**CS300 DSA: Project One**

**Analysis**

Buddy Marcey

[buddy.marcey@snhu.edu](mailto:buddy.marcey@snhu.edu)

Southern New Hampshire University

October 8, 2023

**Analysis**

The three data structures for this project each have pros and cons. This analysis will discuss those pros and cons and develop a recommendation for which data structure we should use to meet the requirements of the assignment. Each will be evaluated based on their efficiency with respect to loading the file into the data structure, sorting (and printing), and searching for a specific element.

We will begin with the vector data structure. A big advantage for vectors is that they are by far the easiest to code. Adding elements one at a time from an input file is simple and efficient, O(N); sorting can be done once the list is built. There are several sort methods that can be performed on the unsorted list that have a complexity of O(N log N), including quicksort, merge sort, and heap sort. Once the list is sorted, binary search, with a complexity of O(log N), can be used to very quickly find a match, halving the remaining list with each mismatch. Insertion into an unsorted list is simply appending a new item to the end, but insertion into a sorted list requires more complexity, including a search followed by pointer reassignment for a linked list.

The main advantage of the hash table is the very efficient insertion or search function when looking for a single element, depending on how collisions are handled. In a perfect hash table, those functions will have an efficiency of O(1). More often though, the hash table will not be perfect, so collisions will cause the search to have to continue looking if a previous bucket was not a match; this still only leads to a worst-case scenario of O(N), where it must check every bucket in the table and only finds the answer on the last item. The major disadvantage to a hash table is that one cannot iterate over the table and produce a sorted list. In order to get a sorted list from the hash table, one would have to create a vector of the values, sort that vector, and then iterate over the vector calling each item from the hash table. Creating the vector would have an efficiency of O(N), sorting would have an efficiency of O(N log N), and searching each value back from the table in order to print would have a (worst-case) efficiency of O(N2). This is obviously quite troublesome. Implementation is in the middle, more difficult to code than a simple vector, but not as difficult as a binary search tree, which we will evaluate next.

A big advantage of a binary search tree is the ability to sort the data as it is implemented. Binary insertion has an efficiency of O(log N). Since the list is already sorted, searches are also conducted at an efficiency of O(log N). Insertion works the same, as it finds the appropriate space and then just reassigns some pointers. The biggest downside of the binary search tree is that it requires more precision when writing the algorithm, as it can be considerably more complex to a beginner programmer; most of the functions require recursion to work properly. Also, the data that is being inserted into the structure needs to be sufficiently randomized in order to generate a balanced tree; a sorted input list will make a one-sided tree with the worst cases for manipulation.

**Recommendation**

Each of the data structures has advantages and disadvantages. Vector is the simplest to implement, hash table has the fastest potential search, and binary search trees are sorted on implementation. Given the scope of our project and the requirements of our client, I would recommend the binary search tree. It is slightly faster to build a sorted list than to build an unsorted vector and then sort it. It is not as efficient for searching for a single item as a hash table, but only by a little, and it is more than made up for by being way more efficient at producing a sorted list for print. If we did not have to provide a sorted list for printing the full data, I would instead recommend a hash table.

**Big-O Analysis**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code (Vector)** | **Line cost** | **Number of Times** | **Total** |
| Open file | 1 | 1 | 1 |
| Get line | 1 | n | n |
| Set first token | 1 | n | n |
| Set second token | 1 | n | n |
| Set third token | 1 | n | n |
| Set fourth token | 1 | n | n |
| Construct course | 1 | n | n |
| Add course to vector | 1 | n | n |
| Close data file | 1 | 1 | 1 |
| Return Vector | 1 | 1 | 1 |
|  |  | Total: | 7n + 3 |
|  |  | Efficiency (load): | O(n) |
| Declare null course | 1 | 1 | 1 |
| For each course (compare) | 1 | n | n |
| Return course | 1 | 1 | 1 |
|  |  | Total: | n + 2 |
|  |  | Efficiency (search): | O(n) |
| Declare variables | 1 | 1 | 1 |
| For each course (outer loop) | 1 | n | n |
| For each course (inner loop) | 1 | n\*n | n\*n |
| Compare | 1 | 1 | 1 |
| Swap | 1 | 1 | 1 |
|  |  | Total: | n2 + 3 |
|  |  | Efficiency (insertion sort): | O(n2) |
| For each course (sorted) | 1 | n | n |
| Print course | 1 | 1 | 1 |
|  |  | Total: | n + 1 |
|  |  | Efficiency (print after sort): | O(n) |
|  |  |  |  |
| **Code (Hash Table)** | **Line Cost** | **Number of Times** | **Total** |
| Open file | 1 | 1 | 1 |
| Get line | 1 | n | n |
| Set first token | 1 | n | n |
| Set second token | 1 | n | n |
| Set third token | 1 | n | n |
| Set fourth token | 1 | n | n |
| Construct course | 1 | n | n |
| Call Insert Function | 4-7 | n | (4-7)n |
| Close file | 1 | 1 | 1 |
|  |  | Total: | (10-13)n + 2 |
|  |  | Efficiency (load): | O(n) |
| Calculate key | 1 | 1 | 1 |
| Search for key | 1 | 1 (perfect hash)  n (worst case hash) | 1  n |
|  |  | Total: | 2 (perfect)  n + 2 (worst) |
|  |  | Efficiency (search): | O(1) (best)  O(n) (worst) |
| For each course in hash table | 1 | n | n |
| Add course to vector | 1 | n | n |
| Sort vector (see vector sort above) | 5 | n\*n | 5n\*n |
| For each course in sorted vector | 1 | n | n |
| Print course | 1 | 1 | 1 |
|  |  | Total: | 3n + 5n2 + 1 |
|  |  | Efficiency (sorted print) | O(n2) |
|  |  |  |  |
| **Code (BST)** | **Line Cost** | **Number of Times** | **Total** |
| Open file | 1 | 1 | 1 |
| Get line | 1 | n | n |
| Set first token | 1 | n | n |
| Set second token | 1 | n | n |
| Set third token | 1 | n | n |
| Set fourth token | 1 | n | n |
| Construct course | 1 | n | n |
| Call insert function | 1-5 (recursive) | n log n | (1-5)n log n |
| Close file | 1 | 1 | 1 |
| Return BST | 1 | 1 | 1 |
|  |  | Total: | 6n log n + 3 |
|  |  | Efficiency (load): | O(n log n) |
| Declare variable | 1 | 1 | 1 |
| While (binary search) | 1-3 | log n | (1-3)log n |
| Return course | 1 | 1 | 1 |
|  |  | Total: | (3-5)log n |
|  |  | Efficiency(search): | O(log n) |
| For each course | 1-3 (recursive) | n | n |
| Print course | 1 | 1 | 1 |
|  |  | Total: | n + 1 |
|  |  | Efficiency (print, since BST is already sorted): | O(n) |