Kashyab Maharjan

Southern New Hampshire University

CS300

Aug 11,2024

**Title: Final Project Part I - Pseudocode and Runtime Analysis**

**Introduction**:

The objective of this project is to create a software application for the Computer Science department at ABCU that enables academic advisers to effectively handle course information. The application will retrieve course data from a file, arrange it, and offer features such as generating a list of all courses in alphabetical order and presenting comprehensive information about a particular course, including its prerequisites. This paper presents the pseudocode for three distinct data structures: vector, hash table, and binary search tree (BST). It also includes a runtime analysis to suggest the most efficient structure for implementation.

**Pseudocode**:

1. **Vector Data Structure**

Opening and Reading the File:

// Open and read data from file

open file "courses.txt"

for each line in file

parse line into courseNumber, name, prerequisites

create a Course object with the parsed data

add Course object to Vector<Course>

end for

close file

Printing Course Information:

// Print course information for a specific course

function searchCourse(Vector<Course> courses, String courseNumber)

for each course in courses

if course.courseNumber equals courseNumber

print course.name

for each prerequisite in course.prerequisites

print prerequisite

end if

end for

end function

Menu Options:

// Main menu

function displayMenu()

print "1: Load courses"

print "2: Print course list in alphanumeric order"

print "3: Print course information and prerequisites"

print "9: Exit"

get user input

if input equals 1

loadCourses()

else if input equals 2

printCourseList(Vector<Course> courses)

else if input equals 3

print "Enter course number: "

get courseNumber

searchCourse(Vector<Course> courses, courseNumber)

else if input equals 9

exit

end if

end function

// Sort and print course list

function printCourseList(Vector<Course> courses)

sort courses by courseNumber

for each course in courses

print course.courseNumber, course.name

end for

end function

| **Vector** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | **1** | **n** | **n** |
| **If course == courseID** | **1** | **n** | **n** |
| **Output course info** | **2** | **1** | **1** |
| **For each prereq for course** | **1** | **n** | **n** |
| **Output prereq info** | **2** | **n** | **n** |
| **Total Cost** | | | **5N + 1** |
| **Runtime** | | | **O(n)** |

2**. Hash Table Data Structure**

Opening and Reading the File:

// Open and read data from file

open file "courses.txt"

for each line in file

parse line into courseNumber, name, prerequisites

create a Course object with the parsed data

add Course object to HashTable<Course>

end for

close file

Printing Course Information:

// Print course information for a specific course

function searchCourse(HashTable<Course> courses, String courseNumber)

if courseNumber exists in HashTable

print course.name

for each prerequisite in course.prerequisites

print prerequisite

end if

end function

Menu Options:

// Main menu

function displayMenu()

print "1: Load courses"

print "2: Print course list in alphanumeric order"

print "3: Print course information and prerequisites"

print "9: Exit"

get user input

if input equals 1

loadCourses()

else if input equals 2

printCourseList(HashTable<Course> courses)

else if input equals 3

print "Enter course number: "

get courseNumber

searchCourse(HashTable<Course> courses, courseNumber)

else if input equals 9

exit

end if

end function

// Sort and print course list

function printCourseList(HashTable<Course> courses)

convert HashTable to Vector

sort Vector by courseNumber

for each course in Vector

print course.courseNumber, course.name

end for

end function

| **Hash Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | **2** | **n** | **n** |
| **If course == courseID** | **1** | **n** | **n** |
| **Output course info** | **1** | **1** | **1** |
| **For each prereq for course** | **2** | **n** | **n** |
| **Output prereq info** | **4** | **n** | **n** |
| **Total Cost** | | | **9n + 1** |
| **Runtime** | | | **O(n)** |

3. **Binary Search Tree Data Structure**

Opening and Reading the File:

// Open and read data from file

open file "courses.txt"

for each line in file

parse line into courseNumber, name, prerequisites

create a Course object with the parsed data

add Course object to BinarySearchTree<Course>

end for

close file

Printing Course Information:

// Print course information for a specific course

function searchCourse(BinarySearchTree<Course> courses, String courseNumber)

node = find(courseNumber) in BinarySearchTree

if node exists

print node.course.name

for each prerequisite in node.course.prerequisites

print prerequisite

end if

end function

**Menu Options:**

// Main menu

function displayMenu()

print "1: Load courses"

print "2: Print course list in alphanumeric order"

print "3: Print course information and prerequisites"

print "9: Exit"

get user input

if input equals 1

loadCourses()

else if input equals 2

printCourseList(BinarySearchTree<Course> courses)

else if input equals 3

print "Enter course number: "

get courseNumber

searchCourse(BinarySearchTree<Course> courses, courseNumber)

else if input equals 9

exit

end if

end function

// Sort and print course list

function printCourseList(BinarySearchTree<Course> courses)

inOrderTraversal(BinarySearchTree)

end function

| **Tree Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | **1** | **n** | **n** |
| **If course == courseID** | **1** | **n** | **n** |
| **Output course info** | **2** | **1** | **1** |
| **For each prereq for course** | **1** | **n** | **n** |
| **Output prereq info** | **4** | **n** | **n** |
| **Total Cost** | | | **7n + 1** |
| **Runtime** | | | **O(n)** |

**Runtime Evaluation:**

ABCU's Computer Science department assessed three distinct data structures—a vector, a hash table, and a binary search tree—to determine their appropriateness for building a course management application. Every data structure has unique benefits and drawbacks depending on its time complexity and operational properties.

Implementing the vector data structure is simple and straightforward. It facilitates uncomplicated storage and retrieval of data, rendering it a favorable option for smaller datasets. Nevertheless, the main drawback of utilizing a vector is its linear search time, which proves to be inefficient when dealing with larger datasets. To find a particular course, it is necessary to go over the entire vector, which leads to a time complexity of O(n). Furthermore, the act of arranging the courses in alphabetical order prior to printing them adds to the time complexity, resulting in a slower overall procedure, especially when the number of courses increases.

Conversely, the hash table provides the quickest search time out of the three data structures, with an average-case time complexity of O(1) for search operations. This feature greatly enhances the speed and effectiveness of locating and accessing course information. However, a notable disadvantage of hash tables is their lack of intrinsic order, necessitating additional measures to achieve a sorted list of courses. The process of converting the hash table entries into a vector and subsequently sorting them provides a time complexity of O(n log n), which is a compromise for the quick access time during searches.

The binary search tree (BST) keeps the structure balanced by keeping the parts in their natural order. This makes operations like searches, additions, and deletions fast, with an average time complexity of O(log n). Furthermore, it provides support for in-order traversal, which simplifies the process of retrieving a sorted list of courses. Nevertheless, the performance of the binary search tree (BST) depends on its state of balance. When the tree becomes unbalanced, its temporal complexity can deteriorate to O(n), resulting in a decrease in efficiency. Furthermore, the development and management of a balanced binary search tree (BST) are more intricate in comparison to the other two data structures.

**Advantages and Disadvantages:**

-**Vector**:

-Advantages: easy to implement, suitable for small datasets.

-Disadvantages: The time it takes to do a linear search becomes inefficient when dealing with huge datasets. Additionally, sorting is necessary in order to obtain the desired ordered output, which further increases the time complexity.

**- Hash Table**:

-Advantages: For searches, the data structure has a fast access time of O(1), making it highly efficient for quickly discovering courses.

-Disadvantages: The system is unable to maintain order, thereby requiring additional processes to convert it into a sorted list.

- **Binary Search Tree:**

-Advantages: It inherently preserves organization, enhancing the efficiency of searches and the generation of ordered results.

-Disadvantages: The process of implementing and managing becomes more intricate, particularly when balanced trees are necessary to achieve optimal performance.

**Recommendation:**

Given the study of the runtime and the application's requirements, I recommend using a hash table for implementation. The hash table offers the most efficient search capabilities, which is crucial for swiftly accessing course information. The increased performance of search operations justifies the decision to transform the hash table into a vector for sorting, despite its inability to preserve order. This strategy effectively combines efficiency and usability, making it the optimal choice for the advising program.