Tanner Meininger

April 8, 2024

PROJECT ONE

VECTOR DATA STRUCTURE

```
STRUCT Course:
    STRING courseNumber
    STRING courseTitle
    VECTOR<string> prerequisites


FUNCTION validateFileFormat(file):
    OPEN file
        IF file does not exist THEN
        PRINT "Error: File does not exist."
        RETURN false
    END IF
    FOR each line in file:
        IF line does not contain at least two parameters THEN
            PRINT "Error: Incorrect file format. Each line must contain at least two parameters."
            RETURN false
        END IF
        IF second parameter on the line does not exist as a course in the file THEN
            PRINT "Error: Prerequisite not found for course."
            RETURN false
        END IF
    END FOR
    CLOSE file
    RETURN true
```

```
FUNCTION loadDataFromFile(file, courses):
    OPEN file
    IF file does not exist THEN
        PRINT "Error: File does not exist."
        RETURN
    END IF
    FOR each line in file:
        courseData = SPLIT line by comma
        courseNumber = courseData[0]
        courseTitle = courseData[1]
        prerequisites = EMPTY LIST
        FOR i from 2 to length of courseData - 1:
            prerequisites.APPEND(courseData[i])
        END FOR
        course = CREATE Course object with courseNumber, courseTitle, and prerequisites
        courses.ADD(course) // Add course object to vector data structure
    END FOR
    CLOSE file

FUNCTION searchCourse(courses, courseNumber):
    FOR each course in courses:
        IF course.courseNumber is equal to courseNumber THEN
            PRINT "Course Number:", course.courseNumber
            PRINT "Course Title:", course.courseTitle
            PRINT "Prerequisites:"
                IF course.prerequisites is not empty THEN
                    FOR each prerequisite in course.prerequisites:
```

```
                                    PRINT prerequisite
                              END FOR
                        ELSE:
                              PRINT "None"
                        END IF
                        RETURN
            END IF
      END FOR
      PRINT "Course not found."

FUNCTION displayMenu():
  PRINT "Menu:"
  PRINT "1. Load Data Structure"
  PRINT "2. Print Course List"
  PRINT "3. Print Course"
  PRINT "9. Exit"
  PRINT "Enter your choice:"

FUNCTION printSortedCourseList(courses):
  SORT courses by courseNumber
  FOR each course in courses:
    PRINT "Course Number:", course.courseNumber
    PRINT "Course Title:", course.courseTitle
    IF course.prerequisites is not empty:
      PRINT "Prerequisites:"
      FOR each prerequisite in course.prerequisites:
        PRINT prerequisite
    ELSE:
            PRINT "None"

// Main program
DECLARE courses as VECTOR of Course objects

REPEAT:
    displayMenu()
    READ choice
    SWITCH choice:
        CASE 1:
            // Prompt user for data file
            PRINT "Enter the path of the course data file:"
            READ filename
            IF validFile(filename) THEN
                LOAD_DATA(filename)
            ELSE:
                PRINT "Invalid file. Please try again."
            END IF
            BREAK
        CASE 2:
            printSortedCourseList(courses)
            BREAK
        CASE 3:
            PRINT "Enter course number:"
            READ courseNumber
            searchCourse(courses, courseNumber)
            BREAK
        CASE 9:
            PRINT "Exiting program."
            EXIT
        DEFAULT:
            PRINT "Invalid choice. Please choose again."
```

HASH TABLE DATA STRUCUTRE

```
STRUCT Course:
    STRING courseNumber
    STRING courseTitle
    VECTOR<string> prerequisites


FUNCTION readAndValidateFile(filename):
    OPEN the file with the given filename
    IF the file does not exist:
        PRINT "Error: File not found."
        RETURN
    FOR each line in the file:
        SPLIT the line into tokens using comma as delimiter
        IF the number of tokens is less than 2:
            PRINT "Error: Invalid line format."
            CONTINUE to the next line
        courseNumber = tokens[0]
        courseName = tokens[1]
        IF the number of tokens is greater than 2:
            FOR each prerequisite in tokens[2:]:
                IF prerequisite does not exist in the course data:
                    PRINT "Error: Prerequisite", prerequisite, "not found."
                    CONTINUE to the next line
    CLOSE the file

FUNCTION parseAndStoreCourses(filename, hashTable):
    OPEN the file with the given filename
    FOR each line in the file:
        SPLIT the line into tokens using comma as delimiter
        courseNumber = tokens[0]
        courseName = tokens[1]
        prerequisites = tokens[2:] // if any

        CREATE a new Course object with courseNumber, courseName, and prerequisites
        hashTable.insert(courseNumber, new Course)
    CLOSE the file

FUNCTION printCourseList(hashTable):
    SORT keys of hashTable in alphanumeric order
    FOR each key in sorted keys:
        PRINT "Course Number:", key
        PRINT "Course Title:", hashTable.lookup(key).courseTitle
        IF hashTable.lookup(key).prerequisites is not empty:
            PRINT "Prerequisites:"
            FOR each prerequisite in hashTable.lookup(key).prerequisites:
                PRINT prerequisite
        ELSE:
            PRINT "None"
```

```
FUNCTION displayMenu():
    PRINT "Menu:"
    PRINT "1. Load Data Structure"
    PRINT "2. Print Course List"
    PRINT "3. Print Course"
    PRINT "9. Exit"
    PRINT "Enter your choice:"

// Main program
DECLARE hashTable as HASH TABLE with Course objects

REPEAT:
    displayMenu()
    READ choice
    SWITCH choice:
        CASE 1:
            // Prompt user for data file
            PRINT "Enter the path of the course data file:"
            READ filename
            IF validFile(filename) THEN
                LOAD_DATA(filename)
            ELSE:
                PRINT "Invalid file. Please try again."
            END IF
            BREAK
        CASE 2:
            printCourseList(hashTable)
            BREAK
```

```
CASE 3:
    PRINT "Enter course number:"
    READ courseNumber
    PRINT "Course Number:", courseNumber
    PRINT "Course Title:", hashTable.lookup(courseNumber).courseTitle
    IF hashTable.lookup(courseNumber).prerequisites is not empty:
        PRINT "Prerequisites:"
        FOR each prerequisite in hashTable.lookup(courseNumber).prerequisites:
            PRINT prerequisite
    ELSE:
        PRINT "No prerequisites"
    BREAK
CASE 9:
    PRINT "Exiting program."
    EXIT
DEFAULT:
    PRINT "Invalid choice. Please choose again."
```

TREE DATA STRUCTURE

```
STRUCT Course:
    STRING courseNumber
    STRING courseTitle
    VECTOR<string> prerequisites

STRUCT TreeNode:
    Course course
    TreeNode leftChild
    TreeNode rightChild

STRUCT Tree:
    TreeNode root

FUNCTION readAndValidateFile(filename):
    OPEN the file with the given filename
    IF the file does not exist:
        PRINT "Error: File not found."
        RETURN false
    FOR each line in the file:
        SPLIT the line into tokens using comma as delimiter
        IF the number of tokens is less than 2:
            PRINT "Error: Invalid line format."
            CLOSE the file
            RETURN false
        courseNumber = tokens[0]
        courseTitle = tokens[1]
        IF the number of tokens is greater than 2:
```

```
        FOR each prerequisite in tokens[2:]:
            IF prerequisite does not exist in the course data:
                PRINT "Error: Prerequisite", prerequisite, "not found."
                CLOSE the file
                RETURN false
    CLOSE the file
    RETURN true

FUNCTION parseAndStoreCourses(filename, tree):
    OPEN the file with the given filename
    FOR each line in the file:
        SPLIT the line into tokens using comma as delimiter
        courseNumber = tokens[0]
        courseTitle = tokens[1]
        prerequisites = tokens[2:] // if any

        CREATE a new Course object with courseNumber, courseTitle, and prerequisites
        newNode = CREATE TreeNode with course
        tree.root = INSERT(tree.root, newNode)
    CLOSE the file

FUNCTION insert(root, newNode):
    IF root is NULL:
        RETURN newNode
    IF newNode.course.courseNumber < root.course.courseNumber:
        root.leftChild = INSERT(root.leftChild, newNode)
    ELSE:
        root.rightChild = INSERT(root.rightChild, newNode)
    RETURN root

FUNCTION printSortedCourseList(root):
    INORDER_TRAVERSE(root)

FUNCTION inorderTraverse(root):
    IF root is NULL:
        RETURN

    inorderTraverse(root.leftChild)
    PRINT "Course Number:", root.course.courseNumber
    PRINT "Course Title:", root.course.courseTitle
    IF root.course.prerequisites is not empty:
        PRINT "Prerequisites:"
        FOR each prerequisite in root.course.prerequisites:
            PRINT prerequisite
    ELSE:
        PRINT "None"

    inorderTraverse(root.rightChild)

FUNCTION displayMenu():
    PRINT "Menu:"
    PRINT "1. Load Data Structure"
    PRINT "2. Print Course List"
    PRINT "3. Print Course"
    PRINT "9. Exit"
    PRINT "Enter your choice:"
```

```
// Main program
DECLARE tree as Tree

REPEAT:
    displayMenu()
    READ choice
    SWITCH choice:
        CASE 1:
            // Prompt user for data file
            PRINT "Enter the path of the course data file:"
            READ filename
            IF validFile(filename) THEN
                LOAD_DATA(filename)
            ELSE:
                PRINT "Invalid file. Please try again."
            END IF
            BREAK
        CASE 2:
            printSortedCourseList(tree.root)
            BREAK
        CASE 3:
            PRINT "Enter course number:"
            READ courseNumber
            inorderTraverse(tree.root, courseNumber)
            BREAK
        CASE 9:
            PRINT "Exiting program."
            EXIT
        DEFAULT:
            PRINT "Invalid choice. Please choose again."
```

EVALUATION

EVALUATION OF RUNTIME AND MEMORY

Cost per Line:

1 (Unless calling a function)

Function Cost: Considered constant compared to n (number of courses)

Vector Data Structure:

Open/Close File: 1 (executed once)

Loop: n (executed once for each line)

Inside Loop:

Split line: n (executed once per line) - Assuming constant cost for line length

Create Course Object: 1 (constant cost)

Total Cost: 1 + n(1 + n) = O(n^2)

Hash Table Data Structure:

Open/Close File: 1 (executed once)

Loop: n (executed once for each line)

Inside Loop:

Split line: n (executed once per line) - Assuming constant cost for line length

Create Course Object: 1 (constant cost)

Hash Table Insert: 1 (amortized constant time for most hash tables)

Total Cost: 1 + n(1 + 1) = O(n)

Tree Data Structure:

Open/Close File: 1 (executed once)

Loop: n (executed once for each line)

Inside Loop:

Split line: n (executed once per line) - Assuming constant cost for line length

Create Course Object: 1 (constant cost)

Tree Insert: log(n) (average for balanced trees)

Total Cost: 1 + n(1 + log(n)) = O(n log n)

ANALYSIS OF DATA STRUCTURES

Vector:

Advantages:

- Simple implementation.
- Random access to elements.
- Can be efficient for sorting courses (O(n log n) using algorithms like Merge Sort or Quicksort) if frequent sorted printing is required.

Disadvantages:

- Slow for searching courses (linear search - O(n))
- Inefficient for adding/removing courses in the middle (requires shifting elements).
- Becomes memory-intensive for large datasets.
- Sorting overhead (O(n log n)) can be a factor if sorting becomes frequent.

Hash Table:

Advantages:

- Fast average-case search for courses (O(1))
- Efficient for adding/removing courses.

Disadvantages:

- Hash collisions can occur, potentially impacting search time.
- Requires additional memory overhead for storing hash table itself.
- Does not inherently maintain a sorted order. Printing courses in sorted order requires iterating through the hash table, which might not have a guaranteed time complexity.

Tree:

Advantages:

- Efficient for searching sorted courses (O(log n))
- Good for keeping courses sorted.
- Maintains a sorted order by its nature, making sorted printing efficient through in-order traversal (O(n)).

Disadvantages:

- More complex to implement compared to vector or hash table.
- May require balancing operations to maintain efficiency for insertions/deletions.

RECOMMENDATION

Based on the analysis and considering the requirements of the advisor's program, the hash table data structure seems to be the best choice. The program spends most of its time reading the file and creating course objects, and while the vector data structure has a worse Big O for this step (O(n^2) vs O(n) for hash table), this difference is less significant compared to the ongoing operations. The program also needs to efficiently search for courses by course number. Hash tables offer a significant advantage here with an average-case search time of O(1) compared to the linear search (O(n)) required for the vector. Adding and removing courses is not a frequent operation, which means that while a vector might be slightly slower for this, the benefit of faster searching outweighs this minor drawback. For the reasons above, a hash table provides the best balance and is the greatest choice for this project.