1 AVL Trees

If we study the cost of performing the various operations on the binary search trees, we can conclude the best behaviour can be achieved when the tree has a number of level which is proportional to $\log_2 n$. In this case, searching a key will take at most $\log_2 n$ steps, as well as inserting it.

The problem occurs when the depth of the tree is not $\log_2 n$. E.g., in the worst case we can have that all the nodes of the tree are on the same branch (of length $n$). Thus it would be nice to be able to maintain during insertion a balanced tree. A binary tree is said to be balanced if, for every node $K$ in the tree, the height of its two subtrees differs at most one. For example, in Figure 1(i) the tree is balanced (e.g., the two subtrees of node $a$ have height 2 and 1); the tree in Figure 1(iii) instead is not balanced: the left subtree of node $c$ has height 3 while the right subtree has height 1.

AVL trees are a a special kind of search trees in which there is guarantee that after each insertion the tree is balanced (and, additionally the whole resulting tree has minimum height). This means that the heights of the two subtrees of each node are either equal or they differ by one.

More formally, an AVL tree is a binary search tree which is either empty or it satisfies the following conditions:

- the left subtree is an AVL tree
- the right subtree is an AVL tree
- the difference between the heights of the left and right subtrees is at most one;

For example, the first two trees in figure 2 are AVL trees, while the third is not.

AVL trees are interesting because they are guaranteed to have a height that is in the order of $\log_2 n$ for a tree containing $n$ nodes.
In order to support the management of AVL trees we will make use of some additional concepts. Given an AVL tree, if the left subtree is higher than the right subtree, then we say that the tree is left high. If the right subtree is higher than the left subtree, then the tree is right high.

Given a tree $t$, the balance factor of $t$ is the difference between the height of the right subtree and the height of the left subtree.

### 1.1 Representation of Nodes in an AVL Trees

Let us represent in Java the nodes of AVL trees similarly as ordinary binary trees. Each node in the tree is going to be represented by a class:

```java
public class Node {
    public int key; // the information
    public BSTree left; // left child
    public BSTree right; // right child
    public int bfactor; // balance factor
}
```

For the sake of simplicity, this time we make the data members public to avoid having to write all the access and modification methods (this is just to keep things simple in this discussion).

The field `bfactor` can contain one of three possible values: 0 or 1 or −1. This is used to keep track of how the tree rooted in such node is balanced. We will use the following convention:

- `bal` = −1 implies that the left subtree of the node has height one greater than the right subtree;
- `bal` = 0 implies that the left and right subtree have the same height;
- `bal` = 1 implies that the right subtree has height one greater than the left subtree.

### 1.2 AVL Tree Operations

The search operation in an AVL tree is performed exactly in the same way as in the case of generic binary search trees, since they are organized according to the same criteria.

#### 1.2.1 Examples

To insert a new element in an AVL tree we first need to find the correct place where the new node should be inserted, just as we did in the case of binary search trees. The main problem is that, after the new node has been inserted, the tree might have lost its AVL property—one subtree might have become too high due to the insertion of the new node. Thus, in order to maintain the AVL structure, further modifications are required. This
is accomplished by following back the branch in which the insertion has been performed towards the root and modifying the subtrees encountered to ensure that they all meet the AVL property.

Let us illustrate this by examples: the tree in figure 3 shows an AVL tree; the numbers next to each node indicate the balance factor for that node.

Let us insert 90 in the tree. The traditional binary search tree insertion will lead to the tree in figure 4. The figure also shows the point where the balance factor indicates a violation of the AVL property.

Since the subtree rooted at 70 is the first one that shows an unbalanced situation, we will proceed in transforming it into a better balanced subtree. The process can be obtained by shifting some of the “weight” from the right to the left, as shown in figure 5.

Now let us consider again the tree of figure 3 and instead let us insert the node 75. This leads to the tree of figure 6.
In this example, the transformation to adjust the tree rooted in 70 is a bit different, as indicated in figure 7.

Note that the first step of the transformation brings us back to the same configuration as the tree of figure 4. Let us now consider a more complex example. Consider the tree of figure 8.

When we insert the node 95 we obtain the tree shown in figure 9.

The transformation is very similar to the one applied to the case of figure 4, just applied to a bigger subtree. This is shown in figure 10.

1.2.2 AVL Rotations

Let us now try to abstract what kind of transformations are required to adjust the AVL subtrees that have lost their proper balance. The transformations we have used in the examples fall in two main categories: left rotations and right rotations. Combinations of left and right rotations are sufficient to restore an AVL tree after an insertion.

Suppose the rotation occurs at a node \( x \). If it is a left rotation, then certain nodes from the right subtree of \( x \) move to its left subtree; the root of the right subtree of \( x \) becomes the new root of the reconstructed subtree. Similarly, if it is a right rotation then nodes move from the left subtree of \( x \) to the right subtree, and the root of the left subtree becomes the root of the modified subtree.
Figure 8: Tree before inserting 95

Figure 9: Tree after inserting 95

Figure 11 describes abstractly how a right rotation (at node $b$) is realized. $T_1$, $T_2$, and $T_3$ are of equal heights (say $h$). The dotted rectangle shows an item insertion in $T_1$ causing the height of the subtree $T_1$ to increase by 1. The subtree at node $a$ is still an AVL tree, but the balance criteria is violated at the root node. We note the following in this tree; because the tree is a binary search tree:

- every key in $T_1$ is smaller than $a$
- every key in $T_2$ is larger than $a$
- every key in $T_2$ is smaller than $b$

Therefore:

- we make $T_2$ the left subtree of $b$
- we make $b$ the right child of $a$
- $a$ becomes the root of the modified tree

The symmetrical behavior is present in the case of a left rotation, and it is illustrated in figure 12. The two rotations described allows to directly resolve the two general unbalanced cases in which

- the insertion has turned the left subtree of a left high AVL tree into a left high tree (the case of figure 11), or
• the insertion has turned the right subtree of a right high AVL tree into a right high tree (the case of figure 12).

There are two more cases that may occur that are not covered above:

• the case in which the left subtree of a left high tree has become right high
• the case in which the right subtree of a right high tree has become left high

These two cases require a more complex transformation pattern, described next. The situation of a left tree of a left high tree becoming right high can be handled using two successive rotations, as shown in figure 13.

We can note the following:

• all keys in $T_3$ are smaller than $c$
• all keys in $T_3$ are larger than $b$
• all keys in $T_2$ are smaller than $b$
• all keys in $T_2$ are larger than $a$
• after the insertion the subtrees with root $a$ and $b$ are still AVL trees
• the balance criteria is violated at the root node $c$
• the balance factor of node $c$ prior to the insertion is -1 and the factor for $a$ is 1.
The dotted boxes in the figure show the insertions that might have created the unbalanced situation.

This is a case where a double rotation is required to solve the problem. One rotation is applied on node $a$ and another on node $c$. If the balance factor of the node where the tree is to be reconstructed and the balance factor of the higher subtree are opposite (-1 and 1) then we know that a double rotation is required. First we make a left rotation on $a$, next, because the tree at node $c$ is left high, we make a right rotation at $c$.

The symmetrical situation is illustrated in figure 14.

Using these four cases we now describe what type of rotation might be required at each node.

Suppose that the tree is to be reconstructed by rotation at node $x$. The subtree with the root node $x$ then requires either a single or a double rotation.

1. Suppose that the balance factor of node $x$ and the balance factor of the root node of the higher subtree of $x$ have the same sign, that is, both positive or both negative.

   (a) If these balance factors are positive, make a single left rotation at $x$. Prior to insertion the right subtree of $x$ was higher than its left subtree. The new item was inserted in the right subtree of $x$, causing the height of the right subtree to increase which in turn violated the balance criteria at $x$.

   (b) If these balance factors are negative, make a single right rotation at node $x$. Prior to insertion, the left subtree of $x$ was higher than its right subtree. The new item was inserted in the left subtree of $x$, causing its height to grow.

2. Support that the balance factor of node $x$ and the balance factor of the higher subtree of $x$ are opposite in sign. To be specific, suppose that the balance factor of node $x$ prior to insertion was $-1$ and suppose that $y$
is the root node of the left subtree of $x$. After insertion, the balance factor of $y$ is 1. That is, after insertion, the right subtree of $y$ grew in height. In this case, we require a double rotation at $x$. First we make a left rotation at $y$ (because $y$ is right high). Then we make a right rotation at $x$. The other case, which is a mirror image of this case, is handled similarly.

### 1.3 Sketch of Java Implementation

The following Java methods implement the left and right rotation at a node.

```java
private void rotateToRight()
{
    Node r; // root of right subtree
    if (root == null)
        System.out.println(''Error in tree'');
    else
    {
        if (root.right.emptyTree())
            System.out.println(''No right subtree to rotate'');
        else
        {
            r = getRight().getRootNode();
```
Figure 14: Left-right rotation at c

```java
root.right = r.left;
r.left = new BSTree(root);
root = r;
}
}

private void rotateToRight()
{
    Node r; // root of left subtree
    if (root == null)
        System.out.println("Error in tree");
    else
    {
        if (getLeft().emptyTree())
            System.out.println("No left subtree to rotate");
        else
        {
            r = getLeft().getRootNode();
            root.left = r.right;
            r.right = new BSTree(root);
            root = r;
        }
    }
}
```
We assume the presence of a method which returns the root of a tree:

```java
public Node getRootNode()
{
    return root;
}
```

Now that we know how to implement both rotations, we next write the Java methods balanceFromLeft and balanceFromRight, which are used to reconstruct the tree at a particular node. These methods use the rotateToLeft and rotateToRight methods and also adjust the balance factors of the nodes affected by the reconstruction. The method balanceFromLeft is called when the subtree is left double high and certain nodes need to be moved to the right subtree. The methods balanceFromRight has similar conventions.

```java
private void balanceFromLeft()
{
    Node p;
    Node w;

    p = root.left.getRootNode();
    switch (p.bfactor)
    {
        case -1: root.bfactor = 0;
                 p.bfactor = 0;
                 rotateToRight();
                 break;
        case 0: System.out.println('CANNOT BALANCE');
                 break;
        case 1: w = p.right.getRootNode();
                 switch (w.bfactor)
                 {
                     case -1: root.bfactor = 1;
                              p.bfactor = 0;
                              break;
                     case 0: root.bfactor = 0;
                              p.bfactor = 0;
                              break;
                     case 1: root.bfactor = 0;
                              p.bfactor = -1;
                              break;
                 }
                 w.bfactor = 0;
                 root.left.rotateToLeft();
                 rotateToRight();
    }
}
```

We now focus on the method that performs the insertion in the AVL tree. The method insertIntoAVL inserts a new item in an AVL tree. The item to be inserted is passed as a parameter. The following steps describe the method:

```java

```
• create a new node for the item to be inserted
• search the tree and find the right place for insertion
• insert the new node
• backtrack the path which was constructed to find the place for the new node in the tree, towards the root. If necessary adjust the balance factors of the nodes or reconstruct the tree at a node on the path.

```java
private boolean insertIntoAVL(int key) {
    if (root == null) {
        root = new Node(key);
        return true;
    } else {
        if (root.key > key) {
            if (root.left == null) {
                root.left = new BSTree(new Node(key));
                if (root.right == null)
                    return true;
                else
                    return false;
            } else {
                if (root.left.insertIntoAVL(key))
                    switch (root.bfactor) {
                        case -1: balanceFromLeft();
                            return false;
                        case 0: root.bfactor = -1;
                            return true;
                        case 1: root.bfactor = 0;
                            return false;
                    }
            }
        } else {
            if (root.right == null) {
                root.right = new BSTree(new Node(key));
                if (root.left == null)
                    return true;
                else
                    return false;
            } else {
                if (root.right.insertIntoAVL(key))
                    switch (root.bfactor) {
                        case -1: balanceFromLeft();
                            return false;
                        case 0: root.bfactor = -1;
                            return true;
                        case 1: root.bfactor = 0;
                            return false;
                    }
            }
        }
    }
}
```
if (root.right.insertIntoAVL(key))
    switch (root.bfactor)
    {
        case -1: root.bfactor = 0;
                return false;
        case 0: root.bfactor = 1;
                return true;
        case 1: balanceFromRight();
                return false;
    }
}