Disjoint Sets: Naive Implementations

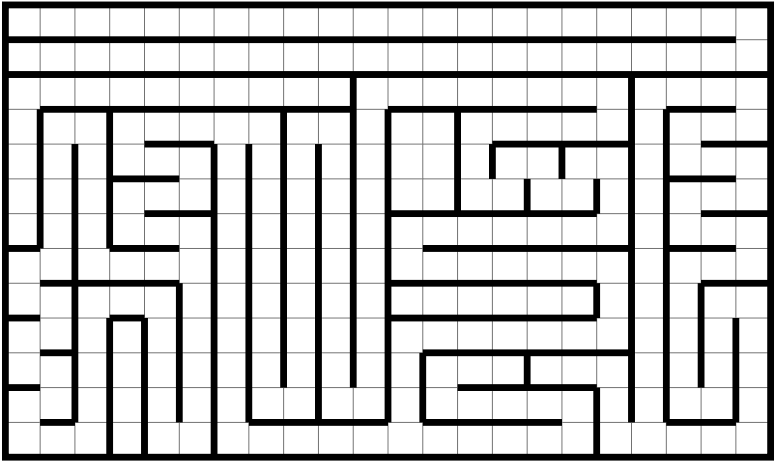
Alexander S. Kulikov

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

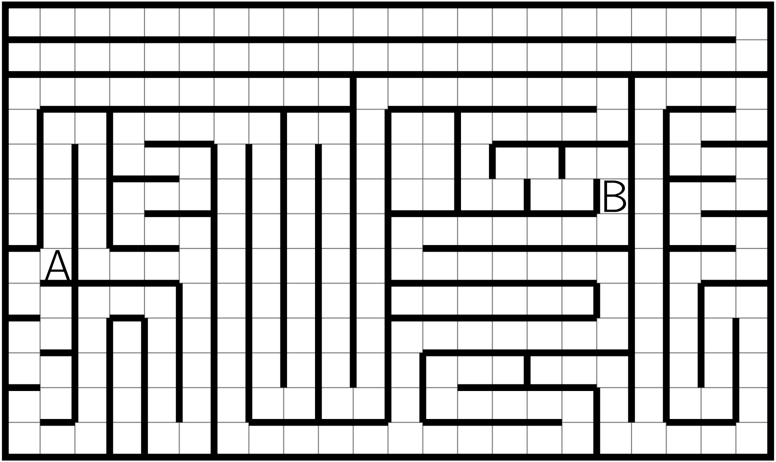
Data Structures
Data Structures and Algorithms

Outline

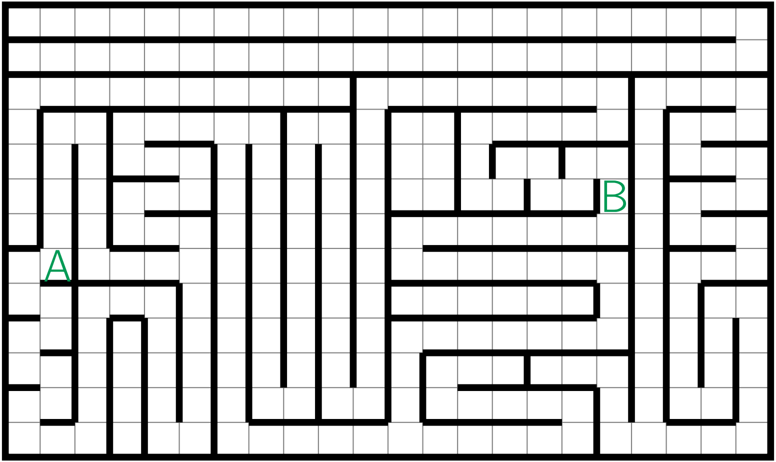
Maze: Is B Reachable from A?



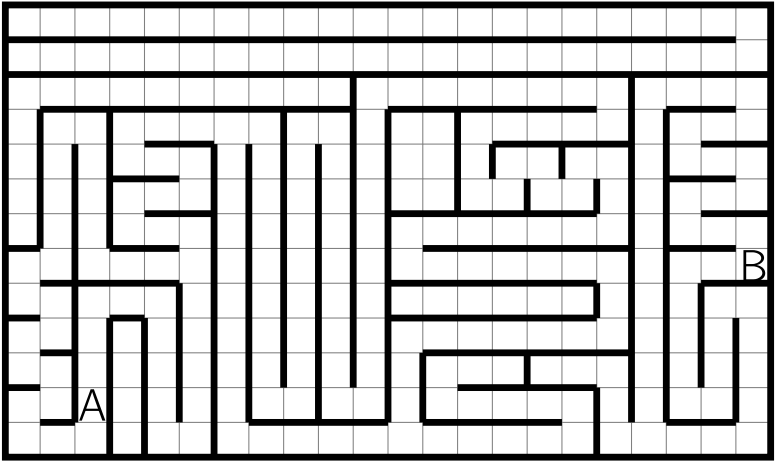
Maze: Is B Reachable from A?



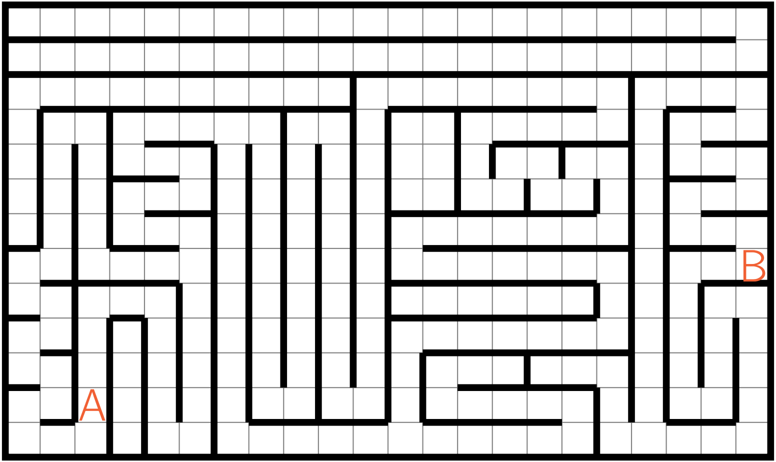
Maze: Is B Reachable from A?



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:

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

Definition

A disjoint-set data structure supports the following operations:

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

Definition

A disjoint-set data structure supports the following operations:

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

$\texttt{Preprocess}(\mathit{maze})$

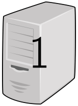
```
for each cell c in maze:
  MakeSet(c)
for each cell c in maze:
  for each neighbor n of c:
    Union(c, n)
```

$\texttt{IsReachable}(A, B)$

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

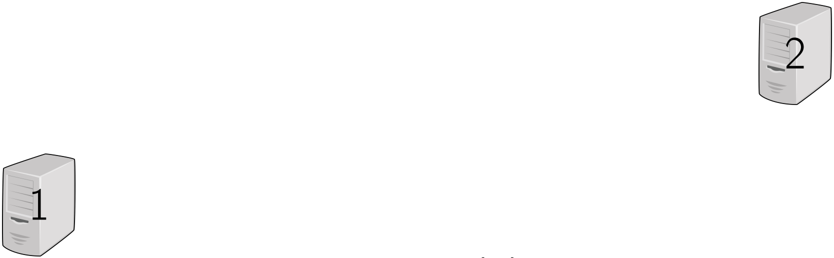
Building a Network

Building a Network



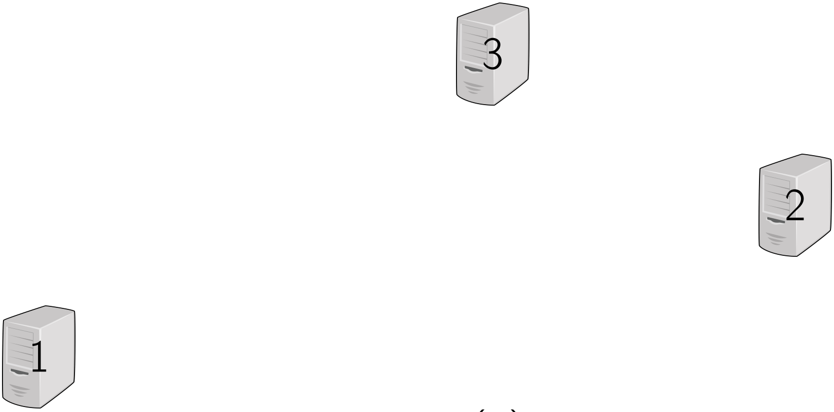
`MakeSet(1)`

Building a Network



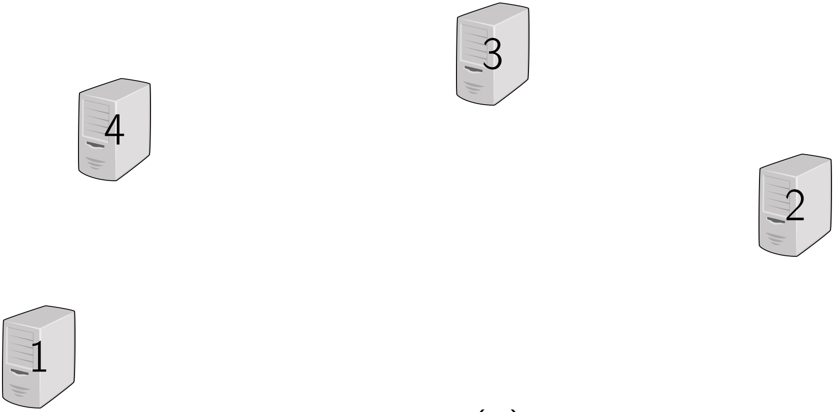
`MakeSet(2)`

Building a Network



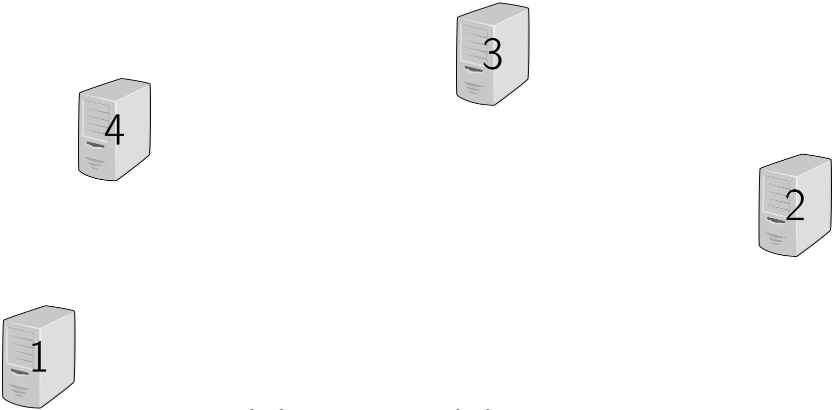
`MakeSet(3)`

Building a Network



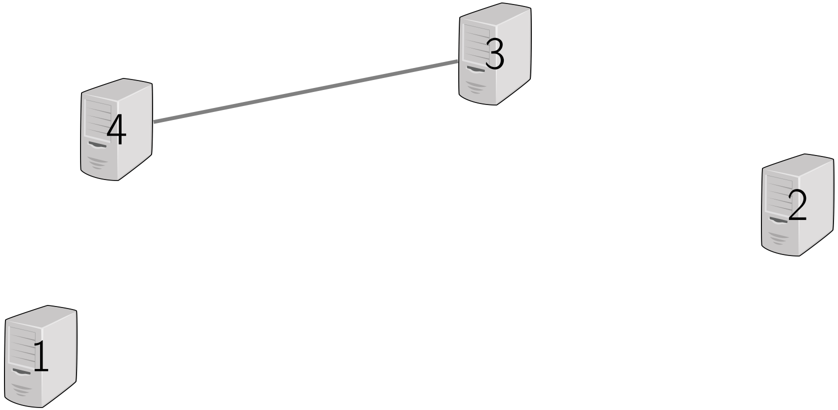
`MakeSet(4)`

Building a Network

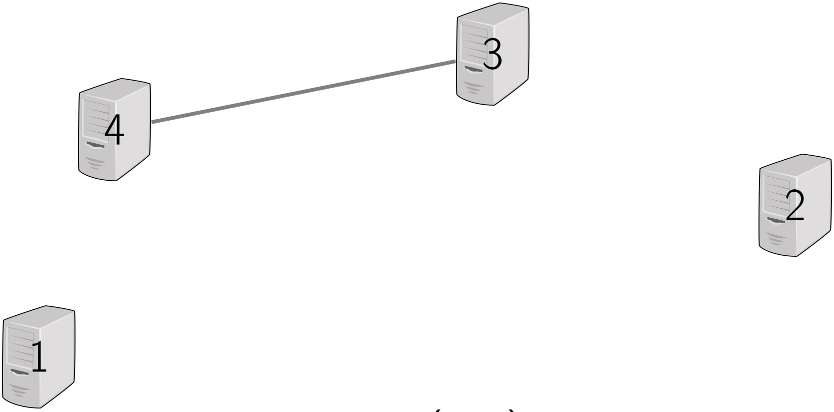


$$\mathtt{Find}(1) = \mathtt{Find}(2) \rightarrow \mathtt{False}$$

Building a Network

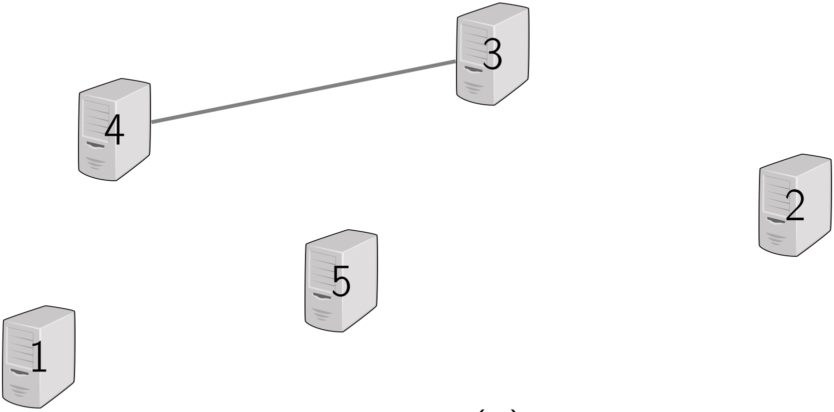


Building a Network



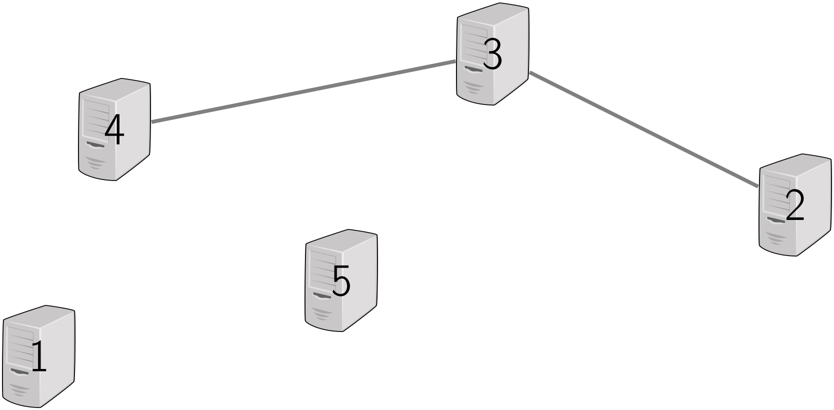
`Union(3,4)`

Building a Network

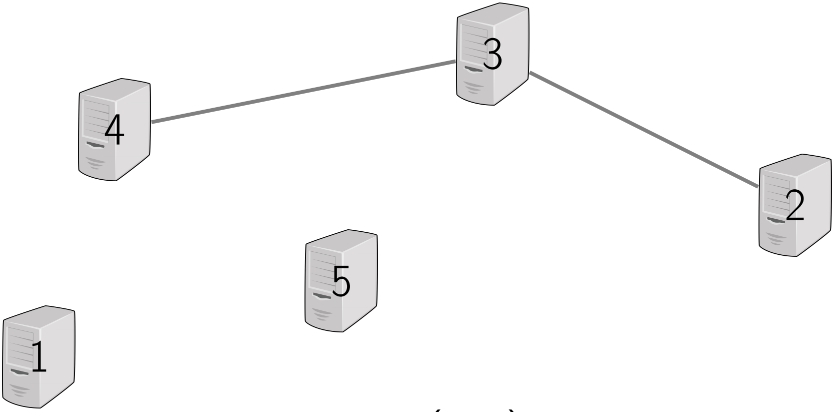


`MakeSet(5)`

Building a Network

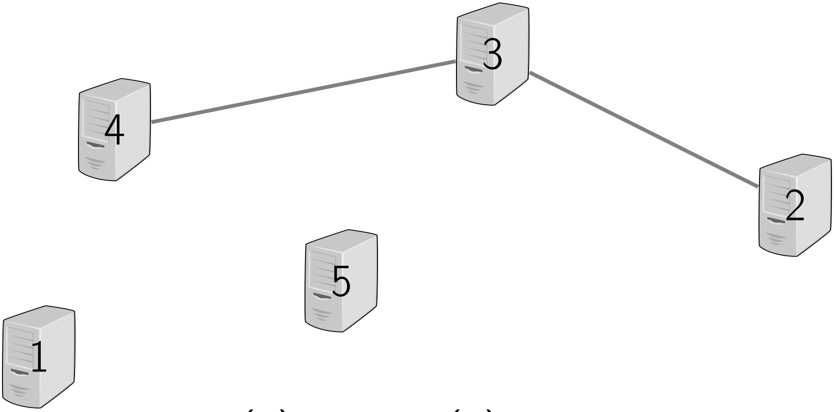


Building a Network



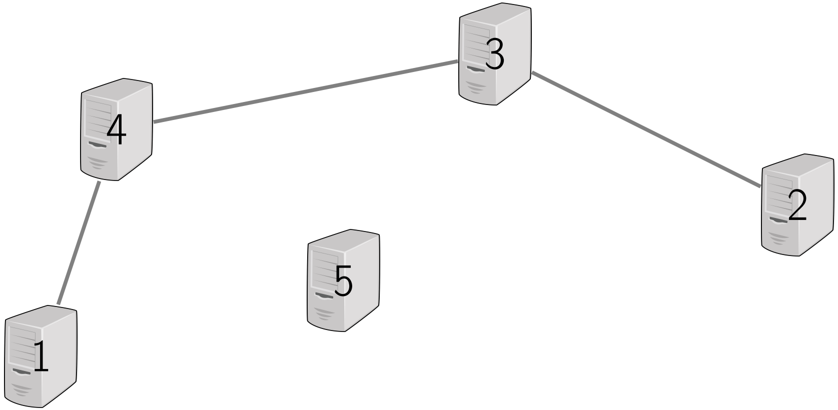
`Union(3,2)`

Building a Network

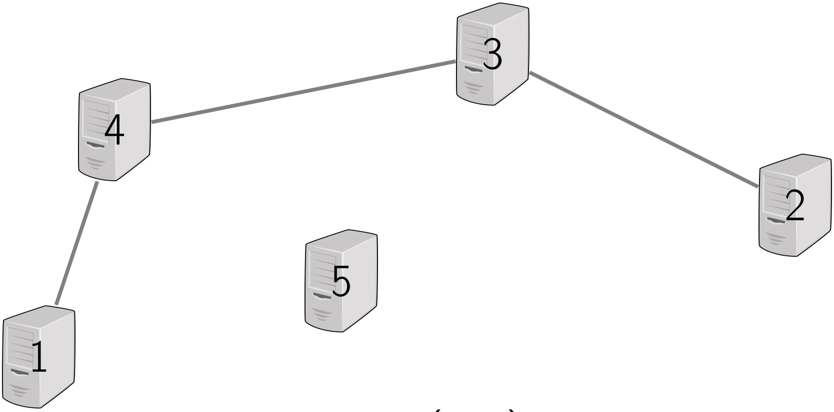


$$\texttt{Find}(1) = \texttt{Find}(2) \to \texttt{False}$$

Building a Network

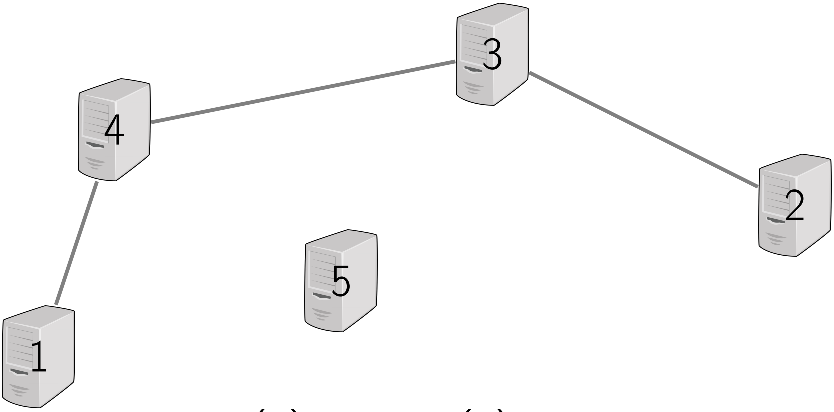


Building a Network



`Union(1,4)`

Building a Network



$$\texttt{Find}(1) = \texttt{Find}(2) \to \texttt{True}$$

Outline

1. Overview
2. Naive Implementations

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

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`$[1 \dots n]$: `smallest`$[i]$ stores the smallest element in the set i belongs to

Example

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

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

MakeSet(i)

smallest[i] $\leftarrow i$

Find(i)

return smallest[i]

MakeSet(i)

```
smallest[i] ← i
```

Find(i)

```
return smallest[i]
```

Running time: $O(1)$

Union(i, j)

```
i_id ← Find(i)
j_id ← Find(j)
if i_id = j_id:
  return
m ← min(i_id, j_id)
for k from 1 to n:
  if smallest[k] in {i_id, j_id}:
    smallest[k] ← m
```

$\texttt{Union}(i, j)$

```
i_id ← Find(i)
j_id ← Find(j)
if i_id = j_id:
    return
m ← min(i_id, j_id)
for k from 1 to n:
    if smallest[k] in {i_id, j_id}:
        smallest[k] ← m
```

Running time: $O(n)$

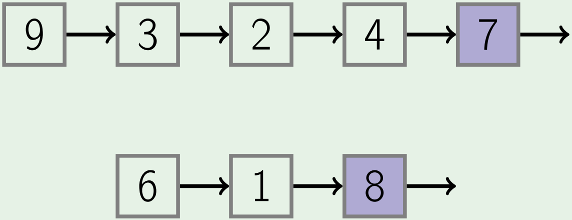
- Current bottleneck: `Union`

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

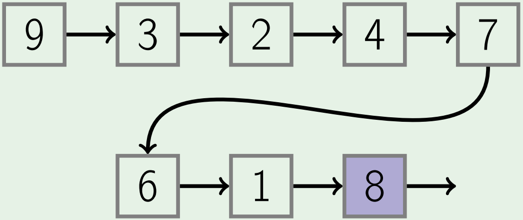
- Current bottleneck: `Union`
- What basic data structure allows for efficient merging?
- Linked list!

- 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 $O(1)$

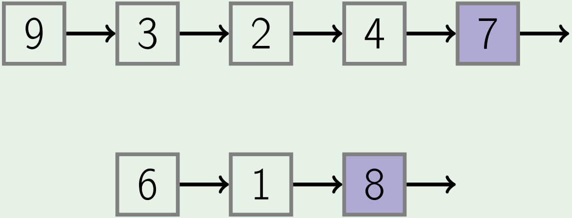
- Pros:
  - Running time of `Union` is $O(1)$
  - Well-defined ID

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

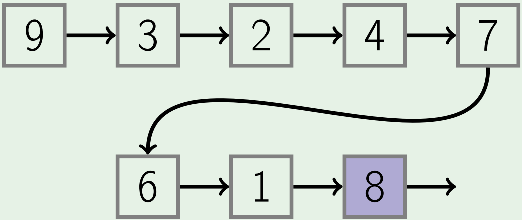
- Pros:
  - Running time of `Union` is $O(1)$
  - Well-defined ID
- Cons:
  - Running time of `Find` is $O(n)$ as we need to traverse the list to find its tail

- 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!

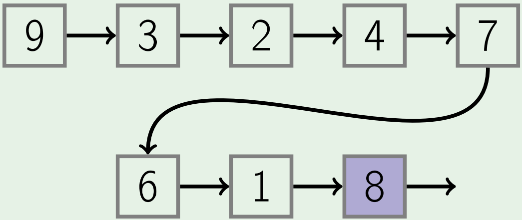
Example: merging two lists



Example: merging two lists

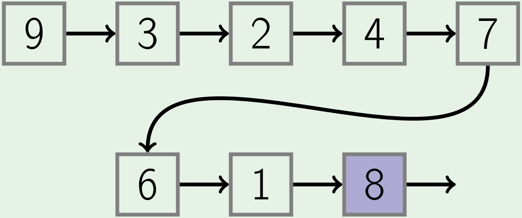


Example: merging two lists



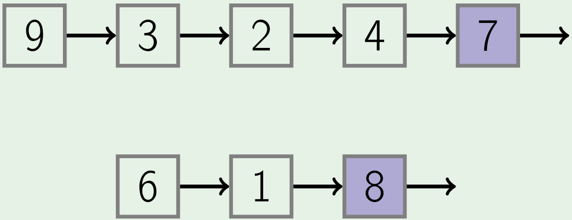
`Find`(9) goes through all elements

Example: merging two lists

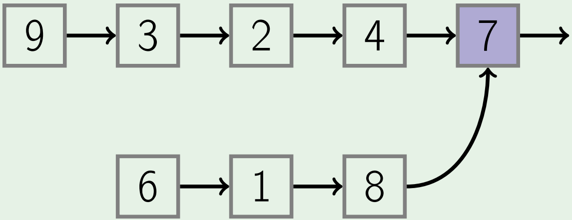


can we merge in a different way?

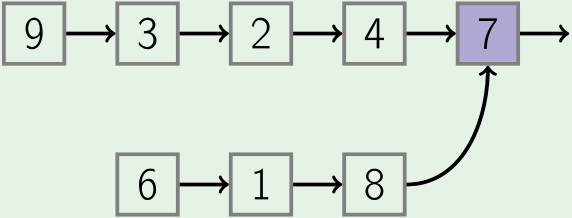
Example: merging two lists



Example: merging two lists

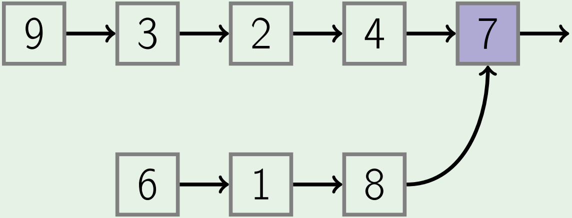


Example: merging two lists



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