**Run Time Analysis**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Vector** | **Hash Table** | **Binary Tree** |
| **Loading Data** | O(1) | O(1) – O(N)  *\*depends on if there are collisions* | O(log N) |
| **Search** | O(n) | O(1) – O(N)  *\*depends on if there are collisions* | O(log N) – O(N)  *\*depends on the balance of the tree* |
| **Sort/Print** | O(N log N) *\*using quick sort* | O(N)  *\*assumes the table is created in order* | O(N)  *\*in order traversal* |

**Advantage Analysis**

Each of the three data structures comes with its own set of strengths and weaknesses. Appending data to an unsorted vector is extremely fast; however, sorting that data afterward results in the slowest performance among the three. In theory, a hash table can consistently perform at an average of Θ(1), assuming it’s large enough to avoid all collisions. In practice, though, limitations in time and memory mean that some collisions are inevitable, causing the performance to range between O(1) and O(N).

Binary trees generally offer a more predictable performance, typically around O(log N), though this can degrade to O(N) in cases where the tree becomes significantly unbalanced—such as when data is inserted in already sorted order. Choosing the right data structure depends heavily on how and how often the data will be accessed. For example, if the dataset is only loaded occasionally and not accessed frequently afterward, initial loading speed might be the only consideration. On the other hand, if frequent searches are needed, a well-implemented hash table could outperform a binary tree, especially if the tree is poorly balanced.

Additionally, binary trees don’t require a separate sorting step and can be traversed in order, which might reduce memory usage since there’s no need to store both sorted and unsorted versions of the data. Overall, both the hash table and binary tree are likely to offer better performance than sorting a vector after loading.

**Recommendations**

The expectation is that the data will only be loaded into memory occasionally, rarely printed in full, but frequently searched. Given this usage pattern, a hash table is likely the most suitable choice. However, to ensure performance remains closer to O(1) rather than degrading toward O(N), the hash function and table size must be carefully tuned to minimize collisions.