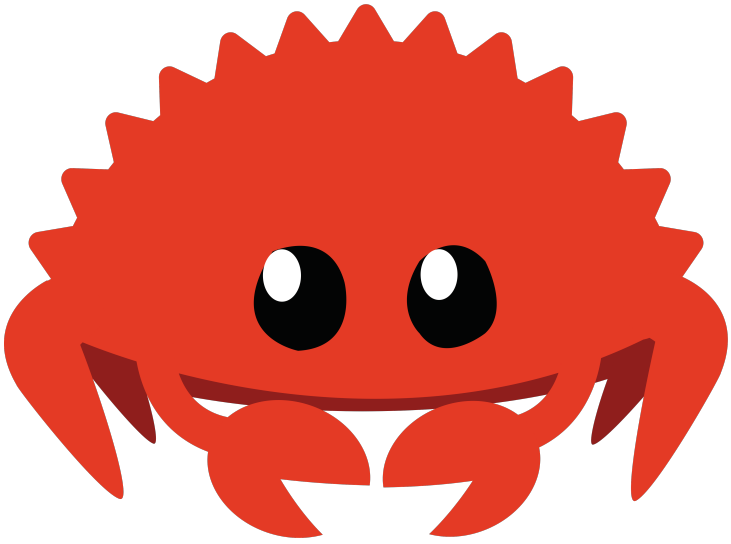
Hands-On 1 - Report

Giuseppe Di Palma - 635525



October 12, 2024

1 Introduction

In this report, we will evaluate how the functions `is_bst` and `max_leaf_path` are implemented. The goal is to verify that the `is_bst` function correctly identifies whether the tree is a valid Binary Search Tree (BST) and that `max_leaf_path` accurately calculates the maximum value of any path between two different leaf nodes.

1.1 Tree and Node Implementation

The binary tree is represented using a vector of `Node` structs, where each node has a key and optional left and right child indices. The `Tree` struct contains the non-empty vector of nodes. When the tree is created, it is required to specify the root key.

2 Function is_bst

The `is_bst` function verifies if the tree is a Binary Search Tree (BST). A tree is a BST if every node satisfies the following conditions:

- The subtrees rooted at the left and right children are both BSTs;
- The maximum key in the subtree rooted at the left child is less than the node's key;
- The minimum key in the subtree rooted at the right child is greater than or equal to the node's key.

The function uses a recursive helper function, `rec_is_bst`, which traverses the tree and checks the BST properties at each node.

```
pub fn is_bst(&self) -> bool {
    self.rec_is_bst(Some(0)).0
}

fn rec_is_bst(&self, node_id: Option<usize>) -> (bool, Option<u32>, Option<u32>) {
    if let Some(id) = node_id {
        assert!(id < self.nodes.len(), 'Node id is out of range');
        let node = &self.nodes[id];

        let (is_left_bst, min_left, max_left) =
            self.rec_is_bst(node.id_left);
        let (is_right_bst, min_right, max_right) =
            self.rec_is_bst(node.id_right);

        if !is_left_bst || !is_right_bst {
            return (false, None, None);
        }

        match (min_left, max_left, min_right, max_right) {
            (None, None, None, None) =>
                return (true, Some(node.key), Some(node.key)),
```

```
            (None, None, Some(min_right), Some(max_right))
                if min_right >= node.key =>
            {
                return (true, Some(node.key), Some(max_right));
            }
            (Some(min_left), Some(max_left), None, None)
                if max_left < node.key =>
            {
                return (true, Some(min_left), Some(node.key));
            }
            (Some(min_left), Some(max_left),
             Some(min_right), Some(max_right))
                if max_left < node.key && min_right >= node.key =>
            {
                return (true, Some(min_left), Some(max_right));
            }
            _ => {
                return (false, None, None);
            }
        }
    }

    (true, None, None)
}
```

Listing 1: The `is_bst` Function

The first thing `is_bst` does is verify that both the subtrees rooted at the left and right children of the nodes are BSTs.
If this is not the case, the tree is not a BST, and the function returns false.
If both subtrees are BSTs, the function proceeds to verify the other two properties explained above, in the various cases:

- If the node is a leaf, it is a BST with both the minimum and maximum value equal to the node's key.
- If the node has only the right child, the node's key should be less than or equal to the minimum key in the child BST.
- If the node has only the left child, the node's key should be greater than the maximum key in the child BST.
- If the node has both children, the node's key should be greater than the maximum key in the left child BST and less than or equal to the right child BST.

If none of these conditions are satisfied, the tree is not a BST, and the function returns false.
In terms of efficiency, this function performs a post-order traversal, resulting in a time complexity of $O(n)$, where n is the number of nodes.

3 Function max_leaf_path

The `max_leaf_path` function computes the maximum sum of a simple path that starts from one leaf and ends at a different leaf. This function is often used to find the "heaviest" path in a binary tree.

The function relies on a recursive helper function `rec_max_leaf_path`, which computes both the maximum path sum passing through a node and the maximum sum ending at a node.

```
pub fn max_leaf_path(&self) -> Option<u32> {
    match self.rec_max_leaf_path(Some(0)) {
        (Some(best), _) => Some(best),
        _ => None,
    }
}

fn add(&self, a: Option<u32>, b: Option<u32>) -> Option<u32> {
    match (a, b) {
        (Some(a), Some(b)) => Some(a.add(b)),
        (_, _) => None,
    }
}

fn rec_max_leaf_path(&self, node_id: Option<usize>) -> (Option<u32>,
    Option<u32>) {
    if let Some(id) = node_id {
        assert!(id < self.nodes.len(), 'Node id is out of range');
        let node = &self.nodes[id];

        let (best_left, max_left) =
            self.rec_max_leaf_path(node.id_left);
        let (best_right, max_right) =
            self.rec_max_leaf_path(node.id_right);

        let best_node =
            self.add(self.add(max_left, max_right), Some(node.key));
        let best = best_left.max(best_right.max(best_node));
        let max = self.add(max_left.max(max_right), Some(node.key));

        if max.is_none() {
            return (best, Some(node.key));
        } else {
            return (best, max);
        }
    }
    (None, None)
}
```

Listing 2: The `max_leaf_path` Function

The `rec_max_leaf_path` function begins by making a recursive call over the left and right child nodes. The best value (the value of the heaviest path in the subtree rooted at the node) is evaluated as the maximum between the best values of the subtrees rooted at the node's children and the value of the heaviest path passing through the node. We can see how the longest path passing through the node is evaluated using the `add` function, which allows safe summation operations over Option<u32> values. This is essential because a node may have one child, both children, or be a leaf.

The maximum value represents the heaviest path between a leaf and the node. It is evaluated by adding the node's key to the maximum between `max_left` and `max_right`. If both `max_left` and `max_right` are None (i.e., the node is a leaf), then the maximum value is simply the node key.

In terms of efficiency, the `rec_max_leaf_path` function also performs a post-order traversal, resulting in a time complexity of $O(n)$.