CS 300 Pseudocode

Shamus Cerny

## Menu:

1. Display the menu options
2. Repeat until the user chooses to exit
   * Ask the user to enter an option
   * If the user chooses 1, then
     + Load the course data and display confirmation message
   * If the user chooses 2, then
     + Check if data has been loaded
     + If not, display message asking user to load data
     + Otherwise display all courses in order
   * If the user chooses 3, then
     + Check if the data has been loaded
     + If not display message asking user to load data
     + Otherwise, ask the user for course id
     + Find the course in data structure and display its information
   * If the user chooses 9, then
     + Exit the program with a goodbye message
   * If the user enters an invalid option, display an error message and ask again

## File Validation:

To validate a course file, follow these steps:

1. Open the file using the provided file name.
2. If the file is not found, display an error message and stop the process.
3. Create an empty list to keep track of course numbers.
4. Read each line in the file one by one:
   * If the line is empty, move to the next line.
   * Split the line into separate pieces based on spaces or commas.
   * If there are fewer than two pieces of information, display an error message and move to the next line.
   * Extract the course number from the first piece and the course title from the second.
   * If there are additional pieces, treat them as prerequisite courses.
   * Check if the course number has already been added to the list:
     + If it has, display an error message and move to the next line.
     + Otherwise, add the course number to the list.
   * For each prerequisite course:
     + Check if it exists in the list of course numbers.
     + If not, display an error message and move to the next prerequisite.
5. Once all lines are processed, display a success message.

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **For each line in the file** | 1 | n | n |
| **If the Line is empty move to next line** | 1 | n | n |
| **Split the line into a list of strings** | 1 | n | n |
| **If there are less than 2 pieces of information display error** | 1 | n | n |
| **Extract Course # and course title** | 1 | n | n |
| **For each prerequisite, check if it exists in the list** | p | n | n + p |
| **Check if course # already exists** | 1 | n | n |
| **Total Cost** | | | 6n + p |
| **Runtime** | | | O(n) |

# Vector:

**Creating a Course Class and Storing Temporary Data:**

To store course data temporarily, follow these steps:

1. Define a structure that holds a course number, course title, and a list of prerequisites.
2. Open the file using the provided file name.
3. If the file is not found, display an error message and stop the process.
4. Create an empty list to store course objects.
5. Read each line in the file one by one:
   * If the line is empty, move to the next line.
   * Create a temporary structure to store the course data.
   * Split the line into separate pieces based on spaces or commas.
   * Extract the course number and title from the first two pieces.
   * If there are additional pieces, store them as prerequisites.
   * Create a new course object using this data.
   * Add the new course object to the list.
6. Once all lines are processed, display a success message.

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Create Course Struct** | 1 | 1 | 1 |
| **For each line in the file** | 1 | n | n |
| **If the Line is empty move to next line** | 1 | n | n |
| **Split the line into a list of strings** | 1 | n | n |
| **Extract Course # and course title** | 1 | n | n |
| **For each prerequisite add to struct** | p | n | n + p |
| **Create course object and add to list** | 1 | n | n |
| **Total Cost** | | | 5n + p |
| **Runtime** | | | O(n) |

**Searching for Course Information:**

To search for a specific course, follow these steps:

1. Go through each course stored in the list.
2. If the course number matches the one being searched for:
   * Display the course number and title.
   * If the course has prerequisites, list them.
   * If it does not, indicate that there are no prerequisites.
   * Stop the search.
3. If no course matches the provided number, display an error message.

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **For each course in the list** | 1 | n | n |
| **If the course # matches** | 1 | n | n |
| **Display course number and title** | 1 | 1 | 1 |
| **For each prerequisite of the course display course #** | p | 1 | p |
| **Total Cost** | | | 2n + p + 1 |
| **Runtime** | | | O(n) |

**Print List of All Courses with prerequisites:**

1. Start with an empty “ last printed” course
2. Repeat until there is no course found that is greater than the last printed course
   * Loop through each course in the vector list
   * Find the smallest course that is still greater than the last printed course
   * Display the course information with its prerequisites
3. Once there are no courses greater than the last printed course all courses should be displayed in alphanumerical order.

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Start with empty “Last Printed” course** | 1 | 1 | 1 |
| **For each course, find the smallest greater than “last printed”** | 1 | n | n |
| **Display course information** | 1 | 1 | 1 |
| **Display prerequisites** | P | n | N\*p |
| **Total Cost** | | | n + np + 2 |
| **Runtime** | | | O(n\*p) |

# Hashtable:

**Creating a Course Class and Storing Data in a Hashtable:**

To store course data in a hashtable, follow these steps:

1. Define a structure that holds a course number, course title, and a list of prerequisites.
2. Open the file using the provided file name.
3. If the file is not found, display an error message and stop the process.
4. Create an empty hashtable to store course objects, where course numbers will serve as keys.
5. Read each line in the file one by one:
   * If the line is empty, move to the next line.
   * Split the line into separate pieces based on spaces or commas.
   * Extract the course number, title, and prerequisites.
   * Create a new course object using this data.
   * Store the course object in the hashtable, using a hashing function to determine its location.
   * Append the course object to the linked list that exists in the current bucket.
6. Once all lines are processed, display a success message.

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Define Course Struct** | 1 | 1 | 1 |
| **Open the file** | 1 | 1 | 1 |
| **Create empty hashtable** | 1 | 1 | 1 |
| **For each line in the file** | 1 | n | n |
| **Split line into string list** | 1 | N | N |
| **Extract course # and title and add create course object** | 1 | N | N |
| **For each prerequisite add to object** | P | n | n+p |
| **Using hash function find bucket to store course** | 1 | N | N |
| **Append course to linked list** | 1 | 1 | 1 |
| **Total Cost** | | | 3n + p + 4 |
| **Runtime** | | | O(n) |

**Searching for Course Information in a Hashtable:**

To search for a specific course in a hashtable, follow these steps:

1. Using the information provided use the hashing function to find the correct bucket.
2. Iterate through the linked list checking if the course matches the information provided.
3. If it does:
   * Retrieve the corresponding course object.
   * Display the course number and title.
   * If the course has prerequisites, list them.
   * If it does not, indicate that there are no prerequisites.
   * Return a success message.
4. If the course number is not found, return an error message.

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Find the correct bucket using hash function** | 1 | 1 | 1 |
| **Iterate through linked list** | K | 1 | k |
| **Display course # and title** | 1 | 1 | 1 |
| **Display each prerequisite** | P | 1 | P |
| **Display error message if not found** | 1 | 1 | 1 |
| **Total Cost** | | | k + p + 3 |
| **Runtime** | | | O(n) |

**Print List of All Courses with prerequisites:**

1. Loop through each bucket in the hashtable to find the smallest course.
   * For each bucket, loop through its linked list.
   * Keep trach of the smallest so far.
2. Display the smallest course and its prerequisites
3. Save the course as the last printed
   * On the next scan only consider courses that are greater than the last printed course
4. Repeat this process until there is no course found
5. If there is no course found that is greater than the last printed course then all courses should be displayed in alphanumerical order.

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Iterate through each bucket in hashtable** | B | N | N\*B |
| **Iterate through each linked list** | K | B\*N | B\*K\*N |
| **Find smallest course that is greater than last printed** | 1 | N | N |
| **Save course as last printed** | 1 | N | N |
| **Display course # and title** | 1 | N | N |
| **Display each prerequisite** | P | N | N\*P |
| **Total Cost** | | | B\*N+B\*K\*N+3n+P\*N |
| **Runtime** | | | O(N\*(B+K+P)) |

# Binary Search Tree (BST):

**Creating a Course Class and Storing Data in a BST:**

1. **Define a structure** to hold course details:
   * Course number
   * Course title
   * List of prerequisites
2. **Define a binary search tree (BST) node structure** that holds:
   * A course object
   * Left and right child pointers
3. **Create an empty binary search tree**
4. **Read each line in the file one by one:**
   * If the line is empty, move to the next line.
   * Split the line into separate pieces based on spaces or commas.
   * Extract the course number, title, and prerequisites.
   * Create a new course object using this data.
   * Insert the course object into the BST using the insertion function.
5. **function to insert a course into the BST**
   * If the tree is empty, set the new node as the root.
   * Otherwise, compare the course number with the current node’s course number.
   * If smaller, go left; if larger, go right.
   * Recursively insert into the correct subtree.
6. **Once all lines are processed, display a success message**.

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Create Course struct** | 1 | 1 | 1 |
| **Create Binary Tree** | 1 | 1 | 1 |
| **For each line in file** | 1 | N | N |
| **Split line into string list** | 1 | 1 | 1 |
| **Extract course # and title** | 1 | 1 | 1 |
| **Extract each prerequisite** | p | 1 | P |
| **Create course object** | 1 | 1 | 1 |
| **Check if tree is empty** | 1 | 1 | 1 |
| **Compare course with current node** | 1 | Log n | Log n |
| **Move left or right based on comparison** | 1 | Log n | Log n |
| **Insert into the correct subtree** | 1 | Log n | Log n |
| **Total Cost** | | | Log n |
| **Runtime** | | | O(n \* log n) |

**Print Given Course Information and Pre-requisites:**

1. Starting at the root
2. Repeat until the course is found
   * Using the given course information if it is less than the current node move left
   * If it is greater than the current node move right
3. Once course is found display the course information and prerequisites
4. If no course is found display an error.

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Start at the root of the BST** | 1 | 1 | 1 |
| **Compare course with current node** | 1 | Log n | Log n |
| **Move left or right based on comparison** | 1 | Log n | Log n |
| **Display course # and title** | 1 | 1 | 1 |
| **Display each prerequisite** | P | 1 | P |
| **If no course found display error message** | 1 | 1 | 1 |
| **Total Cost** | | | Log n + 3 |
| **Runtime** | | | O(log n) |

**Print List of All Courses with prerequisites:**

1. If the root is empty then there are no courses.
2. Starting from the root of the tree
3. Loop through this process
   * Visit the left child – going as far left first
   * Display the course information at the current node
     + Printing the node only when there is not a left child
   * Visit the right child
4. Once the loop reaches the furthest right node all the courses should be displayed in alphanumerical order.

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **If the root is empty, there are no courses** | 1 | 1 | 1 |
| **Start from the root** | 1 | 1 | 1 |
| **Loop through the process (in-order traversal)** | 1 | N | N |
| **Visit the left child** | 1 | N | N |
| **Display course info at the current node** | 1 | N | N |
| **Visit the right child** | 1 | N | N |
| **Total Cost** | | | 4n + 2 |
| **Runtime** | | | O(n) |

## Analysis:

The vector data structure provides a simple and dynamic array-like structure that allows for efficient random access and easy appending of data. Its main advantage is in its straightforward implementation and ease of use, especially for relatively small datasets. However the disadvantage is that searching for a specific course can be inefficient. The hash table, on the other hand can offer faster lookup times if there are minimal collisions. The downside to hash tables is that they can have major performance issues when there are many collisions, and the handling of collisions through linked lists can lead to inefficiencies in some cases. The binary search tree, particularly a balanced search tree, offers more efficient insertions, deletions, and searches. However an unbalanced search tree can seriously degrade performance.

## Recommendation:

Based on the analysis, I recommend using the binary search tree (BST). The BST provides an efficient way to maintain ordered data while ensuring O(log n) time complexity for search, insertion and deletion when balanced. Since there are a large number of courses and the users will need to search and view them in an ordered fashion the BST is an ideal choice. While a hash table could be better in some cases, the potential for collisions and performance degradation in worst-case scenarios make it less reliable for consistent performance. The vectors O(n) search time also makes it unsuitable for large datasets.