Assignment 2 – Baby Mobiles

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 test whether mobiles designed for baby cribs are balanced or not. We model a mobile as a binary tree and each node stores the weight (a non-negative integer) of the component of the mobile that it corresponds to. We define the *imbalance* of a node as the absolute difference between the summed weight of its left and right subtrees. An empty subtree has weight 0.

Informally, our implementation should support the following operations:

- Update the weight of a node.
- Insert a new node.
- Move the subtree rooted at a given node. This allows us to play around with the design of the mobile.
- Find the most unbalanced node, i.e., the one with the largest imbalance. This is an important query, since if the maximum imbalance is too high, the mobile might not be stable enough, which could be harmful to the baby.

Your Task

Maintain a **tree** where each node in the tree holds a **weight** value as its **key**, as well as the property **imbalance**. The value of the "**imbalance**" is equivalent to the absolute difference between the summed weight of the left and right subtree of this node.

Example:

```
Tree:

  A(5)

   / \

 C(2) D(8)

  /

B(10)
```

```
Imbalance:

  A(4)

   / \

C(10) D(0)

  /

 B(0)
```

Move Subtree

You must also support the `move_subtree(node_a, node_b, left_child)` function, where the subtree rooted at `node_a` is made into a child of `node_b`. `left_child` is a boolean value determining if `node_a` is the left child of `node_b` or its right child. If `node_b` already has a child as the intended position of `node_a`, your function should do nothing. You must ensure that the imbalance property is correct for all nodes, after moving the subtree. You can assume that `node_b` isn't in the subtree of `node_a` and you don't have to check for this.

Example:

```
  A(5)

   / \

 C(2) D(8)

  /

B(10)


move_subtree(D,C,false):

  A(5)

   /

 C(2)

  /  \

B(10) D(8)
```

```
Imbalance:

  A(20)

   /

 C(2)

  /  \

B(0) D(0)
```

Find Max Imbalance

Your tree must support the `find_max_imbalance()` operation that returns an integer. When called, the function returns the maximum imbalance in the tree.

Example:

```
  A(5)

   / \

 C(2) D(8)

  /

B(10)


find_max_imbalance() returns 10
```

You should strive to make your implementation as efficient as possible. Note that while we don’t explicitly test for the efficiency of your implementation, *using inefficient implementations may cause timeouts on Ed.*

TO IMPLEMENT:

You will need to implement these functions:

`node.py`

- `__init__`
- `add_left_child, add_right_child`
- `update_weight`

`tree.py`

- `put`
- `move_subtree`
- `find_max_imbalance`

(Note, you can add additional functions and variables to the classes if required, so feel free to modify and extend those as long as you leave the existing function signatures and variables intact.)

Code

`node.py`

This file holds all information about the node in the tree.

Properties

Name	Type	Description
left_child	*Node	Pointer to the left child
right_child	*Node	Pointer to the right child
parent	*Node	Pointer to the parent
weight	int	Weight of the node
imbalance	int	Absolute different between the weight of the left and right subtree

Functions

`__init__(weight)`

- Initialises the properties of the node.

`add_left_child(child_node)`

- Adds the child node as the left child of the node, does nothing if the node already has a left child.
- Runs calculations for updating the imbalance.

`add_right_child(child_node)`

- Adds the child node as the right child of the node, does nothing if the node already has a right child.
- Runs calculations for updating the imbalance.

`is_external()`

- Checks if the node is a leaf.

`get_left_child() (OR node.left_child)`

- Returns the left child of that node.

`get_right_child() (OR node.right_child)`

- Returns the right child of that node.

`update_weight(weight)`

- Sets the weight of the node.
- Runs calculations for updating the imbalance.

`get_imbalance() (OR node.imbalance)`

- Returns the imbalance of that node.

`tree.py`

The main tree file, holds all interaction with trees and nodes.

Properties

Name	Type	Description
root	*Node	Root node of the tree

Functions

`put(node, child, left_child)`

- Add the child as the left or right child (depending on `left_child`) to the node, does nothing if node B also has a child there.

`move_subtree(node_a, node_b, left_child)`

- Move node A to become the left or right child (depending on `left_child`) node B, does nothing if node B also has a child there.
- Runs calculations for updating the imbalance.

`find_max_imbalance()`

- Returns the maximum imbalance of the tree.

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 won't be additional tests added after the due date.

All tests are between 1 to 4 marks each.