Justin Arruda

CS 300 Project One

// Menu loop

while(exit == false) {

print the menu options

switch (input = get next menu valid choice)

case(1):

load file into data structure function

case(2):

print sortCourseList(courseList)

case(3):

input = get course number from user

search for course in course list

printCourseInformation()

case(4):

exit == true:

}

// print out the list of the courses in the Computer Science program in alphanumeric order

sortCourseList(Vector<Course> courseList) {

sort(courseList.begin, courselist.end, courseNumber)

for course in courseList

print courseNumber and courseName

increment to next course

}

Vector Runtime Analysis

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all lines in file** | 1 | n | n |
| **If getline is not empty** | 1 | n | n |
| **Store courseNumber in vector** | 1 | 1 | 1 |
| **Store courseName in vector** | 1 | 1 | 1 |
| **If stringCount > 2** | 1 | n | n |
| **For all prerequisites** | 1 | n | n |
| **Add prerequisite to vector<course>** | 1 | 1 | 1 |
| **Total Cost** | | | 4n + 3 |
| **Runtime** | | | O(n) |

Hash Table Runtime Analysis

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all lines in file** | 1 | n | n |
| **if getline is not empty** | 1 | n | n |
| **Store courseNumber in node** | 1 | 1 | 1 |
| **Store courseName in node** | 1 | 1 | 1 |
| **If stringCount > 2** | 1 | n | n |
| **for each prerequisite of the course** | 1 | n | n |
| **Add prerequisite to node** | 1 | 1 | 1 |
| **Total Cost** | | | 4n + 3 |
| **Runtime** | | | O(n) |

Binary Tree Runtime Analysis

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all lines in file** | 1 | n | n |
| **if getline is not empty** | 1 | n | n |
| **Store courseNumber in node** | 1 | 1 | 1 |
| **Store courseName in node** | 1 | 1 | 1 |
| **If stringCount > 2** | 1 | n | n |
| **for each prerequisite of the course** | 1 | n | n |
| **Add prerequisite to node** | 1 | 1 | 1 |
| **Total Cost** | | | 4n + 3 |
| **Runtime** | | | O(n) |

When evaluating the runtime efficiency of each different type of storage system, we can notice the construction of each data structure is almost identical. The key differences in the data structures will be when searching for specific courses within the course list, or when insert/delete functions are called. An advantage of the vector example is that it was very straightforward and simple to set up and access. This comes with the disadvantage as not having any real performance benefits compared to the other methods and will have diminishing performance in larger data sets. Compare this with the hash table data structure and it can be looked at almost as a middle ground between the vector example and the binary tree example. Slightly more complex than the vector structure but easier to implement than the binary tree. The binary tree example is slightly harder to implement than the hash table but has beneficial performance capabilities. When searching the tree, it has the best performance out of all the examples, which for our scenario makes the most sense. The only downside is that the performance required to modify the tree and delete/insert new nodes is more expensive than the hash table. My recommendation for which data structure code to implement would be the binary tree, seeing as this program would be used by advisors looking up course information, a program with a quick searching and accessing design would be the most beneficial. Under the right conditions the binary tree can achieve O(log N) runtime efficiency when searching the tree. Since that is primarily what the program will be used for, this will be an excellent choice that will provide greater benefits the larger the database of courses becomes.