***// DOUBLE THREADED BINARY TREE***

#include <stdio.h>

#include <stdlib.h>

typedef struct node

{

int data;

struct node \*left;

struct node \*right;

int rightThread;

int leftThread;

} Node;

typedef struct tree

{

Node \*root;

} Tree;

void tree\_initialize(Tree \*tree);

void tree\_insert(Tree \*tree, int data);

*// Finds the inorder successor and predecessor of the given node*

Node \*inorder\_successor(Node \*ptr);

Node \*inorder\_predecessor(Node \*ptr);

*/\* Prints all the elements of the tree in ascending and descending order respectively \*/*

void print\_ascending(Tree \*tree);

void print\_descending(Tree \*tree);

void tree\_destroy(Tree \*tree);

*/\* Advisory: While some guarded discernment could perhaps lead you to more efficient implementations, you are requested to provide suitable recursive solutions for the following function declarations. \*/*

int main()

{

int choice, loop = 1;

Tree my\_tree;

tree\_initialize(&my\_tree);

while (loop)

{

scanf("%d", &choice);

switch (choice)

{

int element;

int data;

case 1:

*/\* Insert elements \*/*

scanf("%d", &element);

tree\_insert(&my\_tree, element);

break;

case 2:

*/\* Print elements in the ascending order \*/*

print\_ascending(&my\_tree);

break;

case 3:

*/\* Print elements in the descending order \*/*

print\_descending(&my\_tree);

break;

default:

tree\_destroy(&my\_tree);

loop = 0;

break;

}

}

return 0;

}

void tree\_initialize(Tree \*tree)

{

tree->root=NULL;

}

void tree\_insert(Tree \*tree, int data)

{

Node\* temp=tree->root;

Node\* prev=NULL;

while(temp!=NULL)

{

if(data==temp->data)

return tree->root;

prev=temp;

if(data<temp->data)

{

if(temp->leftThread==0)

temp=temp->left;

else

break;

}

else

{

if(temp->rightThread==0)

temp=temp->right;

else

break;

}

}

Node\* tmp=(Node\*)malloc(sizeof(Node));

tmp->data=data;

tmp->leftThread=tmp->rightThread=1;

if(prev==NULL)

{

tree->root=tmp;

tmp->left=NULL;

tmp->right=NULL;

}

else if(data<(prev->data))

{

tmp->left=prev->left;

tmp->right=prev;

prev->leftThread=0;

prev->left=tmp;

}

else

{

tmp->left=prev;

tmp->right=prev->right;

prev->rightThread=0;

prev->right=tmp;

}

return tree->root;

}

Node \*inorder\_predecessor(Node \*ptr)

{

if(ptr->leftThread==1)

return ptr->left;

ptr=ptr->left;

while(ptr->rightThread==0)

ptr=ptr->right;

return ptr;

}

Node \*inorder\_successor(Node \*ptr)

{

if(ptr->rightThread==1)

return ptr->right;

ptr=ptr->right;

while(ptr->leftThread==0)

ptr=ptr->left;

return ptr;

}

void print\_ascending(Tree \*tree)

{

if(tree->root==NULL)

printf("Empty");

else

{

Node\* ptr=tree->root;

while(ptr->leftThread==0)

ptr=ptr->left;

while(ptr!=NULL)

{

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

ptr=inorder\_successor(ptr);

}

}

printf("\n");

}

void print\_descending(Tree \*tree)

{

if(tree->root==NULL)

printf("Empty");

else

{

Node\* ptr=tree->root;

while(ptr->rightThread==0)

ptr=ptr->right;

while(ptr!=NULL)

{

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

ptr=inorder\_predecessor(ptr);

}

}

printf("\n");

}

void destroy(Node \*r)

{

}

void tree\_destroy(Tree \*t)

{

}