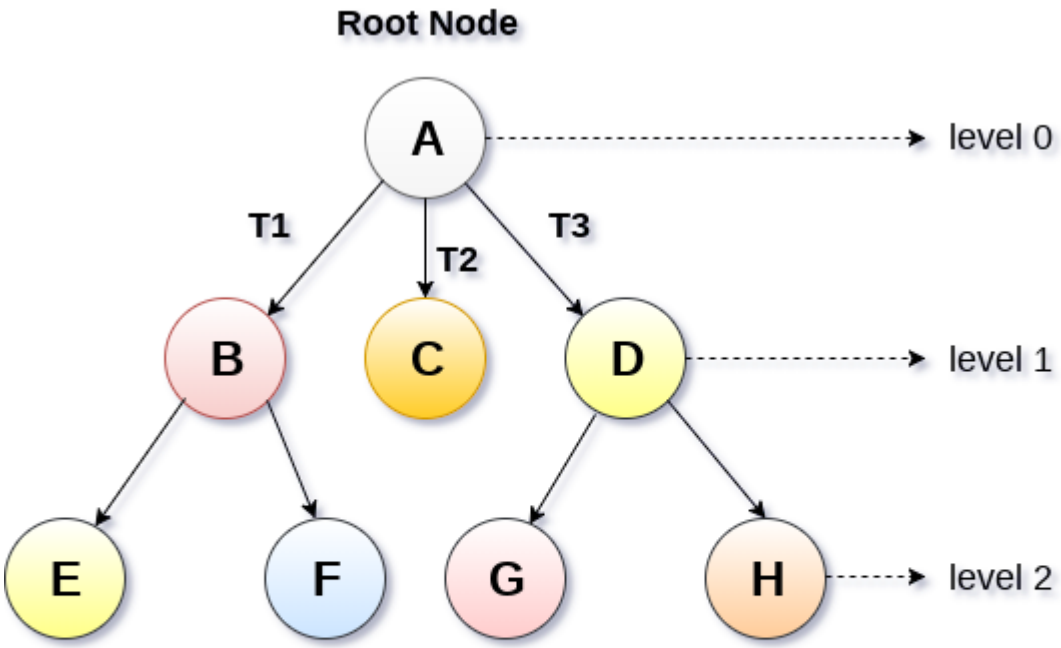
Tree

- A Tree is a recursive data structure containing the set of one or more data nodes where one node is designated as the root of the tree while the remaining nodes are called as the children of the root.
- The nodes other than the root node are partitioned into the non empty sets where each one of them is to be called sub-tree.
- Nodes of a tree either maintain a parent-child relationship between them or they are sister nodes.
- In a general tree, A node can have any number of children nodes but it can have only a single parent.
- The following image shows a tree, where the node A is the root node of the tree while the other nodes can be seen as the children of A.



Tree

Basic terminology

- **Root Node** :- The root node is the topmost node in the tree hierarchy. In other words, the root node is the one which doesn't have any parent.
- **Sub Tree** :- If the root node is not null, the tree T1, T2 and T3 is called sub-trees of the root node.
- **Leaf Node** :- The node of tree, which doesn't have any child node, is called leaf node. Leaf node is the bottom most node of the tree. There can be any number of leaf nodes present in a general tree. Leaf nodes can also be called external nodes.
- **Path** :- The sequence of consecutive edges is called path. In the tree shown in the above image, path to the node E is A→ B → E.
- **Ancestor node** :- An ancestor of a node is any predecessor node on a path from root to that node. The root node doesn't have any ancestors. In the tree shown in the above image, the node F have the ancestors, B and A.
- **Degree** :- Degree of a node is equal to number of children, a node have. In the tree shown in the above image, the degree of node B is 2. Degree of a leaf node is always 0 while in a complete binary tree, degree of each node is equal to 2.
- **Level Number** :- Each node of the tree is assigned a level number in such a way that each node is present at one level higher than its parent. Root node of the tree is always present at level 0.

Binary Tree

Binary tree is a data structure in which each node can have at most 2 children. The node present at the top most level is called the root node. A node with the 0 children is called leaf node. Binary Trees are used in the applications like expression evaluation and many more. We will discuss binary tree in detail, later in this tutorial.

Binary Search Tree

Binary search tree is an ordered binary tree. All the elements in the left sub-tree are less than the root while elements present in the right sub-tree are greater than or equal to the root node element. Binary search trees are used in most of the applications of computer science domain like searching, sorting, etc.

AVL Tree

AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.

B-Tree

B-Tree is a self-balancing search tree

B+ Tree

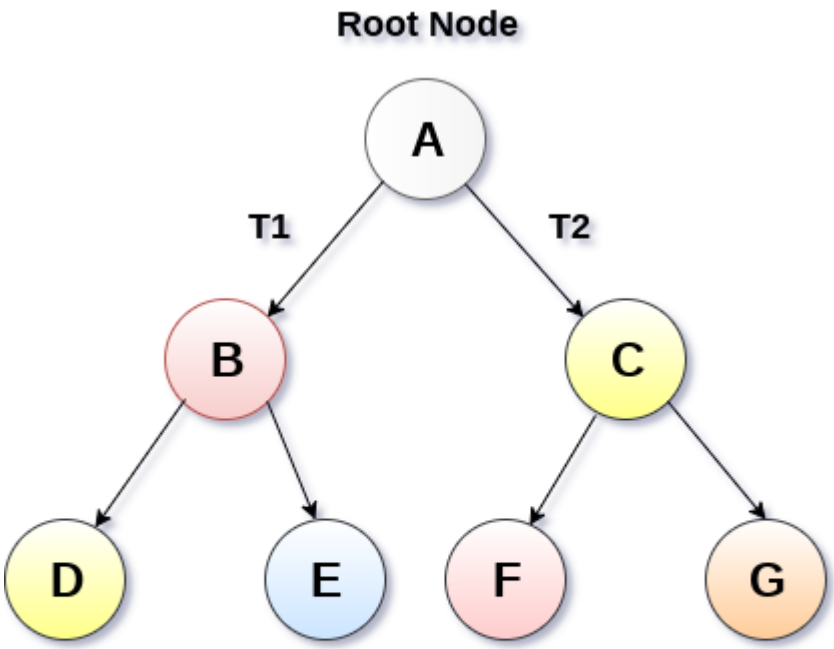
B+ Tree is an extension of B Tree which allows efficient insertion, deletion and search operations.

In B Tree, Keys and records both can be stored in the internal as well as leaf nodes. Whereas, in B+ tree, records (data) can only be stored on the leaf nodes while internal nodes can only store the key values.

Binary Tree

Binary Tree is a special type of generic tree in which, each node can have at most two children. Binary tree is generally partitioned into three disjoint subsets.

1. Root of the node
2. left sub-tree which is also a binary tree.
3. Right binary sub-tree

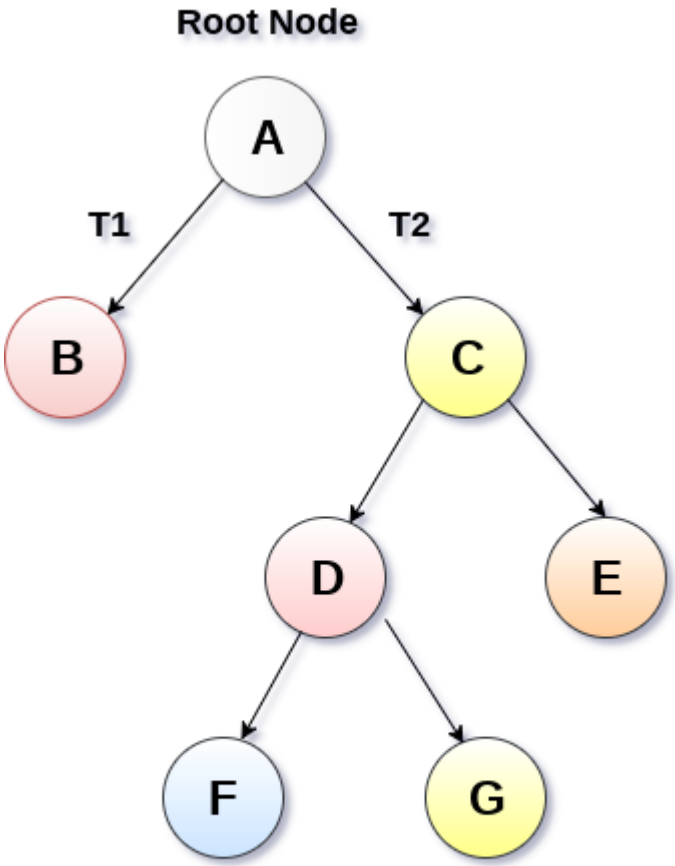


Binary Tree

Types of Binary Tree

1. Strictly Binary Tree

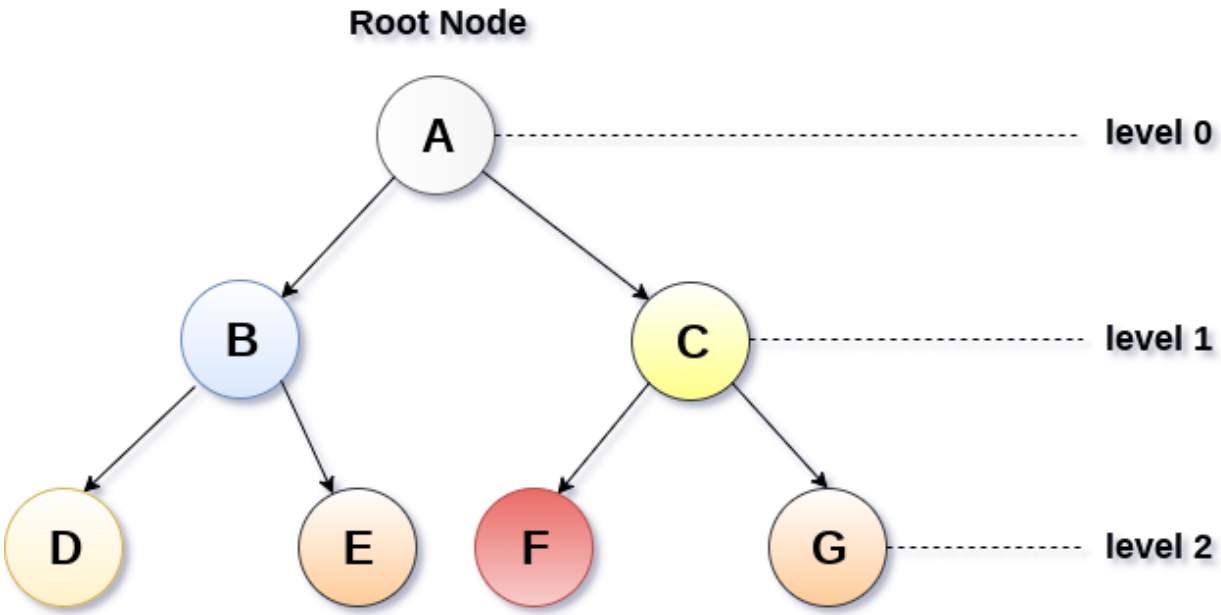
In Strictly Binary Tree, every non-leaf node contain non-empty left and right sub-trees. In other words, the degree of every non-leaf node will always be 2. A strictly binary tree with n leaves, will have (2n - 1) nodes.



Strictly Binary Tree

2.Complete Binary Tree

A Binary Tree is said to be a complete binary tree if all of the leaves are located at the same level d. A complete binary tree is a binary tree that contains exactly 2^l nodes at each level between level 0 and d. The total number of nodes in a complete binary tree with depth d is 2^{d+1}-1 where leaf nodes are 2^{d} while non-leaf nodes are 2^{d}-1.



Complete Binary Tree

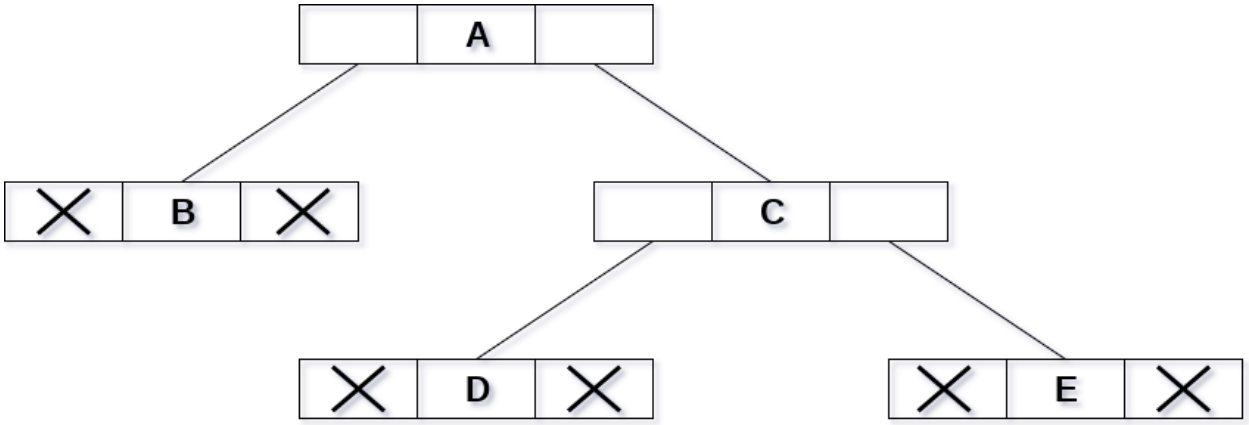
Binary Tree Traversal

SN	Traversal	Description
1	**Pre-order Traversal**	Traverse the root first then traverse into the left sub-tree and right sub-tree respectively. This procedure will be applied to each sub-tree of the tree recursively.
2	**In-order Traversal**	Traverse the left sub-tree first, and then traverse the root and the right sub-tree respectively. This procedure will be applied to each sub-tree of the tree recursively.
3	**Post-order Traversal**	Traverse the left sub-tree and then traverse the right sub-tree and root respectively. This procedure will be applied to each sub-tree of the tree recursively.

Binary Tree representation

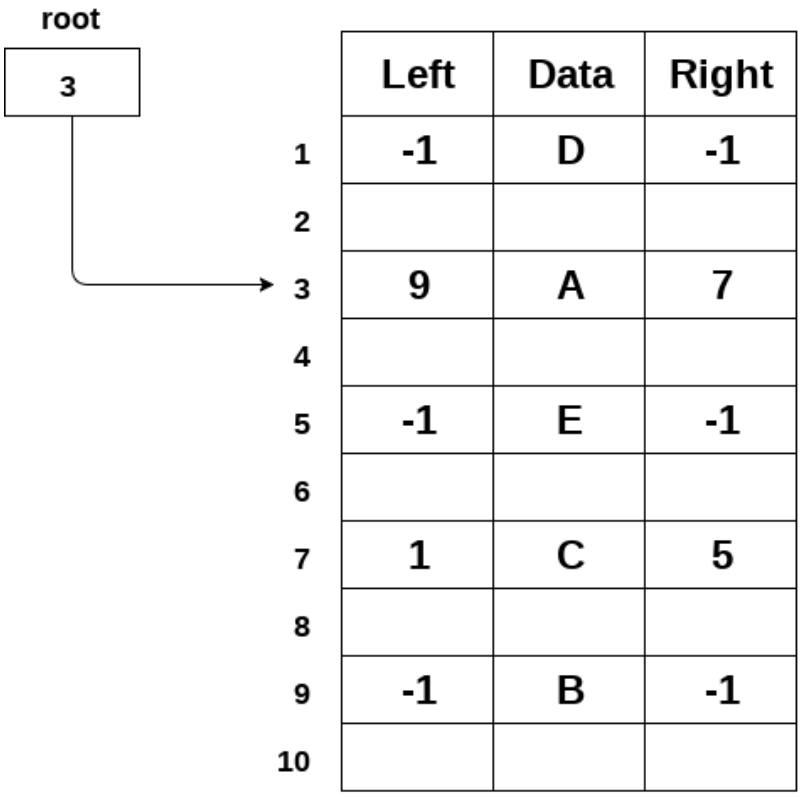
Linked Representation

In this representation, the binary tree is stored in the memory, in the form of a linked list where the number of nodes are stored at non-contiguous memory locations and linked together by inheriting parent child relationship like a tree. every node contains three parts : pointer to the left node, data element and pointer to the right node. Each binary tree has a root pointer which points to the root node of the binary tree. In an empty binary tree, the root pointer will point to null.



In the above figure, a tree is seen as the collection of nodes where each node contains three parts : left pointer, data element and right pointer. Left pointer stores the address of the left child while the right pointer stores the address of the right child. The leaf node contains **null** in its left and right pointers.

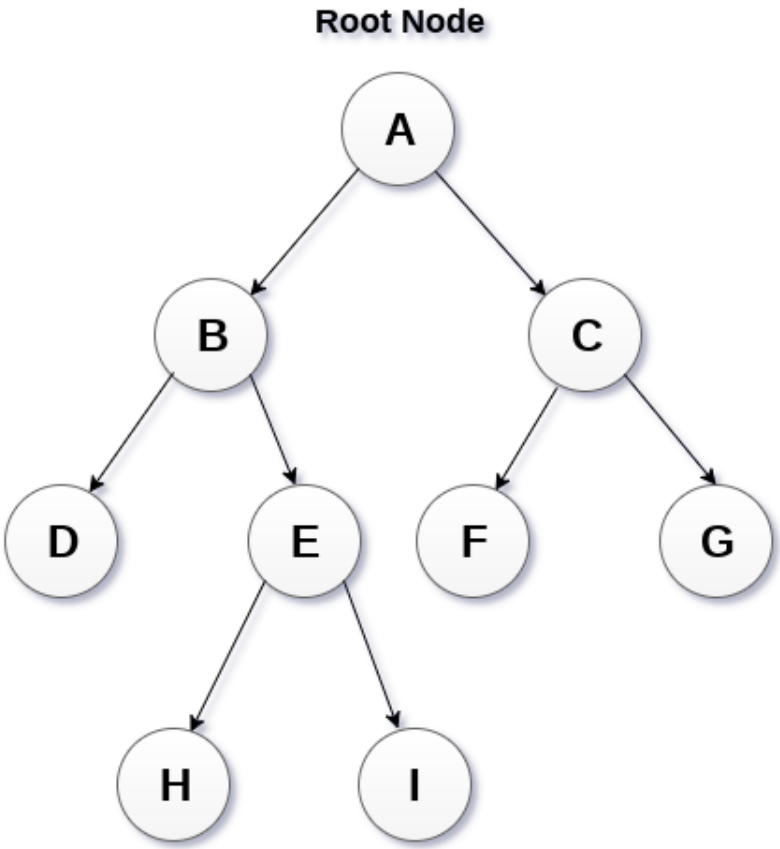
The following image shows about how the memory will be allocated for the binary tree by using linked representation. There is a special pointer maintained in the memory which points to the root node of the tree. Every node in the tree contains the address of its left and right child. Leaf node contains null in its left and right pointers.



Memory Allocation of Binary Tree using linked Representation

Sequential Representation

This is the simplest memory allocation technique to store the tree elements but it is an inefficient technique since it requires a lot of space to store the tree elements. A binary tree is shown in the following figure along with its memory allocation.



A	B	C	D	E	F	G			H	I
1	2	3	4	5	6	7	8	9	10	11

Sequential Representation of Binary Tree

In this representation, an array is used to store the tree elements. Size of the array will be equal to the number of nodes present in the tree. The root node of the tree will be present at the 1st index of the array. If a node is stored at ith index then its left and right children will be stored at 2i and 2i+1 location. If the 1st index of the array i.e. tree[1] is 0, it means that the tree is empty.

Pre-order traversal ROOT->LEFT->RIGHT

Steps

- Visit the root node
- traverse the left sub-tree in pre-order
- traverse the right sub-tree in pre-order

Algorithm

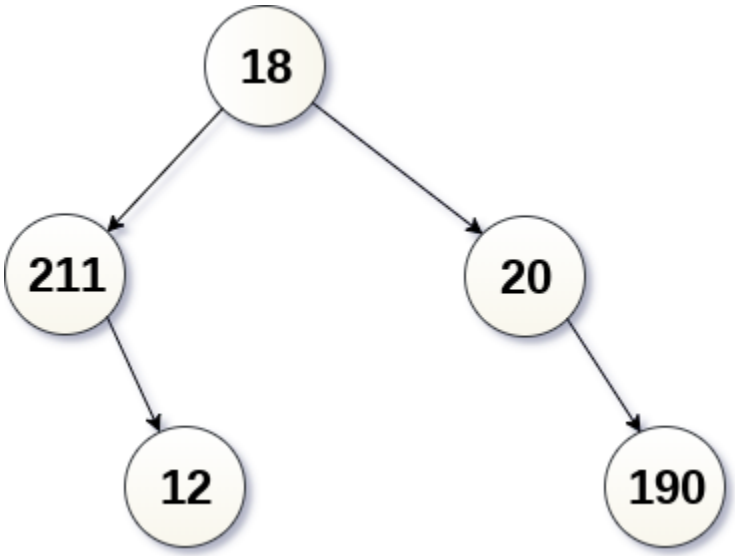
- **Step 1:** Repeat Steps 2 to 4 while TREE != NULL
- **Step 2:** Write TREE -> DATA
- **Step 3:** PREORDER(TREE -> LEFT)
- **Step 4:** PREORDER(TREE -> RIGHT)
  [END OF LOOP]
- **Step 5:** END

C Function

```
1.  void pre-order(struct treenode *tree)
2.      {
3.         if(tree != NULL)
4.         {
5.            printf("%d",tree→ root);
6.            pre-order(tree→ left);
7.            pre-order(tree→ right);
8.         }
9.      }
```

Example

Traverse the following binary tree by using pre-order traversal



- Since, the traversal scheme, we are using is pre-order traversal, therefore, the first element to be printed is 18.
- traverse the left sub-tree recursively. The root node of the left sub-tree is 211, print it and move to left.
- Left is empty therefore print the right children and move to the right sub-tree of the root.
- 20 is the root of sub-tree therefore, print it and move to its left. Since left sub-tree is empty therefore move to the right and print the only element present there i.e. 190.
- Therefore, the printing sequence will be 18, 211, 90, 20, 190.

In-order traversal LEFT->ROOT->RIGHT

Steps

- Traverse the left sub-tree in in-order
- Visit the root
- Traverse the right sub-tree in in-order

Algorithm

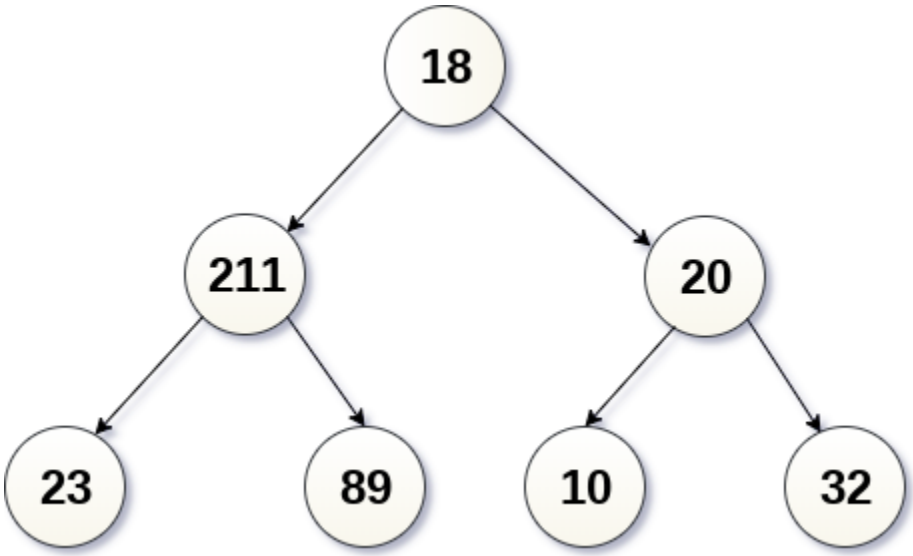
- **Step 1:** Repeat Steps 2 to 4 while TREE != NULL
- **Step 2:** INORDER(TREE -> LEFT)
- **Step 3:** Write TREE -> DATA
- **Step 4:** INORDER(TREE -> RIGHT)
  [END OF LOOP]
- **Step 5:** END

C Function

```
1.  void in-order(struct treenode *tree)
2.     {
3.        if(tree != NULL)
4.        {
5.           in-order(tree→ left);
6.           printf("%d",tree→ root);
7.           in-order(tree→ right);
8.        }
9.     }
```

Example

Traverse the following binary tree by using in-order traversal.



- print the left most node of the left sub-tree i.e. 23.
- print the root of the left sub-tree i.e. 211.
- print the right child i.e. 89.
- print the root node of the tree i.e. 18.
- Then, move to the right sub-tree of the binary tree and print the left most node i.e. 10.
- print the root of the right sub-tree i.e. 20.
- print the right child i.e. 32.
- hence, the printing sequence will be 23, 211, 89, 18, 10, 20, 32.

Post-order traversal LEFT->RIGHT->ROOT

Steps

- Traverse the left sub-tree in post-order
- Traverse the right sub-tree in post-order
- visit the root

Algorithm

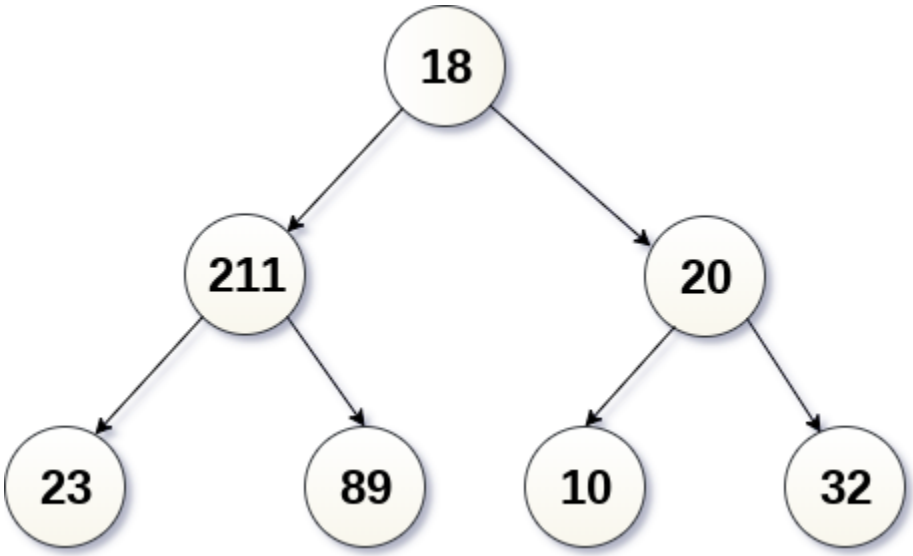
- **Step 1: Repeat Steps 2 to 4 while TREE != NULL**
- **Step 2: POSTORDER(TREE -> LEFT)**
- **Step 3: POSTORDER(TREE -> RIGHT)**
- **Step 4: Write TREE -> DATA**
  [END OF LOOP]
- **Step 5: END**

C Function

```
1.  void post-order(struct treenode *tree)
2.     {
3.        if(tree != NULL)
4.        {
5.           post-order(tree→ left);
6.           post-order(tree→ right);
7.           printf("%d",tree→ root);
8.        }
9.     }
```

Example

Traverse the following tree by using post-order traversal

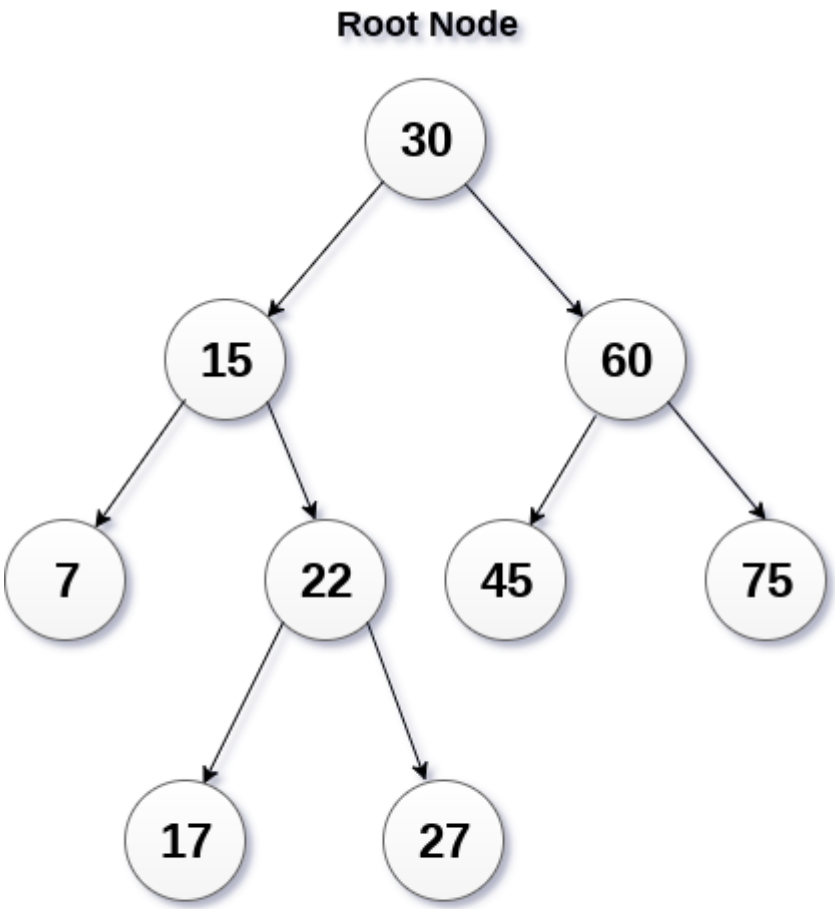


- **Print the left child of the left sub-tree of binary tree i.e. 23.**
- **print the right child of the left sub-tree of binary tree i.e. 89.**
- **print the root node of the left sub-tree i.e. 211.**
- **Now, before printing the root node, move to right sub-tree and print the left child i.e. 10.**
- **print 32 i.e. right child.**
- **Print the root node 20.**
- **Now, at the last, print the root of the tree i.e. 18.**

The printing sequence will be 23, 89, 211, 10, 32, 18.

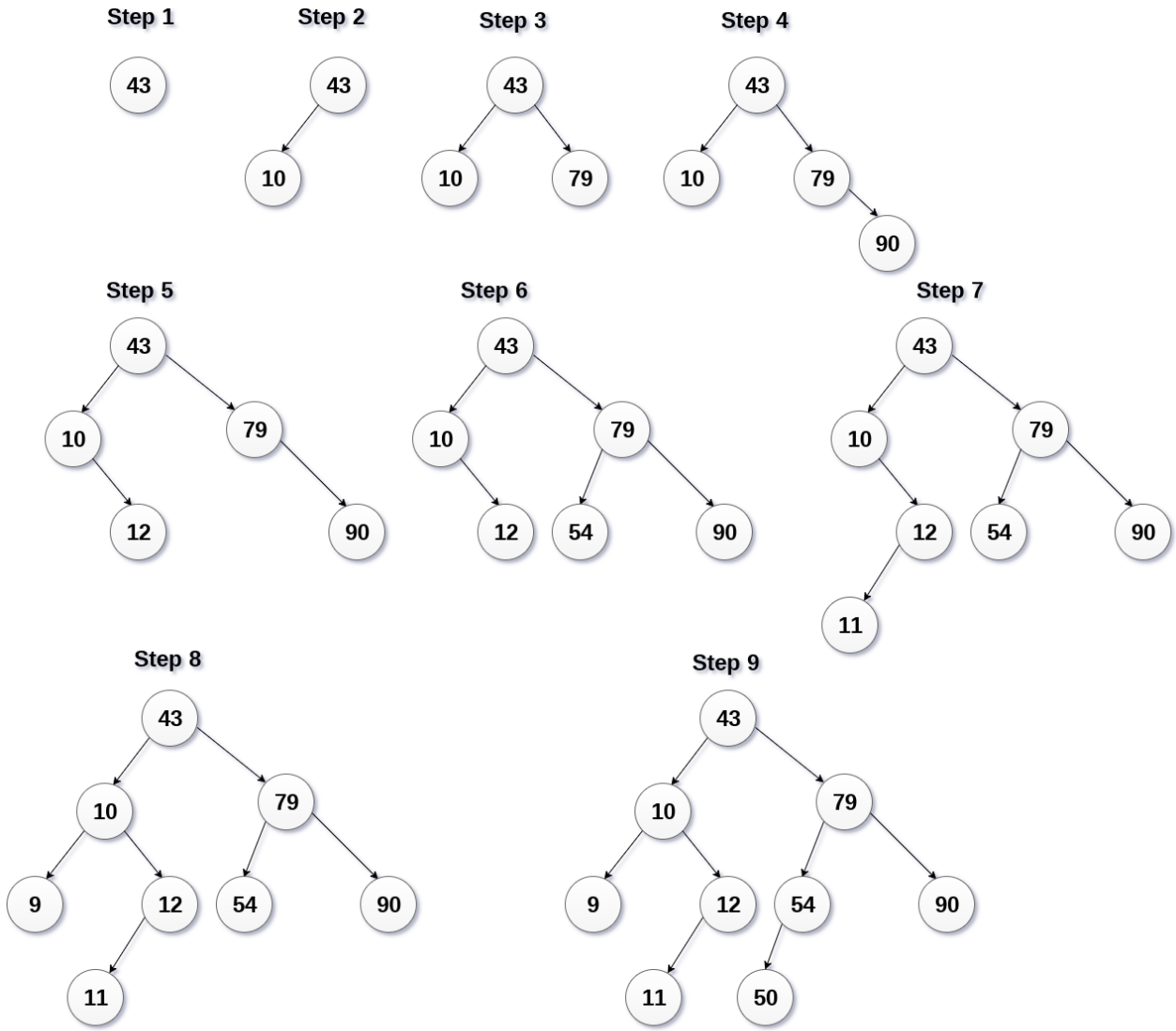
Binary Search Tree

1. Binary Search tree can be defined as a class of binary trees, in which the nodes are arranged in a specific order. This is also called ordered binary tree.
2. In a binary search tree, the value of all the nodes in the left sub-tree is less than the value of the root.
3. Similarly, value of all the nodes in the right sub-tree is greater than or equal to the value of the root.
4. This rule will be recursively applied to all the left and right sub-trees of the root.



Binary Search Tree

A Binary search tree is shown in the above figure. As the constraint applied on the BST, we can see that the root node 30 doesn't contain any value greater than or equal to 30 in its left sub-tree and it also doesn't contain any value less than 30 in its right sub-tree.



Binary search Tree Creation

Operations on Binary Search Tree

There are many operations which can be performed on a binary search tree.

SN	Operation	Description
1	Searching in BST	Finding the location of some specific element in a binary search tree.
2	Insertion in BST	Adding a new element to the binary search tree at the appropriate location so that the property of BST do not violate.
3	Deletion in BST	Deleting some specific node from a binary search tree. However, there can be various cases in deletion depending upon the number of children, the node have.

Insertion

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 node violate the property of binary search tree at each value.

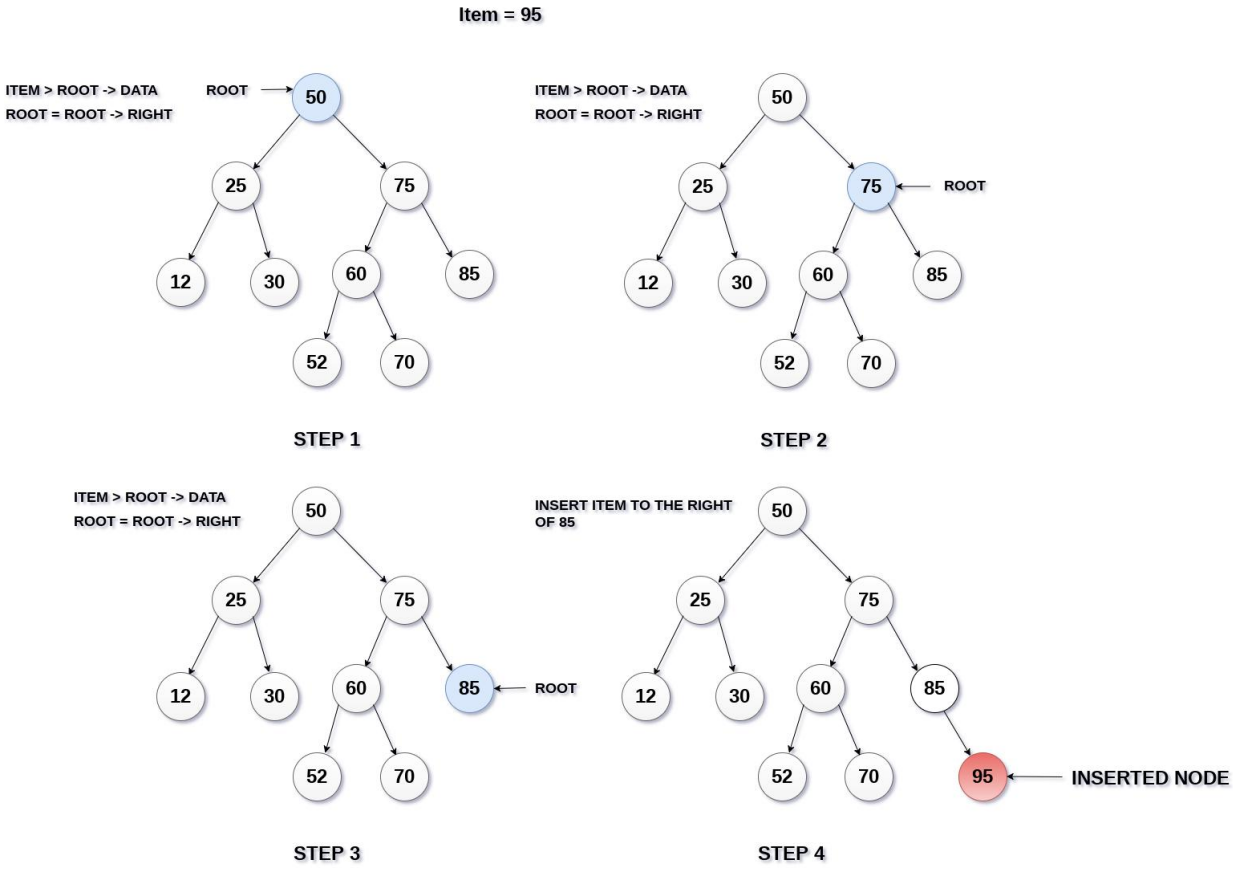
1. Allocate the memory for tree.
2. Set the data part to the value and set the left and right pointer of tree, point to NULL.
3. If the item to be inserted, will be the first element of the tree, then the left and right of this node will point to NULL.
4. Else, check if the item is less than the root 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 -> DATA = ITEM
  SET TREE -> LEFT = TREE -> RIGHT = NULL
  ELSE
  IF ITEM < TREE -> DATA
  Insert(TREE -> LEFT, ITEM)

ELSE
Insert(TREE -> RIGHT, ITEM)
[END OF IF]
[END OF IF]

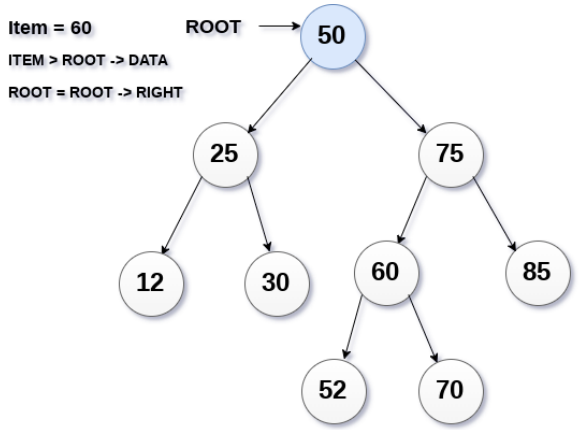
- **Step 2:** END



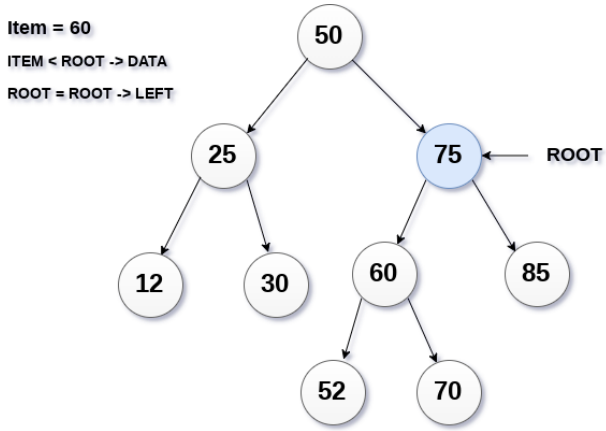
Searching

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.

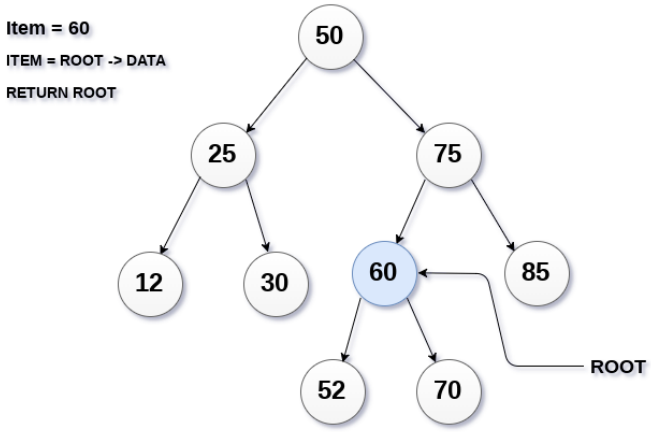
1. Compare the element with the root of the tree.
2. If the item is matched then return the location of the node.
3. Otherwise check if item is less than the element present on root, if so then move to the left sub-tree.
4. If not, then move to the right sub-tree.
5. Repeat this procedure recursively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



STEP 1



STEP 2



STEP 3

Algorithm:

Search (ROOT, ITEM)

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

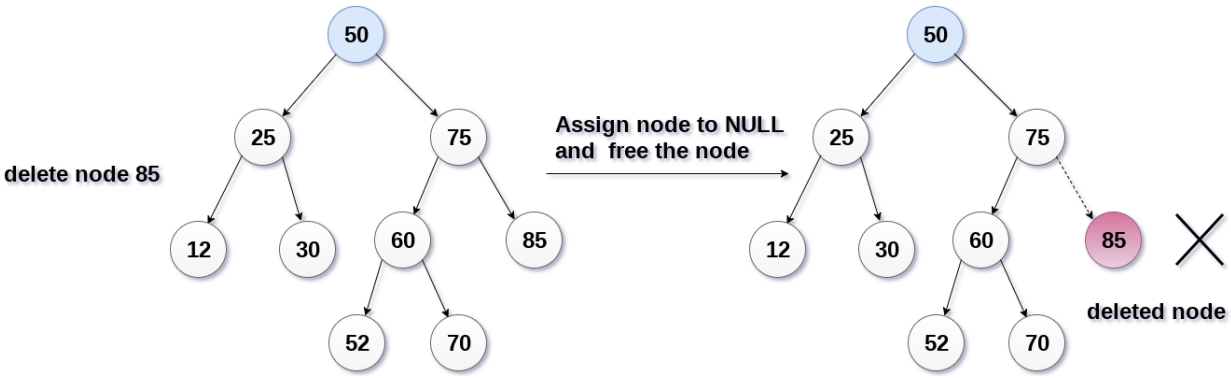
Deletion

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.

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.

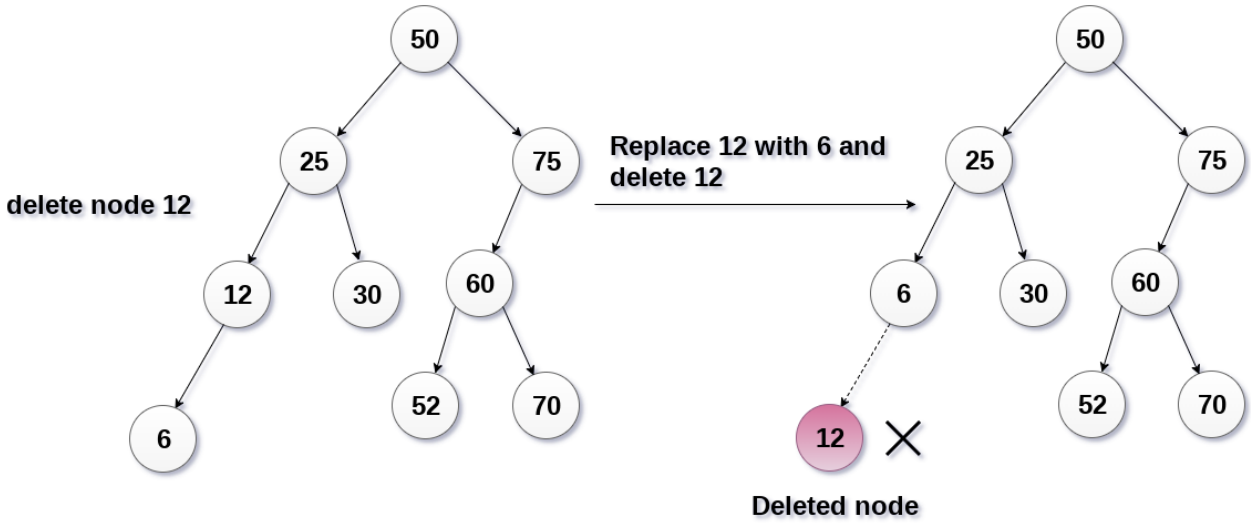
In the following image, we are deleting the node 85, since the node is a leaf node, therefore the node will be replaced with NULL and allocated space will be freed.



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.



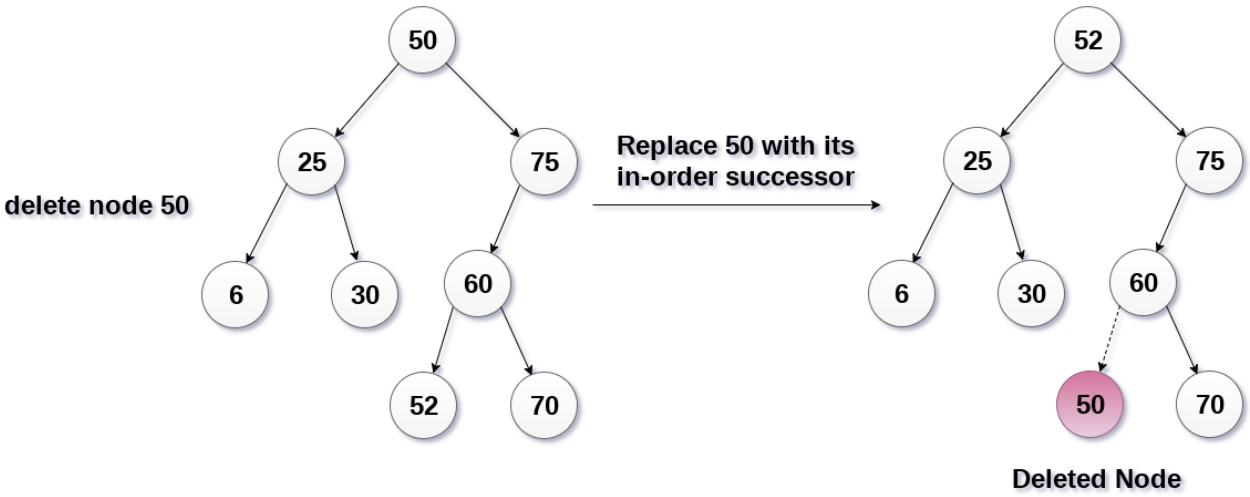
The node to be deleted has two children.

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)

- **Step 1:** 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

AVL Tree

AVL Tree is invented by GM Adelson - Velsky and EM Landis in 1962. The tree is named AVL in honour of its inventors.

AVL Tree can be defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree.

Tree is said to be balanced if balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.

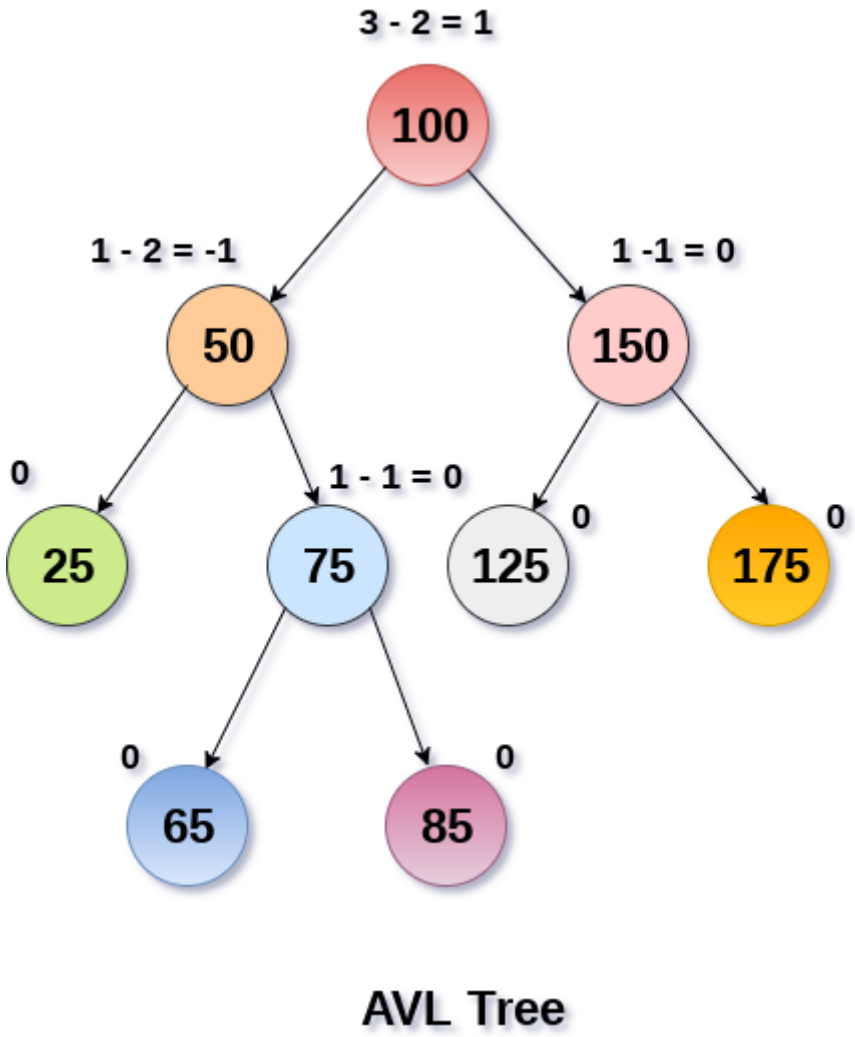
Balance Factor (k) = height (left(k)) - height (right(k))

If balance factor of any node is 1, it means that the left sub-tree is one level higher than the right sub-tree.

If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height.

If balance factor of any node is -1, it means that the left sub-tree is one level lower than the right sub-tree.

An AVL tree is given in the following figure. We can see that, balance factor associated with each node is in between -1 and +1. therefore, it is an example of AVL tree.



AVL Tree

Operations on AVL tree

Due to the fact that, AVL tree is also a binary search tree therefore, all the operations are performed in the same way as they are performed in a binary search tree. Searching and traversing do not lead to the violation in property of AVL tree. However, insertion and deletion are the operations which can violate this property and therefore, they need to be revisited.

SN	Operation	Description
1	Insertion	Insertion in AVL tree is performed in the same way as it is performed in a binary search tree. However, it may lead to violation in the AVL tree property and therefore the tree may need balancing. The tree can be balanced by applying rotations.
2	Deletion	Deletion can also be performed in the same way as it is performed in a binary search tree. Deletion may also disturb the balance of the tree therefore, various types of rotations are used to rebalance the tree.

Insertion

Insertion in AVL tree is performed in the same way as it is performed in a binary search tree. The new node is added into AVL tree as the leaf node. However, it may lead to violation in the AVL tree property and therefore the tree may need balancing.

The tree can be balanced by applying rotations. Rotation is required only if, the balance factor of any node is disturbed upon inserting the new node, otherwise the rotation is not required.

Depending upon the type of insertion, the Rotations are categorized into four categories.

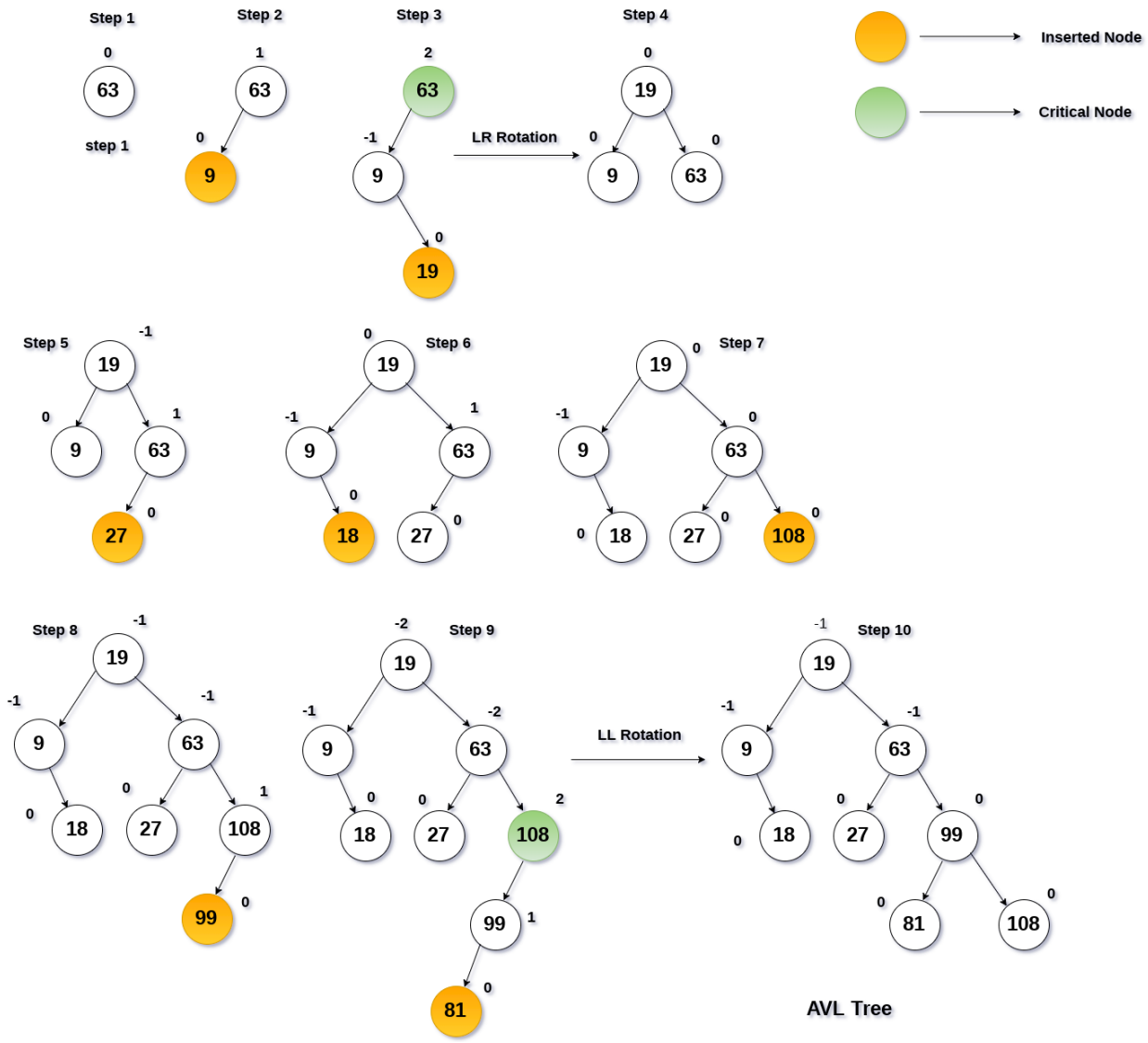
SN	Rotation	Description
1	LL Rotation	The new node is inserted to the left sub-tree of left sub-tree of critical node.
2	RR Rotation	The new node is inserted to the right sub-tree of the right sub-tree of the critical node.
3	LR Rotation	The new node is inserted to the right sub-tree of the left sub-tree of the critical node.
4	RL Rotation	The new node is inserted to the left sub-tree of the right sub-tree of the critical node.

Construct an AVL tree by inserting the following elements in the given order.

63, 9, 19, 27, 18, 108, 99, 81

The process of constructing an AVL tree from the given set of elements is shown in the following figure.

At each step, we must calculate the balance factor for every node, if it is found to be more than 2 or less than -2, then we need a rotation to rebalance the tree. The type of rotation will be estimated by the location of the inserted element with respect to the critical node.



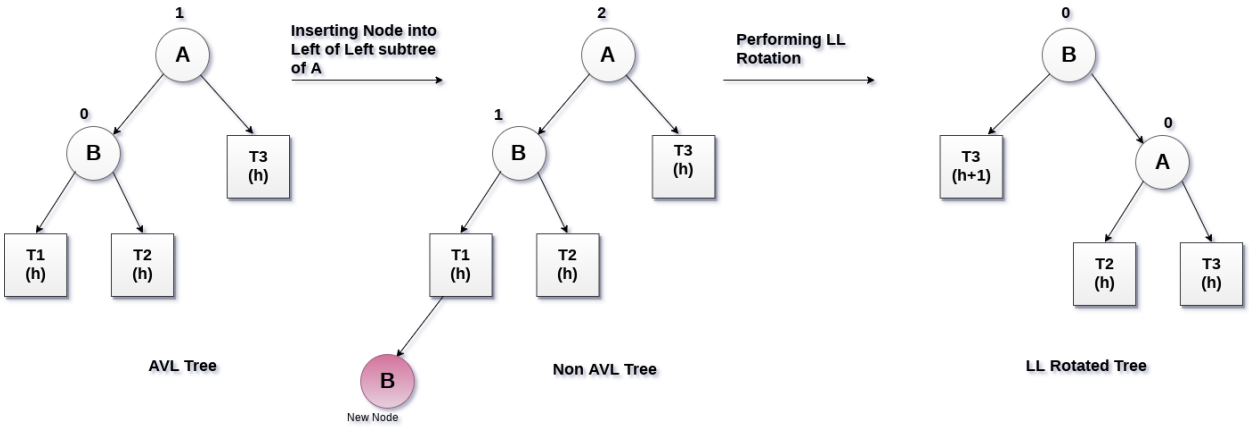
LL Rotation

The tree shown in following figure is an AVL Tree, however, we,need to insert an element into the left of the left sub-tree of A. the tree can become unbalanced with the presence of the critical node A.

The node whose balance factor doesn't lie between -1 and 1, is called critical node.

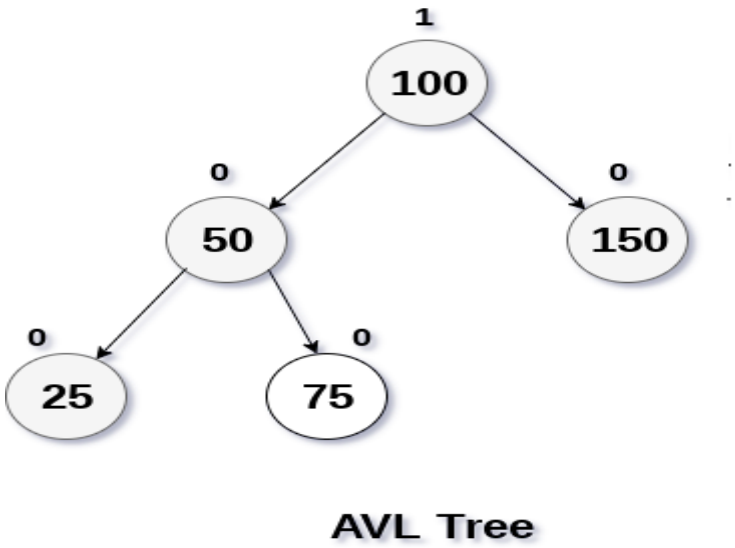
In order to rebalance the tree, LL rotation is performed as shown in the following diagram.

The node B becomes the root, with A and T3 as its left and right child. T1 and T2 becomes the left and right sub-trees of A



Example :

Insert the node with value 12 into the tree shown in the following figure.



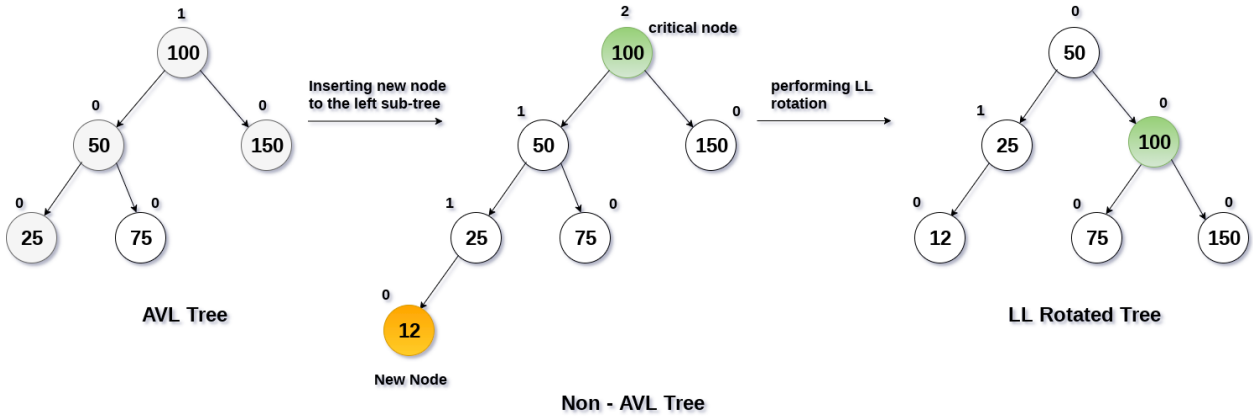
Solution:

12 will be inserted to the left of 25 and therefore, it disturbs the AVLness of the tree. The tree needs to be rebalanced by rotating it through LL rotation.

Here, the critical node 100 will be moved to its right, and the root of its left sub-tree (B) will be the new root node of the tree.

The right sub-tree of B i.e. T2 (with root node 75) will be place to the left of Node A (with value 100).

By following this procedure, the tree will be rebalanced and therefore, it will be an AVL tree produced after inserting 12 into it.

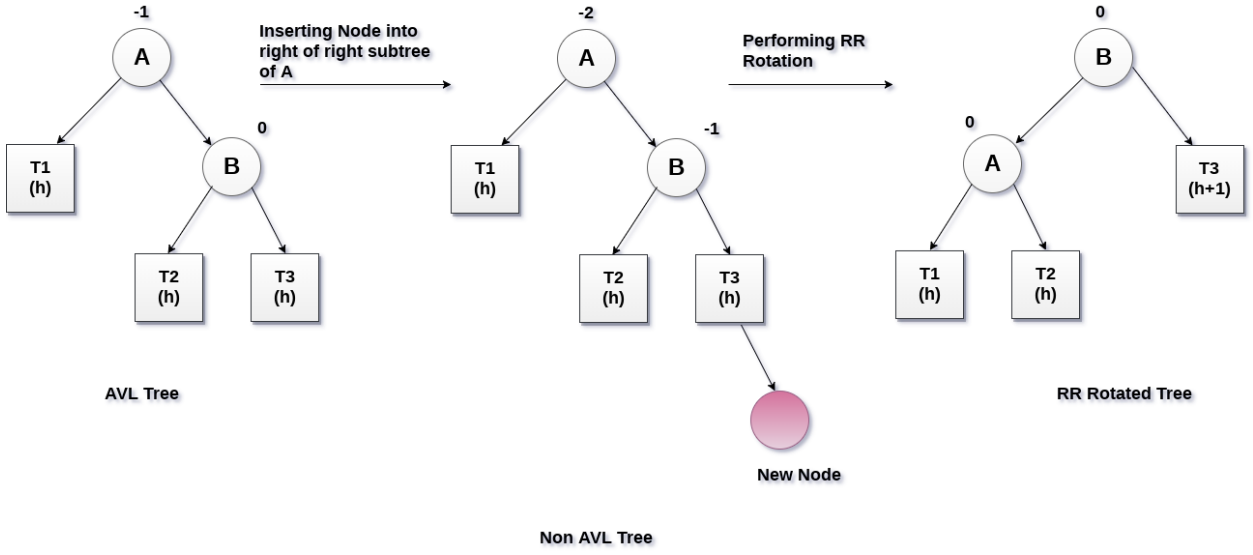


RR Rotation

If the node is inserted into the right of the right sub-tree of a node A and the tree becomes unbalanced then, in that case, RR rotation will be performed as shown in the following diagram.

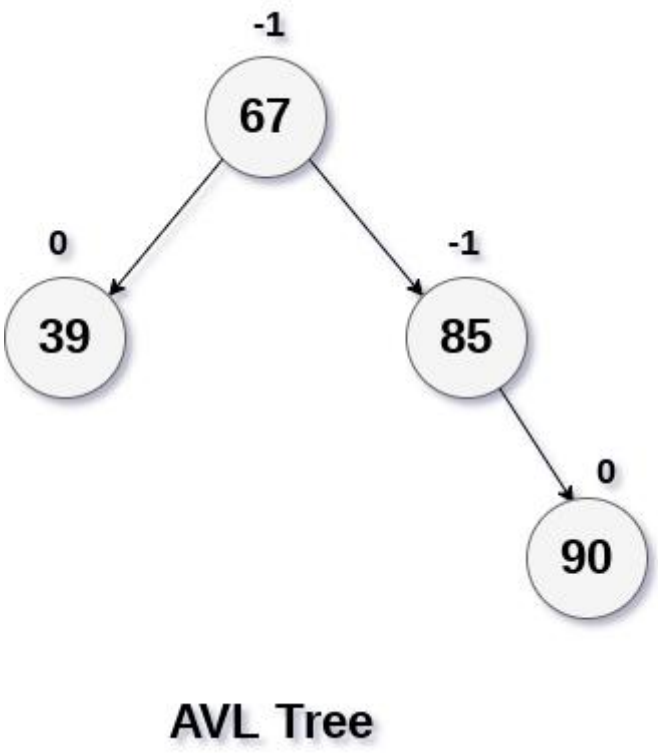
While the rotation, the node B becomes the root node of the tree. The critical node A will be moved to its left and becomes the left child of B.

The sub-tree T3 becomes the right sub-tree of A. T1 and T2 becomes the left and right sub-tree of node A.



Example

Insert 90 into the AVL Tree shown in the figure.

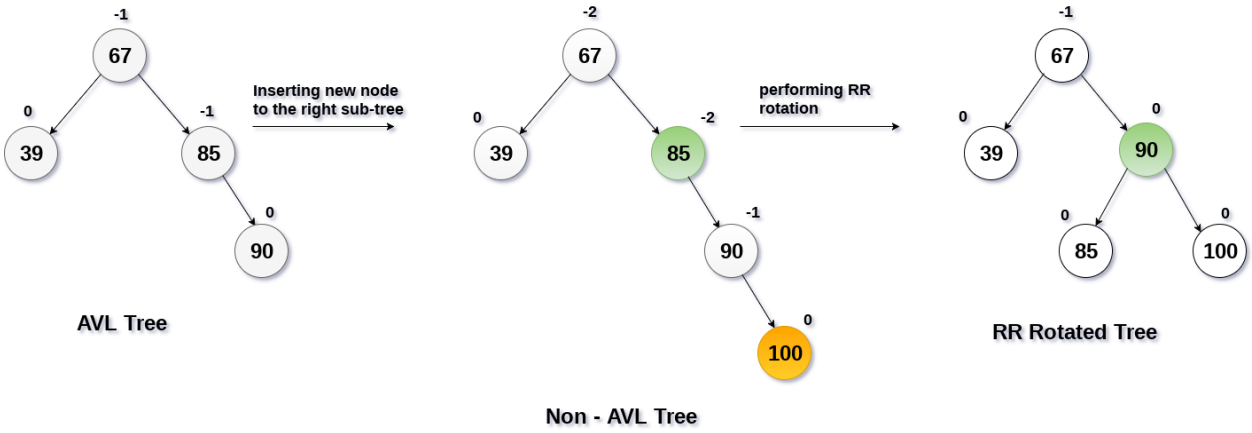


AVL Tree

Solution :

90 is inserted in to the right of the right sub-tree. In this case, critical node A will be 85, which is the closest ancestor to the new node, whose balance factor is disturbed. Therefore, we need to rebalance the tree by applying RR rotation onto it.

The node B will be the node 90 , which will become the root node of this sub-tree. The critical node 85 will become its left child, in order to produce the rebalanced tree which is now an AVL tree.

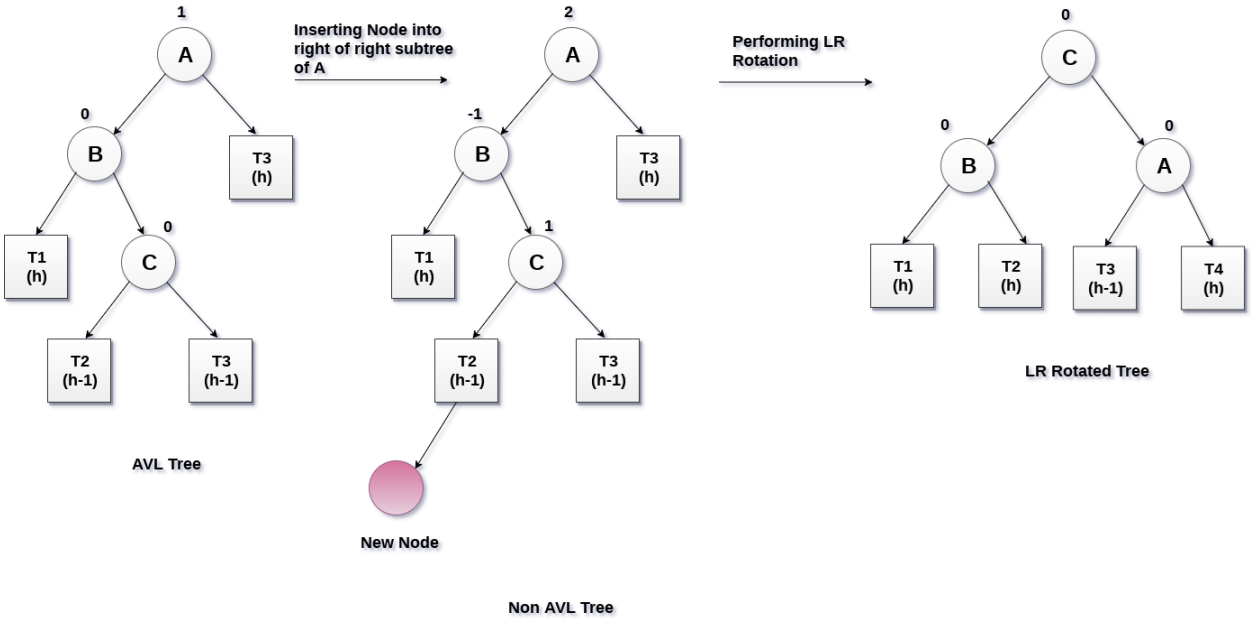


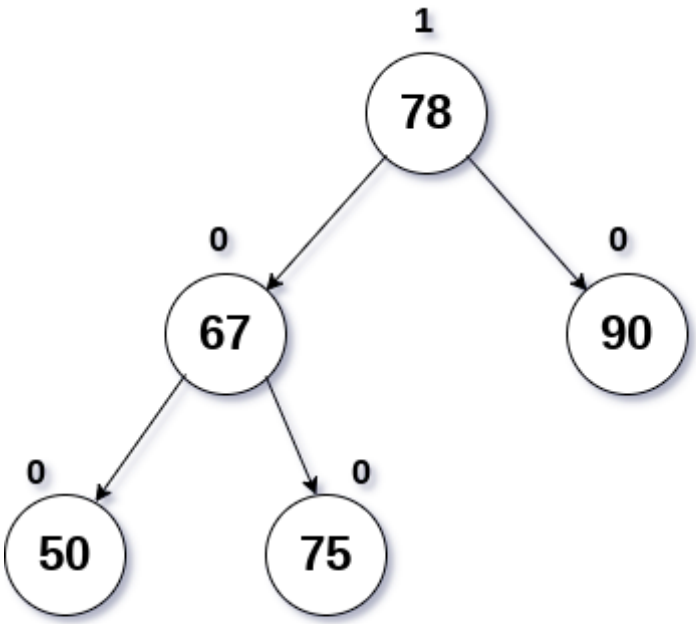
LR Rotation

LR rotation is to be performed if the new node is inserted into the right of the left sub-tree of node A.

In LR rotation, node C (as shown in the figure) becomes the root node of the tree, while the node B and A becomes its left and right child respectively.

T1 and T2 becomes the left and right sub-tree of Node B respectively whereas, T3 and T4 becomes the left and right sub-tree of Node A.





AVL Tree

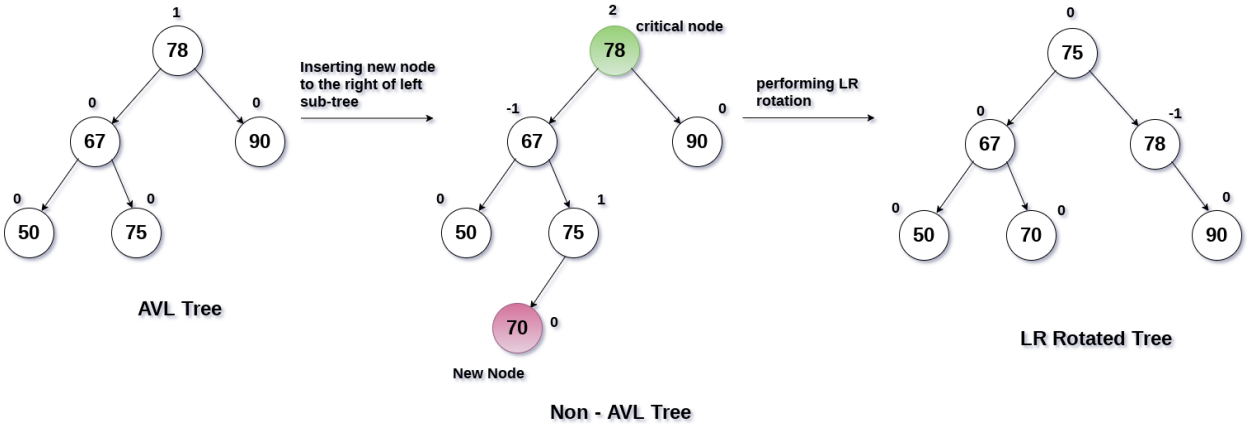
Solution

According to the property of binary search tree, the node with value 70 is inserted into the right of the left sub-tree of the root of tree.

As shown in the figure, the balance factor of the root node disturbed upon inserting 70 and this becomes the critical node A.

To rebalance the tree, LR rotation is to be performed. Node C i.e. 75 will become the new root node of the tree with B and A as its left and right child respectively.

Sub-trees T1, T2 become the left and right sub-tree of B while sub-trees T3, T4 become the left and right sub-tree of A.

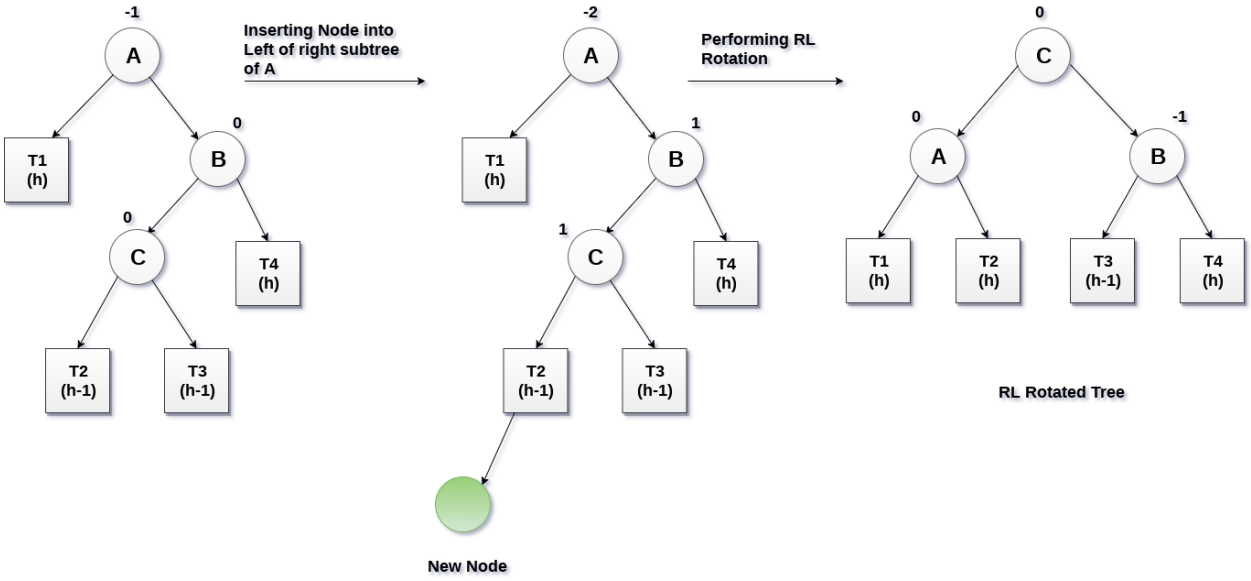


RL Rotation

RL rotations is to be performed if the new node is inserted into the left of right sub-tree of the critical node A. Let us consider, Node B is the root of the right sub-tree of the critical node, Node C is the root of the sub-tree in which the new node is inserted.

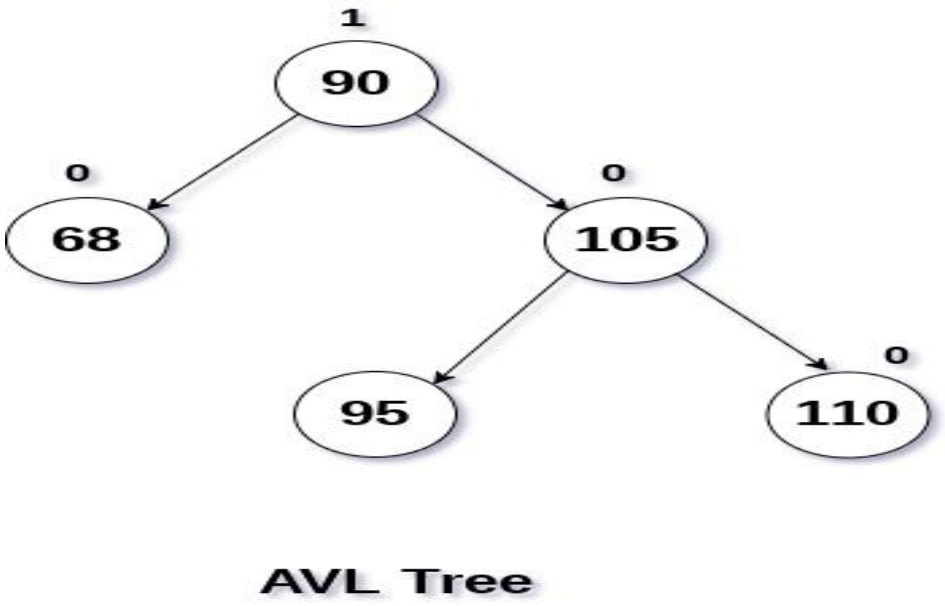
Let T1 be the left sub-tree of the critical node A, T2 and T3 be the left and right sub-tree of Node C respectively, sub-tree T4 be the right sub-tree of Node B.

Since, RL rotation is the mirror image of LR rotation. In this rotation, the node C becomes the root node of the tree with A and B as its left and right children respectively. Sub-trees T1 and T2 becomes the left and right sub-trees of A whereas, T3 and T4 becomes the left and right sub-trees of B.



Example

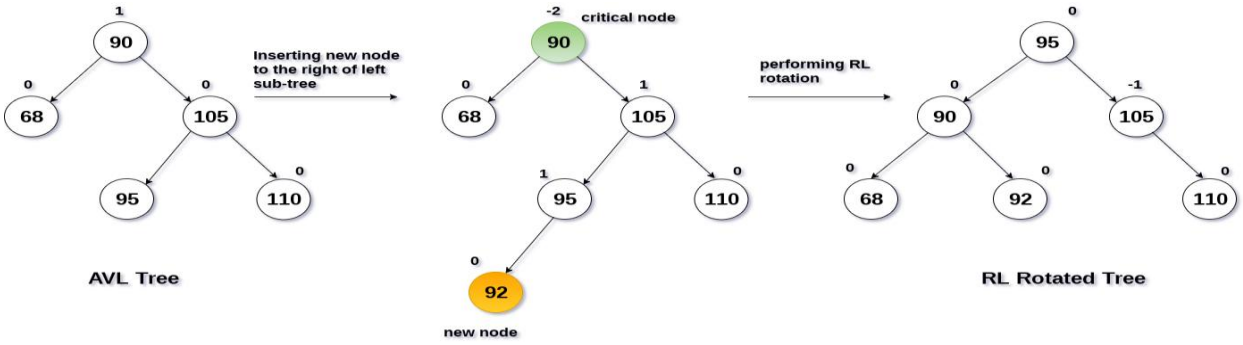
Insert node with the value 92 into the tree shown in the following figure



AVL Tree

Solution :

inserting 92 disturbs the balance factor of the node 92 and it becomes the critical node A with 105 as the node B and 95 with



Deletion in AVL Tree

Deleting a node from an AVL tree is similar to that in a binary search tree. Deletion may disturb the balance factor of an AVL tree and therefore the tree needs to be rebalanced in order to maintain the AVLness. For this purpose, we need to perform rotations. 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.

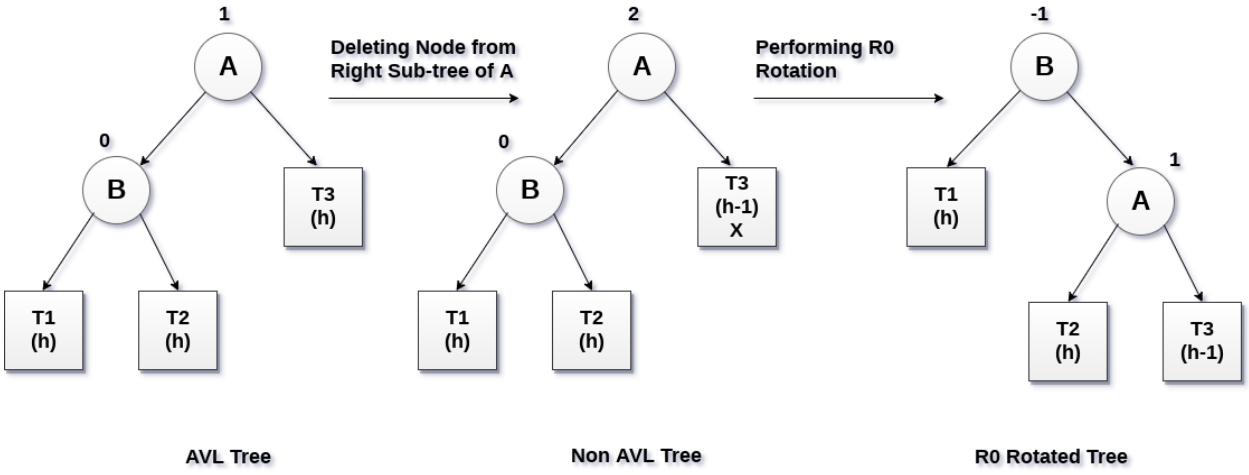
If the node which is to be deleted is present in the left sub-tree of the critical node, then L rotation needs to be applied else if, the node which is to be deleted is present in the right sub-tree of the critical node, the R rotation will be applied.

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:

R0 rotation (Node B has balance factor 0)

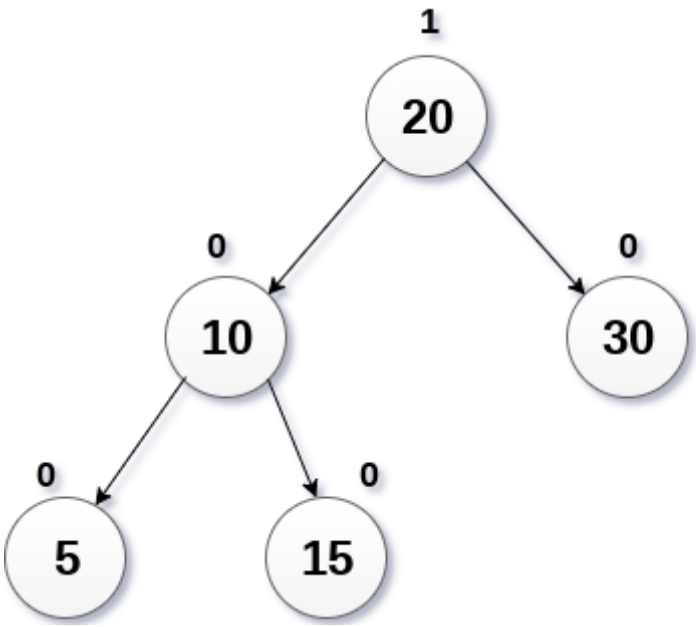
If the node B has 0 balance factor, and the balance factor of node A disturbed upon deleting the node X, then the tree will be rebalanced by rotating tree using R0 rotation.

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.



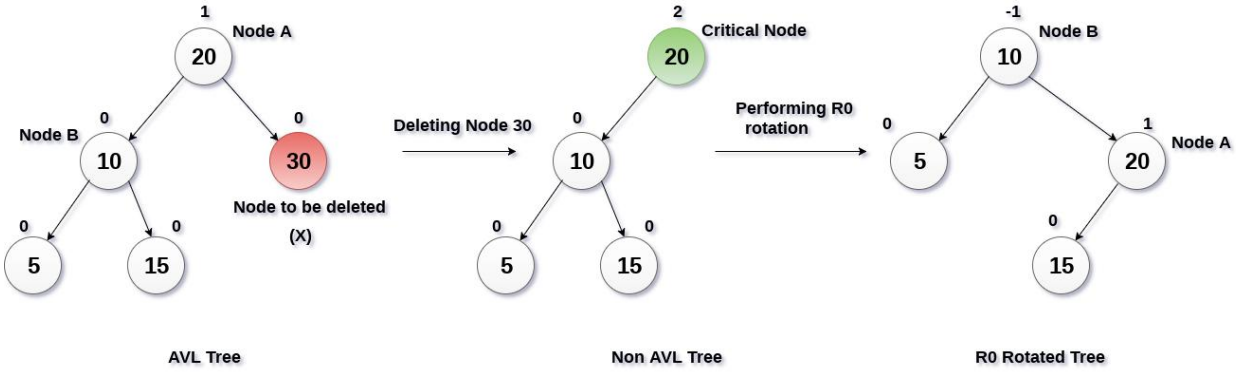
Example:

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



Solution

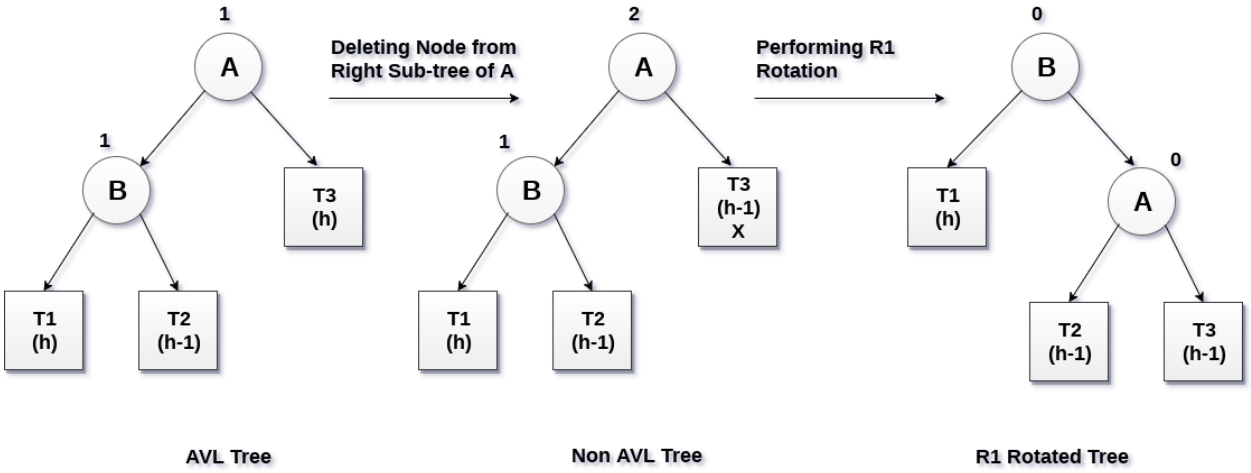
In this case, the node B has balance factor 0, therefore the tree will be rotated by using R0 rotation as shown in the following image. The node B(10) becomes the root, while the node A is moved to its right. The right child of node B will now become the left child of node A.



R1 Rotation (Node B has balance factor 1)

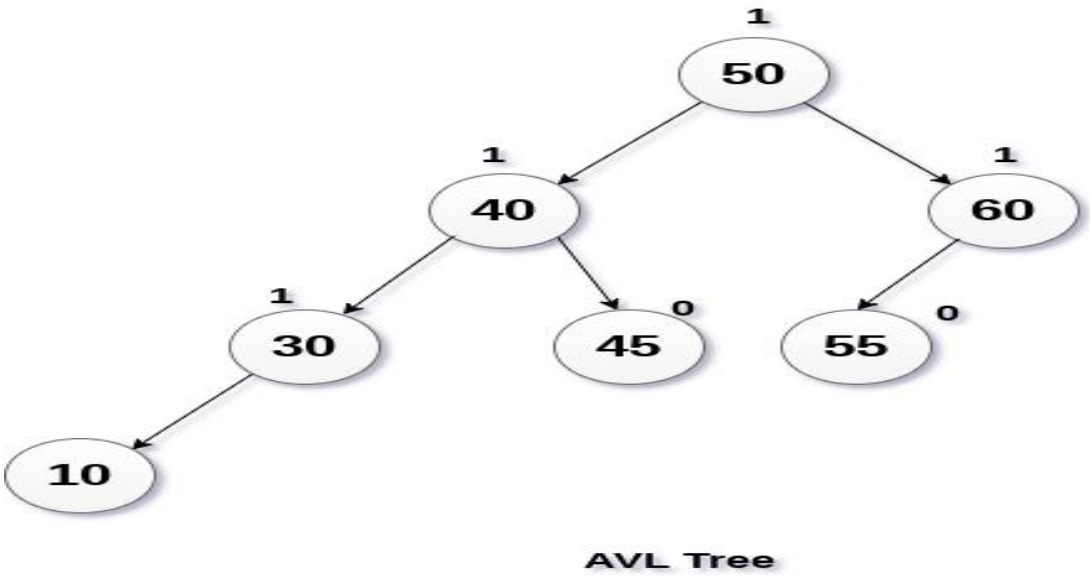
R1 Rotation is to be performed if the balance factor of Node B is 1. In R1 rotation, the critical node A is moved to its right having sub-trees T2 and T3 as its left and right child respectively. T1 is to be placed as the left sub-tree of the node B.

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



Example

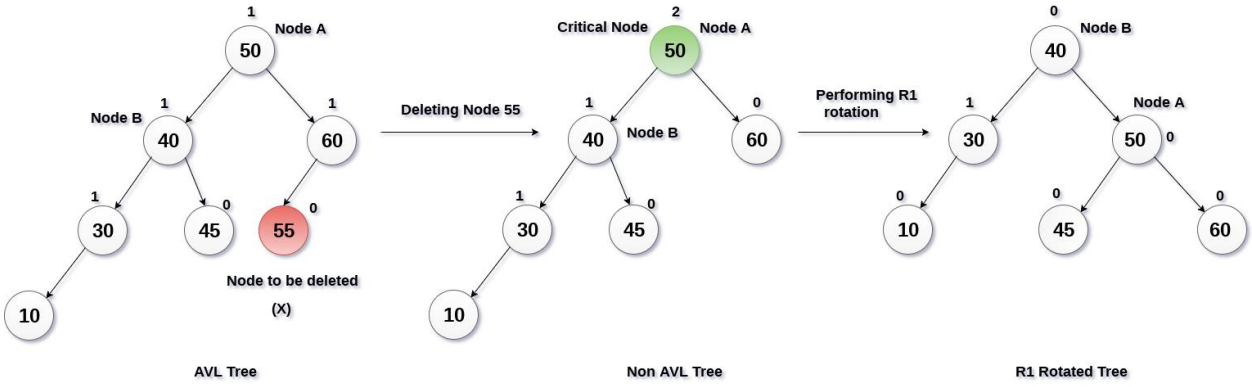
Delete Node 55 from the AVL tree shown in the following image.



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.

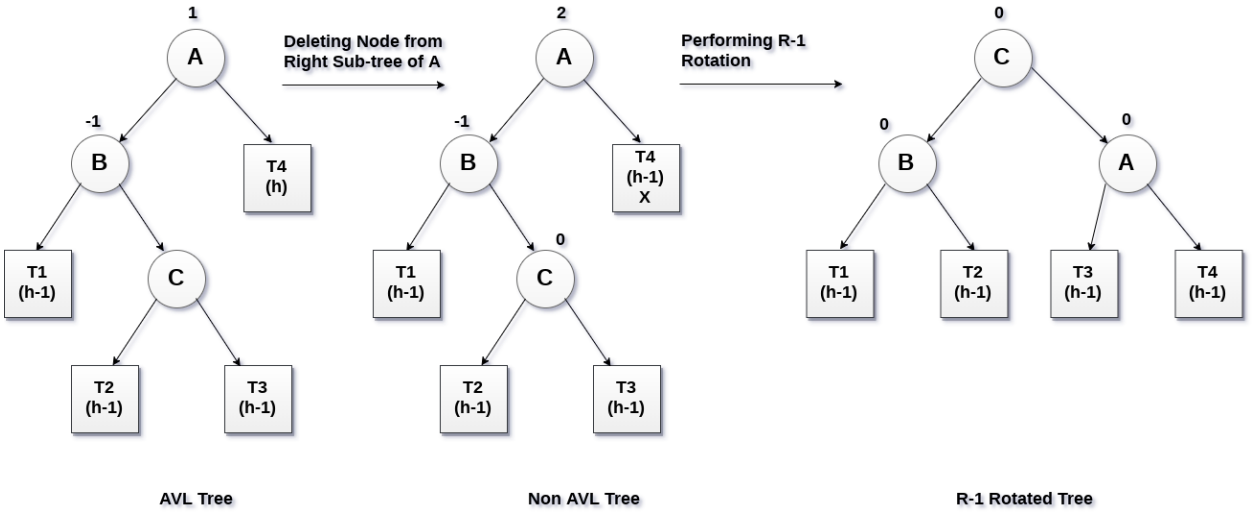


R-1 Rotation (Node B has balance factor -1)

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.

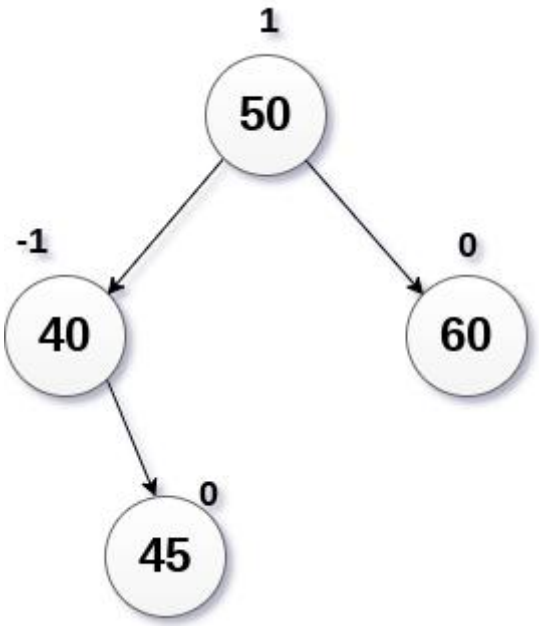
The sub-trees T1, T2 becomes the left and right sub-trees of B whereas, T3, T4 become the left and right sub-trees of A.

The process involved in R-1 rotation is shown in the following image.



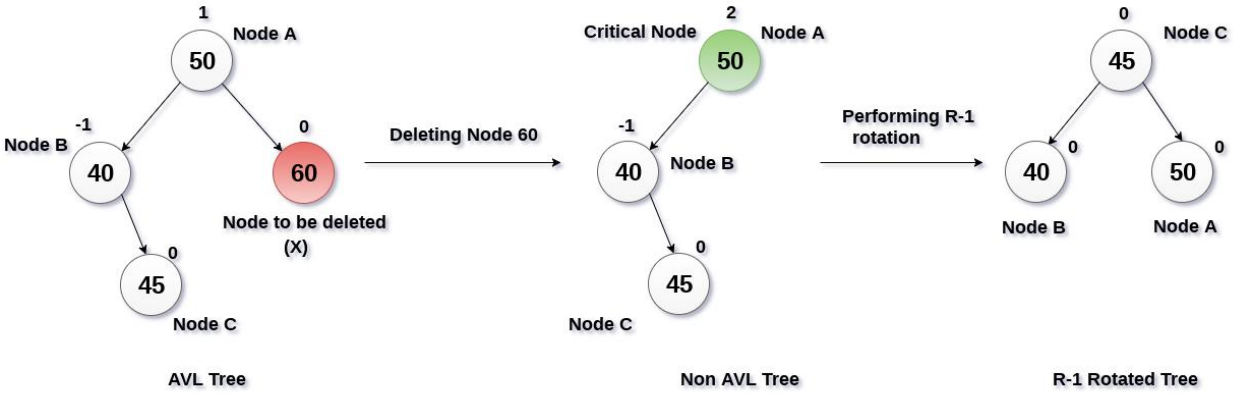
Example

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.



B Tree

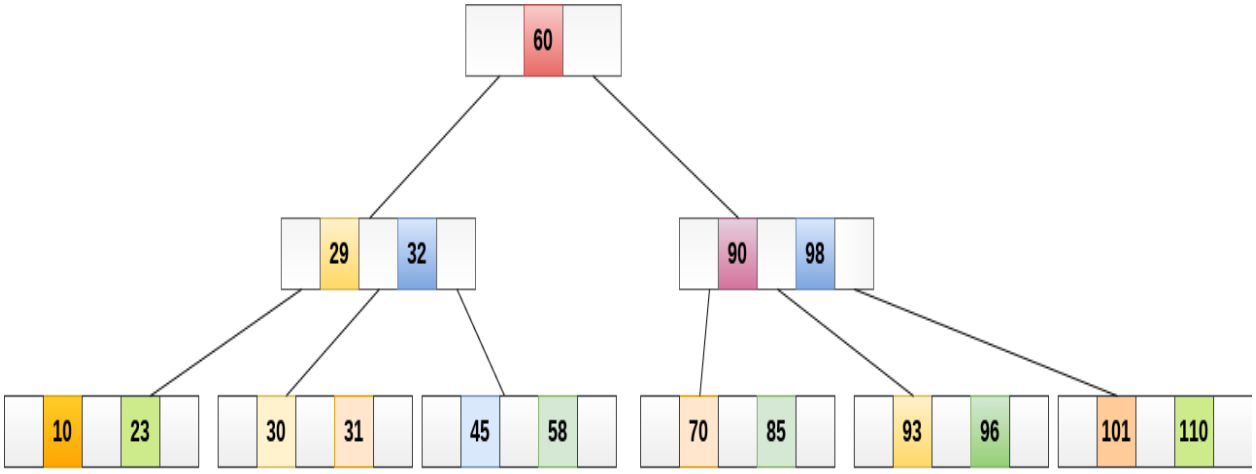
B Tree is a specialized m-way tree that can be widely used for disk access. A B-Tree of order m can have at most m-1 keys and m children. One of the main reason of using B tree is its capability to store large number of keys in a single node and large key values by keeping the height of the tree relatively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

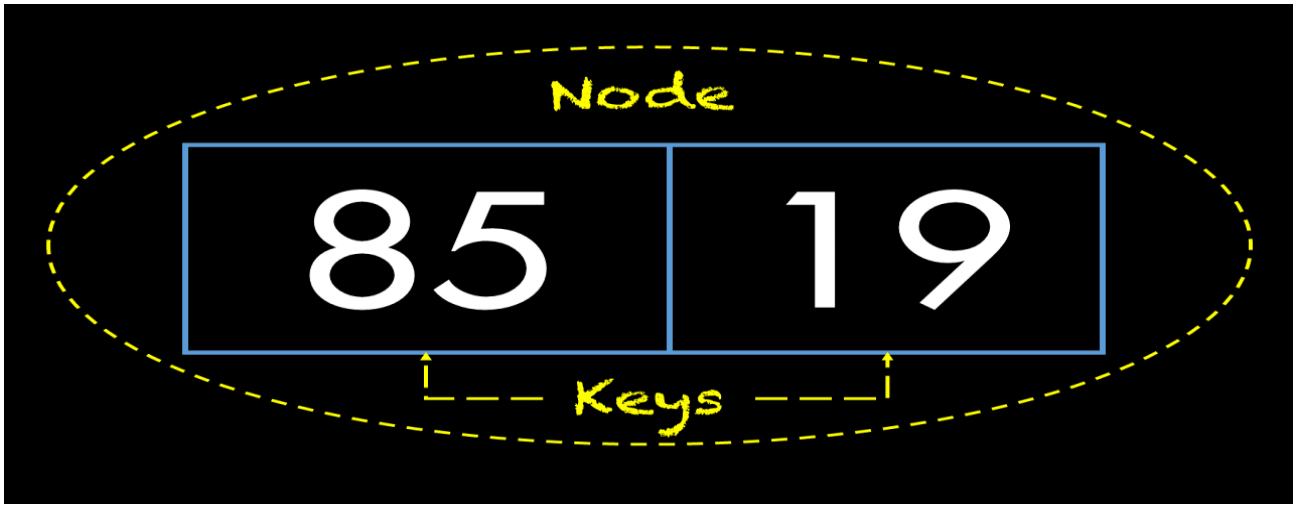
It is not necessary that, all the nodes contain the same number of children but, each node must have m/2 number of nodes.

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.

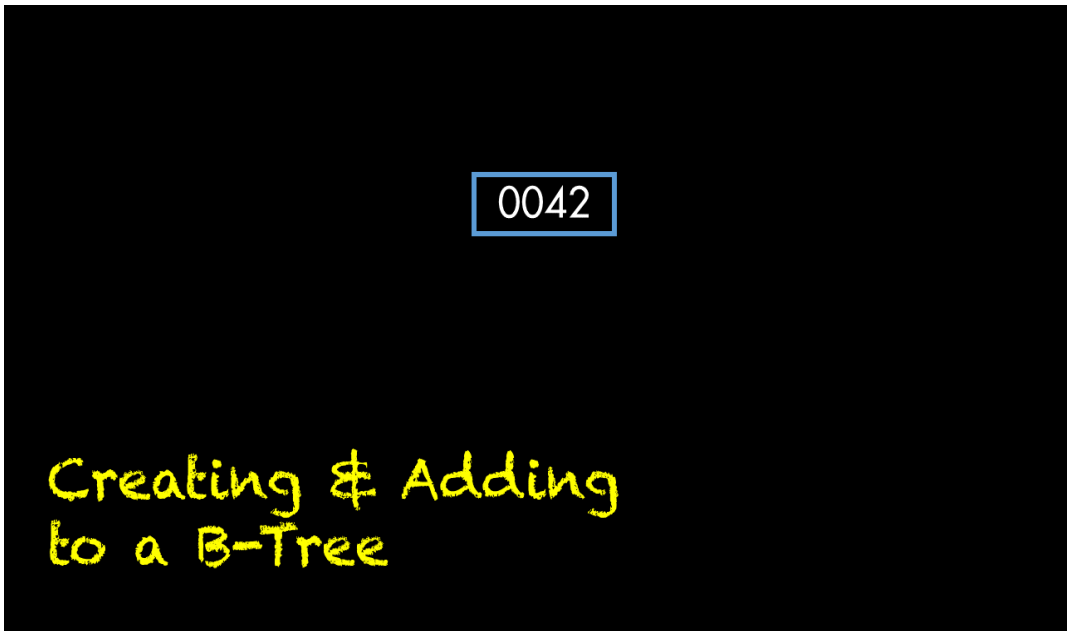
B-Tree Structure



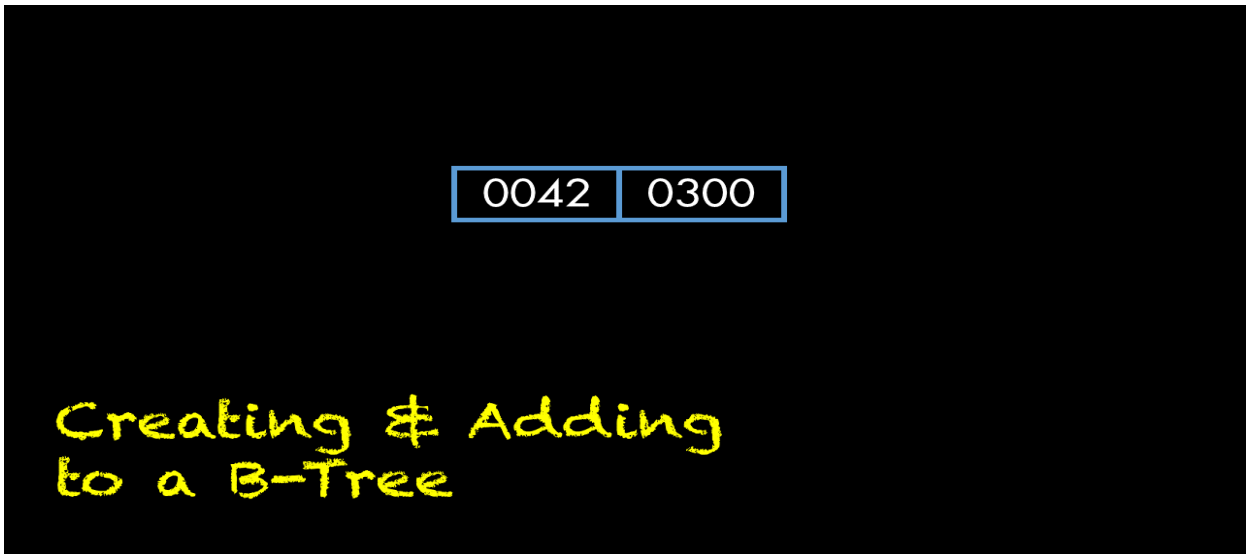
Before we dive into the full structure let's take a look at a single node. B-Tress are setup differently from binary search trees. Instead of nodes storing a single value, B-Tree nodes have the ability to store multiple values, which are called `keys`.

Creating and Adding to a B-Tree

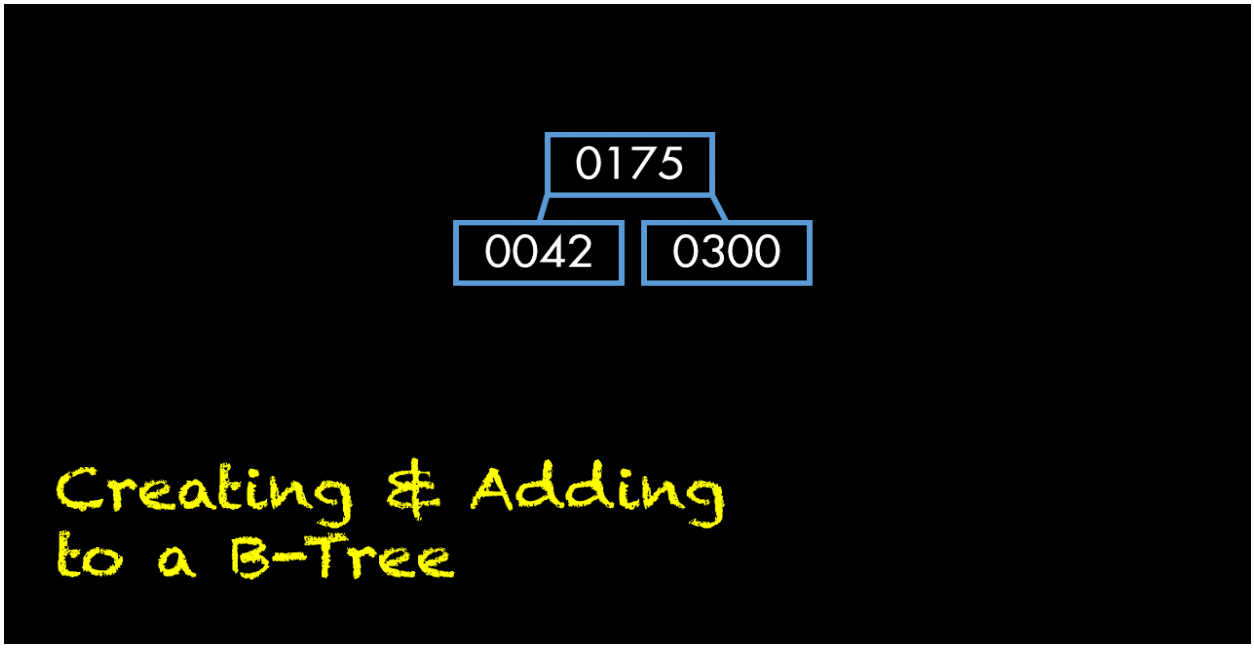
Let's walk through how to create a B-Tree.



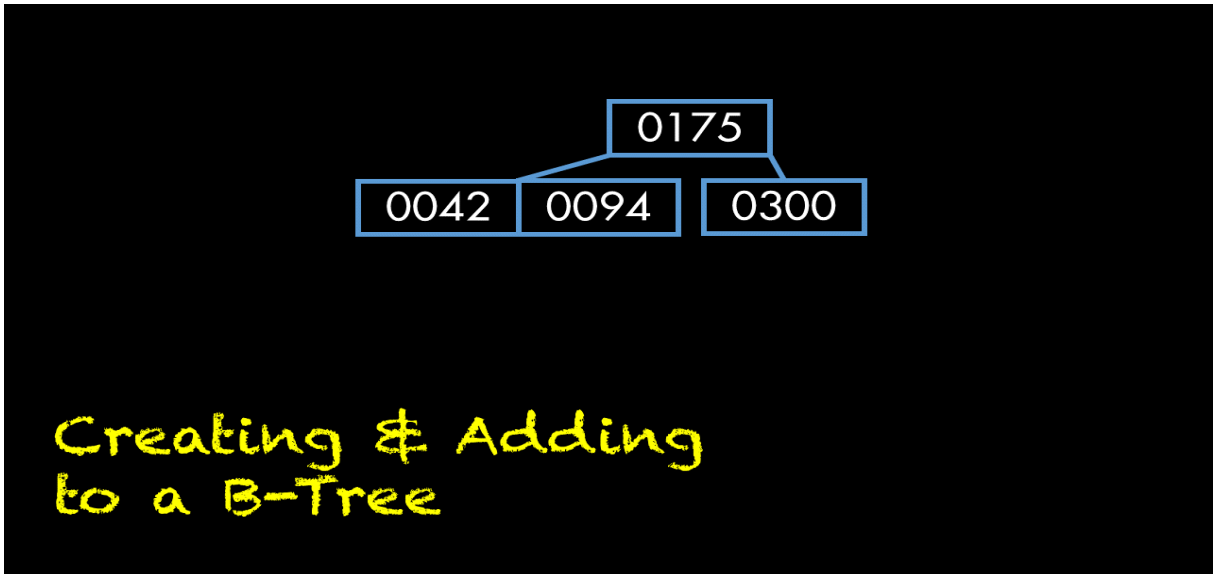
Starting with the integer `0042` our B-Tree will start with a single node which will contain a single key. Make sure to note that this is different from a binary search tree, where the `0042` value would be the node itself.



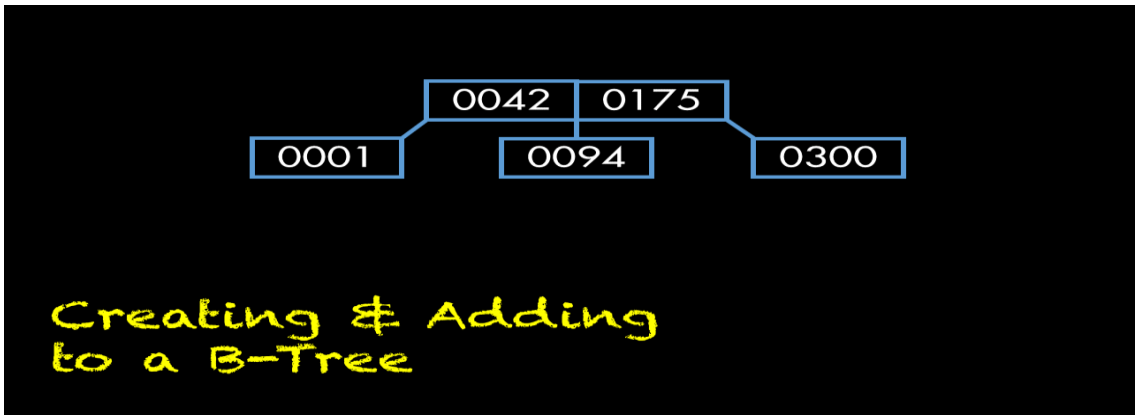
In order to add the value `0300` we will add the key to the same node, so the node will now contain the keys of `0042` and `0300`.



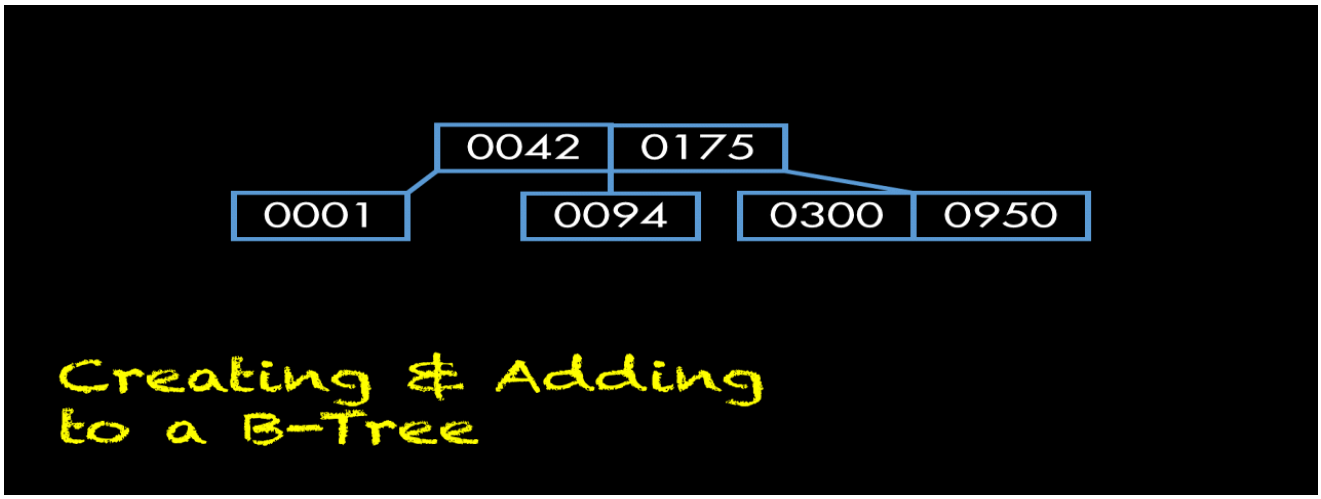
In the next step we see how the self balancing behavior works for B-Trees. In order to add `0175` to the tree we will split the other two values into their own nodes and since `0175` is between the two values we will make it the parent of the other two values.



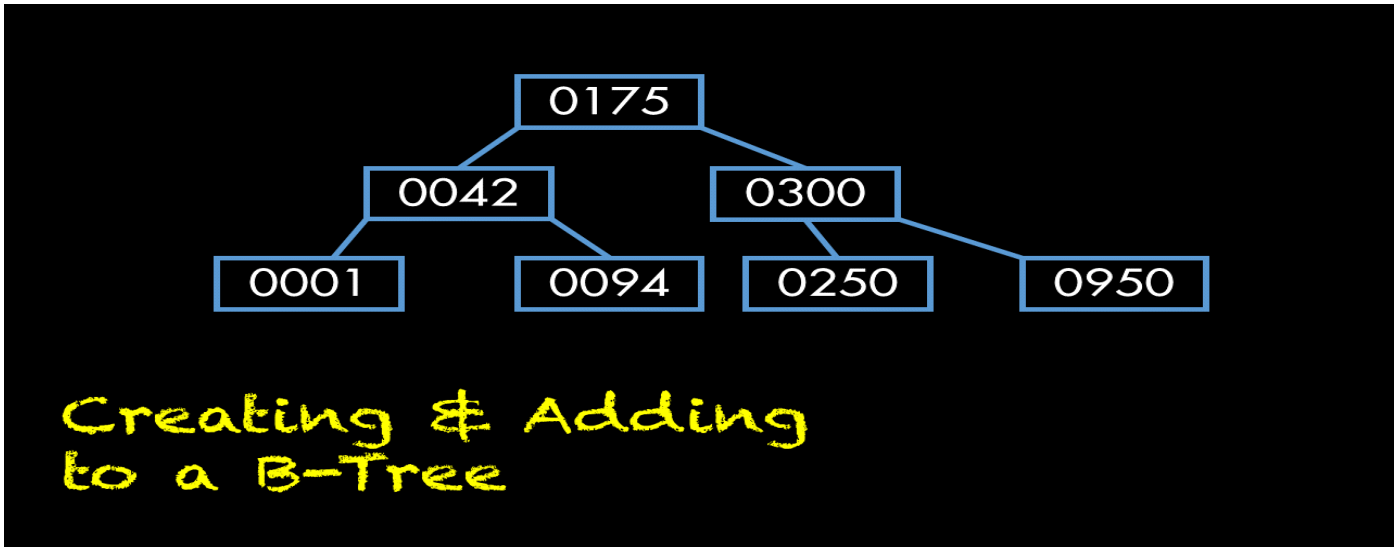
If we want to add the value `0094` to the tree we can insert it as a key inside of the same node that contains the `0042` key. This keeps the integrity of the tree intact.



Now if we want to add the value `0001` we will move `0042` up so it will share a node with the `0175` key. And the new parent node will have three distinct child nodes. Notice how this differs from the behavior of binary search trees. B-Trees are not limited by the two child rule.



If we want to add the large value of `0950` we can insert it at the end of the tree, so it will share a node with the `0300` key.



Lastly, but very importantly, if we want to add the value `0250` to the tree we will have to change the entire structure of the tree. This new value will force us to add a new level to the tree. The `0250` will be a child of the `0300` node since it's less than `0250`. Notice in the illustration how we move the `0175` up to a new level of the tree. This allows us to maintain tree balance while still being able to add new values to the tree.

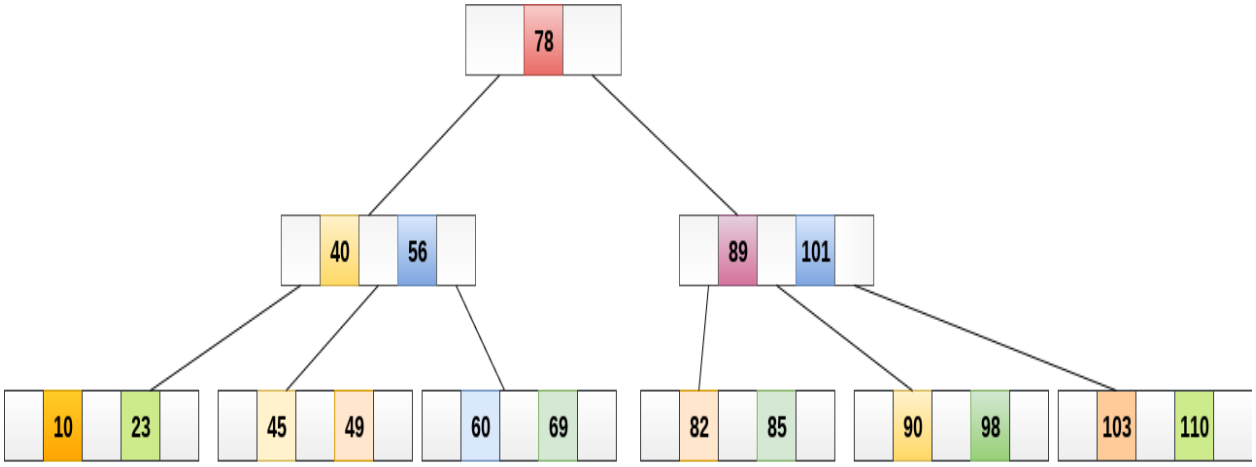
Operations

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.



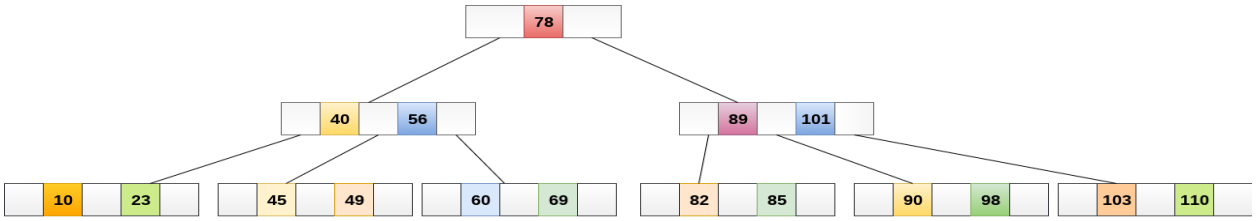
Inserting

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

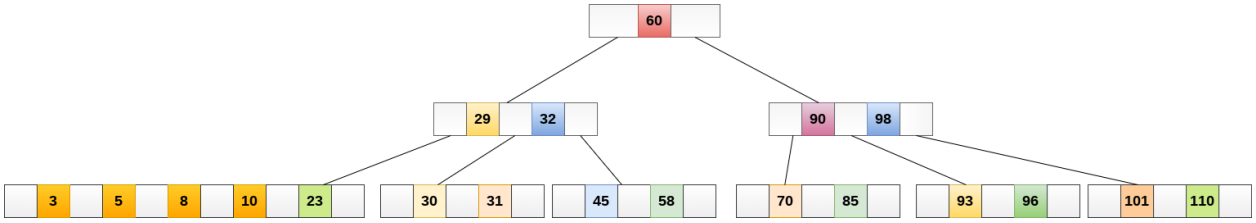
1. Traverse the B Tree in order to find the appropriate leaf node at which the node can be inserted.
2. If the leaf node contain less than m-1 keys then insert the element in the increasing order.
3. Else, if the leaf node contains m-1 keys, then follow the following steps.
    - Insert the new element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

Example:

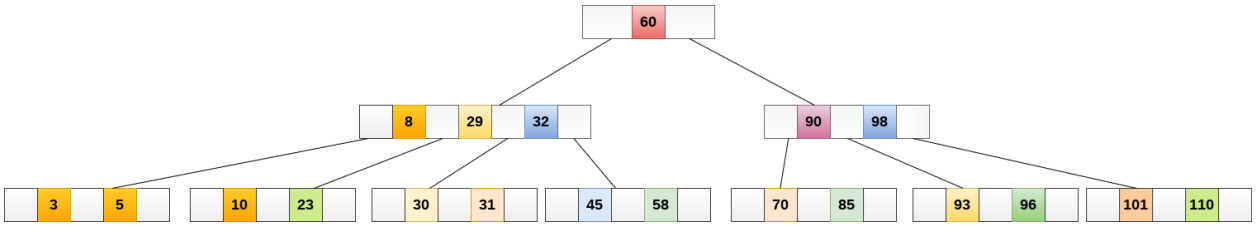
Insert the node 8 into the B Tree of order 5 shown in the following image.



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.



Deletion

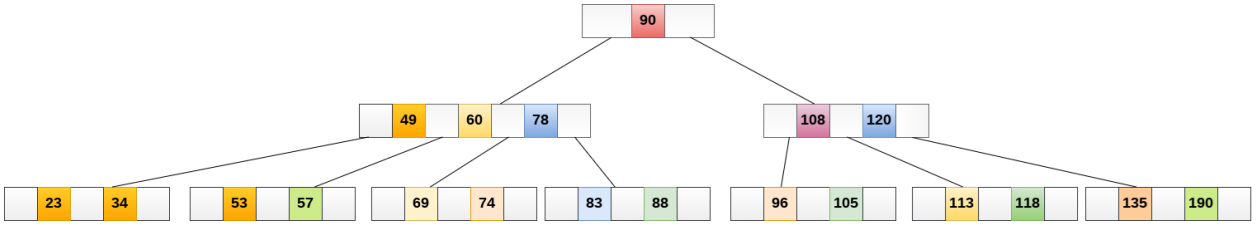
Deletion is also performed at the leaf nodes. The node which is to be deleted can either be a leaf node or an internal node. Following algorithm needs to be followed in order to delete a node from a B tree.

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.
    - 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.
    - 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.

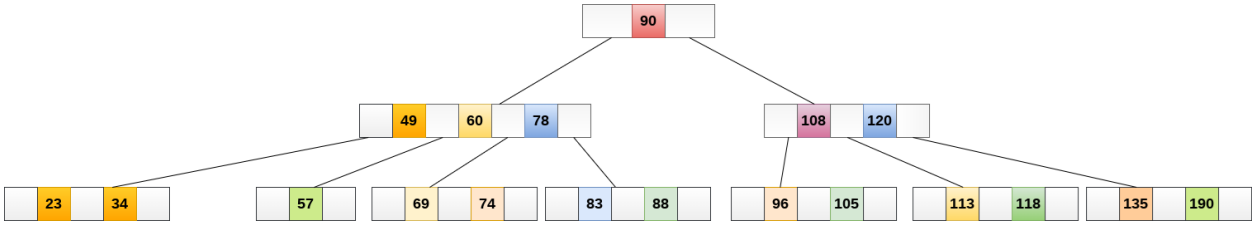
If the the node which is to be deleted is an internal node, then replace the node with its in-order successor or predecessor. Since, successor or predecessor will always be on the leaf node hence, the process will be similar as the node is being deleted from the leaf node.

Example 1

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

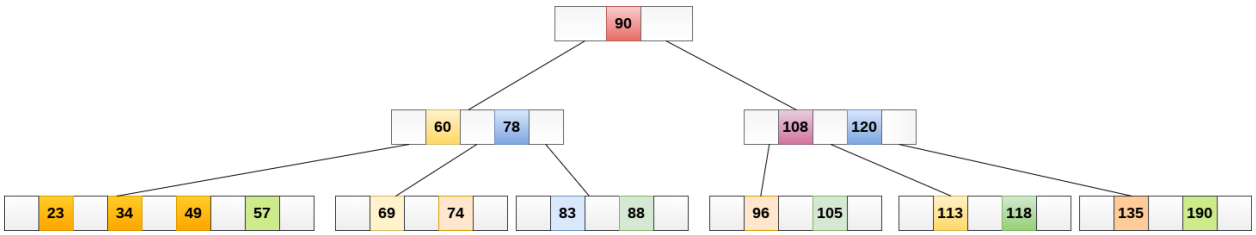


53 is present in the right child of element 49. Delete it.



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.



B+ Tree

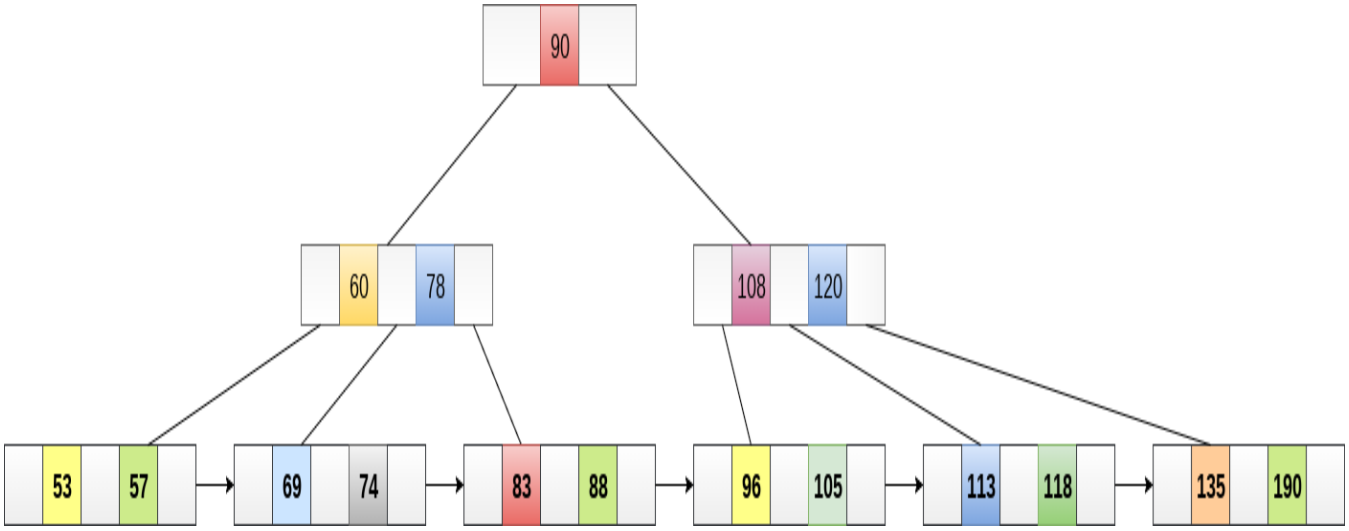
B+ Tree is an extension of B Tree which allows efficient insertion, deletion and search operations.

In B Tree, Keys and records both can be stored in the internal as well as leaf nodes. Whereas, in B+ tree, records (data) can only be stored on the leaf nodes while internal nodes can only store the key values.

The leaf nodes of a B+ tree are linked together in the form of a singly linked lists to make the search queries more efficient.

B+ Tree are used to store the large amount of data which can not be stored in the main memory. Due to the fact that, size of main memory is always limited, the internal nodes (keys to access records) of the B+ tree are stored in the main memory whereas, leaf nodes are stored in the secondary memory.

The internal nodes of B+ tree are often called index nodes. A B+ tree of order 3 is shown in the following figure.



Insertion in B+ Tree

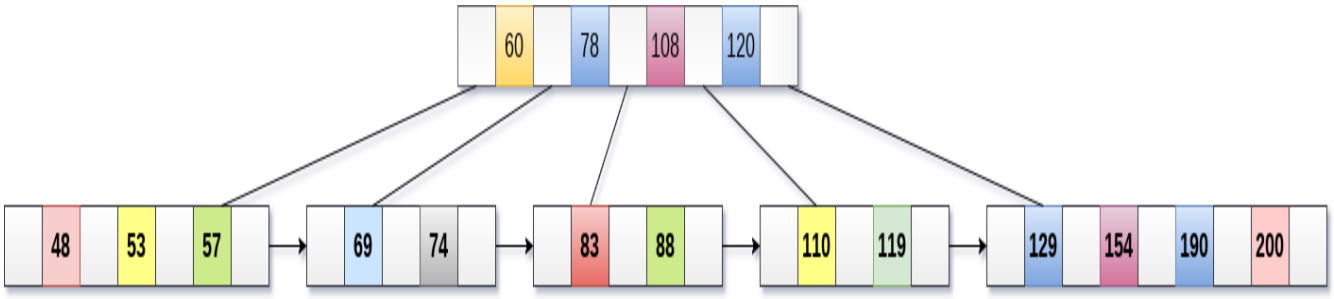
Step 1: Insert the new node as a leaf node

Step 2: If the leaf doesn't have required space, split the node and copy the middle node to the next index node.

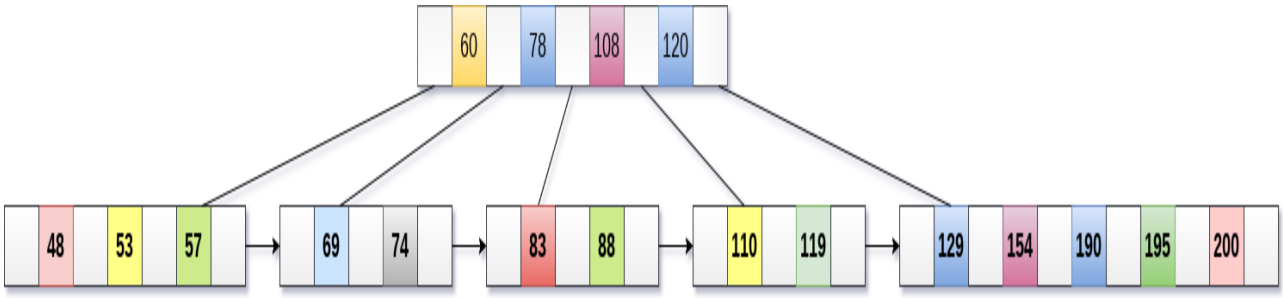
Step 3: If the index node doesn't have required space, split the node and copy the middle element to the next index page.

Example :

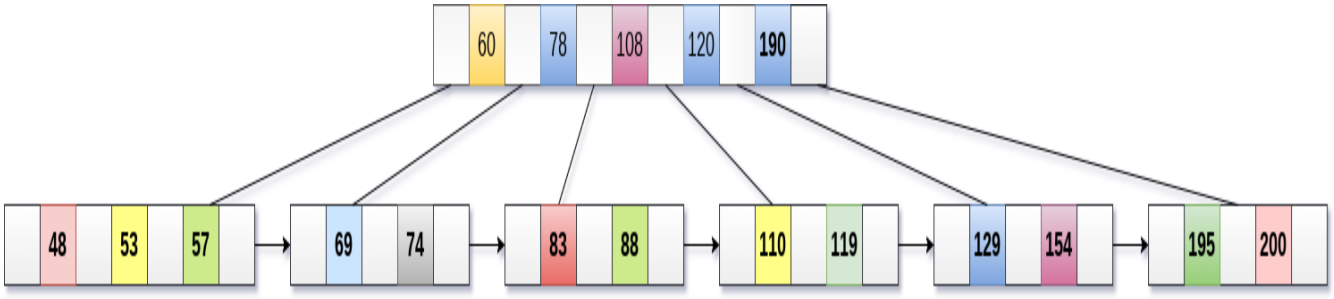
Insert the value 195 into the B+ tree of order 5 shown in the following figure.



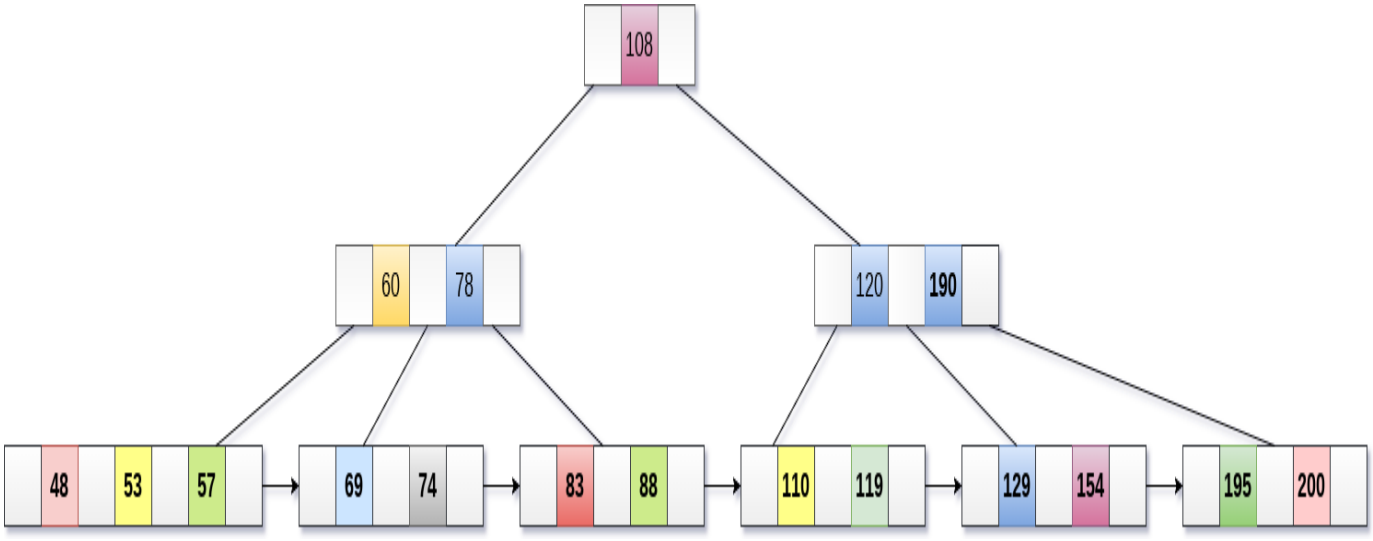
195 will be inserted in the right sub-tree of 120 after 190. Insert it at the desired position.



The node contains greater than the maximum number of elements i.e. 4, therefore split it and place the median node up to the parent.



Now, the index node contains 6 children and 5 keys which violates the B+ tree properties, therefore we need to split it, shown as follows.



Deletion in B+ Tree

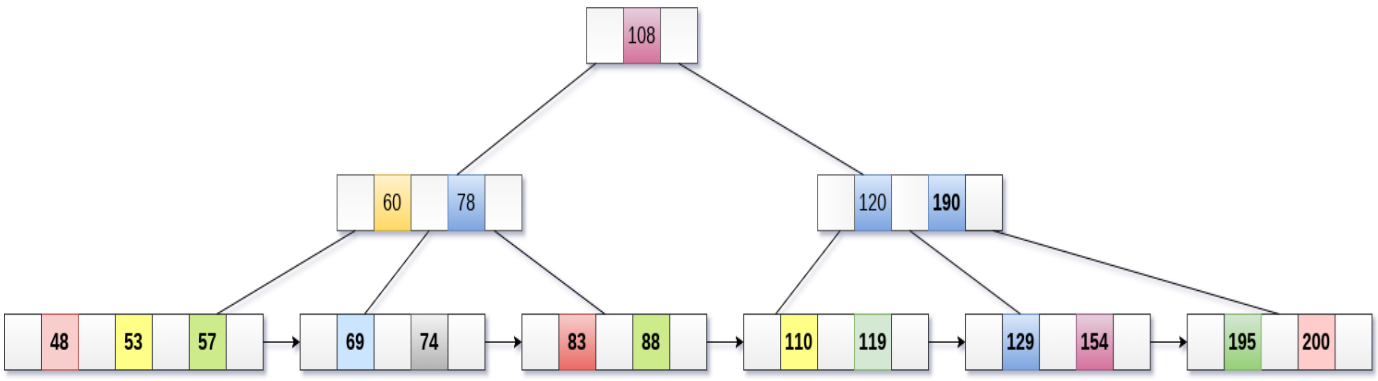
Step 1: Delete the key and data from the leaves.

Step 2: if the leaf node contains less than minimum number of elements, merge down the node with its sibling and delete the key in between them.

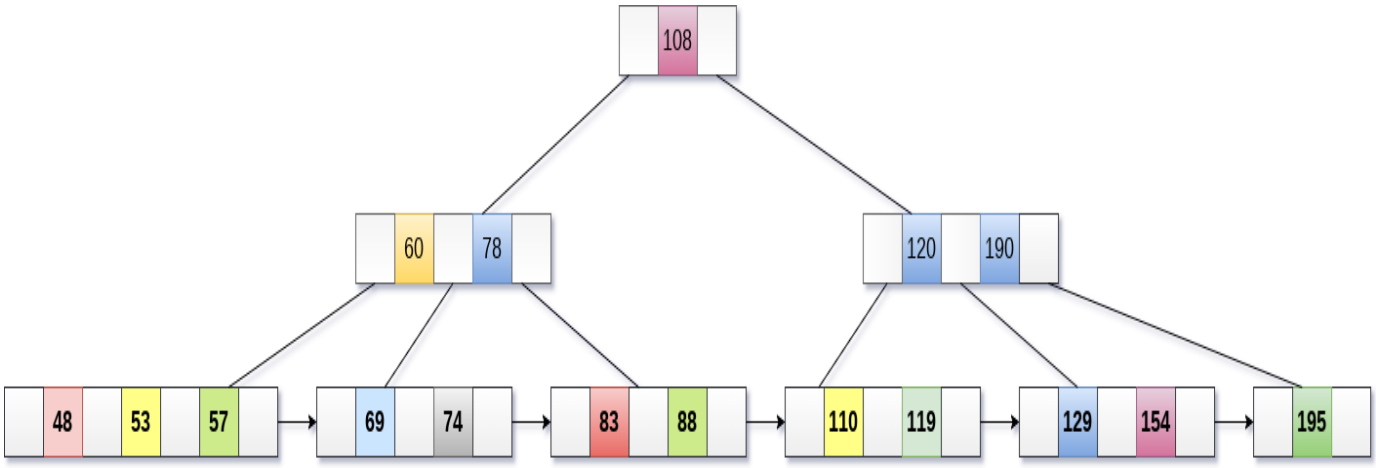
Step 3: if the index node contains less than minimum number of elements, merge the node with the sibling and move down the key in between them.

Example

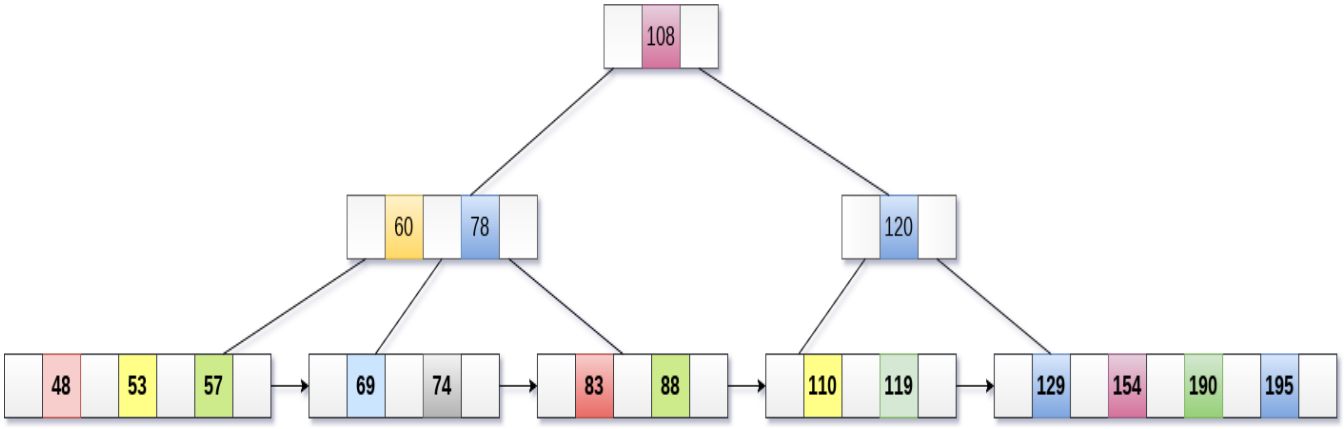
Delete the key 200 from the B+ Tree shown in the following figure.



200 is present in the right sub-tree of 190, after 195. delete it.

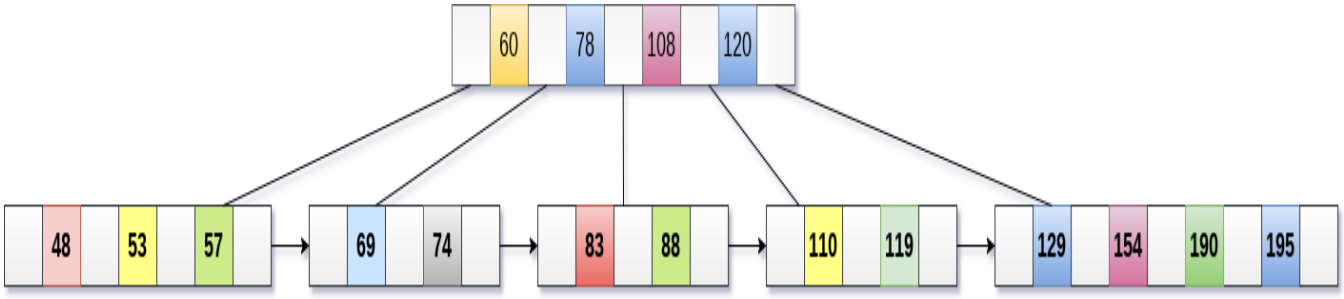


Merge the two nodes by using 195, 190, 154 and 129.



Now, element 120 is the single element present in the node which is violating the B+ Tree properties. Therefore, we need to merge it by using 60, 78, 108 and 120.

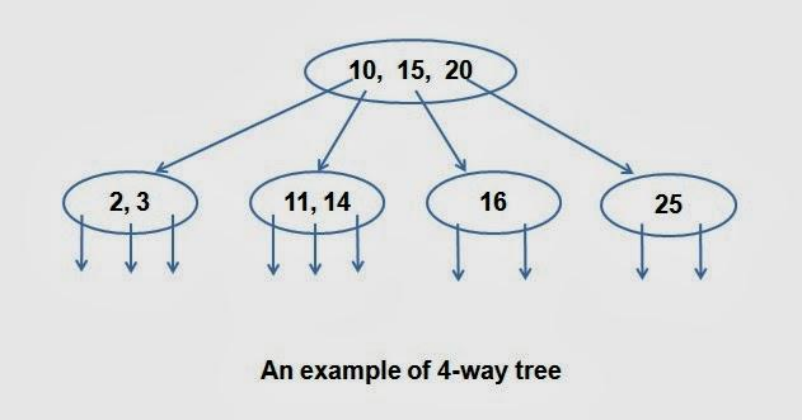
Now, the height of B+ tree will be decreased by 1.



What are B-Trees?

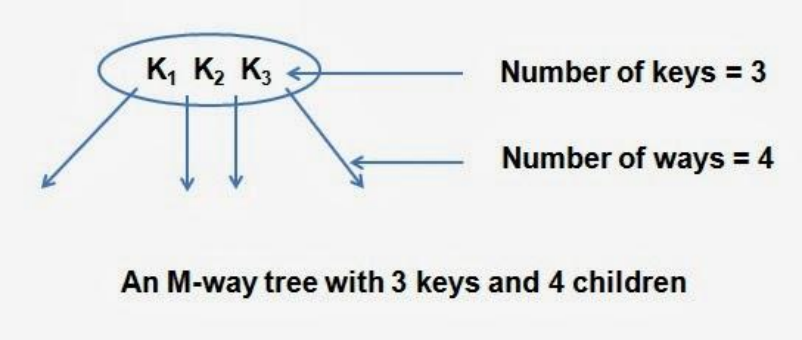
B-tree is another very popular search tree.
The node in a binary tree like AVL tree contains only one record. AVL tree is commonly stored in primary memory. In database application, where huge volume of data is handled, the search tree cannot be accommodated in primary memory. B-trees are primarily meant for secondary storage.

A B-tree is a M-way tree. An M-way tree can have maximum of M children.



An example of 4-way tree

An M-way tree contains multiple keys in a node. This leads to reduction in overall height of the tree. If a node of M-way tree holds K number of keys then it will have K+1 children.



An M-way tree with 3 keys and 4 children

Definition

A B-tree of order M is a M-way search tree with the following properties:

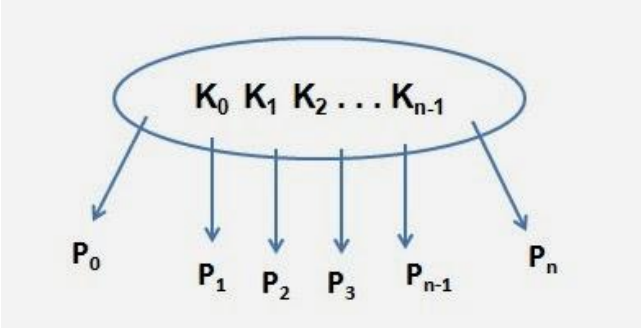
1. The root can have 1 to M-1 keys.

2. All nodes (except the root) have between [(M-1)/2] and M-1 keys.

3. All leaves are at the same depth.

4. If a node has t number of children then it must have (t-1) number of keys.

5. Keys of a node are sorted in ascending order.



6. K0, K1, K2 . . . Kn-1 are the keys stored in the node. Subtrees are pointed by P0, P1 . . . Pn then K0>= all keys of the subtree P1

.

.

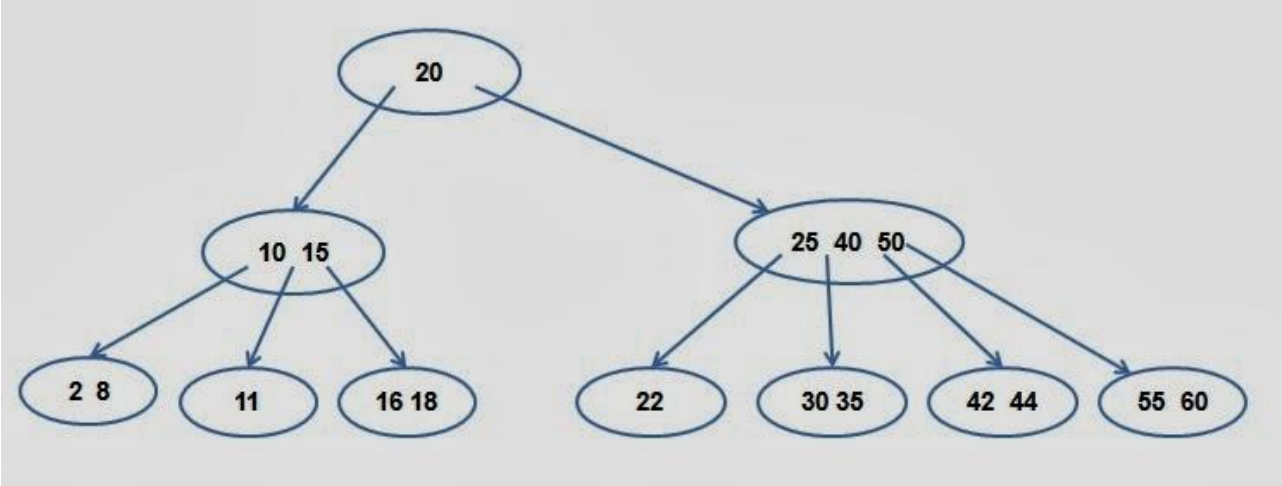
.

.

Kn-1 >= all keys of the subtree Pn-1

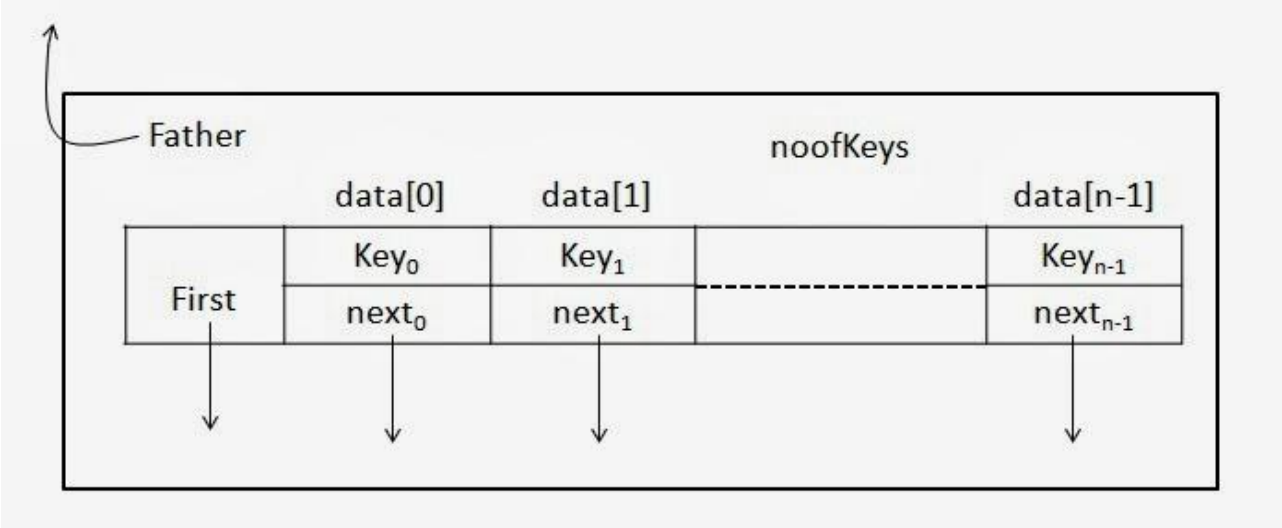
Kn-1 < all keys of the subtree Pn

An example of B-tree of order 4 is shown below:



Representation of a node of B-tree

```
# define MAX 5

struct node;

struct pair

{

        node
*next;

        int
key;

};

struct node

{

        node
*first;

        node
*father;

        pair
data [MAX];

        int
noofkeys;

};
```



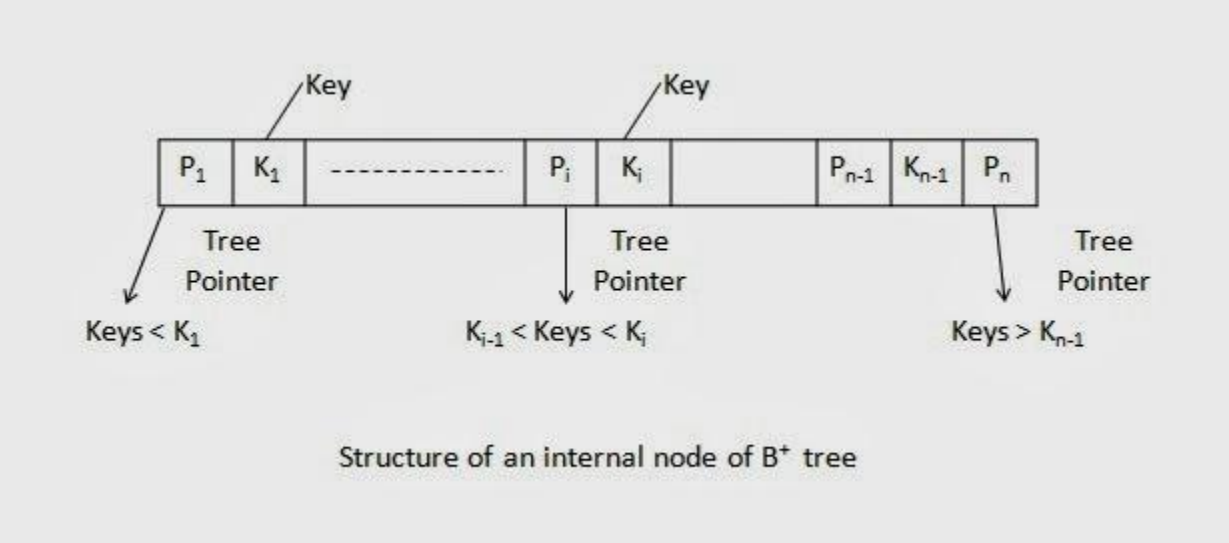
- Structure pair is being used to combine a key and the associated tree pointer.
- Class node can store a maximum of MAX pairs of (key, next). A node with MAX number of keys will give rise to MAX + 1 ways. The additional tree pointer is designated as 'first'.
- 'nooofkeys' gives the actual number of keys stored in a node.
- The pointer 'father' points to the father of a node. 'father' pointer will be NULL for the root node.

What are B+ Trees?

- B+ tree is a variation of B-tree data structure. In a B+ tree, data pointers are stored only at the leaf nodes of the tree. In a B+ tree structure of a leaf node differ from the structure of internal nodes.
- The leaf nodes have an entry for every value of the search field, along with a data pointer to the record (or to the block that contains this record).
- The leaf nodes of the B+ tree are linked together to provide ordered access on the search field to the records.
- Internal nodes of a B+ tree are used to guide the search.
- Some search field values from the leaf nodes are repeated in the internal nodes of the B+ tree.

Structure of Internal node

The structure of the internal nodes is shown below:

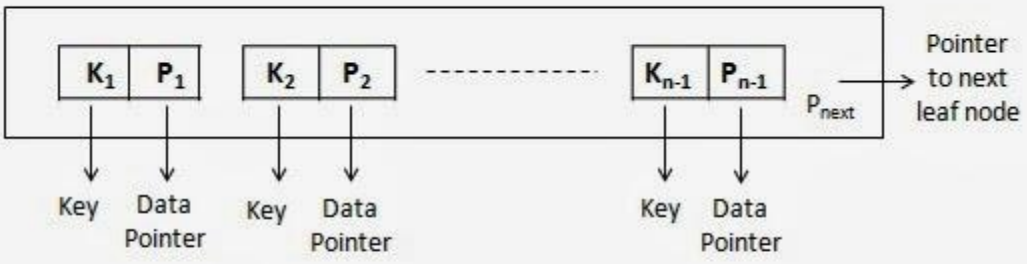


Structure of an internal node of B^+ tree

- Each internal node is of the form < P1, K1, P2, K2 . . . Pn-1, Kn-1, Pn > where Ki is the key and Pi is a tree pointer
- Within each internal node, K1 < K2, . . . < Kn-1
- For all search field value x in the subtree pointed at by Pi, we have Ki-1 x <= Ki.
- Each internal node has at most p tree pointers.
- Each internal node, except the root, has at least [(P/2)] tree pointers.

Structure of a leaf node

The structure of a leaf node of a B+ tree is shown below:



Structure of a leaf node of B+ tree

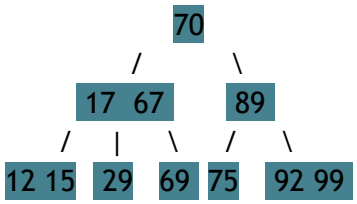
- Each leaf node is of the form <<K1, P1>, <K2, P2> . . . <Kn-1, Pn-1>, Pnext>
- Within each leaf node, K1 < K2 . . . < Kn-1.
- Pi is a data pointer that points to the record whose search field value is Ki.
- Each leaf node has at least ⌈(P/2)⌉ values.
- All leaf nodes are at the same level.

C Program To Perform Insertion, Deletion and Traversal In B-Tree

- B- tree is a multi way search tree. A node in B-tree of order n can have at most n-1 values and n children.
- All values that appear on the left sub-tree are smaller than left most value in the parent node.
- All values that appear on the right sub-tree are greater than right most value in the parent node.

- All values that appear on the middle sub-tree are greater than leftmost value in parent node and smaller than right most value in parent node.

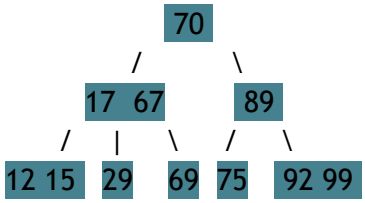
Example for B-tree of Order 3:



Above is an example for B-Tree of order 3.

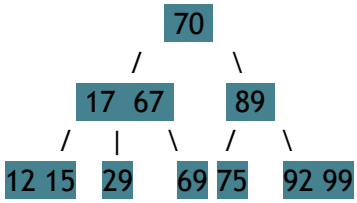
- An intermediate node can have 2 or 3 children.
- Any node can have at most 1 or 2 values.
- Nodes on the left sub-tree are smaller than the left most value in parent node.
- Nodes on the right sub-tree are greater than the right most value in parent node.
- Nodes on the middle sub-tree are smaller than left most value and greater than right most value in parent node.

Searching in B-Tree:



Search the value 29 in above B-Tree.
29 < 70 - So, search in left sub-tree of 70
29 > 17 && 29 < 67 - So, search in middle sub-tree.
29 is the middle child of 17 & 67.

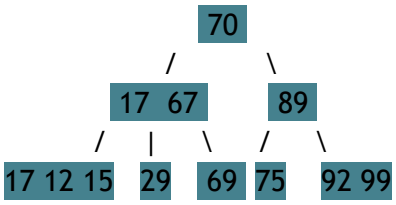
Insertion in B-Tree:



Insert the value 10 to the above B-Tree.
Find the appropriate position to insert the given value in B-Tree.
10 < 70 - Search in left sub-tree of 70.

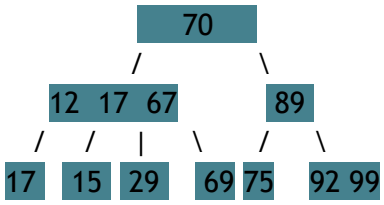
10 < 17 - Search in left sub-tree of 17.

17 needs to be inserted in the left child of 17.

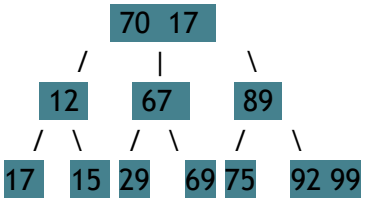


Now, the modified node has 3 values 17, 12 and 15. But, it violates the rule in B-Tree(any node in B-Tree of order can have at most n-1 value).

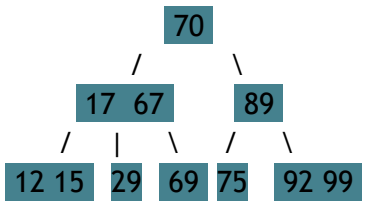
To restore B-Tree, middle value of 17, 12 and 15 is moved to parent node. Then, split the resultant node containing 17 and 15 into two nodes forming left and right sub-tree containing the value 17 and 15 correspondingly.



Now, the parent node violates B-Tree definition. So, restore it.



Deletion in B-Tree:



Delete 69 from the above B-Tree. Search the position of 69 to delete.
69 < 70 - Search in left sub-tree of 70
69 > 17 - Compare 69 with right most value in the search node.
60 > 67 - Search in right sub-tree of 67
69 is the right child of 67.

In case the node which we are trying to delete has only one value(69), then find the predecessor(29) for it(69) and merge the predecessor with the sibiling(29) of the node to be deleted. Then, delete the desired node.