Alex Akers

6/13/2025

CS-300

In my opinion, a vector is the easiest to understand and set up. It stores courses in the order they’re added, which can be helpful when working with lists. However, when you need to search for a course or check its prerequisites, a vector becomes slow because it has to look through each course one by one. This means that if you have a lot of courses, the program can slow down quite a bit. The time it takes increases quickly, especially when validating prerequisites, because you have to use nested loops. This gives the vector a worst-case runtime of O(n²), which I feel isn’t very efficient for large files.

A hash table, on the other hand, is much faster. It allows you to quickly find a course using its number without looping through the whole list. This makes inserting data and checking prerequisites much quicker, especially when dealing with a lot of courses. Most of the time, the runtime is just O(n), which is a big improvement over the vector. One downside is that the courses aren’t stored in any particular order, so if you want to print them in alphabetical order, you’ll need to sort them separately. But even with that, the speed of lookups makes it a very good option.

A binary search tree is also a solid choice, especially because it keeps courses in sorted order by default. This makes it easy to print out the list in order using in-order traversal. If the tree is balanced, search and insert times are fairly quick, around O(log n). But if the tree becomes unbalanced (for example, if the data is already sorted), those times can slow down to O(n). The binary search tree also takes more effort to code, especially when dealing with edge cases or deleting nodes.

Overall, I recommend using the hash table for this project, though the binary tree could also be used with similar performance. These two provide the fastest options for loading data and checking prerequisites, which are the most important tasks in this program. Even though the hash table doesn’t store courses in order, that’s easy to fix by copying the data to a list and sorting it when needed. In my opinion, the hash table keeps the program fast and responsive—especially when working with larger files—and offers the best balance of performance and simplicity.

**Runtime Analysis Chart**

# Vector Runtime Analysis

|  |  |  |  |
| --- | --- | --- | --- |
| **Line Description** | **Cost per Line** | **Number of Executions** | **Total Cost** |
| Open file and check for errors | 1 | 1 | O(1) |
| Loop through each line in the file | 1 | n | O(n) |
| Split line into tokens | 1 | n | O(n) |
| Create course object and set properties | 1 | n | O(n) |
| Loop through prerequisites for each course | 1 | n | O(n) |
| Inner loop to validate prerequisites | 1 | n (nested) | O(n²) |
| Insert course into vector | 1 | n | O(n) |
| Total Runtime |  |  | O(n²) |

# Hash Table Runtime Analysis

|  |  |  |  |
| --- | --- | --- | --- |
| Line Description | Cost per Line | Number of Executions | Total Cost |
| Open file and check for errors | 1 | 1 | O(1) |
| Loop through each line in the file | 1 | n | O(n) |
| Split line into tokens | 1 | n | O(n) |
| Create course object and set properties | 1 | n | O(n) |
| Loop through prerequisites for each course | 1 | n | O(n) |
| Validate prerequisites | 1 | n \* m | O(n \* m) |
| Insert course into table | 1 | n | O(n) |
| Total Runtime |  |  | O(n) |

# Binary Search Tree Runtime Analysis

|  |  |  |  |
| --- | --- | --- | --- |
| Line Description | Cost per Line | Number of Executions | Total Cost |
| Open file and check for errors | 1 | 1 | O(1) |
| Loop through each line in the file | 1 | n | O(n) |
| Split line into tokens | 1 | n | O(n) |
| Create course object and set properties | 1 | n | O(n) |
| Loop through prerequisites for each course | 1 | n | O(n) |
| Search for prerequisite in BST | log n | n | O(n log n) |
| Insert course into BST | log n | n | O(n log n) |
| Total Runtime |  |  | O(n log n) |