**Pseudocode**

// Menu Implementation

// Main Menu

While true

Print "Menu:"

Print "1. Load Course Data"

Print "2. Print All Courses (Alphanumerically)"

Print "3. Print Course Details"

Print "9. Exit"

Input choice

If choice == 1

// Load Data

Call LoadData()

Else If choice == 2

// Print Sorted List

Call PrintSortedCourses()

Else If choice == 3

// Print Course Details

Input courseNumber

Call PrintCourseDetails(courseNumber)

Else If choice == 9

Print "Exiting program."

Exit

Else

Print "Invalid option. Please choose again."

// Load Data (Common for all data structures)

Function LoadData()

Open file "courses.txt"

For each line in file

Split line by comma

Create a Course object with courseNumber, name, and prerequisites

Add Course object to the data structure

Close file

Print "Course data loaded successfully."

// Print All Courses (Vector Implementation)

Function PrintSortedCourses\_Vector()

Sort courses by courseNumber (ascending)

For each course in courses

Print course.courseNumber + ": " + course.name

// Print All Courses (Hash Table Implementation)

Function PrintSortedCourses\_HashTable()

Extract all courseNumbers from hash table

Sort courseNumbers (ascending)

For each courseNumber in sorted list

Print hashTable[courseNumber].courseNumber + ": " + hashTable[courseNumber].name

// Print All Courses (Binary Search Tree Implementation)

Function PrintSortedCourses\_Tree()

Call InOrderTraversal(tree.root)

// Helper for InOrder Traversal

Function InOrderTraversal(node)

If node is not null

InOrderTraversal(node.left)

Print node.courseNumber + ": " + node.name

InOrderTraversal(node.right)

// Print Course Details (Vector Implementation)

Function PrintCourseDetails\_Vector(courseNumber)

For each course in courses

If course.courseNumber equals courseNumber

Print course.name

Print "Prerequisites: " + course.prerequisites

Exit

Print "Course not found."

// Print Course Details (Hash Table Implementation)

Function PrintCourseDetails\_HashTable(courseNumber)

If hashTable contains key courseNumber

Print hashTable[courseNumber].name

Print "Prerequisites: " + hashTable[courseNumber].prerequisites

Else

Print "Course not found."

// Print Course Details (Binary Search Tree Implementation)

Function PrintCourseDetails\_Tree(courseNumber)

Node = SearchBST(tree.root, courseNumber)

If Node is not null

Print Node.courseNumber + ": " + Node.name

Print "Prerequisites: " + Node.prerequisites

Else

Print "Course not found."

// Helper for BST Search

Function SearchBST(node, courseNumber)

If node is null or node.courseNumber equals courseNumber

Return node

If courseNumber < node.courseNumber

Return SearchBST(node.left, courseNumber)

Else

Return SearchBST(node.right)

**Runtime Analysis**

Vector

* Reading a file
  + For “n” lines in the file, the loop iterates once per line. Cost: O(n).
* Parsing data
  + Splitting a line with commas has a linear cost proportional to the number of fields each line has. Cost per line is O(1).
* Creating and storing course objects
  + Making an object takes O(1) and appending it to a vector is also O(1).

Hash table

* Reading a file
  + For “n” lines in the file, the loop iterates once per line. Cost: O(n).
* Parsing data
  + Splitting a line with commas has a linear cost proportional to the number of fields each line has. Cost per line is O(1).
* Creating and storying course objects
  + Creating the object is O(1) and inserting it into a hash table is O(1) on average.

Binary Search tree

* Reading a file
  + For “n” lines in the file, the loop iterates once per line. Cost: O(n).
* Parsing data
  + Splitting a line with commas has a linear cost proportional to the number of fields each line has. Cost per line is O(1).
* Creating and storying course objects
  + Creating a object is O(1). Inserting it into a balance binary search tree takes O(log n) for each course.

Memory usage.

Vector – Memory is proportional to the number of objects. There needs to be additional memory overhead for vector resizing. This is easy to implement and ideal for sequential access. However, searching for specific course is O(n) which is inefficient for large datasets.

Hash table – is the fastest for looking up course details O(1) and you will have direct mapping of course numbers to objects. Yet, sorting alphanumeric lists takes O(n log n) and has memory overhead for hash collisions.

Binary Search tree - Efficient search O(log n) and automatic alphanumeric ordering via in-order traversal. The disadvantage here is it has a slower insertion.

It would be recommended to utilize the Hash table since it offers the fastest lookup, has better insertion and sorting times, and better memory usage. The balance between runtime efficiency and usability makes the hash table the best choice for this application.

**Runtime Chart**

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **Vector** | **Hash table** | **Binary Search tree** |
| Reading file | O(n) | O(n) | O(n) |
| Parsing data | O(n) | O(n) | O(n) |
| Creating course object | O(n) | O(n) | O(n) |
| Inserting into data structure | O(1) per item  (O(n)) | O(1) per item  (O(n)) | O(log n) per item  (O(n log n)) |
| Overall Runtime | O(n) | O(n) | O(log n) per item  (O(n log n)) |