Universitetet i Oslo
Institutt for Informatikk
I. Yu, D. Karabeg

INF2220: algorithms and data structures

Series 2

Topic Balanced trees (Exercises with hints for solution)

Issued: 31. 08. 2016

Classroom

Exercise 1 (Trees) Given a binary tree (not necessesarily a binary search tree), with an arbitary number of nodes. Implement/write a function which

(a) returns the smallest value in the tree

(b) returns the largest value in the tree

(c) returns the length of the longest path from the root to a null pointer

(d) returns the length of the shortest path from the root to a null pointer

Exercise 2 (Nodes in a binary tree)

1. Show that in a (non-empty) binary tree with N nodes, there are $N + 1$ `null` links representing children.
2. A *full* node is a node with 2 children. Prove that the number of full nodes + 1 is equal to the number of leaves in a non-empty binary tree.

Solution: [For Exercise 2] The exercise is taken from from Weiss p.160, exercise 4.4 and 4.5.

4.4 For an N-node tree, there are $2N$ pointers and $N-1$ incoming edges. Therefore, we have $2N - (N-1) = N + 1$ `NULL` pointers.

4.6 We prove it by induction on the number of full nodes.

Base case: Single-node tree → 1 leaf, 0 full node. Tree with one full node → 2 leaves, 1 full node.

Inductive case: Assume that all binary trees with n full nodes have n + 1 leaves.

We need to show that a tree with n+1 full nodes has n + 2 leaves: A tree with n+1 full nodes can be formed from a tree with n full nodes in the following ways. First

we observe: given the tree in the induction case, we have to add 1 full node. That can be done by *turning* an existing node in the given tree from a non-full node into a full node. Alternatively: the tree is extended by a (new) subtree which contains the additional full node. The following is the (simplified) core of the argument:

1. Make a non-full inner node, i.e., a node with 1 child, a full node by adding one leaf node.
2. Add 2 children to a leaf node.
3. Add a tree with one full node to a leaf.

From these cases, the number of full nodes is increased by one and the number of leaves is also increased by one. This induction is slightly simplified (but actually captures intuitively the argument), because we are doing induction on the number full node. Basically what is imprecise is that the cases do not cover all possibilities in that a tree with $n + 1$ full nodes can be turned into one with $n + 2$ such nodes (even if restricting to adding nodes at the leaves, which is ok.) If we want to capture *all* such $n + 2$ trees we must be more careful, but the idea remains the same.

Let's observe: a tree with *no* full nodes is linear (= list-like). A tree with 1 full nodes has *two leaves*. This either is obvious or of course is also follows from the induction.

additonal full node: in old tree

1. Make a non-full inner node, i.e., a node with 1 child, a full node by adding one linear tree (plus append to an arbitrary number of leaf-nodes *one* linear tree).
2. Add 2 linear trees/lists as 2 children to a leaf node (and additionally append to an arbitrary number of leaf-nodes *one* linear tree).

additional full node: in the appended, new tree Add not a leaf a tree with *1* full node (and additionally append to an arbitrary number of leaf-nodes *one* linear tree).

From these cases, the number of full nodes is increased by one and the number of leaves is also increased by one.

As an aside: How would one prove that a tree with 1 full node has 2 leaves? One could say: that's obvious (which it more or less is). If one likes more formal arguments: by induction on the number of nodes does the job. The base case is $n = 3$ which is the smallest tree with one full node. The induction should not restrict itself to add new nodes at the leaves (otherwise one would stick to the form of the tree where the full node is at the root).

Exercise 3 (Red-black tree) Build, step by step, red-black trees that result from inserting the following sequences of elements:

1. `41 38 31 12 19 8`
2. `A L G O R I T H M`

Solution: Seen in the form of complexity measures ("Big-O") there's no big difference between AVL and red-black trees (and other form of balanced trees). The trick is always the same: balancing gives a garantee of $\log n$ worst case. The problem of the AVL trees

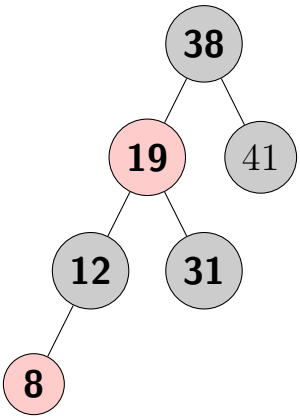


Figure 1: Exercise 2.1

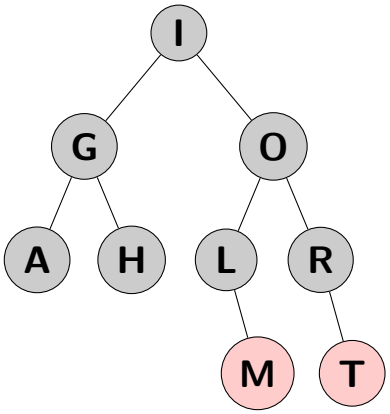


Figure 2: Exercise 2.2

is mainly that they are too strict as far as balancing is concerned. They like to maintain almost perfect balance. It's only "+/- 1". Especially in very large trees and when doing frequent insertion (and/or deletions), the requires frequent rebalance. Red-black trees allow not a constant "+/-1" condition for being balanced but much more loose "factor two". Beside that, they use a clever "coloring scheme" to ensure that condition. It's clever because one does not need to calculate the balance *globally* for all node and compare whether the tree is needs rebalance or not. It suffices to check whether locally the color conditions in the neighborhood of the interted node is satisfied or not. Also the rebalance is done in constant time (which is less easy to see than for the AVLs).

The two resulting red-black trees are given as follows.

Exercise 4 (B-trees)

1. Assume an empty *B-Tree* with $M = 4$ and $L = 4$. Insert the following values in the given order:

   `A B C D G H K M R W Z`

   Show how the tree changes step by step.

2. Assume an empty *B-Tree* with $M = 5$ and $L = 5$. Insert the following values in the given order:

   `2 6 17 20 24 25 27 29 30 31 32 5 21 1 40 45 50 70`

   Show how the tree changes step by step.

3. Assume an empty *B-Tree* with $M = 3$ and $L = 4$.

   - Insert the following values:

     `61 27 19 5 7 25 36 4 42 2 13 44 62 98 43 16 24 29 15`

     Show how the tree changes step by step.

4. Assume an empty *B-Tree* with $M = 3$ and $L = 2$.

   - Insert the following values:

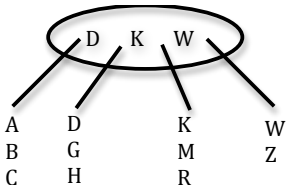


Figure 3: Exercise 3.1

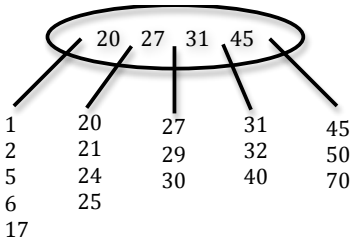


Figure 4: Exercise 3.2

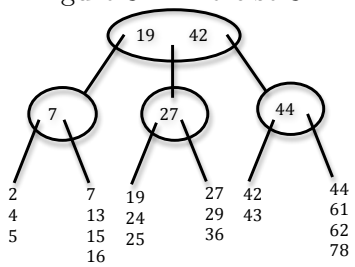


Figure 5: Exercise 3.3

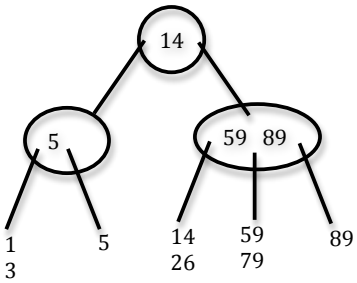


Figure 6: Exercise 3.4 After insertion

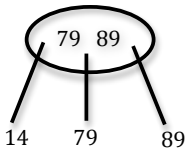


Figure 7: Exercise 3.4 After deletion

```
3 14 1 59 26 5 89 79
```

Show how the tree changes step by step.

- Delete `59, 5, 3, 1` and `26`. Draw the tree after each deletion.

Solution: [B-Trees]

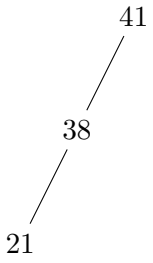
Exercise 5 (Rotations - theory) What do we mean by a *single roation*, and what do we mean by a *double rotation*? Give a few examples on each.

Solution: A single roation is a rotation where the child of a root becomes a new root, making the root a child and the childs subtree a subtree from the root. A double rotation is a rotation where the grandchild becomes a child of the root (first step), and then the same node becomes the root (second step).

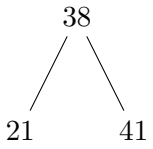
Exercise 6 (AVL trees (extra exercise)) Build, step by step, *AVL* trees that result from inserting the following sequences of elements:

- `41 38 21 12 19 8`
- `A L G O R I T H M`

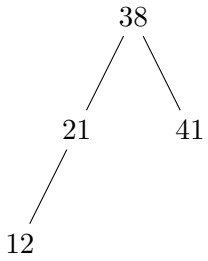
Solution: AVL trees are described in Weiss, Section 4.4. Apart from the AVL-balancing condition, one should particular one explain the two forms of rotation (singe and double). As usual, insertion (without balancing) is very simple; it works, as it ignores the balancing condition, as in ordinary (i.e., unbalanced) binary search trees. After 2 insertions and after having put in the third value, the tree looks as follows, which is clearly unbalance (at least according to AVL).



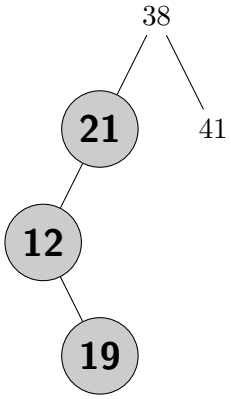
Here's the first time that it's unbalanced. Actually, there's only *one* node which is unbalanced, namely the root. So we need to rotate there; furthermore, it's the case which is called *single* rotation in the book. The result of the rotation is given in



The next insertion of 21 is painless



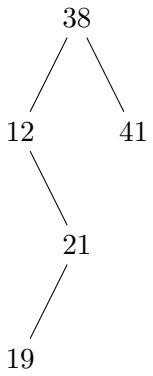
The next one 19, however, leads again to an unbalanced tree:



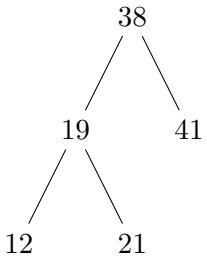
Now it get's a bit more complex: first, there is *more* than one node which is unbalanced. Which ones? 21 and 38. Now the algo must make a choice of which of the unbalanced node to treat by rotation. One could think for a while whether it actually matters (independent

of the fact which choice the concrete algo actually does). The answer is: of course it matters, and only one choice is correct. To see that: the tree is not completely unbalanced, it's unbalanced only by the insertion of *one* leaf node (in this case 19). The only potentially unbalanced nodes are on the paths from that leaf up to the root; in the worst case, up until the root (as in this case). If we balance "high" in the tree (close to the root) the unbalanced sub-tree lower down will remain unbalanced. Thus we need to balance as low as possible, as far away from the root as possible.

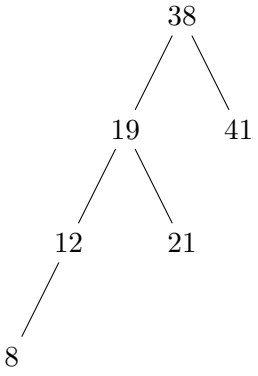
Anyway, the second "complication" or thing to mention is that this is not of the form as the rotation before, this time it's a *double rotation*. If we did as before, i.e., if we did a single rotation, the result would look as follows:



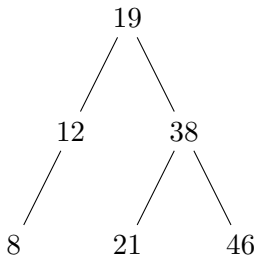
Obviosly, that does not do the job. Therefore we need, as said, "double rotation". The book distinguishes left-right and right-left double rotation, which are symmetric to each other. See Weiss p.145. In our case, the node in question is the one with key 21 and the insertion of the new node has been done to right subtree of the left child (case 2 according to the book). The double rotation is more complex as we have to re-arrange not *two* nodes, but three. Those are emphazised in the earlier pic. After the double rotation,



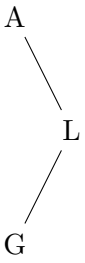
Now the last insertion:



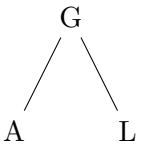
But that again is a simple single rotation. The problematic node is the one carrying 38. The rotation then gives



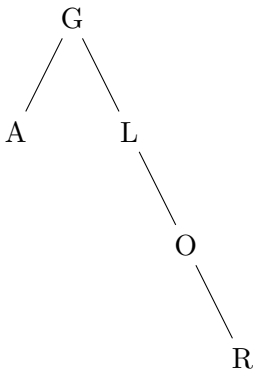
Now the `A L G O R I T H M`. The first critical insertion is the one for `G`.



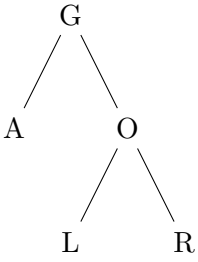
That's the situation for a *double* rotation, and the result of that operation is



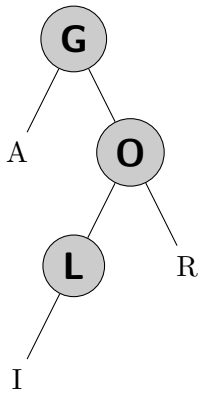
Adding O is straightforward, but already R requires another balance operation.



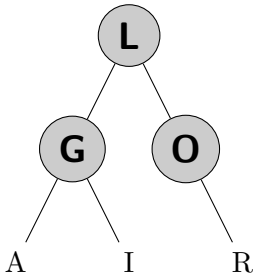
This time, a single rotation does the job:



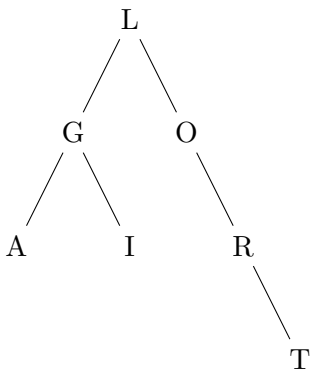
The next one is I and that destroys the balance once again:



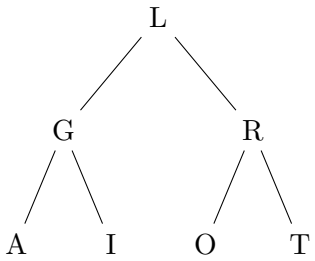
Now, the lowest unbalanced node is the root node.



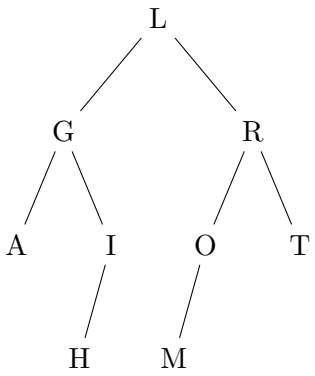
The next node to insert is T. yielding the following tree:



Now, the node labelled O is the lowest unbalanced tree.



The last two insertions of H and M are unproblematic.



□

Lab

Exercise 7 (Binary search trees) In this exercise you are going to implement a binary search tree using two different approaches:

1. You are given a Tree class with an inner class Node:

```
public class Tree {
    Node root;

    private class Node {
        Node right;
        Node left;
        int value;

        Node(int value) {
            this.value = value;
        }
    }
}
```

   Do the following exercises without changing the Node class, i.e. let all functions be a part of the Tree class:

   (a) Implement a function that inserts a value in the BST.

   (b) Implement a function that search for a value in the BST, returning a boolean value

   (c) Implement a function that returns the smallest value in the BST.

2. Assume now that you don't have a Tree class, i.e. only the structure

```
public class Node {
    Node right;
    Node left;
    int value;
```

```
    Node(int value) {
        this.value = value;
    }
}
```

An empty tree is refered to as a null pointer; the root is used to refer to the tree. Implement all of the above functions as recursive methods in the Node class.

Exercise 8 (Binary tree) Given a binary tree whose nodes are given as instances of the following class:

```
class BinNode {
  int data;
  BinNode left;
  BinNode right;
}
```

An empty tree is represented by the null reference.

1. Write a method `int number(BinNode t)` which gives back the *number of nodes*.
2. Write a method `int sum(BinNode t)` which gives back the sum of the integer data values of all nodes in the tree.

Exercise 9 (Binary trees (2)) Revisiting the binary trees and the `BinNode` data structure described in Exercise 8, this exercise here is to provide a slightly different way of solving the same 2 problems. Instead of the methods sketched in Exercise 8, provide two methods with the (alternative) interface

```
int number()

int sum()
```

so that they are local to class `BinNode`, i.e. one should be able to call functions as follows:

```
int number = root.number();

int number = root.sum();
```

Exercise 10 (B-Trees) Write a *general* implementation of insertion for a B-Tree. Note that you have to restructure the tree in case the leaf is full after the insertion.

Exercise 11 (Rotations - programming (can also be blackboard exercises)) Implement a function which makes V the new root node (for a subtree), without destroying the properties of the BST.

References