

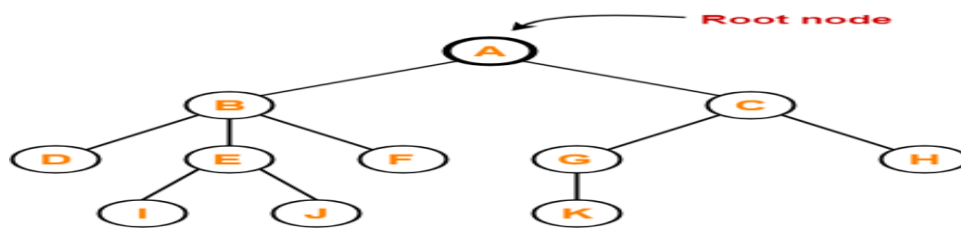
UNIT-IV TREES

Trees: Basic Terminology in Trees

Tree Properties:

The important properties of tree data structure are-

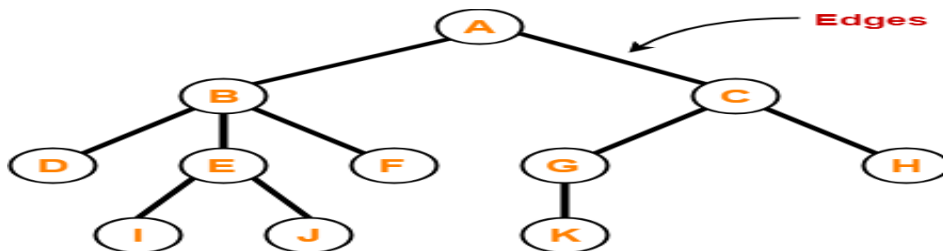
- There is one and only one path between every pair of vertices in a tree.
 - A tree with n vertices has exactly $(n-1)$ edges.
 - A graph is a tree if and only if it is minimally connected.
 - Any connected graph with n vertices and $(n-1)$ edges is a tree.
1. **Root:**
 - The first node from where the tree originates is called as a **root node**.
 - In any tree, there must be only one root node.
 - We can never have multiple root nodes in a tree data structure.



Here, node A is the only root node.

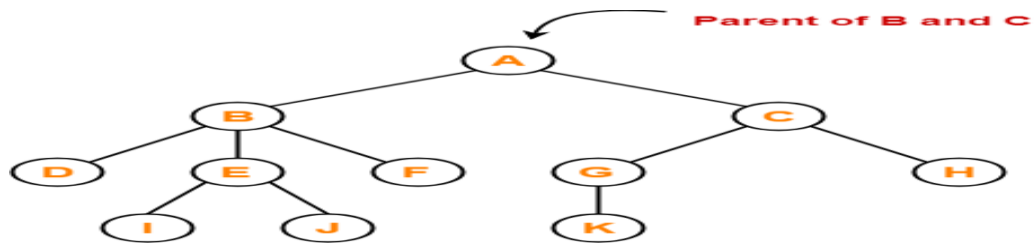
2. **Edge:**

- The connecting link between any two nodes is called as an **edge**.
- In a tree with n number of nodes, there are exactly $(n-1)$ number of edges.



3. **Parent:**

- The node which has a branch from it to any other node is called as a **parent node**.
- In other words, the node which has one or more children is called as a parent node.
- In a tree, a parent node can have any number of child nodes.

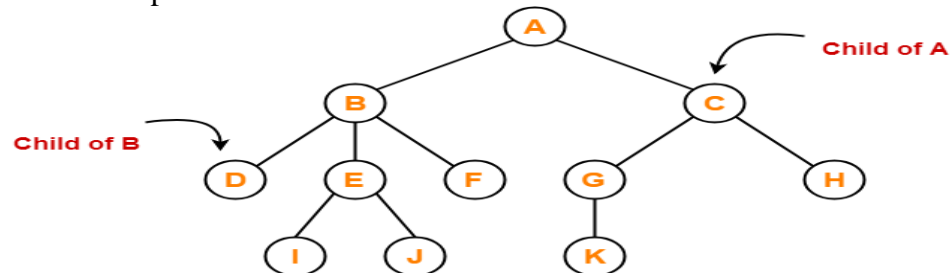


Here,

- Node A is the parent of nodes B and C
- Node B is the parent of nodes D, E and F
- Node C is the parent of nodes G and H
- Node E is the parent of nodes I and J
- Node G is the parent of node K

4. Child:

- The node which is a descendant of some node is called as a **child node**.
- All the nodes except root node are child nodes.

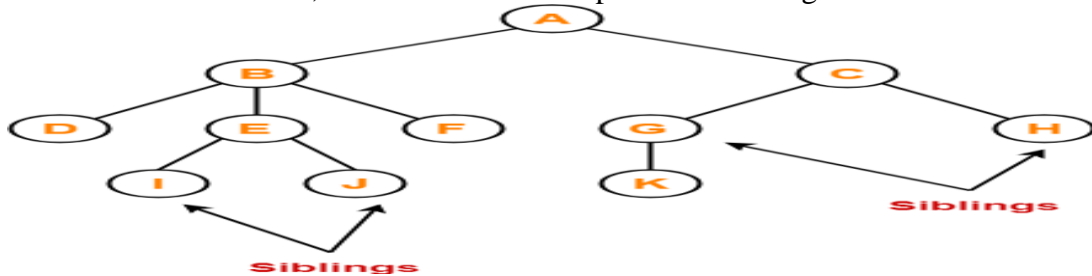


Here,

- Nodes B and C are the children of node A
- Nodes D, E and F are the children of node B
- Nodes G and H are the children of node C
- Nodes I and J are the children of node E
- Node K is the child of node G

5. Sibling:

- Nodes which belong to the same parent are called as siblings.
- In other words, nodes with the same parent are sibling nodes.



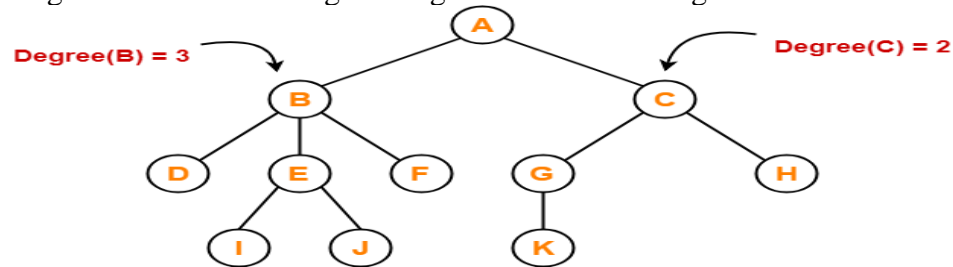
Here,

- Nodes B and C are siblings
- Nodes D, E and F are siblings
- Nodes G and H are siblings
- Nodes I and J are siblings

6. degree:

- Degree of a node is the total number of children of that node.

- Degree of a tree is the highest degree of a node among all the nodes in the tree.

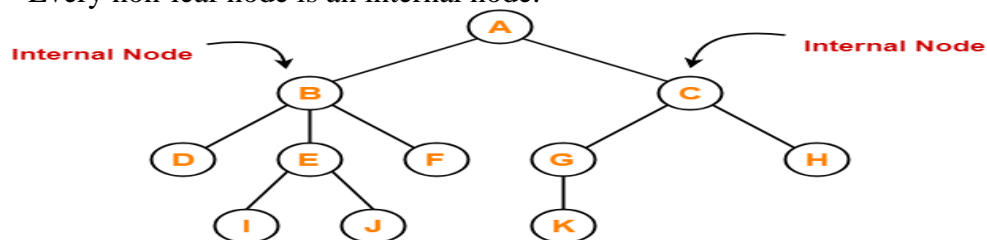


Here,

- Degree of node A = 2
- Degree of node B = 3
- Degree of node C = 2
- Degree of node D = 0
- Degree of node E = 2
- Degree of node F = 0
- Degree of node G = 1
- Degree of node H = 0
- Degree of node I = 0
- Degree of node J = 0
- Degree of node K = 0

7. Internal Node:

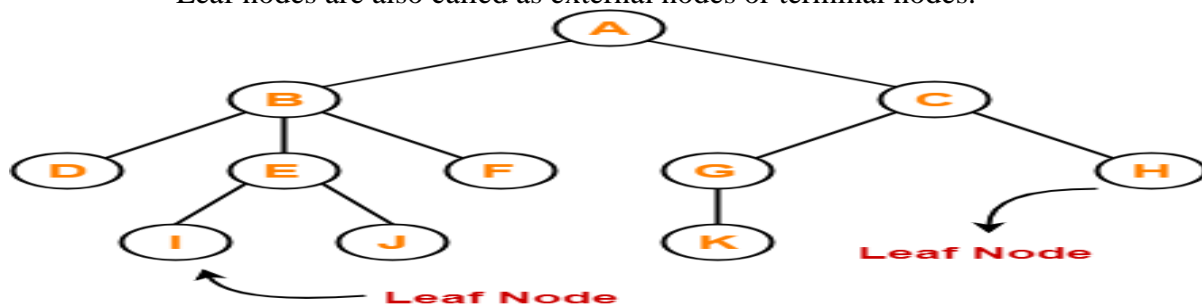
- The node which has at least one child is called as an internal node.
- Internal nodes are also called as non-terminal nodes.
- Every non-leaf node is an internal node.



Here, nodes A, B, C, E and G are internal nodes.

8. Leaf Node:

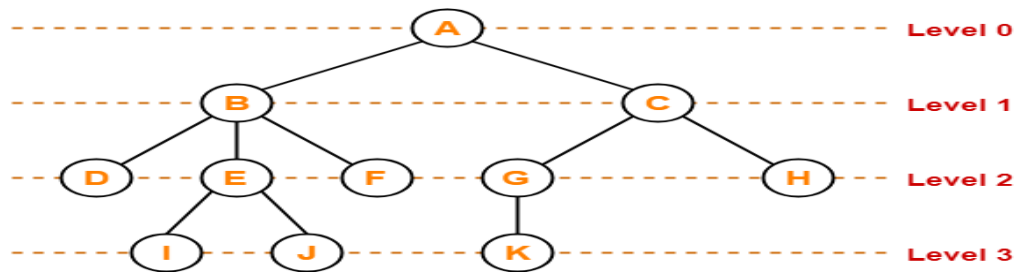
- The node which does not have any child is called as a leaf node.
- Leaf nodes are also called as external nodes or terminal nodes.



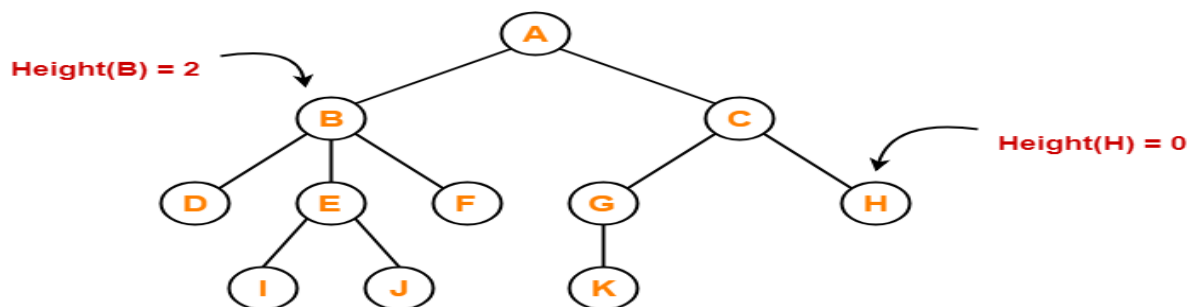
Here, nodes D, I, J, F, K and H are leaf nodes.

9. Level:

- In a tree, each step from top to bottom is called as **level of a tree**.
- The level count starts with 0 and increments by 1 at each level or step.

**10. height:**

- Total number of edges that lies on the longest path from any leaf node to a particular node is called as **height of that node**.
- **Height of a tree** is the height of root node.
- Height of all leaf nodes = 0

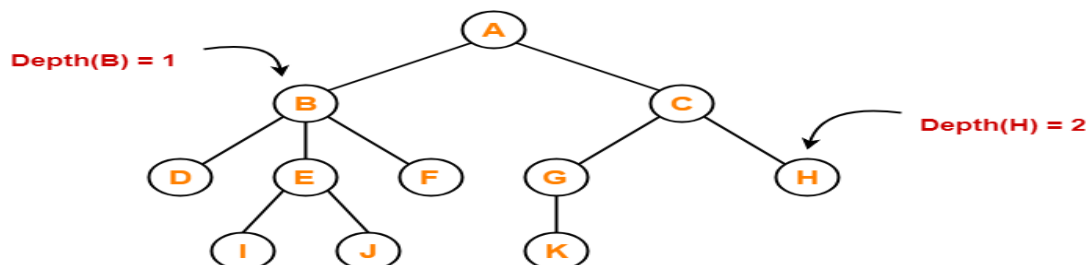


Here,

- | | |
|------------------------|----------------------|
| • Height of node A = 3 | Height of node B = 2 |
| • Height of node C = 2 | Height of node D = 0 |
| • Height of node E = 1 | Height of node F = 0 |
| • Height of node G = 1 | Height of node H = 0 |
| • Height of node I = 0 | Height of node J = 0 |
| • Height of node K = 0 | |

11. Depth:

- Total number of edges from root node to a particular node is called as **depth of that node**.
- **Depth of a tree** is the total number of edges from root node to a leaf node in the longest path.
- Depth of the root node = 0
- The terms “level” and “depth” are used interchangeably.

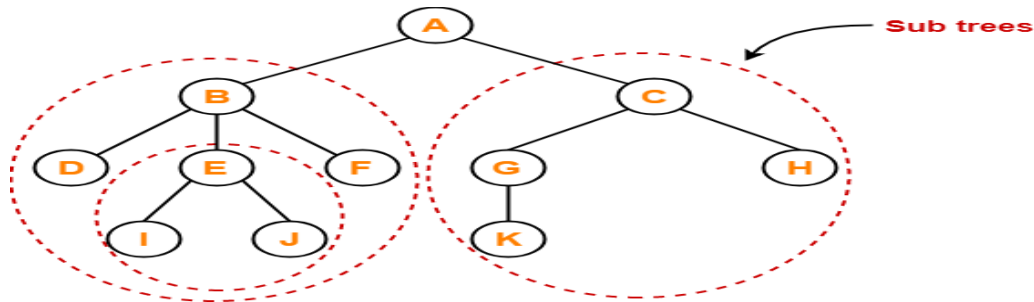


Here,

- Depth of node A = 0
- Depth of node B = 1
- Depth of node C = 1
- Depth of node D = 2
- Depth of node E = 2
- Depth of node F = 2
- Depth of node G = 2
- Depth of node H = 2
- Depth of node I = 3
- Depth of node J = 3
- Depth of node K = 3

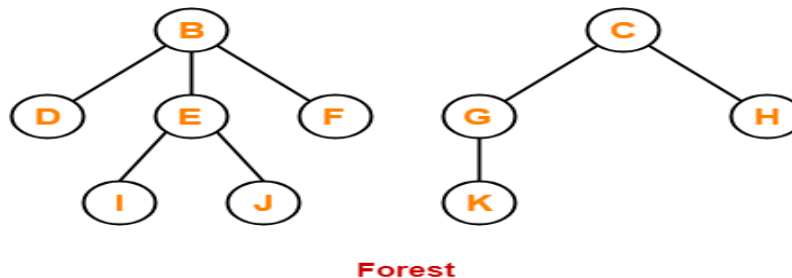
12. Subtree

- In a tree, each child from a node forms a **subtree** recursively.
- Every child node forms a subtree on its parent node.



13. Forest:

A forest is a set of disjoint trees.



Binary Trees-Properties

Binary Tree: A binary tree is a finite set of nodes that is either empty or consist of a root node and two disjoint binary trees called the left subtree and the right subtree.

In other words, a binary tree is a non-linear data structure in which each node has maximum of two child nodes. The tree connections can be called as branches.

Types of Binary Trees:

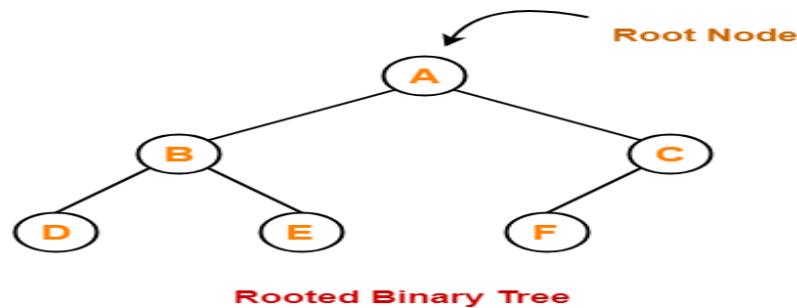
1. Rooted Binary Tree
2. Full / Strictly Binary Tree
3. Complete / Perfect Binary Tree

4. Almost Complete Binary Tree
5. Skewed Binary Tree

1. **Rooted Binary Tree**

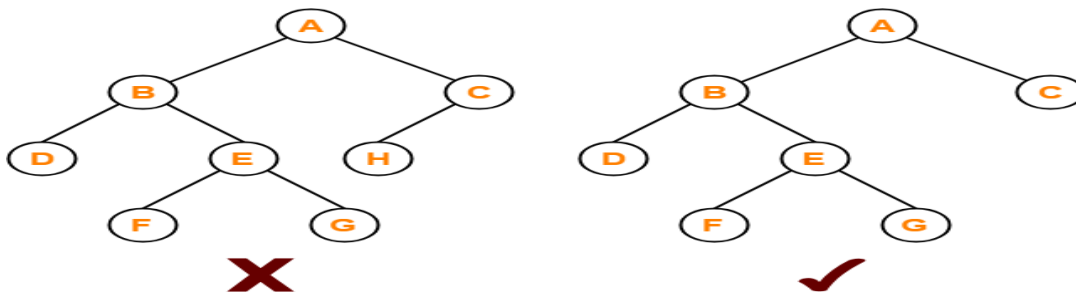
A rooted binary tree is a binary tree that satisfies the following 2 properties-

- It has a root node.
- Each node has at most 2 children.



2. **Full / Strictly Binary Tree**

- A binary tree in which every node has either 0 or 2 children is called as a Full binary tree.
- Full binary tree is also called as Strictly binary tree.



Here,

- First binary tree is not a full binary tree.
- This is because node C has only 1 child.

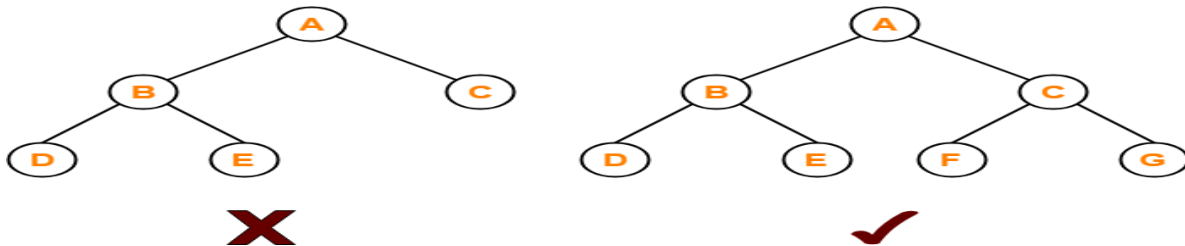
3. **Complete / Perfect Binary Tree**

A complete binary tree is a binary tree that satisfies the following 2 properties-

- Every internal node has exactly 2 children.

- All the leaf nodes are at the same level.

Complete binary tree is also called as Perfect binary tree.



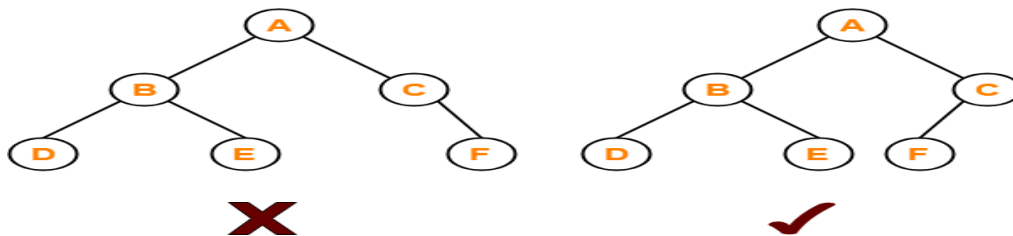
Here,

- First binary tree is not a complete binary tree.
- This is because all the leaf nodes are not at the same level.

4. Almost Complete Binary Tree

An almost complete binary tree is a binary tree that satisfies the following 2 properties-

- All the levels are completely filled except possibly the last level.
- The last level must be strictly filled from left to right.



Here,

- First binary tree is not an almost complete binary tree.
- This is because the last level is not filled from left to right.

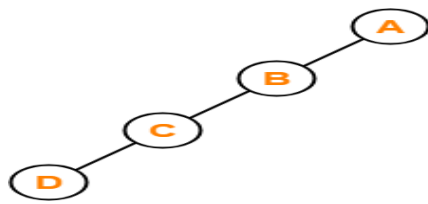
5. Skewed Binary Tree

A skewed binary tree is a binary tree that satisfies the following 2 properties-

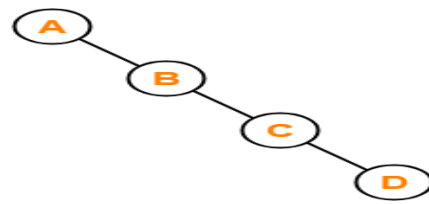
- All the nodes except one node has one and only one child.
- The remaining node has no child.

OR

A skewed binary tree is a binary tree of n nodes such that its depth is $(n-1)$.



Left Skewed Binary Tree



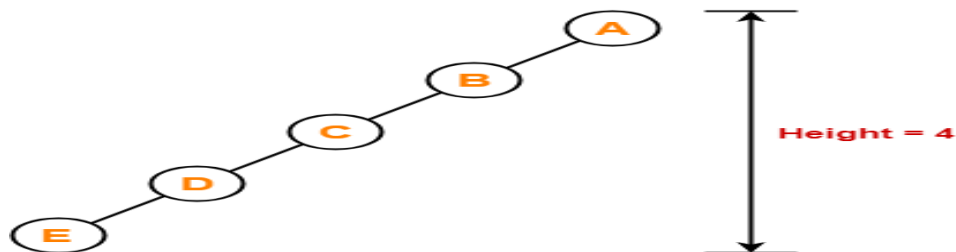
Right Skewed Binary Tree

Properties:

Important properties of binary trees are-

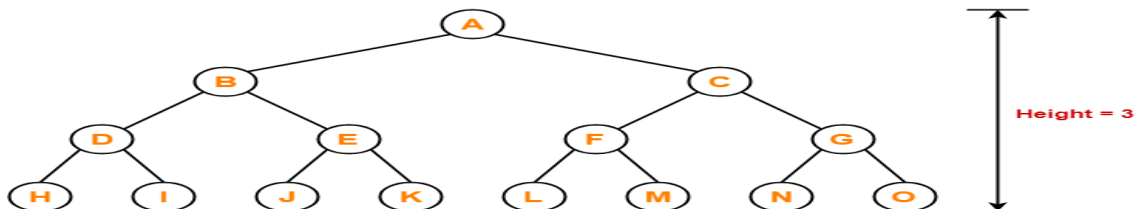
Property-01:

Minimum number of nodes in a binary tree of height $H = H + 1$
 To construct a binary tree of height = 4, we need at least $4 + 1 = 5$ nodes.



Property-02:

Maximum number of nodes in a binary tree of height $H = 2^{H+1} - 1$
 Maximum number of nodes in a binary tree of height 3 = $2^{3+1} - 1 = 16 - 1 = 15$ nodes
 Thus, in a binary tree of height = 3, maximum number of nodes that can be inserted = 15.

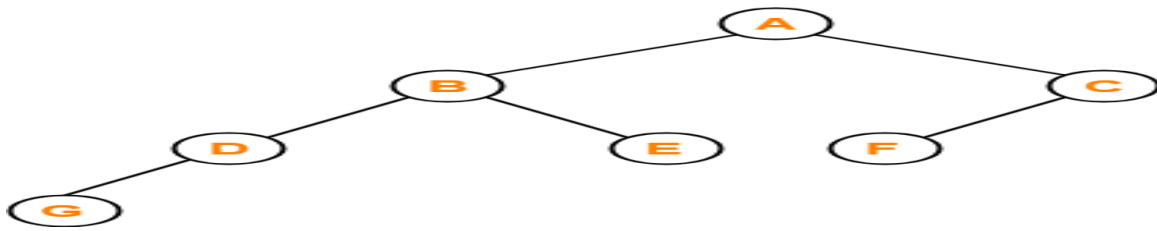


We can not insert more number of nodes in this binary tree.

Property-03:

Total Number of leaf nodes in a Binary Tree = Total Number of nodes with 2 children + 1

Consider the following binary tree-



Here,

- Number of leaf nodes = 3
- Number of nodes with 2 children = 2

Clearly, number of leaf nodes is one greater than number of nodes with 2 children.
This verifies the above relation.

NOTE:

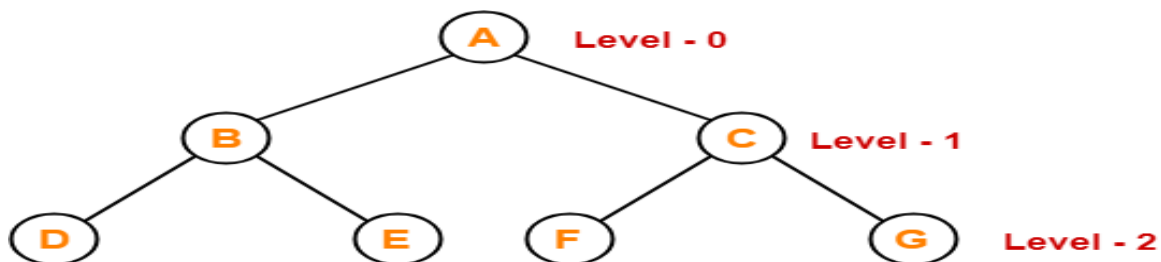
It is interesting to note that-Number of leaf nodes in any binary tree depends only on the number of nodes with 2 children.

Property-04:

Maximum number of nodes at any level 'L' in a binary tree = 2^L

Maximum number of nodes at level-2 in a binary tree = $2^2 = 4$

Thus, in a binary tree, maximum number of nodes that can be present at level-2 = 4.

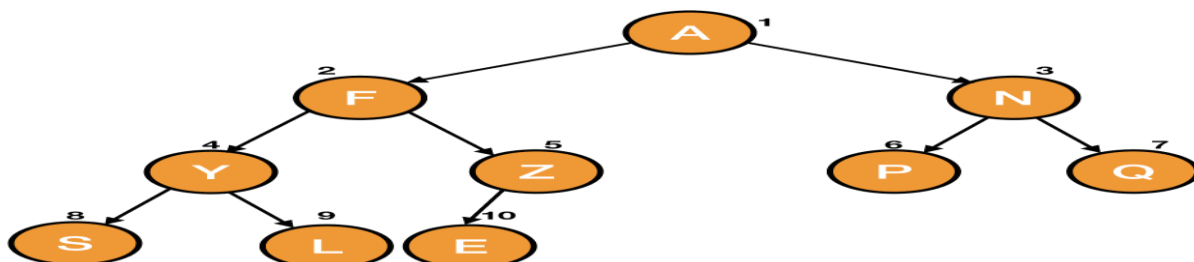


Representation of Binary Trees using Arrays and Linked lists.

Array Representation of Binary Tree:

In the previous chapter, we have already seen to make a node of a tree. We can easily use those nodes to make a linked representation of a binary tree. For now, let's discuss the array representation of a binary tree.

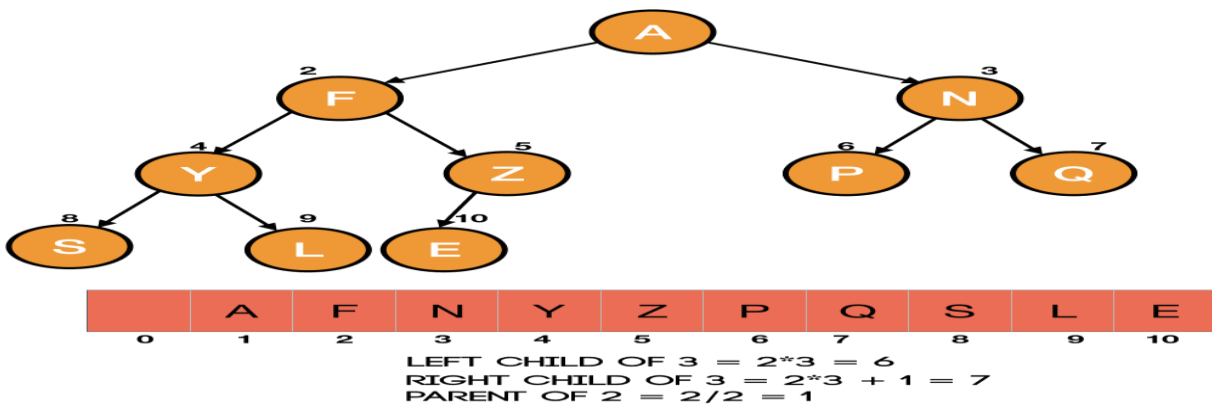
We start by numbering the nodes of the tree from 1 to n(number of nodes).



As you can see, we have numbered from top to bottom and left to right for the same level. Now, these numbers represent the indices of an array (starting from 1) as shown in the picture given below.



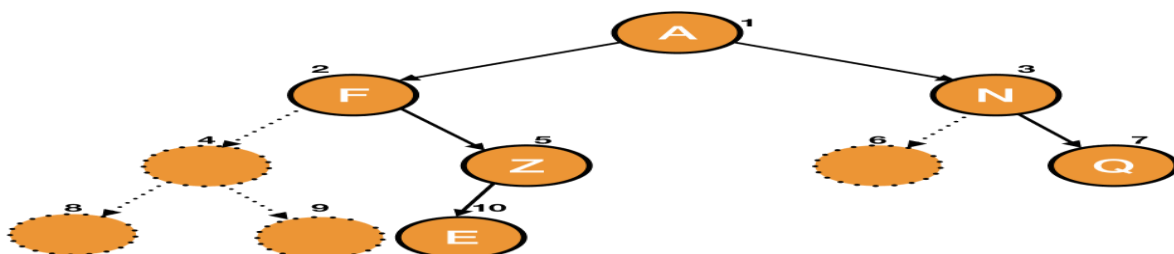
We can also get the parent, the right child and the left child using the properties of a complete binary tree we have discussed above i.e., for a node i , the parent is $\lfloor i/2 \rfloor$, the left child is $2i$ and the right child is $2i+1$.



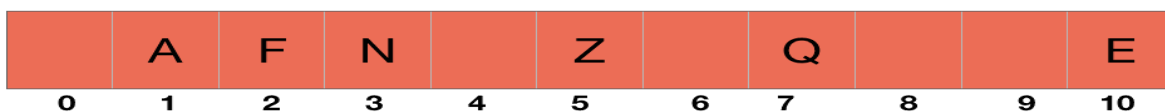
So, we represented a complete binary tree using an array and saw how to get the parent and children of any node. Let's discuss about doing the same for an incomplete binary tree.

Array Representation of Incomplete Binary Tree

To represent an incomplete binary tree with an array, we first assume that all the nodes are present to make it a complete binary tree and then number the nodes as shown in the picture given below.



Now according to these numbering, we fill up the array.

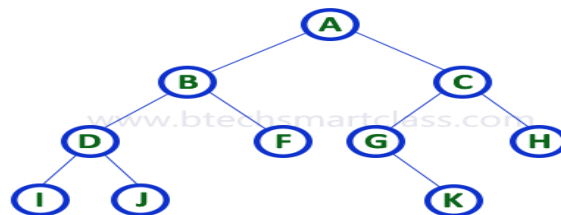


Linked list Representation of Binary Tree:

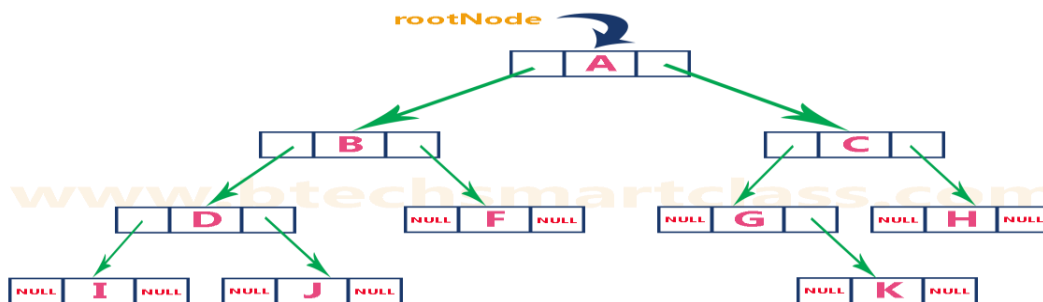
We use a double linked list to represent a binary tree. In a double linked list, every node consists of three fields. First field for storing left child address, second for storing actual data and third for storing right child address. In this linked list representation, a node has the following structure...



Consider the following binary tree...



Linked list Representation of Binary Tree



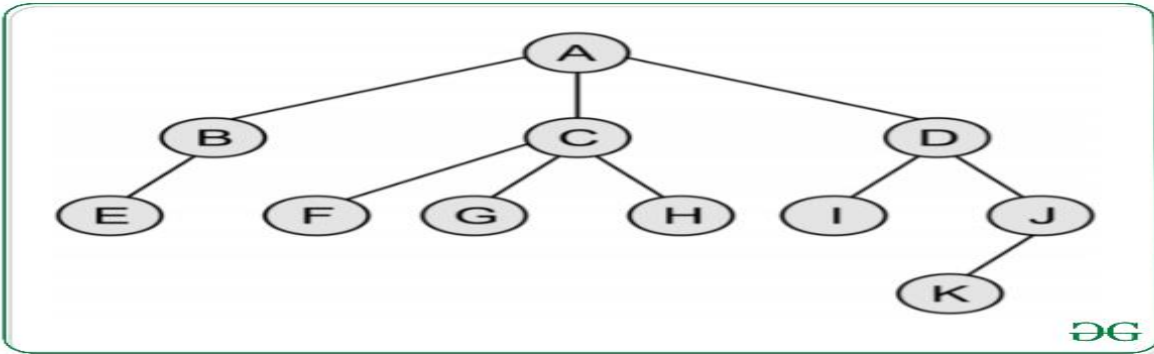
Generic(N-array Tree) to Binary Tree:

the rules to convert a **Generic(N-array Tree)** to **Binary Tree**:

- The root of the Binary Tree is the Root of the Generic Tree.
- The left child of a node in the Generic Tree is the Left child of that node in the Binary Tree.
- The right sibling of any node in the Generic Tree is the Right child of that node in the Binary Tree.

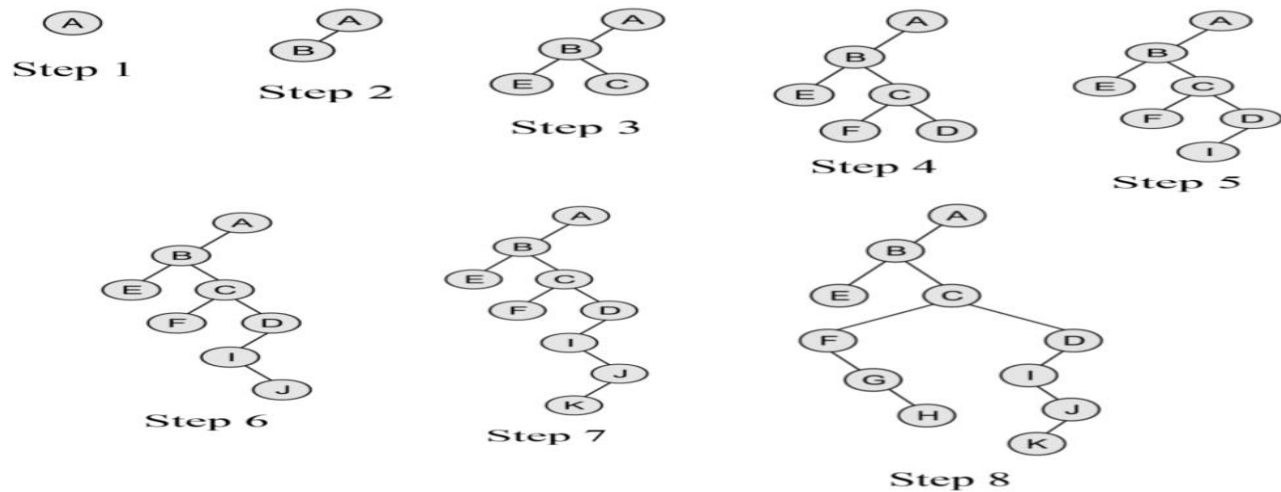
Examples:

Convert the following Generic Tree to Binary Tree:



Below is the Binary Tree of the above Generic Tree:

GeeksforGeeks
A computer science portal for geeks



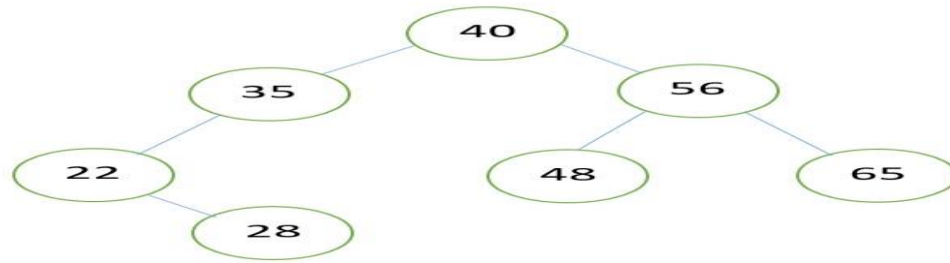
Note: If the parent node has only the right child in the general tree then it becomes the rightmost child node of the last node following the parent node in the binary tree. In the above example, if node **B** has the right child node **L** then in binary tree representation **L** would be the right child of node **D**.

Binary Search Trees- Basic Concepts:

Binary Search Tree (or BST) is a special kind of binary tree in which the values of all the nodes of the left subtree of any node of the tree are smaller than the value of the node. Also, the values of all the nodes of the right subtree of any node are greater than the value of the node.

A binary tree is shown for the element 40, 56, 35, 48, 22, 65, 28.

Following there is an example of binary search tree:



Creation of Binary Search Tree:

```

nodeptr create(nodeptr p)
{
    nodeptr temp;
    int k;
    temp=getnode();
    printf("\nEnter at end -999\n");
    printf("\nEnter the number:");
    scanf("%d",&k);
    temp->info=k;
    while(temp->info!=-999)
    {
        if(p==NULL)
            p=temp;
        else
        {
            printf("\nvisiting order:");
            insert(p,temp);
            temp=getnode();
            printf("\nEnter the number:");
            scanf("%d",&k);
            temp->info=k;
        }
    }
    return p;
}

void insert(nodeptr p, nodeptr temp)
{
    printf("%d\t",p->info);
    if((temp->info<p->info)&&(p->left==NULL))
        p->left=temp;
    else if((temp->info<p->info)&&(p->left!=NULL))
        insert(p->left,temp);
    else if((temp->info>p->info)&&(p->right==NULL))
        p->right=temp;
}
  
```

```
        else if((temp->info>p->info)&&(p->right!=NULL))
            insert(p->right,temp);
    }
```

Advantages of Binary Tree:

1. Searching in Binary tree become faster.
2. Binary tree provides three traversals.
3. One of three traversals give sorted order of elements.
4. Maximum and minimum elements can be directly picked up.
5. It is used for graph traversal and to convert an expression to postfix and prefix forms.

BST Operations: Insertion, Deletion, Tree Traversals

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

- 1. Traversals**
- 2. Search**
- 3. Insertion**
- 4. Deletion**

Tree Traversals:

1. In-order (Left, Root, Right)
2. Pre-order (Root, Left, Right)
3. Post-order (Left, Right, Root)

Algorithm Preorder

Until all nodes are traversed –

Step 1 – Visit root node.

Step 2 – Recursively traverse left subtree.

Step 3 – Recursively traverse right subtree.

Algorithm In-order

Until all nodes are traversed –

Step 1 – Recursively traverse left subtree.

Step 2 – Visit root node.

Step 3 – Recursively traverse right subtree.

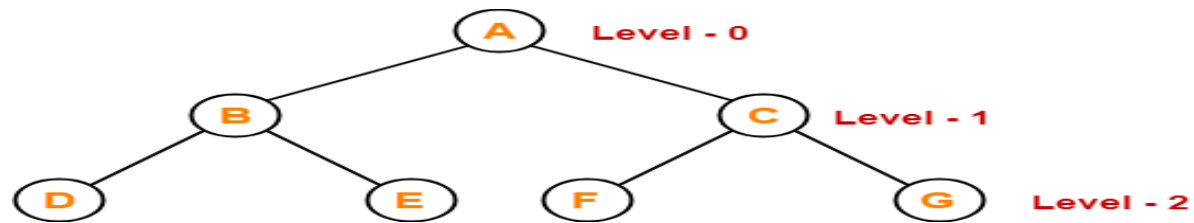
Algorithm Post-order

Until all nodes are traversed –

Step 1 – Recursively traverse left subtree.

Step 2 – Recursively traverse right subtree.

Step 3 – Visit root node.

**In-order (Left, Root, Right)**

We start from A, and following in-order traversal, we move to its left subtree B. B is also traversed in-order. The process goes on until all the nodes are visited.

The output of in-order traversal of this tree will be $D \rightarrow B \rightarrow E \rightarrow A \rightarrow F \rightarrow C \rightarrow G$

Post-order (Left, Right, Root)

We start from A, and following Post-order traversal, we first visit the left subtree B. B is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be $D \rightarrow E \rightarrow B \rightarrow F \rightarrow G \rightarrow C \rightarrow A$

Pre-order (Root, Left, Right)

We start from A, and following pre-order traversal, we first visit A itself and then move to its left subtree B. B is also traversed pre-order. The process goes on until all the nodes are visited.

The output of pre-order traversal of this tree will be $A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G$

/* Binary Search Tree TRAVERSALS */

```

#include<stdio.h>
struct node
{
    int info;
    struct node *left,*right;
};
typedef struct node *nodeptr;
nodeptr getnode();
nodeptr create(nodeptr);
void insert(nodeptr, nodeptr);
void inorder(nodeptr);
void preorder(nodeptr);
void postorder(nodeptr);
main()
{
    nodeptr tree;
    int ch;
    clrscr();
    tree=NULL;
    tree=create(tree);
    while(1)
    {

```

```

printf("\n*****\n\n\tMENU\n");
printf("\n*****\n\n1.in order\n2.pre order");
printf("\n3.post order\n4.exit\nenter your choice:");
scanf("%d",&ch);
switch(ch)
{
    case 1: printf("\nelements in in order is:\n");
            inorder(tree);
            break;
    case 2: printf("\nelements in pre order is:\n");
            preorder(tree);
            break;
    case 3: printf("\nelements in post order is:\n");
            postorder(tree);
            break;
    case 4: exit(0);
}
}
}
nodeptr getnode()
{
    nodeptr p;
    p=(nodeptr)malloc(sizeof(struct node));
    p->info=0;
    p->left=p->right=NULL;
    return p;
}
nodeptr create(nodeptr p)
{
    nodeptr temp;
    int k;
    temp=getnode();
    printf("\nEnter at end -999\n");
    printf("\nEnter the number:");
    scanf("%d",&k);
    temp->info=k;
    while(temp->info!=-999)
    {
        if(p==NULL)
            p=temp;
        else
        {
            printf("\nvisiting order:");
            insert(p,temp);
            temp=getnode();
            printf("\nenter the number:");

```



```

        scanf("%d",&k);
        temp->info=k;
    }
}
return p;
}
void insert(nodeptr p, nodeptr temp)
{
    printf("%d\t",p->info);
    if((temp->info<p->info)&&(p->left==NULL))
        p->left=temp;
    else if((temp->info<p->info)&&(p->left!=NULL))
        insert(p->left,temp);
    else if((temp->info>p->info)&&(p->right==NULL))
        p->right=temp;
    else if((temp->info>p->info)&&(p->right!=NULL))
        insert(p->right,temp);
}

void inorder(nodeptr p)
{
    if(p!=NULL)
    {
        inorder(p->left);
        printf("%d-->",p->info);
        inorder(p->right);
    }
}

void preorder(nodeptr p)
{
    if(p!=NULL)
    {
        printf("%d-->",p->info);
        preorder(p->left);
        preorder(p->right);
    }
}

void postorder(nodeptr p)
{
    if(p!=NULL)
    {
        postorder(p->left);
        postorder(p->right);
        printf("%d-->",p->info);
    }
}

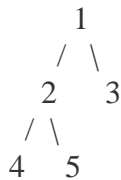
```

In-order Tree Traversal without Recursion

- a. Create an empty stack S.
- b. Initialize current node as root
- c. Push the current node to S and set current = current->left until current is NULL
- d. If current is NULL and stack is not empty then
 - i. Pop the top item from stack.
 - ii. Print the popped item, set current = popped_item->right
 - iii. Go to step 3.
- e. If current is NULL and stack is empty then we are done.

Example:

Let us consider the below tree for example



Step 1 Creates an empty stack: S = NULL

Step 2 sets current as address of root: current -> 1

Step 3 Pushes the current node and set current = current->left until current is NULL

current -> 1

push 1: Stack S -> 1

current -> 2

push 2: Stack S -> 2, 1

current -> 4

push 4: Stack S -> 4, 2, 1

current = NULL

Step 4 pops from S

a) Pop 4: Stack S -> 2, 1

b) print "4"

c) current = NULL /*right of 4 */ and go to step 3

Since current is NULL step 3 doesn't do anything.

Step 4 pops again.

a) Pop 2: Stack S -> 1

b) print "2"

c) current -> 5/*right of 2 */ and go to step 3

Step 3 pushes 5 to stack and makes current NULL

Stack S -> 5, 1

current = NULL

Step 4 pops from S

- a) Pop 5: Stack S -> 1
- b) print "5"
- c) current = NULL /*right of 5 */ and go to step 3

Since current is NULL step 3 doesn't do anything

Step 4 pops again.

- a) Pop 1: Stack S -> NULL
- b) print "1"
- c) current -> 3 /*right of 1 */

Step 3 pushes 3 to stack and makes current NULL

Stack S -> 3
current = NULL

Step 4 pops from S

- a) Pop 3: Stack S -> NULL
- b) print "3"
- c) current = NULL /*right of 3 */

Traversal is done now as stack S is empty and current is NULL.

```
struct tNode
```

```
{  
    int data;  
    struct tNode* left;  
    struct tNode* right;  
};
```

```
/* Structure of a stack node. Linked List implementation is used for  
   stack. A stack node contains a pointer to tree node and a pointer to  
   next stack node */
```

```
struct sNode
```

```
{  
    struct tNode *t;  
    struct sNode *next;  
};
```

```
/* Iterative function for inorder tree traversal */
```

```
void inOrder(struct tNode *root)
```

```
{  
    /* set current to root of binary tree */
```

```

struct tNode *current = root;
struct sNode *s = NULL; /* Initialize stack s */
bool done = 0;

while (!done)
{
    /* Reach the left most tNode of the current tNode */
    if(current != NULL)
    {
        push(&s, current);
        current = current->left;
    }

    /* backtrack from the empty subtree and visit the tNode
       at the top of the stack; however, if the stack is empty,
       you are done */
    else
    {
        if (!isEmpty(s))
        {
            current = pop(&s);
            printf("%d ", current->data);

            current = current->right;
        }
        else
            done = 1;
    }
} /* end of while */
}

/* UTILITY FUNCTIONS */
/* Function to push an item to sNode*/
void push(struct sNode** top_ref, struct tNode *t)
{
    /* allocate tNode */
    struct sNode* new_tNode =
        (struct sNode*) malloc(sizeof(struct sNode));

    if(new_tNode == NULL)
    {
        printf("Stack Overflow \n");
    }
}

```

```
    getchar();
    exit(0);
}

/* put in the data */
new_tNode->t = t;

/* link the old list off the new tNode */
new_tNode->next = (*top_ref);

/* move the head to point to the new tNode */
(*top_ref) = new_tNode;
}

/* The function returns true if stack is empty, otherwise false */
bool isEmpty(struct sNode *top)
{
    return (top == NULL)? 1 : 0;
}

/* Function to pop an item from stack*/
struct tNode *pop(struct sNode** top_ref)
{
    struct tNode *res;
    struct sNode *top;

    /*If sNode is empty then error */
    if(isEmpty(*top_ref))
    {
        printf("Stack Underflow \n");
        getchar();
        exit(0);
    }
    else
    {
        top = *top_ref;
        res = top->t;
        *top_ref = top->next;
        free(top);
        return res;
    }
}
```

```
/* Helper function that allocates a new tNode with the
   given data and NULL left and right pointers. */
```

```
struct tNode* newtNode(int data)
{
    struct tNode* tNode = (struct tNode*)
        malloc(sizeof(struct tNode));
    tNode->data = data;
    tNode->left = NULL;
    tNode->right = NULL;

    return(tNode);
}
```

```
/* Driver program to test above functions*/
```

```
int main()
{
```

```
    /* Constructed binary tree is
```

```
        1
       / \
      2   3
     / \
    4   5
```

```
    */
```

```
    struct tNode *root = newtNode(1);
    root->left      = newtNode(2);
    root->right     = newtNode(3);
    root->left->left = newtNode(4);
    root->left->right = newtNode(5);
```

```
    inOrder(root);
```

```
    getchar();
    return 0;
}
```

Search Operation in BST:

In a binary search tree, the search operation is performed with $O(\log n)$ time complexity. The search operation is performed as follows...

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

Searching for an element in BST:

```
int search(nodeptr p, int data)
```

```
{
    if(p==NULL)
        return -1;
    else
    {
        printf("%d\t",p->info);
        if(data<p->info)
            search(p->left,data);
        else if(data>p->info)
            search(p->right,data);
        else
            return 1;
    }
}
```

```
/* Searching an element in Binary Search Tree */
```

```
#include<stdio.h>
```

```
struct node
```

```
{
    int info;
    struct node *left,*right;
};
```

```
typedef struct node *nodeptr;
```

```
nodeptr getnode();
```

```
nodeptr create(nodeptr);
```

```

void insert(nodeptr, nodeptr);
int search(nodeptr,int);
main()
{
    nodeptr tree;
    int k,x;
    clrscr();
    tree=NULL;
    tree=create(tree);
    printf("\nEnter the searching element:");
    scanf("%d",&k);
    printf("\nSearch element sequence is:");
    x=search(tree,k);
    if(x==1)
        printf("\nSeatching element found in BST");
    else
        printf("\nSearching element not found in BST");
    getch();
}
nodeptr getnode()
{
    nodeptr p;
    p=(nodeptr)malloc(sizeof(struct node));
    p->info=0;
    p->left=p->right=NULL;
    return p;
}
nodeptr create(nodeptr p)
{
    nodeptr temp;
    int k;
    temp=getnode();
    printf("\nEnter at end -999\n");
    printf("\nEnter the number:");
    scanf("%d",&k);
    temp->info=k;
    while(temp->info!=-999)
    {
        if(p==NULL)
            p=temp;
        else
        {
            printf("\nvisiting order:");
            insert(p,temp);
            temp=getnode();
            printf("\nenter the number:");

```



```

        scanf("%d",&k);
        temp->info=k;
    }
}
return p;
}
void insert(nodeptr p, nodeptr temp)
{
    printf("%d\t",p->info);
    if((temp->info<p->info)&&(p->left==NULL))
        p->left=temp;
    else if((temp->info<p->info)&&(p->left!=NULL))
        insert(p->left,temp);
    else if((temp->info>p->info)&&(p->right==NULL))
        p->right=temp;
    else if((temp->info>p->info)&&(p->right!=NULL))
        insert(p->right,temp);
}
int search(nodeptr p, int data)
{
    if(p==NULL)
        return -1;
    else
    {
        printf("%d\t",p->info);
        if(data<p->info)
            search(p->left,data);
        else if(data>p->info)
            search(p->right,data);
        else
            return 1;
    }
}

```

Insertion Operation in BST

In a binary search tree, the insertion operation is performed with $O(\log n)$ time complexity. In binary search tree, new node is always inserted as a leaf node. The insertion operation is performed as follows...

- Step 1 - Create a new Node with given value and set its left and right to NULL.
- Step 2 - Check whether tree is Empty.
- Step 3 - If the tree is Empty, then set root to new Node.
- Step 4 - If the tree is Not Empty, then check whether the value of new Node is smaller or larger than the node (here it is root node).

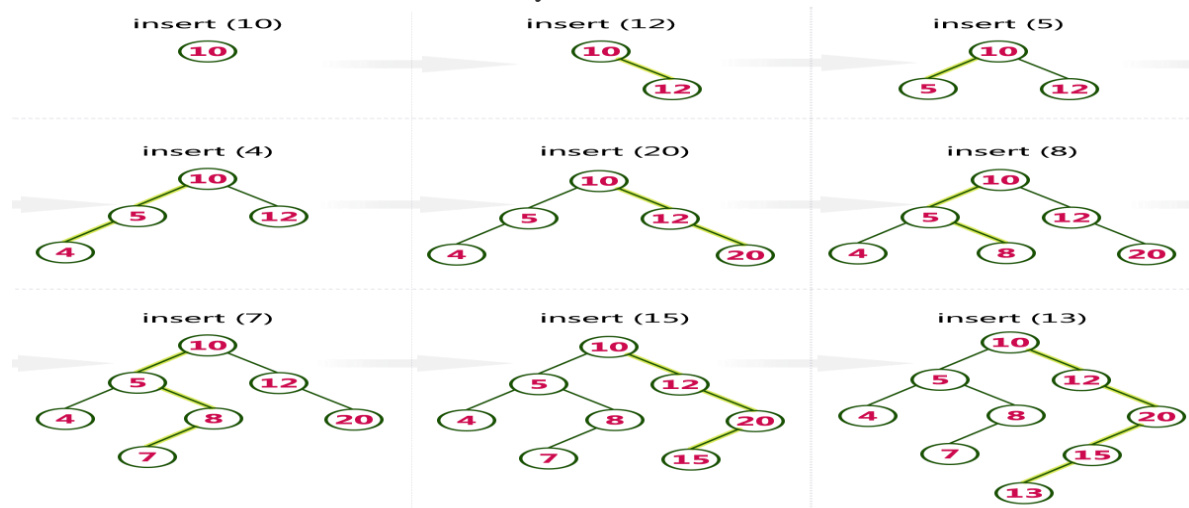
- Step 5 - If new Node is smaller than or equal to the node then move to its left child. If new Node is larger than the node then move to its right child.
- Step 6- Repeat the above steps until we reach to the leaf node (i.e., reaches to NULL).
- Step 7 - After reaching the leaf node, insert the new Node as left child if the new Node is smaller or equal to that leaf node or else insert it as right child.

Example

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

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

Above elements are inserted into a Binary Search Tree as follows...



/* insert an element into the Binary Search Tree */

/* insert an element into the Binary Search Tree */

```
#include<stdio.h>
```

```
struct node
```

```
{
```

```
    int info;
```

```
    struct node *left,*right;
```

```
};
```

```
typedef struct node *nodeptr;
```

```
nodeptr getnode();
```

```
nodeptr create(nodeptr);
```

```
void insert(nodeptr, nodeptr);
```

```
void inorder(nodeptr);
```

```
void preorder(nodeptr);
```

```
void postorder(nodeptr);
```

```
main()
```

```
{
```

```
    nodeptr tree,temp;
```

```
    int a,ch;
```

```
    clrscr();
```

```

tree=NULL;
tree=create(tree);
while(1)
{
    printf("\n*****\n\n\tMENU\n");
    printf("\n*****\n\n1.Insert an element\n2.in order\n3.pre order\n");
    printf("\n4.post order\n5.exit\nenter your choice:");
    scanf("%d",&ch);
    switch(ch)
    {
        case 1: printf("\nEnter the inserted element:");
                scanf("%d",&a);
                temp=getnode();
                temp->info=a;
                insert(tree,temp);
                break;
        case 2: printf("\nelements in in order is:\n");
                inorder(tree);
                break;
        case 3: printf("\nelements in pre order is:\n");
                preorder(tree);
                break;
        case 4: printf("\nelements in post order is:\n");
                postorder(tree);
                break;
        case 5: exit(0);
    }
}
}

nodeptr getnode()
{
    nodeptr p;
    p=(nodeptr)malloc(sizeof(struct node));
    p->info=0;
    p->left=p->right=NULL;
    return p;
}

nodeptr create(nodeptr p)
{
    nodeptr temp;
    int k;
    temp=getnode();
    printf("\nEnter at end -999\n");
    printf("\nEnter the number:");
    scanf("%d",&k);
    temp->info=k;

```

```

        while(temp->info!=-999)
        {
            if(p==NULL)
                p=temp;
            else
            {
                printf("\nvisiting order:");
                insert(p,temp);
                temp=getnode();
                printf("\nEnter the number:");
                scanf("%d",&k);
                temp->info=k;
            }
        }
        return p;
    }
}

void insert(nodeptr p, nodeptr temp)
{
    printf("%d\t",p->info);
    if((temp->info<p->info)&&(p->left==NULL))
        p->left=temp;
    else if((temp->info<p->info)&&(p->left!=NULL))
        insert(p->left,temp);
    else if((temp->info>p->info)&&(p->right==NULL))
        p->right=temp;
    else if((temp->info>p->info)&&(p->right!=NULL))
        insert(p->right,temp);
}

void inorder(nodeptr p)
{
    if(p!=NULL)
    {
        inorder(p->left);
        printf("%d-->",p->info);
        inorder(p->right);
    }
}

void preorder(nodeptr p)
{
    if(p!=NULL)
    {
        printf("%d-->",p->info);
        preorder(p->left);
        preorder(p->right);
    }
}

```

```

}
void postorder(nodeptr p)
{
    if(p!=NULL)
    {
        postorder(p->left);
        postorder(p->right);
        printf("%d-->",p->info);
    }
}

```

Deletion Operation in BST

In a binary search tree, the deletion operation is performed with $O(\log n)$ time complexity. Deleting a node from Binary search tree includes following three cases...

- Case 1: Deleting a Leaf node (A node with no children)
- Case 2: Deleting a node with one child
- Case 3: Deleting a node with two children

Case 1: Deleting a leaf node

We use the following steps to delete a leaf node from BST...

- Step 1 - Find the node to be deleted using search operation
- Step 2 - Delete the node using free function (If it is a leaf) and terminate the function.

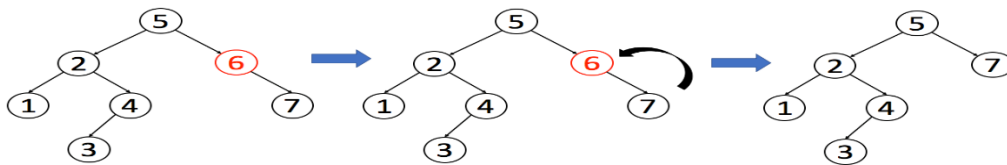


Case 2: Deleting a node with one child

We use the following steps to delete a node with one child from BST...

- Step 1 - Find the node to be deleted using search operation
- Step 2 - If it has only one child then create a link between its parent node and child node.
- Step 3 - Delete the node using free function and terminate the function.

Case 2: One Child

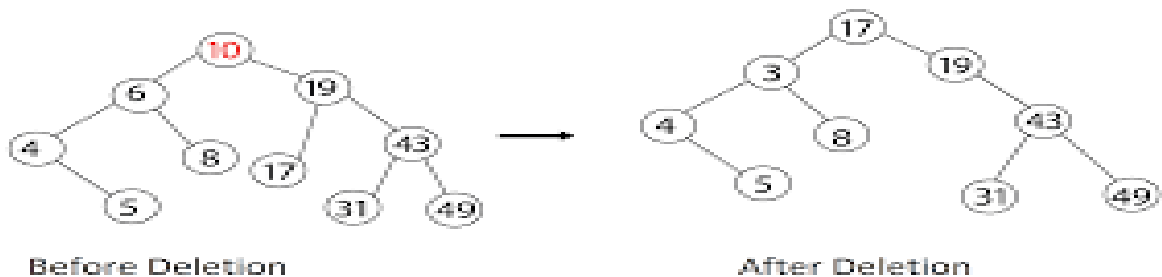


Case 3: Deleting a node with two children

We use the following steps to delete a node with two children from BST...

- Step 1 - Find the node to be deleted using search operation
- Step 2 - If it has two children, then find the largest node in its left subtree (OR) the smallest node in its right subtree.
- Step 3 - Swap both deleting node and node which is found in the above step.
- Step 4 - Then check whether deleting node came to case 1 or case 2 or else goto step 2
- Step 5 - If it comes to case 1, then delete using case 1 logic.
- Step 6 - If it comes to case 2, then delete using case 2 logic.
- Step 7 - Repeat the same process until the node is deleted from the tree.

Deletion — LF = 10
Node deleted to have 2 child.



/* delete an element from BST*/

```
#include<stdio.h>
struct node
{
    int info;
    struct node *left,*right;
};
typedef struct node *nodeptr;
nodeptr getnode();
nodeptr create(nodeptr);
```

```

void insert(nodeptr, nodeptr);
void del(nodeptr,int);
void inorder(nodeptr);
void preorder(nodeptr);
void postorder(nodeptr);
void dell(nodeptr, nodeptr);
void rightchild(nodeptr, nodeptr);
void nochild(nodeptr, nodeptr);
void leftchild(nodeptr, nodeptr);
void twochilds(nodeptr, nodeptr);
nodeptr findmin(nodeptr);
main()
{
    nodeptr tree;
    int a,ch;
    clrscr();
    tree=NULL;
    tree=create(tree);
    while(1)
    {
        printf("\n*****\n\n\tMENU\n");
        printf("\n*****\n\n1.delete an element\n2.in order\n3.pre order\n");
        printf("\n4.post order\n5.exit\nenter your choice:");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1: printf("\nEnter the element to be delete:");
                    scanf("%d",&a);
                    del(tree,a);
                    break;
            case 2: printf("\nelements in in order is:\n");
                    inorder(tree);
                    break;
            case 3: printf("\nelements in pre order is:\n");
                    preorder(tree);
                    break;
            case 4: printf("\nelements in post order is:\n");
                    postorder(tree);
                    break;
            case 5: exit(0);
        }
    }
}

```

```

nodeptr getnode()
{
    nodeptr p;
    p=(nodeptr)malloc(sizeof(struct node));
    p->info=0;
    p->left=p->right=NULL;
    return p;
}

nodeptr create(nodeptr p)
{
    nodeptr temp;
    int k;
    temp=getnode();
    printf("\nEnter at end -999\n");
    printf("\nEnter the number:");
    scanf("%d",&k);
    temp->info=k;
    while(temp->info!=-999)
    {
        if(p==NULL)
            p=temp;
        else
        {
            printf("\nvisiting order:");
            insert(p,temp);
            temp=getnode();
            printf("\nEnter the number:");
            scanf("%d",&k);
            temp->info=k;
        }
    }
    return p;
}

void insert(nodeptr p, nodeptr temp)
{
    printf("%d\t",p->info);
    if((temp->info<p->info)&&(p->left==NULL))
        p->left=temp;
    else if((temp->info<p->info)&&(p->left!=NULL))
        insert(p->left,temp);
    else if((temp->info>p->info)&&(p->right==NULL))
        p->right=temp;
    else if((temp->info>p->info)&&(p->right!=NULL))

```



```
        insert(p->right,temp);
    }

void inorder(nodeptr p)
{
    if(p!=NULL)
    {
        inorder(p->left);
        printf("%d-->",p->info);
        inorder(p->right);
    }
}

void preorder(nodeptr p)
{
    if(p!=NULL)
    {
        printf("%d-->",p->info);
        preorder(p->left);
        preorder(p->right);
    }
}

void postorder(nodeptr p)
{
    if(p!=NULL)
    {
        postorder(p->left);
        postorder(p->right);
        printf("%d-->",p->info);
    }
}

void del(nodeptr p, int a)
{
    nodeptr p1;
    p1=p;
    if(p==NULL)
        printf("\nElement not found in the BST");
    else
    {
        while(p->info!=a)
        {
            if(a<p->info)
            {
                p1=p;
            }
        }
    }
}
```

```

        p=p->left;
    }
    else if (a>p->info)
    {
        p1=p;
        p=p->right;
    }
}
if(p==NULL)
    printf("\nElement not found in BST");
else if(a==p->info)
    del1(p1,p);
}

}
void del1(nodeptr p1, nodeptr p)
{
    if((p->left==NULL)&&(p->right==NULL))
        nochild(p1,p);
    else if((p->left==NULL)&&(p->right!=NULL))
        rightchild(p1,p);
    else if ((p->left!=NULL)&&(p->right==NULL))
        leftchild(p1,p);
    else if((p->left!=NULL)&&(p->right!=NULL))
        twochilds(p1,p);
}
void nochild(nodeptr p1, nodeptr p)
{
    if(p1->left==p)
        p1->left=NULL;
    else
        p1->right=NULL;
    free(p);
}
void leftchild(nodeptr p1, nodeptr p)
{
    if(p1->left==p)
        p1->left=p->left;
    else
        p1->right=p->left;
    p->left=NULL;
    free(p);
}

```

```

}
void rightchild(nodeptr p1, nodeptr p)
{
    if(p1->left==p)
        p1->left=p->right;
    else
        p1->right=p->right;
    p->right=NULL;
    free(p);
}
void twochilds(nodeptr p1, nodeptr p)
{
    int k;
    nodeptr temp;
    temp=findmin(p->right);
    k=p->info;
    p->info=temp->info;
    temp->info=k;
    del(p1->right, temp->info);
}
nodeptr findmin(nodeptr p)
{
    while(p->left!=NULL)
        p=p->left;
    return p;
}

```

Applications-Expression Trees

Find a height of a binary tree:

- The height of a node in a binary tree is the largest number of edges in a path from a leaf node to a target node.
- If the target node doesn't have any other nodes connected to it, the height of that node would be 0.
- The height of a binary tree is the height of the root node in the whole binary tree. In other words, the height of a binary tree is equal to the largest number of edges from the root to the most distant leaf node.

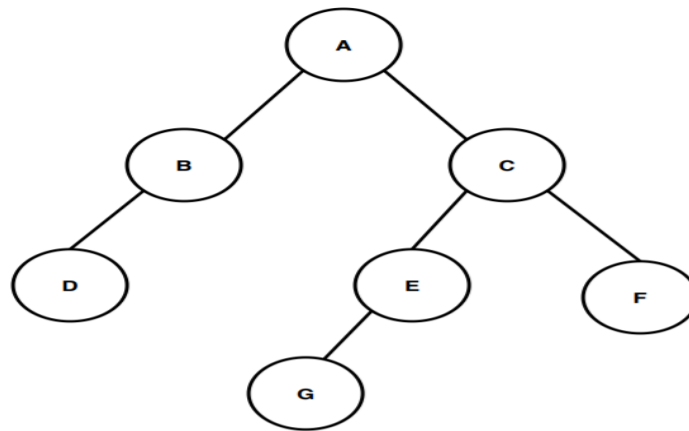
Depth of the tree.

- The depth of a node in a binary tree is the total number of edges from the root node to the target node.

- Similarly, the depth of a binary tree is the total number of edges from the root node to the most distant leaf node.

Note: One important observation here is that when we calculate the depth of a whole binary tree, it's equivalent to the height of the binary tree.

Example



First, we'll calculate the height of node C. So, according to the definition, the height of node C is the largest number of edges in a path from the leaf node to node C. We can see that there are two paths for node C: $C \rightarrow E \rightarrow F$, and $C \rightarrow F$. The largest number of edges among these two paths would be 2; hence, the height of node C is 2.

Now we'll calculate the height of the binary tree. From the root, we can have three different paths leading to the leaf nodes: $A \rightarrow C \rightarrow F$, $A \rightarrow B \rightarrow D$, and $A \rightarrow C \rightarrow E \rightarrow G$. Among these three paths, the path $A \rightarrow C \rightarrow E \rightarrow G$ contains the largest number of edges, which is 3.

Therefore, the height of the tree is 3.

Next, we want to find the depth of node B. We can see that from the root, there's only one path to node B, and it has one edge. Thus, the depth of node B is 1.

As we previously mentioned, the depth of a binary tree is equal to the height of the tree.

Therefore, the depth of the binary tree is 3.

Algorithm: find the height of a binary tree:

Algorithm BTHHeight(root)

```

{
    if root == NULL then
        return 0;
    else
    {

```

```

        leftHeight = BTHHeight(root → left);
        rightHeight = BTHHeight(root → right);
        return max(leftHeight, rightHeight) + 1;
    }
}

```

We start the algorithm by taking the root node as an input. Next, we calculate the height of the left and right child nodes of the root. If the root doesn't have any child nodes, we return the height of the tree as 0.

Then we recursively call all the nodes from the left and right subtree of the root node to calculate the height of the binary tree. Finally, once we calculate the height of the left and right subtree, we take the maximum height among both and add one to it. The number returned by the algorithm would be the height of the binary tree.

Time Complexity Analysis

As a best case scenario, we would have only one node in the binary tree. In such a case, we would only execute the first condition of the algorithm when the root is null and return the height of the tree as 0. Here, the time complexity would be $O(1)$.

Let's suppose that the number of nodes in a binary tree is n .

Therefore, the time complexity would be $O(n)$.

Count the leaf nodes in Binary search tree:

getLeafCount(node)

- 1) if node is NULL then return 0.
- 2) else If left and right child nodes are NULL return 1.
- 3) else recursively calculate leaf count of the tree using below formula.

Leaf count of a tree = Leaf count of left subtree + Leaf count of right subtree

Algorithm

Algorithm LeafCountTree(root)

```

{
    if (root == NULL)
        return 0;
    else if ((root → left == NULL) && (root → right == NULL))
        return 1;
    else
        return LeafCountTree(root → left) + LeafCountTree(root → right);
}

```

Construction of Expression Tree

Let us consider a **postfix expression** is given as an input for constructing an expression tree. Following are the steps to construct an expression tree:

1. Read one symbol at a time from the postfix expression.
2. Check if the symbol is an operand or operator.
3. If the symbol is an operand, create a one node tree and push a pointer onto a stack
4. If the symbol is an operator, pop two pointers from the stack namely T_1 & T_2 and form a new tree with root as the operator, T_1 & T_2 as a left and right child
5. A pointer to this new tree is pushed onto the stack

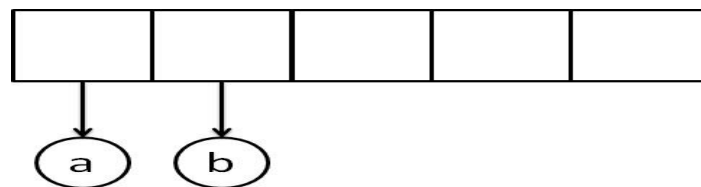
Thus, an expression is created or constructed by reading the symbols or numbers from the left. If operand, create a node. If operator, create a tree with operator as root and two pointers to left and right subtree.

Example - Postfix Expression Construction

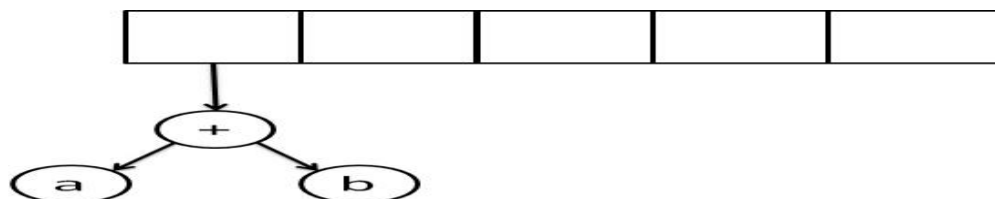
The input is:

$a\ b\ +\ c\ *$

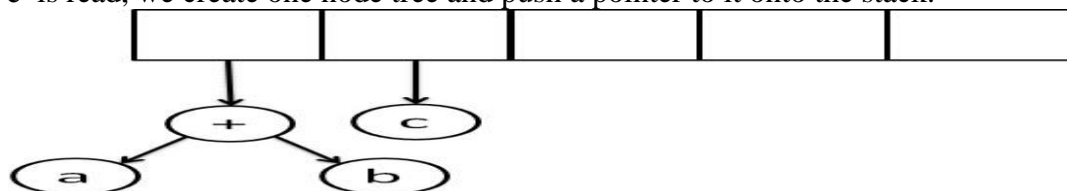
The first two symbols are operands, we create one-node tree and push a pointer to them onto the stack.



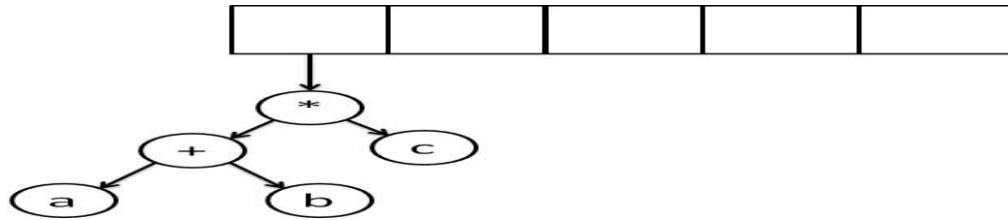
Next, read a '+' symbol, so two pointers to tree are popped, a new tree is formed and push a pointer to it onto the stack.



Next, 'c' is read, we create one node tree and push a pointer to it onto the stack.

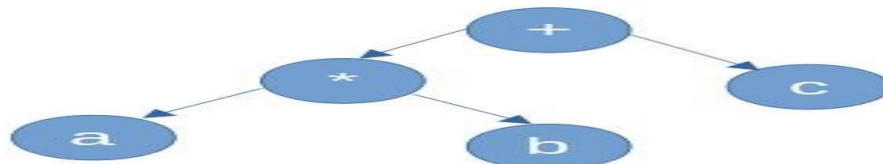


Finally, the last symbol is read ' * ', we pop two tree pointers and form a new tree with a, ' * ' as root, and a pointer to the final tree remains on the stack.



Expression Tree is used to represent expressions. Let us look at some examples of prefix, infix and postfix expressions from expression tree for 3 of the expressions:

- $a*b+c$
- $a+b*c+d$
- $a+b-c*d+e*f$



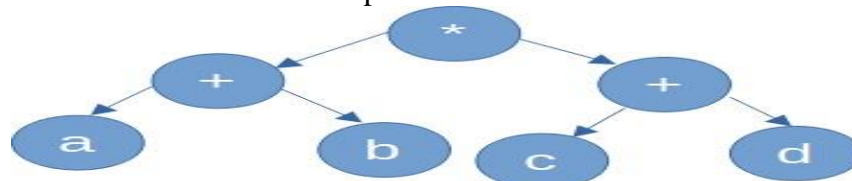
Expression Tree for $a*b+c$

Expressions from Expression Tree

Infix expression	$a * b + c$
Prefix expression	$+ * a b c$
Postfix expression	$a b * c +$

Infix, Prefix and Postfix Expressions from Expression Tree for $(a+b)*(c+d)$

Expression Tree for $a + b * c + d$ can be represented as:



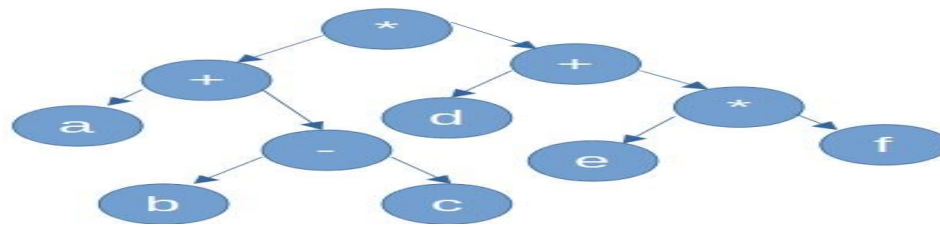
Expression Tree for $a + b * c + d$

Expressions for the binary expression tree above can be written as

Infix expression	$a + b * c + d$
Prefix expression	$* + a b + c d$
Postfix expression	$a b + c d + *$

Infix, Prefix and Postfix Expressions from Expression Tree for $(a+(b-c))*(d+(e*f))$

Expression Tree for $(a + (b - c)) * (d + (e * f))$ can be represented as:



Expression Tree

Expressions for the binary expression tree above can be written as

Infix expression $a + b - c * d + e * f$

Prefix expression $* + a - b c + d * e f$

Postfix expression $a b c - + d e f * + *$

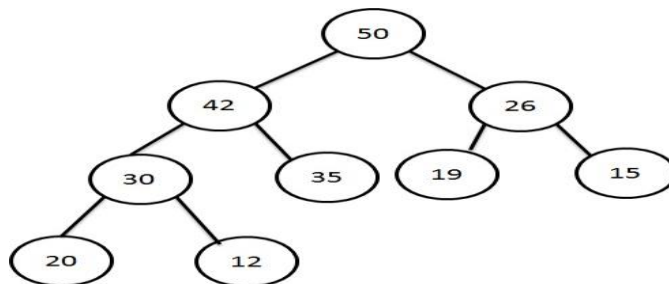
Heap Order Property

Heap order Property that allows operations to be performed quickly. In a heap, the key value of each node is smaller than or greater than the key value of parent with exception of the root because the root has no parent. Here there are two types of Heaps:

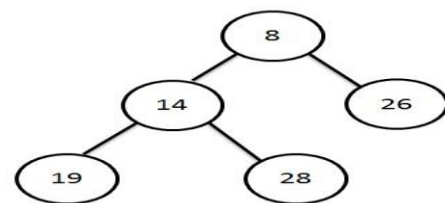
1. Max-Heap
2. Min-Heap

Max-Heap

For every node X, the key value in the parent of X is greater than the key value in the X (except root because root has no parent).



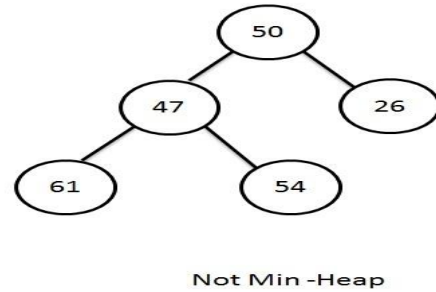
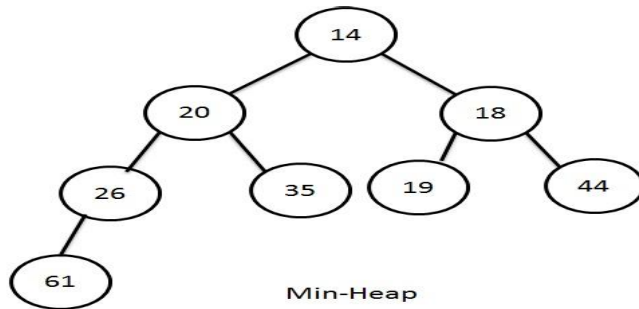
Max-Heap



Not Max-Heap

Min-Heap

For every node X, the key value in the parent of X is smaller than the key value in the X (except root because root has no parent).



Insertion:

Construction of Max Heap:

Input 35 33 42 10 14 19 27 44 26 31

Deletion:

Step 1 – Remove root node.

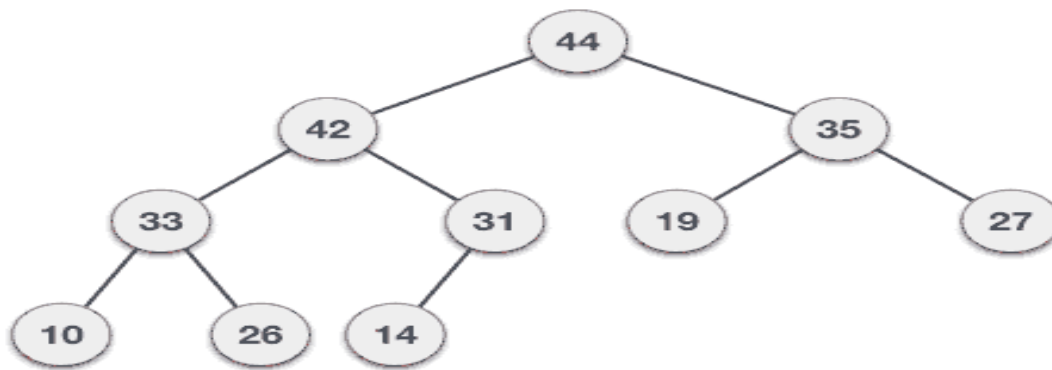
Step 2 – Move the last element of last level to root.

Step 3 – Compare the value of this child node with its parent.

Step 4 – If value of parent is less than child, then swap them.

Step 5 – Repeat step 3 & 4 until Heap property holds.

Delete the node 44:



Heap Sort

Working of Heap sort Algorithm

Now, let's see the working of the Heapsort Algorithm.

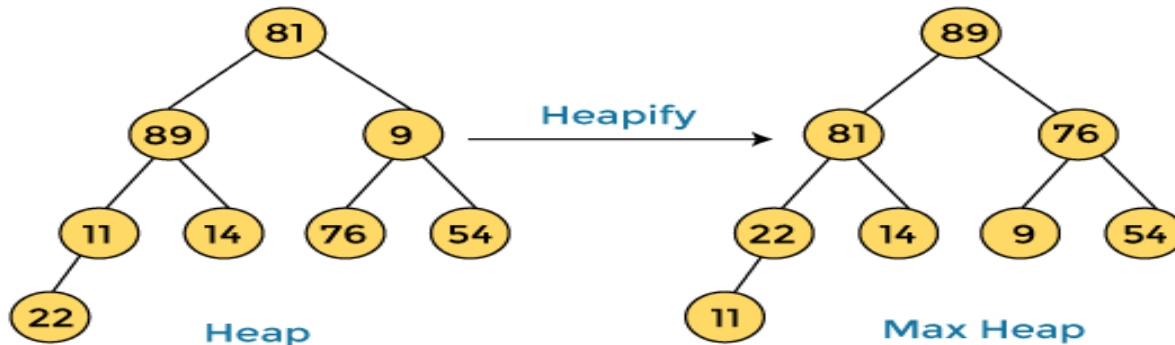
In heap sort, basically, there are two phases involved in the sorting of elements. By using the heap sort algorithm, they are as follows -

- The first step includes the creation of a heap by adjusting the elements of the array.
- After the creation of heap, now remove the root element of the heap repeatedly by shifting it to the end of the array, and then store the heap structure with the remaining elements.

Now let's see the working of heap sort in detail by using an example. To understand it more clearly, let's take an unsorted array and try to sort it using heap sort. It will make the explanation clearer and easier.

81	89	9	11	14	76	54	22
----	----	---	----	----	----	----	----

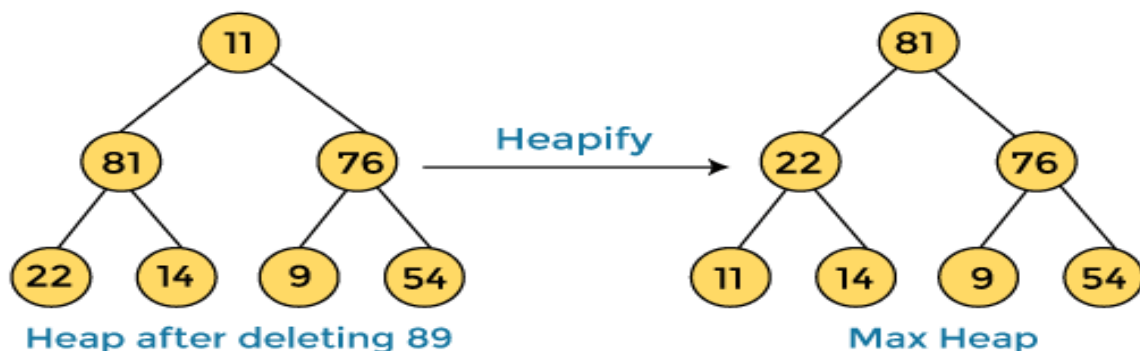
First, we have to construct a heap from the given array and convert it into max heap.



After converting the given heap into max heap, the array elements are -

89	81	76	22	14	9	54	11
----	----	----	----	----	---	----	----

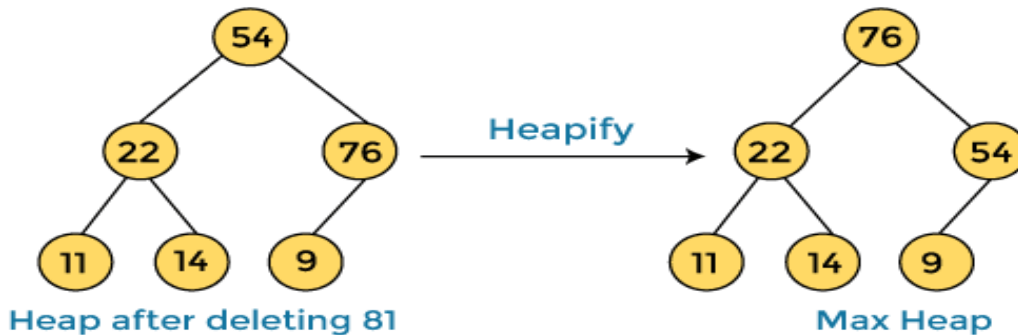
Next, we have to delete the root element (89) from the max heap. To delete this node, we have to swap it with the last node, i.e. (11). After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element **89** with **11**, and converting the heap into max-heap, the elements of array are -

81	22	76	11	14	9	54	89
----	----	----	----	----	---	----	----

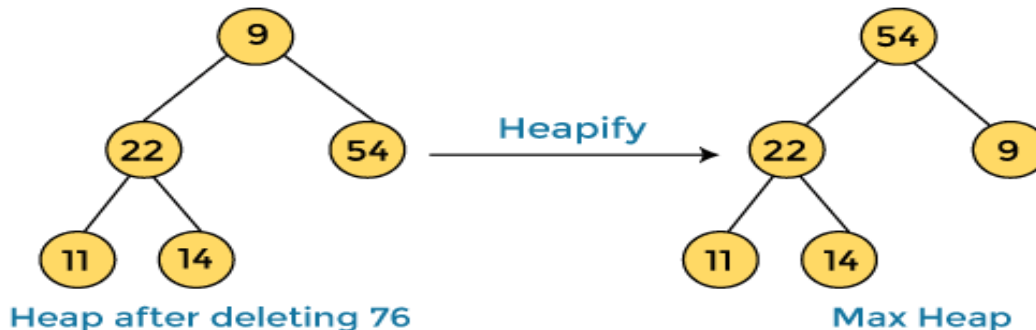
In the next step, again, we have to delete the root element (**81**) from the max heap. To delete this node, we have to swap it with the last node, i.e. (**54**). After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element **81** with **54** and converting the heap into max-heap, the elements of array are -

76	22	54	11	14	9	81	89
----	----	----	----	----	---	----	----

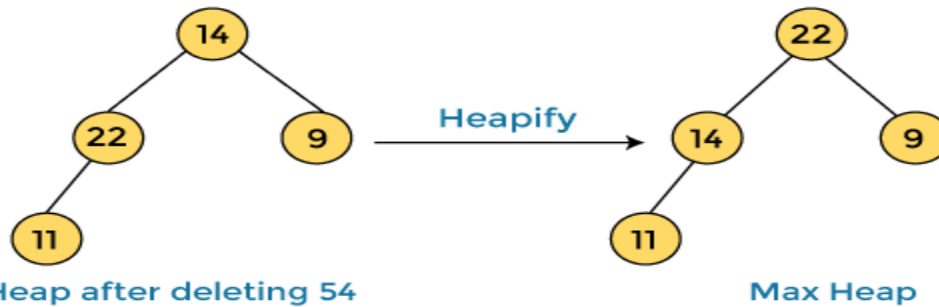
In the next step, we have to delete the root element (**76**) from the max heap again. To delete this node, we have to swap it with the last node, i.e. (**9**). After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element **76** with **9** and converting the heap into max-heap, the elements of array are -

54	22	9	11	14	76	81	89
----	----	---	----	----	----	----	----

In the next step, again we have to delete the root element (**54**) from the max heap. To delete this node, we have to swap it with the last node, i.e. (**14**). After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element **54** with **14** and converting the heap into max-heap, the elements of array are -

22	14	9	11	54	76	81	89
----	----	---	----	----	----	----	----

In the next step, again we have to delete the root element (**22**) from the max heap. To delete this node, we have to swap it with the last node, i.e. (**11**). After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element **22** with **11** and converting the heap into max-heap, the elements of array are -

14	11	9	22	54	76	81	89
----	----	---	----	----	----	----	----

In the next step, again we have to delete the root element (**14**) from the max heap. To delete this node, we have to swap it with the last node, i.e. (**9**). After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element **14** with **9** and converting the heap into max-heap, the elements of array are -

11	9	14	22	54	76	81	89
----	---	----	----	----	----	----	----

In the next step, again we have to delete the root element (**11**) from the max heap. To delete this node, we have to swap it with the last node, i.e. (**9**). After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element **11** with **9**, the elements of array are -

9	11	14	22	54	76	81	89
---	----	----	----	----	----	----	----

Now, heap has only one element left. After deleting it, heap will be empty.



After completion of sorting, the array elements are -

9	11	14	22	54	76	81	89
---	----	----	----	----	----	----	----

Now, the array is completely sorted.

Heap sort complexity

Now, let's see the time complexity of Heap sort in the best case, average case, and worst case. We will also see the space complexity of Heapsort.

1. Time Complexity

Case	Time Complexity
Best Case	$O(n \log n)$
Average Case	$O(n \log n)$
Worst Case	$O(n \log n)$

Consider the following list of unsorted numbers which are to be sort using Heap Sort

82, 90, 10, 12, 15, 77, 55, 23

Step 1 - Construct a Heap with given list of unsorted numbers and convert to Max Heap



list of numbers after heap converted to Max Heap

90, 82, 77, 23, 15, 10, 55, 12

Step 2 - Delete root (90) from the Max Heap. To delete root node it needs to be swapped with last node (12). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 90 with 12.

12, 82, 77, 23, 15, 10, 55, 90

Step 3 - Delete root (82) from the Max Heap. To delete root node it needs to be swapped with last node (55). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 82 with 55.

12, 55, 77, 23, 15, 10, 82, 90

Step 4 - Delete root (77) from the Max Heap. To delete root node it needs to be swapped with last node (10). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 77 with 10.

12, 55, 10, 23, 15, 77, 82, 90

Step 5 - Delete root (55) from the Max Heap. To delete root node it needs to be swapped with last node (15). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 55 with 15.

12, 15, 10, 23, 55, 77, 82, 90

Step 6 - Delete root (23) from the Max Heap. To delete root node it needs to be swapped with last node (12). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 23 with 12.

12, 15, 10, 23, 55, 77, 82, 90

Step 7 - Delete root (15) from the Max Heap. To delete root node it needs to be swapped with last node (10). After delete tree needs to be heapify to make it Max Heap.

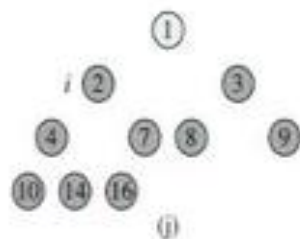
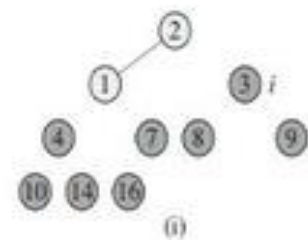
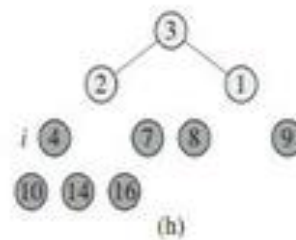
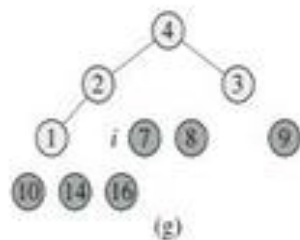
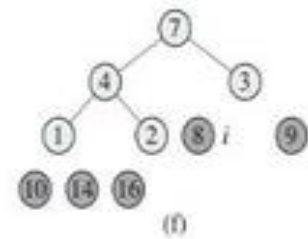
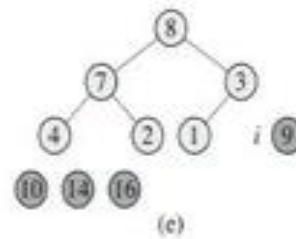
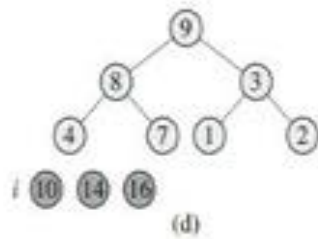
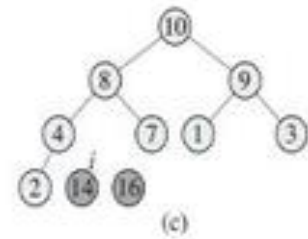
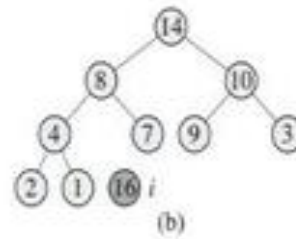
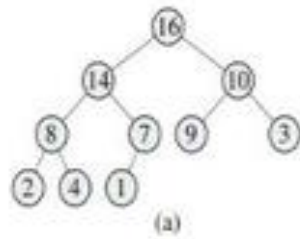


list of numbers after Deleting 15, 12 & 10 from the Max Heap.

10, 12, 15, 23, 55, 77, 82, 90

Whenever Max Heap becomes Empty, the list get sorted in Ascending order

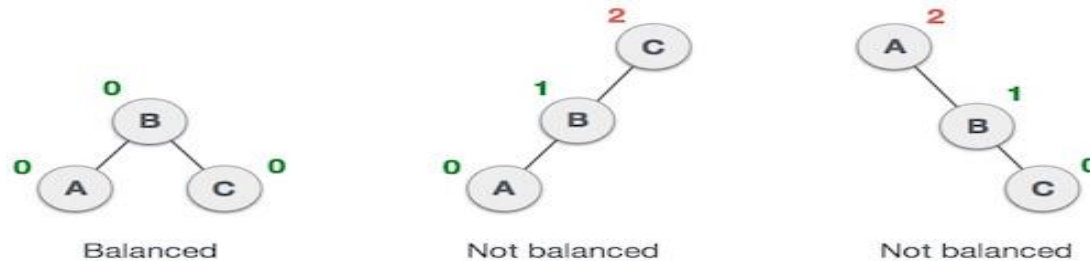
Sort the following list using Heap Sort 14, 8, 10, 16, 7, 9, 3, 2, 4, 1



Balanced Binary Trees- AVL Trees, Insertion, Deletion and Rotations.

Named after their inventor Adelson, Velski & Landis, AVL trees are height balancing binary search tree. AVL tree checks the height of the left and the right sub-trees and assures that the difference is not more than 1. This difference is called the Balance Factor.

Here we see that the first tree is balanced, and the next two trees are not balanced –



In the second tree, the left subtree of C has height 2 and the right subtree has height 0, so the difference is 2. In the third tree, the right subtree of A has height 2 and the left is missing, so it is 0, and the difference is 2 again. AVL tree permits difference (balance factor) to be only 1.

Balance Factor = height(left-subtree) – height(right-subtree)

If the difference in the height of left and right sub-trees is more than 1, the tree is balanced using some rotation techniques.

AVL Rotations

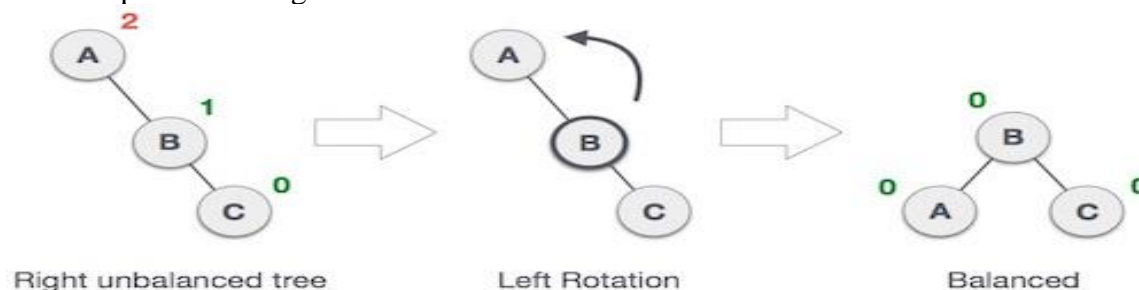
To balance itself, an AVL tree may perform the following four kinds of rotations –

- Left rotation
- Right rotation
- Left-Right rotation
- Right-Left rotation

The first two rotations are single rotations and the next two rotations are double rotations. To have an unbalanced tree, we at least need a tree of height 2. With this simple tree, let's understand them one by one.

Left Rotation

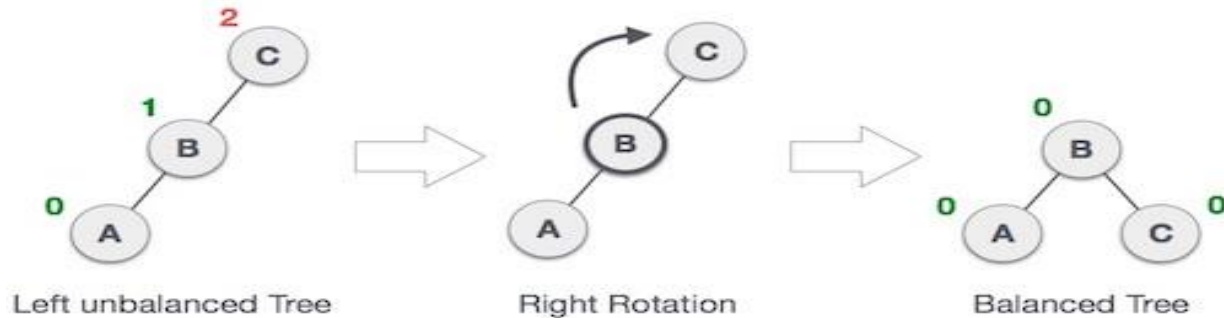
If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation –



In our example, node A has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the left rotation by making A the left-subtree of B.

Right Rotation

AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.

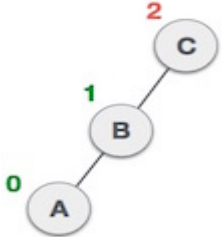
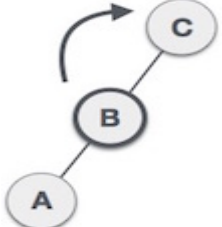
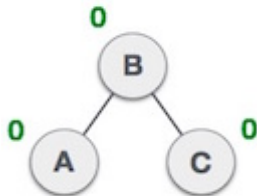


As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.

Left-Right Rotation

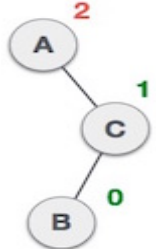
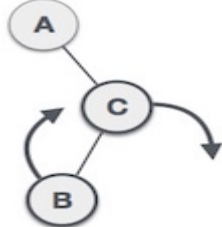
Double rotations are slightly complex version of already explained versions of rotations. To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation. A left-right rotation is a combination of left rotation followed by right rotation.

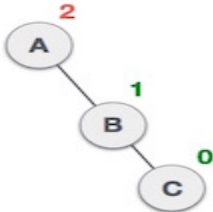
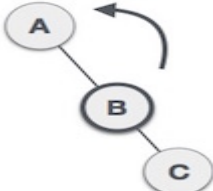
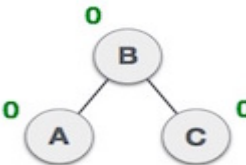
State	Action
	A node has been inserted into the right subtree of the left subtree. This makes C an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.
	We first perform the left rotation on the left subtree of C. This makes A, the left subtree of B.

	<p>Node C is still unbalanced, however now, it is because of the left-subtree of the left-subtree.</p>
	<p>We shall now right-rotate the tree, making B the new root node of this subtree. C now becomes the right subtree of its own left subtree.</p>
	<p>The tree is now balanced.</p>

Right-Left Rotation

The second type of double rotation is Right-Left Rotation. It is a combination of right rotation followed by left rotation.

State	Action
	<p>A node has been inserted into the left subtree of the right subtree. This makes A, an unbalanced node with balance factor 2.</p>
	<p>First, we perform the right rotation along C node, making C the right subtree of its own left subtree B. Now, B becomes the right subtree of A.</p>

	<p>Node A is still unbalanced because of the right subtree of its right subtree and requires a left rotation.</p>
	<p>A left rotation is performed by making B the new root node of the subtree. A becomes the left subtree of its right subtree B.</p>
	<p>The tree is now balanced.</p>

Insertion Operation in AVL Tree

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

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

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

Data Structures

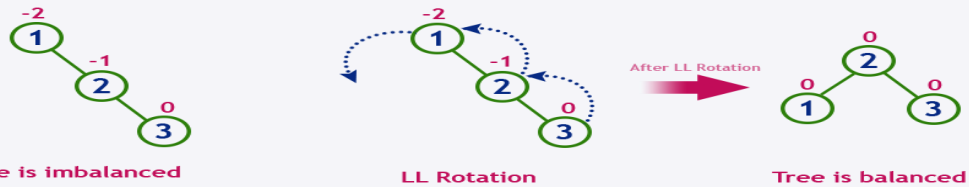
insert 1



insert 2



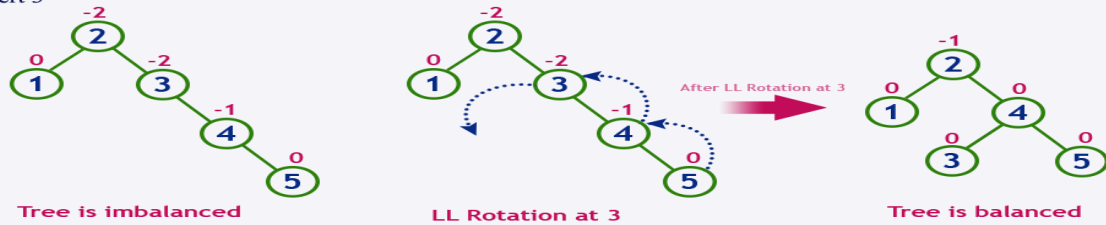
insert 3



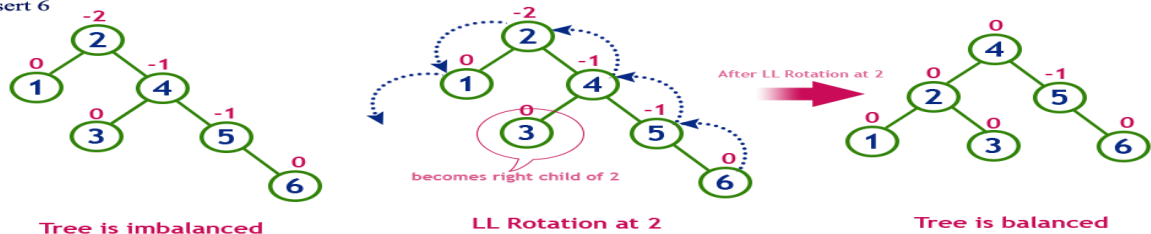
insert 4



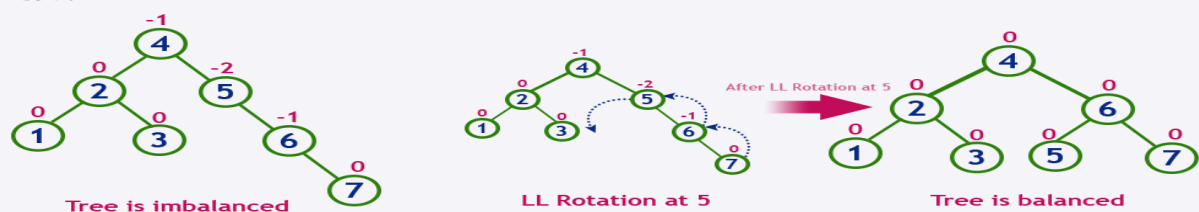
insert 5



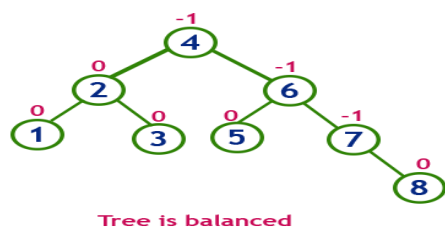
insert 6



insert 7



insert 8



Deletion in AVL Trees

Deletion is also very straight forward. We delete using the same logic as in simple binary search trees. After deletion, we restructure the tree, if needed, to maintain its balanced height.

Step 1: Find the element in the tree.

Step 2: Delete the node, as per the BST Deletion.

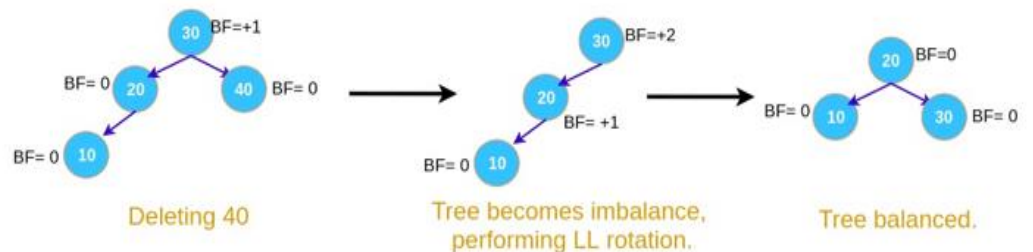
Step 3: Two cases are possible:-

Case 1: Deleting from the right subtree.

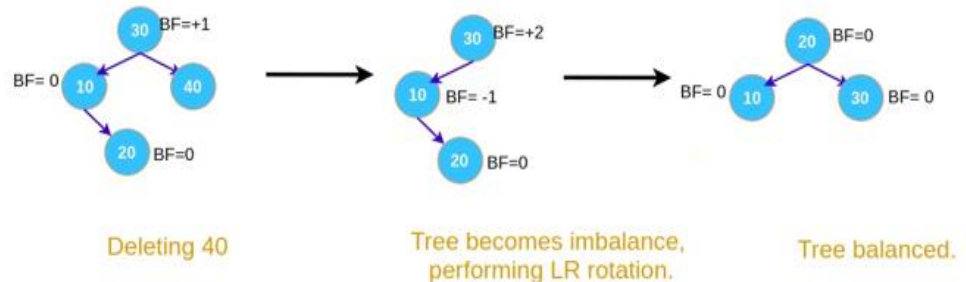
- 1A. If $BF(\text{node}) = +2$ and $BF(\text{node} \rightarrow \text{left-child}) = +1$, perform LL rotation.
- 1B. If $BF(\text{node}) = +2$ and $BF(\text{node} \rightarrow \text{left-child}) = -1$, perform LR rotation.
- 1C. If $BF(\text{node}) = +2$ and $BF(\text{node} \rightarrow \text{left-child}) = 0$, perform LL rotation.

Deletion: Case 1 (deleting from right sub tree)

Case: 1A

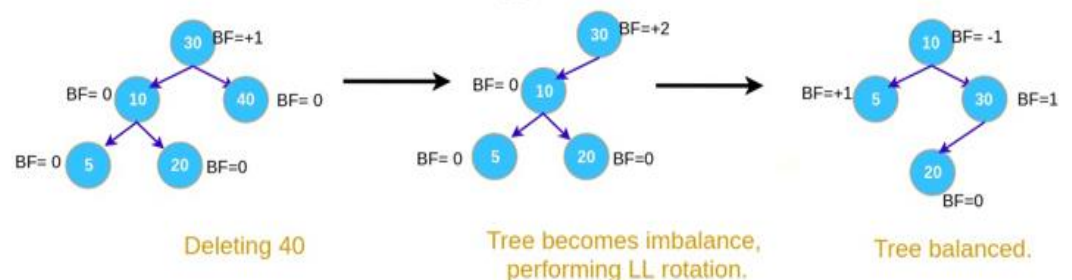


Case: 1B



Guru99.com

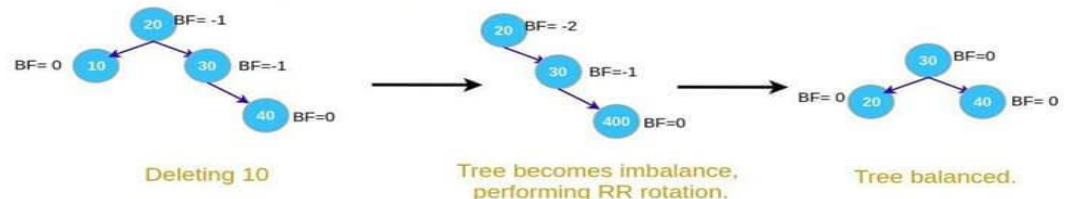
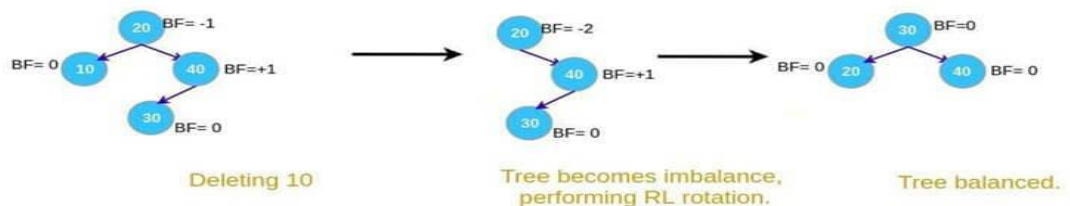
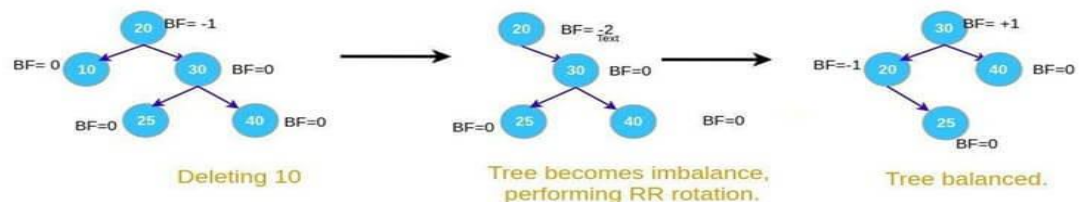
Case: 1C



Case 2: Deleting from left subtree.

- 2A. If $BF(\text{node}) = -2$ and $BF(\text{node} \rightarrow \text{right-child}) = -1$, perform RR rotation.
- 2B. If $BF(\text{node}) = -2$ and $BF(\text{node} \rightarrow \text{right-child}) = +1$, perform RL rotation.
- 2C. If $BF(\text{node}) = -2$ and $BF(\text{node} \rightarrow \text{right-child}) = 0$, perform RR rotation.

Deletion: Case 2 (deleting from left sub tree)

Case: 2A**Case: 2B****Case: 2C****Deletion in AVL Tree**

Deleting a node from an AVL tree is similar to that in a binary search tree. Deletion may disturb the balance factor of an AVL tree and therefore the tree needs to be rebalanced in order to maintain the AVLness. For this purpose, we need to perform rotations. The two types of rotations are L rotation and R rotation. Here, we will discuss R rotations. L rotations are the mirror images of them.

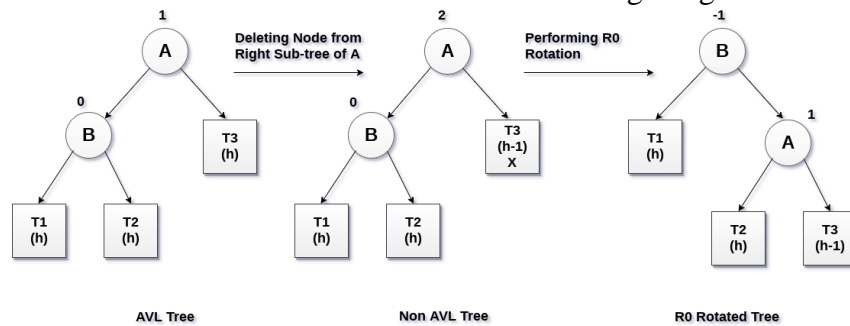
If the node which is to be deleted is present in the left sub-tree of the critical node, then L rotation needs to be applied else if, the node which is to be deleted is present in the right sub-tree of the critical node, the R rotation will be applied.

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

R0 rotation (Node B has balance factor 0)

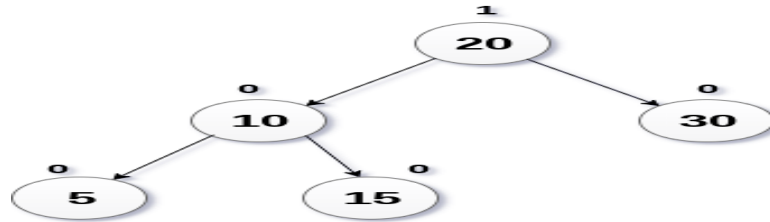
If the node B has 0 balance factor, and the balance factor of node A disturbed upon deleting the node X, then the tree will be rebalanced by rotating tree using R0 rotation.

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



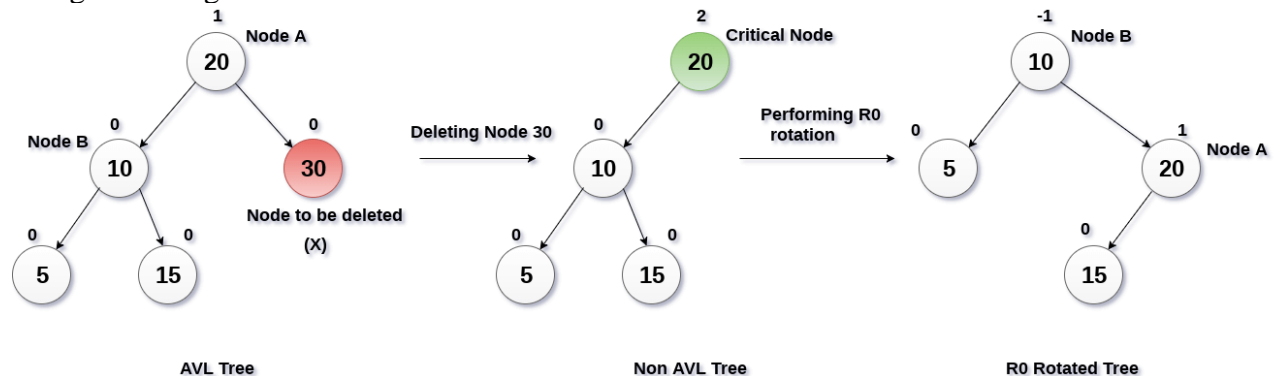
Example:

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



Solution:

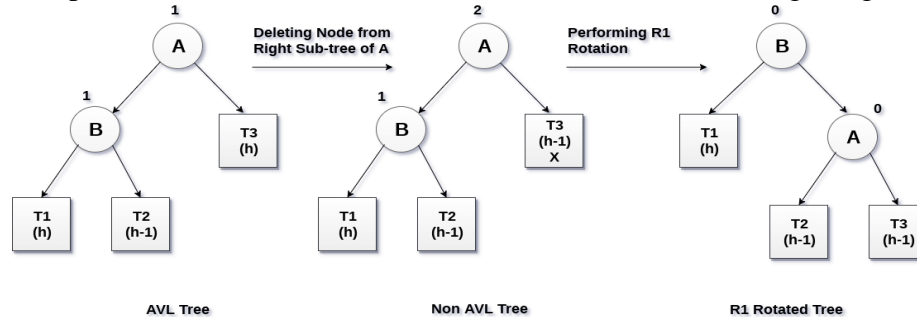
In this case, the node B has balance factor 0, therefore the tree will be rotated by using R0 rotation as shown in the following image. The node B(10) becomes the root, while the node A is moved to its right. The right child of node B will now become the left child of node A.



R1 Rotation (Node B has balance factor 1):

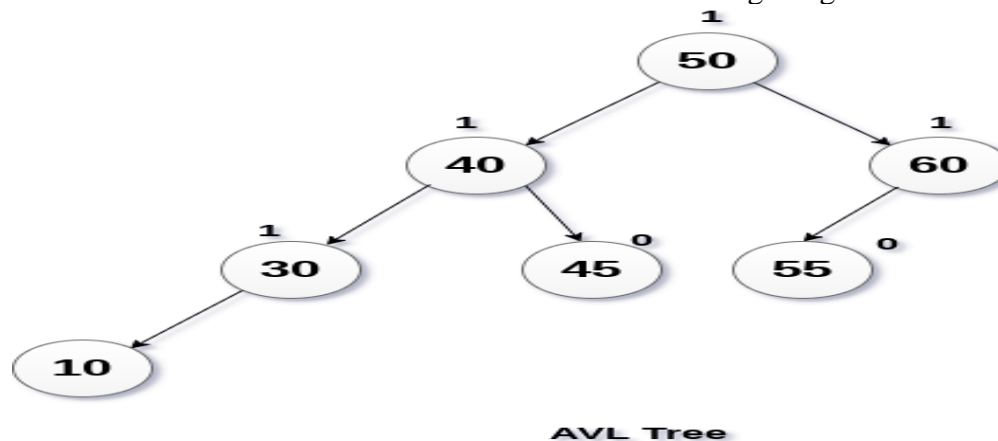
R1 Rotation is to be performed if the balance factor of Node B is 1. In R1 rotation, the critical node A is moved to its right having sub-trees T2 and T3 as its left and right child respectively. T1 is to be placed as the left sub-tree of the node B.

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



Example

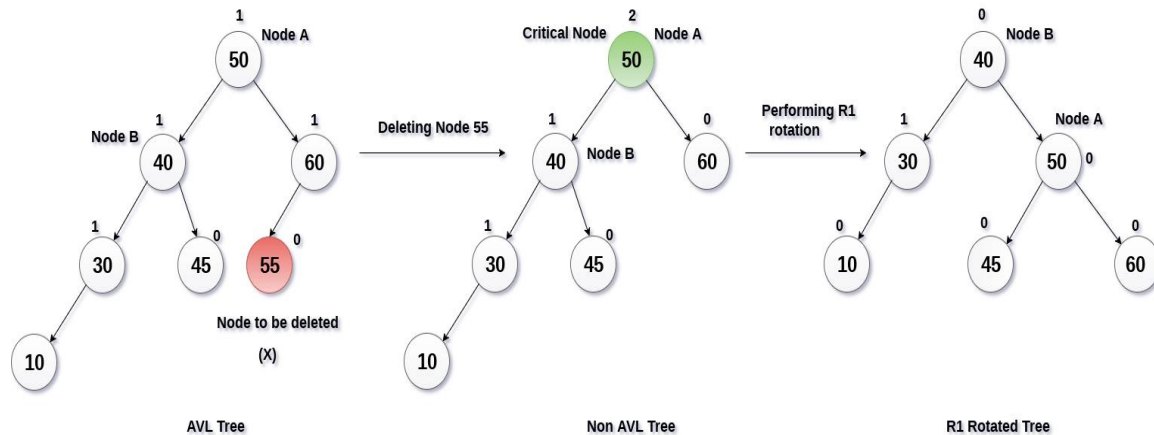
Delete Node 55 from the AVL tree shown in the following image.



Solution :

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

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

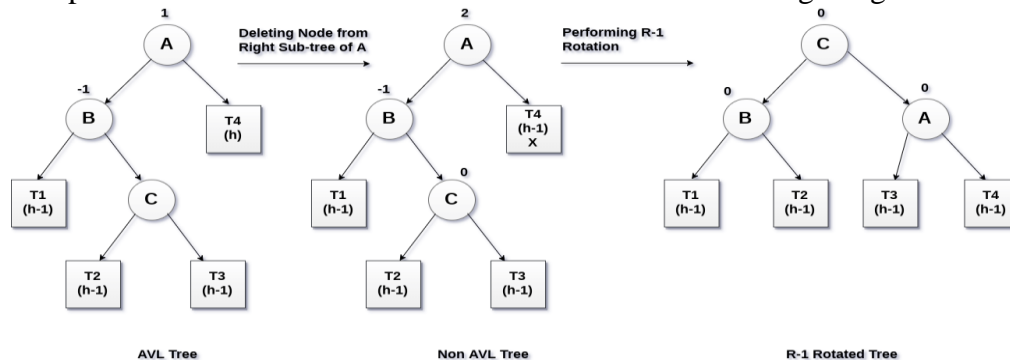


R-1 Rotation (Node B has balance factor -1)

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

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

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



Example

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



Solution:

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

