Binary Search Tree

A binary search tree (BST) is a rooted binary tree data structure with the key of each internal node being greater than all the keys in the respective node's left subtree and less than the ones in its right subtree.

To check whether the given tree is a BST there is the public function

```
pub fn is_bst(&self) -> bool
```

that calls a private recursive helper function `rec_is_bst` with the following parameters:

- `node_id`, the ID of the current node
- `min` and `max`, represent the range within which the current node's key must fall to be considered a valid BST node.

The function follows the following steps:

1. If `node_id` is `None` the function returns `true` since an empty tree is a BST
2. If `node_id` is `Some(id)`
    - asserts that the provided `id` is of a node in the tree
    - retrieves the current node from the tree using the provided `node_id`
    - checks if the key of the current node is within the valid range
    - checks the left and right subtrees using appropriate ranges
        - recur on the left subtree with the key of the current node as `max`
        - recur on the right subtree with the key of the current node as `min`
    - returns `true` if the current node satisfies the BST condition and both the left and right subtrees are BSTs, otherwise it returns `false`

For testing purposes is also provided the function

```
pub fn subtree_is_bst(&self, node_id: usize) -> bool {
        self.rec_is_bst(Some(node_id), u32::MIN, u32::MAX)
}
```

that takes a `node_id` and calls the `rec_is_bst` to verify that the subtree rooted in that node is a BST.

Time Complexity: $O(n)$

- the function uses a recursive approach and traverses all the nodes in the tree once
- at each node are performed operations that cost $O(1)$

- the complexity is then linear in the number of nodes

Space Complexity: $O(n)$

- the space complexity is impacted by the height of the tree due to the recursive nature of the function
  - in a "totally unbalanced" tree the number of recursive calls is $O(n)$
  - in a balanced tree the number of recursive calls is $O(\log(n))$
- the space complexity is given by the size of the tree itself, $O(n)$

Balanced Tree

A binary tree is considered balanced if, for each of its nodes, the heights of its left and right subtrees differ by at most one.

To check whether the given tree is balanced there is the public function

```
pub fn is_balanced(&self) -> bool
```

that calls a private recursive helper function `rec_is_balanced` that takes as parameter the current node `node_id: Option<usize>`.

The function follows the following steps:

1. If `node_id` is `None` the function returns 0 since an empty tree is balanced
2. If `node_id` is `Some(id)`
   - asserts that the provided `id` is of a node in the tree
   - retrieves the current node from the tree using the provided `node_id`
   - recursively calculates the heights of the left and right subtrees
   - returns the height of the current subtree as `1 + max(left_height, right_height)` if both subtrees are balanced, otherwise it returns -1

For testing purposes is also provided the function

```
pub fn subtree_is_balanced(&self, node_id: usize) -> bool {
        self.rec_is_balanced(Some(node_id)) != -1
}
```

that takes a `node_id` and calls the `rec_is_balanced` to verify that the subtree rooted in that node is balanced.

Time Complexity: $O(n)$, same reasoning used for `is_bst`

Space Complexity: $O(n)$

- the space complexity is impacted by the number of recursive calls on the stack
  - $O(\log(n))$ calls, since if the tree is not balanced the functions stop
- the space complexity is given by the size of the tree itself, $O(n)$

Max-Heap

A max-heap is a complete binary tree in which every node satisfies the max-heap property. A node satisfies the max-heap property if its key is greater than or equal to the keys of its children.

To check whether the given tree is a max-heap there is the public function

```
pub fn is_max_heap(&self) -> bool
```

that calls a private helper function `iter_is_max_heap` that takes as parameter the current node `node_id: Option<usize>`.

The function is an iterative inspection of the tree and it exploits a *frontier* to store nodes yet to visit.
The behavior of the function can be represented with the following steps:

1. If `node_id` is `None` the function returns `true` since an empty tree is a max-heap
2. If `node_id` is `Some(id)`
   - asserts that the provided `id` is of a node in the tree
   - retrieves the current node from the tree using the provided `node_id`
   - adds the current node into the frontier
   - while there are nodes in the frontier:
     - pop a node from the frontier
     - if the node has any child but it was previously found a leaf returns `false`
     - if the node has a right child but not a left child return `false`
     - if the node has a left child:
       - if the node has no right child then set `last_level` to `true`, all the next nodes in the frontier must be leaves
       - if the key of the node is smaller than the left child's key return `false`
       - add the left child to the frontier
     - if the node has the right child:
       - if the key of the node is smaller than the right child's key return `false`
       - add the right child to the frontier
     - if the node has no child set `last_level` to `true` as we found a leaf

For testing purposes is also provided the function

```rust
fn subtree_is_max_heap(&self, node_id: usize) -> bool {
        self.iter_is_max_heap(Some(node_id))
}
```

that takes a `node_id` and calls the `iter_is_max_heap` to verify that the subtree rooted in that node is a max-heap.

Time Complexity: $O(n)$

- the function iteratively traverses all nodes in the tree once
- at each node are performed operations that cost $O(1)$
- the complexity is then linear in the number of nodes n

Space Complexity: $O(n)$, classic complexity of a level-traversal approach

- in the worst case, all nodes at a certain level of the tree are stored in the queue.
- the space complexity of the queue is proportional to the number of nodes at the widest level of the tree.
- in a complete binary tree, the widest level can have up to half of the nodes in the tree.
- therefore, the space complexity of the queue is $O(\frac{n}{2})$