Assignment 7

Mouna Kacimi, Werner Nutt, Camilo Thorne

Abstract Data Types and Trees

1. Priority Queues

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 general 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. In this exercise, you are asked to realize a priority queue with binary trees. This implementation is based on two classes, `PQueue` and `Node`. A priority queue element has a point "root" to an element of class `Node`:

```
class PQueue{
   Node root;}
```

Nodes themselves are instances of the class `Node` defined as follows:

```
class Node{
   int key;
   Node left, right, parent;
   int lcount, rcount; // Number of nodes in the left
                       // and right subtree}
```

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 than all the value in the subtree".) Your implementation should support the following methods:

1. `bool isEmpty()`

2. `void insert(int value)`: inserts a value in the queue.
   Hints: Insertion should keep, as far as possible, the binary tree balanced. Each node holds information about how many nodes are there in its right and left subtrees. You can use this information to put a new node into the subtree that has fewer elements. As inserting a new value requires inserting a new node, when inserting you also have to update these counters. 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 the technique explained in the lecture to maintain it.

3. `int extractMax()`: returns the maximum in the queue and deletes it.
   Hints: The maximum value in a heap is in the root. If you delete the root, this leaves a hole that needs to be refilled. To do that, find a leaf to be dropped, put its key value into the root node, and delete the leaf. The new value in the root may violate the heap property. Use the technique from the algorithm "heapify" to correct such violations.

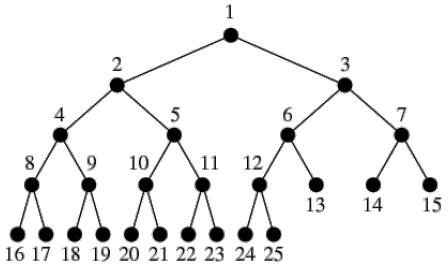
Tasks:

- Write pseudocode for the three methods and possibly for auxiliary methods that you will need.
- Implement the classes in Java.
- Test your implementation and report on the tests.

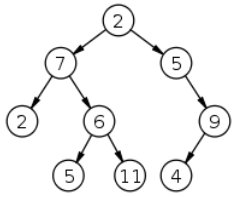
(15 Points)

2. Binary Search Tree Traversal

In this exercise we want to study methods to traverse binary trees. We first introduce a way to specify binary trees. We use the following schema for numbering the positions of a complete binary tree:



That is, the nodes are numbered as one would encounter them when traversing the tree one level after the other, from left to right.
Next, we want to modify the method `inOrderTreeWalk` of the lecture in such a way that it not only prints the key of a node, but also the position of the node. More specifically, the method prints a sequence of pairs *pos*:*key*, where *pos* is the position of the node printed and *key* the key. We call that variant method `inOrderTreeWalkPos`. When printing a tree in this way, we know its precise structure.
Consider, for example, the following binary tree:



For this tree, `inOrderTreeWalkPos` prints the sequence

1:2 2:7 4:2 5:6 10:5 11:11 3:5 7:9 14:4.

Now, consider classes `Tree` and `Node` that are like those introduced in the lecture, but where nodes do not have a parent pointer.

1. Design for the class `Tree` a method

   ```
   void inOrderTreeWalkPos()
   ```

   that prints the sequence of pairs defined above, consisisting of positions and keys.
   Hint: Remember how array positions represent the nodes of a heap.

2. Design a method

   ```
   void insertSorted(int i)
   ```

   that adds a node with key `i` to the current tree. If the current tree is sorted, then also the extended tree must be sorted.
   Hint: Have a look at the methods developed in the lecture.

3. Design a static method

   ```
   Tree constructBST(String filename)
   ```

that reads a sequence of integers from a file and constructs a binary search tree containing exactly those integers.
Hint: Make calls to the previous method.

4. Design a method

   ```
   void printBST(String filename)
   ```

   that prints the integers in the current tree onto a file and satisfies the following condition: if the output sequence is given as input to the algorithm `constructBST`, then the obtained tree is equal to the tree that was printed.
   Hint: The algorithm `InOrderTreeWalk` from the lecture does *not* have this property.

Tasks:

- Write pseudocode for the four methods and possibly for auxiliary methods that you will need.
- Implement the classes in Java.
- Test your implementation and report on the tests.

(15 Points)

Submission: Until Thu, 22 May 2014, 8:30 pm, to

`dsa-submissions AT inf DOT unibz DOT it.`