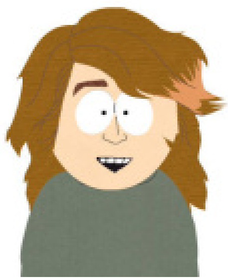
Balanced BST and AVL Trees



Carlos Moreno
cmoreno@uwaterloo.ca
EIT-4103

https://ece.uwaterloo.ca/~cmoreno/ece250

These slides, the course material, and course web site are based on work by Douglas W. Harder

Balanced BST and AVL Trees

Standard reminder to set phones to silent/vibrate mode, please!

Balanced BST and AVL Trees

- Previously, on ECE-250...
  - We talked about Binary Search Trees (BST) and their applicability to searches.
  - We also wanted to use them to maintain a list of ordered values (as in, adding and removing elements).
    - And of course, greedy as we are, never happy enough with what we have, we *also* wanted to do all of this efficiently!!
  - We discussed that this required the height of the tree to be $\Theta(\log n)$, which requires a *balanced* tree.

Balanced BST and AVL Trees

- And now...
  - We'll expand on this issue of Balanced Binary Search Trees.
    - We'll look into two “obvious” types of balance (weight and height).
  - We'll discuss AVL trees — one of the commonly used techniques to efficiently maintain a balanced BST.
    - We'll look into the techniques to perform all the operations and techniques to re-balance the tree.

Balanced BST and AVL Trees

- Recall search on a binary search tree:
  - Data is not hierarchical, but rather linearly or totally ordered.
  - Values in the left sub-tree are all less than the value at the root, and values in the right sub-tree are all greater than the value at the root (we'll continue to assume that we have no duplicate values).
  - This allows us to do the analogous to a binary search — if the value we're searching is less than the current node, we continue searching through the left sub-tree.

Balanced BST and AVL Trees

- Recall search on a binary search tree:
  - The idea being: if each sub-tree always has half as many elements, then we're doing essentially the same as in binary search — with each comparison, we discard half the remaining values.
    - This leads to logarithmic time in the search.

Balanced BST and AVL Trees

- Recall search on a binary search tree:
  - An alternative way to look at this is: the search takes worst-case $\Theta(h)$, where h is the height of the tree
    - This tells us that a tree where the left and right sub-trees of every node have the same number of nodes has logarithmic height with respect to the number of nodes!

Balanced BST and AVL Trees

- We notice that a tree where we insert and remove elements can not have strict balance as described so far:
  - Our definition has to accommodate a ±1 difference (*why?*)

Balanced BST and AVL Trees

- We notice that a tree where we insert and remove elements can not have strict balance as described so far:
  - Our definition has to accommodate a ±1 difference (*why?*)
  - As a simple counter-example: suppose we have a perfect tree, and add one element. How could both sub-trees have the exact same number of elements?

Balanced BST and AVL Trees

- A different type of balance, though leading to the same outcome (logarithmic height with respect to the number of nodes) is height balance:
  - For every node in the tree, the left and right sub-trees have the same height, ±1

Balanced BST and AVL Trees

- We'll now look into AVL trees (named after its two inventors, Adelson-Velskii and Landis), a type of height-balanced binary search trees.

Balanced BST and AVL Trees

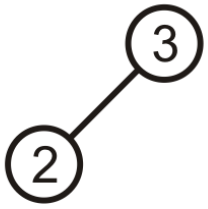
- We'll now look into AVL trees (named after its two inventors, Adelson-Velskii and Landis), a type of height-balanced binary search trees.
- A binary search tree is an AVL tree if:
  - The difference in heights between left and right sub-trees is at most 1, and
  - Both sub-trees are AVL trees

Balanced BST and AVL Trees

- The principle of operation is remarkably simple; let's look at this "prototypical" trick to maintain balance:
  - Let's add 3, 2, 1 (in that order) to a binary search tree:

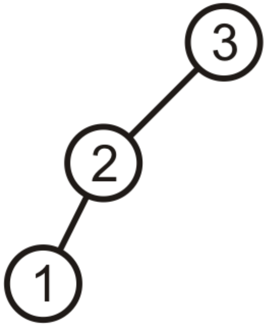
Balanced BST and AVL Trees

- The principle of operation is remarkably simple; let's look at this “prototypical” trick to maintain balance:
  - Let's add 3, 2, 1 (in that order) to a binary search tree:



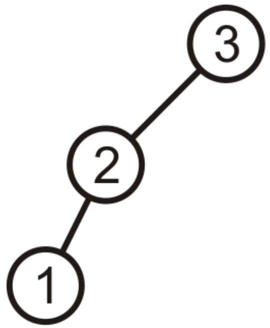
Balanced BST and AVL Trees

- Inserting 1 causes the tree to become imbalanced at node 3 (left sub-tree has height 1, and right sub-tree has height ... ?? you tell me?)



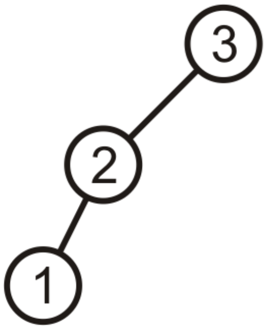
Balanced BST and AVL Trees

- Inserting 1 causes the tree to become imbalanced at node 3 (left sub-tree has height 1, and right sub-tree has height ... ?? you tell me?) — we recall from the first class on trees, that by convention, an empty tree has height −1



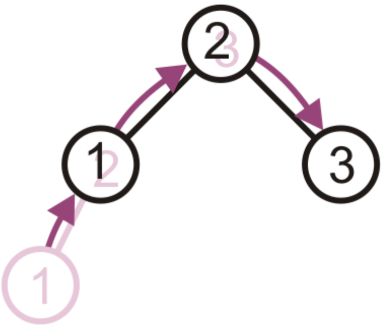
Balanced BST and AVL Trees

- So, we rotate it towards the right:



Balanced BST and AVL Trees

- So, we rotate it towards the right:



Balanced BST and AVL Trees

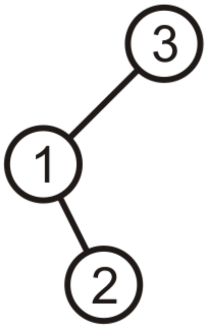
- If we had inserted the sequence 3, 2, 1, the situation would have been essentially the same; the right sub-tree would be the deeper one in that case, so we rotate towards the left in that case.

Balanced BST and AVL Trees

- Inserting the sequence 3, 1, 2, however, does make a significant difference — we end up with the following tree:

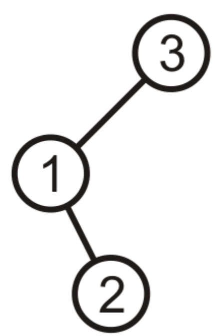
Balanced BST and AVL Trees

- Inserting the sequence 3, 1, 2, however, does make a significant difference — we end up with the following tree:



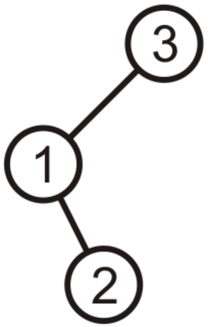
Balanced BST and AVL Trees

- Inserting the sequence 3, 1, 2, however, does make a significant difference — we end up with the following tree:
  - Clearly, we can't rotate to balance (right? *why?*)



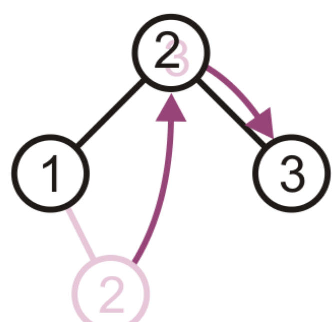
Balanced BST and AVL Trees

- However, we can do a double rotation:
  - First, rotate towards the left the sub-tree 1-2, and then we're in the exact same previous case (so we rotate towards the right)



Balanced BST and AVL Trees

- However, we can do a double rotation:
  - First, rotate towards the left the sub-tree 1-2, and then we're in the exact same previous case (so we rotate towards the right)
  - The “net” effect is the following:

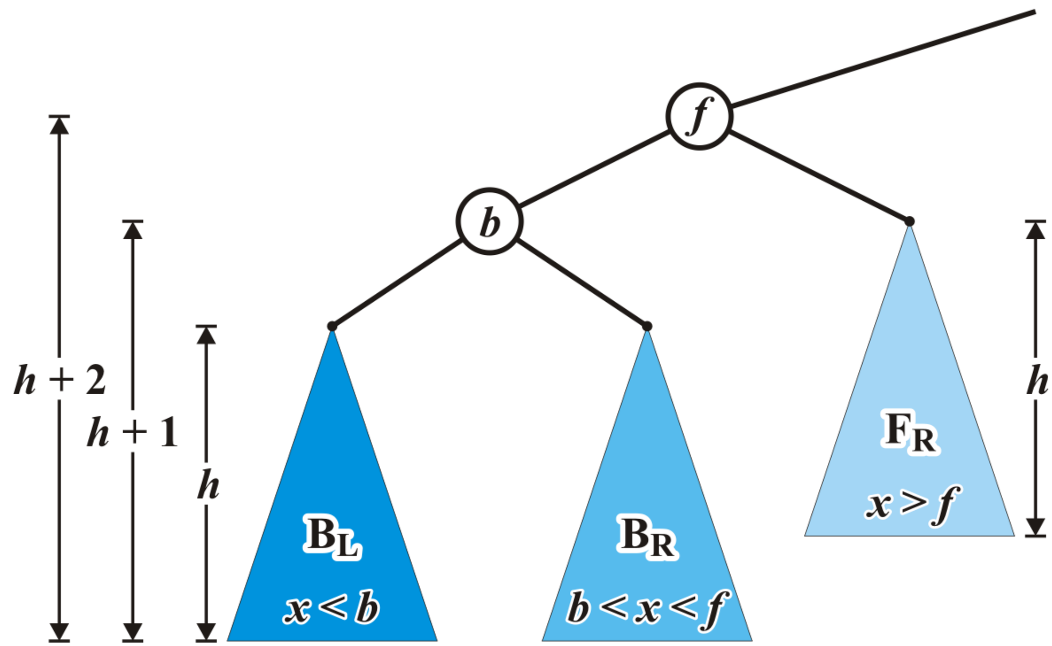


Balanced BST and AVL Trees

- What about a more complicated situation?

Balanced BST and AVL Trees

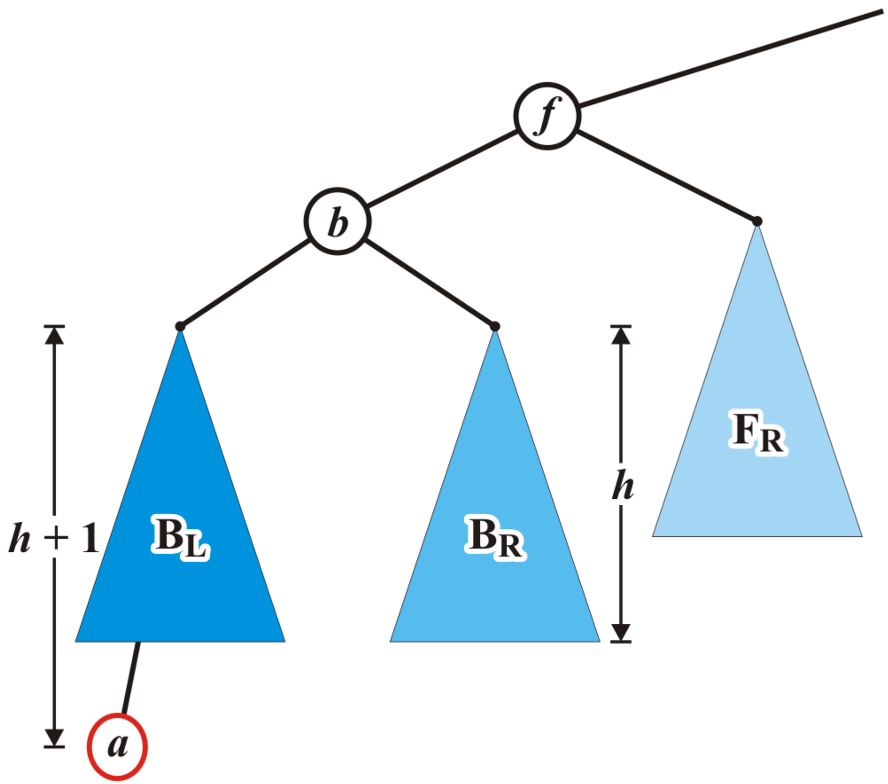
- We insert the value *a*, which makes it into the subtree B_L, without causing any imbalance in it.



(notice that a blue triangle denotes a tree with height *h*)

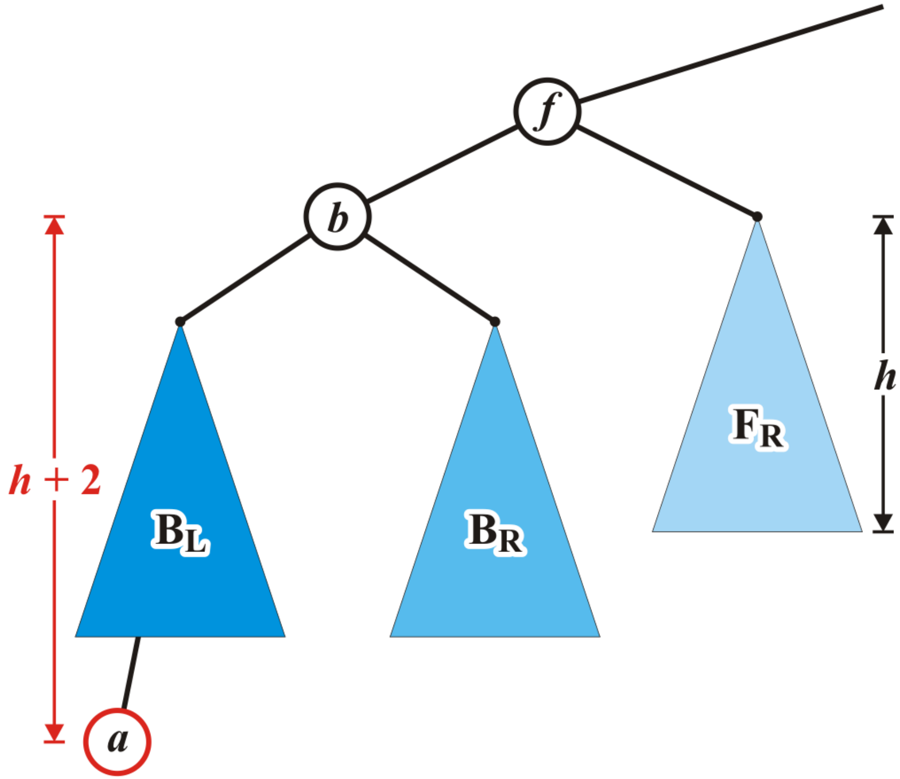
Balanced BST and AVL Trees

- We insert the value *a*, which makes it into the subtree B_L, without causing any imbalance in it.



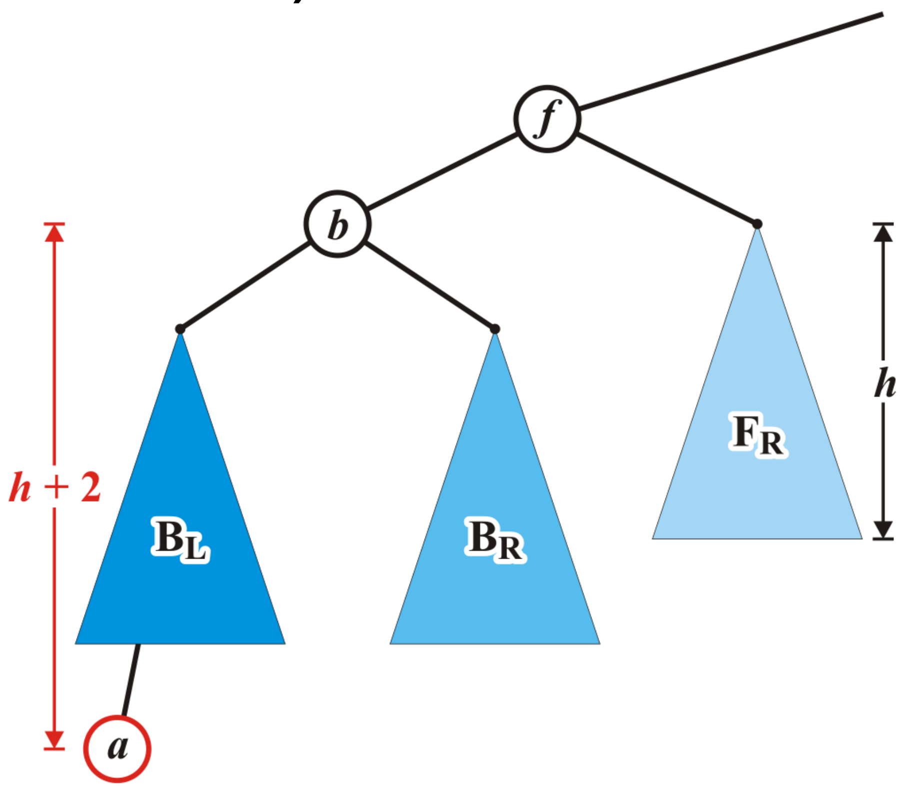
Balanced BST and AVL Trees

- How do we fix the imbalance in *f*? If we try to rotate, what do we do about B_R?



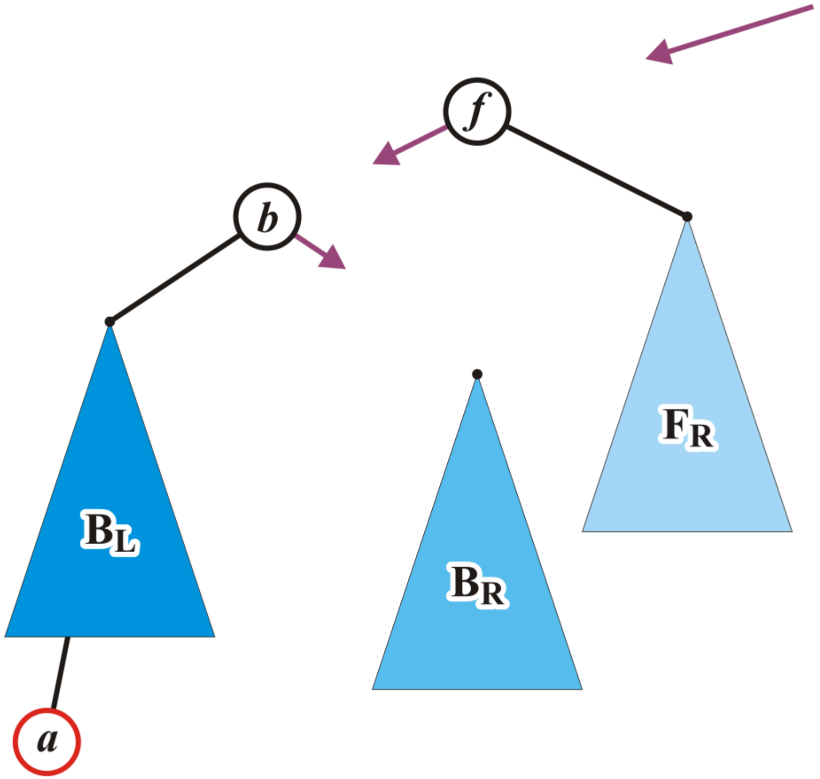
Balanced BST and AVL Trees

- The key detail is that we have to detach B_R from the tree (temporarily — we'll re-attach it after the rotation!)



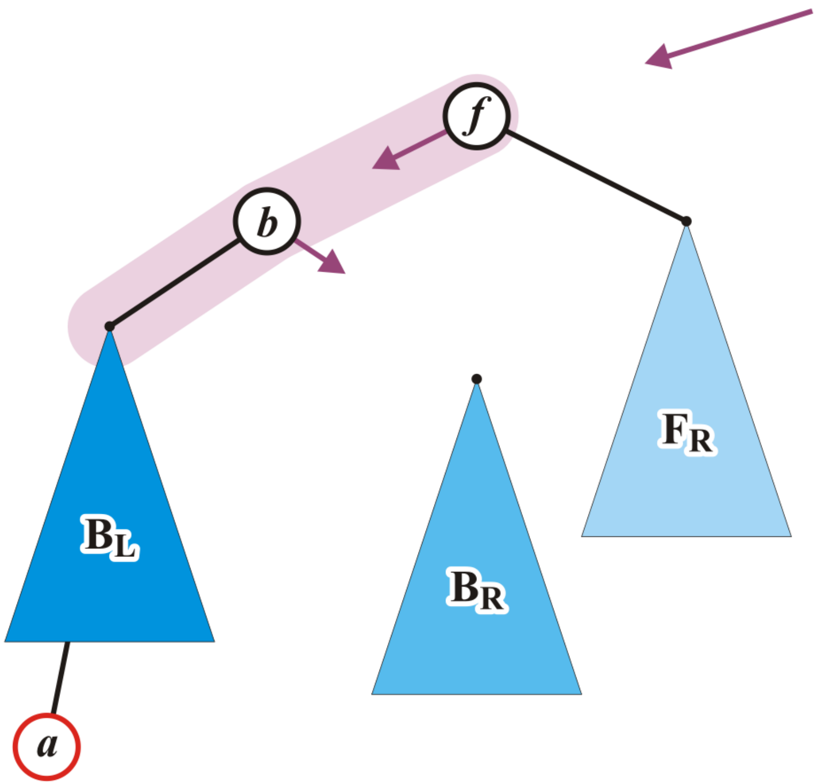
Balanced BST and AVL Trees

- The key detail is that we have to detach B_R from the tree (temporarily — we'll re-attach it after the rotation!)



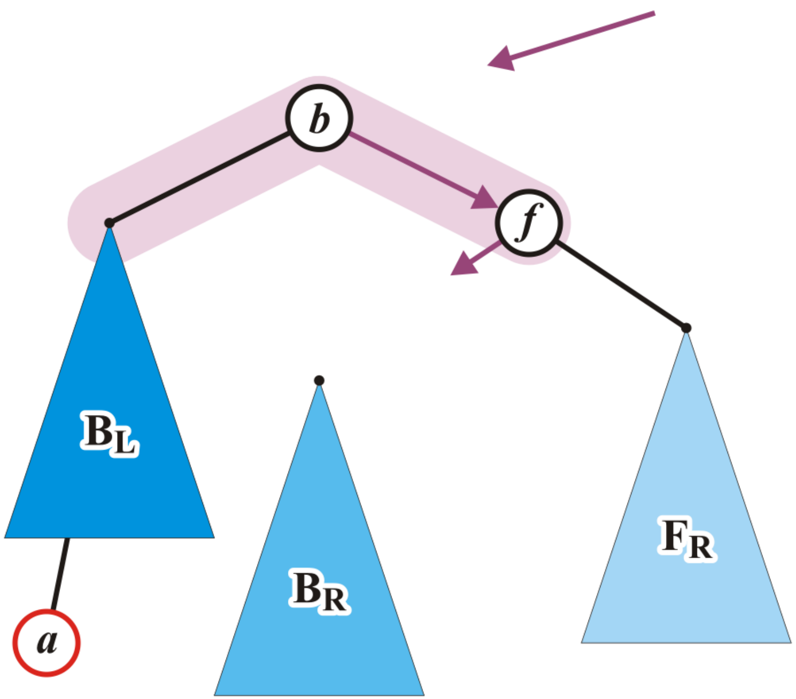
Balanced BST and AVL Trees

- We now see that *f* – *b* – B_L is the sequence that we have to rotate towards the right:



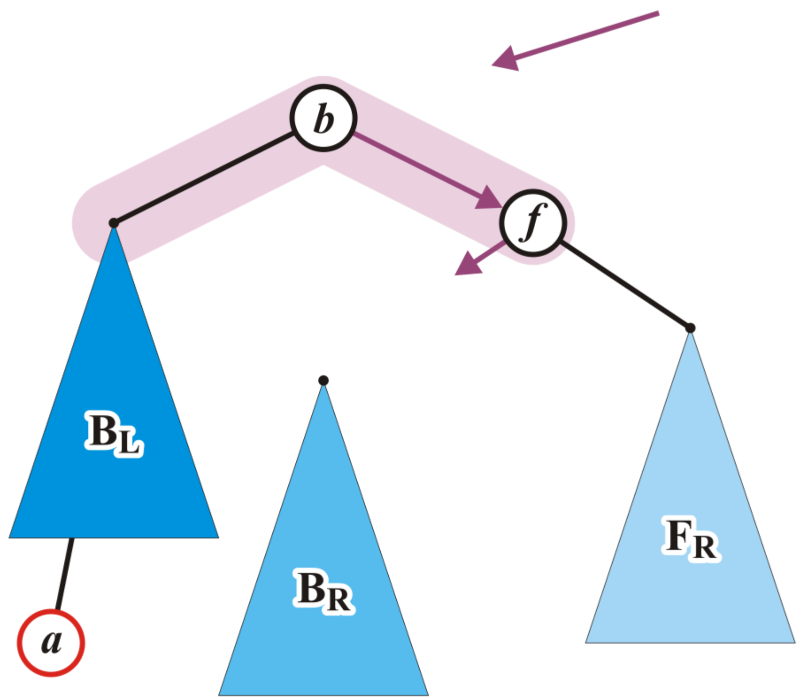
Balanced BST and AVL Trees

- We rotate:



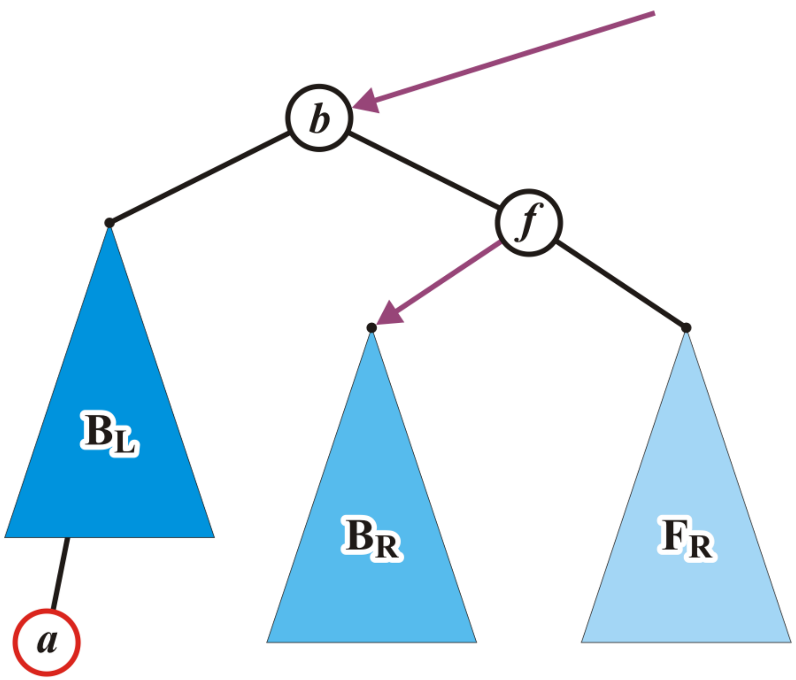
Balanced BST and AVL Trees

- Then re-attach B_R at the obvious place (it is obvious, right?):



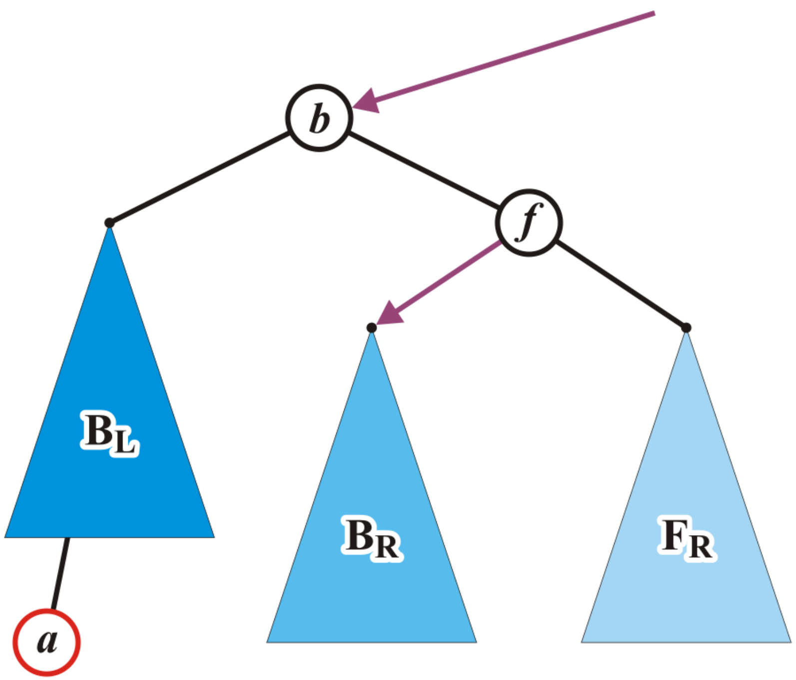
Balanced BST and AVL Trees

- It really is the only “loose” branch where we can attach it — the question is: can we attach it there without breaking the tree's constraints?



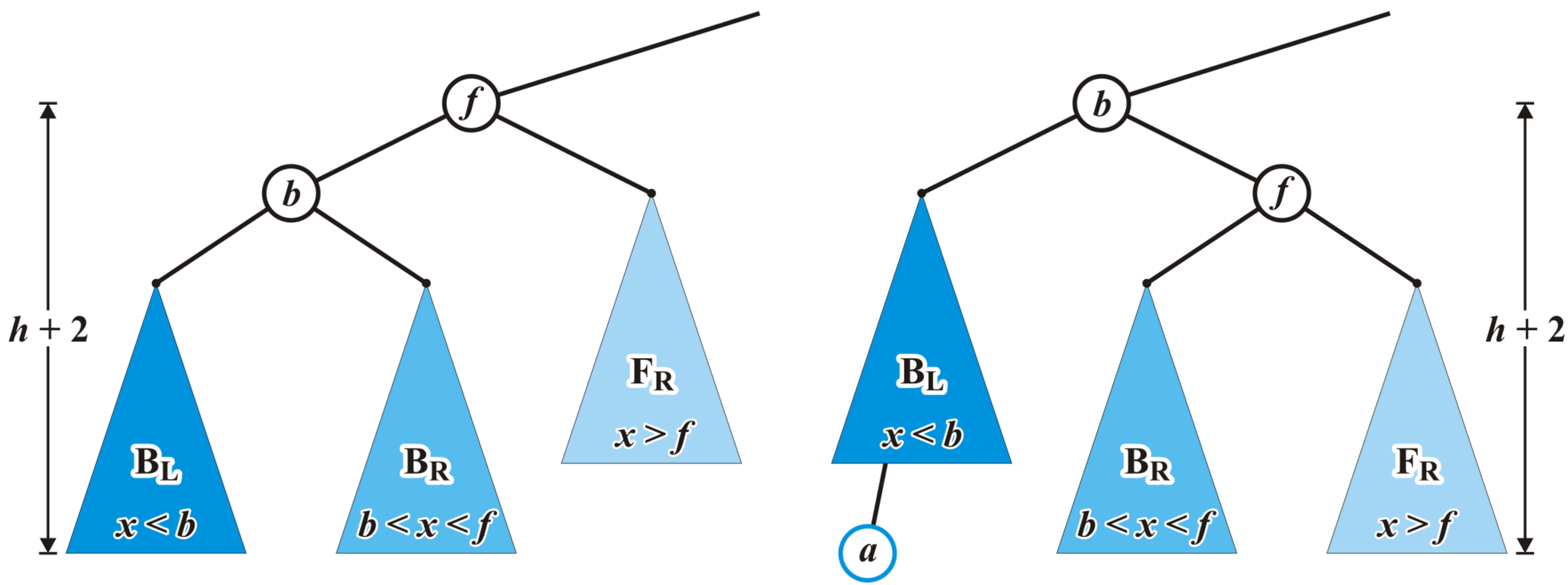
Balanced BST and AVL Trees

- B_R can clearly go as the left sub-tree of *f* — it was originally below *b* which was part of *f*'s left sub-tree. And it can also be part of *b*'s right sub-tree (it was originally *b*'s right sub-tree!):



Balanced BST and AVL Trees

- Another interesting detail is that the height of the tree with root *b* equals the original height of the tree with root *f* — this tells us that the insertion can not affect the balance of any ancestors!



Balanced BST and AVL Trees

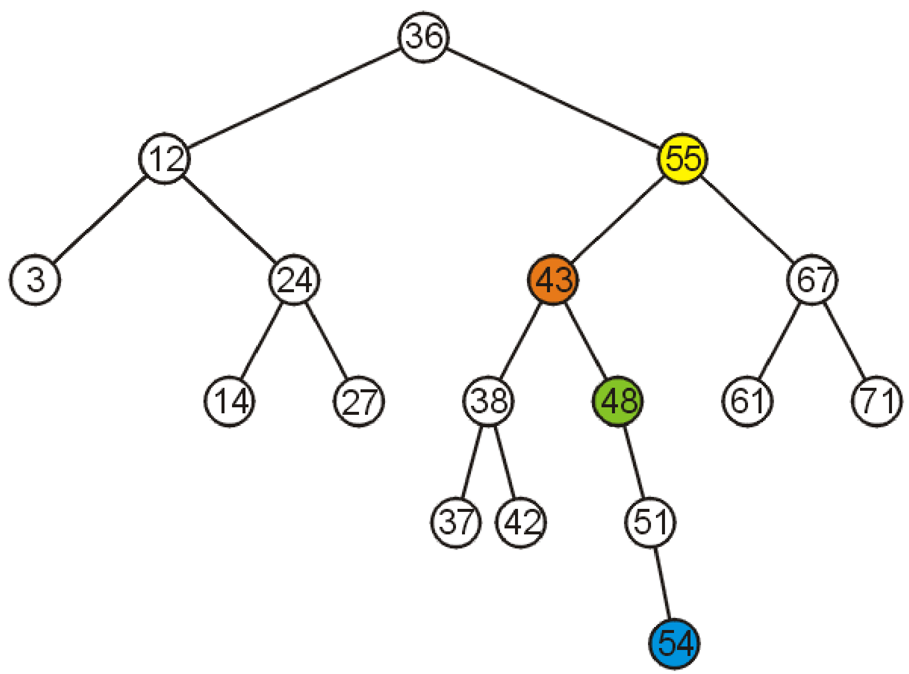
- In other words, this tells us that whenever an imbalance is created, we only need to address it at the deepest level where there is imbalance.

Balanced BST and AVL Trees

- In other words, this tells us that whenever an imbalance is created, we only need to address it at the deepest level where there is imbalance.
  - Things were originally balanced everywhere.
  - Fixing something at a node with depth *d* fixes fixes any imbalance in any node above it.
    - Since nothing else changed in the tree, it must be the case that fixing the imbalance at the deepest node where imbalance is present must be sufficient.

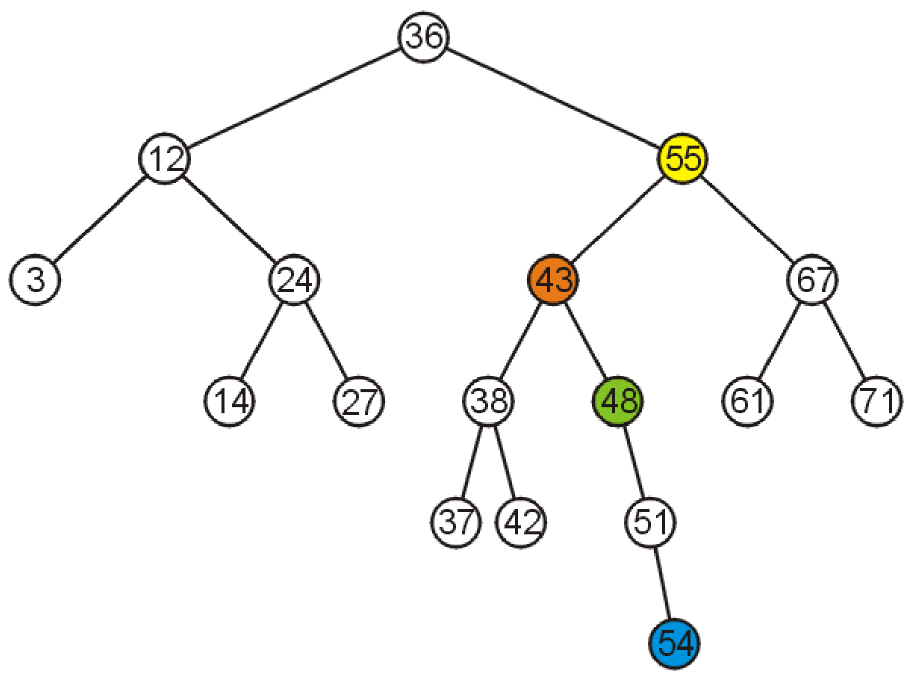
Balanced BST and AVL Trees

- For example, the just added 54 causes imbalance at the root, at 55, and at 48:



Balanced BST and AVL Trees

- However, it's pretty clear that a rotation around 48 fixes the imbalance at all points!

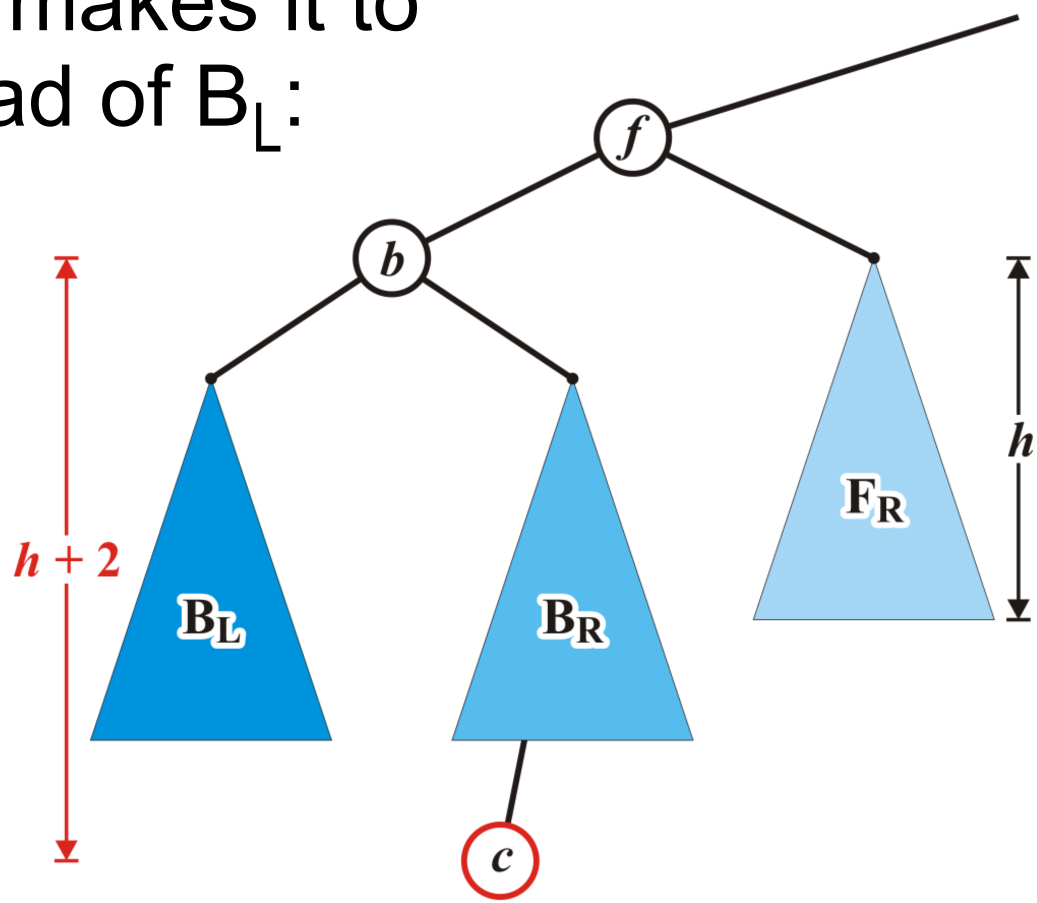


Balanced BST and AVL Trees

- However, not everything is good news...

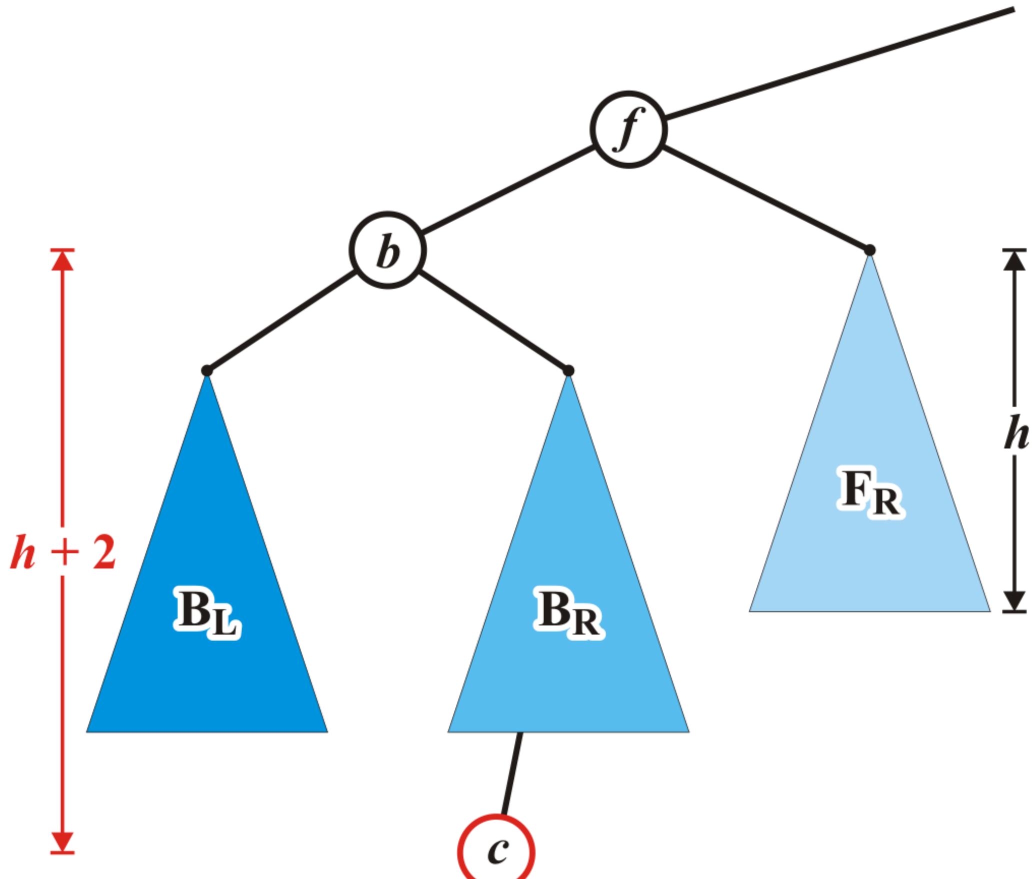
Balanced BST and AVL Trees

- However, not everything is good news... the *uh-oh!* situation happens if the inserted element makes it to B_R instead of B_L:



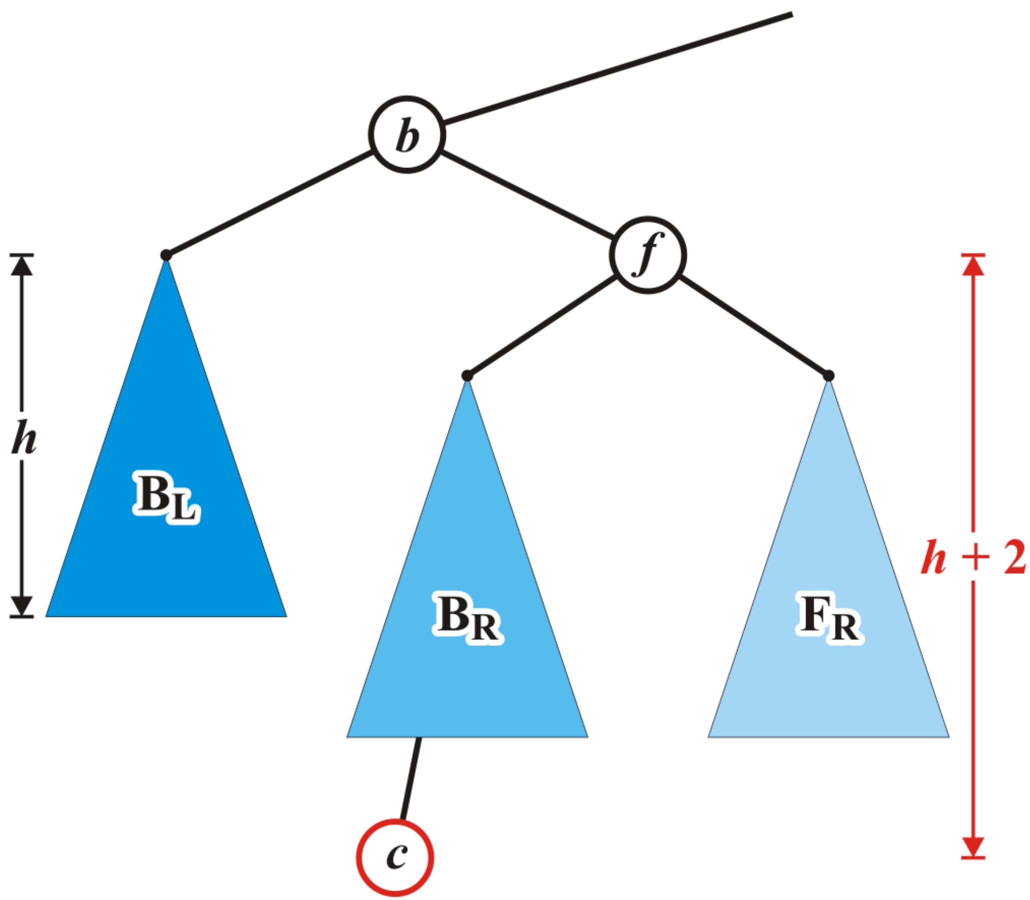
Balanced BST and AVL Trees

- If we try the same trick, we end up with a still imbalanced tree!



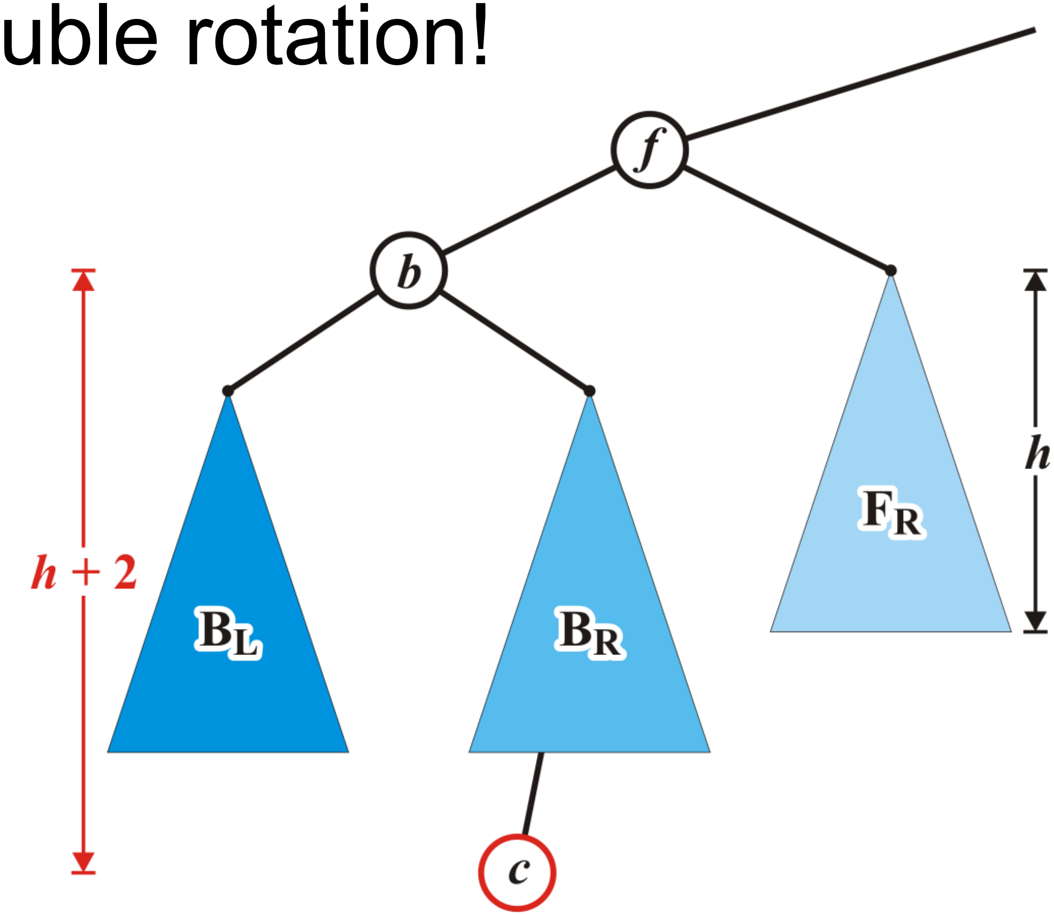
Balanced BST and AVL Trees

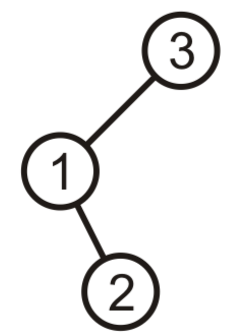
- If we try the same trick, we end up with a still imbalanced tree!



Balanced BST and AVL Trees

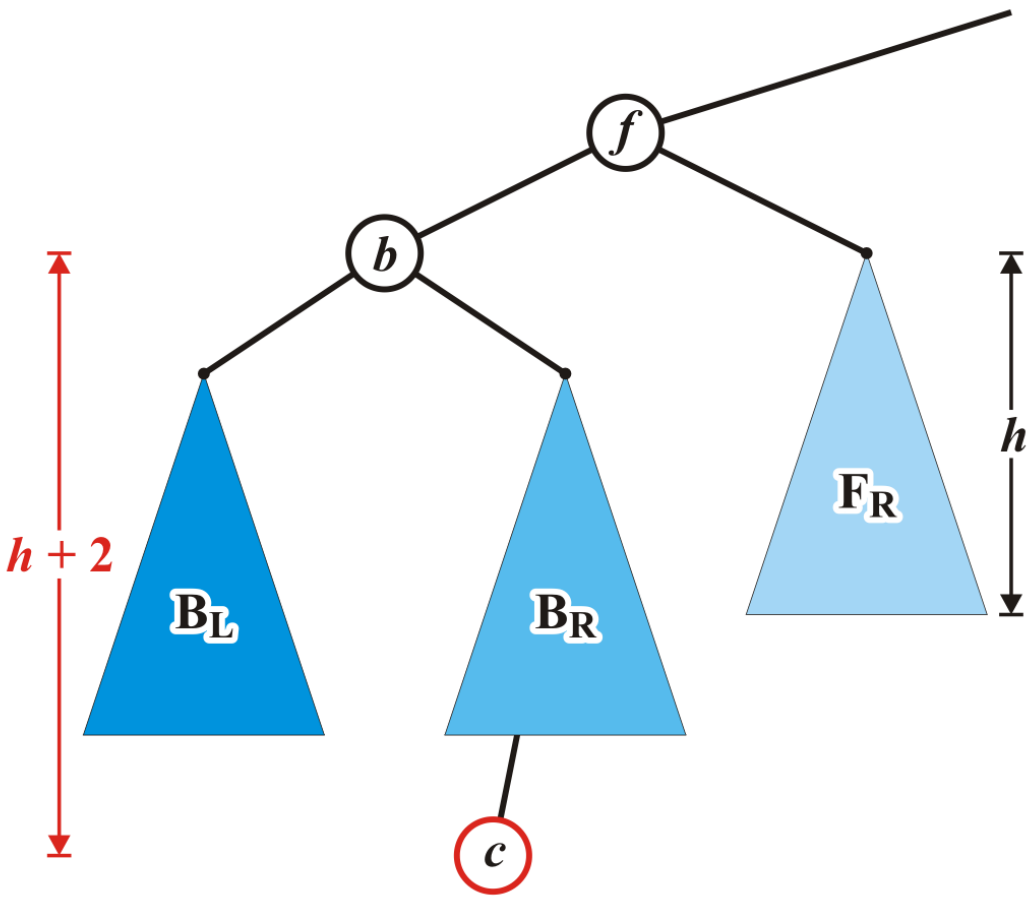
- Notice that what's really going on is that we have a situation similar to the second prototypical example, where we needed a double rotation!

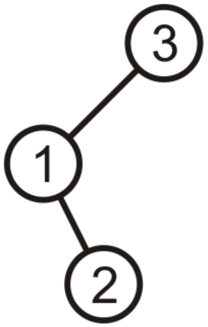




Balanced BST and AVL Trees

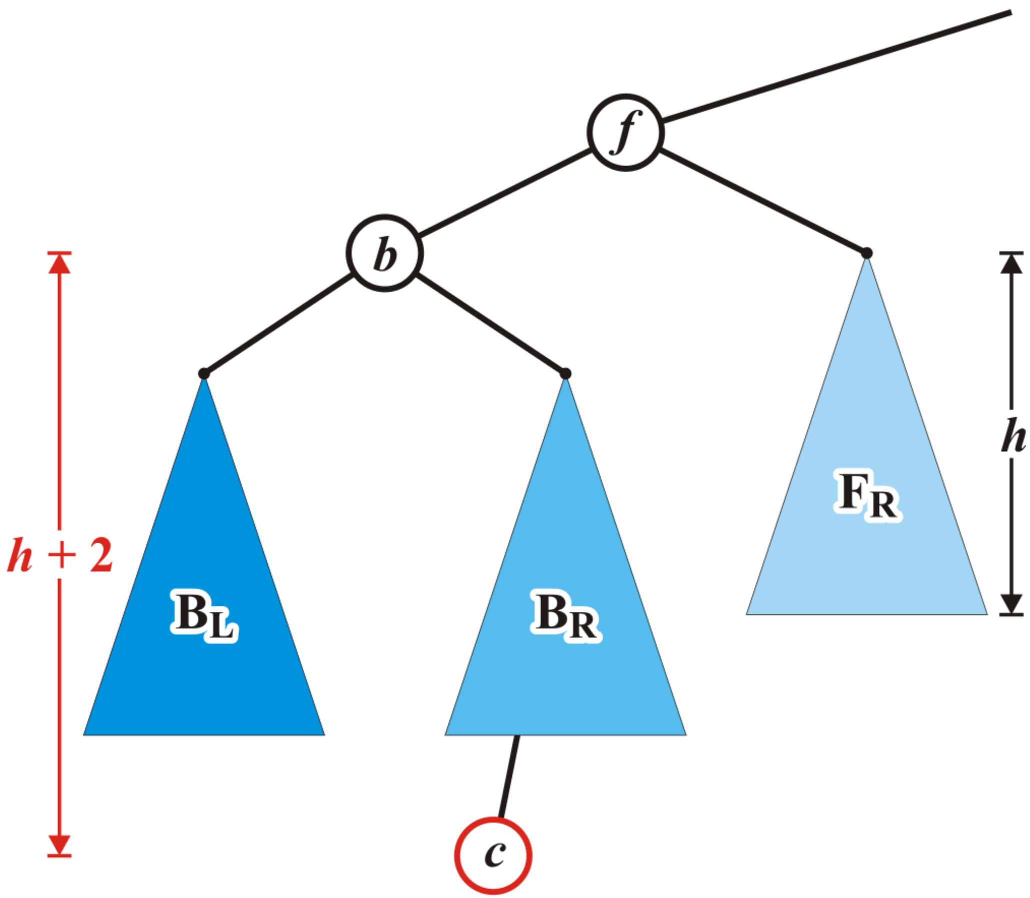
- We saw that we just need to transform the situation into a situation similar to the first case... (*how do we do that?*)

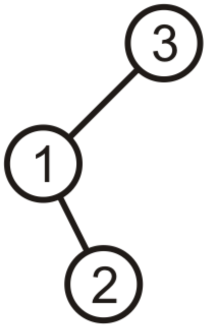




Balanced BST and AVL Trees

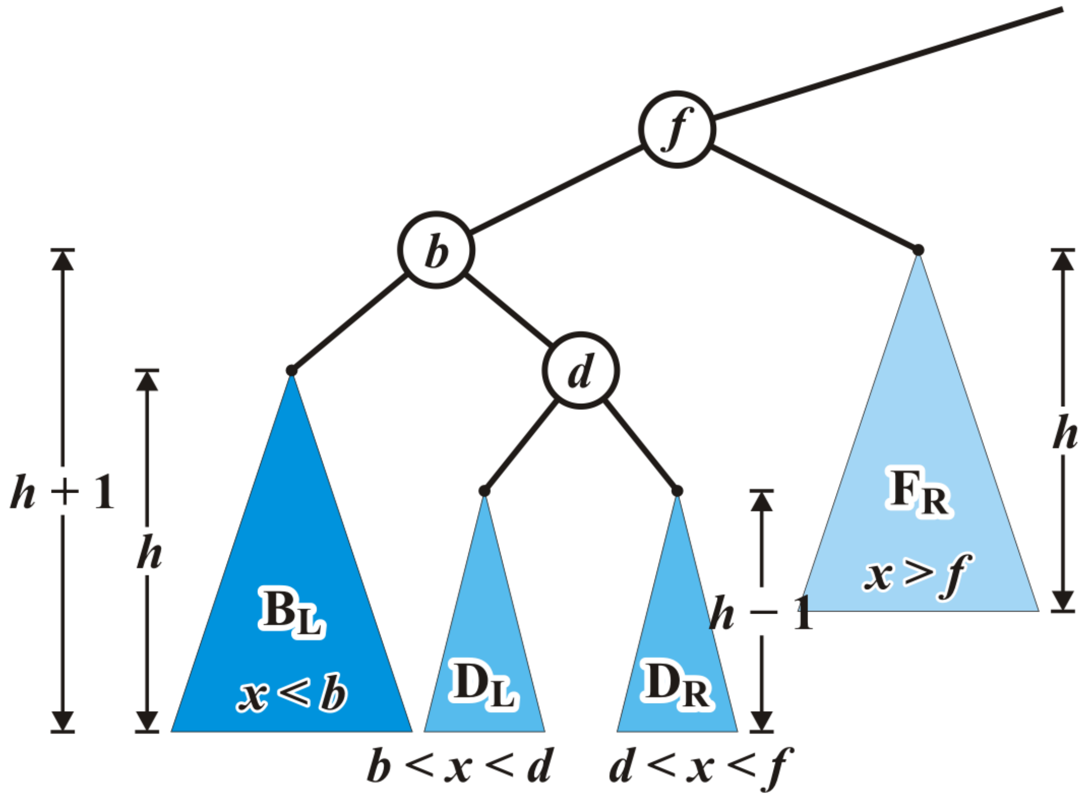
- As we saw with the prototypical example, we first rotate towards the left around *b*. Then we're back in the simpler case!





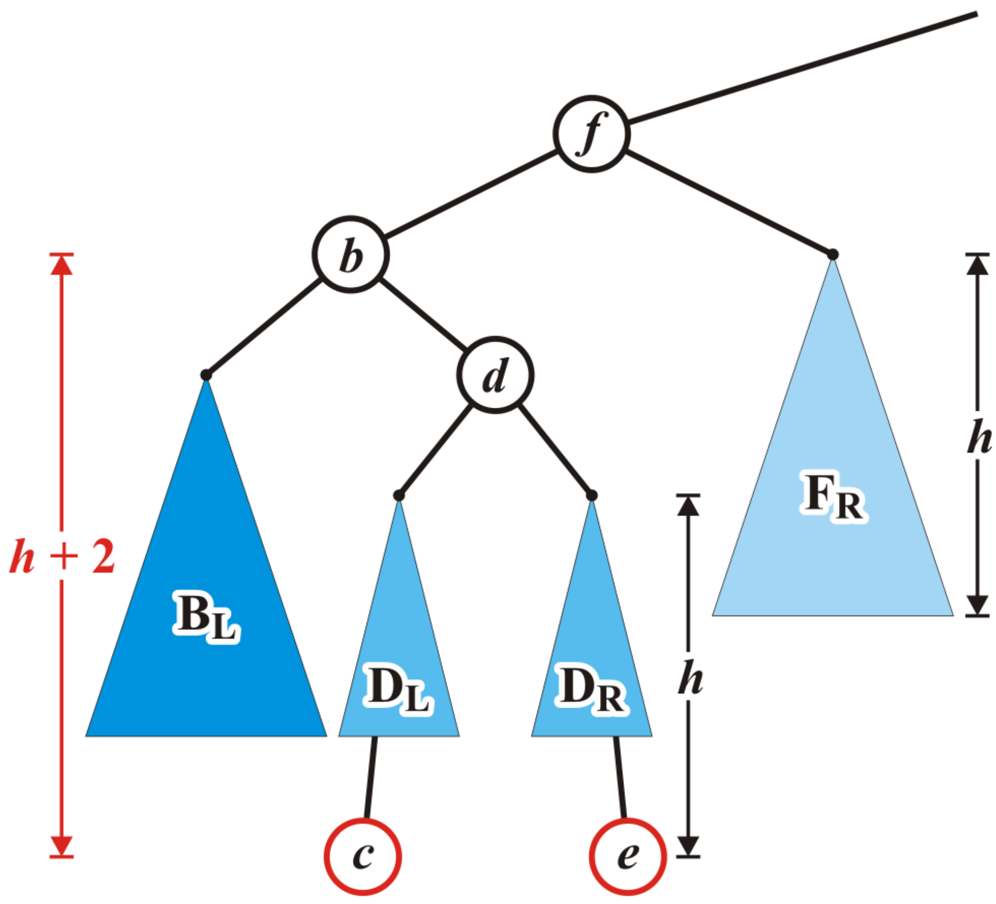
Balanced BST and AVL Trees

- We "zoom in" into the sub-tree B_R:



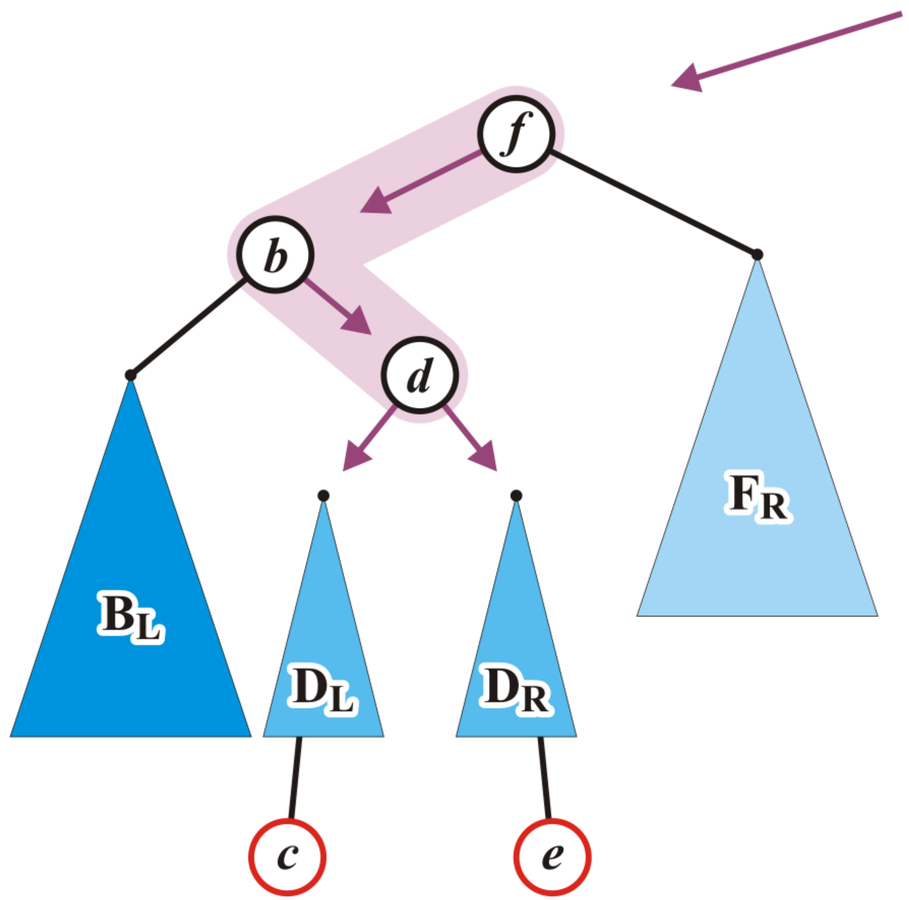
Balanced BST and AVL Trees

- And insert a value (either *c* or *e*)



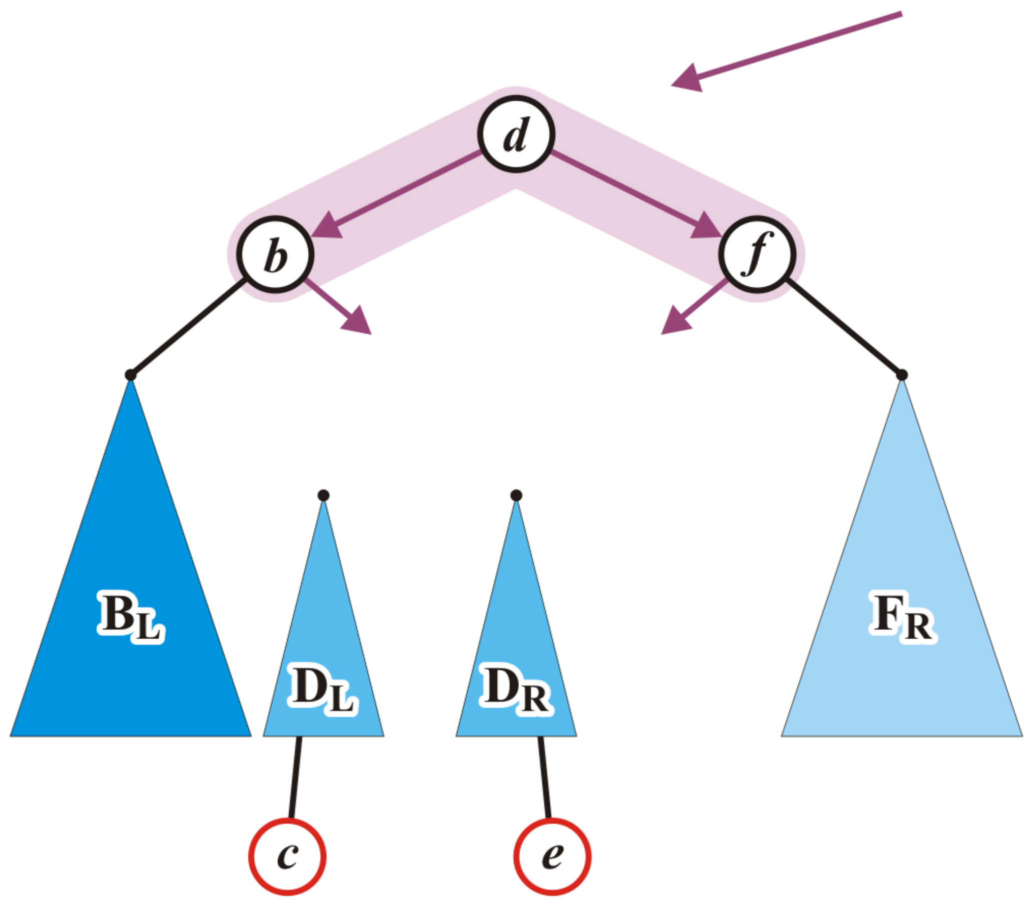
Balanced BST and AVL Trees

- Going from *f* in the direction of the cause of the imbalance, we do a left – right; we first detach things:



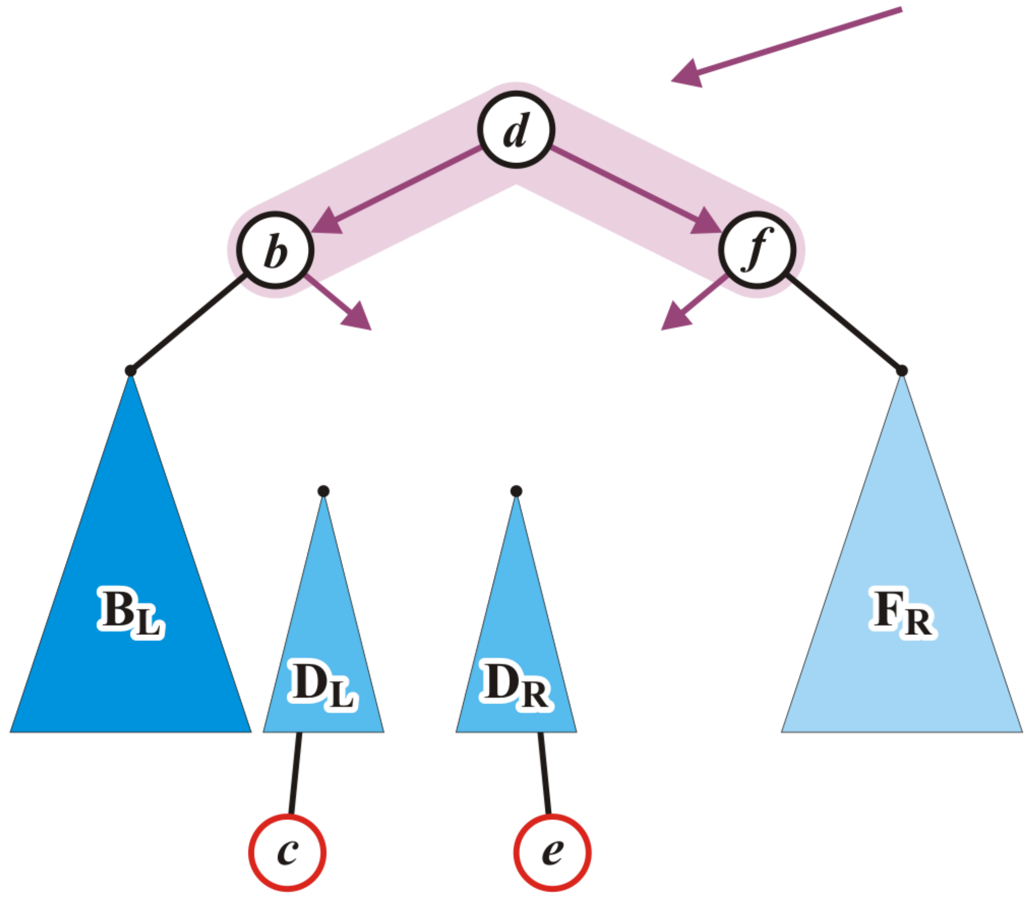
Balanced BST and AVL Trees

- Then readjust:



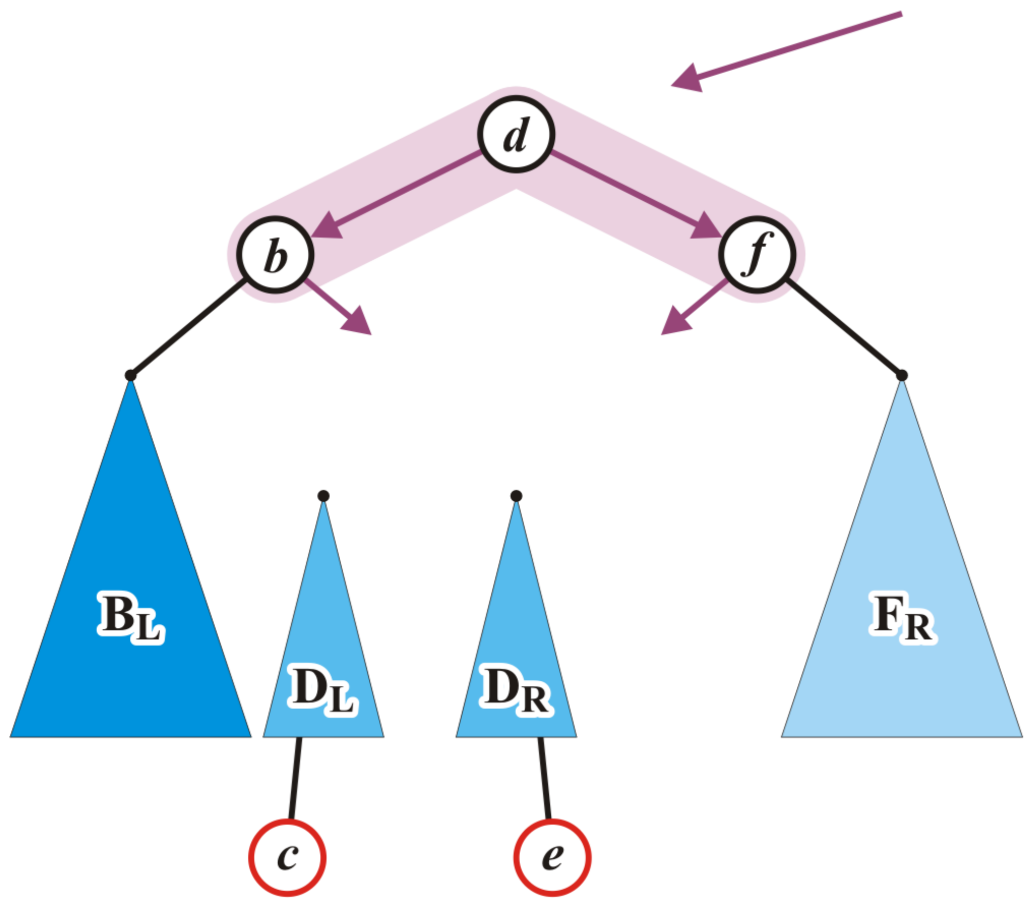
Balanced BST and AVL Trees

- Do I need to make you guess where would D_L and D_R be attached? :-)



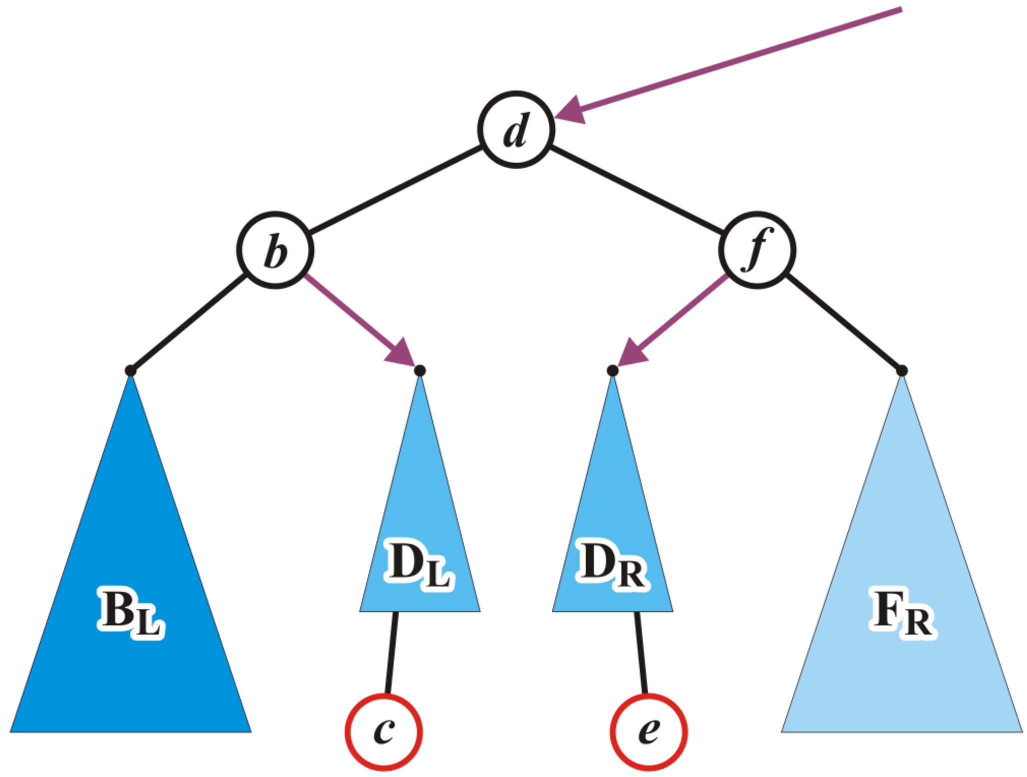
Balanced BST and AVL Trees

- Both can go at the right of *b*, and both can go at the left of *f* — so, we place them according to the constraint due to *d*:



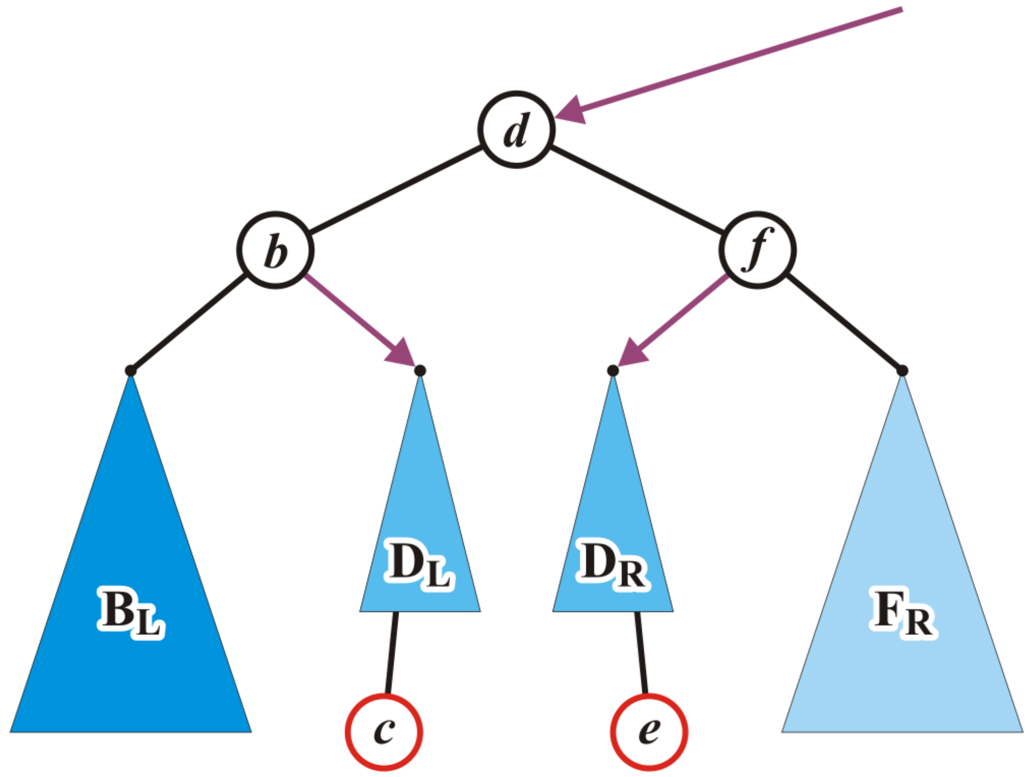
Balanced BST and AVL Trees

- Both can go at the right of *b*, and both can go at the left of *f* — so, we place them according to the constraint due to *d*:



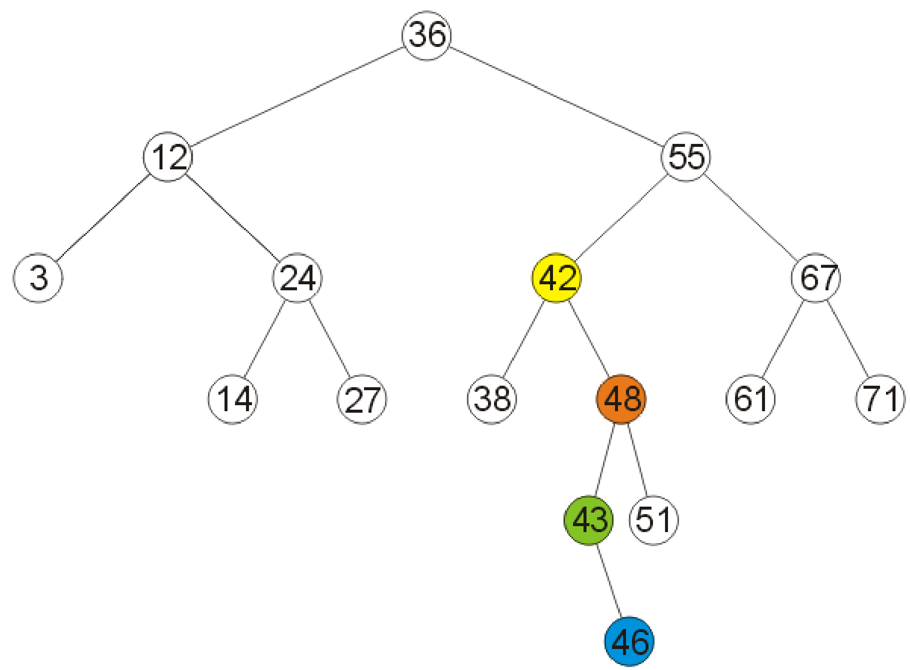
Balanced BST and AVL Trees

- In either case — c, added below D_L, or e, added below D_R — we end up with a balanced tree:



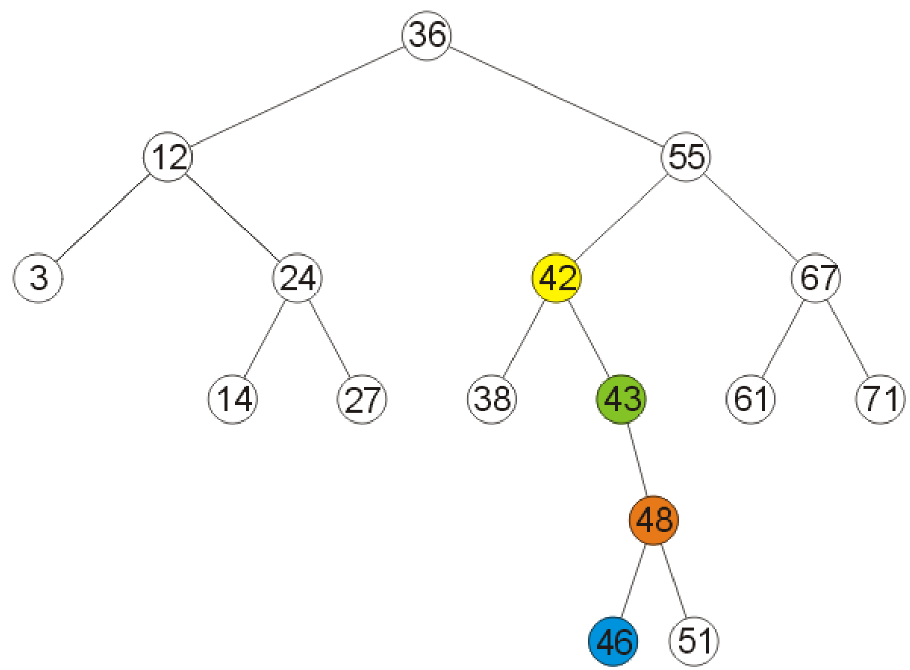
Balanced BST and AVL Trees

- An example: insertion of 46 causes imbalance at nodes 42, 55, and 36 — we address the imbalance at 42 (the deepest node):



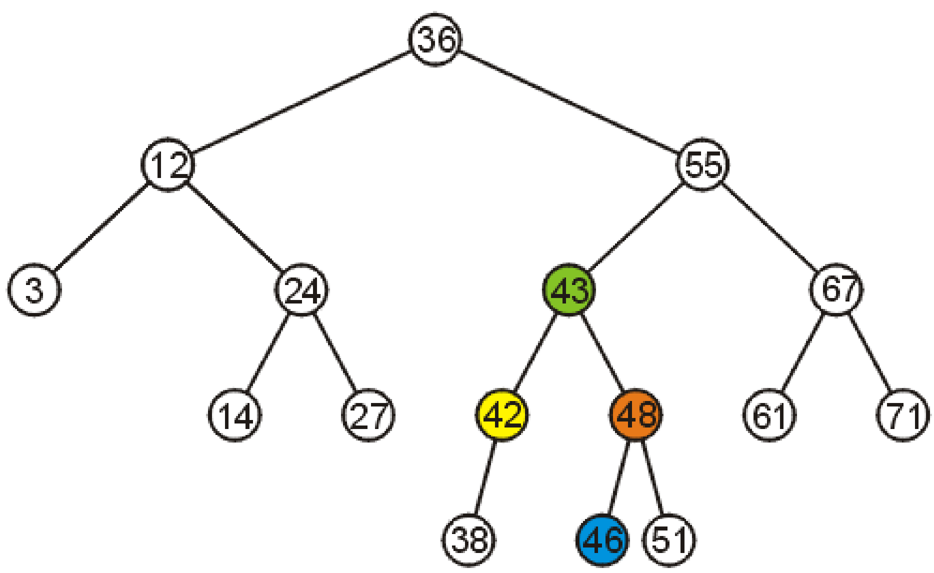
Balanced BST and AVL Trees

- First, rotate towards the right around 43 (detaching 46 and reattaching it as 48's left sub-tree):



Balanced BST and AVL Trees

- Then rotate towards the left around 42:



Balanced BST and AVL Trees

- We observe a piece of good news: these rotations take $\Theta(1)$, and the insertion takes $\Theta(h) = \Theta(\log n)$, so we're in good shape: insertion (including maintaining balance) takes $\Theta(\log n)$
- We'll also see that removals, though a bit more complicated (and more inefficient), also take $\Theta(\log n)$.

Summary

- During today's class:
  - Introduced the notion of balance in BSTs
  - Discussed height-balance (as opposed to weight-balance)
  - Looked into AVL trees:
    - Prototypical examples of re-balancing a tree that becomes unbalanced due to an insertion.
    - Discussed the single-rotation and double-rotation.