Mark Turner – 12/5/2024

CS 300 SNHU – Module 6: Project 1

**Pseudocode for a menu**

**function menu():**

Option = 0

Initialize empty collection 'courses'

While Option is not 9:

Print "Menu:"

Print "1. Load course data"

Print "2. Print sorted list of Computer science courses"

Print "3. Print course title and prerequisites"

Print "9. Exit"

Prompt "Select an option: " and get user input (Option)

If Option == 1:

Call validateFile(fileName)

If file is valid, load the data into the 'courses' collection

Else, print validation error

Else if Option == 2:

Initialize empty list 'cs\_courses'

For each course in 'courses':

If "CS" in course\_number or "CS" in course\_title:

Add course to 'cs\_courses'

Sort 'cs\_courses' by course\_number alphanumerically

For each course in 'cs\_courses':

Print course\_number + ": " + course\_title

Else if Option == 3:

Prompt "Enter the course number: " and get user input

For each course in 'courses':

If course\_number matches:

Print "Course Title: " + course\_title

If course.prerequisites is not empty:

Print each prerequisite

Else if Option == 9:

Print "Exiting the program."

Exit the application

Else:

Print "Invalid option. Please try again."

**Pseudocode that will print out the list of the courses in the Computer Science program in alphanumeric order (Binary tree)**

// binary search tree will maintain order during insertion

**function printCoursesInOrder(course):**

If course is not empty:

Call printCoursesInOrder(course.before) // print prior courses

Print the course information

Call printCoursesInOrder(course.after) // print subsequent courses

**Runtime Analysis: Reading from file and creating course objects (Vector)**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Open file filename | 1 | 1 | 1 |
| Initialize empty List 'courses' | 1 | 1 | 1 |
| For each line in the file: | 1 | n | n |
| Split the line into 'course\_data' | 1 | n | n |
| If 'course\_data' has fewer than 2 items: | 1 | n | n |
| Print error and skip line | 1 | n | n |
| Assign course\_number = course\_data[0] | 1 | n | n |
| Assign course\_title = course\_data[1] | 1 | n | n |
| Initialize empty list 'prerequisites' | 1 | n | n |
| For each item starting from index 2 in 'course\_data': | 1 | n^2 | n^2 |
| Add item to prerequisites | 1 | n^2 | n^2 |
| Add the course\_number, course\_title, and prerequisites as ‘course’ object to ‘courses’ | 1 | n | n |
| Close the file | 1 | 1 | 1 |
| Return ‘courses’ | 1 | 1 | 1 |
| **Total Cost** | | | 8n + 2n^2 + 4 |
| **Runtime** | | | O(n^2) |

**Explanation:** If insertions are constant time (O(1)), which is true when appending to the end of the vector, the worst-case time complexity would be O(n²). The total operations would be n × (n - 1), simplifying to O(n²). Since appending each course/prerequisite is O(1), the overall time complexity remains O(n²), dominating other operations.

**Runtime Analysis: Reading from file and creating course objects (Hash Table)**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Open file fileName | 1 | 1 | 1 |
| If file can't be opened | 1 | 1 | 1 |
| Initialize empty HashTable courses | 1 | 1 | 1 |
| For each line in file | 1 | n | n |
| Split line by commas into course\_data | 1 | n | n |
| If course\_data has fewer than 2 elements | 1 | n | n |
| Assign course\_data[0] to course\_number | 1 | n | n |
| Assign course\_data[1] to course\_title | 1 | n | n |
| Initialize empty list prerequisites | 1 | n | n |
| For each item in course\_data starting at index 2 | 1 | n ^2 | n ^2 |
| Add item to prerequisites | 1 | n ^2 | n ^2 |
| Create Course object with number, title, and prerequisites | 1 | n | n |
| Else, Insert Course object into courses with course\_number as key | 1 | n | n |
| Close file | 1 | 1 | 1 |
| **Total Cost** | | | 8n + 4 + 2n^2 |
| **Runtime** | | | O(n^2) |

**Explanation**: The worst-case time complexity is O(n²), even if p (the number of prerequisites) is at most n - 1 (assuming no course is its own prerequisite). If each course has n - 1 prerequisites, the total operations are n × (n - 1), simplifying to O(n² - n), which is O(n²) in Big-O notation. This dominates the complexity of inserting into a hash table for each course (n) and other linear operations, so the overall complexity is O(n²).

**Runtime Analysis: Reading from file and creating course objects (Binary Search Tree)**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Open file fileName | 1 | 1 | 1 |
| If file can't be opened | 1 | 1 | 1 |
| Initialize empty BinaryTree courses | 1 | 1 | 1 |
| For each line in file | 1 | n | n |
| Split line by commas into course\_data | 1 | n | n |
| If course\_data has fewer than 2 elements | 1 | n | n |
| Print error and exit | 1 | n | n |
| Assign course\_data[0] to course\_number | 1 | n | n |
| Assign course\_data[1] to course\_title | 1 | n | n |
| Initialize empty list prerequisites | 1 | n | n |
| For each item in course\_data starting at index 2 | 1 | n ^2 | n ^2 |
| Add item to prerequisites | 1 | n ^2 | n ^2 |
| Create Course object with number, title, and prerequisites | 1 | n | n |
| If courses.first\_course is empty: | 1 | n | n |
| Assign courses.first\_course to course object | 1 | 1 | 1 |
| Else, Insert Course object into courses with course\_number as key | n\*log n | n\*log n | n\*log n |
| Close file | 1 | 1 | 1 |
| **Total Cost** | | | 9n + 5 + 2n ^2 + n\*log n |
| **Runtime** | | | O(n ^2) |

**Explanation**: The worst-case time complexity is O(n²), even if p (the size of the prerequisites list) can be as large as n - 1 but never equal to n (assuming a course cannot be its own prerequisite). If each course has n - 1 prerequisites, the complexity simplifies to O(n²). With operations like inserting into a binary search tree taking O(log(n)) time, the overall complexity is O(n²), which dominates linear and logarithmic operations.

**Advantages & Disadvantages**

**Vector**

**Advantages:**

* Efficient appending (O(1)) when adding elements to the end.
* Provides O(1) random access to elements by index.

**Disadvantages:**

* Inefficient insertions or deletions elsewhere in the list (O(n)).
* Requires resizing when capacity is exceeded, leading to occasional O(n) time complexity.
* Linear search for elements or prerequisites is O(n), making searches slower.

**Binary Search Tree (BST)**

**Advantages:**

* Maintains sorted order of elements, allowing efficient in-order traversal.
* O(log n) time complexity for search, insert, and delete operations in a balanced tree.
* Flexible structure, allowing dynamic resizing and reordering of elements.

**Disadvantages:**

* Ordering required to maintain O(log n) time complexity; unbalanced trees degrade to O(n).
* More complex to implement and maintain.

**Hash Table**

**Advantages:**

* O(1) average-time lookups and insertions, making it efficient for key-based access.
* No need for ordering, which simplifies insertions.

**Disadvantages:**

* Collisions can degrade performance to O(n) in the worst case (depending on implementation).
* No ordering of elements, which may be a limitation if sorted data is required.

**Final Recommendation**

For the scenario of sorting and outputting the courses, a **Binary Search Tree (BST)** would be the most suitable data structure. It allows efficient insertions while maintaining sorted order, and it supports **O(n)** in-order traversal to print the courses in sorted order. In contrast, the **Hash Table** may require additional steps to sort its contents, as it doesn’t inherently maintain any order. Additionally, while a **Vector** can be sorted, it requires **O(n log n) or worse** time to do so, which is less efficient than the BST's **O(n)** traversal for printing in sorted order.