Hands On 1 Report

This brief report outlines the solution for the first hands-on assignment of the Competitive Programming course, focusing on implementing recursive traversals of binary trees in Rust.

Basic Binary Tree Implementation

The tree is implemented using a `Node` struct that holds a `key` and optional indices (`Option<usize>`) for its left and right children, which refer to nodes stored in a `Vec<Node>` within the `Tree` struct.

Generic `Node<T>` and `Tree<T>` Design

Instead of using `u32` keys as in the original problem, my solution defines `Node` and `Tree` generically over a type parameter `T`. This makes the tree capable of storing different data types, enhancing flexibility.

To achieve this, I replaced the hardcoded key type with a generic type parameter `T`, which required more careful control over method implementations. Specifically, methods like `sum()`, `is_bst()`, and `max_path_sum()` needed their own `impl` blocks with appropriate trait bounds to ensure correctness.

I also adjusted method signatures and return types to avoid relying on default values (e.g., the original `sum()` returned 0 when hitting a `None` child), which can be problematic for generic types.

Additionally, I changed the original panic-based error handling in `add_node(...)` to return a `Result<usize, &str>`, enabling graceful error reporting on invalid insertions. Similarly, `get_node(...)` now returns an `Option<&Node<T>>` for safe access.

Revisiting `sum()`

With a generic key type, the `sum()` method needed to be adapted. Because addition isn't defined for all types, I constrained `T` with the traits `Copy` and `std::ops::Add<Output = T>`.

Unlike the original method that returned 0 for `None` nodes (which assumes a sum-neutral zero), my recursive implementation explicitly handles leaves and nodes with one or two children, avoiding reliance on a generic sum-neutral value hence avoiding type-restriction `T: Default`. That would have been hard to implement since we can't assure that every type default value is sum-neutral).

The time complexity remains straightforward: summing all keys requires visiting every node, so it's $O(n)$.

Checking Binary Search Tree Property (`is_bst()`)

In my opinion, checking whether a tree is a BST can be as easy as it sounds as deceptively tricky. It’s not enough that each node’s left child is less and right child is greater; the entire left and right subtrees must satisfy these bounds recursively.

To handle this, my recursive method propagates bounds down the tree, refining the allowable key range for each subtree. Because we cannot assume min/max values for generic `T`, I used `Option<T>` for the bounds, allowing flexible and safe comparisons without forcing restrictive trait bounds.

This approach ensures that the BST property is checked correctly across all subtrees, not just immediate children.

To keep the design in line with my generic `Tree<T>` implementation I enforced the type restriction over types that would implement the `std::cmp::Ord` trait. This is the only trait we need, since for the method to work we just need to make comparisons over node keys.

Same as the last, the time complexity remains $O(n)$ since the bounds checking has to be performed over every node of the tree.

Maximum Path Sum between two leaves (`max_path_sum()`)

The method computes the maximum sum of keys along any path connecting two leaves of the tree. To keep it in line with the generic `Tree<T>` implementation, we explicitly handle null trees and default value.

The core logic of the method is implemented in a private recursive function `max_path_sum_rec` which returnes a tuple with 2 values for the subtree rooted at a given node:

1. The maximum path sum from any lead up to the node (subtree root).
2. Maximum path sum between any two leaves within the subtree.

The recursion elaborates each individual node as follows:

- **Leaf Nodes:** the max path to root is the node key, and no leaf-to-leaf path exists (`None`)

- **Nodes with one child**: Since leaf-to-leaf paths require two leaves, no new leaf-to-leaf path is formed here. The function propagates the child's max leaf-to-root path plus the current node's key.
- **Nodes with two children**: Both subtrees' max leaf-to-root paths are combined with the current node's key to form a candidate leaf-to-leaf path crossing this node. The method then compares this candidate sum with the max leaf-to-leaf sums from the subtrees to determine the best overall value.

To manage the absence of leaf-to-leaf paths gracefully, the method uses `Option<T>` and carefully merges subtree results, with the above mentioned policy.

Since the function visits each node once, the overall complexity is $O(n)$

Test Suite Overview

To verify the correctness and robustness of the tree (considering both the generic data structure and methods), I wrote an extensive test module covering a wide range of scenarios. The tests use idiomatic Rust unit testing practices and are annotated with visual representation of tree structures where relevant, making the logic easy to follow and validate.

The coverage is as follows:

- **Tree Construction**: verifies that nodes are added correctly, respecting the binary tree structure. It also checks that invalid insertions (i.e., duplicate children or invalid parent IDs) are handled gracefully via `Result<>`.
- **Summing values**: confirms that `sum()` works for various types and tree shapes, including edge cases like empty trees or trees with negative and-or floating-point keys.
- **BST Validation**: Validates both well-formed and malformed binary search trees. It also includes deep structural violations and tests on strings to confirm correctness under generic constraints.
- **Maximum Path Sum**: Evaluates `max_path_sum()` across different cases including balanced, skewed, and entirely negative trees. Tree diagrams help highlight expected behavior.
- **Generic Support**: Ensures that the implementation works correctly for different types, such as `char` and `u8`, and demonstrates the effects of trait bounds like `Ord`, `Add`, and `Copy`.
- **Robustness and Error Handling**: Checks that attempts to perform invalid operations return proper error messages, aligning with the safe and idiomatic handling of `Result` and `Option`.