**Program 6**

**Aim :** To implement BST.

**Theory :** In computer science, **binary search trees** (**BST**), sometimes called **ordered** or **sorted binary trees**, are a particular type of container: data structures that store "items" (such as numbers, names etc.) in memory. They allow fast lookup, addition and removal of items, and can be used to implement either dynamic sets of items, or lookup tables that allow finding an item by its *key* (e.g., finding the phone number of a person by name).

Binary search trees keep their keys in sorted order, so that lookup and other operations can use the principle of binary search: when looking for a key in a tree (or a place to insert a new key), they traverse the tree from root to leaf, making comparisons to keys stored in the nodes of the tree and deciding, on the basis of the comparison, to continue searching in the left or right subtrees. On average, this means that each comparison allows the operations to skip about half of the tree, so that each lookup, insertion or deletion takes time proportional to the logarithm of the number of items stored in the tree. This is much better than the linear time required to find items by key in an (unsorted) array, but slower than the corresponding operations on hash tables.

Binary Search Tree, 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.  
  There must be no duplicate nodes.

The above properties of Binary Search Tree provide an ordering among keys so that the operations like search, minimum and maximum can be done fast. If there is no ordering, then we may have to compare every key to search a given key.

**Algorithm :**

Insert(node \* root, int data)

1. Check if node already exists. If so then do nothing.
2. If root is null then :
3. Make a newnode
4. Set it’s data to given data
5. Return the newnode.
6. If root->data > data :
7. root->left = recursive call to left subtree.
8. Else
9. root->right = recursive call to right subtree.

Delete(node \* root, int data)

1. If root is null :
2. return root
3. if root->data > data :
4. root->left = recursive call to left subtree.
5. else if root->data < data :
6. root->right = recursive call to right subtree.
7. Else :
8. If root->left is null :
9. Initialise node \* temp = root->left.
10. Free(root)
11. Return temp.
12. Else If root->right is null :
    1. Initialise node \* temp = root->right.
    2. Free(root)
    3. Return temp.
13. Else :
14. Initialise node \* temp = min in right subtree.
15. Root->data = temp->data
16. Root->right = delete(root->right, temp->data).
17. Return root.

Search(node \* root, int data)

1. If root is null
2. return 0
3. If root->data > data:
4. int ans = recursive call to left subtree.
5. Return ans.
6. Else if root->data<data:
   1. int ans = recursive call to right subtree.
   2. Return ans.
7. Else:
8. Return 1.

**Program :**

#include<stdio.h>

#include<stdlib.h>

struct node{

int data;

struct node \* left;

struct node \* right;

};

int search(struct node \* , int);

struct node \* min(struct node \*);

struct node \* insert(struct node \* , int);

struct node \* delete(struct node \* , int);

int main(){

struct node \* root = NULL;

int choice;

do{

printf("\n1. Insert\n");

printf("2. Delete\n");

printf("3. Search\n");

printf("4. Exit\n");

printf("\nEnter your choice : ");

scanf("%d",&choice);

switch(choice){

case 1 :

{

int data = 0;

printf("Enter value to be inserted : ");

scanf("%d" , &data);

int isPresent = search(root,data);

if(isPresent)

printf("\n%d already exists in tree\n", data);

else{

root = insert(root,data);

printf("\n%d inserted in tree.\n", data);

}

break;

}

case 2 :

{

int data = 0;

printf("Enter value to be deleted : ");

scanf("%d",&data);

int isPresent = search(root,data);

if(!isPresent)

printf("\n%d can’t be deleted.\n",data);

else{

root = insert(root,data);

printf("\n%d deleted from tree.\n", data);

}

break;

}

case 3 :

{

int data;

printf("Enter value to be searched : ");

scanf("%d",&data);

int isPresent = search(root,data);

if(isPresent){

printf("\n%d is present in tree.\n", data);

}

else{

printf("\n%d isn't present in tree.\n", data);

}

break;

}

default:

break;

}

}while(choice>0&&choice<4);

return 0;

}

int search(struct node \* root,int data){

if(root == NULL)

return 0;

if(root->data>data)

return search(root->left,data);

else if(root->data<data)

return search(root->right,data);

return 1;

}

struct node \* min(struct node \* root){

if(root==NULL)

return NULL;

struct node \* curr = root;

while(curr->left!=NULL)

curr=curr->left;

return curr;

}

struct node \* delete(struct node \* root, int data){

if(root == NULL)

return NULL;

if(root->data>data){

root->left = delete(root->left , data);

}

else if(root->data<data){

root->right = delete(root->right , data);

}

else{

if(root->left == NULL){

struct node \* temp = root->right;

free(root);

root = temp;

}

else if(root->right == NULL){

struct node \* temp = root->left;

free(root);

root = temp;

}

else{

struct node \* temp = min(root->right);

root->data = temp->data;

root->right = delete(root->right , root->data);

}

}

return root;

}

struct node \* insert(struct node \* root , int data){

if(root == NULL){

root = (struct node \* )malloc(sizeof(struct node));

root->data = data;

root->left = root->right = NULL;

return root;

}

if(data>root->data){

root->right = insert(root->right,data);

}

else{

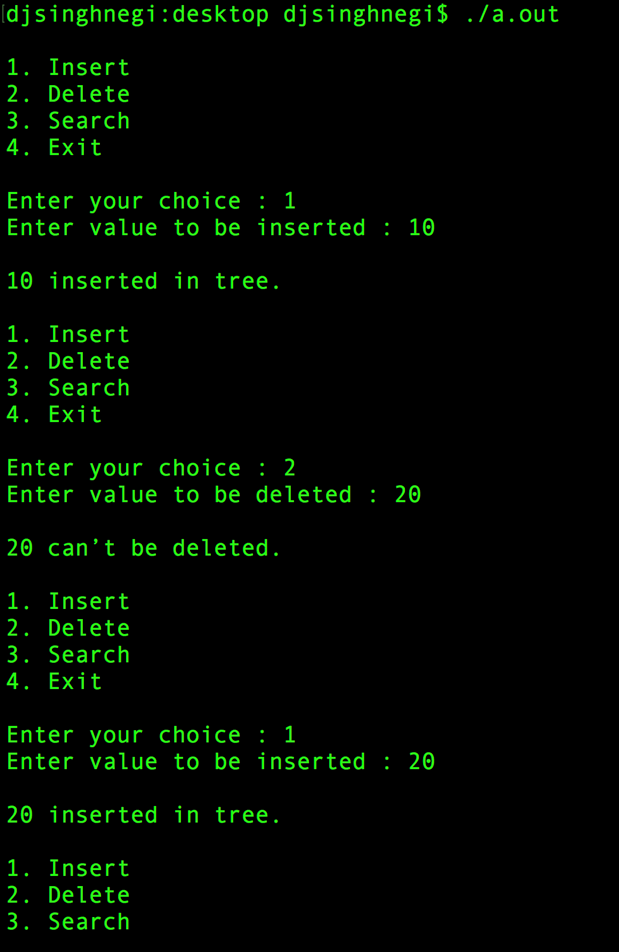
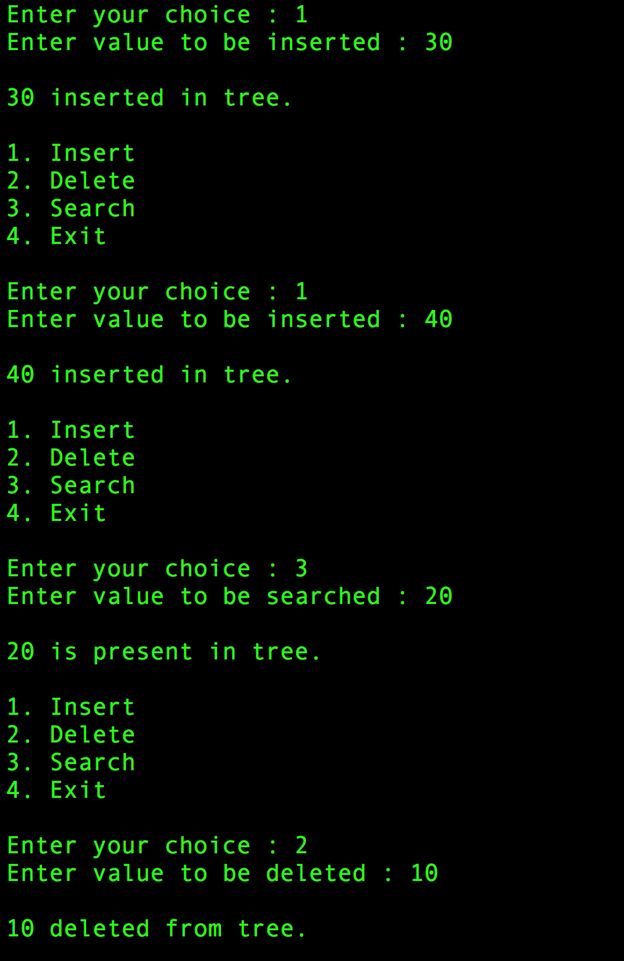
root->left = insert(root->left,data);

}

return root;

}

**Output :**

** **

**Learning :** We learnt about search, insert and delete operations on binary search tree.