DATA STRUCTURE

DS Core Group

ABES Engineering College
College Code-032

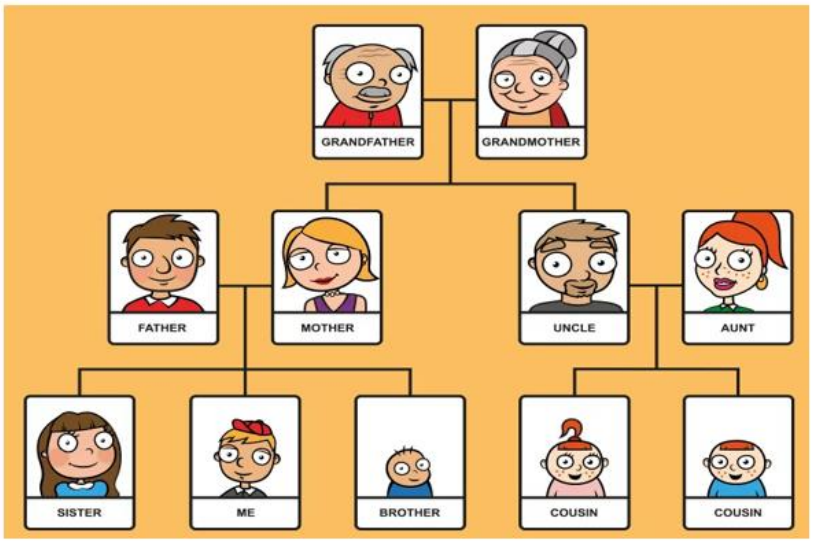
Authors	Akhilesh Kumar Srivastava, Amit Pandey, Puneet Kumar Goyal, Manish Srivastava, Amrita Jyoti, Asmita Dixit
Authorized By	
Creation/Revision Date	May 2021
Version	1.0

Chapter 10
Tree and Tries

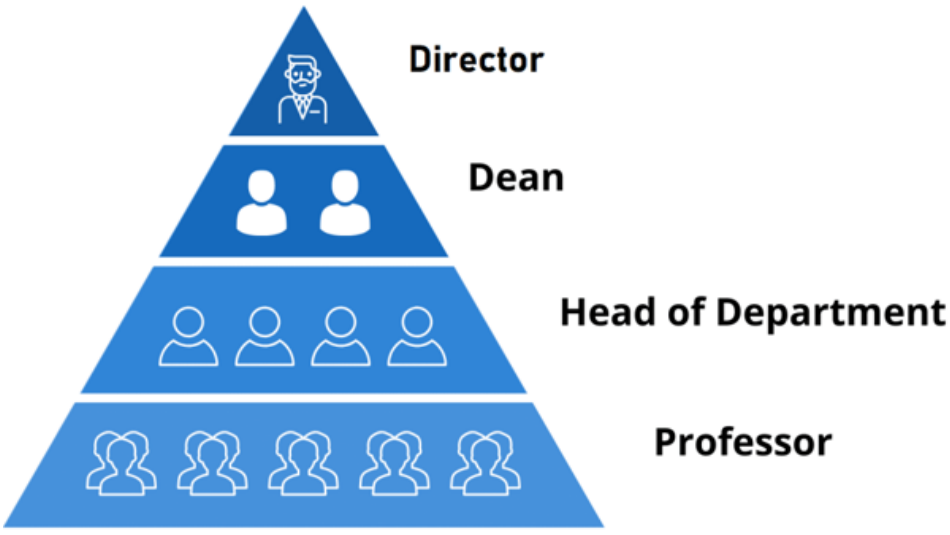
Table of Contents

10.1 Introduction

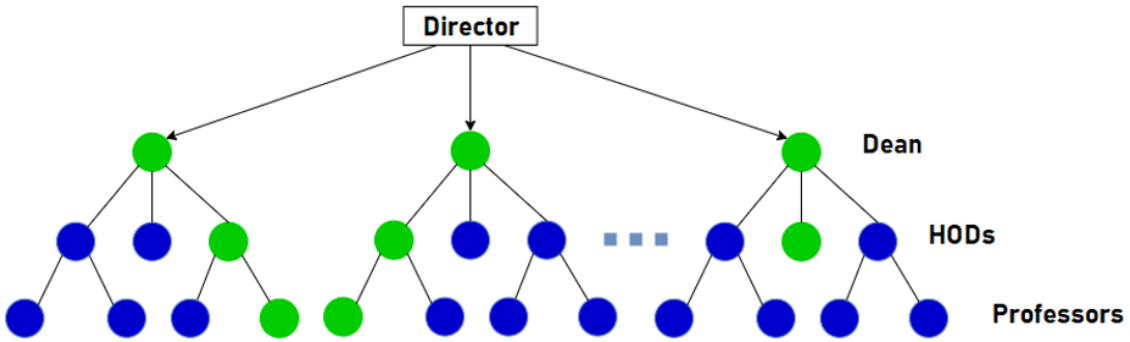
Consider a real-life tree where there are a root, branches and leaves. Sub-branches are formed from the branches. There are no branches from a leaf. A tree in Data structure is a similar structure (also called inverted tree), a hierarchical data structure in which the elements can be represented in nodes at different levels. There is a root, branches and leaf in the Tree Data structure. There are usually fewer elements at the upper level and more elements at the lower level. Consider the family tree given below.



Take another situation in which the hierarchy of an educational institution is represented. There is one Director, 2-3 Deans, 10s of Heads of Departments, many faculty members under each Head of Department.



The structure above can also be represented in the form of the tree given below.



The new TreeView App on Facebook allows integrating family tree with Facebook friends and family. You can generate a family tree automatically using the relationship data on your Facebook info page.

10.1 General Tree

A tree is a non-linear hierarchical data structure that follows a Parent-Child relationship. It is a collection of nodes where one node is defined as ROOT node, and this root can have zero or more children.

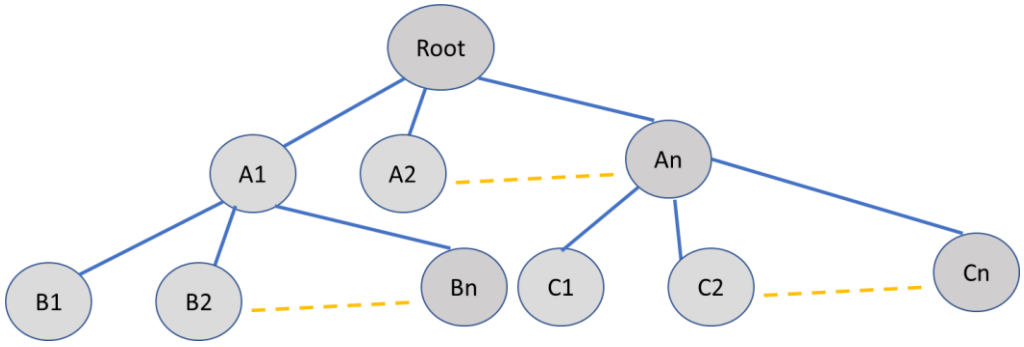


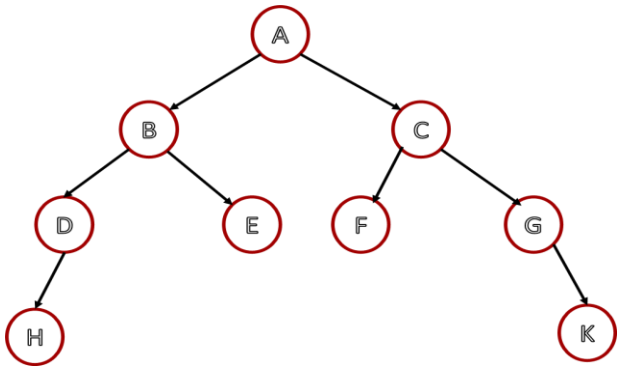
Figure: Decision Tree - Binary Tree Analogy

In this analogy, we are showing that, in each decision, we are choosing a branch, either left(No) or Right(Yes), and eliminating the other possibilities. We continue this approach until we reached a leaf node, i.e., final decision. This same approach, Binary tree follows.

10.2 Binary Tree

It is a type of tree in which each node has either zero, one, or at most two children. Here one child is called left child, and another child is called the right child.

10.2.1 Basic Terminologies



Root Node:

- ✓ Starting node or first element of a tree.
- ✓ A tree may have only one root node.

Edge:

- ✓ It is a link that connects two nodes.
- ✓ If there are n nodes in the tree, there should be (n-1) edges.

Parent node:

A node is called a parent node if node has any number of branches while moving from top to bottom.

Child Node:

- ✓ A node that has an edge from its parent node is called a child node.
- ✓ In a tree all nodes are child nodes except the root node.

Sibling:

- ✓ Child nodes of the same parent nodes are called siblings with each other.

Leaf node:

- ✓ A node that does not has any child is called a leaf node.

Internal node:

- ✓ These are the nodes which have child nodes or nodes other than leaf node.

Degree:

- ✓ It is defined as the number of children or child nodes of a parent node.
- ✓ All leaf nodes have degree zero.

The degree of a tree is equal to the maximum degree among all nodes.

Level of a Tree:

- ✓ It can be defined as each step in tree.
- ✓ The level starts from 0, and it is incremented by 1 while moving from root to leaf.

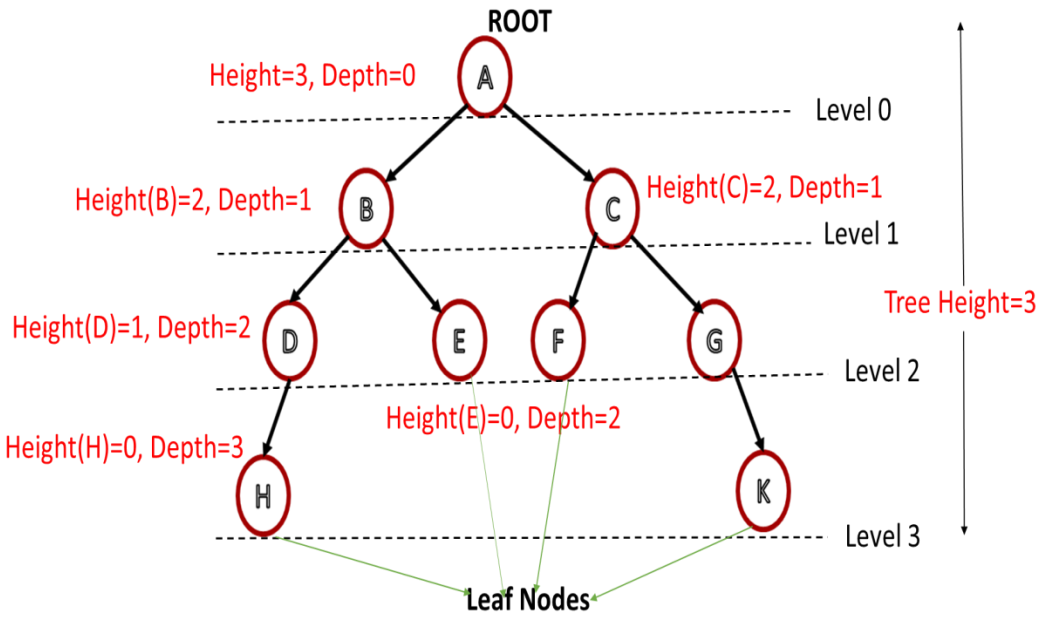
Height:

- ✓ The height of any node is the longest path from the leaf node to that node.
- ✓ The height of the tree is the longest path from the leaf node to the root node.

The height of the leaf node is zero.

Depth:

- ✓ Depth of any node is defined as the longest path from the root node to that node.
- ✓ The depth of a tree is defined as the longest path from the root node to the leaf node.
- ✓ The depth of the root node is zero.



Path:

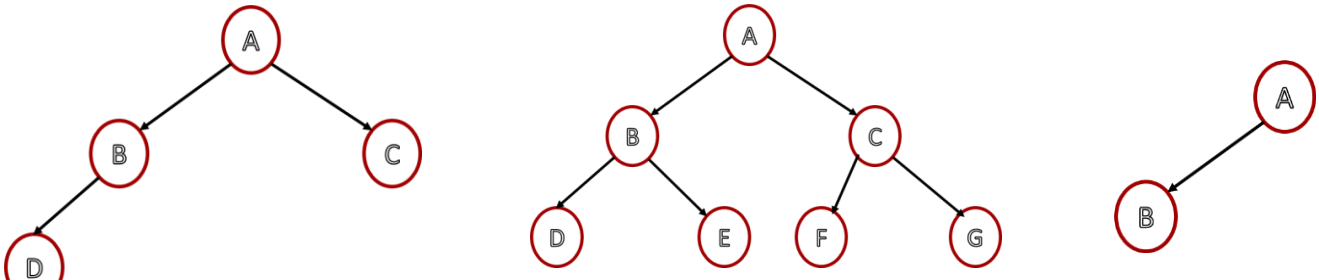
- ✓ A Path can be defined as the sequence of edges from the source node to the destination node.
- ✓ It can also be defined as the sequence of nodes from the source node to the destination node.

Subtree:

- ✓ Node with any number of children is called subtree.

Binary Tree

- ✓ A tree T is called a binary tree if T has either 0, 1, or 2 children.
- ✓ It cannot have more than two children.
- ✓ There is a unique path from the root node to every other node.

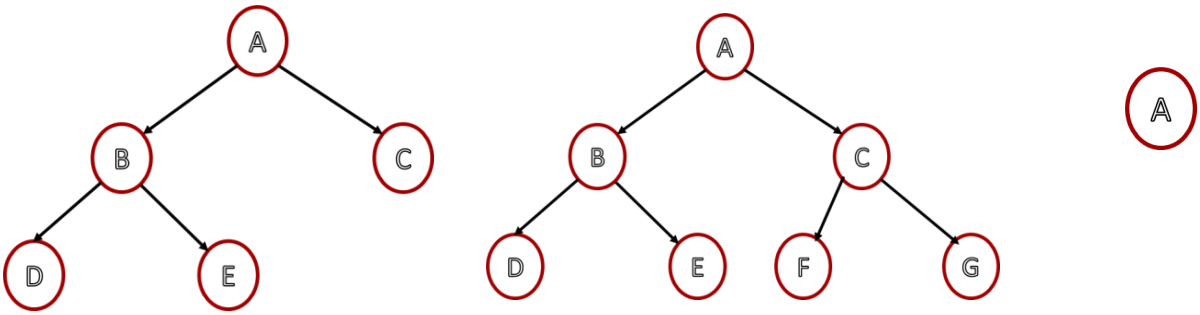


10.2.2 Types of Binary Tree (According to shape)

1) Full/Strictly Binary Tree.
2) Perfect Binary Tree.
3) Complete Binary Tree.
4) Almost Complete Binary Tree.

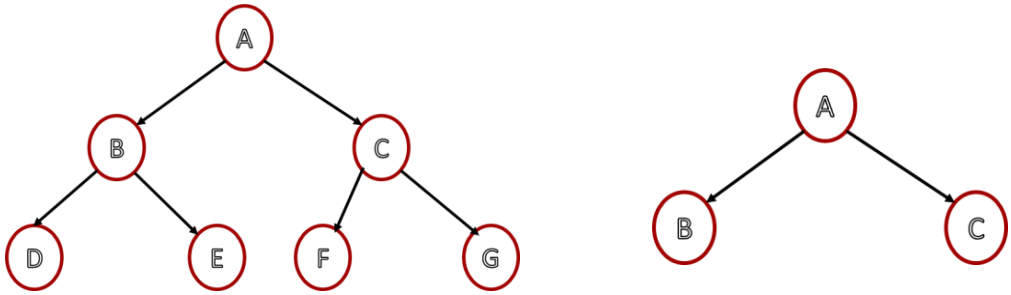
10.2.2.1 Full/Strictly Binary Tree

- ✓ Each node has either 0 or 2 children.



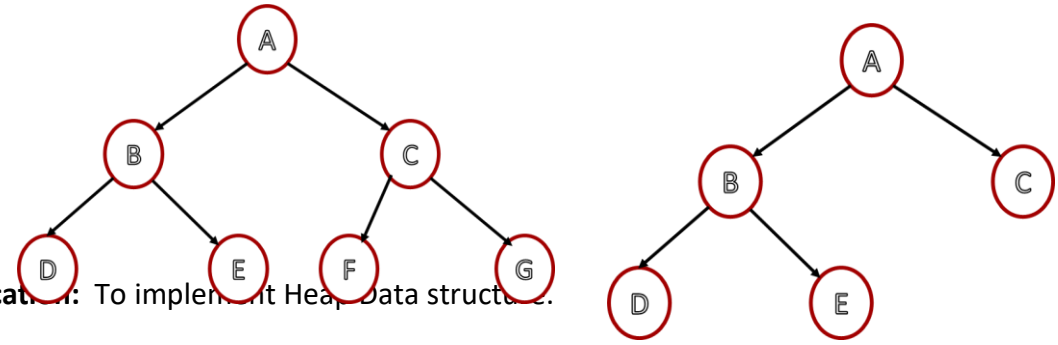
10.2.2.2 Perfect Binary Tree

- ✓ Each node has either 0 or 2 children.
- ✓ All the leaf nodes should be at the same level.



10.2.2.3 Complete Binary Tree

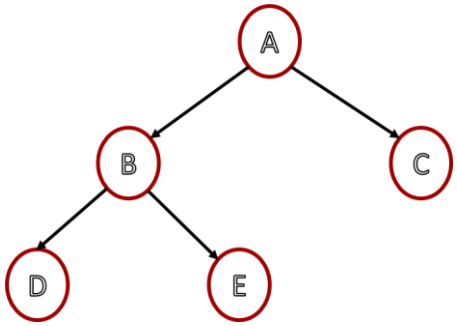
- ✓ All levels are completely filled except possibly the last level.
- ✓ Also last level might or might not be filled completely.
- ✓ If the last level may not be full then all the nodes should be filled from the left.



Applicat[illegible]: To implem[illegible]t Heap Data structu[illegible].

10.2.2.4 Almost Complete Binary Tree

- ✓ All levels are completely filled except the last level.
- ✓ Also, the last level must not be filled completely.
- ✓ If the last level is not full, then all the nodes should be filled from the left.

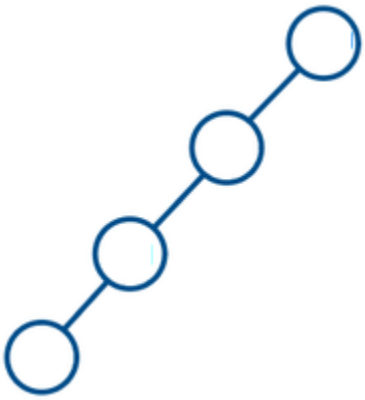


10.2.2.5 Skewed Binary Tree

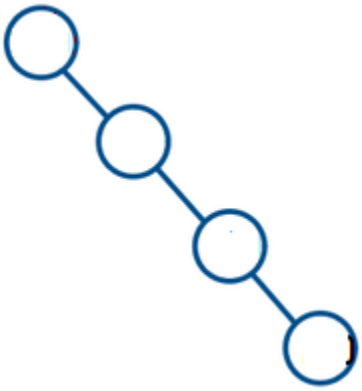
If each node in a binary tree has only one child (except leaf nodes) then this type of binary tree is called skewed trees.

If each node has only a left child then it is called left-skewed trees.

Similarly, If each node has an only right child, then it is called left-skewed trees.



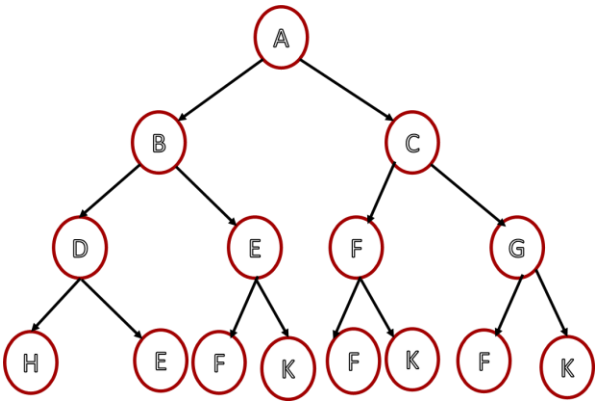
Left Skewed Tree



Right Skewed Tree

10.3 Height & number of nodes in a binary Tree

Derive an expression to define the relationship between height and the number of nodes in a binary tree.



Height at level	Number of nodes at Level h
h = 0	$2^0 = 1$
h = 1	$2^1 = 2$
h = 2	$2^2 = 4$

Therefore, total number of nodes(N) in a binary tree of height h

$= 2^0 + 2^1 + 2^2 + \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots + 2^h$

As this expression forms a G.P.

$=(2^{(h+1)} - 1)/ (2-1)$

$N = 2^{(h+1)} - 1$

$N + 1 = 2^{(h+1)}$

Taking log on both side, we get

$\log_2(N + 1) = h+1$

$$\mathbf{h = \log_2(N + 1)-1}$$

10.3.1 Height & number of nodes in a binary Tree

Case-1: Maximum number of nodes in a binary tree when height 'h' is given.

A binary tree has the maximum number of nodes, if each level in the tree has the maximum number of nodes.

Maximum number of nodes $N_{max} = 2^{(h+1)} - 1$

Case-2: Minimum number of nodes in a binary tree when height 'h' is given.

A binary tree has the minimum number of nodes, if binary tree is either left skewed or right-skewed.

Minimum number of nodes $N_{min} = h+1$

Case-3: Minimum height of a binary tree when N number of nodes are given.

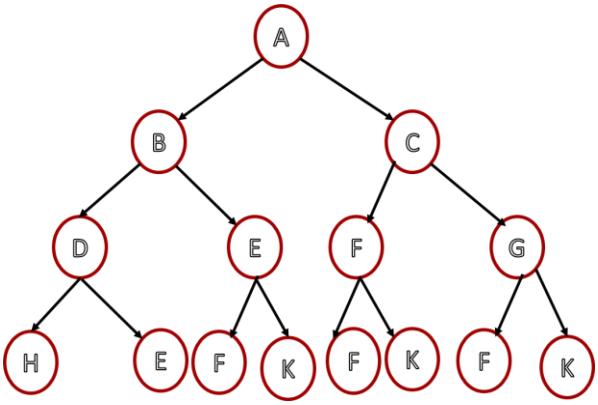
A binary tree has minimum height, if each level in the tree has maximum number of nodes.

Minimum height $h_{min} = \log_2(N + 1)-1$

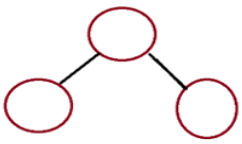
Case-4: Maximum height of a binary tree when 'n' number of nodes are given.

A binary tree has a maximum height, if the binary tree is either left skewed or right-skewed.

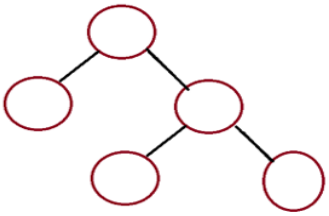
Maximum Height $h_{max} = h+1$



In a binary tree, if n0 is the number of leaf nodes, n1 is the number of nodes single children, and n2 is the number of nodes with 2 children, "n0 = n2+1".



n0 = 2
n2 = 1
n0 = n2 + 1



n0 = 3
n2 = 2
n0 = n2 +1

To prove the above identity,

Total nodes (n) = n0 + n1 + n2 -------- (1)

Total branches for n0 node = 0

Total branches for n1 node = n1

Total children for n2 nodes = 2*n2

Total nodes (n) = number of branches + 1

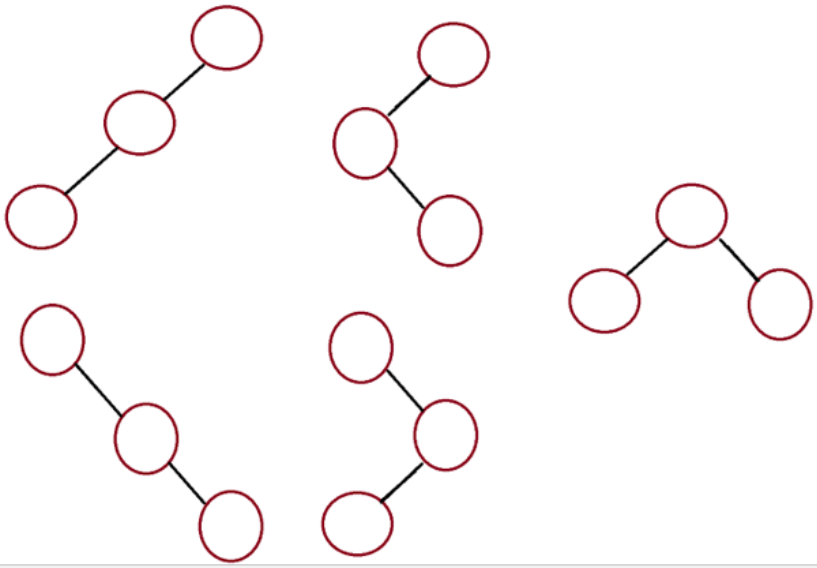
Total nodes (n) = 0+n1+2*n2+1---------- (2)
From (1) and (2)
n0+n1+n2 = 0+n1+2*n2+1
n0 = n2+1. Hence ***Proved***

10.3.2 Finding number of structurally different binary Tree

Number of binary trees with 1 node

Number of binary trees with 2 nodes

Number of binary trees with 3 nodes

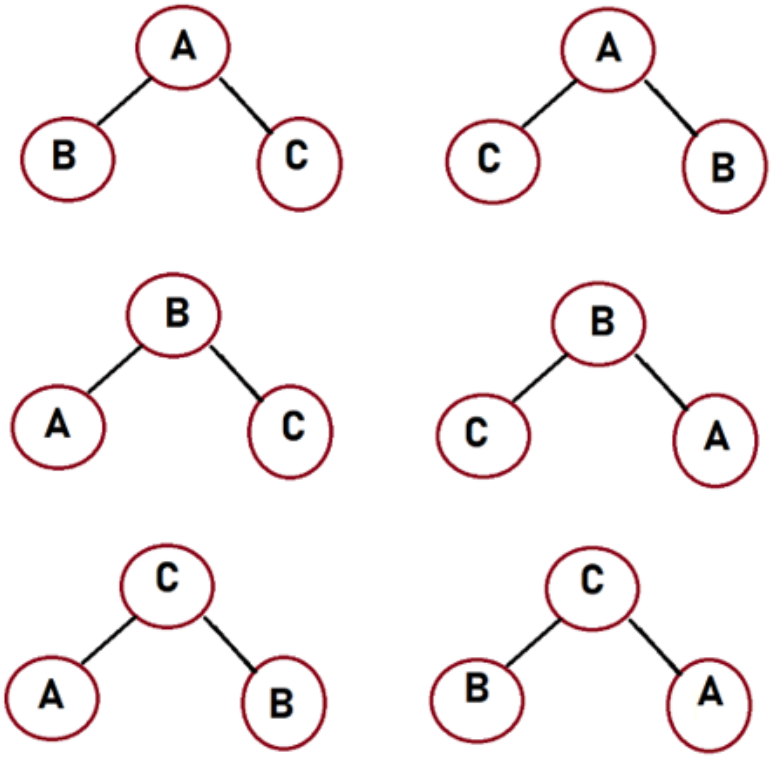


For n node, total binary trees possible are
$^{2n}C_n$ / (n+1)
With 3 nodes, possible trees are
$^{6}C_3$ / 4 = 5 (Evident from structure shown above)

10.3.3 Finding number of different binary Trees with given keys

Take an example 3 keys A, B, C. Tree of one structure possible with these keys are given below

There are 3 ways by which root node can be selected.
The left node can be selected in 2 ways, having selected the root node
Having selected the root and left child, right child can be selected in 1 way.
Total trees possible of a single structure = 3x2x1 = 3!
For an n-node binary tree, it has to be n!
Total structurally different binary tree = 2n C n / (n+1)
Total Trees with the given keys = n! * 2n C n/(n+1)
e.g. with 3 nodes A,B,C a total number of trees possible = 3! x 5 = 30



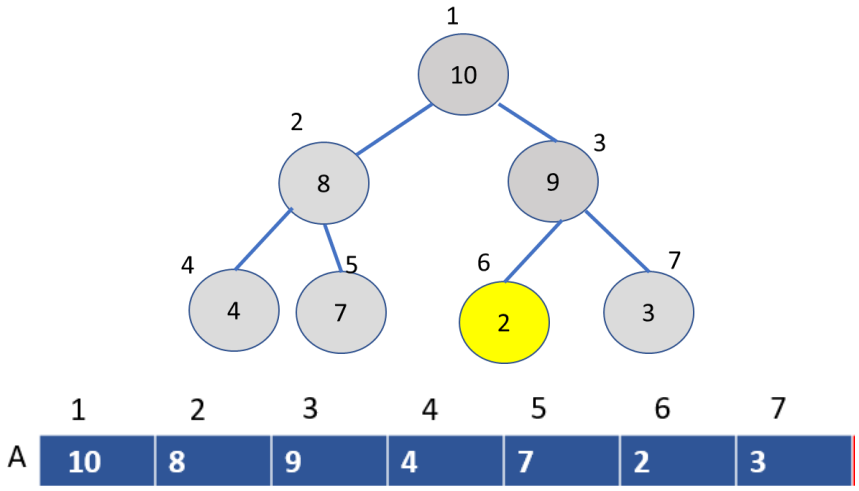
10.4 Representation of binary Tree

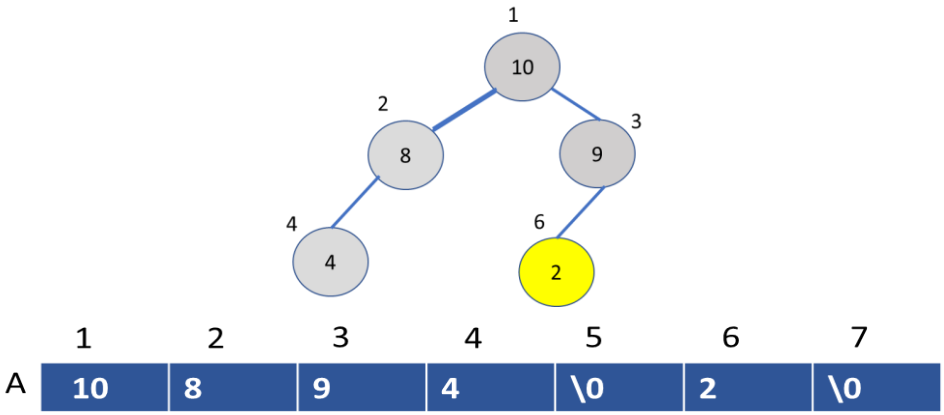
There are two ways to represent the binary tree in memory:

a) Using Array (Static Implementation)
b) Using Linked List(Dynamic Implementation)

10.4.1 Array (Static Implementation)

The binary tree is represented using one-dimensional array in which each element or node is numbered sequentially level by level from left to right. Also, empty nodes are numbered.





Here index zero of the array can be used to store the total number of elements.
Here all non-existing children are represented by "\0" in the array.
Index of the left child of node having i = 2*i
Index of right child of node having index i =2*i +1
Index of parent of the node having index i = i/2
Also, the index of Sibling of any node i will be i+1, if i is a left child of its parent.
Similarly, the index of Sibling of any node i will be i-1, if i is the right child of its parent.

10.4.2 Linked List(Dynamic Implementation)

In the linked representation, the binary tree is a collection of node where each node has at least three fields:

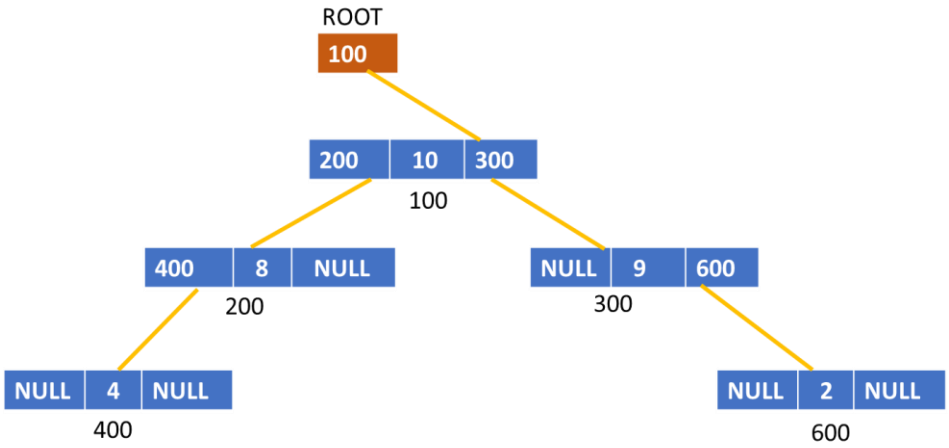
i) Data
ii) Address of the left child
iii) Address of the right child

Left Child Address field	Data	Right Child Address Field

Here Left address field contains the address of the left child and the right address field contains the address of right child.

If node has no left child, then the left address field contains NULL.

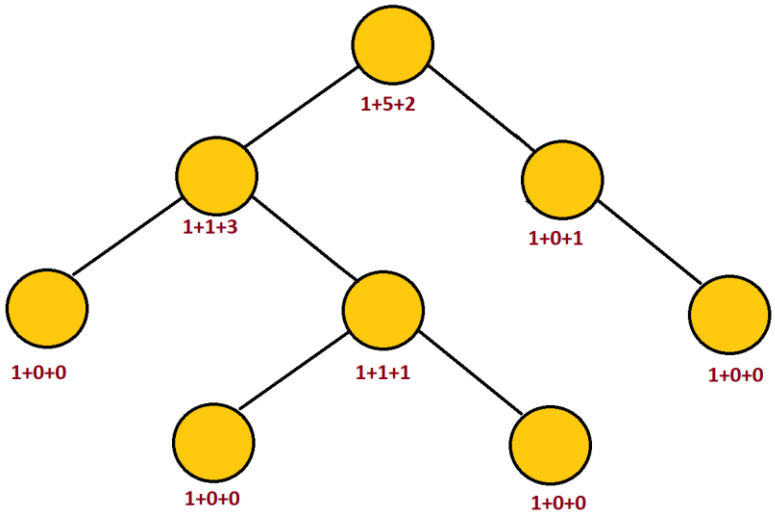
If node has no right child, then the right address field contains NULL.



10.5 Binary Tree Algorithms

10.5.1 Finding Count of Nodes in the Tree

If the node does not exist, its counting is updated as zero. If the node exists, then the number of nodes considering that as root node will be the 1 (the count of that node) plus the nodes in the Left subtree and Right subtree of that node computed recursively.



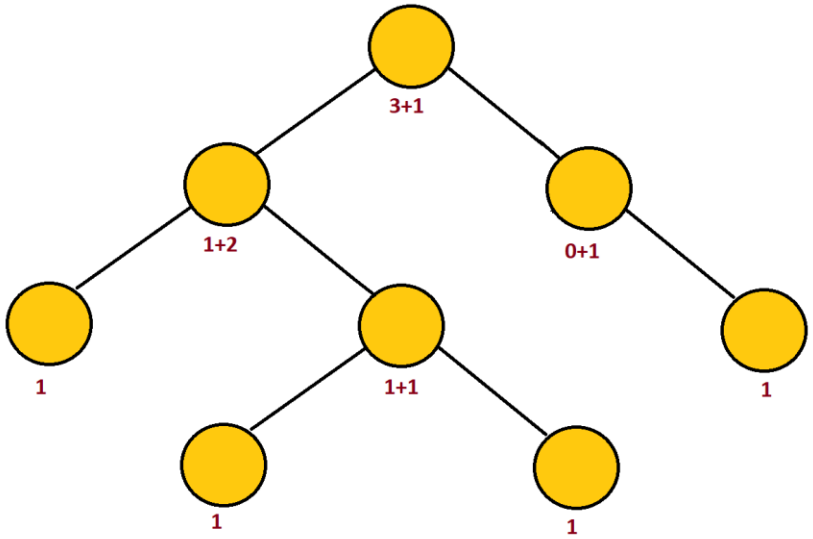
ALGORITHM CountNodes(Root)

BEGIN:

```
        IF Root == NULL THEN
                RETURN 0
        ELSE
                RETURN 1+CountNodes(Root →Left) + CountNodes(Root →Right)
END;
```

10.5.2 Finding the count of Leaf Nodes

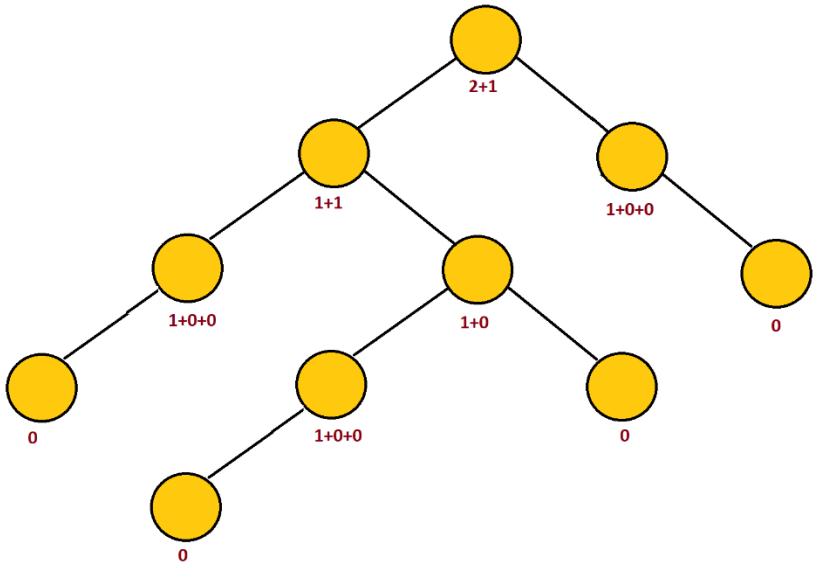
If the node does not exist, the count of the leaf will be 0. If the left and right of a node are NULL, that will be the leaf node, and its counting is updated as 1. If the node exists and it is not the leaf node, then the number of leaf nodes considering that as root node will be the sum of leaf nodes on the Left subtree and Right subtree of that node computed recursively.




```
ALGORITHM CountLeafNodes(Root)
BEGIN:
        IF Root == NULL THEN
                RETURN 0
        ELSE
                IF Root→Left==NULL AND Root→Right ==NULL THEN
                        RETURN 1
                ELSE
                        RETURN CountLeafNodes(Root→Left)+CountLeafNodes(Root→Right)
END;
```

10.5.3 Finding the count of Nodes having one Child

If the node does not exist, the count of N1 nodes will be 0. If the left and right of a node are NULL, that will be the leaf node, and it cannot be the N1 node; hence count is updated as 0. If the node exists and it's one of the child is NULL, then it is definitely the N1 Node and its counting is taken as 1, but there could be more N1 nodes in the subtree starting from that node (found recursively). If both the children exist for the given node, that node cannot be N1 node, but there could be some N1 nodes in the subtree starting from there. These are counted recursively by counting the sum of N1 nodes on the Left subtree and Right subtree of that node.



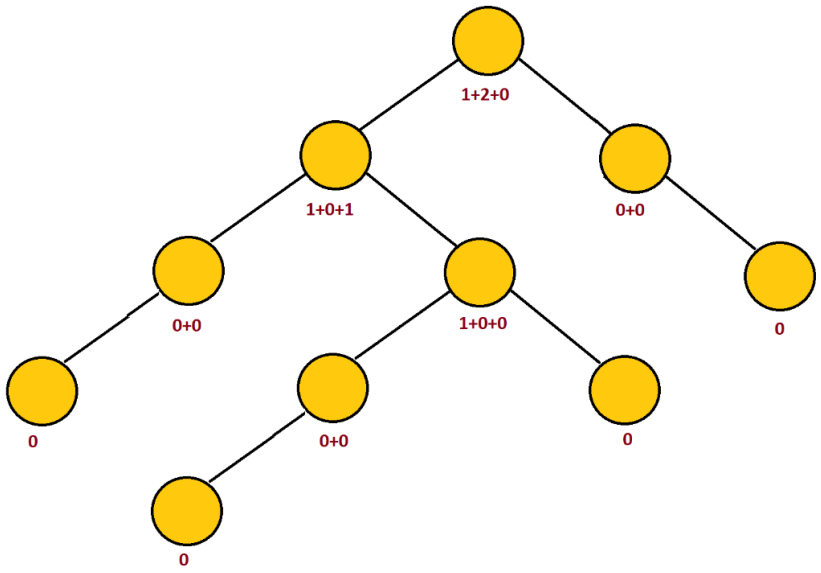
ALGORITHM CountN1Nodes(Root)
BEGIN:

```
IF Root == NULL THEN
    RETURN 0
ELSE
  IF Root→Left==NULL AND Root → Right ==NULL THEN
            RETURN 0
  ELSE
      IF Root→Left!=NULL AND Root → Right !=NULL THEN
            RETURN CountN1Nodes(Root →Left) + CountN1Nodes(Root→Right)
       ELSE
            RETURN 1+ CountN1Nodes(Root→Left) +CountN1Nodes(Root→Right)
```

END;

10.5.4 Finding the count of Nodes having Two Children

If the node does not exist, the count of N2 nodes will be 0. If the left and right of a node are NULL, that will be the leaf node and it cannot be the N2 node, hence count is updated as 0. If both the children exist for the given node then it is definitely the N2 Node and its counting is taken as 1, but there could be more N2 nodes in the subtree starting from that node (found recursively). If the node exists and its one of the child is NULL, that node cannot be N2 node, but there could be some N2 nodes in the subtree starting from there, these are counted recursively by counting the sum of N2 nodes on the Left subtree and Right subtree of that node.



ALGORITHM CountN2Nodes(Root)

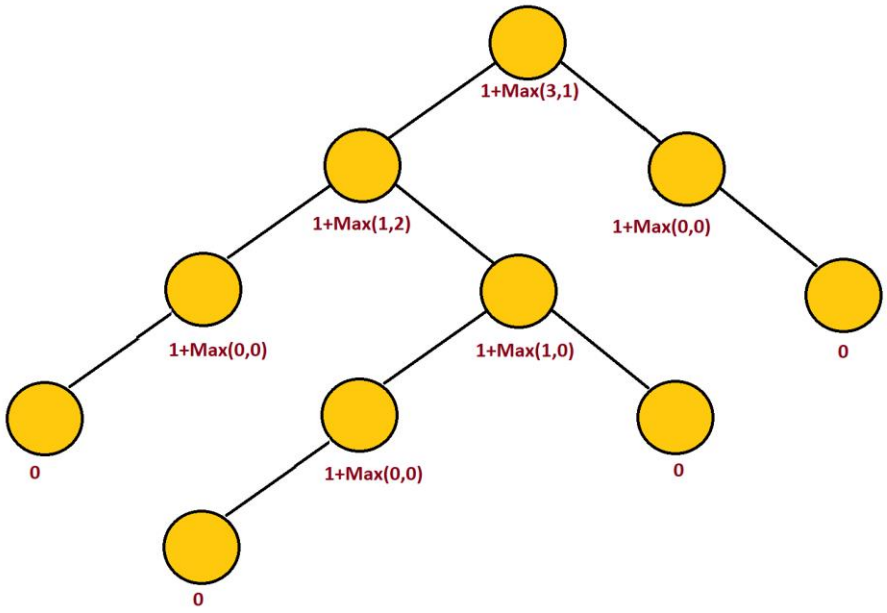
BEGIN:

```
IF Root == NULL THEN
        RETURN 0
ELSE
    IF Root→Left==NULL AND Root → Right ==NULL THEN
        RETURN 0
    ELSE
        IF Root→Left!=NULL AND Root → Right !=NULL THEN
            RETURN 1+ CountN1Nodes(Root→Left)+CountN1Nodes(Root→Right)
        ELSE
            RETURN CountN1Nodes (Root→Left) + CountN1Nodes (Root→Right)
```

END;

10.5.5 Finding Height of a node

If the node does not exist, its height is counted as zero. If a node is a left node, its height is counted as 0. Otherwise, find the height of the left subtree and right subtree recursively, take a maximum of both and add 1 to it. This will be the height of the given node.



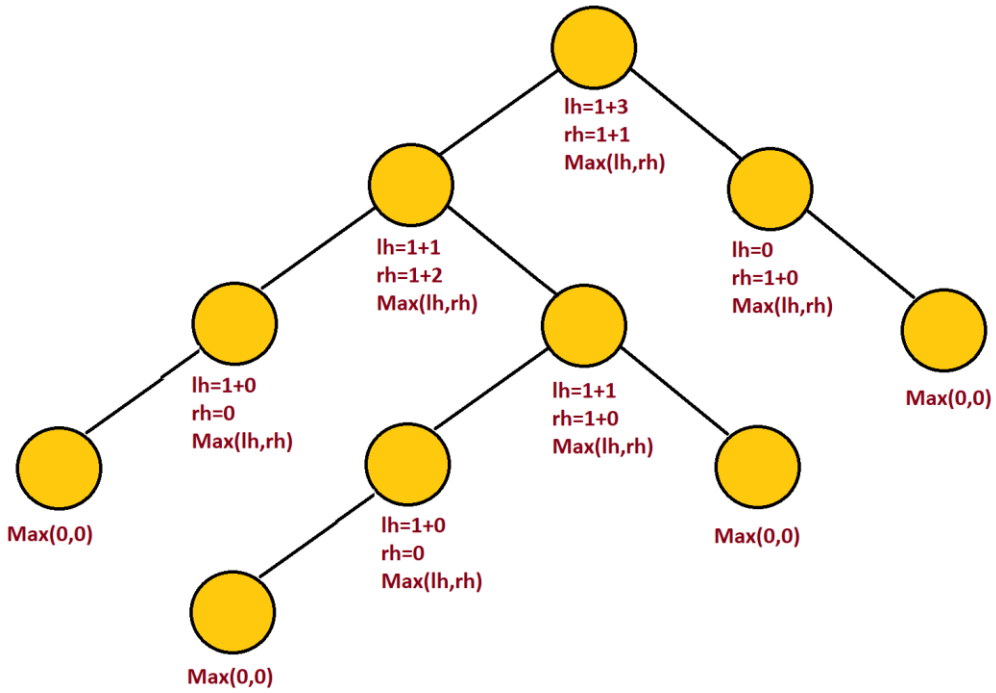
ALGORITHM Height (Root)

BEGIN:

```
IF Root == NULL THEN
        RETURN 0
ELSE
        IF Root→Left==NULL AND Root → Right ==NULL THEN
                RETURN 0
        ELSE
                RETURN 1+ Height(Root →Left) + Height(Root →Right)
```

END;

Method 2 (Through Left height and right height)



ALGORITHM Height (Root)

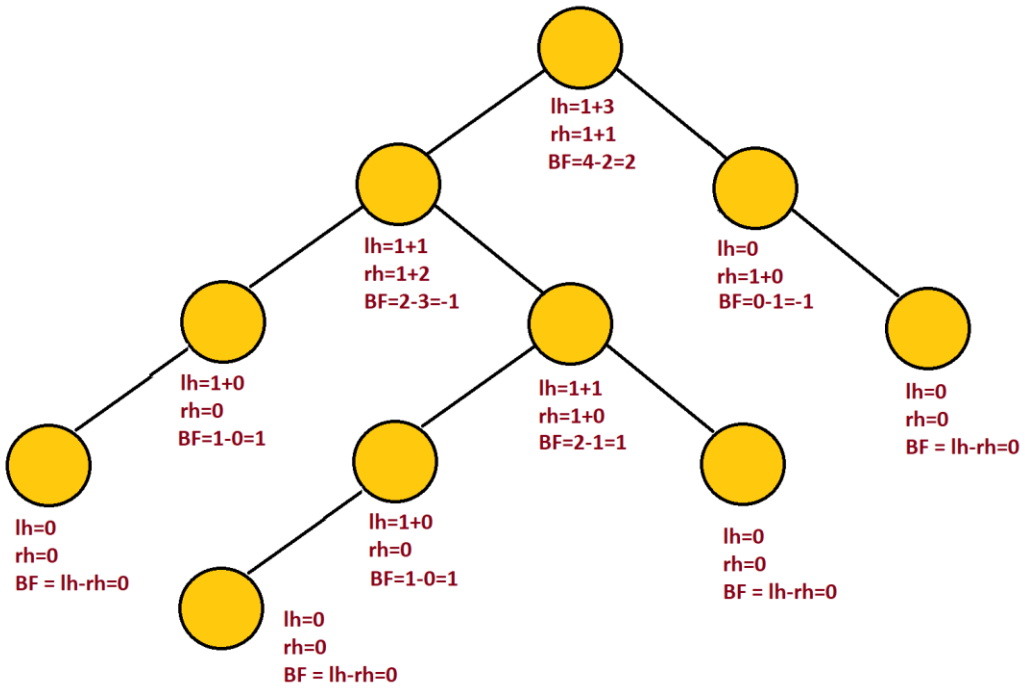
BEGIN:

```
IF Root == NULL THEN
        RETURN 0
ELSE
        IF Root→Left == NULL THEN
                Lh =0
        ELSE
                Lh = 1+Height(Root→Left)

        IF Root→Right == NULL THEN
                Rh =0
        ELSE
                Rh = 1+Height(Root→Left)

        RETURN Maximum(Lh,Rh)
```

END;

10.5.6 Finding Balance Factor of a Node

The balance factor of a node is the difference of height of the node on the left subtree and right subtree.



ALGORITHM BalanceFactor (Root)

BEGIN:

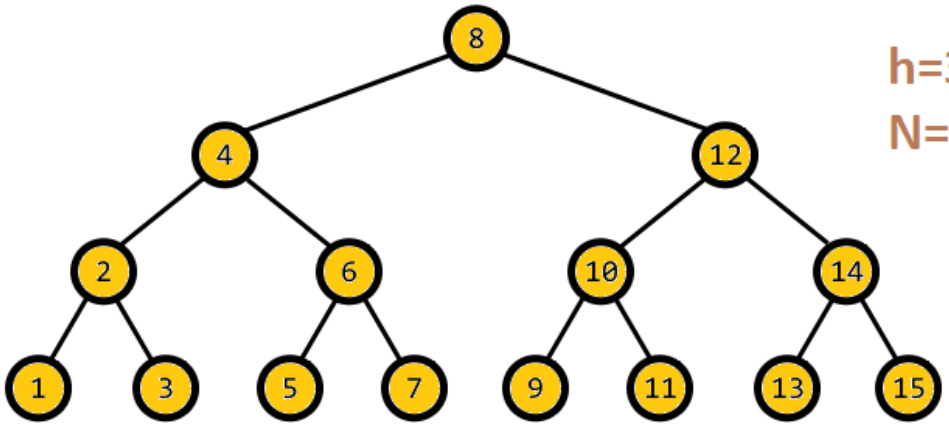
```
IF Root == NULL THEN
        RETURN 0
ELSE
        IF Root→Left == NULL THEN
                Lh =0
        ELSE
                Lh = 1+Height(Root→Left)

        IF Root→Right == NULL THEN
                Rh =0
        ELSE
                Rh = 1+Height(Root→Left)

        RETURN Lh - Rh
```

END;

10.5.7 Finding if the given Binary Tree is complete

A binary tree is complete if this relation is satisfied between number of nodes (N) and Height (H)

$N = 2^{(H+1)} - 1$



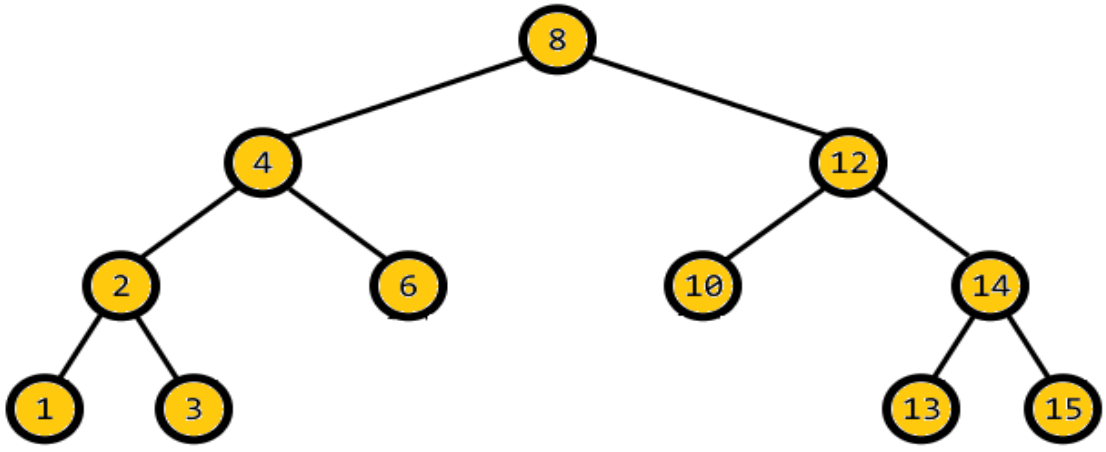
$h=3$

$N=2^{3+1}-1 = 15$

```
ALGORITHM IsComplete (Root)
BEGIN:
        N=CountNodes(Root)
        H=Height(Root)
        IF Power(2, H+1) – 1 == N THEN
                RETURN TRUE
        ELSE
                RETURN FALSE
END;
```

10.5.8 Find if Binary Tree is strictly

A Binary Tree is considered a strictly binary tree if all the nodes in the tree have either zero child or two child. This can be computed in 2 ways. In the first, find out N1 nodes by the function defined in the above text. If the count of N1 nodes is zero, that tree is definitely strictly. The second method recursively checks the condition of both the children recursively.



ALGORITHM IsStrictly (Root)
BEGIN:

```
        IF Root == NULL THEN
                RETURN TRUE
        ELSE
                IF Root→Left == NULL AND Root→Right == NULL THEN
                        RETURN TRUE
                ELSE
                        IF Root→Left!=NULL AND Root→Right !=NULL THEN
                                RETURN IsStrictly(Root→Left) AND IsStrictly(Root→Right)
                        ELSE
                                RETURN FALSE
END;
```

10.6 Creation of Binary Tree

The given function creates the binary tree by asking the user about its structure. First, it enquires about the root node, then about the left, and finally about the right subtree. For every newly created node, the function recursively calls itself to enquire about the left and right subtree of that node.

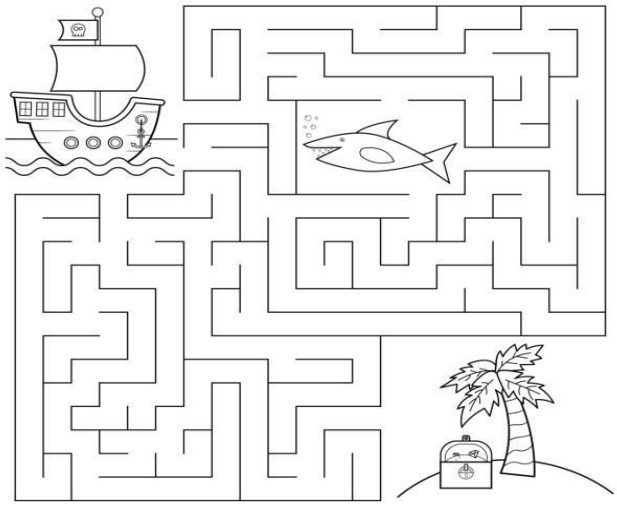
```
ALGORITHM CreateBinaryTree(root)
BEGIN:
        IF root==NULL THEN
                /***** Enquiry about the root Node *****/
                WRITE("Input the data of root node ");
                READ(x);
                q=GetNode(x)
                root=q;
                CreateBinaryTree(root);
        ELSE
                /***** Enquiry about the left Node *****/
                WRITE("Whether left of root exists?(1/0)")
                READ(choice)
                IF choice==1 THEN
                        WRITE("Input the data of left node ");
                        READ(x);
                        q=GetNode(x)
                        root->left=q;
                        CreateBinaryTree(root);
                /***** Enquiry about the right Node *****/
```

```
        WRITE("Whether right of root exists?(1/0)")
        READ(choice)
        IF choice==1 THEN
                WRITE("Input the data of right node ");
                READ(x);
                q=GetNode(x)
                root->right=q;
                CreateBinaryTree(root);
END;
```

10.7 Tree Traversals

DOM is the real-world example, where the concept of tree traversal is used. Here DOM is organized as a tree, and the node of the document which itself is present at the root node while we search any element by using keyword document.getElementbyID.
There are two approaches that are used while performing traversing in the tree. These are based upon DFS and BFS. With the help of DFS we, can do pre-order, in order, and post order traversing and with the help of BFS, we can do Level order traversing in a tree. Let us see these techniques one by one briefly: -

DFS: In Depth First Search, we go to the deepest level of the tree, and once we reached the node that is not having any children, we back track till the point, we find the node whose child is yet to be traversed. This technique is used in pre-order traversal, post-order traversal and in order traversal.
Analogy: - Just like a maze game, we start with a given direction and kept on traversing that path till we found a dead end. Sooner we find the dead-end, we backtrack till the point where another path can be found.



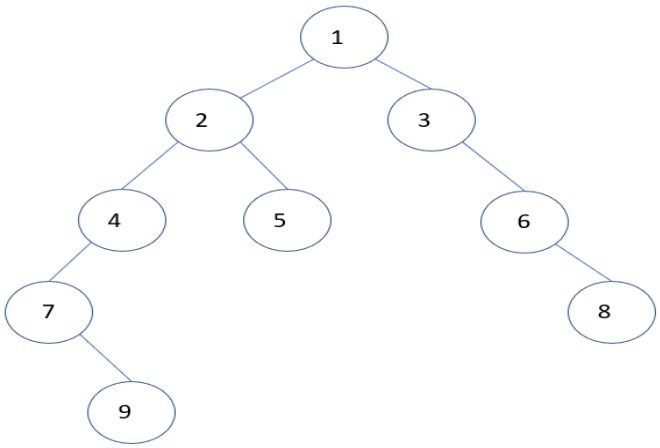
BFS: In BFS, we are traversing the nodes level by level. We are traversing any child of node only after traversing all its siblings.
Example: - Level order traversal.

10.7.1 Pre-order Traversing

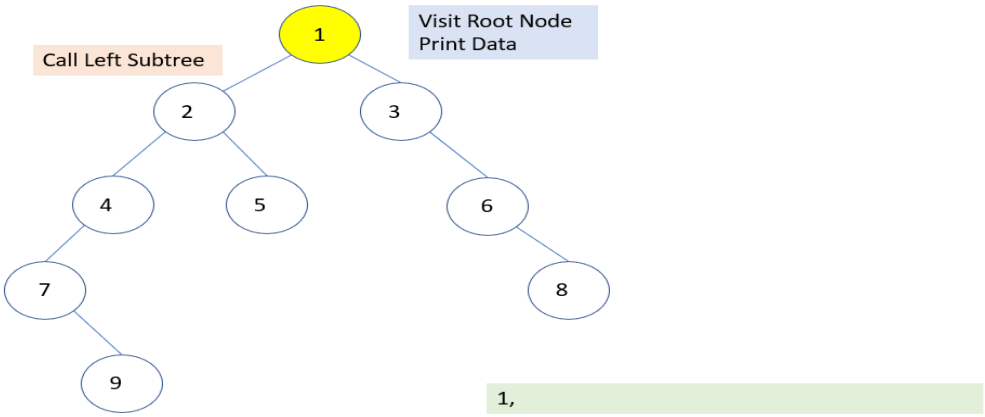
Pre-order traversing is used to find prefix expression. Pre-order traversing is done by using the following steps. These are: -
1) Visit the Root Node and print the data part.
2) Recursively traverse the left subtree.
3) Recursively Traverse the Right subtree.

Diagrammatic Representation

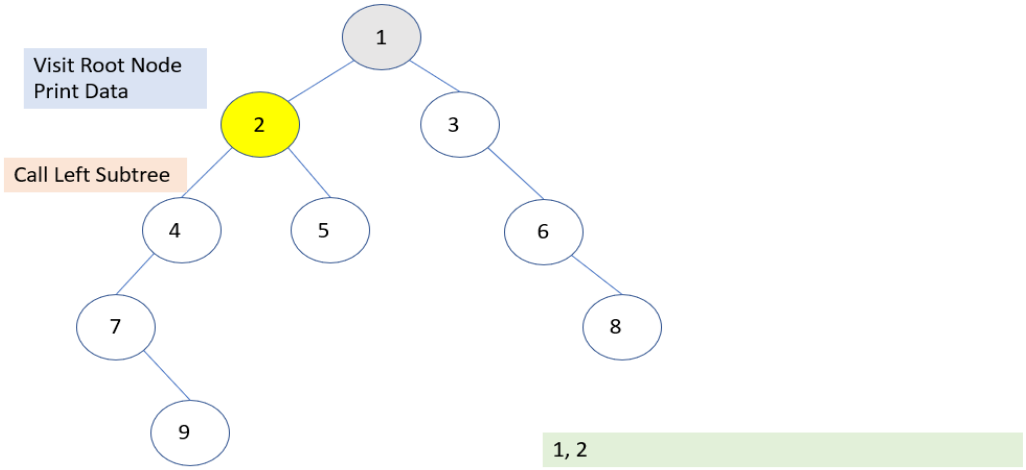
Step 1



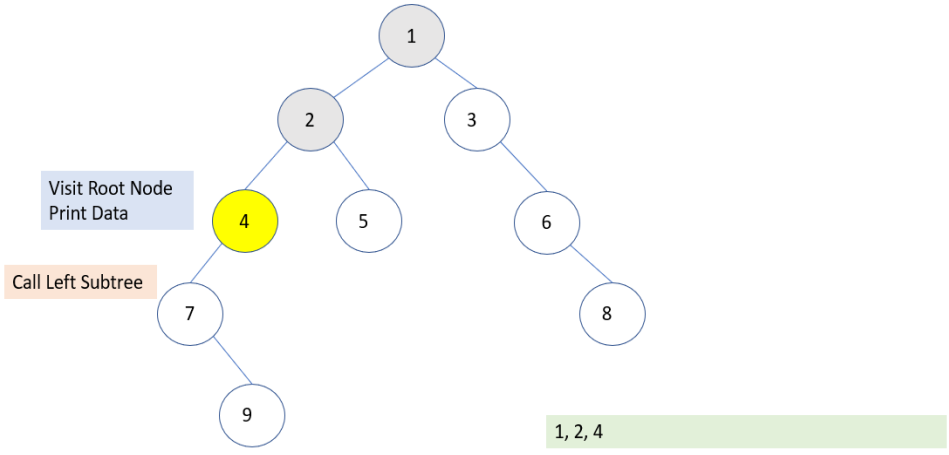
Step 2



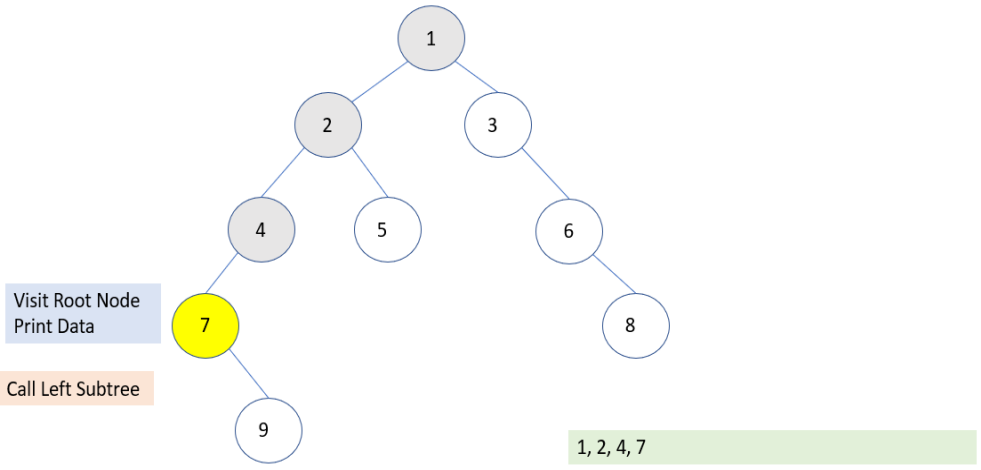
Step 3



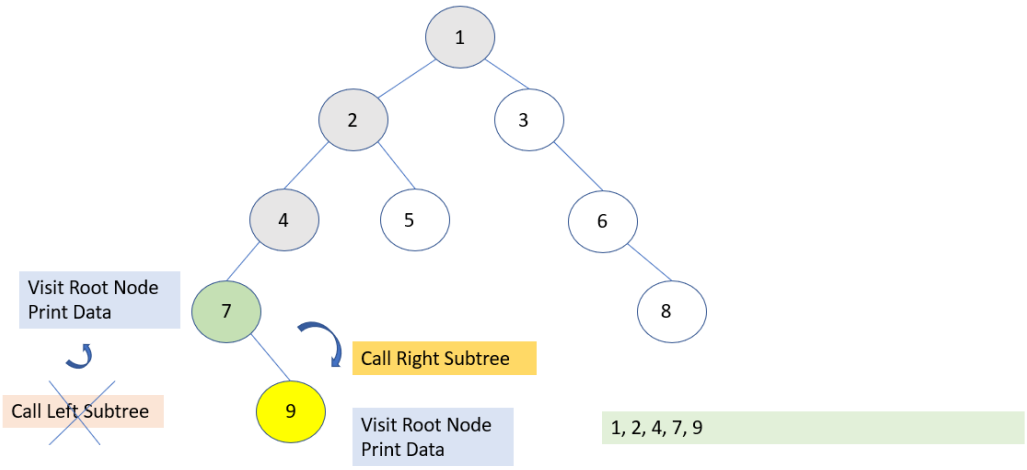
Step 4



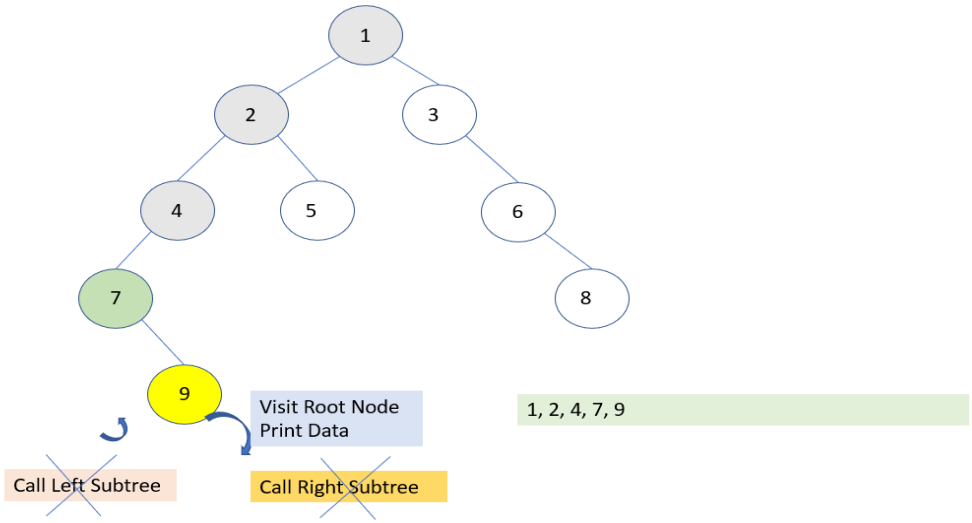
Step 5


1
2
3
4
5
6
7
8
9
Visit Root Node
Print Data
Call Left Subtree
1, 2, 4, 7

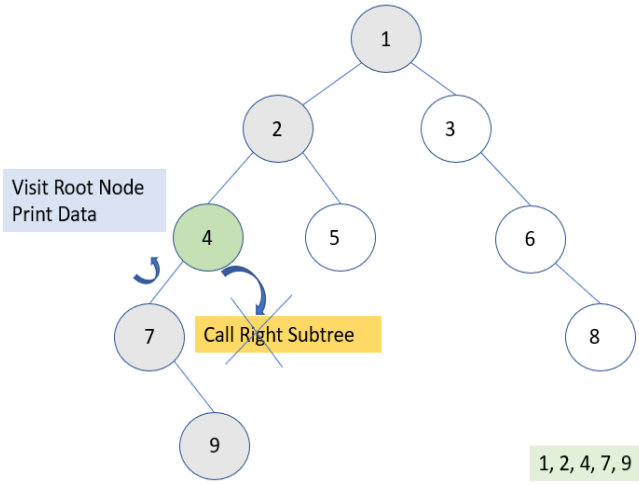
Step 6


1
2
3
4
5
6
7
8
9
Visit Root Node
Print Data
Call Left Subtree
Call Right Subtree
Visit Root Node
Print Data
1, 2, 4, 7, 9

Step 7

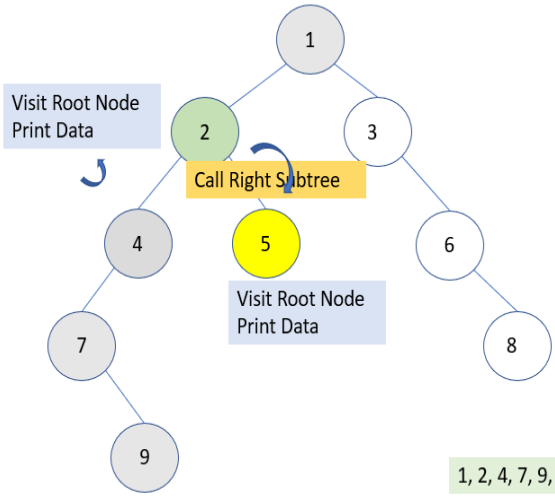

1
2
3
4
5
6
7
8
9
Call Left Subtree
Visit Root Node
Print Data
Call Right Subtree
1, 2, 4, 7, 9

Step 8



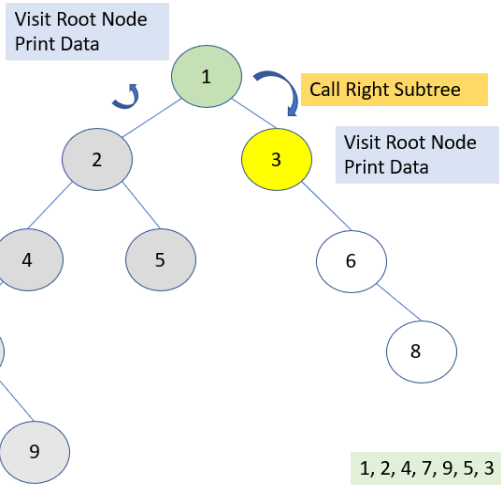
1, 2, 4, 7, 9

Step 9



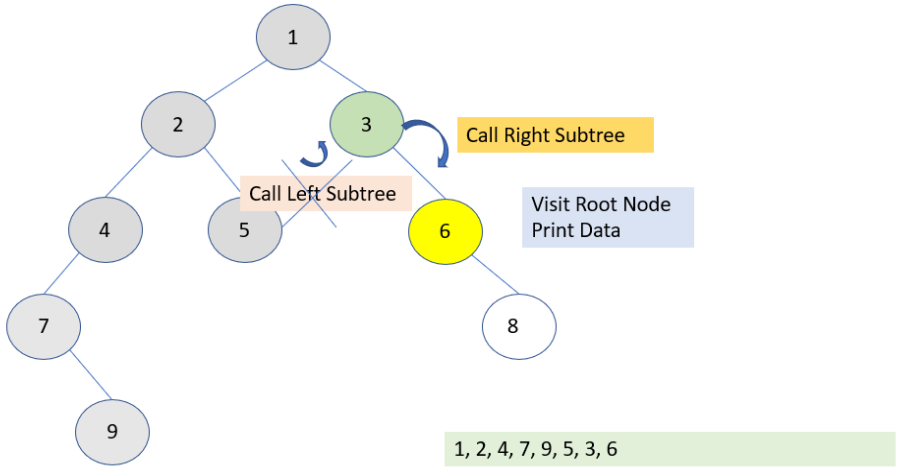
1, 2, 4, 7, 9, 5

Step 10

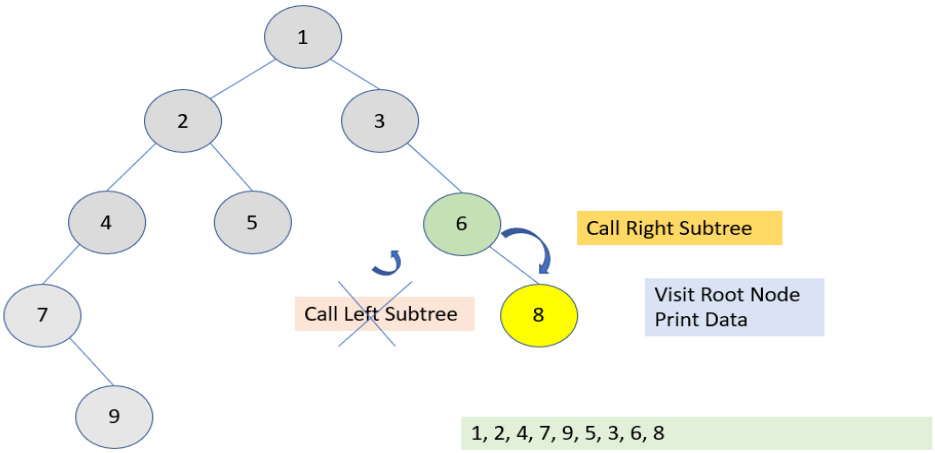


1, 2, 4, 7, 9, 5, 3

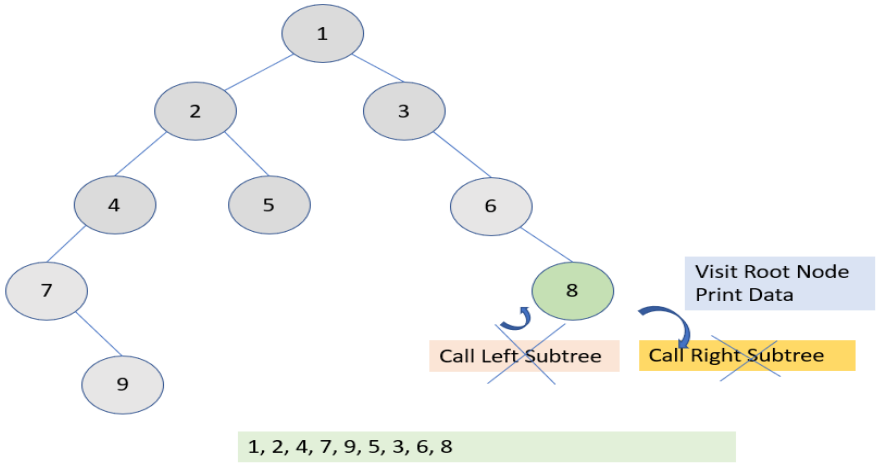
Step 11



Step 12



Step 13



ALGORITHM Preorder_traversal(Root)

BEGIN:

```
IF root == NULL THEN
        RETURN 0
ELSE
        WRITE (root → Data)
        Preorder_Traversal(root→ Left)
        Preorder_Traversal(root → Right)
```

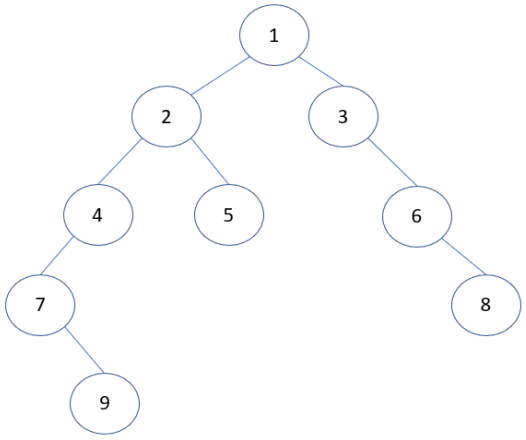
END;

10.7.2 In-order Traversing

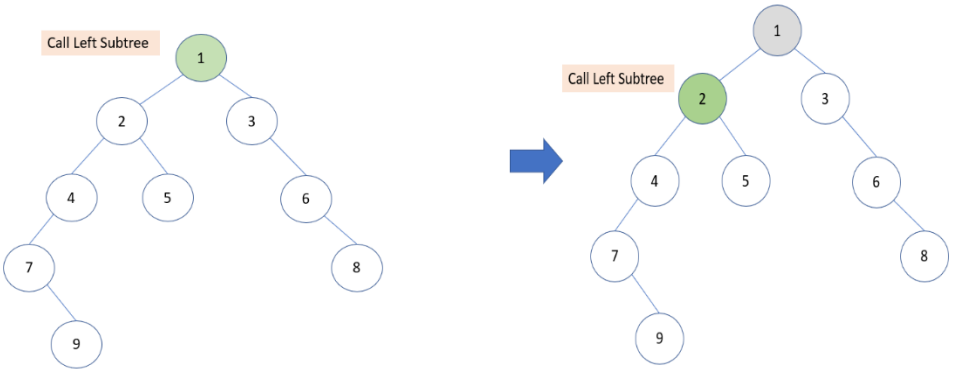
In In-order traversing, we compute the following steps. These are

1) Recursively visit Left Subtree
2) Visit Root Node. Write data part.
3) Recursively call Right Subtree

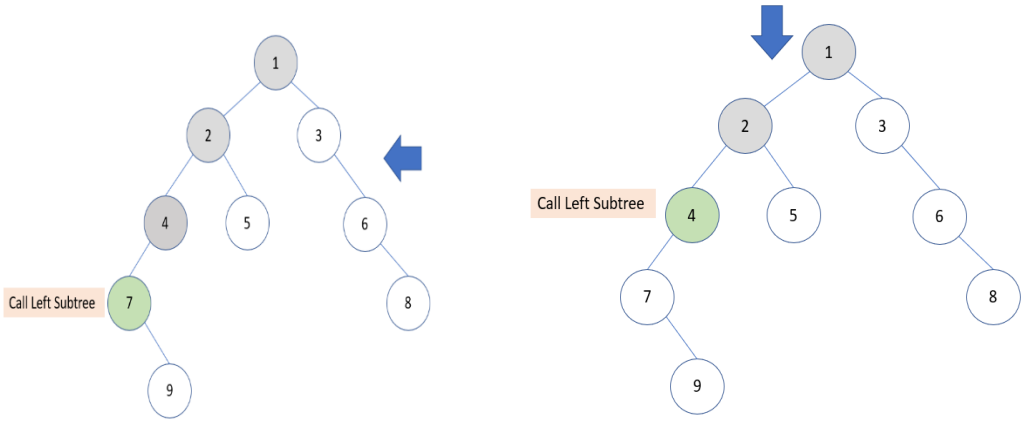
Diagrammatic Representation



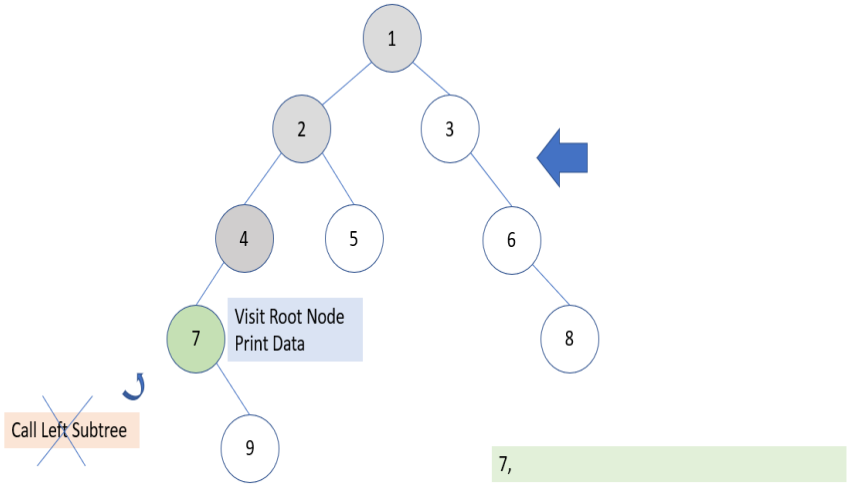
Step 2 **Step 3**



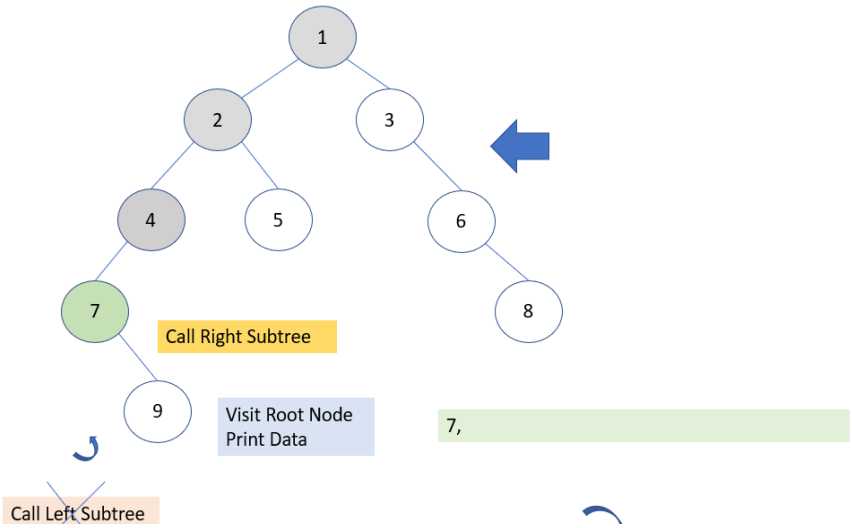
Step 5 **Step 4**



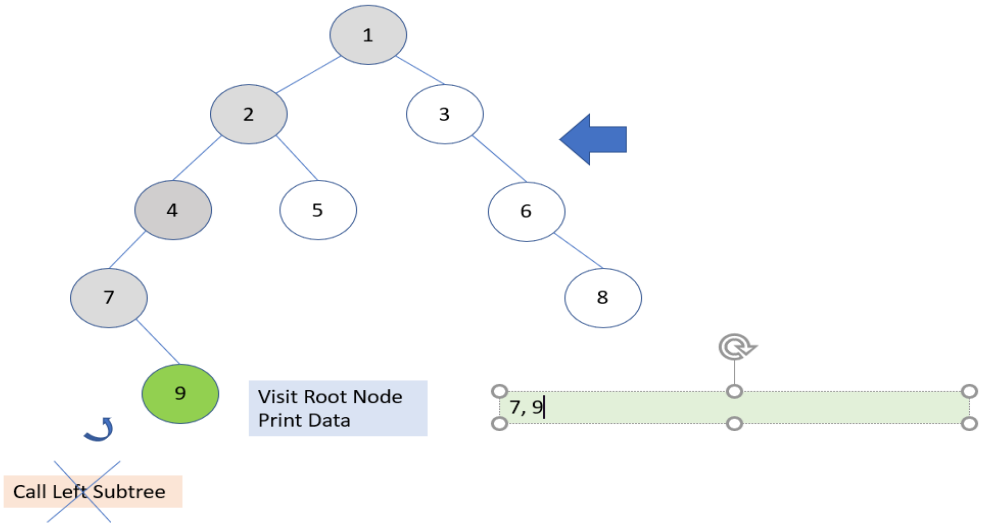
Step 6



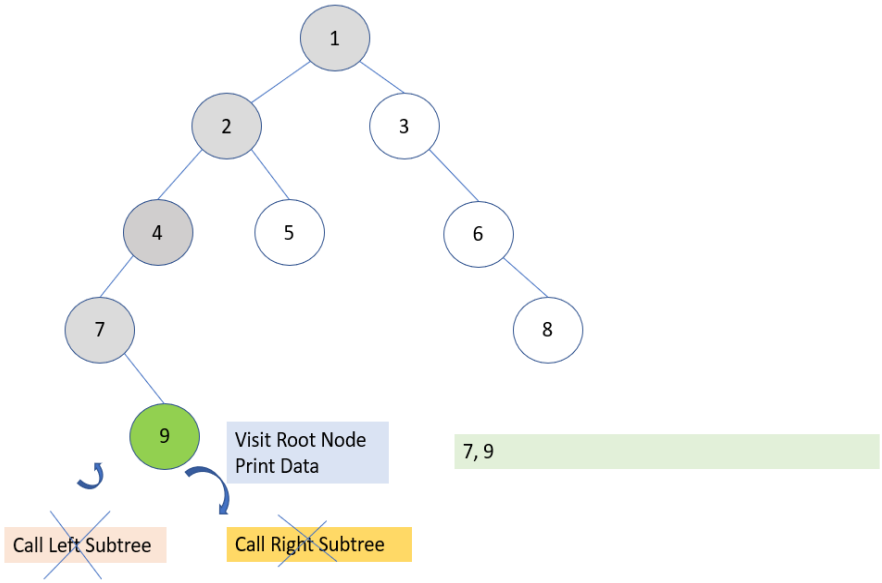
Step 7



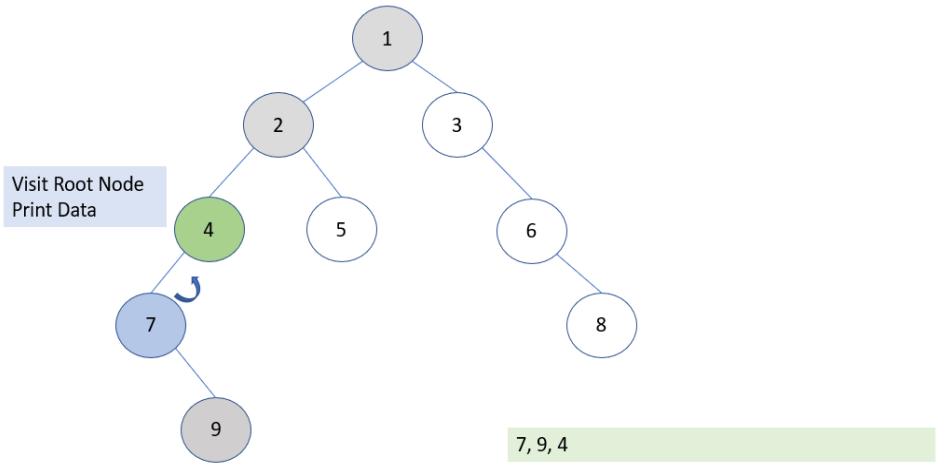
Step 8



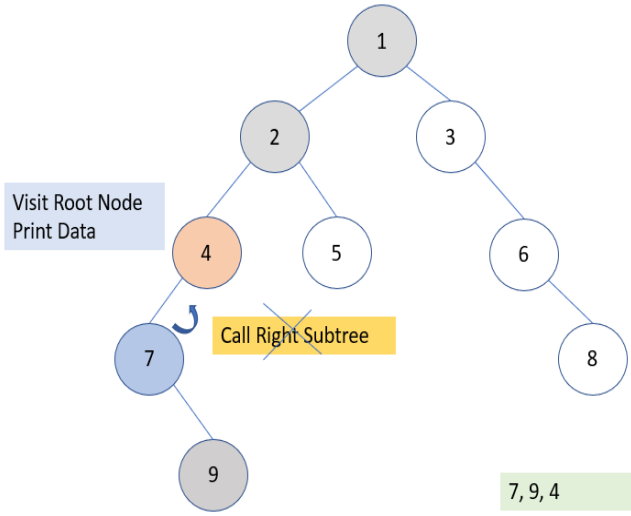
Step 9



Step 10

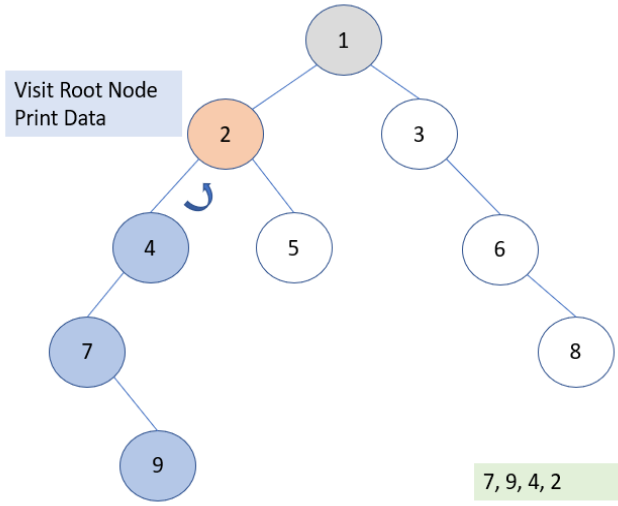


Step 11



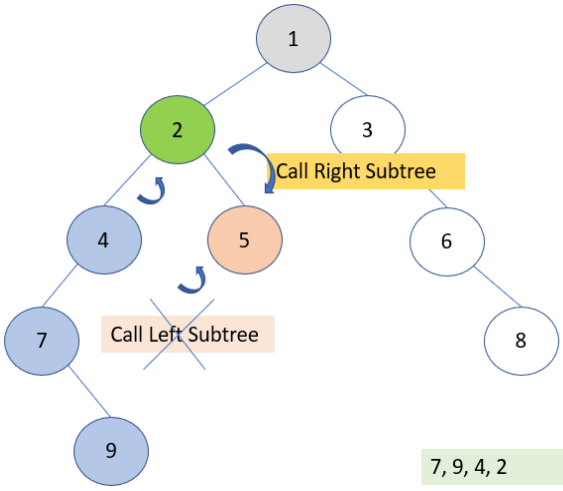
7, 9, 4

Step 12



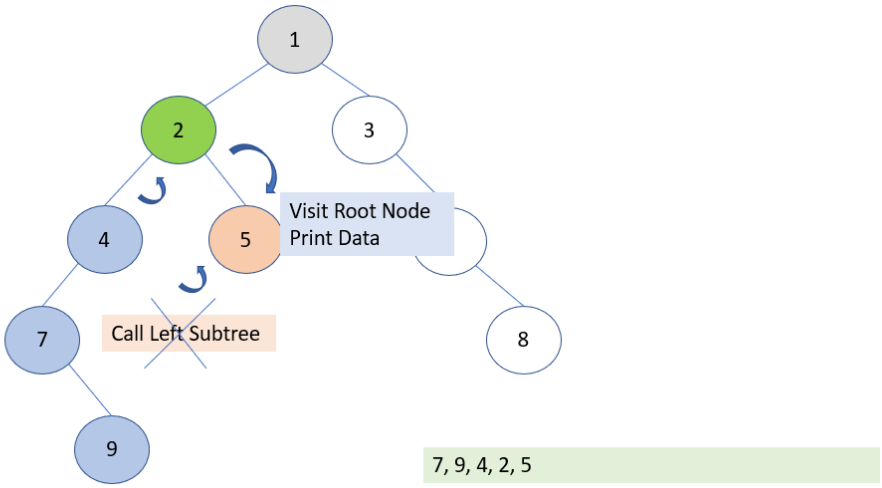
7, 9, 4, 2

Step 13

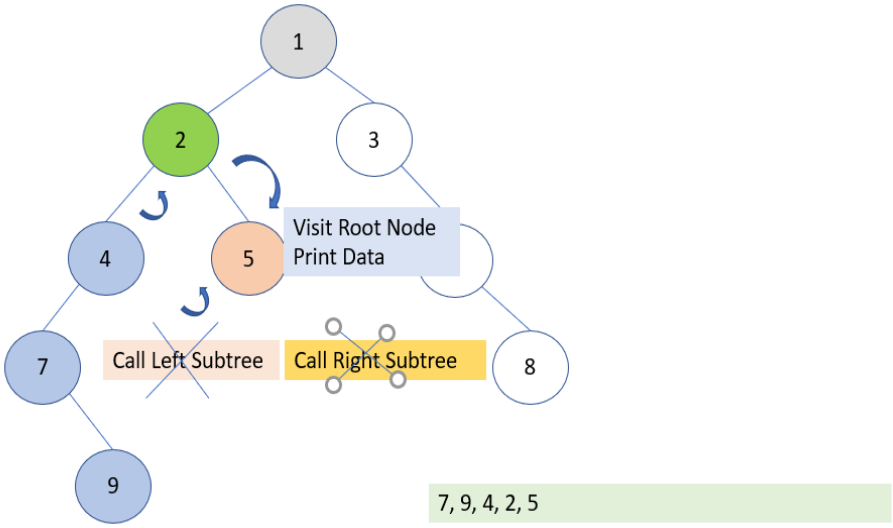


7, 9, 4, 2

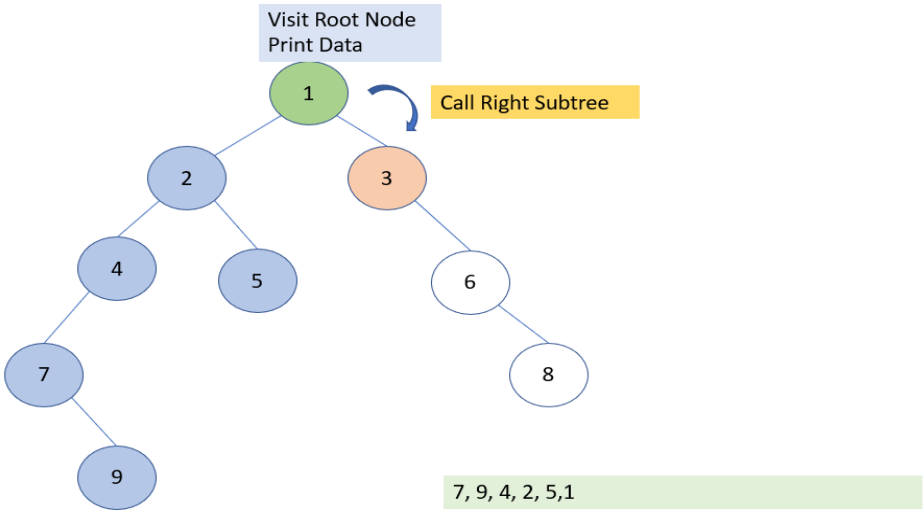
Step 14



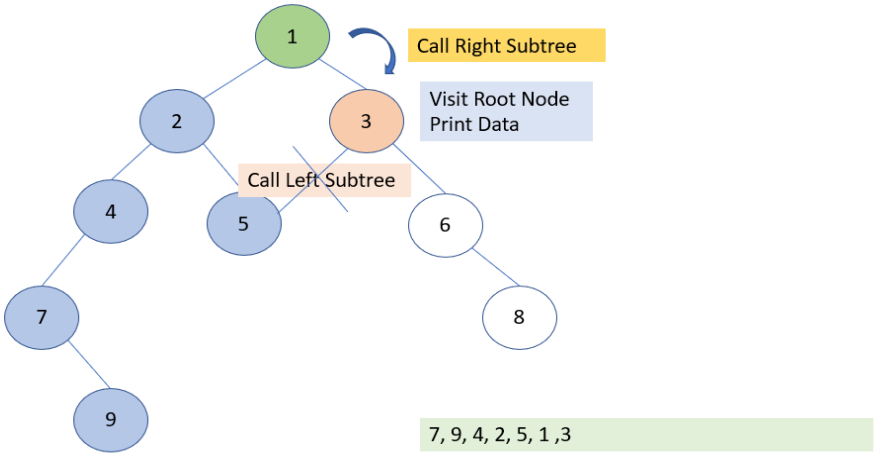
Step 15



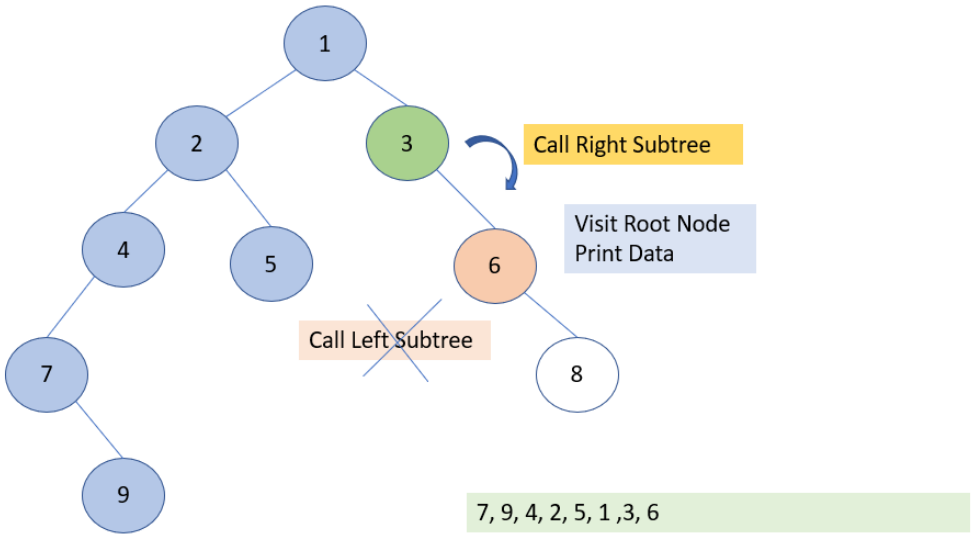
Step 16



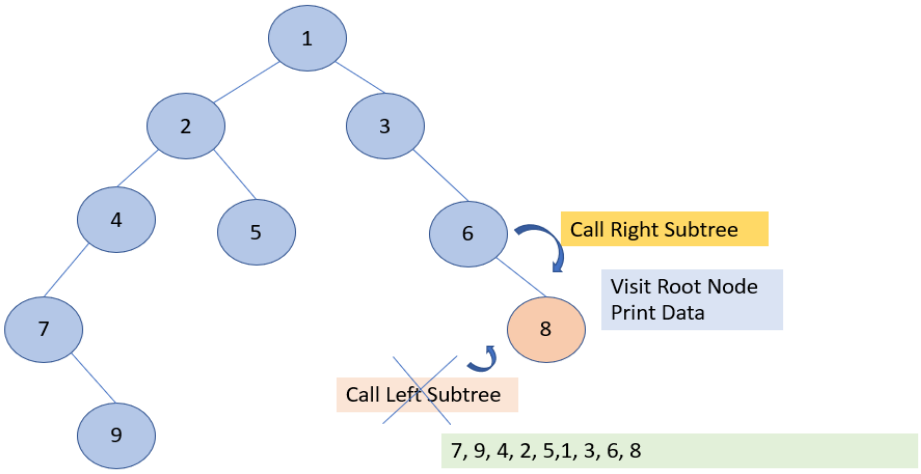
Step 17



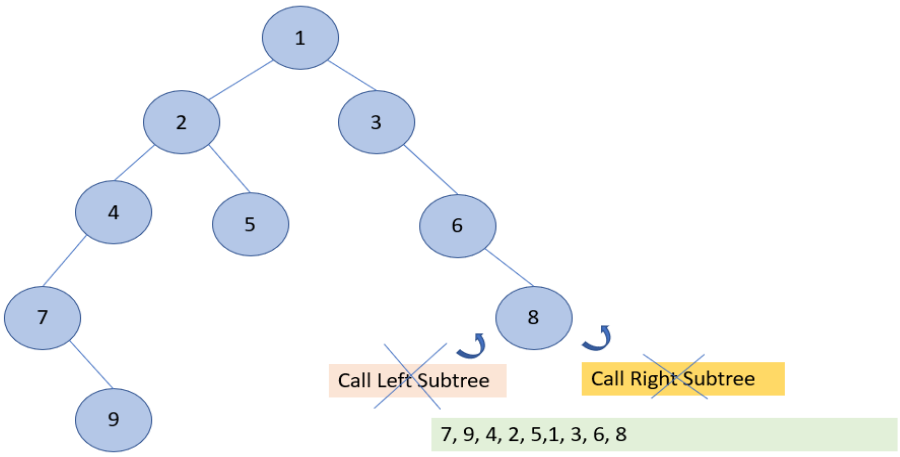
Step 18



Step 19



Step 20



ALGORITHM Inorder_traversal(Root)
BEGIN:

```
IF root == NULL THEN
        RETURN 0;
ELSE
        Inorder_traversal(root→Left)
        WRITE (root→Data)
        Inorder_traversal(root→Right)
```

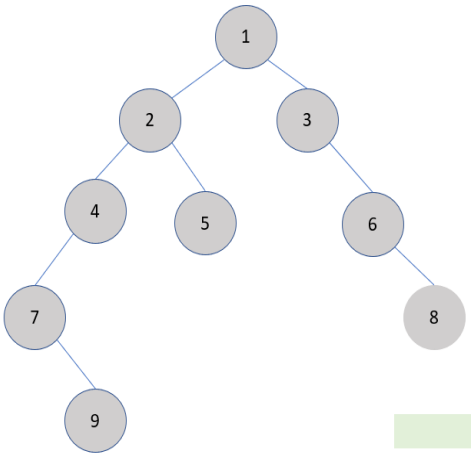
END;

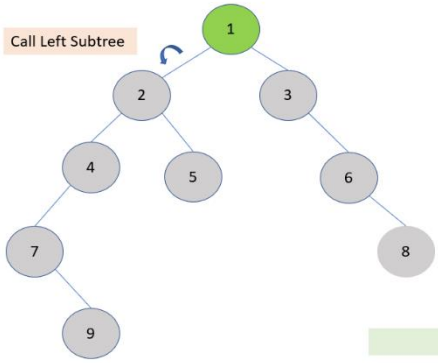
10.7.3 Post order Traversing

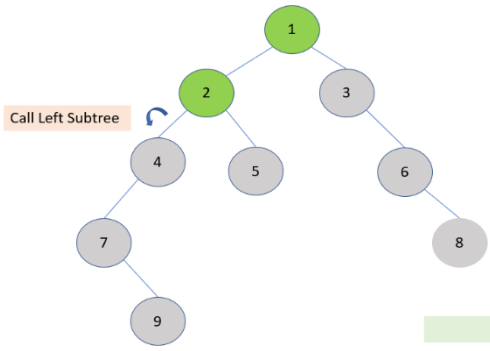
In Post order traversing, we compute the following steps. These are

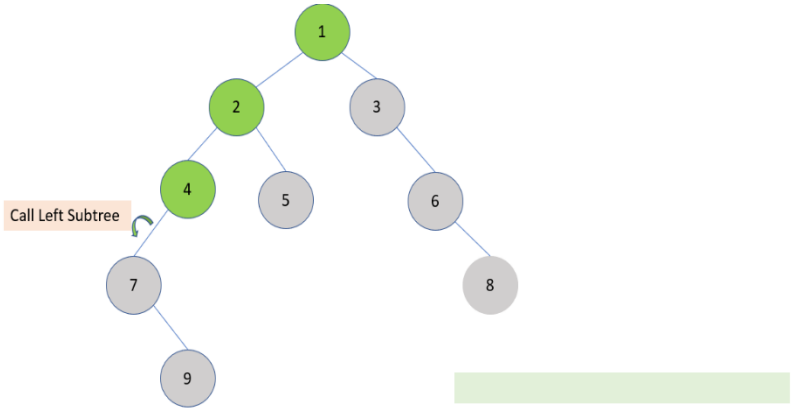
1) Recursively visit Left Subtree
2) Recursively call Right Subtree
3) Visit Root Node. Write data part.

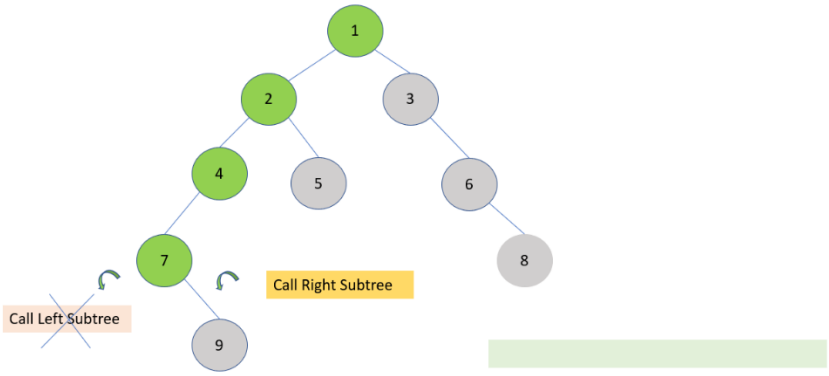
Diagrammatic Representation

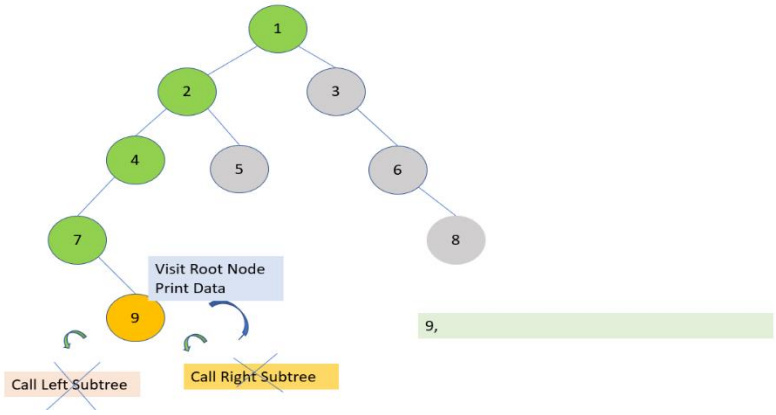


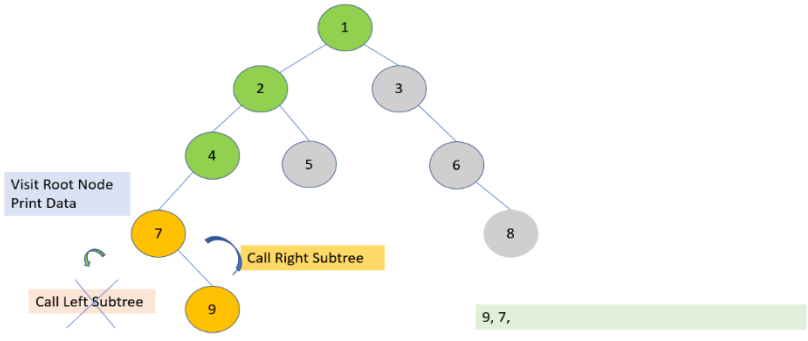


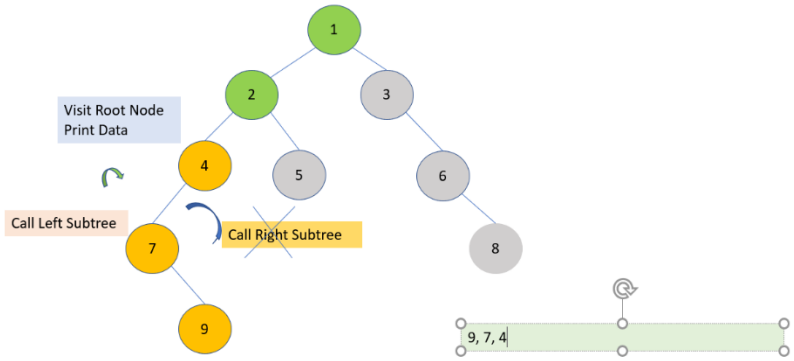


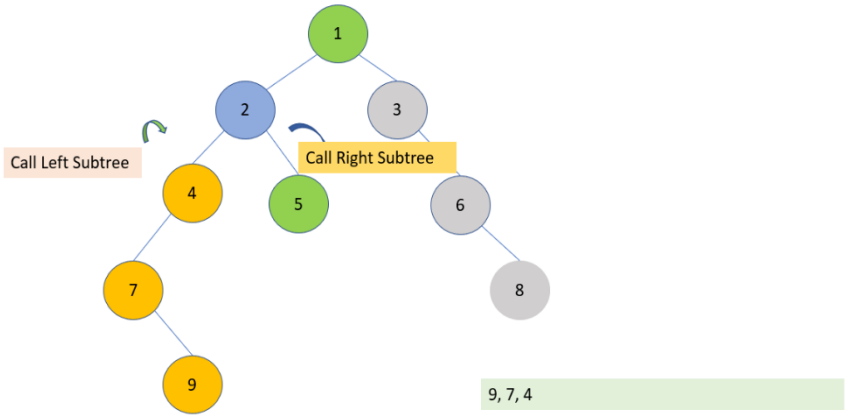

1
2
3
4
5
6
7
8
9
Call Left Subtree

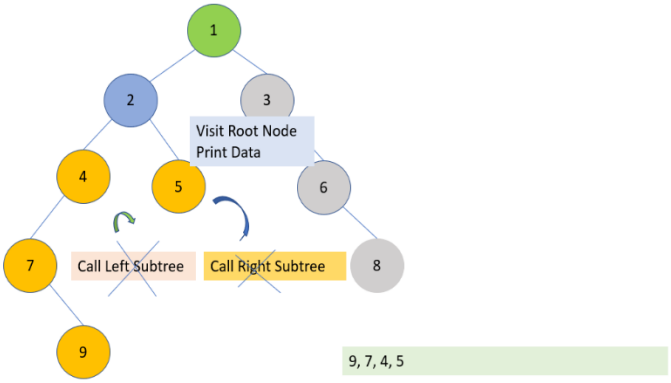

1
2
3
4
5
6
7
8
9
Call Right Subtree
Call Left Subtree

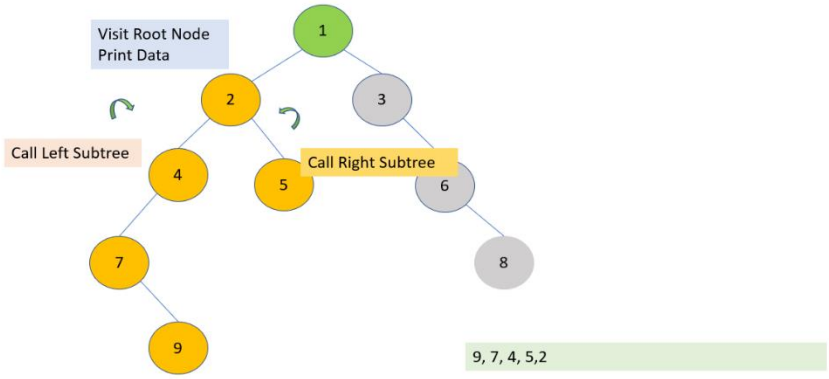

1
2
3
4
5
6
7
8
9
Visit Root Node
Print Data
Call Left Subtree
Call Right Subtree
9,

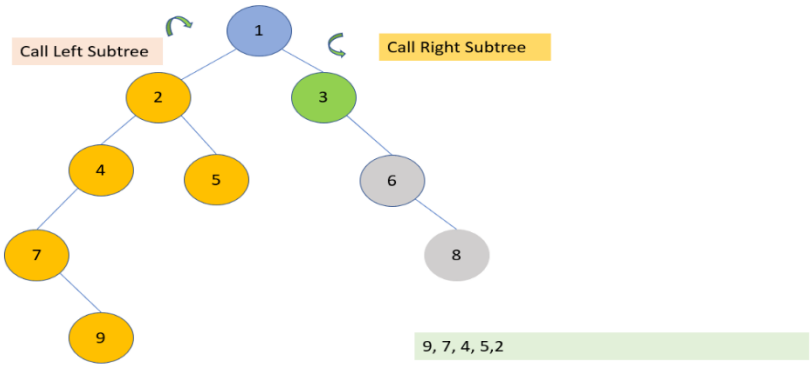

1
2
3
4
5
6
7
8
9
Visit Root Node
Print Data
Call Right Subtree
Call Left Subtree
9, 7,

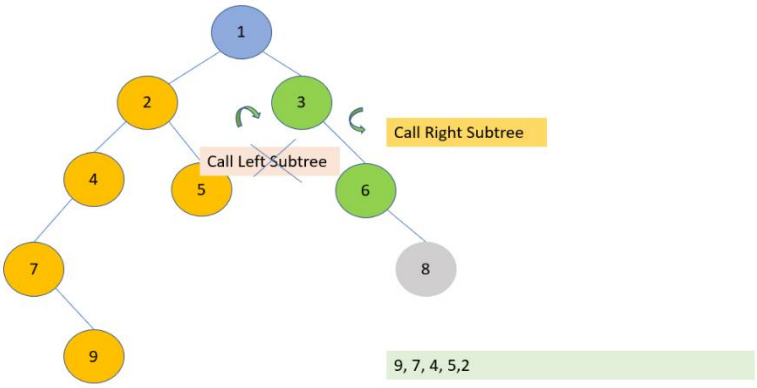

1
2
3
Visit Root Node
Print Data
4
5
6
Call Left Subtree
Call Right Subtree
7
8
9
9, 7, 4

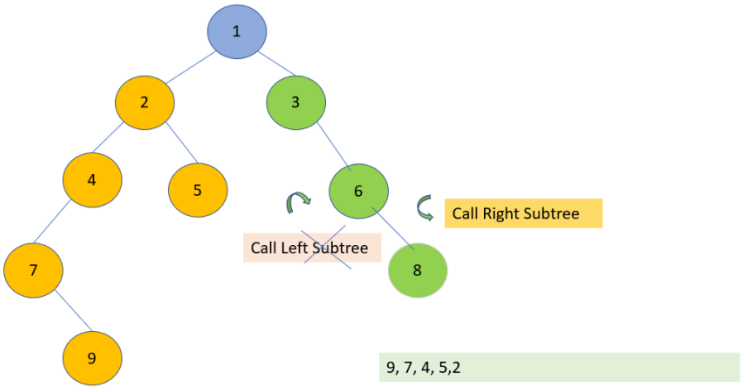

1
2
3
Call Left Subtree
Call Right Subtree
4
5
6
7
8
9
9, 7, 4

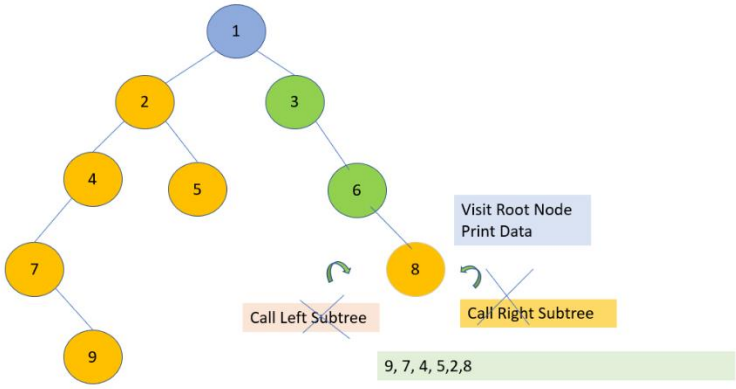

1
2
3
Visit Root Node
Print Data
4
5
6
7
Call Left Subtree
Call Right Subtree
8
9
9, 7, 4, 5

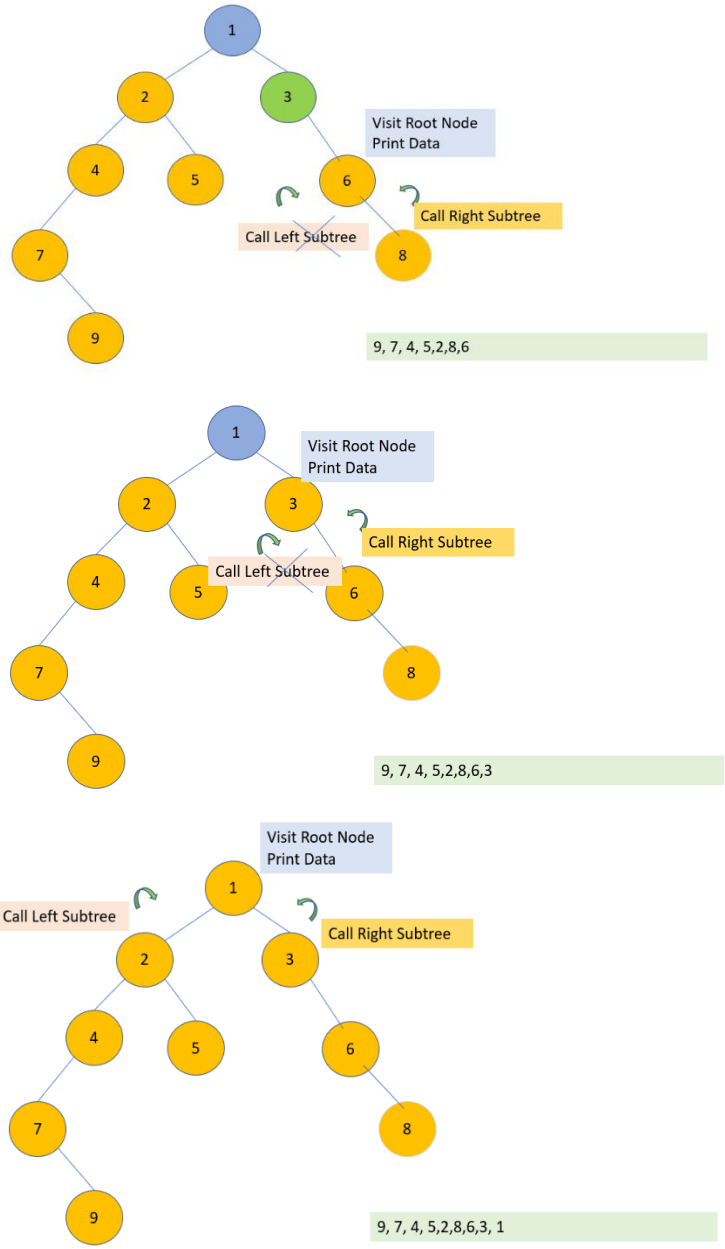

Visit Root Node
Print Data
1
2
3
Call Left Subtree
Call Right Subtree
4
5
6
7
8
9
9, 7, 4, 5,2


Call Left Subtree
Call Right Subtree
1
2
3
4
5
6
7
8
9
9, 7, 4, 5,2


1
2
3
Call Right Subtree
Call Left Subtree
4
5
6
7
8
9
9, 7, 4, 5,2


1
2
3
4
5
6
Call Right Subtree
Call Left Subtree
7
8
9
9, 7, 4, 5,2


1
2
3
4
5
6
Visit Root Node
Print Data
7
8
Call Left Subtree
Call Right Subtree
9
9, 7, 4, 5,2,8



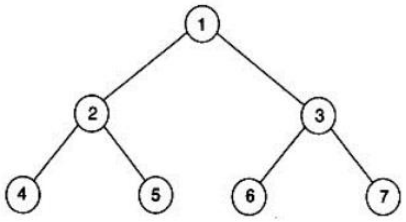
ALGORITHM Postorder_traversal(Root)

BEGIN:

```
IF root == NULL THEN
        RETURN 0;
ELSE
        Postorder_traversal(root→Left)
        Postorder_traversal(root→Right)
        WRITE (root→Data)
```

END;

2.2.4 Multiple Choice Questions

1	If the post-order traversal gives a b - c d * + then the label of the nodes 1, 2, 3 ... will be 
A	+, -, *, a, b, c, d
B	a, -, b, +, c, *, d
C	a, b, c, d, -, *, +
D	-, a, b, +, *, c, d
AN	**A**
DL	**M**

2	The following three are known to be the pre-order, inorder and post-order sequences of a binary tree. But it is not known which is which. MBCAFHPYK KAMCBYPFH MABCKYFPH Pick the true statement from the following.
A	I and II are pre-order and inorder sequences, respectively
B	I and III are pre-order and post-order sequences, respectively
C	II is the inorder sequence, but nothing more can be said about the other two sequence
D	II and III are the pre-order and inorder sequences, respectively
AN	**D**
DL	**M**

3	The preorder traversal of a binary tree is 80,90,61,72,43,53,22,37,1. The inorder traversal of the same tree is 80,61,90,43,72,22,53,1,37. The height of the tree will be?
A	4
B	5
C	6
D	3
AN	**B**

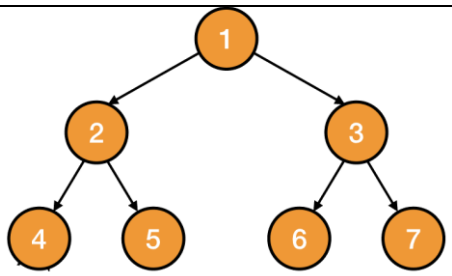
DL	**E**
TP	**4**

4	Draw the binary tree with node a,b,c,d,e,f and g for which the inorder and post-order traversals result in the following sequences: Inorder: a f b c d g e . Postorder: a f c g e d b. The pre-order of the above tree will be ?
A	b a f d c g e
B	b a f c d e g
C	b f a d c g e
D	b f a d c e g
AN	**d**

5	The inorder and pre-order traversal of a binary tree respectively are D B E A F C G and A B D E C F G, then the post-order traversal is?
A	D E B F G C A
B	E D B F G C A
C	D E F G B C A
D	E D B G F C A
AN	**A**
DL	**M**

6	The inorder and pre-order traversal of a binary tree respectively are D B F E G H A and A B D E F G H C, then the post-order traversal is?
A	D F H G E B C A
B	F H D G E B C A
C	D F G A B C H E
D	D F H G E B C A
AN	**D**
DL	**M**

7	Consider the following tree:

	 If the post-order traversal gives a, b, -, c, d, *, + then the label of the nodes 1, 2, 3 ,.. will be:
A	+, -, *, a, b, c, d
B	a, -, b, +, c, *, d
C	a, b, c, d, -, *, +
D	-, a, b, +, *, c, d
AN	**A**
DL	**M**

10.8 Tree Conversion

10.8.1 Building of tree using Inorder Tree Traversal and Preorder Tree Traversal

We can build a tree using Inorder and Preorder Traversal; the first node in preorder traversal will be root node. Using Inorder Traversal, we can determine the left subpart of the tree and the right subpart of the tree.

Preorder: 3, 9, 20, 15, 7

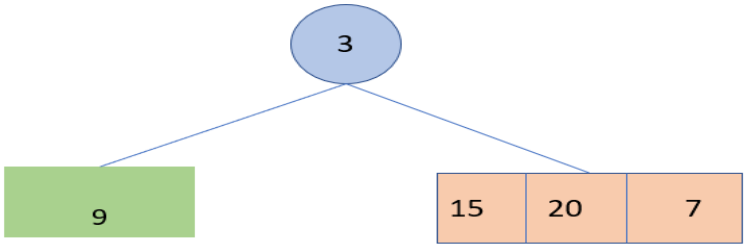
In order: 9, 3, 15, 20, 7

Step 1

Preorder: **3**, 9, 20, 15, 7 (3 will be the root of tree)

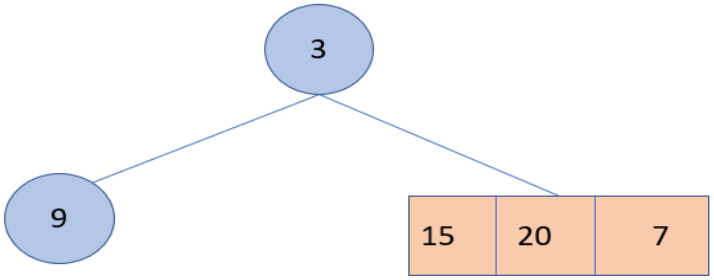
In order: 9, **3**, 15, 20, 7 (Left Root Right).

Here 3 is root of tree, so 9 will lie on LHS, and 15, 20, 7 will lie on RHS



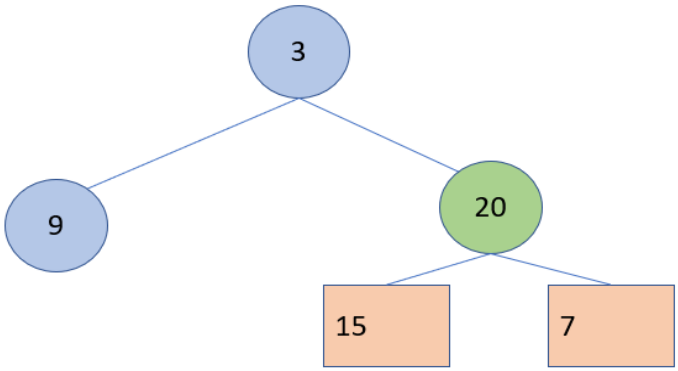
Step 2

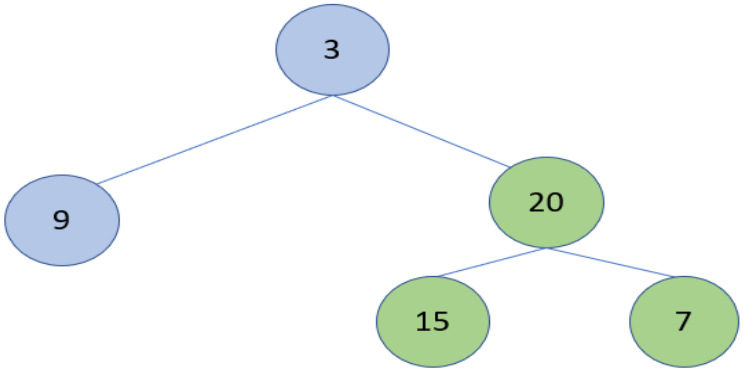
Preorder: 3, **9,** 20, 15, 7 (3 will be the root of tree)
In order: **9**, 3, 15, 20, 7(Left Root Right).
Here 9 is the root of subtree, so nothing on LHS and nothing will lie on RHS



Step 3

Preorder: 3, 9, 20, 15, 7 (20 will be the root of tree)
In order: **9**, 3, 15, **20**, 7(Left Root Right).
Here 20 is the root of subtree, so 15 on LHS and 20 will lie on RHS.





Complexity: - O (n^2)

Building of tree using Inorder Tree Traversal and Preorder Tree Traversal

ALGORITHM Build_Tree(Preorder [], Inorder [], n)

BEGIN:

```
PreStart = 0
PreEnd = n-1    // n is number of elements
InorderStart = 0
InorderEnd = n-1 // n is number of elements
RETURNConstruct (Preorder, Inorder, Prestart, PreEnd, InorderStart, InorderEnd)
```

END;

ALGORITHM Construct(Preorder [], Inorder[], Prestart, PreEnd,InorderStart, InorderEnd)

BEGIN:

```
IF Prestart >PreEnd ||  InorderStart>InorderEnd THEN
        RETURN NULL

Val = Preorder [Prestart]
P=GetNode(Val)
FOR i=0 TO Inorder.length() DO
        IF Val == inorder[i] THEN
                k=i
                BREAK

P→left = Construct (Preorder, Inorder, Prestart + 1, PreStart + (k – InorderStart),
InorderStart, k-1)

P→right = construct (Preorder, Inorder, Prestart + (k – InorderStart)+ 1, PreEnd , k+1,
InorderEnd)
RETURN P
```

END;

10.8.2 Building of tree using Inorder Tree Traversal and Postorder Tree Traversal

We can build a tree using Inorder and Post-order Traversal; the last node in Post-order traversal will be the root node. Using Inorder Traversal, we can determine the left subpart of the tree and the right sub part of the tree.

Postorder: 9, 1, 2, 12, 7, 5, 3, 11, 4, 8.

In order: 9, 5, 1, 7, 2, 12, 8, 4,3, 11.

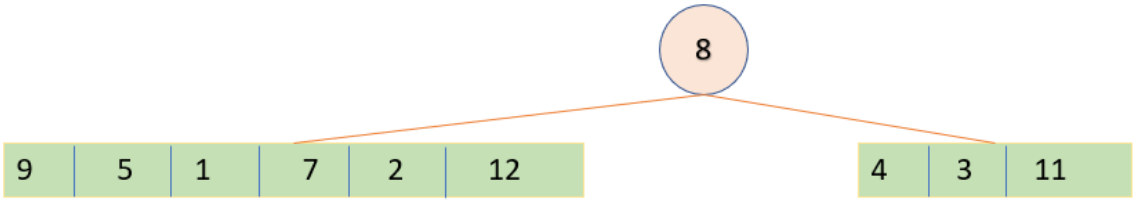
Step 1

Postorder: 9, 1, 2, 12, 7, 5, 3, 11, 4, 8. (Last element in Postorder traversal will be 8.)

In order: 9, 5, 1, 7, 2, 12, 8, 4,3, 11.

Left Subtree will have 9,5,1,7,2,12 and Right Subtree will have 4, 3, 11.

Root will be 8



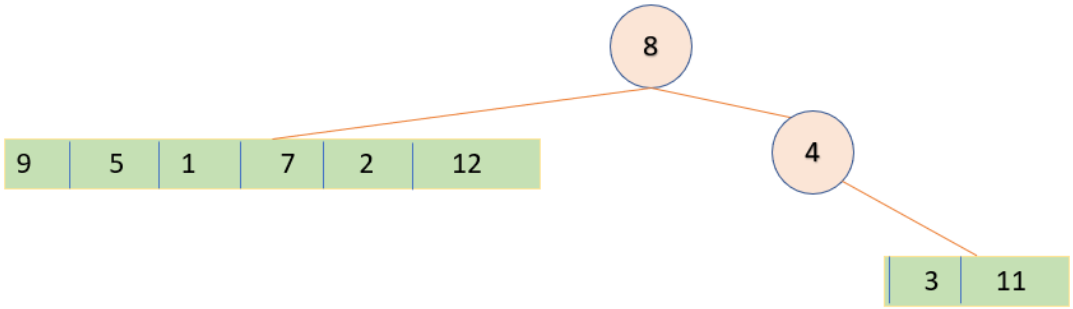
Step 2

Postorder: 9, 1, 2, 12, 7, 5, 3, 11, 4, 8. (Last element in Postorder traversal will be 4.)

In order: 9, 5, 1, 7, 2, 12, 8, 4,3, 11.

Left Subtree will have nothing and Right Subtree will have 3, 11.

Root will be 4



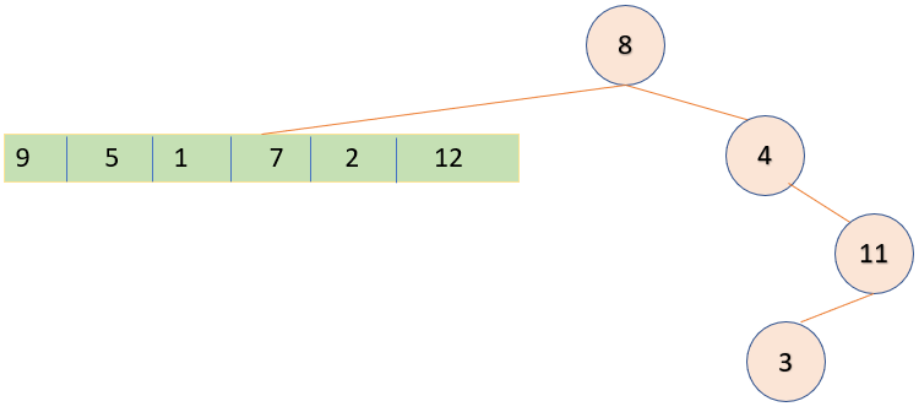
Step 3

Postorder: 9, 1, 2, 12, 7, 5, 3, 11, 4, 8. (Last element in Postorder traversal will be 11.)

In order: 9, 5, 1, 7, 2, 12, 8, 4,3, 11.

Left Subtree will have 3, and Right Subtree will have nothing.

Root will be 11



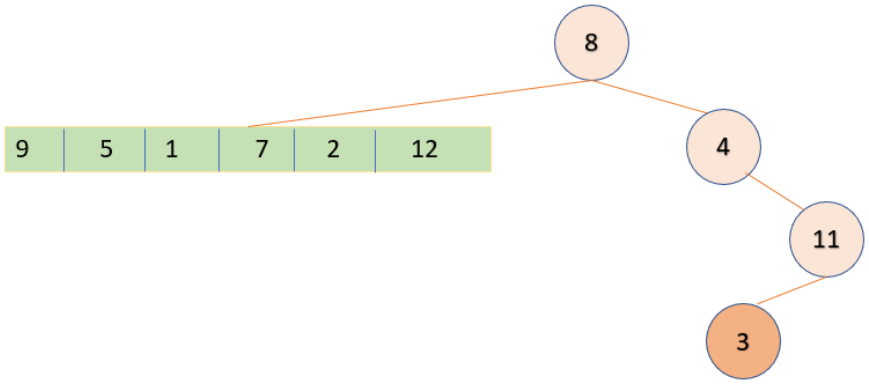
Step 4

Postorder: 9, 1, 2, 12, 7, 5, 3, 11, 4, 8. (Last element in Postorder traversal will be 11.)

In order: 9, 5, 1, 7, 2, 12, 8, 4,3, 11.

Left Subtree will have nothing, and Right Subtree will have nothing.

Root will be 3.



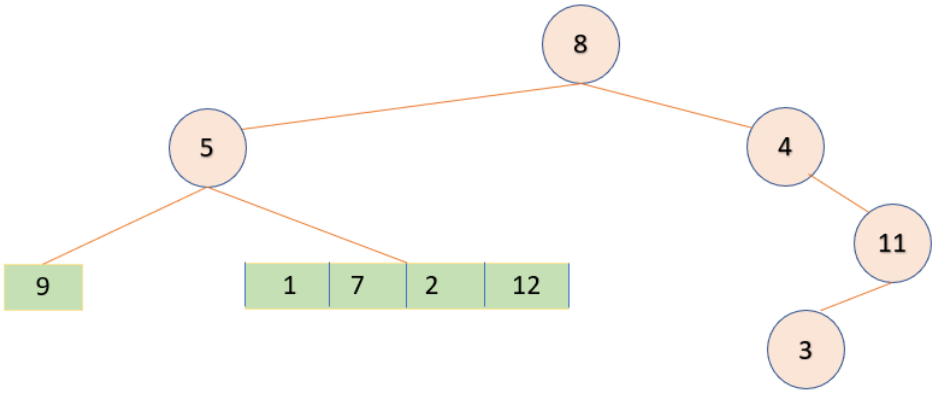
Step 5

Postorder: 9, 1, 2, 12, 7, 5,3, 11, 4, 8. (Last element in Postorder traversal will be 5.)

In order: 9, 5, 1, 7, 2, 12, 8, 4,3, 11.

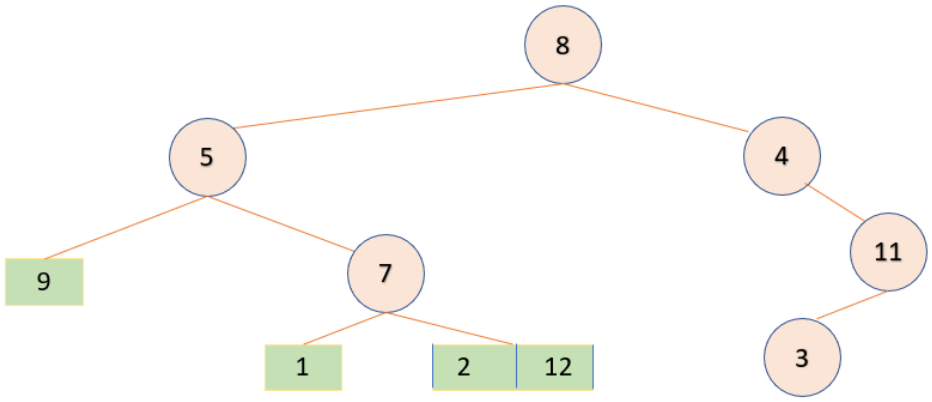
Left Subtree will have 9, and Right Subtree will have 1,7,2,12.

Root will be 5.



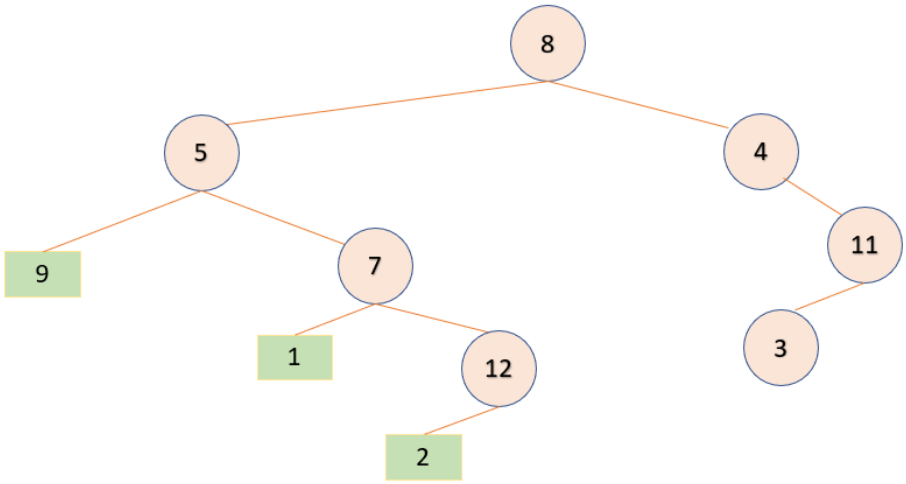
Step 5

Postorder: 9, 1, 2, 12, 7, 5,3, 11, 4, 8. (Last element in Postorder traversal will be 7.)
In order: 9, 5, 1, 7, 2, 12, 8, 4,3, 11.
Left Subtree will have 1, and Right Subtree will have 2,12.
Root will be 7.



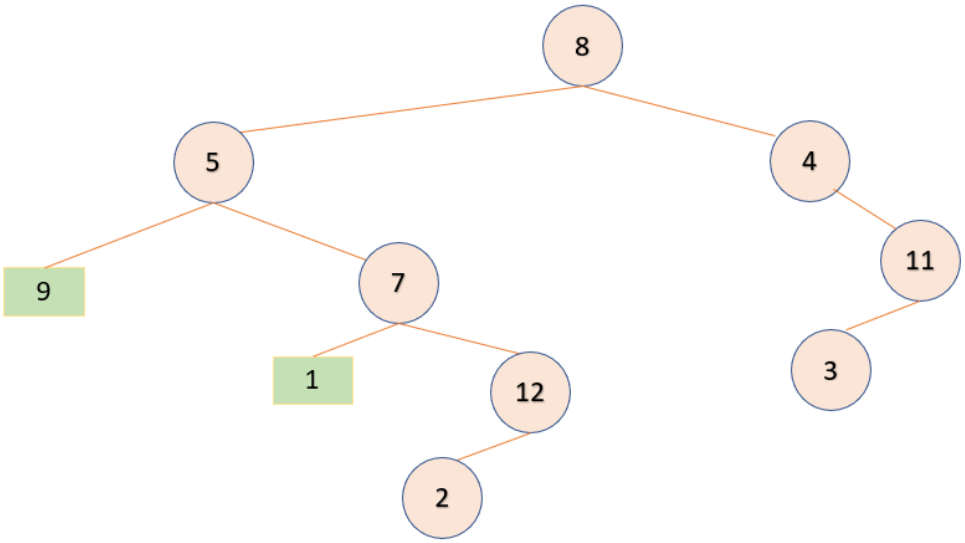
Step 6

Postorder: 9, 1, 2, 12, 7, 5,3, 11, 4, 8. (Last element in Postorder traversal will be 12.)
In order: 9, 5, 1, 7, 2, 12, 8, 4,3, 11.
Left Subtree will have 2 and Right Subtree will have nothing.
Root will be 12.



Step 7: -

Postorder: 9, 1, 2, 12, 7, 5,3, 11, 4, 8. (Last element in Postorder traversal will be 2.)
In order: 9, 5, 1, 7, 2, 12, 8, 4,3, 11.
Left Subtree will have nothing and Right Subtree will have nothing.
Root will be 2.



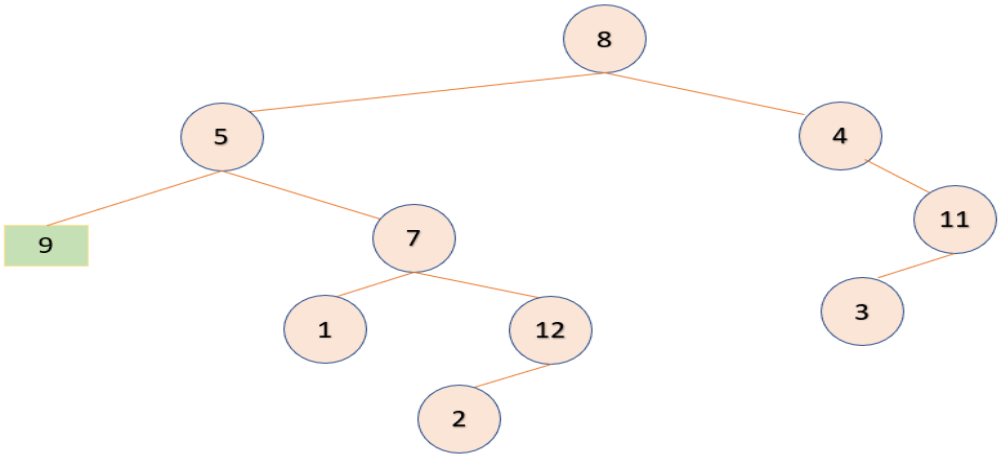
Step 8

Postorder: 9, 1, 2, 12, 7, 5,3, 11, 4, 8. (Last element in Postorder traversal will be 1.)

In order: 9, 5, 1, 7, 2, 12, 8, 4,3, 11.

Left Subtree will have nothing, and Right Subtree will have nothing.

Root will be 1.



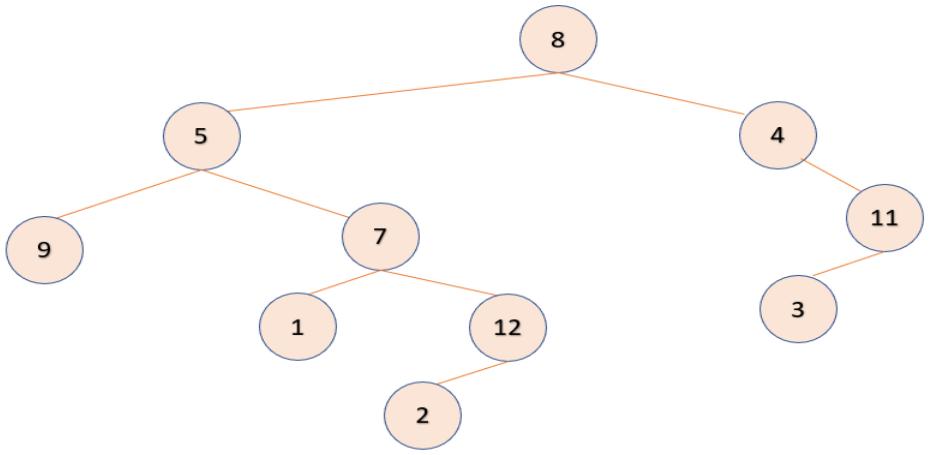
Step 9

Postorder:9, 1, 2, 12, 7, 5,3, 11, 4, 8. (Last element in Postorder traversal will be 9.)

In order: 9, 5, 1, 7, 2, 12, 8, 4,3, 11.

Left Subtree will have nothing, and Right Subtree will have nothing.

Root will be 9.



Algorithm for building Tree from Postorder and Inorder Traversal

ALGORITHM Build_Tree(Postorder [], Inorder [], n)

BEGIN:

PostStart = 0

PostEnd = n-1 // n is number of elements

InorderStart = 0

InorderEnd = n-1 // n is number of elements

RETURNConstruct (Inorder, Postorder, InorderStart, InorderEnd, PostStart, PostEnd)

END;

ALGORITHM Construct (Inorder[], Postorder[], InorderStart, InorderEnd, PostStart, PostEnd)

BEGIN:

IF Poststart>PostEnd || InorderStart>InorderEnd THEN

RETURN NULL

Val = Postorder [Postend];

P=GetNode()

FOR i=0 TO Inorder.length()DO

IF Val == inorder[i] THEN

k=i

BREAK

P→left = Construct (Inorder,Postorder, InorderStart, k-1, PostStart, PostStart + k - (Inorderstart+1))

P→right = construct (Inorder, Postorder, k+1, InorderEnd, Poststart +k –InorderStart, PostEnd -1)

RETURN P

END;

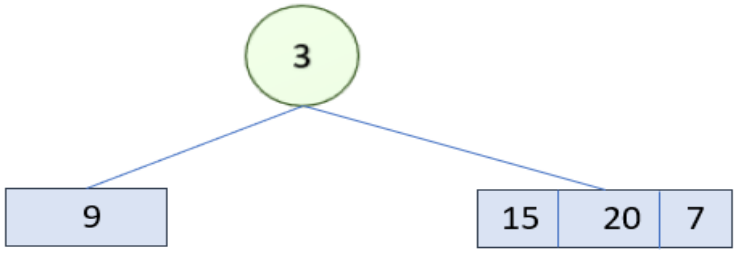
Complexity

Time Complexity: - O (n^2)
Space Complexity :- O(n)

Example: -
Inorder :- 9, 3, 15, 20, 7
Postorder :- 9, 15, 7, 20, 3

Step 1

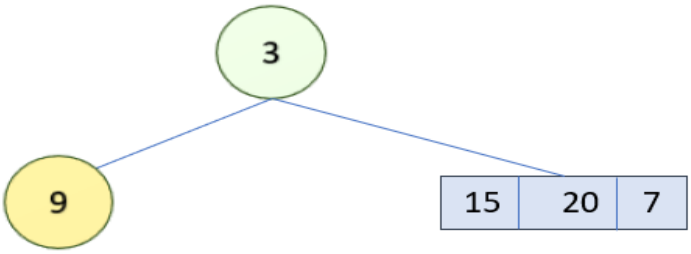
Inorder :- 9, 3, 15, 20, 7
Postorder :- 9, 15, 7, 20, 3 (3 will be root).
3 will be root of tree, 9 will be left subtree and 15, 20, 7 will be on Right subpart.



Step 2

Left Subpart consists of node 9; on searching node 9 in postorder traversal, the node has been found at index one.

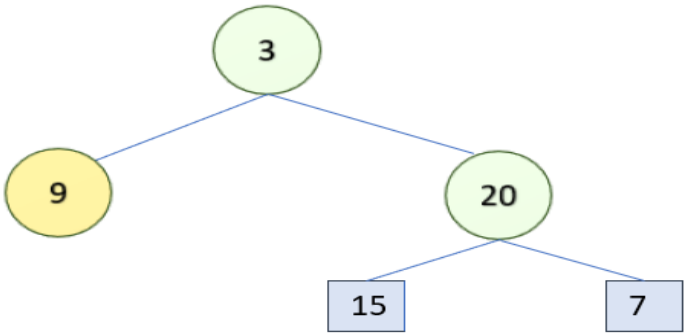
Inorder :- 9, 3, 15, 20, 7
Postorder :- 9, 15, 7, 20, 3 (9 will be root).
9 will be root of subtree, no node will be left subtree and np node will be on Right subpart.



Step 3

Left Subpart does not contain any node; on searching 15, 20, 7 for right subpart, node 20 was found as a root on postorder traversal.
Inorder :- 9, 3, 15, 20, 7
Postorder :- 9, 15, 7, 20, 3 (20 will be root).

20 will be the root of the subtree, 15 will be the left subtree, and 7 will be on the Right subpart.



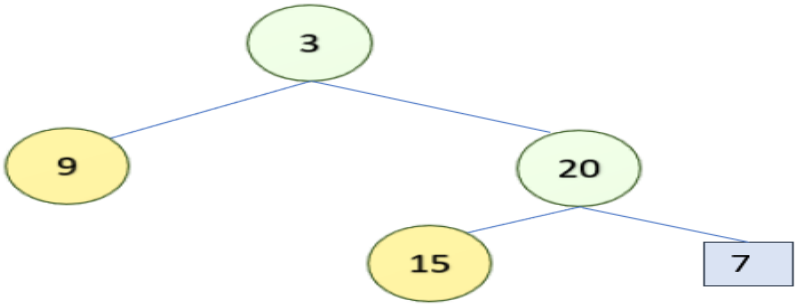
Step 4

Left Subpart contain node 15. 15 is marked with red color in postorder traversal

Inorder :- 9, 3, 15, 20, 7

Postorder :- 9, 15, 7, 20, 3 (20 will be root).

15 will be root of sub tree, no node will be left subtree and no node will be on Right subpart.



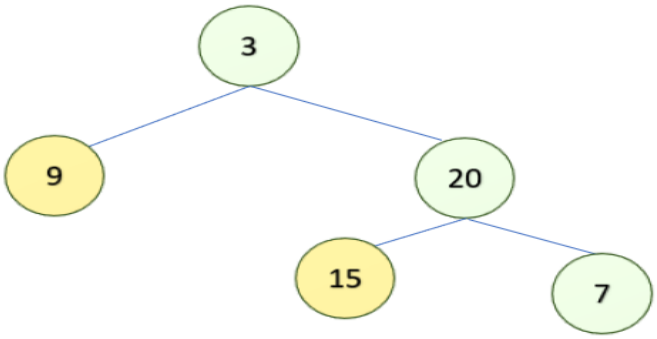
Step 5

The left Subpart does not contain any node. The right subpart of node 20 contains node 7.

Inorder :- 9, 3, 15, 20, 7

Postorder :- 9, 15, 7, 20, 3 (20 will be root).

15 will be root of subtree, no node will be left subtree, and no node will be on the Right subpart.

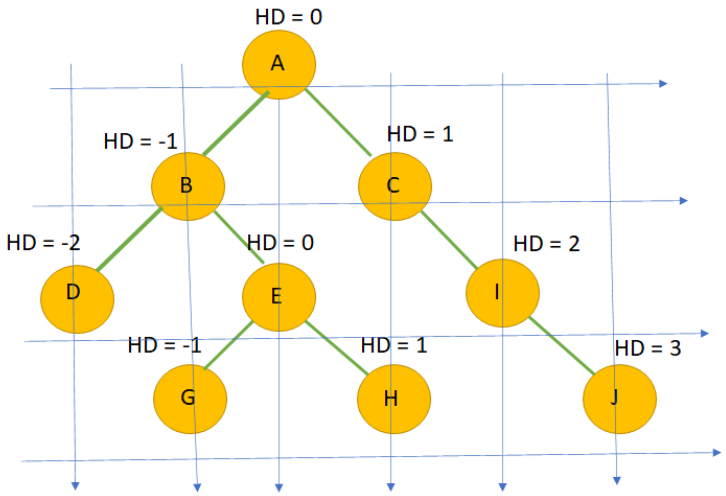


10.9 Top View Traversal

The focus is to find the top-view of a tree using recursion. The top-view of a tree is actually that view of a tree in which when we view the tree from the top, and only those nodes which are visible from the top will be the part of top-view traversal.
One way to find top-view is by looking for horizontal distance, just like a number line pattern. So, horizontal distance to each node is given from the root node of its level. Every node to the left of its root will be subtracted by 1 from its root distance and right will be added by 1.
So we can say that,

1. If for root node Horizontal Distance(HD) = 0
2. Then for left child HD = HD-1(from its parent node)
3. Then for right child HD= HD+1(from its parent node)

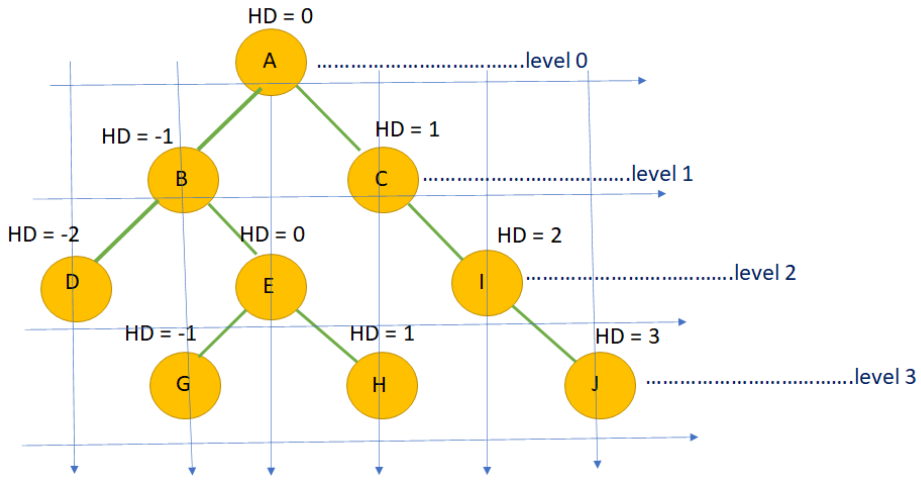
Let us demonstrate it with an example:



From the above example we can see horizontal distance of:

A, E --> 0
B, G -->-1
C, H --> 1
D -->-2
I --> 2
J --> 3

In the above example, we can see that few nodes have the same Horizontal Distance, but as we see from the top, we find that the nodes which have lower level will be visible and all inner nodes are overshadowed.



In the above figure, top-view will consist of nodes : D B A C I J

Create a Map-Table that stores pair of nodes and level corresponding to distance from the root.

10.9.1 TopView Recursive approach

ALGORITHM TopView(Root, DIST, LEVEL)

//Root is node at each level, DIST means distance and LEVELdenotes level of tree

BEGIN:

```
IF Root== NULL THEN
    RETURN
IF M[DIST]== Φ  or M[DIST] > LEVEL THEN
        M[DIST]= pairof(Root, LEVEL)
TopView(Root-->Left, DIST-1, LEVEL+1)
TopView(Root-->Right, DIST+1, level+1)
WRITE(values in sorted order of DIST index of map-Table M)
```

END;

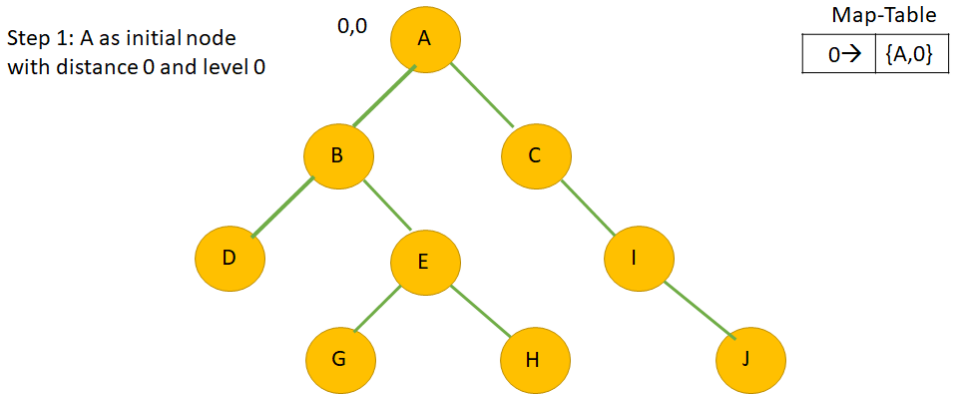
Time Complexity: O(nlogn) as there is n number of nodes and logn is the insertion time it takes to insert in a map-table.

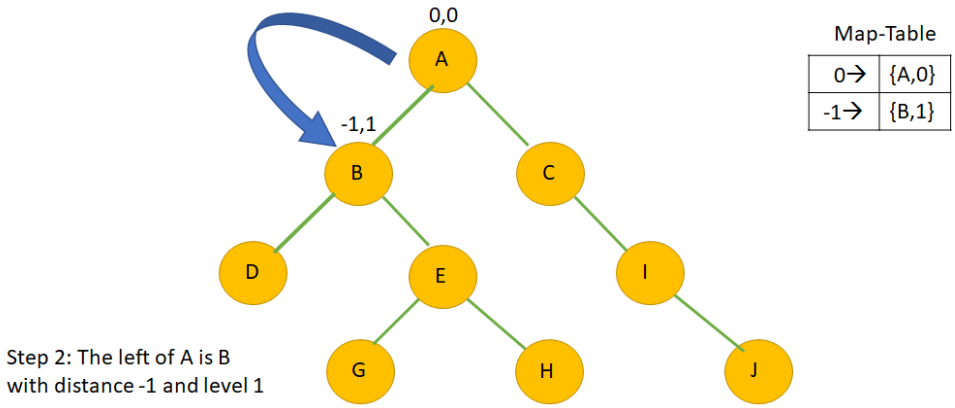
10.9.2 Step-wise explanation

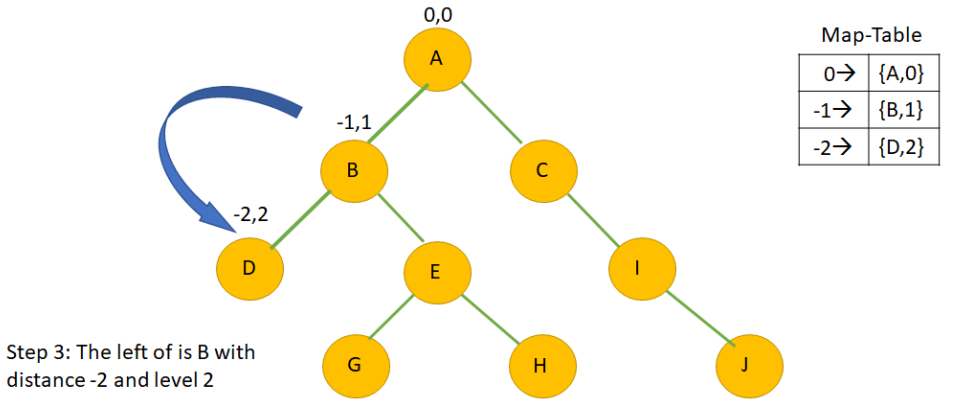
1. The root node, distance and level is passed as parameter. Initially, for above example A, 0, 0.
2. Check if the node passed is empty or has the address.
3. Check whether the distance passed already present in Map-Table.
4. If not, maintain its entry in Map-Table. For example 0 --> {A,0}
5. Recursively call the TopView function for the left of the current node.

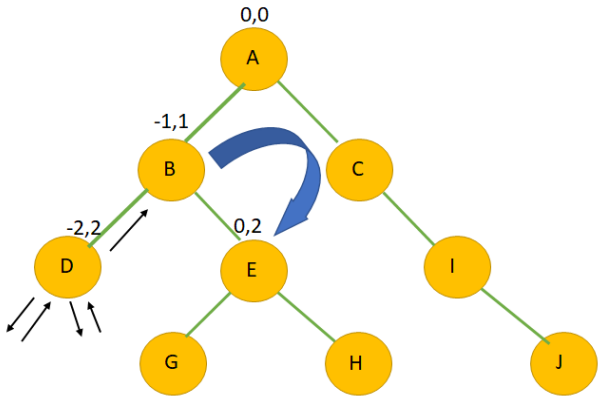
6. If the left node is empty recursively, call the TopView function for the right of the current node.

The below step-wise figures give a better view of TopView binary tree traversal.





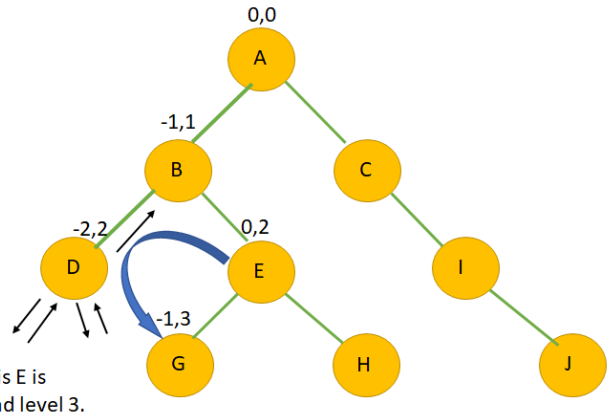




Map-Table

0→	{A,0}
-1→	{B,1}
-2→	{D,2}

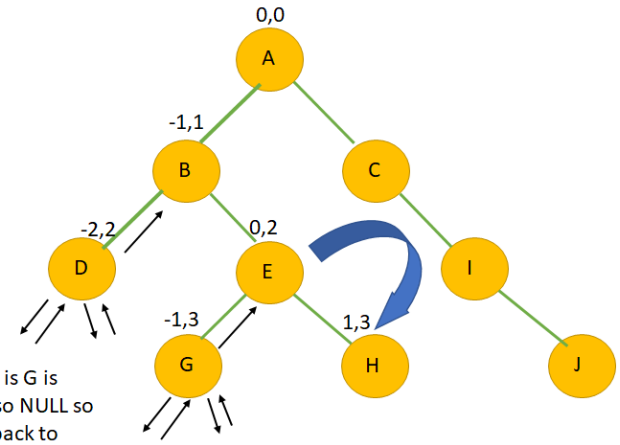
Step 4: The left of is D is NULL and write also NULL so the control goes back to right of B with distance 0 and level 2. As condition is false move to next recursive call.



Map-Table

0→	{A,0}
-1→	{B,1}
-2→	{D,2}

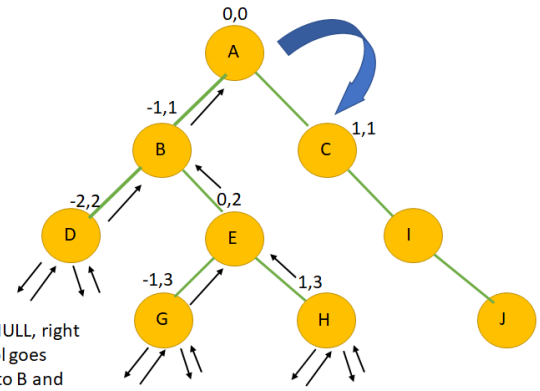
Step 5: The left of is E is with distance -1 and level 3. again the condition false as map already contains the distance and value of level in map is less than present value of G. The control goes to left of G



Map-Table

0→	{A,0}
-1→	{B,1}
-2→	{D,2}
1→	{H,3}

Step 6: The left of is G is NULL and right also NULL so return. Call goes back to right of E with distance 1 and level 3

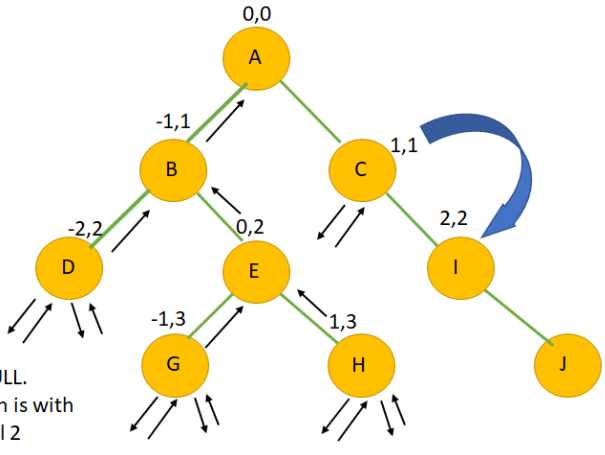


Map-Table

0→	{A,0}
-1→	{B,1}
-2→	{D,2}
1→	{C,1}

As value of H level is greater than C level so Map will be replaced by new value for distance 1

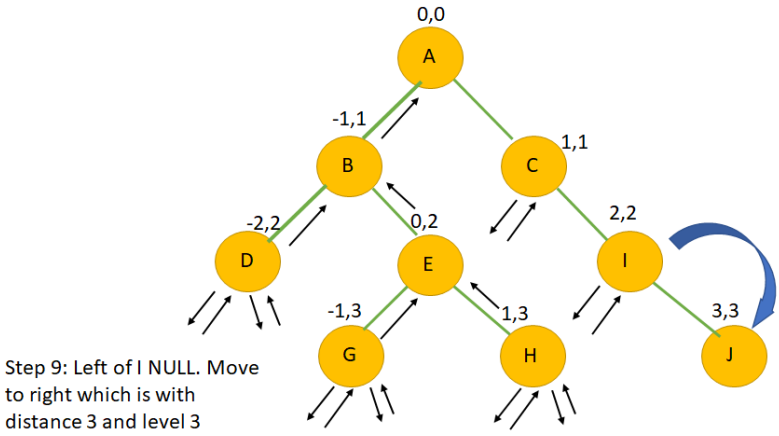
Step 7: Left of H NULL, right of H NULL. Control goes back to E, from E to B and from B to A. The Left sub-tree of root node A traversed now move towards right. Right of A is with distance 1 and Level 1



Map-Table

0→	{A,0}
-1→	{B,1}
-2→	{D,2}
1→	{C,1}
2→	{I,2}

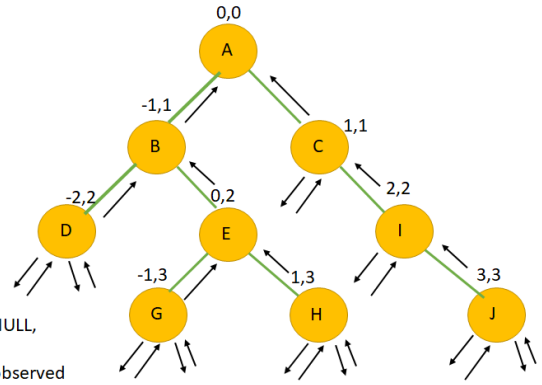
Step 8: Left of C NULL. Move to right which is with distance 2 and level 2



Map-Table

0→	{A,0}
-1→	{B,1}
-2→	{D,2}
1→	{C,1}
2→	{I,2}
3→	{J,3}

Step 9: Left of I NULL. Move to right which is with distance 3 and level 3



Map-Table

0→	{A,0}
-1→	{B,1}
-2→	{D,2}
1→	{C,1}
2→	{I,2}
3→	{J,3}

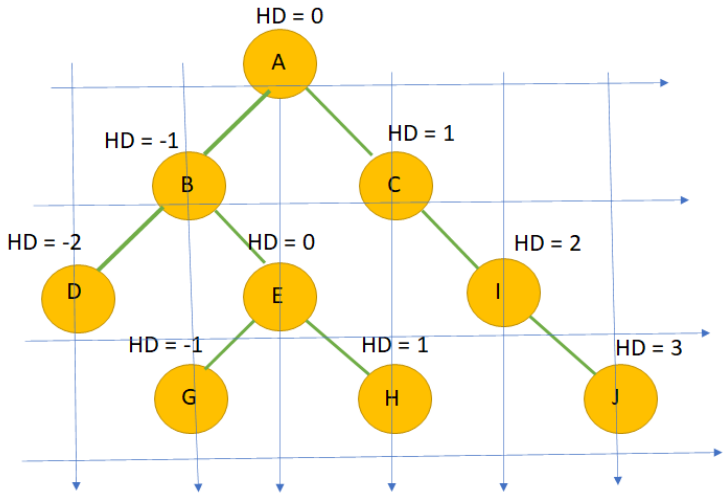
Step 10: Left of J NULL, Right of J NULL. Backtracking it is observed all nodes visited and recursion terminates.

Map prints values in sorted order as per distance values. Thus TopView prints: D B A C I J

10.10 Bottom View

Unlike top view traversal in bottom view traversal, those nodes will be picked which are at a higher level since the view of the tree is from the bottom. The process remains the same by first finding the Horizontal Distance and then by checking the level of the node.

The demonstration of the same can be done by the previous example:



From the above example we can find horizontal distance of each node:

A, E--> 0

B, G -->-1

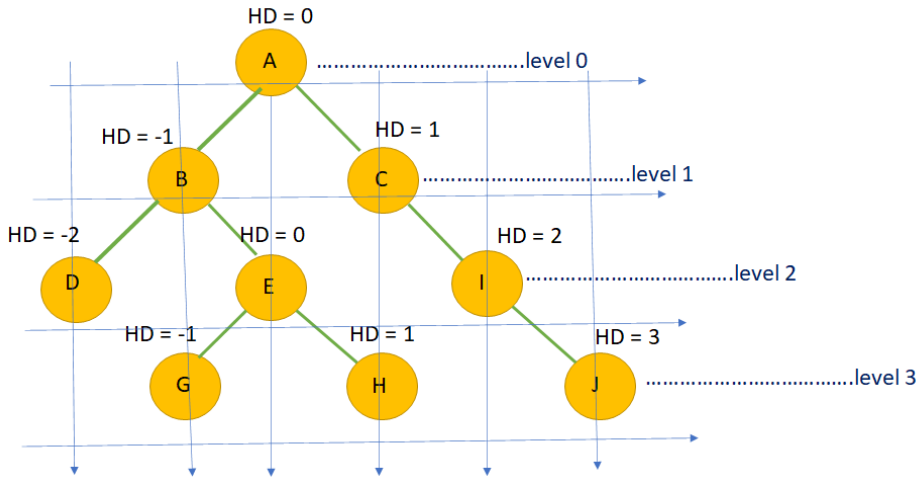
C, H--> 1

D -->-2

I --> 2

J --> 3

In the above example, we can see that few nodes have the same Horizontal Distance, but as we see from the bottom, we find that the nodes which have a higher level will be visible, and all lower level nodes are overshadowed.



In the above figure, the bottom-view will consist of nodes: D GEH I J

Create a Map-Table that stores pair of nodes and level corresponding to distance from root.

10.10.1 BottomView Recursive approach

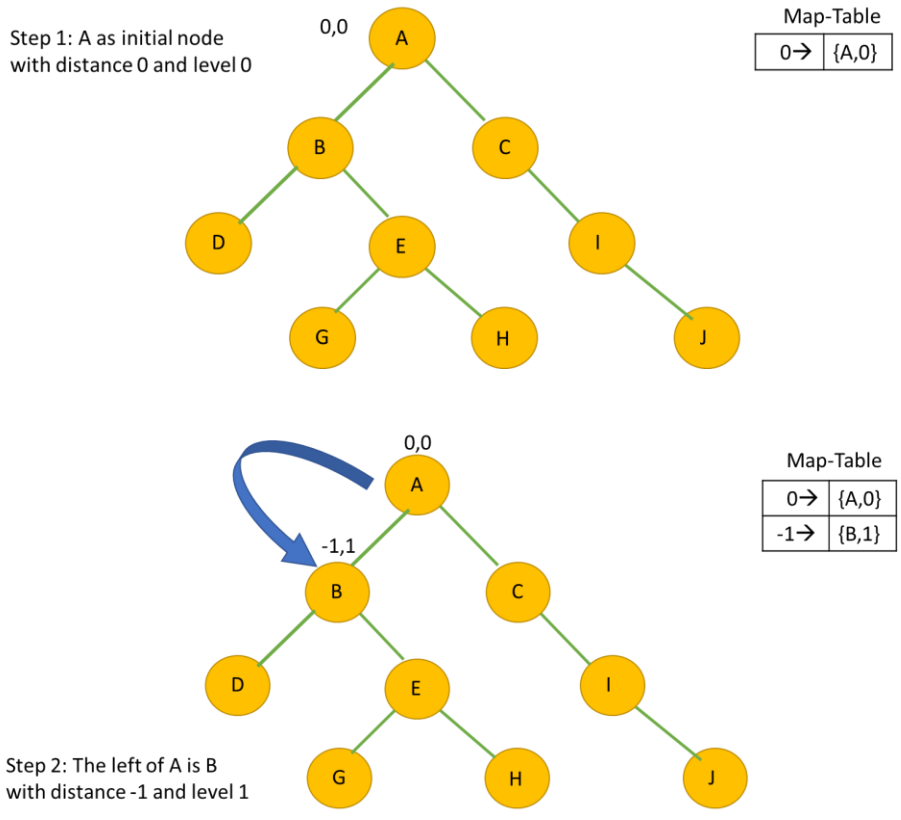
ALGORITHM BottomView(Root, DIST, LEVEL)
//Root is node at each level, DIST means distance and LEVEL denotes level of tree
BEGIN:

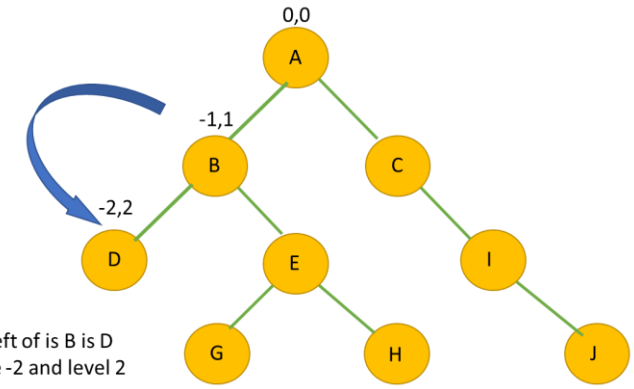
```
IF Root== NULL THEN
    RETURN
IF M[DIST]== Φ  or M[DIST] < LEVEL THEN
        M[DIST]= pairof(Root, LEVEL)
TopView(Root-->Left, DIST-1, LEVEL+1)
TopView(Root-->Right, DIST+1, level+1)
WRITE( values in sorted order of DIST index of map-Table M)
```

END;

Time Complexity: O(nlogn) as there is n number of nodes and logn is the insertion time it takes to insert in a map-table.

10.10.2 Step-wise explanation via diagrammatic approach

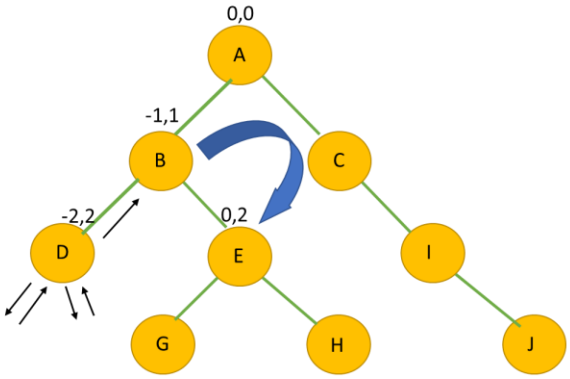




Map-Table

0→	{A,0}
-1→	{B,1}
-2→	{D,2}

Step 3: The left of is B is D with distance -2 and level 2

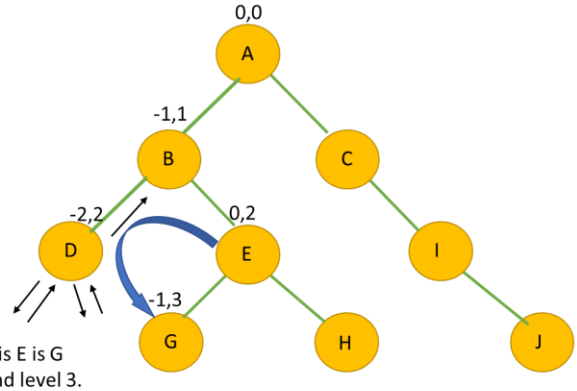


Map-Table

-1→	{B,1}
-2→	{D,2}
0→	{E,2}

Step 4: The left of is D is NULL and write also NULL so the control goes back to right of B i.e., E with distance 0 and level 2.

As the value of A level is less than value of E level so Map-Table will be replaced by new value for distance 0

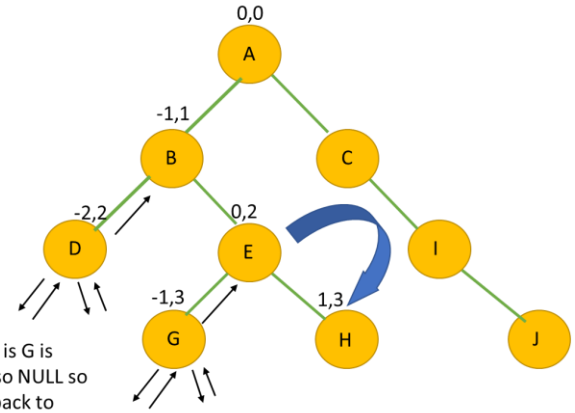


Map-Table

-2→	{D,2}
0→	{E,2}
-1→	{G,3}

Step 5: The left of is E is G with distance -1 and level 3.

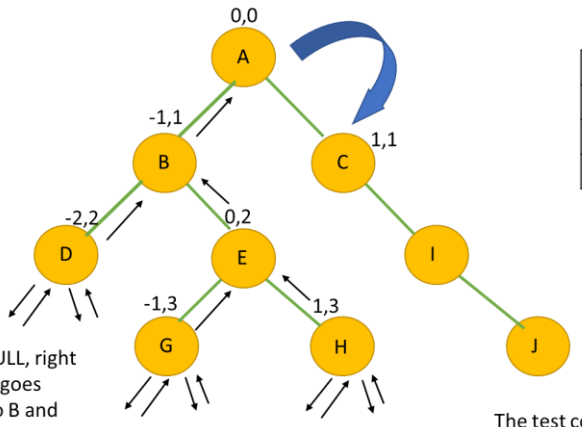
As the value of B level is less than G level so Map-Table will be replaced by new value for distance -1



Map-Table

-2→	{D,2}
0→	{E,2}
-1→	{G,3}
1→	{H,3}

Step 6: The left of is G is NULL and right also NULL so return. Call goes back to right of E i.e., H with distance 1 and level 3

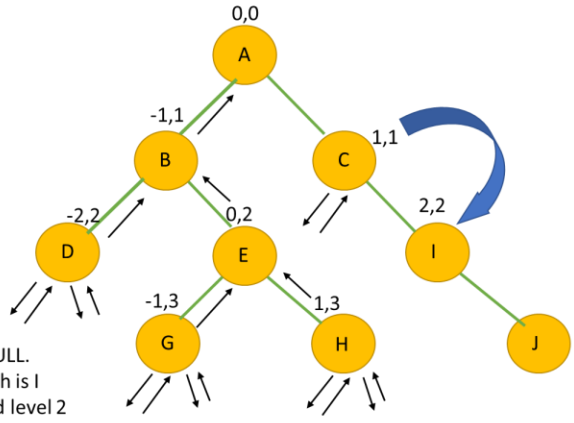


Map-Table

-2→	{D,2}
0→	{E,2}
-1→	{G,3}
1→	{H,3}

Step 7: Left of H NULL, right of H NULL. Control goes back to E, from E to B and from B to A. The Left sub-tree of root node A traversed now move towards right. Right of A is C with distance 1 and Level 1

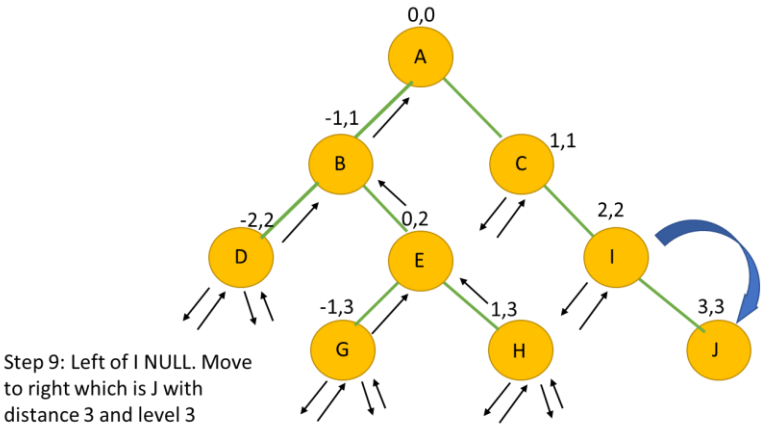
The test condition of if statement is false as 3 < 1 is false. So no change in map-table



Map-Table

-2→	{D,2}
0→	{E,2}
-1→	{G,3}
1→	{H,3}
2→	{I, 2}

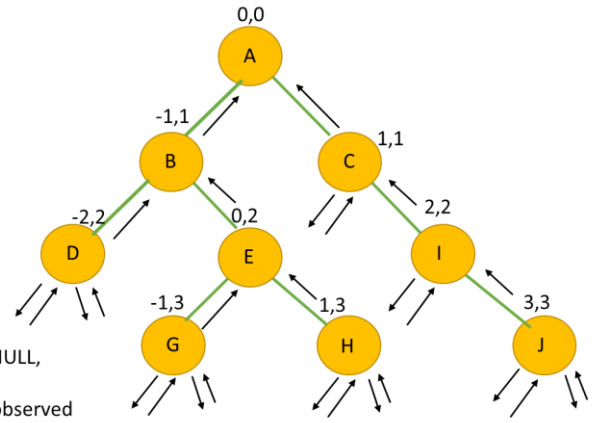
Step 8: Left of C NULL. Move to right which is I with distance 2 and level 2



Map-Table

-2→	{D,2}
0→	{E,2}
-1→	{G,3}
1→	{H,3}
2→	{I, 2}
3→	{J,3}

Step 9: Left of I NULL. Move to right which is J with distance 3 and level 3



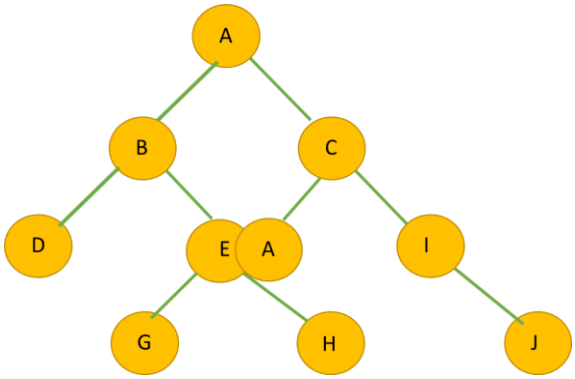
Map-Table

-2→	{D,2}
0→	{E,2}
-1→	{G,3}
1→	{H,3}
2→	{I, 2}
3→	{J,3}

Step 10: Left of J NULL, Right of J NULL. Backtracking it is observed all nodes visited and recursion terminates.

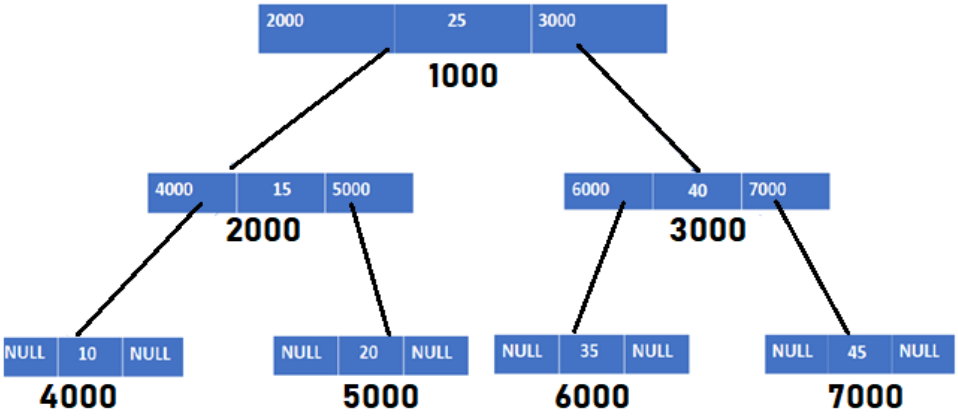
Map-Table prints values in sorted manner as per distance values thus BottomView is D G E H I J

Note: The nodes at the same level and having the same horizontal distances, then in traversal, anyone of the two will be written. As, if properly seen, both the nodes will overlap each other. In the above figure, the left child of node C and right child of node B, i.e., E, will be at the same level and distance; hence either of them will be considered in the traversal.



10.11 Threaded Binary Tree

Consider the Binary Tree given below



The total number of leaf nodes is always one greater than the number of internal nodes. Leaf nodes always contain two null pointers because there are and left or right child. If the node contains one child then also one part (left or right pointer) is NULL. It is clear that from the above diagram, that number of NULL pointers (memory locations) is always one greater than the number of assigned pointers (total nodes).

If the right and left memory fields of the nodes are NULL and we replace them by inorder successor and predecessor, the resulting tree will be known as a threaded binary tree. It is of two types.

10.11.1 Single-Threaded Binary Tree

If we replace only one of the left and right null addresses by the successor (for right) or predecessor (for left), this becomes single threaded binary tree. These are of two types

a) Right Threaded Binary Tree
b) Left Threaded Binary Tree

10.11.2 Double Threaded Binary Tree

If successor and predecessor, respectively, replace both the right and left NULL addresses, it becomes the double threaded binary tree.

NOTE: Now left or right field of the node either contains the address of the left or right child or can store a thread (memory location) that is the address of the in-order predecessor or successor.

The nodesin the threaded binary tree contain the following fields

- Data field

- Left child or Left Thread
- Right child or Right thread

One Boolean variable or character (Indicating that whether a particular node is threaded or not)

Boolean Variable is_right_Thread or Is_left_Thread	left_ child or left Thread or NULL	Data	right_child or right thread or NULL

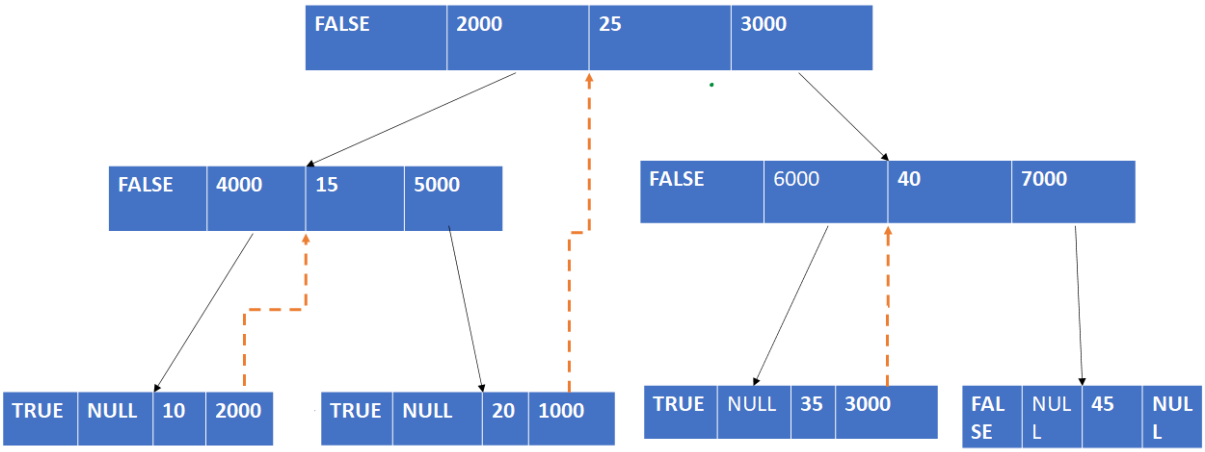
10.11.3 Right Threaded Binary Tree

If for a node right child field is NULL then replace it by an in-order successor

Example: Consider

For tree on the last page,

- Inorder traversal is 10 15 20 25 35 40 45
- The right memory location of node 10 replaced by the address of 15
- The right memory location of node 20 replaced by the address of 25
- The right memory location of node 35 replaced by the address of 40
- The right memory location of node 45 not replaced by any address because there is no in-order successor; hence, this memory location remains unused.
-

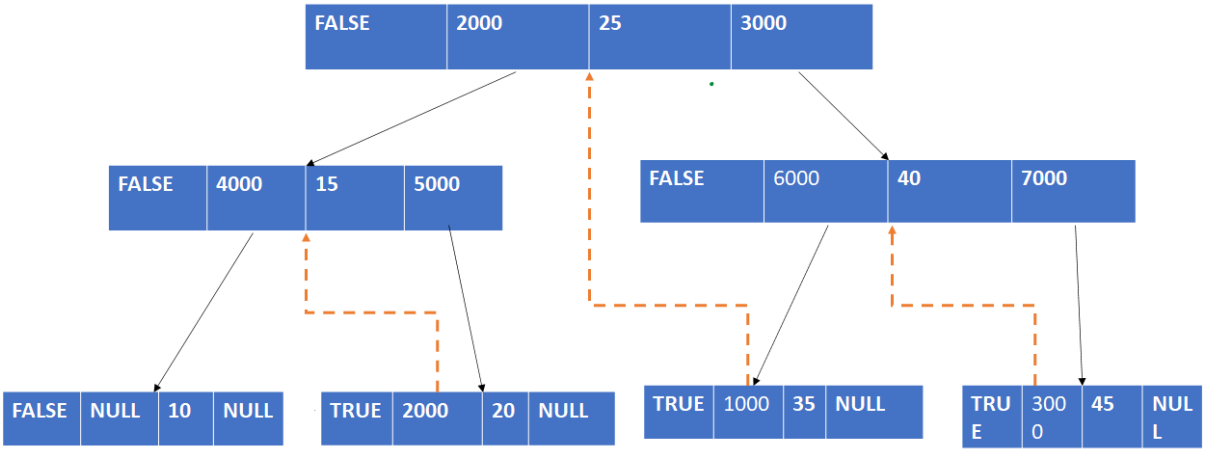


10.11.4 Left Threaded Binary Tree

If for a node leftchild field is NULL then replace it by inorder predecessor

- An in-order traversal of a given tree is 10 15 20 25 35 40 45
- The left memory location of node 10 is not replaced by any address because there is no in-order predecessor.

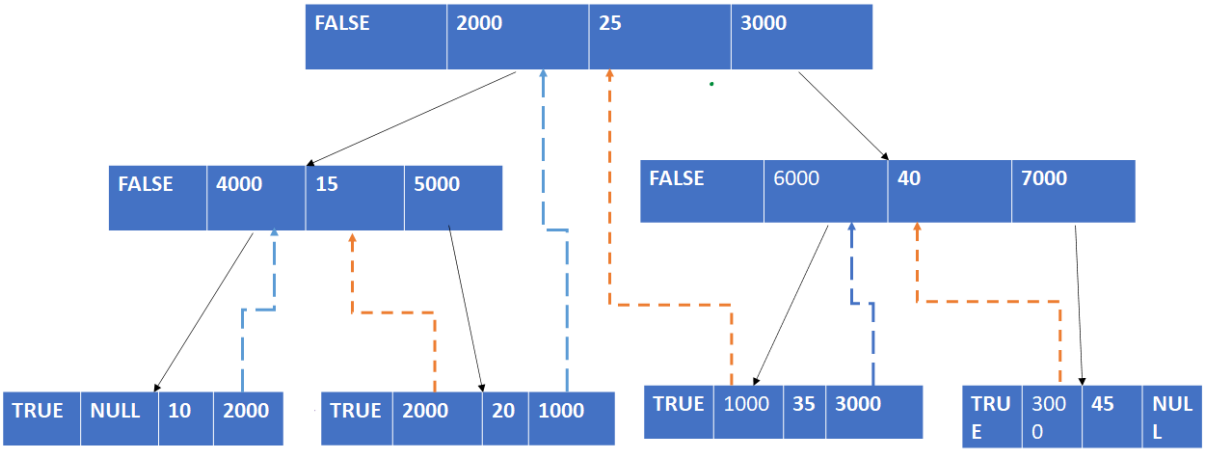
- the left memory location of node 20 replaced by the address of 15
- the left memory location of node 35 replaced by the address of 25
- the left memory location of node 45 replaced by the address of 40



10.11.5 Double Threaded Binary Tree

If both the NULL Fields in a node are to be replaced with valid addresses

- An in-order traversal of a given tree is 10 15 20 25 35 40 45
- the left memory location of node 10 is not replaced by any address because there is no in-order predecessor but the right location is replaced by the address of 15.
- left and right memory location of node 20 replaced by the address of 15 and 25
- left and right memory location of node 35 replaced by the address of 25 and 40
- the left memory location of node 45 replaced by the address of 40, and right memory location not replaced because there is no in-order successor.



10.11.6 Why Threaded Binary Tree

Memory locations (NULL pointers) are unused. Adjust these memory locations in such a way so that traversal becomes easier.
In-order traversals can be done on binary search tree in two ways-

- Recursion
- Use Auxiliary stack

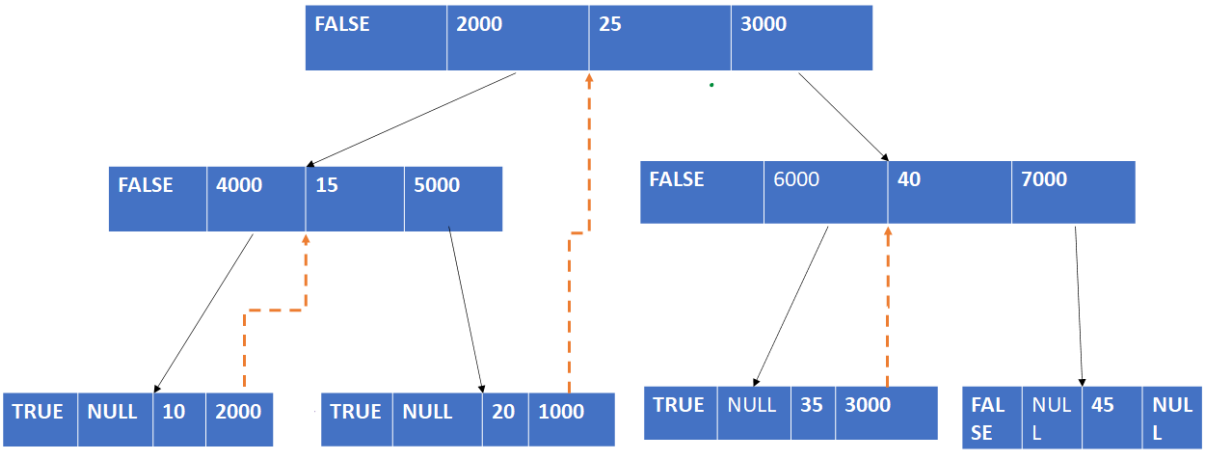
By applying the concept of the threaded binary tree, make traversal easy and faster without using recursion and auxiliary stack.
Inorder Traversal (Iterative): make traversal easy and faster by iteration, and this can be by using thread easily.

10.11.7 Traversal in Threaded Binary Tree

ALGORITHM NodeLeftMost(ROOT)
BEGIN:

```
IF ROOT==NULL THEN
    RETURN NULL
WHILE ROOT→left_Child!=NULL DO
    ROOT=ROOT→left_Child
RETURN ROOT
```

END;



Now traverse the node iteratively.

ALGORITHM inorderTraversal(ROOT)
BEGIN:

```
ROOT=NodeLeftMost(ROOT)
```

```
WHILE ROOT!=NULL DO
        WRITE ROOT→ data DO
        IF ROOT→is_right_Thread THEN
                ROOT=ROOT→right_Child
        ELSE
                ROOT=node_Left_Most(ROOT->right_Child)
END;
```

COMPLEXITY: Algorithm takes O(n) time.

10.12 Applications of Tree

10.12.1 Huffman Coding

Huffman Coding is an application of Binary Tree. It is a technique of compressing data to reduce its size without losing any of the details. The technique was initially developed by David Huffman.

Huffman Coding is generally useful to compress the data in which there are frequently occurring characters. In order to send any data file over the network, the data file can be compressed and send to reduce the cost of transmission.

Huffman coding provides a method for generating its own variable size code for text compression. The variable size code depends on the frequency of the character. For example, if an input character has occurred more times, its variable size code will be shorter. The coding of characters is assigned in the Prefix code pattern, which means no two-character codes are prefixes of each other. For example, code (00, 11) has prefix code property, and code (00, 11, 1) does not as 1 being the prefix of 11.

10.12.1.2 Understanding Huffman Approach:

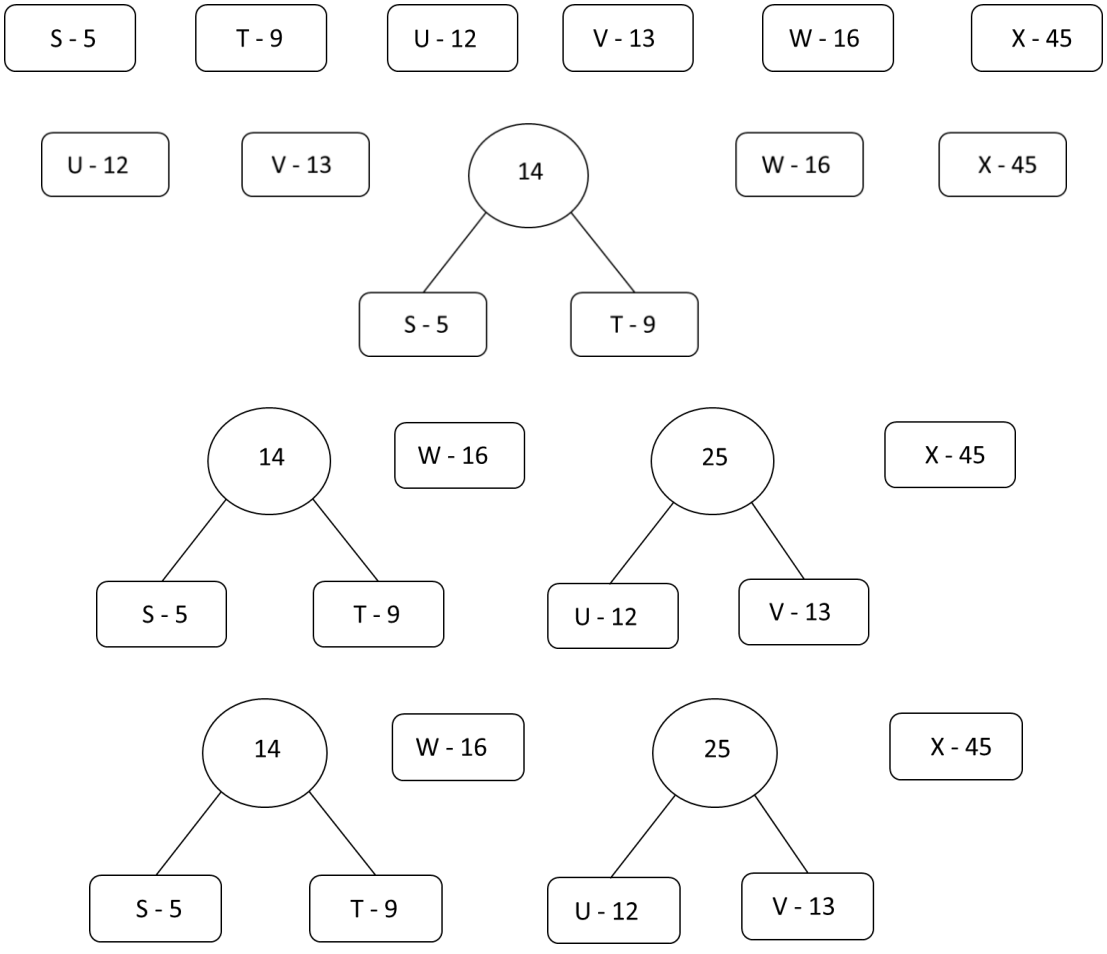
The following step by step analysis gives a view of how a Huffman tree is constructed:

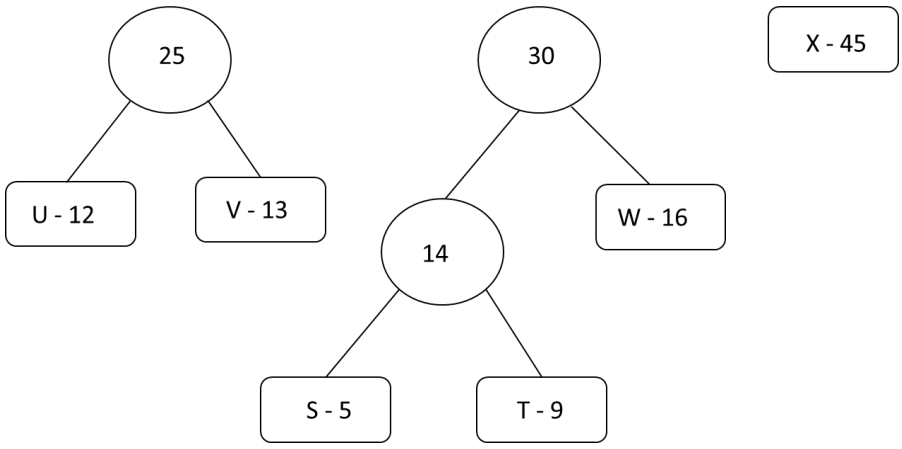
1. For the given initial string, calculate the character frequency or number of times each character repetition count.
2. Store the characters in Priority Queue in increasing order of their frequencies.
3. Each unique character will be a leaf node in the final Huffman Tree.
4. Create a new node by adding two minimum frequencies. The left child of the node created will be the least frequency between the two, and right child is the second minimum frequency.
5. Remove the above-added minimum frequency from the Priority Queue and add the new node created with sum into the list of frequencies.
6. Repeat the above steps from 3 to 5 for all nodes until a single tree is left.

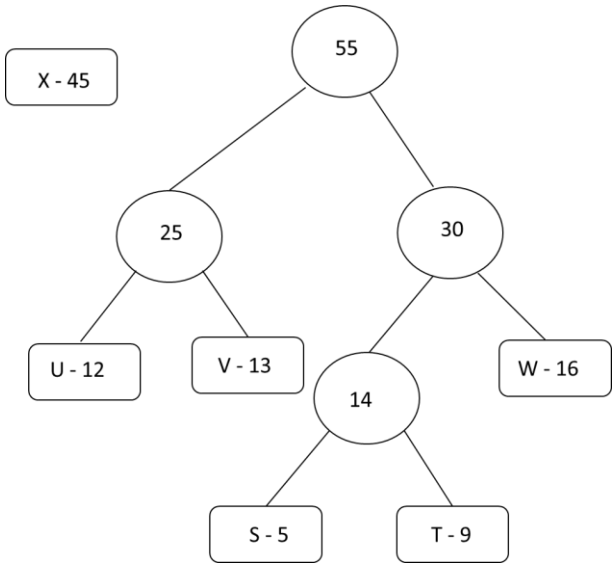
The above building Huffman Tree will provide the Huffman code.

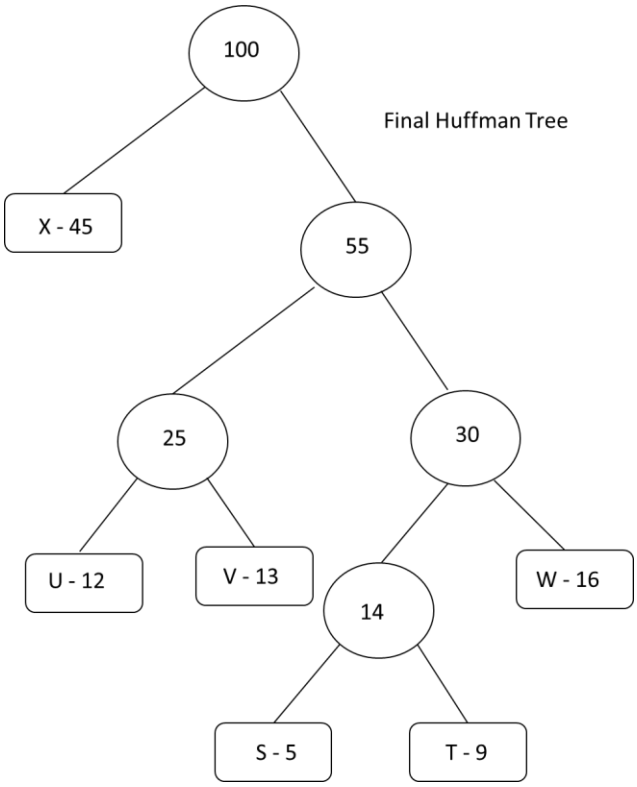
Let us perform the above for a given input string to be sent over a network:

Character	Frequency
S	5
T	9
U	12
V	13
W	16
X	45

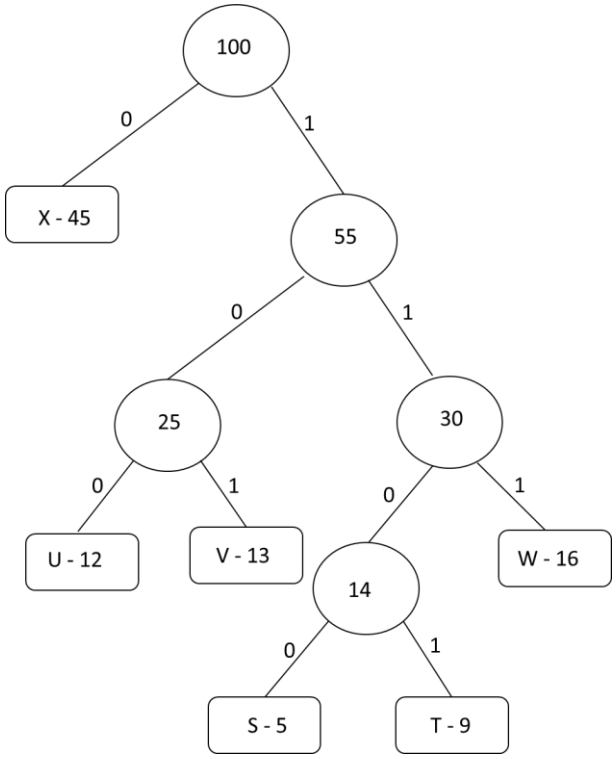



25
30
X - 45
U - 12
V - 13
14
W - 16
S - 5
T - 9


55
X - 45
25
30
U - 12
V - 13
14
W - 16
S - 5
T - 9



Once the tree is built, we Can generate code from the tree by traversing from the root node and assigning 0 to the left child edge and 1 to the right child edge.



For sending the above string over a network, we have to send the tree and the above compressed code.

Now, with the below table we must find out how many bits Huffman Coding is saving if the same string sends over network with variable-sized codes.

Formulas to calculate Huffman encoded bits:

1. Average code length per character = $\frac{\sum (\text{Frequency}_i * \text{Code Length}_i)}{\sum(\text{Frequency}_i)}$
2. Total number of bits in Huffman encoded message = Total number of characters in the message * Average code length per character

Character	Frequency	Codes	Size
S	5	1100	5*4=20
T	9	1101	9*4=36
U	12	100	12*3=36
V	13	101	13*3=39
W	16	111	16*3=48
X	45	0	45*1=45
6*8=48 bits	100 bits		224 bits

Thus for the above table average code length per character = 224/100 = 2.24

Total number of bits in Huffman encoded message = 100 * 2.24 = 224

If the message would have been transmitted over the network without encoding, then total bits= 100 * 8= 800 bits. So, we saved 800 – 224 = 576 bits.

Time Complexity: O(nlogn) as extracting the least frequency from the priority queue takes place 2 *(n-1) times. Thus the overall complexity for encoding each unique character is O(nlogn).

ALGORITHM HuffmanTree(A[ ], N)

// where A is the information about the character & their frequencies, and N is the total number of character appearing in the text file.

BEGIN:

```
Initialize(PQ) //Initialize a PriorityQueue PQ, which contains N elements in A
FOR I = 1 to N DO
    Z=MakeNode(A[i])
    PQInsert(PQ, Z)
```

```
FOR I = 1 to N-1 DO
        X= PQDelete(PQ)
        Y= PQDelete(PQ)
        Z= MakeNode( )
        Z→Data = X→Data + Y→Data
        Z→Left = X
        Z→Right = Y
        PQInsert(PQ, Z)
END;
```

Huffman Coding Applications

Huffman coding is used in conventional compression formats like GZIP, BZIP2, PKZIP, etc. for text and fax transmissions.

10.12.2 Expression Tree

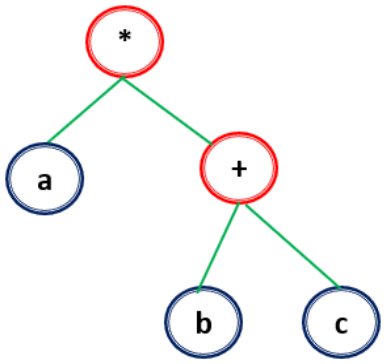
The expression tree refers to the tree where each internal node refers to the operator while each leaf node represents the operands.

Hereby, operator, we mean unary operator, binary operator. For constructing a binary tree, we will use here binary operator like +, *. Note that binary operators are those operators that require two operands in order to perform some operation.

By operand, we mean simple objects that are capable of being manipulated. For example, adding 2 and 3 will fetch 5 as an answer.

Example of Expression tree: -

Let's take an expression tree, **a * b + c**. Here this tree can be shown as below: -



Expression tree- example

Here operands are on the leaves node or are as external node, and operators are on an internal node.

10.12.2.1 Constructing Expression Tree using Infix Expression

In order to construct Expression tree using Infix Expression we need to take care of the precedence rule.

Power operator is having Right to left precedence,

Multiplication and Divide is having left to right precedence, and

Addition and Subtraction are having left to right precedence.

The order of computation is from top to bottom.

Let's draw a tree using the given infix expression. The expression is given as: -

a * b / c + e / f * g + k – x * f

On seeing the above expression, we found that * and / are having higher precedence than + and – operator.

Step 1

The operand that will lie on the leaf node will be evaluated first, so the operator which is having the highest priority will be evaluated firstly. Here we can see that both * and / are having the highest priority and are having left to right precedence order, and both - and + are having lowest priority and are having left to right precedence order, so the root of tree will be last – or + encountered from the right side.

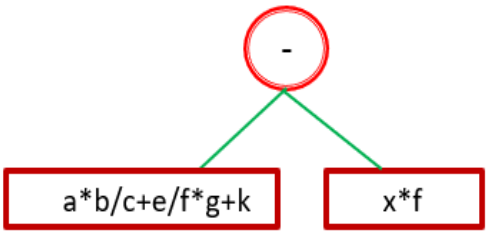


Fig:- minus (lowest priority) will be the root of the tree as it lies last in the expression

Step 2

Now looking towards the left subtree, we found that + lies last from the right side and is having the least priority, so it will be the root node of this subtree.

Here expression a*b/c+e/f*g will lie on the left side while k will lie on right side.

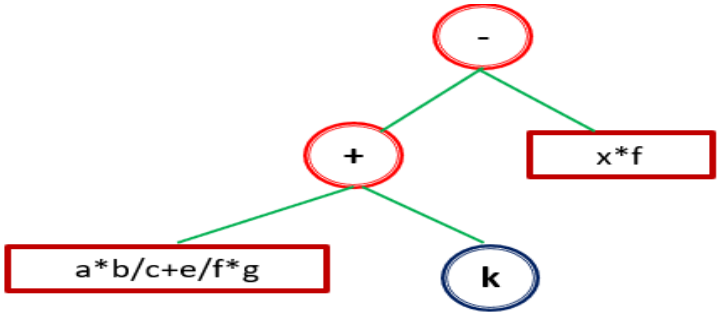


Figure: + will be the root of the left subpart of the tree

Step 3

Now looking towards the left subtree, we found that + lies last from the right side and is having the least priority, so it will be the root node of the subtree obtained in step 2.
Here expression a*b/c will lie on the left side while e/f*g will lie on the right side.

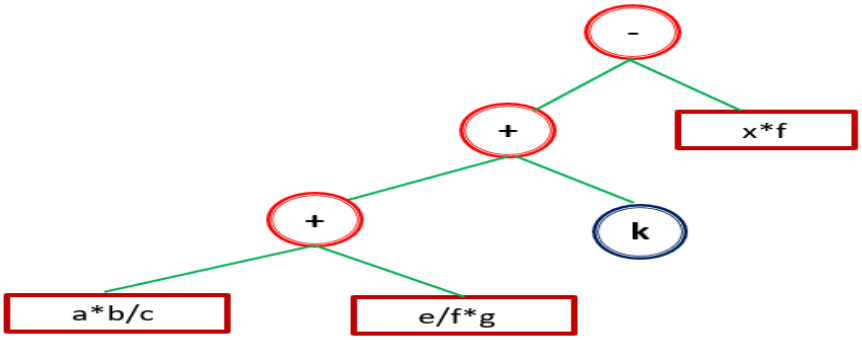


Figure: + will be the root of left subpart of tree

Step 4

Now looking towards the left subtree obtained above, we found that / lies last from right side and is having LR precedence so it will be the root node of the subtree obtained in step 3.
Here expression a*b will lie on left side while c will lie on right side.

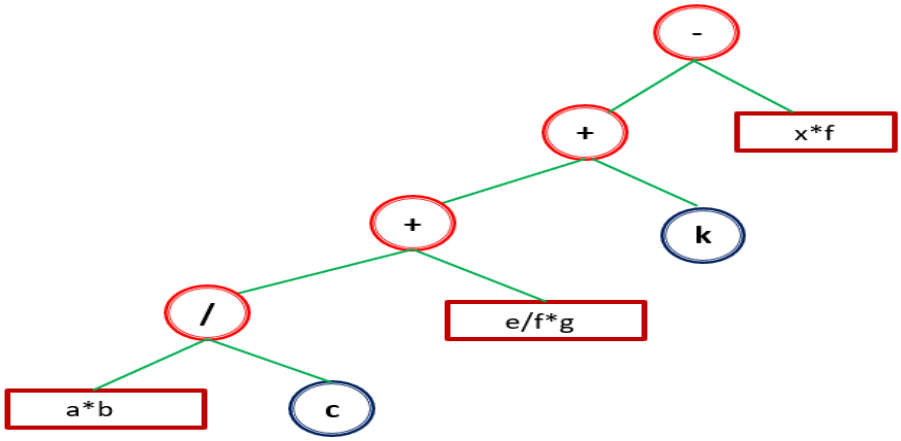


Figure: / will be the root of the left subpart of the tree

Step 5

Now looking towards the left subtree obtained above, we found that * lies last from the right side and is the last operator left, so it will be the root node of the subtree obtained in step 4. Here expression a will lie on the left side while b will lie on the right side.

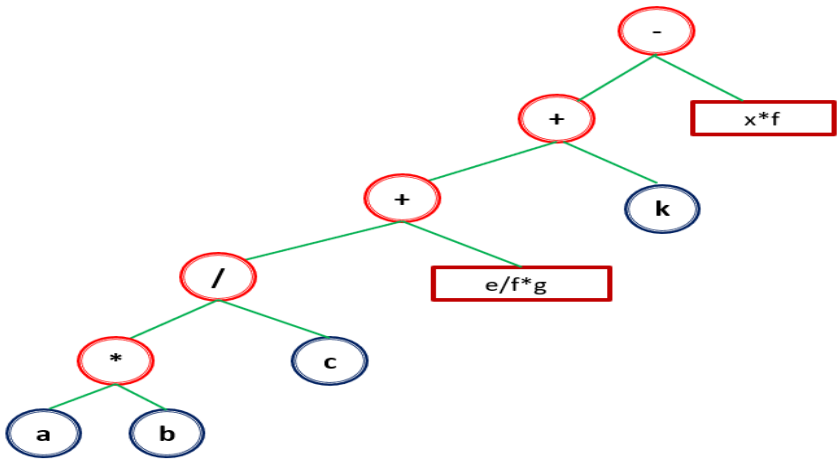


Figure: - * will be the root node of the left subpart

Step 6

Now since there exists no subpart so after backtracking, we found the expression e/f*g that lies as a right child of + node, so expanding this we found * to be the root of this right subpart of the tree as it is the last one to traverse from the right side.

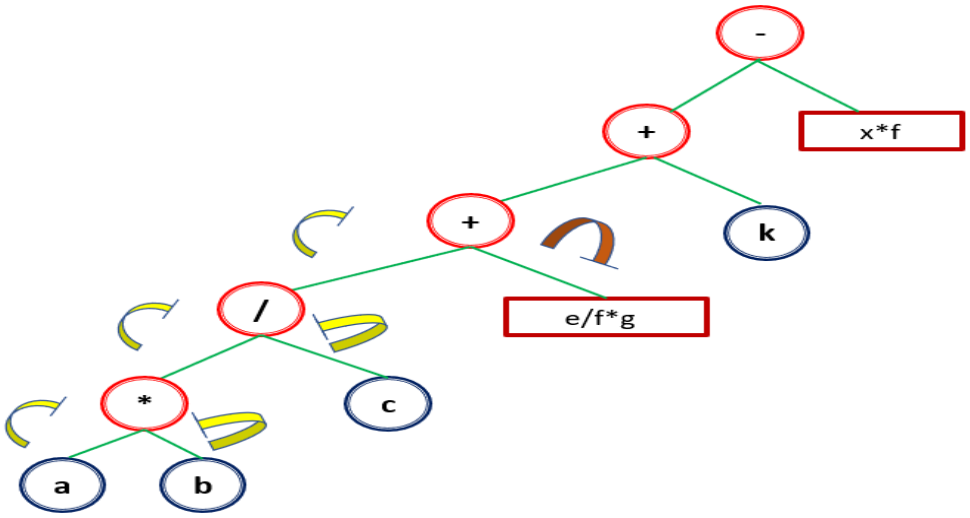


Figure: - Various backtrack performed

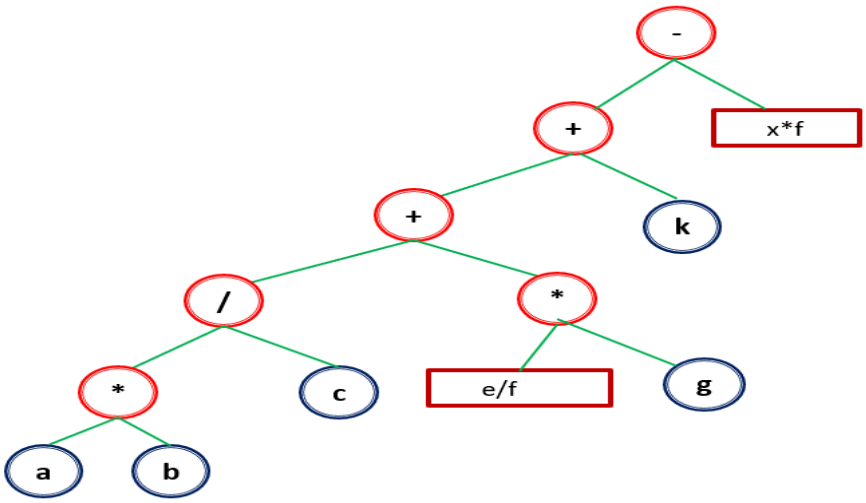


Figure: - * will be the root node of the right subpart

Step 7

Now looking towards the left subtree obtained above in step 6, we found that / lies last from the right side and is the last operator left so it will be the root node of the subtree obtained above.

Here, in the expression tree, e will lie on the left side while f will lie on right side.

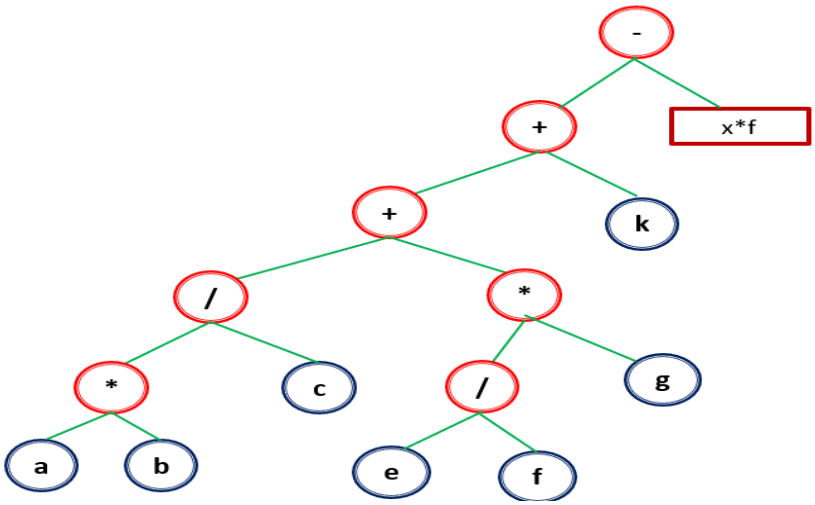


Figure: - / will be the root of this subpart of tree

Step 8

It can be found easily as there are no nodes left in the left subpart of the tree, so we backtrack till we reached the right side of the root node.

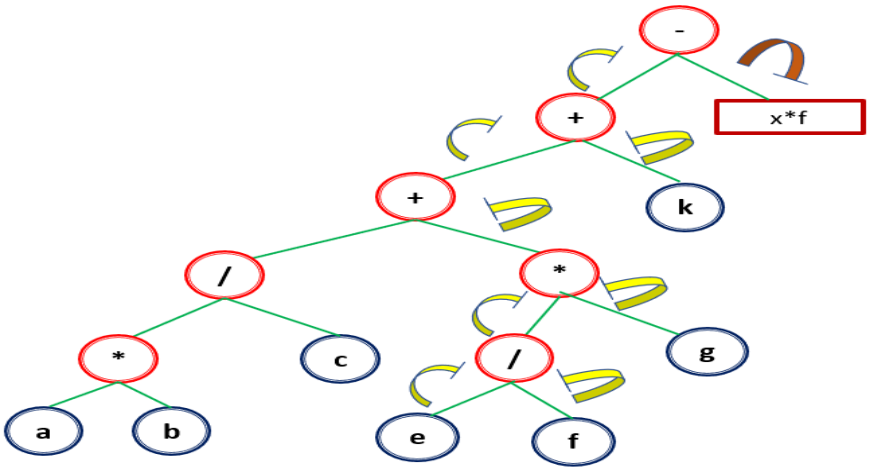


Figure: - Backtracking in order to reach live node

Now looking towards the left subtree obtained above in step 7, we found that * lies last from the right side and is the last operator left, so it will be the root node of the subtree obtained above.

Here, in the expression tree, x will lie on the left side while f will lie on the right side.

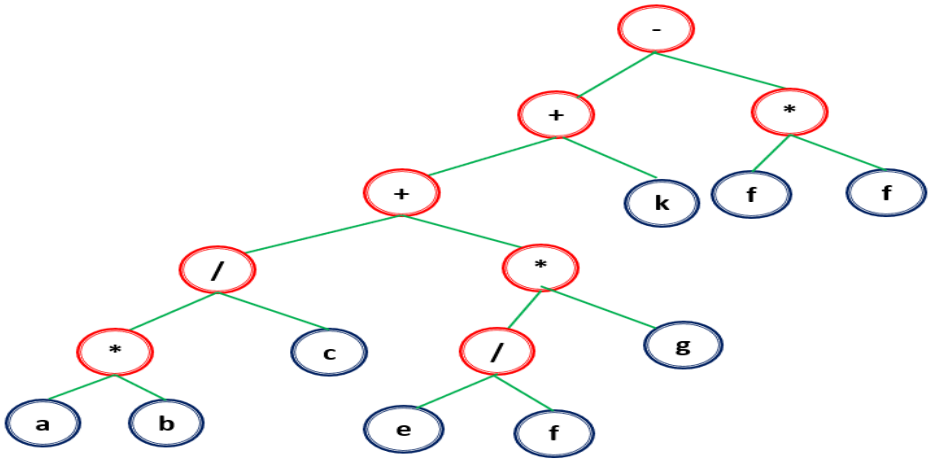


Figure: - Final expression tree

10.12.2.2 Algorithm for building Expression Tree from Infix Expression

The Binary Tree node is supposed to contain the data (operand or operator), left child address, and right child address. The Nodes will be added to the Stack for processing. Considering this, we have taken another field named next.

There are two stacks considered here. One stack (implemented with Linked List) will contain the operands and constructed subtrees, and the other is used for operators.

General functions Push, Pop, IsEmpty, and StackTop are called while building the expression tree.

Prcd is the precedence function that finds if the first operator has precedence over the second one.

```
ALGORITHM prcd(x, y)
BEGIN:
        IF x=='(' OR y=='(‘ THEN
                RETURN FALSE
        ELSE
                IF y==')' THEN
                        RETURN TRUE
                ELSE
                        IF x=='^' OR x=='*' OR x=='/' THEN
                        IF y=='^'THEN
                                RETURN FALSE
                        ELSE
                                RETURN TRUE
                ELSE
                        IF x=='+'||x=='-' THEN
                        IF y=='+'||y=='-' THEN
                                        RETURN TRUE
                                ELSE
                                        RETURN FALSE
END;
/************************************************************/
ALGORITHM IsOperand(x)
BEGIN:
        IF x>='0'&&x<=9 OR x>='a'&&x<='z' OR x>='A'&&x<='Z' THEN
                RETURN TRUE
        ELSE
                RETURN FALSE
END;
/************************************************************/
ALGORITHM BuildExpressionTree(InfixExp[])
BEGIN:
        i=0;
        LLS=NULL        // The Linked List stack
        S                //The character stack
        InitializeStack(S)
```

```
WHILE InfixExp[i]!='\0' DO
        x=InfixExp[i]
        IF IsOperand(x)) THEN          //Operand, Push in Linked List Stack
                p=GetNode(x)
                p→next=LLS
                LLS=p
        ELSE   //Operator
                WHILE !IsEmptyStack(S) AND prcd(StackTop(S),x) DO
                        q=LLS
                        LLS=LLS→next
                        p=LLS
                        LLS=LLS→next
                        y=Pop(&S)    //operator
                       r=GetNode(y)   //Make the node with the operator
                        r→left=p              //Left Child
                        r→right=q             //Right Child
                        r→next=LLS
                        LLS=r
                IF x!=')' THEN
                Push(S,x)
                ELSE
                Pop(S)
                i++

WHILE !IsEmptyStack(S) DO
        q=LLS
          LLS=LLS→next
          p=LLS
          LLS=LLS→next

 y=Pop(S)    //operator
          r=GetNode(y)  //Make the node with the operator
          r→left=p     //Left Child
          r→right=q    //Right Child
          r→next=LLS
          LLS=r
  RETURN LLS;
END;
```

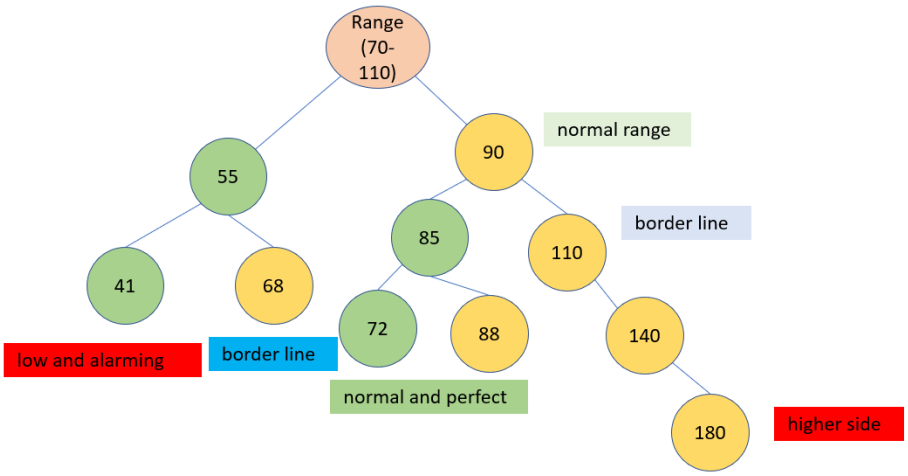
10.13 Binary Search Tree

10.13.1 Introduction

Consider the simple example of monitoring the diabetes level(fasting) of an individual. The normal range lies between 70 to 110. A value greater than 70 will lie on the right side, and value less than 70 will fall on the left side. We have taken several cases based on this in order to make the decision. Suppose the value is 180, for example, it will fall on the right side and be considered high risk. Similarly, if the value is 110, it will also lie on the right side and be considered at the borderline. A value less than 70 will fall on the left-hand side, and based upon categorization; it can fall either at borderline, less severe, and high severe.

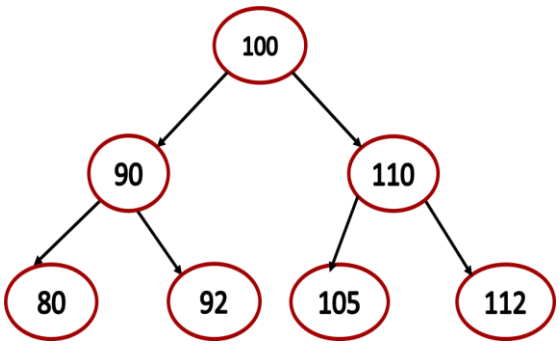
Hence, the Binary Search Tree also works, and a value greater than that of the root node will fall on the right-hand side while less than that will fall on the left-hand side.

Note:- The in-order traversal of the tree will be in sorted order.



Therefore, **A binary Search tree** is the extended form of binary tree in which if any node N has value K then It should have the following properties:

a) All values in the left subtree of node N will be less than its value K.
b) All values in the right subtree of node N will be greater than its value K.
(Assume that BST contains distinct values.)



10.13.2 Linked list Representation of BST

In linked representation, BST is a collection of nodes where each node has at least four fields.

Left address Field	Address to Parent node	Data Field	Right address Field

a) **Left address Field** contains the address of the left child of the node.
b) **Address to the Parent node** contains the address of the parent node.
c) **Data Field** contains the actual value to be stored in the node.
d) **The right address Field contains the** address of the right child of the node.

10.13.3 Operations in BST

1) Searching
2) Insertion
3) Successor
4) Predecessor
5) Deletion.

10.13.3.1 Searching

Method 1 Iterative Method: This algorithm will search whether an ITEM is present in BST or not. If it is present, it will return the address of that node or return NULL if it is not present in BST.

In the search operation of a BST, we start with the root and keep moving left or right using the BST property. If the data we are searching for is the same as data at node, then we return the current node. If the data we are searching is less than nodes data, then start the search in the current node's left subtree; otherwise, search the right subtree of the current node. If the data is not present, we end up in a NULL

```
ALGORITHM SEARCH_BST_I(ROOT, ITEM)
BEGIN:
        WHILE ROOT! = NULL DO
           IF ITEM ==ROOT→DATA THEN
                 Return ROOT
           ELSE IF ITEM< ROOT→DATA
                ROOT = ROOT→LEFT
           ELSE
                 ROOT = ROOT→RIGHT
        RETURN NULL
END;
```

Complexity of Operation

Time Complexity: In general, the time complexity of Search operation in BST is O(h), where h is the height of the BST.

In worst-case, iF BST is either Right Skewed BST or Left Skewed BST, then the time complexity of Search operation in BST is O(N) complexity where N is the number of nodes in BST.

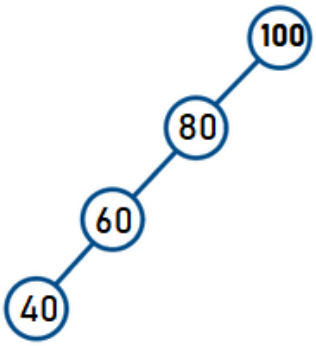


Fig-Left Skewed BST

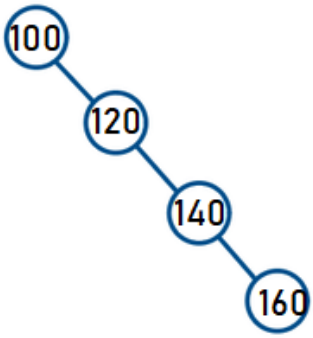


Fig-Right Skewed BST

In the Best or Average case, BST is either complete BST or Almost complete BST, so the height of BST will be either $\log_2 N$ or almost $\log_2 N$. So time complexity is order of $\log_2 N$.

Space Complexity = O(1)

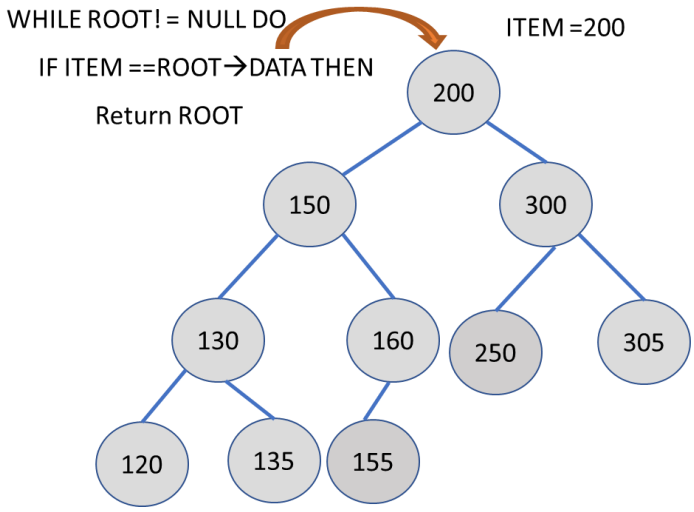


Figure-1 When item is present at Root Node in BST

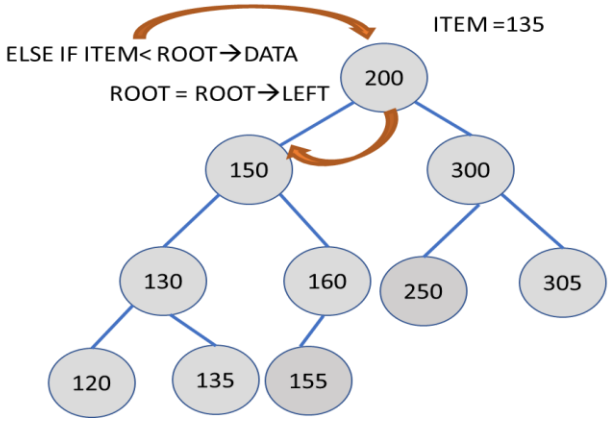


Figure -2

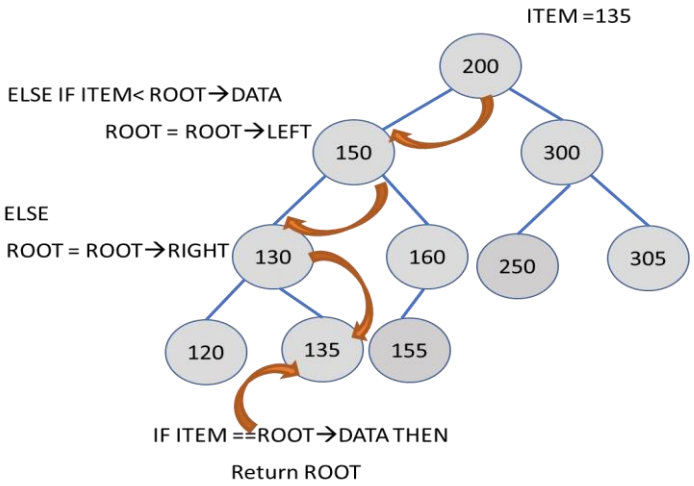


Figure-3 When Item is present in BST

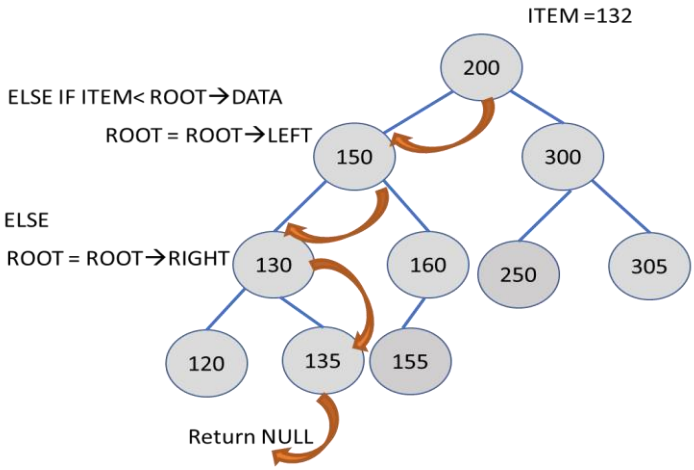


Figure-4 When item is not present in BST

Method 2 Recursive Method: To search an element in BST.

ALGORITHM SEARCH_BST_R(ROOT, ITEM)

BEGIN:

```
IF ROOT == NULL THEN
        Return NULL
IF ITEM ==ROOT→DATA THEN
        Return ROOT
ELSE IF ITEM< ROOT→DATA
        SEARCH_BST_R(ROOT→LEFT, ITEM)
ELSE
        SEARCH_BST_R(ROOT→RIGHT, ITEM)
```

END;

Complexity of Operation

Time Complexity: In general, the time complexity of Search operation in BST is O(h), where h is the height of the BST.

In Worst case, IF BST is either Right-Skewed BST or Left-Skewed BST then the time complexity of Search operation in BST is O(N) complexity, where N is the number of nodes in BST.

In Best or Average case, BST is either complete BST or Almost complete BST, so the height of BST will be either $\log_2 N$ or almost $\log_2 N$. So time complexity is order of $\log_2 N$.

Space Complexity = O(h), where h is the height of the BST.

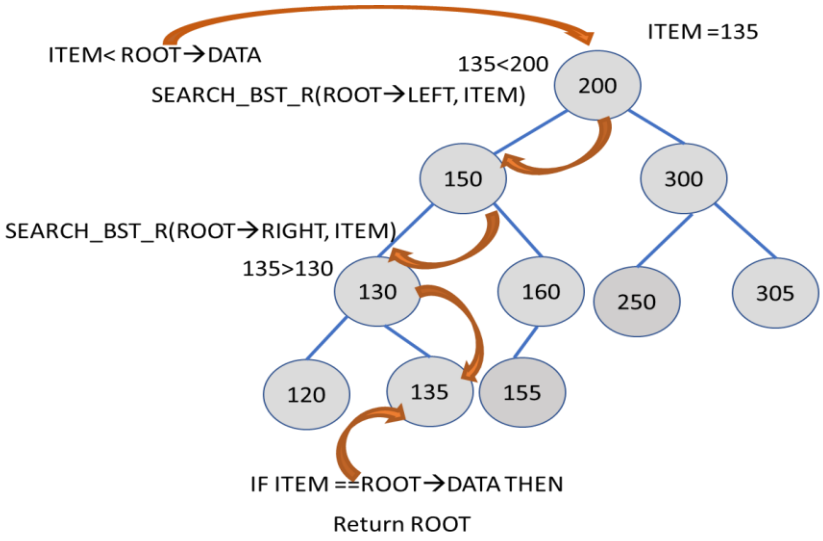


Figure 5

10.13.3.2 Insertion

a) Iterative Method
b) Recursive Method

Algorithm GetNode()
BEGIN:

```
P→Left = NULL
P→Right = NULL
P→Father = NULL
Return P
```

END;

a) Iterative Method: To insert an item in BST. For insertion in BST, if BST is empty initially then new node will be inserted as root node. If BST is not empty then item to be inserted is compared with the value in the current node,
if this new value is found to be greater than the current node value then new node will be inserted in the right subtree of the current node else new node will be inserted in the left subtree of the current node. This procedure is repeated until we find the appropriate leaf node either in the left or right subtree. Then insert new node either as left or right child of this leaf node.

ALGORITHM InsertBST_I(ROOT, item)
BEGIN:

```
P = GetNode()
P→Data = item
IF ROOT == NULL THEN
        ROOT = P
ELSE
        R = ROOT
        Q = NULL
        WHILE R! = NULL DO
                IF item < R→Data THEN
                        Q = R
                        R = R→Left
                ELSE
                        Q = R
                        R = R→Right
        IF item < Q→Data THEN
```

Q→Left = P

P→Father = Q

ELSE

Q→Right = P

P→Father = Q

END;

Complexity of operation:

Time Complexity: In general, the time complexity of Insert operation in BST is O(h), where h is the height of the BST.

In Worst case, IF BST is either Right-Skewed BST or Left-Skewed BST then the time complexity of Insert operation in BST is O(N) complexity where N is the number of nodes in BST.

In Best or Average case, BST is either complete BST or almost complete BST, so the height of BST will be either $\log_2 N$ or almost $\log_2 N$. So time complexity is order of $\log_2 N$.

Space Complexity = O(1)

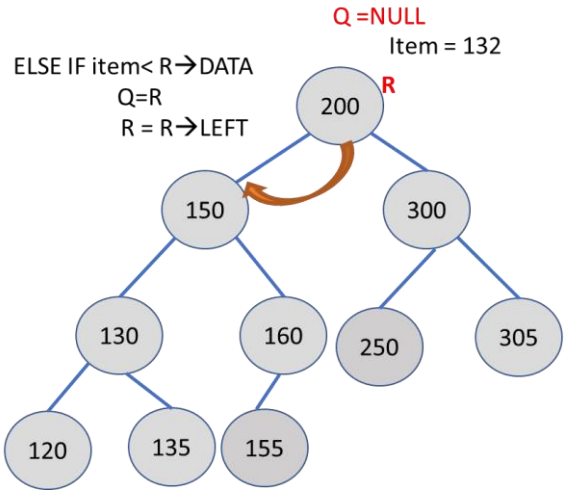


Fig-6

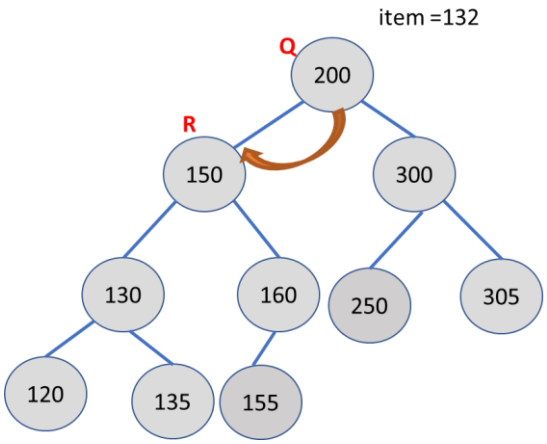


Fig-7

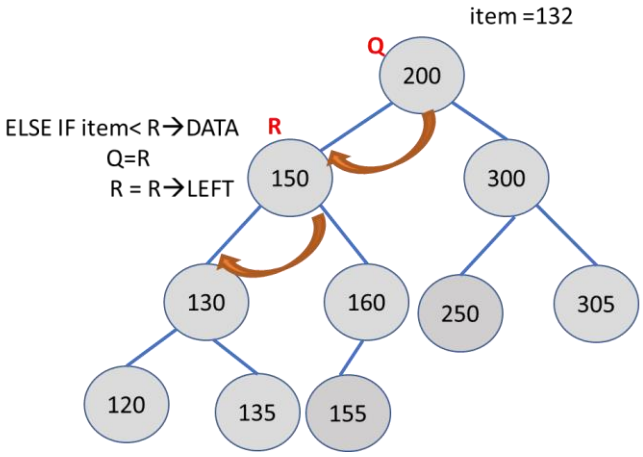


Fig-8

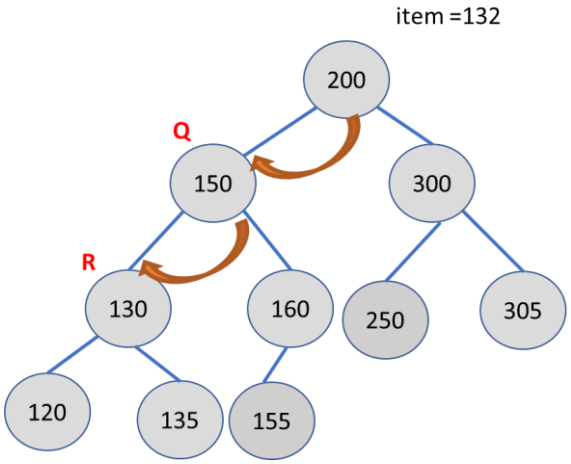


Fig-9

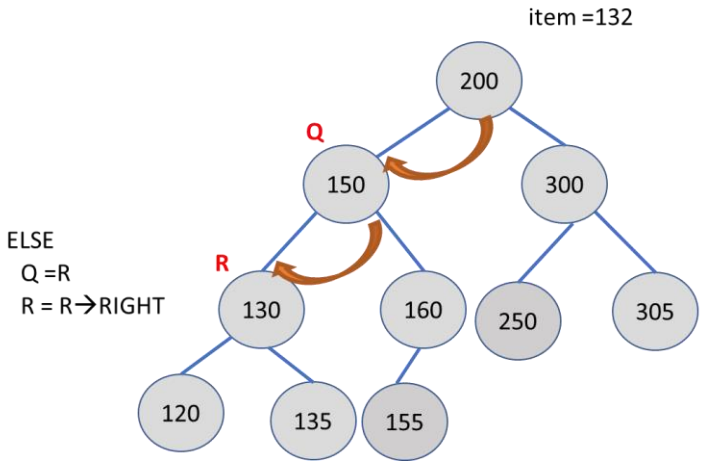


Fig-10

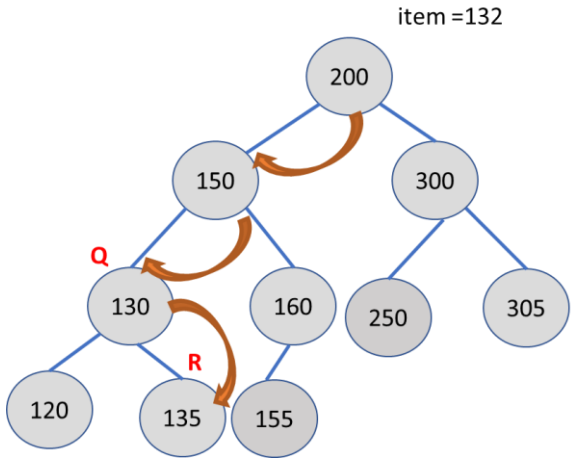


Fig-11

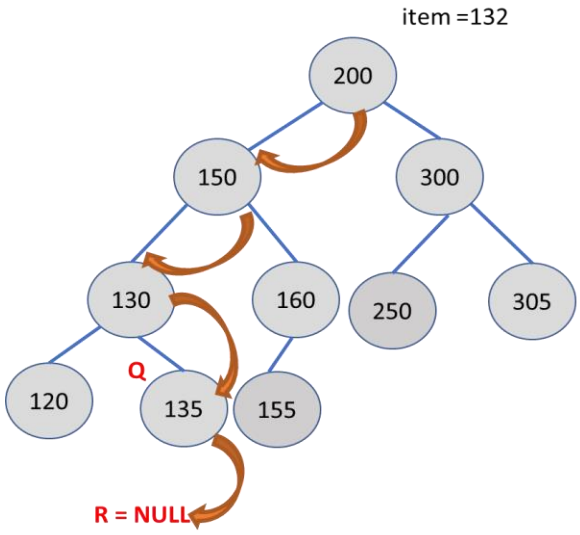


Fig-12

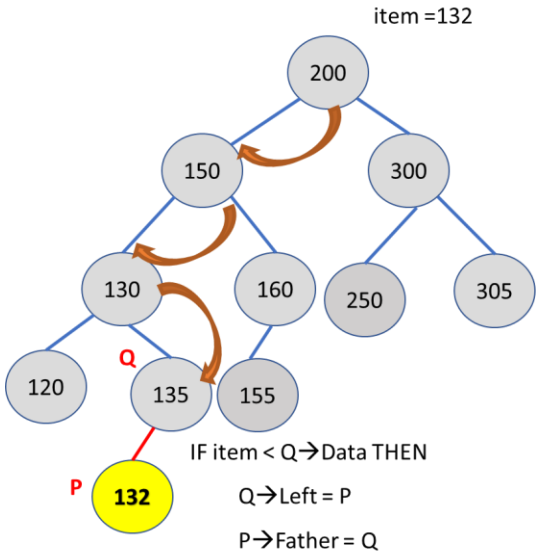


Fig-13

Recursive Method: To insert an item in BST

```
ALGORITHM InsertBST_R(ROOT, Q , Item)
BEGIN:
    IF ROOT == NULL THEN
        P = GetNode()
        ROOT = P
        P→Father = Q
    ELSE IF Item < ROOT→Info THEN
        ROOT→Left = InsertBST_R(ROOT→Left, ROOT, Item)
    ELSE
        ROOT→Right = InsertBST_R(ROOT→Right, ROOT, Item)
    RETURN ROOT
END;
```

Complexity of operation:

Time Complexity: In general, the time complexity of Insert operation in BST is O(h), where h is the height of the BST.

In Worst case, IF BST is either Right-Skewed BST or Left-Skewed BST then the time complexity of Insert operation in BST is O(N) complexity where N is the number of nodes in BST.

In Best or Average case, BST is either complete BST or Almost complete BST, so height of BST will be either $\log_2 N$ or almost $\log_2 N$. So time complexity is order of $\log_2 N$.

Space Complexity = O(h), where h is the height of the BST.

10.13.3.3 Methods to check whether a given node is Left child or Right child of its Parent node

This method will find that any node having address Q is either left child or right child of its Parent node.

If the address of the left child of father node of Q is the same as Q then it is the left child of its parent node; otherwise, it is the right child.

```
ALGORITHM IsLeft_BST(Q)
BEGIN:
    IF Q→Father→Left == Q THEN
        RETURN True
    ELSE
        RETURN False
END;
```

The complexity of operation:

Time Complexity = O(1) if the address of node Q is given

Space Complexity =O(1)

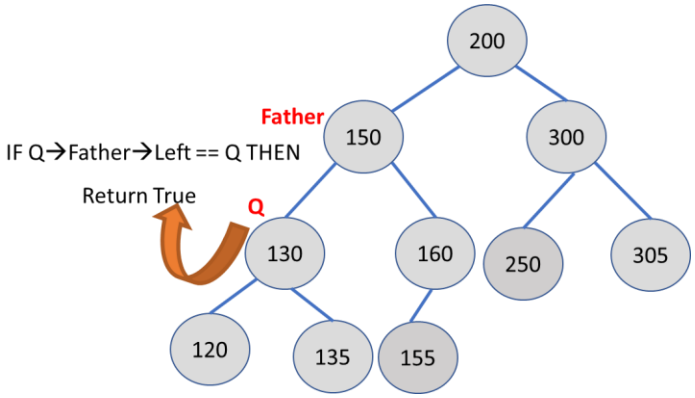


Fig-14

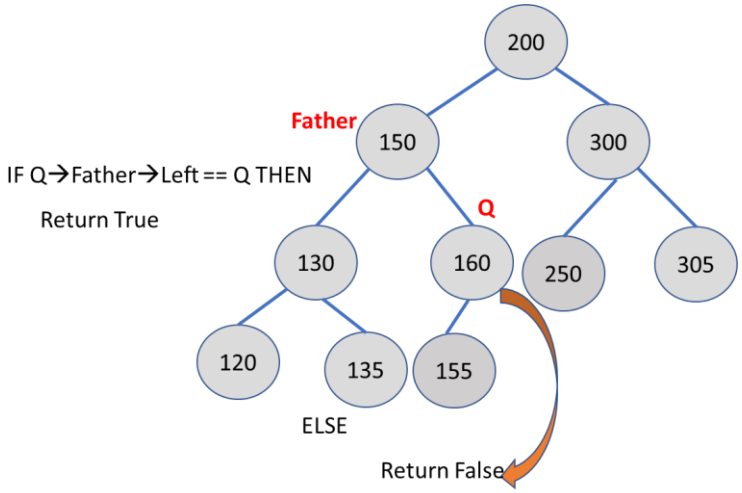


Fig-15

ALGORITHM IsRight_BST(Q)

BEGIN:

IF Q→Father→Right == Q THEN

Return True

ELSE

Return False

END;

Complexity of operation:

Time Complexity = O(1) if the address of node Q is given

Space Complexity =O(1)

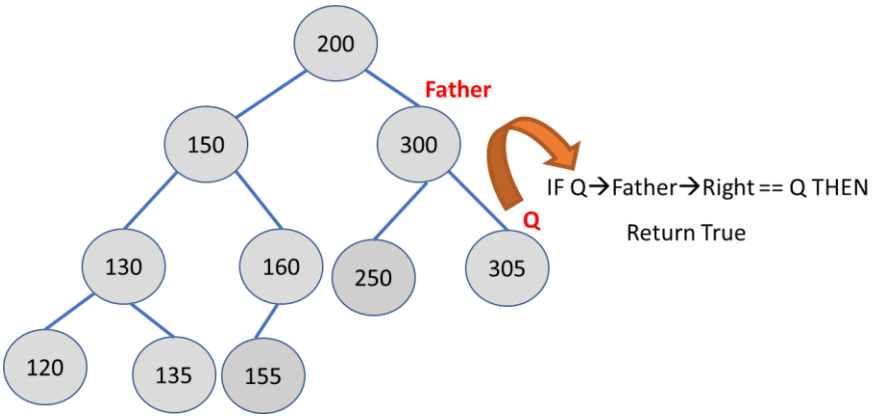


Fig-16

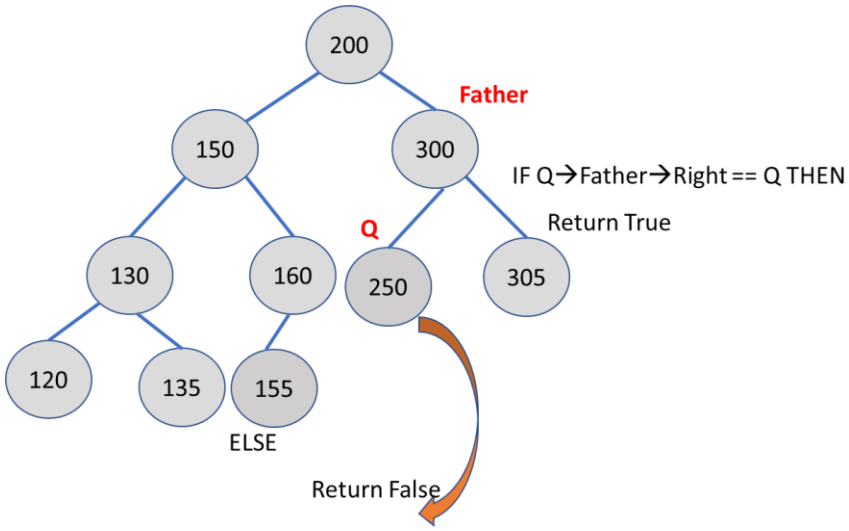


Fig-17

10.13.3.4 Finding node containing minimum value in BST

This method will find the smallest or minimum value in the BST. In BST minimum values are present at the leftmost node, which does not have the left child. There are two methods to find the minimum value in BST:

a) Iterative Method.

b) Recursive Method.

Using an iterative method

ALGORITHM Minimum_BSTI(ROOT)

BEGIN:

```
WHILE ROOT→Left! =NULL DO
        ROOT = ROOT→Left
RETURN ROOT
```

END;

Complexity of operation:

Time Complexity: In general, the time complexity to find the minimum value in BST is O(h), where h is the height of the BST.

Space Complexity = O(1)

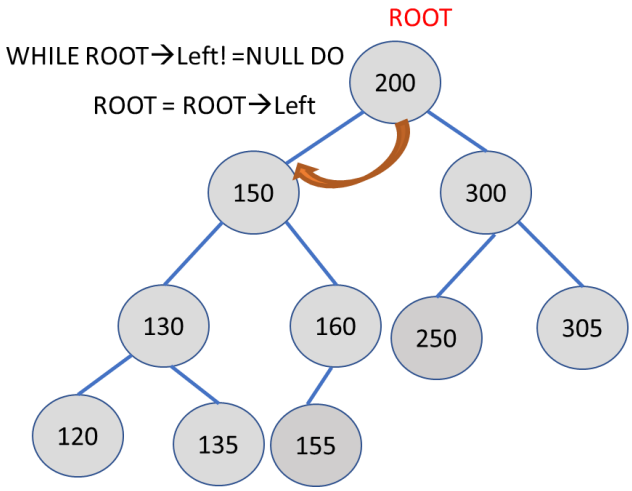


Fig-18

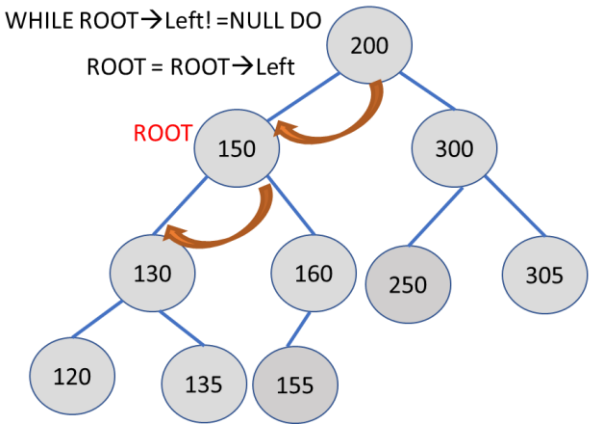


Fig-19

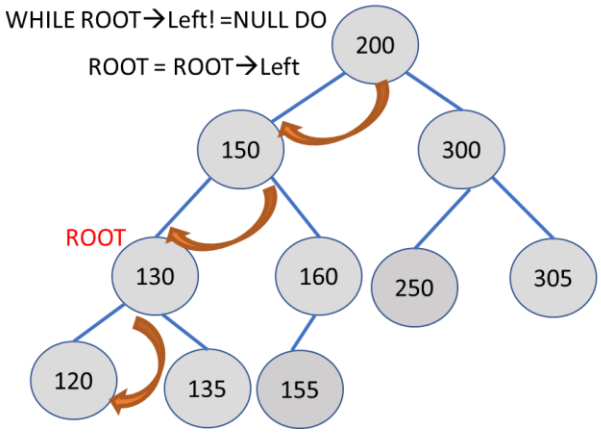


Fig-20

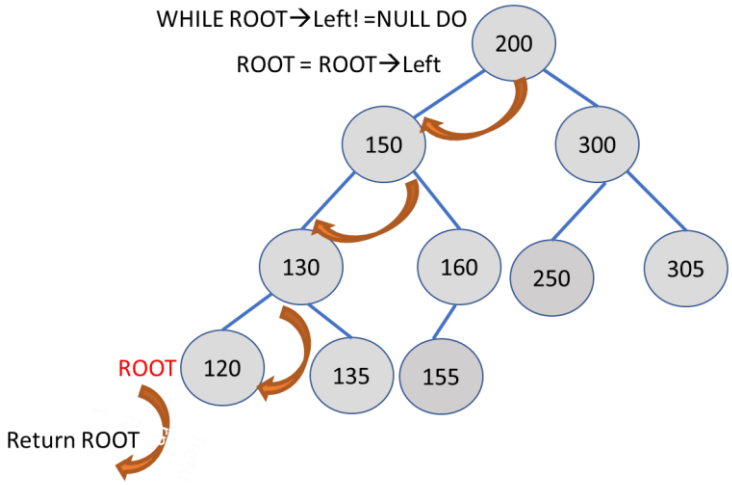


Fig-21

Using a recursive method

ALGORITHM Minimum_BSTR(ROOT)
BEGIN:

IF ROOT→Left ==NULL THEN
RETURN ROOT
Minimum_BSTR(ROOT→Left)

END;

Complexity of operation:

Time Complexity: In general, the time complexity to find the minimum value in BST is O(h), where h is the height of the BST.

Space Complexity = O(h), where h is the height of the BST.

10.13.3.5 Finding node containing minimum value in BST

This method will find the largest or maximum value in the BST. In BST, maximum value is present at the rightmost node, which does not have the right child. There are two methods to find the maximum value in BST:

a) Iterative Method.

b) Recursive Method.

Using iterative method

ALGORITHM Maximum_BST(ROOT)

BEGIN:

WHILE ROOT→Right! =NULL DO

ROOT = ROOT→Right

RETURN ROOT

END;

Complexity of operation:

Time Complexity: In general, the time complexity to find the largest value in BST is O(h) where h is the height of the BST.

Space Complexity = O(1)

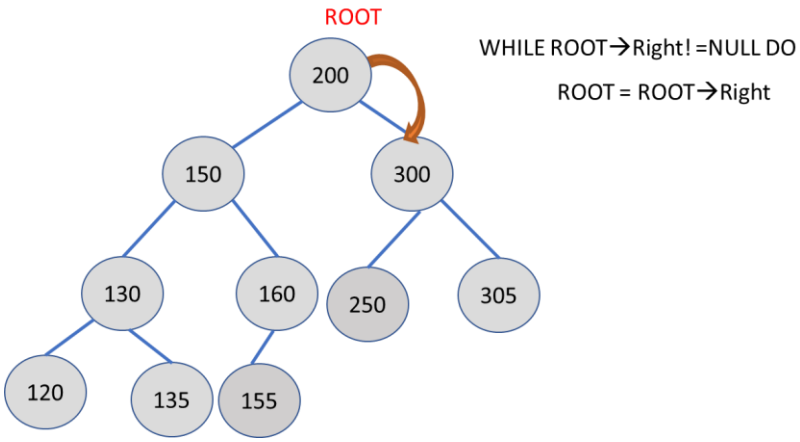


Fig-22

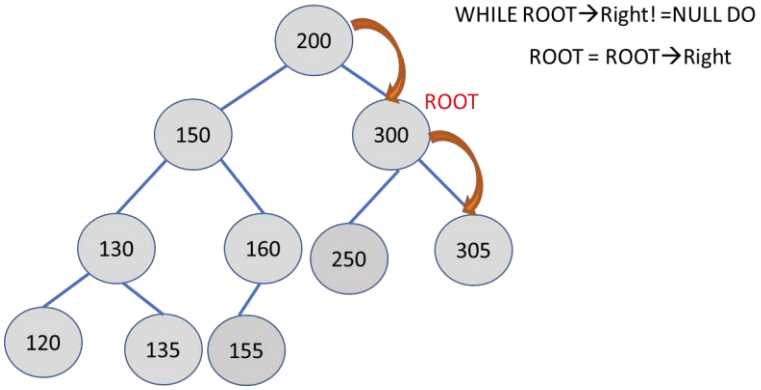


Fig-23

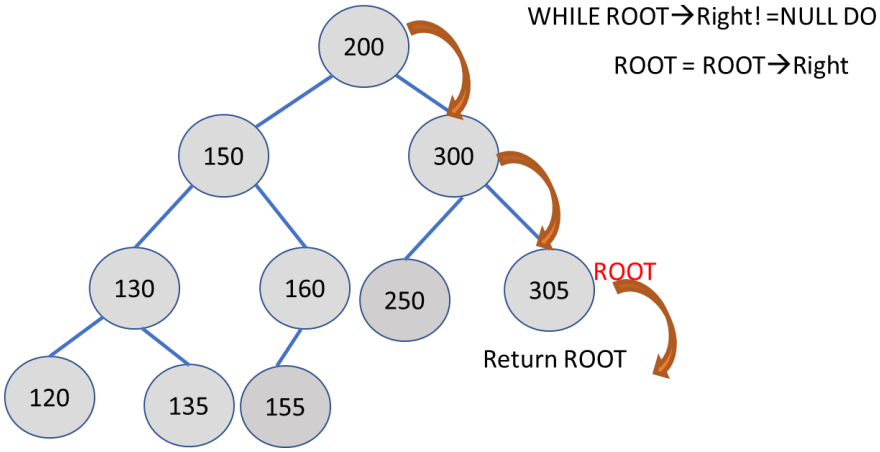


Fig-24

Using recursive method

ALGORITHM Maximum_BSTR(ROOT)

BEGIN:

IF ROOT→Right ==NULL THEN

Return ROOT

Maximun_BSTR(ROOT→Right)

END;

Complexity of operation:

Time Complexity: In general, the time complexity to find the largest value in BST is O(h), where h is the height of the BST.

Space Complexity = O(h), where h is the height of the BST.

10.13.3.5 Inorder Successor of a given node

When we do the inorder traversal of the BST then the node Q just lies behind the given node P then Q is called the inorder Predecessor of node P.

When we do the inorder traversal of the BST then the node Q just lies ahead of the given node P then Q is called the inorder Successor of node P.

If any node T has two children, then its inorder predecessor is the maximum value in its left subtree (i.e. Right most node in its left subtree) and its inorder successor the minimum value in its right subtree (i.e. Left most node in its right subtree).

If it does not have a left child, then a node's inorder Predecessor is its left ancestors.

If it does not have the right child, then a node's inorder Successor is one of its right ancestors.

There will be no predecessor of the first node in the inorder traversal and no successor of the last node in the inorder traversal.

ALGORITHM Inorder_SuccessorBST(P)

BEGIN:

```
IF P→Right! = NULL THEN
        Q = Minimum_BSTI(P→Right)
        RETURN Q
ELSE
        Q = P→Father
        WHILE Q! = NULL && Q→Right == P DO
                P = Q
                Q = Q→Father
        RETURN Q
```

END;

Complexity of operation:

Time Complexity: In general, the time complexity to find an inorder successor in BST is O(h), where h is the height of the BST.

Space Complexity = O(1)

a) If node P has a non-empty Right child

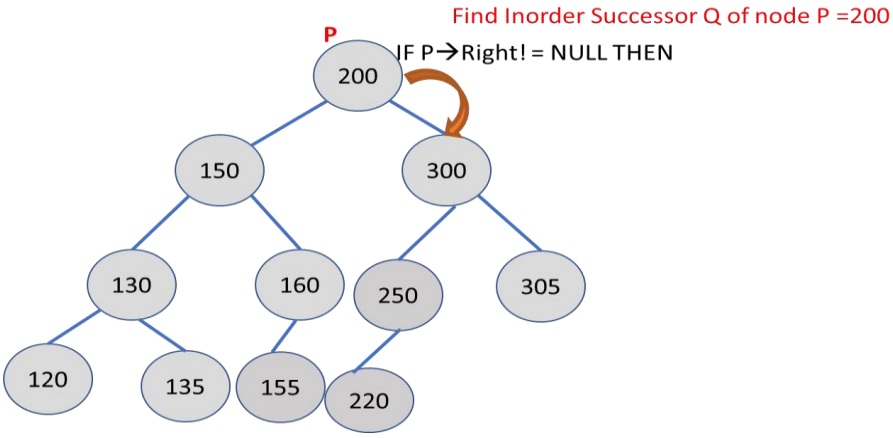


Fig-25

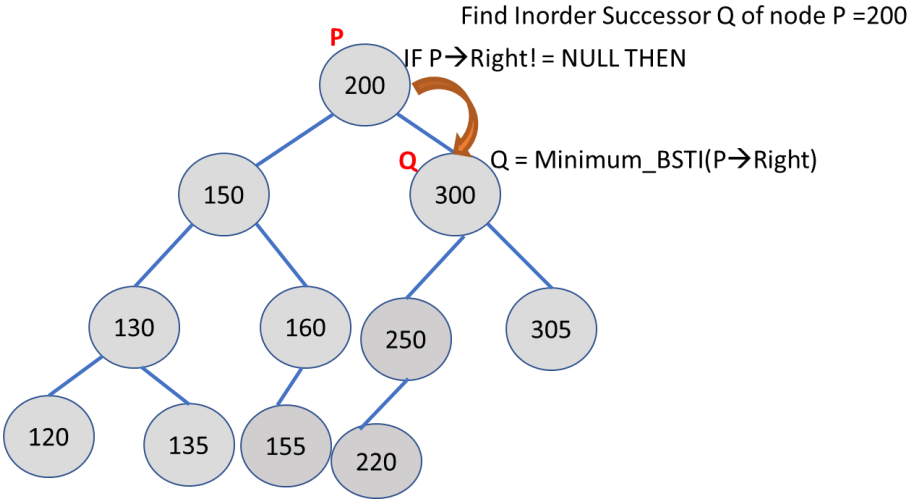


Fig-26

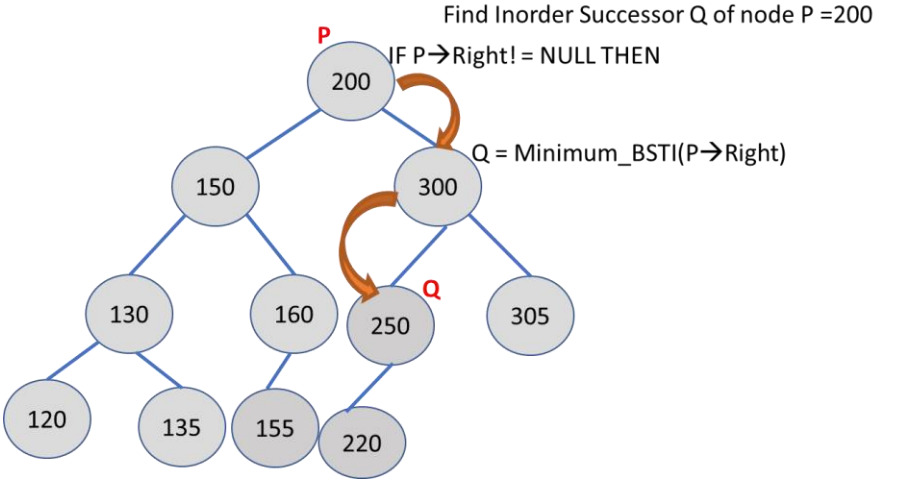


Fig-27

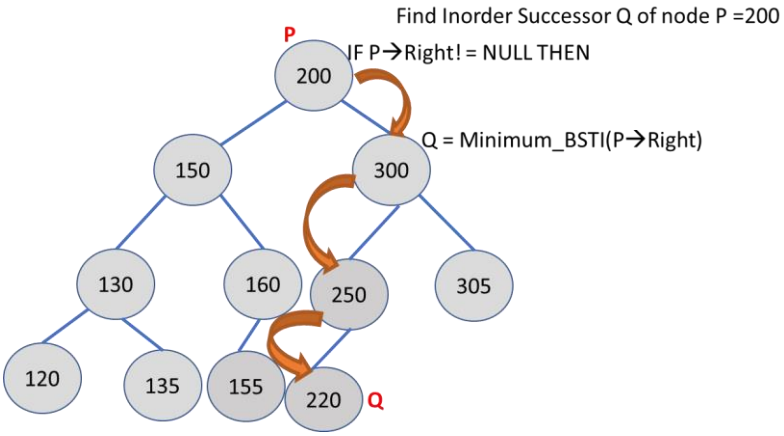


Fig-28

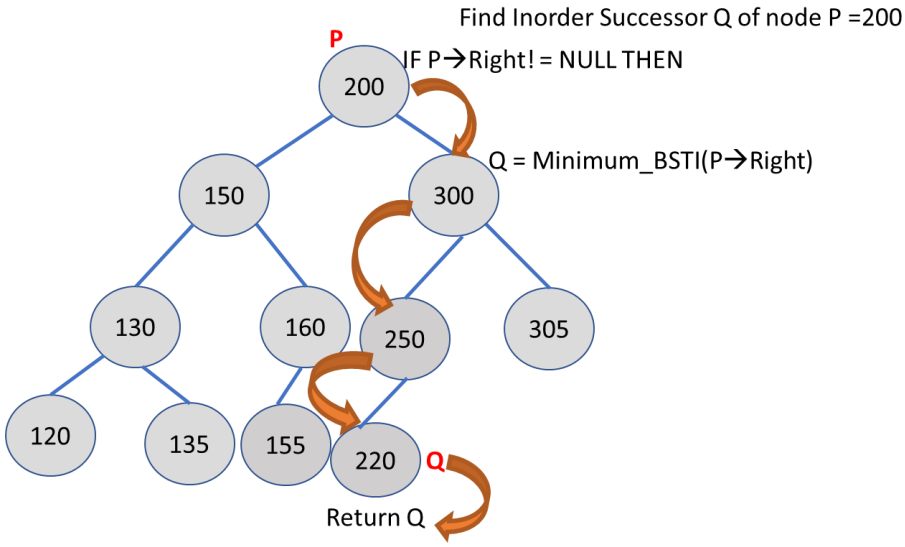


Fig-29

b) If node P has an empty Right child

Find Inorder Successor Q of node P =160

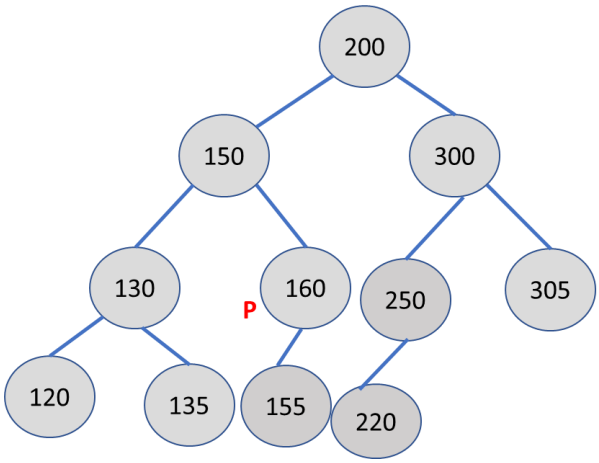


Fig-30

Find Inorder Successor Q of node P =160

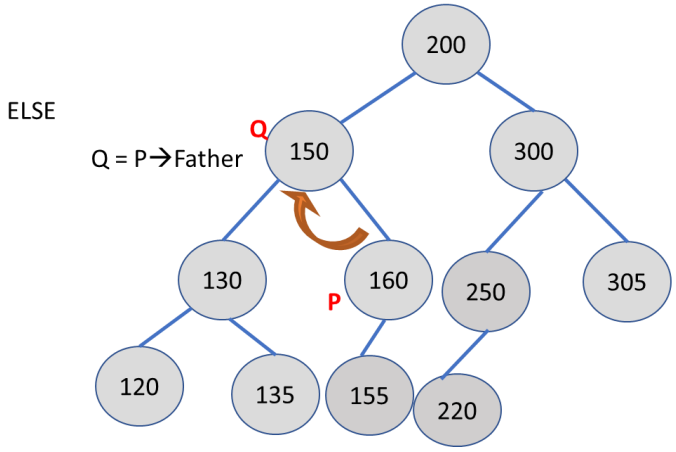


Fig-31

Find Inorder Successor Q of node P =160

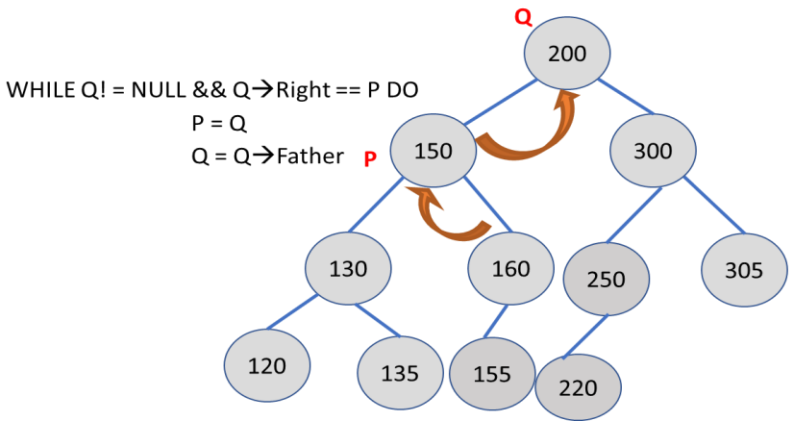
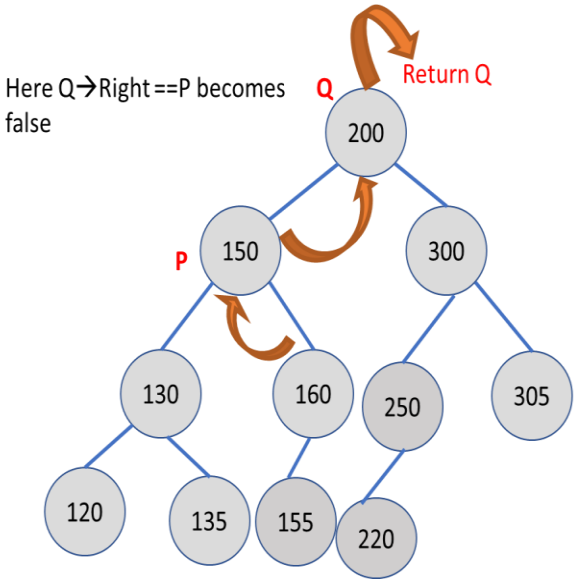


Fig-32



Find Inorder Successor Q of node P =160

Fig-33

Find Inorder Successor Q of node P =305

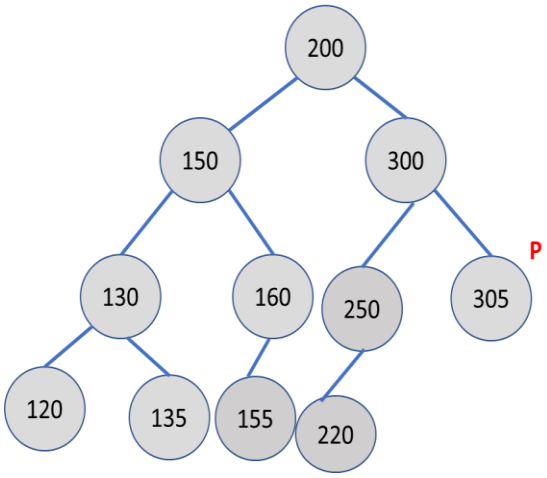


Fig-34

Find Inorder Successor Q of node P =305

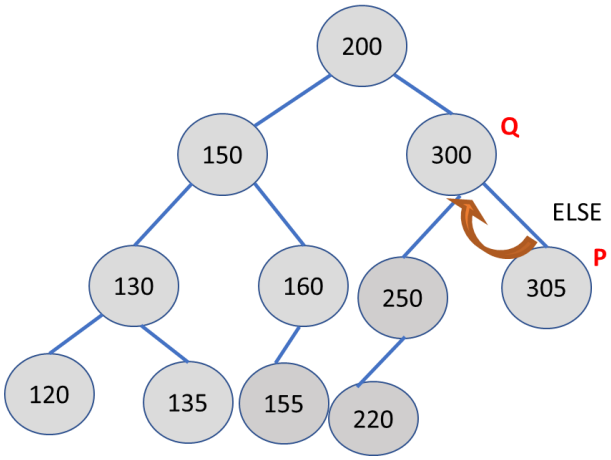


Fig-35

Find Inorder Successor Q of node P =305

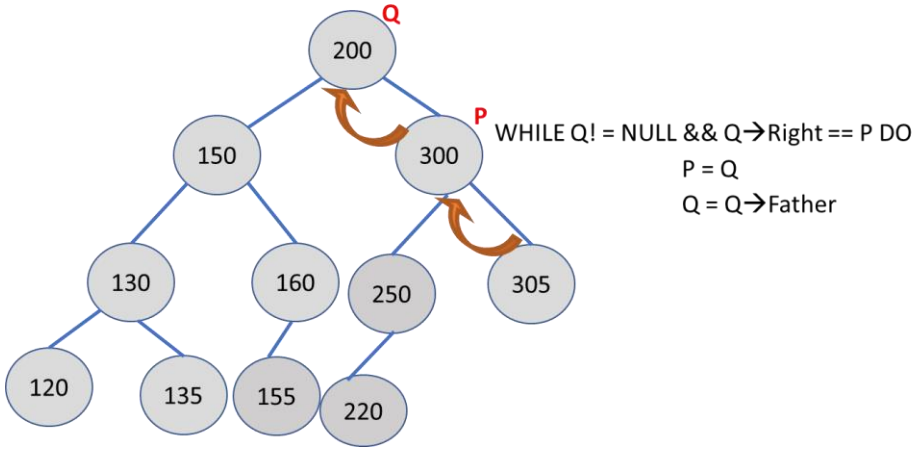


Fig-36

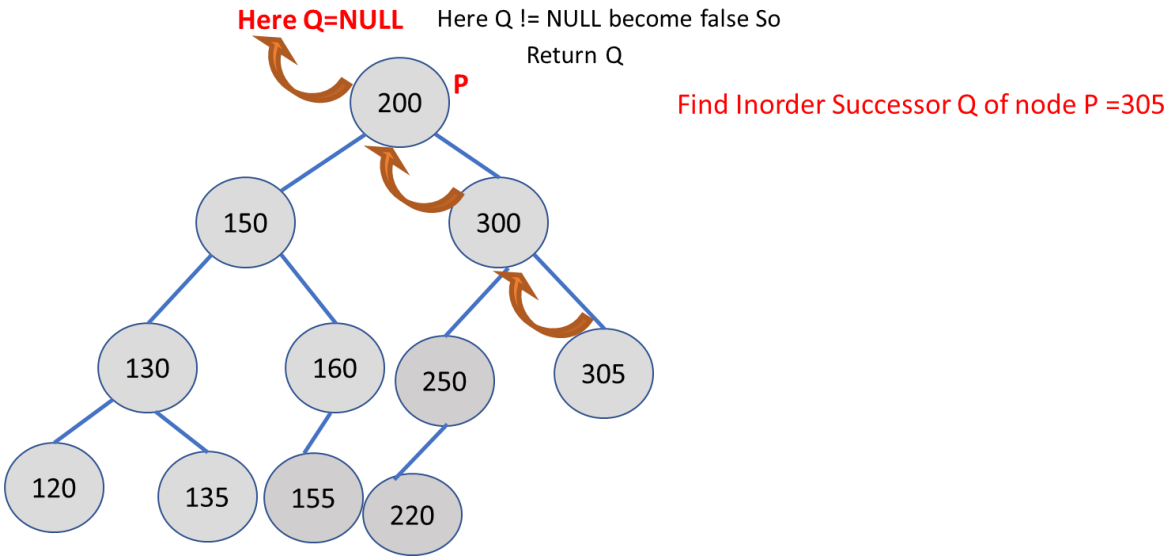


Fig-37

10.13.3.6 Find Inorder Predecessor of given node

ALGORITHMInorder_PredecessorBST(P)
BEGIN:
IF P→Left! = NULL THEN
Q = Maximum_BSTI(P→Left)
RETURN Q
ELSE
Q = P→Father
WHILE Q! = NULL && Q→Left == P DO
P = Q
Q = Q→Father
RETURN Q
END;

Complexity of operation:
Time Complexity: In general, the time complexity to find an inorder predecessor in BST is O(h), where h is the height of the BST.
Space Complexity = O(1)

10.13.3.7 Deletion in BST

This method will delete the given node in BST. There will be three cases to delete the given node:
Case 1: If the node is deleted with zero child i.e., leaf node then return NULL to its parent. This means, make the corresponding child pointer NULL.

Case 2: If the node to be deleted has one child, then simply link the current node's child to its parent.

Case 3: If the node to be deleted has two children, then

a) first find either its inorder successor or inorder predecessor.

b) Then replace the value of the node to be deleted by its inorder successor or inorder predecessor.

c) Simply delete that inorder successor or inorder predecessor using either case 1 or case 2 because inorder successor or inorder predecessor node always has either one child or no child.

```
ALGORITHM BST_Delete(P)
BEGIN:
//Case 1 When Node to be deleted has no child.
IF P→Left== NULL && P→Right == NULL THEN
    Q = P→Father
    IF Q == NULL THEN                    //node to be deleted is root node
        ROOT = NULL
    ELSE
        IF IsLeft(P)==True THEN          //node to be deleted is left or right child of
            Q→Left == NULL                its parent
        ELSE
            Q→Right == NULL
    FreeNode(P)
ELSE                                     //Case 2 When Node to be deleted has one child.
    IF P→Left== NULL || P→Right == NULL THEN
        IF P→Left == NULL THEN
            Child = P→Right
        ELSE
            Child = P→Left
        Q = P→Father
        IF Q == NULL THEN                //node to be deleted is root node
        ROOT = Child
        ELSE
            IF IsLeft(P) == True THEN
                Q→Left = Child
            ELSE
```

```
                    Q→Right = Child
            Child→Father = Q
            FreeNode(P)
    ELSE                    //Case 3 When Node to be deleted has two children.
            SUCCN = Inorder_SuccessorBST(P)
            P→Info = SUCCN→Info //Replace the value of the node to be deleted with
                                value of its successor node
            BST_Delete(SUCCN)     // Delete the successor node
END;
```

Complexity of operation:

Time Complexity: In general, the time complexity to delete operation in BST is O(h), where h is the height of the BST.

Space Complexity = O(h) for recursive one

= O(1) for iterative one

a) Case-1 Delete the node having no child.

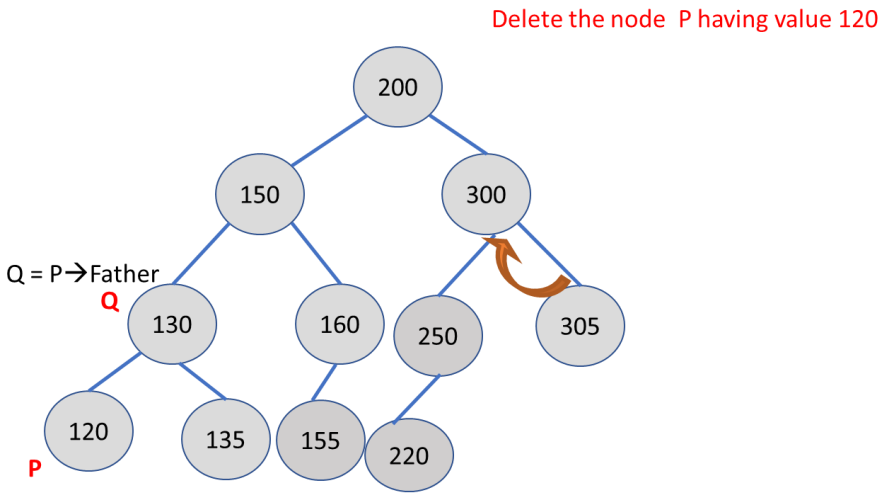


Fig-38

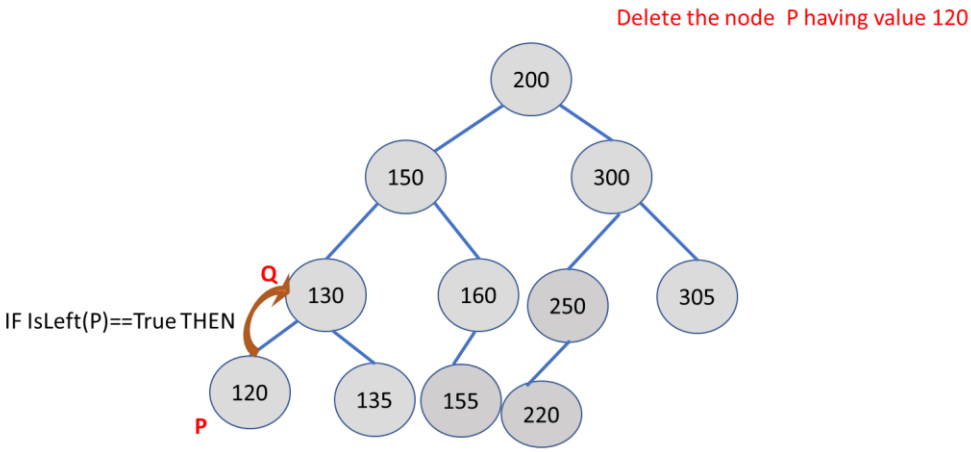


Fig-39

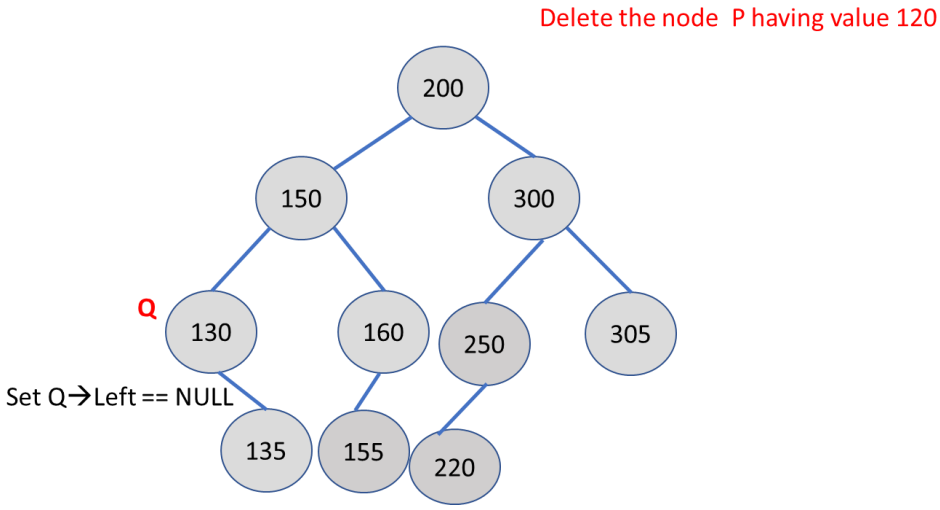


Fig-40

Case 2: Delete the node which has only one child.

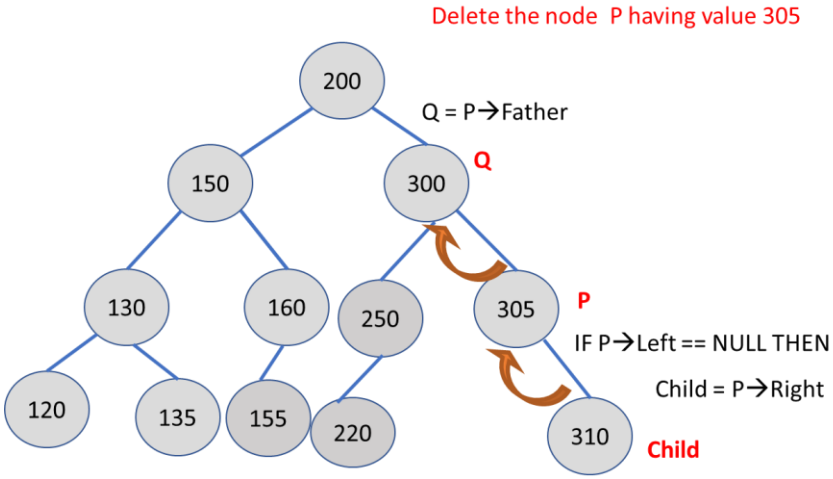


Fig-41

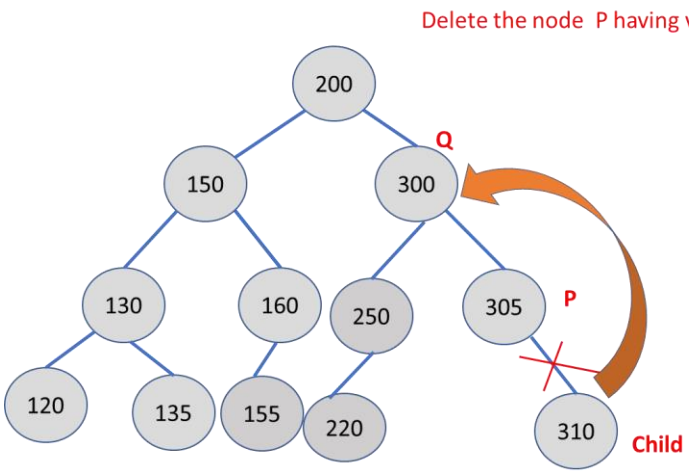


Fig-42

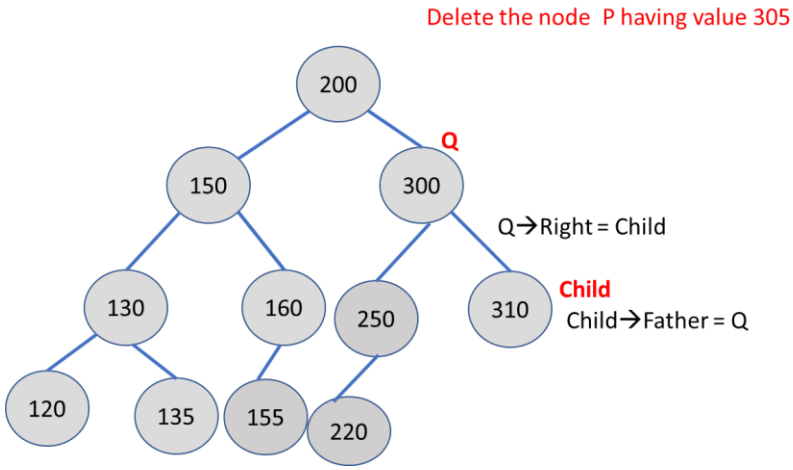


Fig-43

Case 3: Delete the node which has two children

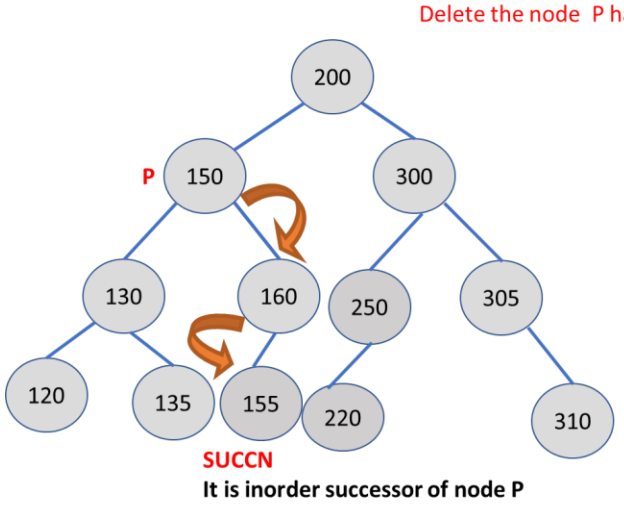


Fig-44

Delete the node P having value 150

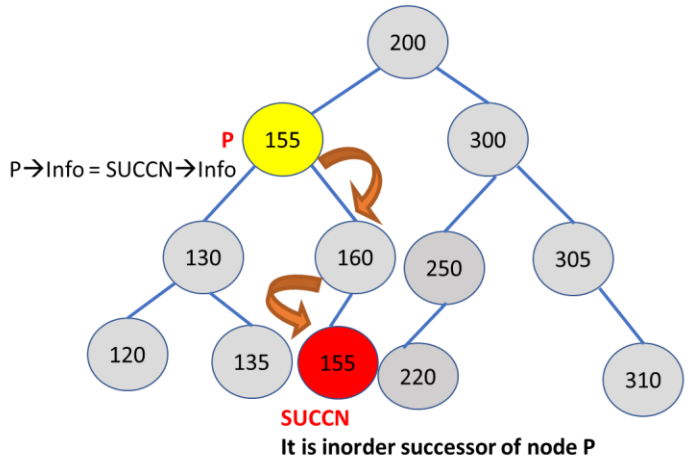


Fig-45

Delete the node P having value 150

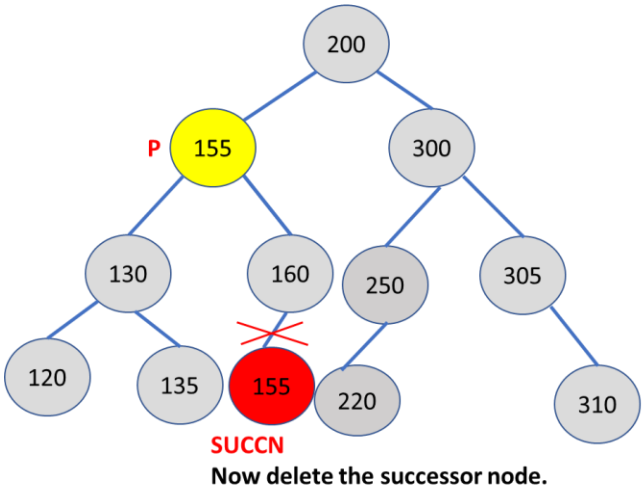


Fig-46

Delete the node P having value 150

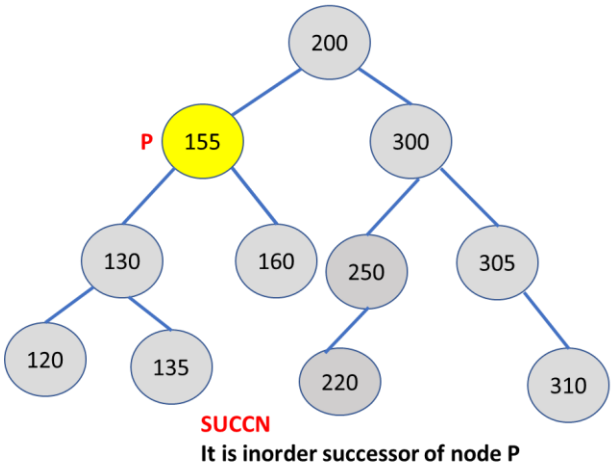


Fig-47

3.11 Gate Questions

Q-1) Let *T* be a binary search tree with 15 nodes. The minimum and maximum possible heights of *T* are: **GATE2017**

Note: The height of a tree with a single node is 0

(A) 4 and 15, respectively
(B) 3 and 14, respectively
(C) 4 and 14, respectively
(D) 3 and 15, respectively

Q-2) The pre-order traversal of a binary search tree is given by 12, 8, 6, 2, 7, 9, 10, 16, 15, 19, 17, 20. Then the post-order traversal of this tree is: **GATE2017**

(A) 2, 6, 7, 8, 9, 10, 12, 15, 16, 17, 19, 20
(B) 2, 7, 6, 10, 9, 8, 15, 17, 20, 19, 16, 12
(C) 7, 2, 6, 8, 9, 10, 20, 17, 19, 15, 16, 12
(D) 7, 6, 2, 10, 9, 8, 15, 16, 17, 20, 19, 12

Q-3) The number of ways in which the numbers 1, 2, 3, 4, 5, 6, 7 can be inserted in an empty binary search tree, such that the resulting tree has height 6, is _____. **GATE2016**

Note: The height of a tree with a single node is 0.

A) 64
B) 32
C) 128
D) 7

Q-4) The height of a tree is the length of the longest root-to-leaf path in it. The maximum and minimum number of nodes in a binary tree of height 5 is **GATE2015**

(A) 63 and 6, respectively
(B) 64 and 5, respectively
(C) 32 and 6, respectively
(D) 31 and 5, respectively

Q-5) Which of the following is/are correct inorder traversal sequence(s) of binary search tree(s)? **GATE2015**

I. 3, 5, 7, 8, 15, 19, 25
II. 5, 8, 9, 12, 10, 15, 25
III. 2, 7, 10, 8, 14, 16, 20
IV. 4, 6, 7, 9, 18, 20, 25

(A) I and IV only
(B) II and III only
(C) II and IV only
(D) II only

Q-6) What are the worst-case complexities of insertion and deletion of a key in a binary search tree? **GATE2015**

(A) θ(logn) for both insertion and deletion
(B) θ(n) for both insertion and deletion
(C) θ(n) for insertion and θ(logn) for deletion
(D) θ(logn) for insertion and θ(n) for deletion

Q-7) While inserting the elements 71, 65, 84, 69, 67, 83 in an empty binary search tree (BST) in the sequence shown, the element in the lowest level is **GATE2015**

(A) 65
(B) 67
(C) 69
(D) 83

Q-8) The preorder traversal sequence of a binary search tree is 30, 20, 10, 15, 25, 23, 39, 35, 42. Which one of the following is the postorder traversal sequence of the same tree?

GATE 2013

(A) 10, 20, 15, 23, 25, 35, 42, 39, 30
(B) 15, 10, 25, 23, 20, 42, 35, 39, 30
(C) 15, 20, 10, 23, 25, 42, 35, 39, 30
(D) 15, 10, 23, 25, 20, 35, 42, 39, 30

Q-9) We are given a set of n distinct elements and an unlabeled binary tree with n nodes. How many ways can we populate the tree with the given set to become a binary search tree?

GATE 2013

(A) 0
(B) 1
(C) n!
(D)$(1/n+1).^{2n}C_n$

3.12 Coding Questions on BST

1. Insertion in a BST.

2. Search a given key in BST.
3. Deletion from BST (Binary Search Tree).
4. Construct a balanced BST from the given keys.
5. Determine whether a given binary tree is a BST or not.
6. Check if the given keys represent the same BSTs or not without building BST.
7. Find inorder predecessor for the given key in a BST.
8. Find the Lowest Common Ancestor (LCA) of two nodes in a BST.
9. Find k^{th} smallest and k^{th} largest element in a BST.
10. Build a Binary Search Tree from a postorder sequence.
11. Build a Binary Search Tree from a preorder sequence.
12. Print complete Binary Search Tree (BST) in increasing order.

10.14 AVL Tree

10.14.1 Introduction

IRCTC or any other Railway system taking the fact into account that newer trains come up very few every year and the code (struct train {}) of trains remains constant for a good period, an AVL implementation of this would be better than any other tree for searching.
AVL trees are beneficial when designing some database where insertions and deletions are not that frequent but frequently lookup for the items present.

10.14.2 Why AVL tree?

Consider the elements 6,4,3,2,1 to be inserted into a binary search tree turns out to be left-skewed as shown in Fig 1(a), and again the insertion of elements 1,2,3,4,6 in that order in the binary search tree results in a right-skewed binary search tree as shown in Fig1(b).

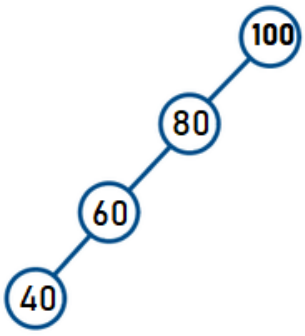
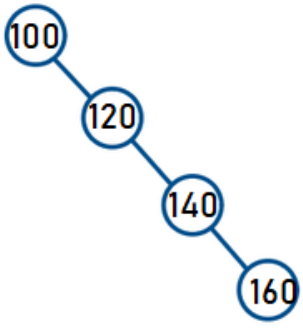


Fig1 (a) Left Skewed



(b) Right Skewed

AVL tree controls the height of the binary search tree by not letting it be skewed. The time taken for all operations in a binary search tree of height h is **O(h)**. However, it can be extended to **O(n)** if the BST becomes skewed (i.e., worst case). By limiting this height to log n, the AVL

tree imposes an upper bound on each operation to be **O(log n)**, where n is the number of nodes.

10.14.3 Definition

One of the balanced trees was introduced in 1962 by GM Adelson - Velsky and EM Landis in 1962 and is known as the AVL tree.

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.

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

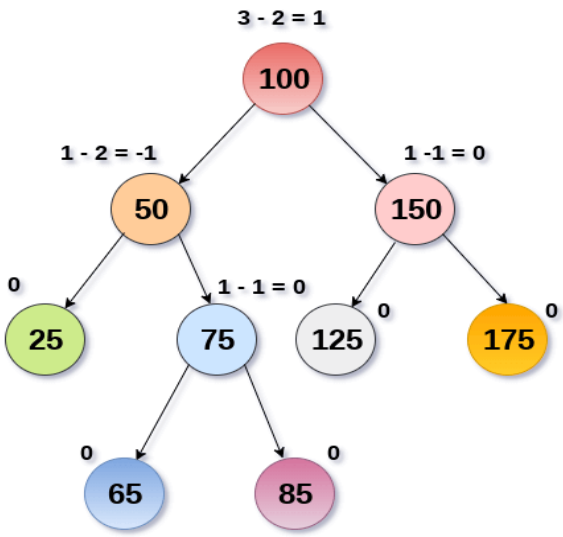
Balance Factor (k) = height (left(k)) - height (right(k))

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 the 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.

We can see that, balance factor associated with each node is in between -1 and +1. Therefore, it is an example of an AVL tree.



10.14.4 Operations on AVL Tree

Because, 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 of the property of the AVL tree. However, insertion and deletion are the operations that can violate this property, and therefore, they need to be revisited.

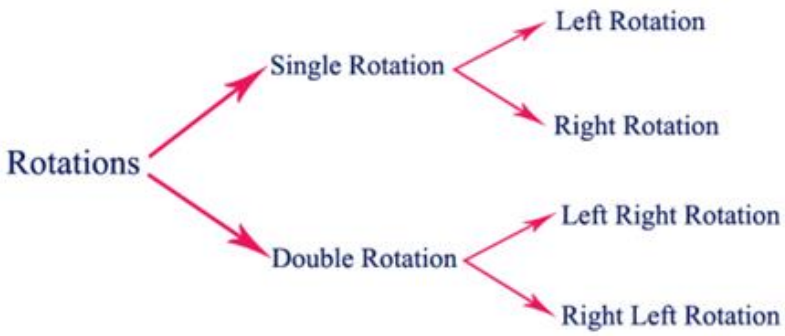
10.14.5 Insertion

Insertion in the AVL tree is performed similarly as it is performed in a binary search tree. However, it may lead to the violation in the AVL tree property, and therefore the tree may need balancing. The tree can be balanced by applying rotations.

10.14.6 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. We perform rotation in the AVL tree only if Balance Factor is other than **-1, 0, and 1**. There are **four** rotations, and they are classified into **two** types.

10.14.7 Rotations



These four types of rotations are as follows:

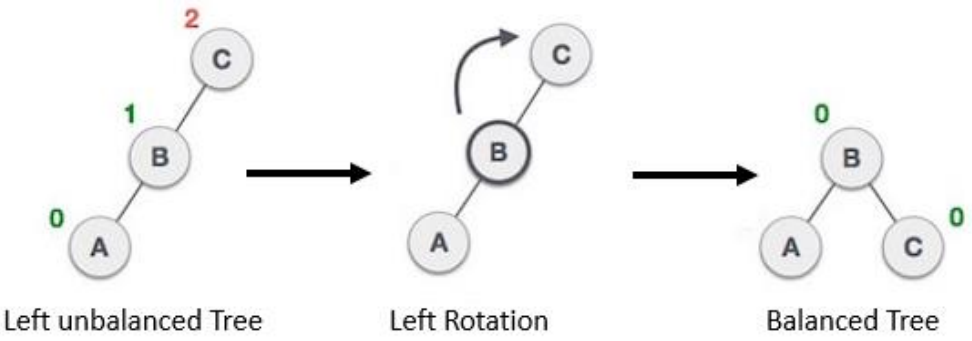
1. L L rotation: Inserted node is in the left subtree of left subtree of A
2. R R rotation: Inserted node is in the right subtree of right subtree of A
3. L R rotation: Inserted node is in the right subtree of left subtree of A
4. R L rotation: Inserted node is in the left subtree of right subtree of A

Where node A is the node whose balance Factor is other than -1, 0, 1.

The first two rotations LL and RR are single rotations and the next two rotations LR and RL are double rotations. For a tree to be unbalanced, minimum height must be at least 2, Let us understand each rotation

10.14.7.1 **LL Rotation**

When BST becomes unbalanced, a node is inserted into the left subtree of the left subtree of C; then we perform LL rotation.LL rotation is a clockwise rotation applied on edge below a node having balance factor 2.

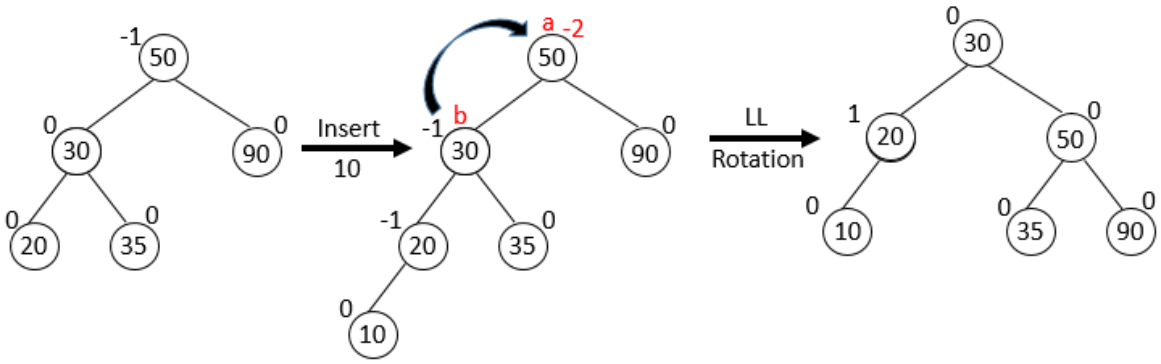


In the above example, node C has a balance factor 2 because node A is inserted in the left subtree of C left subtree. We perform the LL rotation on edge below A.

Logic :

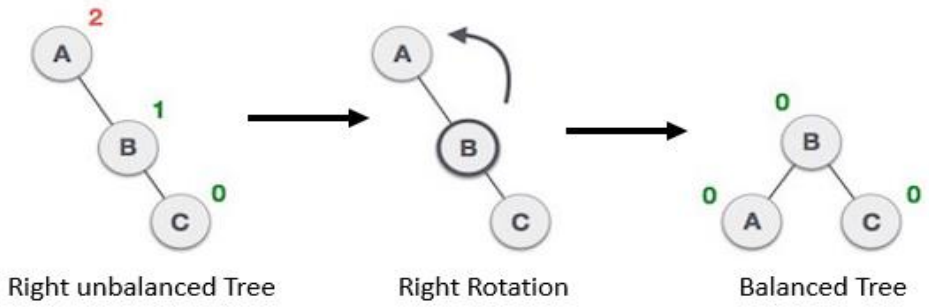
left (a) = right (b)

right (b) = a



10.14.7.2 RR **Rotation**

When BST becomes unbalanced, a node is inserted into the right subtree of the right subtree of A, then we perform RR rotation, RR rotation is an anticlockwise rotation, which is applied on edge below a node having balance factor –2.



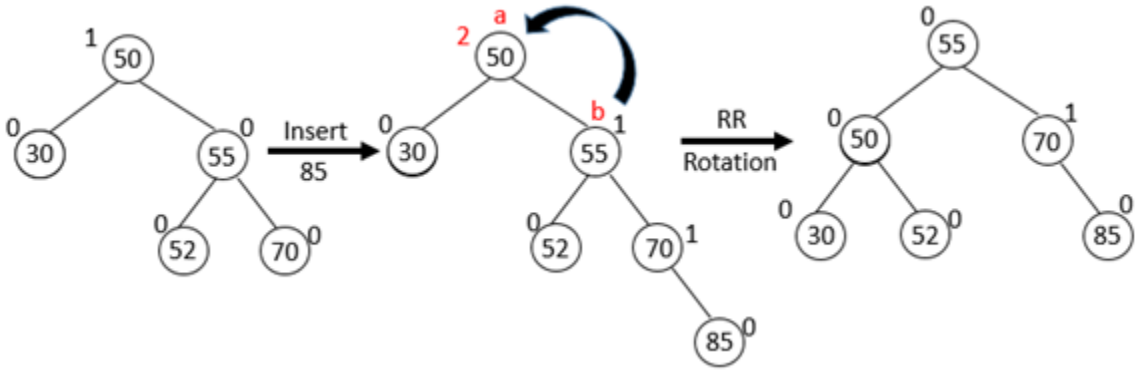
In the above example, node A has a balance factor -2 because node C is inserted in the right subtree of A right subtree. We perform the RR rotation on edge below A.

Logic:

right (a) = left (b)

Left (b) = a

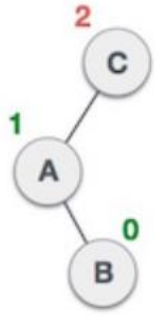
Example



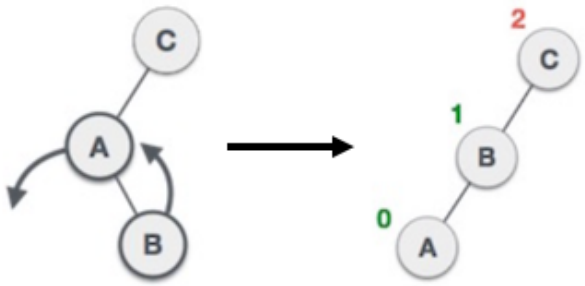
10.14.7.1 **LR Rotation**

Double rotations are a bit tougher than single rotation, which has already explained above. LR rotation = RR rotation + LL rotation, i.e., first RR rotation is performed on subtree, and then LL rotation is performed on the full tree; by full tree we mean the first node from the path of the inserted node whose balance factor is other than -1, 0, or 1.

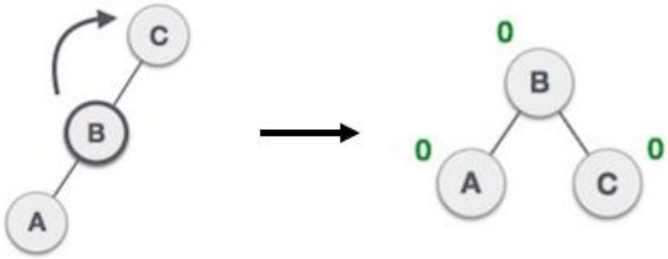
A node B has been inserted into the right subtree of A the left subtree of C, because of which C has become an unbalanced node having balance factor 2. This case is L R rotation where: Inserted node is in the right subtree of the left subtree of C.



Step1: As LR rotation = RR + LL rotation, RR (anticlockwise) on subtree rooted at A is performed first. By doing RR rotation, node **A**, has become the left subtree of **B**. After performing RR rotation, node C is still unbalanced, i.e., having balance factor 2, as inserted node A is in the left of left of **C.**

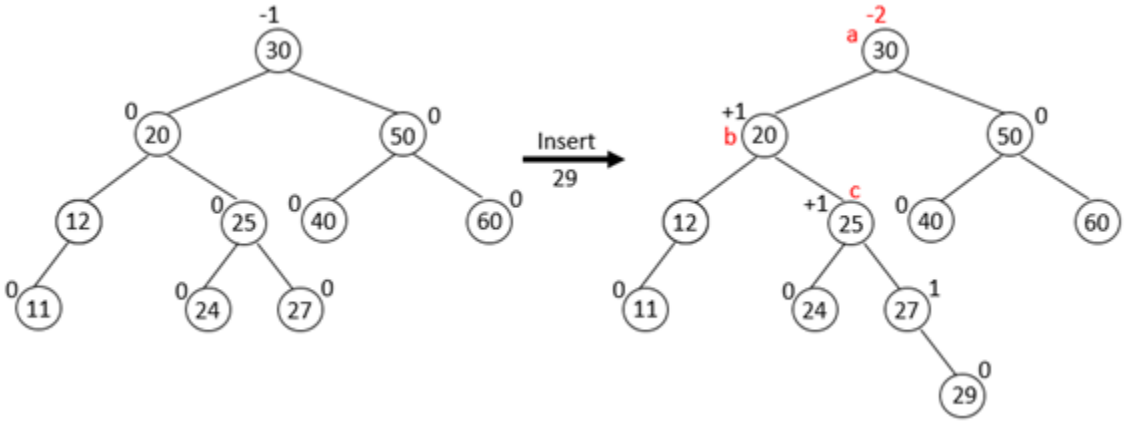


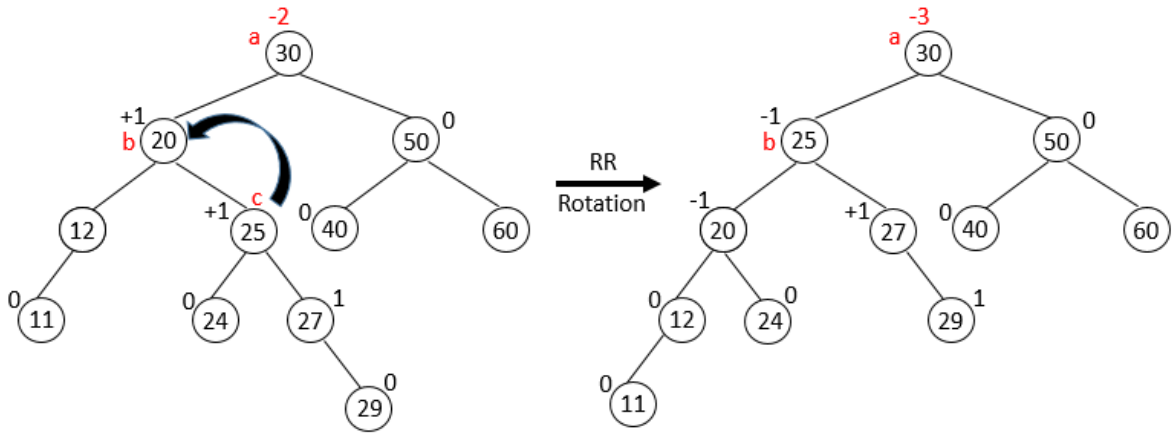
Step 2: Now, we perform LL clockwise rotation on full tree, i.e. on node C. node **C** has become the right subtree of node B, A is the left subtree of B.

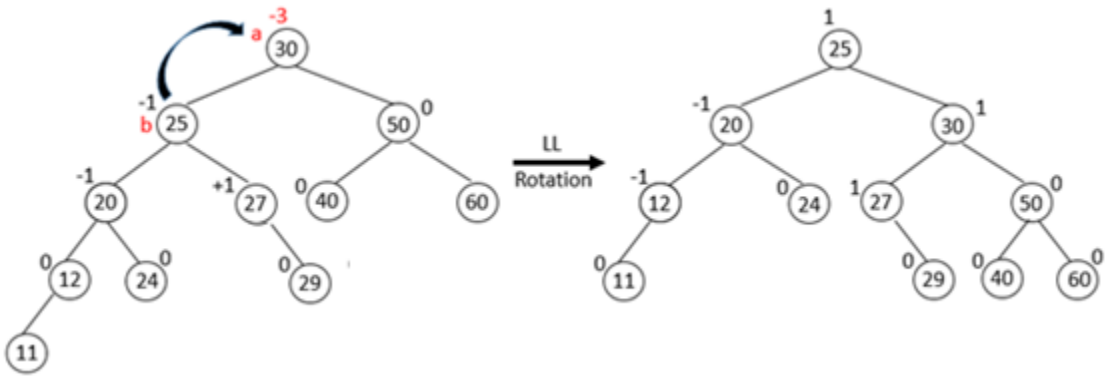


Tree is now balanced.

Example



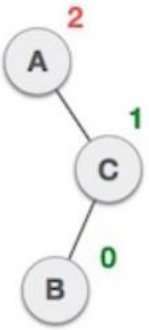




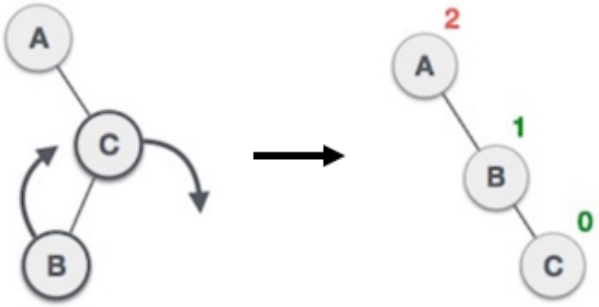
10.14.7.1 **RL Rotation**

As already discussed, that double rotations are bit tougher than single rotation, which has already been explained above. R L rotation = LL rotation + RR rotation, i.e., first LL rotation is performed on subtree and then RR rotation is performed on full tree.We mean the first node from the path of the inserted node whose balance factor is other than -1, 0, or 1.

A node **B** has been inserted into the left subtree of **C** the right subtree of **A**, because of which A has become an unbalanced node having balance factor - 2. This case is RL rotation where: Inserted node is in the left subtree of right subtree of A.



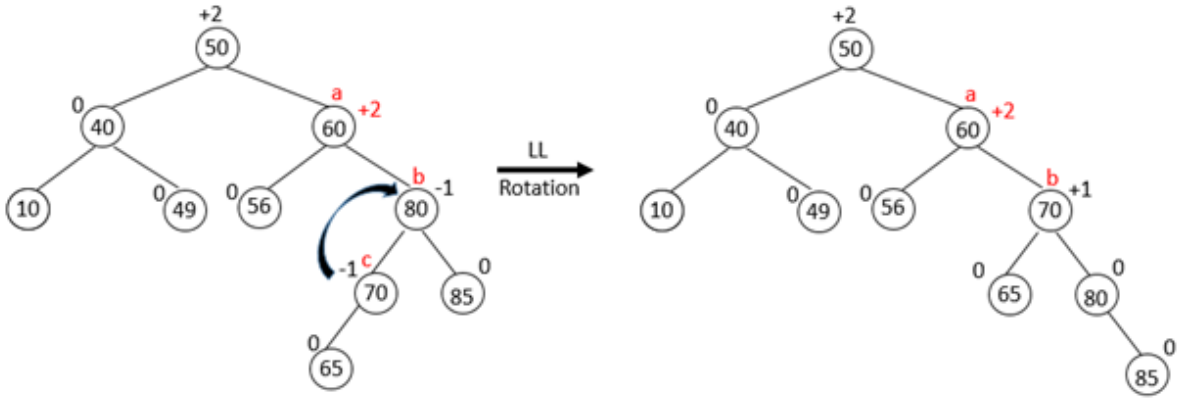
Step1: As RL rotation = LL rotation + RR rotation, hence, LL (clockwise) on subtree rooted at **C** is performed first. By doing RR rotation, node **C** has become the right subtree of **B**. After performing LL rotation, node **A** is still unbalanced, i.e. having balance factor -2, which is because of the right-subtree of the right-subtree node A.



Step2: Now, we perform RR rotation (anticlockwise rotation) on full tree, i.e. on node A. node **C** has now become the right subtree of node B, and node A has become the left subtree of B.

The tree is now balanced.

Example: RL Rotation



Time Complexity

Due to the balancing property, the insertion, deletion, and search operations take *O*(*logn*) in both the average and the worst cases. Therefore, AVL trees give us an edge over Binary Search Trees which have an *O*(*n*) time complexity in the worst-case scenario.

Space Complexity

The space complexity of an AVL tree is *O*(*n*) in both the average and the worst case.

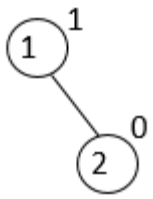
Example: Construct AVL Tree for 1,2, 3, 4,8, 7, 6, 5, 11, 10

Insert 1

Tree is balanced

Insert 2

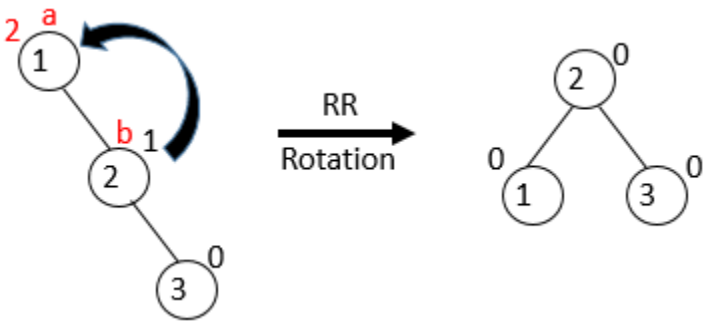
- As 2 > 1, so insert 2 in 1's right sub tree.



Tree is Balanced

Insert 3

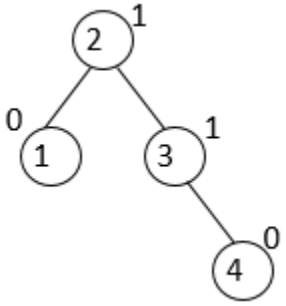
- As 3 > 1, so insert 3 in 1's right sub tree.
- As 3 > 2, so insert 3 in 2's right sub tree.



Tree is balanced after RR Rotation

Insert 4

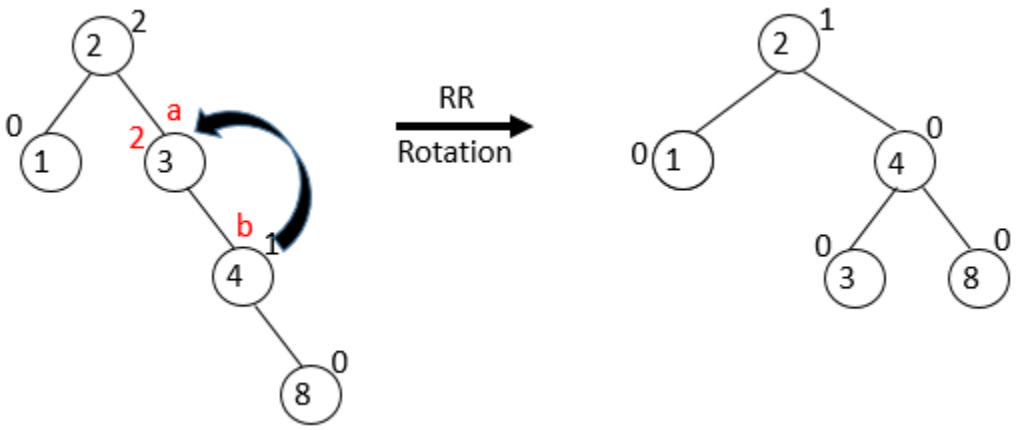
- As 4 > 2, so insert 4 in 2's right sub tree.
- As 4 > 3, so insert 4 in 3's right sub tree.



Tree is Balanced

Insert 8

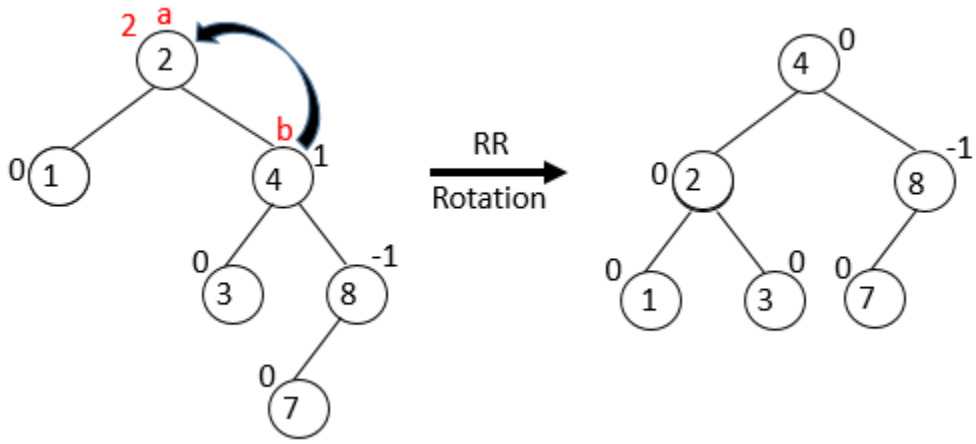
- As 8 > 2, so insert 8 in 2's right sub tree.
- As 8 > 3, so insert 8 in 3's right sub tree.
- As 8 > 4, so insert 8 in 4's right sub tree



Tree is Balanced after RR Rotation

Insert 7

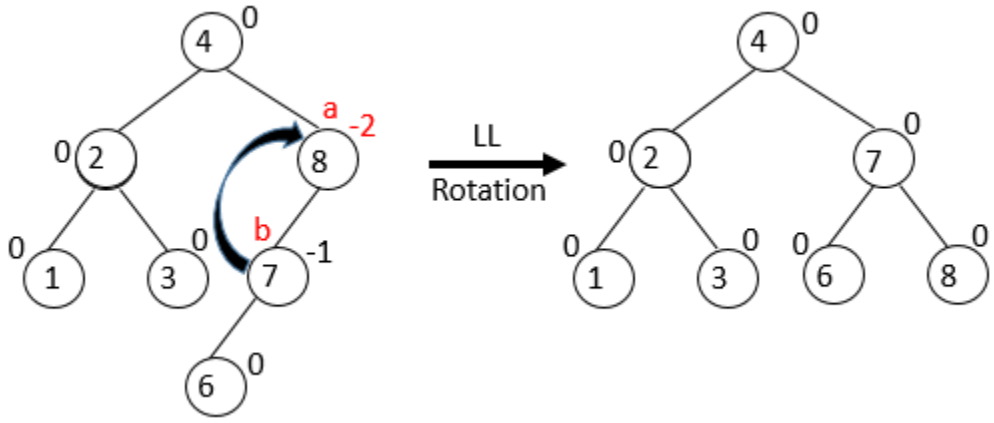
- As 7 > 2, so insert 7 in 2's right sub tree.
- As 7 > 4, so insert 7 in 4's right sub tree.
- As 7 < 8, so insert 7 in 8's left sub tree



Tree is Balanced after RR Rotation

Insert 6

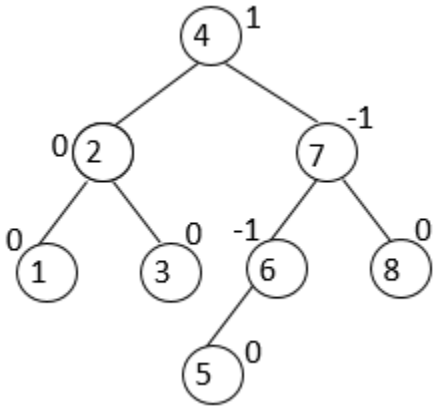
- As 6 > 4, so insert 6 in 4's right sub tree.
- As 6 < 8, so insert 6 in 8's left sub tree.
- As 6 < 7, so insert 6 in 7's left sub tree



Tree is Balanced after LL Rotation

Insert 5

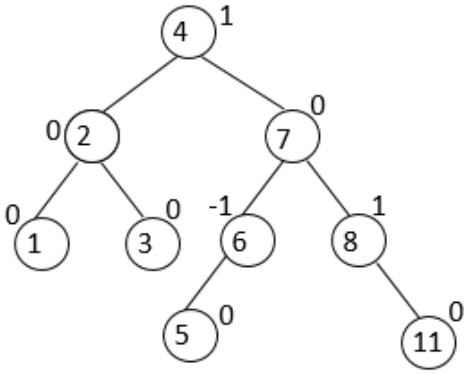
- As 5 > 4, so insert 5 in 4's right sub tree.
- As 5 < 7, so insert 5 in 7's left sub tree.
- As 5 < 6, so insert 5 in 6's left sub tree



Tree is Balanced

Insert 11

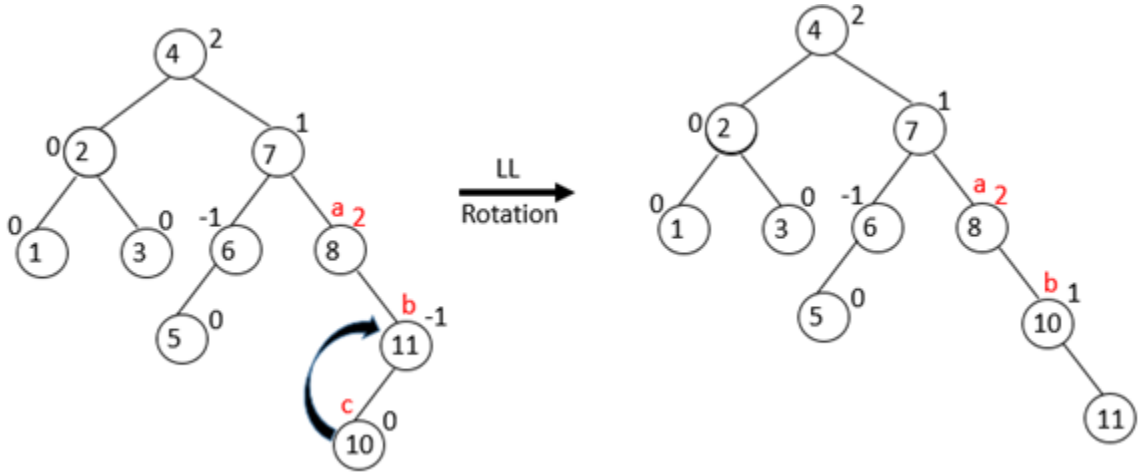
- As 11 > 4, so insert 11 in 4's right sub tree.
- As 11> 7, so insert 11 in 7's right sub tree.
- As 11> 8, so insert 11 in 8's right sub tree.

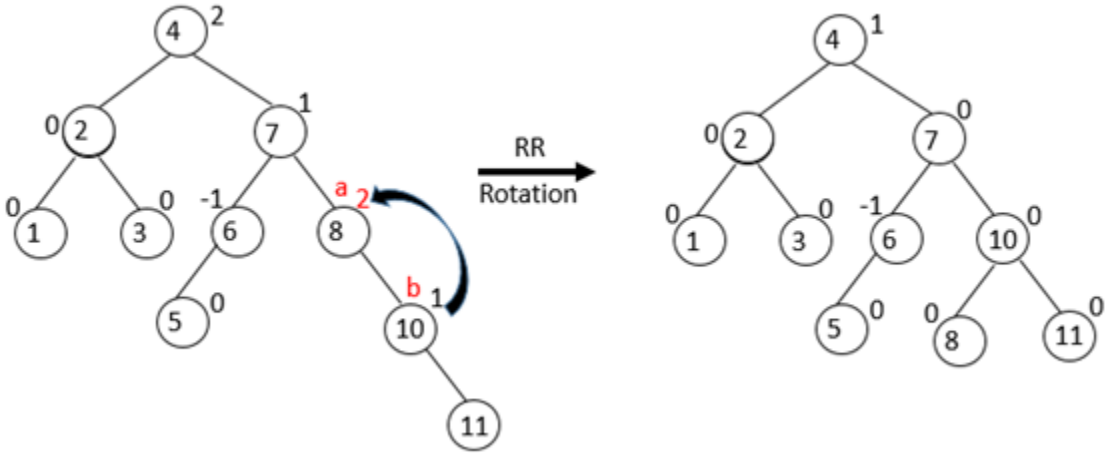


Tree is Balanced

Insert 10

- As 10 > 4, so insert 10 in 4's right sub tree.
- As 10 > 7, so insert 10 in 7's right sub tree.
- As 10 > 8, so insert 10 in 8's right sub tree
- As 10 < 11, so insert 10 in 11's left sub tree
-





Tree is Balanced after RL Rotation

10.15 AVL-Algorithms

10.15.1 Creation of Node

ALGORITHM GetNode()

BEGIN:

```
P→Left = NULL
P→Right = NULL
P→Father = NULL
RETURN P
```

END;

10.15.2 Finding Height

ALGORITHM Height (ROOT)

BEGIN:

```
IF ROOT == NULL THEN
        RETURN 0
ELSE
        IF ROOT →LEFT==NULL AND ROOT →RIGHT ==NULL THEN
        RETURN 0
        ELSE
                RETURN 1+ max(Height (ROOT →LEFT),Height(ROOT →RIGHT))
```

END;

10.15.3 Rotate Left

ALGORITHM RotateLeft (P)

BEGIN:

```
 Q=P→RIGHT
 R=Q→LEFT
Q→LEFT=P
P→RIGHT=R
RETURN Q
```

END;

10.15.4 Rotate Right

ALGORITHM RotateRIGHT(P)
BEGIN:
Q=P→LEFT
R=Q→RIGHT
Q→RIGHT=P
P→LEFT=R
RETURN Q
END;

10.15.5 Rotation LL

ALGORITHM LL(P)
BEGIN:
Q=RotateRIGHT(P)
RETURN Q
END;

10.15.6 Rotation RR

ALGORITHM RR(P)
BEGIN:
Q=RotateLEFT(P)
RETURN Q
END;

10.15.7 Rotation LR

ALGORITHM LR (P)
BEGIN:
Q=P→LEFT
R=RotateLEFT(Q)
P→LEFT=R
R=RotateRIGHT(P)
RETURN R
END;

10.15.7 Rotation RL

ALGORITHM RL(P)
BEGIN:

```
Q=P→RIGHT
R=RotateRIGHT(Q)
P->RIGHT=R
R=RotateLEFT(P)
RETURN R
```

END;

10.15.8 Balance Factor

4.5.9 ALGORITHM BalanceFactor(P)
BEGIN:

```
IF  P→LEFT==NULL THEN
        LH=0
ELSE
        LH=1+ Height (P→LEFT)
IF P→RIGHT==NULL THEN
        RH=0
ELSE
        RH=1+ Height (P→RIGHT)
RETURN (LH – RH)
```

END;

10.15.9 AVL Insertion

4.5.10 ALGORITHM AVLInsert (ROOT, ITEM)
BEGIN:

```
IF ROOT==NULL THEN
        P=GetNode()
        ROOT=P
ELSE
        IF(ITEM<ROOT→data) THEN
                ROOT→LEFT=AVLInsert(ROOT->LEFT,ITEM)
                IF BalanceFactor (ROOT)==2 THEN
```

```
                    IF(ITEM<ROOT→LEFT→data) THEN
                        ROOT=LL(ROOT)
                    ELSE
                        ROOT=LR(ROOT)
            ELSE
                ROOT→RIGHT=AVLInsert (ROOT→RIGHT,ITEM)
                IF  BalanceFactor(ROOT) == – 2 THEN
                    IF(ITEM>ROOT→RIGHT→data) THEN
                        ROOT=RR(ROOT)
                    ELSE
                        ROOT=RL(ROOT)
        RETURN ROOT
END;
```

10.16 Splay Tree

Splay tree is a self-adjusted (balanced) binary search tree in which the currently accessed (searched) item moves to the root so that if this item is accessed (searched) again, then it will take O(1) time.

1- A splay tree is best suited when few items are accessed very frequently from a huge number of nodes or keys.
2- The average time of all operations in splay tree is O(log n), but Worst case time complexity is O(n).

10.16.1 Searching in Splay Tree

Searching in a splay tree just like searching in binary search tree except for currently searched item move on top, i.e. splayed on top (root) if search operation successful otherwise last accessed node becomes new root, i.e. splayed on top of tree.

Cases for searching an item in Splay tree-

Case1- Node is root-

In this case, return the root either successful or not because in both cases root is splayed.

Case2a- Node is Child of root (no grandparent)-

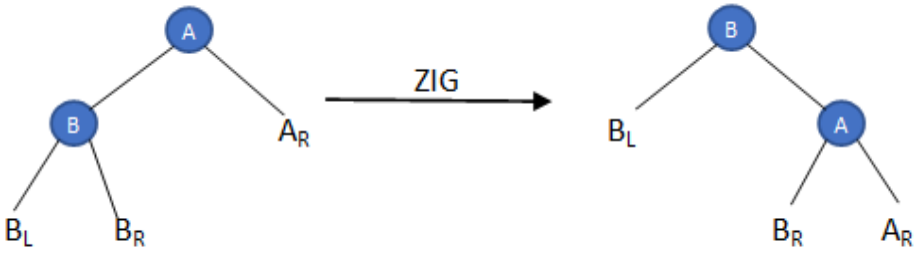
if node is left child of root, then right rotation (zig).

ALGORITHM rightRotate(ROOT)

BEGIN:

TEMP=ROOT→left
ROOT→left=TEMP→right
TEMP→right=ROOT
RETURN TEMP

END;



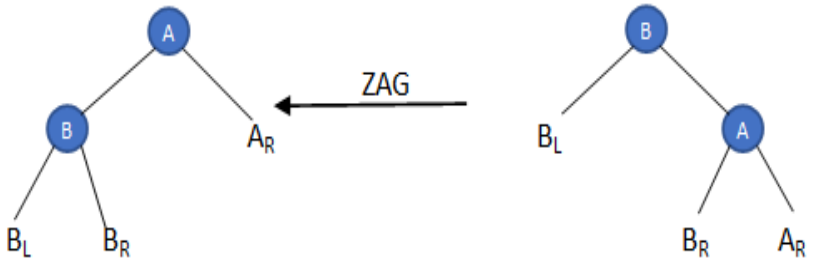
Case2b- Node is Child Of root (no grand parent)-if node is right child then left rotation (zag).

ALGORITHM leftRotate(ROOT)

BEGIN:

TEMP=ROOT→right
ROOT→right=TEMP→left
TEMP→left=ROOT
RETURN TEMP

END;



Case3 - Node having both parent and grandparent-

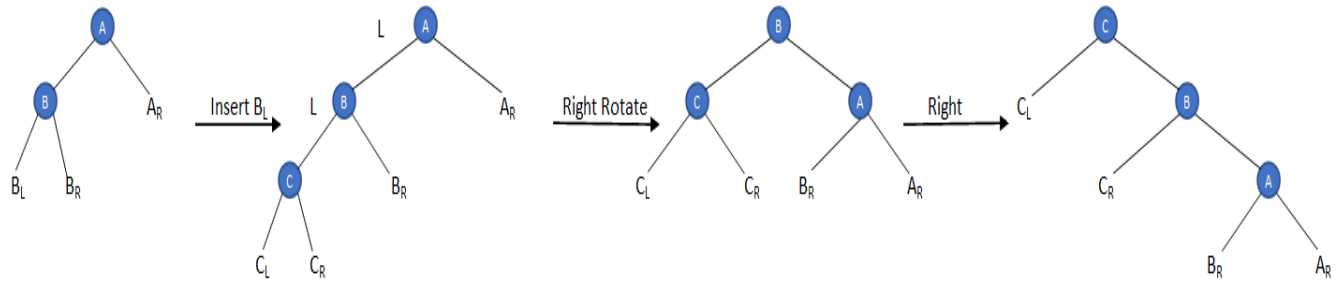
Case a-if Node is left child of parent and parent is also left child of grandparent (Two right rotations)

ALGORITHM leftLeftRotate(ROOT)

BEGIN:

TEMP=ROOT→left→left
ROOT→left→left=rightRotate(TEMP)
RETURN (rightRotate(TEMP))

END;



Case3 - Node having both parent and grandparent-

Case b-If node is right child of its parent and parent is also right child of the grandparent (Two Left Rotations or Zig-Zig).

ALGORITHM leftLeftRotate(ROOT)

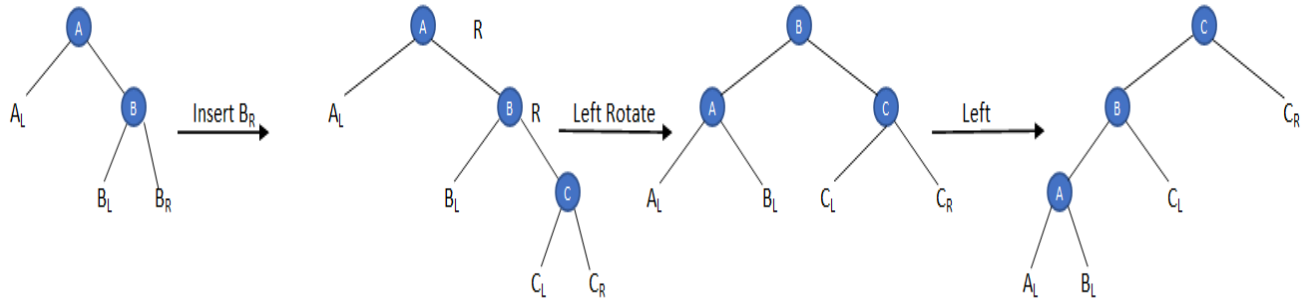
BEGIN:

TEMP=ROOT→right→right

ROOT→right→right=leftRotate(TEMP)

RETURN (leftRotate(TEMP))

END;



Case 4a-

If node is right child of its parent and parent is left child of the grandparent (Right Rotation after left rotation).

ALGORITHM leftRightRotate(ROOT)

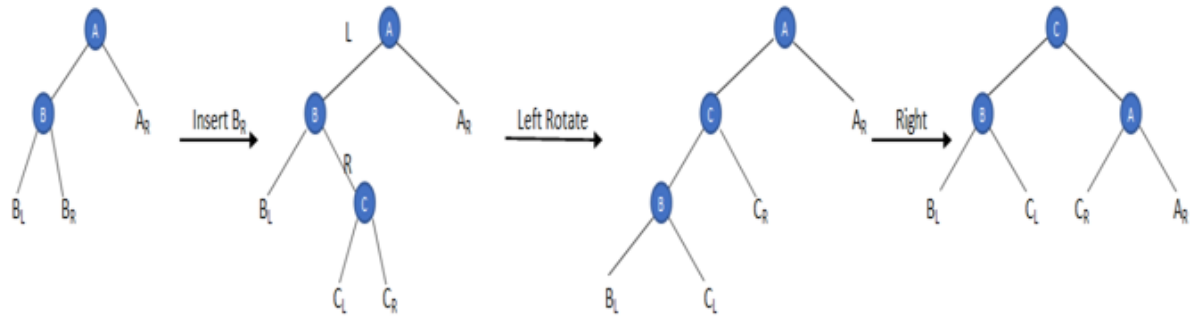
BEGIN:

TEMP=ROOT→left

ROOT→left=leftRotate(TEMP)

RETURN (rightRotate(TEMP))

END;



Case 4b-

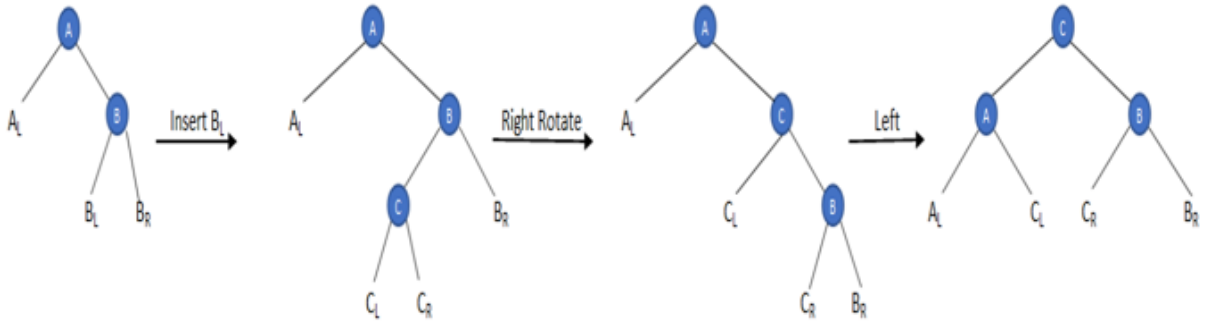
if Node is left child of parent and parent is right child of grandparent (Right Rotation followed by left rotation).

ALGORITHM leftRightRotate(ROOT)

BEGIN:

```
TEMP=ROOT→right
ROOT→right=rightRotate(TEMP)
RETURN (leftRotate(TEMP))
```

END;



5.2 Searching Algorithm-

ALGORITHM search(ROOT,x)

BEGIN:

```
IF ROOT==NULL OR ROOT→data==x THEN
        RETURN ROOT
IF ROOT→data > x
        IF ROOT→left==NULL THEN
                RETURN ROOT
        IF ROOT→left→data >x THEN
                ROOT→left→left=search(ROOT→left→left,x)
                ROOT=rightRotate(ROOT)
        ELSE IF ROOT→left->data < x THEN
                ROOT→left→right=search(ROOT→left→right,x)
```

```
                ROOT=rightRotate(ROOT)
                IF ROOT→left→right !=NULL THEN
                        ROOT→left=leftRotate(ROOT->left)
        RETURN (ROOT→left==NULL)?ROOT:rightRotate(ROOT)
END;
```

Searching Algorithm-

```
        IF ROOT→right==NULL THEN
                RETURN ROOT
        IF ROOT→right→data >x THEN
                ROOT→right→left=search(ROOT→right→left,x)
                IF ROOT→right→left !=NULL THEN
                        ROOT→right=rightRotate(ROOT→right)
        ELSE IF ROOT→right→data < x THEN
                ROOT→right→right=search(ROOT→right→right,x)
                ROOT=leftRotate(ROOT)
        RETURN (ROOT→right==NULL)?ROOT:leftRotate(ROOT)
END;
```

10.17 Interval TREE

Introduction-

In order to maintain a set of intervals so that all operations like insert remove and find overlapping intervals in O(log n) time called interval tree.

Every node must contain-

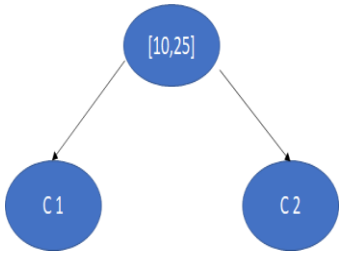
1) An interval (x , y) , x used as key value
2) Max value in subtree rooted with node

10.17.1 Creation/Insertion

Let us suppose given intervals are-

[10,25], [5,35], [15,22], [2,25],[7,10],[35,40]

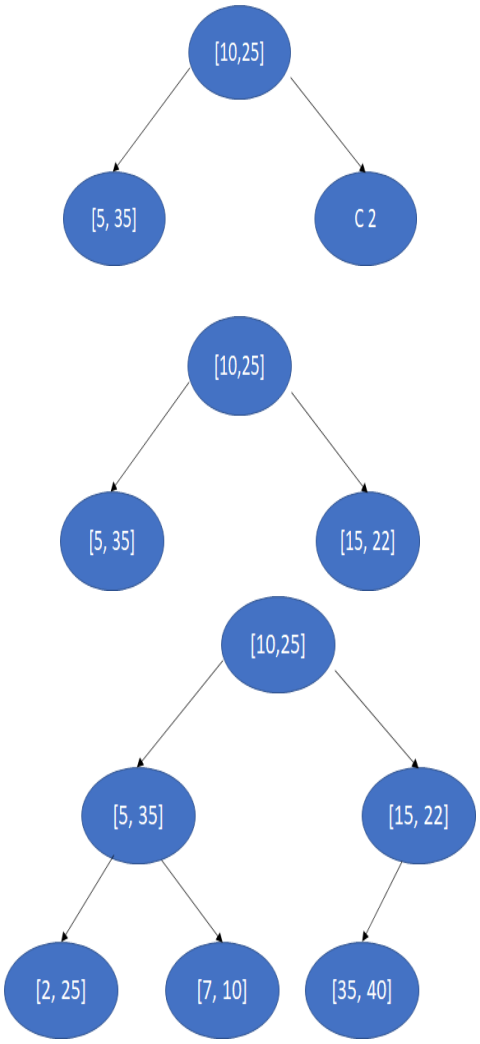
Step1- first node as a Root, so simply first interval set as a root.



Step2-

Now the first value of second given interval is 5, which is less than first value of the root, so this becomes the left child of root.

Repeat step-2 for the insertion of other given intervals.

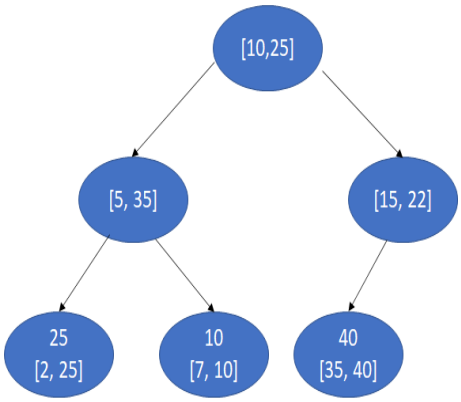


Setting the MAX value-

Step1- Start with leaf node-

The max value in leaf node [2,25] is 25 so max=25

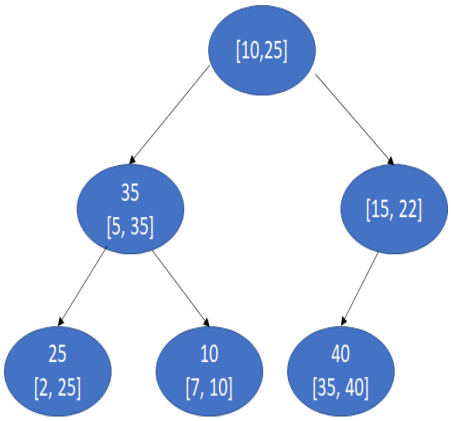
Repeat this step for other leaf nodes and set the max value of all leafs.



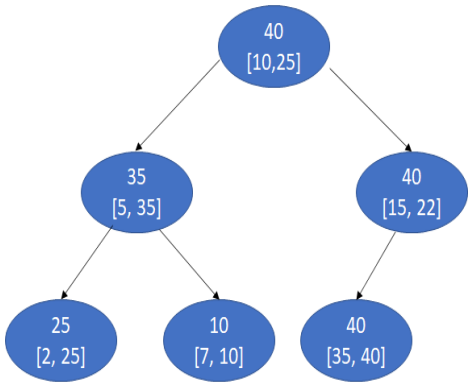
Step-2

Now consider the non-leaf node [5,35] .

In this node the max value is 35. Now, compare this max value with its children max value and set max value which one is greater.



Repeat step 2 for all non leaf nodes and set max value.



10.17.2 Searching an overlap interval

step1-In the first step check the overlap with root node if overlap, return the Boolean value true otherwise, move step2-f left child not equals to NULL, then compare the max value of left

child with min value of the given search interval. If greater, then repeat these steps for left child otherwise, for right child.

Complexity O(logn)

10.18 B-Tree

A B-Tree is a self-balancing tree that maintains sorted data and allows sequential access, insertion, deletion in logarithmic time.B-Tree is commonly used in database and file systems.

10.18.1 Need of Self Balancing Tree

BST is not a self-balancing tree, and if there exists a special case where nodes are only inserted on right side or at left side, then it is known as a skewed tree. The complexity in such a scenario for insertion, search, and deletionwill be O(n), where n is the total nodes in the tree. In order to reduce the complexity of the above operations, self-balancing trees are required—for example, AVL tree, B-Tree, Red Black Tree and many more. Here we are going to discuss B-Tree.

Skewed BST (Worst case complexity of operations) O(n)

B-Tree (Balanced m way tree)

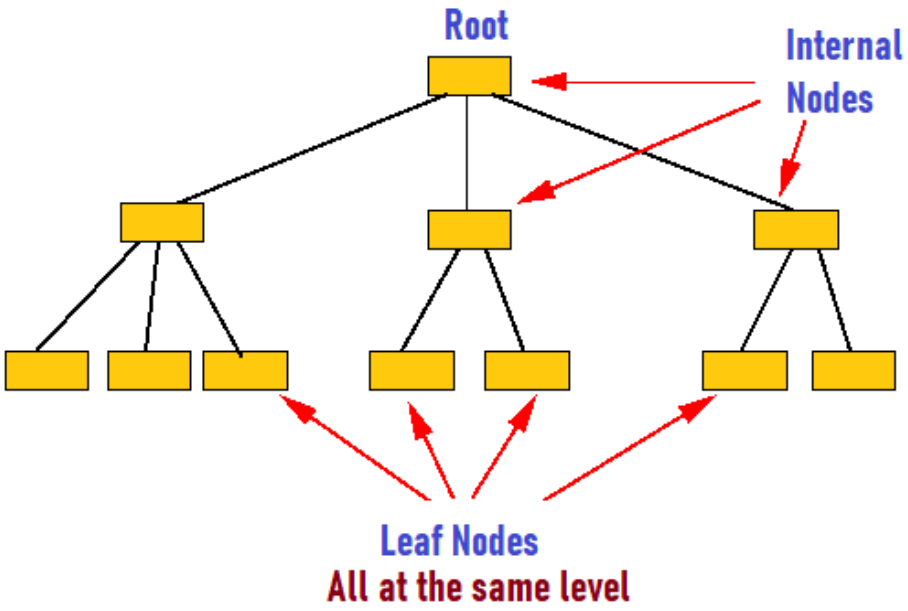
B-Tree is a generalization of BST. Like BST the data is sorted in B-Tree but in B-Tree we can have more than two children. Nodes in the B-Tree can have more than one key value (Restricted by the parameter degree or order).

10.18.2 Properties of B-Tree

A B-Tree is defined by its Order (The maximum children a node can have). Usually, the order is denoted by the term m.

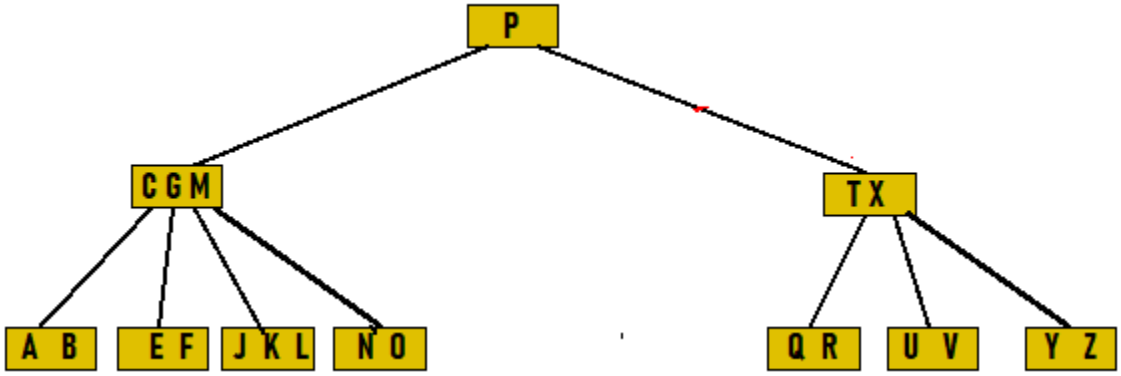
1. Each node (in a B-Tree of order m) has atmost m children.
2. Each node (in a B-Tree of order m) has at most m-1keys.
3. A node with k children has k-1 keys. (1<=k<=m)
4. All leaf nodes are at the same level

5. Every node (except root node)has a restriction of containing at least (m-1)/2 keys.
6. Root node can have number of keys less than (m-1)/2 but at least one key.
7. Root has at least has two children (if it is not leaf).
8. Keys in the nodes are arranged in non-descending order ($k_1 <= K_2 <= K_3 <= \ldots <= K_m$)



For a B-Tree for order- m

1) Minimum children :
   - Root : 2
   - Internal node :$\lceil m/2 \rceil$
2) Maximum children : m
3) Maximum keys : m-1
4) Minimum keys :
   - Root node : 1
   - All others :$\lceil m/2 \rceil - 1$



B-Tree can be useful when there would be large size of data that needs to be read or written. For example, when we are working on databases, then there, we can make use of B-Tree.

10.18.3 Basic Operations on B-Tree

10.18.3.1 Insertion

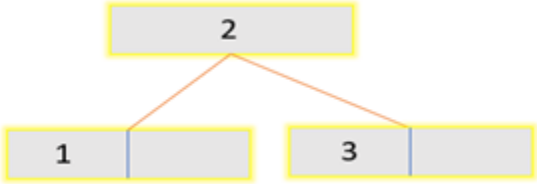
All insertions take place in the leaf nodes. The search is followed starting from root node that directs us to reach the target leaf node (same as in BST). If the target leaf node contains the maximum keys, the key is virtually inserted in that node, and the node splits in two parts while sending the median key up to the parent node. In this process, if the parent node also contains the maximum key, it also splits in the same manner and the median key is sent up. The process continues until we reach the parent that contains the keys less than maximum or to the root node. If the root also contains the maximum keys, it also splits into two parts and the median key is sent up. A new root node is created with this median key.

Example 1: Let us insert keys 1 to 10 in an empty B-Tree(Order 3).
Maximum children :3
Maximum keys :2

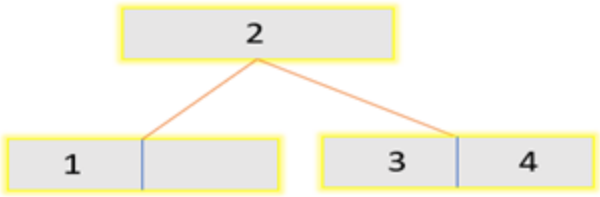
Insert 1

Insert 2

Insert 3
(Splitting after virtually inserting 3)

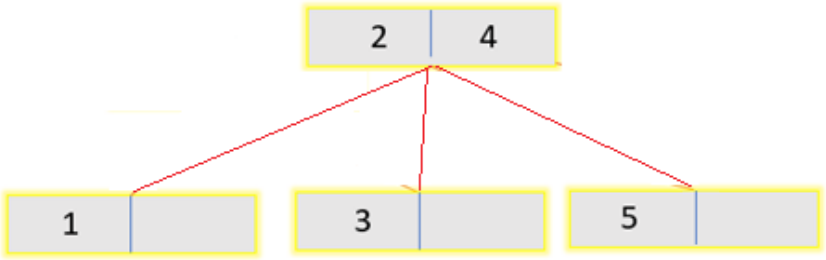


Insert 4



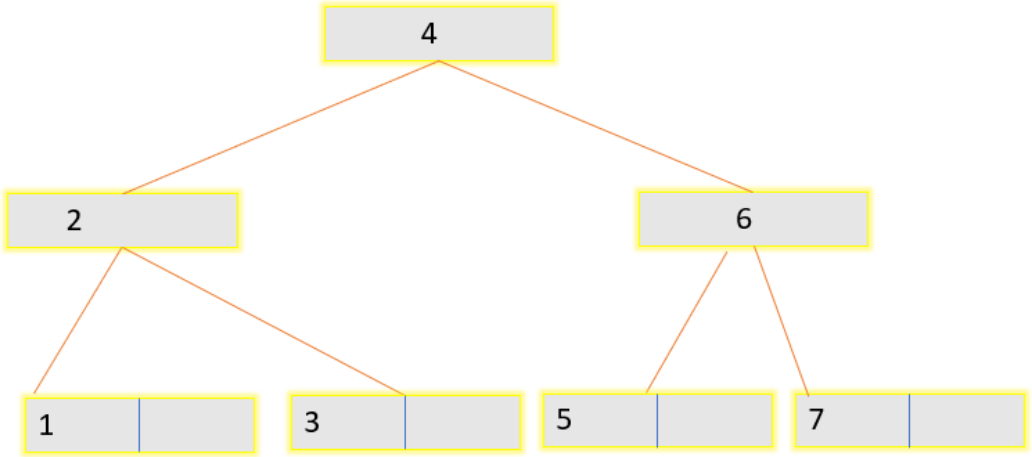
Insert 5

(Splitting after virtually inserting 5)

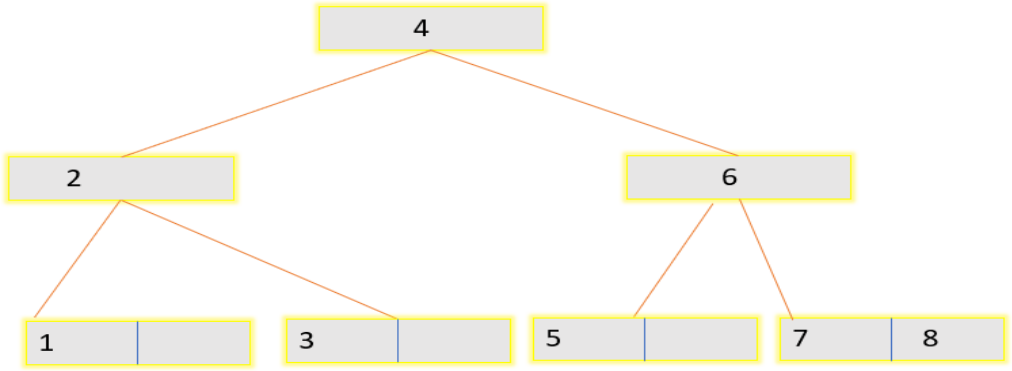


Insert 6,7

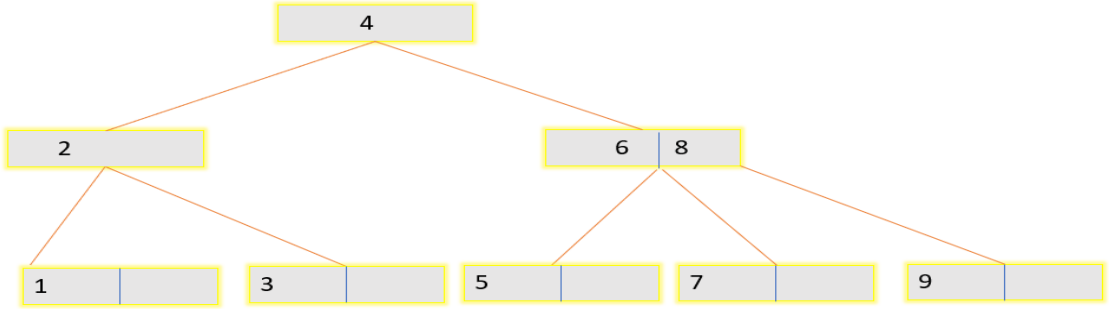
(Splitting after virtually inserting 7)



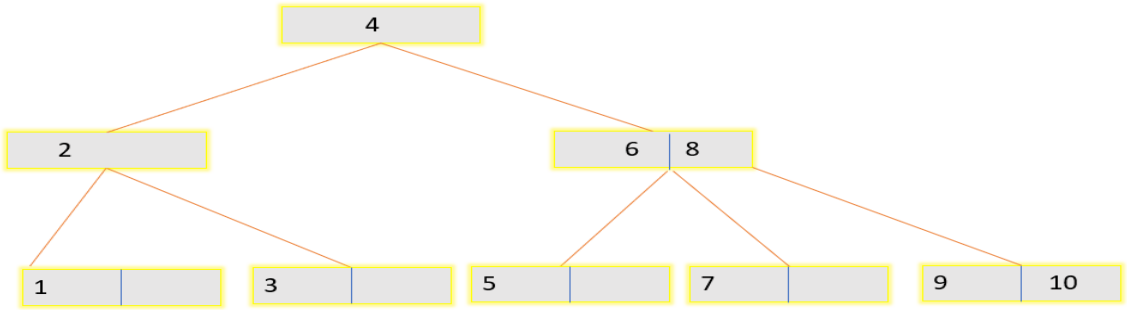
Insert 8



Insert 9



Insert 10



Example 2: Insert the following keys in an empty B-Tree of order 4.

1, 2, 3, 4, 5, 6, 7, 8, 9, 10.

Maximum children in B-Tree of order 4 can be 4.

Maximum keys in one node can be 3.

Minimum keys can be 1.

Insertion of elements in B-Tree follows BST order properties. The key smaller than the given key will go on the left side and the key greater will go on the right side. In the order 4 B-Tree, A node can contain a maximum of three keys. After reaching to a maximum of three keys, while insertion of 4^{th} key, the tree splits and grows upwards.

Step 1

Insert 1: - first key 1 is inserted at root node.

Step 2

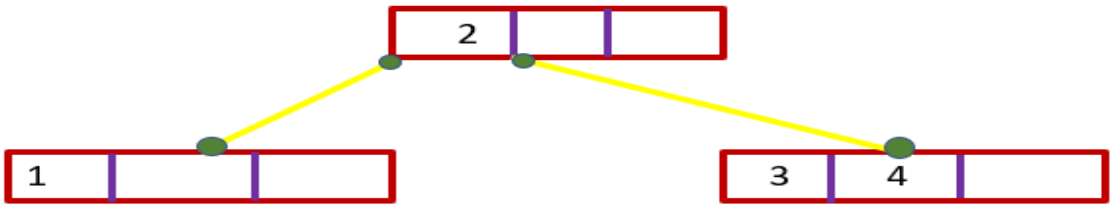
Insert 2: - The second key 2 is greater than 1 hence inserted next to 2.

Step 3

Insert 3: - The third key 3 is inserted to right of 2 as 3>2.

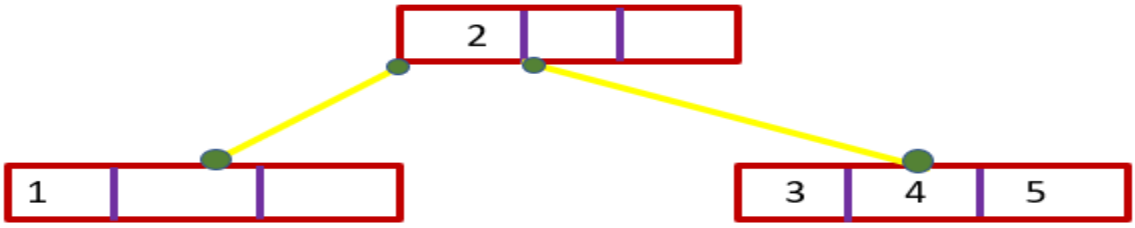
Step 4

Insert 4: Now the node has maximum keys so it splits from the middle and grows upwards. Here 2 will be the root node. Element smaller than 2 will be on LHS and greater than 2 will be on RHS. Note that we have already discussed that root can have 1 key (minimum).



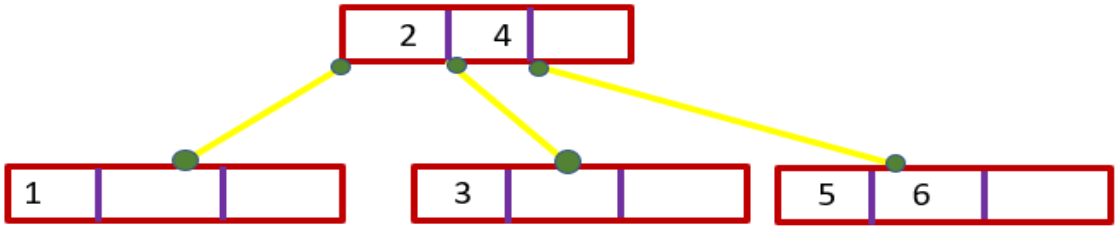
Step 5

Insert 5: - Now 5 will be inserted at leaf node, we search the correct position of 5 from root node, here 5 is greater than 2 hence will be inserted as a right child of 2. Here 5 is inserted to the right to key 4 as 5 > 4.



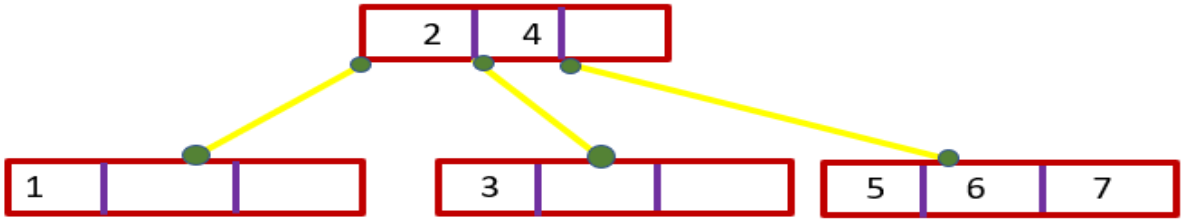
Step 6

Insert 6: - Now the node has maximum keys, so it splits from middle and grows upwards. Here 4 will shift in the root node. Keys less than 4 will be on LHS and greater than 4 will be on RHS.
Key 6 will be inserted as the Right child of node 4 and right to node 5 in leaf node.



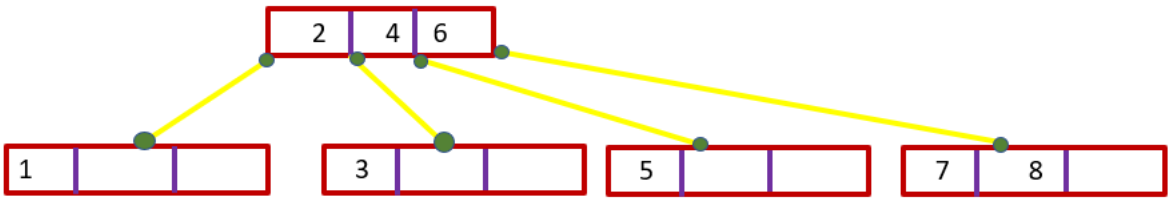
Step 7

Insert 7: - Key 7 will be inserted as the Right child of node 4 and right to node 6 in leaf node.



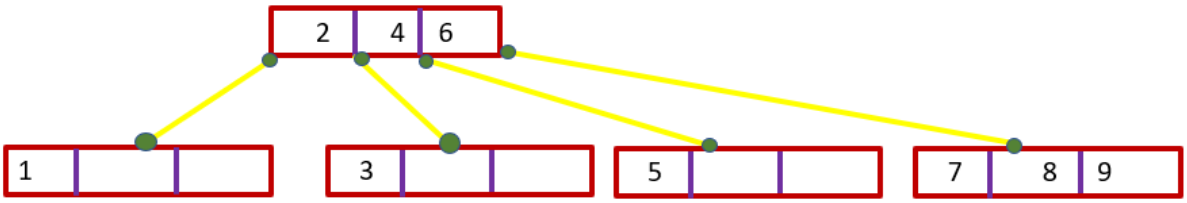
Step 8

Insert 8: - Now the node is having maximum keys, so it splits from the middle and grows upwards. Here 6 will shift towards the root node. Keys less than 6 will be on LHS and greater than 6 will be on RHS.
Key 8 will be inserted as the Right child of node 6 and right to key 7 in leaf node.



Step 9

Insert 9: - Key 9 will be inserted as the Right child of node containing 6 and right to key 8 in the leaf node.

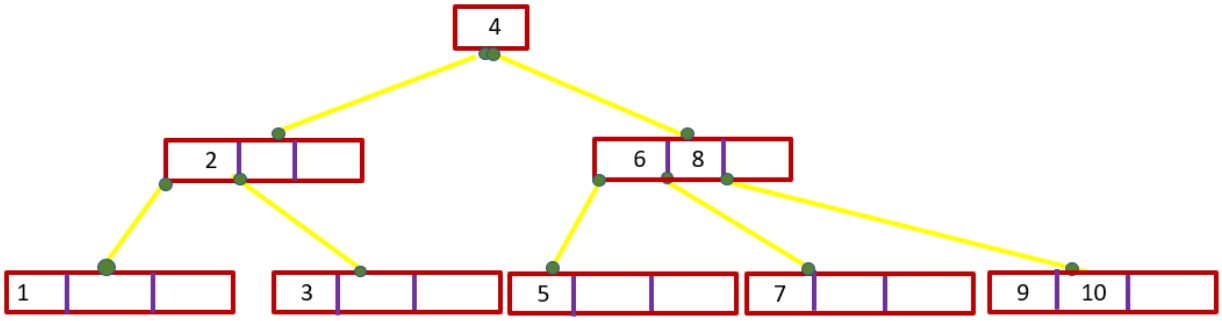


Step 10

Insert 10: - Now the target leaf node is having maximum keys so it splits from the middle and grows upwards. Here 8 will shift towards the root node. Now root node also contains maximum keys; hence another splitting will occur and 4 will move upwards as a new root node of the tree.

Element less than 4 will be on LHS and greater than 4 will be on RHS.

Node 10 will be inserted as the Right child of node containing key 8 and right to key 9 in leaf node.



10.18.3.2 Deletion in B-Tree

The deletion in B-Tree can be done with the help of the following cases: -

- If node to be deleted is in leaf node.
- if node to be deleted is in internal node

Properties of B-Tree that need to be considered: -

Min keys for m order will be ceiling (m/2) -1. Max keys for m order will be m-1.

Min children for m order will be ceiling (m/2). Max children for m order will be m.

Case 1: If node to be deleted is in leaf node

Consider the following B-Tree of order 5. If we want to delete a node from Leaf node,

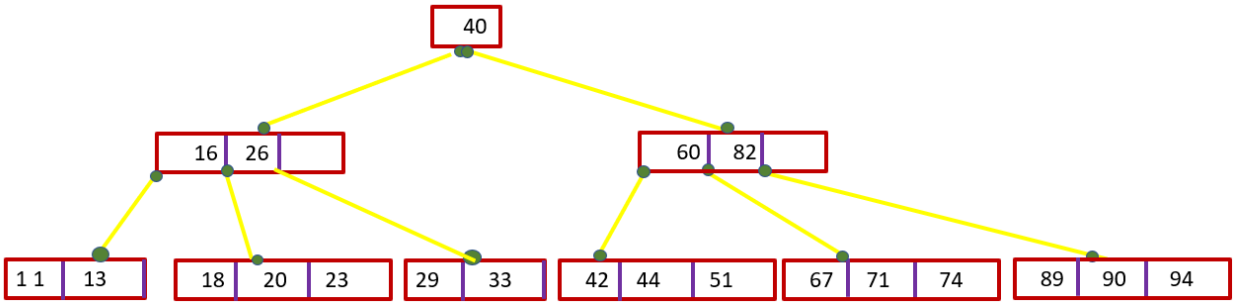


Figure: Initial Tree

Case – a: leaf node contains more than minimum numbers of keys.

For example, if we want to delete 23 key in the above tree. For this, we need first to locate 23. We will start searching this from the root node. As 23 is less than 40, we will search on the left side as the value is greater than 16 and less than 26 it will be lying on the right side of key 16 or at left side of key 26.

Since the minimum number of keys in leaf node can be 2 so we deleted 23 that results in the node having sufficient key.

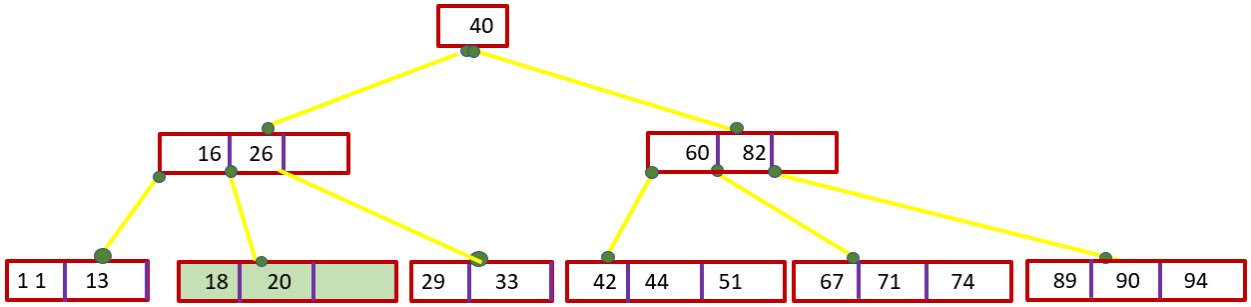
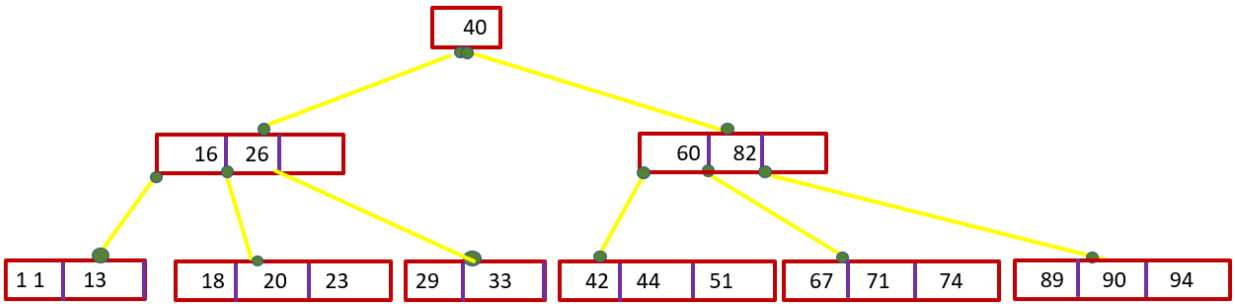


Figure: Delete 23

Case – b: leaf node contains exactly the minimum number of keys.

In this case, we want to delete key 13. This key can be easily found on the left to left side of root 40.



Since the leaf node contains the minimum number of keys, the solution to such case regards the help of the following: -

a. Immediate Left Sibling node of the node containing key to be deleted,

b. Immediate Right Sibling node of the node containing key to be deleted.
c. The parent node of the node containing key to be deleted.

For case a & b, if the immediate sibling contains more than minimum keys, we can transfer keys from sibling to sibling via parent node.

In case c, if both the immediate left and right sibling nodes contain minimum required keys then we can merge these nodes with the parent node.

Here in the above case, the immediate right sibling consists of keys that are greater than minimum required, so we can transfer the key via root node.

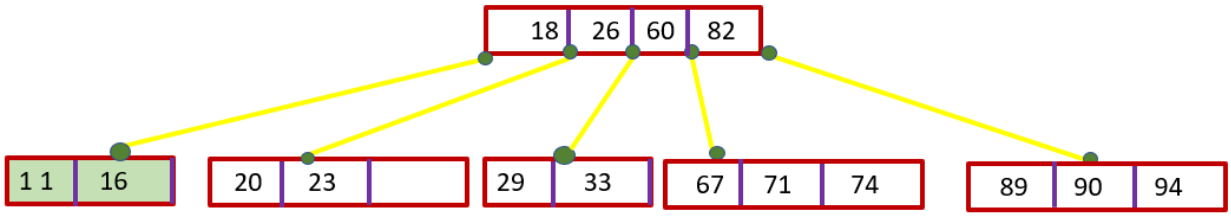


Figure: Delete 13

Take another case; if we want to delete the key 16 from the above formed tree, its immediate right sibling contains the minimum number of keys. In this case, we merge the node which contains key 16 with its right sibling along with bringing the separator key from the parent to this node. After deletion, the tree will be like,

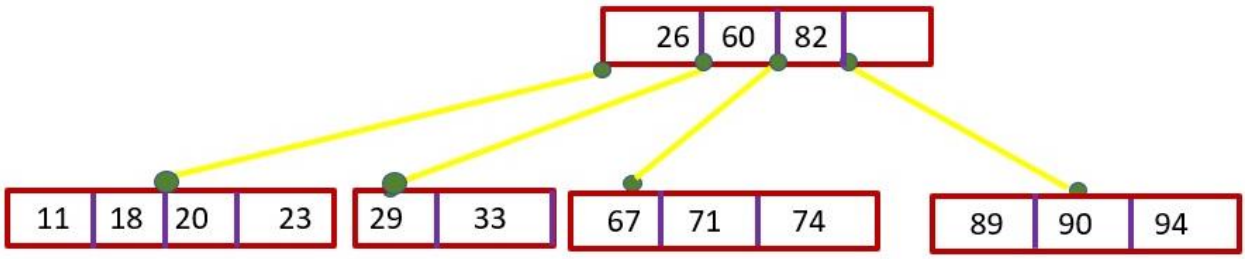


Figure: Tree after deleting 16

Case 2: If we want to delete a key from the Internal node.

If we want to delete any key from the internal node then the following cases could arise.

a) If the required node contains more than required number of keys, in such case, it can be deleted and that key can be replaced with maximum key value from its left child or the minimum key value of its right child. In this case, precedence will be given to that child whose keys are greater than the minimum number of keys required.

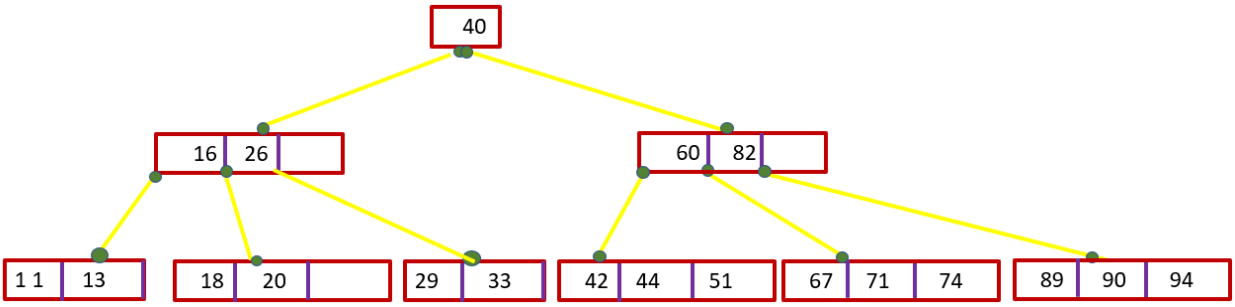


Figure: Delete 82 from Tree

In the above case, we want to delete 82. We can either take 74key (maximum) from the left child or 89 key(minimum) from the right child.

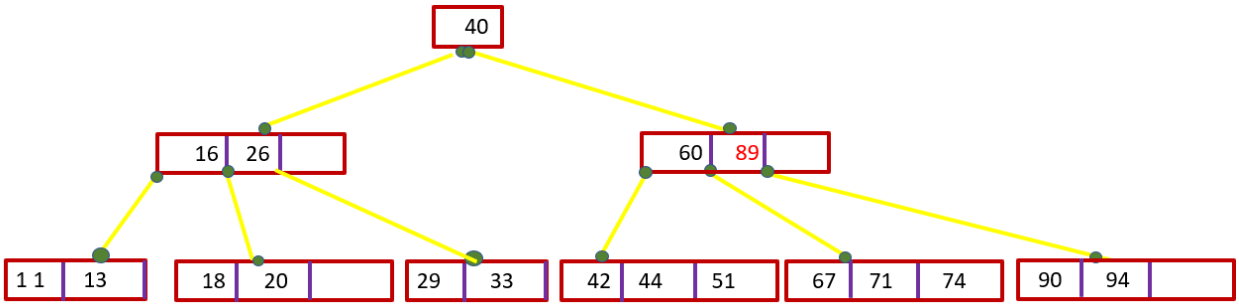


Figure: Tree after deleting key 82

Now, if we want to delete 89 from node, then in this case, 74 will be replaced as its Right child now contains the minimum number of keys.

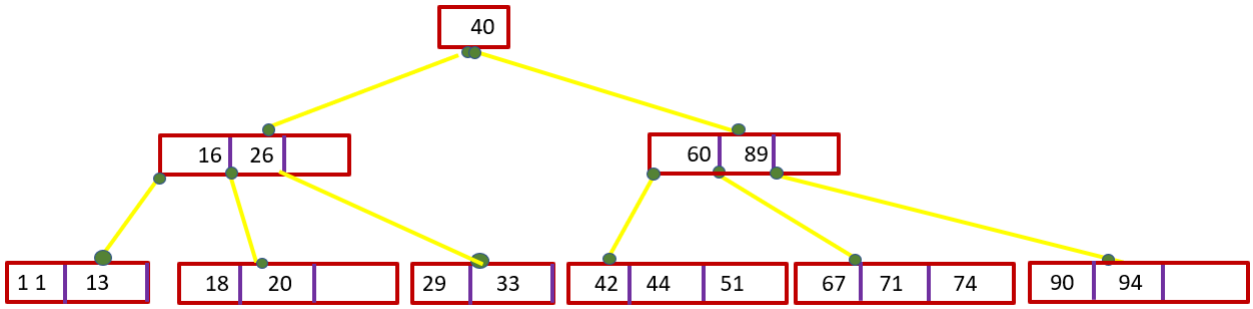


Figure: Tree delete 89

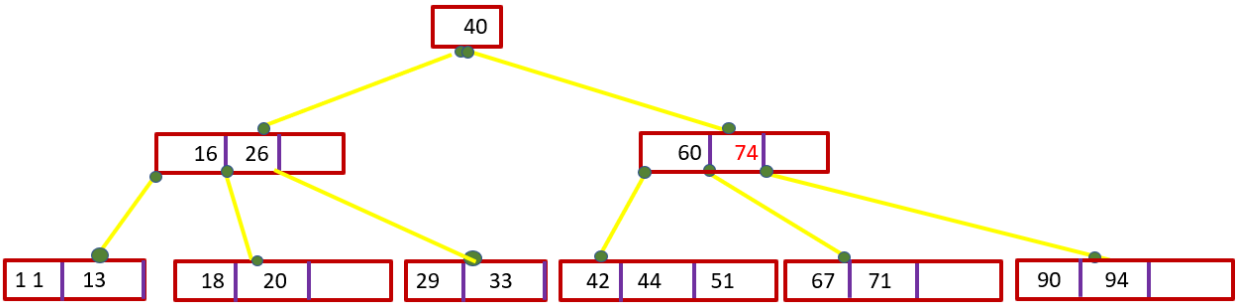


Figure: Tree after deleting 89

b) if the required node contains the minimum number of keys.

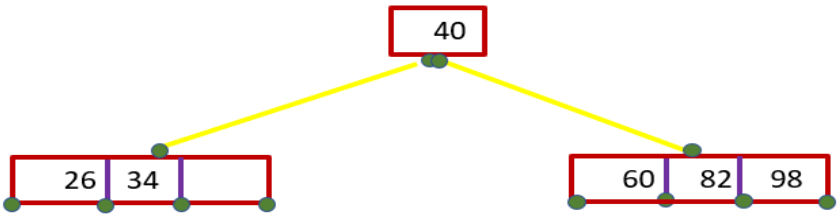


Figure: Delete key 34

If we want to delete 34 from node then as it does not have any child so it will take help of its sibling.

The right sibling of this node contains 3 keys, more than 2, that is minimum one. Here 40 that is root key will be transferred to deleted one i.e. 34 key, and 60 will be the new root.

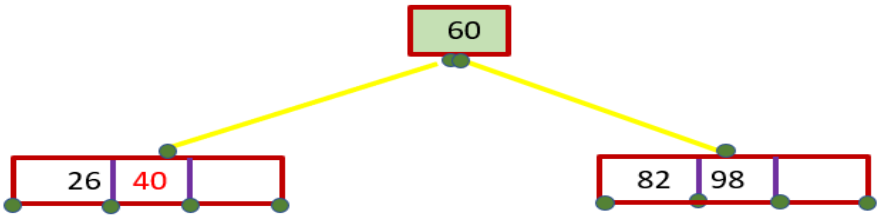
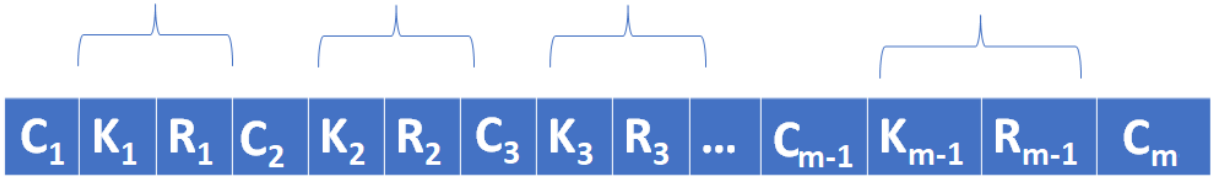


Figure: Tree after deleting 34

10.18.4 Order Computation of a B-Tree

Consider the structure of a B-Tree Node. Each node has a key and Record address pair < K_i, R_i> and the address to the child node (C_i). All the nodes in a B-Tree will have the same structure.



For a node having order m,

- There will be at most m Child nodes
- For each key, there will be a record address. Hence there are m-1 Keys and Record address pairs i.e., < K_i, R_i>.

Let's assume that each key requires x Bytes for storage, each record address be of y Bytes and each child pointer is of size z Bytes. Every B-Tree node should be adjusted in the Hard Disk Sectors. As usual, size of each of the sectors in the Hard disk is 512 Bytes; each B-Tree node should exactly fit into the Hard Disk sectors.

Size of each node = (m-1*)size of key+ (m-1)*size of Record address + (m)*size of Child node Pointer

i.e. ***(m-1)*x + (m-1)*y + (m)*z <=512 Bytes***

Question: Consider a B-Tree of Order m. Considering each key of 8 Bytes, Record address of 4 Bytes, and Child node Address of 9 Bytes, Compute the order of the B-Tree such that it gets adjusted to exactly a Hard Disk Sector.

x = 8 Bytes

y= 4 Bytes

z= 9 Bytes

(m-1)*8+(m-1)*4+(m)*9 <= 512

21m – 12 <=512

21m <= 524

m <= 524/21

m<=24.95

As m cannot be a fraction, it should be 24

10.19 B+ Tree

B+ Tree is an m-way search Tree in which internal nodes behave like the indexes and leaf nodes as data nodes.

10.19.1 Properties of B^+Tree

For any order-m B^+Tree,

1. Each node has at most m children.
2. Each node has at most m-1keys.
3. A node with k children has k-1 keys. (1<=k<=m)
4. All leaf nodes are at the same level
5. Every node (except root node) has a restriction of containing at least (m-1)/2 keys.
6. The root node can have the number of keys less than (m-1)/2 but at least one key.
7. Root has at least had two children (if it is not leaf).
8. Keys in the nodes are arranged in non-descending order ($k_1<=K_2<=K_3<= \ldots <=K_m$)
9. Leaf nodes are connected with each other.

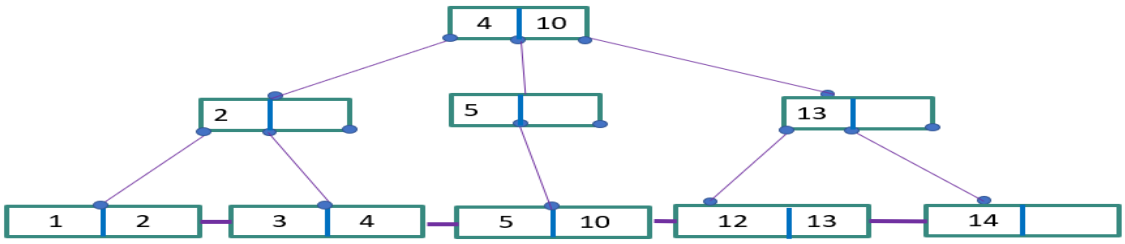
10.19.2 Difference between B-Tree and B^+Tree

In B-Tree data is stored in leaf node as well as in the internal node. In B+ tree data is stored in leaf node only.

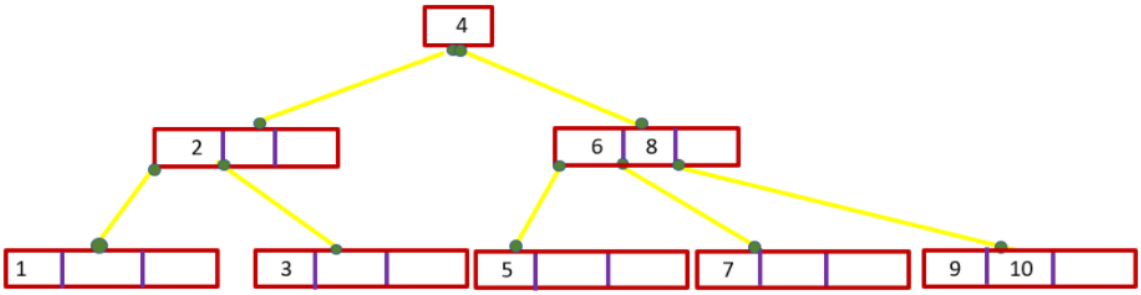
Searching is slower in B-Tree, while searching is faster in B+ tree.

Deletion is complex in B-Tree as compared to deletion performed in B+ tree, which is quite simple.

In B+ tree all leaf nodes are connected together like a linked list.



B+ Tree – leaf nodes are connected together.



B-Tree- all leaf nodes are not connected.

10.19.3 Insertion in B$^+$Tree

Insertion in B$^+$Treeis exactly like other search tree. For fast accessing, all the data lies in leaf node. The internal node, on the other hand holds the index that guides to reach to the leaf nodes that contain the data keys and record addresses.

Let us take an example where we insert the following keys in B$^+$Tree of order 3:-
5, 10, 12, 14, 13, 1, 2, 3, 4.

Minimum children will be 2 Minimum keys will be 1
Maximum children will be 3 Max keys will be 2

Step 1

Insert 5

Since key 5 is the first value to be inserted, it can easily be placed into a node.

Step 2

Insert 10

Now key 10 is being inserted into root node and since maximum keys that can be inserted is 2, it is inserted in increasing order into root node.

Step 3

Insert 13

Now, if we insert 13 in the node then the node will have a count of three, which is greater than the minimum number of keys required; hence we perform splitting from middle and key 10 (the middle element)goes up and forms the new root node.

Here, as all data are present in leaf node, so we have inserted key 10 as a value in the left child of node 10 that is in leaf node. Here it can also be put as a value in the right child of node 10. Only the copy of the data goes up while retaining the same key in the leaf node.

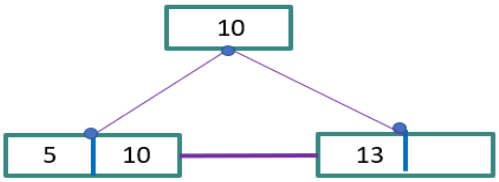


Figure: Insert 13

Step 4

Insert 12

Here the key 12 will be inserted towards the right child of node containing 10.

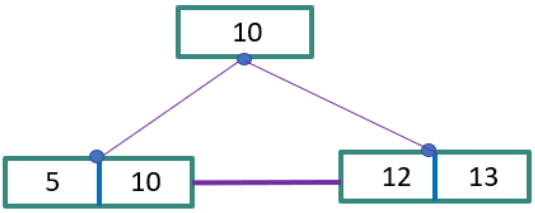


Figure: Insert 12

Step 5

Insert 14

If we insert 14 in the target leaf node, it has to be inserted into the right side of key 13, making count of keys in this node to three, which is greater than the minimum number of keys required. Hence, we perform splitting from the middle and key 13, the middle element, will go up to root node while retaining 13 in the leaf node as well.

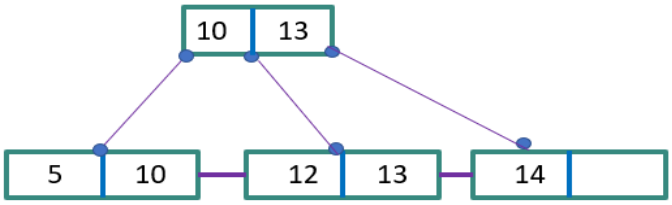


Figure: tree after inserting key 14

Step 6

Insert 1

If we insert 1 in the node then it has to be inserted to the left side of key 5, which will make count of keys of this node to three (greater than the maximum number of keys). Hence, we perform splitting from middle and key 5, the middle element, will go up to the root node. Now in the Root node total keys are 3, which too is not allowed so we perform splitting of root node as well. Middle element 10 will go up forming the new root. Nodes containing 5 and 13 will be its left and right child respectively.

Note ➔ *Splitting of Leaf node requires median key to be retained in the leaf node on the other hand, splitting of internal node does not require so (It is same as that in B-Tree Splitting).*

Here, as all data are present in leaf node so we have inserted key 5 as a value in left child of node 5 in leaf node.

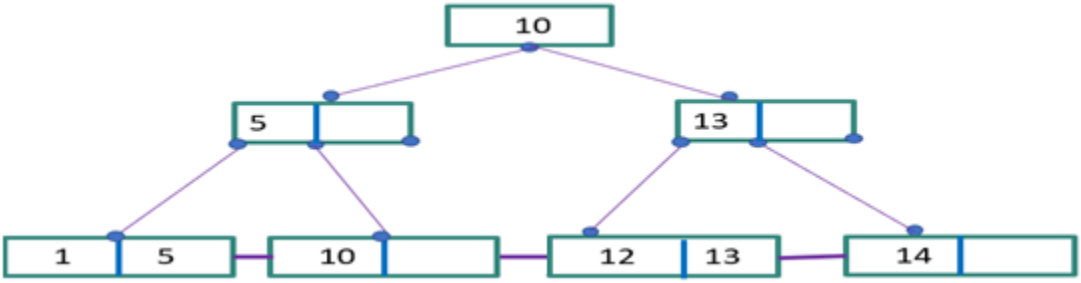


Figure: Tree after inserting 1

Step 7

Insert 2

On insertion of 2 in the leaf node, it will be initially inserted right to key 1, which will make count of keys to 3. Hence splitting occurs and 2 will go to the upper level while retaining it in the leaf node.

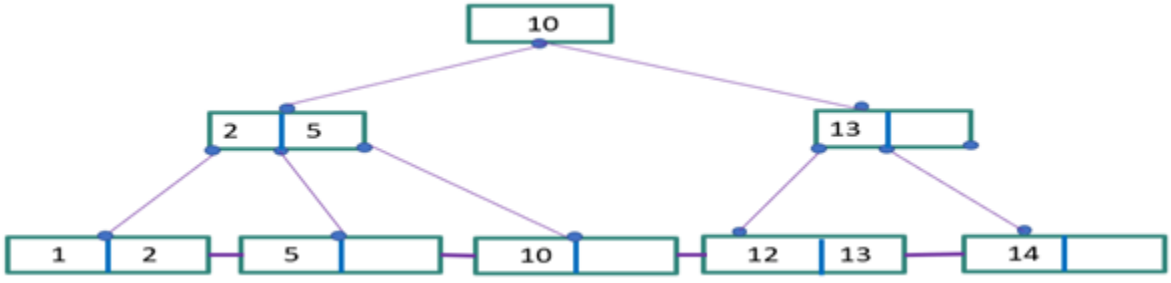


Figure: Tree after inserting key 2

Step 8

Insert 3

Key 3 can be easily inserted as a right child of 2 and at first index so as to get element in increasing order.

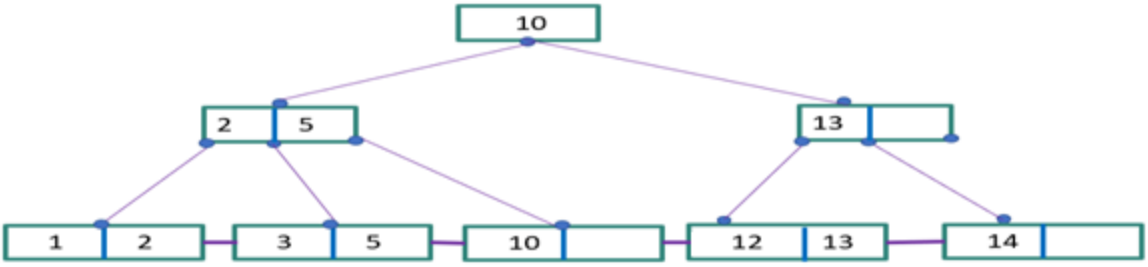


Figure: Tree after inserting key 3

Step 9

Insert 4

If we insert 4, it has to be inserted in the right side of leaf node containing key 3.This will make count of keys of this node to three, which is greater than the maximum number of keys required. Hence, we perform splitting from middle and key 4, the middle element, will go up to internal node. The internal node too contains total keys 3, which is greater than the maximum number of allowed keys. We perform splitting on this node and 4 will go up to be added to the root node. Two new nodes with key 2 and 5 will be left and right child of the separator key 4.

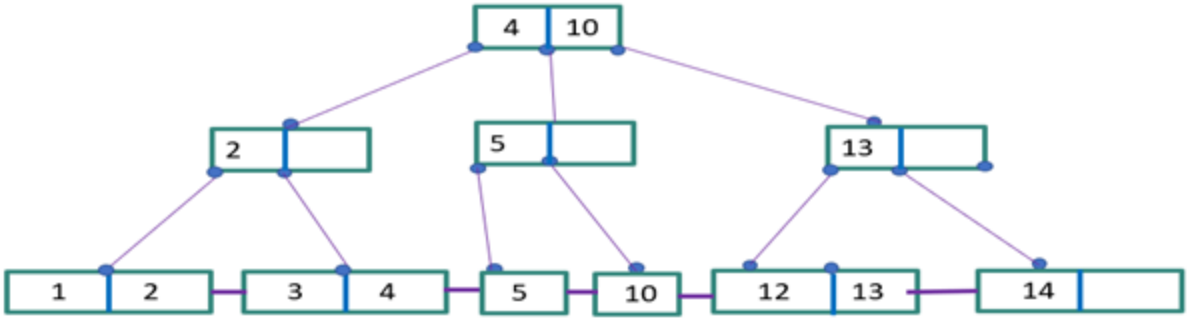


Figure: Tree after inserting key 4

10.19.4 Deletion in B$^+$Tree

Deletion in B$^+$Tree is relatively simpler as compare to B-Tree. In B$^+$Tree, all data lies in leaf node, so we need to traverse till leaf node in order to delete the key, meanwhile, if data to be deleted lies at both leaf and index node then we delete key at both the places.

We will follow this technique while deleting through both Right biasing and left biasing. (By biasing, we mean to say the side where index value is stored in leaf node).

Let us take an example of Right biasing for deleting the nodes in B$^+$Tree of order 4.

Max children = 4 Min children = 2

Max keys = 3 Min keys = 1

The keys to be deleted are 28, 31, 20, 10, 7, 25, 42, 4.

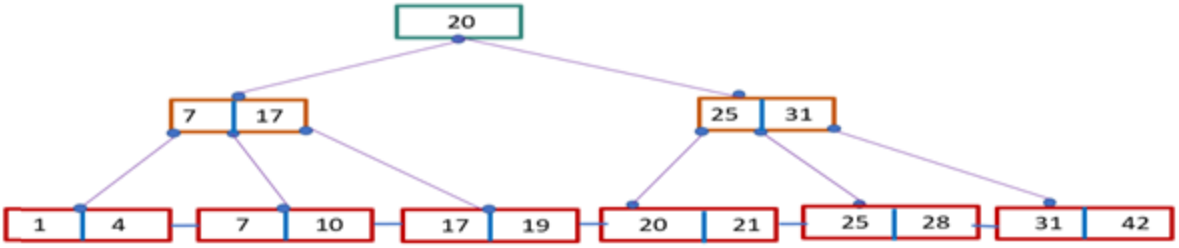


Figure: Initial B+Tree

Step 1: when node contains more than minimum required key

Delete 28: To delete key 28, we need to perform search for it from the root node. Since it lies between index 25 and 31 that is, it lies in the leaf, which is right child of 25.

The number of keys in the target node is 2, which is greater than 1(min keys), so we can delete this key with ease.

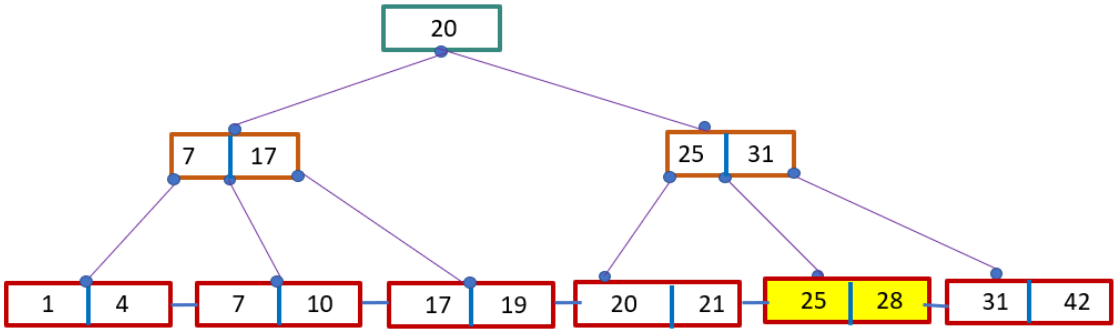


Figure: key 28 to be deleted

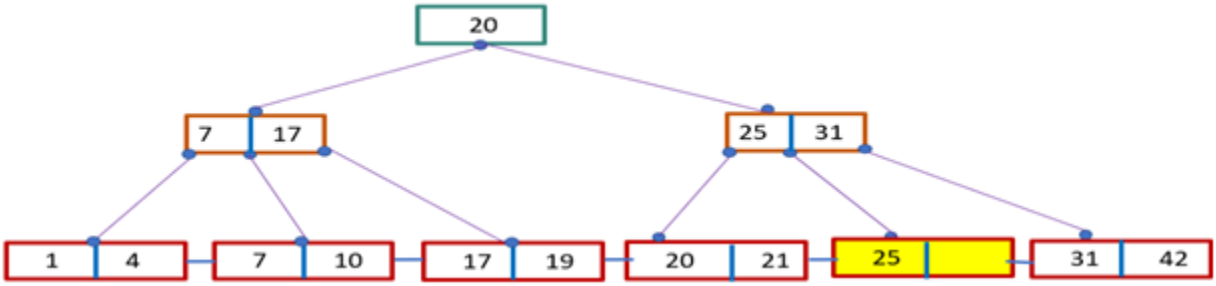


Figure: key 28 deleted

Step 2: When node contains more than minimum required key and key is in index too.

Delete 31: To delete key 31, we need to perform the search from the root node; since it lies after index 25, it lies in leaf, which is right child of 31.

The number of keys in this node is 2, which is greater than 1(min keys), so we can delete this with ease.

In this case, index 31 is replaced by the next minimum value of this node which is 42.

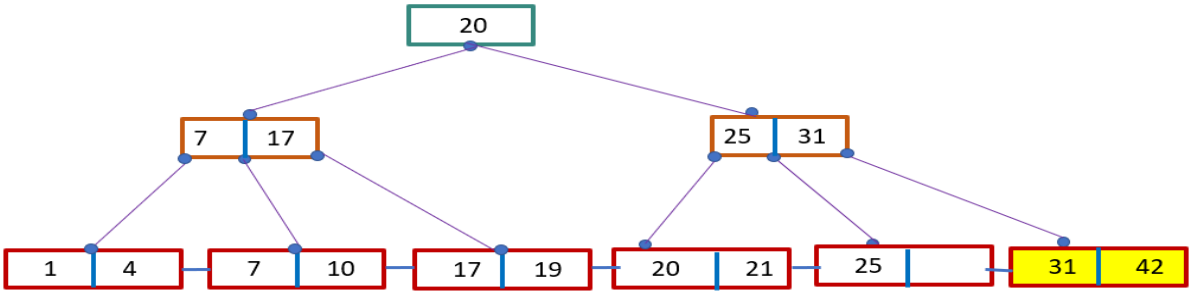


Figure: Delete key 31

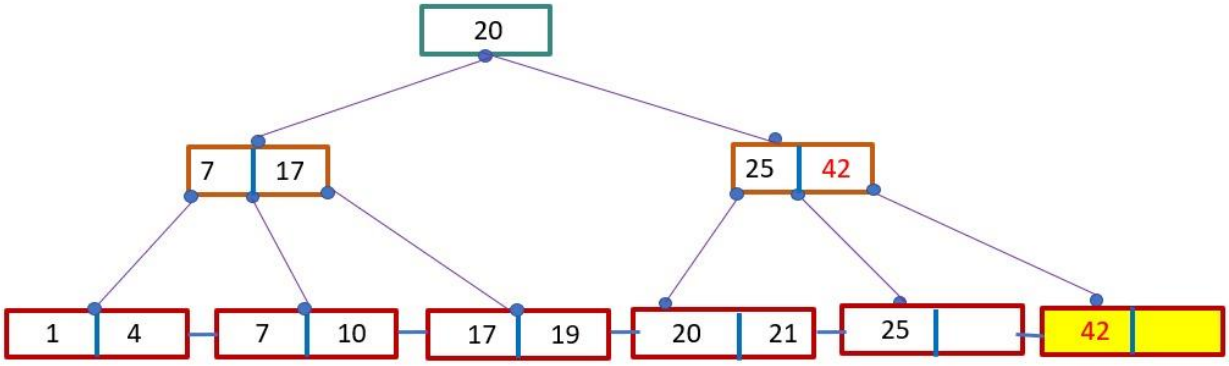


Figure: Tree after deleting 31

Step 3: Delete 20 when node contains more than minimum required key and key is an index.

To delete key 20, we need to search for it from root. It lies in the root itself and further in the leaf node that contains 2 keys which is greater than 1 (min keys). We can delete this with ease. In this case, index 20(root node) is replaced by next minimum value that can be found in the right sub part of tree. It lies in the leaf node, 21 here.

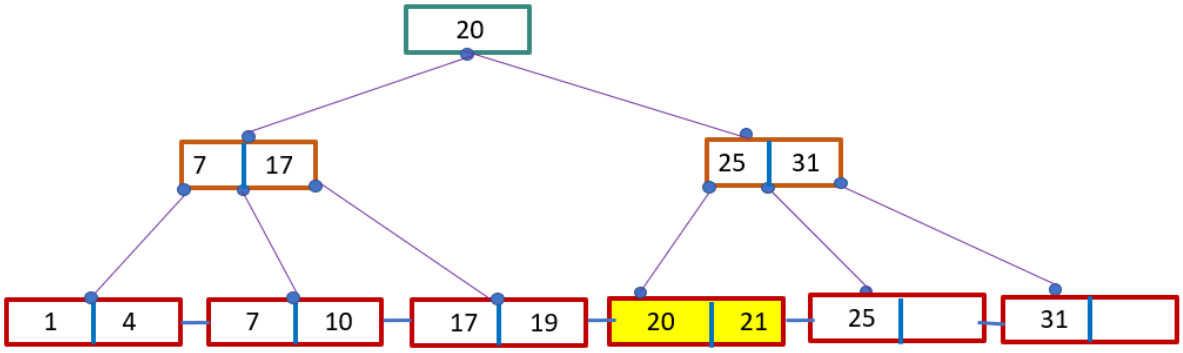


Figure: Delete 20

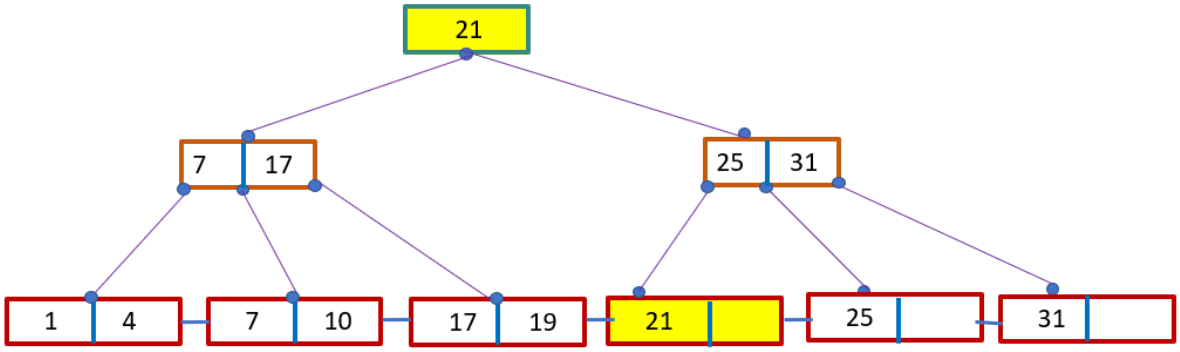


Figure: Tree after deleting 20

Step 4: Delete 10, when node contains more than minimum required keys and key is not index. To delete key 10, we need to search for it from root. It lies in left subpart of tree in a leaf node only.The number of keys in this leaf node is 2, which is greater than 1(min keys), so we can delete this with ease.

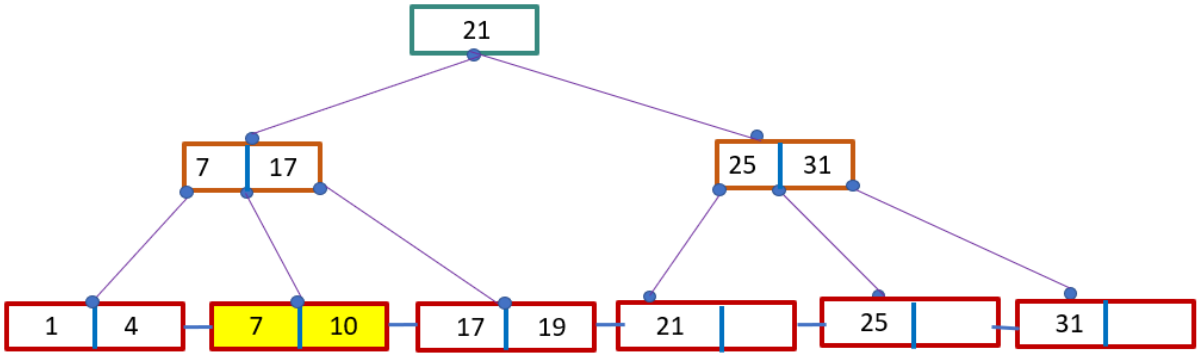
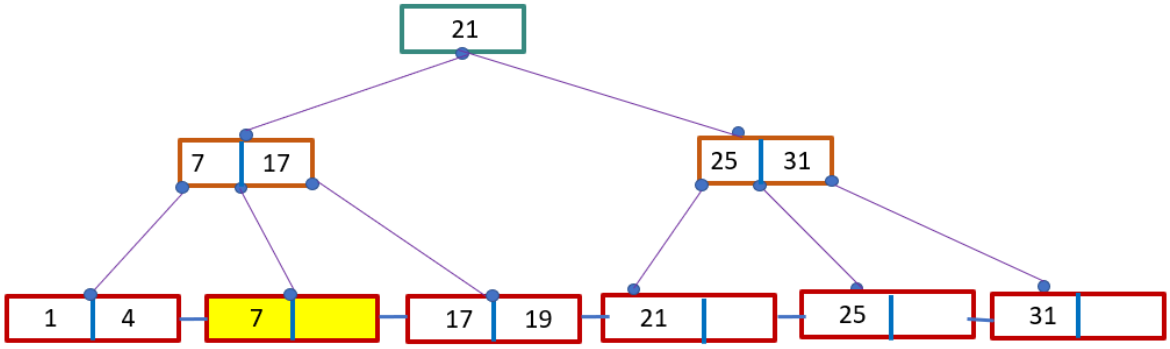


Figure: Delete 10



Tree after deleting 10

Step 5: Delete 7, when node contains more than minimum required key and key is in index.

To delete key 7, we need to search for it from root. It lies in left subpart of tree in a leaf node. It is also part of one of the index node.

The number of keys in this leaf node is 1, which is 1(min keys), so we cannot delete this with ease. Here it can borrow from any one of its siblings as they contain more than minimum keys required.
We are firstly seeking its left sibling, which contains two keys. Hence key 4(max in left sibling) can be shifted to both index and leaf node.
If we have sought its right sibling, then its minimum key would have been shifted to both at index position and leaf node.

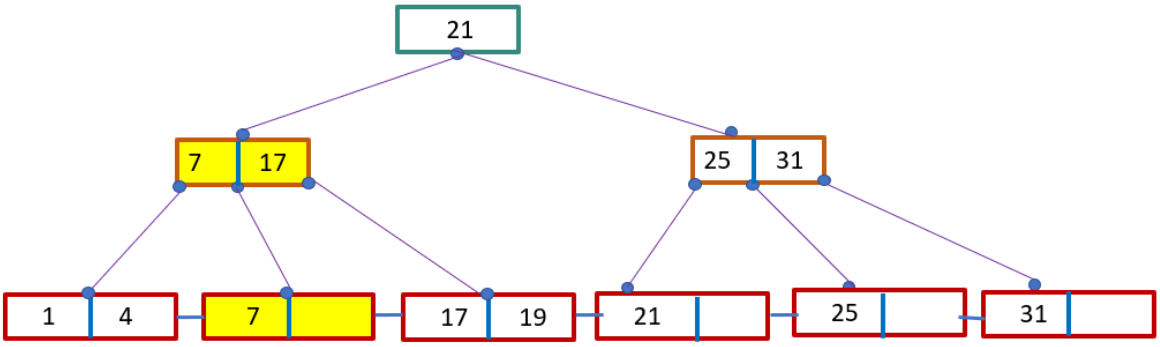


Figure: Delete 7- minimum key at leaf node.

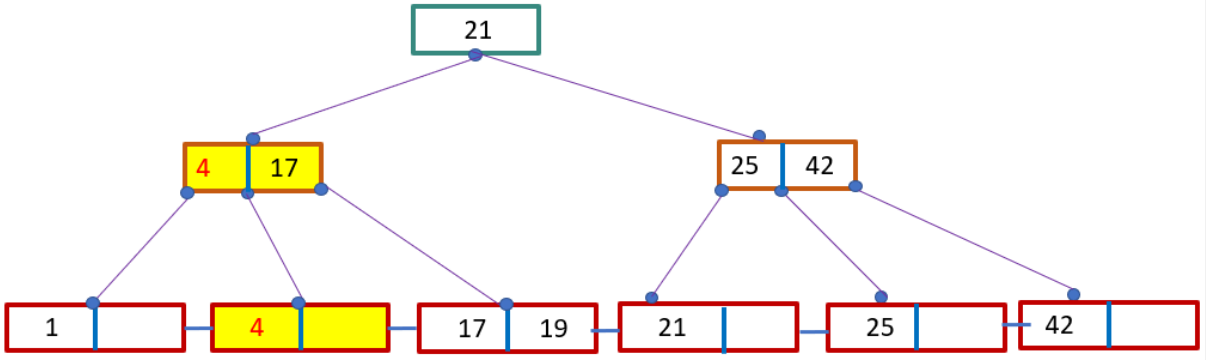


Figure: Tree after performing deletion of key 7

Step 6: Delete 25, when node contains exactly minimum required key.

To delete key 25, we need to search for it from root. It lies in right subpart of tree.
The number of keys in the target leaf node is 1(min key), so we cannot delete this with ease. Here it can borrow from any one of its siblings but they too contain minimum keys.
In this case left, right and target node contains minimum key, but parent contains more than min keys. We will merge the node containing key to be deleted with its left sibling and then we can delete key 25.

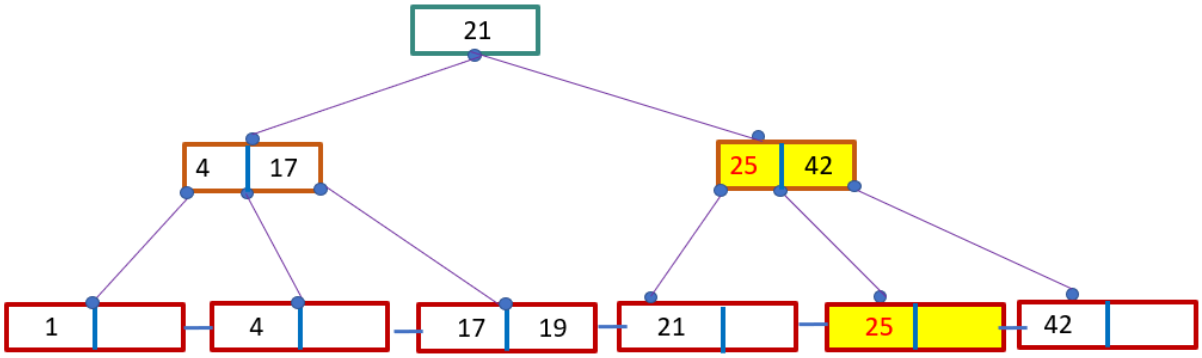


Figure: Delete key 25 from tree.

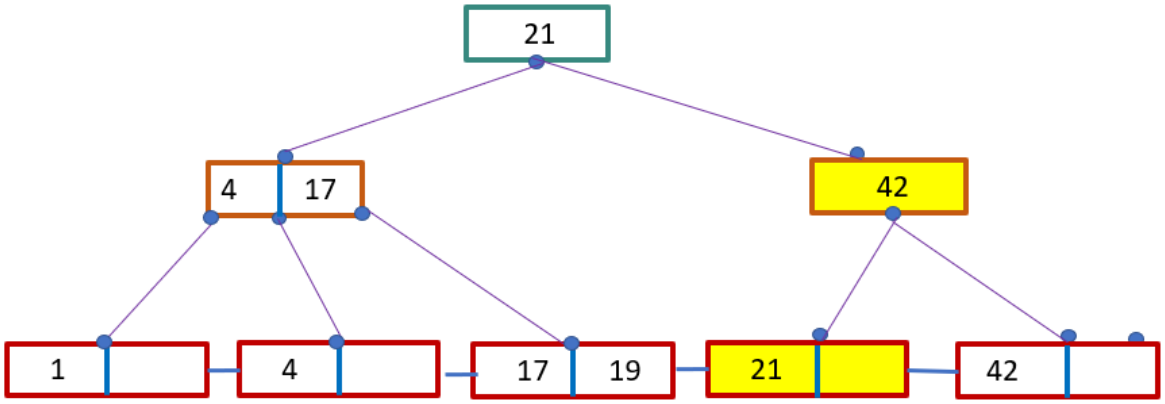


Figure: Tree after deleting 25

Step 7: Delete 42, when node contains exactly minimum required key.

To delete key 42, we need to search for it from root. It lies in right subpart of tree.
The number of keys in the target leaf node is 1, which is 1(min keys), so we cannot delete this easily. It can borrow from any one of its siblings, but that too contain minimum keys.
In this case, we will see the left sibling of index node (with 42)that contains two keys 4 and 17.
So here we shift maximum key that is 17 to another side.

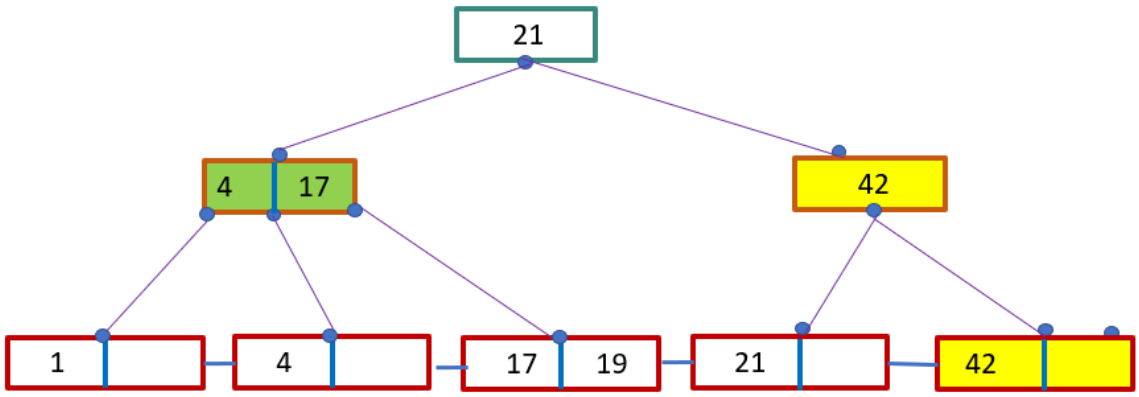


Figure: Delete key 42 from tree

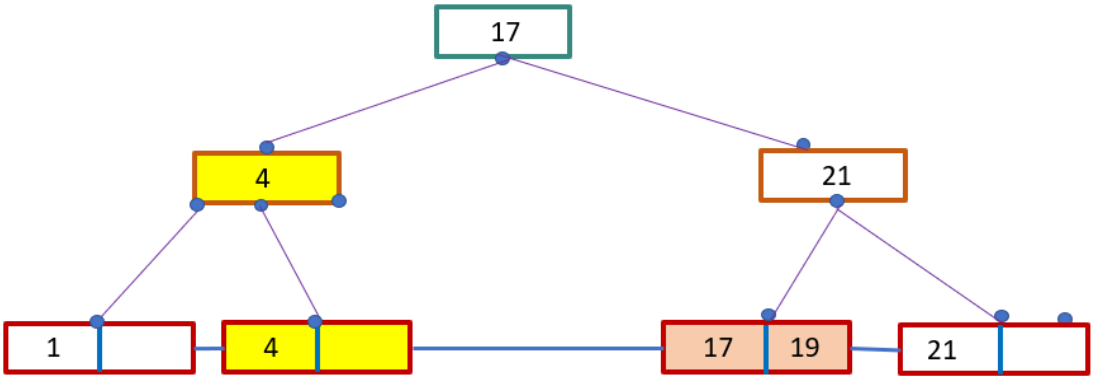


Figure: Tree after deleting 42

Step 8: Delete 4, when node contains exactly minimum required key.

To delete key 4, we need to search for it from root. It lies in the left subpart of tree. Number of keys in the target leaf node is 1 (min keys), so we cannot delete this with ease. It can borrow from any one of its siblings and parents, but all these nodes contain minimum keys.

In this case, we will see the right sibling of the index node containing 4; this also contains 1 key only.

So here we will perform merging of index 4, index 21 and root 17.

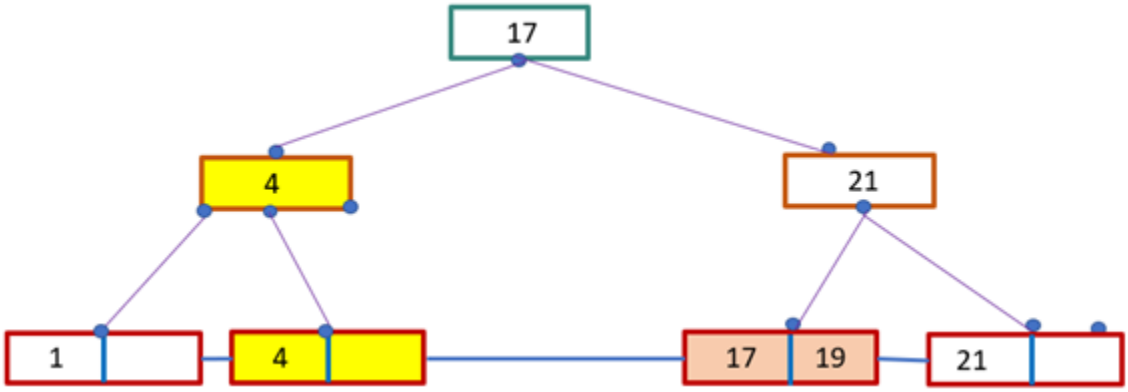


Figure: Delete 4 from Tree

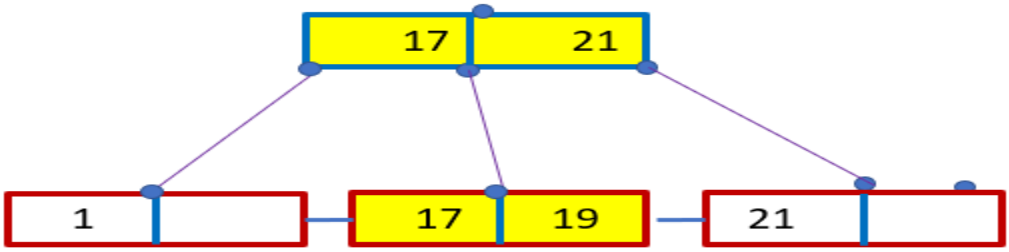
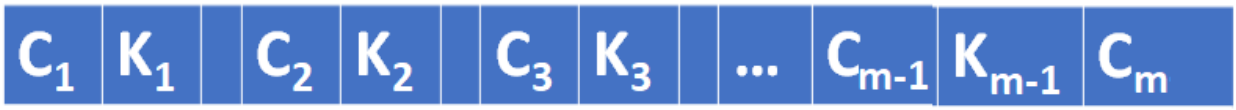


Figure: Tree after deleting 4

10.19.5 Order Computation of a B+Tree

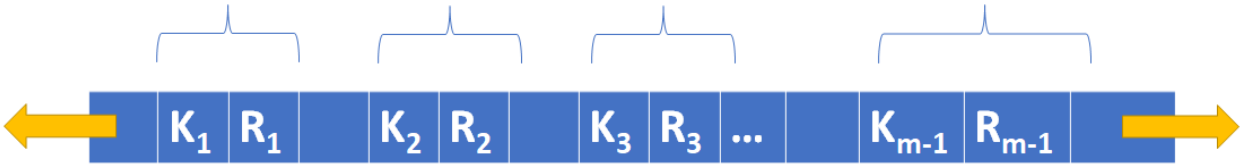
Consider the structure of a B+Tree Nodes. Internal nodes are treated as the indexes that guide us to reach the Leaf nodes containing the address of the desired record. Hence the structure of internal nodes is different from that of the Leaf nodes.

Structure of internal nodes



Each internal node contains the Key and address of the child node. For an order m B+Tree, there would be m-1 keys (maximum) and m child node addresses.

Structure of leaf nodes



Each Leaf node has a key and Record address pair < K_i, R_i>along with the address previous and next leaf node.

For a Leaf node having order m,

- There will be 2 addresses to the left and right leaf nodes
- For each key, there will be a record address. Hence there are m-1 Keys and Record address pairs i.e.< K_i, R_i>.

The order of Leaf node and internal nodes would be different.

For internal nose

Let us assume that each key requires x Bytes for storage; each child pointer is of size z Bytes. Every B+Tree node should be adjusted in the Hard Disk Sectors. As usual size of each of the sectors in the Hard disk is 512 Bytes, each B+Tree internal node should exactly fit into the Hard Disk sectors.

Size of each node = (m-1)*size of key + (m)*size of Child node Pointer

i.e. (m-1)*x + (m)*z <=512 Bytes

For Leaf node

Let us assume that each key requires x Bytes for storage, record address be of y Bytes and each sibling pointer is of size z Bytes. Every B+Tree Leaf node should be adjusted in the Hard Disk Sectors. As usual, the size of each of the sectors in the Hard disk is 512 Bytes; each B+Tree Leaf node should exactly fit into the Hard Disk sectors.

Size of each node = (m-1)*size of key + (m-1)*Record Address + (2)*size of Child node Pointer

i.e. (m-1)*x + (m-1)*y + (2)*w <= 512 Bytes

Question: Consider a B^+Tree of Order m. Considering each key of 8 Bytes, Record address of 4 Bytes, and Child node Address of 9 Bytes, Sibling node address of 9 Bytes, Compute the order of the B^+Treeinteral node and Leaf node.

Key size (x) = 8 Bytes

Record Address (y) = 4 Bytes

Child Pointer (z) = 9 Bytes

Sibling Pointer (w) = 9 Bytes

For Internal node:

(m-1)*8 + m*9 <= 512

17*m – 8 <=512

17*m <= 520

m <= 520/17

As m cannot be fraction, it should be 30

For Leaf node:

(m-1)*8 + (m-1)*4 + 2*9 <= 512

12*m + 6 <=512

12*m <= 506

m <= 506/12

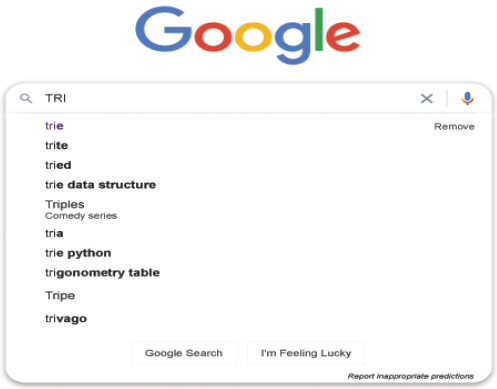
As m cannot be fraction, it should be 40

10.20 TRIE Data Structure

10.20.1 Introduction

In Google search, when we type some character sequence, Google suggests some strings which may or may not be of our choice. Suppose when we start searching TRIE at Google, the first typed character would be T; Google suggests strings that start with T. When we type TR, then Google refines the search string that suggests strings starting with TR and so on.

The diagram given below shows the remaining steps:



From above, we conclude that Google suggests the strings which we enter as a prefix. We can hence conclude that TRIE data structure is that allows the prefix-based search.
TRIE is very helpful to implement dictionaries in the computer as search on the dictionary is also based on prefix.

A typical TRIE Node Contains

- One field of TRIE Node is the children field, which holds the address of children
- The second field is a Boolean variable that indicates the end of text.

Boolean Value IsEndOfText	Children[size]

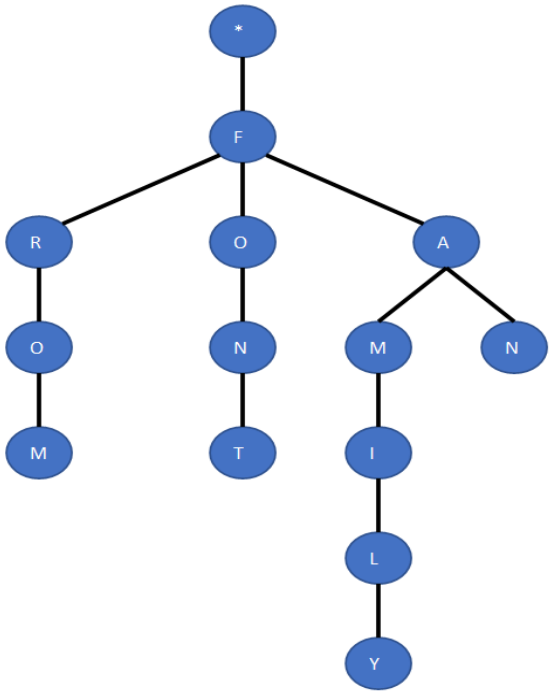
Implementation of Dictionary: There is 26 English alphabets; hence each trie node will contain 26 children addresses. Another field (a Boolean value) indicates the end of text.

Boolean Variable	Children[0]	Children[1]	Chidren[2]		Children[25]
IsEndOfText	NULL	NULL	NULL	...	NULL

In this Trie node,

- We are assuming only the English alphabet. For increasing the number of characters, one will have to increase the size of children nodes.
- Children fields in the Trie node contain the addresses of children of nodes.
- Alphabets denoted by children[i] is **A, B, C,…,Z**for**i= 0,1,2,..,25** respectively

A TRIE data structure looks like



10.20.2 Trie Algorithms

10.20.2.1 Insertion If we wish to insert a new word S in the Trie, having ROOT as the root node.

ALGORITHM Insert (ROOT, S)

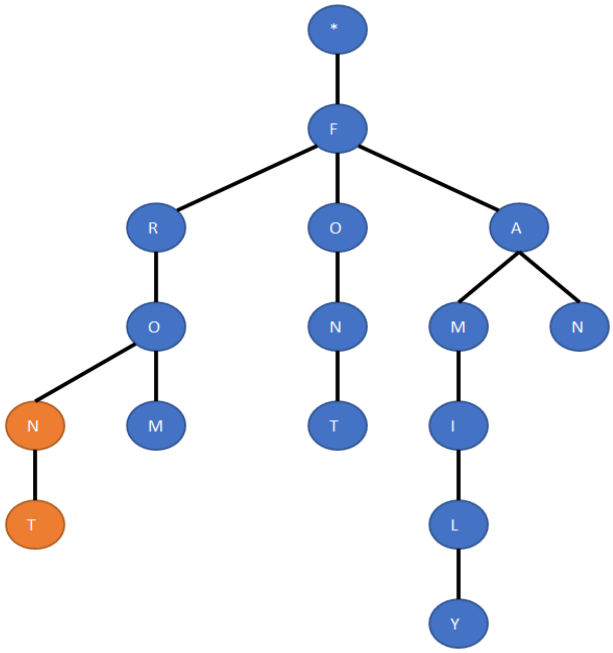
BEGIN:

```
FOR i=0 TO S[i] !='\0'  DO
    X=S[i] - 'A'
    IF ROOT→CHILD[X] == NULL THEN
        ROOT→CHILD[X]=GetNode()
    ROOT= ROOT→CHILD[X]
ROOT→EOW=TRUE
```

END;

let us suppose that we have to add a new world FRONT in existing TRIE.

1- The first three letters of the world FRONT are F, R and O. If these letters are already present in TRIE then we shall move to the node with value F in the first level, move to the node with value R in second level and move to a node with value O in the third level.
2- The next letter in FRONT world is N. In the existing TRIE there is no child with value N of the node O. New node needs to be inserted with value N.
3- The next letter in FRONT world is T. In the existing TRIE there is no child with value T of the node N. New node needs to be inserted with value T.
4- Since T is the last letter of the word FRONT so mark it as the end of text by setting the Boolean variable true,



10.20.2.1 Search

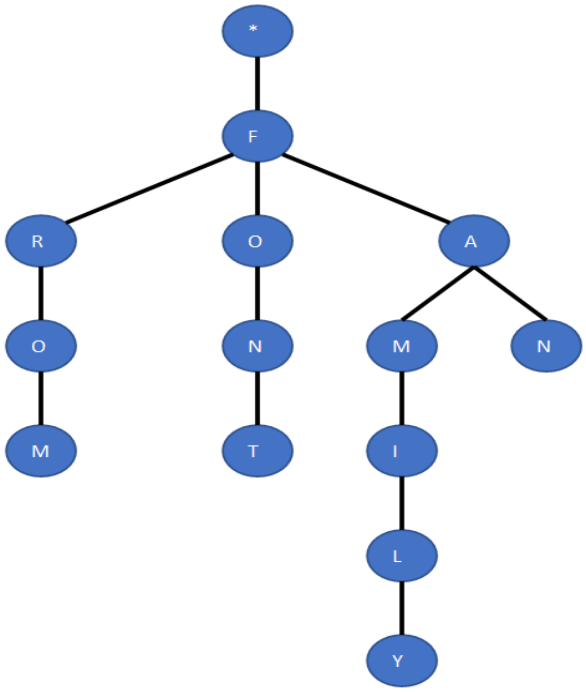
If we wish to Search for a new word, S in the Trie having ROOT as the root node.

```
ALGORITHM search(ROOT,S)
BEGIN:
        FOR i=0 TO S[i] !='\0' AND ROOT !=NULL DO
            ROOT= ROOT→CHILD[S[i] –'A']
        RETURN (ROOT !=NULL && ROOT→EOW)
END;
```

Let us suppose we have to search a world 'FAN' in the given TRIE.

- Firstly search the letter 'F' in TRIE,If it is present, then move to the next level and search next letter. Here 'F' is present, so move next level and search letter 'A'

- Letter ‘A’ is also present, so move next level and search letter ‘N’
- Letter ‘N’ is also present and this is the last letter in the world. Now we have to check in TRIE if after ‘N’ it is marked by the end of text then FAN exist in TRIE otherwise not.



10.21 Segment Tree

10.21.1 Introduction

Segment Tree is used for range queries. E.g.

- Finding minimum in a certain range
- Maximum in a certain range
- Sum of elements in certain range

In the case of many queries (say m), the complexity of operations would be O(mn), where n is the number of elements in the set. In case m and n are huge, m.n will be an even bigger quantity.

10.21.2 Finding Minimum in a range

First Approach

Let us consider we have to find the minimum of 3,-1,2 in enquired range of indexes. If we build a 2-D Array to find this, we can directly store the values in the various possible range.

	0	1	2
0	3	-1	-1
1		-1	-1
2			-1

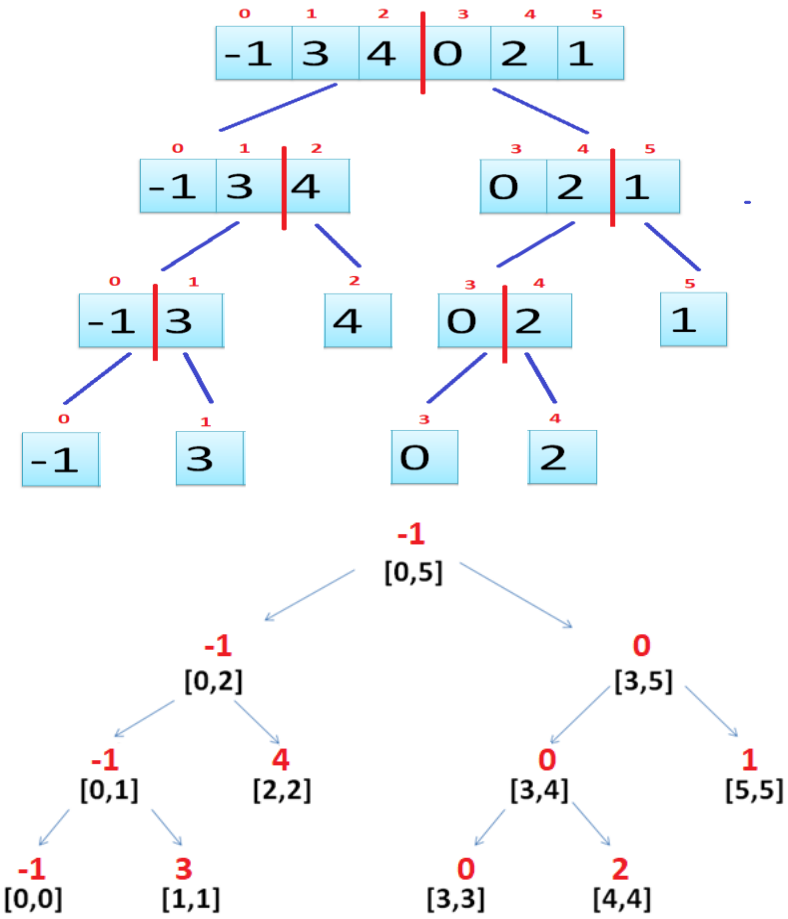
The method requires $O(N^2)$ Time and the same amount of space i.e. $O(N^2)$, for storing the values in the form of Matrix. But the Query requires only O(1) time as all the values are already set in the Matrix.

Second Approach, Segment Tree construction

If the segment tree is formed from the set, it requires O(N) time for the construction and O(logN) time for the Queries.

For this, we need to follow the divide and conquer process in which the array is divided into equal halves repeatedly until the divisions are not possible.

e.g.



The Queries performed on the Segmented Tree can be of 3 types:

- Partial Overlap (Look on both the sides of the Tree i.e., Left and Right SuB-Tree)
- Total Overlap (Stop and return the value at the tree node)
- No Overlap (Stop and return a very large number)

Example 1: Querying for finding Minimum in the range [2,4]

Root node [0,5] has partial overlap with [2,4]; look in left and right both the directions

Left: [0,2] has partial overlap with [2,4], Look in Left and Right both the Directions
Left: [0,1] does not has the overlap with the [2,4]
Return max
Right: [2,2] has a complete overlap with [2,4]
Return 4
Return Min (left, Right) i.e. 4
Right: [3,5] has partial overlap with [2,4], Look in Left and Right both the Directions
Left: [3,4] complete overlap with the [2,4]
Return 0
Right: [5,5] has a no overlap with [2,4]
Return max
Return Min (left, Right) i.e. 0

Return Min (left, Right) i.e. 0

Example 2: Querying for finding Minimum in the range [0,4]
Root node [0,5] has partial overlap with [0,4]; look in left and right both the directions

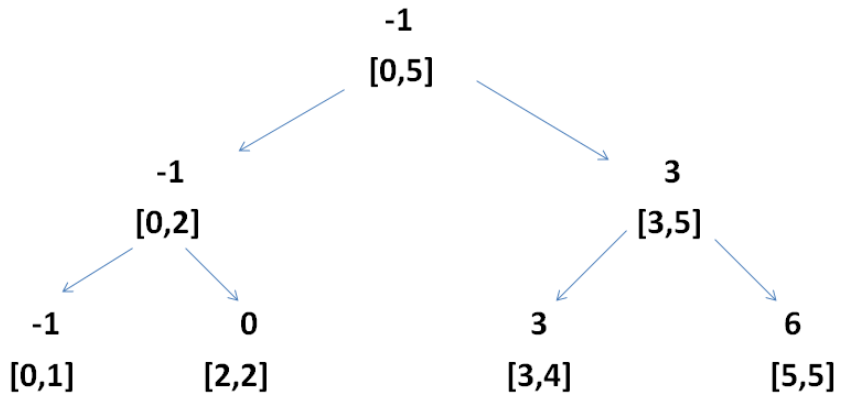
Left: [0,2] has a complete overlap with [0,4],
Return -1
Right: [3,5] has partial overlap with [0,4], Look in Left and Right both the Directions
Left: [3,4] complete overlap with the [0,4]
Return 0
Right: [5,5] has a no overlap with [2,4]
Return max
Return Min (left, Right) i.e. 0

Return Min (left, Right) i.e. -1

10.21.3 Complexity of Query Operations

4.Log_2N as maximum 4 directions are explored in querying

If the number of elements is 4, no of nodes in segment tree is 2x4-1 = 7



If the number of elements be 5, the number of nodes would be 9

If the number of elements be 6, the number of nodes would be 11

If the number of nodes is n, the number of nodes would be 2*n – 1

In case we implement the Segment tree using Array,

Left child of a node at index i will be at 2*i+1 index

Right child of a node at index i will be at 2*i+2 index

Parent of a node at index i will be at (i – 1) /2

10.21.4 Construction of Segment Tree

ALGORITHM ConstructSegTree(input[ ], ST[ ], low, high, pos)

```
BEGIN:
    IF low == high THEN
        ST[pos] = input[low]
        RETURN
    Mid=(low+high)/2
    ConstructSegTree(input,ST, low, mid, 2*pos+1)
    ConstructSegTree(input,ST, mid+1,high, 2*pos+2)
    ST[pos] = min(ST[2*pos+1], ST[2*pos+2])
END;
```

We consider that the elements in the ST array are initialized to a large number. Pos is initialized to 0, low as 0 and high (N-1). N is the number of elements in the original array.

ALGORITHM RangeQueryMin(ST[], qlow,qhigh,low,high,pos)

BEGIN:

```
IF qlow<=low AND qhigh>=high) THEN                    //Case1: Total overlap
        RETURN ST[pos]
IF qlow> high OR qhigh< low THEN                      // Case 2: No overlap
        RETURN Max
Mid = (low+high)/2                                    //Case 3: Partial Overlap
RETURN (Min(RangeQueryMin(ST,qlow,qhigh,low,high,2*pos+1) ,
                RangeQueryMin(ST,qlow,qhigh,low,high,2*pos+2)))
```

END;

The position is initialized to 0 in the first call to RangeQueryMin.

10.21.5 Sum Segment Tree

10.6 Sum-segment tree: To find the sum of numbers in a range