Tyler Doupe 8/7/2024

CS 300 Project One

**Pseudocode for a menu**

// Function to display a menu and prompt user for an input

FUNCTION displayMenu()

PRINT “Option 1: Load file data into the data structure”

PRINT “Option 2: Print all courses alphanumerically”

PRINT “Option 3: Print course information for specified course”

PRINT “Option 4: Exit the program”

WHILE TRUE

PRINT “Enter an option number”

INPUT choice

IF choice == 1

PRINT “Enter file path: ”

INPUT filePath

IF data structure is vector

courseVector = loadCourseFromFile(filePath)

ELSE IF data structure is hash table

courseHashTable = loadCourseFromFile(filePath)

ELSE IF data structure is tree

root = loadCourseFromFile(filePath)

END IF

ELSE IF choice == 2

IF data structure is vector

printAllCourses(courseVector)

ELSE IF data structure is hash table

printAllCourses(courseHashTable)

ELSE IF data structure is tree

printAllCourses(root)

END IF

ELSE IF choice == 3

PRINT “Enter course number: “

INPUT courseNumber

IF data structure is vector

printCourseInfo(courseVector, courseNumber)

ELSE IF data structure is hash table

printCourseInfo(courseHashTable, courseNumber)

ELSE IF data structure is tree

printCourseInfo(root, courseNumber)

END IF

ELSE IF choice == 9

BREAK

ELSE

PRINT “Invalid choice. Please input another value.”

END IF

END WHILE

END FUNCTION

**Pseudocode for Vector alphanumeric order:**

// Function to print all courses in alphanumeric order

FUNCTION printAllCourses(courseVector)

// Sort course information by alphanumeric course number from lowest to highest

SORT courseVector BY courseNumber

// Print sorted list to a display

FOR each course IN courseVector

PRINT course.courseNumber + “: “ + course.courseName

END FOR

END FUNCTION

**Pseudocode for Hash Table alphanumeric order**

// Function to print all courses in alphanumeric order

FUNCTION printAllCourses(courseHashTable)

INITIALIZE vector<STRING> keys = courseHashTable.keys()

// Sort course information by alphanumeric course number from lowest to highest

SORT keys BY key values

// Print sorted list to a display

FOR each key IN keys

course = courseHashTable[key]

PRINT course.courseNumber + “: “ + course.courseName

END FOR

END FUNCTION

**Pseudocode for Tree alphanumeric order**

// Function to print all courses in alphanumeric order

FUNCTION inOrderTraversal(root)

IF root != null

inOrderTraversal(root.left)

PRINT root.course.courseNumber + “: “ + root.course.courseName

inOrderTraversal(root.right)

END IF

END FUNCTION

// Function to print sorted list to a display

FUNCTION printAllCourses(root)

inOrderTraversal(root)

END FUNCTION

**Evaluation**

**Vector Evaluation**

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executed | Total Cost |
| INITIALIZE vector<Course> courseVector | 1 | 1 | 1 |
| OPEN file from filePath | 1 | 1 | 1 |
| WHILE not end of file | 1 | n | n |
| READ each line from file | 1 | n | n |
| PARSE line INTO tokens using commas as delimiters | 1 | n | n |
| IF length of token >= 2 tokens: | 1 | n | n |
| INTIALIZE Course course | 1 | n | n |
| SET course.courseNumber TO token[0] | 1 | n | n |
| SET course.courseName TO token[1] | 1 | n | n |
| IF length of token > 2 delimiters: | 1 | n | n |
| FOR each token from index 2 TO end: | 1 | k | k |
| ADD token TO course.prerequisite | 1 | k | k |
| ADD course to courseVector | 1 | n | n |
| CLOSE file | 1 | 1 | 1 |
| RETURN courseVector | 1 | 1 | 1 |
| **Total Cost** | | | 10n + 3k + 4 |
| **Runtime** | | | *O*(n + k) |

**Hash Table Evaluation**

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executed | Total Cost |
| INITIALIZE HashTable<STRING, Course> courseHashTable | 1 | 1 | 1 |
| OPEN file from filePath | 1 | 1 | 1 |
| WHILE not end of file | 1 | n | n |
| READ each line from file | 1 | n | n |
| PARSE line INTO tokens using commas as delimiters | 1 | n | n |
| IF length of token >= 2 tokens: | 1 | n | n |
| INTIALIZE Course course | 1 | n | n |
| SET course.courseNumber TO token[0] | 1 | n | n |
| SET course.courseName TO token[1] | 1 | n | n |
| IF length of token > 2 delimiters: | 1 | n | n |
| FOR each token from index 2 TO end: | 1 | k | k |
| ADD token TO course.prerequisite | 1 | k | k |
| ADD course to courseHashTable | 1 | n | n |
| CLOSE file | 1 | 1 | 1 |
| RETURN courseHashTable | 1 | 1 | 1 |
| **Total Cost** | | | 10n + 3k + 4 |
| **Runtime** | | | *O*(n + k) |

**BST Evaluation**

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executed | Total Cost |
| INITIALIZE TreeNode root = null | 1 | 1 | 1 |
| OPEN file from filePath | 1 | 1 | 1 |
| WHILE not end of file | 1 | n | n |
| READ each line from file | 1 | n | n |
| PARSE line INTO tokens using commas as delimiters | 1 | n | n |
| IF length of token >= 2 tokens: | 1 | n | n |
| INTIALIZE Course course | 1 | n | n |
| SET course.courseNumber TO token[0] | 1 | n | n |
| SET course.courseName TO token[1] | 1 | n | n |
| IF length of token > 2 delimiters: | 1 | n | n |
| FOR each token from index 2 TO end: | 1 | k | k |
| ADD token TO course.prerequisite | 1 | k | k |
| root = insertCourse(root, course) | O(log n) | n | n log n |
| CLOSE file | 1 | 1 | 1 |
| RETURN root | 1 | 1 | 1 |
| **Total Cost** | | | 10n + 3k + n log n + 4 |
| **Runtime** | | | *O*(n log n + k) |

**Advantages and Disadvantages of Each Data Structure**

Vector:

Vectors are considered easy to implement and understand which makes them a favorable datatype. They provide an O(1) access time when using an index and use less memory overhead. However, when searching for an element in a vector, the worst case time requirement is O(n) as searching may require the entire vector to be scanned. Inserting and deleting elements contained in a vector can also take O(n) time which also makes vectors unappealing.

Hash Table:

Hash Tables provide an O(1) complexity for inserting, deleting and searching for elements, making these actions fairly quick. With a hash table, a user can take advantage of performing a quick look up of an item with the use of a key. However, hash tables require a lot of memory overhead due to their need to hash functions and handle collisions. The worst case for a hash table is when hash collisions occur which would decrease the hash tables performance to O(n) for searching, inserting and deleting operations. Additionally, hash tables are much more complex to implement compared to a vector.

BST:

A BST can be very favorable if searching a balanced tree with a O(log n) running time complexity for insertion, deletion and search operations. It is efficient at retrieving elements in alphanumeric order with an O(n) traversal time and requires less memory overhead than a hash table. However, in order to maintain a O(log n) running time, the tree must remain balanced which can be difficult. With an unbalanced tree, the performance can degrade to O(n) for insertion, deletion and search operations. Additionally, BSTs can be more complex to implement in comparison to a vector.

**Recommendation:**

Following my evaluation and analysis of the three data structures, I would recommend implementing a hash table in the code. Hash tables provide an O(1) complexity for inserting, deleting and searching for elements which can be extremely beneficial when handling a large dataset such as all of the courses at ABCU which will likely have frequent updates to offered courses. With the implementation of a hash table, a course can be looked up extremely efficiently through the use of a key which also makes hash tables very favorable for the desired purpose at ABCU. The use of a key would avoid the need for the courses to be sorted and instead the desired course can be accessed directly. Additionally, although there is potential for hash collisions by implementing a hash table data structure, hash tables are scalable and the worst case performance is O(n). In comparison to vectors, implementing a hash table will be much more complex but will offer a more efficient runtime. Implementing a BST can offer a more efficient runtime but requires the tree to be balanced to offer this enhanced runtime which adds more complexity. In conclusion, a hash table should be considered the best suited data structure type to be implemented with the ABCU application.