/\*

\* BinarySearchTree.hpp

\*

\* Created on: Apr 14, 2020

\* Author: Lesle

\*/

#ifndef BINARYSEARCHTREE\_HPP\_

#define BINARYSEARCHTREE\_HPP\_

#include <algorithm>

#include <iostream>

#include <time.h>

#include "Capstone.hpp"

#include "CSVparser.hpp"

//#include "CS499Capstone.hpp"

using namespace std;

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

// Global definitions visible to all methods and classes

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

// forward declarations

// FIXME (1): Internal structure for tree node

struct BstNode {

Bid bid;

BstNode \*left;

BstNode \*right;

// default constructor

BstNode() {

left = nullptr;

right = nullptr;

}

// initialize with a bid

BstNode(Bid aBid) :

BstNode() {

bid = aBid;

}

};

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

// Binary Search Tree class definition

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

/\*\*

\* Define a class containing data members and methods to

\* implement a binary search tree

\*/

class BinarySearchTree {

private:

BstNode\* root;

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

void inOrder(BstNode\* node);

void postOrder(BstNode\* node);

void preOrder(BstNode\* node);

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

public:

BinarySearchTree();

virtual ~BinarySearchTree();

void InOrder();

void Insert(Bid bid);

void PostOrder();

void PreOrder();

void Remove();

Bid Search();

};

/\*\*

\* Default constructor

\*/

BinarySearchTree::BinarySearchTree() {

// initialize housekeeping variables

root = nullptr;

}

/\*\*

\* Destructor

\*/

BinarySearchTree::~BinarySearchTree() {

// recurse from root deleting every node

}

/\*\*

\* Traverse the tree in order

\*/

void BinarySearchTree::InOrder() {

inOrder(root);

}

/\*\*

\* Traverse the tree in post-order

\*/

void BinarySearchTree::PostOrder() {

postOrder(root);

}

/\*\*

\* Traverse the tree in pre-order

\*/

void BinarySearchTree::PreOrder() {

preOrder(root);

}

/\*\*

\* Insert a bid

\*/

void BinarySearchTree::Insert(Bid bid) {

// FIXME (2a) Implement inserting a bid into the tree

if (root == nullptr) {

root = new BstNode(bid);

} else {

this->addNode(root, bid);

}

}

/\*\*

\* Remove a bid

\*/

void BinarySearchTree::Remove() {

// FIXME (4a) Implement removing a bid from the tree

string bidId;

cout << "Which bid would you like to remove? (use 98109 for test purposes)" << endl;

cin >> bidId;

this->removeNode(root, bidId);

}

/\*\*

\* Search for a bid

\*/

Bid BinarySearchTree::Search() {

// FIXME (3) Implement searching the tree for a bid

string bidId;

cout << "Enter Bid ID for search:" << endl;

cin >> bidId;

BstNode\* current = root;

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

while (current != nullptr) {

// if match found, return it

if (current->bid.bidId.compare(bidId) == 0) {

return current->bid;

}

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

if (bidId.compare(current->bid.bidId) < 0) {

current = current->left;

// else larger so traverse right

} else {

current = current->right;

}

}

Bid bid;

return bid;

}

/\*\*

\* Add a bid to some node (recursive)

\*

\* @param node Current node in tree

\* @param bid Bid to be added

\*/

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

// FIXME (2b) Implement inserting a bid into the tree

// if node is larger then add to left

if (node->bid.bidId.compare(bid.bidId) > 0) {

// if no left node

if (node->left == nullptr) {

// this node becomes left

node->left = new BstNode(bid);

} else {

// recurse down the left node

this->addNode(node->left, bid);

}

} else {

// if no right node

if (node->right == nullptr) {

// this node becomes right

node->right = new BstNode(bid);

} else {

// recurse down the left node

this->addNode(node->right, bid);

}

}

}

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

if (node != nullptr) {

inOrder(node->left);

cout << node->bid.bidId << ": " << node->bid.title << " | "

<< node->bid.amount << " | " << node->bid.fund << endl;

inOrder(node->right);

}

}

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

if (node != nullptr) {

postOrder(node->left);

postOrder(node->right);

cout << node->bid.bidId << ": " << node->bid.title << " | "

<< node->bid.amount << " | " << node->bid.fund << endl;

}

}

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

if (node != nullptr) {

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)

\*/

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

// FIXME (4b) Implement removing a bid from the tree

if (node == nullptr) {

return node;

// recurse down the left subtree

} else if (bidId.compare(node->bid.bidId) < 0) {

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

// recurse down the right subtree

} else if (bidId.compare(node->bid.bidId) > 0) {

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

} else {

// no children so node is a leaf node

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

delete node;

node = nullptr;

// one child to the left

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

BstNode\* temp = node;

node = node->left;

delete temp;

// one child to the right

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

BstNode\* temp = node;

node = node->right;

delete temp;

// two children

} else {

// find the minimum

BstNode\* temp = node->right;

while (temp->left != nullptr) {

temp = temp->left;

}

node->bid = temp->bid;

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

}

}

return node;

}

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

// Static methods used for testing

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

/\*\*

\* 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 bstLoadBids(BinarySearchTree\* bst) {

string csvPath;

cout << "What file would you like to load: (use eBid\_Monthly\_Sales\_Dec\_2016.csv for test purposes) " << endl;

cin >> csvPath;

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;

}

}

/\*\*

\* The one and only main() method

\*/

int bstMain() {

// Define a timer variable

clock\_t ticks;

// Define a binary search tree to hold all bids

BinarySearchTree\* bst;

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:

bst = new BinarySearchTree();

// Initialize a timer variable before loading bids

ticks = clock();

// Complete the method call to load the bids

bstLoadBids(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();

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

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

displayBid(bid);

} else {

cout << "Bid Id " << bid.bidId << " not found." << endl;

}

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

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

break;

case 4:

bst->Remove();

break;

}

}

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

return 0;

}

#endif /\* BINARYSEARCHTREE\_HPP\_ \*/