Robert Heavner
Project One
2/12/2024

Pseudocode for Loading Data into Tree Data Structure

- Outline a Tree data structure to hold Course objects.
- Outline a Course class with properties: courseNumber, name, and requirements (which is a list of Course objects).

```
// Outline the Tree data structure to hold Course objects
class Course {
   String courseNumber
   String name
   List<Course> requirements.
}
class Tree {
   Node root
}
class Node {
   Course course
   Node leftChild
   Node rightChild
}
```

Opening and Reading the File

- Start by defining a function to read course data from a file.
- Open the file containing course information.
- For each line in the file:
- Parse the line to extract course information.
- Split the line by commas to get course details.
- The first item is the courseNumber, the second is the name, and the rest are requirements.
- Validate the line to ensure it has at least a courseNumber and name.
- If requirements are listed, check that each prerequisite exists as a course in the file.
- If the line is valid, proceed to create a Course object.

```
// Function to read course data from a file and load into the Tree
function loadCourseDataFromFile(filename) {
   tree = new Tree()
   file = open(filename, 'read')

   while not file.endOfFile() {
      line = file.readLine()
      courseData = line.split(',')
      courseNumber = courseData[0]
      name = courseData[1]
      requirements = courseData[2..]
```

Robert Heavner
Project One
2/12/2024

```
        // Validate the course data
        if courseNumber is empty or name is empty then
            continue // Skip invalid lines

        // Create a new Course object
        course = new Course(courseNumber, name)

        // Add requirements to the Course object
        for prerequisiteNumber in requirements {
            prerequisiteCourse = findCourseInTree(tree, prerequisiteNumber)
            if prerequisiteCourse is not null then
                course.requirements.add(prerequisiteCourse)
        }

        // Insert the Course object into the Tree
        insertCourseIntoTree(tree, course)
    }

    file.close()
    return tree
}
```

Creating Course Objects and Storing in Tree

- Initialize an empty Tree.
- For each valid line of course data:
- Create a new Course object with the courseNumber and name.
- For each prerequisite courseNumber listed:
- Find the Course object in the Tree with that courseNumber.
- Add that Course object to the current Course's requirements list.
- Insert the current Course object into the Tree.
- Continue until all lines are processed and the Tree is populated with Course objects.

```
// Function to find a course in the Tree by course number
function findCourseInTree(tree, courseNumber) {
    // Tree traversal logic to find the course
    // Returns the Course object if found, else null
}

// Function to insert a Course object into the Tree
function insertCourseIntoTree(tree, course) {
    // Logic to insert a new node with the Course into the Tree
}
```

Robert Heavner
Project One
2/12/2024

Printing Course Information and Requirements from Tree

- Outline a function to print course information given a courseNumber and a Tree.
- Search the Tree for the Course object with the matching courseNumber.
- When the Course is found:
- Print the courseNumber and name.
- If there are requirements:
- Print "Requirements: " followed by each prerequisite's courseNumber and name.
- If there are no requirements, print "No requirements."

```
// Function to print the course information from the Tree
function printCourseInformation(tree, courseNumber) {
    course = findCourseInTree(tree, courseNumber)
    if course is not null then
        print(course.courseNumber + ": " + course.name)
        if course.requirements is not empty then
            print("Requirements: ")
            for prerequisite in course.requirements {
                print(requirements.courseNumber + ": " + requirements.name)
            }
        else
            print("No requirements")
    else
        print("Course not found")
}
```

Menu for the main program

- **Select an option**
- **Load Data**
- **Print Course List ABC/123 Order**
- **Print Course**
- **Exit**

```
// Main program menu pseudocode
function mainMenu() {
    coursesTree = null
    while true {
        print("Select an option:")
        print("1. Load Data Structure")
        print("2. Print Course List ABC/123 Order")
        print("3. Print Course")
        print("4. Exit")
        option = getUserInput()
```

Robert Heavner
Project One
2/12/2024

```
        if option == 1 then
            filename = getUserInput("Enter filename: ")
            coursesTree = loadCourseDataFromFile(filename)
        else if option == 2 and coursesTree is not null then
            printAllCourses(coursesTree)
        else if option == 3 and coursesTree is not null then
            courseNumber = getUserInput("Enter course number: ")
            printCourseInformation(coursesTree, courseNumber)
        else if option == 4 then
            break
        else
            print("Invalid option or load data first.")
    }
}
```

Robert Heavner
Project One
2/12/2024

Evaluation

Evaluate the run-time and memory of data structures that could be used to address the requirements.

- Opening the file: This is a one-time operation, so its cost is O(1).
- Reading each line: Assuming there are n courses, each line of the file will be read once, so this operation is O(n).
- Parsing each line: The cost of parsing each line is constant, O(1), since splitting a string by commas is independent of the number of courses.
- Checking for formatting errors: Checking each line for the correct format is also O(1) per line.
- Creating course objects: For each line, a single course object is created, which is O(1).
- Total cost for reading and parsing: The total cost, not considering prerequisite lookups, would be O(n) since each line is read and processed in constant time.

Opening the program is something that would only be done once, no matter how much information is in the program, the program itself would only be opening once. For each course in the list the program would have to reach each line for that. For every line the program would make a course object to hold the info from that line and then in a one step process for each line, so this would be O(1) per course. The more courses the longer the timing would be, twice as much would be twice as long. O(n) where n stands as the number of courses, where the program list grows so does the timing. In production environments and programs like SQL Anywhere and JAVA there are programs call Profilers that could help determine how expensive each statement would be and how long each line is costing the program, this would be help to have when running this program to see where the inefficiencies are.

Code	Line Cost	# Times Executes	Total Cost
For all courses	1	1	1
If the course is the same as courseNumber	1	n	n
Print out the course information	1	n	n
Print the prerequisite course information	1	n	n
		Total Cost	4n+1
		Runtimes	O(n)

Explain the advantages and disadvantages of each structure in your evaluation.

Vector:

- **Advantages**: Good locality of reference, efficient memory usage when the vector is fully utilized, straightforward implementation, and direct access by index.
- **Disadvantages**: When a vector's current capacity is exceeded, scaling can become expensive and linear search becomes sluggish for huge datasets.

Robert Heavner
Project One
2/12/2024

Hash Table:

- **Advantages**: Quick inserting, removing, and access processes. It is especially effective for lookup operations, which typically take a consistent amount of time.
- **Disadvantages**: The structure of the hash table results in memory overhead, often occurring hash collisions cause poor performance, and the order of elements is not guaranteed.

Tree:

- **Advantages**: Keeps things in order, makes efficient in-order traversal possible, and, if the tree is balanced, can perform operations more quickly than a vector for big datasets.
- **Disadvantages**: Performance depends on tree balance; unbalanced trees can have poor performance, and additional memory overhead for storing node pointers.

Make a recommendation for which data structure you will plan to use in your code.

Using an ABCU program with a balanced tree data structure is what I advise. For this application, trees offer a number of benefits. First, they keep the courses organized, which is useful when displaying an alphabetical list or verifying prerequisites. In huge datasets, the linear time operations of a vector are often slower than the efficient O(log n) time operations for insertion and search in a balanced tree. Despite being more difficult to implement than vectors, trees are an attractive option for the program because of their superior search and insertion performance as well as their capacity to traverse courses in sorted order. Balanced tree avoids the potential performance issues of unbalanced trees, ensuring that the operations remain efficient even as the number of courses grows, making it well-suited for handling the extensive course offerings similar to those at SNHU.