Handson 1 solution

Tommaso Crocetti

Brief introduction

- The solutions use the tree implementation given on the course's website
- The base case for the recursive function is a visit of an empty node, reached after a visit of a leaf

```
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,
        }
    }
}
```

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

Exercise #1

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

Idea of the solution

A binary tree is a binary search tree if each node's key is bigger or equal to all keys in the left subtree and smaller than all in the right subtree. So, to verify these conditions it's not possible to only check on the current node's key and the key of the left and right child because we will miss the check on the global condition (a key must be greater/smaller of ALL key in the respective subtree). So for each node, we compare the current key with the maximum key in the left subtree and with the minimum key in the right subtree. If a node verifies the two conditions with those values, than its key must be greater or equal/smaller than every key in the respective subtrees.

Solution #1

The implementation of the solution is split into two functions:

- check_bst: the wrapper function, calls the recursive function on the root's index of the tree (Some(0)) and returns the recursive result, ignoring the other values returned.

- rec_check_bst: the recursive function, checks the bst condition in every node. The return value is a triple where the first element is a boolean value that states if the node is a root for a valid bst and then it contains two u32 values represent respectively the minimum and maximum value contained in the subtree. If the recursive call happens on an empty node, it just returns the base values (true, 0, u32::MAX), otherwise, it calls recursively on the left and right children, so that it can check the bst condition with the result obtained. At the end, it returns the boolean result, the minimum and maximum of the current subtree as the minimum/maximum between the current node's key and the left and right minimum/maximum.

Code #1

```rust
pub fn check_bst(&self) -> bool {
    let (result, _, _) = self.rec_check_bst(Some(0));
    result
}
```

```rust
fn rec_check_bst(&self, node_id: Option<usize>) -> (bool, u32, u32) {
    if let Some(id) = node_id {
        assert!(id < self.nodes.len(), "Node id is out of range");
        let node = &self.nodes[id];
        let (bst_l, min_l, max_l) = self.rec_check_bst(node.id_left);
        let (bst_r, min_r, max_r) = self.rec_check_bst(node.id_right);
        let result: bool = bst_l && bst_r && node.key >= max_l && node.key < min_r;
        return (result, node.key.min(min_l).min(min_r), node.key.max(max_l).max(max_r));
    }
    (true, 0, u32::MAX)
}
```

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.

Idea of the solution

To find the maximum (simple) path sum in a tree it is necessary to evaluate the maximum between the maximum path sum in the left and right subtree and the maximum sum path that steps on the current node. While the first two values can be simply collected recursively, the last one must be calculated as the sum between the current node's key and the left and right maximum path sum from a leaf to the subtree's root.

Solution #2

As before, the implementation of the solution is split into two functions:

- max_path_sum: the wrapper function, calls the recursive function on the root's index of the tree (Some(0)) and returns the recursive result, ignoring the other value returned.

- rec_max_path_sum: the recursive function, evaluates the maximum path sum on the current node's subtree and the maximum path sum from a leaf to the current node. The result value is a couple of u32. If the recursive call happens on an empty node, it just returns the base values (0, 0), otherwise, it calls recursively on the left and right children, obtaining the values of the two maximum paths sum. The first value returned is the maximum between the left and right solution and the sum of the path stepping in the current node, calculated as explained before. The second value is the sum between the current node's key and the maximum path leaf-child of the left or right subtree, the new maximum path sum leaf-node.

Code #2

```
pub fn max_path_sum(&self) -> u32 {
    let (result, _) = self.rec_max_path_sum(Some(0));
    result
}
```

```
fn rec_max_path_sum(&self, node_id: Option<usize>) -> (u32, u32) {
    if let Some(id) = node_id {
        assert!(id < self.nodes.len(), "Node id is out of range");
        let node = &self.nodes[id];
        let (lres, lpath) = self.rec_max_path_sum(node.id_left);
        let (rres, rpath) = self.rec_max_path_sum(node.id_right);
        return (lres.max(rres).max(lpath + rpath + node.key), lpath.max(rpath) + node.key);
    };
    (0, 0)
}
```