ABCU Course Management System Psuedocode

Lawrence Chen

Data Structure Definition

```
STRUCT Course
    courseNumber AS STRING
    courseTitle AS STRING
    prerequisites AS VECTOR OF STRING
END STRUCT
```

Global Variables

```
courses AS VECTOR OF course
```

File Loading and Validation

```
FUNCTION loadCourseData(filename)
    DECLARE file AS FILE
    DECLARE line AS STRING
    DECLARE tokens AS VECTOR OF STRING
    DECLARE tempCourse AS Course
    DECLARE isValid AS BOOLEAN
    // Clear existing course data
    courses.clear()
    // Attempt to open file
    TRY
        file = OPEN(filename, READ_MODE)
    CATCH FileNotFoundException
        PRINT "Error: Could not open file " + filename
        RETURN FALSE
    END TRY
    // Read and validate each line
    WHILE NOT file.eof() line = file.readLine()
        // Skip empty lines
        IF line.isEmpty() THEN
            CONTINUE
        END IF
        // Parse line into tokens (comma-separated)
        tokens = parseLine(line)
        // Validate minimum requirements
        IF tokens.size() < 2 THEN
```

```
            PRINT "Error: Invalid format on line: " + line
            PRINT "Each line must have at least course number and title"
            file.close()
            RETURN FALSE
        END IF
        // Create temporary course object
        tempCourse.courseNumber = tokens[0].trim()
        tempCourse.courseTitle = tokens[1].trim()
        tempCourse.prerequisites.clear()
        // Add prerequisites if they exist
        FOR i = 2 TO tokens.size() - 1
            tempCourse.prerequisites.add(tokens[i].trim())
        END FOR
        // Add course to vector courses.add(tempCourse)
    END WHILE
    file.close()
    // Validate prerequisites exist as courses
    isValid = validatePrerequisites()
    IF isValid THEN
        PRINT "Course data loaded successfully!"
        PRINT "Total courses loaded: " + courses.size()
        RETURN TRUE
    ELSE courses.clear()
        RETURN FALSE
    END IF
END FUNCTION
```

Code	Line Cost	# Times Executes	Total Cost
DECLARE file AS FILE DECLARE line AS STRING DECLARE tokens AS VECTOR OF STRING ... END TRY	1	1	1
// Read and validate each line WHILE NOT file.eof() line = file.readLine() ... END WHILE	1	n	n

file.close()
...
END FUNCTION
	1	1	1

		Total Cost	n+2
		Runtime	O(n)

Helper Function: Parse Line

```
FUNCTION parseLine(line)
  DECLARE tokens AS VECTOR OF STRING
  DECLARE currentToken AS STRING
  DECLARE i AS INTEGER
  currentToken = "”
  FOR i = 0 TO line.length() - 1
    IF line[i] = ',' THEN
      tokens.add(currentToken)
      currentToken = ""
    ELSE
      currentToken = currentToken + line[i]
    END IF
  END FOR
  // Add the last token
  IF NOT currentToken.isEmpty() THEN
    tokens.add(currentToken)
  END IF
  RETURN tokens
```

```
END FUNCTION
```

Helper Function: Validate Prerequisites

```
FUNCTION validatePrerequisites()
  DECLARE i, j AS INTEGER
  DECLARE prerequisiteFound AS BOOLEAN
  // Check each course's prerequisites
  FOR i = 0 TO courses.size() - 1
    // Check each prerequisite for current course
    FOR j = 0 TO courses[i].prerequisites.size() - 1
      prerequisiteFound = FALSE
      // Search for prerequisite in course list
      FOR k = 0 TO courses.size() - 1
        IF courses[k].courseNumber = courses[i].prerequisites[j] THEN
          prerequisiteFound = TRUE
          BREAK
        END IF
      END FOR
      // If prerequisite not found, report error
      IF NOT prerequisiteFound THEN
        PRINT "Error: Prerequisite " + courses[i].prerequisites[j] +
           " for course " + courses[i].courseNumber + " not found in course list"
        RETURN FALSE
      END IF
    END FOR
  END FOR
  RETURN TRUE
END FUNCTION
```

Course Object Creation and Storage

```
FUNCTION createCourseObject(courseNum, courseTitle, prereqList)
   DECLARE newCourse AS Course

   newCourse.courseNumber = courseNum
   newCourse.courseTitle = courseTitle
   newCourse.prerequisites = prereqList

   RETURN newCourse
END FUNCTION

FUNCTION storeCourseInVector(course)
   courses.add(course)
   PRINT "Course " + course.courseNumber + " stored successfully"
END FUNCTION
```

Code	Line Cost	# Times Executes	Total Cost

FUNCTION storeCourseInVector(course)
   courses.add(course)
   PRINT "Course " + course.courseNumber + " stored successfully"
END FUNCTION
	1	1	1

Total Cost			1
Runtime			O(1)

Search and Print Course Information

```
FUNCTION searchAndPrintCourse()
   DECLARE searchCourseNum AS STRING
   DECLARE courseFound AS BOOLEAN
   DECLARE i AS INTEGER
   IF courses.size() = 0 THEN
      PRINT "No course data loaded. Please load course data first."
      RETURN
   END IF

   PRINT "Enter course number to search: "
   INPUT searchCourseNum

   courseFound = FALSE

   // Search for course in vector
   FOR i = 0 TO courses.size() - 1
      IF courses[i].courseNumber = searchCourseNum THEN
         courseFound = TRUE
         printCourseDetails(courses[i])
         BREAK
      END IF
   END FOR

   IF NOT courseFound THEN
      PRINT "Course " + searchCourseNum + " not found."
   END IF
END FUNCTION

FUNCTION printCourseDetails(course)
   DECLARE i AS INTEGER

   PRINT "Course Number: " + course.courseNumber
   PRINT "Course Title: " + course.courseTitle

   IF course.prerequisites.size() = 0 THEN
      PRINT "Prerequisites: None"
   ELSE
      PRINT "Prerequisites: "
      FOR i = 0 TO course.prerequisites.size() - 1
         IF i > 0 THEN
            PRINT ", "
      END IF
      PRINT course.prerequisites[i]
    END FOR
    PRINT "" // New line
  END IF
  PRINT "------------------------"
END FUNCTION
```

Sort and Print Vector with selection sort

```
FUNCTION sortCoursesSelectionSort()
  DECLARE i, j, minIndex AS INTEGER
  DECLARE temp AS Course

  // Check if courses vector is empty
  IF courses.size() = 0 THEN
    RETURN
  END IF

  // Sort courses alphanumerically by course number using selection sort
  FOR i = 0 TO courses.size() - 2
    minIndex = i
    FOR j = i + 1 TO courses.size() - 1
      IF courses[j].courseNumber < courses[minIndex].courseNumber THEN
        minIndex = j
      END IF
    END FOR

    // Swap if needed
    IF minIndex != i THEN
      temp = courses[i]
      courses[i] = courses[minIndex]
      courses[minIndex] = temp
    END IF
  END FOR
END FUNCTION

  // Display all courses
  FOR i = 0 TO courses.size() - 1
    printCourseDetails(courses[i])
  END FOR

  PRINT "Total courses displayed: " + courses.size()
END FUNCTION
```

Course Hash Table

Data Structures and Variables

```
STRUCT Course
courseNumber AS STRING
courseTitle AS STRING
        prerequisites : LIST of STRING
END STRUCT

DECLARE hashTable : HASH_TABLE of Course objects
DECLARE courseList : LIST of STRING (for validation)
DECLARE fileName : STRING
DECLARE inputFile : FILE
DECLARE line : STRING
DECLARE isValid : BOOLEAN = TRUE
```

File Loading and Data Parsing

Main Program Flow

```
BEGIN LoadCourseData
        PRINT “Enter filename: “
        INPUT fileName

        // Step 1: Open and validate file
        IF OpenFile(fileName) = FALSE THEN
                PRINT “Error: Could not open file “ + fileName
                RETURN FALSE
        END IF

// Step 2: First pass – collect all course numbers for validation
        CollectCourseNumbers()
```

```
    // Step 3: Second pass – parse and validate course data
    IF ParseAndValidateCourses() = TRUE THEN
        PRINT "Course data loaded successfully!"
        RETURN TRUE
    ELSE
PRINT "Error: File validation failed"
        RETURN FALSE
    END IF
END LoadCourseData
```

Code	Line Cost	# Times Executes	Total Cost

BEGIN LoadCourseData
…
END IF
	1	1	1

CollectCourseNumbers()	n	1	n
IF ParseAndValidateCourses() = TRUE THEN	n	1	n
PRINT "Course data loaded successfully!" … END LoadCourseData	1	1	1
Total Cost			2n+2
Runtime			O(n)

File Opening Function

```
FUNCTION OpenFile(fileName)
    TRY
        SET inputFile = OPEN(fileName, READ_MODE)
        IF inputFile is NULL THEN
            RETURN FALSE
        END IF
```

```
RETURN TRUE
CATCH FileException
        RETURN FALSE
    END TRY
END FUNCTION
```

First Pass – Collect Course Numbers

```
FUNCTION CollectCourseNumbers()
    WHILE NOT EndOfFile(inputFile) DO
        READ line from inputFile

        // Skip empty lines
        IF line is empty THEN
            CONTINUE
        END IF

        // Parse the line to get course number (first token)
        Tokens = SplitSTring(line, “,”)
        IF tokens.size >= 1 THEN
            courseNumber = Trim(tokens[0])
            ADD courseNumber to courseList
        END IF
    END WHILE

    // Reset file pointer to beginning for second pass
    ResetFile(inputFile)
END FUNCTION
```

Second Pass – Parse and Validate Courses

```
FUNCTION ParseAndValidateCourses()
    DECLARE lineNumber: INTEGER = 0

    WHILE NOT EndOfFile(inputFile) DO
        INCREMENT lineNumber
        READ line from inputFile
```

```
// Skip empty lines
        IF line is empty THEN
            CONTINUE
        END IF

        // Parse the current line
        IF ParseCourseLine(line, lineNumber) = FALSE THEN
            RETURN FALSE
        END IF
    END WHILE

    CLOSE inputFile
    RETURN TRUE
END FUNCTION
```

Code	Line Cost	# Times Executes	Total Cost
FUNCTION OpenFile()	1	1	1
FUNCTION CollectCourseNumbers()	n	1	n
FUNCTION ParseAndValidateCourses()	n	1	n
		Total Cost	2n+1
		Runtime	O(n)

Parse Individual Course Line

```
FUNCTION ParseCourseLine(line, lineNumber)
    // Split line by comma delimiter
    tokens = SplitString(line, “,”)

    // Validation Rule 1: At least two parameters (course number and title)
    IF tokens.size < 2 THEN
        PRINT “Error on line ” + lineNumber + “: Missing course number or title”
        RETURN FALSE
    END IF

    // Create new course object
    DECLARE newCourse : Course
    newCourse.courseNumber = Trim(tokens[0])
    newCourse.courseTitle = Trim(tokens[1])

    // Validate course number format (not empty)
    IF newCourse.courseNumber is empty THEN
        PRINT “Error on line ” + lineNumber + “: Course number cannot be empty”
            RETURN FALSE
    END IF

    // Validate course title (not empty)
    IF newCourse.courseTitle is empty THEN
        PRINT “Error on line ” + lineNumber + “: Course title cannot be empty”
        RETURN FALSE
    END IF

    // Process prerequisites (tokens from index 2 onwards)
    FOR i = 2 to tokens.size - 1 DO
        prerequisite = Trim(tokens[i])

        // Skip empty prerequisites
        IF prerequisite is NOT empty THEN
            // Validation Rule 2 : Prerequisite must exist as a course
            IF NOT CourseExists(prerequisite, courseList) THEN
                PRINT “Error on line ” + lineNumber + “: Prerequisite ‘” +
            prerequisite + “’ does not exist as a course”
                RETURN FALSE
            END IF

        ADD prerequisite to newCourse.prerequisites
        END IF
    END FOR

// Store course in hash table
    hashTable.Insert(newCourse.courseNumber, newCourse)
    RETURN TRUE
END FUNCTION
```

Prerequisite Validation Helper

```
FUNCTION CourseExists(courseNumber, courseList)
    FOR each course in courseList DO
        IF course equals courseNumber THEN
RETURN TRUE
END IF
END FOR
    RETURN FALSE
END FUNCTION
```

Hash Table Implementation

```
// Hash function for course numbers
FUNCTION HashFunction(courseNumber)
    hash = 0
    FOR each character c in courseNumber DO
        hash = (hash * 31 + ASCII_VALUE(c)) MOD TABLE_SIZE
    END FOR
    RETURN hash
END FUNCTION

//Insert course into hash table
FUNCTION InsertCourse(courseNumber, courseObject)
    Index = HashFunction(courseNumber)

    // Handle collisions using chaining
    IF hashTable[index] is empty THEN
        hashTable[index] = courseObject
    ELSE
        // Chain collision resolution
        ADD courseObject to hashTable[index] chain
    END IF
END FUNCTION
```

Code	Line Cost	# Times Executes	Total Cost
FUNCTION HashFunction(courseNumber) hash = 0	1	1	1

FOR each character c in courseNumber DO
hash = (hash * 31 + ASCII_VALUE(c)) MOD TABLE_SIZE
END FOR
	1	n	n

RETURN hash
END FUNCTION
	1	1	1

FUNCTION InsertCourse(courseNumber, courseObject) … END FUNCTION	1	1	1
		Total Cost	n+3
		Runtime	O(n)

```
// Retrieve course from hash table
FUNCTION GetCourse(courseNumber)
Index = HashFunction(courseNumber)

// Search in the chain at this index
    current = hashTable[index]
    WHILE current is NOT NULL DO
        IF current.courseNumber equals courseNumber THEN
RETURN current
```

```
END IF
current = current.next
END WHILE

RETURN NULL // Course not found
END FUNCTION
```

Print All Courses

```
FUNCTION PrintAllCourses()
    // Create a list to store all courses for sorting
    DECLARE courseNumbers : LIST of STRING


    // Extract all course numbers from hash table
    FOR i = 0 to TABLE_SIZE – 1 DO
        IF hashTable[i] is NOT empty THEN
            current = hashTable[i]
            WHILE current is NOT NULL DO
                ADD current.courseNumber to courseNumbers
                current = current.next
            END WHILE

        END IF

    END FOR

// Sort course number alphanumerically
    Sort(courseNumbers)

    PRINT “\nCourse List (Alphanumeric Order):”
    PRINT “\n============================”

    // Print each course in sorted order
    FOR each courseNumber in courseNumbers DO
        course = GetCourse(courseNumber)
        IF course is NOT NULL THEN
            PRINT courseNumber + “, ” + course.courseTitle
        END IF
    END FOR
    PRINT “”
END FUNCTION
```

Print Specific Course Information

```
FUNCTION PrintSpecificCourse()
    DECLARE searchCourseNumber : STRING

PRINT “Enter course number: ”
    INPUT searchCourseNumber

    // Convert to uppercase for consistent searching
    searchCourseNumber = ToUpperCase(Trim(searchCourseNumber))

//Retrieve course from hash table
    course = GetCourse(searchCourseNumber)

    IF course is NULL THEN
        PRINT “Course ” + searchCourseNumber + “ not found.”
        RETURN
    END IF

    // Print course information
    PRINT “\nCourse Information”
    PRINT “\n================”
    PRINT “Course Number: ” + course.courseNumber
    PRINT “Course Title: ” + course.courseTitle

    // Print prerequisites
    IF course.prerequisites.size = 0 THEN
        PRINT “Prerequisites: None”
    ELSE
        PRINT “Prerequisites: ”
        FOR i = 0 to course.prerequisites.size -1 DO
            PRINT “ - ” + course.prerequisites[i]
        END FOR
    END IF
    PRINT “”
END FUNCTION



FUNCTION SortAndPrintAllCourses()
    DECLARE courseNumbers : LIST of STRING
    DECLARE course : Course
    DECLARE i, j, minIndex AS INTEGER
    DECLARE temp AS STRING

    // Check if hash table has any data
    IF hashTable is empty THEN
        PRINT "No course data loaded. Hash table is empty."
        RETURN
    END IF

    // Extract all course numbers from hash table
    FOR i = 0 TO TABLE_SIZE - 1 DO
        IF hashTable[i] is NOT empty THEN
            current = hashTable[i]
            WHILE current is NOT NULL DO
                ADD current.courseNumber TO courseNumbers
                current = current.next
            END WHILE
        END IF
    END FOR

     // Check if any courses were found
    IF courseNumbers.size = 0 THEN
        PRINT "No courses found in hash table."
        RETURN
    END IF

    // Sort course numbers alphanumerically using selection sort
    FOR i = 0 TO courseNumbers.size - 2 DO
        minIndex = i
        FOR j = i + 1 TO courseNumbers.size - 1 DO
            IF courseNumbers[j] < courseNumbers[minIndex] THEN
                minIndex = j
            END IF
        END FOR


        // Swap if needed
        IF minIndex != i THEN
            temp = courseNumbers[i]
            courseNumbers[i] = courseNumbers[minIndex]
            courseNumbers[minIndex] = temp
        END IF
    END FOR

    PRINT ""
    PRINT "All Courses:"
    PRINT "====================================="
    PRINT ""

    // Print each course in sorted order with full details
    FOR i = 0 TO courseNumbers.size - 1 DO
        course = GetCourse(courseNumbers[i])
        IF course is NOT NULL THEN
            PrintCourseDetails(course)
        END IF
    END FOR

    PRINT "Total courses displayed: " + courseNumbers.size
    PRINT ""
END FUNCTION

// Helper function to print detailed course information
FUNCTION PrintCourseDetails(course)
    DECLARE i AS INTEGER
    PRINT "Course Number: " + course.courseNumber
    PRINT "Course Title: " + course.courseTitle

    // Print prerequisites
    IF course.prerequisites.size = 0 THEN
        PRINT "Prerequisites: None"
    ELSE
        PRINT "Prerequisites: "
        FOR i = 0 TO course.prerequisites.size - 1 DO
            IF i > 0 THEN
                PRINT ", "
            END IF
            PRINT course.prerequisites[i]
        END FOR
        PRINT "" // New line after prerequisites
    END IF
    PRINT "------------------------"
END FUNCTION
```

Data Structure Definitions

```
STRUCT Course
    courseNumber: STRING
    name: STRING
    prerequisites: VECTOR<STRING>
END STRUCT

STRUCT TreeNode
    course: Course
    left: POINTER to TreeNode
    right: POINTER to TreeNode
END STRUCT

CLASS Binary Search Tree
    root: Pointer to TreeNode

    FUNCTION insert(course: Course)
    FUNCTION search(courseNumber: STRING): Course
    FUNCTION inOrderTraversal(): VOID
```

```
END CLASS

FUNCTION readAndValidateCourseFile(filename: STRING): BinarySearchTree
  // Initialize variables
  SET courseTree = new BinarySearchTree()
  SET allCourseNumbers = new SET<STRING>
  SET allPrerequisites = new SET<STRING>
  SET fileLines = new VECTOR<STRING>

  // Step 1: Open and read file
  TRY
    OPEN file with filename for reading
    IF file cannot be opened THEN
      PRINT "Error: Cannot open file " + filename
      RETURN empty courseTree
    END IF

    // Read all lines into memory for two-pass validation
    WHILE not end of file
      READ line from file
      IF line is not empty THEN
        ADD line to fileLines
      END IF
    END WHILE

    CLOSE file

  CATCH file exception
    PRINT "Error reading file: " + exception message
    RETURN empty courseTree
  END CATCH

  // Step 2: First pass - Basic format validation and collect course numbers
  FOR each line in fileLines
    SET tokens = SPLIT line by comma

    // Validate minimum format requirements
    IF tokens.size() < 2 THEN
      PRINT "Error: Line has insufficient data (less than 2 fields): " + line
      RETURN empty courseTree
    END IF

    // Trim whitespace from all tokens
    FOR each token in tokens
      TRIM whitespace from token
    END FOR

    // Validate course number and name are not empty
    IF tokens[0] is empty OR tokens[1] is empty THEN
      PRINT "Error: Course number or name is empty: " + line
      RETURN empty courseTree
    END IF

    // Store course number for prerequisite validation
    ADD tokens[0] to allCourseNumbers

    // Store prerequisites for validation
    FOR i = 2 to tokens.size() - 1
      IF tokens[i] is not empty THEN
        ADD tokens[i] to allPrerequisites
      END IF
    END FOR
  END FOR

  // Step 3: Validate that all prerequisites exist as courses
  FOR each prerequisite in allPrerequisites
    IF prerequisite is not in allCourseNumbers THEN
      PRINT "Error: Prerequisite '" + prerequisite + "' does not exist as a course"
      RETURN empty courseTree
    END IF
  END FOR

  // Step 4: Second pass - Create course objects and populate tree
  FOR each line in fileLines
    SET tokens = SPLIT line by comma
```

```
    // Trim whitespace from all tokens
    FOR each token in tokens
      TRIM whitespace from token
    END FOR

    // Create course object
    SET newCourse = new Course
    SET newCourse.courseNumber = tokens[0]
    SET newCourse.name = tokens[1]

    // Add prerequisites (if any)
    FOR i = 2 to tokens.size() - 1
      IF tokens[i] is not empty THEN
        ADD tokens[i] to newCourse.prerequisites
      END IF
    END FOR

    // Insert course into binary search tree
    courseTree.insert(newCourse)
  END FOR

  PRINT "File loaded successfully. " + allCourseNumbers.size() + " courses processed."
  RETURN courseTree

END FUNCTION
```

Code	Line Cost	# Times Executes	Total Cost
// Initialize variables … END CATCH	1	1	1

FOR each line in fileLines
SET tokens = SPLIT line by comma
…
END FOR
	n	1	n

FOR each line in fileLines
SET tokens = SPLIT line by comma
…
END FOR
	n	1	n

// Create course object … END FUNCTION	1	1	1
		Total Cost	2n+2
		Runtime	O(n)

```
FUNCTION BinarySearchTree.insert(course: Course): VOID
  IF root is NULL THEN
    SET root = new TreeNode
    SET root.course = course
    SET root.left = NULL
    SET root.right = NULL
  ELSE
    CALL insertRecursive(root, course)
  END IF
END FUNCTION
```

Code	Line Cost	# Times Executes	Total Cost

FUNCTION BinarySearchTree.insert(course: Course): ... ELSE	1	1	1
CALL insertRecursive(root, course)	log(n)	1	log(n)
		Total Cost	log(n)+1
		Runtime	O(log(n))

```
FUNCTION insertRecursive(node: POINTER to TreeNode, course: Course): VOID
  IF course.courseNumber < node.course.courseNumber THEN
    IF node.left is NULL THEN
      SET node.left = new TreeNode
      SET node.left.course = course
      SET node.left.left = NULL
      SET node.left.right = NULL
    ELSE
      CALL insertRecursive(node.left, course)
    END IF
  ELSE IF course.courseNumber > node.course.courseNumber THEN
    IF node.right is NULL THEN
      SET node.right = new TreeNode
      SET node.right.course = course
      SET node.right.left = NULL
      SET node.right.right = NULL
    ELSE
      CALL insertRecursive(node.right, course)
    END IF
  ELSE
    // Duplicate course number found
    PRINT "Warning: Duplicate course number " + course.courseNumber + " ignored"
  END IF
END FUNCTION

FUNCTION BinarySearchTree.search(courseNumber: STRING): Course
  RETURN searchRecursive(root, courseNumber)
END FUNCTION
```

```
FUNCTION searchRecursive(node: POINTER to TreeNode, courseNumber: STRING): Course
  IF node is NULL THEN
    RETURN NULL Course (course not found)
  END IF

  IF courseNumber = node.course.courseNumber THEN
    RETURN node.course
  ELSE IF courseNumber < node.course.courseNumber THEN
    RETURN searchRecursive(node.left, courseNumber)
  ELSE
    RETURN searchRecursive(node.right, courseNumber)
  END IF
END FUNCTION

// Print all courses in alphanumeric order
FUNCTION BinarySearchTree.printAllCourses(): VOID
  PRINT "Here is a sample schedule:"
  PRINT ""
  CALL inOrderTraversal(root)
END FUNCTION


FUNCTION inOrderTraversal(node: POINTER to TreeNode): VOID
  IF node is not NULL THEN
    CALL inOrderTraversal(node.left)

    // Print course information
    PRINT node.course.courseNumber + ", " + node.course.name

    CALL inOrderTraversal(node.right)
  END IF
END FUNCTION

// Print specific course with prerequisites
FUNCTION BinarySearchTree.printCourse(courseNumber: STRING): VOID
  SET course = search(courseNumber)
  IF course is NULL THEN
    PRINT "Course " + courseNumber + " not found."
    RETURN
  END IF

  // Print course number and name
  PRINT course.courseNumber + ", " + course.name

  // Print prerequisites if any exist
  IF course.prerequisites.size() > 0 THEN
    PRINT "Prerequisites: "
    FOR i = 0 to course.prerequisites.size() - 1
      PRINT course.prerequisites[i]
      IF i < course.prerequisites.size() - 1 THEN
        PRINT ", "
      END IF
    END FOR
    PRINT "" // New line
  ELSE
    PRINT "Prerequisites: None"
  END IF
END FUNCTION

// Print courses with detailed prerequisite information
FUNCTION BinarySearchTree.printCoursesWithDetails(): VOID
  PRINT "Course Details:"
  PRINT "==============="
  CALL inOrderTraversalWithDetails(root)
END FUNCTION

FUNCTION inOrderTraversalWithDetails(node: POINTER to TreeNode): VOID
  IF node is not NULL THEN
    CALL inOrderTraversalWithDetails(node.left)

    // Print detailed course information
    PRINT node.course.courseNumber + ", " + node.course.name
    IF node.course.prerequisites.size() > 0 THEN
```

```
        PRINT "Prerequisites: "
        FOR i = 0 to node.course.prerequisites.size() - 1
          PRINT "  - " + node.course.prerequisites[i]
        END FOR
      ELSE
        PRINT "Prerequisites: None"
      END IF
      PRINT "" // Blank line for separation

      CALL inOrderTraversalWithDetails(node.right)
    END IF
END FUNCTION

FUNCTION displayMenu():
        WHILE TRUE DO
                PRINT “Welcome to the course planner.\n”
                PRINT “1. Load Data Structure\n”
                PRINT “2. Print Course List\n”
                PRINT “3. Print Course\n”
                PRINT “9. Exit\n”
                PRINT “What would you like to do? ”
                INPUT userChoice

                SWITCH userChoice:
                        CASE 1:
                                CALL loadCourseData()
                        CASE 2:
                                CALL printSortedCourses()
                        CASE 3:
                                CALL printCourseInfo()
                        CASE 9:
                                PRINT “Thank you for using the Course Planner!”
                                RETURN
                        DEFAULT:
                                PRINT “Invalid option. Please try again.”
                END SWITCH
        END WHILE
END FUNCTION
```

Advantages and Disadvantages of Vectors

The first data structure defined in psuedocode here is the vector. As defined here, the vector reads the data file in O(n) time and adds the course objects to the vector in O(1) time. Since objects are always added to the end of the vector, no elements are shifted. The vector class loads the data from the file in O(n) time, where n is the number of lines in the file. One disadvantage of vectors is that after being sorted, the worst-case scenario to find an element in the sorted vector is O(n) time, where the target element is at the opposite end of the vector from the start of iteration (i.e. target is last while start is first, or vice versa).

Advantages and Disadvantages of Hash Tables

The second data structure defined in psuedocode here is the hash table. As defined here, the hash table reads the data file in O(n) time and adds the course objects to the hash table in O(1) time. One advantage of the hash table is that element access is in O(1) time, due to the key-value pair system. One disadvantage of the hash table is that it does not inherently order elements, so items must be accessed one at a time and inserted into a secondary data structure and sorted there before being printed in alphanumeric order.

Advantages and Disadvantages of Binary Search Trees

The final data structure defined in pseudocode here is the binary search tree. As defined here, the binary table reads the data file in O(n) time and adds the course objects to the binary search tree in O(log(n)) time. One advantage of the binary search tree is that if the tree is balanced, it is already sorted and printing the elements in alphanumeric order is trivial. However, the corresponding disadvantage is that it is possible for the data to be sorted before being inserted into the binary search tree, resulting in elements with only one child each until the final element, and requiring O(n) time to access.

Based on my analyses of the data structures, I would recommend the binary search tree. One caveat I would add to the final implementation to maximize its efficacy would be to first use a vector and sort the data such that a balanced tree can be built from it. Once the balanced tree is built, it outperforms vectors and hash tables in other operations relating to the data thanks to the O(log(n)) time for those operations.