**Project One**

**Cheyenne Anderson**

**CS-300 Analysis and Design**

**Professor Grabowski**

**October 12, 2025**

**Pseudocode**

**Menu and Main Program Pseudocode**

// Define Course object structure

STRUCT Course {

STRING courseNumber

STRING courseTitle

LIST<STRING> prerequisites // Store prerequisites as a list of course numbers

}

// Global variable for data structure

// For Vector

Vector<Course> coursesVector

// For Hash Table

HashTable<Course> coursesHashTable

// For Tree (Binary Search Tree - BST)

Tree<Course> coursesTree

// Main Program

FUNCTION main()

// for final program, only one data structure will be chosen

// For this design, we'll use a placeholder variable

DATA\_STRUCTURE courses

DO

printMenu()

INTEGER choice = getMenuChoice()

SWITCH (choice)

CASE 1: // Load File Data

STRING fileName = getInput("Enter the filename to load: ")

// Choose the appropriate loading function based on the chosen structure

courses = loadData(fileName)

BREAK

CASE 2: // Print Alphanumerically Ordered List

printSortedCourses(courses)

BREAK

CASE 3: // Print Course Information and Prerequisites

STRING courseNum = getInput("Enter the course number: ")

searchCourse(courses, courseNum)

BREAK

CASE 9: // Exit

print("Exiting program.")

BREAK

DEFAULT:

print("Invalid choice. Please try again.")

END SWITCH

WHILE choice != 9

END FUNCTION

FUNCTION printMenu()

print("\nMenu:")

print("1: Load Data File")

print("2: Print Course List")

print("3: Print Course Information")

print("9: Exit")

END

**File Loading and Course Object Creation Pseudocode**

// Function to handle the full file loading and data structure

// The return type changes based on the chosen structure

// e.g., Vector<Course> for vector, HashTable<Course> for hash table

FUNCTION loadData(STRING fileName) RETURNS DATA\_STRUCTURE

DATA\_STRUCTURE courses // Placeholder for the data structure (Vector, Hash Table, or Tree)

FILE inputFile = openFile(fileName)

IF inputFile IS NOT open THEN

print("Error: Could not open file " + fileName)

RETURN courses // Return empty structure

END IF

WHILE NOT endOfFile(inputFile) DO

STRING line = readLine(inputFile)

// Check formatting errors (simple check: must contain at least a course number and title)

IF line IS empty OR line DOES NOT contain "," THEN

print("Warning: Skipping malformed line: " + line)

CONTINUE

END IF

Course newCourse = parseLine(line)

// Insertion logic specific to the data structure

// For Vector:

// insertIntoVector(courses, newCourse)

// For Hash Table:

// insertIntoHashTable(courses, newCourse)

// For Tree (BST):

// insertIntoTree(courses, newCourse)

END WHILE

closeFile(inputFile)

print("Data loaded successfully.")

RETURN courses

END

// Pseudocode to parse a single line and create a Course object

FUNCTION parseLine(STRING line) RETURNS Course

LIST<STRING> tokens = split(line, ',')

Course course

// Check if there are enough tokens (Course Number, Course Title, Prereqs...)

IF length(tokens) >= 2 THEN

course.courseNumber = trim(tokens[0])

course.courseTitle = trim(tokens[1])

// The remaining tokens are prerequisites

FOR i FROM 2 TO length(tokens) - 1 DO

add(course.prerequisites, trim(tokens[i]))

END FOR

ELSE

// Handle error or return a partially formed object

print("Error in parsing line: not enough fields.")

END IF

RETURN course

END

**Print Course Information and Prerequisites Pseudocode**

This is a resubmission/adaptation of the required search and print function.

// Vector search

void searchCourse(Vector<Course> courses, String courseNumber) {

for all courses in courses // O(N) worst-case

if courses[i].courseNumber is the same as courseNumber

print out courses[i].courseNumber + " " + courses[i].courseTitle

print("Prerequisites:")

for each prerequisite in courses[i].prerequisites

print the prerequisite course number

break // Exit loop once found

}

// Hash Table search

void searchCourse(HashTable<Course> courses, String courseNumber) {

Course course = courses.get(courseNumber) // O(1) average, O(N) worst-case (collision)

if course is found

print out course.courseNumber + " " + course.courseTitle

print("Prerequisites:")

for each prerequisite in course.prerequisites

print the prerequisite course number

else

print("Course not found.")

}

// Binary Search Tree (BST) search

void searchCourse(Tree<Course> courses, String courseNumber) {

Course course = courses.find(courseNumber) // O(log N) average, O(N) worst-case (unbalanced tree)

if course is found

print out course.courseNumber + " " + course.courseTitle

print("Prerequisites:")

for each prerequisite in course.prerequisites

print the prerequisite course number

else

print("Course not found.")

}

**Print Alphanumerically Ordered List Pseudocode**

This function is specific to the data structure because of how data is accessed and sorted.

**Vector**

FUNCTION printSortedCourses(Vector<Course> courses)

// Create a copy to avoid sorting the original structure if it is needed unsorted

Vector<Course> sortedCourses = copy(courses)

// Sort the vector by courseNumber (alphanumeric order)

// Common algorithms: Merge Sort or Quick Sort

sort(sortedCourses, compareByCourseNumber) // O(N log N)

// Print the sorted list

FOR all courses in sortedCourses DO // O(N)

print(course.courseNumber + " " + course.courseTitle)

END FOR

END

**Hash Table**

FUNCTION printSortedCourses(HashTable<Course> courses)

// Hash Tables don't naturally maintain order. Extract keys (course numbers).

LIST<STRING> courseNumbers = getKeys(courses) // O(N) to iterate through all buckets

// Sort the list of keys (course numbers)

sort(courseNumbers, compareAlphanumeric) // O(N log N)

// Print the list using the sorted keys to look up the full course object

FOR each courseNum in courseNumbers DO // O(N) total iterations

Course course = courses.get(courseNum) // O(1) average lookup

print(course.courseNumber + " " + course.courseTitle)

END FOR

END

**Tree (Binary Search Tree - BST)**

FUNCTION printSortedCourses(Tree<Course> courses)

// BSTs maintain a sorted order based on key (courseNumber).

// In-Order Traversal prints the nodes in non-decreasing order.

inOrderTraversal(courses.root) // O(N)

END

// Helper function for In-Order Traversal

FUNCTION inOrderTraversal(Node node)

IF node IS NOT NULL THEN

inOrderTraversal(node.left)

print(node.course.courseNumber + " " + node.course.courseTitle)

inOrderTraversal(node.right)

END IF

END

**Runtime Analysis**

**File Loading and Object Creation**

The file loading and object creation includes; reading N lines from a file, parsing each line, and then inserting the course object into the data structure. The cost for opening/closing the file and checking for an empty file are constant time, O(1), so we focus on the loop over the N courses.

We can assume the following:

* N is the number of courses
* The cost of reading a line and basic parsing/error checking is constant, O(1), as the length of a single line (course info and prerequisites) bound by a constant, K, that doesn't grow with N.
* The cost of insertion into the data structure = primary difference.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Pseudocode Step (Within loop) | Data Structure | Line Cost | # Times Executes | Total Cost |
| STRING line = readLine(inputFile) | All | 1 | N | N |
| Course newCourse = parseLine(line) | All | 1 | N | N |
| **Insertion:** insertIntoVector(courses, newCourse) | **Vector** | 1 (amortized) | N | N |
| **Insertion:** insertIntoHashTable(courses, newCourse) | **Hash Table** | O(1) avg. (O(N) worst-case) | N | O(N) avg. (O(N2) worst-case) |
| **Insertion:** insertIntoTree(courses, newCourse) | **Tree (BST)** | O(logN) avg. (O(N) worst-case) | N | O(NlogN) avg. (O(N2) worst-case) |
| **Total Cost** | **Vector** | \multicolumn{3}{ | c | }{2N+N=3N} |
| **Total Cost** | **Hash Table** | \multicolumn{3}{ | c | }{2N+N×O(1)≈3N} |
| **Total Cost** | **Tree (BST)** | \multicolumn{3}{ | c | }{2N+N×O(logN)≈O(NlogN)} |
| **Worst-Case Runtime (Big O)** | **Vector** | \multicolumn{3}{ | c | }{**O(N)**} |
| **Worst-Case Runtime (Big O)** | **Hash Table** | \multicolumn{3}{ | c | }{**O(N2)** (due to collisions)} |
| **Worst-Case Runtime (Big O)** | **Tree (BST)** | \multicolumn{3}{ | c | }{**O(N2)** (due to unbalanced tree/skewed data)} |

**Data Structure Analysis**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Data Structure | Advantages | Disadvantages | Option 2: Print Sorted List (Runtime) | Option 3: Search/Print Course (Runtime) |
| **Vector** | Simple to implement.  Excellent for full list traversal.  Insertion is O(1) amortized.  Memory efficient. | Search is O(N) (must traverse list).  Printing sorted list requires explicit O(NlogN) sort every time. | O(NlogN) (for sorting) | O(N) (linear search) |
| **Hash Table** | **Fastest lookups** by key (O(1) average). Fastest file loading (O(N) average). | Poor performance for ordered data tasks.  Must extract keys, sort them O(NlogN), and then look up each course O(N) total.  High memory overhead due to key storage and array/bucket structure. | O(NlogN) (extract and sort keys) | O(1) average |
| **Tree (BST)** | Automatically maintains order, enabling O(N) traversal for Option 2.  Faster search than a vector (O(logN) average). | Slower insertion/file loading than Vector or Hash Table (O(NlogN) average). Memory overhead for pointer storage.  Search and Insertion can degrade to O(N) in the worst case (unbalanced tree). | O(N) (In-Order Traversal) | O(logN) average |

**Recommendation**

Based on the requirements and the Big O analysis, I recommend using the Binary Search Tree - BST data structure.

**Justification:**

1. **Option 2: Print Alphanumerically Ordered List:** This is a crucial requirement. The **BST** is the **only** structure that maintains the course data in alphanumeric order with an O(N) time complexity for printing the sorted list via an in-order traversal. This beating the O(NlogN) required by both the vector (sort time) and the hash table (extract and sort key time).
2. **Option 3: Search/Print Course:** The BST provides an average search time of O(logN), which is significantly faster than the vector's O(N) linear search. While the hash table is slightly faster at O(1) average, O(logN) is very fast.
3. **Overall Balance:** The BST offers the best balance between the two main requirements. It is excellent for ordered access (Option 2) and direct access (Option 3). Its average-case file loading time of O(NlogN) is acceptable, only slightly slower than the vector's O(N) and the hash table's O(N) average.

To mitigate the O(N) worst-case search and insertion time that a basic BST suffers from the implementation should use a Self-Balancing Binary SearchTree. A self-balancing tree guarantees that the search and insertion runtimes remain O(logN) even in the worst case, making the structure efficient for both key requirements.

**References:**

Shaffer, C. A. (2012). *Data structures and algorithm analysis in C++* (4th ed.). Dover Publications.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to algorithms* (4th ed.). MIT Press.