/\* Binary Tree \*/

#include <iostream>

#include <deque>

#include <climits>

#include <vector>

using namespace std;

struct Tree

{

char data;

Tree \*left;

Tree \*right;

Tree \*parent;

};

Tree\* lookUp(struct Tree \*node, char key)

{

if(node == NULL) return node;

if(node->data == key) return node;

else {

if(node->data < key)

return lookUp(node->right, key) ;

else

return lookUp(node->left, key);

}

}

Tree\* leftMost(struct Tree \*node)

{

if(node == NULL) return NULL;

while(node->left != NULL)

node = node->left;

return node;

}

struct Tree \*newTreeNode(int data)

{

Tree \*node = new Tree;

node->data = data;

node->left = NULL;

node->right = NULL;

node->parent = NULL;

return node;

}

struct Tree\* insertTreeNode(struct Tree \*node, int data)

{

static Tree \*p;

Tree \*retNode;

//if(node != NULL) p = node;

if(node == NULL) {

retNode = newTreeNode(data);

retNode->parent = p;

return retNode;

}

if(data <= node->data ) {

p = node;

node->left = insertTreeNode(node->left,data);

}

else {

p = node;

node->right = insertTreeNode(node->right,data);

}

return node;

}

void isBST(struct Tree \*node)

{

static int lastData = INT\_MIN;

if(node == NULL) return;

isBST(node->left);

/\* check if the given tree is BST \*/

if(lastData < node->data)

lastData = node->data;

else {

cout << "Not a BST" << endl;

return;

}

isBST(node->right);

return;

}

int treeSize(struct Tree \*node)

{

if(node == NULL) return 0;

else

return treeSize(node->left) + 1 + treeSize(node->right);

}

int maxDepth(struct Tree \*node)

{

if(node == NULL || (node->left == NULL && node->right == NULL))

return 0;

int leftDepth = maxDepth(node->left);

int rightDepth = maxDepth(node->right);

return leftDepth > rightDepth ?

leftDepth + 1 : rightDepth + 1;

}

int minDepth(struct Tree \*node)

{

if(node == NULL || (node->left == NULL && node->right == NULL))

return 0;

int leftDepth = minDepth(node->left);

int rightDepth = minDepth(node->right);

return leftDepth < rightDepth ?

leftDepth + 1 : rightDepth + 1;

}

bool isBalanced(struct Tree \*node)

{

if(maxDepth(node)-minDepth(node) <= 1)

return true;

else

return false;

}

/\* Tree Minimum \*/

Tree\* minTree(struct Tree \*node)

{

if(node == NULL) return NULL;

while(node->left)

node = node -> left;

return node;

}

/\* Tree Maximum \*/

Tree\* maxTree(struct Tree \*node)

{

while(node->right)

node = node -> right;

return node;

}

/\* In Order Successor - a node which has the next higher key \*/

Tree \*succesorInOrder(struct Tree \*node)

{

/\* if the node has right child, seccessor is Tree-Minimum \*/

if(node->right != NULL) return minTree(node->right);

Tree \*y = node->parent;

while(y != NULL && node == y->right) {

node = y;

y = y->parent;

}

return y;

}

/\* In Order Predecessor - a node which has the next lower key \*/

Tree \*predecessorInOrder(struct Tree \*node)

{

/\* if the node has left child, predecessor is Tree-Maximum \*/

if(node->left != NULL) return maxTree(node->left);

Tree \*y = node->parent;

/\* if it does not have a left child,

predecessor is its first left ancestor \*/

while(y != NULL && node == y->left) {

node = y;

y = y->parent;

}

return y;

}

void reverseOrderPrint(struct Tree \*node)

{

if(node == NULL) return;

if(node->left == NULL && node->right == NULL) {

cout << node->data << " ";

return;

}

reverseOrderPrint(node->right);

cout << node->data << " ";

reverseOrderPrint(node->left);

}

Tree \*lowestCommonAncestor(Tree \*node, Tree \*p, Tree \*q)

{

Tree \*left, \*right;

if(node == NULL) return NULL;

if(node->left == p || node->left == q

|| node->right == p || node->right == q) return node;

left = lowestCommonAncestor(node->left,p,q);

right = lowestCommonAncestor(node->right, p,q);

if(left && right)

return node;

else

return (left) ? left : right;

}

void clear(struct Tree \*node)

{

if(node != NULL) {

clear(node->left);

clear(node->right);

delete node;

}

}

/\* print tree in order \*/

/\* 1. Traverse the left subtree.

2. Visit the root.

3. Traverse the right subtree.

\*/

void printTreeInOrder(struct Tree \*node)

{

if(node == NULL) return;

printTreeInOrder(node->left);

cout << node->data << " ";

printTreeInOrder(node->right);

}

/\* print tree in postorder\*/

/\* 1. Traverse the left subtree.

2. Traverse the right subtree.

3. Visit the root.

\*/

void printTreePostOrder(struct Tree \*node)

{

if(node == NULL) return;

printTreePostOrder(node->left);

printTreePostOrder(node->right);

cout << node->data << " ";

}

/\* print in preorder \*/

/\* 1. Visit the root.

2. Traverse the left subtree.

3. Traverse the right subtree.

\*/

void printTreePreOrder(struct Tree \*node)

{

if(node == NULL) return;

cout << node->data << " ";

printTreePreOrder(node->left);

printTreePreOrder(node->right);

}

/\* In reverse of printTreeInOrder() \*/

void printTreeReverseOrder(struct Tree \*node)

{

if(node == NULL) return;

if(node->left == NULL && node->right == NULL) {

cout << node->data << " ";

return;

}

printTreeReverseOrder(node->right);

cout << node->data << " ";

printTreeReverseOrder(node->left);

}

/\* recursion routine to find path \*/

void pathFinder(struct Tree \*node, int path[], int level)

{

if(node == NULL) return;

// save leaf node

if(node->left == NULL && node->right == NULL) {

path[level] = node->data;

for(int i = 0; i <= level; i++) {

cout << (char)path[i];

}

cout << endl;

return;

}

// save parent node

path[level] = node->data;

pathFinder(node->left, path, level+1);

pathFinder(node->right, path, level+1);

}

bool matchTree(Tree \*r1, Tree \*r2)

{

/\* Nothing left in the subtree \*/

if(r1 == NULL && r2 == NULL)

return true;

/\* Big tree empty and subtree not found \*/

if(r1 == NULL || r2 == NULL)

return false;

/\* Not matching \*/

if(r1->data != r2->data)

return false;

return (matchTree(r1->left, r2->left) &&

matchTree(r1->right, r2->right));

}

bool subTree(Tree \*r1, Tree \*r2)

{

/\*Big tree empty and subtree not found \*/

if(r1 == NULL)

return false;

if(r1->data == r2->data)

if(matchTree(r1, r2)) return true;

return

(subTree(r1->left, r2) || subTree(r1->right, r2));

}

bool isSubTree(Tree \*r1, Tree \*r2)

{

/\* Empty tree is subtree \*/

if(r2 == NULL)

return true;

else

return subTree(r1, r2);

}

/\* change a tree so that the roles of the left

and right hand pointers are swapped at every node \*/

void mirror(Tree \*r)

{

if(r == NULL) return;

Tree \*tmp;

mirror(r->left);

mirror(r->right);

/\* swap pointers \*/

tmp = r->right;

r->right = r->left;

r->left = tmp;

}

/\* create a new tree from a sorted array \*/

Tree \*addToBST(char arr[], int start, int end)

{

if(end < start) return NULL;

int mid = (start + end)/2;

Tree \*r = new Tree;

r->data = arr[mid];

r->left = addToBST(arr, start, mid-1);

r->right = addToBST(arr, mid+1, end);

return r;

}

Tree \*createMinimalBST(char arr[], int size)

{

return addToBST(arr,0,size-1);

}

/\* Breadth first traversal using queue \*/

void BreadthFirstTraversal(Tree \*root)

{

if (root == NULL) return;

deque <Tree \*> queue;

queue.push\_back(root);

while (!queue.empty()) {

Tree \*p = queue.front();

cout << p->data << " ";

queue.pop\_front();

if (p->left != NULL)

queue.push\_back(p->left);

if (p->right != NULL)

queue.push\_back(p->right);

}

cout << endl;

}

/\* get the level of a node element: root level = 0 \*/

int getLevel(struct Tree \*node, int elm, int level)

{

if(node == NULL) return 0;

if(elm == node->data)

return level;

else if(elm < node->data)

return getLevel(node->left, elm, level+1);

else

return getLevel(node->right, elm, level+1);

}

/\* This code prints out all nodes at the same depth (level) \*/

void BreadthFirst\_LevelElement\_Print

(struct Tree \*root, vector<vector<int> > &v;)

{

if(root == NULL) return;

deque<Tree \*> q;

q.push\_back(root);

while(!q.empty()) {

Tree \*p = q.front();

int lev = getLevel(root, p->data, 0);

v[lev].push\_back(p->data);

q.pop\_front();

if(p->left) q.push\_back(p->left);

if(p->right)q.push\_back(p->right);

}

return;

}

/\* levelPrint()

prints nodes at the same level

This is simpler than the BreadthFirstTraversal(root) above

It takes 2D vector with the same size of level (= MaxDepth+1)

and fills elements as we traverse (preOrder) \*/

void levelPrint(struct Tree \*node, vector<vector<char> >&elm;, int level)

{

if(node == NULL) return;

// leaf nodes

if(node->left == NULL && node->right == NULL) {

elm[level].push\_back(node->data);

return;

}

// other nodes

elm[level++].push\_back(node->data);

levelPrint(node->left, elm, level);

levelPrint(node->right, elm, level);

}

/\* find n-th max node from a tree \*/

void NthMax(struct Tree\* t)

{

static int n\_th\_max = 5;

static int num = 0;

if(t == NULL) return;

NthMax(t->right);

num++;

if(num == n\_th\_max)

cout << n\_th\_max << "-th maximum data is "

<< t->data << endl;

NthMax(t->left);

}

/\* Converting a BST into an Array \*/

void TreeToArray(struct Tree \*node, int a[]){

static int pos = 0;

if(node){

TreeToArray(node->left,a);

a[pos++] = node->data;

TreeToArray(node->right,a);

}

}

/\* Separate even/odd level elements \*/

/\* This function is using BFS \*/

void level\_even\_odd(struct Tree \*node)

{

vector<char> evenVec, oddVec;

if (node == NULL) return;

deque<struct Tree\*> que;

que.push\_back(node);

while(!que.empty())

{

struct Tree \*p = que.front();

int level = getLevel(node, p->data, 0) ;

// even level

if (level % 2 == 0)

evenVec.push\_back(p->data);

else

oddVec.push\_back(p->data);

que.pop\_front();

if(p->left) que.push\_back(p->left);

if(p->right) que.push\_back(p->right);

}

cout << "even level elements : ";

for(int i = 0; i < evenVec.size(); i++)

cout << evenVec[i] << " ";

cout << endl << "odd level elements : ";

for(int i = 0; i < oddVec.size(); i++)

cout << oddVec[i] << " ";

cout << endl;

}

int main(int argc, char \*\*argv)

{

char ch, ch1, ch2;

Tree \*found;

Tree \*succ;

Tree \*pred;

Tree \*ancestor;

char charArr[9]

= {'A','B','C','D','E','F','G','H','I'};

Tree \*root = newTreeNode('F');

insertTreeNode(root,'B');

insertTreeNode(root,'A');

insertTreeNode(root,'D');

insertTreeNode(root,'C');

insertTreeNode(root,'E');

insertTreeNode(root,'G');

insertTreeNode(root,'I');

insertTreeNode(root,'H');

/\* is the tree BST? \*/

isBST(root);

/\* size of tree \*/

cout << "size = " << treeSize(root) << endl;

/\* max depth \*/

cout << "max depth = " << maxDepth(root) << endl;

/\* min depth \*/

cout << "min depth = " << minDepth(root) << endl;

/\* balanced tree? \*/

if(isBalanced(root))

cout << "This tree is balanced!\n";

else

cout << "This tree is not balanced!\n";

/\* min value of the tree\*/

if(root)

cout << "Min value = " << minTree(root)->data << endl;

/\* max value of the tree\*/

if(root)

cout << "Max value = " << maxTree(root)->data << endl;

/\* get the level of a data: root level = 0 \*/

cout << "Node B is at level: " << getLevel(root, 'B', 0) << endl;

cout << "Node H is at level: " << getLevel(root, 'H', 0) << endl;

cout << "Node F is at level: " << getLevel(root, 'F', 0) << endl;

/\* separate even/odd level elements \*/

level\_even\_odd(root);

ch = 'B';

found = lookUp(root,ch);

if(found) {

cout << "Min value of subtree " << ch << " as a root is "

<< minTree(found)->data << endl;

cout << "Max value of subtree " << ch << " as a root is "

<< maxTree(found)->data << endl;

}

ch = 'B';

found = lookUp(root,ch);

if(found) {

succ = succesorInOrder(found);

if(succ)

cout << "In Order Successor of " << ch << " is "

<< succesorInOrder(found)->data << endl;

else

cout << "In Order Successor of " << ch << " is None\n";

}

ch = 'E';

found = lookUp(root,ch);

if(found) {

succ = succesorInOrder(found);

if(succ)

cout << "In Order Successor of " << ch << " is "

<< succesorInOrder(found)->data << endl;

else

cout << "In Order Successor of " << ch << " is None\n";

}

ch = 'I';

found = lookUp(root,ch);

if(found) {

succ = succesorInOrder(found);

if(succ)

cout << "In Order Successor of " << ch << " is "

<< succesorInOrder(found)->data << endl;

else

cout << "In Order Successor of " << ch << " is None\n";

}

ch = 'B';

found = lookUp(root,ch);

if(found) {

pred = predecessorInOrder(found);

if(pred)

cout << "In Order Predecessor of " << ch << " is "

<< predecessorInOrder(found)->data << endl;

else

cout << "In Order Predecessor of " << ch << " is None\n";

}

ch = 'E';

found = lookUp(root,ch);

if(found) {

pred = predecessorInOrder(found);

if(pred)

cout << "In Order Predecessor of " << ch << " is "

<< predecessorInOrder(found)->data << endl;

else

cout << "In Order Predecessor of " << ch << " is None\n";

}

ch = 'I';

found = lookUp(root,ch);

if(found) {

pred = predecessorInOrder(found);

if(pred)

cout << "In Order Predecessor of " << ch << " is "

<< predecessorInOrder(found)->data << endl;

else

cout << "In Order Predecessor of " << ch << " is None\n";

}

/\* Lowest Common Ancestor \*/

ch1 = 'A';

ch2 = 'C';

ancestor =

lowestCommonAncestor(root,

lookUp(root,ch1), lookUp(root,ch2));

if(ancestor)

cout << "The lowest common ancestor of " << ch1 << " and "

<< ch2 << " is " << ancestor->data << endl;

ch1 = 'E';

ch2 = 'H';

ancestor =

lowestCommonAncestor(root,

lookUp(root,ch1), lookUp(root,ch2));

if(ancestor)

cout << "The lowest common ancestor of " << ch1 << " and "

<< ch2 << " is " << ancestor->data << endl;

ch1 = 'D';

ch2 = 'E';

ancestor =

lowestCommonAncestor(root,

lookUp(root,ch1), lookUp(root,ch2));

if(ancestor)

cout << "The lowest common ancestor of " << ch1 << " and "

<< ch2 << " is " << ancestor->data << endl;

ch1 = 'G';

ch2 = 'I';

ancestor =

lowestCommonAncestor(root,

lookUp(root,ch1), lookUp(root,ch2));

if(ancestor)

cout << "The lowest common ancestor of " << ch1 << " and "

<< ch2 << " is " << ancestor->data << endl;

ch1 = 'H';

ch2 = 'I';

ancestor =

lowestCommonAncestor(root,

lookUp(root,ch1), lookUp(root,ch2));

if(ancestor)

cout << "The lowest common ancestor of " << ch1 << " and "

<< ch2 << " is " << ancestor->data << endl;

/\* print tree in order \*/

cout << "increasing sort order\n";

printTreeInOrder(root);

cout << endl;

/\* print tree in postorder\*/

cout << "post order \n";

printTreePostOrder(root);

cout << endl;

/\* print tree in preorder\*/

cout << "pre order \n";

printTreePreOrder(root);

cout << endl;

/\* print tree in reverse order\*/

cout << "reverse order \n";

printTreeReverseOrder(root);

cout << endl;

/\* lookUp \*/

ch = 'D';

found = lookUp(root,ch);

if(found)

cout << found->data << " is in the tree\n";

else

cout << ch << " is not in the tree\n";

/\* lookUp \*/

ch = 'M';

found = lookUp(root,ch);

if(found)

cout << found->data << " is in the tree\n";

else

cout << ch << " is not in the tree\n";

/\* printing all paths :

Given a binary tree, print out all of its root-to-leaf

paths, one per line. Uses a recursive helper to do the work. \*/

cout << "printing paths ..." << endl;

int path[10];

pathFinder(root, path, 0);

/\* find n-th maximum node \*/

NthMax(root);

/\* Traversing level-order.

We visit every node on a level before going to a lower level.

This is also called Breadth-first traversal.\*/

cout << "printing with Breadth-first traversal" << endl;

BreadthFirstTraversal(root);

/\* Prints all element at the same depth (level) \*/

int row = maxDepth(root);

vector<vector<int> > levVec(row+1);

BreadthFirst\_LevelElement\_Print(root, levVec);

for(int m = 0; m < levVec.size(); m++) {

cout << "Level at " << m << ": ";

for(int n = 0; n < levVec[m].size(); n++)

cout << (char)levVec[m][n] << " ";

cout << endl;

}

/\* levelPrint()

prints nodes at the same level

This is simpler than the BreadthFirstTraversal(root) above

It takes 2D vector (elm) with the same size of level (= MaxDepth+1)

and fills elements as we traverse (preOrder) \*/

vector<vector<char> > elm;

int mxDepth = maxDepth(root);

elm.resize(mxDepth+1);

int level = 0;

levelPrint(root, elm, level);

cout << "levelPrint() " << endl;

for(int i = 0; i <= mxDepth; i++) {

cout << "level " << i << ": " ;

for(int j = 0; j < elm[i].size(); j++)

cout << elm[i][j] << " ";

cout << endl;

}

/\* convert the tree into an array \*/

int treeSz = treeSize(root);

int \*array = new int[treeSz];

TreeToArray(root,array);

cout << "New array: ";

for (int i = 0; i < treeSz; i++)

cout << (char)array[i] << ' ';

cout << endl;

delete [] array;

/\* subtree \*/

Tree \*root2 = newTreeNode('D');

insertTreeNode(root2,'C');

insertTreeNode(root2,'E');

cout << "1-2 subtree: " << isSubTree(root, root2) << endl;

Tree \*root3 = newTreeNode('B');

insertTreeNode(root3,'A');

insertTreeNode(root3,'D');

insertTreeNode(root3,'C');

insertTreeNode(root3,'E');

cout << "1-3 subtree: " << isSubTree(root, root3) << endl;

Tree \*root4 = newTreeNode('B');

insertTreeNode(root4,'D');

insertTreeNode(root4,'C');

insertTreeNode(root4,'E');

cout << "1-4 subtree: " << isSubTree(root, root4) << endl;

cout << "2-3 subtree: " << isSubTree(root2, root3) << endl;

cout << "3-2 subtree: " << isSubTree(root3, root2) << endl;

/\* swap left and right \*/

mirror(root);

/\* deleting a tree \*/

clear(root);

/\* make a new tree with minimal depth \*/

Tree \*newRoot = createMinimalBST(charArr,9);

return 0;

}