EDAT

Tema 5. Árboles binarios

Escuela Politécnica Superior
Universidad Autónoma de Madrid

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
  - Borrado de un elemento
  - Equilibrado
- Árboles de expresión
  - Definición y recorrido
  - Construcción

Anexo 1: Método de inserción alternativo

Contenidos

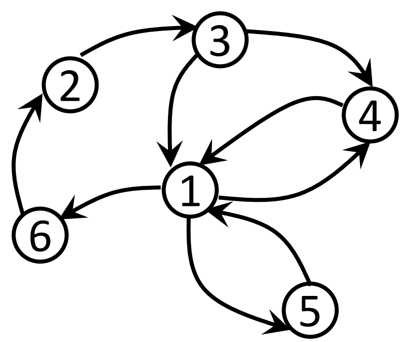
- **Grafos y árboles**
- Árboles binarios
  - Definición
  - Recorrido
  - Completitud
  - Implementación en C
- Árboles binarios de búsqueda
  - Definición
  - Construcción de un árbol e inserción y búsqueda de un elemento
  - Borrado de un elemento
  - Equilibrado
- Árboles de expresión
  - Definición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

Grafos. Definición

- Un **grafo** es una estructura de datos G = (V, A) compuesta de:
  - Un conjunto V de **vértices** (nodos)
  - Un conjunto A de **aristas** (arcos), pares de vértices de V que representan una conexión entre ellos.
  - Grafos dirigidos: las aristas son pares **ordenados** de vértices
  - Grafos no dirigidos: las aristas son pares **no ordenados**
    - Todo grafo no dirigido puede representarse mediante un grafo dirigido en el que para todo $(u, v) \in A, (v, u) \in A$
- Ejemplo de grafo (*dirigido*)

  V = {1, 2, 3, 4, 5, 6}

  A = {(1,4), (1,5), (1,6), (2,3), (3,1), (3,4), (4,1), (5,1), (6,2)}

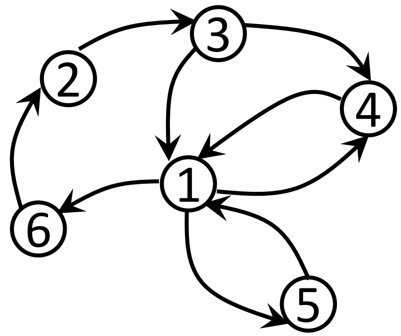


Grafos. Definición

- Un grafo es una EdD general, muy rica y flexible
- Ejemplos:
  - redes de transporte (carreteras, metro)
  - redes de comunicaciones
  - redes sociales (amigos, seguidores,…)
  - enlaces químicos
  - la web como red de enlaces
  - conexiones entre agentes (prestigio, influencia, epidemias!)
  - migraciones humanas o de especies animales
- Todos estos casos tienen en común un conjunto de objetos y unas conexiones entre estos objetos

Grafos. Caminos

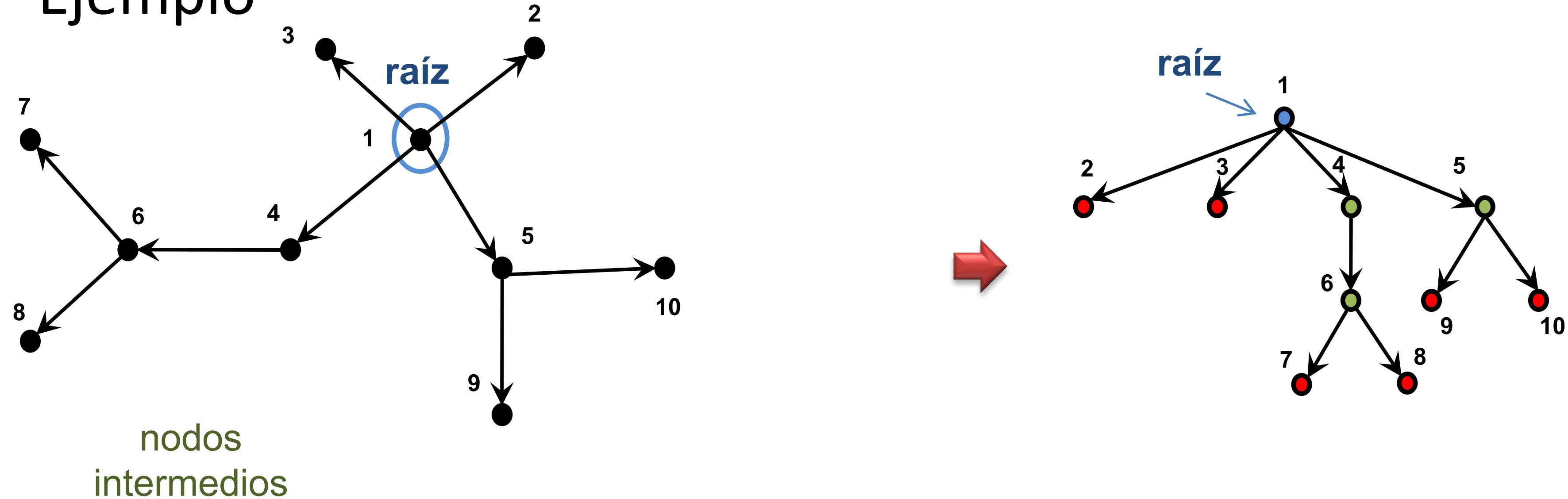
- Un **camino** de un grafo G = (V, A) es una secuencia de nodos de V en los que cada nodo está conectado al siguiente mediante un arco de A
- Ejemplo



  - V = {1, 2, 3, 4, 5, 6}
  - A = {(1,4), (1,5), (1,6), (2,3),…, (5,1), (6,2)}
  - Caminos: {1, 6, 2, 3}, {5, 1, 4}, …

Árboles. Definición

- Un **árbol** con raíz es un grafo dirigido tal que:
  - tiene un nodo distinguido, denominado **raíz**, sin arcos incidentes
  - Cada nodo ≠ raíz recibe un solo arco
  - Cualquier nodo es accesible desde la raíz
- Ejemplo

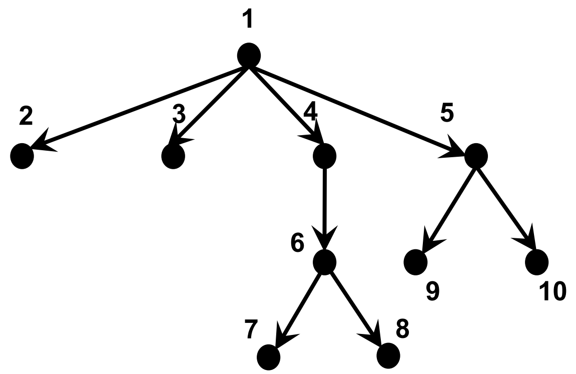


En un árbol:

- Las aristas suelen ser llamadas **ramas**
- Dada una arista $(u, v) \in A$, se dice que u es el **padre** de v, y v el **hijo** de u
- Dos nodos con el mismo padre son **hermanos**

Ejemplo:

- 6 es padre de 7 y 8
- 3 es hijo de 1
- 1 (la raíz) no tiene padre
- 2, 3, 4 y 5 son hermanos



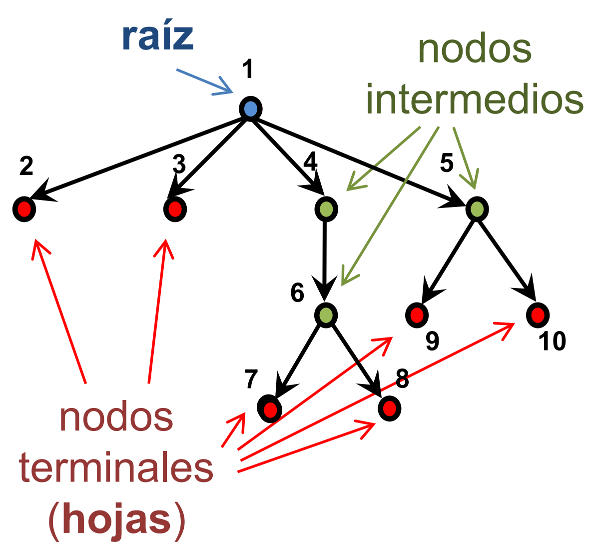
Árboles. Terminología

En un árbol:

- Todos los vértices en un camino desde la raíz hasta un vértice v son **antecesores** de v
- Los **descendientes** de un vértice v son todos aquellos que tienen a v como antecesor
- Los nodos terminales (aquellos sin hijos) son llamados **hojas**
- Los nodos diferentes de la raíz que no son hojas son **nodos intermedios**

Ejemplo:

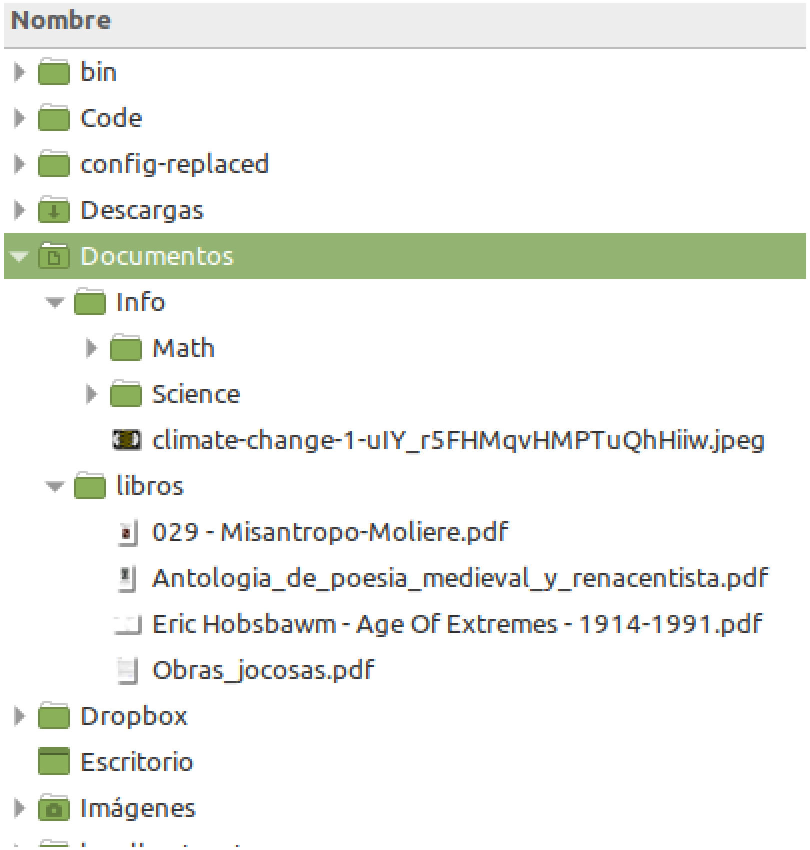
- 1, 4 y 6 son los antecesores de 7
- Los descendientes de 4 son 6, 7 y 8
- Las hojas son 2, 3, 7, 8, 9 y 10



Árboles. Ejemplos

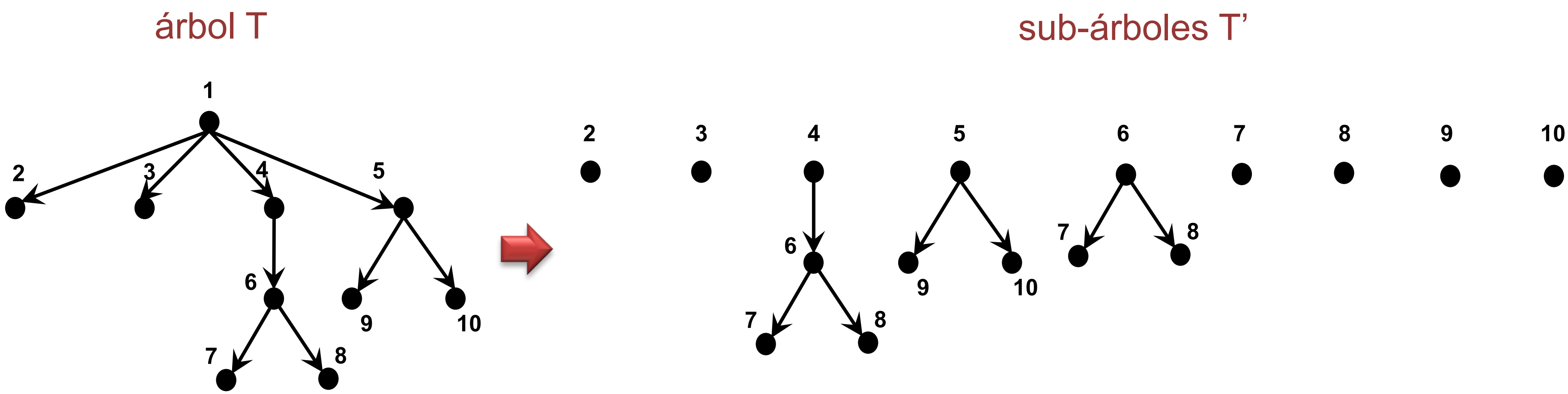
Árboles genealógicos

Carpetas y archivos



Árboles. Sub-árboles

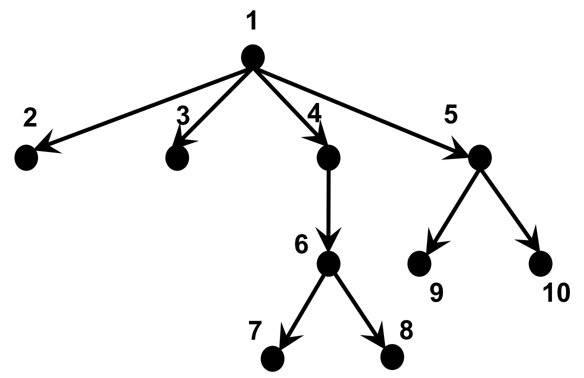
- El **sub-árbol** con raíz v de un árbol T es el subgrafo de T que contiene v y todos sus descendientes
- Cada nodo de un árbol T junto con sus hijos da lugar a nuevo sub-árbol T'



Árboles. Profundidad

- 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 y una hoja del árbol (es -1 si el árbol está vacío, 0 para un árbol con un nodo)
- Ejemplo

T ≡



profundidad(T) = 3

profundidad(1) = 0

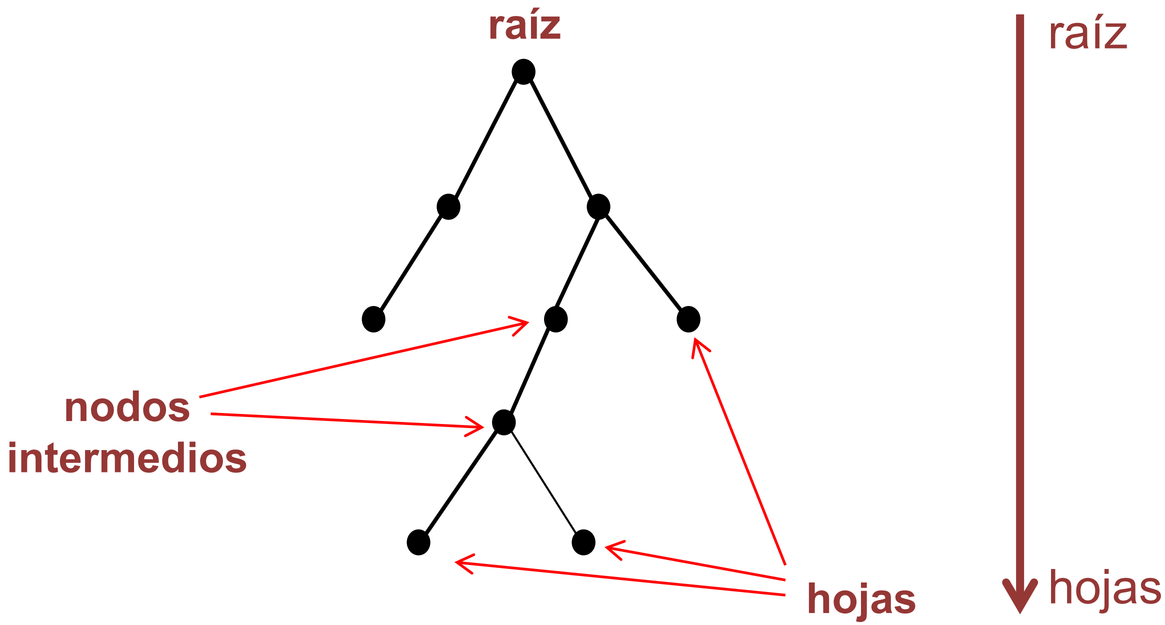
profundidad(5) = 1

profundidad(7) = 3

Contenidos

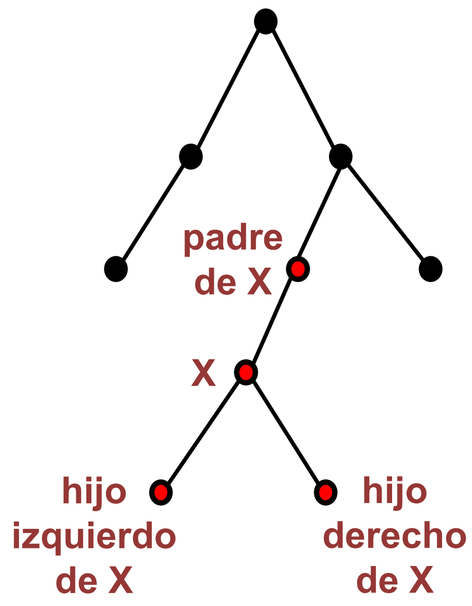
Árboles binarios. Definición

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



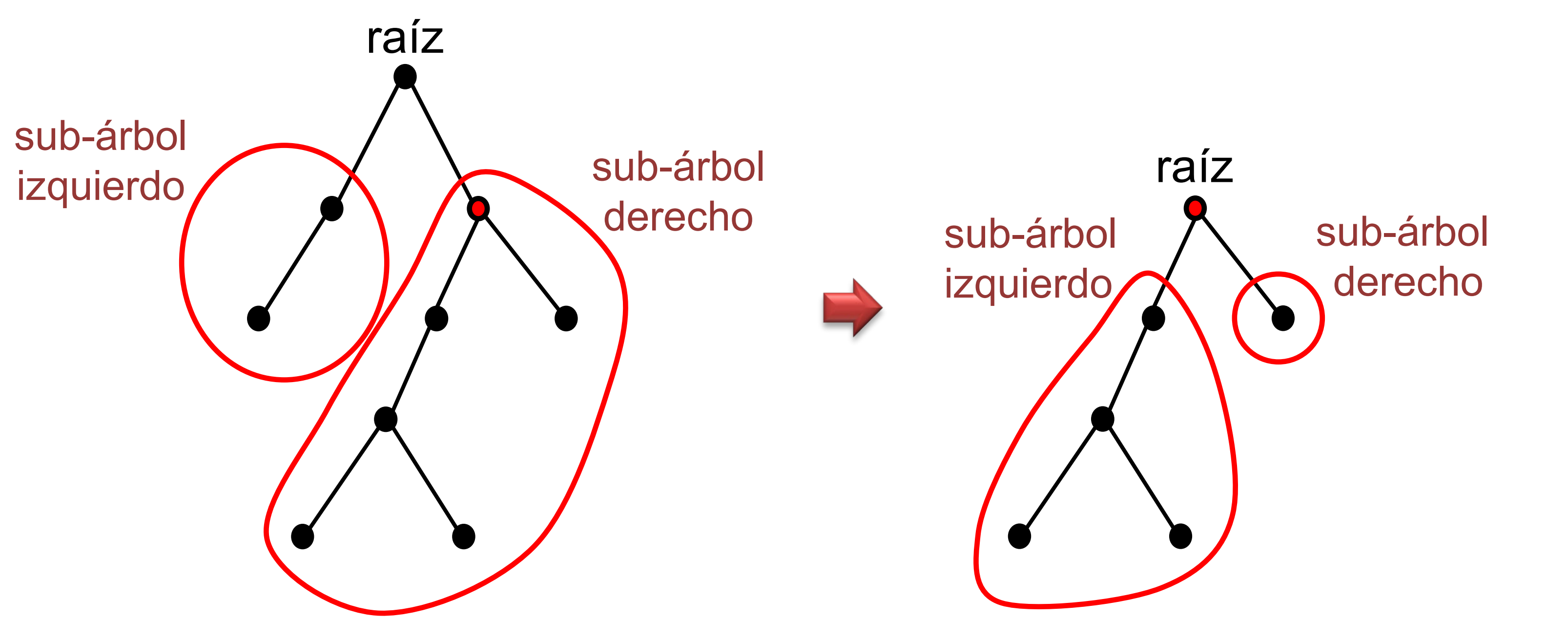
Árboles binarios. Definición

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



Árboles binarios. Definición

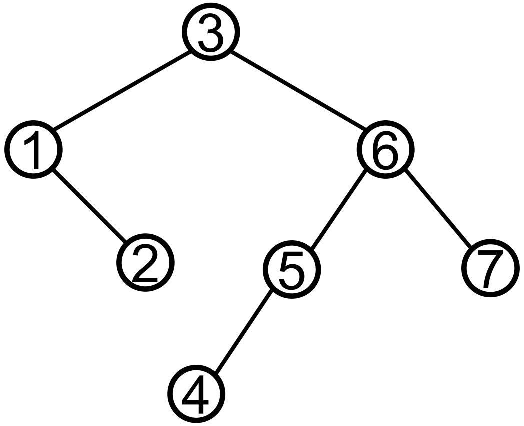
- **Propiedad recursiva de los AB**
  - El hijo izquierdo de la raíz (u otro nodo) forma un nuevo árbol binario con dicho hijo como raíz
  - El hijo derecho de la raíz (u otro nodo) forma un nuevo árbol binario con dicho hijo como raíz



Contenidos

Árboles binarios. Recorrido

- 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 <u>sobre cada nodo</u>
  - Visitar un nodo puede ser p.e. imprimir el contenido del nodo o liberar su memoria



Árboles binarios. Recorrido

- **Recorridos en profundidad:**
  - Profundizar siempre que sea posible, visitando los hijos antes que los hermanos
  - Ejemplos de aplicación: en grafos, caminos entre nodos, encontrar componentes conexas; en árboles, encontrar nodos que cumplan ciertas propiedades (por ejemplo, ficheros jpg dentro de un directorio)
- **Recorrido en anchura:**
  - Recorrido por nivel de profundidad: se visitan los hermanos antes que los hijos
  - Ejemplos de aplicación (en grafos): camino más corto entre dos nodos, *crawling* de la Web

Árboles binarios. Recorridos en profundidad

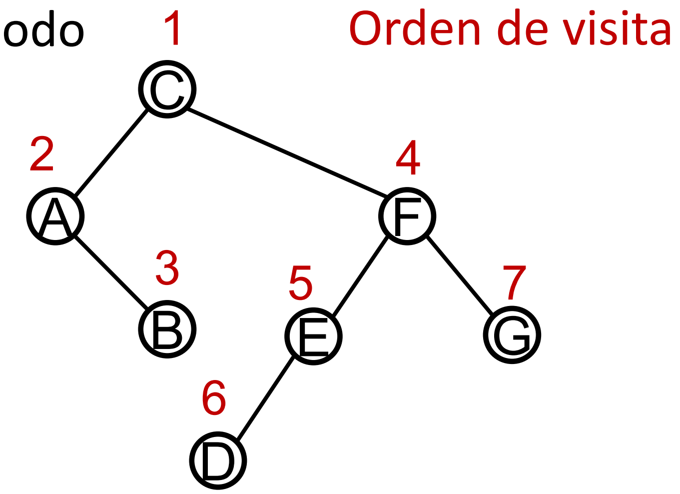
- **Recorridos en profundidad:**
  - Profundizar siempre que sea posible, visitando los hijos antes que los hermanos
  - Se pueden realizar tres pasos en diferentes órdenes:
    - Recorrer recursivamente el hijo izquierdo (y su subárbol)
    - Recorrer recursivamente el hijo derecho (y su subárbol)
    - Visitar el nodo
  - Dependiendo del orden de estas operaciones: preorden, postorden, inorden

Árboles binarios. Recorrido en profundidad: preorden

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

  visitar = *printf* del contenido de un nodo

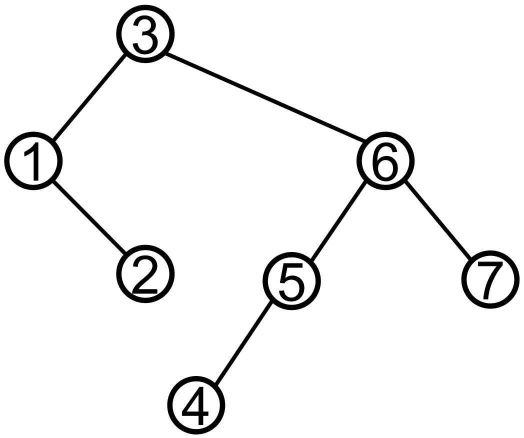
  resultado: C A B F E D G



Árboles binarios. Recorrido en profundidad: preorden

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

```
pre_order(BinaryTree T) {
   // Caso base: árbol vacío
   if bt_isEmpty(T) == TRUE:
      return
   // Llamadas recursivas
   visit(T) // por ej. printf
   pre_order(left(T))
   pre_order(right(T))
}
```



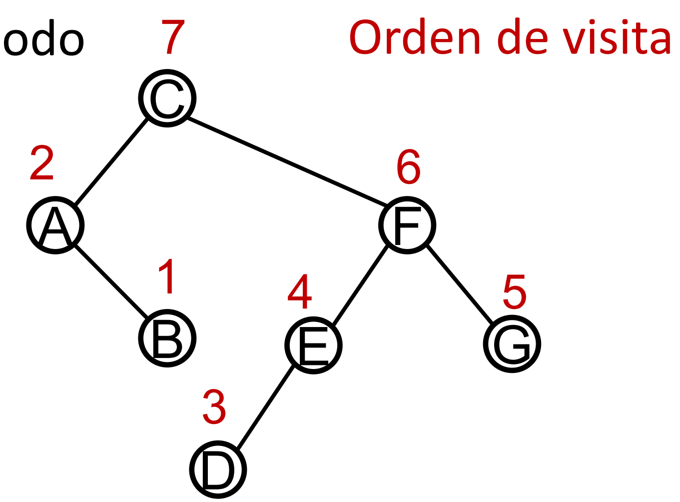
- Observaciones
  - Árbol vacío ≡ no tiene nodos
  - Asociamos nodo ≡ raíz de un subárbol; útil para la recursión

Árboles binarios. Recorrido en profundidad: postorden

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

  visitar = *printf* del contenido de un nodo

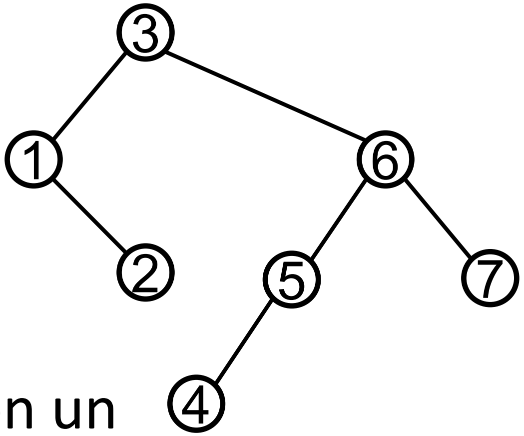
  resultado: B A D E G F C



Árboles binarios. Recorrido en profundidad: postorden

- Pseudocódigo

```
post_order(BinaryTree T) {
   // Si el árbol está vacío, no hace nada, retorna.
   // Si no está vacío:
   if bt_isEmpty(T) == FALSE:
      post_order(left(T))
      post_order(right(T))
      visit(T)
}
```



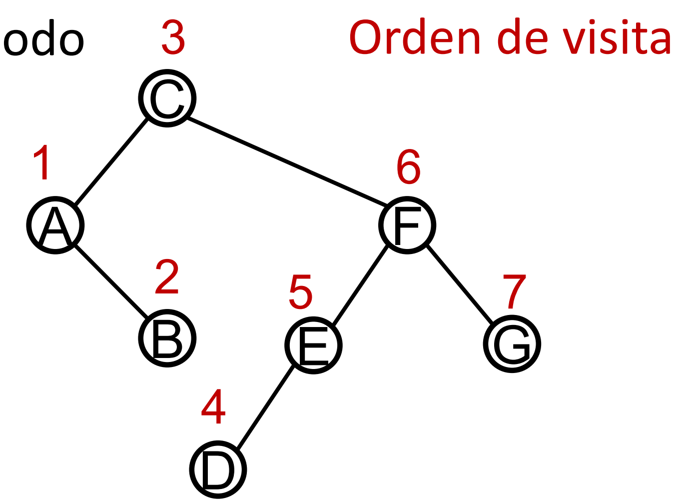
Podría también comprobarse si los hijos son un árbol vacío antes de hacer las llamadas recursivas

Árboles binarios. Recorrido en profundidad: inorden

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

  visitar = *printf* del contenido de un nodo

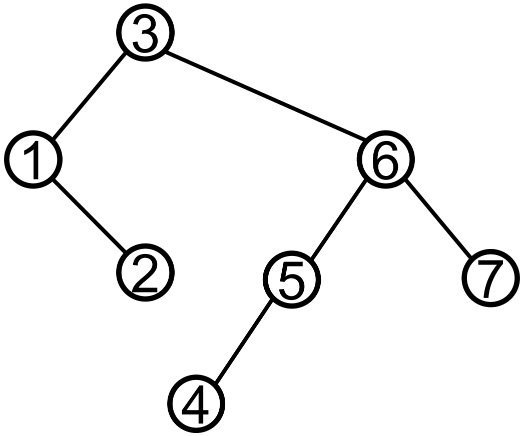
  resultado: A B C D E F G



Árboles binarios. Recorrido en profundidad: inorden

- Pseudocódigo

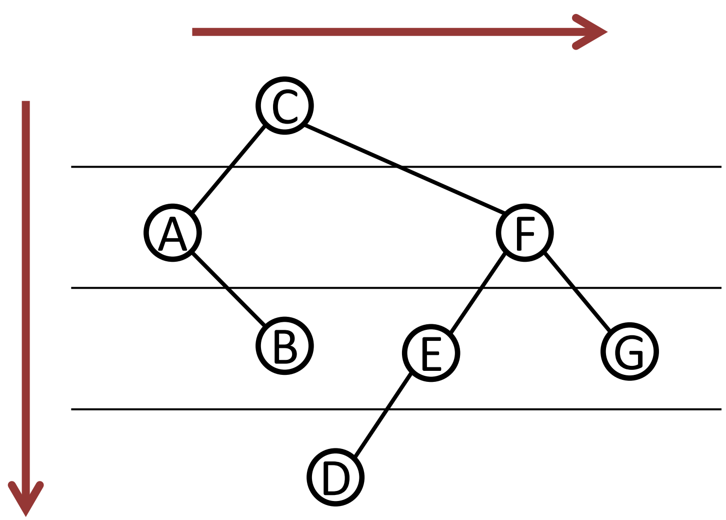
```
in_order(BinaryTree T) {
    // Si el árbol está vacío, no hace nada, retorna.
    // Si no está vacío:
    if bt_is_empty(T) == FALSE:
      in_order(left(T))
      visit(T)
      in_order(right(T))
}
```



Árboles binarios. Recorrido en anchura

- 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



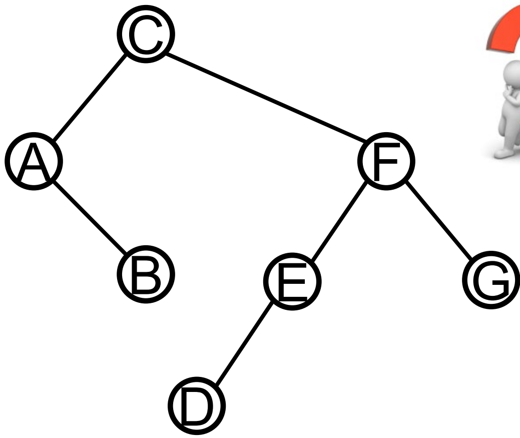
- Búsqueda en anchura: Breadth-first search
- Pseudocódigo
  - Recorre de arriba abajo y de izquierda a derecha
  - Nunca recorre un nodo de nivel *i* sin haber visitado los de nivel *i-1*

```
breadth_first_search(BinaryTree T) {
   Q = queue_new()
   queue_push(Q, T)
   while queue_isEmpty(Q) == FALSE:
      T' = queue_pop(Q)
      visit(T')
      queue_push(Q, left(T'))
      queue_push(Q, right(T'))
   queue_free(Q)
}
```

Árboles binarios. Recorrido en anchura

- Ejemplo

```
breadth_first_search(BinaryTree T) {
   Q = queue_new()
   queue_push(Q, T)
   while queue_isEmpty(Q) == FALSE:
      T' = queue_pop(Q)
      visit(T')
      queue_push(Q, left(T'))
      queue_push(Q, right(T'))
   queue_free(Q)
}
```



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

Árboles. Recorrido en anchura

También se puede usar para árboles NO binarios

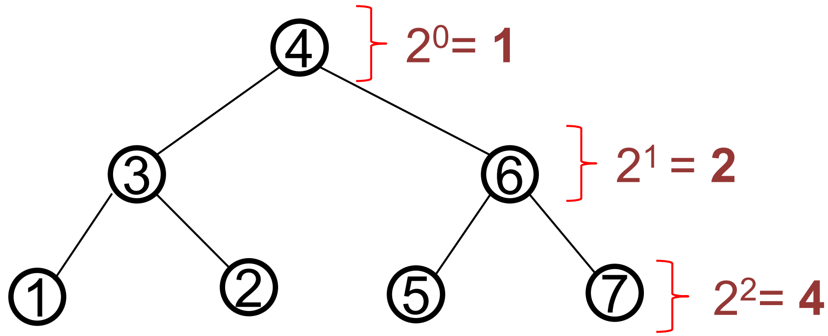
- Recorre de arriba abajo y de izquierda a derecha
- Nunca recorre un nodo de nivel *i* sin haber visitado los de nivel *i-1*

```
breadth_first_search(Tree T) {
   Q = queue_new()
   queue_push(Q, T)
   while queue_isEmpty(Q) == FALSE:
      T' = queue_pop(Q)
      visit(T')
      foreach child H of T'
         queue_push(Q, H)
   queue_free(Q)
}
```

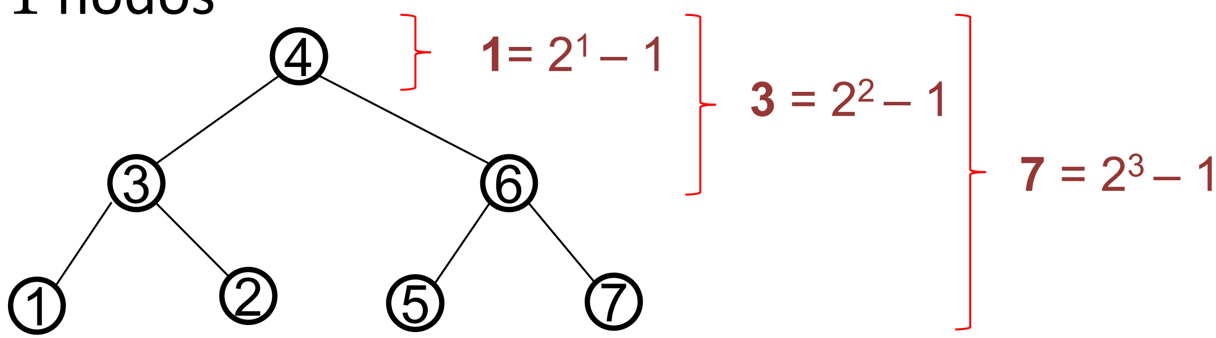
Contenidos

Árboles binarios. Completitud

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



- En total, un árbol de profundidad **p** completo puede albergar $\sum_{k=0}^{p} 2^k = 2^{p+1} - 1$ nodos

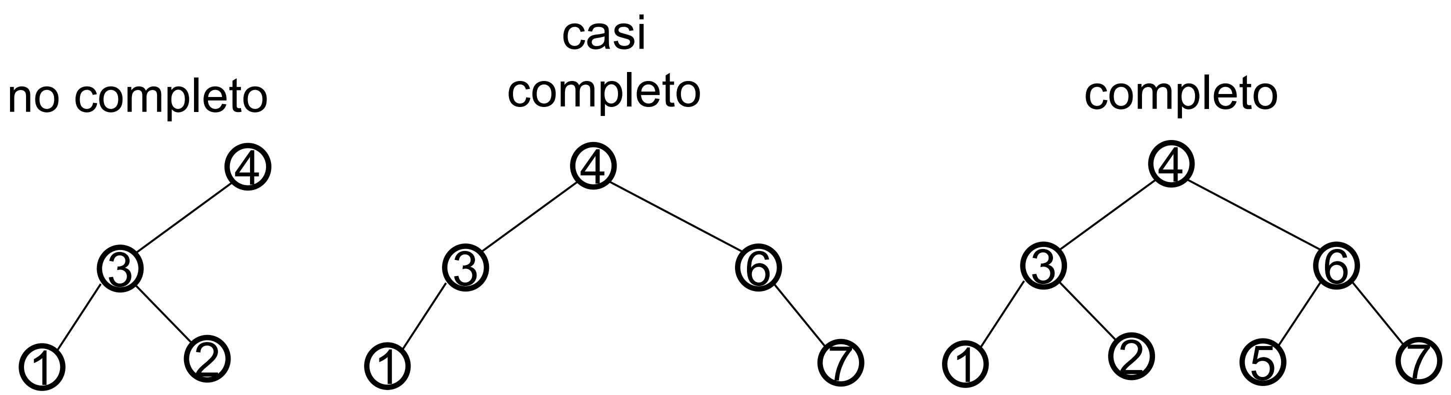


→ profundidad <u>mínima</u> necesaria para albergar **n** nodos:

$p = prof(T) = \lceil \log_2(n + 1)) - 1 \rceil$

Árboles binarios. Completitud

- **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** ≤ **p** están completos, i.e. tienen 2^d nodos
  - (casi completo y tiene exactamente 2^p hojas a profundidad **p**)



Contenidos

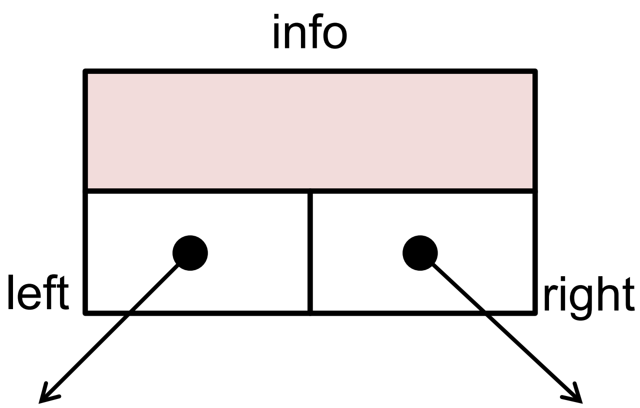
Árboles binarios. Implementación en C

- Estructura de datos de un nodo de un árbol binario

```
// En tree.c

#define info(pnode)  ((pnode)->info)
#define left(pnode)  ((pnode)->left)
#define right(pnode)  ((pnode)->right)

struct _BTNode {
   void *info;
   struct _BTNode *left;
   struct _BTNode *right;
};
```



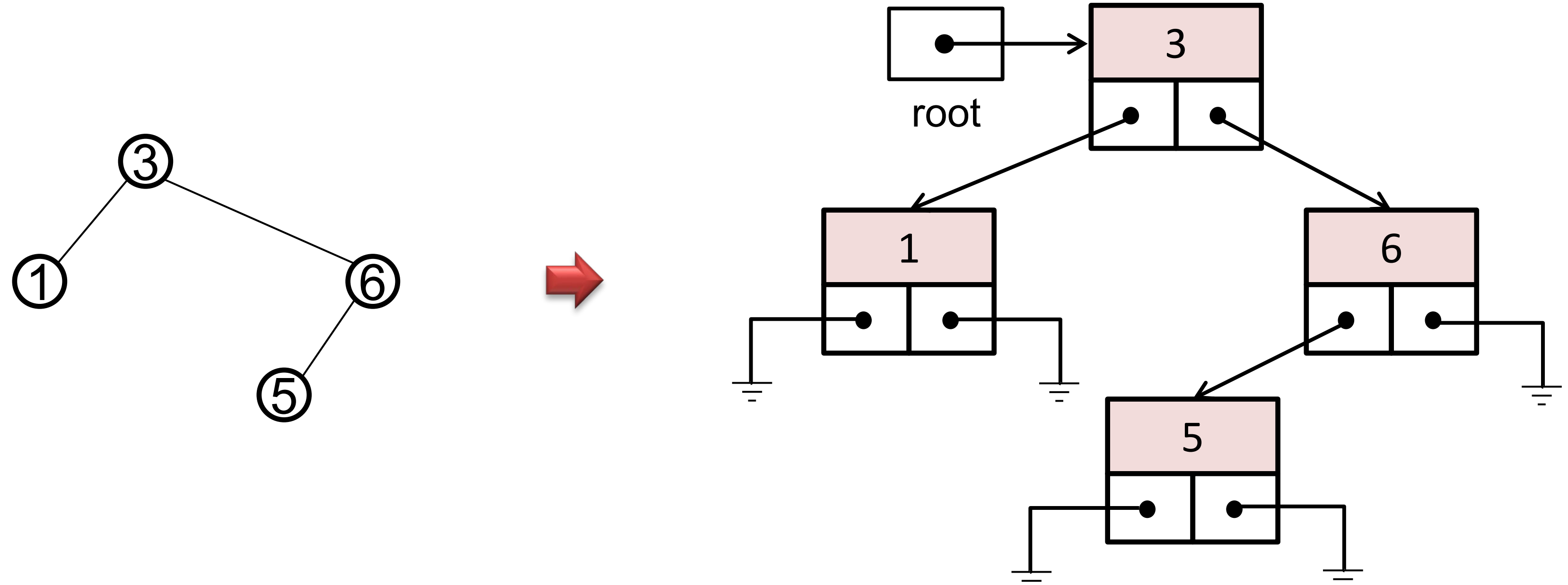

```
// También en tree.c! Los nodos son privados
typedef struct _BTNode BTNode;
```

Árboles binarios. Implementación en C

- Estructura de datos de un árbol binario

```
// En tree.c - privado
#define root(pt) ((pt)->root)

struct _BSTree {
    BTNode *root;    // un árbol es el puntero a su nodo raíz
};
// En tree.h - público
typedef struct _BSTree BSTree;
```



Árboles binarios. Implementación en C

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

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

// Libera memoria de un nodo
void bt_node_free(BTNode *pn);
```

Árboles binarios. Implementación en C


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

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

   info(pn) = left(pn) = right(pn) = NULL;

   return pn;
}

void bt_node_free(BTNode *pn) {
   free(pn);
}
```

Árboles binarios. Implementación en C

- Primitivas del TAD árbol binario

```
// Reserva memoria e inicializa un árbol
BSTree *bt_new();

// Libera la memoria de un árbol y todos sus nodos
void bt_free(BSTree *pt);

// Indica si un árbol tiene algún nodo o no
boolean bt_isEmpty(BSTree *pt);
```

Árboles binarios. Implementación en C


```
struct _BSTree {
   BTNode *root;
};
```

```
BSTree *bt_new() {
   BSTree *pa = NULL;

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

   root(pa) = NULL;

   return pa;
}
```

Árboles binarios. Implementación en C


```
struct _BSTree {
   BTNode *root;
};
```

```
Boolean bt_isEmpty(BSTree *pa) {
   if (!pa) {
      return TRUE;
   }

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

- Implementar la función **bt_free**

Árboles binarios. Implementación en C

- La **liberación de un árbol** se realiza usando la propiedad recursiva de los árboles
  - 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 del hijo derecho, y luego se libera la raíz

- Idea de liberación: recorrido *postorden* desde el nodo raíz, considerando como *visita* de un nodo su liberación

```
bt_free(BTNode T) {
   bt_free(left(T))
   bt_free(right(T))
   bt_node_free(root(T)) // visita de la raíz = liberación de la
}                        // raíz
```

Árboles binarios. Implementación en C


```
// Función públicamente declarada en tree.h
void bt_free(BSTree *pa) {
   if (!pa) return;

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

   free(pa);
}

// Función privada en tree.c
void bt_recFree(BTNode *pn) {




}
```

Árboles binarios. Implementación en C

```
// Función públicamente declarada en tree.h
void bt_free(BSTree *pa) {
   if (!pa) return;

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

   free(pa);
}

// Función privada en tree.c
void bt_recFree(BTNode *pn) {
   if (!pn) return; // caso base de la recursion: árbol vacío

   bt_recFree(left(pn));   // Liberación de subárbol izquierdo
   bt_recFree(right(pn)); // Liberación de subárbol derecho
   bt_node_free(pn);       // visitar nodo = liberar nodo
}
```

Árboles binarios. Implementación en C

- **¿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

Contenidos

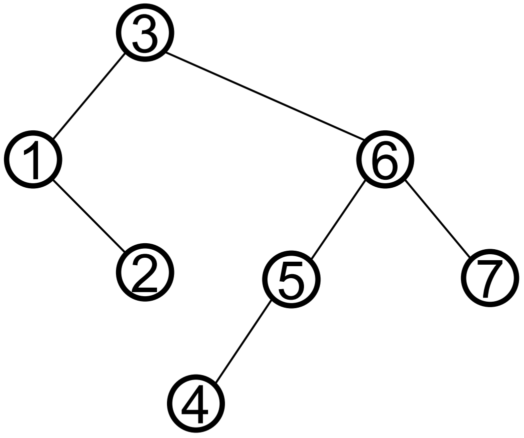
Árboles binarios de búsqueda. Definición

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

  info(T') es mayor que la info de TODOS los nodos de la izquierda y menor que la info de TODOS los nodos de la derecha

  dado un criterio de ordenación para los info(T), que vendrá dado por una primitiva `element_compare(e,e')` para comparar dos elementos del árbol

  - 1,2 < 3
  - 3 < 4,5,6,7
  - 4,5 < 6
  - …



Árboles binarios de búsqueda. Definición

Observaciones:

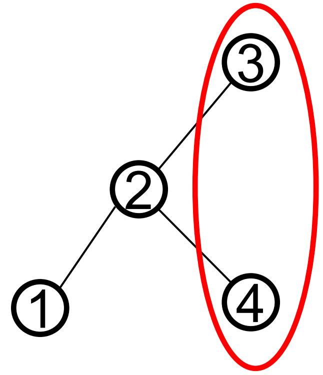
- Cada subárbol de un ABdB es a su vez un ABdB
- Para que se cumpla la condición: Para todo sub-árbol T' de T

  info(T') es mayor que la info de TODOS los nodos de la izquierda y menor que la info de TODOS los nodos de la derecha

  NO basta:

  para todo nodo N de T, info(left(N)) < info(N) < info(right(N))

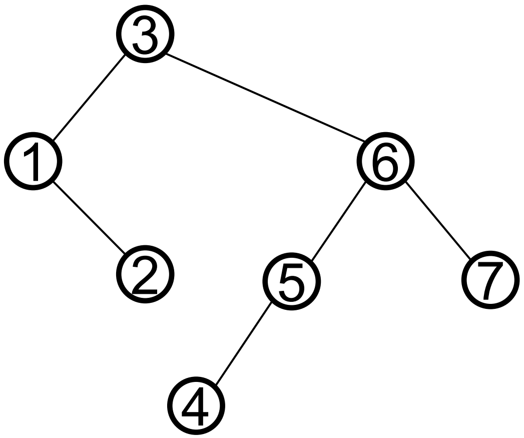
  Contraejemplo:



Satisface la segunda condición pero no la primera

Árboles binarios de búsqueda. Orden medio

- **Recorrido en <u>orden medio</u> de un ABdB**



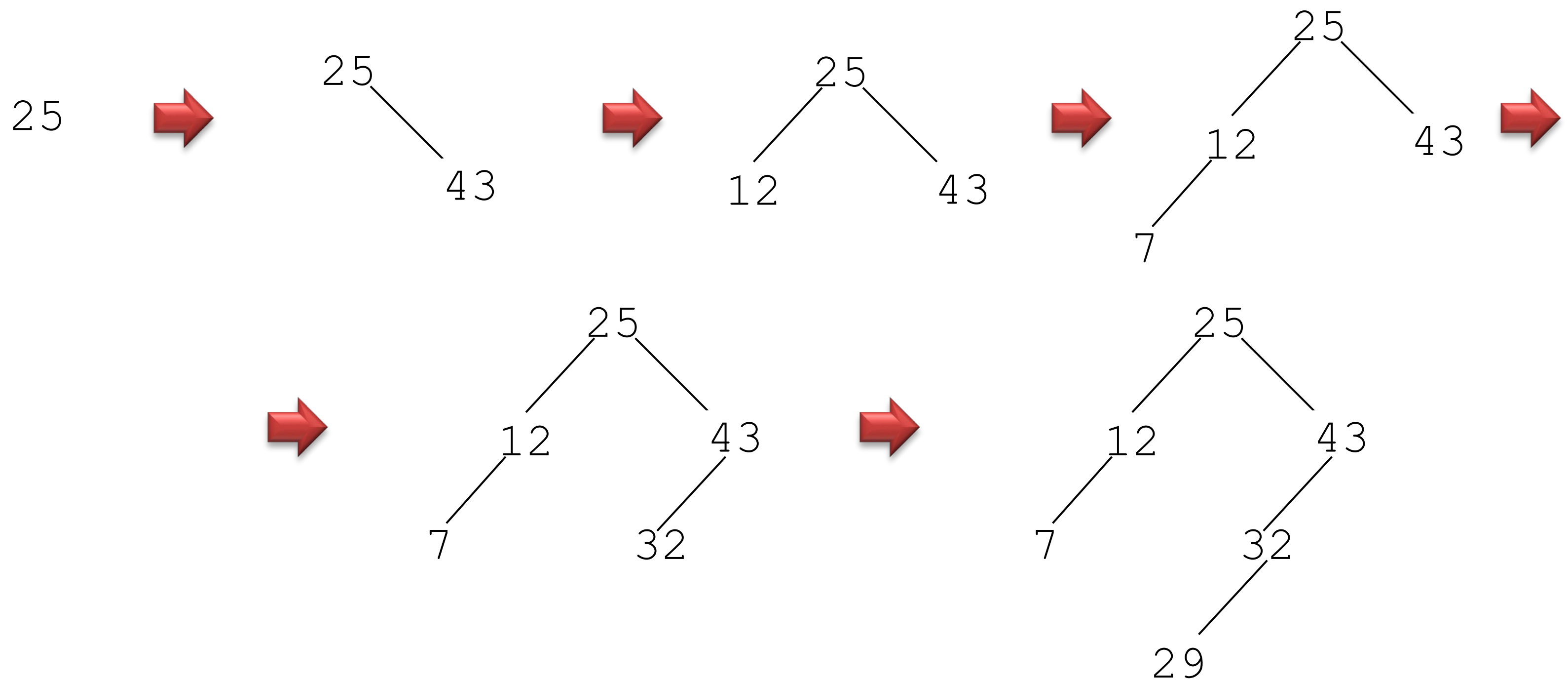
- Salida → 1 2 3 4 5 6 7
- ¡Listado ordenado de los nodos!

Contenidos

Árboles binarios de búsqueda. Creación

- **Creación de ABdB**: inserción de valores uno a uno en ABdB parciales
- Ejemplo

25 43 12 7 32 29



Árboles binarios de búsqueda. Inserción

- Árbol Binario de Búsqueda: Binary Search Tree o BST

La función `bst_insert(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 `bst_insert(left(T),e)`
  - Si **e** > info(**T**), entonces se ejecuta `bst_insert(right(T),e)`
  - Si **e** = info(**T**), no se hace nada (el dato ya está insertado y no vamos a considerar repeticiones).

Árboles binarios de búsqueda. Inserción

Este pseudocódigo se puede refinar de dos maneras, que darán lugar a dos implementaciones alternativas

- Versión 1: la función recursiva `bst_insert` retorna el subárbol resultante de la inserción
- Versión 2: la función recursiva retorna un Status, modificando de alguna manera el subárbol que se le pasa como argumento

Árboles binarios de búsqueda. Inserción

Pseudocódigo versión 1:

```
BinaryTree bst_insert(BinaryTree T)
```

- Si **T** está vacío, se crea un nodo con **e** y se devuelve ese nodo (es decir, el árbol que consiste en ese único nodo)
- Si no, se hace una llamada recursiva a insertar
  - Si **e** < info(**T**), entonces left(**T**) **=** bst_insert (left(**T**), e)
  - Si **e** > info(**T**), entonces right(**T**) **=** bst_insert(right(**T**), e)
  - Si **e** = info(**T**), no se hace nada
  - Devolver **T**

Es decir, se asigna directamente al hijo correspondiente de T el resultado de insertar **e** en el subárbol con raíz en ese hijo, y se devuelve el árbol T modificado

Árboles binarios de búsqueda. Inserción

- Pseudocódigo versión 1, más refinado

```
BinaryTree bst_insert(BinaryTree T, Element e)
   if bt_isEmpty(T):      // Caso base: árbol vacío
      T = bt_node_new()   // creamos un nuevo árbol con e
      if T == NULL:
         return ERROR
      info(T) = e
      return T            // devolvemos el nuevo árbol
   // Caso general: árbol no vacío: modificar el subárbol
que corresponda
   if e < info(T)
      left(T) = bst_insert(left(T), e)
   else if e > info(T)
      right(T) = bst_insert(right(T), e)
   // ignorar caso e = info(T)
   return T               // devolvemos el árbol T, modificado
```

Árboles binarios de búsqueda. Inserción

Implementación en C

- Asumimos una función de comparación entre elementos

```
int element_compare(void *el1, void *el2)
```

- Esta función seguirá la convención de C para todas las funciones de comparación:
  - retorna 0 si los dos elementos son iguales
  - retorna un número negativo si `el1 < el2`
  - retorna un número positivo si `el1 > el2`
- Como regla mnemotécnica para recordar esta convención, es lo que devolvería la resta de dos enteros
  - 3 < 5 => 3 – 5 = - 2 negativo
  - 6 > 2 => 6 – 2 = 4 positivo
  - 3 = 3 => 3 - 3 = 0 iguales

Árboles binarios de búsqueda. Inserción

- Implementación en C

```
Status bst_insert(BSTree *pa, const void *pe) {      // Función pública
    BTNode *pret;
    if (!pa || !pe) return ERROR;
    pret = bt_insertRec(root(pa), pe);
    if (pret == NULL) return ERROR;
    root(pa) = pret;
    return OK;
}
```

Igual que antes, necesitamos una función no recursiva que toma como argumento un BSTree*, y que llama a una función recursiva que opera sobre BTNode*

```
BTNode *bst_insertRec(BTNode *pn, const void *pe){
    //Función privada, devuelve puntero al subárbol resultante de la inserción
    int cmp;
    if (pn == NULL) {            //Caso base: árbol vacío - se ha encontrado el lugar donde insertar
        pn = bt_node_new();
        if (pn == NULL) return NULL;
        pn->info = (void *) pe;
        return pn;               // Devolvemos el nuevo nodo (= un subárbol de un solo nodo)
    }
    // Si todavía no se ha encontrado el hueco donde insertar, insertaremos pe en el subárbol
   // izquierdo ó derecho, según corresponda:
    cmp = element_compare(pe, info(pn));
    if (cmp < 0)
        left(pn) = bst_insertRec(left(pn), pe);
    else if (cmp > 0)
        right(pn) = bst_insertRec(right(pn), pe);
    return pn;                   // Devolvemos el propio pn, con uno de sus subárboles ya modificado
}
```

①

②

③

Árboles binarios de búsqueda. Inserción

- Implementación en C

```
Status bst_insert(BSTree *pa, const void *pe) {     // Función pública
    BTNode *pret;
    if (!pa || !pe) return ERROR;
    pret = bt_insertRec(root(pa), pe);
    if (pret == NULL) return ERROR;
    root(pa) = pret;
    return OK;
}
BTNode *bst_insertRec(BTNode *pn, const void *pe){
    //Función privada, devuelve puntero al subárbol resultante de la inserción
    int cmp;
    if (pn == NULL) {        //Caso base: árbol vacío - se ha encontrado el lugar donde insertar
        pn = bt_node_new();
        if (pn == NULL) return NULL;
        pn->info = (void *) pe;
        return pn;           // Devolvemos el nuevo nodo (= un subárbol de un solo nodo)
    }
    // Si todavía no se ha encontrado el hueco donde insertar, inse
   // izquierdo ó derecho, según corresponda:
    cmp = element_compare(pe, info(pn));
    if (cmp < 0)
        left(pn) = bst_insertRec(left(pn), pe);
    else if (cmp > 0)
        right(pn) = bst_insertRec(right(pn), pe);
    return pn;               // Devolvemos el propio pn,
}
```

Igual que antes, necesitamos una función no recursiva que toma como argumento un BSTree*, y que llama a una función recursiva que opera sobre BTNode*

1

2

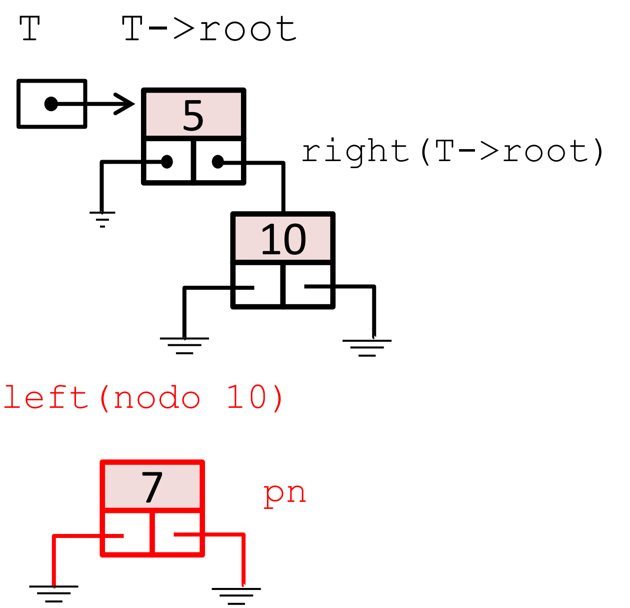
Si hubiera un error en la inserción, esto simplemente asignaría un hijo nulo a la hoja bajo la cual se tendría que haber realizado la inserción

Ejemplo:

```
bst_insertRec(T->root, elemento 7)
   como 5 < 7:
  right(T->root) -> bst_insertRec(right(T->root), 7)

    bst_insertRec(nodo 10, 7)
      como 10 > 7:
      left(nodo 10) -> bst_InsertRec(left(nodo 10),7)

      bst_insertRec(NULL, 7)
         crear nodo pn con 7 y devolverlo
```



Árboles binarios de búsqueda. Inserción

Ejemplo:

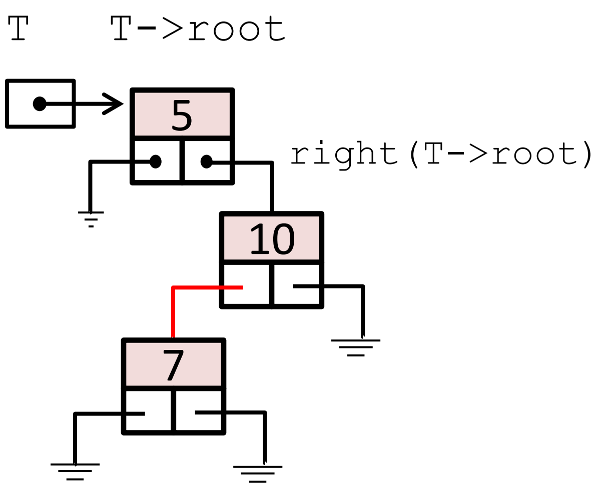
```
bst_insertRec(T->root, elemento 7)
   como 5 < 7:
  right(T->root) -> bst_insertRec(right(T->root), 7)

    bst_insertRec(nodo 10, 7)
      como 10 > 7:
      left(nodo 10) -> bst_InsertRec(left(nodo 10),7)

      bst_insertRec(NULL, 7)
         crear nodo pn con 7 y devolverlo

      left(nodo 10) = nodo 7
      devolver nodo 10

   right(nodo 5) = nodo 10
   devolver T->root (nodo 5)
```



Árboles binarios de búsqueda. Inserción

- Versión 2: la función recursiva retorna un Status, modificando "de alguna manera" el subárbol que se le pasa como argumento.
- Esto requiere usar dobles punteros, y el pseudocódigo y la implementación se detallan en el Anexo 1

Árboles binarios de búsqueda. Creación y búsqueda

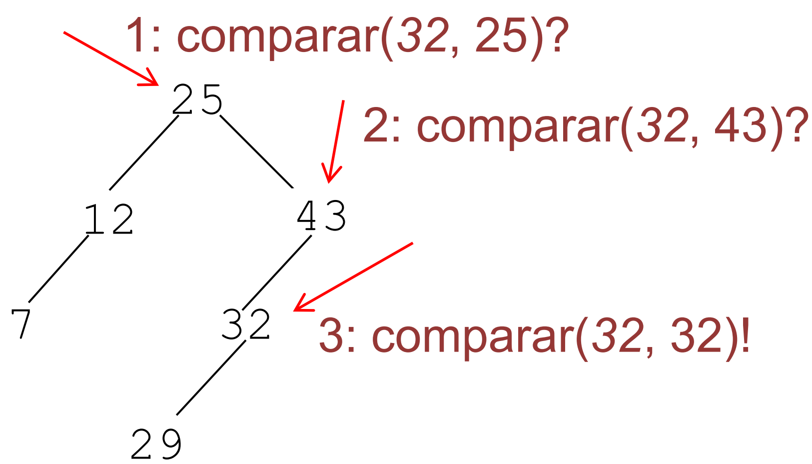
- Dado un vector de valores representados en una lista, la construcción de su correspondiente ABdB es como sigue:

```
Status bst_fromList(BSTree T, List L)
   foreach element e in L:
      st = bt_insert(T, e)
      if st = ERROR:
         bt_free(T)   // Ojo: la lista L no se ha liberado
         return ERROR
   return 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 (u ordenada) es menos eficiente, pues hay que recorrerla de forma secuencial

Árboles binarios de búsqueda. Búsqueda

- Búsqueda de un dato, p.e. 32



- ¿Búsqueda de 33?
  - Se hacen llamadas recursivas hasta llegar a un árbol vacío (el hijo derecho de 32)

Árboles binarios de búsqueda. Búsqueda

- Pseudocódigo

```
BSTree bst_search(BSTree T, Element e)
   if bt_isEmpty(T):
      return NULL
   if e == info(T):
      return T
   if e < info(T):
      return bst_search(left(T), e)
   return bst_search(right(T), e)   // e > info(T)
```

Árboles binarios de búsqueda. Búsqueda

- Problema: el pseudocódigo anterior devuelve un (sub-)árbol, pero solo el árbol completo es accesible desde la interfaz pública

- Solución:
  - la función recursiva de búsqueda (privada) devuelve un BTNode*
  - la función pública de búsqueda devuelve el **elemento** contenido en el nodo devuelto por la función recursiva

Árboles binarios de búsqueda. Búsqueda

- Implementación en C

```
void *bst_search(BSTree *pt, void *pe) {
   BTNode *found = NULL;
   if (!pt || !pe) return NULL; // error
   found = bst_searchRec(root(pt), pe);
   return (found ? info(found) : NULL);
}

BTNode *bst_searchRec(BTNode *pn, void *pe){
   int cmp = 0;
   if (!pn) return NULL; // caso base: elemento no encontrado
   cmp = element_compare(pe, pn->info);
   if (cmp == 0) return pn;  // encontrado elemento
   if (cmp < 0) return bst_searchRec(pn->left, pe);
   return bst_searchRec(pn->right, pe); // cmp > 0
}
```

Árboles binarios de búsqueda. Complejidad búsqueda

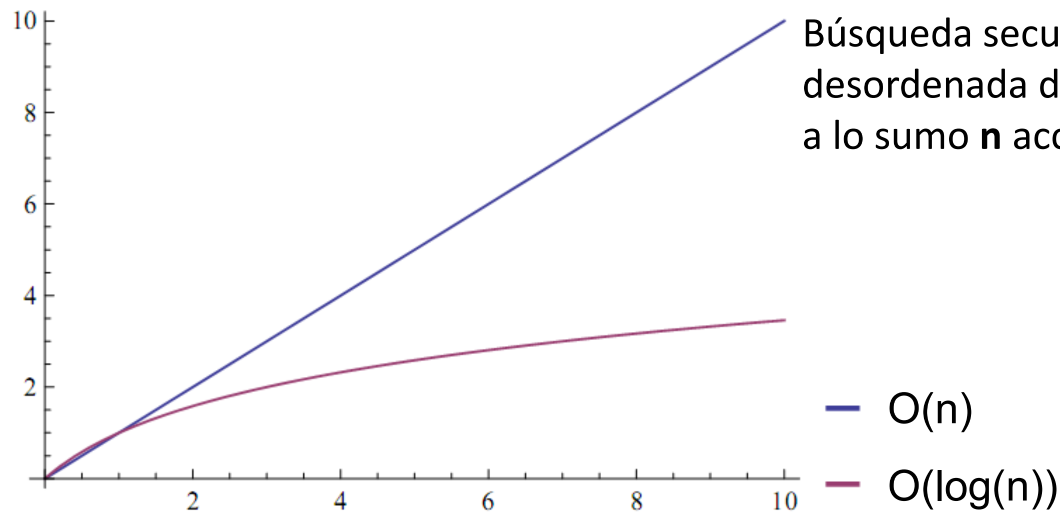
- El coste de una búsqueda es proporcional al número de nodos examinados, es decir, al número de comparaciones necesarias para encontrar un elemento

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

  - A lo sumo, una por cada nivel del árbol

    => el número de comparaciones está acotado por la profundidad del árbol

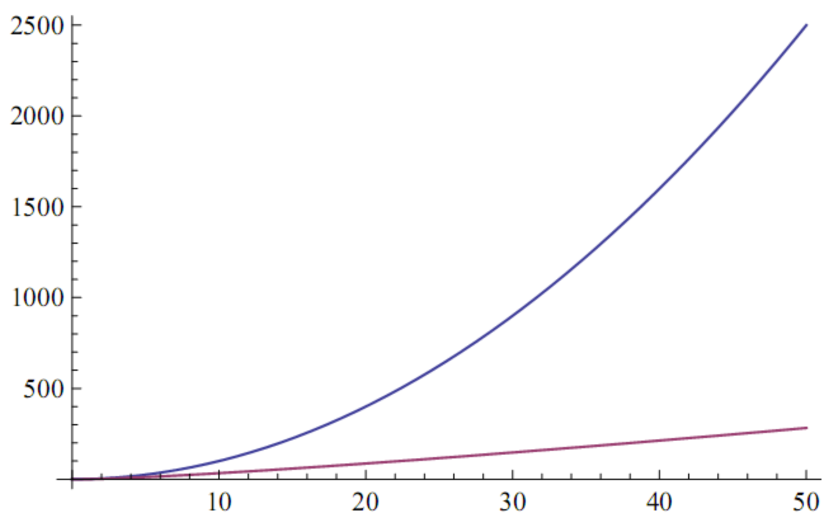
- **<u>Coste</u> (número de accesos/comparaciones) de buscar un dato en un ABdB**
- Para un ABdB con profundidad **p**:
  - a lo sumo **p + 1** accesos ≡ **O(p) en el caso general**
- Para un ABdB casi completo con **n** nodos, **O(p) = O(log n)**
  - a lo sumo tantos accesos como la profundidad del árbol ≡ $\lceil \log_2(n + 1)) - 1 \rceil$ ≡ Orden (log(n)) ≡ **O(log(n))**



Búsqueda secuencial en una lista desordenada de **n** nodos, a lo sumo **n** accesos: **O(n)**

Árboles binarios de búsqueda. Complejidad ordenación

- Para **n** elementos, hay que realizar **n** inserciones
  - Dado árbol (casi) completo, cada inserción es a lo sumo del orden de la profundidad del árbol **p** ≤ ⌈ log2(n + 1))–1⌉ ≡ orden (log(n)) ≡ O(log(n))
  - La creación del árbol es O(n·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 en orden medio ≡ O(n) → ordenación es O(n·log(n)) + O(n) ≡ **O(n·log(n))**



Ordenación de una lista de **n** nodos mediante algoritmos como BubbleSort, InsertSort: **O(n^2)**

Pero otros métodos de ordenación más eficientes son también O(n · log(n))

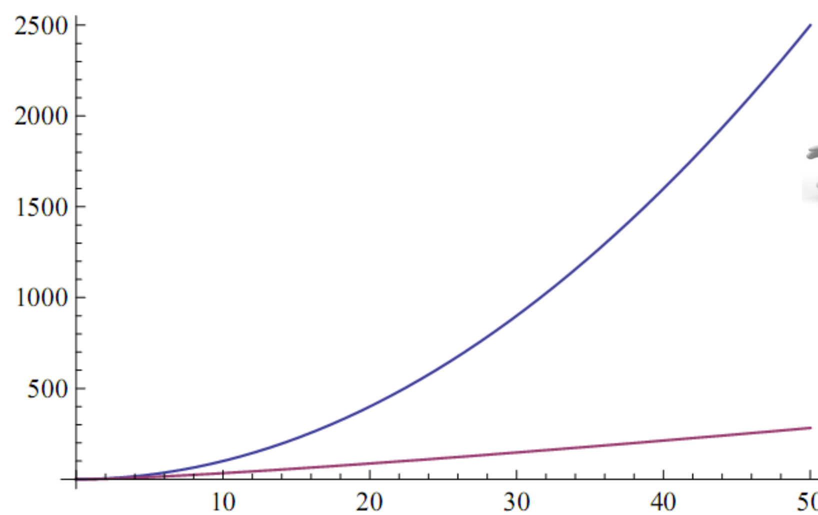
— O(n^2)

— O(n · log(n))

Árboles binarios de búsqueda. Complejidad ordenación

- Para **n** elementos, hay que realizar **n** inserciones

  - Dado árbol (casi) completo, cada inserción es a lo sumo del orden de la profundidad del árbol **p** ≤ ⌈ log2(n + 1))–1⌉ ≡ orden (log(n)) ≡ O(log(n))
  - La creación del árbol es O(n·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 ≡ O(n) → ordenación es O(n·log(n)) + O(n) ≡ **O(n·log(n))**



No es cierto **para todo** ABdB, **p** en general puede ser mayor que O(log(n))

Véase luego árboles equilibrados.

Contenidos

Árboles binarios de búsqueda. Borrado

- Para eliminar un elemento de un ABdB, necesitamos reorganizar el árbol para que siga siendo un ABdB
- Pseudocódigo

```
Status bst_delete(BSTree T, Element e)
   T’ = bst_search(T, e) // buscar nodo con elemento e
   if T’ == NULL:
      return OK // elemento no encontrado, nada que eliminar
   else // reemplazar root(T’) por otro nodo (subárbol)
      return bst_replace_root(T’)
```

Árboles binarios de búsqueda. Borrado

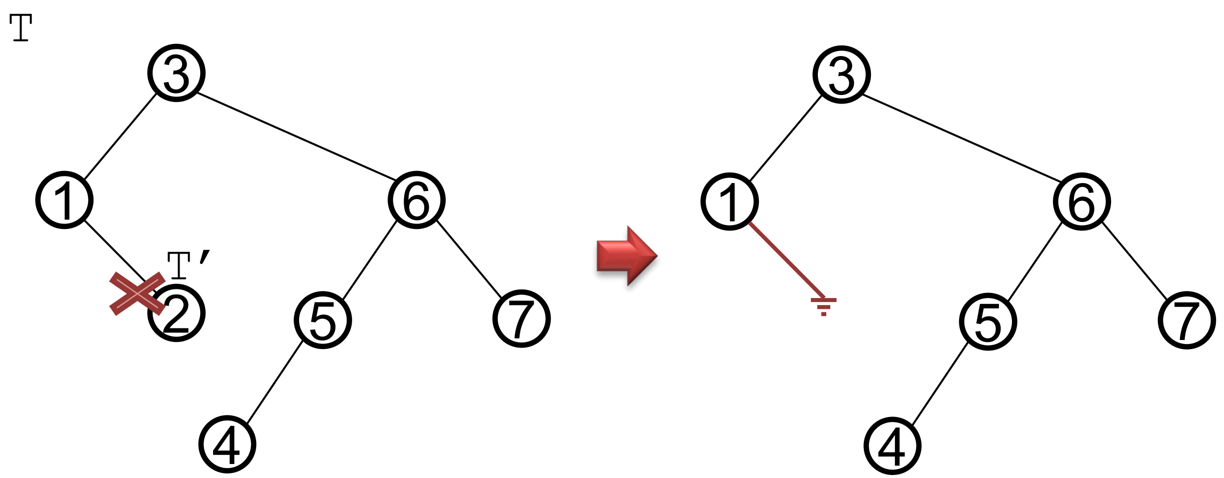
- **Reajuste** de un (sub-)árbol T' por la eliminación de su raíz (`bst_replace_root(T')`). Tres casos:
  1. La raíz de T' es hoja
  2. La raíz de T' tiene 1 hijo
  3. La raíz de T' tiene 2 hijos

Árboles binarios de búsqueda. Borrado

- **Reajuste** de un (sub-)árbol T' por la eliminación 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

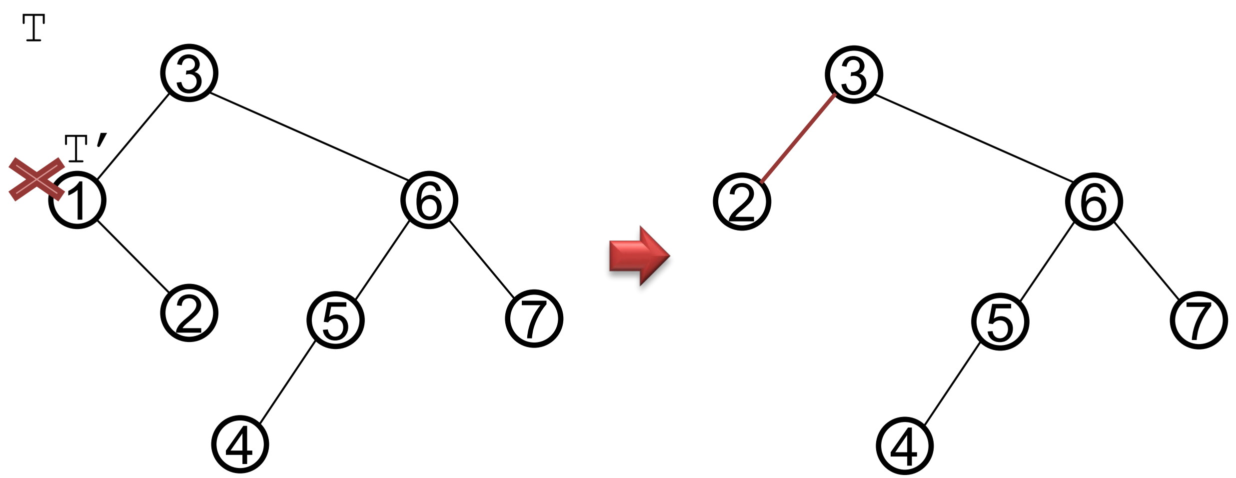
Ejemplo: `bst_delete(T, 2)`



Árboles binarios de búsqueda. Borrado

- **Reajuste** de un (sub-)árbol T' por la eliminación 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 T'
  3. La raíz de T' tiene 2 hijos

  Ejemplo: `bst_delete(T, 1)`



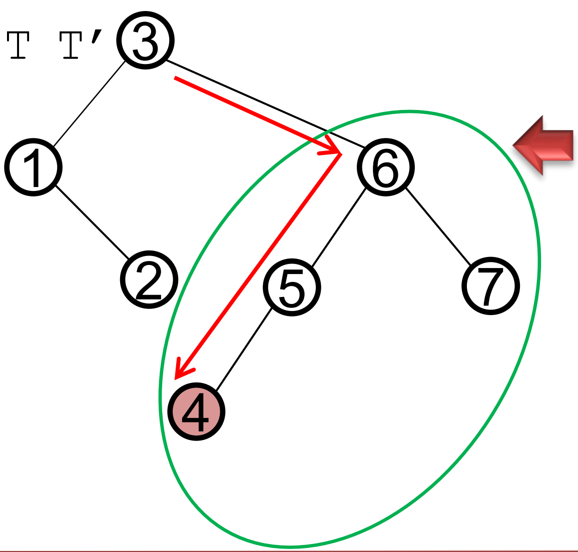
Árboles binarios de búsqueda. Borrado

- **Reajuste** de un (sub-)árbol T’ por la eliminación 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**: reemplazar T’ por su sucesor inmediato (el mínimo de su subárbol derecho, **sucesor en el orden medio)**

  - También podría usarse el predecesor, el máximo de su subárbol izquierdo

Árboles binarios de búsqueda. Borrado

- **Reajuste** de un (sub-)árbol T’ por la eliminación 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**: reemplazar T’ por su sucesor inmediato (su sucesor en el orden medio, el mínimo de su subárbol derecho)



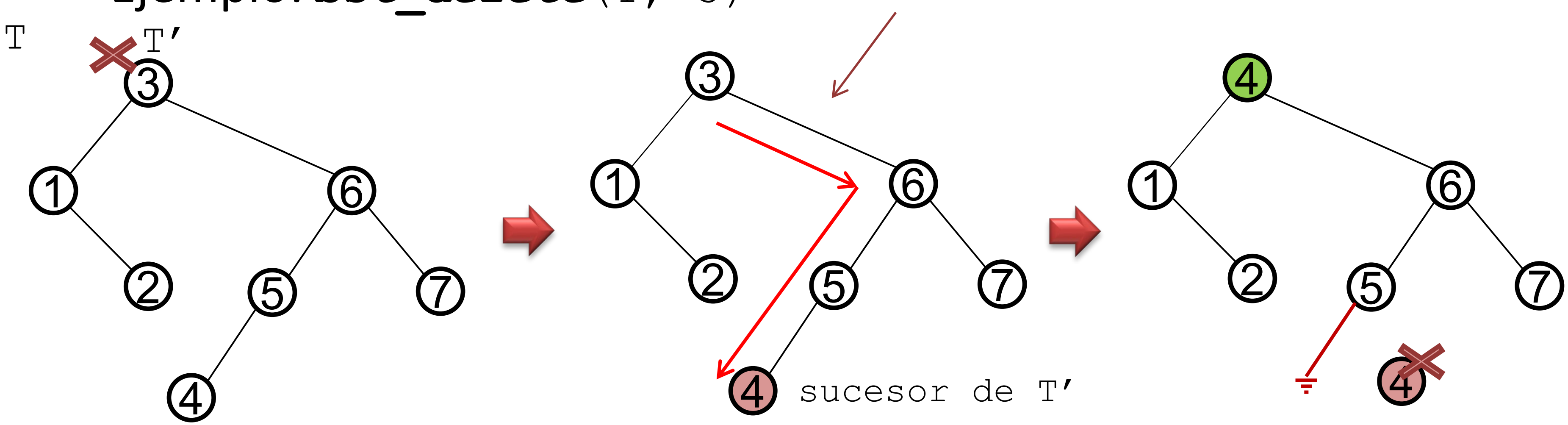
Orden medio: 1 2 3 4 5 6 7
4 es el sucesor de (siguiente a) 3 en este orden
4 es el mínimo del subárbol derecho, con raíz en 6
4 es el nodo más a la izquierda de este subárbol

```
Para obtener el sucesor de T’:
1. Bajar a la derecha de T’ un nivel
2. Bajar a continuación a la izquierda
   hasta el último nivel (nodo hoja)
```

Árboles binarios de búsqueda. Borrado

- **Reajuste** de un (sub-)árbol T’ por la eliminación 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**: reemplazar T’ por su sucesor inmediato: encontrar sucesor, guardar info del sucesor en T’, eliminar, recursivamente, el sucesor

Ejemplo: `bst_delete(T, 3)`

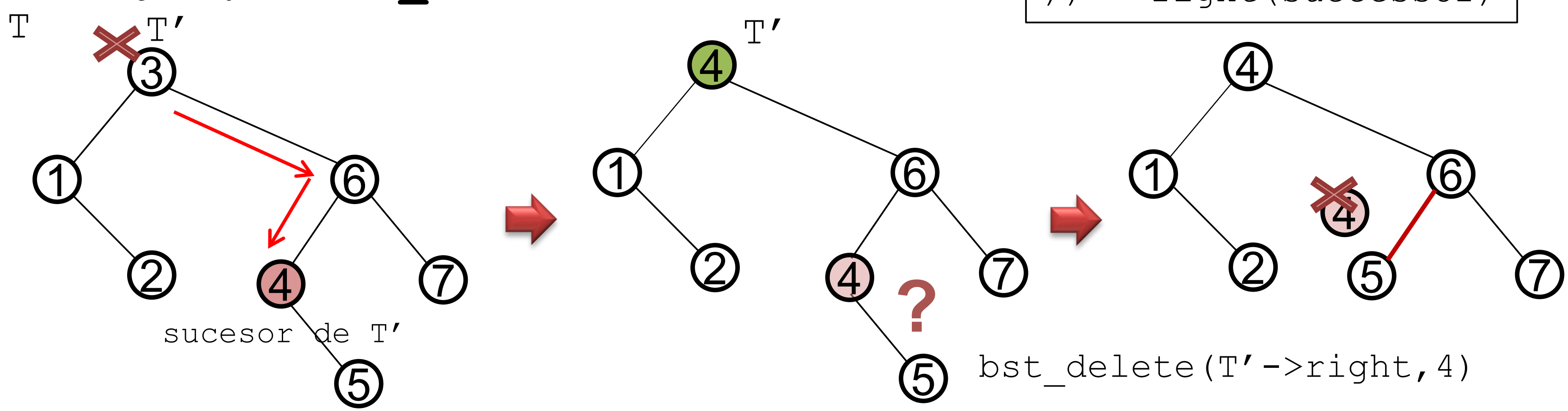


Árboles binarios de búsqueda. Borrado

- **Reajuste** de un (sub-)árbol T’ por la eliminación 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**: reemplazar T’ por su sucesor inmediato: encontrar sucesor, guardar info del sucesor en T’, eliminar, recursivamente, el sucesorc

```
Equivalente:
left(parent(successor
)) = right(successor)
```

Ejemplo: `bst_delete(T, 3)`

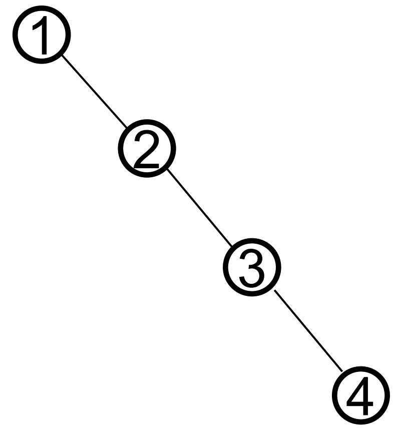


Contenidos

Árboles binarios de búsqueda. Equilibrado

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

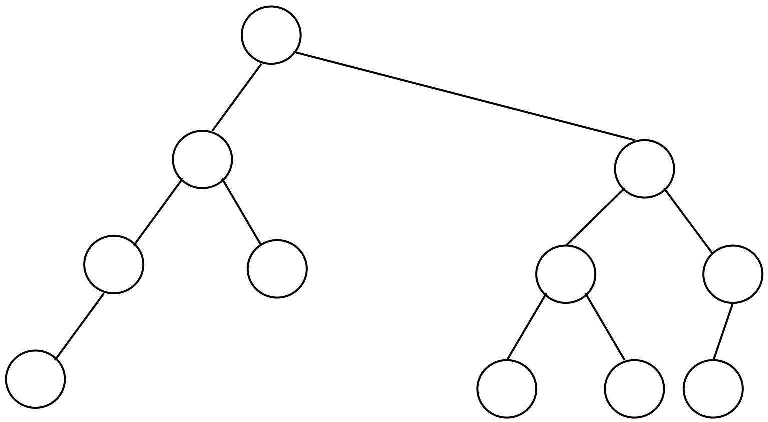
```
L = {1,2,3,4}
bst_create(T, L)
```



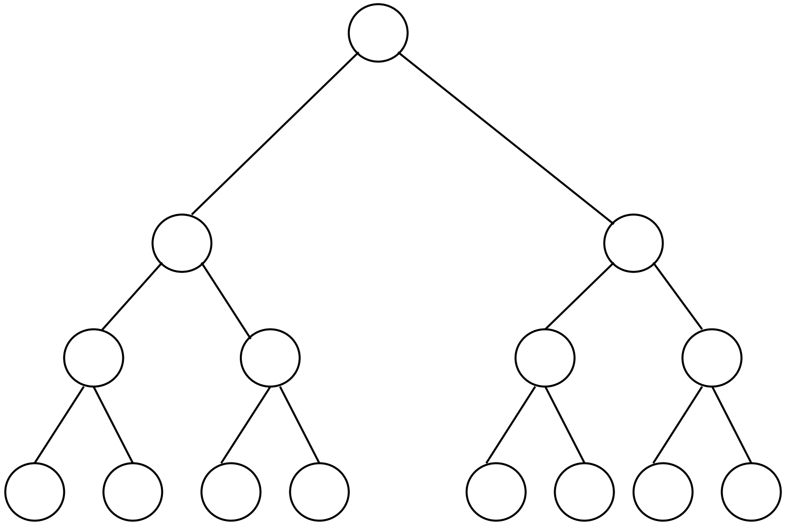
- Es como tener una lista enlazada simple de n elementos
- El acceso ya no es O(log(n)), sino O(n), por lo que:
  - Coste de la búsqueda: ya no es O(log(n)), sino O(n)
  - Coste construcción y ordenación: ya no es n·log(n), sino O(n^2)

Árboles binarios de búsqueda. Equilibrado

- Un árbol está **equilibrado** si para todo nodo la profundidad de sus sub-árboles izquierdo y derecho no difiere en más de una unidad
- Un árbol está **perfectamente equilibrado** si para todo nodo la profundidad de sus sub-árboles izquierdo y derecho es idéntica



Árbol equilibrado



Árbol perfectamente equilibrado

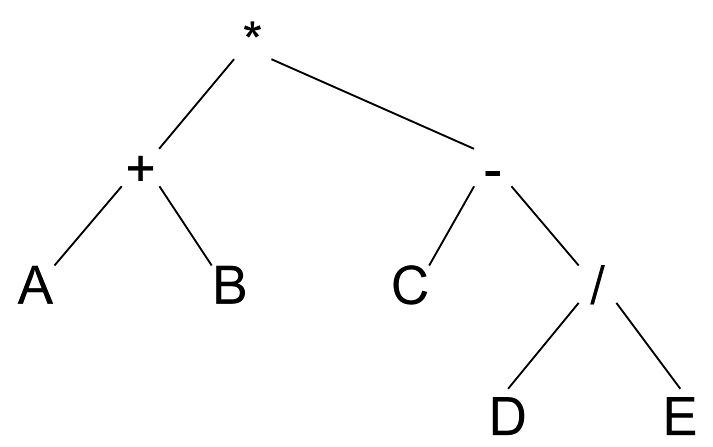
Árboles binarios de búsqueda. Equilibrado

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

Contenidos

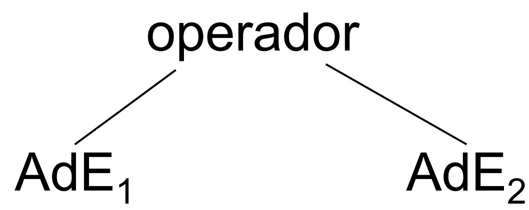
Árboles de expresión. Definición

- 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)

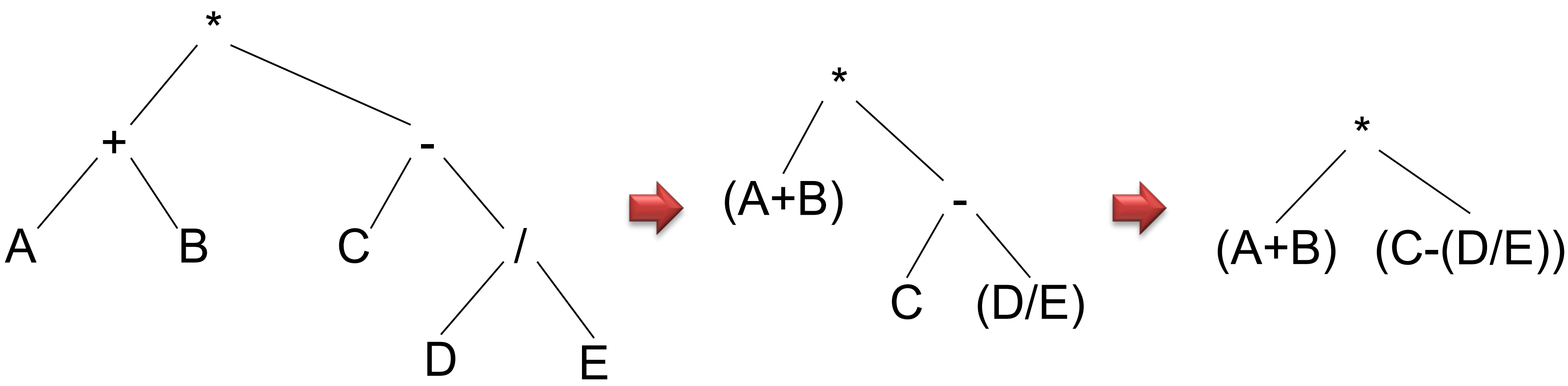


Árboles de expresión. Definición

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

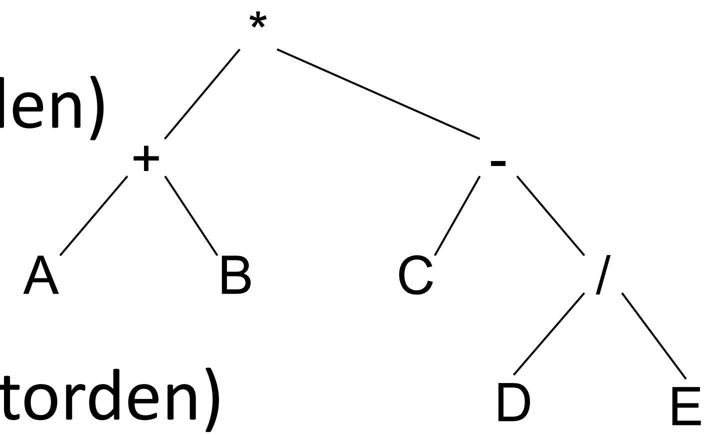


- Un AdE almacena una expresión aritmética



((A+B)*(C–(D/E)))

Árboles de expresión. Recorrido



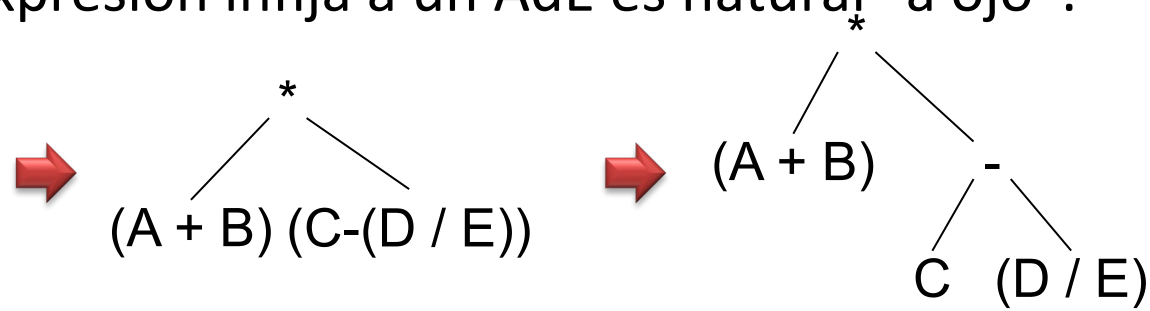
- Recorrido en orden previo (preorden)
  - Salida: * + A B – C / D E
  - Forma prefijo de la expresión
- Recorrido en orden posterior (postorden)
  - Salida: A B + C D E / - *
  - Forma postfijo de la expresión
- Recorrido en orden medio (inorden)
  - *"Imprimiendo" paréntesis al comienzo y al final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

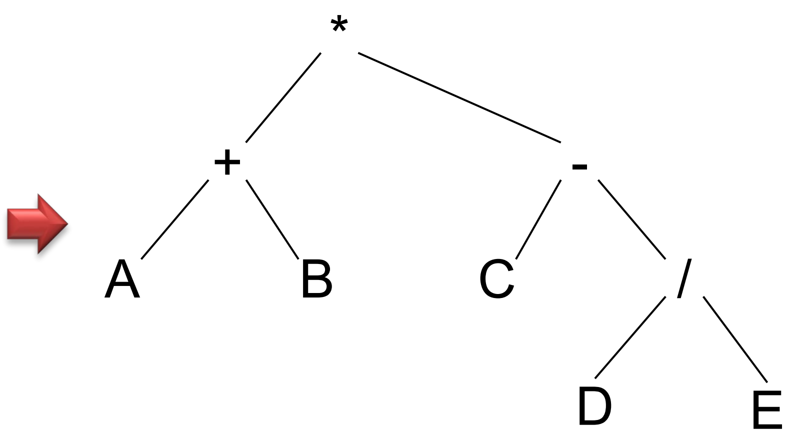
Contenidos

Árboles de expresión. Construcción

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

((A + B) * (C – (D / E)))





  - ¿Algoritmo para construirlo?

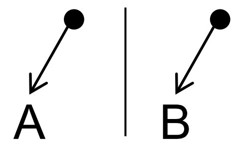
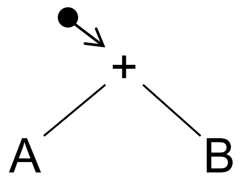
Árboles de expresión. Construcción

- Construcción de un AdE
  - Basada en la evaluación de expresiones sufijo utilizando el **TAD Pila**
  - Seguir el algoritmo de evaluación de expresiones sufijo guardando en una pila los subárboles generados para las sub-expresiones
- El algoritmo más sencillo es el que parte de expresiones postfijo → si se tiene una expresión prefijo o infijo, ésta se pasa a postfijo para evaluarla

  (A+B)*(C-D/E) → a postfijo → A B + C D E / - * → evaluación

Árboles de expresión. Construcción

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

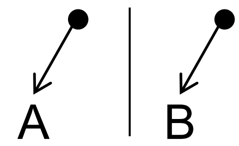
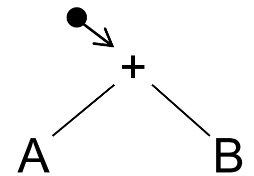
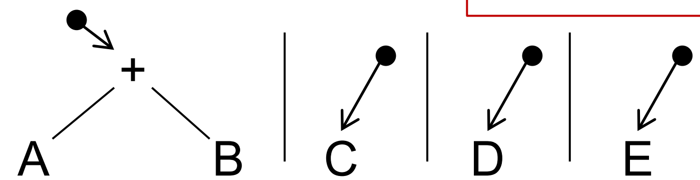
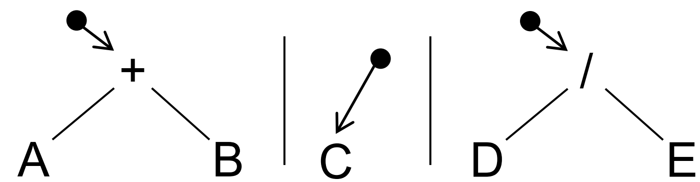
Símbolo	Pila (tope por la derecha)
A, B	
+	


```
Al leer un símbolo,
creamos un árbol con el
símbolo como único nodo y
lo metemos en la pila
```

```
Al leer un operador,
creamos un árbol cuya raíz
es el operador y cuyos
hijos son los dos
elementos superiores de la
pila. Extraemos esos dos
elementos e insertamos el
nuevo árbol en la pila
```

Árboles de expresión. Construcción

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

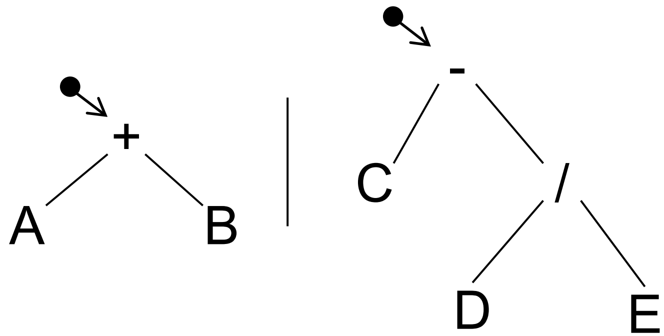
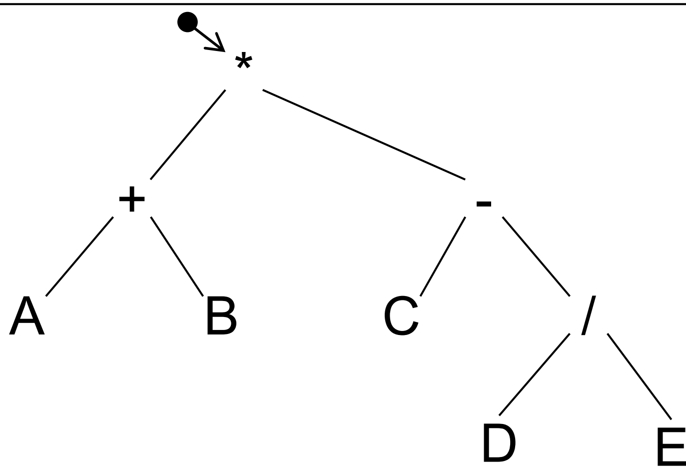
Símbolo	Pila (tope por la derecha)
A, B	
+	
C, D, E	
/	

Nuevos símbolos…

Nuevo árbol con el operador y los dos elementos superiores de la pila

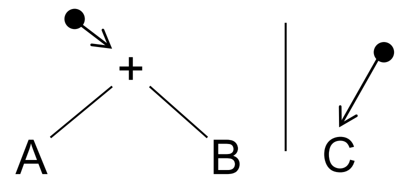
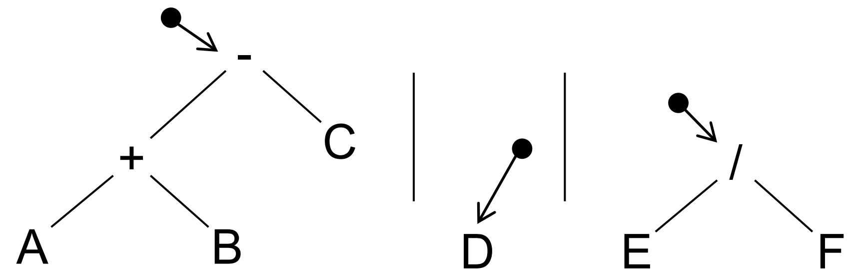
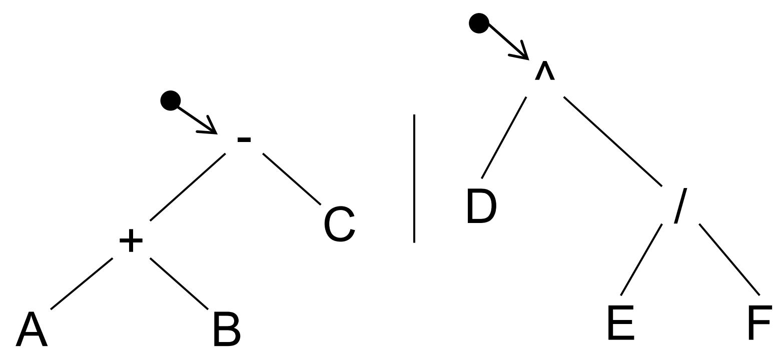
Árboles de expresión. Construcción

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

Símbolo	Pila (tope por la derecha)
-	
*	
;	Pop. Como la pila queda vacía, lo extraído es el árbol de expresión.

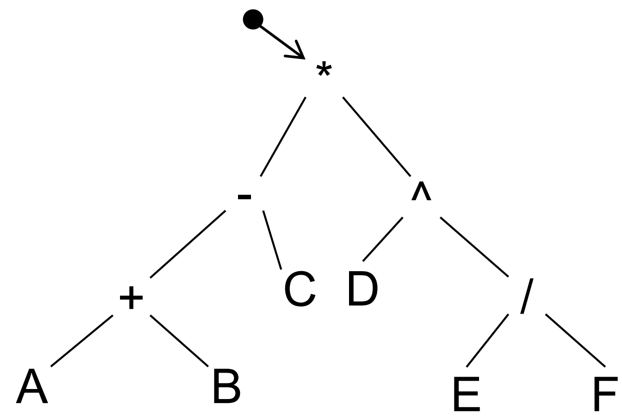
Árboles de expresión. Construcción

- Ejemplo 2: **(A + B – C) * (D ^ (E / F)) → A B + C – D E F / ^ ***

Símbolo	Pila (tope por la derecha)
A, B, +, C	
-, D, E, F, /	
^	

Árboles de expresión. Construcción

- Ejemplo 2: **(A + B – C) * (D ^ (E / F)) → A B + C – D E F / ^ ***

Símbolo	Pila (tope por la derecha)
*	
;	Se hace 1 pop y como la pila queda vacía, lo extraído es el árbol de expresión correcto.

Contenidos

Anexo 1: Método de inserción alternativo en ABdBs

Anexo 1. Inserción alternativa en ABdBs

El pseudocódigo de inserción en un ABdB se puede refinar de dos maneras, como hemos visto:

- Versión 1: la función recursiva `bst_insert` retorna el subárbol resultante de la inserción

- Versión 2: la función recursiva retorna un Status, modificando de alguna manera el subárbol que se le pasa como argumento

Árboles binarios de búsqueda. Inserción

- Pseudocódigo alternativa versión 2

```
Status bst_insert(BSTree T, Element e)
   if bt_isEmpty(T):     // Caso base
      T = bt_node_new()
      if T == NULL:
         return ERROR
      info(T) = e
      return OK
   if e == info(T): // usando element_compare
      return OK;     // El dato ya está en el árbol
   // Caso general
   if e < info(T)
      return bst_insert(left(T), e)
   return bst_insert(right(T), e) // e > info(T)
```

Árboles binarios de búsqueda. Inserción

- Implementación en C (versión doble puntero, alternativa)

```
Status bst_insert(BSTree *pa, const void *pe) {        // Función pública
    if (!pa || !pe) return ERROR;
    return bst_recInsert(&root(pa), pe);
}

Status bst_recInsert(BTNode **ppn, const void *pe) {        // Función privada
    int cmp;

    if (*ppn == NULL) {    //Encontrado lugar donde insertar: nodo nuevo apuntado por *ppn
        *ppn = bt_node_new();
        if (*ppn = NULL)  return ERROR;
        (*ppn)->info = (void *) pe;
        return OK;
    }
   // Si todavía no se ha encontrado el hueco donde insertar, buscarlo en subárbol
   // izquierdo ó derecho, según corresponda:
    cmp = element_compare(pe, info(*ppn));
    if (cmp < 0)
        return bst_recInsert(&left(*ppn), pe); //retorna Status, no un subárbol!
    if (cmp > 0)
        return bst_recInsert(&right(*ppn), pe); //retorna Status, no un subárbol!
    return OK;    // Solo se sale por aquí si el elemento ya estaba en el árbol (cmp = 0)
}
```

Árboles binarios de búsqueda. Inserción

¿Por qué usamos un doble puntero `BTNode **ppn`?

Intuitivamente, necesitamos el doble puntero porque al llegar al caso base de la recursión, es decir, al subárbol (nodo) nulo, necesitamos saber **desde dónde** apuntábamos a NULL

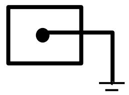
- por ejemplo desde node->right del padre

para así poder hacer que éste ahora apunte al nuevo nodo

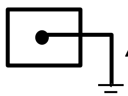
Al retornar un Status, no podemos simplemente retornar el subárbol y hacer que sea la función que llama a `bst_insertRec` la que haga esta asignación

Árboles binarios de búsqueda. Inserción

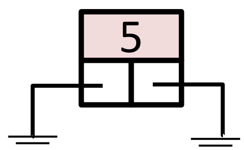
Ejemplo: Árbol vacío

T  T->root = NULL

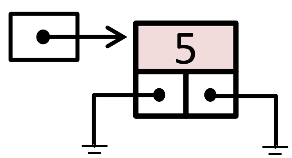
`bst_recInsert(&(T->root),pe)` (donde *pe = 5)

ppn es , con *ppn = T->root = NULL => caso base de la recursión

creamos nuevo nodo



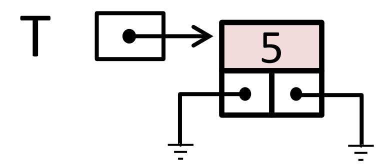
y hacemos que ppn le apunte (con la asignación *ppn = nuevo nodo)



ppn = &(T->root), así que *ppn = nuevo nodo => T-> root = nuevo nodo

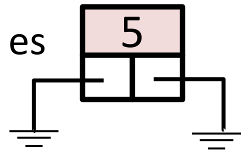
Árboles binarios de búsqueda. Inserción

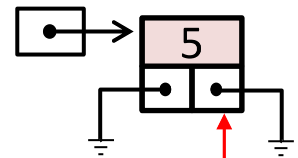
Ejemplo: Árbol con un nodo



`bst_recInsert(&(T->root),pe)` (donde *pe = 10)

ppn = &(T->root), y *ppn = T->root es



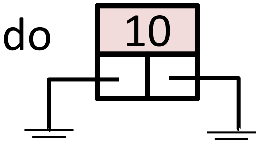


como ((*ppn)->info < *pe) llamamos recursivamente:

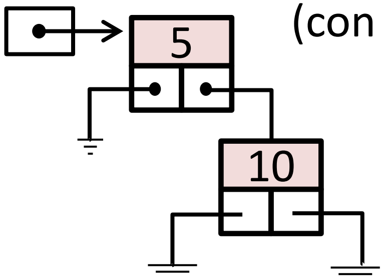
`bst_recInsert(&right(*ppn), pe)`

dentro de la llamada recursiva, ppn es la dirección del hijo derecho de 5,

y *ppn es NULL, por tanto creamos un nuevo nodo



y hacemos que ppn apunte a él



(con *ppn = nuevo nodo)

finalmente devolvemos OK en todas las llamadas hasta terminar