

Priority Queues and Operations on 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. If the queue is empty, it returns a default value or raises an exception, depending on the implementation. (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. The downside is that a queue may become “full” so that no new entries 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`:

```
class PQueue{
    PNode root;}

```

`PNodes` themselves are instances of the class `PNode` defined as follows:

```
class PNode{
    int key;
    PNode 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 into the queue.

Hints: Insertion should keep, as far as possible, the binary tree balanced. Each node holds information about how many nodes there are in its right and left subtrees. 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. Explain the ideas behind those methods.
- Implement the classes with their methods in Java.
- Test your implementation and report on the tests.

(15 Points)

2. Operations on Binary Trees

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

```
class Tree{
    Node root;}

class Node{
    int key;
    Node left, right, parent;} ,
```

together with the method `insert` from the lecture.

In this exercise, you are asked to develop additional methods on trees to this class.

1. `boolean balanced(int x)`: returns `true` if the tree is balanced and `false` otherwise.

For the balance check, we consider as leaves all `null` children of `Node` objects in the tree (as in the definition of red-black trees). A tree is *balanced* if no two such leaf nodes differ in distance from the root by more than one.

2. `Node floor(int x)`: if applied to a binary search tree, this method returns the rightmost node with respect to inorder traversal with a key that is less or equal to `x`; if no such node exists, it returns `null`; if the tree is not a search tree, we do not care what the method returns.

Provide two implementations, a recursive and an iterative one.

3. `List keysAtDepth(int d)`: returns a linked list with exactly those integers that occur at depth `d` in the tree, ordered as they appear from left to right in the tree. (If there are no nodes at depth `d`, the method returns an empty list.)

Make sure that the algorithm is as efficient as possible, that is, look only at nodes with depth at most `d` and restrict list operations to one insertion per integer to be inserted.

4. `static Tree buildFromArray(int[] A)`: returns a tree of minimal height whose nodes contain as keys exactly the entries of the array `A`, in the same order (with respect to inorder traversal) as in the array.

Tasks:

- Write pseudocode for the four methods and possibly for auxiliary methods that you will need. Explain the ideas behind those methods.

- Implement the methods within a Java class `Tree`.
- Test your implementation and report on the tests.

(15 Points)

Please, follow the “Instructions for Submitting Course Work” on the Web page with the assignments, when preparing your coursework.

Also, include name, student ID, code of your lab group (A, B, or C), and email address in your submission.

Submission: Until Mon, 25 May 2015, 11:55 pm, to

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