Data Structure for Disjoint Sets

Chapter 21 (omit 21.4)

Disjoint Set Data Structure

- Its implementation
- Its operations

Relations

- R: a relation defined on a set S of elements
- aRb is true (where $a,b \in S$) $\Rightarrow$ a is related to b by R

Properties

- R is reflexive if aRa is true $\forall\ a \in S$
- R is symmetric if (aRb is true $\Leftrightarrow$ bRa is true) $\forall\ a,b \in S$
- R is transitive if (aRb is true and bRc is true $\Rightarrow$ aRc is true) $\forall\ a,b,c \in S$
- R is an Equivalence Relation (ER) if it is reflexive, symmetric and transitive

Properties

	reflexive	symmetric	transitive	equivalence
=	Y	Y	Y	Y
<	N	N	Y	N
>=	Y	N	Y	N
married	N	Y	?	N
sibling-of	Y	Y	Y	Y
father-of	N	N	N	N

Equivalence Problem

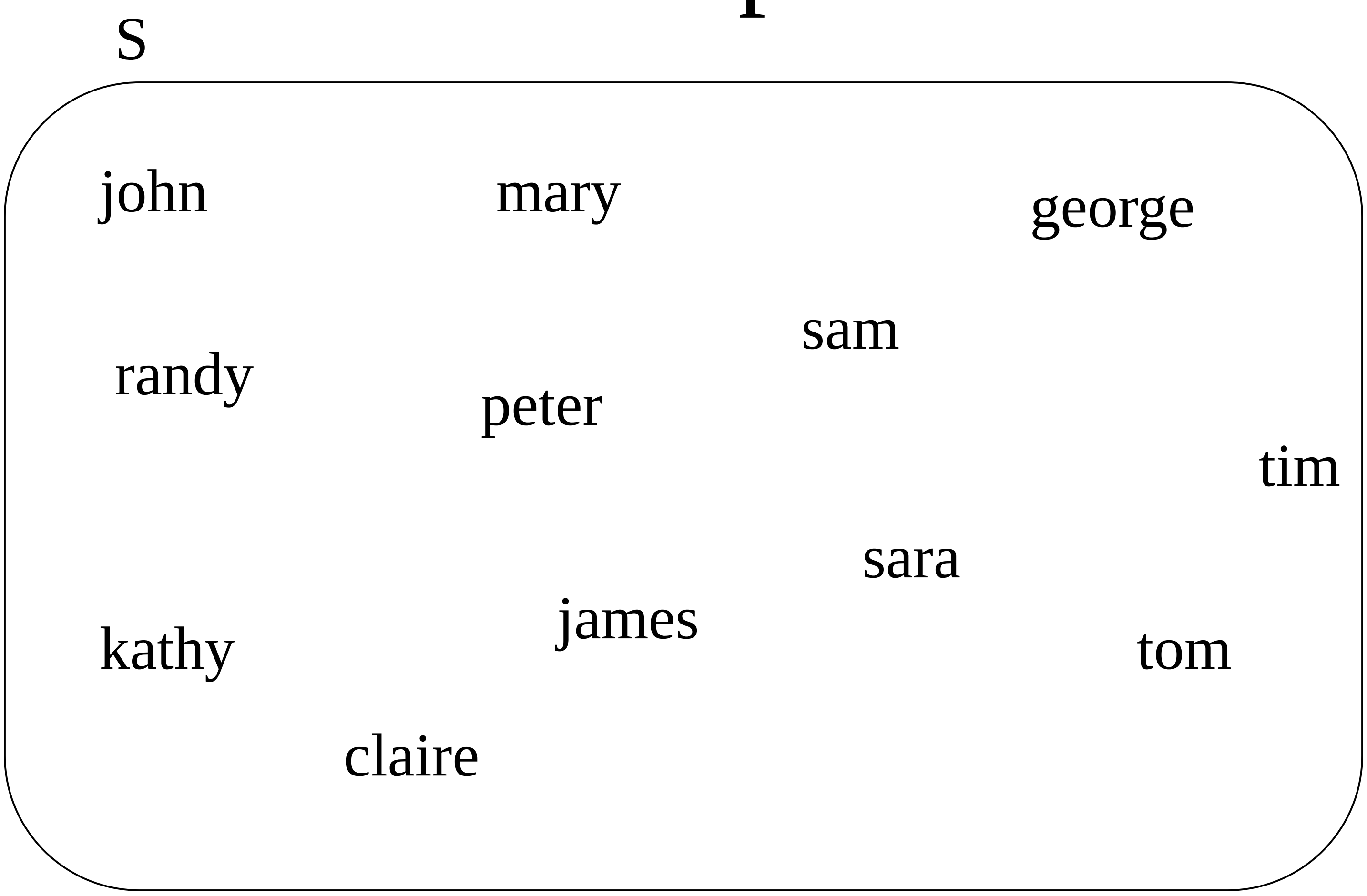
- If R is an ER defined over a set of objects S, for every pair (a, b) where $a,b \in S$, determine if aRb is true or false

Equivalence Class (EC)

- The EC of an element $a \in S$ is the subset of S containing all elements that are related to a by the ER R.

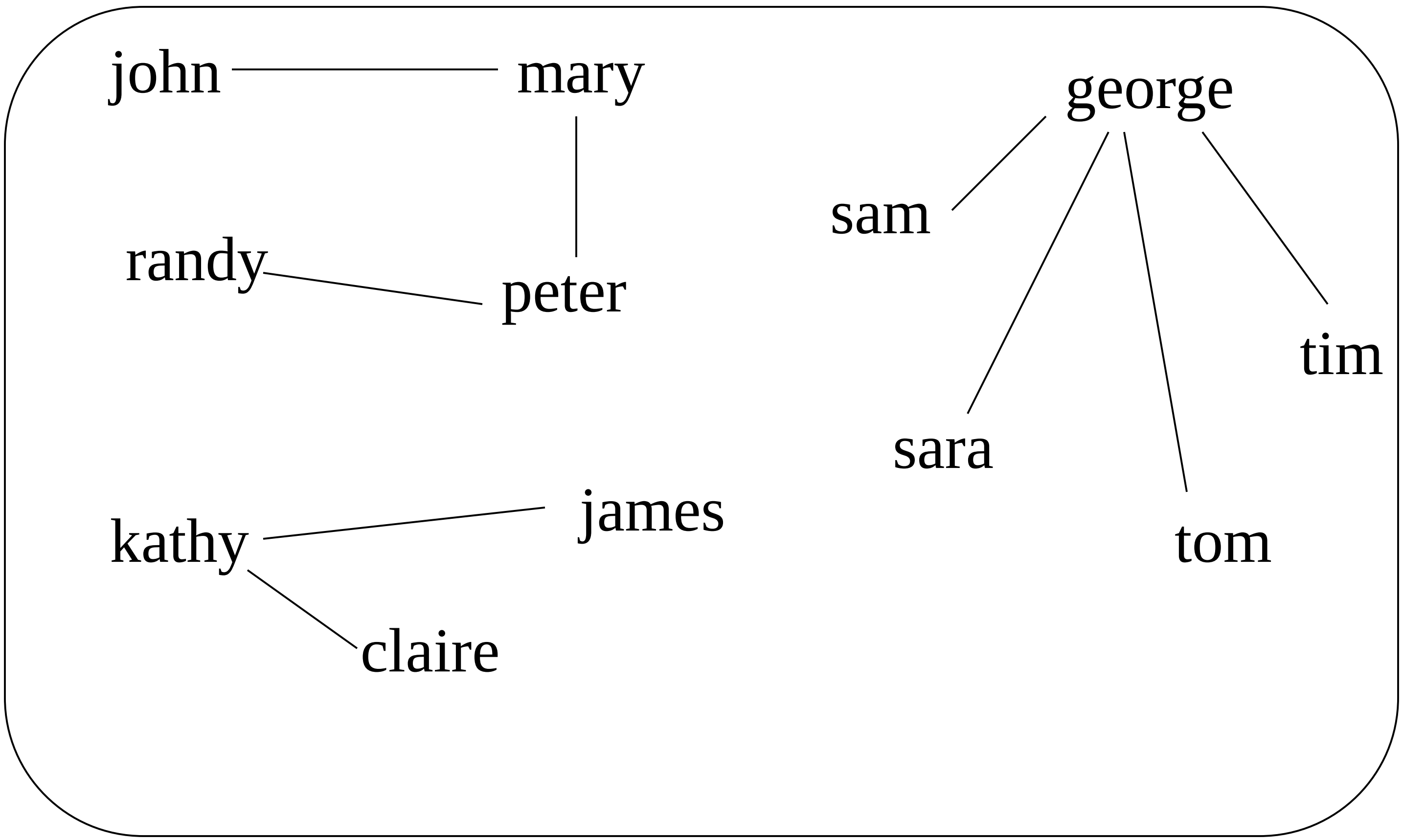
- In other words, if we define an ER R over a set S, R partitions S into a number of disjoint sets that are ECs.

Example


S
john
mary
george
sam
randy
peter
tim
sara
james
kathy
tom
claire

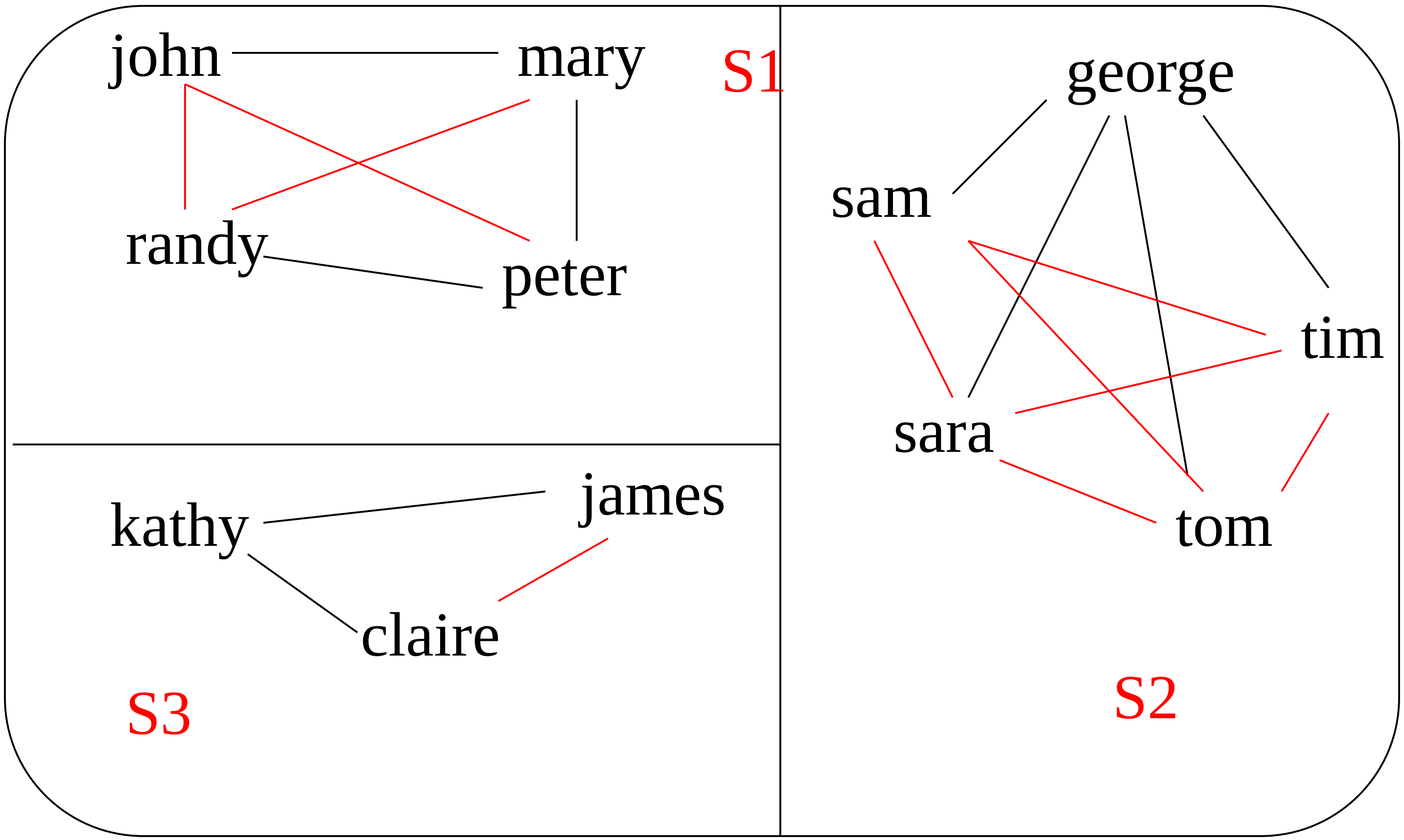
Example

R: sibling of; suppose we know these are siblings



Example

Since R is an ER, additional relations (red lines) are also true, and R divides S into 3 ECs.



ECs

- Now, if we maintain information about all ECs in a database of objects with a relation R defined over these objects, we can answer the question aRb? by checking if a and b belong to the same EC.
- Similarly, we can assert aRb in the database by putting a and b in the same EC.

ECs

- This means that we need to be able to efficiently implement two operations:
  - determine if aRb is true/false – how?
    - true if EC(a)=EC(b); false otherwise
  - assert aRb in the database – how?
    - if EC(a) != EC(b) then merge EC(a) and EC(b)
  - So we need two operations
    - Find(element) returns its EC
    - Union(EC1, EC2) merges the two ECs

Implementation

- Data elements in the database are numbered 1...n
- ECs are named 1...n also (don’t get confused with element numbering)
- Initially we assume that no element is related to any other, i.e. each is in an EC by itself
- Two approaches to implementing such a database so that Find and Union operations can be done very efficiently

Implementation Approach 1

- Use an array A in which A[i] contains the "name" of the EC to which element i belongs.

1	2	3	4	5	6	7	8	9	10
1	**2**	**3**	**4**	**5**	**5**	**4**	**3**	**2**	**1**

- How will you implement Find(i) & Union(i,j)? With what complexities?

Implementation Approach 1

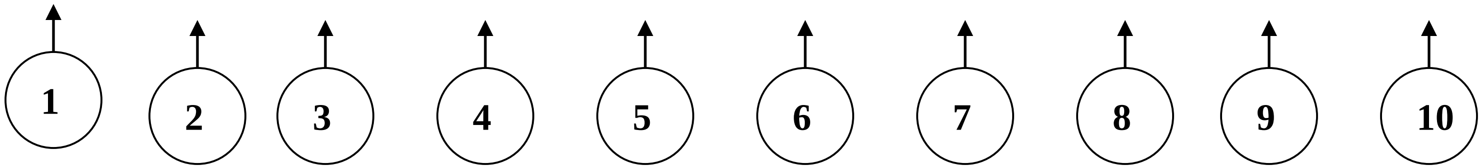
- Find(i) (where i is an element)
  - Returns A[i]
  - Find is Θ(1)
- Union(i,j) (where i and j are names of ECs)
  - How can this be implemented?
  - Go through the array and change all i's to j's or vice versa
  - Union is Θ(n)
  - Result of Union(1,2) on the previous array:

1	2	3	4	5	6	7	8	9	10
2	2	3	4	5	5	4	3	2	2

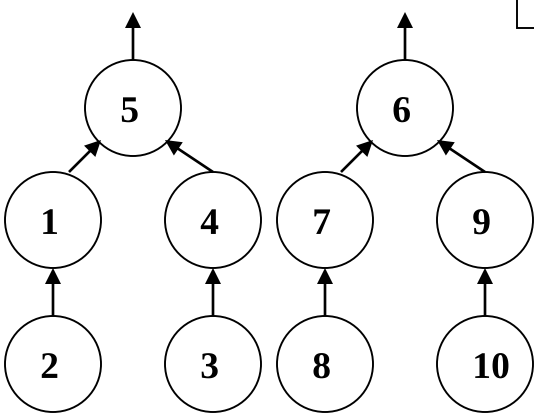
Implementation Approach 2

- Keep all elements in the same EC in a tree.
- The root provides the name of the EC.
- This generates a forest of trees, which is implemented with an array P.
- P[i] = parent of i if i is not the root; else P[i]=0

Implementation Approach 2



1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0

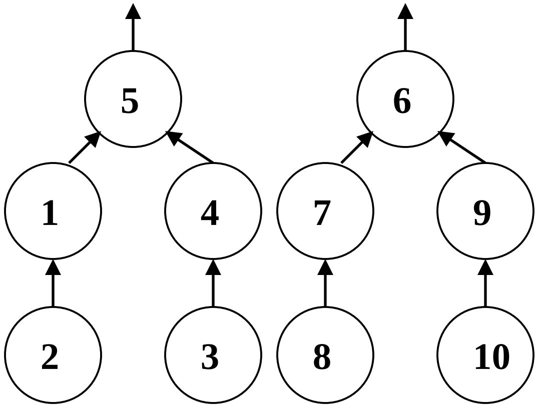


- How can Find and Union be implemented?

1	2	3	4	5	6	7	8	9	10
5	1	4	5	0	0	6	7	6	9

Implementation Approach 2

- Find(element): return the root of the tree containing element.
- Find(i) needs exactly as many steps as the depth of element i in its tree – why?
- Find is O(n) – why?

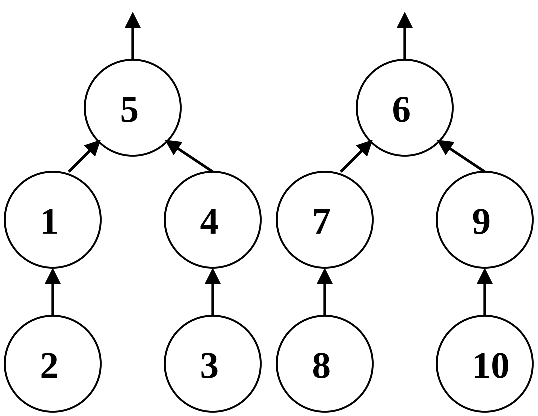


1	2	3	4	5	6	7	8	9	10
5	1	4	5	0	0	6	7	6	9

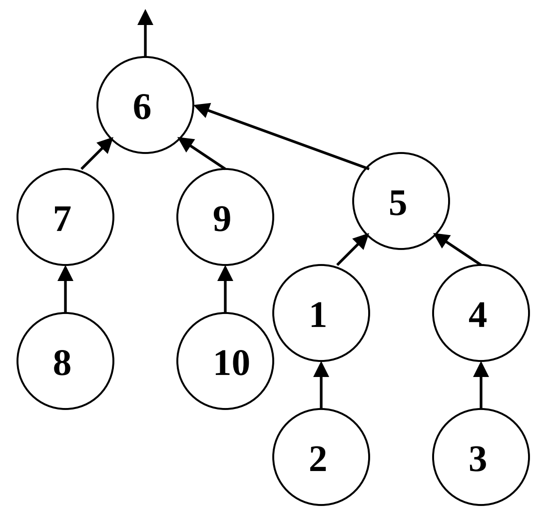
Implementation Approach 2

- Union(i,j) where i & j are roots of trees (names of ECs): if i != j then set P[i]=j OR P[j]=i (be consistent)
- Union's complexity is Θ(1)

1	2	3	4	5	6	7	8	9	10
5	**1**	**4**	**5**	**0**	**0**	**6**	**7**	**6**	**9**



Union(6,5)



1	2	3	4	5	6	7	8	9	10
5	**1**	**4**	**5**	**6**	**0**	**6**	**7**	**6**	**9**

Which approach is better?

- Approach 1
  - Find is $\Theta(1)$ and Union is $\Theta(n)$
- Approach 2
  - Find is $O(n)$ and Union is $\Theta(1)$
  - $O(n)$ better than $\Theta(n)$ in this case: why?
  - So this is the preferred approach
- In either, a sequence of m operations on a database of size n will require $O(mn)$ time

Disjoint Set Operations

- Now, let us take a closer look at the fundamental operations Find and Union under the implementation approach 2 where a disjoint set is implemented as a forest.
- Note that if there are n data elements, the id’s of the elements, names of the disjoint sets, and the indexes of the array implementation range from 1 to n.

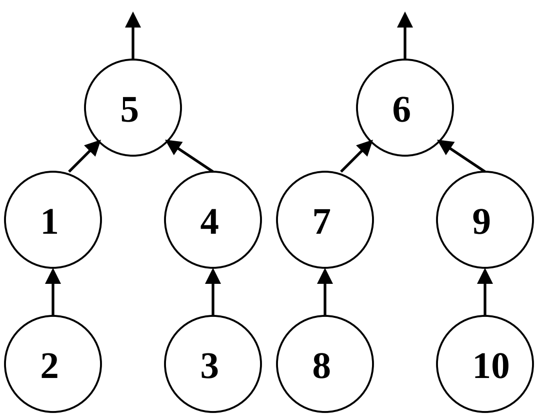
Disjoint Set Operations

- Union
  - Arbitrary
  - By-Size
  - By-Height
- Find
  - without/with Path Compression

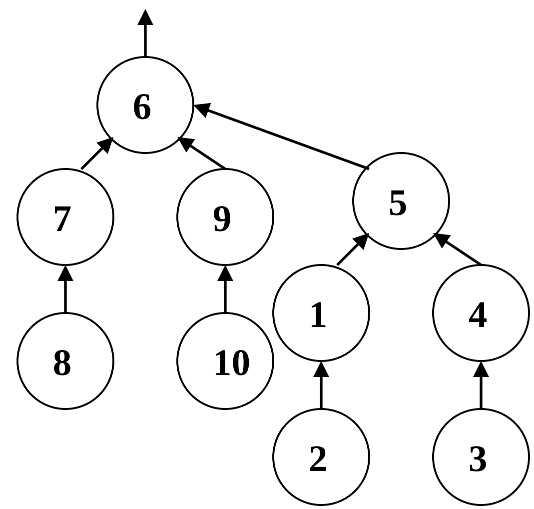
Arbitrary Union

- Union(i,j) where i & j are roots of trees (names of ECs): if i != j then set P[i]=j OR P[j]=i (be consistent)
- Union’s complexity is Θ(1) best, worst and average case.

1	2	3	4	5	6	7	8	9	10
5	1	4	5	0	0	6	7	6	9



Union(6,5)

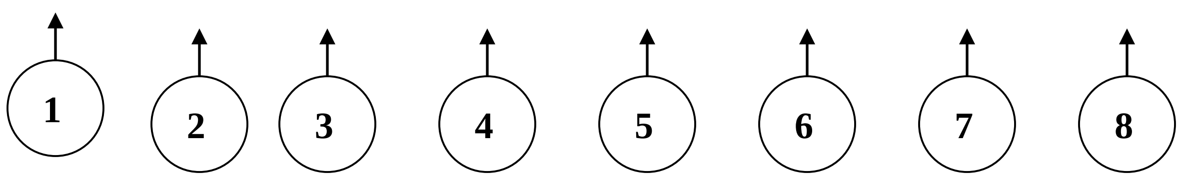


1	2	3	4	5	6	7	8	9	10
5	1	4	5	6	0	6	7	6	9

Arbitrary Union

- With arbitrary union, n-deep trees may be generated.
- Example: Initially,

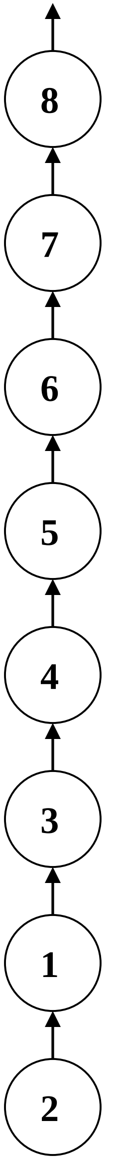
1	2	3	4	5	6	7	8
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**



Example

- Union(Find(1),Find(2));
  Union(Find(3), Find(1));
  Union(Find(4), Find(3));
  Union(Find(5), Find(4));
  Union(Find(6), Find(5));
  Union(Find(7), Find(6));
  Union(Find(8), Find(7))

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



“Smart” Union

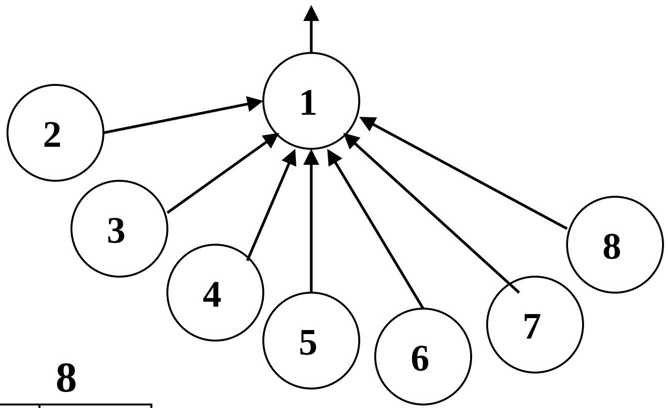
- Use some “smarts” in deciding how to merge two trees!
- Two ways: Union-by-Size and Union-by-Height

Union-by-Size

- Make the smaller tree a subtree of the root of the larger one
  - Implementation: at array cells corresponding to roots, use the negative of tree-size instead of 0
  - Add the two tree sizes when unioning

Union-by-Size

- Redo the previous example:
- Union(Find(1),Find(2));
  Union(Find(3), Find(1));
  Union(Find(4), Find(3));
  Union(Find(5), Find(4));
  Union(Find(6), Find(5));
  Union(Find(7), Find(6));
  Union(Find(8), Find(7))



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

Union-by-Size

- If you start with n nodes in the set originally, any tree resulting from Unions-by-size will have a depth of at most logn.
- Why?

Informal Proof

- We start with each data in its own 1-node (0-depth) tree
- Note that a node's depth will change as a result of a subsequent U-b-s operation only if it is in the smaller of the two trees being merged (unless both are equal size)
- This means that every time the depth of a node increases by 1, the size of its tree at least doubles
- Suppose after many U-b-s operations a data item is k-deep in its tree
- This means that the tree must have at least 2^k nodes

Informal Proof

- So any tree of depth k must have at least 2^k nodes
- Since there are only n data items, $2^k \leq n$
- So $k \leq \log n$
- I.e. the depth of any node $\leq \log n$
- Therefore the depth of any tree $\leq \log n$

Union-by-Size

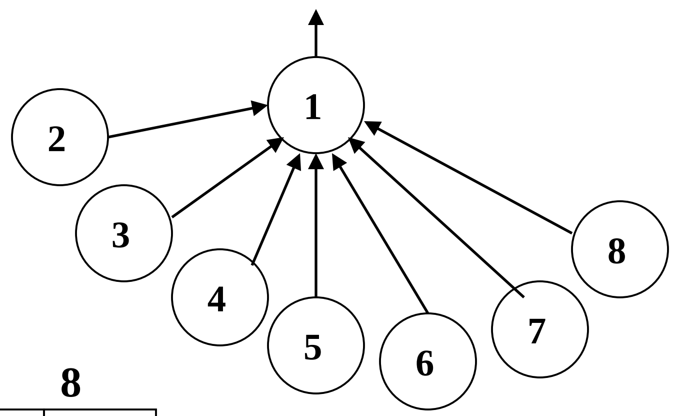
- Any tree resulting from Unions-by-size will have a depth $\leq$ logn
- This implies that Find is O(logn)
  - (an improvement over O(n) with arbitrary Union!)
- Union is still constant time - $\Theta(1)$
- A sequence of m operations on a data set of size n has complexity O(mlogn).

Union-by-Height

- Make the shallower tree a subtree of the root of the deeper one
    - Implementation: at array cells corresponding to roots, use the negative of tree-depth instead of 0
    - Increase the height by 1 when unioning two trees of the same height

Union-by-Height

- Redo the previous example:
- Union(Find(1),Find(2));
  Union(Find(3), Find(1));
  Union(Find(4), Find(3));
  Union(Find(5), Find(4));
  Union(Find(6), Find(5));
  Union(Find(7), Find(6));
  Union(Find(8), Find(7))



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

Union-by-Height

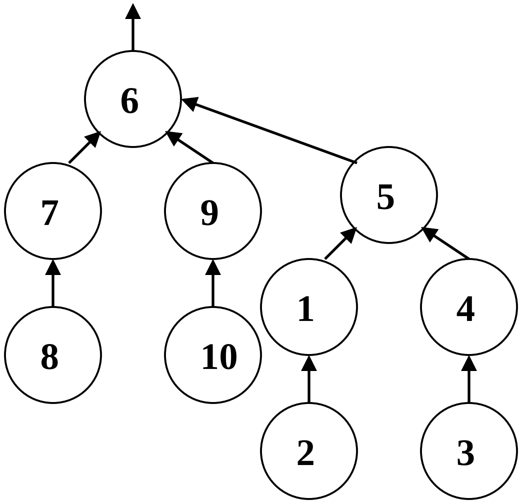
- If you start with n nodes in the set originally, any tree resulting from Unions-by-height will have a depth of at most logn.
- Thinking Assignment: Prove this by appropriately modifying the proof for Union-by-Size

Path Compression

- With smart unions we can bound the depth of any tree to logn.
- But Find operations on leaf nodes will still take logn time.
- We can improve this situation by modifying the Find operation.
- During Find(x), change the parent of every node between x and the root to be the root of the tree.
- An example of **self-adjusting data structure**

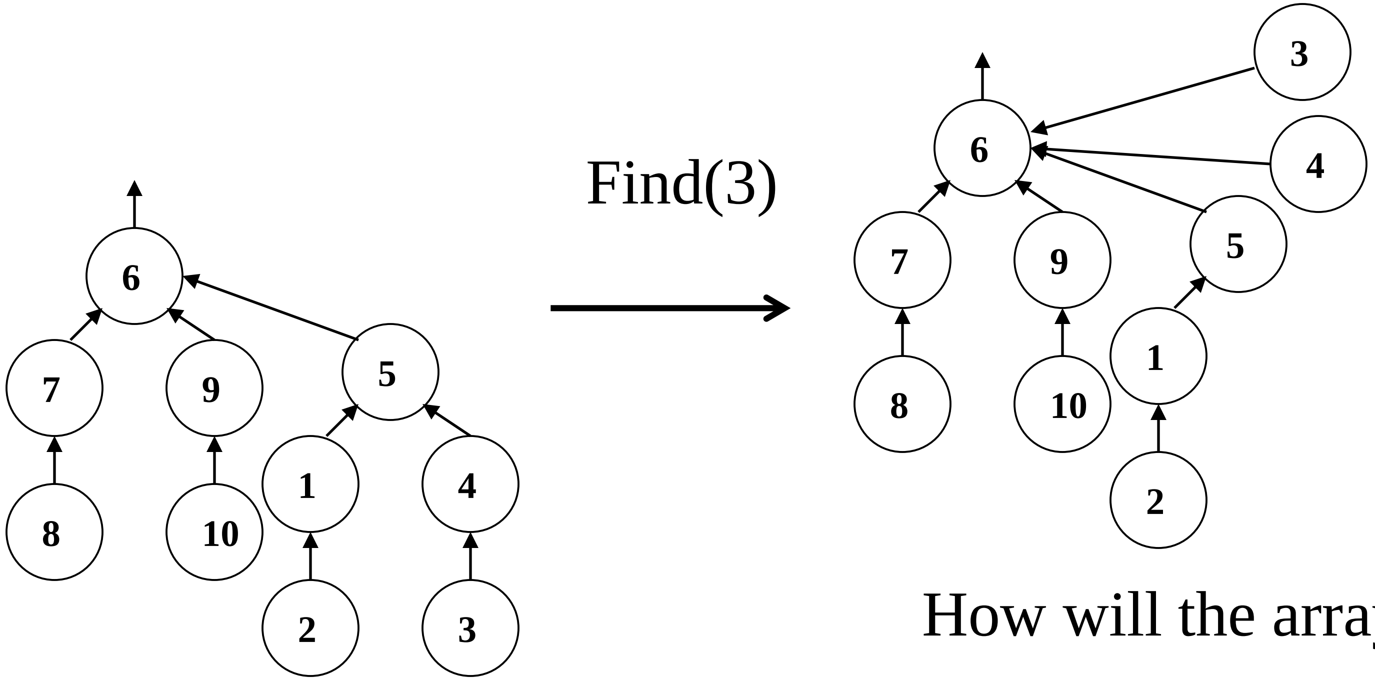
Find without Path Compression

Find(3)



1	2	3	4	5	6	7	8	9	10
5	1	4	5	6	0	6	7	6	9

Find with Path Compression



How will the array change?

1	2	3	4	5	6	7	8	9	10
5	1	4	5	6	0	6	7	6	9

Path Compression

- Why?
  - First, it gradually decreases the depth of the tree
  - Second, items once accessed are generally more likely to be accessed again, and this makes such accesses single-step

Path Compression

- Path compression works well with Union-by-Size
- But how about Union-by-Height?

Path Compression

- As PC changes tree heights, for Union-by-Height to work correctly, tree heights need to be recomputed after every Find
- This is computationally expensive and cancels out the efficiency achieved by PC

Path Compression

- So the most efficient operations can be achieved by Find-with-PC and Union-by-Size

Summary

- Disjoint Set Implementations
  - Approaches 1 & 2: 2 commonly used
- Unions
  - arbitrary, by-size, by-height
- Find
  - with or without Path Compression
- The best combination
  - Approach 2, Union-by-Size, Find-with-PC