47% - MaxFlow, MinCut
41% - AVL, RB
31% - MST, TopSort, Shortest Path

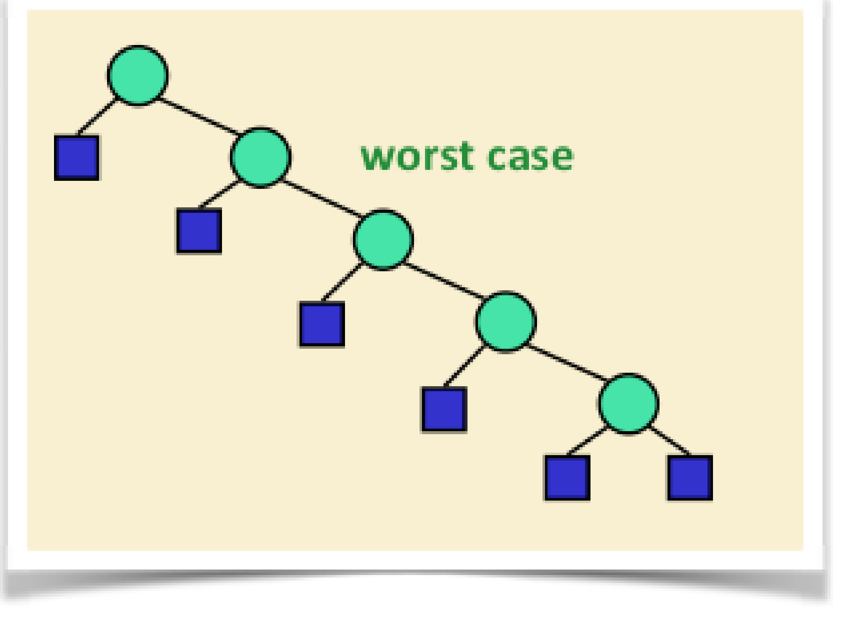
Binary Search Tree

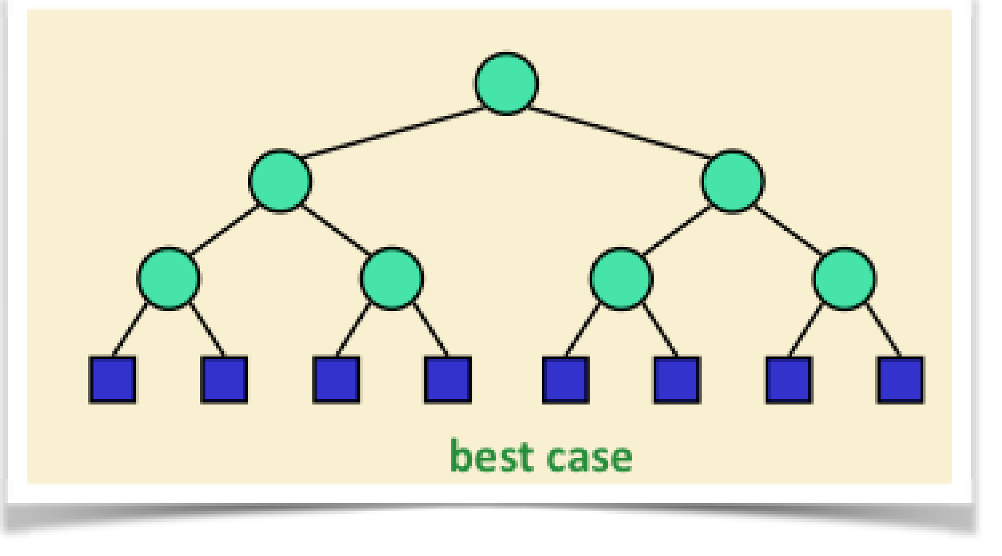
- For a binary search tree with n nodes
  - Search and insertion time is $O(\log n)$
- However, this is only true is the tree is "*balanced*"
- That is, the "height" of the tree is balanced

Binary Search Tree

- In the worst case, insertion and searching time becomes $O(n)$

- Because the height is $O(n)$





Watch the video

- https://youtu.be/ELROG7uppps?t=98

AVL Trees

- Definition
- An empty tree is height-balanced
- If T is non-empty binary tree with left and right sub-trees T_1 and T_2

  T is balanced if and only if

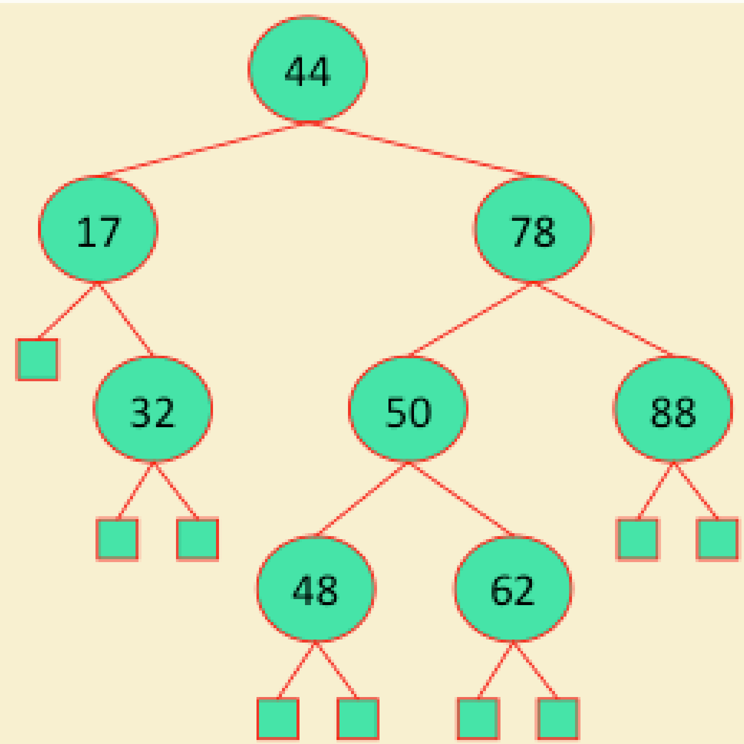
  - T_1 and T_2 are balanced, and
  - $|height(T_1) - height(T_2)| \leq 1$ // **balance factor**

Operations in an AVL Tree

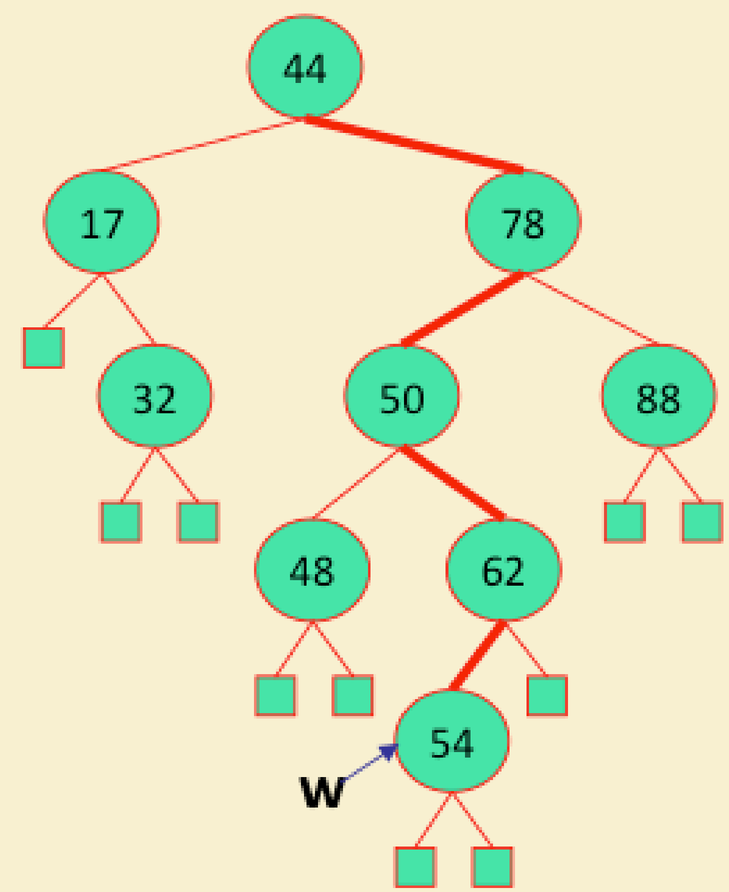
- The height of an AVL tree is $O(\log n)$
- Thus the **search** operation takes $O(\log n)$
  - Performed just like in a binary search tree since AVL tree is a binary search tree
- What we need to show is how to **insert** and **remove** in AVL trees while maintaining
  - the height balanced property
  - the binary search tree order

Insertion in an AVL Tree

- Starts as in a binary search tree
- Always done by expanding an external node



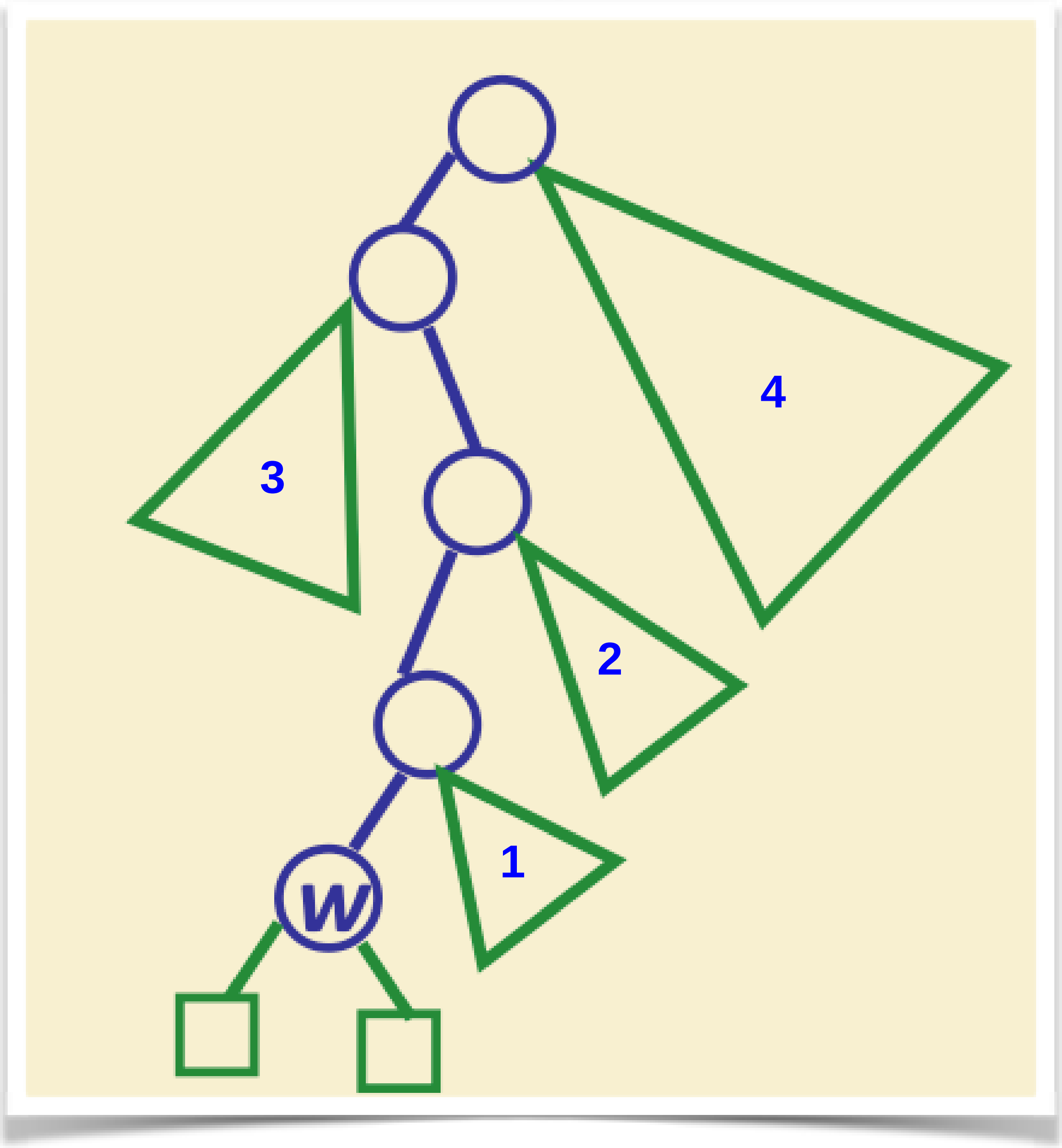
before inserting 54



after insertion

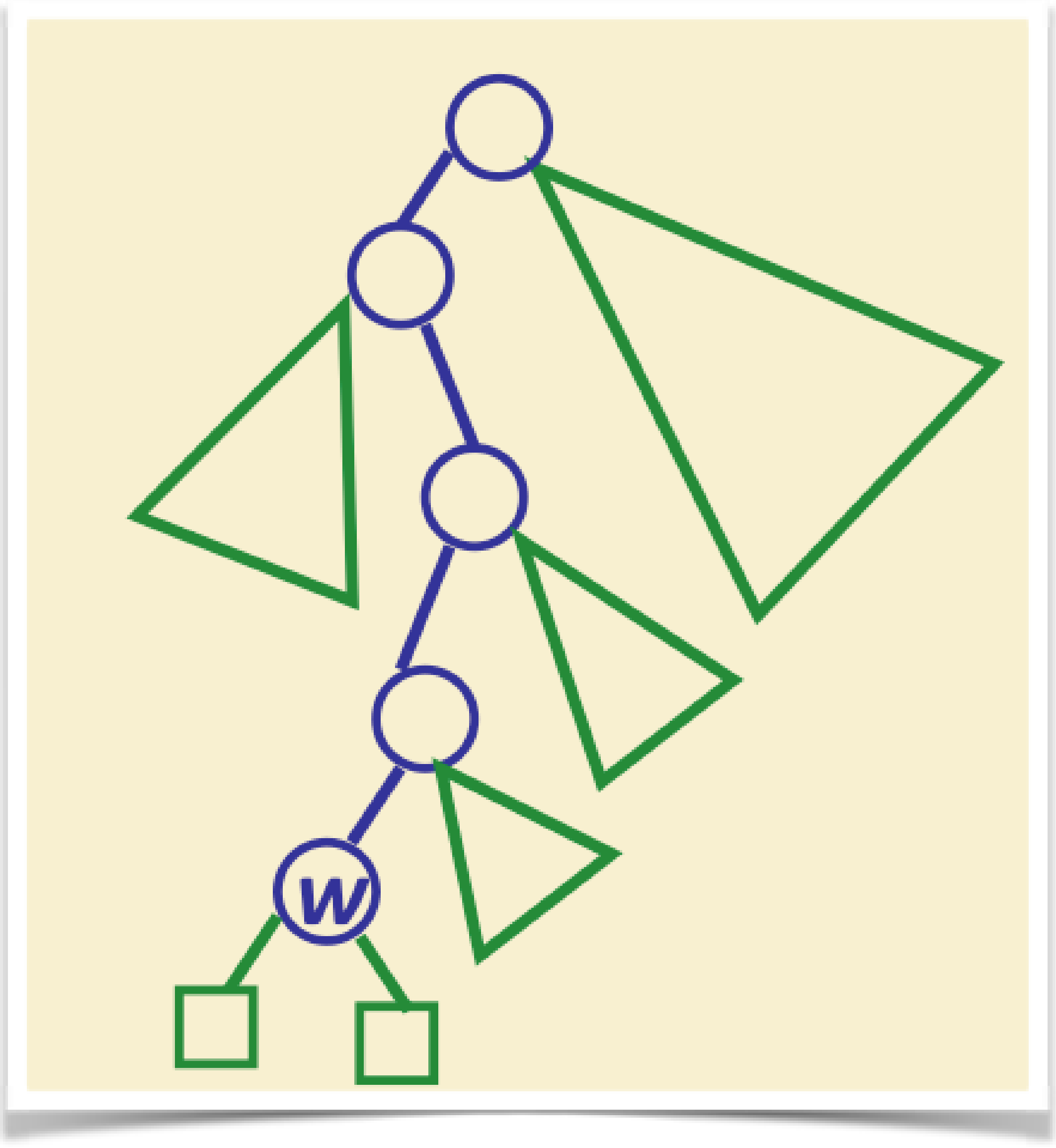
Restructuring

- Tip: after insertion of **w**, heights could change (increase) only for the ancestors of **w**
- Thus **only ancestors of w could be unbalanced**
- Search up the tree from w checking and correcting any unbalanced node



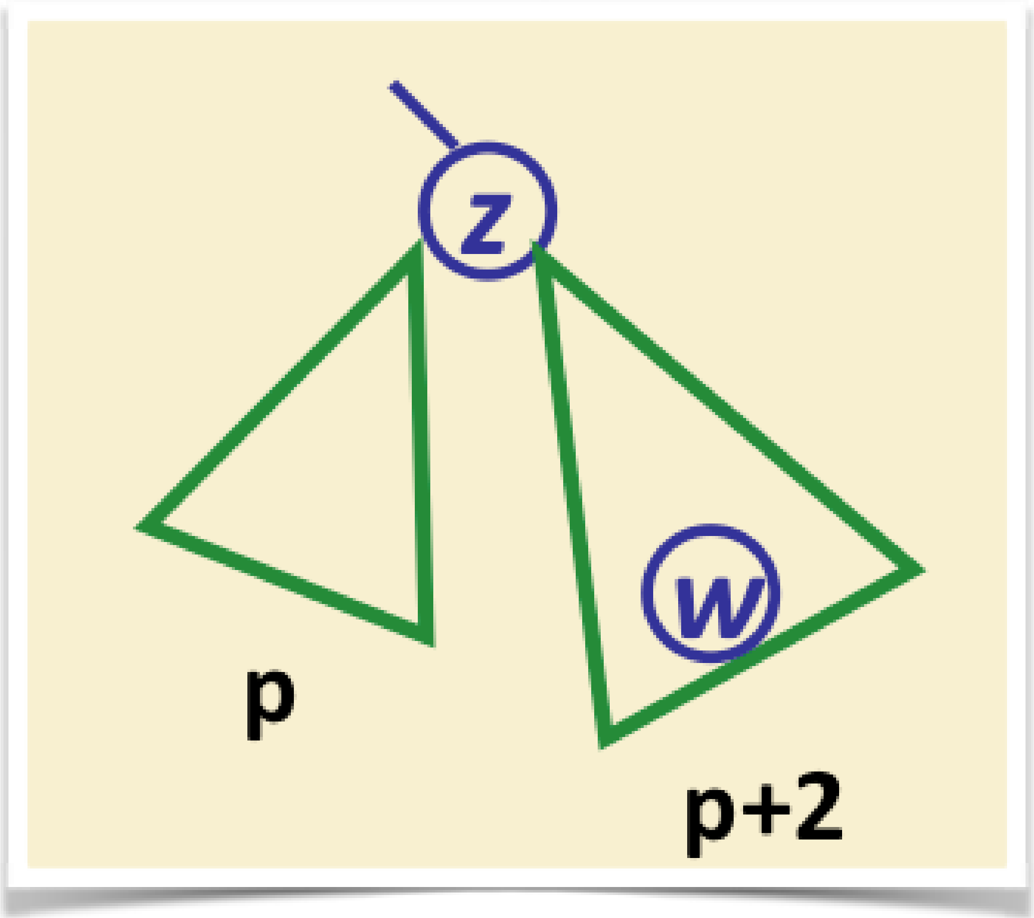
Restructuring

- Follow the path from **w** to the root



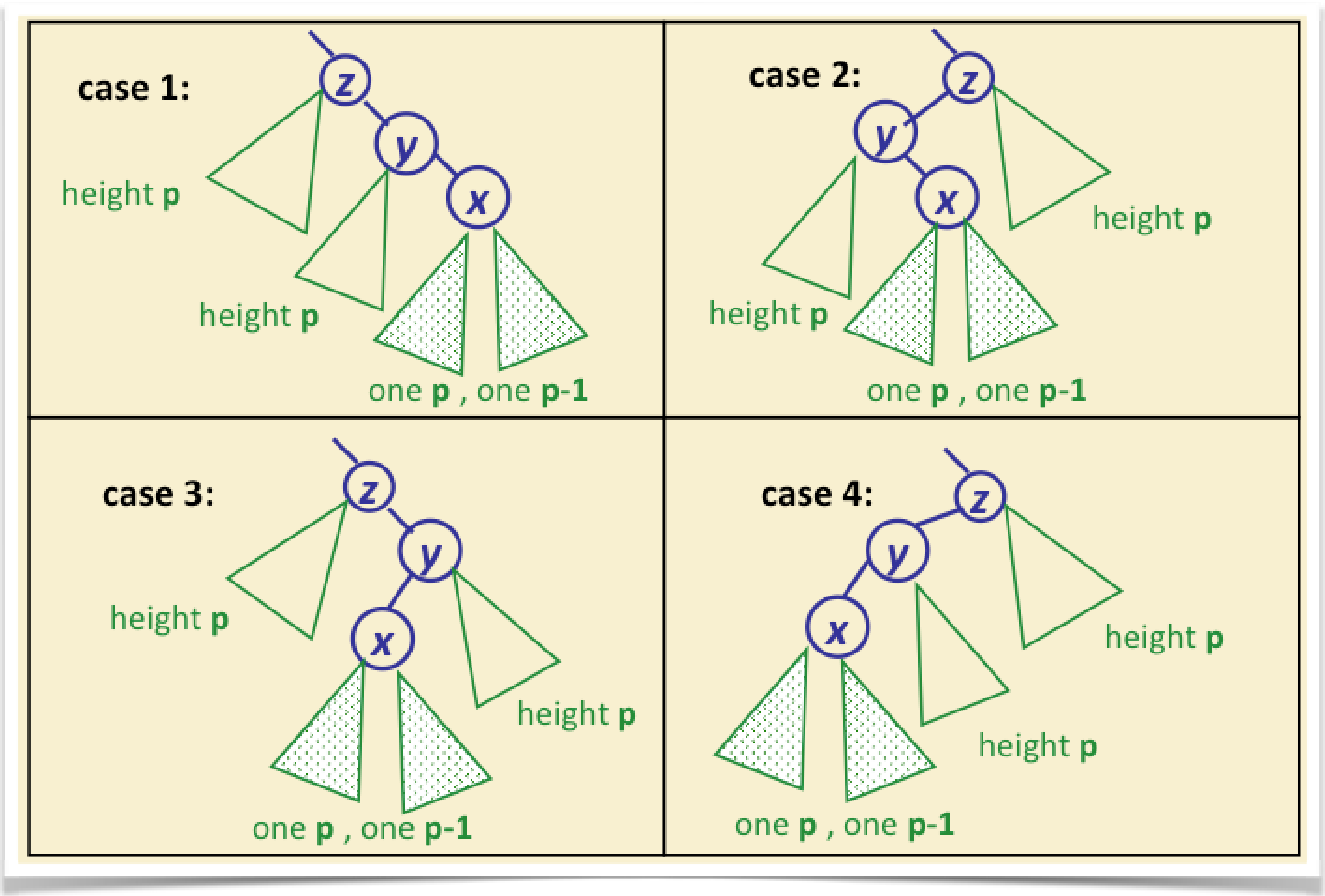
Restructuring

- Suppose the first unbalanced node is at position **z**
- This means that height difference between the left and the right subtree of **z** is more than 1
- In fact, it is **exactly** 2
  - ➢ tree was balanced before insertion
  - ➢ each insertion can change height only by a factor of 1
- **w** is in the higher subtree



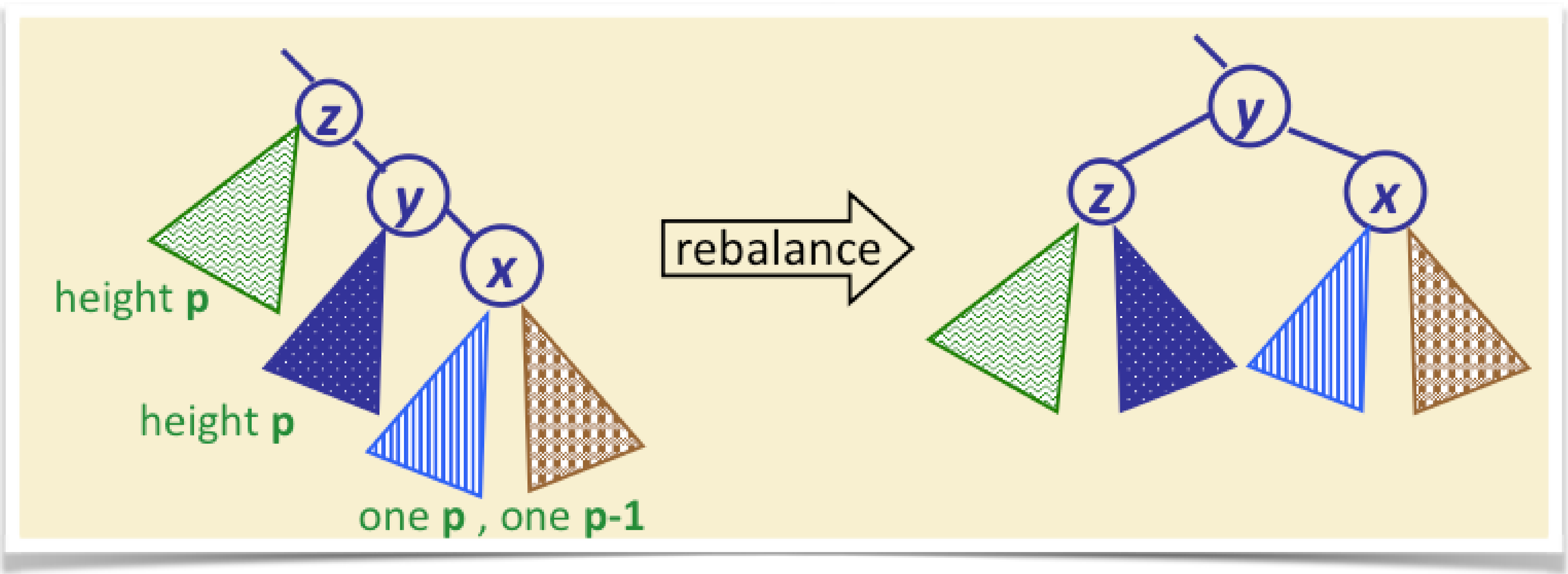
Restructuring

- Even More Complete Picture :)



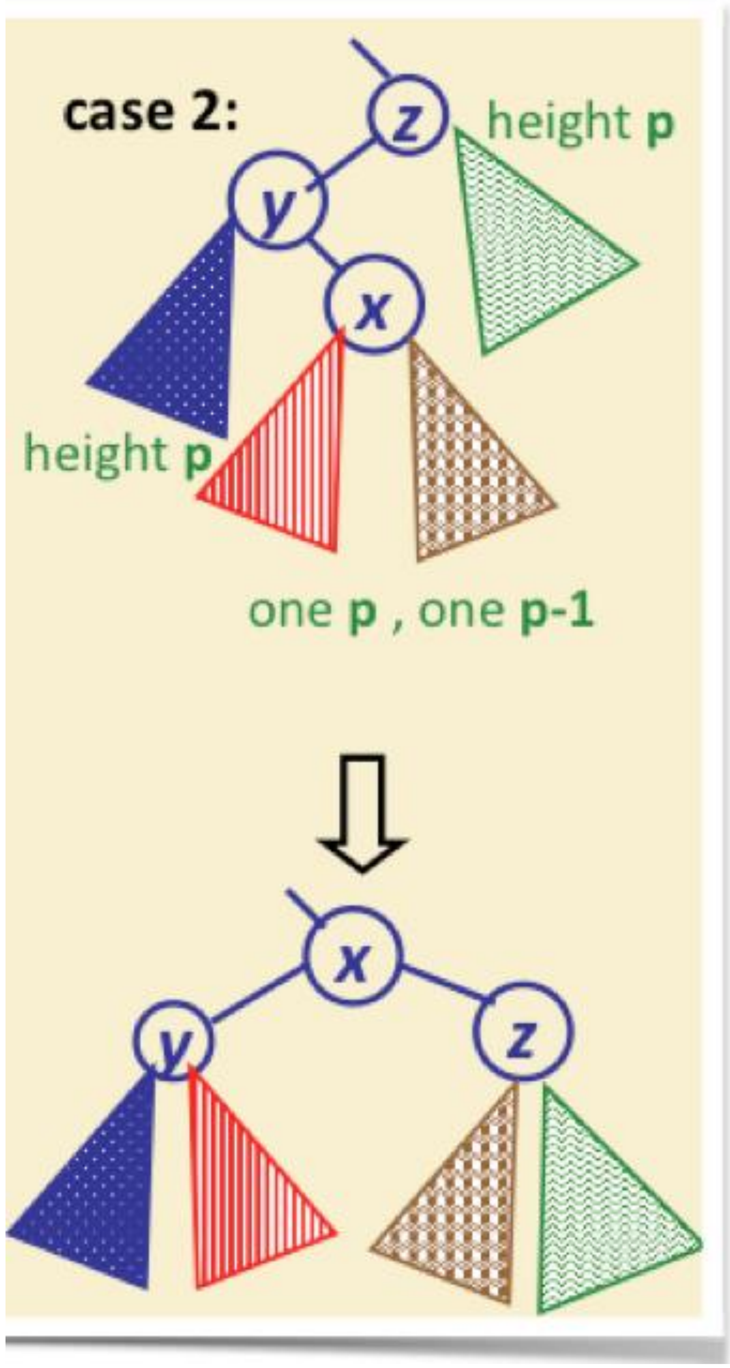
Restructuring

- Finally, let's restructure the tree
- Case 1:

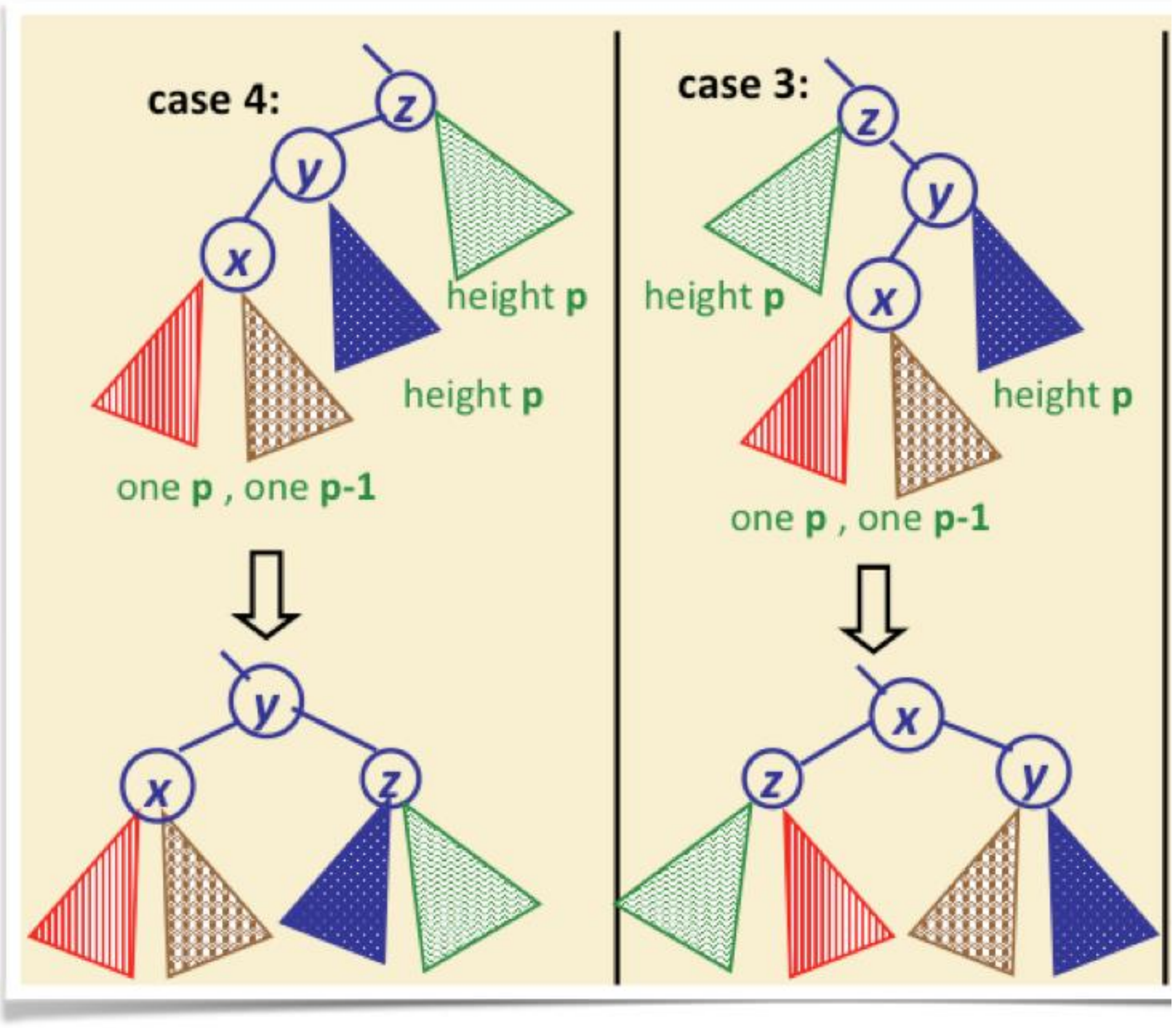


What's the height differences at nodes **x**, **y** and **z** after restructuring?

Restructuring



Restructuring



Trinode Restructuring

- Takes $O(logn)\ +\ O(1)$ – we will prove it!
- No loops, no recursive calls, constant number of comparisons, and changes in parent-child relationships
- Only 1 trinode restructuring is needed per insertion to restore the height balance property

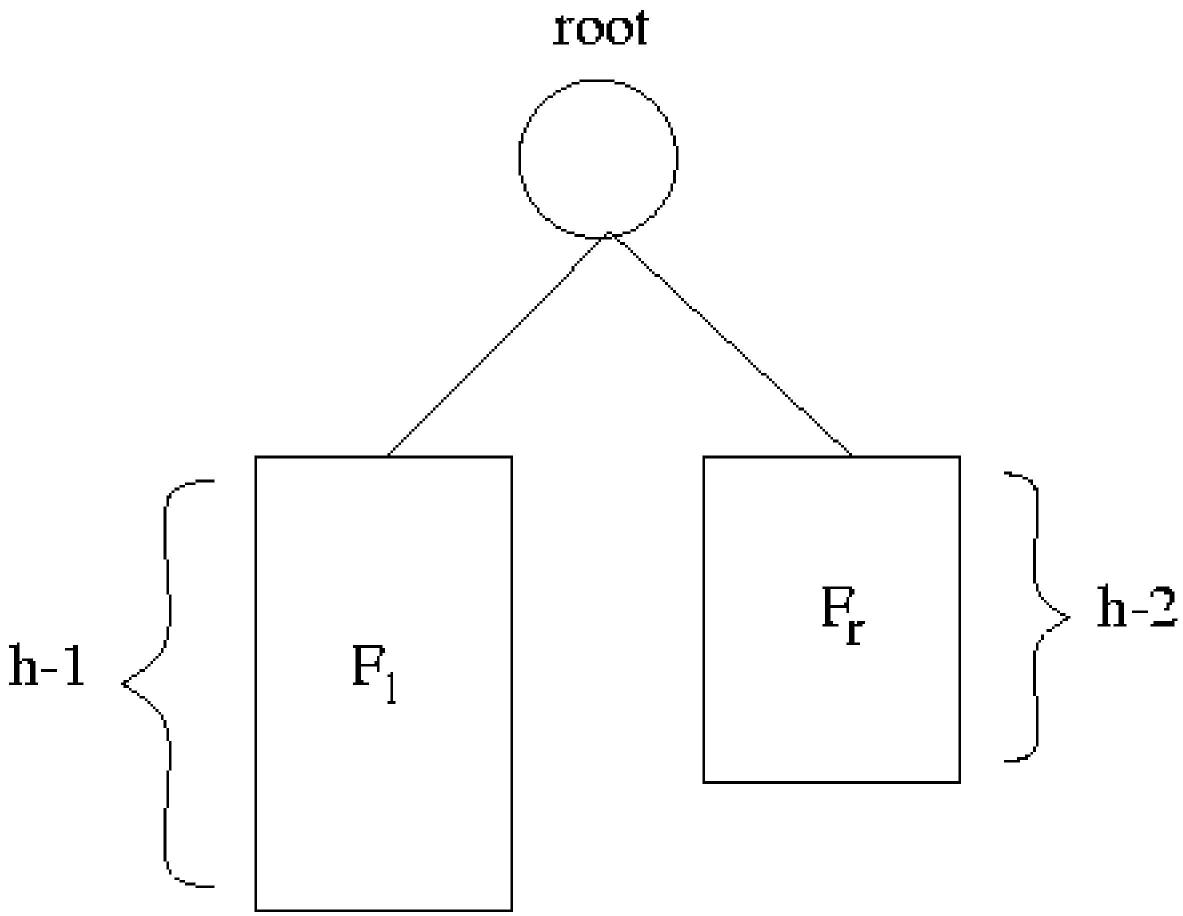
Deletion

- Deletion from an AVL tress may violate the height-balance property, too
- In this case, procedure for restructuring the tree to restore the balance is the same as in the case of insertion, with some changes
  - how to choose x, y, and z
  - repeated restructuring might be needed, max $O(\log n)$
- For further details, please read section 11.3.1 of your text-books

Proof

- Proposition: The height of an AVL tree storing $\boldsymbol{n}$ entries is $O(\log n)$

Let T be an AVL tree of height h. T can be visualized as



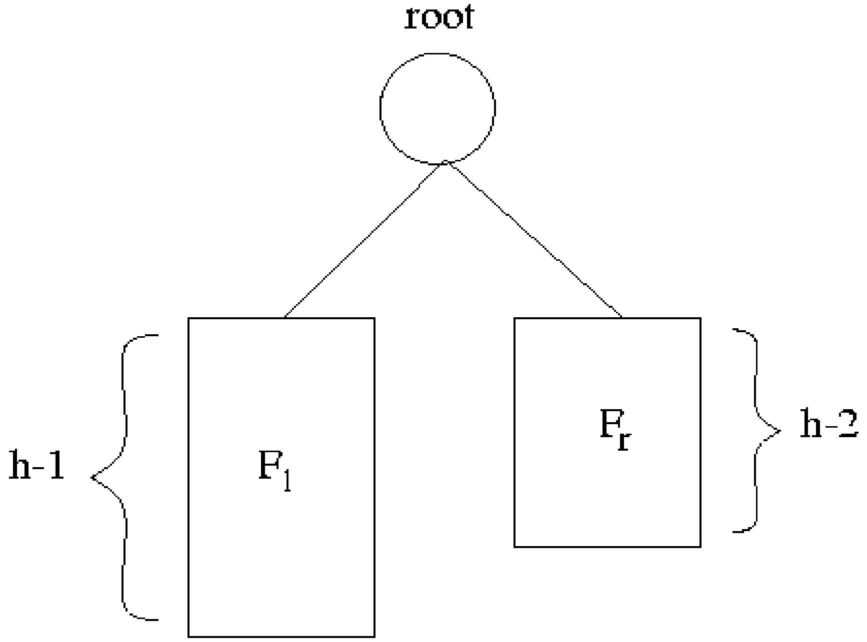
Proof

Let $n(h)$ be the minimum number of **internal nodes** in an AVL tree of height h

we know, $n(1) = 1 \; and \; n(2) = 2$

For $\boldsymbol{h >= 3}$

$$n(h) = 1 + n(h-1) + n(h-2)$$



Proof

$$\boldsymbol{n(h) = 1 + n(h-1) + n(h-2)}$$

Now that we know this, the rest is just algebra

According to the properties of Fibonacci progressions

$n(h) > n(h-1)$, so $n(h-1) > n(h-2)$

By replacing $n(h-1)$ with $n(h-2)$ and dropping the 1, we get

$$\boldsymbol{n(h) > 2n(h-2)}$$

Proof

$$\boldsymbol{n(h) > 2n(h-2)}$$

We can stop at this point. We have shown that $\boldsymbol{n(h)}$ at least **doubles** when $\boldsymbol{h}$ goes **up by 2**. This says that $\boldsymbol{n(h)}$ is **exponential** in $\boldsymbol{h}$, and hence $\boldsymbol{h}$ is **logarithmic** in $\boldsymbol{n}$

But let's continue

$$\boldsymbol{n(h) \ > \ 2\,(2n(h-4)) \ = \ 2^2 n(h-4)}$$

Proof

Thus,

$$For\ any\ i\ >\ 0,\qquad n(h)\ >\ 2^i\, n(h-2i)$$

Let's **get rid of $\boldsymbol{i}$ by expressing it in terms of $\boldsymbol{h}$**, but choose a value that results in making $h-2i$ either 1 or 2.

It is because we know the values for $n(1)$ and $n(2)$

That is, let

$$i\ =\ h/2-1$$

Proof

Thus,

$$For\ any\ i\ >\ 0, n(h)\ >\ 2^i\, n(h-2i)$$

Let's **get rid of *i* by expressing it in terms of *h***, but choose a value that results in making $h-2i$ either 1 or 2.

It is because we know the values for $n(1)$ and $n(2)$

That is, let

$$i\ =\ h/2-1$$

$$n(h) > 2^{h/2-1} n(h-2i) = 2^{h/2-1}$$

Proof

$$n(h) > 2^{h/2-1} n(h-2i) = 2^{h/2-1}$$

By taking logarithmic of both sides

$$\log(n(h)) > (h/2) - 1 \text{ or}$$

$$h < 2\log(n(h)) + 2$$

Or

$$h < C * \log(n)$$

Red & Black Trees

Red-Black Tree

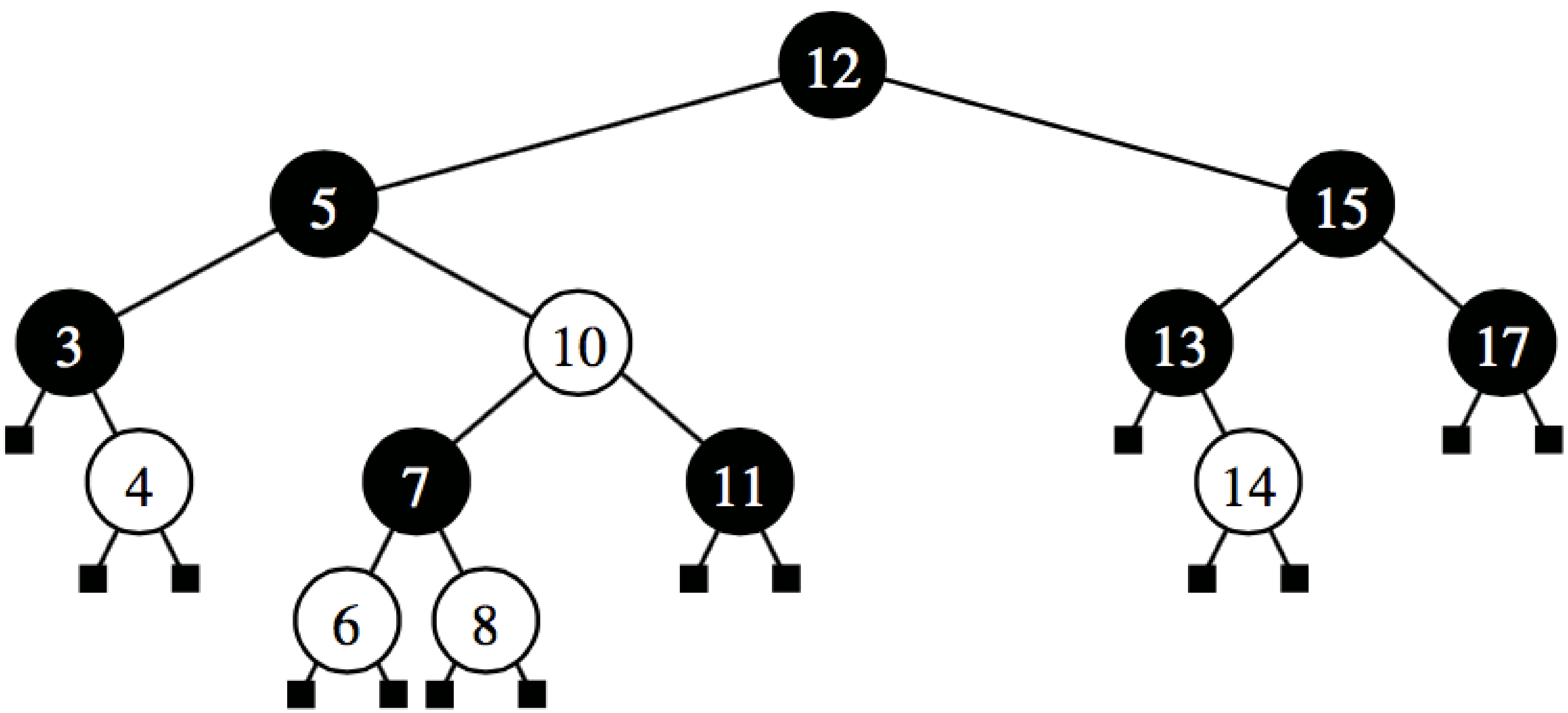
- How is the tree kept balanced?
- During insertions and deletions, it is made sure that certain **Properties** of the tree are not violated.
- Properties:
  - The nodes are colored
  - Arrangement of these colors

Red-Black Trees

- **Colored Nodes:** nodes can be colored either red or black to satisfy the following conditions:
- **Red-Black Properties:**

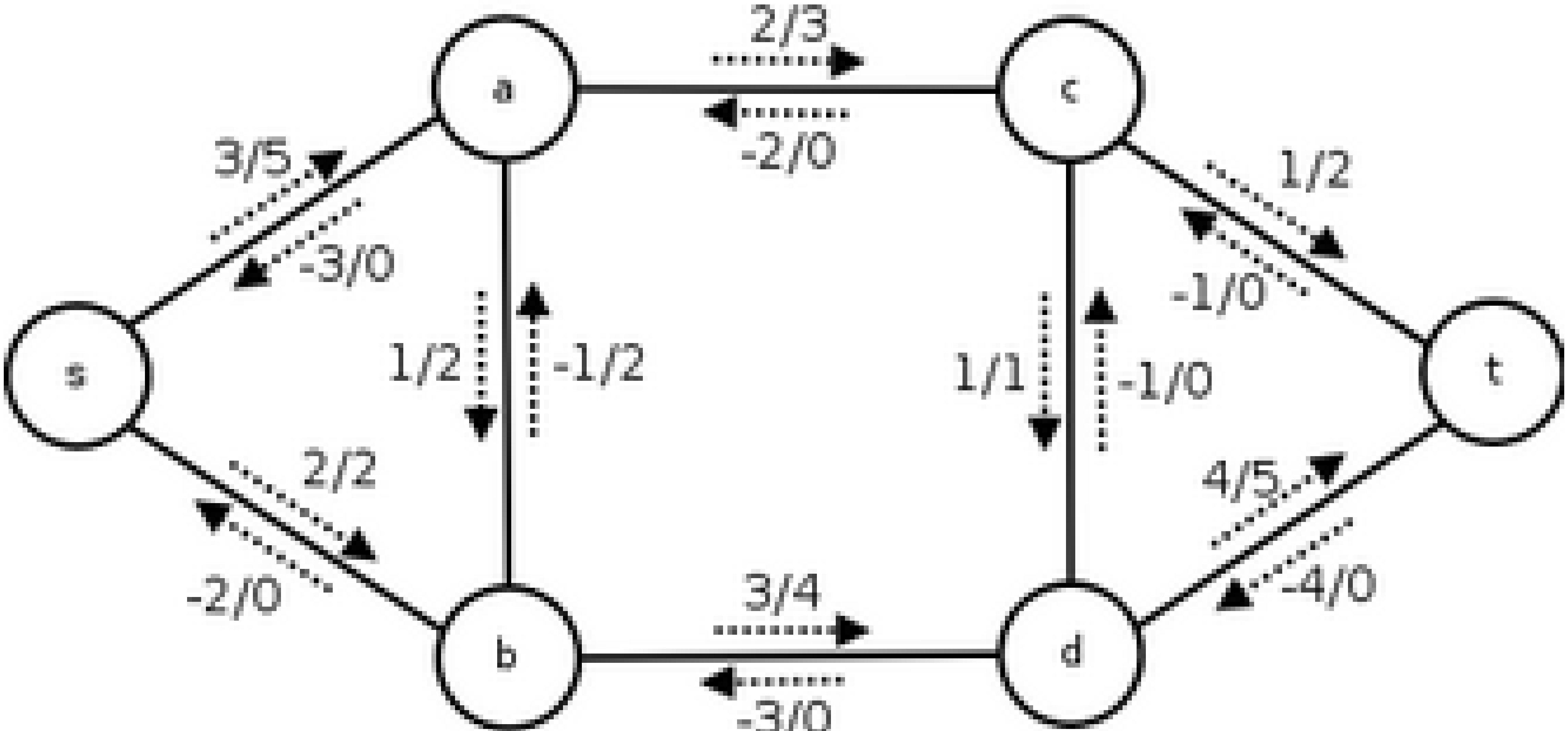
1. **Root Property:** The root is black
2. **External Property:** Every external node is black
3. **Red Property:** The children of a red node are black
4. **Black Property:** Every path from the root-to-frontier contains exactly the same number of black internal nodes (**black depth**)

Red-Black Tree



An example of a red-black tree, with “red” nodes drawn in white. The common black depth for this tree is 3.

Graphs: flow networks



s
a
b
c
d
t
3/5
-3/0
2/2
-2/0
2/3
-2/0
1/2
-1/2
3/4
-3/0
1/1
-1/0
1/2
-1/0
4/5
-4/0

Flow network – <u>directed</u> graph with edges that have 2 values: **capacity** and **flow**.

- **Capacity** can be considered as *bandwidth* (bps, trucks/hour, current, …),

- **Flow** can be considered as *actual load* (e.g. your flat's electrical capacity is 16A, and now you are consuming 2A).

- There are also 2 special vertices**:**

  - **source** (in-degree = 0) (router).

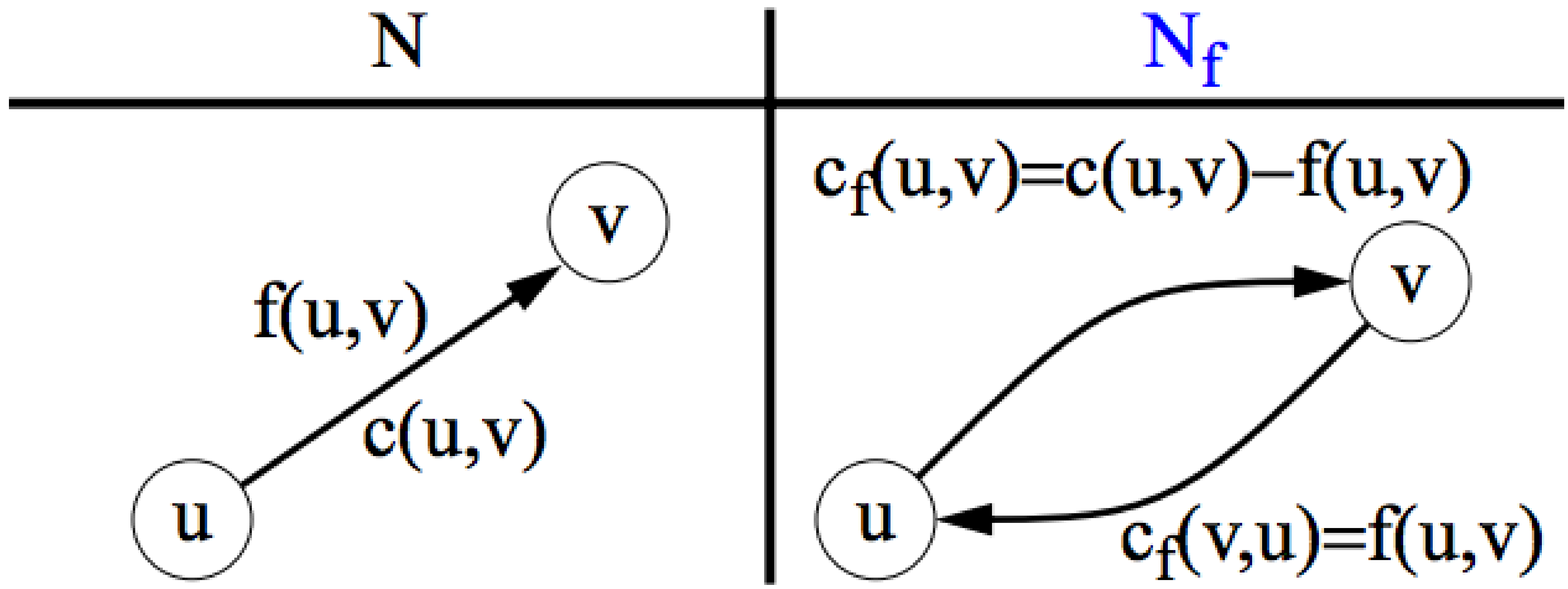
  - **sink (**out-degree = 0**)** traffic your laptop (torrents, youtube, updates) is consuming.

Capacity != Max Speed

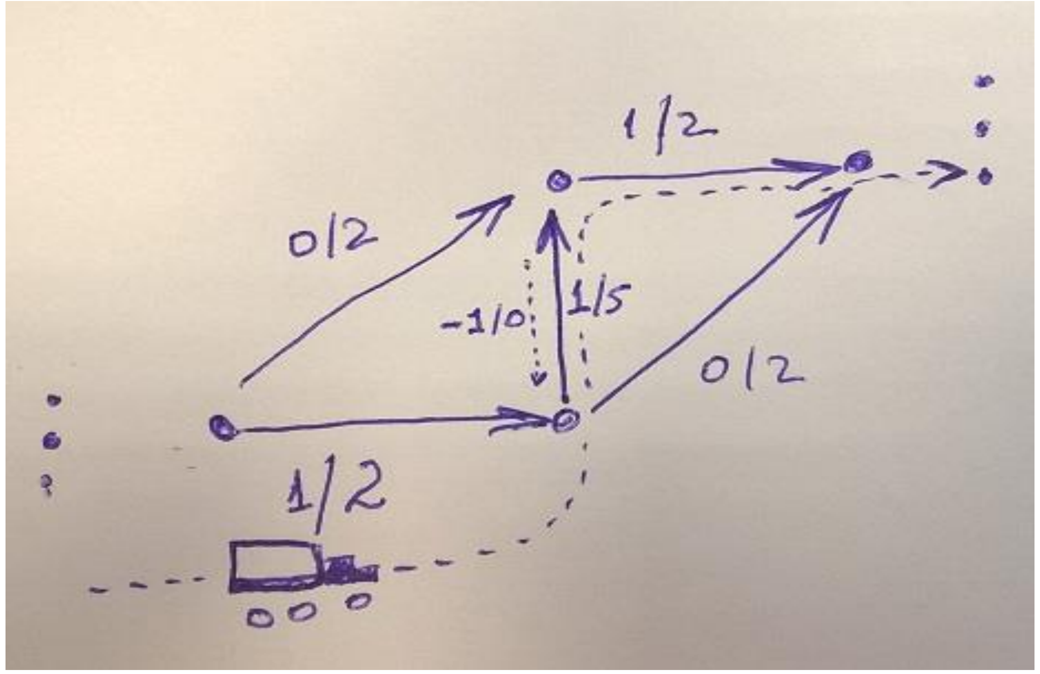
- In terms of internet “speed” of unit can be considered as ping, whereas “capacity” is bandwidth
  - Site can be “loading slow” because of both
  - These 2 roads can have **same CAPACITY** with **different speed** limit:

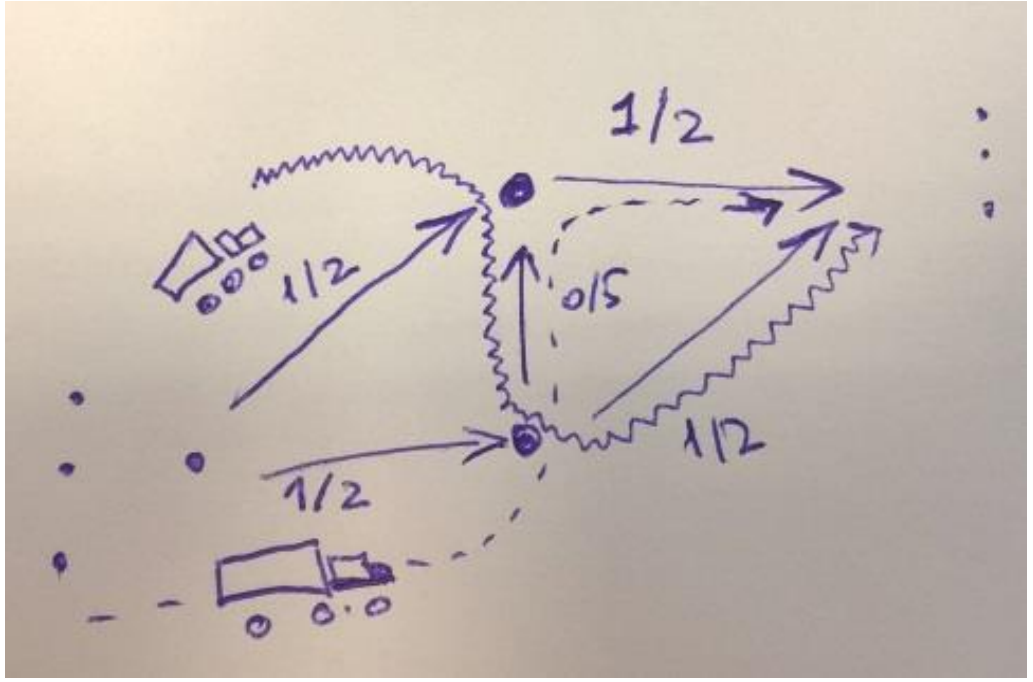
Residual Network

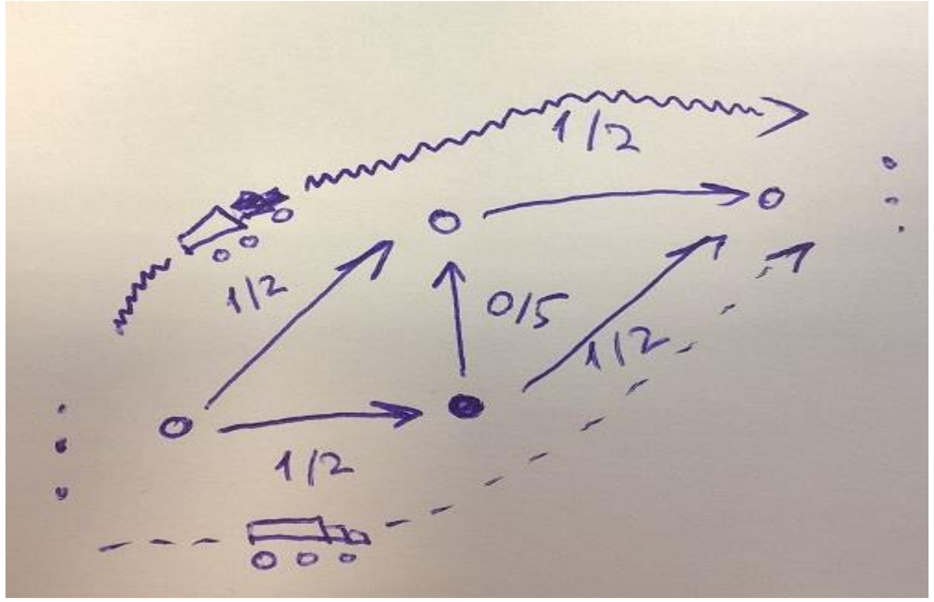
- Residual Network $N_f = (V, Ef, cf, s, t)$



“Fake” edges







Ford-Fulkerson

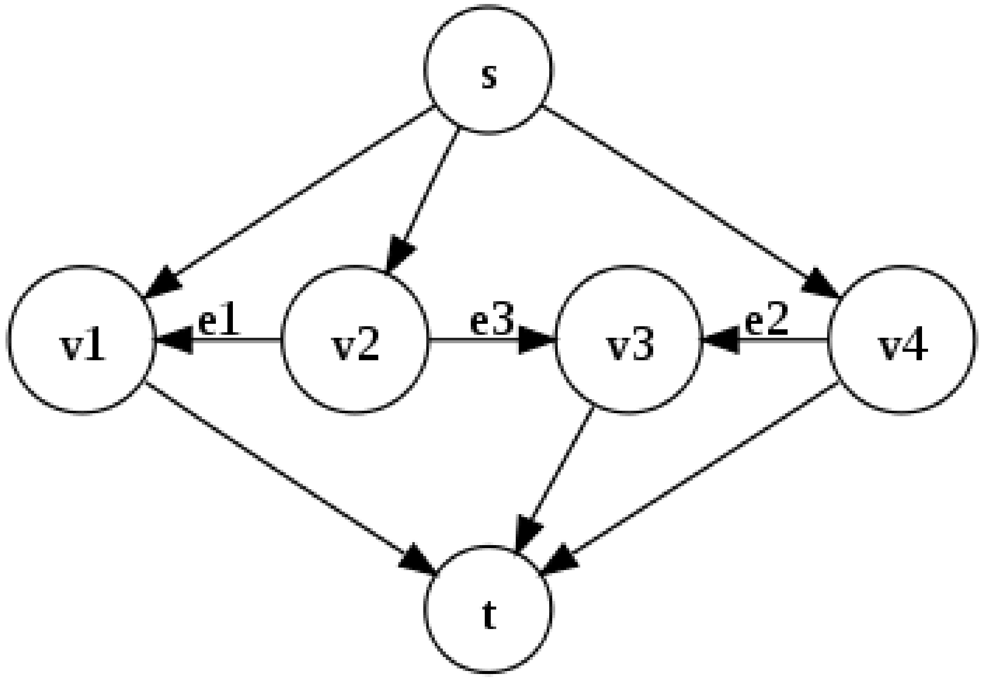
1. Initial **residual network** **c**(u, v) **== adjacency matrix** (if no edge: c = f = 0)
2. Initial **flow** **f**(u, v) == 0
3. LOOP:
    1. *Find **any** augmenting path from **s** to **t** on residual network c(u,v) c(u,v)>0, even if it is a backward edge*
    2. **<u>if none – break</u>**
    3. Find a **bottleneck** (smallest capacity **N** in a path)
    4. For each edge in path “send **N** trucks this way”:
        1. **f’**(u,v) = **f**(u,v) – **N**
        2. **f’**(v,u) = **f**(v,u) + **N**
        3. **c = c – f**
4. return **inCut(t)**

If in **3.a** we search for a **shortest** path, then we get **Edmonds-Karp/Dinic** algorithm

Non-terminating example

https://en.wikipedia.org/wiki/Ford%E2%80%93Fulkerson_algorithm#Non-terminating_example

Step	Augmenting path	Sent flow	Residual capacities		
			e_1	e_2	e_3
0			$r^0 = 1$	r	1
1	$\{s, v_2, v_3, t\}$	1	r^0	r^1	0
2	p_1	r^1	r^2	0	r^1
3	p_2	r^1	r^2	r^1	0
4	p_1	r^2	0	r^3	r^2
5	p_3	r^2	r^2	r^3	0



Graphs: MST, TopSort, Shortest

Prim vs Kruskal

- Attaches **cheapest adjacent** edges
- Works for **connected** graphs
- **Exploratory**!

- Attaches **cheapest** of all which **doesn’t create a cycle**
- Works for **any graphs**
- Requires **full information**

TopSort

- Find **one** “lonely” vertex at the end of the world. No one from the remaining graph depends on this vertex. Attach this vertex to **the left of the list**.

- Remove this vertex (and all incident edges) from the graph.

- Repeat 1+2 until graph is empty OR we find a cycle (no lonely edges).

- * you can also do this in opposite direction :)

Shortest: Dijkstra

- **We know <u>something</u> about the world (initially – incident edge weights)**
  - "It will take *N* minutes to get from <u>source H</u> to X"
    - *In Dijkstra algorithm we pick shortest known N*
- **We discover <u>new fact</u> about the world**
  - "Y is in 5 minutes walk form X"
- **We <u>update our knowledge</u>**
  - "It will take *N+5* minutes to get from <u>source H</u> to Y"
- **Discover new facts, repeat**

Shortest: Floyd-Warshall

- **We know <u>something</u> about the world (initially – incident edge weights)**
  - “It will take N minutes to get from A to X”
  - “It will take M minutes to get from X to B”
- **We discover <u>new facts</u> about the world**
  - We have **X** in both facts!
    Thus, we can get from A to B via X!
- **We <u>update our knowledge</u>**
  - “It will take N+M minutes to get from A to B”
- **Discover new facts, repeat**