# Red-Black Tree Implementation Assignment

## Assignment Tasks Overview

### 1. Insertion

The insertion algorithm ensures that the red-black tree properties remain intact after adding a new value.

#### `balance` function

- Balances the tree during insertion by performing rotations or recoloring as needed.
- **Input**: Node information (color, value, left subtree, right subtree).
- **Output**: A balanced red-black subtree.
- **TODO**: Implement pattern matching for the four rebalancing cases discussed in the Insertion Explanation.

#### `insert` function

- Inserts a value into the red-black tree recursively. After each insertion, the tree is balanced using the `balance` function.
- Ensures the root node is always black after insertion.
- **TODO**: Implement the recursive helper function `insert_aux` and the logic for ensuring the root's color.

#### `insert_tr` function

- Implements insertion in a tail-recursive manner for better efficiency.
- **TODO**: Implement this as an alternative to `insert`.

Refer to the **Insertion Explanation** for a detailed breakdown of the algorithm.

---

### 2. Deletion

The deletion algorithm ensures that red-black tree properties remain intact after removing a node. The key steps are: 1. Find the node to delete. 2. If the node has two children, replace it with the minimum value from the right subtree. 3. Rebalance the tree after deletion using the `balance_delete` function.

#### `balance_delete` function

- Balances the tree after deletion to maintain red-black properties.
- **Input**: Node information (color, value, left subtree, right subtree).
- **Output**: A balanced red-black subtree.
- **TODO**: Implement pattern matching for rebalancing cases.

#### `delete` function

- Implements the full deletion algorithm, including:
  - Finding the node.
  - Handling special cases (e.g., one child or no children).
  - Calling `balance_delete` as needed.
- **TODO**: Implement logic for deleting nodes and rebalancing the tree.
- Use the `min_value_node` helper function to find the smallest value in the right subtree when replacing a node.

Refer to the **Deletion Explanation** for a detailed breakdown of the algorithm.

---

## 3. Validation

To ensure that a red-black tree satisfies all its properties, you will implement the `is_valid` function. This function verifies: 1. All paths from a node to its leaves have the same number of black nodes (black-height). 2. No red node has red children.

#### `black_height` function

- Computes the black height of the tree and checks that it is consistent for all paths.
- **TODO**: Complete the recursive logic to return `0` if a violation is found or the correct black height otherwise.

#### `no_red_with_red_child` function

- Checks that no red node has a red child.
- **TODO**: Complete the recursive logic to traverse the tree and verify this property.

#### `is_valid` function

- Combines the results of `black_height` and `no_red_with_red_child` to determine if the tree is valid.
- **TODO**: Use the helper functions to return `true` if the tree satisfies all properties or `false` otherwise.

Refer to the **Validity Explanation** for a concise breakdown of the requirements.