Program Design and Data Structures, 20.0 c

Course code: 1DL201, Report code: DL201, 67%, DAG, NML,
week: 44 - 02 Semester: Autumn 2017
week: 03 - 11 Semester: Spring 2018

LAB ASSIGNMENT 12

This information is not available in English. Now showing the Swedish version.

After this lab you should be able to ...

- Work with binary search trees.
- Work with red-black trees.

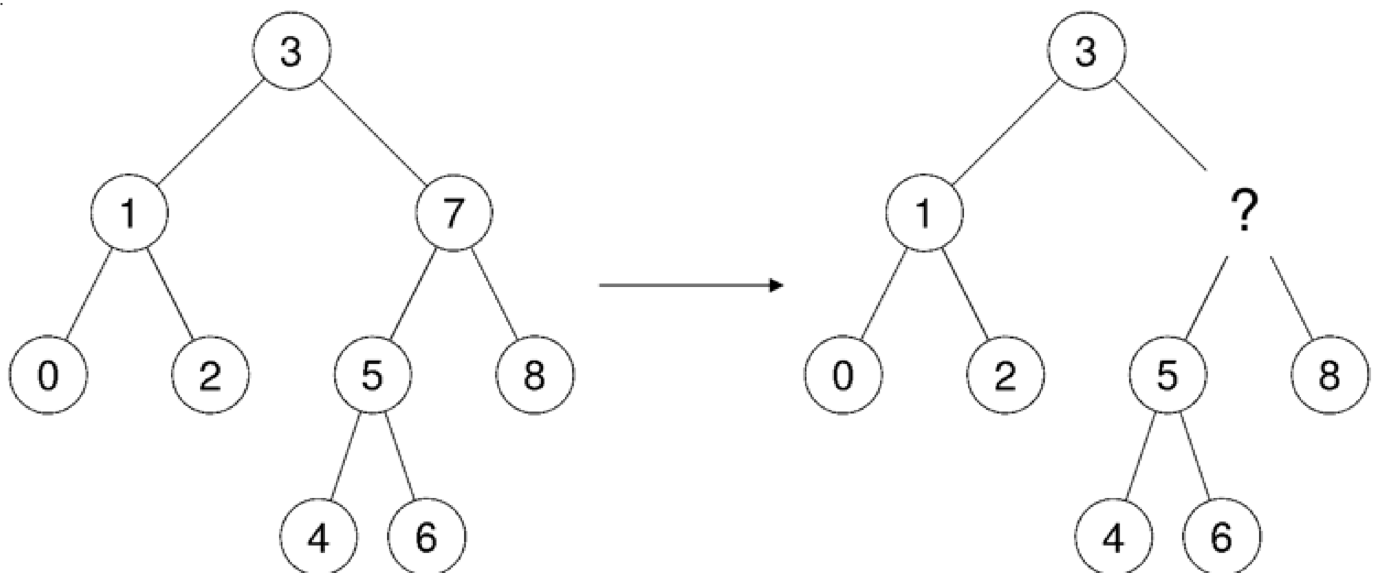
Instructions

- Read the slides on binary search trees (lecture 17, starting from slide 11) and on red-black trees (lecture 18).
- Remember to write function specifications for all functions that you write, and representation conventions/invariants for all data types that you define! Do this *before* you write your Haskell code. Also write recursion variants, which may be specified after the code has been written.
- Also remember to follow the other parts of our coding convention (for identifiers and indentation).

The task

PROBLEM 1

Deleting a leaf from a binary tree is straightforward. Also, deleting a node that has exactly one child is easy: we can simply replace the node with its child. However, when deleting a node with two children from a binary tree, we must rearrange the tree, since it is not possible to immediately combine the remaining parts into a new binary tree. For instance, consider deleting the node with label 7 in the figure below.

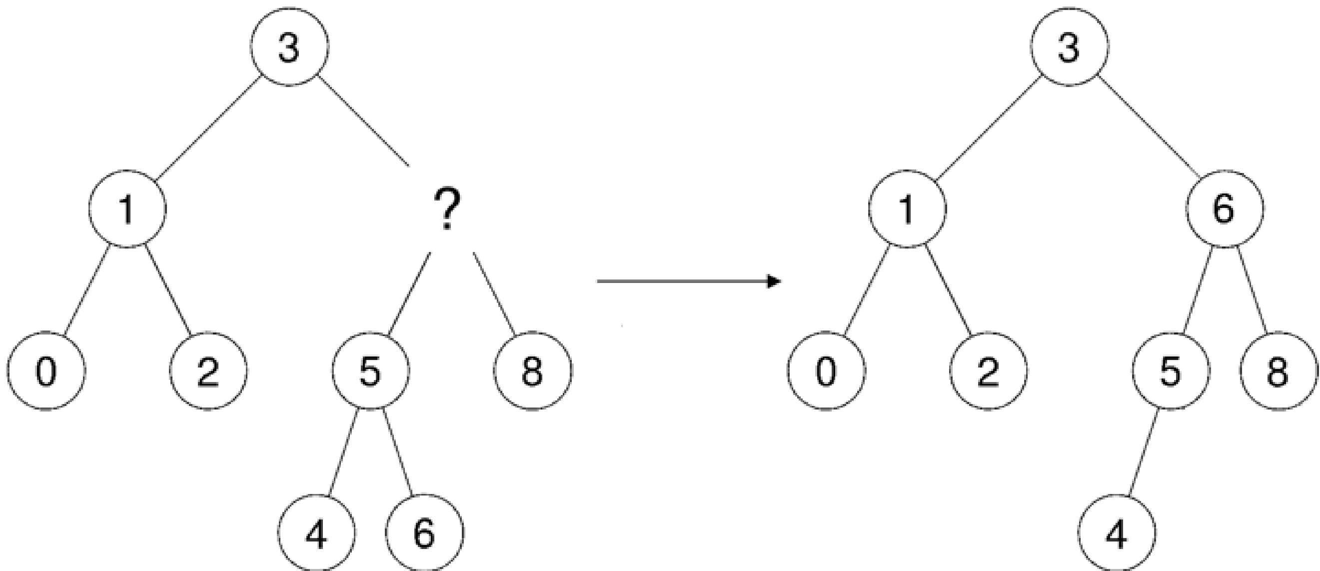


How to rearrange the tree after the deletion depends on what properties we want to preserve for the tree. For this problem, assume that the initial tree was a binary search tree, and that we want the result tree to be a binary search tree.

One way of achieving this is to use the following algorithm for deleting a node with two children:

1. Find the *largest* label in the tree that is *smaller* than the label of the node we want to delete. Let us call this label X.
2. Instead of deleting the node from the tree, replace its label with X.
3. Then, delete X from the node's left subtree.

In the above example, we have X=6, and the algorithm yields:



a) Write a function delete :: BSTree -> Int -> BSTree that implements the above algorithm to delete a label from a binary search tree. Here, binary search trees (i.e., type BSTree) are defined as in the lecture:

```
data BSTree = Void | Node BSTree Int BSTree   deriving (Show)
```

Hint: For step 1. of the algorithm, write an auxiliary function that efficiently finds the largest label in a (non-empty) binary search tree.

b) Test your function by deleting the label 7 from the binary search tree that is depicted on the left in the upper figure. (Optionally: perform this test using HUnit.)

PROBLEM 2

a) Starting with an empty red-black tree, insert - one at a time - the labels 2, 3, 10, 15, 12, 6 and 7. Use the algorithm from lecture 23. (Remember that rebalancing is performed at every level of the recursive insertion algorithm if necessary, i.e., at every node along the path from the newly inserted node back to the tree's root node. The top node in a rebalancing step does *not* need to be the root node of the tree.)

Show what the tree looks like after every insertion of a new node, and after every step in rebalancing, similar to the example on slides 14/15 from lecture 23.

If you do not have a red pen, draw red nodes *without a circle* around their label. (This will make it easier to distinguish between red and black nodes.)

b) Check after every insertion (including rebalancings) that the resulting tree satisfies the invariants for red-black trees, that is:

- The tree is a binary search tree.
- No red node has a red parent.
- Every path from the root to an empty tree contains the same number of black nodes.

Write the number of black nodes on the paths from the root to any empty tree above the tree. (You do not have to present your check of the other two invariants - but make sure they do indeed hold).

PROBLEM 3

When a rebalancing step has been performed, new red nodes can arise such that rebalancing has to be done once more. How can we be sure that the repetition of rebalancing does indeed end, i.e., that we do not need an infinite number of rebalancings?

WHEN YOU ARE DONE

When you are done with all problems (the explorations below are optional), raise your hand or approach an assistant to have your solution graded.

If you pass this lab at least 30 minutes early and other groups are still working on it, we ask you to **help one other group**. Do not simply share your solution with them, but try to understand the (partial) solutions that they have developed so far (which may be different from yours) and the difficulties that they have. Assist them in coming up with their own solutions. Once the group that you are helping completes the lab assignment, you may leave. (If it is still early, the group that received your help should then stay to help another group.)

EXPLORATIONS

If you have finished the lab exercises and are waiting to be graded, or if you wish to explore programming in Haskell further at any time, or want to get some extra practice, have a look at the Explorations page.

Exploration problems are optional and should only be attempted after other problems are completed.
You don't have to show these answers to the lab assistants for grading.