2022

INTRODUCTION TO AVL TREES

ROBERTO MAMELI

Introduction to AVL Trees

Index

INTRODUCTION **2**

OVERVIEW ON TREES, BINARY SEARCH TREES AND AVL TREES **3**

General Definitions 3

Binary Trees and Binary Search Trees (BST) 5

BST Operations 9

BST Lookup *9*

BST Insertion *11*

BST Removal *13*

BST Traversal *17*

AVL Trees 21

Rebalancing *22*

CONCLUSIONS **27**

INTRODUCTION

This tutorial provides a comprehensive introduction to Binary Search Trees and AVL Trees, which are the basis of *libfmrt* library (*Fast Memory Resident Table*, see https://www.roberto-mameli.it/software/ or https://github.com/Roberto-Mameli/libfmrt.git).

OVERVIEW ON TREES, BINARY SEARCH TREES AND AVL TREES

GENERAL DEFINITIONS

In computer science, a **Tree** data structure is a collection of data (represented by Nodes) which is organized in hierarchical structure recursively. Mathematically, a tree is an acyclic connected graph[1] where each node has zero or more children nodes and at most one parent node; in the above definition, connected means that given an arbitrary pair of nodes, there is at least a path between them, while acyclic implies that this path is always unique (if two paths were available, their union would provide a cycle). Summarizing, in a Tree any arbitrary pair of nodes is always connected by one and just one path.

In the Tree data structure, every individual element is called **Node**. Each Node in a tree data structure stores the actual data of that specific element and link to next element in hierarchical structure. Nodes are interconnected through **Edges**. The hierarchical organization of the Tree Data Structure implies that for each pair of Nodes interconnected by an Edge we can introduce a Parent-Child relationship. The **Parent** Node is the predecessor of the **Child**, and vice versa the **Child** is the successor of the **Parent**. Each node can have just one **Parent**, but it may have several **Children**.

Every tree has indeed one and just one special Node, called **Root**, which has no **Parent** and can be considered the origin of the Tree data structure. A Tree also contains several Nodes without any **Child**, which are called **Leaves**.

The conventional graphical representation of a Tree is obtained by indicating Nodes through circles and Edges through links. The **Root** is always represented at the top, while **Leaves** are designed at the bottom of the structure. The following picture depicts a generic Tree:

[1] In mathematics, and more specifically in graph theory, a graph is a structure amounting to a set of objects in which some pairs of the objects are in some sense "related". The objects correspond to mathematical abstractions called vertices and each of the related pairs of vertices is called an edge. Typically, a graph is depicted in diagrammatic form as a set of dots or circles for the vertices, joined by lines or curves for the edges.

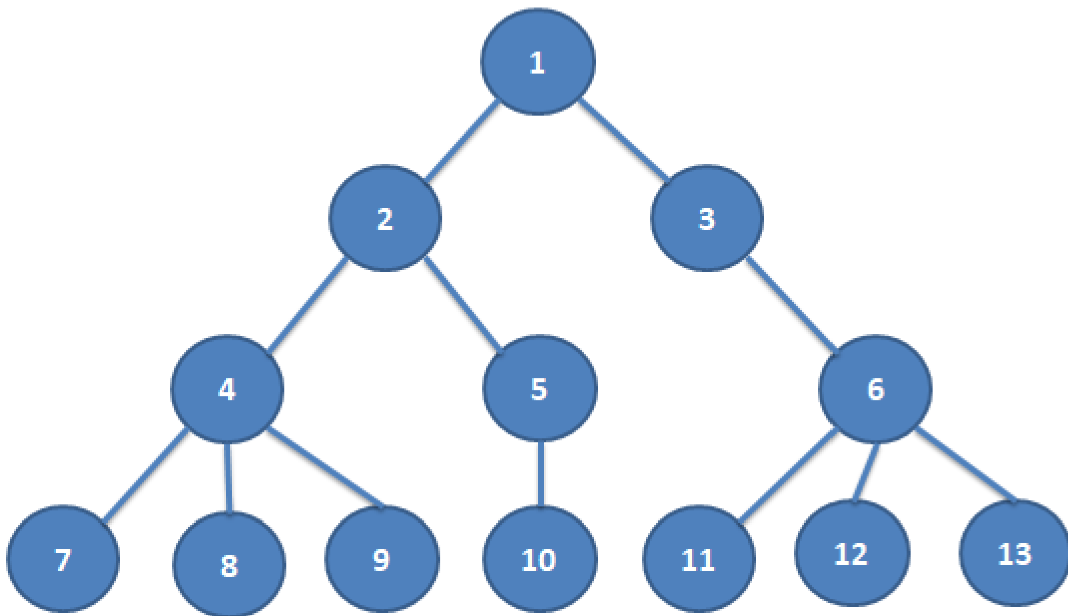

Figure 1 - Example of a generic Tree

Looking at the Tree depicted in Figure 1, we recognize ***1*** as the Root node and nodes ***7***, ***8***, ***9***, ***10***, ***11***, ***12*** and ***13*** as Leaves. Node ***1*** (i.e. the Root) doesn't have any parent node (by definition), but has two children, i.e. ***2*** and ***3***. Node ***3*** has only one child (i.e. node ***6***), which in turn has ***3*** children, i.e. ***11***, ***12*** and ***13***.

By looking at Figure 1, we can give some further definitions. Specifically, we define:

- ***Siblings***: nodes which belong to same Parent (e.g. nodes ***4*** and ***5*** are siblings, and similarly nodes ***7***, ***8*** and ***9*** are siblings as well)

- ***Degree of a node***: the total number of children of that node (e.g. node ***2*** has degree ***2***, node ***3*** degree ***1*** and node ***6*** degree ***3***)

- ***Degree of the tree***: the highest degree among all the nodes. The tree depicted in Figure 1 has degree 3, since the maximum number of children among all nodes is 3.

- ***Level***: the Level can be defined recursively as follows. The Root node has always Level 0. Each node in the tree has a level given by the value of its parent plus 1. In Figure 1, node ***1*** which is the Root, has Level 0, its children (***2*** and ***3***) have Level 1, children of Nodes at Level 1 have Level 2 (i.e. nodes ***4***, ***5*** and ***6***), and so on down to Leafs.

- ***Depth of a Node***: number of edges from the node to the tree's Root node. The Root has always Depth 0, its children have Depth 1, and so on.

- ***Height of a Node***: number of edges on the longest downward path between that node and a leaf. An equivalent (recursive) definition is the following. The Height of a node is given by 1+the maximum Height among all children, where Leaves have Height 0 by definition.

- ***Height of the Tree***: is the Height of the Root Node

- ***Subtree***: Every child of a Node identifies a Subtree, which is the tree whose root is represented by the specified Child. Figure 2 below outlines in different colours subtrees of the Root node of the tree represented in Figure 1.

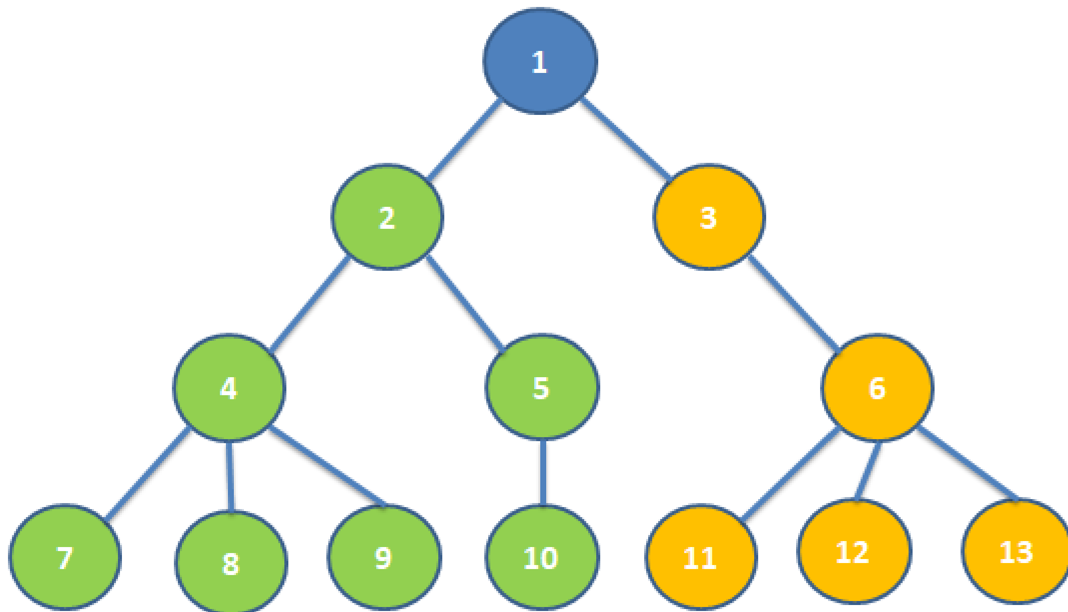

Figure 2 - Subtrees of the Root Node

Binary Trees and Binary Search Trees (BST)

A ***Binary Tree*** is a Tree with Degree 2, i.e. in which every node can have a maximum of two children.

A ***Binary Search Tree*** (***BST***) is a Binary Tree characterized by the following properties:
- The key data associated to nodes belong to a set where an order relationship is defined: given arbitrarily two keys, it is always possible to compare them, i.e. to define an order between them;
- Given any node of the tree, all keys in the left subtree are strictly lower than the key in the node;
- Given any node of the tree, all keys in the right subtree are strictly greater than the key in the node;

The following figure shows an example of Binary Search Tree:

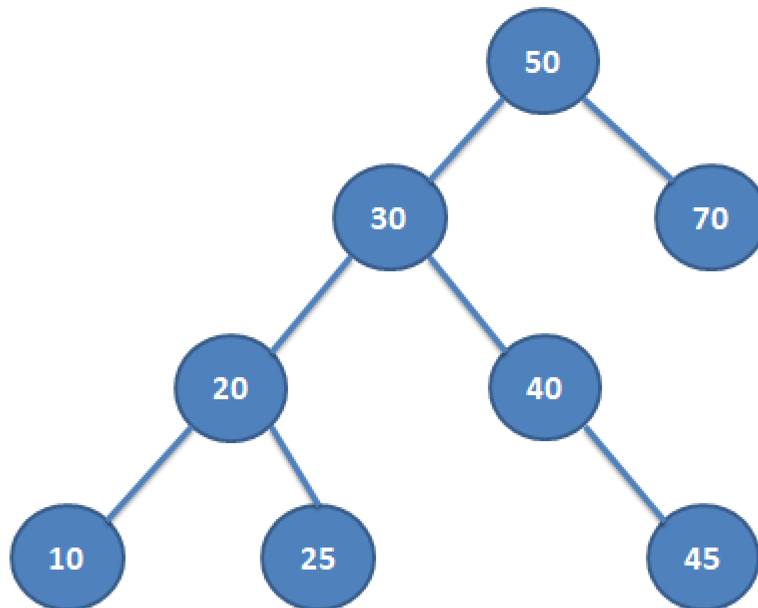

Figure 3 - Example of Binary Search Tree

The Tree depicted in Figure 3 is a Binary Search Tree since all previous properties are satisfied. In fact:

- Values stored in the nodes are non-negative integers, which is an ordered set;
- Given any arbitrary node, it is easy to verify that all nodes in the left subtree are lower and all nodes in the right subtree are greater than the value stored in the node itself

When dealing with binary search trees, some further definitions are introduced. Specifically, we define:

- ***Balance Factor of a Node***: is defined as the difference between the Heights of the right subtree and left subtree

- ***Balanced BST***: a Binary Search Tree is said Balanced if all its nodes are characterized by a balance factor in the interval *{-1,0,1}*

- ***Unbalanced BST***: a Binary Search tree which contains at least a node with Balance Factor outside the interval *{-1,0,1}*

Figure 4 below represents the same Binary Search Tree already shown in Figure 3, where we report the Height (***H***) and the Balance Factor (***BF***) for each node:

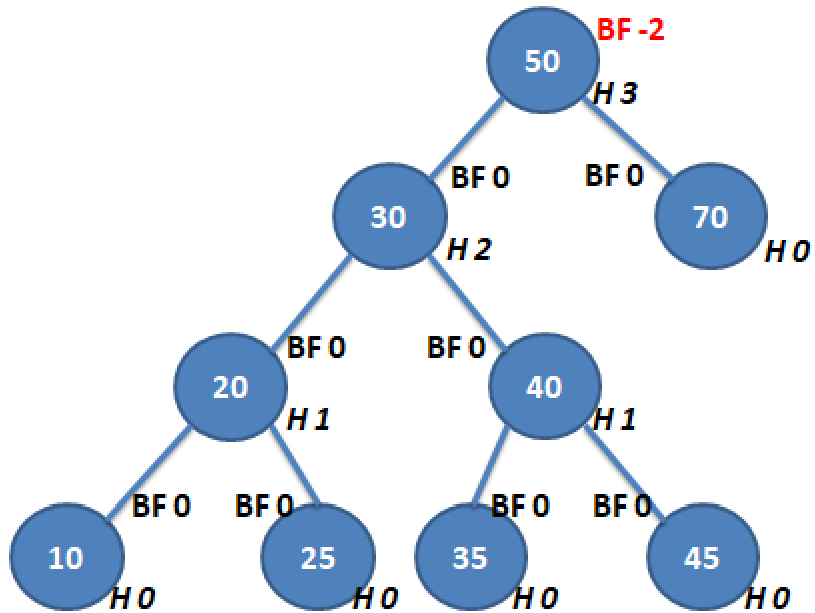

Figure 4 - Example of Unbalanced Binary Search Tree

This is an example of unbalanced BST, since there is at least a node (in this case the Root), with Balance Factor whose absolute value is greater than *1* (outlined in red in the figure). Figure 5 below represents the worst-case scenario of unbalanced Binary Search Tree, which resembles an ordered linked list:

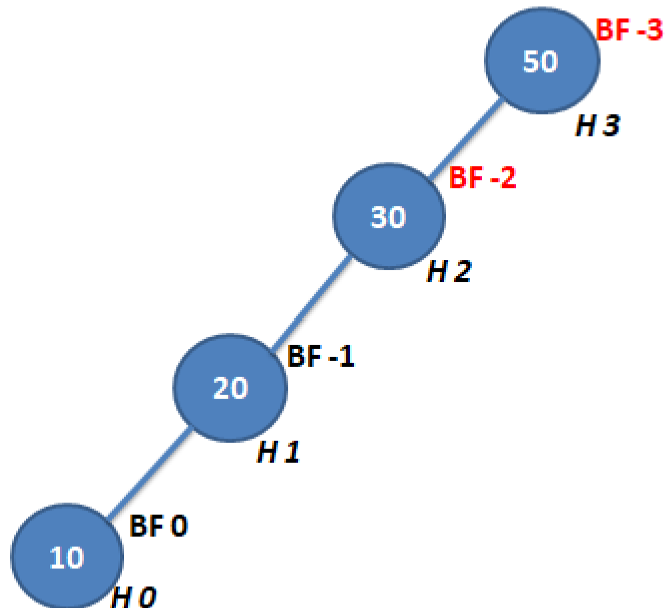

Figure 5 – Worst Case Unbalanced BST

The following figure instead, depicts a balanced Binary Search Tree:

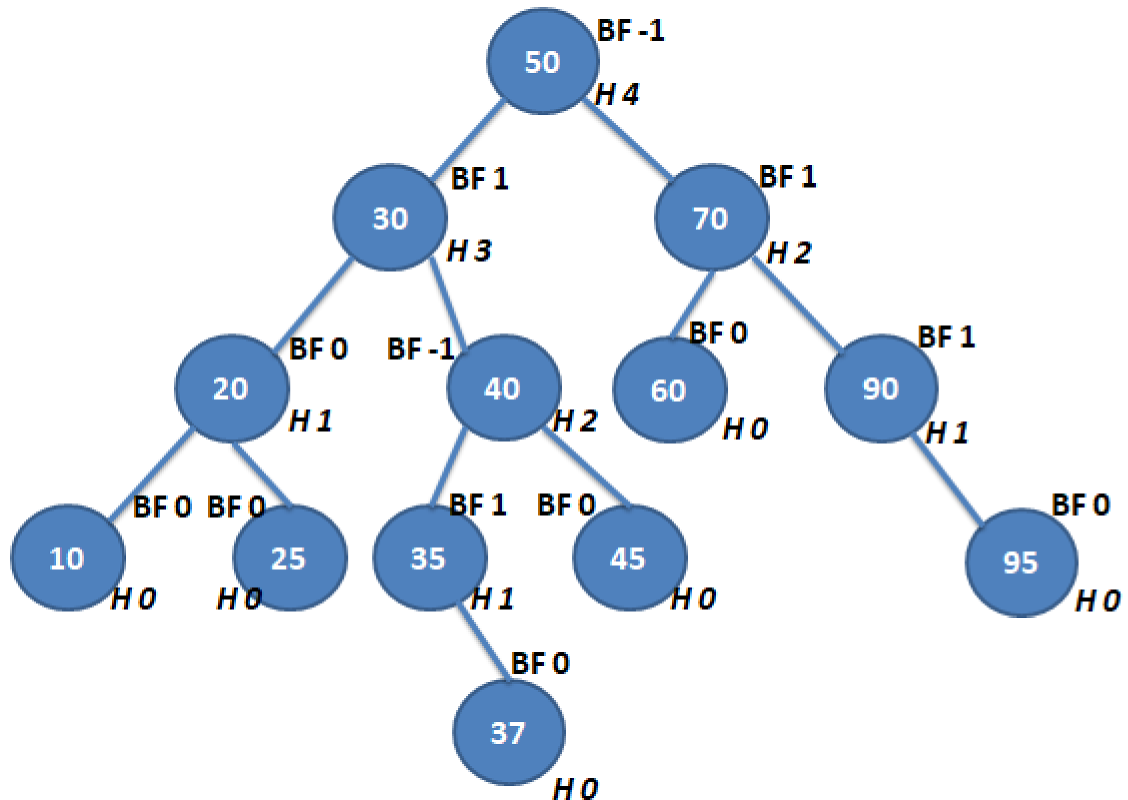

Figure 6 - Example of Balanced Search Binary Tree

It is easy to verify that all nodes of the Tree have a Balance Factor in the interval *{-1,0,1}*, therefore by definition the BST is said Balanced.

There are some interesting properties of BST that come directly from its definition. They are outlined below[2]:

- The minimum key in the BST can be found by starting from the root and descending the tree always on the left subtree (e.g. in Figure 6 we start from ***50*** and then go down on the left until we reach ***10***, which is the minimum key in the whole tree);

- Similarly, the maximum key in the BST can be found by starting from the root and descending the tree on the right subtree;

- Given any node, its immediate successor is the minimum of the right subtree (i.e. the leftmost child of the right child of the concerned node). Considering as an example node 30 of the BST in Figure 6, we see that the right child (40) has a leftmost child which is 35; this is the immediate successor of node 30 in the list;

- Similarly, the immediate predecessor of a node is the maximum of the left subtree (i.e. the rightmost child of the left child of the concerned node). Considering again node 30 in Figure 6, we see that the left child (20) has a rightmost child which is 25; this is the immediate predecessor of node 30 in the list.

[2] Properties are not demonstrated here because demonstration is out of the scope of this document; however, they are quite intuitive so that they can be easily remembered.

BST are very important structures in computer science, since their properties allow for easy and efficient lookup, insertion and removal of items. This is especially true for Balanced BST, which intuitively allow for best efficiency. Section below about BST Operations explains in details the algorithms used in BST, while the subsequent section discusses more deeply a special category of self-balancing Binary Search Trees called AVL Trees, showing how they can be used to achieve $O(\log_2 N)$ computational complexity, which makes them a suitable data structure for efficient handling of huge amount of data.

BST Operations

BST Lookup

The algorithm for lookup into a BST is very simple, since in every node all elements in the left subtree are lower and all elements in the right subtree are greater than the current node.

Therefore, given ***key*** the value to search and denoting with ***node*** the current node and with ***node.key*** the corresponding key, the search algorithm in pseudo code can be written as follows:

```
node search (node, key)
{
        if (node==NULL)
                return NULL;
        if (key==node.key)
                return (node);
        if (key>node.key)
                return (search(node.right,key));
        if (key<node.key)
                return (search(node.left,key));
}
```

This pseudo code provides the node which contains the key parameter given in input, or NULL if such value is not present. It shall be called first by specifying the Root node, then it acts recursively by iterating search on the left or right subtree depending on the comparison between the current value and the key parameter.

The following figure shows an example of traversal of the recursive search algorithm applied to the BST of Figure 6. Let's assume to look for ***key=37*** in the data structure. We start from the root node and compare the key to search with the current node key; in the first comparison, the key is lower (***37<50***), therefore we continue on the left subtree, and compare the key value (***37***) with the corresponding node key (***30***). This time the key is higher, then we go on the right subtree. We continue to apply the algorithm until either we find the element we are looking for or we reach a leaf which contains a different value. The traversal path along the BST is represented by the orange arrow in Figure 7 below.

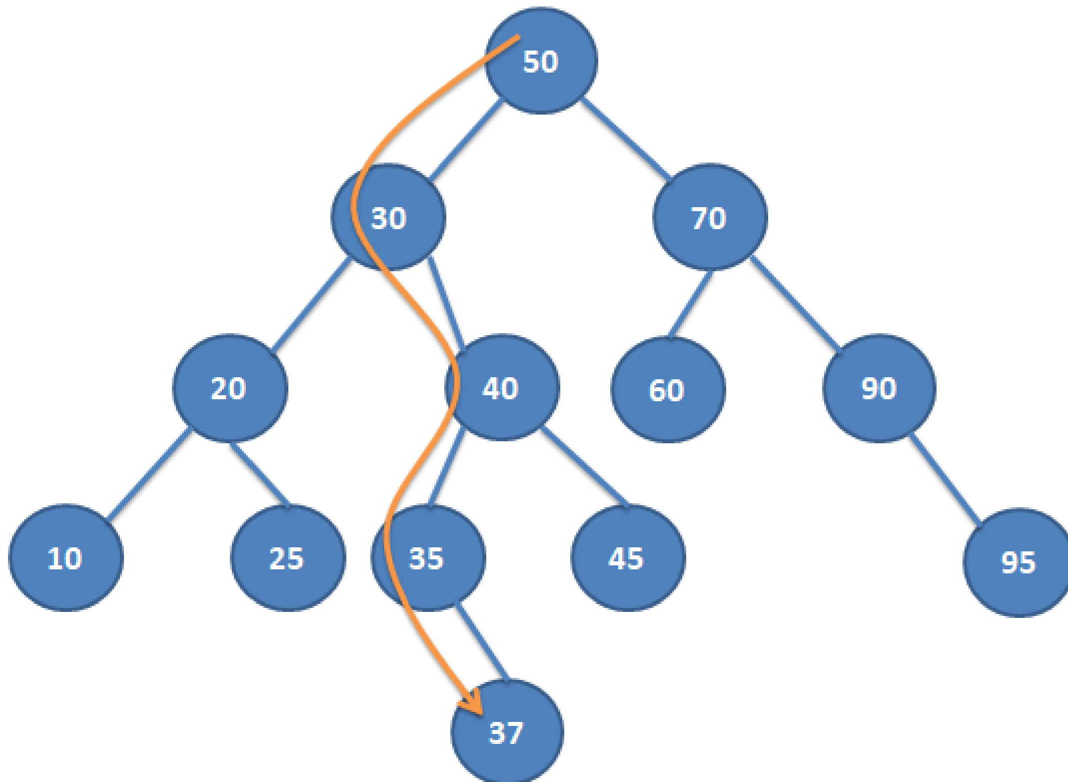

Figure 7 - Search of an element in the BST

It is easy to see that in the worst case the search algorithm requires ***H*** comparisons, where ***H*** is the height of the tree. If the BST is balanced, this value is a logarithmic function of the number of nodes N. In the previous example ***N=13*** and the worst-case scenario requires (***sup*** denotes the first integer immediately above the argument):

$$sup(\log_2 N) = 4$$

comparisons (in fact ***H=4*** in this BST). Conversely, if the tree is completely unbalanced (as in the case of Figure 5), as a matter of fact the search algorithm described above becomes analogous to a linear search algorithm in an ordered linked list, with linear worst-case complexity.

Summarizing, the same lookup algorithm provides either logarithmic or linear complexity depending on the type of BST, respectively balanced or unbalanced.

Binary Search Tree	Complexity
Unbalanced	$O(N)$
Balanced	$O(\log_2 N)$

Table 1 - BST Lookup Complexity

BST Insertion

The algorithm for inserting a new element shall take into account the basic property of BST: given an arbitrary node, all elements in the left subtree are lower and all elements in the right subtree are greater than the current node. Therefore, in order to insert a new node, we shall start from the root node and go down along the tree, descending either on the left or the right subtree, depending whether the new key to insert is respectively lower or higher than the key in the current node. We also assume that the BST cannot contain duplicate keys, which implies that an attempt to insert a key that is already present shall provide an error.

Given the above, the pseudo-code of the insertion algorithm is reported below:

```
node insert (node, key)
{
        if (node==NULL)
                return new Node(key);
        if (key>node.key)
                return (insert(node.right,key));
        if (key<node.key)
                return (insert(node.left,key));
        if (key==node.key)
                return (ERROR_DUPLICATE_KEY);

}
```

It shall be called first by specifying the Root node, then it acts recursively by iterating operations on the left or right subtree depending on the comparison between the current value and the key parameter. Recursion is terminated either when the key is found somewhere in the tree or when a null pointer is found. In the former case, a DUPLICATE KEY ERROR is obtained, and the new element is not inserted. In the latter case, instead, the new key is inserted as a child node of the previous node in the traversal path from root. It is worth to notice that each new element is always inserted as a leaf node.

As an example, let’s assume to insert ***key=36*** in the BST reported in Figure 6. By applying the algorithm described above from the root node, we easily realize that we traverse the tree along the same path shown in Figure 7, until we reach node with ***key=37***. When this node is reached, comparison between the new key and the current one implies that the new node shall be inserted as left child of node with ***key=37*** (in fact ***36<37***). The tree obtained is depicted below:

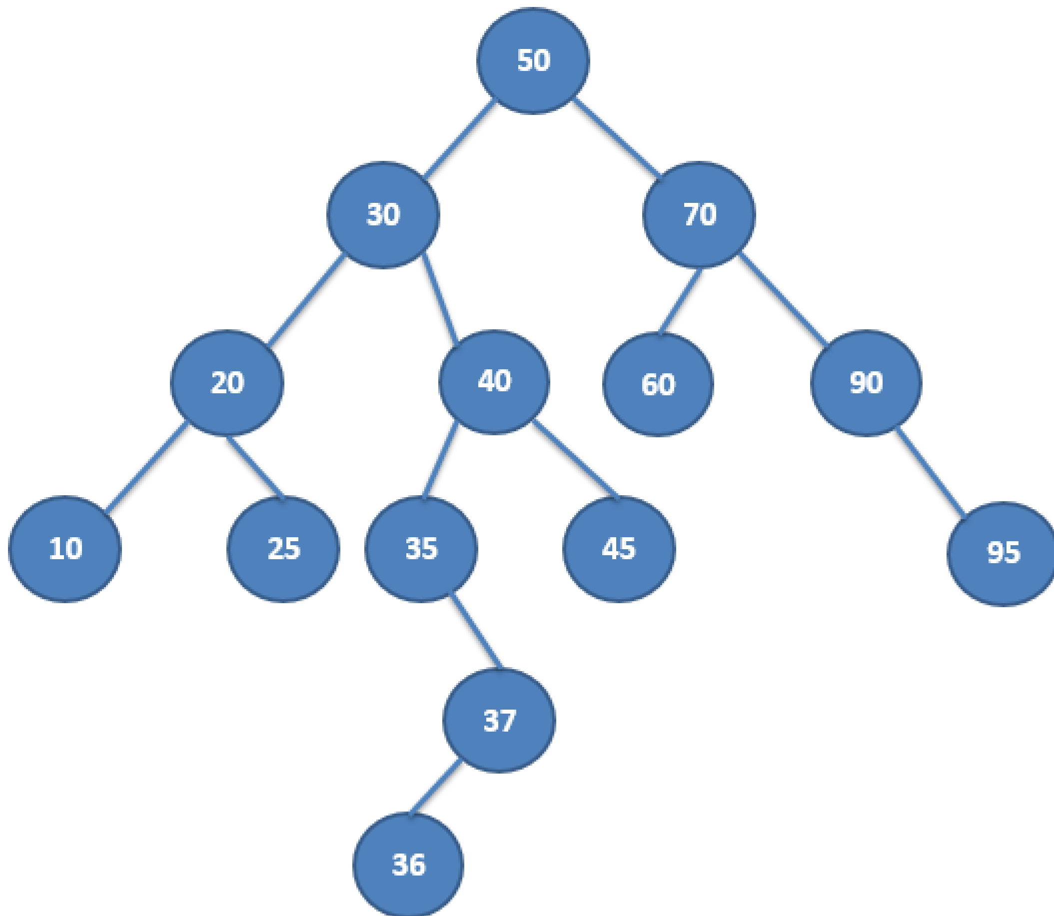

Figure 8 - Insertion of node 36 into the BST

Concerning complexity of the insertion algorithm, we can apply the same considerations already explained about lookup. In the worst case, the number of operations is linearly dependent on the tree height ***H***, therefore the following dependency on the number of nodes ***N*** is still valid:

Binary Search Tree	Complexity
Unbalanced	$O(N)$
Balanced	$O(log_2N)$

Table 2 - BST Insertion Complexity

It is extremely important to observe that the tree obtained after insertion is still a BST, since it satisfies the definition given above. However, if the BST is balanced before insertion, it might be unbalanced after the new node in added. In the example above, the tree is initially balanced (see Figure 6), but it loses this property after insertion, since the balance factor of node with ***Key=35*** becomes ***BF=2*** (the right subtree now has height ***H=2***, while the left subtree is empty). Later on we will show how a BST can be re-balanced after an insertion (and after a deletion as well), in order to keep it balanced and to achieve higher efficiency in read and write operation (i.e. logarithmic rather than linear).

BST Removal

Deletion of an element from a BST must be carried out while preserving its fundamental ordering property. There are 3 different cases that can occur while attempting to delete a specific key from a BST, namely:

1. the element to be deleted is a leaf of the BST;
2. the element to be deleted has only one child subtree;
3. the element to be deleted has both left and right subtrees.

The three cases are managed differently. Specifically, in the first case (the element to be deleted is a leaf of the BST), the element can be simply deleted from the tree, without affecting node ordering. As an example, let's delete ***key=36*** from the BST represented in Figure 8:
:

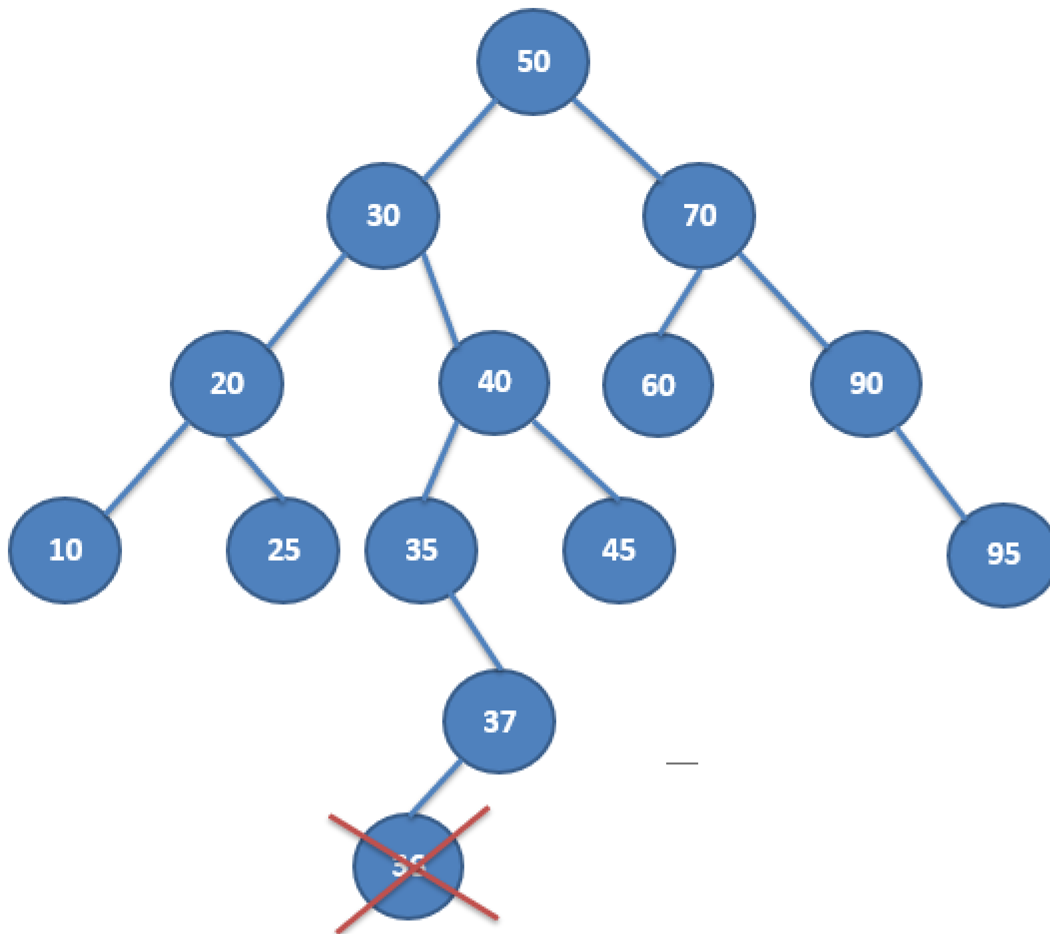

Figure 9 - Deletion of node 36 from the BST (case 1)

The operation consists in simple removal of the leaf containing ***36*** from the structure; the BST properties is not affected by this operation and the result is still a BST.

In the second scenario the element to be deleted has only one child subtree; in this case the task can be accomplished by removing the node and replacing it with its children. The following picture illustrates deletion of ***key=35*** from the BST represented in Figure 8:

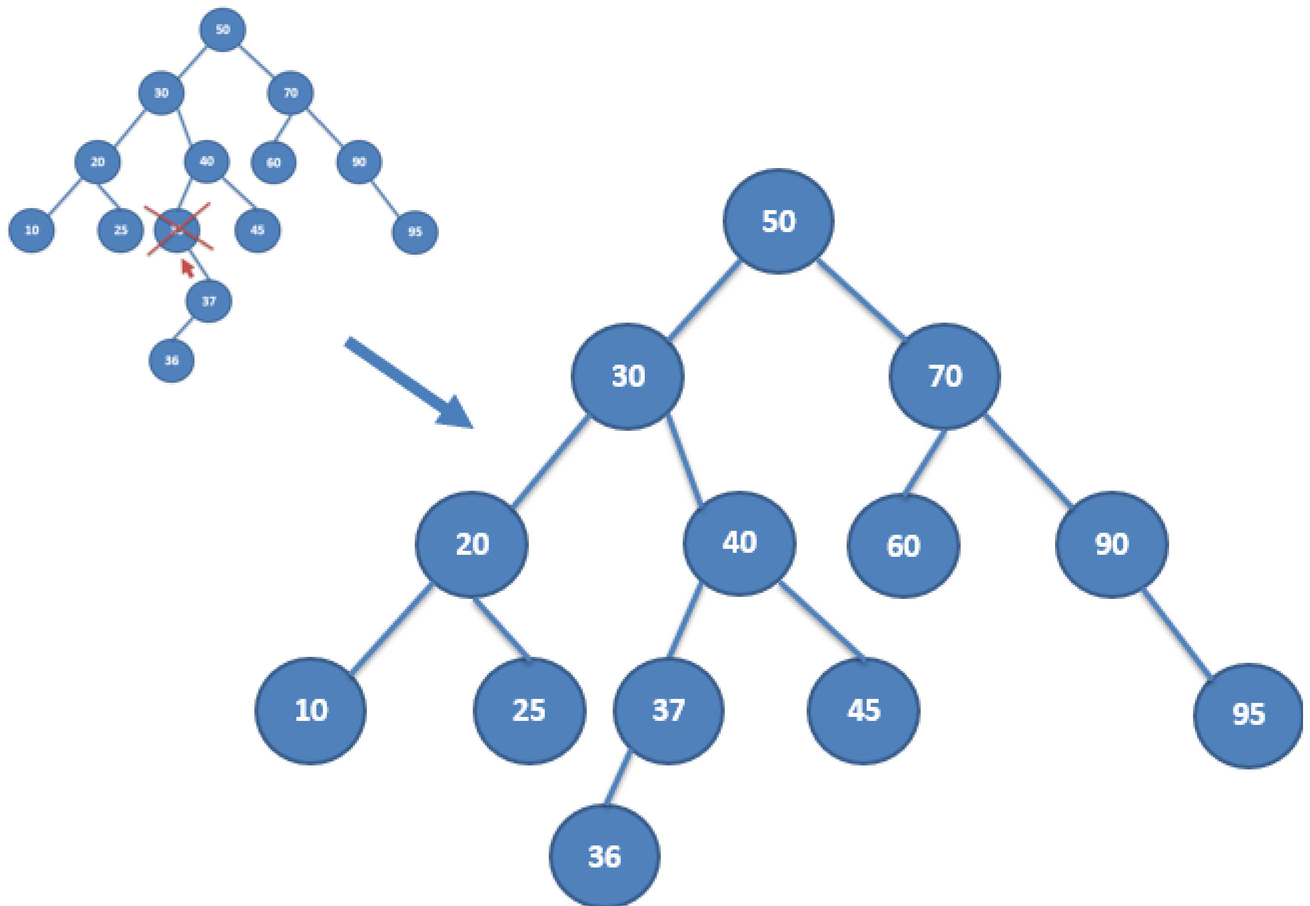

Figure 10 - Deletion of node 35 from BST (case 2)

The resulting tree is still a BST, since ordering among nodes is preserved.

The last scenario occurs when the element to be deleted has both left and right subtrees. In this case we must exchange the content of the node to be removed with its immediate successor (i.e. the leftmost child on the right subtree), then we must remove the latter from the tree. As an example, let's remove ***key=30*** from the BST of Figure 8. First, we identify the leftmost child on the right subtree, which is the node with ***key=35*** in the figure, then we exchange nodes ***30*** and ***35*** in the structure. The result is no more ordered, but the node to be deleted has moved down, and now it necessarily falls either in case 1 or case 2 above (see Figure 11 below).

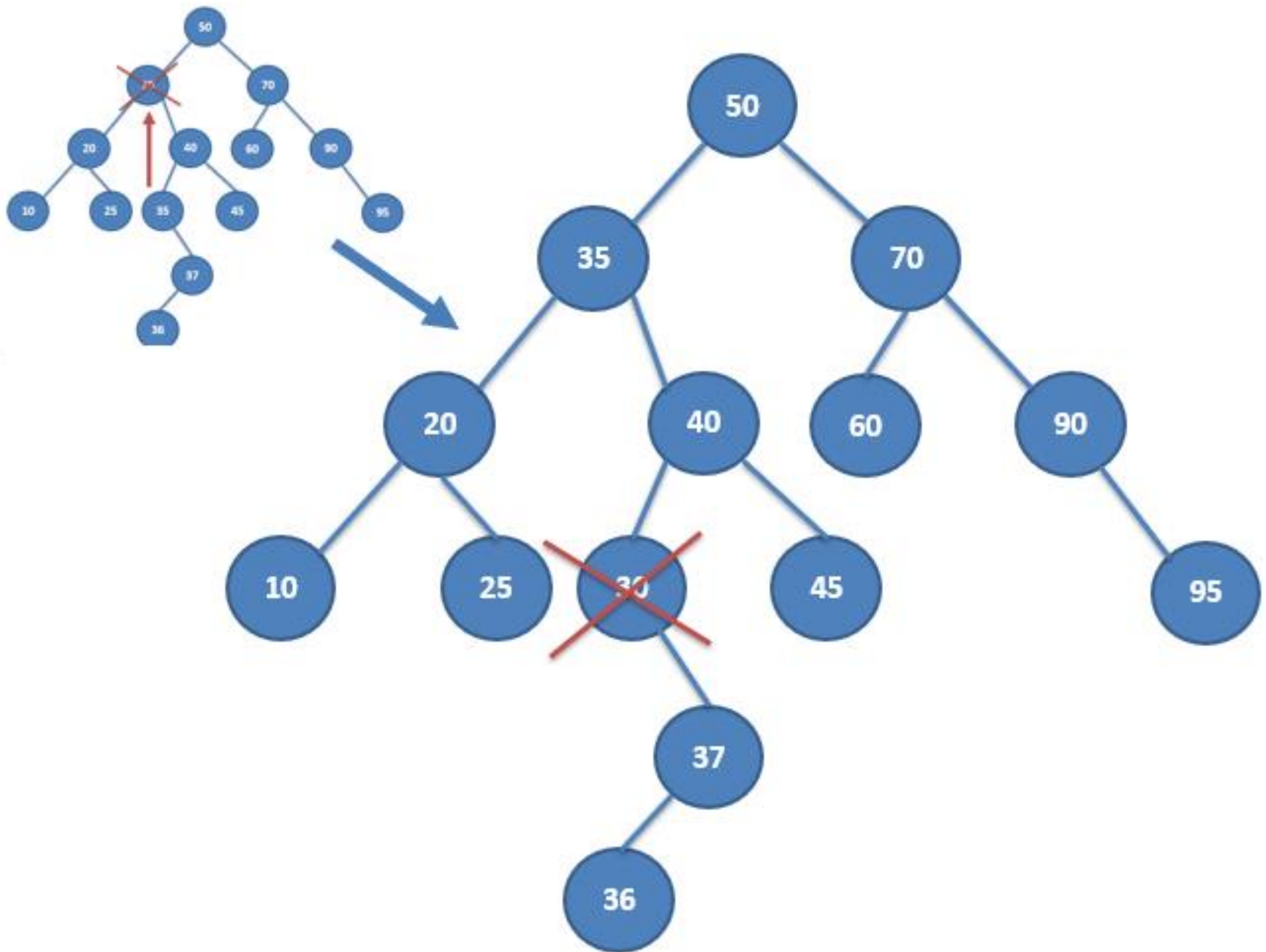

Figure 11 - Deletion of node 30 from BST (case 3) - First step

To complete deletion, now we have to remove the "new" ***30***. In this case we apply the method already described for case 2 above, therefore we remove the node and move up its subtree, obtaining the result shown in Figure 12 below:

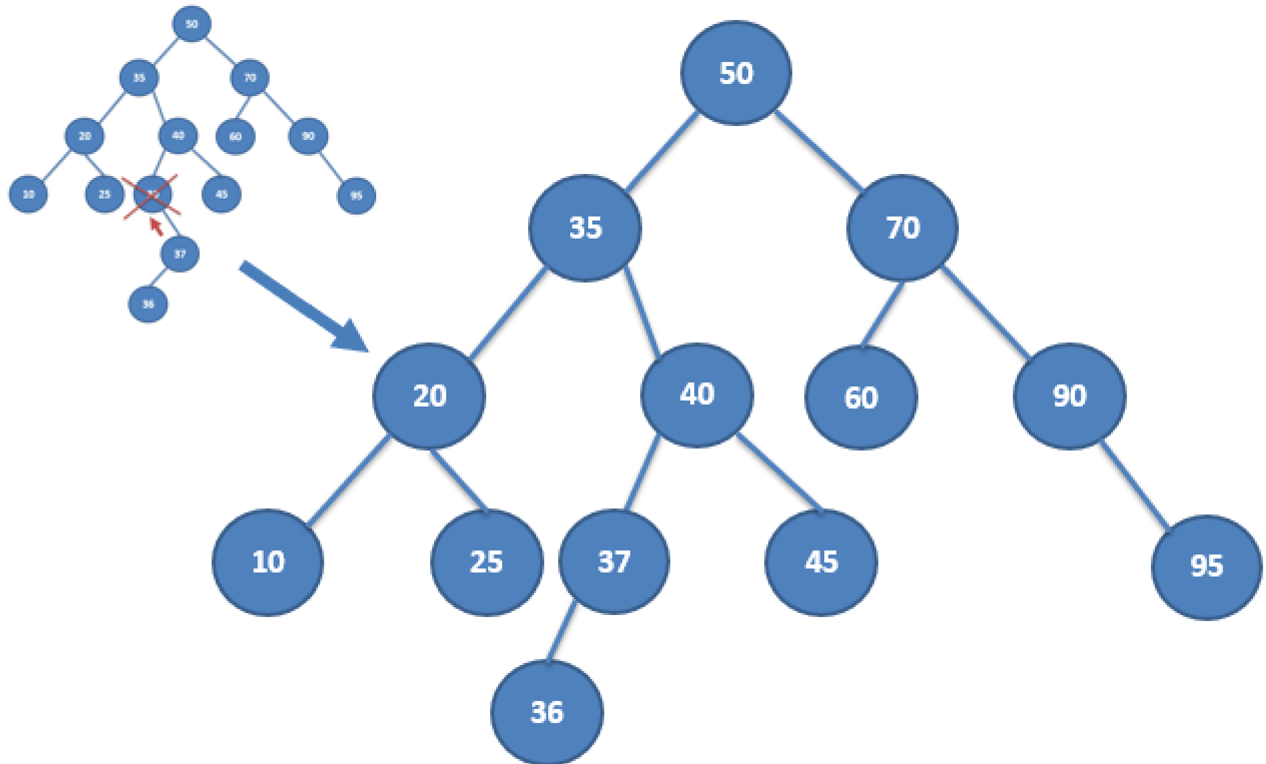

Figure 12 - Deletion of node 30 from BST (case 3) - Last step

It's easy to observe that the final result is still a BST, since ordering is maintained.

The algorithm in pseudo code is reported below:

```
node delete(node, key)
{
  if (node == null)
    return node
  if (key<node.key)
    node.left = delete(node.left, key)
  if (key>node.key)
    node.right = delete(node.right, key)
  // key=node.key
  if (node.left==null || node.right==null) {
      node = (node.left == null) ? node.right : node.left;
    } else {
      Node mostLeftChild = mostLeftChild(node.right);
      node.key = mostLeftChild.key;
      node.right = delete(node.right, node.key);
    }
  }
}
```

Complexity and balancing of the BST removal algorithm can be evaluated with analogous considerations already explained about lookup and insertion. In the worst case, the number of operations is linearly dependent on the tree height ***H***, therefore the following dependency on the number of nodes ***N*** is still valid:

Binary Search Tree	Complexity
Unbalanced	*O(N)*
Balanced	$O(log_2N)$

Table 3 - BST Removal Complexity

Similarly, it is easy to realize that the tree obtained after removal is still a BST, since it satisfies the definition given above. However, if the BST is balanced before deletion, it might be unbalanced after a node is deleted. As already mentioned, section about AVL Trees below shows how a BST can be re-balanced after a delete operation, to keep it balanced and to achieve higher efficiency.

BST Traversal

Tree traversal is the process to visit all the nodes of a tree. Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

Since all nodes are connected via links, the process is always started from the root node (random access in a tree is not allowed).

There are several traversal methods; the main ones are briefly described below:

- In-order Traversal
- Pre-order Traversal
- Post-order Traversal
- Breadth First or Level Order Traversal

In-order Traversal

In the case of In-order Traversal, the left subtree is visited first, then the root and finally the right subtree (remember that every node may represent a subtree itself, so the concept applies recursively until we reach leaves). The following pseudo-code provides a recursive implementation of In-order traversal:

```
void inOrderTraversal (node)
{
        if (node==NULL)
                return;

        inOrderTraversal (node.left);
        print (node.key);
        inOrderTraversal (node.right);

        return;
}
```

When In-order Traversal is applied to a BST, the result is the list of all nodes sorted in ascending order. For example, applying In-order traversal to the BST depicted in Figure 13 we obtain the following list of values:

10 – 20 – 25 - 30 – 35 - 40 - 45 – 50 – 70 - 90

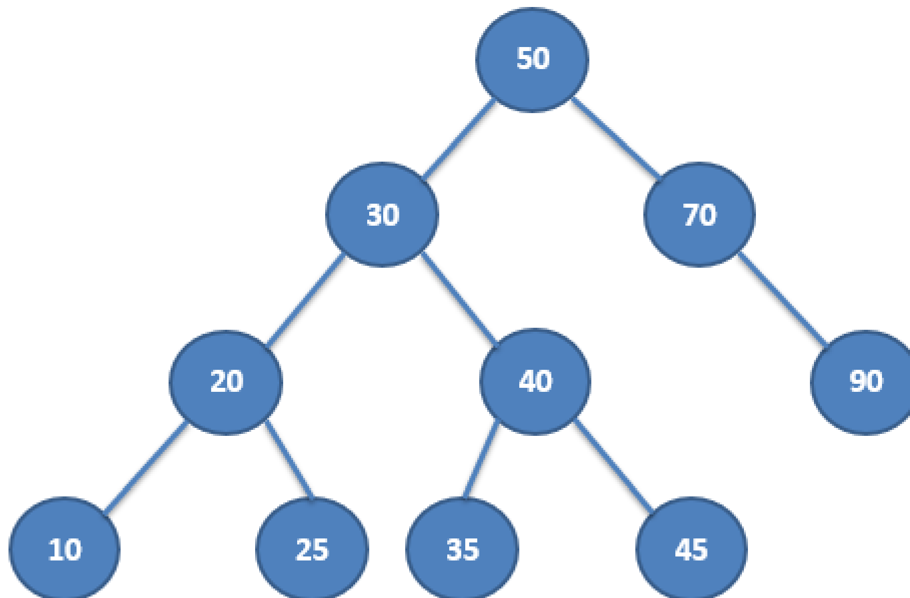

Figure 13 - Example BST

A variant of the In-order Traversal (strictly speaking it is not In-order Traversal) consists in visiting first the right subtree, then the root and later the left subtree (i.e. left and right subtrees are exchanged, but in any case the root is visited in the middle). This algorithm produces the list of elements of the tree in descending order. For example, applying this modified In-order Traversal to the same BST of Figure 13, we obtain the following result:

90 – 70 – 50 – 45 – 40 – 35 – 30 – 25 – 20 – 10

The corresponding algorithm expressed in pseudo-code is reported below:

```
void ModifiedInOrderTraversal (node)
{
        if (node==NULL)
                return;
        ModifiedInOrderTraversal (node.right);
        print (node.key);
        ModifiedInOrderTraversal (node.left);
        return;
}
```

Pre-order Traversal

In Pre-order Traversal, the first node visited is the root node, then traversal goes on with left and right subtrees respectively. When applied to a BST this algorithm does not produce any specific ordering. For example, Pre-order Traversal applied to BST in Figure 13 provides:

50 – 30 – 20 – 10 – 25 – 40 – 35 – 45 – 70 - 90

The algorithm for Pre-order Traversal is very similar to the previous one, the only difference being the order of invocation, since root precedes the left and right subtrees:

```
void PreOrderTraversal (node)
{
        if (node==NULL)
                return;
        print (node.key);
        PreOrderTraversal (node.left);
        PreOrderTraversal (node.right);
        return;
}
```

Post-order Traversal

When Post-order Traversal is used, the root is visited last, so that the traversal order becomes the following: left subtree first, then right subtree and finally the root. As in the previous case, this algorithm does not provide any specific order. For example, applying Post-order Traversal to the same BST of Figure 13, we obtain the following result:

10 – 25 – 20 – 35 - 45 – 40 – 30 – 90 – 70 – 50

The corresponding algorithm expressed in pseudo-code is reported below:

```
void PostOrderTraversal (node)
{
        if (node==NULL)
                return;
        PostOrderTraversal (node.left);
        PostOrderTraversal (node.right);
        print (node.key);
        return;
}
```

Breadth First/Level Order Traversal

The last method is Breadth First, also known as Level Order Traversal. It consists in traversing the tree level by level, starting from level 0 (the root) and descending the tree until all nodes are traversed. For each level, nodes are traversed from left to right. Level Order Traversal applied to BST in Figure 13 provides:

50 – 30 – 70 – 20 – 40 – 90 – 10 – 25 – 35 – 45

An interesting property of Level Order Traversal is the following. Observe that the original BST of Figure 13 is balanced. If we rebuild the BST from scratch by inserting its nodes one-by-one with the order given by Level Order Traversal, each intermediate result is always balanced (insertion follows the algorithm previously described). The following figure represents the sequence of step obtained by creating the tree from scratch and inserting the first five nodes (50 → 30 → 70 → 20 → 40):

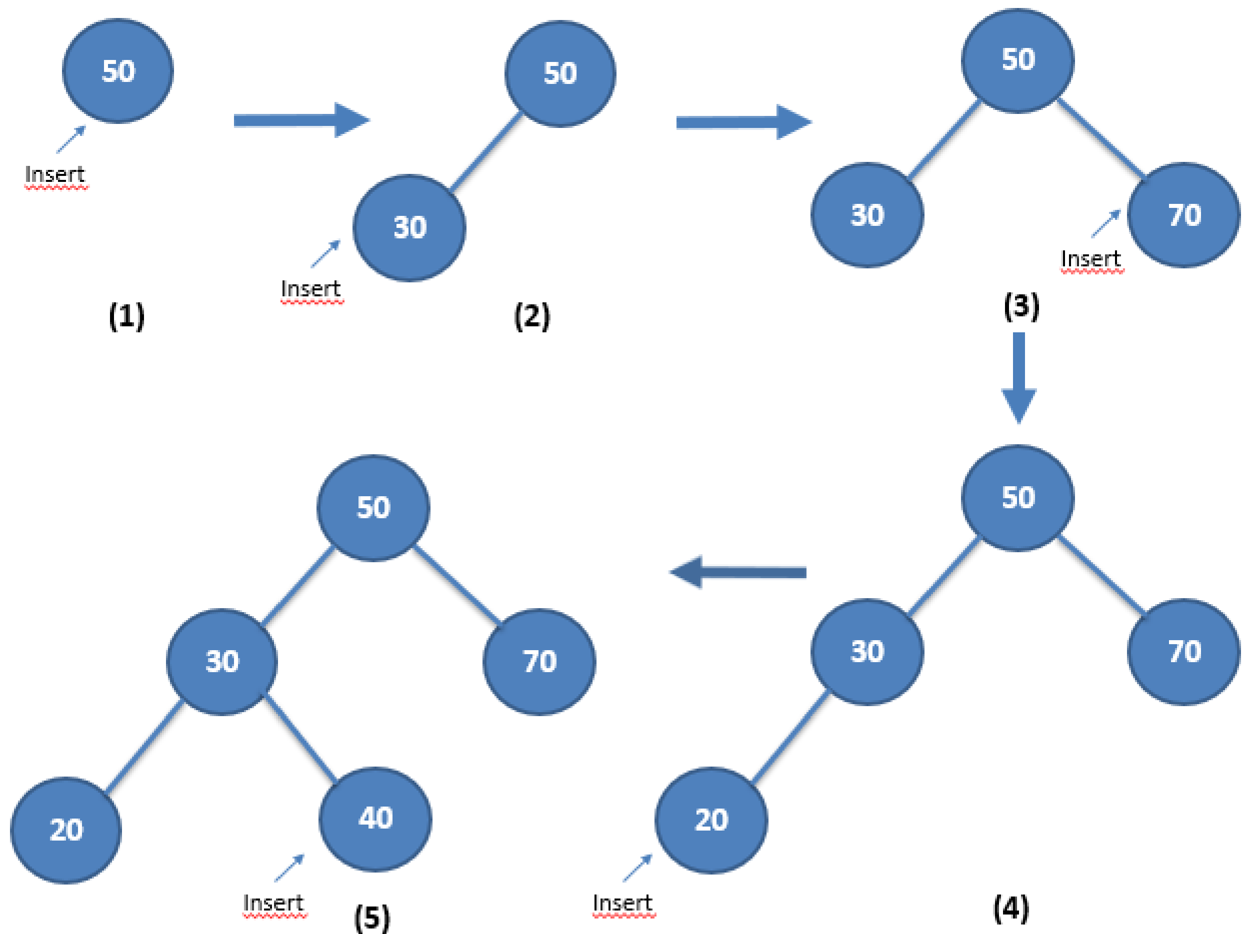

Figure 14 - Insertion of nodes from scratch according to Level Order Traversal

If we continue inserting nodes according to the same criteria, it is easy to realize that all intermediate results are balanced. This conclusion holds true whatever the original BST is, given that it is balanced. This is especially useful when the BST is an AVL tree (i.e. a self-balancing Binary Search Tree, see section AVL Trees below), because the process of re-building the whole tree from scratch is particularly efficient, given that it does not require any rebalancing.

A straightforward implementation of the Level Order Traversal is represented by the following pseudo-code:

```
/*Function to print level order traversal of tree*/
void printLevelorder(tree)
{
        for d = 1 to height(tree)
                printGivenLevel(tree, d);
}

/*Function to print all nodes at a given level*/
void printGivenLevel(tree, level)
{
        if tree is NULL then return;
        if level is 1, then
            print(tree->data);
        else if level greater than 1, then
           printGivenLevel(tree->left, level-1);
```

```
			printGivenLevel(tree->right, level-1);
	}
```

The code above is composed by 2 functions: the first one (***printGivenLevel***) is recursive, it traverses all the tree with a linear complexity (***O(N)***) and prints the content for a given level. This function is invoked first on level 1 (and prints the root), then on level 2, and so on up to the three's height. Since tree height is a logarithmic function of the number of nodes N, the whole algorithm has an overall complexity given by (***$O(N\log_2 N)$***), which is worse than the other algorithms explained above (which all have linear complexity).

A smarter implementation uses an iterative algorithm with exploits a FIFO structure (i.e. a queue) to explore the tree level by level with linear complexity:

```
	void iterativeLevelOrderTraversal (treeNode root)
	{
		Queue<treeNode> treeQueue;

		If (root==NULL) return;

		treeQueue.enqueue (root);
		while (treeQueue.empty()==FALSE)
		{
			treeNode currentNode = treeQueue.dequeue();
			print (currentNode->data);
			if (currentNode->left != NULL)
				treeQueue.enqueue (currentNode->left);
			if (currentNode->right != NULL)
				treeQueue.enqueue (currentNode->right);
		}
	}
```

AVL Trees

In computer science, an AVL tree (named after inventors Adelson-Velsky and Landis) is a self-balancing Binary Search Tree. According to the General Definitions previously introduced, this implies that the balance factor of any node shall be included in the range *{-1, 0, 1}*.

When the BST is initially balanced, it maintains this property upon BST lookup and BST traversal operations. However, it might lose balancing upon BST insertion or BST removal; if this happens, rebalancing is done to restore this property; the tree is rebalanced by one or more tree rotations. This is explained in detail below.

In an AVL tree, all read and write operations (i.e. lookup, insertion, and deletion) take **$O(\log_2 N)$** time in both the average and worst cases, where *N* is the number of nodes in the tree prior to the operation.

Rebalancing

As shown above, insertion and deletion of an element from a balanced BST tree may cause the resulting tree to be unbalanced.

Specifically, when a new element is inserted, it is always added as a leaf. Consequently, the Balance factor of all the nodes along the path from the root to the new leaf is affected. Before insertion, those nodes had a Balance Factor in the interval *{-1, 0, 1}*, after insertion this might be no longer true.

Consider e.g. the insertion of the new ***key=36*** in the balanced BST represented in Figure 15, in which the Balance Factor of all nodes along the path from root is explicitly shown):

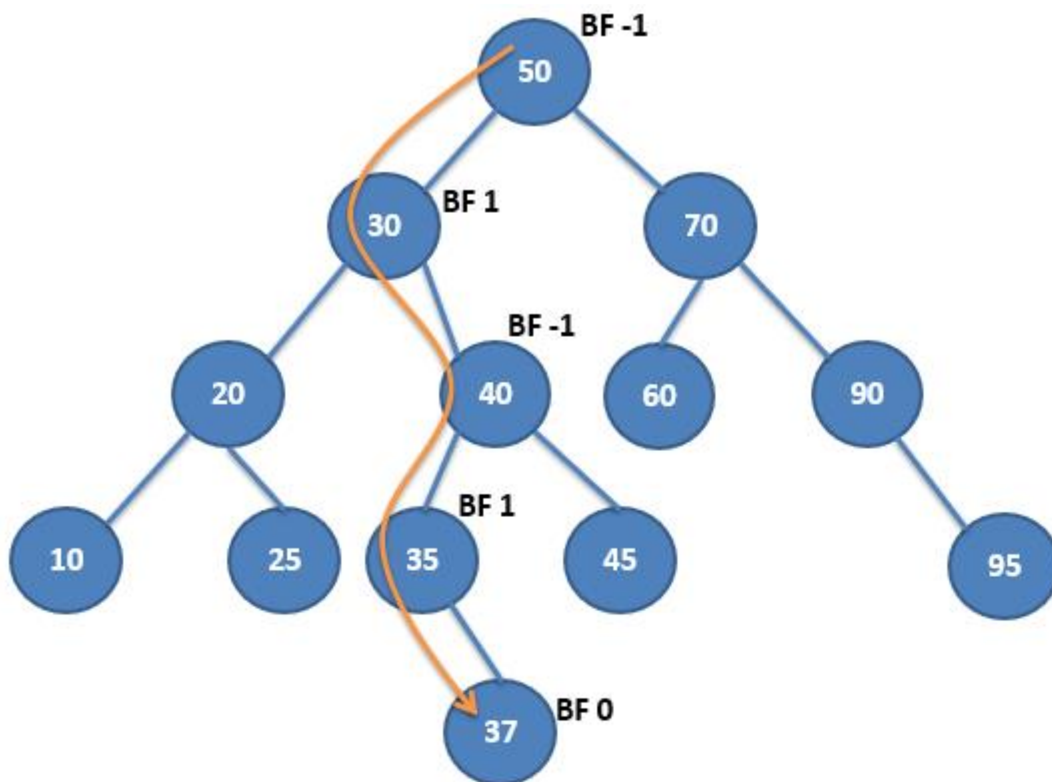

Figure 15 - Balance Factor of nodes along the path to the new node (before insertion)

After insertion of the new leaf, the values for Balance Factors become:

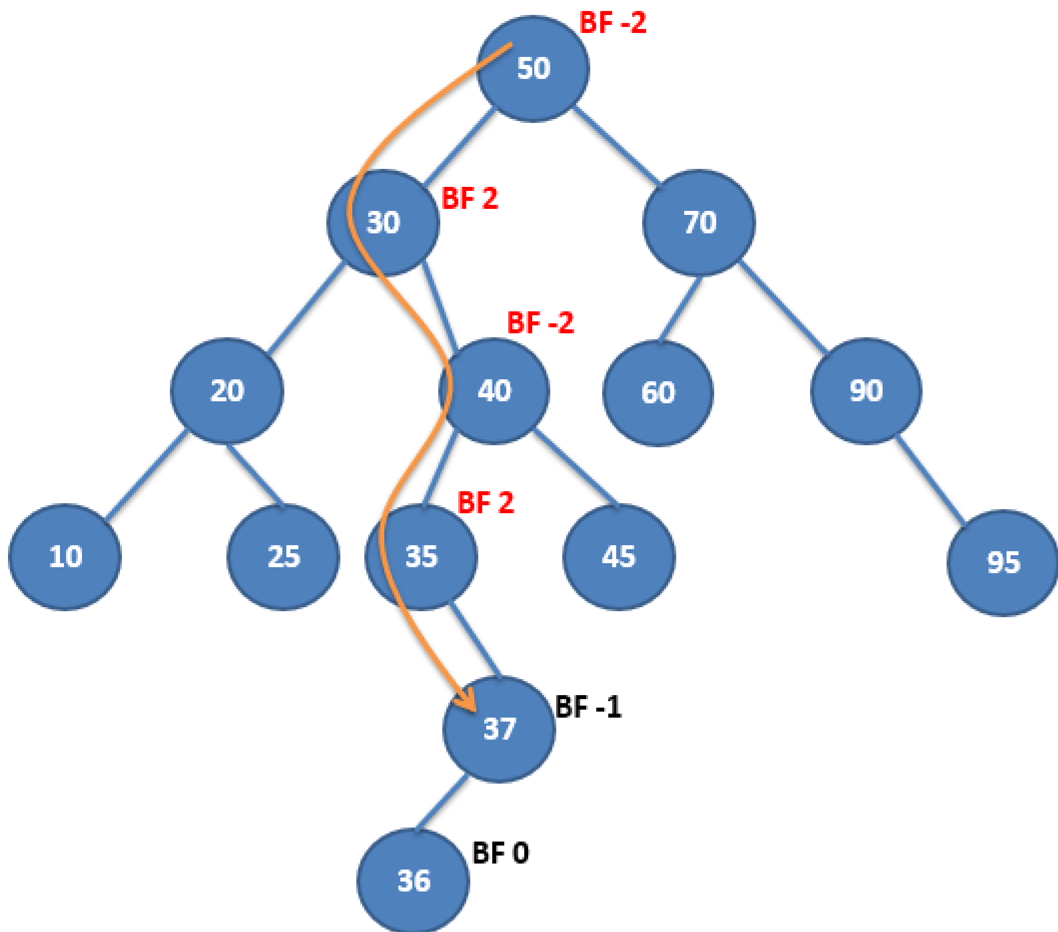

Figure 16 - Balance Factor of nodes along the path to the new node (after insertion)

In this case, all nodes have changed their Balance Factor, and four of them are outside the allowed interval, therefore the tree is unbalanced.

The tree shall be rebalanced going back along the orange path, from the leaf up to the root, and applying proper tree rotations to nodes whose Balance Factor is outside the allowed interval. In the figure above, starting from ***key=36*** and going up, the first node to be considered is ***35*** (i.e. the first node with ***|BF|>1***). The subtree whose root is ***35*** can be rotated as shown below:

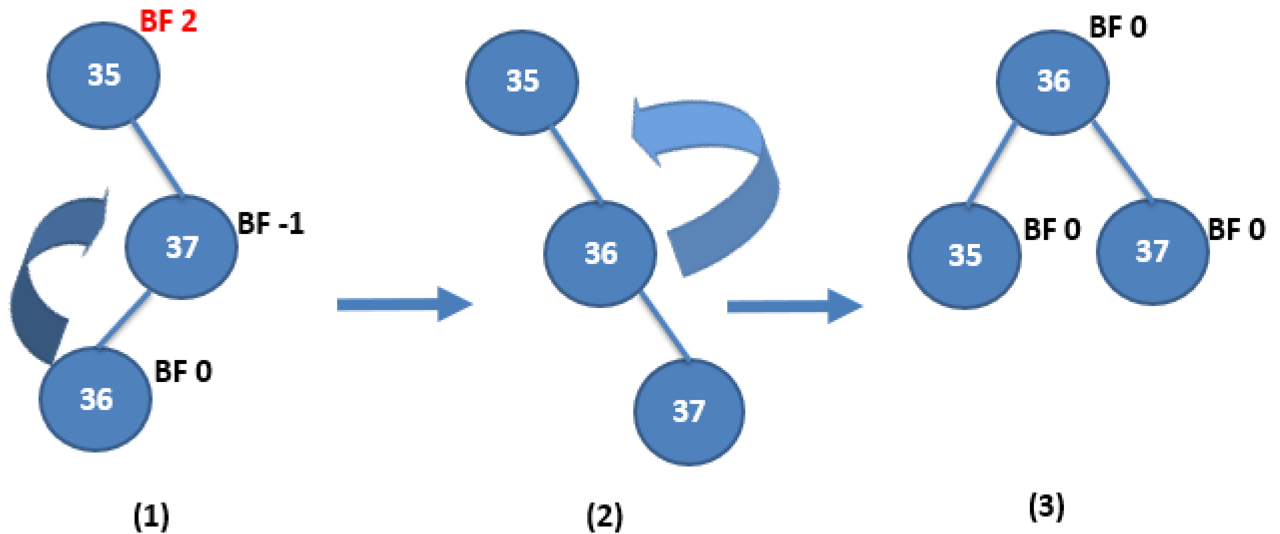

Figure 17 - Rotation of the subtree whose root is 35

In Figure 17, (1) represents the original subtree with root ***35***. First, we perform right rotation of the right subtree in (2), i.e. we exchange the two nodes ***36*** and ***37*** (note that ***37*** becomes the right subtree of ***36***, so to keep correct ordering), then we perform a left rotation around ***36*** obtaining the final result in (3), which is completely balanced. By substituting this result into the original tree, we obtain:

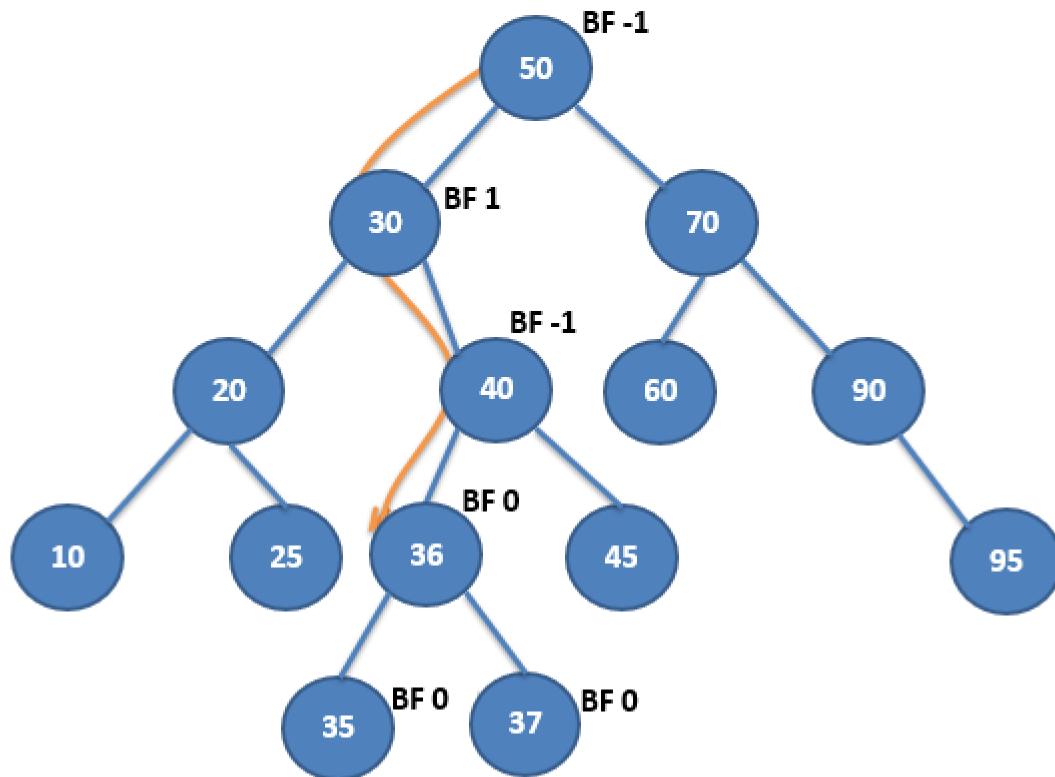

Figure 18 - BST after re-balancing

In the path towards the root along the orange arrow all Balance Factors must be re-computed (they might be affected due to rotation of subtree ***35***). Accidentally, in this example the result is completely

balanced, so Figure 18 already provides the final result. More generally, it may happen that further rotation are needed in order to finalize the operation.

The same concept and process apply as well to node deletion.

It should be noted that, in the worst case, the maximum number of rotations is always limited by three height ***H***, therefore the application of rebalancing after insertion (or deletion) does not affect complexity, which remains ***O(log₂N)***.

Left, Right and Combined Rotations

The basic rotations are summarized below.

If the Balance Factor of a node is ***BF > 1***, the right subtree has higher height against left subtree. If the right subtree of the right child has height higher than the left subtree of the right child, a simple left rotation is enough to rebalance the tree. This is the case shown below:

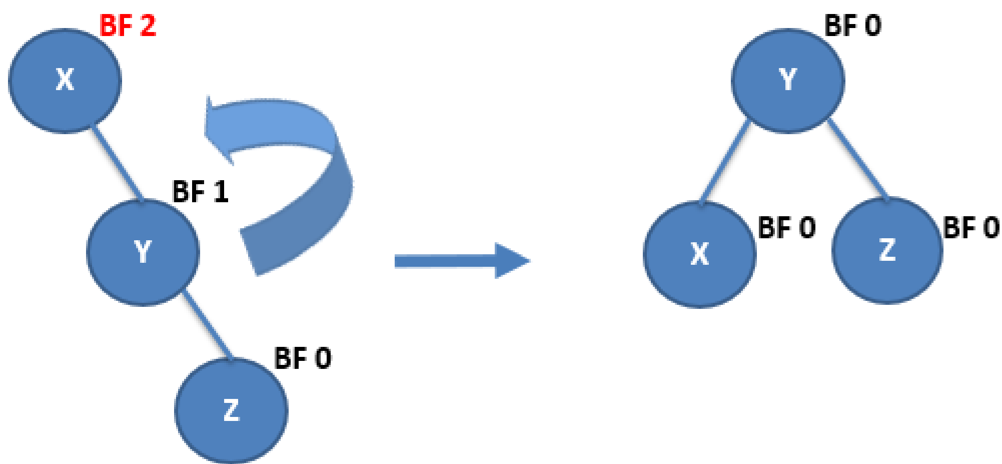

Figure 19 - Left Rotation

Rather, if the right subtree of the right child has height lower than the left subtree of the right child, then we have to perform right rotation of the right subtree followed by left rotation of the root (Figure 20).

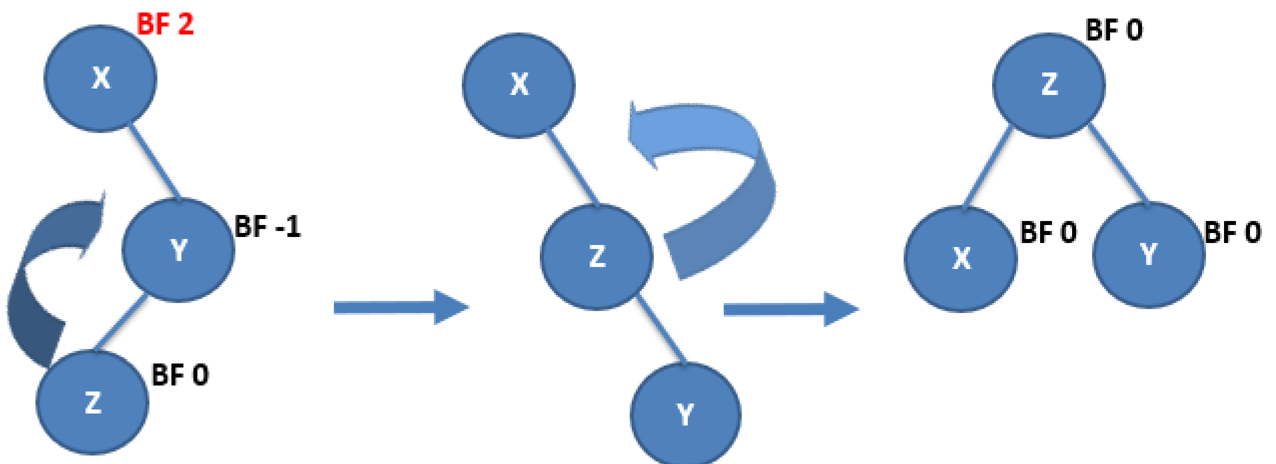

Figure 20 - Combination of Right and Left Rotation

The case of Balance Factor ***BF < -1*** (i.e left subtree with higher height against right subtree) is specular. If ***BF < -1*** and the left subtree of the left child has height higher than the right subtree of the left child, a simple right rotation is enough to rebalance the tree (Figure 21).

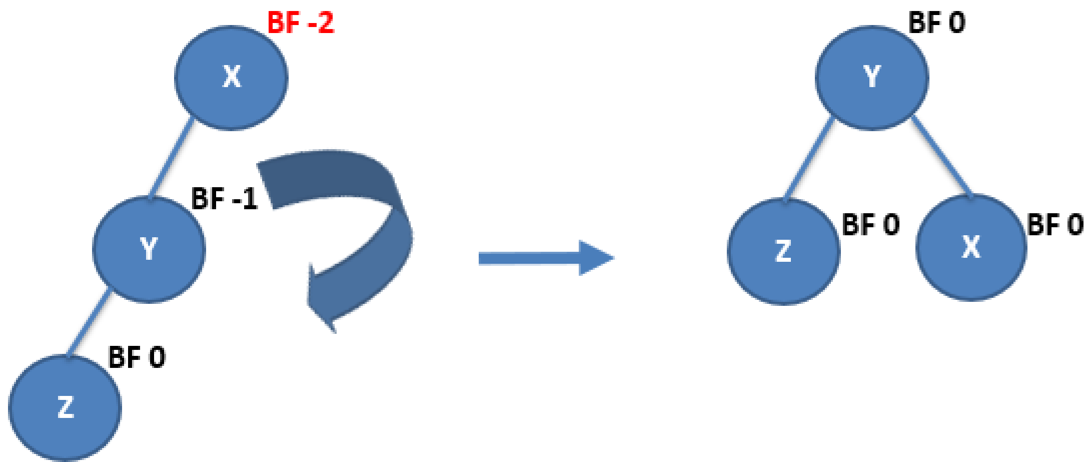

Figure 21 - Right Rotation

Otherwise, if the left subtree of the left child has height lower than the right subtree of the left child, then we have to perform left rotation of the left subtree followed by right rotation of the root (Figure 22):

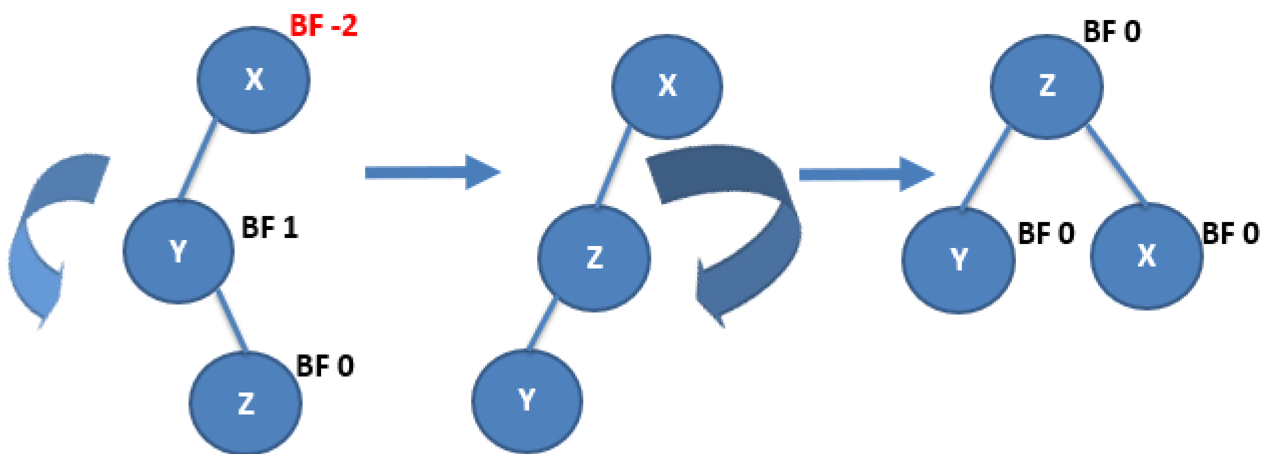

Figure 22 - Combination of Left and Right Rotation

CONCLUSIONS

This tutorial introduces the main concepts about one of the main data structures available in Computer Science, namely trees, and illustrates specifically a particular category of trees that are known as Binary Search Trees (BST for short).

The document focuses on definitions and properties of Binary Search Trees and illustrates the main algorithms applicable to such data structure for lookup, insertion, deletion and traversal. Specific attention is paid to the balance property of Binary Search Trees, showing how it affects the computation complexity of all previously mentioned algorithms. After that, the tutorial describes the rebalancing algorithm used by AVL Trees (a category of self-balancing Binary Search Trees), that ensures balancing and efficient access to data stored in the tree.

The concepts explained in the document constitutes the foundation for the open source *libfmrt* library (*Fast Memory Resident Table*), which provides in-memory efficient handling of big data tables (see https://www.roberto-mameli.it/software/ or https://github.com/Roberto-Mameli/libfmrt.git).