Assignment 5 **Julien Corman, Tiziano Dalmonte, Werner Nutt**

Binary Trees and Priority Queues

This coursework has two parts, the first consists in the implementation of standard operations on binary trees, the second in the implementation of priority queues using binary trees. For Part 1 of this coursework, you are only asked to write code, for Part 2 you are expected to both write code and submit a report. The deadline is Monday, 10 January 2021, which means that you will have two weeks of lecture period to finish this coursework.
The modalities for code submissions are:

- Class and method stubs are available on Codeboard. Your implementation must respect their signatures (but of course, you are free to create and call additional methods if needed).
- The code must be submitted via Codeboard, and a copy of it (zipped .java files) via OLE.
- Use only data structures and methods known from the course.

See the coursework as a suggested way to review the content of the lectures for which you will get feedback.

1. Operations on Binary Trees

We assume that binary trees with positive integer keys are implemented by the classes `Tree` and `Node`, defined as follows:

```
public class Tree{
   Node root;

   static class Node{
     int key;
     Node left, right}

} ,
```

Note that we do not assume the presence of parent pointers.

In this exercise, you are asked to develop additional methods on trees for this class, which are specified below.
Some methods produce lists. These should be simple linked lists, which were introduced in the lecture, and which you used in Assignment 4. You will find a class stub for these on Codeboard.

1. `int size()`: returns the number of nodes in the tree.
2. `Tree copy()`: returns a tree that is a copy of the original tree, with new nodes, but the same structure and the same keys.
3. `List keys()`: returns a simple linked list containing all keys (also duplicates) occurring in the tree, in the order of inorder traversal.

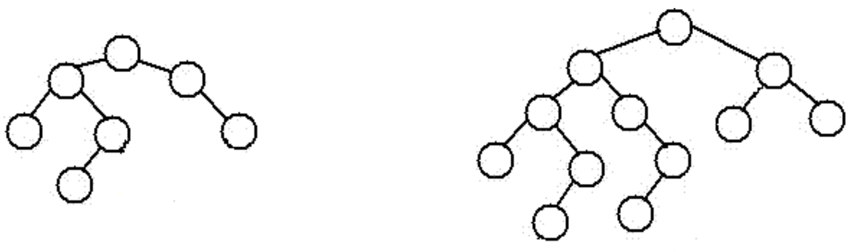
   Hint: Make sure you need not reverse the list.
4. `List keysAtDepth(int d)`: returns a simple linked list with the integers that occur at depth `d` in the tree (the root node has depth 0, its children have depth 1, etc.), ordered as they appear from left to right in the tree; if there are no nodes at depth `d`, returns an empty list.

   Hint: Make sure that the algorithm is as efficient as possible, that is, looks only at nodes with depth at most `d` and restricts list operations to one insertion per node at depth `d`.
5. `int height()`: returns the height of the tree.

   Recall that the height of a node in a tree is the length of the longest path from a leaf to that node (as a special case, we define the height of a null node as -1.) The height of a tree is the height of the root.
6. `boolean isHeightBalanced()`: returns `true` if the tree is height-balanced, and `false` otherwise.

   We say that a node `n` is *height-balanced* if it is null or, otherwise, if (i) the left child `n.left` and the right child `n.right` are height-balanced and (ii) the absolute difference between the height of its right and left children is at most 1. We say that a tree is height-balanced if the root is height-balanced.

   Consider the following example:

   Here, the first tree is height-balanced, but the second tree is not. In the second tree, the height of the left child of the root is 3, whereas the height of the right child of the root is 1, hence the absolute difference is $|3 - 1| = 2$.

7. `boolean isBST()`: returns `true` if the tree is a binary search tree (BST), that is, if for each node `n` in the tree, it is the case that all keys in the left subtree of `n` are less or equal `n.key`, while all keys in the right subtree are greater or equal `n.key`.

8. `int floor(int x)`: the behaviour of the method is only defined for binary search trees, if the tree is not a search tree any result is acceptable; if applied to a binary search tree, it returns an integer that is maximal among all keys of nodes `n` with `n.key` $\leq$ `x`; if there is no such node, it returns the minimal integer that exists in Java (that is `Integer.MIN_VALUE`).

   Provide a recursive implementation for `floor`. Then develop an iterative version, called `floorIter`.

Task:

- Implement the methods in Java, as part of the class `Tree` (and possibly `Node`).

(Weight: 60% of this CW)

2. Priority Queue Realized with Binary Trees

In the lecture, we have introduced priority queues as an abstract data type that supports the operations

- `void insert(int i)`
- `int extractMax().`

With `insert`, one inserts a new value into the queue. The method `extractMax` returns the maximal value currently in the queue and deletes that value from the queue. (In a more generic version of priority queues, which we do not want to implement in this assignment, the method `insert` would insert an object one of whose attributes is the key attribute for the queue. Similarly, `extractMax` would return an object with the maximal value and delete that object from the queue.)
One way to realize priority queues, is to use heaps based on arrays. The downside is that a queue may become "full" so that no new entry can be inserted.
In this exercise, you are asked to realize a priority queue with binary trees. The implementation is based on two classes, `PQueue` and `PNode`. A priority queue object has a pointer "root" to an element of class `PNode`, as follows:

```
public class PQueue{
   PNode root;

   public static class PNode{
     int key;
     PNode left, right, parent;
     int lcount, rcount;
   }

}
```

You will notice the two new fields `lcount` and `rcount`. These are counters that record the number of nodes in the left and right subtree of a node, respectively.
Using the above data structure, implement priority queues as binary trees that have the heap property: for each subtree, the key of the root is greater or equal all other values in the subtree. Your implementation should support the following methods:

1. `boolean isEmpty()`,
2. `void insert(int value)`: inserts a value into the queue.

   Requirements: insertion keeps the binary tree *size-balanced*, that is, for each node, the number of nodes in its left and right subtrees differ at most by one.

   Since each node holds information about the size of its left and right subtrees (`lcount` and `rcount`), you can decide into which subtree to insert a new node, while keeping the tree size-balanced. And since inserting a new value requires inserting a new node, you also have to update these counters when inserting.

   If you have found out where to insert the new node in your tree as a leaf, the insertion of the new value may violate the heap property. Use an appropriate technique to restore it.
3. `int extractMax()`: returns the maximum value in the queue and deletes it. If the queue is empty, throws a `RuntimeException`.

   Requirements: the maximum value in a heap is in the root. If you delete the root, this leaves a gap that needs to be filled. To do this, find a leaf to be dropped, store its key value into the root node, and delete the leaf. Make sure that the deletion does not violate the size balance property.

   The new value in the root may violate the heap property. Use an appropriate technique to restore it.

Tasks:

- Explain the ideas behind your algorithm. In particular, explain how you design the methods and which auxiliary methods you use.
- Implement the methods in Java, as part of the class `PQueue` (and possibly `PNode`).

(Weight: 40% of this CW)

Deliverables.

- Your implementation of the classes in Exercises 1 and 2.
- A brief (but informative) report on the ideas underlying your implementation of priority queues.

Combine all deliverables into one zip file, which you submit via the OLE website of the course.
Submission until Tuesday, 17 January 2023, 23:55, to Codeboard and the OLE submission page.