Competitive Programming and Contests Hands-On 1 report

Salvatore Salerno 2024/2025

Exercise 1 - Checking if a Binary Tree is a Binary Search Tree (BST)

The exercise consists in writing a method which determines whether the binary tree invoking it is a Binary Search Tree (BST) or not. A binary tree t is considered a BST if, for every node $n \in t$, the maximum key in its left subtree is less than n's key, and the minimum key in its right subtree is greater than n's key. A solution to the problem is implemented through the public method `is_bst` and the private method `is_bst_rec`.

The `is_bst` method

The `is_bst` method is a dummy public method that calls `is_bst_rec` to return a boolean value for determining if the tree is a BST. It doesn't take any parameters because the check is implicitly started at the root of the tree (i.e. at index 0). This function serves as a wrapper since `is_bst_rec` requires specific input parameters that the user should not know or be able to provide.

The `is_bst_rec` method

The `is_bst_rec` method provides the solution to the problem. As the name suggests, it is a recursive function and it takes the following inputs: `node_idx`, the index of the node where the check begins; `min`, the minimum key value encountered so far in the upper part of the tree; and `max`, the maximum key value encountered so far in the upper part of the tree.

For the subtree rooted at `node_idx`, the method checks if it satisfies the BST property while updating the `min` and `max` values. If `min` is greater than the current key k or `max` is smaller than it, their values are set to k. If the node has a left subtree (i.e., it has a left child), the method recursively calls `is_bst_rec` on it. In the recursive call the parameter `min` is set as the current minimum, while `max` is set as the key k of the current node. This is because, for the left subtree to be a BST, it needs to have all its keys strictly smaller than k. The methods returns its updated minimum, maximum and a boolean stating if the subtree is a BST. In case the maximum has been updated (i.e. we found a node inside the subtree which has a key greater than k) or the explored subtree is not a BST tree, then the whole tree is not a BST and we can stop the check. If the BST property is still valid, we update the minimum value if a smaller one was found in the left subtree. This is important because if the current tree we are checking is a right subtree, the minimum value from the entire subtree must be compared to the root's key. If the node has a right subtree (i.e., a right child), a similar process is performed. The main difference is that, in this case, we verify that the minimum value (set as the current node's key k) has not been updated in the right subtree visit. Additionally, we update the current `max` if we have found a greater value, in the case the tree is a left subtree. If the method did not fail previously, the current tree is a BST and three values are returned to the caller: `min`, the updated minimum key value after scanning the current subtree; `max`, the updated maximum key value after scanning the current subtree; and `bst` set as true since it has been proved that the current tree is a BST.

This solution allows us to verify the BST property by visiting each node exactly once during the entire check, ensuring an efficient traversal. The Rust implementation uses unmutable variables throughout, except for the `min` and `max` variables within each local function call, since they could be dynamically updated during execution.

Exercise 2 - Maximum Path Sum problem

This exercise involves writing a method to solve the Maximum Path Sum problem. The task is to find the maximum sum of a simple path connecting two leaves in a given tree t. The method should only return the sum of the found path. A solution to the problem is implemented through the public method `max_path_sum` and the private method `max_path_sum_rec`.

The `max_path_sum` method

The `max_path_sum` method is a dummy public method that calls `max_path_sum_rec` to obtain the solution. Since the process always starts at the root of the tree (i.e., index 0), it doesn't require any parameters, unlike `max_path_sum_rec`, whose parameter should not be exposed to the user.

The `max_path_sum_rec` method

The method implements the solution discussed during the lessons, exploiting Rust's pattern matching constructs. Each recursive call begins at the node with index `node_idx`, which is passed as input. During its execution, the method computes bu and mu, representing the maximum path sum encountered so far between two leaves and the sum of the optimal path from a leaf to the current node, respectively. It's important to note that the value of mu does not constitute a valid path since it does not connect two leaves; however, it is utilized in higher-level calls to determine their bu. In the code, both bu and mu are represented as `Option<u32>`, where the value `None` is interpreted as the minimum possible value (i.e., $-\infty$). Additionally, the sums are stored as unsigned values since the keys in the provided implementation of the tree are always positive. To compute bu and mu, the method first recursively calls itself on the left and right subtrees of the current node, returning their local best paths as (bl, ml) and (br, mr). Finally, the execution can determine bu and mu using the following expressions:

$$\begin{aligned} bu &= \max(bl, br, ml + mr + k) \\ mu &= \max(ml, mr) + k \end{aligned}$$

where ks is the current node's key. Note that Rust's `max` function interprets `None` as $-\infty$ like previously implied.

Finally, the method returns the pair (bu,mu) to the caller. If the tree examined is not a subtree (i.e. `node_idx` refers to the root), the solution is contained in bu, while mu is not relevant. For this reason in the `max_path_sum` method, only the value of bu is returned to the user.

Testings the implemented methods

For all the implemented methods in the code tests have been provided to verify their correctness. These tests are organized into the modules `bst_tests` and `sum_tests`. For each test, we construct a tree and validate each intermediate representation using the private function designed for that exercise. For the final, completed tree, the public method is invoked.