Southern New Hampshire University

CS300 DSA: Analysis and Design

Brandon Ernesto Marrero

August 5, 2023

**Reading Data from File:**

USE fstream to be able to open file

CALL fstream to open file

IF the return value is “-1”

File is not found

ELSE file is found

WHILE it is not the EOF (End Of File)

Read each line

IF there are less than two values in a line, return ERROR

ELSE read parameters

IF there is a third or more parameter

IF third or more parameter is in first parameter elsewhere continue

ELSE return Error

Close file

**Creating Course Objects**:

Initialize Course Vector vector courseInfo()

LOOP through file

WHILE not EOF

FOR each line in file

FOR first and second value

Use pushback to add value to vector

IF a 3rd value exists

Use pushback to add value until new line

**Search the Data Structure**:

Ask for Input

LOOP through vector

IF the input is the same as courseNumber

Print out the course information

FOR each prerequisite of the course

Print the prerequisite course information

Start Program

Open the file

Read data

Parse each line

FOR EACH line:

Check for course title

Check for course number

IF course parameters < than two:

Skip course

Display error message: "File isn't formatted properly"

End program

ELSE:

IF prereq is found:

Add to prereq array

IF prereq exists:

Check if prereq comes before the course

Add to hash table

ELSE:

Skip course

Display error message: "Prerequisite not found"

Add course name, course number, and prereq to hash table

CREATE constructor with parameters

Call it CreateCourseObj

INITIALIZE variables for courses and read file

Open the file to read again

WHILE file is open:

Store the course object in a hash table

CREATE constructor with parameters

Call it SearchForSpecificCourse()

Initialize variables for opening the file

Open the file

WHILE file is open:

Print course information

Store data gathered in a hash table

**Create Course Objects and Store:**

Create a class called "Course" with instance variables: courseNumber, name, and prerequisites (as a vector of strings).

Define a constructor for the "Course" class that takes the courseNumber, name, and prerequisites as parameters and initializes the instance variables accordingly.

Initialize an empty vector to store the course objects.

Prompt the user for the filename containing the course information.

Open the file with the provided filename in read mode.

IF the file does not exist, THEN display an error message and terminate the program.

Read each line from the file until the end of the file is reached.

FOR EACH line, do the following:

Split the line into comma-separated values and store them in an array.

IF the number of fields in the array is at least 2 (courseNumber and name), THEN proceed.

Extract the courseNumber and name from the array.

Check if there are any prerequisites listed for the course (fields beyond the first two).

IF there are prerequisites, THEN extract and store them in a separate array.

Create a new "Course" object with the extracted data and the prerequisites array.

Add the newly created "Course" object to the vector of course objects.

Close the file.

Display a message indicating that the file has been successfully read and course objects have been created.

RETURN the vector of course objects.

**Print Out Course Information:**

inOrderTraversal(node):

IF node is not null:

inOrderTraversal(node.left)

print courseNumber, name, and prerequisites of node

inOrderTraversal(node.right)

preOrderTraversal(node):

IF node is not null:

print courseNumber, name, and prerequisites of node

preOrderTraversal(node.left)

preOrderTraversal(node.right)

postOrderTraversal(node):

IF node is not null:

postOrderTraversal(node.left)

postOrderTraversal(node.right)

print courseNumber, name, and prerequisites of node

printCourseInfoBinarySearchTree(tree):

inOrderTraversal(tree.root)

printCourseInfoBinaryTree(root):

queue = createQueue()

enqueue(root)

while queue is not empty:

current = dequeue()

print courseNumber, name, and prerequisites of current

IF current has left child, THEN enqueue left child

IF current has right child, THEN enqueue right child

**Runtime Analysis**

| **Vector** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Create vector** | 1 | 1 | 1 |
| **For each line in the file** | 1 | n | n |
| **Create course item** | 1 | n | n |
| **While prereq exists** | 1 | n | n |
| **Append prereq** | 1 | n | n |
| **Pushback course item** | 1 | n | n |
| **Total Cost** | | | 5n + 1 |
| **Runtime** | | | O(n) |

| **HashTable** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Create hash table** | 1 | 1 | 1 |
| **Insert** | 0 | 0 | 0 |
| **Create course key** | 1 | n | n |
| **If no entry found using key** | 1 | n | n |
| **Assign a node to key** | 1 | n | n |
| **Else** | 1 | n | n |
| **Assign old key to UNIT\_MAX** | 4 | n | 4n |
| **Else** | 1 | n | n |
| **Find next open node** | 1 | n | n |
| **Add new node to end of vector list** | 1 | n | n |
| **For each new line in file** | 1 | n | n |
| **Create vector course item** | 1 | n | n |
| **While prereq exists** | 1 | n | n |
| **Append prereq** | 1 | n | n |
| **Insert course item** | 1 | n | n |
| **Total Cost** | | | 16n + 1 |
| **Runtime** | | | O(n) |

| **Tree** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Add node method** | 0 | 0 | 0 |
| **If root is null, add root** | 1 | 1 | 1 |
| **If node is less than root vars then add to left** | 1 | n | n |
| **If no left node** | 1 | n | n |
| **This node becomes left** | 1 | n | n |
| **If node is greater than the root add to right** | 1 | n | n |
| **If no right node** | 1 | n | n |
| **This node becomes right** | 1 | n | n |
| **For each line in file** | 1 | n | n |
| **Create vector course** | 1 | n | n |
| **While prereq exists** | 1 | n | n |
| **Append prereq** | 1 | n | n |
| **Insert course item** | 1 | n | n |
| **Total Cost** | | | 11n + 2 |
| **Runtime** | | | O(n) |

**Advantages and Disadvantages**

Vector:

A group of course objects are kept in a vector data structure. Because they offer dynamic scaling, vectors are appropriate in situations where the number of elements can fluctuate. This makes it possible to store and retrieve course objects quickly. It is simple to iterate through the elements in the order they were inserted because vectors preserve the order of the elements. Due to their simplicity and use, vectors are advantageous when reading data from a file. Since they can develop dynamically, they are particularly helpful when the size of the data is not known in advance.

Vectors do, however, have significant drawbacks. Since all ensuing components could need to be relocated, insertions and removals in the vector's midsection can be inefficient. This might not be an issue in this specific scenario, as courses are typically read from files sequentially. Additionally, resizing the vector might involve copying elements to a new memory location, which can be expensive in terms of time and memory.

Hash Table:

In this case, course objects are stored on a hash table. By using a hash function to map keys (in this case, course numbers) to their associated values (course objects), hash tables offer quick lookups and retrievals. When searching by course number, this guarantees quick retrieval of course information. Hash tables are especially helpful in situations when speedy data retrieval is crucial.

They do not, however, maintain the order of the components, which in this case might not be important. If not correctly managed, hash collisions—where two different keys produce the same hash value—can cause performance loss. Additionally, because the hash table itself needs to be maintained, hash tables may use more memory than other data structures.

Tree:

The tree traversal techniques used by the pseudocode include in-order, pre-order, and post-order traversals. For hierarchical data structures, such as the course prerequisites in this instance, trees are useful. The rapid searching and insertion capabilities of binary search trees (BSTs) make them suited for preserving courses in a hierarchical order based on course numbers.

Depending on how the nodes are arranged, search activities may become ineffective as a result. For instance, in the worst situation, a skewed tree can end up being simply a linked list. Although balancing strategies like AVL trees or Red-Black trees can solve this problem, they would make the implementation more difficult.

Data Structure Recommendation:

Based on the characteristics of the given implementation and the provided advantages and considerations, my recommendation is to use a Binary Search Tree (BST) for storing and managing the course information. The hierarchical nature of courses and the efficient search times provided by a balanced BST align well with the requirement of searching for courses by their course numbers and organizing them based on prerequisites. While implementing a balanced BST requires careful consideration of balancing techniques, it offers better search efficiency and a more natural representation of the hierarchical relationships between courses.

Vectors might be suitable for simpler scenarios or smaller datasets where search efficiency and hierarchical relationships are not as crucial. However, for the specific use case of this implementation, the benefits of a balanced BST outweigh the simplicity of vectors.