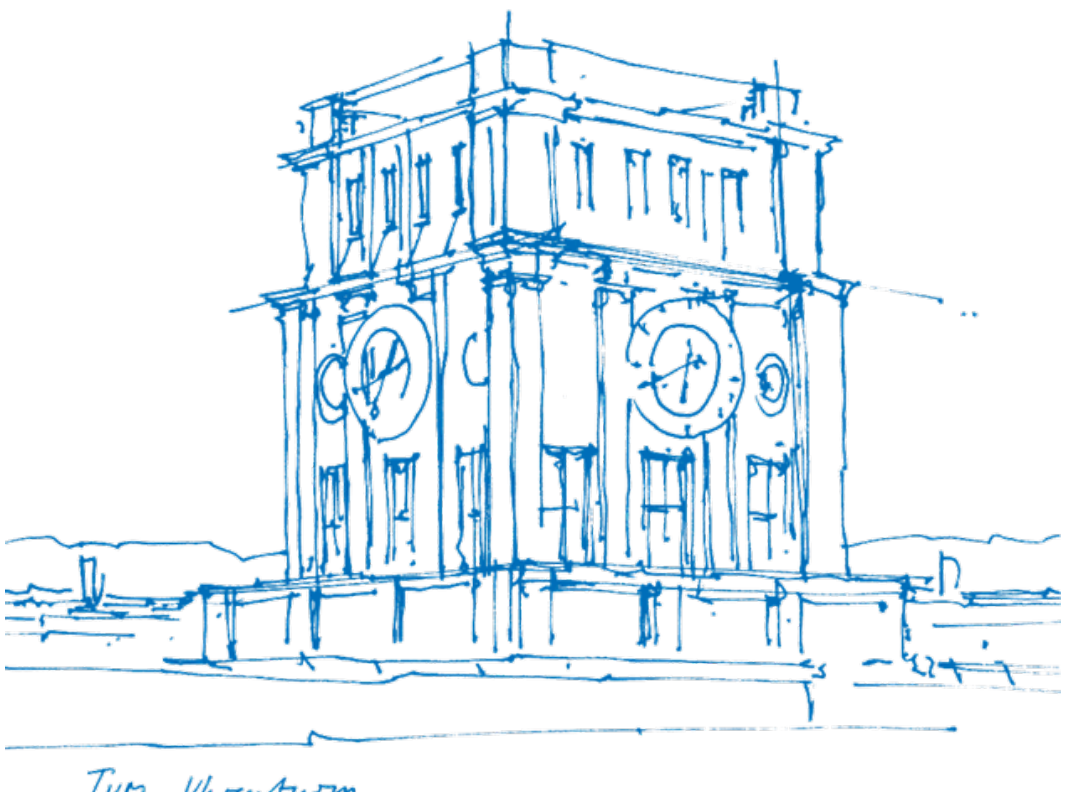
School of Computation, Information and Technology
Technical University of Munich

AACPP 2025

Week 4: Trees



Mateusz Gienieczko, **Mykola Morozov**

School of Computation, Information and Technology
Technical University of Munich

2025.05.27

Second round – survey

Third round

Third deadline – 03.06.2025, 10:00 AM.

TES – Treat Exchange System

We can compute what the account balances after all exchanges will be.

TES – Treat Exchange System

We can compute what the account balances after all exchanges will be.

We have “losers” and “gainers”.

TES – Treat Exchange System

We can compute what the account balances after all exchanges will be.

We have “losers” and “gainers”.

Clearly it makes no sense for gainers to ever send treats.

TES – Treat Exchange System

We can compute what the account balances after all exchanges will be.

We have “losers” and “gainers”.

Clearly it makes no sense for gainers to ever send treats.

Nor does it make sense for losers to ever receive treats.

TES – Treat Exchange System

We can compute what the account balances after all exchanges will be.

We have “losers” and “gainers”.

Clearly it makes no sense for gainers to ever send treats.

Nor does it make sense for losers to ever receive treats.

Transaction always goes gainer → loser.

This is a greedy property, provable by replacement.

TES – Treat Exchange System

Sum all losses (or all gains) for 50%.

To decide transactions, note that it also makes no sense to ever overpay.

Each loser will send exactly as many treats as its loss and vice versa for gainers.

TES – Treat Exchange System

Just take any loser and send treats to any gainers that are not satisfied yet.

Simplest way: put all gainers in one queue, losers in the other.

```
while loser = losers.pop
  while loser.loss > 0
    if loser.loss >= gainers.front.gain
      record(loser, gainers.front, gainers.front.gain)
      gainers.pop
    else
      record(loser, gainers.front, loser.loss)
      gainer.front.gain -= loser.loss
```

$\mathcal{O}(n)$ time and memory.

SLI – Sliding Cat Puzzle

Obvious brute force in $\mathcal{O}(n!)$, but no points for it this time.

Key observation – since the distance travelled does not matter, the concrete index of a slot does not matter, only the remaining slots on left and right.

SLI – Sliding Cat Puzzle

Obvious brute force in $\mathcal{O}(n!)$, but no points for it this time.

Key observation – since the distance travelled does not matter, the concrete index of a slot does not matter, only the remaining slots on left and right.

$\mathrm{DP}[l][r]$ – optimal cost when there are l (r) unvisited slots to the left (right).

We can slide to any l', r' where $l' + r' = l + r - 1$. There's $k = l + r$ such states.

$$\mathrm{DP}[l][r] = \min \begin{cases} \min_{0 \le i < l} \mathrm{DP}[i][k - i - 1] + \mathrm{left}[n - k] \\ \min_{0 \le i < r} \mathrm{DP}[k - i - 1][i] + \mathrm{right}[n - k] \end{cases}$$

Total time $\mathcal{O}(n^3)$, memory $\mathcal{O}(n^2)$.

SLI – Sliding Cat Puzzle

Optimise the above to fill the entire level for a given k in a two $\mathcal{O}(k)$ loops.

For left-slides go $0 \leq l \leq k$ and accumulate the minimum of $\mathrm{DP}[i][k-i-1]$.

Same for right-slides in the other direction.

Now $\mathcal{O}(n^2)$ time and memory.

SLI – Sliding Cat Puzzle

The key observation here is that our slides are constrained only by s.

If we have k unvisited slots and they are all to our right (resp. left), then we can perform any sequence of slides starting from a right-slide (resp. left-slide).

Let's ignore s for now.

SLI – Sliding Cat Puzzle

$\mathrm{DP}[k][L]$ is the best cost if we have already visited $k-1$ slots and now slide left. $\mathrm{DP}[k][R]$ analogously to the right.

The formula is very simple:

$$\mathrm{DP}[k][0] = \min(\mathrm{DP}[k+1][0], \mathrm{DP}[k+1][1]) + \mathrm{left}[k-1]$$
$$\mathrm{DP}[k][1] = \min(\mathrm{DP}[k+1][0], \mathrm{DP}[k+1][1]) + \mathrm{right}[k-1]$$

and computable in $\mathcal{O}(n)$.

SLI – Sliding Cat Puzzle

The constraint from s applies only at the start, restricting the number of initial left-slides to $s - 1$ and right-slides to $n - s$.

To get the solution where we initially slide x times we take:

$$\mathrm{DP}[x+2][R] + \sum_{0 \le i \le x} \mathrm{left}[i]$$

$$\mathrm{DP}[x+2][L] + \sum_{0 \le i \le x} \mathrm{right}[i]$$

We can find the optimum for all possible x values in $\mathcal{O}(n)$.

SLI – Sliding Cat Puzzle

From the DP it's easy to recover the sequence of slides, e.g. $n = 8, s = 5$

$$LLRLRRR$$

A possible strategy is to group the slides in the same direction and slide to the outermost slot.

$$\mathrm{LL}, \mathrm{R}, \mathrm{L}, \mathrm{RRR}$$

$$5, 2, 1, 8, 3, 4, 6, 7$$

One can put all slots except s onto a double-ended queue and pop accordingly.

Recall the plan

- Greedy and dynamic programming (DP)
- **Trees** ← *we are here*
- Graphs
- Ways to turn graphs into trees (DFS, BFS, Dijkstra, MST)
- Ways to run DP on graphs (Toposort)
- Advanced graph algorithms (Matchings, flows)
- Binary Search Trees
- Number theory
- String algorithms (KMP, tries, suffix tables)
- Some problems can't* even be solved efficiently (NP-completeness)

Trees

Foundational data structure.

Many equivalent definitions:

- n nodes, each except for one has a single parent node;
- connected graph on n vertices with $n - 1$ edges;
- recursive: empty tree or a node connected to other trees;
- graph on n vertices where there is exactly one path between each pair of vertices.

Trees – Root, Parent, Child, Depth

Mathematically trees don't have to be rooted, but as a data structure they are.

Root – unique distinguished vertex.

Parent – unique node "up"; root has none.

Child – direct neighbour other than the parent.

Depth – root has depth 0, children of a node with depth d have depth $d + 1$.

Leaves – nodes with no children.

Trees – Ancestor, Descendant

Transitive closure of the parent relation is called *ancestor*.

Of the child relation – *descendant*.

Trees – Forests

Many separate trees are called a *forest*.

Trees – Alternative Definition

Some people define these to be trees:

Trees – Alternative Definition

Some people define these to be trees:

Trees – Alternative Definition

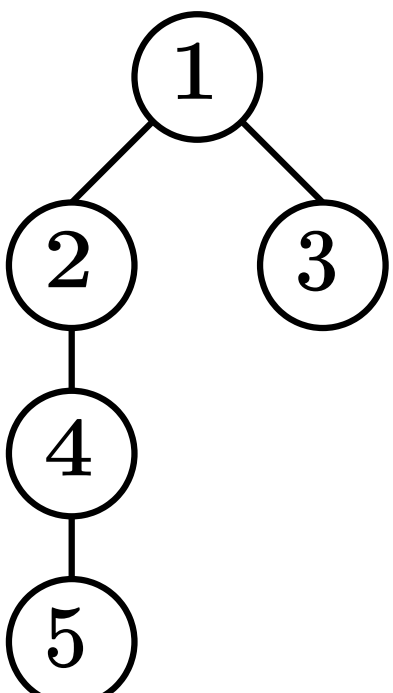
Some people define these to be trees:

They are upside down (leaves at the top) and harder to put into the computer.

Trees – input representations

As a graph – list of edges.

```
5 4
1 2
1 3
2 4
4 5
```

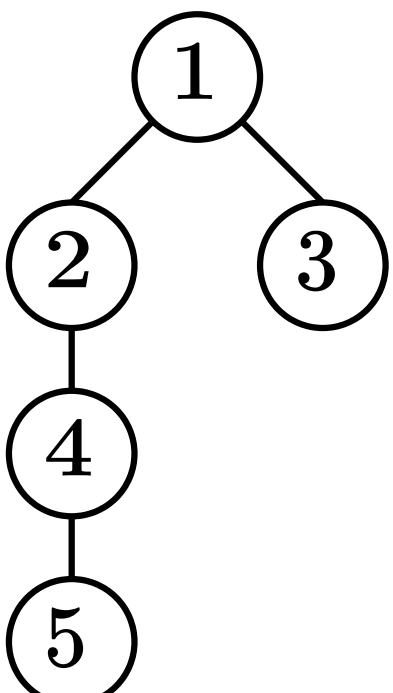


Trees – input representations

As the parent relation.

5

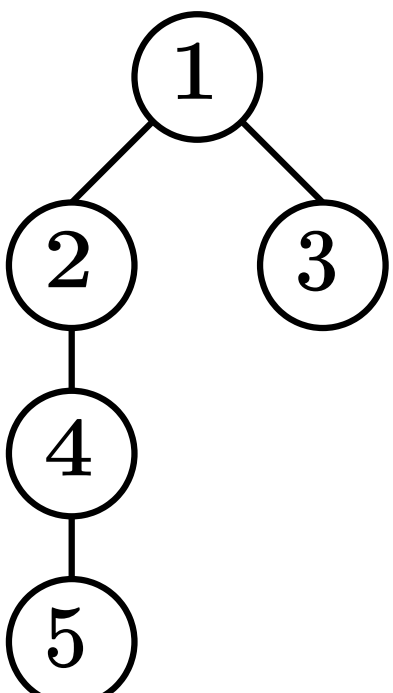
1 1 2 4



Trees – input representations

As lists of children.

```
5
2 2 3
1 4
0
1 5
0
```



Trees – in-memory representations

Parent relation.

One array $P[n]$ where $P[i]$ is the number of the parent node of i.

The value for the root does not matter. Commonly $P[\text{root}] = \text{root}$.

Very memory-efficient.

Trees – in-memory representations

Lists of lists.

Same representation as used for general graphs.

C[v] is a list of all children of v.

In C++ a `vector<vector<int>>`.

In Rust a `Vec<Vec<u32>>`.

More memory, but often much more useful for algorithms.

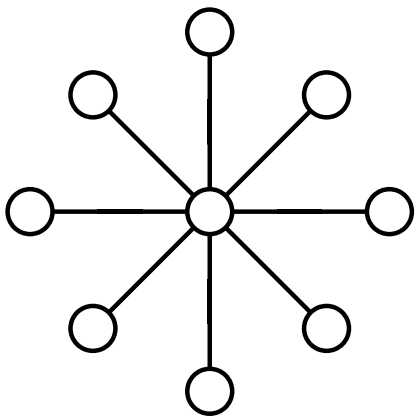
Trees – in-memory representations

Special case – binary trees.

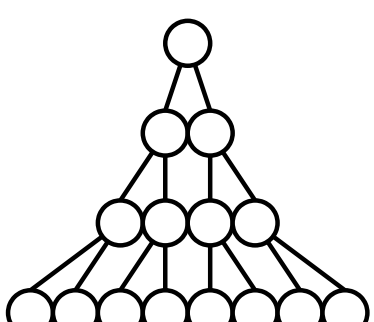
Two arrays, $L[n]$ and $R[n]$, giving the number of the left and right child.

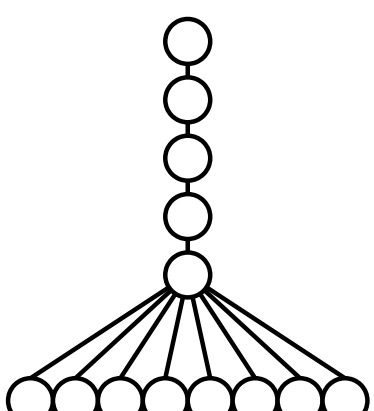
If none, some special value like -1.

Trees – special tree shapes



Path, star, binary, broom.





Trees – DFS

Depth-first search.

Start from the root, process the node, then recurse down to its child.

```
fn DFS(v)
  pre_process(v)
  for child in C[v]
    DFS(child)
  post_process(v)
```

Trees – DFS

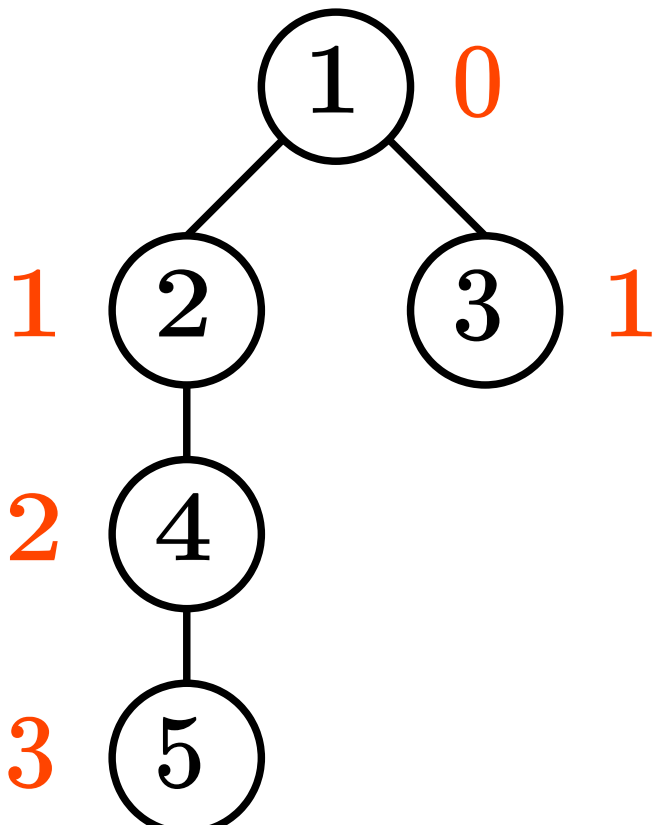
This implicitly uses the thread's stack to store the DFS state.

On our platform it doesn't matter (there is no stack limit), but in practice stack is often limited (e.g. 1MiB) and so using an explicit stack is required for deep trees.

Trees – DFS example

Example usage – depth calculation.

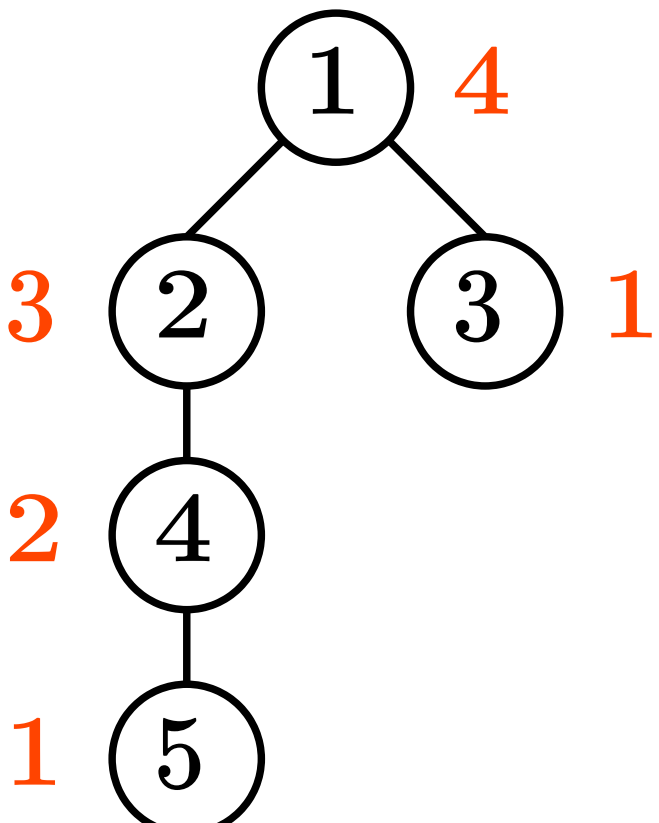
```
fn calculate_depth(v, depth = 0)
  D[v] = depth
  for child in C[v]
    calculate_depth(child, depth +
1)
```



Trees – DFS example

Example usage – subtree sizes

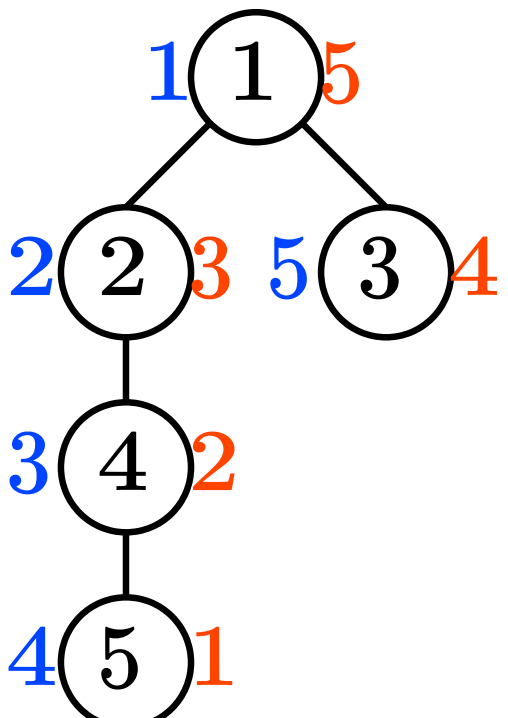
```
fn calculate_size(v)
  S[v] = 1
  for child in C[v]
    calculate_size(child)
    S[v] += S[child]
```



Trees – DFS preorder and postorder

Preorder – give a number when entering.

Postorder – give a number when exiting.



Trees – BFS

Breadth-first search.

Visiting the levels of the tree in order.

All nodes of depth d processed before those of $d + 1$.

```
fn BFS()
  queue = Queue::new()
  queue.push(root)
  while v = queue.pop()
    process(v)
    for child in C[v]
      queue.push(child)
```

Trees – Lowest Common Ancestor

Each two nodes have one path that joins them.

This path always goes through the *Lowest Common Ancestor* (LCA).

Trees – Lowest Common Ancestor

Each two nodes have one path that joins them.

This path always goes through the *Lowest Common Ancestor* (LCA).

Data structure for LCA: log-jumps.

1. Compute depth.
2. Compute $\text{jump}[v][k]$ as the ancestor 2^k higher in the tree. $\text{jump}[v][0] = \text{parent}[v]$.
3. LCA of two vertices can be done by iterating largest possible jump until depth equalises.

Trees – Centroid Decomposition

Method for running divide-and-conquer algorithms on trees.

Idea – take a vertex and split the problem across its children.

Subproblems happen entirely in the subtrees, or involve the pivot vertex.

Effective if the pivot splits into “small” subtrees.

Trees – Centroid Decomposition

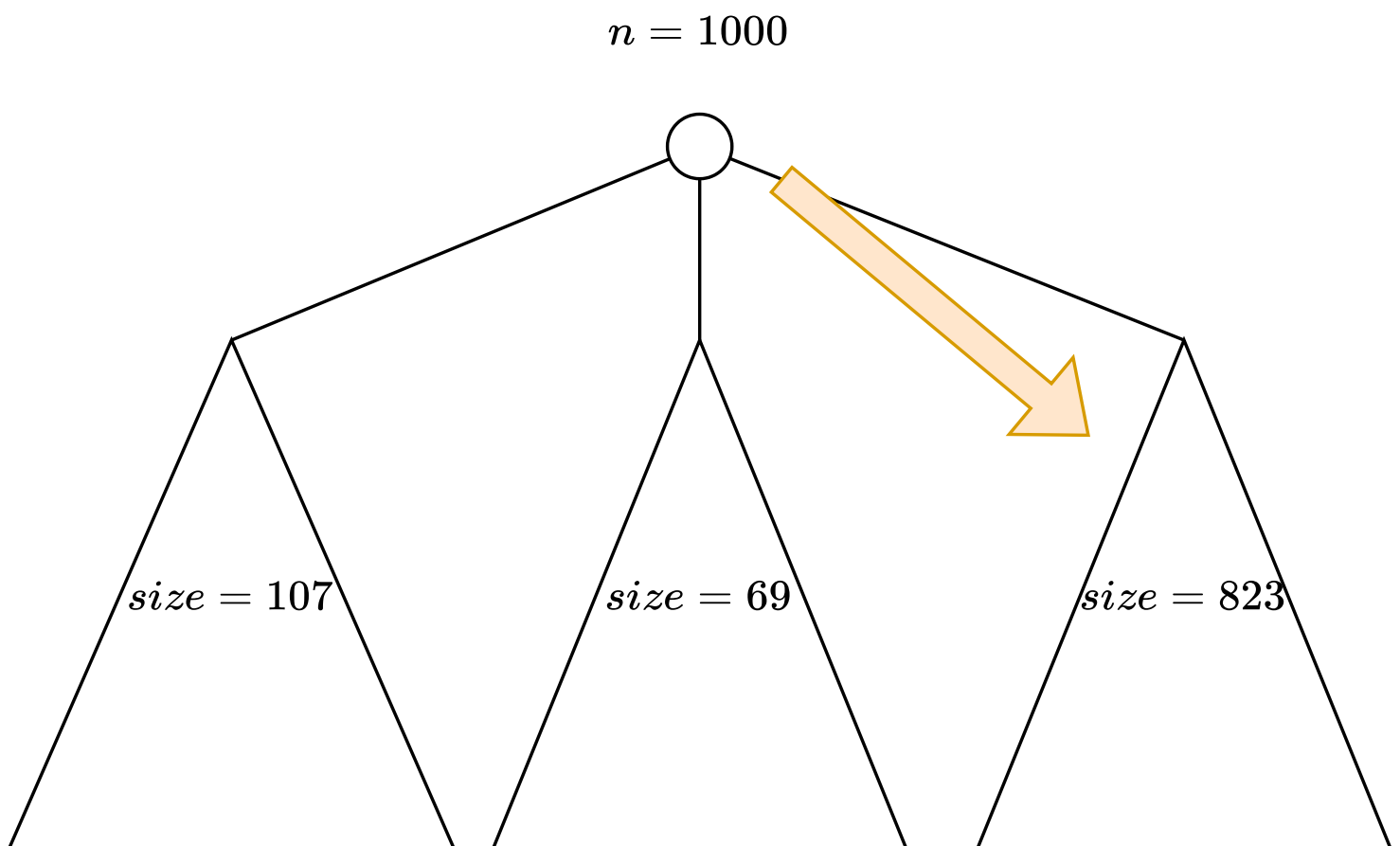
Centroid of a tree on n vertices is a vertex whose subtrees rooted at children are all at most size $\frac{n}{2}$.

Provably there is exactly one or two such vertices.

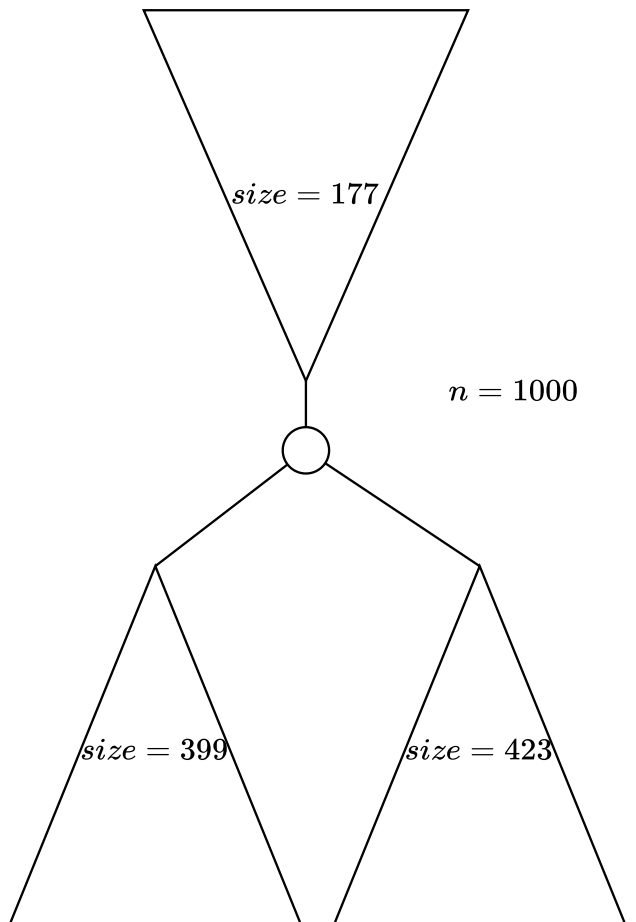
Algorithm to find – root arbitrarily, then compute subtree sizes.

Iteratively move the root to the heaviest subtree until we reach the centroid.

Trees – Centroid Decomposition



Trees – Centroid Decomposition



Trees – Find-Union

Disjoint sets structure.

The universe is numbers, e.g. from 0 to $n - 1$.

We want to be able to merge sets (Union) and ask if two items are in the same set (Find).

Trees – Find-Union

Naive solution – keep all the sets in lists, merge lists on Union.

When Unioning we can update an ID for each element identifying which set they're in and answer Find in $\mathcal{O}(1)$.

Better idea – always Union a smaller set to a larger set.

Trees – Find-Union

Amortised time – single operation may be costly, but it prevents work being done later.

Here – each element can only be moved to a different set at most $\mathcal{O}(\log n)$ times.

So even though a single Union can take $\mathcal{O}(n)$, all Unions together won't exceed $\mathcal{O}(n \log n)$.

Trees – Find-Union

Better idea – *forest of representatives*.

Each set is a tree, given by each element holding a parent.

Find(x) – find root of x by traversing parents.

Union(x, y) – find root of y, set root of x as its parent.

If we keep tree sizes and plug smaller to larger we get amortised $\mathcal{O}(\log n)$ Find.

Trees – Find-Union

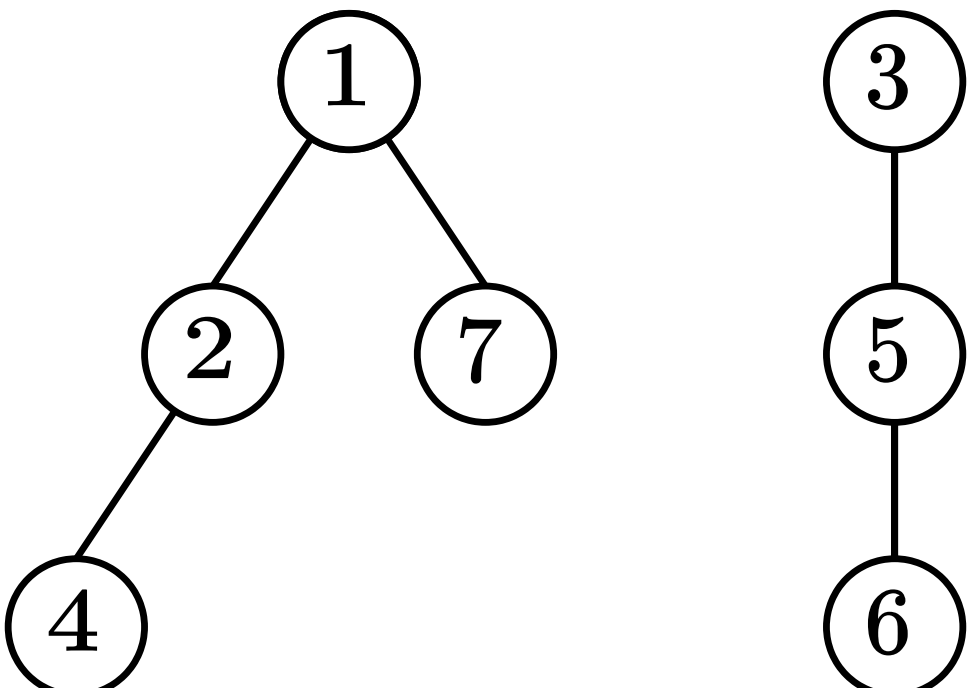
Final tweak – *path compression*.

While going through Find(x) reconnect each node on the path directly to the root.

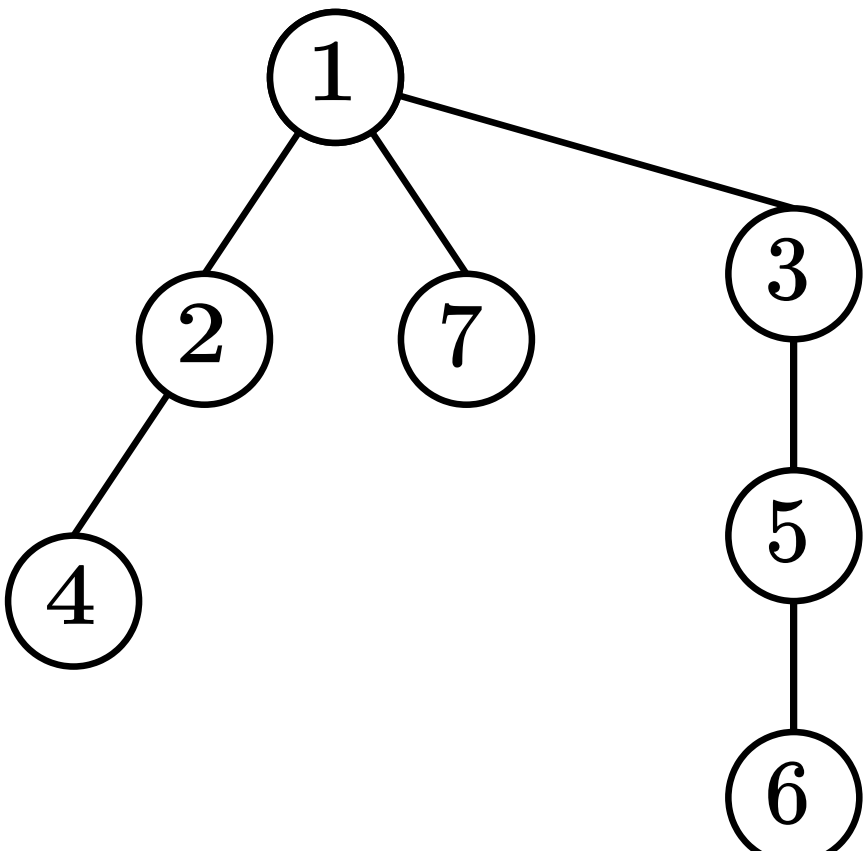
Instead of keeping tree sizes we keep *rank*, an upper bound on height.

This gives amortised $\mathcal{O}(\alpha(n))$ (worst case $\Theta(\log n)$).

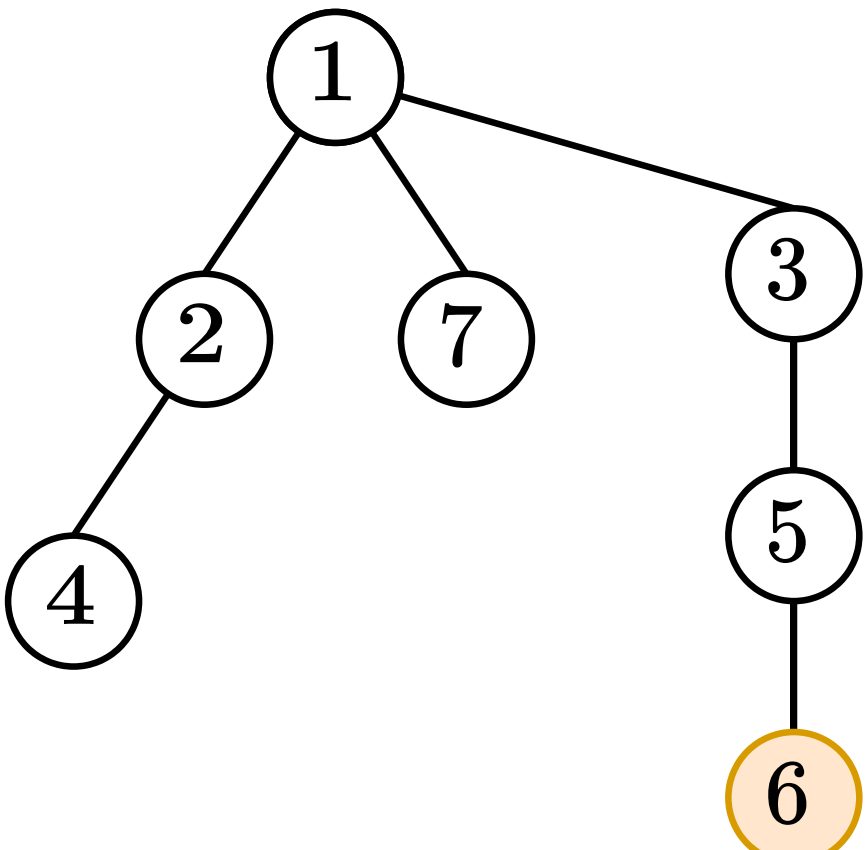
Trees – Find-Union



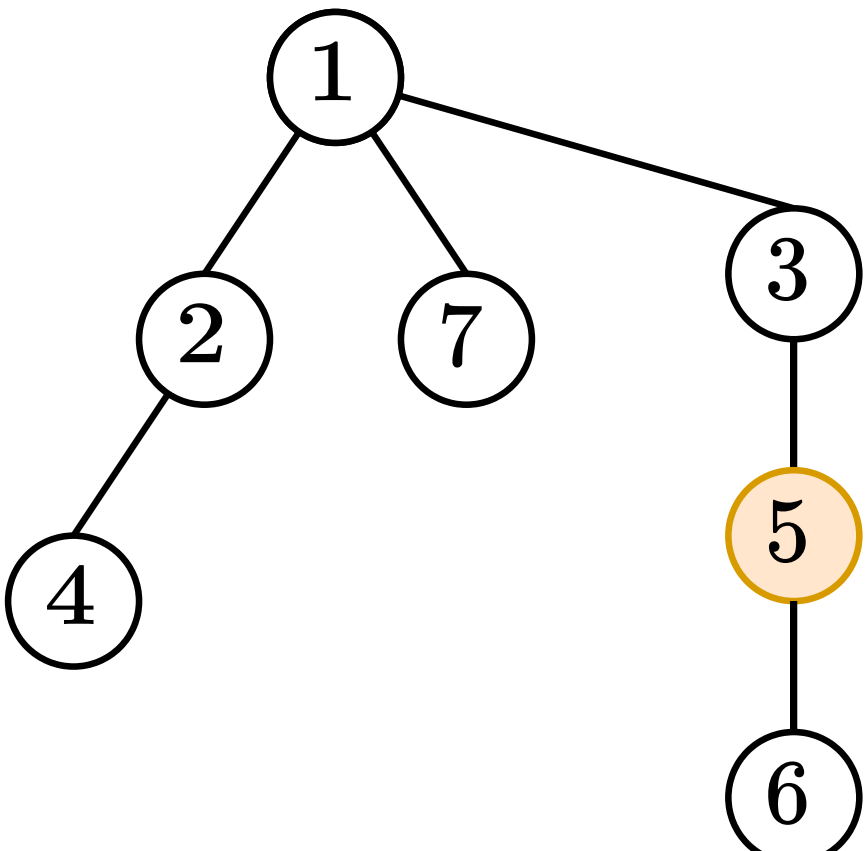
Trees – Find-Union



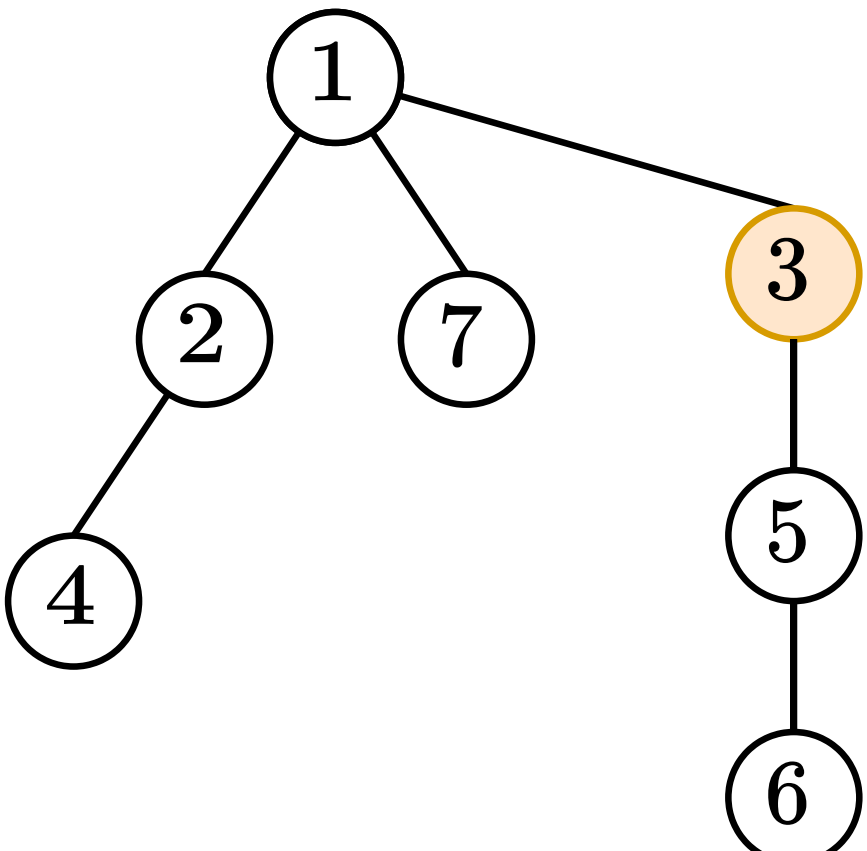
Trees – Find-Union



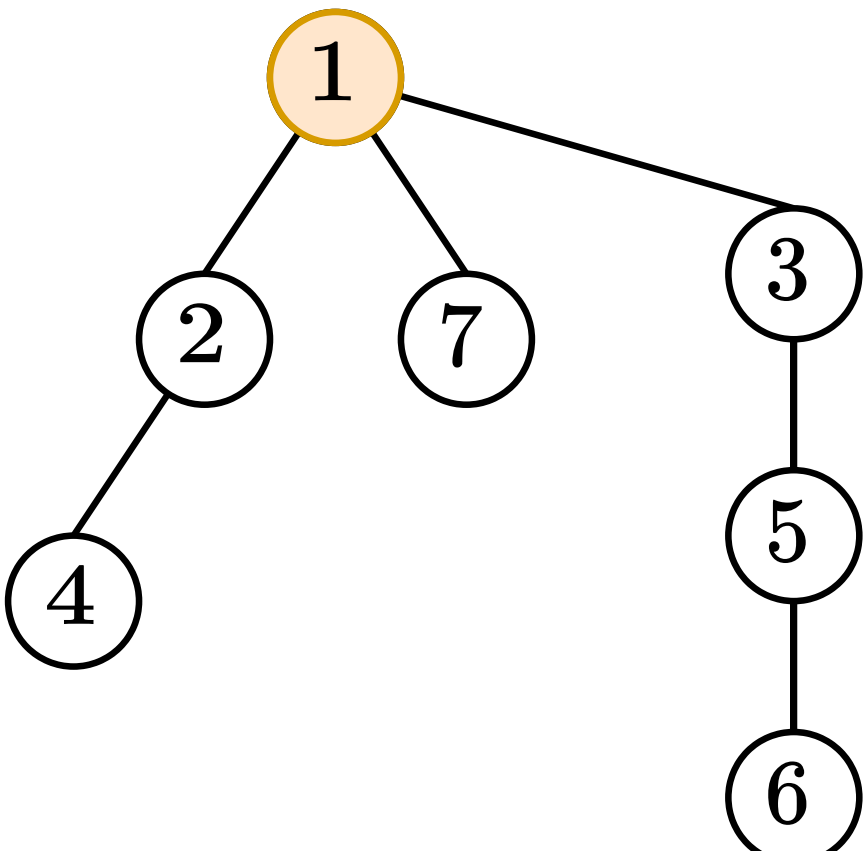
Trees – Find-Union



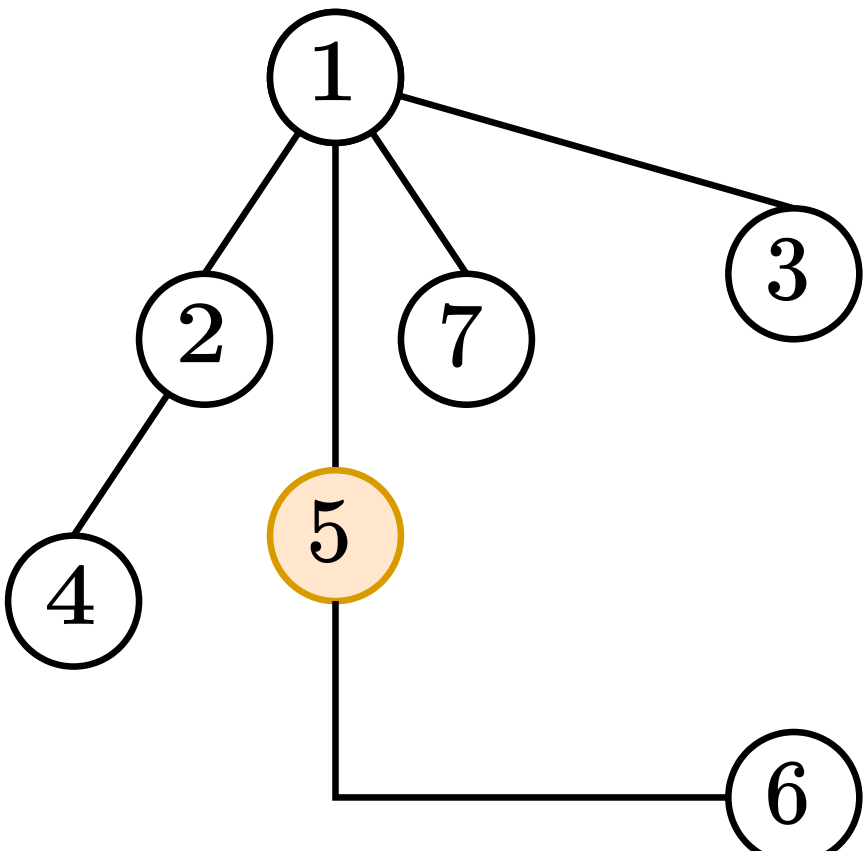
Trees – Find-Union



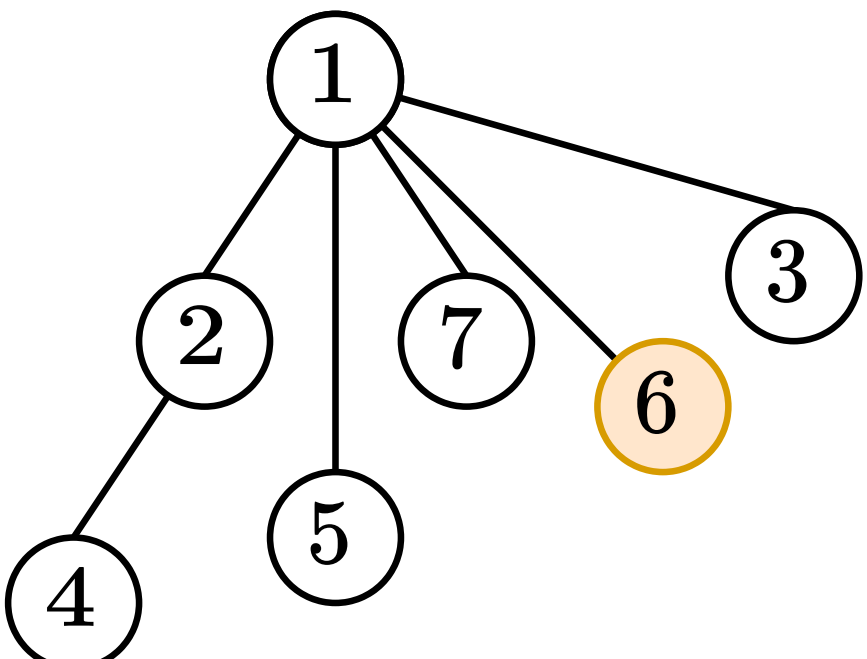
Trees – Find-Union



Trees – Find-Union



Trees – Find-Union



Trees – Find-Union

```
fn union(x, y) {
  rx = find(x)
  ry = find(y)
  if rx != ry {
    if W[rx] > W[ry] { P[ry] = rx }
    else if W[ry] > W[rx] { P[rx] = ry }
    else {
      P[rx] = ry
      W[ry] += 1
    }
  }
}
```

```
fn find(x) {
  p = P[x]
  if p != x {
    // compress
    P[x] = Find(p)
  }
  return P[x]
}
```

Trees – BSTs

This will be a separate topic in the future.

See you next week

FAL and TUV: 03.06.2025,
10:00 AM

Good luck!