Algorithms and Complexity

Connectivity

Paolo Camurati and Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

Online Connectivity

- Problem definition
  - Given a set of N objects (from 0 to N-1)
  - Accept as inputs a sequence of integer pairs (p, q)
    - Where p∈[0, N-1] and q∈[0, N-1]
    - With the meaning that the pair (p, q) indicates that p must be connected to q
  - Produce as outputs
    - Null, if p and q are already connected (directly or **indirectly**)
    - The same pair (p, q), otherwise

Online Connectivity

- In other words we want to be able to
  - Understand whether two objects are connected (directly or undirectly)
  - Connect objects in case they are not connected
- This implies that we should be able to perform two possible operations
  - Find query
    - To find whether two objets are connected
  - Union command
    - To connects two unconnected objects

Online Connectivity

- Notice that
  - We **do not want** to know the path which connect two objects
  - We just want to know **whether** such a path exists or not

This is a problem we will study with graphs

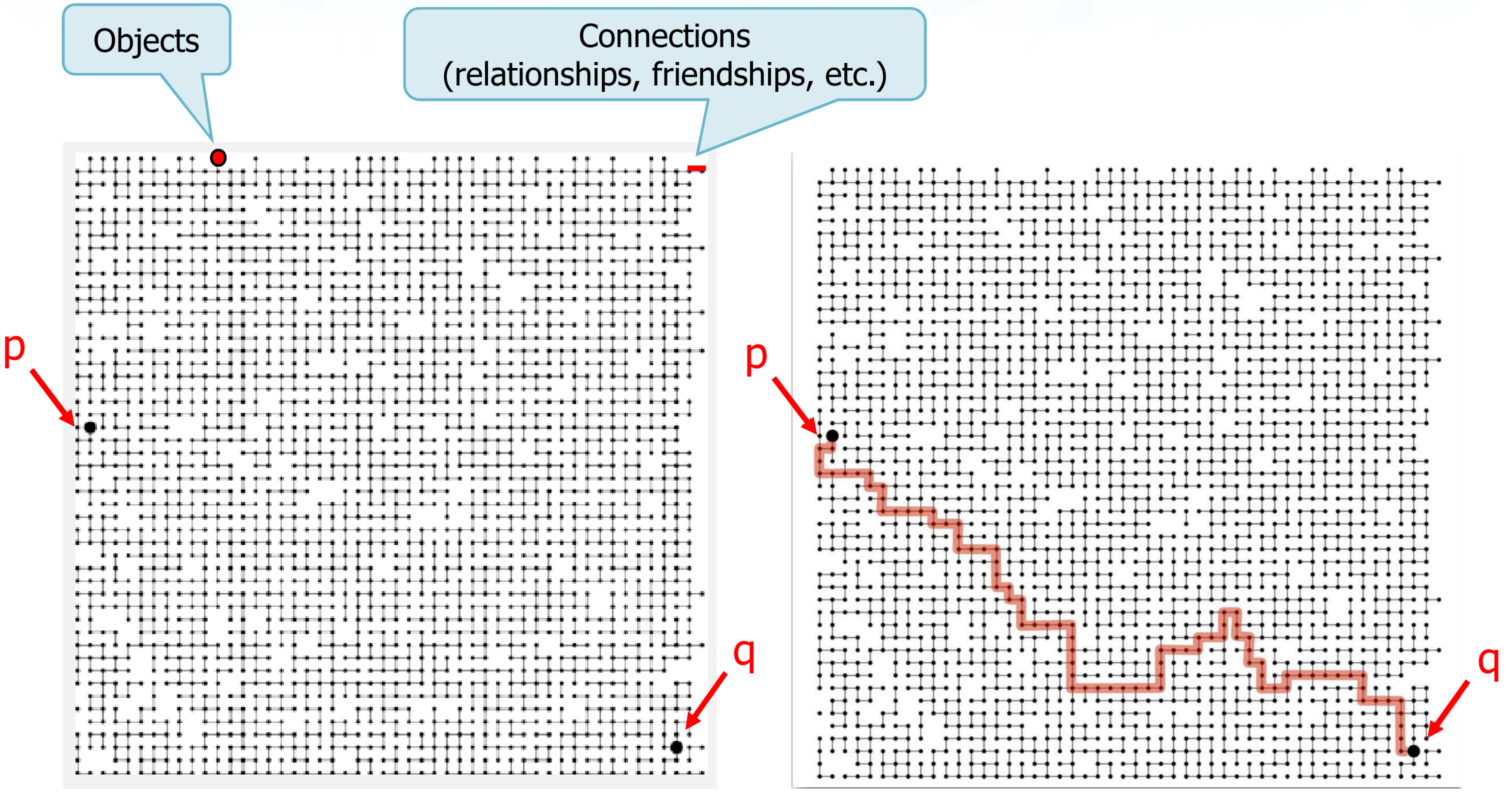
… but we wanto to be as efficient as possible

Applications

- Connectivity has many possibile applications
  - Computer networks
    - Integers p and q represent computers
    - (p, q) connections between computers
  - Electrical networks
    - Integers p and q represent contact points
    - (p, q) wires
  - Social networks
    - Integers p and q represent subscribers
    - (p, q) relationhips
  - …

Applications: An Example

- Is there a path connecting p and q?



Objects

Connections
(relationships, friendships, etc.)

- Yes !

Modeling the objects

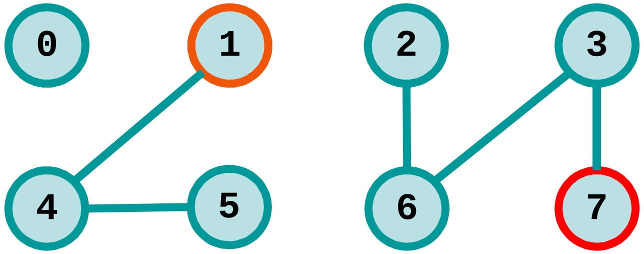
- Applications involve manipulating objects of all types
  - Pixels in a digital photo
  - Computers in a network
  - Friends in a social network
  - Transistors in a computer chip
  - …
- When programming, it is convenient to map objects (whatever they are) to integers
  - To represent **N** object use integer from **0** to **N–1**
  - Use integers as array index

Modeling the connections

- Connectivity is an equivalence relation
  - Reflexive
    - p is connected to p
  - Symmetrical
    - If p is connected to q, q is connected to p
  - Transitive
    - If p is connected to q and q is connected to r, then p is connected to r
- Connectivity can be represented using **connected component**

Modeling the connections

- A **connected component** is a
  - Maximal subset of mutually reachable nodes
  - Where no element is connected to an element outside its connected component



3 connected components
{0}, {1, 4, 5},{2, 3, 6, 7}

Graph: Data structure representing elements (nodes or vertices) and their relationships or connections (edges)

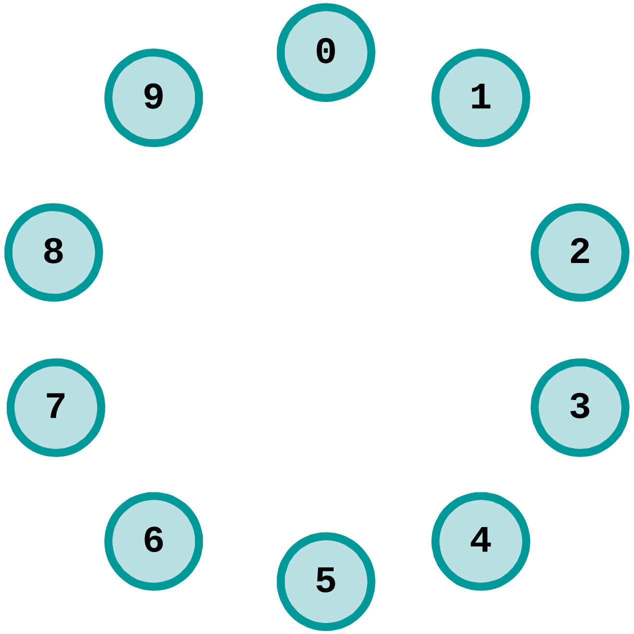
A connected component may have a **leader**, i.e., a **class member** representing all elements withing the component

Implementing the operations

- Given all connected components the
  - Find query
    - Check if two objects are in the same component
  - Union command
    - Replace two connected components with their union

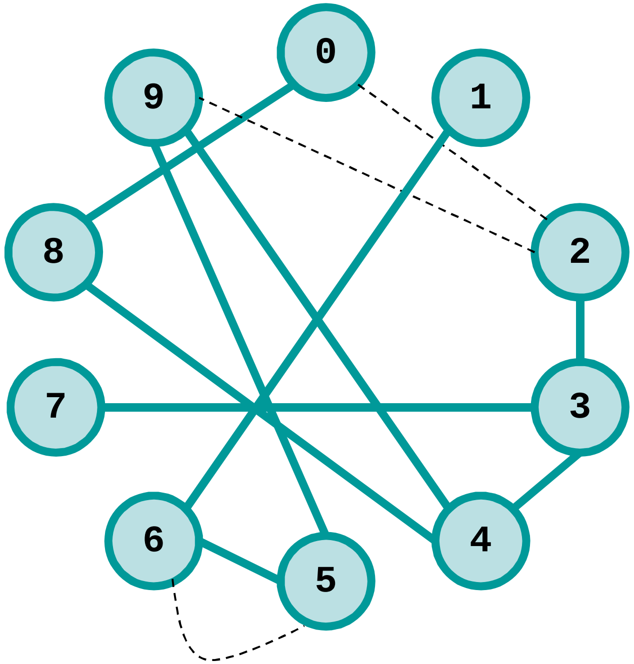
Example

- Input Pairs: 3-4, 4-9, 8-0, 2-3, 5-6, 2-9, 5-9, 7-3, 4-8, 6-5, 0-2, 6-1



Solution

- **Input Pairs:** 3-4, 4-9, 8-0, 2-3, 5-6, 2-9, 5-9, 7-3, 4-8, 6-5, 0-2, 6-1
- **Output:** 3-4, 4-9, 8-0, 2-3, 5-6, - , 5-9, 7-3, 4-8, - , - , 6-1



Trivial solutions

- Trivial solutions
  - For each pair (p, q)
    - Check the connection by visiting the network
    - Search q starting from p (or vice-versa)
    - Cons
      - May require a visit of the entire network for each new pair
  - For each node p
    - Store all nodes reachable (transitive closure)
    - Cons
      - May need a memory size quadratic in the number of nodes of the network

Target solution

- Design efficient data structure for union-find
- Keep into account that
  - The number of objects N can be huge
  - The number of operations M can be huge
- Find queries and union commands may be intermixed
- We will analyze two algorithms
  - An eager approach (quick-find)
  - A lazy approach (quick-union)

Quick-find

Slow union

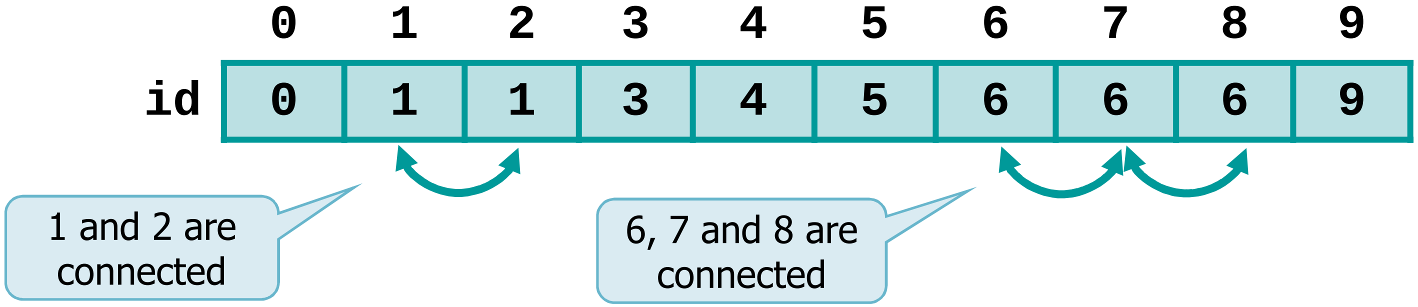
- Hypothesis
  - We do not have the graph (but we can use it to reason on the problem)
  - We work pair by pair
    - We keep and update information necessary to find out connectivity
      - Sets S of connected pairs
      - Initially S includes as many sets as nodes, each node being connected just with itself
  - Abstract operations
    - find: find the set an object belongs to
    - union: merge two sets

Quick-find logic

- Represent sets S_i of connected pairs with array id
  - Initially all objects point to themselves
    - id[i] = i (no connection)

	0	1	2	3	4	5	6	7	8	9
id	0	1	2	3	4	5	6	7	8	9

  - Find
    - If p and q are connected, id[p] = id[q]
    - Do nothing and move to the next pair



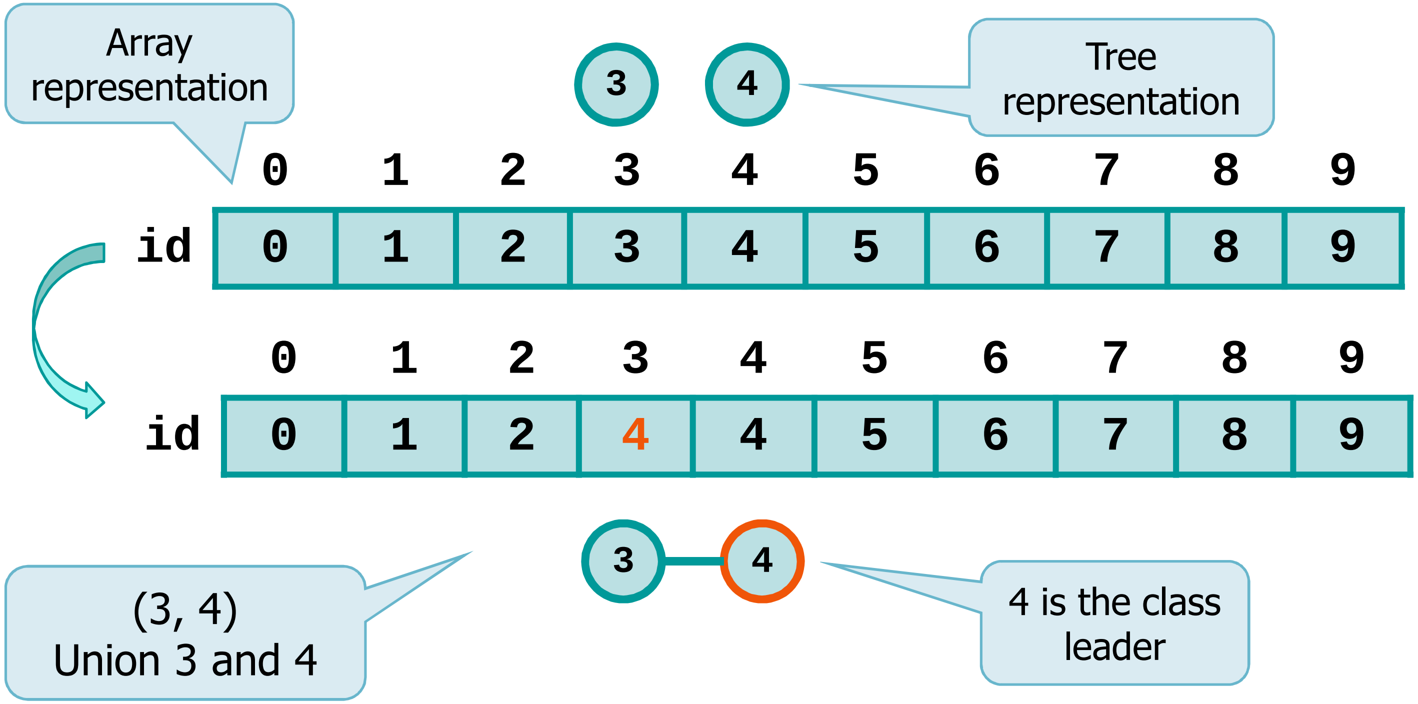
	0	1	2	3	4	5	6	7	8	9
id	0	1	1	3	4	5	6	6	6	9

1 and 2 are connected

6, 7 and 8 are connected

Quick-find logic

- Union
  - If p and q are not connected (i.e., id[p] ≠ id[q])
  - Scan the array, replacing id[p] values with id[q] values



Array representation

Tree representation

	0	1	2	3	4	5	6	7	8	9
id	0	1	2	3	4	5	6	7	8	9

	0	1	2	3	4	5	6	7	8	9
id	0	1	2	4	4	5	6	7	8	9

(3, 4)
Union 3 and 4

4 is the class leader

Implementation

- Repeat for all pairs (p, q)
  - Read the pair (p, q)
  - Execute find on p
    - Find an connected component C_p such that $p \in C_p$
  - Execute find on q
    - Find an connected component C_q such that $q \in C_q$
  - If C_p and C_q coincide
    - Do nothing and move on to the next pair
      - The pair is already connected
    - Otherwise, execute union on C_p and C_q

Implementation

```
#include <stdio.h>

#define N 10000

int main() {
  int i, t, p, q, id[N];
  for(i=0; i<N; i++) {
    id[i] = i;
  }
  do {
    printf ("Input pair p q: ");
    scanf ("%d %d", &p, &q);
    if (id[p] != id[q]) {
      for (t=id[p], i=0; i<N; i++) {
        if (id[i] == t)
          id[i] = id[q];
      }
      printf ("%d-%d\n", p, q);
    }
  } while (p!=q);
}
```

If id[p]==id[q] then p and q are already connected. Nothing has to be done.

Store id[p] into variable t to avoid a nasty bug

Union: replace all id[p] with id[q]

Go-on until p!=q

Example

	0	1	2	3	4	5	6	7	8	9
	0	1	2	3	4	5	6	7	8	9
3,2										
6,4										
3,4										
5,2										
6,2										
0,8										
9,1										
3,8										
6,4										
0,5										

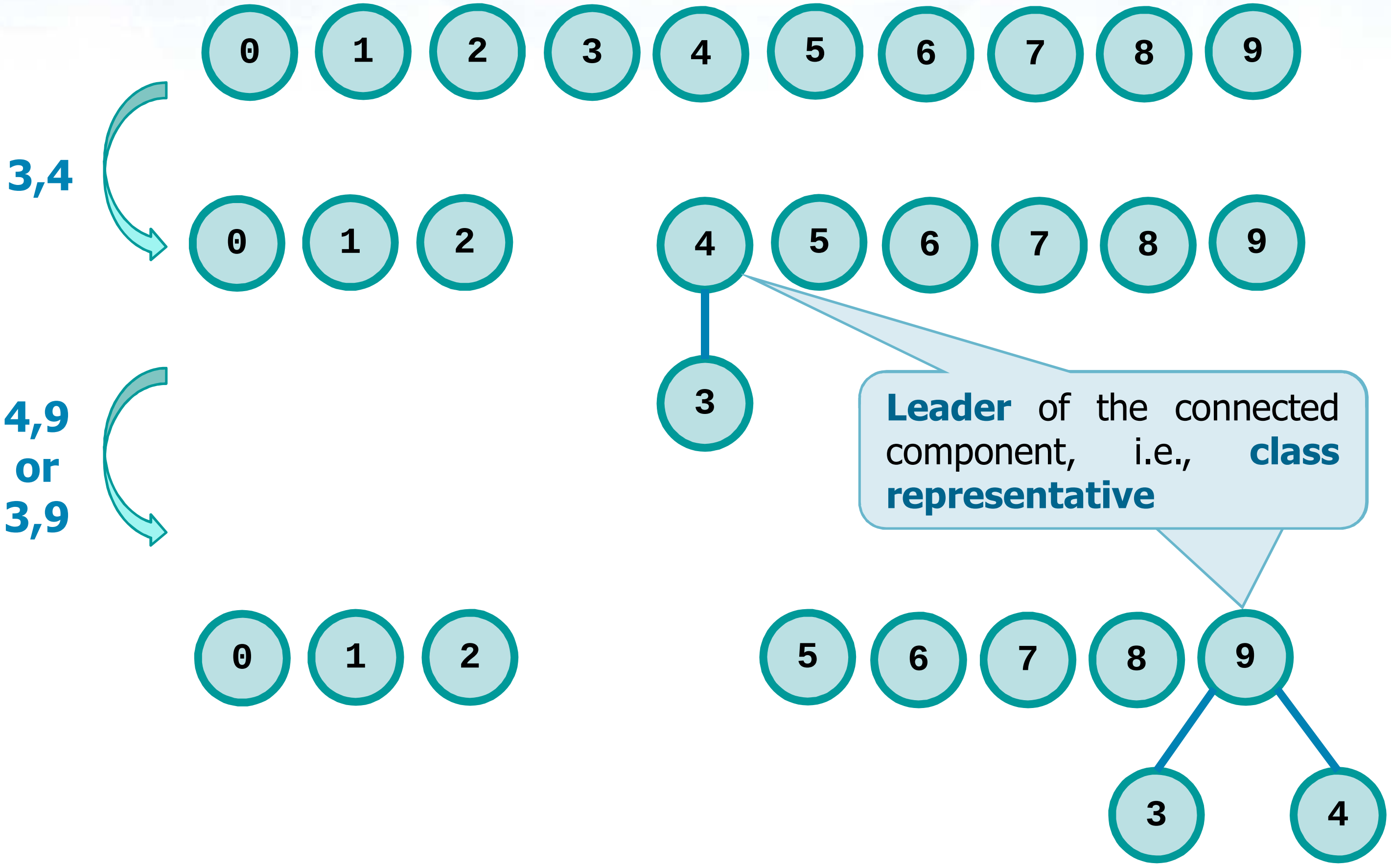
Solution

	0	1	2	3	4	5	6	7	8	9
	0	1	2	3	4	5	6	7	8	9
3,2	0	1	2	2	4	5	6	7	8	9
6,4	0	1	2	2	4	5	4	7	8	9
3,4	0	1	4	4	4	5	4	7	8	9
5,2	0	1	4	4	4	4	4	7	8	9
6,2	0	1	4	4	4	4	4	7	8	9
0,8	8	1	4	4	4	4	4	7	8	9
9,1	8	1	4	4	4	4	4	7	8	1
3,8	8	1	8	8	8	8	8	7	8	1
6,4	8	1	8	8	8	8	8	7	8	1
0,5	8	1	8	8	8	8	8	7	8	1

Tree representation

- Some objects represent the set they belong to
- Other objects point to the object that represents the set they belong to
- For each pair p,q
  - Every id[p] becomes id[q]
  - Every node i with id equal to id[p] goes under node id[q]

Tree representation

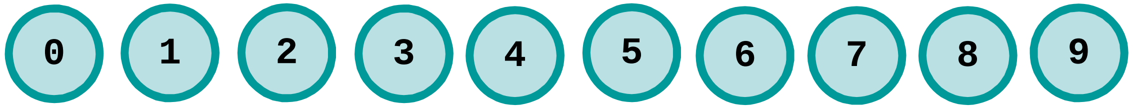


Leader of the connected component, i.e., **class representative**

Exercise

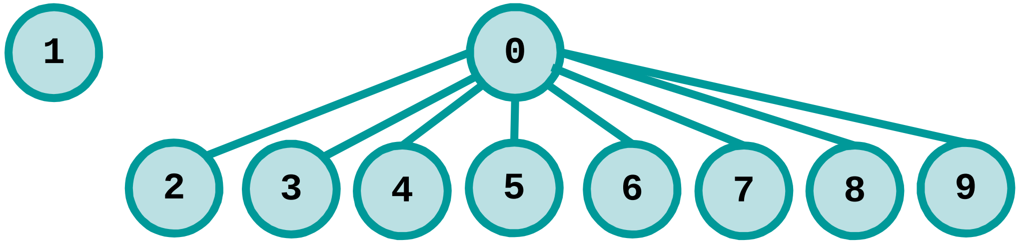
- Input Pairs: 3-4, 4-9, 8-0, 2-3, 5-6, 2-9, 5-9, 7-3, 4-8, 6-5, 0-2

	0	1	2	3	4	5	6	7	8	9
id	0	1	2	3	4	5	6	7	8	9



Solution

	0	1	2	3	4	5	6	7	8	9
id	0	1	0	0	0	0	0	0	0	0



Complexity

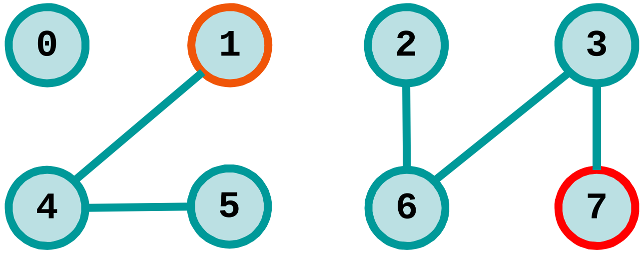
```
do {
  ...
  if (id[p] != id[q]) {
    for (t=id[p], i=0; i<N; i++) {
      if (id[i] == t)
        id[i] = id[q];
    }
    printf ("%d-%d\n", p, q);
  }
} while (p!=q);
```

- Find
  - Reference to id[i]
  - Unit cost
- Union
  - Scan array to replace p values with q values
  - Linear (in the array size) cost
- Overall
  - Number of operations related to
    - # pairs · array size = M · N
  - Quadratic cost
  - Very slow for real-time applications

Quick-union

Not too quick find

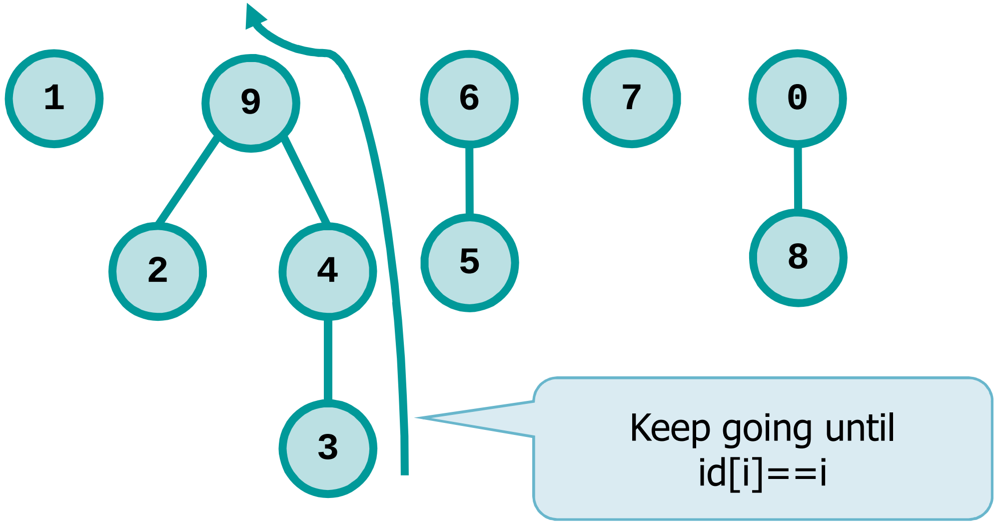
- As with quick-find, represent sets S_i of connected pairs with an array id
  - Initially all objects point to themselves
    - id[i] = i (no connection)
  - Each object points either to an object to which it is connected or to itself (no loops)
    - We write (id[i])* to indicate id[id[id[… id[i]]]], going on until id[i]==i
  - If objects i are j connected
    - (id[i])* = (id[j])*



Quick-union

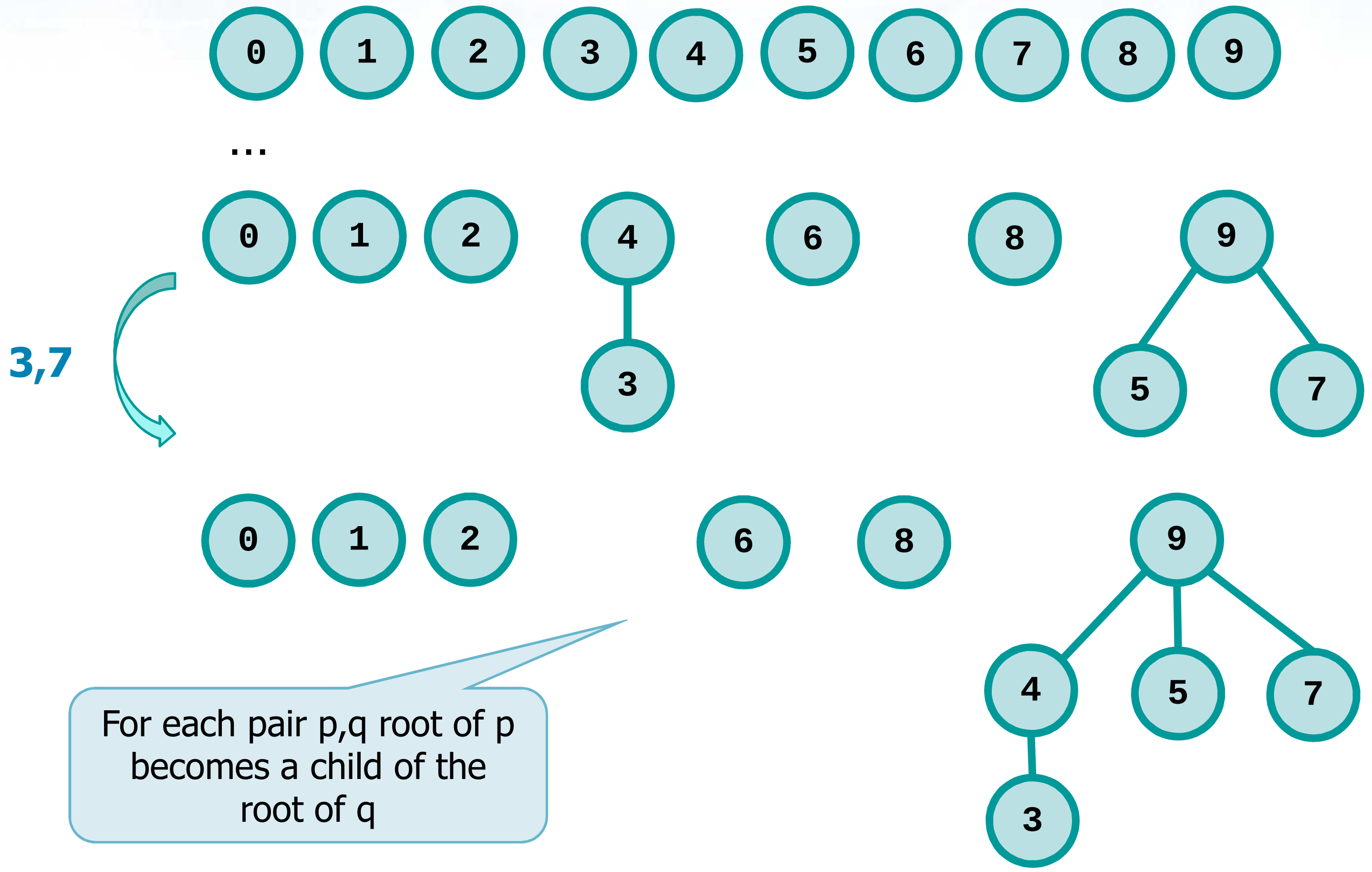
- Connections can be easily followed on the tree representation, moving from bottom to top

	0	1	2	3	4	5	6	7	8	9
id	0	1	9	4	9	6	6	7	0	9



Keep going until id[i]==i

Quick-union logic



For each pair p,q root of p becomes a child of the root of q

Implementation

- Repeat for all the pairs (p, q)
  - Read the pair (p, q)
  - Execute find on p to find the class leader of p
    - Find L_p = (id[p])*
  - Execute find on q to find the class leader of q
    - Find L_q = ((id[q])*
  - If L_p and L_q coincide
    - Do nothing and move on to the next pair
      - The pair is already connected
    - Otherwise, execute union on L_p and L_q
      - L_p=L_q, i.e., id[(id[p])*] = (id[q])*

Implementation

```
#include <stdio.h>

#define N 10000

int main() {
  int i, j, p, q, id[N];
  for (i=0; i<N; i++) {
    id[i] = i;
  }
  do {
    printf ("Input pair p q: ");
    scanf ("%d %d", &p, &q);
    for (i = p; i!= id[i]; i = id[i]);
    for (j = q; j!= id[j]; j = id[j]);
    if (i != j) {
      id[i] = j;
      printf ("%d %d\n", p, q);
    }
  } while (p!=q);
}
```

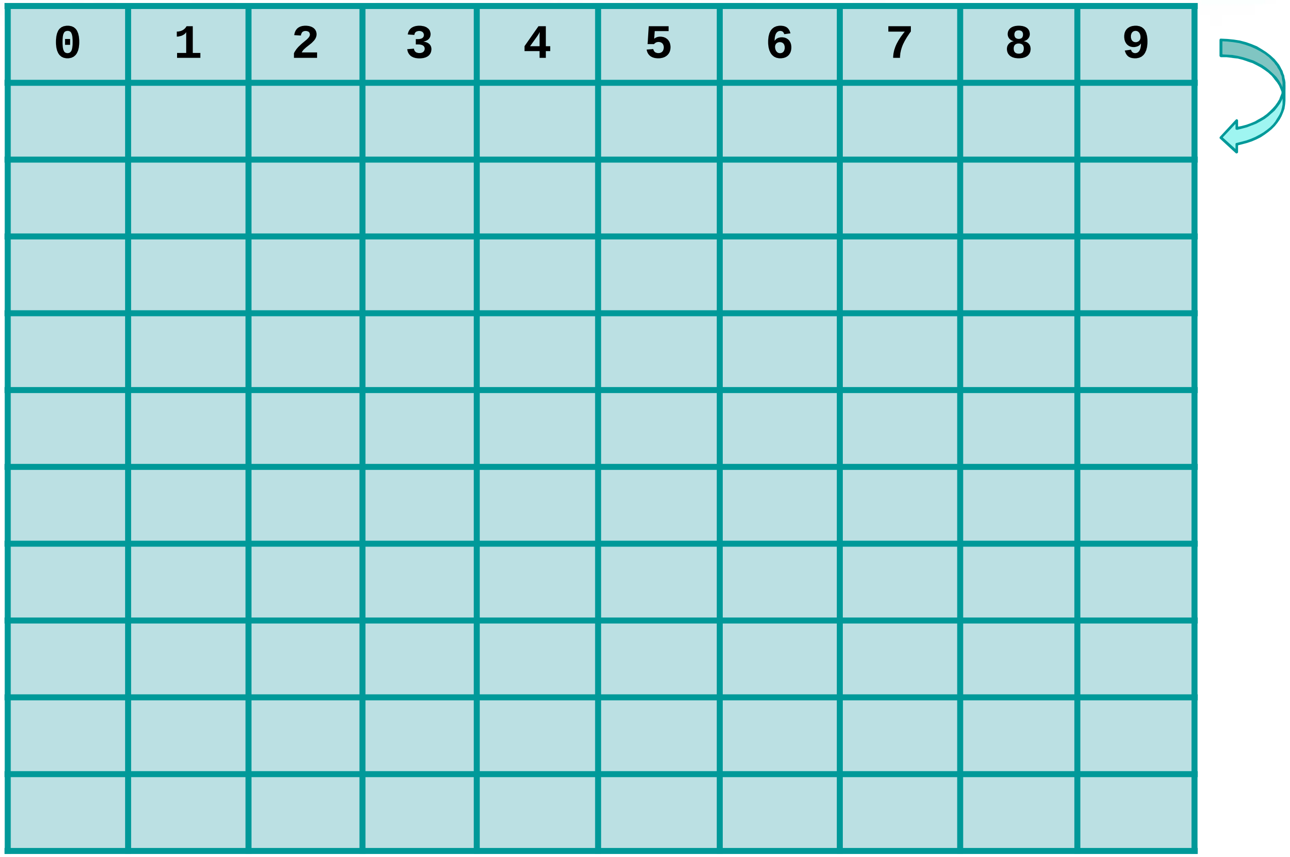
```
i = p;
while (i!= id[i]) {
  i = id[i];
}
```

Find p

Find q

Union p and q

Example



	0	1	2	3	4	5	6	7	8	9
	0	1	2	3	4	5	6	7	8	9
0-2										
2-4										
5-1										
4-8										
7-3										
5-9										
9-4										
5-6										
6-3										
3-5										

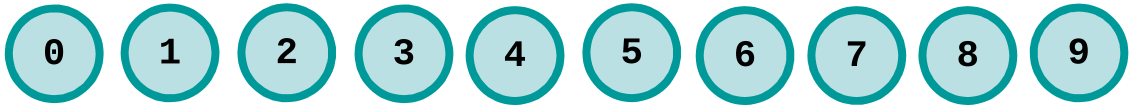
Solution

	0	1	2	3	4	5	6	7	8	9
	0	1	2	3	4	5	6	7	8	9
0-2	**2**	1	2	3	4	5	6	7	8	9
2-4	2	1	**4**	3	4	5	6	7	8	9
5-1	2	1	4	3	4	**1**	6	7	8	9
4-8	2	1	4	3	**8**	1	6	7	8	9
7-3	2	1	4	3	8	1	6	**3**	8	9
5-9	2	**9**	4	3	8	1	6	3	8	9
9-4	2	9	4	3	8	1	6	3	8	**8**
5-6	2	9	4	3	8	1	6	3	**6**	8
6-3	2	9	4	3	8	1	**3**	3	6	8
3-5	2	9	4	3	8	1	3	3	6	8

Exercise

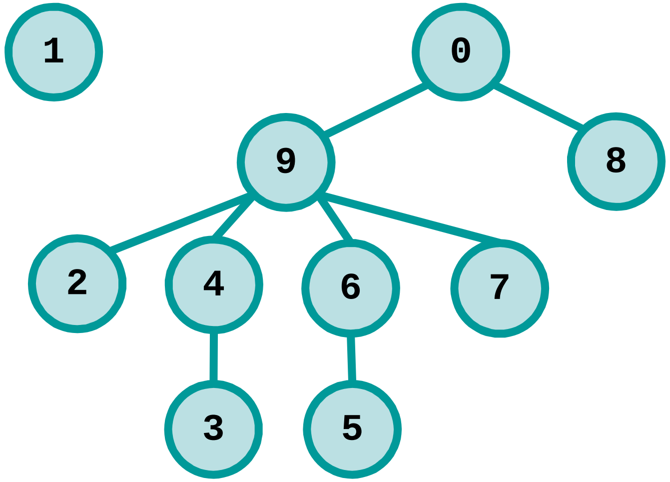
- Input Pairs: 3-4, 4-9, 8-0, 2-3, 5-6, 2-9, 5-9, 7-3, 4-8, 6-5, 0-2

	0	1	2	3	4	5	6	7	8	9
id	0	1	2	3	4	5	6	7	8	9



Solution

	0	1	2	3	4	5	6	7	8	9
id	1	1	9	4	9	6	9	9	0	0



Complexity

- Find
  - Scan a "chain" of objects
  - Upper bound
    - Linear cost in the number of objects
    - In general well below this value (depending on the chain length, tree height)

```
do {
  printf ("Input pair p q: ");
  scanf ("%d %d", &p, &q);
  for (i = p; i!= id[i]; i = id[i]);
  for (j = q; j!= id[j]; j = id[j]);
  if (i != j) {
    id[i] = j;
    printf ("%d %d\n", p, q);
  }
} while (p!=q);
```

Complexity

- Union
  - Simple, as it is enough that an object points to another object, unit cost
- Overall
  - Number of operations related to
    - # pairs · chain length = M · chain length
  - Still too slow for long chains

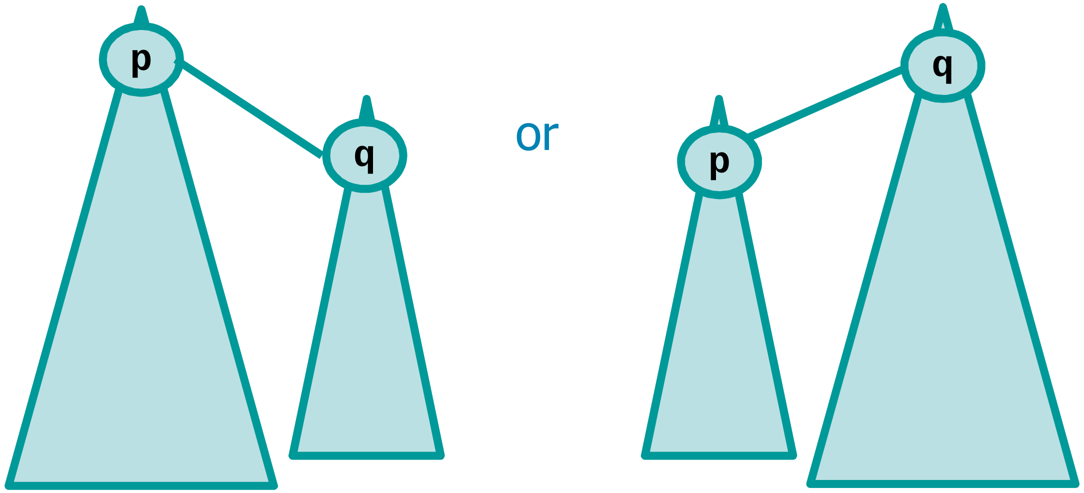
```
do {
  printf ("Input pair p q: ");
  scanf ("%d %d", &p, &q);
  for (i = p; i!= id[i]; i = id[i]);
  for (j = q; j!= id[j]; j = id[j]);
  if (i != j) {
    id[i] = j;
    printf ("%d %d\n", p, q);
  }
} while (p!=q);
```

Quick union optimizations

- Weighted quick union
  - To shorten the chain length
    - Keep track of the **number of elements** in each tree
    - Connect the **smaller** tree to the **larger** one
  - Use an array (array sz) to store tree size

Union by height or "rank",
i.e., always link the
root of **smaller** tree
to root of **larger** tree

- Given two trees
  - According to which one is the larger, there might be 2 solutions



  - It is irrelevant if p appears at the right or at the left of q

Implementation

```
int i, j, p, q, id[N], sz[N];
for(i=0; i<N; i++) {
  id[i] = i; sz[i] = 1;
}
do {
  printf ("Input pair p q: ");
  scanf ("%d %d", &p, &q);
  for (i = p; i!= id[i]; i = id[i]);
  for (j = q; j!= id[j]; j = id[j]);
  if (i == j)
    printf ("pair %d %d already connected\n", p,q);
  else {
    printf ("pair %d %d not yet connected\n", p, q);
    if (sz[i] <= sz[j]) {
      id[i] = j; sz[j] += sz[i];
    } else {
      id[j] = i; sz[i] += sz[j];
    }
  }
} while (p!=q);
```

Find p

Find q

Union: smaller tree below larger tree

Example

We need to represent trees to easily remind the tree size

	0	1	2	3	4	5	6	7	8	9
	0	1	2	3	4	5	6	7	8	9
0-2										
2-4										
5-1										
4-8										
7-3										
5-9										
9-4										
5-6										
6-3										
3-5										

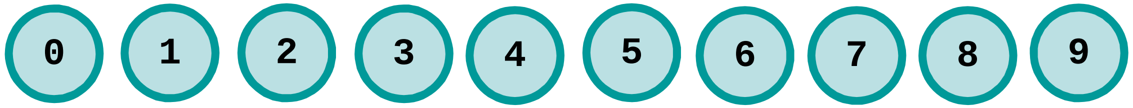
Solution

	0	1	2	3	4	5	6	7	8	9
	0	1	2	3	4	5	6	7	8	9
0-2	2	1	2	3	4	5	6	7	8	9
2-4	2	1	2	3	2	5	6	7	8	9
5-1	2	1	2	3	2	1	6	7	8	9
4-8	2	1	2	3	2	1	6	7	2	9
7-3	2	1	2	3	2	1	6	3	2	9
5-9	2	1	2	3	2	1	6	3	2	1
9-4	2	2	2	3	2	1	6	3	2	1
5-6	2	2	2	3	2	1	2	3	2	1
6-3	2	2	2	2	2	1	2	3	2	1
3-5	2	2	2	2	2	1	2	3	2	1

Exercise

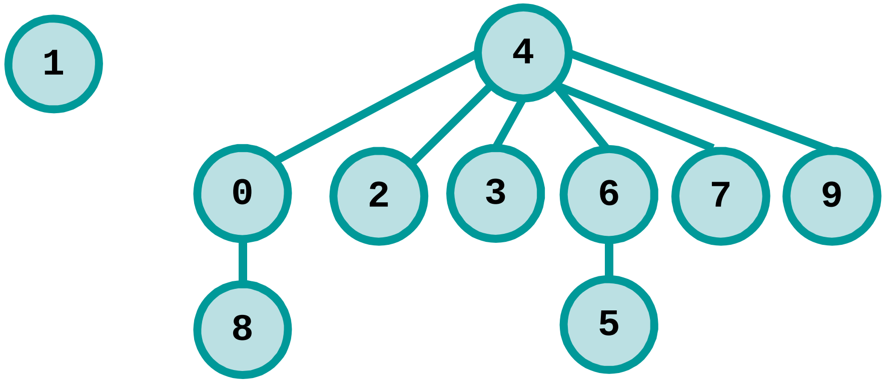
- Input Pairs: 3-4, 4-9, 8-0, 2-3, 5-6, 2-9, 5-9, 7-3, 4-8, 6-5, 0-2

	0	1	2	3	4	5	6	7	8	9
id	0	1	2	3	4	5	6	7	8	9



Solution

	0	1	2	3	4	5	6	7	8	9
id	4	1	4	4	4	6	4	4	0	4



Complexity

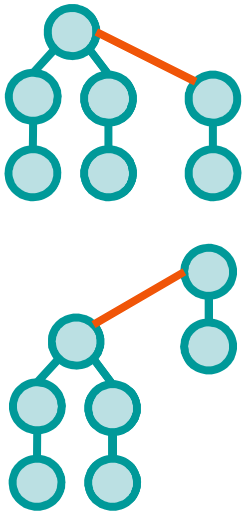
- Find
  - Linear cost in the chain length
- Union
  - Simple, as it is enough that an object points to another object, unit cost
- Overall
  - Number of operations related to
    - # pairs · chain length = M · chain length

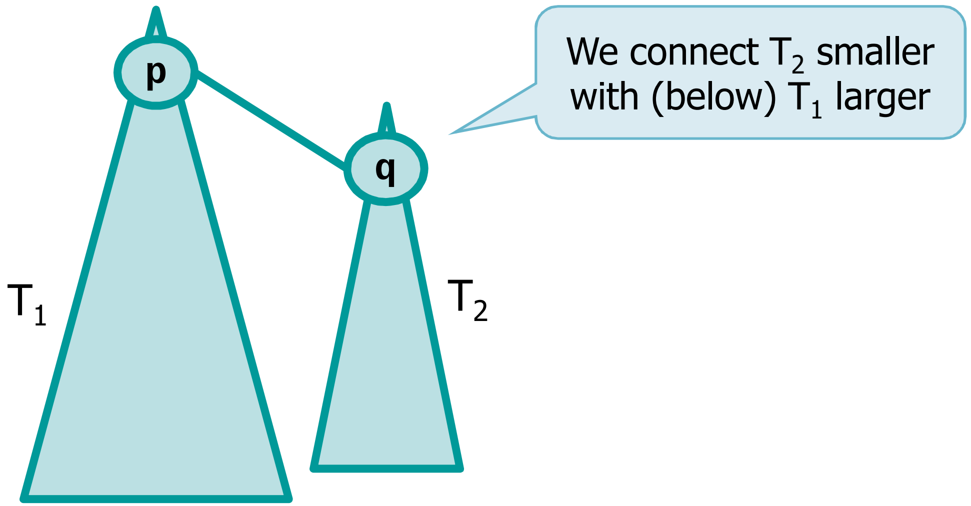
As quick union
… but …

  - **But** chain length **grows logarithmically** !

Complexity

- Why logarithmically?
  - What matters is the maximum distance between a node and the root
    - What matters is the height of the tree
    - What matters is the longest path between the root and a leaf

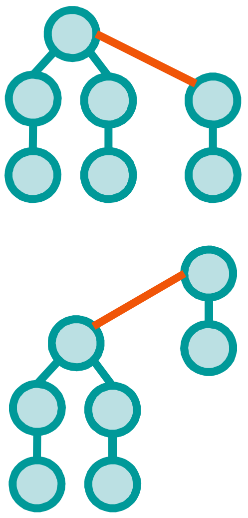


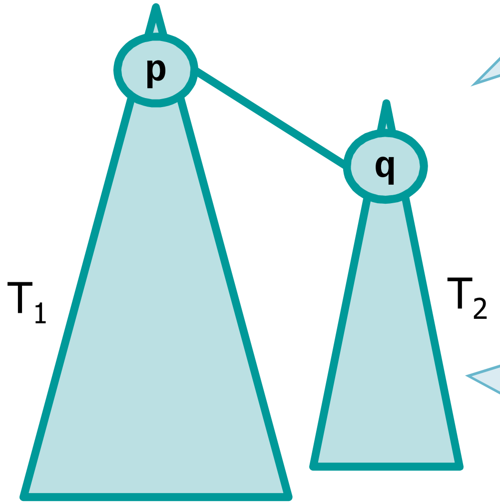


We connect T_2 smaller with (below) T_1 larger

Complexity

- If
  - Height of T_2 < height of T_1 (strictly less)
    - The overall height does not change
    - It remains equal to the height of T_1
  - Height of T_2 = height of T_1
    - The overall height is increased by 1



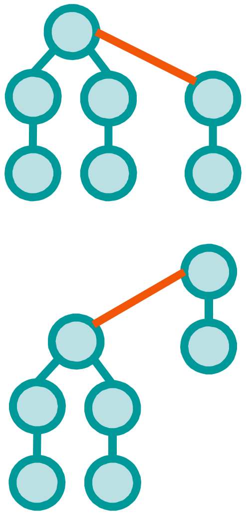


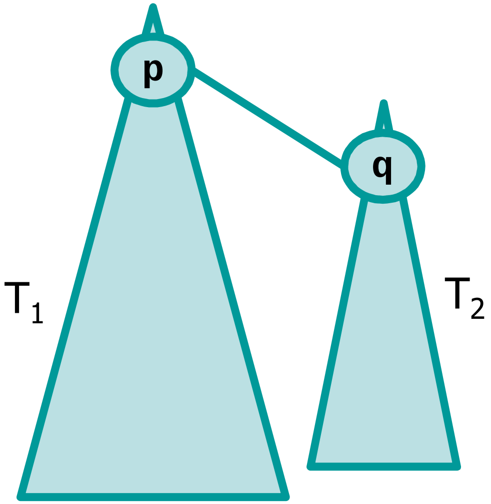
The worst case scenario is the one of union **linking trees of equal size**

The maximum height is the maximum between the one of T_1 and the one of T_2 plus 1

Complexity

- But if the height of T_1 is ≥ the height of T_2
  - Each time we connect a smaller tree to a larger one we generate a tree whose size is **on average at least** twice as big as T_2





Because T_1, being higher, should include, in fact on average, more nodes than T_2

Complexity

- Then
  - At each step the number of elements increases by at least a factor 2
  - After **i steps** there will be at least
    - $(((1 \cdot 2) \cdot 2) \cdot 2 \ldots = 2^i$

    elements
  - As 2^i cannot exceed N
    - $2^i \leq N$

    must hold
  - Thus
    - $\mathbf{i} \leq \log_2 N$

Where i is the number of steps