Experiment No. : 5

Title: Implementation of Dynamic Binary Search Tree using Doubly Linked List (DLL)

Batch: SY-IT(B3) **Roll No.:16010423076** **Experiment No.: 5**

Aim: Write a program for following operations on Binary Search Tree using Doubly Linked List (DLL).

1) Create empty BST,
2) Insert a new element on the BST
3) Search for an element on the BST
4) Delete an element from the BST
5) Display all elements of the BST

Resources needed: C/C++/JAVA editor and compiler

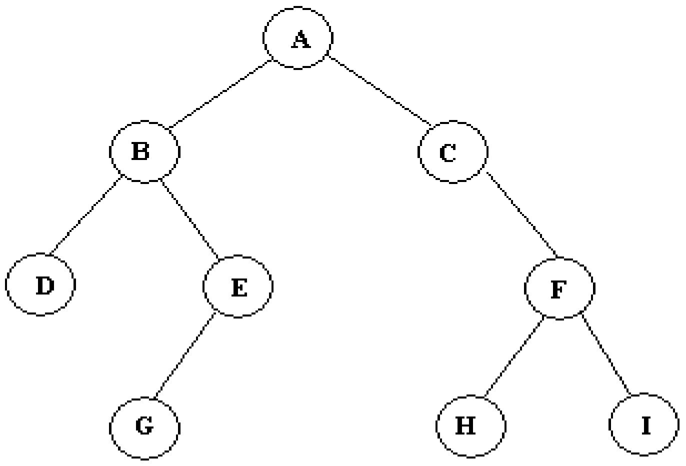
Theory

Binary Tree :-

A binary tree is made of nodes, where each node contains a "left" reference, a "right" reference, and a data element. The topmost node in the tree is called the root.

Every node (excluding a root) in a tree is connected by a directed edge from exactly one other node. This node is called a parent. On the other hand, each node can be connected to arbitrary number of nodes, called children. Nodes with no children are called leaves, or external nodes. Nodes which are not leaves are called internal nodes. Nodes with the same parent are called siblings.

Example



Figure(a): Binary Tree Example

Traversals :

A traversal is a process that visits all the nodes in the tree. Since a tree is a nonlinear data structure, there is no unique traversal. We will consider several traversal algorithms with we group in the following two kinds.

1. depth-first traversal
2. breadth-first traversal

There are three different types of depth-first traversals, :

- PreOrder traversal - visit the parent first and then left and right children;
- InOrder traversal - visit the left child, then the parent and the right child;
- PostOrder traversal - visit left child, then the right child and then the parent.

There is only one kind of breadth-first traversal--the level order traversal. This traversal visits nodes by levels from top to bottom and from left to right.

Consider an example the following tree and its four traversals:

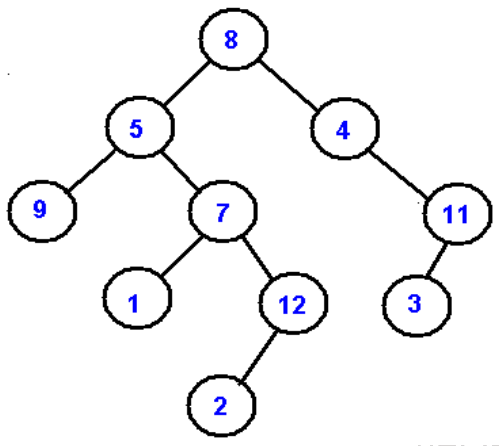
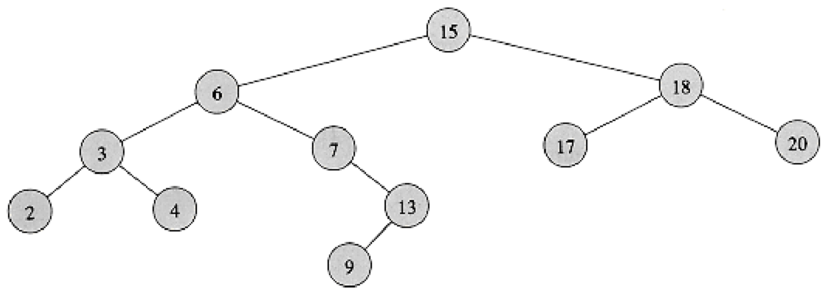


Figure 5.3: Example of Tree Traversals

PreOrder - 8, 5, 9, 7, 1, 12, 2, 4, 11, 3
InOrder - 9, 5, 1, 7, 2, 12, 8, 4, 3, 11
PostOrder - 9, 1, 2, 12, 7, 5, 3, 11, 4, 8
LevelOrder - 8, 5, 4, 9, 7, 11, 1, 12, 3, 2

Binary Search Tree(BST): It is a binary tree where elements are stored in a particular order of left child is less than the parent and parent is less than the right child. This small modification makes BST efficient for searching. It is also called as BST property.
Example of BST following fig is the example BST whose root is 8 and follows the BST property.



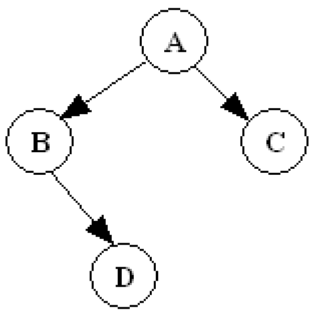
Methods for storing binary trees

Binary trees can be constructed from programming language primitives in several ways.

Nodes and references

In a language with records and references, binary trees are typically constructed by having a tree node structure which contains some data and references to its left child and its right child. Sometimes it also contains a reference to its unique parent. If a node has fewer than two children, some of the child pointers may be set to a special null value, or to a special sentinel node.

Binary Search Tree can be implemented as a linked data structure in which each node is an object with two pointer fields. The two pointer fields *left and right* points to the nodes corresponding to the left child and the right child NULL in any pointer field signifies that there exists no corresponding child.



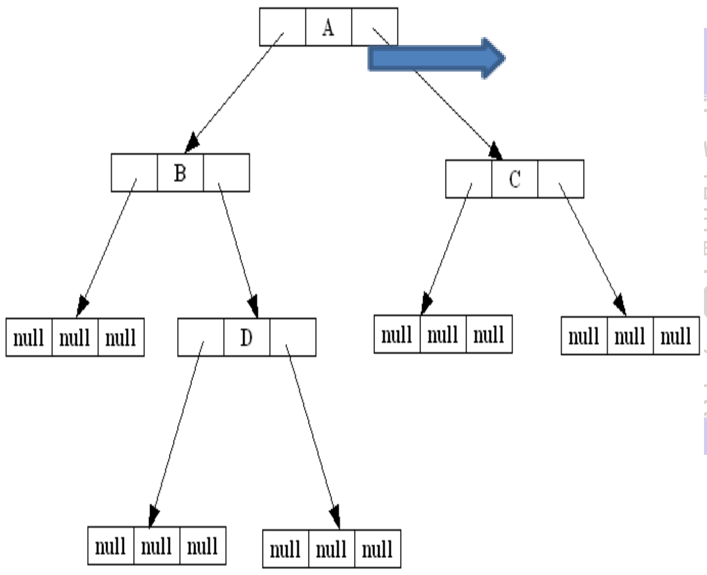


Figure : Binary Search Trees example

Algorithm :

1. createBST() – This function should create a ROOT pointer with NULL value as empty BST.
2. insert(typedef newelement) – This void function should take a newelement as an argument to be inserted on an existing BST following the BST property.
3. typedef search(typedef element) – This function should search for specified element on the non-empty BST and return a pointer to it.
4. typedef delete(typedef element) – This function searches for an element and if found deletes it from the BST and returns the same.

5. **typedef getParent(typedef element) -** This function searches for an element and if found, returns its parent element.
6. DisplayInorder( ) – This is a void function which should go through non- empty BST, traverse it in inorder fashion and print the elements on the way.

NOTE : All functions should be able to handle boundary(exceptional) conditions.

Activity: Write pseudocode for each method and implement the same.

Results: A program depicting the correct behaviour of BST and capable of handling all possible exceptional conditions and the same is reflecting clearly in the output.

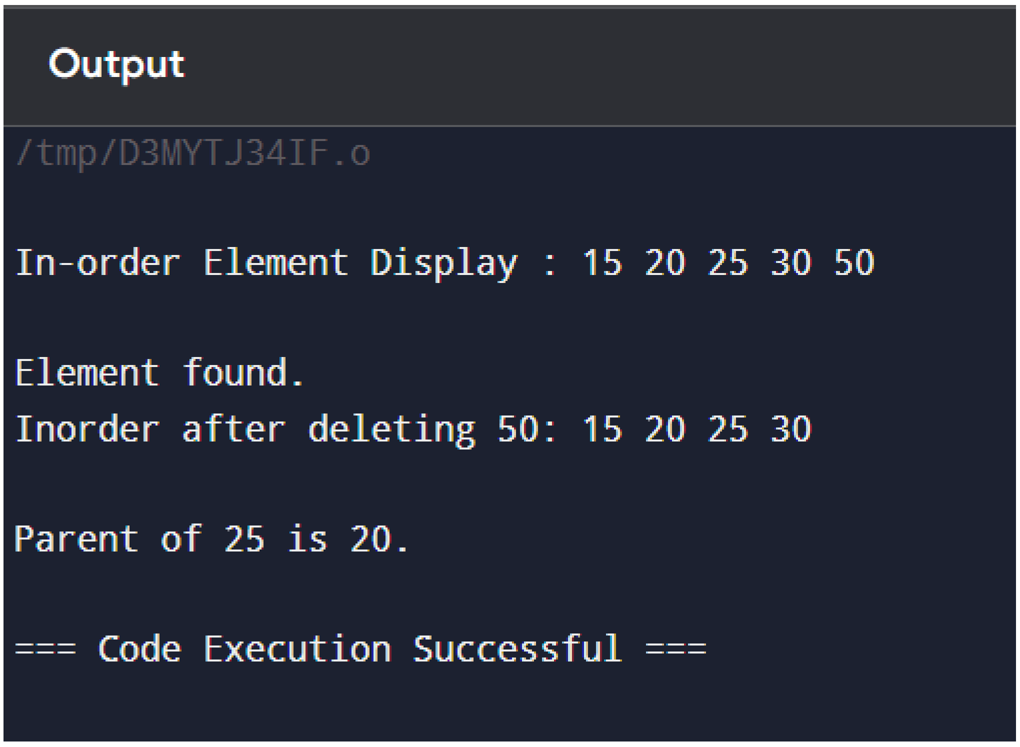
Program with output:

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

//node structure definition
typedef struct Node
{
   int data;
   struct Node *left, *right, *parent;
}Node;



//create node function
Node* createNode(int data,Node* parent)
{
Node* newnode =  (Node*)malloc(sizeof(Node));
newnode->data=data;
newnode->left=newnode->right=NULL;
newnode->parent = parent;
}

Node* createBinarySearchTree(){
   return NULL;
}



//insert function
Node* insert(Node* root, int data)
```

```
{
if(root==NULL){
   return createNode(data, NULL);
}
//data is less than root value
if(data<root->data){
   if(root->left==NULL){
      root->left = createNode(data,root);
   }
   else{
      insert(root->left,data);
   }
}

//data is greater than root value
else{
   if(root->right==NULL){
      root->right = createNode(data,root);
   }
   else{
      insert(root->right,data);
   }
}
return root;
}


//search any element
Node* search(Node* root, int data)
{
   if(root==NULL || root->data==data){
      return root;
   }
   if(data < root->data){
      return search(root->left,data);

   }
   else{
      return search(root->right,data);
   }
}
```

```
//function to find minimum value
//keep going left until end
Node* findMin(Node* root){
    while(root->left != NULL){
        root = root->left;
    }
    return root;
}


//delete element
Node* delete(Node* root,int data){
   if(root==NULL){
       return root;
   }
   if(data < root->data){
       root->left = delete(root->left, data);
   }
   else if(data > root->data){
       root->right = delete(root->right,data);
   }
   //has one bachha
   else{
       if(root->left==NULL){
           Node* temp = root->right;
           if(temp) temp->parent = root->parent;
           free(root);
           return(temp);
       }
       else{
           Node* temp = root->left;
           if(temp) temp->parent = root->parent;
           free(root);
           return(temp);
       }
   }
}



//get parent node
Node* getParent(Node* root, int data){
    Node* node = search(root,data);
    if(node&&node->parent){
        return node->parent;
    }
    return NULL;
}

//In-order display
void displayinOrder(Node* root){
    if(root!=NULL){
        displayinOrder(root->left);
        printf("%d ",root->data);
        displayinOrder(root->right);
    }
}


int main()
{
    Node* root = createBinarySearchTree();

    //insert element
    root = insert(root, 50);
    root = insert(root, 30);
    root = insert(root, 20);
    root = insert(root, 15);
    root = insert(root, 25);

    //in-order display
    printf("\nIn-order Element Display : ");
    displayinOrder(root);
    printf("\n");

    //Search for an element
    int searchvalue = 30;
    Node* found = search(root,searchvalue);
    if(found){
        printf("\nElement found.");
    }
    else{
        printf("\nElement not found.");
    }

     // Delete an element
    root = delete(root, 50);
    printf("\nInorder after deleting 50: ");
    displayinOrder(root);
    printf("\n");

    //get the parent of an element
    Node* parent = getParent(root,25);
    if(parent){
        printf("\nParent of 25 is %d.",parent->data);
    }
    else{
        printf("\nParent not found");
    }
}
```

Output :



Output

```
/tmp/D3MYTJ34IF.o

In-order Element Display : 15 20 25 30 50

Element found.
Inorder after deleting 50: 15 20 25 30

Parent of 25 is 20.

=== Code Execution Successful ===
```

Outcomes: CO4. Demonstrate sorting and searching methods.

Conclusion:

From this experiment, I learned how to implement a Binary Search Tree (BST) in C, including inserting, searching, and deleting nodes efficiently. I gained a deeper understanding of how to manage node relationships, especially using pointers to keep track of parent-child connections. This exercise helped me see the practical benefits of BSTs in organizing and manipulating data.

Grade: AA / AB / BB / BC / CC / CD /DD

Signature of faculty in-charge with date

References:

Books/ Journals/ Websites:

- Y. Langsam, M. Augenstin and A. Tannenbaum, "Data Structures using C", Pearson Education Asia, 1st Edition, 2002.
- https://ds1-iiith.vlabs.ac.in/exp/binary-search-trees/index.html