CSE 326: Data Structures
Disjoint Union/Find

James Fogarty

Autumn 2007

Equivalence Relations

Relation *R* :

- For every pair of elements *(a, b)* in a set S, a *R* b is either true or false.
- If a *R* b is true, then a *is related* to b.

An equivalence relation satisfies:

1. (Reflexive) a *R* a
2. (Symmetric) a *R* b iff b *R* a
3. (Transitive) a *R* b and b *R* c implies a *R* c

A new question

- Which of these things are similar?

  { grapes, blackberries, plums, apples,
  oranges, peaches, raspberries, lemons }

- If limes are added to this fruit salad, and are similar to oranges, then are they similar to grapes?

- How do you answer these questions efficiently?

Equivalence Classes

- Given a set of things…
  { grapes, blackberries, plums, apples, oranges, peaches, raspberries, lemons, bananas }

- …define the equivalence relation
  All citrus fruit is related, all berries, all stone fruits, and THAT'S IT.

- …partition them into related subsets
  { grapes }, { blackberries, raspberries }, { oranges, lemons }, { plums, peaches }, { apples }, { bananas }

Everything in an equivalence class is related to each other.

Determining equivalence classes

- Idea: give every equivalence class a name
  - { oranges, limes, lemons } = “like-ORANGES”
  - { peaches, plums } = “like-PEACHES”
  - Etc.
- To answer if two fruits are related:
  - FIND the name of one fruit’s e.c.
  - FIND the name of the other fruit’s e.c.
  - Are they the same name?

Building Equivalence Classes

- Start with disjoint, singleton sets:
  - { apples }, { bananas }, { peaches }, …
- As you gain information about the relation, UNION sets that are now related:
  - { peaches, plums }, { apples }, { bananas }, …
- E.g. if peaches R limes, then we get
  - { peaches, plums, limes, oranges, lemons }

Disjoint Union - Find

- Maintain a set of pairwise disjoint sets.
  - {3,5,7} , {4,2,8}, {9}, {1,6}
- Each set has a unique name, one of its members
  - {3,<u>5</u>,7} , {4,2,<u>8</u>}, {<u>9</u>}, {<u>1</u>,6}

Union

- Union(x,y) – take the union of two sets named x and y
  - {3,5,7} , {4,2,8}, {9}, {1,6}
  - Union(5,1)

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

Find

- Find(x) – return the name of the set containing x.
  - {3,5,7,1,6}, {4,2,8}, {9},
  - Find(1) = 5
  - Find(4) = 8

Example

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}

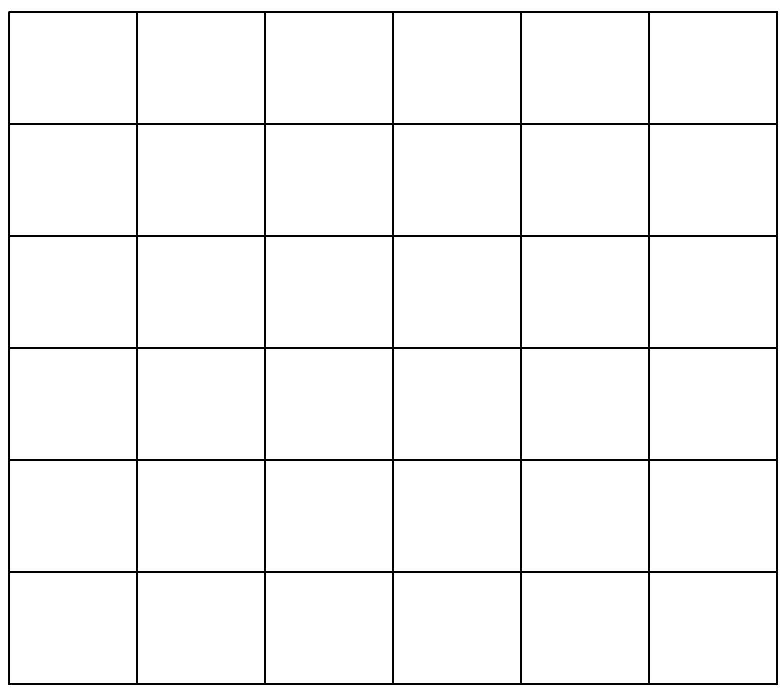
Find(8) = 7
Find(14) = 20

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}

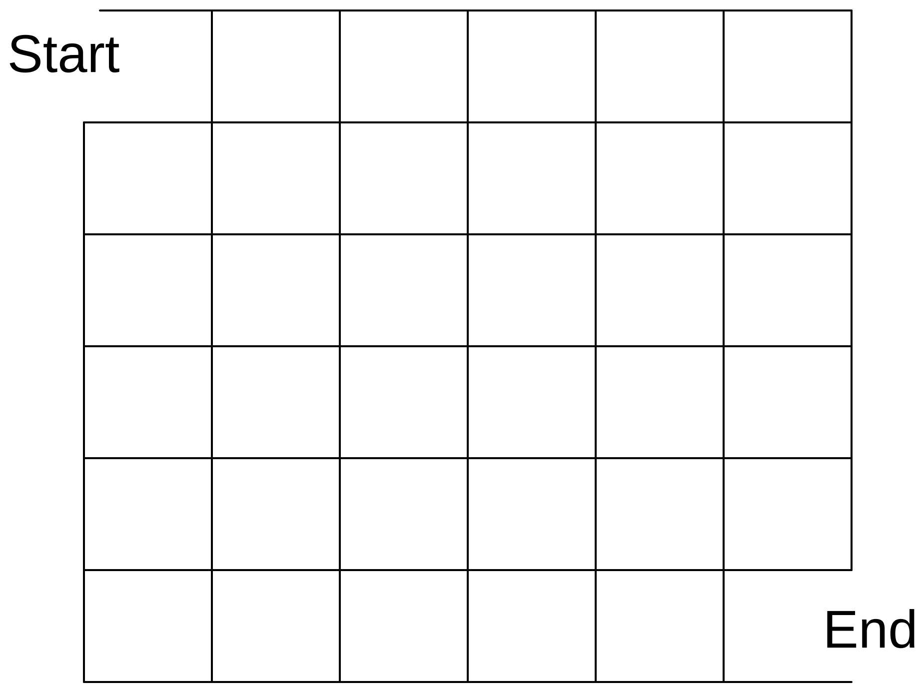
Cute Application

- Build a random maze by erasing edges.



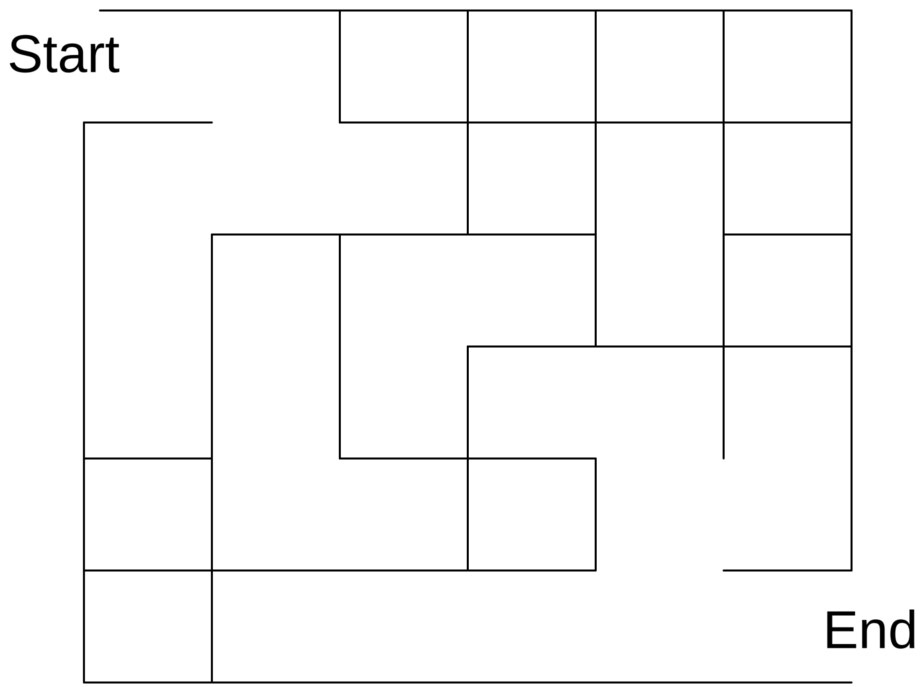
Cute Application

- Pick Start and End



Cute Application

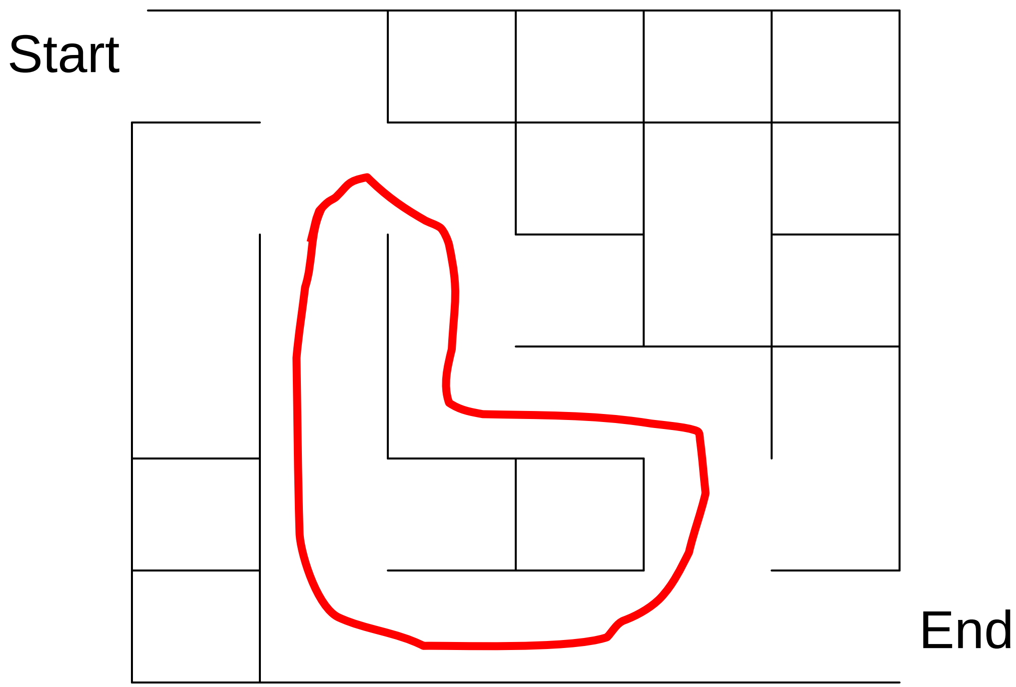
- Repeatedly pick random edges to delete.



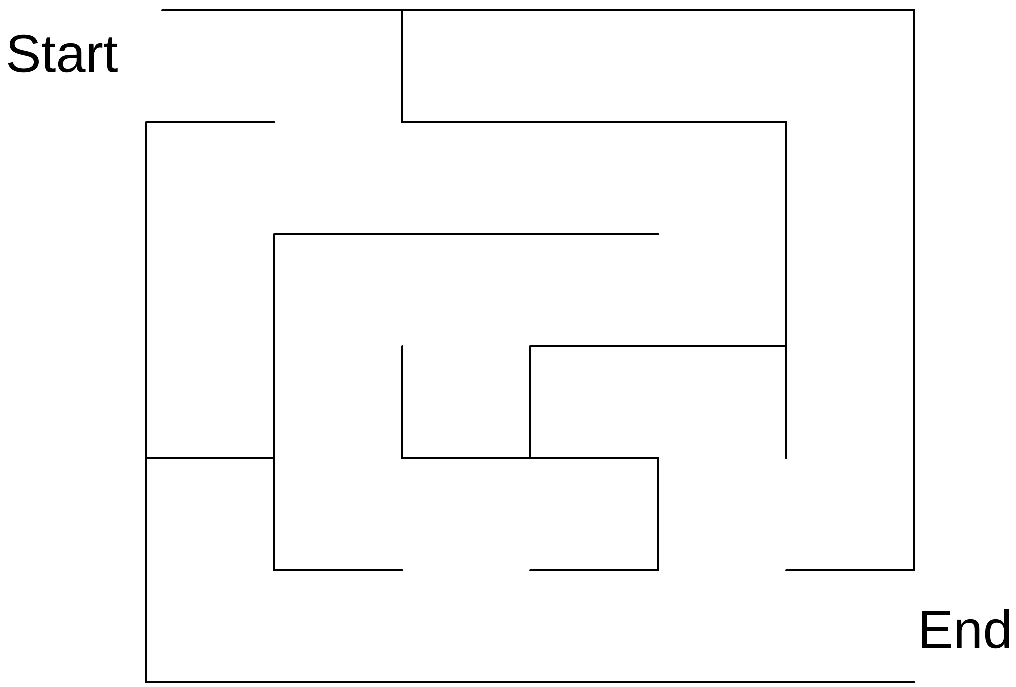
Desired Properties

- 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.

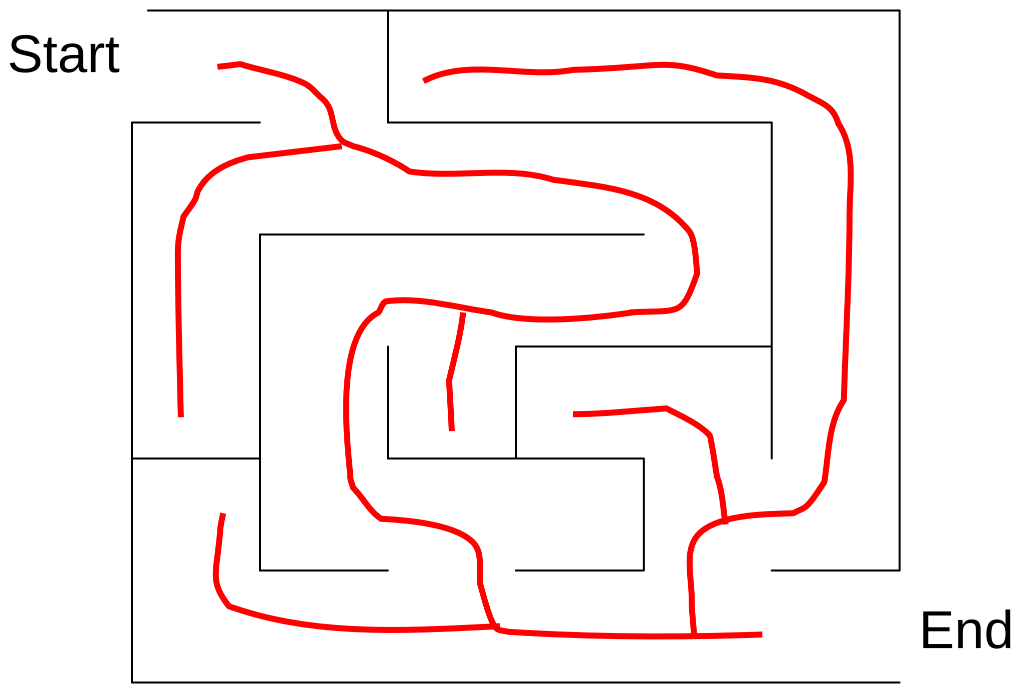
A Cycle



A 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

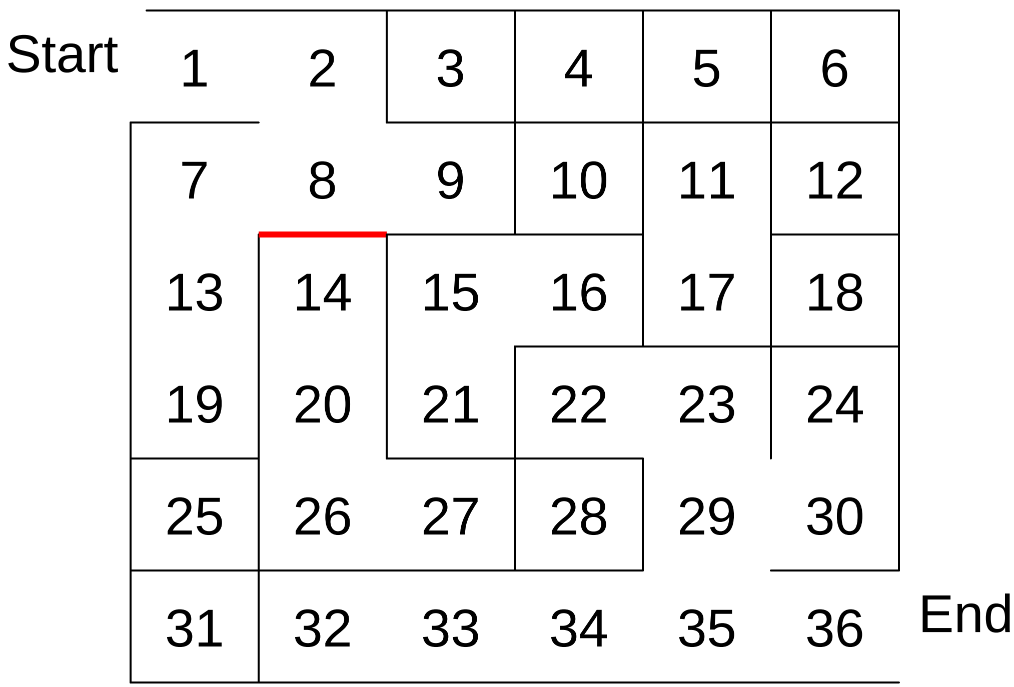
Basic Algorithm

- 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)
  else
     add (x,y) to Maze
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

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}

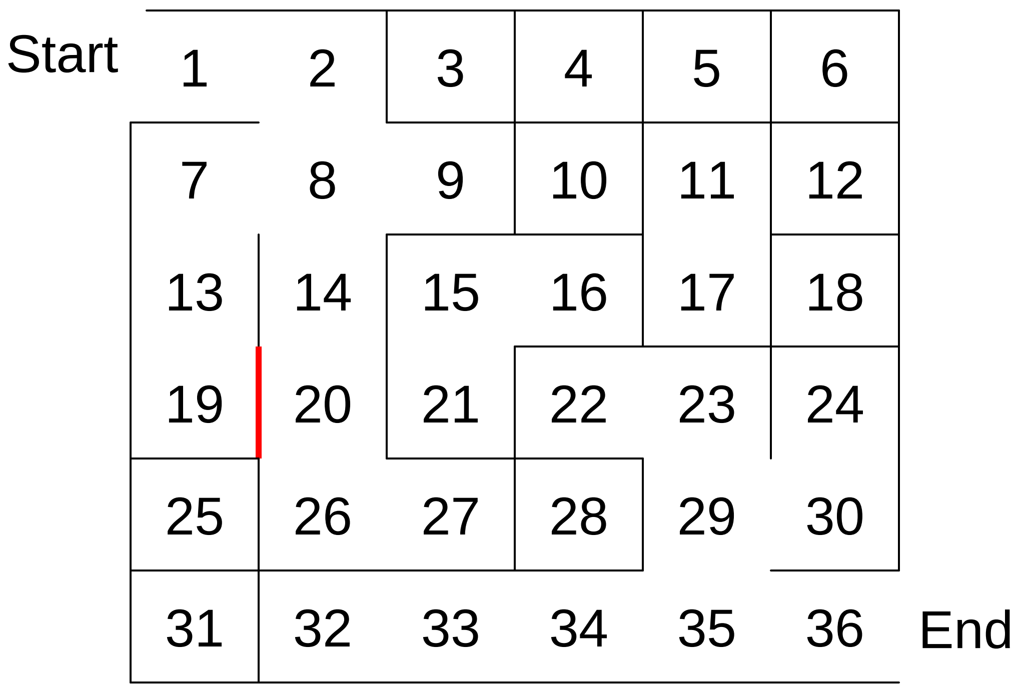
.

.

{22,23,24,29,39,32
33,34,35,36}

Example

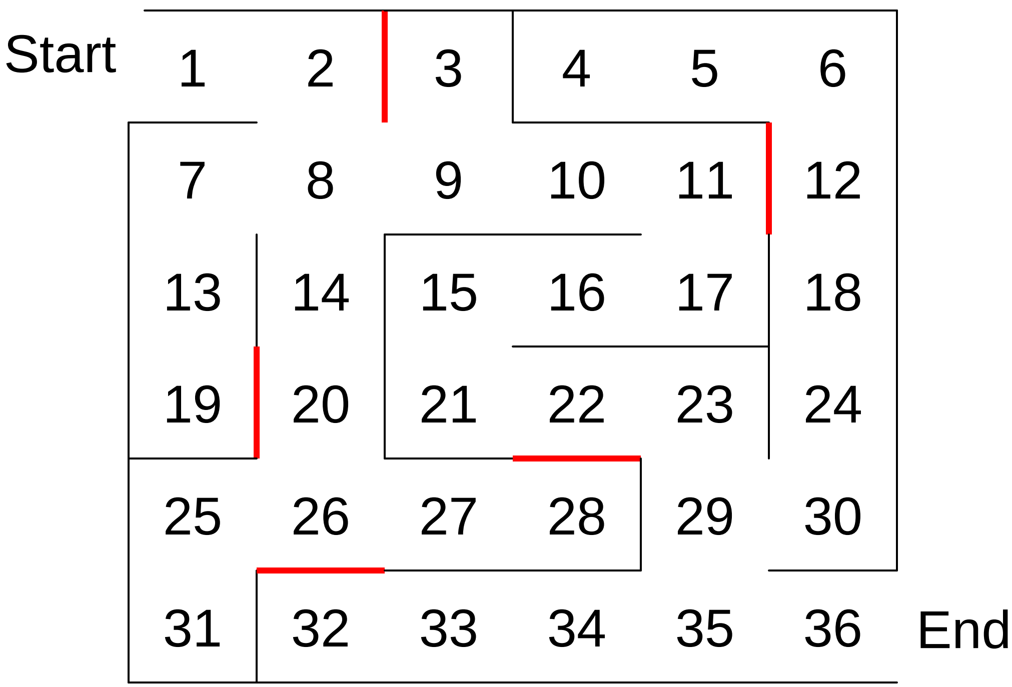
Pick (19,20)



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

Example at the End

S
{1,2,3,4,5,6,7,… 36}



E
Maze

Implementing the DS ADT

- n elements,
  Total Cost of: m finds, $\leq n$-1 unions

- Target complexity: $O(m+n)$
  i.e. $O(1)$ amortized

- $O(1)$ worst-case for find as well as union would be great, but…
  Known result: find and union *cannot* both be done in worst-case $O(1)$ time

Implementing the DS ADT

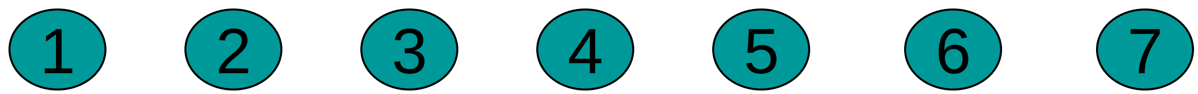
- Observation: *trees* let us find many elements given one root…

- Idea: if we *reverse* the pointers (make them point up from child to parent), we can find a single root from many elements…

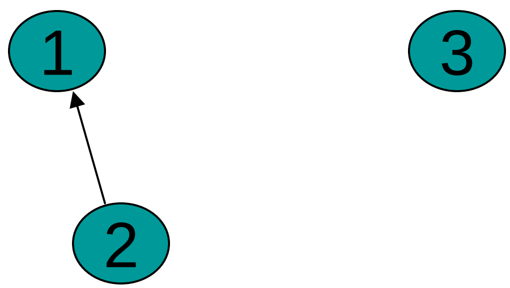
- Idea: Use one tree for each equivalence class. The name of the class is the tree root.

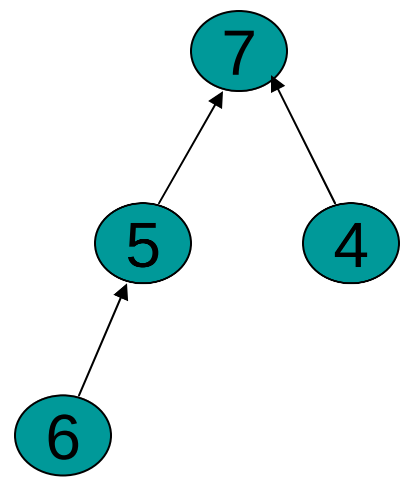
Up-Tree for DU/F

Initial state



Intermediate state

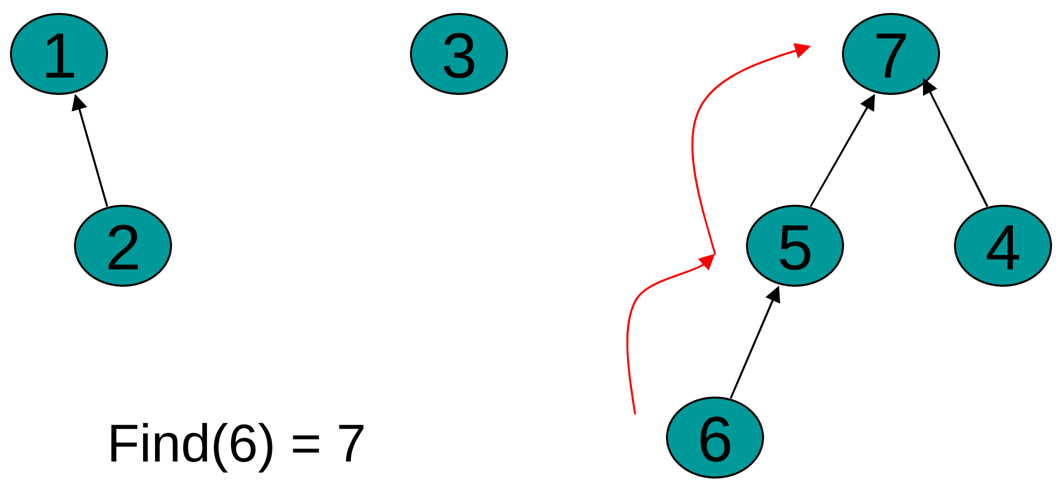




Roots are the names of each set.

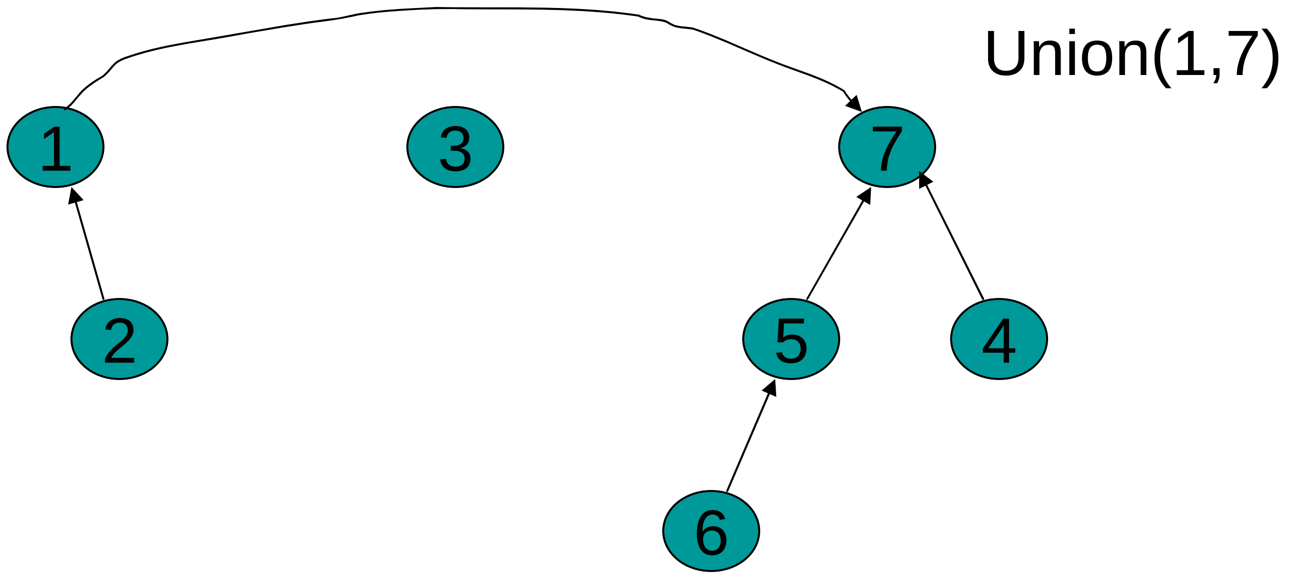
Find Operation

- Find(x) follow x to the root and return the root



Union Operation

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

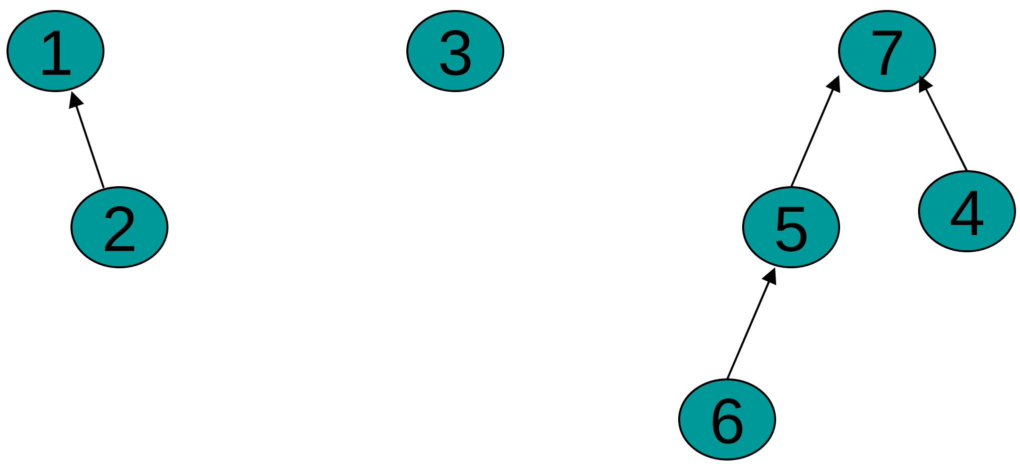


Simple Implementation

- Array of indices

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

Up[x] = 0 means x is a root.



Union

```
Union(up[] : integer array, x,y : integer) : {
//precondition: x and y are roots//
Up[x] := y
}
```

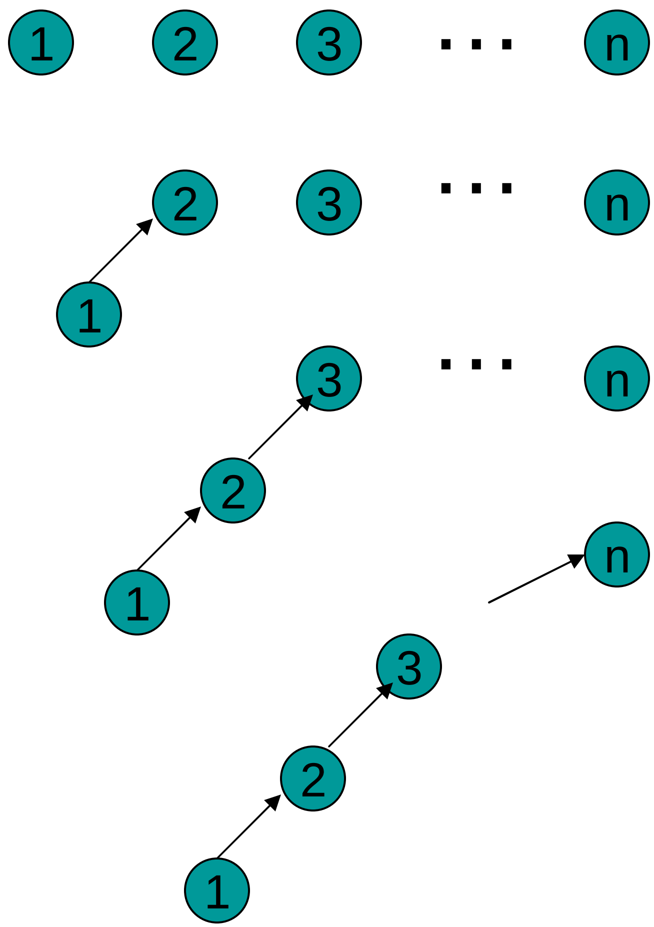
Constant Time!

Exercise

- Design Find operator
  - Recursive version
  - Iterative version

```
Find(up[] : integer array, x : integer) : integer {
//precondition: x is in the range 1 to size//
???
}
```

A Bad Case



Union(1,2)

Union(2,3)

:

:

Union(n-1,n)

Find(1)   n steps!!

Now this doesn't look good ☹

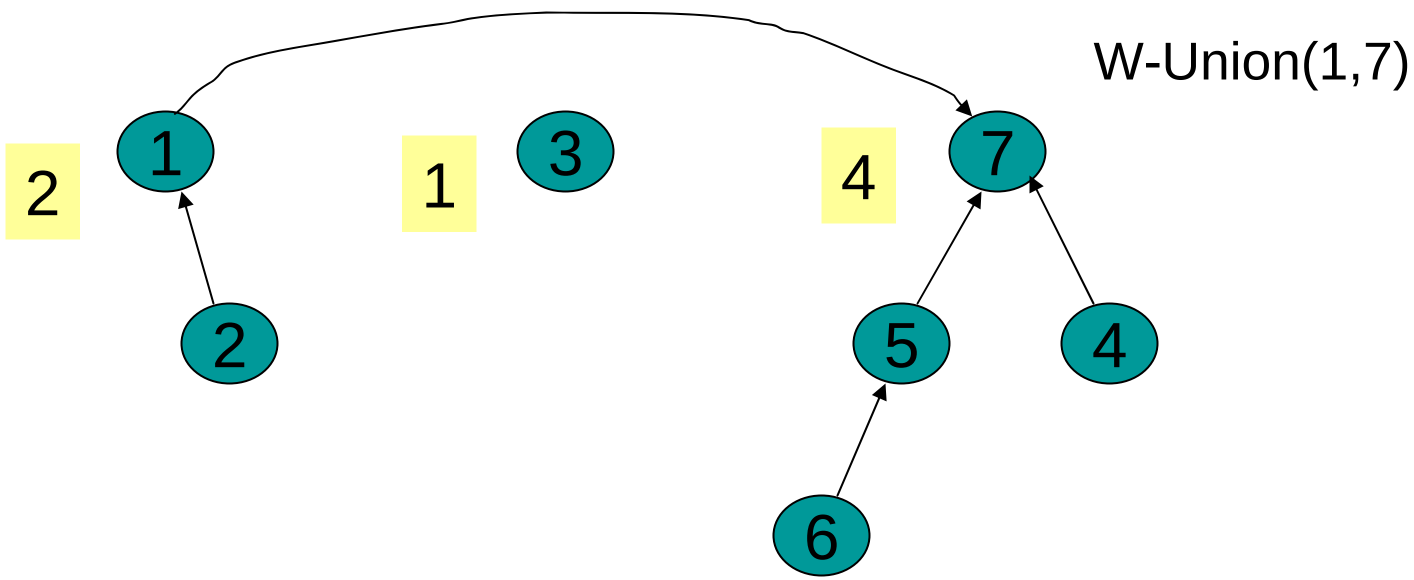
Can we do better? *Yes!*

1. Improve union so that *find* only takes $\Theta(\log n)$
   - Union-by-size
   - Reduces complexity to $\Theta(m \log n + n)$

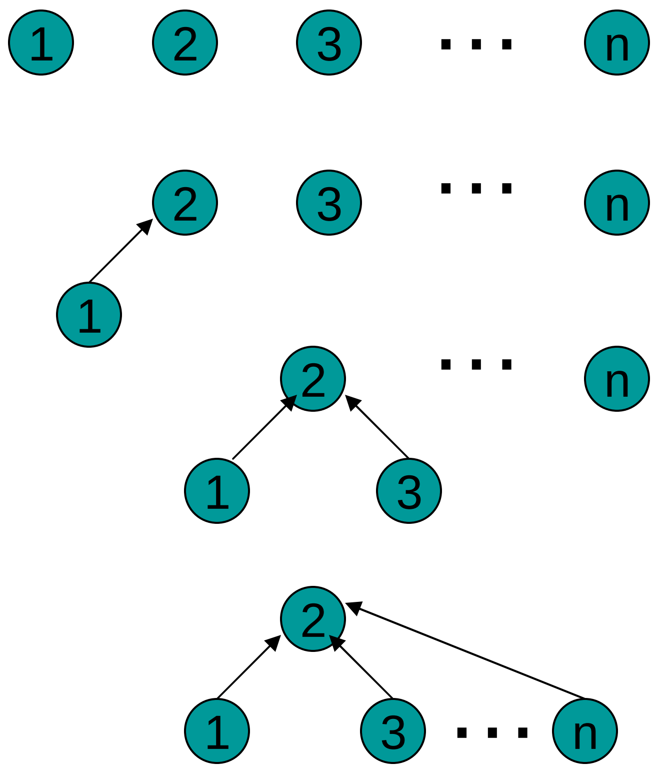
2. Improve find so that it becomes even better!
   - Path compression
   - Reduces complexity to <u>almost</u> $\Theta(m + n)$

Weighted Union

- Weighted Union
  - Always point the smaller tree to the root of the larger tree



Example Again



Union(1,2)

Union(2,3)

:

:

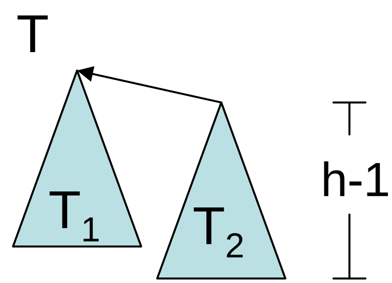
Union(n-1,n)

Find(1) constant time

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



$$W(T_1) \underset{\text{Weighted union}}{\geq} W(T_2) \underset{\text{Induction hypothesis}}{\geq} 2^{h-1}$$

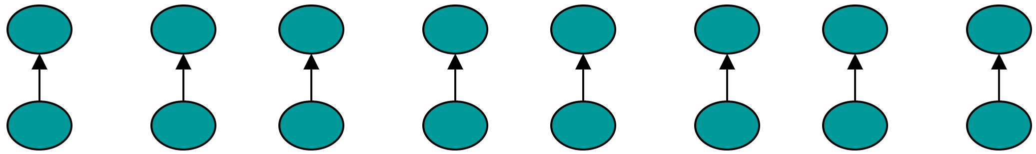
$$W(T) \geq 2^{h-1} + 2^{h-1} = 2^h$$

Analysis of Weighted Union

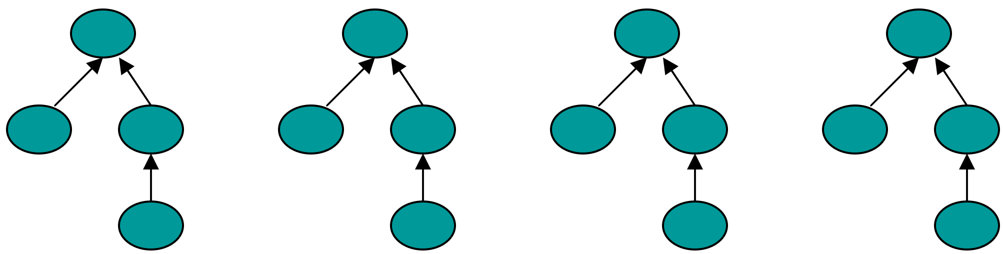
- Let T be an up-tree of weight n formed by weighted union. Let h be its height.
- $n \geq 2^h$
- $\log_2 n \geq h$
- Find(x) in tree T takes O(log n) time.
- Can we do better?

Worst Case for Weighted Union

n/2 Weighted Unions

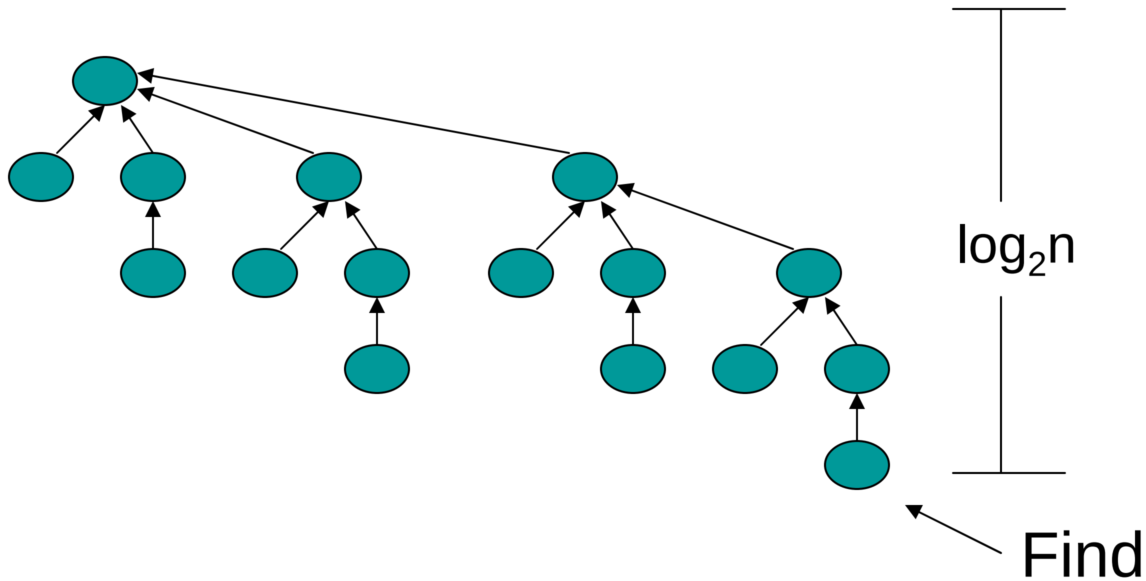


n/4 Weighted Unions



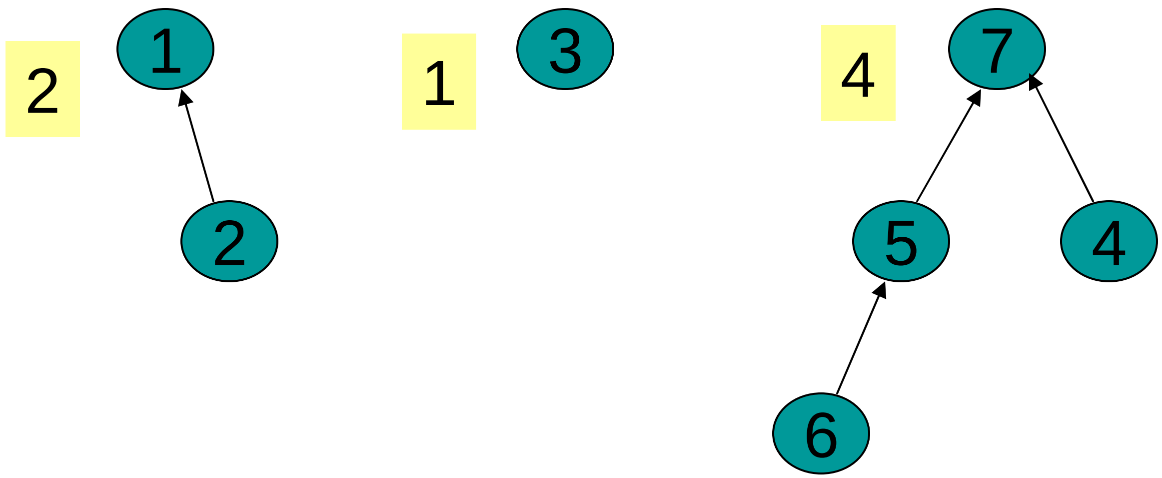
Example of Worst Cast (cont')

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.

Elegant Array Implementation



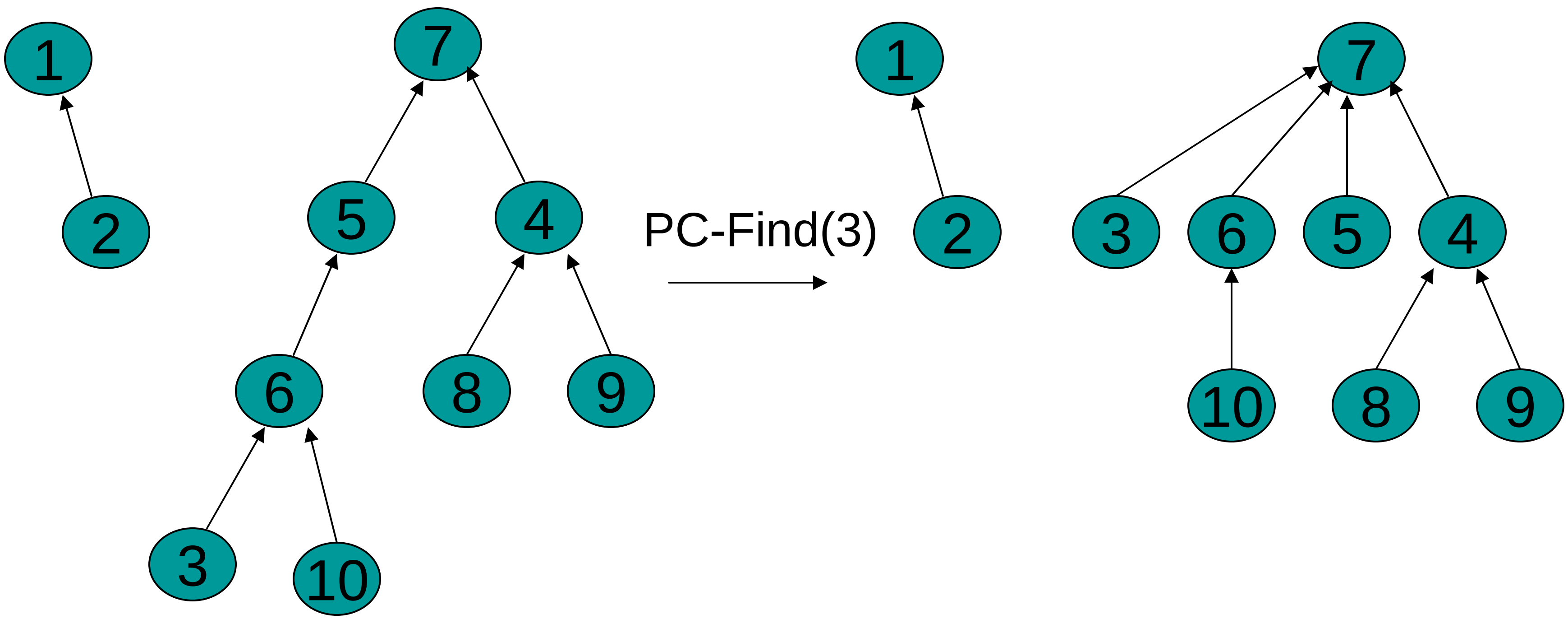
	1	2	3	4	5	6	7
up	0	1	0	7	7	5	0
weight	2		1				4

Weighted Union

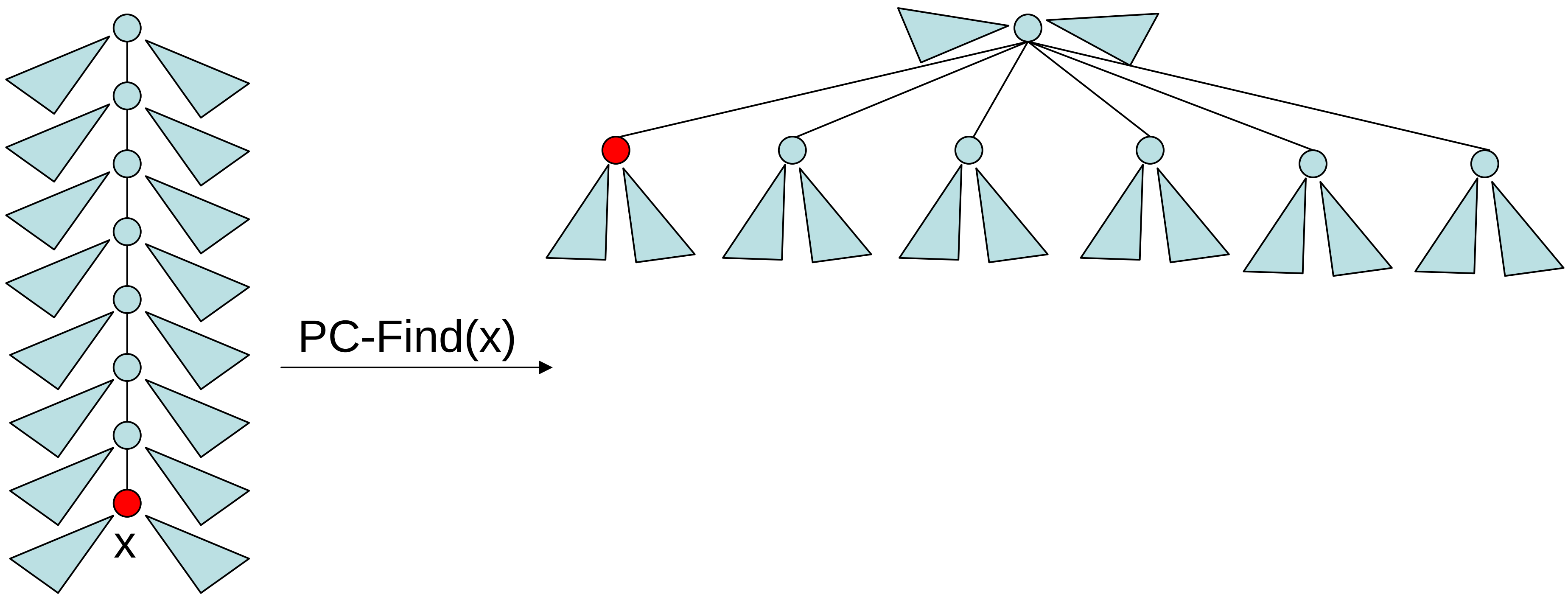
```
W-Union(i,j : index){
//i and j are roots//
  wi := weight[i];
  wj := weight[j];
  if wi < wj then
    up[i] := j;
    weight[j] := wi + wj;
  else
    up[j] :=i;
    weight[i] := wi + wj;
}
```

Path Compression

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

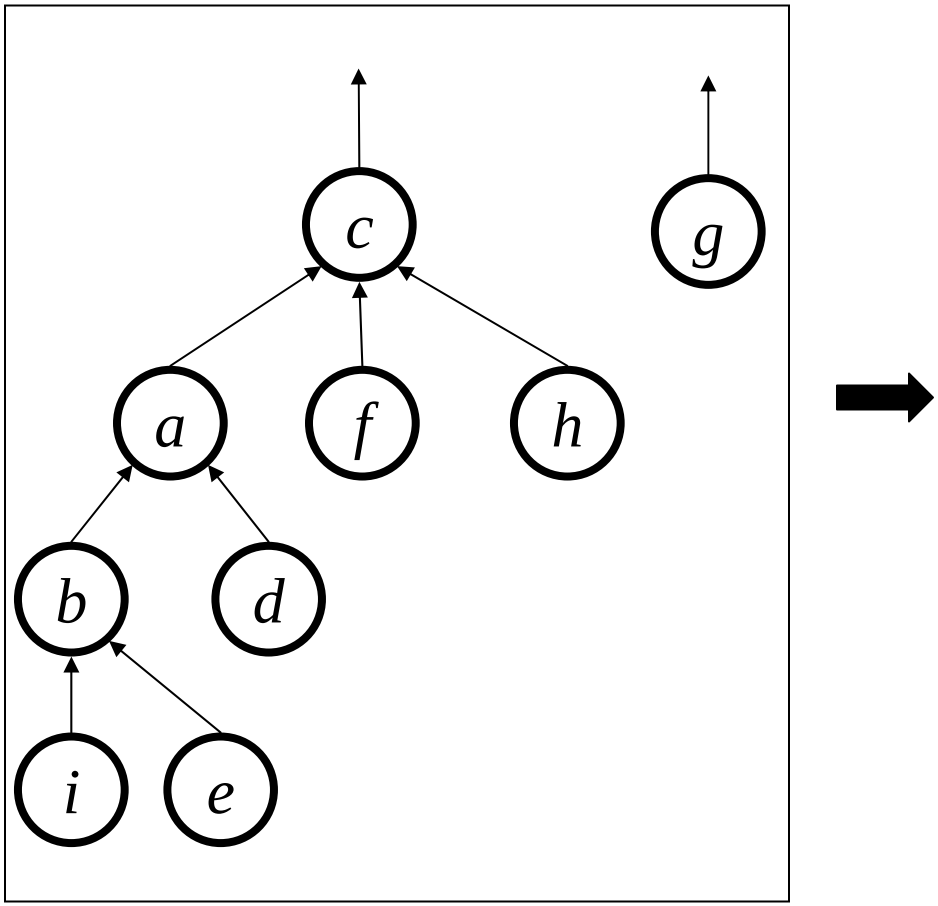


Self-Adjustment Works



Student Activity

Draw the result of Find(e):



Path Compression Find

```
PC-Find(i : index) {
  r := i;
  while up[r] ≠ 0 do //find root//
    r := up[r];
  if i ≠ r then  //compress path//
    k := up[i];
    while k ≠ r do
      up[i] := r;
      i := k;
      k := up[k]
  return(r)
}
```

Interlude: A Really Slow Function

Ackermann's function is a <u>really</u> big function $A(x, y)$ with inverse $\alpha(x, y)$ which is <u>really</u> small

How fast does $\alpha(x, y)$ grow?

$\alpha(x, y) = 4$ for x **far** larger than the number of atoms in the universe (2^{300})

α shows up in:

- Computation Geometry (surface complexity)
- Combinatorics of sequences

A More Comprehensible Slow Function

log* x = number of times you need to compute log to bring value down to at most 1

E.g. log* 2 = 1

log* 4 = log* 2^2 = 2

log* 16 = log* 2^{2^2} = 3 (log log log 16 = 1)

log* 65536 = log* $2^{2^{2^2}}$ = 4 (log log log log 65536 = 1)

log* 2^{65536} = …………… = 5

Take this: $\alpha(m,n)$ grows even slower than log* n !!

Disjoint Union / Find with Weighted Union and PC

- 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)
  - Log * n < 7 for all reasonable n. Essentially constant time per operation!
- Using "ranked union" gives an even better bound theoretically.

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.

Find Solutions

Recursive

```
Find(up[] : integer array, x : integer) : integer {
//precondition: x is in the range 1 to size//
if up[x] = 0 then return x
else return Find(up,up[x]);
}
```

Iterative

```
Find(up[] : integer array, x : integer) : integer {
//precondition: x is in the range 1 to size//
while up[x] ≠ 0 do
  x := up[x];
return x;
}
```