**LAB SESSION 8: THREADED BINARY SEARCH TREES DATE:16.08.2023**

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

**PROBLEM DEFINITION:**

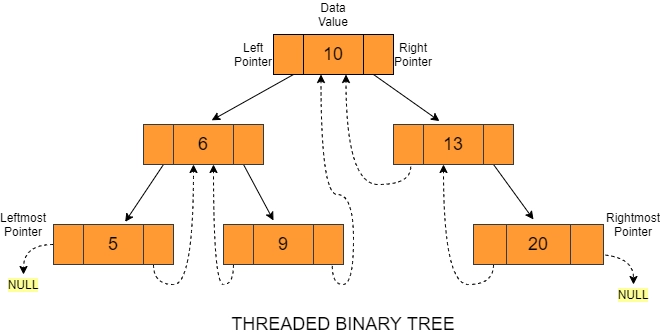
Develop a C program to create a Threaded Binary Search tree.

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).

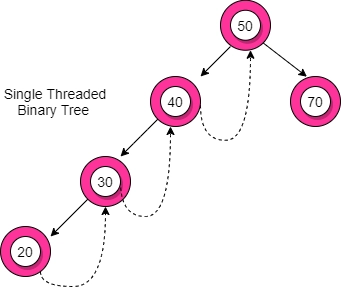


**Types of Threaded Binary tree**

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,

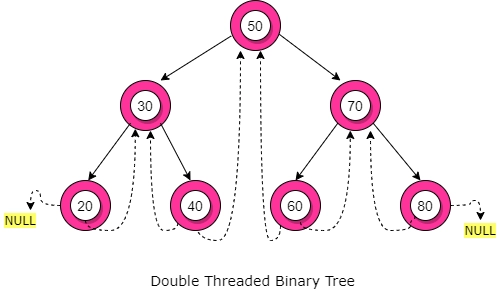
we need to use extra boolean variables in the

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
2. Delete from a Threaded BST

#include<stdio.h>

#include<stdlib.h>

typedef enum {false, true} bool;

struct node{

    struct node \*lchild;

    bool lthread;

    int info;

    struct node \*rchild;

    bool rthread;

};

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

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

struct node \* case\_a(struct node \*, struct node \*, struct node \*);

struct node \* case\_b(struct node \*, struct node \*, struct node \*);

struct node \* case\_c(struct node \*, struct node \*, struct node \*);

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

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

void inorder(struct node \*);

void preorder(struct node \*);

void search (struct node \*, int);

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

    struct node \*temp, \*par = NULL, \*ptr = root;

    int found = 0;

    while(ptr != NULL){

        if(ikey == ptr->info){

            found = 1;

            break;

        }

        par = ptr;

        if(ikey < ptr->info){

            if(ptr->lthread == false)

                ptr = ptr->lchild;

            else break;

        }

        else{

            if(ptr->rthread == false)

                ptr = ptr->rchild;

            else break;

        }

    }

    if(found)

        printf("Duplicate\n");

    else{

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

        temp->info = ikey;

        temp->lthread = true;

        temp->rthread = true;

        if(par == NULL){

            root = temp;

            temp->lchild = NULL;

            temp->rchild = NULL;

        }

        else if(ikey > par->info){

            temp->rchild = par->rchild;

            temp->lchild = par;

            par->rchild = temp;

            par->rthread = false;

        }

        else{

            temp->lchild = par->lchild;

            temp->rchild = par;

            par->lchild = temp;

            par->lthread = false;

        }

    }

    return root;

}

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

    struct node \*par = NULL, \*ptr = root;

    int found = 0;

    while(ptr!= NULL){

        if(dkey == ptr->info){

            found = 1;

            break;

        }

        par = ptr;

        if(dkey < ptr->info){

            if(ptr->lthread == false)

                ptr = ptr->lchild;

            else break;

        }

        else{

            if(ptr->rthread == false)

                ptr = ptr->rchild;

            else break;

        }

    }

    if(found == 0)

        printf("Element not found\n");

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

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

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

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

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

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

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

    return root;

}

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

    if(par == NULL)

        root = NULL;

    else if(ptr == par->lchild){

        par->lthread = true;

        par->lchild = ptr->lchild;

    }

    else{

        par->rthread = true;

        par->rchild = ptr->rchild;

    }

    free(ptr);

    printf("Deleted the element\n");

    return root;

}

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

    struct node \*child, \*s, \*p;

    if(ptr->lthread == false)

        child = ptr->lchild;

    else child = ptr->rchild;

    if(par == NULL)

        root = child;

    else if(ptr == par->lchild)

        par->lchild = child;

    else par->rchild = child;

    s = in\_succ(ptr);

    p = in\_pred(ptr);

    if(ptr->lthread == false)

        p->rchild = s;

    else{

        if(ptr->rthread == false)

            s->lchild = p;

    }

    free(ptr);

    printf("Deleted the element\n");

    return root;

}

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

    struct node \*succ = ptr->rchild, \*parsucc = ptr;

    while(succ->lchild != NULL){

        parsucc = succ;

        succ = succ->lchild;

    }

    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 \* in\_succ(struct node \* ptr){

    if(ptr->rthread == true)

        return ptr->rchild;

    else{

        ptr = ptr->rchild;

        while(ptr->lthread == false)

            ptr = ptr->lchild;

        return ptr;

    }

}

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

    if(ptr->lthread == true)

        return ptr->lchild;

    else{

        ptr = ptr->lchild;

        while(ptr->rthread == false)

            ptr = ptr->rchild;

        return ptr;

    }

}

void inorder(struct node \* root){

    struct node \*ptr = root;

    if(ptr == NULL){

        printf("Empty\n");

        return;

    }

    while(ptr->lthread == false)

        ptr = ptr->lchild;

    while(ptr!= NULL){

        printf(" %d", ptr->info);

        ptr = in\_succ(ptr);

    }

}

void preorder(struct node \* root){

    struct node \* ptr = root;

    if(root == NULL){

        printf("Empty\n");

        return;

    }

    while(ptr != NULL){

        printf(" %d", ptr->info);

        if(ptr->lthread == false)

            ptr = ptr->lchild;

        else if (ptr->rthread == false)

            ptr = ptr->rchild;

        else{

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

                ptr = ptr->rchild;

            if(ptr != NULL)

                ptr = ptr->rchild;

        }

    }

}

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

    while(ptr->lthread == false || ptr->rthread == false){

        if(ptr->info == skey){

            printf("Element found\n");

            return;

        }

        else if(ptr->info > skey && ptr->lthread == false)

            ptr = ptr->lchild;

        else if (ptr->info < skey && ptr->rthread == false)

            ptr = ptr->rchild;

    }

    printf("Element not found\n");

}

int main(){

    struct node \*root = NULL;

    int s, key, n;

    char c;

    printf("Do you want to insert many elements? (y/n): ");

    scanf("%c", &c);getchar();

    if(c == 'y'){

        printf("Enter number of elements to insert: ");

        scanf("%d", &n);

        for(int i = 0; i < n; i++){

            printf("Insert: ");

            scanf("%d", &key);

            root = insert(root, key);

        }

    }

    do{

        printf("\n1. Insertion of a new element\n"

                "2. Deletion of an existing element\n"

                "3. Searching for a given element\n"

                "4. Perform inorder and preorder traversals\n"

                "5. Exit\n");

        printf("Enter your option: ");

        scanf("%d", &s);

        switch(s){

            case 1: printf("Insert: ");

                    scanf("%d", &key);

                    root = insert(root, key);

                    break;

            case 2: printf("Delete: ");

                    scanf("%d", &key);

                    root = delete(root, key);

                    break;

            case 3: printf("Enter the element: ");

                    scanf("%d", &key);

                    search(root, key);

                    break;

            case 4: printf("Inorder: "); inorder(root);

                    printf("\nPreorder: "); preorder(root);

                    break;

            case 5: break;

            default: printf("Invalid input\n");

        }

    }while(s != 5);

    return 0;

}

