|  |  |  |  |
| --- | --- | --- | --- |
| **The worst-case time complexity for reading and creating course objects. Vector** | **Line Cost** | **#Times Executes** | **Total Cost** |
| string openFile(string fileName) | 1 | 1 | 1 |
| fstream myFile | 1 | 1 | 1 |
| vector <Courses>& course | 1 | 1 | 1 |
| Initialize string line, word | 1 | 1 | 1 |
| Initialize int row = 0 | 1 | 1 | 1 |
| Try | 1 | 1 | 1 |
| myFile open (fileName.csv) | 1 | 1 | 1 |
| If (myFile is open) Then | 1 | n | n |
| while (myFile) | 1 | n | n |
| GetLine (myFile, line) | 1 | 1 | 1 |
| stringstream ss(line) | 1 | 1 | 1 |
| while (getline(ss, word, ‘,’)) | n | n | n2 |
| first word in line = new Course() | 1 | 1 | 1 |
| course push back(word) | 1 | n | n |
| If (course size is >= 2 word) Then | 1 | n | n |
| course word [2] = course prerequisite[0] | 1 | n | n |
| course word[3] = course prerequisite[1] | 1 | n | n |
| row += 1 | 1 | n | n |
| For each prerequisite in course | 1 | n | n |
| If (prerequisite == courseNumber) | 1 | n | n |
| return course | 1 | 1 | 1 |
| Else | 1 | 1 | 1 |
| throw error | 1 | 1 | 1 |
| Catch (myFile) | 1 | 1 | 1 |
| Print “Error opening: ” myFile | 1 | 1 | 1 |
| Close myFile | 1 | 1 | 1 |
| **Total Cost** |  |  | n2 + 9n + 16 |
| **Runtime** |  |  | 0(n2) |

|  |  |  |  |
| --- | --- | --- | --- |
| **Worst-case time complexity: opening and reading a file in a hash table** |  |  |  |
| string openFile(string fileName) | 1 | 1 | 1 |
| fstream myFile | 1 | 1 | 1 |
| Initialize string line | 1 | 1 | 1 |
| Try | 1 | 1 | 1 |
| myFile open (fileName.csv) | 1 | 1 | 1 |
| If (myFile is open) Then | 1 | 1 | 1 |
| while (myFile is good && has two or more parameters) | 1 | n | n |
| **Procedure** GetLine (myFile, line) | 1 | n | n |
| Else | 1 | 1 | 1 |
| Throw exception | 1 | 1 | 1 |
| Catch (myFile) | 1 | 1 | 1 |
| Print “Error opening:” myFile | 1 | 1 | 1 |
| Close myFile | 1 | 1 | 1 |
| **Total Cost** |  |  | 2n + 11 |
| **Runtime** |  |  | O(n) |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Worst-case time complexity: pseudocode to create an object in a hash table** | **Line Cost** | **#Times Execute** | **Total Cost** |
| HashTable\* courses | 1 | 1 | 1 |
| For each line in myFile | 1 | n | n |
| line[i] = new Course() | 1 | 1 | 1 |
| Assign each new course to an instance variable | 1 | 1 | 1 |
| If there are 3 or more parameters in the object, Then | 1 | 1 | 1 |
| For each prerequisite | 1 | n | n |
| If the prerequisite does not have a courseNumber, Then | 1 | n | n |
| Print "prerequisite not valid” | 1 | n | n |
| Else | 1 | 1 | 1 |
| **Procedure** Courses->Insert variable | 1 | n | n |
| close file | 1 | 1 | 1 |
| **Total Cost** |  |  | 5n + 6 |
| **Runtime** |  |  | O(n) |

|  |  |  |  |
| --- | --- | --- | --- |
| **Worst case time complexity: opening and reading a file BST** | **Line Cost** | **#Times Execute** | **Total Cost** |
| string openFile(string fileName) | 1 | 1 | 1 |
| fstream myFile | 1 | 1 | 1 |
| Initialize string line | 1 | 1 | 1 |
| Try | 1 | 1 |  |
| myFile open (fileName.csv) | 1 | 1 | 1 |
| If (myFile is open) Then | 1 | 1 | 1 |
| while (myFile is good && has two or more parameters) | 1 | n | n |
| Procedure GetLine (myFile, line) | 1 | n | n |
| Else | 1 | 1 | 1 |
| Throw exception | 1 | 1 | 1 |
| Catch (myFile) | 1 | 1 | 1 |
| Print “Error opening: ” myFile | 1 | 1 | 1 |
| Close myFile | 1 | 1 | 1 |
| **Total Cost** |  |  | 2n + 10 |
| **Run Time** |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Worst case time complexity: pseudocode to create object BST** | **Line Cost** | **#Times Execute** | **Total Cost** |
| Create the Binary Search Tree Courses | **1** | **1** | **1** |
| For each line in myFile | **1** | **1** | **1** |
| line[i] = new Course() | **1** | **1** | **1** |
| If there are 3 or more parameters in the line, Then | **1** | **1** | **1** |
| For each p in prerequisite | **1** | **n** | **n** |
| If prerequisite does not have a courseNumber, Then | **1** | **1** | **1** |
| Throw exception | **1** | **1** | **1** |
| Insert (Course course) | **1** | **1** | **1** |
| If root is null | **1** | **n** | **n** |
| root = new Node(course) | **1** | **n** | **n** |
| Else | **1** | **1** | **1** |
| this->addNode(root, course) | **1** | **n** | **n** |
| **Total Cost** |  |  | **4n + 8** |
| **Runtime** |  |  | **O(n)** |

**Evaluation:**

**Analysis of the Advantages and Disadvantages of each data structured**

The binary search tree, hash table, and vector data structure share common operations such as inserting, searching, and deleting data. However, each data structure has its own algorithms to implement these operations. Furthermore, an algorithm uses resources such as runtime and memory, and this computational complexity measures an algorithm's efficiency. The user requirements for a program will determine the appropriate data structure. Moreover, a data structure dictates the specific algorithm that will be used. Each data structure has advantages and disadvantages.

One of the biggest advantages of the binary search tree (BST) is that it has an ordering property. This ordering property enables fast searching for an element. A good use case for the BST is when the data needs to be sorted. If the BST is perfect and the height of the tree is small, then this will produce even faster searches. The best-case time complexity for a BST search, deletion, and insertion is O(*logN*). Traversing the elements in order is easily accomplished by utilizing a BST. Another advantage of BST is that it is memory efficient. A BST does not use more memory than it needs.

A disadvantage of the BST is that an imbalanced tree will result in a longer time complexity for the same operations used on a self-balancing tree. Implementing a balanced tree is critical for time complexity. A BST worst-case time complexity for search, insertion and deletion is O(n). However, a self-balancing BST worst-case time complexity for the same operations is O(log(n)).

The biggest advantage of the hash table is that the average time complexity for search, inserting, and deletion is O(1). These operations are significantly faster for the hash table compared to the BST and vector data structure. A good use case for the hash table is finding all unique elements. Since the average time complexity for search is O(1), this would make the hash table very efficient.

Although the hash table has its advantages, it does come with some disadvantages. One disadvantage of the hash tables is generally, it does not preserve ordering. This means that the inserted data will not be in any specific order. Hash tables have a space complexity of O(n) due to the extra memory needed for the links in chaining. However, using the open-addressing collision technique is highly efficient in memory usage. If the chain becomes too long, then the worst-case for the search operation is O(n).

An advantage of the vector data structure is that it’s the simplest data structure to implement compared to a BST and a hash table. Furthermore, vectors are dynamic arrays, which means that they can resize themself when elements are added or deleted. A vectors data structure access operation’s worst-case time complexity is O(1).

A disadvantage of the vector data structure is that it's one-dimensional. This limitation prohibits elements from having different data types. Common operations such as search, insertion, and deletion are slower for vectors compared to BST and hash tables. The worst-case time complexity for these operations is O(n). Moreover, vectors use contiguous memory, and their worst-case space complexity is O(n).

**Recommendation**

After analyzing all three data structures and the user requirements, I recommend using the BST data structure. ABCU requires the program to sort the course information by alphanumeric course number from lowest to highest and print the sorted list to a display. An advantage of the BST is that it has an ordering property that will accomplish sorting the course numbers. Furthermore, if a binary tree is traversed in-order, the output will produce sorted key values in ascending order. More specifically, a self-balancing BST has a worst-case time complexity for the operations of in-order traversal, search, insert, and deletion of O(logn). This is far better than a vector or a hash table for the same operations. All three data structures have the same space complexity of O(n); therefore, space complexity did not influence my final recommendation. Although implementing a self-balancing BST can be demanding, the efficiency of the data structure is worth the challenge.

Citation:

Vahid, F., Lysecky, S., Siu, R., Nkenge, W. L., Edgcomb, A., & Yuen, J. Y. (2018). *CS 300: Data Structures and Algorithms*. Zybooks. Retrieved November 12, 2022, from <https://learn.zybooks.com/zybook/CS-300-T2806-OL-TRAD-UG.22EW2/chapter/4/section/1>

*Binary search tree*. GeeksforGeeks. (n.d.). Retrieved December 6, 2022, from <https://www.geeksforgeeks.org/binary-search-tree-data-structure>