Juan Rodriguez

CS-300

When looking at different ways to manage course data, we explored three main data structures: vectors, hash tables, and binary search trees. Each has its own benefits and drawbacks. Here's a breakdown of each one and why I chose hash tables as the best option.

Vectors are simple to use and understand, which is a big plus. They're great for going through data one by one quickly, making them a good choice for smaller datasets or when you don't need to search for or delete items often. However, vectors could be more efficient when searching or deleting items. To find something, you have to look through each item one by one, which can take a long time if you have a lot of data (O(n) time complexity). Deleting an item is also slow because you have to shift all the other items to fill the gap, which is also O(n). So, vectors might not be the best choice for large datasets.

Hash tables are super efficient for inserting, deleting, and searching data. On average, these operations only take constant time (O(1)), which is really fast. This makes hash tables great for large datasets where you need quick access to your data.

But hash tables have their downsides, too. If many items get stored in the same spot (called a collision), the performance can slow down to O(n). To prevent this, you need a good hash function and ways to handle collisions, like chaining or open addressing. Also, hash tables use more memory than other data structures because of the extra space needed to handle collisions.

Binary search trees (BSTs), when balanced, are efficient for searching, inserting, and deleting items, each taking O(log n) time. This makes them good for keeping data in order and quickly finding items. They also allow you to easily get data in sorted order, which can be useful. However, if the tree becomes unbalanced, these operations can become as slow as O(n). This adds complexity to the implementation.

After analyzing these data structures, I recommend using a hash table. Hash tables provide the fastest average time for inserting, deleting, and searching data. They are especially good for large datasets where quick access is important. Although hash tables can slow down if there are too many collisions, a well-designed hash function and good collision handling can minimize this risk. Overall, hash tables offer the best balance of speed and efficiency for my course management system.

**Cost Analysis for Vector:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code Line** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Open the file | 1 | 1 | 1 |
| Check if the file is successfully opened | 1 | 1 | 1 |
| Read each line from the file | 1 | n | n |
| Split the line by commas | 1 | n | n |
| Check if the line has at least two tokens | 1 | n | n |
| Create a new Course object | 1 | n | n |
| Add any prerequisites to the course (inner loop) | 1 | n \* p | np |
| Add the course to the vector | 1 | n | n |
| Close the file | 1 | 1 | 1 |
| Validate prerequisites (outer loop) | 1 | n | n |
| Validate prerequisites (inner loop) | 1 | n \* p | np |
| Total Cost | - | - | 2n + 2np + 3 |
| **Big O** | - | - | **O(n + np)** |

**Cost Analysis for Hash Table:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code Line** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Open the file | 1 | 1 | 1 |
| Check if the file is successfully opened | 1 | 1 | 1 |
| Read each line from the file | 1 | n | n |
| Split the line by commas | 1 | n | n |
| Check if the line has at least two tokens | 1 | n | n |
| Create a new Course object | 1 | n | n |
| Add any prerequisites to the course (inner loop) | 1 | n \* p | np |
| Insert the course into the hash table | 1 | n | n |
| Close the file | 1 | 1 | 1 |
| Validate prerequisites (outer loop) | 1 | n | n |
| Validate prerequisites (inner loop) | 1 | n \* p | np |
| Total Cost | - | - | 2n + 2np + 3 |
| **Big O** | - | - | **O(n + np)** |

**Cost Analysis for Binary Search Tree:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code Line** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Open the file | 1 | 1 | 1 |
| Check if the file is successfully opened | 1 | 1 | 1 |
| Read each line from the file | 1 | n | n |
| Split the line by commas | 1 | n | n |
| Check if the line has at least two tokens | 1 | n | n |
| Create a new Course object | 1 | n | n |
| Add any prerequisites to the course (inner loop) | 1 | n \* p | np |
| Insert the course into the BST | log(n) | n | nlog(n) |
| Close the file | 1 | 1 | 1 |
| Validate prerequisites (outer loop) | 1 | n | n |
| Validate prerequisites (inner loop) | 1 | n \* p | np |
| Total Cost | - | - | 3n + np + nlog(n) + 3 |
| **Big O** | - | - | **O(nlog(n) + np)** |