hay que insertar en la estructura de árbol B+: insertar la clave 56 (y la dirección del nuevo nodo) en el nodo raíz que no tiene espacio ⇒ PARTICIÓN DE LA RAÍZ. La altura del árbol se incrementa en 1

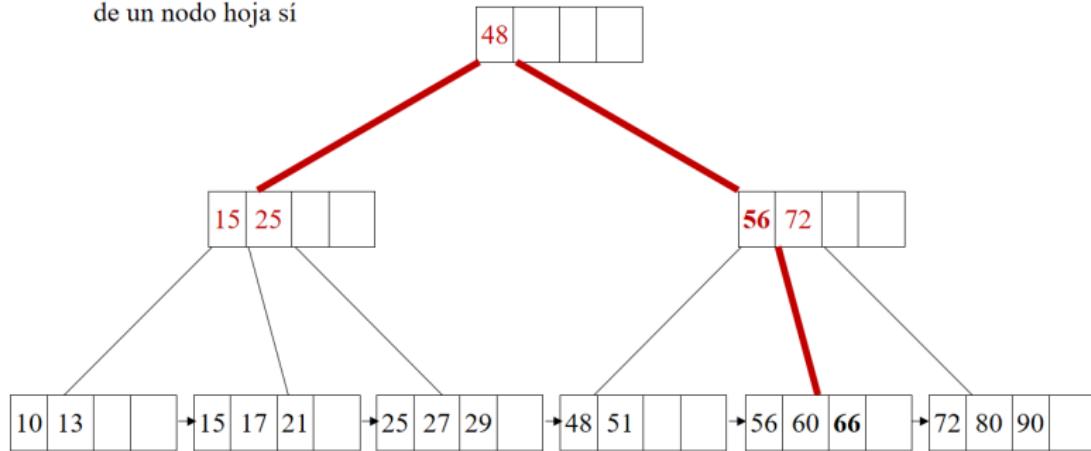


Partición de la raíz



Árbol resultante

La partición de un nodo interno no produce duplicidad de las claves mientras que la de un nodo hoja sí



Eliminación de claves en un árbol B+ de orden m

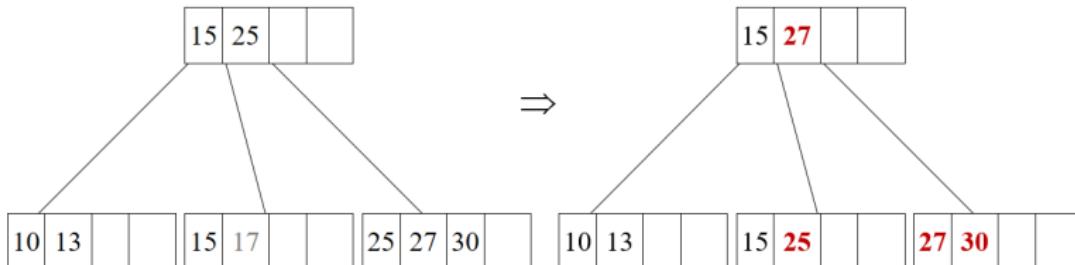
Empleando la misma técnica que para la búsqueda se localiza el nodo hoja en el que se encuentra la clave que se desea eliminar y se elimina teniendo en cuenta los siguientes casos:

- ① Si el número de claves que quedan en el nodo hoja es mayor o igual que el número mínimo permitido ($n \geq m$) termina la operación de eliminación. Las claves en los nodos internos no se modifican aunque sean una copia de la clave eliminada
- ② Si el número de claves que quedan en el nodo hoja es menor que el número mínimo de claves permitido ($n < m$) se examina el nodo hermano y se distinguen otros dos casos:
 - 2.1 Si el número de claves en el nodo hermano (n') es mayor que el número mínimo permitido ($n' > m$) \Rightarrow **Distribución de claves** y punteros entre los dos nodos
 - 2.2 Si el nodo hermano tiene el número mínimo de claves ($n' = m$) \Rightarrow **Fusión de Nodos** que puede llegar hasta la raíz reduciendo la altura del árbol en una unidad

Caso 2.1 Distribución de claves

- Se distribuyen las claves y punteros entre los dos nodos de forma que ambos tengan al menos el número mínimo permitido (m)
- A continuación se debe modificar el valor de la clave en el nodo padre manteniendo el orden de búsqueda

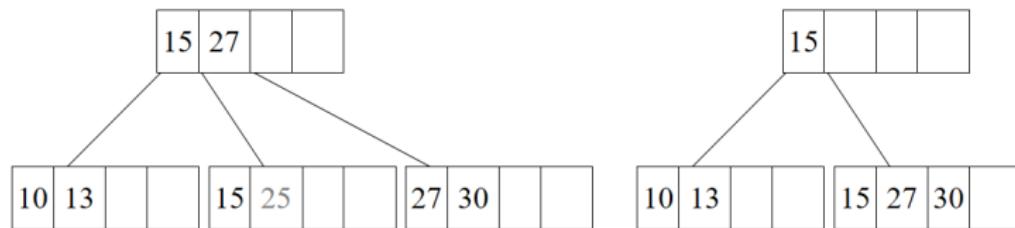
Ejemplo: eliminar la clave 17



Caso 2.2 Fusión de nodos

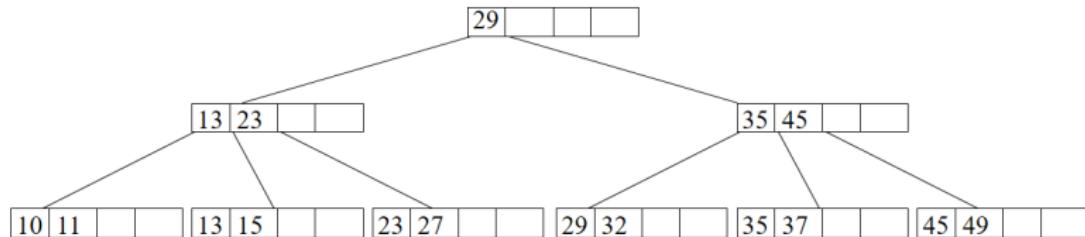
- Se combinan las claves y punteros de los dos nodos en uno solo
- Supone eliminar un nodo de la estructura de árbol B+ y, por tanto, eliminar la clave y el puntero correspondiente en el nodo padre

Ejemplo: eliminar la clave 25

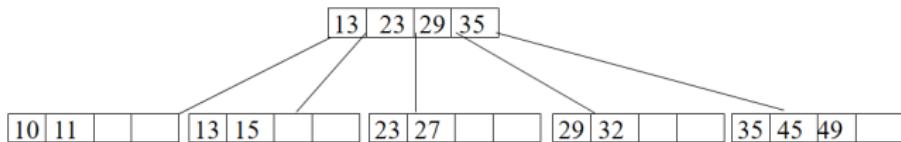


- Si la eliminación supone que el nodo padre quede con menos claves que las mínimas permitidas se repetirá de forma recursiva el algoritmo de eliminación
- Los efectos de la eliminación se pueden propagar hasta la raíz: si es necesaria la fusión de sus dos únicos hijos, el nodo combinado forma la nueva raíz y el árbol disminuye su altura

Ejemplo de eliminación con disminución de altura



Eliminar clave 37: fusión de nodos hasta la raíz



Contenido

7 Tema 7. Organización de Índices

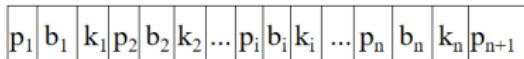
- Introducción
- Organización en ÁRBOLES B+
- Organización en ÁRBOLES B
- Ejercicios

3 Organización en ÁRBOLES B

- Un fichero índice organizado en árbol B tiene la estructura de un árbol equilibrado en el que cualquier camino desde la raíz del árbol hasta un nodo hoja tiene la misma longitud (mismo número de niveles)
- Características comunes a los árboles B+:
 - Cada nodo contiene n claves / $m \leq n \leq 2m$
 - El nodo raíz puede contener n claves / $1 \leq n \leq 2m$
 - Las claves en los nodos se encuentran clasificadas
 - Todos los nodos hoja se encuentran en el mismo nivel
- Características diferentes:
 - Los valores posibles de la clave aparecen una sola vez, es decir, se elimina la característica de que todas las claves se encuentran en los nodos hoja \Rightarrow elimina el almacenamiento redundante de claves repetidas

Estructura de los nodos internos

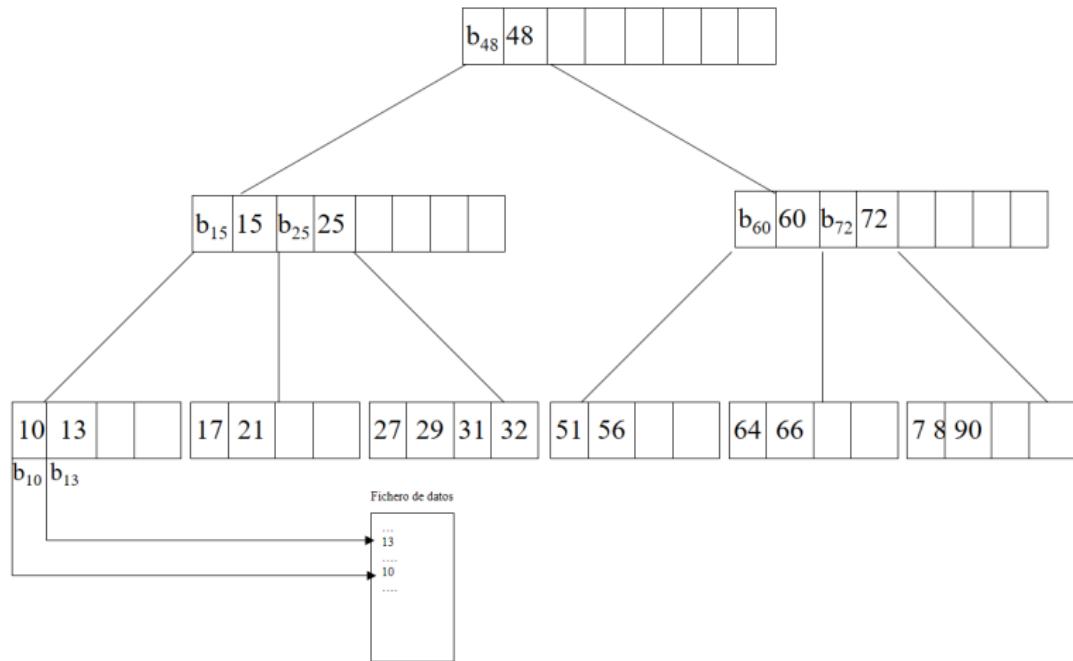
- Un índice organizado en árbol B, se almacena utilizando menos nodos que organizado en árbol B+
- Como las claves que aparecen en un nodo interno, no pueden aparecer de nuevo en el árbol, debeadirse un campo para indicar donde se almacena el registro correspondiente \Rightarrow Distinta estructura para los nodos internos



p_1	\rightarrow	$k_b < k_1$
p_2	\rightarrow	$k_1 < k_b < k_2$
\dots		
p_i	\rightarrow	$k_{i-1} < k_b < k_i$
\dots		
p_n	\rightarrow	$k_{n-1} < k_b < k_n$
p_{n+1}	\rightarrow	$k_n < k_b$

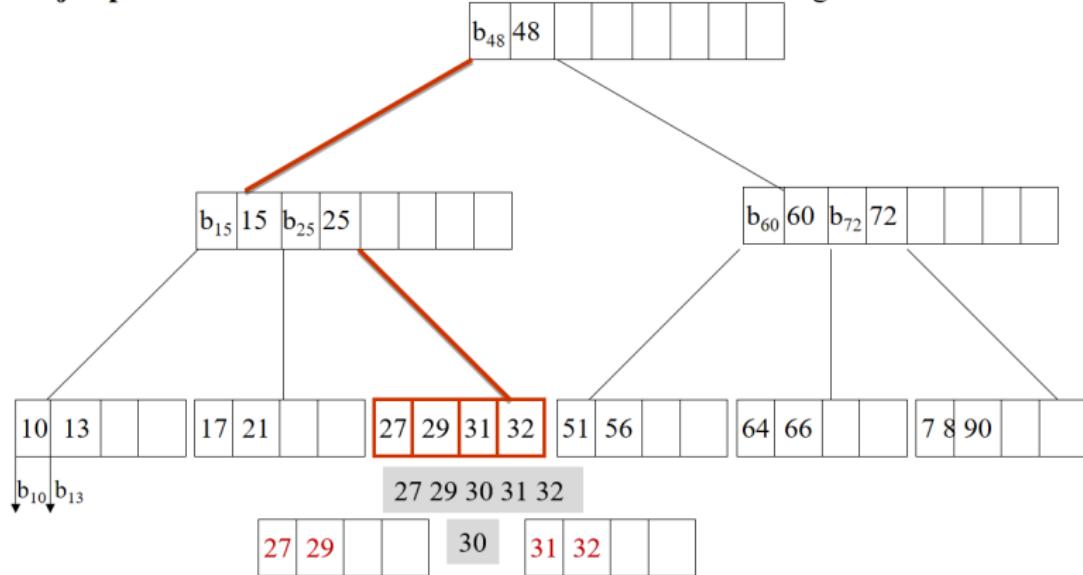
$b_1 \rightarrow$ dirección registro con clave k_1
 $b_2 \rightarrow$ dirección registro con clave k_2
 \dots
 $b_i \rightarrow$ dirección registro con clave k_i
 \dots
 $b_n \rightarrow$ dirección registro con clave k_n

Ejemplo: árbol B de orden 2

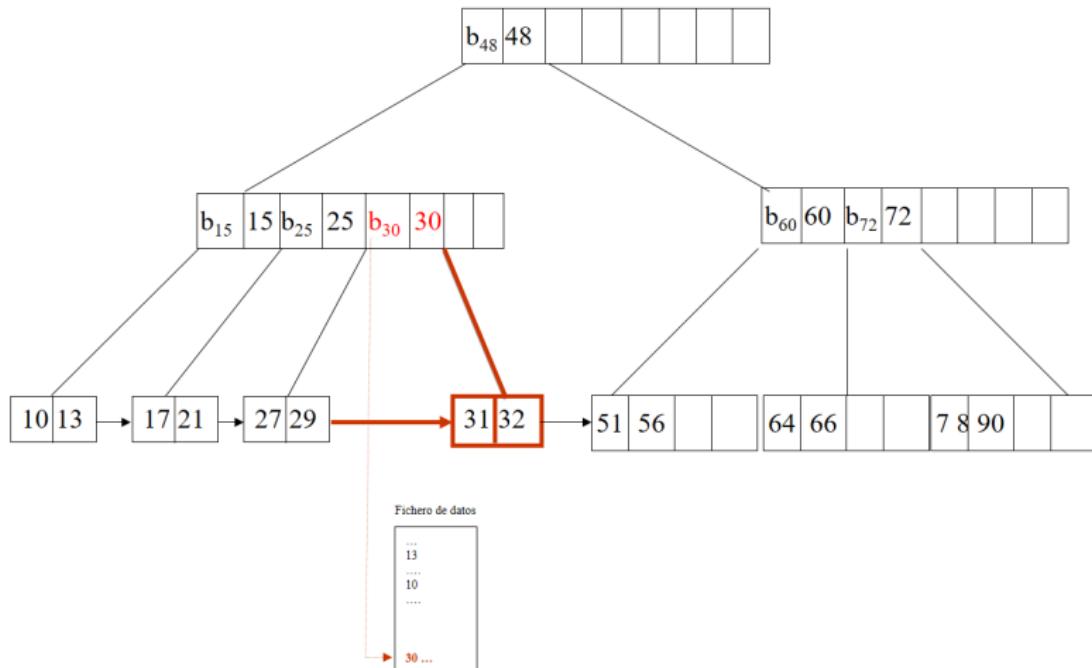


Inserción en árbol B similar a inserción en B+

Ejemplo: insertar la clave 30 en el árbol B de orden 2 de la figura



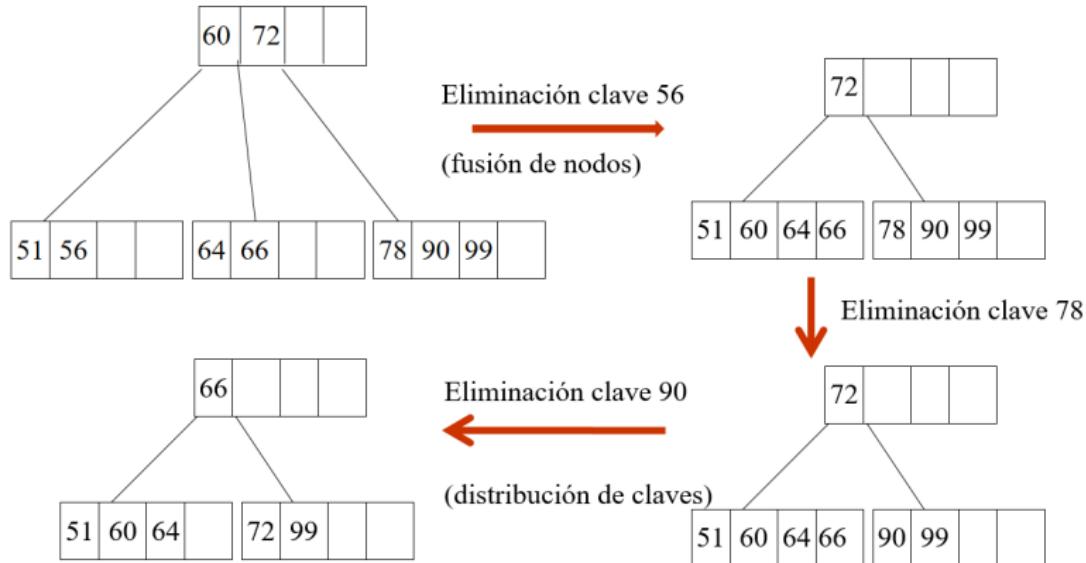
Inserción en árbol B similar a inserción en B+ (continuación)



Eliminación en Árbol B

- La eliminación es más complicada que en los árboles B+ ya que la clave puede aparecer en un nodo hoja o en un nodo interno
 - **Nodo hoja:** se elimina y se comprueba el número de claves que quedan
 - Si $n \geq m$ Acaba proceso
 - Si $n < m$ Fusión de nodos o distribución de claves
 - **Nodo interno:** se elimina y se sustituye por clave más derecha del subárbol izquierdo o mas izquierda que subárbol derecho ¡clave que hay que eliminar! ¡y verificar “n”!
 - Si $n \geq m$ Acaba proceso
 - Si $n < m$ Fusión de nodos o distribución de claves

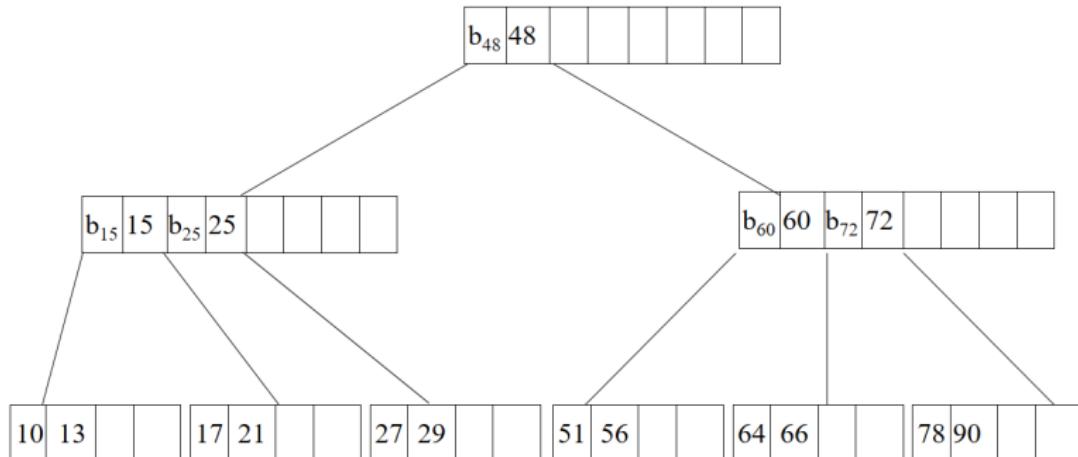
Ejemplo1. Eliminación árbol B en nodo hoja



Ejemplo2. Eliminación árbol B en nodo interno

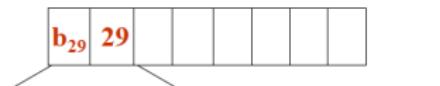
Eliminar clave **48**

1. sustituir por clave más derecha subárbol izquierdo(29)

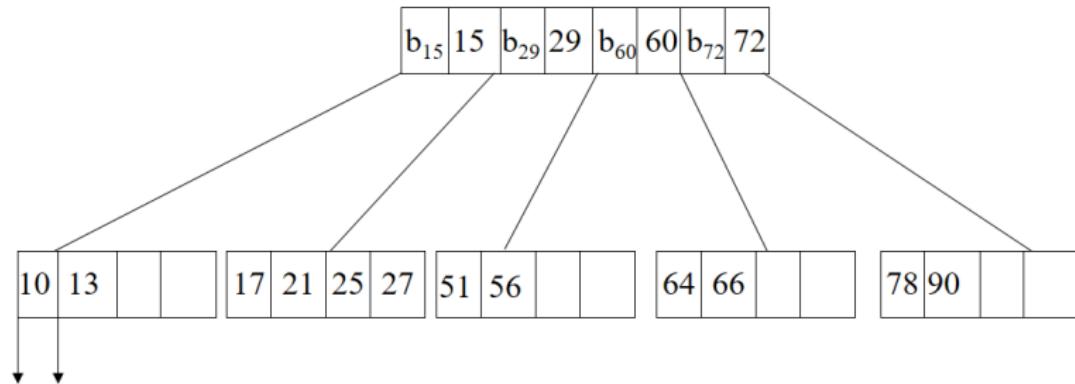


Eliminar clave **48**:

1. sustituir por clave más derecha subárbol izquierdo(29)
2. eliminar 29 ... fusión de nodos ... eliminar nodo ... (eliminar clave 25) ...



Árbol resultante después de eliminar clave 48



Comparación de las dos organizaciones

- Ventajas de organización en árbol B frente a B+:
 - elimina almacenamiento redundante de claves ⇒ menos espacio de almacenamiento
 - búsquedas ligeramente más rápidas (no siempre es necesario llegar a la base del árbol)
- Desventajas:
 - los nodos internos son más grandes que los nodos hoja
 - la eliminación es más complicada que en los árboles B+ ya que la clave puede aparecer en un nodo hoja o en un nodo interno
- En índices extensos, las ventajas de los árboles B no tienen importancia y suele preferirse la sencillez de los árboles B+

Contenido

7 Tema 7. Organización de Índices

- Introducción
- Organización en ÁRBOLES B+
- Organización en ÁRBOLES B
- Ejercicios

Ejercicios

Dada la secuencia de claves enteras **19, 5, 90, 9, 12, 15, 35, 45, 95, 20, 50 y 17**

- ① Dibujar el **árbol B** de orden 3 que se obtiene al ir insertando las claves en el orden que indica la secuencia
- ② En el árbol B resultante de la pregunta anterior, eliminar la clave 12 y dibujar el árbol resultante. Eliminar después la clave 5 y dibujar el árbol resultante
- ③ Dibujar el árbol B+ de orden 3 que se obtiene al ir insertando las claves en el orden que indica la secuencia
- ④ En el árbol B+ resultante de la pregunta anterior, eliminar la clave 12 y dibujar el árbol resultante. Eliminar después la clave 5 y dibujar el árbol resultante
- ⑤ Repetir los ejercicios anteriores para árboles de orden 2
- ⑥ Eliminar la clave 72 del árbol B de la transparencia 28