Project 1

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

}

**// Menu**

Create schedule object to hold courses

Initialize string coursekey

Initialize Course acourse

Initialize int choice to 0

Initialize int choice2 to 0

while (choice does not equal 9) {

output "Menu:"

output " 1. Load Schedule\n"

output " 2. Display\n"

output " 3. Remove Course\n"

output " 9. Exit\n"

output "Enter choice: "

wait for input and store in choice

switch (choice) {

case 1:

LoadCourses(fileName, schedule)

break

case 2:

while (choice2 equals 0) {

output "1 ). Display Schedule\n"

output "2 ). Display Course\n"

output "Enter choice: "

wait for input and store in choice2

switch (choice2) {

case 1:

print schedule

break

case 2:

output "Enter course number: "

wait for input and store in courseKey

set acourse to schedule.Search(courseKey)

if (acourse is empty) output "Course is not in schedule.\n”

else print acourse

break

}

}

Set choice2 to 0

break

case 3:

output "Enter course number: "

wait for input and store in courseKey

if (coursekey is not found in schedule) {

output "Course does not exist.\n"

break

}

else remove courseKey from schedule

output courseKey " removed.\n"

break

}

}

output "goodbye.\n"

**// Evaluation**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Initialize fstream fileStream to get contents of file** | 1 | 1 | 1 |
| **Initialize string line to hold a single line in file** | 1 | 1 | 1 |
| **Initialize stringstream lineStream to get contents of each line** | 1 | 1 | 1 |
| **Initialize string token to hold a single word in line** | 1 | 1 | 1 |
| **Open fileName with fileStream** | 1 | 1 | 1 |
| **Initialize int count to hold the token count per line in file** | 1 | 1 | 1 |
| **Get line from fileStream until none left** | 1 | n | n |
| **Fill lineStream with current line** | 1 | n | n |
| **Set count to 1** | 1 | n | n |
| **Create a course for each line in file** | 1 | n | n |
| **Get token from lineStream up to ‘,’ until none left** | 1 | 2n | 2n |
| **if (count == 1)** | 1 | n | n |
| **set a course’s courseNumber to token** | 1 | n | n |
| **increment count** | 1 | n | n |
| **else if (count == 2)** | 1 | n | n |
| **set a course’s courseName to token** | 1 | n | n |
| **increment count** | 1 | n | n |
| **else** |  |  |  |
| **if (token exists in courses as a course)** | 1 | n | n |
| **add token to a course’s PreReqs** | 1 | n | n |
| **else output file format error** | 1 | 1 | 1 |
| **increment count** | 1 | n | n |
| **if (count < 2)** | 1 | 1 | 1 |
| **output "Error in file format”** | 1 | 1 | 1 |
| **push a course to back of courses** | 1 | n | n |
| **Clear lineStream for next line** | 1 | n | n |
| **Total Cost** | | | 17n + 6 |
| **Runtime** | | | O(n) |

**// Hashtable**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Initialize fstream fileStream to get contents of file** | 1 | 1 | 1 |
| **Initialize string line to hold a single line in file** | 1 | 1 | 1 |
| **Initialize stringstream lineStream to get contents of each line** | 1 | 1 | 1 |
| **Initialize string token to hold a single word in line** | 1 | 1 | 1 |
| **Open fileName with fileStream** | 1 | 1 | 1 |
| **Initialize int count to hold the token count per line in file** | 1 | 1 | 1 |
| **Get line from fileStream until none left** | 1 | n | n |
| **Fill lineStream with current line** | 1 | n | n |
| **Set count to 1** | 1 | n | n |
| **Create a course for each line in file** | 1 | n | n |
| **Get token from lineStream up to ‘,’ until none left** | 1 | 2n | 2n |
| **if (count == 1)** | 1 | n | n |
| **set a course’s courseNumber to token** | 1 | n | n |
| **increment count** | 1 | n | n |
| **else if (count == 2)** | 1 | n | n |
| **set a course’s courseName to token** | 1 | n | n |
| **increment count** | 1 | n | n |
| **else** |  |  |  |
| **if (token exists in Htable as a course)** | 1 | n-1 | n |
| **add token to a course’s PreReqs** | 1 | n-1 | n |
| **else output file format error** | 1 | 1 | 1 |
| **increment count** | 1 | n-1 | n |
| **if (count < 2)** | 1 | 1 | 1 |
| **output "Error in file format”** | 1 | 1 | 1 |
| **add a course to Htable** | n | n | n2 |
| **Clear lineStream for next line** | 1 | n | n |
| **Create key for a course by hashing a course’s courseNumber** | 1 | 1 | 1 |
| **Create Node\* node to retrieve node using key** | 1 | 1 | 1 |
| **if (node == nullptr)** | 1 | 1 | 1 |
| **Create new node newCourse with a course and key** | 1 | 1 | 1 |
| **Insert contents of newCourse into table at position[key]** | 1 | 1 | 1 |
| **else if (node’s key == UNIT\_MAX)** | 1 | 1 | 1 |
| **Update node’s key to key** | 1 | 1 | 1 |
| **Update node’s course to acourse** | 1 | 1 | 1 |
| **Update node’s next to nullptr** | 1 | 1 | 1 |
| **else** |  |  |  |
| **while (node’s next != nullptr)** | 1 | n | n |
| **set node to node’s next** | 1 | 1 | 1 |
| **Create new node newCourse with a course and key** | 1 | 1 | 1 |
| **Set node’s next to newCourse** | 1 | 1 | 1 |
| **Total Cost** | | | n2 + 17n + 9 |
| **Runtime** | | | O(n2) |

**// Tree Data**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Initialize fstream fileStream to get contents of file** | 1 | 1 | 1 |
| **Initialize string line to hold a single line in file** | 1 | 1 | 1 |
| **Initialize stringstream lineStream to get contents of each line** | 1 | 1 | 1 |
| **Initialize string token to hold a single word in line** | 1 | 1 | 1 |
| **Open fileName with fileStream** | 1 | 1 | 1 |
| **Initialize int count to hold the token count per line in file** | 1 | 1 | 1 |
| **Get line from fileStream until none left** | 1 | n | n |
| **Fill lineStream with current line** | 1 | n | n |
| **Set count to 1** | 1 | n | n |
| **Create course for each line in file** | 1 | n | n |
| **Get token from lineStream up to ‘,’ until none left** | 1 | 2n | 2n |
| **if (count == 1)** | 1 | n | n |
| **set a course’s courseNumber to token** | 1 | n | n |
| **increment count** | 1 | n | n |
| **else if (count == 2)** | 1 | n | n |
| **set a course’s courseName to token** | 1 | n | n |
| **increment count** | 1 | n | n |
| **else** | 1 | n | n |
| **if (token exists in bst as a course)** | 1 | n | n |
| **add token to a course’s PreReqs** | 1 | n | n |
| **else output file format error** | 1 | 1 | 1 |
| **increment count** | 1 | n | n |
| **if (count < 2)** | 1 | 1 | 1 |
| **output "Error in file format”** | 1 | 1 | 1 |
| **insert a course into bst** | n | n | n2 |
| **Clear lineStream for next line** | 1 | n | n |
| **if (acourse’s courseNumber < current node’s courseNumber)** | 1 | 1 | 1 |
| **if (node’s left child is empty)** | 1 | 1 | 1 |
| **add new Node with course at node’s left child** | 1 | 1 | 1 |
| **else recursively traverse node’s left sub-tree** | 1 | n | n |
| **else** |  |  |  |
| **if (node’s right child is empty)** | 1 | 1 | 1 |
| **add new Node with course at node’s right child** | 1 | 1 | 1 |
| **else recursively traverse node’s right sub-tree** | 1 | n | n |
| **if (root is empty) set root to new node with a course** | 1 | 1 | 1 |
| **else addNode(root, course)** | n+3 | 1 | n+3 |
| **Total Cost** | | | n2 + 17n + 9 |
| **Runtime** | | | O(n2) |

**// Advantages/ Disadvantages**

Data structures help the storage of and provide convenience while retrieving the data from storage device. Each data structure used in this project has its unique advantages and disadvantages. The first of the three data structures that will be evaluated are vectors. The advantages of using vectors are that they are easy to implement, can be searched in O(logn) time if sorted with binary search, and insertion at the back of the vector occurs in constant time. Some disadvantages of vectors are that they must be sorted to fully utilize search capabilities, removing elements shifts the index of the other elements, and it can take up more space than necessary. The second data structure that will be evaluated are hash tables. Hash tables allow direct access to items and allow editing in constant time. They tend to be more efficient than search trees or any other table lookup structure. The main disadvantages of hash tables are that they take up more space than other data structures and they tend to run slower due to cache misses caused by elements being randomly stored in memory. The final data structure to be evaluated is the tree. The tree data structure has fast access speeds, can retrieve item in order, and can edit elements in O(logn) time. The disadvantages of using tree structure is that it is sensitive, the data structure can cause stack overflow when using recursion and the shape of the tree is dependent on the first item that is inserted.

**// Recommendation**

After studying the different types of data structures and evaluating the project’s functional requirements, I think that a binary tree would be the most complete solution for storing course objects. This is primarily due to the courses being sorted in alphabetical order the binary tree having the tree sort algorithm built into the data structure. While the other data structure has the ability to sort through data in alphabetic order, it is more cumbersome and can cause slower results. While the binary search tree may not have all of the features the other data structures have, it should be the most efficient tool for this purpose.