AVL-Tree Operations

AVL tree operations

AVL Trees Invariant: The balance of every node is -1, 0, or 1. When inserting an element to an AVL tree we allow breaking the invariant and then, by re-balancing, we fix it again.

- AVL find: Same as BST find
- AVL insert:
  - First BST insert, then check balance and potentially fix the AVL tree
  - Four different balance cases
- AVL Delete: like insert we do the deletion and then have several balance cases

AVL re-balancing via Rotations

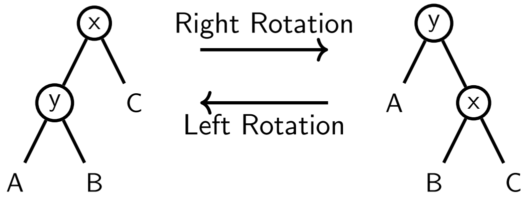
When we insert into an AVL tree, all nodes meet the balance invariant initially.

We find where the value should go, just like in a BST tree, and insert a new leaf there.

However, that may break the balance invariant of the AVL tree.

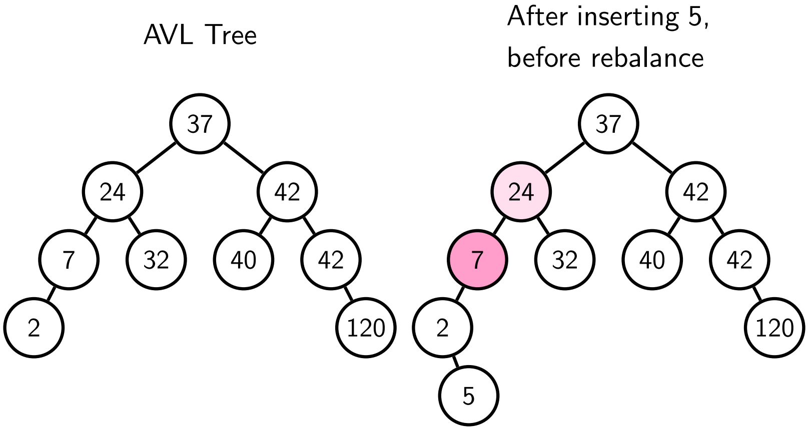
AVL re-balancing via Rotations

We will fix imbalances by a series of rotations:



- $A < y < B < x < C$: rotation preserves this order
- x's right child (C) remains unchanged
- y's left child (A) remains unchanged
- Right rotation:
  - y's right child (B) becomes x's left child
  - x becomes y's right child
- Left rotation:
  - x's left child (B) becomes y's right child
  - y becomes x's left child

AVL tree insert example[1]

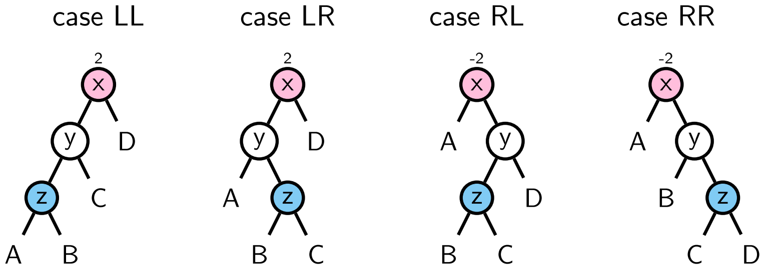


We only find the imbalance in a node on *return* from the insert call to its child node, and fix the *lowest* node with an imbalance first.

[1]Shaffer, *Data Structures and Algorithm Analysis*

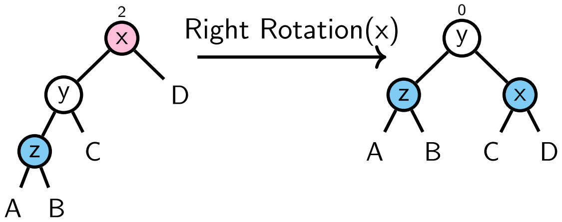
AVL tree insert

Let x be the lowest node where an imbalance occurs, following an insert into subtree z. This imbalance is found on returning from the nested recursive insert call up to node x. There are 4 different cases possible:



AVL tree insert: Case LL

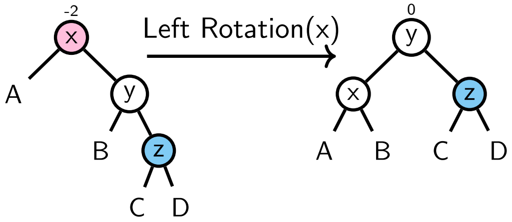
This can be fixed with a *right rotation* at x:



- Before insertion, balance at x had to be 1
- Inserting into z caused imbalance at x, so, after insertion but before rotation, $h(y) = h(D) + 2$
- After insertion, but before rebalancing, $h(y - z - \dots) = h(D) + 2$ and $h(y - C - \dots) = h(D) + 1$ (otherwise y would be the lowest node with imbalance)
- After rotation, balance at y is 0

AVL tree insert: Case RR

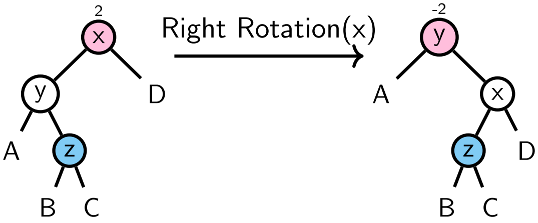
This case is symmetric to LL and can be fixed with a *left rotation* at x:



- Before insertion, balance at x had to be -1
- After rotation, balance at y is 0

AVL tree insert: Case LR

This case raises a probem: the necessary right rotation at x alone does not fix the imbalance:



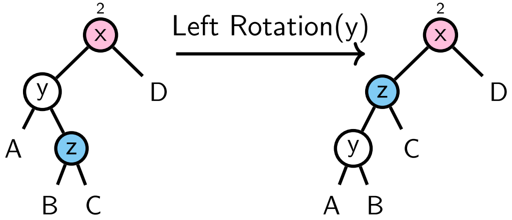
STILL UNBALANCED . . . just the opposite way!

Actually turns it into the RL case

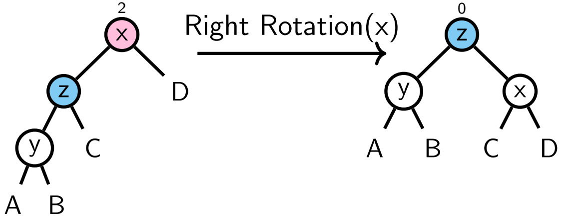
Solution:

- Do a **left rotation** at y first
- Then do a **right rotation** at x.

AVL tree insert: Solution to Case LR

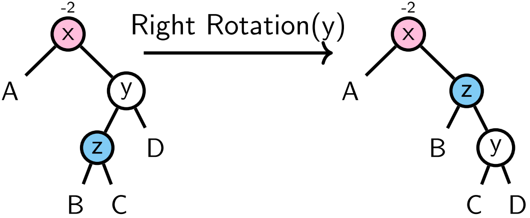


This results in a simple LL case which can be fixed by a right rotation at x:

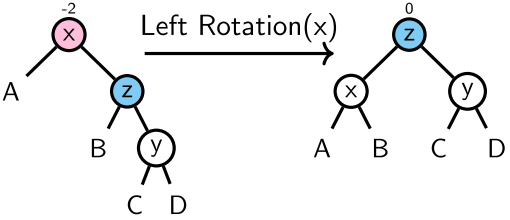


AVL tree insert: Case RL

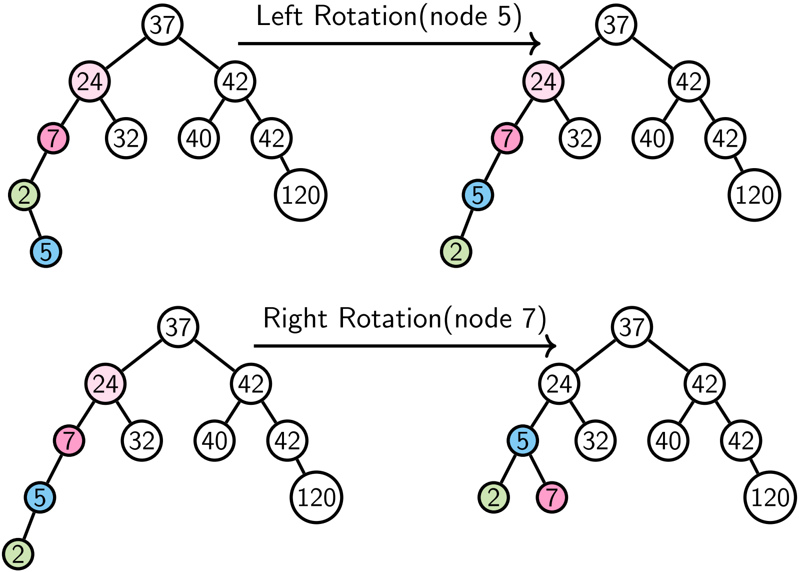
This case is symmetric to the LR case



This results in a simple RR case which can be fixed by a left rotation at x:



AVL tree insert example [Shaffer]



AVL tree deletion

To delete from an AVL tree, the general approach is to modify the BST delete algorithm
(c.f. `dsa-slides-04-03-binary-search-trees.pdf`):

- Delete the node from the tree using the BST algorithm
- On returning up the tree, rebalance as necessary just as for AVL Tree insert

Further reading on AVL trees

- https://www.programiz.com/dsa/avl-tree has a very nice explanation, explains deletion as well and has full code implementations.
- https://www.tutorialspoint.com/data_structures_algorithms/avl_tree_algorithm.htm also has some nice explanations.
- https://www.cs.usfca.edu/~galles/visualization/AVLtree.html allows you to insert and delete values in an AVL tree and animates the operations.