

UNIT -II

Binary Search Trees: Definition and Operations, AVL Trees: Definition and Operations, Applications.

B Trees: Definition and Operations.

2.1.INTRODUCTION

In a binary tree, every node can have a maximum of two children but there is no need to maintain the order of nodes basing on their values. In a binary tree, the elements are arranged in the order they arrive at the tree from top to bottom and left to right.

A binary tree has the following time complexities...

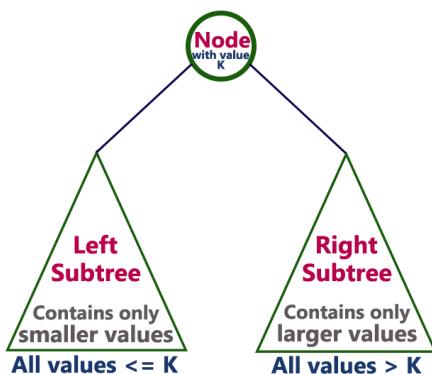
1. **Search Operation - O(n)**
2. **Insertion Operation - O(1)**
3. **Deletion Operation - O(n)**

2.1. BINARY SEARCH TREE

To enhance the performance of binary tree, we use a special type of binary tree known as **Binary Search Tree**. Binary search tree mainly focuses on the search operation in a binary tree. Binary search tree can be defined as follows...

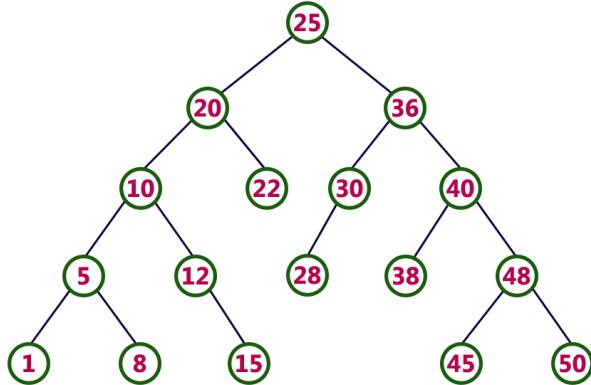
Binary Search Tree is a binary tree in which every node contains only smaller values in its left subtree and only larger values in its right subtree.

In a binary search tree, all the nodes in the left subtree of any node contains smaller values and all the nodes in the right subtree of any node contains larger values as shown in the following figure...

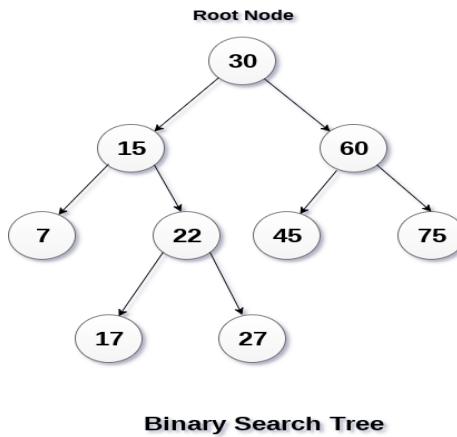


Example

The following tree is a Binary Search Tree. In this tree, left subtree of every node contains nodes with smaller values and right subtree of every node contains larger values.



Every binary search tree is a binary tree but every binary tree need not to be binary search tree.



Advantages of using binary search tree

1. Searching become very efficient in a binary search tree since, we get a hint at each step, about which sub-tree contains the desired element.
2. The binary search tree is considered as efficient data structure in compare to arrays and linked lists. In searching process, it removes half sub-tree at every step. Searching for an element in a binary search tree takes $O(\log_2 n)$ time. In worst case, the time it takes to search an element is $O(n)$.
3. It also speed up the insertion and deletion operations as compare to that in array and linked list.

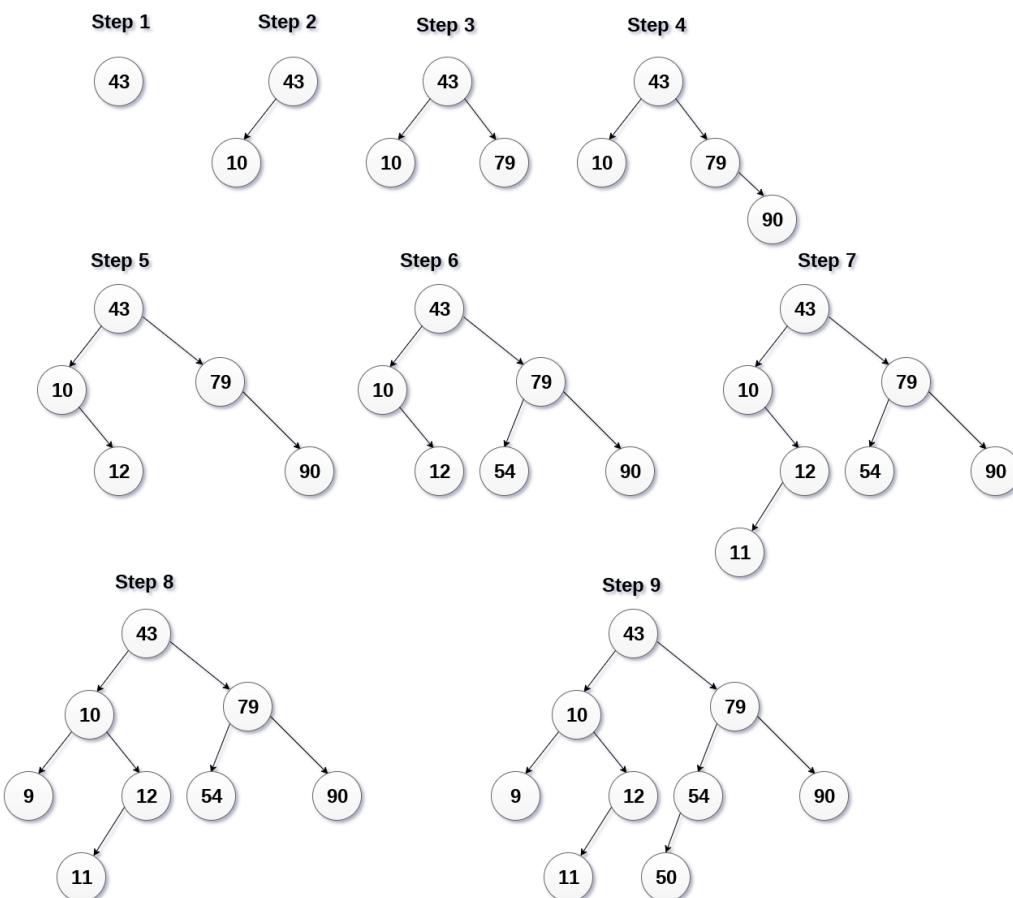
Example1:

Create the binary search tree using the following data elements.

43, 10, 79, 90, 12, 54, 11, 9, 50

1. Insert 43 into the tree as the root of the tree.
2. Read the next element, if it is lesser than the root node element, insert it as the root of the left sub-tree.
3. Otherwise, insert it as the root of the right of the right sub-tree.

The process of creating BST by using the given elements, is shown in the image below.



Binary search Tree Creation

Example2

Construct a Binary Search Tree by inserting the following sequence of numbers...

10,12,5,4,20,8,7,15 and 13

2.1.1. OPERATIONS ON A BINARY SEARCH TREE

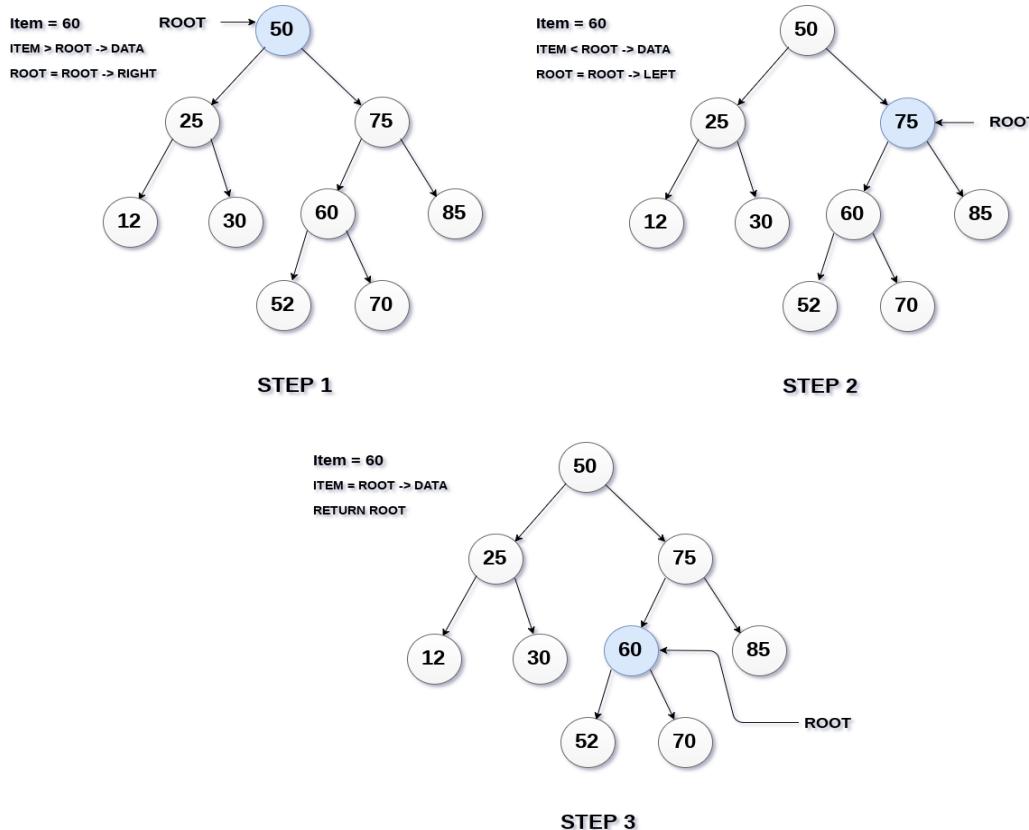
The following operations are performed on a binary search tree...

1. Search
2. Insertion
3. Deletion

1. Search Operation in BST

Searching means finding or locating some specific element or node within a data structure. However, searching for some specific node in binary search tree is pretty easy due to the fact that, element in BST are stored in a particular order.

1. Compare the element with the root of the tree.
2. If the item is matched then return the location of the node.
3. Otherwise check if item is less than the element present on root, if so then move to the left sub-tree.
4. If not, then move to the right sub-tree.
5. Repeat this procedure recursively until match found.
6. If element is not found then return NULL.



Algorithm:

Search (ROOT, ITEM)

- **Step 1:** IF ROOT \rightarrow DATA = ITEM OR ROOT = NULL
 Return ROOT
 ELSE
 IF ROOT < ITEM \rightarrow DATA
 Return search(ROOT \rightarrow LEFT, ITEM)
 ELSE
 Return search(ROOT \rightarrow RIGHT, ITEM)
 [END OF IF]
 [END OF IF]
- **Step 2:** END

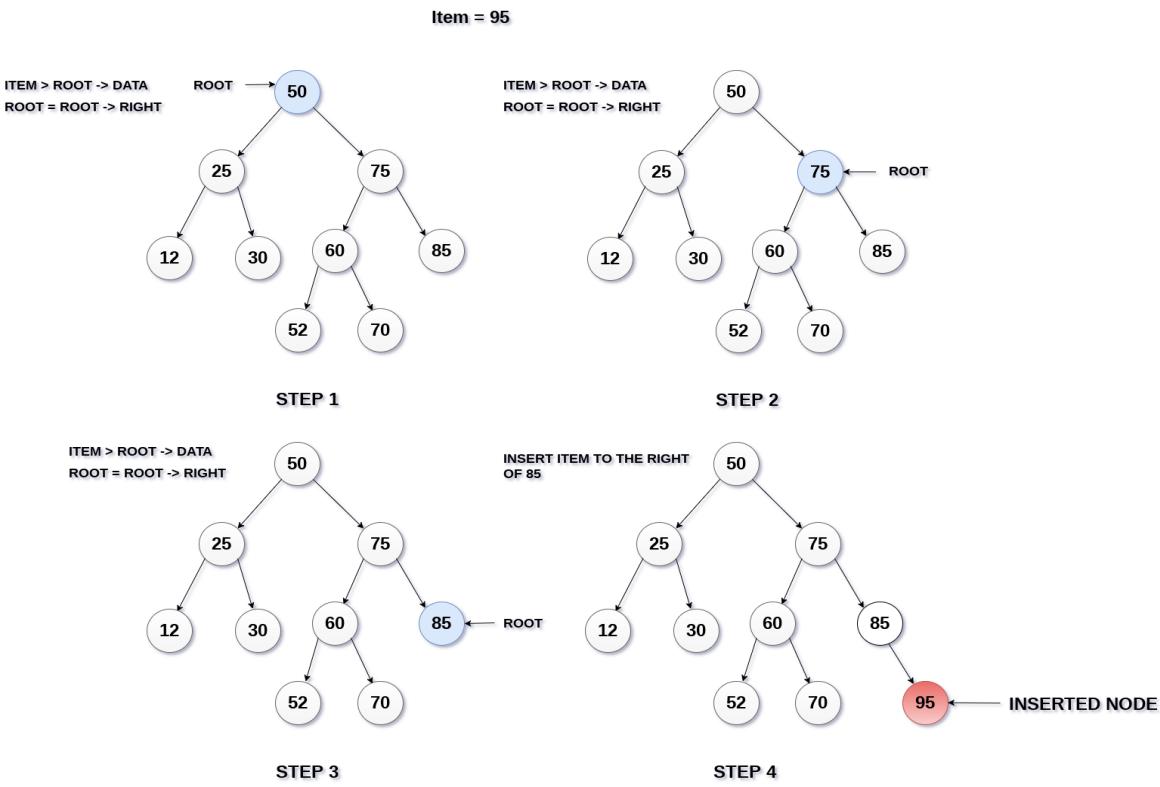
2.Insert Operation in BST

Insert function is used to add a new element in a binary search tree at appropriate location. Insert function is to be designed in such a way that, it must not violate the property of binary search tree at each value.

1. Allocate the memory for tree.
2. Set the data part to the value and set the left and right pointer of tree, point to NULL.
3. If the item to be inserted, will be the first element of the tree, then the left and right of this node will point to NULL.
4. Else, check if the item is less than the root element of the tree, if this is true, then recursively perform this operation with the left of the root.
5. If this is false, then perform this operation recursively with the right sub-tree of the root.

Insert (TREE, ITEM)

- Step 1: IF TREE = NULL
 Allocate memory for TREE
 SET TREE \rightarrow DATA = ITEM
 SET TREE \rightarrow LEFT = TREE \rightarrow RIGHT = NULL
 ELSE
 IF ITEM < TREE \rightarrow DATA
 Insert(TREE \rightarrow LEFT, ITEM)
 ELSE
 Insert(TREE \rightarrow RIGHT, ITEM)
 [END OF IF]
 [END OF IF]
- Step 2: END



3.Delete Operation in BST

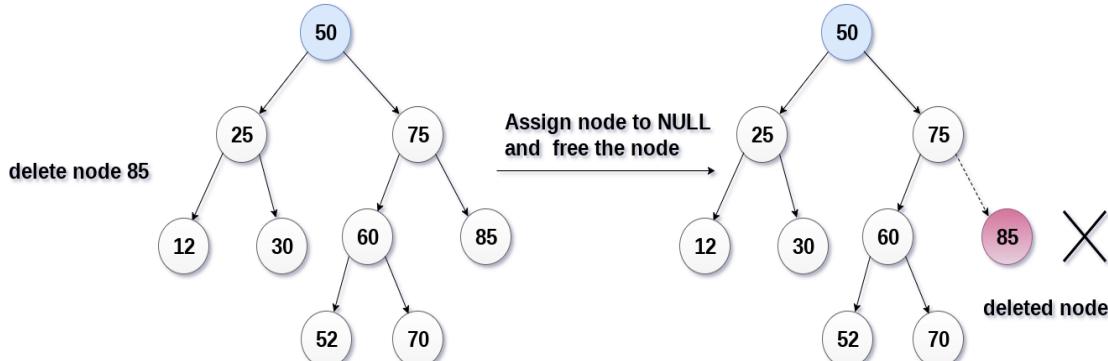
Delete function is used to delete the specified node from a binary search tree. However, we must delete a node from a binary search tree in such a way, that the property of binary search tree doesn't violate.

There are **three situations of deleting a node** from binary search tree.

a)The node to be deleted is a leaf node

It is the simplest case, in this case, replace the leaf node with the NULL and simple free the allocated space.

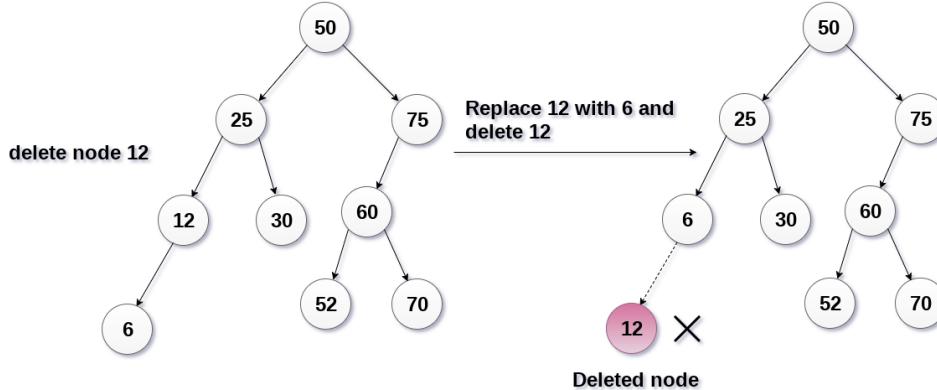
In the following image, we are deleting the node 85, since the node is a leaf node, therefore the node will be replaced with NULL and allocated space will be freed.



b)The node to be deleted has only one child.

In this case, replace the node with its child and delete the child node, which now contains the value which is to be deleted. Simply replace it with the NULL and free the allocated space.

In the following image, the node 12 is to be deleted. It has only one child. The node will be replaced with its child node and the replaced node 12 (which is now leaf node) will simply be deleted.



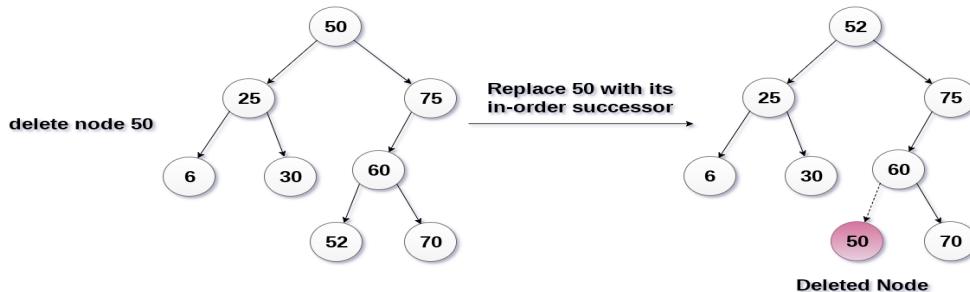
c)The node to be deleted has two children.

It is a bit complexed case compare to other two cases. However, the node which is to be deleted, is replaced with its in-order successor or predecessor recursively until the node value (to be deleted) is placed on the leaf of the tree. After the procedure, replace the node with NULL and free the allocated space.

In the following image, the node 50 is to be deleted which is the root node of the tree. The in-order traversal of the tree given below.

6, 25, 30, 50, 52, 60, 70, 75.

replace 50 with its in-order successor 52. Now, 50 will be moved to the leaf of the tree, which will simply be deleted.



Algorithm Delete (TREE, ITEM)

Step1: IF TREE=NULL

```
    Write "item not found in the tree" ELSE IF ITEM < TREE -> DATA  
        Delete(TREE->LEFT,ITEM)  
    ELSE IF ITEM>TREE->DATA  
        Delete(TREE->RIGHT,ITEM)  
    ELSE IF TREE->LEFT AND TREE->RIGHT  
        SET TEMP = findLargestNode(TREE -> LEFT)  
        SET TREE -> DATA = TEMP -> DATA  
        Delete(TREE -> LEFT, TEMP -> DATA)  
    ELSE  
        SET TEMP = TREE  
    IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL  
        SET TREE = NULL  
    ELSE IF TREE -> LEFT != NULL  
        SET TREE = TREE -> LEFT  
    ELSE  
        SET TREE = TREE -> RIGHT  
    [END OF IF]  
    FREE TEMP  
    [END OF IF]
```

Step 2: END

2.2. AVL TREES

AVL tree is a height-balanced binary search tree. That means, an AVL tree is also a binary search tree but it is a balanced tree. A binary tree is said to be balanced if, the difference between the heights of left and right subtrees of every node in the tree is either -1, 0 or +1. In other words, a binary tree is said to be balanced if the height of left and right children of every node differ by either -1, 0 or +1. In an AVL tree, every node maintains an extra information known as **balance factor**. The AVL tree was introduced in the year 1962 by G.M. Adelson-Velsky and E.M. Landis.

An AVL tree is defined as follows...

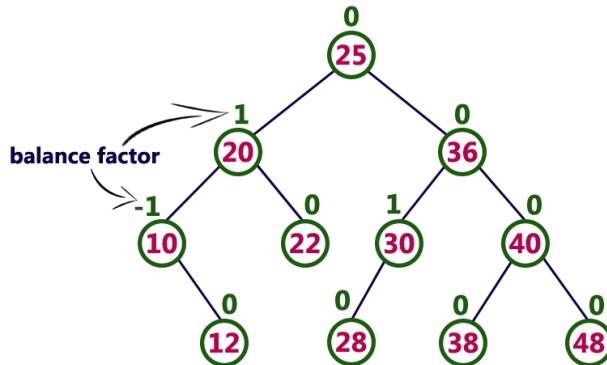
An AVL tree is a balanced binary search tree. In an AVL tree, balance factor of every node is either -1, 0 or +1.

Balance factor of a node is the difference between the heights of the left and right subtrees of that node. The balance factor of a node is calculated either **height of left subtree - height of**

right subtree (OR) height of right subtree - height of left subtree. In the following explanation, we calculate as follows...

$$\text{Balance factor} = \text{heightOfLeftSubtree} - \text{heightOfRightSubtree}$$

Example of AVL Tree



The above tree is a binary search tree and every node is satisfying balance factor condition. So this tree is said to be an AVL tree.

Every AVL Tree is a binary search tree but every Binary Search Tree need not be AVL tree.

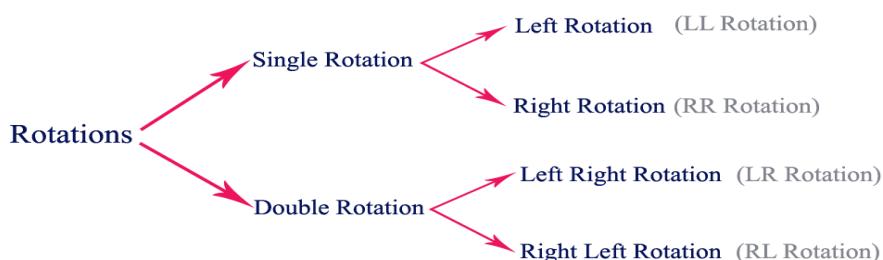
2.2.1.AVL Tree Rotations

In AVL tree, after performing operations like insertion and deletion we need to check the **balance factor** of every node in the tree. If every node satisfies the balance factor condition then we conclude the operation otherwise we must make it balanced. Whenever the tree becomes imbalanced due to any operation we use **rotation** operations to make the tree balanced.

Rotation operations are used to make the tree balanced.

Rotation is the process of moving nodes either to left or to right to make the tree balanced.

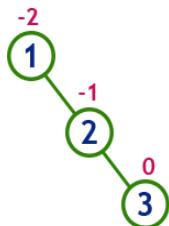
There are **four** rotations and they are classified into **two** types.



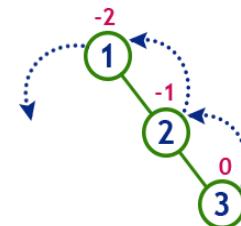
i) Single Left Rotation (LL Rotation)

In LL Rotation, every node moves one position to left from the current position. To understand LL Rotation, let us consider the following insertion operation in AVL Tree...

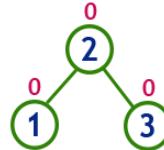
insert 1, 2 and 3



Tree is imbalanced



To make balanced we use
LL Rotation which moves
nodes one position to left

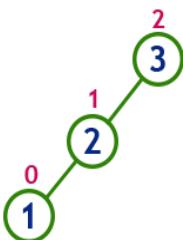


After LL Rotation
Tree is Balanced

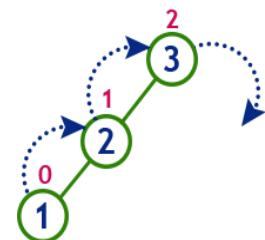
ii) Single Right Rotation (RR Rotation)

In RR Rotation, every node moves one position to right from the current position. To understand RR Rotation, let us consider the following insertion operation in AVL Tree...

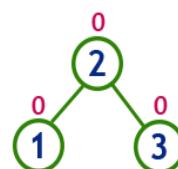
insert 3, 2 and 1



Tree is imbalanced
because node 3 has balance factor 2



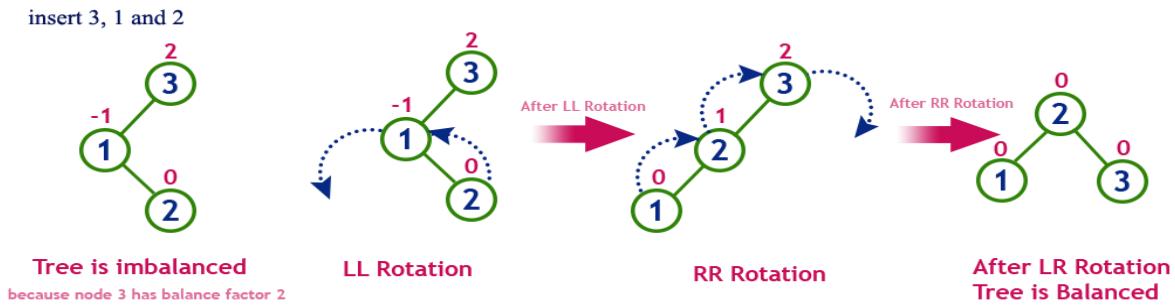
To make balanced we use
RR Rotation which moves
nodes one position to right



After RR Rotation
Tree is Balanced

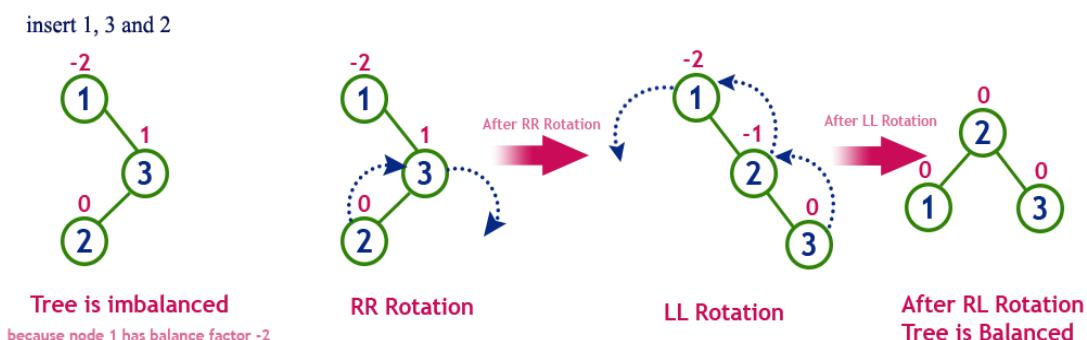
iii) Left Right Rotation (LR Rotation)

The LR Rotation is a sequence of single left rotation followed by a single right rotation. In LR Rotation, at first, every node moves one position to the left and one position to right from the current position. To understand LR Rotation, let us consider the following insertion operation in AVL Tree...



iv) Right Left Rotation (RL Rotation)

The RL Rotation is sequence of single right rotation followed by single left rotation. In RL Rotation, at first every node moves one position to right and one position to left from the current position. To understand RL Rotation, let us consider the following insertion operation in AVL Tree...



2.2.2. Operations on an AVL Tree

The following operations are performed on AVL tree...

1. Search
2. Insertion
3. Deletion

i) Search Operation in AVL Tree

In an AVL tree, the search operation is performed with $O(\log n)$ time complexity. The search operation in the AVL tree is similar to the search operation in a Binary search tree. We use the following steps to search an element in AVL tree...

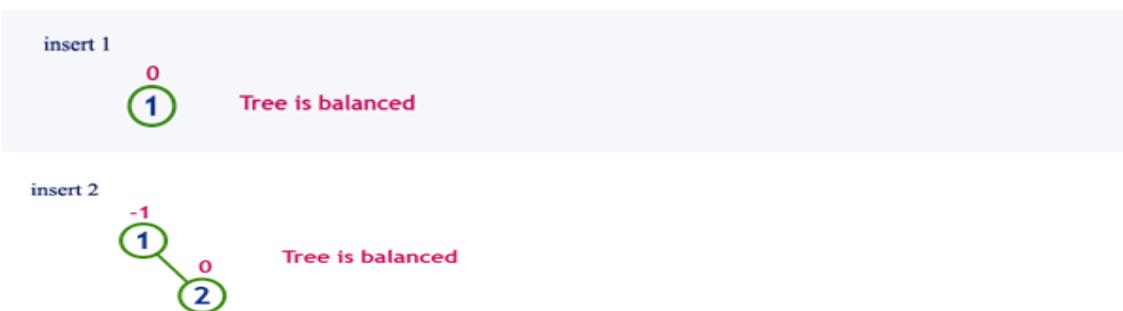
- **Step 1** - Read the search element from the user.
- **Step 2** - Compare the search element with the value of root node in the tree.
- **Step 3** - If both are matched, then display "Given node is found!!!" and terminate the function
- **Step 4** - If both are not matched, then check whether search element is smaller or larger than that node value.
- **Step 5** - If search element is smaller, then continue the search process in left subtree.
- **Step 6** - If search element is larger, then continue the search process in right subtree.
- **Step 7** - Repeat the same until we find the exact element or until the search element is compared with the leaf node.
- **Step 8** - If we reach to the node having the value equal to the search value, then display "Element is found" and terminate the function.
- **Step 9** - If we reach to the leaf node and if it is also not matched with the search element, then display "Element is not found" and terminate the function.

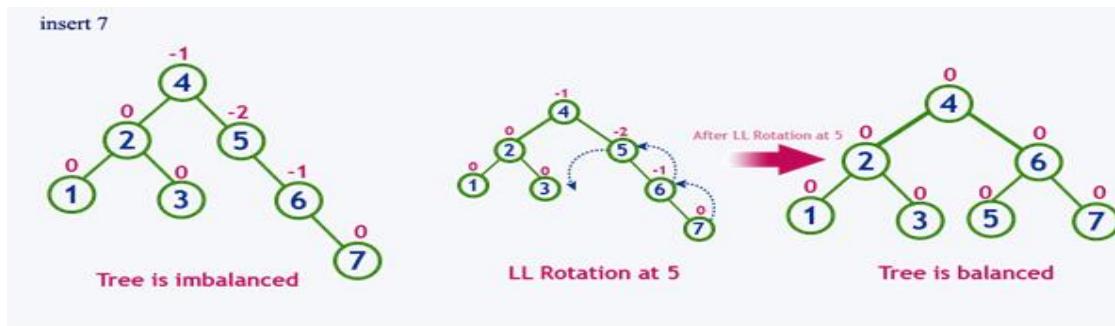
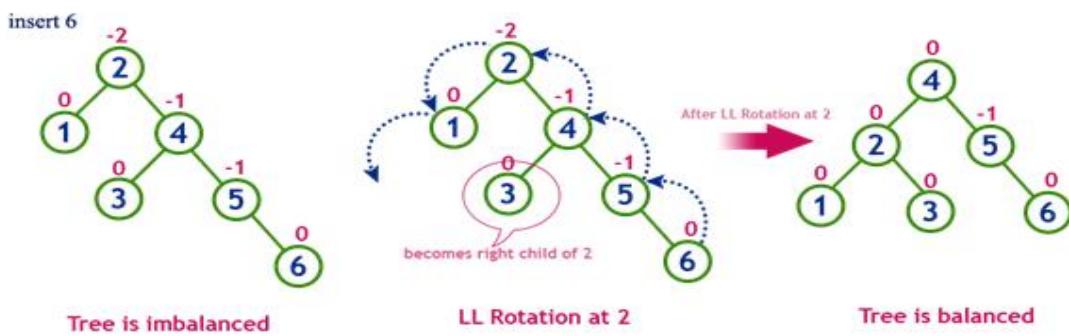
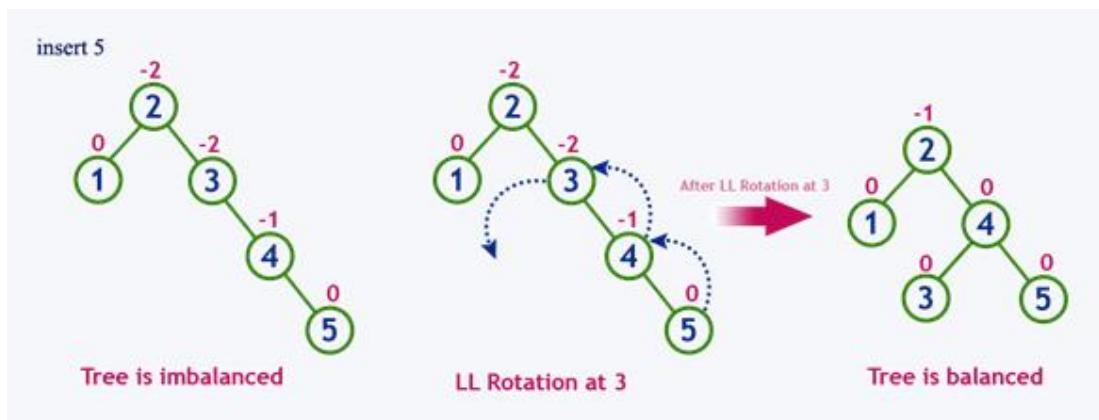
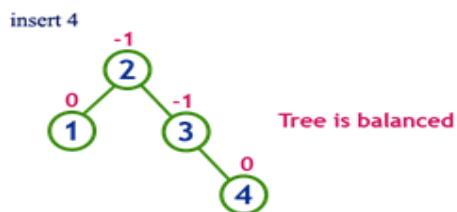
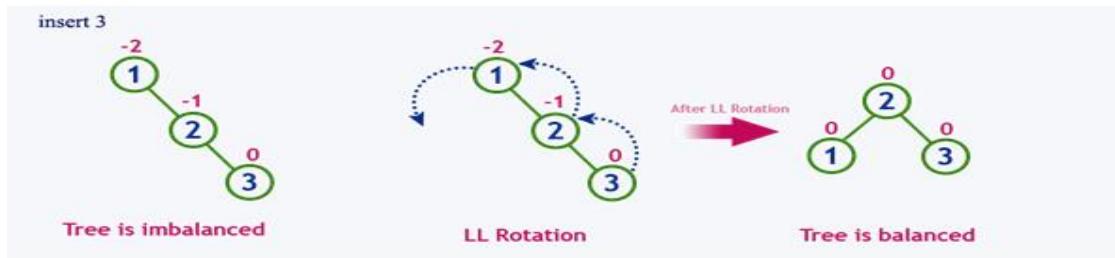
ii) Insertion Operation in AVL Tree

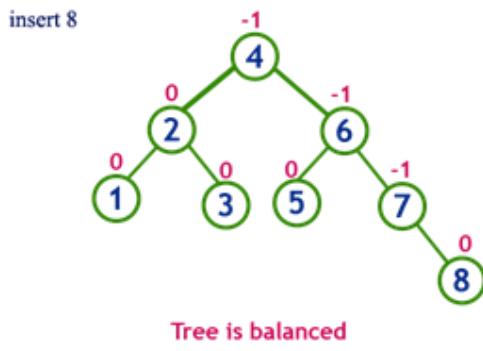
In an AVL tree, the insertion operation is performed with **O(log n)** time complexity. In AVL Tree, a new node is always inserted as a leaf node. The insertion operation is performed as follows...

- **Step 1** - Insert the new element into the tree using Binary Search Tree insertion logic.
- **Step 2** - After insertion, check the **Balance Factor** of every node.
- **Step 3** - If the **Balance Factor** of every node is **0 or 1 or -1** then go for next operation.
- **Step 4** - If the **Balance Factor** of any node is other than **0 or 1 or -1** then that tree is said to be imbalanced. In this case, perform suitable **Rotation** to make it balanced and go for next operation.

Example: Construct an AVL Tree by inserting numbers from 1 to 8.







iii) Deletion Operation in AVL Tree

The deletion operation in AVL Tree is similar to deletion operation in BST. But after every deletion operation, we need to check with the Balance Factor condition. If the tree is balanced after deletion go for next operation otherwise perform suitable rotation to make the tree Balanced.

The two types of rotations are **L rotation** and **R rotation**. Here, we will discuss R rotations. L rotations are the mirror images of them.

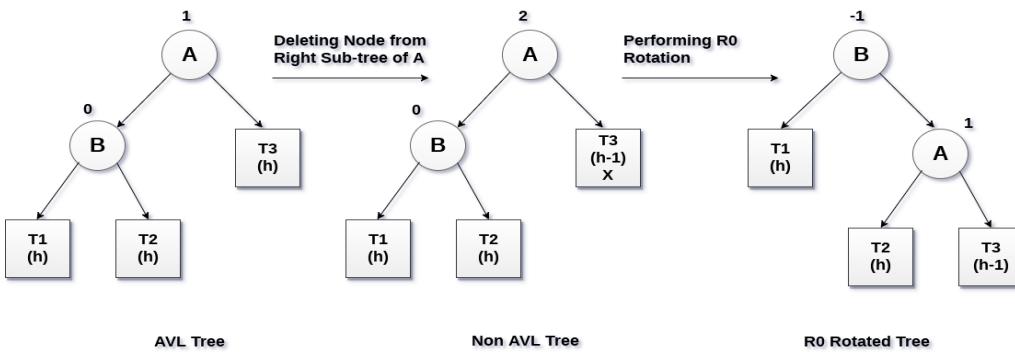
If the node which is to be deleted is present in the left sub-tree of the critical node, then L rotation needs to be applied else if, the node which is to be deleted is present in the right sub-tree of the critical node, the R rotation will be applied.

Let us consider that, A is the critical node and B is the root node of its left sub-tree. If node X, present in the right sub-tree of A, is to be deleted, then there can be three different situations:

a) R0 rotation (Node B has balance factor 0)

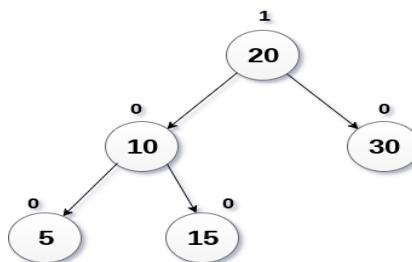
If the node B has 0 balance factor, and the balance factor of node A disturbed upon deleting the node X, then the tree will be rebalanced by rotating tree using R0 rotation.

The critical node A is moved to its right and the node B becomes the root of the tree with T1 as its left sub-tree. The sub-trees T2 and T3 becomes the left and right sub-tree of the node A. the process involved in R0 rotation is shown in the following image.



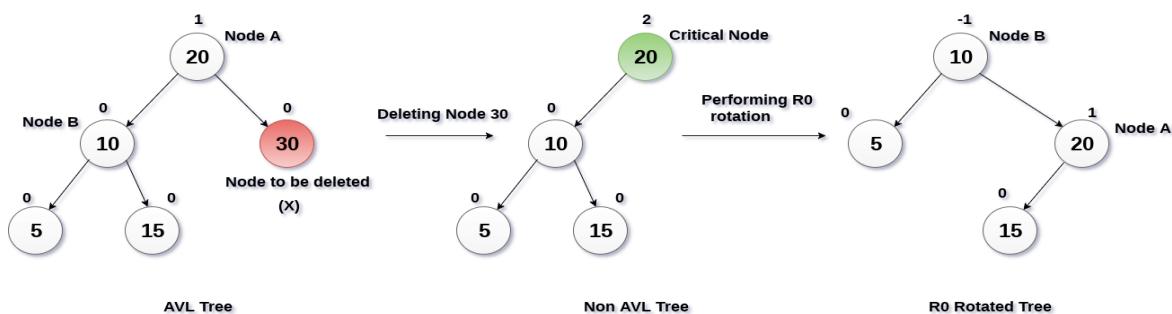
Example:

Delete the node 30 from the AVL tree shown in the following image.



Solution

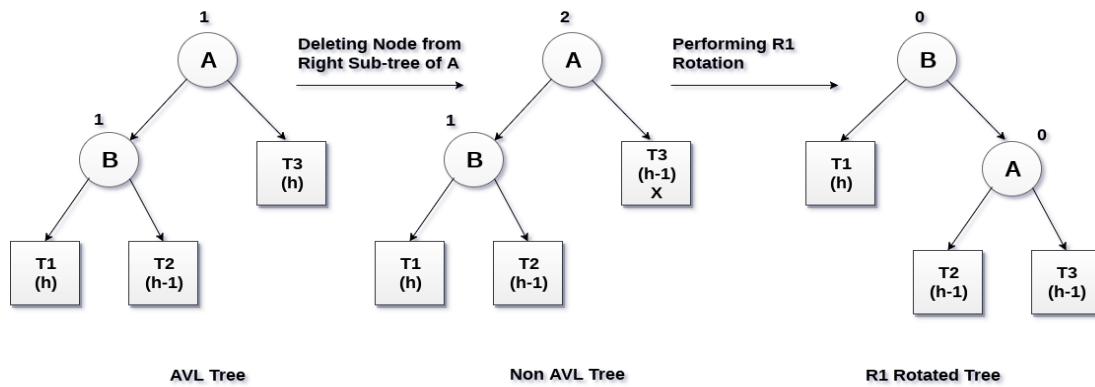
In this case, the node B has balance factor 0, therefore the tree will be rotated by using R0 rotation as shown in the following image. The node B(10) becomes the root, while the node A is moved to its right. The right child of node B will now become the left child of node A.



b) R1 Rotation (Node B has balance factor 1)

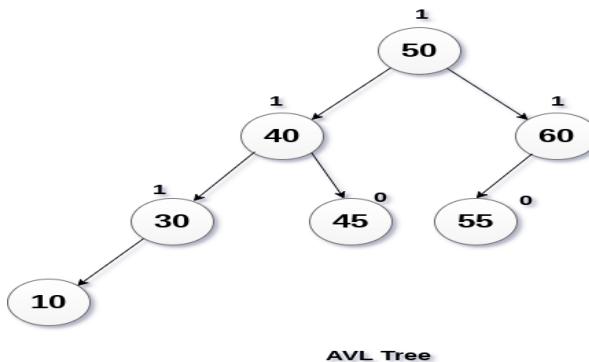
R1 Rotation is to be performed if the balance factor of Node B is 1. In R1 rotation, the critical node A is moved to its right having sub-trees T2 and T3 as its left and right child respectively. T1 is to be placed as the left sub-tree of the node B.

The process involved in R1 rotation is shown in the following image.



Example

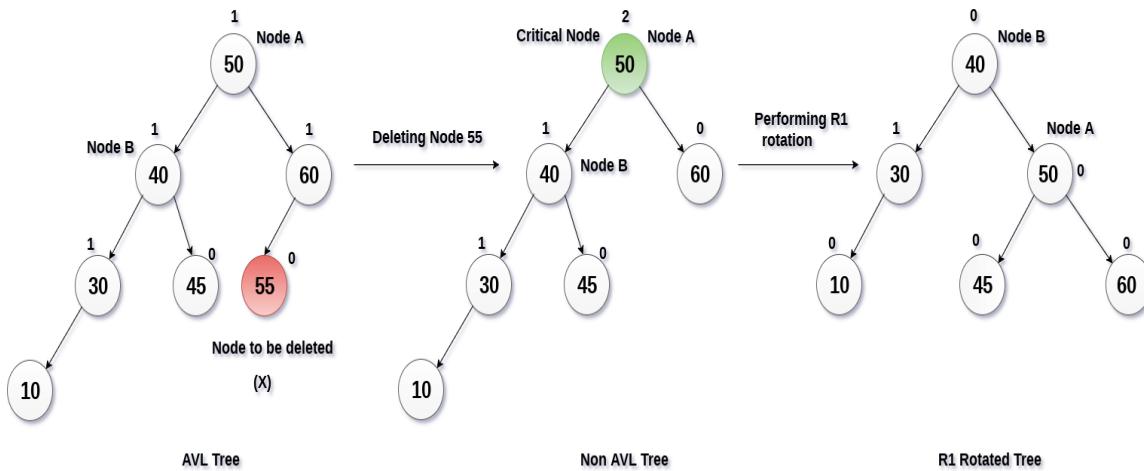
Delete Node 55 from the AVL tree shown in the following image.



Solution :

Deleting 55 from the AVL Tree disturbs the balance factor of the node 50 i.e. node A which becomes the critical node. This is the condition of R1 rotation in which, the node A will be moved to its right (shown in the image below). The right of B is now become the left of A (i.e. 45).

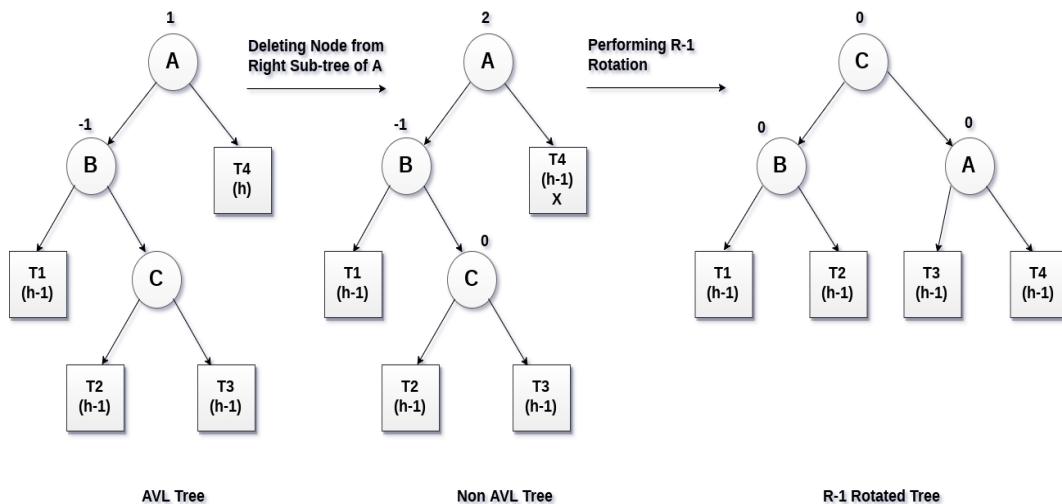
The process involved in the solution is shown in the following image.



c)R-1 Rotation (Node B has balance factor -1)

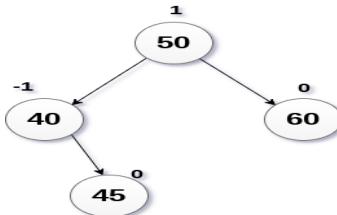
R-1 rotation is to be performed if the node B has balance factor -1. This case is treated in the same way as LR rotation. In this case, the node C, which is the right child of node B, becomes the root node of the tree with B and A as its left and right children respectively.

The sub-trees T1, T2 becomes the left and right sub-trees of B whereas, T3, T4 become the left and right sub-trees of A. The process involved in R-1 rotation is shown in the following image.



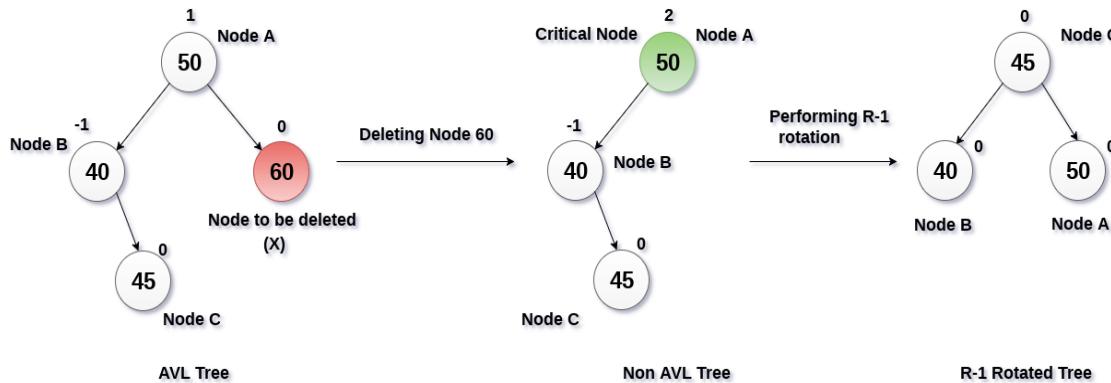
Example

Delete the node 60 from the AVL tree shown in the following image.



Solution:

in this case, node B has balance factor -1. Deleting the node 60, disturbs the balance factor of the node 50 therefore, it needs to be R-1 rotated. The node C i.e. 45 becomes the root of the tree with the node B(40) and A(50) as its left and right child.



2.2.4. Applications of A VL Trees

AVL trees are applied in the following situations:

- There are few insertion and deletion operations
- Short search time is needed
- Input data is sorted or nearly sorted

AVL tree structures can be used in situations which require fast searching. But, the large cost of re balancing may limit the usefulness.

2.3. B Tree

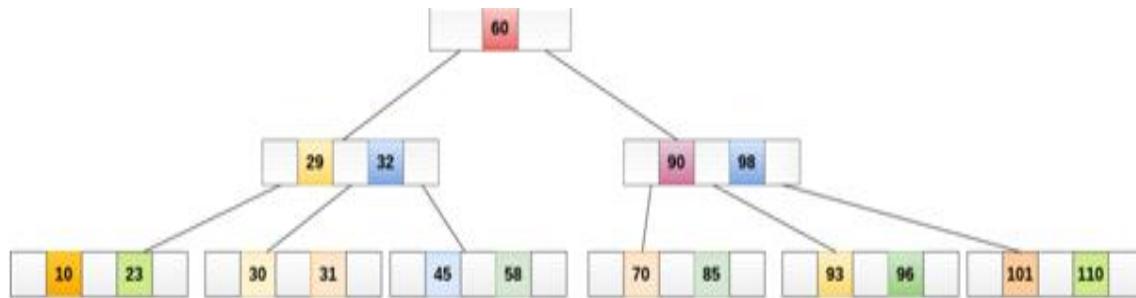
B Tree is a specialized m-way tree that can be widely used for disk access. A B-Tree of order m can have at most $m-1$ keys and m children. One of the main reason of using B tree is its capability to store large number of keys in a single node and large key values by keeping the height of the tree relatively small.

A B tree of order m contains all the properties of an M way tree. In addition, it contains the following properties.

1. Every node in a B-Tree contains at most m children.
2. Every node in a B-Tree except the root node and the leaf node contain at least $m/2$ children.
3. The root nodes must have at least 2 nodes.
4. All leaf nodes must be at the same level.

It is not necessary that, all the nodes contain the same number of children but, each node must have $m/2$ number of nodes.

A B tree of order 4 is shown in the following image.



While performing some operations on B Tree, any property of B Tree may violate such as number of minimum children a node can have. To maintain the properties of B Tree, the tree may split or join.

2.3.1.Operations of B Trees

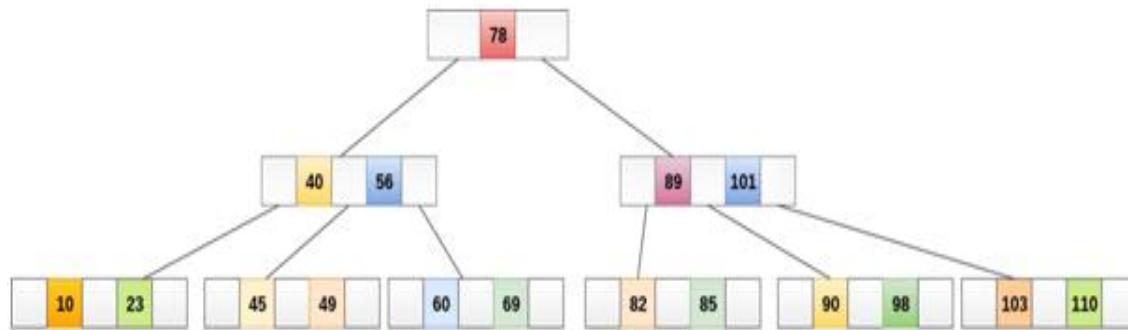
- 1.Searching
- 2.Insertion
- 3.Deletion

i) Searching :

Searching in B Trees is similar to that in Binary search tree. For example, if we search for an item 49 in the following B Tree. The process will something like following :

1. Compare item 49 with root node 78. since $49 < 78$ hence, move to its left sub-tree.
2. Since, $40 < 49 < 56$, traverse right sub-tree of 40.
3. $49 > 45$, move to right. Compare 49.
4. match found, return.

Searching in a B tree depends upon the height of the tree. The search algorithm takes $O(\log n)$ time to search any element in a B tree.



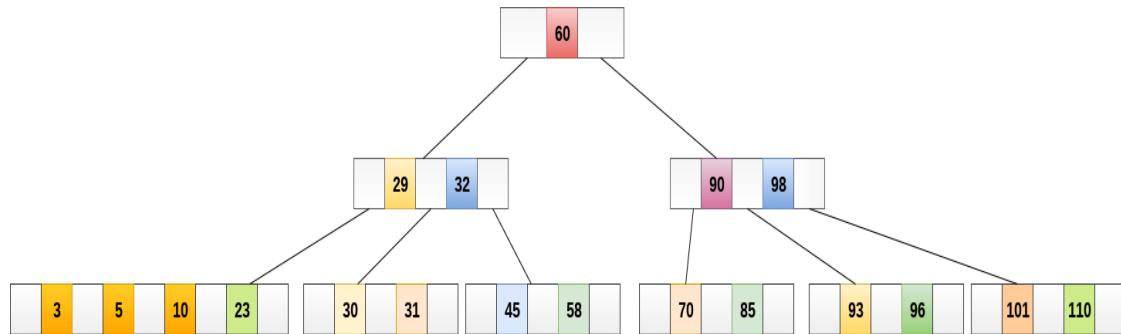
ii) Inserting

Insertions are done at the leaf node level. The following algorithm needs to be followed in order to insert an item into B Tree.

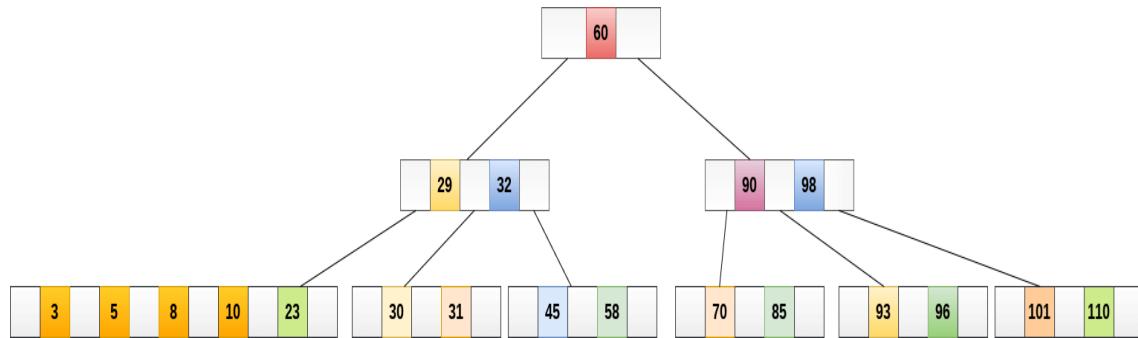
1. Traverse the B Tree in order to find the appropriate leaf node at which the node can be inserted.
2. If the leaf node contain less than $m-1$ keys then insert the element in the increasing order.
3. Else, if the leaf node contains $m-1$ keys, then follow the following steps.
 - o Insert the new element in the increasing order of elements.
 - o Split the node into the two nodes at the median.
 - o Push the median element upto its parent node.
 - o If the parent node also contain $m-1$ number of keys, then split it too by following the same steps.

Example:

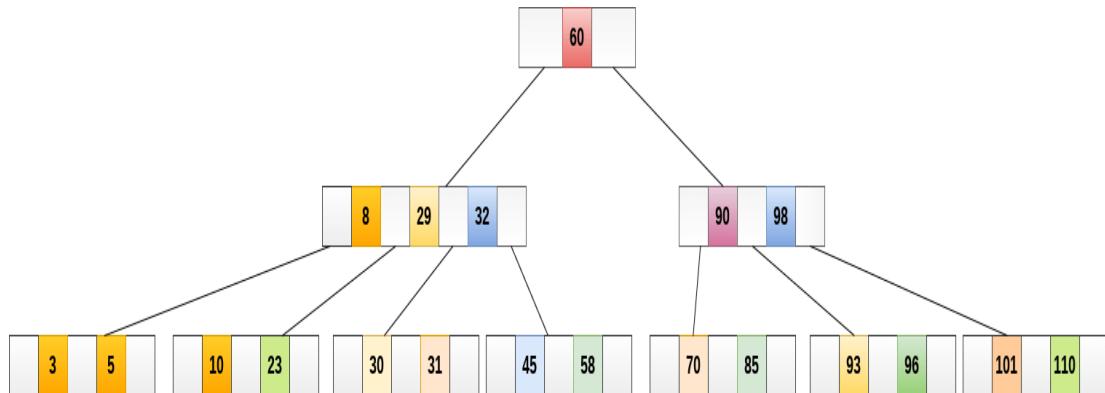
Insert the node 8 into the B Tree of order 5 shown in the following image.



8 will be inserted to the right of 5, therefore insert 8.



The node, now contain 5 keys which is greater than $(5 - 1 = 4)$ keys. Therefore split the node from the median i.e. 8 and push it up to its parent node shown as follows.



iii) Deletion

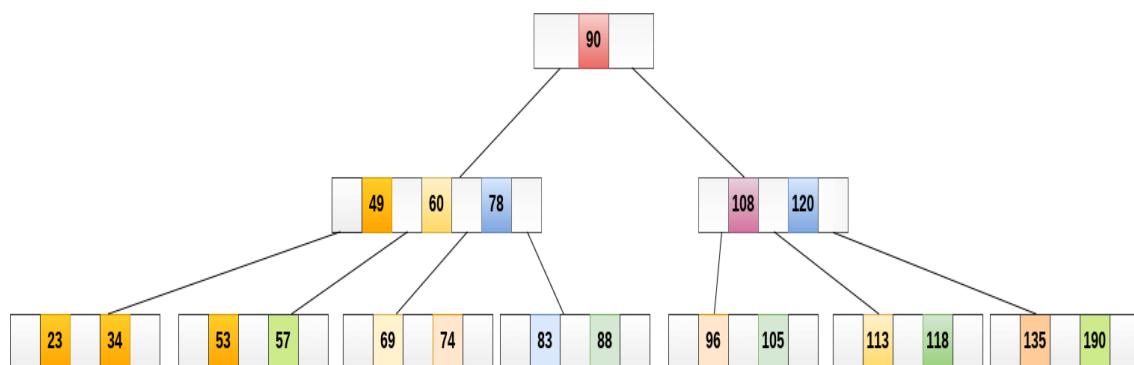
Deletion is also performed at the leaf nodes. The node which is to be deleted can either be a leaf node or an internal node. Following algorithm needs to be followed in order to delete a node from a B tree.

1. Locate the leaf node.
2. If there are more than $m/2$ keys in the leaf node then delete the desired key from the node.
3. If the leaf node doesn't contain $m/2$ keys then complete the keys by taking the element from eight or left sibling.
 - o If the left sibling contains more than $m/2$ elements then push its largest element up to its parent and move the intervening element down to the node where the key is deleted.
 - o If the right sibling contains more than $m/2$ elements then push its smallest element up to the parent and move intervening element down to the node where the key is deleted.
4. If neither of the sibling contain more than $m/2$ elements then create a new leaf node by joining two leaf nodes and the intervening element of the parent node.
5. If parent is left with less than $m/2$ nodes then, apply the above process on the parent too.

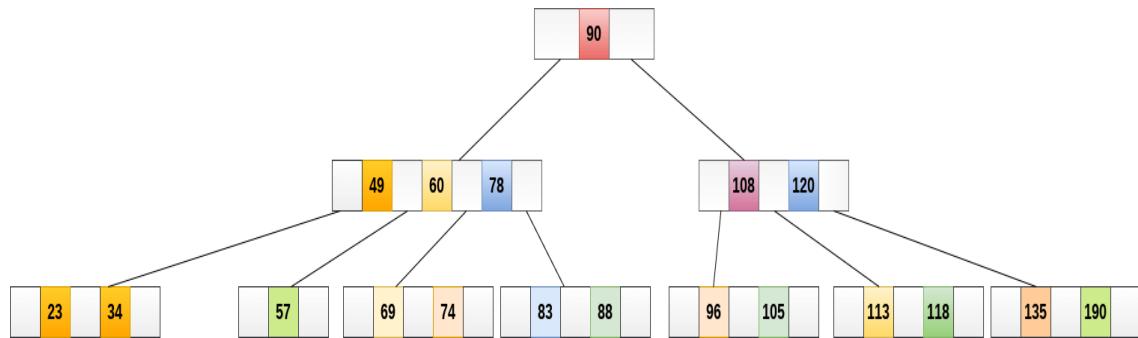
If the the node which is to be deleted is an internal node, then replace the node with its in-order successor or predecessor. Since, successor or predecessor will always be on the leaf node hence, the process will be similar as the node is being deleted from the leaf node.

Example 1

Delete the node 53 from the B Tree of order 5 shown in the following figure.

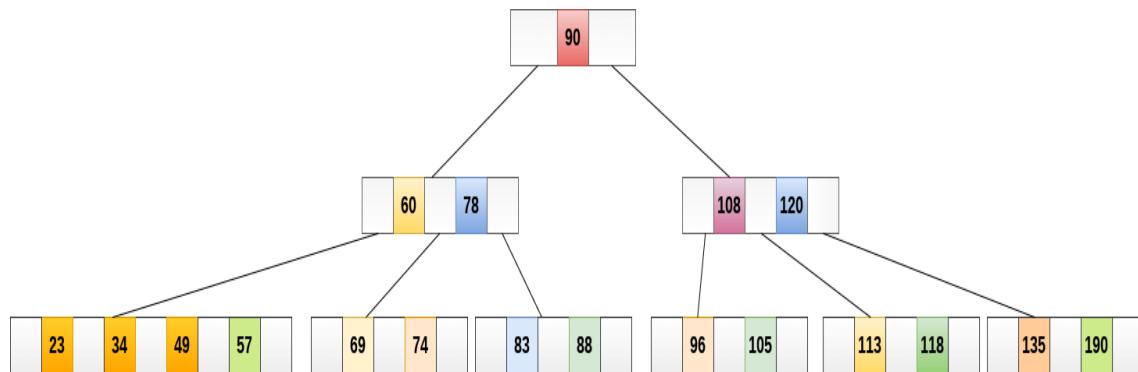


53 is present in the right child of element 49. Delete it.



Now, 57 is the only element which is left in the node, the minimum number of elements that must be present in a B tree of order 5, is 2. it is less than that, the elements in its left and right sub-tree are also not sufficient therefore, merge it with the left sibling and intervening element of parent i.e. 49.

The final B tree is shown as follows.



2.3.2.Application of B tree

B tree is used to index the data and provides fast access to the actual data stored on the disks since, the access to value stored in a large database that is stored on a disk is a very time consuming process.

Searching an un-indexed and unsorted database containing n key values needs $O(n)$ running time in worst case. However, if we use B Tree to index this database, it will be searched in $O(\log n)$ time in worst case.

PART-A (2 Marks)

1. Define a binary tree

Ans : A binary tree is a finite set of nodes which is either empty or consists of a root and two disjoint binary trees called the left sub-tree and right sub-tree.

2. Define a binary search tree

Ans : A binary search tree is a special binary tree, which is either empty or it should satisfy the following characteristics:

Every node has a value and no two nodes should have the same value i.e) the values in the binary search tree are distinct

- The values in any left sub-tree is less than the value of its parent node
- The values in any right sub-tree is greater than the value of its parent node
- The left and right sub-trees of each node are again binary search trees

3. Define AVL Tree.

Ans: AVL stands for Adelson-Velskii and Landis. An AVL tree is a binary search tree which has the following properties:

1. The sub-trees of every node differ in height by at most one.
2. Every sub-tree is an AVL tree.

Search time is $O(\log n)$. Addition and deletion operations also take $O(\log n)$ time.

4. What do you mean by balanced trees?

Ans: Balanced trees have the structure of binary trees and obey binary search tree properties. Apart from these properties, they have some special constraints, which differ from one data structure to another. However, these constraints are aimed only at reducing the height of the tree, because this factor determines the time complexity.

Eg: AVL trees, Splay trees.

5. What are the categories of AVL rotations?

Ans: Let A be the nearest ancestor of the newly inserted node which has the balancing factor ± 2 . Then the rotations can be classified into the following four categories:

Left-Left: The newly inserted node is in the left subtree of the left child of A.

Right-Right: The newly inserted node is in the right subtree of the right child of A.

Left-Right: The newly inserted node is in the right subtree of the left child of A.

Right-Left: The newly inserted node is in the left subtree of the right child of A.

6. What do you mean by balance factor of a node in AVL tree?

Ans: The height of left subtree minus height of right subtree is called balance factor of a node in AVL tree. The balance factor may be either 0 or +1 or -1. The height of an empty tree is -1.

7. List out the steps involved in deleting a node from a binary search tree.

Ans:

- Deleting a node is a leaf node (ie) No children
- Deleting a node with one child.
- Deleting a node with two Childs.

8. What is ‘B’ Tree?

Ans: A B-tree is a tree data structure that keeps data sorted and allows searches, insertions, and deletions in logarithmic amortized time. Unlike self-balancing binary search trees, it is optimized for systems that read and write large blocks of data. It is most commonly used in database and file systems.

PART-B (10 Marks)

1. Explain the AVL tree insertion and deletion with suitable example.
2. Describe the algorithms used to perform single and double rotation on AVL tree.
3. Explain about B+ trees with suitable algorithm.
4. How to insert and delete an element into a binary search tree and write down the code for the insertion routine with an example.
5. What are binary search tree? Write an algorithm for deleting a node in a binary search tree.
6. Create a binary search tree for the following numbers start from an empty binary search tree. 45,26,10,60,70,30,40 Delete keys 10,60 and 45 one after the other and show the trees at each