//steven guo

//11/20/19

//this program shows the bst functions

#include<stdio.h>

#include<stdlib.h>

#include <queue>

#include <iostream>

using namespace std;

struct node

{

int key;

struct node\* left, \* right;

};

// A utility function to create a new BST node

struct node\* newNode(int item)

{

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

temp->key = item;

temp->left = temp->right = NULL;

return temp;

}

// A utility function to do inorder traversal of BST

void inorder(struct node\* root)

{

if (root != NULL)

{

inorder(root->left);

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

inorder(root->right);

}

}

void postorder(struct node\* root)

{

if (root != NULL)

{

inorder(root->left);

inorder(root->right);

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

}

}

void preorder(struct node\* root)

{

if (root != NULL)

{

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

inorder(root->left);

inorder(root->right);

}

}

/\* A utility function to insert a new node with given key in BST \*/

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

{

/\* If the tree is empty, return a new node \*/

if (node == NULL) return newNode(key);

/\* Otherwise, recur down the tree \*/

if (key < node->key)

node->left = insert(node->left, key);

else if (key > node->key)

node->right = insert(node->right, key);

/\* return the (unchanged) node pointer \*/

return node;

}

int treeHeight(node\* root)

{

// Base Case

if (root == NULL)

return 0;

// Create an empty queue for level order tarversal

queue<node\*> q;

// Enqueue Root and initialize height

q.push(root);

int height = 0;

while (1)

{

// nodeCount (queue size) indicates number of nodes

// at current lelvel.

int nodeCount = q.size();

if (nodeCount == 0)

return height;

height++;

// Dequeue all nodes of current level and Enqueue all

// nodes of next level

while (nodeCount > 0)

{

node\* node = q.front();

q.pop();

if (node->left != NULL)

q.push(node->left);

if (node->right != NULL)

q.push(node->right);

nodeCount--;

}

}

}

struct node\* minValue(struct node\* node)

{

struct node\* current = node;

/\* loop down to find the leftmost leaf \*/

while (current && current->left != NULL)

current = current->left;

return current;

}

struct node\* deleteNode(struct node\* root, int key)

{

// base case

if (root == NULL) return root;

// If the key to be deleted is smaller than the root's key,

// then it lies in left subtree

if (key < root->key)

root->left = deleteNode(root->left, key);

// If the key to be deleted is greater than the root's key,

// then it lies in right subtree

else if (key > root->key)

root->right = deleteNode(root->right, key);

// if key is same as root's key, then This is the node

// to be deleted

else

{

// node with only one child or no child

if (root->left == NULL)

{

struct node\* temp = root->right;

free(root);

return temp;

}

else if (root->right == NULL)

{

struct node\* temp = root->left;

free(root);

return temp;

}

// node with two children: Get the inorder successor (smallest

// in the right subtree)

struct node\* temp = minValue(root->right);

// Copy the inorder successor's content to this node

root->key = temp->key;

// Delete the inorder successor

root->right = deleteNode(root->right, temp->key);

}

return root;

}

int minValue1(struct node\* node)

{

struct node\* current = node;

/\* loop down to find the leftmost leaf \*/

while (current->left != NULL)

{

current = current->left;

}

return(current->key);

}

int maxValue(struct node\* node)

{

/\* loop down to find the rightmost leaf \*/

struct node\* current = node;

while (current->right != NULL)

current = current->right;

return (current->key);

}

struct node\* search(struct node\* root, int key)

{

// Base Cases: root is null or key is present at root

if (root == NULL || root->key == key)

return root;

// Key is greater than root's key

if (root->key < key)

return search(root->right, key);

// Key is smaller than root's key

return search(root->left, key);

}

// Driver Program to test above functions

int main()

{

int option, value;

struct node\* root = NULL;

root = insert(root, 30);

insert(root, 10);

insert(root, 45);

insert(root, 38);

insert(root, 20);

insert(root, 50);

insert(root, 25);

insert(root, 33);

insert(root, 8);

insert(root, 12);

while (1) {

cout << "1. insert" << endl;

cout << "2. tree height" << endl;

cout << "3. minimum value" << endl;

cout << "4. maximum value" << endl;

cout << "5. delete" << endl;

cout << "6. search" << endl;

cout << "7. traversal" << endl;

cout << "8. exit" << endl;

cin >> option;

switch (option)

{

case 1:

cout << "enter number to enter" << endl;

cin >> value;

insert(root, value);

break;

case 2:

cout << "the height is: " << treeHeight(root) << endl;

break;

case 3:

cout << "the minimum value is: " << minValue1(root) << endl;

break;

case 4:

cout << "the maximum value is: " << maxValue(root) << endl;

break;

case 5:

cout << "enter the number to delete" << endl;

cin >> value;

deleteNode(root, value);

break;

case 6:

cout << "enter the number to search" << endl;

cin >> value;

if (search(root, value) > 0)

{

cout << "the number is in the BST" << endl;

}

else

{

cout << "the number is not in the BST" << endl;

}

break;

case 7:

cout << "preorder: " << endl;

preorder(root);

cout << endl;

cout << "inorder: " << endl;

inorder(root);

cout << endl;

cout << "postorder: " << endl;

postorder(root);

cout << endl;

break;

case 8: exit(1);

}

}

return 0;

}





