Union Find

- Union Find
- Quick Find
- Quick Union
- Weighted Quick Union
- Path Compression
- Dynamic Connectivity

Philip Bille

Union Find

- Union Find
- Quick Find
- Quick Union
- Weighted Quick Union
- Path Compression
- Dynamic Connectivity

Philip Bille

Union Find

- Union find. Maintain a dynamic family of sets supporting the following operations:
  - INIT(n): construct sets {0}, {1},…, {n-1}
  - UNION(i,j): forms the union of the two sets that contain i and j. If i and j are in the same set nothing happens.
  - FIND(i): return a representative for the set that contains i.

INIT(9)

{0} {1} {2} {3} {4} {5} {6} {7} {8}

{1, 0, 6} {8, 3, 2, 7} {4, 5} —UNION(5,0)→ {1, 0, 6, 4, 5} {8, 3, 2, 7}

Union Find

- Applications.
  - Dynamic connectivity.
  - Minimum spanning tree.
  - Unification in logic and compilers.
  - Nearest common ancestors in trees.
  - Hoshen-Kopelman algorithm in physics.
  - Games (Hex and Go)
  - Illustration of clever techniques in data structure design.

Union Find

- Union Find
- **Quick Find**
- Quick Union
- Weighted Quick Union
- Path Compression
- Dynamic Connectivity

Philip Bille

Quick Find

- Quick find. Maintain array id[0..n-1] such that id[i] = representative for i.
  - INIT(n): set elements to be their own representative.
  - UNION(i,j): if FIND(i) ≠ FIND(j), update representative for all elements in one of the sets.
  - FIND(i): return representative.

INIT(9)

{0} {1} {2} {3} {4} {5} {6} {7} {8}

	0	1	2	3	4	5	6	7	8
id[]	0	1	2	3	4	5	6	7	8

{1, 0, 6} {8, 3, 2, 7} {4, 5} → UNION(5,0) → {1, 0, 6, 4, 5} {8, 3, 2, 7}

	0	1	2	3	4	5	6	7	8
id[]	1	1	3	3	5	5	1	3	3

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

Quick Find

```
INIT(n):
   for k = 0 to n-1
      id[k] = k
```

```
FIND(i):
   return id[i]
```

```
UNION(i,j):
   iID = FIND(i)
   jID = FIND(j)
   if (iID ≠ jID)
      for k = 0 to n-1
         if (id[k] == iID)
            id[k] = jID
```

{1, 0, 6} {8, 3, 2, 7} {4, 5} → UNION(5,0) → {1, 0, 6, 4, 5} {8, 3, 2, 7}

	0	1	2	3	4	5	6	7	8
id[]	1	1	3	3	5	5	1	3	3

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

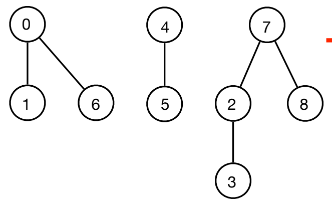
- Time.
  - O(n) time for INIT, O(n) time for UNION, and O(1) tid for FIND.

Union Find

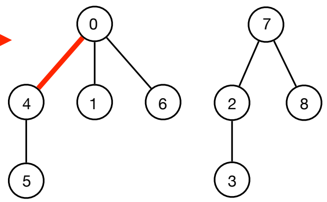
- Union Find
- Quick Find
- **Quick Union**
- Weighted Quick Union
- Path Compression
- Dynamic Connectivity

Quick Union

- Quick union. Maintain each sets as a rooted tree.
- Store trees as array p[0..n-1] such that p[i] is the parent of i and p[root] = root. Representative is the root of tree.
  - INIT(n): create n trees with one element each.
  - UNION(i,j): if FIND(i) ≠ FIND(j), make the root of one tree the child of the root of the other tree.
  - FIND(i): follow path to root and return root.



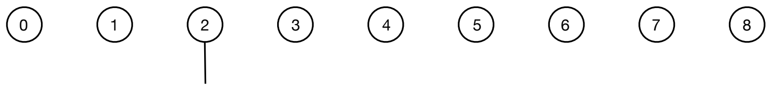
	0	1	2	3	4	5	6	7	8
p[]	0	0	7	2	4	4	0	7	7



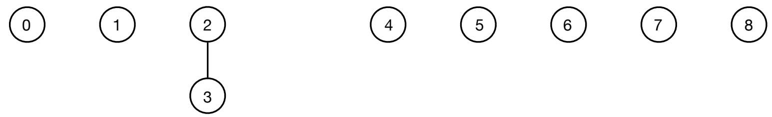
0	1	2	3	4	5	6	7	8
0	0	7	2	0	4	0	7	7

INIT(9)

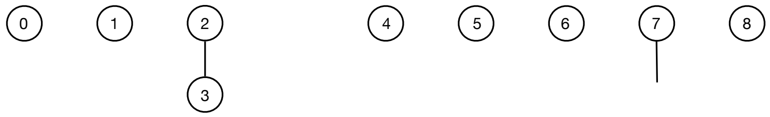
UNION(3,2)



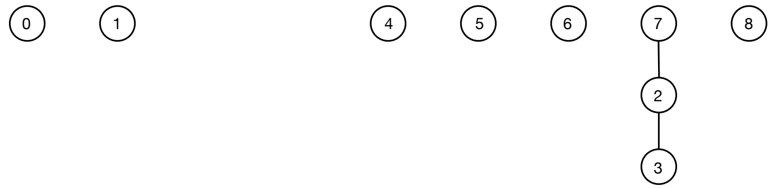
UNION(3,2)



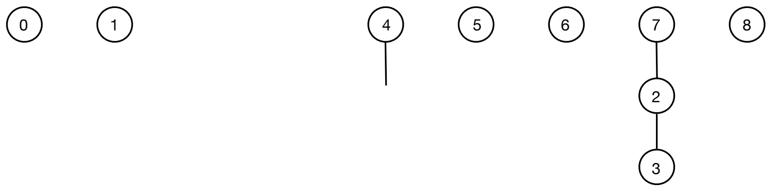
UNION(2,7)



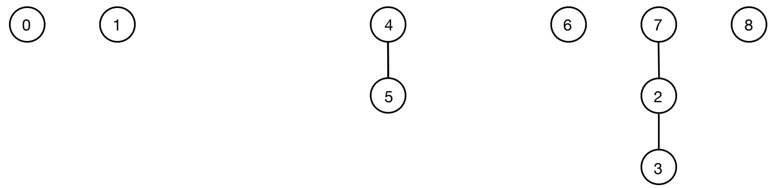
UNION(2,7)



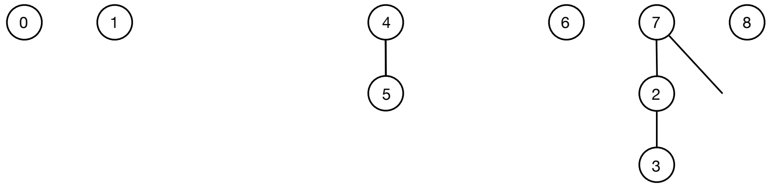
UNION(5,4)



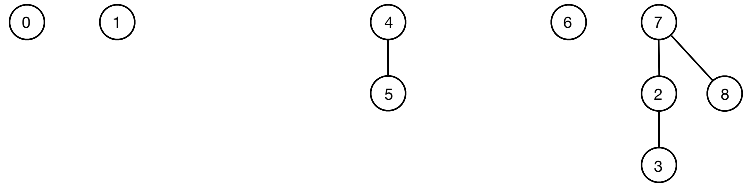
UNION(5,4)



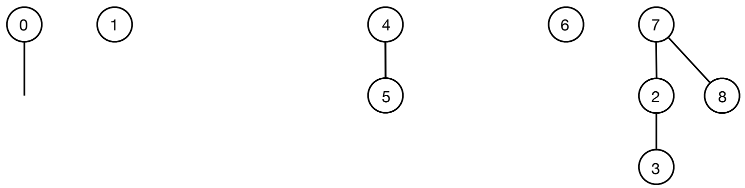
Union(8,3)



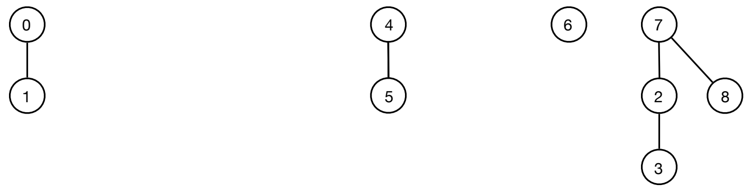
Union(8,3)



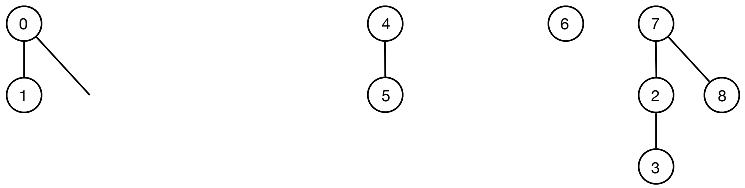
Union(1,0)



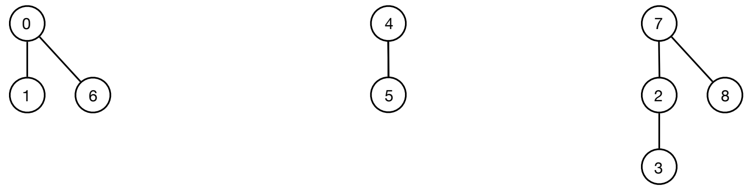
Union(1,0)



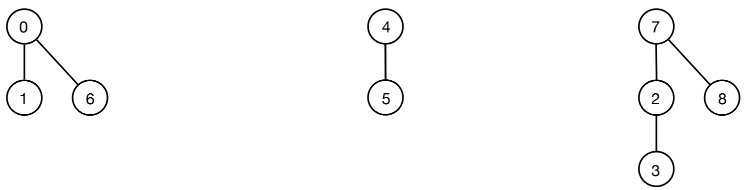
UNION(6,1)



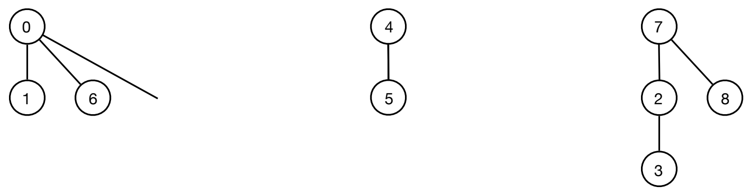
UNION(6,1)



UNION(7,3)



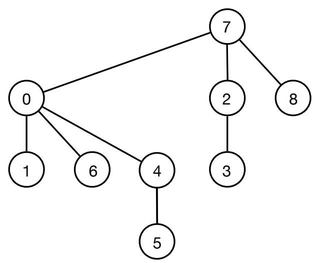
UNION(5,0)



Union(5,0)

Union(6,2)

Union(6,2)



Quick Union

- Init(n): create n trees with one element each.
- Union(i,j): if Find(i) ≠ Find(j), make the root of one tree the child of the root of the other tree.
- Find(i): follow path to root and return root.

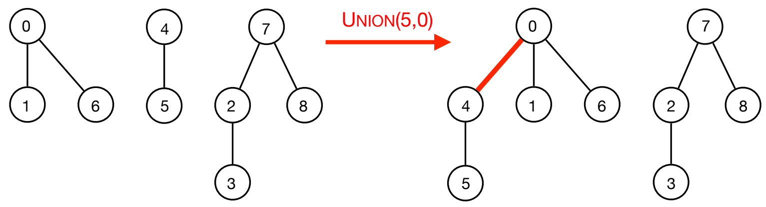
- Exercise. Show data structure after each operation in the following sequence.
  - Init(7), Union(0,1), Union(2,3), Union(5,1), Union(5,0), Union(0,3), Union(5,2), Union(4,3), Union(4,6).

Quick Union

```
INIT(n):
   for k = 0 to n-1
      p[k] = k

FIND(i):
   while (i != p[i])
      i = p[i]
   return i
```

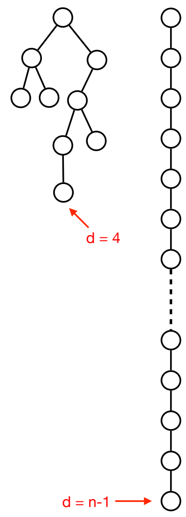
```
UNION(i,j):
   ri = FIND(i)
   rj = FIND(j)
   if (ri ≠ rj)
      p[ri] = rj
```



- Time.
  - O(n) time for INIT, O(d) tid for UNION and FIND, where d is the depth of the tree.

Quick Union

- UNION and FIND depend on the depth of the tree.
- Bad news. Depth can be n-1.
- Challenge. Can combine trees to limit the depth?

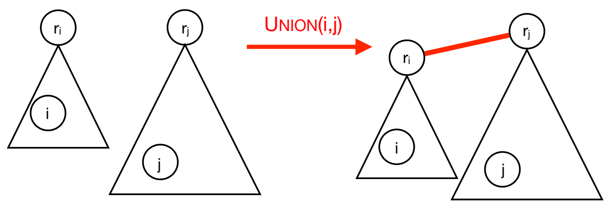


Union Find

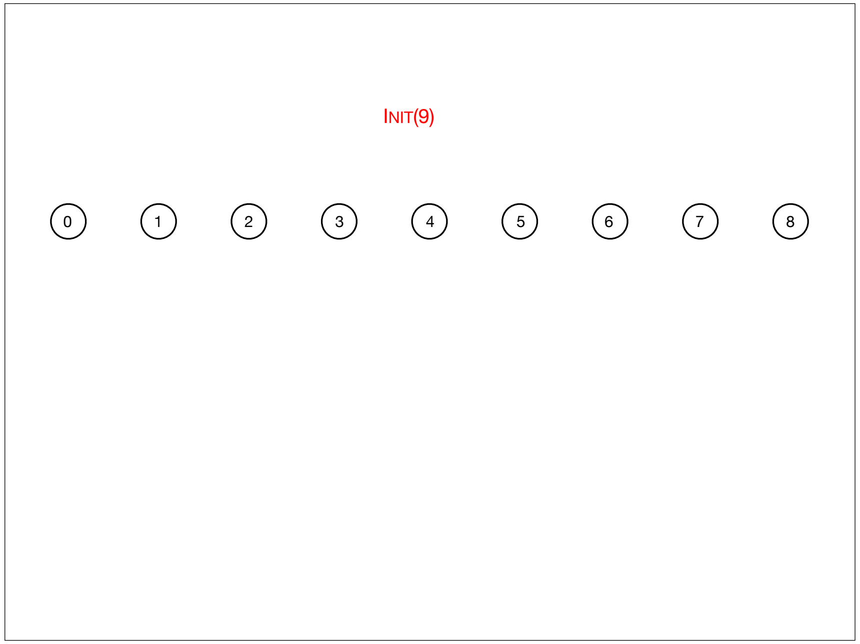
- Union Find
- Quick Find
- Quick Union
- **Weighted Quick Union**
- Path Compression
- Dynamic Connectivity

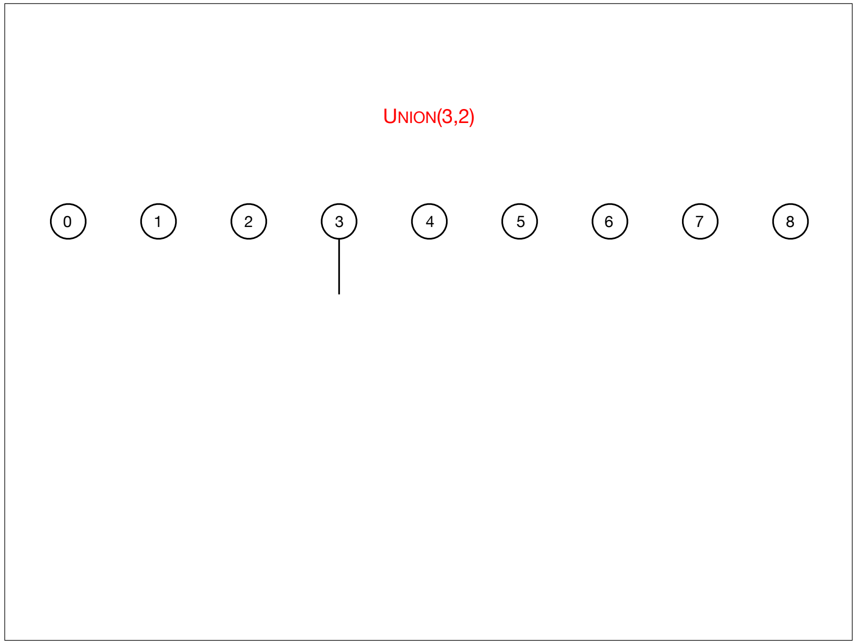
Weighted Quick Union

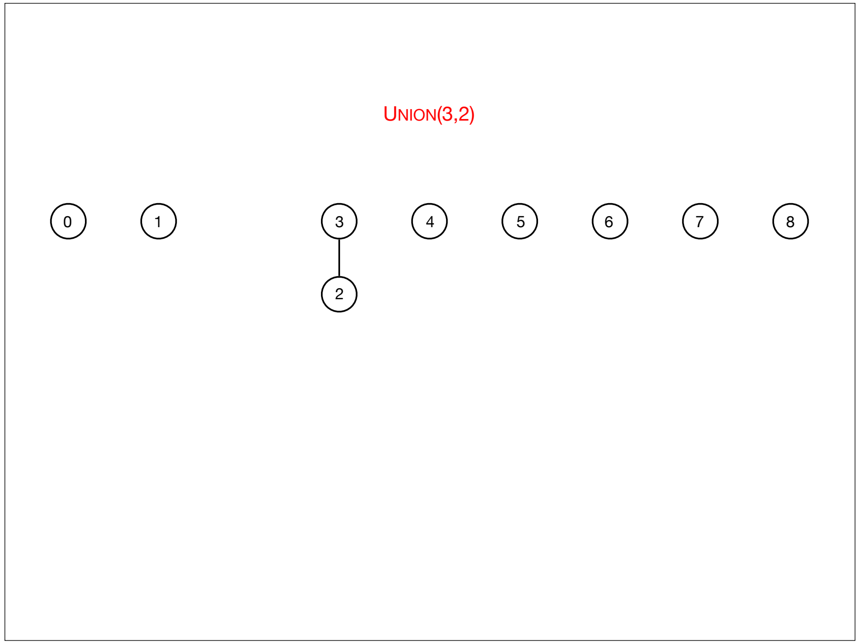
- Weighted quick union. Extension of quick union.
- Maintain extra array sz[0..n-1] such sz[i] = the size of the subtree rooted at i.
  - INIT: as before + initialize sz[0..n-1].
  - FIND: as before.
  - UNION(i,j): if FIND(i) ≠ FIND(j), make the root of the smaller tree the child of the root of the larger tree.
- Intuition. UNION balances the trees.

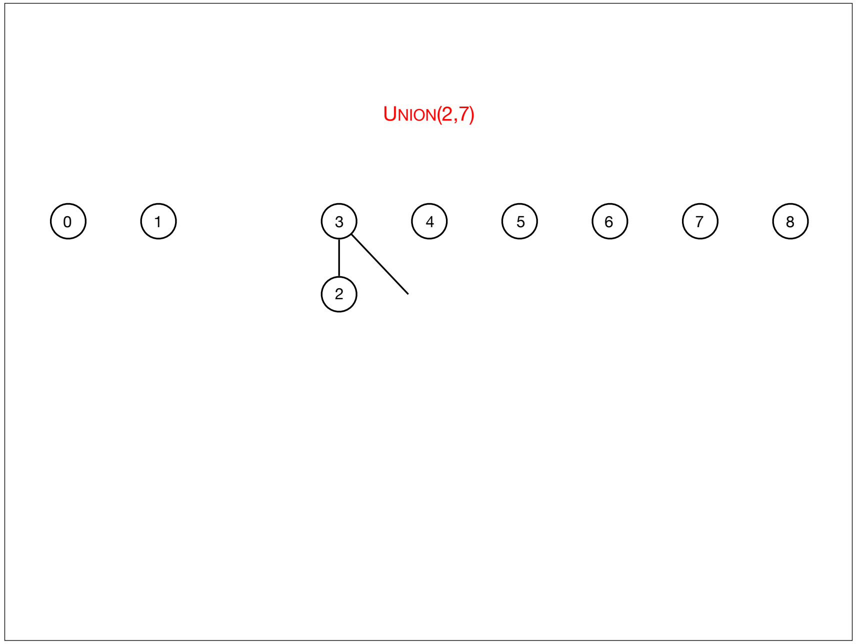


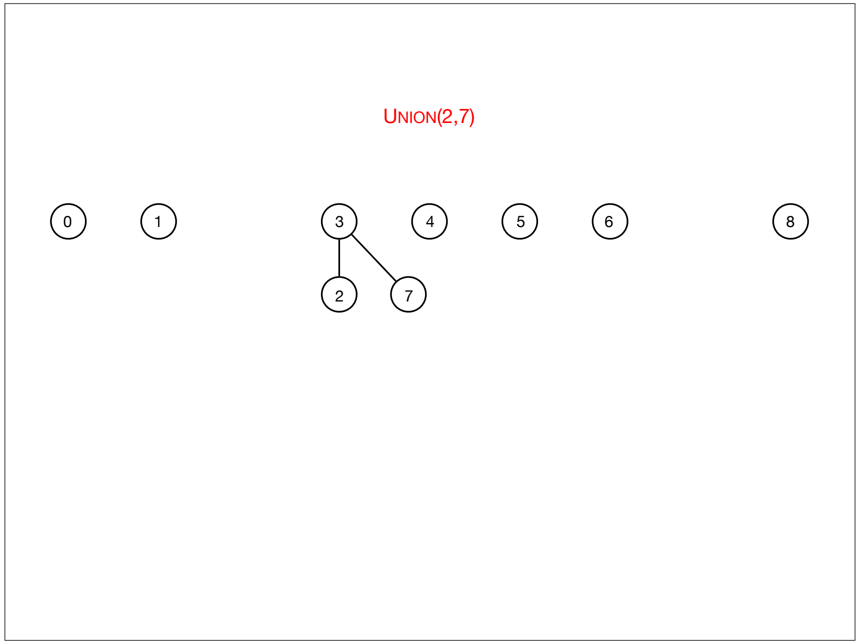
$sz[r_j] = sz[r_i] + sz[r_j]$

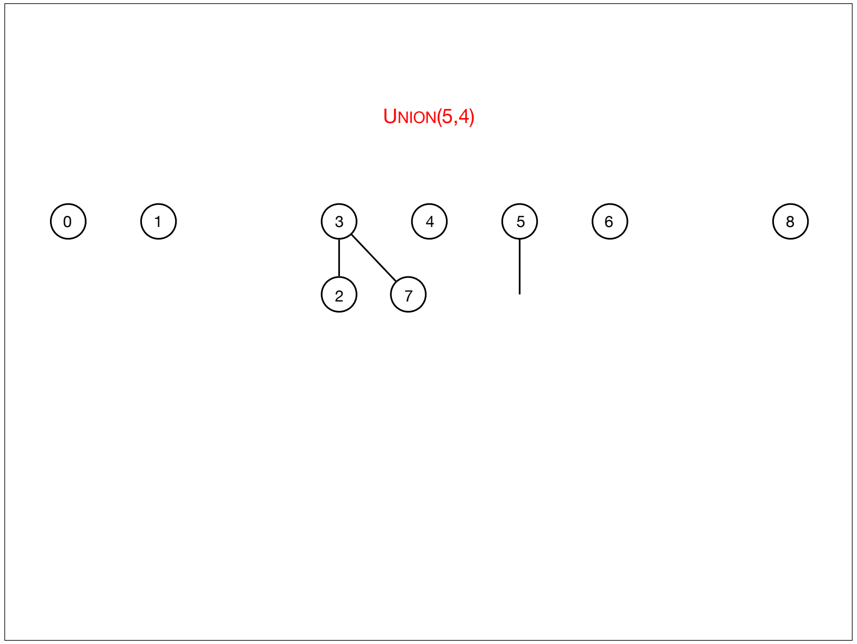

Init(9)
0 1 2 3 4 5 6 7 8

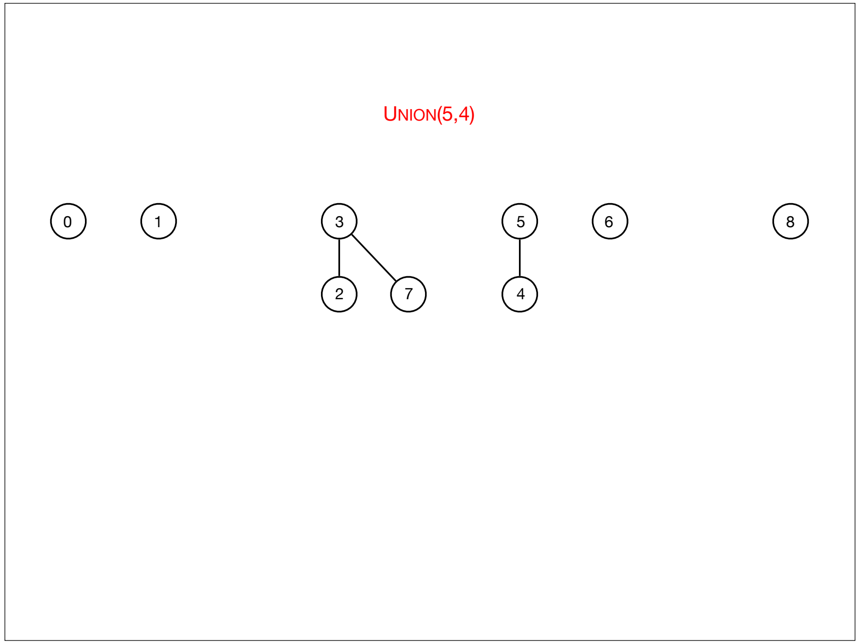

Union(3,2)
0 1 2 3 4 5 6 7 8

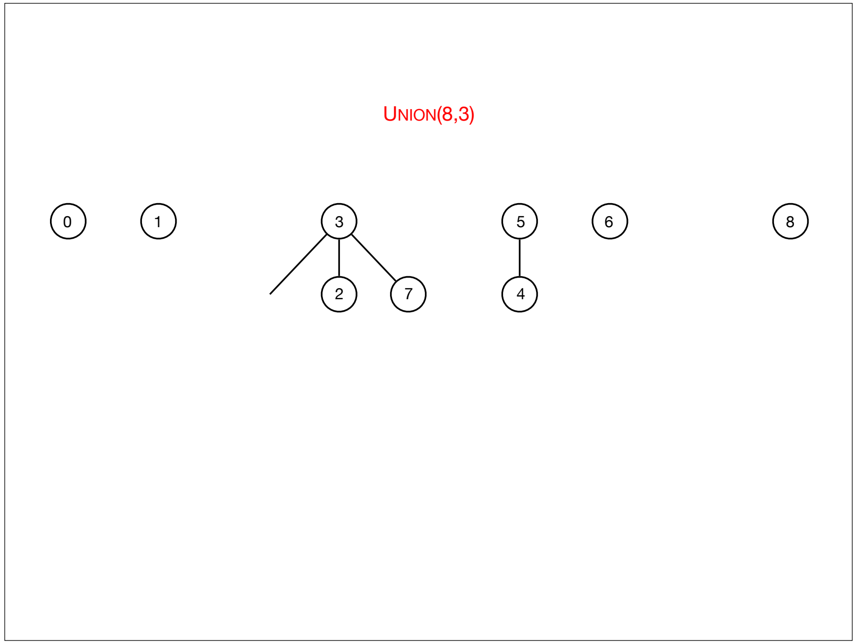

Union(3,2)
0 1 3 4 5 6 7 8
2


Union(2,7)
0 1 3 4 5 6 7 8
2

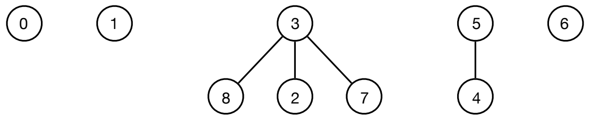

UNION(2,7)
0
1
3
4
5
6
8
2
7


UNION(5,4)
0
1
3
4
5
6
8
2
7

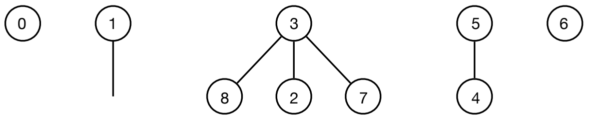

UNION(5,4)
0
1
3
5
6
8
2
7
4


UNION(8,3)
0
1
3
5
6
8
2
7
4

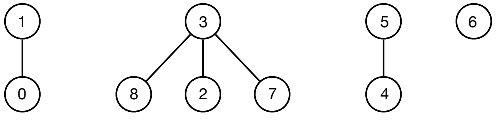
Union(8,3)



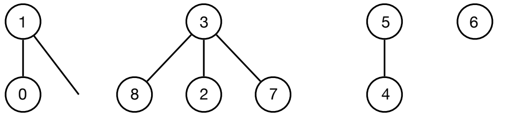
Union(1,0)



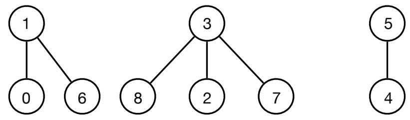
Union(1,0)



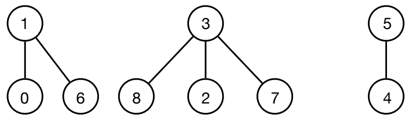
Union(6,1)



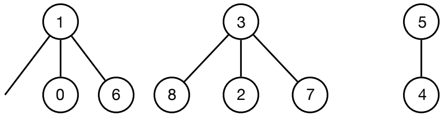
UNION(6,1)



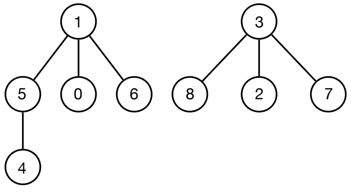
UNION(7,3)



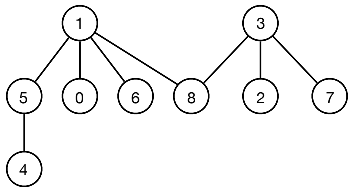
UNION(5,1)



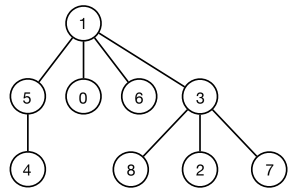
UNION(5,1)



UNION(6,3)

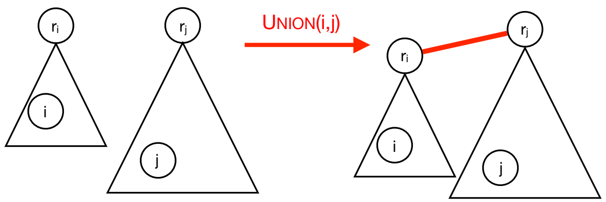


UNION(6,3)



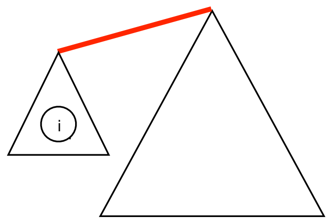
Weighted Quick Union

```
UNION(i,j):
   ri = FIND(i)
   rj = FIND(j)
   if (ri ≠ rj)
      if (sz[ri] < sz[rj])
         p[ri] = rj
         sz[rj] = sz[ri] + sz[rj]
      else
         p[rj] = ri
         sz[ri] = sz[ri] + sz[rj]
```



Weighted Quick Union

- Lemma. With weighted quick union the depth of a node is at most $\log_2 n$.
- Proof.
  - Consider node i with depth d_i.
  - Initially $d_i = 0$.
  - d_i increases with 1 when the tree is combined with a larger tree.
  - The combined tree is at least twice the size.
  - We can double the size of trees at most $\log_2 n$ times.
  - $\Longrightarrow d_i \leq \log_2 n$.



Union Find

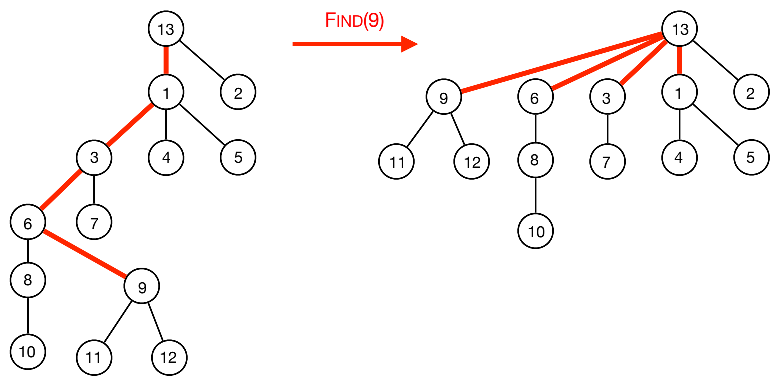
Data structure	Union	Find
quick find	O(n)	O(1)
quick union	O(n)	O(n)
weighted quick union	O(log n)	O(log n)

- Challenge. Can we do even better?

Union Find

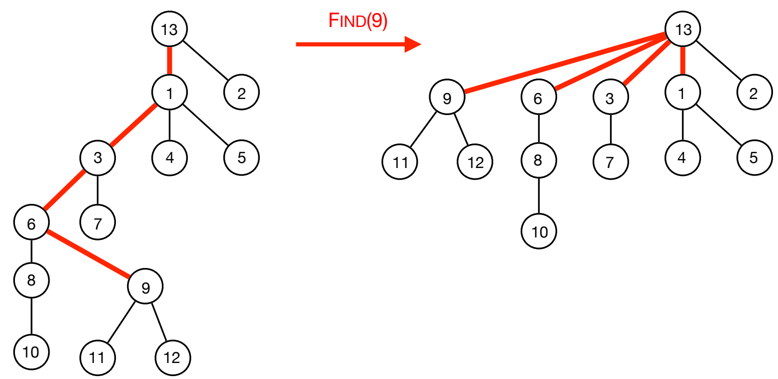
Path Compression

- Path compression. Compress path on Find. Make all nodes on the path children of the root.
- No change in running time for a single Find. Subsequent Find become faster.
- Works with both quick union and weighted quick union.



Path Compression

- Theorem [Tarjan 1975]. With path compression any sequence of m Find and Union operations on n elements take $O(n + m\,\alpha(m,n))$ time.
- $\alpha(m,n)$ is the inverse of Ackermanns function. $\alpha(m,n) \leq 5$ for any practical input.
- Theorem [Fredman-Saks 1985]. It is not possible to support m Find and Union operations $O(n + m)$ time.

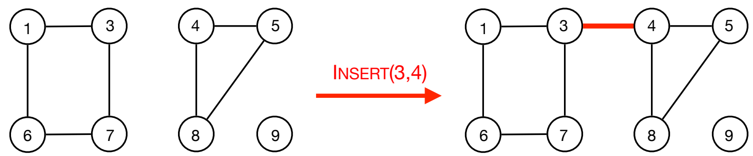


Union Find

- Union Find
- Quick Find
- Quick Union
- Weighted Quick Union
- Path Compression
- Dynamic Connectivity

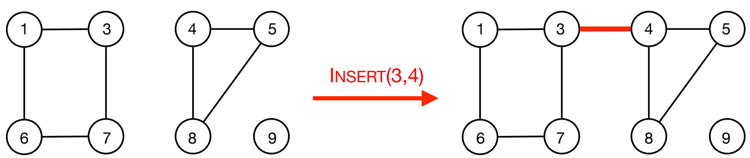
Dynamic Connectivity

- Dynamic connectivity. Maintain a dynamic graph supporting the following operations:
  - INIT(n): create a graph G with n vertices and no edges.
  - CONNECTED(u,v): determine if u og v are connected.
  - INSERT(u, v): add edge (u,v). We assume (u,v) does not already exists.



Dynamic Connectivity

- Implementation with union find.
  - INIT(n): initialize a union find data structure with n elements.
  - CONNECTED(u,v): FIND(u) == FIND(v).
  - INSERT(u, v): UNION(u,v)



- Time
  - O(n) time for INIT, O(log n) time for CONNECTED, and O(log n) time for INSERT

Union Find

- Union Find
- Quick Find
- Quick Union
- Weighted Quick Union
- Path Compression
- Dynamic Connectivity