Armando Gomez

6-2 Project One

CS-300-H2998 DSA Analysis and Design

23EW2

December 03, 2023

STRUCTURE Course

string courseCode

string name

string prerequisiteOne

string prerequisiteTwo

FUNCTION readFile(filePath)

OPEN file at filePath

courses = []

WHILE reading line from file

course = parseLine(line)

courses.append(course)

CLOSE file

RETURN courses

FUNCTION parseLine(line)

PARSE line into course details

RETURN new Course(course details)

**// Vector Operations**

FUNCTION loadCoursesVector(filePath)

RETURN readFile(filePath)

FUNCTION printCoursesVector(courses)

SORT courses in alphanumeric order

FOR EACH course in courses

PRINT course details

FUNCTION findCourseVector(courses, courseCode)

FOR EACH course in courses

IF course code matches

PRINT course details

**// Hash Table Operations**

FUNCTION loadCoursesHashTable(filePath)

courses = readFile(filePath)

hashTable = new HashTable()

FOR EACH course in courses

hashTable.insert(course.courseCode, course)

RETURN hashTable

FUNCTION printCoursesHashTable(hashTable)

FOR EACH course in hashTable

PRINT course details in alphanumeric order

FUNCTION findCourseHashTable(hashTable, courseCode)

course = hashTable.search(courseCode)

IF course is found

PRINT course details

**// Binary Search Tree Operations**

FUNCTION loadCoursesTree(filePath)

courses = readFile(filePath)

bst = new BinarySearchTree()

FOR EACH course in courses

bst.insert(course)

RETURN bst

FUNCTION printCoursesTree(node)

IF node is not NULL

printCoursesTree(node.left)

PRINT node.course details

printCoursesTree(node.right)

FUNCTION findCourseTree(node, courseCode)

IF node is NULL

RETURN "Course not found"

IF node.courseCode is courseCode

RETURN node.course details

IF node.courseCode < courseCode

RETURN findCourseTree(node.right, courseCode)

ELSE

RETURN findCourseTree(node.left, courseCode)

**// Menu System**

FUNCTION mainMenu()

dataStructure = NULL

courses = NULL

WHILE true

PRINT "1. Load Data Structure"

PRINT "2. Print Course List"

PRINT "3. Print Course"

PRINT "4. Exit"

choice = getUserInput()

SWITCH choice

CASE 1:

dataStructureType = getUserInput("Choose Data Structure: Vector, HashTable, BST")

filePath = getUserInput("Enter file path:")

IF dataStructureType is "Vector"

dataStructure = "Vector"

courses = loadCoursesVector(filePath)

ELSE IF dataStructureType is "HashTable"

dataStructure = "HashTable"

courses = loadCoursesHashTable(filePath)

ELSE IF dataStructureType is "BST"

dataStructure = "BST"

courses = loadCoursesTree(filePath)

CASE 2:

IF courses is not NULL

IF dataStructure is "Vector"

printCoursesVector(courses)

ELSE IF dataStructure is "HashTable"

printCoursesHashTable(courses)

ELSE IF dataStructure is "BST"

printCoursesTree(courses.root)

ELSE

PRINT "Load data first."

CASE 3:

courseCode = getUserInput("Enter Course Code:")

IF dataStructure is "Vector"

findCourseVector(courses, courseCode)

ELSE IF dataStructure is "HashTable"

findCourseHashTable(courses, courseCode)

ELSE IF dataStructure is "BST"

findCourseTree(courses.root, courseCode)

CASE 4:

EXIT

**Runtime Analysis**

**Vector**

Table 1: Read File

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **Cost per Line** | **Times Executes** | **Total Cost** |
| Open File | O(1) | 1 | O(1) |
| Read line | O(1) | n | O(n) |
| Parse line | O(1) | n | O(n) |
| Close file | O(1) | 1 | O(1) |
| **Total Cost** |  |  | **O(n)** |

Table 2: Create Course Object

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **Cost per Line** | **Times Executes** | **Total Cost** |
| Create Course object | O(1) | n | O(n) |
| Append to vector | O(1)\* | n | O(n)\* |
| **Total Cost** |  |  | **O(n)** |

**Hash Table**

Table 1: Read File

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **Cost per Line** | **Times Executes** | **Total Cost** |
| Open File | O(1) | 1 | O(1) |
| Read line | O(1) | n | O(n) |
| Parse line | O(1) | n | O(n) |
| Close file | O(1) | 1 | O(1) |
| **Total Cost** |  |  | **O(n)** |

Table 2: Create Course Object

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **Cost per Line** | **Times Executes** | **Total Cost** |
| Create Course object | O(1) | n | O(n) |
| Insert into hash table | O(1)\*\* | n | O(n)\*\* |
| **Total Cost** |  |  | **O(n)** |

**BinarySearchTree**

Table 1: Read File

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **Cost per Line** | **Times Executes** | **Total Cost** |
| Open File | O(1) | 1 | O(1) |
| Read line | O(1) | n | O(n) |
| Parse line | O(1) | n | O(n) |
| Close file | O(1) | 1 | O(1) |
| **Total Cost** |  |  | **O(n)** |

Table 2: Create Course Object

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **Cost per Line** | **Times Executes** | **Total Cost** |
| Create Course object | O(1) | n | O(n) |
| Insert into BST | O(log n)\*\*\* | n | O(n log n) |
| **Total Cost** |  |  | **O(n log n)** |

**Vector:**

There are benefits to using a vector. First, it's easy to implement and allows for access to elements by index, which is useful when iterating through all the courses. Vectors also ensure that data is stored continuously, which can improve memory access patterns. However, one downside is that resizing the vector when it exceeds its capacity can be costly since it involves copying all the elements to a memory location. Additionally, searching for an element in a vector takes time (O(n)) which becomes inefficient when dealing with large datasets.

**Hash Table:**

The hash table is excellent in situations where fast searches are needed, as it has a time complexity of O(1) for insertions and searches on average. This means it is very efficient for tasks like locating a course. However, one downside of hash tables is that they do not have a built-in ordering mechanism, which can be problematic if the advisor requires the courses to be sorted. Furthermore, if there are instances of hash collisions, the performance of a hash table can decrease significantly. Nevertheless, this issue can usually be resolved by using a hash function.

**Binary Search Tree(BST):**

A Binary Search Tree (BST) offers a balanced approach, providing an average time complexity of O(log n), for insertions, deletions, and searches when the tree is balanced. This makes it efficient for both storing and retrieving data in a timely manner. By traversing the BST in an orderly manner, we can efficiently access elements in an order, which's particularly useful when printing courses in alphanumeric order. However, it's important to note that one drawback is the potential for the tree to become unbalanced depending on the order of insertions. In some cases, performance can degrade to O(n) in the worst-case scenario.

**Recommendation**

Based on the analysis of complexity and the specific requirements commonly associated with academic advising programs, my suggestion would be to utilize a binary search tree (BST) as the main data structure in your code. A BST provides a performance profile with an average time complexity of O(log n) for essential operations like insertion, deletion, and search. This is especially advantageous in an academic environment where courses are frequently added, removed, or searched.

One of the compelling reasons to opt for a BST is its inherent capability to maintain data in a sorted order. This feature proves beneficial when generating course lists in an alphanumeric sequence—a common necessity for academic advising programs. By traversing the tree in order (in order traversal), we can efficiently list courses in their order without requiring additional sorting algorithms. In contrast to vectors, which may not handle insertions and deletions efficiently (within the middle of the collection), BSTs handle these operations more effectively. This adaptability makes BSTs well suited for accommodating changes in course lists—an expectation that aligns well with a setting.

Although hash tables offer performance for lookups with an average time complexity of O(1), they lack the capability to maintain an ordered list of courses, which is a significant drawback in this application. On the other hand, vectors are simple and efficient for iteration, but they fall short in terms of performance when it comes to frequent insertions, deletions, and maintaining an ordered list. The BST strikes the balance between efficiency, order maintenance, and flexibility, making it the most suitable choice for the advisor program. Its ability to handle data and maintain an ordered structure aligns well with the typical requirements of an academic advising system.