Full program for a Binary Search Tree illustrating:

* Searching
* Minimum
* Maximum
* Inorder Successor
* Inorder Predecessor
* Insertion
* Deletion

#include<stdio.h>

#include<stdlib.h>

struct BSTNode

{

/\*

\* this is the basic structure representing a node of the Tree

\*/

**int** key; // key stores the data part of the node.

struct BSTNode \*left,\*right,\*parent;

/\*

\* the left pointer points to the left child

\* right pointer points to the right child

\* parent points to the parent of this node.

\*/

};

struct BST

{

/\*

\* Any binary search tree can be identified alone by its root.

\*/

struct BSTNode \*root;

};

struct BSTNode\*makeNode(**int** key)

{

struct BSTNode\*newNode;

newNode=(struct BSTNode\*)malloc(sizeof(struct BSTNode));

newNode->key=key;

newNode->left=newNode->right=newNode->parent=NULL;

**return** newNode;

}

**void** inorderTraversal(struct BSTNode\*root)

{

**if**(root==NULL)

**return**;

inorderTraversal(root->left);

printf("%d ",root->key);

inorderTraversal(root->right);

}

struct BSTNode\*treeMinimum(struct BSTNode\*node)

{

**while**(node->left!=NULL)

node=node->left;

**return** node;

}

struct BSTNode\*treeMaximum(struct BSTNode\*node)

{

**while**(node->right!=NULL)

node=node->right;

**return** node;

}

struct BSTNode\*inorderSuccessor(struct BSTNode\*x)

{

struct BSTNode\*y;

**if**(x->right!=NULL)

**return** treeMinimum(x->right);

y=x->parent;

**while**(y!=NULL && x==y->right)

{

x=y;

y=x->parent;

}

**return** y;

}

struct BSTNode\*inorderPredecessor(struct BSTNode\*x)

{

struct BSTNode\*y;

**if**(x->left!=NULL)

**return** treeMinimum(x->left);

y=x->parent;

**while**(y!=NULL && x==y->left)

{

x=y;

y=x->parent;

}

**return** y;

}

**void** BSTInsert(struct BST\*T,struct BSTNode\*newNode)

{

**int** direction;

struct BSTNode\*currentNode,\*previousNode;

**if**(T->root==NULL)

T->root=newNode;

**else**

{

currentNode=T->root;

**while**(currentNode!=NULL)

{

previousNode=currentNode;

**if**(currentNode->key > newNode->key)

currentNode=currentNode->left;

**else**

currentNode=currentNode->right;

}

**if**(newNode->key < previousNode->key)

previousNode->left=newNode;

**else**

previousNode->right=newNode;

newNode->parent=previousNode;

}

}

**void** transplant(struct BST\*T,struct BSTNode\*u,struct BSTNode\*v)

{

**if**(u->parent==NULL)

T->root=v;

**else** **if**(u->parent->left!=NULL && u->parent->left->key == u->key)

u->parent->left=v;

**else** **if**(u->parent->right!=NULL && u->parent->right->key == u->key)

u->parent->right=v;

**if**(v!=NULL)

v->parent=u->parent;

}

**void** BSTDelete(struct BST\*T,struct BSTNode\*z)

{

struct BSTNode\*y;

**if**(z->right == NULL)

transplant(T,z,z->left);

**else** **if**(z->left == NULL)

transplant(T,z,z->right);

**else**

{

y=treeMinimum(z->right);

**if**(y->parent!=z)

{

transplant(T,y,y->right);

y->right=z->right;

y->right->parent=y;

}

transplant(T,z,y);

y->left=z->left;

y->left->parent=y;

}

free(z);

}

/\*

the following search functions searches and returns node with key value k

if no such node is found it returns null.

Note that the search functionality is implemented in the sub tree rooted at x.

\*/

struct BSTNode\*iterativeBSTSearch(struct BSTNode\*x,**int** k)

{

**while**(x!=NULL && x->key!=k)

**if**(x->key < k)

x=x->left;

**else**

x=x->right;

**return** x;

}

struct BSTNode\*recursiveBSTSearch(struct BSTNode\*x,**int** k)

{

**if**(x==NULL || k==x->key)

**return** x;

**else** **if**(x->key < k)

**return** recursiveBSTSearch(x->left,k);

**else**

**return** recursiveBSTSearch(x->right,k);

}

**int** main()

{

struct BST\*T;

T=(struct BST\*)malloc(sizeof(struct BST\*));

T->root=NULL;

struct BSTNode \*\*nodeArray;

nodeArray=(struct BSTNode\*\*)malloc(sizeof(struct BstNode\*)\*10);

nodeArray[0]=makeNode(500);

nodeArray[1]=makeNode(200);

nodeArray[2]=makeNode(700);

nodeArray[3]=makeNode(100);

nodeArray[4]=makeNode(300);

nodeArray[5]=makeNode(400);

nodeArray[6]=makeNode(600);

nodeArray[7]=makeNode(0);

nodeArray[8]=makeNode(900);

nodeArray[9]=makeNode(800);

BSTInsert(T,nodeArray[0]);

BSTInsert(T,nodeArray[1]);

BSTInsert(T,nodeArray[2]);

BSTInsert(T,nodeArray[3]);

BSTInsert(T,nodeArray[4]);

BSTInsert(T,nodeArray[5]);

BSTInsert(T,nodeArray[6]);

BSTInsert(T,nodeArray[7]);

BSTInsert(T,nodeArray[8]);

BSTInsert(T,nodeArray[9]);

inorderTraversal(T->root);

**return** 0;

}