Joseph Cassello Jr.
Professor Gopalakrishnan
CS 300 - Project One
December 8, 2024

```
# LoadCourses Function (Reads file and processes data)
FunctionLoadCourses(filePath):
    Open the file at filePath for reading
    If the file cannot be opened:
        Print “Error: Unable to open file.”
        Exit function

    Initialize an empty vector, hash table, or BST depending on the chosen data structure

    For each line in the file:
        Remove leading and trailing whitespace from the line
        Split the line into tokens using a comma or space
        If the number of tokens < 2:
            Print “Error: Invalid line format. Must have at least course number and title.”
            Skip to the next line
        Assign the first token to courseNumber
        Assign the second token to courseTitle
        Assign any remaining tokens to prerequisites (can be empty)

        For each prerequisites in prerequisites:
            If prerequisite does not exist in the data structure:
                Print “Error: Prerequisite {prerequisite} does not exist in file.”
                Skip to the next line
```

```
            Create a new course object using courseNumber, courseTitle, and prerequisites
            Insert the course object into the data structure (Vector, hash Table, or BST)

Close file
Return dataStructure

# Course Object and Insertion
Class Course:
    Attributes:
        courseNumber (string)
        courseTitle (string)
        Prerequisites (List of Strings)

    Constructor(courseNumber, courseTitle, prerequisites):
        Set courseNumber = courseNumber
        Set courseTitle = courseTitle
        Set prerequisites = prerequisites

# For Vector
Function to InsertCourseVector(vector, course):
    Append the course object to the vector

# For Hash Table
Function InsertCourseHashTable(hashTable, course):
    Insert the course object into the hash table using courseNumber as the key

# For Binary Search Tree (BST)
Function InsertCourseBST(BST, course):
    If BST is empty:
        Set course as the root node
```

Else:

Recursively compare course.courseNumber with the current node's courseNumber

If course.courseNumber < current node's courseNumber:

Insert into the left subtree

Else:

Insert into the right subtree

PrintCourseList Function (Prints all courses in alphanumeric order)

Function PrintCourseList(dataStructure):

If the data structure is not empty:

If using Vector:

Sort the vector by courseNumber

For each course in Vector:

Print course.courseNumber + ": " + course.courseTitle

If course.prerequisites is empty:

Print "Prerequisites: None"

If using Hash Table:

Get the list of all courses from the hash table

Sort by courseNumber

For each course in sorted list:

Print course.courseNumber + ": " + course.courseTitle

If course.prerequisites is empty:

Print "Prerequisites: None"

If using BST:

Perform an in-order traversal of the BST

For each course:

Print course.courseNumber + ": " + course.courseTitle

If course.prerequisites is empty

Print “Prerequisites: None”

#PrintCourseDetails Function (Print details for a specific course)

Function PrintCourseDetails(dataStructure, courseNumber):

    If the course is found in the data structure:

        Print course.courseNumber + “: “ + course.courseTitle

        If course.prerequisites is not empty:

            Print “Prerequisites: “ + list of prerequisites

        Else:

            Print “Prerequisites: None”

    Else:

        Print “Error: Course not found.”

Main Program (Menu and Operations)

Main():

    Prompt user to input the file path of the course data

    Call LoadCourses(filePath) to load and validate courses into the chosen data structure
    If loading is successful:

        Print “Courses loaded successfully.”

    While True:

        Display menu options:

            1. Load file data

            2. Print all courses

            3. Print details of a specific course

            9. Exit

        Get user input for choice

        If choice == 1:

            Prompt user for the file path

            Call LoadCourses(filePath) to reload and validate data

        Else if choice == 2:

```
        Call PrintCourseList(dataStructure)
Else if choice == 3:
        Prompt user to input courseNumber
        Call PrintCourseDetails(dataStructure, courseNumber)
Else if choice == 9:
        Print “Exiting program.”
        Break
Else:
        Print “Invalid choice. Please try again.”
```

Runtime Analysis:

Operation	Vector	Hash Table	Binary Search Tree
Reading file	O(n)	O(n)	O(n)
Parsing each line	O(1) per line	O(1) per line	O(1) per line
Creating Course Objects	O(1) per line	O(1) per line	O(1) per line
Inserting into Data Structure	O(1) per insert	O(1) average-case	O(log n) per insert
Memory Usage	O(1) per course	O(1) per course	O(1) per course
Total for n courses	O(n)	O(n)	O(n log n)

Data Structure Evaluation:

Vector:

A vector is a dynamic array, which is great for cases where you frequently add new elements. It provides quick access to elements by index, with an O(1) time complexity. However, searching for a specific element or sorting the vector can be costly. If you are searching by value, it will take O(n) time, as a linear search is needed. Sorting a vector takes O(n log n) time, and while sorting is not required all the time, it can become a bottleneck as the list grows, especially if you are frequently sorting to display courses in order. For the advising program, using a vector would require linear search to find individual courses and frequent sorting to display them, which could slow things down as the course list expands.

Hash Table:

Hash tables are known for their efficiency in looking up, inserting, and deleting elements, all with an average time complexity of O(1). This makes them an excellent choice for quickly

finding courses by their course number. However, one downside is that hash tables do not maintain any inherent order. Since the program requires courses to be displayed in alphanumerical order, a hash table would need extra steps to extract the data and sort it, which adds some overhead. Another drawback is that hash collisions can degrade performance, especially if the hash table is not well managed. While hash tables are great for fast lookups, the lack of ordering makes them less ideal for tasks that need sorted data.

Binary Search Tree:

A Binary Search Tree (BST) is designed to keep data sorted. With balanced trees, searching, inserting, and deleting operations run in O(log n) time, making them very efficient for handling large datasets. The BST naturally supports ordered data, which means courses can be printed in alphanumerical order with an in order traversal, which is exactly what is needed for this program. However, a BST can lose its efficiency if it becomes unbalanced. In the worst case, an unbalanced BST can degenerate into a linked list, turning operations into O(n) time. Keeping the tree balanced requires additional effort, which can complicate the implementation. But overall, the ability to maintain sorted data makes the BST a strong choice.

Recommendation:

After considering all three data structures, I believe the Binary Search Tree is the best option for this program. The main reason is that the BST naturally keeps the data sorted, which directly addresses the need to print courses in alphanumerical order. The time complexity for searching and inserting into a balanced BST is O(log n), which is efficient even as the number of

courses grows. While maintaining balance in the tree adds some complexity, the benefits of having sorted data and efficient retrieval make it worthwhile.

Although hash tables offer fast lookups, they do not keep the data in any particular order, meaning we would have to sort the data each time before printing it, adding extra overhead. Vectors, on the other hand, would require linear searches and O(n log n) sorting time, which is not ideal for larger datasets. Overall, the BST strikes the best balance between performance and functionality, especially when considering the need for sorted course data.