Assignment 2: Colour Trees

All submitted work must be *done individually* without consulting someone else's solutions in accordance with the University's Academic Dishonesty and Plagiarism policies.

Story

We are asked to design a tree capable of storing a certain propagating colour. Which colour is propagated is determined by the colours sorted in its subtree. There is a hierarchy of colours, and an ordering exists to determine which is the dominating colour that will be propagated. This tree is used to do some elemental property checking to run some "what if" scenarios.

Informally, our implementation should support the following:

- Update the colour of a node.
- Insert a new node.
- Swap two subtrees. This is intended to help us answer questions such as "What changes to the propagating colors if the descendants were different?".
- **Property checking**: Given a node, does every descendant (up to k levels deeper) of this node have a certain color.

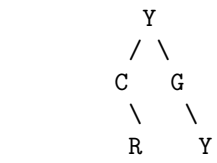
About the tree

The tree contains a colour propagation, based on the hierarchy:

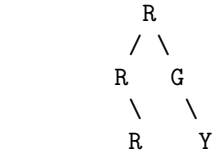
1. RED (R)
2. GREEN (G)
3. BLUE (B)
4. CYAN (C)
5. YELLOW (Y)

With **RED** being the strongest colour.

The tree:



Will produce the propagations:



Code

You are asked to implement 2 major files, `node.py` and `tree.py`.

node.py

This is the implementation of an element in the tree, or a node of the tree.

Properties

- `colour` - The colour of the node.
- `propagated_colour` - The colour that has been propagated.
- `parent` - A pointer to the parent node, `None` if it is the root.
- `children` - A list of pointers to the children of this node.

Functions set_parent(self, parent: 'Node') -> None

Sets the parent of the node, takes a parent Pointer (type Node).

update_colour(self, colour: Colour) -> None

Updates/Changes the colour of the node, by setting the `colour` attribute and performing any relevant calculations for that node.

add_child(self, child_node: 'Node') -> None

Adds the `child_node` to the list of children for the node.

remove_child(self, child_node: 'Node') -> None

Removes the `child_node` from the list of children of this node.

Tree

This is the tree implementation, and where most of the interaction will come from.

Properties

- `root` - A pointer to the root node of the tree.

Functions update_node_colour(self, n: Node, new_colour: Colour) -> None

Updates the colour of the given node `n` with the colour `new_colour`. Should also perform any *relevant actions* relating to the tree.

put(self, parent: Node, child: Node) -> None

Adds a node into the tree, by adding `child` to `parent`. This should also perform any relevant calculations that are required.

rm(self, child: Node) -> None

Removes the child from the tree. **PLEASE NOTE** if this node has children, they are removed at the same time, we can consider this entire subtree rooted at this `child` node removed from the tree.

swap(self, subtree_a: Node, subtree_b: Node) -> None

Swaps the tree rooted at `subtree_a` with the tree rooted at `subtree_b` and performs any relevant calculations.

is_coloured_to_depth_k(self, start_node: Node, colour: Colour, k: int) -> bool

Perform the property checks to see whether all nodes in the subtree (up and including level `k` starting from the start node) have the same colour!

Testing

We have provided you with some test cases in the `tests` directory of this repository. We will be using unit tests provided with the `unittest` package of python.

Running Tests

From the base directory (the one with `node.py` and `tree.py`), run

```
python -m unittest -v tests/test_sample_tree.py tests/test_sample_node.py
```

Or, running all the tests by:

```
python -m unittest -vv
```

Marking

You will be marked using a range of public and hidden tests. There will **not** be additional tests added after the due date.

All tests are **between 1 to 5 marks each**.