Tirgul 5

- AVL trees

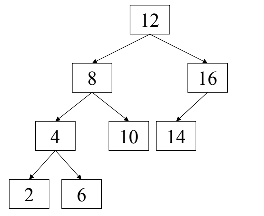
Binary search trees (reminder)

- Each tree node contains a value.
- For every node, its left subtree contains smaller values and its right subtree contains larger values.
- The time complexity of a search operation is proportional to the tree's depth.
- The good case: If the tree is balanced, then every operation takes $O(\log n)$ time.
- The bad case: The tree might get very unbalanced.
  For example, when inserting ordered numbers to the tree, the resulting height will be exactly n.

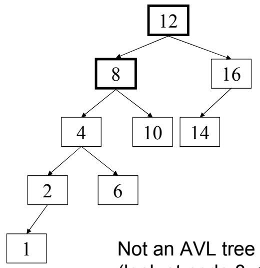
AVL Trees

- Balanced Trees: After insert and delete operations we "fix" the tree to keep it (almost) balanced.
- **AVL Tree:** A binary search tree with the following additional balance property: For any node in the tree, the height of the left and right subtrees can differ by 1 at most.
- Note that we require this balance property for every node, not just the root.

Example



An AVL tree

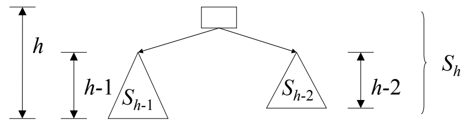


Not an AVL tree
(look at node 8, 12)

AVL trees are "reasonably" balanced

- We would like to prove that the "deepest" tree with *n* nodes still has only logarithmic depth.
- Another way to look at the same problem is proving that the smallest tree with depth *h* has size at least c^h for some $c > 1$ (in fact in our case $c = 1.3$)
- If we prove the second claim, then a tree with *n* nodes must have at least depth at least $\log_{1.3} n$ (otherwise it is a counter example for the second claim.)
- Let's try to build the minimal tree with depth *h*

Minimal AVL tree of height h



Look at the minimal tree of depth h, denote it S_h

Since the root's height is h, one of its sons' height must be h-1. From the balance condition, the other son has height either h-1 or h-2.

Therefore , in the minimal tree the root has one son with a sub tree of depth $h-1$ and one son of depth $h-2$.

How do these sub trees look like? They are minimal i.e.

they are the minimal trees S_{h-1} and S_{h-2}, respectively.

The Maximal Height of an AVL Tree

The smallest AVL tree of depth 1 has 1 node, and the smallest AVL tree of depth 2 has 2 nodes.

Therefore we get:

Claim: $S_h = S_{h-1}+S_{h-2}+1$ ($S_1 = 1$; $S_2 = 2$)

Fact : $S_h \geq 2^{h/2}$

Theorem: For any AVL tree with n nodes and height h: $h = O(\log n)$.

Proof:

$$n \geq S_h = 2^{h/2}$$
$$\Rightarrow h/2 \leq \log(n) \Rightarrow h = O(\log n)$$

A lower bound on S_h

We know that $S_h = S_{h-1}+S_{h-2}+1$ ($S_1 = 1$; $S_2 = 2$)

It is easy to show by induction that $S_h \geq S_{h-1}$

We shall see by induction on $h > 2$ that

$S_h \geq (2^0 + 2^1+2^2+ \ldots + 2^{\lfloor h/2 \rfloor})$

Base : $S_3 = 4 \geq 2^0+2^1$, $S_4 = 7 \geq 2^0+2^1+2^2$

$S_h = S_{h-1} + S_{h-2} + 1 \geq 2(S_{h-2}) + 1$ (monotonicity)

$\geq 2(2^0+\ldots+ 2^{\lfloor (h-2)/2 \rfloor})+1 = (2^1+\ldots+ 2^{\lfloor h/2 \rfloor}) + 2^0$.

By geometric series some we get that

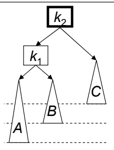
$S_h \geq (2^{\lfloor h/2 \rfloor +1}- 1)/(2-1) \geq 2^{h/2}$

How to maintain balance

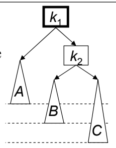
- General rule: after an insert or delete operation, we fix all nodes that got unbalanced.
- Since the height of any subtree has changed by at most 1, if a node is not balanced this means that one son has a height larger by exactly two than the other son.
- Next we show the four possible cases that cause a height difference of 2. In all figures, marked nodes are unbalanced nodes.

(only) Four imbalance cases

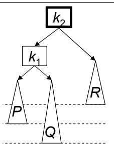
Case 1: The left subtree is higher than the right subtree, and this is caused by the left subtree of the left child.



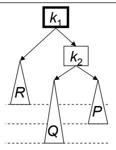
Case 4: The symmetric case to case 1



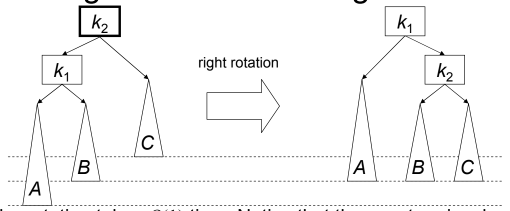
Case 2: The left subtree is higher than the right subtree, and this is caused by the right subtree of the left child.



Case 3: The symmetric case to case 2

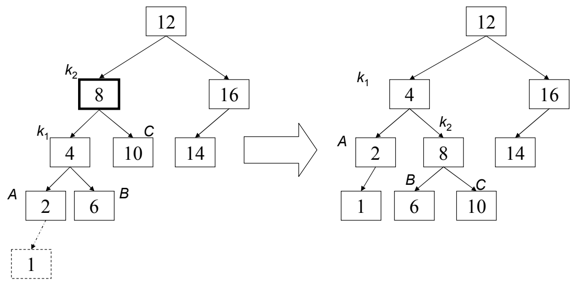


Single Rotation - Fixing case 1



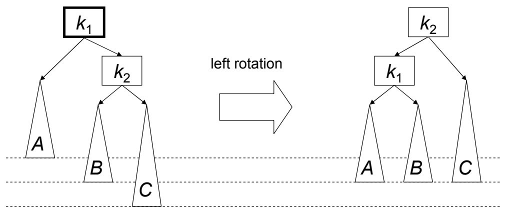
- The rotation takes $O(1)$ time. Notice that the new tree is a legal search tree.
- For insert - it must be the case that subtree *A* had been increased, so after the rotation, the tree has height as before the insert.
- For delete, it must be the case that *C* had been decreased, so after the rotation, the tree has height shorter by 1.

Example (caused by insertion)



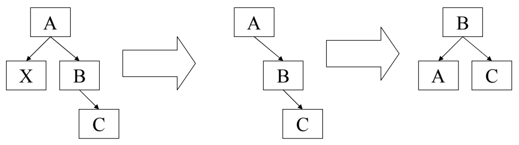
• Notice that the tree height has not changed after the rotation (since it was an insert operation).

Single Rotation for Case 4



Example (caused by deletion)

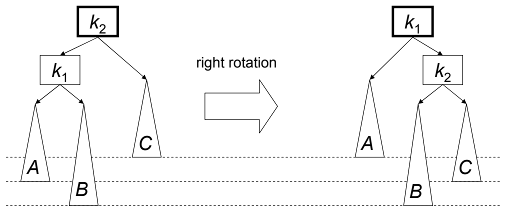
Deleting X and performing a single rotation:



• For the rotation, k_1 is node A, and k_2 is node B. We make k_2 the root, and k_1 its left son.

• Notice that the tree height has changed after the rotation (since it was a delete operation).

Fixing case 2 - first try...

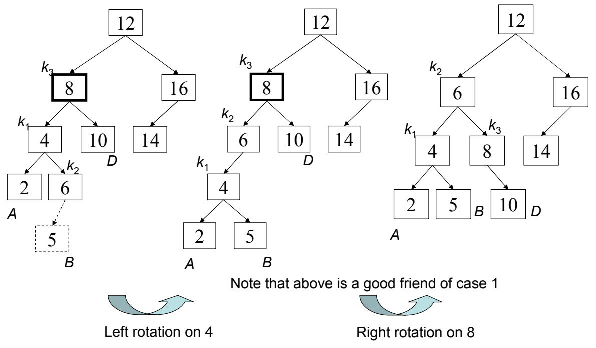


Single rotation doesn't help - the tree is still not balanced!

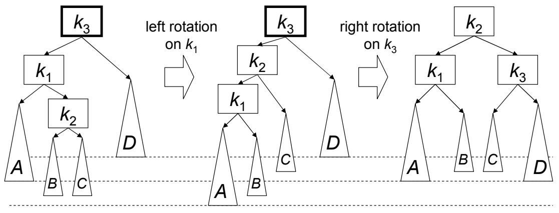
What can we do?

Use rotations on k_1's sub tree to reduce case 2 to case 1!

Example (caused by insertion)

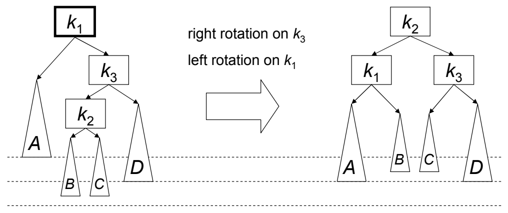


Double Rotation to fix case 2



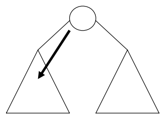
• After insertion - original height (of the root) stays the same.

Double Rotation to fix case 3



Insert and delete operations

- First, we insert/delete the element, as in a regular binary search tree, and then we re-balance.
- Observation: only nodes on the path from the root to the node we changed may become unbalanced



If we went left from the root, then the right subtree was not changed, thus it remains balanced.
This continues when we go down the tree.

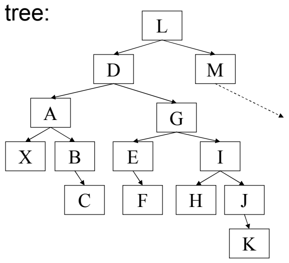
Insert and delete operations (continued)

- After adding/deleting a leaf, start to go up back to the root, and while going up, re-balance every node on the way (if needed). The path is $O(\log n)$ long, and each node balance takes $O(1)$, thus the total time for every operation is $O(\log n)$.

- In fact, in the insertion we can do better - after the first balance (when going up), the subtree that was balanced has height as before, so all higher nodes are now balanced again. We can find this node in the pass down to the leaf, so one pass is enough.

Delete requires two passes

- In more sophisticated balanced trees (like red-black and B-trees), delete also requires one pass. Here this is not the case. For example, deleting X in the following tree:



A note about implementation

- In programming exercise 2, you will be required to implement a balanced tree.
- Converting an algorithm to a real program requires much thought. When done correctly, many bugs are avoided.
- Important principles:
  - Do it as general as possible, without cut & paste. Although more complicated to design, it will reduce your total work time.
  - Methods should be short and simple. If some method becomes too complicated, you missed something, and this is a sure bug!

Common mistakes in THW1