

# TEMA 1

## Introducción a los TADs.

### Los tipos lineales

PROGRAMACIÓN Y ESTRUCTURAS DE DATOS

## Introducción. Los tipos lineales

- 1. Introducción a los TADs
- 2. Vectores
- 3. Listas
- 4. Pilas
- 5. Colas

## 1. Introducción a los TADs

- TAD: Tipo Abstracto de Datos
- *Tipo de datos:*
  - Clasifica los objetos de los programas (**variables, parámetros, constantes**) y determina los valores que pueden tomar
  - También determina las operaciones que se aplican
    - Entero: operaciones aritméticas enteras (suma, resta, ...)
    - Booleano: operaciones lógicas (y, o, ...)
- *Abstracto:*
  - La manipulación de los datos sólo dependen del comportamiento descrito en su especificación (**qué hace**) y es independiente de su implementación (**cómo se hace**)
  - Una especificación → Múltiples implementaciones

3

## 1. Introducción a los TADs

- Especificación de un TAD:
  - Consiste en establecer las propiedades que lo definen
  - Para que sea útil debe ser:
    - Precisa: sólo produzca lo imprescindible
    - General: sea adaptable a diferentes contextos
    - Legible: sea un comunicador entre especificador e implementador
    - No ambigua: evite problemas de interpretación
  - Definición informal (lenguaje natural) o formal (algebraica)

4

## 1. Introducción a los TADs

- Implementación de un TAD:
  - Consiste en determinar una representación para los valores del tipo y en codificar sus operaciones a partir de esta representación
  - Para que sea útil debe ser:
    - Estructurada: facilita su desarrollo
    - Eficiente: optimiza el uso de recursos → Evaluación de distintas soluciones mediante la complejidad (espacial y temporal)
    - Legible: facilita su modificación y mantenimiento

5

## 1. Introducción a los TADs

### ESPECIFICACIÓN ALGEBRAICA (I)

- Especificación algebraica (ecuacional): establece las propiedades de un TAD mediante ecuaciones con variables cuantificadas universalmente, de manera que las propiedades dadas se cumplen para cualquier valor que tomen las variables
- Pasos:
  - Identificación de los objetos del TAD y sus operaciones (declaración del TAD, módulos que usa, parámetros)
  - Definición de la firma (sintaxis) de un TAD (nombre del TAD y perfil de las operaciones)
  - Definición de la semántica (significado de las operaciones)
- Operación: es una función que toma como parámetros (entrada) cero o más valores de diversos tipos, y produce como resultado un solo valor de otro tipo. El caso de cero parámetros representa una constante del tipo de resultado

6

## 1. Introducción a los TADs

### ESPECIFICACIÓN ALGEBRAICA (II)

MODULO ... USA ... (bool, entero, natural...)

PARAMETRO TIPO ...

OPERACIONES

...

...

FPARAMETRO

TIPO (GÉNERO) ...

OPERACIONES

...

...

FMODULO

MODULO VECTOR USA BOOL, ENTERO

//en todas las ecuaciones,  $c \leq i, j \leq f$

PARAMETRO TIPO item

OPERACIONES

$c, f: \rightarrow \text{int}$  //límites inf. y sup.  
error( )  $\rightarrow$  item

FPARAMETRO

TIPO vector

OPERACIONES

crear( )  $\rightarrow$  vector  
asig( vector, int, item )  $\rightarrow$  vector  
recu( vector, int )  $\rightarrow$  item  
esvaciaPOS( vector, int )  $\rightarrow$  bool

7

## 1. Introducción a los TADs

### ESPECIFICACIÓN ALGEBRAICA (III)

MODULO NATURAL1

TIPO natural

OPERACIONES

cero :  $\rightarrow$  natural

suc : natural  $\rightarrow$  natural

FMODULO

Mediante aplicación sucesiva de cero y suc se obtienen los distintos valores del tipo:

cero, suc(cero), suc(suc(cero)), ...

8

## 1. Introducción a los TADs

### ESPECIFICACIÓN ALGEBRAICA (IV)

MODULO NATURAL2

TIPO natural

OPERACIONES

cero : → natural

suc : natural → natural

sum : natural natural → natural

FMODULO

¿sum(cero, suc(cero)) y suc(cero) denotan valores distintos?

¿ sum(cero, suc(cero)) y sum(suc(cero), cero) denotan el mismo valor?

9

## 1. Introducción a los TADs

### ESPECIFICACIÓN ALGEBRAICA (V)

- Solución:

- Utilización de ecuaciones de la forma  $t_1 = t_2$ , donde  $t_1$  y  $t_2$  son términos sintácticamente correctos del mismo tipo
- Semánticamente, expresa que el valor construido mediante el término  $t_1$  es el mismo que el valor construido mediante el término  $t_2$
- Para no tener que escribir infinitas ecuaciones, se admite que los términos que aparecen en una ecuación tengan variables

10

## 1. Introducción a los TADs

### ESPECIFICACIÓN ALGEBRAICA (VI)

MODULO NATURAL3

TIPO natural

OPERACIONES

cero : → natural

suc : natural → natural

suma : natural natural → natural

VAR

x, y: natural

ECUACIONES

suma(x, cero) = x

suma(cero, x) = x

suma(x, suc(y)) = suc(suma(x, y))

FMODULO

11

## 1. Introducción a los TADs

### EJERCICIOS

- Sea el conjunto de los números *naturales* con las operaciones *cero* y *suc*. Define la sintaxis y la semántica de las operaciones “==” y “<=” que permiten realizar una ordenación de los elementos del conjunto

12

## 1. Introducción a los TADs

### EJERCICIOS

- Completa en esta misma hoja las ecuaciones que aparecen a continuación y que expresan el comportamiento de las operaciones de: *resta* en el conjunto de los números Naturales en el que sólo existen las operaciones *cero*:  $\rightarrow \text{natural}$  y la operación *suc*:  $\text{natural} \rightarrow \text{natural}$  (devuelve el sucesor de un número Natural). Se asume que el primer operando de la *resta* es siempre mayor o igual que el segundo.

*resta* (*natural,natural*)  $\rightarrow$  *natural*

*resta*: *natural natural*  $\rightarrow$  *natural*

*resta*(, ) = .....

*resta*(, ) = .....

13

## 1. Introducción a los TADs

### ESPECIFICACIÓN ALGEBRAICA (VII)

- ¿Cómo podemos estar seguros de que no son necesarias más ecuaciones?
- Propiedades importantes: *consistencia* y *completitud*
  - Si se ponen ecuaciones de más, se pueden igualar términos que están en clases de equivalencia diferentes, mientras que si se ponen de menos, se puede generar un número indeterminado de términos incongruentes con los representantes de las clases existentes

14

# 1. Introducción a los TADs

## ESPECIFICACIÓN ALGEBRAICA (VIII)

- Clasificación de las operaciones:
  - **Constructoras:** devuelven un valor del tipo
    - Generadoras: permiten generar, por aplicaciones sucesivas, todos los valores del TAD a especificar
    - Modificadoras: el resto
  - **Consultoras:** devuelven un valor de un tipo diferente
- En general, las operaciones modificadoras y consultoras se especifican en términos de las generadoras. En ocasiones, una operación modificadora puede especificarse en términos de otras modificadoras o consultoras. Diremos que se trata de una **operación derivada**

15

# 1. Introducción a los TADs

## ESPECIFICACIÓN ALGEBRAICA (IX)

- **Ecuación condicional:** es equivalente a un conjunto finito de ecuaciones no condicionales

```

si (n1 <> n2) entonces
    saca(añade(s, n1), n2) = añade(saca(s, n2), n1)
sino
    saca(añade(s, n1), n2) = saca(s, n2)
fsi
  
```

```

saca(añade(s, n1), n2) =
  si (n1 <> n2) entonces   añade(saca(s, n2), n1)
                            sino      saca(s, n2)
  fsi
  
```

16

## 1. Introducción a los TADs

### ESPECIFICACIÓN ALGEBRAICA (X)

- Operaciones auxiliares: se introducen en una especificación para facilitar su escritura y legibilidad. Son invisibles para los usuarios del TAD (también se les llama ocultas o privadas)

17

## 1. Introducción a los TADs

### ESPECIFICACIÓN ALGEBRAICA (XI)

- Tratamiento de errores: puede ocurrir que alguna operación sea una función parcial (no se puede aplicar sobre ciertos valores del dominio de los datos)

#### MODULO NATURAL4

TIPO natural

OPERACIONES

cero :  $\rightarrow$  natural

suc, pred : natural  $\rightarrow$  natural

suma, mult : natural natural  $\rightarrow$  natural

VAR x, y: natural;

ECUACIONES

suma(cero, x) = x

suma(x, cero) = x

suma(x, suc(y)) = suc(suma(x, y))

mult(cero, x) = cero

mult(x, cero) = cero

mult(suc(y), x) = suma(mult(y, x), x)

pred(suc(x)) = x

FMODULO

¿Cuánto vale pred(cero)?

18

# 1. Introducción a los TADs

## ESPECIFICACIÓN ALGEBRAICA (XII)

- Tratamiento de errores:
  - Se añade una constante a la firma que modeliza un valor de error:  $\text{error}_{\text{nat}} \rightarrow \text{natural}$
  - Se añade una ecuación que completa la especificación de pred:  $\text{pred}(\text{cero}) = \text{error}_{\text{nat}}$
  - Se supondrá que los valores sobre los que se aplica una operación en una ecuación normal están libres de error

19

# 1. Introducción a los TADs

## ESPECIFICACIÓN ALGEBRAICA (XIII)

### MODULO NATURAL4

TIPO natural

OPERACIONES

**$\text{error}_{\text{nat}} : \rightarrow \text{natural}$**

cero :  $\rightarrow \text{natural}$

suc, **pred** :  $\text{natural} \rightarrow \text{natural}$

suma, **mult** :  $\text{natural} \text{ natural} \rightarrow \text{natural}$

VAR x, y: natural;

ECUACIONES

suma(cero, x) = x

suma(x, cero) = x

suma(x, suc(y)) = suc(suma(x, y))

mult(cero, x) = cero

mult(x, cero) = cero

mult(suc(y), x) = suma(mult(y, x), x)

pred(suc(x)) = x

**pred (cero) =  $\text{error}_{\text{nat}}$**

FMODULO

20

## 1. Introducción a los TADs

### IMPLEMENTACIÓN (I)

- Dada una especificación de un tipo, se pueden construir diversas implementaciones
- Cada implementación se define en un módulo diferente, llamado módulo de implementación
- La construcción de estos módulos consta de dos fases:
  - Elección de una representación para los diferentes tipos definidos en la especificación
  - Codificación de las operaciones en términos de la representación elegida

21

## 1. Introducción a los TADs

### IMPLEMENTACIÓN (II)

- Mecanismos de abstracción en los lenguajes de programación:
  - Encapsulamiento de la representación del TAD
  - Ocultación de la información, para limitar las operaciones posibles sobre el TAD
  - Genericidad, para lograr implementaciones genéricas válidas para distintos tipos
  - Herencia, para reutilizar implementaciones
- Los lenguajes de programación tradicionales (Fortran, Basic, Pascal, C) resultan ineficientes para utilizar los mecanismos de abstracción
- Es necesario emplear lenguajes modernos (ADA, C++, Java, C#)

22

## 1. Introducción a los TADs

Preguntas de tipo test: Verdadero vs. Falso

- EsVacia: PILA → BOOLEAN. Si P y Q son pilas:  $Q = \text{EsVacia}(P)$ , es un uso sintácticamente correcto de la operación
- En la especificación de un TAD, una operación consultora devuelve un valor del tipo definido
- Sea el siguiente TAD:

*MÓDULO NATURALEXAMEN*

*TIPO natural*

*OPERACIONES*

*uno: → natural; siguiente: natural → natural*

*sumar: natural natural → natural*

*FMÓDULO*

Si  $N$  es un *natural*:  $N = \text{sumar}(\text{uno}, \text{siguiente}(\text{uno}))$  es un uso sintácticamente incorrecto de la operación *sumar*

23

## 2. Vectores

- Un vector es un conjunto ordenado de pares <índice, valor>. Para cada índice definido dentro de un rango finito existe asociado un valor. En términos matemáticos, es una correspondencia entre los elementos de un conjunto de índices y los de un conjunto de valores

24

## 2. Vectores

### ESPECIFICACIÓN ALGEBRAICA

```

MODULO VECTOR USA BOOL, ENTERO
//en todas las ecuaciones, c ≤ i, j ≤ f
PARAMETRO TIPO item
    OPERACIONES
        c, f: → int //límites inf. y sup.
        error( ) → item
FPARAMETRO
TIPO vector
OPERACIONES
    crear( ) → vector
    asig( vector, int, item ) → vector
    recu( vector, int ) → item
    esvaciapos( vector, int ) → bool

```

```

VAR v: vector; i, j: int; x, y: item;
ECUACIONES
si ( i < > j ) entonces
    asig( asig( v, i, x ), j, y ) = asig( asig( v, j, y ), i, x )
si no asig( asig( v, i, x ), j, y ) = asig( v, i, y ) fsi

    recu( crear( ), i ) = error( )
    recu( asig(v, i, x ), j )
    si ( i == j ) entonces x
    si no recu( v, j ) fsi

    esvaciapos( crear( ), i ) = CIERTO
    esvaciapos( asig( v, i, x ), j )
    si ( i == j ) entonces FALSO
    si no esvaciapos( v, j ) fsi

```

**FMODULO**

25

## 2. Vectores

### REPRESENTACIÓN DE VECTORES

```

//Vector de item

const int kTam = 10;
class TVector {
    friend ostream& operator << ( ostream&, TVector& );

public:
    TVector( );
    TVector( const TVector &v );
    ~TVector( );
    TVector& operator =( TVector &v );

    TItem & Recu( int i );
    void Asig( int i, TItem c );
    bool Esvaciapos( int i );

private:
    TItem fv[ kTam ]; //tamaño fijo
    // TItem *fv; tamaño dinámico
    int fLong;
};

```

Cambio de Asig y Recu por la sobrecarga del operador corchete:

TItem & operator [] ( int i );

**TItem&**

```

TVector::operator[](int indice){
    if (indice>=1 && indice<=fLong)
        return (fv[indice-1]);
    else
        return (error); }

```

26

## 2. Vectores

EJERCICIOS *eliminar*

- Sea un vector de números naturales. Utilizando exclusivamente las operaciones *asignar* y *crear*, define la sintaxis y la semántica de la operación *eliminar* que borra las posiciones pares del vector marcándolas con “0” (para calcular el resto de una división, se puede utilizar la operación MOD)

27

## 2. Vectores

EJERCICIOS *operación M*

- Dada la sintaxis y la semántica de la operación M que actúa sobre un vector:

$M(\text{vector}) \rightarrow \text{vector}$

Var  $v$ : vector;  $i$ : int;  $x$ : item;

$M(\text{crear}()) = \text{crear}()$

si  $i == 1$  entonces

$M(\text{asig}(v, i, x)) = M(v)$

si no  $M(\text{asig}(v, i, x)) = \text{asig}(M(v), i - 1, x)$

a) Aplicar la operación M al siguiente vector:

$\text{asig}(\text{asig}(\text{asig}(\text{crear}(), 3, a), 1, b), 2, c)$

b) Explicar en un párrafo qué es lo que hace la operación M

28

## 2. Vectores

### EJERCICIOS *palíndromo*

- Utilizando las operaciones definidas en clase para la definición del tipo vector definir la sintaxis y la semántica de la operación *palíndromo* que indica si un vector de naturales con 100 elementos es palíndromo. Por ejemplo, el vector 1,25,12,3,3,12,25,1 es palíndromo (se ha simplificado el ejemplo con un vector de 8 elementos). IMPORTANTE: se asume que el vector está creado con tamaño 100, está lleno y el rango de las posiciones es de 1 a 100 (en este orden).

29

## 2. Vectores

### Preguntas de tipo test: Verdadero vs. Falso

- El tipo de datos vector se define como un conjunto en el que sus componentes ocupan posiciones consecutivas de memoria
- Un vector es un conjunto ordenado de pares <índice, valor>

30

### 3. Listas

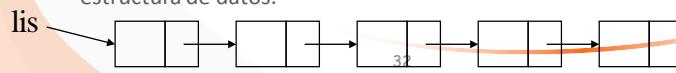
- Una lista es una secuencia de cero o más elementos de un mismo tipo de la forma  $e_1, e_2, \dots, e_n \quad \forall n \geq 0$
- De forma más general:  $e_p, e_{\text{sig}(p)}, \dots, e_{\text{sig}(\text{sig} \dots n) \dots (p)}$   
Al valor  $n$  se le llama *longitud de la lista*. Si  $n = 0$  tenemos una *lista vacía*. A  $e_1$  se le llama primer elemento, y a  $e_n$  último elemento
- Propiedades:
  - Se establece un orden secuencial estricto sobre sus elementos por la *posición* que ocupan dentro de la misma. De esta forma  $e_i$  precede a  $e_{\text{sig}(i)}$  para  $i = 1, 2, \dots, n-1$  y  $e_{\text{sig}(i)}$  sucede a  $e_i$  para  $i = 1, 2, \dots, n-1$ . Por último, el elemento  $e_n$  ocupa la posición  $i$
  - La lista nos permite conocer cualquier elemento de la misma *accediendo a su posición*, algo que no podremos hacer con las pilas y con las colas. Utilizaremos el concepto generalizado de posición, con una ordenación definida sobre la misma, por lo tanto no tiene por qué corresponderse exactamente con los números enteros, como clásicamente se ha interpretado este concepto
- Una *lista ordenada* es un tipo especial de lista en el que se establece una relación de orden definida entre los items de la lista

31

### 3. Listas

#### REPRESENTACIÓN DE LISTAS

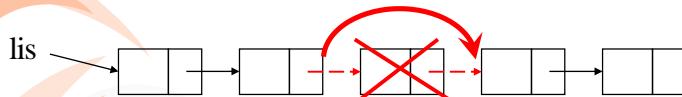
- El TAD *lista* se utiliza para almacenar listas de un número variable de objetos.
- Representación secuencial (internamente un *array*)
  - **A partir de tipos base (“arrays”)**
  - **A partir de tipos definidos por el usuario (“tvector” –herencia o layering –)**
- Representación enlazada (internamente *punteros a nodo*)
  - **A partir de tipos base (“punteros a nodo”)**
  - Cada objeto se almacena en un nodo, que se enlaza con el siguiente.
  - La lista es un puntero al primer nodo, o NULL si está vacía.
  - Un nodo es un contenedor para almacenar información, y tiene dos partes:
    - La información del objeto que se desea guardar.
    - Uno o más punteros para enlazar el nodo con otros nodos y construir la estructura de datos.



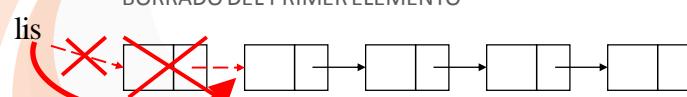
### 3. Listas

#### REPRESENTACIÓN ENLAZADA DE LISTAS (I)

- Operaciones Básicas sobre las listas:
  - Averiguar si la lista está vacía
  - Búsqueda de un elemento
  - Inserción y Borrado de un elemento: hay que distinguir si es al principio, en una posición intermedia o al final de la lista
    - BORRADO DE UN ELEMENTO INTERMEDIO O FINAL



- BORRADO DEL PRIMER ELEMENTO

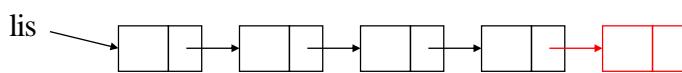


33

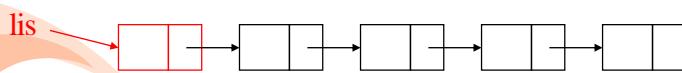
### 3. Listas

#### REPRESENTACIÓN ENLAZADA DE LISTAS (II)

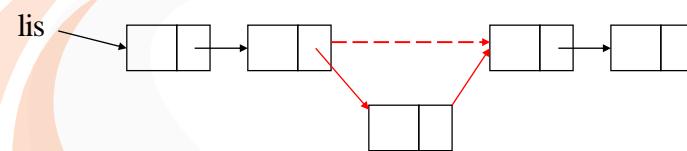
- INSERCIÓN DE UN ELEMENTO AL FINAL



- INSERCIÓN DE UN ELEMENTO AL PRINCIPIO



- INSERCIÓN EN UNA POSICIÓN INTERMEDIA



34

### 3. Listas

#### LISTAS ORDENADAS

- Una lista ordenada es una lista que en todo momento se mantiene ordenada.
- Se consigue insertando siempre los nodos en la posición que les corresponda según el orden definido (los borrados no desordenan la lista).
- Ventajas frente a la lista simple:
  - El tiempo medio de búsqueda se reduce (no es necesario recorrer toda la lista para comprobar que un nodo no está).
  - No es necesario ordenar la lista.

### 3. Listas

#### ESPECIFICACIÓN ALGEBRAICA (I)

```

MODULO LISTA USA BOOL, NATURAL
PARAMETRO TIPO item, posicion
OPERACIONES
  == ( posicion, posicion ) → bool
  error_item() → item
  error_posicion() → posicion
FPARAMETRO
TIPO lista
OPERACIONES
  crear() → lista
  inscabeza( lista, item ) → lista
  esvacia( lista ) → bool
  concatenar( lista, lista ) → lista
  longitud( lista ) → natural
  primera, ultima( lista ) → posicion
  anterior, siguiente( lista, posicion ) → posicion
  insertar( lista, posicion, item ) → lista
  borrar( lista, posicion ) → lista
  obtener( lista, posicion ) → item

```

<sup>36</sup>

### 3. Listas

#### ESPECIFICACIÓN ALGEBRAICA (II)

**VAR** L<sub>1</sub>, L<sub>2</sub>; lista; x: item; p: posición;

#### ECUACIONES

esvacia( crear( ) ) = CIERTO  
 esvacia( inscabeza( L<sub>1</sub>, x ) ) = FALSO

concatenar( crear( ), L<sub>1</sub> ) = L<sub>1</sub>  
 concatenar( L<sub>1</sub>, crear( ) ) = L<sub>1</sub>  
 concatenar( inscabeza( L<sub>1</sub>, x ), L<sub>2</sub> ) = inscabeza( concatenar( L<sub>1</sub>, L<sub>2</sub> ), x )

longitud( crear( ) ) = 0  
 longitud( inscabeza( L<sub>1</sub>, x ) ) = 1 + longitud( L<sub>1</sub> )

primera( crear( ) ) = error\_posicion( ); ultima( crear( ) ) = error\_posicion( )  
**si** esvacia( L<sub>1</sub> ) **entonces**

ultima( inscabeza( L<sub>1</sub>, x ) ) = primera( inscabeza( L<sub>1</sub>, x ) )  
**si no** ultima( inscabeza( L<sub>1</sub>, x ) ) = ultima( L<sub>1</sub> )

anterior( L<sub>1</sub>, primera( L<sub>1</sub> ) ) = error\_posicion( ); siguiente( L<sub>1</sub>, ultima( L<sub>1</sub> ) ) = error\_posicion( )  
**si** p != ultima( L<sub>1</sub> ) **entonces** anterior( L<sub>1</sub>, siguiente( L<sub>1</sub>, p ) ) = p  
 anterior( inscabeza( L<sub>1</sub>, x ), primera( L<sub>1</sub> ) ) = <sup>37.</sup> primera( inscabeza( L<sub>1</sub>, x ) )

### 3. Listas

#### ESPECIFICACIÓN ALGEBRAICA (III)

**si** p != primera( L<sub>1</sub> ) **entonces** siguiente( L<sub>1</sub>, anterior( L<sub>1</sub>, p ) ) = p  
 siguiente( inscabeza( L<sub>1</sub>, x ), primera( inscabeza( L<sub>1</sub>, x ) ) ) = primera( L<sub>1</sub> )

insertar( crear( ), p, x ) = crear( )  
**si** p == primera( inscabeza( L<sub>1</sub>, x ) ) **entonces**

insertar( inscabeza( L<sub>1</sub>, x ), p, y ) = inscabeza( inscabeza( L<sub>1</sub>, y ), x )  
**si no** insertar( inscabeza( L<sub>1</sub>, x ), p, y ) = inscabeza( insertar( L<sub>1</sub>, p, y ), x )

borrar( crear( ), p ) = crear( )  
**si** p == primera( inscabeza( L<sub>1</sub>, x ) ) **entonces**

borrar( inscabeza( L<sub>1</sub>, x ), p ) = L<sub>1</sub>  
**si no** borrar( inscabeza( L<sub>1</sub>, x ), p ) = inscabeza( borrar( L<sub>1</sub>, p ), x )

obtener( crear( ), p ) = error\_item( )  
**si** p == primera( inscabeza( L<sub>1</sub>, x ) ) **entonces**

obtener( inscabeza( L<sub>1</sub>, x ), p ) = x  
**si no** obtener( inscabeza( L<sub>1</sub>, x ), p ) = obtener( L<sub>1</sub>, p )

## 3. Listas

### ENRIQUECIMIENTO DE LAS LISTAS

#### OPERACIONES

sublista( lista, posicion, natural ) → lista

inversa (lista) → lista

**VAR** L: lista; x, y: item; n: natural; p: posicion;

#### ECUACIONES

sublista( L, p, 0 ) = crear( )

sublista( crear( ), p, n ) = crear( )

**si** p == primera( inscabeza( L, x ) ) **entonces**

sublista( inscabeza( L, x ), p, n ) = inscabeza( sublista( L, primera( L ), n - 1 ), x )

**si no** sublista( inscabeza( L, x ), p, n ) = sublista( L, p, n )

inversa( crear( ) ) = crear( )

inversa( inscabeza( crear( ), x ) ) = inscabeza( crear( ), x )

inversa( inscabeza( L, x ) ) = insertar( inversa( L ), ultima( inversa( L ) ), x )

39

## 3. Listas

### REPRESENTACIÓN DE LISTAS (I)

```
class TLista {
    friend ostream&
        operator<<(ostream&, TLista&);
    friend class TPosicion;
public:
    TLista();
    ~TLista();
    void InsCabeza(int);
    TPosicion Primera();
    int& Obtener(TPosicion&);
    void Borrar(TPosicion&);
private:
    TNodo *lis;
};
```

```
class TNodo {
    friend class TLista; friend class TPosicion;
public:
    TNodo(); ~TNodo();
private:
    int dato; TNodo *sig; };

class TPosicion {
    friend class TLista;
public:
    TPosicion(); ~TPosicion();
    bool EsVacia();
    TPosicion Siguiente();
    TPosicion& operator=(TPosicion&);
private:
    TNodo* pos; };
```

### 3. Listas

#### REPRESENTACIÓN DE LISTAS (II)

```
TLista::TLista() { lis = NULL; }

TLista::~TLista() {
    TPosicion p, q;
    q = Primera();
    while(!q.EsVacia()) {
        p = q;
        q = q.Siguiente();
        delete p.pos;
    }
    lis = NULL;
}
```

```
void
TLista::InsCabeza(int i) {
    TNodo* aux = new TNodo;
    aux->dato = i;
    if(lis == NULL) {
        aux->sig = NULL;
        lis = aux;
    } else {
        aux->sig = lis;
        lis = aux;
    }
}
```

41

### 3. Listas

#### REPRESENTACIÓN DE LISTAS (III)

```
TPosicion
TLista::Primera() {
    TPosicion p; p.pos = lis;
    return p;}
int&
TLista::Obtener(TPosicion& p) {
    return p.pos->dato;}
ostream&
operator<<(ostream& os, TLista& l) {
    TPosicion p;
    p = l.Primera();
    while(!p.EsVacia()) {
        os << l.Obtener(p) << ' ';
        p = p.Siguiente();}
    return os; }
```

```
TNodo::TNodo() {
    dato = 0; sig = NULL; }

TNodo::~TNodo() {
    dato = 0; sig = NULL; }

TPosicion::TPosicion() { pos = NULL; }

TPosicion::~TPosicion() {
    pos = NULL; }

bool
TPosicion::EsVacia() {
    return pos == NULL; }
```

42

### 3. Listas

#### REPRESENTACIÓN DE LISTAS (IV)

```

TPosicion
TPosicion::Siguiente() {
    TPosicion p;
    p.pos = pos->sig;
    return p;
// ¿si pos es NULL?
}

TPosicion&
TPosicion::operator=(TPosicion& p) {
    pos = p.pos;
    return *this;
}

```

```

int
main(void)
{
    TLista l;
    l.InsCabeza(1); l.InsCabeza(3);
    l.InsCabeza(5); l.InsCabeza(7);
    cout << l << endl;
    TPosicion p;
    p = l.Primera();
    cout << "Primer elemento: "
        << l.Obtener(p) << endl;
    p = p.Siguiente();
    cout << "Segundo elemento: "
        << l.Obtener(p) << endl;
}

```

43

### Vectores y Listas

#### Aplicaciones reales

- Clasificación de los equipos de la liga de fútbol.
- Gestión de la lista de espera de un hospital para intervenciones quirúrgicas.



Clasificación 1ª división

EQUIPO	PTOS	PJ	PG	PE	PP	GF	GC
1 Barcelona	50	19	16	2	1	53	12
2 At. Madrid	50	19	16	2	1	47	11
3 Real Madrid	47	19	15	2	2	53	21
4 Ath.Bilbao	36	19	11	3	5	32	24
5 Villarreal	34	19	10	4	5	37	21
6 Real Sociedad	32	19	9	5	5	36	28
7 Sevilla FC	30	19	8	5	5	36	30
8 Valencia	23	19	7	2	10	26	31
9 Granada	23	19	7	2	10	19	25
10 Levante	23	19	6	5	8	18	27
11 Getafe	23	19	7	2	10	20	31
12 Espanyol	22	19	6	4	9	22	25
13 Osasuna	21	19	6	3	10	17	29
14 Málaga	20	19	5	5	9	19	24
15 Celta de Vigo	19	19	5	4	10	23	31
16 Almería	19	19	5	4	10	21	38
17 Elche	18	19	4	6	9	17	28
18 Valladolid	16	19	3	7	9	21	33
19 Rayo	16	19	5	1	13	19	45
20 Real Betis	11	19	2	5	12	16	38

44

### 3. Listas

#### EJERCICIOS *borraultimo*

- Completa las ecuaciones que aparecen a continuación y que expresan el comportamiento de las operaciones de *borraultimo* (borra el último elemento de la lista) en una lista de acceso por posición:

La sintaxis de la operación es la siguiente:

*borraultimo(lista)* → lista

**borraultimo:lista → lista**

*borraultimo(crear()) = .....*

**si** *esvacia(l1)* **entonces** *borraultimo(inscabeza(l1,x)) = .....*

**si no** *borraultimo(inscabeza(l1,x)) = .....*

Donde:  $x \in \text{elemento}$

$l1 \in \text{lista}$

45

### 3. Listas

#### EJERCICIOS *quita\_pares*

- Definir la sintaxis y la semántica de la operación *quita\_pares* que actúa sobre una lista y devuelve la lista original en la que se han eliminado los elementos que ocupan las posiciones pares

46

### 3. Listas

#### EJERCICIOS operación X

- Explicar qué hace la operación X, cuya sintaxis y semántica aparecen a continuación:

$X(\text{lista}) \rightarrow \text{lista}$

$X : \text{lista} \rightarrow \text{lista}$

$X(\text{crear}()) \rightarrow \text{crear}()$

$X(\text{inscabeza}(l, i)) \iff$

**si** ( $\text{longitud}(l) == 0$ ) **entonces**  $\text{crear}()$

**si no**  $\text{inscabeza}(X(l), i)$

Donde:  $l \in \text{lista}$ ,  $i \in \text{ítem}$

- Simplificar la siguiente expresión: ( $IC = \text{inscabeza}$ )

$X(IC(IC(IC(IC(\text{crear}(), a), b), c), d))$

47

### 3. Listas

#### Preguntas de tipo test: Verdadero vs. Falso

- La operación BorrarItem tiene la siguiente sintaxis y semántica:

BorrarItem: LISTA, ITEM  $\rightarrow$  LISTA

$\text{BorrarItem}(\text{Crear}, i) = \text{Crear}$

$\text{BorrarItem}(IC(L1, j), i) = \text{si } (i == j) \text{ entonces } L1$

sino  $IC(\text{BorrarItem}(L1, i), j)$

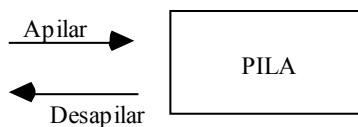
Esta operación borra todas las ocurrencias del ítem que se encuentra en la lista

- La complejidad temporal de obtener un elemento en un vector ordenado mediante búsqueda binaria o en una lista ordenada es la misma

48

## 4. Pilas

- Una pila es una lista en la que todas las inserciones y borrados se realizan en un único extremo, llamado *tope* o *cima*. Sabemos por tanto que el último elemento insertado en la pila será el primero en ser borrado de la misma, de ahí que también se les llame listas “LIFO” (Last In, First Out). También podemos conocer cuál es el elemento que se encuentra en la *cima*



49

## 4. Pilas

### ESPECIFICACIÓN ALGEBRAICA

**MODULO PILA USA BOOL**

**PARAMETRO**

TIPO item

OPERACIONES

error( ) → item

**FPARAMETRO**

**TIPO pila**

**OPERACIONES**

crear( ) → pila

apilar( pila, item ) → pila

desapilar( pila ) → pila

cima( pila ) → item

esvacia( pila ) → bool

**VAR** p: pila, e: item;

**ECUACIONES**

desapilar( crear( ) ) = crear( )

desapilar( apilar( p, e ) ) = p

cima( crear( ) ) = error( )

cima( apilar( p, e ) ) = e

esvacia( crear( ) ) = CIERTO

esvacia( apilar( p, e ) ) = FALSO

**FMODULO**

50

## 4. Pilas

### REPRESENTACIÓN SECUENCIAL DE PILAS (I)

- Representación secuencial (internamente un *array*)
  - A partir de tipos base (“arrays”)
  - A partir de tipos definidos por el usuario (“tvector” –herencia o *layering* –)
- Tipos de algoritmos
  - Realizando las inserciones por la primera componente. Ineficiente
  - Utilizando un cursor que indique la posición actual del primer elemento de la pila
- Ventajas y desventajas
  - Desventaja: tamaño máximo de la pila
  - Ventaja: sencillez de implementación

51

## 4. Pilas

### REPRESENTACIÓN SECUENCIAL DE PILAS (II)

```
const kMax = 10;
class TPila {
public:
    TPila(); TPila( TPila & ); ~TPila(); TPila& operator=( TPila & );
    TItem& Cima();
    void Apilar( TItem& );
    ...
private:
    TItem fpila[ kMax ]; //tamaño fijo
    // TItem *fpila; tamaño dinámico
    int ftope;
};
TPila::TPila() {
    fpila = new TItem[ 10 ]; //sólo si el vector es dinámico
    ftope = 0; }
void
TPila::Desapilar() {
    ftope --; }
```

52

## 4. Pilas

### REPRESENTACIÓN ENLAZADA DE PILAS (I)

- Representación enlazada (internamente *punteros a nodo*)
  - A partir de tipos base (“punteros a nodo”)
  - A partir de tipos definidos por el usuario (“tlista” –herencia o *layering*–)
- Ventajas
  - Ventaja: no hay definido un tamaño para la pila

53

## 4. Pilas

### REPRESENTACIÓN ENLAZADA DE PILAS (II)

```
class TPila {
public:
    TPila( ); TPila( TPila & ); ~TPila( ); TPila& operator=( TPila & );
    TItem& Cima( );
    void Apilar( TItem& );
    ...
private:
    struct TNodo {
        TItem dato;
        TNodo *sig; };
    TNodo *fp;
};

TPila::TPila() { fp = NULL; }

void
TPila::Desapilar( ) {
    TNodo *aux;
    aux = fp;
    fp = fp -> sig;
    delete aux; }
```

54

## 4. Pilas

### REPRESENTACIÓN ENLAZADA DE PILAS (III)

```
//HERENCIA PRIVADA
class TPila: private TLista {
public:
    TPila(); TPila( TPila & ); ~TPila();
    void Apilar( TItem& );
    void Desapilar();
    ...
};

TPila::TPila(): TLista() { };
TPila::TPila( TPila &p ): TLista( p ) { };
~TPila() { }
void
TPila::Apilar( TItem &a ) { InsCabeza( a ); }
void
TPila::Desapilar() { Borrar( Primera() ); }
```

```
//LAYERING O COMPOSICIÓN
class TPila {
public:
    TPila(); TPila( TPila & ); ~TPila();
    void Apilar( TItem& ); void Desapilar();
    ...
private: TLista L;
};

TPila::TPila(): L() { };
TPila::TPila( TPila &p ): L( p.L ) { };
~TPila() { }
void
TPila::Apilar( TItem &a ) { L.InsCabeza( a ); }
void
TPila::Desapilar() { L.Borrar( L.Primera() ); }55
```

## 4. Pilas

### EJERCICIOS

- Dar la sintaxis y la semántica de la operación **base**, que actúa sobre una pila y devuelve la base de la pila (el primer elemento que se ha apilado)

## 5. Colas

- Una cola es otro tipo especial de lista en el cual los elementos se insertan por un extremo (*fondo*) y se suprimen por el otro (*tope*). Las colas se conocen también como listas “FIFO” (First In First Out). Las operaciones definidas sobre una cola son similares a las definidas para las pilas con la salvedad del modo en el cual se extraen los elementos



57

## 5. Colas

### ESPECIFICACIÓN ALGEBRAICA

MODULO COLA USA BOOL  
 PARAMETRO  
 TIPO item  
 OPERACIONES  
 $\text{error}() \rightarrow \text{item}$   
 FPARAMETRO  
 TIPO cola  
 OPERACIONES  
 $\text{crear}() \rightarrow \text{cola}$   
 $\text{encolar}(\text{cola}, \text{item}) \rightarrow \text{cola}$   
 $\text{desencolar}(\text{cola}) \rightarrow \text{cola}$   
 $\text{cabeza}(\text{cola}) \rightarrow \text{item}$   
 $\text{esvacia}(\text{cola}) \rightarrow \text{bool}$

VAR c: cola, x: item;  
**ECUACIONES**  
 $\text{desencolar}(\text{crear}()) = \text{crear}()$   
**si**  $\text{esvacia}(c)$  **entonces**  
 $\text{desencolar}(\text{encolar}(c, x)) = \text{crear}()$   
**si no**  $\text{desencolar}(\text{encolar}(c, x)) =$   
 $\text{encolar}(\text{desencolar}(c), x)$   
 $\text{cabeza}(\text{crear}()) = \text{error}()$   
**si**  $\text{esvacia}(c)$  **entonces**  
 $\text{cabeza}(\text{encolar}(c, x)) = x$   
**si no**  $\text{cabeza}(\text{encolar}(c, x)) = \text{cabeza}(c)$   
 $\text{esvacia}(\text{crear}()) = \text{CIERTO}$   
 $\text{esvacia}(\text{encolar}(c, x)) = \text{FALSO}$   
**FMODULO**

58

## 5. Colas

### ENRIQUECIMIENTO DE LAS COLAS

#### OPERACIONES

concatena( cola, cola ) → cola

**VAR** c, q: cola; x: item;

#### ECUACIONES

concatena( c, crear( ) ) = c

concatena( crear( ), c ) = c

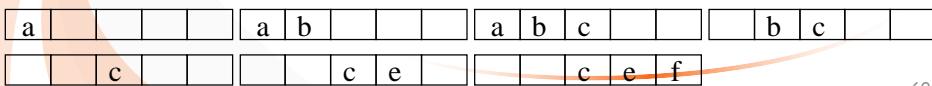
concatena( c, encolar( q, x ) ) = encolar( concatena( c, q ), x )

59

## 5. Colas

### REPRESENTACIÓN SECUENCIAL DE COLAS (I)

- Representación secuencial (internamente un *array*)
  - A partir de tipos base (“arrays”)
  - A partir de tipos definidos por el usuario (“tvector” –herencia o layering–)
- Tipos de algoritmos
  - Utilizando un array (*fv*) para almacenar los elementos y dos enteros (*tope* y *fondo*) para indicar la posición de ambos extremos
    - Inicializar: *tope* = 0; *fondo* = -1;
    - Condición de cola vacía: *fondo* < *tope*
    - Inserción: *fondo*++; *fv[fondo]* = *x*;
    - Borrado: *tope*++;



60

## 5. Colas

### REPRESENTACIÓN SECUENCIAL DE COLAS (II)

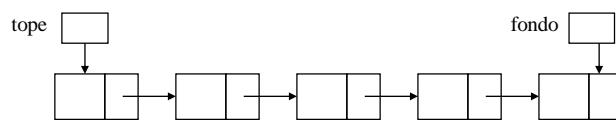
- Problema:
  - Hay huecos pero no puedo insertar
- Soluciones:
  - Cada vez que se borra un elemento, el resto se desplaza una posición a la izquierda para que *tope* siempre esté en la primera posición. ¿Qué problemas presenta esta solución? aumentamos la complejidad de la operación desencolar
  - Colas circulares. *Array* como un círculo en el que la primera posición sigue a la última. Condición de cola vacía  $\text{tope} == \text{fondo}$
- Ventajas y desventajas
  - Desventaja: tamaño máximo de la cola
  - Ventaja: sencillez de implementación

61

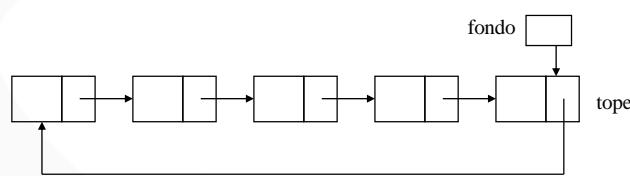
## 5. Colas

### REPRESENTACIÓN ENLAZADA DE COLAS (I)

- Representación enlazada (internamente *punteros a nodo*)
  - A partir de tipos base (“punteros a nodo”)



- Colas circulares enlazadas, en las que sólo se necesita un puntero. El siguiente elemento apuntado por *fondo* es el primero a desencolar



62

## 5. Colas

### REPRESENTACIÓN ENLAZADA DE COLAS (II)

- A partir de tipos definidos por el usuario (“tlista” –herencia o layering–)
- Ventajas
  - Ventaja: no hay definido un tamaño para la cola

63

## Pilas y Colas

### Aplicaciones reales

- Gestión del uso de recursos compartidos: colas de impresión, colas de procesos pendientes de ejecución, colas de mensajes, etc.
- Los editores de texto proporcionan normalmente un botón deshacer que cancela las operaciones de edición recientes y restablece el estado anterior del documento (almacenamiento en una pila).
- Los Navegadores en Internet almacenan en una pila las direcciones de los sitios más recientemente visitados.



64

## EJERCICIOS

- Dada la clase *TVector* que almacena un vector dinámico de enteros y un entero que contiene la dimensión del vector, definid en C++:
  - La clase *TVector*
  - Constructor por defecto (dimensión 10 y componentes a -1)

65

## EJERCICIOS

- Dada la clase *TPila* definid en C++ el método *Apilar*

66

## Pilas y colas

Preguntas de tipo test: Verdadero vs. Falso

- La semántica de la operación *cima* del tipo *pila* vista en clase es la siguiente:  
**VAR** p: pila, e: item;  
cima( crear( ) ) = error( )  
cima( apilar( p, e ) ) = cima( p )
- Es posible obtener una representación enlazada de una cola utilizando un único puntero que apuntará al fondo de la cola

## TEMA 2

# La eficiencia de los algoritmos

PROGRAMACIÓN Y ESTRUCTURAS DE DATOS

## La eficiencia de los algoritmos

- 1. Noción de complejidad
  - Complejidad temporal, tamaño del problema y paso
- 2. Cotas de complejidad
  - Cota superior, inferior y promedio
- 3. Notación asintótica
  - $O$ ,  $\Omega$ ,  $\Theta$
- 4. Obtención de cotas de complejidad

# 1. Noción de complejidad

## DEFINICIÓN

- Cálculo de complejidad: determinación de dos parámetros o funciones de coste:
  - Complejidad espacial : Cantidad de recursos espaciales ( de almacén) que un algoritmo consume o necesita para su ejecución
  - Complejidad temporal : Cantidad de tiempo que un algoritmo necesita para su ejecución
- Posibilidad de hacer
  - Valoraciones
    - el algoritmo es: “bueno”, “el mejor”, “prohibitivo”
  - Comparaciones
    - el algoritmo A es mejor que el B

3

# 1. Noción de complejidad

## COMPLEJIDAD TEMPORAL

- Factores de complejidad temporal:
  - Externos
    - La máquina en la que se va a ejecutar
    - El compilador: variables y modelo de memoria
    - La experiencia del programador
  - Internos
    - El número de instrucciones asociadas al algoritmo
- Complejidad temporal :  $Tiempo(A) = C + f(T)$ 
  - $C$  es la contribución de los factores externos (constante)
  - $f(T)$  es una función que depende de  $T$  (talla o tamaño del problema)

4

# 1. Noción de complejidad

## COMPLEJIDAD TEMPORAL

- **Talla** o tamaño de un problema:
  - Valor o conjunto de valores asociados a la **entrada** del problema que representa una medida de su tamaño respecto de otras entradas posibles
- **Paso** de programa:
  - Secuencia de operaciones con contenido semántico cuyo coste es independiente de la Talla del problema
  - Unidad de medida de la complejidad de un algoritmo
- **Expresión de la complejidad temporal:**
  - Función que expresa el número de pasos de programa que un algoritmo necesita ejecutar para cualquier entrada posible (para cualquier talla posible)
  - No se tienen en cuenta los factores externos

5

# 1. Noción de complejidad

## COMPLEJIDAD TEMPORAL. Ejemplos

```
int ejemplo1 (int n)
{
    n+ = n;
    return n;
}
```

$$f(\text{ejemplo1}) = 1 \text{ pasos}$$

Estas instrucciones se ejecutan siempre el mismo número de veces.  
Sea cual sea el valor de n. (1 paso)

```
int ejemplo2 (int n)
{
    int i;
    for (i=0; i ≤ 2000; i++)
        n+ = n;
    return n;
}
```

$$f(\text{ejemplo2}) = 1 \text{ pasos}$$

6

# 1. Noción de complejidad

COMPLEJIDAD TEMPORAL. Ejemplos

```
int ejemplo3 (int n)
{
    int i, j;
    j = 2;
    for (i=0; i ≤ 2000; i++)
        j=j*j;

    for (i=0; i ≤ n; i++)
    {
        j = j + j;
        j = j - 2;
    }

    return j;
}
```

$$f(\text{ejemplo3}) = 1 + 1 \cdot (n + 1) \text{ pasos}$$

1

Estas instrucciones se ejecutan siempre el mismo número de veces. Sea cual sea el valor de n. (1 paso)

2

El bucle for se repetirá n+1 veces.

3

Estas dos instrucciones dentro del bucle for siempre se van a repetir el mismo número de veces, por lo que se consideran 1 paso

7

# 1. Noción de complejidad

COMPLEJIDAD TEMPORAL. Ejemplos

```
int ejemplo4 (int n)
{
    int i, j,k;
    k = 1;

    for (i=0; i ≤ n; i++)
        for (j=1; j ≤ n; j++)
            k = k + k;

    return k;
}
```

$$f(\text{ejemplo4}) = 1 + 1 \cdot n \cdot (n + 1) \text{ pasos}$$

1

Estas instrucciones se ejecutan siempre el mismo número de veces. Sea cual sea el valor de n. (1 paso)

2

El bucle for i=0; i<=n; i++) se repite n+1 veces

3

El bucle for (j=1; j<=n; j++) se repite n veces

4

La instrucción k=k+k es 1 paso

```
int ejemplo5 (int n)
{
    int i, j,k;
    k = 1;
    for (i=0; i ≤ n; i++)
        for (j=i; j ≤ n; j++)
            k = k + k;

    return k;
}
```

$$f(\text{ejemplo5}) = 1 + \sum_{i=0..n} (\sum_{j=i..n} 1) \text{ pasos}$$

## 1. Noción de complejidad

COMPLEJIDAD TEMPORAL. Ejemplos

```
int ejemplo4 (int n)
{ int i, j, k;
  k = 1;
  for (i=0; i <= n; i++)
    for (j=1; j <= n; j++)
      k = k + k;
  return k;
}
```

$$f(\text{ejemplo4}) = 1 + 1 \cdot n \cdot (n + 1) \text{ pasos}$$

1

Estas instrucciones se ejecutan siempre el mismo número de veces. Sea cual sea el valor de n. (1 paso)

```
int ejemplo5 (int n)
{ int i, j, k;
  k = 1;
  for (i=0; i <= n; i++)
    for (j=i; j <= n; j++)
      k = k + k;
  return k;
}
```

$$f(\text{ejemplo5}) = 1 + \sum_{i=0..n} (\sum_{j=i..n} 1) \text{ pasos}$$

2

El bucle for  $i=0; i \leq n; i++$  se repite  $n+1$  veces

3

El bucle for  $j=i; j \leq n; j++$  se repite  $n-i+1$  veces

4

La instrucción  $k=k+k$  es 1 paso

## 1. Noción de complejidad

COMPLEJIDAD TEMPORAL. Ejemplos

```
int ejemplo5 (int n)
{ int i, j, k;
  k = 1;

  for (i=0; i <= n; i++)
    for (j=i; j <= n; j++)
      k = k + k;

  return k;
}
```

$$f(\text{ejemplo5}) = 1 + \sum_{i=0..n} (\sum_{j=i..n} 1) \text{ pasos}$$

1

3

2

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

$$3 \sum_{i=0}^n 1 \cdot (n - i + 1) =$$

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

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

Resolución de sumatorios:

$$\sum_{i=m}^n C = C \cdot (n - m + 1)$$

$$\sum_{i=1}^n i = \frac{(a_1 + a_n)n}{2} \text{ (S.P.A)}$$

S.P.A

$a_1=n+1$
$a_n=1$
$n^{\circ} \text{ términos}=n+1$

10

# 1. Noción de complejidad

## COMPLEJIDAD TEMPORAL. Ejercicios

```
for(i = sum = 0; i < n; i++) sum += a[i];
```

```
for(i = 0; i < n; i++) {
    for(j = 1, sum = a[0]; j <= i; j++) sum += a[j];
    cout << "La suma del subarray " << i << " es " << sum << endl; }
```

```
for(i = 4; i < n; i++) {
    for(j = i-3, sum = a[i-4]; j <= i; j++) sum += a[j];
    cout << "La suma del subarray " << i-4 << " es " << sum << endl; }
```

```
for(i = 0, length = 1; i < n-1; i++) {
    for(i1 = i2 = k = i; k < n-1 && a[k] < a[k+1]; k++, i2++);
    if(length < i2 - i1 + 1) length = i2 - i1 + 1; }
```

11

# 1. Noción de complejidad

## COMPLEJIDAD TEMPORAL. Solución (I)

for( $i = \text{sum} = 0; i < n; i++$ )  $\text{sum} += a[i]$ ;

$$\sum_{i=0..n-1} 1 \text{ pasos} = n \cdot 1 \text{ pasos} = O(n)$$

$$\sum_{i=0}^{n-1} 1 = 1 \cdot (n - 1 - 0 + 1) = n$$

Límite superior  
Límite inferior

```
for(i = 0; i < n; i++) {
    for(j = 1, sum = a[0]; j <= i; j++) sum += a[j];
    cout << "La suma del subarray " << i << " es " << sum << endl; }
```

El bucle exterior se repite  $n$  veces. El bucle interior se repite  $i$  veces, con  $i$  desde 1 hasta  $n-1$ . Por tanto:

$$\text{Nº pasos: } \sum_{i=0..n-1} (1[\text{del cout}] + \sum_{j=1..i} (1)) = \sum_{i=0..n-1} (1 + i) = \frac{(1+n) * n}{2}$$

Límite superior:  $i$   
Límite inferior: 1  
Sumatorio=Límite superior – Límite inferior +1 =  $i-1+1 = i$

Complejidad:  $O(n^2)$

12

# 1. Noción de complejidad

## CONCLUSIONES

- Sólo nos ocuparemos de la complejidad temporal
- Normalmente son objetivos contrapuestos  
(complejidad temporal <--> complejidad espacial)
- Cálculo de la complejidad temporal:
  - *a priori*: contando pasos
  - *a posteriori*: generando instancias para distintos valores y cronometrando el tiempo
- Se trata de obtener la función. Las unidades de medida (paso, sg, msg, ...) no son relevantes (todo se traduce a un cambio de escala)
- El nº de pasos que se ejecutan siempre es función del tamaño (o talla) del problema

13

# 2. Cotas de complejidad

## INTRODUCCIÓN

- Dado un vector X de  $n$  números naturales y dado un número natural  $z$ :
  - encontrar el índice  $i : X_i = z$
  - Calcular el número de pasos que realiza

El número de veces que se ejecuta el bucle “mientras” dependerá del tamaño del vector y de la distribución interna de los elementos

1 paso

```

funcion BUSCAR (var X:vector[N]; z: N): devuelve N
var i:natural  fvar;
comienzo
  i:=1;
  mientras (i ≤ |X|) ∧ (Xi≠z) hacer
    i:=i+1;
  fmientras
  si i= |X|+1 entonces devuelve 0 (*No encontrado*)
  si_no devuelve i
fin

```

14

## 2. Cotas de complejidad

### EL PROBLEMA

- No podemos contar el número de pasos porque depende:
  - del **tamaño (TALLA)** del problema  $|X|$
  - de la **instancia** del problema que se pretende resolver (posible valor que puedan tomar las variables de entrada)
- Ejemplo:

Vector	Elemento	Nº PASOS
X	z	
( 0, 1 )	1	
( 1, 2, 3 )	1	
( 2 )	3	
( 1, 0, 2, 4 )	3	
( 1, 0, 2, 4 )	0	
( 1, 0, 2, 4 )	1	

15

## 2. Cotas de complejidad

### LA SOLUCIÓN: cotas de complejidad

- Cuando aparecen diferentes casos para una misma talla genérica  $n$ , se introducen las cotas de complejidad:
  - **Caso peor:** cota superior del algoritmo  $\rightarrow C_s(n)$
  - **Caso mejor:** cota inferior del algoritmo  $\rightarrow C_i(n)$
  - Término medio: cota promedio  $\rightarrow C_m(n)$
- Todas son funciones del tamaño del problema ( $n$ )
- La cota promedio es difícil de evaluar *a priori*
  - Es necesario conocer la distribución de la probabilidad de entrada
  - No es la media de la inferior y de la superior (ni están todas ni tienen la misma proporción)

16

## 2. Cotas de complejidad

EJERCICIO: cotas superior e inferior

```

funcion BUSCAR (var X:vector[N]; z: N): devuelve N
var i:natural fvar;
comienzo
    i:=1;
    mientras (i ≤ |X|) ∧ (Xi≠z) hacer
        i:=i+1;
    fmientras
    si i = |X|+1 entonces devuelve 0 (*No encontrado*)
    si no devuelve i
fin

```

- Talla del problema: nº de elementos de X:  $n$
- ¿Existe caso mejor y caso peor?
  - Caso mejor: el elemento está el primero:  $X_1=z \rightarrow c_i(n) = 1$
  - Caso peor: el elemento no está:  $\forall i 1 \leq i \leq |X|, X_i \neq z \rightarrow c_s(n) = n+1$

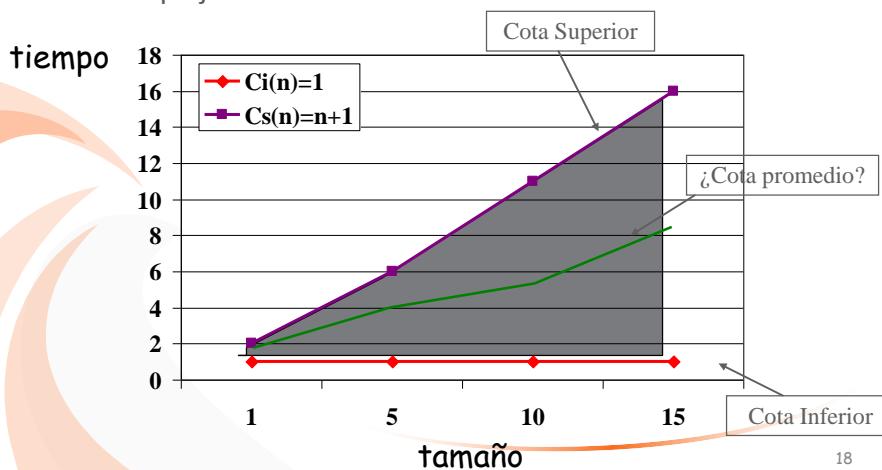
$$1 + \sum_{i=1}^n 1 = 1 + (n - 1 + 1) = n + 1$$

17

## 2. Cotas de complejidad

EJERCICIO: cotas superior e inferior

- Complejidad función “BUSCAR”



18

## 2. Cotas de complejidad

### CONCLUSIONES

- La **cota promedio** no la calcularemos. Sólo se hablará de complejidad por término medio cuando la cota superior y la inferior coinciden
- El estudio de la complejidad se hace para **tamaños grandes** del problema por varios motivos:
  - Los resultados para tamaños pequeños o no son fiables o proporcionan poca información sobre el algoritmo
  - Es lógico invertir tiempo en el desarrollo de un buen algoritmo sólo si se prevé que éste realizará un gran volumen de operaciones
- A la complejidad que resulta de tamaños grandes de problema se le denomina **complejidad asintótica** y la notación utilizada es la notación asintótica

19

## 3. Notación asintótica

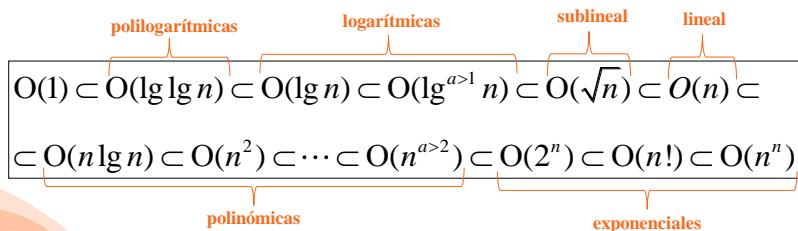
### INTRODUCCIÓN

- Notación matemática utilizada para representar la complejidad espacial y temporal cuando  $n \rightarrow \infty$
- Se definen tres tipos de notación:
  - Notación  $O$  (big-omicron)  $\Rightarrow$  caso peor
  - Notación  $\Omega$  (omega)  $\Rightarrow$  caso mejor
  - Notación  $\Theta$  (big-theta)  $\Rightarrow$  caso promedio

20

### 3. Notación asintótica

Teorema de la escala de complejidad



□  $f(n) + g(n) + t(n) \in O(\max(f(n), g(n), t(n)))$

□ Ejemplos:

- $n + 1$  pertenece a  $O(n)$
- $n^2 + \log n$  pertenece a  $O(n^2)$
- $n^3 + 2^n + n \log n$  pertenece a  $O(2^n)$

□ Válido para Notación  $\Omega$  y Notación  $\Theta$

21

### 3. Notación asintótica

NOTACIÓN O: escala de complejidad

Complejidad	$n = 32$	$n = 64$
$n^3$	3 seg.	26 seg.
$2^n$	5 días	58·10 <sup>6</sup> años

- Tiempos de respuesta para dos valores de la talla y complejidades  $n^3$  y  $2^n$ .  
(paso = 0'1 mseg.)

– Queda clara la necesidad del cálculo de complejidad

```
función POT_2 (n: natural): natural
  opción
    n = 1: devuelve 2
    n > 1: devuelve 2 * POT_2(n-1)
  fopción
ffunción
```

Coste lineal  
1 seg.

```
función POT_2 (n: natural): natural
  opción
    n = 1: devuelve 2
    n > 1: devuelve POT_2(n-1)+POT_2(n-1)
  fopción
ffunción
```

Coste exponencial  
miles de años

22

## 4. Obtención de cotas de complejidad

### INTRODUCCIÓN

- Etapas para obtener las cotas de complejidad:
  1. Determinación de la **TALLA** o tamaño (de la instancia) del problema
  2. Determinación del **CASO MEJOR Y PEOR**: instancias para las que el algoritmo tarda más o menos (**INSTANCIAS de la TALLA**)
    - No siempre existe mejor y peor caso ya que existen algoritmos que se comportan de igual forma para cualquier instancia del mismo tamaño
  3. Obtención de las **COTAS** para cada caso. Métodos:
    - **cuenta de pasos**
    - relaciones de recurrencia (**funciones recursivas**)

23

## 4. Obtención de cotas de complejidad

### INTRODUCCIÓN

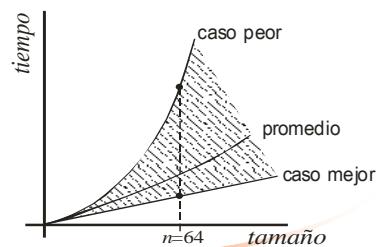
```
función FACTORIAL (n:natural): natural
```

- La talla es  $n$  y no existe caso mejor ni peor

```
función BUSCA (v: vector[natural]; x:natural)
```

- La talla es  $n=|v|$
- **caso mejor**: instancias donde  $x$  está en  $v[1]$
- **caso peor**: instancias donde  $x$  no está en  $v$

- Se trata de delimitar con una región el tiempo que tarda un algoritmo en ejecutarse



24

## 4. Obtención de cotas de complejidad

Ejemplos: MÁXIMO de vector

Cálculo del máximo de un vector

```

funcion MÁXIMO (var v : vector[n]; n:entero) : entero
var i, max : entero fvar
comienzo
    max:=v[1]
    para i:=2 hasta n hacer
        si v[i]>max entonces max:=v[i] fsi
    fpara
    devuelve max
fin

```

1 paso

- determinar la **TALLA** del problema:  $n = \text{tamaño del vector}$

- mejor caso**  $c_i = 1 + \sum_{i=2}^n 1 = 1 + (n - 2 + 1) = n \in \Omega(n)$

La condición  $v[i] > max$  NUNCA se cumple

- peor caso**  $c_s = 1 + \sum_{i=2}^n 1 = 1 + (n - 2 + 1) = n \in O(n)$

La condición  $v[i] > max$  SIEMPRE se cumple

25

## 4. Obtención de cotas de complejidad

Ejemplos : BUSQUEDA BINARIA en vector

Búsqueda de un elemento en un vector ordenado (Búsqueda binaria)

```

funcion BUSCA (var v:vector[N]; x,pri,ult: natural): natural
var m: natural fvar
comienzo
    repetir
        m:= (pri+ult)/2
        si v[m]>x entonces ult:= m-1
                sino pri:= m+1
        fsi
    hasta (pri>ult) v v[m]=x
        si v[m]=x entonces devuelve m
                sino devuelve 0
        fsi
fin

```

26

## 4. Obtención de cotas de complejidad

Ejemplos : BUSQUEDA BINARIA en vector

- Determinar la **TALLA** del problema:  $n=\text{tamaño del vector}$
- **Mejor caso:**  $x$  está en la mitad del vector
- **Peor caso:**  $x$  no está en el vector
- **Complejidades**
  - **mejor caso:**  $1+1=2 \in \Omega(1)$
  - **peor caso**
    - +  **$1+u \cdot 1$** , donde  $u$  es el nº de veces que se ejecuta el bucle
      - 1<sup>a</sup> iteración: Talla = **ult-pri+1** =  $n$
      - 2<sup>a</sup> iteración: Talla = **ult-pri+1** =  $n/2$
      - 3<sup>a</sup> iteración: Talla = **ult-pri+1** =  $n/4$
      - .....
      - $k$ -ésima iteración: Talla = **ult-pri+1** =  $n/2^{(k-1)}$
      - .....
      - última iteración: Talla = **ult-pri+1** =  $n/2^{(u-1)} = 1$
  - Es decir, en la última iteración sólo nos queda 1 elemento  
Despejando  $u$ :  

$$n=2^{u-1} \rightarrow \log_2 n = u - 1 \rightarrow \log_2 n + 1 = u$$

$$1 + u \cdot 1 = 1 + (\log_2 n + 1) \in \Theta(\log_2 n)$$

27

## 4. Obtención de cotas de complejidad

Algoritmos de ordenación

- Directos
  - Inserción directa
  - Inserción binaria
  - Selección directa
  - Intercambio directo (burbuja)

28

## 4. Obtención de cotas de complejidad

### Algoritmos de ordenación

#### INSERCIÓN DIRECTA

- Divide lógicamente el vector en dos partes: origen y destino
- Comienzo:
  - *destino* tiene el primer elemento del vector
  - *origen* tiene los  $n-1$  elementos restantes
- Se va tomando el primer elemento de *origen* y se inserta en *destino* en el lugar adecuado, de forma que *destino* siempre está ordenado
- El algoritmo finaliza cuando no quedan elementos en *origen*
- Características
  - caso mejor: vector ordenado ascendente
  - caso peor: vector ordenado inversamente

29

## 4. Obtención de cotas de complejidad

### Algoritmos de ordenación

#### INSERCIÓN DIRECTA

```

funcion INSERCIÓN_DIRECTA (var a:vector[natural]; n: natural)
var i,j: entero; x:natural fvar
comienzo
  para i:=2 hasta n hacer
    x:=a[i]; j:=i-1
    mientras (j>0) ^ (a[j]>x) hacer
      a[j+1]:=a[j]
      j:=j-1
    fmientras
    a[j+1]:=x
  fpara
fin

```

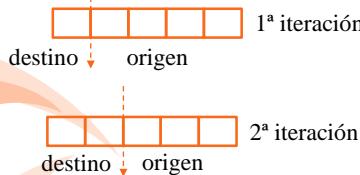
30

## 4. Obtención de cotas de complejidad

### Algoritmos de ordenación

#### INSERCIÓN DIRECTA

- Se divide el vector en dos partes: origen y destino



- En cada iteración se coloca el elemento  $a[i]$  del subvector "origen" en su posición correcta del subvector "destino"  $a[1..i-1]$

31

## 4. Obtención de cotas de complejidad

### Algoritmos de ordenación

#### INSERCIÓN BINARIA

EL BUCLE `mientras` BUSCA EL SITIO DEL PRIMER ELEMENTO DE ORIGEN EN DESTINO.  
EL COSTE DE ESTE BUCLE ES SIEMPRE  $\log_2(i)$  (YA QUE LO HACE CON UNA BÚSQUEDA BINARIA).  
Y LA SUMA PARA  $i=2..n$  DE  $\log_2(i)$  ES POR APROXIMACIÓN  $n \cdot \log_2(n)$

```

funcion INSERCIÓN_BINARIA (var a:vector[natural]; n: natural)
var i, j, iz, de, m: entero; x:natural fvar
comienzo
    para i:=2 hasta n hacer
        x:=a[i]; iz:=1; de:=i-1
        mientras (iz≤de) hacer
            m:= (iz+de)/2
            si a[m]>x entonces de:= m-1
            sino iz:=m+1
        fsi
    fmientras
    para j:=i-1 hasta iz hacer (*decreciente*)
        a[j+1]:=a[j]
    fpara
    a[iz]:=x
fpara
fin

```

32

## 4. Obtención de cotas de complejidad

### Algoritmos de ordenación

#### INSERCIÓN BINARIA

- Es una mejora del algoritmo de inserción directa
- Cambia en un punto:
  - Cuando busca la posición donde debe ir el elemento en el subvector “destino” lo hace de forma dicotómica. Es decir, dividiendo el vector “destino” en dos mitades sucesivamente hasta encontrar la posición correcta.
  - Cuando encuentra la posición mueve los restantes elementos hacia la derecha

33

## 4. Obtención de cotas de complejidad

### Algoritmos de ordenación

#### SELECCIÓN DIRECTA

EN CADA ITERACIÓN, SELECCIONA EL MÍNIMO DE ORIGEN Y LO INTERCAMBIA POR EL ÚLTIMO DE DESTINO.

```
funcion SELECCION_DIRECTA (var a:vector[natural]; n:natural)
var i, j, posmin: entero; min:natural fvar
comienzo
  para i:=1 hasta n-1 hacer
    min:=a[i]; posmin:=i
    para j:=i+1 hasta n hacer
      si a[j]<min entonces
        min:=a[j]; posmin:=j
      fsi
    fpara
    a[posmin]:=a[i]; a[i]:=min
  fpara
fin
```

Este algoritmo busca el menor elemento del subvector  $a[i..n-1]$  y lo intercambia con el elemento que está en la posición  $i$

34

## 4. Obtención de cotas de complejidad

### Algoritmos de ordenación

#### INTERCAMBIO DIRECTO (Burbuja)

Ir recorriendo el vector de derecha a izquierda  $n-i$  veces e ir intercambiando cada elemento con el anterior si es que es menor.

```
funcion INTERCAMBIO_DIRECTO (var a:vector[natural]; n:natural )
var i,j:entero  fvar
comienzo
    para i:=2 hasta n hacer
        para j:=n hasta i hacer
            si a[j]< a[j-1] entonces
                SWAP(a[j],a[j-1])
            fsi
        fpara
    fpara
fin
```

## TEMA 3 El tipo árbol

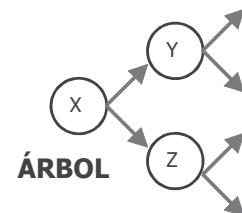
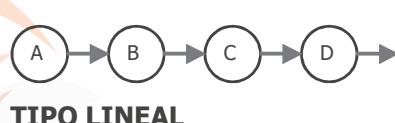
PROGRAMACIÓN Y ESTRUCTURAS DE DATOS

### Tipo árbol

- 1. Definiciones generales
- 2. Árboles binarios
- 3. Árboles de búsqueda
  - 3.1. Árboles binarios de búsqueda
  - 3.2. Árboles AVL
  - 3.3. Árboles 2-3
  - 3.4. Árboles 2-3-4

## 1. Definiciones generales (I)

- La estructura de datos árbol aparece porque los elementos que lo constituyen mantienen una estructura jerárquica, obtenida a partir de estructuras lineales, al eliminar el requisito de que cada elemento tiene como máximo un sucesor:

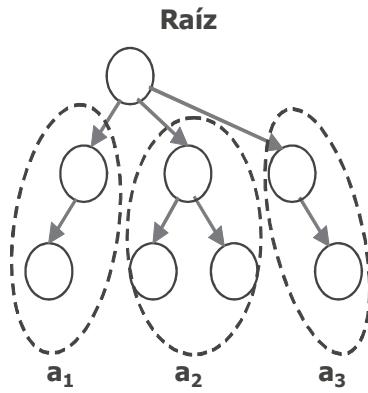


- Los elementos de los árboles se llaman nodos

3

## 1. Definiciones generales (II)

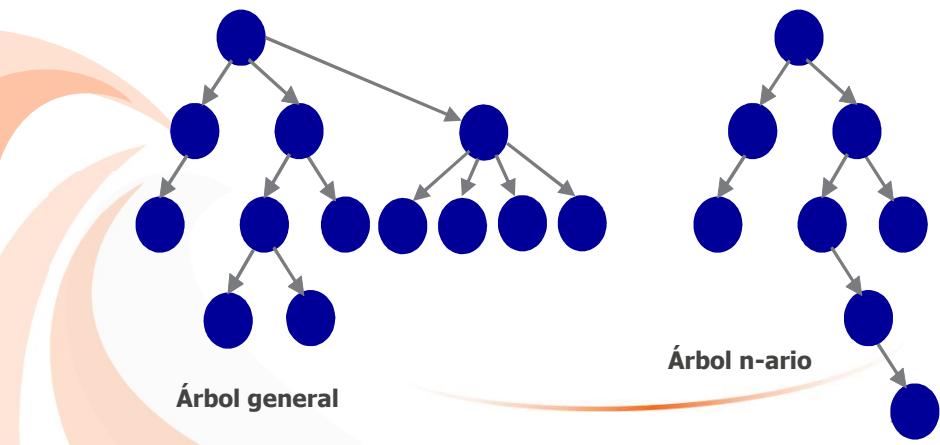
- Definición inductiva de árbol:
  - un único nodo es un **árbol (raíz)**
  - dados  $n$  árboles  $a_1, \dots, a_n$  se puede construir uno nuevo como resultado de enraizar un nuevo nodo con los  $n$  árboles. Los árboles  $a_i$  pasan a ser **subárboles** del nuevo árbol y el nuevo nodo se convierte en raíz del nuevo árbol
- Árbol vacío o nulo  $\Rightarrow 0$  nodos



4

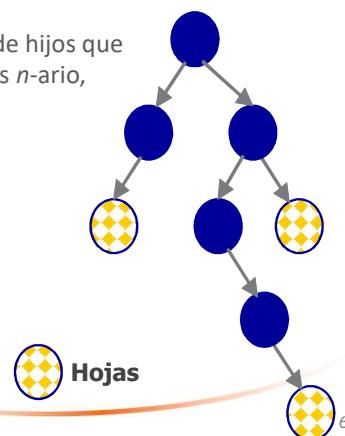
## 1. Definiciones generales (III)

- El proceso de enraizar puede involucrar:
  - un nº indeterminado de subárboles (**árboles generales**)
  - o bien, un nº máximo  $n$  de subárboles (**árboles n-arios**)



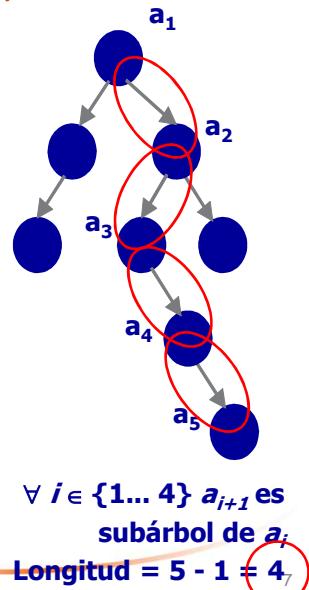
## 1. Definiciones generales (IV)

- Un árbol n-ario con  $n = 2$  se denomina **árbol binario**
- La información almacenada en los nodos del árbol se denomina **etiqueta**
- Las **hojas** son árboles con un solo nodo (árboles binarios: árbol compuesto por una raíz y 2 subárboles vacíos)
- **Grado** de un árbol es el número máximo de hijos que pueden tener sus subárboles (si el árbol es  $n$ -ario, el grado es  $n$ )



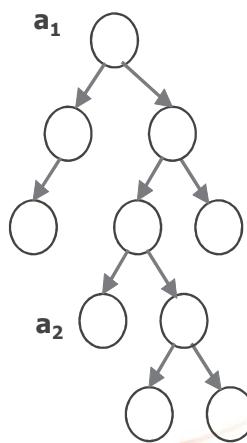
## 1. Definiciones generales (V)

- Camino:
  - es una secuencia  $a_1, \dots, a_s$  de árboles tal que,  $\forall i \in \{1 \dots s - 1\}$ ,  $a_{i+1}$  es subárbol de  $a_i$
  - el número de subárboles de la secuencia menos uno, se denomina **longitud del camino**  
(Consideraremos que existe un camino de longitud 0 de todo subárbol a sí mismo)



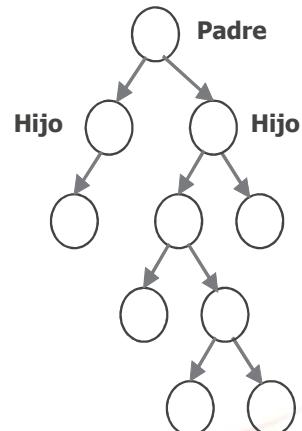
## 1. Definiciones generales (VI)

- $a_1$  es **ascendiente** de  $a_2$  (y  $a_2$  es **descendiente** de  $a_1$ ) si existe un camino  $a_1, \dots, a_2$   
(Según la definición de camino, todo subárbol es ascendiente/descendiente de sí mismo)
- Los ascendientes (descendientes) de un árbol, excluido el propio árbol, se denominan **ascendientes (descendientes) propios**



## 1. Definiciones generales (VII)

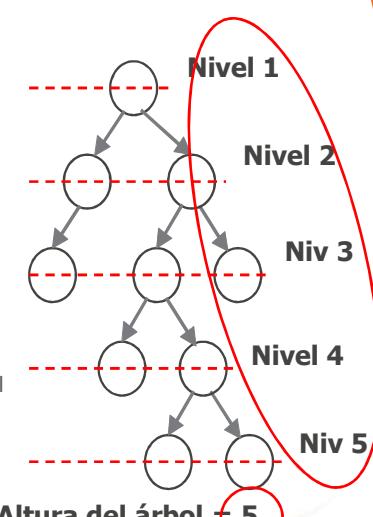
- **Padre** es el primer ascendiente propio, si existe, de un árbol
- **Hijos** son los primeros descendientes propios, si existen, de un árbol
- **Hermanos** son subárboles con el mismo padre
- **Profundidad** de un subárbol es la longitud del único camino desde la raíz a dicho subárbol



9

## 1. Definiciones generales (VIII)

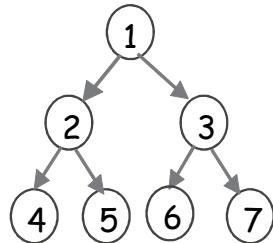
- **Nivel** de un nodo:
  - el nivel de un árbol vacío es 0
  - el nivel de la raíz es 1
  - si un nodo está en el nivel  $i$ , sus hijos están en el nivel  $i + 1$
- **Altura (profundidad)** de un árbol:
  - es el máximo nivel de los nodos de un árbol



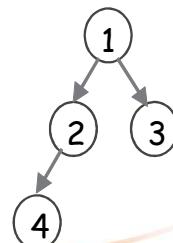
10

## 1. Definiciones generales (IX)

- **Árbol lleno** es un árbol en el que todos sus subárboles tienen  $n$  hijos (siendo  $n$  el grado del árbol) y todas sus hojas tienen la misma profundidad



- **Árbol completo** es un árbol cuyos nodos corresponden a los nodos numerados (la numeración se realiza desde la raíz hacia las hojas y, en cada nivel, de izquierda a derecha) de 1 a  $n$  en el árbol lleno del mismo grado. **Todo árbol lleno es completo**



11

## 2. Árboles binarios

- Definición de árbol binario y propiedades
- Especificación algebraica
- Recorridos
- Enriquecimiento de la especificación
- Representación secuencial y enlazada
- Otras operaciones interesantes
- Ejercicios

12

## 2. Árboles binarios

### DEFINICIÓN

- Un árbol binario es un conjunto de elementos del mismo tipo tal que:
  - o bien es el conjunto vacío, en cuyo caso se denomina **árbol vacío** o **nulo**
  - o bien no es vacío, y por tanto existe un elemento distinguido llamado **raíz**, y el resto de los elementos se distribuyen en dos subconjuntos disjuntos, cada uno de los cuales es un árbol binario llamados, respectivamente **subárbol izquierdo** y **subárbol derecho** del árbol original

13

## 2. Árboles binarios

### PROPIEDADES (I)

- Propiedades:
  - El máximo número de nodos en un **nivel i** de un árbol binario es  $N(i) = 2^{i-1}$   $i \geq 1$

#### Demostración

#### **Base inducción**

nivel 1 (raíz):  $N(1) = 2^{1-1} = 2^0 = 1$  (se cumple)

#### **Paso inductivo**

Se desea probar  $N(i-1) \Rightarrow N(i)$ , es decir, a partir de la suposición “temporal” de que  $N$  es cierta para  $i-1$  debemos probar que es cierta para  $i$

nivel  $i-1$ :  $N(i-1) = 2^{(i-1)-1} = 2^{i-2}$  (suponemos cierto)

$$\text{nivel } i : N(i) = N(i-1) * 2 = 2^{i-2} * 2 = 2^{i-2+1} = 2^{i-1}$$

14

## 2. Árboles binarios

### PROPIEDADES (II)

- El máximo número de nodos en un **árbol binario de altura k** es

$$N(k) = 2^k - 1, k \geq 1$$

#### Demostración

nivel 1:  $2^{1-1} = 1$  nodo  
 nivel 2:  $2^{2-1} = 2$  nodos  
 nivel 3:  $2^{3-1} = 4$  nodos  
 ...  
 nivel k:  $2^{k-1}$  nodos

$$\text{Altura } k = 2^{1-1} + 2^{2-1} + \dots + 2^{k-1} =$$

$$\left[ \text{S.P.G. } (r = 2, a_1 = 2^0, n = k) \right]$$

$$S_n = \begin{cases} \frac{a_1(r^n - 1)}{r - 1} & ; r \neq 1 \\ n a_1 & ; r = 1 \end{cases}$$

$$= 1 (2^k - 1) / 2 - 1 = 2^k - 1$$

15

## 2. Árboles binarios

### ESPECIFICACIÓN ALGEBRAICA (I)

#### MODULO ARBOLES\_BINARIOS

USA BOOL, NATURAL

##### PARAMETRO TIPO item

OPERACIONES A

error\_item() → item

##### FPARAMETRO arbin

##### OPERACIONES

crea\_arbin() → arbin

enraizar( arbin, item, arbin ) → arbin

raiz( arbin ) → item

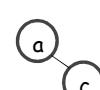
esvacio( arbin ) → bool

hijoiz, hijode( arbin ) → arbin

altura( arbin ) → natural



E(create, a, crear)



E(create, a, E(create, c, crear))

VAR i, d: arbin; x: item;

ECUACIONES

$$\boxed{\text{raiz( crea_arbin( ) ) = error_item( )}}$$

$$\text{raiz( enraizar( i, x, d ) ) = x}$$

$$\boxed{\text{hijoiz( crea_arbin( ) ) = crea_arbin( )}}$$

$$\boxed{\text{hijoiz( enraizar( i, x, d ) ) = i}}$$

$$\boxed{\text{hijode( crea_arbin( ) ) = crea_arbin( )}}$$

$$\boxed{\text{hijode( enraizar( i, x, d ) ) = d}}$$

$$\boxed{\text{esvacio( crea_arbin( ) ) = CIERTO}}$$

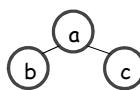
$$\boxed{\text{esvacio( enraizar( i, x, d ) ) = FALSO}}$$

$$\boxed{\text{altura( crea_arbin( ) ) = 0}}$$

$$\boxed{\text{altura( enraizar( i, x, d ) ) =}}$$

$$1 + \max(\text{altura}(i), \text{altura}(d))$$

FMODULO



E(E(create, b, crear) a, E(create, c, crear))

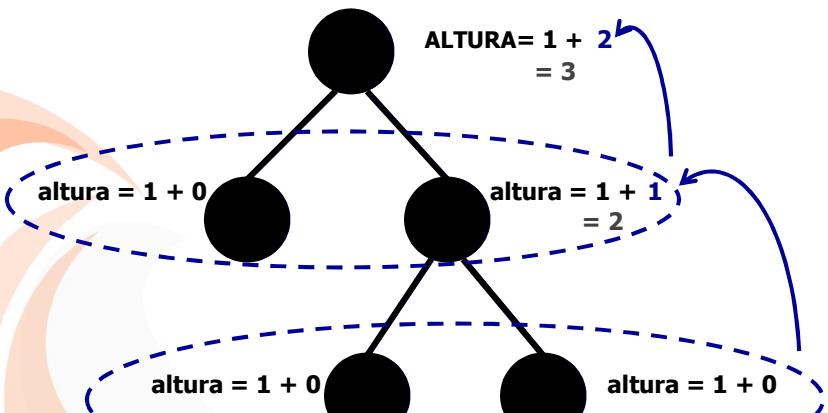
16

## 2. Árboles binarios

ESPECIFICACIÓN ALGEBRAICA (II)

$$\text{altura}(\text{crea\_arbin}()) = 0$$

$$\text{altura} = 1 + \max(\text{altura}(\text{hijoiz}), \text{altura}(\text{hijode}))$$



17

## 2. Árboles binarios

EJERCICIOS *nodosHoja*

2) Sea un árbol binario. Especificar la sintaxis y semántica de las operaciones:

**nodosHoja**, que devuelve el número de nodos hoja de un árbol binario

18

## 2. Árboles binarios

### EJERCICIOS simétricos y todos

3) Sea un árbol binario cuyas etiquetas son números naturales. Especificar la sintaxis y semántica de las operaciones:

- a) **simétricos**, que comprueba que 2 árboles binarios son simétricos
- b) **todos**, que calcula la suma de todas las etiquetas de los nodos del árbol

Nota: Especificar la sintaxis de todas las operaciones de árboles binarios usadas

19

## 2. Árboles binarios

### EJERCICIOS transforma

4) Se define la operación transforma que recibe un árbol binario y devuelve un árbol binario. Explicar qué hace esta operación detallando el comportamiento de las dos ecuaciones que aparecen a continuación:

**VAR**  $i, d$ : arbin;  $x$ : item,

$\text{transforma}(\text{crea\_arbin}()) = \text{crea\_arbin}()$

$\text{transforma}(\text{enraizar}(i, x, d)) =$

$\text{enraizar}(\text{transforma}(i), x + \text{todos}(i) + \text{todos}(d), \text{transforma}(d))$

Nota: La operación *todos* calcula la suma de todas las etiquetas de los nodos del árbol (números naturales)

20

## 2. Árboles binarios

### EJERCICIOS *quita\_hojas*

- 5) Utilizando exclusivamente las operaciones *crea\_arbin()* y *enraizar(arbin, item, arbin)* definir la sintaxis y la semántica de la operación *quita\_hojas* que actúa sobre un árbol binario y devuelve el árbol binario original sin sus hojas

21

## 2. Árboles binarios

### EJERCICIOS *dos\_hijos*

- 6) Especificar la sintaxis y la semántica de la operación *dos\_hijos* que actúa sobre un árbol binario y devuelve CIERTO si todos los nodos tienen dos hijos (excepto los nodos hoja)

22

## 2. Árboles binarios

### RECORRIDOS

- Recorrer un árbol es visitar cada nodo del árbol una sola vez
- Recorrido de un árbol es la lista de etiquetas del árbol ordenadas según se visitan los nodos
- Se distinguen dos categorías básicas de recorrido:
  - **recorridos en profundidad**
  - **recorridos en anchura o por niveles**

23

## 2. Árboles binarios

### RECORRIDOS EN PROFUNDIDAD (I)

- Si representamos por I: ir hacia la izquierda, R: visitar o escribir el ítem, D: ir hacia la derecha, existen 6 posibles formas de recorrido en profundidad: RID,IRD, IDR, RDI, DRI y DIR. Si sólo queremos hacer los recorridos de izquierda a derecha quedan 3 formas de recorrido:

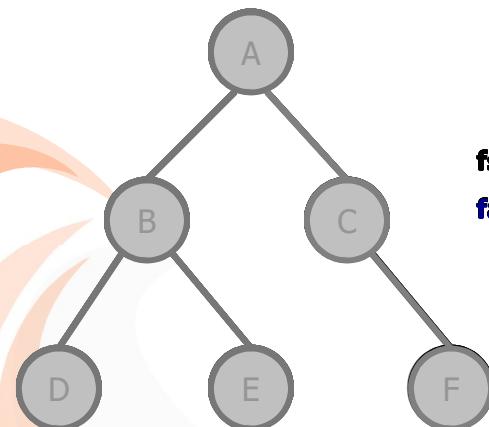
- 1. RID o preorden** (orden previo)
- 2. IRD o inorden** (orden simétrico)
- 3. IDR o postorden** (orden posterior)

(El recorrido en postorden (**DIR**) es el inverso especular del recorrido preorden (**RID**), es decir, se recorre el árbol en preorden, visitando primero el subárbol derecho antes que el izquierdo, y se considera la lista resultante como el inverso de la solución)

24

## 2. Árboles binarios

RECORRIDOS EN PROFUNDIDAD (II)



**algoritmo preorden ( a : arbin )**

si ( no esvacio( a ) ) entonces

    escribe ( raiz ( a ) )

    preorden ( hijoiz ( a ) )

    preorden ( hijode ( a ) )

fsi

falgoritmo

**PREORDEN: A B D E C F (RID)**

25

## 2. Árboles binarios

RECORRIDOS EN PROFUNDIDAD (III)

Tema 3. El tipo árbol

**algoritmo inorder ( a : arbin )**

si ( no esvacio( a ) ) entonces

    inorden ( hijoiz ( a ) )

    escribe ( raiz ( a ) )

    inorden ( hijode ( a ) )

fsi

falgoritmo

**algoritmo postorden ( a : arbin )**

si ( no esvacio( a ) ) entonces

    postorden ( hijoiz ( a ) )

    postorden ( hijode ( a ) )

    escribe ( raiz ( a ) )

fsi

falgoritmo

26

## 2. Árboles binarios

### RECORRIDO EN ANCHURA (NIVELES)

- Consiste en visitar los nodos desde la raíz hacia las hojas, y de izquierda a derecha dentro de cada nivel

```
algoritmo niveles ( a : arbin )
var c: cola de arbin; aux: arbin; fvar
encolar(c, a)
mientras no esvacia(c) hacer
    aux := cabeza(c)
    escribe (raiz(aux))
    desencolar(c)
    si no esvacio(hijoiz(aux)) entonces encolar(c, hijoiz(aux))
    si no esvacio(hijode(aux)) entonces encolar(c, hijode(aux))
fmientras
falgoritmo
```

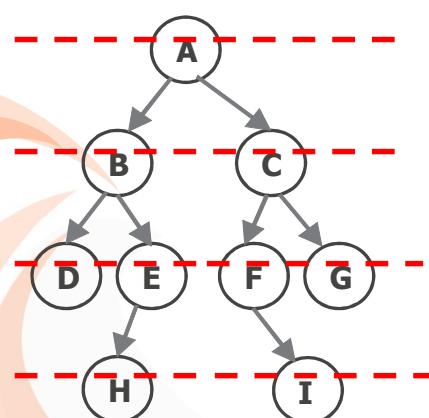
27

## 2. Árboles binarios

### EJEMPLO DE RECORRIDOS (I)

- Niveles:

**a b c d e f g h i**



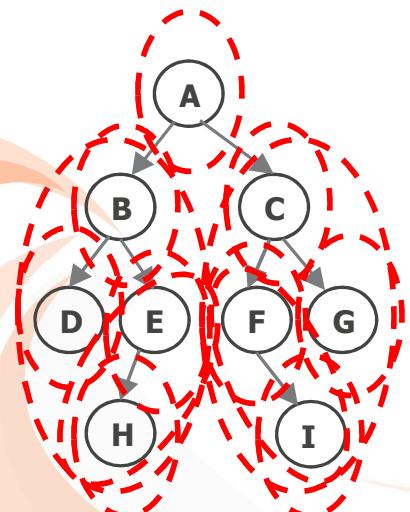
28

## 2. Árboles binarios

EJEMPLO DE RECORRIDOS (II)

- Inorden: **(IRD)**

**d b h e a f i c g**



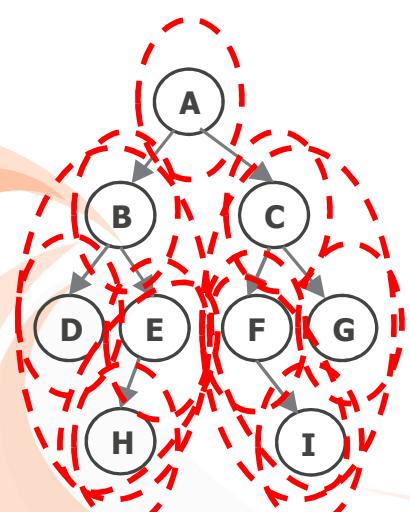
29

## 2. Árboles binarios

EJEMPLO DE RECORRIDOS (III)

- Postorden: **(IDR)**

**d h e b i f g c a**



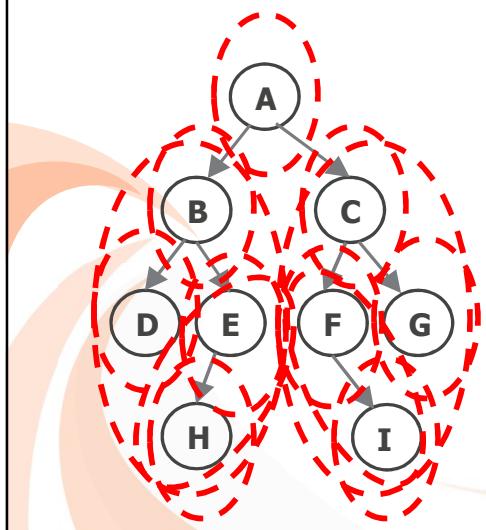
30

## 2. Árboles binarios

EJEMPLO DE RECORRIDOS (IV)

- Preorden: **(RID)**

**a b d e h c f i g**



31

## 2. Árboles binarios

ENRIQUECIMIENTO DE LA ESPECIFICACIÓN

### OPERACIONES

**preorden, inorder, postorden( arbin ) → lista**

**nodos( arbin ) → natural**

**eshoja( arbin ) → bool**

VAR i, d: arbin; x: item;

### ECUACIONES

**preorden( crea\_arbin() ) = crea\_lista()**

**preorden( enraizar( i, x, d ) ) = concatenar( insiz( x, preorden( i ) ), preorden( d ) ) (RID)**

**inorder( crea\_arbin() ) = crea\_lista()**

**inorder( enraizar( i, x, d ) ) = concatenar( insde( inorder( i ), x ), inorder( d ) ) (IRD)**

**postorden( crea\_arbin() ) = crea\_lista()**

**postorden( enraizar( i, x, d ) ) = insde( concatenar( postorden( i ), postorden( d ) ), x ) (IDR)**

**nodos( crea\_arbin() ) = 0**

**nodos( enraizar( i, x, d ) ) = 1 + nodos( i ) + nodos( d )**

**eshoja( crea\_arbin() ) = FALSO**

**eshoja( enraizar( i, x, d ) ) = esvacio( i ) ∧ esvacio( d )**

32

## 2. Árboles binarios

### REPRESENTACIÓN SECUENCIAL Y ENLAZADA (I)

- Representación secuencial

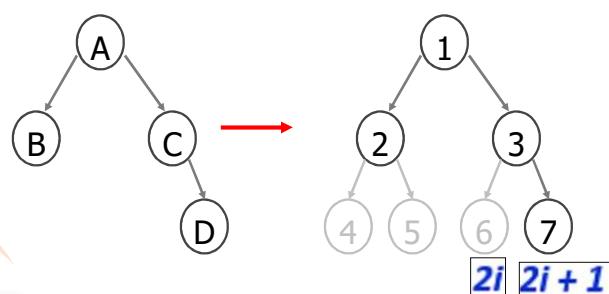
Se numeran secuencialmente los nodos del árbol hipotéticamente lleno desde la raíz a las hojas por niveles (comenzando por el nivel 1, después el nivel 2, etc.) y de izquierda a derecha en cada nivel. La representación secuencial se puede hacer usando un vector unidimensional:

- la raíz se guarda en la dirección 1
- si un nodo  $n$  está en la dirección  $i$ , entonces su hijo izquierdo estará en la dirección  $2i$  y su hijo derecho en la dirección  $2i + 1$

33

## 2. Árboles binarios

### REPRESENTACIÓN SECUENCIAL Y ENLAZADA (II)



A	B	C				D
1	2	3	4	5	6	7
$i$			$2i$	$2i + 1$		

34

## 2. Árboles binarios

### REPRESENTACIÓN SECUENCIAL Y ENLAZADA (III)

- Representación enlazada

```

typedef int TItem;
class TNodo;

class TArbin{
    public:
        TArbin ( );
        TArbin (const TArbin & origen);
        ~TArbin ( );
        TArbin & operator = (const TArbin & a);
        void Enraizar ( TArbin &iz, const TItem c, TArbin &de);
        TItem & Raiz ( );
        TArbin Hijoiz ( ); TArbin HijoDe ( );
        bool EsVacio ( );
        int Altura ( );
    private:
        void Copiar (const TArbin & origen);
        TNodo *farb;
        TItem item_error;
};
```

35

## 2. Árboles binarios

### REPRESENTACIÓN SECUENCIAL Y ENLAZADA (IV)

```

class TNodo{
    friend class TArbin;
    private:
        TItem item;
        TArbin fiz, fde;
    };

/* -----
TArbin::TArbin ( ) {farb = NULL; }
-----*/
TArbin::TArbin (const TArbin & origen){
    Copiar (origen);
}
/* -----
void
TArbin::Copiar (const TArbin & origen){
    if (origen.farb != NULL){
        TNodo *aux = new TNodo();
        aux -> item = origen.farb -> item;
        farb = aux;
        (farb -> fiz).Copiar (origen.farb -> fiz);
        (farb -> fde).Copiar (origen.farb -> fde);
    }
    else farb = NULL;
}
```

36

## 2. Árboles binarios

### REPRESENTACIÓN SECUENCIAL Y ENLAZADA (V)

```

TArbin::~TArbin () {
    if (farb != NULL){
        delete farb;
        farb = NULL;}
}
/* -----
TArbin &
TArbin::operator = (const TArbin & a){
    this → ~TArbin();
    Copiar (a);
    return *this;
}
/* -----
void
TArbin::Enraizar (TArbin &iz, const TItem c, TArbin &de){
    TNodo *aux = new TNodo();
    aux → fitem = c;
    (aux → fiz).farb = iz.farb;
    (aux → fde).farb = de.farb;
    iz.farb = de.farb = NULL;
    this → ~TArbin();
    farb = aux;
}
//deja vacíos el árbol original (*this), iz y de

```

37

## 2. Árboles binarios

### REPRESENTACIÓN SECUENCIAL Y ENLAZADA (VI)

```

TItem &
TArbin::Raiz (){
    if (farb != NULL) return farb → fitem;
    else return item_error;
}

/* -----
TArbin
TArbin::HijoIzq (){
    if (farb != NULL) return farb → fiz;
    else return TArbin();
}

/* -----
TArbin
TArbin::HijoDe (){
    if (farb != NULL) return farb → fde;
    else return TArbin();
}

```

38

## 2. Árboles binarios

### REPRESENTACIÓN SECUENCIAL Y ENLAZADA (VII)

```
bool  
TArbin::EsVacio ( ){  
    return (farb == NULL)  
}  
  
/* ----- */  
  
int  
TArbin::Altura ( ){  
    int a1, a2;  
  
    if (farb != NULL){  
        a1 = (farb -> fiz).Altura( );  
        a2 = (farb -> fde).Altura( );  
        return (1 + (a1 < a2 ? a2 : a1));  
    }  
    else return 0;  
}
```

39

## 2. Árboles binarios

### REPRESENTACIÓN SECUENCIAL Y ENLAZADA (VIII)

```
/* Programa de prueba  
  
int  
main ( ){  
  
    TArbin a, b, c;  
  
    a.Enraizar (b, 1, c);  
    b.Enraizar (a, 2, c);  
    cout << "el hijo izquierdo del árbol contiene un " << (b.HijoIzq( )).Raiz( );  
    // ESCRIBE 1  
    cout << "la altura del árbol es " << b.Altura() << endl;  
    // ESCRIBE 2  
}
```

40

## 2. Árboles binarios

REPRESENTACIÓN SECUENCIAL Y ENLAZADA (IX)

¿Constructor y destructor de TNodo?

```
TNodo::TNodo( ):fiz(),fde(){
    item=0;
}
```

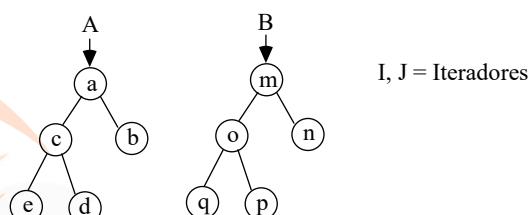
```
TNodo::~TNodo( ){
    item=0;
}
```

41

## 2. Árboles binarios

OTRAS OPERACIONES INTERESANTES (I)

- Además de todas las operaciones vistas anteriormente, utilizaremos las operaciones de asignación y “movimiento” de árboles e iteradores:



I, J = Iteradores

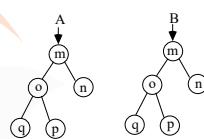
42

## 2. Árboles binarios

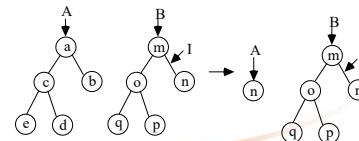
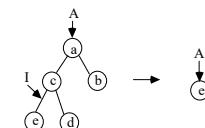
### OTRAS OPERACIONES INTERESANTES (II)

- a) Asignación ( copia ) entre árboles e iteradores:

– a1) **A = B**. Hace una copia de B en A



– a2) **A = I**. Hace una copia sobre el árbol A, de la rama del árbol a la que apunta el Iterador I



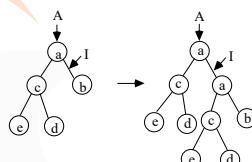
43

## 2. Árboles binarios

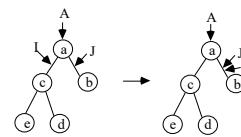
### OTRAS OPERACIONES INTERESANTES (III)

- a) Asignación ( copia ) entre árboles e iteradores:

– a3) **I = A**. Hace una copia sobre la rama del árbol a la que apunta el Iterador I del árbol A



– a4) **I = J**. Sirve para inicializar el Iterador I de forma que apunte al mismo nodo al que apunta el Iterador J



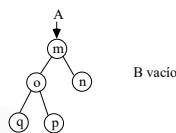
44

## 2. Árboles binarios

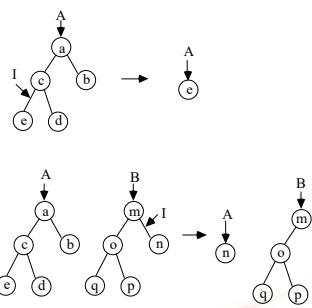
### OTRAS OPERACIONES INTERESANTES (IV)

- b) *Movimiento de ramas entre árboles e iteradores:*

- b1) **Mover ( A, B ).** Mueve el árbol B al árbol A. B se queda vacío



- b2) **Mover ( A, I ).** Mueve la rama del árbol A a la que apunta el Iterador I al árbol A



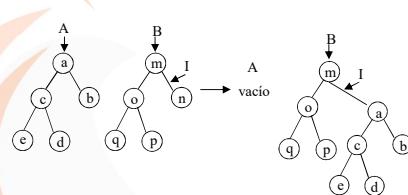
45

## 2. Árboles binarios

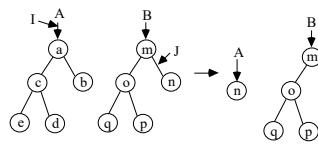
### OTRAS OPERACIONES INTERESANTES (V)

- b) *Movimiento de ramas entre árboles e iteradores:*

- b3) **Mover ( I, A ).** Mueve el árbol A a la rama del árbol a la que apunta el Iterador I



- b4) **Mover ( I, J ).** Mueve la rama del árbol a la que apunta el Iterador J a la rama del árbol a la que apunta el Iterador I



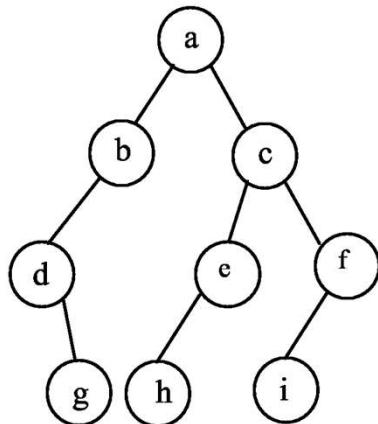
46

## 2. Árboles binarios

### EJERCICIOS recorridos

**1a)** Dado el siguiente árbol binario, calcular los recorridos preorden, postorden, inorden y niveles

**1b)** ¿Se puede reconstruir un árbol binario dando solamente su recorrido inorden? ¿Cuántos recorridos como mínimo son necesarios? ¿Cuáles?



47

## 2. Árboles binarios

### Preguntas de tipo test: Verdadero vs. Falso

- El nivel de un nodo en un árbol coincide con la longitud del camino desde la raíz a dicho nodo
- Dado un único recorrido de un árbol binario lleno, es posible reconstruir dicho árbol
- Un árbol binario completo con  $n$  nodos y altura  $k$  es un árbol binario lleno para esa misma altura

48

### 3. Árboles de búsqueda (I)

- Árboles de búsqueda = Árboles n-arios de búsqueda = Árboles multicamino de búsqueda
- Son un tipo particular de árboles, que pueden definirse cuando el tipo de los elementos del árbol posee una relación  $\leq$  de orden total
- Un árbol multicamino de búsqueda T es un árbol n-ario vacío o cumple las siguientes propiedades:
  - 1. La raíz de T contiene  $A_0, \dots, A_{n-1}$  subárboles y  $K_1, \dots, K_{n-1}$  etiquetas
  - 2.  $K_i < K_{i+1}, 1 \leq i < n-1$
  - 3. Todas las etiquetas del subárbol  $A_i$  son:
    - menores que  $K_{i+1} \quad 0 \leq i < n-1$
    - mayores que  $K_i \quad 0 < i \leq n-1$
  - 4. Los subárboles  $A_i, 0 \leq i \leq n-1$  son también árboles multicamino de búsqueda

$K_1$	$K_2$	$K_3$	$\dots$	$K_{n-1}$
$A_0$	$A_1$	$\dots$	$A_{n-1}$	

49

### 3. Árboles de búsqueda (II)

- Algoritmo de búsqueda
  - Para buscar un valor  $x$  en el árbol, primero se mira el nodo raíz y se realiza la siguiente comparación:
    - $x < K_i$  ó  $x > K_i$  ó  $x = k_i$  ( $1 \leq i \leq n-1$ )
    - 1) En el caso que  $x = K_i$ , la búsqueda ya se ha completado
    - 2) Si  $x < K_i$ , entonces por la definición de árbol multicamino de búsqueda,  $x$  debe estar en el subárbol  $A_{i-1}$ , si éste existe en el árbol
    - 3) Si  $x > K_i$ ,  $x$  debe estar en  $A_i$
- Los árboles multicamino de búsqueda son útiles cuando la memoria principal es insuficiente para utilizarla como almacenamiento permanente
- En una representación enlazada de estos árboles, los punteros pueden representar direcciones de disco en lugar de direcciones de memoria principal. ¿Cuántas veces se accede a disco cuando se realiza una búsqueda? ¿Cómo se puede reducir el número de accesos a disco?

50

## 3.1. Árboles binarios de búsqueda

### ESPECIFICACIÓN ALGEBRAICA (I)

- Propiedades
  - todos los elementos en el subárbol izquierdo son  $\leq$  que la raíz,
  - todos los elementos en el subárbol derecho son  $\geq$  que la raíz,
  - los dos subárboles son binarios de búsqueda
  - en algunas variantes no se permite la repetición de etiquetas

**MODULO ARBOL\_BIN\_BUSQUEDA USA BOOL, ARBOLES\_BINARIOS**

**PARAMETRO TIPO item**

**OPERACIONES**

$<, ==, >$ : item, item  $\rightarrow$  bool  
 $\text{error\_item}()$   $\rightarrow$  item

**FPARAMETRO**

**OPERACIONES**

$\text{insertar}(\text{arbin}, \text{item}) \rightarrow \text{arbin}$   
 $\text{buscar}(\text{arbin}, \text{item}) \rightarrow \text{bool}$   
 $\text{borrar}(\text{arbin}, \text{item}) \rightarrow \text{arbin}$   
 $\text{min}(\text{arbin}) \rightarrow \text{item}$

51

## 3.1. Árboles binarios de búsqueda

### ESPECIFICACIÓN ALGEBRAICA (II)

```
VAR i, d: arbin; x, y: item;
ECUACIONES
  insertar( crea_arbin(), x ) =
    enraizar( crea_arbin(), x, crea_arbin() )
  si ( y < x ) entonces
    insertar( enraizar( i, x, d ), y ) =
      enraizar( insertar( i, y ), x, d )
  si no si ( y > x ) insertar( enraizar( i, x, d ), y ) =
    enraizar( i, x, insertar( d, y ) ) fsi

  buscar( crea_arbin(), x ) = FALSO
  si ( y < x ) entonces
    buscar( enraizar( i, x, d ), y ) = buscar( i, y )
  si no si ( y > x ) entonces
    buscar( enraizar( i, x, d ), y ) = buscar( d, y )
  si no buscar( enraizar( i, x, d ), y ) = CIERTO fsi
```

```
borrar( crea_arbin(), x ) = crea_arbin()
si ( y < x ) entonces
  borrar( enraizar( i, x, d ), y ) =
    enraizar( borrar( i, y ), x, d )
si no si ( y > x ) entonces
  borrar( enraizar( i, x, d ), y ) =
    enraizar( i, x, borrar( d, y ) ) fsi
si ( y==x ) y esvacio( d ) entonces
  borrar( enraizar( i, x, d ), y ) = i fsi
si ( y==x ) y esvacio( i ) entonces
  borrar( enraizar( i, x, d ), y ) = d fsi
si ( y==x ) y no esvacio( d ) y no esvacio( i ) entonces
  borrar( enraizar( i, x, d ), y ) =
    enraizar( i, min( d ), borrar( d, min( d ) ) ) fsi
min( crea_arbin() ) = error_item()
si esvacio( i ) entonces min( enraizar( i, x, d ) ) = x
si no min( enraizar( i, x, d ) ) = min( i ) fsi
```

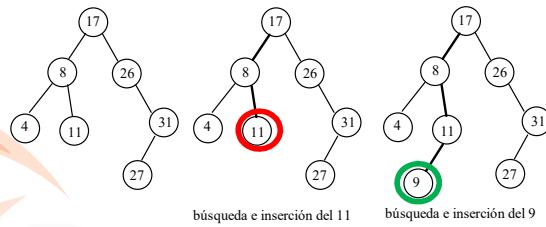
**FMODULO**

52

## 3.1. Árboles binarios de búsqueda

### OPERACIONES BÁSICAS (I)

- Búsqueda e inserción de un elemento



- Recorrido en inorder: todas las etiquetas ordenadas ascendenteamente
- ¿Cuál es el coste de las operaciones de búsqueda e inserción en el ABB?
- ¿Qué pasa si insertamos una serie de elementos ordenados en un ABB inicialmente vacío?

53

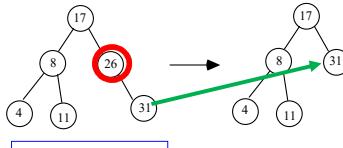
## 3.1. Árboles binarios de búsqueda

### OPERACIONES BÁSICAS (II)

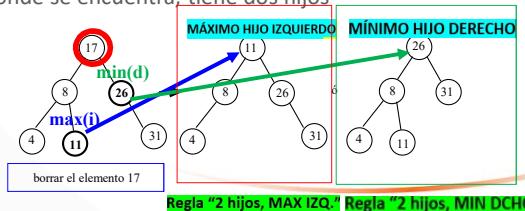
- Borrado** de un elemento

- El nodo donde se encuentra es una hoja **Regla "hoja"**
- El nodo donde se encuentra tiene un único hijo. El nodo a eliminar es sustituido por su hijo **Regla "1 hijo"**

borrado del elemento 26



- El nodo donde se encuentra tiene dos hijos



Regla "2 hijos, MAX IZQ." Regla "2 hijos, MIN DCHO."

54

### 3.1. Árboles binarios de búsqueda

EJERCICIOS *inserción y borrado*

- 1) En un árbol binario de búsqueda inicialmente vacío,
  - a) Insertar los siguientes elementos: 20, 10, 30, 40, 5, 15, 50, 22, 25, 24, 26, 3, 35, 38, 39, 37
  - b) Sobre el árbol resultante, realizar el borrado de: 5, 3, 30, 22, 39  
utilizar el criterio de sustituir por el  
**menor de la derecha**  
**mayor de la izquierda**

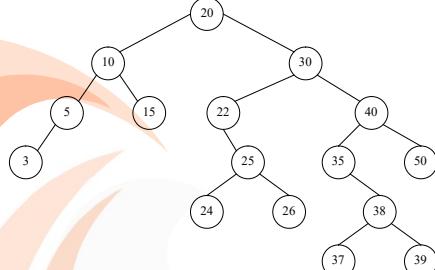
<https://www.cs.usfca.edu/~galles/visualization/BST.html>

55

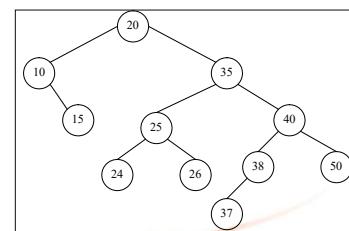
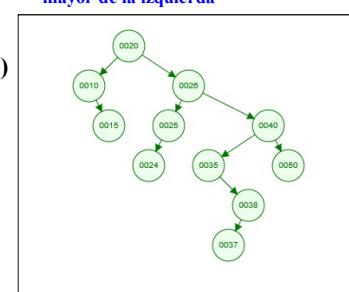
### 3.1. Árboles binarios de búsqueda

EJERCICIOS *inserción y borrado: SOLUCIÓN*

1) a)



b)



**menor de la derecha**

56

### 3.1. Árboles binarios de búsqueda

Preguntas de tipo test: Verdadero vs. Falso

- En el borrado de un elemento que se encuentre en un nodo con dos hijos no vacíos en un árbol binario de búsqueda, tenemos que intercambiar el elemento a borrar por el menor del subárbol de la izquierda o por el mayor del subárbol de la derecha
- El menor elemento en un árbol binario de búsqueda siempre se encuentra en un nodo hoja
- El coste temporal (en su peor caso) de insertar una etiqueta en un árbol binario de búsqueda es lineal respecto al número de nodos del árbol

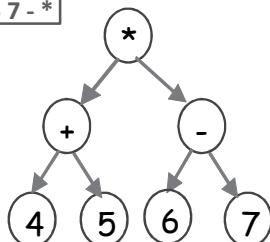
57

### Árboles binarios

Aplicaciones

- Evaluación de expresiones aritméticas (AB).

**4 5 + 6 7 - \***



- Árboles de Huffman (ABB).

- Emisor: transmisión de mensajes codificados
- Receptor: árbol decodificador

- Treesort (ABB).

- Ordenación de elementos utilizando un ABB
- Si está equilibrado  $\rightarrow \Theta(n \log n)$

58

## 3.2. Árboles AVL

### DEFINICIONES (I)

- La eficiencia en la búsqueda de un elemento en un árbol binario de búsqueda se mide en términos de:
  - Número de comparaciones
  - La altura del árbol
- Árbol completamente equilibrado: los elementos del árbol deben estar repartidos en igual número entre el subárbol izquierdo y el derecho, de tal forma que la diferencia en número de nodos entre ambos subárboles sea como mucho 1
- Problema: el mantenimiento del árbol
- Árboles AVL: desarrollado por Adelson-Velskii y Landis (1962). Los AVL son árboles balanceados (equilibrados) con respecto a la altura de los subárboles:  
*"Un árbol está equilibrado respecto a la altura si y solo si para cada uno de sus nodos ocurre que las alturas de los dos subárboles difieren como mucho en 1"*
- Consecuencia 1. Un árbol vacío está equilibrado con respecto a la altura
- Consecuencia 2. El árbol equilibrado óptimo será aquél que cumple:

$$n = 2^h - 1,$$

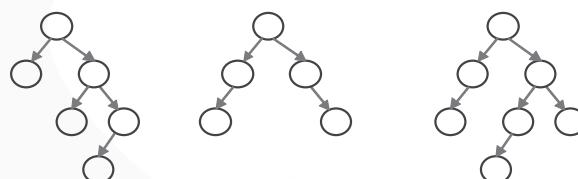
donde  $n = \text{nº nodos}$  y  $h = \text{altura}$

1

## 3.2. Árboles AVL

### DEFINICIONES (II)

- Si T es un árbol binario no vacío con TL y TR como subárboles izquierdo y derecho respectivamente, entonces T está balanceado con respecto a la altura si y solo si
  - TL y TR son balanceados respecto a la altura, y
  - $| h_L - h_R | \leq 1$  donde  $h_L$  y  $h_R$  son las alturas respectivas de TL y TR
- El factor de equilibrio FE ( T ) de un nodo T en un árbol binario se define como  $h_R - h_L$ . Para cualquier nodo T en un árbol AVL, se cumple  $FE(T) = -1, 0, 1$



2

## 3.2. Árboles AVL

### OPERACIONES BÁSICAS. INSERCIÓN (I)

- Representación de árboles AVL

- Mantener la información sobre el equilibrio de forma implícita en la estructura del árbol
- Atribuir a, y almacenar con, cada nodo el factor de equilibrio de forma explícita

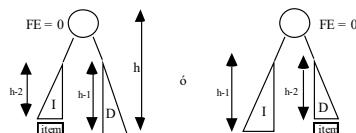
TNodoArb {

```

    TItem item;
    TArbBin fiz, fde;
    int FE; }
```

- Inserción en árboles AVL. Casos:

- Después de la inserción del ítem, los subárboles I y D igualarán sus alturas

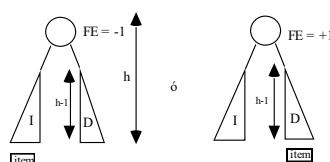


3

## 3.2. Árboles AVL

### OPERACIONES BÁSICAS. INSERCIÓN (II)

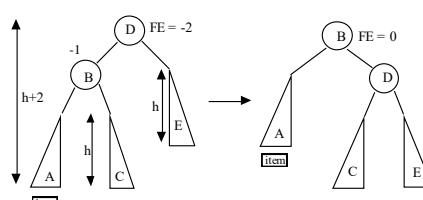
- Después de la inserción, I y D tendrán distinta altura, pero sin vulnerar la condición de equilibrio



- Si  $h_I > h_D$  y se realiza inserción en I, ó  $h_I < h_D$  y se realiza inserción en D

**Formas de rotación: II, ID, DI, DD**

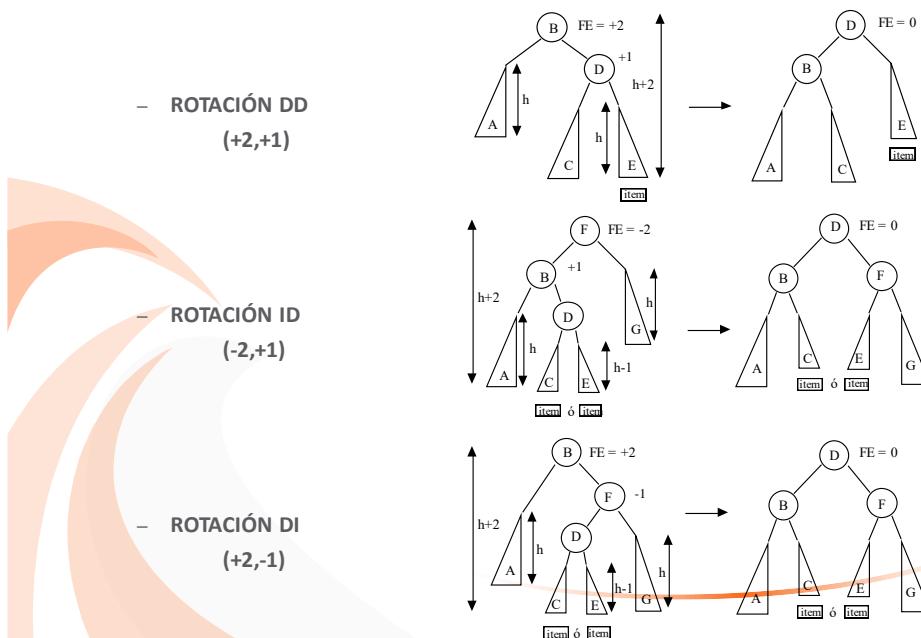
- ROTACIÓN II  
(-2,-1)



4

## 3.2. Árboles AVL

### OPERACIONES BÁSICAS. INSERCIÓN (III)

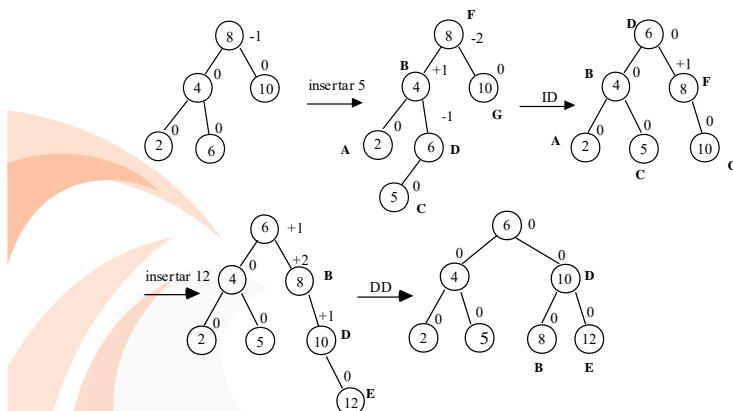


5

## 3.2. Árboles AVL

### OPERACIONES BÁSICAS. INSERCIÓN. EJEMPLO (IV)

- Ejemplo. Insertar en el siguiente árbol los elementos 5 y 12



- Hay que tener en cuenta que la actualización del FE de cada nodo se efectúa desde las hojas hacia la raíz del árbol

6

## 3.2. Árboles AVL

### OPERACIONES BÁSICAS. INSERCIÓN. IMPLEMENTACIÓN (V)

```

ALGORITMO INSERTAR
ENTRADA/SALIDA
    A: AVL; c : Item
    VAR I : Iterador ; Crece : Integer ;
    METODO
        I = Primer ( A );
        InsertarAux ( I, c, Crece );
    fMETODO
ALGORITMO INSERTARAUX
ENTRADA/SALIDA I : Iterador; Crece: Integer; c : Item ;
VAR Crecelz, CreceDe : Integer ; B : Arbol ;
METODO
    si EsVacioArblt ( I ) entonces
        B = Enraizar ( c ) ; Mover ( I, B ) ; Crece = TRUE ;
    sino
        CreceHijo = Crecelz = CreceDe = FALSE ;
        si ( c < Obtener ( I ) ) entonces
            INSERTARAUX ( HijoIzq ( I ), c, Crecelz );
            CreceHijo = Crecelz ;
        sino
            si ( c > Obtener ( I ) ) entonces
                INSERTARAUX ( HijoDer ( I ), c, CreceDe );
                CreceHijo = CreceDe ;
            fsi
        fsi
        si CreceHijo entonces
            caso de:
                1) ( Crecelz y FE ( I ) = 1 ) ó ( CreceDe y FE ( I ) = -1 ):
                    Crece = FALSE ; FE ( I ) = 0;
                2) Crecelz y FE ( I ) = 0 : FE ( I ) = -1 ; Crece = TRUE;
                3) CreceDe y FE ( I ) = 0 : FE ( I ) = 1 ; Crece = TRUE;
                4) Crecelz y FE ( I ) = -1 : EquilibrarIzquierda ( I, Crece );
                5) CreceDe y FE ( I ) = 1 : EquilibrarDerecha ( I, Crece );
            fcaso
        sino
            Crece=FALSE;
        fsi
    fsi
fMETODO

```

7

## 3.2. Árboles AVL

### OPERACIONES BÁSICAS. INSERCIÓN. IMPLEMENTACIÓN (VI)

```

ALGORITMO EQUILIBRARIZQUIERDA
ENTRADA/SALIDA I : Iterador; Crece: Integer;
VAR J, K: Iterador; in E2;
METODO
    si ( FE ( HijoIzq ( I ) ) = -1 entonces          //ROTACIÓN II
        Mover ( J, HijoIzq ( I ) );
        Mover ( HijoIzq ( I ), HijoDer ( J ) );
        Mover ( HijoDer ( J ), I );
        FE ( J ) = 0; FE ( HijoDer ( J ) ) = 0;
        Mover ( I,J );
    sino
        Mover ( J, HijoIzq ( I ) );                  //ROTACIÓN ID
        Mover ( K, HijoDer ( J ) );
        E2 = FE ( K );
        Mover ( HijoIzq ( I ), HijoDer ( K ) );
        Mover ( HijoDer ( J ), HijoIzq ( K ) );
        Mover ( HijoIzq ( K ), J );
        Mover ( HijoDer ( K ), I );
        FE ( K ) = 0;
    caso de E2
        -1: FE ( HijoIzq ( K ) ) = 0; FE ( HijoDer ( K ) ) = 1;
        +1: FE ( HijoIzq ( K ) ) = -1; FE ( HijoDer ( K ) ) = 0;
        0: FE ( HijoIzq ( K ) ) = 0; FE ( HijoDer ( K ) ) = 0;
    fcaso
    Mover ( I, K );
    fsi
    Crece = FALSE;
fMETODO

```

8

## 3.2. Árboles AVL

### EJERCICIOS inserción

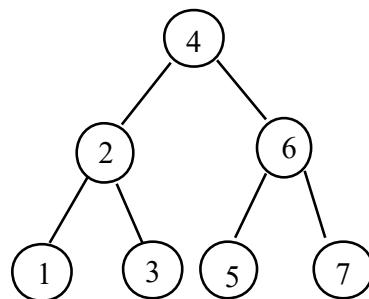
- 1) Construir un árbol AVL formado por los nodos insertados en el siguiente orden con etiquetas 4, 5, 7, 2, 1, 3, 6
- 2) Insertar las mismas etiquetas con el siguiente orden: 1, 2, 3, 4, 5, 6, 7

9

## 3.2. Árboles AVL

### EJERCICIOS inserción: SOLUCIÓN

- 1) La solución para los 2 ejercicios es la siguiente:



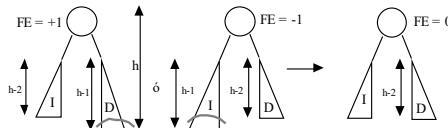
10

## 3.2. Árboles AVL

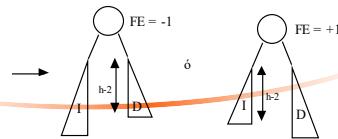
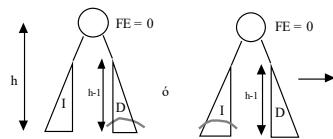
### OPERACIONES BÁSICAS. BORRADO (I)

- Borrado en árboles AVL. Casos:

– Borrar el ítem nos llevará en el árbol a un  $FE = 0$ , no será necesario reequilibrar



– Borrar el ítem nos llevará en el árbol a un  $FE = \pm 1$ , en este caso tampoco será necesario reequilibrar



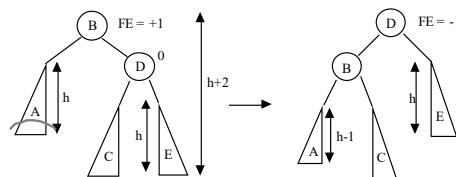
11

## 3.2. Árboles AVL

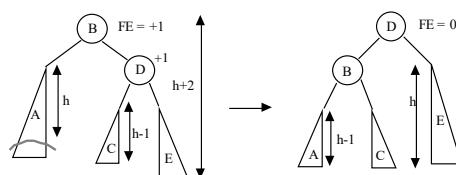
### OPERACIONES BÁSICAS. BORRADO (II)

- Rotaciones simples

#### – ROTACIÓN DD (+2,0)



(+2,+1)  
La altura del árbol decrece



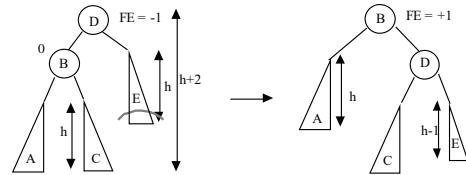
12

## 3.2. Árboles AVL

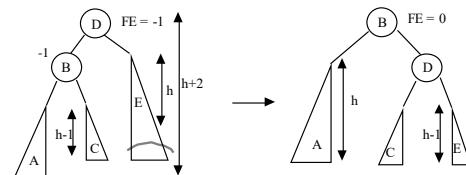
### OPERACIONES BÁSICAS. BORRADO (III)

- Rotaciones simples

#### - ROTACIÓN II (-2,0)



#### (-2,-1) La altura del árbol decrece



13

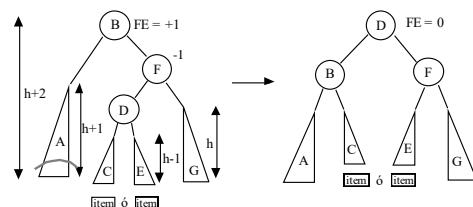
## 3.2. Árboles AVL

### OPERACIONES BÁSICAS. BORRADO (IV)

- Rotaciones dobles

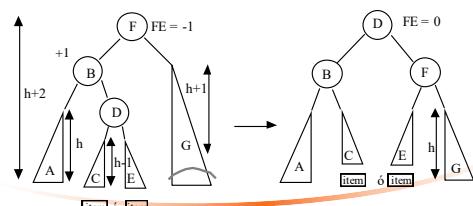
#### - ROTACIÓN DI (+2,-1)

La altura del árbol decrece



#### - ROTACIÓN ID (-2,+1)

La altura del árbol decrece



14

## 3.2. Árboles AVL

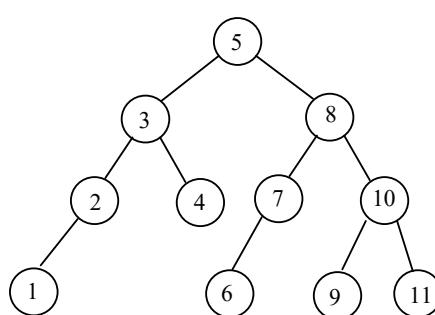
### OPERACIONES BÁSICAS. INSERCIÓN Y BORRADO

- Estudio de las complejidades de ambos algoritmos
  - El análisis matemático del algoritmo de inserción es un problema todavía no resuelto. Los ensayos empíricos apoyan la conjectura de que la altura esperada para el árbol AVL de  $n$  nodos es
 
$$h = \log_2(n) + c \quad / \quad c \text{ es una constante pequeña}$$
  - Estos árboles deben utilizarse sólo si las recuperaciones de información (búsquedas) son considerablemente más frecuentes que las inserciones → debido a la complejidad de las operac. de equilibrado
  - Se puede borrar un elemento en un árbol equilibrado con  $\log(n)$  operaciones (en el caso más desfavorable)
- Diferencias operacionales de borrado e inserción:
  - Al realizar una inserción de una sola clave se puede producir como máximo una rotación (de dos o tres nodos)
  - El borrado puede requerir una rotac. en todos los nodos del camino de búsqueda
  - Los análisis empíricos dan como resultado que, mientras se presenta una rotación por cada dos inserciones,
  - sólo se necesita una por cada cinco borrados. El borrado en árboles equilibrados, pues, tan sencillo (o tan complicado) como la inserción

## 3.2. Árboles AVL

### EJERCICIOS borrado

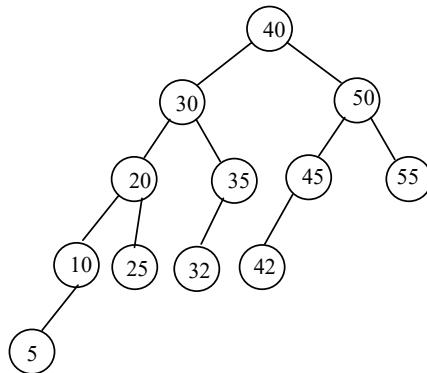
- 1) Dado el siguiente árbol AVL de entrada, efectuar los siguientes borrados en el mismo: 4, 8, 6, 5, 2, 1, 7. (Nota: al borrar un nodo con 2 hijos, sustituir por el mayor de la izquierda)



## 3.2. Árboles AVL

### EJERCICIOS borrado

- 2) Dado el siguiente árbol AVL de entrada, efectuar los siguientes borrados en el mismo: 55, 32, 40, 30. (Nota: al borrar un nodo con 2 hijos, sustituir por el mayor de la izquierda)

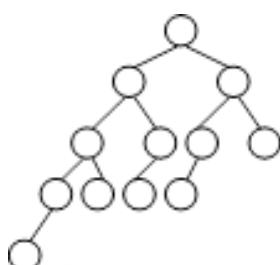


21

## 3.2. Árboles AVL

### Preguntas de tipo test: Verdadero vs. Falso

- Los árboles AVL son aquellos en los que el número de elementos en los subárboles izquierdo y derecho difieren como mucho en 1
- Cuando se realiza un borrado en un árbol AVL, en el camino de vuelta atrás para actualizar los factores de equilibrio, como mucho sólo se va a efectuar una rotación
- El siguiente árbol está balanceado con respecto a la altura

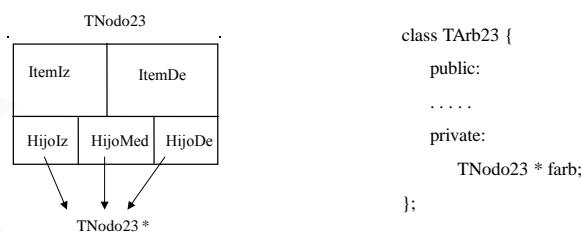


23

## 3.3. Árboles 2-3

### DEFINICIONES

- Un árbol 2-3 es un árbol que está vacío o satisface las siguientes propiedades:
  - Los nodos pueden tener 2 ó 3 hijos (2-nodo ó 3-nodo)
  - Cumple las propiedades de árbol multicamino de búsqueda
  - Todas las hojas están en el mismo nivel
- Representación



1

## 3.3. Árboles 2-3

### OPERACIONES BÁSICAS. PROPIEDADES

- Operaciones básicas:
  - Búsqueda (similar a los árboles multicamino de búsqueda)
  - Inserción (se realiza en las hojas. Se pueden producir reestructuraciones del árbol en el camino de vuelta)
  - Borrado (se realiza en las hojas. Se pueden producir reestructuraciones del árbol en el camino de vuelta)
- Propiedades:
  - En un árbol 2-3 de altura  $h$  tenemos:
    - $2^h - 1$  elementos si todos los nodos son del tipo 2-nodo
    - $3^h - 1$  elementos si todos los nodos son del tipo 3-nodo
 por lo que la altura de un árbol 2-3 con  $n$  elementos se encuentra entre los límites:  $\log_3(n+1)$  y  $\log_2(n+1)$
  - Las reestructuraciones se realizan desde las hojas hacia la raíz

2

## 3.3. Árboles 2-3

### OPERACIONES BÁSICAS. INSERCIÓN (I)

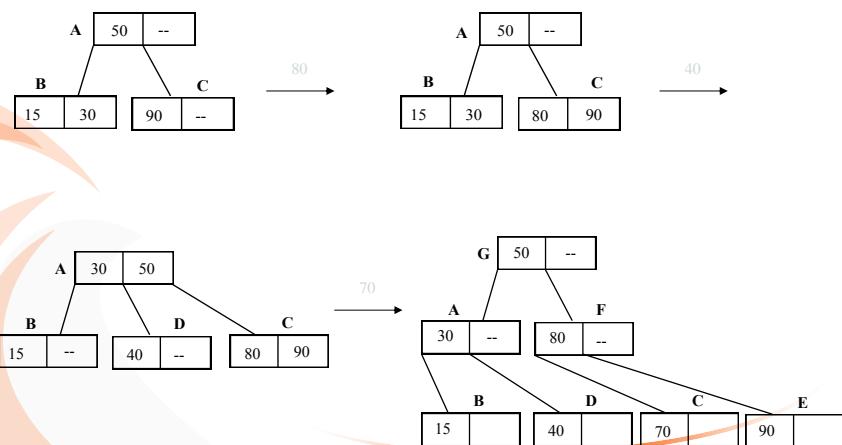
- Pueden ocurrir dos casos:
  - El elemento a insertar irá en un 2-nodo: INSERCIÓN DIRECTA
  - El elemento a insertar irá en un 3-nodo: HAY QUE CREAR UN NUEVO NODO

3

## 3.3. Árboles 2-3

### OPERACIONES BÁSICAS. INSERCIÓN (II)

- Ejemplo: Insertar en el siguiente árbol 2-3 los elementos: 80, 40 y 70



4

### 3.3. Árboles 2-3

#### EJERCICIOS inserción

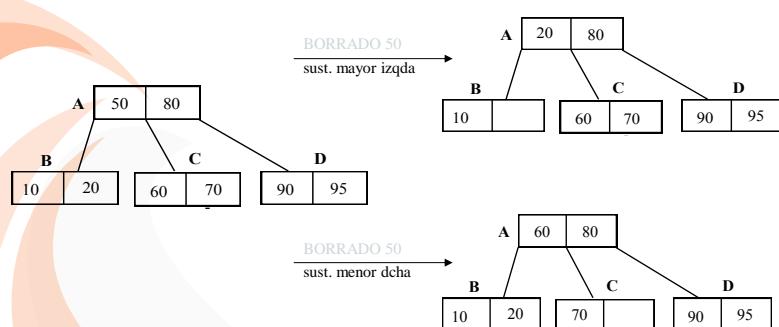
- 1) En el árbol 2-3 obtenido anteriormente, insertar los elementos 45, 47, 35, 33, 48, 49, 43 y 42

5

### 3.3. Árboles 2-3

#### OPERACIONES BÁSICAS. BORRADO (I)

- Pueden ocurrir dos casos:
  - El elemento a borrar está en un 3-nodo: BORRADO DIRECTO
  - El elemento a borrar irá en un 2-nodo: realizar una COMBINACIÓN o ROTACIÓN

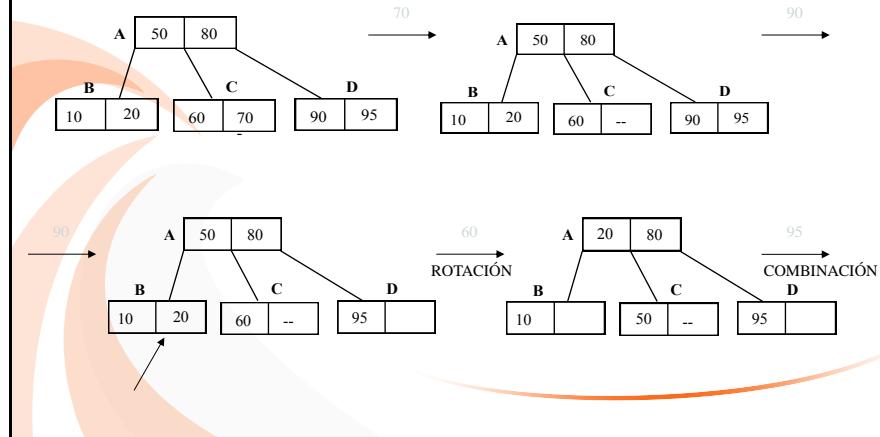


6

### 3.3. Árboles 2-3

#### OPERACIONES BÁSICAS. BORRADO (II)

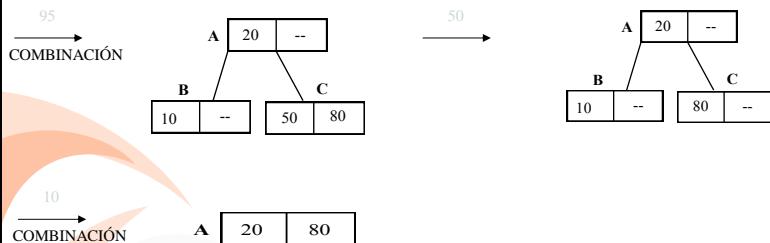
- Ejemplo: Borrar en el siguiente árbol 2-3 los elementos: 70, 90, 60, 95, 50 y 10. (Criterios: (1) si el nodo tiene dos hijos hay que sustituir por el mayor de la izquierda, (2) si el 2-nodo tiene dos hermanos, consultar el hermano de la izquierda)



7

### 3.3. Árboles 2-3

#### OPERACIONES BÁSICAS. BORRADO (III)

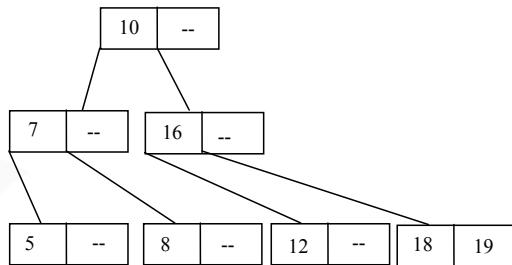


8

### 3.3. Árboles 2-3

#### EJERCICIOS borrado

- Dado el siguiente árbol 2-3 borrar los elementos 10, 7 y 18. (Criterios: (1) si el nodo tiene dos hijos hay que sustituir por el menor de la derecha, (2) si el 2-nodo tiene dos hermanos, consultar el hermano de la derecha)

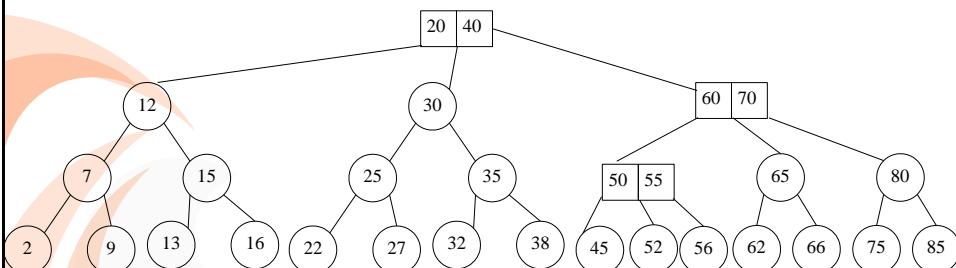


9

### 3.3. Árboles 2-3

#### EJERCICIOS borrado

- Dado el siguiente árbol 2-3 borrar los elementos 20, 30, 70 y 12. (Criterios: (1) si el nodo tiene dos hijos hay que sustituir por el menor de la derecha, (2) si el 2-nodo tiene dos hermanos, consultar el hermano de la derecha)



10

### 3.3. Árboles 2-3

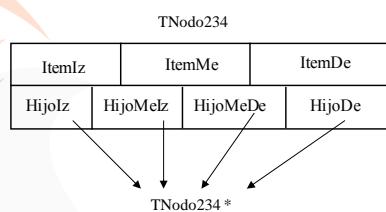
Preguntas de tipo test: Verdadero vs. Falso

- Un árbol 2-3 es un árbol 2-camino de búsqueda
- El número mínimo de elementos que se pueden almacenar en un árbol 2-3 de altura  $h$  es  $3^h - 1$
- El grado del árbol 2-3 es 2

## 3.4. Árboles 2-3-4

### DEFINICIONES

- Un árbol 2-3-4 es un árbol que está vacío o satisface las siguientes propiedades:
  - Los nodos pueden tener 2, 3 ó 4 hijos (2-nodo, 3-nodo ó 4-nodo)
  - Cumple las propiedades de árbol multicamino de búsqueda
  - Todas las hojas están en el mismo nivel
- Representación



```
class TArb234 {
public:
.....
private:
    TNodo234 * farb;
};
```

1

## 3.4. Árboles 2-3-4

### OPERACIONES BÁSICAS. PROPIEDADES

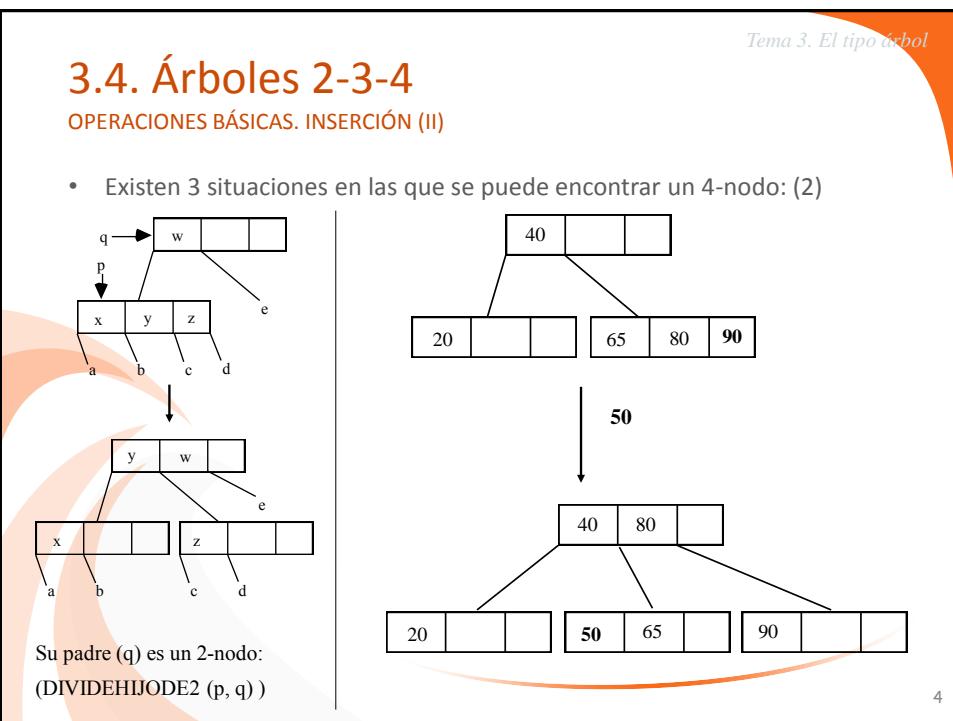
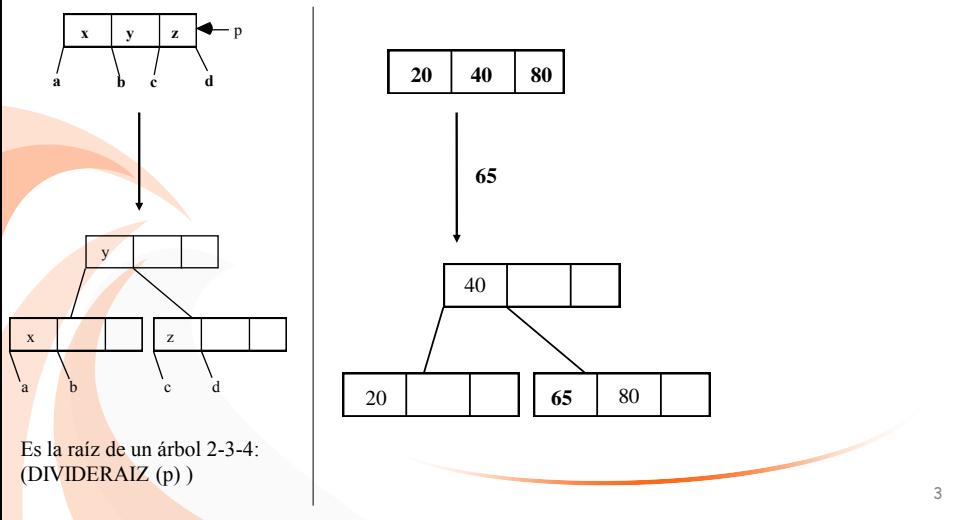
- Operaciones básicas:
  - Búsqueda (similar a los árboles multicamino de búsqueda)
  - Inserción (se realiza en las hojas. Se pueden producir reestructuraciones del árbol)
  - Borrado (se realiza en las hojas. Se pueden producir reestructuraciones del árbol)
- Propiedades:
  - En un árbol 2-3-4 de altura h tenemos:
    - $2^h - 1$  elementos si todos los nodos son del tipo 2-nodo
    - $4^h - 1$  elementos si todos los nodos son del tipo 4-nodo
 por lo que la altura de un árbol 2-3-4 con n elementos se encuentra entre los límites:  $\log_4(n+1)$  y  $\log_2(n+1)$
  - Las reestructuraciones se realizan desde la raíz hacia las hojas

2

## 3.4. Árboles 2-3-4

### OPERACIONES BÁSICAS. INSERCIÓN (I)

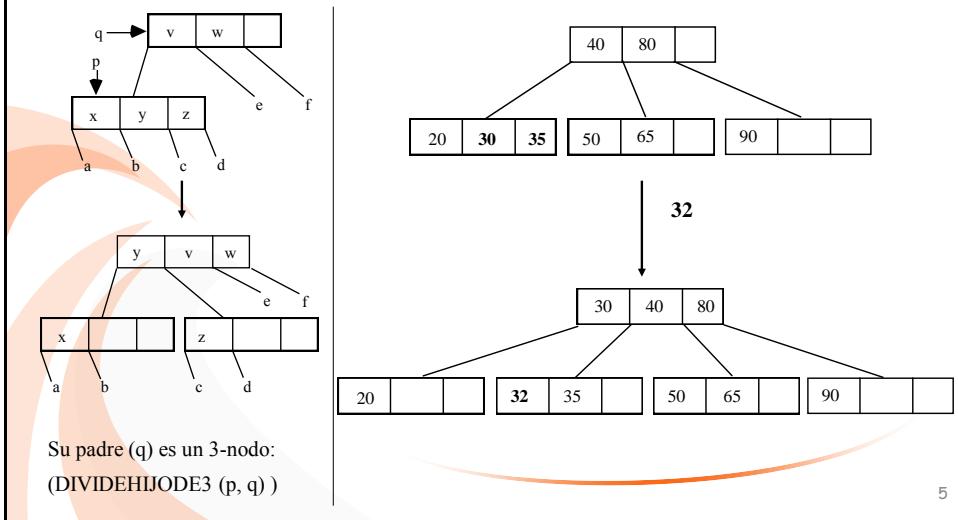
- Existen 3 situaciones en las que se puede encontrar un 4-nodo: (1)



## 3.4. Árboles 2-3-4

### OPERACIONES BÁSICAS. INSERCIÓN (III)

- Existen 3 situaciones en las que se puede encontrar un 4-nodo: (3)



## 3.4. Árboles 2-3-4

### OPERACIONES BÁSICAS. INSERCIÓN (IV)

ALGORITMO insertar (A: TArb234, y: item)

si raíz es 4-nodo → DIVIDERAIZ

si en el camino hasta la hoja me encuentro un 4-nodo → DIVIDEHIJO

## 3.4. Árboles 2-3-4

### OPERACIONES BÁSICAS. INSERCIÓN (V)

```

ALGORITMO insertar (A: TArb234, y: item)
VAR p, q: TNodo234*; noencontrado: Boolean; B: TArb234; FVAR
    p = A.farb; q = p;
    si EsVacio( A ) entonces A = ENRAIZAR(A, y, B)
    sino
        si p es 4-nodo entonces DIVIDERAZ( A ) fsi
        noencontrado = VERDADERO;
        mientras noencontrado hacer
            si p es 4-nodo entonces
                si q es 2-nodo entonces DIVIDEHIJODE2( p, q );
                sino DIVIDEHIJODE3( p, q ); fsi
                p = q;
            fsi
            caso de COMPARAR( y, p ):
                0:// Clave de y coincide con clave en p
                    ERROR, ETIQUETA EXISTENTE;
                1:// p apunta a un nodo hoja
                    PONER( y, p ); noencontrado = FALSO;
                2:// clave( y ) < itemLz.clave( p )
                    q = p; p = p → HiLz;
                3:// itemLz.clave(p)<clave(y)<itemMe.clave(p)
                    q = p; p = p → HiMeLz;
                4://itemMe.clave(p)<clave(y)<itemDe.clave(p)
                    q = p; p = p → HiMeDe;
                5:// clave(y) > itemDe.clave(p)
                    q = p; p = p → HiDe;
            fcaso
        fmientras
    fsi
FALGORITMO

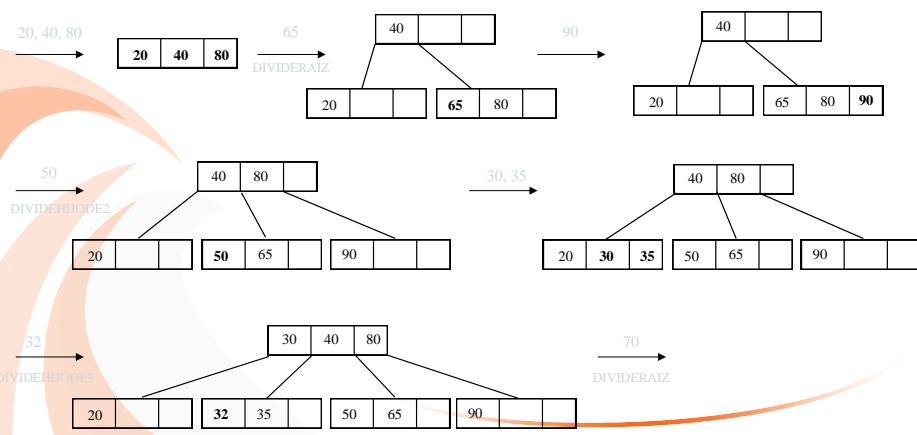
```

7

## 3.4. Árboles 2-3-4

### OPERACIONES BÁSICAS. INSERCIÓN. EJEMPLO (VI)

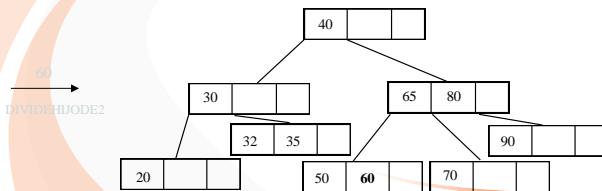
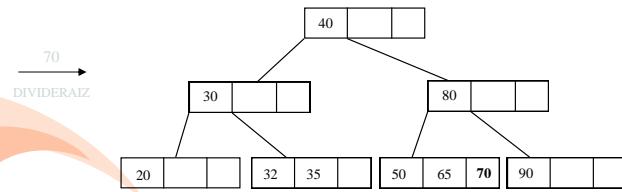
- # **Ejemplo.** Insertar en un árbol 2-3-4 inicialmente vacío los siguientes items: 20, 40, 80, 65, 90, 50, 30, 35, 32, 70, 60



8

## 3.4. Árboles 2-3-4

OPERACIONES BÁSICAS. INSERCIÓN. EJEMPLO (VII)

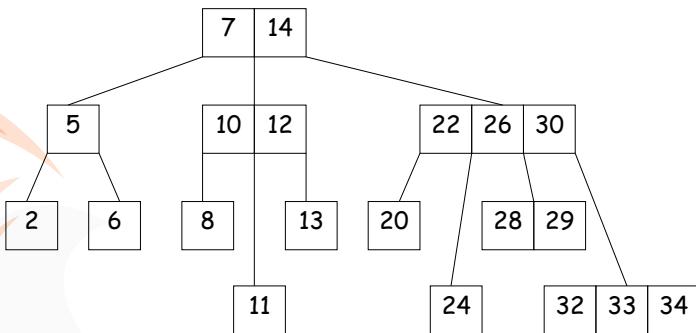


9

## 3.4. Árboles 2-3-4

EJERCICIOS *inserción*

- Dado el siguiente árbol 2-3-4, insertar los elementos 21 y 35



10

## 3.4. Árboles 2-3-4

### OPERACIONES BÁSICAS. BORRADO (I)

- Se reduce al borrado de un elemento en una hoja
- En el movimiento de búsqueda, cuando pasemos a un nodo en el siguiente nivel, éste nodo debe ser 3-nodo ó 4-nodo; si no es así (es 2-nodo) hay que reestructurar
  - $p$  = nodo donde estamos
  - $q$  = siguiente nodo en la búsqueda
  - $r$  = uno de los nodos adyacentes a  $q$  (si hay dos adyacentes, escogemos  $r$  según criterio –hermano de la izquierda o hermano de la derecha–)

11

## 3.4. Árboles 2-3-4

### OPERACIONES BÁSICAS. BORRADO (II)

- Algoritmo de Borrado de 1 elemento
  - Comenzar  $p$  = raíz
  - Escoger  $q$  = siguiente en el camino de búsqueda
  - Reestructuraciones si procede                                    // $q$  es 2-nodo
  - Mientras  $q$  no sea un nodo hoja ó  $q$  sea 2-nodo
    - $p = q$
    - $q$  = siguiente en el camino de búsqueda
    - Reestructuraciones si procede                                    // $q$  es 2-nodo
  - Borrar el elemento    // $q$  está en un nodo hoja y  $q$  no es 2-nodo
- Casos:
  1.  **$p$  es una hoja:**  $p$  sólo puede ser 2-nodo si es la raíz
  2.  **$q$  es 3-nodo ó 4-nodo:** la búsqueda continúa en  $q$  sin reestructurar

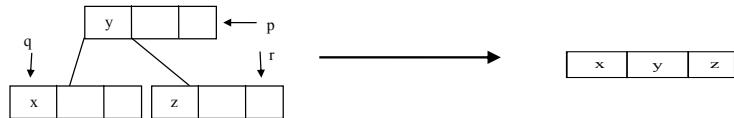
12

## 3.4. Árboles 2-3-4

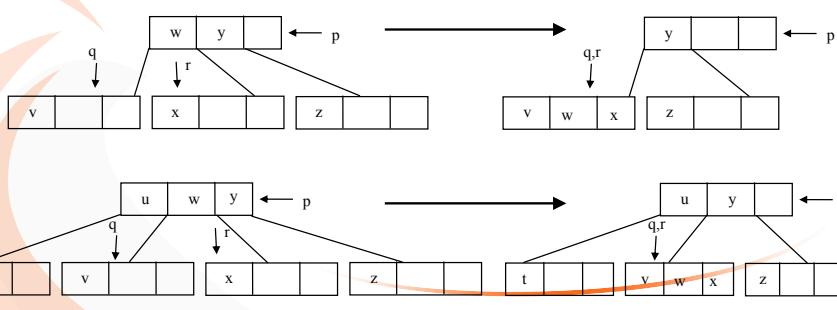
### OPERACIONES BÁSICAS. BORRADO (III)

#### 3. q es 2-nodo y r es 2-nodo (COMBINACIÓN):

1. p es 2-nodo: es la raíz



2. p es 3-nodo ó 4-nodo

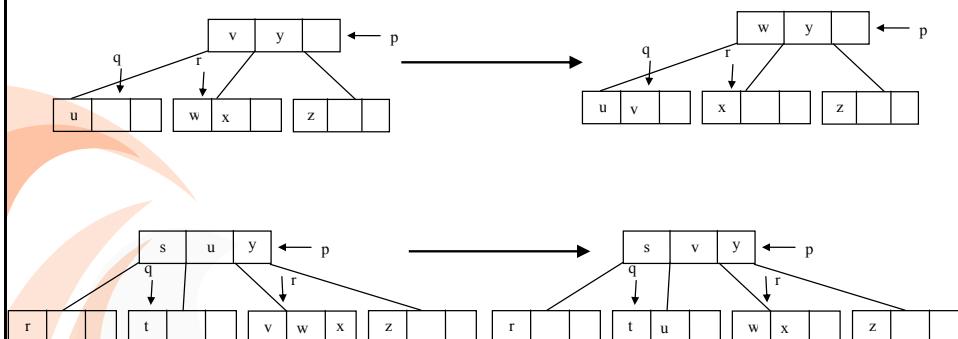


13

## 3.4. Árboles 2-3-4

### OPERACIONES BÁSICAS. BORRADO (IV)

#### 4. q es 2-nodo y r es 3-nodo ó 4-nodo (ROTACIÓN):

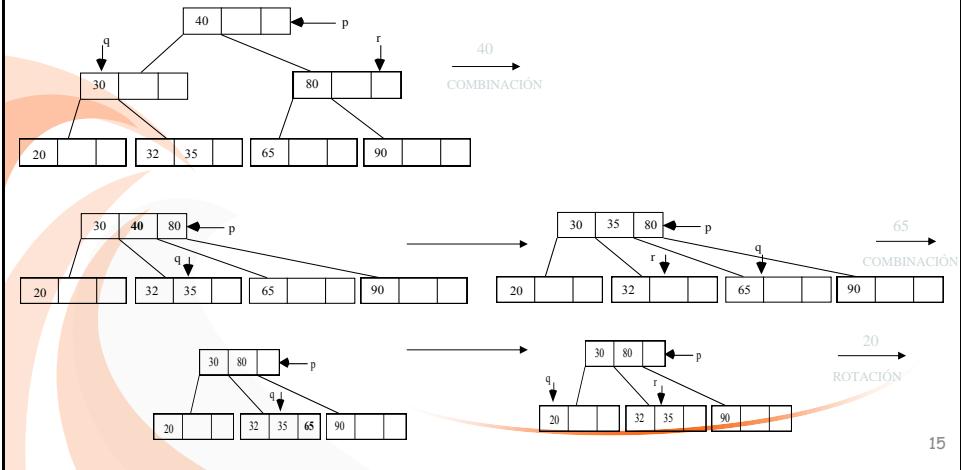


14

## 3.4. Árboles 2-3-4

### OPERACIONES BÁSICAS. BORRADO. EJEMPLO (V)

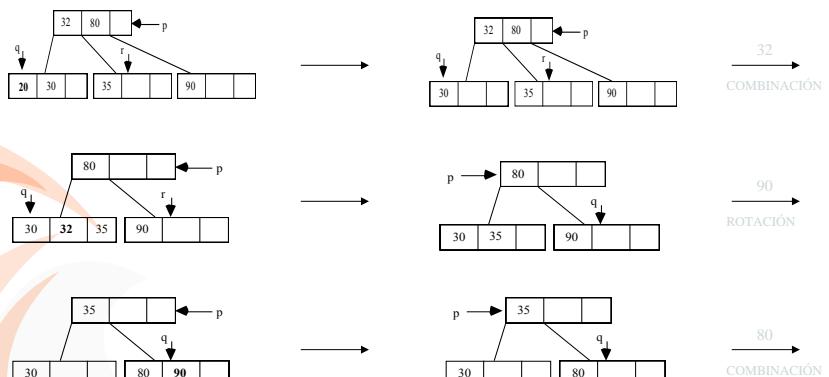
- # **Ejemplo.** Borrar en el siguiente árbol 2-3-4 los siguientes ítems: 40, 65, 20, 32, 90, 80. (Criterios: (1) si el nodo tiene dos hijos hay que sustituir por el mayor de la izquierda, (2) Si hay dos nodos adyacentes a q, entonces r será el hermano de la izquierda)



15

## 3.4. Árboles 2-3-4

### OPERACIONES BÁSICAS. BORRADO. EJEMPLO (VI)

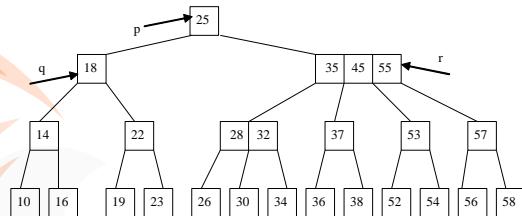


16

## 3.4. Árboles 2-3-4

### EJERCICIOS borrado

- 1) Borrar en el siguiente árbol 2-3-4: 25,18,55 y 35. (Criterios: (1) si el nodo tiene dos hijos hay que sustituir por el mayor de la izquierda, (2) Si hay dos nodos adyacentes a q, entonces r será el hermano de la izquierda)



17

## 3.4. Árboles 2-3-4

Preguntas de tipo test: Verdadero vs. Falso

- El árbol 2-3-4 no vacío tiene como mínimo una clave en cada nodo
- La complejidad temporal en el peor caso de la operación inserción en un árbol 2-3-4 es  $\log_2(n+1)$
- Un árbol 2-3-4 es un árbol binario completo

18

# Árboles de búsqueda

## Aplicaciones

- **Acceso a grandes ficheros de datos. Organización interna de una BD.**
  - Restricción: Los índices residen en disco.
  - Problema: Accesos a disco muy costoso.
  - Solución: Organizar árboles con múltiples claves “n” por nodo
    - Recuperar un nodo de este árbol (un acceso al fichero índice) selecciona una entre “n” alternativas, frente a una entre dos (Árbol Binario).
    - El índice puede diseñarse para que el tamaño de cada nodo coincida con el de un bloque de disco.

## TEMA 4 El tipo conjunto

PROGRAMACIÓN Y ESTRUCTURAS DE DATOS

### Tipo conjunto

- 1. Definiciones generales
- 2. Diccionario
  - 2.1. Tabla de dispersión
- 3. Cola de prioridad
  - 3.1. Montículo
  - 3.2. Cola de prioridad doble
    - 3.2.1. Montículo doble

# 1. Tipo Conjunto

## DEFINICIONES

- Un conjunto es una colección de elementos, cada uno de los cuales puede ser un conjunto, o un elemento primitivo que recibe el nombre de átomo
- Todos los miembros del conjunto son distintos
- El orden de los elementos no es importante (distinto de las listas)

### Notación de Conjuntos

- Se representa encerrando sus miembros entre llaves  $\{1,2,5\}$
- Relación fundamental, la de pertenencia:  $\in \{x / x \in \text{Naturales}\}, \{x / x < 8\}$
- Existe un conjunto especial sin elementos:  $\emptyset$
- $A \subset B$  si todo elemento de A también lo es de B
- Conjunto Universal: formado por todos los posibles elementos que puede contener

3

# 1. Tipo Conjunto

## SINTAXIS

**MODULO GENERICO** ModuloConjunto

**MODULO** Conjunto USA Boolean, Natural

**SINTAXIS**

Crear()  $\rightarrow$  Conjunto  
Insertar(Conjunto, Ítem)  $\rightarrow$  Conjunto  
Eliminar(Conjunto, Ítem)  $\rightarrow$  Conjunto  
Pertenece(Conjunto, Ítem)  $\rightarrow$  Boolean  
EsVacíoConjunto(Conjunto)  $\rightarrow$  Boolean  
Cardinalidad(Conjunto)  $\rightarrow$  Natural  
Unión(Conjunto, Conjunto)  $\rightarrow$  Conjunto  
Intersección(Conjunto, Conjunto)  $\rightarrow$  Conjunto  
Diferencia(Conjunto, Conjunto)  $\rightarrow$  Conjunto

**VAR**

C, D: Conjunto;                    x, y: Ítem;

4

# 1. Tipo Conjunto

## SEMÁNTICA (I)

```

EsVacíoConjunto( Crear )  $\leftrightarrow$  Cierto
EsVacíoConjunto( Insertar( C, x ) )  $\leftrightarrow$  Falso
Insertar( Insertar( C, x ), y )  $\leftrightarrow$ 
    si ( x == y ) entonces Insertar( C, x ) //no se permiten elementos repetidos
    sino Insertar( Insertar( C, y ), x ) //da igual el orden de inserción de los elem.
Eliminar( Crear, x )  $\leftrightarrow$  Crear
Eliminar( Insertar( C, x ), y )  $\leftrightarrow$ 
    si ( x == y ) entonces C
    sino Insertar( Eliminar( C, y ), x )
Pertenece( Crear, x )  $\leftrightarrow$  Falso
Pertenece( Insertar( C, x ), y )  $\leftrightarrow$ 
    si ( x == y ) entonces Cierto
    sino Pertenece( C, y )
Cardinalidad( Crear )  $\leftrightarrow$  0
Cardinalidad(Insertar(C,x))  $\leftrightarrow$  1+Cardinalidad(C)
Unión( Crear, C )  $\leftrightarrow$  C
Unión( Insertar( C,x ), D )  $\leftrightarrow$ 
    si ( Pertenece( D, x ) ) entonces Unión( C, D )
    sino Insertar( Unión( C, D ), x )

```

5

# 1. Tipo Conjunto

## SEMÁNTICA (II)

```

Diferencia( Crear, C )  $\leftrightarrow$  Crear
Diferencia( Insertar( C,x ), D )  $\leftrightarrow$ 
    si ( Pertenece( D, x ) )
    entonces Diferencia( C, D )
    sino Insertar( Diferencia( C, D ), x )
Intersección( Crear, D )  $\leftrightarrow$  Crear
Intersección( Insertar( C,x ), D )  $\leftrightarrow$ 
    si ( Pertenece( D, x ) )
    entonces Insertar( Intersección( C, D ), x )
    sino Intersección( C, D )

```

6

# 1. Tipo Conjunto

## IMPLEMENTACIÓN

# Mediante un vector

-Vector de bits/enteros (cada componente corresponde a un elemento del conjunto universal)

1	0	0	0	1	0
0	1	2	3	4	5

-Vector de elementos

1	9	0	4	2	
---	---	---	---	---	--

# Almacenar los elementos conforme se inserten  
(mediante listas, árboles, ...):

Espacio proporcional al conjunto representado

7

# 1. Tipo Conjunto

## EJERCICIO

# Rellenar la siguiente tabla de complejidades (peor caso):

m=elem. conjunto. n=elem. conjunto. Univ.	Vector de Bits	Lista ordenada	Lista desordenada
Búsqueda			
Inserción			
Unión			

8

## 2. DICCIONARIO

### DEFINICIÓN

- \* **Subtipo del CONJUNTO, con las operaciones:**
  - # CREAR
  - # INSERTAR
  - # BORRAR
  - # BÚSQUEDA

9

## 2. DICCIONARIO

### IMPLEMENTACIÓN

#### Implementaciones sencillas:

- Mediante listas o vectores

#### Búsqueda, Inserción y Borrado:

Listas:	$O(n)$
Vector Bits:	$O(1)$
Vector Elementos:	$O(n)$

- Mediante TAD Tabla de Dispersion (HASHING)

10

## 2.1. TABLA DE DISPERSIÓN (HASHING)

### DEFINICIÓN

\* **HASHING:** Utilizaremos la información del elemento a almacenar para buscar su posición dentro de la estructura

# Operaciones:

- #Búsqueda. O(1)
- #Inserción. O(1)
- #Borrado. O(1)

11

## 2.1. TABLA DE DISPERSIÓN (HASHING)

### MÉTODO

# Dividir el conjunto en un número finito “B” de clases

# Se usa función de dispersión H, tal que  $H(x)$  será un valor entre 0 y  $B-1$

**Formas de dispersión:**

Abierta: No impone tamaño límite al conjunto

Cerrada: usa un tamaño fijo de almacenamiento (limita el tamaño)

12

## 2.1. Tabla Hash. Dispersión Cerrada

### DEFINICIÓN

- # Los elementos se almacenan en tabla de tamaño fijo (**TABLA DE DISPERSIÓN**)
- # La tabla se divide en **B** clases, y cada una podrá almacenar **S** elementos
- # La Función de dispersión se implementa mediante una función aritmética

$$H(x) = x \bmod B$$

13

## 2.1. Tabla Hash. Dispersión Cerrada

### INSERCIÓN

Caso COLISIÓN:  $x_1, x_2$       (SINÓNIMOS/  $H(x_1) = H(x_2)$ )

#### ESTRATEGIA DE REDISPERSION:

- Elegir sucesión de localidades alternas dentro de la tabla, hasta encontrar una vacía  
 $H(x), h_1(x), h_2(x), h_3(x), \dots$
- Si ninguna está vacía: no es posible insertar

14

## 2.1. Tabla Hash. Dispersión Cerrada

### INSERCIÓN. EJEMPLO

**Ejemplo.** Insertar en una tabla de dispersión cerrada de tamaño  $B=7$ , con función de dispersión  $H(x)=x \text{ MOD } B$ , y con estrategia de redispersión la siguiente posición de la tabla, los siguientes elementos: 23, 14, 9, 6, 30, 12, 18, 25

0	14	0	14	0	14	0	14	0	14	0	14	0	14	0	14											
1		1		1		1		1		1		1		1												
2	23	2	23	2	23	2	23	2	23	2	23	2	23	2	23											
3		3	9	3	9	3	9	3	9	3	9	3	9	3	9											
4		4		4		4		4		4		4		4												
5		5		5		5		5		5		5		5												
6		6		6		6		6		6		6		6												
23, 14 un sólo intento			9 dos intentos			6 un sólo intento			30 tres intentos			12 un sólo intento														
Nº TOTAL DE INTENTOS HASTA LA CLAVE 18: <b>14</b>																										
18 cinco intentos																										
25 tabla llena																										

15

## 2.1. Tabla Hash. Dispersión Cerrada

### BÚSQUEDA. BORRADO

#### BÚSQUEDA DE ELEMENTOS

Buscar en sucesión de localidades alternas dentro de la tabla, hasta encontrar una vacía:

$$H(x), h_1(x), h_2(x), h_3(x), \dots$$

#### BORRADO DE ELEMENTOS

Hay que distinguir durante la búsqueda:

- Casillas vacías
- Casillas suprimidas

Durante la inserción las casillas suprimidas se tratarán como espacio disponible.

16

## 2.1. Tabla Hash. Dispersión Cerrada

### ANÁLISIS (I)

#### # ESTRATEGIA DE REDISPERSIÓN LINEAL ("siguiente posición"):

- No eficiente. Larga secuencia de intentos

$$h_i(x) = (H(x) + 1 \cdot i) \text{ MOD } B / \quad c=1 \quad h_i(x) = (h_{i-1}(x) + 1) \text{ MOD } B$$

#### # ESTRATEGIA DE REDISPERSIÓN ALEATORIA:

$$h_i(x) = (H(x) + c \cdot i) \text{ MOD } B / \quad c>1 \quad h_i(x) = (h_{i-1}(x) + c) \text{ MOD } B$$

Sigue produciendo AMONTONAMIENTO: siguiente intento sólo en función del anterior

c y B no deben tener factores primos comunes mayores que 1

#### # E.R. CON 2ª FUNCION DE HASH:

$$k(x) = (x \text{ MOD } (B-1)) + 1 \\ h_i(x) = (H(x) + k(x) \cdot i) \text{ MOD } B \quad h_i(x) = (h_{i-1}(x) + k(x)) \text{ MOD } B$$

B debe ser primo

17

## 2.1. Tabla Hash. Dispersión Cerrada

### ANÁLISIS (II)

#### # LA MEJOR FUNCIÓN DE DISPERSIÓN:

- Que sea fácil de calcular
- Que minimice el nº de colisiones
- Que distribuya los elementos de forma azarosa
- Debe hacer uso de toda la información asociada a las etiquetas

18

## 2.1. Tabla Hash. Dispersión Cerrada

### ANÁLISIS (III)

- Estrategia de redispersión aleatoria

- c y B no deben tener factores primos comunes mayores que 1 para que busque en todas las posiciones de la tabla

- Ejemplo → c=4; B=6

$$h_i(x) = (H(x) + c \cdot i) \text{ MOD } B = (h_{i-1}(x) + c) \text{ MOD } B$$

$$H(10) = 10 \text{ MOD } 6 = 4$$

X	X	X	
0	1	2	3 4 5

$$h_1(10) = (4+4 \cdot 1) \text{ MOD } 6 = (4+4) \text{ MOD } 6 = 2$$

$$h_2(10) = (4+4 \cdot 2) \text{ MOD } 6 = (2+4) \text{ MOD } 6 = 0$$

$$h_3(10) = (0+4) \text{ MOD } 6 = 4; h_4(10) = (4+4) \text{ MOD } 6 = 2$$

- Ejemplo → ¿c=6; B=9? ¿c=2; B=9?

### # Estrategia de redispersión con 2<sup>a</sup> función hash

- B debe ser primo para que busque en todas las posiciones de la tabla

- c=k(x) → 1...B-1 // k(x) = (x MOD (B-1)) + 1

$$h_i(x) = (H(x) + k(x) \cdot i) \text{ MOD } B = (h_{i-1}(x) + k(x)) \text{ MOD } B$$

$$B=7; k(x)=1\dots6$$

$$B=11; k(x)=1\dots10$$

19

## 2.1. Tabla Hash. Dispersión Cerrada

### EJERCICIOS

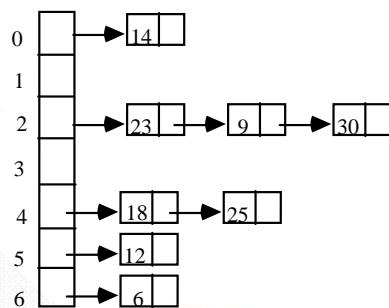
- 1) Insertar en una tabla de dispersión cerrada de tamaño  $B=7$ , con función de dispersión  $H(x) = x \text{ MOD } B$ , y con estrategia de redispersión *segunda función hash*, los siguientes elementos: 23, 14, 9, 6, 30, 12, 18

20

## 2.1. Tabla Hash. Dispersión Abierta

### DEFINICIÓN

- Elimina el problema del CLUSTERING SECUNDARIO (colisiones entre claves no sinónimas)
- Las colisiones se resuelven utilizando una lista enlazada



21

## 2.1. Tabla Hash

### FACTOR DE CARGA ( $\alpha$ )

$$\alpha = \frac{n}{|B|}$$

$n$  = nº elem. de la tabla.  $B$  = tamaño de la tabla

HASH CERRADO:  $0 \leq \alpha \leq 1$

HASH ABIERTO:  $\alpha \geq 0$  (No hay límite en el nº de elementos en cada casilla).

22

## 2.1. Tabla Hash

### FACTOR DE CARGA (II)

E: N° Esperado de Intentos.  
c.éx: con éxito. s.éx: sin éxito.

$\alpha$	H.C.L.		H.C.Aleat.		H.Abrierto	
	E c.éx	E s.éx.	E c.éx	E s.éx.	E c.éx	E s.éx.
0.1	1.06					
0.25	1.17					
0.5	1.50					
0.75	2.50	8.5	1.9	4.0	1.8	2.0
0.9	5.50	50.5	2.6	10.0	1.9	2.0
0.95	10.50					

23

## 2.1. Tabla Hash

### COMPARACIÓN HASH ABIERTO Y CERRADO

- H.A. es más eficiente y con menor degradación (cuanto más lleno funciona mejor que el H.C.)
- H.A. requiere espacio para los elementos de la lista, por lo que H.C. es más eficiente espacialmente
- Reestructuración de las tablas de dispersión:  
 $n \geq 0,9 B$  (H.C.)  
 $n \geq 2 B$  (H.A.)  
 → Nueva tabla con el doble de posiciones

24

### 3. COLA DE PRIORIDAD

#### DEFINICION (I)

- Conjunto de elementos ordenados con las operaciones:  
Crear () → ColaPrioridad  
EsVacio () → Boolean  
Insertar (ColaPrioridad, Item) → ColaPrioridad  
BorrarMínimo (ColaPrioridad) → ColaPrioridad  
BorrarMáximo (ColaPrioridad) → ColaPrioridad  
Búsqueda (ColaPrioridad, Item) → Boolean  
Cardinalidad (ColaPrioridad) → Natural  
Copiar (ColaPrioridad) → ColaPrioridad  
Mínimo (ColaPrioridad) → Item  
Máximo (ColaPrioridad) → Item

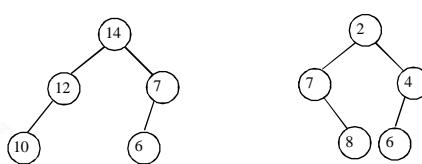
25

### 3. COLA DE PRIORIDAD

#### DEFINICION (II)

- Árbol Mínimo (Máximo):

Árbol en el que la etiqueta de cada nodo es menor (mayor) que la de los hijos.



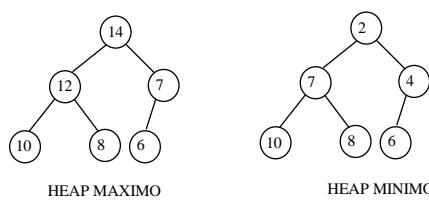
26

### 3. COLA DE PRIORIDAD

#### DEFINICION (III)

- Heap Mínimo (Máximo):

Árbol binario completo en que además es ARBOL MINIMO o MAXIMO.



27

### 3. COLA DE PRIORIDAD

#### DEFINICION (IV)

- Implementación Cola Prioridad:

- LISTA DESORDENADA:

INSERCIÓN:  $O(1)$

BORRADO:  $O(n)$

- LISTA ORDENADA: ascendente o descendente.

INSERCIÓN:  $O(n)$

BORRADO:  $O(1)$

- ARBOL BINARIO DE BUSQUEDA:

INSERCIÓN:  $O(n)$

BORRADO:  $O(n)$

28

### 3. COLA DE PRIORIDAD

#### DEFINICION (V)

- Implementacion Cola Prioridad:

- **HEAP o MONTICULO:**

INSERCIÓN:	$O(\log n)$
BORRADO:	$O(\log n)$

29

### 3.1. HEAP MAXIMO (MINIMO)

#### INSERCIÓN

- METODO:

1.- Insertar en la posición correspondiente para que siga siendo un árbol completo.

2.- Reorganizar para que cumpla las condiciones del HEAP:

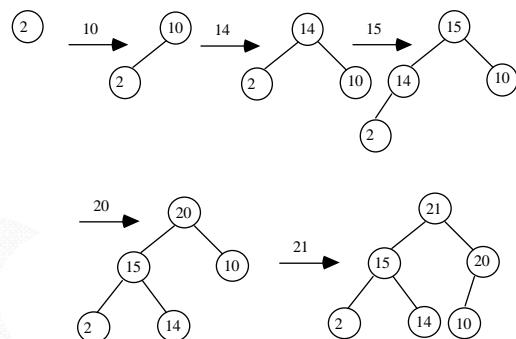
- Comparar con el nodo padre: si no cumple las condiciones del árbol mínimo/máximo, entonces intercambiar ambos.

30

## 3.1. HEAP MAXIMO (MINIMO)

### INSERCIÓN. EJEMPLO

- Insertar: 2, 10, 14, 15, 20 y 21 en un heap máximo inicialmente vacío



31

## 3.1. HEAP MAXIMO (MINIMO)

### BORRADO.

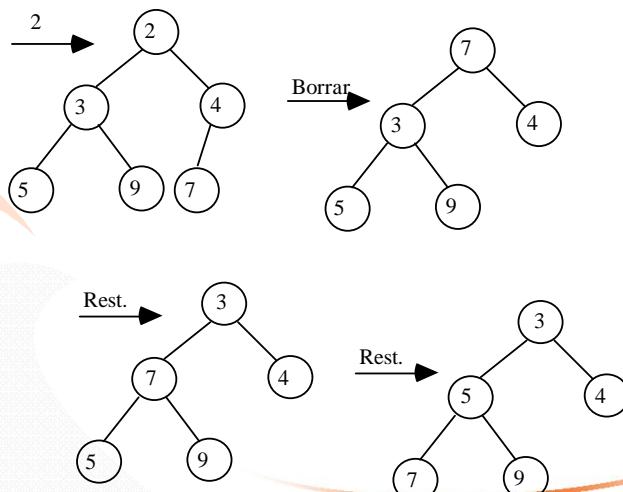
- *METODO:*

- Se sustituye la raíz con el elemento más a la derecha en el nivel de las hojas
- Mientras no sea un HEAP se hunde ese elemento sustituyéndolo con el más pequeño (montículo mínimo) o el mayor (montículo máximo) de sus hijos

32

### 3.1. HEAP MAXIMO (MINIMO)

BORRADO. EJEMPLO

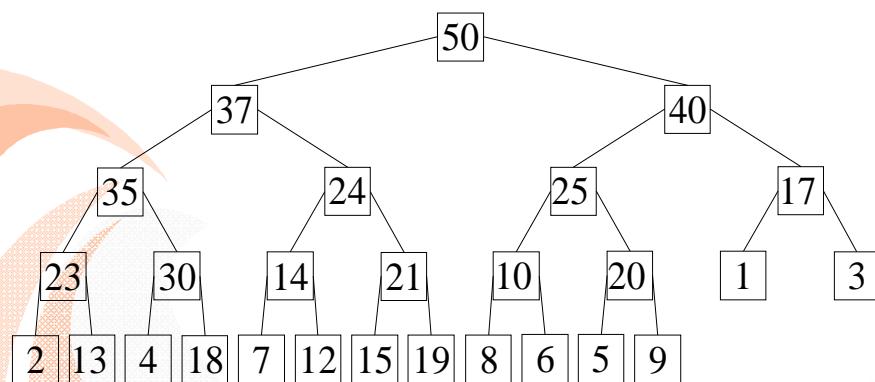


33

### 3.1. HEAP MAXIMO (MINIMO)

BORRADO. EJEMPLO II

- Realiza dos borrados sobre el siguiente montículo máximo.



34

## 3.1. HEAP MAXIMO (MINIMO)

### INSERCIÓN. EJEMPLO II

- Sobre el resultado anterior, realiza las inserciones: 60, 36

35

## 3.1. HEAP MAXIMO (MINIMO)

### REPRESENTACIÓN

- ENLAZADA: problema en inserción al necesitar realizar recorridos ascendentes.
- SECUENCIAL (en un vector):  
Hijos de  $p[i]$  son  $p[2 \cdot i]$  y  $p[2 \cdot i + 1]$ . Padre de  $p[i]$  es  $p[i] \text{ DIV } 2$  con DIV la división entera

36

## 3.1. HEAP MAXIMO (MINIMO)

### APLICACIÓN HEAPSORT

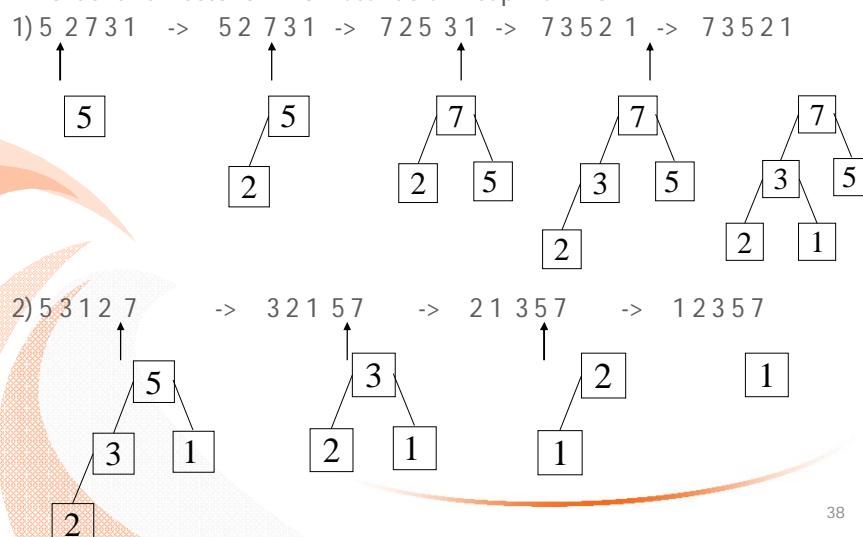
- Algoritmo de ordenación de un vector de elementos
- METODO:
  - 1) Insertar los elementos en un HEAP
  - 2) Realizar borrados de la raíz del HEAP
- IMPLEMENTACIÓN (UN SÓLO VECTOR):
  - 1) Dejar parte izquierda del vector para el HEAP, y parte derecha para los elementos todavía no insertados.
  - 2) Borrar la raíz del HEAP llevándola a la parte derecha del vector.
- COMPLEJIDAD:
  $O(n \log n)$

37

## 3.1. HEAP MAXIMO (MINIMO)

### HEAPSORT. EJERCICIO

- Ordenar el vector 5 2 7 3 1 usando un heap máximo



38

### 3.1. HEAP MAXIMO (MINIMO)

HEAPSORT. EJERCICIO

- Ordenar el vector 9 5 7 4 8 6 2 1 usando un heap mínimo

39

### 3.2. COLA DE PRIORIDAD DOBLE (DEAP)

DEFINICIÓN (I)

- Cola de Prioridad doble: cola de prioridad en la que se soporta la operación de borrado de la clave máxima y mínima.
- DEAP: Es un Heap que soporta las operaciones de cola de prioridad doble.

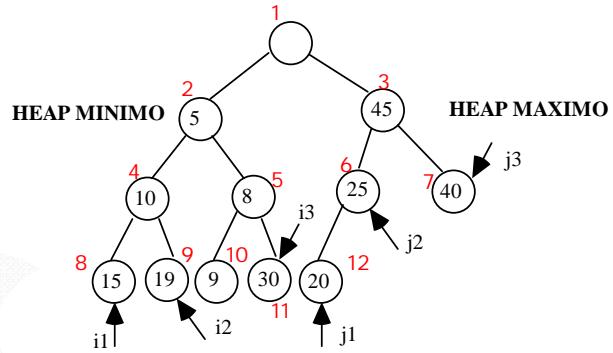
DEFINICION: es un árbol binario completo el cual o es vacío o satisface las siguientes propiedades:

- 1) La raíz no contiene elementos
- 2) El subárbol izquierdo es un HEAP mínimo
- 3) El subárbol derecho es un HEAP máximo
- 4) Si el subárbol derecho no es vacío:
  - Sea "i" cualquier nodo del subárbol izquierdo.
  - Sea "j" el nodo correspondiente en el subárbol derecho. Si "j" no existe, entonces sea el nodo del subárbol derecho que corresponde al padre de "i".
  - Entonces: clave (i) < clave (j)

40

## 3.2. COLA DE PRIORIDAD DOBLE (DEAP)

DEFINICIÓN (II)



41

## 3.2. COLA DE PRIORIDAD DOBLE (DEAP)

IMPLEMENTACIÓN

Igual que en un HEAP, sólo que la primera posición no se utilizará.

$$j = i + 2^{(\log_2 i) - 1} \quad (\text{parte entera})$$

Si  $j > n$  Entonces  $j = j \text{ DIV } 2$

Ejemplo:

simétrico de i1 ( $i=8$ ).  $j = 8 + 2^{(\log_2 8) - 1} = 8 + 2^2 = \underline{\underline{12}}$

simétrico de i2 ( $i=9$ ).  $j = 9 + 2^{(\log_2 9) - 1} = 9 + 2^2 = 13$ . como  $j > n \rightarrow j = 13 \text{ DIV } 2 = \underline{\underline{6}}$

42

## 3.2. COLA DE PRIORIDAD DOBLE (DEAP)

### INSERCIÓN

1) Se inserta el elemento en el siguiente índice del árbol completo

2) Se compara el nodo insertado con el nodo simétrico correspondiente, realizando el intercambio en caso que no se cumpla la condición 4 de la definición del DEAP:

$$i = n - 2^{(\log_2 n) - 1} \quad (\text{ parte entera})$$

3) Actualizar el HEAP mediante el proceso de “ascensión” del elemento insertado.

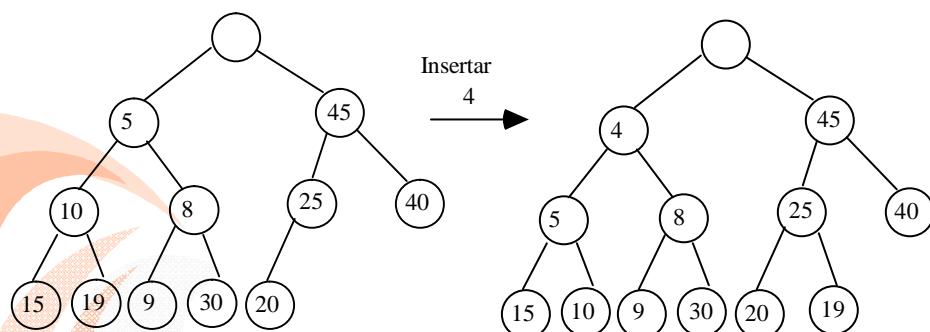
Ejemplo:

simétrico de j1 ( $j=12$ ).  $i = 12 - 2^{(\log_2 12) - 1} = 12 - 2^2 = 8$

43

## 3.2. COLA DE PRIORIDAD DOBLE (DEAP)

### INSERCIÓN



44

## 3.2. COLA DE PRIORIDAD DOBLE (DEAP)

### INSERCIÓN

- # Sobre el DEAP anterior insertar: 35, 7, 50, 17, 12, 27, 55

45

## 3.2. COLA DE PRIORIDAD DOBLE (DEAP)

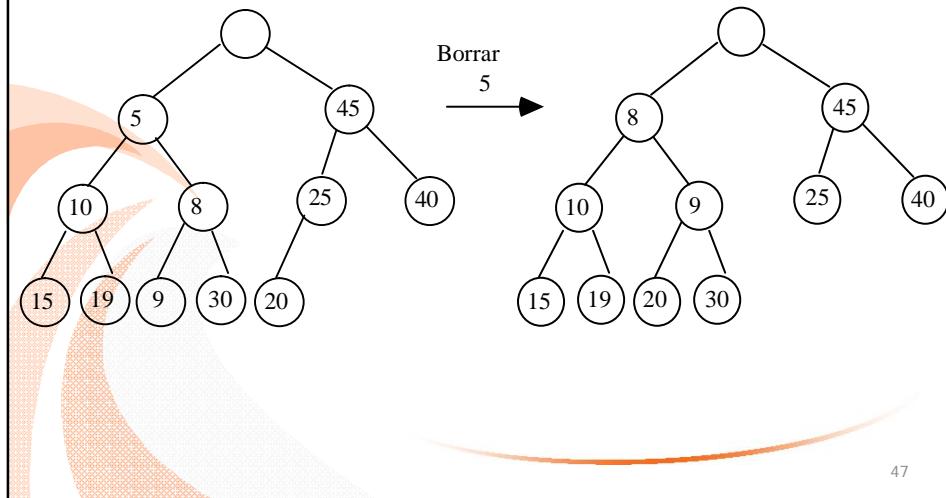
### BORRADO

- 1) Intercambiar la raíz a borrar del HEAP con el elemento más a la derecha del último nivel del árbol, y borrar éste.
- 2) Actualizar el HEAP, “hundiendo” la clave intercambiada.
- 3) Comprobar que la clave intercambiada no incumpla la condición del DEAP con su correspondiente nodo simétrico
- 4) Actualizar el montículo en el que quede la clave intercambiada

46

## 3.2. COLA DE PRIORIDAD DOBLE (DEAP)

BORRADO

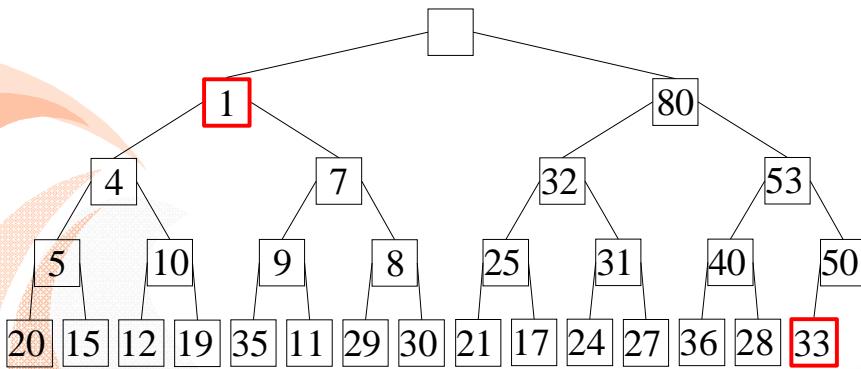


47

## 3.2. COLA DE PRIORIDAD DOBLE (DEAP)

### BORRADO

MONTÍCULO DOBLE: Borrar los elementos mínimo y máximo de forma sucesiva.



48

## TEMA 5 El tipo grafo

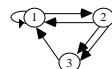
PROGRAMACIÓN Y ESTRUCTURAS DE DATOS

### Tipo grafo

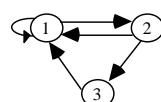
- 1. Concepto de grafo y terminología
- 2. Especificación algebraica
- 3. Representación de grafos
- 4. Grafos dirigidos
  - 4.1. Recorrido en profundidad o DFS
  - 4.2. Recorrido en anchura o BFS
  - 4.3. Grafos acíclicos dirigidos o GAD
  - 4.4. Componentes fuertemente conexos
- 5. Grafos no dirigidos
  - 5.1. Algoritmos de recorrido

## 1. Concepto de grafo y terminología (I)

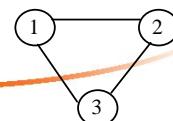
- Estructura de los grafos
  - cada nodo puede tener más de un sucesor y más de un predecesor
- Clasificación
  - MULTIGRAFO, no tiene ninguna restricción (arcos reflexivos y múltiples ocurrencias del mismo arco)



- DIGRAFO, no hay múltiples ocurrencias del mismo arco



- GRAFO NO DIRIGIDO, hay relaciones irreflexivas y simétricas entre nodos (todos sus arcos no orientados → aristas)



3

## 1. Concepto de grafo y terminología (II)

- Definición de grafo
  - Un **grafo G** consiste en dos conjuntos V y A donde:  $G=(V, A)$ 
    - 1) V es un conjunto finito no vacío de vértices o nodos.
    - 2) A es un conjunto de aristas o arcos, tales que cada arista  $a_j$  de A está identificada por un único par  $(v_j, v_k)$  de nodos de V, denotada por  $a_j = (v_j, v_k)$
- Un Arco Orientado o ARCO es aquél en que el orden de los vértices es importante:  $\langle v_j, v_k \rangle$
- Arco No Orientado o ARISTA: orden de los vértices no importa. Relaciones simétricas:  $(v_j, v_k)$
- El **número máximo de aristas o arcos** que pueden existir en un grafo de  $n$  vértices sin contar las aristas que unen un vértice consigo mismo  $(v_i, v_i)$  son:

Grafo no dirigido:

$$n(n-1)/2$$

Grafo dirigido:

$$n(n-1)$$

4

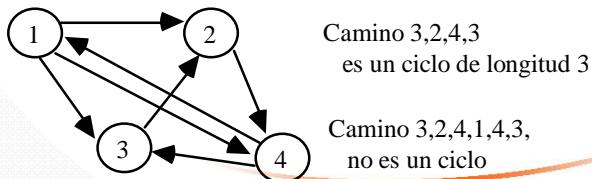
## 1. Concepto de grafo y terminología (III)

- Los vértices  $v_1$  y  $v_2$  son adyacentes si  $(v_1, v_2)$  es una arista en  $A(G)$ , la cual diremos que es **incidente sobre** ambos vértices.
  - En el caso de grafos dirigidos  $\langle v_1, v_2 \rangle$  será **incidente a**  $v_1$  y  $v_2$ . Además diremos que  $v_1$  es **adyacente hacia**  $v_2$ , y  $v_2$  es **adyacente desde**  $v_1$ .
  - Adyacencia de un vértice: es el conjunto de nodos del grafo tales que existe un arco que los relacione:
- $$Ay(x) = \{v_i \in V / \exists (x, v_i) \in A\}$$
- Para los digrafos podemos hablar de:
    - Adyacencia de entrada**, es el conjunto de nodos para los que hay una arista en el grafo que relaciona a ambos, con destino el vértice  $x$ :
$$AyE(x) = \{v_i \in V / \exists \langle v_i, x \rangle \in A\}$$
    - Adyacencia de salida** es:
$$AyS(x) = \{v_i \in V / \exists \langle x, v_i \rangle \in A\}$$
  - Grado de entrada o ingrado (gradoE)**: cardinal del conjunto *Adyacencia de Entrada*.
  - Grado de salida (gradoS)**: cardinal del conjunto *Adyacencia de Salida*.
  - Grado de un Vértice**:  $grado(v) = \text{CARD}(Ay(v)) = \text{gradoE} + \text{gradoS}$

5

## 1. Concepto de grafo y terminología (IV)

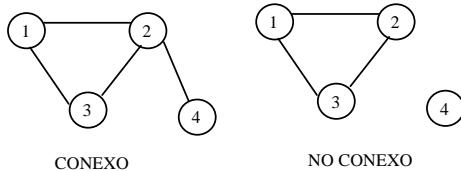
- Un **camino desde  $v_p$  hasta  $v_q$**  en el grafo  $G$  es una secuencia de vértices  $v_p, v_1, v_2, \dots, v_n, v_q$  tal que  $(v_p, v_1), (v_1, v_2), (v_2, v_3), \dots, (v_n, v_q)$  son aristas de  $A(G)$ .
- Se llama **longitud de un camino** al número de aristas del mismo.
- Se llama **camino simple** a aquél en el que todos los vértices, excepto el primero y el último, son distintos.
- Un **ciclo** es un camino simple en el que el vértice primero y último coinciden.



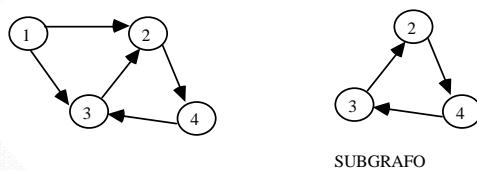
6

## 1. Concepto de grafo y terminología (V)

- Un **grafo no dirigido** se dice que es **conexo**: si  $\forall vi, vj \in V(G)$  existe un camino de  $v_i$  a  $v_j$  en  $G$



- Un **subgrafo** de un grafo  $G=(V, A)$ , es un grafo  $G'=(V', A')$  tal que  $V' \subset V$  y  $A' \subset A$



- Un **árbol extendido** de un grafo  $G=(V, A)$  es un subgrafo  $T=(V', A')$  de  $G$  tal que  $T$  es un árbol y  $V'=V$

7

## 2. Especificación algebraica (I)

**MODULO GENERICO** ModuloGrafo

**MODULO** Grafo **USA** Boolean

**PARAMETRO**

**TIPO** Vértice, Ítem

**SINTAXIS**

ErrorÍtem() → Ítem

**TIPO**

Grafo

**SINTAXIS**

Crear() → Grafo

EsVacioGrafo(Grafo) → Boolean

InsertarArista(Grafo, Vértice, Vértice, Ítem) → Grafo

RecuperarArista(Grafo, Vértice, Vértice) → Ítem

BorrarArista(Grafo, Vértice, Vértice) → Grafo

**VAR** G: Grafo; x, y, z, t: Vértice;

p, q: Ítem;

8

## 2. Especificación algebraica (II)

### SEMANTICA

```

EsVacioGrafo(Crear)           ⇔      Cierto
EsVacioGrafo(InsertarArista(G, x, y, p)) ⇔ Falso
BorrarArista(Crear, z, t) ⇔  Crear
BorrarArista(InsertarArista(G,x,y,p), z, t) ⇔
    si (x == z) y (y == t)
    entonces G //Multigrafo: BorrarArista(G, z, t)
    sino InsertarArista(BorrarArista(G, z, t), x, y, p)
RecuperarArista(Crear, x, y) ⇔ ErrorÍtem
RecuperarArista(InsertarArista(G,x,y,p), z, t) ⇔
    si (x == z) y (y == t)
    entonces p
    sino RecuperarArista(G, z, t)
InsertarArista (InsertarArista (G,x,y,p), z,t,q) ⇔
    si (x == z) y (y == t)
    entonces InsertarArista(G,z,t,q)
    sino InsertarArista(InsertarArista(G,z,t,q), x, y, p)

```

FIN MODULO

9

## 3. Representación de grafos (I)

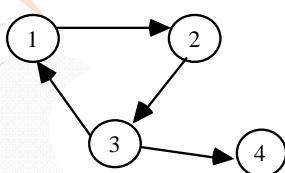
- Dado un grafo  $G$  con  $n$  vértices,  $n \geq 1$ , la **matriz de adyacencia de  $G$**  es una matriz  $A$  cuadrada bidimensional  $n \times n$  tal que:

$$\begin{array}{ll}
 A(i, j) = 1 \text{ ó CIERTO} & \text{si existe } (v_i, v_j) \\
 A(i, j) = 0 \text{ ó FALSO} & \text{si no existe } (v_i, v_j)
 \end{array}$$

### MATRIZ

### ADYACENCIA

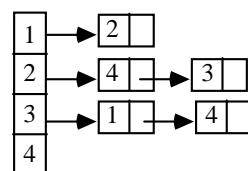
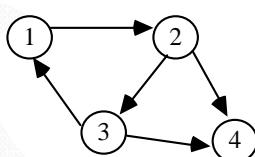
$$\begin{bmatrix}
 0 & 1 & 0 & 0 \\
 0 & 0 & 1 & 0 \\
 1 & 0 & 0 & 1 \\
 0 & 0 & 0 & 0
 \end{bmatrix}$$



10

### 3. Representación de grafos (II)

- En el caso que representásemos un **grafo no dirigido** podríamos observar cómo la matriz de adyacencia sería simétrica respecto la diagonal ante la equivalencia de cada arista con sus dos arcos. Además, la diagonal principal tendrá todos sus valores a cero, a causa de no permitirse relaciones reflexivas entre nodos.
- Una segunda posibilidad de representación es la **lista de adyacencia** donde se cambia la matriz por una lista para cada vértice  $v_i$  en  $G$ , y en la que los nodos de cada lista contienen los vértices que son adyacentes desde el vértice  $v_i$ .



LISTA DE ADYACENCIA

11

### 4. Grafos dirigidos

#### RECORRIDO EN PROFUNDIDAD O DFS (I)

- Generalización del preorden
- Recorre todos los vértices que sean accesibles desde v
- Cuando se hayan visitado todos se continúa desde otro vértice que no haya sido visitado

#### ALGORITMO DFS

ENTRADA:  $v$  : Vértice ;  $G$  : Grafo ;  
ENTRADA-SALIDA: Visitados: conjunto vertices;

VAR :  $w$ : Vértice;

METODO

INSERTAR (visitados, v)

VISITAR (v)

Para todo  $w \in AyS(v)$ Si NO PERTENECE ( $w$ , visitados)DFS ( $w$ ,  $G$ , visitados)

fsi

fpara

fMETODO

12

## 4. Grafos dirigidos

RECORRIDO EN PROFUNDIDAD O DFS (II)

- Tipos de arcos:
- **DEL ARBOL**: los que conducen a nodos no visitados, y forman un **BOSQUE EXTENDIDO EN PROFUNDIDAD**.
- **DE RETROCESO**: los que van desde un vértice a uno de sus antecesores en el bosque extendido
- **DE AVANCE**: los que van de un vértice a un descendiente propio y no son arcos DEL ARBOL
- **DE CRUCE**: los que van de un vértice a otro que no es un antecesor ni descendiente

13

## 4. Grafos dirigidos

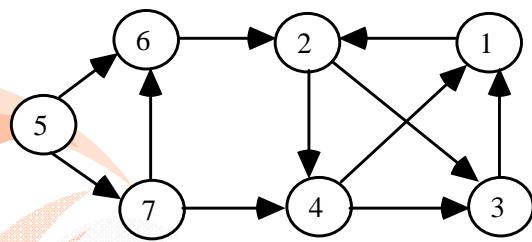
COMPONENTES FUERTEMENTE CONEXOS

- **Prueba de Aciclidad**: Si se encuentra un arco de retroceso durante la búsqueda en profundidad el grafo no será acíclico.
- **Componente Fuertemente conexo de un grafo dirigido**: Conjunto maximal de vértices en el cual existe un camino que va desde cualquier vértice del conjunto hasta cualquier otro vértice también del conjunto.
- **GRAFO DIRIGIDO FUERTEMENTE CONEXO**: aquel que tiene un sólo componente fuerte.

14

## 4. Grafos dirigidos

RECORRIDO EN PROFUNDIDAD O DFS (III)



(Lista de adyacencia de menor a mayor)

Recorridos DFS:

1,2,3,4  
2,3,1,4  
3,1,2,4  
4,1,2,3  
5,6,2,3,1,4,7

15

## 4. Grafos dirigidos

EJERCICIO RECORRIDO EN PROFUNDIDAD O DFS (IV)

- Dado el grafo anterior realizar el recorrido DFS partiendo del vértice 5 (suponed la lista de adyacencia ordenada de mayor a menor). Clasificar los arcos y dibujar el bosque extendido en profundidad

16

## 4. Grafos dirigidos

### EJERCICIO 1. RECORRIDO EN PROFUNDIDAD O DFS

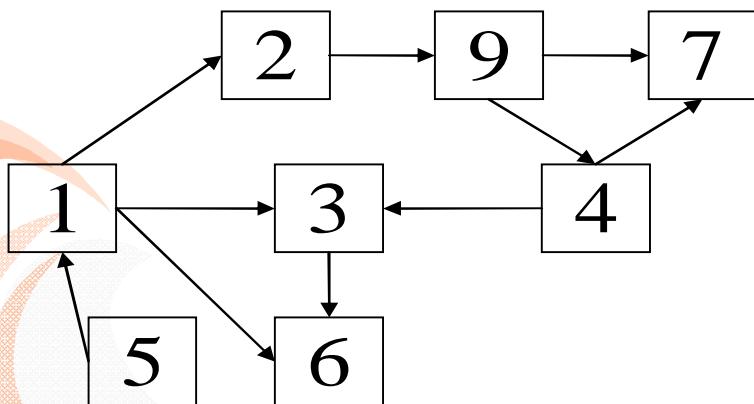
Sobre el siguiente grafo dirigido, responder:

- A) Realiza el recorrido DFS(1), la adyacencia de salida ordenada de menor a mayor, y obtén el árbol extendido en profundidad.
- B) Etiqueta los arcos
- C) ¿Este grafo tiene ciclos? ¿Por qué?
- D) ¿Es un grafo fuertemente conexo? Justifica tu respuesta.

17

## 4. Grafos dirigidos

### EJERCICIO 1. RECORRIDO EN PROFUNDIDAD O DFS



18

## 4. Grafos dirigidos

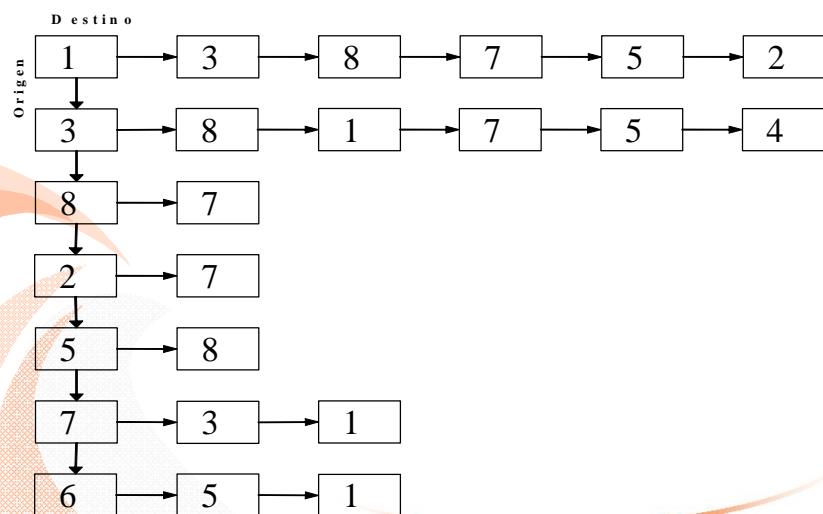
### EJERCICIO 2. RECORRIDO EN PROFUNDIDAD O DFS

- Dado el grafo dirigido representado por la lista de listas que se muestra a continuación, calcular el árbol extendido en profundidad que parte del vértice 1. Para recorrer las listas de adyacencia, seguir el orden izquierda a derecha de cada lista.

19

## 4. Grafos dirigidos

### EJERCICIO 2. RECORRIDO EN PROFUNDIDAD O DFS



20

## 4. Grafos dirigidos

### RECORRIDO EN ANCHURA O BFS (I)

- Primero visita todos los adyacentes a  $v_i$  y después los adyacentes a los visitados

ALGORITMO **BFS**

ENTRADA  $v: TVértice; G: TGrafo;$

VAR  $w_1, w_2: TVértice; visitados: TConjunto(TVértice); Q: TCola(TVértice);$

METODO

    INSERTAR(visitados, v);

    ENCOLAR(Q, v);

    VISITAR(v);

    mientras no EsVacia(Q) hacer

$w_1 = CABEZA(Q); DESENCOLAR(Q);$

        para todo  $w_2 \in L(w_1)$  hacer // Lista adyacencia

            si no PERTENECE( $w_2, visitados$ ) entonces

                VISITAR( $w_2$ );

                ENCOLAR(Q,  $w_2$ );

                INSERTAR( $w_2$ , visitados);

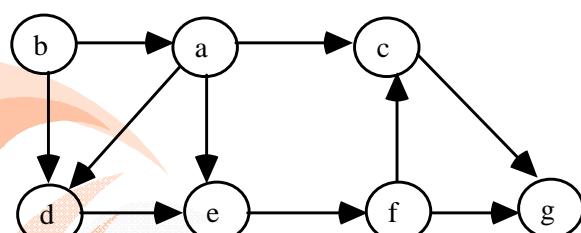
    fsi

fpara  
fmiéntas  
fMETODO

21

## 4. Grafos dirigidos

### RECORRIDO EN ANCHURA O BFS (II)



Recorridos BFS:

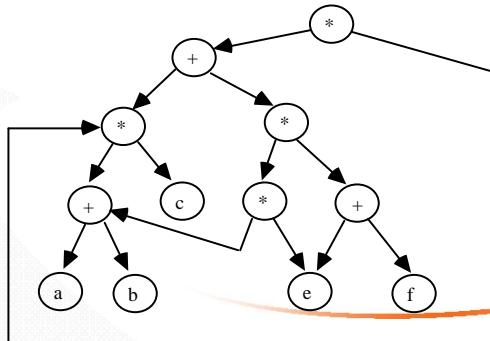
a,c,d,e,g,f  
b,a,d,c,e,g,f

22

## 4. Grafos dirigidos

### GRAFOS ACÍCLICOS DIRIGIDOS O GAD (I)

- Dígrafo que no tiene ciclos.
- Son más generales que los árboles, pero menos que los grafos dirigidos arbitrarios.
- Son útiles para representar la estructura sintáctica de expresiones aritméticas con subexpresiones comunes.
- Ejemplo:  $((a+b) * c + ((a+b) * e) * (e+f))^* ((a+b) * c)$



23

## 4. Grafos dirigidos

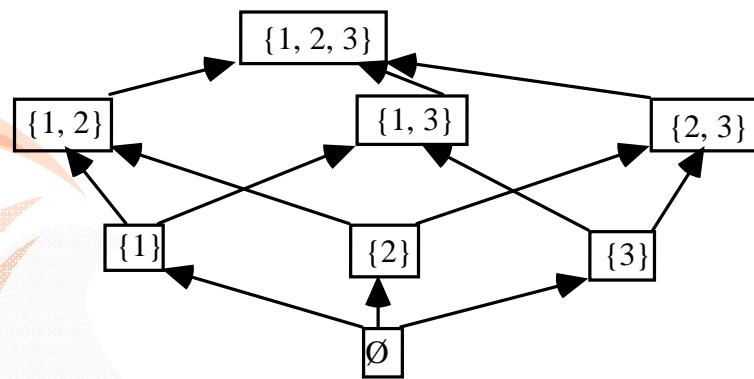
### GRAFOS ACÍCLICOS DIRIGIDOS O GAD (II)

- Otra aplicación de los GAD consiste en la representación de órdenes parciales:
- Un **orden parcial R en un conjunto S** es una relación binaria tal que:
  - $\forall a \in S, a R a$  es FALSO ( $R$  es irreflexiva)
  - $\forall a, b, c \in S, \text{ si } a R b \text{ y } b R c \text{ entonces } a R c$  ( $R$  es transitiva)
- **Ejemplo.** Sea el conjunto  $S=\{1, 2, 3\}$ , y sea el conjunto de todos los subconjuntos de  $S$ :
 
$$P(S) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$$
 La relación  $\subset$  es un orden parcial en  $P(S)$ , ya que cumple:
  - $\forall A \in P(S), A \subset A$  es FALSO (irreflexividad)
  - Si  $A \subset B$  y  $B \subset C \rightarrow A \subset C$  (transitividad)

24

## 4. Grafos dirigidos

GRAFOS ACÍCLICOS DIRIGIDOS O GAD (III)



25

## 5. Grafos no dirigidos

ALGORITMOS DE RECORRIDO

- **EN PROFUNDIDAD (DFS):**
- Se puede usar el mismo algoritmo que en grafos dirigidos
- TIPOS DE ARCOS:
  - Del Árbol
  - De Retroceso (= avance. No existen de cruce)
  
- **EN AMPLITUD (BFS):**
- Igual que para digrafos
- Se puede construir un bosque de extensión:
  - (x,y) será del árbol si "y" (no se había visitado anteriormente) se visitó primero partiendo de "x" del ciclo interno de BFS.
  - Toda arista que no es del árbol es una arista de retroceso (conecta dos vértices ninguno de los cuales es antecesor del otro).

26