MENU LOOP PSEUDOCODE

```
START

DECLARE is_data_loaded = False

FUNCTION display_menu()
        PRINT "1. Load Data Structure of Choice"
        PRINT "2. Print Course List"
        PRINT "3. Print Course Info"
        PRINT "9. Exit"

WHILE True:
        CALL display_menu()
        GET user_choice
        switch(user_choice)

                1:
                        PROMPT user for “(V)ector, (H)ash Table, (T)ree”
                        LOAD .csv INTO chosen data_structure
                        SET is_data_loaded = True
                2:
                        IF is_data_loaded is False
                                PRINT "Please load data first."
                        ELSE
                                PrintAll(dataStructure)
                3:
                        IF is_data_loaded is False
                                PRINT "Please load data first."
                        ELSE
                                PROMPT "Enter course number:"
                                PrintOne(dataStructure, courseNum)
                9: Default:
                        BREAK



END
```

CLASS Course PSEUDOCODE

```
//maybe struct actually? [APPLIES TO ALL DATA STRUCTURES]
DEFINE Class Course with
        - courseNumber : STRING
        - courseTitle : STRING
        - prerequisites : LIST of STRING

        Course(number, title, prereqs) //constructor
                courseNumber = number
                courseTitle = title
                prerequisites = prereqs
END CLASS
```

VECTOR PSEUDOCODE

```
FUNCTION PrintAllVector(vector)
        SORT courseList BY course.courseNumber (alphanumeric)
        FOR Course IN vector
                PRINT “Course Number: ”  + course.courseNumber
                PRINT “Course Title: ”  + course.courseTitle
                IF course.prerequisites IS NOT empty
                        PRINT "Prerequisites: "
                        FOR prereq IN course.prerequisites
                                PRINT "  - " + prereq


FUNCTION InsertInToVector(vector, course)
        vector.push_back(course)
```

HASH TABLE PSEUDOCODE

```
FUNCTION PrintAllHashTable(hashTable)
        EXTRACT all values FROM hashTable INTO courseList
        SORT courseList BY course.courseNumber (alphanumeric)
        FOR each course IN courseList
                PRINT “Course Number: ”  + course.courseNumber
                PRINT “Course Title: ”  + course.courseTitle
                IF course.prerequisites IS NOT empty
                        PRINT "Prerequisites: "
                        FOR prereq IN course.prerequisites
                                PRINT "  - " + prereq


FUNCTION InsertIntoHashTable(hashTable, course)
        SET key = hash conversion of course.courseNumber
        APPEND course AT hashTable[key]
                IF collision:
                        MOVE tree nodes
```

BINARY SEARCH TREE PSEUDOCODE

```
FUNCTION InsertIntoBST(course)
        CALL recursiveInsert(root, course)

FUNCTION recursiveInsert(node, course)
        IF node IS NULL
                RETURN new BSTNode with course
        IF course.courseNumber < node.data.courseNumber
                node.left = recursiveInsert(node.left, course)
        ELSE
                node.right = recursiveInsert(node.right, course)
        RETURN node
```

```
START

FUNCTION InOrderPrint(node)
        // INORDER: LEFT -> ROOT -> RIGHT
        IF node IS NOT NULL
                CALL RecursivePrint(node.left)

                PRINT "Course Number: " + node.data.courseNumber
                PRINT "Title: " + node.data.title

                IF node.data.prerequisites IS NOT EMPTY
                PRINT "Prerequisites: "
                FOR prereq IN node.data.prerequisites
                PRINT "  - " + prereq
                ELSE
                PRINT "Prerequisites: None"

                PRINT newline

                CALL RecursivePrint(node.right)

END
```

BIG O ANALYSIS

VECTOR

Insertion is O(1), because push_back is constant time and no shifting is needed
Search is O(N), because each element needs to be checked at worst
Sort can be O(n log n), with something like quicksort

HASH TABLE

Insertion is O(1), if a good hashing function is used, since everything maps to a bucket
Search should be close to O(1) unless all map to single bucket
Sort would be close to O(N log N) to extract and sort them

BINARY SEARCH TREE

Tree has height log n so insertion would be O(log n)
Search would be O(log n) since requires looking from root to leaf (based on height)
Tree would already be sorted since tree maintains structure, to print would be O(n)

Conclusion: If the system is going to be used for searches a lot then a hash table would be recommended since it is the quickest even at its worst performance. If students / teachers are going to frequently print out the whole list of courses then I would say a BST is the best option. Since I feel like students and teachers are likely to search for single courses often and only

occasionally print the full list, Hash Table offers the most efficient average performance overall and for that reason its what I'll be using for Project Two in module 7.