Tema 5. Árboles
Árboles Binarios de Búsqueda

Estructura de Datos y Algoritmos (EDA)

Material elaborado por los profesores de la asignatura

Índice

- Conceptos básicos
- TAD Árboles generales
- TAD Árboles binarios
- **TAD Árboles Binarios de Búsqueda (ABB)**
- Equilibrado de árboles.

Motivación de los ABB

- Los árboles binarios no ordenados son de poco interés.
  - Su única utilidad es la representación de información jerárquica (sólo grado 2!!!).
- La búsqueda en una lista ordenada es poco eficiente (O(n)).
- Los árboles binarios de búsqueda son una solución eficiente para realizar búsquedas eficientes en colecciones ordenadas de elementos.
- En inglés: Binary Search Tree (BST)

Árboles Binarios de Búsqueda (ABB)

- ABB = Árbol binario en el que TODOS sus nodos cumplen las siguientes condiciones:
  1. Cada nodo está asociado a una clave de ordenación. La clave puede ser de cualquier tipo siempre que sea un tipo comparable. Además de la clave, el nodo puede tener asociado también un elemento o valor.

  Ejemplo: ABB que almacene una agenda de contactos. La clave de cada nodo puede ser el email del contacto y el valor podría ser el nombre del contacto.

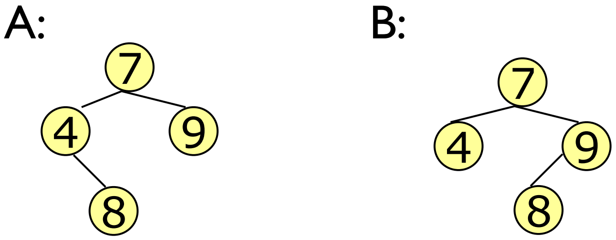
  Nota: Por simplificar, sólo representaremos las claves, que por lo general serán números enteros.

Árboles Binarios de Búsqueda (ABB)

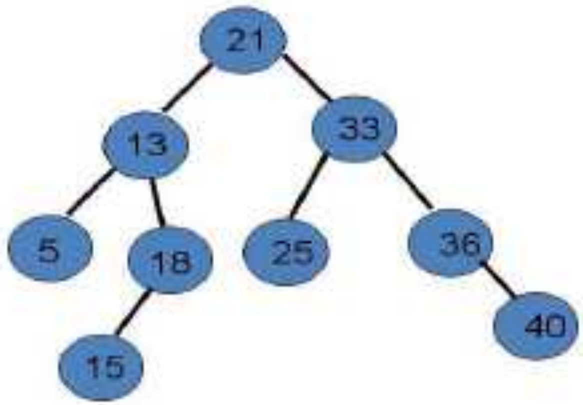
- ABB = Árbol binario en el que TODOS sus nodos cumplen las siguientes condiciones:
  1. Cada nodo está asociado a una clave de ordenación.
  2. Además para cada nodo, el valor de la clave de la raíz de su subárbol izquierdo es menor que el valor de la clave del nodo, y
  3. El valor de la clave raíz del subárbol derecho es mayor que el valor de la clave del nodo.

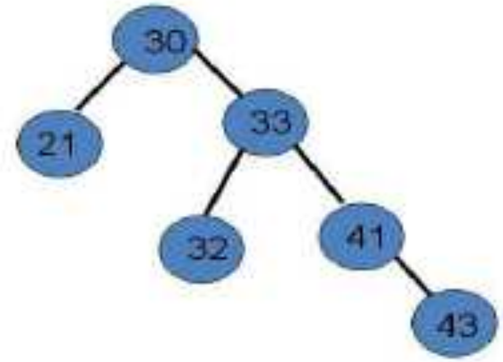
Árboles Binarios de Búsqueda (ABB)

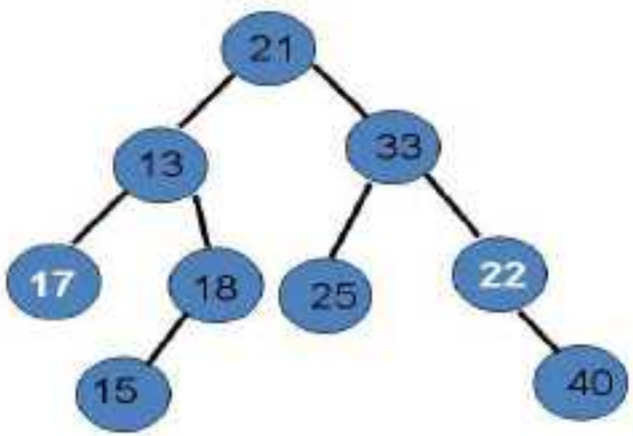
- ¿Cuál de estos dos árboles binarios de enteros es un ABB?

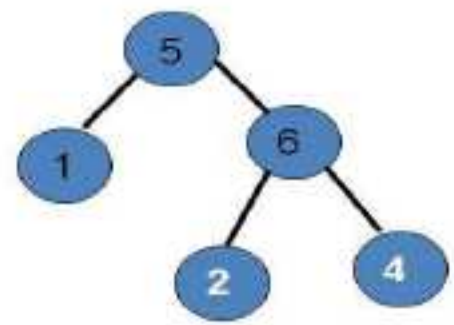


Árboles Binarios de Búsqueda (ABB)

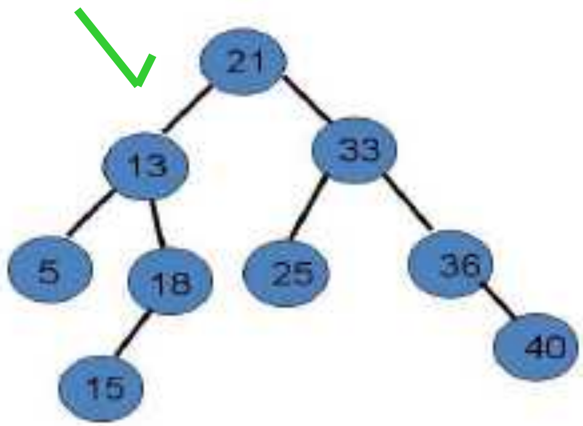


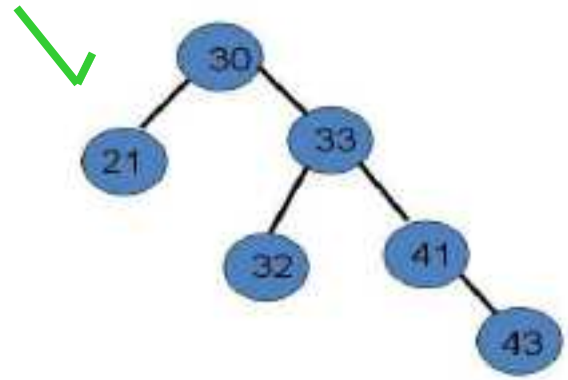


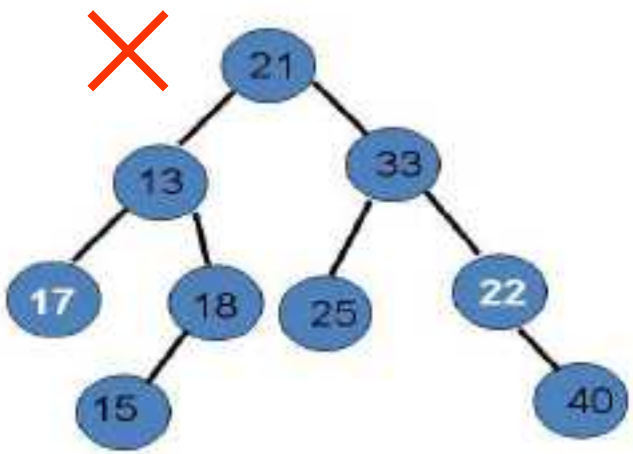


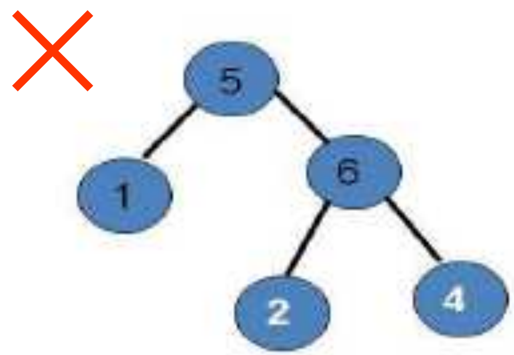


Árboles Binarios de Búsqueda (ABB)









Árboles Binarios de Búsqueda (ABB)

- Utilidad
  - Almacenar estructuras lineales (que normalmente serían listas) **mejorando la complejidad de las búsquedas**
    - En el caso peor
    - En el caso medio
- Motivación: AB no ordenados son de poco interés
  - La falta de ordenación en un AB hace injustificable una estructura enlazada de árbol, prefiriéndose una lista
- Problema
  - un ABB puede llegar a degenerar en una lista
- Solución: **equilibrio en altura de un ABB (tema siguiente)**

Árboles Binarios de Búsqueda (ABB)

- Ventajas
  - Permite almacenar una colección ordenada de elementos de una forma más eficiente al mejorar **la complejidad de las búsquedas.**
  - El número de accesos al árbol es menor que en una lista**.**

Especificación de un ABB

- Aunque las claves y valores de un nodo binario de búsqueda pueden ser de cualquier tipo (las claves deben ser de un tipo comparable), por simplificar nos centraremos en un ABB con claves enteras y valores de tipo String.

Especificación de un TAD ABB (=BST)

```java
public interface IBSTree{

    //number of nodes
    public int getSize();

    //length of the largest plus 1
    public int getHeight();

    //shows the preorder tree trasversal
    public void showPreOrder();

    //shows the in-order tree trasversal
    public void showInOrder();

    //shows the post-order tree trasversal
    public void showPostOrder();

    //shows the level order tree trasversal
    public void showLevelOrder();
```

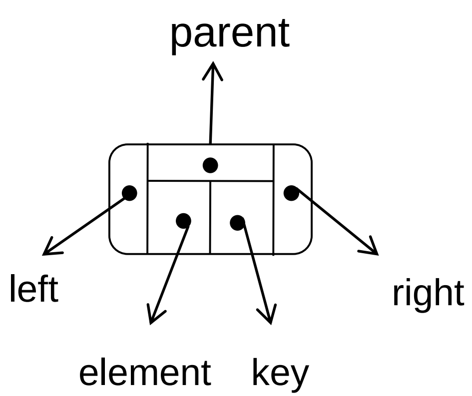
Especificación de un TAD ABB (cont.)

```
//inserts a new node
public void insert(int key, String elem);
//removes the node with key
public void remove(int key);
//searches the node with key and returns its elem
public String find(int key);
```

Árboles binarios de búsqueda: Implementación **Clase BSTNode**

```
public class BSTNode {
    public int key;
    public String elem;

    public BSTNode parent;
    public BSTNode leftChild;
    public BSTNode rightChild;

    public BSTNode(int key, String element) {
        this.key = key;
        this.elem = element;
    }
}
```

La clave tiene que ser siempre de un tipo comparable



Árboles binarios de búsqueda: Implementación **Clase BSTree**

```
public class BSTree implements IBSTree  {

    BSTNode root;

    public BSTree() {

    }
    public BSTree(BSTNode root) {
        this.root=root;
    }
```

Implementación del método getSize

El tamaño del un árbol se puede ver como el tamaño del subárbol que cuelga de la raiz

```
public int getSize() {
    return getSize(root);
}

//auxiliary recursive method to calculate the size of a subtree (node)
public int getSize(BSTNode node) {
    if (node == null) {
        return 0;
    } else {
        int result = 1 + getSize(node.leftChild) + getSize(node.leftChild);
        return result;
    }
}
```

Podemos usar un método auxiliar para definir la recursión sobre un nodo cualquiera.

Método showInOrder()

```java
public void showInOrder() {
    showInOrder(root);
}
public void showInOrder(BSTNode node) {
    //base case, we do not show anything
    if (node == null) return;

    //first, we visit the left child
    showInOrder(node.leftChild);
    //now, we visit the root.
    System.out.println(node.elem);
    //finally, we visit the right child
    showInOrder(node.rightChild);
}
```

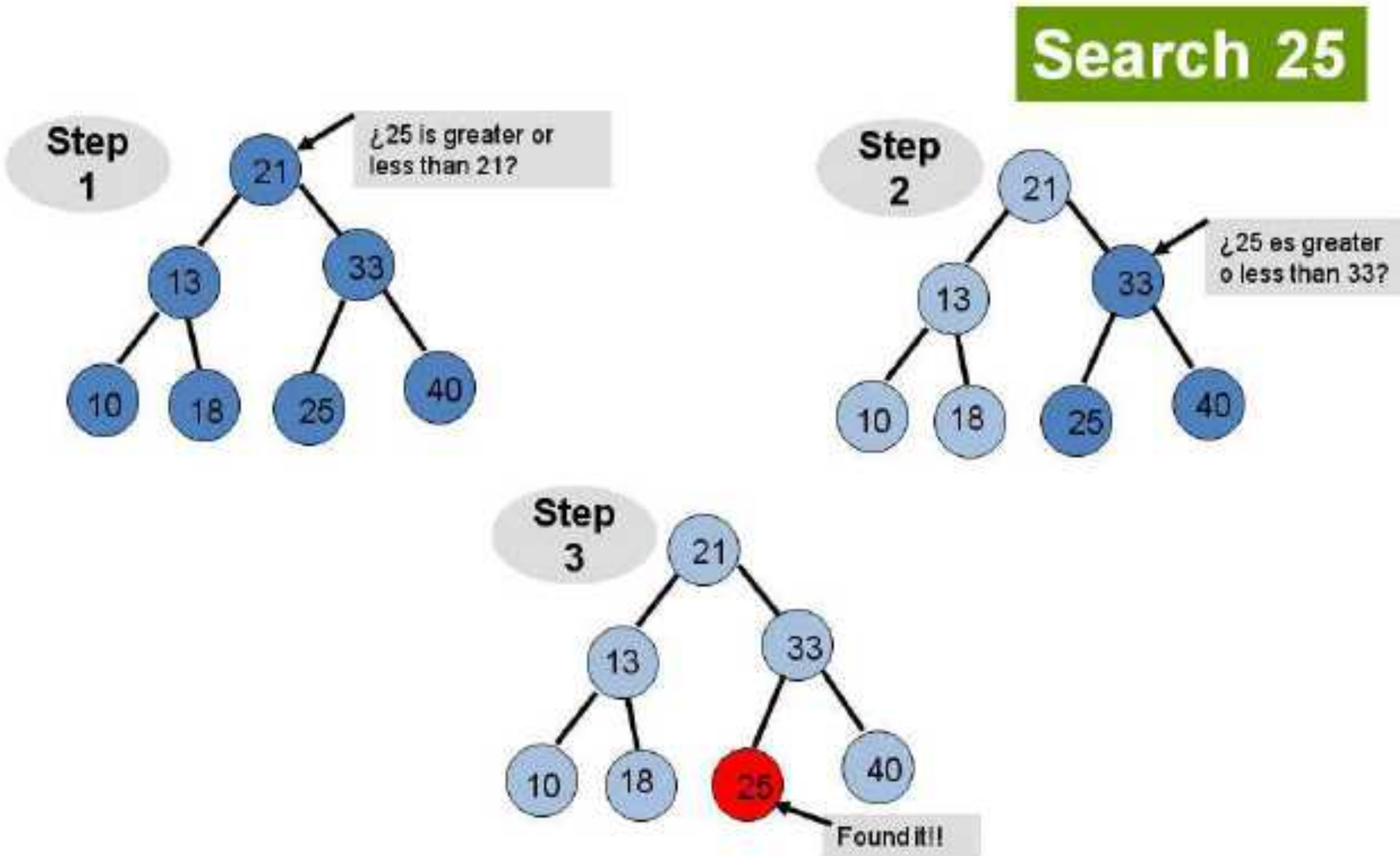
Problemas a resolver

- Ejercicio implementa el método **getHeight** que devuelve la altura de un nodo. Recuerda que la altura de un árbol es la longitud de su rama más larga + 1.
- Implementa el método que devuelve la altura del árbol.
- Implementa el método **getDepth** que devuelve la profundidad de un nodo.
- Implementa los métodos **showPreOrder** y **showPostOrder**

TAD ABB: Búsqueda

- Funcionamiento:
  - se va recorriendo el árbol
  - si el nodo actual no es el buscado se decide si hay que buscar por la derecha o la izquierda
  - el algoritmo para al encontrar el nodo o llegar al árbol vacío
- Puede desarrollarse:
  - como algoritmo recursivo del nodo del árbol
  - como algoritmo iterativo del árbol

TAD ABB: ejemplo de búsqueda



Búsqueda (versión iterativa)

```
public String findIt(int key) {

    BSTNode searchNode=root;
    while (searchNode!=null) {
        int keyVisit=searchNode.key;
        if (key==keyVisit) {
            //found it!!!
            return searchNode.elem;
        } else if (key<keyVisit) {
            searchNode=searchNode.leftChild;
        } else {
            searchNode=searchNode.rightChild;
        }
    }
    System.out.println(key + " does not exist");
    return null;
}
```

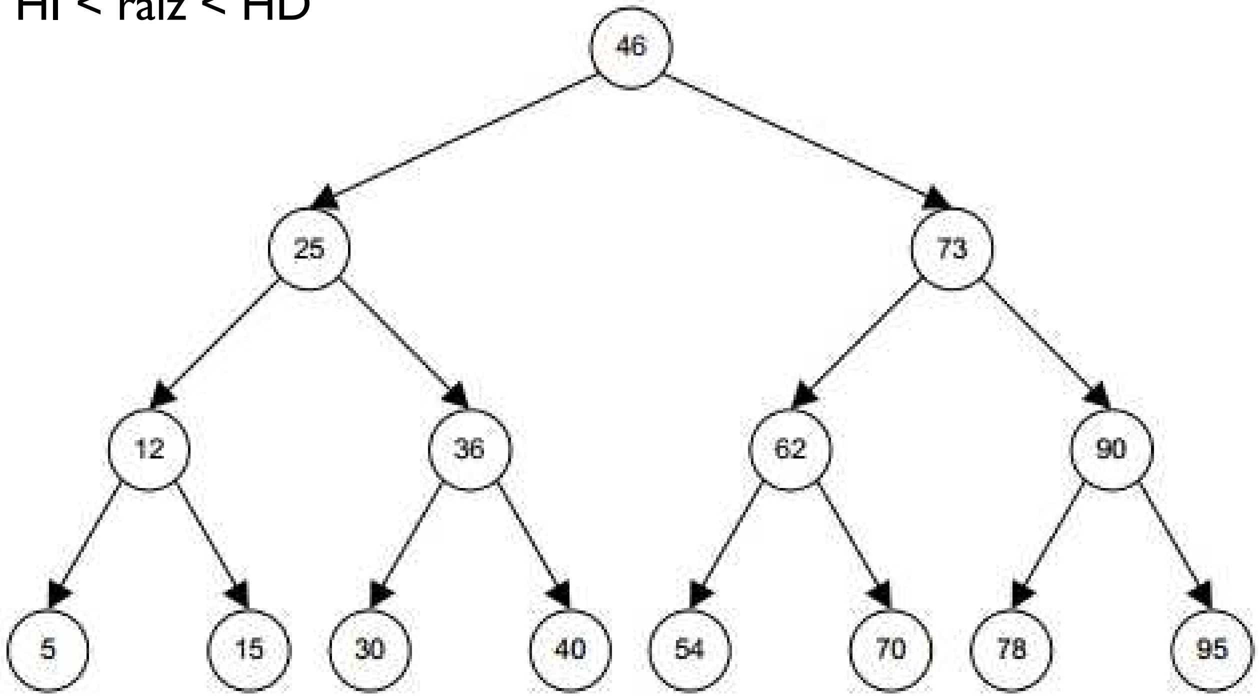
Búsqueda recursiva

```
public String find(int key) {

    return find(root,key);

}
//auxiliary recursive method to search a node
//inside a subtree
private static String find(BSTNode currentNode, Integer key) {
String result=null;
if (currentNode == null) {
//System.out.prinltn(key + " does not exist!");
} else {
if (key.equals(currentNode.key))
result= currentNode.elem;
else if (key.compareTo(currentNode.key) < 0)
result=find(currentNode.left, key);
else
result=find(currentNode.right, key);
}
return result;
}
```

Árboles Binarios de Búsqueda (ABB)

¿Cuál es la complejidad de la operación de búsqueda?
Buscar 70

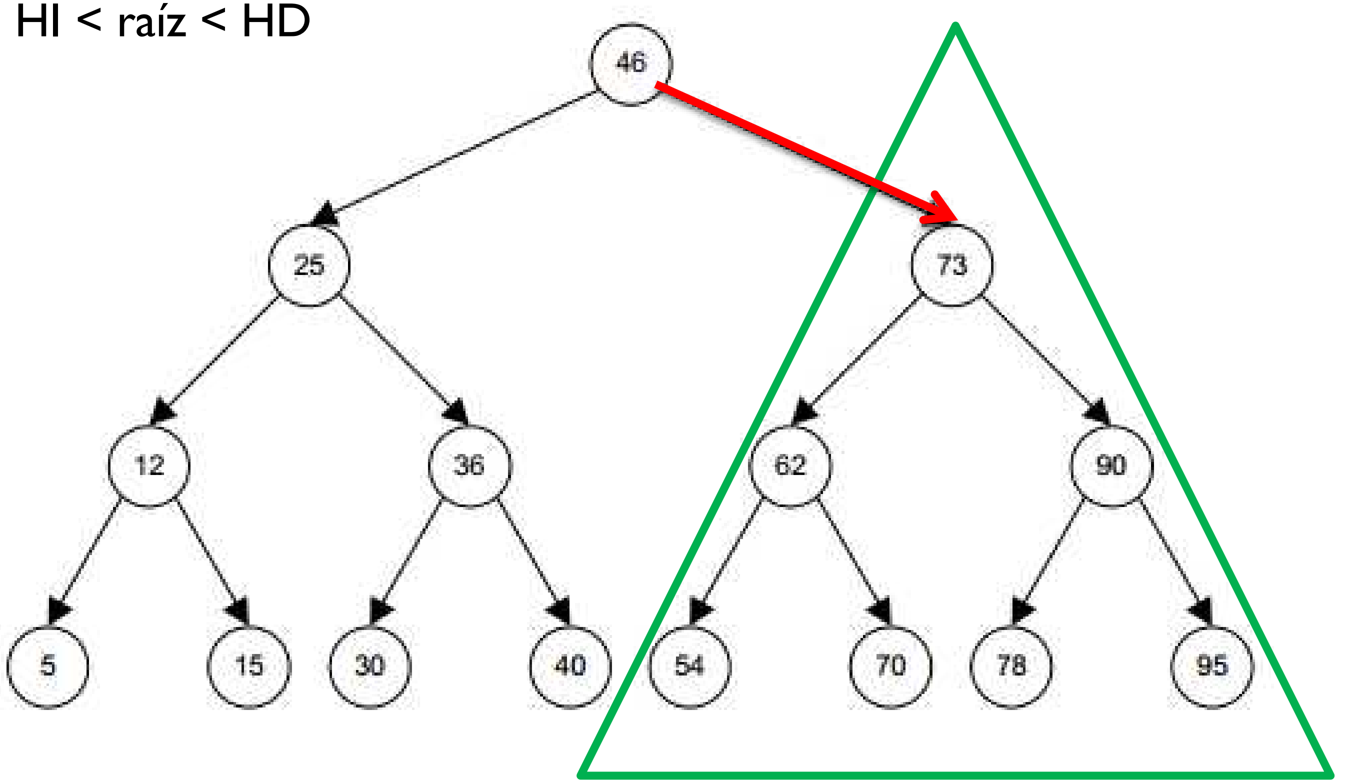
HI < raíz < HD



Árboles Binarios de Búsqueda (ABB)

Buscar 70

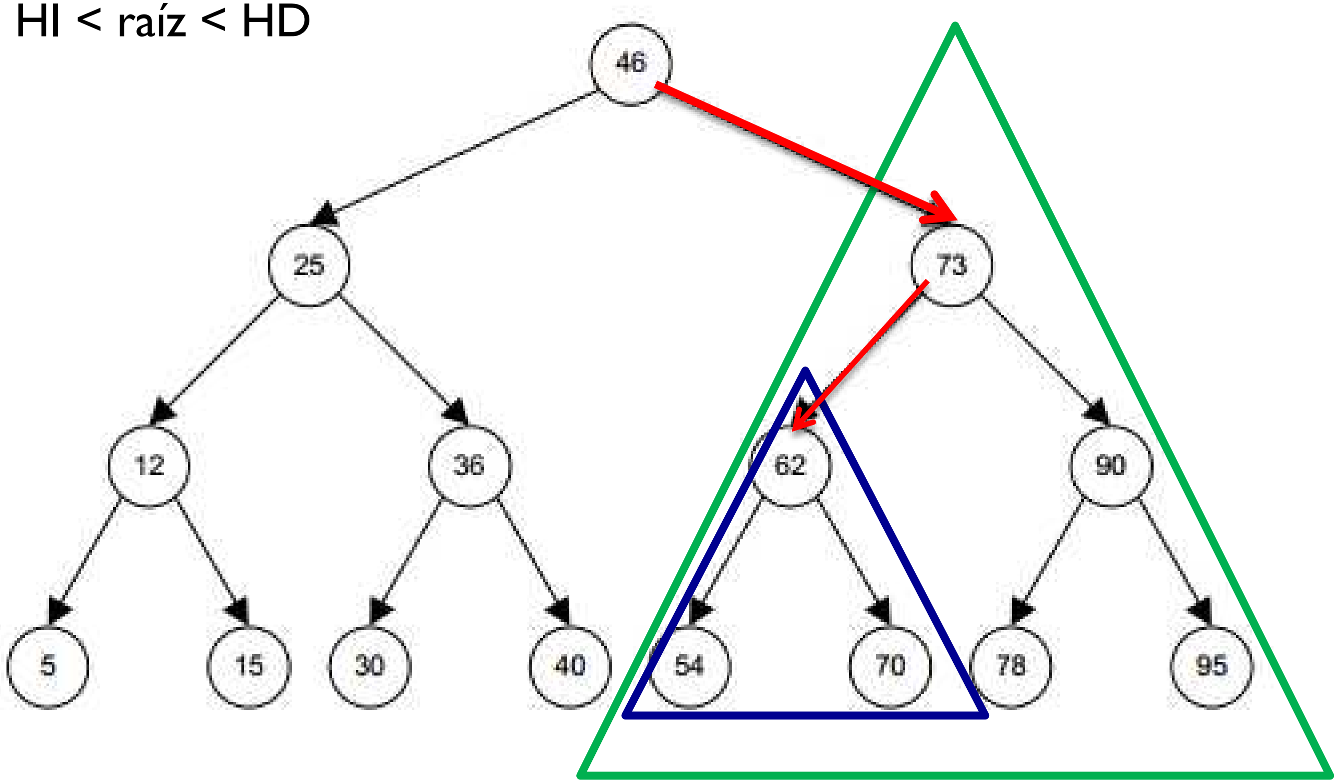
HI < raíz < HD



Árboles Binarios de Búsqueda (ABB)

Buscar 70

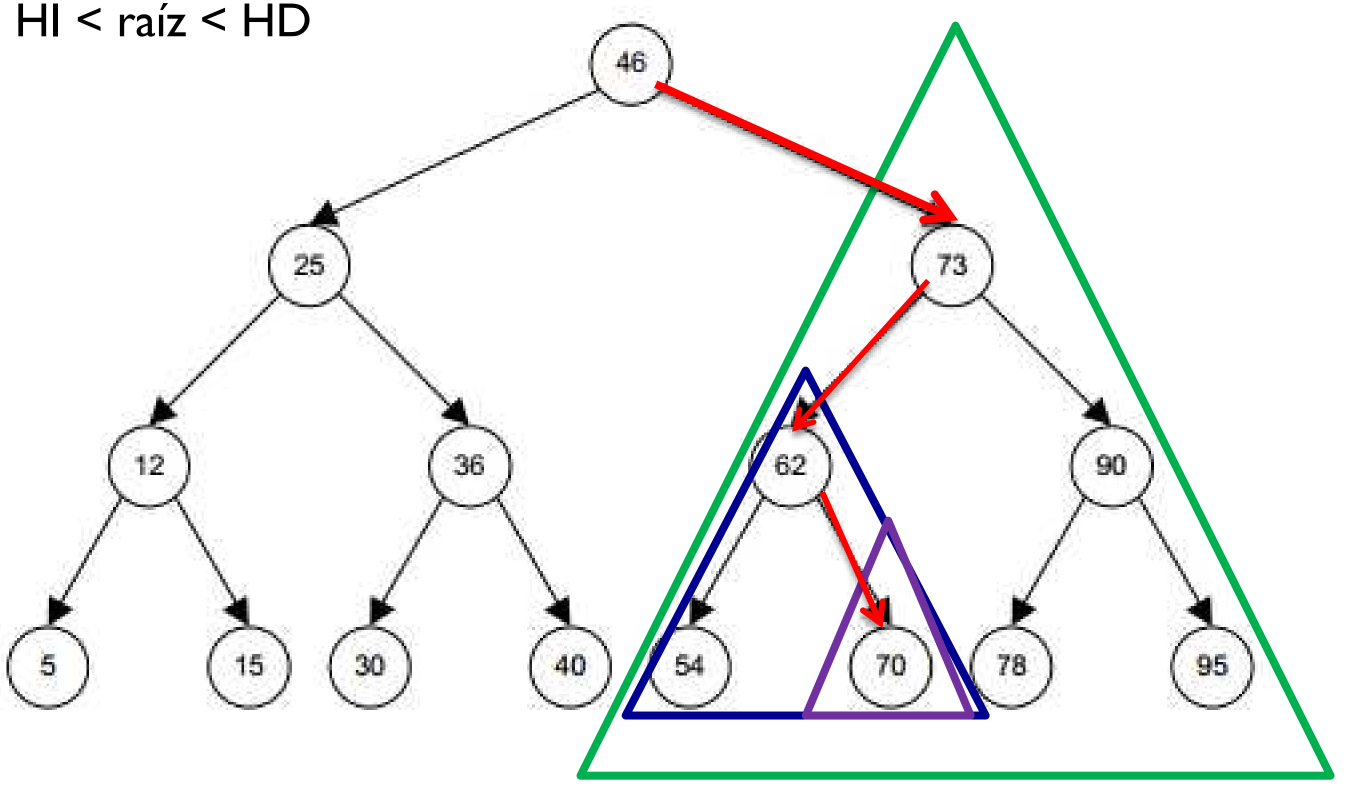
HI < raíz < HD



Árboles Binarios de Búsqueda (ABB)

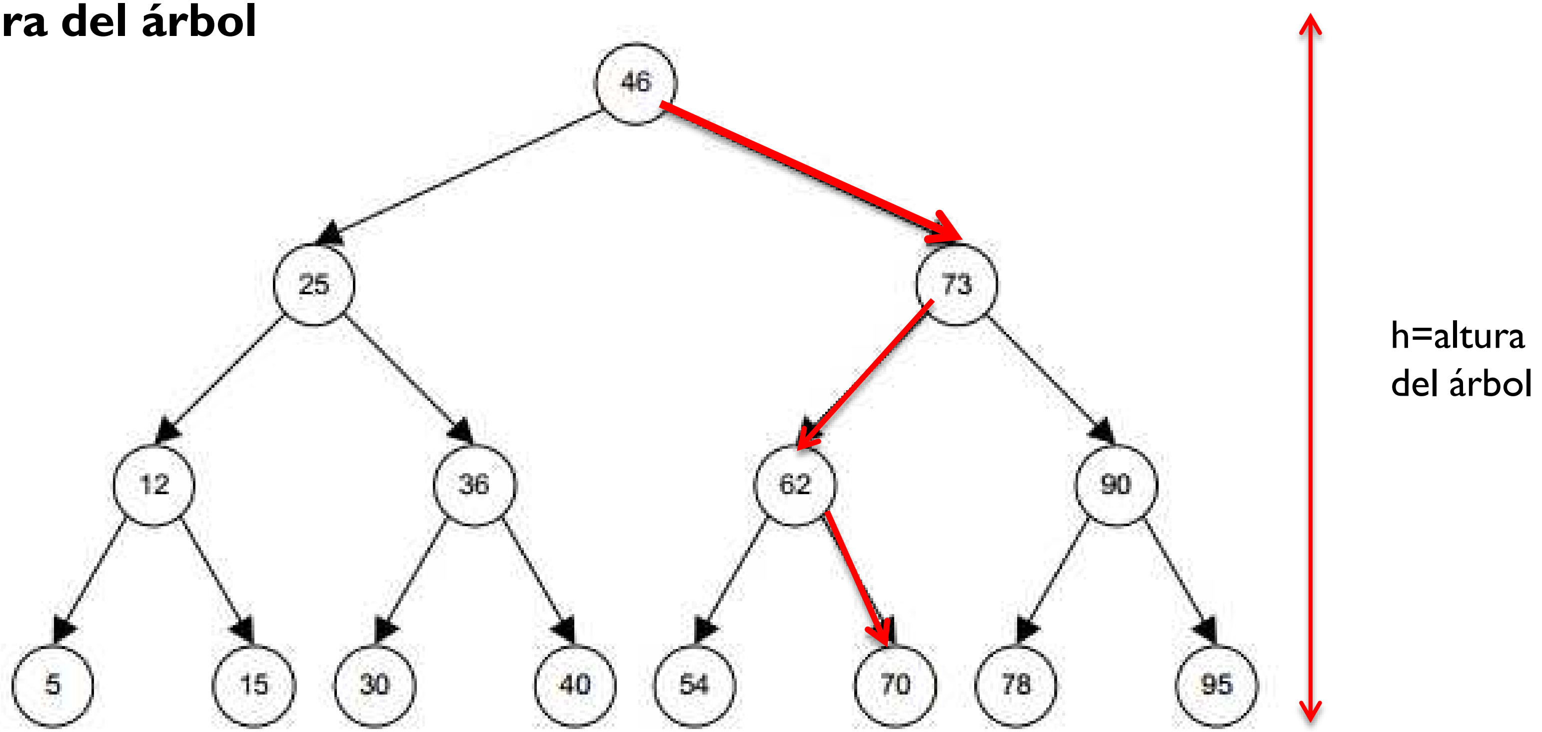
Buscar 70

HI < raíz < HD



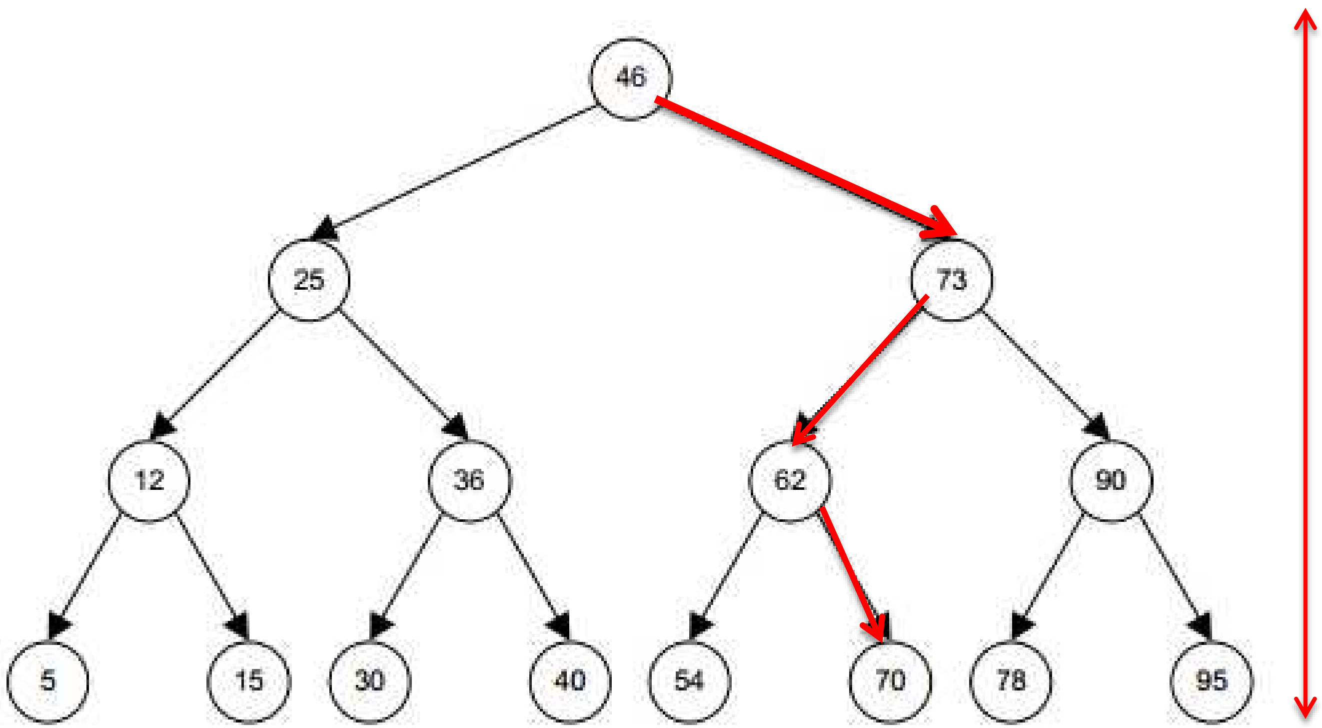
Árboles Binarios de Búsqueda (ABB)

En el peor de los casos, en la búsqueda se realiza h comparaciones, donde h es la altura del árbol



Árboles Binarios de Búsqueda (ABB)

Por tanto, la complejidad será O(h) donde h es la altura del árbol

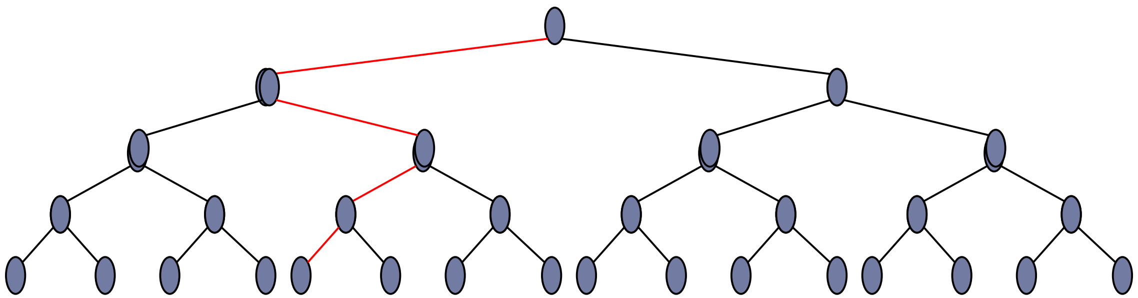


h=altura del árbol

Árboles Binarios de Búsqueda (ABB)

Siempre dividimos el espacio de búsqueda en 2

- $Paso_1 = n / 2^1$
- $Paso_2 = n / 2^2$
- $Paso_3 = n / 2^3$
- .
- .
- $Paso_h = n / 2^h = 1$, donde h es la altura del árbol



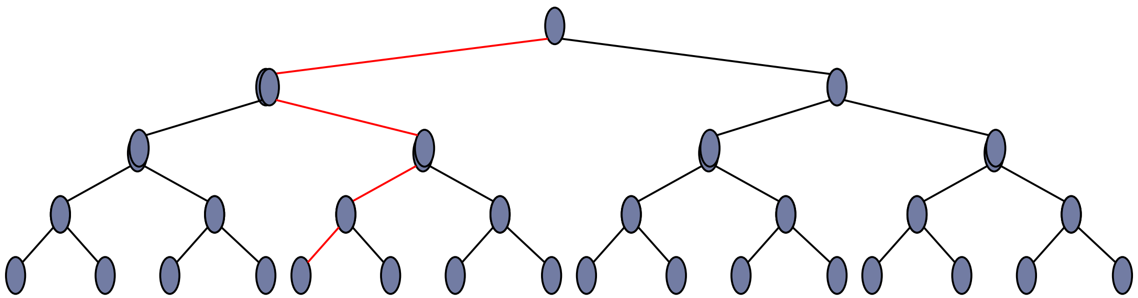
$n = 2^h$
$h = \log_2(n)$

En el peor de los casos, el número de pasos necesarios para buscar un nodo será igual a la altura del árbol = h

Árboles Binarios de Búsqueda (ABB)

Siempre dividimos el espacio de búsqueda en 2

- $Paso_1 = n / 2^1$
- $Paso_2 = n / 2^2$
- $Paso_3 = n / 2^3$
- .
- .
- $Paso_h = n / 2^h = 1$, donde h es la altura del árbol

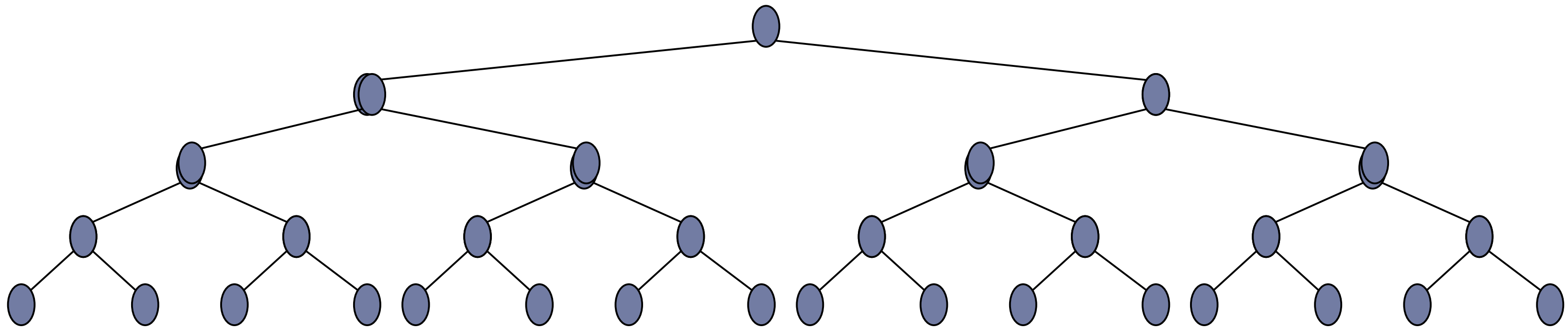


$$n = 2^h$$
$$h = \log_2(n)$$

$$\mathbf{O(h)=O(\log_2 n)}$$

Árboles Binarios de Búsqueda (ABB)

Complejidad Búsqueda binaria $O(\mathrm{Log}_2 n)$



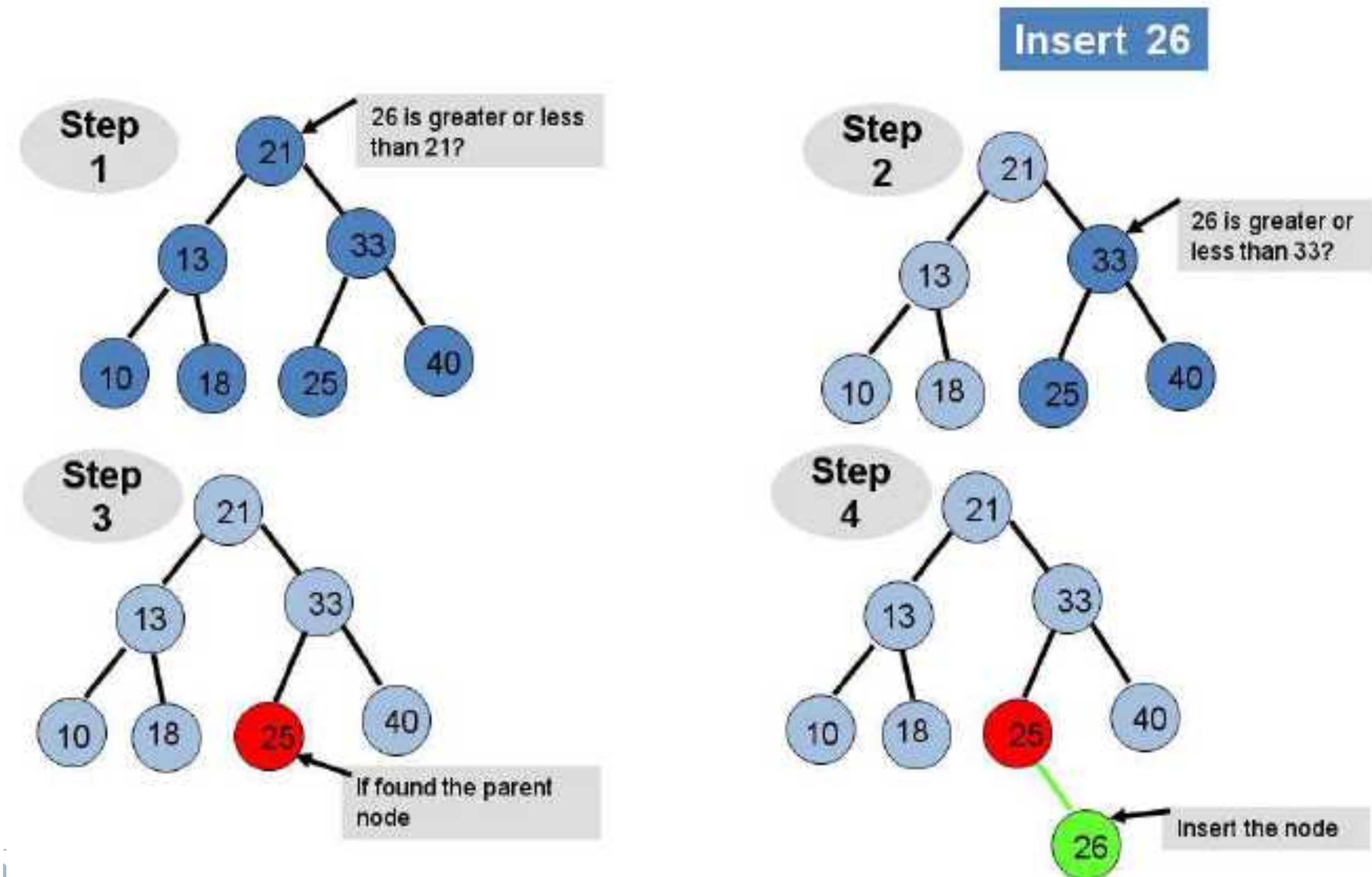
En un árbol de n nodos, el camino más largo que hay que recorrer es de orden log n.

- Si n=32 => log 32 = 5
- Si n=1024 => log 1024= 10

TAD ABB: inserción

- Los nodos se insertan siempre como nodos hoja
- El algoritmo de inserción garantiza para cada nodo del árbol que:
  - Su subárbol izquierdo contiene claves menores
  - Su subárbol derecho contiene claves mayores
- Funcionamiento:
  - si el árbol estuviera vacío, se inserta el nodo en la raíz
  - si no, se va recorriendo el árbol:
    - en cada nodo se decide si hay que insertar a la derecha o la izquierda
    - si el subárbol en que hay que insertar es vacío, se inserta el nuevo elemento
    - si el subárbol en que hay que insertar no es vacío hay que recorrerlo hasta encontrar el lugar que le corresponde al nodo en ese subárbol
  - es un algoritmo recursivo

TAD ABB: ejemplo de inserción



TAD ABB: ejemplo de inserción

Insertar 8, 5, 1, 4, 6, 7, 20, 12, 23

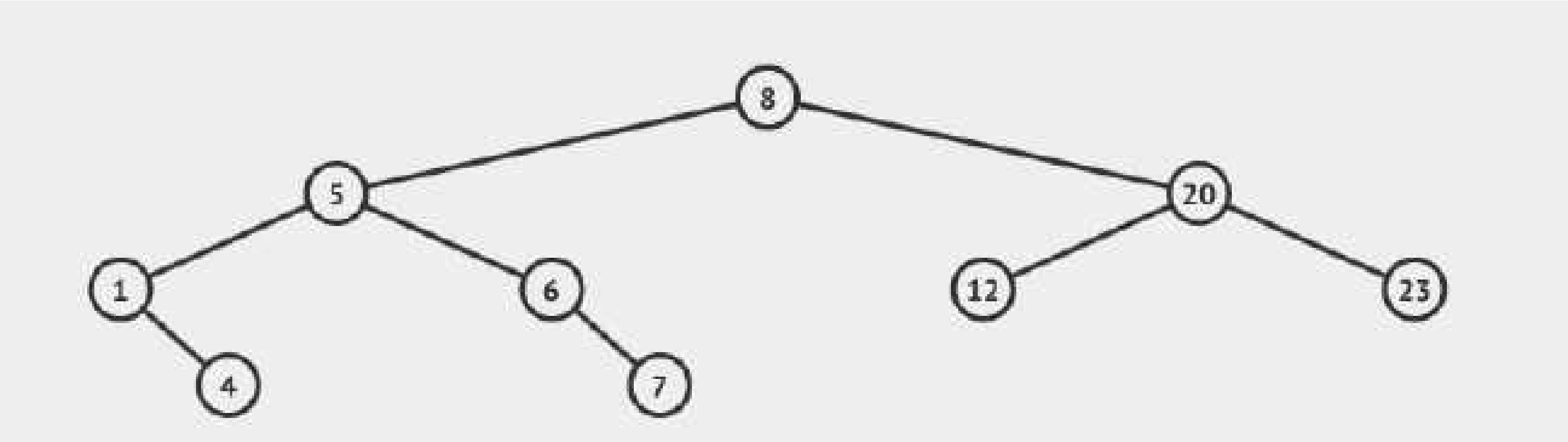


ABB: Implementación **Inserción (1/2)**

```
public void insert(int key, String element) {
    BSTNode newNode=new BSTNode(key,element);
    if (root==null) root=newNode;
    else insert(newNode, root);
}
```

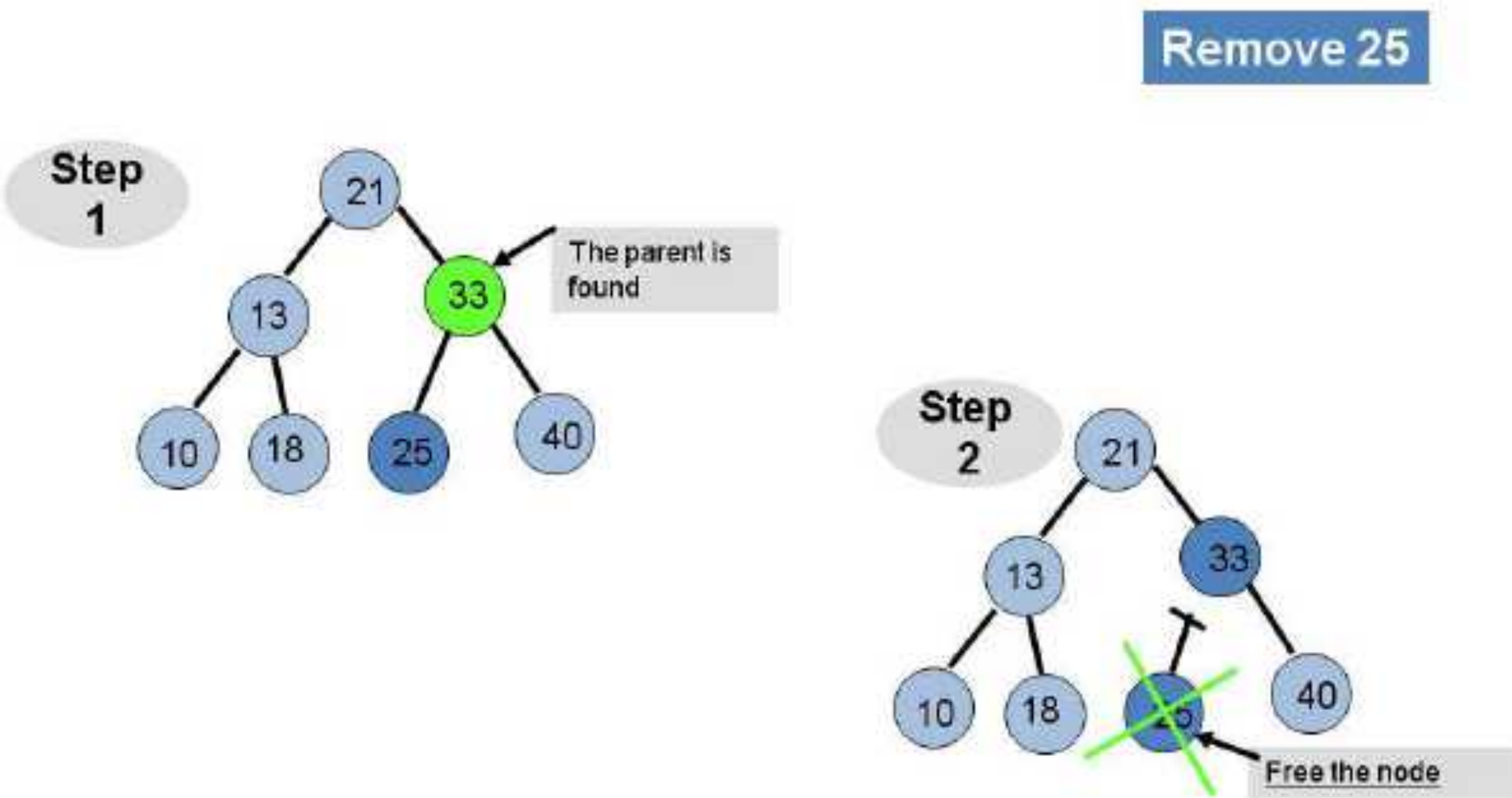
ABB: Implementación **Inserción (2/2)**

```java
public void insert(BSTNode newNode, BSTNode node) {

    int key=newNode.key;
    if (key==node.key) {
        System.out.println(key + " already exists. Duplicates are not allowed!!!.");
        return;
    }
    if (key<node.key) {
        if (node.leftChild==null) {
            node.leftChild=newNode;
            newNode.parent=node;
        } else insert(newNode,node.leftChild);
    } else {
        if (node.rightChild==null) {
            node.rightChild=newNode;
            newNode.parent=node;
        } else insert(newNode,node.rightChild);
    }


}
```

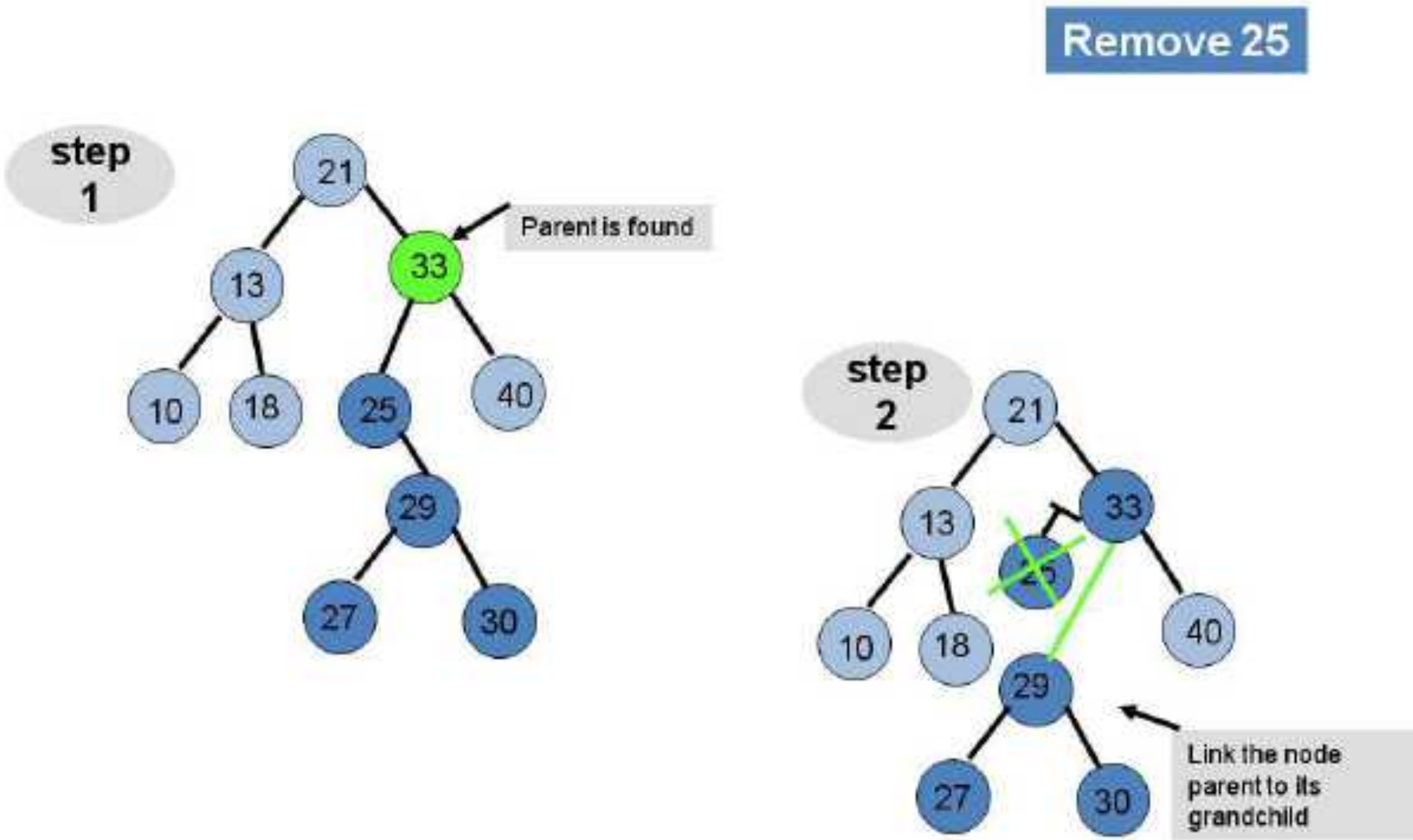
TAD ABB: Borrado

- Funcionamiento: Buscar el nodo a borrar,
  1. Si es un **nodo hoja**, basta con que su padre haga referencia a vacío
  2. Si **no es nodo hoja** hay que sustituirlo por otro
     a) El nodo a borrar sólo tiene un hijo: sustituirlo por su hijo
     b) El nodo a borrar tiene dos hijos, sustituirlo por:
        - El **mayor** de su subárbol izquierdo (predecesor) o
        - El **menor** de su subárbol derecho (sucesor)
     - En realidad, no se sustituye un nodo por otro, sino que se reemplazan la clave y valor del nodo por la clave y valor del predecesor (o sucesor, según la estrategia escogida). Además, será necesario eliminar dicho nodo predecesor (o sucesor), que siempre será un nodo hoja o un nodo con un único hijo.
  - Si el nodo a borrar es la raíz, hay que modificar la raíz, aplicando el caso que corresponda

TAD ABB: Ejemplo de borrado



TAD ABB: Ejemplo de borrado



TAD ABB: Ejemplo de borrado

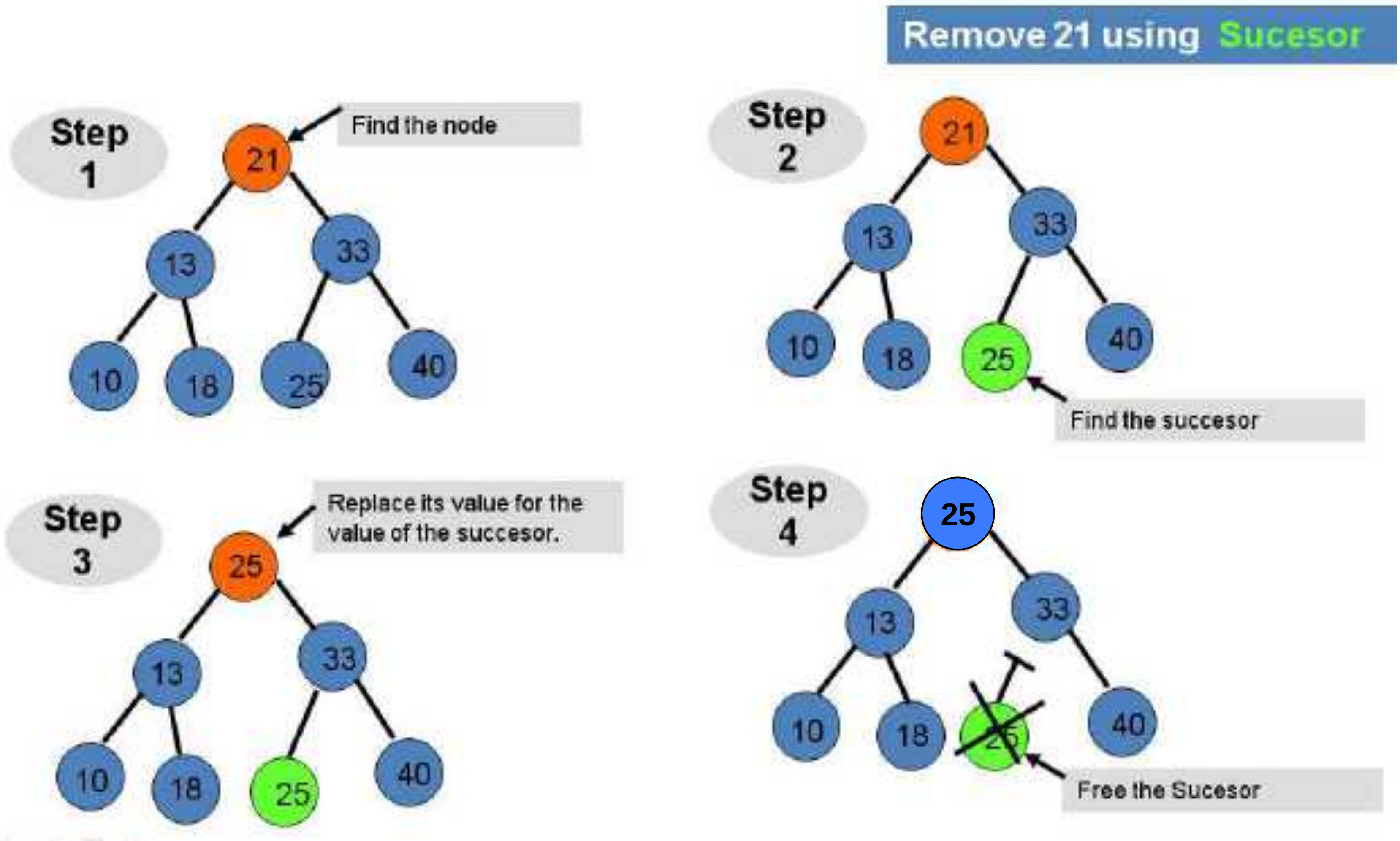


ABB: Implementación Borrado **(1/10)**

```
public void remove(int key) {
    if (root == null) {
        System.out.println("Cannot remove: The tree is empty");
        return;
    }

    //removing the root is a special case
    if (key==root.key) removeRoot();
    else remove(key,root);

}
```

ABB: Implementación Borrado **(2/10)**

```
public void removeRoot() {
    //if the root is a leave, then root should be null
    if (root.leftChild==null && root.rightChild==null) root=null;
    //the root only has a child
    else if (root.leftChild==null ||root.rightChild==null)  {
        if (root.leftChild==null) root=root.rightChild;
        else  root=root.leftChild;
        root.parent=null;
    } else {
        remove(root.key,root);
    }
}
```

ABB: Implementación Borrado **(3/10)**

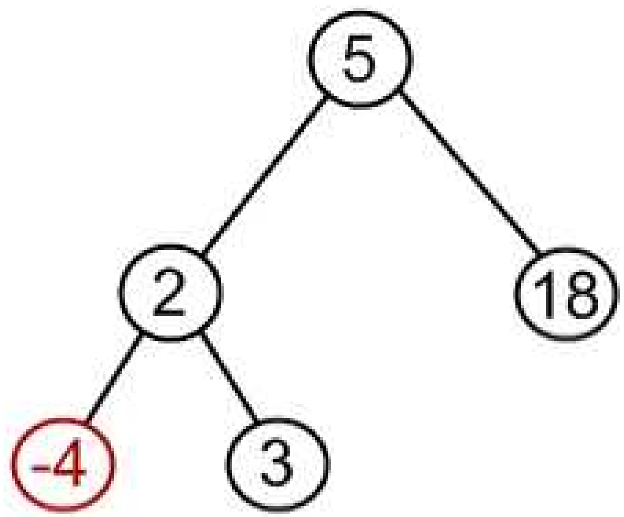
Lo primero es encontrar el nodo a borrar. Eso se puede conseguir mediante llamadas recursivas del método remove sobre el subárbol izquierdo o derecho, dependiendo de si la clave buscada es menor o mayor que la clave del nodo actual.

```
private boolean remove(int key, BSTNode node) {
    if (node == null) {
        System.out.println("Cannot remove: The key doesn't exist");
        return false;
    }

    if (key<node.key) return remove(key,node.leftChild);

    if (key>node.key) return remove(key,node.rightChild);
```

Nota que si llegamos a un nodo nulo, quiere decir que la clave no ha sido encontrada.

ABB: Implementación Borrado **(4/10)**

1. **Una vez localizado, distinguimos 3 casos:**
   PRIMER CASO: El nodo a borrar es una hoja => Rompemos la referencia del padre a ese nodo.



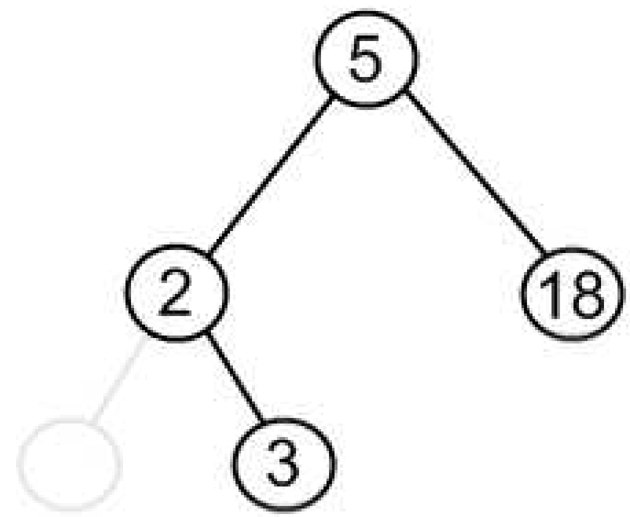


ABB: Implementación Borrado **(5/10)**

1. **Una vez localizado, distinguimos 3 casos:**
   PRIMER CASO: El nodo a borrar es una hoja => Rompemos la referencia del padre a ese nodo.

```
//First case: the node is a leaf.
if (node.leftChild==null && node.rightChild==null) {
    BSTNode parent=node.parent;

    //we must break the references between the node and its parent.
    //We have to find out if node is the left
    //or right child of its parent.
    if (parent.leftChild==node) parent.leftChild=null;
    else parent.rightChild=null;
    return true;
}
```

ABB: Implementación Borrado **(6/10)**

SEGUNDO CASO: El nodo a borrar tiene un único hijo => su hijo debe ser conectado directamente al padre del nodo que queremos borrar

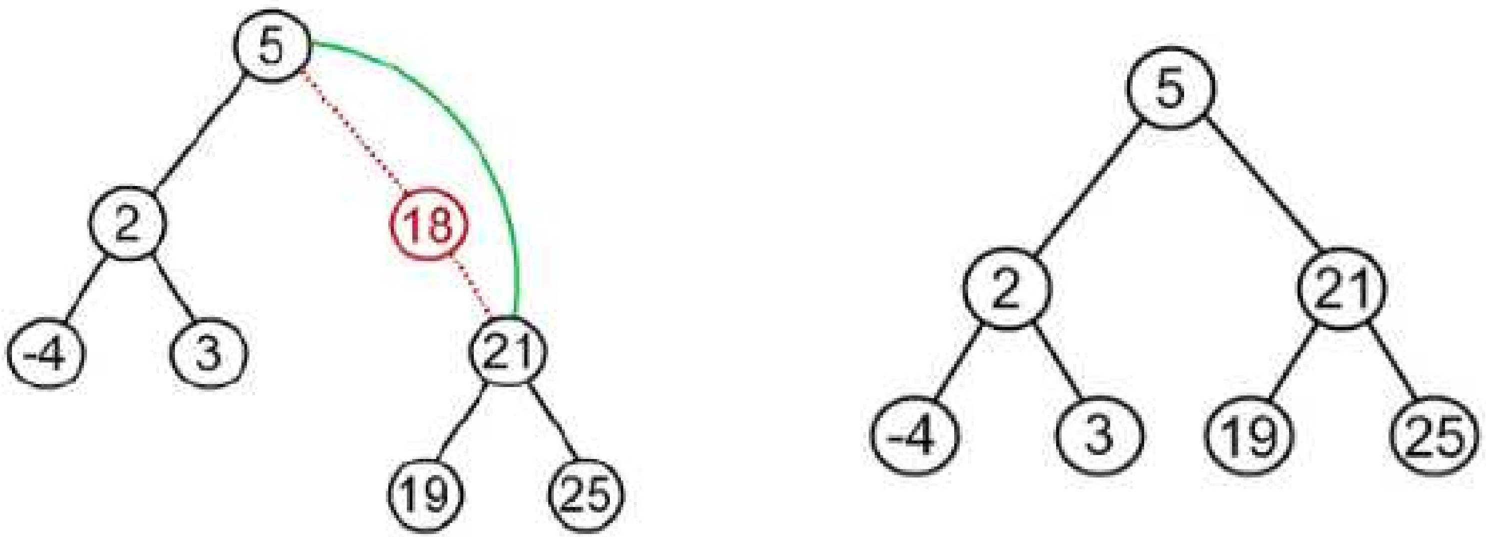


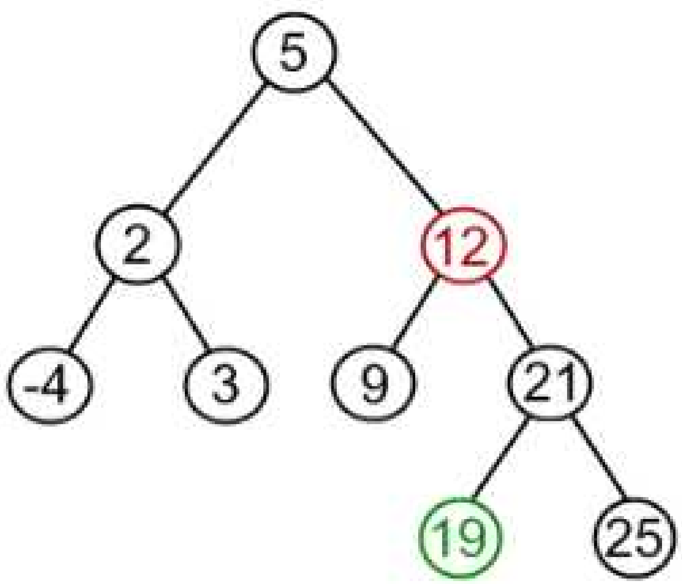
ABB: Implementación Borrado **(7/10)**

SEGUNDO CASO: El nodo a borrar tiene un único hijo => su hijo debe ser conectado directamente al padre del borro que queremos borrar

```
//Second case is one the node only has a child: left or right
if (node.leftChild==null ||node.rightChild==null){
    //its only child is its right child
    BSTNode grandChild=null;
    if (node.leftChild==null)
        grandChild=node.rightChild;
    else
        grandChild=node.leftChild;

    BSTNode grandParent=node.parent;
    if (grandParent.leftChild==node)
        grandParent.leftChild=grandChild;
    else
        grandParent.rightChild=grandChild;
    //the grand child must point its grand parent.
    grandChild.parent=grandParent;
    return true;
}
```

ABB: Implementación Borrado **(8/10)**

TERCER CASO: El nodo a borrar tiene dos hijos =>



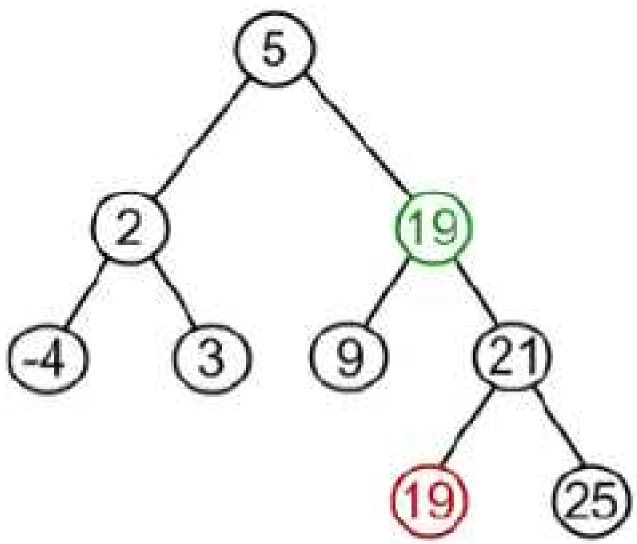

```
private BSTNode findMin(BSTNode node) {
    if (node==null) return null;
    BSTNode minNode=node;
    while (minNode.leftChild!=null) {
        minNode=minNode.leftChild;
    }
    return minNode;
}
```

Buscamos su sucesor. También se puede usar el predecesor

Para encontrar el sucesor, deberemos buscar en su hijo derecho, el nodo con la key más pequeña.

ABB: Implementación Borrado **(9/10)**

TERCER CASO: El nodo a borrar tiene dos hijos =>



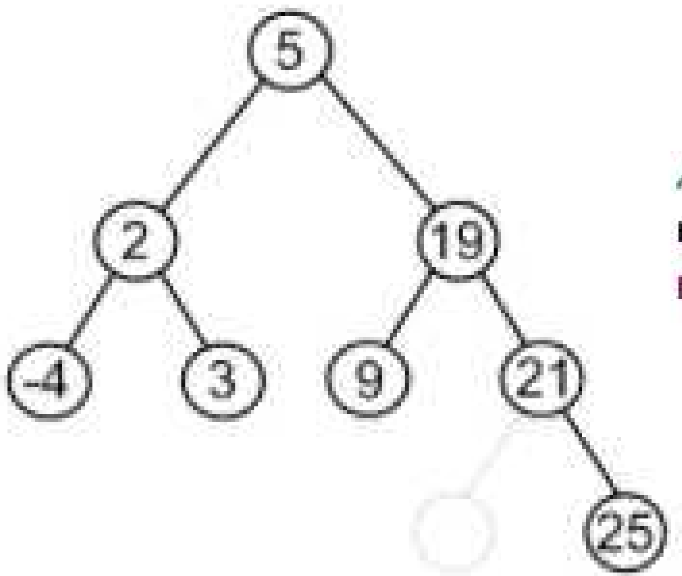

```
//Third case: node has two childs.
//We can replace its value by
//the minimum value in its right child

BSTNode sucesorNode = findMin(node.rightChild);

node.elem=sucesorNode.elem;
node.key=sucesorNode.key;
```

Reemplazamos su clave y elemento por los del nodo sucesor.

ABB: Implementación Borrado **(10/10)**

TERCER CASO: El nodo a borrar tiene dos hijos =>

Finalmente, borramos el sucesor

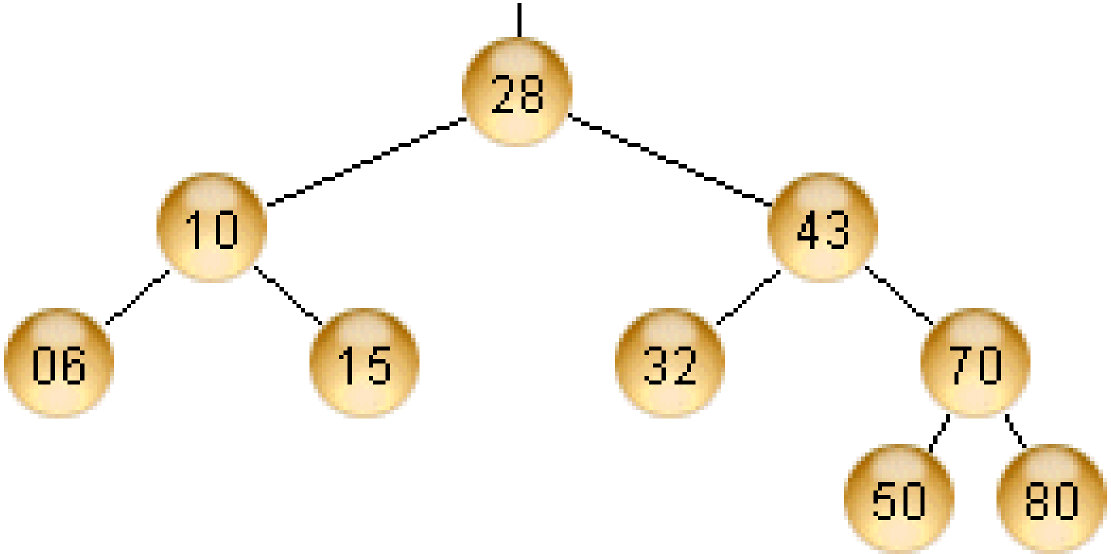



```
//Finally, we must remove the sucesorNode
remove(sucesorNode.key,node.rightChild);
return true;
```

Nota que el sucesor siempre será un nodo hoja o un nodo con un único hijo.

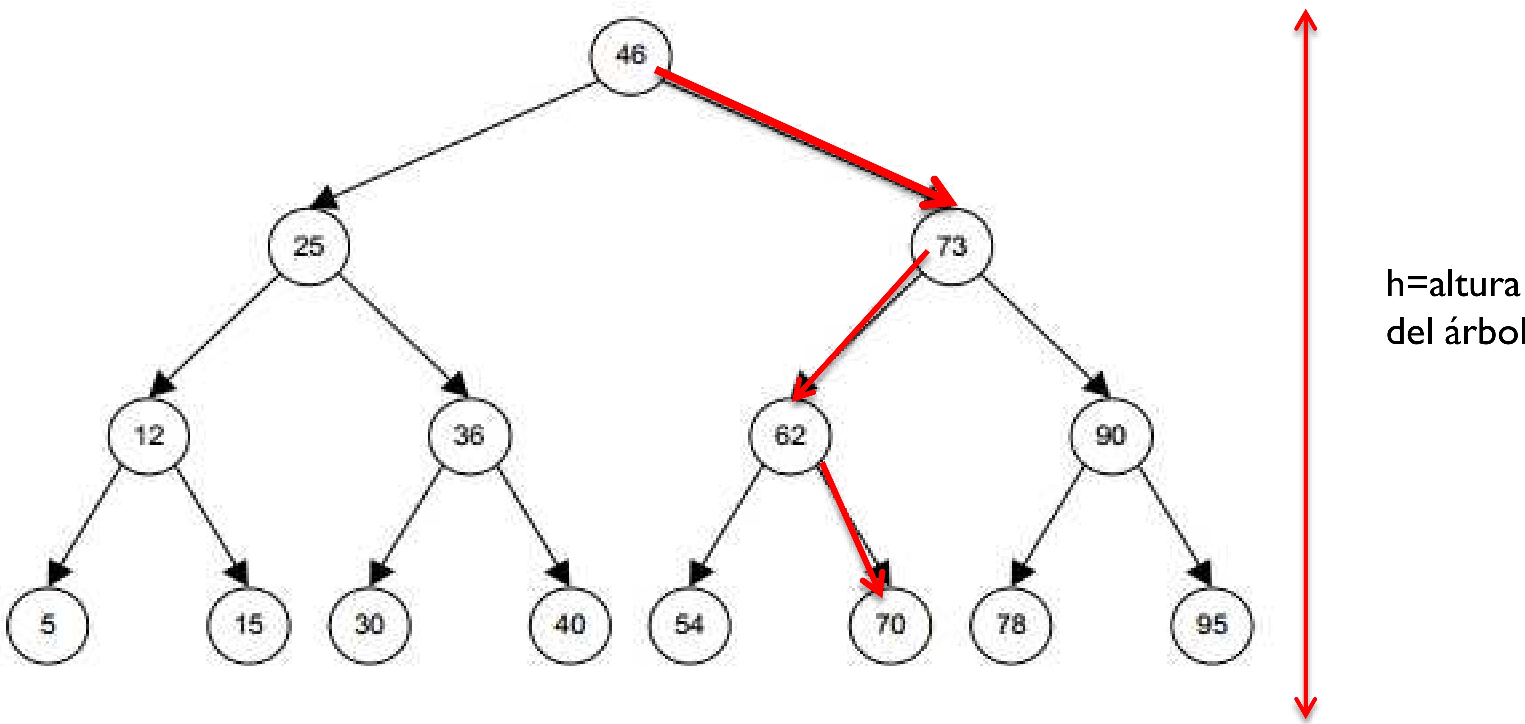
TAD ABB: Ejercicios. Borrado

- Elimina la secuencia 70, 15, 28, 43, 32 del siguiente árbol



Árboles Binarios de Búsqueda (ABB)

- La complejidad de las tres operaciones (búsqueda, inserción y borrado) es O(h) donde h es la altura del árbol. En el peor de los casos, se realizan h comparación, siendo ha la altura del árbol



TAD ABB

- La complejidad aumentará cuando h -> n (árbol degenerado). Complejidad O(n). El siguiente árbol es el resultado de insertar la secuencia: 1, 3, 8, 9, 12, 15