**LAB SESSION 7: BINARY SEARCH TREES**

AIM: To implement Binary search trees and perform the listed operations on such trees.

**PROBLEM DEFINITION:**

Develop a C program to create a binary search tree and perform the following operations

1. Insertion of a new element

2. Deletion of an existing element

3. Searching for a given element

4. Perform in order, pre order and post order traversals

5. Find the maximum and minimum value

**THEORY**:

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.

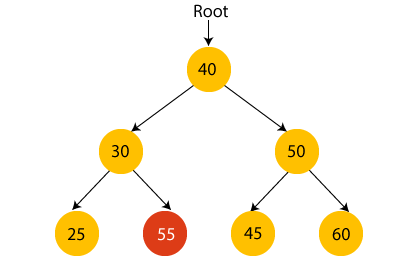
Let's understand the concept of Binary search tree with an example.



In the above figure, we can observe that the root node is 40, and all the nodes of the left subtree are smaller than the root node, and all the nodes of the right subtree are greater than the root node.

Similarly, we can see the left child of root node is greater than its left child and smaller than its right child. So, it also satisfies the property of binary search tree. Therefore, we can say that the tree in **the above image is a binary search tree**.

Suppose if we change the value of node 35 to 55 in the above tree, check whether the tree will be binary search tree or not.



In the above tree, the value of root node is 40, which is greater than its left child 30 but smaller than right child of 30, i.e., 55. So, **the above tree does not satisfy** the property of Binary search tree. Therefore, the above tree is not a binary search tree.

**ALGORITHMS**:

1. Algorithm for inorder traversal:
   * If the tree is empty, return.
   * Traverse the left subtree by recursively calling the inorder function on the left child.
   * Visit the root node.
   * Traverse the right subtree by recursively calling the inorder function on the right child.
2. Algorithm for preorder traversal:
   * If the tree is empty, return.
   * Visit the root node.
   * Traverse the left subtree by recursively calling the preorder function on the left child.
   * Traverse the right subtree by recursively calling the preorder function on the right child.
3. Algorithm for postorder traversal:
   * If the tree is empty, return.
   * Traverse the left subtree by recursively calling the postorder function on the left child.
   * Traverse the right subtree by recursively calling the postorder function on the right child.
   * Visit the root node.
4. Algorithm for searching:
   * If the tree is empty, return NULL.
   * If the target key is less than the current node’s key, recursively search the left subtree.
   * If the target key is greater than the current node’s key, recursively search the right subtree.
   * If the target key is equal to the current node’s key, return the current node.
5. Algorithm for finding the minimum value:
   * Traverse the left subtree of the root node until a leaf node is reached.
   * Return the value of the leaf node.
6. Algorithm for finding the maximum value:
   * Traverse the right subtree of the root node until a leaf node is reached.
   * Return the value of the leaf node.

**PROGRAM AND OUTPUT:**

#include<stdio.h>

#include<stdlib.h>

#include<math.h>

#define MAX 1000

struct node {

struct node \*lchild;

int info;

struct node \*rchild;

};

struct node \*queue[MAX];

int front = - 1, rear = - 1;

struct node \*search(struct node \*ptr,int skey) {

    if(ptr==NULL) {

        printf("Key not found/n");

        return NULL;

    }

    else if(skey<ptr->info)

        return search(ptr->lchild,skey);

    else if(skey>ptr->info)

        return search(ptr->rchild,skey);

    else

        return ptr;

}

struct node \*insert(struct node \*ptr,int ikey) {

    if(ptr == NULL) {

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

        ptr->info = ikey;

        ptr->lchild = NULL;

        ptr->rchild = NULL;

    }

    else if(ikey < ptr->info)

        ptr->lchild = insert(ptr->lchild,ikey);

    else if(ikey > ptr->info)

        ptr->rchild = insert(ptr->rchild,ikey);

    else

        printf("Duplicate key\n");

    return ptr;

}

struct node \*del(struct node \*ptr, int dkey) {

    struct node \*tmp,\*succ,\*parsucc;

    if(ptr== NULL) {

        printf("%d not present in the tree",dkey);

        return ptr;

    }

    if(dkey < ptr->info)

        ptr->lchild = del(ptr->lchild,dkey);

    else if(dkey > ptr->info)

        ptr->rchild = del(ptr->rchild,dkey);

    else {

        if(ptr->lchild != NULL && ptr->rchild != NULL) {

            parsucc = ptr;

            succ = ptr->rchild;

            while(succ->lchild != NULL){

                parsucc = succ;

                succ = succ->lchild;

            }

            ptr->info = succ->info;

            ptr->rchild = del(ptr->rchild,succ->info);

        }

        else {

            tmp = ptr;

            if(ptr->lchild != NULL)

                ptr = ptr->lchild;

            else if(ptr->rchild != NULL)

                ptr = ptr->rchild;

            else ptr = NULL;

            free(tmp);

        }

    }

    return ptr;

}

void preorder(struct node \*p) {

    if(p==NULL)

        return;

    printf("%d ",p->info);

    preorder(p->lchild);

    preorder(p->rchild);

}

void postorder(struct node \*p) {

    if(p==NULL)

        return;

    postorder(p->lchild);

    postorder(p->rchild);

    printf("%d ",p->info);

}

void inorder(struct node \*p) {

    if(p==NULL)

        return;

    inorder(p->lchild);

    printf("%d " ,p->info);

    inorder(p->rchild);

}

int max\_item(struct node\* p)

{

    while(p != NULL && p->rchild != NULL)

        p = p->rchild;

    return p->info;

}

int min\_item(struct node \*p)

{

    while(p != NULL && p->lchild != NULL)

        p = p->lchild;

    return p->info;

}

//printing tree

void insert\_queue(struct node \*item)

{

    if (rear == MAX - 1)

    {

        printf ("Queue Overflow.......\n");

        return;

    }

    if (front == - 1)

        front = 0;

    rear = rear + 1;

    queue[rear] = item;

}

struct node \* del\_queue()

{

    struct node \*item;

    if (front == - 1|| front == rear + 1)

    {

        printf ("Queue underflow.......\n");

        exit(1);

    }

    item = queue[front];

    front = front + 1;

    return item;

}

int queue\_empty ()

{

    if (front == - 1|| front == rear + 1)

        return 1;

    else

        return 0;

}

int height(struct node\* p){

    int lh, rh;

    if(p == NULL)

        return 0;

    lh = 1 + height(p->lchild);

    rh = 1 + height(p->rchild);

    if(lh > rh)

        return lh;

    else

        return rh;

}

void give\_space(int space, struct node \*ptr, int mode){

    if(mode == 1)

    {

        if(ptr->lchild == NULL)

        {

            for(int i = 0; i < space-1; i++)

                printf("  ");

        }

        else

        {

            space /= 2;

            for(int i = 0; i < space-1; i++)

                printf("  ");

            for(int i = 0; i < space; i++)

                printf("--");

        }

    }

    if(mode == 2)

    {

        if(ptr->rchild == NULL)

        {

            for(int i = 0; i < space; i++)

                printf("  ");

        }

        else

        {

            space /= 2;

            for(int i = 0; i < space; i++)

                printf("--");

            for(int i = 0; i < space; i++)

                printf("  ");

        }

    }

}

void print\_tree(struct node\* root){

    int h = height(root);

    int space = pow(2,h);

    int level = 0;

    int lnc =0;

    struct node \*ptr = root, filler;

    filler.info = -1;

    filler.lchild = NULL;

    filler.rchild = NULL;

    if(root == NULL)

        return;

    insert\_queue(ptr);

    while(level <= h)

    {

        if(lnc == 0){

            printf("\n");

            lnc = pow(2,level);

            level ++;

            space /= 2;

        }

        ptr = del\_queue ();

        lnc--;

//printing node and arms

        give\_space(space, ptr, 1);

        if(ptr->info == -1)

            printf("  ");

        else

            printf ("%2d",ptr->info);

        give\_space(space,ptr, 2);

// queueing

        if (ptr->lchild != NULL)

            insert\_queue(ptr->lchild );

        else

            insert\_queue(&filler);

        if (ptr->rchild != NULL)

            insert\_queue(ptr->rchild );

        else

            insert\_queue(&filler);

    }

    front = -1;

    rear = -1;

}

int main()

{

    struct node\* root = NULL;

    int n, key, op;

    printf("Create Binary Search Tree : ");

    printf("\nEnter no. of elements : ");

    scanf("%d",&n);

    for(int i = 0; i <n; i++){

        printf("Enter item to insert : ");

        scanf("%d",&key);

        root = insert(root,key);

    }

    print\_tree(root);

    while(1){

        printf("\n1. Insert item\n2. Delete item\n3. Search item\n4. Inorder , Preorder, Postorder traversal and display tree\n5. Max and Min of BST\n6. exit\n");

        printf("Enter Option : "); scanf("%d",&op);

        switch(op){

            case 1:

                printf("\nEnter item : "); scanf("%d",&key);

                root = insert(root,key);

                break;

            case 2:

                printf("\nEnter item : "); scanf("%d",&key);

                root = del(root, key);

                break;

            case 3:

                printf("\nEnter item : "); scanf("%d",&key);

                root = search(root,key);

                break;

            case 4:

                printf("\nIn order : "); inorder(root); printf("\n");

                printf("pre order : "); preorder(root); printf("\n");

                printf("post order : "); postorder(root); printf("\n");

                print\_tree(root);

                break;

            case 5:

                printf("\nLargest item in the BST : %d\n",max\_item(root));

                printf("Smallest item in the BST : %d\n",min\_item(root));

                break;

            case 6:

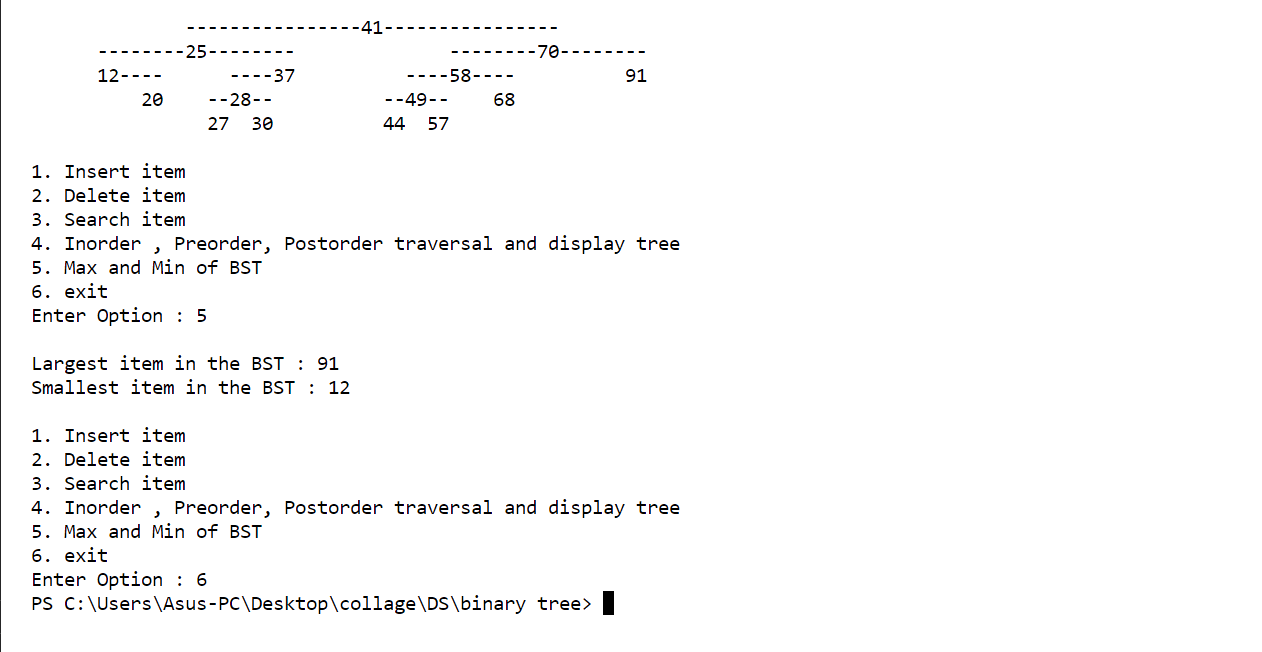
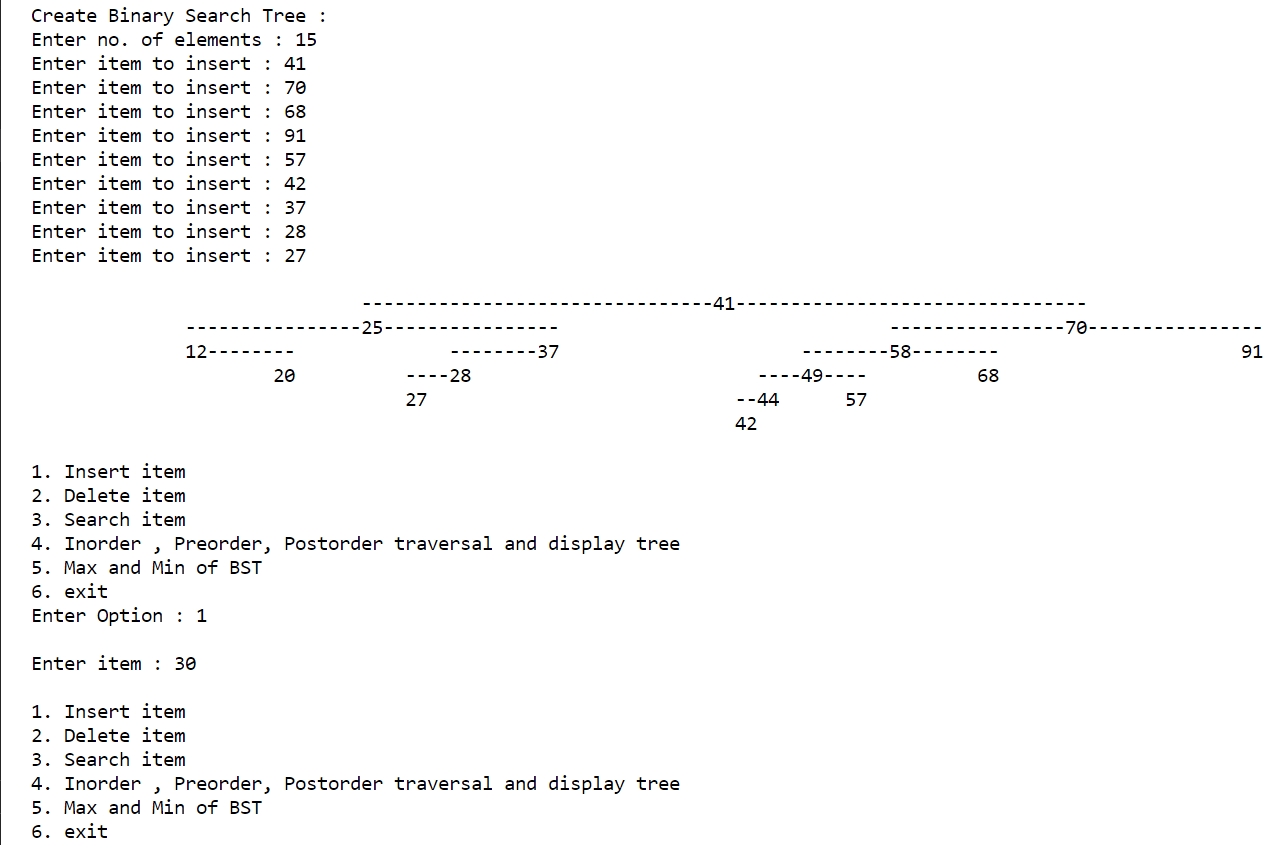
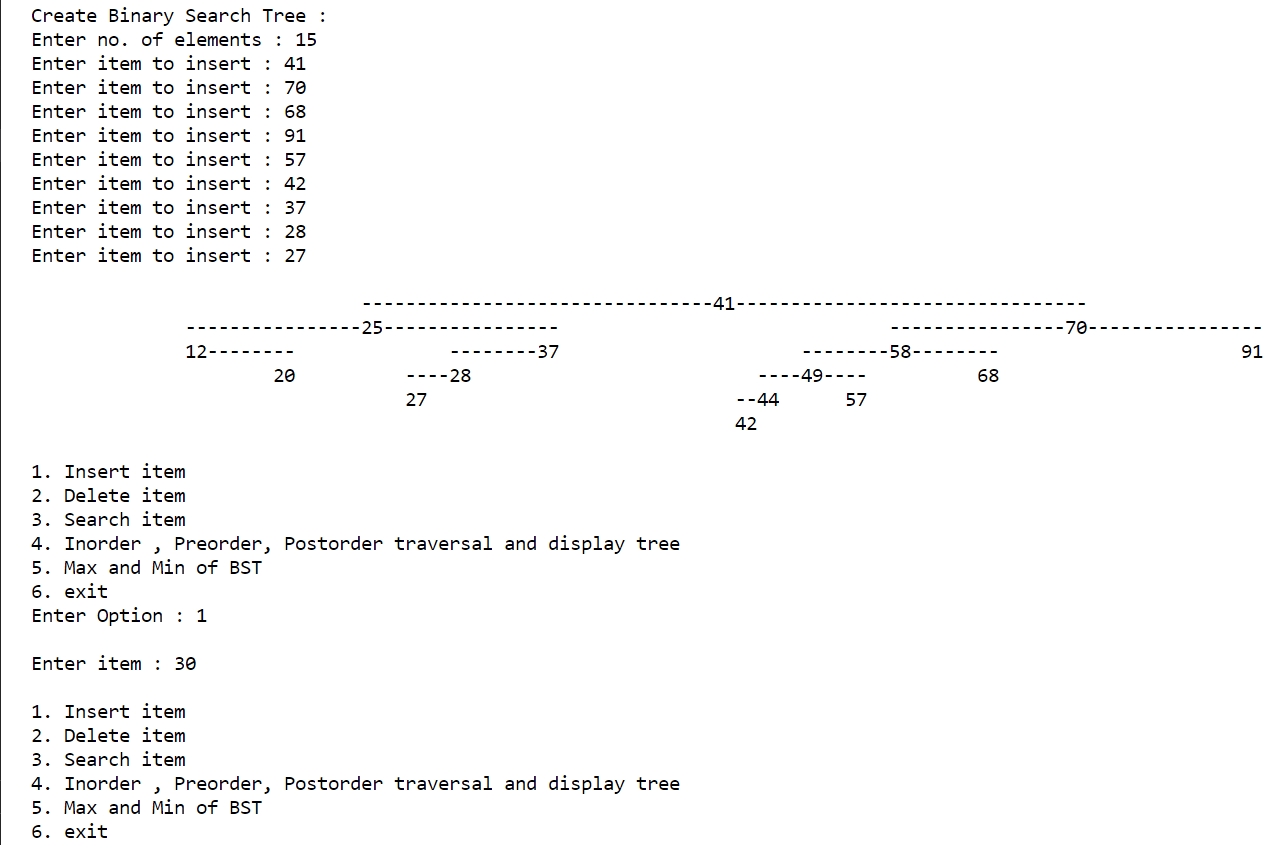
                return 0;

            default: printf("\nInvalid input !\n");

        }

    }

}



**CONCLUSION**: insertion , deletion , searching, traversals were used in binary search tree