Abbigale Junker

Professor Yurik

CS-300

15 February 2025

Project One Menu Pseudocode

**Menu**

WHILE choice is not 9

DISPLAY menu options

PROMPT user for choice

Choice = 1:  // Load data

START timer

LOAD data

DISPLAY data size

CALCULATE time

DISPLAY  time in clock ticks

DISPLAY time in seconds

BREAK

Choice = 2:  // Print sorted list of Computer Science courses

IF using vector

CALL printSortedCourses\_Vector(courseList)

ELSE IF using hashTable

CALL printSortedCourses\_HashTable(courseTable)

ELSE IF using binaryTree

CALL printSortedCourses\_BinaryTree(courseTree)

BREAK

Choice = 3:  // Search and print course details

GET user input for course

LOOP through courses until user input is found

IF user input is found

DISPLAY course number and title

IF course prerequisites is empty

DISPLAY no prerequisites

ELSE

DISPLAY prerequisites

BREAK

DISPLAY course not found error

BREAK

Choice = 9:  // Exit the program

DISPLAY “Good bye.”

EXIT menu program

ELSE:

DISPLAY invalid choice error

Project One Alphanumeric Order Pseudocode

// sorting for a vector

FUNCTION printSortedCourses\_Vector(courseList)

SORT courseList by course courseNumber

FOR each course in courseList

DISPLAY course courseNumber and course courseName

// sorting for a hash table

FUNCTION printSortedCourses\_HashTable(courseTable)

EXTRACT all courses from courseTable into list

SORT list by course courseNumber

FOR each course in the sorted list

DISPLAY course courseNumber and course courseName

// sorting for binary tree

FUNCTION printSortedCourses\_BinaryTree(courseTree)

TRAVERSE tree in in-order traversal

FOR each course in traversal result

DISPLAY course courseNumber and course courseName

Project One Evaluation

Vector analysis

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for each course in courseList** | 1 | n | n |
| **if the course is the same as courseNumber** | 1 | n | n |
| **Print out the course information** | 1 | 1 | 1 |
| **for each prerequisite of the course** | 1 | n | n |
| **print the prerequisite course information** | 1 | n | n |
| **Total Cost** | | | 4n + 1 |
| **Runtime** | | | O(n) |

Hash Table analysis

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **If courseNumber exists in courses** | 1 | 1 | 1 |
| Set course = courses[courseNumber] | 1 | 1 | 1 |
| **PRINT “Course Number: ” + course.number** | 1 | 1 | 1 |
| PRINT “Course Name: ” + course.title | 1 | 1 | 1 |
| **If course.prerequisites is not empty** | 1 | 1 | 1 |
| **PRINT “prerequisites: ”** | 1 | 1 | 1 |
| **For each prerequisite in course.prerequisites** | 1 | n | n |
| **Print prerequisite** | 1 | n | n |
| **Else print course not found error** | 1 | 1 | 1 |
| **Total Cost** | | | 4 + 2n |
| **Runtime** | | | O(n) |

Binary Search Tree analysis

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Set cur = course.root** | 1 | 1 | 1 |
| While cur is not null | 1 | n | n |
| **If courseNumber == cur.number** | 1 | n | n |
| PRINT “Course Number: ” + cur.number | 1 | 1 | 1 |
| **PRINT “Course title: ” + cur.title** | 1 | 1 | 1 |
| **If cur.prerequisites is not empty** | 1 | 1 | 1 |
| **PRINT “Prerequisites: ”** | 1 | 1 | 1 |
| **FOR each prerequisite in cur.prerequisites** | 1 | n | n |
| **Print prerequisite** | 1 | n | n |
| **ELSE print “No prerequisites”** | 1 | 1 | 1 |
| **Total Cost** | | | 2n |
| **Runtime** | | | O(n) |

Each structure has advantages and disadvantages. The vector is simple to implement and has fast insertions at the end, but it is slow to search as it would have to run through the list linearly. The hash table is more complex and is capable of quick insertions and deletions, but collisions can affect performance and it uses more memory than a vector. The binary search tree is generally faster than a vector, maintains a sorted order, and is more memory efficient than a hash table, but it is the most complex to implement in comparison to the other two and has slower insertions and deletions. Based on these advantages and disadvantages, I would recommend a hash table since it is still efficient even in the worst case, being hash collisions. Even though the graphs above state their runtimes to be O(n) in a search in the worst case, the average runtime for searches in a hash table is O(1) because it uses a hash function to map keys to indices, allowing direct access without iteration. As long as collisions are minimized, lookups, insertions, and deletions remain constant time, making it ideal for this project.