**LAB SESSION 8: THREADED BINARY SEARCH TREES**

**AIM:** To implement Threaded Binary Search tree and perform the listed operations on such trees.

**PROBLEM DEFINITION:**

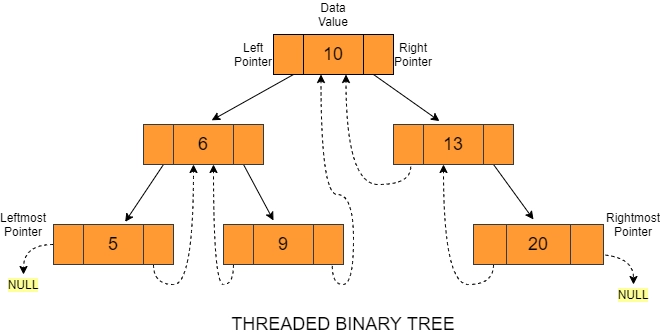
Develop a C program to create a Threaded Binary Search tree.

Provide options to the user to perform the following operations on the binary tree:

1. Insertion of a new element
2. Deletion of an existing element
3. Searching for a given element
4. Performing inorder and preorder traversal on the tree.

**THEORY:**

Threaded Binary Tree is a variant of a normal Binary Tree that facilitates faster tree traversal and does not require a Stack or Recursion. It decreases the memory wastage by setting the null pointers of a leaf node to the in-order predecessor or in-order successor. In a Threaded Binary Tree, the nodes will store the in-order predecessor/successor instead of storing NULL in the left/right child pointers. So the basic idea of a threaded binary tree is that for the nodes whose right pointer is null, we store the in-order successor of the node (if-exists), and for the nodes whose left pointer is null, we store the in-order predecessor of the node(if-exists).

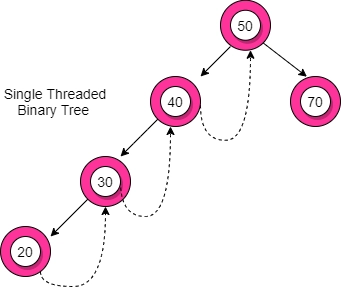


**Types of Threaded Binary tree**

There are two types of Threaded Binary Trees:

1. **Single-Threaded Binary Tree**

In this type, if a node has a right null pointer,



then this right pointer is threaded towards the

in-order successor’s node if it exists.

The structure of a node in a binary threaded tree

is quite similar to that of a binary tree, but with

some modifications. In threaded binary trees,

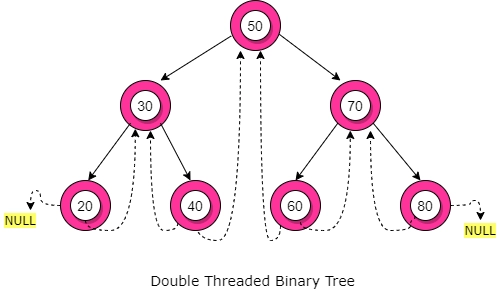
we need to use extra boolean variables in the

node structure. For single-threaded binary trees,

we use only the rightThread variable

1. **Double-Threaded Binary Tree**

In this type, the left null pointer of a node is



made to point towards the in-order predecessor

node and the right null pointer is made to point

towards the in-order successor node.

Here, the leftThread and rightThread Boolean

variables help us to differentiate whether the

left/right pointer stores the in-order

predecessor/successor or left child/right child

**Advantages of Threaded Binary Tree**

* No need for stacks or recursion: Unlike binary trees, threaded binary trees do not require a stack or recursion for their traversal.
* Optimal memory usage: Another advantage of threaded binary tree data structure is that it decreases memory wastage. In normal binary trees, whenever a node’s left/right pointer is NULL, memory is wasted. But with threaded binary trees, we are overcoming this problem by storing its inorder predecessor/successor.
* Time complexity: In-order traversal in a threaded binary tree is fast because we get the next node in O(1) time than a normal binary tree that takes O(Height). But insertion and deletion operations take more time for the threaded binary tree.
* Backward traversal: In a double-threaded binary tree, we can even do a backward traversal

**Disadvantages of Threaded Binary tree**

* Complicated insertion and deletion: By storing the inorder predecessor/ successor for the node with a null left/right pointer, we make the insertion and deletion of a node more time-consuming and a highly complex process.
* Extra memory usage: We use additional memory in the form of *rightThread* and *leftThread*  to distinguish between a thread from an ordinary link.

**ALGORITHMS**

1. Insert into a Threaded BST

* Initialize three pointers tmp, par, and ptr to NULL.
* Initialize a variable found to 0.
* Set ptr to the root of the binary search tree.
* While ptr is not NULL, do the following:
  + If the key to be inserted is equal to the current node’s key, set found to 1 and break out of the loop.
  + Set par to ptr.
  + If the key to be inserted is less than the current node’s key, move to the left child of the current node if it exists. Otherwise, break out of the loop.
  + If the key to be inserted is greater than the current node’s key, move to the right child of the current node if it exists. Otherwise, break out of the loop.
* If found is 1, print “Duplicate key”.
* Otherwise, create a new node tmp and set its key to the key to be inserted.
* Set tmp’s left and right threads to true.
* If par is NULL, set root to tmp, and set tmp’s left and right children to NULL.
* Otherwise, if the key to be inserted is less than par’s key, set tmp’s left child to par’s left child, set tmp’s right child to par, set par’s left thread to false, and set par’s left child to tmp.
* Otherwise, set tmp’s right child to par’s right child, set tmp’s left child to par, set par’s right thread to false, and set par’s right child to tmp.
* Return root.

2. Delete from a Threaded BST

* Initialize two pointers par and ptr to NULL.
* Initialize a variable found to 0.
* Set ptr to the root of the binary search tree.
* While ptr is not NULL, do the following:
  + If the key to be deleted is equal to the current node’s key, set found to 1 and break out of the loop.
  + Set par to ptr.
  + If the key to be deleted is less than the current node’s key, move to the left child of the current node if it exists. Otherwise, break out of the loop.
  + If the key to be deleted is greater than the current node’s key, move to the right child of the current node if it exists. Otherwise, break out of the loop.
* If found is 0, print “dkey not present in the tree”.
* Otherwise, if ptr has no children, call case\_c to delete the node.
* Otherwise, if ptr has only one child, call case\_b to delete the node.
* Otherwise, call case\_a to delete the node.
* Return root.

**PROGRAM AND OUTPUT:**

#include<stdio.h>

#include<stdlib.h>

#include<stdbool.h>

struct node{

    struct node\* left;

    bool lthread;

    int info;

    bool rthread;

    struct node\* right;

};

struct node \*in\_succ (struct node \*ptr)

{

    if (ptr->rthread == true)

        return ptr->right ;

    else

    {

        ptr=ptr->right;

        while(ptr->lthread==false)

            ptr=ptr->left;

        return ptr;

    }

}

struct node \*in\_pred (struct node \*ptr)

{

    if (ptr->lthread == true)

        return ptr->left;

    else

    {

        ptr=ptr->left;

        while(ptr->rthread==false)

            ptr=ptr->right;

        return ptr;

    }

}

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

    struct node \*tmp, \*par, \*ptr;

    int found = 0;

    ptr = root;

    par = NULL;

    while(ptr != NULL){

        if(ikey == ptr->info){

            found = 1;

            break;

        }

        par = ptr;

        if(ikey < ptr->info){

            if(ptr->lthread == false)

                ptr = ptr->left;

            else

                break;

        }

        else {

            if(ptr->rthread == false){

                ptr = ptr->right;

            }

            else

                break;

        }

    }

    if(found)

        printf("Duplicate key\n");

    else{

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

        tmp->info = ikey;

        tmp->lthread = true;

        tmp->rthread = true;

        if(par == NULL){

            root = tmp;

            tmp->left = NULL;

            tmp->right = NULL;

        }

        else if(ikey < par->info){

            tmp->left = par->left;

            tmp->right = par;

            par->lthread = false;

            par->left = tmp;

        }

        else{

            tmp->right = par->right;

            tmp->left = par;

            par->rthread = false;

            par->right = tmp;

        }

    }

    return root;

}

void search(struct node\* root, int ikey){

    struct node \*par, \*ptr;

    int found = 0;

    ptr = root;

    par = NULL;

    while(ptr != NULL){

        if(ikey == ptr->info){

            found = 1;

            break;

        }

        par = ptr;

        if(ikey < ptr->info){

            if(ptr->lthread == false)

                ptr = ptr->left;

            else

                break;

        }

        else {

            if(ptr->rthread == false){

                ptr = ptr->right;

            }

            else

                break;

        }

    }

    if(found)

        printf("\n Found! \n");

    else

        printf("\nNot Found! \n");

}

struct node \* case\_a(struct node \*root, struct node \*par, struct node \*ptr)

{

    if (ptr == NULL)

        root = NULL;

    else if (par->left == ptr)

    {

        par->lthread = true;

        par->left = ptr->left ;

    }

    else

    {

        par->rthread = true;

        par->right = ptr->right ;

    }

    free(ptr);

    return root;

}

struct node \* case\_b(struct node \*root, struct node \*par, struct node \*ptr)

{

    struct node \*child, \*p, \*s ;

    if (ptr->lthread == false)

        child = ptr->left ;

    else

        child = ptr->right ;

    if (par == NULL)

        root = child;

    else if (par->left == ptr)

        par->left = child;

    else

        par->right = child;

    s=in\_succ(ptr);

    p=in\_pred(ptr);

    if( ptr->lthread==false)

        p->right=s;

    else if( ptr->rthread==false)

        s->left=p;

    free(ptr);

    return root;

}

struct node \* case\_c(struct node \*root, struct node \*par, struct node \*ptr)

{

    struct node \*succ, \*parsucc;

    parsucc = ptr;

    succ = ptr->right ;

    while (succ->left != NULL )

    {

        parsucc = succ;

        succ = succ->left ;

    }

    ptr->info = succ->info ;

    if (succ->lthread == true && succ->rthread == true )

        root = case\_a (root,parsucc,succ);

    else

        root = case\_b (root,parsucc,succ);

    return root;

}

struct node \* del (struct node \*root, int dkey)

{

    struct node \* par, \*ptr;

    int found=0;

    ptr = root;

    par = NULL;

    while (ptr != NULL)

    {

        if (dkey == ptr->info)

        {

            found=1;

            break;

        }

        par = ptr;

        if (dkey < ptr->info)

        {

            if (ptr->lthread==false)

                ptr = ptr->left;

            else

                break;

        }

        else

        {

            if (ptr->rthread==false)

                ptr = ptr->right;

            else

                break;

        }

    }

    if (found == 0 )

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

    else if (ptr->lthread == false && ptr->rthread == false )

        root = case\_c(root, par,ptr);

    else if (ptr->lthread == false )

        root = case\_b(root, par,ptr);

    else if (ptr->rthread == false)

        root = case\_b(root, par,ptr);

    else

        root = case\_a(root, par,ptr);

    return root;

}

void inorder(struct node \*root)

{

    struct node \*ptr;

    if (root== NULL)

    {

        printf("Tree is empty");

        return;

    }

    ptr=root;

    while(ptr->lthread==false)

        ptr=ptr->left;

    while(ptr!= NULL)

    {

        printf("%d ", ptr->info);

        ptr=in\_succ(ptr);

    }

}

void perorder(struct node \*root)

{

    struct node \*ptr;

    if (root== NULL)

    {

        printf("Tree is empty");

        return;

    }

    ptr=root;

    while(ptr!= NULL)

    {

        printf("%d ", ptr->info);

        if(ptr->lthread==false)

            ptr=ptr->left;

        else if (ptr->rthread==false)

            ptr=ptr->right;

        else

        {

            while(ptr!= NULL && ptr->rthread==true)

                ptr=ptr->right;

            if (ptr!= NULL)

                ptr=ptr->right;

        }

    }

}

int main(){

    int op, n, key;

    struct node \*root = NULL;

    printf("create list : \n");

    printf("Enter number of elements : ");

    scanf("%d",&n);

    for(int i = 0; i < n; i++){

        printf("Enter item : ");

        scanf("%d",&key);

        root = insert(root, key);

    }

    while(true){

        printf("\n\n\*\*\*Menu\*\*\*");

        printf("\n1. Insert\n2. Delete\n3. Search\n4. Inorder an Perorder traversal\n5. Exit\n");

        printf("Enter option : ");

        scanf("%d",&op);

        switch(op){

        case 1: printf("Enter key: ");scanf("%d",&key);

                root = insert(root,key);

                break;

        case 2: printf("Enter key: ");scanf("%d",&key);

                root = del(root,key);

                break;

        case 3: printf("Enter key: ");scanf("%d",&key);

                search(root,key);

                break;

        case 4:

                printf("Inorder : "); inorder(root);

                printf("\nPerorder : "); perorder(root);

                break;

        case 5: return 0;

        default:

            printf("Input Error! \n");

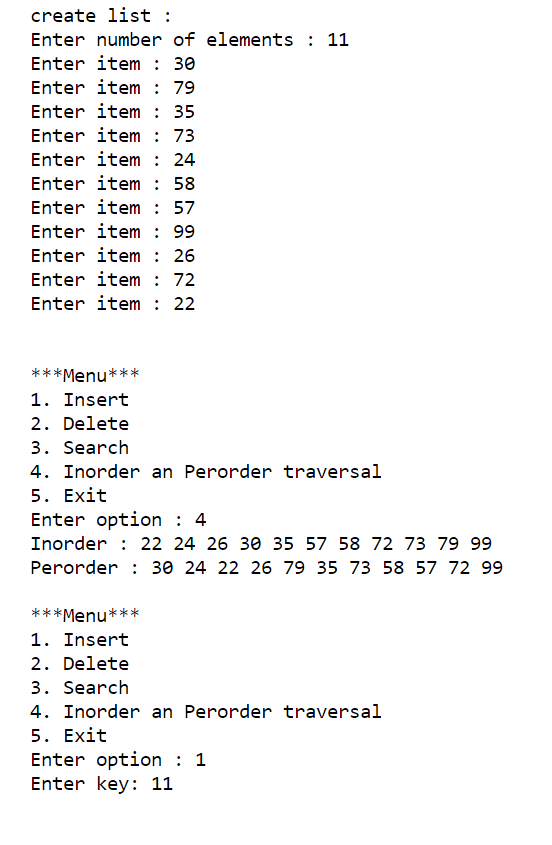
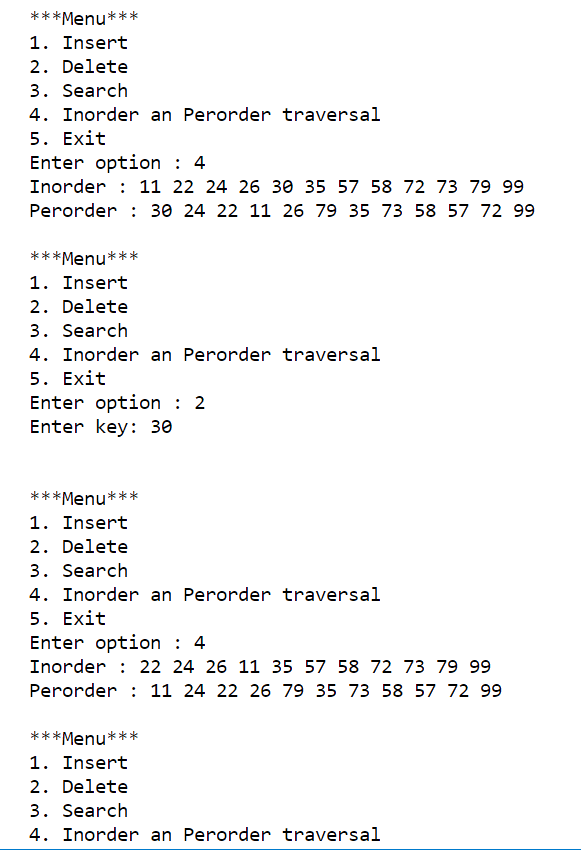
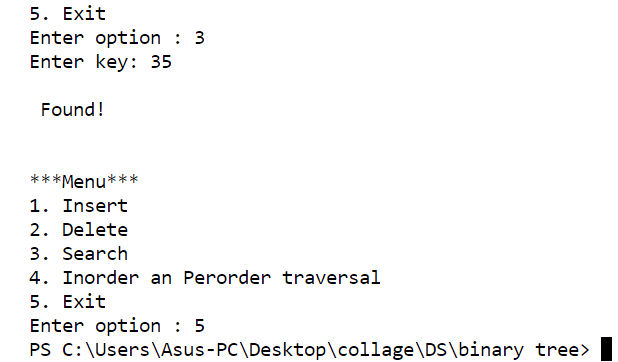
        }

    }

    return 0;

}

// 30 79 35 73 24 58 57 99 26 72 22

**** **** 

**CONCLUSION: concepts regarding threaded binary trees was understood and implimented**