5-3 Milestone: Tree Data Structure Pseudocode

Fred Wahab

CS-300

// initialize variables

Initialize class Course {

string courseID

string courseName

vector<string> coursePrereqs

};

Initialize vector<Course> courses;

Initialize string line variable for file navigation;

// load file and place data into courses vector

Open file into file variable using ifstream;

while going through each line in the file {

vector<string> courseLine

Course newCourse

parse each line seperated by “,” into courseLine vector

if courseLine.size < 2, return error “invalid data file”

for each element in courseLine {

if element index equals 0, add to newCourse.courseID

if element index equals 1, add to newCourse.courseName

if courseLine.size > 2

if element at index courseLine.size -1 does not equal courses.Course.courseID, return “prerequisites not found”

if element index > 2, pushback to newCourse.course

}

}

pushback newCourse to courses

};

Close file;

// search vector for matching course and print

get string searchID from input

bool courseFound equals false

for every course in courses {

if course.courseID equals searchID {

courseFound equals true

then print

“Course ID: “ << course.courseID

“Course Name: “ << course.courseName

“Prerequisites: “

for every prereq in course.coursePrereqs, print prereq

}

}

if courseFound equals false, print “Course not found”

**// Hashtable**

int DEFAULT\_SIZE equals 8

class Hashtable {

// initialize Node structure

struct Node {

Course course

Node\* next

int key

Node() {

key equals UINT\_MAX

next equals null

}

Node(Course acourse): Node() {

course equals acourse

}

Node(Course acourse, int akey) : Node(acourse) {

key equals akey

}

};

int size equals DEFAULT\_SIZE

vector<Node> table;

};

int Hashtable::Hash(string courseNum) {

return atoi(courseNum.substr(4).c\_str()) % size

}

Hashtable::Add(Course acourse) {

Create key for acourse by hashing courseNumber

Create Node\* node to retrieve node using key

if node equals null {

Create new node newCourse with acourse and key

Insert contents of newCourse into table at position[key]

}

else if node’s key equals UINT\_MAX) {

node’s key equals key

node’s course equals acourse

node’s next equals null

}

else {

while node’s next does not equal null {

node equals node’s next

Create new node newCourse with acourse and key

node’s next equals newCourse

}

}

Hashtable::Print() {

Initialize vector of Nodes sortedTable

for each node in table {

if node’s key does not equal UINT\_MAX {

push node to back of sortedTable

}

Create Node\* listNode and set to node’s next

while listNode does not equal null {

push\_back listNode to sortedTable

listNode equals listNode’s next

}

SelectionSort(sortedTable)

}

for each node in sortedTable {

Invoke node’s course print()

}

}

Hashtable::SelectionSort(vector<Node> &sortedTable) {

initialize int min

for i equals 0 to sortedTable minus 1 {

set min to i

for j equals i plus 1 to end of sortedTable {

if courseNumber at j is < courseNumber at min set min to j

}

Swap node at i with min

}

}

Course Hashtable::Search(string courseNum) {

Create empty course obj

For each node in this table

if node’s course’s courseNumber equals courseNum, return node’s course

Create Node\* listNode and set to node’s next

while listNode does not equal null {

if listNode’s courseNumber equals courseNum, return listNode’s course

listNode equals listNode’s next

}

return empty

}

LoadCourses(string fileName, Hashtable &Htable) {

Initialize fstream fileStream to get contents of file

Initialize string line to hold a single line in file

Initialize stringstream lineStream to get contents of each line

Initialize string token to hold a single word in line

Open fileName with fileStream

Initialize int count to hold the token count per line in file

Get line from fileStream until none left

Fill lineStream with current line

Set count to 1

Create Course acourse for each line in file

Get token from lineStream up to ‘,’ until none left

if count equals 1 {

acourse’s courseNumber equals token

increment count

}

else if count equals 2 {

acourse’s courseName equals token

increment count

}

else {

if token exists in Htable as a course, add token to acourse’s PreReqs

else output file format error

increment count

}

if count is less than 2 {

output "Error in file format, each course must have course # and name."

}

Add acourse to Htable

Clear lineStream for next line

}

**// Tree Data**

class BST {

struct Node {

Course course;

Node\* left;

Node\* right;

Node() {

left equals nullptr;

right equals nullptr;

}

Node(Course acourse) : Node() {

course equals acourse;

}

~Node() {

delete left;

delete right;

}

};

Node\* root;

void BST::InOrder() {

inOrder(root)

}

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

if (node is not empty) {

recursively traverse node’s left sub-tree

invoke node’s course print()

recursively traverse node’s right sub-tree

}

}

void BST::Insert(Course acourse) {

if (root is empty) set root to new node with acourse.

else addNode(root, acourse)

}

void BST::addNode(Node\* node, Course acourse) {

if (acourse’s courseNumber < current node’s courseNumber) {

if (node’s left is empty) add new Node with course at node’s left

else recursively traverse node’s left sub-tree

}

else {

if (node’s right is empty) add new Node with course at node’s right

else recursively traverse node’s right sub-tree

}

}

void BST::Remove(string courseNum) {

removeNode(root, nullptr, courseNum)

}

void BST::removeNode(Node\* node, Node\* par, string courseNum) {

if (node’s course courseNumber equals courseNum ) {

if (node’s left is nullptr AND node’s right is nullptr) set node to nullptr

else if (node’s left is not nullptr) {

if (par is nullptr) set root to root’s left

else if (par’s left is node) set par’s left to node’s left

else set par’s right to node’s left

}

else if (node’s right is not nullptr) {

if (par is nullptr) set root to root’s right

else if (par’s left is node) set par’s left to node’s right

else set par’s right to node’s right

}

else {

set Node pointer suc to node’s right

while (suc’s left is not nullptr) {

set par to suc

set suc to suc’s left

}

Set Node pointer temp to suc

removeNode(suc, par, courseNum)

set node to temp

}

}

else if (node’s course courseNumber > courseNum) removeNode(node’s left, node, courseNum)

else removeNode(node’s right, node, courseNum)

}

Course BST::Search(string courseNum) {

set Node pointer current to root

while (current is not nullptr) {

if (current’s course courseNumber equals courseNum) return current’s course

if (current’s course courseNumber is greater than courseNum) set current to current’s left

else set current to current’s right

}

create empty course

return empty course

}

void LoadCourses(string fileName, BST &bst) {

Initialize fstream fileStream to get contents of file

Initialize string line to hold a single line in file

Initialize stringstream lineStream to get contents of each line

Initialize string token to hold a single word in line

Open fileName with fileStream

Initialize int count to hold the token count per line in file

Get line from fileStream until none left

Fill lineStream with current line

Set count to 1

Create Course acourse for each line in file

Get token from lineStream up to ‘,’ until none left

if (count equals 1) {

set acourse’s courseNumber to token

increment count

}

else if (count equals 2) {

set acourse’s courseName to token

increment count

}

else {

if (token exists in bst as a course) add token to acourse’s PreReqs

else output file format error

increment count

}

if (count is less than 2) {

output "Error in file format, each course must have course # and name."

}

Insert acourse into bst

clear lineStream for next line

}