5 AVL trees: deletion

Summer Term 2010

Robert Elsässer

Albert-Ludwigs-Universität Freiburg

Definition of AVL trees

Definition: A binary search tree is called AVL tree or height-balanced tree, if for each node v the height of the right subtree $h(T_r)$ of v and the height of the left subtree $h(T_l)$ of v differ by at most 1.

Balance factor:

$$bal(v) = h(T_r) - h(T_l) \in \{-1, 0, +1\}$$

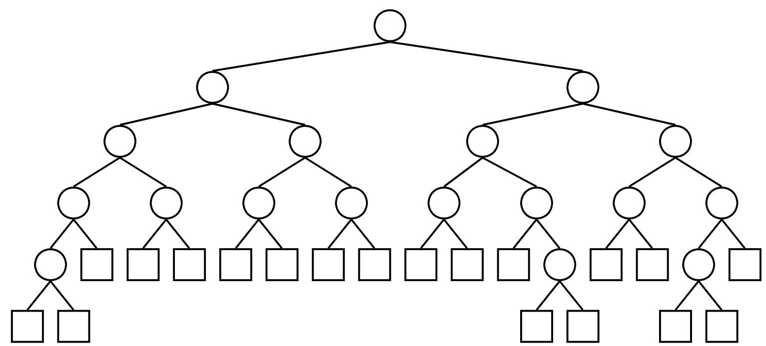
Deletion from an AVL tree

- We proceed similarly to standard search trees:

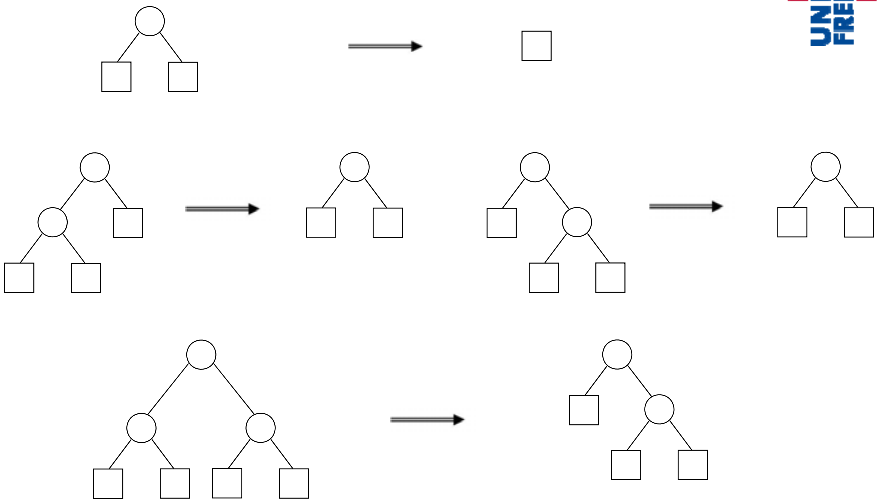
  1. Search for the key to be deleted.
  2. If the key is not contained, we are done.
  3. Otherwise we distinguish three cases:
     (a) The node to be deleted has no internal nodes as its children.
     (b) The node to be deleted has exactly one internal child node.
     (c) The node to be deleted has two internal children.

- After deleting a node the AVL property may be violated (similar to insertion).
- This must be fixed appropriately.

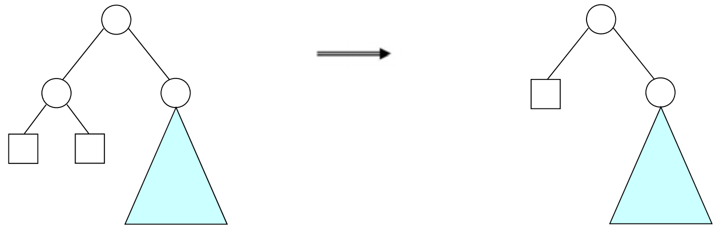
Example



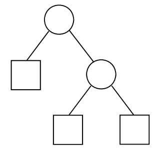
Node has only leaves at children



Node has only leaves at children

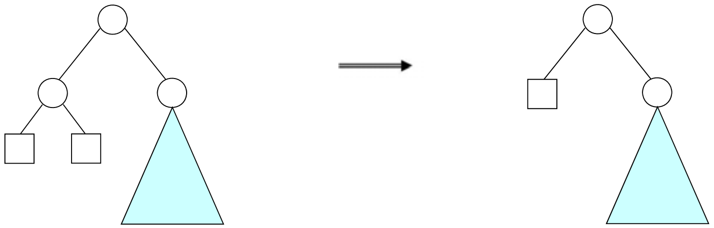


height $\in \{1, 2\}$



Case1: height = 1: Done!

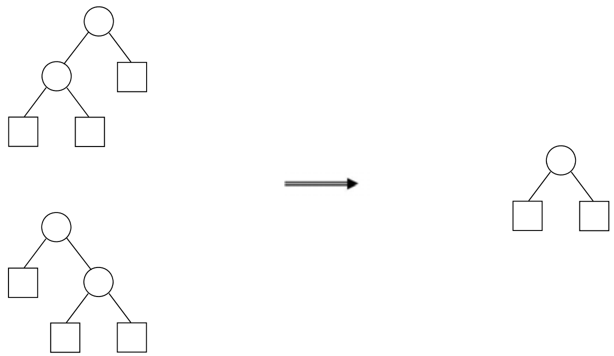
Node has only leaves at children



Case 2: height = 2

Note: height may have decreased by 1!

Node has one internal node as a child



Node has two internal nodes as children

- First we proceed just like we do in standard search trees:

  1. Replace the content of the node to be deleted p by the content of its symmetrical successor *q*.

  2. Then delete node *q*.

- Since *q* can have at most one internal node as a child (the right one), cases 1 and 2 apply for *q*.

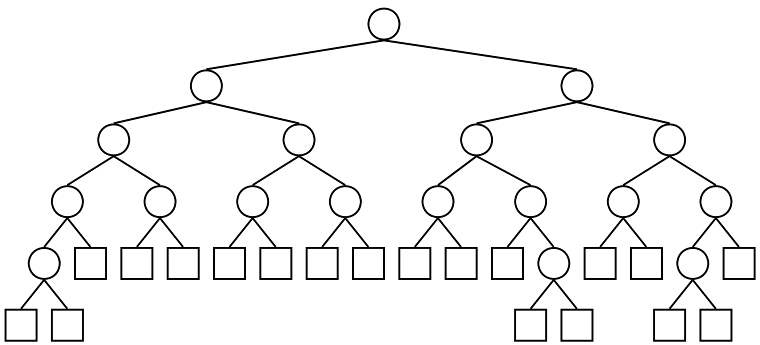
The method *upout*

- The method *upout* works similarly to *upin*.
- It is called recursively along the search path and adjusts the balance factors via rotations and double rotations.
- When *upout* is called for a node *p*, we have (see above):

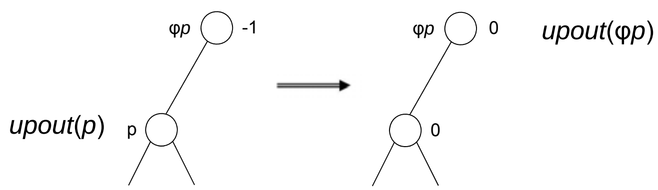
  1. *bal*(*p*) = 0
  2. The height of the subtree rooted in *p* has decreased by 1.

- *upout* will be called recursively as long as these conditions are fulfilled (invariant).
- Again, we distinguish 2 cases, depending on whether *p* Is the left or the right child of its parent φ*p*.
- Since the two cases are symmetrical, we only consider the case where *p* is the left child of φ*p*.

Example

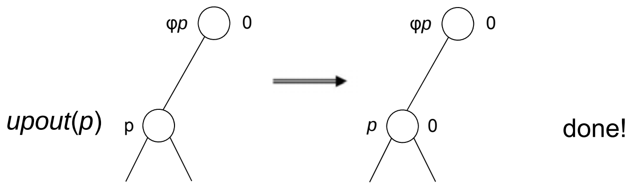


Case 1.1: *p* is the left child of φ*p* and *bal*(φ*p*)= -1



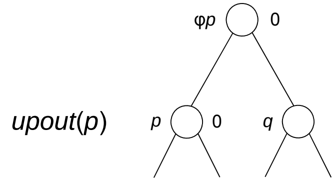
- Since the height of the subtree rooted in *p* has decreased by 1, the balance factor of φ*p* changes to 0.
- By this, the height of the subtree rooted in φ*p* has also decreased by 1 and we have to call *upout*(φ*p*) (the invariant now holds for φ*p*!).

Case 1.2: *p* is the left child of *φp* and *bal*(*φp*)= 0



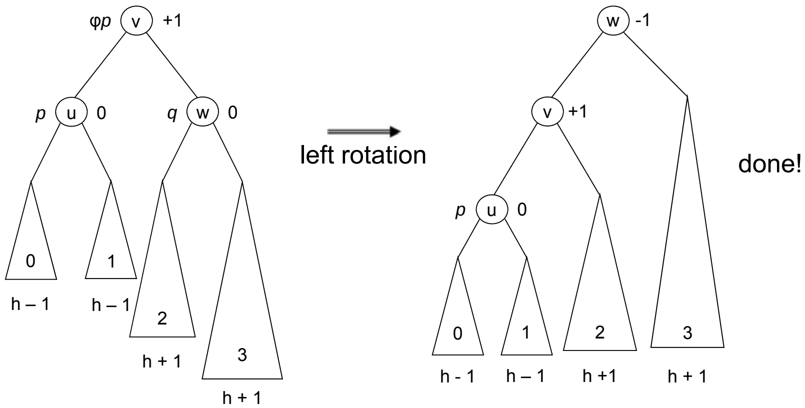
- Since the height of the subtree rooted in *p* has decreased by 1, the balance factor of *φp* changes to 1.
- Then we are done, because the height of the subtree rooted in *φp* has not changed.

Case 1.3: *p* is the left child of *φp* and *bal*(*φp*)= +1

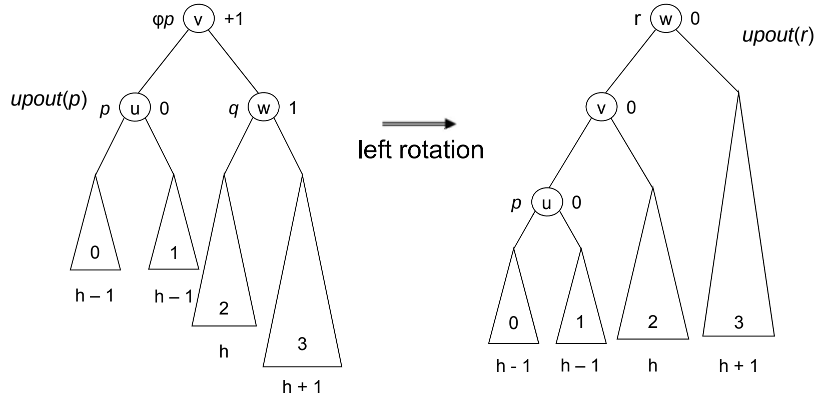


- Then the right subtree of *φp* was higher (by 1) than the left subtree before the deletion.
- Hence, in the subtree rooted in *φp* the AVL property is now violated.
- We distinguish three cases according to the balance factor of *q*.

Case 1.3.1: *bal*(*q*) = 0

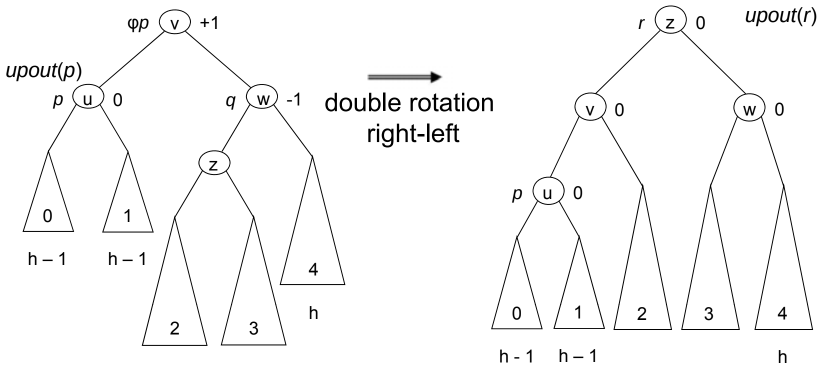


Case 1.3.2: *bal*(*q*) = +1



- Again, the height of the subtree has decreased by 1, while *bal*(*r*) = 0 (invariant).
- Hence we call *upout*(*r*).

Case 1.3.3: *bal*(*q*) = -1



- Since *bal*(*q*) = -1, one of the trees 2 or 3 must have height *h*.
- Therefore, the height of the complete subtree has decreased by 1, while *bal*(*r*) = 0 (invariant).
- Hence, we again call *upout*(*r*).

Observation

- Unlike insertions, deletions may cause recursive calls of *upout* after a double rotation.
- Therefore, in general a single rotation or double rotation is not sufficient to rebalance the tree.
- There are examples where for all nodes along the search path rotations or double rotations must be carried out.
- Since h ≤ 1.44 ... $\log_2(n) + 1$, we may conclude that the deletion of a key form an AVL tree with *n* keys can be carried out in at most $O(\log n)$ steps.
- AVL trees are a *worst-case efficient* data structure for finding, inserting and deleting keys.