Project 1 Evaluation

# Vector

## Runtime Analysis

### Loading data into Structure

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executes | Total Cost |
| for all courses | 1 | n | n |
| Add Prerequisites to course | 1 | n | n |
| Validate prerequisites | 1 | n | n |
| Total Cost | | | 3n |
| RunTime | | | O(n) |

### Searching/Accessing course

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executes | Total Cost |
| Best Case: Course is at index 0 | 1 | 1 | 1 |
| Worst Case: Course not found or last index | 1 | n | n |
| Total Cost | | | n |
| RunTime | | | O(n) |

### Printing the Courses

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executes | Total Cost |
| for all courses | 1 | n | n |
| if course is 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) |

## Discussion

The vector data structure is a great data structure when managing small quantities of items. They require less boilerplate and are easier to reason with. However as a vector scales, searching, sorting, and removing items becomes increasingly expensive. In this example, we can see that as the list of courses grows, searching the vector will scale proportionally. If we were to remove items earlier in the vector, this becomes computationally expensive as each entry will need to be moved to a new index.

# Tree

## Runtime Analysis

### Loading data into Structure

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executes | Total Cost |
| for all courses | 1 | n | n |
| Add Prerequisites to course | 1 | n | n |
| Validate prerequisites | 1 | n | n |
| Total Cost | | | 3n |
| RunTime | | | O(n) |

### Searching/Accessing course

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executes | Total Cost |
| Best Case: Course is at root | 1 | 1 | 1 |
| Worst Case: Course not found or on leaf | 1 | LogN | LognN |
| Total Cost | | | LogN |
| RunTime | | | O(LogN) |

### Printing the Courses

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executes | Total Cost |
| for all courses | 1 | n | n |
| if course is 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) |

## Discussion

A tree is a fantastic data structure for sorted data. This is because as the set of data scales, operations like searching and sorting scale at a logarithmic rate. This is due the branching of higher and lower values allowing algorithms to not need to check every node within the tree. Instead, each check in a tree traversal eliminates entries for the branches it does not traverse.

While performant, a tree can become difficult for developer to reason with. In situations where there is deletion or insertion, there can be a lot of extra data movement when reconnecting nodes. A tree is most efficient when it is balanced and contains data that is not repetitive . In situations where there is repetitive data, traversing a tree becomes similar to a linked list or an array and it loses it’s advantage.

# Hash Table

### Loading data into Structure

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executes | Total Cost |
| for all courses | 1 | n | n |
| Add Prerequisites to course | 1 | n | n |
| Validate prerequisites | 1 | n | n |
| Total Cost | | | 3n |
| RunTime | | | O(n) |

### Searching/Accessing course

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executes | Total Cost |
| Best Case: Course is found | 1 | 1 | 1 |
| Worst Case: Course not found or last index | 1 | 1 | 1 |
| Total Cost | | | 1 |
| RunTime | | | O(1) |

### Printing the Courses

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executes | Total Cost |
| for all courses | 1 | n | n |
| if course is 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) |

## Discussion

A hash table is a phenomenal data structure for searching. As a set of data scales, a hash table will not require any additional executions to fetch items from the structure. This makes it a powerful tool for relational data because of the lookup power. However, in situations where adding items causes a large number of collisions, a hash table can become more expensive as it will need to iterate through the items within its collection until there is not a collision.

# Assessment

The data structure that I recommend for this project is hash table. While each data structure will perform similarly when loading data from the file, a hash table will likely outperform the rest due to the fact that each course will have a unique courseID which will be used to create the key. Because of this, the number of collisions should be nonexistent or minimal at best. The additional benefit of the hash table will come from searching courses for instances such as printing or updating. It will take a single execution versus needing to iterate through the collection at O(n) time like the vector or O(logN) like the tree.