# CS 300 Pseudocode Document

**//Vector - Milestone 1 – NOTE: My original submission I believe was incorrect and built a hash table. I changed the code to build a vector table for this submission.**

// Method to load courses from a CSV file into a vector-based data structure (e.g., VectorTable)

FUNCTION loadCourses(csvPath: STRING, dataStructure: VectorTable)

// Try to open the CSV file

file = openFile(csvPath)

// Handle case where file cannot be opened

IF file == NULL THEN

RETURN ERROR\_FILE\_NOT\_FOUND

// Loop through the file, reading line by line

WHILE NOT endOfFile(file) DO

// Read a line from the file

line = readLine(file)

// Split the line into its components (course name, course ID, prereqs)

values = splitLine(line)

// Check if the line contains at least Course ID and Name

IF values.size < 2 THEN

RETURN ERROR\_INVALID\_LINE // Handle invalid format (missing Course ID or Name)

// Create a new Course object

course = NEW Course()

course.courseId = values[1] // Course ID

course.name = values[0] // Course Name

// Initialize the prerequisite list for the course

course.prereqs = EMPTY\_LIST

// Handle case where the course has prerequisites

IF values.size > 2 THEN

FOR j = 2 TO values.size - 1 DO

prerequisite = values[j]

// Check if prerequisite already exists for this course

IF prerequisiteExists(course.courseId, prerequisite, dataStructure) THEN

CONTINUE // Skip duplicate prerequisites

ELSE

course.prereqs.append(prerequisite) // Add the valid prerequisite

// Add the course to the vector data structure

dataStructure.add(course)

// Close the file after processing

closeFile(file)

RETURN SUCCESS

END FUNCTION

// Struct to store course details

STRUCT Course

courseId: STRING // Unique course identifier

name: STRING // Name of the course

prereqs: LIST // List of prerequisite course IDs

END STRUCT

// Vector-based table to store courses

CLASS VectorTable

courses: LIST // Dynamic array (vector) to store courses

// Method to add a course to the vector table

FUNCTION add(course: Course)

courses.append(course) // Add the course to the vector

END FUNCTION

// Method to find a course by courseId

FUNCTION find(courseId: STRING): Course

FOR i = 0 TO courses.size - 1 DO

IF courses[i].courseId == courseId THEN

RETURN courses[i] // Return course if found

RETURN NULL // Return NULL if course not found

END FUNCTION

// Method to print all Computer Science courses in alphanumeric order

FUNCTION printAllCSCourses()

// Create a temporary list to store only CS courses

csCourses = EMPTY\_LIST

// Filter all courses that start with "CS" (assuming "CS" indicates Computer Science)

FOR i = 0 TO courses.size - 1 DO

IF startsWith(courses[i].courseId, "CS") THEN

csCourses.append(courses[i])

// Sort csCourses by courseId (alphanumeric order)

csCourses.sort() // Sorting based on the courseId

// Print all CS courses

FOR i = 0 TO csCourses.size - 1 DO

PRINT "Course ID: " + csCourses[i].courseId + ", Course Name: " + csCourses[i].name

END FUNCTION

// Method to print the title and prerequisites of a given course

FUNCTION printCourseAndPrereqs(courseId: STRING)

course = find(courseId) // Find the course by courseId

IF course == NULL THEN

PRINT "Course not found."

RETURN

// Print the course title

PRINT "Course Title: " + course.name

// Print the prerequisites if they exist

IF course.prereqs.size > 0 THEN

PRINT "Prerequisites: "

FOR i = 0 TO course.prereqs.size - 1 DO

PRINT course.prereqs[i]

ELSE

PRINT "No prerequisites."

END FUNCTION

END CLASS

// Method to load courses into the VectorTable

FUNCTION loadCoursesToVectorTable(csvPath: STRING, vectorTable: VectorTable)

// Reuse logic from loadCourses, but call vectorTable.add() to store courses

loadCourses(csvPath, vectorTable)

END FUNCTION

// Helper function to check if prerequisite already exists in the vector table

FUNCTION prerequisiteExists(courseId: STRING, prereqId: STRING, dataStructure: VectorTable): BOOLEAN

course = dataStructure.find(courseId) // Find course in the vector

IF course == NULL THEN

RETURN FALSE // Course not found

RETURN prereqId IN course.prereqs // Check if prereqId is already in prerequisites

END FUNCTION

// Helper function to split line into components

FUNCTION splitLine(line: STRING): LIST

RETURN SPLIT(line, ",") // Split the line by commas and return the list of values

END FUNCTION

// Helper function to open a file and return the file handle

FUNCTION openFile(filePath: STRING): FILE

file = OPEN(filePath)

IF file == ERROR THEN

RETURN NULL // Return NULL if file cannot be opened

RETURN file

END FUNCTION

// Helper function to close the file

FUNCTION closeFile(file: FILE)

CLOSE(file)

END FUNCTION

// Helper function to read a line from the file

FUNCTION readLine(file: FILE): STRING

RETURN READLINE(file) // Read a single line from the file

END FUNCTION

// Helper function to check if the end of the file is reached

FUNCTION endOfFile(file: FILE): BOOLEAN

RETURN EOF(file) // Return TRUE if the end of the file is reached

END FUNCTION

// Helper function to check if a string starts with a given prefix

FUNCTION startsWith(str: STRING, prefix: STRING): BOOLEAN

RETURN SUBSTRING(str, 0, prefix.length) == prefix

END FUNCTION

**//Hash Table - Milestone 2**

START

// Open the file containing course data

OPEN the file "course\_data.txt"

// Initialize an empty list to store all courses

CREATE an empty list "courseList"

// Loop through each line in the file

FOR each line in the file:

// Split the line by commas to extract course number, title, and prerequisites

SPLIT the line by commas into values: course number, title, prerequisites

// Edge case: Validate that there are at least 2 parameters (course number and title)

IF the number of values < 2:

// Log an error and skip the line if it contains less than 2 values

PRINT "Invalid line format, skipping this line"

CONTINUE to the next line

// Assign values from the line to the course object

courseNumber = values[0]

courseTitle = values[1]

// Edge case: Check for missing course number or title

IF courseNumber is empty OR courseTitle is empty:

// Log an error and skip the line if essential data is missing

PRINT "Missing course number or title, skipping this line"

CONTINUE to the next line

// Handle prerequisites if they exist

IF the number of values > 2:

// Store the prerequisites as a list (starting from the third value)

prerequisites = LIST(values[2 to end])

ELSE:

// Initialize prerequisites as an empty list if none are provided

prerequisites = EMPTY\_LIST

// Create a new "Course" structure and assign attributes

DEFINE a "Course" structure:

courseNumber = courseNumber

courseTitle = courseTitle

prerequisites = prerequisites

// Add the course to the courseList

ADD the Course object to "courseList"

// Edge case: Validate that all prerequisites exist in the courseList

FOR each course in courseList:

FOR each prerequisite in course.prerequisites:

// Check if the prerequisite exists in the courseList

IF prerequisite is not found in courseList:

// Log an error and skip invalid prerequisite

PRINT "Prerequisite [" + prerequisite + "] does not exist for course " + course.courseNumber

REMOVE the prerequisite from the course's prerequisites list

// Sort the courses in alphanumeric order by courseNumber

SORT "courseList" by courseNumber

// Print all Computer Science courses (assuming courses starting with "CS" are Computer Science courses)

PRINT "List of all Computer Science courses (in alphanumeric order):"

FOR each course in courseList:

IF course.courseNumber starts with "CS":

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

// Define a method to print a given course’s title and prerequisites

FUNCTION printCourseDetails(courseNumber):

// Search for the course by courseNumber

FIND the course in "courseList"

IF course is found:

PRINT "Course Number: " + course.courseNumber

PRINT "Course Title: " + course.courseTitle

// Edge case: Check if the course has prerequisites

IF course.prerequisites is not empty:

PRINT "Prerequisites:"

FOR each prerequisite in course.prerequisites:

PRINT prerequisite

ELSE:

PRINT "No prerequisites"

ELSE:

PRINT "Course not found."

// Loop to prompt user for input to print course details

PRINT "Enter a course number to view its details or type 'exit' to finish:"

WHILE True:

INPUT courseNumber

IF courseNumber equals "exit":

BREAK the loop

ELSE:

CALL printCourseDetails(courseNumber)

// Close the file after processing

CLOSE the file

ENDEnd//Binary Search Tree – Milestone 3

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

|  |
| --- |
| **Vector Pseudocode Analysis:** |
| Code Operation | Line Cost | # Times Executes | Total Cost |
| Create Vector | 1 | 1 | 1 |
| For each line in file (loop) | 1 | n | n |
| Create vector course item | 1 | n | n |
| While prerequisites exist (loop) | 1 | n (worst-case) | n |
| Append prerequisites to list | 1 | n (worst-case) | n |
| Pushback course item into vector | 1 | n | n |
| Total Cost |  |  | 5n + 1 |

|  |
| --- |
| **Hash Table Pseudocode Analysis:** |
| ode Operation | Line Cost | # Times Executes | Total Cost |
| Create hash table | 1 | 1 | 1 |
| For each line in file (loop) | 1 | n | n |
| Create course object | 1 | n | n |
| While prerequisites exist (loop) | 1 | n (worst-case) | n |
| Append prerequisites to list | 1 | n (worst-case) | n |
| Insert course object into hash table | O(1) | n | n \* O(1) = n |
| Handle collisions (linear probing) | O(n) | Occasional | Worst-case O(n) |

|  |
| --- |
| **Binary Search Tree (BST) Pseudocode Analysis:** |
| Create root node | 1 | 1 | 1 |
| For each line in file (loop) | 1 | n | n |
| Create course object | 1 | n | n |
| While prerequisites exist (loop) | 1 | n (worst-case) | n |
| Append prerequisites to list | 1 | n (worst-case) | n |
| Insert course into tree | O(log n) | n | n \* O(log n) = O(n log n) |
| Total Cost |  |  | O(n log n) |
| Create root node | 1 | 1 | 1 |

The vector data structure provides the simplest way to implement and store the data. It is easy to append unless you must resize the vector. The issues it has is that finding a course by course number requires scanning the entire list, which is an inefficient process. Secondly, sorting the vector alphanumerically requires a lot of time. Lastly, as the vector scales larger and larger, the searching and sorting becomes worse due to the linear nature of the vector storage system. While the number of courses in a specific major may not me that large, the number of courses offered by a whole school may be too larger for this structure to efficiently manage.

The hash tables are highly efficient for searching, inserting, and retrieving course information. They can handle a large amount of data as long as the hash function distributes keys evenly which would reduce the likelihood of collisions. Also, this structure makes it easy to see if there is a prerequisite for a course. The disadvantages are that the tables do not maintain data in any sorted order so printing a list alphanumerically requires a separate sorting step. If the key is inefficient and there are a lot of collisions, then the efficiency of the list will degrade quickly, but if you develop a good hash function this should not be an issue. Lastly, hash tables require more memory that the other data structures because you need to maintain empty slots for collision resolution.

The BST structures are great for maintaining order so that printing a list in alphanumeric order is easy. The searches are efficient and looking up prerequisites does not take much time. Also, insertion, deletion, and searching can be done quickly. The problems occur if the tree becomes unbalanced. The tree is more complex when compared to a hash table or vector and maintaining the balance requires more complex logic.

I would recommend the hash table structure for this task. It offers O(1) average-case insertion and lookup, which is efficient for handling a potentially large dataset of courses. While collisions can increase complexity, using a good hash function and resizing the table can mitigate the impact of collisions, keeping performance near O(n) in most cases. It balances well between speed and ease of implementation compared to trees, where balancing is necessary to avoid degradation to O(n^2). I think that this structure would produce the quickest results for the most used functions which I believe would be the search function. While it may be the slowest for printing out in alphanumeric order, I think that they may only need to do that once. I am an advisor, and we have a separate file/ sheet that breaks down all the classes in alphanumeric order. None of the students or myself ever must call on the data structure to print this out, it is just updated each semester as new classes are added, or others are taken away.