Handson 1 solution

Tommaso Crocetti

Brief introduction

- The solutions use the tree implementation given on the course's website, extended to support both signed and unsigned integers.

```
pub struct Node<T>
where
    T: Integer + Bounded + Copy + Add<Output = T> + Sub<Output = T>,
{
    key: T,
    id_left: Option<usize>,
    id_right: Option<usize>,
}
impl<T> Node<T>
where
    T: Integer + Bounded + Copy + Add<Output = T> + Sub<Output = T>,
{
    fn new(key: T) -> Self {
        Self {
            key,
            id_left: None,
            id_right: None,
        }
    }
}
```

```rust
pub struct Tree<T>
where T: Integer + Bounded + Copy + Add<Output = T> + Sub<Output = T>,
{
    nodes: Vec<Node<T>>,
}
impl<T> Tree<T>
where
    T: Integer + Bounded + Copy + Add<Output = T> + Sub<Output = T>,
{
    pub fn with_root(key: T) -> Self {
        Self {
            nodes: vec![Node::<T>::new(key)],
        }
    }

    pub fn check_bst($self) -> bool
    fn rec_check_bst($self, node_id: Option<usize>) -> (bool, T, T)
    pub fn max_path_sum(&self) -> Option<T>
    fn rec_max_path_sum($self, node_id: Option<usize>) -> (Option<T>, Option<T>)
    //add_node and other methods...
}
```

- The base case for the recursive functions is a visit of an empty node, reached after a visit of a leaf.
- The given file "lib.rs" contains the definition of other methods, for the aim of the hands-on, only the functions check_bst, rec_check_bst, max_path_sum, and rec_max_path_sum are needed.

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 than ALL keys 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, then 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 the other two T 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, T::max_value(), T::min_value()), because in this way every key would be lower than the minimum and higher that the maximum.
  - In all other cases, it calls recursively on the left and right children, so that it can check the bst condition in the lower subtrees and also compute the local bst check with the other results obtained. At the end of a node's visit, it returns the boolean result of the bst check in all subtrees, 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, T, T) {
    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 = node.key >= max_l && node.key < min_r;
        return (
            result && bst_l && bst_r,
            node.key.min(min_l).min(min_r),
            node.key.max(max_l).max(max_r),
        );
    }
    (true, T::max_value(), T::min_value())
}
```

This solution exploits a post-order visit, with constant time operations in each node. Given the number of nodes n, the time complexity is Θ(n), and space complexity is Θ(1) since it doesn't require any additional space.

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 from one leaf to another leaf 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 current node. To build a correct solution, it's important to check if the current subtree has at least two leaves, otherwise, the solution returned could be inconsistent.

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 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 the sum from a leaf to the current node. The result value is a a couple of Option T values. None on the first value means that there isn't a couple of leaves in the subtree, None on the second value means that there isn't even a leaf.
  - If the recursive call happens on an empty node, it just returns the base values (None, None).
  - Otherwise, it calls recursively on the left and right children, so it can check if the subtrees contain already a valid solution or at least a valid max path, so to build the current solution. Valid paths are built starting from a leaf key. When a node has no valid solution in the subtrees but has two valid paths, it gives back a solution that is the sum of the paths and its key. If a node has at least one solution in the subtree, it returns the maximum between the valid solutions and the sum of the valid paths and its key.

Code #2

```
fn rec_max_path_sum(&self, node_id: Option<usize>) -> (Option<T>, Option<T>) {
    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);
        match (lres, rres) {
            (Some(left_res), Some(right_res)) => {
                return (
                    Some(
                        left_res
                            .max(right_res)
                            .max(lpath.unwrap() + rpath.unwrap() + node.key),
                    ),
                    Some(lpath.unwrap().max(rpath.unwrap()) + node.key),
                );
            }
            (Some(left_res), None) => match rpath {
                None => {
                    return (Some(left_res), Some(lpath.unwrap() + node.key));
                }
```

```
            Some(right_path) => {
                return (
                    Some(left_res.max(lpath.unwrap() + right_path + node.key)),
                    Some(lpath.unwrap().max(right_path) + node.key),
                );
            }
        },
        (None, Some(right_res)) => match lpath {
            None => {
                return (Some(right_res), Some(rpath.unwrap() + node.key));
            }
            Some(left_path) => {
                return (
                    Some(right_res.max(left_path + rpath.unwrap() + node.key)),
                    Some(left_path.max(rpath.unwrap()) + node.key),
                );
            }
        },
        (None, None) => match (lpath, rpath) {
            (Some(left_path), Some(right_path)) => {
                return (
                    Some(left_path + right_path + node.key),
                    Some(left_path.max(right_path) + node.key),
                );
            }
```

```
                (Some(left_path), None) => {
                    return (None, Some(left_path + node.key));
                }
                (None, Some(right_path)) => {
                    return (None, Some(right_path + node.key));
                }
                (None, None) => {
                    return (None, Some(node.key));
                }
            },
        }
    };
  (None, None)
}
```

As the previous solution, it uses a post-order visit on the tree with constant time operation in each node, so it requires so Θ(n) time complexity and Θ(1) space complexity in total.

Thanks for the attention!

Here you can find my updated Github repository for the course