6-2 PROJECT ONE

Christopher Davidson | DSA: Analysis and Design | 2/16/25

Efficient management and retrieval of course information requires selecting a data structure that optimizes insertion, retrieval, and sorting operations. This evaluation compares three data structures: vector, hash table, and binary search tree (BST), based on their ability to efficiently store and access course data. The system must print all computer science courses in alphanumeric order and retrieve details for a specific course, including its title and prerequisites. To accomplish this, the following pseudocode outlines the steps for loading course data from a file, storing it efficiently, searching for specific courses, and presenting the information in an ordered format. A runtime complexity analysis compares the efficiency of each data structure, highlighting their strengths and weaknesses to determine the most effective option for course retrieval and organization.

1. File Input and Validation
    1.1. Function: loadCoursesFromFile(filePath)
        1.1.1. Open the file at filePath.
        1.1.2. If the file does not exist, print "Error: File not found." and return.
        1.1.3. Initialize data structures:
            1.1.3.1. Create an empty data structure (vector, hash table, or BST).
            1.1.3.2. Create an empty set validCourseNumbers to track course IDs.
        1.1.4. First Pass: Validate Course Data Format
            1.1.4.1. Loop through each line:
                1.1.4.1.1. Split the line by commas into tokens.
                1.1.4.1.2. If less than two tokens, print "Invalid format" and skip the line.
                1.1.4.1.3. Store the course number in validCourseNumbers.
        1.1.5. Second Pass: Validate Prerequisites

1.1.5.1. Rewind file to beginning.

1.1.5.2. Loop through each line:

1.1.5.2.1. Split the line into tokens.

1.1.5.2.2. For each prerequisite token beyond the second token:

1.1.5.2.2.1. If prerequisite is not in validCourseNumbers, print "Invalid prerequisite: [prerequisite]".

1.1.6. Return validated data.

2. Course Object Creation and Storage

2.1. Struct: Course

2.1.1. courseNumber (string) – Unique identifier for the course.

2.1.2. name (string) – Course title.

2.1.3. prerequisites (list of strings) – Stores prerequisites.

2.2. Function: createCourseObjects(filePath)

2.2.1. Initialize the selected data structure.

2.2.2. Open the file at filePath.

2.2.3. Loop through each line:

2.2.3.1. Split the line into tokens.

2.2.3.2. Create a new Course object:

2.2.3.2.1. Assign courseNumber and name.

2.2.3.2.2. Store additional tokens as prerequisites.

2.2.3.3. Insert Course into the selected data structure.

2.2.4. Return populated data structure.

3. Insert Course into Data Structure

3.1. Function: insertCourse(dataStructure, course)

3.1.1. If the data structure is empty, set the root (or first element) to course.

3.1.2. Else, insert recursively:

3.1.3. If courseNumber is less than the current node, move left:

3.1.3.1. If left is empty, insert course.

3.1.3.2. Otherwise, recurse left.

3.1.4. If courseNumber is greater, move right:

3.1.4.1. If right is empty, insert course.

3.1.4.2. Otherwise, recurse right.

4. Search for a Course by ID

4.1. Function: searchCourse(dataStructure, courseNumber)

4.1.1. Start at the root (for BST) or index (for vector/hash table).

4.1.2. While the current node is not null:

4.1.2.1. If courseNumber matches, return course.

4.1.2.2. If courseNumber is smaller, move left.

4.1.2.3. Otherwise, move right.

4.1.3. If not found, print "Course not found.".

5. Remove a Course

5.1. Function: removeCourse(dataStructure, courseNumber)

5.1.1. Find the node containing courseNumber.

5.1.1.1. If found:

5.1.1.1.1. Case 1: No children – Delete node.

5.1.1.1.2. Case 2: One child – Replace node with child.

5.1.1.1.3. Case 3: Two children – Find minimum value in right subtree, replace, then delete.

5.1.2. Ensure data structure remains valid.

6. Print Course Information

6.1. Function: printCourseInformation(dataStructure, courseNumber)

6.1.1. If data structure is empty, print "No courses available." and return.

6.1.2. Search for courseNumber using searchCourse(dataStructure, courseNumber).

6.1.3. If not found, print "Course not found." and return.

6.1.4. Print course details:

6.1.4.1. "Course Number: " + course.courseNumber

6.1.4.2. "Title: " + course.name"

6.1.4.3. "Prerequisites: " + list of prerequisites" (or "No prerequisites." if none).

7. Sorting and Displaying Course List

7.1. Function: printSortedCourses(dataStructure)

7.1.1. Vector: Sort using O(n log n) sorting algorithm.

7.1.2. Hash Table: Convert to a list, then sort (O(n log n)).

7.1.3. BST: Perform in-order traversal (O(n)).

7.1.4. Print all course information.

8. Menu Implementation

8.1. Function: displayMenu()

8.1.1. Loop until the user selects Option 9 (Exit).

8.1.2. Display options:

8.1.2.1. 1. Load Course Data

8.1.2.2. 2. Print All Courses (Sorted)

8.1.2.3. 3. Search Course

8.1.2.4. 9. Exit

8.1.3. Get user input and call corresponding function.

Operation	Vector	Hash Table	Binary Search Tree
Insert Course	O(1) (append) / O(n) (sorted insert)	O(1) (average) / O(n) (worst)	O(log n) (average) / O(n) (worst)
Search Course	O(n)	O(1) (average) / O(n) (worst)	O(log n) (average) / O(n) (worst)
Remove Course	O(n)	O(1) (average) / O(n) (worst)	O(log n) (average) / O(n) (worst)
Print Courses	O(n log n) (sort)	O(n) (must iterate)	O(n) (in-order traversal)

Each data structure offers distinct advantages and trade-offs in managing course information. Vectors are simple to implement and efficient for small datasets, allowing for fast indexing and benefiting from CPU caching. However, searching and deletion operations require O(n) complexity, and explicit sorting is needed before retrieving courses in order, making them inefficient for large datasets. Hash tables provide the fastest search time with O(1) average lookup, making them ideal for frequent data retrieval. However, they do not maintain data in order, requiring additional sorting operations. Collisions can degrade performance, and hash tables generally consume more memory due to hash functions and linked lists for handling collisions.

Binary Search Trees (BSTs) naturally maintain sorted data, making them efficient for searching and retrieving courses in O(log n) time when balanced. They provide a good balance between searching and sorting without needing additional processing. However, BSTs can become unbalanced, leading to O(n) worst-case complexity, and they require additional logic for self-balancing, such as AVL trees. While BSTs offer efficient retrieval of sorted data, they are

more complex to implement than vectors and hash tables. Given the need for both fast searching and ordered retrieval, BSTs provide the best balance between efficiency and usability.