Exercise: Red-Black Trees and AA-Trees

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 **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. Red-Black Tree

You are given class **RedBlackTree** your task is to implement all the methods with missing implementation:

- **`int size()`** – returns the **size** of the tree
- **`boolean isEmpty()`** – returns **whether** the tree is empty (the root is null)
- **`Value get(Key key)`** – returns the **value** of the specified key otherwise returns null
- **`boolean contains(Key key)`** – returns **whether** the key is present
- **`void put(Key key, Value val)`** – adds **new entry** to the tree if the key
- **`void deleteMin()`** – removes the **min** element by key in the tree
- **`void deleteMax()`** – removes the **max** element by key in the tree
- **`void delete(Key key)`** – we can proceed in a similar manner to **delete** any node that has one child (or no children), but what can we do to delete a node that has two children? We are left with two links but have a place in the parent node for only one of them. An answer to this dilemma, first proposed by T. Hibbard in 1962, is to delete a node x by replacing it with its successor. Because x has a right child, its successor is the node with the smallest key in its right subtree. The replacement preserves order in the tree because there are no keys between x.key and the successor's key. We accomplish the task of replacing x with its successor in four easy steps:

    - **Save** a link to the node to be deleted in t
    - **Set** x to point to its successor **min** (t.right).
    - **Set** the right link of x (which is supposed to point to the BST containing all the keys larger than x.key) to **deleteMin** (t.right), the link to the BST containing all the keys that are larger than x.key after the deletion.
    - **Set** the left link of x (which was null) to t.left (all the keys that are less than both the deleted key and its successor).
- `int height()` – returns the **height** of the tree **-1** for empty **0** for single element (assume it is zero based)
- `Key min()` – returns the **min** key
- `Key max()` – returns the **max** key
- `Key floor(Key key)` - If a given key is less than the key at the root of a BST, then the floor of key (the largest key in the BST less than or equal to key) must be in the left subtree If key is greater than the key at the root, then the floor of key could be in the right subtree, but only if there is a key smaller than or equal to key in the right subtree if not (or if key is equal to the key at the root) then the key at the root is the floor of key.
- `Key ceiling(Key key)` – finding the ceiling is similar to floor, interchanging right and left
- `Key select (int rank)` – suppose that we seek the key of rank **k** (the key such that precisely **k** other keys in the BST are smaller). If the number of keys **t** in the left subtree is larger than **k**, we look (recursively) for the key of rank **k** in the left subtree; if **t** is equal to **k**, we return the key at the root; and if **t** is smaller than **k**, we look (recursively) for the key of rank **k - t - 1** in the right subtree.
- `int rank(Key key)` – if the given key is equal to the key at the root, we return the number of keys **t** in the left subtree; if the given key is less than the key at the root, we return the rank of the key in the left subtree; and if the given key is larger than the key at the root, we return **t** plus one (to count the key at the root) plus the rank of the key in the right subtree.
- `Iterable<Key> keys()` – returns **all** the keys inside the tree
- `Iterable<Key> keys(Key lo, Key hi)` – returns the keys from **lower** bound to upper **bound** inside the tree
- `int size(Key lo, Key hi)` – returns the **number** of **keys** between the lower and upper bound

Hints: There are some private methods that should help try to implement them, however, the tests won't require any of them.

2. AA Tree

You are given class **AATree** your task is to implement all the methods with missing implementation, it should be pretty simple:

- `boolean isEmpty()` – returns **whether** the tree is empty or not
- `void clear()` – **clears** the tree and makes it empty
- `void insert(T element)` – **adds** an element
- `int countNodes()` – returns the **number** of nodes
- `boolean search(T element)` – returns **whether** the element is present or not
- `void inOrder(Consumer<T> consumer)` – the **consumer accepts** all the elements **in order** traversal
- `void preOrder(Consumer<T> consumer)` – the **consumer accepts** all the elements **pre order** traversal
- `void postOrder(Consumer<T> consumer)` – the **consumer accepts** all the elements **post order** traversal