Module Five Lesson Review

Module Overview

- GitHub Repository Link
- UML Diagram Link
- ChatGPT Link

Binary Search Trees (BST)

Key Properties:

1. **Node Structure**: Each node has a key (and possibly an associated value), and two distinguished sub-trees, commonly denoted as left and right.
2. **Left Sub-tree**: All the keys in a node's left sub-tree are less than the node's key.
3. **Right Sub-tree**: All the keys in a node's right sub-tree are greater than the node's key.
4. **Distinct Keys**: No two nodes in the tree have the same key (assuming simple BST).

- Benefits of BST:
  1. **Ordered Structure**: This allows for efficient in-order traversal.
  2. **Dynamic Data Set**: BSTs can grow and shrink during the runtime, making it ideal for situations where the dataset is dynamic.
  3. **Efficient Operations**: Basic operations like search, insertion, and deletion can be efficient, especially in balanced BSTs.

Main Operations:

1. **Search**: Locates a node with a given key in the tree.
2. **Insertion**: Adds a new node with a specified key to the tree.
3. **Deletion**: Removes a node with a specified key from the tree.
4. **Traversal**: Processes all nodes of the tree in a specific order, e.g., in-order, pre-order, or post-order.

- Running Time:
  - **Best Case**: (O(log n))
    - Scenario: The BST is balanced.
  - **Worst Case**: (O(n))
    - Scenario: The BST is skewed.

Drawbacks:

1. **Skewed Trees**: Without balancing, BSTs can become skewed, leading to operations becoming inefficient.
2. **Balancing Overhead**: Trees that self-balance (like AVL or Red-Black Trees) can have overheads in maintaining the balance.

Conclusion:

Binary Search Trees provide a versatile data structure that supports ordered operations. While they offer many advantages, care must be taken (especially in the case of simple BSTs) to ensure they remain efficient in real-world applications by preventing them from becoming too skewed or unbalanced.

Find common Ancestor

- **Steps**:
  1. Create a function that takes two nodes and a BST as input.
  2. Search for the nodes in the BST.
  3. Determine the first common ancestor of the nodes.
  4. Return the common ancestor node.

Tree Rotations

- **Steps**:
  1. Define a function to perform the four types of rotations:
     - Left rotation
     - Right rotation
     - Left-Right rotation
     - Right-Left rotation
  2. The function should take in the tree and a specific node for rotation.
  3. Return the tree with the applied rotation.

Testing

- **Check for Balance**:
  - Update the balance factor for each node in the tree.
  - Recursively check the balance of the left and right subtrees.
- **Find Common Ancestor**:
  - Check if the value is in between the two nodes.
  - Compare values of each node.
  - Recurse through either the left or right subtree.
  - Determine the first common ancestor and return it.
- **Rotations**:
  - Test for the following scenarios:
    - Left rotation
    - Right rotation
    - Left-Right rotation
    - Right-Left rotation

Code Implementation

- **Strategy Design Pattern**

  - I chose to implement the Strategy Design Pattern for the following reasons:
  - The Strategy Pattern defines a set of algorithms, encapsulates each one, and makes them interchangeable. It lets the algorithm vary independently from the clients that use it.
  - To implement the various balancing algorithms and rotations in a way that is interchangeable and independent from the client code.
  - The concrete Tree class is the context, which uses a strategy to execute the algorithm. It contains a reference to a Strategy object and defines an interface that lets Strategy access its data.

Review and Submission

- **Challenges**:
  - Implementing the Strategy Design Pattern.
  - Implementing and Printing the tree after each rotation.
  - Creating the test cases.
    * I ended up hardcoding the test cases.
    * The Insert method created a balanced tree, so I had to manually create a skewed tree for the test cases.
  - Understanding instance methods. USE SELF BEN!
- **Feedback**:
  - This was an intersting module, and I enjoyed learning about the various balancing algorithms and rotations.
  - conceptually, I understood the material quite well, but I struggled with the implementation.
  - I was excited to implement a new design pattern, and found it quite fun to create the UML diagram and implement the code.
  - Unfortunately I didn't have any interactions with other students during this module, so I was unable to discuss the material with them.

Readings

- **CLRS**:
  - Chapter 12: Binary Search Trees
  - Chapter 13: Red-Black Trees
- **Algorithms Illuminated Part 2 - Tim Roughgarden**
  - Chapter 11: Search Trees
- **Discrete Mathematics - Richard Johnsonbaugh**
  - Chapter 8: Graph Theory
  - Chapter 9: Trees
- **Gang of Four Design Patterns**
  - Chapter 5: Behavioral Patterns(Strategy)