Algorithms and Data Structures

AVL: Balanced Search Trees

Ulf Leser

Content of this Lecture

- AVL Trees
- Searching
- Inserting
- Deleting

History

- Adelson-Velskii, G. M. and Landis, E. M. (1962). "An information organization algorithm (in Russian)", Doklady Akademia Nauk SSSR. 146: 263–266.
  - Georgi Maximowitsch Adelson-Welski (russ. Георгий Максимович Адельсон-Вельский; weitere gebräuchliche Transkription Adelson-Velsky und Adelson-Velski; * 8. Januar 1922 in Samara) ist ein russischer Mathematiker und Informatiker. Zusammen mit J.M. Landis entwickelte er 1962 die Datenstruktur des AVL-Baums. Er lebt in Ashdod, Israel.
  - Jewgeni Michailowitsch Landis (russ. Евгений Михайлович Ландис; * 6. Oktober 1921 in Charkiw, Ukraine; † 12. Dezember 1997 in Moskau) war ein sowjetischer Mathematiker und Informatiker ... Zusammen mit G. Adelson-Velsky entwickelte Landis 1962 die Datenstruktur des AVL-Baums.
  - Source: http://www.wikipedia.de/

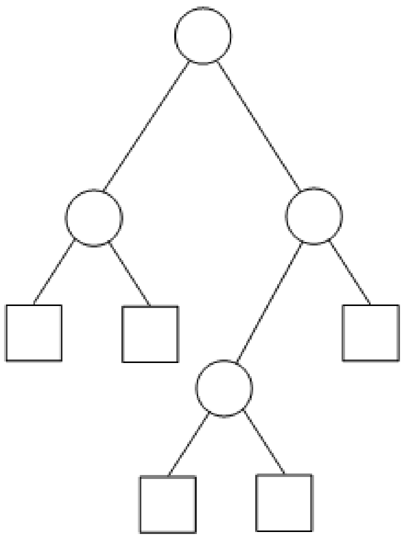
Balanced Trees

- General search trees: Searching / inserting / deleting is O(log(n)) on average, but O(n) in worst-case
- Complexity directly depends on tree height
- Balanced trees are binary search trees with certain constraints on tree height
  - Intuitively: All leaves have "similar" depth: ~log(n)
  - Accordingly, searching / deleting / inserting is in O(log(n))
  - Difficulty: Keep the height constraints during tree updates
    - Without reorganizing the entire tree, i.e., within O(log(n))
- First proposal of balanced trees is attributed to [AVL62]
- Many others since then: brother-, B-, B*-, BB-, ... trees

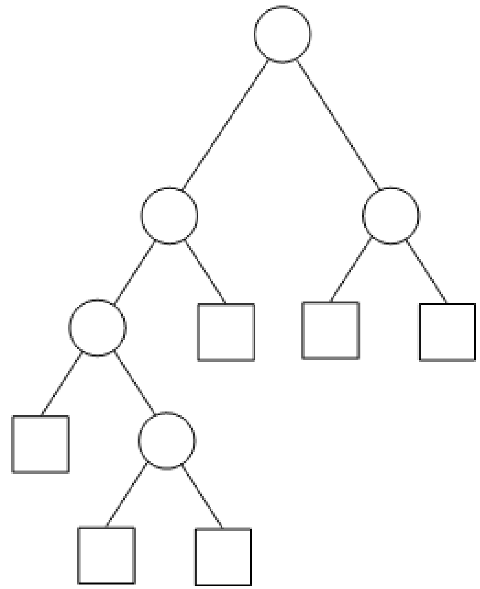
AVL Trees

- Definition
  An AVL tree T=(V, E) is a binary search tree in which the following constraint holds:
  $$\forall v \in V: |height(v.leftChild) - height(v.rightChild)| \leq 1$$
- Remarks
  - AVL trees are height–balanced
    - Condition does not imply that the level of all leaves differ by at most 1
  - Will call this constraint height constraint (HC)
  - AVL trees are search trees, i.e., the search constraint (SC) must hold: Right child is larger than parent is larger than left child

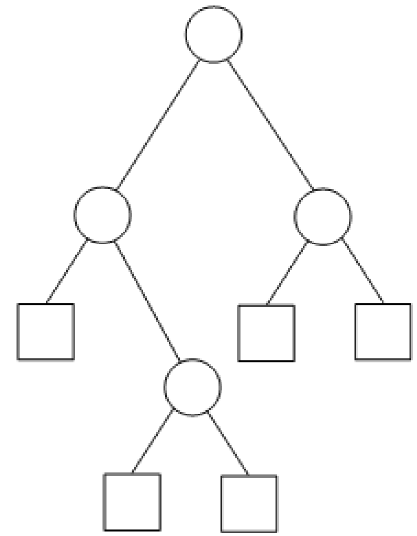
Examples [source: S. Albers, 2010]



AVL?

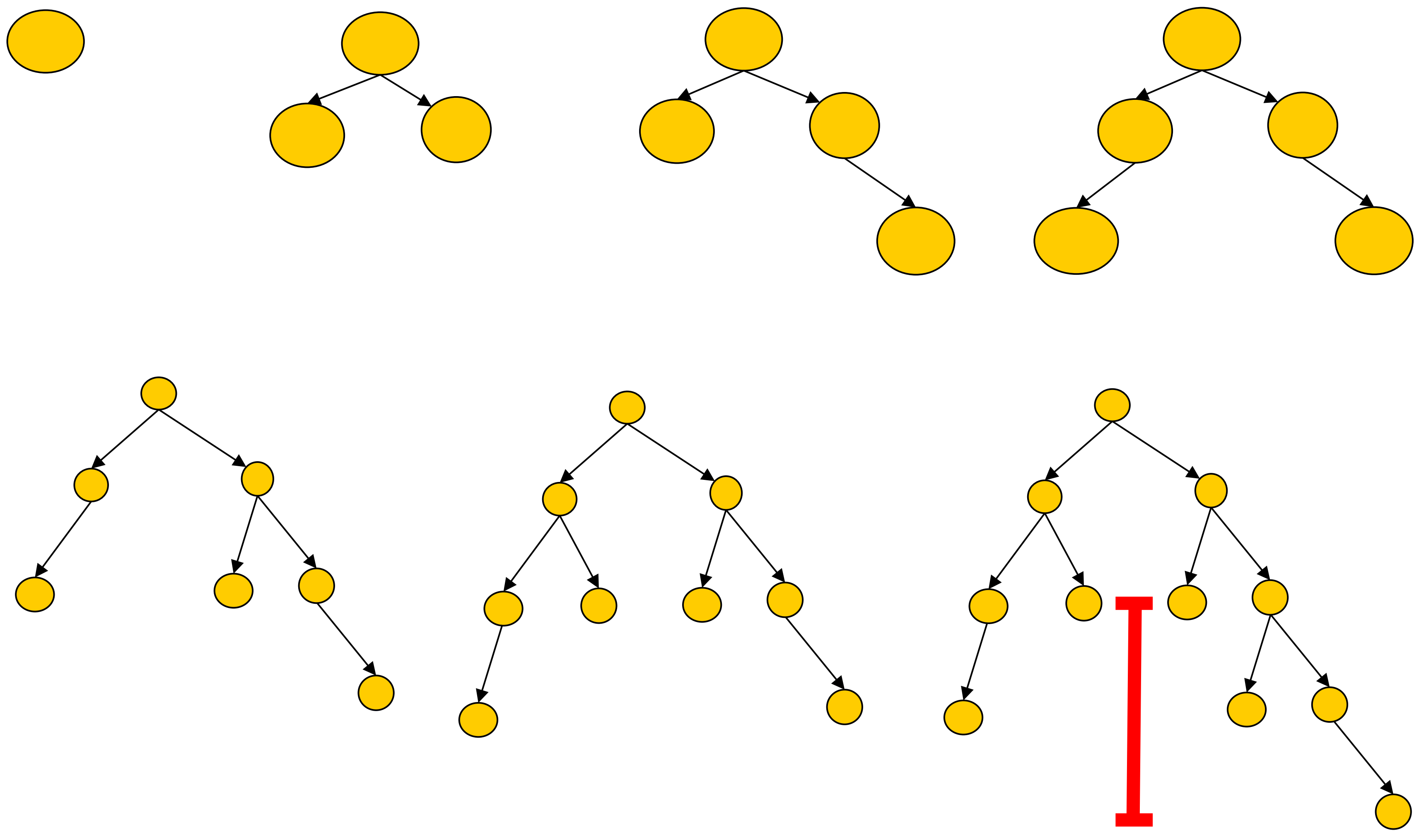


AVL?

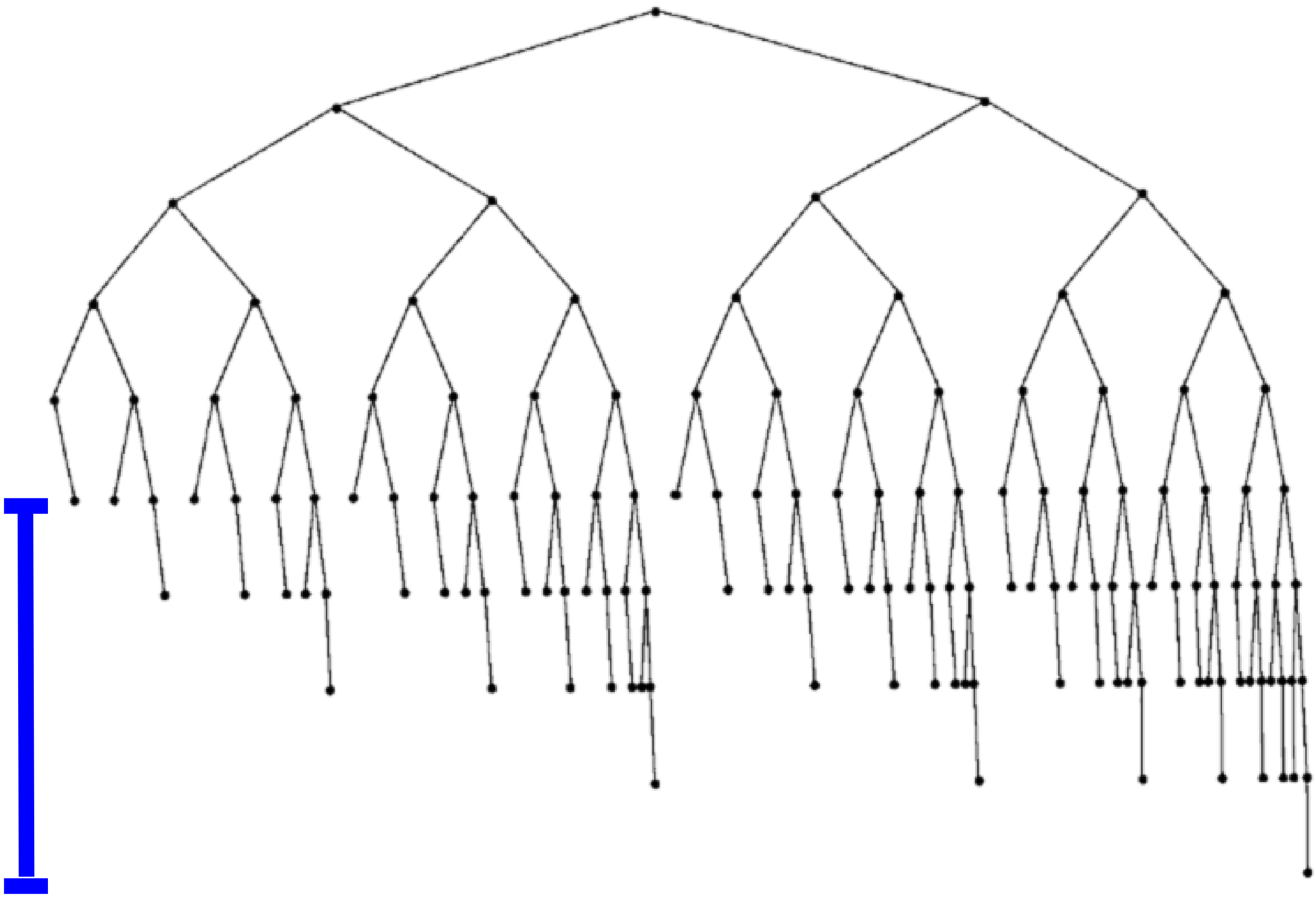


AVL?

„Unbalanced“

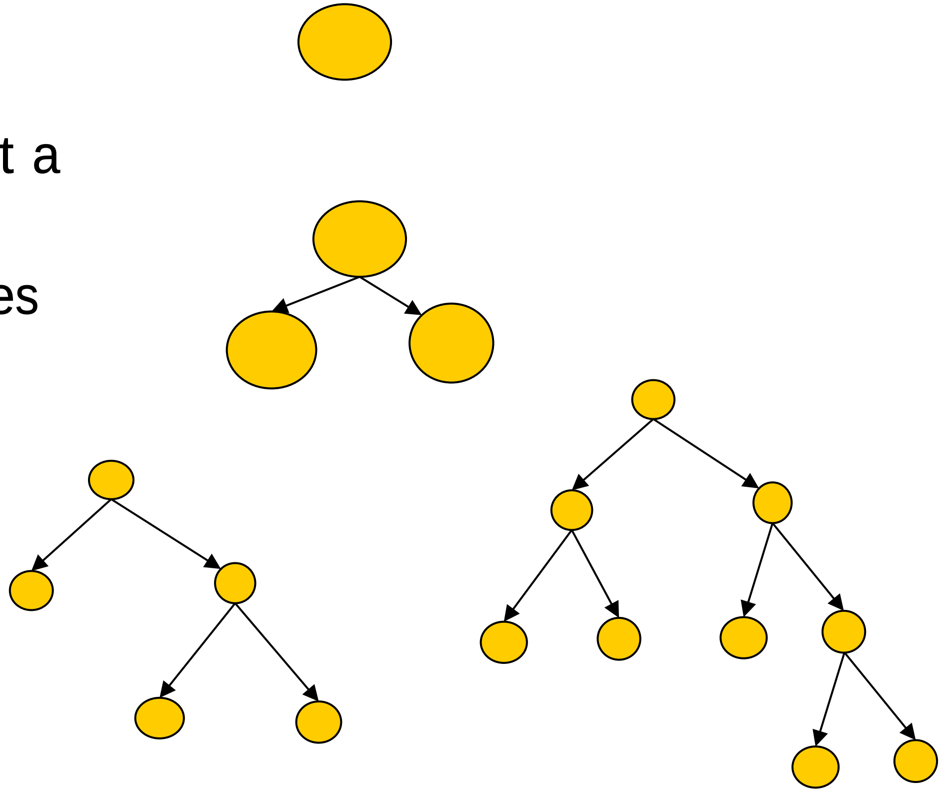


Worst-Case



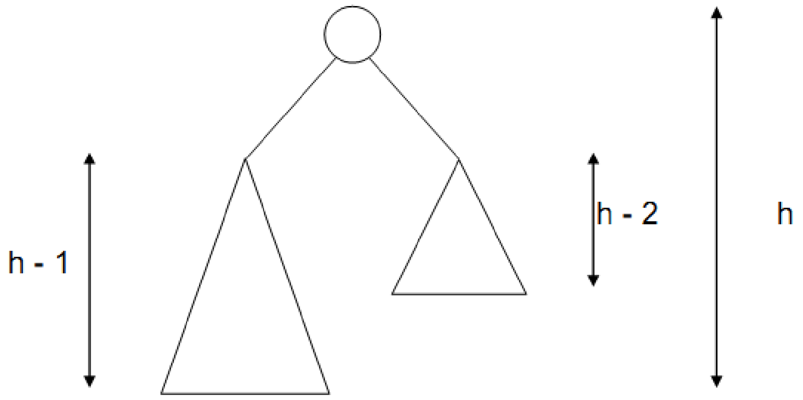
Height of an AVL Tree

- Lemma
  An AVL tree T with n nodes has height $h \leq O(\log(n))$
- Proof by induction
  - We construct AVL trees with the minimal # of nodes (n) at a given height h
  - Let m be the number of leaves
  - $h=0 \Rightarrow m=1$
  - $h=1 \Rightarrow m=2$
  - $h=2 \Rightarrow m>3$
  - $h=3 \Rightarrow m>5$



Height of an AVL Tree

- Lemma
  An AVL tree T with n nodes has height $h \leq O(\log(n))$
- Proof by induction
  - We construct AVL trees with the minimal # of nodes at a given height h
  - Let m(h) be the minimal number of leaves of an AVL tree of height h
  - It holds: $m(h) = m(h-1)+m(h-2)$



  - Such “maximally unbalanced” trees are called Fibonacci-Trees

Proof Continued

- These are exactly the Fibonacci numbers fib
  - 0, 1, 1, 2, 3, 5, 8...
- Recall (from Fibonacci search)

$$fib(i) \sim \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{i+1} = \frac{1}{\sqrt{5}} * \left(\frac{1+\sqrt{5}}{2}\right) * \left(\frac{1+\sqrt{5}}{2}\right)^{i} = c * 1{,}61...^{i}$$

- Since h “starts” at i=2:

$$m(h) = fib(h+2) \sim c * 1{,}61^{h+2} = c * 1{,}61 * 1{,}61 * 1{,}61^{h} = c' * 1{,}61^{h}$$

- This yields (recall that n=m+m-1)

$$\frac{n+1}{2c'} \sim 1{,}61^{h} \quad \Rightarrow \quad h \sim \log(n)$$

Content of this Lecture

- AVL Trees
- Searching
- Inserting
- Deleting

Searching in an AVL Tree

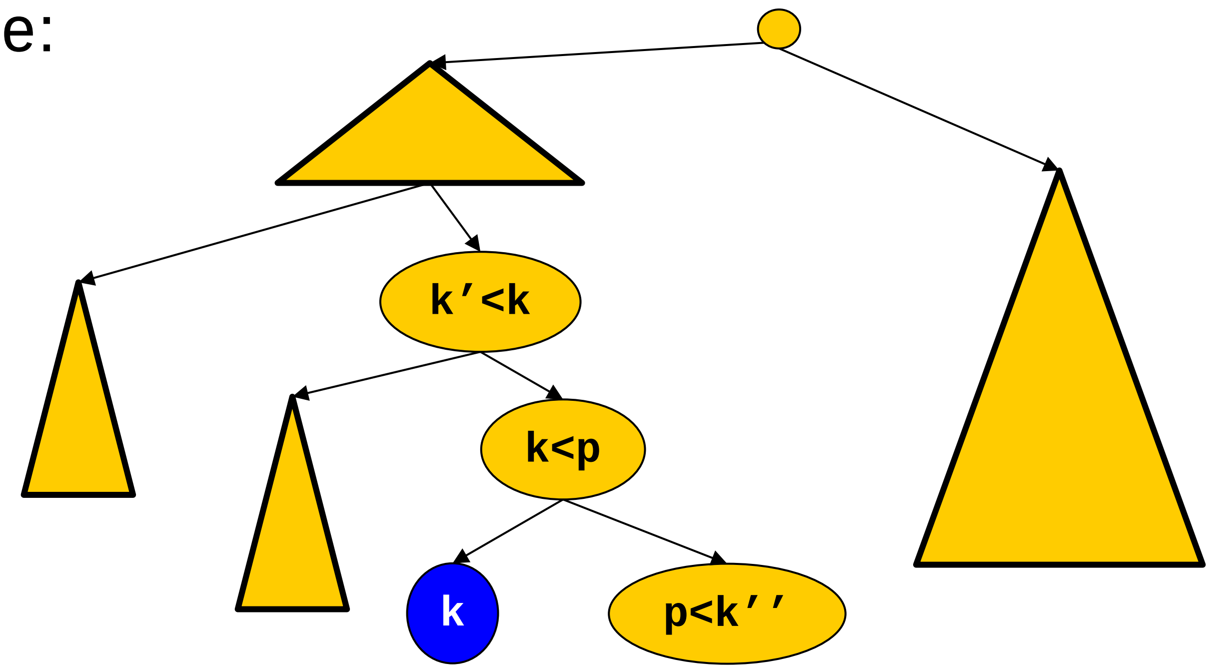
- Searching is in O(log(n))
  - Follows directly from the worst-case height
- Note: The best-case height is ceil(log(n)), so best-case and worst-case asymptotically are of the same order

Inserting

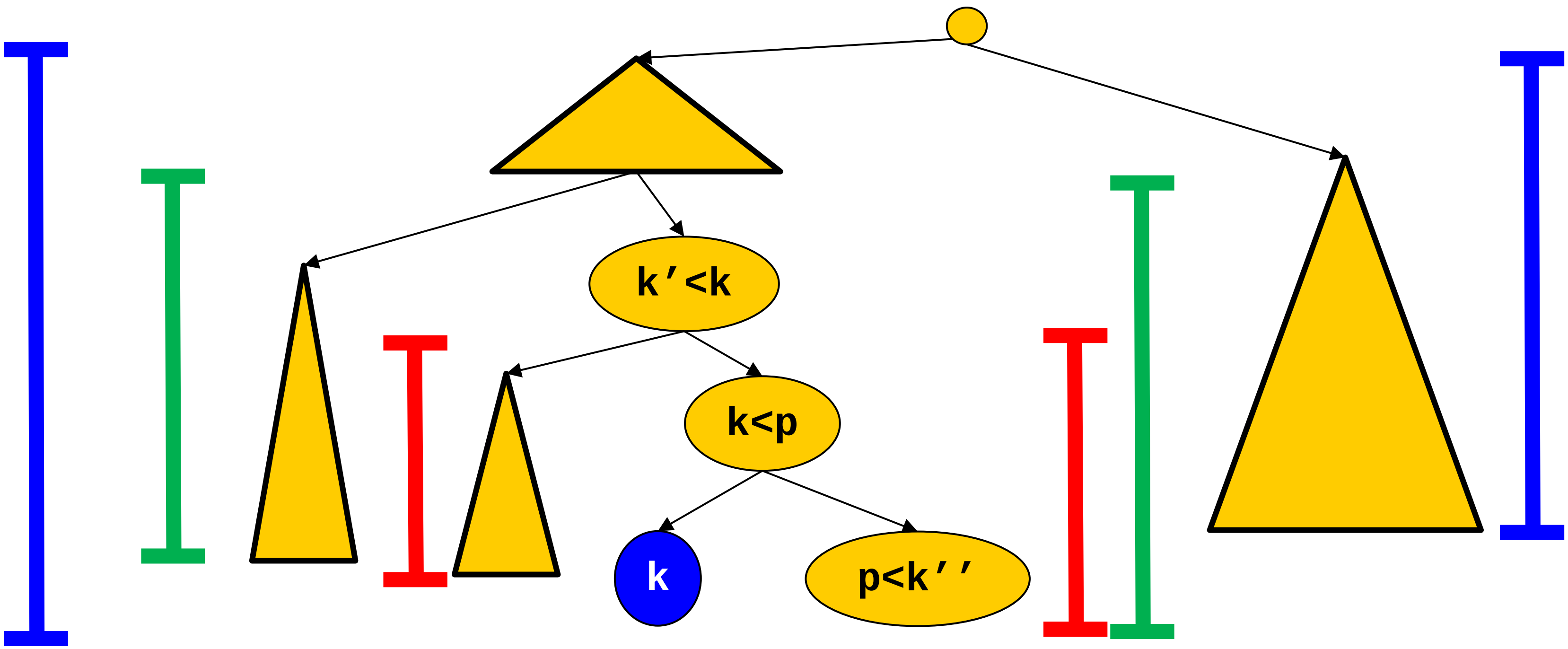
- This requires more work
- The trick is to insert nodes without hurting the height constraint (HC)
- We first explain the procedure(s) and then prove that HC always holds after insertion of a node if HC held before this insertion

Framework

- Assume AVL tree T=(V, e) and we want to insert k, k∉V
- As usual, we first check whether k∈V and end in a node v where we know that k cannot be in the subtree rooted at v
- What are the possible situations?
- This is one:

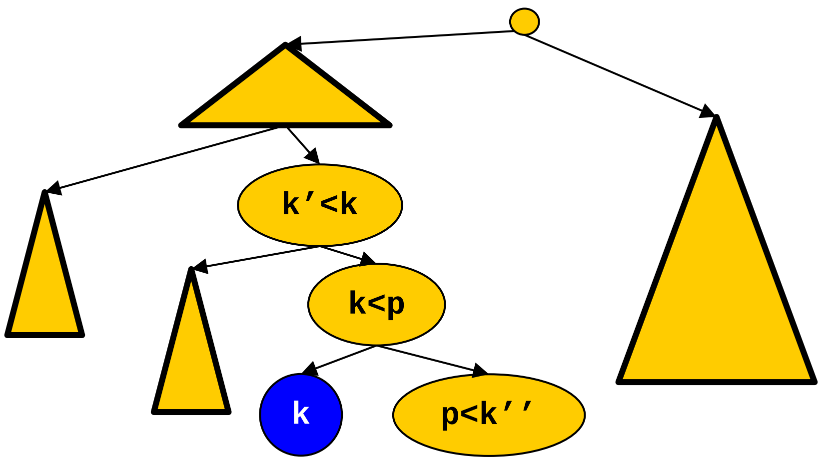


Height Constraints



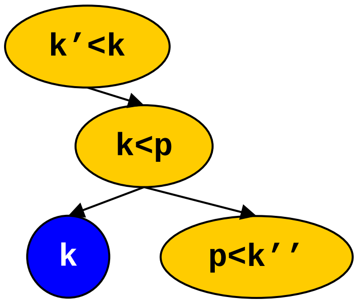
How to Proof the HC

- Before insertion, HC and SC held
  - Note: k’’ cannot have children
- We now only look at this particular case
- Height constraint
  - The height of only one subtree changes – left child of p
  - Adding k does not hurt HC in p (because k’’ exists)
  - Thus, HC also holds after insertion
- Search constraint (we have k’<k<p<k’’)
  - Since k is larger than k’, it must be in the right subtree of k’
  - Since k is smaller than p, it must be in the left subtree of p
  - This subtree didn’t exit and is created now
  - Thus, SC holds after insertion



The Essential Information

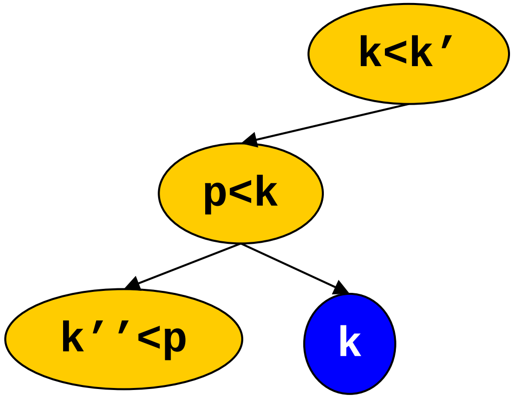
- Before insertion, HC and SC held
  - Note: k’’ cannot have children
- We now only look at this particular case



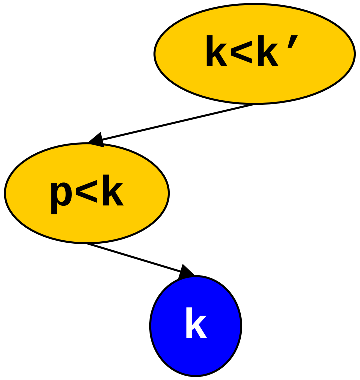
- Height constraint
  - The height of only one subtree changes – left child of p
  - Adding k does not hurt HC in p (because k’’ exists)
  - Thus, HC also holds after insertion
- Search constraint (we have k’<k<p<k’’)
  - Since k is larger than k’, it must be in the right subtree of k’
  - Since k is smaller than p, it must be in the left subtree of p
  - This subtree didn’t exit and is created now
  - Thus, SC holds after insertion

Other Cases

- Also trivial



- Problem
  - The left subtree of k' changes its height
  - We have to look at the height of the right subtree of k' to decide what to do
  - Actually, we only need to know if it is larger, smaller, or equal in height to the left subtree (before insertion)



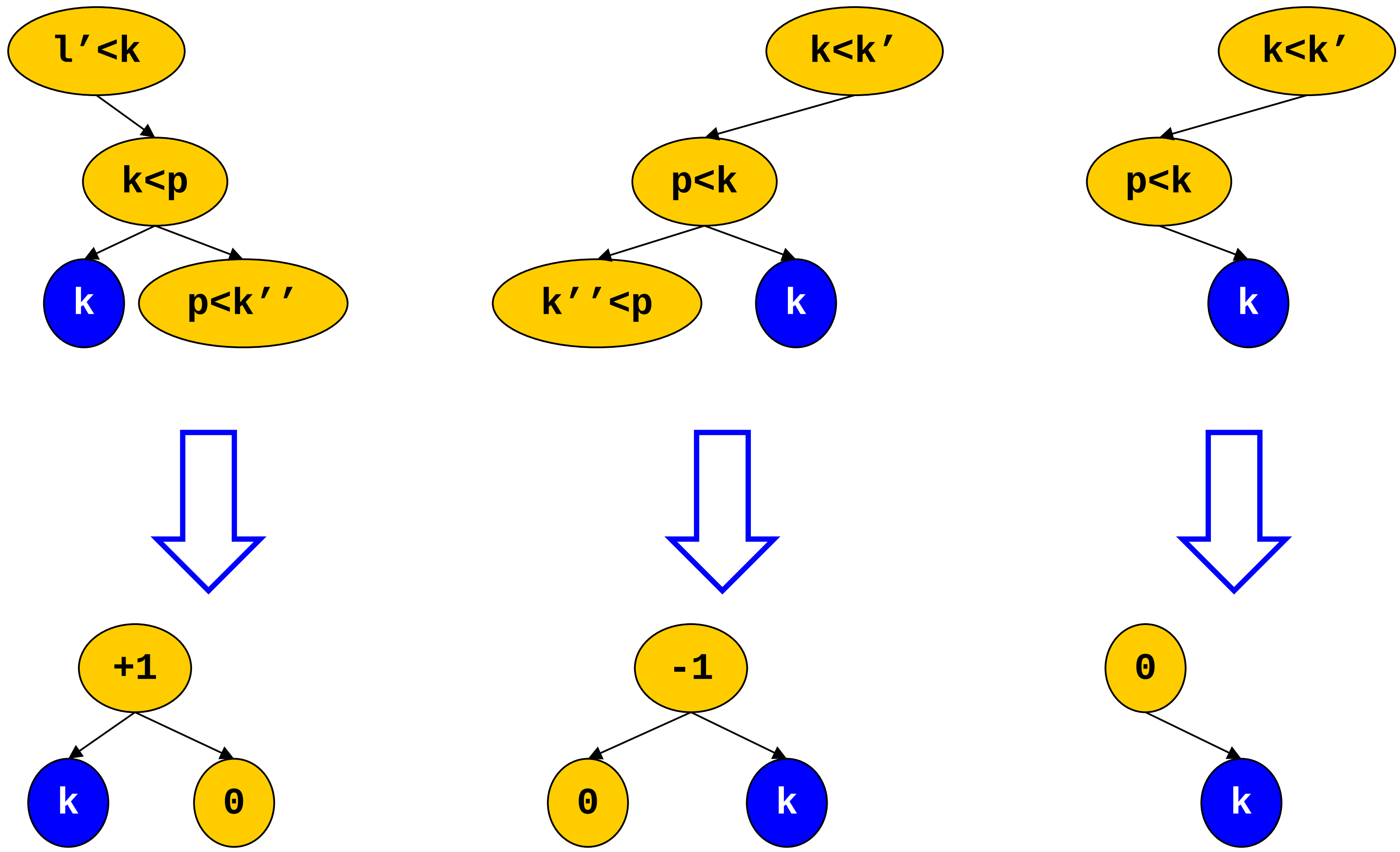
Abstraction

- We assume that we found the position of k such that SC holds after insertion
  - We don't need to check from now on – its part of the case
- To check HC, we need to know the height differences in every node that is an ancestor of the new position of k
- Definition
  L*et T=(V, E) be a tree and p∈V. We define*

  $$bal(p) = height(\ right\_child(p)) - height(\ left\_child(p))$$

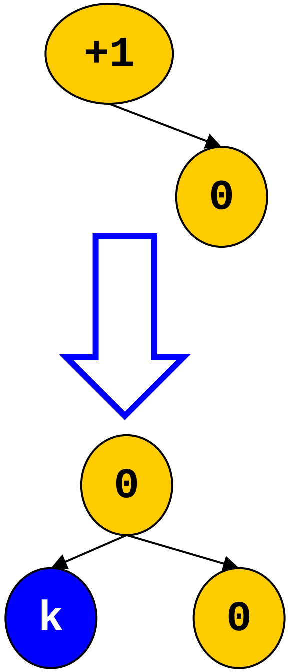
- Clearly, if T is an AVL tree, then $bal(p) \in \{-1, 0, 1\}$

New Presentation



More Systematic

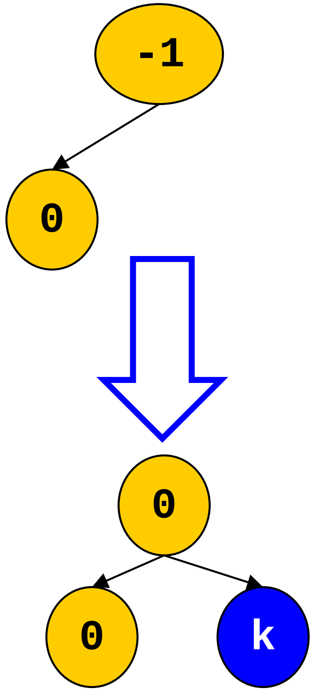
- Assume AVL tree T=(V, e) and we want to insert k, k∉V
- We found the node p under which we want to insert k
- Three possible cases:



- Case 1: bal(p)=+1
  - Then there exists a right “subtree” of p (one node only)
  - We insert k as left child
  - Height of p doesn’t change
    - Ancestors of p remain unaffected
  - Adapt bal(p) and we are done

Case 2

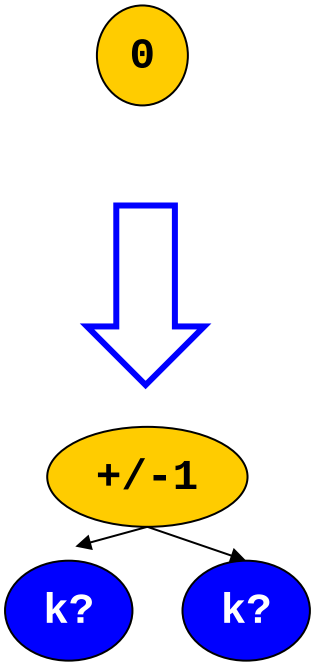
- Assume AVL tree T=(V, e) and we want to insert k, k∉V
- We found the node p under which we want to insert k
- Three possible cases:



- Case 2: bal(p)=-1
  - Then there exists left “subtree” of p (one node onl)
  - We insert k as right child
  - Height of p doesn’t change
    - Ancestors of p remain unaffected
  - Adapt bal(p) and we are done

Case 3

- Assume AVL tree T=(V, e) and we want to insert k, k∉V
- We found the node p under which we want to insert k
- Three possible cases:



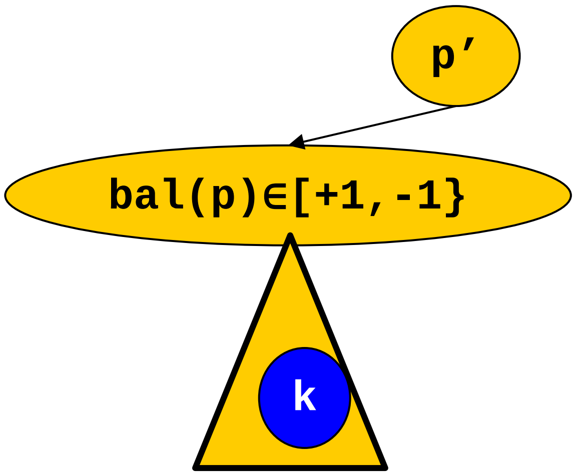
- Case 3: bal(p)=0
  - There is neither a left nor a right subtree of p (p is a leaf)
  - We insert k as left or right child
  - Height of p changes
  - Ancestors of p are affected
  - Adapt bal(p) and look at parent(p)

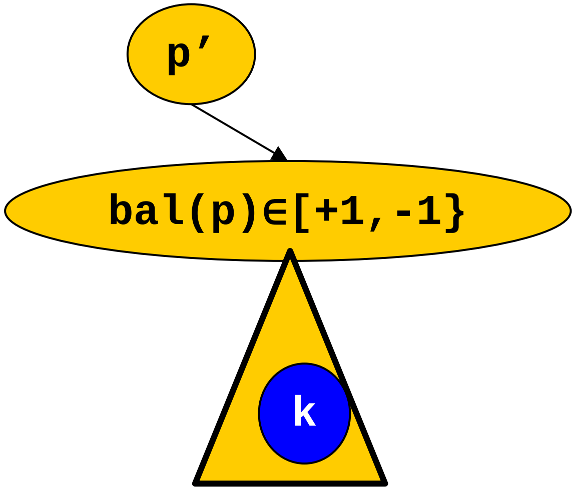
Up the Tree

- In case 3 (bal(p)=0) we have to see if HC is hurt in any of the ancestors of p
- We call a procedure upin(p) recursively
  - We look at the parent p' of p
  - We check bal(p') to see if the height change in p breaks HC in p'
  - If not, we are done
  - If yes, we can either fix it locally or propagate further up the tree
- "Fixing locally" (i.e., with constant work) is the main trick behind AVL trees
- It implies that we never have to call upin(p) more than O(log(n)) times – the height of any AVL tree with n nodes

Subcases

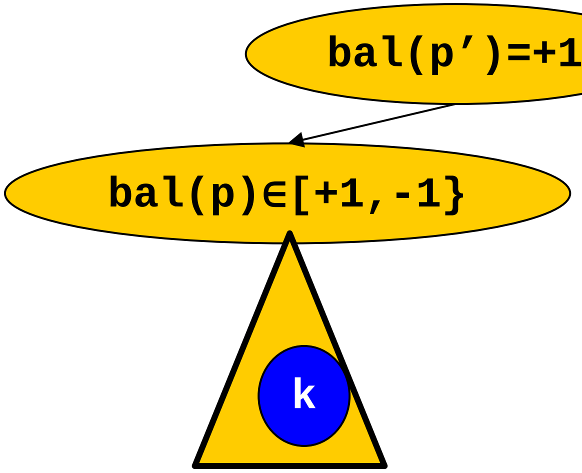
- p can either be the left or the right child of its parent p‘
- Note that bal(p) must be +1 or -1 when upin() is called
  - We call this PC, precondition of upin()
  - In the first call, bal(p)=0 before insertion, thus +1/-1 afterwards
  - In later calls: We have to check!
- Case 3.1 Case 3.2

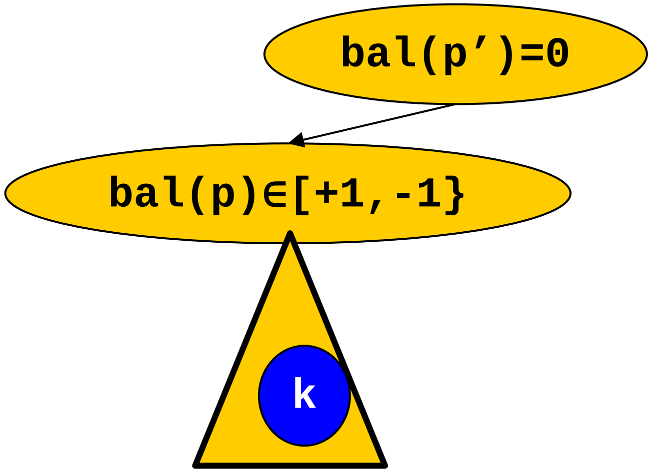




Subcases of Case 3.1

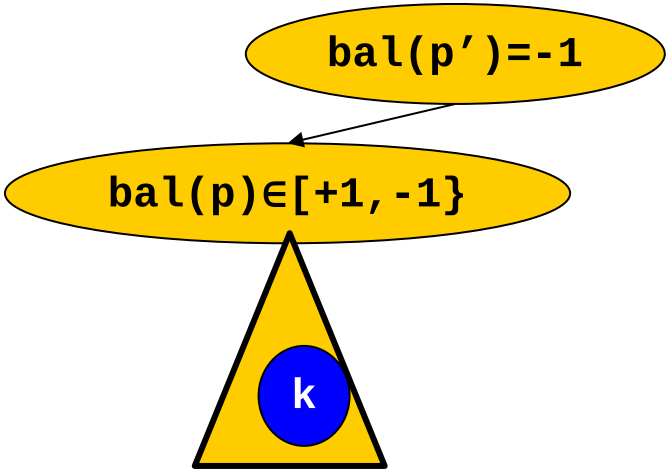
- Case 3.1.1
  - Right subtree of p‘ is higher than left subtree
  - Left subtree has just grown by 1
  - Thus, height of p‘ doesn‘t change
  - Adapt bal(p‘) and we are done
- Case 3.1.2
  - Left and right subtree of p’ have same height
  - Thus, height of p’ changes
  - Adapt bal(p’) and call upin(p’)
    - Note that bal(p’) now is +1 or -1
    - PC holds



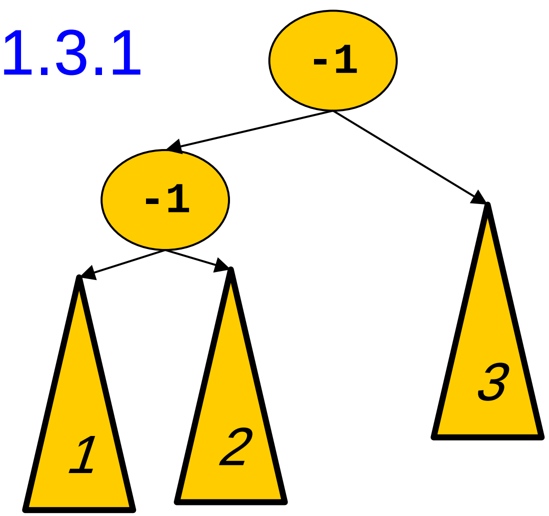


Subcases of Case 3.1

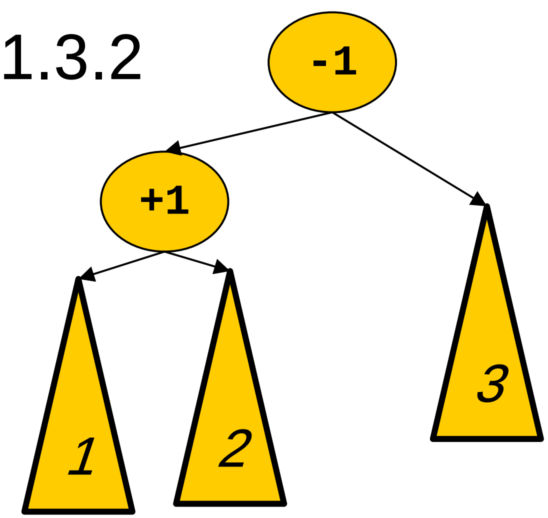
- Case 3.1.3
  - Left subtree of p‘ was already higher than right subtree
  - And has even grown
  - HC is hurt in p’
  - Fix locally
  - How?



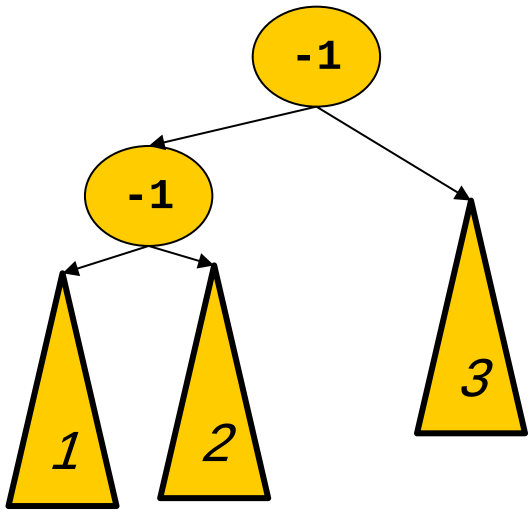
- Case 3.1.3.1



Case 3.1.3.2

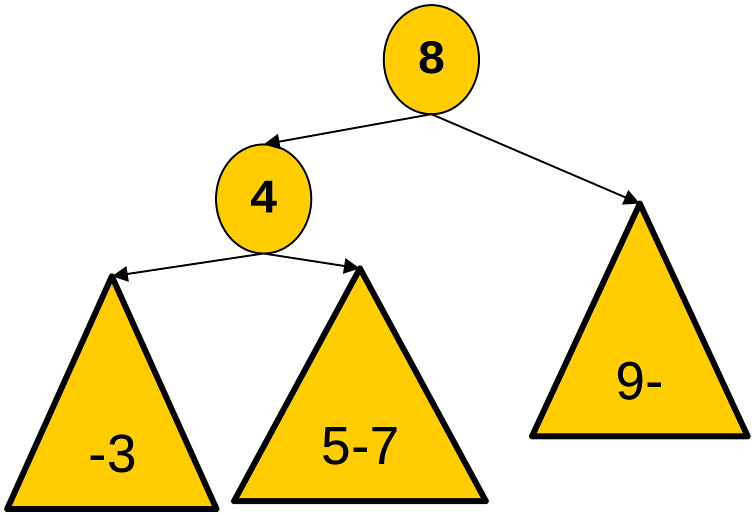


A Closer Look



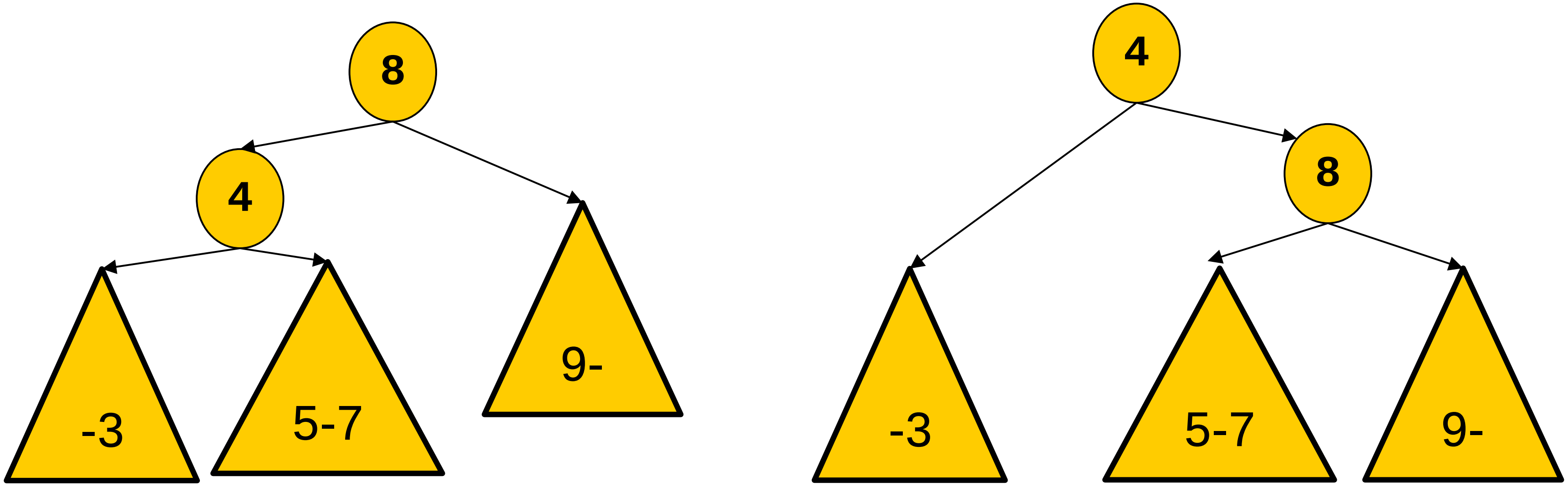
- Subtree 1 contains values smaller than p (and than p')
- Subtree 2 contains values larger than p, but smaller than p'
- Subtree 3 contains values larger than p' (and than p)
- Can we rearrange the subtree rooted in p' such that FC and HC hold?

Example



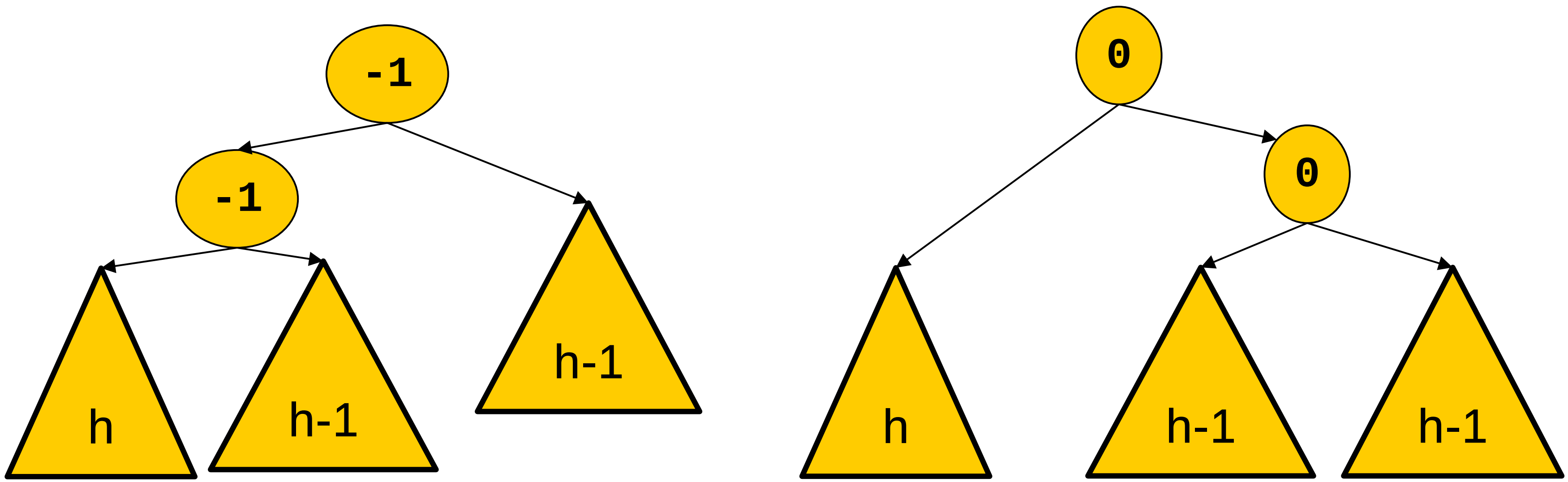
- Subtree 1 contains values smaller than p (and than p')
- Subtree 2 contains values larger than p, but smaller than p'
- Subtree 3 contains values larger than p' (and than p)
- You may change the root node

Rotation



- We rotate nodes p and p' to the left
- Clearly, SC holds
- Impact on HC?

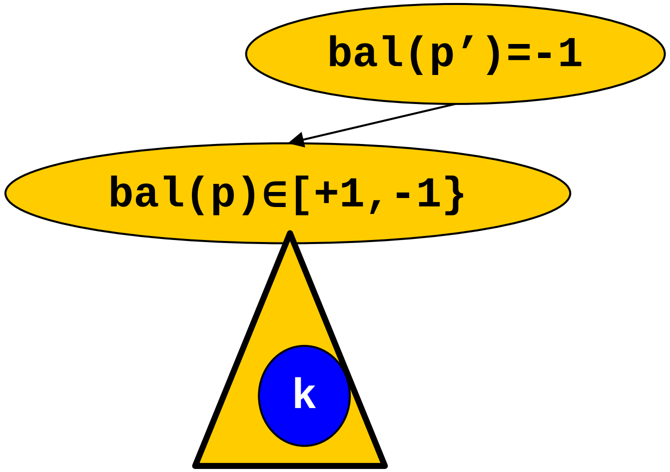
Rotation and HC



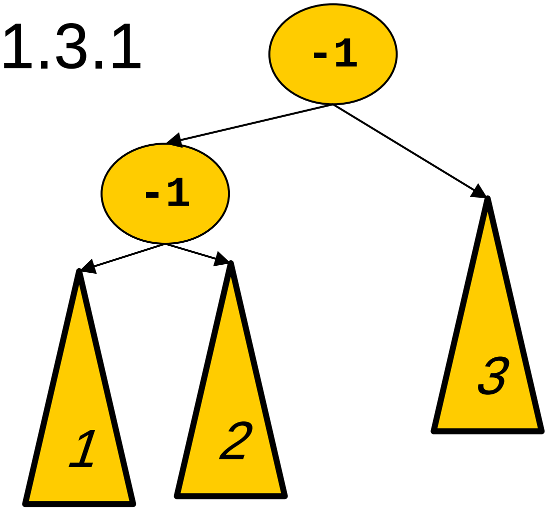
- HC holds after rotation
- Further, height of subtree has not changed – no need for further upin()’s

Recall ...

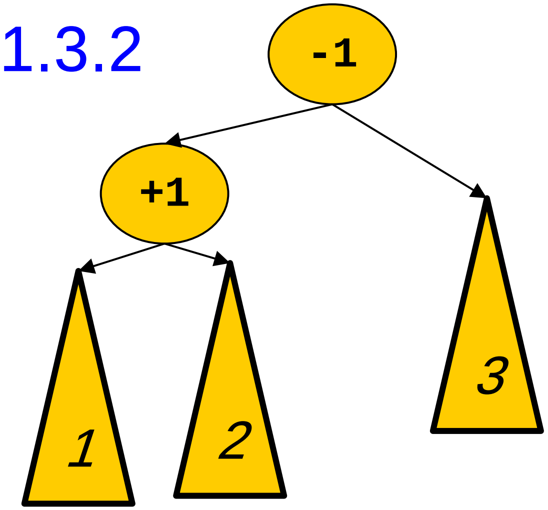
- Case 3.1.3
  - Left subtree of p‘ was already higher than right subtree
  - And has even grown
  - HC is hurt in p’
  - Fix locally
  - How?



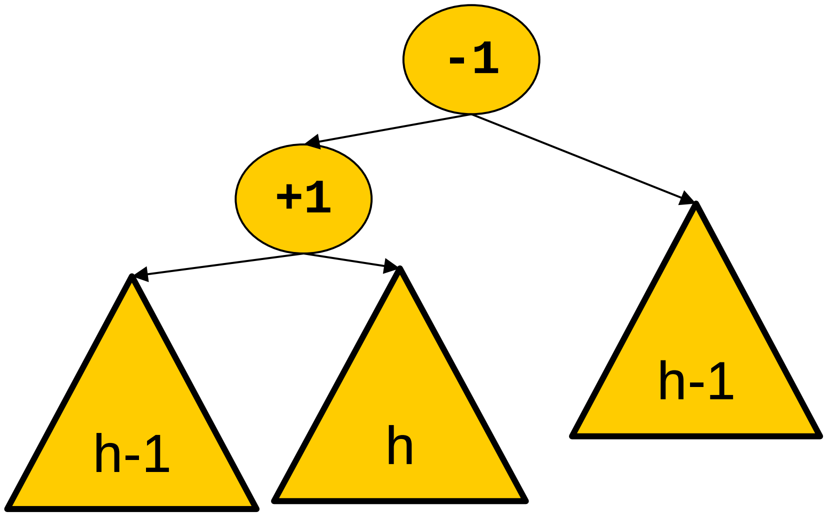
- Case 3.1.3.1



Case 3.1.3.2

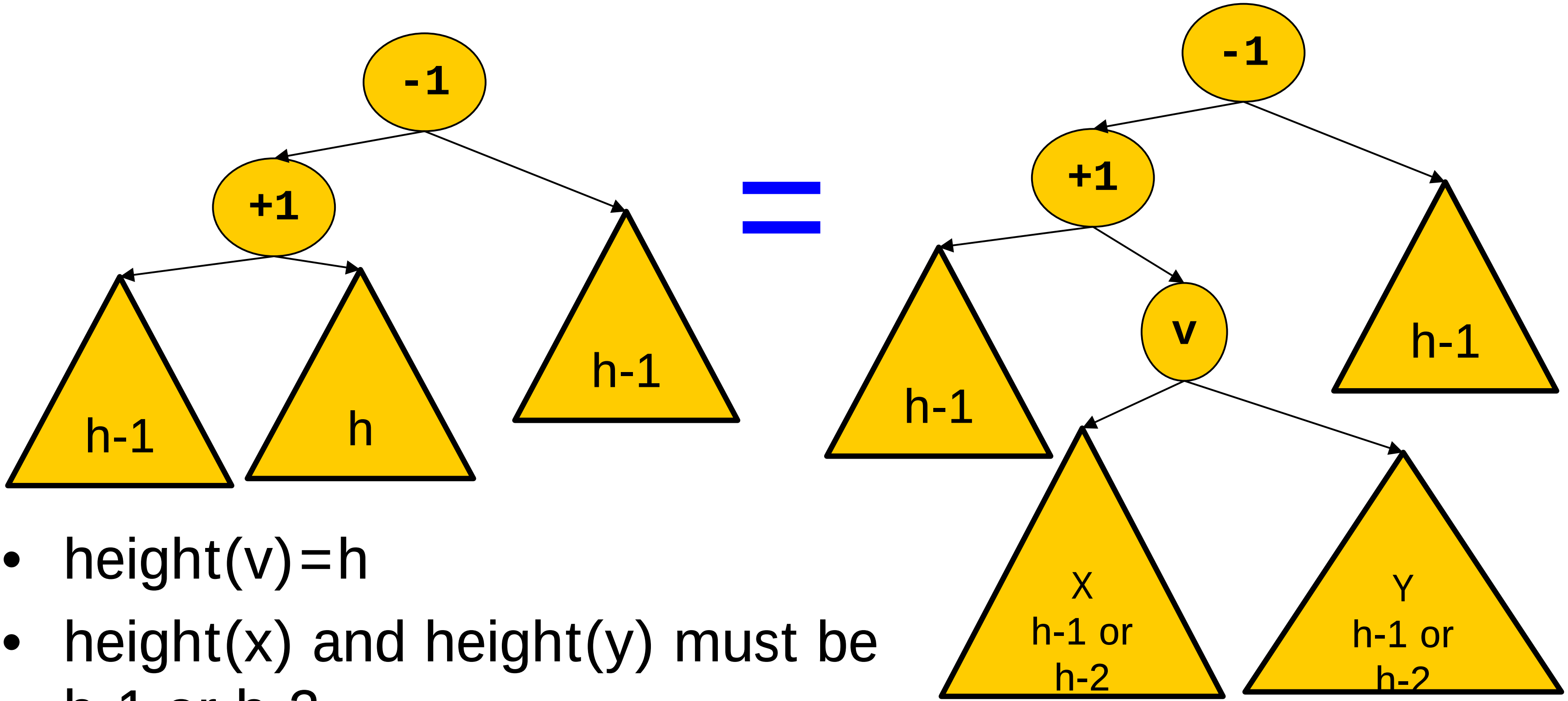


More Intricate



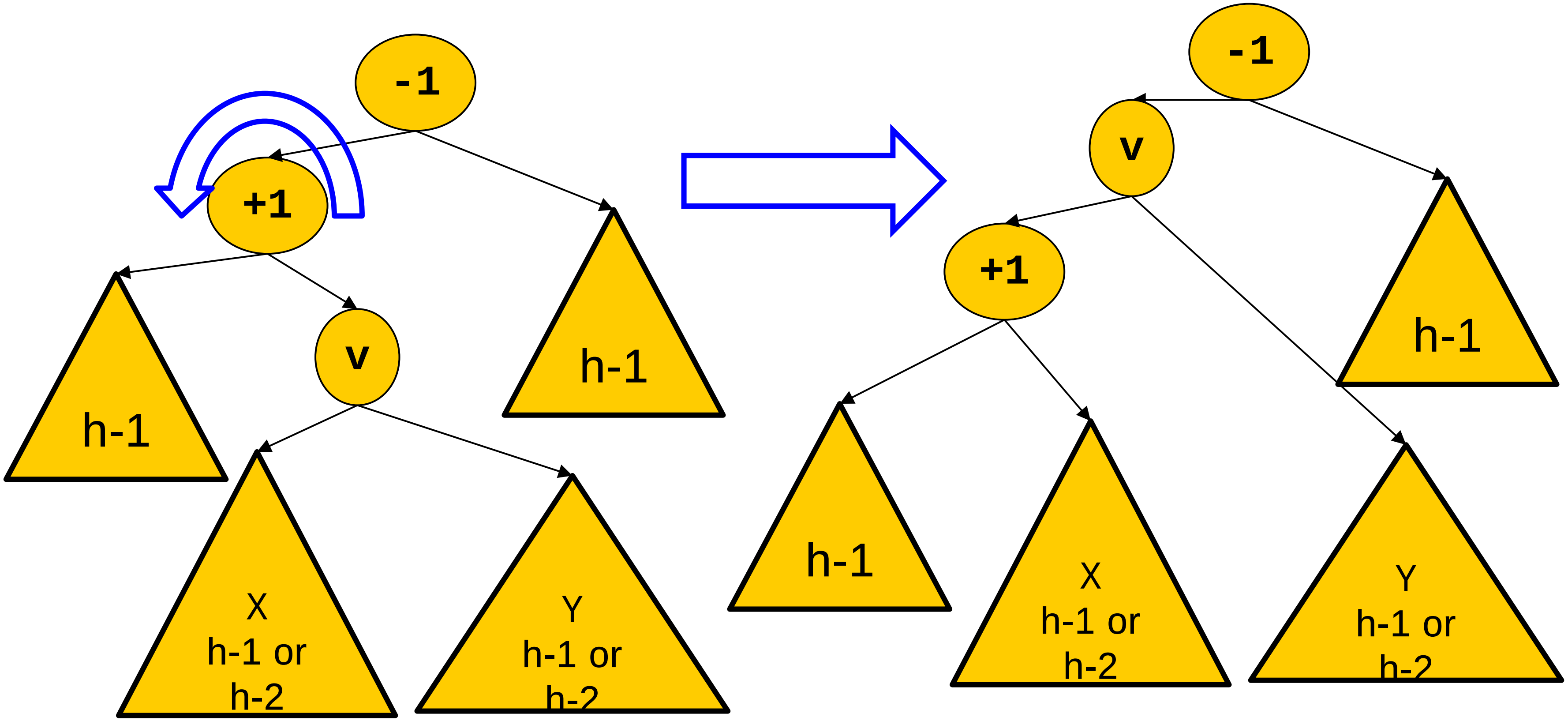
- If we rotated to the right, p (the new root) would have a left subtree of height h-1 and a right subtree of height h+1
  - Forbidden by HC
- We have to taker a closer look to “break” the subtree of height h

One More Level of Detail

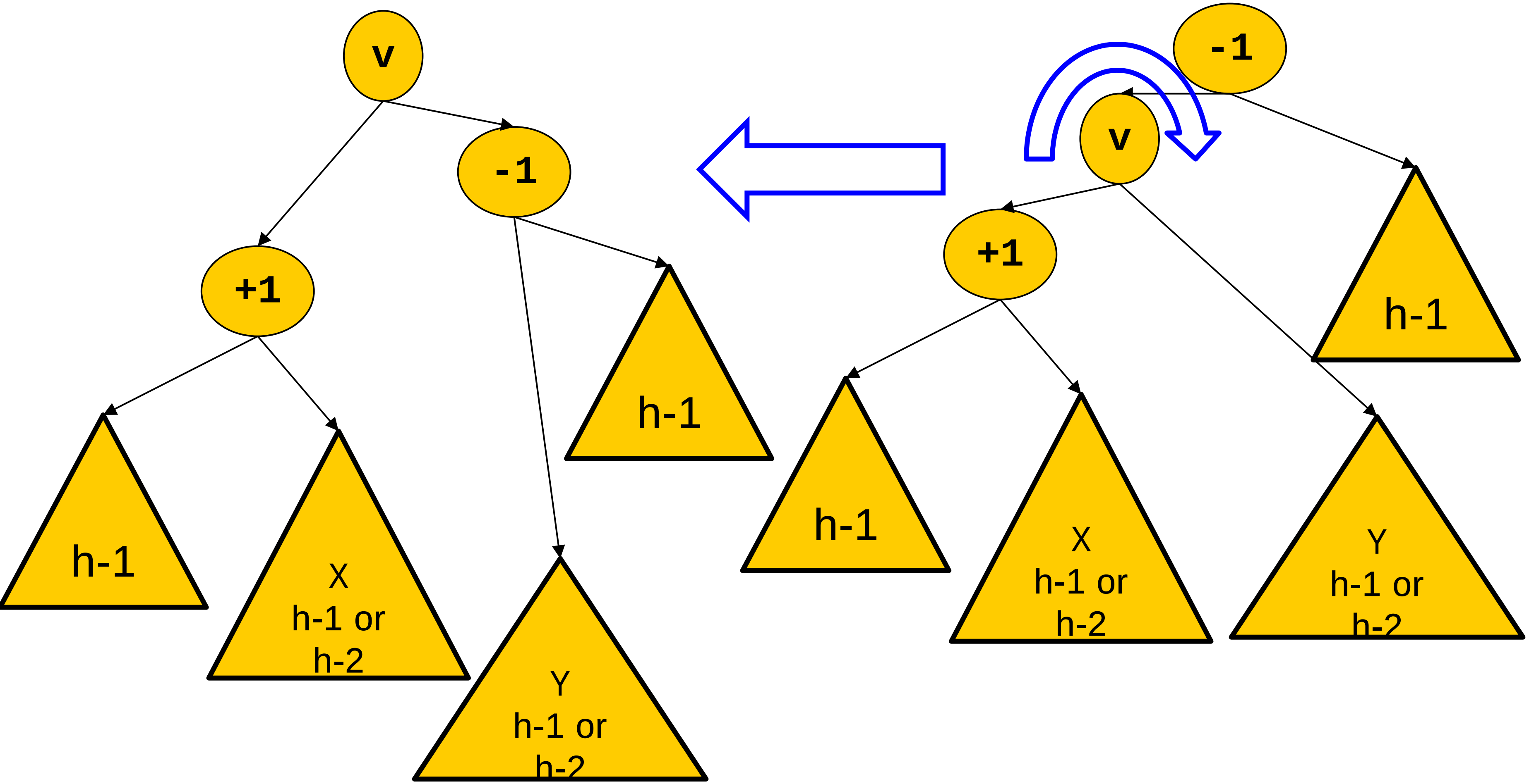


- height(v)=h
- height(x) and height(y) must be h-1 or h-2
- Since the subtree rooted at p has just grown in height, this growth must have happened below v (because bal(p)=+1), so we must have height(x)≠height(y)

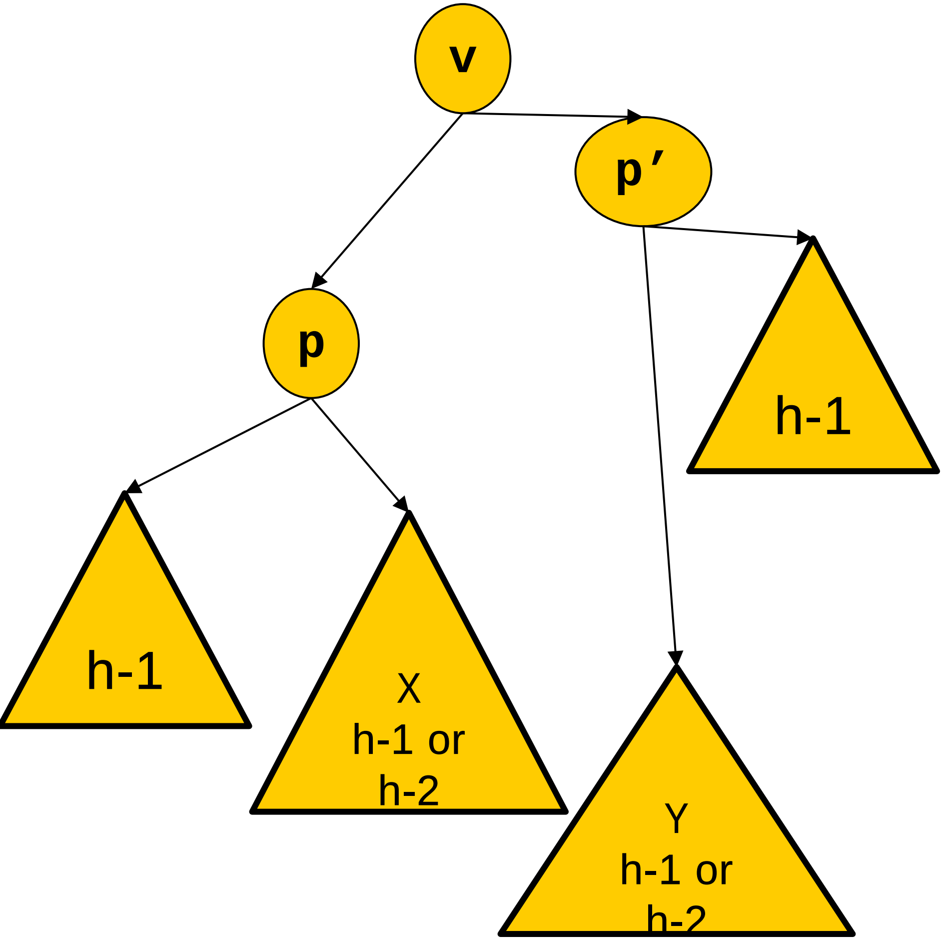
Double Rotation



Double Rotation

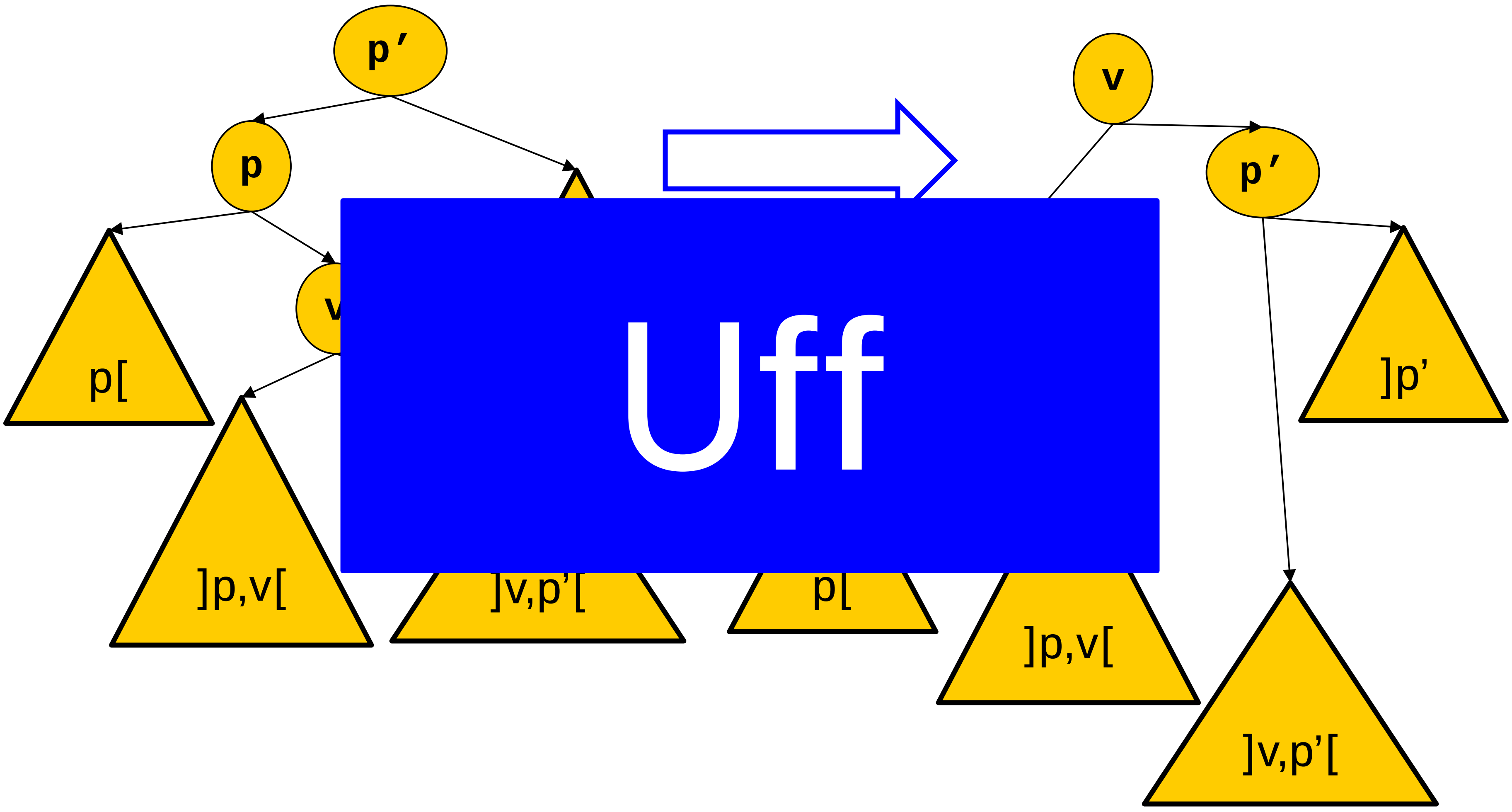


AVL Constraints



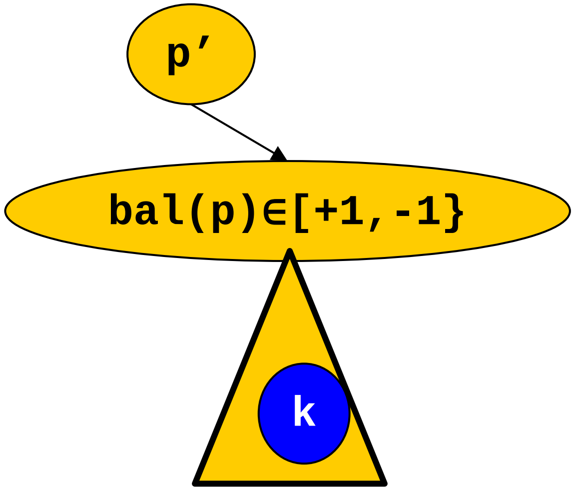
- Adaptation
  - bal(p) ∈ {0, -1}
  - bal(p') ∈ {0, +1}
  - bal(v) ∈ {-1, +1}
- Height constraint
  - Holds in every node
- Need to call upin(v)?
  - No: Subtree had height h+1 and still has height h+1
- Search constraint?

Search Constraint



Are we Done?

- Case 3.2

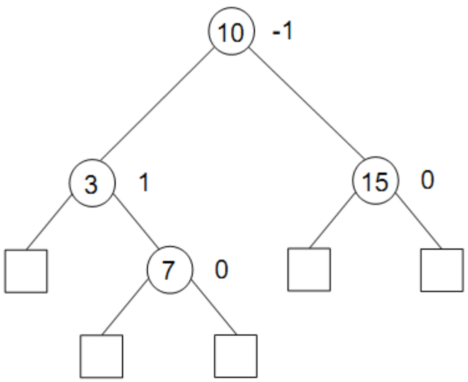


- Similar solution
  - If bal(p’)=-1, adapt and finish
  - If bal(p’)=0, adapt and call upin(parent(p’)
  - If bal(p’)=+1, then
    - Case 3.2.3.1: Rotate left in p
    - Case 3.2.3.1: Rotate right in p, then rotate left in v

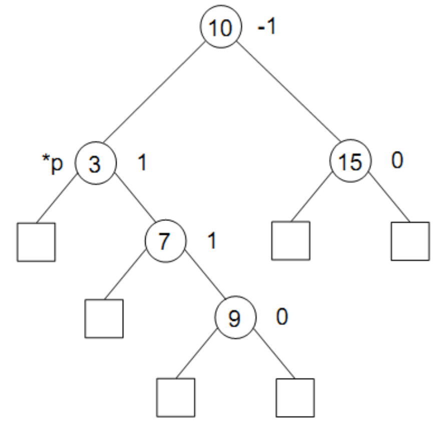
Summary

- We found the node p under which we want to insert k
- Major cases
  - If k<p and rightChild(p)≠null: Insert k (new left child)
  - If k>p and leftChild(p)≠null: Insert k (new right child)
  - If p has no children: Insert k and call upin(p)
- Procedure upin(p)
  - If p=leftChild(p')
    - If bal(p')=1: Set bal(p')=0, done
    - If bal(p')=0: Set bal(p')=-1, call upin(p')
    - If bal(p')=-1:
      - If bal(p)=-1: Rotate right in p, done
      - If bal(p)=+1: Rotate left in p, right in v, done
  - Else (p=rightChild(p'))
    - …

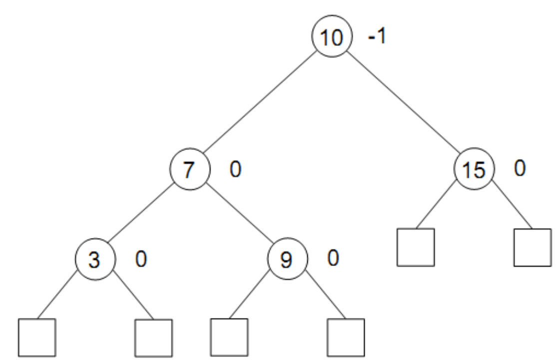
Example



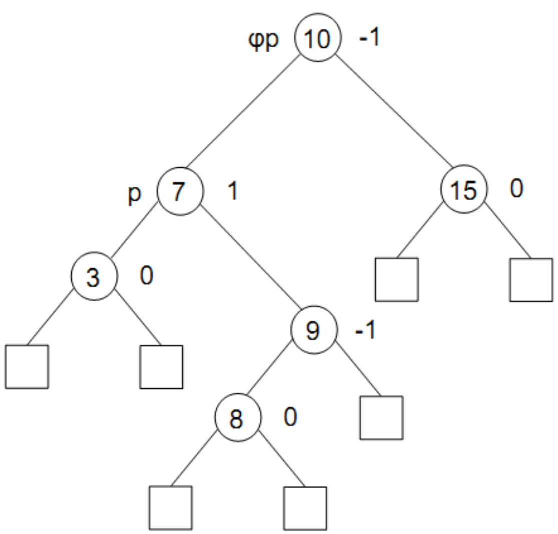
insert 9



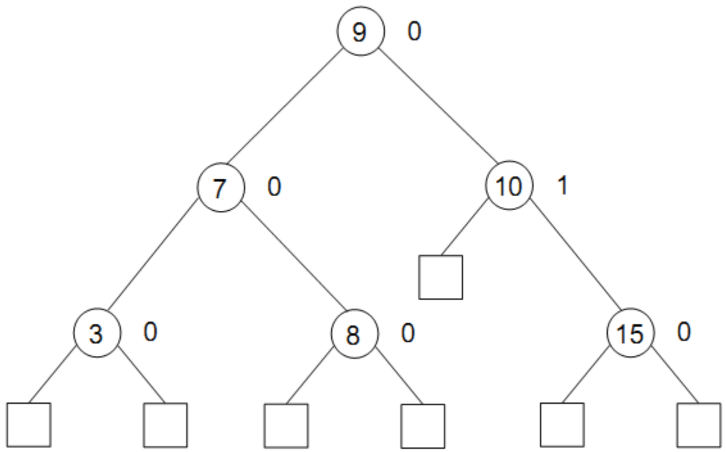
- HC hurt in p
- rotate left in p

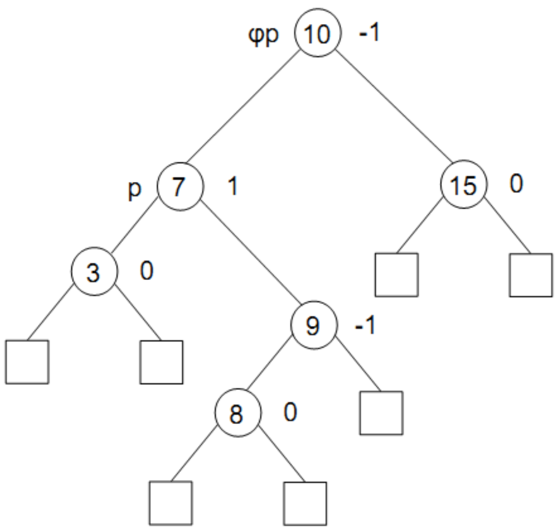


insert 8



Example





- p changes height
- HC hurt in root
- Rotate left in p, then right in root

Content of this Lecture

- AVL Trees
- Searching
- Inserting
- Deleting

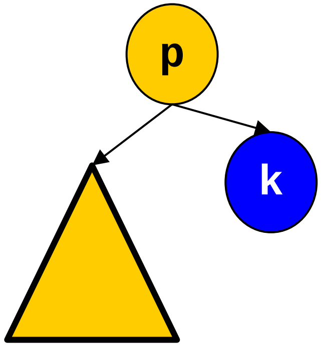
Deleting a Key

- Follows the same scheme as insertions
- We will be a bit more sloppy than for insertions – details can be found in [OW]

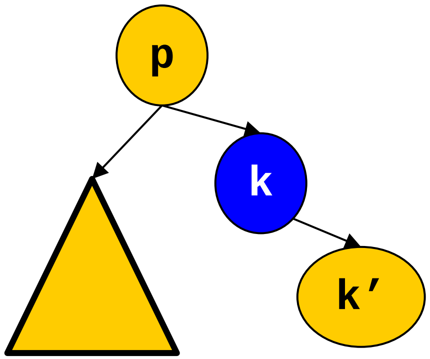
- First find the node p which holds k (to be deleted)
- We will again find cases where we have to do nothing, cases where we have to rotate, and cases where we have to propagate changes up the tree
- Note: In contrast to insertion, whenever we rotate, we still have to propagate changes further
- Thus, on average deletions are more costly than insertions

Major Cases

- Case 1: k has no children
  - Remove k, adapt bal(p)
  - If bal(p) is set to 0, then height has shrunken by 1
    - All other cases are easily resolved locally
  - Then call upout(p)

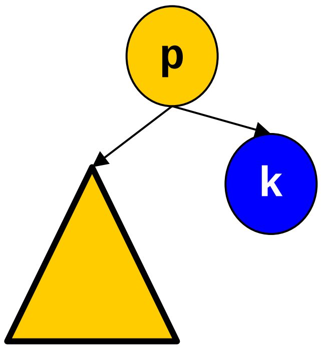


- Case 2: k has only one child
  - Replace k with k‘
    - k‘ cannot have children, or HC would not hold in k
  - Height and balance of k (now k’) has changed
  - Call upout(k’)



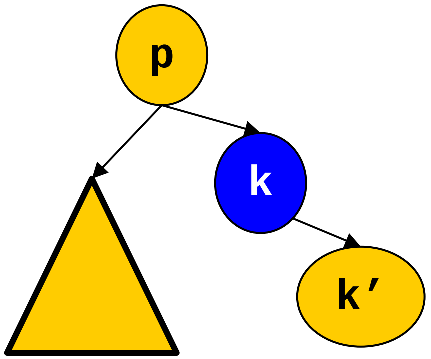
Invariant

- Case 1: k has no children
  - Remove k
  - If bal(p) is set to 0, then height has shrunken by 1
    - All other cases are easily resolved locally
  - Then call upout(p)



- bal(p)=0
- Height of p decreased by 1

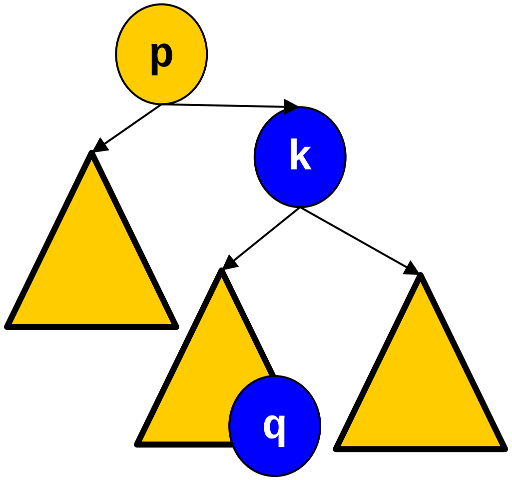
- Case 2: k has only one child
  - Replace k with k‘
    - k‘ cannot have children, or HC would not hold in k
  - Height and balance of k (now k’) has changed
  - Call upout(k’)



- bal(p)=0
- Height of p decreased by 1

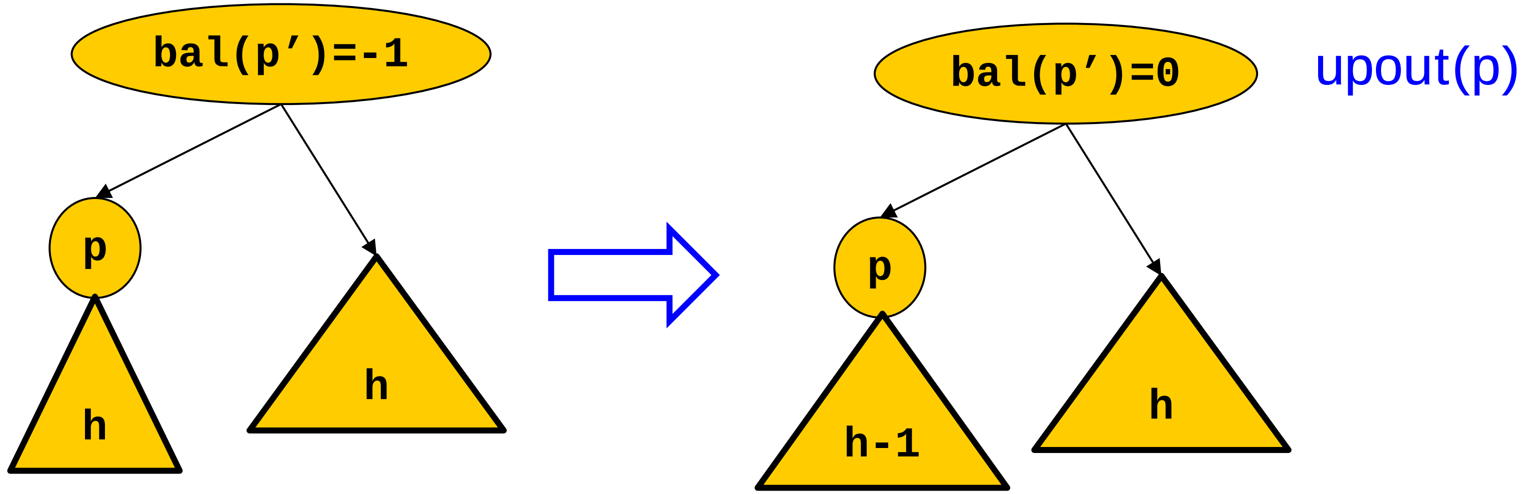
Case 3

- Case 3: k has two children
  - Recall natural search trees
  - We search the symmetric predecessor q of k
  - Replace k with q and call delete(q)



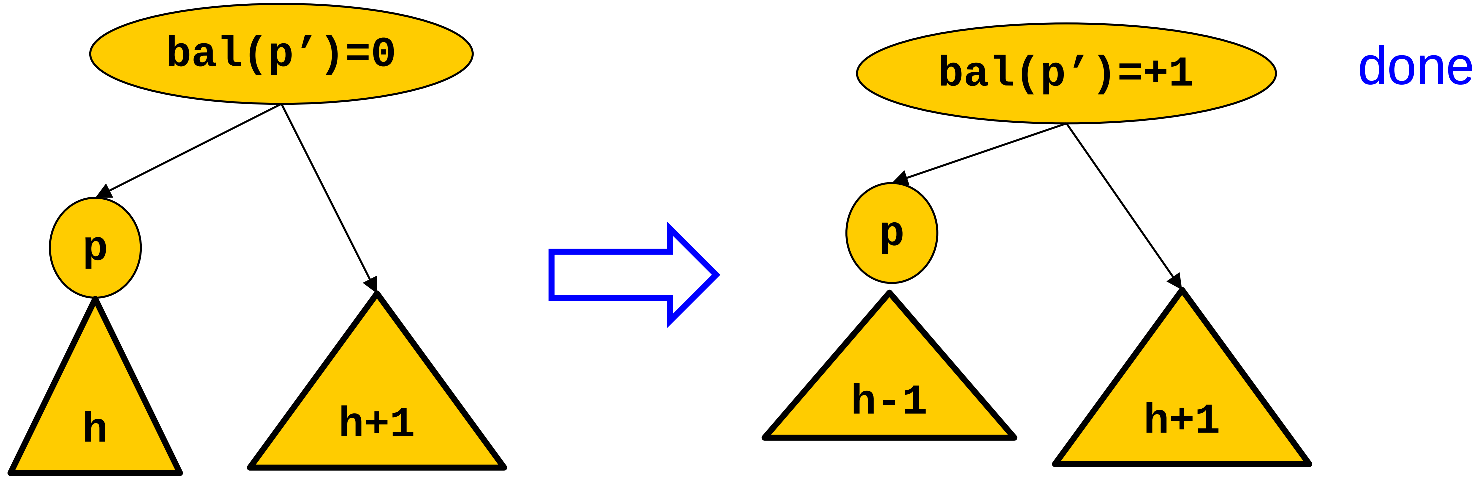
Procedure upout(p)

- Whenever we call upout(p), then the height of p has decreased by 1 and bal(p)=0
- Let p be the left child of its parent p'
  - Again, the case of p being the right child of p' is symmetric
- Case 1; bal(p')=-1



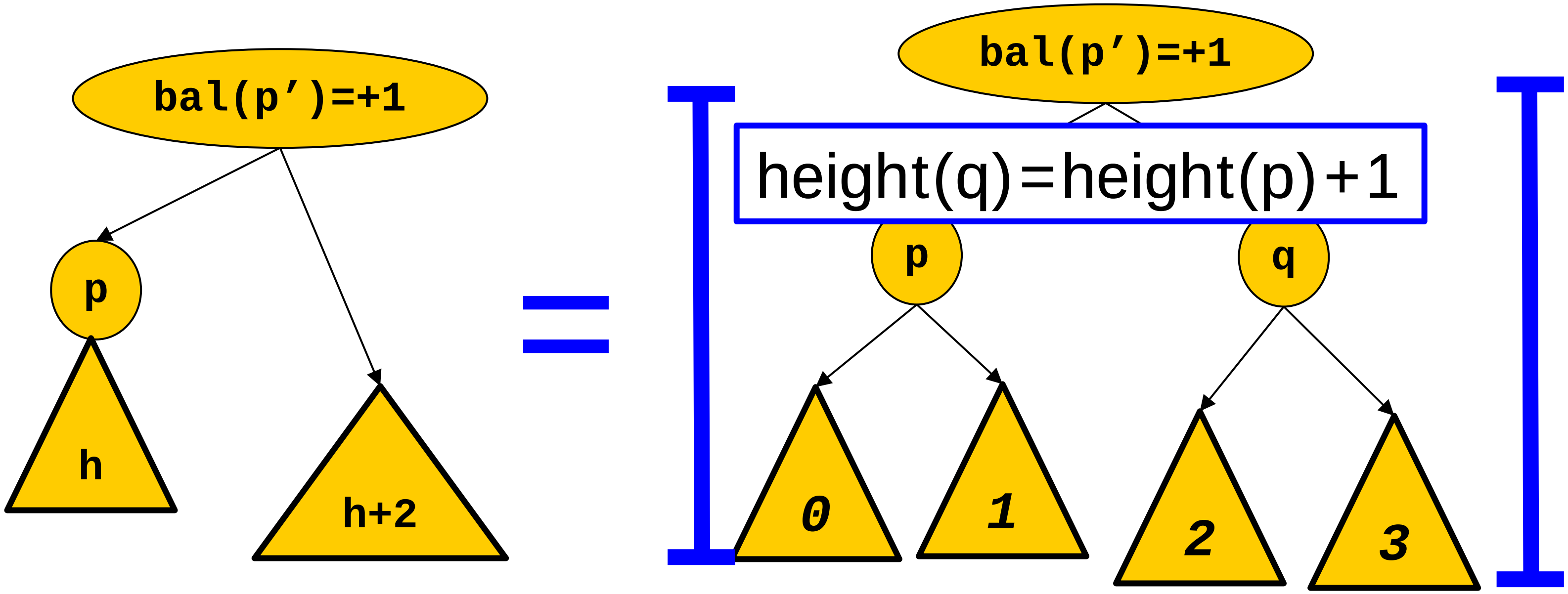
Procedure upout(p)

- Whenever we call upout(p), then the height of p has decreased by 1 and bal(p)=0
- Let p be the left child of its parent p'
  - Again, the case of p being the right child of p' is symmetric
- Case 2: bal(p')=0



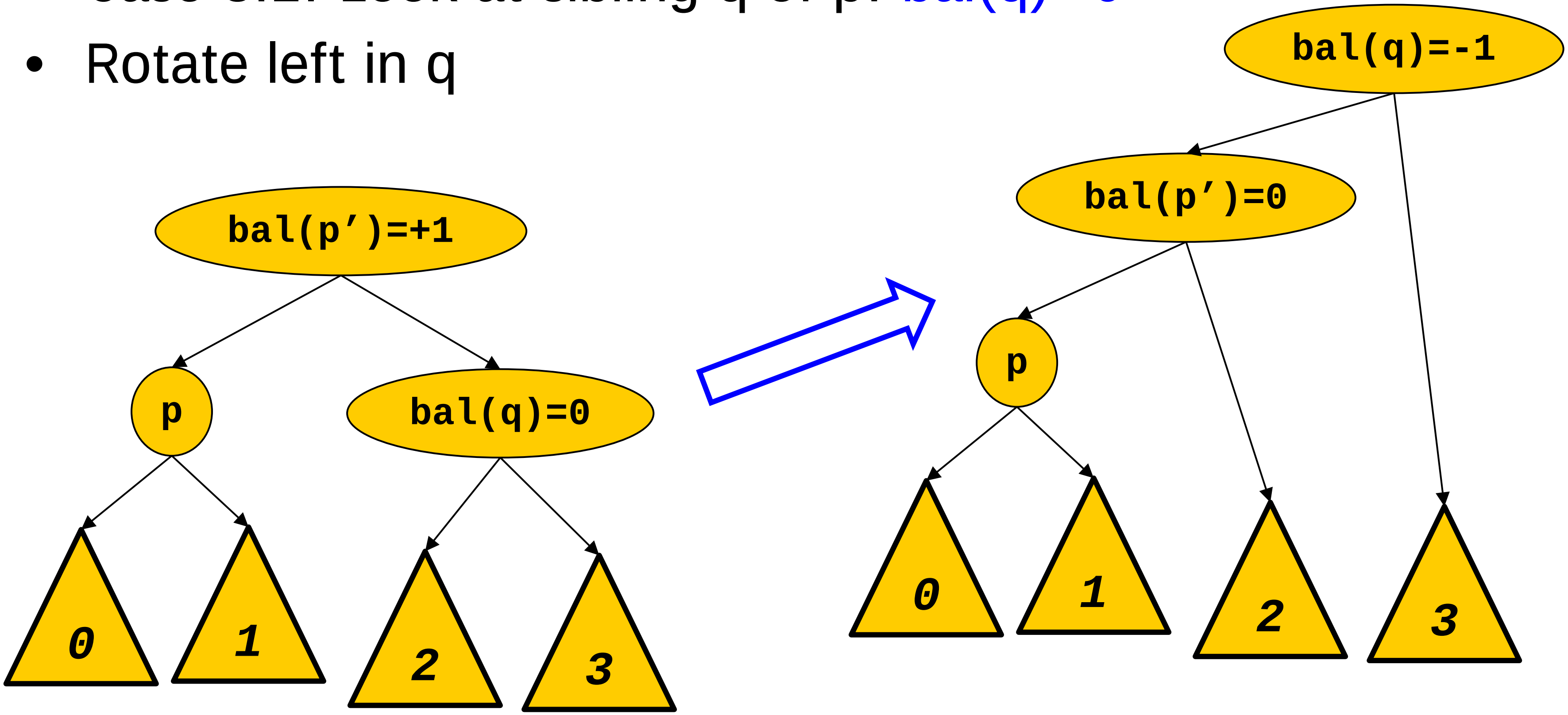
Procedure upout(p)

- Whenever we call upout(p), then the height of p has decreased by 1 and bal(p)=0
- Let p be the left child of its parent p’
  - Again, the case of p being the right child of p’ is symmetric
- Case 3: bal(p’)=+1



Subcase 1

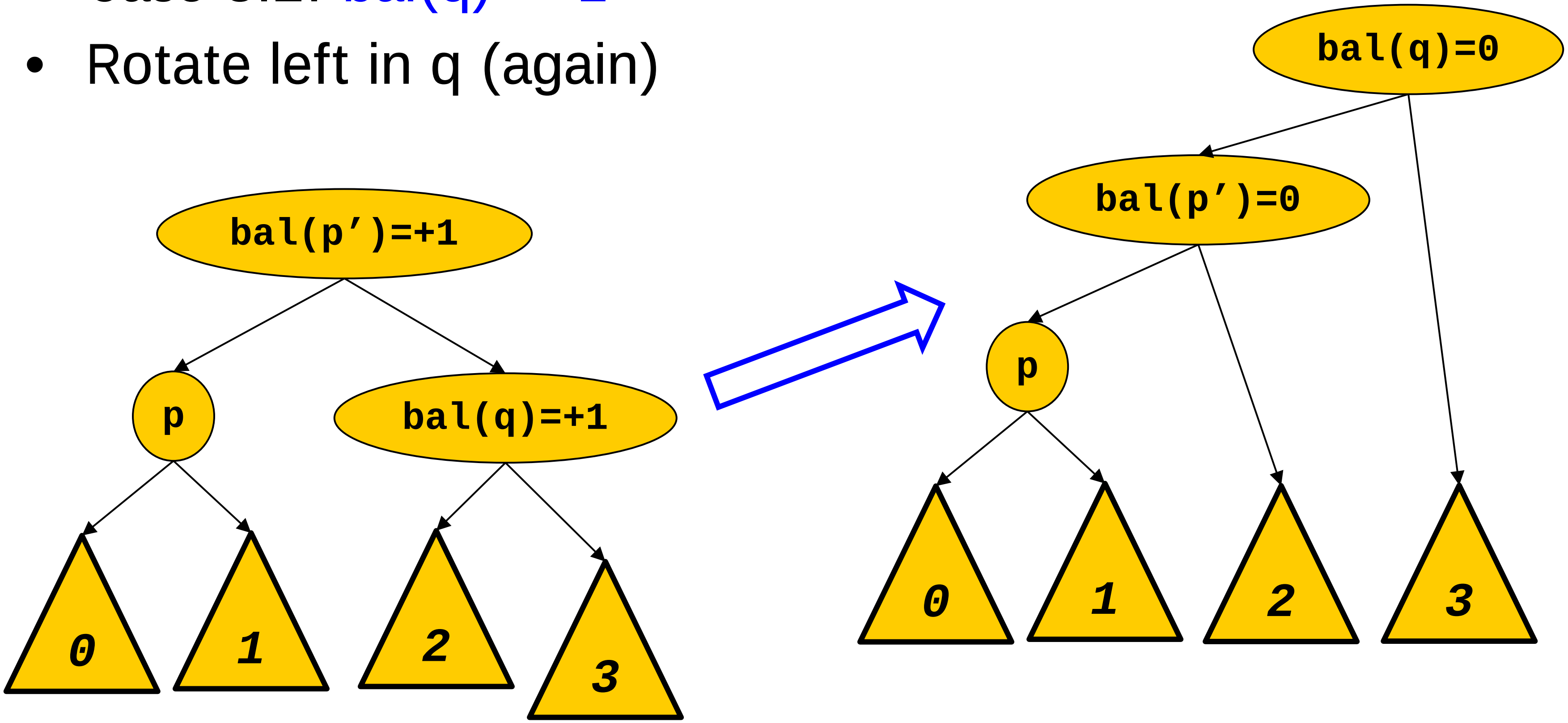
- Case 3.1: Look at sibling q of p: bal(q)=0
- Rotate left in q



Height has not changed - done

Subcase 2

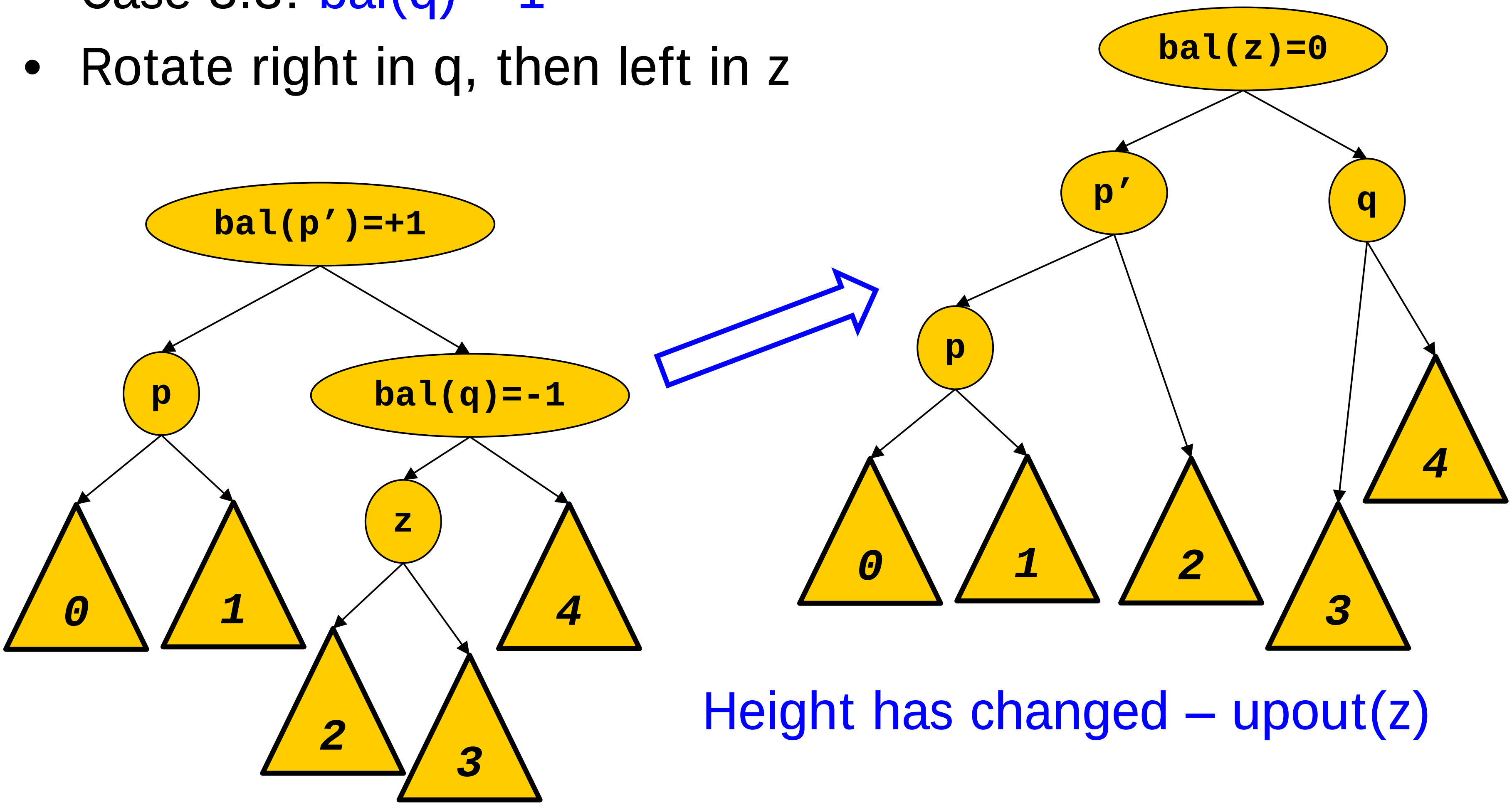
- Case 3.2: bal(q)=+1
- Rotate left in q (again)



Height has changed – upout(q)

Subcase 3

- Case 3.3: bal(q)=-1
- Rotate right in q, then left in z



Height has changed – upout(z)

Summary AVL Trees

- With a little work, we reached our goal: Searching, inserting, and deleting is possible in $O(\log(n))$
- One can also prove that ins/del are in $O(1)$ on average
  - Because reorganizations are rare and usually stop very early
- AVL trees are a “work-horse” DS for keeping a sorted list
  - JAVA uses red-black trees, a class of trees also including AVL trees
- AVL trees are bad as disk-based DS
  - Disk blocks (b) are much larger than one key, and following a pointer means one head seek
  - Better: B-Trees: Trees of order b with constant height in all leaves
    - B typically ~1000
    - Finding a key only requires $O(\log_{1000}(n))$ seeks