**Michael Steenkamp  
Amy Burns  
Project One – Pseudocode  
April 2nd, 2022**

**Table of Contents**

|  |  |
| --- | --- |
| **Description** | **Page** |
| Main | 2 |
| Load Data | 3 |
| Vector – Store Subject | 4 |
| Vector – Print List | 4 |
| Vector – Print Course | 4 |
| Hash Map – Background Understanding | 5 |
| Hash Map – Store Subject | 6 |
| Hash Map – Print List | 6 |
| Hash Map – Print Course | 7 |
| BST – Background Understanding | 7 |
| BST – Store Subject | 8 |
| BST – Print List | 8 |
| BST – Print Course | 9 |
| Time And Space Complexity | 10 - 15 |
| Reflection And Recommendation | 16 - 17 |

//Main Function  
main()  
{  
 // Display Options  
 Display Menu  
  
 // Get decision  
 Get user input  
  
 // Ensure that user input is valid  
 While user input is invalid  
 get user input  
  
 While input is not option 4  
 {  
 // Logic according to each option  
 // Option 1: Load data  
 If user choose option one  
 Call Function loadData()  
  
 // Option 2: Print Course List  
 Else If user chooses option two  
 Call Function printList()  
  
 // Option 3: Print Course  
 Else If user chooses option three  
 Call Function printCourse()  
  
 // Option 4: Exit Program  
 Else  
 Display exit message  
 Exit program  
 }  
}

// Load Data into data structure  
loadData()  
{   
 // Get file name  
 Get file name from input  
  
 // Open file to read  
 Open file using ifstream  
  
 While file is not open  
 Print error message  
 Get file name from input  
 Open file using ifstream  
  
 // While you can read line from file  
 While getline works  
 Declare stringstream  
 Read id using getline and stringstream and separator specifier  
 Read name using getline and stringstream and separator specifier  
  
 // There is less than two parameters  
 If id or name is empty  
 Print error message  
 Exit Function  
   
 // Used to store prerequisites  
 Declare string vector  
  
 // While there are still some parameters  
 While you can keep reading  
 Read prerequisites using getline and stringstream and separator specifier  
  
 // Check that prerequisite is already in data structure  
 If checkPrerequisite(string) // Boolean function  
 Append prerequisite to vector  
  
 // After all prerequisites have been read and stored  
 Call function to store subject // Function will change according to specific data structure  
  
 // Close file  
 Close File  
}

// Store subject into data structure – Vector   
storeSubject\_Vector(string ID, string Name, vector<string> Prerequisites)  
{  
 // Append the new subject to the end of classes vector  
 // Vector used to store all subjects  
 Push Back to vector  
 Set ID  
 Set Name  
 Set prerequisites  
}  
  
// Print List – Vector  
printList\_Vector()  
{  
 Declare variable to store minimum subject index  
 Initialize variable to 0  
  
 // Store all subjects that have been printed  
 Declare vector to store subjects that have been printed  
   
 For all subjects in classes vector  
 For all subjects in classes vector  
 // Check if current subject comes earlier than variable  
 If current subject has not been printed AND is in CS department  
 If subject ID is less than subject ID at index variable   
 Set variable equal to current subject  
 Display subject at index variable  
 Store subject at index variable into vector used to store printed subjects  
}  
  
// Print Course – Vector  
printCourse\_Vector()  
{  
 Get user input for course  
   
 While user input is not valid  
 Print error message  
 Get user input for course  
   
 For all subjects in classes vector  
 If subject is same as user input  
 Print subject and prerequisites  
 Exit Function  
   
 // Did not find subject  
 Print error message  
 Exit Function  
}

## // Used for each Node in Hash Map Struct subject

## {

## String ID String Name

## String Vector // used to store prerequisites

## } // Hash Map class Decleration Class hashTable {

## Struct Node { subject next Node pointer } Node Pointer Vector (nodes) Methods…

}  
  
// Used to create hash value using ID  
hash(string)  
{  
 Declare int variable stringHash  
 Initialize stringHash to 5381  
  
 For each character in ID  
 stringHash is equal to (stringHash times 33) plus character ASCII value  
  
 return stringHash modular classes size  
}

// Store subject into data structure – Hash Map  
storeSubject\_HashMap(string ID, string Name, vector<string> Prerequisites)  
{  
 // get index  
 Declare variable index  
 Set index equal to hash()  
  
 If nodes at index is empty  
 // Node initializes values with parameters, O(1)  
 Set nodes at index equal to new Node(parameters)  
 else  
 Declare Node pointer  
 Set node pointer equal to nodes at index  
   
 //iterate through chain until at the end  
 While node pointer next != nullptr  
 // iterate  
 node pointer is equal to node pointer next  
   
 // At the end of chain  
 Set node pointer next equal to new Node(parameters)  
}  
  
// Print List – Hash Map  
printList\_HashMap()  
{  
 Declare variable to store minimum subject index  
 Initialize variable to first subject in nodes  
  
 // Store all subjects that have been printed  
 Declare vector to store subjects that have been printed  
  
 Delcare node pointer  
  
 For all items in hash table  
 For all items in hash table  
 If nodes at index is not empty  
 Set node pointer equal to nodes at index  
  
 While node pointer is not equal to nullptr  
 // Check if current subject comes earlier than variable  
 If current subject has not been printed AND is in CS department  
 If subject ID is less than subject ID at index variable   
 Set variable equal to current subject  
  
 Display subject at index variable  
 Store subject at index variable into vector used to store printed subjects  
}  
  
// Print Course – Hash Map  
printCourse\_HashMap()  
{  
 Get user input  
  
 While user input is not valid  
 Print error message  
 Get user input  
  
 // Get index  
 Get hash value using user input and hash function  
   
 If nodes at index hash value is not nullptr  
 Declare node pointer  
 Initialize node pointer to point to nodes at index hash value  
 While node pointer is not nullptr  
 If node pointer ID is same as user input  
 Print subject and prerequisites  
 Exit Function  
 Set node pointer equal to next node  
  
 //Not found  
 Print error message  
 Exit Function  
}  
  
// Used for each Node in BST  
Struct Subject  
{  
 String ID  
 String Name  
 String Vector // Used to store prerequisites  
  
 Bid string  
 Node Pointer left  
 Node Pointer right  
  
 Constructors…  
}  
  
// Used to create BST  
Class BST  
{  
 Node Pointer root  
  
 Methods…  
}

// Store subject into data structure – BST  
storeSubject\_BST(string ID, string Name, vector<string> Prerequisites)  
{  
 If root is same as nullptr  
 Set root equal to new Node(parameters)  
 Else  
 // Call function and pass starting subject pointer (root) as parameter  
 Call Function addSubject  
}  
  
addSubject\_BST(Subject\* subject, string ID, string Name, vector<string> Prerequisites)  
{  
 // Check left subtree  
 If ID is less than subject ID  
 // Check Leaf  
 If subject left pointer is equal to nullptr  
 Set subject left pointer equal to new Node(parameters)  
 Else  
 // Go into left subtree  
 Recursively call addSubject\_BST(subject left pointer, …)  
 // Check right subtree  
 Else  
 // Check Leaf  
 If subject right pointer is equal to nullptr  
 Set subject right pointer equal to new Node(parameters)  
 Else  
 // Go into right subtree  
 Recursively call addSubject\_BST(subject right pointer, …)  
}  
  
// Print List – BST  
printList\_BST()  
{  
 // Call function and pass starting subject pointer (root) as parameter  
 Call Function displayList\_BST  
{  
  
display\_BST(Subject\* subject)  
{  
 If subject is not equal to nullptr  
 Recursively call display\_BST(subject left pointer)  
  
 If subject is in CS department  
 Display subject  
  
 Recursively call display\_BST(subject right pointer)  
}

// Print Course – BST  
printCourse\_BST()  
{  
 Get user input  
  
 While user input is not valid  
 Print error message  
 Get user input  
  
 Declare node pointer  
 Initialize node pointer to point to root  
  
 While node pointer does not equal nullptr  
 // Found  
 If node pointer ID is same as user input  
 Print subject and prerequisites  
 Exit Function  
   
 // Not Found - Traverse  
 If node pointer ID is greater than user input  
 // Inside left subtree  
 Set node pointer equal to left node pointer  
  
 Else  
 // Inside right subtree  
 Set node pointer equal to right node pointer  
  
 //Not found  
 Print error message  
 Exit Function  
}

//Main Function

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Display Menu | 1 | 1 | 1 |
| Get user input | 1 | n – Until Success | n |
| Option 1 - Vector | n | 1 | n |
| Option 1 - Hash Map | n | 1 | n |
| Option 1 - BST | Log n | 1 | Log n |
| Option 2 - Vector | n^2 | 1 | n^2 |
| Option 2 - Hash Map | n^2 | 1 | n^2 |
| Option 2 - BST | Log n | 1 | Log n |
| Option 3 - Vector | n | 1 | n |
| Option 3 - Hash Map | n | 1 | n |
| Option 3 - BST | n | 1 | n |
| Option 4 | 1 | 1 | 1 |
| **Total Cost** | | | Vector 1 – 2n + 2 |
|  | | | Vector 2 – n + n^2 + 2 |
|  | | | Vector 3 – 2n + 2 |
|  | | | Hash Map 1 – 2n + 2 |
|  | | | Hash Map 2 – n + n^2 + 2 |
|  | | | Hash Map 3 – 2n + 2 |
|  | | | BST 1 – n + log n + 2 |
|  | | | BST 2 - n + log n + 2 |
|  | | | BST 3 – 2n + 2 |
| **Runtime** | | | Vector 1 – O (n) |
|  | | | Vector 2 – O (n^2) |
|  | | | Vector 3 -O (n) |
|  | | | Hash Map 1 – O (n) |
|  | | | Hash Map 2 – O (n^2) |
|  | | | Hash Map 3 – O (n) |
|  | | | BST 1 – O (log n) |
|  | | | BST 2 - O (log n) |
|  | | | BST 3 - O (n) |

// Load Data into data structure

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Get file name from input | 1 | 1 | **n** |
| Open File | 1 | **n – Until Success** | **n** |
| While getline works, Declare stringstream Read id  Read name | 1 | **1** | **1** |
| If id or name is empty, Print error message Exit Function | 1 | **1** | **1** |
| While keep reading, Read prerequisites | 1 | **n – Number of Prereq** |  |
| If checkPrerequisite() | n | **n – Number of Prereq** | **n** |
| Append prerequisite | 1 | **1** | **1** |
| store subject - Vector | 1 | **1** | **1** |
| store subject - Hash Map | n | **1** | **n** |
| store subject - BST | log n | **1** | log n |
| Close File | 1 | **1** | **1** |
|  |  | **Total Cost** | Vector – 3n + 5 |
|  |  |  | Hash Map – 4n + 4 |
|  |  |  | BST – 3n + log n + 4 |
|  |  | **Runtime** | Vector – O (n) |
|  |  |  | Hash Map – O (n) |
|  |  |  | BST – O (log n) |

// Store subject into data structure – Vector

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Push Back to vector | 1 | 1 | 1 |
| Set ID | 1 | 1 | 1 |
| Set Name | 1 | 1 | 1 |
| Set prerequisites | 1 | 1 | 1 |
| **Total Cost**  **Runtime** | | | 4 |
| O (1) |

// Print List – Vector

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Declare variable | 1 | 1 | 1 |
| Declare vector | 1 | 1 | 1 |
| For all subjects  For all subjects | 1 | n^2 | n^2 |
| If not been printed AND in CS department | 1 | 1 | 1 |
| Set variable | 1 | 1 | 1 |
| Display subject | 1 | 1 | 1 |
| Store subject into vector | 1 | 1 | 1 |
| **Total Cost**  **Runtime** | | | n^2 + 6 |
| O (n^2) |

// Print Course – Vector

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Get user input | 1 | 1 | 1 |
| While not valid Print error  Get user input | 1 | n | n |
| For all subjects | 1 | n | n |
| If subject same as input | 1 | 1 | 1 |
| Print | 1 | 1 | 1 |
| Exit Function | 1 | 1 | 1 |
| Print error message | 1 | 1 | 1 |
| Exit Function | 1 | 1 | 1 |
| **Total Cost**  **Runtime** | | | 2n + 6 |
| O (n) |

// Store subject into data structure – Hash Map

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Declare variable | 1 | 1 | 1 |
| Set index equal to hash() | n – Hash Function | 1 | n |
| If empty | 1 | 1 | 1 |
| Set nodes | 1 | 1 | 1 |
| Else |  |  |  |
| Declare Node pointe | 1 | 1 | 1 |
| Set node pointer | 1 | 1 | 1 |
| While next != nullptr | 1 | n | n |
| // iterate | 1 | 1 | 1 |
| Set pointer next | 1 | 1 | 1 |
| **Total Cost**  **Runtime** | | | 2n + 7 |
| O (n) |

// Print List – Hash Map

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Declare variable | 1 | 1 | 1 |
| Initialize variable | 1 | 1 | 1 |
| Declare vector | 1 | 1 | 1 |
| Delcare node pointer | 1 | 1 | 1 |
| For all items  For all items | 1 | n^2 | n^2 |
| If nodes is not empty | 1 | 1 | 1 |
| Set node pointer | 1 | 1 | 1 |
| While node pointer | 1 | n | n |
| If not been printed AND is in CS department | 1 | 1 | 1 |
| If subject ID is less than subject ID | 1 | 1 | 1 |
| Set variable | 1 | 1 | 1 |
| Display subject | 1 | 1 | 1 |
| Store subject | 1 | 1 | 1 |
| **Total Cost**  **Runtime** | | | n \* n^2 + 11 |
| O (n^2) |

// Print Course – Hash Map

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Get user input | 1 | 1 | 1 |
| While input is not valid Print error message Get user input | 1 | n | n |
| Get hash value | n – Hash Function | 1 | n |
| If nodes | 1 | 1 | 1 |
| Declare node pointer | 1 | 1 | 1 |
| Initialize node pointer | 1 | 1 | 1 |
| While node pointer | 1 | n | n |
| If node pointer ID is same as user input | 1 | 1 | 1 |
| Print subject and prerequisites | 1 | 1 | 1 |
| Exit Function | 1 | 1 | 1 |
| // Iterate | 1 | 1 | 1 |
| Print error message | 1 | 1 | 1 |
| Exit Function | 1 | 1 | 1 |
| **Total Cost**  **Runtime** | | | 3n + 10 |
| O (n) |

// storeSubject – BST

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| If not root | 1 | 1 | 1 |
| Set root | 1 | 1 | 1 |
| Else |  |  |  |
| Call Function addSubject | log n | 1 | log n |
| **Total Cost**  **Runtime** | | | log n + 2 |
| O (log n) |

// addSubject – BST

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| If ID less subject ID | 1 | 1 | 1 |
| If not subject left pointer | 1 | 1 | 1 |
| Set subject left pointer | 1 | 1 | 1 |
| Else |  |  |  |
| Recursion (subject left pointer) | Log n | 1 | Log n |
| Else |  |  |  |
| If not subject right pointer | 1 | 1 | 1 |
| Else |  |  |  |
| Recursion (subject right pointer) | Log n | 1 | Log n |
| **Total Cost**  **Runtime** | | | 2log n + 4 |
| O (log n) |

// Print List – BST

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Call Function displayList\_BST | Log n | 1 | Log n |
| **Total Cost**  **Runtime** | | | Log n |
| O (log n) |

// displayList – BST

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| If subject | 1 | 1 | 1 |
| Recursion (subject left pointer) | Log n | 1 | Log n |
| If subject in CS department | 1 | 1 | 1 |
| Display subject | 1 | 1 | 1 |
| Recursion (subject right pointer) | Log n | 1 | Log n |
| **Total Cost**  **Runtime** | | | 2 log n + 3 |
| O (log n) |

// Print Course – BST

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Get user input | 1 | 1 | 1 |
| While not valid Print error  Get user input | 1 | n | n |
| Declare node pointer | 1 | 1 | 1 |
| Initialize node pointer | 1 | 1 | 1 |
| While node pointer | 1 | n | n |
| If node pointer ID is same as user input | 1 | 1 | 1 |
| Print subject and prerequisites | 1 | 1 | 1 |
| Exit Function | 1 | 1 | 1 |
| If node pointer ID is greater than user input | 1 | 1 | 1 |
| Set node pointer equal to left node pointer | 1 | 1 | 1 |
| Else |  |  |  |
| Set node pointer equal to right node pointer | 1 | 1 | 1 |
| Print error message | 1 | 1 | 1 |
| Exit Function | 1 | 1 | 1 |
| **Total Cost**  **Runtime** | | | 2n + 11 |
| O (n) |

Reflection And Recommendation

All three of the showcased data structures have their specific advantages and disadvantages according to three categories: adding, sorting, and searching. In this reflection I will describe these positives and negatives for each data structure, I will then give my recommendation as to which data structure I believe should be used.

The first data structure that I will be reflecting on is the vector. The vector can be used in almost any scenario and is used as the core data structure for many abstract data structures. Vectors have a worst-case scenario for adding data of O (n + m), where n is the number of elements to be inserted and m is the number of elements to be moved. I particularly chose to append data to the vector so that we can minimize the time complexity by making it O (1). A vector can return data at a given index in constant time; however, when searching by item and not by index, it requires us to search through each index until that specific item is found; thus, taking O (n). If the data was sorted then we would be able to use a binary search algorithm to search in logarithmic time but this would then require sorting to take place, which would affect the time complexity of the data structure. Vectors can either be sorted after all the data has been inserted or during the process of insertion. There are many algorithms that can be used to sort a vector, with the best algorithm having a worst-case scenario of O (n log n).

The second data structure that I will be reflecting on is the hash map. The hash map is an abstract data structure that generally makes use of vectors, arrays, or linked lists. Hash maps are very unique when it comes to storing data; when given data they use a hash function to calculate the specific items hash value, this hash value is used as the index value to store the data. If there is already data at that specific hash value then a process called “chaining” takes place, this is where a linked list is used to link data together who have the same hash value. Retrieving data from a hash map has a worst-case scenario time complexity of O (n); however, more often than not, data can be retrieved in relative constant time; this is due to each item and their specific hash value being unique (most of the time). Sorting a hash map is not so easy because data is stored at random according to their specific hash value, but you can iterate through the hash map, store each item into a sortable data structure, and then sort the data inside the sortable data structure.

The last data structure that I will be reflecting on is the binary search tree. The binary search tree data structure makes use of a linked list and a set of rules to distinguish itself from other trees. Inserting data into a binary search tree is fairly straightforward and time conservative, in the worst-case scenario inserting data will result in a time complexity of O (log n). The data structure has a set of rules that guides how data is inserted thus, these rules can be defined to benefit the time complexity of accessing data. Searching for data in a binary search tree is done by comparing and then either traversing into a specific subtree, returning the current item, or returning null; the time complexity for searching is also O (log n). Sorting data is fairly straightforward because data is sorted into the tree as they get inserted; they can then be displayed in ascending or descending order according to how the tree is traversed.

In conclusion I believe that I could have modified the vector and hash map data structures to make them more efficient and less dependable on memory. I believe that the binary search tree is the best data structure for this specific situation, it has a relatively fast insertion, searching and sorting algorithms, not to mention it requires less memory than the other data structures provided. The binary search tree can store and sort data at the same time in O (log n) time which is much faster than O (n) when it comes to the average case, not to mention it is also relatively fast in regards to searching in the average case scenario. Thus, I conclude this reflection by recommending the binary search tree.