**Project One**

**Nathan Anglin**

**CS-300-H7076 DSA: Analysis and Design**

**Menu Pseudocode**

***//All methods use the same names for clarity, pseudocode is assumed to be 3 separate programs,***

***//each with only the vector, hash table or tree pseudocode included in that associated program.***

Int choice = 0

WHILE (choice is NOT EQUAL to 4)

PRINT “Menu:

1. Load Courses
2. Print Course List
3. Print Course
4. Exit”

Switch(choice)

Case 1:

PRINT “Please enter name of file to import course data from”

GET inputFile (string) from user

openFile(inputFile)

break

Case 2:

printAlpha()

Case 3:

PRINT “Please enter course number to print information for:”

GET courseNumber(string) from user

printCourse(courseNumber)

PRINT “exiting program”

RETURN 0

**Vector Pseudocode [Worst case Big 0]**

***//Course Object***

struct Course

Strings: courseNumber, courseTitle

Vector of strings: prereq

int: numPrereq

***//Vector of Courses declared in main***

vector<course> courseDatabase;

***//Open File - O(1) + O(n^3)***

bool openFile (ifstream inputFile)

inputFile = filename [1]

open (inputFile) [1]

IF (openFile failed) [1]

PRINT "File not found" [-]

return error [-]

ELSE

CALL loadFileData(inputFile, fileData)) [n^3]

***//Load file data - O(n^2) + O(1)***

bool loadFileData(inputFile, fileData)

CREATE tempCourse [1]

CREATE tempString (string) [1]

CREATE tempIndex (int) [1]

WHILE inputFile has valid data [n]

GETLINE from inputFile and store in tempDataString [1]

IF tempDataString is not empty [1]

IF(GET data and store in tempString, parse on "," or endline) successful [1]

tempCourse.courseNumber = tempString [1]

IF(GET data and store in tempString, parse on "," or endline) successful [1]

tempCourse.courseTitle = tempString [1]

***//Check that line has at least 2 param.***

ELSE [-]

PRINT "error, file does not have at least two [-]

parameters on each line"

RETURN error [-]

//Loop and get all Prerequisites

WHILE(GET data and store in tempString, parse on "," or endline) successful [n]

tempCourse.prereq.pushback(tempstring) [1]

tempNumPrereq++ [1]

tempCourse. numPrereq = tempNumPrereq [1]

courseList.push\_back(tempCourse.courseNumber) [1]

CALL addCourse(tempCourse) [2]

***//Check for valid Prerequisites - O(n^3) + O(n)***

FOR(int i = 0; i < courseDatabase(size); i++) [n]

FOR(int j; j < courseDatabase.at(i).numPrereq – 1; j++) [n]

CREATE int k = 0

WHILE(courseDatabase.at(i).rereq(j) is NOT EQUAL to [n] courseDatabase.at(k).courseNumber && k < courseDatabase.at(i).numPreq)

k++ [1]

IF(k == courseDatabase.at(i).numPreq) [-]

PRINT “Error, prerequisite not in list, exiting” [-]

RETURN error [-]

inputFile.close [1]

end loadFileData [1]

***//Add course to vector - O(1)***

Bool addCourse(course)

IF (courseCatalog.push\_back(course) ) successful [1]

RETURN true [1]

***//Print Input Course***

Void printCourse(String courseNumber)

FOR(int i = 0; i < courseDatabase.size() - 1; i++)

IF (courseDatabase.at(i).courseNumber EQUALS courseNumber)

PRINT "courseNumber, title"

FOR(int j = 0; j < courseDatabase.at(i).numPrereqs – 1; j++)

PRINT prereq(j)

***//Print Alphanumeric sorted list***

Void printAlpha(courseDatabase)

sortAlpha(courseDatabase)

FOR (int i = 0; i < courseDatabase.size() -1; i++)

printCourse(courseDatabase.at(i)

***//Sort Alphanumerically using the sort method in algorithm.h library***

Void sortAlpha(vector<course> courseDatabase)

Sort(courseDatabase.begin(), courseDatabase.end() [](course a, course b)

RETURN a.courseNumber < b.courseNumber

**Hash Table Pseudocode [Worst case Big 0]**

***//Course Object***

struct Course

Strings: courseNumber, courseTitle

Vector of strings: prereq

int: numPrereq

***//Node Struct***

struct Node

Course course

Unsigned int key

Node \*next

//Declared in MAIN

CREATE vector<string> courseNumbersList

***//Open File – O(n) + (n^2)***

bool openFile (ifstream inputFile)

inputFile = filename [1]

open (inputFile) [1]

IF (openFile failed) [-]

PRINT "File not found" [-]

return error [-]

ELSE

CALL loadFileData(inputFile, fileData)) [n^2]

***//Load file data - O(n^2) + O(n^2)***

bool loadFileData(inputFile, fileData)

CREATE tempString (string) [1]

CREATE tempCourse (Course) [1]

CREATE tempNumPrereq (int) set to 0 [1]

WHILE inputFile has valid data [n]

GETLINE from inputFile and store in tempDataString [1]

IF tempDataString is not empty [1]

IF(GET data and store in tempString, parse on "," or endline) successful [1]

tempCourse.courseNumber = tempString [1]

courseNumbersList.push\_back(tempString); ***//For sorting [1]***

IF(GET data and store in tempString, parse on "," or endline) successful [1]

tempCourse.courseTitle = tempString

***//Check that line has at least 2 param.***

ELSE [-]

PRINT "error, file does not have at least two parameters

on each line" [-]

RETURN error

***//Loop and get all prerequisites***

WHILE(GET data and store in tempString, parse on "," or endline) [n]

tempCourse.prereq.pushback(tempstring) [1]

tempNumPrereq++ [1]

tempCourse. numPrereq = tempNumPrereq [1]

CALL insertCourse(tempCourse) [1]

***//Check for valid Prerequisites – O(n^2)***

FOR(int i = 0; i < Nodes.size() ; i++) [n]

FOR(int j; j < nodes.at(i).numPrereq – 1; j++) [n]

CREATE int k = 0 [1]

WHILE(nodes.at(i).rereq(j) is NOT EQUAL to [n]

nodes.at(k).courseNumber && k < nodes.at(i).numPreq)

k++ [1]

IF(k == nodes.at(i).numPreq) [1]

PRINT “Error, prerequisite not in list, exiting” [1]

RETURN error [1]

inputFile.close

end loadFileData

***//Add new Course – O(1)***

insertCourse(course)

***//Create a key for given course***

Key = course.courseNumber(converted to int) % 100 (tablesize 100) [1]

oldNode = nodes.at(key) [1]

***//If empty add new node***

IF oldNode == null [-]

newNode = new Node(course, key) [-]

CALL insert(key, newNode) [-]

ELSE [-]

***//If no node in this spot add here***

IF oldNode->Key == default value [1]

oldNode->key = key [1]

oldNode->course = course [1]

oldNode->next = null [1]

ELSE

***//Else look for next open spot***

WHILE (oldNode->next is NOT EQUAL to null) [1]

oldNode = oldNode-> next

oldNode->key = key [1]

oldNode->course = course [1]

oldNode->next = null [1]

***//Print Input Course***

Void printCourse(String courseNumber)

***//Create a key for given course***

Key = courseNumber(converted to int) % 100 (tablesize 100)

Node\* node = nodes.at(key)

***//If vector is empty***

IF (node is EQUAL to nullptr OR key is EQUAL to default value)

PRINT “Course not found”

RETURN

***//If vector has 1 entry and it matches***

IF (node != nullptr AND node->key != default value and node->course.courseNumber EQUALS courseNumber)

PRINT " node->course.courseNumber, node->course.title"

FOR(int j = 0; j < node->course.courseNumber.numPrereqs – 1; j++)

PRINT node->course.courseNumber).prereq(j)

RETURN

***//Iterate through table until you find match***

WHILE (node != null)

IF(node->course.courseNumber EQUALS courseNumber)

PRINT " node->course.courseNumber, node->course.title"

FOR(int j = 0; j < node->course.courseNumber.numPrereqs – 1; j++)

PRINT node->course.courseNumber).prereq(j)

RETURN

node = node->next

***//If you get to this point, course was not found return error***

PRINT “Course not found”

RETURN

***//Print Alphanumeric sorted list***

Void printAlpha()

Sort(courseNumbersList.begin(), courseNumbersList.end())

FOR (int I = 0; I < courseNumbersList.size(); i++)

printCourse(courseNumbersList.at(i)

**Tree Pseudocode [Worst case Big 0]**

***//Course Struct***

struct Course

Strings: courseNumber, courseTitle

Vector of strings: prereq

int: numPrereq

***//Node Struct***

struct Node

Bid bid

Left pointer

Right pointer

***//Declared in MAIN***

CREATE vector<string> courseNumbersList

***//Open File***

bool openFile (ifstream inputFile)

inputFile = filename [1]

open (inputFile) [1]

IF (openFile failed) [-]

PRINT "File not found" [-]

return error [-]

ELSE

CALL loadFileData(inputFile, fileData))

***//Load file data – O(n^2)***

bool loadFileData(inputFile, fileData)

CREATE tempString (string) [1]

CREATE tempCourse (Course) [1]

CREATE tempNumPrereq (int) set to 0 [1]

WHILE inputFile has valid data [n]

GETLINE from inputFile and store in tempDataString [1]

IF tempDataString is not empty [1]

IF(GET data and store in tempString, parse on "," or endline) successful [1]

tempCourse.courseNumber = tempString [1]

courseNumbersList.push\_back(tempString); ***//For sorting [1]***

IF(GET data and store in tempString, parse on "," or endline) successful [1]

tempCourse.courseTitle = tempString [1]

***//Check that line has at least 2 param.***

ELSE [-]

PRINT "error, file does not have at least two parameters [-]

on each line"

RETURN error [-]

***//Loop and get all prerquisites***

WHILE(GET data and store in tempString, parse on "," or endline) [n]

tempCourse.prereq.pushback(tempstring) [1]

tempNumPrereq++ [1]

tempCourse. numPrereq = tempNumPrereq [1]

***//Check for valid prereqs – O(n^2)***

IF node is not equal to null [1]

RECURSIVE call (node Left) [n]

FOR (int i ; i < courseNumber.numPrereq ; i++) [n]

tempStr = "rereq" + converted to string (courseNumber.numPrereq)

IF search(courseNumber.tempStr) NOT successful

PRINT "Prerequisite not found, file is invalid." [-]

RETURN error [-]

RECURSIVE call (node Right) [n]

inputFile.close

end loadFileData

***//Add new Course – O(n^2)***

insertNode(Node\* node, Course tempCourse) //insert node, used in loadFileData

IF (currNode courseNumber > tempCourse courseNumber ) [1]

IF there is no left node [1]

node Left = new Node(tempCourse) [1]

ELSE [1]

Recurse down the left node [n]

ELSE [1]

IF there is no right node [1]

node Right = new Node(tempCourse) [n]

ELSE [1]

Recurse down the right node [n]

**//Print out all courses and prerequisites**

void printCourseInformation(Node\* node)

IF node is not equal to null

RECURSIVE call (node Left)

PRINT "courseNumber, title"

FOR(int i = 0; i < prereq.size() – 1; i++)

PRINT prereq(i)

RECURSIVE call (node right)

***//Search for specific node***

Course search(string courseNumber)

WHILE current is not null

IF this node matches

RETURN this node

IF courseNumber is smaller than current node

TRAVERSE left

ELSE courseNumber is larger

TRAVERSE right

RETURN blank courseNumber if no match found

***//Print Input Course***

Void printCourse(String courseNumber)

tempCourse = search(courseNumber)

PRINT "tempCourse.courseNumber, tempCourse.title"

FOR(int j = 0; j < tempCourse.at(i).numPrereqs – 1; j++)

PRINT tempCourse.prereq(j)

***//Print Alphanumeric sorted list***

Void printAlpha()

Sort(courseNumbersList.begin(), courseNumbersList.end())

FOR (int I = 0; I < courseNumbersList.size(); i++)

printCourse(courseNumbersList.at(i)

| **Vector Data Structure Worst Case Runtime** | **Line Cost** | **# Times Executes** | **Total Cost** | **Section Runtime** |
| --- | --- | --- | --- | --- |
| **Open File** | 3 | 1 | 3 | O(1) |
| **Load File Data** | 14 | n | 14n | O(n) |
| **Check For Formatting Errors** | 6 | n^3 | 6(n^3) | O(n^2) |
| **Create Object** | 2 | n | 2n | O(n) |
| **Total Cost** | | | 3 + 14n +6n^3 + 2n |  |
| **Runtime** | | | O(n^2) | O(n^2) |

| **Hash Table Data Structure Worst Case Runtime** | **Line Cost** | **# Times Executes** | **Total Cost** | **Section Runtime** |
| --- | --- | --- | --- | --- |
| **Open File** | 3 | 1 | 3 | O(1) |
| **Load File Data** | 14 | n | 14n | O(n) |
| **Check For Formatting Errors** | 6 | n^3 | 6(n^3) | O(n^2) |
| **Create Object** | 10 | 1 | 10 | O(1) |
| **Total Cost** | | | 3 + 14n +6n^3 + 10 |  |
| **Runtime** | | | O(n^2) | O(n^2) |

| **Tree Data Structure Worst Case Runtime** | **Line Cost** | **# Times Executes** | **Total Cost** | **Section Runtime** |
| --- | --- | --- | --- | --- |
| **Open File** | 3 | 1 | 3 | O(1) |
| **Load File Data** | 14 | n | 14n | O(n) |
| **Check For Formatting Errors** | 6 | n^3 | 6(n^3) | O(n^2) |
| **Create Object** | 2 | n | 2n^2 | O(n^2) |
| **Total Cost** | | | 3 + 14n +6n^3 +2n^2 |  |
| **Runtime** | | | O(n^2) | O(n^2) |

**Evaluation**

In the code I have written, all groups of code have a worst-case runtime of O(n^2). This is because the code to check to see if the prerequisite is in the data structure is very intensive. Because this is the highest value among all parts of the code, the code has that overall value. The Open File code and Load file data are also very similar so those do not change through the different data structures. The interesting part is the amount of time to load file data. This changes for each type of data structure. For Vector, it is O(n) for Hash table, it is O(1) and for Tree it is O(n^2). While the Hash table is slightly better here, there are more factors to consider.

***Vector Data Structure Pros and Cons***

Pros: Easy to understand

Easy to edit

Cons: Long search times

***Hash Table Data Structure Pros and Cons***

Pros: Very fast searches at O(1)

Cons: You must manage collisions

Hash tables become inefficient when there are many collisions

***Tree Data Structure Pros and Cons***

Pros: Fast Insertion and deletion

Fast Searches

Cons: Takes a long time to modify or move elements in the list

My recommendation for ABCU’s system to manage course and their requirements would be the Hash Table data structure. This comes from many points, first of which is searching. I feel that the customer will be searching for courses and what requirements they have more than any other operation. Because a very similar operation is used to insert a new class, this data structure is also the most efficient at loading data. I also believe that the number of classes available can fit within a set number, and therefore the size of the hash table can be static. For example, if the college at the current time has 500 classes and has never had more than 600 different classes available, you could easily use a size of 700 and have very few collisions. Although it will take more work to write this code, it is by far the best for the job.