**ABCU Course Advising Program: Pseudocode and Runtime Analysis**

**1. Pseudocode for All Data Structures**

**1.1 Vector Data Structure**

text

// Main Program

BEGIN

coursesVector = empty vector of Course objects

WHILE true DO

DISPLAY menu options

GET userChoice

CASE userChoice OF

1: loadDataStructure()

2: printCourseList()

3: printCourse()

9: EXIT program

DEFAULT: DISPLAY "Invalid option"

END CASE

END WHILE

END

// Load Data Structure Function

FUNCTION loadDataStructure()

TRY

OPEN "ABCU\_Advising\_Program\_Input.txt" as file

FOR EACH line IN file DO

IF line is empty THEN CONTINUE

// Parse course data

tokens = SPLIT line by commas

IF tokens.length < 2 THEN

DISPLAY "Format error in line: " + line

CONTINUE

END IF

courseNumber = TRIM(tokens[0])

courseTitle = TRIM(tokens[1])

prerequisites = empty vector

// Parse prerequisites if they exist

FOR i FROM 2 TO tokens.length-1 DO

prereq = TRIM(tokens[i])

IF prereq is not empty THEN

ADD prereq TO prerequisites

END IF

END FOR

// Create and store course object

course = NEW Course(courseNumber, courseTitle, prerequisites)

ADD course TO coursesVector

END FOR

CLOSE file

DISPLAY "Data loaded successfully"

CATCH file error

DISPLAY "Error opening file"

END TRY

END FUNCTION

// Print Course List Function

FUNCTION printCourseList()

IF coursesVector is empty THEN

DISPLAY "No courses loaded. Please load data first."

RETURN

END IF

// Sort courses alphanumerically

sortedCourses = SORT coursesVector BY courseNumber

DISPLAY "Alphanumeric Course List:"

FOR EACH course IN sortedCourses DO

DISPLAY course.courseNumber + ", " + course.courseTitle

END FOR

END FUNCTION

// Print Individual Course Function

FUNCTION printCourse()

IF coursesVector is empty THEN

DISPLAY "No courses loaded. Please load data first."

RETURN

END IF

DISPLAY "Enter course number:"

GET targetCourse

found = false

FOR EACH course IN coursesVector DO

IF course.courseNumber = targetCourse THEN

DISPLAY course.courseNumber + ", " + course.courseTitle

IF course.prerequisites is not empty THEN

DISPLAY "Prerequisites: " + JOIN(course.prerequisites, ", ")

ELSE

DISPLAY "Prerequisites: None"

END IF

found = true

BREAK

END IF

END FOR

IF NOT found THEN

DISPLAY "Course not found"

END IF

END FUNCTION

**1.2 Hash Table Data Structure**

text

// Main Program

BEGIN

coursesHashTable = empty hash table

WHILE true DO

DISPLAY menu options

GET userChoice

CASE userChoice OF

1: loadDataStructure()

2: printCourseList()

3: printCourse()

9: EXIT program

DEFAULT: DISPLAY "Invalid option"

END CASE

END WHILE

END

// Load Data Structure Function

FUNCTION loadDataStructure()

TRY

OPEN "ABCU\_Advising\_Program\_Input.txt" as file

FOR EACH line IN file DO

IF line is empty THEN CONTINUE

// Parse course data

tokens = SPLIT line by commas

IF tokens.length < 2 THEN

DISPLAY "Format error in line: " + line

CONTINUE

END IF

courseNumber = TRIM(tokens[0])

courseTitle = TRIM(tokens[1])

prerequisites = empty vector

// Parse prerequisites if they exist

FOR i FROM 2 TO tokens.length-1 DO

prereq = TRIM(tokens[i])

IF prereq is not empty THEN

ADD prereq TO prerequisites

END IF

END FOR

// Create and store course object

course = NEW Course(courseNumber, courseTitle, prerequisites)

INSERT course INTO coursesHashTable WITH KEY courseNumber

END FOR

CLOSE file

DISPLAY "Data loaded successfully"

CATCH file error

DISPLAY "Error opening file"

END TRY

END FUNCTION

// Print Course List Function

FUNCTION printCourseList()

IF coursesHashTable is empty THEN

DISPLAY "No courses loaded. Please load data first."

RETURN

END IF

// Extract all courses and sort

allCourses = empty vector

FOR EACH key IN coursesHashTable DO

course = GET course FROM coursesHashTable USING key

ADD course TO allCourses

END FOR

sortedCourses = SORT allCourses BY courseNumber

DISPLAY "Alphanumeric Course List:"

FOR EACH course IN sortedCourses DO

DISPLAY course.courseNumber + ", " + course.courseTitle

END FOR

END FUNCTION

// Print Individual Course Function

FUNCTION printCourse()

IF coursesHashTable is empty THEN

DISPLAY "No courses loaded. Please load data first."

RETURN

END IF

DISPLAY "Enter course number:"

GET targetCourse

course = GET course FROM coursesHashTable USING targetCourse

IF course exists THEN

DISPLAY course.courseNumber + ", " + course.courseTitle

IF course.prerequisites is not empty THEN

DISPLAY "Prerequisites: " + JOIN(course.prerequisites, ", ")

ELSE

DISPLAY "Prerequisites: None"

END IF

ELSE

DISPLAY "Course not found"

END IF

END FUNCTION

**1.3 Binary Search Tree Data Structure**

text

// Main Program

BEGIN

coursesBST = empty binary search tree

WHILE true DO

DISPLAY menu options

GET userChoice

CASE userChoice OF

1: loadDataStructure()

2: printCourseList()

3: printCourse()

9: EXIT program

DEFAULT: DISPLAY "Invalid option"

END CASE

END WHILE

END

// Load Data Structure Function

FUNCTION loadDataStructure()

TRY

OPEN "ABCU\_Advising\_Program\_Input.txt" as file

FOR EACH line IN file DO

IF line is empty THEN CONTINUE

// Parse course data

tokens = SPLIT line by commas

IF tokens.length < 2 THEN

DISPLAY "Format error in line: " + line

CONTINUE

END IF

courseNumber = TRIM(tokens[0])

courseTitle = TRIM(tokens[1])

prerequisites = empty vector

// Parse prerequisites if they exist

FOR i FROM 2 TO tokens.length-1 DO

prereq = TRIM(tokens[i])

IF prereq is not empty THEN

ADD prereq TO prerequisites

END IF

END FOR

// Create and store course object

course = NEW Course(courseNumber, courseTitle, prerequisites)

INSERT course INTO coursesBST USING courseNumber as key

END FOR

CLOSE file

DISPLAY "Data loaded successfully"

CATCH file error

DISPLAY "Error opening file"

END TRY

END FUNCTION

// Print Course List Function

FUNCTION printCourseList()

IF coursesBST is empty THEN

DISPLAY "No courses loaded. Please load data first."

RETURN

END IF

DISPLAY "Alphanumeric Course List:"

PERFORM inOrderTraversal(coursesBST.root)

END FUNCTION

// In-order traversal helper function

FUNCTION inOrderTraversal(node)

IF node is not null THEN

inOrderTraversal(node.left)

DISPLAY node.course.courseNumber + ", " + node.course.courseTitle

inOrderTraversal(node.right)

END IF

END FUNCTION

// Print Individual Course Function

FUNCTION printCourse()

IF coursesBST is empty THEN

DISPLAY "No courses loaded. Please load data first."

RETURN

END IF

DISPLAY "Enter course number:"

GET targetCourse

course = SEARCH coursesBST FOR targetCourse

IF course exists THEN

DISPLAY course.courseNumber + ", " + course.courseTitle

IF course.prerequisites is not empty THEN

DISPLAY "Prerequisites: " + JOIN(course.prerequisites, ", ")

ELSE

DISPLAY "Prerequisites: None"

END IF

ELSE

DISPLAY "Course not found"

END IF

END FUNCTION

**2. Runtime Analysis**

**2.1 Big O Analysis for File Reading and Course Object Creation**

| Operation | Vector | Hash Table | Binary Search Tree |
| --- | --- | --- | --- |
| Open file | O(1) | O(1) | O(1) |
| Read each line | O(n) | O(n) | O(n) |
| Parse line | O(m) per line | O(m) per line | O(m) per line |
| Create course object | O(1) per course | O(1) per course | O(1) per course |
| Insert into data structure | O(1) amortized | O(1) average | O(log n) average |
| **Total Time Complexity** | **O(n)** | **O(n)** | **O(n log n)** |

**Note:** n = number of courses, m = number of tokens per line

**2.2 Detailed Cost Analysis (n courses)**

**Vector Implementation:**

* File operations: O(n)
* Parsing each line: O(n × m)
* Creating course objects: O(n)
* Inserting into vector: O(n) total (O(1) per insertion amortized)
* **Overall: O(n)**

**Hash Table Implementation:**

* File operations: O(n)
* Parsing each line: O(n × m)
* Creating course objects: O(n)
* Inserting into hash table: O(n) total (O(1) per insertion average)
* **Overall: O(n)**

**Binary Search Tree Implementation:**

* File operations: O(n)
* Parsing each line: O(n × m)
* Creating course objects: O(n)
* Inserting into BST: O(n log n) total (O(log n) per insertion)
* **Overall: O(n log n)**

**3. Data Structure Analysis**

**3.1 Advantages and Disadvantages**

**Vector:**

* *Advantages:* Simple implementation, predictable memory usage, fast iteration
* *Disadvantages:* Slow search (O(n)), requires sorting for ordered output, inefficient for large datasets

**Hash Table:**

* *Advantages:* Fast search (O(1) average), fast insertion, efficient for course lookup
* *Disadvantages:* No inherent ordering, requires extra memory for buckets, potential collisions

**Binary Search Tree:**

* *Advantages:* Natural ordering, efficient search (O(log n) average), no sorting required for output
* *Disadvantages:* More complex implementation, worst-case O(n) if unbalanced, slower insertion than hash table

**3.2 Recommendation**

**Recommended Data Structure: Binary Search Tree**

**Justification:**

Based on the Big O analysis and the specific requirements of the ABCU advising program, the Binary Search Tree provides the optimal balance of efficiency and functionality:

1. **Natural Ordering:** The BST inherently maintains courses in sorted order, making Option 2 (printing alphanumeric list) extremely efficient with O(n) in-order traversal without requiring additional sorting.
2. **Efficient Search:** Course lookup (Option 3) operates in O(log n) time on average, which is significantly faster than the Vector's O(n) linear search.
3. **Memory Efficiency:** While the Hash Table offers O(1) average search time, the BST provides better worst-case performance guarantees and doesn't require the overhead of hash functions and collision resolution.
4. **Scalability:** For the expected number of computer science courses (likely dozens to low hundreds), the O(n log n) insertion time during loading is acceptable, especially since loading is a one-time operation per session.
5. **Requirements Fit:** The BST directly supports both primary requirements - ordered listing and efficient individual course lookup - without compromising either function.

The Binary Search Tree provides the best overall performance characteristics for this specific application while maintaining clean, maintainable code structure.