* **Eli Rubin-Calvert Project One 6/11/2023 CS 300**

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

* **Vector Pseudocode:**
* Begin Program
* Start with the first bid and set it to be the minimum.
* Loop through all the remaining bids.
* Compare the title of the current bid with the title of the minimum bid that has been set.
* If the title of the current bid is smaller, update the minimum index to be the index of the current bid.
* Repeat loop and compare until the end of the list is reached.
* If the minimum index has been updated, swap the current bid with the bid at the minimum index.
* Move to the next bid and repeat steps until all the bids have been sorted.
* Sort list in ascending order based on the title of each bid.
* Define a function named quickSort that takes in three parameters: a vector of bids, the start of the array, and the end of the array.
* If the starting index is greater than or equal to the ending index, return.
* Choose the last element of the array as the pivot value.
* Set two pointers, one at the beginning and the other at the end of the array.
* Swap the pivot element with the element pointed to by the left pointer.
* Set mid equal to the value of the left pointer.
* Recursively call the quickSort function on the subarray from the starting index to the index before the mid element.
* Recursively call the quickSort function on the subarray from the index after the mid element to the ending index.
* End program
* **Hash Table Pseudocode:**
* **Part 1:**
* Begin program
* void LoadDataFile(Hashtable<Course> courses, String filePath) {
* -Open file at filePath
* -If file opened successfully
* -For each line in file
* -Split line into components separated by comma
* -If components size < 2
* -Print error about invalid format (not enough parameters)
* -Continue to next line
* -Else
* - Course c = new Course with course number and title from components
* -For index 2 to end of components
* -c.addPrerequisite(components[index])
* -End For
* -courses.insert(c.getCourseNumber(), c)
* -End If
* -Close file
* -Else
* -Print error about unable to open file
* -End If
* }
* - void PrintCourseInformation(Hashtable<Course> courses, String courseNumber) {
* - Enter For loop
* -For each course c in courses
* -For each prerequisite p in c.getPrerequisites()
* -If courses does not contain p
* -Print error about invalid prerequisite
* -If courses contains courseNumber
* -Course c = courses.get(courseNumber)
* -Print out the course information
* -For each prerequisite of the course
* -Print the prerequisite course information
* -End For
* -Else
* -Print error about course not found
* End If
* End program
* **Part 2.**
* Begin Program
* -Procedure LoadCoursesIntoHashTable(Hashtable<Course> &courses, String filePath)
  + - Open file at filePath

- If file opened successfully

- For each line in file

- Split line into array of strings, components, separated by commas

- If size of components is less than 2

- Print error about invalid line format

- Continue to next line

- End If

- End program

- Set course.number to components[0

- Set course.title to components[1]

-If size of components is greater than 2

- For index from 2 to end of components

- Add components[index] to course.prerequisites

- End -Add course to courses hashtable with key as course.number-Close file

**Part 3;**

* Begin program
* -int numPrerequisiteCourses(Hashtable<Course> courses, String courseNumber) {
* -if courses contains key courseNumber
* -course = courses.get(courseNumber)
* -totalPrerequisites = prerequisites of course
* -for each prerequisite p in totalPrerequisites
* -if courses contains key p
* -add prerequisites of courses.get(p) to totalPrerequisites
* -print number of totalPrerequisites
* -else
* -print error about course not found
* }
* -void printSampleSchedule(Hashtable<Course> courses) {
* -for each courseNumber in courses.keys
* -course = courses.get(courseNumber)
* -print course.number, course.title
* -for each prerequisite in course.prerequisites
* -if courses contains key prerequisite
* -prerequisiteCourse = courses.get(prerequisite)
* -print prerequisiteCourse.number, prerequisiteCourse.title
* }
* -void printCourseInformation(Hashtable<Course> courses, String courseNumber) {
* -if courses contains key courseNumber
* -course = courses.get(courseNumber)
* -print out the course.number and course.title
* -for each prerequisite of the course
* -if courses contains key prerequisite
* -prerequisiteCourse = courses.get(prerequisite)
* -print the prerequisiteCourse.number and prerequisiteCourse.title
* -else
* -print error about course not found
* End program

**Tree Pseudocode:**

**Part 1:**

* Begin program

-Void LoadCourses(Tree<Course> &courses, String filePath)

-Open file at filePath

-If file opened successfully

-For each line in file

- Split line into array of strings, components

-If size of components is less than 2

-Print error about invalid line format

-Continue to next line

-Close file

-Else

-Print error about file not being opened

-End loop

- For each prerequisite of the targetCourse

- Create a Course object, prerequisiteCourse

- prerequisiteCourse = search the courses tree for course with prerequisite

-If prerequisiteCourse is not null

-Print the prerequisite course information

-Else

-Print error about invalid prerequisite

-Else

-Print error about course not found

-End If

-End loop

-Part 2:

-Void PrintCourseInfo(Tree<Course> courses, String courseNumber

- Create a Course object, targetCourse

- Define object integer, data type

- Initialize function

-targetCourse = search the courses tree for course with courseNumber

-If targetCourse is not null

-Print out the course information (course number, title)

-Insert course into vector data structure

- End program

Part 3:

* Begin program

-Void numPrerequisites(Tree<Course> courses)

- Create a list totalPrerequisites and assign it to prerequisites of courses

- For each prerequisite p in totalPrerequisites

- Search courses tree for course with courseNumber equal to p

-If course found, add its prerequisites to totalPrerequisites

- End For

-Print the number of totalPrerequisites

-End loop

-Void printCourseInfo(Tree<Course> courses, String courseNumber)

-Search courses tree for course with courseNumber

-If course found

-Print out the course information

-For each prerequisite of the course

-Search courses tree for course with prerequisite

-If course found, print the prerequisite course information

-End For

-Else

-Print error message about course not found

-End If

-End loop

- End program

**2: Create pseudocode for a menu**

1. Begin Program and create Course object

2. Load the Data Structure from file:

3. Create function "LoadDataStructure":

4. Open data file

5. If file is open:

6. While not end of file:

7. Read line into "Course" object

8. Add "Course" object to "courseList" data structure

9. End While

10. Close file

11. Else:

12. Print "Error: Could not open file."

13. End Function

14. Create function "PrintCourseList":

15. For each "Course" in "courseList":

16. Print "Course" details

17. End For

18. End Function

19. Create function "PrintCourse" takes "courseCode":

20. For each "Course" in "courseList":

21. If "Course.code" is equal to "courseCode":

22. Print "Course" details

23. End If

24. End For

25. End Function

26. While true:

27. Print "Menu:"

28. Print "1. Load Data Structure"

29. Print "2. Print Course List"

30. Print "3. Print Course"

31. Print "4. Exit"

32. Take input into "choice"

33. If "choice" is 1:

34. Call "LoadDataStructure"

35. ElseIf "choice" is 2:

36. Call "PrintCourseList"

37. ElseIf "choice" is 3:

38. Print "Enter course code:"

39. Take input into "courseCode"

40. Call "PrintCourse" with "courseCode"

41. ElseIf "choice" is 4:

42. Break

43. Else:

44. Print "Invalid choice. Try again."

45. End If

46. End Loop

47. End Program

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

**Vector pseudocode**

Create function "sortCourses" that takes (Vector<Course> courses):

For i from 0 to size of courses-1:

For j from i+1 to size of courses:

If courses[i].number is greater than courses[j].number:

Swap courses[i] and courses[j]

End For

End Function

Create function "printSortedCourses" that takes (Vector<Course> courses):

Call "sortCourses" with "courses"

For each "Course" in "courses":

Print "Course" number, name, and other details

End For

End Function

**Hashtable pseudocode**

Create function "sortCourses" that takes (Hashtable<Course> courses):

Initialize a Hashtable<Course> sortedCourses

For each "Course" in "courses":

Insert "Course" into "sortedCourses"

End For

Call "sortCourses" with "sortedCourses"

End Function

Create function "printSortedCourses" that takes (Hashtable<Course> courses):

Hashtable<Course> sortedCourses = Call "sortCourses" with "courses"

For each "Course" in "sortedCourses":

Print "Course" number, name, and other details

End For

End Function

**Tree pseudocode**

Create function "inOrderTraversal" that takes (Tree<Course> node, Vector<Course> sortedCourses):

If node is not null:

Call "inOrderTraversal" with node's left child

Add node's "Course" to "sortedCourses"

Call "inOrderTraversal" with node's right child

End If

End Function

Create function "sortCourses" that takes (Tree<Course> courses):

Initialize a Tree<Course> sortedCourses

Call "inOrderTraversal" with root of "courses" and "sortedCourses"

Return "sortedCourses"

End Function

Create function "printSortedCourses" that takes (Tree<Course> courses):

Tree<Course> sortedCourses = Call "sortCourses" with "courses"

For each "Course" in "sortedCourses"

Print "Course" number, name, and other details

End For

End Function

**4. Evaluate the run-time and memory of data structures that could be used to address the requirements.**

Opening a file is a constant time operation, O(1). Iterating through each line in the file is a linear time operation, O(n), where n is the number of lines in the file.

Splitting a line into components and checking the size of components are linear time operations, O(n), where n is the number of components.

Adding a course to a hashtable is usually a constant time operation, O(1) but in the worst-case scenario (if all keys hash to the same index), it can become a linear time operation, O(n), where n is the number of entries in the hashtable.

* “Open file at filePath": This is a constant time operation, O(1).
* "For each line in file": This will run n times, where n is the number of lines in the file. So, its cost is n.
* "Split line into array of strings, components, separated by commas": This operation is done n times and each operation takes m time, where m is the number of components in a line. So, its cost is n\*m.
* "If size of components is less than 2", "Print error about invalid line format", and "Continue to next line": These operations are done n times and each operation takes constant time. So, their costs are n.
* "Create new Course object, course", "Set course.number to components[0]", and "Set course.title to components[1]": These operations are done n times and each operation takes constant time. So, their costs are n.
* "If size of components is greater than 2" and "For index from 2 to end of components": These operations are done n times and each operation takes m time, where m is the number of components in a line. So, their costs are n\*m.
* "Add components[index] to course.prerequisites": This operation is done nm times and each operation takes constant time. So, its cost is nm.
* "Add course to courses hashtable with key as course.number": This operation is done n times and each operation takes constant time on average. So, its cost is usually n. But in the worst-case scenario, its cost is n^2.
* "Close file": This is a constant time operation, O(1).

**5: Evaluation;**

Advantages of using a vector include dynamic size which can be useful if the number of courses is not known in advance and random access which can be useful if you need to access courses by index. Vectors are also often simple to use compared to other data structures. Disadvantages of using a vector include sorting because If you want to keep your data sorted, you'll need to sort the vector after each insertion which can be costly (O(n log n) time), and search. Searching for a specific course would take O(n) time as you'd need to iterate through the whole vector.

Advantages of using a hash table include fast access and insertion, deletion, and retrieval can be done in O(1) time. It is also good for efficient searching, it is efficient for printing a specific course with its prerequisites as you can retrieve a course by its course number in O(1) time. Disadvantages of using a hash table include hash tables not keeping data in sorted order. So, it's not efficient to print the course list in sorted order. Space complexity is another disadvantage for using hash tables as they can use more memory due to the need for a load factor to reduce the number of collisions. This might be inefficient if memory is a concern.

Advantages of using a tree include that they keep elements in sorted order, which makes it efficient for printing the course list in sorted order. They are also logarithmically efficient. If you use a binary search tree, insert, delete, and search operations can be done in O(log n) time. Disadvantages of using a tree include being more complex to implement and understand compared to vectors and hash tables and Inefficient for retrieval by key: Unlike hash tables, binary search trees would take O(log n) time to retrieve a course by its course number.

**6. Recommendation:**

I recommend using a tree data structure, specifically a binary search tree to print a list of all the Computer Science courses in alphanumeric order. binary search trees maintain order of their elements and would provide an efficient way to print courses in sorted order. In an optimally balanced binary search tree, insertions, deletions, and retrievals are done in O(log n) time complexity. This is an efficient way to maintain sorted data compared to other data structures like vectors or hash tables, which would need a full sorting pass (O(n log n) time complexity) before printing in sorted order. The course number could be used as the key for the binary search tree.