Zach Gutman

CS-300-10324

Project 1 – Runtime analysis chart findings

Runtime analysis:

|  |  |  |  |
| --- | --- | --- | --- |
| **Vector** | **Line Cost** | **# Times Executed** | **Total Cost** |
| **Create Vector** | 1 | 1 | 1 |
| **Lines in file** | 1 | n | n |
| **Create Vector course items** | 1 | n | N |
| **- While prereq exists** | 1 | n | n |
| **- Append prereq** | 1 | n | n |
| **Pushback course items** | 1 | N | N |
| **Total Cost:** | | | **5n+1** |
| **Runtime:** | | | **O(n)** |

|  |  |  |  |
| --- | --- | --- | --- |
| **Hashtable** | **Line Cost** | **# times Executed** | **Total Cost** |
| **Create Hash Table** | 1 | 1 | 1 |
| **Insert Method** | 0 | 0 | 0 |
| **- Create key for course** | 1 | n | n |
| **- No entry key found** | 1 | n | n |
| **- Assign node to key** | 1 | n | n |
| **- ELSE** | 1 | n | n |
| **- assign old node key to UNIT\_MAX, Set to key,  set old node to course and old node to next to null pointer** | 4 | n | 4n |
| **- ELSE** | 1 | n | n |
| **- Find next open node** | 1 | n | n |
| **- add a new newNode to end** | 1 | n | n |
| **- for each new line in file** | 1 | n | n |
| **- Create vector course item** | 1 | n | n |
| **- while prereq exists** | 1 | n | n |
| **- append prereq** | 1 | n | n |
| **insert course item** | 1 | n | n |
| **total cost:** | | | 16n+1 |
| **runtime:** | | | O(n) |

|  |  |  |  |
| --- | --- | --- | --- |
| **Tree** | **Line Cost** | **# times executed** | **total cost** |
| **add node method** | 0 | 0 | 0 |
| **- if root is null, add root** | 1 | 1 | 1 |
| **- if node is less than root  then add to left** | 1 | n | n |
| **- if no left node** | 1 | n | n |
| **- this node becomes left** | 1 | n | n |
| **- if node is greater than root  add to right** | 1 | n | n |
| **- for each new line in file** | 1 | n | n |
| **- Create vector course item** | 1 | n | n |
| **- while prereq exists** | 1 | n | n |
| **- append prereq** | 1 | n | n |
| **insert course item** | 1 | n | n |
| total cost | | | 11n+2 |
| runtime | | | O(n) |

Advantages Analysis:

All 3 options have some strong points and some weaknesses, this is why after all 3 different methods exist in the first place. Using An append method I found to be very fast and effective, although sorting this data later was very slow.

The hash table should be the fastest of the 3 just based on our tables above, although the hash table has to be quite large in order to prevent collisions. The only thing is that time and memory are not infinite resources for us. This means that in theory it should be the best, but in practice that may not be completely true. This could be a strong option considering that has tables tend to be simple. For our uses I don’t think the Hash table is the best fit- simply because we won’t be making many edits to our tables and searching shouldn’t be the primary concern when it comes to our academic program. We could assign courses certain codes (like we do for SNHU) like CS300, but then we would also need to create a list that the table would reference everytime. Slowing it down further potentially.

The binary tree I think could be the most consistent of the tables with depending on data is being read. Although with the tree, if data is sorted and then loaded this could slow down the tree by a lot potentially. If the binary tree’s data is loaded while unsorted it could be very fast. For our uses though I think this would be unlikely. As students would want to check off classes that they completed or classes they are still required to complete. It would make logical sense for us to show students courses in order of completion too. Similar to the hash table we could assign courses a level like CS**300** this would make it easy for students to find specific courses.

Finally our vector sorting. This option could work for us nicely I think. The vector would allow us to store our data sequentially quickly and easily. The vector also can be quite agile when it comes to random access. The issue with the vector though would be that searching for 1 class out of say 30 could be costly and slow. If students just wanted to see the entire catalog of classes and then go through the classes sequentially this would be a very strong option for us.

Ultimately all these methods could work for us, really what we would need to know when designing this system would be the intended user experience. I think if we were to mimic how SNHU does their sorting system then I would personally recommend we go with a Binary tree table. Assigning the courses a code like 300 would sort them into groups and let students intuitively understand that those classes come after the 100’s and 200’s. Secondly if we assume that most students are going to use the data infrequently then it would be alright if our tree is a little slow because traffic will usually be quite low. Finally it would give students the biggest picture for their course completion, thus giving them the most information possible.