Runtime Analysis

Victoria Keyser

Vectors provide a simple and efficient way to store elements in contiguous memory locations. They offer constant time access to elements by index, making them suitable for this program to access specific classes randomly. However, vectors incur additional costs when resizing and adding classes to the course. As elements are added beyond the vector's capacity, it needs to reallocate memory and potentially copy existing elements to a new location, increasing the total cost. The insertion and deletion operations at the beginning or middle of the vector require shifting elements, resulting in an increased time in total cost.

Hash tables offer fast and consistent time complexity for searching, inserting, or deleting elements. They achieve this by using a hash function to map keys to indexes in an array, providing constant time average case performance. Additionally, hash tables support dynamic resizing, allowing them to accommodate elements as needed. However, hash tables may encounter collisions, where multiple keys hash to the same index, leading to degraded performance. To mitigate this, I need to change portions of the code and use chaining techniques.

Trees, being a hierarchical data structure, maintain order among elements, making them suitable for this program to store sorted data, i.e., prerequisites to courses. Although they provide efficient operations to insert, delete, and search, trees have a more complex implementation compared to vectors and hash tables. They require maintaining parent-child relationships and balancing mechanisms to ensure optimal performance. Additionally, trees may require more memory due to nodes and pointers. Despite these drawbacks, trees would be a good choice for this program because of the sorted data while maintaining order.

Based on the analysis conducted, including the Big O analysis results and consideration of the total cost in the readDataFromFile function, it appears that all three data structures offer consistent runtime performance. However, the hash table emerged as the most suitable choice for efficiently storing and retrieving course data. Hash tables provide consistent time complexity for searching, inserting, and deleting elements. However, deleting elements is not noted in the instructions from the advisors in this program, which makes it well-suited for dynamic course management systems. Additionally, hash tables offer versatility in data storage, allowing handling a wide variety of data types. This flexibility is favorable where diverse course data, such as course number, title, and prerequisites, is contained. Therefore, based on the analysis and requirements of the program, I recommend using a hash table as the primary data structure.

|  |  |  |  |
| --- | --- | --- | --- |
| **Vector** | | | |
| code | line cost | # executions | total cost |
| function readCourseDataFromFile |  |  |  |
| Try | 1 | 1 | 1 |
| open the file | 1 | 1 | 1 |
| if file is not open | 1 | 1 | 1 |
| print error message | 1 | 1 | 1 |
| return | 1 | 1 | 1 |
| while there are lines to read from the file | 1 | n | n |
| read a line | 1 | n | n |
| split into components: courseNumber, courseTitle, and prerequisites | 1 | n | n |
| if the line is successfully split | 1 | n | n |
| create an empty vector | 1 | n | n |
| while there are prerequisites | 1 | n | n |
| add each prerequisite to the prerequisiteList | 1 | n | n |
| create a new course object with courseNumber, courseTitle, and prerequisites | 1 | n | n |
| add the new course object to the courses vector | 1 | n | n |
| else | 1 | 1 | 1 |
| print error message | 1 | 1 | 1 |
| close the file | 1 | 1 | 1 |
| catch any opening error | 1 | 1 | 1 |
| print error message | 1 | n | n |
|  |  | total cost | 10n + 9 |
|  |  | runtime | O(n) |
| **Tree** | | | |
| code | line cost | # executions | total cost |
| function readCourseDataFromFile |  |  |  |
| try | 1 | 1 | 1 |
| open the file | 1 | 1 | 1 |
| if file is not open | 1 | 1 | 1 |
| print error message | 1 | 1 | 1 |
| return root | 1 | 1 | 1 |
| while there are lines in the file | 1 | n | n |
| read a line | 1 | n | n |
| if line doesn't adhere to expected format | 1 | n | n |
| print error message | 1 | n | n |
| continue | 1 | n | n |
| create a new course object | 1 | n | n |
| parse courseNumber, courseTitle, and prerequisites | 1 | n | n |
| split the line by comma | 1 | n | n |
| courseNumber = first part | 1 | n | n |
| courseTitle = second part | 1 | n | n |
| prerequisites = third part | 1 | n | n |
| for each prerequisite in prerequisitesStr | 1 | n | n |
| add prerequisite to course's prerequisite list | 1 | n | n |
| create a new node with the extracted course | 1 | n | n |
| create a new course node with course as data and left null and right children | 1 | n | n |
| insert new node into the tree | 1 | n | n |
| close the file | 1 | 1 | 1 |
| catch any expection | 1 | 1 | 1 |
| print error message | 1 | n | n |
| return root | 1 | n | n |
|  |  | total cost | 18n + 7 |
|  |  | runtime | O(n) |
| **Hash table** | | | |
| code | line cost | # executions | total cost |
| function readCourseDataFromFile |  |  |  |
| open the file | 1 | 1 | 1 |
| if file is not open | 1 | 1 | 1 |
| print error message | 1 | 1 | 1 |
| return | 1 | 1 | 1 |
| while there are lines in the file | 1 | n | n |
| course course | 1 | n | n |
| split the vector | 1 | n | n |
| parse courseNumber, courseTitle, and prerequisites from the line | 1 | n | n |
| if not valid file format | 1 | n | n |
| print error message | 1 | 1 | 1 |
| skip to the next line | 1 | 1 | 1 |
| create a course object | 1 | n | n |
| store the course object in the hash table | 1 | n | n |
| close the file | 1 | 1 | 1 |
|  |  | total cost | 7n+7 |
|  |  | runtime | O(n) |