**THREADED BINARY TREES**

**EXPT NO: 6**  **DATE: 2/12/21**

**AIM**

1)Write a program to perform following operations in threaded binary search tree.   
  a. Insert a node  
  b. Delete a node  
  c. Display Tree

2) Write a program to implement following traversals in threaded binary search tree.   
  a. In-order

  b. Pre-order

**THEORY**

In a normal binary search tree, the traversals are done with recursive and non-recursive approach. But both this method requires a stack. Recursive version uses a run-time stack and non-recursive version uses a user defined stack. Traversal is a common operation and if the tree has to be traversed frequently then using a stack isn’t efficient since extra time and space is needed in maintaining the stack.

A binary tree with nodes has 2n pointers out of which n+1 are always NULL. So we can see that about half the space allocated for pointers is wasted. We can utilize this waste space to contain some useful information a left NULL pointer can be used to store the address of Inorder predecessor of the node and a right NULL pointer can be used to store the address of the Inorder successor of the node. There pointers are called as **threads** and a binary tree which is implemented using threads is called a **Threaded binary tree**.

**Thread Binary Tree Implementation**

While implementing a threaded binary tree we should be able to distinguish real children pointer from threads. For this we can attach two Boolean variables to each node, and these variables will be used to determine whether the left and right pointers of that node are thread or child pointers.

struct node {

struct node \*left;

struct node \*right;

int info;

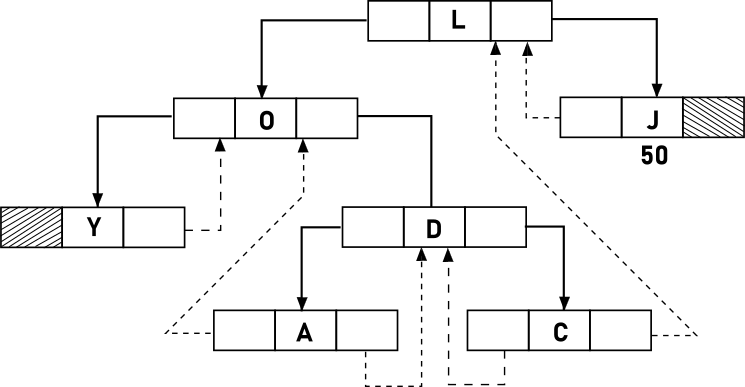
boolean Rthread;

boolean Lthread;

};

Here we have two boolean members Lthread and Rthread to differentiate between a thread and child pointer. If Lthread is true then left pointer isa thread of Inorder predecessor otherwise it contains address of left child. Similarly, if Rthread is tur the right pointer is a thread to Inorder successor otherwise it contains address of right child. The first node in Inorder traversal has no predecessor and the last node has no successor.

**Example (A):**



**Inorder Successor of a node in in-threaded tree**

If the right pointer of the node is a thread, then there is no need to do anything because the right thread points to the Inorder successor. If node’s right pointer is not a thread i.e., node has right child then to find the Inorder successor, we move to this right child and keep on moving left till we find a no left child.

**Inorder Predecessor of a node in in-threaded tree**

lf the left pointer is a thread; then thus thread will point to the Inorder predecessor. If the left pointer is not a thread i.e., node has a left child then to find the Inorder predecessor, we move to this left child and keep on moving right till we find a node with right child.

**Traversals**

* Inorder Traversal
* Preorder Traversal

**Inorder Traversal**

The first node to be visited in the Inorder traversal is the last node in the leftmost branch starting from the root. So, we start from the root and move left till we get a node with no left child, and this node is visited. Now with the help of in\_succ() function we find the Inorder successor of each node and visit it. The right pointer of the last node is NULL and we have marked it as thread. So, we stop our process when we get a node whose right thread is NULL.

**Preorder traversal**

In preorder traversal first the root is visited, so we start from the root node. If the node has a left child, then that left child will be visited, otherwise if the node has a right child, then that right child will be visited. If the node has neither left or right child (Leaf node) then with the help of right thread, we will reach that Inorder successor of the node which has a right subtree

**Insertion**

**Case 1:** Insertion in empty tree

**Case 2:** when new node inserted as the left child

**Case 3:** when new node is inserted as right child

**Deletion**

**Case 1:** Leaf node o be deleted.

**Case 2:** Node to be deleted has only one child.

**Case 3:** Node to be deleted has two children.

**ALGORITHMS**

**1)**

**struct node\*insert(struct node\*p,int data)**

1. Declare struct node\*par=NULL, \*ptr=p,\*t

2. while(ptr!=NULL)

1. par=ptr

2. if(ptr->data>data)

1. if(ptr->lthread==false)

1. ptr=ptr->left

2. else

1. break

3. else if ptr->data < data

1. if ptr->Rthread==false

1. ptr=ptr->right

2. else

1. break

4. else

1. Output duplicate data

2. return p

3. t=(struct node\*)malloc(sizeof(struct node))

4. t->data=data

5. t->lthread=true

6. t->rthread=true

7. if(par==NULL)

1. t->left=NULL

2. t->right=NULL

3. p=t

8. else if(data<par->data)

1. t->left=par->left

2. t->right=par

3. par->left=t

4. par->rthread=false

9. else

1. t->right=par->right

2. t->left=par

3. par->right =t

4. par->rthread=false

10. return p

**struct node\*insucc(struct node\*p)**

1. if(p->rthread==true)

1. return p->right

2. else

1. p=p->right

2. while(p->lthread==false)

1. p=p->left

3. return p

**struct node\*del(struct node\*p,int data)**

1. Declare int found=0

2. struct node\*par=NULL,\*ptr=p

3. while(ptr!=NULL)

1. if(data==ptr->data)

1.found=1

2. break

2. par=ptr

3. if(ptr->data>data)

1. if ptr->Lthread==false

1. ptr=ptr->left

2. else

1. break

4. else

1.if(ptr->rthread==false)

1. ptr=ptr->right

2. else

1. break

4. if(found)

1. Output data not found

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

1. p=case\_c(p,par,ptr)

6. else if(ptr->lthread==false||ptr->rthread==false)

1. p=case\_b(p,par,ptr)

7. else

1. p=case\_a(p,par,ptr)

8. return p

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

1. if(par==NULL)

1. p=NULL

2. else if(par->left==ptr)

1. par->left=ptr->left

2. par->lthread=true

3. else

1. par->right=ptr->right

2. par->rthread=true

4. free(ptr)

5. return p

**struct node\*inpre(struct node\*p)**

1. if(p->lthread==true)

1. return p->left

2. else

1. p=p->left

2. while(p->rthread==false)

1. p=p->right

3. return p

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

1. Declare struct node\*parsucc=ptr,\*succ=ptr->right

2. while(succ->lthread==false)

1. parsucc=succ

2. succ=succ->left

3. ptr->data=succ->data

4. par=parsucc

5. ptr=ptr->succ

6. if(ptr->lthread==true&&ptr->rthread==true)

1. p=case\_a(p,par,ptr)

7. else

1. p=case\_b(p,par,ptr)

8. return p

**preorder(struct node \*p)**

1. Declare struct node\*ptr=p

2. while(ptr!=NULL)

1. Output ptr->data

2. if(ptr->lthread==false)

1. ptr=ptr->left

3. else if(ptr->rthread==false)

1. ptr=ptr->right

4. else

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

1. ptr=ptr->right

2. if(ptr!=NULL)

1. ptr=ptr->right

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

1. Declare struct node\*child,\*pre,\*succ

2. if(ptr->lthread==false)

1. child=ptr->left

3. else

1. child=ptr->right

4. if(par==NULL)

1. p=child

5. else if(par->left==ptr)

1. par->left=child

6. else

1. par->right=child

7. succ=insucc(ptr)

8. pre=inpre(ptr)

9. if(ptr->lthread==false)

1. pre->right=succ

10. else

1. if(ptr->rthread==false)

1. succ->left=pre

11. free(ptr)

12. return p

**void inorder(struct node\*p)**

1. Declare struct node\*ptr=p

2. while(ptr->lthread==false

1. ptr=ptr->left

3. while(ptr!=NULL)

1. Output ptr->data

2. ptr=insucc(ptr)

**CODES**

**1)**

#include<stdio.h>

#include<stdlib.h>

#include<stdbool.h>

#define CNT 10

struct node

{

int data;

bool Lthread,Rthread;

struct node \*left;

struct node \*right;

};

struct node \*del(struct node \*root,int ikey);

struct node \*caseA(struct node \*root, struct node \*par, struct node \*ptr);

struct node \*caseB(struct node \*root, struct node \*par, struct node \*ptr);

struct node \*caseC(struct node \*root, struct node \*par, struct node \*ptr);

struct node \*inSucc(struct node \*ptr);

struct node \*inPred(struct node \*ptr);

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

{

struct node \*temp,\*par,\*ptr;

int found=0;

ptr=root;

par=NULL;

while(ptr!=NULL)

{

if(ikey==ptr->data)

{

found=1;

break;

}

par=ptr;

if(ikey<ptr->data)

{

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

{

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

temp->data=ikey;

temp->Lthread=true;

temp->Rthread=true;

if(par==NULL)

{

root=temp;

temp->left=NULL;

temp->right=NULL;

}

else if(ikey<par->data)

{

temp->left=par->left;

temp->right=par;

par->Lthread=false;

par->left=temp;

}

else

{

temp->left=par;

temp->right=par->right;

par->Rthread=false;

par->right=temp;

}

}

return root;

}

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

{

struct node \*par,\*ptr;

int found=0;

ptr=root;

par=NULL;

while(ptr!=NULL)

{

if(ikey==ptr->data)

{

found=1;

break;

}

par=ptr;

if(ikey<ptr->data)

{

if(ptr->Lthread==false)

ptr=ptr->left;

else

break;

}

else

{

if(ptr->Rthread==false)

ptr=ptr->right;

else

break;

}

}

if(!found)

printf("dKEY NOT PRESENT IN TREE\n");

else if(ptr->Lthread==false&&ptr->Rthread==false)

root=caseC(root,par,ptr);

else if(ptr->Lthread==false)

root=caseB(root,par,ptr);

else if(ptr->Rthread==false)

root=caseB(root,par,ptr);

else

root=caseA(root,par,ptr);

return root;

}

void display(struct node \*root,int space)

{

int i;

space+=CNT;

if(root->Rthread==false)

display(root->right,space);

printf("\n");

for(i=CNT;i<space;i++)

printf(" ");

printf("%d\n",root->data);

if(root->Lthread==false)

display(root->left,space);

}

int main()

{

int choice,num,space=0;

struct node \*root=NULL;

while(1)

{

printf("\n");

printf("1: INSERT\n");

printf("2: DELETE\n");

printf("3: DISPLAY\n");

printf("ENTER THE CHOICE\n");

scanf("%d",&choice);

switch(choice)

{

case 1:

printf("ENTER THE NUMBER TO INSERTED\n");

scanf("%d",&num);

root=insert(root,num);

break;

case 2:

printf("ENTER THE NUMBER TO BE DELETED\n");

scanf("%d",&num);

root=del(root,num);

break;

case 3:

display(root,0);

}

}

}

struct node \*caseA(struct node \*root, struct node \*par, struct node \*ptr)

{

if(par==NULL)

root=NULL;

else if(ptr==par->left)

{

par->Lthread=true;

par->left=ptr->left;

}

else

{

par->Rthread=true;

par->right=ptr->right;

}

free(ptr);

return root;

}

struct node \*caseB(struct node \*root, struct node \*par, struct node \*ptr)

{

struct node \*child,\*s,\*p;

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=inSucc(ptr);

p=inPred(ptr);

if(ptr->Lthread==false)

p->right=s;

else

{

if(ptr->Rthread==false)

s->left=p;

}

free(ptr);

return root;

}

struct node \*caseC(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->data=succ->data;

if(succ->Lthread==true&&succ->Rthread==true)

root=caseA(root,parsucc,succ);

else

root=caseB(root,parsucc,succ);

return root;

}

struct node \*inSucc(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 \*inPred(struct node \*ptr)

{

if(ptr->Lthread==true)

{

return ptr->left;

}

else

{

ptr=ptr->left;

while(ptr->Rthread==false)

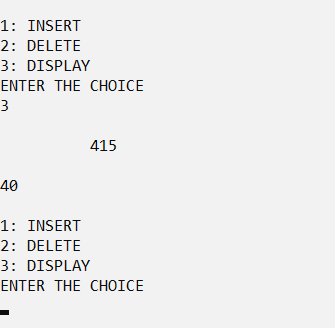
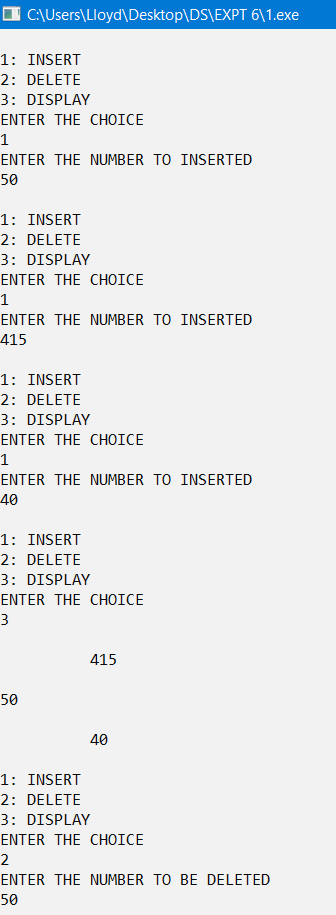
ptr=ptr->right;

return ptr;

}

}

**OUTPUT**



**2)**

#include<stdio.h>

#include<stdlib.h>

#include<stdbool.h>

#define CNT 10

struct node

{

int data;

bool Lthread,Rthread;

struct node \*left;

struct node \*right;

};

struct node \*del(struct node \*root,int ikey);

struct node \*caseA(struct node \*root, struct node \*par, struct node \*ptr);

struct node \*caseB(struct node \*root, struct node \*par, struct node \*ptr);

struct node \*caseC(struct node \*root, struct node \*par, struct node \*ptr);

struct node \*inSucc(struct node \*ptr);

struct node \*inPred(struct node \*ptr);

void inorder(struct node \*p);

void preorder(struct node \*p);

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

{

struct node \*temp,\*par,\*ptr;

int found=0;

ptr=root;

par=NULL;

while(ptr!=NULL)

{

if(ikey==ptr->data)

{

found=1;

break;

}

par=ptr;

if(ikey<ptr->data)

{

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

{

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

temp->data=ikey;

temp->Lthread=true;

temp->Rthread=true;

if(par==NULL)

{

root=temp;

temp->left=NULL;

temp->right=NULL;

}

else if(ikey<par->data)

{

temp->left=par->left;

temp->right=par;

par->Lthread=false;

par->left=temp;

}

else

{

temp->left=par;

temp->right=par->right;

par->Rthread=false;

par->right=temp;

}

}

return root;

}

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

{

struct node \*par,\*ptr;

int found=0;

ptr=root;

par=NULL;

while(ptr!=NULL)

{

if(ikey==ptr->data)

{

found=1;

break;

}

par=ptr;

if(ikey<ptr->data)

{

if(ptr->Lthread==false)

ptr=ptr->left;

else

break;

}

else

{

if(ptr->Rthread==false)

ptr=ptr->right;

else

break;

}

}

if(!found)

printf("dKEY NOT PRESENT IN TREE\n");

else if(ptr->Lthread==false&&ptr->Rthread==false)

root=caseC(root,par,ptr);

else if(ptr->Lthread==false)

root=caseB(root,par,ptr);

else if(ptr->Rthread==false)

root=caseB(root,par,ptr);

else

root=caseA(root,par,ptr);

return root;

}

void display(struct node \*root,int space)

{

int i;

space+=CNT;

if(root->Rthread==false)

display(root->right,space);

printf("\n");

for(i=CNT;i<space;i++)

printf(" ");

printf("%d\n",root->data);

if(root->Lthread==false)

display(root->left,space);

}

int main()

{

int choice,num,space=0;

struct node \*root=NULL;

while(1)

{

printf("\n");

printf("1: INSERT\n");

printf("2: DELETE\n");

printf("3: DISPLAY\n");

printf("4: INORDER\n");

printf("5: PREORDER\n");

printf("ENTER THE CHOICE\n");

scanf("%d",&choice);

switch(choice)

{

case 1:

printf("ENTER THE NUMBER TO INSERTED\n");

scanf("%d",&num);

root=insert(root,num);

break;

case 2:

printf("ENTER THE NUMBER TO BE DELETED\n");

scanf("%d",&num);

root=del(root,num);

break;

case 3:

display(root,0);

break;

case 4:

printf("INORDER: ");

inorder(root);

break;

case 5:

printf("PREORDER: ");

preorder(root);

break;

}

}

}

struct node \*caseA(struct node \*root, struct node \*par, struct node \*ptr)

{

if(par==NULL)

root=NULL;

else if(ptr==par->left)

{

par->Lthread=true;

par->left=ptr->left;

}

else

{

par->Rthread=true;

par->right=ptr->right;

}

free(ptr);

return root;

}

struct node \*caseB(struct node \*root, struct node \*par, struct node \*ptr)

{

struct node \*child,\*s,\*p;

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=inSucc(ptr);

p=inPred(ptr);

if(ptr->Lthread==false)

p->right=s;

else

{

if(ptr->Rthread==false)

s->left=p;

}

free(ptr);

return root;

}

struct node \*caseC(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->data=succ->data;

if(succ->Lthread==true&&succ->Rthread==true)

root=caseA(root,parsucc,succ);

else

root=caseB(root,parsucc,succ);

return root;

}

struct node \*inSucc(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 \*inPred(struct node \*ptr)

{

if(ptr->Lthread==true)

{

return ptr->left;

}

else

{

ptr=ptr->left;

while(ptr->Rthread==false)

ptr=ptr->right;

return ptr;

}

}

void inorder(struct node \*p)

{

struct node \*ptr=p;

while(ptr->Lthread==false)

ptr=ptr->left;

while(ptr!=NULL)

{

printf("%d ",ptr->data);

ptr=inSucc(ptr);

}

}

void preorder(struct node \*p)

{

struct node \*ptr=p;

while(ptr!=NULL)

{

printf("%d ",ptr->data);

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;

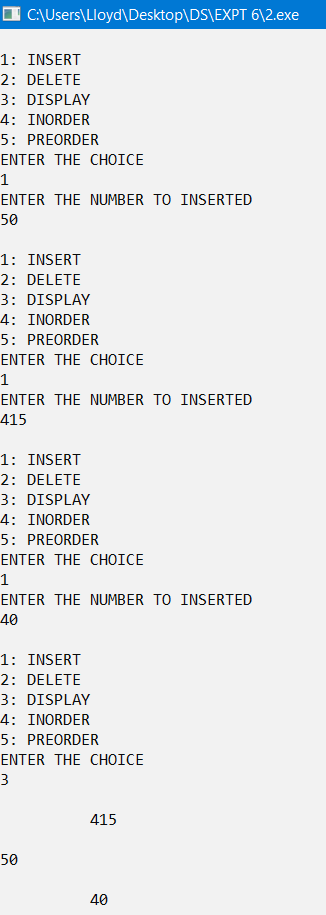
if(ptr!=NULL)

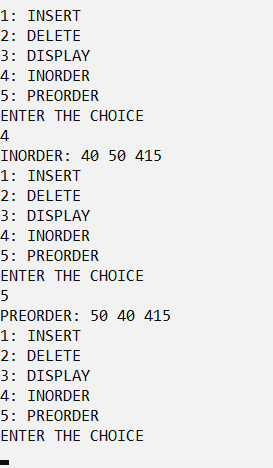
ptr=ptr->right;

}

}

}

**OUTPUT**

****

**CONCLUSION**

The given problem statements were successfully compiled and executed.

**LEARNINGS AND FINDINGS**

This experiment manifests

1. Concept of Threaded Binary Trees
2. Traversals involved in Threaded binary trees

Threaded binary trees eliminates the need of using stacks for traversals thus makes it an efficient implementation of binary trees. However, it is a complex design to implement and it does not solve the problem of unbalanced growth or shrinking of trees.

|  |  |
| --- | --- |
| **SR NO.** | **COMPILATION TIME** |
| 1 | 0.59 s |
| 2 | 0.20 s |