Disjoint Set Class

Fall 2020
School of Software Engineering
South China University of Technology

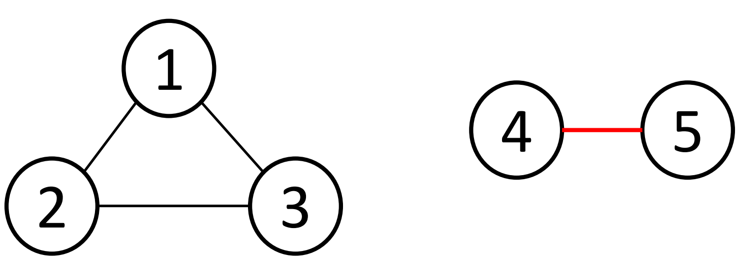
Equivalence Relations

- A relation R is defined on set S if for every pair of elements a, b ϵ S, a R b is either true or false.
- An equivalence relation is a relation R that satisfies the 3 properties:
  - Reflexive: a R a for all a ϵ S
  - Symmetric: a R b iff b R a; a, b ϵ S
  - Transitive: a R b and b R c implies a R c
- some examples
  - Relation “≤”, “≥”---not equivalence relation
  - Relation “be in the same class”, “Electrical connectivity” --- equivalence relation

Equivalence Classes

- Given an equivalence relation R, decide whether a pair of any elements a, b ϵ S is such that a R b.

- The equivalence class of an element a ϵ S is the subset of S of all elements related to a.
- Different equivalence classes of S are disjoint.
  - Every member of S appears in exactly one equivalence class



Dynamic Equivalence Problem

- Given an equivalence relation R, decide whether a pair of elements a, b $\in$ S is such that a R b.

- Check whether a and b are in the same equivalence class

- Equivalence Problem
  - the problem of assigning the members of a set to equivalence classes.

Dynamic Equivalence Problem

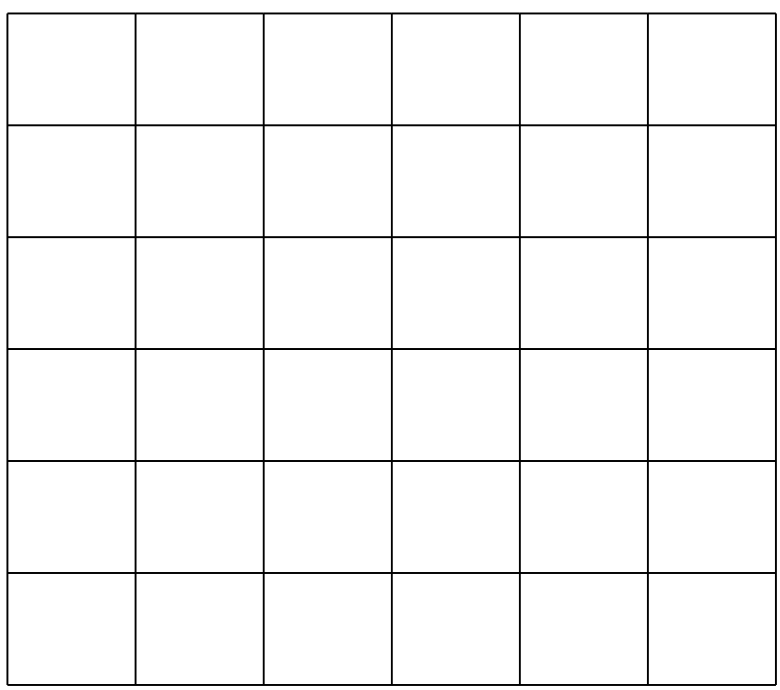
- Strategy:
  - Starting with each element in a singleton set. These singleton sets are **disjoint.**
  - two operations:
    - Find the equivalence class (set) of a given element
    - Union of two sets
- It is a dynamic (on-line) problem because the sets change during the operations and Find must be able to cope!

Disjoint Union/Find

- A set of pairwise disjoint sets.
  - Each set has a unique name, one of its members
  - {3,5,7} , {4,2,8}, {9}, {1,6}
- Find(x) – return the name of the set containing x.
  - Find(6) = 1
  - Find(4) = 8
  - Find(9) = 9
- Union(x,y) – take the union of two sets named x and y
  - Union(5,1) = {3,5,7,1,6}, {4,2,8}, {9},

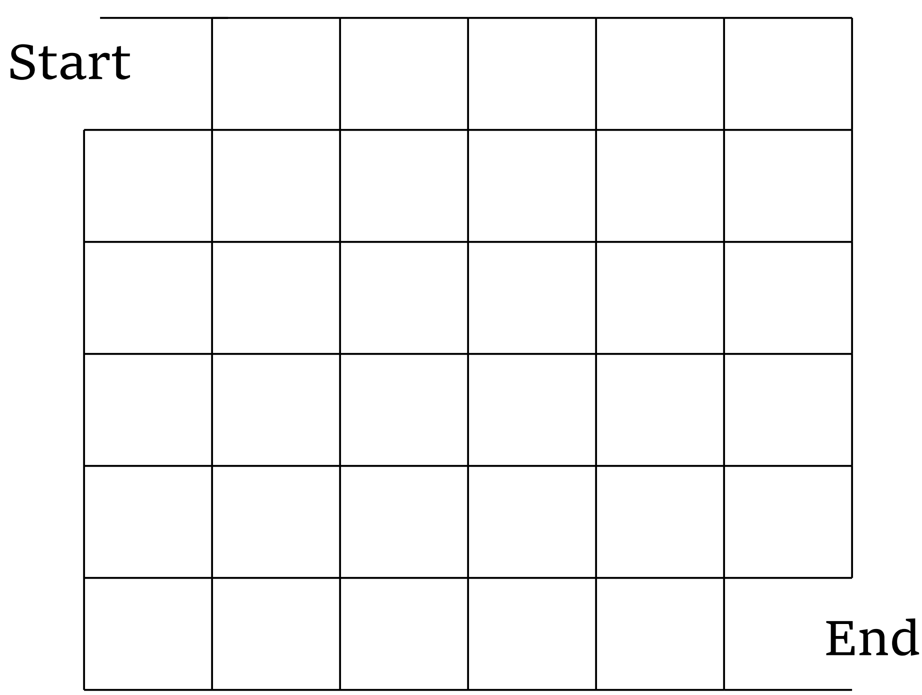
An Application

- Build a random maze by erasing edges.



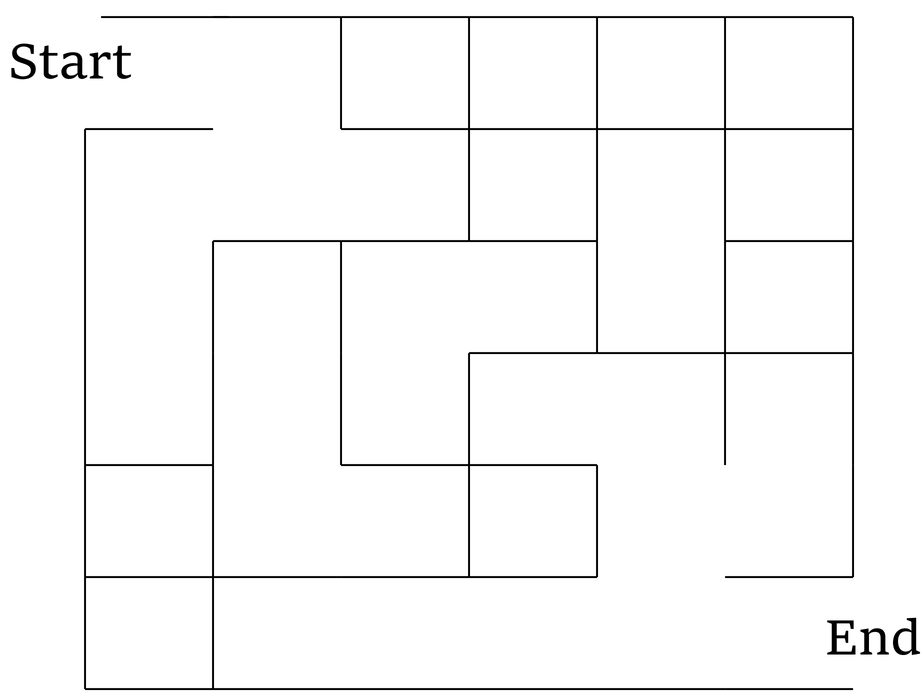
An Application (ct'd)

- Pick Start and End



An Application (ct'd)

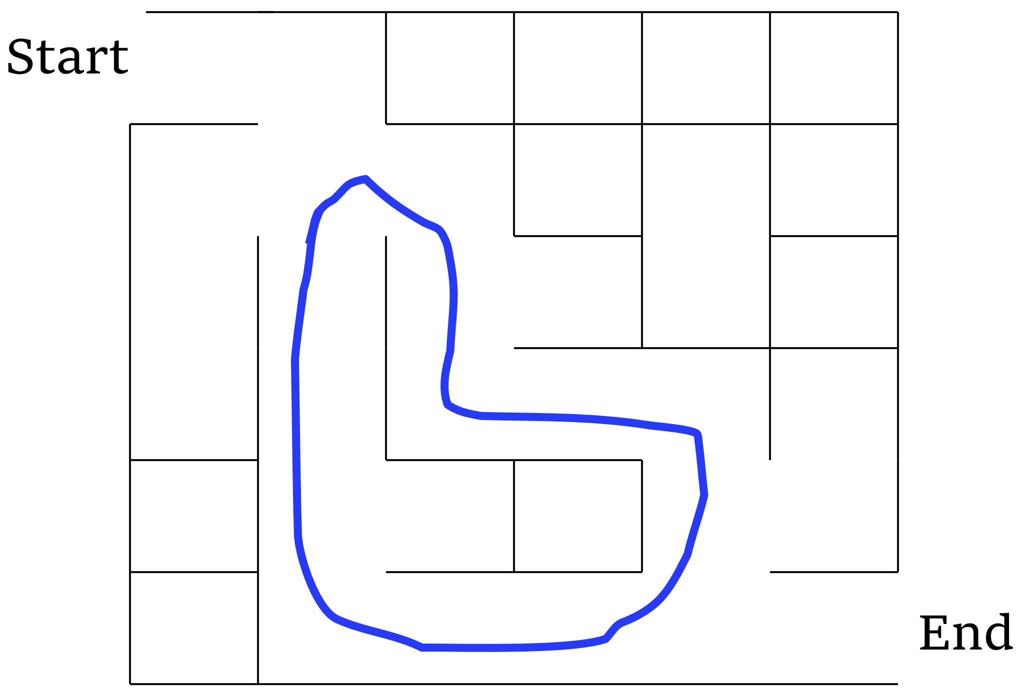
- Repeatedly pick random edges to delete.



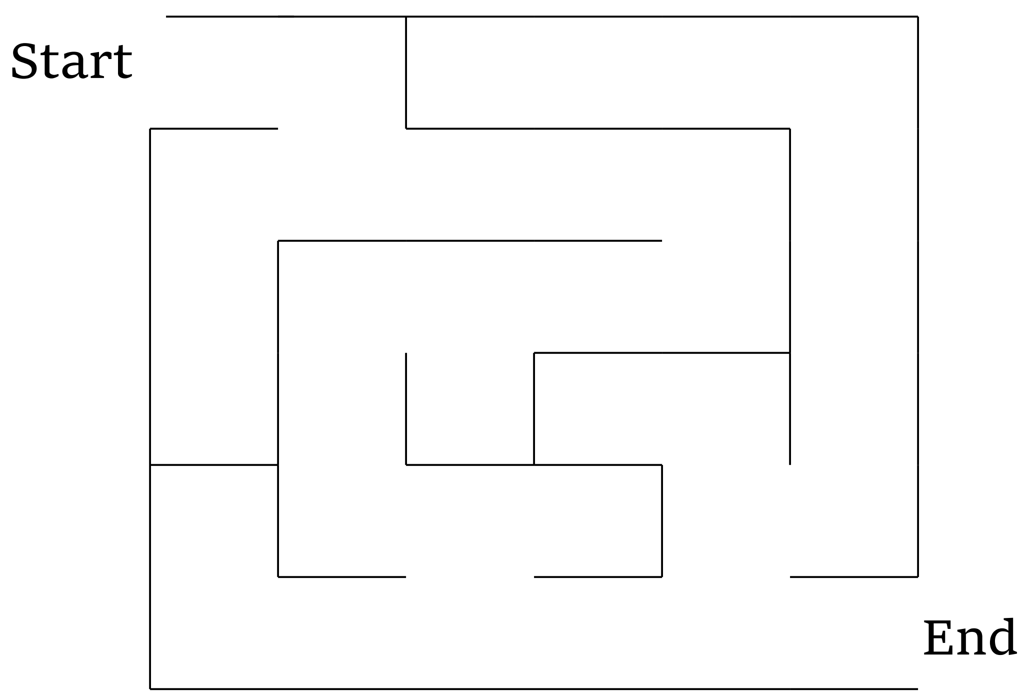
Desired Properties

- None of the boundary is deleted
- Every cell is reachable from every other cell.
- There are no cycles – no cell can reach itself by a path unless it retraces some part of the path.

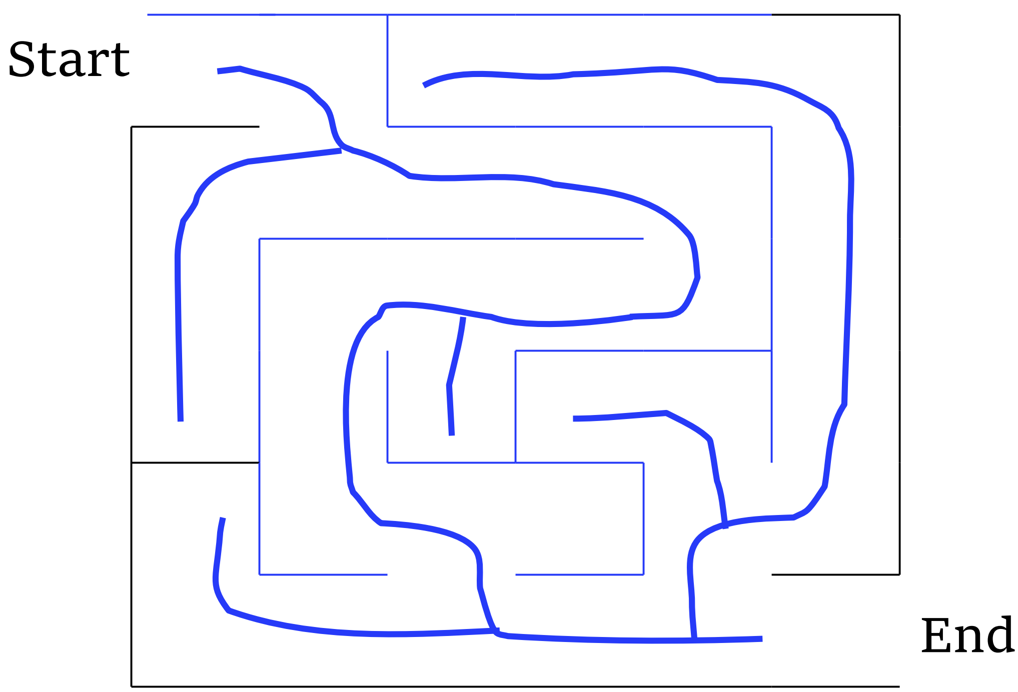
A Cycle (we don't want that)



A Good Solution



Good Solution : A Hidden Tree



Number the Cells

We have disjoint sets S ={ {1}, {2}, {3}, {4},… {36} } each cell is unto itself.
We have all possible edges E ={ (1,2), (1,7), (2,8), (2,3), … } 60 edges total.

Start	1	2	3	4	5	6	
	7	8	9	10	11	12	
	13	14	15	16	17	18	
	19	20	21	22	23	24	
	25	26	27	28	29	30	
	31	32	33	34	35	36	End

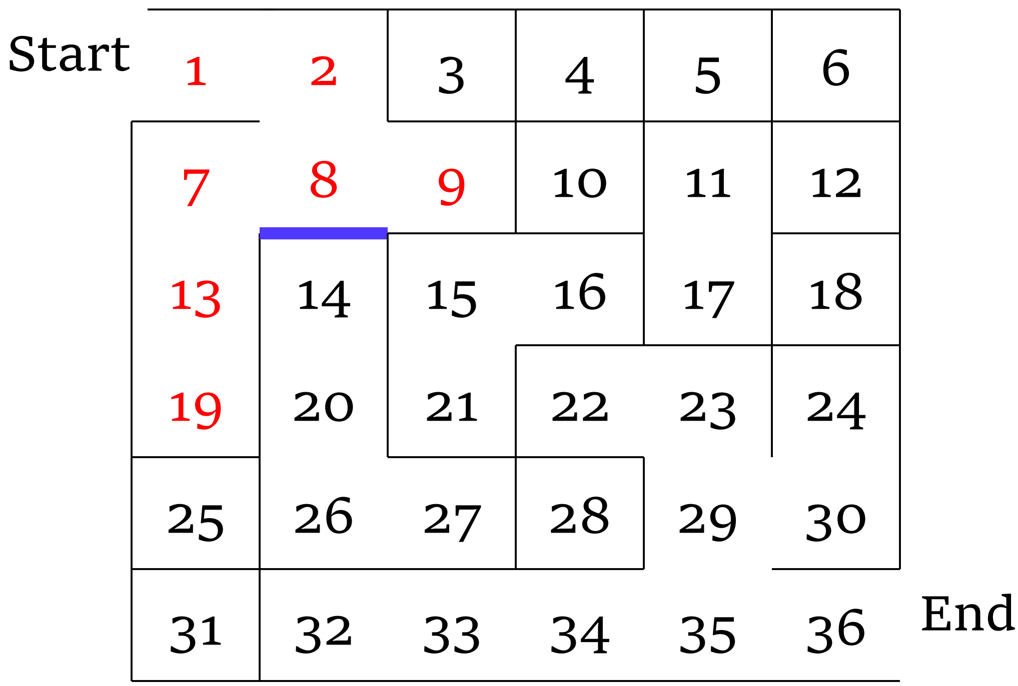
Basic Algorithm

- S = set of sets of connected cells
- E = set of edges
- Maze = set of maze edges initially empty

```
While there is more than one set in S
  pick a random edge (x,y) and remove from E
  u := Find(x);  v := Find(y);
  if u ≠ v then
    Union(u,v)  //knock down the wall between the
                // cells (cells in  the same set are
                // connected)
  else
     add (x,y) to Maze //don't remove because there is
                       // already a path between x and y
All remaining members of E together with Maze form
the maze
```

Example Step

Pick (8,14)



S

{1,2,7,8,9,13,19}, {3}, {4}, {5}, {6}, {10}, {11,17}, {12}, {14,20,26,27},{15,16,21}

.

.

{22,23,24,29,30,32,33,34,35,36}

Example

Pick (8,14)

S
{1,2,7,8,9,13,19}
{3}
{4}
{5}
{6}
{10}
{11,17}
{12}
{14,20,26,27}
{15,16,21}
.
.
{22,23,24,29,39
,32, 33,34,35,36}

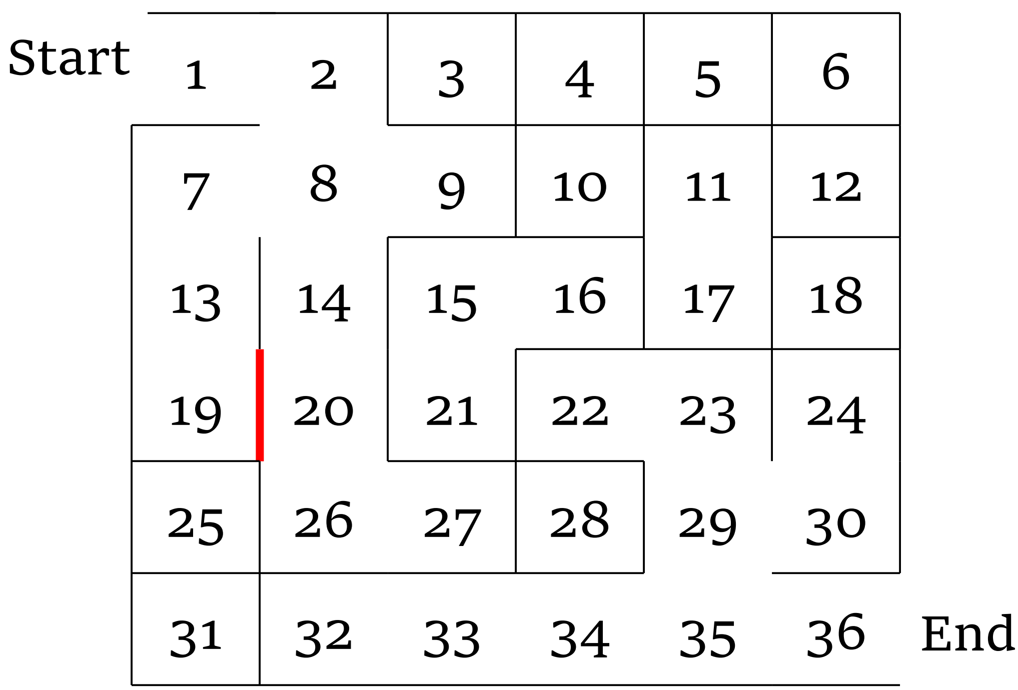
Find(8) = 7
Find(14) = 20

Union(7,20)

S
{1,2,7,8,9,13,19,
14,20 26,27}
{3}
{4}
{5}
{6}
{10}
{11,17}
{12}
{15,16,21}
.
.
{22,23,24,29,39
,32,33,34,35,36}

Example

Pick (19,20)



S

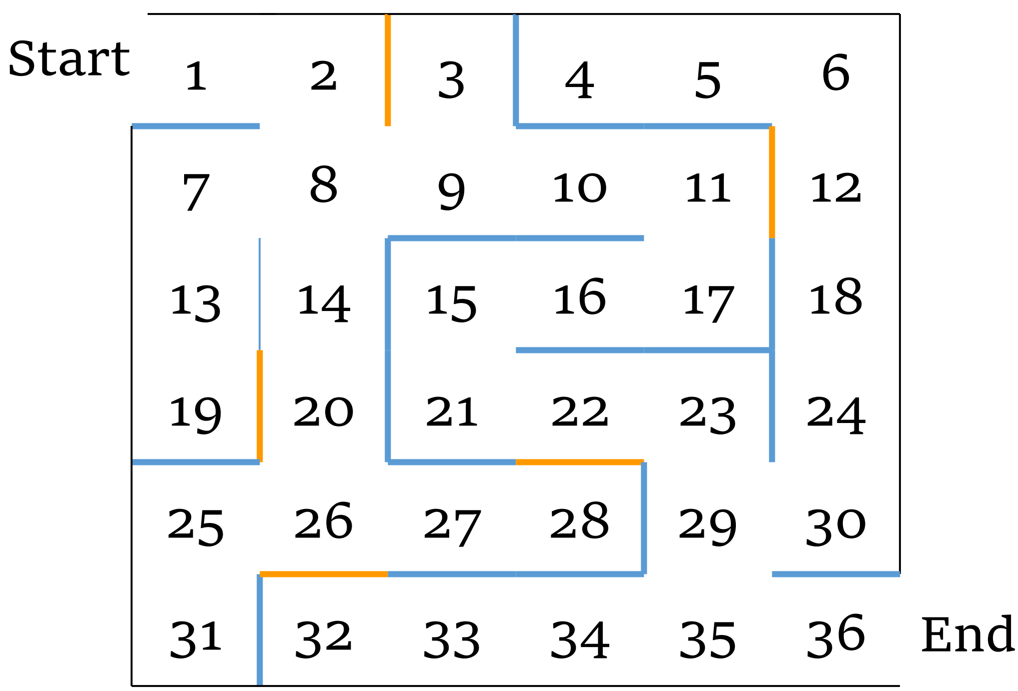
{1,2,7,8,9,13,19,14,20,26,27}, {3}, {4}, {5}, {6}, {10}, {11,17}, {12}, {15,16,21}

.

.

{22,23,24,29,30,32,33,34,35,36}

Example at the End



S
{1,2,3,4,5,6,7,... 36}

How to implement Find&Union

- two strategies
  - One ensures that the find can be executed in constant worst-case time
  - The other ensures that the union can be executed in constant worst-case time
  - Both (find and union) cannot be done simultaneously in constant worst-case time.

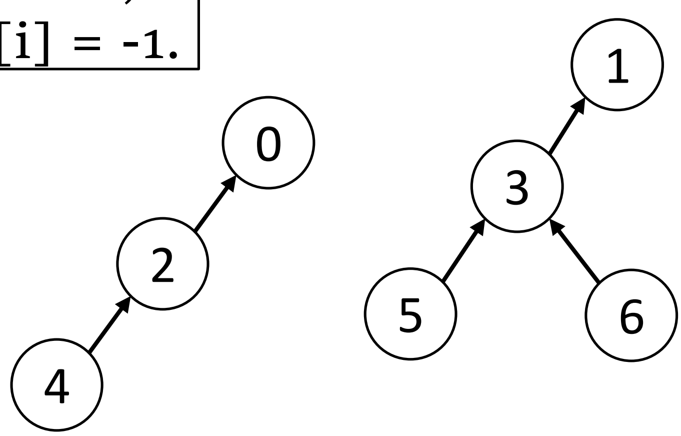
Up-Tree for D-U/F

- The union can be executed in constant worst-case time

  Up-Tree: use a tree to represent each set; each element has a parent link

element i	0	1	2	3	4	5	6
s[i]	**-1**	**-1**	**0**	**1**	**2**	**3**	**3**

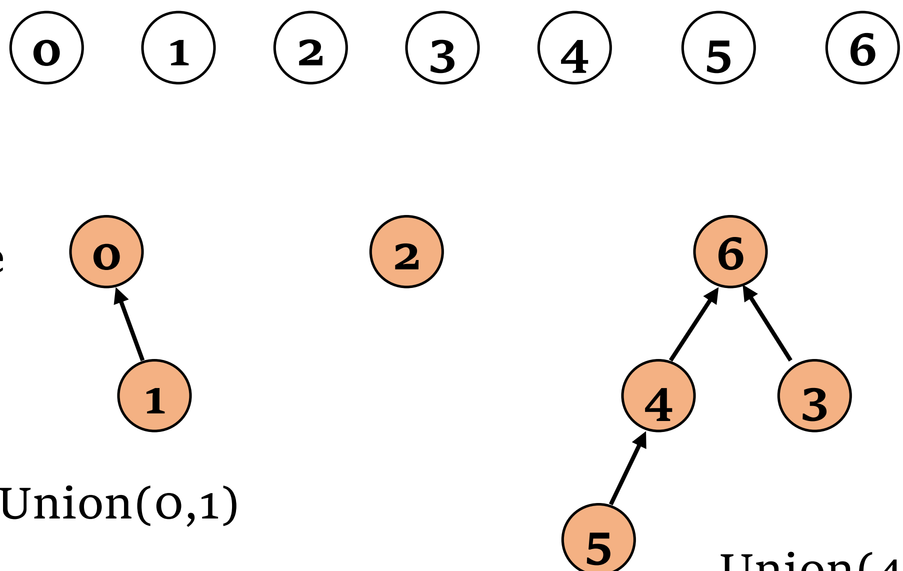
s[i] represents the parent of element i;
If i is a root, s[i] = -1.



D-U/F with Up-Tree

Initial state
(forest)

Intermediate
state



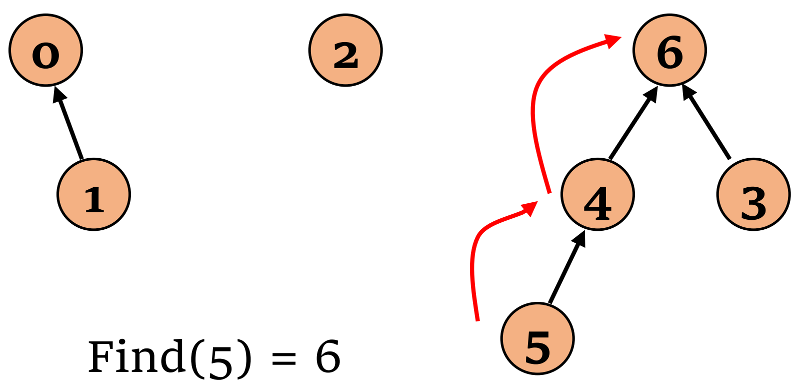
Union(0,1)

Union(4,5)
Union(6,3)
Union(6,4)

Roots are the names of each set.

Find Operation

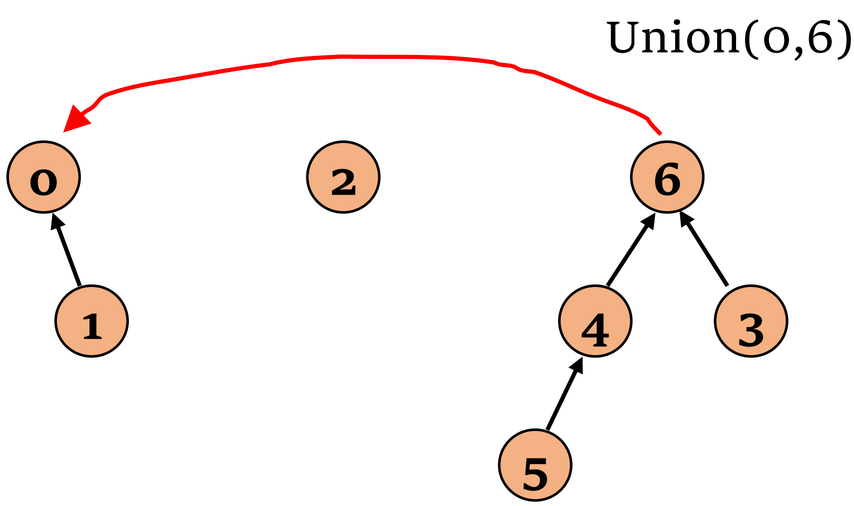
- Find(x) follow x to the root and return the root (which is the name of the class).



Find(5) = 6

Union Operation

- Union(i,j) - assuming i and j roots, point j to i.



Simple Implementation

```
class DisjSets{
public:
   explicit DisjSets( int numElements );
   int find( int x ) const;
   int find( int x );
   void unionSets( int root1, int root2 );
private:
   vector<int> s;
};

/**
 * Construct the disjoint sets object.
 * numElements is the initial number of
 * disjoint sets.
*/
DisjSets::DisjSets( int numElements ) :
s{ numElements, - 1 }
{
}
```

Union

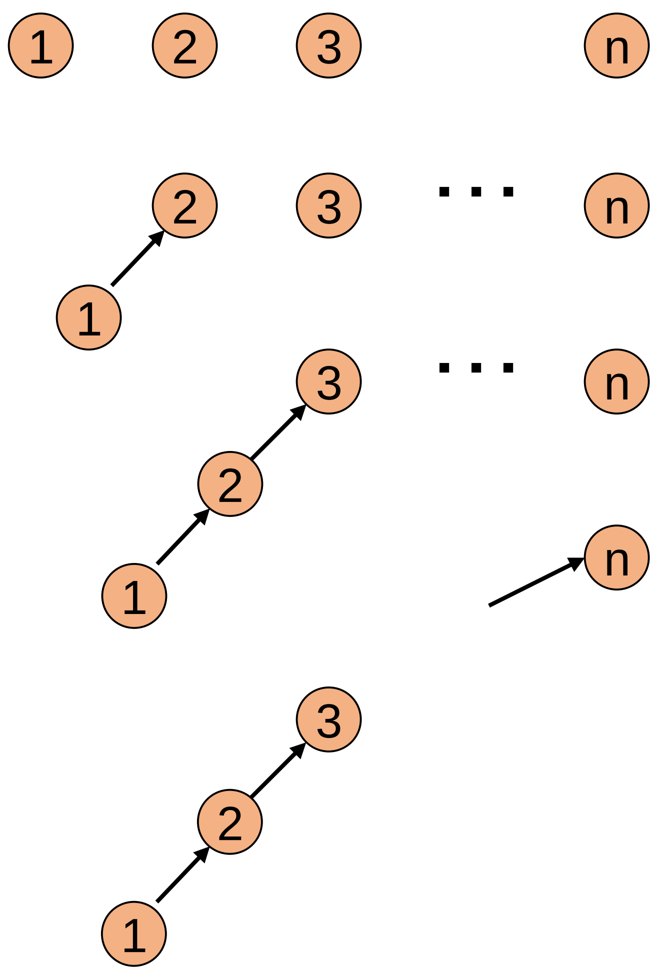
```
/**
 * Union two disjoint sets.
 * For simplicity, we assume root1 and
 * root2 are distinct and represent set
 * names.
 * root1 is the root of set 1.
 * root2 is the root of set 2.
 */
void DisjSets::unionSets( int root1, int root2 )
{
   s[ root2 ] = root1;
}
```

Constant Time!

Find

```
/**
 * Perform a find.
 * Error checks omitted again for simplicity.
 * Return the set containing x.
*/
int DisjSets::find( int x ) const
{
   if( s[ x ] < 0 )
      return x;
   else
      return find( s[ x ] );
}
```

A Bad Case



Union(1,2)

Union(2,3)

:

:

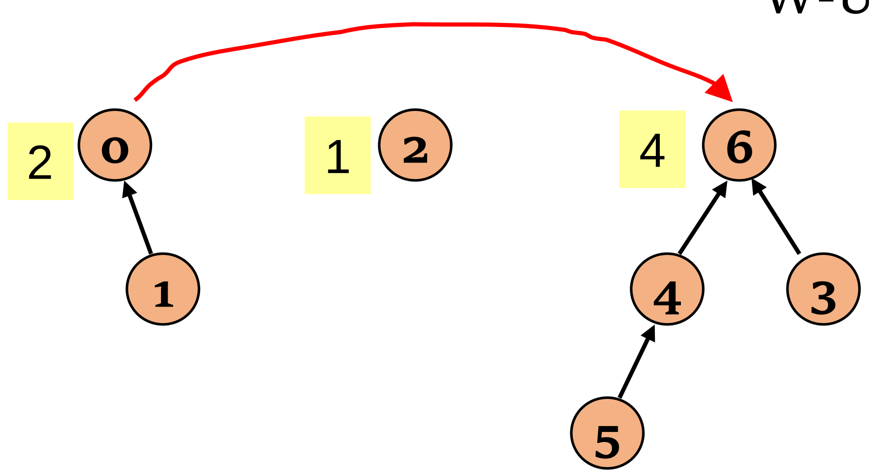
Union(n-1,n)

Find(1) n steps!!

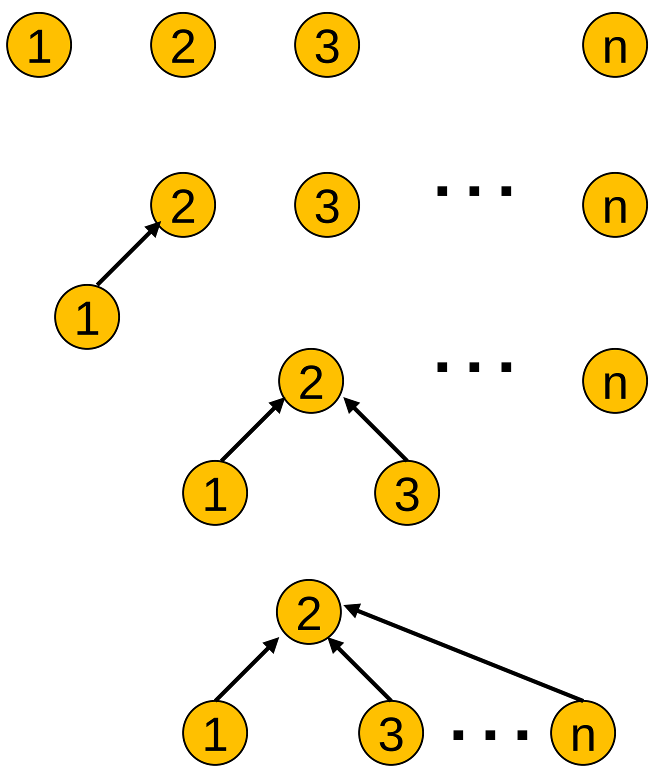
Weighted Union

- Weighted Union (weight = number of nodes)
  - Always point the smaller tree to the root of the larger tree (Union-by-Size)

W-Union(6,0)



Example Again



Union(1,2)

Union(2,3)

:

:

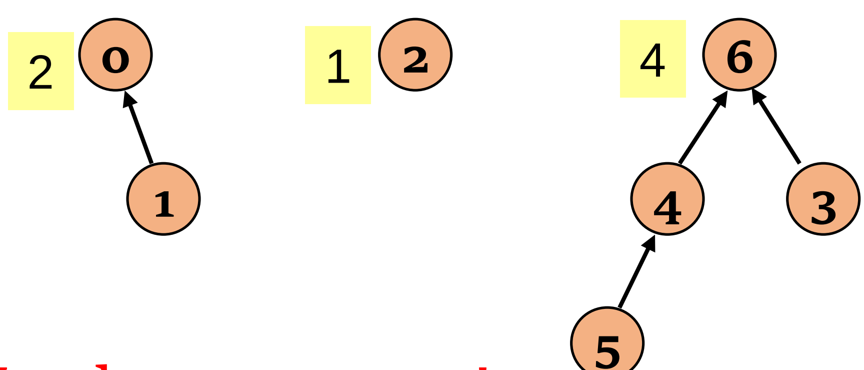
Union(n-1,n)

Find(1)   constant time

Weighted Union

- How to save weight information?

element i	0	1	2	3	4	5	6
s[i]	-1	0	-1	6	6	4	-1
weight	2		1				4

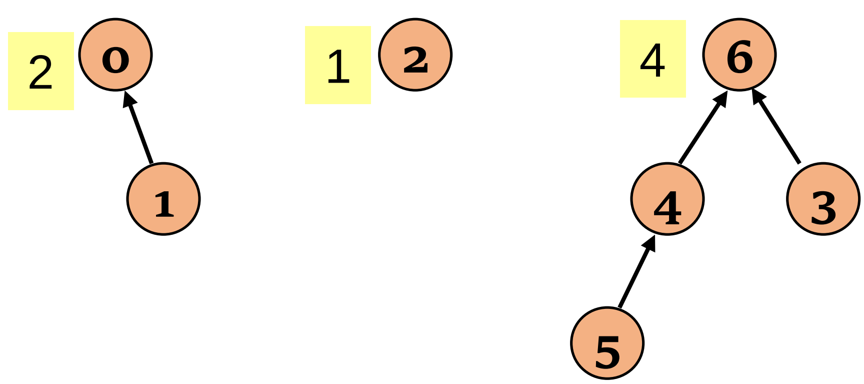


Need more space!

Weighted Union

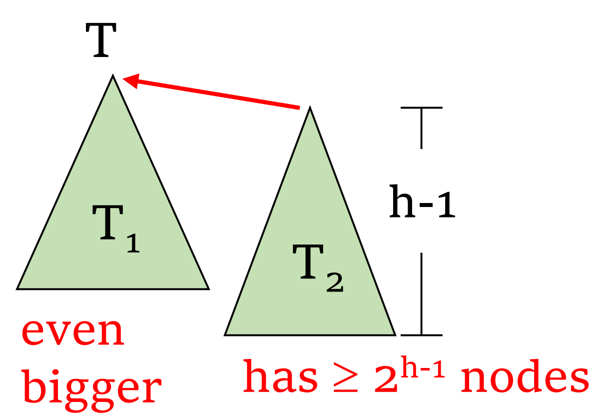
- Less space to save weight information.

element i	0	1	2	3	4	5	6
s[i]	-2	0	-1	6	6	4	-4



Analysis of Weighted Union

- With weighted union an up-tree of height h has weight at least 2^h.
- Proof by induction
  - Basis: h = 0. The up-tree has one node, $2^0 = 1$
  - Inductive step: Assume true for all h' < h.



Minimum weight up-tree of height h formed by weighted unions

$$W(T_1) \geq W(T_2) \geq 2^{h-1}$$

Weighted union

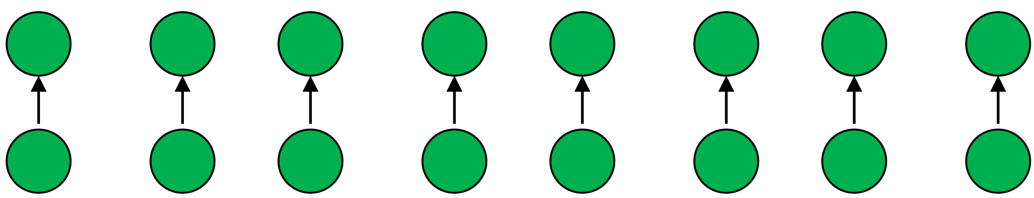
Induction hypothesis

Analysis of Weighted Union

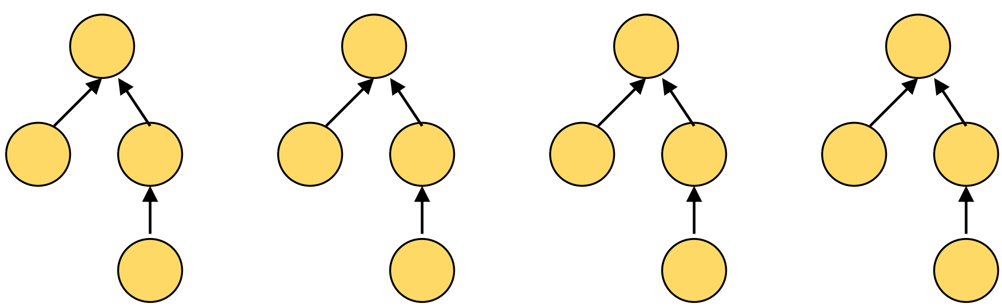
- Let T be an up-tree of weight n formed by weighted union. Let h be its height.
- $N \geq 2^h$
- $\log_2 N \geq h$
- Find(x) in tree T takes O(log N) time.
- Can we do better?

Worst Case for Weighted Union

N/2 Weighted Unions

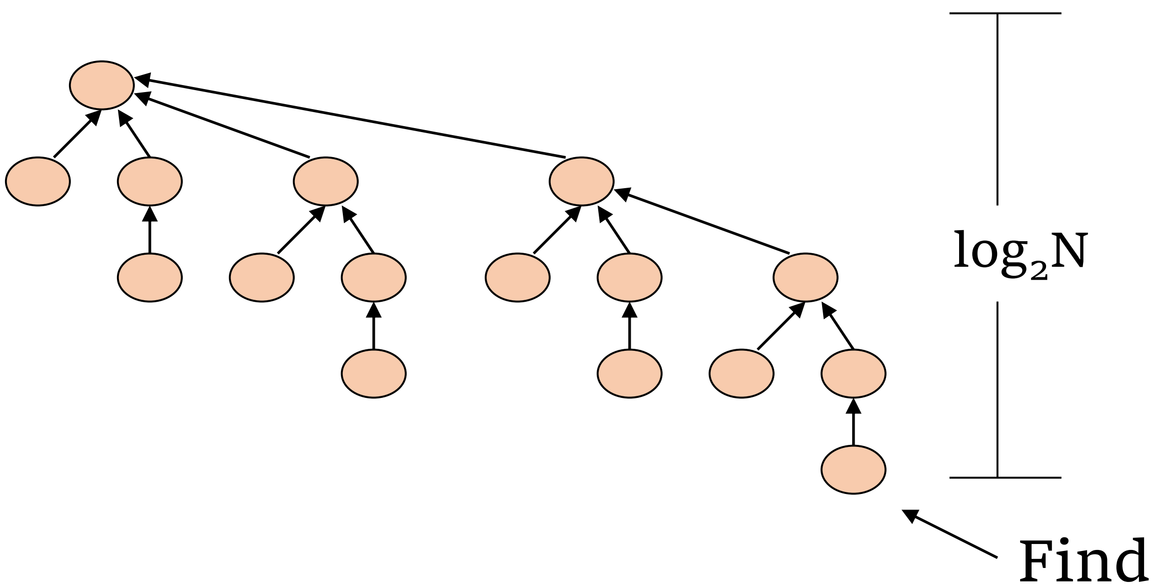


N/4 Weighted Unions



Example of Worst Cast (cont’)

After N -1 = N/2 + N/4 + …+ 1 Weighted Unions



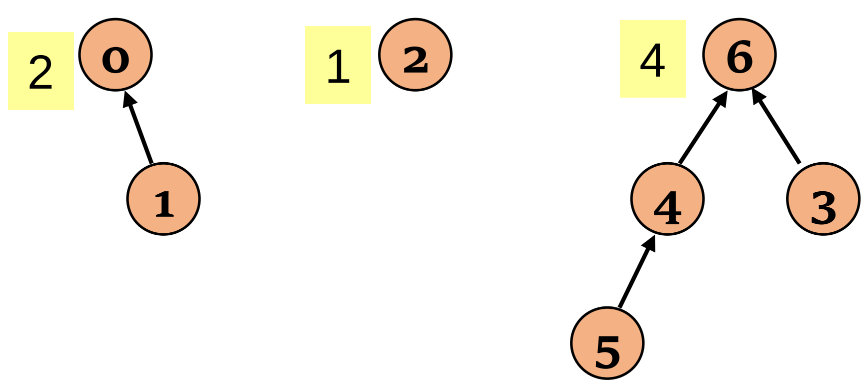
If there are $N = 2^k$ nodes then the longest path from leaf to root has length k.

An alternative implementation

- Union-by- height

element i	0	1	2	3	4	5	6
s[i]	**-2**	**0**	**-1**	**6**	**6**	**4**	**-3**

- height - 1

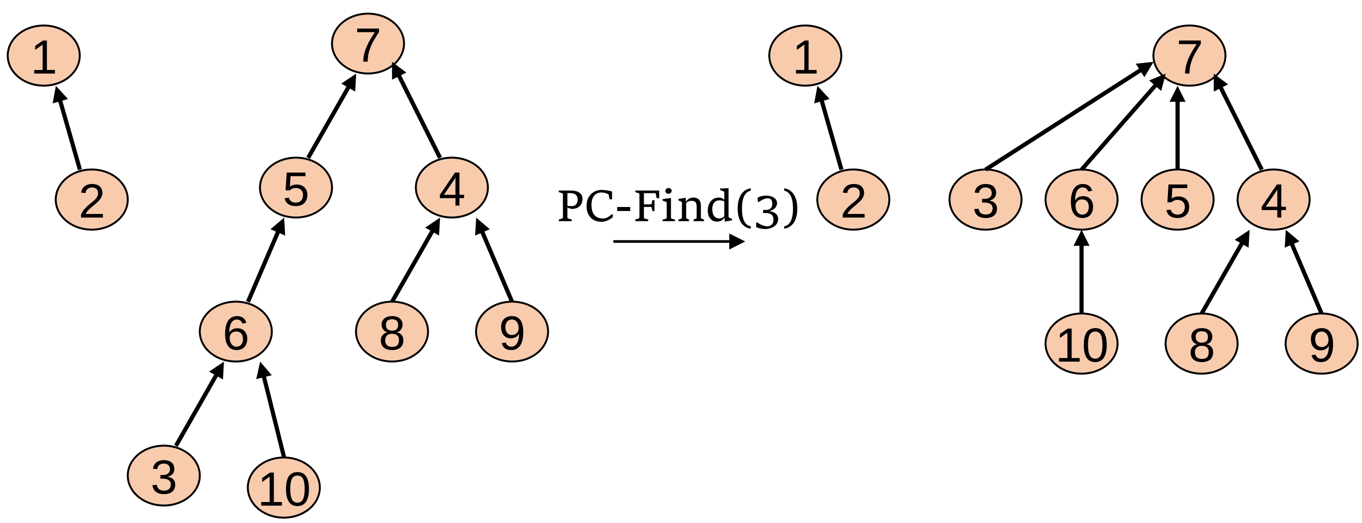


Union-by- height

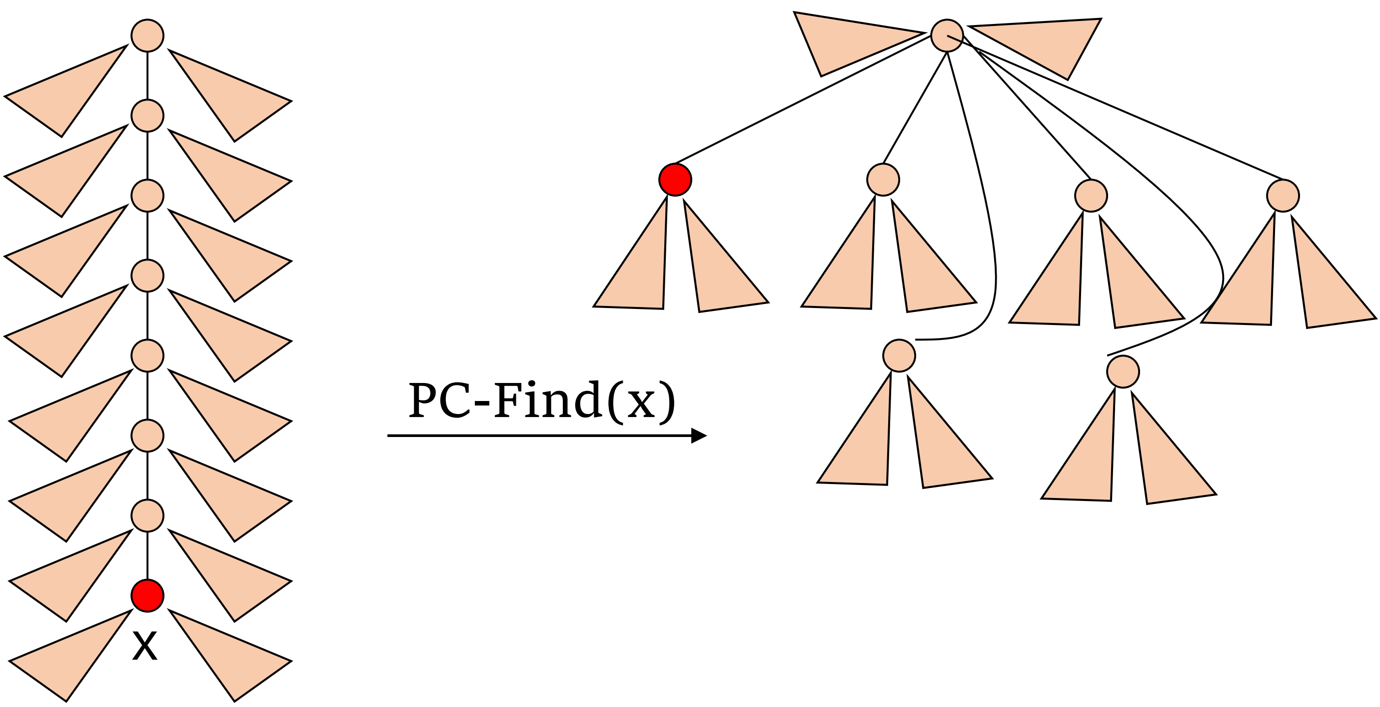
```
/**
 * Union two disjoint sets.
 * For simplicity, we assume root1 and root2
 * are distinct and represent set names.
 * root1 is the root of set 1.
 * root2 is the root of set 2.
*/
void DisjSets::unionSets( int root1, int root2 )
{
   if( s[ root2 ] < s[ root1 ] ) // root2 is deeper
      s[ root1 ] = root2; // Make root2 new root
   else
   {
      if( s[ root1 ] == s[ root2 ] )
      --s[ root1 ]; // Update height if same
      s[ root2 ] = root1; // Make root1 new root
   }
}
```

Path Compression

- On a Find operation point all the nodes on the search path directly to the root.



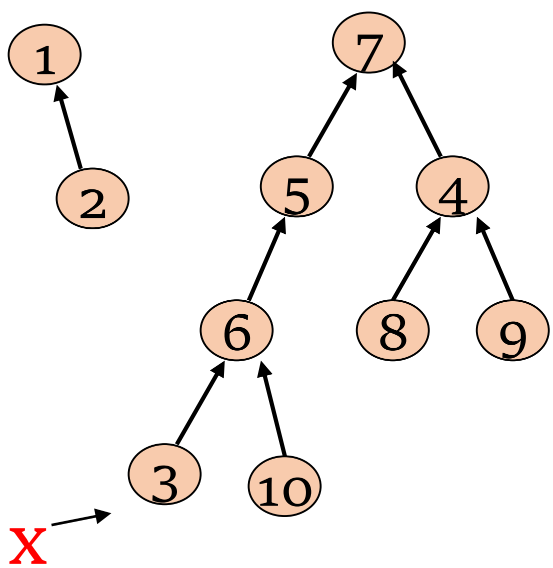
Self-Adjustment Works



Path Compression Find

```
/**
  * Perform a find with path compression.
  * Error checks omitted again for simplicity.
  * Return the set containing x.
*/
int DisjSets::find( int x )
{
    if( s[ x ] < 0 )
        return x;
    else
        return s[ x ] = find( s[ x ] );
}
```

Example



return s[x] = find(s[x]);

Disjoint Union / Find with Weighted Union and Path Compression

- Worst case time complexity for a W-Union is $O(1)$ and for a PC-Find is $O(\log N)$.

- Time complexity for $m \geq N$ operations on N elements is $O(m \log^* N)$.
  - Here, $\log^* N$ is iterated logarithm and a very slow growing function. ($\log^* 2^{65536} = 5$)
  - Essentially constant time per operation!

Amortized Complexity

- For disjoint union / find with weighted union and path compression.
  - average time per operation is essentially a constant.
  - worst case time for a PC-Find is O(log N).

- An individual operation can be costly, but over time the average cost per operation is not.

Homework 6

- Textbook Exercises 8.1,8.2