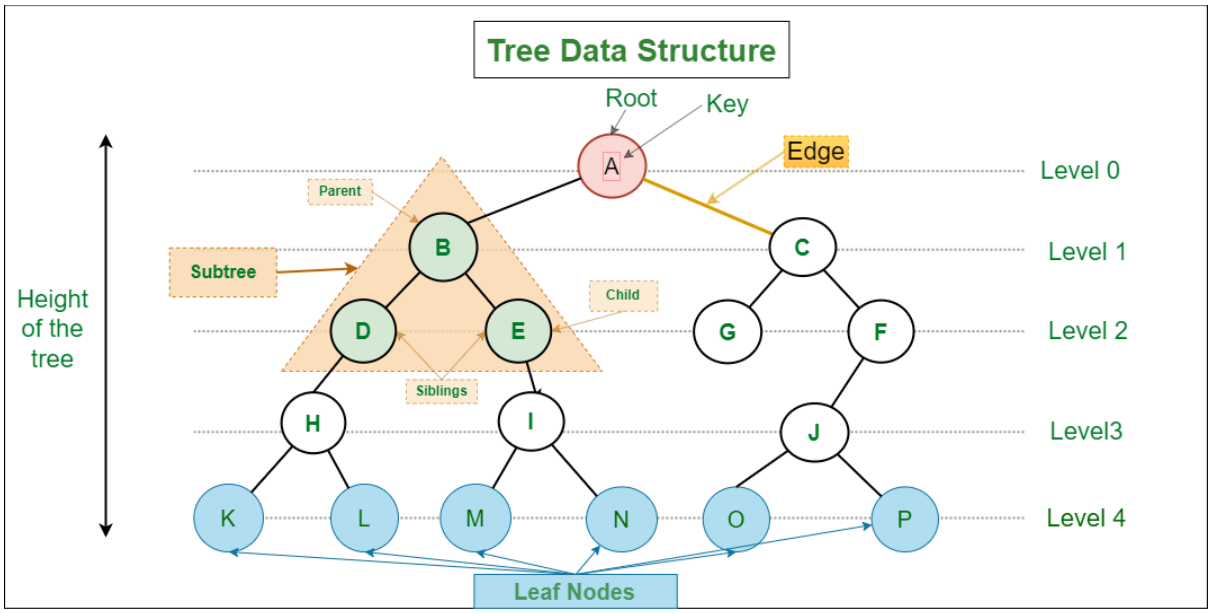
DEPARTMENT OF
BASIC SCIENCES & HUMANITIES
VIGNAN'S
INSTITUTE OF INFORMATION TECHNOLOGY
(AUTONOMOUS)
Beside VSEZ, Duvvada, Vadlapudi Post, Gajuwaka, Visakhapatnam - 530 049.
0891 - 27 55 222 / 333
0891 - 27 52 333
www.vignaniit.edu.in

UNIT-4: TREES

*A **tree data structure** is a hierarchical structure that is used to represent and organize data in a way that is easy to navigate and search.*

It is a collection of nodes that are connected by edges and has a hierarchical relationship between the nodes.



Basic Terminology

Forest: A collection of disjoint trees.

Root node: The root node A is the topmost node in the tree. If A=NULL, then it means the tree is empty.

Parent Node: The node having further sub-branches is called Parent Node. C is a parent node of G and F.

Child Node: Any sub node of a given node is called a child node. G and F are child nodes of C.

Sub trees: If the root node A is NOT NULL, then the trees T_1 and T_2 are called the sub-trees of A.

Leaf node: A node that has no children is called a leaf node or the terminal node.

Path: A Sequence of consecutive edges is called a path.

Ancestor Node: An ancestor of a node is any predecessor node on the path from root to that node.

Descendant Node: A descendant node is any successor node on any path from the node to a leaf node. A leaf node does not have any descendants.

Level number: Every node in the tree is assigned a level number in such a way that the root node is at level 0, children of the root node are at level number 1. Thus, every node is at one level higher than its parent. So all child nodes have a level number given by parent's level number +1.

Degree: Degree of a node is equal to the number of children that a node has. The degree of a leaf node is zero.

In-Degree: In-degree of a node is the number of edges arriving at that node.

Out-Degree: Out-degree of a node is the number of edges leaving that node.

Degree of tree: The maximum degree of a tree is node having maximum degree is called degree of tree.

Height of the Tree: In a tree data structure, the number of edges from the leaf node to the particular node in the longest path is known as the height of that node. In the tree, the height of the root node is called "Height of Tree".

Basic Operation of Tree Data Structure:

- **Create** – create a tree in the data structure.
- **Insert** − Inserts data in a tree.
- **Search** − Searches specific data in a tree to check whether it is present or not.
- **Traversal**:
    - **Preorder Traversal** – perform Traveling a tree in a pre-order manner in the data structure.
    - **In order Traversal** – perform Traveling a tree in an in-order manner.
    - **Post-order Traversal** –perform Traveling a tree in a post-order manner.

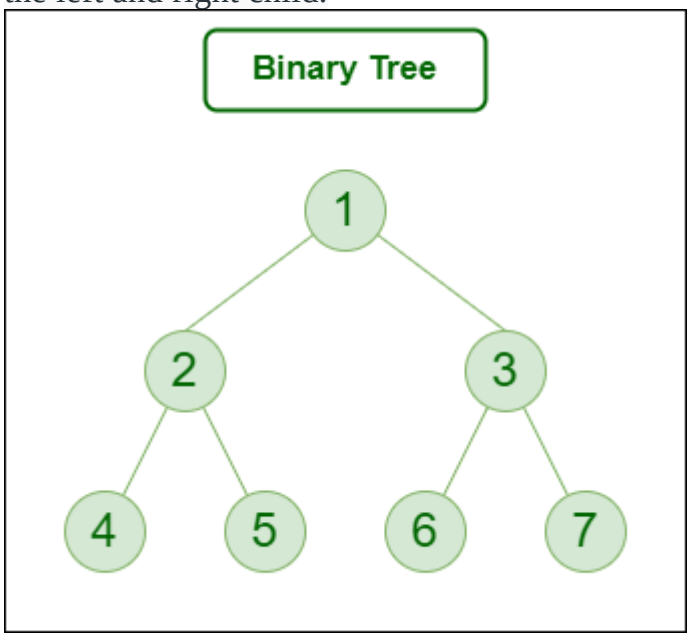
Application of Tree Data Structure:

- **File System:** This allows for efficient navigation and organization of files.
- **Data Compression**: Huffman coding is a popular technique for data compression that involves constructing a binary tree where the leaves represent characters and their frequency of occurrence. The resulting tree is used to encode the data in a way that minimizes the amount of storage required.
- **Compiler Design:** In compiler design, a syntax tree is used to represent the structure of a program.
- **Database Indexing**: B-trees and other tree structures are used in database indexing to efficiently search for and retrieve data.

Types of Trees:

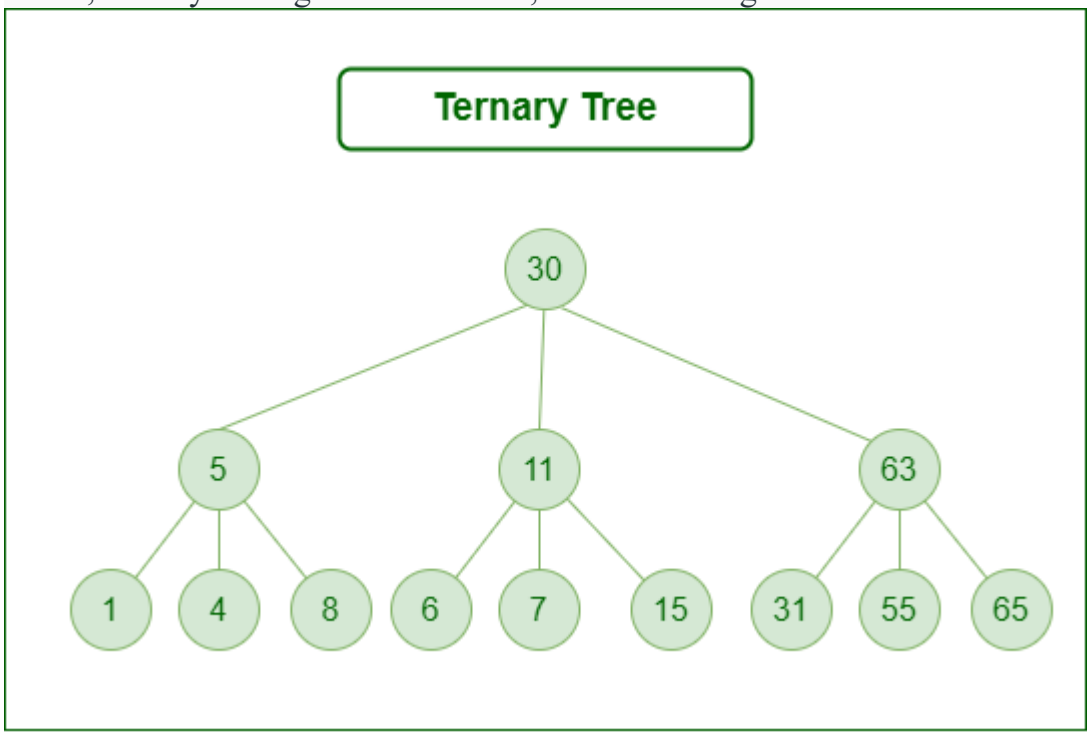
1. Binary Tree

A binary Tree is defined as a Tree data structure with at most 2 children. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.



2. Ternary Tree

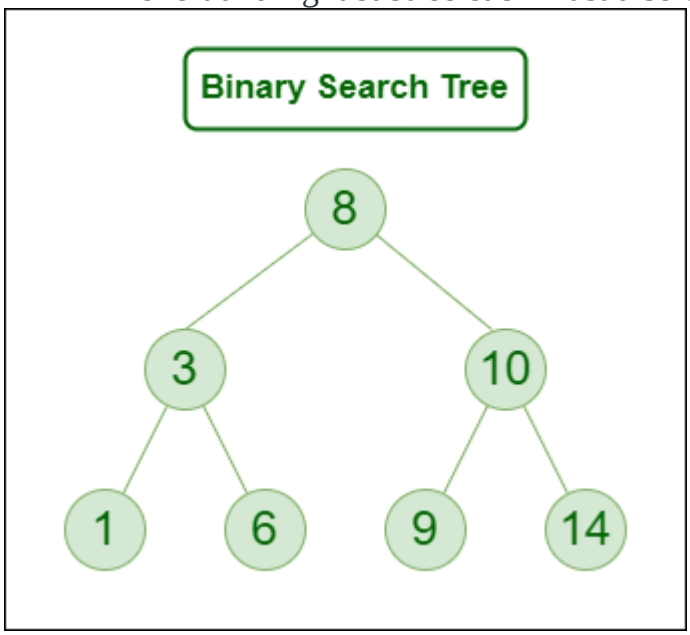
A Ternary Tree is a tree data structure in which each node has at most three child nodes, usually distinguished as “left”, “mid” and “right”.



3. Binary Search Tree

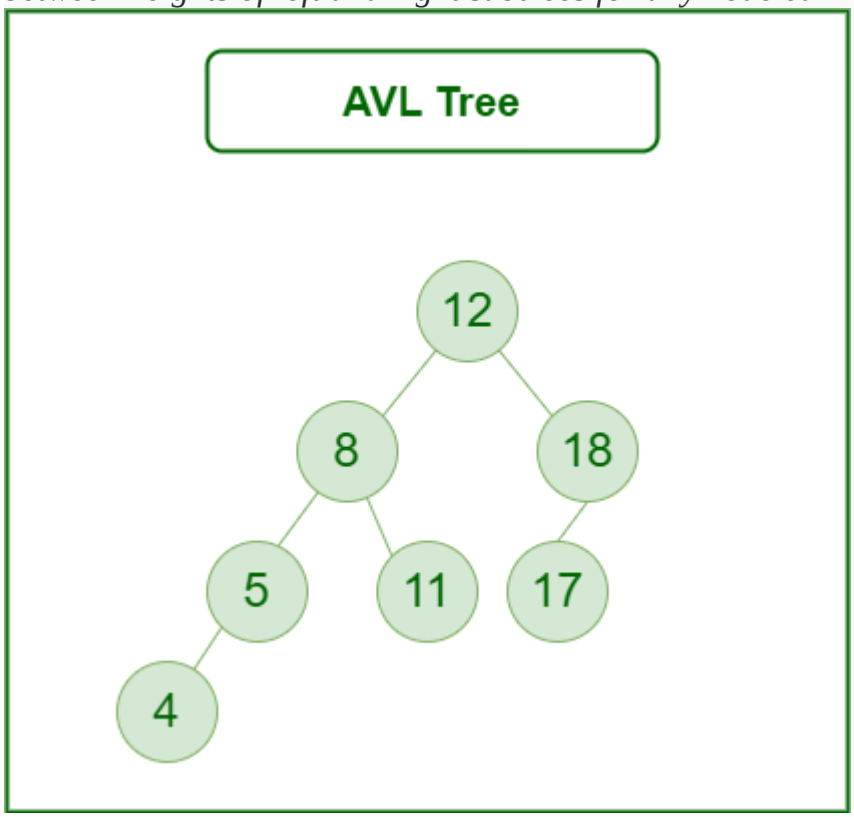
A **binary Search Tree** is a node-based binary tree data structure that has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node’s key.
- The right subtree of a node contains only nodes with keys greater than the node’s key.
- The left and right subtree each must also be a binary search tree.



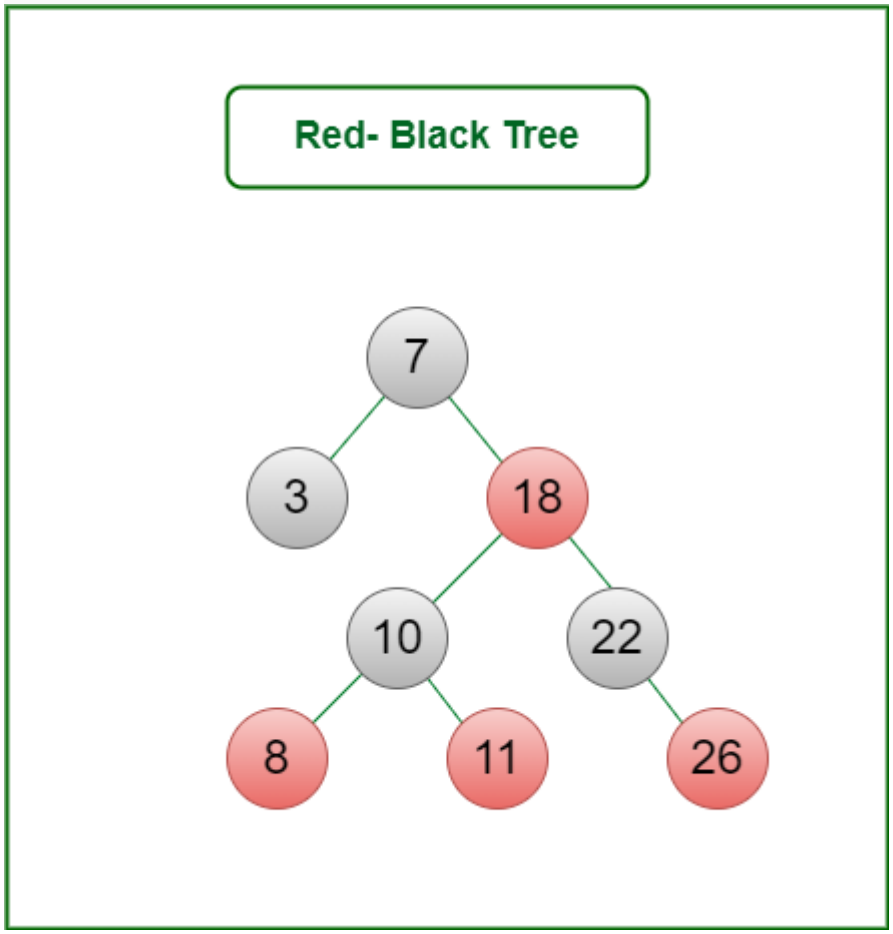
4. AVL Tree

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



5. Red-Black Tree

A red-black tree is a kind of self-balancing binary search tree where each node has an extra bit, and that bit is often interpreted as the color (red or black). These colors are used to ensure that the tree remains balanced during insertions and deletions.



6. B Tree

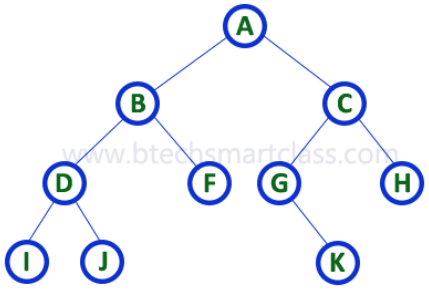
B-Tree is a self-balancing search tree. In most of the other self-balancing search trees (like AVL and Red-Black Trees), it is assumed that everything is in the main memory.

7. B+ Tree

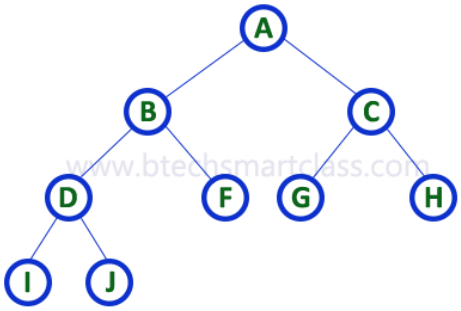
B+ tree eliminates the drawback B-tree used for indexing by storing data pointers only at the leaf nodes of the tree. Thus, the structure of leaf nodes of a B+ tree is quite different from the structure of internal nodes of the B tree.

Types of Binary Tree

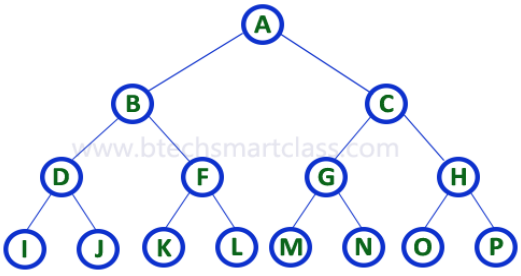
Binary tree: A tree in which every node can have a maximum of two children is called as Binary tree. i.e., it can have either 0, 1 or 2 children.



Strictly Binary tree: A binary tree in which every node has either two or zero number of children is called strictly binary tree. It is also called as *full binary tree*, *proper binary tree* or *2-binary tree*.



Complete Binary tree: 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.

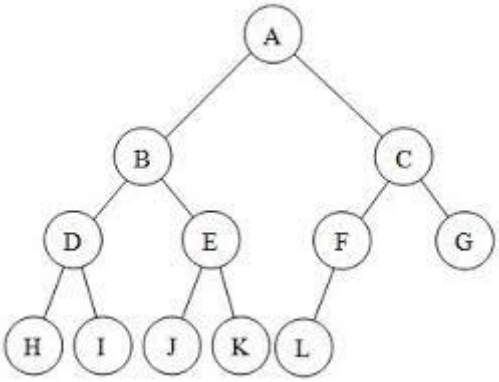


Almost Complete Binary tree:

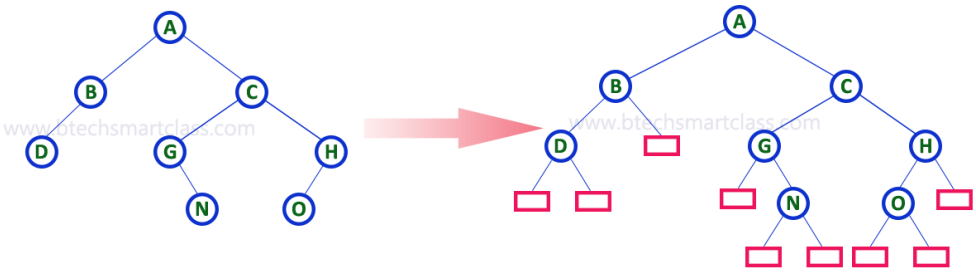
A Binary Tree of depth d is Almost Complete if:

1. The tree is Complete Binary Tree (All nodes) till level (d-1).

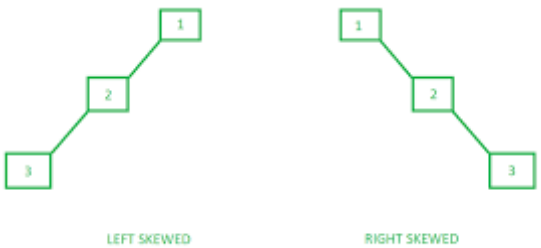
2. At level d, (i. e the last level), if a Node is present, then all the Nodes to the left of that node should also be present.



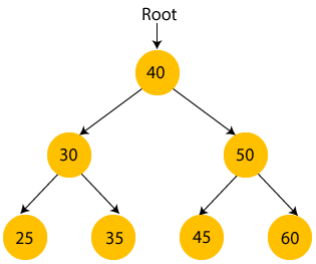
Extended Binary tree: The full binary tree obtained by adding dummy nodes to a binary tree is called extended binary tree.



Skew tree: A skew tree is defined as a binary tree in which every node except the leaf has only one child node. There are two types of skew tree, i.e. left skewed binary tree and right skewed binary tree.



Binary Search Tree: A binary search tree follows some order to arrange the elements. In a Binary search tree, the value of left node must be smaller than the parent node, and the value of right node must be greater than the parent node. This rule is applied recursively to the left and right subtrees of the root.



The Abstract Data Type of Binary Tree

structure *Binary_Tree*(abbreviated *BinTree*) is

objects: a finite set of nodes either empty or consisting of a root node, left *Binary_Tree*, and right *Binary_Tree*.

functions:
for all *bt*, *bt1*, *bt2* ∈ *BinTree*, *item* ∈ *element*

Bintree Create()	::= creates an empty binary tree
Boolean IsEmpty(*bt*)	::= if (*bt*==empty binary tree) return *TRUE* else return *FALSE*
BinTree MakeBT(*bt1*, *item*, *bt2*)	::= return a binary tree whose left subtree is *bt1*, whose right subtree is *bt2*, and whose root node contains the data *item*
Bintree Lchild(*bt*)	::=if (IsEmpty(*bt*)) return error else return the left subtree of *bt*
element Data(*bt*)	::=if(IsEmpty(*bt*)) return error else return the data in the root node of *bt*
Bintree Rchild(*bt*)	::= if (IsEmpty(*bt*)) return error else return the right subtree of *bt*

Properties of a Binary Tree: A tree is a connected acyclic graph. In many ways, a tree is the simplest non-trivial types of a graph. It has several good properties such as the fact that there exists a unique path between every two vertices. The following theorems list some simple properties of trees: Let T be a tree. Then the following properties hold true:

1. There exists a unique path between every two vertices
2. The number of vertices is one more than the number of edges in the tree.
3. A tree with two or more vertices has at least two leaves.

Property 1
Property 1 comes from the definition of a tree. As a tree is a connected graph, there exists at least one path between every two vertices. However, if there are two or more paths between a pair of vertices, there would be a circuit in the graph and so the graph cannot be a tree.
Property 2

Property 2 can be proved using mathematical induction. Let there be a tree T with the total number of edges e and the total number of vertices v.

Induction Step A tree with one vertex contains no edge, and a tree with two vertices has one edge.

Induction hypothesis Let us consider that there is an edge {a, b} in T such that the removal of the edge {a, b} divides T into two disjoint trees T1 and T2, where T1 contains the vertex a and all the vertices whose paths to a in T do not contain the edge {a, b}, and T2 contains the vertex b and all the vertices whose paths to b do not contain the edge {a, b}.

Since both T1 and T2 have utmost v-1 vertices, it follows from the hypothesis that

For T1 => e1=v1-1 and

For T2=> e2=v2-1,

Where e1 and e2 are the number of edges and v1 and v2 are the number of vertices in T1 and T2 respectively.

This e1+e2=v1+v2-2

Since e=e1+e2+1 and V=v1+v2, we have e=v-1 as shown in the following figure:

Property 3

Property 3 follows from property 2, that is, the sum of degrees of the vertices in any graph is equal to 2e, which is equal to 2v-2, in a tree. Since a tree with more than one vertex cannot have any isolated vertex, there must be at least two vertices of in-degree 1 in the tree.

Other Properties

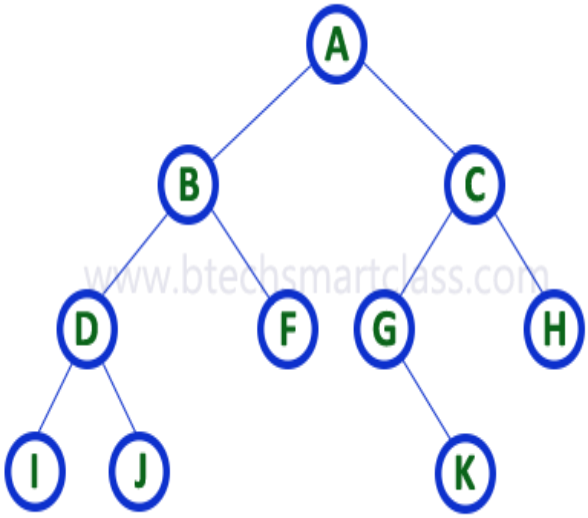
1. The maximum number of nodes of level i in a binary tree is 2^{i-1}, where $i \geq 1$.
2. The maximum number of nodes of depth d in a binary tree is 2^{d-1}, where $d \geq 1$.

Binary Tree Representations

A binary tree data structure is represented using two methods. Those methods are as follows...

1. **Array Representation**
2. **Linked List Representation**

Consider the following binary tree...



1. Array Representation of Binary Tree

In array representation of a binary tree, we use one-dimensional array (1-D Array) to represent a binary tree. Consider the above example of a binary tree and it is represented as follows...

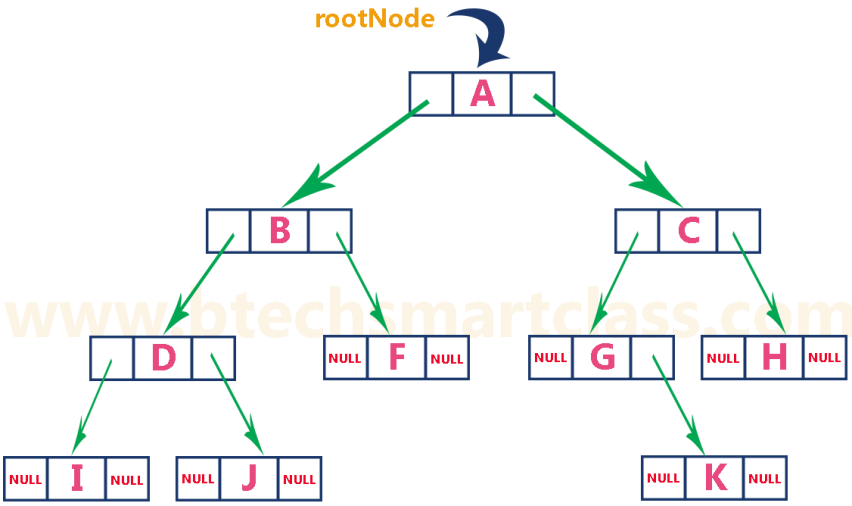
To represent a binary tree of depth **'n'** using array representation, we need one dimensional array with a maximum size of **2n + 1**.

2. Linked List Representation of Binary Tree

We use a double linked list to represent a binary tree. In a double linked list, every node consists of three fields. First field for storing left child address, second for storing actual data

and third for storing right child address.
In this linked list representation, a node has the following structure...

The above example of the binary tree represented using Linked list representation is shown as follows...



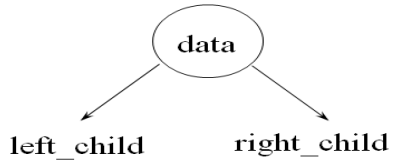
Binary Tree Representations

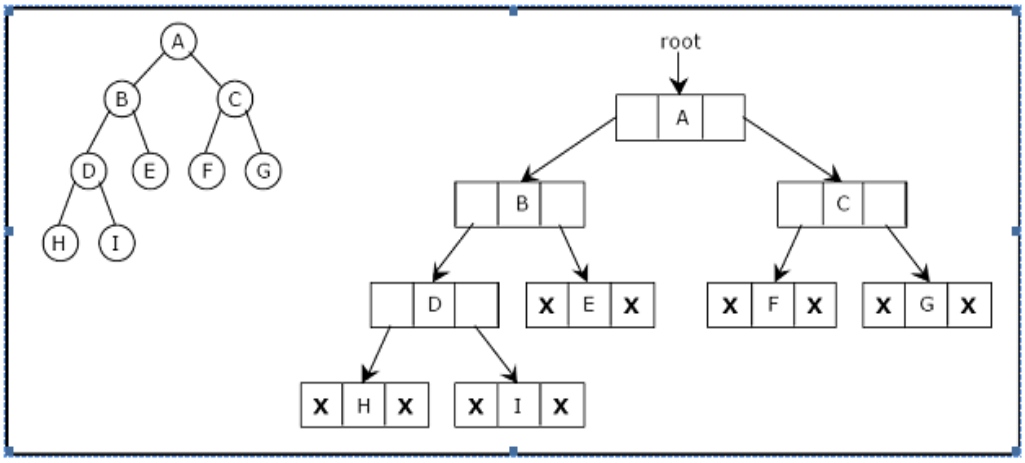
If a complete binary tree with *n* nodes (depth =log *n* + 1) is represented sequentially, then forany node with index *i*, 1<=*i*<=*n*, we have:

- *parent*(*i*) is at *i*/2 if *i*!=1. If *i*=1, *i* is at the root and has no parent.
- *left_child*(*i*) ia at 2*i* if 2*i*<=*n*. If 2*i*>n, then *i* has no left child.
- *right_child*(*i*) ia at 2*i*+1 if 2*i* +1 <=*n*. If 2*i* +1 >n, then *i* has no right child.

Linked representation:

```
typedef struct node *tree_pointer;

typedef struct node {

 int data;

 tree_pointer left_child, right_child;

};
```

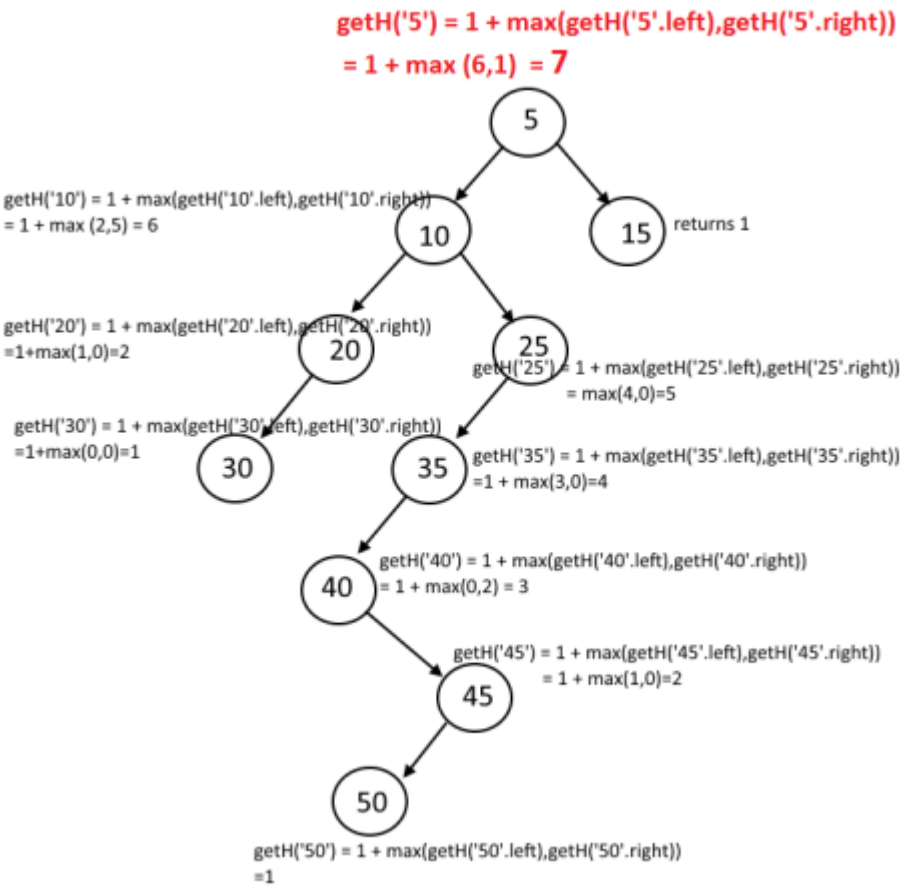




<u>Find the Maximum Depth OR Height of a Binary Tree</u>

Algorithm to find maximum depth or height of a binary tree as following:

- Get the height of left sub tree, say left Height
- Get the height of right sub tree, say right Height
- Take the Max(left Height, right Height) and add 1 for the root and return
- Call recursively.



Traversing a Binary Tree

Traversing a binary tree is the process of visiting each node in the tree exactly once in a systematic way. Unlike linear data structures in which the elements are traversed sequentially, tree is a non linear data structure in which the elements can be traversed in many different ways. There are different algorithms for tree traversals. These algorithms differ in the order in which the nodes are visited. The following are 3 different traversal techniques:

1. Pre-order traversal
2. In-order traversal
3. Post-order traversal

Pre-order Traversal: To traverse the non-empty binary tree in pre-order, the following operations are performed recursively at each node. The algorithm works by:

1. Visiting the root node R
2. Traversing the left sub-tree of R in preorder
3. Traversing the right sub-tree of R in preorder

Algorithm for pre-order traversal as following:
Step 1: repeat steps 2 and 4 while Tree != NULL
Step 2: write tree->data
Step 3: preorder(tree->left)
Step4: preorder(tree->right)
Step 5: end

In-order Traversal: To traverse the non-empty binary tree in In-order, the following operations are performed recursively at each node. The algorithm works by:

1. Traversing the left sub-tree of R in inorder
2. Visiting the root node R
3. Traversing the right sub-tree of R in inorder.

Algorithm for In-order traversal as following:
Step 1: repeat steps 2 and 4 while Tree != NULL
Step 2: inorder(tree->left)
Step 3: write tree->data
Step4: inorder(tree->right)
Step 5: end

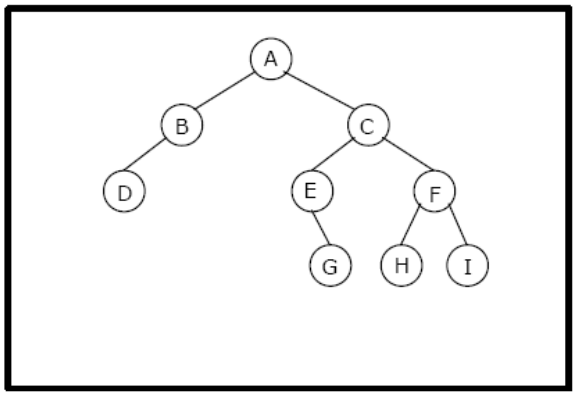
Post-order Traversal: To traverse the non-empty binary tree in post-order, the following operations are performed recursively at each node. The algorithm works by:

1. Traversing the left sub-tree of R in postorder
2. Traversing the right sub-tree of R in postorder
3. Visiting the root node R.

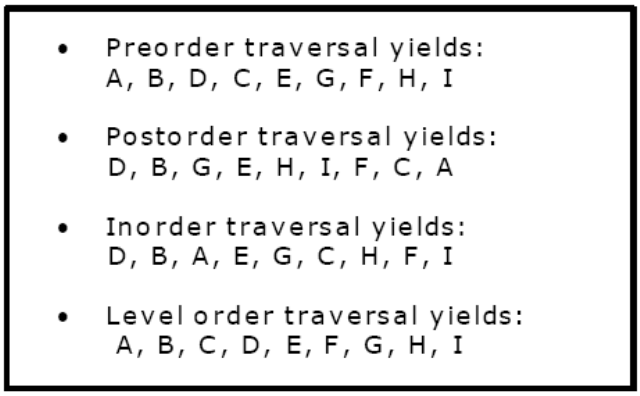
Algorithm for post-order traversal as following:
Step 1: repeat steps 2 and 4 while Tree != NULL
Step 2: postorder(tree->left)
Step 3: postorder(tree->right)
Step 4: write tree->data
Step 5: end

Example 1:

Traverse the following binary tree in pre, post, inorder and level order.



Binary Tree



Pre, Post, Inorder and level order Traversing

Creation of Binary tree from in, pre post order traversals:

Sometimes it is required to construct a binary tree if its traversals are known. From a single traversal it is not possible to construct unique binary tree. However any of the two traversals are given then the corresponding tree can be drawn uniquely:

- Inorder and preorder
- Inorder and postorder
- Inorder and level order

The basic principle for formulation is as follows:

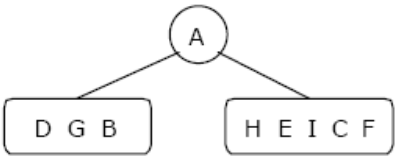
If the preorder traversal is given, then the first node is the root node. If the postorder traversal is given then the last node is the root node. Once the root node is identified, all the nodes in the left sub-trees and right sub-trees of the root node can be identified using inorder. Same technique can be applied repeatedly to form sub-trees.

It can be noted that, for the purpose mentioned, two traversal are essential out of which one should be inorder traversal and another preorder or postorder; alternatively, given preorder and postorder traversals, binary tree cannot be obtained uniquely.

Example 1:
Construct a binary tree from a given preorder and inorder sequence:
Preorder: A B D G C E H I F
Inorder: D G B A H E I C F
Solution: From Preorder sequence **A** B D G C E H I F, the root is: A
From Inorder sequence D G B **A** H E I C F, we get the left and right sub trees:

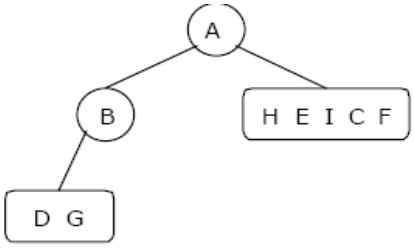
Left sub tree is: D G B
Right sub tree is: H E I C F

The Binary tree up to this point looks like:



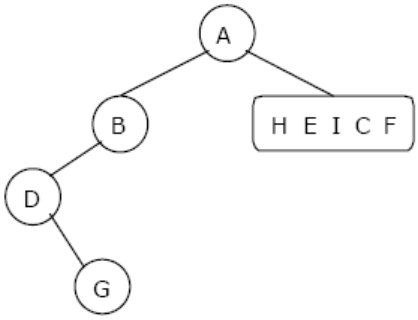
To find the root, left and right sub trees for D G B:

From the preorder sequence **B** D G, the root of tree is: B
From the inorder sequence D G **B,** we can find that D and G are to the left of B.
The Binary tree up to this point looks like:



To find the root, left and right sub trees for D G:

From the preorder sequence **D** G, the root of the tree is: D
From the inorder sequence **D** G, we can find that there is no left node to D and G is at the right of D.
The Binary tree upto this point looks like:

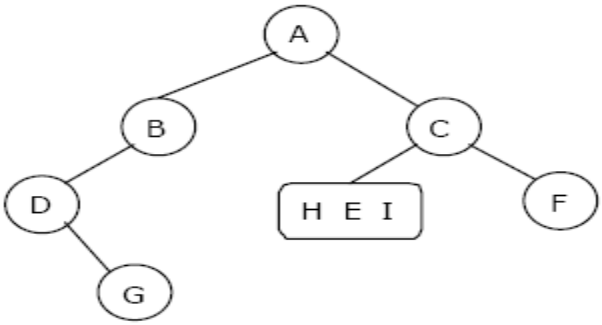


To find the root, left and right sub trees for H E I C F:

From the preorder sequence **C** E H I F, the root of the left sub tree is: C

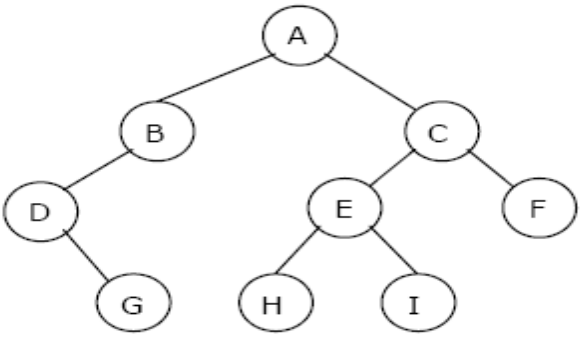
From the inorder sequence H E I **C** F, we can find that H E I are at the left of C and F is at the right of C.

The Binary tree upto this point looks like:



To find the root, left and right sub trees for H E I:

From the preorder sequence **E** H I, the root of the tree is: E
From the inorder sequence H **E** I, we can find that H is at the left of E and I is at the right of E.
The Binary tree up to this point looks like:



- ***Non Recursive Traversal Algorithms:***

Inorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex, pushing each vertex onto the stack and stop when there is no left son of vertex.
2. Pop and process the nodes on stack if zero is popped then exit. If a vertex with right son exists, then set right son of vertex as current vertex and return to step one.

Preorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path by pushing the right son of vertex onto stack, if any and process each vertex. The traversing ends after a vertex with no left child exists.
2. Pop the vertex from stack, if vertex $\neq 0$ then return to step one otherwise exit.

Postorder Traversal:

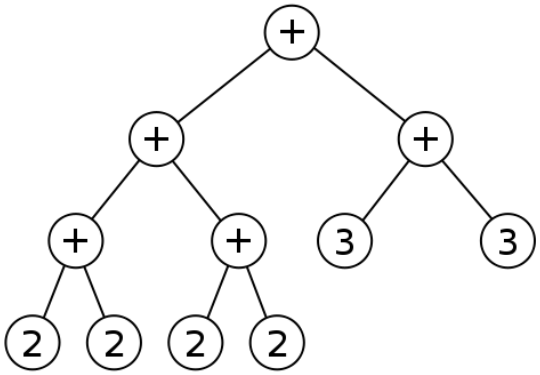
Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex. At each vertex of path push vertex on to stack and if vertex has a right son push –(right son of vertex) onto stack.
2. Pop and process the positive nodes (left nodes). If zero is popped then exit. If a negative node is popped, then ignore the sign and return to step one.

Applications of a Binary Tree:

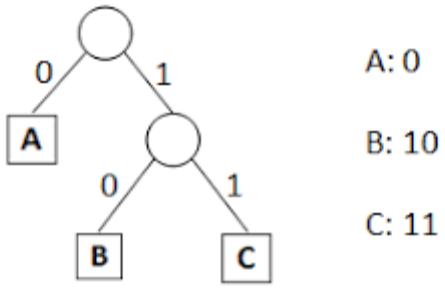
- **Expression trees** (e.g., in compiler design) for checking that the expressions are well formed and for evaluating them.

– leaf nodes are operands (e.g. constants)
– non leaf nodes are operators



- **Huffman coding trees**, for implementing data compression algorithms:

– each leaf is a symbol in a given alphabet
– code of the symbol constructed following the path from root to the leaf (left link is 0 and right link is 1)



Binary Search Tree: Definition: A binary search tree, also known as an ordinary binary search tree, is a variant of binary trees in which the nodes are arranged in an order. In a binary search tree, all the nodes in the left-subtree have a value less than that of the root node. Correspondingly, all the nodes in the right subtree have a value either equal to or greater than the root node. The same rule is applicable to every sub-tree in the tree.

To summarize, a binary search tree is a binary tree with the following properties:

- The left sub tree of a node N contains values are less than N's value
- The right sub tree of a node N contains values that are greater than N's value
- Both the left and the right binary trees also verify these properties and thus, are binary search trees.

Operations on Binary Search Trees: The following are the basic operations on binary search tree:

- Searching for a Node in a Binary Search Tree
- Inserting a New node in a Binary Search Tree
- Deleting a Node from a Binary Search Tree
    - Deleting a node that has no child
    - Deleting a node with one child
    - Deleting a node with two children

Searching for a node in Binary Search Tree:
The search function is used to find whether a given value is present in the tree or not. The searching process begins at the root node. The function first checks if the binary search tree is empty. If it is empty, then the value we are searching for is not present in the tree. So, the search algorithm terminates by displaying an appropriate message. However, if these are nodes in the tree, then the search function checks if the key value of the current node is equal to the value to be searched. If not, it checks if the value to be searched for is less than the value of the current node, in which case it should be recursively called on the left child node. In case the value is greater than the value of the current node, it should be recursively called on the right child node.

Algorithm to Search for a node in Binary Search Tree:

```
Searchelement(TREE, VAL)
Step1:  if tree->data=val or tree=NULL
            Return tree
            Else if val<tree->data
                    Return searchelement(TREE->LEFT,VAL)
```

```
        Else
                Return seachelement(TREE->RIGHT, VAL)
        End if
Step 2: Stop
```

Inserting an element into a Binary Search Tree

The insert function is used to add a new node with a given value at the correct position in the binary search tree. Adding the node at the correct position means that the new node should not violate the properties of the binary search tree. The initial code for the insert function is similar to the search function. This is because we first find the correct position where the insertion has to be done and then add the node at that position. The insertion function changes the structure of the tree. Therefore, when the insert function is called recursively, the function should return the new tree pointer.

Algorithm to insert a new node in Binary Search Tree:

```
insert (TREE, VAL)
Step1:  if  tree=NULL
        Allocate memory for TREE
        SET TREE->DATA = VAL
        SET TREE->LEFT = TREE->RIGHT=NULL
        Else if val<tree->data
                insert(TREE->LEFT, VAL)
        Else
                insert (TREE->RIGHT, VAL)
        End if
Step 2: Stop
```

Deleting an element from a Binary Search Tree

Case 1: Deleting a Node that has no children

We can simply remove the deleted node without any issue.

Case 2: Deleting a Node with one Child

To handle this case, the node's child is set as the child of the node's parent. In other words, replace the node with its child. Now, if the node is left child of its parent, the node's child becomes the left child of the node's parent. Correspondingly, if the node is the right child of its parent, the node's child becomes the right child of the node's parent.

Case 3: Deleting a Node with Two Children

To handle this case, replace the node's value with its in-order predecessor (largest value in the left subtree) or in order successor (smallest value in the right subtree). The in-order

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

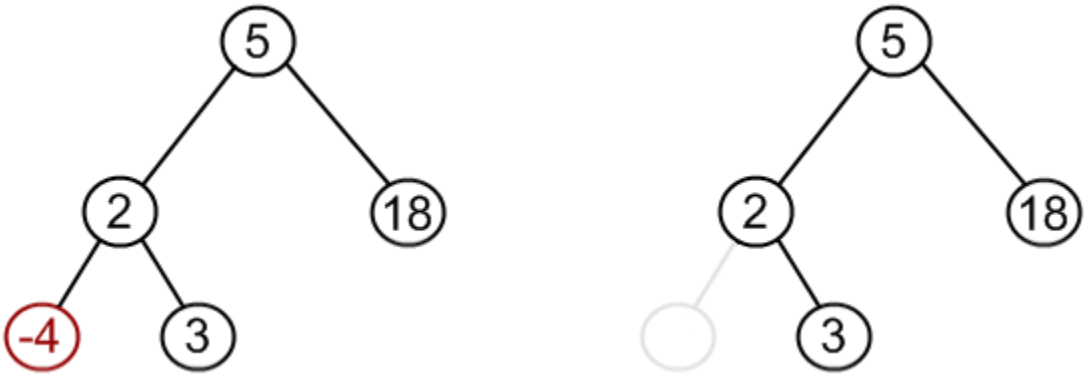
- search for a node to remove;
- if the node is found, run remove algorithm.

Remove algorithm in detail

1. Node to be removed has no children.

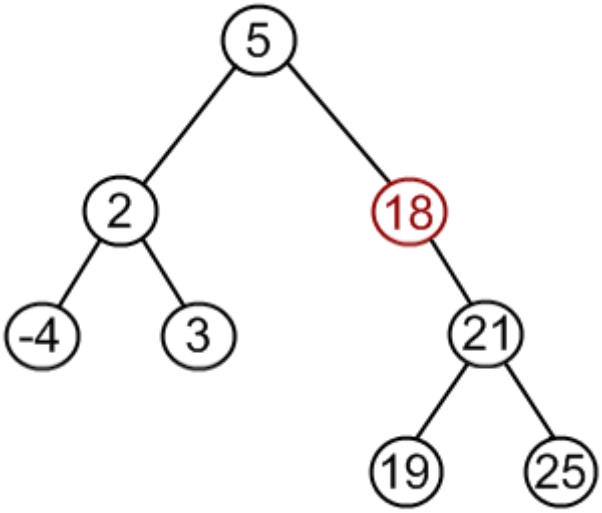
   This case is quite simple. Algorithm sets corresponding link of the parent to NULL and disposes the node.

   Example. Remove -4 from a BST.

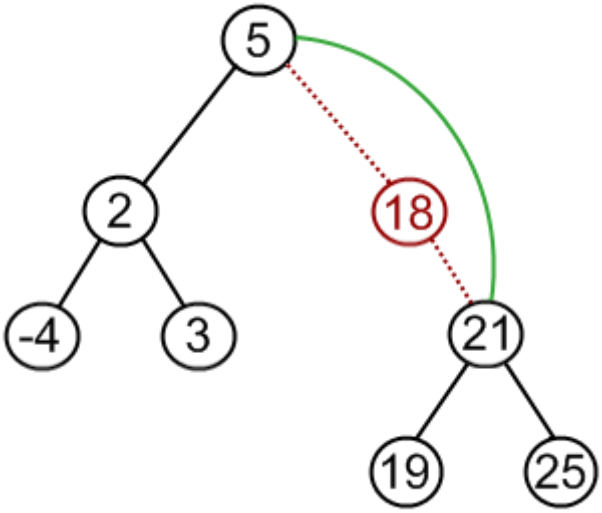


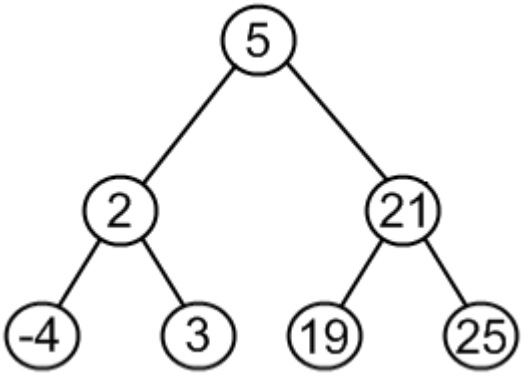
2. Node to be removed has one child.

   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.



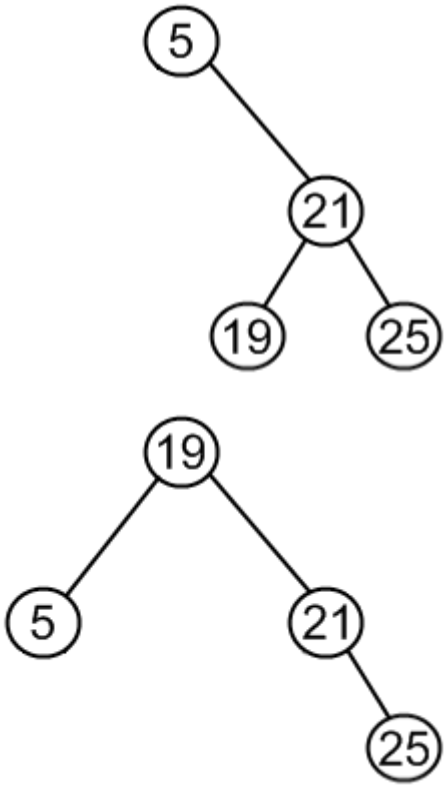
Example. Remove 18 from a BST.





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 binary-search trees. For example those BSTs:



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

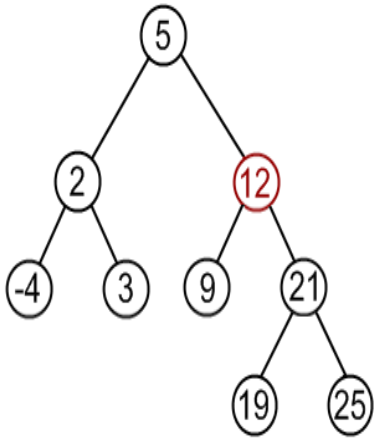
- choose minimum element from the right subtree (19 in the example);
- replace 5 by 19;
- hang 5 as a left child.

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

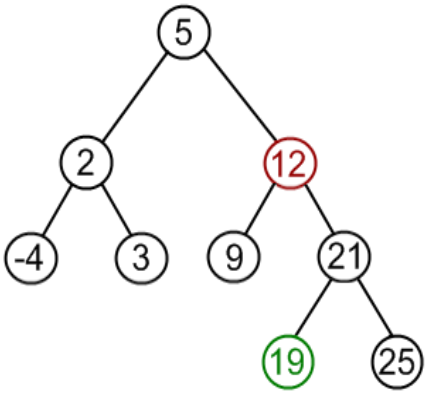
- find a minimum value in the right subtree;
- replace value of the node to be removed with found minimum. Now, right subtree contains a duplicate!
- apply remove to the right subtree to remove a duplicate.

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

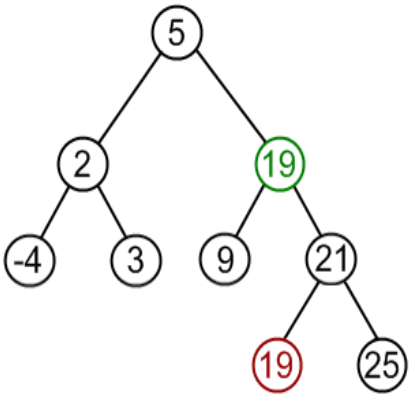
Example. Remove 12 from a BST.



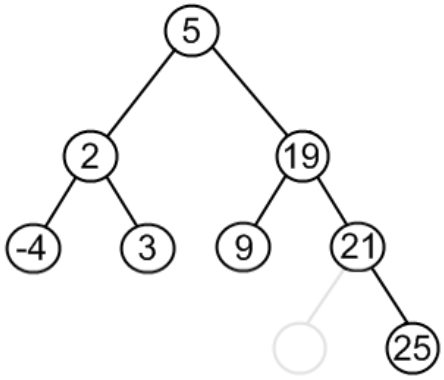
Find minimum element in the right subtree of the node to be removed. In current example it is 19.



Replace 12 with 19. Notice, that only values are replaced, not nodes. Now we have two nodes with the same value.



Remove 19 from the left subtree.


5
2
19
-4
3
9
21
25