MENU PSEUDOCODE:

**Function main() (Menu)**

Initialize a variable courses to hold the data structure.

Initialize choice to 0.

Enter a loop that continues until choice is 9.

Display the menu options:

* + - 1. Load Data
      2. Print All Courses
      3. Print Course Info
      4. Exit

Read the user's choice.

Use a switch statement on choice:

**Case 1:** Call loadCourseData and assign the result to courses.

**Case 2:** If courses is not empty, call the printSortedCourses function for data structure.

**Case 3:** If courses is not empty, prompt for a course number and call the correct searchCourse function.

**Case 9:** Exit the program.

**Default:** Print "Invalid choice."

FILE HANDLING PSEUDOCODE:

* **Function loadCourseData(filename):**
  + **Input:** filename (string)
  + **Output:** The data structure (vector, hash table, or tree root).

Initialize an empty data structure

Initialize an empty list called allCourseNumbers to store course numbers for prerequisite checks

Open the file based on filename. If the file can’t open, print an error and close out iof the program.

**First Pass**

* + - * Read the file
      * For each line, split it into tokens based on the comma
      * Check for file format errors: if the number of tokens is less than 2, print "format error" message
      * Add the course number to allCourseNumbers

Close the file.

Re-open the file to start again

**Second Pass**

* + - * Read the file line by line again.
      * Split the line into tokens.
      * Extract the courseNumber, courseTitle, and remaining tokens as prereqs
      * For each prerequisite, ensure it exists in allCourseNumbers list. if a prerequisite is not found, print a "prerequisite does not exist" error
      * if checks pass, create a new Course object with the extracted data.
      * **Store the object:**
        + For a **vector**, send the new Course object to the vector.
        + For a **hash table**, insert the new Course object using courseNumber as the key.
        + For a **binary search tree**, insert the new Course object into the tree.

Close the file.

Return data structure.

Runtime Analysis and Recommendation:

n = courses

We are evaluating the wort-case run time for file loading and object creation per each different structure.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ***Structure*** | ***Cost per Line*** | ***# executes*** | ***Total cost*** | ***Runtime*** |
| Vector | 1 parse, 1 push | n | 2n | O(n) |
| Hash Table | 1 parse, 1 hash/insert | n | 2n | O(n) |
| Binary Search Tree | 1 parse, O(n) insert | n | 2xO(n) | O(n^2) |

Based on the advisor’s requirements, there are many advantages and disadvantages to vectors, hash tables, and tree data structures.

Vector:

Advantages: Simple to use, good on memory

Disadvantages: Inefficient search method (O(n)), and requires Sort for the “Print all Courses” function

Hash Table:

Advantages: Very fast for searches, (O(1)) and insertion (O(1))

Disadvantages: Data is stored in unorganized manner. To print an organized list will require moving objects to vector first, and then sorting.

Binary Search Tree:

Advantages: Average search performance and insertion performance. Can print a sorted list without needing to specify a sort.

Disadvantages: Bad run-time for loading. Complex implementation.

Recommendation:

Overall, I would recommend a binary search tree. The advisor’s requirements are to print a sorted list and search for a certain course. The search tree checks both of these boxes. It can print an entire list in O(n) time, and does not require a sort like vectors and hash tables. Yes, hash-table is faster case search, the tree will suffice and be able create a list that is sorted quickly.