**CS 300 Project One**

Class Course:

String courseNumber

String courseTitle

Vector<Course> coursesVector

Hashtable<Course> coursesHashtable

Tree<Course> coursesBinaryTree

Function InitializeDataStructure(choice):

SWITCH choice

CASE "Vector":

RETURN new Vector<Course>

CASE "Hashtable":

RETURN new Hashtable<Course>

CASE "Tree":

RETURN new Tree<Course>

DEFAULT:

PRINT "Invalid choice. Please try again."

RETURN NULL

Function loadDataStructure(dataStructure, filePath):

OPEN file at filePath

IF file is not found

PRINT "Oops! Looks like the file couldn't be found."

RETURN

ENDIF

dataStructure = InitializeDataStructure(dataStructure)

IF dataStructure is NULL

PRINT "Initialization failed. Please try again."

RETURN

ENDIF

FOR each line in file

SET courseData to each line in parsedLine

IF length of courseData is less than two

PRINT "Oops! Something's wrong with the file format."

CLOSE file

RETURN

ENDIF

SET courseNumber, courseTitle, prerequisites TO courseData

SET newCourse to createCourseObject(courseNumber, courseTitle, prerequisites)

IF dataStructure is Vector<Course>

APPEND newCourse to dataStructure

ELSE IF dataStructure is Hashtable<Course>

SET hashValue to calculateHash(courseNumber)

INSERT hashValue and newCourse into dataStructure

ELSE IF dataStructure is Tree<Course>

INSERT newCourse into dataStructure

ENDIF

ENDFOR

CLOSE file

Function loadDataIntoVector(Vector<Course> courses, String filePath):

loadDataStructure(courses, filePath)

Function loadDataIntoHashtable(Hashtable<Course> coursesHashtable, String filePath):

loadDataStructure(coursesHashtable, filePath)

Function loadDataIntoTree(Tree<Course> coursesBinaryTree, String filePath):

loadDataStructure(coursesBinaryTree, filePath)

Function printSortedCourseList(dataStructure):

IF dataStructure is Vector<Course>

CALL sortList(dataStructure)

FOR each course in dataStructure

PRINT "Course: " AND course.courseNumber AND course.courseTitle

ENDFOR

ELSE IF dataStructure is Hashtable<Course>

FOR each bucket in dataStructure

FOR each course in bucket

PRINT "Course: " AND course.courseNumber AND course.courseTitle

ENDFOR

ENDFOR

ELSE IF dataStructure is Tree<Course>

CALL printTree(dataStructure.root)

ENDIF

Function printCourse(dataStructure, courseNumber):

SET foundCourse equal to NULL

FOR each course in dataStructure

IF course.courseNumber is equal to courseNumber

SET foundCourse equal to course

BREAK

ENDIF

ENDFOR

IF foundCourse is not NULL

PRINT "Course Information:"

PRINT "Course Number: " AND foundCourse.courseNumber

PRINT "Course Title: " AND foundCourse.courseTitle

PRINT "Prerequisites:"

FOR each prerequisite in foundCourse.prerequisites

PRINT prerequisite

ENDFOR

ELSE:

PRINT "Course not found."

ENDIF

Function sortList(courses):

CALL quicksort(courses, 0, LENGTH(courses) - 1)

Function displayMenu():

PRINT "Menu:"

PRINT "1. Load Data Structure"

PRINT "2. Print Course List"

PRINT "3. Print Course"

PRINT "4. Exit"

INPUT choice

RETURN choice

Function quicksort(courses, low, high):

IF low < high

SET partitionIndex to partition(courses, low, high)

CALL quicksort(courses, low, partitionIndex - 1)

CALL quicksort(courses, partitionIndex + 1, high)

Function partition(courses, low, high):

SET pivot TO vector[low]

SET lowEndIndex TO low

SET i TO low

WHILE i is less than or equal to high

IF vector[i] is less than pivot

SWAP vector[i] AND vector[lowEndIndex]

SET lowEndIndex TO lowEndIndex + 1

ENDIF

SET i to i + 1

ENDWHILE

SWAP vector[low] AND vector[lowEndIndex]

RETURN lowEndIndex

END Function

// Main program

dataStructure = NULL

REPEAT

choice = displayMenu()

SWITCH choice

CASE 1:

PRINT "Enter file path: "

INPUT filePath

PRINT "Choose Data Structure (Vector, Hashtable, Tree): "

INPUT dataStructureChoice

SWITCH dataStructureChoice

CASE "Vector":

loadDataIntoVector(dataStructure, filePath)

CASE "Hashtable":

loadDataIntoHashtable(dataStructure, filePath)

CASE "Tree":

loadDataIntoTree(dataStructure, filePath)

DEFAULT:

PRINT "Invalid data structure choice. Please try again."

END SWITCH

CASE 2:

printSortedCourseList(dataStructure)

CASE 3:

PRINT "Enter course number: "

INPUT courseNumber

printCourse(dataStructure, courseNumber)

CASE 4:

PRINT "Exiting program."

EXIT

DEFAULT:

PRINT "Invalid choice. Please try again."

END SWITCH

UNTIL choice is 4

**Evaluation**

**Runtime Analysis:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **Vector** | **Hashtahle** | **Binary Tree** |
| Load Data | O(n) | O(n) | O(n) |
| Search Course | O(n) | O(1) - O(n) | O(log n) - O(n) |
| Sort/Print Courses | O(n log n) | O(n) | O(n) |

The runtime analysis details the given runtime complexity for each of the three data structures and their use of the given operations. Vectors are especially efficient when it comes to sorting or printing the courses since each element can be directly accessed. Hash tables often have O(1) for loading data on average, but collisions can lead to a worst-case scenario of O(n) which requires specific hashing methods to compensate. The Binary Tree, depending on how well balanced the tree is, can experience runtime for searches from O(log n) to O(n).

**Vector Structure:**

For a Vector data structure there are certain advantages to utilizing this structure, for instance, vectors are simple to use and are straightforward in terms of their complexity. They are usually a better fit for smaller data sets that don’t deal with large volumes of data that need to be parsed for a single entry. Vectors uniquely allow for direct access to an element using an index, so it does provide the benefit of quick retrieval for a specific item. However, on the downside vectors don’t compensate very well when needing to resize the size of the structure. Resizing vectors requires the entire array to be copied into a new array of the new determined size. Which is greatly inefficient for very large data sets. Also, inserting or deleting items in a vector requires shifting each element into the new location which is very time consuming.

**Hashtable:**

Hashtable structures provide for a fast average-case performance for operations including inserting, deleting, or searching for specified elements. They also allow for quick retrieval through direct access of the intended element. On the downside though, Hashtables can result in collisions when inserting new elements into the structure. This often results in linear probing or chaining which is time consuming in comparison. When there are many collisions, the runtime can usually degrade to that of O(n) which decreases in performance as the number of elements increases.

**Binary Tree:**

Binary trees are especially useful for search operations which average a runtime of (O(log n)) if a tree is well balanced. This will degrade on unbalanced trees, however. In a worst-case scenario for a Binary Tree being very unbalanced the runtime can be as bad as O(n) greatly increasing the runtime per the number of elements to search. Also, when inserting new nodes in a Binary Treed, new memory needs to be allocated when adding a new node and inserting that node in the correct order, making it not as advantageous as the other structures for insertion/deletion.

**Recommendation:**

Based on the runtime analysis chart and the advantages and disadvantages of each data structure I would recommend using a Hashtable structure for this assignment. Since accessing each course in the course structure, the Hashtable can provide quick sorting of each course while also enabling average runtime complexity for accessing the course. It may result in a worst-case scenario of O(n) for loading the course object into the structure however, but Hashtable can handle the large set of data due to its efficient use of hashing and determining where to insert the course object.