Rogan Page

Ricardo Scarello

CS 300

12 Feb. 24

**CS 300 Project One**

**Updated Pseudocode:**

FUNCTION openFile(filename):

TRY:

fileHandle = OPEN(filename, "read")

RETURN fileHandle

EXCEPT FileNotFoundError:

PRINT "Error: File not found."

RETURN NULL

FUNCTION readFile(fileHandle):

courses = []

WHILE NOT EOF(fileHandle):

line = READ\_LINE(fileHandle)

courseInfo = line.split(',')

IF LEN(courseInfo) >= 2:

courses.append(courseInfo)

ELSE:

PRINT "Error: Insufficient parameters in line:", line

RETURN courses

FUNCTION parseLine(line):

courseInfo = line.split(',')

IF LEN(courseInfo) >= 2:

RETURN courseInfo

ELSE:

PRINT "Error: Insufficient parameters in line:", line

RETURN NULL

FUNCTION createCourseObject(courseInfo):

courseNumber = courseInfo[0]

courseTitle = courseInfo[1]

prerequisites = courseInfo[2:]

RETURN Course(courseNumber, courseTitle, prerequisites)

FUNCTION printCourseInformation(course):

PRINT course.courseNumber + ": " + course.courseTitle

IF course.prerequisites is not empty:

FOR EACH prerequisite IN course.prerequisites:

printCourseInformation(prerequisite)

FUNCTION displayMenu():

PRINT "1. Load Data Structure"

PRINT "2. Print Course List"

PRINT "3. Print Course"

PRINT "4. Exit"

RETURN getInput()

FUNCTION loadMenuOption():

filename = getInput("Enter filename: ")

fileHandle = openFile(filename)

IF fileHandle is not NULL:

courses = readFile(fileHandle)

RETURN courses

ELSE:

RETURN NULL

FUNCTION printCourseListMenuOption(courses):

sortedCourses = sortCourses(courses)

FOR EACH course IN sortedCourses:

PRINT course.courseNumber + ": " + course.courseTitle

FUNCTION printCourseMenuOption(courses):

courseNumber = getInput("Enter course number: ")

course = findCourseByNumber(courseNumber, courses)

IF course is not NULL:

printCourseInformation(course)

ELSE:

PRINT "Error: Course not found."

FUNCTION exitMenuOption():

PRINT "Exiting program."

EXIT PROGRAM

FUNCTION sortCourses(courses):

RETURN sorted(courses, key=lambda x: x.courseNumber)

FUNCTION printSortedCourseList(courses):

sortedCourses = sortCourses(courses)

FOR EACH course IN sortedCourses:

PRINT course.courseNumber + ": " + course.courseTitle

FUNCTION main():

courseTree = new BinarySearchTree()

WHILE TRUE:

choice = displayMenu()

IF choice == 1:

courses = loadMenuOption()

IF courses is not NULL:

FOR EACH courseInfo IN courses:

IF length(courseInfo) >= 2:

courseObject = createCourseObject(courseInfo)

prerequisitesExist = TRUE

FOR EACH prerequisite IN courseObject.prerequisites:

IF NOT courseTree.contains(prerequisite):

prerequisitesExist = FALSE

EXIT FOR

IF prerequisitesExist:

courseTree.insert(courseObject)

ELSE:

PRINT "Error: Prerequisite not found for course " + courseObject.courseNumber

ELSE:

PRINT "Error: Insufficient parameters in line: " + line

PRINT "Course data loaded successfully."

ELSE IF choice == 2:

PRINT "Printing Course List:"

printSortedCourseList(courseTree)

ELSE IF choice == 3:

PRINT "Printing Course Information:"

courseNumber = getInput("Enter course number: ")

courseNode = courseTree.search(courseNumber)

IF courseNode is not NULL:

printCourseInformation(courseNode)

ELSE:

PRINT "Error: Course not found."

ELSE IF choice == 4:

exitMenuOption()

ELSE:

PRINT "Invalid choice. Please try again."

CALL main()

**Data Structure Analysis and Recommendation for ABCU's Advising Program**

In designing a data structure for ABCU's advising program, it is crucial to analyze the advantages and disadvantages of each potential option: vector, hash table, and tree. Each structure offers unique characteristics that can impact performance and efficiency in addressing the program's requirements.

**Vector:** Vectors provide simplicity and random access to elements, making them suitable for small to medium-sized datasets. However, their inefficiency in insertion and deletion operations (O(n)) and the requirement for contiguous memory allocation limit their suitability for large datasets. Additionally, sorting vectors incurs a performance cost (O(n log n)), which may affect responsiveness in sorting course data.

**Hash Table:** Hash tables offer fast insertion and retrieval operations (O(1) on average) and constant-time access to elements. They are well-suited for large datasets and can handle collisions through techniques like chaining or open addressing. However, maintaining sorted order in a hash table requires additional effort, and performance degradation can occur with high collision rates or poor hash functions.

**Tree:** Trees automatically maintain sorted order and are efficient for searching and printing in sorted order (O(log n) for search and traversal). While suitable for ordered data, trees exhibit slower insertion and deletion compared to hash tables (O(log n)), and memory overhead due to pointer-based structure may be a concern. Unbalanced trees can lead to performance degradation in worst-case scenarios.

**Recommendation:** Considering the requirements of ABCU's advising program, the hash table data structure emerges as the most suitable choice. Hash tables provide fast insertion and retrieval operations crucial for loading data and accessing course information. Despite the need for additional effort to maintain alphanumeric order, this can be addressed with sorting functions leveraging the hash table's fast retrieval capability. With a balance of performance and simplicity, the hash table aligns well with the program's needs.

In conclusion, the hash table data structure stands out as the optimal choice for ABCU's advising program, offering efficient operations for data loading, retrieval, and sorting. Its strengths in handling large datasets and maintaining constant-time access make it an ideal solution for the program's requirements.

By leveraging the advantages of hash tables, ABCU's advising program can achieve optimal performance and responsiveness, enhancing the efficiency of academic advising processes.