Corey Gaspar

4/8/2025

Project One Evaluation

Run Time Analysis

|  |  |  |  |
| --- | --- | --- | --- |
|  | Vector | Hash Table | Binary Tree |
| **Load Data** | O(1) | O(1) – O(N) | O(log N) |
| **Search** | O(n) | O(1) – O(N) | O(log N) – O(N) |
| **Sort/Print** | O(N log N) | O(N) | O(N) |

Each data structure has its pros and cons. For example, it is quick and easy to add data to an unsorted vector but it becomes slower to add data as more and more data is added to it.

Depending on if the hash table is large enough, it can work pretty fast and avoid collisions. However, memory and time are limited and some collisions are bound to happen. This can slow down the hash table data structure to between O(1) and 0(N).

A binary tree is another potential option. Typically, a binary tree works at O(logN). However, if the tree becomes unbalanced for whatever reason, for example, if sorted data is added, it could possibly slow down to O(N).

No matter how you need to access data, choosing the correct data structure is important. For example, if you only add data every so often, it may not be the best idea to optimize. On the other hand, if you plan to add and search through data consistently, a hash table may be more optimal compared to a binary tree, especially if the tree is unbalanced.

A binary tree doesn't need sorting and can be read in order, which saves memory compared to keeping both sorted and unsorted data. In many cases, a binary tree or hash table is a better choice than sorting a vector.

Based on these findings, I recommend that hash tables might be best to use. Hash tables have an average O(1) time complexity for both loading data and searching for data, which makes it efficient when frequently searching and adding data. A hash table is beneficial when handling large datasets and do not require sorting. It is simple to manage collisions with a hash table by resizing as needed.