Gabriel Romero

Project 1

Runtime Analysis, Pros and Cons and Recommendation

Runtime Analysis for Loading of file:

The program prompts the user for a file name then passes the file name to the openCSV function. Within that function there is an open validation check, a while loop, several if validation checks with one nested while loop and vector push back. The validations break each line apart using the ‘,’ delimiter and stores the string within a token. The token is then grabbed and stored into a temp value to be validated and then moved into the object form. There is one last validation followed by creating a new struct variable and adding the data from each line. The struct is then pushed into the vector<struct> (coursesList) and that is the end of the function.

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **IF fileName open failed** | 1 | n | n |
| **WHILE fileName has data** | 1 | n | n |
| **IF workingData not empty** | 1 | n | n |
| **IF data exists between ','** | 1 | n | n |
| **IF data exists between ','** | 1 | n | n |
| **While remaining data exists between ',' on line** | 1 | n | n |
| **IF tempRow.couresNumber && tempRow.courseDescription not empty** | 1 | n | n |
| **Total Cost** | | | 7n |
| **Runtime** | | | O(n) |

Runtime Analysis for Validating Prerequisites:

Validating prerequisites makes use of a temporary vector built out of successfully passed course numbers. It makes use of the find function to search the valid course list for existing prereqs. If it is found it continues on to the next prereq in the list if empty nothing is done. If however the course is not found, it sends and error message and removes the offending course from the vector list of approved validated courses.

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for( int i = 0; i < courseInfo.prereqs.length; i++)** | 1 | n | n |
| **iterate = find(validCourseList.begin(), validCourseList.end(), courseInfo.prereqs[i])** | 1 | 1 | 1 |
| **IF (iterate != validCourseList.end())** | 1 | n | n |
| **ELSE** | 1 | n | n |
| **coursesList.erase(position of invalid course object in vector)** | 1 | n | n |
| **Total Cost** | | | 5n + 1 |
| **Runtime** | | | O(n) |

Vector Pros and Cons:

One of the biggest draws of using a vector is the simplicity of implementing one. It can store a variety of data types within as a secure list but that is also its draw back. The simplicity does not allow for a more complex traversals and requires deeper algorithms to fully take advantage of sorting methods and searching. Inserting into a vector can become and issue as well as it will have to move n indices around where n is the number of indices needed for insert.

Hashtable Pros and Cons:

The implementation of a hashtable can be a bit cumbersome if not used to generating the data structure. I think this to be somewhat of a con. Although, the key value setup for hashtables can be invaluable when dealing with searching, inserts and deleting. They have a runtime of O(1) when compared to BST (Binary search trees) of O(Logn). Sorting in hashtables can be cumbersome if not done with the proper setup as well. It would be best to have the data sorted beforehand, but then you must keep in mind deletions and insertions where the data can get manipulated and out of order. A good work around would be using the values of the hash tables and sorting them. Like vectors table size also plays a role in the runtimes and efficiencies of hashtables. The larger the table the larger the run time you can expect, with certain methods being used. There is also the possibility of data collisions which require other methods to navigate around. Overall Hashtables are easier on your memory and have their uses.

Binary Search Tree Pros and Cons:

Binary Search trees are a bit less complex to implement from scratch. They offer easy sorting methods using inorder traversal. This is due to the nature of the structure itself. If the BST is self-balanced, we are looking at a solid runtime across the methods of 0(Logn) so you will know what to expect regardless of tour data size. This could be great for bigger projects but for smaller simpler ones, this could be unacceptable. For a BST there is no need to know the data size you before adding data. This is great if you want to jump into just storing your data without knowing the limit of what is coming in.

Recommendation:

This is a tough one between hash table and BST. I do not think a simple vector would work here all that well due to the expanding nature of the data. Who knows if we would end up with more and more classes as time progressed? Even so, we would have a slower runtime due to the way vectors are traversed for searching algorithms and sorting ones. I would have to go with BST on this one especially if we can maintain a balanced tree. Even if not, it allows for our number of classes to grow without concern for changing the table size like we would have to do in a hash table. The search and sort are basically built into the data structure since it is built around being sorted. This makes it handy for when students are constantly looking up courses and prereqs. The reason I see hash tables also being an issue would be the constant insertion of courses and deletion. Although the data won’t be constantly shifting every day, there will be additions and removals at some point, and who knows to the extent of how many. I believe having a system in which we can maintain a consistent run time regardless of the amount of data we pump into it, would be extremely beneficial to a growing project.