Joshua Asari

CS-300-T2624

December 5, 2021

Print Pseudocode / Evaluation

// Vector

void vectorSortPrint(Vector<Course> courses)

{

// Sorts the vector of courses by its courseNumber

sort(courses.begin(), courses.end(),[] (Course a, Course b)

Return a.courseNumber < b.courseNumber;

For each course in courses

Print courseName

}

**//** HashTable

void hashSortPrint(HashTable<Course> courses)

{

Transfer hashtable data into a vector

Run the vector sort print method

}

// Tree

Void treeSortPrint(Tree<Course> courses)

{

Do a InOrder Traversal of tree

Print at each node

}

**Vector**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Vector<Course> courses**  **Vector<string> temp** | **2** | **1** | **1** |
| **Ifstream infile(“filename”)** | **1** | **1** | **1** |
| **while(getline(infile, line))** | **1** | **n** | **n** |
| **Stringstream ss(line)** | **1** | **n** | **n** |
| **while(ss.good())** | **1** | **n** | **n** |
| **String substr;**  **getline(ss, substr, ‘,’)**  **temp.push\_back(substr)** | **3** | **n** | **n** |
| **parseLine(temp,courses) // Method to parse the line to push to the vector** | **1** | **n** | **n** |
|  |  | **Total cost** | **5n + 2** |
|  |  | **Runtime** | **O(n)** |

**HashTable**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **HashTable\* table = new HashTable()**  **Vector<string> temp** | **2** | **1** | **1** |
| **Ifstream infile(“filename”)** | **1** | **1** | **1** |
| **while(getline(infile, line))** | **1** | **n** | **n** |
| **Stringstream ss(line)** | **1** | **n** | **n** |
| **while(ss.good())** | **1** | **n** | **n** |
| **String substr;**  **getline(ss, substr, ‘,’)**  **temp.push\_back(substr)** | **3** | **n** | **n** |
| **table.insert(parseLine(temp)); // Method to parse the line then insert it to the hash table** | **1** | **n** | **n** |
|  |  | **Total cost** | **5n + 2** |
|  |  | **Runtime** | **O(n)** |

**Tree**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Tree\* tree = new Tree()**  **vector<string> temp** | **2** | **1** | **1** |
| **Ifstream infile(“filename”)** | **1** | **1** | **1** |
| **while(getline(infile, line))** | **1** | **n** | **n** |
| **Stringstream ss(line)** | **1** | **n** | **n** |
| **while(ss.good())** | **1** | **n** | **n** |
| **String substr;**  **getline(ss, substr, ‘,’)**  **temp.push\_back(substr)** | **3** | **n** | **n** |
| **tree.insert(parseLine(temp)); // Method to parse the line then insert it to the hash table** | **1** | **n** | **n** |
|  |  | **Total cost** | **5n + 2** |
|  |  | **Runtime** | **O(n)** |

All 3 data structures are implemented in the same way where a new vector / hashtable / tree is created alongside a vector of strings called temp. This vector of strings will hold the information from the file one line at a time. Then the pseudocode will open the file, and while getline(infile, line) is valid, it will stringstream ss(line) to hold a file line. Then the line will be separated by commas and pushed into the temp string vector. After that, the parseLine will be ran, in which it is a method that creates a new Course by using the vector<string> temp that is holding the line information. Since the line information is sorted by courseNumber, courseName, then any prerequisites, it is easy to create a new Course variable with the information saved from the line. After a new Course variable is created, each of the data structures just pushes that new Course variable into their respective data structure.

**Advantages and Disadvantages**

After dealing with all 3 data types, there are advantages and disadvantages for them. For vectors, they are not a fixed structure and can be dynamically added onto. They are also easy to implement. Although, if the data size is too big, traversing gives a constant time of O(n). This is different to a Hashtable as a Hashtable can have an O(1) if each bucket has at most one entry, and the access by key is O(1). Hash Tables work really well in a large data set. Although, the downside of hash tables is that they are hard to implement. Lastly, Trees are easy to implement and do not require much space as they are just nodes with roots with a left and right connection. The downside is that the traversal for trees is the height of the trees, so O(h), but the worst case for trees is O(log n).

Because of the information of the Big O value, the recommended data structure would be to use vectors. This is because they are easy to implement and since the data set is not large, there will not be a huge discrepancy between using a hashtable vs using a vector, although a hashtable would be faster as it will most of the time have a O(1).