Tonya Sanders

8/11/2024

CS 300

Project One

Pseudocode:

// Load necessary text parsing libraries and headers

// Define a structure named Course to store course details

STRUCT Course

courseID : STRING

courseName : STRING

preCount : INTEGER

preList : STRING

FUNCTION Course() // Constructor to initialize the structure

courseID = ""

courseName = ""

preCount = 0

preList = ""

END FUNCTION

END STRUCT

FUNCTION Main()

// Read command-line arguments

DECLARE filePath AS STRING

SET filePath = GetCommandLineArgument()

IF filePath IS EMPTY THEN

// Load default CSV file path if no arguments are provided

SET filePath = "default/path/to/csv"

END IF

DECLARE menuChoice AS CHAR

DECLARE dataChoice AS CHAR

DECLARE courseList AS List OF Course

DECLARE courseTree AS BinaryTree

DECLARE courseTable AS HashTable

// Menu Loop

WHILE menuChoice <> '9'

// Output menu options

OutputMenu()

// Get user input for menu choice and data structure choice

SET menuChoice = GetMenuChoice()

SET dataChoice = GetDataChoice()

// Validate user input

IF menuChoice NOT IN ('1', '2', '3', '4', '9') THEN

THROW Error("Invalid choice")

END IF

IF menuChoice = '1' THEN

// Call file parser and load data into the selected data structure

IF dataChoice = 'B' THEN

courseTree = LoadBidsToTree(filePath)

ELSE IF dataChoice = 'V' THEN

courseList = LoadBidsToVector(filePath)

ELSE IF dataChoice = 'H' THEN

courseTable = LoadBidsToHashTable(filePath)

END IF

// Output number of records in the CSV file

PRINT "Records loaded: " + GetRecordCount()

ELSE IF menuChoice = '2' THEN

// Validate the list

IF dataChoice = 'B' THEN

VALIDATE ValidateTree(courseTree)

ELSE IF dataChoice = 'V' THEN

VALIDATE ValidateList(courseList)

ELSE IF dataChoice = 'H' THEN

VALIDATE ValidateTable(courseTable)

END IF

ELSE IF menuChoice = '3' THEN

// Search and print course details

DECLARE userSearch AS STRING

SET userSearch = GetUserSearchInput()

IF dataChoice = 'B' THEN

PrintCourseTree(userSearch, courseTree)

ELSE IF dataChoice = 'V' THEN

PrintCourseList(userSearch, courseList)

ELSE IF dataChoice = 'H' THEN

PrintCourseTable(userSearch, courseTable)

END IF

ELSE IF menuChoice = '4' THEN

// Print each course in alphabetical order

IF dataChoice = 'B' THEN

PrintTree(courseTree)

ELSE IF dataChoice = 'V' THEN

SortList(courseList)

PrintList(courseList)

ELSE IF dataChoice = 'H' THEN

SortTable(courseTable)

PrintTable(courseTable)

END IF

ELSE IF menuChoice = '9' THEN

// Exit the application

PRINT "Goodbye"

EXIT

END IF

END WHILE

END FUNCTION

// Define a class for a binary tree structure

CLASS BinaryTree

STRUCT Node

course : Course

right : POINTER TO Node

left : POINTER TO Node

FUNCTION Node() // Constructor to initialize the node

right = NULL

left = NULL

END FUNCTION

END STRUCT

root : POINTER TO Node

// Method to print tree details

FUNCTION PrintTree()

// Implementation to print the tree in alphabetical order

END FUNCTION

FUNCTION BinaryTree() // Constructor to initialize the binary tree

root = NULL

END FUNCTION

END CLASS

// Define a class for a hash table structure

CLASS HashTable

STRUCT Bucket

course : Course

key : STRING

next : POINTER TO Bucket

END STRUCT

hashTable : List OF Bucket

FUNCTION Hash() RETURNS INTEGER

// Implementation of hash function

END FUNCTION

FUNCTION PrintTable()

// Implementation to print the hash table in alphabetical order

END FUNCTION

END CLASS

// FUNCTION to sort a list using quicksort

FUNCTION SortList(courseList : List OF Course)

DECLARE lowIndex AS INTEGER = 0

DECLARE highIndex AS INTEGER = GetListSize(courseList) - 1

IF lowIndex >= highIndex THEN

RETURN

END IF

DECLARE pivotIndex AS INTEGER

SET pivotIndex = Partition(courseList, lowIndex, highIndex)

SortList(courseList, lowIndex, pivotIndex)

SortList(courseList, pivotIndex + 1, highIndex)

END FUNCTION

// FUNCTION to partition a list for quicksort

FUNCTION Partition(courseList : List OF Course, lowIndex : INTEGER, highIndex : INTEGER) RETURNS INTEGER

DECLARE pivot AS Course

SET pivot = GetMidPointElement(courseList, lowIndex, highIndex)

WHILE lowIndex <= highIndex

WHILE GetElement(courseList, lowIndex) < pivot

INCREMENT lowIndex

END WHILE

WHILE GetElement(courseList, highIndex) > pivot

DECREMENT highIndex

END WHILE

IF lowIndex <= highIndex THEN

SwapElements(courseList, lowIndex, highIndex)

INCREMENT lowIndex

DECREMENT highIndex

END IF

END WHILE

RETURN highIndex

END FUNCTION

// FUNCTION to print the course list

FUNCTION PrintList(courseList : List OF Course)

FOR EACH course IN courseList

PRINT "Course ID: " + course.courseID

PRINT "Course Name: " + course.courseName

FOR i = 0 TO course.preCount - 1

PRINT "Prerequisite Course: " + ExtractCourseID(course.preList, i)

END FOR

END FOR

END FUNCTION

// FUNCTION to print the binary tree

FUNCTION PrintTree(root : POINTER TO Node)

IF root = NULL THEN

RETURN

END IF

PrintTree(root.left)

PRINT "Course ID: " + root.course.courseID

PRINT "Course Name: " + root.course.courseName

FOR i = 0 TO root.course.preCount - 1

PRINT "Prerequisite Course: " + ExtractCourseID(root.course.preList, i)

END FOR

PrintTree(root.right)

END FUNCTION

// FUNCTION to print the hash table

FUNCTION PrintTable(courseTable : HashTable)

DECLARE currentBucket AS POINTER TO Bucket

FOR EACH bucket IN courseTable.hashTable

SET currentBucket = bucket

WHILE currentBucket <> NULL

PRINT "Course ID: " + currentBucket.course.courseID

PRINT "Course Name: " + currentBucket.course.courseName

FOR i = 0 TO currentBucket.course.preCount - 1

PRINT "Prerequisite Course: " + ExtractCourseID(currentBucket.course.preList, i)

END FOR

SET currentBucket = currentBucket.next

END WHILE

END FOR

END FUNCTION

Runtime Analysis:

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Vector** | **Hash Table** | **Binary Tree** |
| **Loading Data** | O(1) | O(1) – O(N) | O(log N) |
| **Search** | O(n) | O(1) – O(N) | O(log N) – O(N) |
| **Sort/Print** | O(N log N) | O(N) | O(N) |

Each data structure has unique pros and cons. For example, appending data to an unsorted vector is very fast, but the efficiency decreases substantially when sorting is required. On the other hand, hash tables can potentially achieve average performance of O(1) if they are large enough to prevent collisions. However, because time and memory are finite, some collisions are inevitable, which can cause hash table performance to range from O(1) to O(N). Binary trees typically operate at O(log N), offering consistent performance, though this depends on the structure of the data. If the tree becomes unbalanced, such as when inserting already sorted data, its efficiency can decline to O(N).

The choice of data structure should be guided by how and how often the data will be accessed. If data loading happens infrequently, the speed of the initial load might not be as critical. However, if frequent searches are needed, a hash table might be more effective than a binary tree, assuming a well-designed hash function and a balanced tree.

Binary trees also have the advantage of not requiring data to be pre-sorted, and they can be traversed in order. This could save memory when there’s no need to keep both sorted and unsorted lists. Generally, both binary trees and hash tables tend to outperform a sorted vector, making them better choices when data access efficiency is important.