COMP 251

Algorithms & Data Structures (Winter 2022)

Disjoint Sets

School of Computer Science
McGill University

Based on Based on (Cormen *et al.*, 2002) & slides of (Curles,2008), (Waldispuhl,2020) and (Langer,2014).

Outline

- Introduction.
- Operations.

Introduction - Motivation

- You have a set of nodes (numbered 1-9) on a network. You are given a sequence of pairwise connections between them:

3-7

8-2

1-6

5-7

4-8

3-5

disorganized → need a data structure

Q: Are nodes 2 and 4 (indirectly) connected?

Q: Are nodes 3 and 8 connected?

Q: Are any of the paired connections redundant?

Q: How many sub-networks do we have?

hard to answer these

Introduction - Motivation

- You have a set of nodes (numbered 1-9) on a network. You are given a sequence of pairwise connections between them:

3-7
8-2
1-6
5-7
4-8
3-5

data = same
organization = different

Data Structure

create a partition of connected elements

{3-5-7}
{2-4-8}
{9}
{1-6}

Disjoint sets

Q: Are nodes 2 and 4 (indirectly) connected? yes O(1)

Q: Are nodes 3 and 8 connected? no

Q: Are any of the paired connections redundant? yes?

Q: How many sub-networks do we have? 4

Introduction - Motivation

Q: Are nodes 2 and 4 (indirectly) connected?

Q: Are nodes 3 and 8 connected?

Q: Are any of the paired connections redundant?

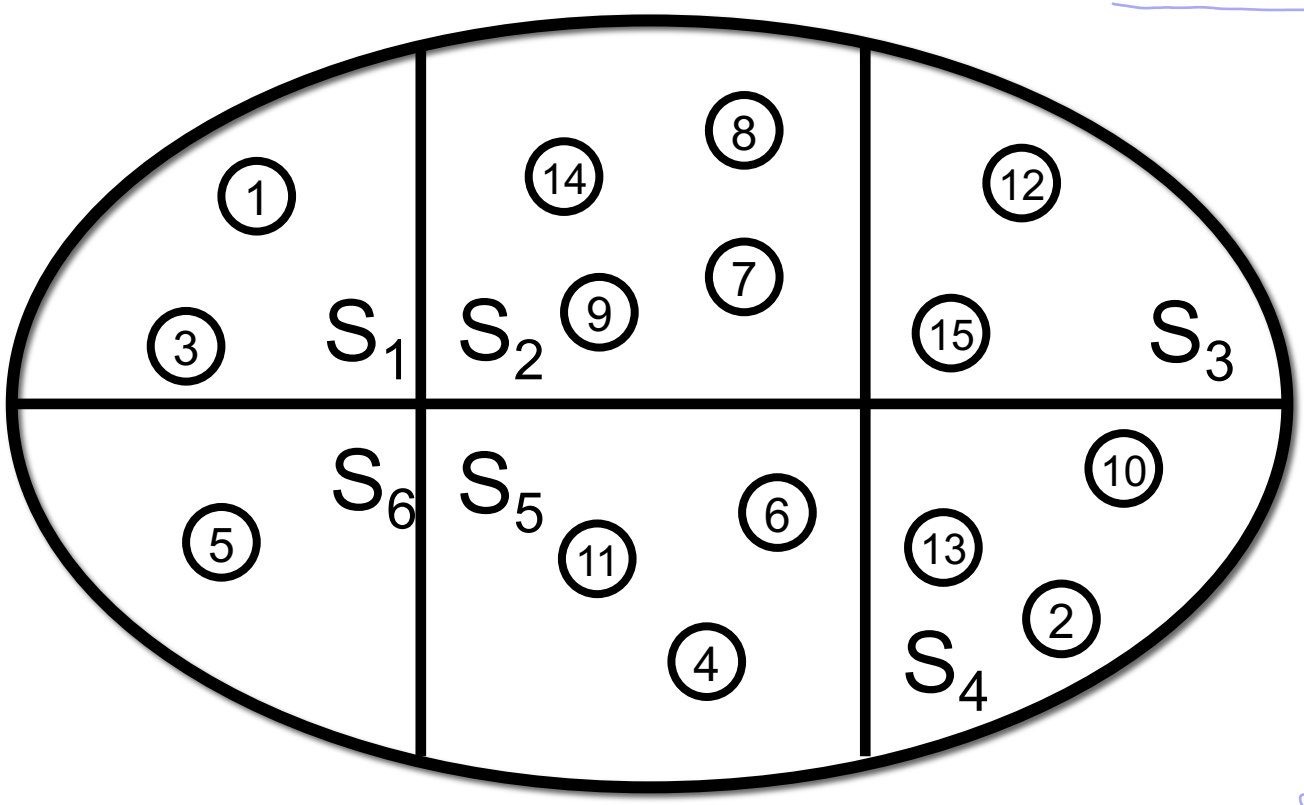
Q: How many sub-networks do we have?

- These kind of questions arises in a huge number of areas.
  - Networks
  - Transistor interconnects
  - Compilers
  - Image segmentation
  - Graph problems (upcoming topic)
  - Etc

doesn't solve all problems, solves some

Introduction - Partition

Generalization: Set of object partitioned into disjoint subsets.



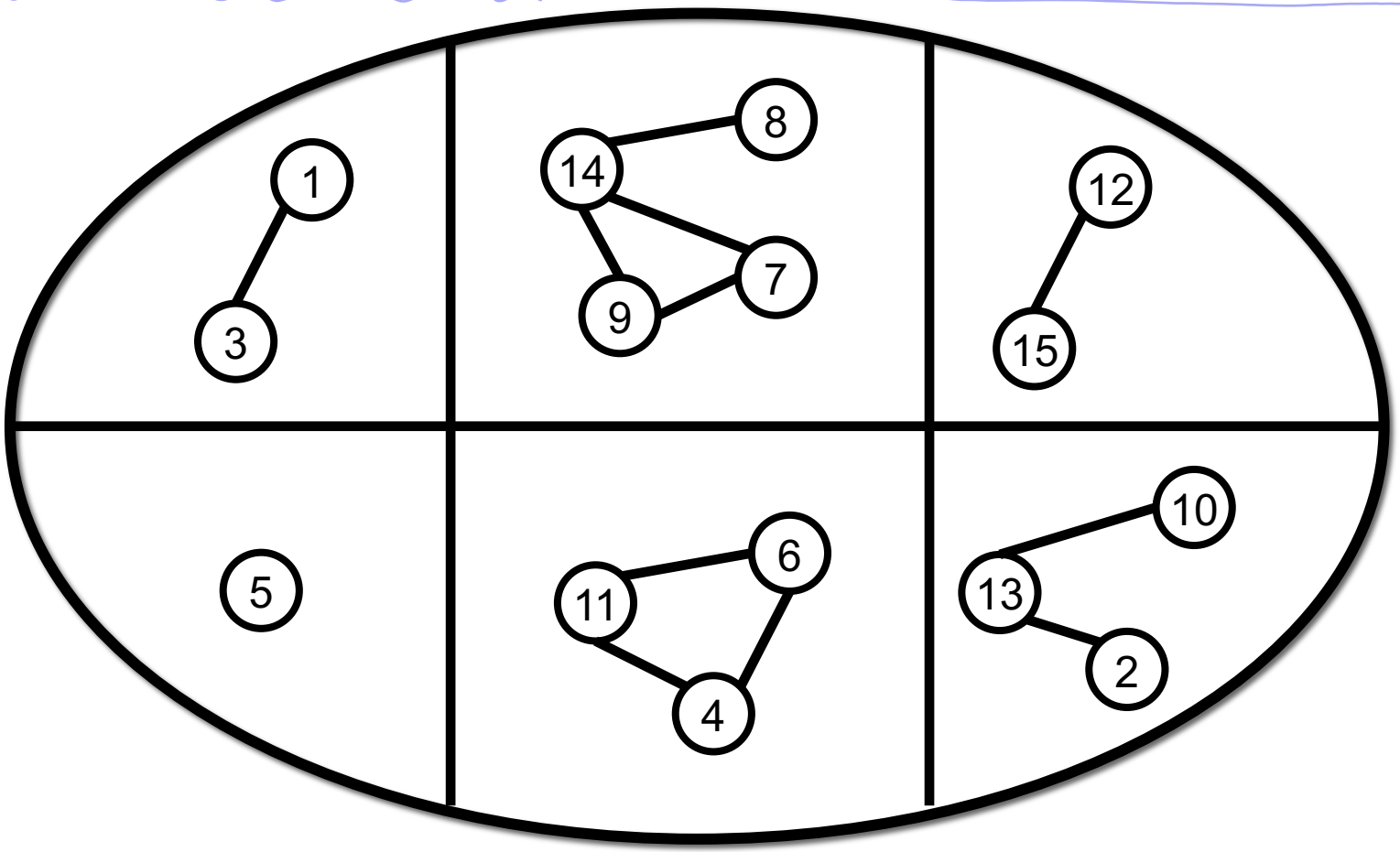
$$S = S_1 \cup S_2 \cup \ldots \cup S_n \begin{cases} S_i \neq \emptyset \ \forall i \in \{1,\ldots,n\} \\ S_i \cap S_j = \emptyset \ iff \ i \neq j \end{cases}$$

no empty partitions

partitions are disjoint (no overlap)

Introduction – Graph example

Connected component: Set of nodes connected by a path.



Question: Given 2 nodes A & B, are they in the same component?

Introduction – Equivalence relation

A relation that is:

- Reflexive $\forall a \in S, (a,a) \in R$
- Symmetric $\forall a,b \in S, (a,b) \in R \Rightarrow (b,a) \in R$
- Transitive $\forall a,b,c \in S, (a,b) \in R \; and \; (b,c) \in R \Rightarrow (a,c) \in R$

Example:

For any undirected graph, the connections define an equivalence relation on vertices.

- For all u $\in$ V, there is a path of length 0 from u to u.
- For all u,v $\in$ V, There is a path from u to v, iff there is a path from v to u.
- For all u,v,w $\in$ V, if there is a path from u to v and a path from v to w, then there is a path from u to w.

Introduction – Equivalence relation

equiv relation example

Java

equals() defines an equivalence relation on objects

http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#equals(java.lang.Object)

reflexive a.equals(a) returns true

symmetric a.equals(b) == b.equals(a)

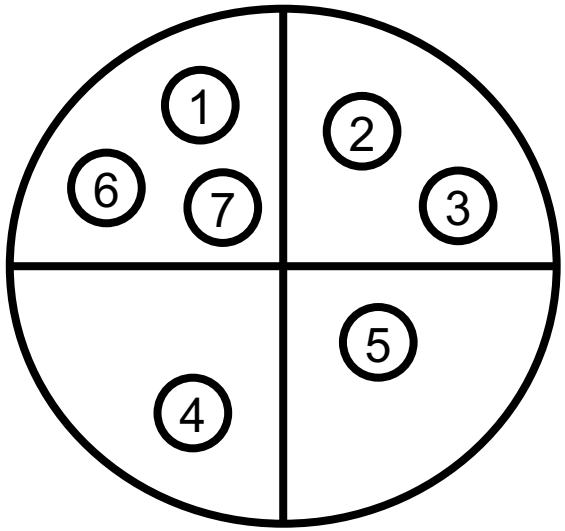
transitive a.equal(b) and b.equals(c) ⟹ a.equals(c)

Taken from Langer2014

Introduction – Equivalence relation

An equivalence relation is the same as a partition:

Each equivalence relation provides a partition of the underlying set into disjoint equivalent classes (Wikipedia).

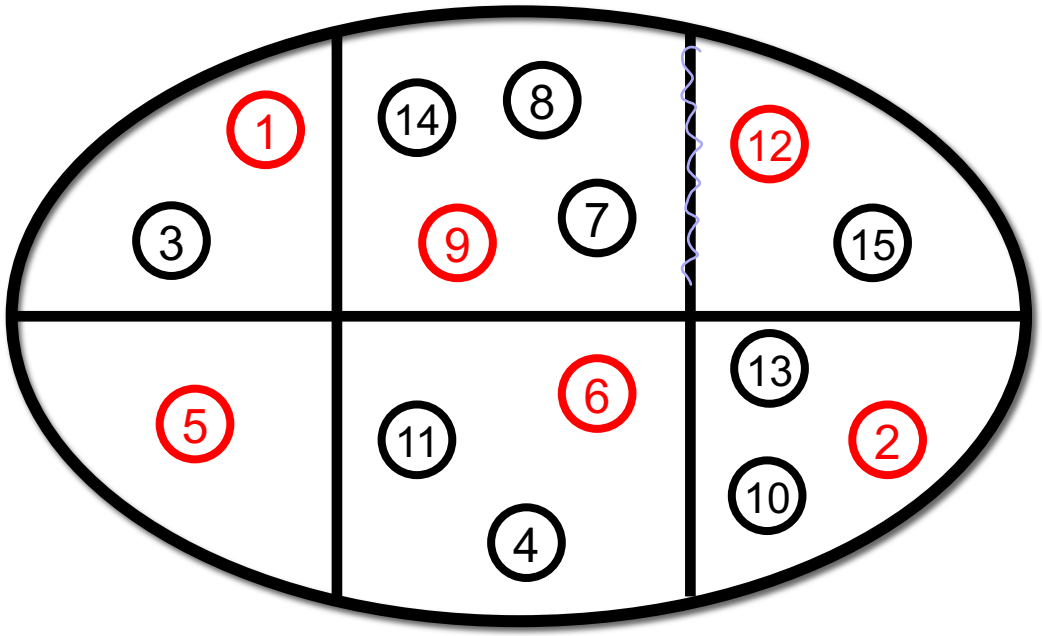


	1	2	3	4	5	6	7
1	1	0	0	0	0	1	1
2	0	1	1	0	0	0	0
3	0	1	1	0	0	0	0
4	0	0	0	1	0	0	0
5	0	0	0	0	1	0	0
6	1	0	0	0	0	1	1
7	1	0	0	0	0	1	1

i is equivalent to j if they belong to the same set.

(more constrained that general relation)

Introduction – Disjoint set ADT



union(12,9)

find(3) = 1

find(11) = 6
find(4) = 6
sameset(11,4) = true

i is equivalent to j if they belong to the same set.

Each set in the partition has a unique name (a representative member for convenience).

- find(i) returns the representative of the set that contains i.
- sameset(i,j) returns the boolean value find(i)==find(j)
- union(i,j) merges the sets containing i and j.

if representative is the same

deletes divisions

Introduction - Operations

- Maintain disjoint sets:
  - {3, 5, 7}, {4, 2, 8}, {9}, {1, 6}
- Each set has a representative:
  - {3, 5, 7}, {4, 2, 8}, {9}, {1, 6}
- Find(x) returns the representative of the set containing x
  - Find(1) = 5
  - Find(4) = 8
- Union(x,y) takes the union of the two sets that contains x and y
  - Union(5,1) = {3, 5, 7, 1, 6}, {4, 2, 8}, {9}
  - Union(9,6) = {3, 5, 7}, {4, 2, 8}, {9, 1, 6}

any number can be the representive in some cases

Introduction - Operations

- Maintain disjoint sets:
  - {3, 5, 7}, {4, 2, 8}, {9}, {1, 6}
- Each set has a representative:
  - {3, 5, 7}, {4, 2, 8}, {9}, {1, 6}
- Q: Are nodes 2 and 4 (indirectly) connected?
  - Find(2) == Find(4)
- Q: Are nodes 3 and 8 connected?
  - Find(3) == Find(8)
- Q: How many sub-networks do we have?
  - Number of representatives.
- Q: Are any of the paired connections redundant?
  - Connections – (|S1-1| + |S2-1| + |S3-1| + |S4-1)| = 6 – (2+2+0+1) = 1

check their representatives

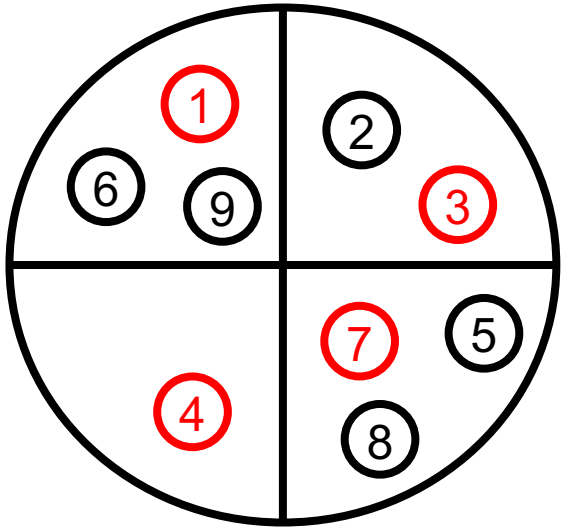
Outline

- Introduction.
- Operations.

Operations – find

Rep[]

1	1
2	3
3	3
4	4
5	7
6	1
7	7
8	7
9	1



constant time

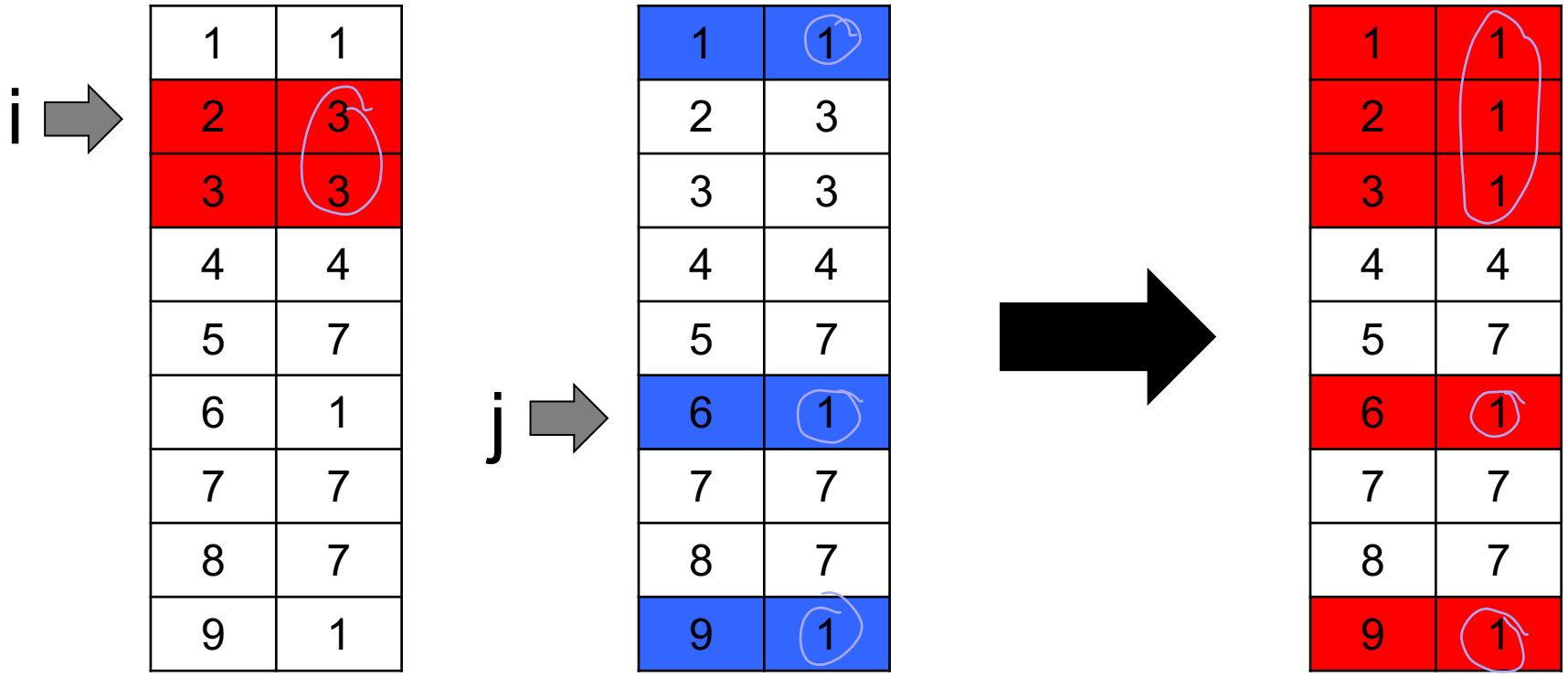
Let Rep[i] ∈ { 1, 2, … , n } be the representative of the set containing i.

find(i) { return rep[i]; }

Operations – union

- union(i,j) merges the sets containing i and j.

Example: union(2,6)



1	1
2	3
3	3
4	4
5	7
6	1
7	7
8	7
9	1

1	1
2	3
3	3
4	4
5	7
6	1
7	7
8	7
9	1

1	1
2	1
3	1
4	4
5	7
6	1
7	7
8	7
9	1

Operations – union

```
union(i,j) {
     if rep[i] != rep[j] {
          prevrepi = rep[i];
          for (k=1; k<=n; k++) {
               if rep[k] == prevrepi {
                    rep[k] = rep[j];
               }
          }
     }
}
```

check they are 2 different partitions

update reps

go through table and check if each element needs to be updated

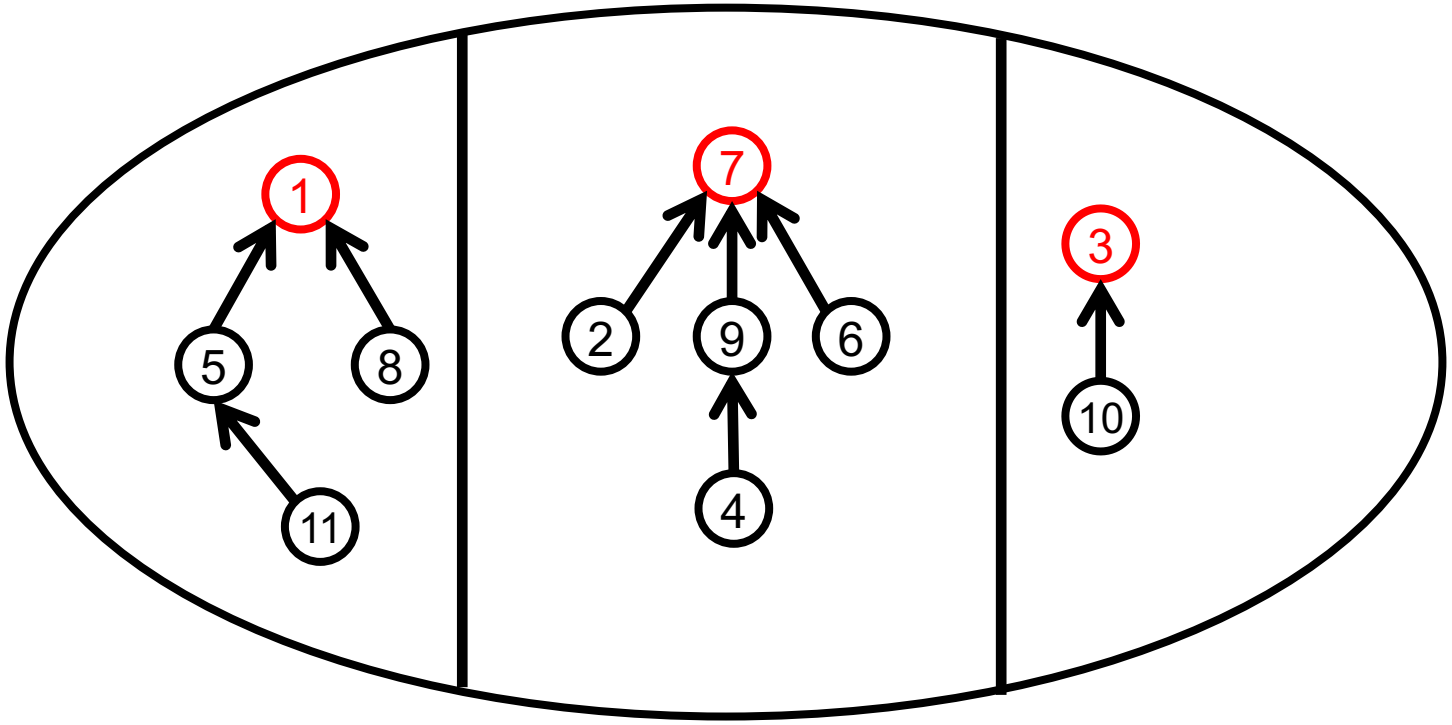
- store value of rep[i] because it may change during the execution of the algorithm.

! prevrepi

- O(n) running time… slow.

Union – forest & tree representation

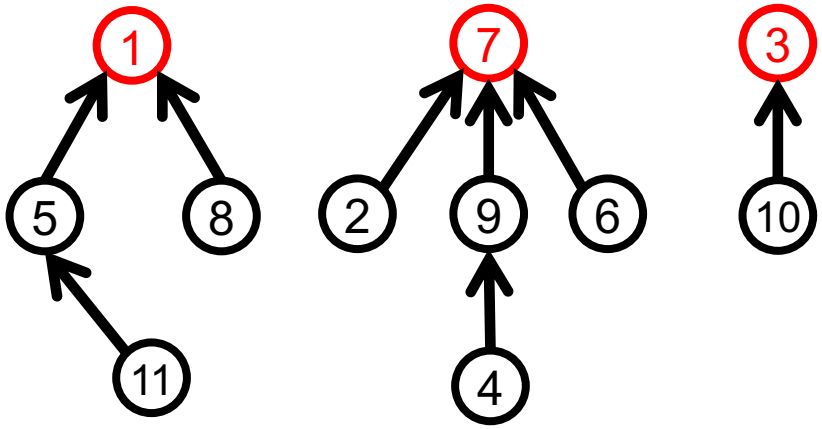
- Represent the disjoint sets by a forest of rooted trees.
  - Roots are the representative (i.e. find(i) == findroot(i)).
  - Each node points to its parent.



Note:
The tree structure does not necessarily represent the relationship between the stored objects.

Union – table representation

	p[]
1	1
2	7
3	3
4	9
5	1
6	7
7	7
8	1
9	7
10	3
11	5



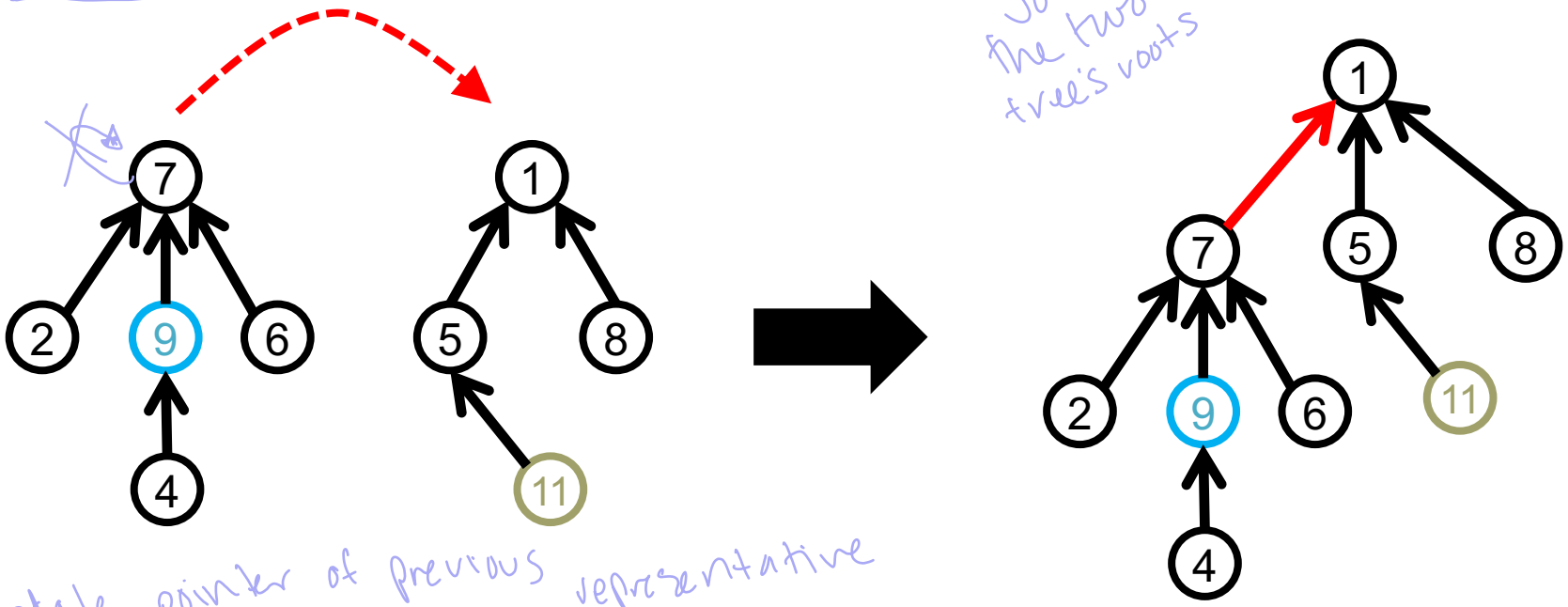
- Non-root nodes hold index of their parent.
- Root nodes store their own value.

Operations – Union

union(9,11)

easier / faster

Root of the tree of 11 becomes the parent of the root of the tree of 9.



update pointer of previous representative

Operations – Find & Union

```
find(i) {
      if p[i] == i {
            return i;
      } else {
            return find(p[i]);
      }
}

union(i,j) {
      if find(i) != find(j) {
            p[find(i)] = find(j);
      }
}
```

traverse tree by parents until reach the root

Remark: Arbitrarily merge the set on i into the set of j.

update representative pointer to rep of other tree instead of itself

Operations – Union - > Worst case

union(1,2)

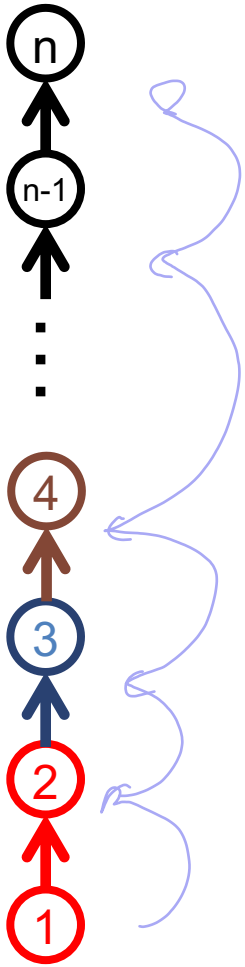
union(1,3)

union(1,4)

…

union(1,n)

Then, find(1) is O(n)…



Union – Heuristic – Definitions

- The **depth** of a node is the number of edges from the node to the tree's root node. A root node will have a depth of 0.
- The **height** of a node is the number of edges on the *longest path* from the node to a leaf. A leaf node will have a height of 0.

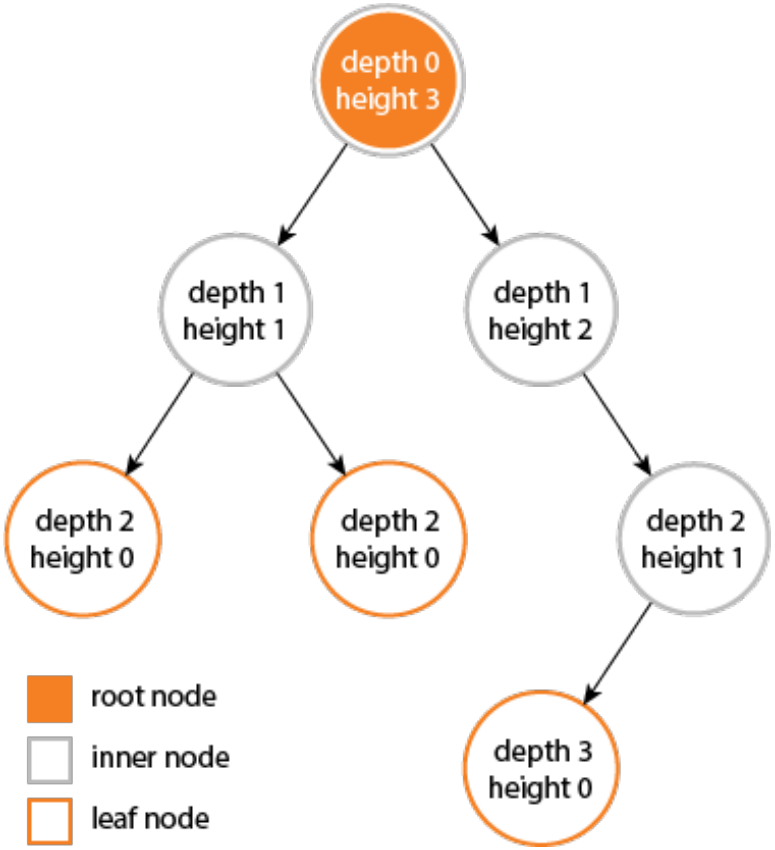
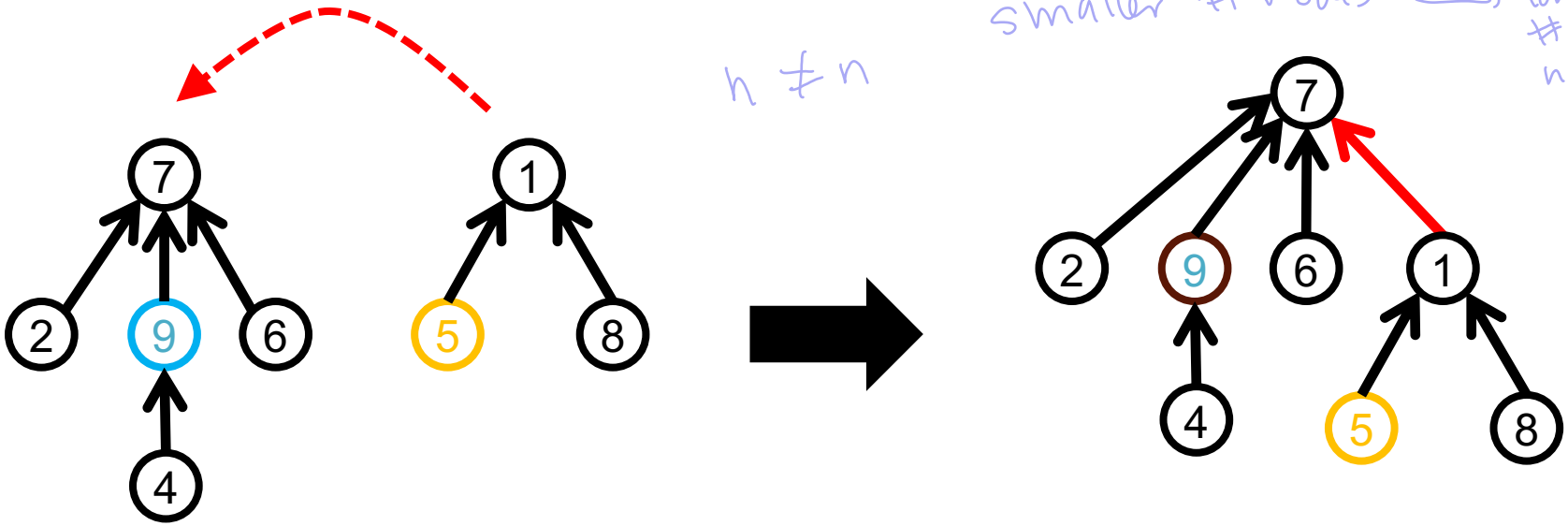


Image and definitions taken from:
https://stackoverflow.com/

Union – Heuristic – by size

Heuristic to control the height to the trees after merging.

Idea: Merge tree with smaller number of nodes into the tree with the largest number of nodes (In practice, we can also use rank which is an upper bound on the height of nodes).

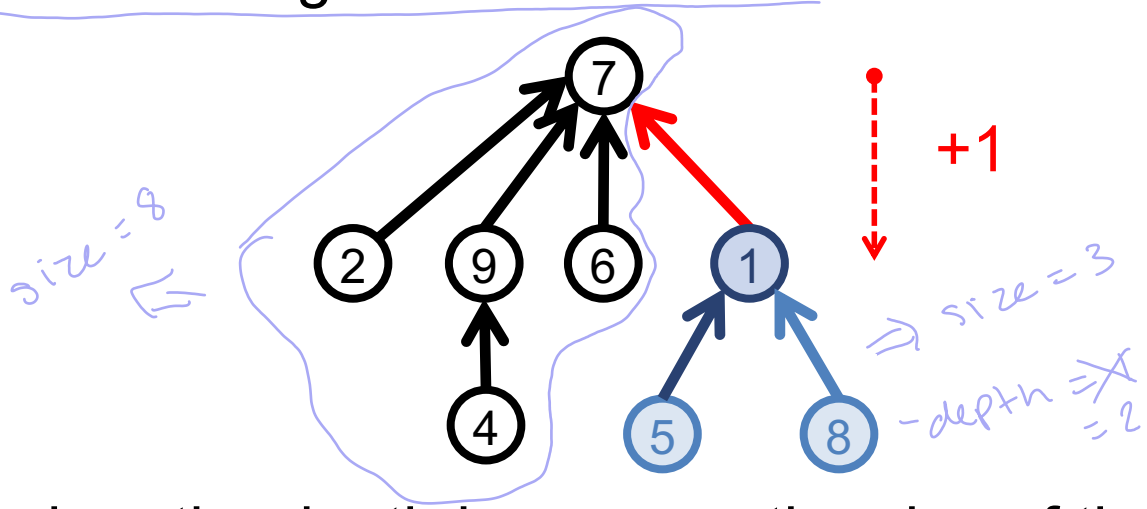


Union – Heuristic – by size

Claim: The depth of any node is at most log n.

Proof:

- If union causes the depth of a node to increase, then this node must belong to the smallest tree.



- Thus, when the depth increases, the size of the (merged) tree containing this node will at least double.
- But we can double the size of a tree at most log n times.

Union – Heuristic – by height

Idea: Merge tree with smaller height into tree with larger height.

Claim: The height of trees obtained by union-by-height is at most log n.

Corollary: An union-by-height tree of height h has at least $n_h \geq 2^h$ nodes.

Proof (Corollary):

- Base case: a tree of height 0 has one node.
- Induction: (hypothesis) $n_h \geq 2^h$. Show $n_{h+1} \geq 2^{h+1}$.

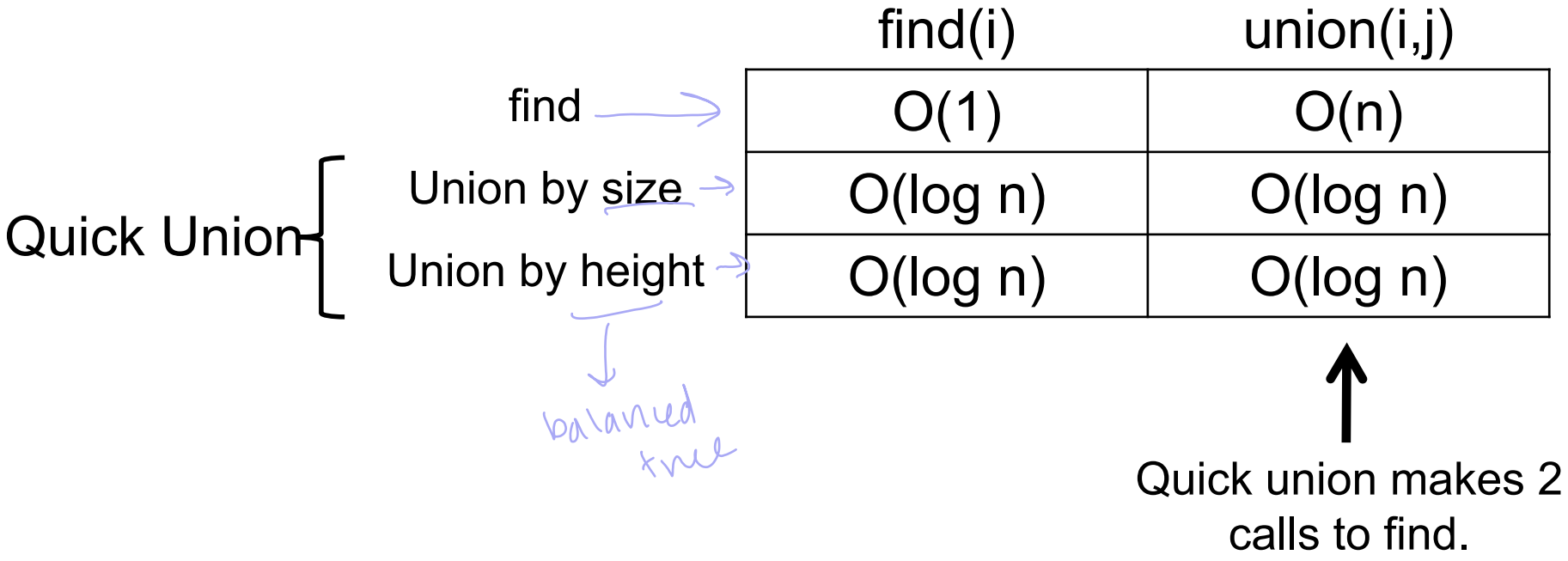
try first

Heuristic – supplemental material

The base case k=0 is easy, since a tree of height 0 always has just 1=2^0 node (the root). Suppose the claim is true for h=k. Now consider a union-by height tree of height k+1. There must have been a union that brought two trees together and increased the height of one of them from k to k+1. Let those two trees (at the time of that union) be T1 and T2. We know that both T1 and T2 were of height k before the union. [Why? If one of them were of height less than k, then union-by-height would have changed the root of that shorter one to make it point to the root of the taller one, and the height of the unioned tree would still be k. But its not; the unioned tree is of height k+1.] Now we can apply the induction hypothesis: the trees T1 and T2 each have at least 2^k nodes. Thus, the unioned tree has at least 2^k + 2^k = 2^(k+1) nodes.

Taken from Langer 2014

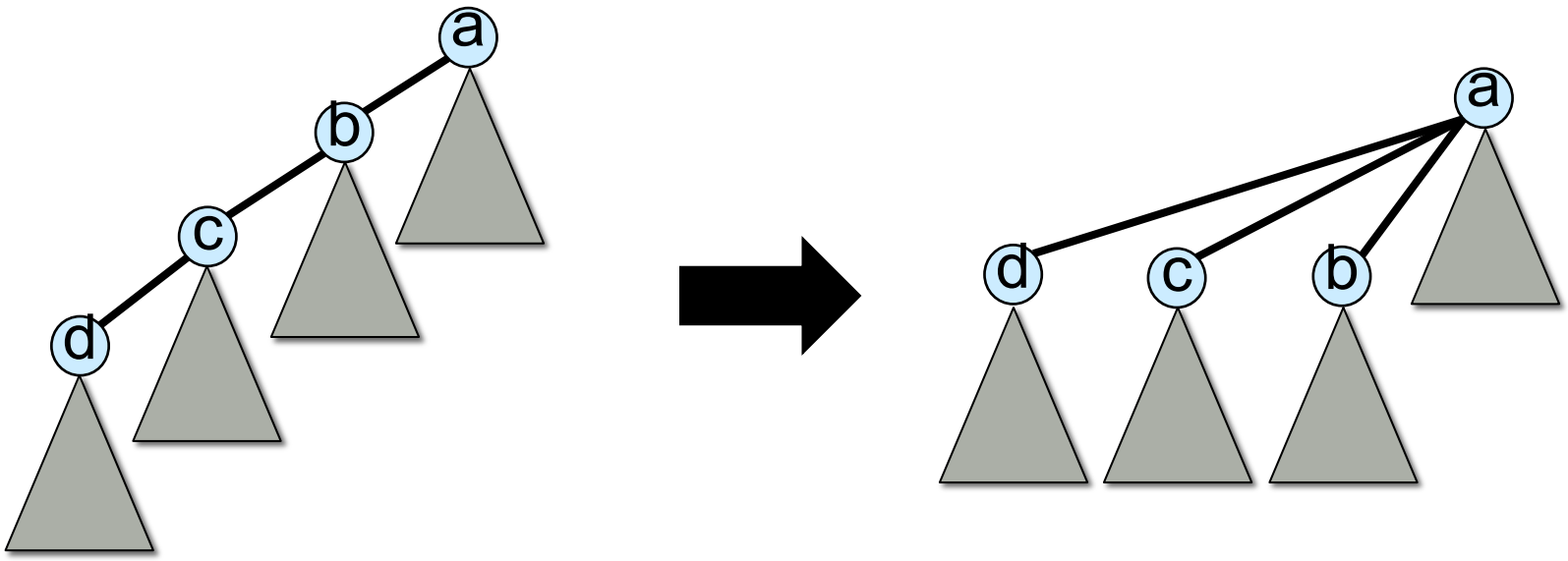
Union – running time



Note: These are worst case complexities.

union-find – heuristics – path compression

- Find path = nodes visited during the execution of find() on the trip to the root.
- Make all nodes on the find path direct children of root.



union-find – heuristics – path compression

```
find(i) {
    if p[i] == i {
        return i;
    } else {
        p[i] = find(p[i]);
        return p[i];
    }
}
```

"O(1)"

union-find – running time

- Use union by size and path compression.
- **m union or find operations** take O(m α(n)).

What is α(n) ?

n	α(n)
0 - 2	0
3	1
4 - 7	2
8 - 2047	3
2048 – $A_4(1)$	4

Where $A_4(1) >> 10^{80}$!!

union-find – running time

- Use union by size and path compression.
- Huge Practical problem.
  - 10^{10} edges connecting 10^9 nodes.
  - The heuristics reduces time from 3.000 years to 1 minute
  - Supercomputer wont help much
  - Good algorithm makes solution possible
  - Good algorithms makes it possible to solve problems that could not otherwise be addressed.

Outline

- Introduction.
- Operations.


time for questions