# Matthew Cochrane Project One

#include fstream

#include sstream

**class** Course {

public:

string courseNumber

string courseName

vector prerequisites

}

**vector readCourses(string courseFile)** {

Open courseFile for reading

Initialize Vector<Course> courses

try {

for line in courseFile {

split line by comma

if (there are less than 2 items in split line) {

throw exception

}

assign courseNumber of ith element with first line item

assign courseName of ith element with second line item

if (third item in line exists) {

add third item to prerequisites of ith element

if (fourth item in line exists) {

add fourth item to prerequisites of ith element

}

}

}

Close file

for course in courses {

for prerequisite in course.prerequisites {

for courseNumber in courses {

if (prequisite has no match) {

throw exception

}

}

}

}

}

catch (string line){

Output error

}

return Vector<Course> courses

}

**// Vector pseudocode**

**int numPrerequisiteCourses(Vector<Course> courses, Course c) {**

**return c.prerequisites.size()**

**}**

**void printSampleSchedule(Vector<Course> courses) {**

**for course in courses {**

**print “courseNumber, “**

**print “courseName, “**

**if (prequisites exists) {**

**print split string of prerequisites**

**}**

**}**

**}**

**void printCourseInformation(Vector<Course> courses, String courseNumber) {**

**for course in courses {**

**if (current iteration courseNumber is a match) {**

**print “courseNumber, “**

**print “courseName” newline**

**print “Prerequisites: “**

**if (prerequisites exists) {**

**print split string of prerequisites**

**}**

**}**

**}**

**}**

// Hashtable pseudocode

**class** HashTable {

private:

Initialize Hashtable<Course> courses

string key

string value

int hash(string key) {

initialize hashValue variable

for (each char in courseNumber) {

Convert char to int

Sum to hashValue

}

return hashValue

}

public:

for course in Vector<Course> courses {

Hashtable key equals hash(courseNumber)

Value equals string of course information

}

}

**int numPrerequisiteCourses(Hashtable<Course> courses) {**

**for key in courses {**

**if (prequisites exists) {**

**Sum to count**

**}**

**return count**

**}**

**void printSampleSchedule(Hashtable<Course> courses) {**

**for key in courses {**

**print value**

**}**

**}**

**void printCourseInformation(Hashtable<Course> courses, String courseNumber) {**

**for key in courses {**

**if (courseNumber matches first 7 chars of value) {**

**print value**

**}**

**}**

**}**

// Tree pseudocode

struct Node {

string course

left node pointer

right node pointer

courseNode(string course) {

this course = course

this left = null

this right = null

}

}

**class** Tree {

public:

Initialize root pointer

root = null

for each course in courses starting at second element {

if (first course) {

Create new courseNode with course as parameter

Assign to root

}

else {

Create new courseNode with course as parameter

Assign to root

Create node pointer nextNode

Assign to root

Create node pointer parent

while(true) {

Set parent node to nextNode

Convert courseNumber to int

if (courseNumberInt less than nextNode course pointer) {

Assign to left nextNode pointer

if(next is null) {

Assign to parent left pointer

return

}

}

else {

Assign to right nextNode pointer

if (next is null) {

Assign to parent right pointer

return

}

}

}

}

}

}

**int numPrerequisiteCourses(Tree<Course> courses) {**

**count = 0**

**while (courseNode not null) {**

**Sum prerequisite size to count**

**Assign courseNode to left pointer**

**Assign courseNode to right pointer**

**}**

**return count**

**}**

**void printSampleSchedule(Tree<Course> courses) {**

**if (courses not null) {**

**output node and newline**

**call printSampleSchedule with left pointer**

**call printSampleSchedule with right pointer**

**}**

**}**

**void printCourseInformation(Tree<Course> courses, String courseNumber) {**

**if (courses not null) {**

**if (first 7 chars of node equals courseNumber) {**

**output node**

**}**

**call printCourseInformation with left pointer**

**call printCourseInformation with right pointer**

**}**

**}**

void abcVector(Vector<Course> courses) {

Apply sort function to courses

for course in courses {

Output course and newline

}

}

void abcHashTable(HashTable<Course> courses) {

Apply sort function to courses

for course in courses {

Output course and newline

}

}

void abcTree(Tree<Course> courses) {

if (at the end of tree) {

return

}

Call abcTree with left node pointer parameter

Output node at end of left branch

Call abcTree with right node pointer parameter

}

**int** main() {

Display menu options

Get input from user

switch(input) {

case load data:

call readCourses(fileName)

create vector<Course> coursesVector

create HashTable<Course> coursesHashTable

create Tree<Course> coursesTree

case print course list:

call abcVector

call abcHashTable

call abcTree

case print course:

get courseNumber from user

call the three printCourseInformation methods

case exit:

goodbye

}

}

**Runtime analyses**

| **Vector code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **return c.prerequisites.size()** | 1 | 1 | 1 |
| **for course in courses** | 1 | n | n |
| **print courseNumber** | 1 | n | n |
| **print courseName** | 1 | n | n |
| **if (prerequisites exists)** | 1 | n | n |
| **print split string of prerequisites** | 1 | n | n |
| **for course in courses** | 1 | n | n |
| **if current courseNumber is a match** | 1 | n | n |
| **print courseNumber** | 1 | n | n |
| **print courseName** | 1 | n | n |
| **if (prerequisites exists)** | 1 | n | n |
| **print “Prerequisites: “** | 1 | n | n |
| **print split string of prerequisites** | 1 | n | n |
| **Total Cost** | | | 12n + 1 |
| **Runtime** | | | O(n) |

|  |  |  |  |
| --- | --- | --- | --- |
| **Hastable code** | **Line cost** | **# Times Executes** | **Total Cost** |
| **for key in courses** | 1 | n | n |
| **if (prerequisites exists)** | 1 | n | n |
| **Sum to count** | 1 | n | n |
| **return count** | 1 | 1 | 1 |
| **for key in courses** | 1 | n | n |
| **print value** | 1 | n | n |
| **for key in courses** | 1 | n | n |
| **if (courseNumber = first 7 chars of value)** | 1 | n | n |
| **print value** | 1 | n | n |
| **Total Cost** | | | 8n + 1 |
| **Runtime** | | | O(n) |

|  |  |  |  |
| --- | --- | --- | --- |
| **Tree code** | **Line cost** | **# Times Executes** | **Total Cost** |
| **Initialize count to zero** | 1 | 1 | 1 |
| **while (courseNode not null)** | 1 | n | n |
| **Sum prerequisite size to count** | 1 | n | n |
| **Assign courseNode to left pointer** | 1 | n | n |
| **Assign courseNode to right pointer** | 1 | n | n |
| **return count** | 1 | 1 | 1 |
| **if (courses not null)** | 1 | n | n |
| **output node and newline** | 1 | n | n |
| **call printSampleSchedule with left pointer** | 1 | n | n |
| **call printSampleSchedule with right pointer** | 1 | n | n |
| **if (courses not null)** | 1 | n | n |
| **if (first 7 chars of node equals courseNumber)** | 1 | 1 | 1 |
| **output node** | 1 |  | 1 |
| **call printCourseInformation with left pointer** | 1 |  | n |
| **call printCourseInformation with right pointer** | 1 |  | n |
| **Total Cost** | | | 11n + 4 |
| **Runtime** | | | O(n) |

**Advantages and Disadvantages of each data structure:**

**Vector**

Pros

* Constant time access to any element
* Memory-efficient
* Built-in library in C++

Cons

* Insertion and deletion at an index is inefficient compared to others
* Bad for large data sets
* Must all be one type

**Hashtable**

Pros

* Constant time for searching, insertion, and deletion
* Great for large data sets
* Keys can be of any type

Cons

* Suboptimal for small data sets
* Two or more keys can map to the same bucket
* Little to no ordering capability

**Tree**

Pros

* O(logn) for a search
* Hierarchical structure
* Natural ordering

Cons

* Can be an imbalanced tree
* Requires a lot of memory
* Relatively complex

**Recommendation**

After reviewing the runtime analysis, my recommendation is to use a hashtable for the course code. The robust hashing system is great for this project. Since each course has a different course number, each generated key is guaranteed to be unique, so hash collision is nothing to worry about. This does require additional code for the hashtable class, but after that it is clearly the best choice for querying this kind of data. The constant time for insertion and deletion means that any updates can be made efficiently.