Programación II

Tema 5. Árboles binarios

Iván Cantador y Rosa M. Carro
Escuela Politécnica Superior
Universidad Autónoma de Madrid

Contenidos

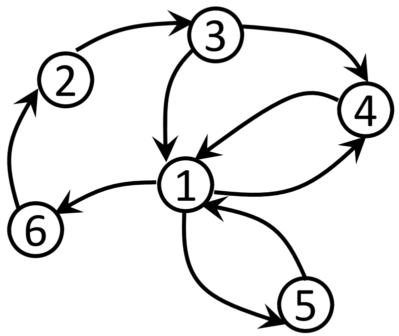
- Grafos y árboles
- Árboles binarios
  - Definición
  - Recorrido
  - Completitud
  - Implementación en C
- Árboles binarios de búsqueda
  - Definición
  - Construcción de un árbol e inserción y búsqueda de un elemento
  - Extracción de un elemento
  - Equilibrado
- Árboles de expresión
  - Definición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

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

Grafos. Definición

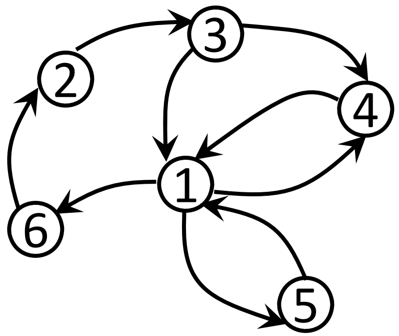
- Un **grafo** es una estructura de datos G = (V, R) compuesta de:
  - Un conjunto V de **vértices** (nodos)
  - Un conjunto R de **ramas** (arcos), conexiones entre los vértices de V
- Ejemplo de grafo (*dirigido*)



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

Grafos. Caminos

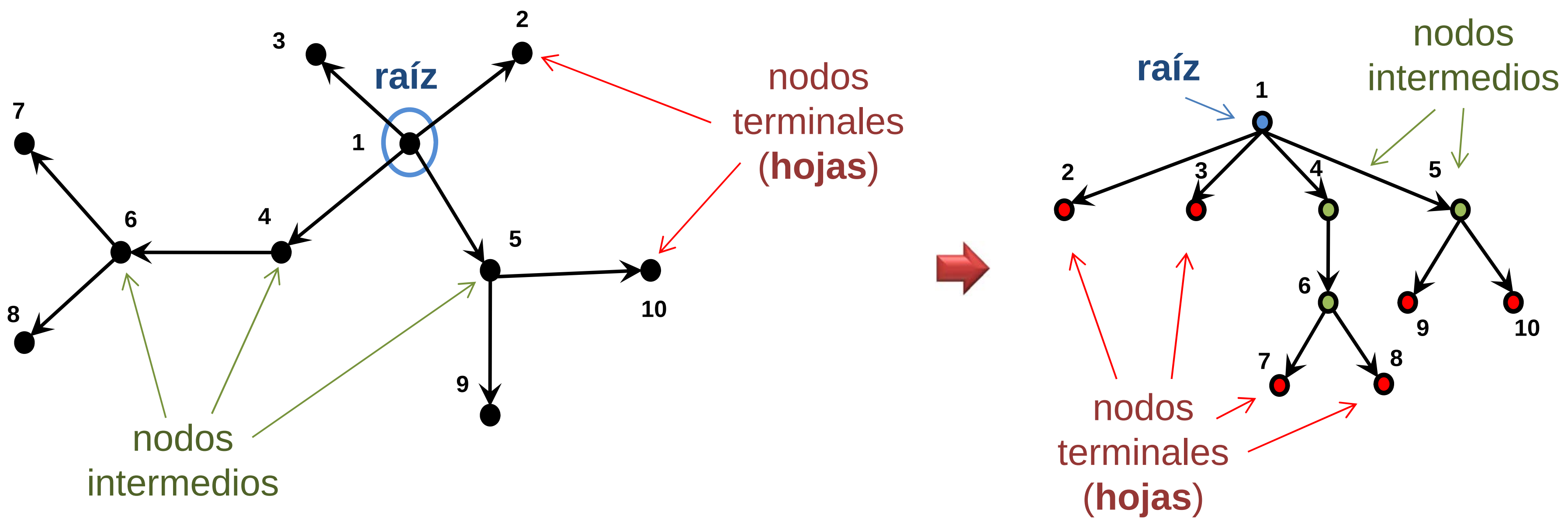
- Un **camino** de un grafo G = (V, R) es una secuencia de nodos de V en los que cada nodo es adyacente al siguiente mediante un arco de R
- Ejemplo



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

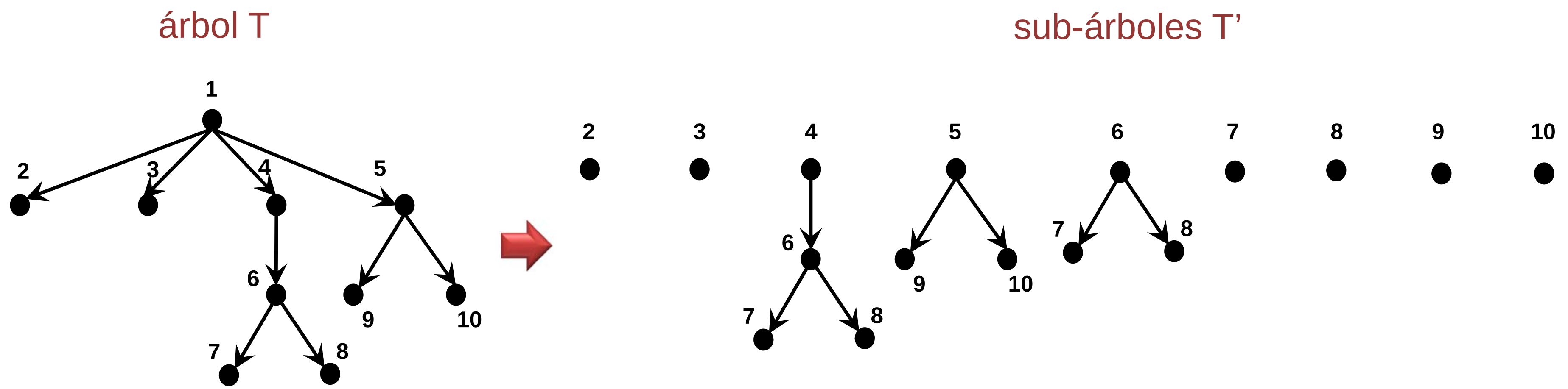
Árboles. Definición

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



Árboles. Sub-árboles

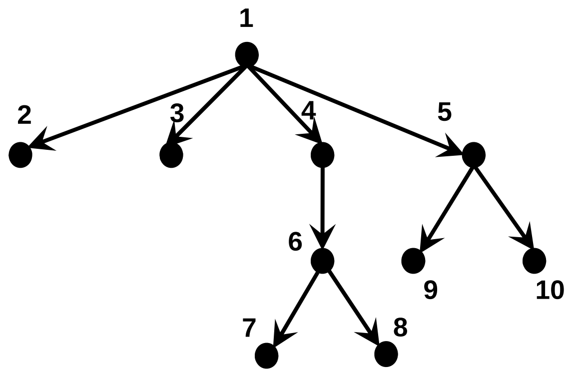
- Un **sub-árbol** de un árbol T es un subconjunto de nodos de T conectados mediante ramas de T
- Cada nodo de un árbol T junto con sus hijos da lugar a nuevo sub-árbol T'



Árboles. Nodos padre e hijo

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

T ≡



6 es padre de 7 y 8

3 es hijo de 1

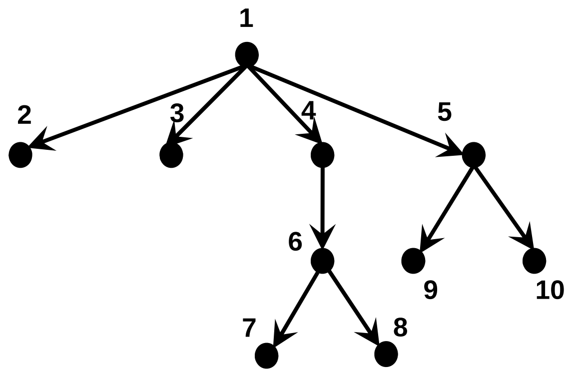
1 (la raíz) no tiene padre

2 (una hoja) no tiene hijos

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

T ≡



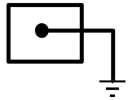
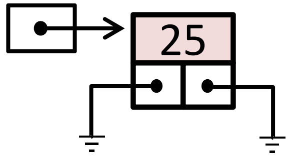
profundidad(T) = 3

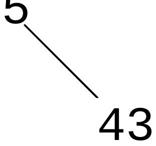
profundidad(1) = 0

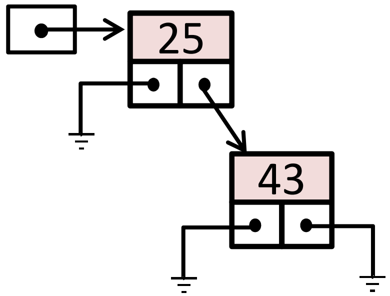
profundidad(5) = 1

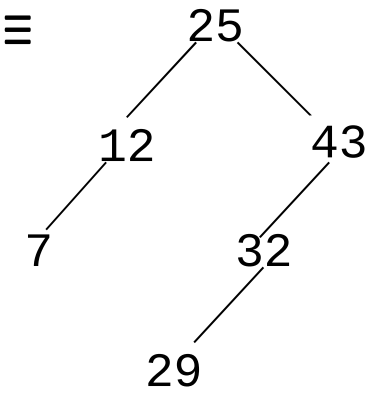
profundidad(7) = 3

Árboles. Profundidad

- **Profundidad de un árbol**
  - T ≡ Ø  profundidad(T)= -1
  - T ≡ 25 

profundidad(T)= 0
  - T ≡ 25 



profundidad(T)= 1
  - T ≡ 

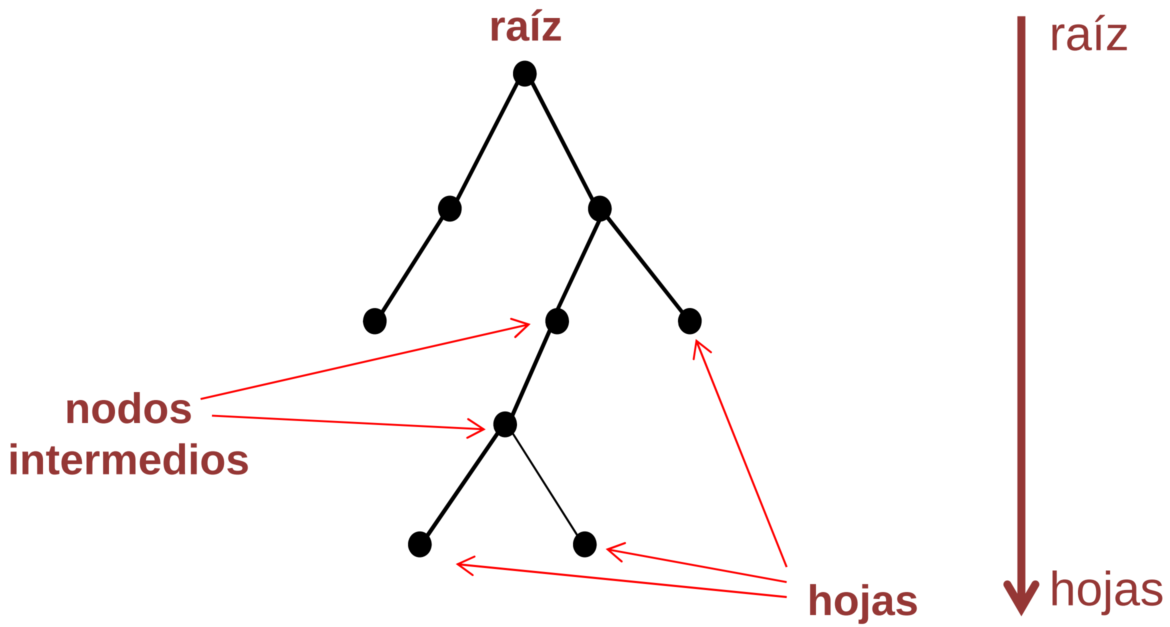
profundidad(T)= 3
(la mayor profundidad de todas las hojas)

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

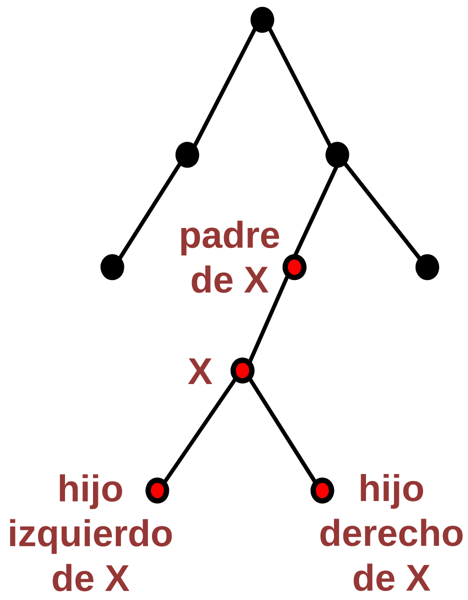
Árboles binarios. Definición

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



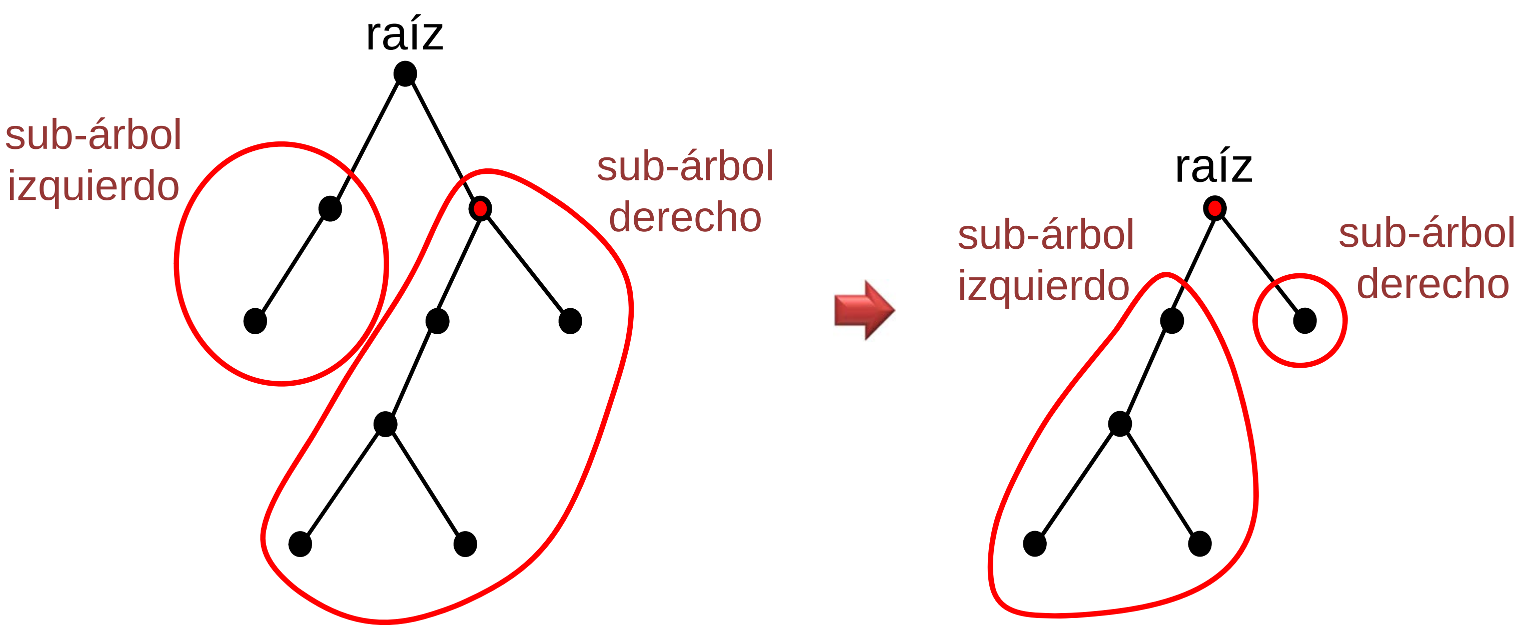
Árboles binarios. Definición

- En un AB:
  - Todo nodo excepto el 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 con dicho hijo como raíz
  - El hijo derecho de la raíz (u otro nodo) forma un nuevo árbol con dicho hijo como raíz

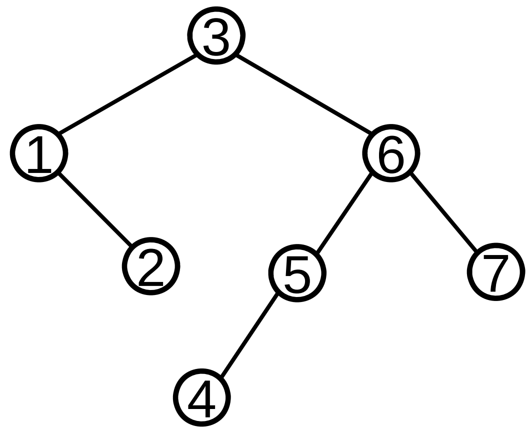


Contenidos

- 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. 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

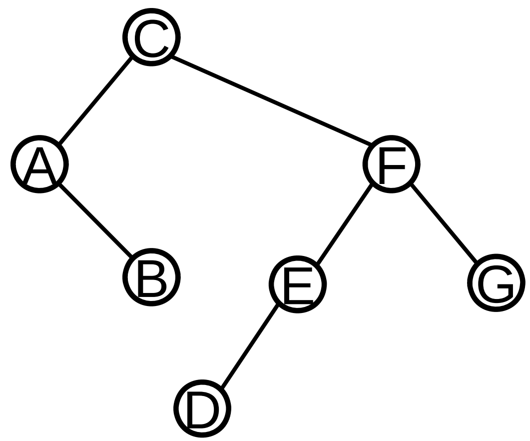
- **Recorridos en profundidad**
  - preorden, postorden, inorden
  - Ejemplo de aplicación (en grafos): encontrar componentes conexas
- **Recorrido en anchura**
  - recorrido por nivel
  - Ejemplos de aplicación (en grafos): camino más corto entre dos nodos, *crawling* Web

Árboles binarios. Recorrido en profundidad: preorden

- **Preorden** = orden previo
- Desde la raíz y recursivamente:
  1. Visitamos un 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

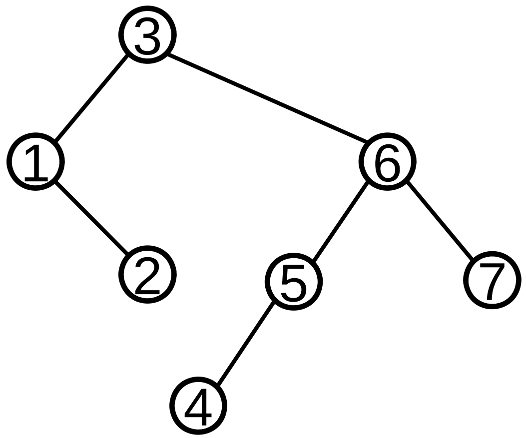
  resultado: C A B F E D G



Árboles binarios. Recorrido en profundidad: preorden

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

```
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
}
```



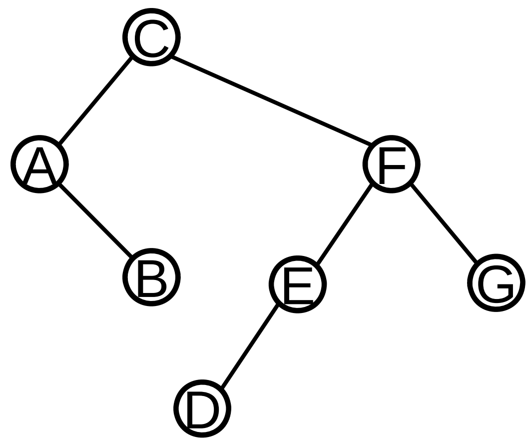
- Observaciones
  - Árbol vacío ≡ no tiene nodos
  - Árbol de un nodo ≡ un nodo raíz sin hijos
  - Asociamos nodo ≡ raíz de un subárbol; útil para la recursión

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 un nodo n
- Ejemplo

  visitar = *printf* del contenido de un nodo

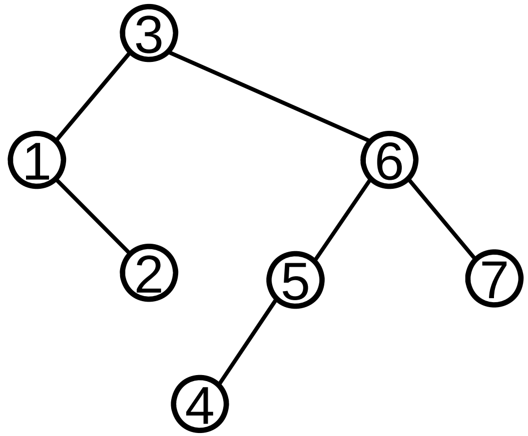
  resultado: B A D E G F C



Árboles binarios. Recorrido en profundidad: postorden

- Pseudocódigo compacto

```
ab_postorden(ArbolBinario T) {
   // Si el árbol está vacío, no hace nada, retorna.
   // Si no está vacío:
   si ab_vacio(T) = FALSE:
      ab_postorden(izq(T))
      ab_postorden(der(T))
      nodoab_visitar(T)
}
```



- Pseudocódigo más eficiente

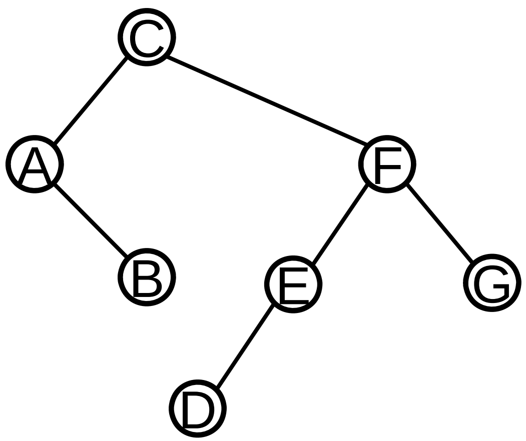
```
ab_postorden(ArbolBinario T) {
   // Si el árbol está vacío, no hace nada, retorna.
   // Si no está vacío:
   si ab_vacio(T) = FALSE:
      si ab_vacio(izq(T)) = FALSE:  //Si hijo vacío, no baja
         ab_postorden(izq(T))
      si ab_vacio(der(T)) = FALSE:
         ab_postorden(der(T))
      nodoab_visitar(T)
}
```

Árboles binarios. Recorrido en profundidad: inorden

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

  visitar = *printf* del contenido de un nodo

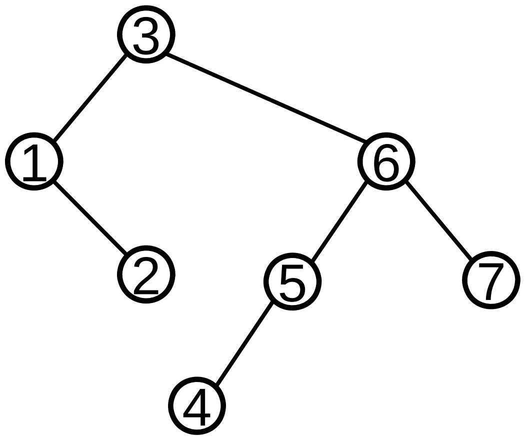
  resultado: A B C D E F G



Árboles binarios. Recorrido en profundidad: inorden

- Pseudocódigo compacto

```
ab_inorden(ArbolBinario T) {
     // Si el árbol está vacío, no hace nada, retorna.
     // Si no está vacío:
     si ab_vacio(T) = FALSE:
     ab_inorden(izq(T))
     nodoab_visitar(T)
     ab_inorden(der(T))
}
```



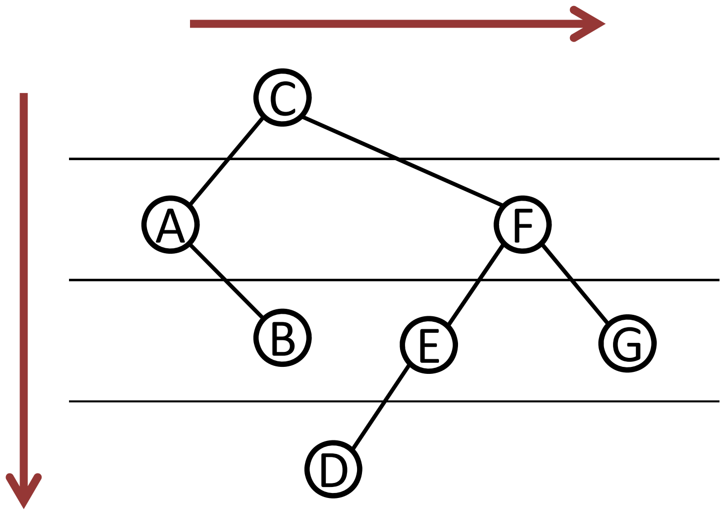
- Pseudocódigo más eficiente

```
ab_inorden(ArbolBinario T) {
     // Si el árbol está vacío, no hace nada, retorna.
     // Si no está 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))
}
```

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



Árboles binarios. Recorrido en anchura

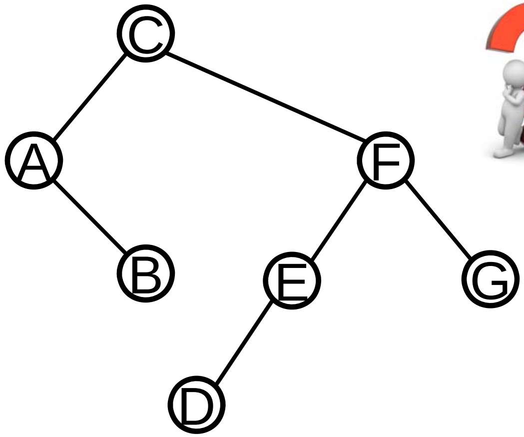
- Pseudocódigo
  - Recorre de arriba abajo y de izquierda a derecha
  - Nunca recorre un nodo de nivel *i* sin haber visitado los de nivel *i-1*

```
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)
}
```

Árboles binarios. Recorrido en anchura

- 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)
}
```



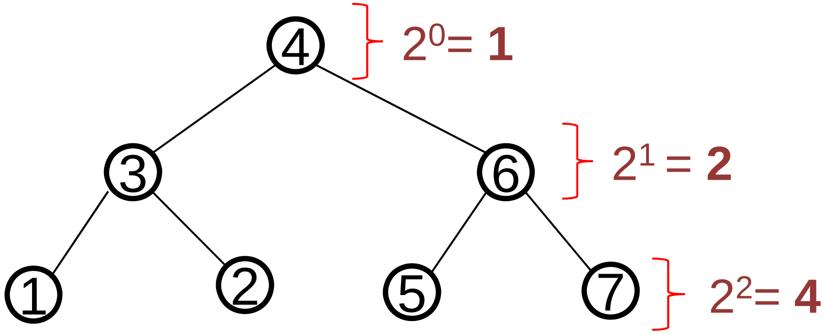
VISITAR	Q
	C
C	A F
A	F B
F	B E G
B	E G
E	7 D
G	D
D	

Contenidos

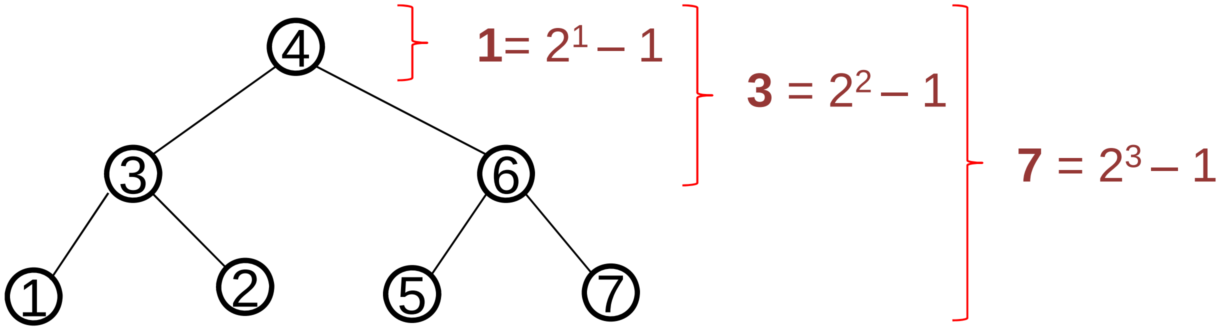
- Grafos y árboles
- **Árboles binarios**
  - Definición
  - Recorrido
  - **Completitud**
  - Implementación en C
- Árboles binarios de búsqueda
  - Definición
  - Construcción de un árbol e inserción y búsqueda de un elemento
  - Extracción de un elemento
  - Equilibrado
- Árboles de expresión
  -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

Árboles binarios. Completitud

- 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

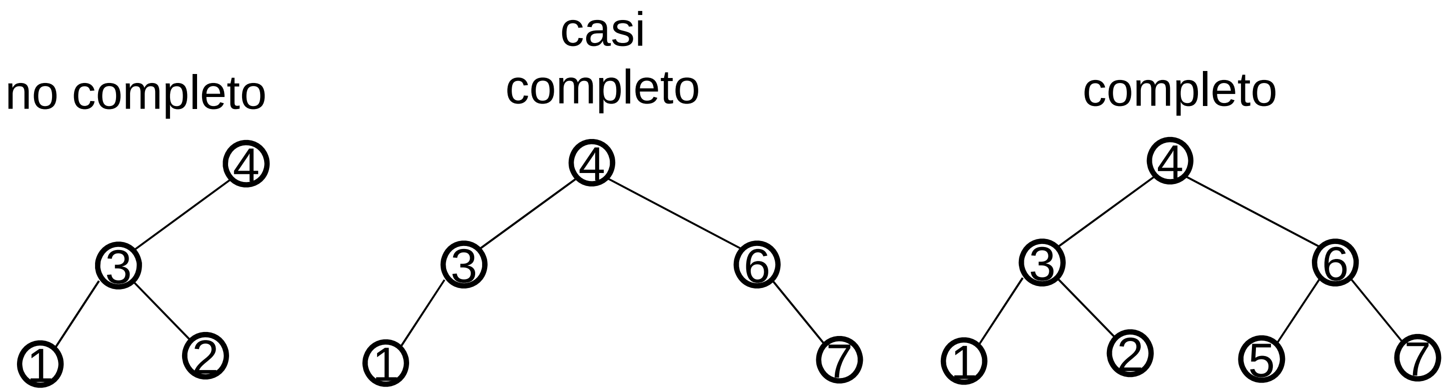


→ profundidad <u>mínima</u> necesaria para albergar **n** nodos:

$p = \mathrm{prof}(T) = [\log_2(n + 1)) - 1]$

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

- 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

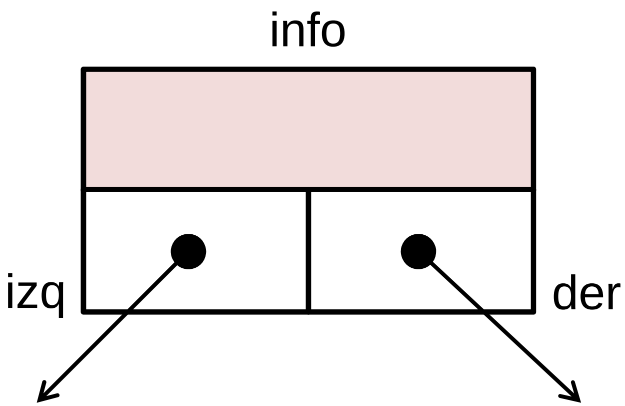
- Estructura de datos de un nodo de un árbol binario

```c
// 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;
```

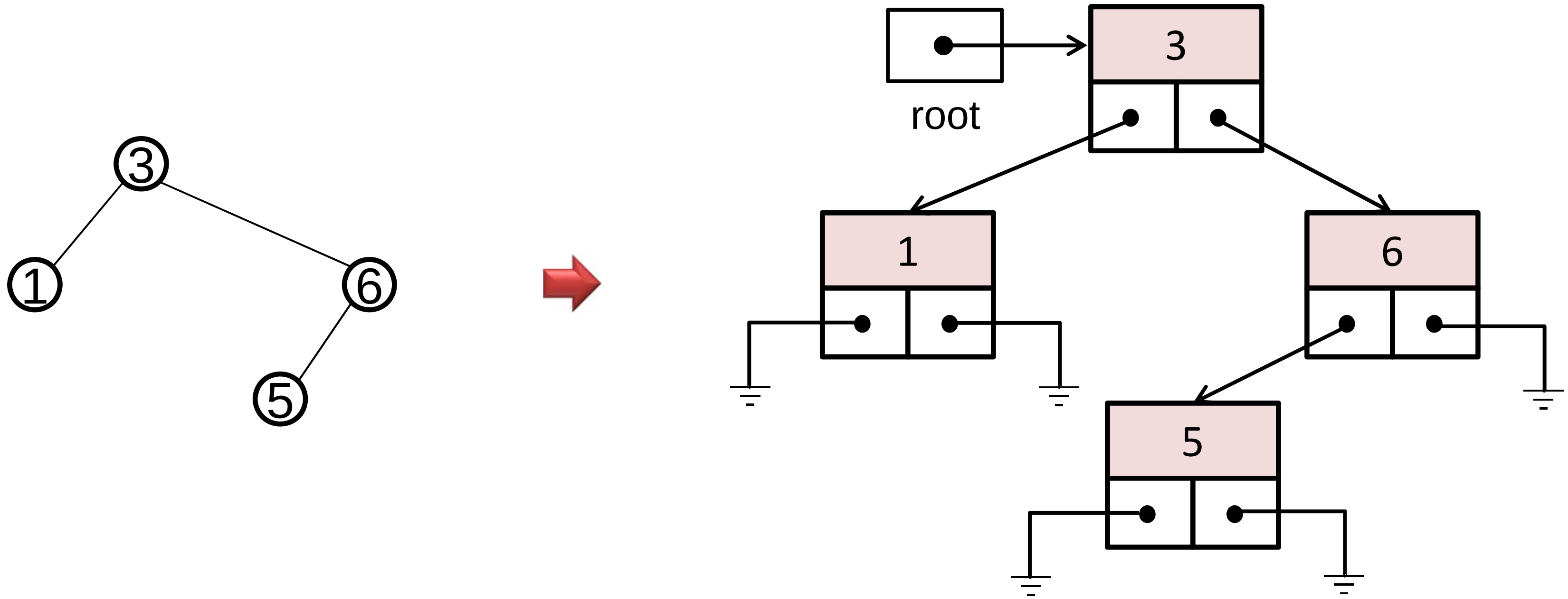


Árboles binarios. Implementación en C

- 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;
```



Árboles binarios. Implementación en C

- 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
```

Árboles binarios. Implementación en C


```
// 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);
   }
}
```

Árboles binarios. Implementación en C

- 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() {
   ArbolBinario *pa = NULL;

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

   root(pa) = NULL;

   return pa;
}
```

Árboles binarios. Implementación en C

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

```
boolean ab_vacio(ArbolBinario *pa) {
   if (!pa) {
      return TRUE;
   }

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

Árboles binarios. Implementación en C

- Implementar la función **ab_liberar**

Árboles binarios. Implementación en C

- 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

- Idea 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(raiz(T))   // visita de la raíz = liberación de la raíz
}
```

Árboles binarios. Implementación en C


```
// 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) {









}
```

Árboles binarios. Implementación en C


```
// 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
}
```

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

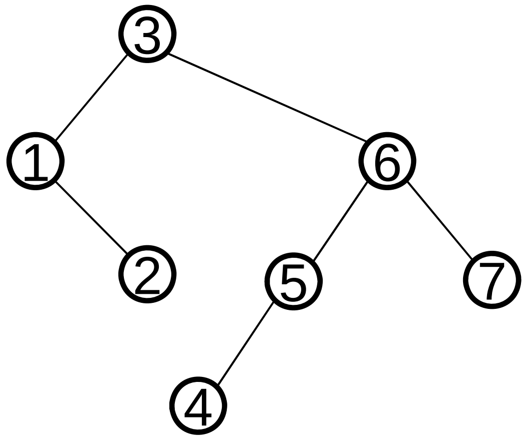
- Grafos y árboles
- Árboles binarios
  - Definición
  - Recorrido
  - Completitud
  - Implementación en C
- **Árboles binarios de búsqueda**
  - **Definición**
  - Construcción de un árbol e inserción y búsqueda de un elemento
  - Extracción de un elemento
  - Equilibrado
- Árboles de expresión
  -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

Árboles binarios de búsqueda. Definición

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

  `info(izq(T')) < info(T') < info(der(T'))`

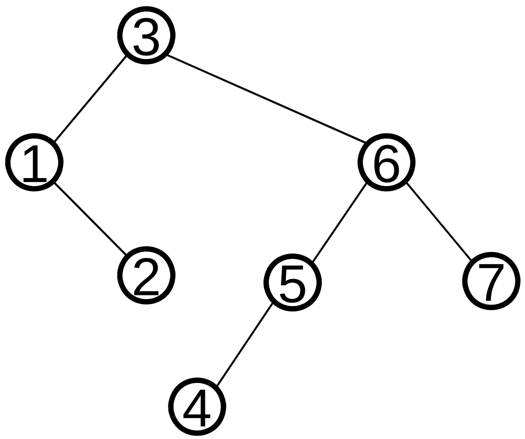
  dado un criterio de ordenación para los info(T), que vendrá dado por una primitiva **`elemento_comparar(e,e')`** del TAD Elemento



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

Árboles binarios de búsqueda. Orden medio

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



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

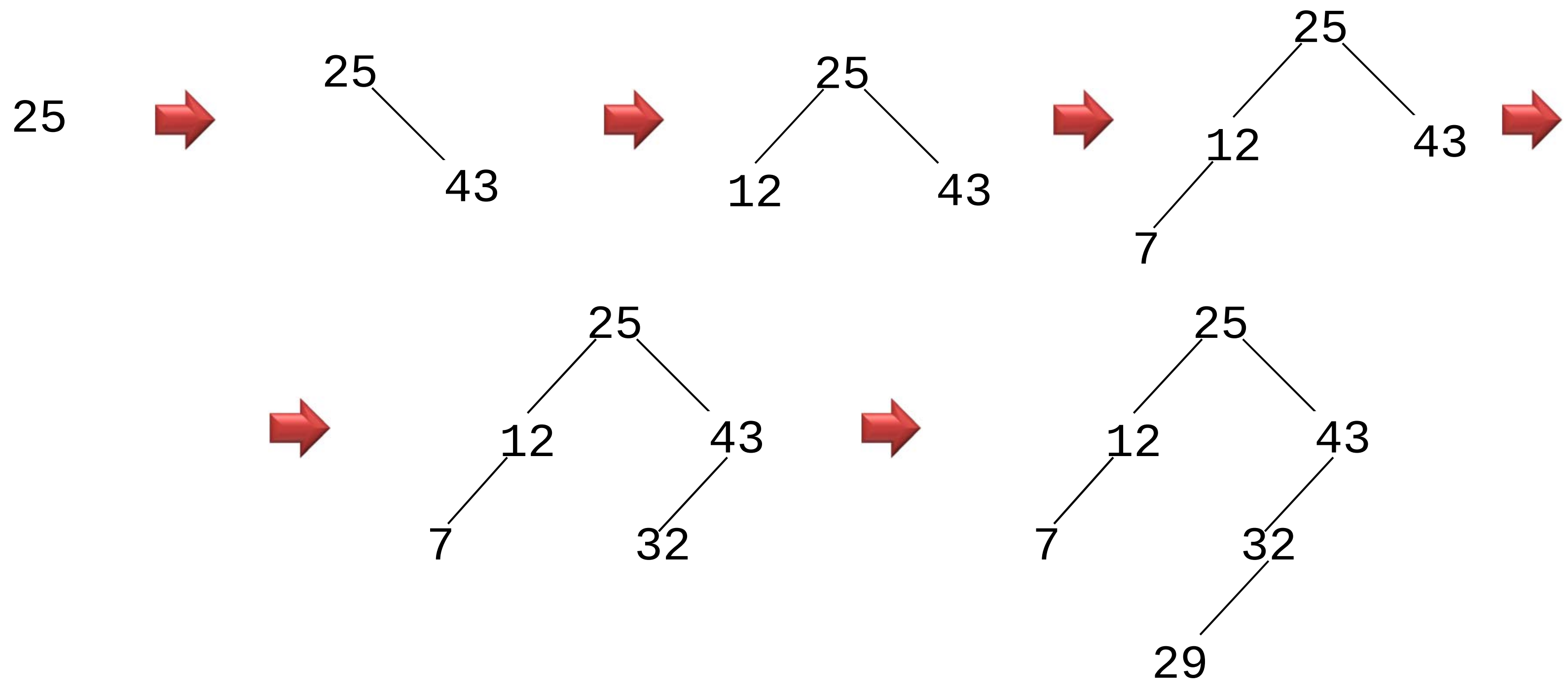
Contenidos

- 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 de búsqueda. Creación

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

25 43 12 7 32 29



Árboles binarios de búsqueda. Inserción

- 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 ignora devolviendo OK (el dato ya está insertado y no vamos a considerar repeticiones).

Árboles binarios de búsqueda. Inserción

- 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
   si e = info:
      devolver OK;  // El dato ya está en el árbol
   si no:  // Caso general
      si e < info(T)
         devolver abdb_insertar(izq(T), d)
      else
         devolver abdb_insertar(der(T), d)
```

Árboles binarios de búsqueda. Inserción

- 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 == NULL) {   //Encontrado lugar donde insertar: nodo nuevo apuntado por *ppn
        *ppn = nodoab_crear();
        if (*ppn = NULL)  return ERROR;
        if (elemento_copiar ((*ppn)-> info, pe) == NULL) {
            nodoab_liberar(ppn);
            return ERROR;
        }
        return OK;
    }
   // Si todavía no se ha encontrado el hueco donde insertar, buscarlo en subárbol
   // izquierdo ó derecho, según corresponda:
    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);
    return OK;    // Solo se sale por aquí si el elemento ya estaba en el árbol (cmp = 0)
}
```

Árboles binarios de búsqueda. Creación y búsqueda

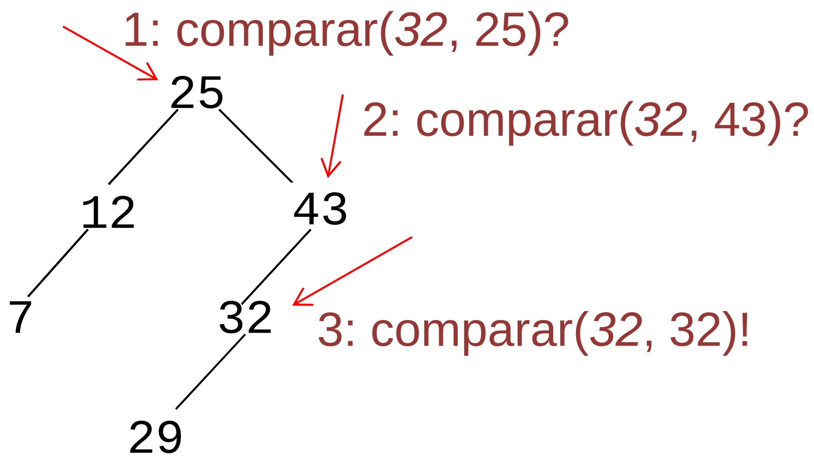
- Dado un vector de valores representados en una lista, la construcción de su correspondiente ABdB es como sigue:

```
status abdb_crear(ArbolBinario T, Lista L)
   mientras lista_vacia(L)=FALSE Y 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 (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

Árboles binarios de búsqueda. Búsqueda

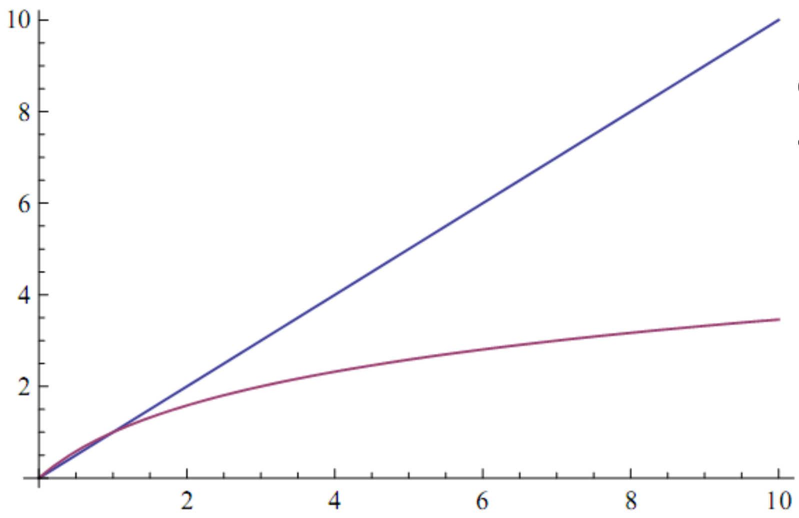
- Pseudocódigo

```
Arbol abdb_buscar(ArbolBinario T, Elemento e)
  si ab_vacio(T) = TRUE:
    devolver NULL
  si no, si info(T) = e:
    devolver T
  si no, si e < info(T):
    devolver abdb_buscar(izq(T), e)
  si no:
    devolver abdb_buscar(der(T), e)
```

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

Árboles binarios de búsqueda. Complejidad búsqueda

- **<u>Coste</u> (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 ≡ $\lceil \log_2(n + 1))-1 \rceil$ ≡ Orden (log(n)) ≡ **O(log(n))**

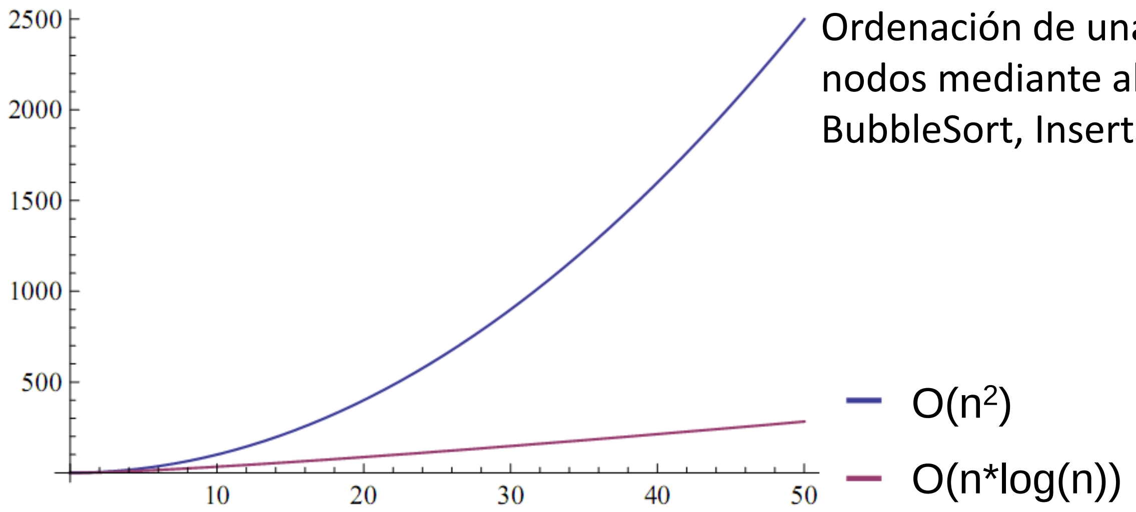


Búsqueda secuencial en una lista desordenada de **n** nodos,
a lo sumo **n** accesos: **O(n)**

— O(n)
— O(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 actual del árbol ≤ ⌈ 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))**



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

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 de búsqueda. Extracción

- Pseudocódigo

```
status abdb_extraer(ArbolBinario T, Elemento e)
   si ab_vacio(T) = TRUE:
      devolver ERROR  // No encuentra el elemento en T
   T' = abdb_buscar(T, e) // Buscar devuelve el nodo donde está e
   si T' = NULL:
      devolver OK
   si no:
      devolver abdb_reajustar(T')
```

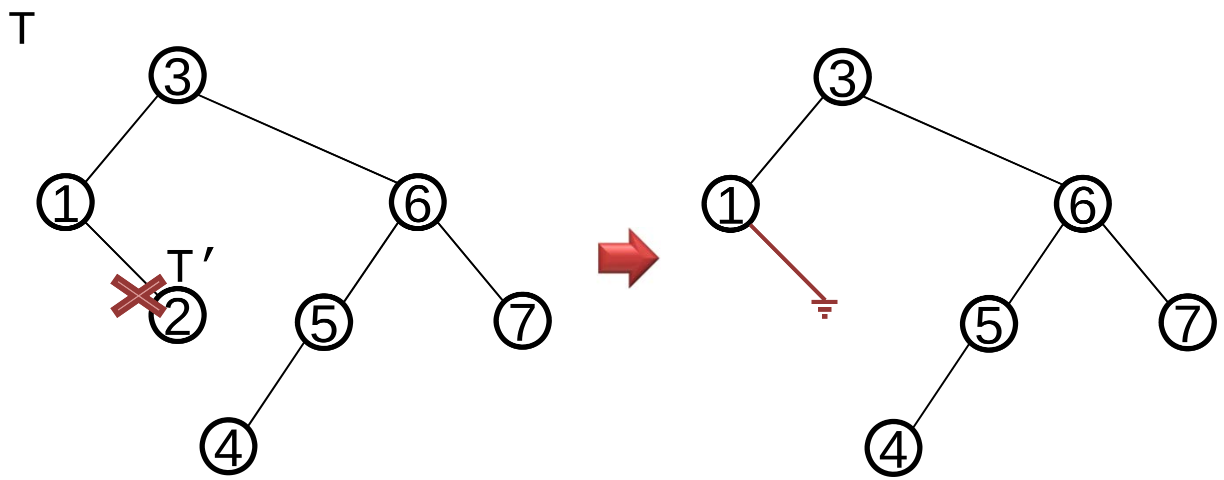
Árboles binarios de búsqueda. Extracción

- **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

Árboles binarios de búsqueda. Extracción

- **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

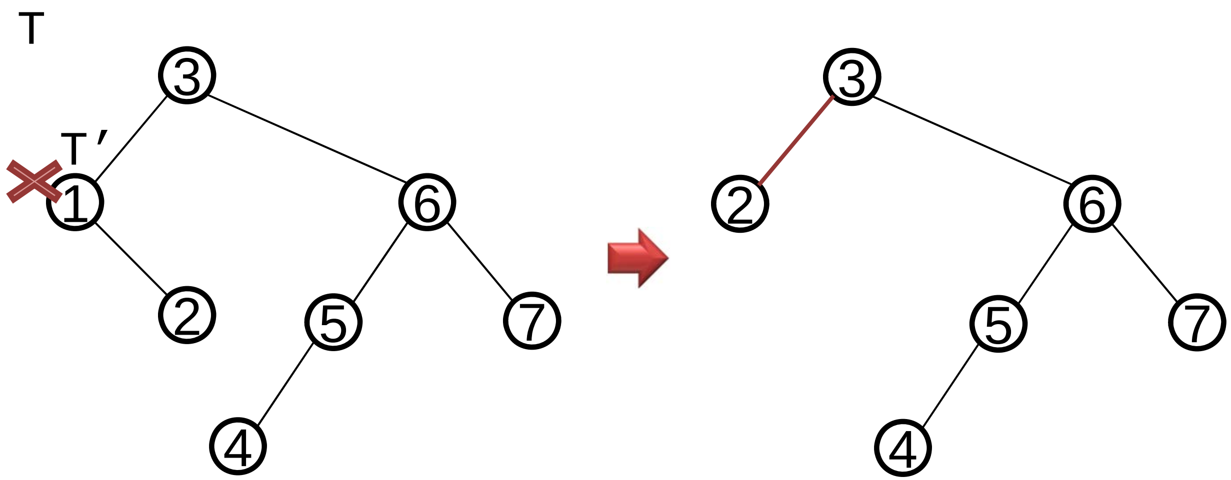
Ejemplo: **abdb_extraer**(T, 2)



Árboles binarios de búsqueda. Extracción

- **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)`

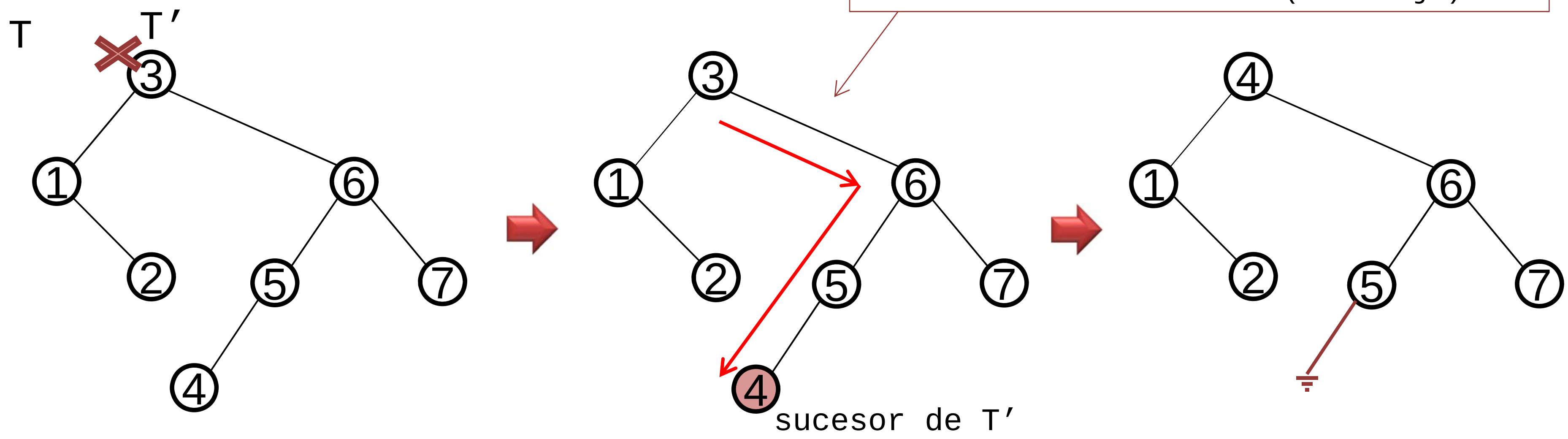


Árboles binarios de búsqueda. Extracción

- **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

```
El sucesor de T’ se obtiene:
1.  Bajando a la derecha de T’ un nivel
2.  Bajando a continuación a la izquierda
    hasta el último nivel (nodo hoja)
```

Ejemplo: **`abdb_extraer`**`(T, 3)`



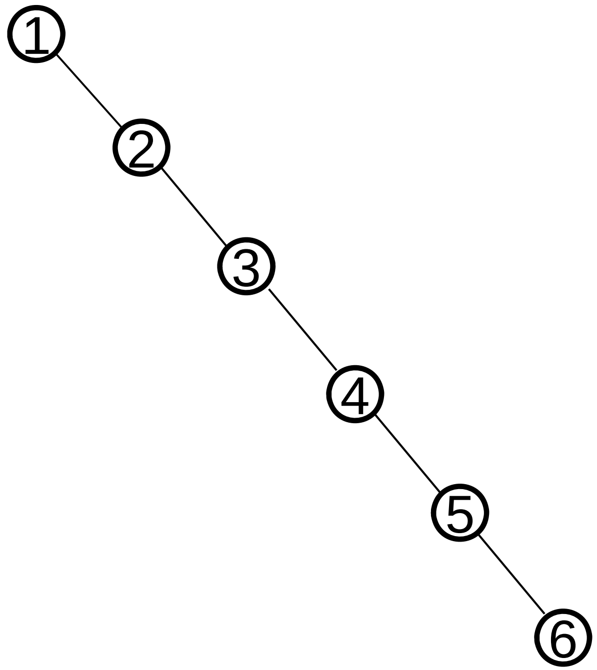
Contenidos

- 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 de búsqueda. Equilibrado

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

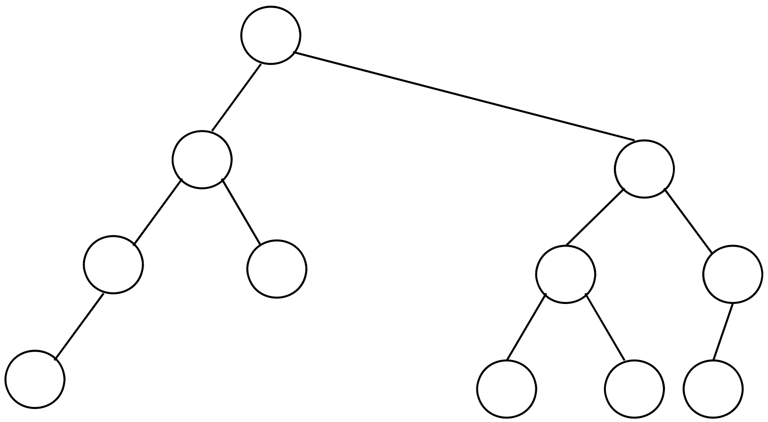
```
L = {1,2,3,4,5,6}
abdbCrear(T, L)
```



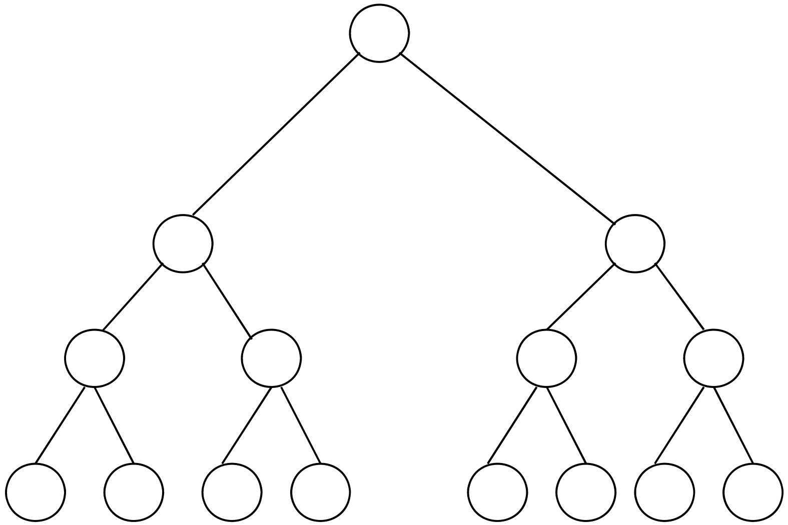
- Es como tener una lista enlazada simple de n elementos
- El acceso ya no es O(log(n)), sino O(n), por lo que:
  - Coste de la búsqueda: ya no es O(log(n)), sino O(n)
  - Coste de la ordenación: ya no es n·log(n), sino O(n^2)

Árboles binarios de búsqueda. Equilibrado

- 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

Árboles binarios de búsqueda. Equilibrado

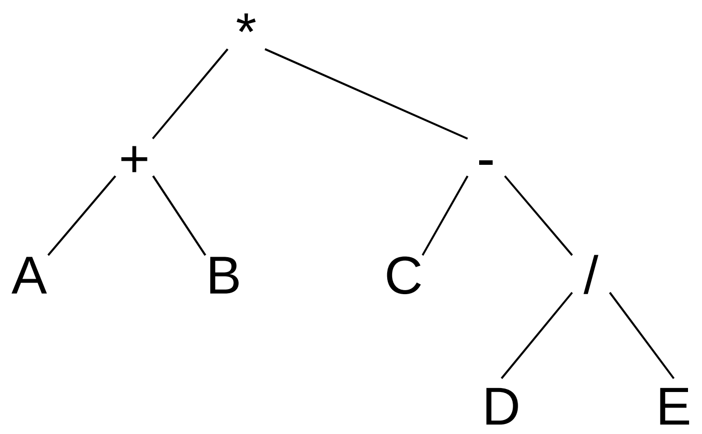
- **¿Cómo crear ABdB equilibrados?**
  - Mediante el **algoritmo AVL** (se estudiará en otra asignatura)

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

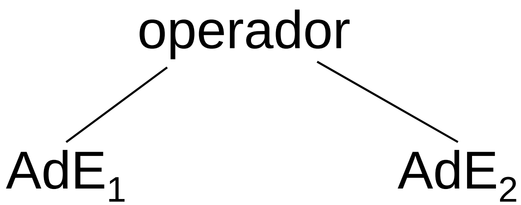
Árboles de expresión. Definición

- 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)

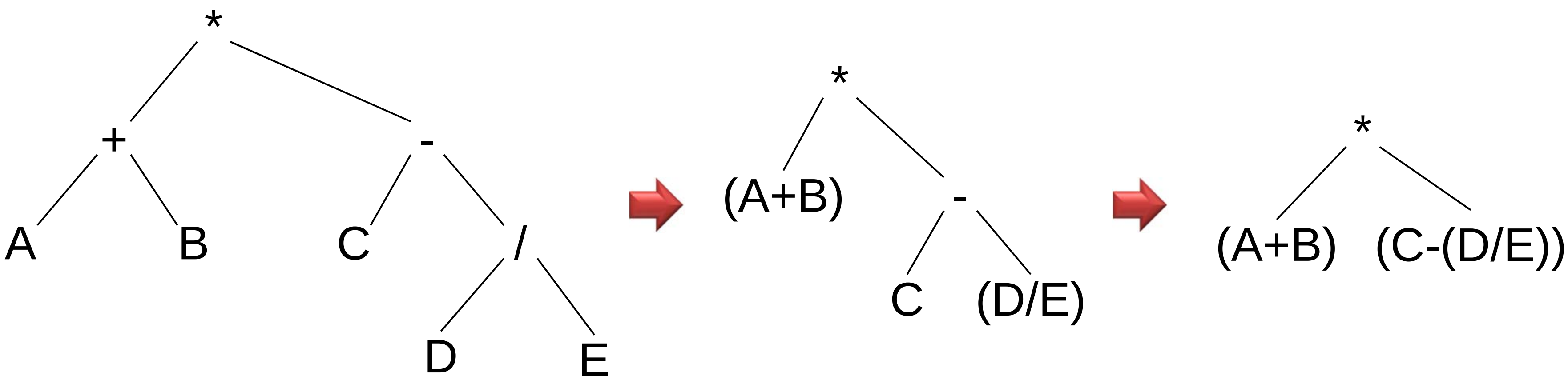


Árboles de expresión. Definición

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

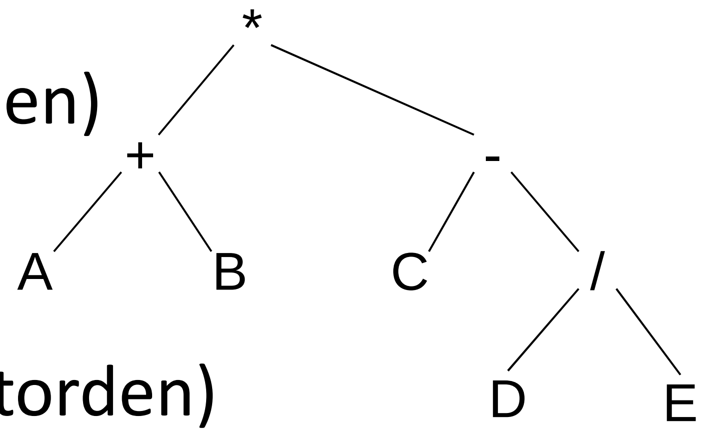


- Un AdE almacena una expresión aritmética



((A+B)*(C–(D/E)))

Árboles de expresión. Recorrido



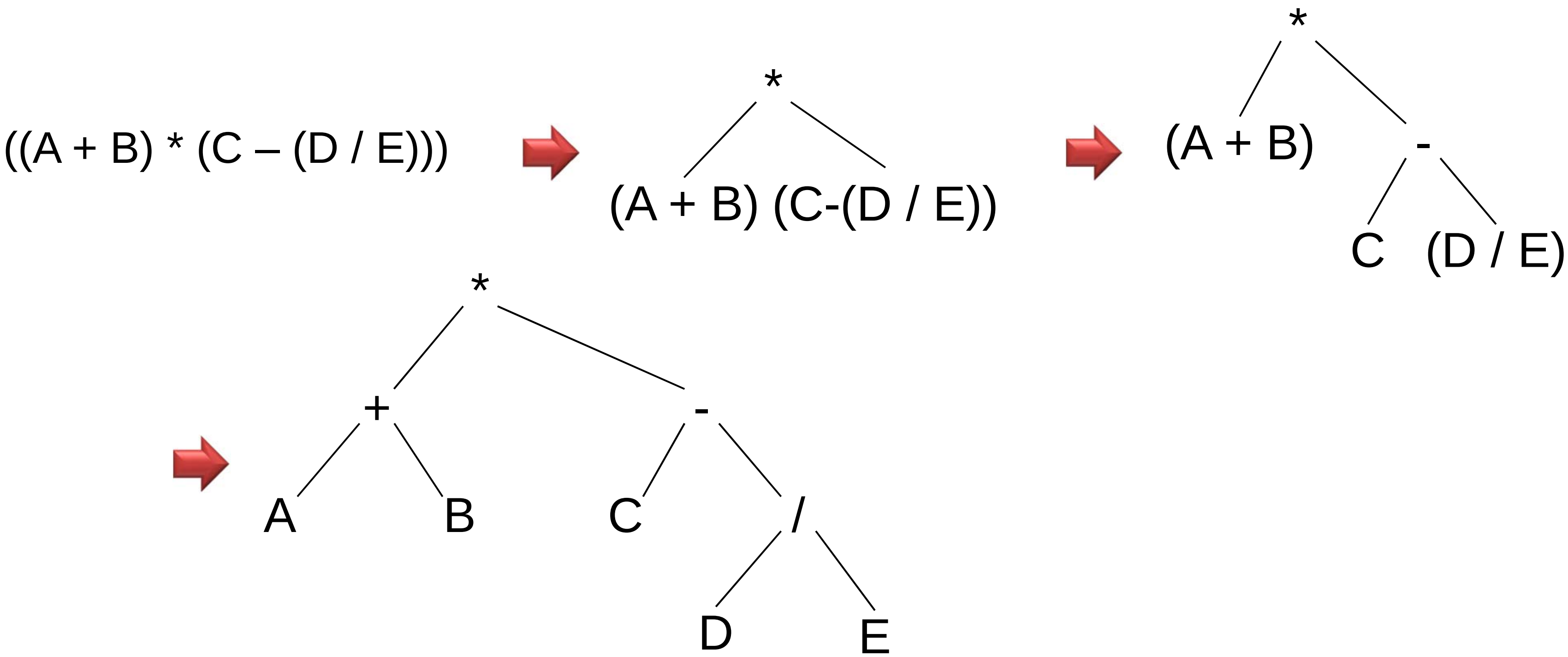
- Recorrido en orden previo (preorden)
  - Salida: * + A B – C / D E
  - Forma prefijo de la expresión
- Recorrido en orden posterior (postorden)
  - Salida: A B + C D E / - *
  - Forma postfijo de la expresión
- Recorrido en orden medio (inorden)
  - *"Imprimiendo" paréntesis al comienzo y al final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

Contenidos

- 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 de expresión. Construcción

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

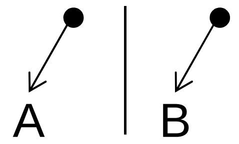
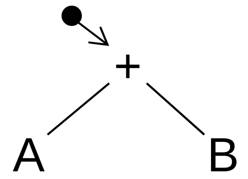
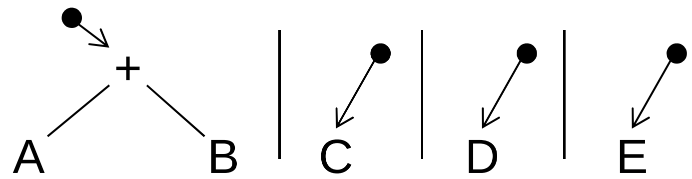
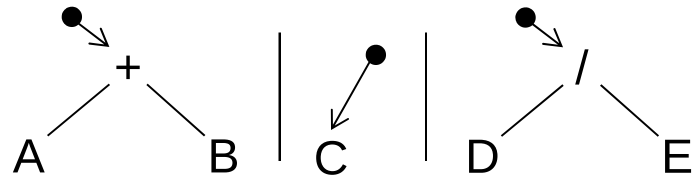


Árboles de expresión. Construcción

- Construcción de un AdE
  - Basada en la evaluación de expresiones mediante el **TAD Pila**
  - Consistente en la evaluación de una expresión <u>guardando en una pila árboles</u> generados para sub-expresiones
- El algoritmo de evaluación más sencillo es el de <u>expresiones postfijo</u>
  - 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

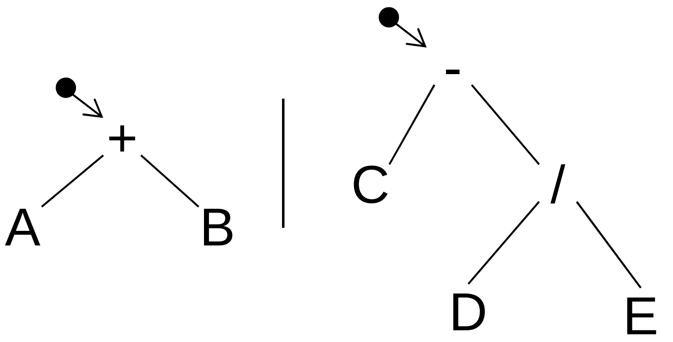
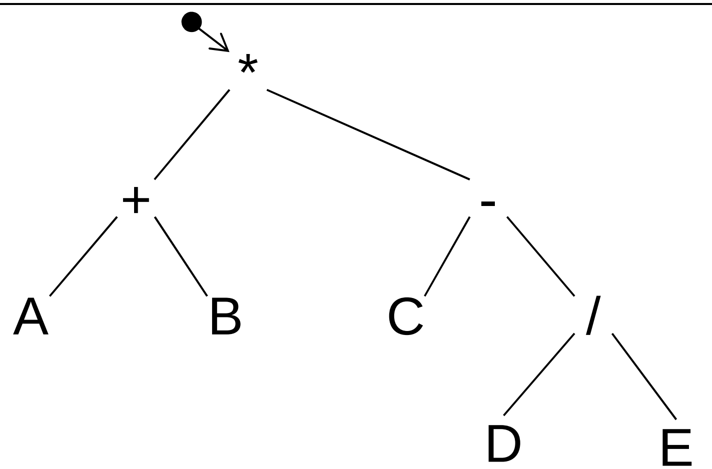
Árboles de expresión. Construcción

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

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

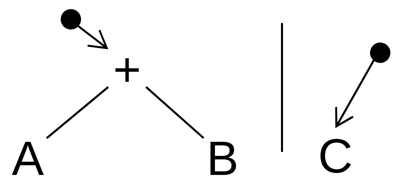
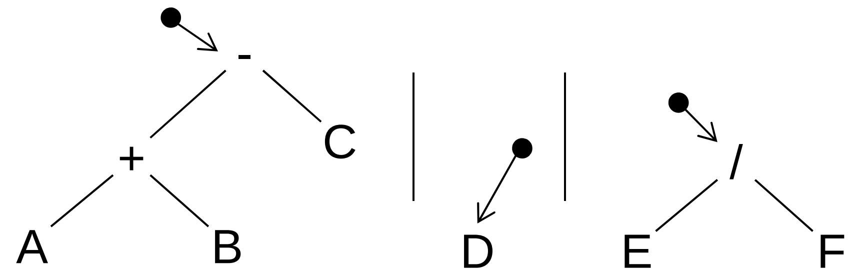
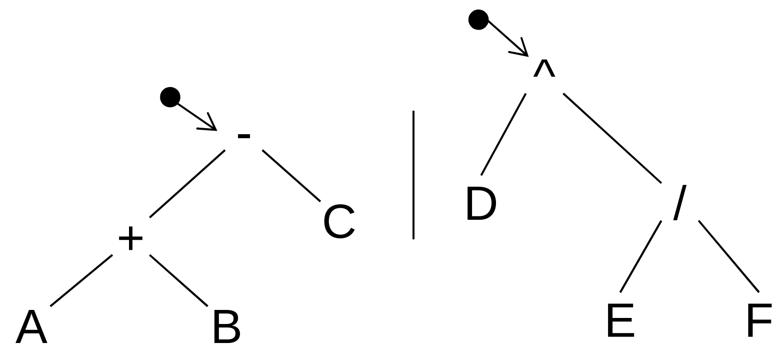
Árboles de expresión. Construcción

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

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

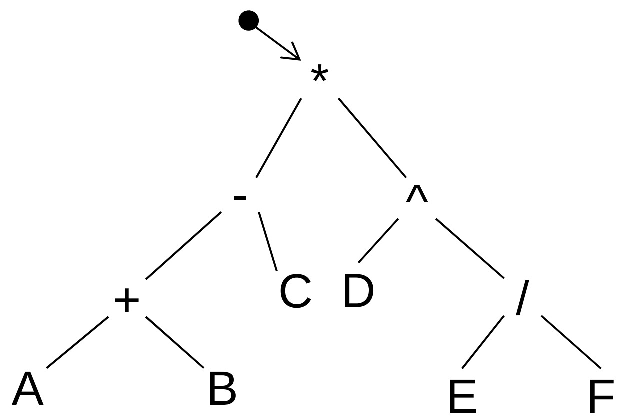
Árboles de expresión. Construcción

- 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 pila queda vacía, lo extraído es el árbol de expresión