Data and Runtime analyses

Runtime Table for line cost, number of times executed, total cost, and runtime

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** |  | **Total Cost** |
| Vector (for all courses) | 1 | n |  | n |
| Vector (if the course is the same... | 1 | n |  | n |
| Vector (print out the course info...) | 1 | 1 |  | 1 |
| Hashtable (for all courses) | 1 | n |  | n |
| Hashtable (if the course is the same) | 1 | n |  | n |
| Hashtable (print out course info...) | 1 | 1 |  | 1 |
| Tree (for all courses) | 1 | n |  | n |
| Tree (if the course is the same...) | 1 | n |  | n |
| Tree (print out course info...) | 1 | 1 |  | 1 |
| Total cost |  |  |  | 6N + 3 |
| RunTime |  |  |  | 0(n) |
|  |  |  |  |  |
|  |  |  |  |  |

To analyze the worst-case running time (Big O) of the operations involved in reading the file,” ***CourseData.txt***” creating course objects, and populating the data structures, I have to break down the pseudocode and estimate the cost per line of code, considering that there are n courses stored in the data structure:

If we use the example data file provided for the list of courses:

We have 8 lines of data (m = 8) and a maximum of 8 courses (n = 8) since some courses have prerequisites.

evaluating the runtime complexity based on this data shows:

1. **Reading the File:**

Cost: O(m) (8 lines)

Number of executions: 1

1. **Parsing a Line:**

Cost: O(1) per line (assuming constant time for string splitting)

Number of executions: m (8 lines)

1. **Validating Course Number:**

Cost: O(1) per line (assuming constant time for course number validation)

Number of executions: m (8 lines)

1. **Checking if the Course Exists:**

Cost: O(n) per line (assuming linear search in each data structure)

Number of executions: m (8 lines)

1. **Creating a Course Object:**

Cost: O(1) per line (assuming constant time for object creation)

Number of executions: m (8 lines)

1. **Adding a Course Object to Data Structures:**

Cost: O(1) per line (assuming constant time for adding to data structures)

Number of executions: m (8 lines)

1. **Pushing onto the CourseStack:**

Cost: O(2m) (twice for each line)

Number of executions: 2m (16 times)

Calculating the overall runtime complexity:

* The total cost for reading and parsing the file: O(m) + O(1) \* m + O(1) \* m = O(m)
* The total cost for validating course numbers: O(1) \* m = O(m)
* The total cost for checking if the course exists: O(n) \* m = O(8) \* 8 = O(64)
* The total cost for creating and adding course objects to data structures: O(1) \* m = O(m)
* The total cost for pushing onto the CourseStack: O(2m) = O(m)

In the worst case, the dominant factor for runtime complexity is still checking if the course exists in the data structures, which is O(m\*n) in our case: O(64). The other operations have a lower impact on the overall complexity.

Analysis for all 3 different data structures:

1. Using Vectors:

In the provided pseudocode using vectors, analyzing the runtime complexity requires:

1. **Loading Course Data from File:**

Cost: O(m) (assuming there are m lines of course data)

Number of executions: 1 (per file load)

1. **Checking if a Course Exists:**

Cost: O(n) (linear search through the vector of courses)

Number of executions: O(m) (for each course loaded from the file)

1. **Counting Total Number of Prerequisite Courses:**

Cost: O(k) where k is the total number of prerequisite courses (includes nested prerequisites)

Number of executions: O(1) (per course, not dependent on the number of courses)

1. **Retrieving a Course Object by Course Number:**

Cost: O(n) (linear search through the vector of courses)

Number of executions: O(1) (per course retrieval, not dependent on the number of courses)

1. **Printing Course Information:**

Cost: O(1) (constant time per course)

Number of executions: O(1) (per course)

Summarizing the runtime complexity:

* Loading Course Data from File: O(m) for the entire file.
* Checking if a Course Exists: O(n) per course (O(m \* n) in total for m courses).
* Counting Total Number of Prerequisite Courses: O(k) per course, but this is not dependent on the number of courses, so it's O(1) per course in practice.
* Retrieving a Course Object by Course Number: O(n) per course (O(m \* n) in total for m courses).
* Printing Course Information: O(1) per course.

The dominant factor in the runtime complexity for this code is typically the linear search through the vector of courses (checking if a course exists or retrieving a course object), which is O(n) per operation. Therefore, the overall runtime complexity is primarily determined by the number of courses in the vector (n) and the number of lines of course data loaded from the file (m).

In this specific pseudocode, the worst-case complexity is O(m \* n), where m is the number of lines of course data, and n is the number of courses stored in the vector.

1. Using Hash Tables:

Analyzing the runtime complexity of the above code when using a hash table:

1. **Opening and Validating File:**

Cost: O(m) (assuming there are m lines of course data)

Number of executions: 1 (per file validation)

1. **Creating Course Objects and Storing in Hash Table:**

Cost: O(k) where k is the total number of prerequisites (including nested prerequisites)

Number of executions: O(m) (for each course loaded from the file)

1. **Printing Course Information from Hash Table:**

Cost: O(n) where n is the number of unique course numbers in the hash table

Number of executions: O(1) (per course retrieval from the hash table)

**Overall, summary:**

* Opening and Validating File: O(m) for the entire file.
* Creating Course Objects and Storing in Hash Table: O(k) per course (O(m \* k) in total for m( courses).
* In our case m=8 so (O(8\*K) in total.
* Printing Course Information from Hash Table: O(n) for all unique courses.

The dominant factor in the runtime complexity for this code is typically the number of prerequisites (k) because, for each course, we need to iterate through its prerequisites. However, the number of prerequisites doesn't depend on the number of courses in this case. The second factor is the number of unique course numbers (n) stored in the hash table.

Therefore, the overall runtime complexity is primarily determined by the number of lines of course data loaded from the file (m), the number of prerequisites (k), and the number of unique courses (n).

1. Using Trees:

analyzing the runtime complexity of the code when using a tree data structure:

1. **Loading Data into Tree:**

* Cost: O(m \* k) where m is the number of courses (lines) and k is the average number of prerequisites per course.
* Number of executions: O(m) (for each course loaded from the file)

1. **Printing Course Information from Tree:**

* Cost: O(n + k) where n is the total number of courses and k is the average number of prerequisites per course.
* Number of executions: O(1) (per course retrieval from the tree)

In a summary, the runtime complexity:

* Loading Data into Tree: O(m \* k) for the entire file.
* Printing Course Information from Tree: O(n + k) for all courses.

The dominant factor in this runtime complexity for this code is the number of courses (m) and the average number of prerequisites per course (k). The number of prerequisites (k) can significantly affect the loading process as well as the printing process, but the number of courses (m) has a more significant impact on the overall complexity.

Therefore, the overall runtime complexity is primarily determined by the number of lines of course data loaded from the file (m) and the average number of prerequisites per course (k).

1. Comparison:

The choice of data structure depends on the specific requirements and operations we need to perform. Each of the three data structures (Vector, Hashtable, and Tree) has its strengths and weaknesses in terms of runtime complexity (Big O notation) for various operations.

* ***Vector*:**

***Strengths***:

Fast access to elements by index (O(1)).

Efficient memory usage for simple lists.

***Weaknesses***:

Slower insertion and deletion in the middle (O(n)) if elements need to be shifted.

***Recommendation:***

If the program requires frequent retrieval of courses by their index (e.g., course lookup by position in a list), and the number of courses is not expected to be extremely large, a vector can provide good performance.

* ***Hashtable***:

***Strengths:***

Fast retrieval of elements by key (average case O(1)).

Well-suited for cases where we need to quickly find courses based on their course number.

***Weaknesses:***

Hash collisions can lead to performance degradation (worst case O(n)).

Extra memory overhead due to hash table structure.

***Recommendation:***

If our primary use case involves looking up courses by their course number, a hashtable can provide excellent performance. However, we should design our hash function to minimize collisions.

* ***Tree:***

***Strengths***:

Balanced trees provide efficient insertion, deletion, and retrieval (O(log n)).

Useful for scenarios where we need to maintain courses in a sorted order.

***Weaknesses***:

Slightly slower than hashtables for direct lookups (O(log n) vs. O(1)).

Greater memory usage compared to simpler data structures.

***Recommendation:***

If the program requires maintaining courses in a sorted order or performing range queries (e.g., finding all courses within a certain range of course numbers), a balanced tree structure can be a good choice.

Ultimately, the "best" data structure depends on the specific requirements and usage patterns of your course management program. It may even be beneficial to use a combination of these data structures to optimize different operations. For example, you could use a Hashtable for fast course number lookups and a Tree for maintaining a sorted list for reporting purposes.

Consider your program's expected usage patterns, data size, and required operations to make an informed decision about which data structure(s) to use.

Recommendations:

In our case If we have to alphabetically assort the courses list by name, a balanced tree data structure would be the most suitable choice among the three options: Here's why:

*Vector:*

Sorting a vector alphabetically is doable, but it can be slower for large datasets compared to other data structures designed for sorting.

*Hashtable:*

Hash tables prioritize speedy data retrieval by key, not sorting. To alphabetically sort courses, you must extract, sort, and store them elsewhere. This extra step can be less efficient than using a structure designed for sorting initially.

*Tree (Balanced):*

Balanced tree data structures, such as AVL trees, are excellent for maintaining a sorted order. We can insert courses into the tree in alphabetical order, and the tree structure will keep them sorted efficiently. Retrieving a sorted list of courses from a balanced tree has a time complexity of O(n), where "n" is the number of courses. This is significantly better than sorting algorithms applied to a vector.

Therefore, if sorting courses by name alphabetically is a primary requirement, using a balanced tree data structure is the most efficient choice in terms of runtime complexity (O(n)) and overall performance.