Representación de datos

Estructuras de datos: Arboles

Índice

Arboles

Por qué son importantes? Objetivos

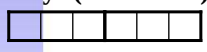
Dado un conjunto de elementos ejecutar las operaciones básicas de la manera más eficiente:

- **Búsqueda**
- **Inserción**
- **Eliminación**

Arboles

Por que son importantes: Objetivos

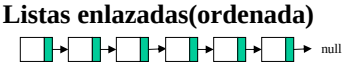
Arrays (ordenados)



Busqueda de un elemento
- **por indice: O(1)**
- **por contenido: O(log n)**

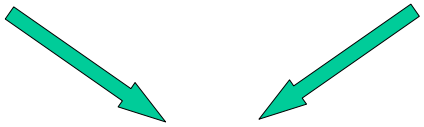
Insercion de un elemento:O(n)

Listas enlazadas(ordenada)



Búsqueda de un elemento:
- **por indice: O(n)**
- **por contenido: O(n)**

Insercion de elemento:O(1)



ARBOLES

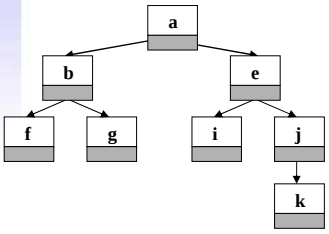
Búsqueda de un elemento O(log n)

Inserción de un elemento O(1)

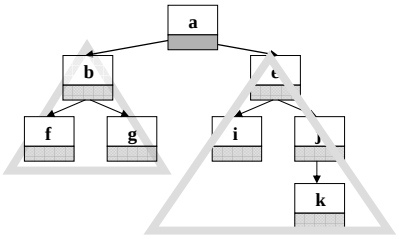
Arboles

¿Qué son ?. Características

- Un **árbol** es una estructura de datos **no lineal** que almacena los elementos **jerárquicamente**
- Se pueden definir de 2 formas:



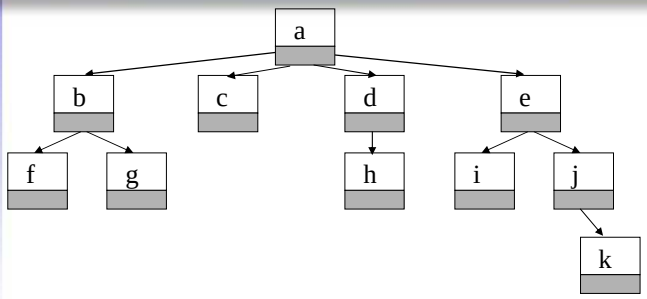
Definición NO Recursiva



Definición Recursiva

Arboles

Definición no recursiva

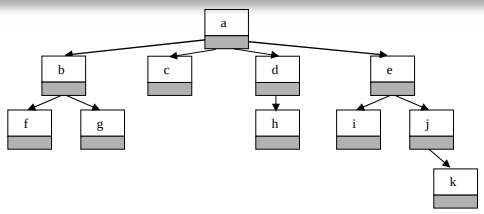


Un **árbol** consiste en:

- Un conjunto de **nodos**
- Un conjunto de **arcos orientados(ramas)** que conectan pares de nodos mediante la relación **padre-hijo**

Arboles

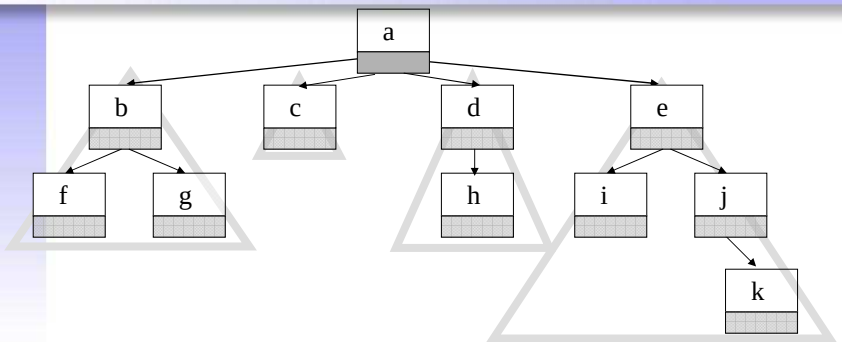
Definición no recursiva



- Definiciones
  - En un árbol se distingue un nodo “r” como **raíz** (no tiene padre)
  - Cada nodo “n” (excepto el raíz) está conectado mediante una rama a un único nodo **padre** “p”. Se dice que “p” es padre de “n” y que “n” es **hijo** de “p”
  - Los nodos que no tienen hijos se denominan **hojas**

Arboles

Definición recursiva

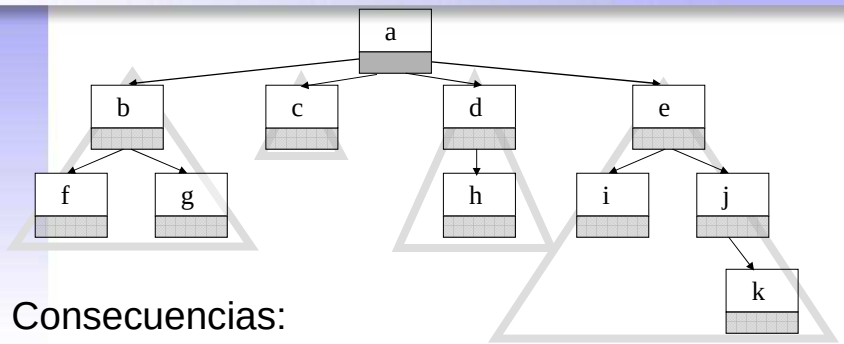


Un **árbol** A es o bien vacío o contiene:

- Una **raíz**
- Cero o más **subárboles** no vacíos A1, A2, A3, etc. cuyas raíces están conectadas mediante una **rama** a la raíz de A.

Arboles

Definición recursiva



- Consecuencias:
  - Cada nodo de un árbol define un subárbol que es aquel que le tiene por raíz.
  - Se produce una identificación entre nodo y subárbol.

Arboles

Terminología y propiedades básicas

- ***Ascendientes y Descendientes*** (Df recursiva)
  - Se llama **ascendiente** de un nodo al propio nodo o a cualquiera de los ascendientes de su nodo padre.
  - ***Para todos los nodos del árbol el nodo raíz es un ascendiente***
  - Se dice que un nodo “n1” es **descendiente** de otro “n2” si “n2” es a su vez **ascendiente** de “n1”
  - Un **subárbol** del árbol A con raíz en el nodo “n” es el árbol formado por todos los descendientes de “n” en el árbol A (incluido el propio nodo “n”)

Arboles

Terminología y propiedades básicas

- ***Parentesco***
  - Si un nodo “n1” es el nodo padre de “n2” se dice que “n2” es **hijo** de “n1”
  - Dos nodos que son hijos del mismo padre se dicen que son **hermanos**
- ***Nodos externos e internos***:
  - Un nodo es **interno** si tiene uno o más hijos
  - Un nodo es **externo** si no tiene hijos
  - Los nodos externos se llaman también **hojas**

Arboles

Terminología y propiedades básicas

- **Un camino es**

  - la secuencia de arcos que une dos nodos de un árbol.
- **Longitud de un camino entre 2 nodos**
  - Nº de arcos presentes entre ambos nodos
- **Tamaño de un nodo**
  - Nº de descendientes incluyendo el propio nodo
- **Tamaño de un árbol**
  - Tamaño del nodo raíz

Arboles

Terminología y propiedades básicas

- **Profundidad de un nodo n (3 definiciones)**
  - Longitud del camino desde la raíz hasta el nodo n
  - Nº de ascendientes del nodo n (excluido él)
  - Definición recursiva
    - Si es el nodo raíz: profundidad = 0 (caso base)
    - En otro caso: profundidad n = 1 + profundidad nodo padre
- **Altura de un nodo n (2 definiciones)**
  - Longitud del camino que va desde el nodo n hasta la hoja más profunda bajo él.
  - Definición recursiva
    - Si es una hoja (nodo externo): Altura = 0 (caso base)
    - En otro caso: altura de n = 1 + máxima altura de sus hijos

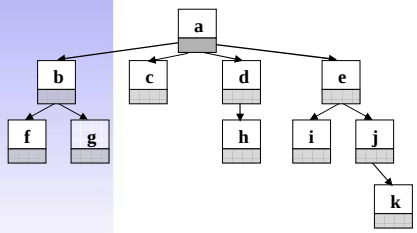
Arboles

Terminología y propiedades básicas

- **Profundidad de un árbol**
  - Es la **profundidad máxima** alcanzada en uno de sus nodos. Siempre coincidirá con la profundidad de uno de los **nodos externos (hojas**)
- **Altura de un árbol**
  - Es la **altura** de su nodo **raí**z.
- **Árboles ordenados**
  - Se dice que un árbol está **ordenado** si hay un **orden lineal** para los hijos de cada nodo. De modo podemos identificar que nodo hijo se recorre en primer lugar cuál en segundo, tercero, etc.

Arboles

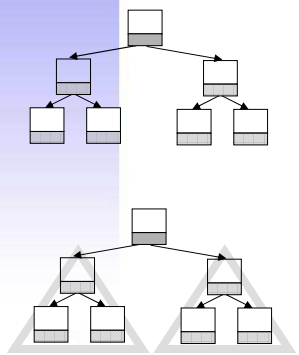
Terminología y propiedades básicas



Tamaño de árbol: 11
Altura del árbol: 3
Profundidad del árbol: 3

Nodo	Altura	Profundidad	Tamaño	Interno Externo
a	3	0	11	Interno
b	1	1	3	Interno
c	0	1	1	Externo
d	1	1	2	Interno
e	2	1	4	Interno
f	0	2	1	Externo
g	0	2	1	Externo
h	0	2	1	Externo
i	0	2	1	Externo
j	1	2	2	Interno
k	0	3	1	Externo

Arboles Binarios



- ***Árboles binarios:***

– Un árbol **binario** es un árbol en el cual cada nodo tiene **como máximo dos** hijos

- el primer nodo hijo se denomina **izquierdo**
- el segundo nodo hijo se denomina **derecho**

– Los árboles binarios se pueden definir también **recursivament**e. En este caso se dice que el árbol o es vacío o contiene:

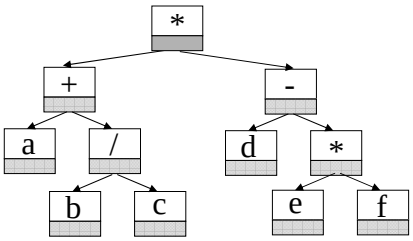
- El nodo **raíz**
- El **subárbol izquierdo** y
- El **subárbol derecho**

Arboles binarios

Ejemplos

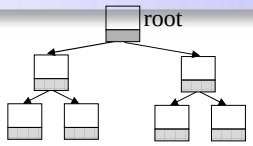
- Expresiones aritméticas con operadores binarios
- Arboles genealógicos
- Representación de un campeonato organizado con “estructura copera”

Ejemplo: (a+b/c)*(d-e*f)



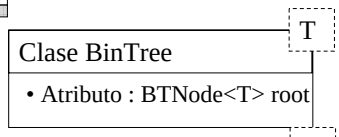
Representación de Arboles

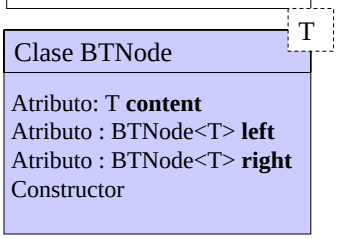
BinTree:estructura dinámica




```
public class BinTree<T>{
 BTNode<T> root;
}
```

```
public class BTNode<T>{
  T content;
  BTNode<T> left;
  BTNode<T> right;
  public BTNode(T elem){
    this.content=elem;
  }
}
```



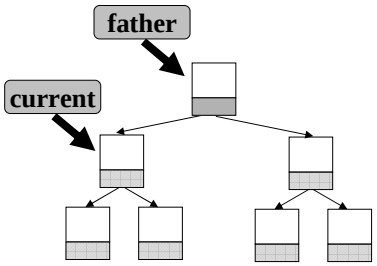


Representación de Arboles

BinTreeItr: la clase iteradora sobre una estructura dinámica

class BinTree<T>

•Atributo: BinTree<T> bTree
•Atributo: BTNode<T> current
•Atributo: Stack<T> father
•Métodos:
boolean isEmpty();
void emptyTree();
void goLeft();
void goRight();
void goBrother();
void goRoot();
void inserRoot(T o);
void insertLeftChild(T o);
void insertRightChild(T o);
T getCurrent();



class BTNode<T>
•Atributo: T **content** •Atributo: BTNode<T> **left** •Atributo: BTNode<T> **right** •Constructor

Arboles

Métodos principales y métodos auxiliares

Métodos desplazamiento	***Significado***
void **goRoot()**	Ubica el elemento *actual* en la **raíz** del árbol
void **goLeft()**	Ubica el elemento *actual* en el hijo **izquierdo** de actual
void **goRight()**	Ubica el elemento *actual* en el hijo **derecho** de actual
void **goBrother()**	Ubica el elemento *actual* en el **hermano** de actual

Métodos consulta	***Significado***
boolean **isInternal()**	Devuelve *true* si el nodo *actual* es **interno**
boolean **isExternal()**	Devuelve *true* si el nodo *actual* es **externo** (hoja)
boolean **isRoot()**	Devuelve *true* si el nodo *actual* trata del nodo **raíz**
int **size()**	Devuelve el **tamaño** del árbol
boolean **isEmpty()**	Devuelve *true* si el árbol está **vacío** y false en caso contrario
T **getActual()**	Devuelve el contenido del nodo *actual*
void setActual(T o)	**Modifica** el contenido del nodo *actual* por el de o

Arboles

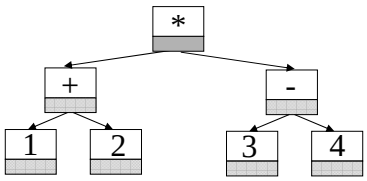
clase BinTreeItr: la clase iteradora

Métodos inserción	*Significado*
void **insertRoot** (T o)	Inserta el elemento o como elemento raíz del árbol
void **insertLeftChild** **(T o)**	Inserta el elemento o como nodo izquierdo de *actual*
void **insertRightChild** **(T o)**	Inserta el elemento o como nodo derecho de *actual*

Arboles

Ejemplo de construcción de un árbol a través de la clase iteradora

```
BinTree<String> bTree = new BinTree<String>();
bTreeItr.insertRoot(new String(“*”));
bTreeItr.insertLeftChild(new String(“+”));
bTreeItr.insertRightChild(new String(“-”));
bTreeItr.goLeft();
bTreeItr.insertLeftChild(new String(“1”));
bTreeItr.insertRightChild(new String(“2”));
bTreeItr.goRoot();
bTreeItr.goRight();
bTreeItr.insertLeftChild(new String(“3”));
bTreeItr.insertRightChild(new String(“4”));
```



Representación de árboles

Resolución de metódos recursivamente

- Se resuelve en 4 fases:
  1. Especificación / parametrización del problema
  2. Definición y solución de casos triviales y generales
  3. Diseño del método
  4. Codificación del método

Representación de árboles

Los 3 principios de la recursión (Weiss)

1. *Casos base*: se debe tener siempre al menos un caso base que pueda resolverse sin recursión

2. *Progreso*: cualquier llamada resursiva debe progresar hacia un caso base

3. "*Puede creerlo*": asuma siempre que toda llamada recursiva interna funciona correctamente

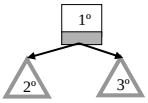
Recorrido en árboles

Una tarea que se realiza con frecuencia sobre un TAD árbol es ejecutar una determinada operación **para cada uno** de los elementos del árbol.

Realizar una operación para todos los elementos del árbol implica la "**visita**" a cada nodo. Este proceso es conocido como **recorrido del árbol**. Existen básicamente 3 tipos:

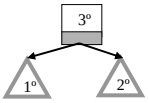
1. **PreOrden**

   Visitar la raíz
   Recorrer en preorden el subárbol izquierdo
   Recorrer en preorden el subárbol derecho



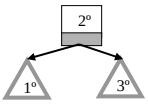
2. **PostOrden**

   Recorrer en postorden el subárbol izquierdo
   Recorrer en postorden el subárbol derecho
   Visitar la raíz



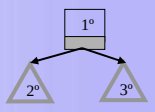
3. **InOrden (Orden central)**

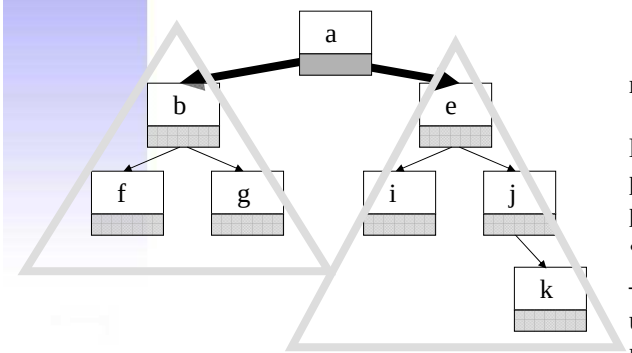
   Recorrer en inorden el subárbol izquierdo
   Visitar la raíz
   Recorrer en inorden el subárbol derecho



Arboles

Recorrido en preorden





Preorden (Def)

- Primero se procesa el **nodo**
- Luego se procesan recursivamente **sus hijos**
- Si el árbol está ordenado los subárboles hijos se procesan en el orden previamente establecido.

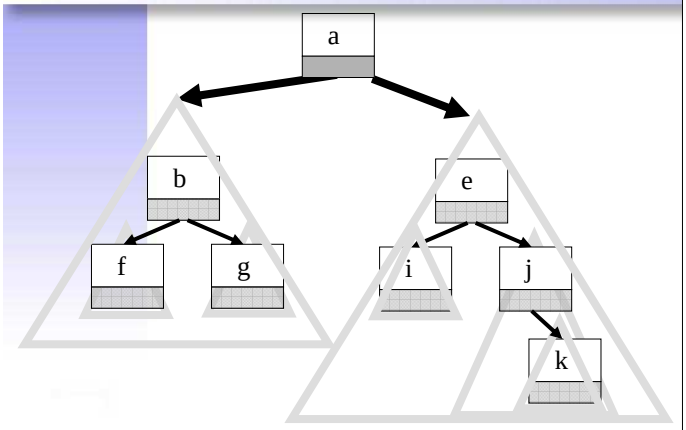
• ***Utilidad***

– Cuando queremos producir una ordenación lineal de los nodos en las que el nodo padre va siempre antes que sus hijos.

– Ej: capítulos de un libro

Arboles

Recorrido en preorden



Preorden
1. Primer nodo (a)
2. Árbol izquierda
1. Primer Nodo **(b)**
2. Árbol izda
   1. Primer nodo **(f)**
   2. Árbol izda (null)
   3. Árbol dcha (null)
3. Árbol dcha
   1. Primer nodo **(g)**
   2. Árbol izda (null)
   3. Árbol Dcha (null)

3. Árbol derecha
1. Primer Nodo **(e)**
2. Árbol izda
   1. Primer nodo **(i)**
   2. Árbol izda (null)
   3. Árbol dcha (null)
3. Árbol dcha
   1. Primer nodo **(j)**
   2. Árbol izda (null)
   3. Árbol dcha
      1. Primer nodo **(k)**
      2. Árbol izda (null)
      3. Árbol dcha (null)

Orden en el que se procesan los nodos:
a,b,f,g,e,i,j,k

Arboles

Ejemplo I: Recorrido en preorden

Fase I : Especificación/Parametrización

Entrada:
Arbol binario

Salida:
Impresión por pantalla de los elementos del árbol visitados en preOrden

Fase II : Solución casos triviales y generales

Caso(s) trivial(es)
- Si **árbol es vacío** entonces No escribir nada

Caso(s) general(es)
- Si **árbol no es vacío** entonces
  - Escribir el valor del nodo raíz
  - Escribir el subárbol izquierdo en preOrden
  - Escribir el subárbol derecho en preOrden

Arboles

Ejemplo I: Recorrido en preorden

Fase III : Diseño del subprograma

```
subprograma PreOrden(Arbol)

 Si Arbol NO vacío
   imprimir arbol.raiz
   PreOrden(arbol.subarbolIzquierdo);
   PreOrden(arbol.subarbolDerecho);
 si no
    No hacer nada
```


Arboles

Ejemplo I: Recorrido en preorden

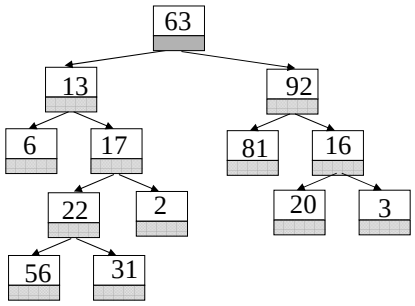
Fase IV : Programación

```
public void printPreOrder() {
  this.printPreOrder1(bTree.root);
}
```

```
private void printPreOrder(BTNode<T> arbol) {
   if (arbol != null) {
       System.out.println(arbol.content);
       printPreOrder(arbol.left);
       printPreOrder(arbol.right);
   }
}
```

Representación de árboles
Ejercicio I

Desarrollar un método que dado un árbol binario (p.ej. que representa un conjunto de números enteros) y un objeto x, devuelva cierto si algún nodo de árbol tiene el valor x y falso en caso contrario



Ejemplo: binTree.esta(new Integer(22)) devuelve *true*

Representación de árboles
Ejercicio I

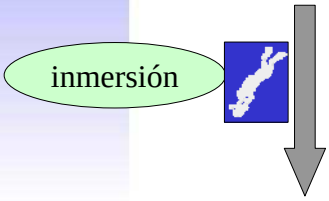
Fase I : Especificación/Parametrización

Entrada:
Un árbol y un objeto x

Salida:
Un valor lógico que será cierto si algún nodo del árbol tiene el valor x y falso en caso contrario.

Representación de árboles
Ejercicio I: El problema de la inmersión

public boolean esta(T x)



En cada paso necesitamos saber además de x, el **Arbol** sobre el cual estamos buscando el elemento

private boolean esta(**BTNode<T> arbol**, T x)

Se define un nuevo método con más parámetros

Representación de árboles
Ejercicio I

Fase II : Solución casos triviales y generales

Caso(s) trivial(es)
- Si **árbol es vacío** entonces devolver falso
- Si **árbol no es vacío y su raíz es igual al elemento buscado** entonces devolver cierto

Caso(s) general(es)
- Si **árbol no es vacío y su raíz no es igual al elemento buscado** devolver cierto si el elemento está en el subárbol izquierdo o en el subárbol derecho, en caso contrario devolver falso

Representación de árboles
Ejercicio I

Fase III : Diseño del subprograma

Si árbol NO es vacío entonces
  Si la raíz es igual al elemento buscado entonces
    devolver cierto
  si no
    devolver (esta (subárbol izquierdo, x) O
      esta (subárbol derecho, x))
si no
  devolver falso

Representación de árboles
Ejercicio I

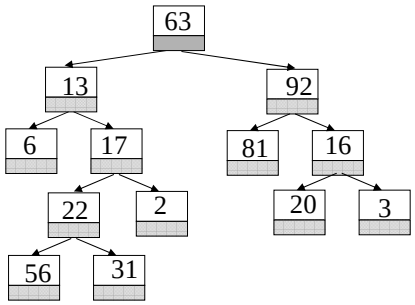
Fase IV : Programación

```
private boolean esta(BTNode<T> arbol, T x) {
  if (arbol != null) {
    if (arbol.content.equals(x))
      return true;
    else
      return (esta(arbol.left, x) ||
              esta(arbol.right, x) );
  }
  return false;
}
```

```
public boolean esta(T x) {
  return esta(bTree.root, x);
}
```

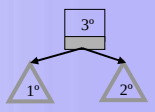
Representación de árboles
Ejercicio clase

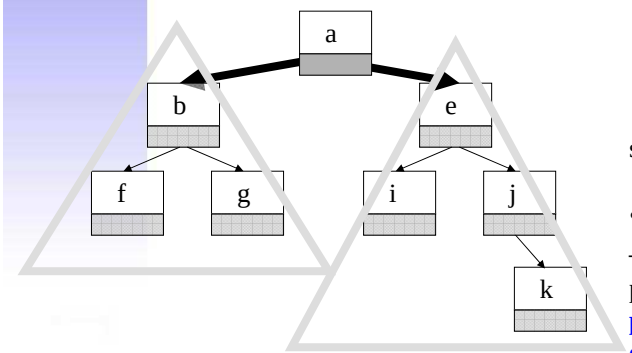
Implementa el método *int profundidad()*, que calcula y devuelve la profundidad de un árbol binario



Ejemplo: binTree.profundidad() devuelve 4

Arboles
Recorrido en postorden





Postorden (Def)

- Primero se procesan recursivamente los **subárboles hijos**
- Si el árbol está ordenado se sigue el orden establecido
- Por último el **nodo**

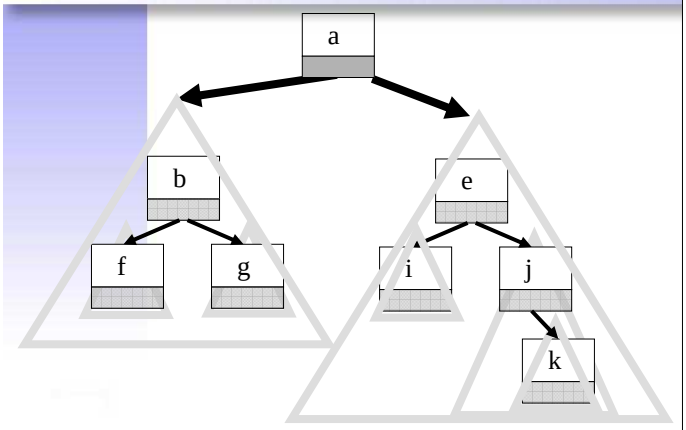
• ***Utilidad***

– Para poder resolver el problema del nodo padre, **primero** se necesita resolver **el de los hijos**.

– Ejemplo: Tamaño de un directorio en un sistema de ficheros.

Arboles

Recorrido en postorden



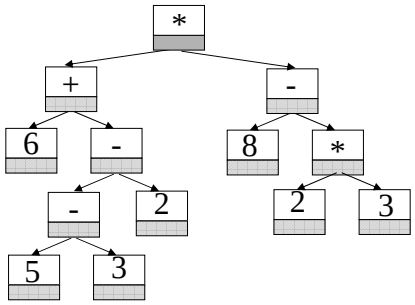
Orden en el que se procesan los nodos:
f,g,b,i,k,j,e,a

Postorden
1. Árbol izquierda
1. Arbol izquierda
1.Árbol izda (null)
2. Árbol dcha (null)
3. Nodo **(f)**
2. Árbol dcha
1.Árbol izda (null)
2. Árbol dcha (null)
3. Nodo **(g)**
3. Nodo **(b)**
2. Árbol derecha
1. Arbol izquierda
1.Árbol izda (null)
2. Árbol dcha (null)
3. Nodo **(i)**
2. Árbol dcha
1.Árbol izda (null)
2. Árbol dcha (null)
1. Arbol izda. (null)
2. Árbol drcha (null)
3. Nodo (null) **(k)**
3. Nodo **(j)**
3. Nodo (e)
3. Nodo (a)

Representación de árboles

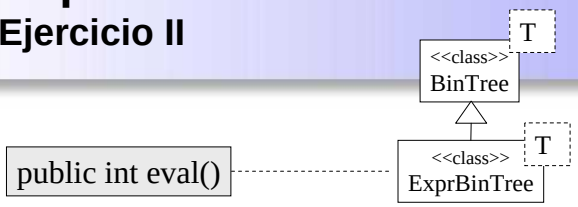
Ejercicio II

Desarrollar un método que dado una estructura de tipo árbol, que representa una expresión aritmética cuyos operandos básicos son dígitos, obtenga el valor de dicha expresión.



Ejemplo: binTree.eval() devuelve 12

Representación de árboles
Ejercicio II



Fase I : Especificación/Parametrización

Entrada:
Arbol binario que representa un representación aritmética cuyos operandos básicos son dígitos

Salida:
Resultado de evaluar la expresión.

private int eval(**BTNode<T> arbol**)

Representación de árboles
Ejercicio II

Fase II : Solución casos triviales y generales

Caso(s) trivial(es)
- Si **árbol es vacío** entonces devolver 0
- Si **árbol tiene como raíz un dígito** entonces devolver el valor del dígito

Caso(s) general(es)
- Si **árbol tiene como raíz un operador** entonces
  - op1 = evaluar subárbol izquierdo
  - op2 = evaluar subárbol derecho
  - ejecutar el operador sobre los resultados obtenidos, es decir, op1 y op2

Representación de árboles
Ejercicio II

Fase III : Diseño del subprograma

Si árbol NO es vacío entonces
  Si árbol tiene como raíz un dígito entonces
    devolver el valor del dígito
  si no //el árbol tiene como raíz un operador
    op1 = evaluar (subárbol izquierdo)
    op2 = evaluar (subárbol derecho)
    ejecutar operador de la raíz sobre los resultados obtenidos
    (op1 y op2) y devolver el resultado
si no
  devolver 0

Representación de árboles
Ejercicio II

Fase IV : Programación

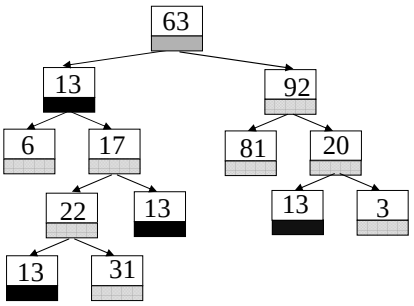
```
class ExprBinTree<T> extends BinTree<T> {
```

```
private int eval(BTNode<T> arbol) {
  Character c;
  if (arbol != null) {
    c = (Character) arbol.content;
    if (Character.isDigit (c.charValue()) )
      return Integer.parseInt(c.toString());
    else {
      int left = eval(arbol.left);
      int right = eval(arbol.right);
      return evaluate(c.charValue(), left, right);
    }
   }
  return 0;
}
```

```
public int eval() {
   return this.eval(bTree.root);
}
```

Representación de árboles
Ejercicio clase

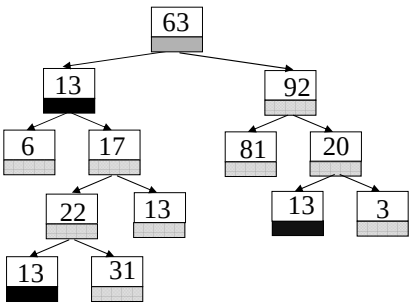
Implementar el método *int count(T x)*, que dado un árbol binario devuelve el número de apariciones del objeto x dicho árbol.



Ejemplo: binTree.count(new Integer(13)) devuelve 4

Representación de árboles
Ejercicio clase

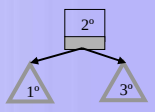
Implementar el método *int countLeft(T x)*, que dado un árbol binario devuelve el número de apariciones del objeto x en nodos izquierdos de dicho árbol.

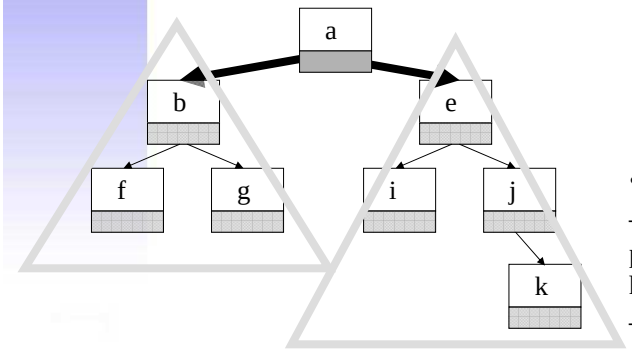


Ejemplo: binTree.countLeft(new Integer(13)) devuelve 3

Arboles

Recorrido en orden simétrico





Orden simétrico (Def)

- Primero se procesa el árbol de la **izquierda**
- Luego el **nodo**
- Por último el árbol de la **derecha**

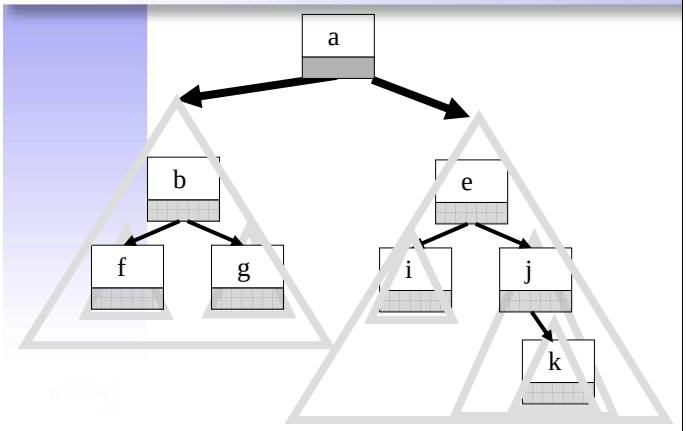
• ***Utilidad***

– Cuando el nodo padre debe procesarse entre los dos nodos hijos.

– Ejemplo: Generar una expresión algebraica a partir de un árbol de expresión

Arboles

Recorrido en orden simétrico

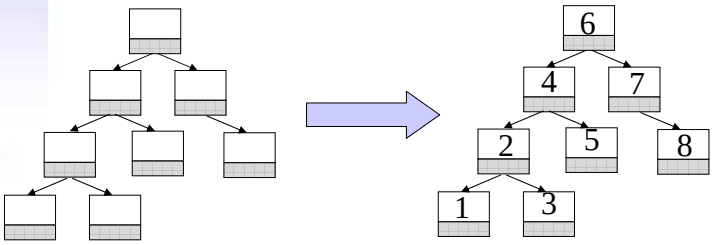


Orden en el que se procesan los nodos:
f,b,g,a,i,e,j,k

Postorden
1. Árbol izquierda
1. Arbol izquierda
1.Árbol izda (null)
2. Nodo **(f)**
3. Árbol dcha (null)
2. Nodo **(b)**
3. Árbol dcha
1.Árbol izda (null)
2. Nodo **(g)**
3. Árbol dcha (null)
2. Nodo (a)
3. Árbol derecha
1. Arbol izquierda
1.Árbol izda (null)
2. Nodo **(i)**
3. Árbol dcha (null)
2. Nodo (e)
3. Árbol dcha
1.Árbol izda (null)
2. Nodo **(j)**
3. Árbol dcha (null)
1. Arbol izda. (null)
2. Nodo (null) **(k)**
3. Árbol drcha (null)

Representación de árboles
Ejercicio clase

Implementar el método *enumerar()*, que dado un árbol binario cuyos nodos están vacios, devuelve el mismo arbol cuyos nodos están enumerados siguiendo un recorrido InOrder y empezando desde el 1.



Representación de árboles
Ejercicio IV

Realizar un programa que dado un fichero que contiene una expresión aritmética sencilla en notación prefija (operador operando operando) construye un árbol representado "dinámicamente"

Ejemplo: + + 3 5 – 6 8

Representación de árboles
Ejercicio IV

Fase I : Especificación/Parametrización

Entrada:
Fichero de entrada en el que se encuentra una expresión en notación prefija

Salida:
Un iterador sobre árbol binario Btree que representa la expresión contenida en el fichero.

Representación de árboles
Ejercicio IV

Fase II : Solución casos triviales y generales

Caso(s) trivial(es)
- Si **final de fichero** entonces árbol vacío
- Si **no final de fichero y carácter obtenido no es operador** entonces árbol tendrá como nodo raíz el carácter leído y como subárboles izquierdo y derecho el árbol vacío

Caso(s) general(es)
- Si **no final de fichero y carácter obtenido es un operador** entonces árbol tendrá como raíz el carácter leído. Sus subárboles izquierdo y derecho se crean con el resto del contenido del fichero.

Representación de árboles
Ejercicio IV

Fase III : Diseño del subprograma

Si NO final de fichero entonces
  Crear nodo para la raíz del árbol
  Leer un carácter del fichero sobre el nodo creado
  Si carácter obtenido es un operador
    creaArbol(subarbolIzquierdo)
    creaArbol(subarbolDerecho)
    construye árbol con el nodo raíz y los dos subárboles
  si no
    construye árbol con el nodo raíz y dos subárboles vacíos
si no
  árbol vacío

Representación de árboles
Ejercicio IV

Fase IV : Programación

```
private BTNode<T> createNode() {
  char c;
  BTNode<T> btNode = new BTNode<T>();
  if (in.available() != 0) {
    c=(char) in.readByte();
    btNode.content = new Character(c);
    if (!Character.isDigit(c)) {
      BTNode<T> btNodeLeft = createNode();
      BTNode<T> btNodeRight = createNode();
      btNode.left = btNodeLeft;
      btNode.right = btNodeRight;
    }
    else{
      btNode.left = null;
      btNode.right = null;
    }
    return btNode;
  }
  else return null;
}
```

```
BufferedReader in;

public BinTree loadTree() {
   BinTree<T> bTree = new BinTree<T>();
   bTree.root = createNode();
}
```

Representación de árboles

Ejercicio I: Versión mejorada del método esta()

Fase IV : Programación

```
public boolean esta(BTNode<T> arbol, T x) {
  if (arbol != null) {
     if (arbol.content.equals(x) )
         return true;
     else
        if (esta(arbol.left, x) )
           return true;
        else esta(arbol.right, x);
   }
  return false;
}
```


Representación de árboles

Organización de los datos

En la primera versión del método **está,** se recorre prácticamente todo el árbol de entrada independientemente de si el elemento buscado se encuentra en el árbol.

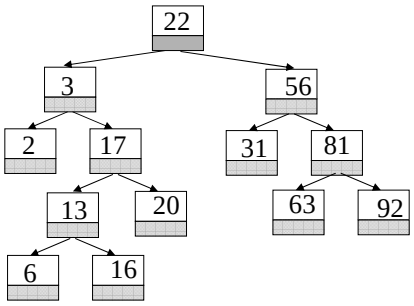
En la segunda versión de la función **esta** se recorren todos los nodos que preceden (en un recorrido en preorden) al que contiene el valor buscado. Para determinar que un elemento no está en el árbol se recorren todos los nodos.

PROCESO COSTOSO

Organización del árbol que prime una búsqueda más eficiente

Arboles binarios de búsqueda

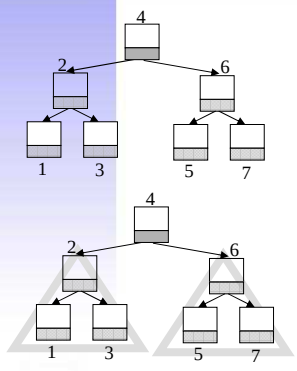
Que propiedad caracteriza al siguiente árbol ?



Para todos los nodos de este árbol se cumple que:

- todos los valores de los nodos de su subárbol izquierdo son menores que el valor del nodo raíz; y
- todos los valores de los nodos de su subárbol derecho son mayores que el valor del nodo raíz.

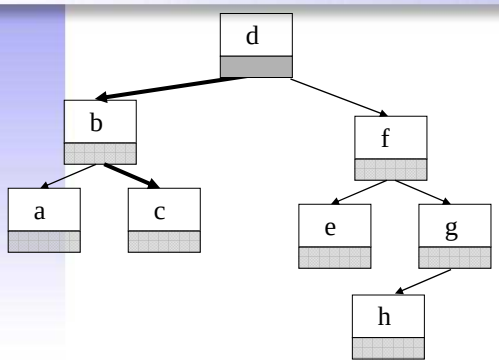
Arboles binarios de búsqueda



- ***Árboles binarios de búsqueda:***
  - Es un árbol binario que satisface la propiedad de la **búsqueda ordenada**
  - Esto significa que cualquier nodo X del árbol cumple que:
    - Todos los nodos del **subárbol izquierdo** tienen una **clave menor** que la clave de X
    - Todos los nodos del **subárbol derecho** tienen una **clave mayor** que la clave de X
    - No se permiten elementos duplicados

Arboles binarios de búsqueda

Operaciones básicas: Búsqueda



Ejemplo buscamos la “c”:
- c < d: Subárbol izquierdo
- c > b: Subárbol derecho
- c = c: Elemento encontrado

- ***Buscar*** : Nos desplazamos por las ramas izquierda o derecha dependiendo del resultado de la comparación.
  - Si la clave buscada coincide con la del nodo raíz ya hemos encontrado el elemento
  - Si llegamos a una hoja y aún no lo hemos encontrado el elemento no está en el árbol
  - Si es menor que la del nodo raíz buscamos en el subárbol izquierdo
  - Si es mayor que la del nodo raíz buscamos en el subárbol derecho

Arboles binarios de búsqueda

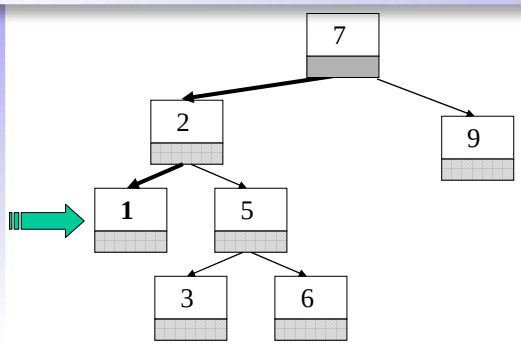
Operaciones básicas: Búsqueda. Implementación

```
public boolean find(Comparable<T> x) {
  return find(bTree.root, x); //inmersión}
```

```
private boolean find(BTNode<T> arbol, Comparable<T> x) {
  if (arbol == null)
    return false;
  else {
      if (x.compareTo(arbol.content) == 0)
          return true;
      else if (x.compareTo(arbol.content) > 0)
          return find(arbol.right, x);
      else
          return find(arbol.left, x);
  }
}
```

Arboles binarios de búsqueda

Operaciones básicas: Búsqueda del menor



El elemento menor es la hoja del árbol ubicada **<u>más a la izquierda</u>** (Nodo.left == null)

Arboles binarios de búsqueda

Operaciones básicas: Búsqueda del menor

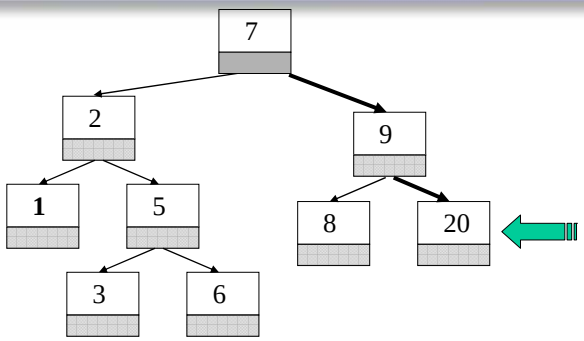
Implementación

```
public T findMin() {
 return findMin(bTree.root); //inmersión}
```

```
private T findMin(BTNode<T> arbol) {
  if (arbol.left != null)
     return findMin(arbol.left);
  else
     return arbol.content;
}
```

Arboles binarios de búsqueda

Operaciones básicas: Búsqueda del mayor



El elemento mayor es la hoja del árbol ubicada **<u>más a la derecha</u>** (Nodo.right == null)

Arboles binarios de búsqueda

Operaciones básicas: Búsqueda del mayor

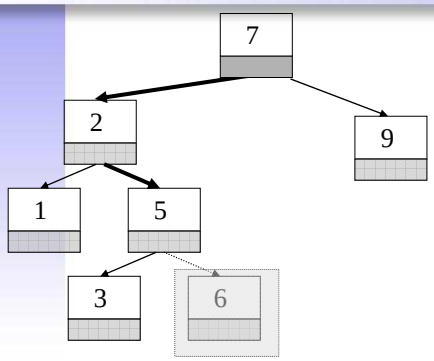
Implementación

```
public T findMax() {
  return findMax(bTree.root);}
```

```
private T findMax(BTNode<T> arbol) {
  if (arbol.right != null)
     return findMax(arbol.right);
  else
     return arbol.content;
}
```

Arboles binarios de búsqueda

Operaciones básicas: Inserción



Ejemplo insertar el 6

- ***Insertar*** :Nos desplazamos dependiendo del resultado de la comparación, cuando lleguemos a una hoja insertamos.
  - Si la clave del elemento a insertar coincide con la del nodo raíz no hacemos nada
  - Si llegamos a un nodo hoja
    - **Si es menor insertamos a la izquierda**
    - **Si es mayor insertamos a la derecha**
  - Si es menor buscamos en el subárbol izquierdo
  - Si es mayor buscamos en el subárbol derecho

Arboles binarios de búsqueda

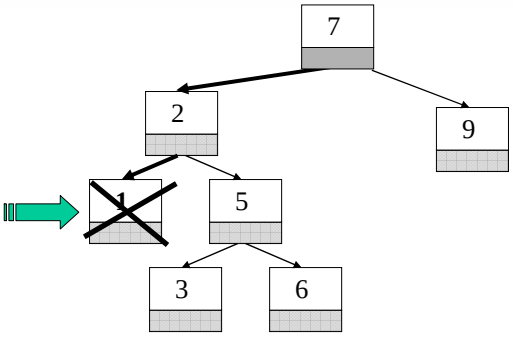
Operaciones básicas: Inserción
Implementación

```
public void insert (Comparable<T> x) {
  bTree.root = insert(bTree.root,x);}
```

```
private BTNode<T> insert1 (BTNode<T> arbol, Comparable<T> x) {
  if (arbol == null)
      arbol = new BTNode(x, null, null);
  else {
    if ( x.compareTo(arbol.content) < 0)
      arbol.left = insert1(arbol.left, x);
    if ( x.compareTo(arbol.content) > 0)
      arbol.right = insert1(arbol.right, x);
  }
  return arbol;
}
```

Arboles binarios de búsqueda

Operaciones básicas: Eliminación del menor



Se elimina el elemento del árbol ubicado **<u>más a la izquierda</u>** (Nodo.left == null)

Arboles binarios de búsqueda

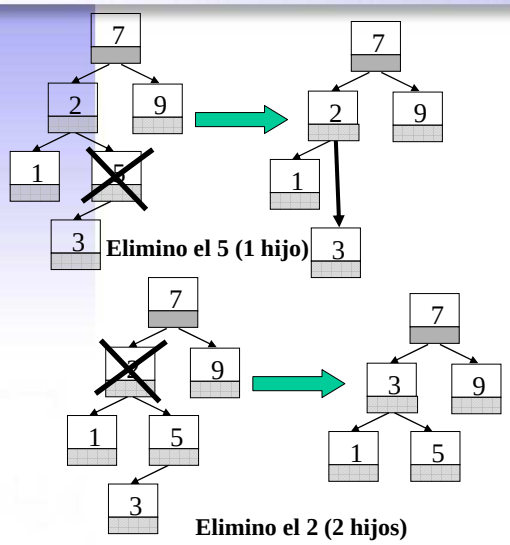
Operaciones básicas: Eliminación del menor
Implementación

```
public void removeMin() {
  bTree.root=removeMin(bTree.root);}
```

```
private BTNode<T> removeMin1(BTNode<T> arbol) {
  if (arbol.left != null)
    arbol.left = removeMin1(arbol.left);
  else
    arbol = arbol.right;
  return arbol;
}
```

Arboles binarios de búsqueda

Operaciones básicas: Eliminación



- ***Eliminar*** : Es complicado ya que los nodos internos mantienen al árbol conectado.Para eliminar:
  - Si se trata de una **hoja** se elimina directamente
  - Si tiene **un único hijo** se elimina el nodo haciendo que su nodo padre pase a referenciar a su nodo hijo
  - Si tiene **dos hijos** :
    - Se sustituye el nodo por el menor elemento de su subárbol derecho
    - Se elimina el nodo correspondiente a dicho menor elemento

Arboles binarios de búsqueda

Operaciones básicas: Eliminación

```
public void remove(Comparable<T> x) {
  bTree.root=remove(bTree.root, x);}
```

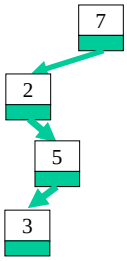
```
private BTNode<T> remove(BTNode<T> arbol, Comparable<T> x) {
 if (x.compareTo(arbol.content) > 0)
    arbol.right = remove1(arbol.right, x);
 else  if (x.compareTo(arbol.content) < 0)
    arbol.left = remove1(arbol.left, x);
 else  if (arbol.left != null && arbol.right != null) {
     arbol.content = findMin(arbol.right);
     arbol.right = removeMin(arbol.right);
  }
  else if (arbol.left!=null)
      arbol = arbol.left;
  else
      arbol = arbol .right;
 return arbol;
}
```

Arboles binarios de búsqueda

Inconvenientes

Supongamos que la entrada de datos es la siguiente:

7 2 5 3



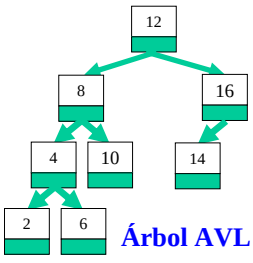
Árbol Degenerado O(N)

Problema: El tamaño del árbol depende mucho de la aleatoriedad de la entrada

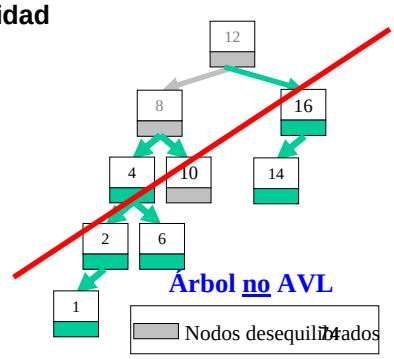
Árboles AVL

(Adelson-Velskii y Landis):

- **Definición**:
  - Son árboles binarios de búsqueda con una propiedad adicional de equilibrio:
  - Las **alturas** de los hijos derecho e izquierdo sólo pueden diferir en **1 unidad**



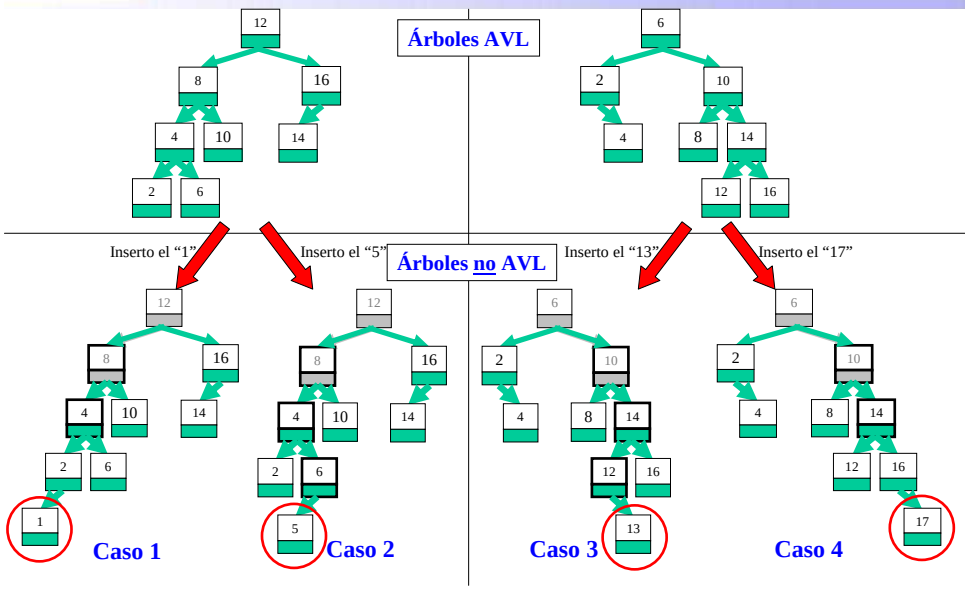
Árbol AVL



Árbol no AVL

Árboles AVL

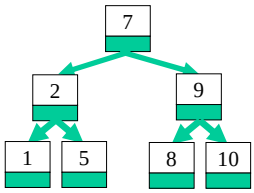
4 posibilidades de romper equilibrio al insertar:



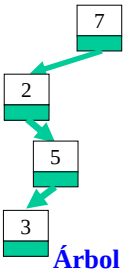
Arboles binarios de búsqueda

Complejidad de las Operaciones básicas:

- ***Coste de una operación*** es proporcional a la profundidad del último nodo accedido.
  - Si el árbol está equilibrado (AVL) logarítmico O(logN) (equivale a búsqueda binaria)
  - Si el árbol es degenerado, es lineal O(N) (equivale a búsqueda lineal)



Árbol Equilibrado O(log N)



Árbol Degenerado O(N)

Arboles binarios de búsqueda

Complejidad de las Operaciones básicas:

- Coste de una operación básica
  - Buscar
  - Insertar
  - Eliminar

  Es proporcional al nº de nodos consultados
- Coste de acceso a cada nodo:

  1+ (profundidad del nodo)

Conclusiones

Árboles Binarios de búsqueda

- Los árboles binarios de búsqueda son muy importantes en el diseño de algoritmos.
- Soportan todas las **operaciones básicas**:
  - búsqueda
  - Inserción
  - eliminación
- Todas las operaciones se realizan en **tiempo logarítmico**

Enlaces

- Un applet demostrativo de (de AVL's y otros), Arsen Gogeshvili.
  - http://webdiis.unizar.es/asignaturas/EDA/AVLTree/avltree.html