

Programación II

Tema 5. Árboles binarios

Iván Cantador

Escuela Politécnica Superior
Universidad Autónoma de Madrid

Contenidos

1

- Grafos y árboles
- Árboles binarios
 - Definición
 - Recorrido
 - Completitud
 - Implementación en C
- Árboles binarios de búsqueda
 - Definición
 - Construcción de un árbol e inserción y búsqueda de un elemento
 - Extracción de un elemento
 - Equilibrado
- Árboles de expresión
 - Definición y recorrido
 - Construcción

Contenidos

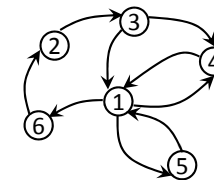
2

- **Grafos y árboles**
- Árboles binarios
 - Definición
 - Recorrido
 - Completitud
 - Implementación en C
- Árboles binarios de búsqueda
 - Definición
 - Construcción de un árbol e inserción y búsqueda de un elemento
 - Extracción de un elemento
 - Equilibrado
- Árboles de expresión
 - Definición y recorrido
 - Construcción

Grafos. Definición

3

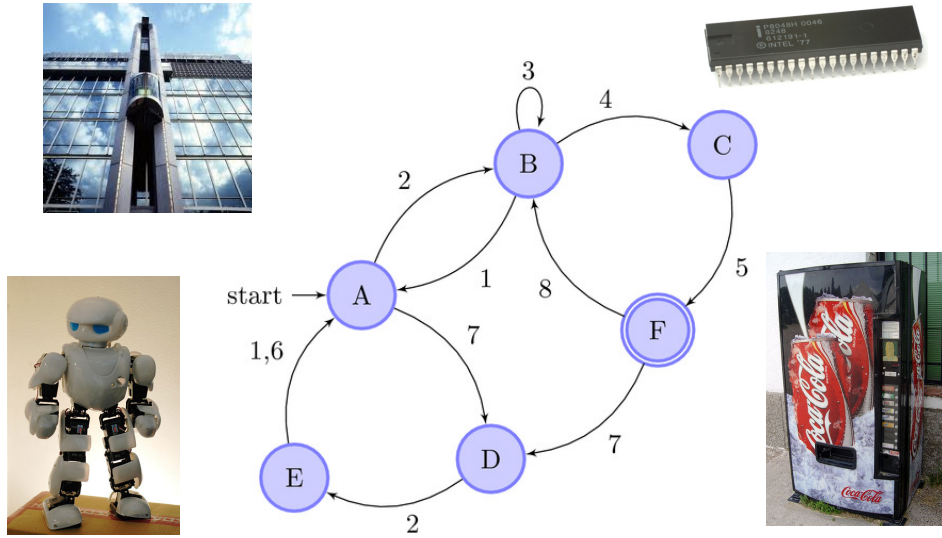
- Un **grafo** es una estructura de datos $G = (V, R)$ compuesta de:
 - Un conjunto V de **vértices** (nodos)
 - Un conjunto R de **ramas** (arcos), conexiones entre los vértices de V
- Ejemplo de grafo (*dirigido*)



- $V = \{1, 2, 3, 4, 5, 6\}$
- $R = \{(1,4), (1,5), (1,6), (2,3), \dots, (5,1), (6,2)\}$
- Un grafo es una EdD general, muy rica y flexible

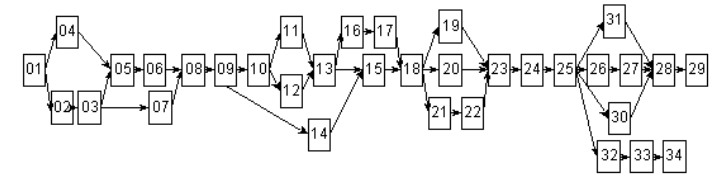
Grafos. Ejemplos (I)

- Máquinas de estados (autómatas finitos deterministas)

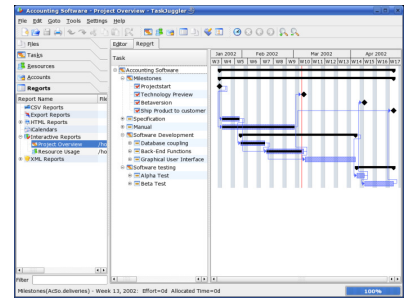


Grafos. Ejemplos (II)

- Planificación de tareas



Task nr	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34
Duration	3	4	4	6	5	2	3	4	3	2	3	2	1	5	2	3	2	2	1	1	1	2	4	5	2	1	1	2	1	3	2	1	2	2
Resource Use	4	4	3	4	5	5	4	3	4	8	4	5	4	3	3	3	6	7	4	4	4	4	7	8	8	3	3	6	8	3	3	3	3	3



Grafos. Ejemplos (III)

- Redes de transporte

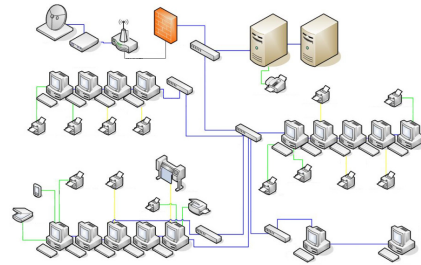
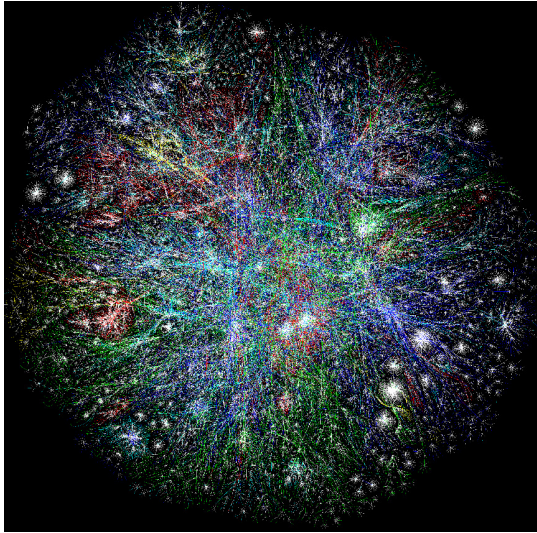


Grafos. Ejemplos (IV)

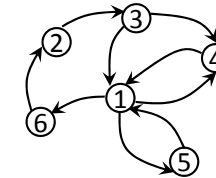
- Redes sociales



- Internet – redes de ordenadores

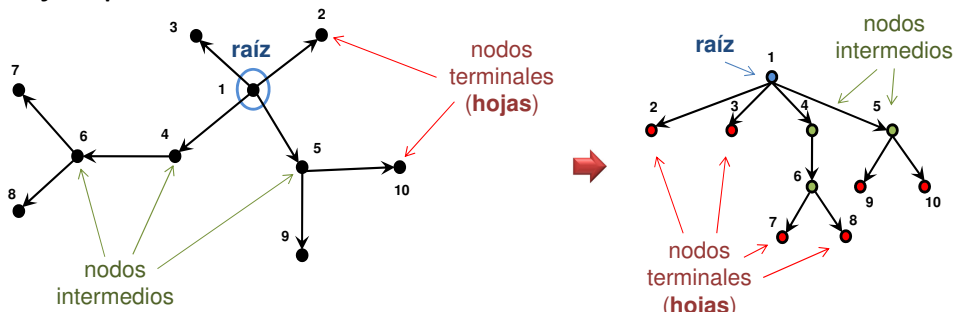


- Un **camino** de un grafo $G = (V, R)$ es una secuencia de nodos de V en los que cada nodo es adyacente al siguiente mediante un arco de R
- Ejemplo

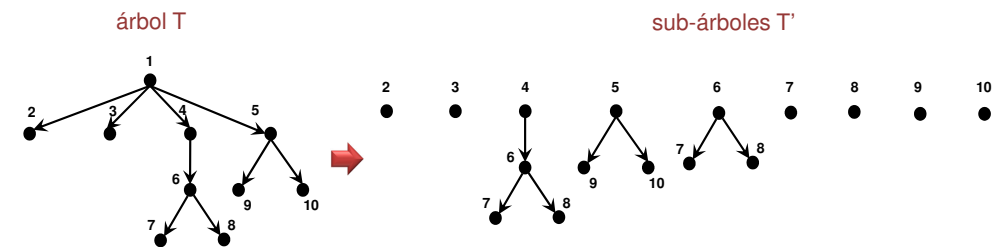


- $V = \{1, 2, 3, 4, 5, 6\}$
- $R = \{(1,4), (1,5), (1,6), (2,3), \dots, (5,1), (6,2)\}$
- **Caminos:** $\{1, 6, 2, 3\}, \{5, 1, 4\}, \dots$

- Un **árbol** ordenado con raíz es un grafo tal que:
 - tiene un único nodo, denominado **raíz**, sin ramas incidentes
 - cada nodo \neq raíz recibe una sola rama
 - cualquier nodo es accesible desde la raíz
- Ejemplo

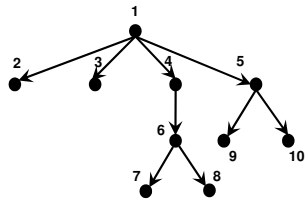


- Un **sub-árbol** de un árbol T es un subconjunto de nodos de T conectados mediante ramas de T
- Cada nodo de un árbol T junto con sus hijos da lugar a un nuevo sub-árbol T'



- En un árbol $A=(V, R)$:
 - Un nodo $u \in V$ es **padre** de otro nodo $v \in V$ si existe un arco $r = (u, v) \in R$
 - Un nodo $v \in V$ es **hijo** de otro nodo $u \in V$ si existe un arco $r = (u, v) \in R$
- Ejemplo

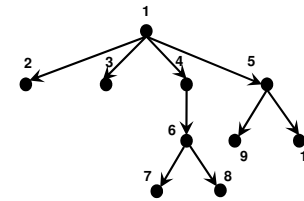
$T \equiv$



6 es padre de 7 y 8
 3 es hijo de 1
 1 (la raíz) no tiene padre
 2 (una hoja) no tiene hijos

- La **profundidad (nivel) de un nodo** es el número de ramas entre el nodo y la raíz (es 0 para la raíz)
- La **profundidad (altura) de un árbol** es el máximo número de ramas entre la raíz una hoja del árbol (es -1 si el árbol está vacío, 0 para un árbol con un nodo)
- Ejemplo

$T \equiv$

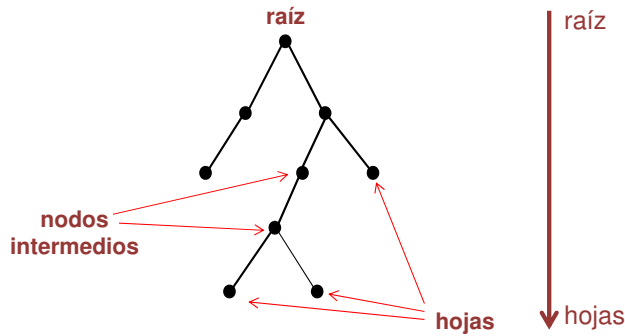


profundidad(T) = 3
 profundidad(1) = 0
 profundidad(5) = 1
 profundidad(7) = 3

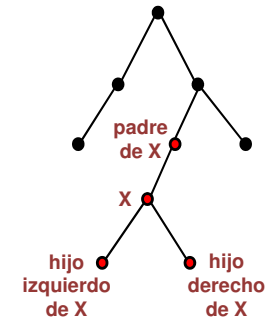
- Profundidad de un árbol**
 - $T \equiv \emptyset \rightarrow$ profundidad(T)= -1
 - $T \equiv 25 \rightarrow$ profundidad(T)= 0
 - $T \equiv 25 \rightarrow$ profundidad(T)= 1
 - $T \equiv$ profundidad(T)= 3
 (la mayor profundidad de todas las hojas)

- Grafos y árboles
- Árboles binarios**
 - Definición**
 - Recorrido
 - Compleitud
 - Implementación en C
- Árboles binarios de búsqueda
 - Definición
 - Construcción de un árbol e inserción y búsqueda de un elemento
 - Extracción de un elemento
 - Equilibrado
- Árboles de expresión
 - Definición y recorrido
 - Construcción

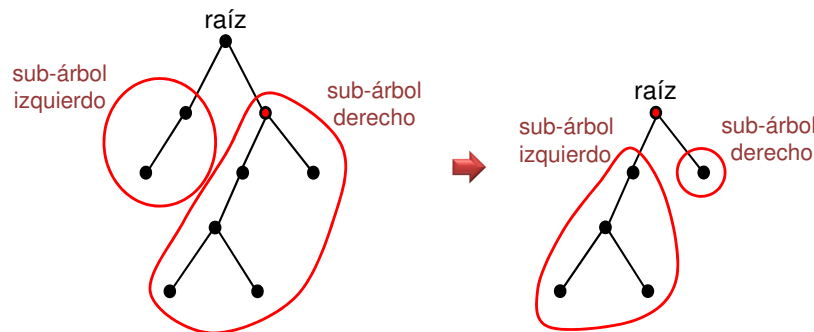
- Un **árbol binario** (AB) es un árbol ordenado con raíz tal que:
 - cada nodo tiene a lo sumo 2 hijos
- Ejemplo



- En un AB:
 - Todo nodo excepto el raíz tiene un **nodo padre**
 - Todo nodo tiene a lo sumo 2 **nodos hijos**: hijo izquierdo e hijo derecho



- Propiedad recursiva de los AB
 - El hijo izquierdo de la raíz (u otro nodo) forma un nuevo árbol con dicho hijo como raíz
 - El hijo derecho de la raíz (u otro nodo) forma un nuevo árbol con dicho hijo como raíz

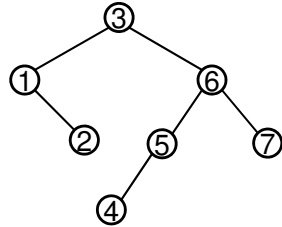


- Grafos y árboles
- **Árboles binarios**
 - Definición
 - **Recorrido**
 - Completitud
 - Implementación en C
- Árboles binarios de búsqueda
 - Definición
 - Construcción de un árbol e inserción y búsqueda de un elemento
 - Extracción de un elemento
 - Equilibrado
- Árboles de expresión
 - Definición y recorrido
 - Construcción

Árboles binarios. Recorrido

20

- Un árbol se puede recorrer de distintas formas, pero siempre **desde la raíz**
- Para el recorrido normalmente se usa la **propiedad recursiva** de los árboles
- Cuando se aplica un algoritmo de visita de árboles se implementa la función "**visitar**" que puede realizar distintas operaciones **sobre cada nodo**
 - Visitar un nodo puede ser p.e. imprimir el contenido del nodo o liberar su memoria



Árboles binarios. Recorrido

21

• Recorridos en profundidad

- preorden, postorden, inorden
- Ejemplo de aplicación (en grafos): encontrar componentes conexas

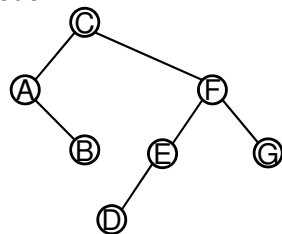
• Recorrido en anchura

- recorrido por nivel
- Ejemplos de aplicación (en grafos): camino más corto entre dos nodos, *crawling Web*

Árboles binarios. Recorrido en profundidad: preorden 22

- **Preorden** = orden previo
- Desde la raíz y **recursivamente**:
 1. Visitamos nodo actual n
 2. Recorremos en orden previo el hijo izquierdo de nodo actual n
 3. Recorremos en orden previo el hijo derecho de nodo actual n
- Ejemplo

visitar = *printf* del contenido de un nodo
resultado: C A B F E D G

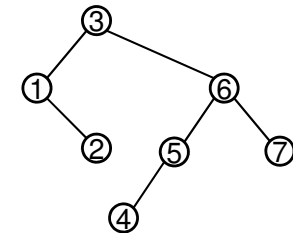


Árboles binarios. Recorrido en profundidad: preorden 23

- Algoritmo recursivo
 - Caso base / condición de parada
 - Caso general / llamada recursiva

• Pseudocódigo

```
ab_preorden(ArbolBinario T) {  
    // Árbol vacío  
    si ab_vacio(T) = TRUE:  
        volver  
    si no:  
        nodoab_visitar(T) // printf  
        ab_preorden(izq(T))  
        ab_preorden(der(T))  
        volver  
}
```



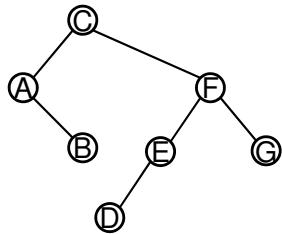
• Observaciones

- Árbol vacío \equiv no tiene nodos
- Árbol de un nodo \equiv un nodo raíz sin hijos
- Asociamos nodo \equiv raíz de un subárbol; útil para la recursión

Árboles binarios. Recorrido en profundidad: postorden 24

- **Postorden** = orden posterior
- Desde la raíz y recursivamente:
 1. Recorremos en orden posterior el hijo izquierdo de nodo actual n
 2. Recorremos en orden posterior el hijo derecho de nodo actual n
 3. Visitamos nodo actual n
- Ejemplo

visitar = *printf* del contenido de un nodo
resultado: B A D E G F C



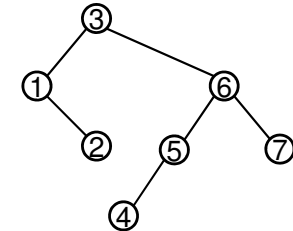
Árboles binarios. Recorrido en profundidad: postorden 25

- Pseudocódigo compacto

```
ab_postorden(ArbolBinario T) {  
  // Árbol no vacío  
  si ab_vacio(T) = FALSE:  
    ab_postorden(izq(T))  
    ab_postorden(der(T))  
    nodoab_visitar(T)  
}
```

- Pseudocódigo más eficiente

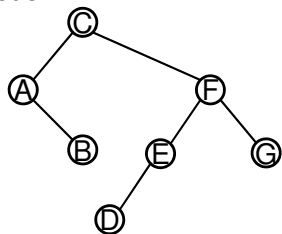
```
ab_postorden(ArbolBinario T) {  
  // Árbol no vacío  
  si ab_vacio(T) = FALSE:  
    si ab_vacio(izq(T)) = FALSE:  
      ab_postorden(izq(T))  
    si ab_vacio(der(T)) = FALSE:  
      ab_postorden(der(T))  
    nodoab_visitar(T)  
}
```



Árboles binarios. Recorrido en profundidad: inorden 26

- **Inorden** = orden medio
- Desde la raíz y recursivamente:
 1. Recorremos en orden posterior el hijo izquierdo de nodo actual n
 2. Visitamos nodo actual n
 3. Recorremos en orden posterior el hijo derecho de nodo actual n
- Ejemplo

visitar = *printf* del contenido de un nodo
resultado: A B C D E F G



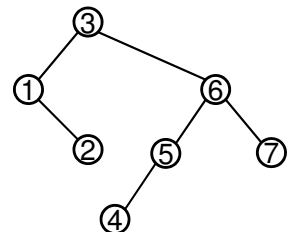
Árboles binarios. Recorrido en profundidad: inorden 27

- Pseudocódigo compacto

```
ab_inorden(ArbolBinario T) {  
  // Árbol no vacío  
  si ab_vacio(T) = FALSE:  
    ab_inorden(izq(T))  
    nodoab_visitar(T)  
    ab_inorden(der(T))  
}
```

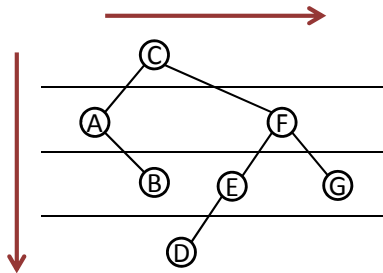
- Pseudocódigo más eficiente

```
ab_inorden(ArbolBinario T) {  
  // Árbol no vacío  
  si ab_vacio(T) = FALSE:  
    si ab_vacio(izq(T)) = FALSE:  
      ab_inorden(izq(T))  
    nodoab_visitar(T)  
    si ab_vacio(der(T)) = FALSE:  
      ab_inorden(der(T))  
}
```



- Recorrido en anchura = recorrido por nivel
- Algoritmo
 - Recorre de izquierda a derecha y de arriba a abajo
 - Nunca recorre un nodo de nivel i sin haber visitado todos los de nivel $i-1$
 - Implementación mediante el TAD Cola, sin recursividad
- Ejemplo

resultado: C A F B E G D



- Pseudocódigo
 - Recorre de arriba abajo y de izquierda a derecha
 - Nunca recorre un nodo de nivel i sin haber visitado los de nivel $i-1$

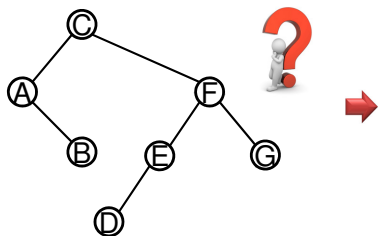
```

ab_anchura (ArbolBinario T) {
    Q = cola_crear()
    cola_insertar(Q, T)
    mientras cola_vacia(Q) = FALSE:
        T' = cola_extraer(Q)
        nodoab_visitar(T')
        para cada hijo H de T':
            cola_insertar(Q, H)
        cola_liberar(Q)
}
    
```

- Ejemplo

```

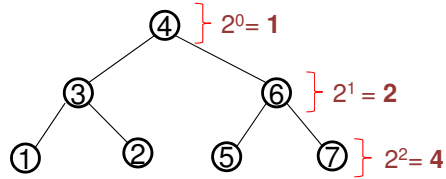
ab_anchura (ArbolBinario T) {
    Q = cola_crear()
    cola_insertar(Q, T)
    mientras cola_vacia(Q) = FALSE:
        T' = cola_extraer(Q)
        nodoab_visitar(T')
        para cada hijo H de T':
            cola_insertar(Q, H)
        cola_liberar(Q)
}
    
```



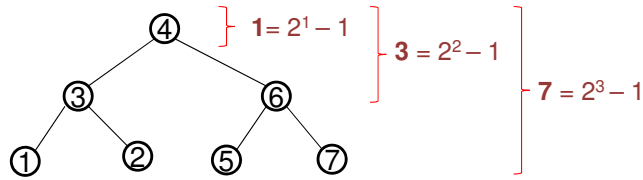
VISITAR	Q
	C
C	A F
A	F B
F	B E G
B	E G
E	7 D
G	D
D	

- Grafos y árboles
- **Árboles binarios**
 - Definición
 - Recorrido
 - **Compleitud**
 - Implementación en C
- Árboles binarios de búsqueda
 - Definición
 - Construcción de un árbol e inserción y búsqueda de un elemento
 - Extracción de un elemento
 - Equilibrado
- Árboles de expresión
 - Definición y recorrido
 - Construcción

- Cada nivel de profundidad d de un AB puede albergar 2^d nodos



- En total, un árbol de profundidad p completo puede albergar $(2^{p+1}-1)$ nodos



→ profundidad mínima necesaria para albergar N nodos:

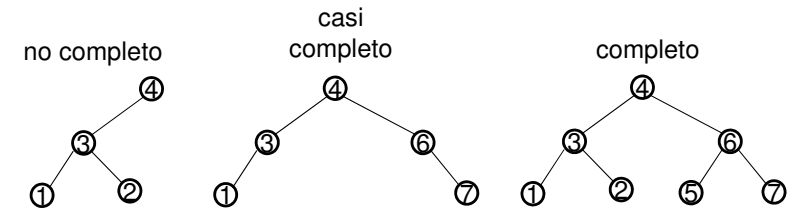
$$p = \text{prof}(T) = \lceil \log_2(N + 1) \rceil - 1$$

• AB casi completo

- Todos los niveles con profundidad $d < p$ están completos, i.e. tienen 2^d nodos

• AB completo

- Todos los niveles con profundidad $d \leq p$ están completos, i.e. tienen 2^d nodos
(= casi completo y tiene exactamente 2^p hojas a profundidad p)



Contenidos

- Grafos y árboles
- **Árboles binarios**
 - Definición
 - Recorrido
 - Completitud
 - **Implementación en C**
- Árboles binarios de búsqueda
 - Definición
 - Construcción de un árbol e inserción y búsqueda de un elemento
 - Extracción de un elemento
 - Equilibrado
- Árboles de expresión
 - Definición y recorrido
 - Construcción

Árboles binarios. Implementación en C

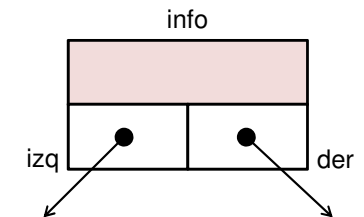
- Estructura de datos de un nodo de un árbol binario

```
// En arbolbinario.c
```

```
#define info(pnodo) ((pnodo)->info)
#define izq(pnodo) ((pnodo)->izq)
#define der(pnodo) ((pnodo)->der)
```

```
struct _NodoAB {
    Elemento *info;
    struct _NodoAB *izq;
    struct _NodoAB *der;
};
```

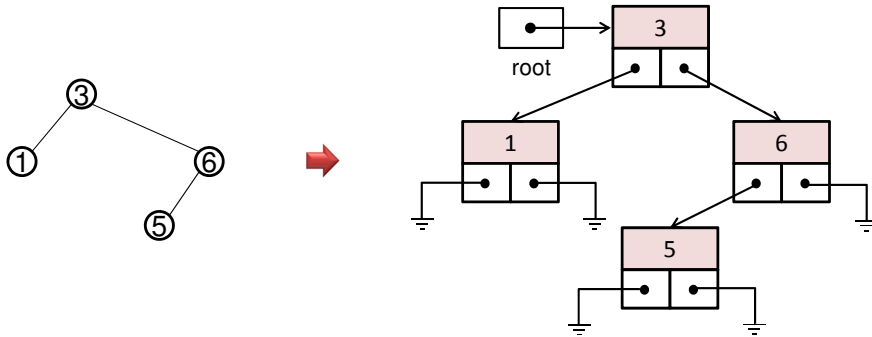
```
typedef struct _NodoAB NodoAB;
```



- Estructura de datos de un árbol binario

```
// En arbolbinario.c
#define root(pab) ((pab)->root)

struct _ArbolBinario {
    NodoAB *root; // un árbol es el puntero a su nodo raíz
};
// En arbolbinario.h
typedef struct _ArbolBinario ArbolBinario;
```



- Funciones de creación y liberación de un nodo de un árbol binario

```
NodoAB *nodoab_crear();
// Crea un nuevo nodo e inicializa sus campos a NULL

void nodoab_liberar(NodoAB *pn);
// Libera memoria de un nodo tras llamar a elemento_liberar
```

```
// La inicialización de info se hará tras la llamada a nodoab_crear
NodoAB *nodoab_crear() {
    NodoAB *pn = NULL;

    pn = (NodoAB *) malloc(sizeof(NodoAB));
    if (!pn) return NULL;

    info(pn) = izq(pn) = der(pn) = NULL;

    return pn;
}

void nodoab_liberar(NodoAB *pn) {
    if (pn) {
        elemento_liberar(info(pn));
        free(pn);
    }
}
```

- Primitivas del TAD árbol binario

```
ArbolBinario *ab_crear();
// Reserva memoria e inicializa un árbol

boolean ab_vacio(ArbolBinario *pa);
// Indica si un árbol tiene algún nodo o no

void ab_liberar(ArbolBinario *pa);
// Libera la memoria de un árbol y todos sus nodos
```

```
struct _ArbolBinario {  
    NodoAB *root;  
};
```

```
ArbolBinario *ab_crear() {
```



```
}
```

```
struct _ArbolBinario {  
    NodoAB *root;  
};
```

```
ArbolBinario *ab_crear() {  
    ArbolBinario *pa = NULL;
```

```
    pa = (ArbolBinario *) malloc(sizeof(ArbolBinario));  
    if (!pa) return NULL;
```

```
    root(pa) = NULL;
```

```
    return pa;
```

```
}
```

```
struct _ArbolBinario {  
    NodoAB *root;  
};
```

```
boolean ab_vacio(ArbolBinario *pa) {
```



```
}
```

```
struct _ArbolBinario {  
    NodoAB *root;  
};
```

```
boolean ab_vacio(ArbolBinario *pa) {
```

```
    if (!pa) {  
        return TRUE;  
    }
```

```
    if (!root(pa)) {  
        return TRUE;  
    }
```

```
    return FALSE;
```

```
}
```

- Implementar la función **ab_liberar**



- La **liberación de un árbol** se realiza usando la propiedad de recursión
 - El hijo izquierdo de un nodo forma un nuevo árbol con dicho hijo como raíz
 - El hijo derecho de un nodo forma un nuevo árbol con dicho hijo como raíz
 → Para liberar un árbol desde su raíz: primero se libera el árbol del hijo izquierdo y el árbol hijo derecho, y luego se libera la raíz
- Esquema de liberación: recorrido *postorden* de árbol T considerando como *visita* de un nodo su liberación

```
ab_liberar(T) {
    ab_liberar(izq(T))
    ab_liberar(der(T))
    liberar(root(T)) // visita de la raíz = liberación de la raíz
}
```

```
// Función públicamente declarada en arbolbinario.h
void ab_liberar(ArbolBinario *pa) {
    if (!pa) return;

    ab_liberar_rec(root(pa)); // Primera llamada: root

    free(pa);
}
```

```
// Función privada en arbolbinario.c
void ab_liberar_rec(NodoAB *pn) {
```



```
// Función públicamente declarada en arbolbinario.h
void ab_liberar(ArbolBinario *pa) {
    if (!pa) return;

    ab_liberar_rec(root(pa)); // Primera llamada: root

    free(pa);
}

// Función privada en arbolbinario.c
void ab_liberar_rec(NodoAB *pn) {
    if (!pn) return;

    if (izq(pn)) { // Liberación de subárbol izquierdo
        ab_liberar_rec(izq(pn));
    }
    if (der(pn)) { // Liberación de subárbol derecho
        ab_liberar_rec(der(pn));
    }
    nodoab_liberar(pn); // visitar nodo = liberar nodo
}
```



- ¿Cuál sería la implementación de primitivas para insertar y extraer elementos de un árbol binario?
 - La respuesta se abordará a continuación al estudiar los árboles binarios de búsqueda

- Grafos y árboles
- Árboles binarios
 - Definición
 - Recorrido
 - Completitud
 - Implementación en C
- Árboles binarios de búsqueda
 - **Definición**
 - Construcción de un árbol e inserción y búsqueda de un elemento
 - Extracción de un elemento
 - Equilibrado
- Árboles de expresión
 - Definición y recorrido
 - Construcción

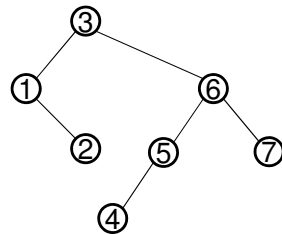
Árboles binarios de búsqueda. Definición

50

- Un **Árbol Binario de Búsqueda** (ABdB) es un árbol binario T tal que \forall sub-árbol T' de T se cumple que

$$\text{info}(\text{izq}(T')) < \text{info}(T') < \text{info}(\text{der}(T'))$$

dado un criterio de ordenación para los $\text{info}(T)$, que vendrá dado por una primitiva `elemento_comparar(e, e')` del TAD Elemento

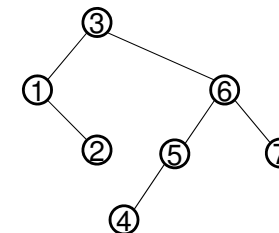


- Nota: Cada subárbol de un ABdB es a su vez un ABdB

Árboles binarios de búsqueda. Orden medio

51

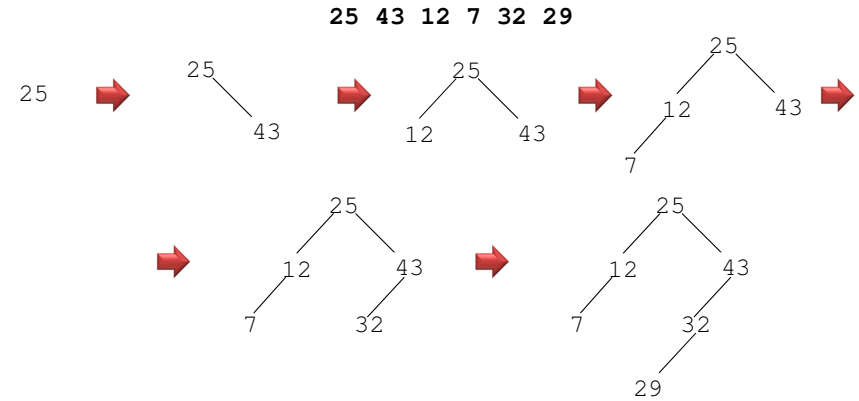
- Recorrido en orden medio de un ABdB



- Salida \rightarrow 1 2 3 4 5 6 7
- ¡Listado ordenado de los nodos!

- Grafos y árboles
- Árboles binarios
 - Definición
 - Recorrido
 - Completitud
 - Implementación en C
- Árboles binarios de búsqueda
 - Definición
 - Construcción de un árbol e inserción y búsqueda de un elemento
 - Extracción de un elemento
 - Equilibrado
- Árboles de expresión
 - Definición y recorrido
 - Construcción

- Creación de ABdB: inserción iterativa de valores en ABdB parciales
- Ejemplo



- La función `abdb_insertar(T, e)` que introduce un dato `e` en un árbol `T` realiza lo siguiente:
 - Si `T` está vacío, se crea un nodo con `e` y se inserta
 - Si no, se hace una llamada recursiva a insertar
 - Si `e < info(T)`, entonces se ejecuta insertar (`izq(T)`)
 - Si `e > info(T)`, entonces se ejecuta insertar (`der(T)`)
 - Caso excepcional: si dato `e = info(T)`, se devuelve ERROR o se ignora devolviendo OK

- Pseudocódigo

```

status abdb_insertar(ArbolBinario T, Elemento e)
si ab_vacio(T) // Caso base
    T = nodoab_crear()
    si (T == NULL) devolver ERROR
    info(T) = e
    devolver OK
else // Caso general
    si info(T) < e
        devolver abdb_insertar(izq(T), d)
    else
        devolver abdb_insertar(der(t), d)
    
```

- Implementación en C

```

status abdb_insertar(ArbolBinario *pa, Elemento *pe) { // Función pública
    if (!pa || !pe) return ERROR;

    return abdb_insertar_rec(&root(pa), pe);
}

status abdb_insertar_rec(NodoAB **ppn, Elemento *pe) { // Función privada
    int cmp;

    if (!*ppn) {
        *ppn = nodoab_crear(pe); // Ojo: crea nodo con copia de pe en info
        if (!*ppn) return ERROR;
    }

    cmp = elemento_comparar(pe, info(*ppn));
    if (cmp < 0) {
        return abdb_insertar_rec(&izq(*ppn), pe);
    }
    if (cmp > 0) {
        return abdb_insertar_rec(&der(*ppn), pe);
    }
    if (cmp == 0) {
        return OK; // El elemento ya estaba en el árbol: se podría devolver ERROR
    }

    return OK;
}
    
```

- Dado un vector de valores representados en una lista, la construcción de su correspondiente ABdB es como sigue:

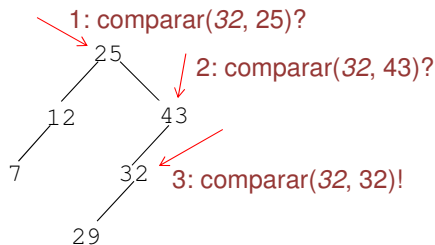
```

status abdb_crear(ArbolBinario T, Lista L)
    mientras lista_vacia(L) == FALSE && st == OK
        e = lista_extraerIni(L)
        st = abdb_insertar(T, e)
        si st == ERROR
            ab_liberar(T) // Ojo: la lista L no se ha recuperado
            devolver ERROR
        devolver OK
    
```

- Una vez creado el ABdB

- Recorrer el árbol en orden medio recupera los datos ordenados
- Buscar un dato en el árbol es muy eficiente
 - Buscar en una lista desordenada es menos eficiente, pues hay que recorrerla de forma secuencial

- Búsqueda de un dato, p.e. 32



- ¿Búsqueda de 33?

- Se hacen llamadas recursivas hasta llegar a un árbol vacío

- Pseudocódigo

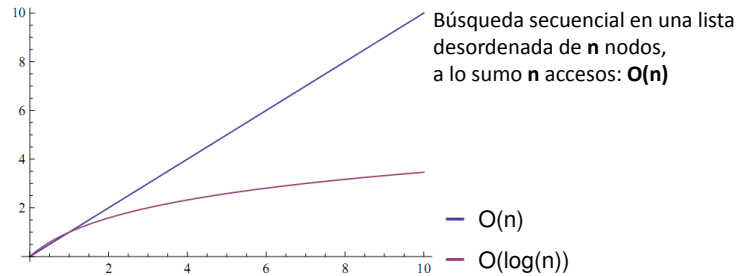
```

Arbol abdb_buscar(ArbolBinario T, Elemento e)
    si ab_vacio(T) == TRUE
        devolver NULL
    else si info(T) == e
        devolver T
    else si e < info(T)
        devolver abdb_buscar(izq(T), e)
    else
        devolver abdb_buscar(der(T), e)
    
```

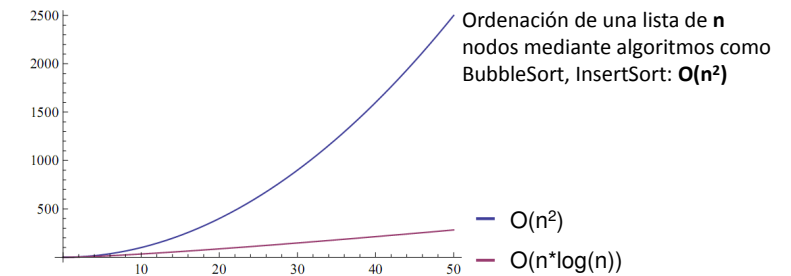
- ¿Cuántas comparaciones en promedio se tienen que hacer para encontrar un dato en un ABdB?



- **Coste (número de accesos/comparaciones) de buscar un dato en un ABdB**
- Para un ABdB (casi) completo con profundidad **p**:
 - a lo sumo **p** accesos
- Para un ABdB (casi) completo de **n** nodos:
 - a lo sumo tantos accesos como la profundidad del árbol $\equiv \lceil \log_2(n + 1) - 1 \rceil \equiv \text{Orden}(\log(n)) \equiv \mathbf{O(\log(n))}$



- Para **n** elementos, hay que realizar **n** inserciones
 - Dado árbol (casi) completo, cada inserción es a lo sumo del orden de la profundidad actual del árbol $\leq \lceil \log_2(n + 1) - 1 \rceil \equiv \text{orden}(\log(n)) \equiv O(\log(n))$
 - La creación del árbol es $O(n \cdot \log(n))$, pues involucra **n** inserciones de orden $O(\log(n))$
 - Una vez creado el árbol, éste se puede usar para ordenar sus elementos recorriéndolo por orden medio $\equiv O(n) \rightarrow$ ordenación es $O(n \cdot \log(n)) + O(n) \equiv \mathbf{O(n \cdot \log(n))}$



Contenidos

62

- Grafos y árboles
- Árboles binarios
 - Definición
 - Recorrido
 - Completitud
 - Implementación en C
- **Árboles binarios de búsqueda**
 - Definición
 - Construcción de un árbol e inserción y búsqueda de un elemento
 - **Extracción de un elemento**
 - Equilibrado
- Árboles de expresión
 - Definición y recorrido
 - Construcción

Árboles binarios de búsqueda. Extracción

63

- Pseudocódigo

```
status abdb_extraer(ArbolBinario T, Elemento e)
si ab_vacio(T) == TRUE
    devolver ERROR
T' = abdb_buscar(T, e) // Buscar devuelve el nodo donde está e
if T' == NULL
    devolver OK
else
    devolver abdb_reajustar(T')
```

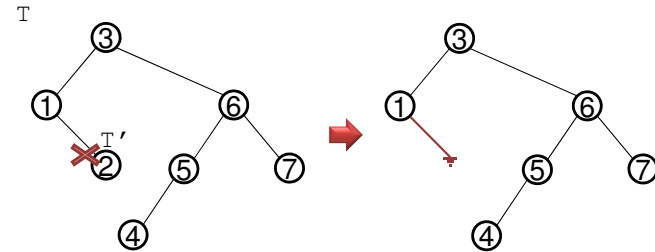

• **Reajuste** de un (sub-)árbol T' por la extracción de su raíz

1. La raíz de T' es hoja
2. La raíz de T' tiene 1 hijo
3. La raíz de T' tiene 2 hijos

• **Reajuste** de un (sub-)árbol T' por la extracción de su raíz

1. La raíz de T' es hoja: reajustar puntero del padre de (la raíz de) T' a NULL, eliminar T'
2. La raíz de T' tiene 1 hijo
3. La raíz de T' tiene 2 hijos

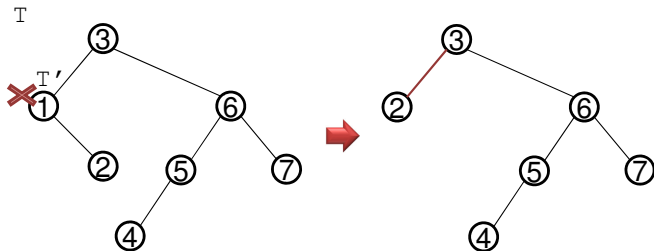
Ejemplo: `abdb_extraer(T, 2)`



• **Reajuste** de un (sub-)árbol T' por la extracción de su raíz

1. La raíz de T' es hoja
2. La raíz de T' tiene 1 hijo: reajustar puntero del padre de T' al hijo de T' , eliminar de T'
3. La raíz de T' tiene 2 hijos

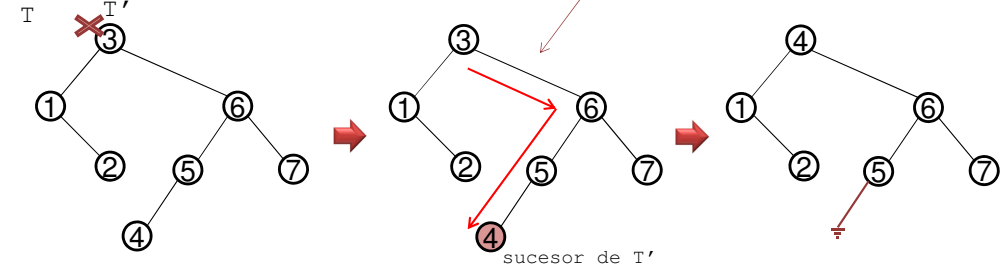
Ejemplo: `abdb_extraer(T, 1)`



• **Reajuste** de un (sub-)árbol T' por la extracción de su raíz

1. La raíz de T' es hoja
2. La raíz de T' tiene 1 hijo
3. La raíz de T' tiene 2 hijos: buscar “sucesor” de T' , guardar info del sucesor en T' , extraer sucesor

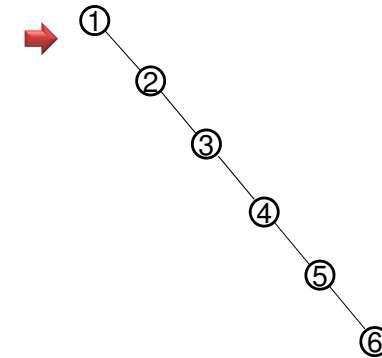
Ejemplo: `abdb_extraer(T, 3)`



- Grafos y árboles
- Árboles binarios
 - Definición
 - Recorrido
 - Completitud
 - Implementación en C
- **Árboles binarios de búsqueda**
 - Definición
 - Construcción de un árbol e inserción y búsqueda de un elemento
 - Extracción de un elemento
 - **Equilibrado**
- Árboles de expresión
 - Definición y recorrido
 - Construcción

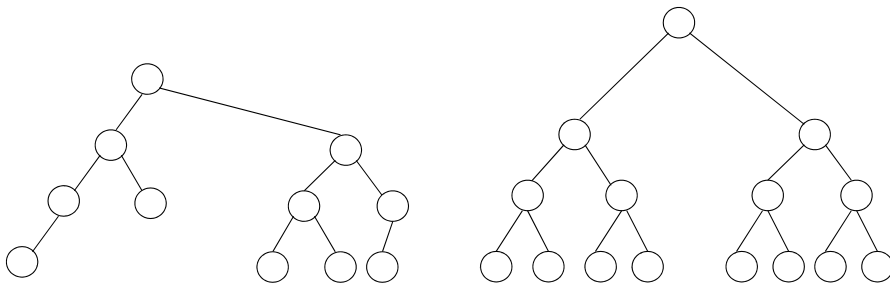
- Problema de ABdB: **árboles no equilibrados**

$L = \{1, 2, 3, 4, 5, 6\}$
abddbCrear(T, L)



- El acceso ya no es $O(\log(n))$, sino $O(n)$, por lo que:
 - Coste de la búsqueda: $O(n \cdot \log(n)) \rightarrow O(n)$
 - Coste de la ordenación: $O(n) \rightarrow O(n^2)$

- Un árbol está **equilibrado** si para todo nodo el número de niveles de sus sub-árboles no difieren en más de una unidad
- Un árbol con máximo número k de hijos por nodo está **perfectamente equilibrado** si todo nodo tiene k hijos



Árbol equilibrado

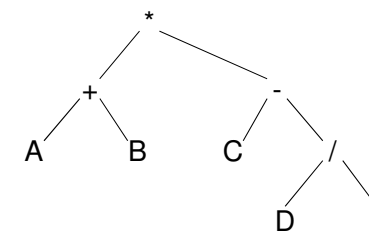
Árbol perfectamente equilibrado



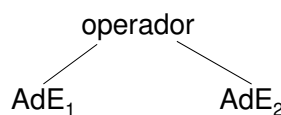
- **¿Cómo crear ABdB equilibrados?**
 - Mediante el **algoritmo AVL** (se estudiará en otra asignatura)

- Grafos y árboles
- Árboles binarios
 - Definición
 - Recorrido
 - Completitud
 - Implementación en C
- Árboles binarios de búsqueda
 - Definición
 - Construcción de un árbol e inserción y búsqueda de un elemento
 - Extracción de un elemento
 - Equilibrado
- **Árboles de expresión**
 - **Definición y recorrido**
 - Construcción

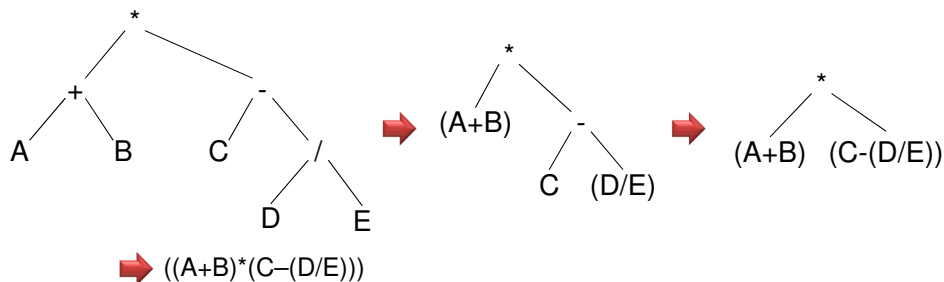
- Un **Árbol de Expresión (AdE)** es un árbol binario donde:
 - Los nodos tienen operadores
 - Las hojas tienen operandos
 - (Todo nodo tiene 2 hijos, i.e., operador sobre dos valores)



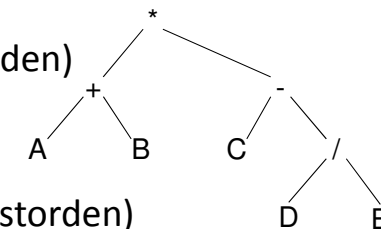
- Los sub-árboles (de más de un nodo) de un AdE son AdE



- Un AdE almacena una expresión aritmética

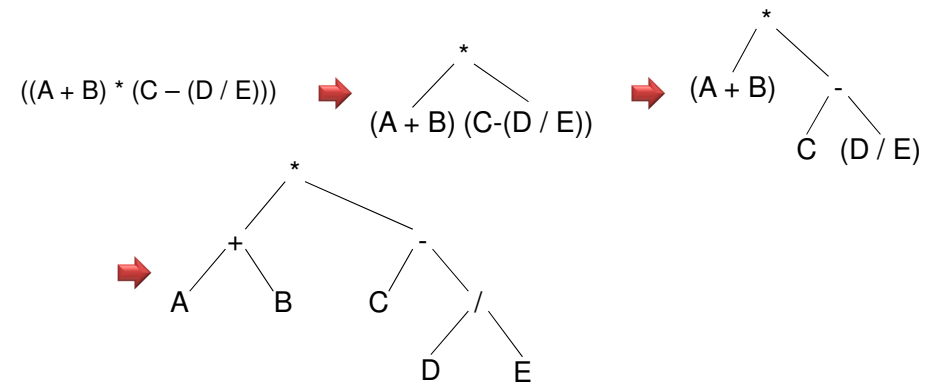


- Recorrido en orden previo (preorden)
 - Salida: * + A B - C / D E
 - Forma prefijo de la expresión
- Recorrido en orden posterior (postorden)
 - Salida: A B + C D E / - *
 - Forma postfijo de la expresión
- Recorrido en orden medio (inorden)
 - “Imprimiendo” paréntesis al comienzo y al final de la llamada a cada sub-árbol
 - Salida: ((A + B) * (C - (D / E)))
 - Forma infijo de la expresión



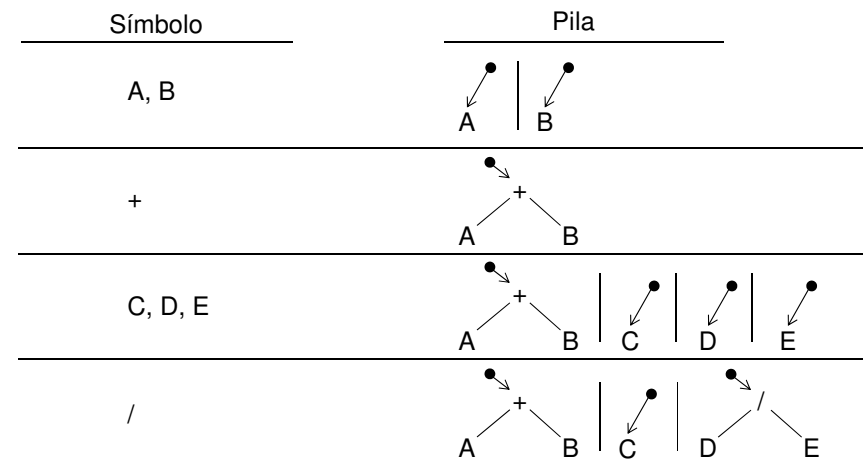
- Grafos y árboles
- Árboles binarios
 - Definición
 - Recorrido
 - Completitud
 - Implementación en C
- Árboles binarios de búsqueda
 - Definición
 - Construcción de un árbol e inserción y búsqueda de un elemento
 - Extracción de un elemento
 - Equilibrado
- **Árboles de expresión**
 - Definición y recorrido
 - **Construcción**

- Construcción de un AdE
 - El paso de una expresión infija a un AdE es natural “a ojo”:



- Construcción de un AdE
 - Basada en la evaluación de expresiones mediante el **TAD Pila**
 - Consistente en la evaluación de una expresión guardando en una pila árboles generados para sub-expresiones
- El algoritmo de evaluación más sencillo es el de expresiones postfijo
 - Si se tiene una expresión prefijo o infija, ésta se pasa a postfijo para evaluarla
 - $(A+B)*(C-D/E) \rightarrow$ a postfijo $\rightarrow A B + C D E / - *$ \rightarrow evaluación

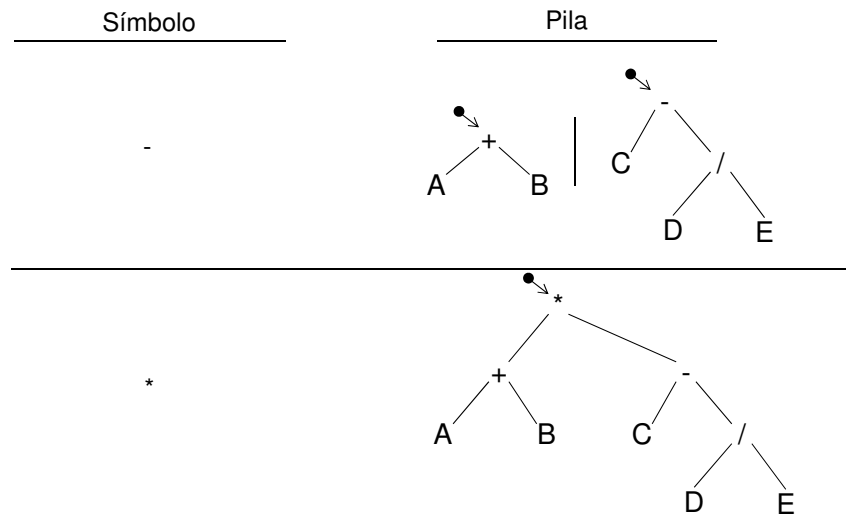
- Ejemplo 1: $A B + C D E / - *$



Árboles de expresión. Construcción

80

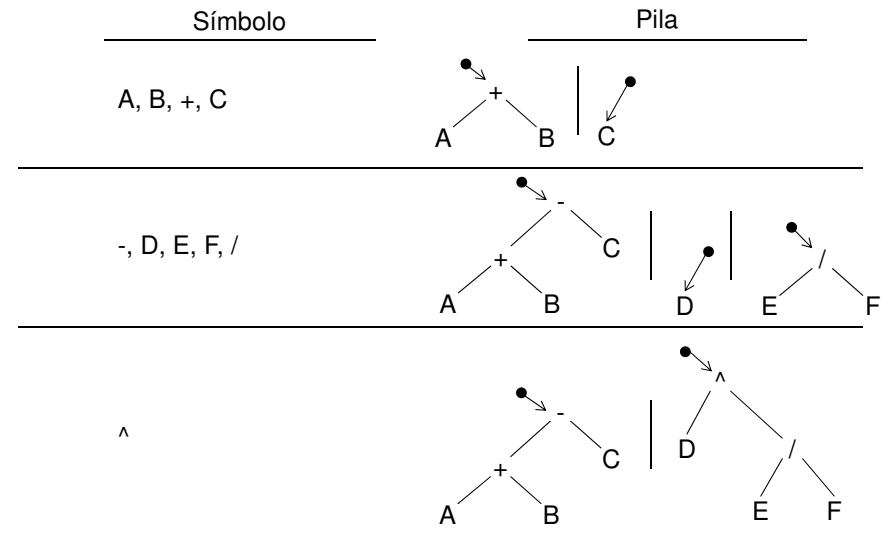
- Ejemplo 1: $A B + C D E / - *$



Árboles de expresión. Construcción

81

- Ejemplo 2: $(A + B - C) * (D \wedge (E / F)) \rightarrow A B + C - D E F / \wedge *$



Árboles de expresión. Construcción

82

- Ejemplo 2: $(A + B - C) * (D \wedge (E / F)) \rightarrow A B + C - D E F / \wedge *$

