UNIT-IV TREES:

BNR

* In linear data structure, data is organized in sequential order and in non-linear data structure, data is organized in random order.

Tree: Tree is a non-linear data structure which organizes data in hierarchical structure and this is a recursive definition.

(or) Tree is a non-linear data structure which allows to associate a parent and child relationship between various data to arrange in a hierarchical.

* consider a tree representing your family structure:

you start with your grand parent then come to your parent and finally you and your siblings, so the example of trees are generalized and organization chart

* An organization's structure is another example of a hierarchy.
* The trees are used to help and analyze the electrical circuit and to represent the structure of mathematical formulas.
* The trees are used to organize information in database systems and to represent the syntactic structure of source programs in compilers.
* A tree data structure is having upside down and often called inverted trees because they are normally drawn with the root at the top and the leaves are at the bottom.
* The first node of the tree is called the root that has no parent, it can have only child nodes.
* If this root node is connected by another node, the root is then a parent node and the connected node is a child.

* The leaves on the other hand have no children in tree struct-re

* Tree can be defined recursively in the following manner:

1. A single node by itself is a tree, this node is called the root node of the tree.

2. Suppose n is a node and trees $T_1, T_2, T_3, ----, T_K$ with roots $n_1, n_2, n_3, ----, n_K$ respectively, a new tree may be constructed by making n the parent of $n_1, n_2, n_3, ----, n_K$.

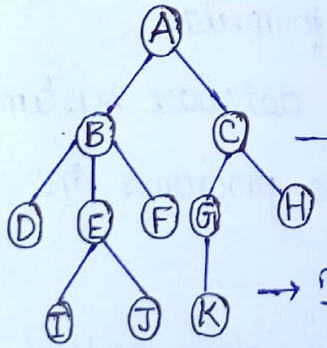
3. In this tree, n is the root, $T_1, T_2, T_3, ---T_K$ are the sub trees.

4. The nodes $n_1, n_2, n_3, ----, n_K$ are the child nodes of n.

* 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 structur

→ Trees consists of nodes which are connected by edges.

* In a tree data structure, if we have N number of nodes then we can have a maximum of N-1 number of links.



Tree with 11 Nodes and 10 edges.

→ In any tree with 'N' Nodes there will be maximum of 'N-1' edges.

→ In a tree every individual element is called as Node

Examples of a trees:

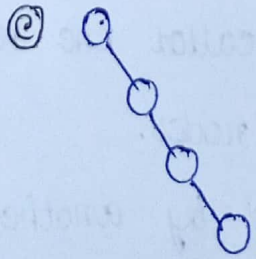
(a) It is a empty tree

(b)

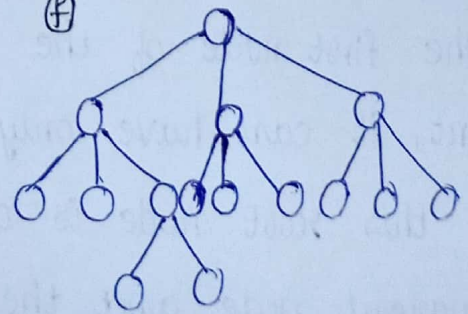
(c)

(d)

(e)



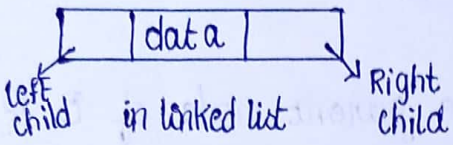
(f)



Basic terminologies of trees:

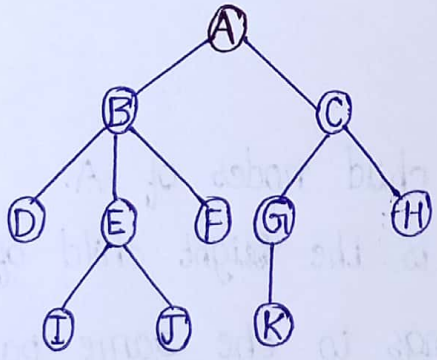
1. Node: We use the term node rather than vertex in our trees. This is the main component in tree structure, it stores the actual data along with link to other nodes.

Ex: (data) in arrays



in linked list

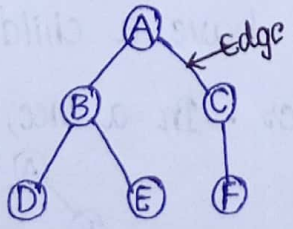
2. Root: A root is a special designated node in a tree structure, it is a node which has no parent. Every tree must have root node. In a tree data structure, the first node is called as Root node. We can say that root node is the origin of tree data structure. In any tree, there must be only one root node. We never have multiple root nodes in a tree.



Here 'A' is the 'root' node.

3. Edge (or) Branch: It is a connecting link between any two nodes. A node can have more than one edge.

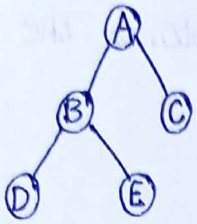
In a tree with 'N' number of nodes, there will be maximum of 'N-1' number of edges.



Node A has 2 edges or 2 branches.

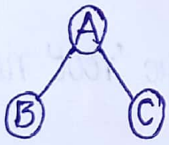
Node B has 2 branches & Node C has 1 branch.

4. parent: The parent of a node is the immediate predecessor of that node. parent is a node that has an edge to a child node.



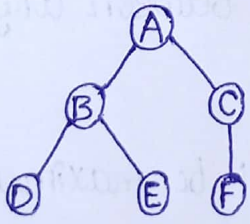
In above tree Structure, A is the parent node of B & C.
B is the parent node of D & E.

5. child: The immediate successor of a node are called child nodes. A child which is placed at left sides is called left child and a child which is placed at the right side is called right child of a node. Any parent node can have any number of child nodes. In a tree all nodes except root node are child nodes.



In the above tree, B and C are child nodes of A.
B is the left child of A and C is the right child of A.

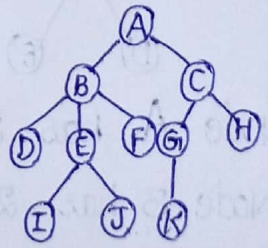
6. Siblings: The nodes which belongs to the same parent are called as siblings.



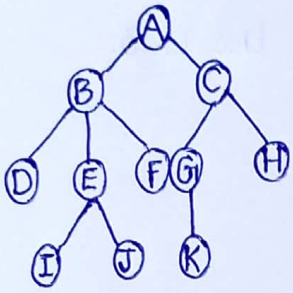
Here, B & C are Siblings. D & E are siblings.

7. Leaf: Leaf is a node that does not have a child node in the tree. Leaf nodes are also called as external Nodes. In a tree, leaf node is also called as 'Terminal' node.

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

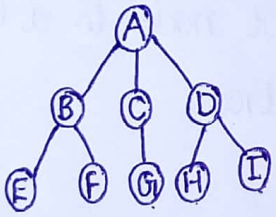


8. Internal Nodes: The node which has atleast one child is called internal node (or) The nodes other than leaf nodes are called internal nodes
* The root node is also said to be internal node, if the tree has more than one node. Internal nodes are also called as 'Non-terminal' nodes



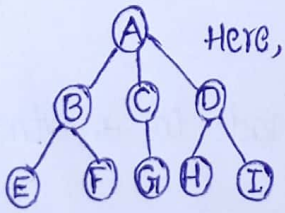
Here, A, B, C, E, G are internal nodes.
Every non-leaf node is called as 'internal' node.

9. Degree of Node: The number of edges connecting to a particular node is called degree of Node (or) Total no. of children of a node is called as degree of a node.



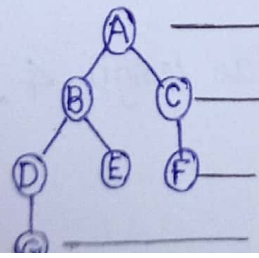
Here, Degree of Node A is 3.
Degree of Node B is 2.
Degree of Node C is 1.
Degree of Node D is 2.
Degree of E, F, G, H, I is 0.

10. Degree of a tree: The highest degree of a node among all the nodes in a tree is called as 'Degree of Tree'.



Here, Maximum degree of a node is 3. So this is the 'Degree of Tree'.

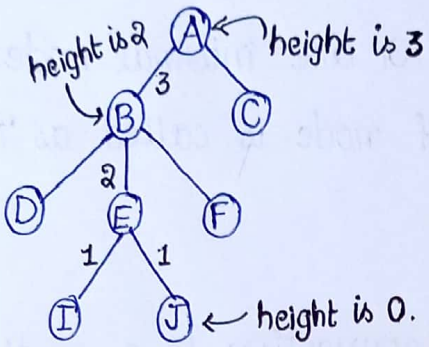
11. Level: In a tree data structure, the root node is said to be at level 0 and the children of root node are at level 1 and the children of the nodes which are at level 1 will be at level 2 and so on



Here, the root node 'A' at level 0, B and C are at level 1; D, E, F are at level 2; G is at level 3.

12. Height: The total number of edges from leaf node to a particular node in the longest path is called as height of that Node

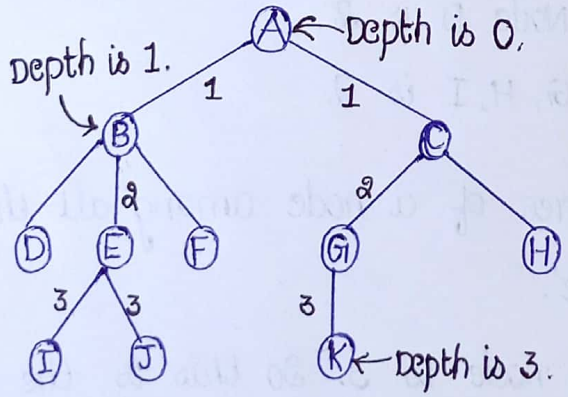
* In a tree, height of the root Node is said to be height of the tree.
* In a tree, height of all leaf nodes is '0'.



Here, Height of tree is 3.

13. Depth: The total number of edges from root node to a particular node is called as depth of that Node.

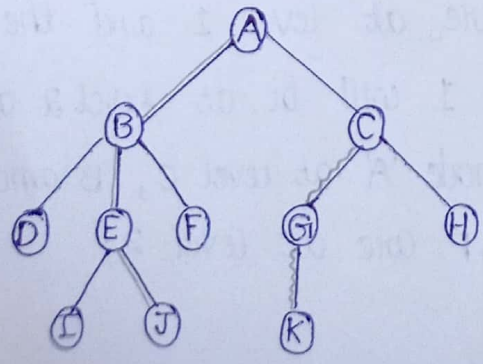
* In a tree, the total number of edges from root node to a leaf node in the longest path is said to be Depth of the tree.
* In a tree, depth of root node is '0'.



Here, Depth of tree is 3.

14. path: 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.

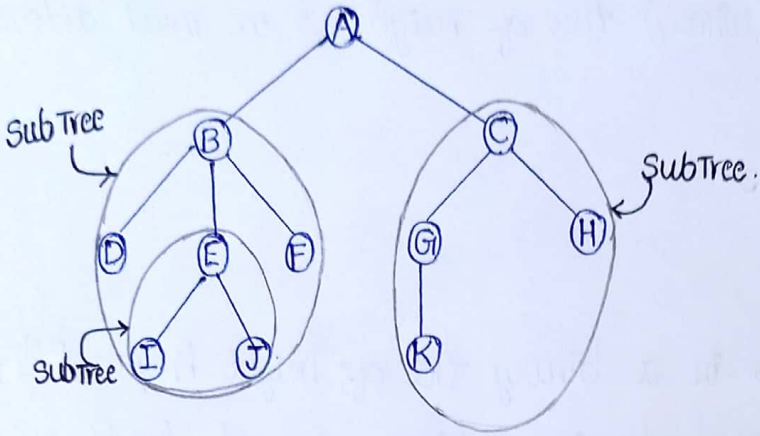


Here, path between A & J is A-B-E-J

path between C & K is C-G-K.

* The path A-B-E-J has length 4.

15. Sub Tree: In a tree data structure, each child from a subtree recursively. Every child node will form a subtree on its parent node.

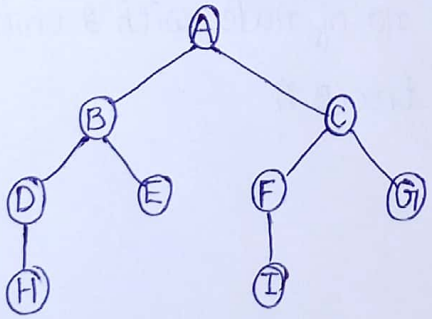


Binary Tree:

In a normal tree, every node can have any number of children. Binary tree is a special type of tree data structure in which every node can have a maximum of 2 children. one is known as left child and the other is known as right child.

* In a binary tree, every node can have either 0 children or 1 child or 2 children but not more than 2 children.

Example:



properties of a Binary Tree:

* The maximum number of nodes at level 'l' of a binary tree is 2^{l-1}.

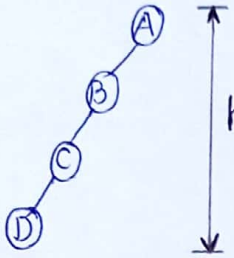
→ For root ; $l = 1 \Rightarrow$ number of nodes $= 2^{l-1} = 2^{1-1} = 2^0 = 1$.

Here level is number of nodes on path from root to the node (including root and node). Level of root is 1.

Since in Binary tree every node has atmost 2 children, next level would have twice nodes i.e., 2

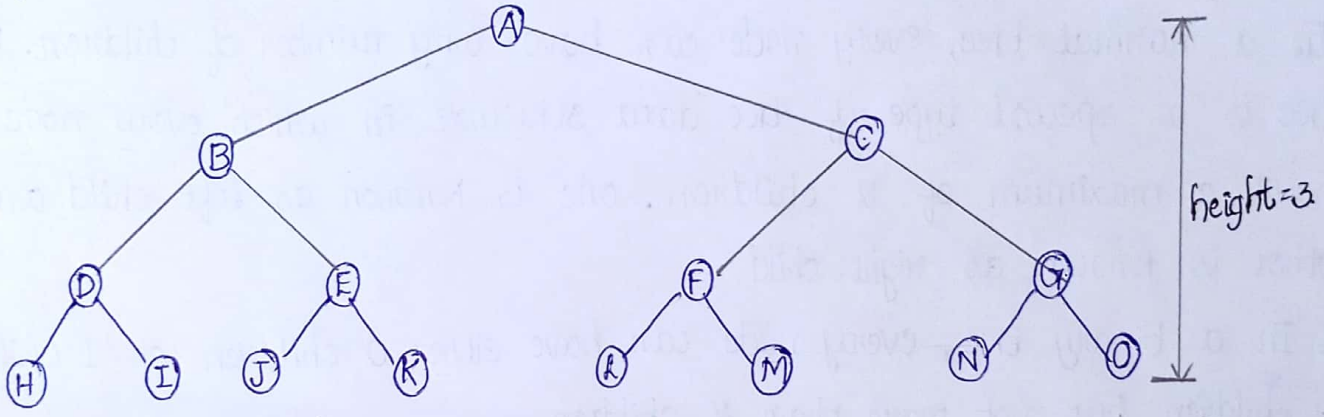
* Maximum number of nodes in a binary tree of height 'h' is $2^{h+1}-1$.
* Minimum number of nodes in a binary tree of height h is h+1.

for example; to construct a binary tree of height=3 we need atleast 3+1=4 nodes



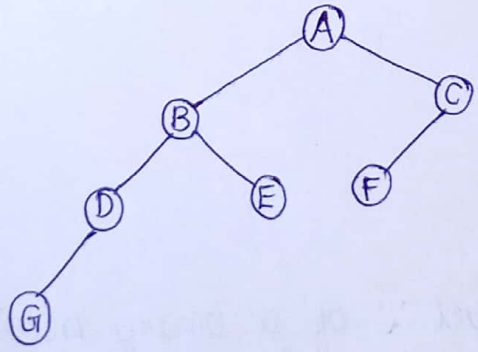
* Maximum number of nodes in a binary tree of height h is $2^{h+1}-1$.

for example: Maximum number of nodes in a binary tree of height 3 = $2^{3H}-1=15$ nodes



* Total number of leaf nodes in a B.T = Total no. of nodes with 2 children + 1.

for example: consider the following Binary tree (B.T)



Here, No. of leaf nodes = 3.

No. of nodes with 2 children = 2 (i.e., A & B).

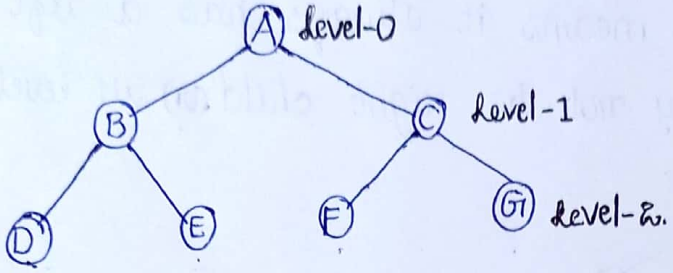
clearly, number of leaf nodes is one greater than number of nodes with 2 children.

* If a binary tree has 'n' no. of ~~tree~~ nodes then no. of edges are 'n-1'.

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

For example:

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



*

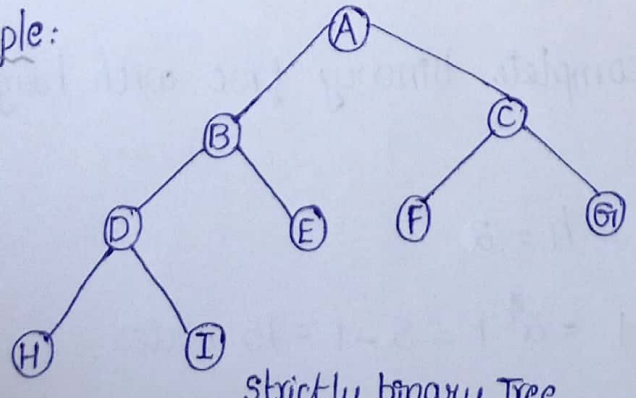
Types of Binary Trees: There are different kinds of binary trees such as

1. Full Binary Tree / strictly Binary Tree
2. Almost complete Binary Tree / Incomplete Binary Tree
3. complete Binary Tree / perfect Binary tree
4. Left skewed binary tree.
5. Right skewed binary tree.

1. Full / strictly Binary Tree:

Every node must have two children except the leaf nodes (or) if and only if each node has exactly two child nodes (or) no nodes.

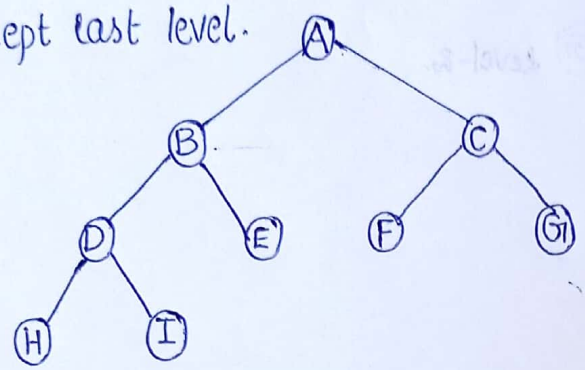
Example:



Strictly binary Tree

2. Almost / Incomplete Binary Tree:

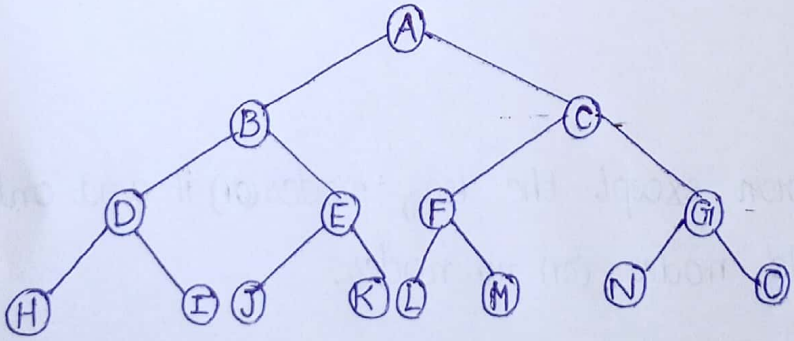
Every node must have two children in all levels except in last level but filled from left to right. (or) each node has left child whenever it has a right child. That means it always has a left child but for a left child there may not be right child (or) all levels are entirely filled except last level.



3. complete / perfect Binary Tree:

In complete binary tree, all the nodes must have exactly two children and at every level of complete binary tree there must be 2^{level} number of nodes. For example at level 2 there must be $2^2 = 4$ nodes and at level 3 there must be $2^3 = 8$ nodes.

* A binary tree in which every internal node has exactly two children and all leaf nodes are at same level is called complete Binary Tree.



level 0 $\Rightarrow 2^0 = 1$ node

level 1 $\Rightarrow 2^1 = 2$ nodes

level 2 $\Rightarrow 2^2 = 4$ nodes

level 3 $\Rightarrow 2^3 = 8$ nodes.

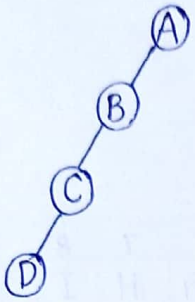
* The total number of nodes in complete binary tree with height 'h' is $2^{h+1} - 1$.

In above tree, height of the tree = $h = 3$.

$\therefore$ Total no. of nodes = $2^{h+1} - 1 = 2^{3+1} - 1 = 2^4 - 1 = 8 - 1 = 15$ nodes.

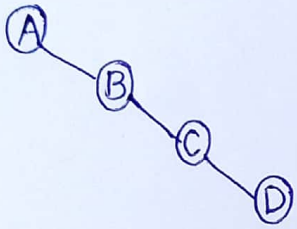
4. Left skewed binary Tree:

The binary tree is left skewed binary tree in which each node is having its left child only.



5. Right skewed Binary Tree:

The binary Tree is right skewed binary tree in which each node is having its right child only.



Binary Tree representation:

Binary tree can be represented by using following two ways:

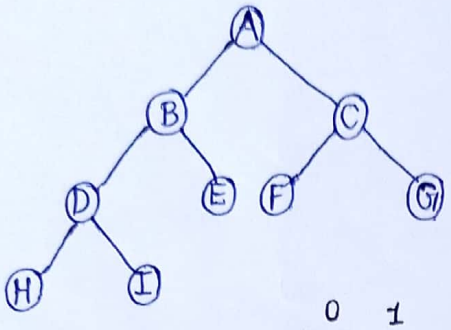
1. Array representation
2. Linked list representation

1. Array representation:

* In array representation of binary tree, we use a one dimensional array (1-D Array) to represent a binary tree.
* This type of representation is static means that a block of memory for an array is to be allocated before going to store the actual data
* In this representation the nodes of the tree are stored level by level, starting from the root node.
* To represent a binary tree of depth 'n' using array representation, we need one dimensional array of maximum size of $2^{n+1}-1$.

* There are two cases for array representation of binary tree:

Case-I: *consider the following binary tree:



Array representation is :

0	1	2	3	4	5	6	7	8
A	B	C	D	E	F	G	H	I

If the node is at i^{th} index:

→ Left child would be at = $[(2*i)+1]$

→ Right child would be at = $[(2*i)+2]$

→ parent would be at = $\left[\frac{(i-1)}{2}\right]$.

Example: ① Let i = 4 i.e., node E.

For i = 4 the left child would be at = $[(2*i)+1] = [(2*4)+1] = 9$

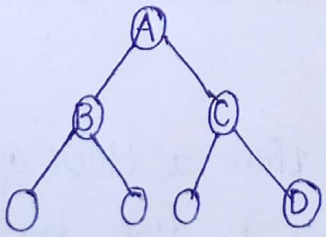
But there is no 9th position i.e., node E has no left child.

② Let i = 6 i.e., node G.

parent node would be at = $\left[\frac{(i-1)}{2}\right] = \left[\frac{6-1}{2}\right] = [2.5]$ (Floor value will be considered)

So, i = 2 i.e., node C is the parent node of the node G.

* consider the following binary tree:



Array representation is :

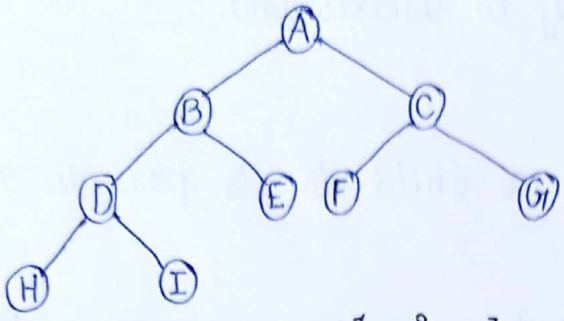
0	1	2	3	4	5	6
A	B	C	-	-	-	D

For i = 2 i.e., node C ; the right child would be at = $[(2*i)+2]$

= $[(2*2)+2] = 6$ i.e., node D.

∴ Node D is the right child of node C.

case - II : * consider the following binary tree:



Array representation is:

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

If a node is at index i :

→ left child would be at (i) = (2*i)

→ Right child would be at (i) = [(2*i)+1]

→ parent at (i) = $\left[\frac{i}{2}\right]$.

Example: ① For the above binary tree;

Let i = 5 i.e., node E

The parent node (i) = $\left[\frac{i}{2}\right] = \left[\frac{5}{2}\right] = [2.5] = 2$ (since floor value is considered)

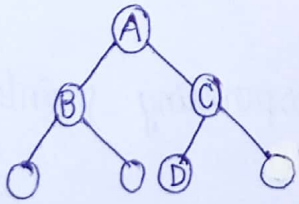
i.e., Node B is the parent node of node E.

② Let i = 4 i.e., node D

The right child would be at (i) = [(2*i)+1] = [(2*4)+1] = 9 i.e., node I.

∴ I node is the right child of node D.

* consider the following binary tree:



Array representation is:

1	2	3	4	5	6	7
A	B	C	-	-	D	-

For i = 3 i.e., node C

The left child would be at (i) = (2*i) = (2*3) = 6 i.e., node D.

∴ D is the left child of node C.

Advantages:

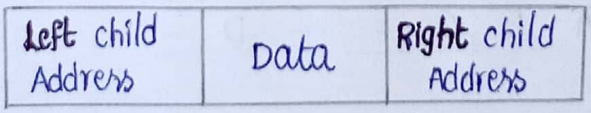
* This representation is very easy to understand.
* Programming is very easy.
* It is very easy to move from a child to its parents and vice versa.
* This is the best representation for complete and full binary tree representation.
* The data is stored without any pointers.

Disadvantages:

* Most of the array entries are empty means that implies lot of memory area wasted.
* Insertion and deletion of nodes needs a lot of data movement.
* Execution time high.
* This is not suited for trees other than full and complete tree.

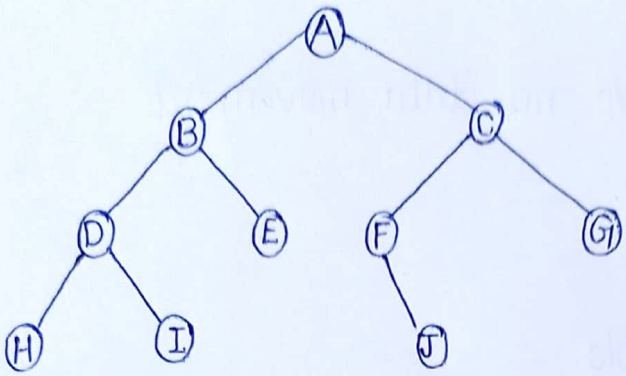
2. Linked list representation:

* we use 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.
* when a node has no children, the corresponding pointer fields (left, right) stores NULL.

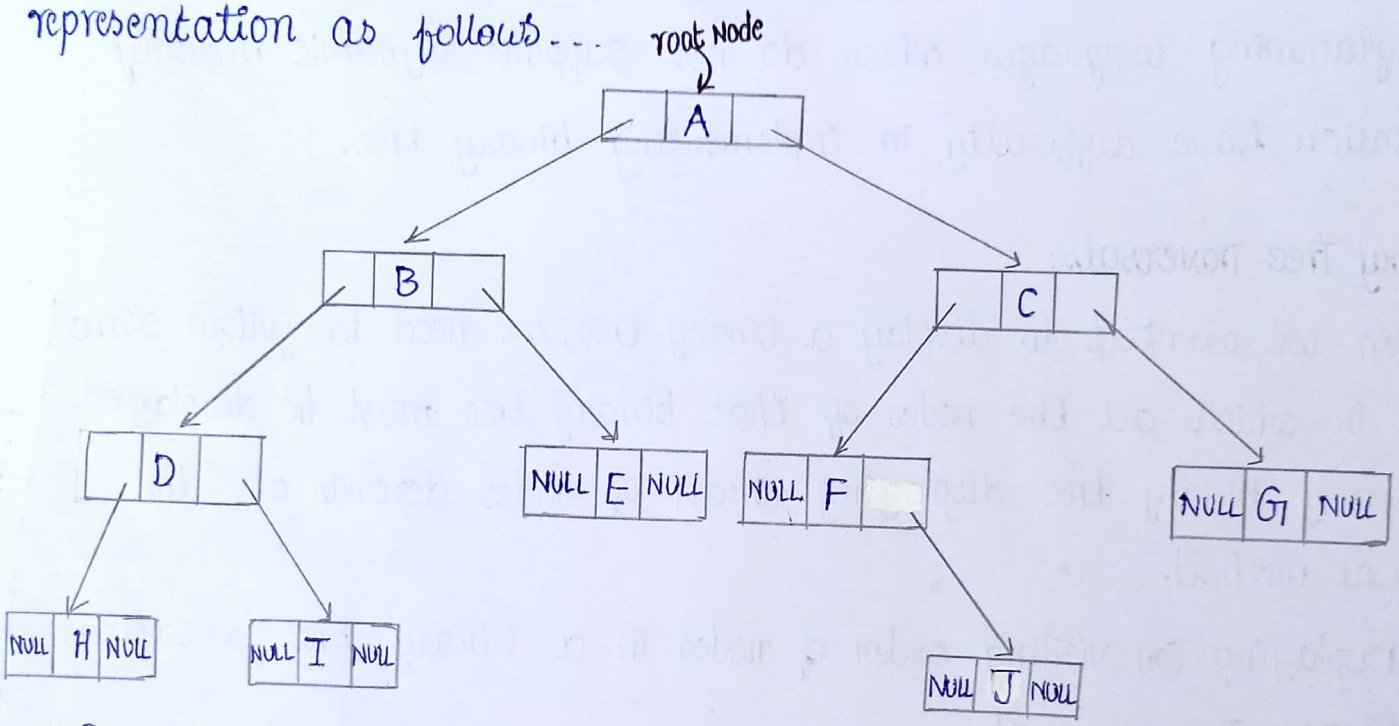


Node (link representation)

consider the following binary tree:



The above example of binary tree represented using linked list representation as follows...



* In this we should declare a structure for tree node

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

* This type of representation is dynamic means that a block of memory required for a tree need not to be allocated before. they are allocated only on demand.

Advantages:

* No wastage of memory.
* Insertion and deletion involve no data movement.
* Dynamic memory allocation.

Disadvantages:

* Random access is not possible.
* In this pointer fields occupy more space than data field.
* programming languages which do not support dynamic memory allocation have difficulty in implementing binary tree.

Binary Tree Traversals:

* when we wanted to display a binary tree, we need to follow some order in which all the nodes of that binary tree must be displayed.
* In any binary tree displaying order of nodes depends on the traversal method.

→ Displaying (or) visiting order of nodes in a binary tree is called as Binary Tree Traversal.

There are three types of binary tree traversals.

1. In-order traversal
2. pre-order traversal
3. post-order traversal.

1. In-order Traversal (left-Root-right):

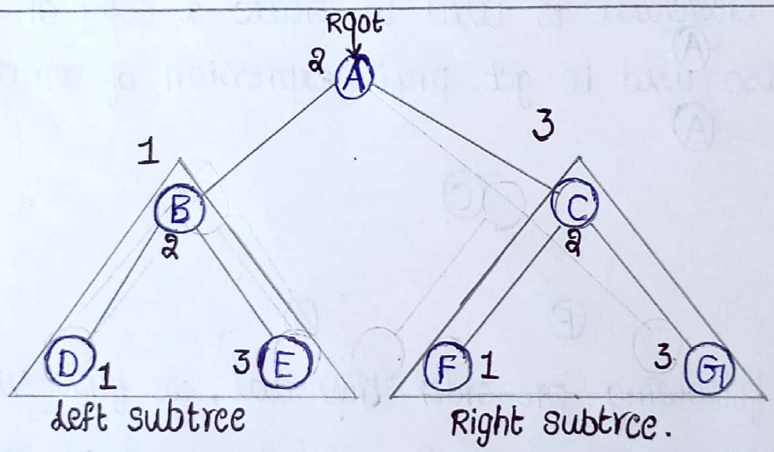
In this traversal method, the left subtree is visited first, then the root and later the right subtree.

* 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.

→ this is performed recursively for all nodes in the tree.

Inorder Traversal Algorithm:

Algorithm Inorder (tree)
1. Traverse the left subtree, i.e., call Inorder (left-subtree)
2. Visit the root
3. Traverse the right subtree, i.e., call Inorder (right-subtree)



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 nodes are visited. The output of in-order traversal of this tree will be

$$D \rightarrow B \rightarrow E \rightarrow A \rightarrow F \rightarrow C \rightarrow G.$$

Function to create In-order traversal:

```
void inorder_traversal(struct node* root) {
    if (root != NULL) {
        inorder_traversal(root->leftChild);
        printf("%d ", root->data);
        inorder_traversal(root->rightChild);
    }
}
```

2. Pre-order Traversal (Root-left-right):

→ Visit the root node.

→ Traverse the left subtree of the root

→ Traverse the right subtree of the root

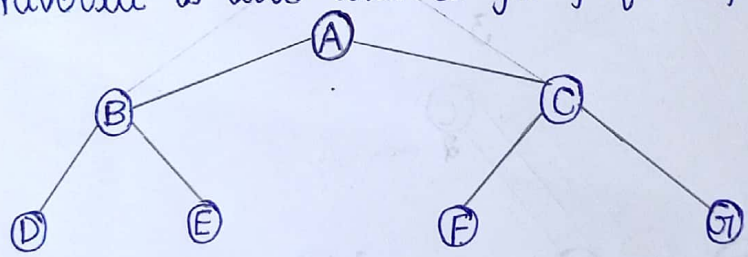
* This pre-order traversal is applicable for every root node for all subtrees in the tree.

* preorder Tree Traversal Algorithm:

```
Algorithm Preorder (tree)
1. Visit the root
2. Traverse the left subtree i.e, call Preorder (left-subtree)
3. Traverse the right subtree i.e, call Preorder (right-subtree)
```

Uses of preorder: preorder traversal is used to create a copy of the tree.

* preorder traversal is also used to get prefix expression of an expression tree.



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 nodes are visited. The output of pre-order traversal of this tree will be

A → B → D → E → C → F → G

Function to create pre-order traversal:

```
void preorder_traversal (struct node* root) {
    if (root != NULL) {
        printf ("%d ", root → data);
        preorder_traversal (root → leftchild);
        preorder_traversal (root → rightchild);
    }
}
```

3. post-order Traversal (left-right-root):

→ Traverse the left subtree of the root

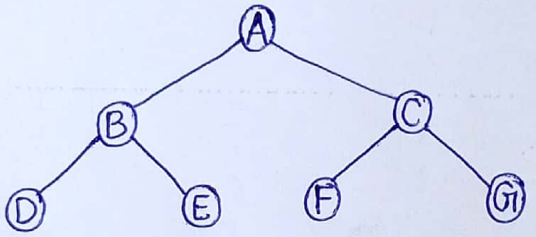
→ Traverse the right subtree of the root

→ visit the root node.

* This is recursively performed until the right most node is visited.

* post-order Traversal Algorithm:

```
Algorithm postorder(tree)
1. Traverse the left subtree, i.e., call postOrder (left-subtree)
2. Traverse the right subtree, i.e., call postOrder(right-subtree)
3. visit the tree
```



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 nodes are visited. The output will be D → E → B → F → G → C → A.

Function to create post-order traversal:

```
void postorder_traversal(struct node* root) {
    if (root != NULL) {
        postorder_traversal(root → leftChild);
        postorder_traversal(root → rightChild);
        printf("%d ", root → data);
    }
}
```

Uses of post-order:

* postorder traversal is used to delete the tree.

* postorder traversal is also useful to get the postfix expression of an expression tree.

Create a binary tree from Inorder, postorder, preorder traversals:

→ We are creating a binary tree with the help of these tree traversals.

→ Here tree traversals output has been given and we need to construct a binary tree.

→ From a single traversal it is not easy to construct a binary tree. If two traversals are known then the tree can be constructed easily.

→ The basic principles for creating a binary tree is stated as follows:

1. If the pre-order traversal is given then the first node is given to construct.
2. If the postorder traversal is given then the last node is given to construct.
3. Once the root node is identified, then all the nodes in the left sub tree and right sub tree of root node can be easily identified.

NOTE:

* For creating a binary tree two traversals are required out of which one is Inorder traversal and another one is either pre-order or postorder traversal.
* The preorder and postorder traversals are given, then it is not possible to construct a binary tree.

→ There are two ways of constructing a binary tree. They are

1. From preorder and Inorder traversals
2. From postorder and Inorder traversals.

1. construction of binary tree from preorder and Inorder traversals:

The general procedure for creating a binary tree is as follows:

step-1: Scan the preorder traversal from left to right

step-2: For each node scanned locate its position in inorder traversal. Let the scanned node be X.

Step-3: The node X preceeding X inorder form its left subtree and nodes succeeding to it forms right subtree.

Step-4: Repeat step-1 for each node in the preorder.

Algorithm:

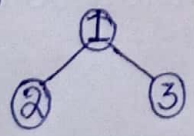
1. Take 1st element in pre-order traversal and make it a root node.
2. search for the element obtained from step-1 in inorder traversal.
3. Take left part of inorder traversal and repeat steps 1 & 2 with the next element in preorder traversal as a root.
4. Take right part of inorder traversal and repeat steps 1 & 2 with the next element in preorder traversal as a root.

Example: *Input:

In[] = {2, 1, 3}

pre[] = {1, 2, 3}

output:



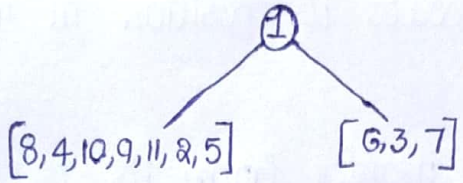
* Let us see the process of constructing tree from given Inorder and preorder traversals:

preorder = {1, 2, 4, 8, 9, 10, 11, 5, 3, 6, 7} (root-Left-right)

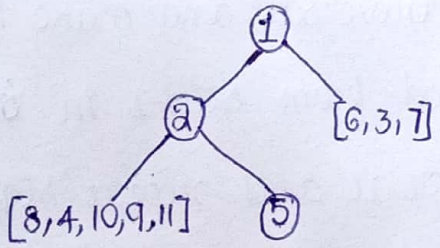
inorder = {8, 4, 10, 9, 11, 2, 5, 1, 6, 3, 7} (Left-root-right)

① The first element in preorder traversal is 1, therefore 1 is root node.

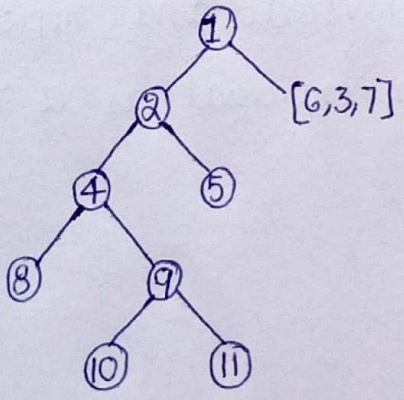
② we search "1" in inorder traversal to find left and right subtrees of root. Everything on left of 1 in inorder is in left subtree and everything on right is in right subtree.



③ In the left subtree the root node is the element which comes first in preorder when we scan from left to right. So element 2 is root node of left subtree. All elements on left of 2 in inorder is in left subtree and all elements on right of 2 in inorder is in right tree.

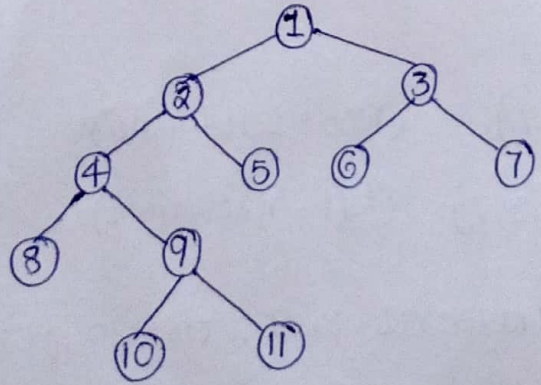


④ we recurse the above process for the subtrees...



⑤ In the subtree [6,3,7] the root node will be 3. From inorder traversal the left child node will be 6 and right child node will be 7.

∴ output of given inorder and preorder traversals will be



2. constructing of binary tree from postorder and inorder traversals:

Algorithm:

Step-1: Scan the postorder traversal from right to left.

Step-2: For each node scanned, locate its position in inorder traversal. Let the scanned node be X.

Step-3: The node preceeding of x in inorder forms its left sub tree and the nodes succeeding of x as forms its right sub tree.

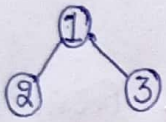
Step-4: Repeat step-1 for each node in the postorder.

Example: * Input

in[] = {2, 1, 3}

post[] = {2, 3, 1}

output:



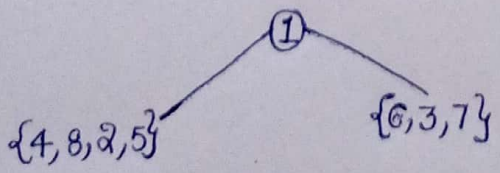
* Let us see the process of constructing tree from given inorder and postorder traversals:

postorder = {8, 4, 5, 2, 6, 7, 3, 1} (left-right-root)

inorder = {4, 8, 2, 5, 1, 6, 3, 7} (left-root-right)

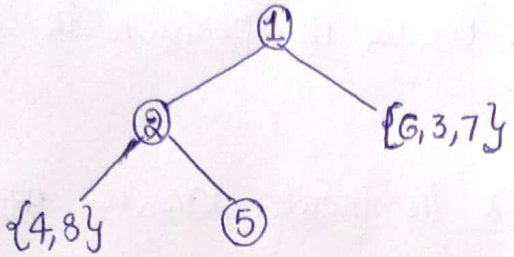
① We first find the last node in postorder. The last node is 1. We know this value is root as root always appear in the end of postorder traversal.

② We search "1" in inorder to find left and right subtrees of root. Everything on left of 1 in inorder is in left subtree and everything on right of 1 is in Right subtree.

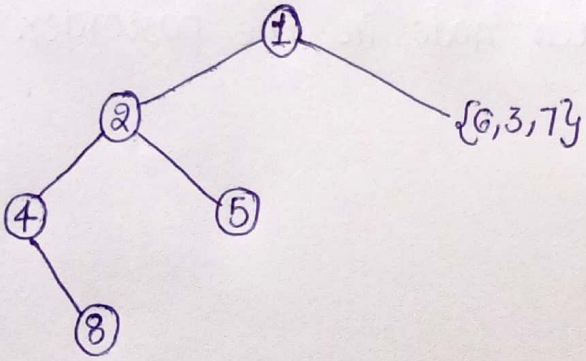


③ In the left subtree the root node is the element which comes first in

postorder when we scan from right to left. So element 2 is root node of left subtree. All elements on left of 2 in inorder is in left subtree and all elements on right of 2 in inorder is in right subtree.

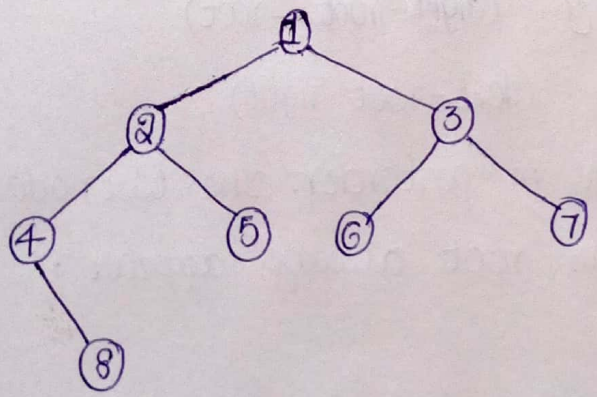


④ we recurse the above process for the subtrees...



⑤ In the subtree {6, 3, 7} the root node will be 3. From inorder traversal the left node will be 6 and right node will be 7.

∴ output of given inorder and postorder traversals will be



Implementation in C

Live Demo

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

struct node {
   int data;

   struct node *leftChild;
   struct node *rightChild;
};

struct node *root = NULL;

void insert(int data) {
   struct node *tempNode = (struct node*) malloc(sizeof(struct node));
   struct node *current;
   struct node *parent;

   tempNode->data = data;
   tempNode->leftChild = NULL;
   tempNode->rightChild = NULL;

   //if tree is empty
   if(root == NULL) {
      root = tempNode;
   } else {
      current = root;
      parent = NULL;

      while(1) {
         parent = current;

         //go to left of the tree
         if(data < parent->data) {
            current = current->leftChild;

            //insert to the left
            if(current == NULL) {
               parent->leftChild = tempNode;
               return;
            }
         }  //go to right of the tree
         else {
            current = current->rightChild;

            //insert to the right
            if(current == NULL) {
               parent->rightChild = tempNode;
               return;
            }
         }
      }
   }
}

struct node* search(int data) {
   struct node *current = root;
   printf("Visiting elements: ");

   while(current->data != data) {
      if(current != NULL)
         printf("%d ",current->data);

      //go to left tree
      if(current->data > data) {
         current = current->leftChild;
      }
      //else go to right tree
      else {
         current = current->rightChild;
      }

      //not found
      if(current == NULL) {
         return NULL;
      }
   }

   return current;
}

void pre_order_traversal(struct node* root) {
   if(root != NULL) {
      printf("%d ",root->data);
      pre_order_traversal(root->leftChild);
      pre_order_traversal(root->rightChild);
   }
}

void inorder_traversal(struct node* root) {
   if(root != NULL) {
      inorder_traversal(root->leftChild);
      printf("%d ",root->data);
      inorder_traversal(root->rightChild);
```

```
      inorder_traversal(root->rightChild);
   }
}

void post_order_traversal(struct node* root) {
   if(root != NULL) {
      post_order_traversal(root->leftChild);
      post_order_traversal(root->rightChild);
      printf("%d ", root->data);
   }
}

int main() {
   int i;
   int array[7] = { 27, 14, 35, 10, 19, 31, 42 };

   for(i = 0; i < 7; i++)
      insert(array[i]);

   i = 31;
   struct node * temp = search(i);

   if(temp != NULL) {
      printf("[%d] Element found.", temp->data);
      printf("\n");
   }else {
      printf("[ x ] Element not found (%d).\n", i);
   }

   i = 15;
   temp = search(i);

   if(temp != NULL) {
      printf("[%d] Element found.", temp->data);
      printf("\n");
   }else {
      printf("[ x ] Element not found (%d).\n", i);
   }

   printf("\nPreorder traversal: ");
   pre_order_traversal(root);

   printf("\nInorder traversal: ");
   inorder_traversal(root);

   printf("\nPost order traversal: ");
   post_order_traversal(root);

   return 0;
}
```

If we compile and run the above program, it will produce the following result –

Output

```
Visiting elements: 27 35 [31] Element found.
Visiting elements: 27 14 19 [ x ] Element not found (15).

Preorder traversal: 27 14 10 19 35 31 42
Inorder traversal: 10 14 19 27 31 35 42
Post order traversal: 10 19 14 31 42 35 27
```

6. Explain the Binary Search Tree operations with algorithms?

(A) Binary Search Tree:

In Binary search Tree (BST), the worst case timing complexity of all the three operations is O(log n), where 'n' is no. of nodes present in the tree.

* There are various operations performed on binary search tree.
* The three basic operations are

1. Search
2. Insertion
3. Deletion

operation	worst case time complexity
Searching of an element	O(n)
Insertion of an element	O(1) if we can insert at any location. O(n) if we need to insert in the empty slot of binary tree.
Deletion of an element	O(n)

Scanned with CamScanner

1. Searching : Searching is used to determine whether the given element is found or not.

→ If the element is found then it returns the pointer where the element is found. If the element is not found then points a message as "the element is not found".

→ The searching process can be start with the root node.

→ The searching process first checks, if the binary search tree is empty or not. If it is empty then the value we are searching is not present in the tree.

→ However if there are nodes in the tree then the search function checks if the key value of the current node is equal to our searching node.

→ If not, then if the key is greater than root's key, we recur for right subtree of root node, otherwise we recur for the left subtree.

Algorithm:

step-1: Compare the element with the root of the tree.

step-2: If it is matched then print "element is found".

step-3: otherwise check if element is less than the root, if so then move to the left sub-tree.

step-4: If not, then move to the right sub-tree.

step-5: Repeat this procedure recursively until match found.

step-6: If element is not found then return NULL.

(or) search (ROOT, ELEMENT)

- Step 1: If ROOT→data = Element or ROOT = NULL

  Return ROOT

  Else

Scanned with CamScanner

If ROOT < ROOT→data

Return search (ROOT→LEFT, ELEMENT)

else

Return search (ROOT→ RIGHT, ELEMENT)

[END of IF]

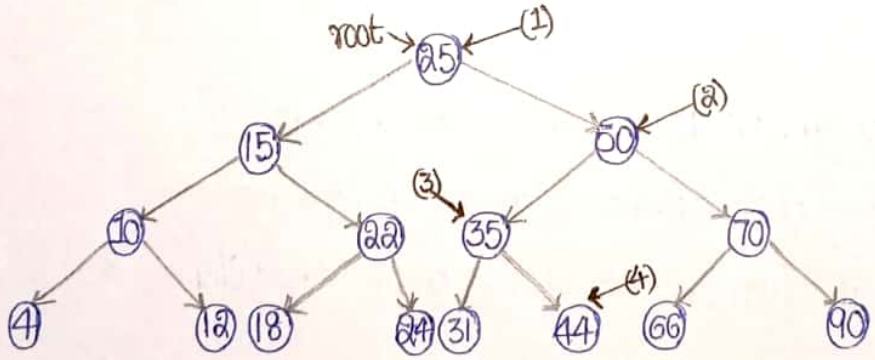
[END of IF]

- step-2: END

Example: search for 45 in the tree

(key fields are show in node rather than in seperate 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 Binary Search Tree.(BST).



2. Insertion: The insertion operation is used to insert a new node into an existing binary search tree.

* If our inserted element is found in the given binary search tree then insertion is not possible.

* If our inserted element is not found then the new node is inserted at the corresponding proper position.

Scanned with CamScanner

* 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.

Algorithm:

step-1: create a new node temp with given value element and set its left and right field to NULL.

step-2: If root is equal to NULL, set temp as root.

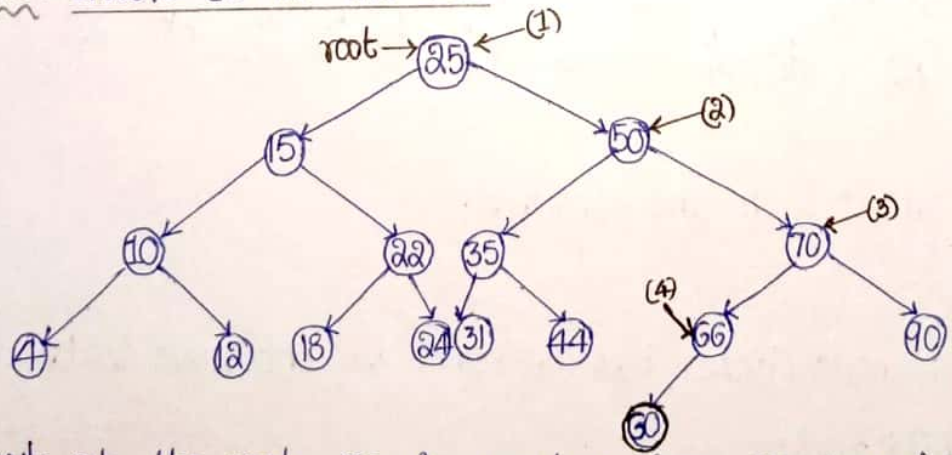
step-3: If root is not equal to empty, check if the element is smaller (or) larger than the root node.

step-4: If element is smaller than (or) equal to the node, then move to its left child. If element is larger than the node, then move to its right child.

step-5: Repeat the step-3 and step-4 until we reach a leaf node.

step-6: After reaching a leaf node, then insert the newNode as left child if newNode is smaller (or) equal to that leaf else insert it as right child.

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.

Scanned with CamScanner

3. Deletion: To delete a node from the binary search tree will take up three cases. Each of these cases to be handled with its own way.

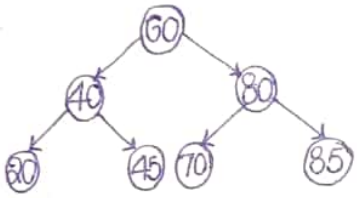
1) The node to be deleted is a leaf
2) The node to be deleted has one child either left child (or) right child.
3) The node to be deleted has both left child and right child.

case-1: The node to be deleted is a leaf:

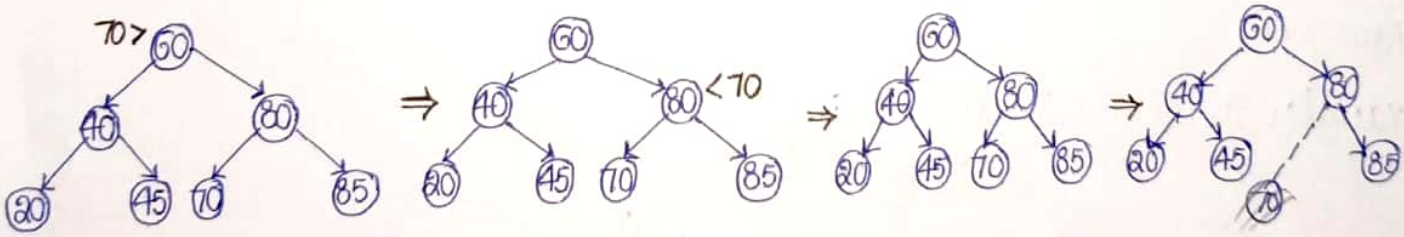
Algorithm:

Step-1: simply deallocate the memory allocated to the node.

Example: consider the binary search tree:



If we have to delete a node as 70, we can simply remove the node without any changes of tree structure. This is the simplest case in deletion.



case-2: Deleting a node with one children:

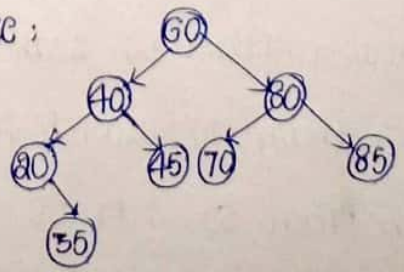
Algorithm:

Step-1: copy the child node (either left (or) right) to the node to be deleted.

Step-2: Delete the child node.

Example: consider the binary search tree:

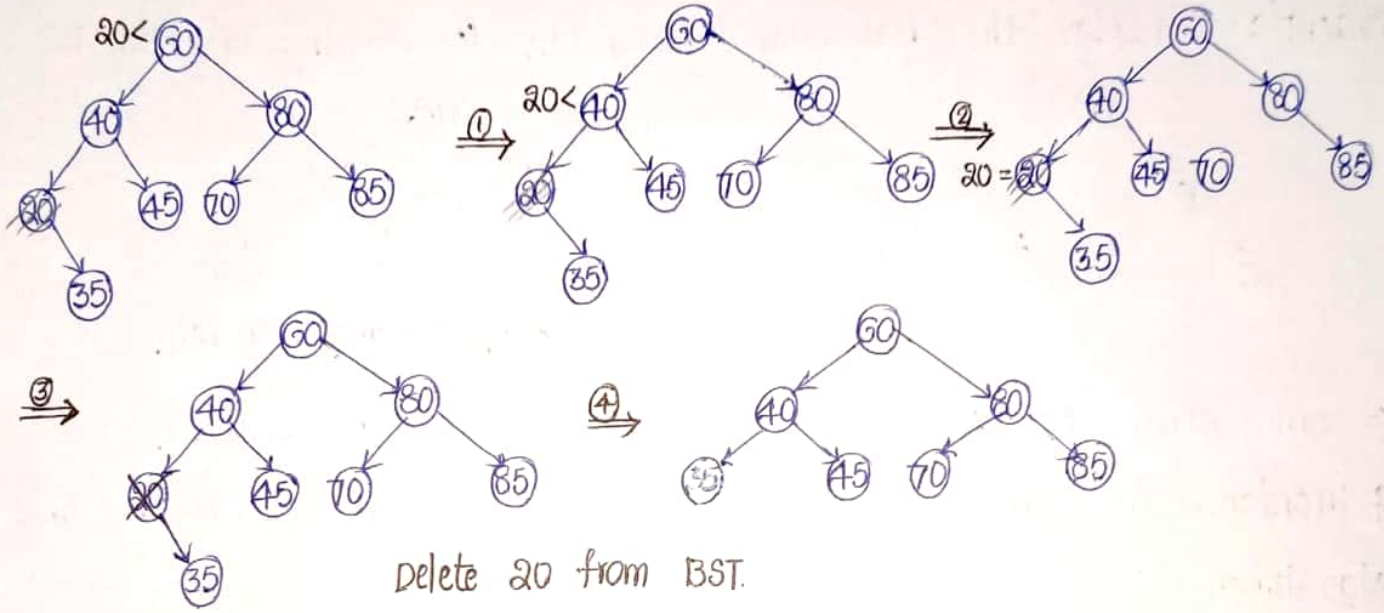
Now we have delete the node 20.



Scanned with CamScanner

→ To handle this case, the nodes of deleted one i.e., deleted node is replaced with its children. If the node was left child of its parent, the left child becomes as a parent node. similarly if the node was right child of its parent, the right child becomes a parent node.

Now, for deletion of 20 is carried out by the following steps:



Delete 20 from BST.

case-3: Deleting a node with both left child and right child:

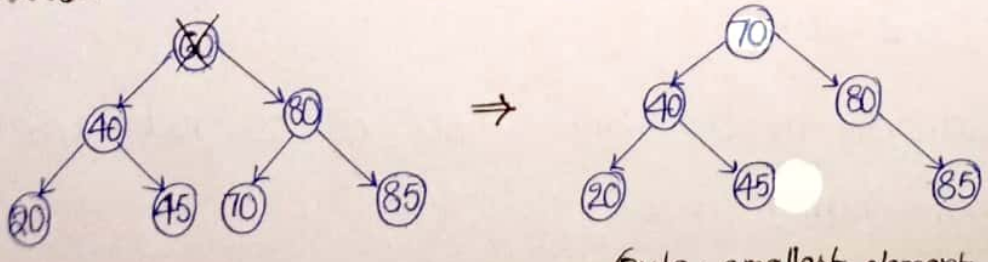
Algorithm-①:

step-1: Find the minimum value node in the right sub tree say, temp. (It is the left most node in the right subtree of the node to be deleted).

step-2: copy the data of temp in the node to be deleted.

step-3: Delete the temp node from the right subtree of the node to be deleted.

Example: consider the following binary tree for deleting 60 from BST.



(Rule: smallest element in RST)

Scanned with CamScanner

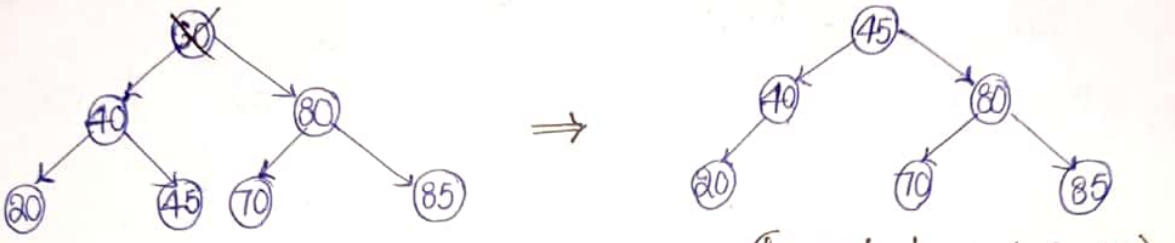
Algorithm-②:

step-1: Find the maximum value node in the left subtree say, temp.

step-2: copy the data of temp in the node to be deleted.

step-3: Delete the temp node from the left subtree of the node to be deleted.

Example: consider the following binary tree for deleting 60 from BST:



(largest element in LST).

* some other operations are inorder(), preorder(), postorder().

4. inorder(): The function performs as in-order traversal on the tree.

Algorithm:

step-1: Traverse the elements in the left subtree of the root node i.e, call inorderInBST (root → left).

step-2: visit the root node i.e, print the data field of the root.

step-3: Traverse the elements in the right subtree of the root node i.e, call inorderInBST (root → right).

5. preorder(): The function performs a preorder traversal on the tree.

Algorithm: step-1: visit the root node.

Step-2: Traverse the elements in the left subtree of the root node i.e, call preorderInBST (root → left)

step-3: Traverse the elements in the right subtree of the root node ie, call preorderInBST (root → right)

Scanned with CamScanner

6. postorder() : The function performs a postorder traversal on the tree.

Algorithm:

Step-1: Traverse the elements in the left subtree of the root node i.e., call postorderInBST (root → left)

Step-2: Traverse the elements in the right subtree of the root node i.e., call postorderInBST (root → right)

Step-3: Visit the root node.

Scanned with CamScanner

5. Discuss about Balanced Binary Tree. Give suitable example.

(A) Balanced Binary Tree:

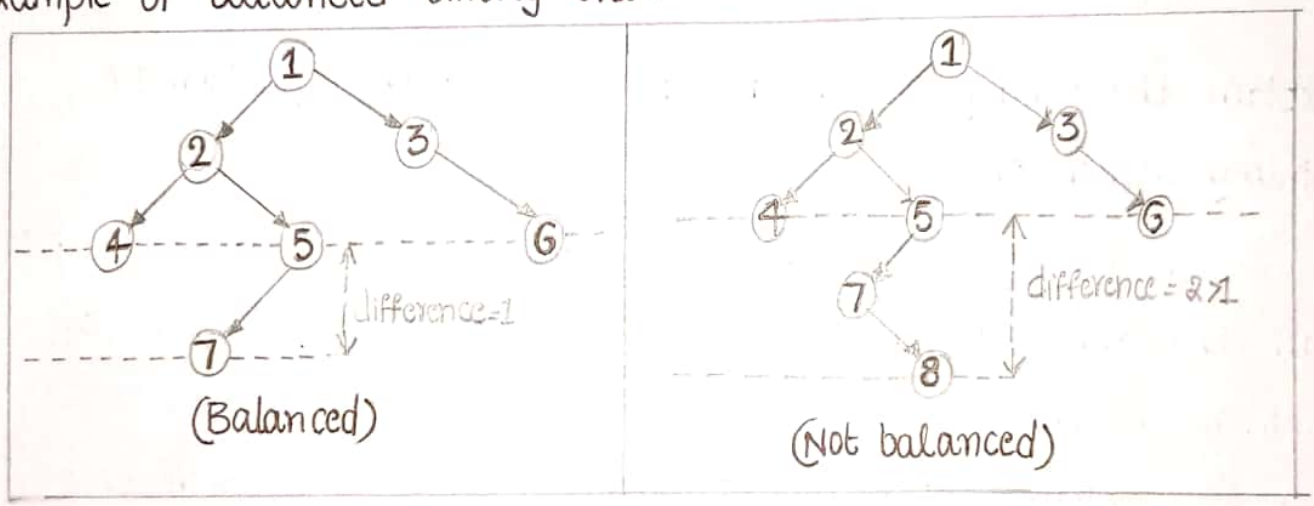
A balanced binary tree is a binary tree in which the left and right subtrees of every node differ in height by no more than 1.

* one may also consider binary trees where no leaf is much farther away from the root than any other leaf.

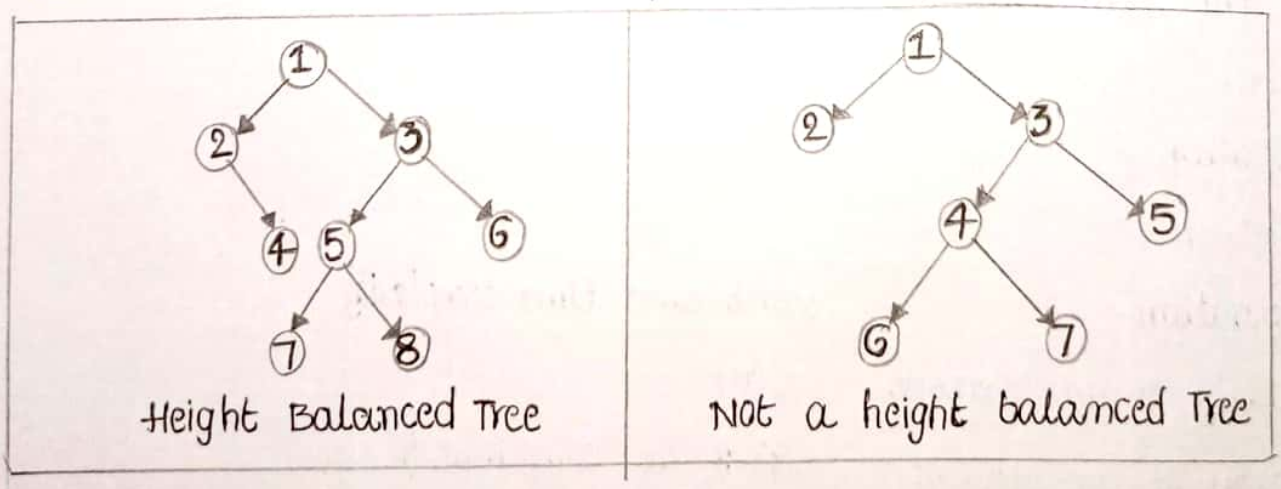
* A binary tree is balanced if for any two leaves the difference of the depth is at most 1.

Example of balanced binary tree:

*



* Let us consider another example for balanced binary tree:

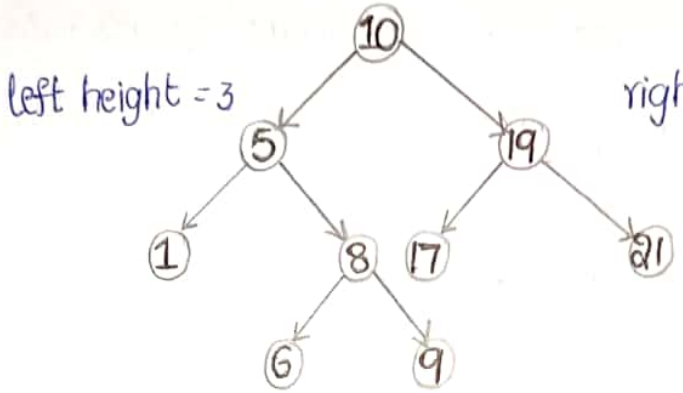


* Balanced binary Tree is also be called as self balancing binary search tree or height balanced binary search tree.

Scanned with CamScanner

Mechanism:

At node (5), again height difference is 1, so we will go down left and right subtrees and check if they are balanced.



∴ so it's height difference is "1". so it is balanced binary tree.

Scanned with CamScanner

3. What is the threaded Binary Tree? Explain types of the threaded Binary tree.

(A) Threaded Binary Tree:

→ A threaded binary tree is a binary tree but with a difference in storing the NULL pointers.

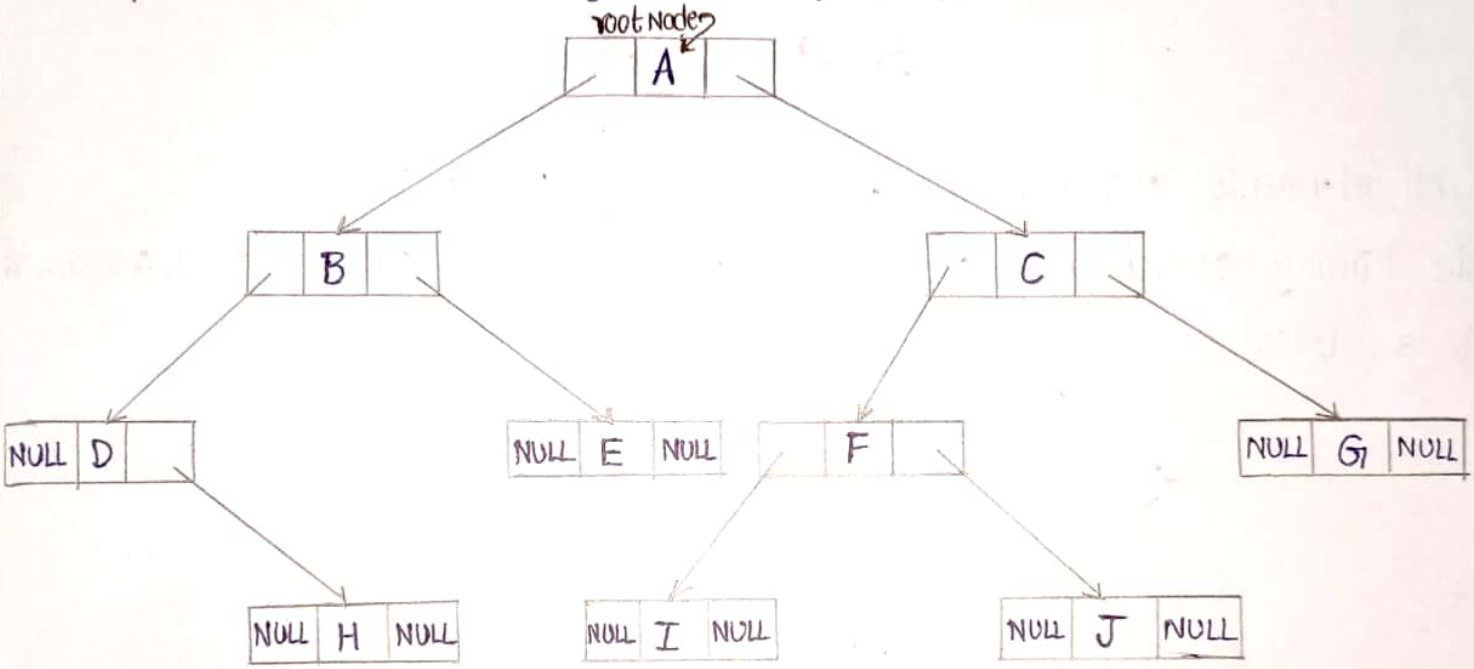
→ A binary tree is represented using array representation or linked list representation. When a binary tree is represented using linked list representation, if any node is not having a child we use NULL

Scanned with CamScanner

pointer in that position. In any binary tree linked list representation, there are more number of NULL pointer than actual pointers.

→ Generally, in any binary tree linked list representation, if there are 2N number of reference fields, then N+1 number of reference fields are filled with NULL (N+1 are NULL out of 2N). This NULL pointer does not play, any role except indicating there is no link (no child).

Example: consider the binary tree as follows:



→ Threaded Binary Tree is also a binary tree in which all left child pointers that are NULL (in linked list representation) points to its in-order predecessor, and all right child pointers that are NULL (in linked list representation) points to its in-order successor.

* we have the pointers reference the next node in an inorder traversal; called threads.
* The binary tree containing the threads are called threaded binary tree.
* In the linked list representation of threaded binary tree will be denoted using dotted lines.

Scanned with CamScanner

Need of Threaded Binary tree:

* Binary trees have a lot of wasted space: the leaf node each have 2 null pointers. we can use these pointers to help us in inorder traversals.
* Threaded binary tree makes the tree traversal faster since we do not need stack or recursion for traversal.

Types of Threaded Binary Tree:

1. Single threaded or one way (left or right) Threaded binary tree.
2. Double threaded or Multi way Threaded Binary tree.

1. Single Threaded Binary Tree: It is a one-way Threaded binary tree. They are again two types:

(i) Left way Threaded Binary Tree.

(ii) Right way Threaded Binary Tree.

i.e., Each node is threaded towards either the in-order predecessor or successor (left or right) means all right null pointers will point to inorder successor OR all left null pointers will point to inorder predecessor.

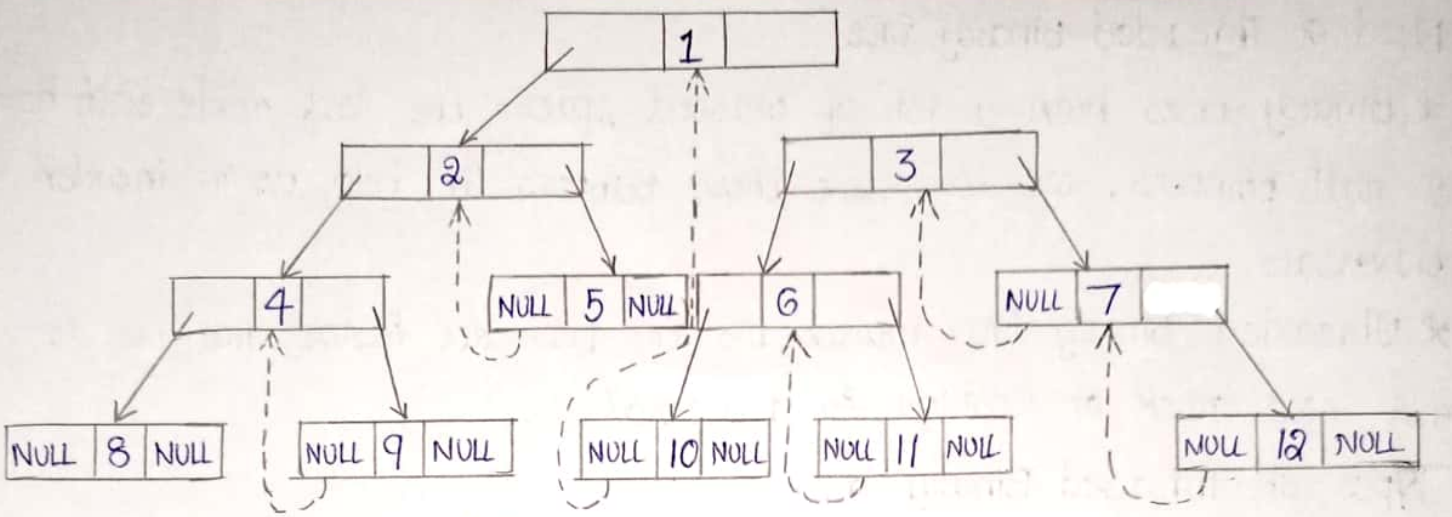
(i) Left way Threaded Binary Tree:

The tread will appear in the left field. The left field null pointer is made to point with its inorder predecessor, which is called left way threaded binary Tree.

inorder: 8-4-9-2-5-1-10-6-11-3-7-12.

Now, the left way threaded binary tree is as follows:

Scanned with CamScanner



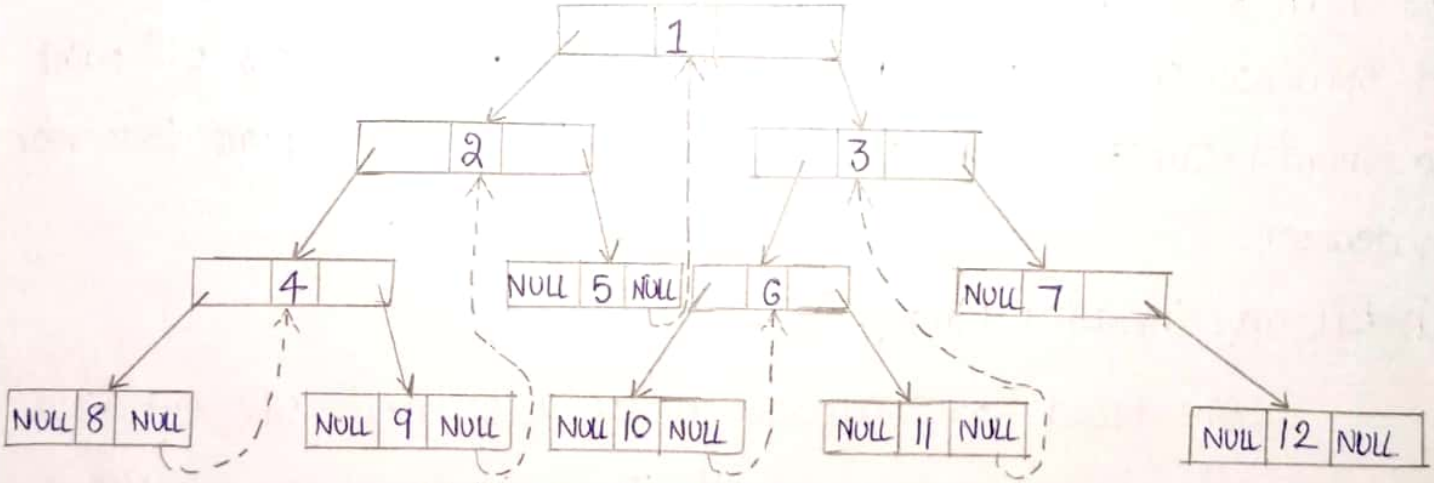
Left Threaded binary Tree (in link representation)

(ii) Right way Threaded binary tree:

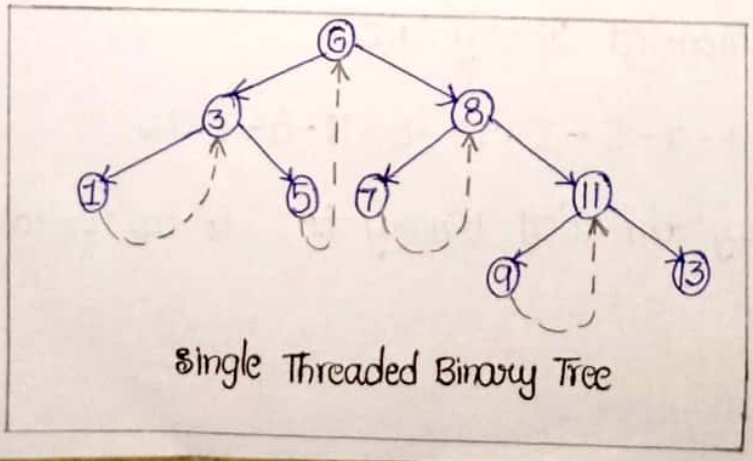
The Thread will appear in the right field. The right field null pointer is made to point with its inorder successor, which is called right way threaded binary tree.

inorder : 8-4-9-2-5-1-10-6-11-3-7-12.

Now, the right way threaded binary tree is as follows:



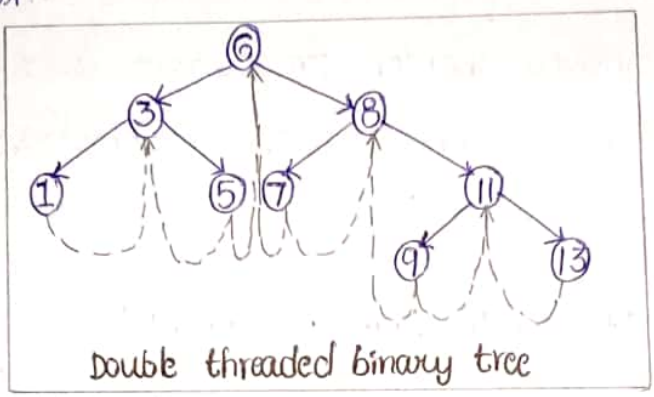
Right way Threaded Binary Tree (in link representation)



Single Threaded Binary Tree

Scanned with CamScanner

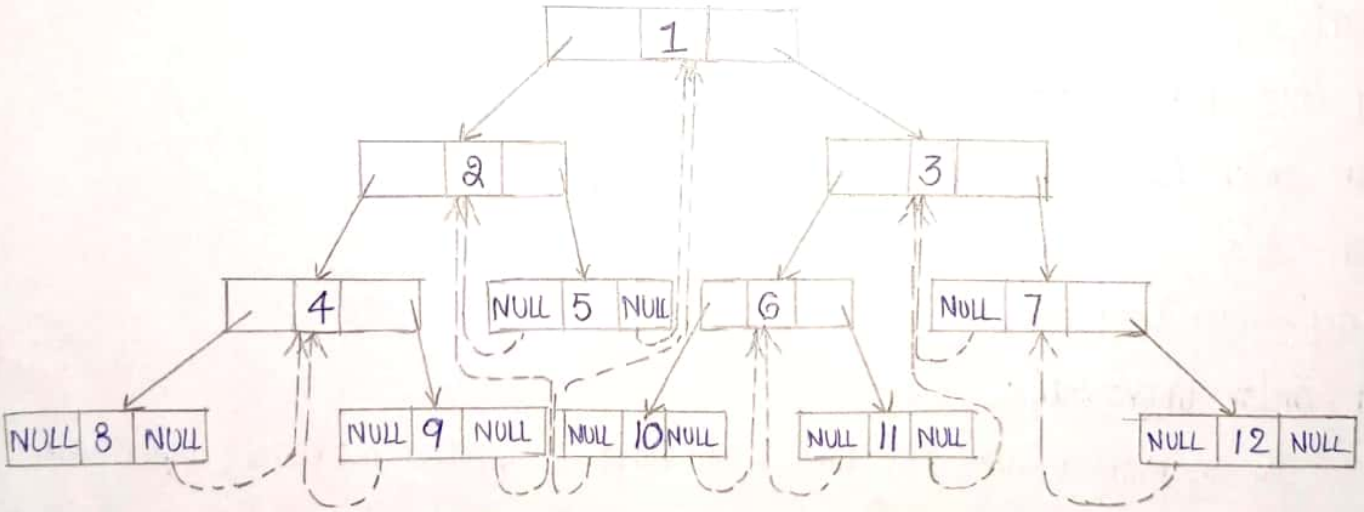
2. Double Threaded Binary Tree: It is a multi way Threaded Binary tree i.e., Each node is threaded towards both the inorder predecessor and successor (left and right) means all right null pointers will point to the inorder successor AND all left null pointers will point to the inorder predecessor.



Double threaded binary tree

inorder: 8-4-9-2-5-1-10-6-11-3-7-12.

Now, the double threaded binary tree is as follows:



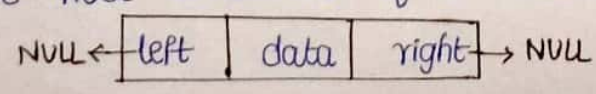
Double threaded binary tree.

Operations of a Single Threaded binary Tree:

The two primary operations in single Threaded binary Tree are:

1) Insert node into tree
2) print or Traverse the tree.

* NOTE * The node structure of a Threaded Binary Tree (Linked list)



Scanned with CamScanner

Advantages of Threaded Binary tree:

1. By doing threading we neglect the recursive method of traversing a tree, which makes use of stack and consumes more and time.
2. The node can keep record of its root.
3. It enables linear traversal of elements in the tree.
4. The node contains pointers inorder predecessor and successor. The threaded tree enables forward and backward traversal as given by its inorder passion.

Scanned with CamScanner