Jaden Williams

SNHU – CS300

2/16/2025

Project One

For the three project milestones, I used a slightly different format for each one. I think I like how I did the first milestone; it makes the most sense to read and honestly is easiest to follow. It is very interesting to compare the differences between each milestone.

**Course Structure**

A Course is a structure with the following properties:

* courseID: a string to hold the course identifier.
* courseName: a string for the course name.
* preCount: an integer to track the number of prerequisites.
* preList: a string to hold the list of prerequisite courses.

**Constructor (Course())**

When a Course is created, the constructor initializes:

* courseID and courseName as empty strings.
* preCount as 0.
* preList as an empty string.

Milestone One:

### **Function: Parser(String)**

This function reads and processes a file, which is located at the path given by the input string. Here’s how it works:

1. Create a temporary list (tempList).
2. Open the file at the provided path.
3. Read the file row by row:
   1. If both the first and second values are present, do the following:
      1. Set the first value as the courseID for a new Course.
      2. Set the second value as the courseName for the same Course.
   2. Continue reading columns in that row until there are no more values in the row.
      1. For each prerequisite found, increase preCount and append the prerequisite name to preNames.
   3. Set preCount and preList in the Course structure.
4. Add the processed Course to the tempList.
5. Once the entire file is processed, return tempList.

### **Function: searchList(String)**

This function searches for a Course by its courseID:

1. Create a temporary Course object.
2. Loop through all courses in the list:
   1. If a course's courseID matches the input string, set tempCourse to that Course.
3. Return the tempCourse.

### **Function: printCourse(String)**

This function prints the details of a course:

1. Create a temporary Course object.
2. Set tempCourse by calling searchList(String) to find the course.
3. Print the courseID and courseName to the console.
4. Loop through the prerequisites (from 0 to preCount):
   1. For each prerequisite in preList, recursively call printCourse() to print details of those prerequisites.

Milestone Two:

### **HashTable Class**

* **Bucket Structure**:
  + Each bucket contains a Course object.
  + Each bucket also has a key (used to identify the course) and a pointer to the next bucket in case of a collision (chaining).
* **HashTable Methods**:
  + hash (): A method to generate a hash value based on the course identifier (courseID).
  + printAll(): A method to print all courses in the hash table.
  + hashTable: A list to store the individual buckets of the hash table.

### **Main Program**

* **Step 1**: Create a new list called courseList to store courses (structured as CourseMap).
* **Step 2**: Prompt the user to enter the path to the CSV file containing course data.
  + If no file path is provided, use the default location.
* **Step 3**: Call txtParser() with the provided CSV file path to parse the data.
* **Step 4**: Call validateList() with courseList to check if all courses and their prerequisites are valid.
* **Step 5**: Prompt the user to enter a course ID (e.g., "CSCI100") to search for.
* **Step 6**: Call printCourse() with the user input to print course details, including prerequisites.
* **End of Main Program**

### **txtParser Function**

* **Open the CSV File**: Open the file located at the provided file path using the text parsing libraries.
* **Loop Through Each Row**: Process each row until the end of the file:
  + If the row contains both a course ID and a course name, continue.
  + Call the hash () function with the course ID to determine the correct position in the hash table.
  + Add the course to a temporary list (tempList) based on the hashed position.
  + Set the course ID and course name in the Course structure.
  + For each prerequisite listed in the row:
    - Increment the preCount by 1 for each prerequisite.
    - Concatenate the prerequisites into a string (preNames).
  + After processing all prerequisites, update the Course structure with:
    - preCount (number of prerequisites)
    - preList (list of prerequisites).
* **Return** the populated list (tempList) containing all parsed courses.

### **searchList Function**

* **Search for a Course**:
  + Create a temporary Course object to store the result.
  + Retrieve the bucket at the hashed position corresponding to the course ID.
  + If the course ID matches a course in the bucket, return that course.
  + If not found, move to the next bucket in the chain and repeat.

### **printCourse Function**

* **Print Course Details**:
  + Retrieve the Course object from the hash table using the hash () function and the provided course ID.
  + For each course, print:
    - courseID
    - courseName
  + If the course has prerequisites, print each prerequisite:
    - For each prerequisite, recursively call printCourse() to print the prerequisite course details.

### **validateList Function**

* **Validate Course List**:
  + Create a temporary Course object.
  + Set a valid variable to True initially.
  + For each course in the list:
    - If any course is found with invalid prerequisites (i.e., a prerequisite that cannot be found in the list), set valid to False.
    - Check if all courses in the prerequisite list are valid by recursively checking if each prerequisite is a valid course in the list.
  + If any issues are found, return False; otherwise, return True.

### **Hash Function**

* **Create a Hash for the Course ID**:
  + Design a hash function that generates a hash value based on the course ID.
  + Consider the structure of the course ID, ensuring that the first part is considered, as it may appear frequently in a specific department.
  + The last part of the course ID might be less unique, so the hash function should account for both the department code and course number.
  + Return the hash value to locate the correct bucket for the course.

Milestone Three:

### **Binary Tree Setup**

* **Define BinaryTree class**: You create a class to manage a binary tree where each node stores a Course. This class contains:
  + A Node structure that holds a Course, pointers to the left and right nodes, and the root pointer.
  + Methods for printing courses (printCourse()) and creating a binary tree (BinaryTree()).

### **Main Program Logic**

* **Create a new BinaryTree**: In the main function, you create a BinaryTree named courseTree that will store Course objects.
* **Get file path from the user**: You ask the user for the path to a CSV file. If the user doesn't provide one, you use a default location.
* **Call txtParser()**: The txtParser function is called, which will process the CSV file from the given file path.
* **Call validateList()**: The validateList() function is called, which will check that the course data in the tree is valid.
* **Search and Print Course**: You prompt the user to search for a course by entering a courseID, and the program will display the matching course details.

### **txtParser Function**

* **Open the file**: The function opens the CSV file specified in the provided path.
* **Loop through the file**: You read the file line by line. If the line contains a course ID and course name, you extract those values:
  + The courseID goes into the Course struct.
  + The courseName goes into the Course struct.
* **Process prerequisites**: If there are prerequisite courses listed in the file, you loop through them, incrementing preCount for each prerequisite and adding their names to preList.
* **Return the list**: After parsing the file, you return the list of courses.

### **searchList Function**

* **Create a temporary course node**: You create a temporary Node to search for the course.
* **Find the course in the list**: You use a hash table to look up a course based on the courseID. If a course with the specified ID is found, you return it.

### **printCourse Function**

* **Create a temporary course**: You start with the root node of the tree and look for the course with a matching courseID.
* **Print course details**: Once the correct course is found, you print the courseID and courseName.
* **Print prerequisites**: For each prerequisite in preList, you call printCourse() recursively to print details about the prerequisites.
* **Traverse the tree**: If the current course's courseID is less than the search string, you move to the left node in the tree. If it’s greater, you move to the right node.

### **validateList Function**

* **Validate prerequisites**: You loop through each course in the list and check if its prerequisites are valid:
  + For each prerequisite, you use searchList to find it in the tree.
  + If a prerequisite course doesn't exist or is invalid, you set a flag (valid) to False and break the loop.
* **Return validation result**: Finally, you return whether the list is valid based on the checks.

Run Time Analysis:

Vector Data Structure

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executed | Total Cost |
| For all courses | 1 | n | n |
| If course is same as course number | 1 | n | n |
| For each prerequisite of course | 1 | 1 | 1 |
| For each prerequisite of course | 1 | n | n |
| Print the prerequisite information | 1 | n | n |
| **Total Cost** | | | 4n+1 |
| **Runtime** | | | O(n) |

Binary Search Tree

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executed | Total Cost |
| If root equals null pointer plus node | 1 | 1 | 1 |
| If node is less than root add to left node | 1 | n | n |
| If left node does not exist | 1 | n | n |
| Current node equals left node | 1 | n | n |
| Else if node is greater than root add to right | 1 | n | n |
| If right node does not exist | 1 | n | n |
| Current node equals right node | 1 | n | n |
| For each line in file | 1 | n | n |
| Create vector with course item | 1 | n | n |
| While the prerequisite exists | 1 | n | n |
| Append the prerequisite | 1 | n | n |
| Insert the function for course item | 1 | n | n |
| **Total Cost** | | | 11n+2 |
| **Runtime** | | | O(n) |

Hash Table

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executed | Total Cost |
| Create key for courses | 1 | n | n |
| If key not found | 1 | n | n |
| Assign node key | 1 | n | n |
| Else | 1 | n | n |
| Set key to max, set old node next to null pointer | 3 | n | 3n |
| Else | 1 | n | n |
| Locate next new node | 1 | n | n |
| Add new node to end | 1 | n | n |
| For each line in file | 1 | n | n |
| Create the vector with course item | 1 | n | n |
| If prerequisite exists | 1 | n | n |
| Append | 1 | n | n |
| Insert course item | 1 | n | n |
| **Total Cost** | | | 14n+1 |
| **Runtime** | | | O(1) |

Advantage Analysis:

Each data structure has pros and cons. Adding data to an unsorted vector is very fast but sorting it later can be slow.

A hash table can perform at its best, O (1), if it’s large enough to avoid collisions. However, since space and time are limited, some collisions are inevitable, and performance will range from O (1) to O(N).

A binary tree usually operates efficiently, but if the data is loaded in a sorted order, it can become unbalanced, slowing performance to O(N).

The best data structure depends on how and how often you need to access the data. If data is added infrequently, the choice doesn’t matter much after the initial load. But if the data is frequently searched, a hash table might be faster than a binary tree, especially if the tree is unbalanced or the hash function is good.

Finally, the binary tree doesn’t require sorting and can be traversed in order, which saves memory if both sorted and unsorted versions aren’t needed. The binary tree and hash table tend to perform better than sorting a vector.

My Recommendation and Final Thoughts:

I will be entirely honest; this was a bit of a challenge for me trying to start it. I am not 100% sure I still did it correctly and added it correctly, but I am confident enough in my work. However, based on this analysis and the work we have done in class I believe the Hash Table to be the best data structure to utilize for this project. The hash table and function will need to be optimized to limit collisions and allow for the code to run as efficiently as possible.