Jarrod Helmers

CS-300

Dr. Webb

14 April, 2024

Module Six: Project One

// Reading File and Loading Data Structures:

// Reading File

Use fstream to open file

Create function void loadCourses(string csvPath, dataStructure)

Call to open file, if return value is -1, file not found

Else file found

While not End Of File

Read each line

IF Less than two values in a line, return ERROR

ELSE read parameters

IF third or more parameter

IF third or more parameter is in first parameter elsewhere, continue

ELSE return Error

Close file

// Vector

BEGIN:

PROCEDURE OpenFile(filePath):

TRY:

OPEN file at filePath

IF successful THEN RETURN file handle ELSE RETURN NULL

CATCH IOError:

PRINT "Error: Unable to open file"

RETURN NULL

END TRY

END PROCEDURE

PROCEDURE ParseFile(fileHandle):

courseDataList = EMPTY LIST

FOR EACH line IN fileHandle:

IF line HAS AT LEAST TWO PARAMETERS THEN

courseDataList.APPEND line

END IF

END FOR

RETURN courseDataList

END PROCEDURE

PROCEDURE CheckPrerequisites(courseDataList):

courseMap = EMPTY MAP

FOR EACH courseData IN courseDataList:

courseId, courseTitle, prerequisites = PARSE courseData

courseMap[courseId] = (courseTitle, prerequisites)

END FOR

RETURN courseMap

END PROCEDURE

PROCEDURE PrintCourseList(courseMap):

FOR EACH courseId IN SORTED courseMap KEYS:

PRINT "Course:", courseId, "Title:", courseMap[courseId][0], "Prerequisites:", courseMap[courseId][1]

END FOR

END PROCEDURE

PROCEDURE Main():

fileHandle = OpenFile("course\_data.txt")

IF fileHandle IS NOT NULL THEN

courseDataList = ParseFile(fileHandle)

IF courseDataList IS NOT EMPTY THEN

courseMap = CheckPrerequisites(courseDataList)

PrintCourseList(courseMap)

END IF

END IF

END PROCEDURE

END.

// Hash Table

BEGIN:

Class HashTable:

DATA: nodes (vector of Node structs), tableSize (unsigned int)

METHOD Constructor(size): Initialize nodes vector with size, tableSize to size

METHOD Insert(course): Insert course into hash table

METHOD PrintCourseList(): Print all courses stored in the hash table

Class Node:

DATA: course (Course struct), key (unsigned int), next (pointer to Node)

METHOD Constructor(course, key): Initialize course, key, next to nullptr

Pseudocode for Insert(course) Method:

METHOD Insert(course):

key = CALCULATE\_HASH(course.courseId)

node = nodes[key]

IF node IS NULL THEN

nodes[key] = NEW Node(course, key)

ELSE:

WHILE node.next IS NOT NULL:

node = node.next

node.next = NEW Node(course, key)

END IF

END METHOD

END.

// Binary Search Tree

BEGIN:

STRUCT: Course (courseId, courseTitle, prerequisites)

STRUCT: Node (Course, left, right)

Class BinarySearchTree:

DATA: root (pointer to Node)

METHOD Constructor(): Initialize root to nullptr

METHOD Insert(course): Insert course into binary search tree

METHOD PrintCourseList(): Print all courses stored in the binary search tree

METHOD PrintCourse(courseId): Print course details for a given courseId

Pseudocode for Insert(course) Method:

METHOD Insert(course):

IF root IS NULL THEN

root = NEW Node(course)

ELSE:

INSERT\_RECURSIVELY(root, course)

END IF

END METHOD

Pseudocode for INSERT\_RECURSIVELY(node, course) Method:

METHOD INSERT\_RECURSIVELY(node, course):

IF courseId < node.course.courseId THEN

IF node.left IS NULL THEN

node.left = NEW Node(course)

ELSE:

INSERT\_RECURSIVELY(node.left, course)

END IF

ELSE:

IF node.right IS NULL THEN

node.right = NEW Node(course)

ELSE:

INSERT\_RECURSIVELY(node.right, course)

END IF

END IF

END METHOD

END.

// Printing Course Information:

// Print Course Information

void printCourseInformation(Course course)

Print course ID, name, and prerequisites if available

// Vector

void printCourseList(vector<Course>& courses)

For each course in courses

Print course information

// Hash Table

void printCourseList(HashTable\* hashTable)

For each course in hashTable

Print course information

// Tree

void printCourseList(BinarySearchTree\* bst)

Traverse the tree in order

Print course information

// Menu:

// Menu

void displayMenu()

Print menu options:

1. Load Data Structure

2. Print Course List

3. Print Course

4. Exit

// Sorting and Printing Alphanumeric Course List:

// Sorting and Printing Alphanumeric Course List

// Vector

void sortAndPrintCourseList(vector<Course>& courses)

Sort courses by course ID in alphanumeric order

Print the sorted list of courses

// Hash Table

void sortAndPrintCourseList(HashTable\* hashTable)

Extract all courses from hashTable and store them in a vector

Sort courses by course ID in alphanumeric order

Print the sorted list of courses

// Tree

void sortAndPrintCourseList(BinarySearchTree\* bst)

Traverse the tree in order and store courses in a vector

Sort courses by course ID in alphanumeric order

Print the sorted list of courses

| **Operation** | **Vector** | **Hash Table** | **Tree** |
| --- | --- | --- | --- |
| **Open File (1 time)** | **O(1)** | **O(1)** | **O(1)** |
| **Read Line (m times)** | **O(m)** | **O(m)** | **O(m)** |
| **Parse Parameters (m times)** | **O(m)** | **O(m)** | **O(m)** |
| **Additional Parameter Check (m times)** | **O(m)** | **O(m)** | **O(m)** |
| **Total** | **O(m)** | **O(m)** | **O(m \* log n)** |

**Vector**:

Opening the file: O(1)

Reading each line from the file (n lines): O(n)

Parsing each line and creating a course object: O(n)

Inserting the course object into the vector: O(1) (amortized)

Total Runtime: O(n)

Advantages:

Straightforward implementation.

Fast access by index.

Disadvantages:

Slower insertion and deletion times for large datasets due to shifting elements.

**Hash Table**:

Opening the file: O(1)

Reading each line from the file (n lines): O(n)

Parsing each line and creating a course object: O(n)

Inserting the course object into the hash table: O(1) (average case)

Total Runtime: O(n)

Advantages:

Fast average-case insertion, deletion, and lookup times.

Efficient for large datasets.

Disadvantages:

The worst-case time complexity for operations can be high if collisions occur frequently.

Requires tuning for optimal performance.

**Tree (Binary Search Tree)**:

Opening the file: O(1)

Reading each line from the file (n lines): O(n)

Parsing each line and creating a course object: O(n)

Inserting the course object into the binary search tree: O(log n) on average, O(n) in worst case (unbalanced)

Total Runtime: O(n log n) on average (assuming balanced tree), O(n^2) in worst case (unbalanced)

Advantages:

Efficient for sorted data retrieval.

Can maintain a sorted order without additional sorting operations.

Disadvantages:

If the tree becomes unbalanced, it can degrade to O(n) time complexity for operations.

Requires additional memory for pointers and node structure

After analyzing the task of reading a file, parsing data, and creating course objects, I've found that both the vector and hash table offer linear time complexity (O(n)). However, I recommend using a vector for this task due to its simplicity of implementation and the absence of specific requirements for fast lookup operations. It provides a straightforward solution with acceptable runtime complexity and minimal memory overhead. If fast lookup operations become a priority in the future, a hash table might be a better choice.