CS-300 Project One:

Owen McCostis

Pseudocode:

Code Shared By All Three Data Structures:

```
// All three data structures will store the same Course objects

// Define a Course structure
STRUCT Course:
      courseNumber: String
      name: String
      prerequisites: List of Strings
END STRUCT

// CSV Input File Format: courseNumber, courseName, [prerequisites]

// Split the string 'line' by the character 'delimiter' and returns
a list of tokens
FUNCTION split(line, delimiter):
      // Implementation details are abstracted
      RETURN list_of_tokens
END FUNCTION

// Print course info and prerequisites
FUNCTION printCourseInfo(course):
      PRINT "Course Number: " + course.courseNumber
      PRINT "Course Name: " + course.name
      IF course.prerequisites is empty:
            PRINT "Prerequisites: None"
      ELSE:
            PRINT "Prerequisites:"
            FOR EACH p IN course.prerequisites:
```

```
                PRINT " - " + p
            END FOR
      END IF
      PRINT "[Line]"
END FUNCTION
```

Vector Data Structure (Milestone 1)

```
// Open and read file, parse lines, and check formatting:
FUNCTION loadDataIntoVector(filename, coursesVector,
courseNumbersSet):
      OPEN FILE filename FOR reading
      IF file NOT open:
            PRINT "Error: Cannot open file."
            RETURN
      END IF

      // While not end of file
      WHILE NOT EOF:
            line = READ line FROM file
            IF line IS empty:
                  CONTINUE
            END IF

            tokens = split(line, ',')
            IF length(tokens) < 2:
                  PRINT "Error: Invalid line format."
                  CONTINUE
            END IF

            // Create course objects
            course = new Course
            course.courseNumber = tokens[0]
            course.name = tokens[1]

            FOR i FROM 2 TO length(tokens)-1:
                  APPEND tokens[i] TO course.prerequisites
            END FOR

            APPEND course TO coursesVector
            ADD course.courseNumber TO courseNumbersSet
```

```
      END WHILE

      CLOSE file

      // Validate prerequisites
      FOR EACH c IN coursesVector:
            FOR EACH p IN c.prerequisites:
                  IF p NOT IN courseNumbersSet:
                        PRINT "Error: Prerequisite doesn't exist."
                  END IF
            END FOR
      END FOR
END FUNCTION

// Print all courses in sorted (alphanumeric) order
FUNCTION printAllCoursesVector(coursesVector):
      // Sort coursesVector by courseNumber
      SORT coursesVector BY course.courseNumber IN ASC ORDER

      FOR EACH c IN coursesVector:
            PRINT c.courseNumber + ", " + c.name
      END FOR
END FUNCTION

// Search for a specific course
FUNCTION searchCourseVector(coursesVector, courseNumber):
      FOR EACH c IN coursesVector:
            IF c.courseNumber == courseNumber:
                  RETURN c
            END IF
      END FOR
      RETURN null
END FUNCTION
```

```
// Menu
FUNCTION vectorMenu():
      coursesVector = empty list OF Course
      courseNumbersSet = empty set OF String
      dataLoaded = FALSE

      DO:
            PRINT "Menu:"
            PRINT "1. Load Data"
            PRINT "2. Print All Courses"
            PRINT "3. Print A Course"
            PRINT "9. Exit"

            choice = READ integer FROM user

            IF choice == 1:
                  CALL loadDataIntoVector("courses.txt",
coursesVector, courseNumbersSet)
                  dataLoaded = TRUE
            ELSE IF choice == 2:
                  IF NOT dataLoaded:
                        PRINT "Please load data first."
                  ELSE:
                        printAllCoursesVector(coursesVector)
                  END IF
            ELSE IF choice == 3:
                  IF NOT dataLoaded:
                        PRINT "Please load data first."
                  ELSE:
                        PRINT "Enter course number:"
                        cn = READ string
                        found = searchCourseVector(coursesVector, cn)
```

```
                        IF found == null:
                              PRINT "Course not found."
                        ELSE:
                              printCourseInto(found)
                        END IF
                  ELSE IF choice == 9:
                        PRINT "Exiting."
                        BREAK
                  ELSE:
                        PRINT "Invalid choice. Try again."
                  END IF
      WHILE choice != 9
END FUNCTION
```

Hash Table Data Structure (Milestone 2)

```
// Load data into a hash table
FUNCTION loadDataIntoHashTable(filename, coursesMap,
courseNumbersSet):
      // coursesMap is a map from String (courseNumber) to Course
      OPEN FILE filename FOR reading
      IF file NOT open:
            PRINT "Error: Cannot open file"
            RETURN
      END IF

      // While not end of file
      WHILE NOT EOF:
            line = READ line
            IF line IS empty:
                  CONTINUE
            END IF

            tokens = split(line, ',')
```

```
            IF length(tokens) < 2:
                  PRINT "Error: Invalid line format."
                  CONTINUE
            END IF

            // Create course objects
            course = new Course
            course.courseNumber = tokens[0]
            course.name = tokens[1]

            FOR i FROM 2 TO length(tokens)-1:
                  APPEND tokens[i] TO course.prerequisites
            END FOR

            coursesMap[course.courseNumber] = course
            ADD course.courseNumber TO courseNumbersSet
      END WHILE

      CLOSE file

      // Validate prerequisites
      FOR EACH key IN coursesMap:
            c = coursesMap[key]
            FOR EACH p IN c.prerequisites:
                  IF p NOT IN courseNumbersSet:
                        PRINT "Error: Prerequisite doesn't exist."
                  END IF
            END FOR
      END FOR
END FUNCTION

// Print all courses in sorted (alphanumeric) order
FUNCTION printAllCoursesHashTable(coursesMap):
```

```
      tempList = empty list OF Course
      FOR EACH key IN coursesMap:
            APPEND coursesMap[key] TO tempList
      END FOR

      SORT tempList BY course.courseNumber IN ASC ORDER

      FOR EACH c IN tempList:
            PRINT c.courseNumber + ", " + c.name
      END FOR
END FUNCTION

// Search for a specific course
FUNCTION searchCourseHashTable(coursesMap, courseNumber):
      IF courseNumber IN coursesMap:
            RETURN coursesMap[courseNumber]
      ELSE:
            RETURN null
      END IF
END FUNCTION

// Menu
FUNCTION hashTableMenu():
      coursesMap = empty map(String -> Course)
      courseNumbersSet = empty set OF String
      dataLoaded = FALSE

      DO:
            PRINT "Menu:"
            PRINT "1. Load Data"
            PRINT "2. Print All Courses"
            PRINT "3. Print A Course"
            PRINT "9. Exit"
```

```
          choice = READ integer FROM user

          IF choice == 1:
               loadDataIntoHashTable("courses.txt", coursesMap,
courseNumbersSet)
               dataLoaded = TRUE
          ELSE IF choice == 2:
               IF NOT dataLoaded:
                    PRINT "Please load data first."
               ELSE:
                    printAllCoursesHashTable(coursesMap)
               END IF
          ELSE IF choice == 3:
               IF NOT dataLoaded:
                    PRINT "Please load data first."
               ELSE:
                    PRINT "Enter course number:"
                    cn = READ string
                    found = searchCourseHashTable(coursesMap, cn)
                    IF found == null:
                         PRINT "Course not found."
                    ELSE:
                         printCourseInfo(found)
                    END IF
               END IF
          ELSE IF choice == 9:
               PRINT "Exiting."
               BREAK
          ELSE:
               PRINT "Invalid choice. Try again."
          END IF
     WHILE choice != 9
```

```
END FUNCTION
```

Binary Search Tree Data Structure (Milestone 3)

```
// Define a Binary Search Tree (BST) Node Structure
STRUCT BSTNode:
      course: Course
      left: BSTNode
      right: BSTNode
END STRUCT

// Define a BST Structure
STRUCT BST:
      root: BSTNode
END STRUCT

// Insert a Course into the tree by courseNumber
FUNCTION insertBST(root, course):
      IF root IS null:
            root = new BSTNode
            root.course = course
            root.left = null
            root.right = null
            RETURN root
      END IF

      IF course.courseNumber < root.course.courseNumber:
            root.left = insertBST(root.left, course)
      ELSE IF course.courseNumber > root.course.courseNumber:
            root.right = insertBST(root.right, course)
      END IF
      RETURN root
END FUNCTION
```

```
// Search for a course by courseNumber in the tree
FUNCTION searchBST(root, cn):
      IF root IS null:
            RETURN null
      END IF

      IF cn == root.course.courseNumber:
            RETURN root.course
      ELSE IF cn < root.course.courseNumber
            RETURN searchBST(root.left, cn)
      ELSE:
            RETURN searchBST(root.right, cn)
      END IF
END FUNCTION

// Traverse the tree in-order and perform an action on each course
FUNCTION inOrderTraverse(root, actionFunction):
      IF root IS NOT null:
            inOrderTraverse(root.left, actionFunction)
            actionFunction(root.course)
            inOrderTraverse(root.right, actionFunction)
      END IF
END FUNCTION

// Load data into BST
FUNCTION loadDataIntoBST(filename, bst, courseNumbersSet):
      OPEN FILE filename FOR reading
      IF file NOT open:
            PRINT "Error: Cannot open file."
            RETURN
      END IF

      // While not end of file
```

```
WHILE NOT EOF:
      line = READ line
      IF line IS empty:
            CONTINUE
      END IF

      tokens = split(line, ‘,’)
      IF length(tokens) < 2:
            PRINT “Error: Invalid line.”
            CONTINUE
      END IF

      // Create course objects
      course = new Course
      course.courseNumber = tokens[0]
      course.name = tokens[1]

      FOR i FROM 2 TO length(tokens)-1:
            APPEND tokens[i] TO course.prerequisites
      END FOR

      bst.root = insertBST(bst.root, course)
      ADD course.courseNumber TO courseNumbersSet
END WHILE

CLOSE file

// Validate prerequisites
FUNCTION validatePrereqs(c):
      FOR EACH p IN c.prerequisites:
            foundC = searchBST(bst.root, p)
            IF foundC IS null:
                  PRINT “Error: Prerequisite doesn’t exist.”
```

```
                END IF
            END FOR
        END FUNCTION

        inOrderTraverse(bst.root, validatePrereqs)
END FUNCTION

// Print all courses in sorted (alphanumeric) order
FUNCTION printAllCoursesBST(bst):
        FUNCTION printLine(c):
            PRINT c.courseNumber + ", " + c.name
        END FUNCTION

        inOrderTraverse(bst.root, printLine)
END FUNCTION

// Menu
FUNCTION bstMenu():
        bst = new BST
        bst.root = null
        courseNumbersSet = empty set OF String
        dataLoaded = FALSE

        DO:
            PRINT "Menu:"
            PRINT "1. Load Data"
            PRINT "2. Print All Courses"
            PRINT "3. Print A Course"
            PRINT "9. Exit"

            choice = READ integer FROM user

            IF choice == 1:
```

```
                loadDataIntoBST("courses.txt", bst,
courseNumbersSet)
                dataLoaded = TRUE
          ELSE IF choice == 2:
                IF NOT dataLoaded:
                      PRINT "Please load data first."
                ELSE:
                      printAllCoursesBST(bst)
                END IF
          ELSE IF choice == 3:
                IF NOT dataLoaded:
                      PRINT "Please load data first."
                ELSE:
                      PRINT "Enter course number:"
                      cn = READ string
                      found = searchBST(bst.root, cn)
                      IF found IS null:
                            PRINT "Course not found."
                      ELSE:
                            printCourseInfo(found)
                      END IF
                END IF
          ELSE IF choice == 9:
                PRINT "Exiting."
                BREAK
          ELSE:
                PRINT "Invalid choice. Try again."
          END IF
     WHILE choice != 9
END FUNCTION
```

Runtime Analysis:

Vector:

Reading each line	O(n)
Splitting line and creating object	O(1) per line, O(n) total
Inserting into vector	O(1) amortized per insertion, O(n) total
Adding courseNumber to a set	Typically O(1) on average, so O(n) total
Validating prerequisites	Each course checks its prerequisites. Assume total prerequisites = P. Checking membership in a set is O(1). Total O(P). Typically $P \leq kn$ for some constant k, so O(n)
Overall complexity	O(n)

Hash Table:

Reading all lines	O(n)
Creating objects	O(n)
Inserting into hash table	O(1) average per insertion, O(n) total
Validating prerequisites	Same as vector, O(n) total assuming O(1) membership checks in a set.
Overall complexity	O(n)

Binary Search Tree:

Reading lines	O(n)
Creating objects	O(n)
Inserting into BST	On average O(log n) per insertion, O(n log n) total worst case (assuming BST is balanced).
Validating prerequisites	O(n) searches. Each search O(log n) average, O(n log n) total
Overall complexity	O(n log n)

Advantages and Disadvantages:

	Advantages	**Disadvantages**
Vector	● Simple insertion at the end ● Easy to iterate repeatedly ● Searching for a course is O(n) in the worst case	● O(n) search for a course because it uses linear search ● Printing all courses in sorted order requires sorting O(n log n) ● Sorting is O(n log n) if needed frequently
Hash Table	● O(1) average insertion and search by courseNumber ● Checking prerequisites is very efficient	● Printing courses in sorted order requires extracting all keys and sorting them, O(n log n) ● Worst-case performance can degrade to O(n) if the hash table is poorly managed or if collisions occur
Binary Search Tree	● In-order traversal automatically gives sorted order O(n) ● Searching for a specific course is O(log n)	● Insertions are O(log n) each, total O(n log n) to build ● If unbalanced, performance can degrade to O(n^2)

Recommendation:

If the academic advisors in the Computer Science department at ABCU need to print the entire sorted list, a Binary Search Tree allows sorted output in O(n) time without additional sorting after the initial build. Both the vector and hash table would require sorting O(n log n) each time. The hash table is preferable for quick searches and the vector is the most simple, but would be the slowest for repeated searches.

Since both printing a sorted list and searching for individual courses are important, a Binary Search Tree makes for a good compromise: O(log n) searches and O(n) sorted print. The initial load cost of O(n log n) is worth the benefits. Therefore, I recommend a BST data structure since it provides efficient searching and easy sorted output without needing to re-sort the data each time.