**Jess Dowd**  
CS 300: Analysis and Design  
Michael Rissover  
October 17, 2024

**CS 300 Project One**

**1. Resubmit pseudocode from previous pseudocode assignments and update as necessary**

**A. Pseudocode for Vector (Resubmitted from Milestone One)**

**Opening and Reading the File**

Open the file "CourseData.txt."

If the file opens successfully then:

While there are still lines to read:

Read the current line.

Split the line by commas to get the course number, course title, and prerequisites.

Call the function to validate the line with course number, course title, and prerequisites.

If the file fails to open, display "Error: Could not open file."

**Function: validateLine(courseNumber, courseTitle, prerequisites)**

If the course number or course title is missing, display "Error: Missing course number or title" and exit.

If there are prerequisites:

For each prerequisite:

If the prerequisite isn’t found in the course list, display "Error: Prerequisite [prerequisite] does not exist" and exit.

**Creating Course Objects**

Struct: Course

Include fields for course number, course title, and prerequisites.

Declare a vector named "courses" to hold Course objects.

While there are still lines to read:

Read the current line.

Split the line to get course number, course title, and prerequisites.

Create a new Course object.

Assign the course number, course title, and prerequisites to the new Course object.

Add the new Course object to the "courses" vector.

**Printing Course Information**

Function: searchCourse(Vector<Course> courses, String courseNumber)

For each course in the "courses" vector:

If the course's course number matches the searched course number:

Print the course number and course title.

If there are prerequisites:

Print the prerequisites.

If no prerequisites, print "No prerequisites."

Return from the function.

If no course matches, print "Course not found."

Request user input for a course number to search:

Input "Enter course number to search: " into "searchNumber."

Call the searchCourse function with "courses" and "searchNumber."

**B. Pseudocode for Hash Table (Resubmitted from Milestone Two)**

**Opening and Reading the File**

Open the file "CourseData.txt."

If the file opens successfully then:

While there are still lines to read:

Read the current line.

Split the line by commas to get the course number, course title, and prerequisites.

Call the function validateLine(courseNumber, courseTitle, prerequisites) to validate the line.

If the file fails to open, display "Error: Could not open file."

**Function: validateLine(courseNumber, courseTitle, prerequisites)**

If course number or course title is missing:

Display "Error: Missing course number or title" and exit.

If there are prerequisites:

For each prerequisite in prerequisites:

If the prerequisite isn’t found in the course list:

Display "Error: Prerequisite [prerequisite] does not exist" and exit.

**Creating Course Objects**

Struct: Course

Include fields for course number, course title, and prerequisites.

Declare a hash table named "courseHashTable."

While there are still lines to read:

Read the current line.

Split the line to get course number, course title, and prerequisites.

Create a new Course object.

Assign course number, course title, and prerequisites to the new Course object.

Store the new Course object in "courseHashTable" using the course number as the key.

**Printing Course Information**

Function: printCourseInfo(String courseNumber)

If courseNumber exists in courseHashTable:

Get the Course object from courseHashTable.

Print course number and course title.

If there are prerequisites:

Print the prerequisites.

Else, print "No prerequisites."

Else, print "Course not found."

**C. Pseudocode for Binary Search Tree (Resubmitted from Milestone Three)**

**Opening and Reading the File**

Open the file "CourseData.txt."

If the file opens successfully then:

While there are still lines to read:

Read the current line.

Split the line by commas to get the course number, course title, and prerequisites.

Call the function validateLine(courseNumber, courseTitle, prerequisites) to validate the line.

If the file fails to open, display "Error: Could not open file."

**Creating Course Objects**

Struct: Course

Include fields for course number, course title, and prerequisites.

Declare a binary search tree named "courseBST."

While there are still lines to read:

Read the current line.

Split the line to get course number, course title, and prerequisites.

Create a new Course object.

Assign course number, course title, and prerequisites to the new Course object.

Insert the new Course object into "courseBST" using the course number as the key.

**Printing Course Information**

Function: printCourseInfo(String courseNumber)

If courseNumber exists in courseBST:

Get the Course object from courseBST.

Print course number and course title.

If there are prerequisites:

Print the prerequisites.

Else, print "No prerequisites."

Else, print "Course not found."

**2. Create pseudocode for a menu**

Function: displayMenu()

While True:

Print "1. Load course data from file."

Print "2. Print all courses in alphanumeric order."

Print "3. Print course title and prerequisites."

Print "9. Exit."

Input "Enter your option: " into option

If option == 1:

Call loadCourseData()

Else if option == 2:

Call printCourseList()

Else if option == 3:

Input "Enter the course number: " into courseNumber

Call printCourseInfo(courseNumber)

Else if option == 9:

Print "Exiting the program."

Break

Else:

Print "Invalid option. Please enter a valid option."

**3. Design pseudocode that will print out the list of the courses in the Computer Science program in alphanumeric order.**

Function: printCourseList()

If the data structure is empty:

Print "No courses available."

Return from function.

Call the appropriate function for the selected data structure to print courses in alphanumeric order:

If using a vector, call printCoursesVector().

If using a hash table, call printCoursesHashTable().

If using a binary search tree, call printCoursesBST().

1. **Vector**

Function: printCoursesVector(Vector<Course> courses)

If courses is empty:

Print "No courses available."

Return from function.

Call sort(courses) by course number in alphanumeric order.

For each course in the sorted "courses" vector:

Print course number and course title.

1. **Hash Table**

Function: printCoursesHashTable(HashTable<Course> courseHashTable)

If courseHashTable is empty:

Print "No courses available."

Return from function.

Create an empty list named "courseList."

For each key in courseHashTable:

Add the Course object associated with the key to "courseList."

Call sort(courseList) by course number in alphanumeric order.

For each course in the sorted "courseList":

Print course number and course title.

1. **BST**

Function: printCoursesBST(BinarySearchTree<Course> courseBST)

If courseBST is empty:

Print "No courses available."

Return from function.

Call inOrderTraversal(courseBST root)

Function: inOrderTraversal(Node node)

If node is not null:

Call inOrderTraversal(node.left) // Traverse the left subtree

Print node.courseNumber and node.courseTitle // Print the current course

Call inOrderTraversal(node.right) // Traverse the right subtree

**4. Runtime Analysis and Evaluation**

**A. Runtime Analysis for Vector Pseudocode**

| **Line** | **Cost per Execution** | **# Times Executed** | **Total Cost** |
| --- | --- | --- | --- |
| If courses is empty | 1 | 1 | 1 |
| Print "No courses available." | 1 | 0 (if not empty) | 0 |
| Return from function | 1 | 0 (if not empty) | 0 |
| Sort courses (alphanumeric order) | O(n log n) | 1 | O(n log n) |
| For each course in the sorted vector | 1 | n | n |
| Print course number and course title | 1 | n | n |
| **Total Cost** |  |  | O(n log n) + n |
| **Total Runtime** |  |  | **O(n log n)** |

**B. Runtime Analysis for Hash Table Pseudocode**

| **Line** | **Cost per Execution** | **# Times Executed** | **Total Cost** |
| --- | --- | --- | --- |
| If courseHashTable is empty | 1 | 1 | 1 |
| Print "No courses available." | 1 | 0 (if not empty) | 0 |
| Return from function | 1 | 0 (if not empty) | 0 |
| Create an empty list named "courseList" | 1 | 1 | 1 |
| For each key in courseHashTable | 1 | n | n |
| Add the Course object to "courseList" | 1 | n | n |
| Sort courseList (alphanumeric order) | O(n log n) | 1 | O(n log n) |
| For each course in the sorted "courseList" | 1 | n | n |
| Print course number and course title | 1 | n | n |
| **Total Cost** |  |  | O(n log n) + 3n + 2 |
| **Total Runtime** |  |  | **O(n log n)** |

1. **Runtime Analysis for Binary Search Tree (BST) Pseudocode**

| **Line** | **Cost per Execution** | **# Times Executed** | **Total Cost** |
| --- | --- | --- | --- |
| If courseBST is empty | 1 | 1 | 1 |
| Print "No courses available." | 1 | 0 (if not empty) | 0 |
| Return from function | 1 | 0 (if not empty) | 0 |
| Call inOrderTraversal(courseBST root) | O(n) | 1 | O(n) |
| **Total Cost** |  |  | O(n) + 1 |
| **Total Runtime** |  |  | **O(n)** |

**5. Advantages and Disadvantages**

Advantages of Hash Tables they are Extremely fast search, insert, and delete operations. They have average time complexity of O(1) that makes hash tables good for quick lookup. Disadvantages are Hash tables do not maintain any specific order so more steps (like extracting data to a list and sorting) are needed with ordered data which adds more time complexity.

Advantages of Vectors they are efficient for small datasets since access by index is very fast (O(1)). Disadvantages are inserting and deleting elements can be inefficient because it requires shifting elements which has a time complexity of O(n).

Advantages of A binary search tree is it maintains elements in a naturally sorted order which means that data can be retrieved in alphanumeric order without any extra sorting steps. Disadvantages are if the tree becomes the time complexity can degrade to O(n) which will make the operations slower.

**6. Recommendation**

After analyzing the three data structures (vector, hash table, and binary search tree), I recommend using the **binary search tree (BST)** for this program. The project requires that courses be printed in alphanumeric order. A **hash table** has the fastest search and insert operations with constant time complexity (O(1))but it does not have inherent order so more steps are needed to sort the data before printing. The **binary search tree (BST)** seems like the best choice because it inherently has the data in sorted order. This lets the program print the courses in alphanumeric order without extra sorting. Also the BST has search and insert operations with an average time complexity of O(log n) which is reasonable. One risk here is that the tree could be unbalanced.

Each data structure has advantages and disadvantages for the requirements of the

program. A disadvantage to using a vector is having to search the list for a specific course. Until

a match is found the program much check each item. However, the vector method has the

advantage of being the fastest method for reading the file and adding the course objects. This

method is straightforward. Of the three methods the runtime was the shortest at 5n+1.

Hash tables have the advantage of being able to search a list fast. Creating a key,

locations can easily be searched and printed. This is a slower implementation when creating the

initial list and a spot found to insert each course. Hash tables do not allow the table itself to be

sorted. To print an alphanumeric list of all courses each value must be extracted, sorted, and then

printed. Due to all of these reasons this is not the best data structure for this program.

Binary trees have the advantage over vector because of the fast ability to sort. Knowing.

It is not as easy as a hash table, but quicker than a vector. The search time is O(h) where h is the

height of the tree.

I would recommend a vector sort for this project. I think that being able to quick sort to

print the entire catalogue is more valuable. The loss of time during the search is not as quite as

bad as the utility of the sort. In my opinion I think vector is the best option