Justin Osman

CS 300

Nathan Lebel

10/21/2023

*Runtime Analysis Chart (structure as a whole)*

|  |  |  |  |
| --- | --- | --- | --- |
| **Type of Data Structure** | **Reading file (O)** | **Creation of Course Objects (O)** | **Total (O)** |
| Vector | O(n) | O(1) | O(n) |
| Hash Table | O(n) | O(1) | O(n) |
| Balance BST/ Tree | O(n) | O(log n) | O(n log n) |

*Runtime Analysis for Vector(by line of code)*

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **Cost for each line** | **# of executions** | **Total Cost/Big O** |
| Open the file | 1 | 1 | O(1) |
| For each line in file | 1 | n | O(n) |
| Analyze Line | 1 | n | O(n) |
| Check correct format | 1 | n | O(n) |
| Create course object | 2 | n | O(2n) |
| For i from 2 to length | 1 | k(varies) | O(k) |
| Find course w/# | 2 | k(varies) | O(2k) |
| Check if prereq. Exist | 1 | k(varies) | O(k) |
| Append courses | 1 | n | O(n) |
| Output error message | 1 | 1 | O(1) |
| Output course name/# | 1 | n | O(n) |
| Output Prereqs | 1 | k | O(k) |
| Close file | 1 | 1 | O(1) |
| Total/Worst Case |  |  | O(2n + 2k) = O(n+k) |

*Runtime Analysis for Hash Table(by line of code)*

|  |  |  |  |
| --- | --- | --- | --- |
| **Line of code** | **Cost for each line** | **# of executions** | **Total Cost/Big O** |
| Open file | 1 | 1 | O(1) |
| For each line in file | 1 | n | O(n) |
| Split the lines | 1 | n | O(n) |
| Check format | 1 | n | O(n) |
| Create Course object | 1 | n | O(n) |
| For i from 2 to array | 1 | k(varies) | O(k) |
| Find Course w/# | 1 | k(varies) | O(k) |
| Check for prereqs | 1 | k | O(k) |
| Append courses | 1 | n | O(n) |
| Output error message | 1 | 1 | O(1) |
| Output course name/# | 1 | n | O(n) |
| Output prereqs | 1 | k | O(k) |
| Close file | 1 | 1 | O(1) |
| Total/Worst Case |  |  | O(n) + O(k) |

*Runtime Analysis for Tree(by line of code)*

|  |  |  |  |
| --- | --- | --- | --- |
| **Line of code** | **Cost for each line** | **# of executions** | **Total Cost/Big O** |
| Open file | 1 | 1 | O(1) |
| Loop thru each line | 1 | n | O(n) |
| Read a line | 1 | n | O(n) |
| Split the line | 1 | n | O(n) |
| Create course object | 1 | n | O(n) |
| Insert course into tree | 1 | n | O(n) |
| Close file | 1 | 1 | O(1) |
| Output error message | 1 | 1 | O(1) |
| Output course name/# | 1 | n | O(n) |
| Output prereqs | 1 | k(varies) | O(k) |
| Total/Worst Case |  |  | O(n) + O(k) |

*Analysis*

Vectors are straightforward to use and allow for fast sequential data access with constant-time complexity. Their downside is they suffer from slow insertions, especially in the middle or beginning, which require shifting all subsequent elements, resulting in linear time complexity. Searching for specific elements may involve iterating through the entire vector, leading to linear time complexity. Vectors also allocate consecutive memory, which can be inefficient for active resizing.

Hash tables offer efficient operations such as insertion, search, and deletion with constant-time average-case complexity, making them valuable for quick data retrieval. On the other hand, they can exhibit predictable performance issues in cases where hash collisions become frequent, potentially causing operations to slow down. Effective collision resolution strategies are essential to get rid of these issues. Also, hash tables may consume additional memory due to the necessity of internal structures for managing key-value pairs. Striking a balance between memory utilization and performance is important, particularly when handling large datasets. Despite these considerations, hash tables remain a great choice because of their speedy average-case performance.

Trees maintain data in a sorted order, facilitating ordered data retrieval. Balanced binary search trees, such as AVL or Red-Black trees, guarantee logarithmic height, ensuring efficient O(log n) search, insertions, and deletions. However, implementing and maintaining balanced trees can be tricky. Trees can consume more memory than other structures due to internal nodes, and worst-case scenarios, such as unbalanced trees, can result in O(n) performance.

Based on my analysis I will be using the Hash Table data structure. Hash tables offer constant-time average-case performance for data retrieval, which is crucial for fast lookups. Even though hash collisions can make life difficult, they can be managed effectively with suitable hashing strategies. Hash tables strike a balance between simplicity and efficiency, aligning well with the project's goals. I believe the Hash Table data structure provides the best balance of efficiency, simplicity, and ease of implementation for this project. It offers fast data retrieval and can efficiently handle the lookup requirements while avoiding potential complexities or memory overhead associated with the other structures.