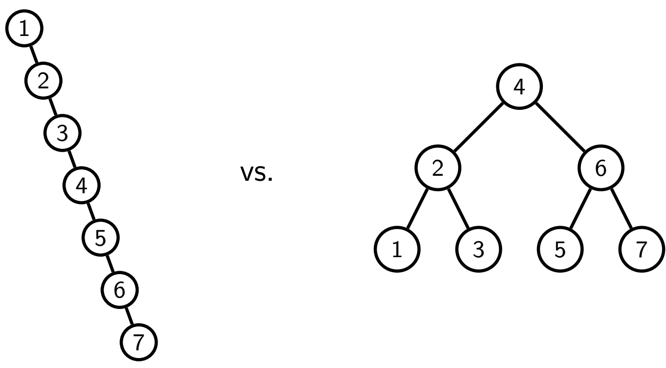
AVL Trees

Balancedness of trees matters



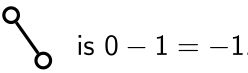
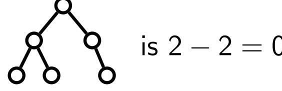
Can we assume extra conditions to make sure that the height of the tree is under control?

AVL Tree

The **imbalance** at a node is

$$\text{the height of its left subtree} - \text{the height of its right subtree}$$

Examples:

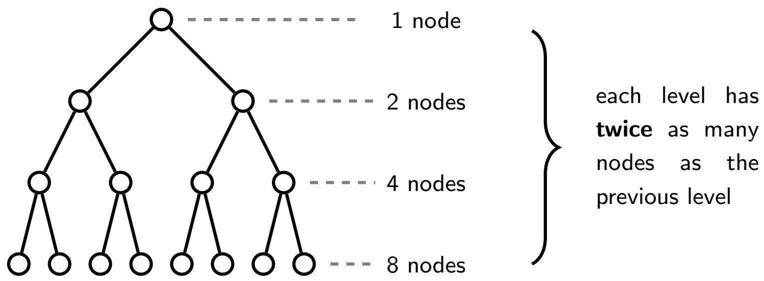
- The imbalance at a leaf node is $0 - 0 = 0$.
- The imbalance of the root of  is $0 - 1 = -1$.
- The imbalance of the root of  is $2 - 2 = 0$.

Definition

A binary tree is said to be **AVL** when the imbalance at each 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

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

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

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

By the way, the formula which calculates the number of nodes could be also written as

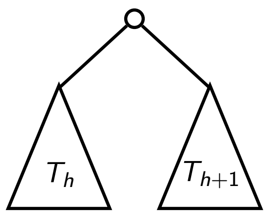
$$2^0 + 2^1 + 2^2 + 2^3 + \cdots + 2^{h-1}$$

Fibonacci trees = Minimal AVL trees of a given height

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

- If the height is 2 – two options: or $\Longrightarrow$ size is 2
- If the height is 3:

  $\Longrightarrow$ 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:



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

We see that there are two **minimal** AVL trees of height 2 and four **minimal** AVL trees of height 3. 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** $T_1, T_2, T_2, \ldots$, where T_h is the minimal AVL tree of height h (up to ordering of children):

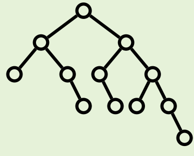
- T_0 is the empty tree
- T_1 is the one element tree
- T_{h+2} is obtained by making T_h and T_{h+1} children of the root node (as shown in the picture on the previous slide).

For example, to construct T_4 we combine T_1 and T_2. Because

$T_2 =$ $\qquad$ $T_3 =$

we obtain that T_4 is the following tree

and T_5 is the following tree



and so on.

Fibonacci trees and Fibonacci numbers

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

h	$\|T_h\|$
0	0
1	1
2	$1 + \|T_0\| + \|T_1\| = 2$
3	$1 + \|T_1\| + \|T_2\| = 4$
4	$1 + \|T_2\| + \|T_3\| = 7$
5	$1 + \|T_3\| + \|T_4\| = 12$
6	$1 + \|T_4\| + \|T_5\| = 20$
$\vdots$	$\vdots$

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	$F_4 + F_5 = 8$
$\vdots$	$\vdots$

$$|T_{h+2}| = 1 + |T_h| + |T_{h+1}|$$
$$1 + |T_{h+2}| = (1 + |T_h|) + (1 + |T_{h+1}|)$$

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

$$\implies |T_h| + 1 = F_{h+2}$$

We can slightly modify the sum

$$|T_{h+2}| = 1 + |T_h| + |T_{h+1}|$$

to

$$1 + |T_{h+2}| = (1 + |T_h|) + (1 + |T_{h+1}|)$$

Then, we see that the sizes of Fibonacci trees are computed similarly to Fibonacci numbers.

In particular, if we substitute F_{h+2} for $1 + |T_{k+2}|$, F_h for the first bracket and F_{h+1} for the second bracket we obtain the following formula:

$$F_{k+2} = F_k + F_{k+1}$$

This is precisely the formula which defines Fibonacci numbers!

There is one caveat, though. The sequence $|T_0|, |T_1|, |T_2|, ...$ is shifted by two elements, when compared $F_0, F_1, F_2, ...$. We have that

$$|T_h| + 1 = F_{h+2}$$

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 $|T_h| = F_{h+2} - 1$)

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

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

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

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

$\Longrightarrow$ **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

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

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 $\log_2 n - 1$.

Furthermore, $F_{h+2} - 1 \leq n$ gives us the upper bound on the height of our tree. In very simplified terms we can assume that $F_h \approx 1.61^h \times a$ (where a is some constant, e.g. $a = \frac{1}{\sqrt{5}}$).

Then,

$$1.61^{h+2} \times a \;\approx\; F_h - 1 \leq n$$

from which we obtain that

$$\log_{1.61}(1.61^{h+2} \times a) \leq \log_{1.61} n$$

Because

$$\log_{1.61}(1.61^{h+2} \times a) = \log_{1.61}(1.61^{h+2}) + \log_{1.61} a = h + 2 + \log_{1.61} a$$

and because constants don't matter for $\mathcal{O}$, we obtained that

$$h \text{ is in } \mathcal{O}(\log n).$$

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

Invariants

Invariants are conditions about the data structure/state of your program

- which you can assume to be true before executing an operation, then
- when you are modifying the data structure, you can violate them,
- as long as you fix them before you finish with the operation.

  (Therefore, the next time, when you execute another operation, you can again assume that the invariants are true.)

Invariants help you to write programs correctly!

Analogy from the real life: Invariants are like kitchen posters (e.g. plates go to the cupboards, cutlery goes to the drawer, ...):

- When you come to the kitchen, you can assume that everything is as the kitchen poster says.
- When you're working in the kitchen, you take plates and knifes out. In other words, you don't care what the poster says.
- But, before you finish, you put everything back in place!

  (So that when somebody else comes to the kitchen, they know where to find everything.)

In short: We tidy up the kitchen after we are done!

Example

Loop invariants:

```
1  while (<loop condition>) {
2      // at the beginning of the loop, we assume that
3      // the invariants hold
4
5      <loop body> // here we can violate the invariants
6
7      // here we have to make sure that the invariants
8      // hold again for the next iteration
9  }
10
11 // the invariant are true after the loop finishes
```

AVL Trees: The invariant is that the imbalance of every node is -1, 0, or 1. When inserting an element to an AVL tree we first break the invariant and then, by balancing, we fix it again.