

# Tecnología de la programación

## Sesión 24

### Objetivos de la sesión

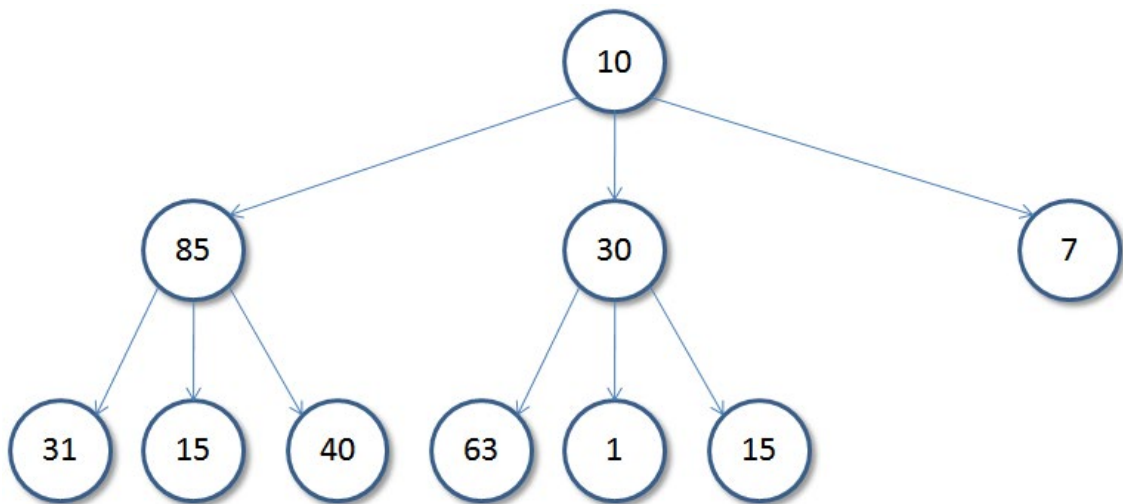
1. Implementación de árboles binarios (transparencias 19 a 24)
2. Árboles enriquecidos con recorrido (transparencias 25 a 37)
3. Árboles de búsqueda (37 a 55)

### Guion

#### Introducción

En esta sesión se ve gran parte del tema. Son muchas transparencias, pero tranquilidad: es más o menos todo el rato lo mismo.

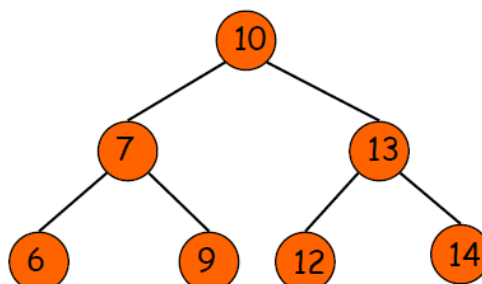
Os recuerdo que un árbol (en general) puede ser esto:



Si nos damos cuenta, en la imagen anterior cada nodo tiene como mucho tres hijos. En un árbol genérico, cada nodo podría tener cualquier número de hijos.

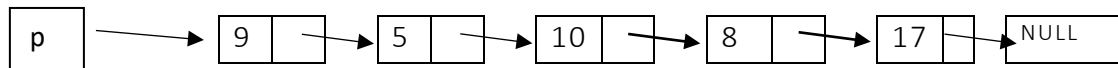
Cuando restringimos y decimos que cada nodo tiene como mucho dos, entonces estamos ante un árbol binario (de cualquier cosa, números, caracteres, etc, un árbol binario NO significa que almacene datos binarios).

#### Ejemplo de árbol binario:

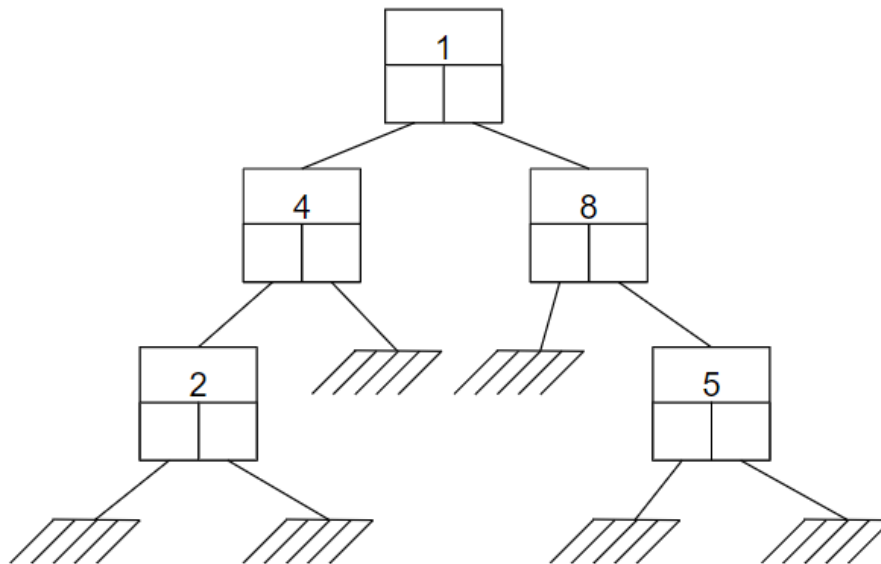


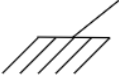
Ya hicimos la especificación del TAD Árbol Binario en la clase anterior, e incluso lo usamos como si nos lo hubiesen dado ya implementado. Ahora vamos a implementarlo y usaremos una representación dinámica.

En estructuras anteriores teníamos una lista enlazada de nodos, donde nodo era un registro con dos componentes: el dato y un puntero al siguiente.



Ahora vamos a hacerlo parecido, solo que en vez de un puntero tendremos dos: uno que apunte al subárbol izquierdo y otro al derecho.



Donde  representa NULL. La implementación es bastante fácil, todo es directo y no hay ningún bucle. **Mírala con calma línea a línea.**

## IMPLEMENTACIÓN DINÁMICA DE ÁRBOLES BINARIOS

### Implementación

#### tipo

```

Celda = registro
    tElemento dato
    puntero a Celda izdo, dcho
freg
arbolBin = puntero a Celda
  
```

**Interpretación.** Un árbol es un puntero a una celda en la que se

encuentran la raíz del árbol y dos punteros, izdo que apunta a al subárbol izquierdo, y dcho que apunta al subárbol derecho. Los punteros de los nodos hoja del árbol apuntan a NULL

```
acción iniciarArbol(sal arbolBin A)
{Pre: }
{Post: inicia A como un árbol vacío}
principio
    A = NULL
fin
```

```
función izquierdo(arbolBin A) dev arbolBin
{Pre: A es un árbol no vacío}
{Post: devuelve el árbol izquierdo que pende del nodo raíz de A}
principio
    dev(dest(A).izdo)
fin
```

```
función derecho(arbolBin A) dev arbolBin
{Pre: A es un árbol no vacío}
{Post: devuelve el árbol derecho que pende del nodo raíz de A}
principio
    dev(dest(A).dcho)
fin
```

```
función raíz(arbolBin A) dev tElemento
{Pre: A es un árbol no vacío}
{Post: devuelve el dato situado en el nodo raíz de A}
principio
    dev(dest(A).dato)
fin
```

```
función árbolVacío(arbolBin A) dev booleano
{Pre: A es un árbol iniciado o creado previamente}
{Post: devuelve Verdad si A está vacío y Falso en caso contrario}
principio
    dev(A==NULL)
fin
```

```
acción enraizar(sal arbolBin A, e/s arbolBin Aiz, e/s arbolBin Ader,
ent tElemento d)
{Pre: Aiz y Ader son árboles iniciados o contruidos previamente}
{Post: Construye el árbol A cuya raíz es el dato d, y del cual penden
los árboles Aiz y Ader. El acceso a los subárboles de A mediante Aiz y
Ade queda anulado}
principio
    A = reservar(Celda)
    dest(A).dato = d
    dest(A).izdo = Aiz
    dest(A).dcho = Ader
    Aiz = NULL // Acceso a Aiz queda anulado
    Ader = NULL // Acceso a Ader queda anulado
fin
```

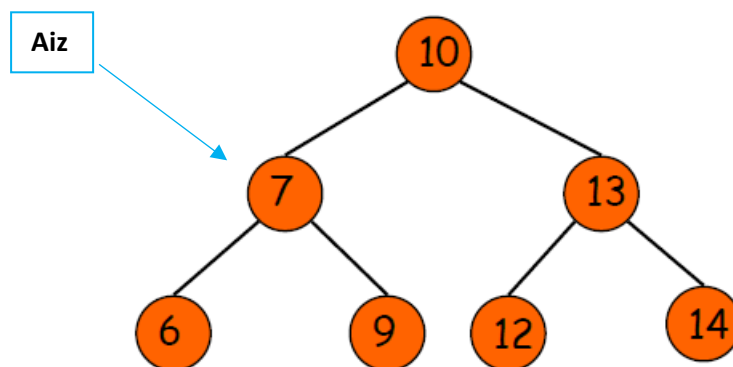
Por primera y única vez en la asignatura, vamos a devolver un TAD.

## fin\_implementación

### IMPORTANTE

Como ya hemos dicho otras veces, no solemos devolver un TAD en una función, sino hacer una acción. Sin embargo, por el tipo de estructura, en este caso nos interesa. Eso sí, al tener una implementación dinámica, vamos a tener cuidado por utilizar punteros. Por eso en la acción enraizar, en la postcondición pone “El acceso a los subárboles de A mediante Aiz y Ade queda anulado” y en el código los igualamos a NULL una vez utilizados esos punteros. De esta manera, evitaremos construir un árbol que luego pueda ser modificado desde otro sitio.

Si no anuláramos Aiz y Ader, podríamos haber construido un árbol y tener esta situación:



Como veis, Aiz es un puntero que apunta al subárbol izquierdo (a lo mismo que apuntaría `dest(A).izdo`, es decir, el puntero que está en la Celda que contiene al 10 y apunta al subárbol izquierdo). **¡¡¡Desde Aiz podemos modificar el propio árbol A!!!** Por este motivo, como ahora sí que usamos funciones que devuelven el TAD `ArbolBin`, hemos añadido esa postcondición para tener controlados posibles comportamientos inesperados.

**Lo bueno de trabajar con árboles, es que todas las operaciones del TAD tienen complejidad  $O(1)$ .**

### ENRIQUECIMIENTO DE ÁRBOLES BINARIOS

Vamos a dar ahora más operaciones para árboles binarios. Dicho con otras palabras, vamos a enriquecerlos y lo haremos con operaciones para recorrerlos.

Un recorrido consiste simplemente en visitar todos los nodos de un árbol exactamente una vez. Hay dos tipos de recorrido, en anchura (de izquierda a derecha) y en profundidad (de arriba abajo).

Dentro de los recorridos en profundidad, hay tres que son muy conocidos y utilizados:

- Preorden
- Inorden
- Postorden

La forma de recorrer se hace en tres pasos (dependiendo del orden, dar lugar a los tres tipos de recorrido):

- Acceso a raíz de un nodo
- Acceso al subárbol izquierdo de ese nodo
- Acceso al subárbol derecho de ese nodo

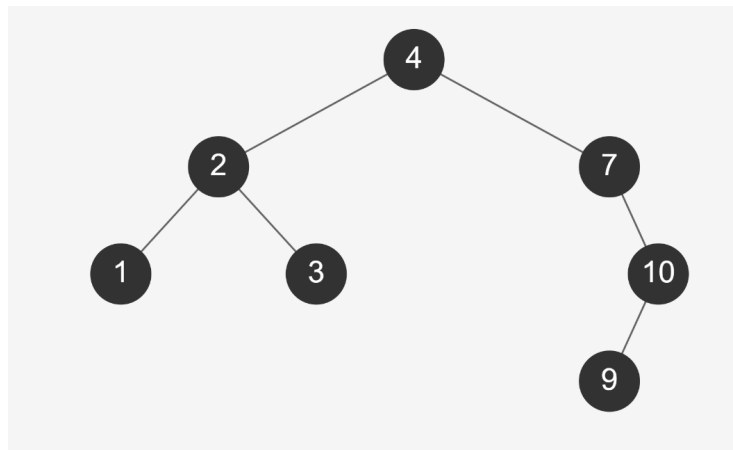
Así, los recorridos son:

- Preorden: raíz → izquierdo → derecho
- Inorden: izquierdo → raíz → derecho
- Postorden: izquierdo → derecho → raíz

Las transparencias 28, 29 y 30 tienen un ejemplo de cada tipo.

**Nota:** un árbol binario viene caracterizado por dos recorridos, no por uno. Es decir, existen árboles distintos que tienen, por ejemplo, el mismo recorrido en preorden.

**EJERCICIO (no hay que entregarlo):** Decir el recorrido en preorden, inorden y postorden del árbol:



## ESPECIFICACIÓN E IMPLEMENTACIÓN DEL TAD ÁRBOL BINARIO ENRIQUECIDO

¡¡Es muy fácil!! Échale un ojo a las transparencias 33 a 37.

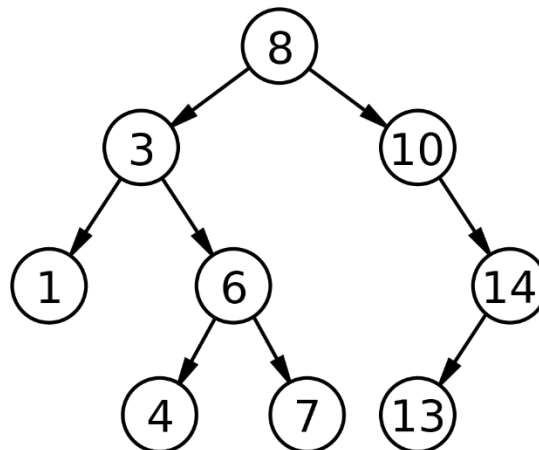
### ÁRBOLES DE BÚSQUEDA

Es simplemente un tipo particular de árbol binario que permite hacer búsquedas en complejidad  $O(\log n)$ .

Las propiedades clave son las siguientes (¡¡son claves porque te proporcionan la definición del TAD recursivamente!!)

- Todos los elementos del subárbol izquierdo son menores estrictos que el elemento raíz
- Elemento raíz es menor estricto que todos los elementos del subárbol derecho
- Subárboles izquierdo y derecho son árboles de búsqueda

Esto es un ejemplo de árbol de búsqueda:



**OJO: os soléis equivocar.** Árbol de búsqueda NO significa que “encima” de un nodo siempre haya cosas menores. Para nada. Mirad por ejemplo el 13, su padre es el 14. Y el 14 tiene de padre un número más pequeño, el 10. **Revisad muy bien la definición para entenderla.** No solo el 8 tiene que cumplir la propiedad de que sea mayor que todos los del subárbol izquierdo y menor que los del subárbol derecho. Sino que cada elemento de cada subárbol también tiene que cumplirlo. Por ejemplo, el 6 es mayor que el 4 y menor que el 7. Si sustituimos el 14 por un 12 ya NO tenemos un árbol de búsqueda, aunque todo lo que esté a la derecha del 8 sean números mayores.

En este TAD hay operaciones que no tenían sentido en general con árboles binarios, pero ahora sí que lo tienen.

- Insertar: en un árbol de búsqueda se sabe dónde, en uno binario hay muchas posibilidades
- Eliminar elementos
- Saber si un elemento está:
  - En árbol binario general se puede hacer con un recorrido (pre-in-postorden) siendo la acción tratamiento el ver si el elemento coincide con la raíz → Complejidad  $O(n)$
  - Árboles de búsqueda → incluimos operación en la especificación para implementarla de manera más eficiente →  $O(\log n)$

### ESPECIFICACIÓN TAD ÁRBOL DE BÚSQUEDA

Simplemente vamos a coger el TAD arbolBin y añadirle más operaciones.

**tad** árbolBúsqueda

**usa**

arbolBin

**género**

tElemento

**operaciones**

**función** esIgual(tElemento d1, tElemento d2) devuelve booleano  
{Pre: }

Realmente esIgual y esMenor son operaciones que pertenecerían a tElemento, pero las pongo aquí para darnos cuenta de que hay que tener cuidado si tElemento es un TAD en vez de un tipo primitivo.

{Post: devuelve verdad si d1 es igual a d2 y falso en caso contrario}

**función** esMenor(tElemento d1, tElemento d2) devuelve booleano

{Pre: }

{Post: devuelve verdad si d1 es menor a d2 y falso en caso contrario}

**acción** insertar(e/s arbolBin A, ent tElemento d)

{Pre: A es un árbol de búsqueda iniciado previamente, y d no está en A}

{Post: Inserta en A el elemento d de manera que A sigue siendo un árbol de búsqueda.}

**acción** borrar(e/s arbolBin A, ent tElemento d)

{Pre: A es un árbol de búsqueda iniciado previamente, y d está en A}

{Post: Elimina de A el elemento d de manera que A sigue siendo un árbol de búsqueda}

**función** está?(arbolBin A, tElemento d) devuelve booleano

{Pre: A es un árbol de búsqueda iniciado previamente}

{Post: devuelve Verdad si d está en A y Falso en caso contrario}

**fin\_especificación**

## IMPLEMENTACIÓN TAD ÁRBOL DE BÚSQUEDA

La representación es la del TAD Árbol Binario, ahora simplemente vamos a añadir tres operaciones nuevas:

1. Insertar
2. Borrar
3. Esta?

Al igual que cualquier ejercicio de árboles, todo se hace recursivamente. El método insertar y el esta? son fáciles, pero con el borrar hay que tener cuidado. Las transparencias tienen animaciones que os ayudarán a entender cómo funcionan.

**función** está?(arbolBin A, tElemento d) devuelve booleano

{Pre: A es un árbol de búsqueda iniciado previamente}

{Post: devuelve Verdad si d está en A y Falso en caso contrario}

**principio**

**si** árbolVacio(A) **entonces**

        dev(FALSO)

**sino**

**si** esIgual(raíz(A),d) **entonces**

            dev(VERDAD)

**sino**

**si** esMenor(d,raíz(A)) **entonces**

                dev(está?(izquierdo(A),d))

**sino**

                dev(está?(derecho(A),d))

**fsi**

**fsi**

**fin**

**acción** insertar(e/s arbolBin A, ent tElemento d)  
 {Pre: A es un árbol de búsqueda iniciado previamente, y d no está en A}  
 {Post: Inserta en A el elemento d de manera que A sigue siendo un árbol de búsqueda.}

**variables**

arbolBin iz,de

**principio**

**si** árbolVacio(A) **entonces**

iniciarArbol(iz)

iniciarArbol(de)

enraizar(A,iz,de,d)

**sino**

**si** esMenor(d,raíz(A)) **entonces**

iz = izquierdo(A)

insertar(iz,d)

**sino**

de = derecho(A)

insertar(de,d)

**fsi**

**fsi**

**fin**

Necesitamos variables auxiliares!!!

Si está vacío, entonces realizamos la inserción enraizando con subárboles vacíos

Si no está vacío, nos "movemos" al subárbol donde tenemos que insertar

**acción** borrar(e/s arbolBin A, ent tElemento d)  
 {Pre: A es un árbol de búsqueda iniciado previamente, y d está en A}  
 {Post: Borra de A el elemento d de manera que A sigue siendo un árbol de búsqueda.}

**variables**

puntero a Celda aux

**principio**

**si** not(árbolVacio(A)) **entonces**

**si** esMenor(d,raíz(A)) **entonces**

aux = dest(A).izdo

borrar(aux,d)

**sino**

**si** esMenor(raíz(A),d) **entonces**

aux = dest(A).dcho

borrar(aux,d)

**sino**

**si** árbolVacio(izquierdo(A)) **entonces**

aux = A

A = dest(A).dcho

liberar(aux)

**sino**

dest(A).dato = máximo(izquierdo(A))

aux = dest(aux).izdo

borrar(aux,dest(A).dato)

**fsi**

**fsi**

**fsi**

**fsi**

**fin**

Todos los casos están explicados en las transparencias y con animaciones

Se podría hacer con el máximo del árbol izquierdo o con el mínimo del derecho.



Como veis, hemos necesitado una función máximo para calcular el máximo de un árbol de búsqueda. Pero es muy fácil:

```
función máximo(arbolBin A) dev tElemento
{Pre: A es un árbol de búsqueda iniciado previamente y es no vacío}
{Post: Devuelve el máximo elemento del árbol de búsqueda A}
principio
    si(árbolVacío(derecho(A)) entonces
        dev raíz(A)
    sino
        dev máximo(derecho(A))
    fsi
fin
```