Universidad Carlos III de Madrid

Algorithms and Data Structures (ADS)
Bachelor in Informatics Engineering
Computer Science Department

Binary Search Trees.

Authors: Julian Moreno Schneider
Isabel Segura-Bedmar

April 2011
Departamento de Informtica,
Laboratorio de Bases de Datos Avanzadas (LaBDA)
http://labda.inf.uc3m.es/

Binary Search Trees (BST).

Definition.

Most of the information has been sourced from the books [1, 2].

A binary search tree is a binary tree T such that each internal node v of T stores an entry (k,x) such that:

- Keys stored at nodes in the left subtree of v are less than or equal to k.
- Keys stored at nodes in the right subtree of v are greater than or equal to k.

Keys provide a way of performing a search by making a comparison at each internal node v, which can stop at v or continue at v's left or right child. Binary trees are an excellent data structure for storing the entries of a dictionary, assuming we have an order relation defined on the keys. The main property is the possibility of storing the keys in an ordered way. An inorder traversal of the nodes of a binary search tree should visit the keys in nondecreasing order. Each node may store a key and a value. Key and value may be objects from different classes. For example, If a binary search tree is used to store a telephone directory, its nodes may consist of a String key (name) and a Long value (the phone number). Keys cannot be null. Binary search trees may store duplicate keys depending on our decision. Figure1 shows two examples of binary search trees with numeric keys and without values.

Implementation of a binary search tree.

Figure3 shows the implementation of a node for a binary search tree. You can observe that the node has an object Key of the class E (any class), an object value of the class F (any class), a reference to its parent node, and references to its left and right children, respectively. Figure 2 shows two examples of trees that are not binary search trees because their keys are not ordered.

Some of the possible operations of the Binary Search Tree ADT are:

- isEmpty(): tests if the tree is empty.
- clear(): sets the tree to null (that is, its root is null and size sets to 0).

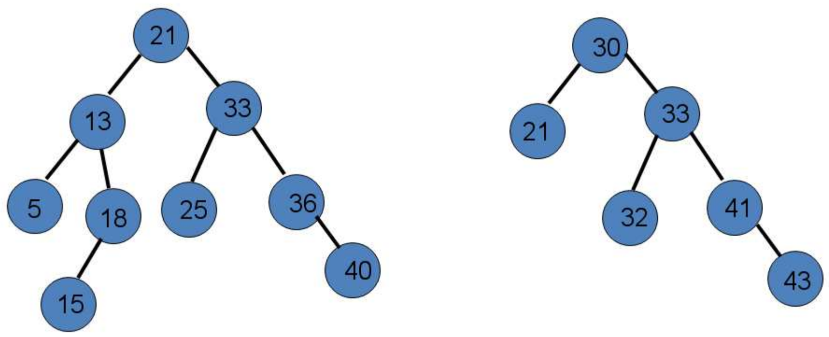


Figure 1: Example of binary search trees.

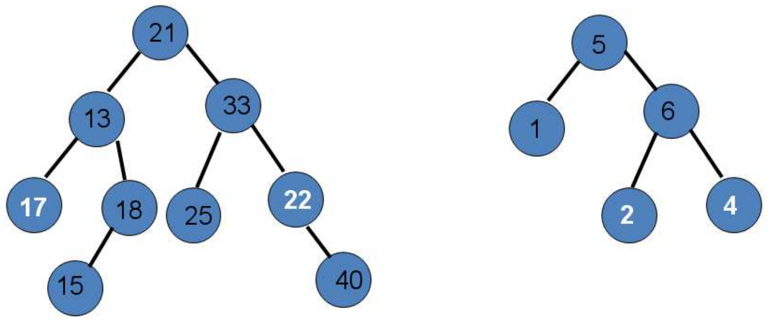
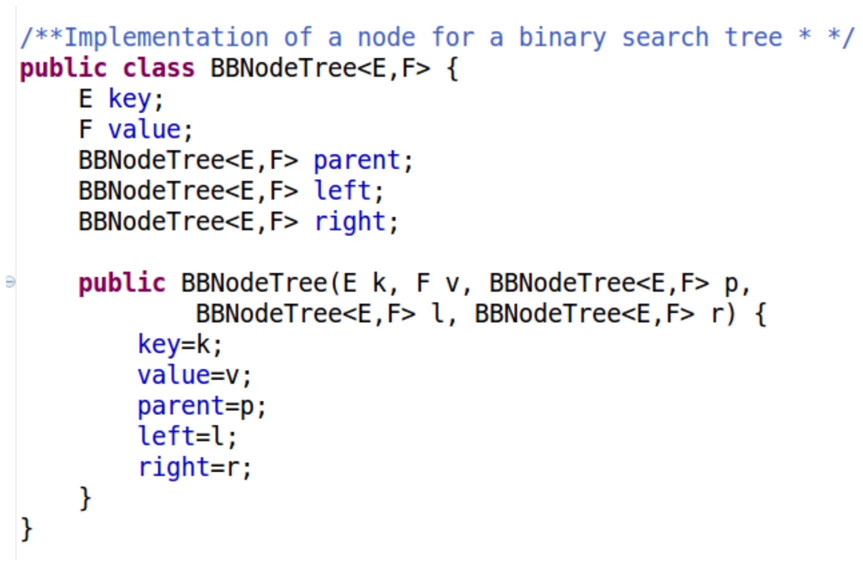


Figure 2: These trees are not binary search trees.

- firstKey(): returns the first key in the tree (that is, the lowest key). Figure 4 shows the implementation of this method.
- lastKey(): returns the last key in the tree (that is, the highest key).
- getValue(E k): returns the value stored in the node whose key is k; e.o.c it returns null.
- find(E k): this method returns a node with key k, if it exists. Returns null if k is not found. Due to the ordered keys of the tree, it is needed to test if the search key is less than, equal to, or greater than the key stored at node v (see Figure 5). If k is smaller then the search continues in the left subtree; if k is equal then the search terminates successfully; if k is greater then the search continues in the right subtree. If we reach an empty (null) node, then the search terminates unsuccessfully. Figure 6 shows the implementation of its operations.

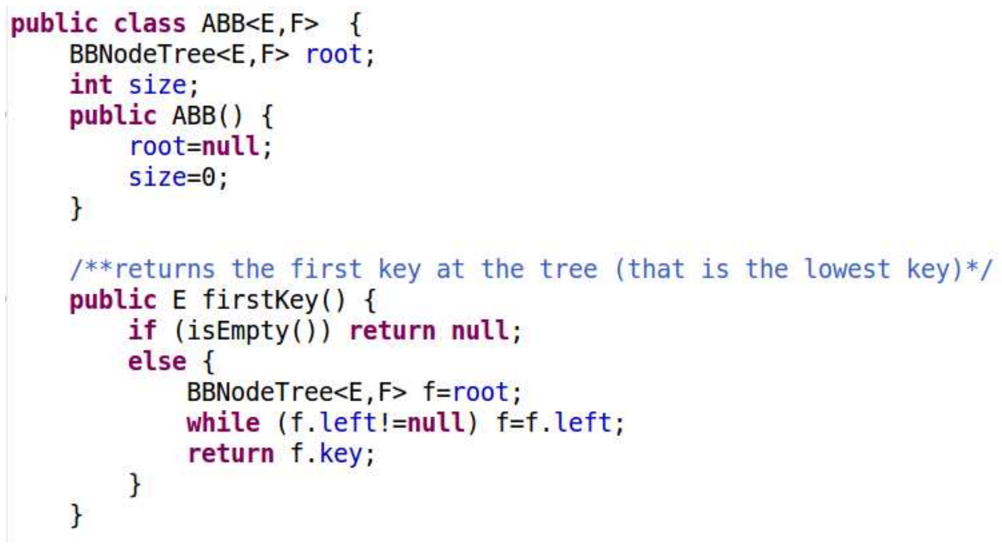


```java
/**Implementation of a node for a binary search tree * */
public class BBNodeTree<E,F> {
    E key;
    F value;
    BBNodeTree<E,F> parent;
    BBNodeTree<E,F> left;
    BBNodeTree<E,F> right;

    public BBNodeTree(E k, F v, BBNodeTree<E,F> p,
            BBNodeTree<E,F> l, BBNodeTree<E,F> r) {
        key=k;
        value=v;
        parent=p;
        left=l;
        right=r;
    }
}
```

Figure 3: Implementation of a node of a binary search tree.

- findAll(k): this method returns a list of all nodes with keys equal to k.



```java
public class ABB<E,F>  {
    BBNodeTree<E,F> root;
    int size;
    public ABB() {
        root=null;
        size=0;
    }

    /**returns the first key at the tree (that is the lowest key)*/
    public E firstKey() {
        if (isEmpty()) return null;
        else {
            BBNodeTree<E,F> f=root;
            while (f.left!=null) f=f.left;
            return f.key;
        }
    }
```

Figure 4: This implementation returns the lowest key in a tree.

Figure 7 shows the recursive implementation of the method find (called searchNode). Worst-case running time of searching in a BST is simple. The algorithm searchNode is recursive and executes a constant number of primitive

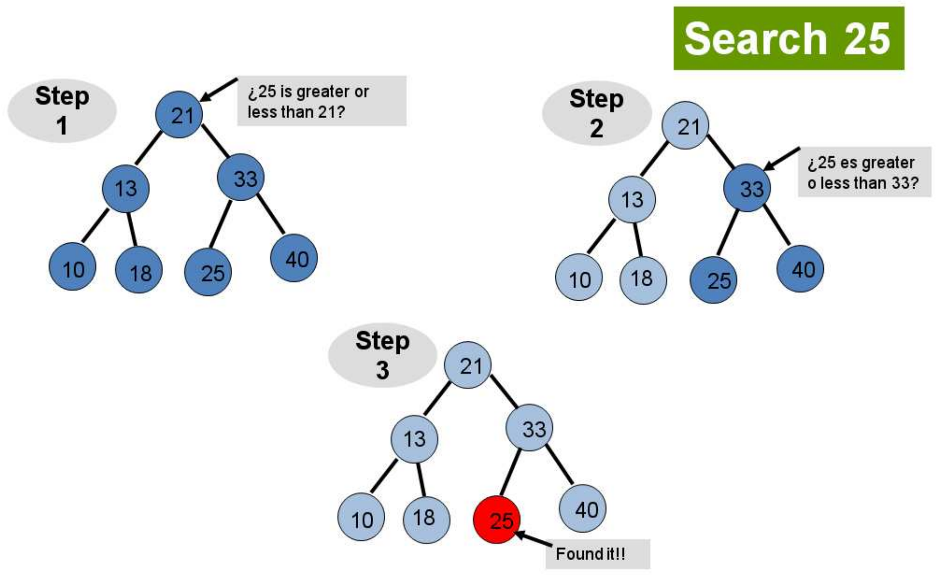
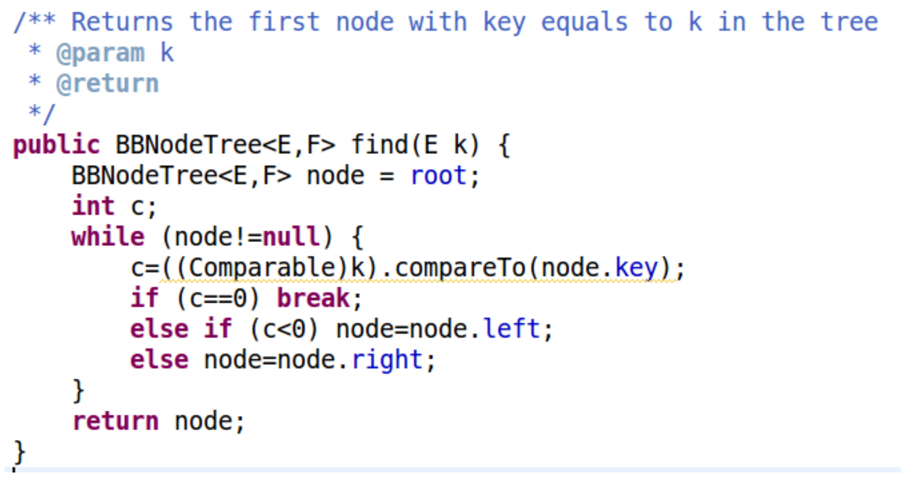

Search 25
Step 1
¿25 is greater or less than 21?
21
13
33
10
18
25
40
Step 2
¿25 es greater o less than 33?
Step 3
Found it!!

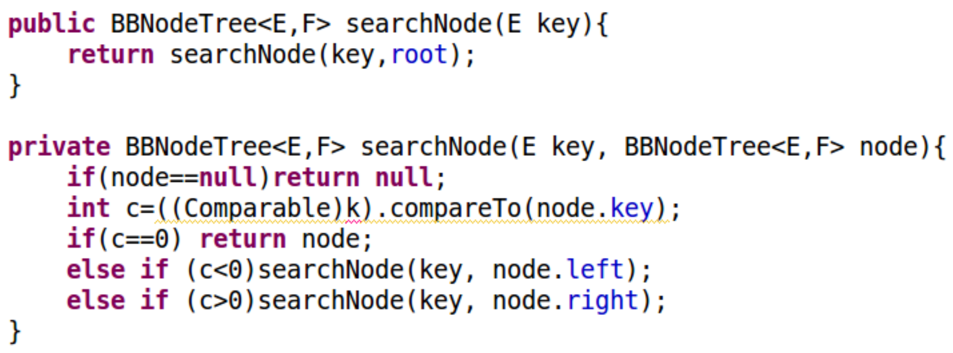
Figure 5: Searching a node whose key is 25.



```
/** Returns the first node with key equals to k in the tree
 * @param k
 * @return
 */
public BBNodeTree<E,F> find(E k) {
    BBNodeTree<E,F> node = root;
    int c;
    while (node!=null) {
        c=((Comparable)k).compareTo(node.key);
        if (c==0) break;
        else if (c<0) node=node.left;
        else node=node.right;
    }
    return node;
}
```

Figure 6: The implementation of the iterative find.

operations for each recursive call. Each recursive call of searchNode is made on a child of the previous node. That is, searchNode is called on the nodes of a path of T that starts at the root and goes down one level at a time. Thus, the number of such nodes is bounded by h + 1, where h is the height of T. For example, we would spend O(1) time per node encountered in the search, method find on tree T runs in O(h) time, where h is the height of the binary search tree



```java
public BBNodeTree<E,F> searchNode(E key){
    return searchNode(key,root);
}

private BBNodeTree<E,F> searchNode(E key, BBNodeTree<E,F> node){
    if(node==null)return null;
    int c=((Comparable)k).compareTo(node.key);
    if(c==0) return node;
    else if (c<0)searchNode(key, node.left);
    else if (c>0)searchNode(key, node.right);
}
```

Figure 7: The recursive implementation of the method find.

T. Admittedly, the height h of T can be as large as n, but we expect that it is usually much smaller. findAll(k) has O(h+s) where s is the number of returned nodes. Indeed, we will show how to maintain an upper bound of O(logn) on the height of a search tree T using AVL. There are additional methods for searching through predecessors and successors of a key or entry, but their performance is similar to that of find.

Binary search trees allow implementations of the insert and remove operations using algorithms that are fairly straightforward, but not trivial. We now explain some operations more difficult than the previous ones.

- insert(k,x): inserts a node with key k and value x (see Figure 8). If there is some node with this key, then its value is replaced by x. To insert a new node in the tree, we have to find the proper place to set the node. For this purpose, we will traverse the tree by means of a recursive method shown in Figure 9.

- remove(k): Removes a node with key equals to e, and return it.

- removeAll(k); removes all nodes whose keys are equals to e.

The implementation of the remove(k) operation is a bit more complex, since we do not wish to create any ”holes” in the tree. We begin our implementation of operation removeNode(k) using a recursive method that traverses the tree T searching a node w that has the key k. If there is no node with key k in the tree T, we return null (and we are done). If there is a node w, then we wish to remove it and we distinguish three cases (of increasing difficulty):

- If the node w is a leaf, then we should find its parent and free the node w (see Figure 10). Its implementation is shown in Figure 11.

- If one of the children of node w is null, we simply remove w and restructure T by replacing w with its non-null child. (See Figure 12). Its implementation is shown in Figure 13.

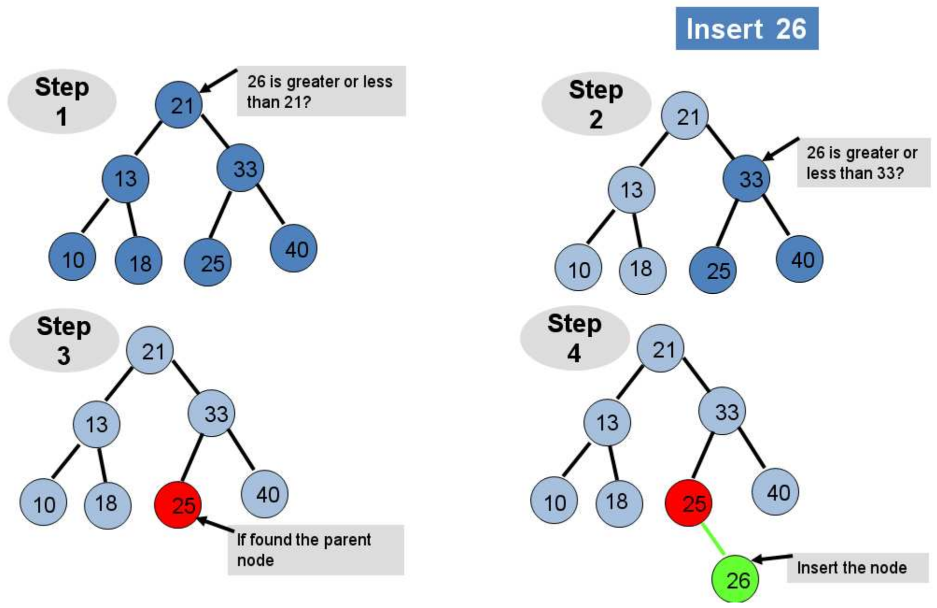


Figure 8: Inserting a node whose key is 26.

- If both children of node w are internal nodes (non-null), we cannot simply remove the node w from T, since this would create a "hole" in T. Instead, we proceed as follows (see Figure 14; its implementation is shown in Figure 15.):

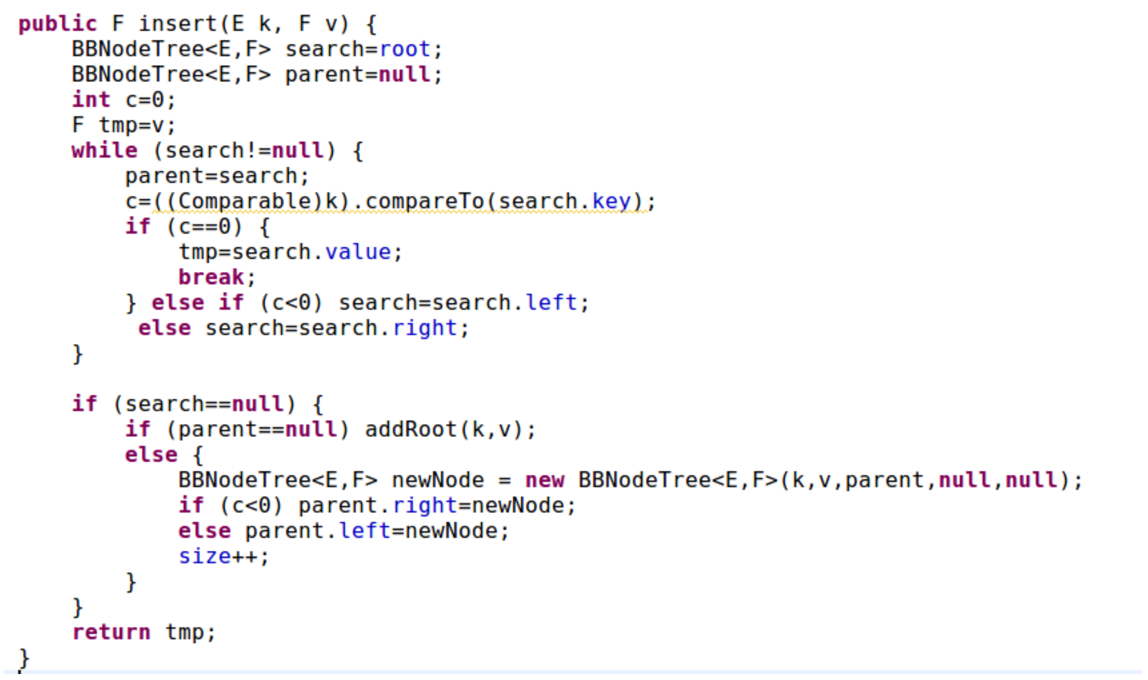
  – We find the first node y that follows w in an inorder traversal of T. Node y is the left-most internal node in the right subtree of w, and is found by going first to the right child of w and then down T from there, following left children.

  – We save the entry stored at w in a temporary variable t, and move the entry of y into w. This action has the effect of removing the former entry stored at w.

  – We remove node y from T. This action replaces y with its right child.

  – We return the entry previously stored at w, which we had saved in the temporary variable t.

As with searching and insertion, this removal algorithm traverses a path from the root to an external node, possibly moving an entry between two nodes of this path.



```
public F insert(E k, F v) {
    BBNodeTree<E,F> search=root;
    BBNodeTree<E,F> parent=null;
    int c=0;
    F tmp=v;
    while (search!=null) {
        parent=search;
        c=((Comparable)k).compareTo(search.key);
        if (c==0) {
            tmp=search.value;
            break;
        } else if (c<0) search=search.left;
         else search=search.right;
    }

    if (search==null) {
        if (parent==null) addRoot(k,v);
        else {
            BBNodeTree<E,F> newNode = new BBNodeTree<E,F>(k,v,parent,null,null);
            if (c<0) parent.right=newNode;
            else parent.left=newNode;
            size++;
        }
    }
    return tmp;
}
```

Figure 9: Implementation of the method insert.

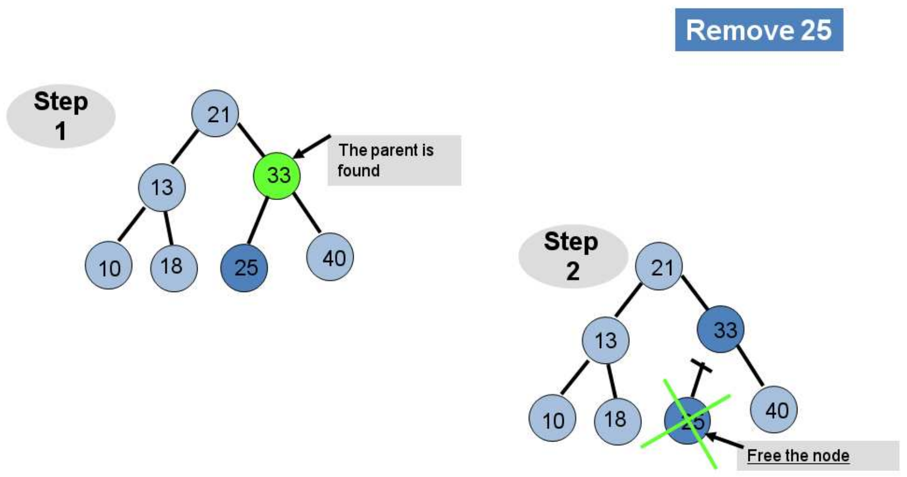
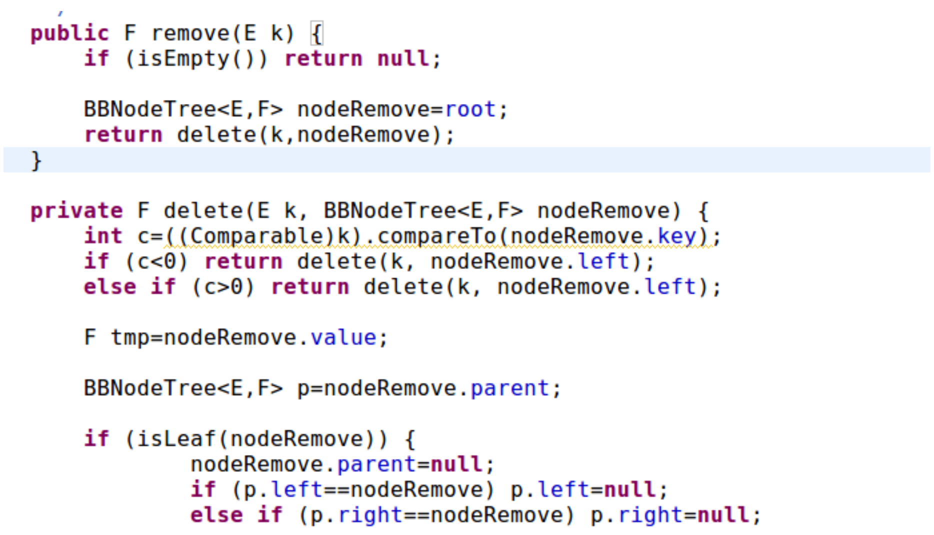


Figure 10: Removing a leaf.

Performance of a Binary Search Tree

The analysis of the search, insertion, and removal algorithms are similar. We spend O(1) time at each node visited, and, in the worst case, the number of



```
public F remove(E k) {
    if (isEmpty()) return null;

    BBNodeTree<E,F> nodeRemove=root;
    return delete(k,nodeRemove);
}

private F delete(E k, BBNodeTree<E,F> nodeRemove) {
    int c=((Comparable)k).compareTo(nodeRemove.key);
    if (c<0) return delete(k, nodeRemove.left);
    else if (c>0) return delete(k, nodeRemove.left);

    F tmp=nodeRemove.value;

    BBNodeTree<E,F> p=nodeRemove.parent;

    if (isLeaf(nodeRemove)) {
            nodeRemove.parent=null;
            if (p.left==nodeRemove) p.left=null;
            else if (p.right==nodeRemove) p.right=null;
```

Figure 11: Part of the implementation of the method remove (case: removing a leaf node).

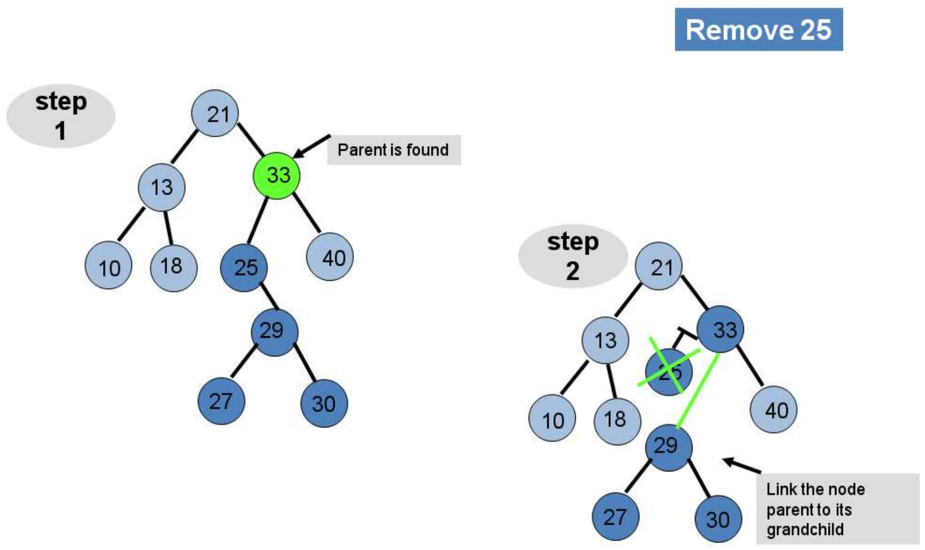
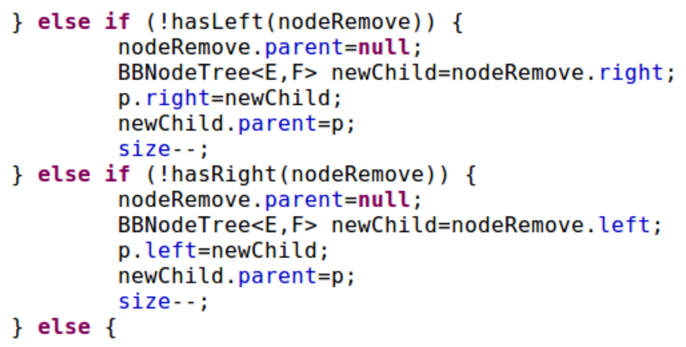


Figure 12: Removing a node with an only node.

nodes visited is proportional to the height h of T. Thus, a binary search tree T is an efficient implementation of a dictionary with n entries only if the height of T is small. In the best case, T has height h = log(n + 1), which yields



```
} else if (!hasLeft(nodeRemove)) {
        nodeRemove.parent=null;
        BBNodeTree<E,F> newChild=nodeRemove.right;
        p.right=newChild;
        newChild.parent=p;
        size--;
} else if (!hasRight(nodeRemove)) {
        nodeRemove.parent=null;
        BBNodeTree<E,F> newChild=nodeRemove.left;
        p.left=newChild;
        newChild.parent=p;
        size--;
} else {
```

Figure 13: Part of the implementation of the method remove (case: removing a node with an only child).

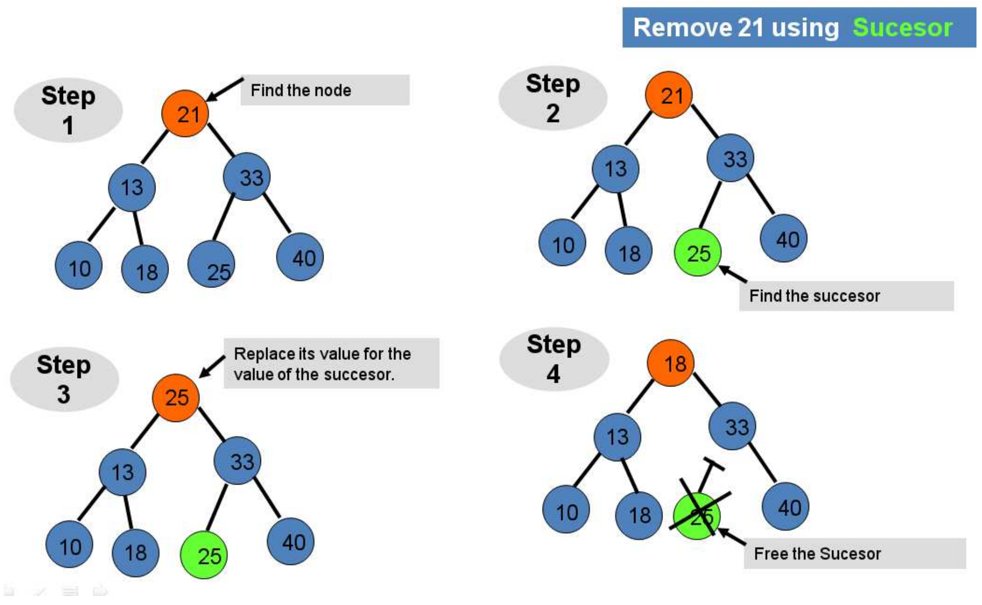
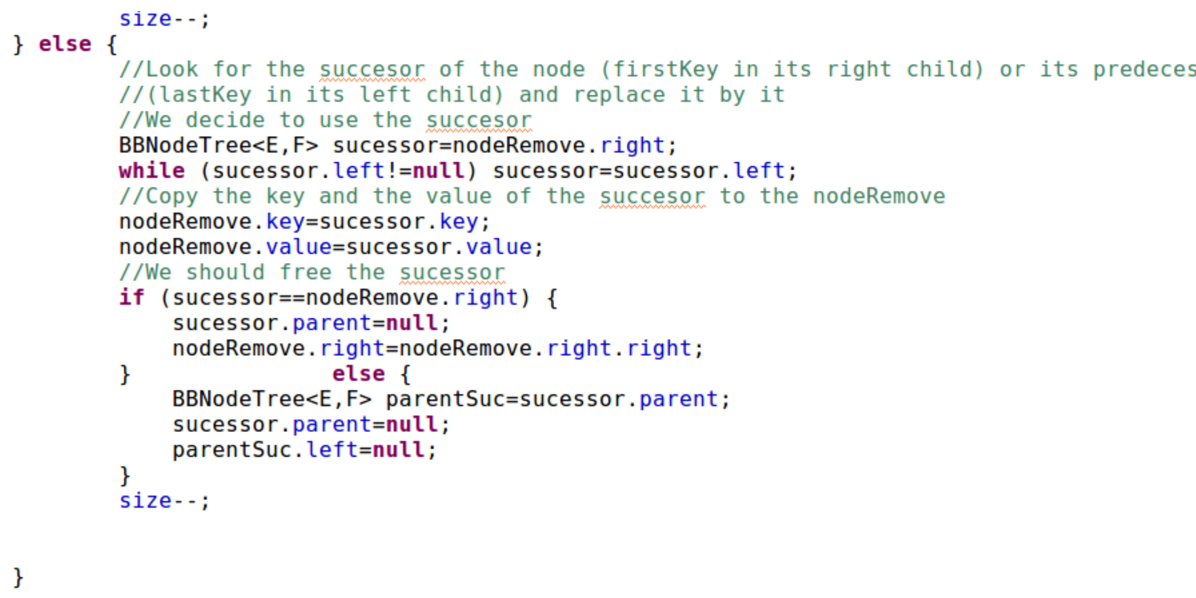


Figure 14: Removing a node with two children.

logarithmic-time performance for all the update operations. In the worst case, however, T has height n, in which case it would look and feel like an ordered list implementation. Such a worst-case configuration arises, for example, if we insert a series of entries with keys in increasing or decreasing order. This case is shown in Figure 16.

The performance of a binary search tree is summarized in Table 1. Should be taken into account that search and update operations varies dramatically depending on the tree's height. We can assume that, on average, a binary



```
            size--;
    } else {
            //Look for the succesor of the node (firstKey in its right child) or its predeces
            //(lastKey in its left child) and replace it by it
            //We decide to use the succesor
            BBNodeTree<E,F> sucessor=nodeRemove.right;
            while (sucessor.left!=null) sucessor=sucessor.left;
            //Copy the key and the value of the succesor to the nodeRemove
            nodeRemove.key=sucessor.key;
            nodeRemove.value=sucessor.value;
            //We should free the sucessor
            if (sucessor==nodeRemove.right) {
                sucessor.parent=null;
                nodeRemove.right=nodeRemove.right.right;
            }               else {
                BBNodeTree<E,F> parentSuc=sucessor.parent;
                sucessor.parent=null;
                parentSuc.left=null;
            }
            size--;


    }
```

Figure 15: Part of the implementation of the method remove (case: removing a node with two children).

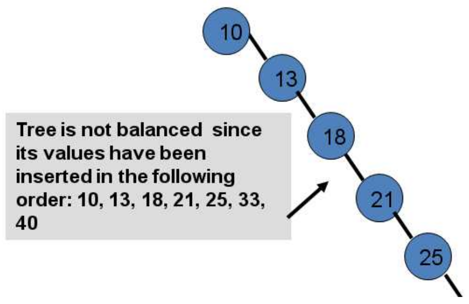

10
13
18
21
25
Tree is not balanced since its values have been inserted in the following order: 10, 13, 18, 21, 25, 33, 40

Figure 16: Part of the implementation of the method remove (case: removing a node with two children).

search tree with n keys has expected height O(log(n)).

Methods	**Time**
isEmtpy(), size()	O(1)
find, insert, remove	O(h)
findAll()	O(h+s)

Table 1: Performance of a binary search tree

Bibliography

[1] M. Goodrich and R. Tamassia, *Data structures and algorithms in Java*. Wiley-India, 2009.

[2] M. Weiss, *Data structures and problem solving using Java.* Addison Wesley Publishing Company, 2002.