Devin Perry – CS300 – Project 1

// Vector pseudocode -------------------------------------------------

***Void readFile (Vector<Course> courses) {***

**Open file of relevant course information**

**Read each line in file, and for each**

**Ensure that there are at least two parameters per line**

**For each prerequisite listed, ensure it also has a course number**

**Ensure that there are no errors**

**Depending on error, print a message relating to it**

***}***

***Void loadFileIntoDataSctructure (Vector<Course> courses) {***

**Create an empty vector**

**For each line in the file**

**Parse each line and store the course number, course title, and prerequisites in variables**

**Create a course object with the parsed data**

**Add the course object to the vector with an identity of the course number as the key**

**For each prerequisite in the prerequisite list, add the prerequisite before the course that requires it in the vector**

***}***

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

**total prerequisites = prerequisites of course**

**for each prerequisite p in total prerequisites**

**add prerequisites in vector to total prerequisites**

**print number of total prerequisites**

}

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

**for all courses in vector**

**print course name**

**if course has prerequisites before it in vector**

**for each prerequisite**

**print prerequisites**

}

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

**for all courses in vector**

**if the course = course number**

**print course information**

**for each prerequisite of the course**

**print the prerequisite course number**

}

// Hashtable pseudocode ----------------------------------------------

***Void readFile (HashTable<Course> courses) {***

**Open file of relevant course information**

**Read each line in file, and for each**

**Ensure that there are at least two parameters per line**

**For each prerequisite listed, ensure it also has a course number**

**Ensure that there are no errors**

**Depending on error, print a message relating to it**

**}**

***Void loadFileIntoDataSctructure (HashTable<Course> courses) {***

**Create an empty hashtable**

**For each line in the file**

**Parse each line and store the course number, course title, and prerequisites in variables**

**Create a course object with the parsed data**

**Add the course object to the hashtable with an identity of the course number as the key**

**For each prerequisite in the prerequisite list, correspond the key to the other course key that needs it in the hashtable**

**}**

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

**total prerequisites = right side of Hashtable**

**for each prerequisite in total prerequisites**

**add prerequisite in hashtable to total prerequisites**

**print number of total prerequisites**

}

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

**for all courses in hashtable**

**print course name**

**if value has paired prerequisites**

**for each prerequisite**

**print prerequisites**

}

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

**for all courses in hashtable**

**if the course = course number**

**print course information**

**for each prerequisite of the course**

**print the prerequisite course number**

}

// Tree pseudocode ---------------------------------------------------

***Void readFile (Tree<Course> courses) {***

**Open file of relevant course information**

**Read each line in file, and for each**

**Ensure that there are at least two parameters per line**

**For each prerequisite listed, ensure it also has a course number**

**Ensure that there are no errors**

**Depending on error, print a message relating to it**

**}**

***Void loadFileIntoDataSctructure (Tree<Course> courses) {***

**Create an empty binary tree**

**For each line in the file**

**Parse each line and store the course number, course title, and prerequisites in variables**

**Create a course object with the parsed data**

**Add the course object to the binary tree with an identity of the course number as the key**

**For each prerequisite in the prerequisite list, add the prerequisite as a child node to the course object in the binary tree**

**}**

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

**total prerequisites = left and right children of course**

**for each prerequisite in total prerequisites**

**add left and right nodes of total prerequisites**

**print number of total prerequisites**

}

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

**for all courses in tree**

**print course name**

**if course has left child node**

**print left node as prerequisite**

**if course had right child node**

**print right node as prerequisite**

}

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

**For each course object in binary tree**

**if the course = course number**

**Print the course number and course title**

**for each prerequisite of the course**

**Print the prerequisite course number**

}

The menu pseudocode can be applied to each of the above since they all use the same function names besides their specific data structure type (vector, hashtable, tree).

// Menu pseudocode ---------------------------------------------------

**Accept user input for one of the following options**

**Cout “1. Load Data Structure”**

**Cout “2. Print Course List”**

**Cout “3. Print Course”**

**Cout “4. Exit”**

**Depending on selection, each case will do something different**

**Case 1: execute the *readFile() & loadFileIntoDataSctructure()***

**functions from above depending on the type of data**

**structure**

**Case 2: execute Print List of Course Alphanumerically pseudocode**

**below**

**Case 3: execute the *printCourseInformation()* from above**

**If no course is found with given number, print error**

**Case 4: cout “exit” & close the program**

Like the menu pseudocode above, this code can be applied to all the functions if minor substitutions are made.

// Print List of Course Alphanumerically pseudocode ------------------

**Execute the *readFile() & loadFileIntoDataSctructure()*functions from above depending on the type of data structure**

**Extract the course objects from the data structure**

**Read the course number from the course object**

**Organize list of the courses in ascending order with**

**the lowest course numbers first and the highest last.**

**Execute *printCourseInformation()* for the courses in the order they were organized**

## Example Runtime Analysis

*When you are ready to begin analyzing the runtime for the data structures that you have created pseudocode for, use the chart below to support your work. This example is for printing course information when using the vector data structure. As a reminder, this is the same pairing that was bolded in the pseudocode from the first part of this document.*

| **Given Example Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | 1 | n | n |
| **if the course is the same as courseNumber** | 1 | n | n |
| **print out the course information** | 1 | 1 | 1 |
| **for each prerequisite of the course** | 1 | n | n |
| **print the prerequisite course information** | 1 | n | n |
| **Total Cost** | | | 4n + 1 |
| **Runtime** | | | O(n) |

| hashtable’s numPrerequisiteCourses function | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **total prerequisites = right side of Hashtable** | 1 | 1 | 1 |
| **for each prerequisite in total prerequisites** | 1 | n | n |
| **add prerequisite in hashtable to total prerequisites** | 1 | n | n |
| **print number of total prerequisites** | 1 | 1 | 1 |
|  |  |  |  |
| **Total Cost** | | | 2n + 2 |
| **Runtime** | | | O(n) |

**Runtime Analysis**

To compare the three data structures worst case run times, we can look at the chart above and get an idea of how to compare them. We need to find the cost per line of code in each situation and the number of times it will execute. The line cost is usually 1, and the number of times executed is either a defined number like 1, or n where n is the number of times executed. We can see that for the case of a vector and the printing course information, all lines cost 1. The number of times executed are either n or 1. The 1 makes sense because we only want to print the information once. The n’s also make sense because we need to redo those lines for as many courses there are. The cost is also n because we multiply the cost of 1 by n, which is just n. This makes a total cost of 4n + 1. We can use this method to look at the other structures, which will be in my analysis below. I filled out a similar chart for the scenario of the hashtable’s numPrerequisiteCourses function.

**Advantages and Disadvantages**

The three types of data structures have their various pros and cons. I will begin by looking at the advantages of the vector data structure. Elements in the structure can be accessed very quickly because they are stored contiguously, and adding elements to a vector is very quick and efficient. A disadvantage of a vector is that inserting or deleting an element in the middle can be slow. Rescaling a vector can also take lots of resources. As for the hashtable, most of the time accessing information in the table is quick and the size of it can be adjusted dynamically as needed. However, collisions within the table can slow down the whole process and there is not much order of the elements. Finally, the tree is great when it comes to order unlike the hash table. It can also have elements changed quickly whether that means updating, adding, or deleting. Trees do often require more memory than the others, so that can be a concern. It is also worth considering the value of what it costs to run each data structure. Each has the possibility to have conflicts or collisions, but some have “better” outcomes if that does happen. The vector can have a worst case run time of O(1), but if it needs to resize itself to fit the new element it can be O(n). This would result in a potential O(n2). The hash table could have a run time of O(n) because potentially the courses could all match up to the same correspondent. Lastly, a tree could have a potential worst run time of O(log n).

**Recommendation**

After comparing the three data structures, I think the best data structure to use for this situation is the hash table. From a technical perspective it is quick to add and retrieve the elements from within it, and it can be adjusted dynamically for when more courses are added. Collisions are a risk for a hash table, but it has the “best” worst case scenario for run time of O(n). The vector could result in O(n2), and the tree O(log n), so the hash table beats both of them in that case. The mishaps can be reduced by using a good hash function and implementing strategies in the code to mitigate collisions.