**Lance Tran**

**CS-300**

**Project 1**

**Pseudocode**

**Function Main()**  
 DECLARE filePath AS STRING  
 DECLARE menuChoice, dataChoice AS INTEGER  
 DECLARE courses AS Collection<Course> // The generic collection to hold our data  
  
 // Handle command-line arguments  
 IF HasCommandLineArgument()  
 filePath = GetCommandLineArgument()  
 ELSE  
 filePath = "default.csv"  
 END IF  
  
 // Main menu loop  
 DO  
 Output "Menu Options:"  
 Output "1. Load data"  
 Output "2. Validate data"  
 Output "3. Search for course"  
 Output "4. Print courses"  
 Output "9. Exit"  
 Input menuChoice  
  
 IF menuChoice IS NOT 9  
 Output "Choose data structure:"  
 Output "1. Vector"  
 Output "2. Hash Table"  
 Output "3. Binary Search Tree"  
 Input dataChoice  
 END IF  
  
 SWITCH menuChoice  
 CASE 1: // Load data  
 courses = LoadCourses(filePath, dataChoice)  
 IF courses IS NOT NULL  
 Output "Successfully loaded " + courses.Count() + " records."  
 END IF  
 CASE 2: // Validate data  
 IF courses IS NOT NULL  
 courses.Validate()  
 ELSE  
 Output "No data loaded yet."  
 END IF  
 CASE 3: // Search and print course  
 IF courses IS NOT NULL  
 Output "Enter course ID to search:"  
 Input userSearch AS STRING  
 courses.PrintCourse(userSearch)  
 ELSE  
 Output "No data loaded yet."  
 END IF  
 CASE 4: // Print all courses  
 IF courses IS NOT NULL  
 courses.PrintAll()  
 ELSE  
 Output "No data loaded yet."  
 END IF  
 CASE 9: // Exit  
 Output "Good bye."  
 DEFAULT:  
 Output "Invalid menu choice. Please try again."  
 END SWITCH  
 WHILE menuChoice IS NOT 9  
End Function  
  
  
// --- Data structure classes (Polymorphism) ---  
  
// The abstract base class for all data structures  
Class Collection<Course>  
 METHOD ABSTRACT Load(filePath AS STRING)  
 METHOD ABSTRACT Validate()  
 METHOD ABSTRACT PrintCourse(courseID AS STRING)  
 METHOD ABSTRACT PrintAll()  
 METHOD ABSTRACT Count() AS INTEGER  
End Class  
  
// The specific implementations for each data structure  
Class VectorCollection inherits Collection<Course>  
 // Internal state: A vector to hold the course data  
 DECLARE courseList AS VECTOR<Course>  
  
 METHOD Load(filePath AS STRING)  
 // ... (Implementation for reading CSV into vector)  
 METHOD Validate()  
 // ... (Implementation for validating courses in vector)  
 METHOD PrintCourse(courseID AS STRING)  
 // ... (Implementation for searching/printing from vector)  
 METHOD PrintAll()  
 // This method will sort the vector before printing  
 // Calls sortList() on courseList  
 METHOD Count() AS INTEGER  
 // Returns the size of courseList  
End Class  
  
Class HashTableCollection inherits Collection<Course>  
 // Internal state: The hash table  
 DECLARE courseTable AS HashTable  
  
 METHOD Load(filePath AS STRING)  
 // ... (Implementation for reading CSV into hash table)  
 METHOD Validate()  
 // ... (Implementation for validating courses in hash table)  
 METHOD PrintCourse(courseID AS STRING)  
 // ... (Implementation for searching/printing from hash table)  
 METHOD PrintAll()  
 // This method must extract all courses, sort them, and then print  
 // Create temp vector, copy courses from hash table buckets to vector  
 // Call sortList() on the temp vector  
 // Print the sorted temp vector  
 METHOD Count() AS INTEGER  
 // Returns the number of courses in the hash table  
End Class  
  
Class BSTCollection inherits Collection<Course>  
 // Internal state: The Binary Search Tree  
 DECLARE bst AS BinarySearchTree  
  
 METHOD Load(filePath AS STRING)  
 // ... (Implementation for reading CSV and inserting into BST)  
 METHOD Validate()  
 // ... (Implementation for validating courses in BST)  
 METHOD PrintCourse(courseID AS STRING)  
 // ... (Implementation for searching/printing from BST)  
 METHOD PrintAll()  
 // This uses the BST's in-order traversal to print in alphabetical order  
 // Calls the recursive printTree() on the BST  
 METHOD Count() AS INTEGER  
 // Returns the number of courses in the BST  
End Class  
  
  
// --- Helper function for loading ---  
Function LoadCourses(filePath AS STRING, dataStructureChoice AS INTEGER) AS Collection<Course>  
 DECLARE courses AS Collection<Course>  
  
 SWITCH dataStructureChoice  
 CASE 1:  
 courses = NEW VectorCollection()  
 CASE 2:  
 courses = NEW HashTableCollection()  
 CASE 3:  
 courses = NEW BSTCollection()  
 DEFAULT:  
 Output "Invalid data structure choice."  
 RETURN NULL  
 END SWITCH  
  
 courses.Load(filePath)  
 RETURN courses  
End Function  
  
  
// --- Corrected or simplified helper functions ---  
  
Function printTree(node AS Node) // In-order traversal  
 IF node IS NULL THEN RETURN  
 printTree(node.left)  
 Output node.course.courseID, node.course.courseName  
 // Loop through preList and output prereq courses  
 printTree(node.right)  
End Function  
  
Function partition(vector, lowestIndex, highestIndex)  
 pivot = vector[highestIndex] // Simple pivot selection  
 i = lowestIndex - 1 // Index of smaller element  
  
 FOR j = lowestIndex TO highestIndex - 1  
 IF vector[j].courseID <= pivot.courseID  
 i++  
 Swap(vector[i], vector[j])  
 END IF  
 END FOR  
 Swap(vector[i + 1], vector[highestIndex])  
 RETURN i + 1  
End Function  
  
// Other functions like `sortList()` (`quicksort`) can be kept but named more clearly  
// and referencing the new class structures.

# Run Time Analysis

Vector:

Loading: O(1)

Search: O(n)

Sorting/Printing: O(n log n) on specific sort methods

Hash:

Loading: O(1)-O(n)

Search: O(1)-O(n)

Sorting/Printing: O(n)

Binary Tree

Loading: O(log n)  
  
Search: O(log n) - O(n)

Sorting/Printing: O(n)

# Advantage Analysis

Vector

Pros: Vectors are really straightforward conceptually as you are allocating contiguous spaces. Retrieving information is a constant, so it’s extremely fast, loading data is like knowing where a book is in a library.

Cons: Very inefficient when shuffling, deleting or inserting data is slow, you must determine usage size from the beginning and incorrectly estimating is costly, because allocated space cannot be dynamically used.

Binary Tree

Pros: Traversal provides a natural order sorting, so that also makes it good for querying over a range. While having more memory overhead, it also is more on the loading side than on the operation or resizing spaces. Because the it is log n, time complexity does not scale quickly.

Cons: It needs to be balanced, otherwise the load and search times will skew on the long side. Dynamically balancing a binary tree would take a lot of overhead.

Hash

Pros: Fast on average, as in most cases the time complexity is linear, good for small datasets before n gets too large.

Cons: Poor design will result in frequent collisions, there’s no order, and resizing is an expensive operation.

# Recommendation

Because the program is something that looks up courses and prerequisites, having a hash data structure would provide the fastest searches. The concern about collisions occurring is minimal because courses aren’t frequently added. That also means that they don’t need to be resized very often. It would be easy to control the risk of collision. The main issue would be printing lots of courses at once.