Kevin Raddatz

CS300

Project 1 Runtime Analysis

06/13/2021

//Vector Data Structure

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| FOR each line in the file: | 1 | n | n |
| Create New Vector of Type String | 1 | n | n |
| Add Course Number as String to vector (Vector.Pushback(courseNumber)) | 1 | n | n |
| Add Course Name to vector as string | 1 | n | n |
| Add N prerequisites to vector (Number and name) | 1 | n | n |
| Take User Input as String | 1 | 1 | 1 |
| FOR all Courses: | 1 | n | n |
| If course is equal to input | 1 | n | n |
| Put Course information to output | 1 | n | n |
| Program should return | 1 | n | n |
| Else increment counter | 1 | n | n |
| IF Counter is equal to length of all course plus one | 1 | n | n |
| Put “Course Not Found” to output | 1 | n | n |
| Program should return | 1 | n | n |
| **Total Cost** | | | 13n + 1 |
| **Runtime** | | | O(13n) |

For the vector data structure, there is a few benefits and drawbacks of using this type of a data structure. First, a benefit of using this type of data structure is that it is quite simplistic and easy to understand. A major drawback, however, is that the code looks as if it’d end up quite long winded and inefficient for the purpose.

//Hash Table Data Structure

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Take Each Line as input | 1 | n | n |
| For each line in File | 1 | n | n |
| create new course object | 1 | n | n |
| Determine initial bucket: key % 10 | 1 | n | n |
| While the number of bucketsProbed is less than N | 1 | n | n |
| If the bucket is empty: | 1 | n | n |
| Bucket = course | 1 | n | n |
| Function should return true | 1 | n | n |
| Bucket = (bucket + 1) % N | 1 | n | n |
| Increment bucketsProbed variable | 1 | n | n |
| **Total Cost** | | | 10n |
| **Runtime** | | | O(10n) |

For the hash table data structure, this data structure is implemented more efficiently than the vector data structure. This is a major benefit of this data structure. The hash table structure is a bit confusing, both conceptually and coding wise, so this would be the main drawback of using this type of a data structure.

//Tree Data Structure

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Define Node Structure | 1 | 1 | 1 |
| Define root = NULL | 1 | 1 | 1 |
| Define left and right nodes =NULL | 2 | 1 | 1 |
| Take input of variable | 1 | n | n |
| IF root is null | 1 | n | n |
| Input is equal to root | 1 | n | n |
| ELSE Create new node | 1 | n | n |
| New Node stores user input. | 1 | n | n |
| Root->left new node //create new node with user input | 1 | n | n |
| Root-> right new node | 1 | n | n |
| **Total Cost** | | | 7n + 3 |
| **Runtime** | | | O(7n) |

For the tree data structure, this topic can be a bit confusing conceptually. However, of the three options available, it is likely the easiest to write code for as well as being more efficient than other available data structures.

**Recommendation:** The recommendation for the ABCU project is to use a tree data structure to store course and prerequisite information. From the runtime analysis, the tree data structure will run most efficiently and be the simplest way to implement the desired functionalities for this project.