**Module 6 Project 1**

Jermaine Wiggins

Southern New Hampshire University

CS300 Analysis and Design

Prof. Sathish Gopalakrishnan

Feb 16, 2025

**Module 6 Project 1**

**First 3 in previous millstone assignments**

**Read and check File is valid function name; checkFile(ifstream, fileName)**

**Load file function name; loadFile(dataStructure, fileName)**

**Search Structure name; searchStruct(dataStructure, courseToSearch)**

**Sort Structure name; sortStruct(dataStructure)**

**Partition Function (vector)**

**SET** variables for high and low to beginning and end of vector

**FIND** midpoint

**SET** pivot to courses[midpoint].courseId

**WHILE** not done

**WHILE** courses[low] less than pivot increment low by 1

**WHILE** pivot less than courses[high] decrement high by 1

**IF** low greater than high end loop

**ELSE** swap high and low

**RETURN** high

**Quicksort Function**

**CREATE** variable mid and set to 0

**IF** low greater than or equal to end return

**SET** mid to partition (courses, low, high)

**RECURSIVELY** sort lower end with quicksort

**RECURSIVELY** sort high end with quicksort

**Sort Hashtable**

**CREATE** empty map to store course structure

**ITERATE** through hashtable

**IF** hashtable[index] not empty add to empty map

**CALL** quicksort function to sort

**Print Structure name; printStruct(dataStructure)**

**Print vector**

**FOR** course in courses

**PRINT** allCourse info (course code and course name)

**FOR** prerequisite in prerequisites vector

**PRINT** prerequisite

**Print hashtable**

**FOR** course in empty map

**PRINT** allCourse info (course code and course name)

**FOR** prerequisite in prerequisites vector

**PRINT** prerequisite

**Print Binary Search Tree In Order Left → Root → Right**

**If** node is null return

**Call** in order method on node **→** left to recursively visit left side of tree

**Print** bid info

**If** size of prerequisite vector > 0

**Loop** through vector and print all prerequisites if any

**Call** in order method on node **→** right to recursively visit right side of tree

**PRINT** menu options

**GET** user input

**DO WHILE** user input not 4

**CASE 1: checkFile(ifstream, fileName) loadFile(ifstream, fileName)**

**CASE 2: sortStruct(dataStructure), printStruct(dataStructure)**

**CASE 3: GET** user input for courseToSearch

**searchStruct(dataStructure, courseToSearch)**

**CASE 4: PRINT** “Exiting program”

**Run time analysis Big O**

|  |  |  |  |
| --- | --- | --- | --- |
| **Data Structure** | **Insert** | **Search** | **Sort and Print** |
| **Vector** | **O(1) insert at end** | **O(1) - O(N)** | **O(N log N)** |
| **Hash table<vector>** | **O(1) - O(N)** | **O(1) - O(N)** | **O(N log N)** |
| **Binary search tree<linkedlist>** | **O(log N) - O(N) unbalanced trees** | **O(log N) - O(N) unbalanced trees** | **O(N)** |

**Insert**

For each structure the file is opened O(1) then each line is read O(N) and parsed O(N). Next an object is created O(1) and stored in the appropriate structure, Big O depending on the structure, (vector O(1) since it’s inserted at the end, hash table also O(1) but degrades to O(N) if many collisions and the BST would be O(log N) (because must traverse tree to insert if not head) for a balanced tree but O(N) if unbalanced.

**Search**

To search each structure the structure must be traversed, best case its first item searched worst case it’s at the end, for a vector it would be O(1) – O(N), hash table being the same, and BST O(log N) - O(N), where N is the height if the tree.

**Sort and print**

For a vector that inserts elements at the end when loaded, a quick sorting algorithm is used. Quicksort works by dividing the dataset into two parts about log N times, and at each step, it compares and processes all N elements. Since printing is O(N) and sorting is O(N log N), the complexity stays O(N log N). this is also the same for a hash table because you’ll extract each element to a vector O(N) the follow the same process leading to O(N log N). For a BST since every node is visited once with an in-order traversal the time complexity will always be O(N).

|  |  |  |
| --- | --- | --- |
| Data structure | Advantages | Disadvantages |
| Vector | Easiest to implement  Fast insertions  Fast searches small datasets | If data set is large search and sort can degrade drastically |
| Hash table | Fast searches  Fast insert if not many collisions | No way to sort besides extracting then sorting new structure  If many collisions run time degrades to O(N) |
| Binary search tree | Sorted during insertion  Fastest sort and print time | Most complex to implement  Uses most memory, stores course info and pointers |

The best data structure for this project depends on the number of courses in the file. If the file is small, using a vector would be ideal because it is the easiest to implement, uses the least memory, and has the best time complexity for insertion. For larger files, while a vector still provides the fastest insertion, a BST may be the better choice. The courses are inserted in order, so printing requires only an in-order traversal rather than sorting and printing. Additionally, a balanced BST offers efficient search performance.