TD7 - Pale machine (midterm)

INF411

> Before starting :
>
> - Create a new project “TD7”.
> - Download TD7.zip and extract the files in the project folder.

Remark: the exercices 1 and 2 are independent.

***The Java documentation** is available via the provided link on the french webpage*

1 Images and hash functions

The goal of this exercise is to implement an efficient solution for removing duplicates in a collection of input images.

We will deal with binary images (pixels are either blakc or white) that we will represent with the class `BinaryImage`.

```
class BinaryImage {
  public static final boolean WHITE = false;
  public static final boolean BLACK = true;

  /** 2D array representing the image pixels */
```

```java
    final boolean[][] pixels;

    /** Create an image of n rows and m columns */
    BinaryImage(int n, int m) {
      assert n >= 1 && m >= 1;
      this.pixels = new boolean[n][m];
    }
  }
```

In order to efficiently check the existence of duplicates we will make use of hash tables. We remind you the existence of the class `HashSet<K>` provided by Java, allowing us to store a *set* of elements whose type is `K`.

1.1 Checking equality between two images

In the class `BinaryImage`, implement method `boolean equals(Object o)` which returns `true` if the current image if equal to the image `o` (the color of their pixels do coincide), `false` otherwise.

Test your code with the class `Test11`.

Submit the file `BinaryImage.java` via the form on the french webpage.

1.2 Removing duplicates

In the class `BinaryImage`, complete method `int hashCode()` which return the hash value of the image. Two images that are equal should have the same hash value.

In the class `BinaryImage`, complete method `BinaryImage[] deleteDuplicates(BinaryImage[] t)` which takes as input an array `t` of images and returns a new array without duplicates. More precisely is the array `t` has size `k` and contains `d` distinct images then the output array will have size `d` and it will do not contain any duplicate.

> *Suggestion : the class `HashSet<K>` provides an implementation, based on hash tables, of a* set *storing elements of type `K` (in particular, this class provides methods for adding a new element and checking whether a given element does exist in the set).*

The your code with the class `Test12`.

Submit the file `BinaryImage.java` via the form on the french webpage.

2 Red-Black binary trees

In this exercise we consider *red-black trees* which are a variant of binary search trees allowing to get well balanced trees: the height of a red-black tree containing n nodes is $O(\log_2 n)$.

A *red-black tree* is defined as binary search tree whose node have colors (red or black) satisfying the conditions below:

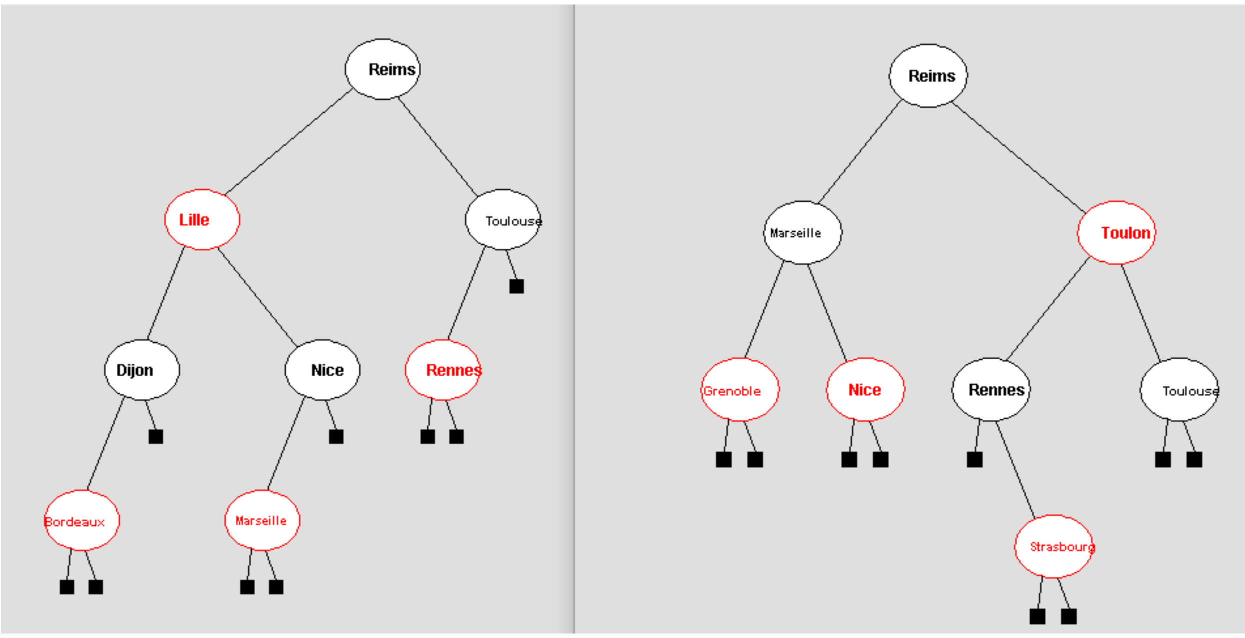
1. the root node is black and all leaves are black,
2. the children of a red nodes are black,
3. for each node v in the tree, all paths from v to the leaves contain the same number of black nodes.

```
class RBT extends IntegerPoint2D {
  final static boolean RED = true;
  final static boolean BLACK = false;

  boolean color;
  RBT left, right;
  String element;

  /* Construct a node of a Red-Black tree */
  RBT(boolean color, RBT left, String element, RBT right) {
    this.color = color;
    this.left = left;
    this.element = element;
    this.right = right;
  }
}
```

A tree (of a sub-tree) is represented by its root using the class `RBT` (*Red-Black Tree*). A node contains a field `String element` (the data to store), and tow references to the left and right children. Each node also stores the information concerning its color (field `boolean color`). The colors *red* and *black* are encoded by two constants `final static boolean RED` and `final static boolean BLACK`.

As already done (see TD6 and related lecture), the *empty tree* is represented by the `null` reference. The leaves (which must be black) are represented by sentinel nodes, encoded by `null` (please refer to small black squares in the image below).

Two examples of Red-Black trees:



For the debugging of your code it could be useful to get a visual layout of a given tree `t`. It suffices to use the instruction

```
new DrawBinaryTree(t);
```

which computes a visual layout of the tree as depicted in the images aboe.

The main goal of this exercise is to implement a function for checking whether an input tree is a valid red-black tree (we assume that the tree is a valid binary search tree).

2.1 Validity for red nodes

In the class `RBT`, complete method `boolean isRedValid(RBT t)` which returns `true` if the tree `t` satisfies condition (2) for all red nodes.

The your code with the class `Test21`.

Submit the file `RBT.java` (go to the french webpage).

2.2 Validity for black nodes

In the class `RBT`, complete method `boolean isBlackValid(RBT t)` which returns `true` if the tree `t` satisfies condition (3) of the definition given above.

Test your code wih the class `Test22`.

Submit `RBT.java` (go to the french webpage).

2.3 Validity of a Red-Black tree

In the class `RBT`, complete method `boolean isValid(RBT t)` which returns `true` if the tree `t` is a red-black tree.

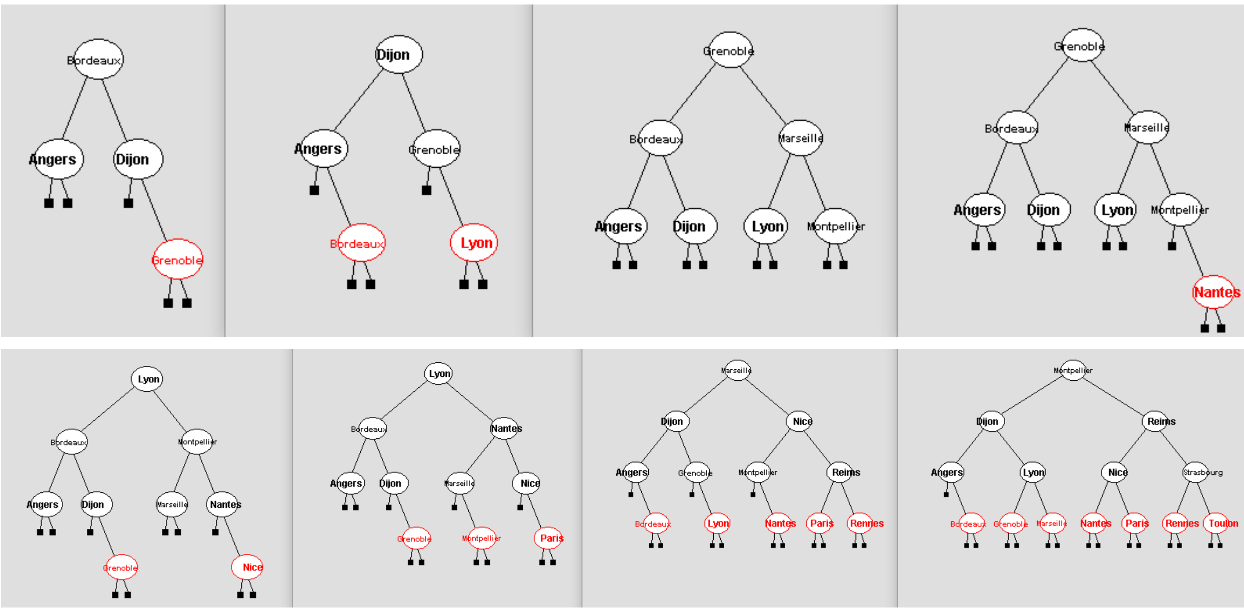
Test your code wih the class `Test23`.

Submit `RBT.java` (go to the french webpage).

2.4 Construction of a tree from a sorted list: height computation

In order to construct a red-black tree starting from a *sorted list* a simple solution is to proceed in the following way (illustrated by the pictures below). First we estimate the height of the red-black tree (assuming that we seek for a perfect balance). Then we construct a binary search tree whose nodes are all black: the only exception is represented by nodes in the last level (the one that could be not full) which could be red.

A few examples of red-black trees of different sizes are depicted below:



In the class `RBT`, complete method `int estimateBlackHeight(int n)` which returns the integer $h \geq 0$ satisfying (for $n \geq 0$) :

$$2^h - 1 \leq n < 2^{h+1} - 1$$

Remak : the function above allows to estimate the number of full *levels in the tree. We assume that for the empty tree the function above returns 0 (no inner nodes) and for the tree of size 1 (only one inner node, the root) it returns 1.*

Test your code wih the class `Test24`.

Submit `RBT.java` (go to the french webpage).

2.5 Construction of the tree starting from a sorted list

In the class `RBT`, complete method `RBT ofList(LinkedList<String> l)` which returns a red-black tree containing all elements of the list `l` as described above.

Remarke : the function `ofList(LinkedList<String> l)` is allowed to modify the list `l` (if this is useful for you).

Test your code wih the class `Test25`.

Submit `RBT.java` (go to the french webpage).