Algorithms and Data Structures

AVL: Balanced Search Trees

Marius Kloft

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
- First proposal of balanced trees is attributed to [AVL62]
- Many others since then: brother-, B-, B*-, BB-, … trees

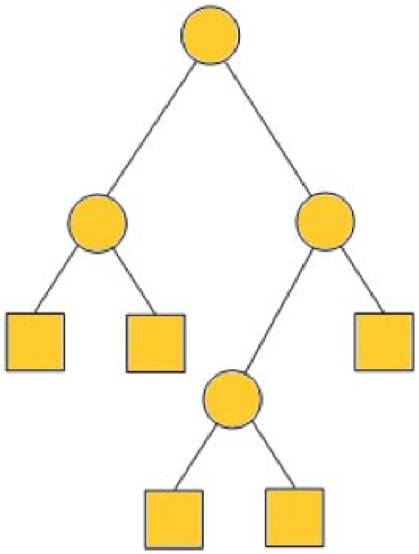
AVL Trees

- Definition

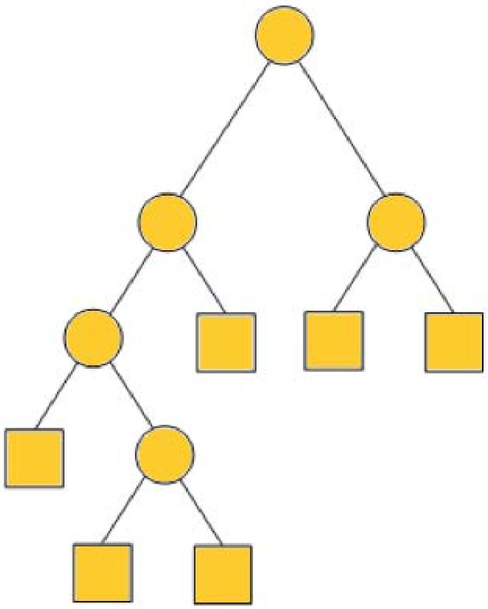
  An AVL tree T=(V, E) is a binary search tree in which the following constraint holds:

  $\forall v \in V: |height(v.leftChild) - height(v.rightChild)| \leq 1$

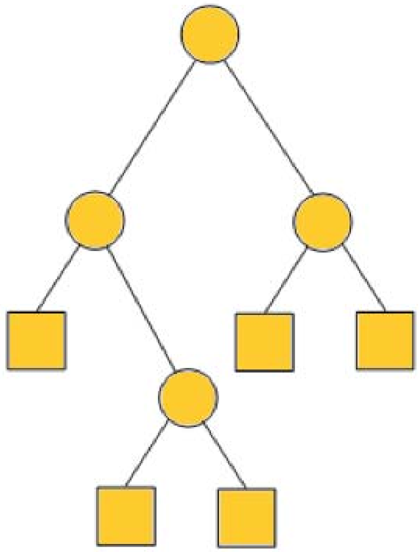
Quiz [source: OW]



AVL?



AVL?



AVL?

Check AVL condition: For all nodes v, *|height(v.leftChild) - height(v.rightChild)| ≤ 1*

AVL Trees

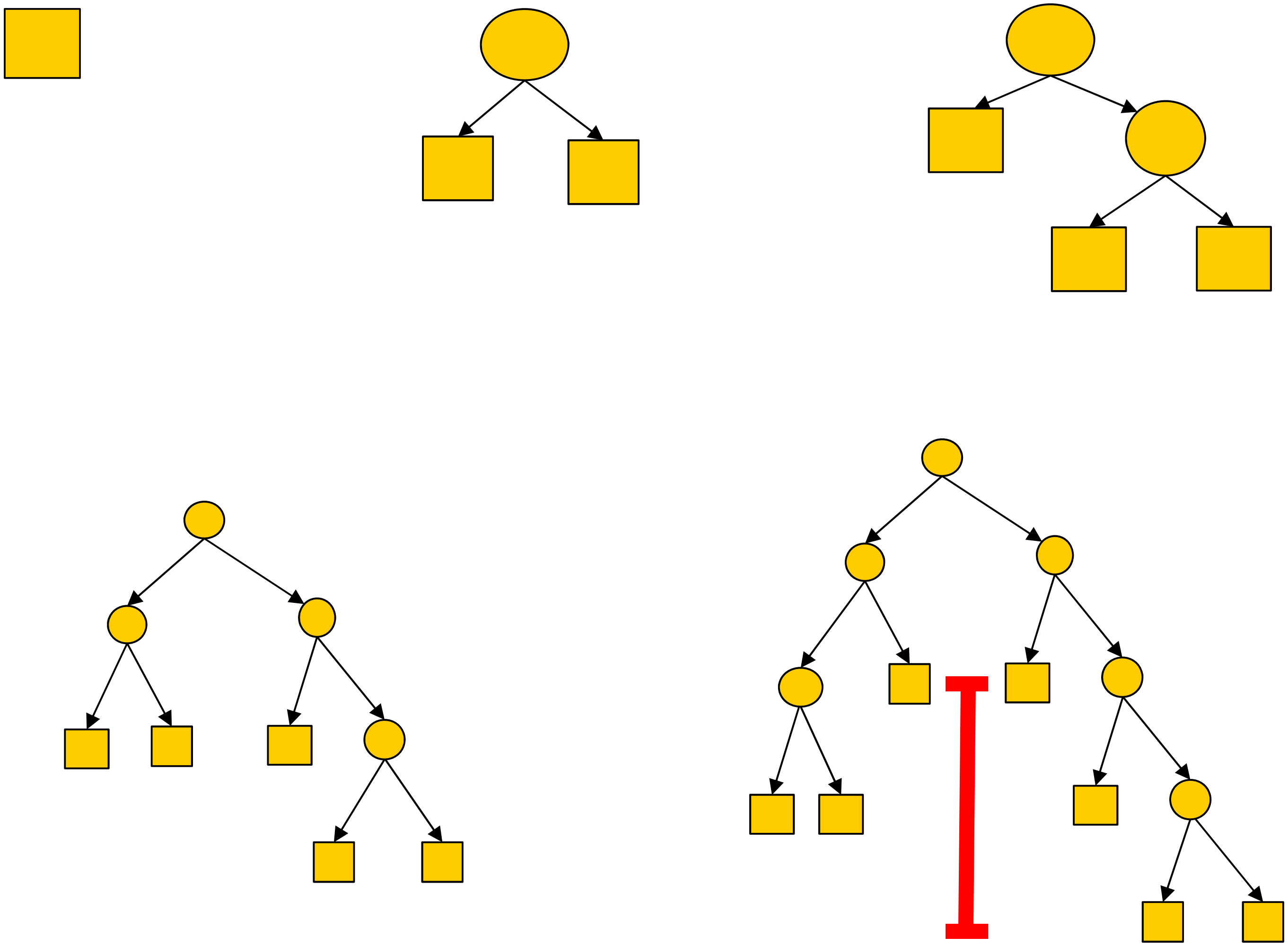
- Definition

  An AVL tree T=(V, E) is a binary search tree in which the following constraint holds:

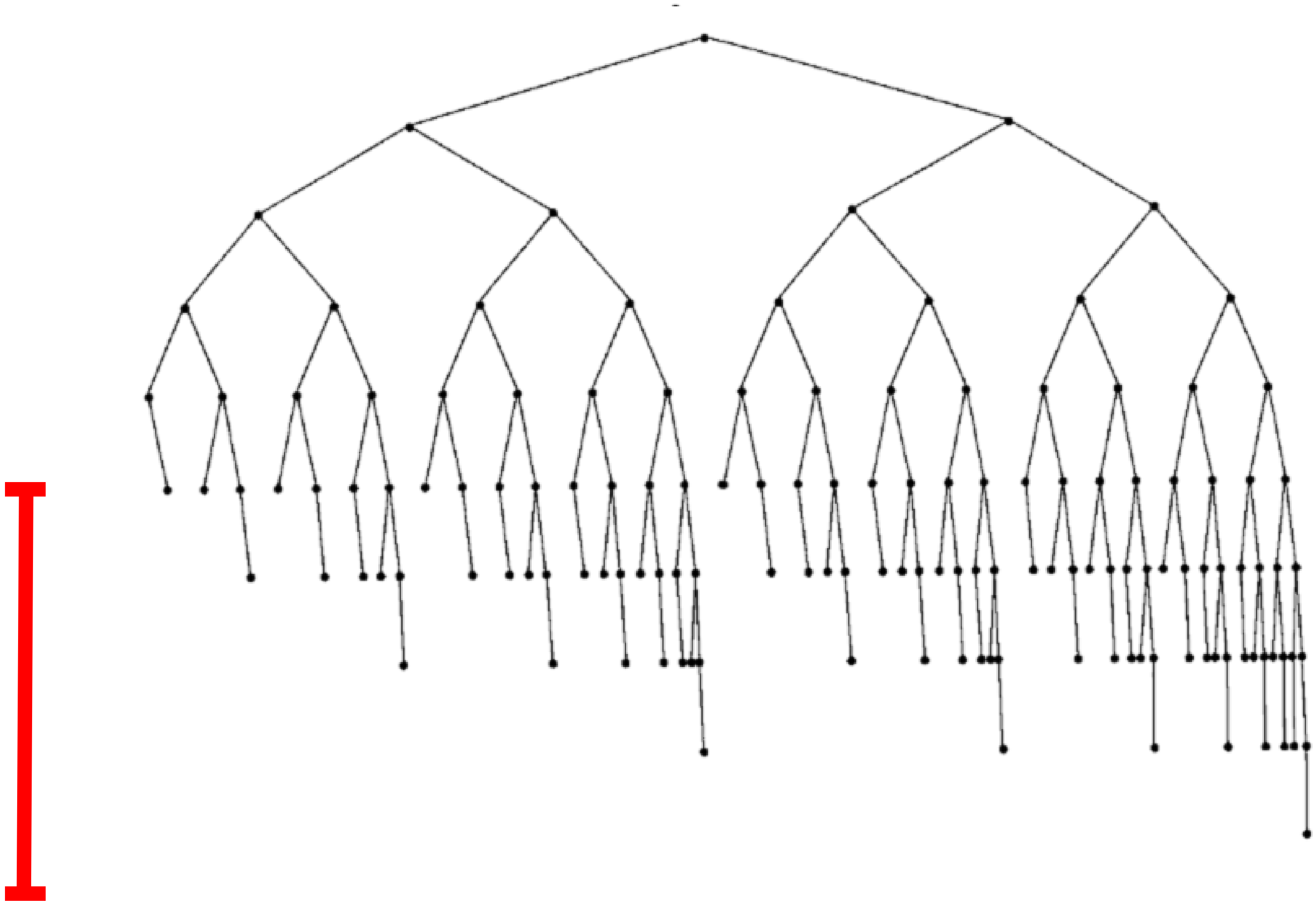
  $\forall v \in V: |height(v.leftChild) - height(v.rightChild)| \leq 1$

- Remarks
  - AVL trees are height–balanced
  - 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

HC Does Not Imply That the Level of All Leaves Can Differ by More Than 1

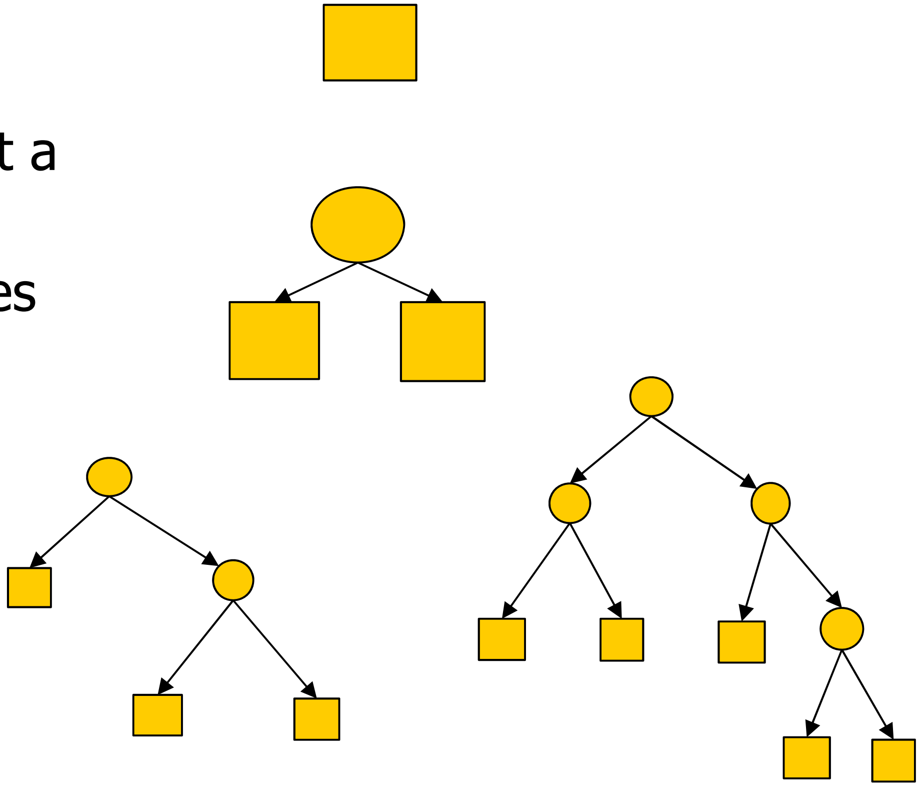


Worst-Case: How “Unbalanced” Can AVL Trees Be?



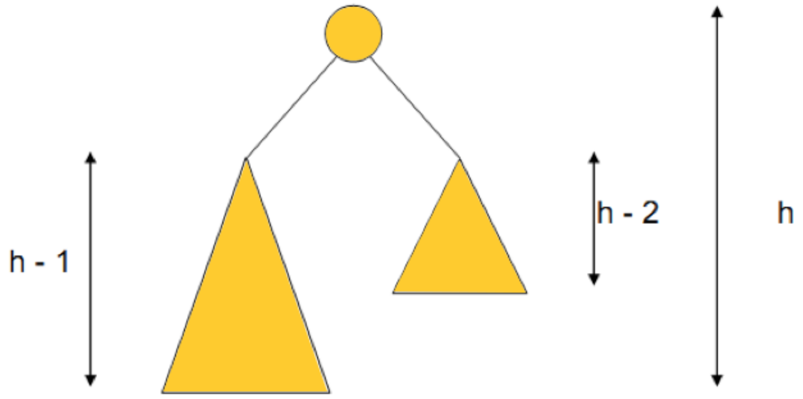
Height of an AVL Tree

- Lemma
  An AVL tree T with n nodes has height h ≤ 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 ⇒ m=1
  - h=1 ⇒ m=2
  - h=2 ⇒ m≥3
  - h=3 ⇒ 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

- Fibonacci series: 0, 1, 1, 2, 3, 5, 8…
  - Def.: fib(0)=0, fib(1)=1, fib(i)=fib(i-1)+fib(i-2)
- Since h "starts" in i=2: $m(h) = \mathrm{fib}(h+2)$
- We know (→ Fibonacci search):
  - fib(i) = round$\left(\frac{\phi^i}{\sqrt{5}}\right) \approx \frac{\phi^i}{\sqrt{5}}$
  - Where ϕ ≔golden ratio ≈ 1.62
- Hence: $m(h) \approx \frac{\phi^{h+2}}{\sqrt{5}}$
- We know n=2m(h)-1, thus

$$n \approx 2 * \frac{\phi^{h+2}}{\sqrt{5}} - 1 \quad \Rightarrow \quad h \leq c * \log(n)$$

Content of this Lecture

- AVL Trees
- Searching
- Inserting
- Deleting

Searching in an AVL Tree

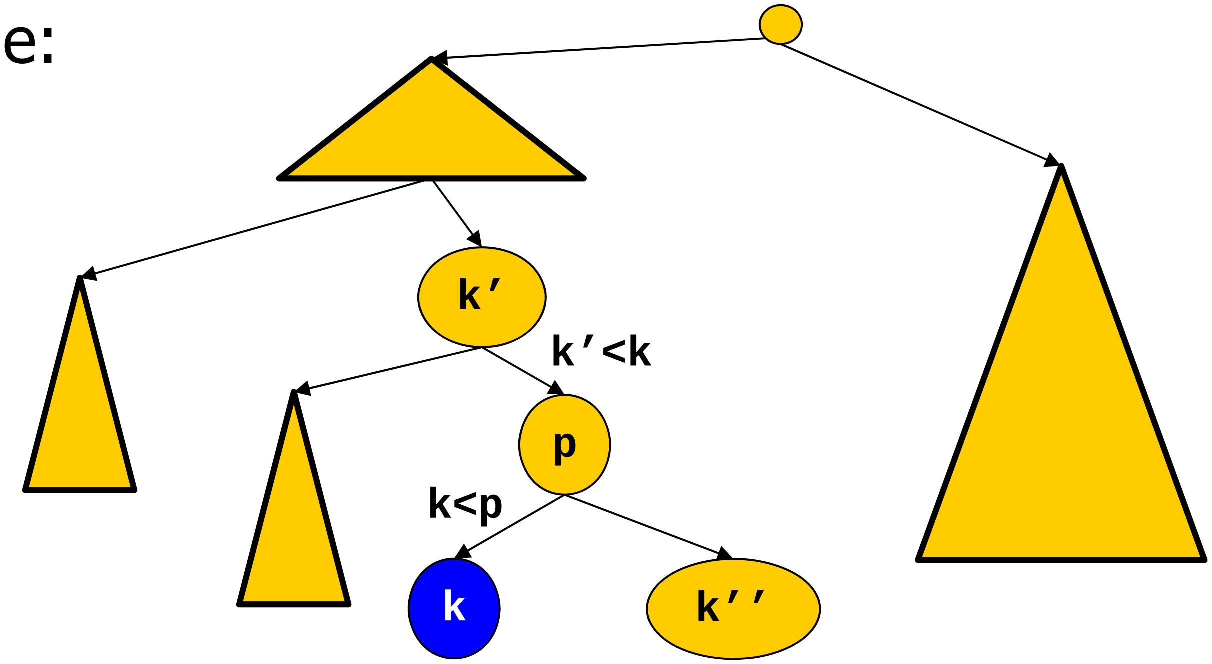
- Searching is in O(log(n))
  - Follows directly from the worst-case height
- Note: The best-case height is ceil(log(n)), so best-case and worst-case asymptotically are of the same order

Inserting

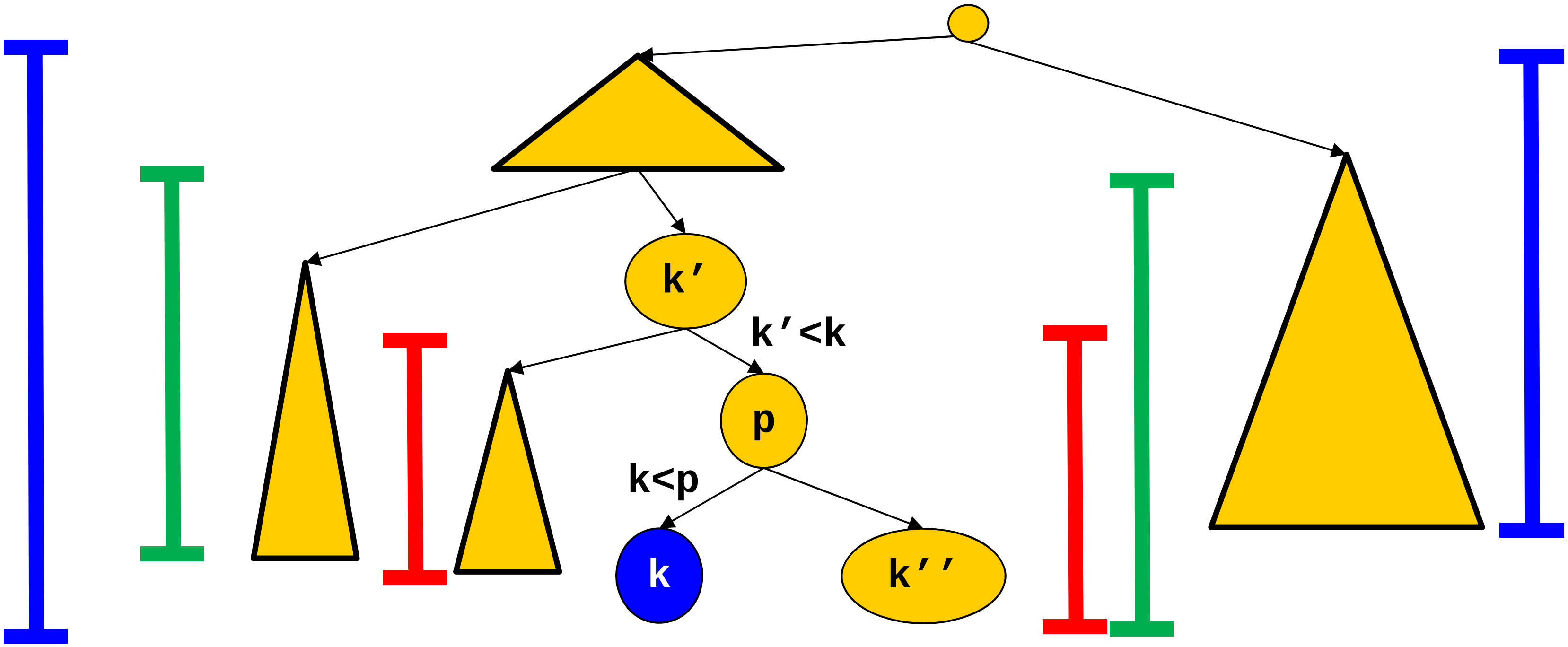
- This requires more work
- The trick is to insert nodes efficiently without hurting the height constraint (HC)
- We first explain the procedure(s) and then prove that HC always holds after insertion of a node if HC held before this insertion

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:

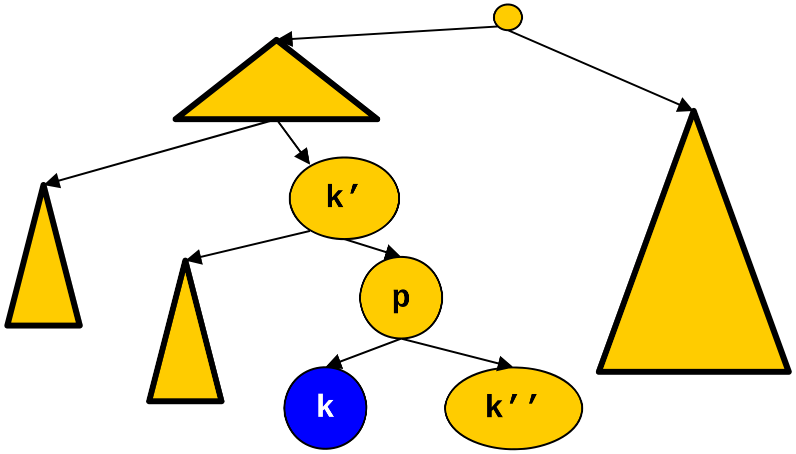


Height Constraints



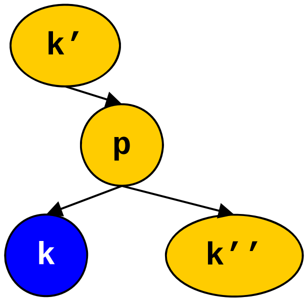
How to Prove the HC

- Before insertion, HC 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



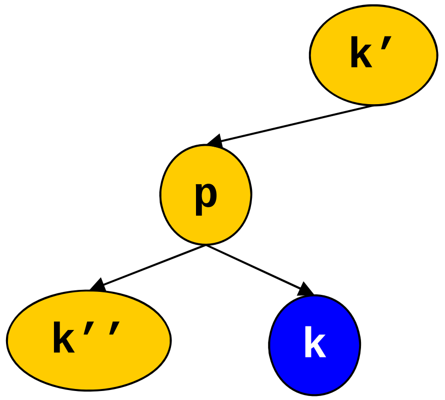
The Essential Information

- Before insertion, HC 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

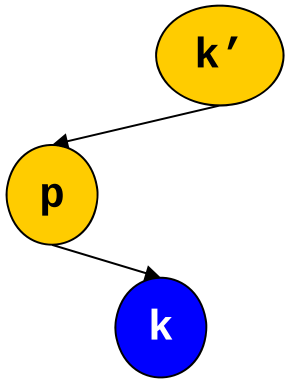


Other Cases

- 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)



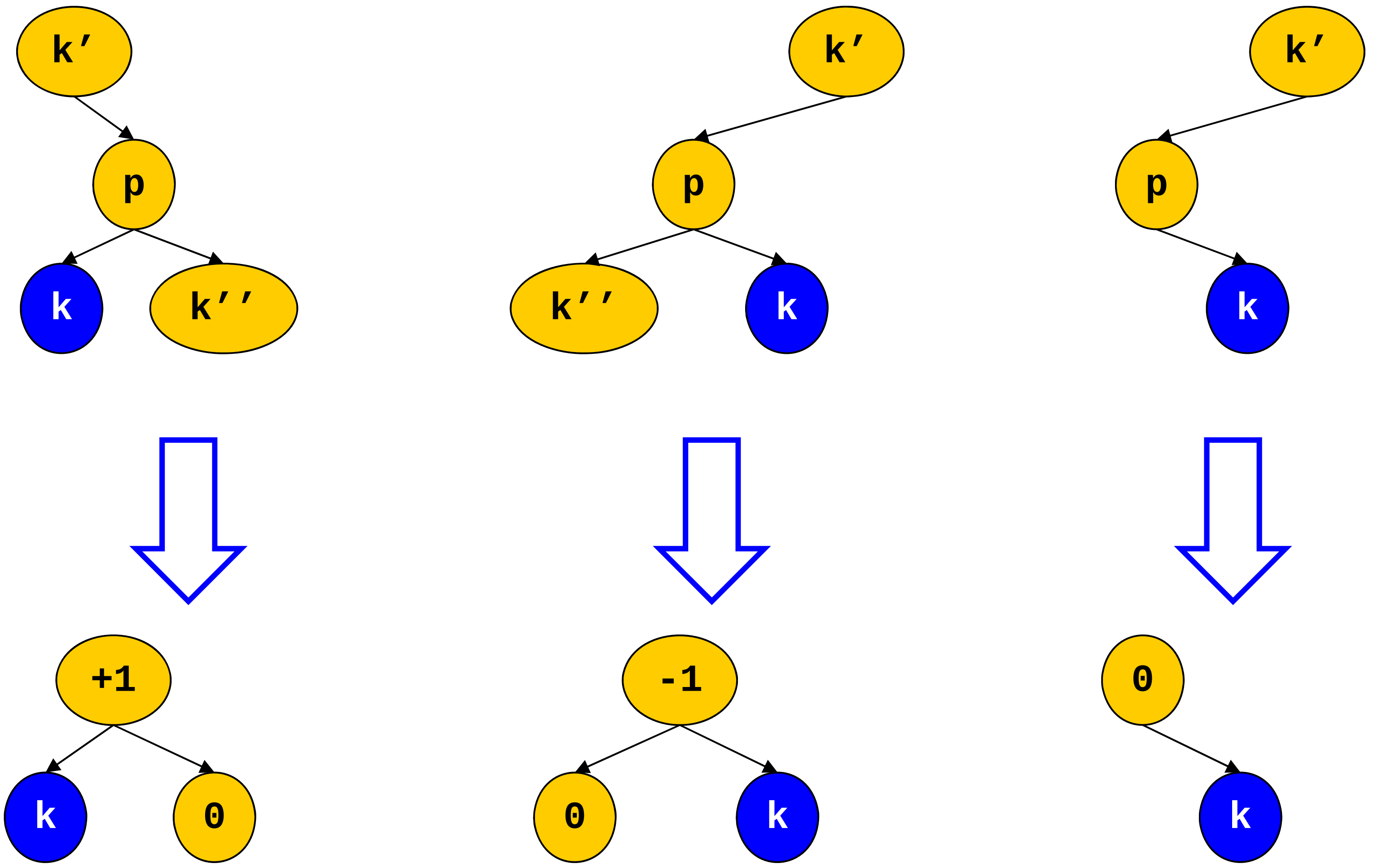
Abstraction

- We assume that we found the position of k such that SC holds after insertion
- 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))

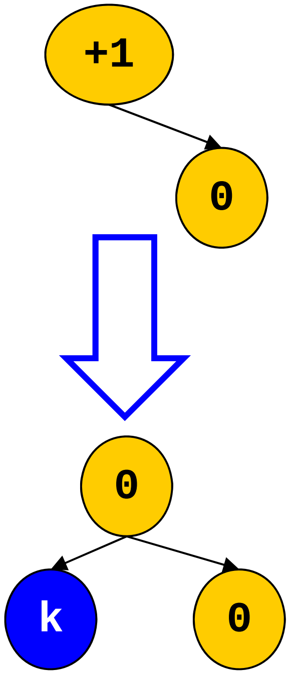
- Clearly, if T is an AVL tree, then ∀p: bal(p) ∈ {-1, 0, 1}

New Presentation



More Systematic

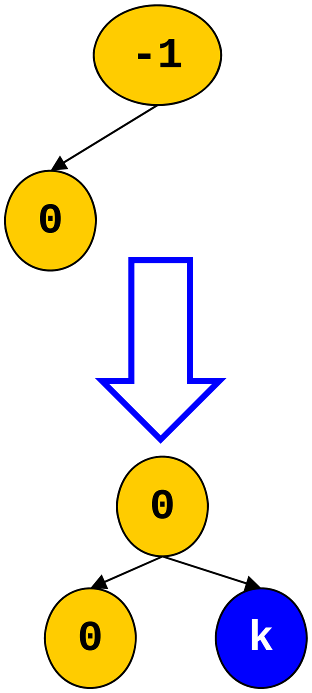
- Assume AVL tree T=(V, E) and we want to insert k, k∉V
- We found the node p under which we want to insert k
- Three possible cases



- 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

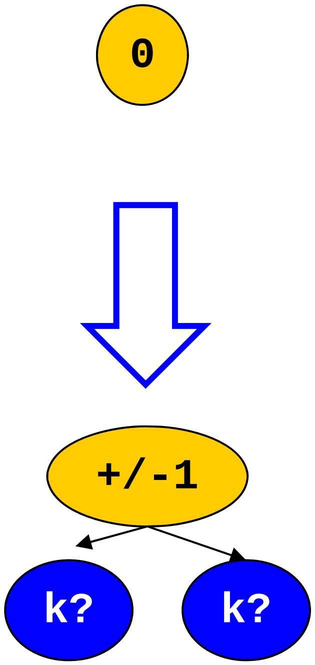
- Assume AVL tree T=(V, e) and we want to insert k, k∉V
- We found the node p under which we want to insert k
- Three possible cases



- Case 2: bal(p)=-1
  - Then there exists a left "subtree" of p (one node only)
  - 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, $k \notin V$
- We found the node p under which we want to insert k
- Three possible cases



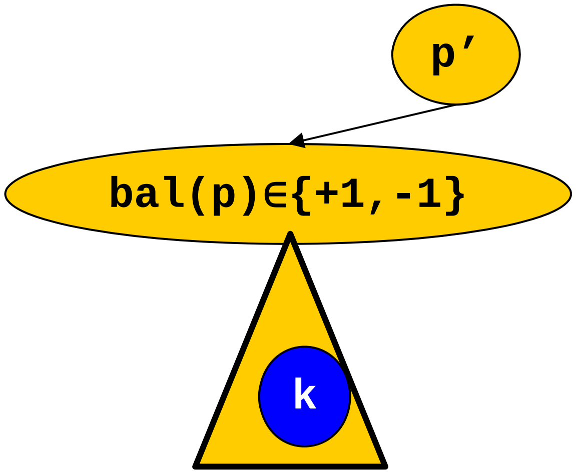
- Case 3: bal(p)=0
  - There is neither a left nor a right subtree of p (p is a leaf)
  - We insert k as left or right child
  - Height of p changes (HC valid?)
  - Ancestors of p are affected
  - Adapt bal(p) and look at parent(p)

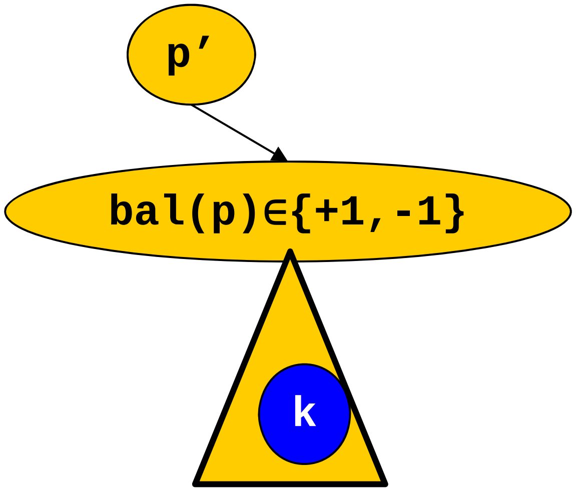
Up the Tree

- 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 update bal(p’) and, if bal(p’) ∈{+1,-1}, call upin(p’)
  - If yes, we fix the problem locally and we are done (no further recursive calls of upin)
- “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 an AVL tree with n nodes

Subcases

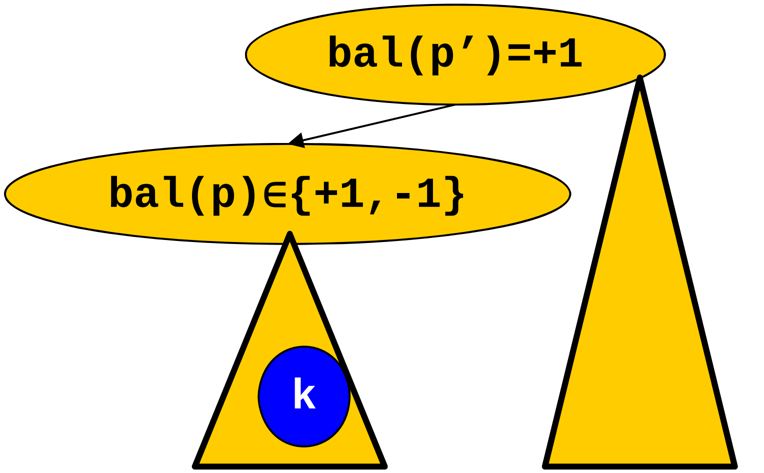
- p can either be the left or the right child of its parent p‵
- Note that bal(p) must be +1 or -1 when upin() is called
  - We call this PC, the precondition of upin()
  - In the first call, bal(p)=0 before insertion, thus +1/-1 afterwards
  - In later calls: We have to check
- Case 3.1 Case 3.2



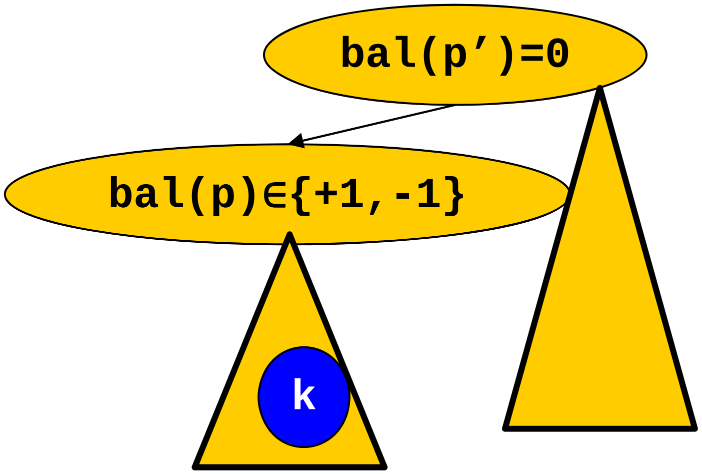


Subcases of Case 3.1

- Case 3.1.1 (bal(p’)=+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

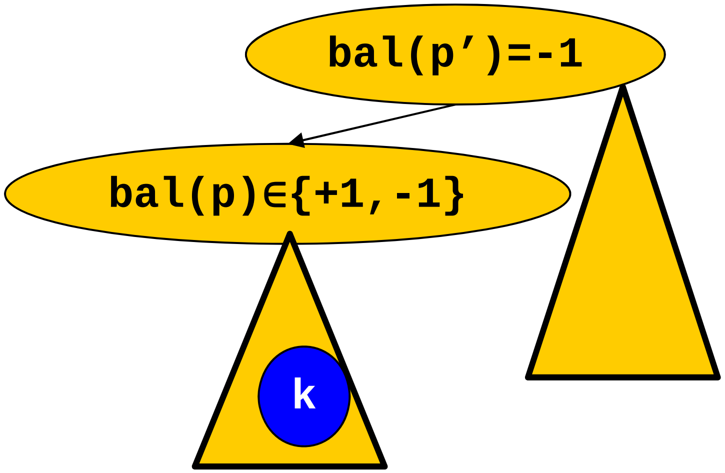


- Case 3.1.2 (bal(p’)=0)
  - Left and right subtree of p’ have same height
  - Thus, height of p’ changes
  - Adapt bal(p’) and call upin(p’)
    - bal(p’) now is -1
    - PC holds

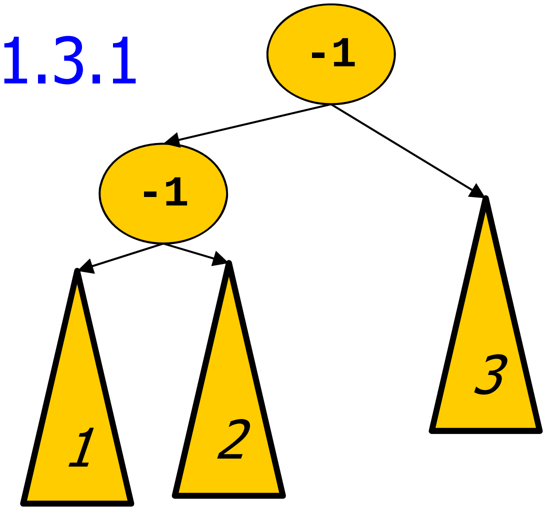


Subcases of Case 3.1

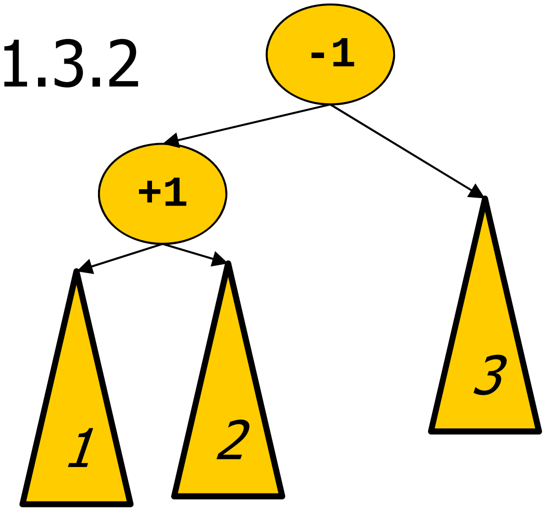
- Case 3.1.3 (bal(p’)=-1)
  - Left subtree of p‘ was already higher than right subtree
  - And has even grown further
  - HC is hurt in p’
  - Fix locally – but how?



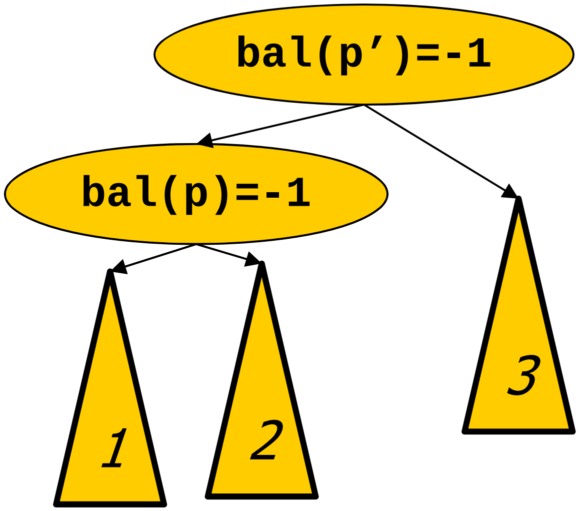
- Case 3.1.3.1



Case 3.1.3.2

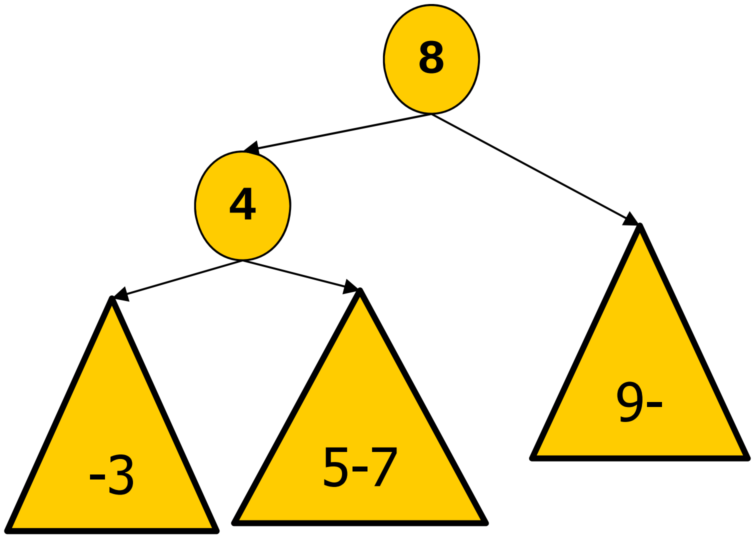


A Closer Look



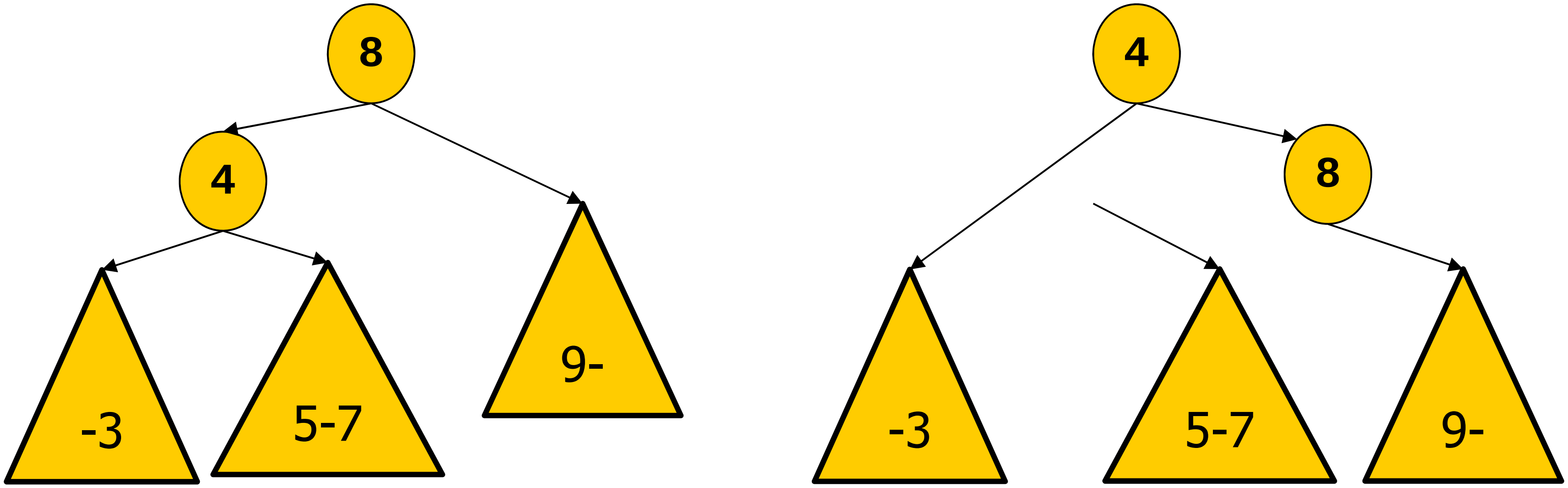
- Subtree 1 contains values smaller than p (and than p')
- Subtree 2 contains values larger than p, but smaller than p'
- Subtree 3 contains values larger than p' (and than p)
- Can we rearrange the subtree rooted in p' such that SC and HC hold?

Example



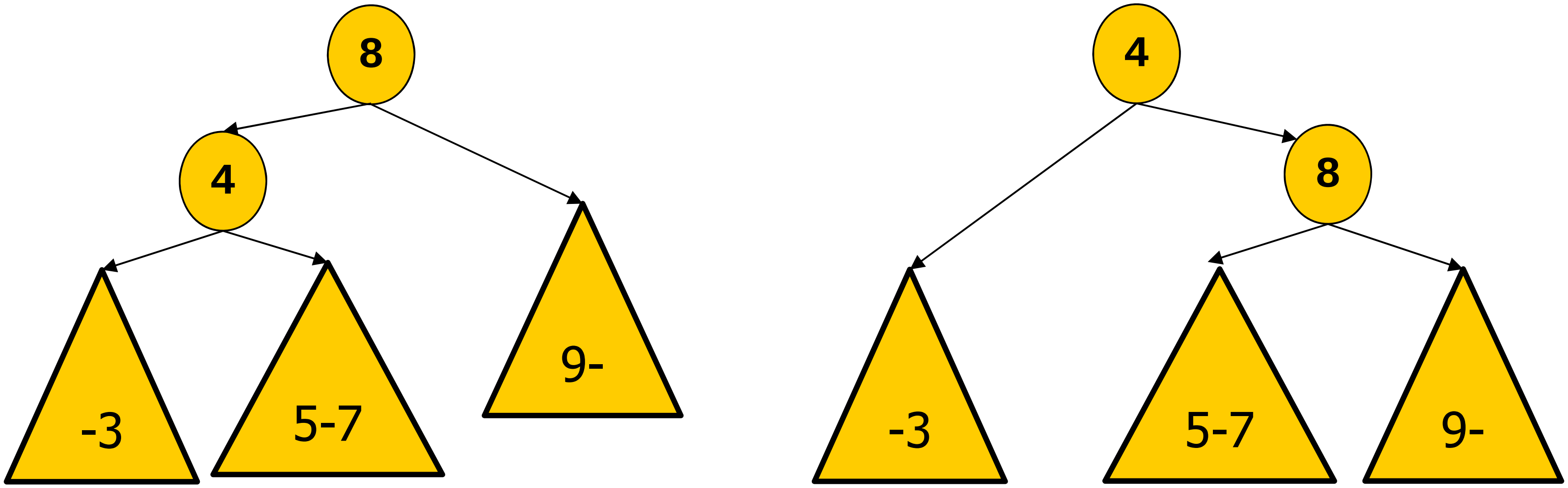
- Subtree 1 contains values smaller than p (and than p’)
- Subtree 2 contains values larger than p, but smaller than p’
- Subtree 3 contains values larger than p’ (and than p)
- We change the root node

Rotation



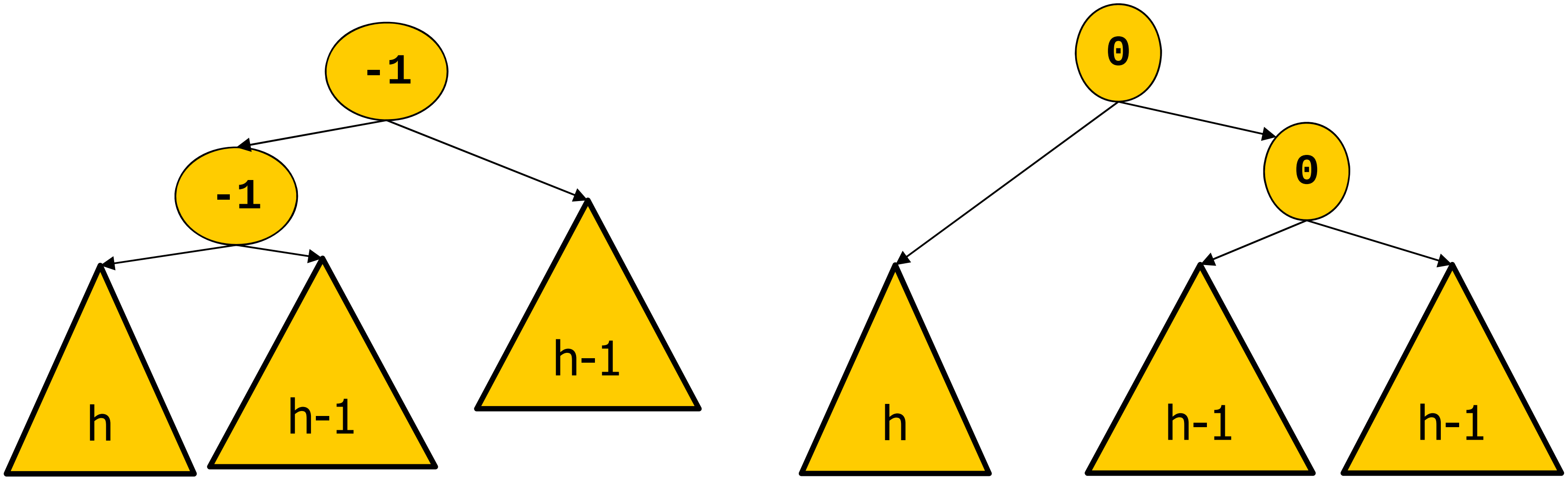
- Rotate nodes p and p’ to the right

Rotation



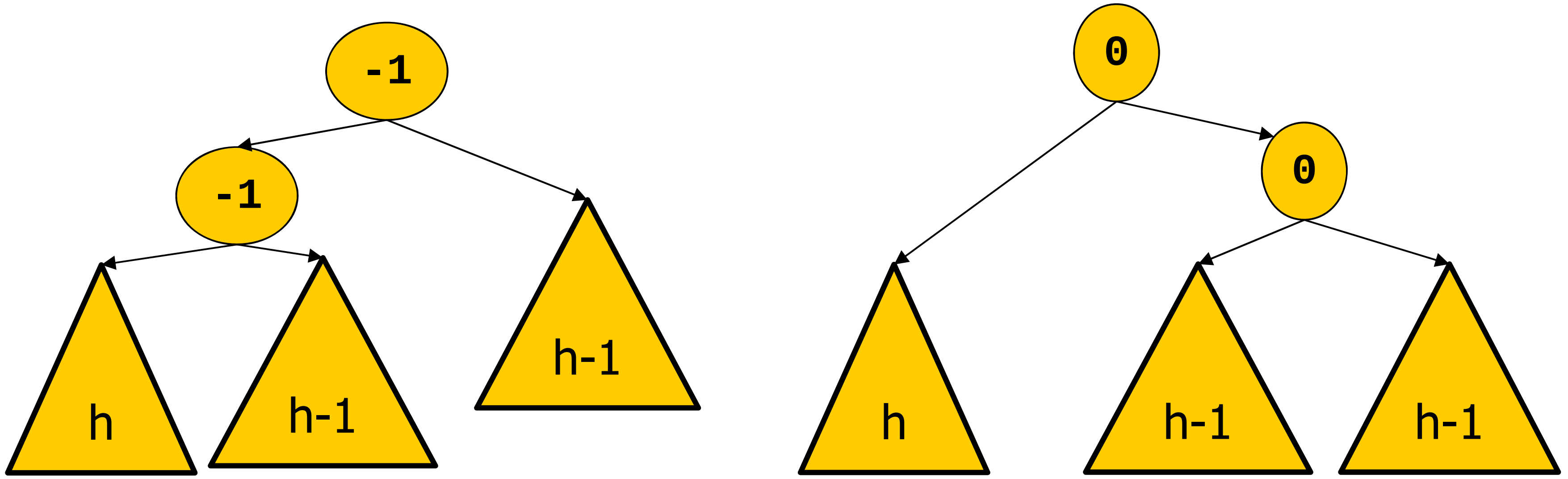
- Rotate nodes p and p’ to the right
- Clearly, SC holds
- Impact on HC?

Rotation and HC



- Before rotation
  - HC hurt in left subtree (height is h+2) versus right subtree (height is h+1)
  - Subtree before insertion had height h+1

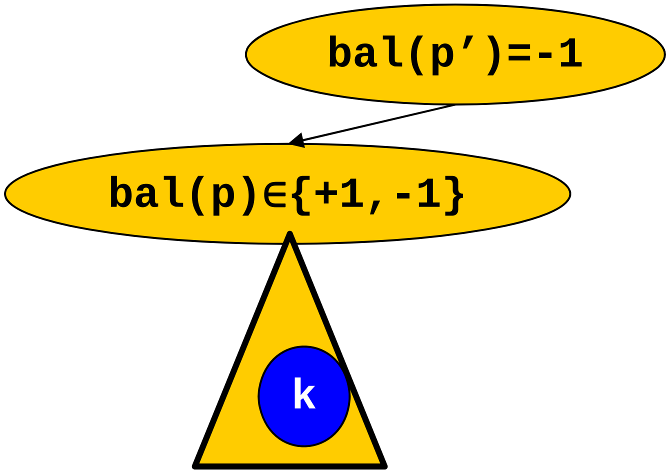
Rotation and HC



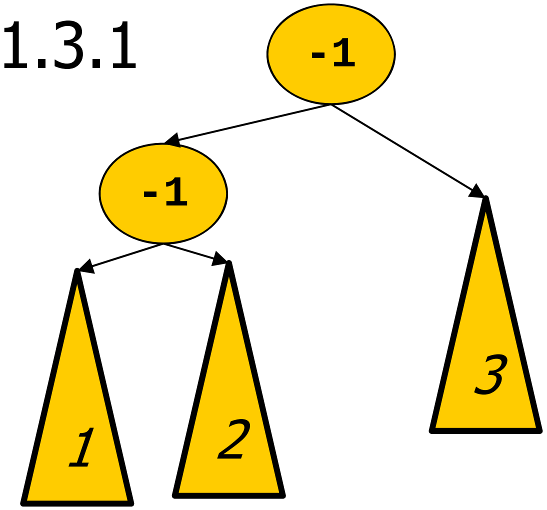
- Before rotation
  - HC hurt in left subtree (height is h+2) versus right subtree (height is h+1)
  - Subtree had height h+1
- After rotation
  - HC holds
  - Height of subtree unchanged
  - No further upin()

Recall …

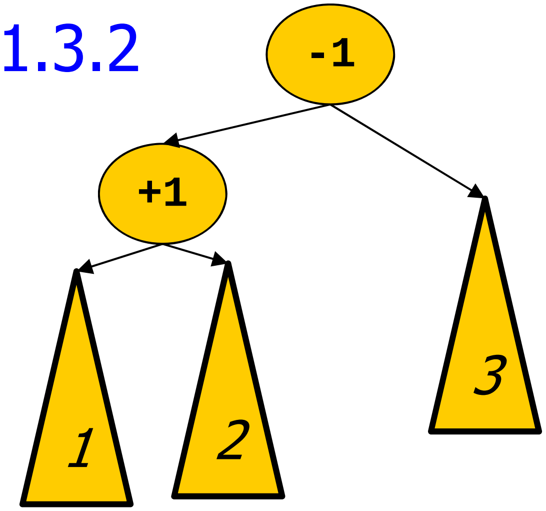
- Case 3.1.3
  - Left subtree of p` was already higher than right subtree
  - And has even grown
  - HC is hurt in p’
  - Fix locally
  - How?



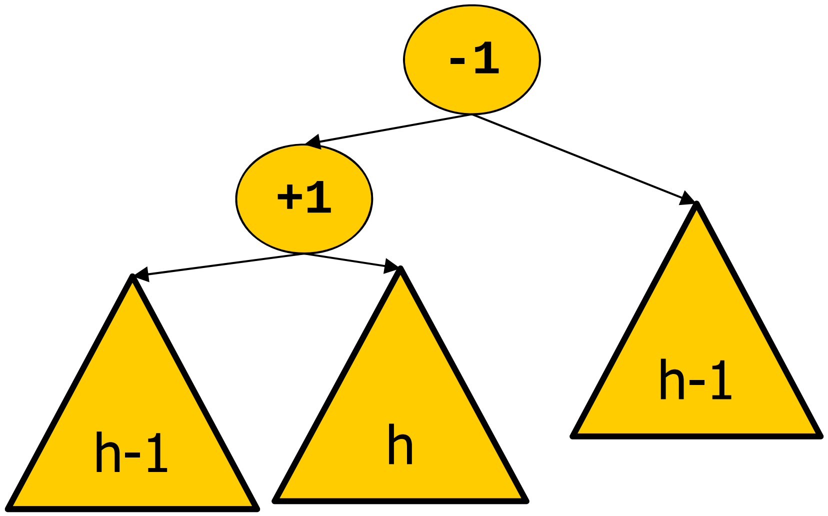
- Case 3.1.3.1



Case 3.1.3.2

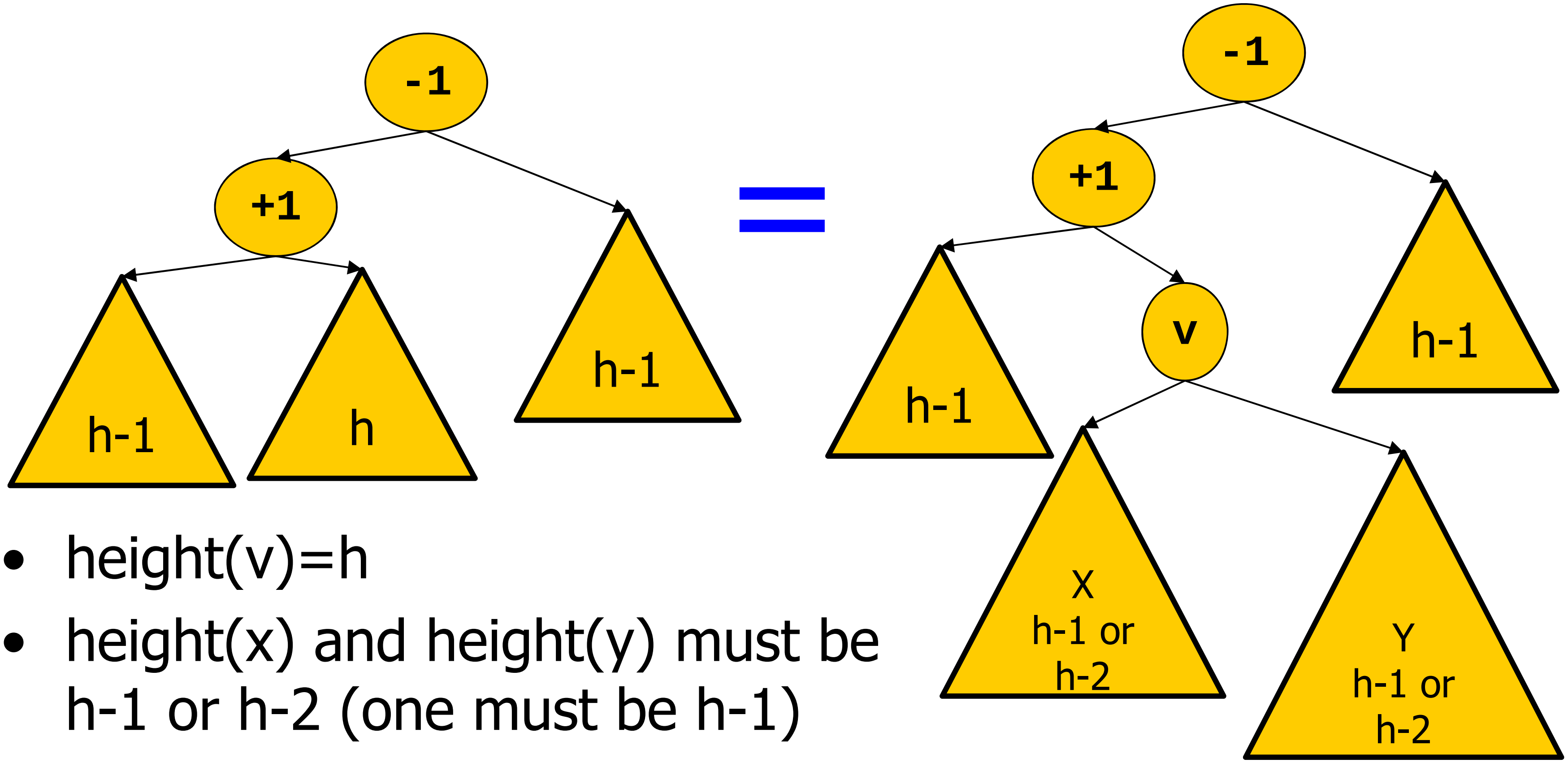


More Intricate



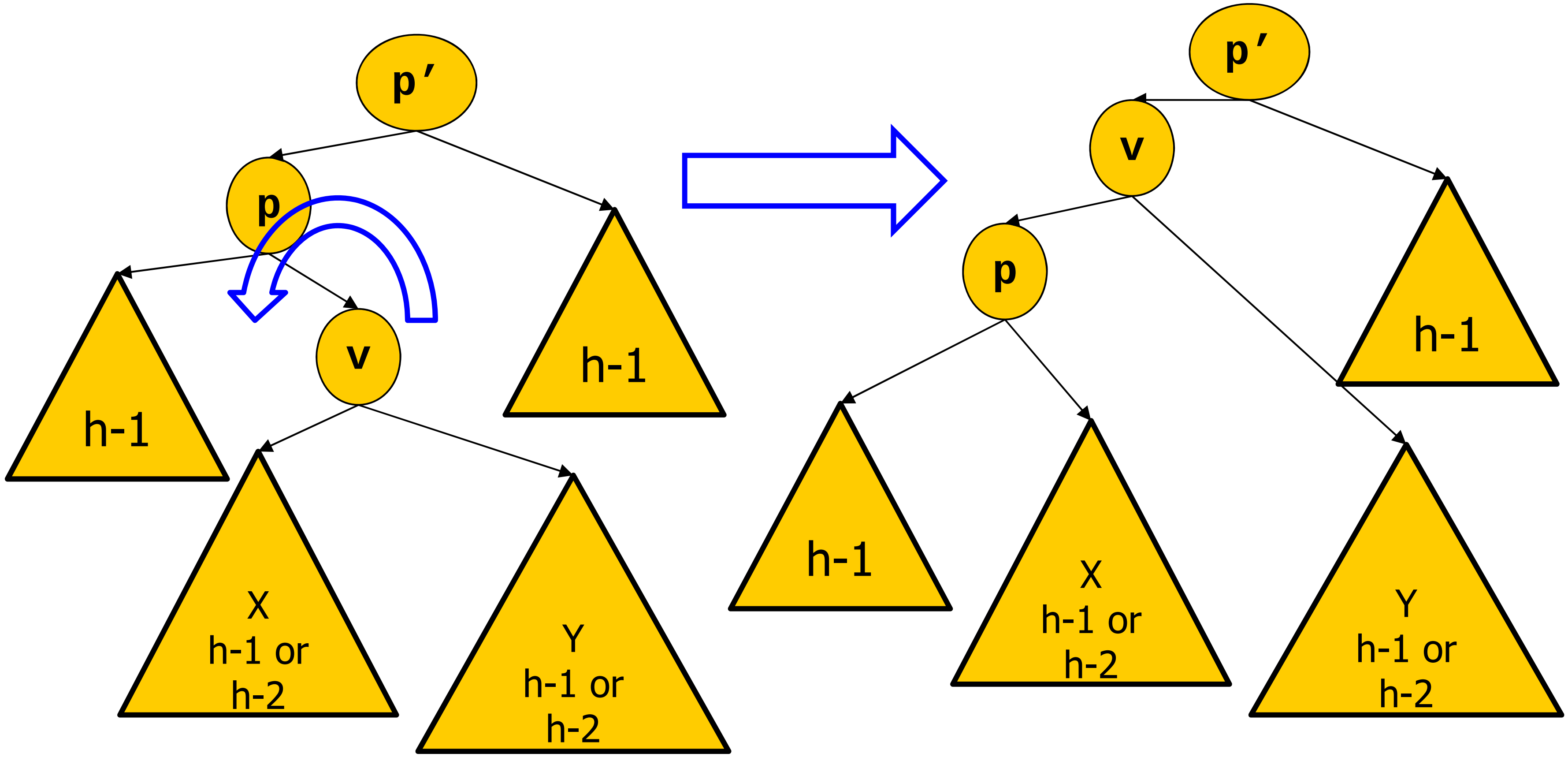
- HC hurt (heights h+2 versus h)
- If we rotated to the right, p (the new root) would have a left subtree of height h-1 and a right subtree of height h+1
- Forbidden by HC
- We have to “break” the subtree of height h

One More Level of Detail

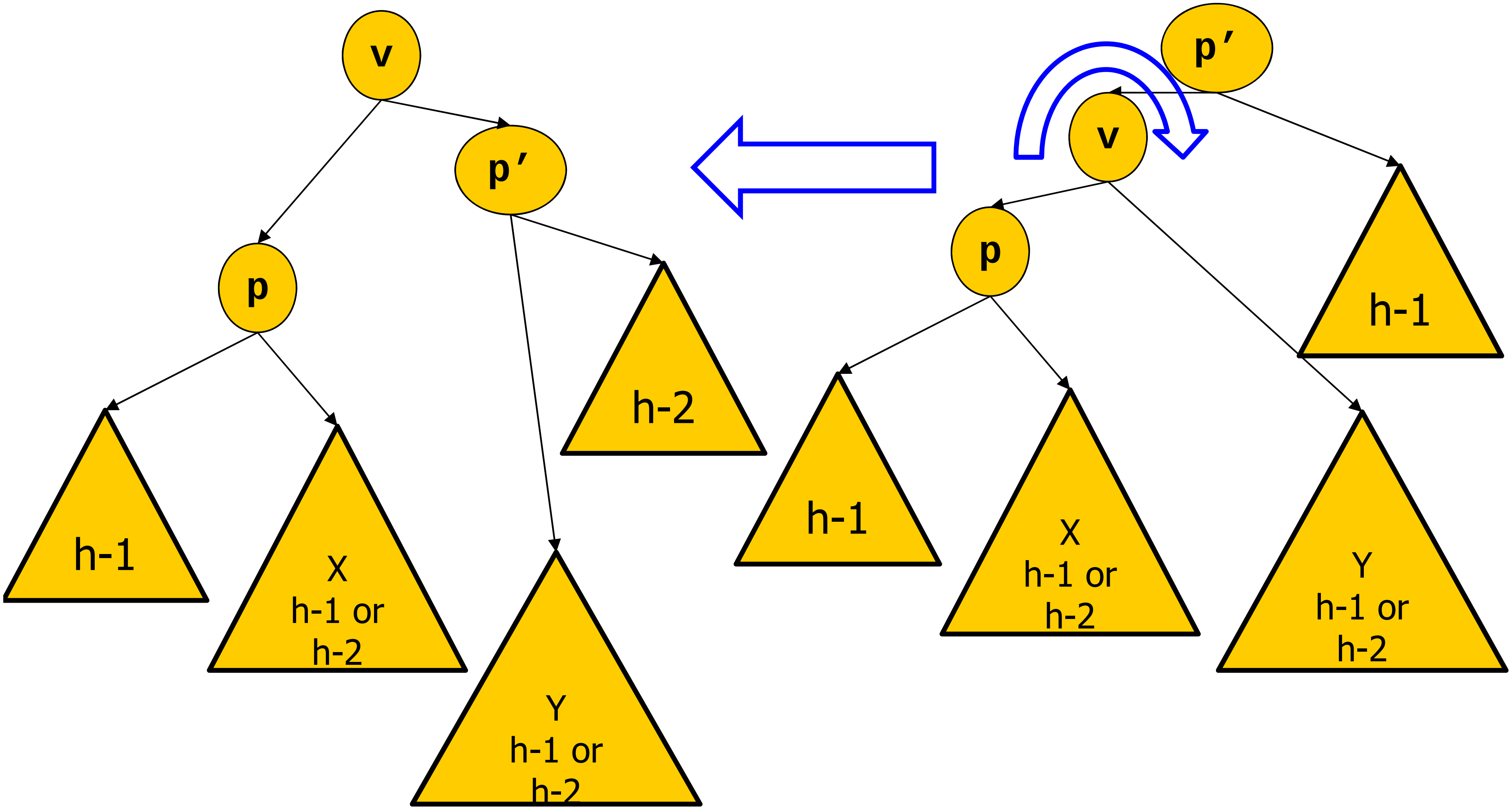


- height(v)=h
- height(x) and height(y) must be h-1 or h-2 (one must be h-1)
- Since the subtree rooted at p has just grown in height, this growth must have happened below v (because bal(p)=+1), so we must have height(x)≠height(y)

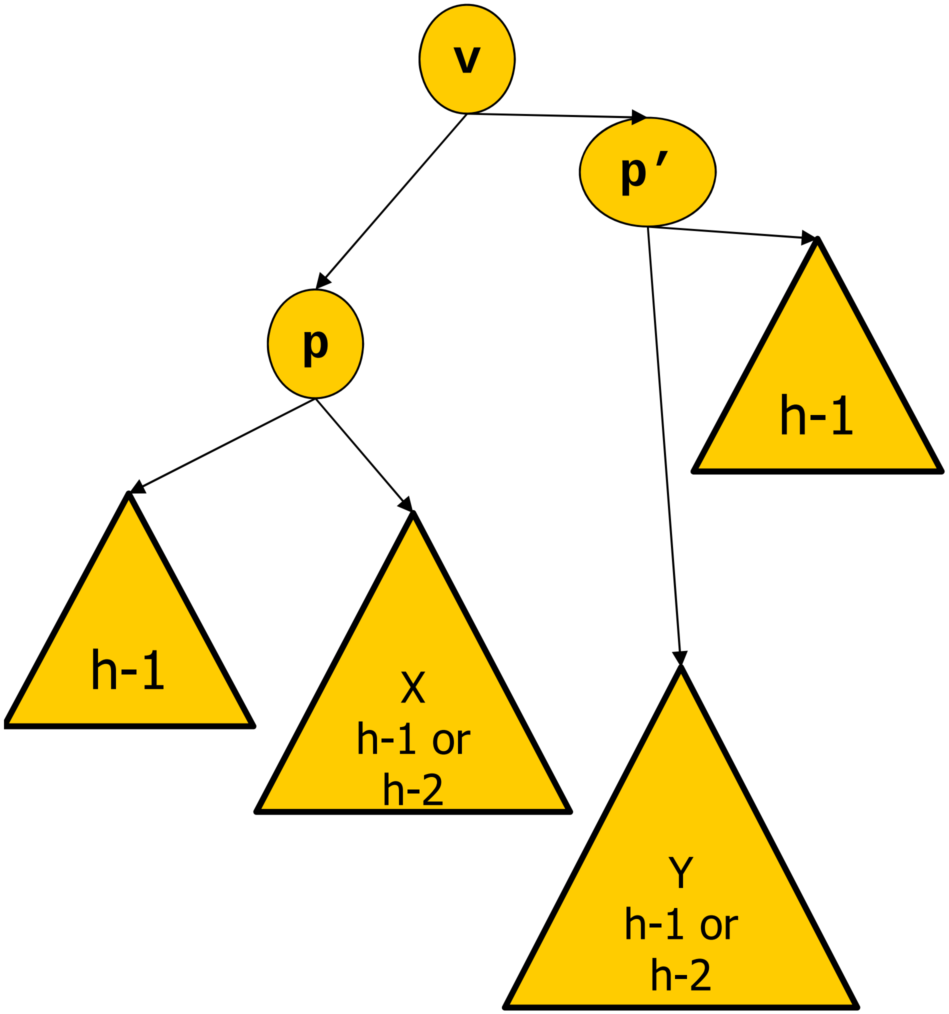
Double Rotation



Double Rotation

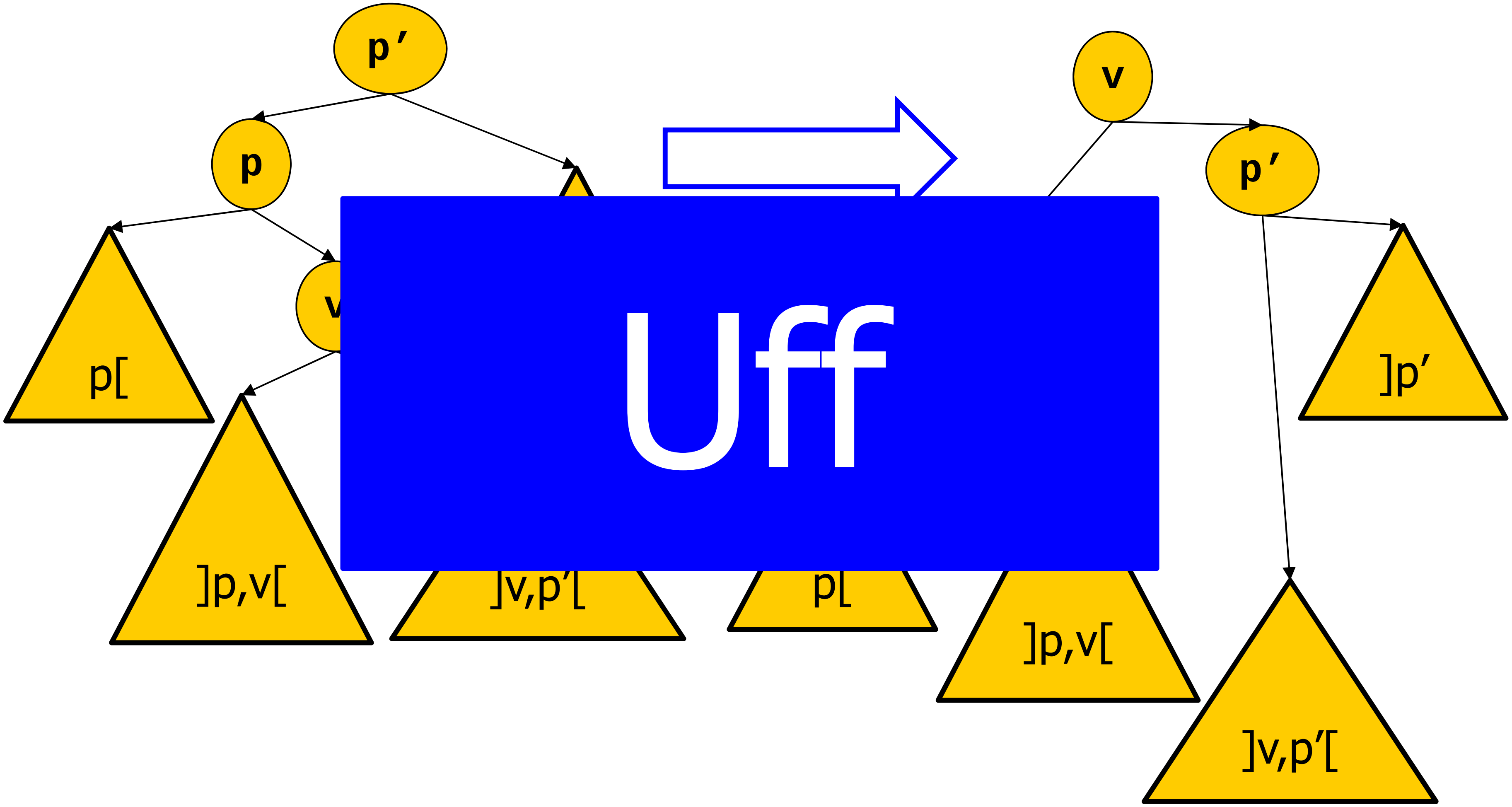


AVL Constraints



- Adaptation
  - bal(p) ∈ {0, -1}
  - bal(p') ∈ {0, +1}
  - bal(v) = 0
- 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?

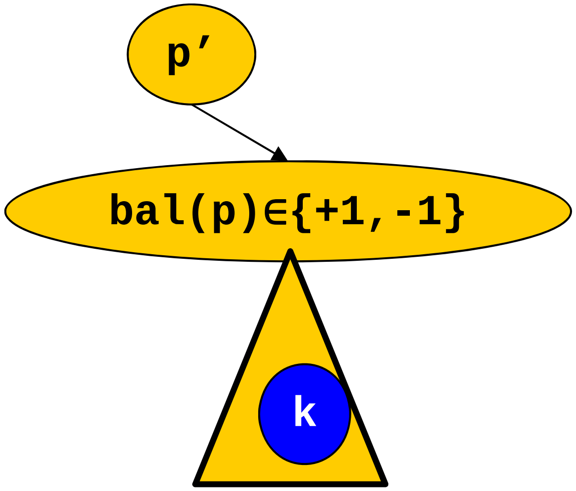
Search Constraint



Uff

Are we Done?

- Case 3.2

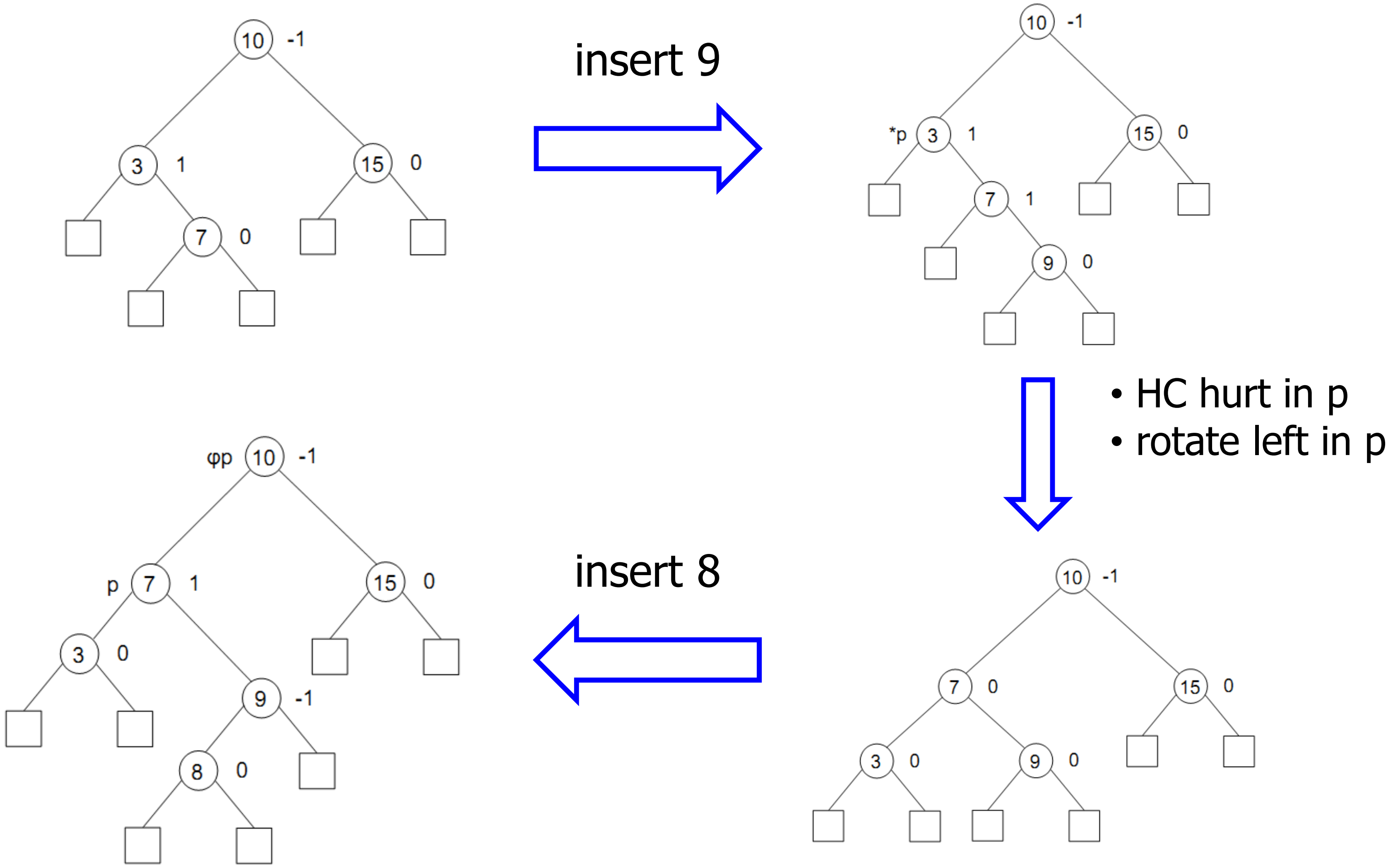


- Similar solution
  - If bal(p’)=-1, adapt and finish
  - If bal(p’)=0, adapt and call upin(parent(p’)
  - If bal(p’)=+1, then
    - Case 3.2.3.1: Rotate left in p
    - Case 3.2.3.1: Rotate right in p, then rotate left in v

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

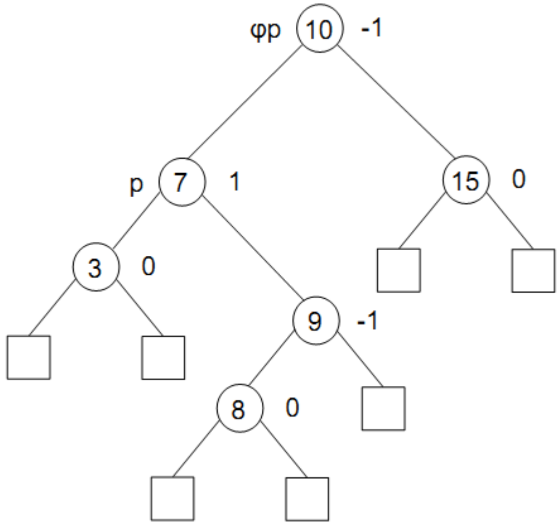


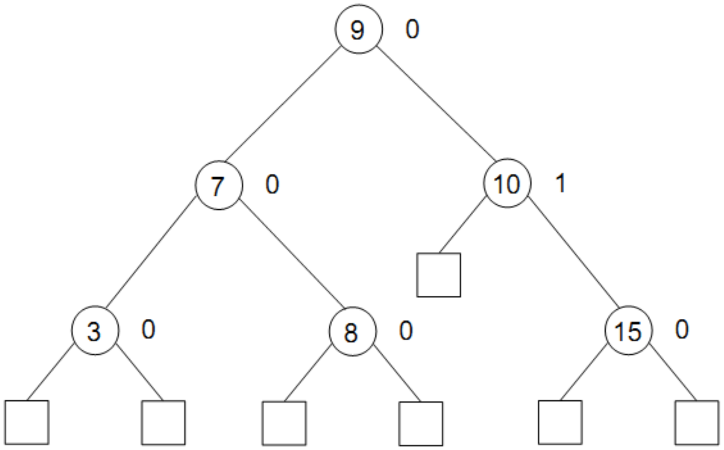
insert 9

- 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

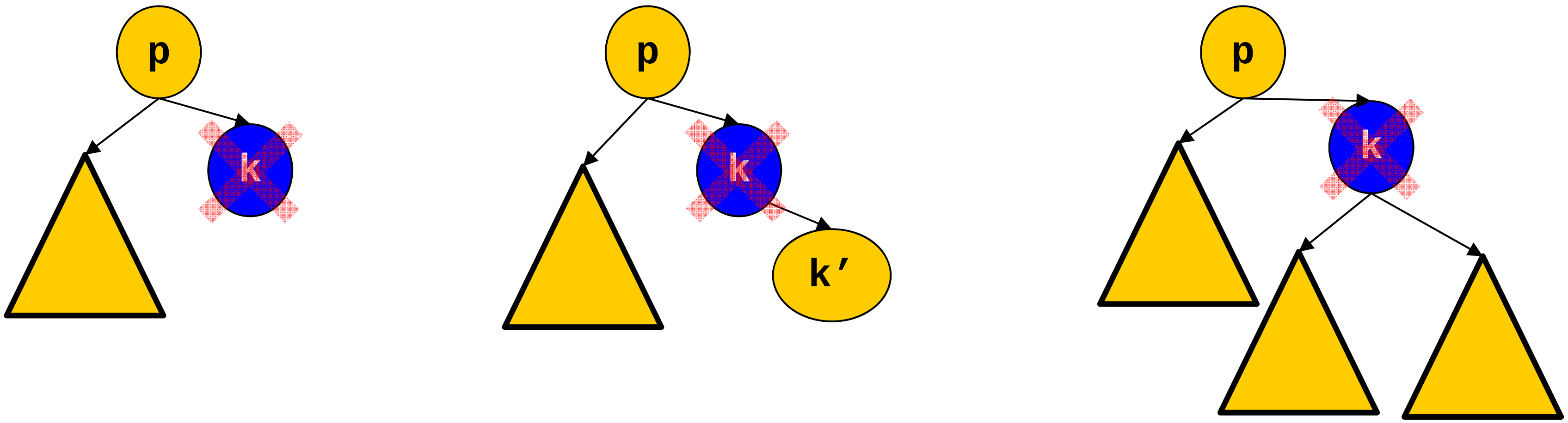
- AVL Trees
- Searching
- Inserting
- Deleting

Deleting a Key

- Follows the same scheme as insertions
  - 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 or double rotate, and cases where we have to propagate changes up the tree

Major Cases

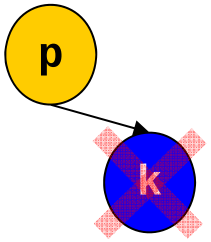
- Case 1: k has no children
- Case 2: k has only one child
- Case 3: k has two children

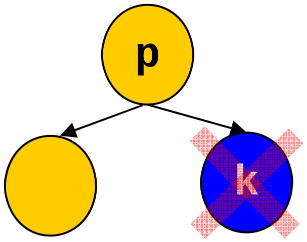


Case 1: k has no children

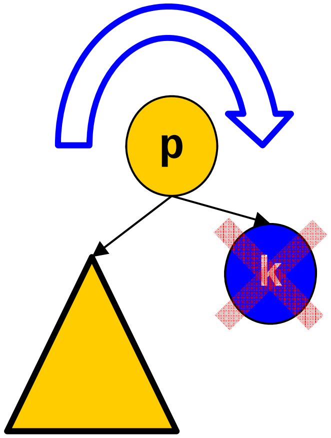
The other subtree rooted at p …

- Case 1.1: … is empty
  - Remove k, adapt bal(p)
  - call upout(p)
    - Because height of subtree rooted at p has changed



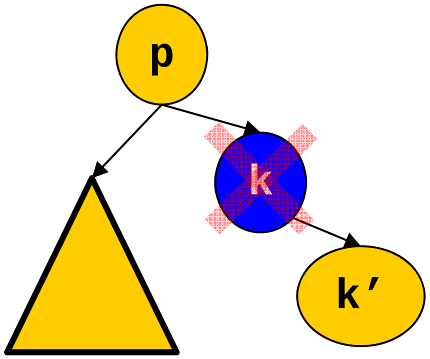


- Case 1.2: … has exactly one key
  - Remove k, adapt bal(p)
  - Done
- Case 1.3: … has two or three keys
  - Remove k, adapt bal(p)
  - Rotate right in p
  - call upout(p)



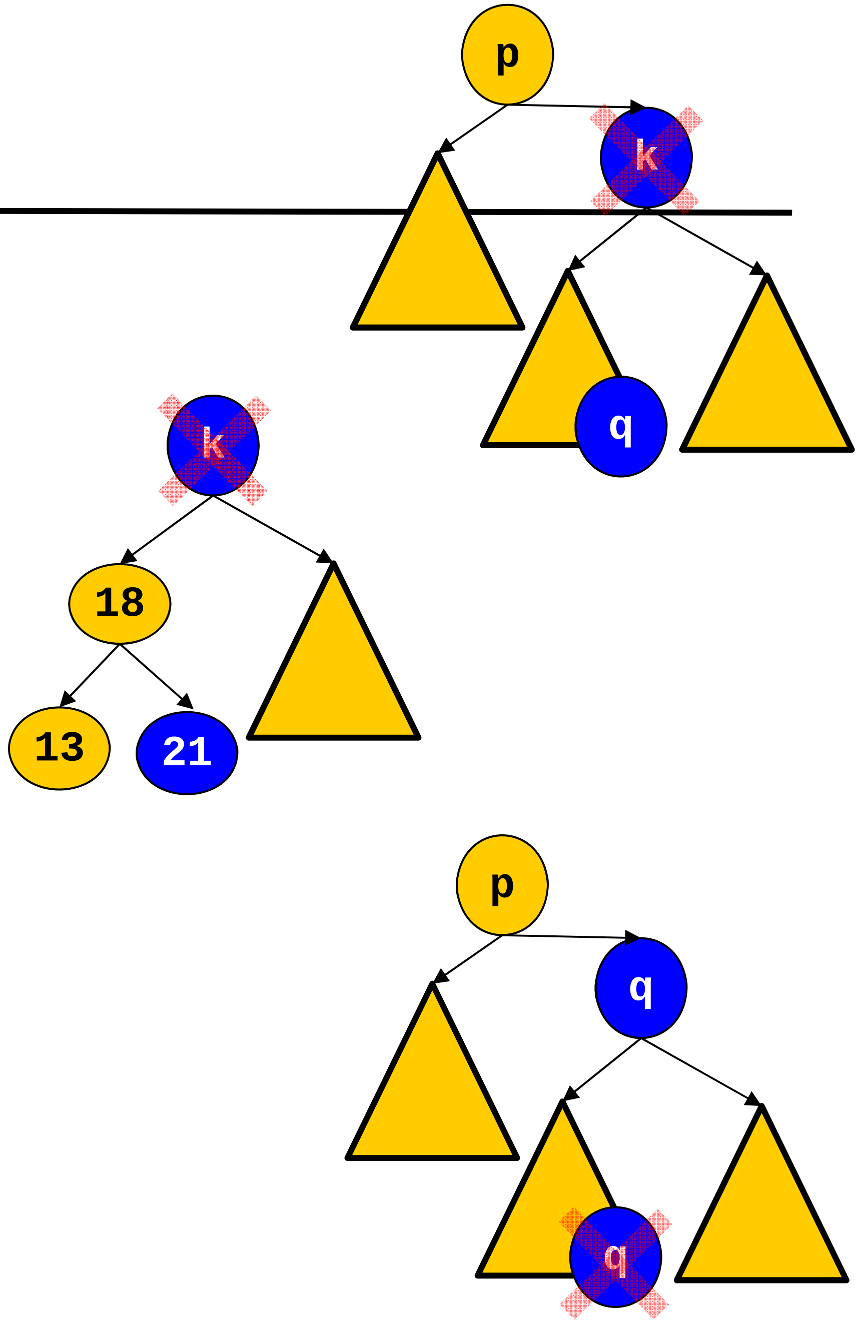
Case 2: k has only one child

- Replace k with k`
- k` cannot have children, or HC would not hold in k
- Height and balance of k (now k’) has changed
- Update bal(p) and call upout(p)



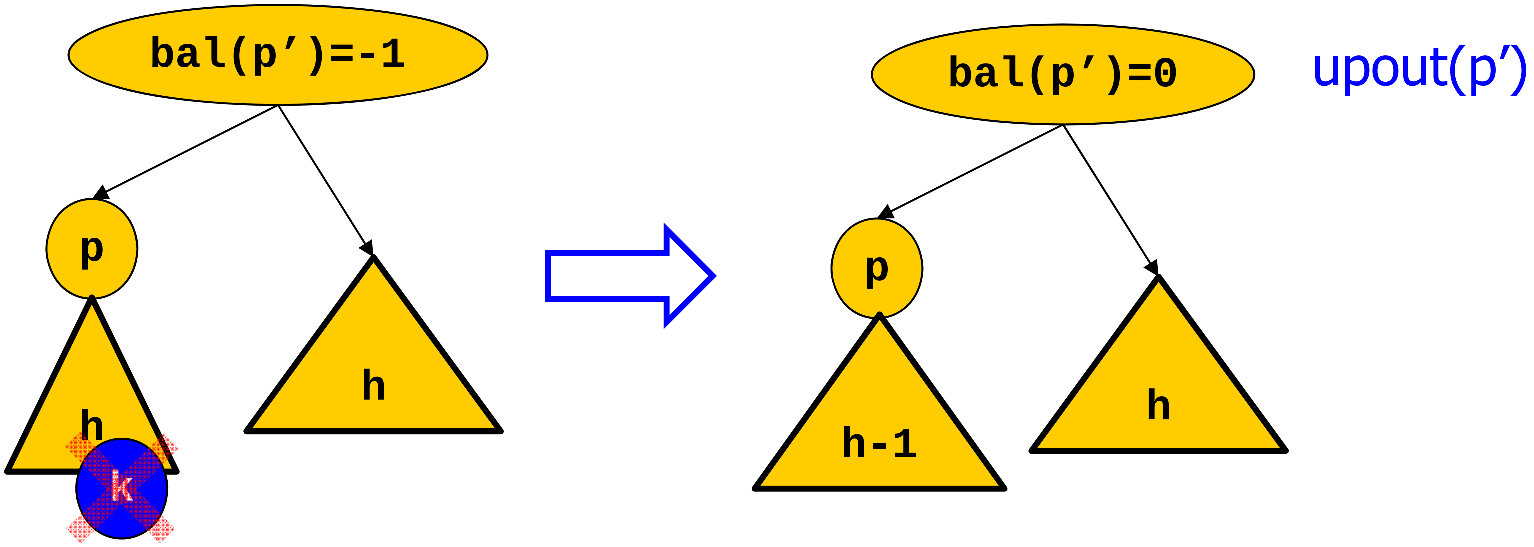
Case 3: k has two children

- Recall natural search trees
- We search the symmetric predecessor q of k
  - Which is the largest value in the left subtree of k
- Replace k with q and remove the old q by calling delete(q) as discussed in Case 1 and Case 2
  - Note that the old q has either no child (Case 1) or exactly one child (Case 2)



Procedure upout(p)

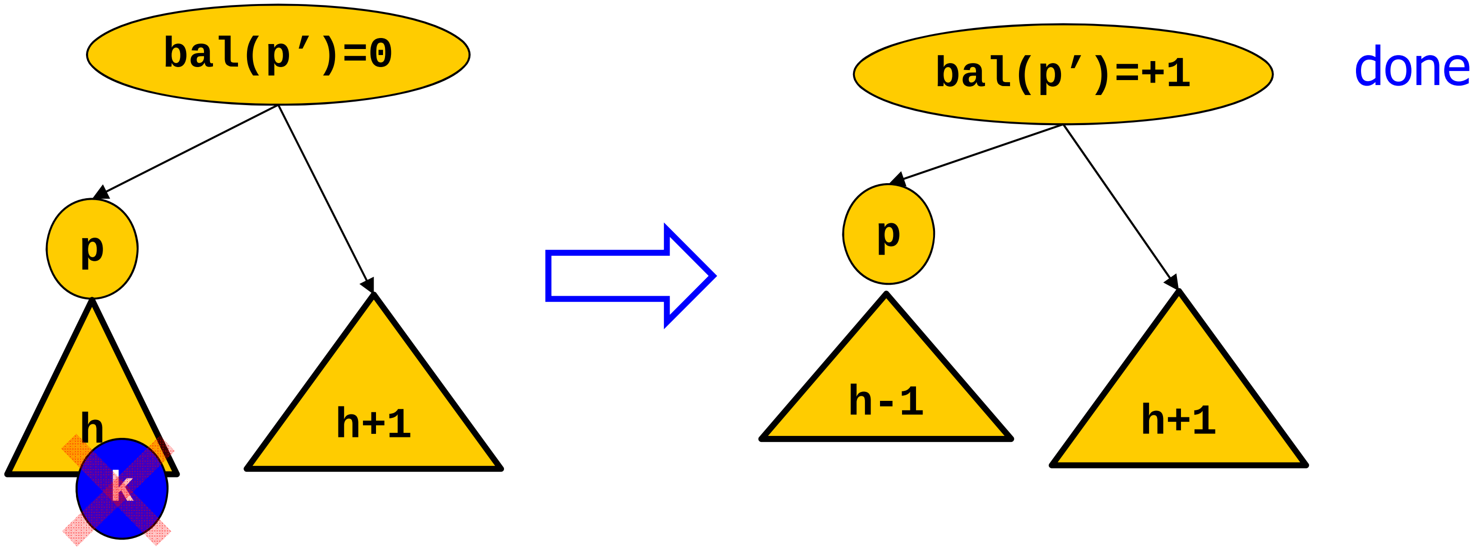
- Invariant: Whenever we call upout(p), the height of p has decreased by 1 and bal(p)=0
- Let p be the left child of its parent p’
  - Again, the case of p being the right child of p’ is symmetric
- Case 1; bal(p’)=-1



upout(p’)

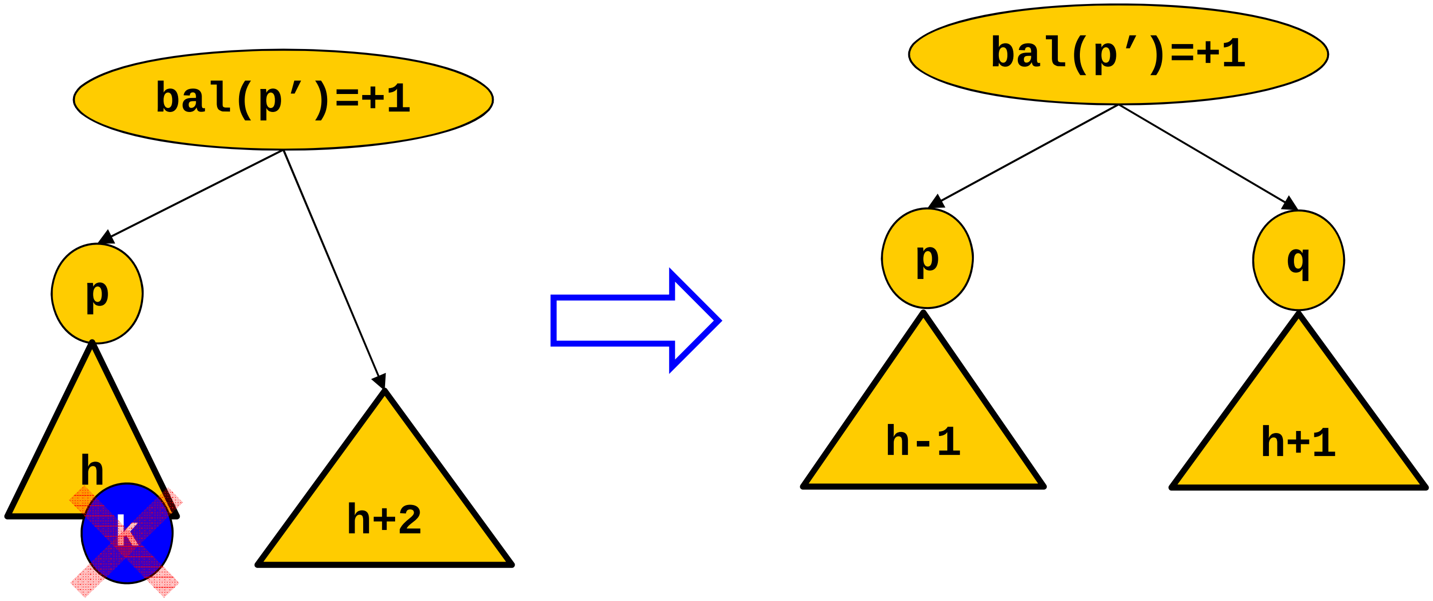
Procedure upout(p)

- Whenever we call upout(p), 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



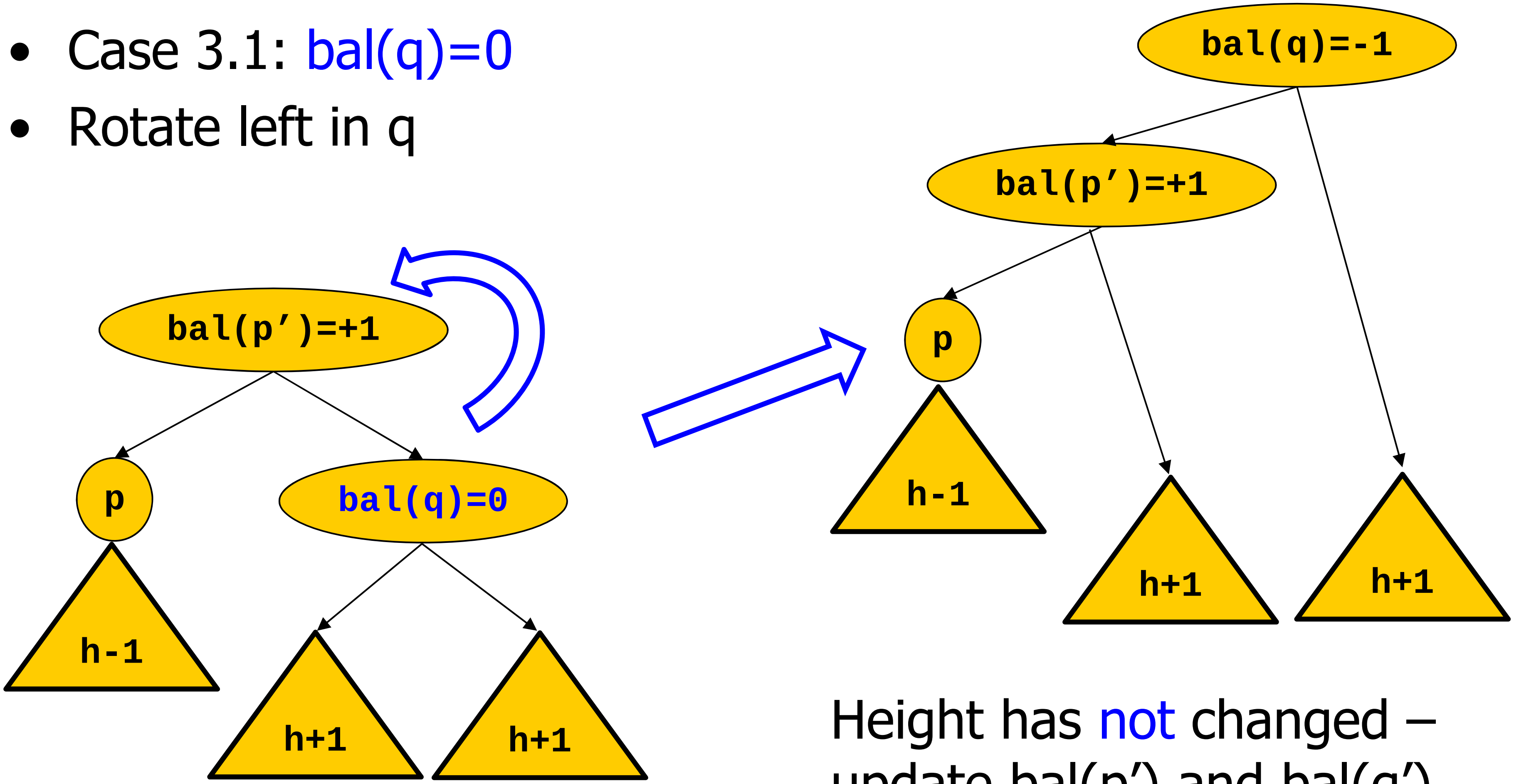
Procedure upout(p)

- Whenever we call upout(p), 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

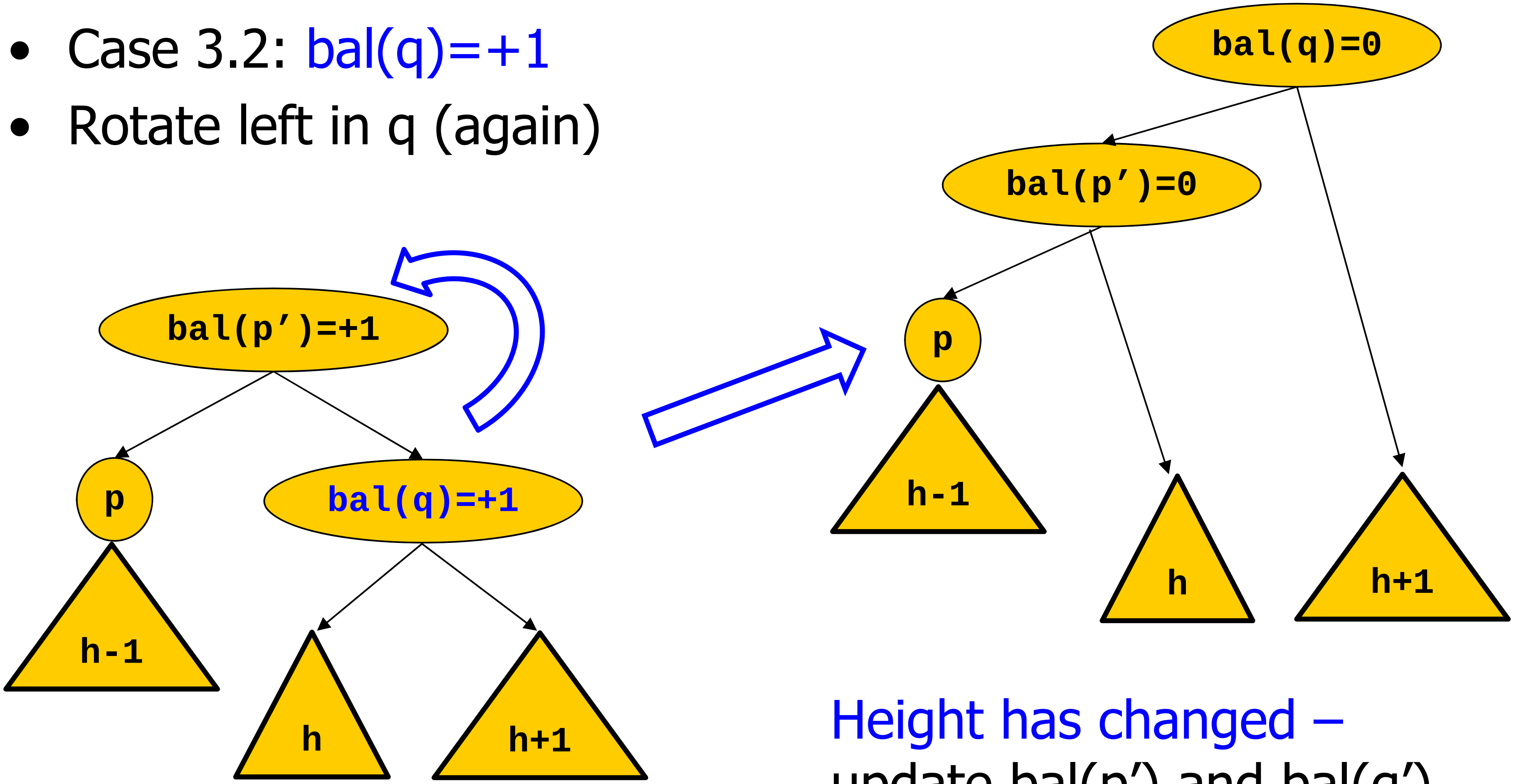
- Case 3.1: bal(q)=0
- Rotate left in q



Height has not changed – update bal(p') and bal(q') and done

Subcase 2

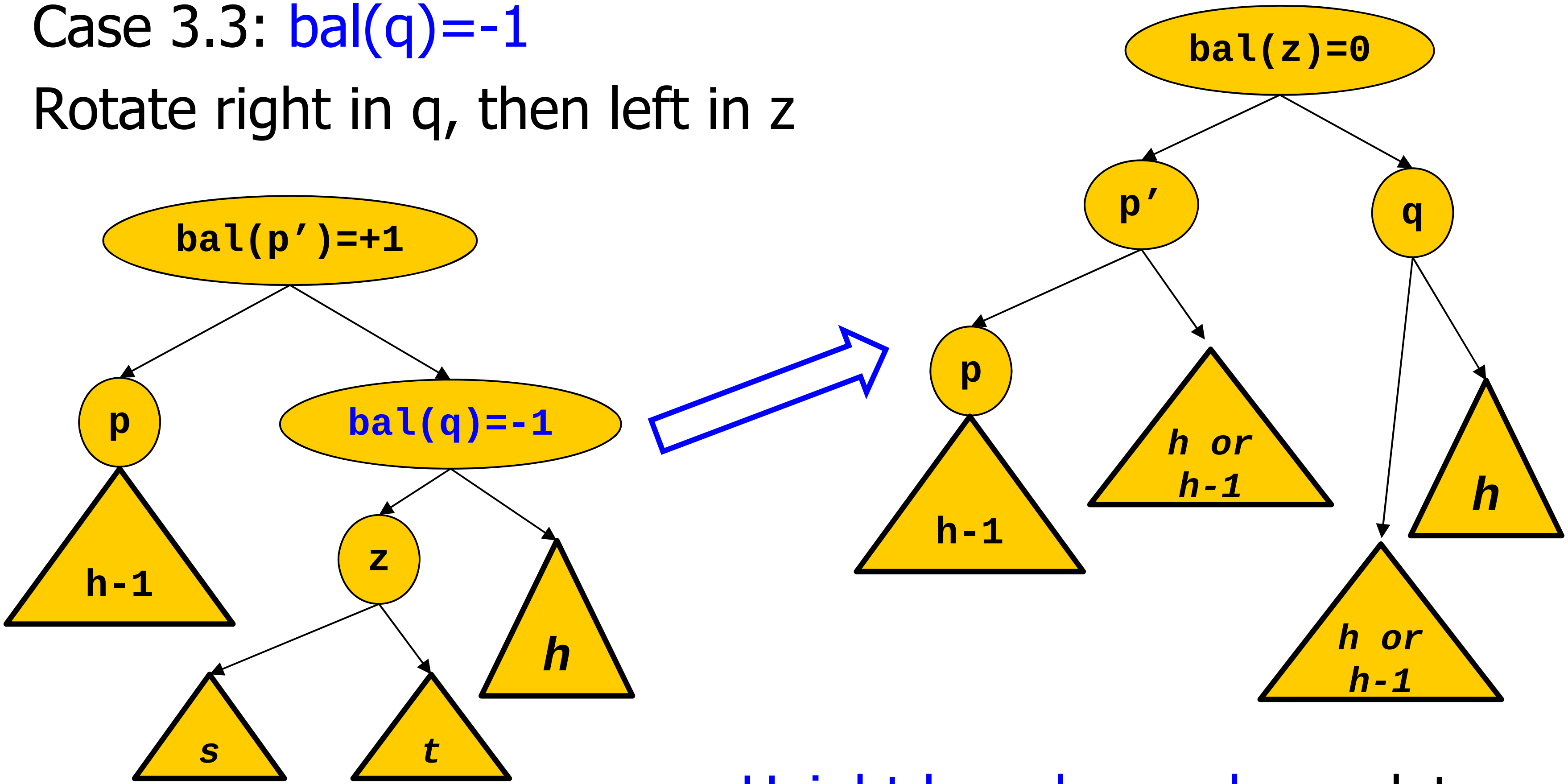
- Case 3.2: bal(q)=+1
- Rotate left in q (again)



Height has changed – update bal(p’) and bal(q’) and call upout(q)

Subcase 3

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



Either s or t has height h
and the other one h or h-1

Height has changed – update balance factors & call 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 show that ins/del are in O(1) on average
  - Because reorganizations are rare and usually stop very early
- AVL trees are a "work-horse" for keeping a sorted list
- 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