Output:

Enter file name: People.txt

Order to display (LMR,MLR,RML): LMR

LMR Order

--------------

Alisa

Ann

Carlos

David

Frank

John

Kathy

Lailee

Lisa

Mark

Michael

Richard

Rochelle

Steve

Susan

Count: 15

Depth: 4

Enter value to find: Susan

Find: Susan found!

Remove: Susan removed!

LMR Order

--------------

Alisa

Ann

Carlos

David

Frank

John

Kathy

Lailee

Lisa

Mark

Michael

Richard

Rochelle

Steve

Count: 14

Depth: 4

Press any key to continue . . .

Enter file name: People.txt

Order to display (LMR,MLR,RML): RML

RML Order

--------------

Susan

Steve

Rochelle

Richard

Michael

Mark

Lisa

Lailee

Kathy

John

Frank

David

Carlos

Ann

Alisa

Count: 15

Depth: 4

Enter value to find: Steve

Find: Steve found!

Remove: Steve removed

RML Order

--------------

Susan

Rochelle

Richard

Michael

Mark

Lisa

Lailee

Kathy

John

Frank

David

Carlos

Ann

Alisa

Count: 14

Depth: 4

Press any key to continue . . .

Enter file name: People.txt

Order to display (LMR,MLR,RML): MLR

MLR Order

--------------

Mark

Frank

Carlos

Alisa

Ann

David

Kathy

John

Lisa

Lailee

Richard

Michael

Steve

Rochelle

Susan

Count: 15

Depth: 4

Enter value to find: Mark

Find: Mark found!

Remove: Mark removed

MLR Order

--------------

Frank

Carlos

Alisa

Ann

David

Kathy

John

Lisa

Lailee

Richard

Michael

Steve

Rochelle

Susan

Count: 14

Depth: 5

Press any key to continue . . .

Node.h

#pragma once

#include <iostream>

#include <string>

using namespace std;

class node

{

friend class tree;

public:

node(string);

~node();

void insert(node\*);

void put(ostream &out);

void LMR(ostream &out);

void RML(ostream &out);

void MLR(ostream &out);

int count();

int depth();

private:

string data;

node \*left, \*right;

};

Tree.h

#pragma once

#include "node.h"

class tree

{

public:

tree();

~tree();

void insert(string);

void show(string, ostream &out);

bool find(string);

bool remove(string);

int count();

int depth();

private:

node\* remove(node\*, string);

node\* root;

};

Node.cpp

#include "node.h"

node::node(string data)

{

this->data = data;

left = NULL;

right = NULL;

}

//Deconstructor - deletes child nodes also

node::~node()

{

if (left != NULL) delete left;

if (right != NULL) delete right;

}

//Inserts node beginning at this current node

void node::insert(node\* n)

{

if (n->data < data)

if (left != NULL)

left->insert(n);

else left = n;

else if (right != NULL)

right->insert(n);

else right = n;

}

//Output to stream

void node::put(ostream &out)

{

out << data << endl;

}

//Outputs left child, self, then right child

void node::LMR(ostream &out)

{

if (left != NULL) left->LMR(out);

put(out);

if (right != NULL) right->LMR(out);

}

//Right, Middle, Left

void node::RML(ostream &out)

{

if (right != NULL) right->RML(out);

put(out);

if (left != NULL) left->RML(out);

}

//Middle, Left, Right

void node::MLR(ostream &out)

{

put(out);

if (left != NULL) left->MLR(out);

if (right != NULL) right->MLR(out);

}

//Counts nodes below this one

int node::count()

{

int count = 1; //self

if (left != NULL) count += left->count();

if (right != NULL) count += right->count();

return count;

}

//Gives depth relative to this this node

int node::depth()

{

int l = left != NULL ? 1 + left->depth() : 0; //if theres is a child, add their depth + 1 (the addition is here because a single node is depth 0)

int r = right != NULL ? 1 + right->depth() : 0;

return l > r ? l : r; //return max of left and right depths

}

Tree.cpp

#include "tree.h"

tree::tree()

{

root = NULL;

}

tree::~tree()

{

if (root != NULL)

delete root;

}

//Add new string to the tree; sort at input

void tree::insert(string data)

{

//NEW CODE

if (root == NULL)

root = new node(data);

else root->insert(new node(data));

}

//Output in specific order

void tree::show(string order, ostream &out)

{

if (root == NULL) return; //nothing to show

if (order == "LMR")

root->LMR(out);

else if (order == "RML")

root->RML(out);

else if (order == "MLR")

root->MLR(out);

}

//Returns true if string is found in the tree

bool tree::find(string data)

{

node \*p;

p = root;

while (p != NULL)

{

if (p->data == data) return true;

else if (data < p->data) p = p->left;

else p = p->right;

}

return false;

}

//Removes a string from the tree; true is successful

bool tree::remove(string data)

{

node \*temp = remove(root, data);

if (temp == NULL) //Something is wrong

return false;

root = temp;

return true;

}

//Removes a node and returns new structure

node\* tree::remove(node\* base, string data)

{

//Base case

if (base == NULL)

return base;

if (data < base->data)

base->left = remove(base->left, data);

else if (data > base->data)

base->right = remove(base->right, data);

else

{

// Check if only one child (or none)

if (base->left == NULL)

{

node \*temp = base->right;

base->right = NULL;

delete base;

return temp;

}

else if (base->right == NULL)

{

node \*temp = base->left;

base->left = NULL;

delete base;

return temp;

}

// Two children, add right tree to left tree

node \*temp = base->left;

temp->insert(base->right);

// Remove and remove the target node

base->left = NULL;

base->right = NULL;

delete base;

return temp;

}

return base;

}

//Counts node in the tree

int tree::count()

{

if (root != NULL)

return root->count();

else return 0;

}

//Gives depth of the tree

int tree::depth()

{

if (root != NULL)

return root->depth();

else return -1; //no nodes, invalid depth

}