

# Disjoint Sets: Naive Implementations

Alexander S. Kulikov

Steklov Institute of Mathematics at St. Petersburg  
Russian Academy of Sciences

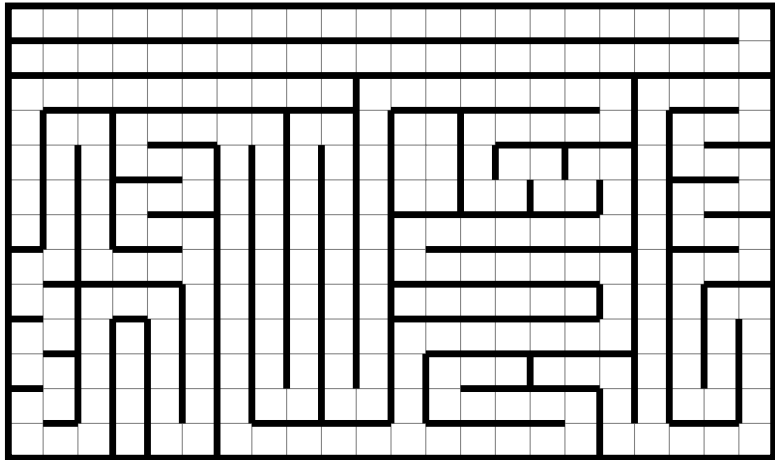
Data Structures  
Data Structures and Algorithms

# Outline

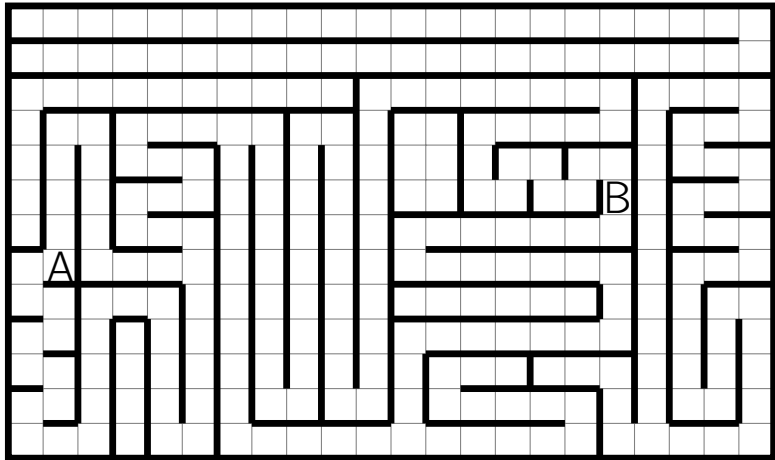
1 Overview

2 Naive Implementations

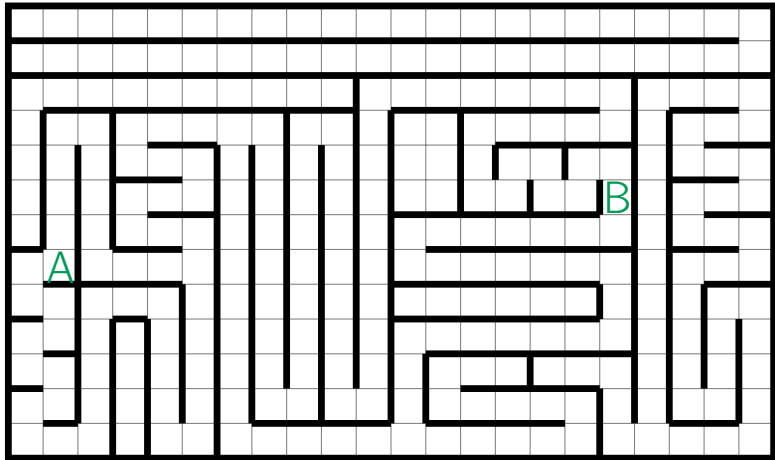
Maze: Is B Reachable from A?



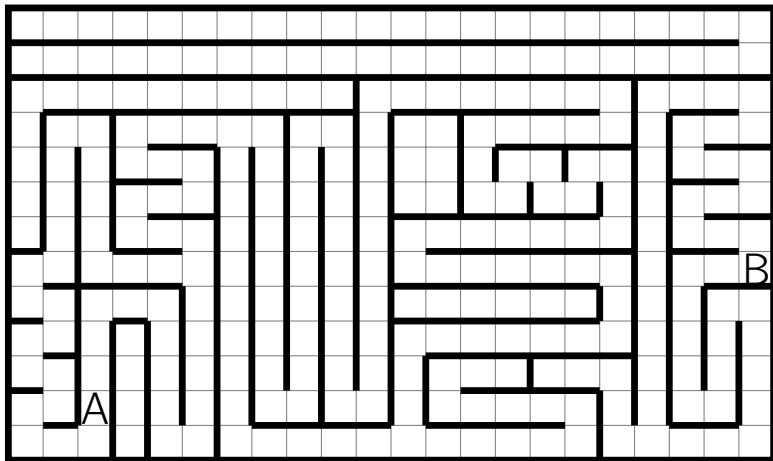
Maze: Is B Reachable from A?



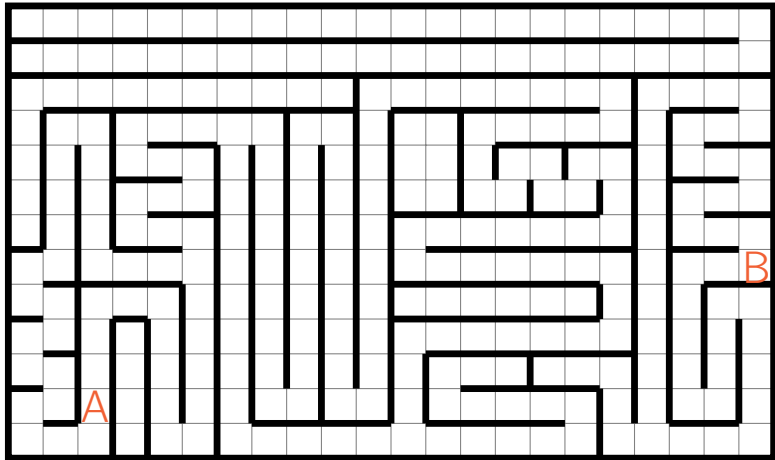
Maze: Is B Reachable from A?



Maze: Is B Reachable from A?



Maze: Is B Reachable from A?



## Definition

A disjoint-set data structure supports the following operations:

- `MakeSet(x)` creates a singleton set  $\{x\}$



## Definition

A disjoint-set data structure supports the following operations:

- `MakeSet( $x$ )` creates a singleton set  $\{x\}$
- `Find( $x$ )` returns ID of the set containing  $x$ :

## Definition

A disjoint-set data structure supports the following operations:

- $\text{MakeSet}(x)$  creates a singleton set  $\{x\}$
- $\text{Find}(x)$  returns ID of the set containing  $x$ :
  - if  $x$  and  $y$  lie in the same set, then  $\text{Find}(x) = \text{Find}(y)$


## Definition

A disjoint-set data structure supports the following operations:

- $\text{MakeSet}(x)$  creates a singleton set  $\{x\}$
- $\text{Find}(x)$  returns ID of the set containing  $x$ :
  - if  $x$  and  $y$  lie in the same set, then  $\text{Find}(x) = \text{Find}(y)$
  - otherwise,  $\text{Find}(x) \neq \text{Find}(y)$

## Definition

A disjoint-set data structure supports the following operations:

- $\text{MakeSet}(x)$  creates a singleton set  $\{x\}$
- $\text{Find}(x)$  returns ID of the set containing  $x$ :
  - if  $x$  and  $y$  lie in the same set, then  $\text{Find}(x) = \text{Find}(y)$
  - otherwise,  $\text{Find}(x) \neq \text{Find}(y)$
- $\text{Union}(x, y)$  merges two sets containing  $x$  and  $y$

## Preprocess(*maze*)

for each cell  $c$  in *maze*:

    MakeSet( $c$ )

for each cell  $c$  in *maze*:

    for each neighbor  $n$  of  $c$ :

        Union( $c, n$ )

## Preprocess(*maze*)

```
for each cell  $c$  in  $maze$ :  
    MakeSet( $c$ )  
for each cell  $c$  in  $maze$ :  
    for each neighbor  $n$  of  $c$ :  
        Union( $c, n$ )
```

## IsReachable( $A, B$ )

```
return Find( $A$ ) = Find( $B$ )
```

# Building a Network

# Building a Network



MakeSet(1)



# Building a Network



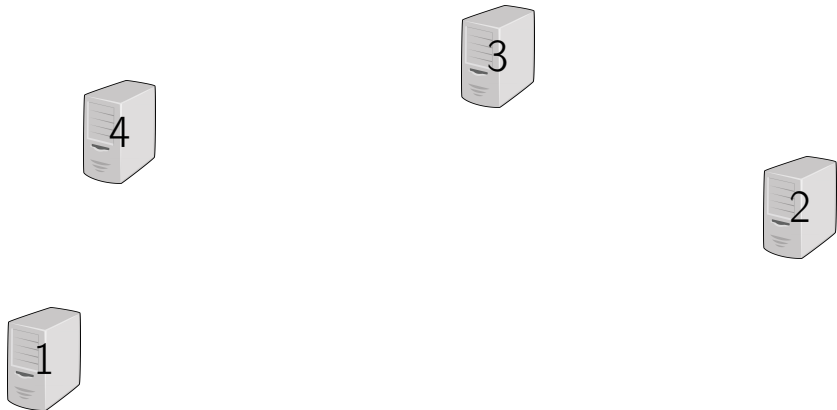
MakeSet(2)

# Building a Network



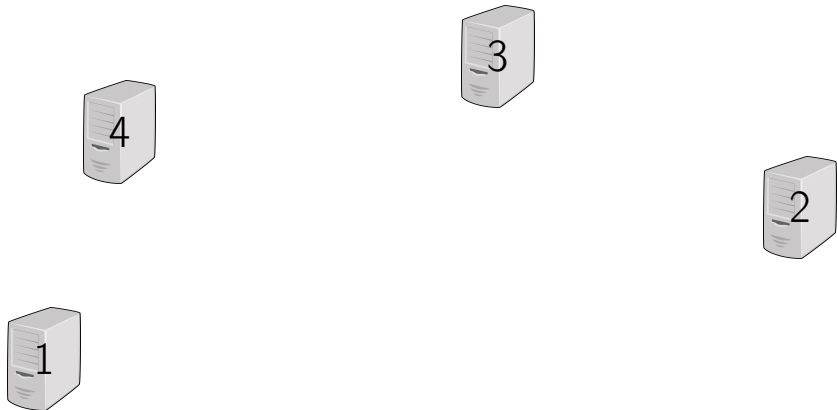
MakeSet(3)

# Building a Network



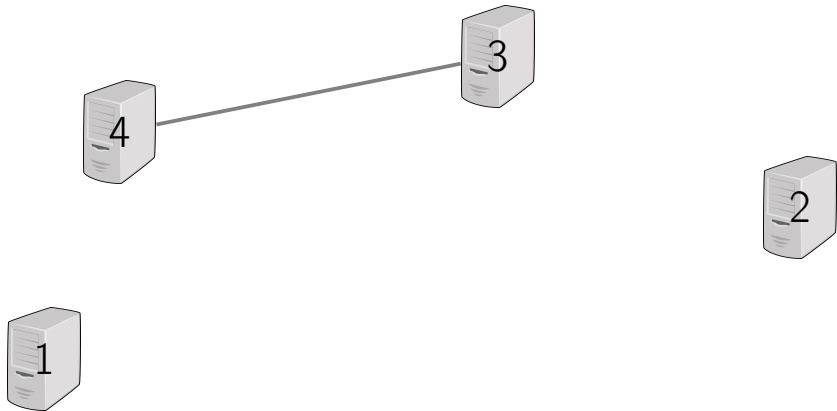
MakeSet(4)

# Building a Network

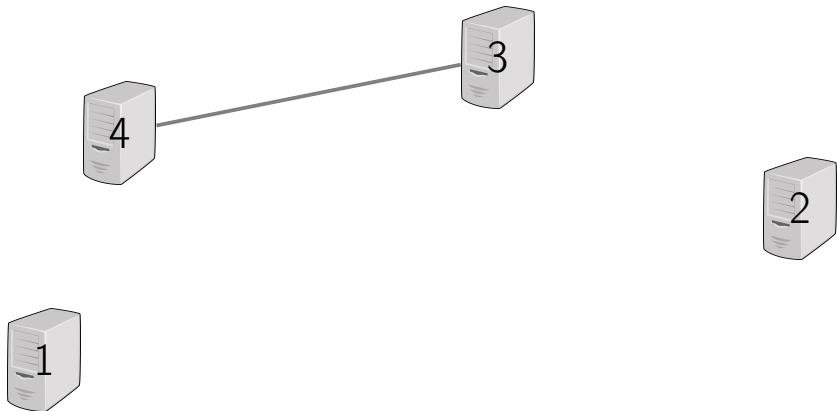


$\text{Find}(1) = \text{Find}(2) \rightarrow \text{False}$

# Building a Network

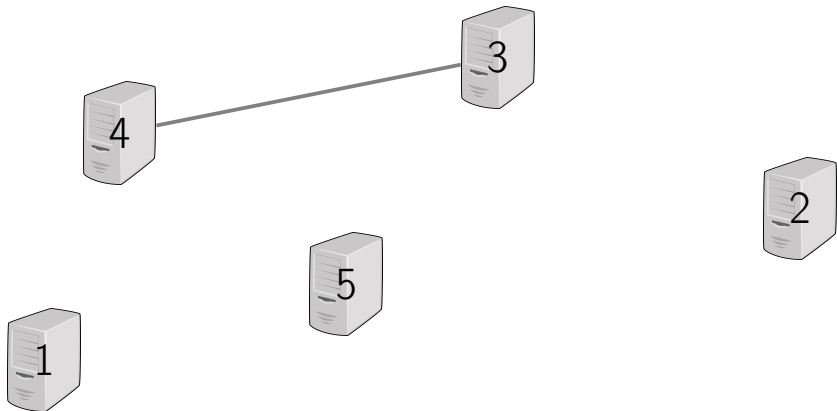


# Building a Network



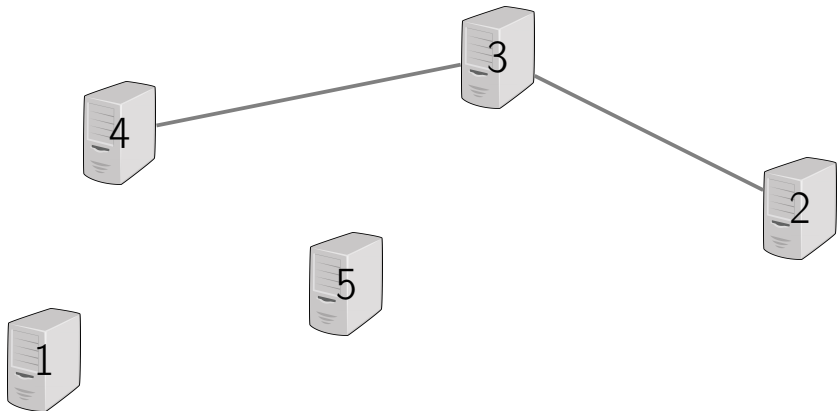
`Union(3, 4)`

# Building a Network



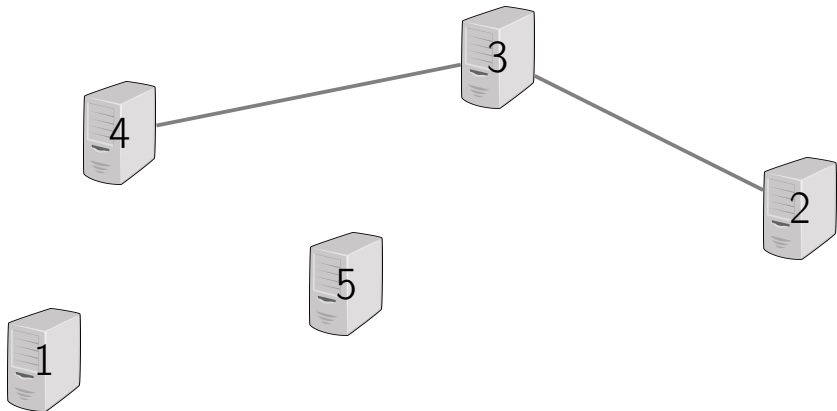
MakeSet(5)

# Building a Network



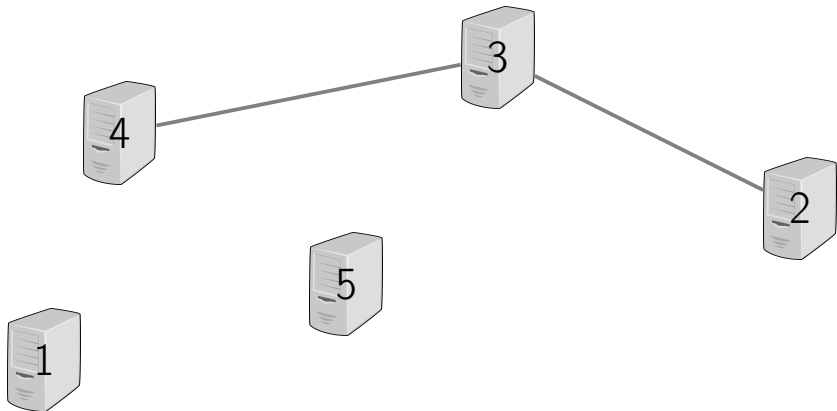


# Building a Network



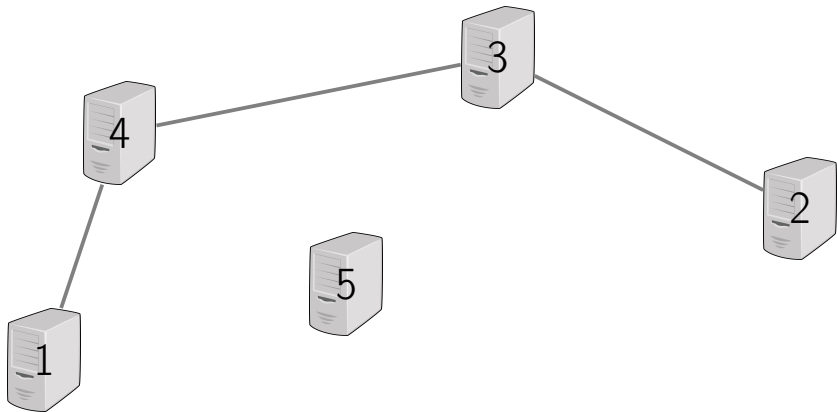
Union(3, 2)

# Building a Network

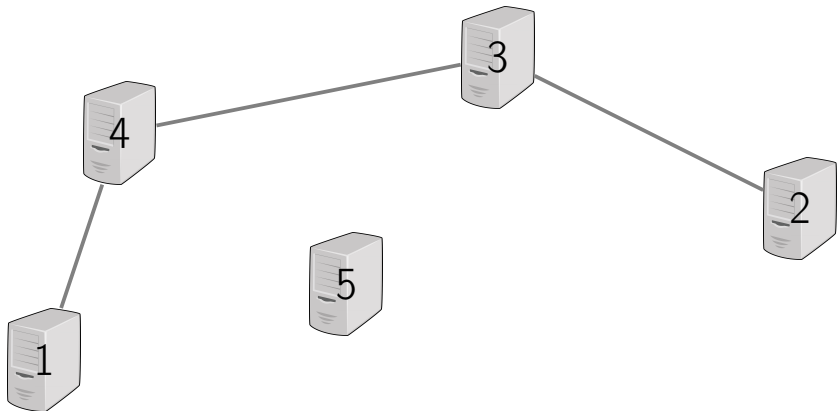


$\text{Find}(1) = \text{Find}(2) \rightarrow \text{False}$

# Building a Network

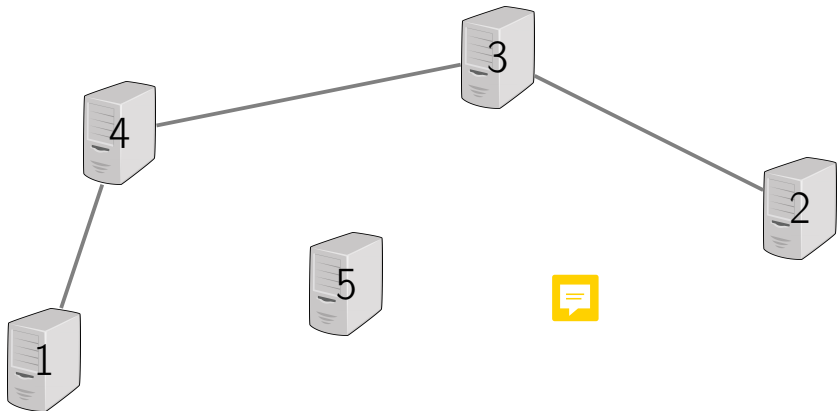


# Building a Network



Union(1, 4)

# Building a Network



$\text{Find}(1) = \text{Find}(2) \rightarrow \text{True}$

# Outline

- 1 Overview
- 2 Naive Implementations

For simplicity, we assume that our  $n$  objects are just integers  $1, 2, \dots, n$ .

# Using the Smallest Element as ID

- Use the smallest element of a set as its ID



# Using the Smallest Element as ID

- Use the smallest element of a set as its ID
- Use array `smallest[1...n]`:  
`smallest[i]` stores the smallest element in the set  $i$  belongs to

## Example

$\{9, 3, 2, 4, 7\}$     $\{5\}$     $\{6, 1, 8\}$

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

MakeSet( $i$ )

$\text{smallest}[i] \leftarrow i$

Find( $i$ )

return  $\text{smallest}[i]$

MakeSet( $i$ )

$\text{smallest}[i] \leftarrow i$

Find( $i$ )

return  $\text{smallest}[i]$

Running time:  $O(1)$

## Union( $i, j$ )

$i\_id \leftarrow \text{Find}(i)$

$j\_id \leftarrow \text{Find}(j)$

if  $i\_id = j\_id$ :

    return

$m \leftarrow \min(i\_id, j\_id)$

for  $k$  from 1 to  $n$ :

    if  $\text{smallest}[k] \in \{i\_id, j\_id\}$ :

$\text{smallest}[k] \leftarrow m$

## Union( $i, j$ )

```
 $i\_id \leftarrow \text{Find}(i)$   
 $j\_id \leftarrow \text{Find}(j)$   
if  $i\_id = j\_id$ :  
    return  
 $m \leftarrow \min(i\_id, j\_id)$   
for  $k$  from 1 to  $n$ :  
    if  $\text{smallest}[k] \in \{i\_id, j\_id\}$ :  
         $\text{smallest}[k] \leftarrow m$ 
```

Running time:  $O(n)$

- Current bottleneck: Union

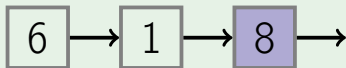
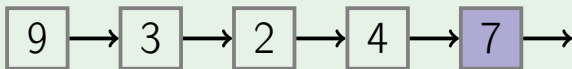
- Current bottleneck: Union
- What basic data structure allows for efficient merging?



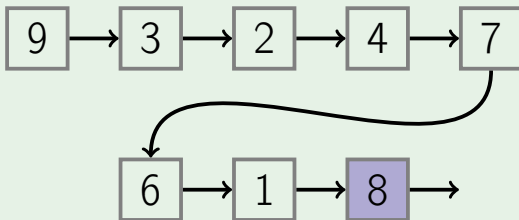
- Current bottleneck: Union
- What basic data structure allows for efficient merging?
- Linked list!

- Current bottleneck: Union
- What basic data structure allows for efficient merging?
- Linked list!
- Idea: represent a set as a linked list, use the list tail as ID of the set

## Example: merging two lists



## Example: merging two lists



- Pros:

- Pros:

- Running time of Union is  $O(1)$

- Pros:

- Running time of `Union` is  $O(1)$
- Well-defined ID

- Pros:
  - Running time of `Union` is  $O(1)$
  - Well-defined ID
- Cons:



- Pros:

- Running time of Union is  $O(1)$
- Well-defined ID

- Cons:

- Running time of Find is  $O(n)$  as we need to traverse the list to find its tail

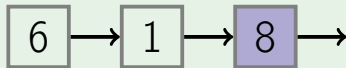
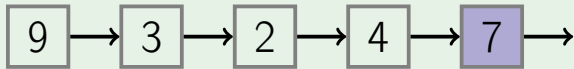
- Pros:

- Running time of Union is  $O(1)$
- Well-defined ID

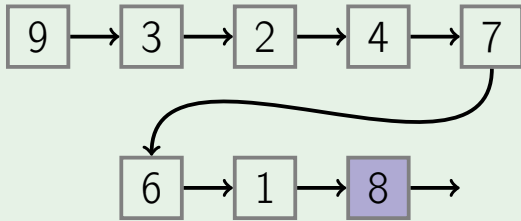
- Cons:

- Running time of Find is  $O(n)$  as we need to traverse the list to find its tail
- Union( $x, y$ ) works in time  $O(1)$  **only** if we can get the tail of the list of  $x$  and the head of the list of  $y$  in constant time!

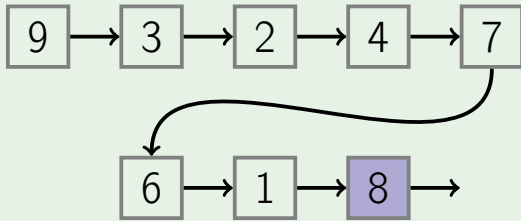
## Example: merging two lists



## Example: merging two lists

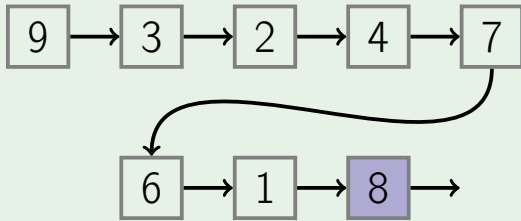


## Example: merging two lists



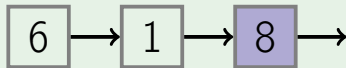
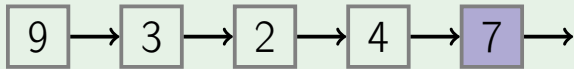
`Find(9)` goes through all elements

## Example: merging two lists

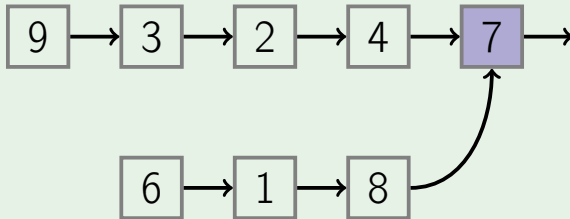


can we merge in a different way?

## Example: merging two lists

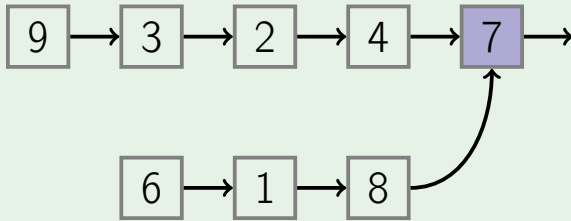


## Example: merging two lists



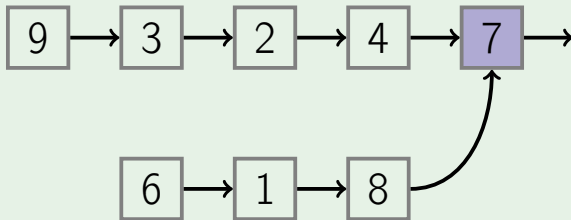


## Example: merging two lists



instead of a list we get a tree

## Example: merging two lists



we'll see that representing sets as trees gives a very efficient implementation: nearly constant amortized time for all operations