Jadavpur University

Department of Electronics and Telecommunication Engineering,

Faculty of Engineering & Technology

DSA LAB REPORT

2nd Year First Semester 2020



Name : RAHUL SAHA

Roll: 001910701009

Group 1

**IMPLEMENTATION OF BINARY TREES AND BINARY SEARCH TREES**

Date Of Submission: 13/05/2021

**Introduction**

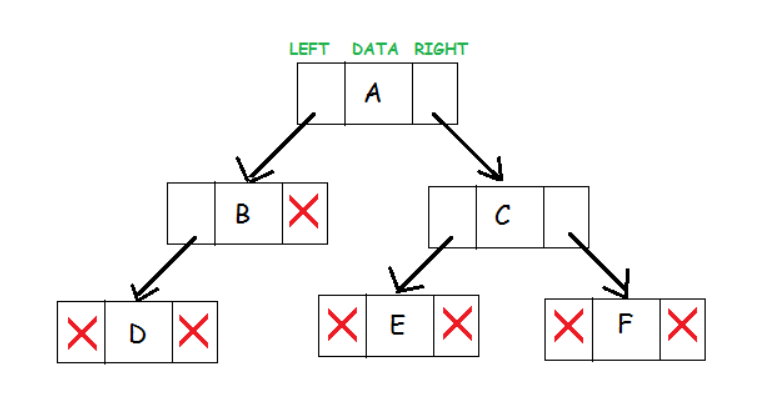
Binary Tree

A binary tree is a hierarchical data structure in which each node has at most two children generally referred as left child and right child.

Each node contains three components:

1. Pointer to left subtree
2. Pointer to right subtree
3. Data element

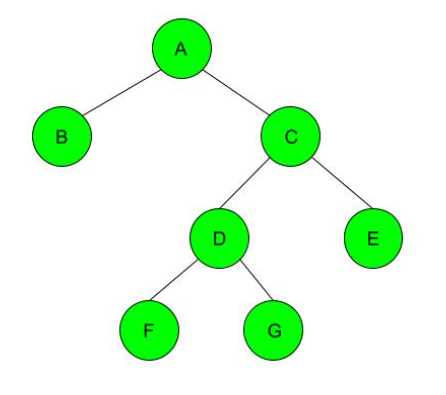
*Here is a representation of a binary tree:*



Strictly Binary Tree

If every non leaf node in a binary tree has nonempty left and right subtrees, the tree is called a strictly binary tree. A strictly binary tree with n leaves always contains 2n -1 nodes.

*Example:*



Complete Binary Tree

A complete binary tree is a binary tree in which all the levels are completely filled except possibly the lowest one, which is filled from the left.

*Example:*



Source Code:

|  |
| --- |
| **#include<stdio.h>** **#include<conio.h>** **#include<stdlib.h>** **#include<stdbool.h>**  //structure defining the tree  **typedef** **struct node {**  **int** item;  **struct node \*left, \*right;** }Tree;  Tree \*root=**NULL**;  //creates new node Tree \*create\_node(**int** num) {  Tree \*new\_node;  new\_node=(Tree \*)**malloc**(**sizeof**(Tree));  new\_node->item = num;  new\_node->left = new\_node->right = **NULL**;  **return** new\_node; };  //inserts as left child **void** insert\_left(Tree \*new\_root,**int** val) {   new\_root->left=create\_node(val); }  //inserts as right child **void** insert\_right(Tree \*new\_root,**int** val){  new\_root->right=create\_node(val); }  //search for the root node given as input from the user **bool** search\_node(Tree \*temp,**int** user\_root, **char** a, **int** num){  **if**(temp == **NULL**)  **return** 0;  **if**(temp->item == user\_root &&(temp->left==**NULL** || temp->right==**NULL**))  {  **if**(a == 'l' || a == 'L')//for left child  {  **if**(temp->left == **NULL**)  {  insert\_left(temp, num);  **return** 1;  }  **else**   **return** 0;  }  **else**   {  **if**(temp->right == **NULL**)//for right child  {  insert\_right(temp, num);  **return** 1;  }  **else**   **return** 0;  }    }  **else**  **return** search\_node(temp->left, user\_root, a, num) || search\_node(temp->right, user\_root, a, num); }  //creates the tree **void** create\_tree(){  **char** a;  **printf**("\n\nEnter the node in format specified:");  **int** num,user\_root;  **scanf**("%d %d %c",&num,&user\_root,&a);  **bool** found = 0;  **if**(a=='l'||a=='L'){  found = search\_node(root,user\_root,a,num);    }  **else** **if**(a=='r'||a=='R'){  found =search\_node(root,user\_root,a,num);  }  **if**(!found)  **printf**("The parent node doesn't exist\n"); }  //checkes if the tree is stricty binary **bool** check\_strict\_binary(Tree \*root) {  **if** (root == **NULL**)// Check if tree is empty  **return** **true**;    **if** (root->left == **NULL** && root->right == **NULL**)// Check if children are present  **return** **true**;   **if** ((root->left) && (root->right))  **return** (check\_strict\_binary(root->left) && check\_strict\_binary(root->right));   **return** **false**; }  //checkes if the tree is complete binary **bool** check\_complete\_binary(Tree \*root, **int** index, **int** numberNodes) {  **if** (root == **NULL**)// Check if the tree is complete  **return** **true**;   **if** (index >= numberNodes)  **return** **false**;   **return** (check\_complete\_binary(root->left, 2 \* index + 1, numberNodes) && check\_complete\_binary(root->right, 2 \* index + 2, numberNodes)); }  //driver code **int** main(){  system("cls");  **printf**("\t\tCheck is a Binary Tree is Scritly Binary or Complete Binary\n\n");  **int** val,num\_nodes,index=0;  **printf**("Enter the number of nodes you want to enter:");  **scanf**("%d",&num\_nodes);  **printf**("\n\nEnter the first Node:");  **scanf**("%d",&val);  root=create\_node(val);  **printf**("\n\nEnter the node data in format 'data' 'root\_node(i.e the node under which you want to attach)' 'left(l) or right chid(r)'");  **for**(**int** i=1;i<num\_nodes;i++){  create\_tree();  }  **if**(check\_strict\_binary(root)){  **printf**("\n\nThe tree is a Strictly Binary Tree");  }  **else**{  **printf**("\n\nThe tree is not a Strictly Binary Tree");  }  **if**(check\_complete\_binary(root,index,num\_nodes)){  **printf**("\n\nThe tree is a Complete Binary Tree");  }  **else**{  **printf**("\n\nThe tree is not a Complete Binary Tree\n\n");  }  **return** 0; } |

**OUTPUT CONSOLE**

As soon as the root node is entered, all the other nodes will be entered in the following format.

|  |
| --- |
| *data, parent node, left child(l)* ***or*** *right child(r)* |

1. To check if the tree is strictly binary or not

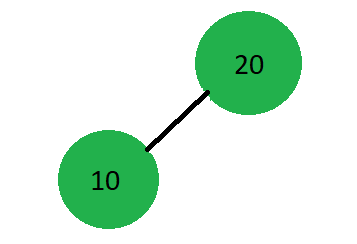


Steps of creating this binary tree

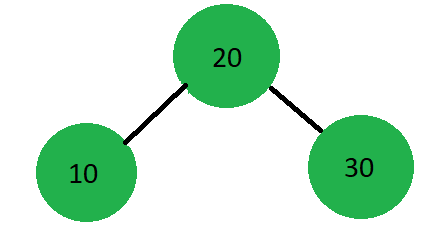
Step 1:



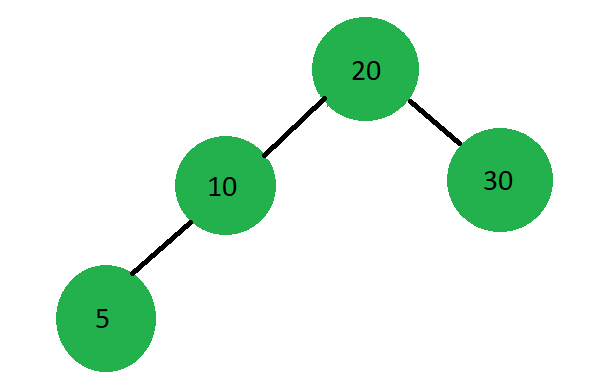
Step 2:



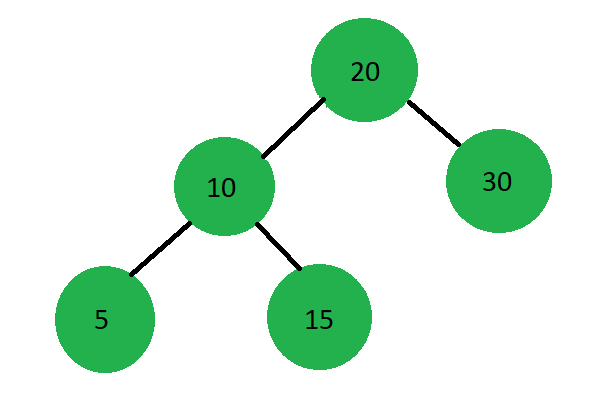
Step 3:



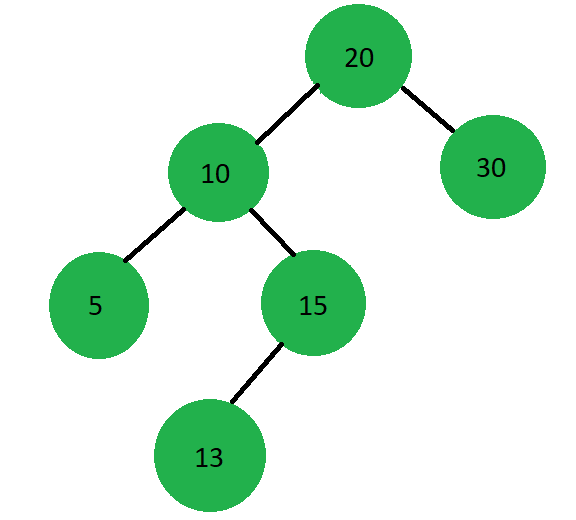
Step 4:



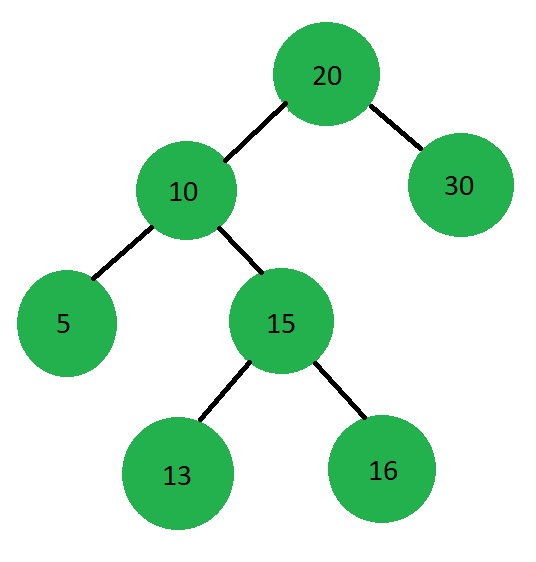
Step 5:



Step 6:



Step 7:



1. Checking if the tree is a complete binary tree

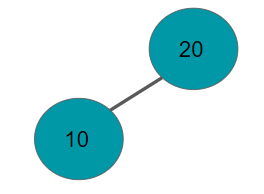


Steps of creating this binary tree

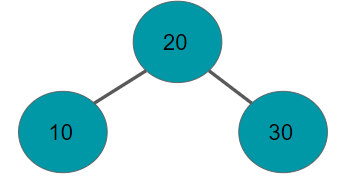
Step1:



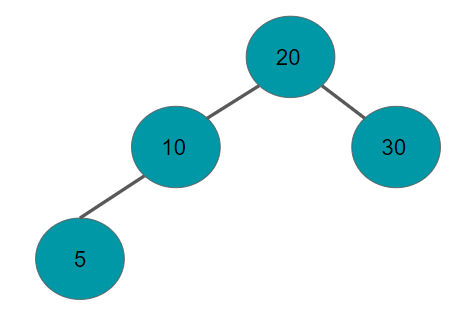
Step 2:



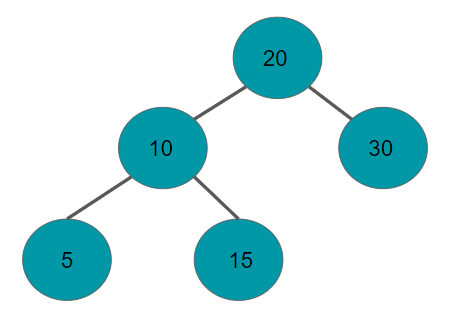
Step 3:



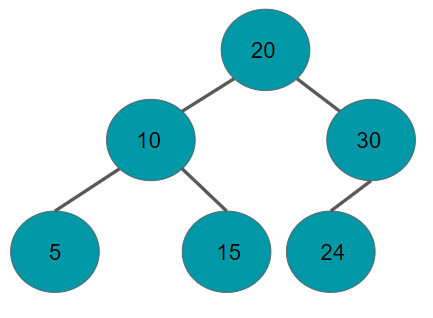
Step 4:



Step 5:



Step 6:



1. Checking whether a binary tree is both strictly binary tree and complete binary tree

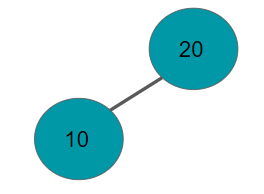


Steps of creating this binary tree

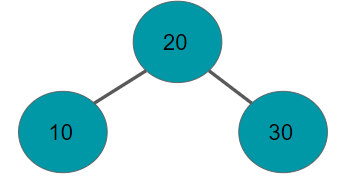
Step 1:



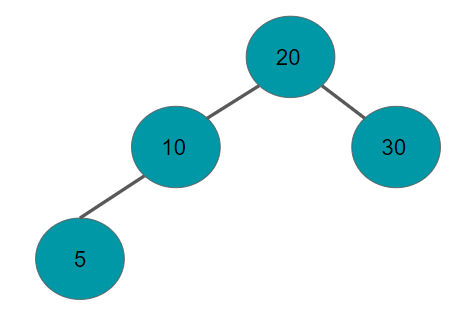
Step 2:



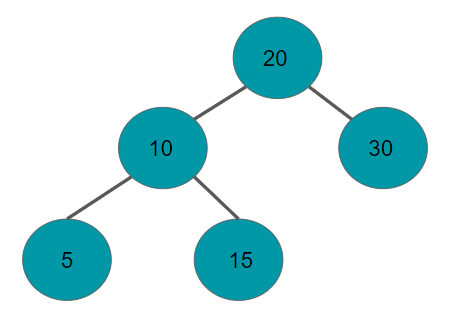
Step 3:



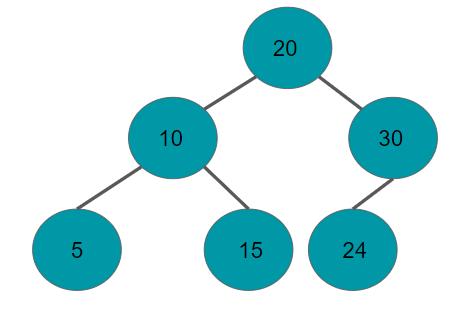
Step 4:



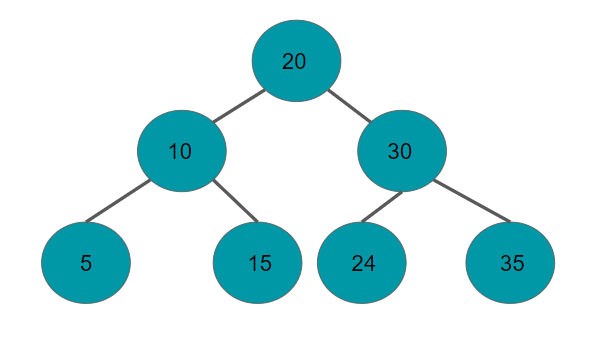
Step 5:



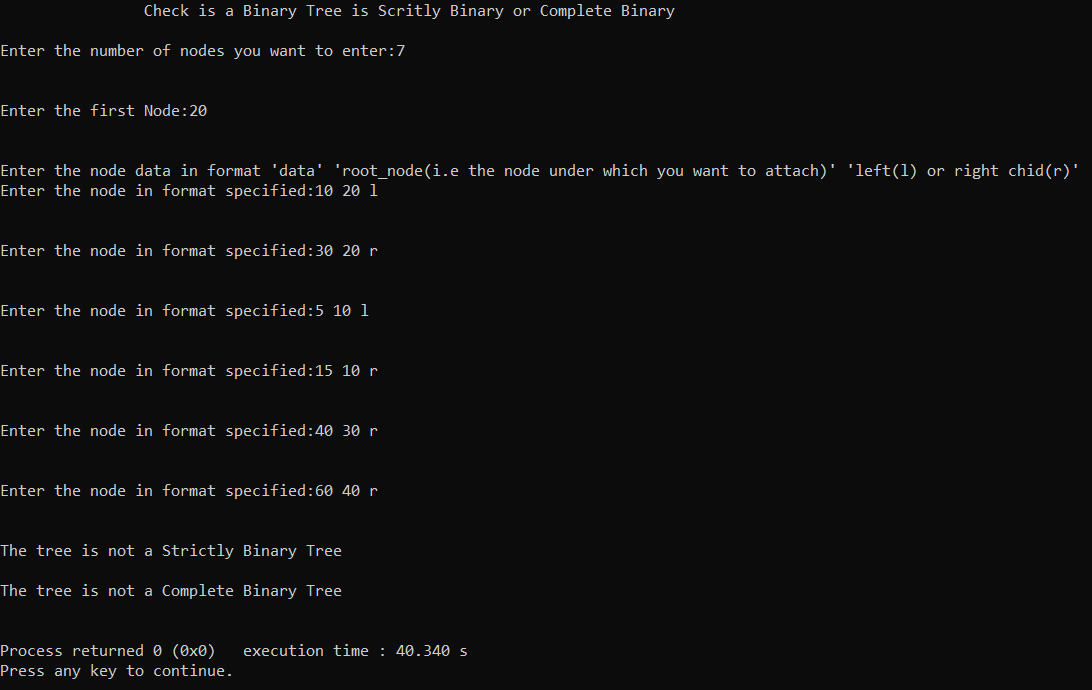
Step 6:



Step 7:



1. Checking whether the binary tree is neither a complete binary tree nor a strictly binary tree.

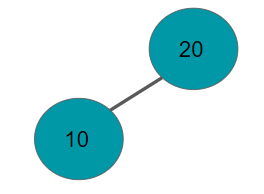


Steps for creating this binary tree

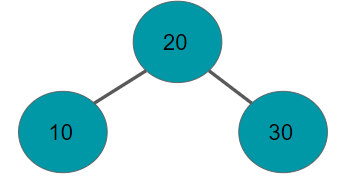
Step 1:



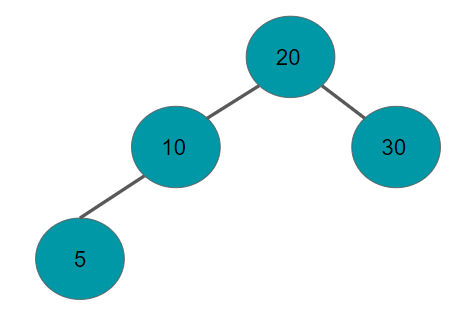
Step 2:



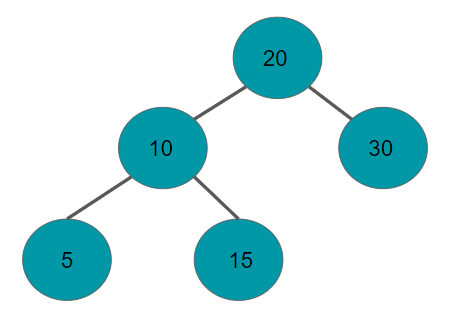
Step 3:



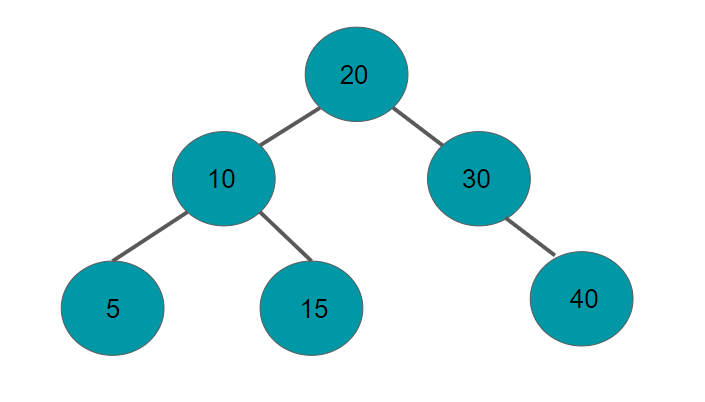
Step 4:



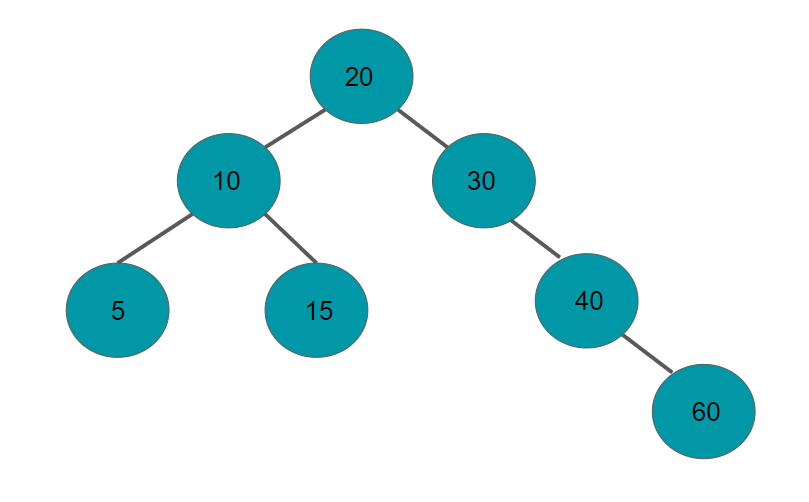
Step 5:



Step 6:



Step 7:

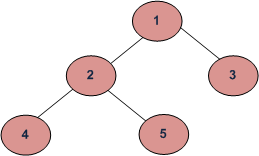


**Tree Traversal**

Traversing a tree means visiting every node in the tree. Linear data structures like arrays, stacks, queues, and linked lists have only one way to read the data. But a hierarchical data structure like a tree can be traversed in different ways.

There are basically three traversal techniques for a binary tree. We will be showing the algorithms for each traversal.

Inorder Traversal:



|  |
| --- |
| Algorithm for Inorder Traversal:  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) |

Uses of Inorder

In case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order. To get nodes of BST in non-increasing order, a variation of Inorder traversal where Inorder traversal is reversed can be used.

Example: Inorder traversal for the above-given figure is 4 2 5 1 3.

Pre Order Traversal:

|  |
| --- |
| Algorithm for Preorder Traversal:   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 a prefix expression on of an expression tree.

Example: Preorder traversal for the above given figure is 1 2 4 5 3.

Post Order Traversal

|  |
| --- |
| Algorithm for Postorder Traversal:   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 root. |

Uses of Postorder

Postorder traversal is used to delete the tree. Postorder traversal is also useful to get the postfix expression of an expression tree.

Example: Postorder traversal for the above given figure is 4 5 2 3 1.

Source code

|  |
| --- |
| **#include <stdio.h>** **#include<conio.h>** **#include<stdlib.h>** **#define max 100**  //structure defining the tree  **typedef** **struct node {**  **int** item;  **struct node \*left, \*right;** }Tree;  Tree \*root=**NULL**; //creates new node Tree \*create\_node(**int** num) {  Tree \*new\_node;  new\_node=(Tree \*)**malloc**(**sizeof**(Tree));  new\_node->item = num;  new\_node->left = new\_node->right = **NULL**;  **return** new\_node; };  //inserts as left child **void** insert\_left(Tree \*new\_root,**int** val) {   new\_root->left=create\_node(val); }  //inserts as right child **void** insert\_right(Tree \*new\_root,**int** val){  new\_root->right=create\_node(val); }  //search for the root node given as input from the user **bool** search\_node(Tree \*temp,**int** user\_root, **char** a, **int** num){  **if**(temp == **NULL**)  **return** 0;  **if**(temp->item == user\_root &&(temp->left==**NULL** || temp->right==**NULL**))  {  **if**(a == 'l' || a == 'L')//for left child  {  **if**(temp->left == **NULL**)  {  insert\_left(temp, num);  **return** 1;  }  **else**   **return** 0;  }  **else**   {  **if**(temp->right == **NULL**)//for right child  {  insert\_right(temp, num);  **return** 1;  }  **else**   **return** 0;  }    }  **else**  **return** search\_node(temp->left, user\_root, a, num) || search\_node(temp->right, user\_root, a, num); }  //creates the tree **void** create\_tree(){  **char** a;  **printf**("\n\nEnter the node in format specified:");  **int** num,user\_root;  **scanf**("%d %d %c",&num,&user\_root,&a);  **bool** found = 0;  **if**(a=='l'||a=='L'){  found = search\_node(root,user\_root,a,num);    }  **else** **if**(a=='r'||a=='R'){  found =search\_node(root,user\_root,a,num);  }  **if**(!found)  **printf**("The parent node doesn't exist\n"); }   //Funtion for Inorder traversal **void** inorder\_traversal(Tree\* root) {  **if** (root == **NULL**)  **return**;  inorder\_traversal(root->left);  **printf**("%d ", root->item);  inorder\_traversal(root->right); }  //Funtion For Preorder Traversal  **void** preorder\_traversal(Tree\* root) {  **if** (root == **NULL**)   **return**;  **printf**("%d ", root->item);  preorder\_traversal(root->left);  preorder\_traversal(root->right); }  //Function for Postorder Traversal  **void** postorder\_traversal(Tree\* root) {  **if** (root == **NULL**)   **return**;  postorder\_traversal(root->left);  postorder\_traversal(root->right);  **printf**("%d ", root->item); }  // Driver code **int** main() {  system("cls");  **printf**("\t\tInorder,Preorder,Postorder traversal\n\n");  **int** val,num\_nodes,index=0;  **printf**("Enter the number of nodes you want to enter:");  **scanf**("%d",&num\_nodes);  **printf**("\n\nEnter the first Node:");  **scanf**("%d",&val);  root=create\_node(val);  **printf**("\n\nEnter the node data in format 'data' 'parent\_node(i.e the node under which you want to attach)' 'left(l) or right chid(r)'");  **for**(**int** i=1;i<num\_nodes;i++){  create\_tree();  }   **printf**("\n\nInorder traversal \n");  inorder\_traversal(root);   **printf**("\n\nPreorder traversal \n");  preorder\_traversal(root);   **printf**("\n\nPostorder traversal \n");  postorder\_traversal(root); } |

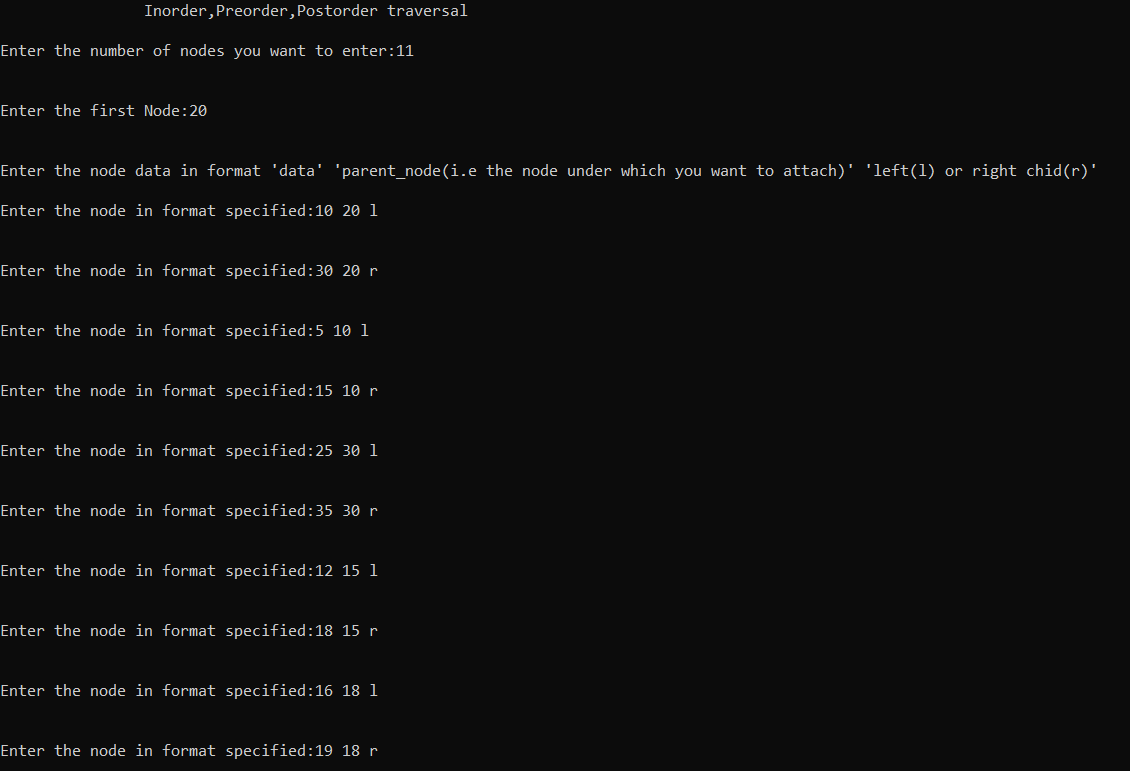
**Output Console**

In the output console,

1. Specify the number of nodes
2. Write down the root node
3. Write the non-root nodes in the following format

|  |
| --- |
| <root> <parent\_node> <left (l) / right (r) > |

Run - I

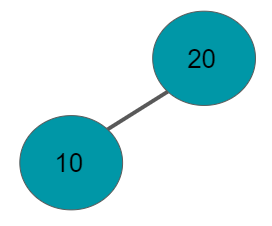


Steps for creating this binary tree:

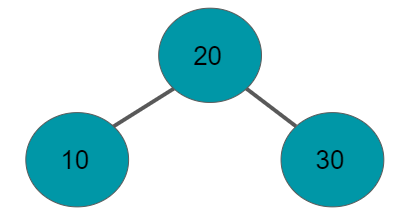
Step 1:



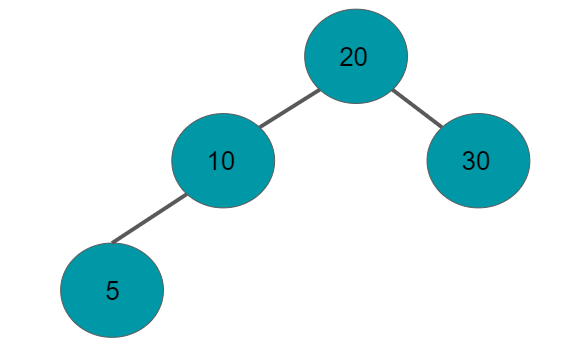
Step 2:



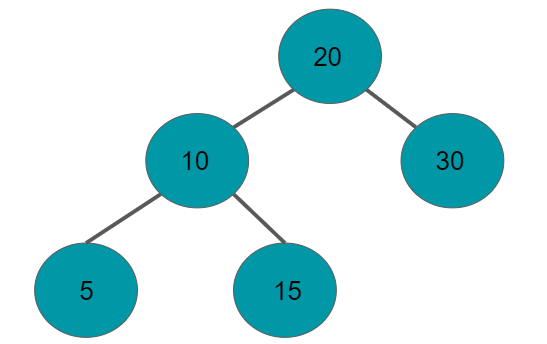
Step 3:



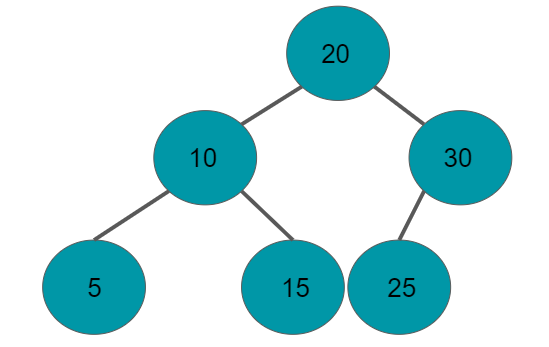
Step 4:



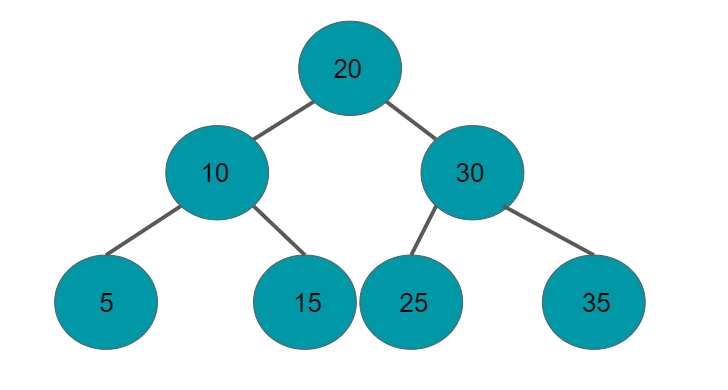
Step 5:



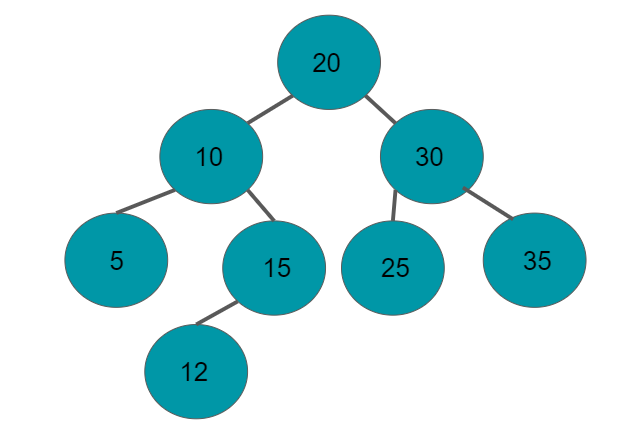
Step 6:



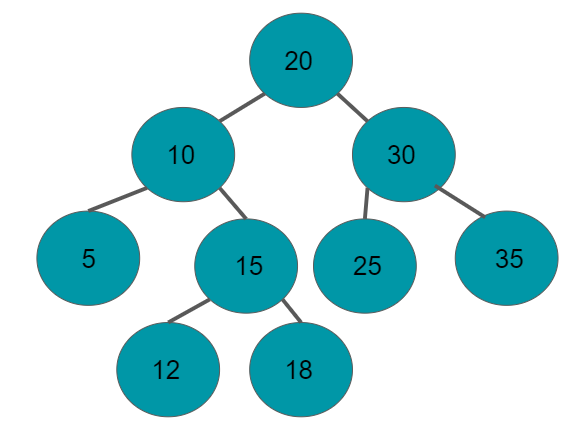
Step 7:



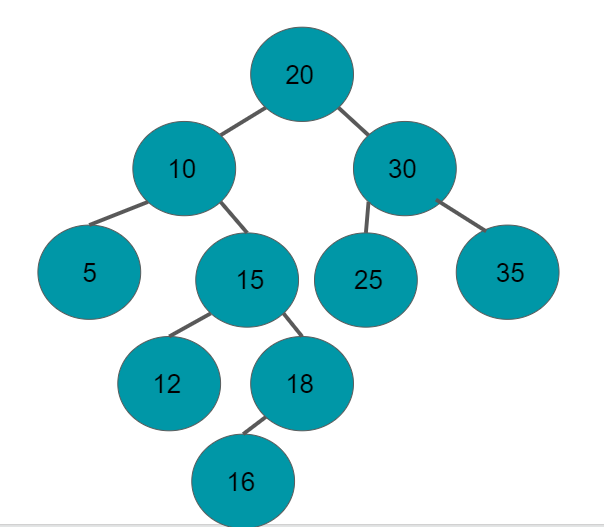
Step 8:



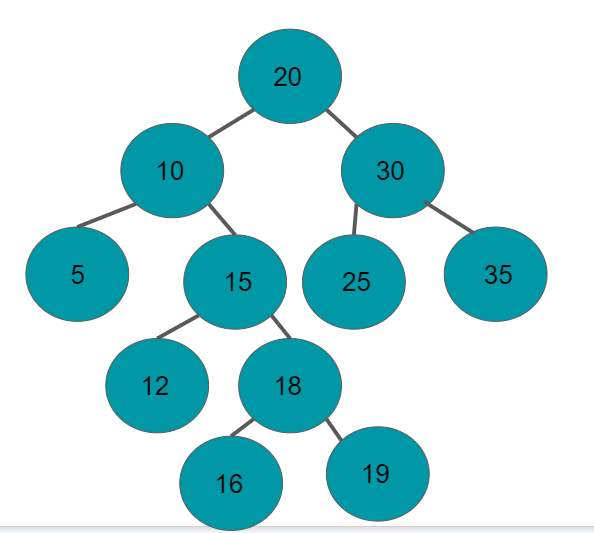
Step 9:



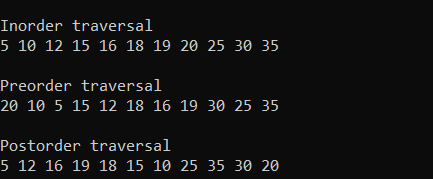
Step 10:



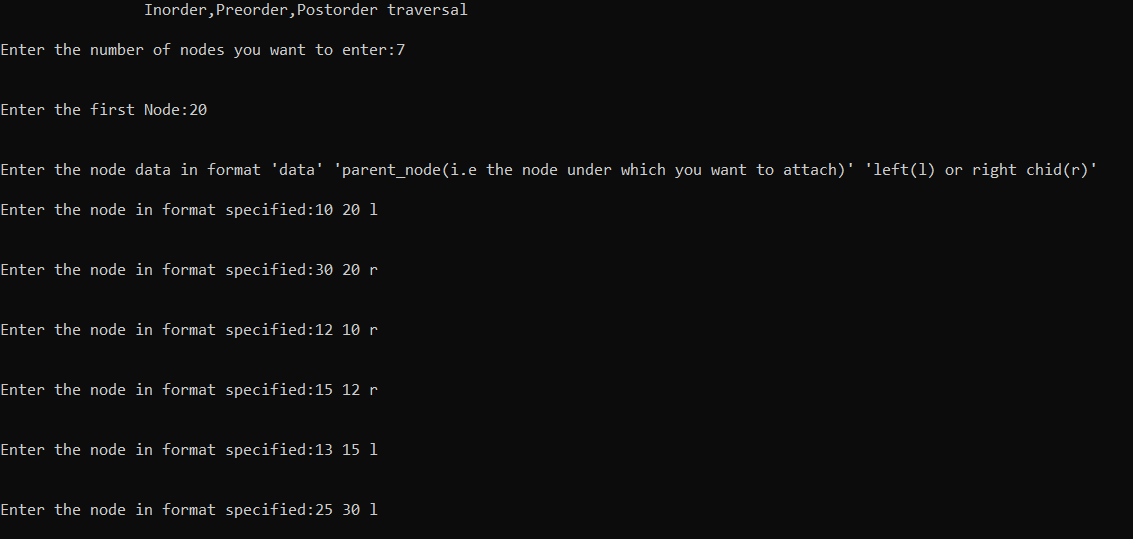
Step 11:



RESULTS



RUN - II

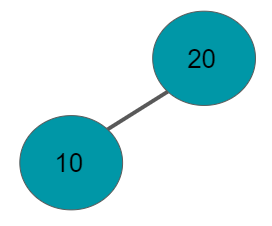


Steps for creating this tree:

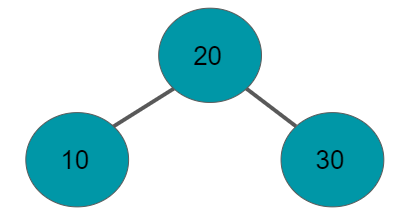
Step 1:



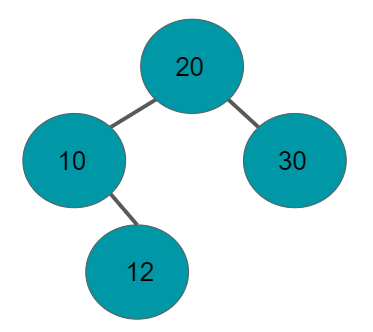
Step 2:



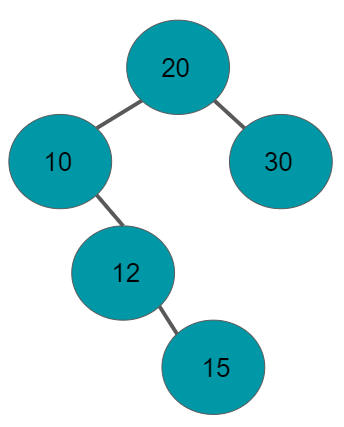
Step 3:



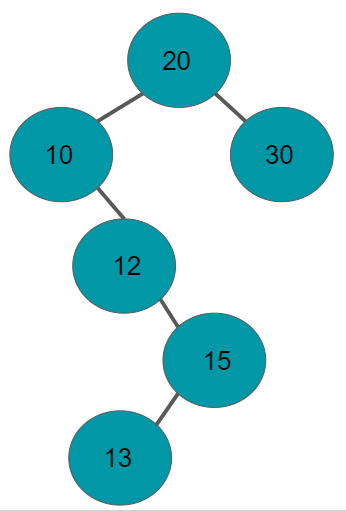
Step 4:



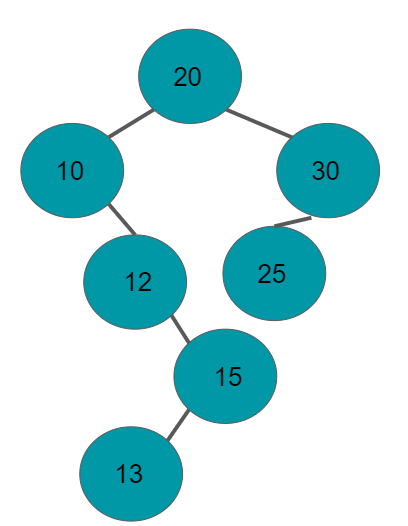
Step 5:



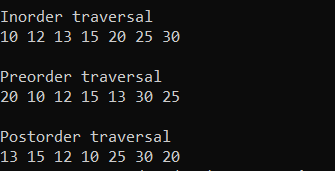
Step 6:



Step 7:



RESULTS

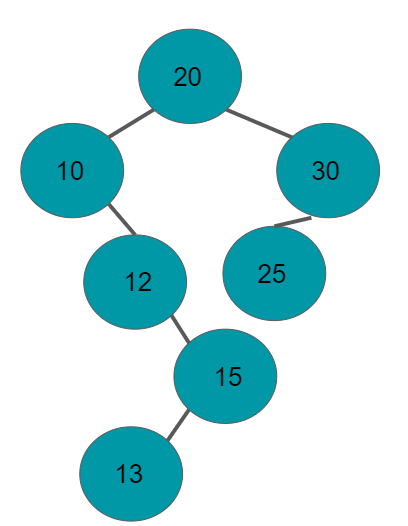


**3. Based on a random set of inputted integers create a binary search tree with the integers as the nodes. Delete at least a) one leaf node, b) a node with one child and c) a node with two children from the tree you have created.**

In a binary search tree,

1. All nodes of left subtrees are less than the root node.
2. All nodes of right subtrees are more than the root node.
3. The sub-subtrees, sub-sub-subtrees and so on until leaf nodes obeys the above two properties.

Example

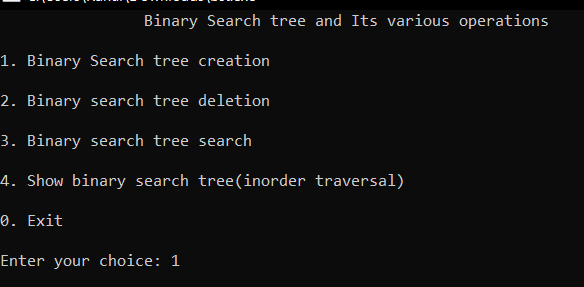


Source Code:

|  |
| --- |
| **#include<stdio.h>** **#include<stdlib.h>** **#include<conio.h>**  **typedef** **struct node {**  **int** item;  **struct node \*left, \*right;** }Tree;  Tree\* root = **NULL**; //creates new node Tree \*create\_node(**int** num){  Tree \*new\_node;  new\_node=(Tree \*)**malloc**(**sizeof**(Tree));  new\_node->item = num;  new\_node->left = new\_node->right = **NULL**;  **return** new\_node; };  //Funtion for Inorder traversal **void** inorder\_traversal(Tree\* root) {  **if** (root == **NULL**)  **return**;  inorder\_traversal(root->left);  **printf**("%d ", root->item);  inorder\_traversal(root->right); }  Tree\* insert\_node(Tree\* root, **int** val){  **if** (root == **NULL**) //If the tree is empty, return a new,single node  **return** create\_node(val);   **if** (val < root->item)  root->left = insert\_node(root->left, val);  **else**  root->right = insert\_node(root->right, val);  **return** root;//return the (unchanged) root pointer   }  //Function to find the inorder successor struct node \*inorder\_successor(Tree \*node) {  Tree\*current = node;  **while** (current && current->left != **NULL**)//Finds the leftmost leaf  current = current->left;  **return** current; }  Tree\* delete\_node(Tree \*root,**int** val){  **if**(root==**NULL**){  **printf**("\n\nValue not found");  **return** root;  }  **if**(val < root->item)  root->left=delete\_node(root->left,val);  **else** **if**(val > root->item)  root->right=delete\_node(root->right,val);  **else**{  **if**(root->left==**NULL** && root->right==**NULL**){//Node with no elements  **free**(root);  **return** **NULL**;  }  **else** **if**(root->left == **NULL**) {//Node with one element  Tree\* temp = root->right;  **free**(root);  **return** temp;  }  **else** **if** (root->right == **NULL**) {//Node with one element  Tree\* temp = root->left;  **free**(root);  **return** temp;  }  //Node with more than two children   Tree\* temp = inorder\_successor(root->right);  root->item = temp->item;//Placing the inorder successor in position of the node to be deleted  root->right = delete\_node(root->right, temp->item);//Deleting the inorder successor  }  **return** root; }  **void** search(Tree \*root, **int** val, Tree \*parent){  **if**(root==**NULL**){  **printf**("\n\nValue not found");  **return**;  }  **if**(val==root->item){  **if**(parent==**NULL**)  **printf**("\n\nThe node with value %d is root node",val);  **else** **if**(val<parent->item)  **printf**("\n\nThe node node with value %d is the left child of the parent node %d",val,parent->item);  **else**   **printf**("\n\nThe node node with value %d is the right child of the parent node %d",val,parent->item);  **return**;  }  **else** **if**(val<root->item)  **return** search(root->left,val,root);  **return**(search(root->right,val,root)); }  **void** create\_tree(){  system("cls");  **int** val;  **char** ch;  **printf**("\t\tBinary search tree creation");  **do**{  **printf**("\n\nEnter the node value:");  **scanf**("%d",&val);  root=insert\_node(root,val);  **printf**("\nThe inorder traversal of the tree is------> ");  inorder\_traversal(root);  **printf**("\nDo you want to enter more nodes(press 'Y' for yes and 'N' for no):");  fflush(**stdin**);  **scanf**("%c",&ch);  }**while**(ch=='Y'||ch=='y'); } **int** main(){  **int** ch;  **do**{  system("cls");  **int** val;  Tree\* node=**NULL**;  **printf**("\t\tBinary Search tree and Its various operations\n\n");  **printf**("1. Binary Search tree creation\n\n");  **printf**("2. Binary search tree deletion\n\n");  **printf**("3. Binary search tree search\n\n");  **printf**("4. Show binary search tree(inorder traversal)\n\n");  **printf**("0. Exit\n\n");  **printf**("Enter your choice: ");  **scanf**("%d",&ch);  **switch** (ch)  {  **case** 1:   create\_tree();  **break**;  **case** 2: system("cls");  **printf**("\t\tBinary Search Tree Deletion operation\n\n");  **printf**("Enter the value of the Node to be deleted: ");  **scanf**("%d",&val);  **printf**("\n\nThe Binary Search Tree before Deletion---->");  inorder\_traversal(root);  delete\_node(root,val);  **printf**("\n\nThe Binary Search Tree after Deletion---->");  inorder\_traversal(root);  getch();  **break**;  **case** 3: system("cls");  **printf**("\t\tBinary Search Tree Searching operation\n\n");  **printf**("Enter the value of the Node to be Searched: ");  **scanf**("%d",&val);  search(root,val,**NULL**);  getch();  **break**;  **case** 4:  system("cls");  **printf**("\t\tDisplaying the Binary search tree\n\n");  inorder\_traversal(root);  getch();  **break**;  **case** 0: **exit**(0);  **break**;  **default**: **printf**("Enter valid choice");  }   }**while**(1); } |

**OUTPUT CONSOLE:**

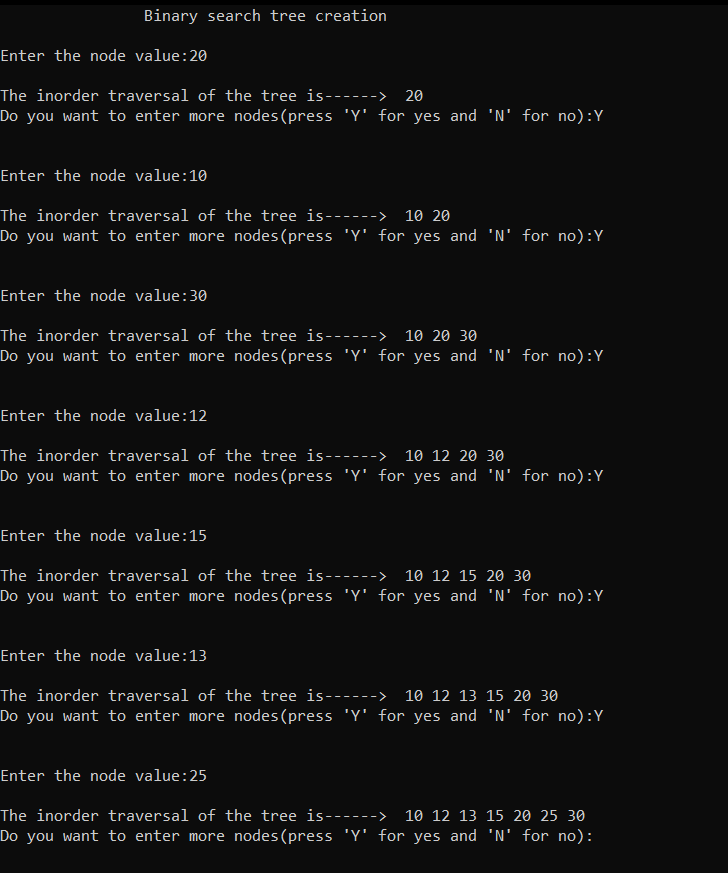
*Menu template*



Inserting elements to a binary search tree

|  |
| --- |
| Algorithm: 1. Start from the root. 2. Compare the inserting element with root, if less than root, then recurse for left, else recurse for right.  3. After reaching the end, just insert that node at left(if less than current) else right. |

|  |
| --- |
|  |

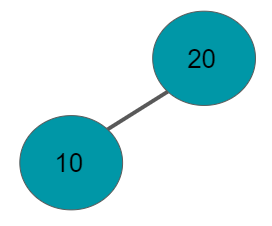


*Steps on how this tree has been created*

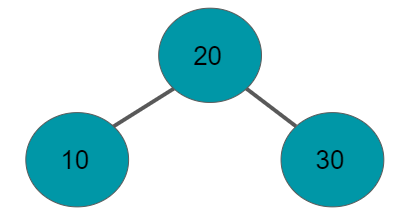
Step 1: Element 20 has been inserted



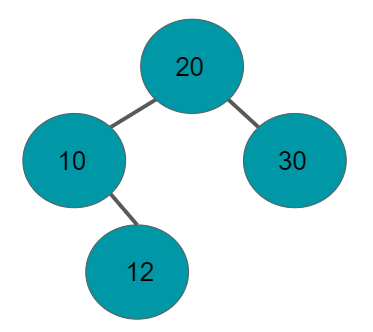
Step 2: Element 10 is inserted into the binary search tree. Since 10 is less than 20 it will be added as a left child to 20.



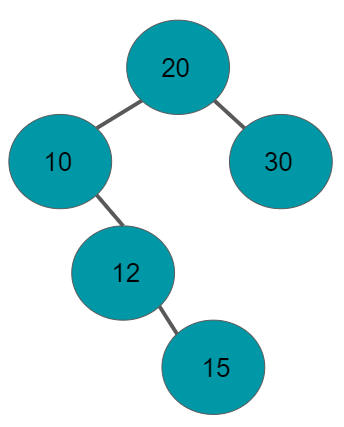
Step 3: Element 30 is inserted into the binary search tree. Since 30 is more than 20 it will be added as a right child to 20.



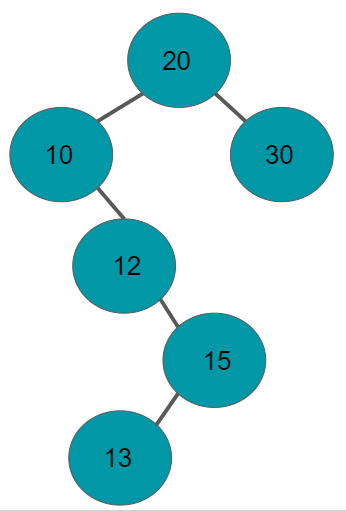
Step 4: Element 12 is inserted into the binary search tree. Since 12 is more than 10 but less than 20 it will be added as a right child to 10.



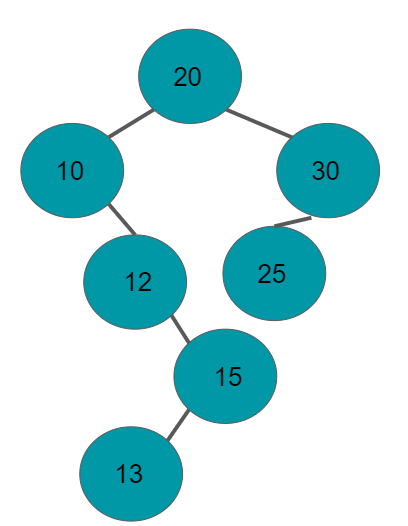
Step 5: Element 15 is inserted into the binary search tree. Since 15 is more than 12 and 10 but less than 20 it will be added as a right child to 12.



Step 6: Element 13 is inserted into the binary search tree. Since 13 is more than 12 and 10 but less than 15 it will be added as a left child to 15.

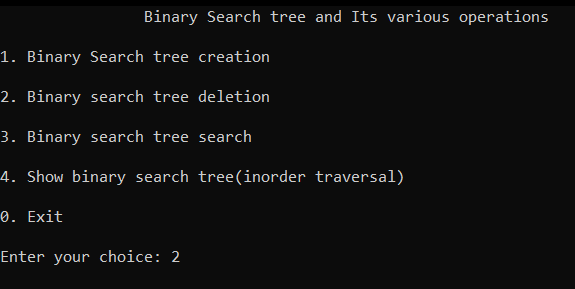


Step 7: Element 25 is inserted into the binary search tree. Since 25 is more than 20 but less than 30 it will be added as a left child to 30.



Deleting elements from the binary search tree

To delete a node from the binary search tree we will select the option 2 from the menu



We will demonstrate tree deletion operations performed in the binary search tree

|  |
| --- |
| 1. Delete a leaf node 2. Delete a node with only one child. 3. Delete a node with two child.   Task 1: Delete a leaf node  Steps to follow:      Binary Search Tree before deletion:    Binary Search Tree after deletion of the leaf node ‘13’    Task 2: Delete a node with only one child    Binary Search Tree before deletion    Binary Search Tree after deletion:    Task 3: Delete a node with two children      Binary search tree before deletion:    Binary Search Tree after deletion    Searching a Node:    To search a node from the binary search tree we will select the option 3 from the menu     1. Searching of root node → 20        1. Searching for a left child root. In our case it is node with element 10        1. Searching for a right child root. In our case it is node with element 30.        1. Searching for the node 15.        1. Searching for the node value which is not present in the binary search tree. |

**CONCLUSION**

In this laboratory, we have successfully been introduced with binary trees and binary search trees and kept a very important point in our mind that trees are hierarchical data structures. We understood various categories of a binary tree including strictly binary tree and complete binary tree.

The most primary rule has been incorporated while constructing the binary trees that nodes are inserted from left to right at the same level.

We have studied in detail about the various traversal techniques in a binary tree, such as inorder, pre-order and post-order traversals.

We performed several functions with binary search trees like constructing, deleting, searching and the most significant of all this is searching. We observed that the order of searching is in O(h), where h is the height of the binary search tree.

There are different categories of binary trees and binary search trees keeping the time and space complexity, efficiency and applications in concern. Some of them are AVL Trees, Red-Black Trees, Threaded Binary Trees, K-D Trees, etc.

Some of the applications of Binary Trees are:

1. In large stream of datas
2. Indexing
3. Computer Networking
4. Operating Systems