Binary Tree Fun

Your assignment is to work with binary trees (using the provided generic BTNode class) and implement several recursive methods. Each method operates on a binary tree and either returns information about the tree, produces a transformed version, or both. The following sections explain each method, with a simple “diagram” (sketch) to help you understand what is happening at each step.

1. copy

Purpose:
Create a new tree that is an exact duplicate of the given tree.
What to Do:

- Traverse the tree recursively.
- For each node, create a new node with the same data.
- Recursively copy the left and right children.

How It Works:

- **Base Case:** If the node is `null`, return `null`.
- **Recursive Case:** Create a new node and set its left child to the copy of the current node’s left and the right child to the copy of the current node’s right.

Diagram:

```
Original Tree:          Copy Process:
      [A]                Create new [A]
     /   \              /           \
   [B]   [C]   ===>   copy(B)     copy(C)
```

Figure 1. The original tree’s root “A” is copied and then the left and right subtrees are recursively copied.

2. replace

Purpose:
Replace every occurrence of a specific value (`oldValue`) in the tree with a new value (`newValue`). Return the number of replacements made.

What to Do:

- Traverse the tree recursively.
- At each node, check if its data equals `oldValue` (use the `.equals()` method for correct comparison).
- If it does, update the data to `newValue` and increment a counter.
- Do not create new nodes; modify the existing tree.

Diagram:

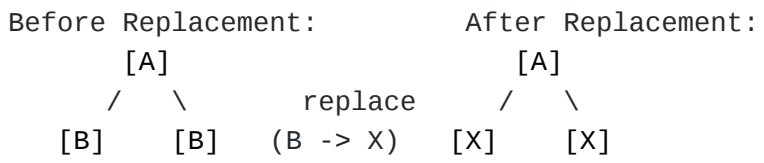

Figure 2. All nodes with value "B" are replaced with "X".

3. countNodesAtDepth

Purpose:
Count the number of nodes present at a specified depth in the tree.
What to Do:

- Define the root as depth 0.
- Traverse the tree and decrease the target depth at each level.
- When the target depth is 0, count that node.

Diagram:

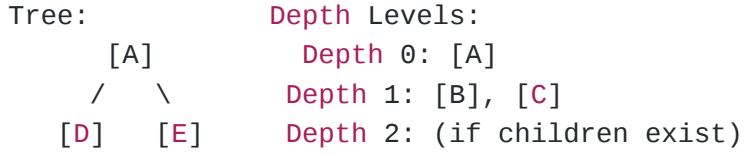

Figure 3. For depth 1 in this tree, there are 2 nodes ([B] and [C]).

4. allSame

Purpose:
Determine if all the nodes in the tree have the same value.

What to Do:

- If the tree is empty or has only one node, return `true`.
- Otherwise, recursively check that each child's data equals the parent's data.
- If any child's value is different, return `false`.

Diagram:

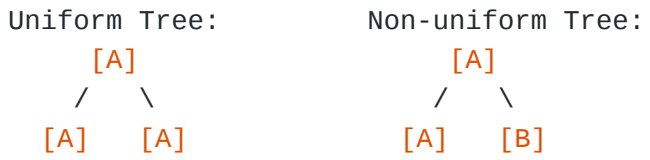

Figure 4. The left tree returns true, while the right returns false because "B" differs from "A".

5. leafList

Purpose:
Collect the data values from all leaf nodes (nodes with no children) into a List.
What to Do:

- Traverse the tree and check if a node is a leaf (both left and right are `null`).
- Add the leaf's data to a list.
- Maintain the order of discovery (typically left-to-right).

Diagram:

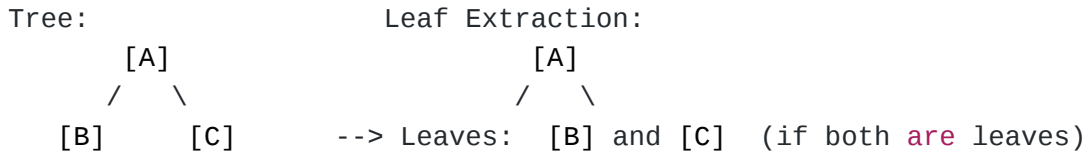

Figure 5. Leaves (nodes with no children) are added to the list.

6. reflect

Purpose:
Reflect (mirror) the tree horizontally.
What to Do:

- At every node, swap its left and right children.

- Apply this swap recursively throughout the tree.
- Do not create any new nodes; modify the tree in place.

Diagram:

Figure 6. The left and right children of each node are swapped, mirroring the tree.

7. condense

Purpose:
Remove any node that has exactly one child so that the resulting tree only contains nodes that either have two children or are leaves.
What to Do:

- Recursively traverse the tree.
- For any node with exactly one child, remove that node and replace it with its sole child.
- This “condensation” should be done in place, and you should return the modified tree’s root.

Diagram:

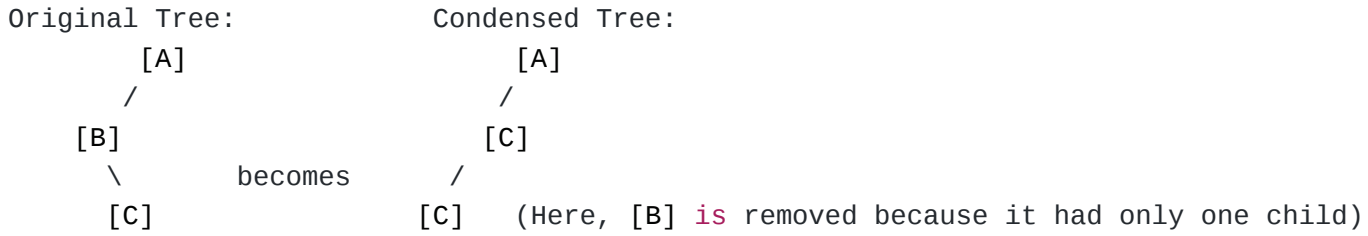

Figure 7. A node with only one child (here, [B]) is removed, linking its child ([C]) directly to its parent.

Happy coding!