CS 300 Project One

**Course Management Pseudocode Document**

**1. Pseudocode for Each Data Structure**

**Vector Implementation**

**1. Load Data from File and Parse Each Line**

BEGIN

OPEN the file "courses.txt" for reading

IF the file cannot be opened THEN

PRINT "Error opening file."

EXIT program

END IF

CREATE an empty vector called courseVector

FOR each line in the file DO

TRIM whitespace from the line

SPLIT the line into tokens by comma (",")

IF there are less than 2 tokens THEN

PRINT "Error: Line does not contain at least two parameters."

CONTINUE to next line

END IF

SET courseNumber = token[0]

SET courseTitle = token[1]

SET prerequisites = empty list

IF there are more than 2 tokens THEN

FOR each token from token[2] to the end of the line DO

ADD token to prerequisites list

END FOR

END IF

IF there are prerequisites THEN

FOR each prerequisite in prerequisites DO

IF prerequisite does not exist as a course number in the courseVector THEN

PRINT "Error: Prerequisite course does not exist: " + prerequisite

CONTINUE to next line

END IF

END FOR

END IF

CREATE a new Course object with courseNumber, courseTitle, and prerequisites

ADD the new Course object to courseVector

END FOR

CLOSE the file

END

**2. Create Course Objects and Store Them in the Vector**

STRUCT Course

courseNumber (string)

courseTitle (string)

prerequisites (list of strings)

BEGIN

CREATE a function "CreateCourseObject" to process a single line:

SET courseNumber = token[0]

SET courseTitle = token[1]

SET prerequisites = empty list

IF there are more than 2 tokens THEN

FOR each token from token[2] to the end of the line DO

ADD token to prerequisites list

END FOR

END IF

RETURN a new Course object with courseNumber, courseTitle, and prerequisites

END

BEGIN

CREATE a loop to process each line in the file and add Course objects to courseVector:

FOR each line in the file DO

CALL CreateCourseObject(line)

ADD the returned Course object to courseVector

END FOR

END

**3. Search and Print Course Information**

BEGIN

CREATE a function "SearchCourse" that takes courseNumber as an argument:

FOR each Course object in courseVector DO

IF Course.courseNumber is equal to courseNumber THEN

PRINT "Course Number: " + Course.courseNumber

PRINT "Course Title: " + Course.courseTitle

IF Course.prerequisites is not empty THEN

PRINT "Prerequisites: " + join(Course.prerequisites, ", ")

ELSE

PRINT "No prerequisites"

END IF

RETURN from function

END IF

END FOR

PRINT "Course not found."

END

**Hash Table Implementation**

**1. Opening the File and Parsing the Data**

BEGIN

OPEN the file "courses.txt" for reading

IF the file cannot be opened THEN

PRINT "Error opening file."

EXIT program

END IF

...

FOR each line in the file DO

SPLIT the line into tokens

...

IF prerequisites are present THEN

FOR each prerequisite in prerequisites DO

IF prerequisite does NOT exist in courseTable THEN

PRINT "Error: Prerequisite course does not exist"

CONTINUE to next line

END IF

END FOR

END IF

...

END FOR

END

**2. Creating Course Objects and Storing Them in the Hash Table**

STRUCT Course

courseNumber (string)

courseTitle (string)

prerequisites (list of strings)

FUNCTION CreateCourseObject(courseNumber, courseTitle, prerequisites)

CREATE a new Course object

SET Course.courseNumber = courseNumber

SET Course.courseTitle = courseTitle

SET Course.prerequisites = prerequisites

RETURN Course object

BEGIN

FOR each line in the file DO

PARSE line into courseNumber, courseTitle, and prerequisites

CALL CreateCourseObject(courseNumber, courseTitle, prerequisites)

INSERT courseNumber, Course object into courseTable

END FOR

END

**3. Printing Course Information and Prerequisites**

FUNCTION PrintCourseInfo(courseNumber)

IF courseNumber exists in courseTable THEN

SET course = courseTable[courseNumber]

PRINT "Course Number: " + course.courseNumber

PRINT "Course Title: " + course.courseTitle

IF course.prerequisites is NOT empty THEN

PRINT "Prerequisites: " + join(course.prerequisites, ", ")

ELSE

PRINT "No prerequisites"

END IF

ELSE

PRINT "Course not found."

END IF

END

**Binary Search Tree (BST) Implementation**

**1. Opening the File, Reading Data, and Validating File Format**

BEGIN

OPEN file "courses.txt" FOR reading

IF file cannot be opened THEN

PRINT "Error: Unable to open file"

EXIT program

END IF

// Process each line in the file

FOR each line in the file DO

SPLIT the line into tokens (course number, title, prerequisites)

// Check if the line has at least two tokens (course number and title)

IF number of tokens < 2 THEN

PRINT "Error: Invalid line format"

CONTINUE to next line

END IF

// Extract the course number, title, and prerequisites

courseNumber = first token

courseTitle = second token

prerequisites = remaining tokens (if any)

// Validate prerequisites

IF prerequisites are present THEN

FOR each prerequisite in prerequisites DO

// Check if the prerequisite exists as a course in the file

IF prerequisite does NOT exist in course data THEN

PRINT "Error: Prerequisite course does not exist"

CONTINUE to next line

END IF

END FOR

END IF

END FOR

CLOSE file

END

**2. Creating Course Objects and Storing Them in the Binary Search Tree**

STRUCT Course

courseNumber (string)

courseTitle (string)

prerequisites (list of strings)

FUNCTION CreateCourseObject(courseNumber, courseTitle, prerequisites)

CREATE a new Course object

SET Course.courseNumber = courseNumber

SET Course.courseTitle = courseTitle

SET Course.prerequisites = prerequisites

RETURN Course object

BEGIN

DEFINE BinarySearchTree tree

// Process each line in the file

FOR each line in the file DO

SPLIT the line into tokens (course number, title, prerequisites)

// Extract course data

courseNumber = first token

courseTitle = second token

prerequisites = remaining tokens (if any)

// Create course object for valid data

courseObject = CreateCourseObject(courseNumber, courseTitle, prerequisites)

// Insert the course object into the binary search tree using the course number as key

tree.insert(courseNumber, courseObject)

END FOR

RETURN tree // Binary search tree containing all course objects

END

**3. Printing Course Information and Prerequisites**

FUNCTION PrintCourseInfo(courseNumber)

// Search the binary search tree by course number

IF courseNumber exists in tree THEN

SET course = tree.search(courseNumber)

PRINT "Course Number: " + course.courseNumber

PRINT "Course Title: " + course.courseTitle

// Print prerequisites if present

IF course.prerequisites is NOT empty THEN

PRINT "Prerequisites: " + join(course.prerequisites, ", ")

ELSE

PRINT "No prerequisites"

END IF

ELSE

PRINT "Course not found."

END IF

END

**2. Menu System Pseudocode**

BEGIN

REPEAT

PRINT "1. Load Course Data"

PRINT "2. Print Course List"

PRINT "3. Print Course Details"

PRINT "9. Exit"

GET userInput

IF userInput == 1 THEN

CALL LoadDataFromFile

ELSE IF userInput == 2 THEN

CALL PrintSortedCourseList

ELSE IF userInput == 3 THEN

PRINT "Enter course number: "

GET courseNumber

CALL SearchCourse(courseNumber)

ELSE IF userInput == 9 THEN

PRINT "Exiting program."

EXIT program

ELSE

PRINT "Invalid option, try again."

END IF

UNTIL userInput == 9

END

**3. Sorting and Displaying Course List**

**Vector Sorting (O(n log n))**

FUNCTION PrintCourseList():

Sort coursesVector by courseNumber

FOR each Course in coursesVector:

PRINT Course.courseNumber, Course.courseTitle

**Hash Table Sorting (O(n log n))**

FUNCTION PrintCourseList():

Convert coursesHashTable values to list

Sort list by courseNumber

FOR each Course in list:

PRINT Course.courseNumber, Course.courseTitle

**BST Sorting (O(n))**

FUNCTION PrintCourseList():

InOrderTraversal(coursesBST)

FUNCTION InOrderTraversal(node):

IF node is not null:

InOrderTraversal(node.left)

PRINT node.Course.courseNumber, node.Course.courseTitle

InOrderTraversal(node.right)

**4. Runtime & Memory Analysis**

Operation efficiency relies on data structure. The table below compares Vector, Hash Table, and Binary Search Tree (BST) Big-O complexity for various operations.

**Runtime Complexity Table**

| **Operation** | **Vector (O notation)** | **Hash Table (O notation)** | **BST (O notation)** |
| --- | --- | --- | --- |
| File Reading | O(n) | O(n) | O(n) |
| Insertion | O(1) (append) | O(1) (hashing) | O(log n) |
| Searching | O(n) | O(1) | O(log n) |
| Sorting | O(n log n) | O(n log n) | O(n) |
| Printing Course List | O(n log n) | O(n log n) | O(n) |

**5. Data Structure Evaluation**

Course data management requires the right data structure for efficiency and scalability. Vector, hash table, and binary search tree (BST) data structures have pros and cons depending on temporal complexity, memory utilization, and implementation ease. Data structures' efficiency, strengths, and weaknesses are compared here.

**Vector**

A dynamically resizable vector stores course data easily. Many programming languages include vector support, simplifying them. Vectors' sequential storage simplifies course listing. They effectively allocate memory without human intervention since they automatically expand to accommodate more components.

**Advantages**

Vectors are ideal for short courses since they are straightforward. Lite data structure with low memory use. Structure grows without human memory management with dynamic resizing. Iterating over a vector to show all courses is straightforward since items are stored in contiguous memory locations, making retrieval fast without sorting.

**Disadvantages**

Vectors are simple yet inefficient for searching and sorting. Course search is inappropriate for large datasets due to its O(n) linear scan. Since courses are saved in the order they are entered, manually sorting them before printing increases complexity to O(n log n). Changing components to insert or remove a course anywhere requires O(n) time. Vectors are inappropriate for frequent dataset switching applications.

**Hash Table**

A hash table links course numbers to objects using a hash algorithm. Its near-instant lookup rates make it ideal for frequent app searches and updates.

**Advantages**

Hash tables' primary advantage is O(1) search, insertion, and delete speeds. Hashes are faster than vectors for large datasets because they give instantaneous course access without iteration. For dynamic course management, they outperform vectors and binary search trees because they can insert and remove sections instantly. Hash tables can handle massive datasets without restructuring.

**Disadvantages**

Quick hash tables need more RAM for storage. This extra RAM lowers key-index collisions. Hash collisions slow performance and need chaining or open addressing. Another problem is that hash tables don't sort. Course values must be separated and sorted separately, increasing the time complexity to O(n log n). The unordered structure of hash databases makes frequent course retrieval difficult.

**Binary Search Tree (BST)**

A binary search tree (BST) organizes course data hierarchically. Courses are smaller on the left subtree and larger on the right for all nodes. This design simplifies searching, adding, and removing.

**Advantages**

BSTs naturally sort, which is great. BSTs may access courses in ascending order without sorting using an O(n) in-order traversal, unlike vectors and hash tables. A balanced BST searches for courses in O(log n) time, faster than a vector for large datasets. Logarithmic insertions and deletions are efficient in balanced trees. Applications that need fast lookups and ordered output benefit from BSTs.

**Disadvantages**

Despite their efficiency, BSTs are harder to design than vectors or hash tables. Maintaining balance is the largest issue since an uneven BST may search like a vector in O(n). Self-balancing BSTs like AVL or Red-Black trees fix issue, however rebalancing needs extra computation. BST insertions and deletions need more processing than hash tables since nodes may need to be rebuilt to maintain tree integrity. BSTs are too complicated for quick insertions and deletions.

**6. Recommendation**

Course data management structures are academic advisor-driven. Search efficiency and output sorting are top concerns. These demands help hashes and BSTs.

A hash table speeds up course information retrieval. hash tables are the quickest method to acquire course details in real time as searching, inserting, and removing require O(1). When aiding students, academic advisors must research course prerequisites and descriptions. The process is quicker using hash tables. For course additions and revisions, hash tables handle frequent insertions and modifications fast. Hash tables' unsorted nature is problematic. Creating sorted course lists for academic advisors takes O(n log n) time. Hashes aren't good for ordered course retrieval.

I like a BST for course sorting. Due to their ordered data structure, BSTs allow O(n) in-order traversal to find ascending courses. Use BSTs to create alphabetized course lists without sorting. BST course searches are quicker than vectors at O(log n). BST efficacy relies on balance. Unbalanced BSTs search and insert at O(n), making them vectors. Though more sophisticated, self-balancing BSTs like AVL or Red-Black trees may prevent this.

Search performance or sorted output should determine hash table or BST option. With O(1) search operations, a hash table is optimal for fast retrieval. For ordered course printing, BST's O(n) in-order traversal beats vector's O(n log n) sorting. The combination of a hash table for quick searches with a BST for ordered listing may balance fast retrieval and ordered listing. This would speed up course list organization for academic counselors. Course management system use case determines best way. Hash tables help academic advisors find courses individually. For organized course listings, use a BST.