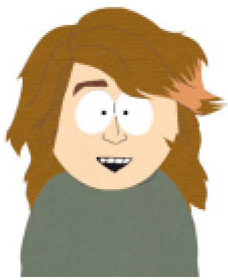
Balanced BST and AVL Trees



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

- 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

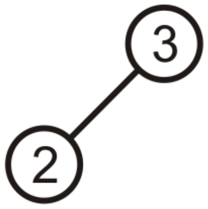
- We'll now look into AVL trees (named after its two inventors, Adelson-Velskii and Landis), a type of height-balanced binary search trees.
- A binary search tree is an AVL tree if:
  - The difference in heights between left and right sub-trees is at most 1, and
  - Both sub-trees are AVL trees

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:

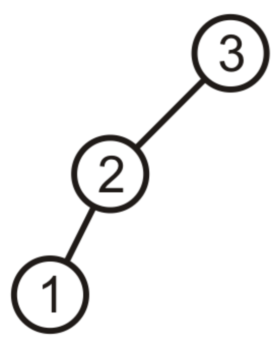
Balanced BST and AVL Trees

- 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:



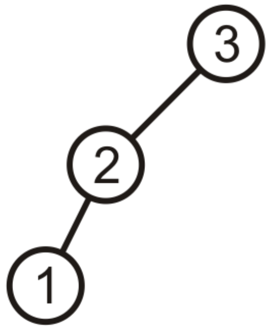
Balanced BST and AVL Trees

- 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?)



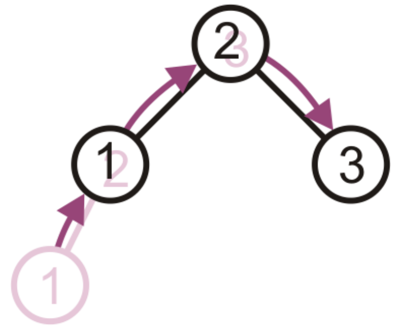
Balanced BST and AVL Trees

- So, we rotate it towards the right:



Balanced BST and AVL Trees

- So, we rotate it towards the right:



Balanced BST and AVL Trees

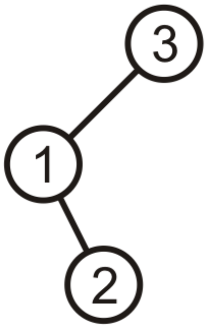
- If we had inserted the sequence 3, 2, 1, the situation would have been essentially the same; the right sub-tree would be the deeper one in that case, so we rotate towards the left in that case.

Balanced BST and AVL Trees

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

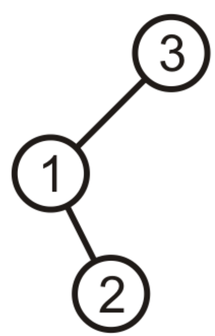
Balanced BST and AVL Trees

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



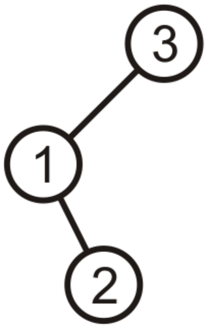
Balanced BST and AVL Trees

- 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?*)



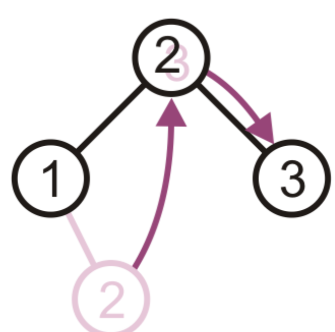
Balanced BST and AVL Trees

- 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:

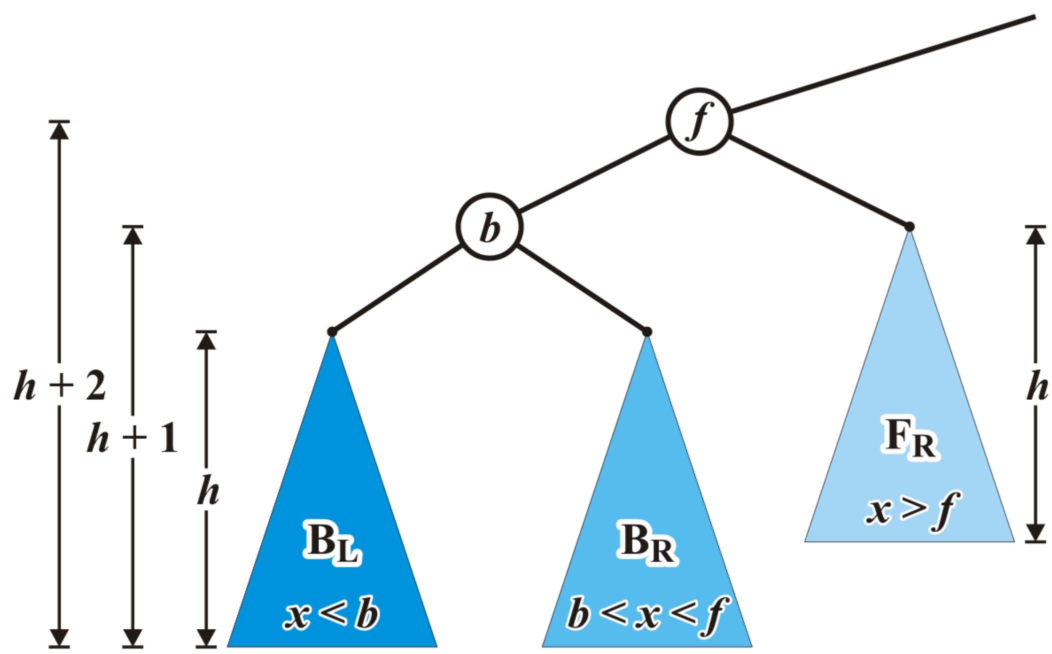


Balanced BST and AVL Trees

- What about a more complicated situation?

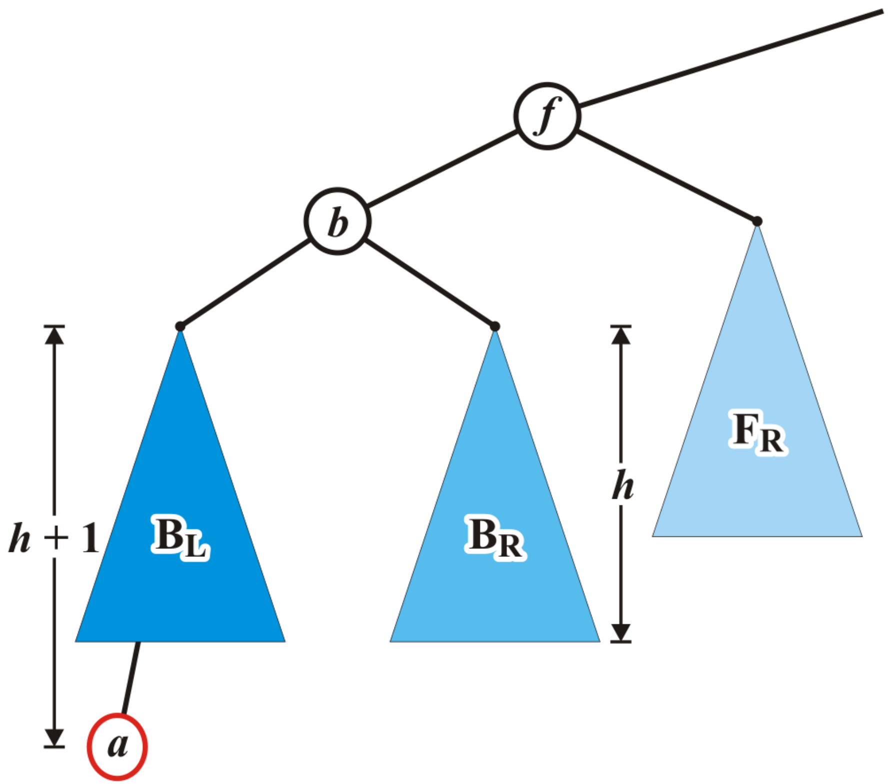
Balanced BST and AVL Trees

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



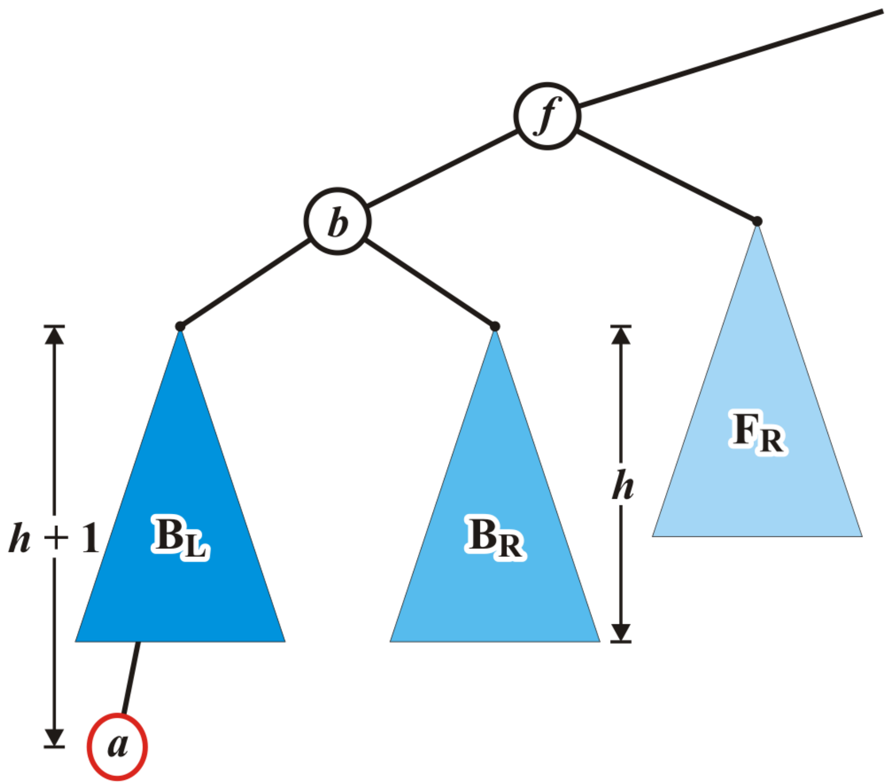
Balanced BST and AVL Trees

- 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

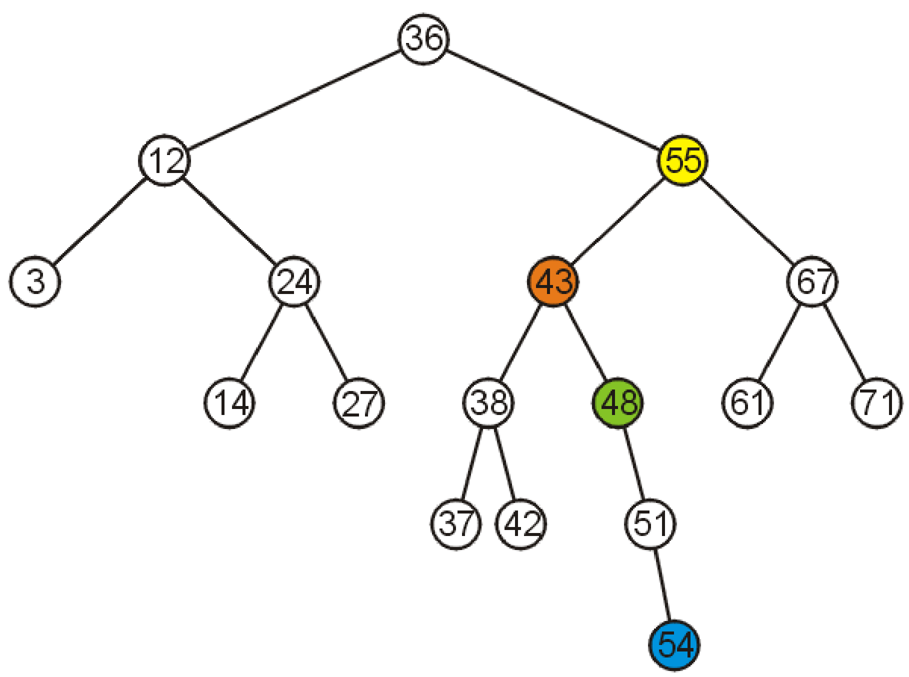
- We'll look into this during class
  - Notice that once we've figured this out, we're pretty much done:
  - An insertion can only cause one of the sub-trees to increase height by at most one.
    - Since these rotations take Θ(1) (right? *why?*), and the insertion takes Θ(h) = Θ(log n), then we're in good shape: insertion (including maintaining balance) takes Θ(log n)
  - We'll also see (most likely the following class) that removals are a bit more complicated (more inefficient), but still Θ(log n).

Balanced BST and AVL Trees

- We'll also take a look at several examples to illustrate these rotations in actual trees.
  - One interesting detail we'll discuss is that whenever an imbalance is created, we only need to fix it at the deepest level (that will automatically fix the imbalance at all levels)

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!