Eric Foster

CS300

Module 6

Runtime Analysis

n = number of courses stored data structure

m = average number of data elements per line

k= the average length of strings

**Vector Data Structure:**

Opening File and Reading Lines

|  |  |  |  |
| --- | --- | --- | --- |
| Line | Line Cost | Times Executed | Total Cost |
| File file = open("courses.txt"); | 1 | 1 | 1 |
| List<String> lines = file.readLines(); | 1 | n | n |
| file.close(); | 1 | 1 | 1 |
| Total Cost | | | n+2 |
| RunTime | | | O(n) |

Reading Each Line and Creating Objects

|  |  |  |  |
| --- | --- | --- | --- |
| Line | Line Cost | Times Executed | Total Cost |
| FOR each line in lines { | 1 | n | n |
| String[] data = line.split(","); | m | n | n\*m |
| IF (data.length < 2) { | 1 | n | n |
| print("Error"); | 1 | n | n |
| String courseNumber = data[0].trim(); | k | n | n\*k |
| String title = data[1].trim(); | k | n | n\*k |
| Vector<String> prerequisites = new Vector<String>(); | 1 | n | n |
| IF (data.length > 2) | 1 | n | n |
| FOR (int i = 2; i < data.length; i++) | 1 | n\*m | n\*m |
| prerequisites.add(data[i].trim()); | k | n\*(m-2) | n\*(m-2)\*k |
| Course course = createCourse(courseNumber, title, prerequisites); | 1 | n | n |
| courses.add(course); | 1 | n | n |
| Total Cost | | | 7n + 2nm + (m-2)nk |
| RunTime | | | O(nmk) |

**Hash Table Data Structure:**

Opening File and Reading Lines

|  |  |  |  |
| --- | --- | --- | --- |
| Line | Line Cost | Times Executed | Total Cost |
| File file = open("courses.txt"); | 1 | 1 | 1 |
| List<String> lines = file.readLines(); | 1 | n | n |
| file.close(); | 1 | 1 | 1 |
| Total Cost | | | n+2 |
| RunTime | | | O(n) |

Reading Each Line and Creating Objects

|  |  |  |  |
| --- | --- | --- | --- |
| Line | Line Cost | Times Executed | Total Cost |
| FOR each line in lines { | 1 | n | n |
| String[] data = line.split(","); | m | n | n\*m |
| IF (data.length < 2) { | 1 | n | n |
| print("Error: Line does not have at least course number and title"); | 1 | n | n |
| String courseNumber = data[0].trim(); | k | n | n\*k |
| String title = data[1].trim(); | k | n | n\*k |
| Vector<String> prerequisites = new Vector<String>(); | 1 | n | n |
| IF (data.length > 2) | 1 | n | n |
| FOR (int i = 2; i < data.length; i++) | 1 | n\*m | n\*m |
| prerequisites.add(data[i].trim()); | k | n\*(m-2) | n\*(m-2)\*k |
| Course course = createCourse(courseNumber, title, prerequisites); | 1 | n | n |
| courses.put(courseNumber, course); | 1 | n | n |
| Total Cost | | | 7n + 2nm + (m-2)nk |
| RunTime | | | O(nmk) |

**Binary Tree Data Structure:**

Opening File and Reading Lines

|  |  |  |  |
| --- | --- | --- | --- |
| Line | Line Cost | Times Executed | Total Cost |
| File file = open("courses.txt"); | 1 | 1 | 1 |
| List<String> lines = file.readLines(); | 1 | n | n |
| file.close(); | 1 | 1 | 1 |
| Total Cost | | | n+2 |
| RunTime | | | O(n) |

Reading Each Line and Creating Objects

|  |  |  |  |
| --- | --- | --- | --- |
| Line | Line Cost | Times Executed | Total Cost |
| FOR each line in lines { | 1 | n | n |
| String[] data = line.split(","); | m | n | n\*m |
| IF (data.length < 2) { | 1 | n | n |
| print("Error: Line does not have at least course number and title"); | 1 | n | n |
| String courseNumber = data[0].trim(); | k | n | n\*k |
| String title = data[1].trim(); | k | n | n\*k |
| List<String> prerequisites = new List<String>(); | 1 | n | n |
| IF (data.length > 2) | 1 | n | n |
| FOR (int i = 2; i < data.length; i++) | 1 | n\*m | n\*m |
| prerequisites.add(data[i].trim()); | k | n\*(m-2) | n\*(m-2)\*k |
| Course course = createCourse(courseNumber, title, prerequisites); | 1 | n | n |
| courses.insert(course); | 1 | n | n |
| Total Cost | | | 7n + 2nm + (m-2)nk |
| RunTime | | | O(nmk) |

**Recommendation:**

Over the length of the milestones three data structures were analyzed: vectors, hash tables, and binary trees. The total cost for opening and reading the file is *n* + 2 which is a consistent runtime of O(*n*) across all three data structures. Also consistent across the data structures is the total cost of reading each line and creating objects; 7*n* + 2*nm* + (*m*-2)*nk* and having a runtime of O(*nmk*)., However, each structure has very distinct advantages and disadvantages. For instance, Vectors are relatively simple and for accessing each element and provide effectual sequential access with O(*1*) complexity, but for insertions and searches they incur O(*n*) complexity. The insertions, deletions, and lookups in Hash tables have O(*1*) time complexity, making them very efficient for accessing course data by course number. However, they require more memory and detailed handling of hash functions and collisions. Binary trees, especially the self-balancing ones, provide ordered storage and efficient O(*log n*) complexity for insertions, deletions, and lookups, but they can quickly become complex and degrade to O(*n*) for unbalanced trees.

Considering these factors, I would recommend using a hash table. Hash tables offer the best average-case performance for insertions, deletions, and lookups, aligning with the need for fast and efficient course data retrieval. They handle large datasets well without significant performance degradation, despite the additional complexity of designing an efficient hash function and managing collisions. The scalability and overall efficiency of hash tables make them the most suitable choice for the advisor's requirements, providing a balanced trade-off between complexity and performance.