String: - representation of strings, concatenation, substring searching and deletion.

Trees: - m-ary Tree, Binary Trees – level and height of the tree, complete-binary tree representation using array, tree traversals (Recursive and non-recursive), applications. Binary search tree – creation, insertion and deletion and search operations, applications.

String Representation

In C programming, array of characters is called a string. A string is terminated by a null character /0.

For example:

Here, "c string tutorial" is a string. When, compiler encounter strings, it appends a null character `/0` at the end of string.

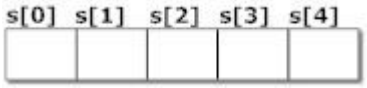
Declaration of strings

Before we actually work with strings, we need to declare them first.

Strings are declared in a similar manner as arrays. Only difference is that, strings are of char type.

Using arrays

char s[5];



Initialization of strings

In C, string can be initialized in a number of different ways.

For convenience and ease, both initialization and declaration are done in the same step.

Using arrays

```
char c[] = "abcd";
```

OR,

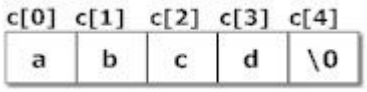
```
char c[50] = "abcd";
```

OR,

```
char c[] = {'a', 'b', 'c', 'd', '\0'};
```

OR,

```
char c[5] = {'a', 'b', 'c', 'd', '\0'};
```



String concatenation

String is nothing but an array of characters (OR char data types). While doing programming in C language, you may have faced challenges wherein you might want to compare two strings, concatenate strings, copy one string to another & perform various string manipulation operations. All of such kind of operations plus many more oth

ng.h" header file. In order to use these string functions you must include string.h file in your C program.

In simple words, String is an one dimensional array of characters. This library function concatenates a string onto the end of the other string.i.e, The strcat() function appends s2 to the end of s1. String s2 is unchanged.

WAP to concatenate two string with using string functions

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

int main()
{
   char a[100], b[100];

   printf("Enter the first string\n");
```

```
    gets(a);

    printf("Enter the second string\n");
    gets(b);

    strcat(a,b);

    printf("String obtained on concatenation is %s\n",a);


}
```

WAP to concatenate two string without using string functions

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

int main()
{
   char s1[50], s2[30];
   int i=0,j=0;
   printf("\nEnter String 1 :");
   gets(s1);
   printf("\nEnter String 2 :");
   gets(s2);
   while(s1[i]!='\0')
{
   i++;
}
while(s2[j]!='\0')
{
```

```
  s1[i]=s2[j];
  i++;
  j++;
 }
    s1[i] = '\0';
   printf("nConcated string is :%s", s1);
}
```

Substring searching and deletion

C programming code to check if a given string is present in another string, For example the string "programming" is present in "c programming". If the string is present then it's location (i.e. at which position it is present) is printed.

Substring searching

```
#include<stdio.h>

#include<string.h>

int main()

{

char s1[20],s2[20];

int I,j,m,l,l1;

printf(“Enter the first string”);

gets(s1);

printf(“Enter the second string”);

gets(s2);

l=strlen(s1);

l1=strlen(s2);
```

```
for(i=0;1<l;i++)
{
 j=0;
 if(s1[i]==s2[j])
 {
    m=i;
sign=0;
while((s2[j]!='\0')&&(sign!=1))
{
 if(s1[m]==s2[j])
{
m++;
j++;
}
else
 sign=1;
}
}
}
if(sign==0)
{
printf("The given substring is present in the location %d",i+1);
}
```

```
else
{
printf("Substring not present");
}
}
```

Substring Deletion

```
#include<stdio.h>
#include<string.h>
int main()
{
char s1[20],s2[20];
int I,j,m,l,l1;
printf("Enter the first string");
gets(s1);
printf("Enter the second string");
gets(s2);
l=strlen(s1);
l1=strlen(s2);
for(i=0;1<l;i++)
{
  j=0;
  if(s1[i]==s2[j])
  {
```

```
      m=i;
    sign=0;
while((s2[j]!='\0')&&(sign!=1))
 {
       if(s1[m]==s2[j])
           {
           m++;
           j++;
           }
else
           sign=1;
}
}
}
if(sign==0)
           {
           printf("The given substring is present in the location %d",i+1);
           m=i+l1;
           while(s1[m]!='\0')
                      {
                      s1[i]=s1[m];
                      i++;
```

```
            m++;
            }
        s1[i]='\0';
        printf("Substring after deletion is");
        puts(s1);
        }
else
        {
        printf("Substring is not present");
        }
}
```

Tree

Definition

Tree data structure is a collection of data (Node) which is organized in hierarchical structure and this is a recursive definition

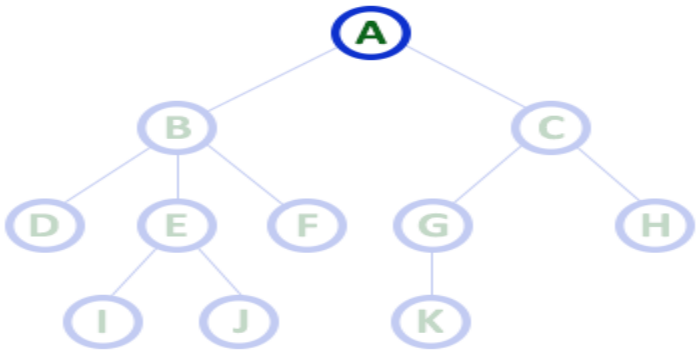
In tree data structure, every individual element is called as Node. Node in a tree data structure, stores the actual data of that particular element and link to next element in hierarchical structure.

Tree Terminologies

In a tree data structure, we use the following terminology...

1. Root

In a tree data structure, the first node is called as **Root Node**. Every tree must have root node. We can say that root node is the origin of tree data structure. In any tree, there must be only one root node. We never have multiple root nodes in a tree.

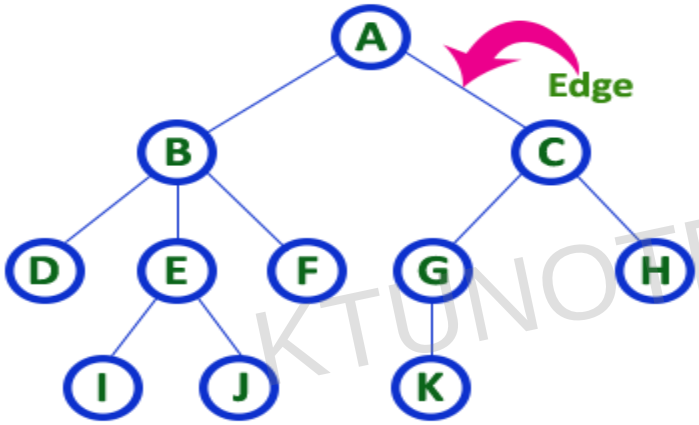


Here 'A' is the 'root' node

- In any tree the first node is called as ROOT node

2. Edge

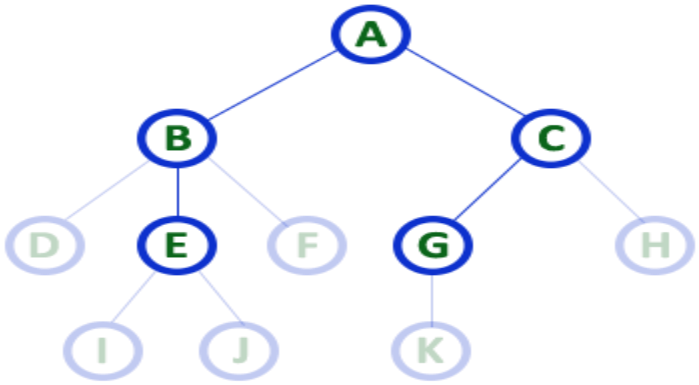
In a tree data structure, the connecting link between any two nodes is called as **EDGE**. In a tree with '**N**' number of nodes there will be a maximum of '**N-1**' number of edges.



- In any tree, 'Edge' is a connecting link between two nodes.

3.Parent

In a tree data structure, the node which is predecessor of any node is called as **PARENT NODE**. In simple words, the node which has branch from it to any other node is called as parent node. Parent node can also be defined as "**The node which has child / children**".

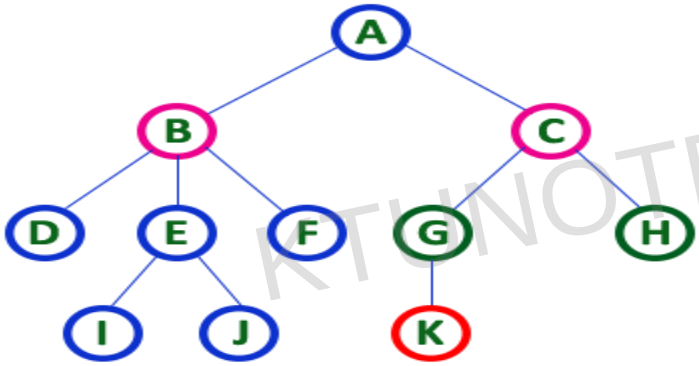


Here A, B, C, E & G are Parent nodes

- In any tree the node which has child / children is called 'Parent'
- A node which is predecessor of any other node is called 'Parent'

4. Child

In a tree data structure, the node which is descendant of any node is called as **CHILD Node**. In simple words, the node which has a link from its parent node is called as child node. In a tree, any parent node can have any number of child nodes. In a tree, all the nodes except root are child nodes.

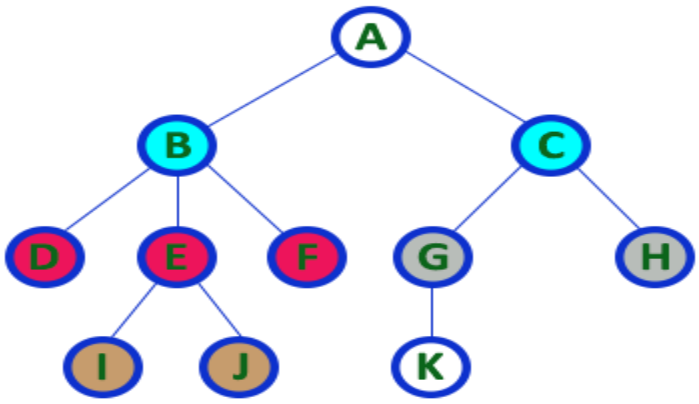


Here B & C are Children of A
Here G & H are Children of C
Here K is Child of G

- descendant of any node is called as CHILD Node

5. Siblings

In a tree data structure, nodes which belong to same Parent are called as **SIBLINGS**. In simple words, the nodes with same parent are called as Sibling nodes.



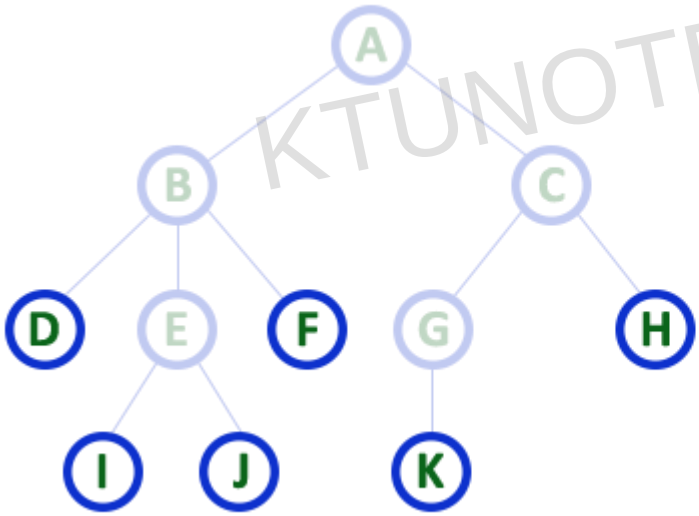
Here B & C are Siblings
Here D E & F are Siblings
Here G & H are Siblings
Here I & J are Siblings

- In any tree the nodes which has same Parent are called 'Siblings'
- The children of a Parent are called 'Siblings'

6. Leaf/Terminal Node

In a tree data structure, the node which does not have a child is called as **LEAF Node**. In simple words, a leaf is a node with no child. The degree of a node is zero then it is a leaf

In a tree data structure, the leaf nodes are also called as **External Nodes**. External node is also a node with no child. In a tree, leaf node is also called as '**Terminal**' node.



Here D, I, J, F, K & H are Leaf nodes

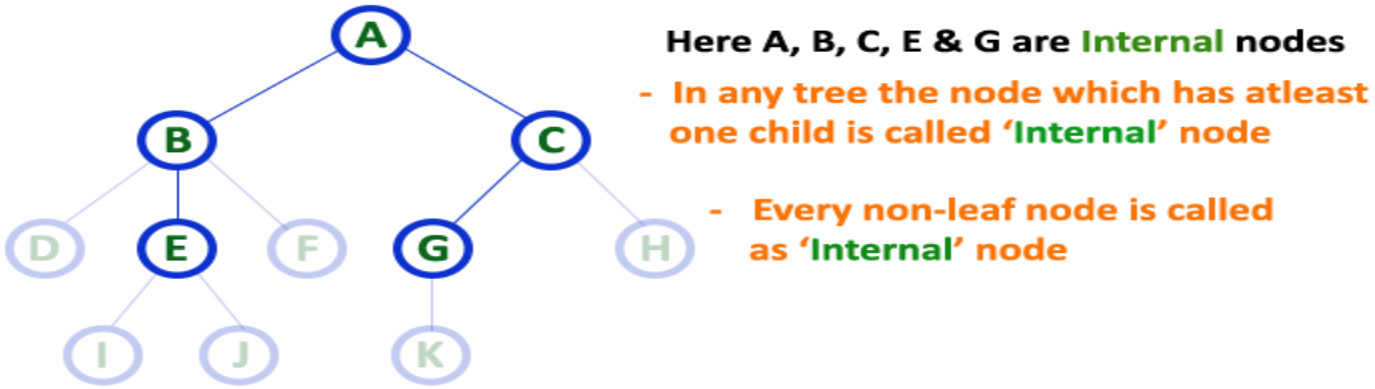
- In any tree the node which does not have children is called 'Leaf'
- A node without successors is called a 'leaf' node

7. Internal Nodes/Non Terminal Nodes

In a tree data structure, the node which has atleast one child is called as **INTERNAL Node**. In simple words, an internal node is a node with atleast one child.

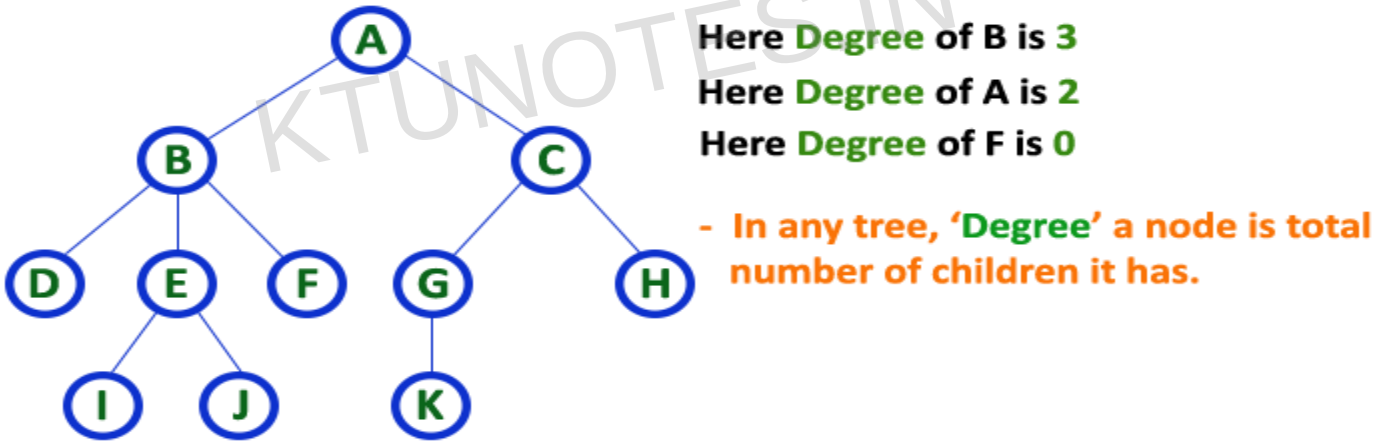
In a tree data structure, nodes other than leaf nodes are called as **Internal Nodes**. **The root node is also**

said to be Internal Node if the tree has more than one node. Internal nodes are also called as '**Non-Terminal**' nodes.



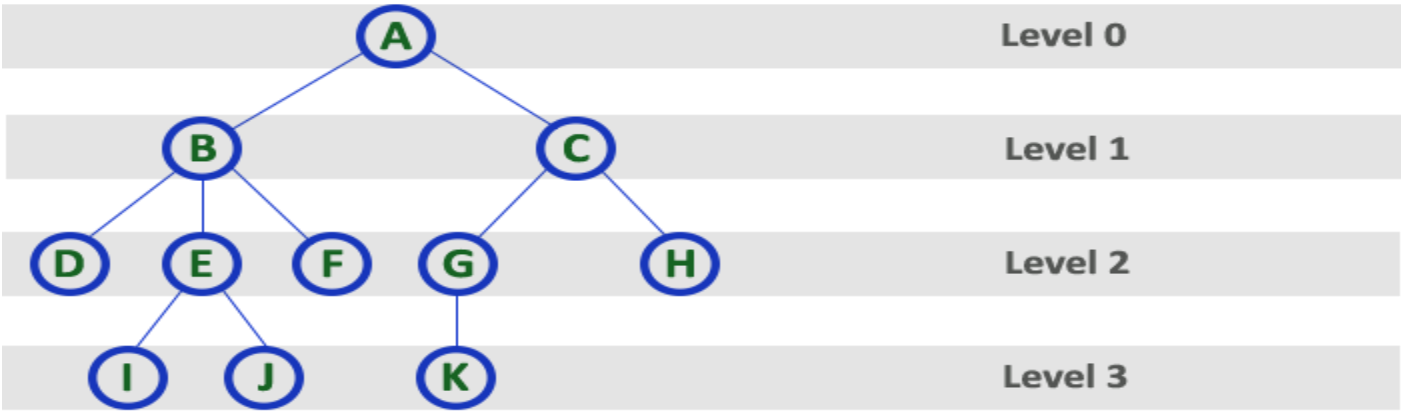
8. Degree

In a tree data structure, the total number of children of a node is called as **DEGREE** of that Node. In simple words, the Degree of a node is total number of children it has. The highest degree of a node among all the nodes in a tree is called as '



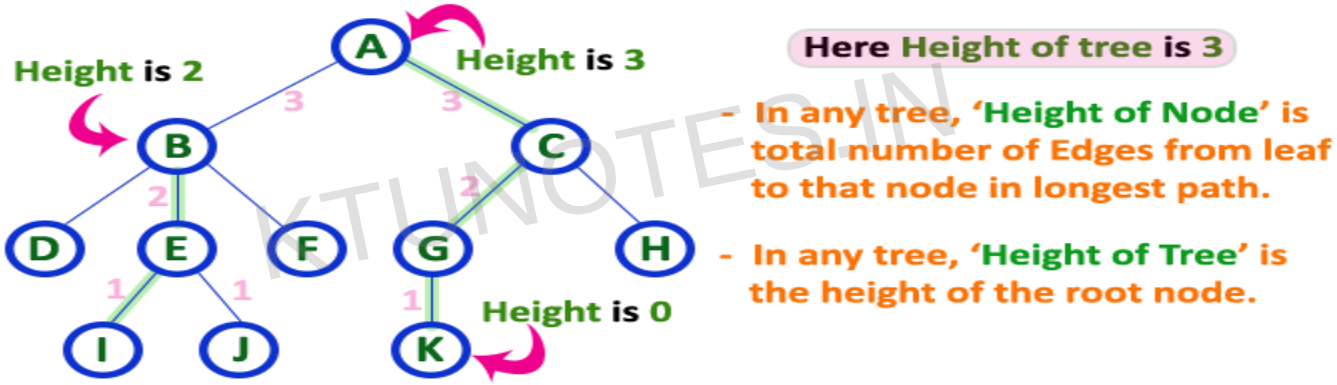
9. Level

In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on... In simple words, in a tree each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each level (Step).



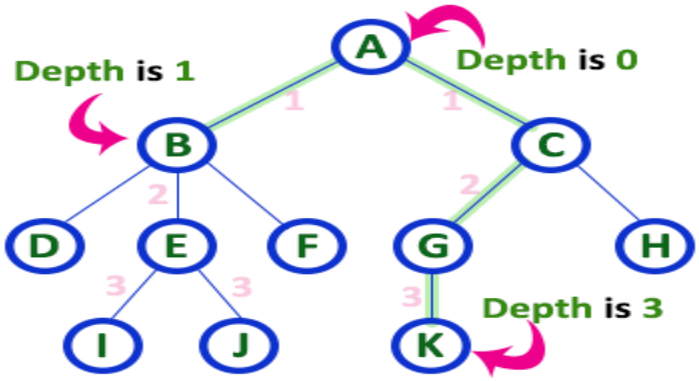
10. Height

In a tree data structure, the total number of egdes from leaf node to a particular node in the longest path is called as **HEIGHT** of that Node. In a tree, height of the root node is said to be **height of the tree**. In a tree, **height of all leaf nodes is '0'.**



11. Depth

In a tree data structure, the total number of egdes from root node to a particular node is called as **DEPTH** of that Node. In a tree, the total number of edges from root node to a leaf node in the longest path is said to be **Depth of the tree**. In simple words, the highest depth of any leaf node in a tree is said to be depth of that tree. In a tree, **depth of the root node is '0'.**

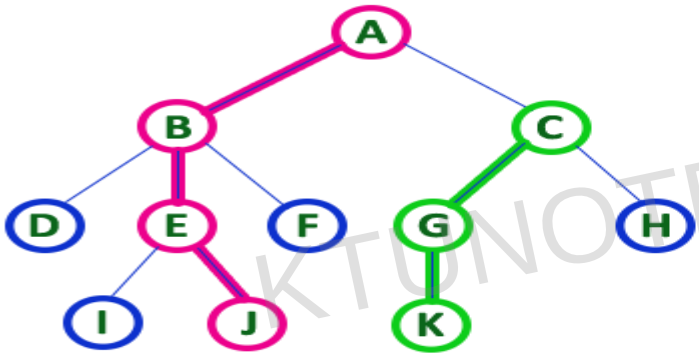


Here Depth of tree is 3

- In any tree, 'Depth of Node' is total number of Edges from root to that node.
- In any tree, 'Depth of Tree' is total number of edges from root to leaf in the longest path.

12. Path

In a tree data structure, the sequence of Nodes and Edges from one node to another node is called as **PATH** between that two Nodes. **Length of a Path** is total number of nodes in that path. In below example **the path A - B - E - J has length 4**.



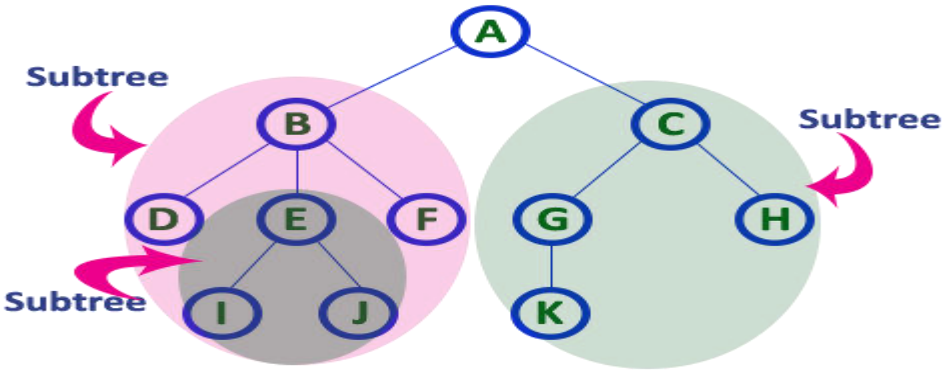
- In any tree, 'Path' is a sequence of nodes and edges between two nodes.

Here, 'Path' between A & J is
A - B - E - J

Here, 'Path' between C & K is
C - G - K

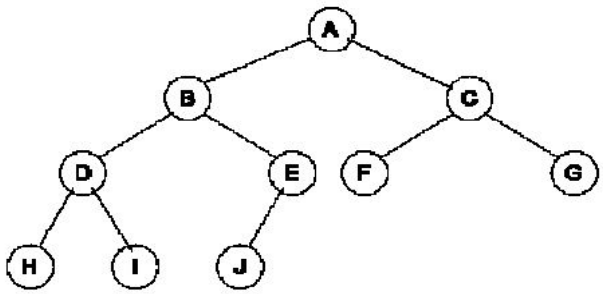
13. Sub Tree

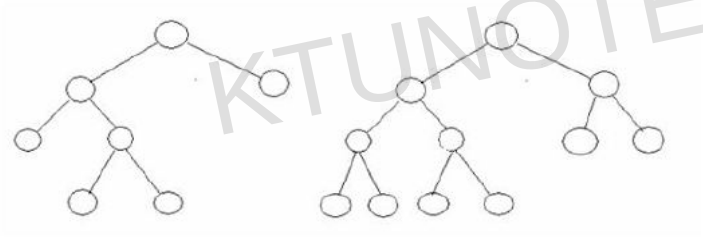
In a tree data structure, each child from a node forms a subtree recursively. Every child node will form a subtree on its parent node.



Binary Tree:

A Tree in which each node has a degree of atmost 2. i.e. it can have either 0,1 or 2 children.



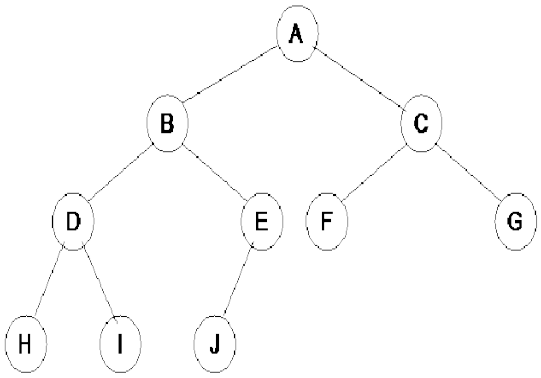


Here, leaves are H, I, J. Except these, remaining internal nodes has atmost 2 nodes as their child.

Complete Binary Tree

- A complete binary tree is a tree that is completely filled, with the possible exception of the bottom level.
- The bottom level is filled from left to right.

A binary tree T with n levels is complete if all levels except possibly the last are completely full, and the last level has all its nodes to the left side.You may find the definition of complete binary tree in the books little bit different from this. A perfectly complete binary tree has all the leaf nodes. In the complete binary tree, all the nodes have left and right child nodes except the bottom level. At the bottom level, you will find the nodes from left to right. The bottom level may not be completely filled, depicting that the tree is not a perfectly complete one. Let's see a complete binary tree in the figure below:

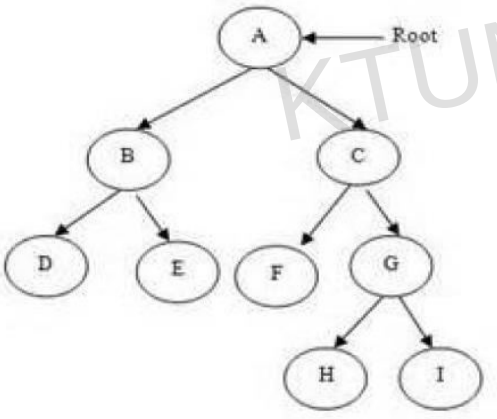


In the above tree, we have nodes as A, B, C, D, E, F, G, H, I, J. The node D has two children at the lowest level whereas node E has only left child at the lowest level that is J. The right child of the node E is missing. Similarly node F and G also lack child nodes. This is a complete binary tree according to the definition given above. At the lowest level, leaf nodes are present from left to right but all the inner nodes have both children. Let's recap some of the properties of complete binary tree.

- A complete binary tree of height h has between 2^h to 2^{h+1} –1 nodes.

Full Binary tree:

- Strictly BT either has 0 or 2 subtrees



Tree Traversal

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot random access a node in a tree. There are three ways which we use to traverse a tree −

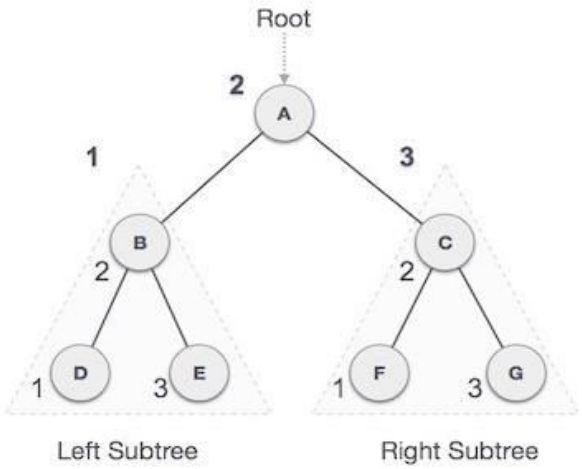
- In-order Traversal
- Pre-order Traversal

- Post-order Traversal

In-order Traversal

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.

If a binary tree is traversed **in-order**, the output will produce sorted key values in an ascending order.



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 inorder traversal of this tree will be

Algorithm

Until all nodes are traversed −
Step 1 Traverse the left subtree in Inorder
Step 2 − Visit root node.
Step 3 − Traverse the right subtree in Inorder

Non-Recursive Algorithm

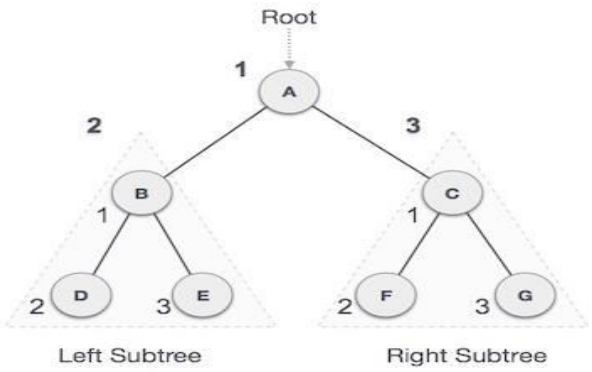
This algorithm traverses the tree non-recursively in Inorder using STACK. Initially push NULL onto stack and then set P=root and the repeat the following steps until NULL is popped from stack
Step 1 – Proceed down the leftmost path rooted at P, pushing each node N onto stack and stopping when a node N with no left child is pushed onto stack
Step 2 – Pop and process the nodes on stack. If NULL is popped then exit. If a node N with a right child P->right is processed, set P=P->right and return to step 1.

```
void intrav(nodeptr tree)
{
  if(tree!=NULL)
  {
    intrav(tree->left);
```

```
    printf("->%d",tree->data);
    intrav(tree->right);
  }
}
```

Preorder Traversal

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.



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 −

Algorithm

Until all nodes are traversed −
Step 1 − Visit root node.
Step 2 − Traverse the left subtree in Preorder.
Step 3 − Traverse the right subtree in Preorder.

Non-Recursive Algorithm

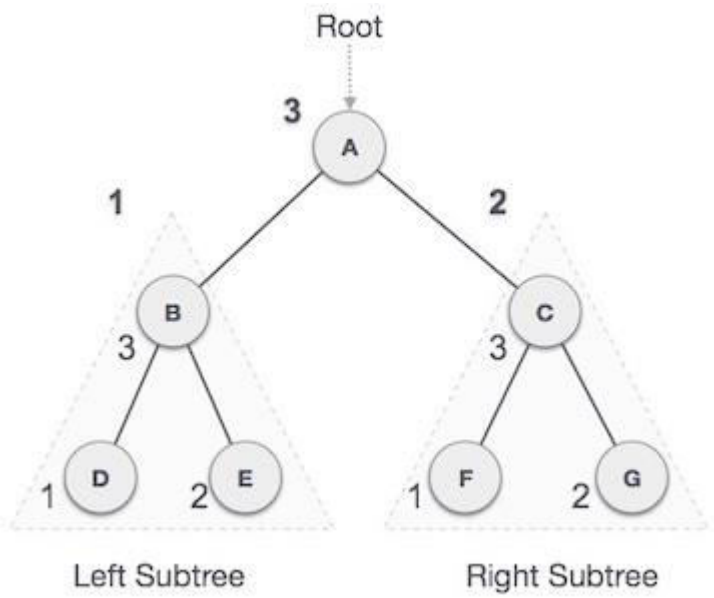
This algorithm traverses the tree non-recursively in preorder using STACK. Initially push NULL onto stack and then set P=root and the repeat the following steps input p!=NULL
Step 1 – Proceed down the leftmost path rooted at P, processing each node N on the path and pushing each right child, P->right. If any , onto stack. The traversing ends after a node N with no left child L(N) is processed. Therefore the pointer P is updated using the assignment P=P->left and the traversing stops when P->left=NULL.
Step 2 – Pop the element from stack and assign to P. If P≠NULL, then return step 1 else exit

```
void pretrav(nodeptr tree)
{
  if(tree!=NULL)
  {
    printf("->%d",tree->data);
    pretrav(tree->left);
    pretrav(tree->right);
```

```
    }
}
```

Post-order Traversal

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.



We start from **A**, and following pre-order traversal, we first visit the left subtree **B**. **B** is also traversed

versal of this

D → E → B → F → G → C → A

Algorithm

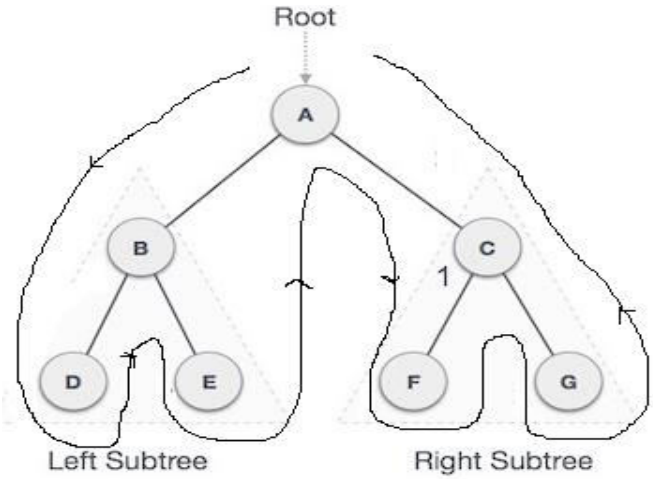
Until all nodes are traversed −
Step 1 − Traverse the left subtree in Postorder.
Step 2 − Traverse the right subtree in Postorder.
Step 3 − Visit root node.

Non-Recursive Algorithm

This algorithm traverses the tree non-recursively in Inorder using STACK. Initially push NULL onto stack and then set P=root and the repeat the following steps until NULL is popped from stack
Step 1 – Proceed down the leftmost path rooted at P. At each node N of the path and push N of the path, push N onto stack and if N has a right child P->right, push P->right onto
Step 2 – Pop and process nodes on stack. If Null is popped, then exit. Return to step 1

void posttrav(nodeptr tree)

```
{
  if(tree!=NULL)
  {
    posttrav(tree->left);
    posttrav(tree->right);
    printf("->%d",tree->data);
  }
}
```



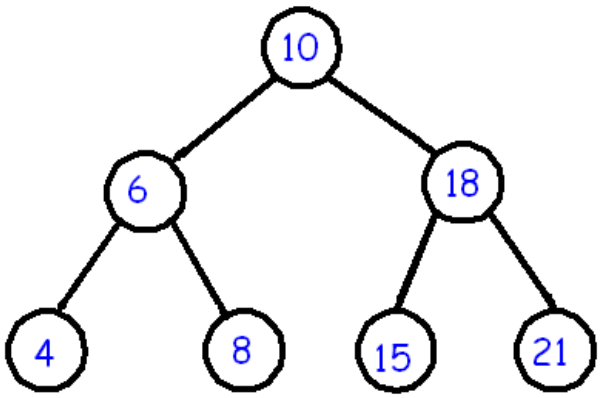
Preorder- A->B->D->E->C->F->G

Inorder- C-

Postorder-D->E->B->F->G->C->A

Binary Search Trees

We consider a particular kind of a binary tree called a Binary Search Tree (BST). The basic idea behind this data structure is to have such a storing repository that provides the efficient way of data sorting, searching and retrieving.



A BST is a binary tree where nodes are ordered in the following way:

- each node contains one key (also known as data)
- the keys in the left subtree are less then the key in its parent node, in short L < P;
- the keys in the right subtree are greater the key in its parent node, in short P < R;

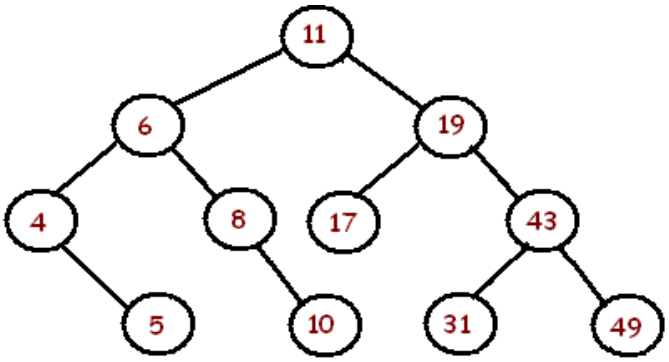
- duplicate keys are not allowed.

In the following tree all nodes in the left subtree of 10 have keys < 10 while all nodes in the right subtree > 10. Because both the left and right subtrees of a BST are again search trees; the above definition is recursively applied to all internal nodes:

Exercise. Given a sequence of numbers:

`11, 6, 8, 19, 4, 10, 5, 17, 43, 49, 31`

Draw a binary search tree by inserting the above numbers



Searching

To search a given key in Bianry Search Tree, we first compare it with root, if the key is present at root, we return root. If key is greater than root's key, we recur for right subtree of root node. Otherwise we recur for left subtree.

Now, let's see more detailed description of the search algorithm. Like an add operation, and almost every operation on BST, search algorithm utilizes recursion. Starting from the root,

Steps

1. check, whether value in current node and searched value are equal. If so, **value is found.** Otherwise,
2. if searched value is less, than the node's value:
    - if current node has no left child, **searched value doesn't exist in the BST;**
    - otherwise, handle the left child with the same algorithm.
3. if a new value is greater, than the node's value:

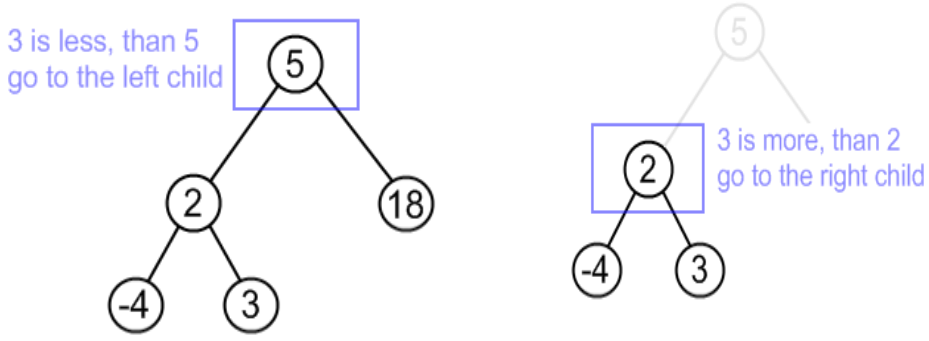
- if current node has no right child, **searched value doesn't exist in the BST;**
- otherwise, handle the right child with the same algorithm.

Algorithm

1. void search(node *tree, int digit)
2. if(tree==NULL) then step 3 otherwise goto step 4
3. printf("The Number does not exists");
4. else if(digit==tree->num) then goto step 5 otherwise go to step 6
5. printf("%d is found",digit);
6. else if(digit<tree->num) then goto step 7 otherwise go to step 8
7. search(tree->left,digit)
8. else search(tree->right,digit)

Example

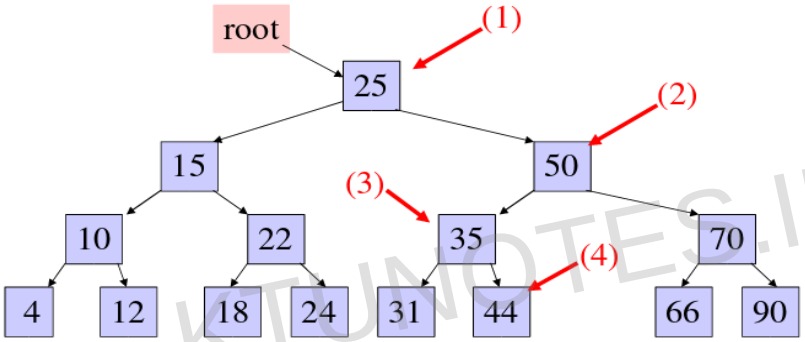
Search for 3 in the tree, shown above.



Example: search for 45 in the tree

(key fields are show in node rather than in separate obj ref to by data field):

1. start at the root, 45 is greater than 25, search in right subtree
2. 45 is less than 50, search in 50's left subtree
3. 45 is greater than 35, search in 35's right subtree
4. 45 is greater than 44, but 44 has no right subtree so 45 is not in the BST



Insertion

Steps for insertion

- Check whether root node is present or not(tree available or not). If root is NULL, create root node.
- If the element to be inserted is less than the element present in the root node, traverse the left sub-tree recursively until we reach T->left/T->right is NULL and place the new node at T->left(key in new node < key in T)/T->right (key in new node > key in T).
- If the element to be inserted is greater than the element present in root node, traverse the right sub-tree recursively until we reach T->left/T->right is NULL and place the new node at T->left/T->right.

Algorithm

1. void insert(node *tree,int digit)
2. if(tree==NULL) then step from 3 to 6 then goto step 6
3. tree=(node*)malloc(sizeof(node));
4. tree->left=tree->right=NULL;
5. tree->num=digit;
6. if(digit<tree->num) then goto step 7 otherwise step 8
7. tree->left=insert(tree->left,digit)
8. if(digit>tree->num) then goto step 9 otherwise goto step 10
9. tree->right=insert(tree->right,digit)
10. if(digit==tree->num)then goto step 11
11. print “Duplicate Nodes”
12. return(tree)
13. End

A new key is always inserted at leaf. We start searching a key from root till we hit a leaf node. Once a leaf node is found, the new node is added as a child of the leaf node.

Example:

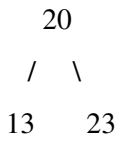
Insert 20 into the Binary Search Tree.

Tree is not available. So, create root node and place 10 into it.

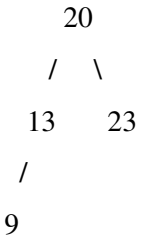
20

Insert 23 into the given Binary Search Tree. 23 > 20 (data in root). So, 23 needs to be inserted in the right sub-tree of 20.

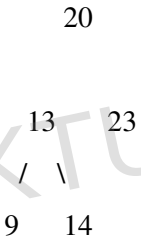
Insert 13 into the given Binary Search Tree. 13 < 20(data in root). So, 13 needs to be inserted in left sub-tree of 20.



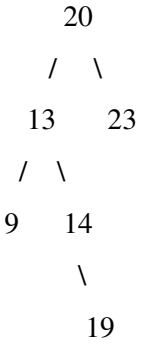
Insert 9 into the given Binary Search Tree.



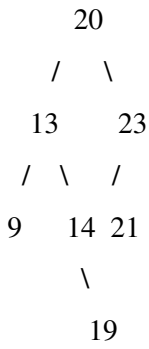
Inserting 14.



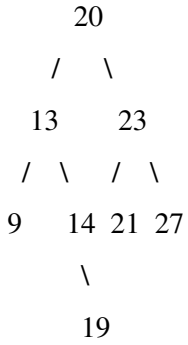
Inserting 19.



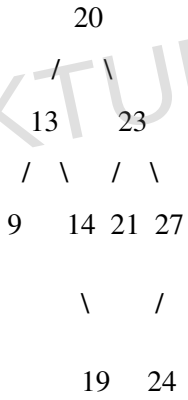
Inserting 21.



Inserting 27.

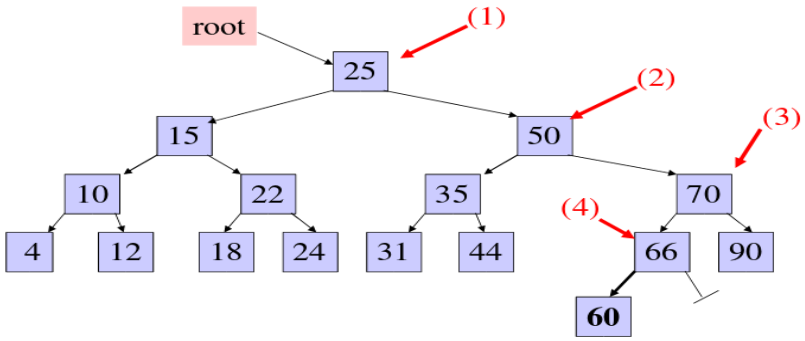


Inserting 24.



Example: insert 60 in the tree:

1. start at the root, 60 is greater than 25, search in right subtree
2. 60 is greater than 50, search in 50's right subtree
3. 60 is less than 70, search in 70's left subtree
4. 60 is less than 66, add 60 as 66's left child



Deletion

Remove operation on binary search tree is more complicated, than add and search. Basically, in can be divided into two stages:

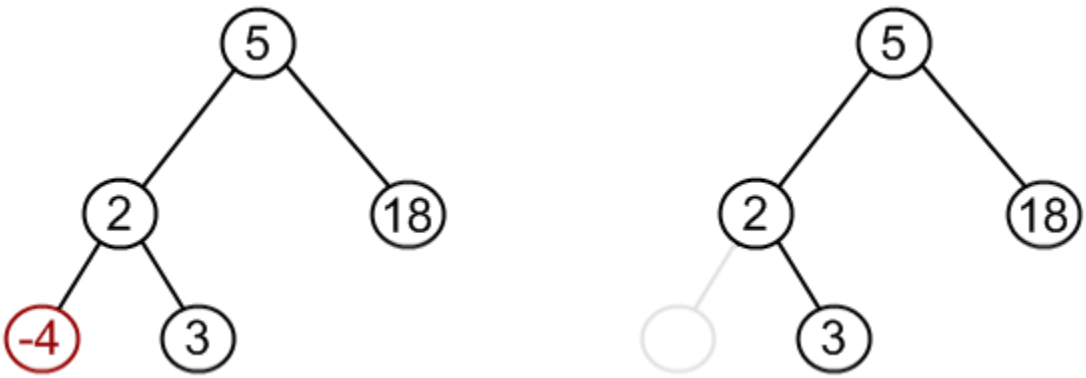
- search for a node to remove;

Now, let's see more detailed description of a remove algorithm. First stage is identical to algorithm for lookup, except we should track the parent of the current node. Second part is more tricky. There are three cases, which are described below.

1. Node to be removed has no children.

   This case is quite simple. Algorithm sets corresponding link of the parent to NULL and disposes the node.

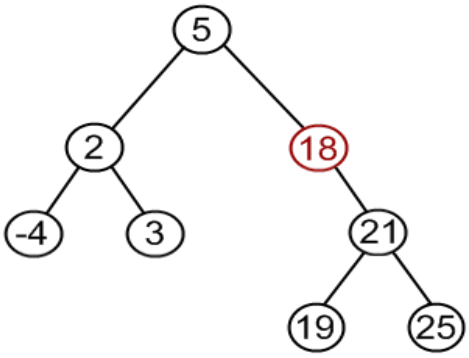
   Example. Remove -4 from a BST.

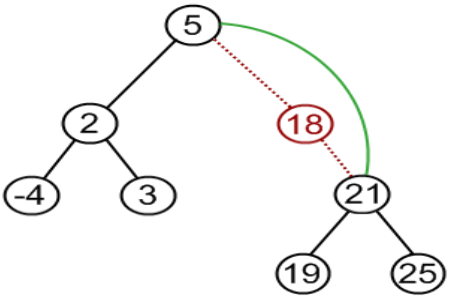


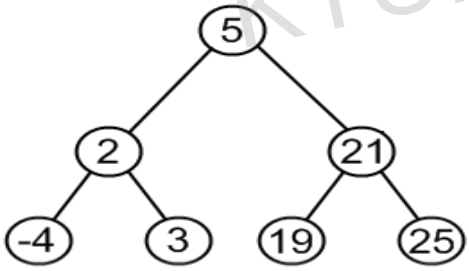
2. Node to be removed has one child.

It this case, node is cut from the tree and algorithm links single child (with it's subtree) directly to the parent of the removed node.

Example. Remove 18 from a BST.

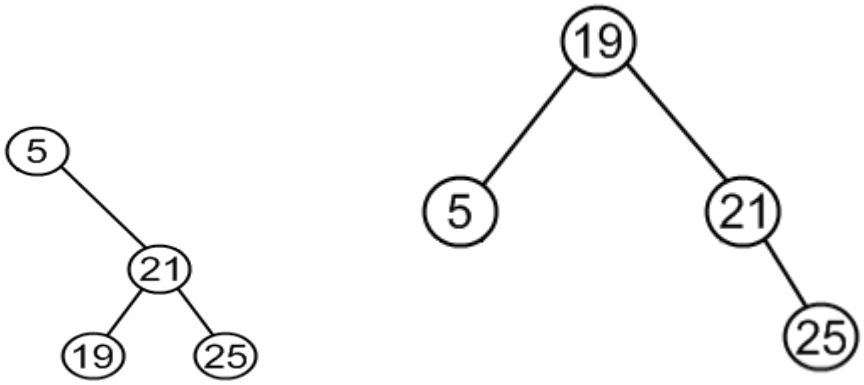






3. Node to be removed has two children.

This is the most complex case. To solve it, let us see one useful BST property first. We are going to use the idea, that the same set of values may be represented as different BST



contains the same values {5, 19, 21, 25}. To transform first tree into second one, we can do following:

- choose minimum element from the right subtree (19 in the example);
- replace 5 by 19;
- hang 5 as a left child.

The same approach can be utilized to remove a node, which has two children:

- find a minimum value in the right subtree;
- replace value of the node to be removed with found minimum. Now, right subtree
- apply remove to the right subtree to remove a duplicate.

Notice, that the node with minimum value has no left child and, therefore, it's removal may result in first or second cases only.

Example. Remove 12 from a BST.

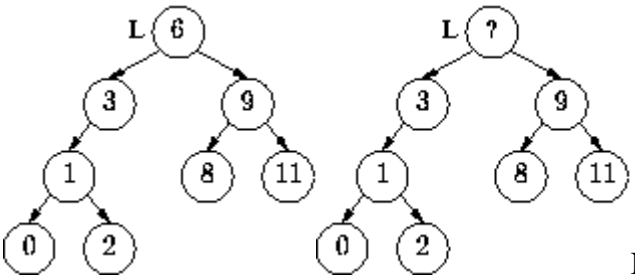
1) ***Node to be deleted is leaf:*** Simply remove from the tree.

```
          50                            50
        /    \         delete(20)       /   \
       30     70       --------->     30     70
      /  \    /  \                      \    /  \
    20   40  60   80                     40  60   80
```

2) ***Node to be deleted has only one child:*** Copy the child to the node and delete the child

```
          50                            50
        /    \         delete(30)       /   \
       30     70       --------->     40     70
         \    /  \                          /  \
         40  60   80                       60   80
```

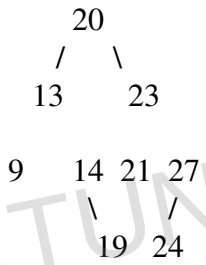
3) ***Node to be deleted has two children***



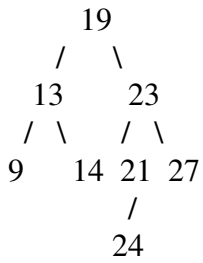
Now, what value can we move into the vacated node and have a binary search tree? Well, here's how to figure it out. If we choose value X, then:

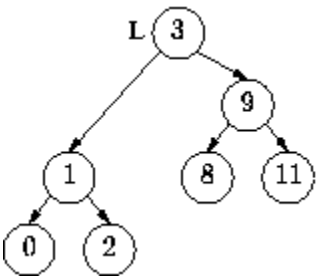
1. everything in the left subtree must be smaller than X.
2. everything in the right subtree must be bigger than X.

Let's suppose we're going to get X from the left subtree. (2) is guaranteed because everything in the left subtree is smaller than everything in the right subtree. What about (1)? If X is coming from the left subtree, (1) says that there is a unique choice for X - we must choose X to be the largest value in the left subtree. In our example, 3 is the largest value in the left subtree. So if we put 3 in the vacated node and delete it from its current position we will have a BST with 6 deleted.



Delete 20 from the above binary tree. Find the smallest in the left subtree of 20. So, replace 19 with 20.





Algorithm

```
void deletenode(node *tree,int digit)
{
struct node *r,*q;
if(tree==NULL)
{
print “Tree is empty”
exit(0);
}
if(digit<tree->num)
deletenode(tree->left,digit);
else if(digit>tree->num)
deletenode(tree->right,digit);
q=tree;
if((q->right==NULL)&&(q->left==NULL))
q=NULL;
else if(q->right==NULL)
tree=q->left;
else if (q->left==NULL)
tree=q->right;
else
delnum(q->left,digit);
free(q)
}
delnum(int struct node *r,int digit)
{
struct node * q;
if(r->right!=NULL)
delnum(r->right,digit);
```

Else

q->num=r->num;

q=r;

r=r->left;

}

Construct binary Tree from inorder and preorder traversal

The following procedure demonstrates on how to rebuild tree from given inorder and preorder traversals of a binary tree:

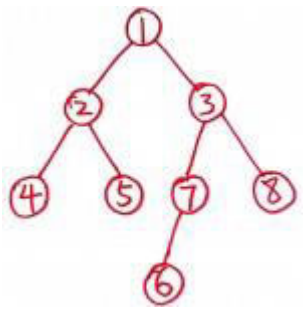
- Preorder traversal visits Node, left subtree, right subtree recursively
- Inorder traversal visits left subtree, node, right subtree recursively
- Since we know that the first node in Preorder is its root, we can easily locate the root node in the inorder traversal and hence we can obtain left subtree and right subtree from the inorder traversal recursively

Example 1:in-order: 4 2 5 (1) 6 7 3 8
pre-order: (1) 2 4 5 3 7 6 8
Construct binary tree

Soln: From the pre-order array, we know that first element is the root. We can find the root in in-order array. Then we can identify the left and right sub-trees of the root from in-order array.

Using the length of left sub-tree, we can identify left and right sub-trees in pre-order array. Recursively, we can build up the tree.

For this example, the constructed tree is:



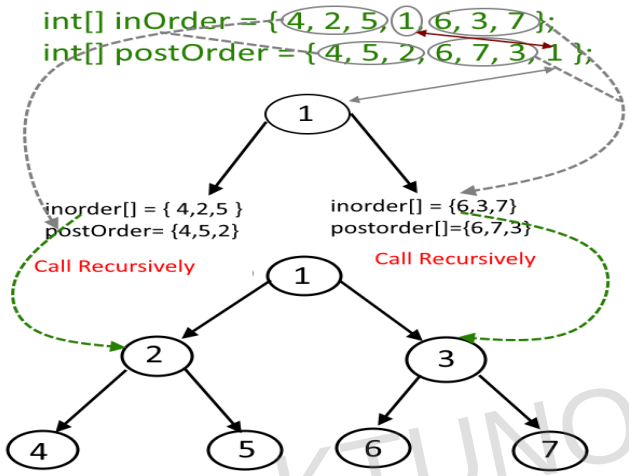
Example 2: inOrder = { 4, 2, 5, 1, 6, 3, 7 };

postOrder = { 4, 5, 2, 6, 7, 3, 1 };.Construct binary tree

Soln

- Last element in the *postorder []* will be the *root* of the tree, here it is 1.

- Now the search element 1 in *inorder[]*, say you find it at position *i*, once you find it, make note of elements which are left to *i* (this will construct the leftsubtree) and elements which are right to *i* (this will construct the rightSubtree).
- Suppose in previous step, there are X number of elements which are left of '*i*' (which will construct the leftsubtree), take first X elements from the postorder[] traversal, this will be the post order traversal for elements which are left to *i*. similarly if there are Y number of elements which are right of 'i' (which will construct the rightsubtree), take next Y elements, after X elements from the postorder[] traversal, this will be the post order traversal for elements which are right to *i*
- From previous two steps construct the left and right subtree and link it to root.left and root.right respectively.
- See the picture for better explanation.



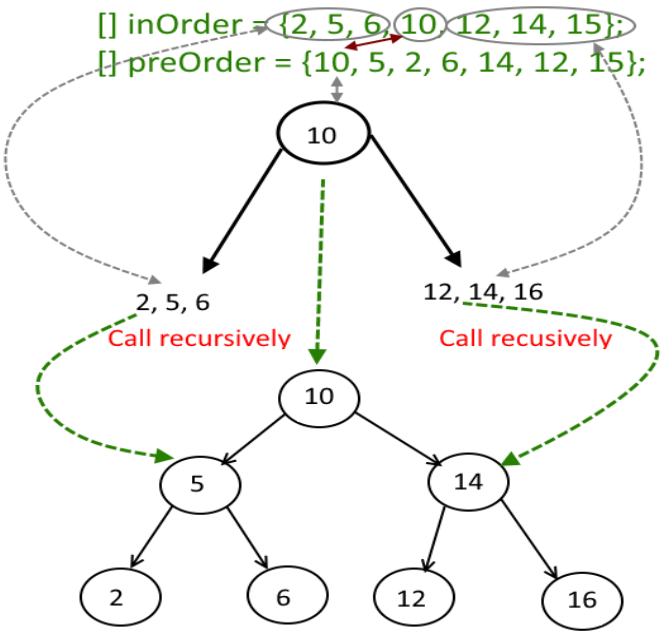
Example 3:inOrder = {2,5,6,10,12,14,15};

preOrder = {10,5,2,6,14,12,15};Construct binary tree

Soln

- First element in *preorder[]* will be the *root* of the tree, here its 10.
- Now the search element 10 in *inorder[]*, say you find it at position *i*, once you find it, make note of elements which are left to *i* (this will construct the leftsubtree) and elements which are right to *i* (this will construct the rightSubtree).
- See this step above and recursively construct left subtree and link it root.left and recursively construct right subtree and link it root.right.

- See the picture and code.



Example 4: Preorder Traversal: 1 2 4 8 9 10 11 5 3 6 7

Inorder Traversal: 8 4 10 9 11 2 5 1 6 3 7

Construct binary tree

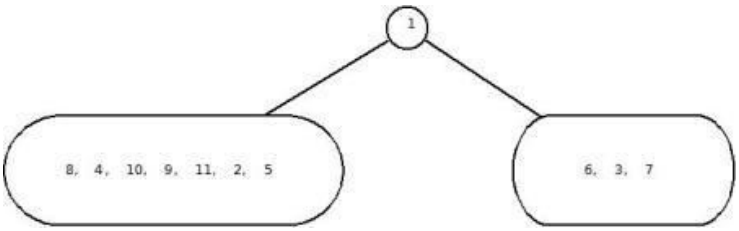
Soln

Iteration 1:

Root – {1}

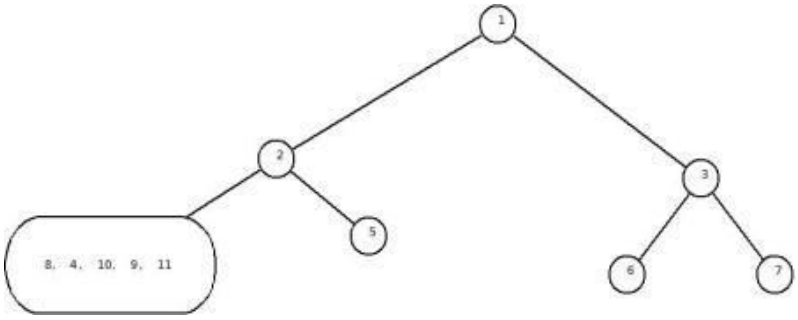
Left Subtree – {8,4,10,9,11,2,5}

Right Subtree – {6,3,7}



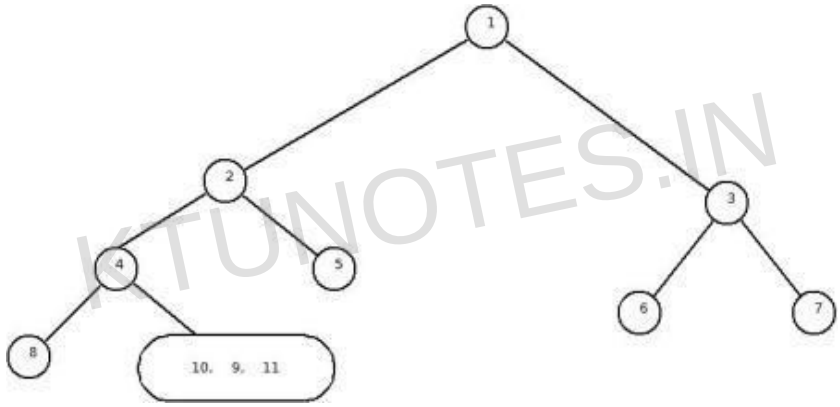
Iteration 2:

Root – {2} Left Subtree – {8,4,10,9,11} Right Subtree – {5}	Root – {3} Left Subtree – {6} Right Subtree – {7}



Iteration 3:

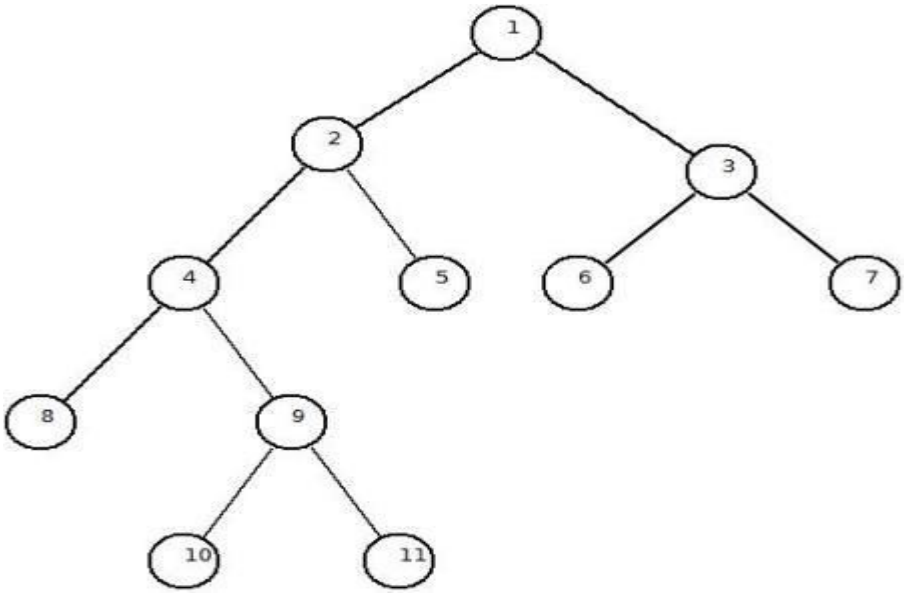
Root – {2} Left Subtree – {8,4,10,9,11} Right Subtree – {5}		Root – {3} Left Subtree – {6} Right Subtree – {7}
Root – {4} Left Subtree – {8} Right Subtree – {10,9,11}	Done	Done



Iteration 4:

Root – {2} Left Subtree – {8,4,10,9,11} Right Subtree – {5}		Root – {3} Left Subtree – {6} Right Subtree – {7}
Root – {4} Left Subtree – {8} Right Subtree – {10,9,11}	Done	Done

Done	R – {9} Left ST – {10} Right ST- {11}	Done	Done



The following are the two versions of programming solutions even though both are based on above mentioned algorithm:

- Creating left preorder, left inorder, right preorder, right inorder lists at every iteration to construct tree
- Passing index of preorder and inorder traversals and using the same input list to construct tree

Advantages of trees

Trees are so useful and frequently used, because they have some very serious advantages:

- Trees reflect structural relationships in the data
- Trees are used to represent hierarchies
- Trees provide an efficient insertion and searching
- Trees are very flexible data, allowing to move subtrees around with minumum effort

Application of tree

1.Manipulate hierarchical data.

2.Make information easy to search (see tree traversal).

3. Manipulate sorted lists of data.

4. As a workflow for compositing digital images for visual effects.

5.Router algorithms

6.Windows file system