**Vector Data Structure**   
  
 Complete Pseudocode with Menu and Vector Data Structure

Structure Course

number as string

title as string

prerequisites as vector of string

Main Procedure

Declare courses as vector of Course

Declare choice as integer

While True

Print "Menu Options:"

Print "1. Load Data Structure"

Print "2. Print Course List"

Print "3. Print Course"

Print "4. Exit"

Input choice

If choice == 1

LoadAndValidateCourses("courseData.csv", courses)

ElseIf choice == 2

PrintCourseList(courses)

ElseIf choice == 3

Print "Enter course number: "

Input courseNumber

PrintCourseInformation(courseNumber, courses)

ElseIf choice == 4

Print "Exiting program."

Exit Main Procedure

Else

Print "Invalid option, please try again."

Procedure LoadAndValidateCourses(fileName, courses)

Open fileName for reading

If file is not open

Print "Error opening file"

Return

While not end of file

Read line from file

Split line into tokens based on comma delimiter

If tokens size is less than 2

Print "Error: Each line must have at least a course number and title."

Continue to next iteration

Declare newCourse as Course

newCourse.number = tokens[0]

newCourse.title = tokens[1]

If tokens size is greater than 2

For i from 2 to tokens size - 1

Add tokens[i] to newCourse.prerequisites

Add newCourse to courses

Close file

Procedure PrintCourseList(courses)

Sort courses by number

For each course in courses

Print course.number + ", " + course.title

Procedure PrintCourseInformation(courseNumber, courses)

Declare found as boolean = false

For each course in courses

If course.number == courseNumber

Print "Course Number: " + course.number

Print "Course Title: " + course.title

If Size(course.prerequisites) > 0

Print "Prerequisites: "

For each prerequisite in course.prerequisites

Print prerequisite

Else

Print "No prerequisites"

found = true

Break

If not found

Print "Course " + courseNumber + " not found."

Procedure Sort(courses)

// Implement an appropriate sorting algorithm here,

// such as Bubble Sort, Selection Sort, or any other,

// to sort the courses vector based on course numbers.

Evaluation of Runtime and Memory for Vector

For the given pseudocode, the core operations involve:

1. Opening and reading each line of the file.

2. Parsing each line to create course objects.

Pseudocode Evaluation

1. Open file for reading: Constant time, (O(1)), as it's a single operation.

2. While not end of file:

- Reading a line: (O(1)) per line, executed (n) times for (n) courses, totaling (O(n)).

- Split line into tokens: Assuming the worst case where each course has the maximum number of prerequisites listed, let the maximum number of tokens per line be (m). Splitting a line is (O(m)), where (m) is smaller but proportional to the length of the line.

- For loop to add prerequisites: For each course, adding a prerequisite is (O(1)), but if a course has (k) prerequisites, this loop runs (k) times, making it (O(k)) per course. Since (k) is less than or equal to (m-2) (accounting for the course number and title), we can simplify this to (O(m)) per course in the worst case.

Given that (m) is the maximum number of elements per line and (k \leq m), the total runtime for reading the file and creating course objects can be approximated as (O(n times m)), assuming (m) is significantly smaller than (n) and exhibits less variation between courses. The most time-consuming operation is the repeated parsing and object creation for each line in the file.

**Vector Data Structure Evaluation**

Advantages

- Dynamic Size: Vectors can dynamically resize, which is beneficial for storing course data where the total number of courses might not be known beforehand.

- Direct Access: Offers (O(1)) access time to elements by index, which is useful for direct access operations if course numbers are indexed accordingly.

- Ease of Use: Vectors are simple to use and implement, making them a desirable choice for straightforward data storage needs like this.

Disadvantages

- Resizing Cost: When a vector's capacity is exceeded, it must be resized and elements copied to a new location, which incurs an (O(n)) cost, though this is amortized.

- Inefficient Inserts/Deletes: Except at the end, inserting or deleting elements in the middle of a vector can be costly, (O(n)), due to the need to shift elements.

Conclusion

While vectors offer simplicity and direct access, their resizing behavior and the costs associated with insertions and deletions at arbitrary positions could be drawbacks, especially with large datasets. However, for the specific operations of loading and accessing course data (primarily done sequentially), these disadvantages are minimal. The critical evaluation factor is how the data is used; for static lists of courses accessed frequently without many modifications, vectors are a suitable choice.  
  
**Hash Table Data Structure**

// Define constants and structures

Set DEFAULT\_SIZE to 179

// Define the Course structure with properties: number, title, prerequisites

Define structure Course with properties:

number as string

title as string

prerequisites as list of string

// HashTable class for managing courses

Class HashTable

Private:

Define Node structure with properties: course, next

Declare nodes as vector of Node

Declare tableSize as unsigned int

Function hash(courseNumber) returns unsigned int

// Calculate and return hash value for a course number

Public:

Constructor HashTable(size = DEFAULT\_SIZE)

// Initialize the hash table with given or default size

Set tableSize to size

Initialize nodes vector to size

Function Insert(course)

// Insert a new course into the hash table

Declare hashKey as unsigned int = hash(course.number)

Create a new Node with course

If nodes[hashKey] is empty

nodes[hashKey] = newNode

Else

Append newNode to the end of the chain at nodes[hashKey]

Function PrintAll()

// Print all courses in alphanumeric order

Declare courseList as list of string

For each bucket in nodes

If bucket is not empty

Add bucket.course.number to courseList

Traverse chain, adding numbers to courseList

Sort courseList

For each courseNumber in courseList

PrintCourse(courseNumber)

Function PrintCourse(courseNumber)

// Print a specific course's information

Declare hashKey as unsigned int = hash(courseNumber)

Search for Node in nodes[hashKey] with courseNumber

If found

Print course details and prerequisites

Function Search(courseNumber) returns Course

// Search for a course by number and return it

Declare hashKey as unsigned int = hash(courseNumber)

Search for Node in nodes[hashKey] with courseNumber

If found, return Course

Else, return an indication that the course was not found

// Main procedure for user interaction

Procedure Main()

Declare hashTable as HashTable(DEFAULT\_SIZE)

Declare choice as integer

While True

Print "Menu Options:"

Print "1. Load Data Structure"

Print "2. Print Course List"

Print "3. Print Course"

Print "4. Exit"

Input choice

If choice == 1

LoadCourses("courseData.csv", hashTable)

ElseIf choice == 2

hashTable.PrintAll()

ElseIf choice == 3

Print "Enter course number:"

Input courseNumber

hashTable.PrintCourse(courseNumber)

ElseIf choice == 4

Print "Exiting program."

Exit

Else

Print "Invalid option, please try again."

// LoadCourses procedure for loading courses from a file into the hashTable

Procedure LoadCourses(fileName, hashTable)

Open fileName for reading

If file is not open

Print "Error opening file"

Return

While not end of file

Read line from file

Parse line into courseNumber, title, and prerequisites

Create a Course object with parsed data

hashTable.Insert(Course)

Close file

**Evaluation of Runtime and Memory for Hash Table**

Opening the File, Reading Data, and Parsing

1. File Opening: (O(1)), a single operation regardless of the file size.

2. Reading each line from the file: (O(n)), where (n) represents the number of courses (lines in the file). Reading each line is an operation that takes constant time, but this action must be repeated for every course.

3. Parsing each line: Assuming the maximum number of tokens per line (course number, title, and prerequisites) is (m), parsing a line is (O(m)). However, since (m) is typically small and fixed, this can be considered constant for each line.

4. Creating course objects: (O(1)) for each course, done (n) times for a total of (O(n)).

Inserting into the Hash Table

1. Hash function computation: (O(1)), a simple operation such as modulo by table size.

2. Inserting a new course: In the best-case scenario, the time complexity is (O(1)) when there are no collisions. In the worst-case scenario, if all (n) courses hash to the same index, inserting could deteriorate to (O(n)) as it would involve traversing the linked list at that index.

Hence, the worst-case time complexity for reading the file and generating course objects stands at (O(n)). When it comes to inserting all courses into the hash table, the worst-case scenario might reach (O(n^2)) owing to collisions. However, with an efficient hash function and a well-distributed hash space, the average time complexity is approximated to (O(n)).

Hash Table Evaluation

Advantages

- Fast Lookup: Under the assumption of an effective hash function and uniformly distributed data, hash tables offer an average time complexity of (O(1)) for search, insertion, and deletion operations.

- Flexibility: They are highly flexible and can accommodate a dynamic number of entries without a significant increase in search time.

- Direct Access: Provides direct access to data through keys, making it efficient for operations like checking course prerequisites or retrieving course details.

Disadvantages

- Collision Handling: Collisions are inevitable, and the efficiency of a hash table can significantly decrease with a poor choice of hash function or when the table becomes too crowded, leading to many collisions.

- Memory Overhead: Hash tables may require more memory than other data structures due to the need for a larger array size to minimize collisions.

- Unordered: Hash tables do not maintain any order among the stored keys, complicating operations that require sorted data. As seen in the pseudocode, printing all courses in alphanumeric order requires collecting all keys, sorting them, and then accessing each course, which adds complexity and overhead.

Conclusion

While hash tables offer efficient data access and are well-suited for scenarios with dynamic datasets and frequent insertions, searches, and deletions, they do come with the trade-off of potential inefficiency due to collisions and increased memory usage. For the specific use case of managing course information, the hash table's benefits of fast access and flexibility are compelling, but care must be taken in hash function selection and collision handling to maintain efficiency. Additionally, the unordered nature of hash tables introduces extra steps for tasks requiring sorted data, which is a significant consideration for this application.  
  
**Tree Data Structure**

// Define Course Structure

Structure Course

courseNumber as string

name as string

prerequisites as list of string

leftChild as Course

rightChild as Course

// Function to Load Courses from File and Build Tree

Function LoadCoursesFromFile(fileName) returns Course

Initialize root as null

Try

Open fileName for reading as file

Catch FileNotFound error

Print "Error: File not found."

Exit

While not end of file

Read line from file

Trim whitespace from line

If line is empty continue to the next iteration

Split line by comma into parts

If length of parts is less than 2

Print "Format error in line: " line

Continue to the next iteration

courseNumber, name = parts[0], parts[1]

prerequisites = parts[2 to end] // if there are any

course = Create new Course instance with courseNumber, name, prerequisites

root = InsertCourseIntoTree(root, course)

Close file

Return root

// Function to Insert Course into Tree

Function InsertCourseIntoTree(root, course) returns Course

If root is null

Return course

If course.courseNumber < root.courseNumber

root.leftChild = InsertCourseIntoTree(root.leftChild, course)

Else

root.rightChild = InsertCourseIntoTree(root.rightChild, course)

Return root

// Function to Print Course Information

Function PrintCourseInfo(course)

Print "Course Number: " + course.courseNumber

Print "Name: " + course.name

If course.prerequisites is not empty

Print "Prerequisites: " + course.prerequisites.join(", ")

Else

Print "No prerequisites."

// In-Order Traversal to Print Courses in Alphanumeric Order

Function InOrderTraversalPrint(root)

If root is not null

InOrderTraversalPrint(root.leftChild)

PrintCourseInfo(root)

InOrderTraversalPrint(root.rightChild)

// Main Program with Menu

Procedure Main()

Declare root as Course = null

Declare choice as integer

While True

Print "Menu Options:"

Print "1. Load Data Structure"

Print "2. Print Course List"

Print "3. Print Course"

Print "4. Exit"

Input choice

If choice == 1

root = LoadCoursesFromFile("CS\_300\_Course\_Information.txt")

ElseIf choice == 2

If root is null

Print "Data structure is empty. Please load data first."

Else

InOrderTraversalPrint(root)

ElseIf choice == 3

Print "Enter course number:"

Input courseNumber

course = SearchForCourse(root, courseNumber)

If course is not null

PrintCourseInfo(course)

Else

Print "Course not found."

ElseIf choice == 4

Print "Exiting program."

Exit

Else

Print "Invalid option, please try again."

// Function to Search for a Course in the Tree

Function SearchForCourse(root, courseNumber) returns Course

If root is null or root.courseNumber == courseNumber

Return root

If courseNumber < root.courseNumber

Return SearchForCourse(root.leftChild, courseNumber)

Else

Return SearchForCourse(root.rightChild, courseNumber)

Evaluation of Runtime and Memory for Tree

Loading Courses from File

- Opening the file: (O(1)), a constant time operation.

- Reading and parsing each line: This operation is (O(n)) where (n) is the number of courses. Parsing complexity can be considered (O(m)) for each line, with (m) being the number of elements (course number, name, prerequisites) in a line, though (m) is small and constant.

- Inserting into the BST (Binary Search Tree): The insertion operation for a Binary Search Tree (BST) is (O(log n)) in the average case for balanced trees but degrades to (O(n)) in the worst case for unbalanced trees, where all nodes are inserted in such a way that it creates a degenerate tree (effectively turning it into a linked list).

Searching for a Course

- Searching: The search operation in a BST has an average time complexity of (O(log n)) and a worst-case complexity of (O(n)) for unbalanced trees.

Printing All Courses (In-Order Traversal)

- Traversal: In-order traversal of a BST is (O(n)) as it visits each node exactly once.

Total Complexity for Loading and Operations

- Worst-case total complexity: Considering unbalanced scenarios, loading data (including parsing and insertion), and searching can reach (O(n^2)) for loading and (O(n)) for searching.

- Average-case total complexity: For balanced trees or average scenarios, the complexity is more favorable, with (O(n log n)) for loading data and (O(log n)) for searching.

**Tree Data Structure Evaluation**

Advantages

- Ordered Data Management: BSTs (Binary Search Tree) inherently maintain data in a sorted order, facilitating operations like in-order traversal to print courses in alphanumeric order.

- Efficient Searching: In balanced trees, searching for specific courses is efficient ((O(log n))), making it suitable for frequent lookups.

- Dynamic Size: Trees can dynamically grow with the dataset without needing resizing or rehashing operations.

Disadvantages

- Potential for Degradation: In the worst case, where the tree becomes unbalanced, performance can degrade, with operations approaching (O(n)).

- Complexity of Balancing: Ensuring the tree remains balanced to preserve (O(log n)) operation time requires additional algorithms (e.g., AVL, Red-Black Trees), increasing implementation complexity.

- Memory Overhead: Each node in the tree requires additional space for pointers to child nodes, which can lead to higher memory usage compared to flat data structures like arrays or hash tables.

Conclusion

For the tree data structure, especially a binary search tree (BST), provides efficient searching capabilities and manages ordered data effectively, making it suitable for handling course data. However, it's crucial to maintain the tree's balance to prevent performance degradation. The intrinsic ordering of BSTs enables the natural presentation of courses in alphanumeric order, which matches well with the requirements for displaying course information. The complexity and risk of imbalance underscore the importance of meticulous implementation and the adoption of self-balancing trees to ensure optimal performance.  
  
  
  
Considering the Big O analysis and evaluation of vector, hash table, and tree data structures for managing Computer Science courses, I recommend adopting a hash table for this application. The hash table's capacity for (O(1)) average time complexity in search, insertion, and deletion operations, given a good hash function and evenly distributed data, makes it particularly well-suited for managing dynamic course information efficiently. The justification for this choice hinges on several key considerations derived from the analysis:

**Efficiency in Lookup, Insertion, and Deletion**

- Hash Table: Offers (O(1)) average time complexity for insertion, deletion, and lookup operations, assuming a good hash function and load factor. This is particularly advantageous for the course management system, where efficient access to course information and prerequisites is critical.

- Vector: While straightforward and offering (O(1)) access by index, the vector's (O(n)) complexity for insertion and deletion (in the worst case) and the need for sorting to maintain order make it less efficient for dynamic data management.

- Tree: Offers a time complexity of (O(log n)) for insertion, deletion, and lookup operations in balanced trees. However, performance can degrade to (O(n)) in scenarios where the tree becomes unbalanced, necessitating the use of additional mechanisms to maintain balance and thereby adding complexity.

**Handling of Course Data**

- The course management system benefits from the hash table's ability to handle dynamic datasets efficiently, accommodating frequent updates, additions, and queries with minimal performance degradation.

- The hash table's direct access to data through keys (course numbers) is ideal for quickly retrieving course titles and prerequisites without the need for traversing the entire dataset.

**Considerations and Trade-offs**

- Memory Usage: Hash tables may have higher memory overhead compared to vectors due to the need for a larger array size to minimize collisions. However, this trade-off is justified by the significant gains in operation efficiency and speed.

- Ordering of Courses: While hash tables do not maintain intrinsic order (necessitating a sort of operation for ordered displays), the impact of this limitation is mitigated by the infrequent requirement for sorted course lists compared to the frequent need for fast access to specific course information.

**Conclusion and Recommendation**

Given the requirements for the Computer Science department's course management system—particularly the need for efficient, dynamic access to course information and the handling of prerequisite relationships—the hash table emerges as the most suitable data structure. It strikes an optimal balance between operational efficiency (for lookups, inserts, and deletes) and the flexibility needed to manage the courses effectively. The slight overhead of ensuring a good hash function and managing collisions is outweighed by the significant performance benefits, making hash tables the recommended choice for this application.

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Vector | Hash Table | Tree |
| Insertion | O(n) | O(1) | O(log n) |
| Search/Lookup | O(1) | O(1) | O(log n) |
| Deletion | O(n) | O(1) | O(log n) |
| Overall | O(n log n) | O(n) | O(n log n) |

Imagine a chart that succinctly compares the Big O notation for three core operations insertion, search/lookup, and deletion across three distinct data structures: vector, hash table, and tree. Additionally, this chart features an Overall Assessment row, which encapsulates the overall performance of each data structure based on a pseudocode analysis specifically tailored to managing Computer Science courses. This comparison aims to highlight the strengths and weaknesses of each data structure in the context of efficiently managing and accessing course data.

**How the Notations Were Derived:**

- Vector:

- Insertion: Given the necessity to maintain an ordered list, insertion in a vector may necessitate shifting elements to accommodate the new entry, leading to a worst-case scenario with a time complexity of (O(n)).

- Search/Lookup: Accessing an element in a vector by index is a direct operation, thus (O(1)).

- Deletion: Similar to insertion, deleting an element may necessitate shifting elements to fill the gap, also (O(n)).

- Overall: Considering operations like sorting, which may be required for maintaining order, the overall complexity was assessed as (O(n log n)).

- Hash Table:

- Insertion/Search/Lookup/Deletion: Hash tables are designed for fast access, with an average-case complexity of (O(1)) for these operations, assuming a good hash function and avoiding excessive collisions.

- Overall: The overall performance is dominated by the need to handle collisions and potentially resize the table, leading to an (O(n)) complexity in the worst case.

- Tree (Assuming a Balanced Binary Search Tree):

- Insertion/Search/Lookup/Deletion: These operations benefit from the binary search property, achieving (O(log n)) complexity by halving the search space with each step.

- Overall: The complexity for overall operations, including balancing the tree, was considered as (O(n log n)), assuming cases where balancing actions (like rotations in AVL or Red-Black Trees) are necessary.

**Summary of the Chart:**

The chart illuminates the trade-offs between these data structures in terms of operation efficiency. Vectors excel at direct access but lag in modifications, hash tables offer excellent average-case performance for all operations, and trees provide a balanced approach with logarithmic operation times, albeit with potential complexity in maintaining balance.

This evaluation, grounded in the specific pseudocode developed for managing course data—including file reading, data parsing, and the specific operations needed—enables a tailored recommendation for choosing the most suitable data structure based on the application's performance requirements and operational characteristics.