COMP251: Disjoint sets

Jérôme Waldispühl & Giulia Alberini

School of Computer Science

McGill University

Based on slides from M. Langer (McGill)

Problem

Let G=(V,E) be undirected graph, and A, B $\in$ V two nodes of G.

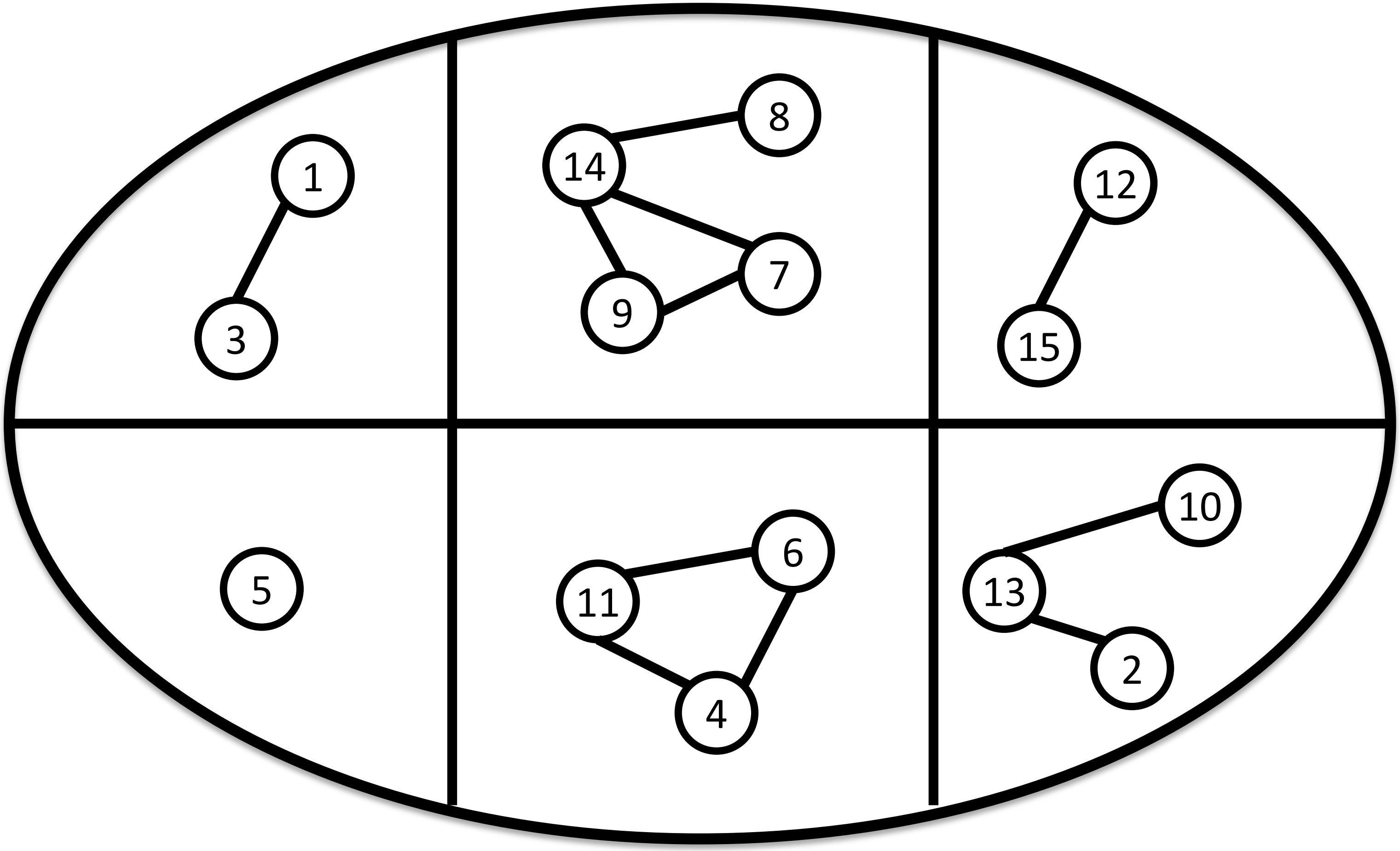
Question: Is there a path between A and B?

But **we are not interested in knowing the path between A and B**.

Is there a faster way to solve this problem than with an explicit search (i.e., faster than DFS or BFS)?

Connected components

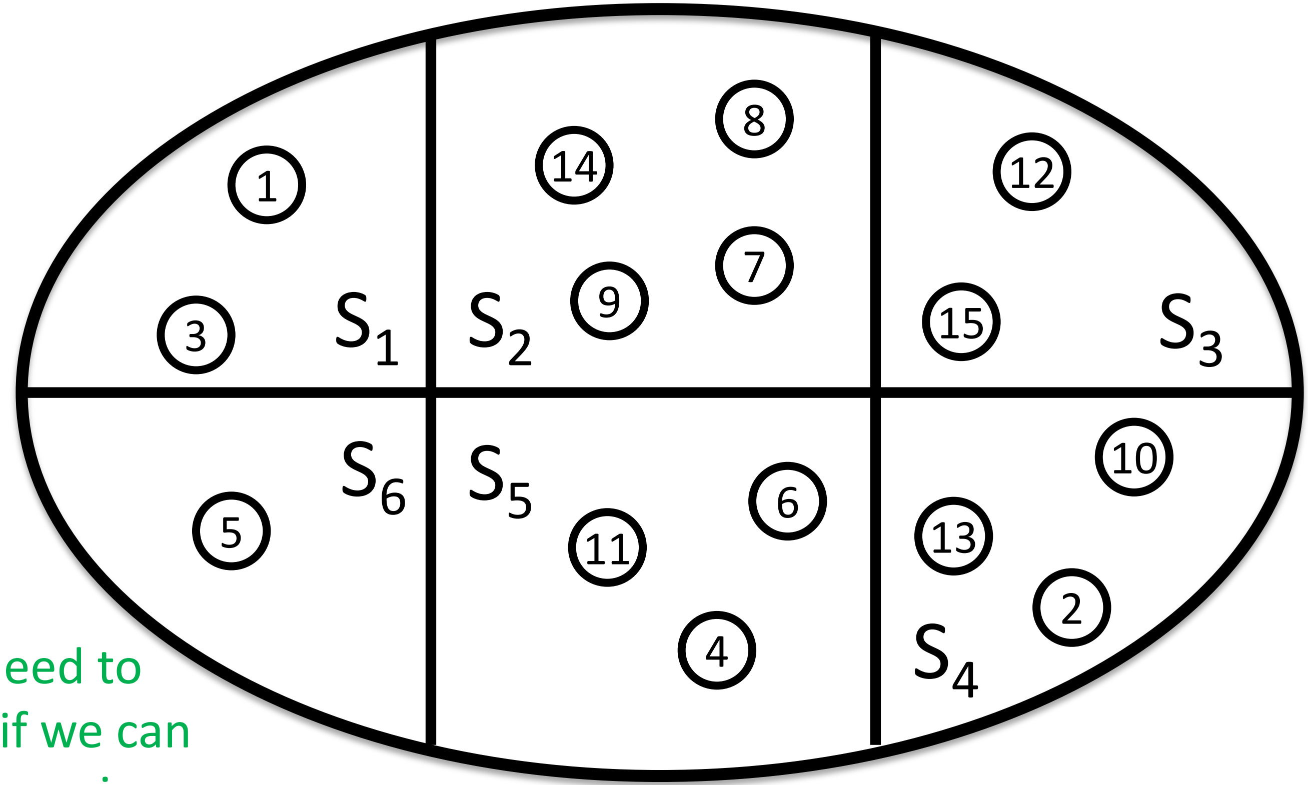
Connected component: Set of nodes connected by a path.



Our question becomes: Are 2 nodes A & B in the same component?

Partition

Generalization: Set of object partitioned into disjoint subsets.



We do not need to store edges if we can guarantee there is a path between two nodes iff they are in the same component.

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

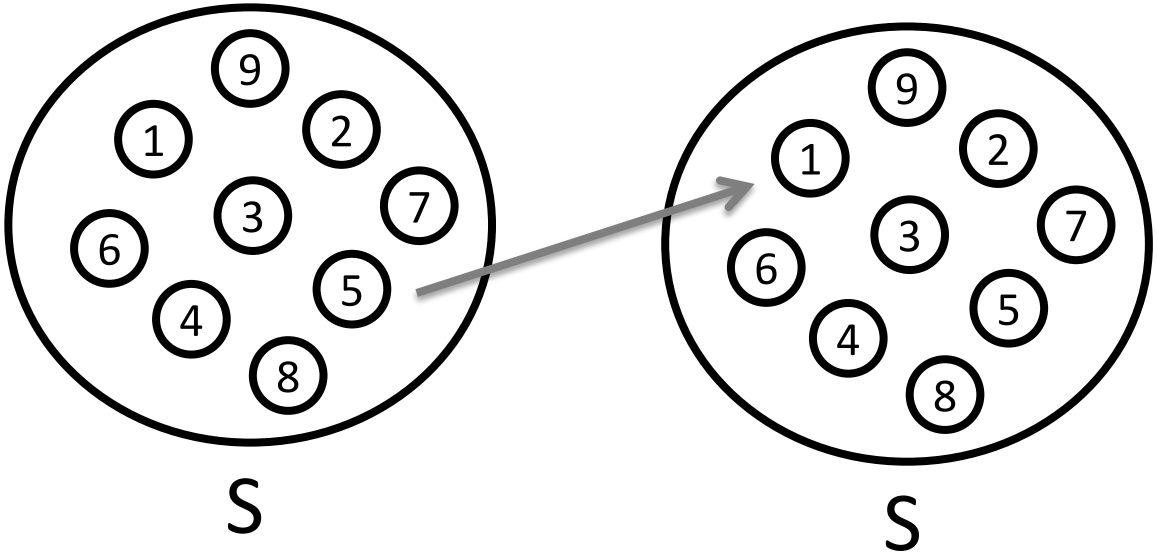
Motivations

- Data structure used to manage sets and perform classical operations (union, find, intersection…)
- Used in many algorithms we will cover in upcoming lectures (e.g. Kruskal, Floyd-Marshall)

Map vs. Relation

Map or function

f:



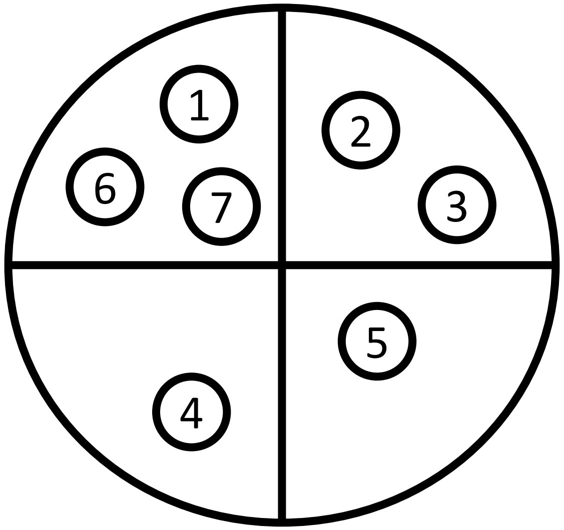
{(a,f(a))}

Relation R $\subseteq$ { (a,b) : a,b $\in$ S }

$$
\begin{array}{c|ccccc|}
 & 1 & 2 & 3 & 4 & 5 \\
\hline
1 & 0 & 1 & 1 & 0 & 1 \\
2 & 0 & 1 & 0 & 1 & 0 \\
3 & 1 & 0 & 0 & 1 & 0 \\
4 & 1 & 1 & 0 & 0 & 1 \\
5 & 0 & 1 & 1 & 0 & 0 \\
\hline
\end{array}
$$

Any Boolean matrix defines a relation

Equivalence relation



$$\begin{array}{c|ccccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\
2 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
3 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
4 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
5 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
6 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\
7 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\
\hline
\end{array}$$

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

(more constrained that general relation)

Equivalence relation

- Reflexivity $\forall a \in S, (a,a) \in R$
- Symmetry $\forall a,b \in S, (a,b) \in R \Rightarrow (b,a) \in R$
- Transitivity $\forall a,b,c \in S, (a,b) \in R \; and \; (b,c) \in R \Rightarrow (a,c) \in R$

Example (Java)

`equals()` defines an equivalence relation on objects.

- Reflexivity:

  `a.equals(a)`returns `True`

- Symmetry:

  `a.equals(b)== b.equals(a)`

- Transitivity:

  `a.equals(b)`and `b.equals(c)`implies `a.equals(c)`

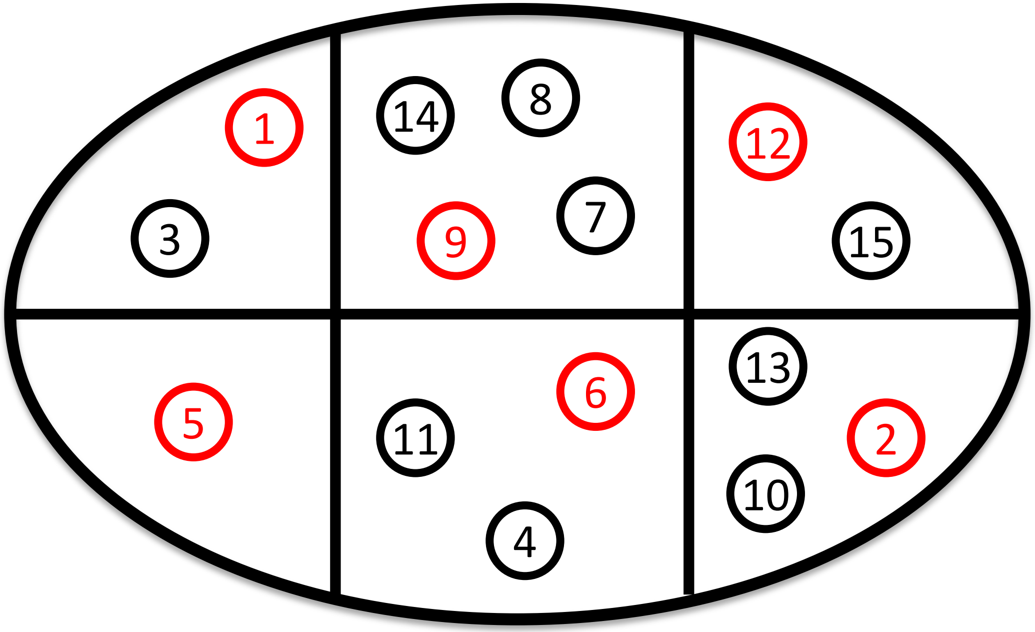
Example (Graph)

For any **undirected** graph G(V,E), *pathconnected(u,v)* defines an equivalence relation on vertices ($u, v \in V$).

- Reflexivity: There is a path of length 0 from u to u
- Symmetry: There is a path from u to v iff there is a path from v *to* u
- Transitivity: If there is a path from u to v *and* a path from v to w*, then there* a path from u to w

Our objective is to design a data structure that will store this equivalence relation on vertices.

Disjoint set ADT



Each set in the partition as a representative member.

- *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*.

Union of disjoint sets

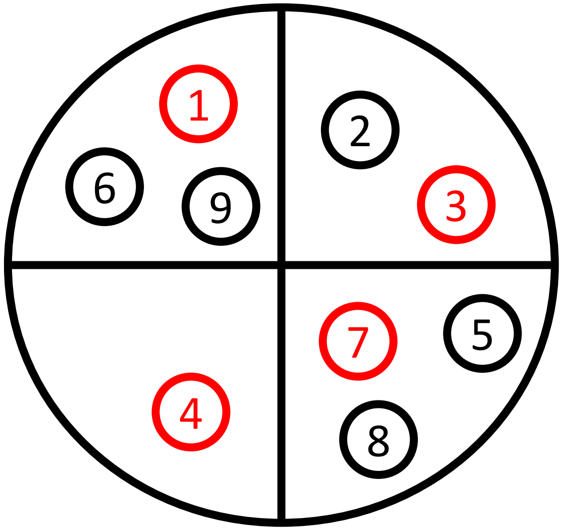
union(i,j) merges the sets containing i and j.

- Does nothing if i and j are already in the same set.
- Otherwise, we merge the set **and** need a policy to decide who will be the representative of the new merged set.

Quick find

Rep[]

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

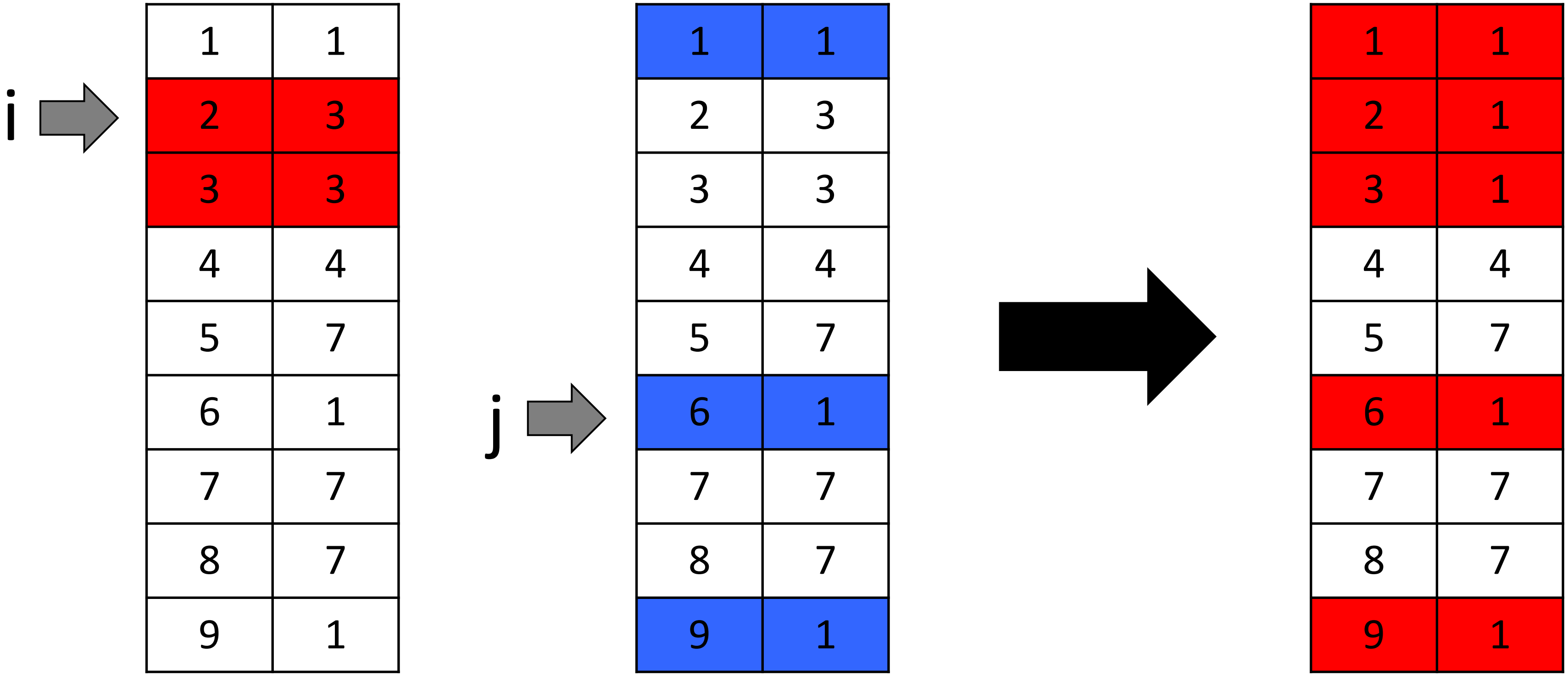


Let Rep[i] ∈ { 1, 2, … , n } be the representative of the set containing i.

(Quick find) & union

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

(Quick find) & union

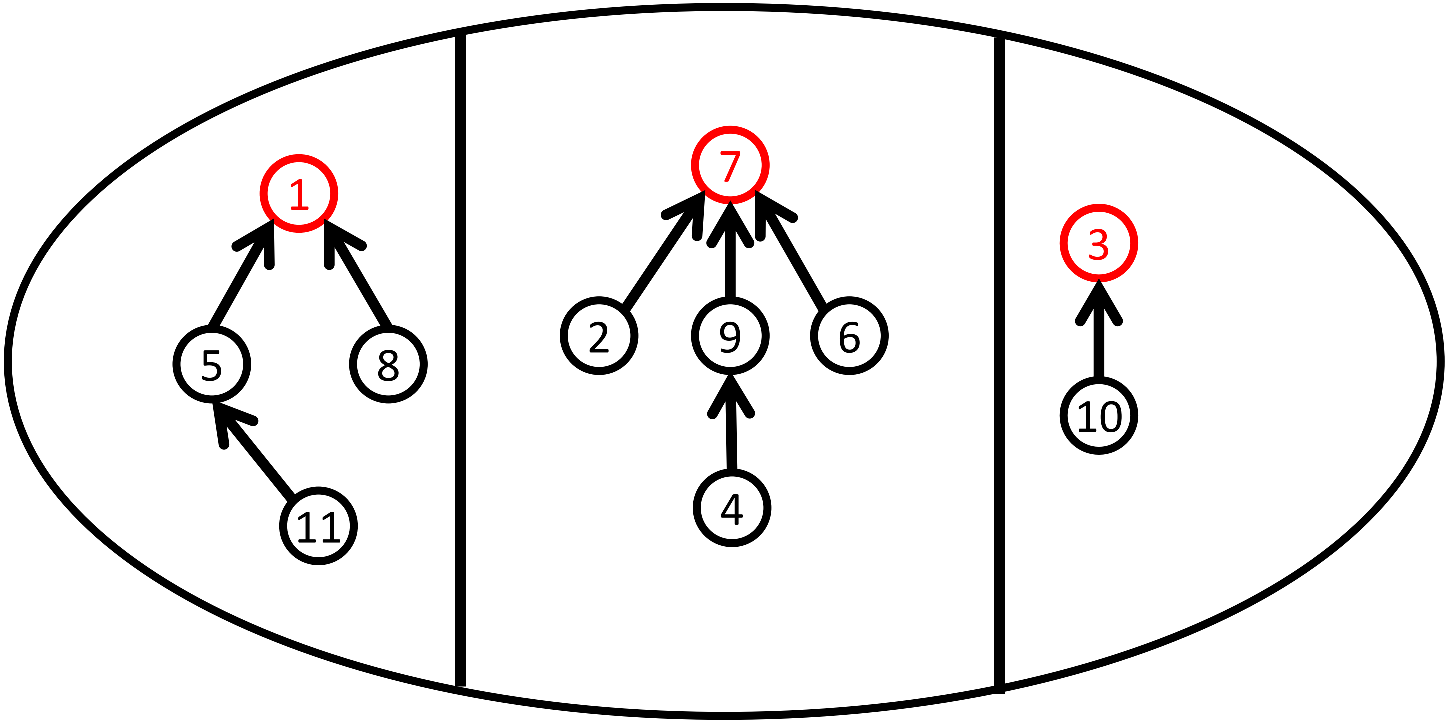
```
union(i,j) {
   if rep[i] != rep[j] {
      prevrepi = rep[i];
      for (k=1; k<=n; k++) {
         if rep[k] == prevrepi {
            rep[k] = rep[j];
         }
      }
   }
}
```

- store value of rep[i] because it may change during the execution of the algorithm.
- O(n) running time… **slow…**

Can we do better?

Tree representation & forests

- 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.

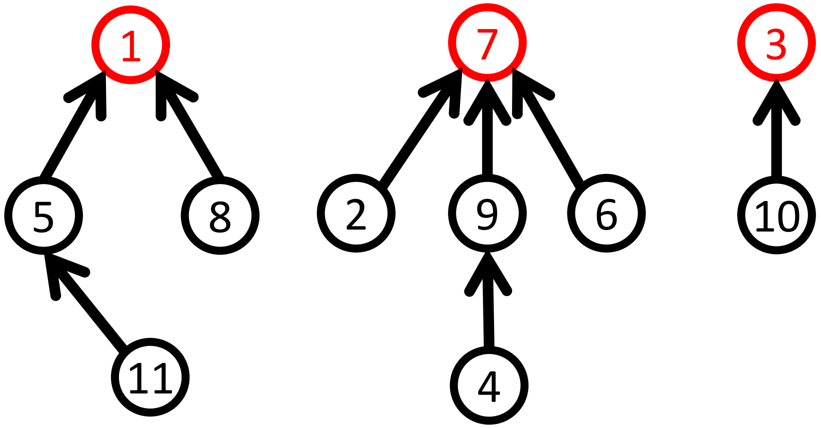


IMPORTANT NOTE: *The tree structure does not necessarily represent the relationship between the stored objects.*

Array 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.

Find & Union

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

Find becomes a bit more complex & expensive

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

But union is much simpler & faster!

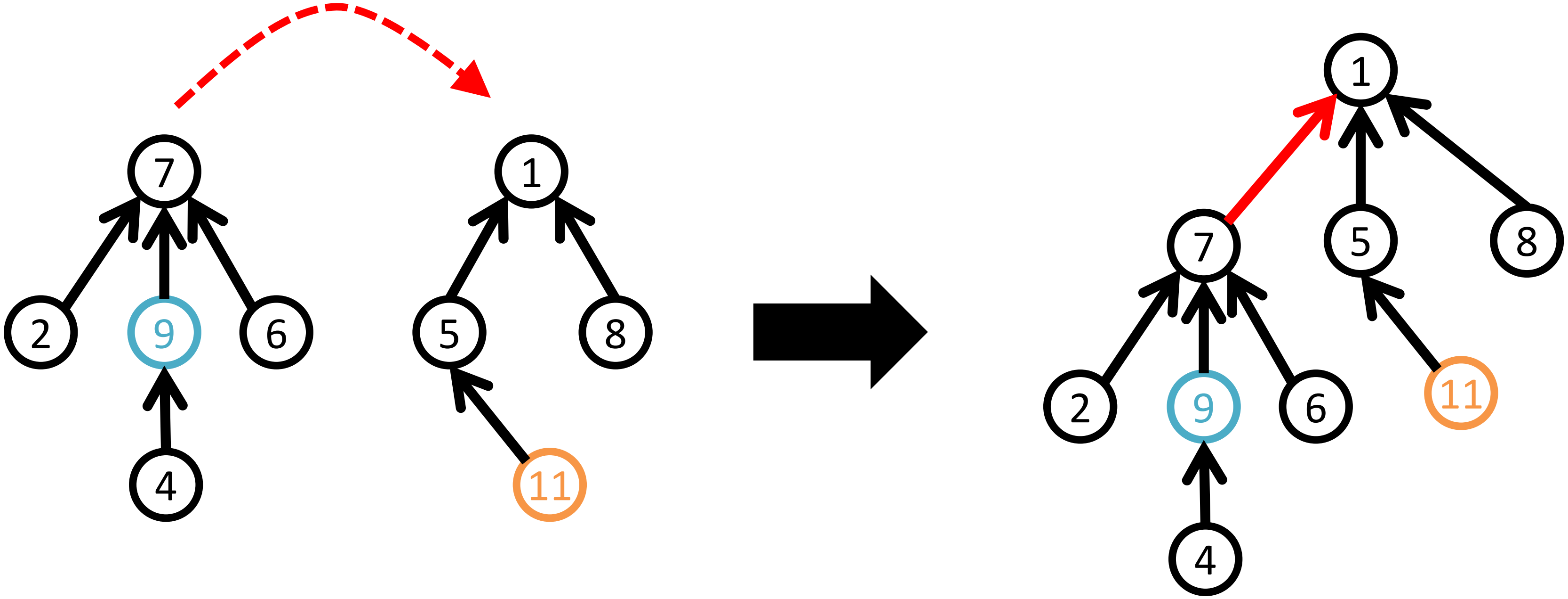
Remark: Arbitrarily merge the set on i into the set of j.

Union example

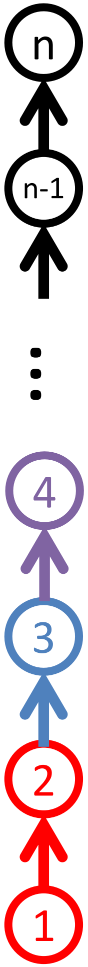
union(9,11)

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

How do you decide how to merge the trees?



Worst case



union(1,2)

union(1,3)

union(1,4)

…

union(1,n)

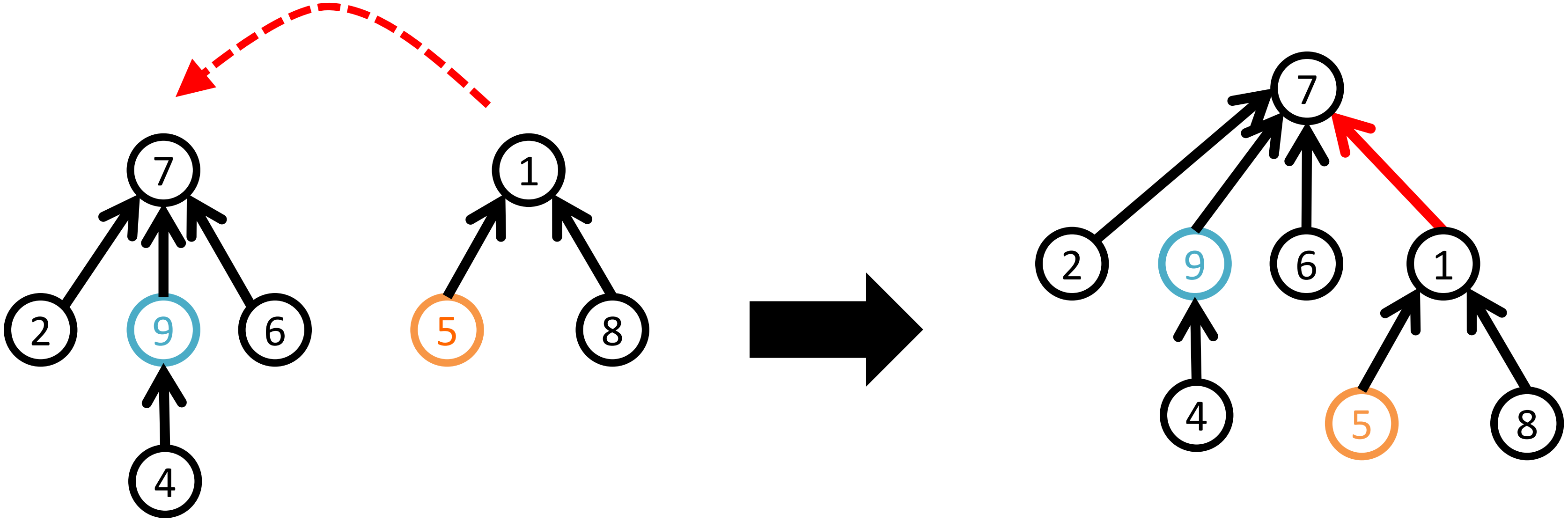
Then, find(1) is O(n)…

This does not seem like a successful improvement… Can we do better?

Union by size

We will use a heuristic to control the height of the trees after merging, and thus guarantee the efficient of find().

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

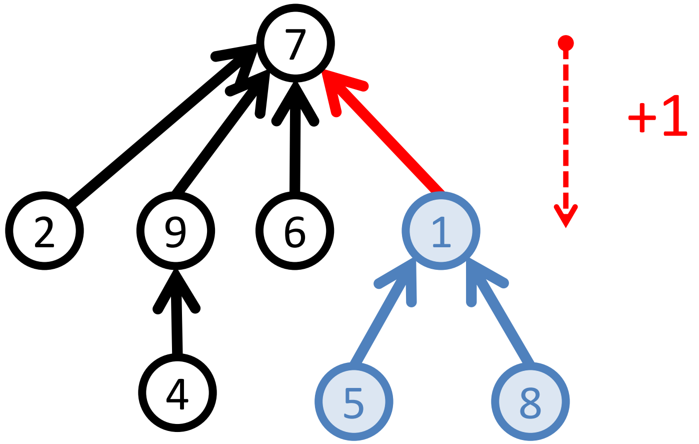


Union by size

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 (by definition of union).



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

Union 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: A 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 $n_{h+1} \geq 2^{h+1}$.

if the height of the union increases, then both trees should be of height h. Thus, the new tree has 2 subtrees with at least 2^h nodes.

Running time

		find(i)	union(i,j)
	Quick find	O(1)	O(n)
Quick Union	Union by size	O(log n)	O(log n)
	Union by height	O(log n)	O(log n)

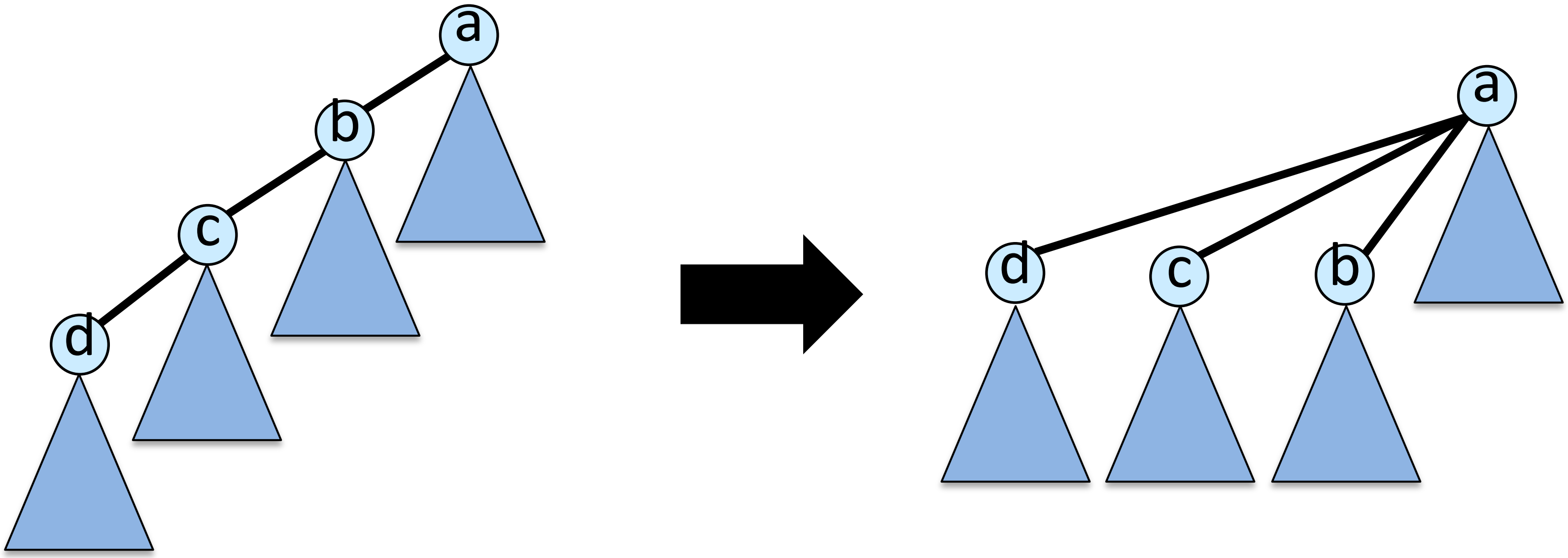
↑

Quick union makes
2 calls to find.

Note: These are worst case complexities.

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.



Path compression

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

Running time

- Use union by size and path compression.
- Worst case running time is O(log n).
- However, we can show that **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 $A_4(1) >> 10^{80}$!!

Appendix

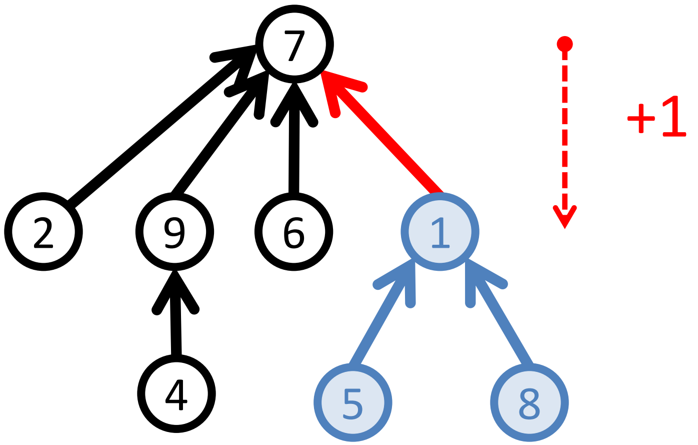
EXTENDED PROOF OF UNION BY SIZE

Union 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.



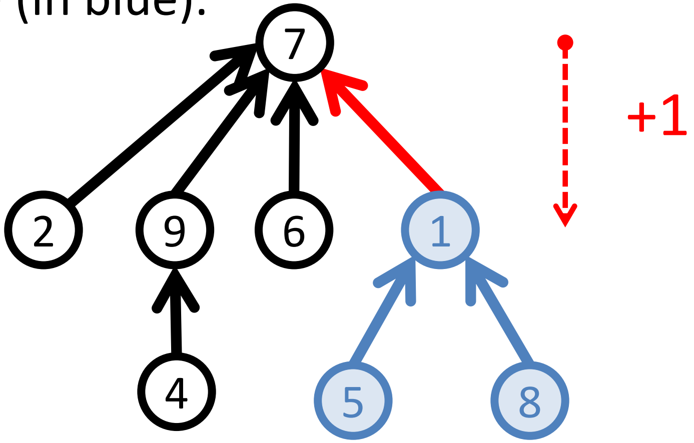
In the example above, we are grafting the blue tree to the root of the back tree because the former is smaller (e.g., this could result from the operation `union(5,4)` for instance). Indeed, the paths from any node of the blue tree to the new root (i.e., the depth) increased by 1 because of the red edge.

Union by size

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

Proof:

The size of the new tree (after union) will have at least twice the number of nodes in the smallest tree (in blue).



In other words, after union, the size of the tree that contains a node whose depth has increased will at least double. It also means that every time the depth of a node increases, the size of tree at least double.

Union by size

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

Proof:

But we can double the size of a tree at most log n times.

Indeed, let k be the number of union operations that increased the depth of that node. After k unions, the number N_k of nodes in that tree is at least: $N_k \geq 2^k$, which implies $\log_2 N_k \geq k$.

The depth of a node can only increase $\log n$ times. Thus, the maximum depth of any node is $O(\log n)$ and the height of the tree is $O(\log n)$.