//============================================================================

// Name : BinarySearchTree.cpp

// Author : Ro Jackson

// Version : 1.0

// Copyright : Copyright � 2023 SNHU COCE

// Description : Lab 5-2 Binary Search Tree

//============================================================================

#include <iostream>

#include <time.h>

#include "CSVparser.hpp"

using namespace std;

//============================================================================

// Global definitions visible to all methods and classes

//============================================================================

// forward declarations

double strToDouble(string str, char ch);

// define a structure to hold bid information

struct Bid {

string bidId; // unique identifier

string title;

string fund;

double amount;

Bid() {

amount = 0.0;

}

};

// Internal structure for tree node

struct Node {

Bid bid;

Node \*left;

Node \*right;

// default constructor

Node() {

left = nullptr;

right = nullptr;

}

// initialize with a bid

Node(Bid aBid) :

Node() {

bid = aBid;

}

};

//============================================================================

// Binary Search Tree class definition

//============================================================================

/\*\*

\* Define a class containing data members and methods to

\* implement a binary search tree

\*/

class BinarySearchTree {

private:

Node\* root;

void addNode(Node\* node, Bid bid);

void inOrder(Node\* node);

void postOrder(Node\* node);

void preOrder(Node\* node);

Node\* removeNode(Node\* node, string bidId);

public:

BinarySearchTree();

virtual ~BinarySearchTree();

void InOrder();

void PostOrder();

void PreOrder();

void Insert(Bid bid);

void Remove(string bidId);

Bid Search(string bidId);

};

/\*\*

\* Default constructor

\*/

BinarySearchTree::BinarySearchTree() {

//root is equal to nullptr

this-> root = nullptr

}

/\*\*

\* Destructor

\*/

BinarySearchTree::~BinarySearchTree() {

// recurse from root deleting every node

}

/\*\*

\* Traverse the tree in order

\*/

void BinarySearchTree::InOrder() {

// call inOrder fuction and pass root

inOrder (root);

}

/\*\*

\* Traverse the tree in post-order

\*/

void BinarySearchTree::PostOrder() {

// postOrder root

postOrder(root);

}

/\*\*

\* Traverse the tree in pre-order

\*/

void BinarySearchTree::PreOrder() {

// preOrder root

preOrder(root);

}

/\*\*

\* Insert a bid

\*/

void BinarySearchTree::Insert(Bid bid) {

// if root equarl to null ptr

// root is equal to new node bid

// else

// add Node root and bid

Node\* currentNode = root;

if (root == NULL) {

root = new Node(bid);

}

else{

while (currentNode != NULL) {

if (bid.bidId < currentNode ->bid.bidId) {

if (currentNode -> left == nullptr) {

currentNode -> left = new Node(bid);

currentNode = NULL;

}

else {

currentNode = currentNode -> left;

if (currentNode -> right == nullptr) {

currentNode -> = new Node(bid);

currentNode = NULL;

}

}

else{

currentNode = currentNode -> right;

}

}

}

\* Remove a bid

\*/

void BinarySearchTree::Remove(string bidId) {

// remove node root bidID

Node\* par = NULL;

Node\* curr = root;

while (curr != NULL) {

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

if (par == NULL) {

root = nullptr;

}

else if (par->left == curr) {

par->left = NULL;

}

else {

par->right = NULL;

}

}

else if (curr->right == NULL) {

if (par == NULL) {

root = curr->left;

}

else if (par->left == curr) {

par->left = curr->left;

}

else {

par->right = curr->left;

}

}

else if (curr->left == NULL) {

if (par == NULL) {

root = curr->right;

}

else if (par->left == curr) {

par->left = curr->right;

}

else {

par->right = curr->right;

}

}

else {

Node\* suc = curr->right;

while (suc->left != NULL) {

suc = suc->left;

}

Node successorData = Node(suc->bid);

Remove(suc->bid.bidId);

curr->bid = successorData.bid;

}

//cout << "\nNode found and removed" << endl;

return;

}

else if (curr->bid.bidId < bidId) {

par = curr;

curr = curr->right;

}

else {

par = curr;

curr = curr->left;

}

}

cout << "\nValue not found" << endl;

return;

}

}

/\*\*

\* Search for a bid

\*/

Bid BinarySearchTree::Search(string bidId) {

// set current node equal to root

// keep looping downwards until bottom reached or matching bidId found

// if match found, return current bid

// if bid is smaller than current node then traverse left

// else larger so traverse right

Bid bid;

Node\* currentNode = root;

while (currentNode != NULL) {

}

if (currentNode->bid.bidId == bidId) {

return currentNode->bid;

}

else if (bidId < currentNode->bid.bidId) {

currentNode = currentNode->left;

}

else {

currentNode = currentNode->right;

}

}

//not found

//cout << "Value not found." << endl;

return bid;

}

/\*\*

\* Add a bid to some node (recursive)

\*

\* @param node Current node in tree

\* @param bid Bid to be added

\*/

void BinarySearchTree::addNode(Node\* node, Bid bid) {

//addNode root

// if node is larger then add to left

// if no left node

if (bid.key < node ->bid.key) {

if (node->left == nullptr)

// this node becomes left

// else recurse down the left node

} else {

addNode(node -> left, bid);

}

// else

// if no right node

// this node becomes right

} else{

if (node-> right == nullptr) {

node ->right = new Node(bid);

} else {

addNode(node ->right, bid);

}

}

void BinarySearchTree::inOrder(Node\* node) {

//inOrder root

//if node is not equal to null ptr

//InOrder not left

//output bidID, title, amount, fund

//InOder right

if (node == NULL) {

return;

}

inOrder (node-> left);

cout << node ->bid.bidId << " : " << node -> bid.title << " | " << node ->bid.amount << " | " << node->bid.fund << endl;

inOrder(node ->right);

}

void BinarySearchTree::postOrder(Node\* node) {

//postOrder root

//if node is not equal to null ptr

//postOrder left

//postOrder right

//output bidID, title, amount, fund

if (node == NULL) {

return;

}

postOrder (node-> left);

postOrder (node -> right);

cout << node -> bid.bidId << " : " << node-> bid.title << " | " node ->bid.amount << " | " << node-> bid.fund << endl;

postOrder( node -> left);

postOrder (node -> right);

}

void BinarySearchTree::preOrder(Node\* node) {

//preOder root

//if node is not equal to null ptr

//output bidID, title, amount, fund

//postOrder left

//postOrder right

if (node == NULL) {

return;

}

preOrder(node -> left);

preOrder(node ->right);

cout << node-> bid.bidId << " : " << node-> bid.title << " | " node-> bid.amount << " | "<< node-> bid.fund << endl;

preOrder(node ->left);

preOrder(node -> right);

}

/\*\*

\* Remove a bid from some node (recursive)

\*/

Node\* BinarySearchTree::removeNode(Node\* node, string bidId) {

// Implement removing a bid from the tree

// if node = nullptr return node

// (otherwise recurse down the left subtree)

// check for match and if so, remove left node using recursive call

// (otherwise recurse down the right subtree)

// check for match and if so, remove right node using recursive call

// (otherwise no children so node is a leaf node)

// if left node = nullptr && right node = nullptr delete node

// (otherwise check one child to the left)

// if left node != nullptr && right node = nullptr delete node

// (otherwise check one child to the right)

// if left node = nullptr && right node != nullptr delete node

// (otherwise more than one child so find the minimum)

// create temp node to right

// while left node is not nullptr keep moving temp left

// make node bid (right) equal to temp bid (left)

// remove right node using recursive call

// return node

if (node == nullptr) {

return node;

}

if (bidId < node-> bid.bidId) {

node->left = removeNode (node->, bidId);

}

else if (bidId> node->bid.bidId){

node->right = removeNode(node-> right, bidId);

}

else{

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

delete node;

node = nullptr;

}

else if (node->left != nullptr && node-> right == nullptr) {

Node\* temp = node;

node = node->left;

delete temp;

}

else if (node->left == nullptr && node-> right != nullptr){

Node\* temp = node;

node = node-> right;

delete temp;

}

else {

Node\* temp = findMin(node-> right);

node-> bid = temp->bid;

node->right = removeNode (node-> right, temp->bid.bidId);

}

}

return node;

}

//help function to fidn the minimum node in subtree

Node\* BinarySearchTree ::findMin(Node\* node) {

while (node-> left != nullptr){

node = node->left;

}

return node;

}

//============================================================================

// Static methods used for testing

//============================================================================

/\*\*

\* Display the bid information to the console (std::out)

\*

\* @param bid struct containing the bid info

\*/

void displayBid(Bid bid) {

cout << bid.bidId << ": " << bid.title << " | " << bid.amount << " | "

<< bid.fund << endl;

return;

}

/\*\*

\* Load a CSV file containing bids into a container

\*

\* @param csvPath the path to the CSV file to load

\* @return a container holding all the bids read

\*/

void loadBids(string csvPath, BinarySearchTree\* bst) {

cout << "Loading CSV file " << csvPath << endl;

// initialize the CSV Parser using the given path

csv::Parser file = csv::Parser(csvPath);

// read and display header row - optional

vector<string> header = file.getHeader();

for (auto const& c : header) {

cout << c << " | ";

}

cout << "" << endl;

try {

// loop to read rows of a CSV file

for (unsigned int i = 0; i < file.rowCount(); i++) {

// Create a data structure and add to the collection of bids

Bid bid;

bid.bidId = file[i][1];

bid.title = file[i][0];

bid.fund = file[i][8];

bid.amount = strToDouble(file[i][4], '$');

//cout << "Item: " << bid.title << ", Fund: " << bid.fund << ", Amount: " << bid.amount << endl;

// push this bid to the end

bst->Insert(bid);

}

} catch (csv::Error &e) {

std::cerr << e.what() << std::endl;

}

}

/\*\*

\* Simple C function to convert a string to a double

\* after stripping out unwanted char

\*

\* credit: http://stackoverflow.com/a/24875936

\*

\* @param ch The character to strip out

\*/

double strToDouble(string str, char ch) {

str.erase(remove(str.begin(), str.end(), ch), str.end());

return atof(str.c\_str());

}

/\*\*

\* The one and only main() method

\*/

int main(int argc, char\* argv[]) {

// process command line arguments

string csvPath, bidKey;

switch (argc) {

case 2:

csvPath = argv[1];

bidKey = "98223";

break;

case 3:

csvPath = argv[1];

bidKey = argv[2];

break;

default:

csvPath = "eBid\_Monthly\_Sales.csv";

bidKey = "98223";

}

// Define a timer variable

clock\_t ticks;

// Define a binary search tree to hold all bids

BinarySearchTree\* bst;

bst = new BinarySearchTree();

Bid bid;

int choice = 0;

while (choice != 9) {

cout << "Menu:" << endl;

cout << " 1. Load Bids" << endl;

cout << " 2. Display All Bids" << endl;

cout << " 3. Find Bid" << endl;

cout << " 4. Remove Bid" << endl;

cout << " 9. Exit" << endl;

cout << "Enter choice: ";

cin >> choice;

switch (choice) {

case 1:

// Initialize a timer variable before loading bids

ticks = clock();

// Complete the method call to load the bids

loadBids(csvPath, bst);

//cout << bst->Size() << " bids read" << endl;

// Calculate elapsed time and display result

ticks = clock() - ticks; // current clock ticks minus starting clock ticks

cout << "time: " << ticks << " clock ticks" << endl;

cout << "time: " << ticks \* 1.0 / CLOCKS\_PER\_SEC << " seconds" << endl;

break;

case 2:

bst->InOrder();

break;

case 3:

ticks = clock();

bid = bst->Search(bidKey);

ticks = clock() - ticks; // current clock ticks minus starting clock ticks

if (!bid.bidId.empty()) {

displayBid(bid);

} else {

cout << "Bid Id " << bidKey << " not found." << endl;

}

cout << "time: " << ticks << " clock ticks" << endl;

cout << "time: " << ticks \* 1.0 / CLOCKS\_PER\_SEC << " seconds" << endl;

break;

case 4:

bst->Remove(bidKey);

break;

}

}

cout << "Good bye." << endl;

return 0;

}