**Jarrod Schantz**

**CS-300-10336-M01 DSA: Analysis and Design**

**10/11/2025**

**Part 1: Pseudocode**

First, let's define a common Course object that we'll use for all three data structures. This helps keep the logic consistent.

**Course Object Pseudocode**

STRUCTURE Course

STRING courseNumber

STRING name

VECTOR<STRING> prerequisites

END STRUCTURE

**1. Vector Implementation**

**File Reading & Data Loading**

FUNCTION LoadCoursesIntoVector(filePath)

// Create a vector to hold Course objects

VECTOR<Course> courses

// Open the file

OPEN file at filePath for reading

// Read file line by line

WHILE there is another line in the file

// Read the current line

line = READ line from file

[cite\_start]// Parse the comma-separated values [cite: 7]

tokens = SPLIT line by comma

// Check for formatting errors (at least 2 tokens: number and name)

IF size of tokens < 2 THEN

PRINT "Skipping malformed line: " + line

CONTINUE to next line

END IF

// Create a new Course object

Course newCourse

newCourse.courseNumber = tokens[0]

newCourse.name = tokens[1]

[cite\_start]// Add any prerequisites if they exist [cite: 8, 9]

FOR i from 2 to size of tokens - 1

ADD tokens[i] to newCourse.prerequisites

END FOR

// Add the new course object to our vector

ADD newCourse to courses vector

END WHILE

CLOSE file

RETURN courses

END FUNCTION

**Print Sorted Course List**

FUNCTION PrintSortedVector(courses)

// Use a standard sorting algorithm (like Quicksort or Mergesort)

// to sort the vector by courseNumber

SORT courses vector based on courseNumber in alphanumeric order

// Print the sorted list

PRINT "Course List:"

FOR each course in courses vector

PRINT course.courseNumber + ", " + course.name

END FOR

END FUNCTION

**Print Single Course Info**

FUNCTION PrintVectorCourseInfo(courses, courseNumToFind)

// Perform a linear search for the course

found = FALSE

FOR each course in courses vector

IF course.courseNumber equals courseNumToFind THEN

PRINT course.courseNumber + ", " + course.name

PRINT "Prerequisites: "

IF course.prerequisites is empty THEN

PRINT "None"

ELSE

FOR each prereq in course.prerequisites

PRINT prereq

END FOR

END IF

found = TRUE

BREAK // Exit loop once found

END IF

END FOR

IF found is FALSE THEN

PRINT "Course not found."

END IF

END FUNCTION

**2. Hash Table Implementation**

**File Reading & Data Loading**

FUNCTION LoadCoursesIntoHashTable(filePath)

// Create a hash table to hold Course objects

HASHTABLE<STRING, Course> courses

// File opening and parsing logic is identical to the vector version...

// ...

// After creating the newCourse object:

// Add the new course object to our hash table, using courseNumber as the key

INSERT newCourse into courses hash table with key newCourse.courseNumber

// ... end of while loop and file closing

RETURN courses

END FUNCTION

**Print Sorted Course List**

FUNCTION PrintSortedHashTable(courses)

// Hash tables are not inherently sorted. We must extract, sort, and print.

// 1. Create a temporary vector

VECTOR<Course> sortedList

// 2. Iterate through the hash table and copy elements to the vector

FOR each course in courses hash table

ADD course to sortedList vector

END FOR

// 3. Sort the temporary vector

SORT sortedList vector based on courseNumber in alphanumeric order

// 4. Print the sorted list

PRINT "Course List:"

FOR each course in sortedList vector

PRINT course.courseNumber + ", " + course.name

END FOR

END FUNCTION

**Print Single Course Info**

FUNCTION PrintHashTableCourseInfo(courses, courseNumToFind)

// Search for the course by its key (courseNumber)

IF courseNumToFind exists as a key in courses hash table THEN

course = GET course from hash table with key courseNumToFind

PRINT course.courseNumber + ", " + course.name

PRINT "Prerequisites: "

IF course.prerequisites is empty THEN

PRINT "None"

ELSE

FOR each prereq in course.prerequisites

PRINT prereq

END FOR

END IF

ELSE

PRINT "Course not found."

END IF

END FUNCTION

**3. Binary Search Tree Implementation**

**File Reading & Data Loading**

FUNCTION LoadCoursesIntoBST(filePath)

// Create a Binary Search Tree

BinarySearchTree courses

// File opening and parsing logic is identical to the vector version...

// ...

// After creating the newCourse object:

// Insert the new course object into our BST

courses.Insert(newCourse) // Assuming Insert is modified to take a Course object

// ... end of while loop and file closing

RETURN courses

END FUNCTION

// NOTE: The Insert/addNode functions from your file would need a slight

// modification to compare course.courseNumber instead of bid.bidId

**Print Sorted Course List**

// The InOrder traversal naturally prints the BST in sorted order.

FUNCTION PrintSortedBST(courses)

PRINT "Course List:"

courses.InOrder() // This will traverse and print each course in alphanumeric order

END FUNCTION

// Modify the provided inOrder traversal to print Course info

FUNCTION inOrder(node)

IF node is not null THEN

inOrder(node's left child)

PRINT node.course.courseNumber + ", " + node.course.name

inOrder(node's right child)

END IF

END FUNCTION

**Print Single Course Info**

FUNCTION PrintBSTCourseInfo(courses, courseNumToFind)

// Search for the course by courseNumber

foundCourse = courses.Search(courseNumToFind) // Assuming Search returns a Course object

IF foundCourse is not empty THEN

PRINT foundCourse.courseNumber + ", " + foundCourse.name

PRINT "Prerequisites: "

IF foundCourse.prerequisites is empty THEN

PRINT "None"

ELSE

FOR each prereq in foundCourse.prerequisites

PRINT prereq

END FOR

END IF

ELSE

PRINT "Course not found."

END IF

END FUNCTION

**Main Program Menu Pseudocode**

This pseudocode would be the main driver for the program, regardless of which data structure is chosen.

FUNCTION Main()

// Initialize a data structure (e.g., VECTOR<Course> allCourses)

// or HASHTABLE, or BST

choice = 0

WHILE choice is not 9

DISPLAY MENU

PRINT "1. Load Course Data"

PRINT "2. Print Course List"

PRINT "3. Print Course Info"

PRINT "9. Exit"

GET user input for choice

IF choice is 1 THEN

// Get file path from user

filePath = GET user input

// Call the appropriate Load function

allCourses = LoadCoursesIntoVector(filePath) // or HashTable/BST version

PRINT "Data Loaded."

ELSE IF choice is 2 THEN

IF allCourses is empty THEN

PRINT "Please load data first (Option 1)."

ELSE

// Call the appropriate Print Sorted function

PrintSortedVector(allCourses) // or HashTable/BST version

END IF

ELSE IF choice is 3 THEN

IF allCourses is empty THEN

PRINT "Please load data first (Option 1)."

ELSE

// Get course number to find from user

courseNum = GET user input

// Call the appropriate Print Info function

PrintVectorCourseInfo(allCourses, courseNum) // or HashTable/BST version

END IF

ELSE IF choice is 9 THEN

PRINT "Exiting program."

ELSE

PRINT "Invalid option. Please try again."

END IF

END WHILE

END FUNCTION

**Part 2: Evaluation and Recommendation**

Here is the runtime analysis for the **file loading and data structure creation** portion of the pseudocode.

**Runtime Analysis (Big O)**

Let n be the number of courses in the input file.

**1. Vector**

| **Code (from LoadCoursesIntoVector)** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| VECTOR<Course> courses | 1 | 1 | 1 |
| WHILE there is another line | 1 | n | n |
| line = READ line | 1 | n | n |
| tokens = SPLIT line | c₁ (constant) | n | c₁n |
| Course newCourse | 1 | n | n |
| ADD newCourse to courses vector | 1 (amortized) | n | n |
| **Total Cost** |  |  | **4n + c₁n + 1** |
| **Worst-Case Runtime** |  |  | **O(n)** |

**2. Hash Table**

| **Code (from LoadCoursesIntoHashTable)** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| HASHTABLE<STRING, Course> courses | 1 | 1 | 1 |
| WHILE there is another line | 1 | n | n |
| line = READ line | 1 | n | n |
| tokens = SPLIT line | c₁ (constant) | n | c₁n |
| Course newCourse | 1 | n | n |
| INSERT newCourse into courses | O(1) avg / O(n) worst | n | n (avg) / n² (worst) |
| **Total Cost (Average)** |  |  | **4n + c₁n + 1** |
| **Total Cost (Worst)** |  |  | **3n + c₁n + n² + 1** |
| **Worst-Case Runtime** |  |  | **O(n²) (average case is O(n))** |

**3. Binary Search Tree**

| **Code (from LoadCoursesIntoBST)** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| BinarySearchTree courses | 1 | 1 | 1 |
| WHILE there is another line | 1 | n | n |
| line = READ line | 1 | n | n |
| tokens = SPLIT line | c₁ (constant) | n | c₁n |
| Course newCourse | 1 | n | n |
| courses.Insert(newCourse) | O(log k) avg / O(k) worst | n (for k=1 to n) | n log n (avg) / n² (worst) |
| **Total Cost (Average)** |  |  | **3n + c₁n + n log n + 1** |
| **Total Cost (Worst)** |  |  | **3n + c₁n + n² + 1** |
| **Worst-Case Runtime** |  |  | **O(n²) (average case is O(n log n))** |

*(Note: The worst case for both Hash Table and BST insertion occurs with poor hash functions or sorted input data, respectively.)*

**Advantages and Disadvantages**

**Vector**

* **Advantages**: Simple to implement. Good memory locality which can be fast for iteration. Data loading is a very fast O(n).
* **Disadvantages**: Searching for a specific course (Option 3) is slow at O(n). Printing a sorted list (Option 2) requires a full O(n log n) sort operation every single time the user selects it.

**Hash Table**

* **Advantages**: Extremely fast for looking up a single course (Option 3), with an average time of O(1). Data loading is efficient on average at O(n).
* **Disadvantages**: It is inherently unordered. To print a sorted list (Option 2), it must copy all elements to a temporary vector and sort it, which is an expensive O(n log n) operation. The worst-case performance for loading and searching can be very poor (O(n²) and O(n) respectively).

**Binary Search Tree**

* **Advantages**: It keeps the elements sorted at all times. This makes printing a sorted list (Option 2) extremely efficient at O(n) using an in-order traversal. Searching for a single course (Option 3) is also very efficient, with an average time of O(log n).
* **Disadvantages**: The data loading process is slightly slower on average (O(n log n)) than the others. The main weakness is that if the input data is already sorted (or nearly sorted), the tree can become unbalanced and "skewed", degrading its performance to O(n) for searches and O(n²) for loading, which is no better than a linked list.

**Part 3: Recommendation**

Based on the requirements and the analysis, my recommendation is to use the **Binary Search Tree (BST)**.

**Justification:**

The program has two primary functions for the user: printing a sorted list and searching for a specific course.

* The **BST** excels at both. Printing the sorted list (Option 2) is a simple O(n) in-order traversal, which is the most efficient solution possible since all n items must be visited. Searching (Option 3) is a fast O(log n) on average.
* The **Hash Table** is the fastest for searching (O(1)), but it is very inefficient for the sorted list requirement (O(n log n)), which is a major part of the program.
* The **Vector** is the slowest for both key requirements, with O(n log n) for sorting and O(n) for searching.

While the BST has a potential worst-case scenario if the input data is sorted, this is a risk that can be mitigated by using a self-balancing BST (like an AVL or Red-Black Tree) in a real-world implementation. For this academic project, the standard BST provides the best balance of performance across all required features, making it the most suitable choice.

I hope this detailed breakdown is helpful for your project! You should be able to copy this structure and content into a Word document for your submission. Let me know if you have any other questions.