First hands-on report

Francesco Scarfato

October 18, 2024

Given a binary tree T, the objective of the hands-on is to implement a method to check if this tree is a *binary search tree* (BST) and a method to compute the *sum of the maximum simple path connecting two leaves.*

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

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

The methods implemented operates on a tree-like structure (vector) in which each node has

- `key` representing the value stored in the node;
- `id_left` and `id_right` representing the index to visit the left and right son.

1 Binary Search Tree check

Definition 1 *T is a binary search tree if it is valid:*

$$\forall \; node \; n \in T \mid left(n) < n < right(n)$$

To prove that, it is possible to use the *in order visit* on the tree. Staying on the current node n, this traversal

1. visit the left son of n;
2. return from the left son and does computation in n;
3. visit the right son of n.

The idea is to ensure the proper order of the nodes concerning definition 1 through in order visit.
To check if the tree is a BST, two methods have been implemented: the main method `is_bst` and the auxiliary function `rec_bst`.

```
pub fn is_bst(&self) -> bool {
    self.rec_bst(Some(0), &mut None)
}
```

This method is implemented in the `Tree` structure and is exposed externally to be called on the tree itself. It calls simply the auxiliary function to start from the root of the tree (index of the tree `Some(0)`) and return `true` if the tree is a BST, otherwise `false`.
The auxiliary function does all the work and has the following signature:

```
fn rec_bst(&self, current_node: Option<usize>, previous_node: &mut
    Option<u32>) -> bool
```

The parameters to pass are:

- `current_node: Option<usize>` which is the index of the current node to visit;
- `previous_node: &mut Option<u32>` which is a mutable reference to the value (`u32`) of the previous node. It is necessary to check that nodes are visited in increasing order.

`rec_bst` is a recursive function (as you can guess by the name), so it has a base case that is checked by the first `if let` statement.

```
if let Some(current_id) = current_node
```

If `current_node` doesn't store an index, the tree is empty or the algorithm has arrived at a leaf. In this case the definition 1 is valid yet, so it is returned `true`. Otherwise, it checks the left subtree through in order traversal (as explained before).

```
if !self.rec_bst(tree[current_id].id_left, previous_node) {
    return false;
}
```

If the left subtree isn't a binary search tree, then the whole tree isn't a BST, so it returns `false`.

```
if let Some(previous_id) = previous_node {
    if *previous_id > tree[current_id].key {
        return false;
    }
}
```

This is the key part of the algorithm because it checks whether the BST's property is respected. It checks if the previous node is smaller than the current one and returns `false` if it isn't.

```
*previous_node = Some(tree[current_id].key);
```

After the check, it updates the reference to the value of the current node and checks if the right subtree is a BST with the following statement:

```
return self.rec_bst(tree[current_id].id_right, previous_node);
```

2 Maximum Path Sum

To compute the sum of the maximum simple path, the algorithm operates a post order visit and keeps trace of the current maximum path and the overall maximum path. As previously explained, two methods are implemented also here: the main method `max_path_sum` and the auxiliary function `rec_max_path_sum`.

```
pub fn max_path_sum(&self) -> Option<u32> {
    self.rec_max_path_sum(Some(0)).0
}
```

`max_path_sum` returns the maximum path sum and, if the tree is empty, returns `None`. It calls the auxiliary function with parameter `Some(0)` to start from the root of the tree.
The auxiliary function is recursive and has the following signature:

```
fn rec_max_path_sum(&self, current_node: Option<usize>) -> (Option<u32>,
    Option<u32>)
```

The input of this function is the index of the current node being processed, while the output is a pair:

- the maximum path sum processed so far;
- the current path sum which is the maximum path from the current node down to a leaf.

At the end, it will be interesting only the first value. If the tree is empty or the current node doesn't exist (`None`), this is the base case and the function returns the pair `(None, None)`. This indicates that no valid path exists beyond this point.

```
let (left_max, left_sum) = self.rec_max_path_sum(node.id_left);
let (right_max, right_sum) = self.rec_max_path_sum(node.id_right);
```

Otherwise, it does two recursive calls: one on the left subtree and another one on the right subtree. This post order traversal returns the maximum path so far of the two subtrees and their current path sum.

```
    let path_sum = Some(node.key +
        left_sum.unwrap_or(0).max(right_sum.unwrap_or(0)));
```

It computes the current maximum path sum adding the current key to the max between the right path sum and left path sum. `left_sum.unwrap_or(0).max(right_sum.unwrap_or(0))` computes the maximum between the two paths and checks if are valid. If the paths are `None`, then `left_sum` and `right_sum` have the default value 0.
The next step is to compute the maximum path sum seen so far, so the method compares the maximum path sum of the left subtree, the maximum path sum of the right subtree and the current path sum; it

stores the maximum between these three values in `max_sum`. With the help of Rust iterators, it is possible to do that elegantly:

```rust
let max_sum = left_max
    .into_iter()
    .chain(right_max)
    .chain(Some(
        node.key + left_sum.unwrap_or(0) + right_sum.unwrap_or(0),
    ))
    .max();
```

More in depth, the steps are:

1. create an iterator from `left_max` with `left_max.into_iter()`;
2. create an iterator from `right_max` and link with the iterator created in the step before with `.chain(right_left)`;
3. create the last iterator from `Some(node.key + left_sum.unwrap_or(0)+ right_sum.unwrap_or(0))` and link with the previous one with `.chain(...)`. At this point, there is only one iterator that contains all the three values to compare.
4. Return the maximum element of the iterator with the method `.max()`.

The last step is to return the pair.