# CS 300 Pseudocode Document

**//Prerequisites Used for all data structures**

FUNCTION displayCourseWithPrerequisites(courseTable, courseNumber)

// Attempt to retrieve the course from the table

course = courseTable.lookup(courseNumber)

IF course IS NULL THEN

PRINT "Course '" + courseNumber + "' not found in catalog."

RETURN

ENDIF

PRINT "Course: " + course.courseNumber + " - " + course.courseTitle

PRINT "Prerequisites:"

IF course.prerequisites IS EMPTY THEN

PRINT " - None (This course has no prerequisites)"

ELSE

FOR EACH prereqNumber IN course.prerequisites DO

prereqCourse = courseTable.lookup(prereqNumber)

IF prereqCourse IS NOT NULL THEN

PRINT " - " + prereqCourse.courseNumber + ": " + prereqCourse.courseTitle

ELSE

PRINT " - " + prereqNumber + " (Missing course record)"

ENDIF

ENDFOR

ENDIF

END  
  
**//Menu**

WHILE true DO

PRINT "Menu:"

PRINT " 1. Load Courses"

PRINT " 2. Display All Courses"

PRINT " 3. Find Course"

PRINT " 9. Exit"

INPUT choice

SWITCH choice

CASE 1:

CALL loadCoursesFromFile(...)

CASE 2:

CALL printCourses(...)

CASE 3:

INPUT courseNumber

CALL printCourse(...)

CASE 9:

PRINT "Goodbye!"

EXIT PROGRAM

DEFAULT:

PRINT "Invalid selection."

ENDSWITCH

ENDWHILE  
 **//Vector - Milestone 1**

STRUCT Course

String courseNumber

String courseTitle

Vector<String> prerequisites

ENDSTRUCT

FUNCTION loadCourses\_Vector(filename, OUT Vector<Course> courses)

DECLARE Set<String> courseNumbers

OPEN file

IF file NOT opened THEN

PRINT "Error: File could not be opened."

RETURN

FOR EACH line IN file DO

SPLIT line BY ',' INTO tokens

IF tokens.length < 2 THEN

PRINT "Error: Invalid line format."

CONTINUE

CREATE new Course c

c.courseNumber = tokens[0]

c.courseTitle = tokens[1]

FOR i FROM 2 TO tokens.length - 1 DO

c.prerequisites.ADD(tokens[i])

ENDFOR

courses.ADD(c)

courseNumbers.ADD(c.courseNumber)

ENDFOR

FOR EACH c IN courses DO

FOR EACH prereq IN c.prerequisites DO

IF prereq NOT IN courseNumbers THEN

PRINT "Warning: Missing prerequisite " + prereq

ENDIF

ENDFOR

ENDFOR

END

FUNCTION quickSortCourses(Vector<Course> &courses, int low, int high)

IF low < high THEN

pivot = partition(courses, low, high)

CALL quickSortCourses(courses, low, pivot - 1)

CALL quickSortCourses(courses, pivot + 1, high)

END

FUNCTION partition(Vector<Course> &courses, int low, int high)

pivot = courses[high].courseNumber

i = low - 1

FOR j FROM low TO high - 1 DO

IF courses[j].courseNumber < pivot THEN

i = i + 1

SWAP courses[i] WITH courses[j]

ENDFOR

SWAP courses[i + 1] WITH courses[high]

RETURN i + 1

END

FUNCTION printCourses\_Vector(Vector<Course> courses)

CALL quickSortCourses(courses, 0, courses.size - 1)

FOR EACH course IN courses DO

PRINT course.courseNumber + ": " + course.courseTitle

END

FUNCTION searchCourse\_Vector(Vector<Course> courses, String courseNumber)

FOR EACH course IN courses DO

IF course.courseNumber == courseNumber THEN

PRINT course.courseNumber + ": " + course.courseTitle

IF course.prerequisites IS EMPTY THEN

PRINT "Prerequisites: None"

ELSE

PRINT "Prerequisites:"

FOR EACH prereq IN course.prerequisites DO

PRINT " - " + prereq

RETURN

ENDFOR

PRINT "Course not found."

END

Vectors are simple to implement and memory efficient. They allow fast appends and linear iteration, which makes them ideal for loading and displaying course data. However, vectors are not ideal for searching or validating prerequisites, since those operations require scanning through the list one item at a time resulting in O(n) time complexity. This becomes inefficient as the number of courses grows, especially when validating multiple prerequisites per course.

**//Hash Table - Milestone 2**

CLASS Course

String courseNumber

String courseTitle

List<String> prerequisites

ENDCLASS

FUNCTION loadCourses\_HashTable(filename) RETURNS HashTable<Course>

DECLARE courseTable AS HashTable<Course>

DECLARE Set<String> validCourseNumbers

OPEN file

IF file NOT opened THEN

PRINT "Error: File could not be opened."

RETURN empty table

FOR EACH line IN file DO

SPLIT line INTO tokens

IF tokens.length < 2 THEN CONTINUE

courseNumber = tokens[0]

courseTitle = tokens[1]

prerequisites = tokens[2+]

CREATE Course c(courseNumber, courseTitle, prerequisites)

courseTable[courseNumber] = c

validCourseNumbers.ADD(courseNumber)

ENDFOR

FOR EACH course IN courseTable DO

FOR EACH prereq IN course.prerequisites DO

IF prereq NOT IN validCourseNumbers THEN

PRINT "Warning: Missing prerequisite " + prereq

ENDIF

ENDFOR

ENDFOR

RETURN courseTable

END

FUNCTION printCourses\_HashTable(HashTable<Course> courseTable)

DECLARE courseNumbers AS List<String>

FOR EACH key IN courseTable DO

courseNumbers.ADD(key)

CALL quickSortStrings(courseNumbers, 0, courseNumbers.size - 1)

FOR EACH courseNumber IN courseNumbers DO

course = courseTable[courseNumber]

PRINT course.courseNumber + ": " + course.courseTitle

END

FUNCTION quickSortStrings(List<String> &arr, int low, int high)

IF low < high THEN

pivot = partition(arr, low, high)

CALL quickSortStrings(arr, low, pivot - 1)

CALL quickSortStrings(arr, pivot + 1, high)

END

FUNCTION partition(List<String> &arr, int low, int high)

pivot = arr[high]

i = low - 1

FOR j FROM low TO high - 1 DO

IF arr[j] < pivot THEN

i = i + 1

SWAP arr[i] WITH arr[j]

ENDFOR

SWAP arr[i + 1] WITH arr[high]

RETURN i + 1

END

FUNCTION searchCourse\_HashTable(HashTable<Course> courseTable, String courseNumber)

IF courseNumber NOT IN courseTable THEN

PRINT "Course not found."

RETURN

course = courseTable[courseNumber]

PRINT course.courseNumber + ": " + course.courseTitle

IF course.prerequisites IS EMPTY THEN

PRINT "Prerequisites: None"

ELSE

FOR EACH prereq IN course.prerequisites DO

IF prereq IN courseTable THEN

PRINT " - " + courseTable[prereq].courseTitle

ELSE

PRINT " - " + prereq + " (Missing)"

END

Hash tables offer excellent performance for lookups and validations. Each course can be inserted and retrieved in constant time on average (O(1)), making it ideal for frequent access operations like verifying if a course exists or checking its prerequisites. The trade-off is that hash tables don’t maintain order, so an extra sorting step is needed to display courses alphabetically or numerically. They also consume more memory due to internal structure overhead. Despite that, they provide a near-instant response for most advising tasks.

**//Binary Search Tree – Milestone 3**

STRUCT Course

String courseNumber

String courseTitle

List<String> prerequisites

ENDSTRUCT

CLASS Node

Course course

Node left

Node right

ENDCLASS

CLASS BinarySearchTree

Node root

FUNCTION Insert(Course)

FUNCTION Search(courseNumber) RETURNS Course

FUNCTION InOrderTraversal()

ENDCLASS

FUNCTION loadCourses\_BST(filename) RETURNS BinarySearchTree

DECLARE courseMap AS Map<String, Course>

DECLARE bst AS BinarySearchTree

OPEN file

IF file NOT opened THEN

PRINT "Error: File could not be opened."

RETURN bst

FOR EACH line IN file DO

SPLIT line INTO tokens

IF tokens.length < 2 THEN CONTINUE

CREATE Course c

c.courseNumber = tokens[0]

c.courseTitle = tokens[1]

c.prerequisites = tokens[2+]

courseMap[c.courseNumber] = c

ENDFOR

FOR EACH course IN courseMap DO

FOR EACH prereq IN course.prerequisites DO

IF prereq NOT IN courseMap THEN

PRINT "Warning: Missing prerequisite " + prereq

ENDIF

bst.Insert(course)

ENDFOR

RETURN bst

END

FUNCTION printCourses\_BST(BinarySearchTree bst)

CALL inOrderTraversal(bst.root)

FUNCTION inOrderTraversal(Node node)

IF node IS NOT NULL THEN

CALL inOrderTraversal(node.left)

PRINT node.course.courseNumber + ": " + node.course.courseTitle

CALL inOrderTraversal(node.right)

END

FUNCTION searchCourse\_BST(BinarySearchTree bst, String courseNumber)

course = bst.Search(courseNumber)

IF course IS NULL THEN

PRINT "Course not found."

RETURN

PRINT course.courseNumber + ": " + course.courseTitle

IF course.prerequisites IS EMPTY THEN

PRINT "Prerequisites: None"

ELSE

FOR EACH prereq IN course.prerequisites DO

PRINT " - " + prereq

END

BSTs naturally store data in sorted order, allowing in-order traversal to produce an ordered list without sorting. They also support reasonably efficient insertions and searches in average cases (O(log n)). However, BSTs are more complex to implement and manage, especially if balancing is not enforced. In worst-case scenarios (e.g., inserting sorted input into an unbalanced BST), performance degrades to O(n), matching a simple vector.

**Runtime Analysis**

**Vector:**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | 1 | n | n |
| **Read each line of file** | 1 | n | n |
| |  | | --- | | **Parse line into course object** |  |  | | --- | |  | | 1 | n | n |
| |  | | --- | | **Insert course into vector** |  |  | | --- | |  | | 1 | n | n |
| **print the prerequisite course information** | 1 | n | n |
| **Total Cost** | | | 4n |
| **Runtime** | | | O(n) |

**HASH TABLE:**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Read each line of file** | 1 | n | n |
| **Parse line into course object** | 1 | n | n |
| **Hash and insert into table** | 1 | n | n |
| **Total Cost** | | | 3n |
| **Runtime** | | | O(n) |

**BINARY SEARCH TREE:**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Read each line of file** | 1 | n | n |
| |  | | --- | | **Parse line into course object** |  |  | | --- | |  | | 1 | n | n |
| |  | | --- | | **Insert into BST (worst-case)** |  |  | | --- | |  | | 1 | n | n^2 |
| **Total Cost** | | | (n^2) + 2n |
| **Runtime** | | | O(n^2) if worst  O(n log n) average |

**Thoughts**:

Each data structure has its strengths and weaknesses when it comes to supporting ABCU’s course advising system.   
  
**Vectors** are straightforward and efficient for loading and displaying course data, especially in smaller datasets. They’re easy to work with and great for quick setups, but not ideal when it comes to searching or checking prerequisites since those tasks require scanning through the entire list, which gets slow as the data grows. You also have to manually sort the list to show courses alphabetically.   
  
In contrast, **hash tables** are built for speed. They let you look up courses and validate prerequisites almost instantly. The downside is they don’t keep things in order, so if you need to print courses alphabetically, you’ll have to extract and sort the keys separately. Still, their speed and efficiency make them a strong candidate for tools that rely on frequent access and validation.

**Binary Search Trees** fall somewhere in between. They naturally keep data sorted, which is a huge plus when displaying an ordered course list. They also handle searches pretty efficiently unless they become unbalanced, in which case performance can slow down significantly. They’re also a bit more complex to implement and maintain.

After weighing everything, the **hash table** stands out as the best choice for this project. It handles lookups and prerequisite checks quickly and efficiently, and although sorting takes a bit more work, that trade-off is worth it. Overall, it’s the best fit for a tool that needs to be both fast and reliable.

Based on the project’s goals and the performance trade-offs across all three data structures, I recommend using a **hash**. The ability to quickly search for courses and validate prerequisites is essential, and hash tables excel in both areas with constant time performance in most cases. While it doesn’t maintain order by default, that can be easily solved by exporting and sorting the keys when needed. Compared to vectors and binary search trees, the hash table offers the best mix of speed, simplicity, and reliability especially for a tool like this that will be used frequently and needs to scale well over time. It's the most practical and efficient option for ABCU's advising system.