CS 300 - Project 1

Conor Steward

conor.steward@snhu.edu

10/7/24

Operation	Vector	Hash Table	Binary Search Tree
Load Courses	**O(n * m)**	**O(n^2) (with collisions)**	**O(n^2 + n * m) (unbalanced)**
Search Course	**O(n + m)**	**O(n + m) (with collisions)**	**O(n + m) (unbalanced)**
Print Alphanumeric List	**O(n log n) (for sorting)**	**O(n log n) (sorting after retrieval)**	**O(n) (in-order traversal)**

Pseudocode for the Menu:

```
START
    // Initialize variables for the selected data structure
    DECLARE 'selectedDataStructure' as STRING
    DECLARE 'fileLoaded' as FALSE
    // Display the menu and get the user’s choice
    FUNCTION displayMenu()
        PRINT "Welcome to the ABCU Course Advising Program"
        PRINT "1. Load file data into the data structure"
        PRINT "2. Print alphanumerically ordered list of all courses"
        PRINT "3. Print course title and prerequisites for an individual course"
        PRINT "9. Exit"
```

```
    PROMPT user for their choice
    RETURN user’s choice
END FUNCTION
// Main program loop
DO
    SET 'choice' to displayMenu()
    IF 'choice' equals 1
        CALL loadFileData()
    ELSE IF 'choice' equals 2
        IF 'fileLoaded' is TRUE
            CALL printCoursesSorted()
        ELSE
            PRINT "Error: You must load the file data first."
    ELSE IF 'choice' equals 3
        IF 'fileLoaded' is TRUE
            CALL printCourseDetails()
        ELSE
            PRINT "Error: You must load the file data first."
    ELSE IF 'choice' equals 9
        PRINT "Exiting the program."
        EXIT
    ELSE
        PRINT "Invalid option. Please choose a valid option."
```

```
        END IF
    WHILE 'choice' is not 9
END
```

Pseudocode for Sorting Courses in a Vector:

```
START
    // Step 1: Define a Course structure
    DEFINE 'Course' struct with the following fields:
        - courseNumber: STRING
        - courseName: STRING
        - prerequisites: LIST of STRING
    // Step 2: Declare a vector to store Course objects
    DECLARE 'courseVector' to store Course objects
    // Step 3: Parse each line from the file and create Course objects
    FOR each line in 'courseLines'
        SPLIT the line by commas into 'tokens'
        // Step 4: Create a new Course object
        CREATE 'course'
        SET 'course.courseNumber' to tokens[0]
        SET 'course.courseName' to tokens[1]
         // Step 5: Parse and add prerequisites to the course
        IF tokens.length > 2
            FOR each token from the third element onward
```

```
            ADD token to 'course.prerequisites'
        // Step 6: Add the course to the vector
        ADD 'course' to 'courseVector'
    PRINT "All courses have been loaded into the vector."
END
```

Pseudocode for Searching and Printing Course Information from a Vector:

```
START
    FUNCTION printCourseInfo(courseVector, searchCourseNumber)
        DECLARE 'found' as FALSE
        FOR each 'course' in 'courseVector'
            IF 'course.courseNumber' equals 'searchCourseNumber'
                PRINT "Course Number: " + course.courseNumber
                PRINT "Course Name: " + course.courseName
                IF 'course.prerequisites' is NOT empty
                    PRINT "Prerequisites: "
                    FOR each 'prerequisite' in 'course.prerequisites'
                        PRINT prerequisite
                    END FOR
                ELSE
                    PRINT "Prerequisites: None"
                END IF
                SET 'found' to TRUE
```

```
            BREAK
        END IF
    END FOR

    IF 'found' is FALSE
        PRINT "Course not found."
    END IF
  END FUNCTION
END
```

Pseudocode for Sorting Courses in a Hash Table:

```
START
  // Step 1: Define a Course structure
  DEFINE 'Course' struct with the following fields:
    - courseNumber: STRING
    - courseName: STRING
    - prerequisites: LIST of STRING
  // Step 2: Declare a hash table to store courses
  DECLARE 'courseHashTable' with hash function
  // Step 3: Parse each line from the file and store it as a Course object
  FOR each line in 'courseLines'
    SPLIT the line by commas into 'tokens'
    // Step 4: Create a new Course object
```

```
        CREATE 'course'
        SET 'course.courseNumber' to tokens[0]
        SET 'course.courseName' to tokens[1]
        // Step 5: Parse and add prerequisites to the course
        IF tokens.length > 2
            FOR each token from the third element onward
                ADD token to 'course.prerequisites'
                // Step 6: Insert the course into the hash table
        CALL 'courseHashTable.Insert(course.courseNumber, course)'
        PRINT "All courses have been loaded into the hash table."
END
```

Pseudocode for Sorting and Printing from a Hash Table:

```
START
    FUNCTION printCourseInfo(courseHashTable, searchCourseNumber)
        // Step 1: Search for the course in the hash table
        DECLARE 'course' and set it to 'courseHashTable.Search(searchCourseNumber)'
        // Step 2: If course is found, print its details
        IF 'course' is not NULL
            PRINT "Course Number: " + course.courseNumber
            PRINT "Course Name: " + course.courseName
            IF 'course.prerequisites' is not empty
                PRINT "Prerequisites: "
```

```
            FOR each 'prerequisite' in 'course.prerequisites'
                PRINT prerequisite
            ELSE
                PRINT "Prerequisites: None"
        ELSE
            PRINT "Course not found.”
    END FUNCTION
END
```

Pseudocode for Sorting Courses in a Binary Tree:

```
START
    // Step 1: Define a Course structure
    DEFINE 'Course' struct with the following fields:
        - courseNumber: STRING
        - courseName: STRING
        - prerequisites: LIST of STRING
    // Step 2: Define a Binary Search Tree (BST) Node
    DEFINE 'BSTNode' with the following fields:
        - 'course': Course
        - 'left': BSTNode
        - 'right': BSTNode
    // Step 3: Declare the root of the binary search tree
    DECLARE 'root' as NULL
```

```
    // Step 4: Function to insert a course into the binary search tree
    FUNCTION insertBST(root, course)
        IF root is NULL
            CREATE a new 'BSTNode' with 'course' and set it as the root
        ELSE IF course.courseNumber < root.course.courseNumber
            CALL insertBST(root.left, course)
        ELSE
            CALL insertBST(root.right, course)
    END FUNCTION
    // Step 5: Parse each line from the file and insert into the BST
    FOR each line in 'courseLines'
        SPLIT the line by commas into 'tokens'
        CREATE 'course' object
        SET 'course.courseNumber' to tokens[0]
        SET 'course.courseName' to tokens[1]
        IF tokens.length > 2
            FOR each token from the third element onward
                ADD token to 'course.prerequisites’
        // Insert the course into the binary search tree
        CALL insertBST(root, course)
       PRINT "All courses have been added to the binary search tree."
END
```

Pseudocode for Searching and Printing from a Binary Tree:

```
START
    FUNCTION searchBST(root, searchCourseNumber)
        IF root is NULL
            PRINT "Course not found."
        ELSE IF searchCourseNumber equals root.course.courseNumber
            PRINT "Course Number: " + root.course.courseNumber
            PRINT "Course Name: " + root.course.courseName
            IF root.course.prerequisites is NOT empty
                PRINT "Prerequisites: "
                FOR each prerequisite in root.course.prerequisites
                    PRINT prerequisite
            ELSE
                PRINT "Prerequisites: None"
        ELSE IF searchCourseNumber < root.course.courseNumber
            CALL searchBST(root.left, searchCourseNumber)
        ELSE
            CALL searchBST(root.right, searchCourseNumber)
    END FUNCTION
    // Call the search function
    CALL searchBST(root, userInput)
END
```

Vector Pseudocode for Loading, Searching, and Printing in Alphanumeric Order:

```
FUNCTION loadFileIntoVector()
    DECLARE 'courseVector' as an empty vector
    FOR each line in the file
        SPLIT the line by commas into 'tokens'
        CREATE a new 'Course' object
        SET 'course.courseNumber' to tokens[0]
        SET 'course.courseName' to tokens[1]
        IF tokens.length > 2
            ADD the remaining tokens as prerequisites
        ADD 'course' to 'courseVector'
    END FOR
END FUNCTION

FUNCTION printVectorSorted()
    SORT 'courseVector' by 'course.courseNumber' in ascending order
    FOR each 'course' in 'courseVector'
        PRINT "Course Number: " + course.courseNumber
        PRINT "Course Name: " + course.courseName
    END FOR
END FUNCTION

FUNCTION searchCourseInVector(searchCourseNumber)
```

```
    FOR each 'course' in 'courseVector'
        IF 'course.courseNumber' equals 'searchCourseNumber'
            PRINT "Course Number: " + course.courseNumber
            PRINT "Course Name: " + course.courseName
            IF 'course.prerequisites' is NOT empty
                PRINT "Prerequisites: " + course.prerequisites
            ELSE
                PRINT "No prerequisites"
            END IF
            RETURN
    PRINT "Course not found."
END FUNCTION
```

Hash Table Pseudocode for Loading, Searching, and Printing in Alphanumeric Order:

```
FUNCTION loadFileIntoHashTable()
    DECLARE 'courseHashTable' as an empty hash table
    FOR each line in the file
        SPLIT the line by commas into 'tokens'
        CREATE a new 'Course' object
        SET 'course.courseNumber' to tokens[0]
        SET 'course.courseName' to tokens[1]
        IF tokens.length > 2
```

```
            ADD the remaining tokens as prerequisites
        CALL 'courseHashTable.Insert(course.courseNumber, course)'
    END FOR
END FUNCTION

FUNCTION printHashTableSorted()
    RETRIEVE all course objects from 'courseHashTable'
    SORT courses by 'course.courseNumber' in ascending order
    FOR each 'course' in the sorted list
        PRINT "Course Number: " + course.courseNumber
        PRINT "Course Name: " + course.courseName
    END FOR
END FUNCTION

FUNCTION searchCourseInHashTable(searchCourseNumber)
    SET 'course' to 'courseHashTable.Search(searchCourseNumber)'
    IF 'course' is NOT NULL
        PRINT "Course Number: " + course.courseNumber
        PRINT "Course Name: " + course.courseName
        IF 'course.prerequisites' is NOT empty
            PRINT "Prerequisites: " + course.prerequisites
        ELSE
            PRINT "No prerequisites"
```

```
    ELSE
        PRINT "Course not found."
END FUNCTION
```

Binary Tree Pseudocode for Loading, Searching, and Printing in Alphanumeric Order:

```
FUNCTION loadFileIntoBinarySearchTree()
    DECLARE 'root' as NULL
    FOR each line in the file
        SPLIT the line by commas into 'tokens'
        CREATE a new 'Course' object
        SET 'course.courseNumber' to tokens[0]
        SET 'course.courseName' to tokens[1]
        IF tokens.length > 2
            ADD the remaining tokens as prerequisites
        CALL insertBST(root, course)
    END FOR
END FUNCTION

FUNCTION printBinarySearchTreeInOrder(root)
    IF root is NOT NULL
        CALL printBinarySearchTreeInOrder(root.left)
        PRINT "Course Number: " + root.course.courseNumber
        PRINT "Course Name: " + root.course.courseName
```

```
    CALL printBinarySearchTreeInOrder(root.right)
END FUNCTION

FUNCTION searchCourseInBinarySearchTree(searchCourseNumber)
   CALL searchBST(root, searchCourseNumber)
END FUNCTION
```

Conducting a runtime analysis of sorting and searching courses with differing data structures ensures accurate recommendations to shareholders. The runtime analysis evaluates the worst-case running time for reading the file, creating course objects, and storing them in the data structures which in this case are Vector, Hash Table, and Binary Search Tree. The Big O notation is used to represent the complexity of the operations. Assuming there are n'' courses in the input file, the following are predictions of algorithm complexity and the pseudocode for evaluating each within code.

Vector Implementation: Loading Courses into the Vector

For the worst-case time complexity each line involves constant time operations (O(1)), except for iterating through prerequisites, which can be at most 'm'. In the worst case , if all courses have prerequisites, looping through 'n' courses takes O(n * m). This is the total time complexity.

```
FOR each line in 'courseLines'  // Executes n times
    SPLIT the line by commas into 'tokens'  // O(1) per line; total O(n)
    CREATE 'course' object  // O(1) per line; total O(n)
    SET 'course.courseNumber' to tokens[0]  // O(1)
    SET 'course.courseName' to tokens[1]  // O(1)
    IF tokens.length > 2
   FOR each token from the third element onward  // If there are m prerequisites, worst case O(m)
          ADD token to 'course.prerequisites'  // O(1) for each token
    ADD 'course' to 'courseVector'  // O(1)
```

Searching and Printing Course Information from Vector:

Considering worst-case time complexity, all ‘n’ courses are checked. For each course, there are ‘m’ prerequisites to print. This results a total complexity: O(n + m).

```
FOR each 'course' in 'courseVector'  // O(n)
    IF 'course.courseNumber' equals 'searchCourseNumber'  // O(1)
        PRINT course information  // O(1)
        FOR each 'prerequisite' in 'course.prerequisites'  // O(m)
            PRINT prerequisite  // O(1)
        BREAK
```

Loading Courses into the Hash Table:

Considering the worst-case time complexity, for each course O(1) operations are the most common except for iterating over the prerequisites. Using hash table insert, O(1) on average, but in the worst case, it can be O(n) with collisions. This makes the total complexity average O(n * m), but in the worst case with collisions, $O(n^2)$. This reliance on a lack of collisions for efficiency should take the developers skills in consideration before choosing this as an efficient option.

```
FOR each line in 'courseLines'  // Executes n times
    SPLIT the line by commas into 'tokens'  // O(1)
    CREATE 'course' object  // O(1)
    SET 'course.courseNumber' to tokens[0]  // O(1)
    SET 'course.courseName' to tokens[1]  // O(1)

    IF tokens.length > 2
        FOR each token from the third element onward  // O(m)
            ADD token to 'course.prerequisites'  // O(1)
    CALL 'courseHashTable.Insert(course.courseNumber, course)'  // O(1)
```

Searching and Printing Course Information from Hash Table:

Considering the worst-case time complexity, searching the hash table takes O(1) on average, but in the worst case, with collisions, it can take O(n). This makes the total complexity, on average O(m), but worst case O(n + m). Like with loading courses into the hash table, the skills of the developer should be taken into consideration.

```
DECLARE 'course' and set it to 'courseHashTable.Search(searchCourseNumber)'  // O(1)
average, O(n) worst case
IF 'course' is found
    PRINT course information  // O(1)
    FOR each 'prerequisite' in 'course.prerequisites'  // O(m)
        PRINT prerequisite  // O(1)
```

Loading Courses into the Binary Search Tree:

Considering worst-case time complexity each course Inserted into a BST takes O(log n) if balanced, but O(n) if the tree becomes skewed. Looping through prerequisites takes O(m). This means the total complexity is $O(n * \log n + n * m)$ in a balanced tree, but $O(n^2 + n * m)$ in the worst case.

```
FOR each line in 'courseLines'  // Executes n times
    SPLIT the line by commas into 'tokens'  // O(1)
    CREATE 'course' object  // O(1)
    SET 'course.courseNumber' to tokens[0]  // O(1)
    SET 'course.courseName' to tokens[1]  // O(1)
     IF tokens.length > 2
        FOR each token from the third element onward  // O(m)
            ADD token to 'course.prerequisites'  // O(1)
    CALL insertBST(root, course)  // O(log n) in a balanced tree, O(n) in an unbalanced tree
```

Searching and Printing Course Information from BST

Considering the worst-case time complexity, searching the tree takes O(log n) if balanced, O(n) if unbalanced. Printing prerequisites takes O(m). This means the total complexity is O(log n + m) in a balanced tree, O(n + m) in an unbalanced tree.

```
CALL searchBST(root, searchCourseNumber)  // O(log n) in a balanced tree, O(n) in an unbalanced tree
IF course is found
    PRINT course information  // O(1)
    FOR each 'prerequisite' in root.course.prerequisites  // O(m)
        PRINT prerequisite  // O(1)
```

When analyzing Vector, Hash Table, and Binary Search Tree for managing and searching course information, each data structure offers particular advantages and disadvantages in terms of time complexity.

Vectors are straightforward to use by comparison to binary search trees and hash tables. Since vectors store data sequentially they are compact in memory with no overhead from pointers or additional structures like in trees or hash tables. If the goal is to iterate over the entire list or process elements sequentially, vectors are optimal because all elements are stored contiguously. Sorting a vector can be easily achieved and has a time complexity of O(n log n).

Searching for an individual course by course number requires a linear scan with a complexity of O(n), which is inefficient for large datasets compared to structures like hash tables or binary search trees. Also, a vector does not maintain any inherent ordering unless explicitly sorted, so searching or inserting an item in a specific order requires extra work. While inserting a course is efficient, if you need to maintain a sorted list, insertion can become slow as it requires shifting elements O(n).

Vectors are ideal for scenarios where data needs to be accessed sequentially or where memory efficiency is key. However, they are not well-suited for fast searching or frequent insertions/deletions in large datasets.

Fast insertion and search using hash tables offer constant time complexity O(1) for inserting and searching in the average case, making them the most efficient structure for lookups when there are no hash collisions. Since the data is accessed via a hashed key, there's no need to sort or maintain an ordered list for lookups.

In the event of hash collisions, in the case that multiple courses are hashed to the same index, searching or inserting could degrade to O(n), which is as bad as a vector. Hash tables do

not maintain any natural order, which makes them unsuitable for tasks that require sorted data. A poor choice of hash function could lead to clustering or high collision rates, which would degrade performance to that of a vector. Hash tables are highly effective for fast lookups and insertions when the key is frequently accessed. However, their lack of inherent order and possible memory overhead might make them less suitable for systems that require sorting or memory-efficient solutions.

Binary search trees(BTS) automatically maintain an ordered structure, making it easy to retrieve data in sorted order via in-order traversal, which can be done in O(n) time. In a balanced BST, search and insertion operations are performed in logarithmic time O(log n), which is efficient for both small and large datasets. BSTs are useful for querying a range of values since in-order traversal will retrieve the elements in sorted order.

If the tree becomes unbalanced, the time complexity for search, insertion, and deletion degrades to O(n). Self-balancing BSTs ensure logarithmic operations, but they introduce additional complexity and overhead in maintaining the balanced property. Each node in a BST contains pointers to child nodes, which leads to higher memory usage compared to arrays or vectors.

BSTs are great for scenarios where sorted order and efficient lookups are important. However, they require careful balancing to avoid performance degradation.

I recommend using the Hash Table for the course management system. Hash Table provides the fastest time complexity for both search and insertion operations, with O(1) time for both. This is critical when managing a large set of courses where lookups for course information will be frequent. In contrast, Vector requires O(n) time for search and could be inefficient as the

number of courses grows. Similarly, a Binary Search Tree only offers O(log n) time in the best-case or balanced scenario, and O(n) in the worst case (unbalanced tree). Although, Hash Table scales well with larger datasets while Vector would slow down as the number of courses grows, and Binary Search Tree might become unbalanced without careful management, the hash table continues to perform efficiently. Since courses are identified by a unique course number, hash functions can distribute the course numbers uniformly across the table, minimizing collisions. Since the primary focus of the system is on fast course lookup and retrieval, where sorting is not necessarily required, the Hash Table allows direct access to course information based on a unique key without the need for maintaining a sorted order. If the system occasionally requires sorted output, sorting can be done as a secondary operation. However, the primary requirement is quick lookups and updates, which the Hash Table handles best. While Hash Tables require some extra memory for handling collisions the trade-off is the performance gains on lookups and insertions. Vector is space-efficient but fails in lookup speed, while Binary Search Tree requires additional pointers, leading to increased memory usage compared to hash tables. By carefully choosing a good hash function, collisions in the Hash Table can be minimized. Even in cases where collisions occur, the hash table's performance will still be competitive, with operations degrading to O(n) only in rare cases. Given the course data structure hash collisions are unlikely to be frequent.