Haley Candia Perez

February 12, 2026

CS-300

Project One

Pseudocode:

```
STRUCT Course
        courseID
        courseName
        preList (array of courseIDs)

        CONSTRUCTOR Course()
                SET courseID = ""
                SET courseName = ""
                SET preList = empty array
END STRUCT

CLASS BinaryTree

        STRUCT Node
                Course data
                Node left
```

```
        Node right
    END STRUCT

    Node root

    CONSTRUCTOR BinaryTree()
        SET root = NULL
    END CONSTRUCTOR
END CLASS

FUNCTION InsertCourse(tree, Course cf)

    CREATE newNode WITH c
    SET newNode.left = NULL
    SET newNode.right = NULL

    IF tree.root IS NULL THEN
        SET tree.root = newNode
        RETURN
    END IF

    SET currentNode = tree.root
```

```
        WHILE TRUE
            IF c.courseID < currentNode.data.courseID THEN
                IF currentNode.left IS NULL THEN
                    SET currentNode.left = newNode
                    BREAK
                ELSE
                    SET currentNode = currentNode.left
                END IF
            ELSE
                IF currentNode.right IS NULL THEN
                    SET currentNode.right = newNode
                    BREAK
                ELSE
                    SET currentNode = currentNode.right
                END IF
            END IF
        END WHILE
END FUNCTION

FUNCTION LoadCoursesFromFile (filePath, tree)

        OPEN filePath FOR reading
```

```
IF file cannot be opened THEN
        DISPLAY “Error: File cannot be opened.”
        RETURN FALSE
END IF

WHILE NOT end of file
        READ line
        TRIM whitespace

        IF line IS empty THEN
                CONTINUE
        END IF

        SPLIT line by comma INTO tokens

        IF LENGTH(tokens) < 2 THEN
                DISPLAY “Error: Invalid line format.”
                CONTINUE
        END IF

        CREATE Course newCourse
        SET newCourse.courseID = tokens[0]
```

```
            SET newCourse.courseName = tokens[1]

            FOR i FROM 2 TO LENGTH(tokens) - 1
                IF tokens[i] IS NOT empty THEN
                    ADD tokens[i] TO newCourse.preList
                END IF
            END FOR

            CALL InsertCourse (tree, newCourse)

        END WHILE
        CLOSE file
        RETURN TRUE
END FUNCTION

FUNCTION ValidateCourses(tree)

        RETURN ValidateNode (tree.root, tree)

END FUNCTION

FUNCTION ValidateNode (node, tree)
```

```
    IF node IS NULL THEN
        RETURN TRUE
    END IF

    FOR EACH prerequisite IN node.data.preList
        IF SearchCourse (tree, prerequisite) IS empty THEN
            RETURN FALSE
        END IF
    END FOR

    IF ValidateNode (node.left, tree) IS FALSE THEN
        RETURN FALSE
    END IF

    IF ValidateNode (node.right, tree) IS FALSE THEN
        RETURN FALSE
    END IF

    RETURN TRUE
END FUNCTION

FUNCTION SearchCourse (tree, courseID)
```

```
    SET currentNode = tree.root

    WHILE currentNode IS NOT NULL

        IF currentNode.data.courseID == courseID THEN
            RETURN currentNode.data
        ELSE IF  courseID < currentNode.data.courseID THEN
            SET currentNode = currentNode.left
        ELSE
            SET currentNode = currentNode.right
        END IF
    END WHILE

    RETURN empty Course object
END FUNCTION

FUNCTION PrintCourseInfo (tree, courseID)

    SET course = SearchCourse (tree, courseID)

    IF course IS empty THEN
        DISPLAY "Course not found."
```

```
            RETURN
        END IF

        DISPLAY “Course ID: ” + course.courseID
        DISPLAY “Course Name: ” + course.courseName

        IF course.preList IS empty THEN
            DISPLAY “Prerequisites: None”
        ELSE
            DISPLAY “Prerequisites:”
            FOR EACH prerequisite IN course.preList
                DISPLAY prerequisite
            END FOR
        END IF
END FUNCTION

FUNCTION PrintAllCourses (node)

        IF node IS NOT NULL THEN
            CALL PrintAllCourses (node.left)
            DISPLAY node.data.courseID + “, ” + node.data.courseName
            CALL PrintAllCourses ( node.right)
```

```
        END IF
END FUNCTION
//called as CALL PrintAllCourses(tree.root)

FUNCTION Main

        DECLARE tree AS BinaryTree
        DECLARE dataLoaded AS BOOLEAN = FALSE
        DECLARE choice

        WHILE choice != 9

                DISPLAY "======================"
                DISPLAY "ABCU Computer Science Courses"
                DISPLAY "1. Load Data"
                DISPLAY "2. Print Course List"
                DISPLAY "3. Print Course Information"
                DISPLAY "9. Exit"
                DISPLAY "======================"

                READ choice
```

```
IF choice == 1 THEN
        PROMPT “Enter file path: ”
        READ filePath

        IF LoadCoursesFromFile (filePath, tree) == TRUE
                IF ValidateCourses (tree) == TRUE THEN
                        SET dataLoaded = TRUE
                        DISPLAY “Data loaded successfully.”
                ELSE
                        DISPLAY “Error: Invalid prerequisite found.”
                END IF
        END IF

ELSE IF choice == 2 THEN
        IF dataLoaded == FALSE THEN
                DISPLAY “Error: Load data first.”
        ELSE
                CALL PrintAllCourses (tree.root)
        END IF

ELSE IF choice == 3 THEN
        IF dataLoaded == FALSE THEN
```

```
                DISPLAY “Error: Load data first.”
            ELSE
                PROMPT “Enter course ID: ”
                READ courseID
                CALL PrintCourseInfo (tree, courseID)
            END IF
        ELSE IF choice == 9 THEN
            DISPLAY “Exiting program.”
        ELSE
            DISPLAY “Invalid selection.”
        END IF
    END WHILE
END FUNCTION
```

## Vector

| Operation | Total Cost | Notes |
|---|---|---|
| Open file and read lines | n + 1 | 1 for opening file + n for each line |
| Create Course objects | n | 1 per course |
| Assign course data | n | courseID, courseName, prerequisites |
| Add to vector | n | 1 per course |

Total Cost: 4n + 1

Worst-case Runtime: O(n)

## Hash Table

| Operation | Total Cost | Notes |
|---|---|---|
| Open file and read lines | n + 1 | 1 for opening file + n for each line |
| Create Course objects | n | 1 per course |
| Assign course data | n | courseID, courseName, prerequisites |
| Compute hash key & insert | n | 1 per course (depends on collisions) |

Total Cost: 4n + 1

Worst-case Runtime: O(n) if many collisions, O(1) average

## Binary Search Tree

| Operation | Total Cost | Notes |
|---|---|---|
| Open file and read lines | n + 1 | 1 for opening file + n for each line |
| Create Course objects | n | 1 per course |
| Assign course data | n | courseID, courseName, prerequisites |
| Insert into BST | n^2 worst-case | O(n log n) if balanced, O(n^2) if unbalanced |

Total Cost: n^2 + 3n + 1 (worst-case, unbalanced)

Worst-case Runtime: O(n^2) worst-case, O(n log n) average

**Analysis of Advantages and Disadvantages**

When evaluating the three data structures (vector, hash table, and binary search tree) for the ABCU Computer Science course program, I found that each has distinct advantages and disadvantages.

A vector is easy to use and simple to understand, but finding a course in it can take longer because you have to check each course one by one. Sorting the courses in order also takes extra steps, which makes it slower when working with bigger lists.

A hash table is great for quickly finding a course because it can usually find it right away. However, has tables don't keep the courses in order, so if you want to print them alphabetically, it takes more work. They also use a bit more memory.

A binary search tree is a good balance. It lets you quickly find a course and, at the same time, keeps the courses in order so printing them alphabetically is easy. The only downside is that if the tree becomes unbalanced, searches can be slower.

**Recommendation**

Considering that the advisors want to see all courses in order and find course details, I would say that the binary search tree would be the best choice. It handles both tasks well, and it keeps everything organized without needing extra steps.