Runtime

Key is on bottom

Vector Data Structure

| **Code** | **Line Cost** | **# Times Executed** | **Total Cost** |
| --- | --- | --- | --- |
| Open File | O(1) | 1 | O(1) |
| Initialize Initial Variables | O(1) | 1 | O(1) |
| Loop | O(n) | n | O(n) |
| - Parse File | O(1) | n | O(n) |
| - Validate Prerequisites are Met | O(m) | n | O(n \* m) |
| - Create Course/Append to Vector | O(1) | n | O(n) |
| Close File | O(1) | 1 | O(1) |
| Main Body Print Function | O(k) | 1 | O(k) |
| - Loop Course Object in Vector | O(1) | k | O(k) |
| - Validate that Code Matches | O(1) | k | O(k) |
| - Print Course Information | O(1) | k | O(k) |
| - Course Not Found | O(1) | 1 | O(1) |

The most variable step of this code is within the loop. More specifically, the line that validates the prerequisites requires the introduction of a variable to represent the amount of them in a particular course. (m) Overall, a Vector structure’s strength is its simplicity.

Runtime complexity is O(n\*m)

Hashtable Data Structure

| **Code** | **Line Cost** | **# Times Executed** | **Total Cost** |
| --- | --- | --- | --- |
| Load Course | O(1) | 1 | O(1) |
| Open File | O(1) | 1 | O(1) |
| Load File and Create Hash Table | O(1) | 1 | O(1) |
| Try | O(1) | 1 | O(1) |
| Catch | O(1) | 1 | O(1) |
| Close File | O(1) | 1 | O(1) |
| Main Body Loop | O(n) | n | O(n) |
| - Validate Format for Parsing | O(1) | n | O(n) |
| - Parse Course Data | O(1) | n | O(n) |
| - Validate Prerequisites | O(m) | n | O(n \* m) |
| - Append Course Data to Hash Table | O(1) | n | O(n) |
| Print(PrintInfo) | O(1) | 1 | O(1) |
| Tokenize ValidFormat Data | O(1) | n | O(n) |
| Tokenize ParseCourse for Course Objects | O(1) | n | O(n) |
| Verify CourseData Prerequisites Match with Hash Table | O(1) | n | O(n) |
| Print Course Information and Prerequisites | O(m) | n | O(n \* m) |

This data structure also has the same validation step, which requires a new variable. The runtime complexity remains the same as the Vector structure.

Runtime complexity is O(n\*m)

Tree Data Structure

| **Code** | **Line Cost** | **# Times Executed** | **Total Cost** |
| --- | --- | --- | --- |
| LoadData(fileName) | O(1) | 1 | O(1) |
| Try | O(1) | 1 | O(1) |
| Catch | O(1) | 1 | O(1) |
| Open file with name "fileName" for reading | O(1) | 1 | O(1) |
| For each line in the file | O(n) | n | O(n) |
| Call parseLine(line) | O(1) | n | O(n) |
| ParseLine(line) | O(1) | n | O(n) |
| If tokens < 2: Print "Format Error" | O(1) | n | O(n) |
| If tokens >= 2: courseNum = tokens[0]... | O(1) | n | O(n) |
| For each preReq in prerequisites | O(m) | n | O(n \* m) |
| If preReq does not exist in dictionary: | O(1) | n | O(n) |
| Create a new Course object | O(1) | n | O(n) |
| Set courseNum, title attributes | O(1) | n | O(n) |
| Add preReq to the Course object's list | O(1) | n \* m | O(n \* m) |
| For each course in the treeDataStruct | O(k) | k | O(k) |
| Print (courseNum, title) | O(1) | k | O(k) |
| For each preReq in course's prerequisites | O(m) | k \* m | O(k \* m) |
| Print (preReq) | O(1) | k \* m | O(k \* m) |

By far the most variable of all the structures, the Tree uses several loops. One is for each prerequisite (m) and another iterates through each course, which is now a new variable (k).

Runtime complexity is O(n\*m + k\*m)

KEY:

n = number of lines in file

k = number of courses in structure

m = number of prerequisites in course

Analysis

Hashtable and Vector are simple in design, this is good under the consideration of calculating complexity and boasting generally faster runtimes, but the simplicity of these structures actually are a detriment to all but their speed. The most complicated to calculate for, the Tree data structure, is broadly the more efficient structure for managing all of the different variables that the program calls for.

In other words, despite the Tree having a much more complex and variable runtime than its comparisons, it is structured to be more capable and adaptable, rather than structured to be fast. The Tree Data Structure would be my recommendation.