**Big O Analysis of Vector, Hashtable, and BST**

**Vector**

|  |  |  |  |
| --- | --- | --- | --- |
| Read File and Create a course | Line cost | Times Executed | Cost |
| Create a vector of string type named openFile w/ parameters (string and line | 1 | 1 | 1 |
| initialize vector of strings contents to hold contents of the file is opened; | 1 | 1 | 1 |
| initialize string line to hold one single line; | 1 | 1 | 1 |
| initialize ifstream inStream to get contents of file openfile with instream using filename ; | 1 | 1 | 1 |
| if instream can’t open the file { | 1 | n | 2n |
| print the error message “unable to open open file.”; | 1 | 1 | 1 |
| pull line from instream until no more lines in the opened file ; | 1 | n | 8n |
| push line to the end of contents; | 1 | n | n |
| **Analysis** | 8 | 5 +3n | 5 + 11n  **Big 0 Value**  Worst Case =  **O(N)** |

**Hash Table**

|  |  |  |  |
| --- | --- | --- | --- |
| Open and Read File to Hash Table | Line cost | Time executed | Total cost |
| Initialize fstream fileStream to get contents of file; | 1 | 1 | 1 |
| Initialize string line to hold a single line in file; | 1 | 1 | 1 |
| Initialize stringstream lineStream to get contents of each line ; | 1 | 1 | 1 |
| Initialize string token to hold a single word in line; | 1 | 1 | 1 |
| Open fileName w/ fileStream ; | 1 | 1 | 1 |
| Initialize integer count to hold token count per the line in file; | 1 | 1 | 1 |
| Get line from fileStream until no line is left; | 1 | N | N |
| Fill lineStream with the current line ; | 1 | N | N |
| Set count to 1; | 1 | N | N |
| Create new Course\* aCourse for each line in file ; | 1 | N | N |
| Get token from lineStream up to ‘,’ until none left; | 1 | 2n | 2n |
| If (count is equal to 1) { | 1 | N | N |
| Set aCourse courseNumber to token; | 1 | N | N |
| Incrementally increase count; | 1 | N | N |
| Else if ( count equals 2) { | 1 | N | n |
| Set aCourse courseName to token; | 1 | N | N |
| Incrementally increase count; | 1 | n | N |
| If( token is existing in HashTable as course) { | 1 | n-1 | N |
| Add token to aCourse PreReqs; | 1 | n-1 | N |
| Output “format error”; | 1 | 1 | 1 |
| Incrementally increase count; | 1 | n | N |
| If ( count is less than 2) { | 1 | 1 | 1 |
| Output “Error : Course must contain course name and course number.” | 1 | 1 | 1 |
| assign aCourse to HashTable; | 1 | n |  |
| **Analysis** | **Big O Value**  **O()** | | |

|  |  |  |  |
| --- | --- | --- | --- |
| Create Course | Line Cost | Time Executed | Total Cost |
| For ( each node in table) { | 1 | 1 | 1 |
| If (nodes key is not equal to UINT\_MAX) { | 1 | 1 | 1 |
| Output node key; | 1 | 1 | 1 |
| Output node course; | 1 | 1 | 1 |
| Create Node of list node and set to node pointer to next; | 1 | 1 | 1 |
| While ( listNode is not equal to null) { | 1 | N | N |
| Output listNode key; | 1 | 1 | 1 |
| Output listNode course; | 1 | 1 | 1 |
| Set listNode to listNode pointer to next; | 1 | 1 | 1 |
| **Analysis** | 4 + n  **Big O Value**  **O(N)** | | |

**BST**

|  |  |  |  |
| --- | --- | --- | --- |
| Open and read file | Line cost | Time executed | Total cost |
| Initialize fstream fileStream to get contents of file; | 1 | 1 | 1 |
| Initialize string line to hold a single line in file; | 1 | 1 | 1 |
| Initialize stringstream lineStream to get contents of each line; | 1 | 1 | 1 |
| Initialize string token to hold a single word in line; | 1 | 1 | 1 |
| Open fileName with fileStream; | 1 | 1 | 1 |
| Initialize int count to hold the token count per line in file; | 1 | 1 | 1 |
| Get line from fileStream until none left; | 1 | N | N |
| Fill lineStream with current line; | 1 | N | N |
| Set count to 1; | 1 | N | N |
| Create Course acourse for each line in file; | 1 | N | N |
| Get token from lineStream up to ‘,’ until none is left; | 1 | 2N | 2n |
| if (count equals 1) { | 1 | N | N |
| set aCourse’s courseNumber to token; | 1 | N | N |
| incrementally increase count ; | 1 | N | N |
| else if (count == 2) { | 1 | N | N |
| set aCourse’s courseName to token; | 1 | N | N |
| incrementally increase count; | 1 | N | N |
| if (token exists in bst as course) { | 1 | N | N |
| add token to aCourse’s PreReqs; | 1 | N | N |
| output file format error; | 1 | 1 | 1 |
| increment count | 1 | N | N |
| if (count less than 2) { | 1 | 1 | 1 |
| output "Error in file format, each course must have course # and name." ; | 1 | 1 | 1 |
| Insert aCourse into bst; | 1 | N |  |
| clear lineStream for next line ; | 1 | n | N |
| **Analysis** | 6 + + 16N  ***Big O Value***  ***O()*** | | |

|  |  |  |  |
| --- | --- | --- | --- |
| Create a new Course | Line Cost | Time Executed | Total Cost |
| if (acourse courseNumber is less than current node’s courseNumber) | 1 | 1 | 1 |
| if (node left child node is empty) { | 1 | 1 | 1 |
| add new Node with course at node left child; | 1 | 1 | 1 |
| recursively traverse node left sub-tree | 1 | N | N |
| if (node’s right child is empty) { | 1 | 1 | 1 |
| add new Node with course at node right child; | 1 | 1 | 1 |
| recursively traverse node right sub-tree; |  | n | n |
| **Analysis** | 3 + 2n  **Big O Value**  **O(N)** | | |

**Data Structure Analysis**

**Vector**

When looking in o the Vector data structure, some positives of this structure are apparent. It is easy to implement the vector structure within the code. The run time of adding elements to the back end of a vector is constant. Some cons that stand out in implementing vectors is: The vector must first be sorted in order to take advantage of this data structure. Elements must be shifted in order to take items out or add items to a vector. This makes for inefficient use of time in the algorithm.

**Hash Table**

Looking at the Hash table as a viable option of a data structure to be used, we can see pros and con’s within it as well. A pro for a hash table would be the ability to directly access any element in the table, via a key. Another is the ability to add or delete an element in the table, no matter the size of the table. If implemented properly, hash tables are the most efficient in terms of speed. A Con of hash tables is they take up more space than what is needed, through resizing by a factor of 2 generally. Another is retrieving an element that does not store the order of the table. One final issue with the hash table is the table stores some elements randomly, based on the size of buckets, etc, causing the possibility of some ways of missing elements when searching

**BST**

Binary search trees are an efficient way to store data within a program. They have pros and cons like any other data structure. One pro is the fact that when retrieving elements in the tree, each element is retrieved in order. Another is that you can insert and delete elements in O(logN) time. Another is the efficiency of speed to access elements with the tree. A con of the BST is the shape of the tree is dependent on the first item inserted into the tree. Another is the tree must maintain a form of balance in order to be utilized correctly.

**Overall Analysis**

In my analysis of the Vector, Hash Table, and Binary Search Tree, data structures, I factored in the pros and cons of what should be factored into my decision on which data structure will be used in the program for ABCU. All the data structures were efficient in their own manner, but also presented issues individual to each style as well. My decision to use the BST in the code was based on a few different factors. One factor was the storing of courses does not require them to be sorted and stored in a certain order. This is not necessary to access the courses within a BST. Since the tree uses inOrder to access the courses within the BST, access to courses in alphabetical order happens without the additional function of sorting. In terms of searching the BST the time of O(N) is nearly as good as constant time. The hash table is only faster than the BST, but with other factors playing into that decision. The speed of the BST is still efficient enough to use properly and be useful. In my opinion, the Binary Search Tree is the best Data structure to use within the program for ABCU.