Hands On 1: Tree Data Structure Implementation in Rust

1.1 Introduction

The solutions for each problem have been implemented using the `Tree` struct provided in the course. The functions to check various properties (such as BST, balance, and max-heap) are implemented as methods for this struct. These functions are recursively applied to each node in the tree, with additional functionality to add nodes and compute tree properties.

1.2 IsBST

The `is_bst` method checks if the tree is a Binary Search Tree (BST). To verify if a tree is a BST, we ensure that for each node, the maximum value in the left subtree is smaller than the node's value, and the minimum value in the right subtree is greater than the node's value. The `rec_is_bst` helper function recursively checks each node's left and right subtrees, ensuring the BST properties are satisfied. For efficiency, the function returns a triple of condition state, minimum value, and maximum value for each node, rather than separately calculating the min and max for each subtree.

1.3 IsBalanced

To check if a tree is balanced, we check whether the height of the left and right subtrees for each node differ by at most 1. The `is_balanced` method utilizes a recursive helper function, `rec_is_balanced`, to calculate the height of each subtree and check if it meets the balance condition. If at any point the subtree is unbalanced (height difference > 1), the function returns `-1`, indicating that the tree is not balanced.

1.4 IsMaxHeap

A tree is considered a max-heap if it is complete (no missing nodes in any level) and if each node's key is greater than or equal to its children's keys. The method `is_max_heap` checks if the tree satisfies the max-heap property by recursively traversing the tree and ensuring the following:

1. The tree is complete, i.e., nodes at the last level are added from left to right.
2. Each parent node has a key greater than or equal to its children.

1.5 Testing

To ensure the correctness of the implemented functions, I have created a set of tests for each function. I designed these tests on paper, outlining the expected tree structure and results. Each test checks a specific property of the tree (sum, BST, balance, max-heap). These tests are executed using `cargo test`.

Sample Test:

The test cases include:

- **Sum Test:** Verifying the sum of all node keys in the tree.
- **BST Test:** Verifying that the tree satisfies the Binary Search Tree property.
- **Balanced Test:** Verifying that the tree is balanced, with left and right subtrees' heights differing by at most 1.
- **Max-Heap Test:** Verifying that the tree satisfies the max-heap property, where each parent node is greater than or equal to its children.

1.6 Source Code

The complete source code for the tree and its operations can be found in the `lib.rs` file in the `/hands-on/1/lib.rs` folder. To test the code, execute the following command from the project directory:

```
cargo test --package hands-on-1 --lib -- tests --nocapture
```

This command runs all the tests to verify the correctness of the tree's properties.

Conclusion:

In this report, I have implemented a tree data structure with various properties, such as checking if it is a Binary Search Tree (BST), if it is balanced, and if it is a max-heap. Each function is tested with multiple test cases to ensure correctness. The tests can be run using the `cargo test` command in the Rust environment.