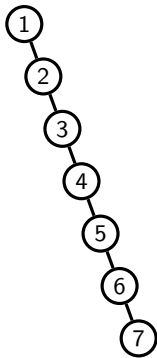
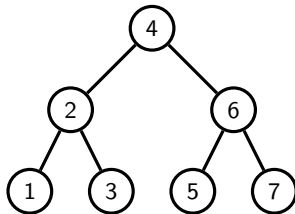


AVL Trees

Balancedness of trees matters



vs.



Can we assume extra conditions to make sure that the height of the tree is under control?


AVL Tree

The **height** of a node is the length of the longest path from that node to a leaf node (compare to the height of a tree)

The **balance** at a node is

$$\left(\begin{array}{c} \text{The height of} \\ \text{the left subtree} \end{array} \right) - \left(\begin{array}{c} \text{The height of} \\ \text{the right subtree} \end{array} \right)$$

Examples:

- Note that the height of an empty tree is -1
- The balance at a leaf node is $(-1) - (-1) = 0$.
- The balance at the root of  is $(-1) - 0 = -1$.

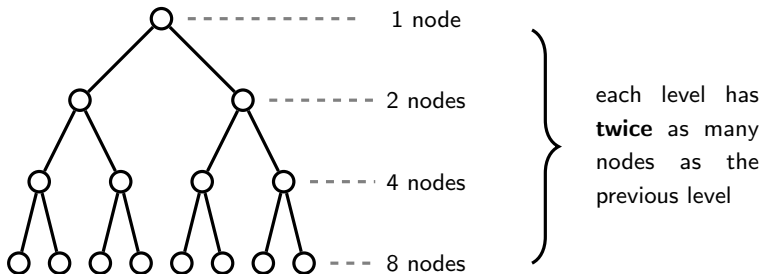
- The balance of the root of  is $1 - 1 = 0$.

AVL Tree

Definition: A Binary Search Tree is said to be **AVL** when the balance at *every* node is either 1, 0 or -1 .

Perfect Binary Tree = Maximal AVL tree of a given height

Assume that the tree is **perfectly balanced**, that is, the balance of each node is 0. How many nodes does the tree have?



If the tree has height h , then the number of nodes is



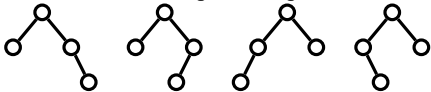
$$1 + 2 + 4 + 8 + \cdots + 2^h = 2^{h+1} - 1$$

Another way of saying that the tree is perfectly balanced is that

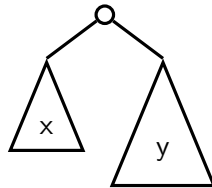
1. every node, except for leaf nodes, has exactly two children and
2. all leaf nodes are on the same level.

Fibonacci trees = Minimal AVL trees of a given height

How many nodes does the tree have if the balance of each (non-leaf) node is either 1 or -1?

- If the height is 1 – two options:  or  \Rightarrow size is 2
- If the height is 2:  \Rightarrow size is *always* 4

- In general, we obtain the **Fibonacci tree of height $h+2$** (called T_{h+2}), from the Fibonacci trees of height h and $h+1$ (called T_h and T_{h+1} , respectively) as:



$$\Rightarrow \text{the size of } T_{h+2} = 1 + \text{size of } T_h + \text{size of } T_{h+1}$$

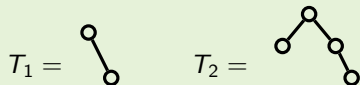
We see that there are two **minimal** AVL trees of height 1 and four **minimal** AVL trees of height 2. However, those minimal trees are all the same, except for the ordering of children. Similarly, the minimal AVL trees of larger heights are also of the same size.

For now, we are only interested in the **size** of a minimal AVL tree of a certain height. Because all minimal AVL trees of a given height have the same size, we can pick just one representative AVL tree for every height.

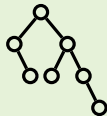
The following procedure describes a construction of **Fibonacci trees** $T_{-1}, T_0, T_1, T_2, T_3, \dots$, where T_h is the minimal AVL tree of height h (up to ordering of children):

- T_{-1} is the empty tree
- T_0 is the one element tree
- T_{h+2} is obtained by making T_h and T_{h+1} children of the root node (as shown in the picture on the previous slide).

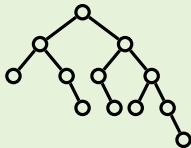
For example, to construct T_3 we combine T_1 and T_2 . Because



we obtain that T_3 is the following tree



and T_4 is the following tree



and so on.

Fibonacci trees and Fibonacci numbers

Denote the size of T_h as $|T_h|$:

h	$ T_h $
-1	0
0	1
1	$1 + T_0 + T_1 = 2$
2	$1 + T_1 + T_2 = 4$
3	$1 + T_2 + T_3 = 7$
4	$1 + T_3 + T_4 = 12$
5	$1 + T_4 + T_5 = 20$
\vdots	\vdots

k	F_k
0	0
1	1
2	$F_0 + F_1 = 1$
3	$F_1 + F_2 = 2$
4	$F_2 + F_3 = 3$
5	$F_3 + F_4 = 5$
6	$F_4 + F_5 = 8$
\vdots	\vdots

$$\begin{aligned} |T_{h+2}| &= 1 + |T_h| + |T_{h+1}| \\ 1 + |T_{h+2}| &= (1 + |T_h|) + (1 + |T_{h+1}|) \end{aligned}$$

$$\text{vs} \quad F_{k+2} = F_k + F_{k+1}$$

$$\text{initial values plus equation} \implies |T_h| + 1 = F_{h+3}$$

Computing the bounds

If an AVL tree has height h then its size is

- \leq the size of the perfectly balanced tree of height h , and
- \geq the size of Fibonacci tree of height h (that is, $|T_h|$).

Therefore (because $|T_h| = F_{h+3} - 1$)

$$F_{h+3} - 1 \leq \text{the size of the tree} \leq 2^{h+1} - 1$$

Binet's formula:
$$F_k = \frac{\left(\frac{\sqrt{5}+1}{2}\right)^k - \left(\frac{\sqrt{5}-1}{2}\right)^k}{\sqrt{5}} \approx O(1.61^k)$$

\implies the size of an AVL tree is exponential in its height

\implies the height of an AVL tree is logarithmic in its size

\implies **an AVL tree of size n has height $\mathcal{O}(\log n)$**

If we have a tree of height h which is AVL, we know that the size of our tree could be as small as $|T_h|$, or as big as the size of the perfectly balanced tree of height h . However, in general it is somewhere in between.

Let n be the size of an AVL tree, then we have that

$$F_{h+3} - 1 \leq n \leq 2^{h+1} - 1$$

These are conditions on size, given that we know the height of our AVL tree. Conversely, if we know the size and we know that the tree is AVL, then what implications does this have for the height? Let's express the conditions for height in terms of n .

For example $n \leq 2^{h+1} - 1$, gives us that

$$\log_2 n \leq \log_2(2^{h+1} - 1) \leq \log_2(2^{h+1}) = h + 1.$$

In other words, height h is at least $\log_2 n - 1$.

Consequences for time complexities

For a Binary Search Tree implemented as a height-balanced tree (e.g. AVL tree), where n = the number of nodes of the tree:

- `search(x)` goes through at most $O(\log n)$ -many levels
 $\implies O(\log n)$ steps
- `insert(x)` :
 1. We first find the leaf where to insert `x` $\implies O(\log n)$ steps,
 2. then, insert it there $\implies O(1)$ steps,
 3. finally, on the way up, in each node we do balancing
 $\implies O(\log n)$ -many times we do $O(1)$ steps
 $\implies O(\log n)$ steps in total
- `delete(x)` is similar to `insert`, it also takes $O(\log n)$ steps

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

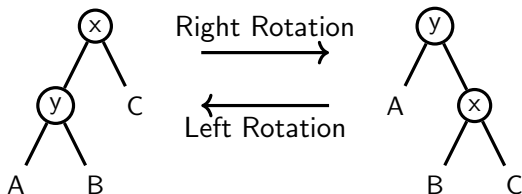
When we insert into an AVL tree, all nodes meet the balance invariant initially.

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

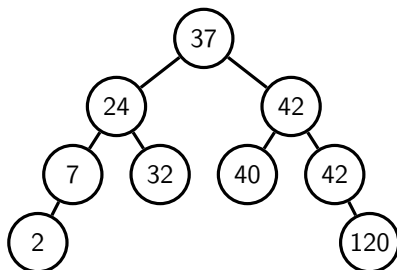
We will fix imbalances by a series of rotations:



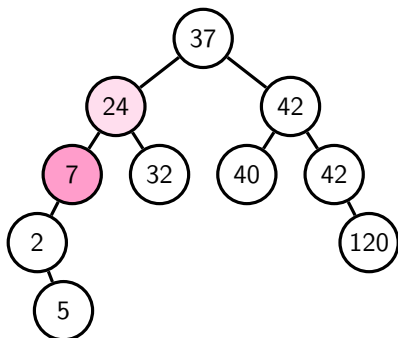
- $A < y < B < x < C$: rotation preserves this order
- x 's right child (C) remains unchanged
- y 's left child (A) remains unchanged
- Right rotation:
 - y 's right child (B) becomes x 's left child
 - x becomes y 's right child
- Left rotation:
 - x 's left child (B) becomes y 's right child
 - y becomes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

AVL tree insert example¹

AVL Tree



After inserting 5,
before rebalance



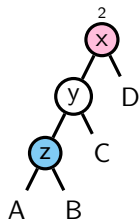
We only find the imbalance in a node on *return* from the insert call to its child node, and fix the *lowest* node with an imbalance first.

¹Shaffer, *Data Structures and Algorithm Analysis*

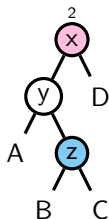
AVL tree insert

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:

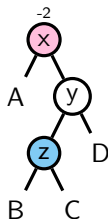
case LL



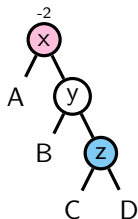
case LR



case RL

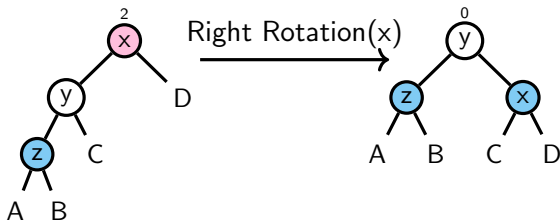


case RR



AVL tree insert: Case LL

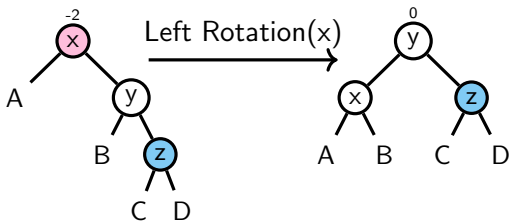
This can be fixed with a *right rotation* at x :



- 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

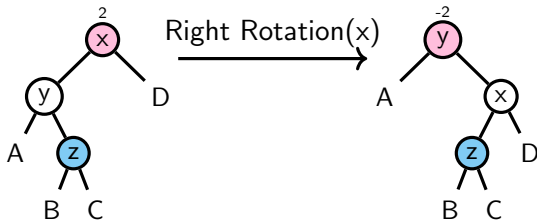
This case is symmetric to LL and can be fixed with a *left rotation* at x :



- 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 problem: the necessary right rotation at x alone does not fix the imbalance:



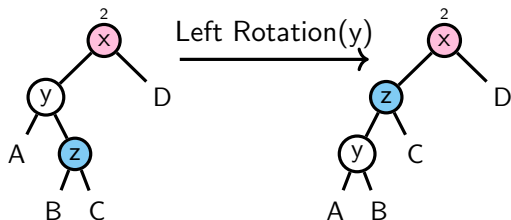
STILL UNBALANCED ... just the opposite way!

Actually turns it into the RL case

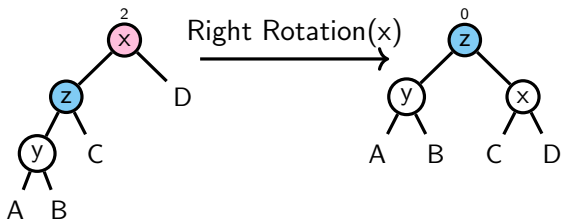
Solution:

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

AVL tree insert: Solution to Case LR

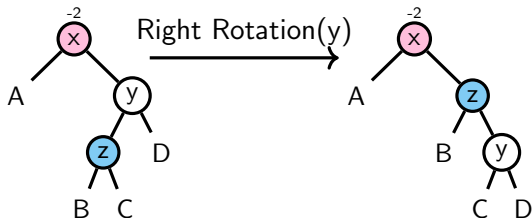


This results in a simple LL case which can be fixed by a right rotation at x :

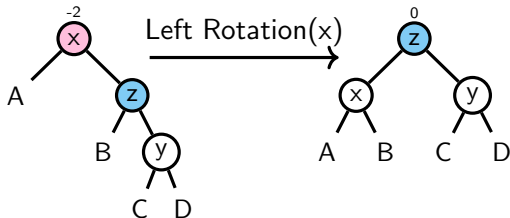


AVL tree insert: Case RL

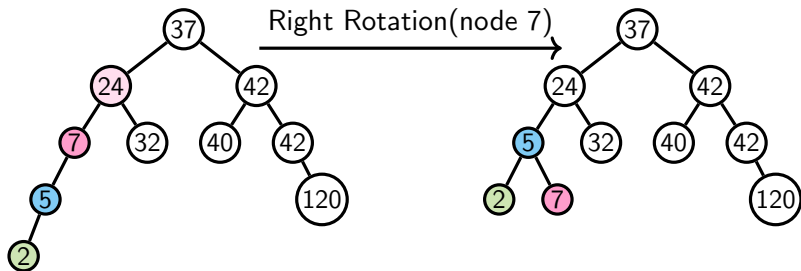
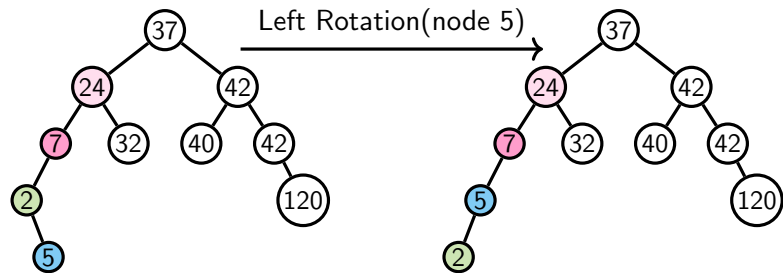
This case is symmetric to the LR case



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.