Christian Foster

April 13, 2024

Assignment 6-2

Professor Jamaliannasrabadi

**Vector**

Procedure ReadDataFromFile(filePath):

Try:

Open the file at filePath

If unable to open file:

Print "Error: Unable to open file."

Exit procedure

Initialize an empty vector to store Course objects

For each line in the file:

Parse line into courseNumber, courseTitle, and prerequisites

If line parameters < 2:

Print "Error: Invalid file format."

Continue to next line

Create new Course object with parsed data

Add Course object to the vector

Close the file

Catch Exception as e:

Print "Error: Unable to open or read file."

Exit procedure

End Procedure

Procedure DisplayCourseInfo(courseList, targetCourseNumber):

foundCourse = false

For each course in courseList:

If course.courseNumber is equal to targetCourseNumber:

foundCourse = true

Print "Course Number:", course.courseNumber

Print "Course Title:", course.courseTitle

If Length of course.prerequisites > 0:

Print "Prerequisites:"

For each prerequisite in course.prerequisites:

Print " - ", prerequisite

Else:

Print "No prerequisites"

Break from loop

If foundCourse is false:

Print "Course not found."

End Procedure

Function IsCourseNumberInList(targetCourseNumber, courseList):

For each course in courseList:

If course.courseNumber equals targetCourseNumber:

Return true

Return false

End Function

| **Vector** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| for each line in the file | 1 | n | n |
| Parse line into courseNumber, courseTitle, etc. | 1 | n | n |
| Create new Course object with parsed data | 1 | n | n |
| Add Course object to the vector | 1 | n | n |
| If unable to open file | 1 | 1 | 1 |
| If line parameters < 2 | 1 | n | n |
| If Length of course.prerequisites > 0 | 1 | n | n |
| For each prerequisite in course.prerequisites | 1 | n | n |
| If foundCourse is false | 1 | 1 | 1 |
| For each course in courseList | 1 | n | n |
| If course.courseNumber equals targetCourseNumber | 1 | n | n |
| Print course information | 1 | 1 | 1 |
| For each prerequisite in course.prerequisites | 1 | n | n |
| If course.courseNumber equals targetCourseNumber | 1 | n | n |
| Total Cost: |  |  | 8n + 5 + max(n) |
| Runtime: |  |  | O(n) |

**Hash Tables**

FUNCTION openFile(filename):

file = open(filename, 'r')

RETURN file

EXCEPT FileNotFoundError:

PRINT "Error: File not found."

EXIT program

FUNCTION parseCourseData(line):

tokens = SPLIT line BY delimiter

IF LENGTH of tokens < 2:

PRINT "Error: Insufficient parameters in line."

EXIT program

course\_number = tokens[0]

course\_title = tokens[1]

IF LENGTH of tokens > 2:

prerequisites = tokens[2:]

ELSE:

prerequisites = empty list

RETURN course\_number, course\_title, prerequisites

FUNCTION loadDataIntoHashTable(file):

hashTable = empty hash table

FOR EACH line IN file:

course\_number, course\_title, prerequisites = parseCourseData(line)

hashTable[course\_number] = {'title': course\_title, 'prerequisites': prerequisites}

FUNCTION printCourseInformation(hashTable):

FOR EACH course\_number, course\_info IN hashTable:

PRINT "Course Number:", course\_number

PRINT "Title:", course\_info['title']

PRINT "Prerequisites:", course\_info['prerequisites']

| **Hash Table** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Open the file | 1 | 1 | 1 |
| Parse line into courseNumber, courseTitle, etc. | 1 | n | n |
| Split line by delimiter | 1 | n | n |
| If LENGTH of tokens < 2 | 1 | n | n |
| If LENGTH of tokens > 2 | 1 | n | n |
| Create new Course object with parsed data | 1 | n | n |
| Add Course object to the hash table | 1 | n | n |
| Print course information | 1 | n | n |
| Accessing course\_info['title'] | 1 | n | n |
| Accessing course\_info['prerequisites'] | 1 | n | n |
| Total Cost: |  |  | 9n + 1 |
| Runtime: |  |  | O(n) |

**Tree**

START PROGRAM

DEFINE method loadCourses(string csvPath, BinarySearchTree \*bst)

DECLARE ifstream variable courseFile // will be used to open and read the CSV file

courseFile.open(csvPath)

WHILE courseFile is open

DECLARE string variable line // used to store data as file is parsed

READ line from courseFile using getline(courseFile, line, ',')

PRINT line

DEFINE structure to hold course data

Struct Course

DECLARE courseID variable // stores the course ID

DECLARE title variable // stores the name of the course

DECLARE preReq1 variable // stores the first prerequisite

DECLARE preReq2 variable // stores the second prerequisite

DECLARE preReqCount variable // stores the number of prerequisites

DEFINE structure of a Node

DECLARE course variable of type Course

DECLARE Node \*left

DECLARE Node \*right

DEFINE default constructor for Node

DECLARE Node() method

INITIALIZE left to null pointer

INITIALIZE right to null pointer

DEFINE constructor for Node that takes a course as parameter

Node(Course aCourse) : Node() // delegation of Node() constructor

SET this->course equal to aCourse

DEFINE class BinarySearchTree

DEFINE private data members

PRIVATE:

DECLARE root node pointer

DEFINE public data members

PUBLIC:

INVOKE constructor BinarySearchTree()

INVOKE destructor ~BinarySearchTree()

INVOKE InOrder() method of type void

INVOKE Insert(Course course) method of type void and pass in course

INVOKE Remove(Course course) method of type void and pass in course

INVOKE Search() method of type Course and pass in courseID

DEFINE constructor BinarySearchTree()

SET root node equal to null pointer

DEFINE destructor ~BinarySearchTree()

INVOKE DestroyRecursive(root) method and pass in root node

DEFINE InOrder()

INVOKE inOrder(root) method and pass in root node

DEFINE Insert(Course course) method and pass in course

IF root equals null pointer

SET root equal to new Node(course) and pass in course

DEFINE Remove() method and pass in courseID

INVOKE removeNode(root, courseID) method and pass in root and courseID

DEFINE Search() method of type Course and pass in courseID

DECLARE current node pointer

SET current equal to root

WHILE current is not equal to null pointer

IF current courseID is equal to courseID

RETURN current course

IF courseID is less than current courseID

SET current equal to current left node

ELSE

SET current equal to current right node

RETURN course

DEFINE addNode(node, course) method and pass in node and course

IF node courseID is greater than courseID

IF node left is equal to null pointer

SET node left equal to new Node(course) and pass in course

ELSE

INVOKE addNode(node->left, course) method and pass in left node and course

ELSE

IF node right is equal to null pointer

SET node right equal to new Node(course) and pass in course

ELSE

INVOKE addNode(node->right, course) method and pass in right node and course

DEFINE inOrder(node) method and pass in node

IF node is not equal to null pointer

INVOKE inOrder(node->left) // traverse the left subtree - recursive

PRINT courseID, title, preReq1, preReq2

INVOKE inOrder(node->right) // traverse the right subtree - recursive

DEFINE removeNode(node, courseID) method and pass in node and courseID

IF node is equal to null pointer

RETURN node

IF courseID is less than node courseID

SET node left equal to removeNode(node left, courseID)

ELSE IF courseID is greater than node courseID

SET node right equal to removeNode(node right, courseID)

ELSE // node has no children

IF node left and node right are null pointers

DELETE node

node is set to null pointer

ELSE IF node left is not null and node right is null

INITIALIZE temp node equal to node

SET node equal to node left

DELETE temp node

ELSE IF node left is null and node right is not null

INITIALIZE temp node equal to node

SET node equal to node right

DELETE temp node

ELSE // node has two children

INITIALIZE temp node equal to node right

WHILE temp left is not null pointer

INITIALIZE temp equal to temp left

SET node course equal to temp course

SET node right equal to removeNode(node right, temp courseID)

RETURN node

DEFINE DestroyRecursive(node) method and pass in node

IF root is not null pointer

INVOKE DestroyRecursive(node->left) method and pass in left node

INVOKE DestroyRecursive(node->right) method and pass in right node

DELETE root

DEFINE displayCourse(Course course) method and pass in course

PRINT courseID, title, preReq1, preReq2

END PROGRAM

| **Binary Tree** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Open the file | 1 | 1 | 1 |
| Read line from file | 1 | n | n |
| Declare structure to hold course data | 1 | 1 | 1 |
| Declare structure of a Node | 1 | 1 | 1 |
| Define default constructor for Node | 1 | 1 | 1 |
| Define constructor for Node | 1 | n | n |
| Define class BinarySearchTree | 1 | 1 | 1 |
| Define private data members | 1 | 1 | 1 |
| Define public data members | 1 | 1 | 1 |
| Define constructor BinarySearchTree | 1 | 1 | 1 |
| Define destructor BinarySearchTree | 1 | 1 | 1 |
| Define InOrder method | 1 | 1 | 1 |
| Define Insert method | 1 | n | n |
| Define Remove method | 1 | n | n |
| Define Search method | 1 | n | n |
| Define addNode method | 1 | n | n |
| Define inOrder method | 1 | n | n |
| Define removeNode method | 1 | n | n |
| Define DestroyRecursive method | 1 | 1 | 1 |
| Display course information | 1 | n | n |
| Print course information | 1 | n | n |
| Total Cost |  |  | 13n + 22 |
| Runtime |  |  | O(n) |

**Advantages and Disadvantages**:

Let's analyze each data structure (vector, hash table, and tree) based on the requirements and their characteristics:

**Vector:**

**Advantages:**

Simple to use and understand.

Provides constant time access to elements by index.

Suitable for scenarios where the number of elements is known and doesn't change frequently.

**Disadvantages:**

Insertion and deletion operations can be expensive, especially when performed frequently or in the middle of the vector.

Resizing the vector can be costly if it exceeds its capacity.

Searching for specific elements may require linear time complexity.

**Hash Table:**

**Advantages:**

Provides constant-time average-case access to elements.

Efficient for both insertion and lookup operations.

Can handle many elements without significant performance degradation.

**Disadvantages:**

Hash collisions can occur, leading to performance degradation if not handled properly.

Difficult to maintain order and perform range queries.

Requires a good hash function to distribute keys evenly across the table.

**Tree (Binary Search Tree):**

**Advantages:**

Provides efficient search, insertion, and deletion operations with logarithmic time complexity on average.

Maintains elements in sorted order, allowing for efficient range queries.

Can be easily balanced to ensure optimal performance in terms of height.

**Disadvantages:**

It is unsuitable for scenarios with a skewed distribution of elements, as it may lead to degenerate trees with poor performance.

Insertion and deletion operations can be more complex compared to vectors and hash tables.

Requires additional memory overhead for maintaining left and right pointers.

Based on the requirements and the analysis of the three data structures, the recommendation would be to use a hash table for storing the course data.

**Recommendation:**

Hash tables offer constant-time (O(1)) average-case complexity for insertion, deletion, and lookup operations, ensuring consistent performance regardless of data size. In contrast, binary search trees, while having O(log n) complexity on average, can degrade to O(n) in the worst-case scenario, impacting efficiency. When it comes to data retrieval, hash tables excel, providing direct access to values through hash functions, unlike binary search trees which may suffer from degraded performance if unbalanced. Additionally, hash tables are simpler to implement and manage compared to binary search trees, as they do not require balancing. Though hash tables may consume more memory due to potential collisions and hash function requirements, they remain efficient, particularly for smaller datasets.

Overall, given these factors, a hash table seems to be the most suitable choice for the provided code. It offers efficient operations for storing and accessing course information, which aligns well with the requirements of the tasks described in the code.