# CS-300 Pseudocode Document - Project 1 (Henly)

## Function Signatures

This section is for using vectors to store course information. Below, I have expanded the example pseudocode with detailed pseudocode to cover all the program requirements, such as reading, validating, and storing course information.

**//Vector - Milestone 1**

void loadCourses(Vector<Course> courses, String fileName) {

// Step 1: Open the file

File inputFile

OPEN fileName AS inputFile

IF inputFile IS NOT OPEN THEN

PRINT "Error: Unable to open the file."

RETURN

ENDIF

// Step 2: Read Data from File

WHILE inputFile HAS LINES LEFT DO

String line

READ line FROM inputFile

// Step 3: Parse Each Line

SPLIT line BY "," INTO tokens

IF LENGTH(tokens) < 2 THEN

PRINT "Error: Incorrect format in line:", line

CONTINUE

ENDIF

// Step 4: Create a Course Object

Course course

course.courseNumber = tokens[0]

course.courseName = tokens[1]

// Step 5: Store Prerequisites

FOR INTEGER i = 2 TO LENGTH(tokens) - 1 DO

ADD tokens[i] TO course.prerequisites

ENDFOR

// Step 6: Store Course in Vector

ADD course TO courses

ENDWHILE

// Step 7: Close File

CLOSE inputFile

}

void validatePrerequisites(Vector<Course> courses) {

// Step 8: Validate Prerequisites

FOR EACH course IN courses DO

FOR EACH prerequisite IN course.prerequisites DO

BOOLEAN prerequisiteExists = FALSE

FOR EACH checkCourse IN courses DO

IF checkCourse.courseNumber == prerequisite THEN

prerequisiteExists = TRUE

BREAK

ENDIF

ENDFOR

IF prerequisiteExists == FALSE THEN

PRINT "Error: Prerequisite", prerequisite, "does not exist for course", course.courseNumber

ENDIF

ENDFOR

ENDFOR

}

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

// Step 9: Search and Print Course Information

BOOLEAN courseFound = FALSE

FOR EACH course IN courses DO

IF course.courseNumber == courseNumber THEN

PRINT "Course Number:", course.courseNumber

PRINT "Course Name:", course.courseName

PRINT "Prerequisites:", course.prerequisites

courseFound = TRUE

BREAK

ENDIF

ENDFOR

IF courseFound == FALSE THEN

PRINT "Course not found."

ENDIF

}

**// Hash Table - Milestone 2**

void loadCourses(HashTable<String, Course> courses, String fileName) {

// Step 1: Open the file

File inputFile

OPEN fileName AS inputFile

IF inputFile IS NOT OPEN THEN

PRINT "Error: Unable to open the file."

RETURN

ENDIF

// Step 2: Read Data from File

WHILE inputFile HAS LINES LEFT DO

String line

READ line FROM inputFile

// Step 3: Parse Each Line

SPLIT line BY "," INTO tokens

IF LENGTH(tokens) < 2 THEN

PRINT "Error: Incorrect format in line:", line

CONTINUE

ENDIF

// Step 4: Create a Course Object

Course course

course.courseNumber = tokens[0]

course.courseName = tokens[1]

// Step 5: Store Prerequisites

FOR INTEGER i = 2 TO LENGTH(tokens) - 1 DO

ADD tokens[i] TO course.prerequisites

ENDFOR

// Step 6: Store Course in Hash Table

INSERT (course.courseNumber, course) INTO courses

ENDWHILE

// Step 7: Close File

CLOSE inputFile

}

void validatePrerequisites(HashTable<String, Course> courses) {

// Step 8: Validate Prerequisites

FOR EACH course IN courses DO

FOR EACH prerequisite IN course.prerequisites DO

IF courses.CONTAINS\_KEY(prerequisite) == FALSE THEN

PRINT "Error: Prerequisite", prerequisite, "does not exist for course", course.courseNumber

ENDIF

ENDFOR

ENDFOR

}

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

// Step 9: Search and Print Course Information

IF courses.CONTAINS\_KEY(courseNumber) THEN

Course course = courses[courseNumber]

PRINT "Course Number:", course.courseNumber

PRINT "Course Name:", course.courseName

PRINT "Prerequisites:", course.prerequisites

ELSE

PRINT "Course not found."

ENDIF

}

**// Tree Data Structure - Milestone 3**

void loadCourses(BinarySearchTree<Course> courses, String fileName) {

// Step 1: Open the file

File inputFile

OPEN fileName AS inputFile

IF inputFile IS NOT OPEN THEN

PRINT "Error: Unable to open the file."

RETURN

ENDIF

// Step 2: Read Data from File

WHILE inputFile HAS LINES LEFT DO

String line

READ line FROM inputFile

// Step 3: Parse Each Line

SPLIT line BY "," INTO tokens

IF LENGTH(tokens) < 2 THEN

PRINT "Error: Incorrect format in line:", line

CONTINUE

ENDIF

// Step 4: Create a Course Object

Course course

course.courseNumber = tokens[0]

course.courseName = tokens[1]

// Step 5: Store Prerequisites

FOR INTEGER i = 2 TO LENGTH(tokens) - 1 DO

ADD tokens[i] TO course.prerequisites

ENDFOR

// Step 6: Insert Course into Binary Search Tree

INSERT course INTO courses

ENDWHILE

// Step 7: Close File

CLOSE inputFile

}

void validatePrerequisites(BinarySearchTree<Course> courses) {

// Step 8: Validate Prerequisites

FOR EACH course IN courses DO IN-ORDER TRAVERSAL

FOR EACH prerequisite IN course.prerequisites DO

BOOLEAN prerequisiteExists = FALSE

FOR EACH checkCourse IN courses DO IN-ORDER TRAVERSAL

IF checkCourse.courseNumber == prerequisite THEN

prerequisiteExists = TRUE

BREAK

ENDIF

ENDFOR

IF prerequisiteExists == FALSE THEN

PRINT "Error: Prerequisite", prerequisite, "does not exist for course", course.courseNumber

ENDIF

ENDFOR

ENDFOR

}

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

// Step 9: Search and Print Course Information

Course course = FIND courseNumber IN courses

IF course IS NOT NULL THEN

PRINT "Course Number:", course.courseNumber

PRINT "Course Name:", course.courseName

PRINT "Prerequisites:", course.prerequisites

ELSE

PRINT "Course not found."

ENDIF

}

// Function Definitions:

// BinarySearchTree<Course> - A binary search tree data structure to store course objects.

// INSERT - Inserts a course object into the binary search tree in sorted order by course number.

// FIND - Searches the binary search tree for a course with the specified course number and returns the course if found.

// IN-ORDER TRAVERSAL - A tree traversal method to ensure all nodes are visited in sorted order.

**// Menu Pseudocode - Project 1**

void displayMenu() {

PRINT "Welcome to ABCU's Advising Program!"

PRINT "1. Load course data into data structure"

PRINT "2. Print list of all courses in alphanumeric order"

PRINT "3. Print course information and prerequisites for a specific course"

PRINT "9. Exit program"

PRINT "Enter your choice: "

}

void menu(Vector<Course> vectorCourses, HashTable<String, Course> hashTableCourses, BinarySearchTree<Course> bstCourses) {

INTEGER choice = 0

WHILE choice != 9 DO

displayMenu()

READ choice FROM USER

SWITCH choice DO

CASE 1:

PRINT "Loading data into data structures..."

loadCourses(vectorCourses, "courses.txt")

loadCourses(hashTableCourses, "courses.txt")

loadCourses(bstCourses, "courses.txt")

PRINT "Data loaded successfully."

BREAK

CASE 2:

PRINT "Choose the data structure to print from:"

PRINT "1. Vector"

PRINT "2. Hash Table"

PRINT "3. Binary Search Tree"

INTEGER dsChoice = 0

READ dsChoice FROM USER

IF dsChoice == 1 THEN

sortAndPrintCourses(vectorCourses)

ELSE IF dsChoice == 2 THEN

sortAndPrintCourses(hashTableCourses)

ELSE IF dsChoice == 3 THEN

sortAndPrintCourses(bstCourses)

ELSE

PRINT "Invalid choice."

ENDIF

BREAK

CASE 3:

PRINT "Enter course number to search: "

STRING courseNumber

READ courseNumber FROM USER

PRINT "Choose the data structure to search from:"

PRINT "1. Vector"

PRINT "2. Hash Table"

PRINT "3. Binary Search Tree"

READ dsChoice FROM USER

IF dsChoice == 1 THEN

searchCourse(vectorCourses, courseNumber)

ELSE IF dsChoice == 2 THEN

searchCourse(hashTableCourses, courseNumber)

ELSE IF dsChoice == 3 THEN

searchCourse(bstCourses, courseNumber)

ELSE

PRINT "Invalid choice."

ENDIF

BREAK

CASE 9:

PRINT "Exiting the program. Goodbye!"

BREAK

DEFAULT:

PRINT "Invalid choice. Please enter a valid option."

ENDSWITCH

ENDWHILE

}

**// Sort & Print Courses Pseudocode - Project 1**

void sortAndPrintCourses(Vector<Course> courses) {

SORT courses BY courseNumber

FOR EACH course IN courses DO

PRINT "Course Number:", course.courseNumber

PRINT "Course Name:", course.courseName

PRINT "Prerequisites:", course.prerequisites

ENDFOR

}

void sortAndPrintCourses(HashTable<String, Course> courses) {

Vector<Course> courseList = courses.VALUES()

SORT courseList BY courseNumber

FOR EACH course IN courseList DO

PRINT "Course Number:", course.courseNumber

PRINT "Course Name:", course.courseName

PRINT "Prerequisites:", course.prerequisites

ENDFOR

}

void sortAndPrintCourses(BinarySearchTree<Course> courses) {

FOR EACH course IN courses DO IN-ORDER TRAVERSAL

PRINT "Course Number:", course.courseNumber

PRINT "Course Name:", course.courseName

PRINT "Prerequisites:", course.prerequisites

ENDFOR

}

### Final Runtime Analysis

| **Data Structure** | **Operation** | **Runtime** | **Explanation** |
| --- | --- | --- | --- |
| **Vector** | Loading Data | O(n) | Each course is processed and added sequentially. |
|  | Validate Prerequisites | O(n^2) | Compares each course against all others stored in the vector. |
|  | Search Course | O(n) | Requires scanning the entire vector. |
|  | Print Sorted Courses | O(n log n) | Vector must be sorted first, and then printed in O(n) time. |
|  | **Total** | O(n log n) | Comprehensive runtime for sorting and printing is O(n log n). |
| **Hash Table** | Loading Data | O(n) | Each course is inserted in constant average time, O(1). |
|  | Validate Prerequisites | O(n \* m) | Each prerequisite is checked, where 'm' is the average number of prerequisites per course. |
|  | Search Course | O(1) | Direct access allows constant time lookup. |
|  | Print Sorted Courses | O(n log n) | Extract entries in O(n), sort in O(n log n), and print in O(n). |
|  | **Total** | O(n log n) | The total runtime for extraction, sorting, and printing is O(n log n). |
| **Binary Search Tree** | Loading Data | O(n log n) | Each insertion takes O(log n) if the tree is balanced. |
|  | Validate Prerequisites | O(n \* log n) | Searching for each prerequisite takes O(log n). |
|  | Search Course | O(log n) | Efficient search in a balanced BST. |
|  | Print Sorted Courses | O(n) | In-order traversal of the BST prints courses in sorted order. |
|  | **Total** | O(n log n) | Balanced tree operations provide efficient overall performance. |

### Evaluation

#### Runtime and Memory Analysis

To evaluate the performance of the three data structures used to address the requirements, both runtime complexity and memory usage were considered.

The vector data structure has a loading time of O(n), with validation of prerequisites taking O(n^2) in the worst case and searching requiring O(n). Memory-wise, vectors require contiguous storage, which can lead to fragmentation issues for large datasets, but they are simple to implement and effective for small datasets.

The hash table, on the other hand, offers O(n) runtime for loading data, O(n \* m) for validation (with O(1) average lookup), and O(1) for searching. Hash tables generally require more memory compared to vectors due to collision handling, but they excel in average lookup and insertion times. Extracting and sorting from a hash table, however, is more complex.

The binary search tree (BST) has a loading runtime of O(n log n) if balanced, O(n \* log n) for validation, and O(log n) for searching. Memory usage for BSTs involves storing pointers, leading to some overhead, but it allows efficient in-order traversal for sorted output. However, BSTs can degrade to O(n) if unbalanced.

Based on the analysis, the hash table is recommended as it provides the most efficient average-case performance for the key operations of loading, searching, and validating. While sorting is less straightforward with a hash table compared to a BST, the overall performance gains make it the best fit. The BST is a good alternative if maintaining sorted order is crucial, but the risk of unbalanced performance makes it less favorable. The vector is the least efficient for large datasets, making it less suitable for this application, although it is easier to implement for simpler use cases.

#### Recommendation

Based on the runtime and memory analysis, the **hash table** data structure is recommended for this application. It provides average-case constant time complexity (O(1)) for loading, searching, and validating course prerequisites, which makes it highly efficient for most operations. While sorting is less efficient with hash tables compared to binary search trees, the overall performance gains in insertion and search operations make it the most suitable choice.

The **binary search tree** is a good alternative if maintaining the data in sorted order at all times is crucial, as it allows efficient in-order traversal. However, the risk of an unbalanced tree leading to O(n) performance makes it less favorable compared to the hash table for this application.

The **vector** data structure is the least efficient in terms of searching and validation operations, as both require linear time complexity (O(n)). While it is simpler to implement, it does not meet the performance requirements for efficiently handling large datasets.