**MENU**

**Function Main():**

**While True:**

**Display "Select an option:"**

**Display "1. Load Data Structure"**

**Display "2. Print Course List"**

**Display "3. Print Course"**

**Display "4. Exit"**

**Option = Read user input**

**If Option is "1":**

**Display "Select data structure: Vector(1), Hash Table(2), Tree(3)"**

**DataStructureOption = Read user input**

**FileName = Prompt for file name**

**LoadIntoDataStructure(FileName, DataStructureOption)**

**ElseIf Option is "2":**

**PrintCourseList(DataStructureOption)**

**ElseIf Option is "3":**

**CourseCode = Prompt for course code**

**PrintCourseDetails(CourseCode, DataStructureOption)**

**ElseIf Option is "4":**

**Exit program**

**Else:**

**Display "Invalid option"**

**End Function**

**Hash Table Data Structure:**

Function LoadIntoHashTable(fileName: String):

Open file with name fileName for reading

If file fails to open:

Display error message

Return

End If

While not end of file:

Read line

If line is properly formatted:

Course course = ParseCourse(line)

courseHashTable.Insert(course.code, course)

Else:

Display error for the line

End If

End While

Close file

End Function

Function PrintCourseListHashTable():

List<Course> courses = courseHashTable.Values()

Sort courses by course code

For each course in courses:

Print course.code and course.title

End For

End Function

Function PrintCourseHashTable(courseCode: String):

Course course = courseHashTable.Get(courseCode)

If course is not null:

Print course.title and course.prerequisites

Else:

Display "Course not found"

End If

End Function

**Vector Data Structure:**

**Function LoadIntoVector(fileName: String):**

**Open file with name fileName for reading**

**If file fails to open:**

**Display error message**

**Return**

**End If**

**While not end of file:**

**Read line**

**If line is properly formatted:**

**Course course = ParseCourse(line)**

**Append course to courseVector**

**Else:**

**Display error for the line**

**End If**

**End While**

**Close file**

**End Function**

**Function PrintCourseListVector():**

**Sort courseVector by course code**

**For each course in courseVector:**

**Print course.code and course.title**

**End For**

**End Function**

**Function PrintCourseVector(courseCode: String):**

**For each course in courseVector:**

**If course.code equals courseCode:**

**Print course.title and course.prerequisites**

**Return**

**End If**

**End For**

**Display "Course not found"**

**End Function**

TREE DATA STRUCTURE

**Function LoadIntoTree(fileName: String):**

**Open file with name fileName for reading**

**If file fails to open:**

**Display error message**

**Return**

**End If**

**While not end of file:**

**Read line**

**If line is properly formatted:**

**Course course = ParseCourse(line)**

**courseTree.Insert(course.code, course)**

**Else:**

**Display error for the line**

**End If**

**End While**

**Close file**

**End Function**

**Function PrintCourseListTree():**

**courseTree.InOrderTraversal(PrintCourseNode)**

**End Function**

**Function PrintCourseNode(node: TreeNode):**

**Print node.value.code and node.value.title**

**End Function**

**Function PrintCourseTree(courseCode: String):**

**Course course = courseTree.Search(courseCode)**

**If course is not null:**

**Print course.title and course.prerequisites**

**Else:**

**Display "Course not found"**

**End If**

**End Function**

**Utility function:**

Function ParseCourse(line: String) -> Course:

If line format does not match expected format:

Return null

Extract data from line (e.g., using delimiters or known structure)

Create a new Course object with the extracted data

Return the new Course object

End Function

**Runtime Analysis**

| **Operation/Method** | **Vector** | **Hash Table** | **BST Tree** |
| --- | --- | --- | --- |
| **Load Data (Insertion)** | **O(1)\*** | **O(1)\*** | **O(log n) or O(n)** |
| **Sorting for Printing Course List** | **O(n log n)** | **O(n log n)** | **N/A (Inherent Order)** |
| **Search by Course Code (Print Course)** | **O(n)** | **O(1)\*\*** | **O(log n) or O(n)** |

The O(1) here is the average time for insertion. In real-world scenarios, vectors (like arrays) might occasionally need more time to resize, but this is amortized O(1). Similarly, hash tables can have O(1) on average for insertion, but it could be O(n) in worst-case scenarios (like when handling collisions or resizing).

The O(1) here for hash tables assumes good hashing without many collisions. In the worst case, if many items hash to the same key, this could degrade to O(n).

The "O(log n) or O(n)" for the BST Tree signifies that in a balanced tree, operations can be as fast as O(log n). However, in the worst-case scenario where the tree becomes unbalanced, it could degrade to O(n).

**Pseudocode Evaluation:** My pseudocode described the process of opening a file, reading data, parsing each line, and checking for formatting errors. The function **ParseCourse** is designed for parsing and checking formatting, and the **LoadIntoX** functions (where X is the data structure type) cover opening and reading the file.

**2. Run-Time Analysis:**

When analyzing the worst-case running time:

For reading a file and creating course objects:

* The cost of reading a line is constant, say **c1**.
* The cost of parsing a line and checking for errors is also constant, say **c2**.
* The cost of inserting into data structure varies: **c3** for vector (average), **c4** for hash table (average), and **c5** or **c6** for BST (depending on balanced vs. unbalanced).

If there are **n** courses, the worst-case running time for loading all courses into the data structure:

* Vector: �×(�1+�2+�3)*n*×(*c*1+*c*2+*c*3)
* Hash Table: �×(�1+�2+�4)*n*×(*c*1+*c*2+*c*4)
* BST: �×(�1+�2+�6)*n*×(*c*1+*c*2+*c*6) [taking the worst-case unbalanced scenario]

**3. Analysis of Each Data Structure:**

**Vector:** *Advantages:*

* Continuous memory, which is cache-friendly.
* Amortized O(1) for insertions at the end.

*Disadvantages:*

* Searching is O(n) unless sorted and then applied binary search.
* Need to resize when capacity is full, which is an overhead.

**Hash Table:** *Advantages:*

* Average O(1) for insertions, deletions, and search.
* Direct mapping helps in quick access.

*Disadvantages:*

* Collisions can decrease efficiency.
* Memory overhead for storage of keys and handling collisions.

**BST:** *Advantages:*

* Maintains order, which can be useful for certain operations.
* O(log n) operations if the tree is balanced.

*Disadvantages:*

* Can degrade to O(n) if the tree becomes unbalanced.
* Requires pointers to left and right children, increasing memory overhead.

**4. Recommendation:**

For this specific application, where the operations seem to be a mix of loading data, searching for specific courses, and displaying a sorted list:

* **Hash Table** might be the best choice for quick insertions and fetching a specific course. However, it will have overhead when sorting courses for display.
* If displaying sorted courses frequently is a primary operation, then a balanced **Binary Search Tree** (like AVL or Red-Black Tree) would be recommended as they inherently maintain order and offer O(log n) performance for most operations.