Lab: Red-Black Trees and AA-Trees Lab

This document defines the lab for "Data Structures – Advanced (Java)" course @ Software University.

Please submit your solutions (source code) of all below described problems in Judge.

Write Java code for solving the tasks on the following pages. Code should compile under the Java 8 and above standards you can write and locally test your solution with the Java 13 standard, however, **Judge will run the submission with Java 10 JRE**. Avoid submissions with **features included after the Java 10** release doing **otherwise** will result in a **compile time error**.

Any code files that are part of the task are provided as **Skeleton**. In the beginning, import the project skeleton, do not change any of the interfaces or classes provided. You are free to add additional logic in the form of methods in both interfaces and implementations you are not allowed to delete or remove any of the code provided. Do not change the names of the files as they are part of the test logic. **Do not change the packages** or move any of the files provided inside the skeleton if you have to add a new file add it in the same package of usage.

Some **tests may be provided** within the skeleton – use those for local **testing and debugging**, however, there **is no guarantee that there are no hidden tests added inside Judge**.

Please follow the exact instructions for uploading the solutions for each task. Submit as **.zip archive** the files contained inside **"...\src\main\java"** folder this should work for all tasks regardless of the current DS implementation.

In order for the solution to compile the tests **successfully** the project **must** have a **single Main.java** file containing a single **public static void main(String[] args)** method even an empty one within the **Main class**.

Some of the problems will have simple **Benchmark tests** inside the skeleton. You can try to run those with **different values** and **different implementations** in order to **observe** behavior. However, **keep** in mind that the result comes **only as numbers** and this data may be **misleading** in some situations. Also, the tests are not started from the command prompt which may **influence** the **accuracy** of the results. Those tests are only added as an **example** of **different data structures performance** on their **common** operations.

1. Check Red Node

Add the following constants to your **`RedBlackTree`** class:

```
private static final boolean RED = true;
private static final boolean BLACK = false;
```

Now create a helper method that will check if a node is red:

```
private boolean isRed(Node<T> node) {
    if (node == null) {
        return false;
    }
    return node.color;
}
```

2. Change the Node Data Structure

First, you will need to add a **color bit** to our node class:

```
public static class Node<T> {
    private T value;
    private Node<T> left;
    private Node<T> right;
    private boolean color;
    private int count;

    public Node(T value) {
        this.count = 1;
        this.value = value;
        this.color = RED;
    }

    public Node(T value, boolean color) {
        this.count = 1;
        this.value = value;
        this.color = color;
    }
}
```

3. Left Rotation

Create a method that will accomplish the left rotation for a given node.

```
private Node<T> rotateLeft(Node<T> node) {
    Node<T> temp = node.right;
    node.right = temp.left;
    temp.left = node;
    node.color = RED;
    temp.color = BLACK;
    node.count = 1 + this.getNodesCount(node.left) + this.getNodesCount(node.right);
    return temp;
}
```

4. Right Rotation

Create a method that will perform right rotation on a given node. The code is similar to the left rotation.

```
private Node<T> rotateRight(Node<T> node) {
    Node<T> temp = node.left;
    node.left = temp.right;
    temp.right = node;
    temp.color = node.color;
    node.color = RED;
    node.count = 1 + this.getNodesCount(node.left) + this.getNodesCount(node.right);
    return temp;
}
```

5. Flip Colours

Implement a method that will make a node "**black**" and its children "**red**".

```
private void flipColors(Node<T> node) {
    node.color = RED;
    node.left.color = BLACK;
    node.right.color = BLACK;
}
```

6. Insert

Implement the existing **`insert()`** method. It should create new **red** node for every insert, **balance** the tree and **recolour** the nodes if needed.

Hint: There are hints inside the presentation.

7. Run Unit Tests