HandsOn 1 24/25

October 4, 2024

2024 · # rust, # algorithms, # data-structures · notes

This is the text of first hands-on of the the course *Competitive Programming and Contests* at University of Pisa in the accademic year 2024-25.

The objective of this hands-on is to implement recursive traversals of a binary tree in Rust. These exercises are valuable for preparing for coding interviews and are worth attempting even if you are not enrolled in the course.

Basic Binary Tree Implementation

Let's begin by describing a basic binary tree implementation in Rust.

In our implementation, a node is represented as a struct with three fields: the `key` of the node, and the ids of its `id_left` and `id_right` children. We represent the entire tree using a vector of `Node`s. Each node is implicitly assigned an ID that corresponds to its position in the vector.

Therefore, a node is defined as follows.

```
struct Node {
    key: u32,
    id_left: Option<usize>,
    id_right: Option<usize>,
}

impl Node {
    fn new(key: u32) -> Self {
        Self {
            key,
            id_left: None,
            id_right: None,
        }
    }
}
```

We have chosen to use `u32` as the data type for the `key`. Implementing a generic version of the `Node<T>` structure is left as an exercise, albeit potentially quite boring one. Both `id_left` and `id_right` are of type `Option<usize>` and store the IDs of the left and right children of the node, respectively. If a child does not exist, the corresponding ID is set to `None`.

To create a node, you can use the `new` function and specify its `key`. The newly created node is considered a leaf and, thus, both children are `None`.

Now, we are prepared to define the struct `Tree`, which is just a vector of nodes.

```
struct Tree {
    nodes: Vec<Node>,
}
```

In our implementation, we have chosen not to allow empty trees. This simplifies the code a little bit. However, it's easy to reverse this decision if necessary.

You can create a new tree using the `with_root(key: u32)` function, which initializes a new tree with a root having the specified `key`. The ID of the root node is always `0`.

We have also decided to restrict operations to only insertions of new nodes; that is, deletions or modifications of existing nodes are not allowed. This limitation aligns with our objectives, as our primary focus is on tree traversal.

To insert a new node, you can use the `add_node` method. When adding a new node, you need to specify its `parent_id`, its `key`, and a boolean value, `is_left`, which indicates whether the node should be the left or right child of its parent. The method panics if the `parent_id` is invalid or if the parent node has already assigned the child we are trying to insert.

The implementation of a tree is as follows.

```rust
impl Tree {
    pub fn with_root(key: u32) -> Self {
        Self {
            nodes: vec![Node::new(key)],
        }
    }

    /// Adds a child to the node with `parent_id` and returns the id of the new node.
    /// The new node has the specified `key`. The new node is the left  child of the
    /// node `parent_id` iff `is_left` is `true`, the right child otherwise.
    ///
    /// # Panics
    /// Panics if the `parent_id` does not exist, or if the node `parent_id ` has
    /// the child already set.
    pub fn add_node(&mut self, parent_id: usize, key: u32, is_left: bool) -> usize {
        assert!(
            parent_id < self.nodes.len(),
            "Parent node id does not exist"
        );
        if is_left {
            assert!(
                self.nodes[parent_id].id_left == None,
                "Parent node has the left child already set"
            );
        } else {
            assert!(
                self.nodes[parent_id].id_right == None,
                "Parent node has the right child already set"
            );
        }

        let child_id = self.nodes.len();
        self.nodes.push(Node::new(key));

        let child = if is_left {
            &mut self.nodes[parent_id].id_left
        } else {
            &mut self.nodes[parent_id].id_right
        };

        *child = Some(child_id);

        child_id
    }
}
```

Computing the Sum of Keys in a Binary Tree

Let's implement a simple tree traversal to compute the sum of the keys in a binary tree. This can serve as an example for implementing the solutions for the three exercises below.

We will use a recursive function called `rec_sum(&self, node_id: Option<usize>)`. This function takes a `node_id` as input and computes the sum of all the keys in the subtree rooted at `node_id`. There are two possibilities. If `node_id` is `None`, the subtree is empty, and thus, the sum is `0`. However, if `node_id` refers to a valid node, the sum of the keys is equal to the key of the current node plus the sums of its left and right subtrees. These latter sums are computed recursively.

Here is the Rust code. Note that we have the `sum` method, which is responsible for calling `rec_sum` at the root.

```rust
/// Returns the sum of all the keys in the tree
pub fn sum(&self) -> u32 {
    self.rec_sum(Some(0))
}

/// A private recursive function that computes the sum of
/// nodes in the subtree rooted at `node_id`.
fn rec_sum(&self, node_id: Option<usize>) -> u32 {

    if let Some(id) = node_id {
        assert!(id < self.nodes.len(), "Node id is out of range");
        let node = &self.nodes[id];

        let sum_left = self.rec_sum(node.id_left);
        let sum_right = self.rec_sum(node.id_right);

        return sum_left + sum_right + node.key;
    }

    0
}
```

The code described so far is here.

Exercise #1

Write a method to check if the binary tree is a **Binary Search Tree**.

Exercise #2

Write a method to solve the Maximum Path Sum problem. The method must return the sum of the maximum simple path connecting two leaves.

Test Your Solutions

In the code snippet below, we provide a (limited) set of tests for the `sum` method. This code also shows how to construct a binary tree using our implementation. To ensure the robustness of your solutions, we strongly recommend adding a comprehensive suite of tests

```rust
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_sum() {
        let mut tree = Tree::with_root(10);

        assert_eq!(tree.sum(), 10);

        tree.add_node(0, 5, true); // id 1
        tree.add_node(0, 22, false); // id 2

        assert_eq!(tree.sum(), 37);

        tree.add_node(1, 7, false); // id 3
        tree.add_node(2, 20, true); // id 4

        assert_eq!(tree.sum(), 64);
    }
}
```

Submission

Submit a file `lib.rs` and a file `Handson_1_solution_YOUR_NAME.pdf` to rossano.venturini@gmail.com by 21/10/2024.

- Source code `lib.rs` contains your implementations and a large set of tests.
- A report `Handson_1_solution_YOUR_NAME.pdf` that briefly describes your implementations.

Before submitting your solutions,

- make sure your implementation successfully passes all the tests.
- use `cargo fmt` to format your code.
- use `cargo clippy` to check your code.
- use Grammarly to improve your English and avoid tpyos :-). There is an extension for vscode.

Cheating

Very important! You are allowed to verbally discuss solutions with other students, **BUT** you must implement all solutions by yourself. Therefore, sharing implementations with others is strictly **forbidden**.

Enjoy Reading This Article?

Here are some more articles you might like to read next:

- HandsOn 3
- HandsOn 2
- Mo's Algorithm
- Dynamic Prefix Sums with Fenwick Tree
- The Power of Prefix Sums