Report HandsOn 1

> The implementation of binary trees has been slightly modified so that negative numbers can also be handled. Essentially the type of the `key` attribute of the tree has been changed from `u32` to `i32`.

Binary Search Tree

The `is_bst()` method makes use of the auxiliary function `is_bst_priv(node_id)`, which takes as input the index of the node in the vector `nodes`, representing the root tree from the subtree for which the property is to be verified. Wanting of course to verify it for the whole tree, we pass as index `0` representing the root.

Now let us analyse the `is_bst_priv(node_id)` function. If the index of the passed node is `None`, it means that the parent of `node_id` that invoked the procedure has no left or right subtree. So an empty tree obviously respects the property of being a binary search tree, but since it contains no values, we set `MIN` as the *maximum* element in the left subtree and `MAX` as the *minimum* value in the right subtree.

If the `node_id` exists, it will have an associated `v` value representing its identifier, i.e. its index in the `nodes` vector. Now before computing the necessary information for the current subtree, we decide to use a *bottom-up* approach, and so make recursive calls first to the left subtree and then to the right subtree, obtaining their return values. For each subtree, we will therefore have the *minimum* and *maximum* element and a boolean value `is_bst_*` to indicate whether the subtree in question is a binary search tree. We now obtain the value of the current node via the node's `key` attribute in the vector.

We can now compute the information to be passed to the current node's parent.

The minimum in this subtree will be the minimum between the left subtree, the right subtree, and the current node joining the two subtrees. For the maximum, the procedure is symmetrical. Instead, to verify that the current subtree is a binary search tree, it is necessary:

- Firstly, that both subtrees are already BSTs.
- That the current node is greater than or equal to (let us assume that we always put the nodes with equal value in the left subtree) the maximum value in the left subtree.
- And that the value of the current node is also less than or equal to the minimum value found in the right subtree.

These three conditions combined make it possible to check whether the current subtree is a BST. Then returning to procedure `is_bst()` will return the last value of the triple, which will represent the boolean value associated with the complete tree.

Maximum Path Sum

> The exercise trace on GeeksForGeeks requires the calculation of the maximum path sum between two special nodes, where special nodes are understood to be those which are connected to at most one more node. So essentially these can only be the leaves of the tree and the root only in the case where one of the two subtrees is empty.

Also for this function we use an auxiliary procedure `maximum_path_sum_priv(node_id)` invoked on the index of the root node.

In this case, for each subtree, we decide to return a pair in which the first value represents the *highest cost from a leaf* of the subtree to reach the current root/node, while the second value represents the *maximum path sum* between two leaves of the subtree (which may then be a candidate for the trees containing the subtree being analysed).

In the base case, where the node is empty, the highest cost path from a leaf to the current node does not exist, and so `None` is returned, while the maximum path sum will have the lowest possible value (`MIN`) as its cost.

In the case where the node exists, we associate `n` with the value of the key, and again we use a *bottom-up* approach by first computing the information in the two subtrees.

Now it is first necessary to calculate the maximum path sum from a leaf of the subtree to `v` called `max_p`, and the maximum path sum passing through `v` called `sum_l_to_r`.

In the case where the maximum path of both subtrees is `None` it will mean that `v` is a leaf, and so the maximum cost from a leaf to `v` is the key value of `v` itself, while the maximum path sum from always remains `MIN` since there are no leaves that can be joined via `v`.

Also in the second and third cases, the value associated with the maximum path sum is `MIN`, always because there are no leaves to the right or left that can be connected to another belonging to the existing subtree. Instead, only the path of the existing subtree will be taken into account when calculating the maximum path.

In the last case, where both the left and right subtrees exist, for the maximum path to `v` we take the max between the left and right subtrees, and for the maximum path sum we do the sum of the maximum paths of the two subtrees plus the current node.

Finally, we restore the calculated maximum path (`max_p`) and the maximum path sum between the one in the left subtree, the one in the right subtree, and the calculated one passing through the current node.

To respect the constraint stated at the beginning, in the case where `u` is the root of the entire tree, and this is a special node, we also add the best path from a leaf to the root (which will belong to the only existing subtree) to the candidate values as maximum path sum.

The `maximum_path_sum` function will then return the last value of the tuple returned on the call made by the helper function on the root node of the tree.