Next Computational Problem:

Efficiently Asserting and Retrieving Relations Between Data Items

Debswapna Bhattacharya, PhD
Assistant Professor
Computer Science and Software Engineering
Auburn University

Problem specifications

Inputs:

a pair of data items a and b belonging to a set S
a **binary equivalence** relation R

Output: True or False

Correctness criterion: True if and only if R holds between a and b, False otherwise

Inputs:

a pair of data items a and b belonging to a set S
a binary equivalence relation R

Output: None

Correctness criterion: If R did not hold between a and b before, then R must hold between a and b after.

What is a binary equivalence relation?

Binary Relation

- R: a binary 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 Problems

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

- If aRb is false, assert aRb so it is true

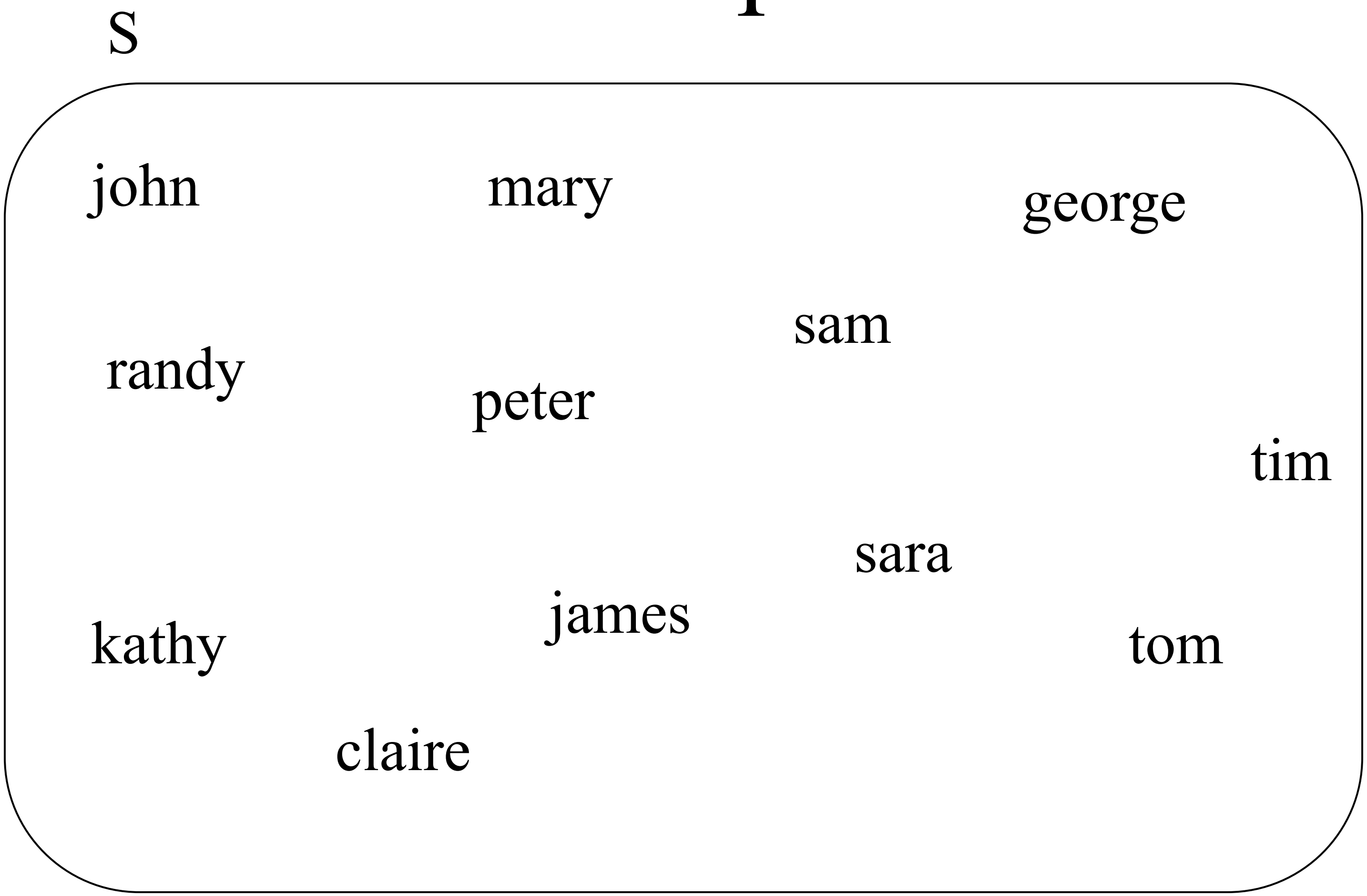
What Data Structure should we use to represent relationships between data items so equivalence problems can be efficiently solved?

Equivalence Class (EC)

- The EC of an element a∈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 (non-overlapping) sets** that are ECs.

Example



S

john

mary

george

sam

randy

peter

tim

sara

james

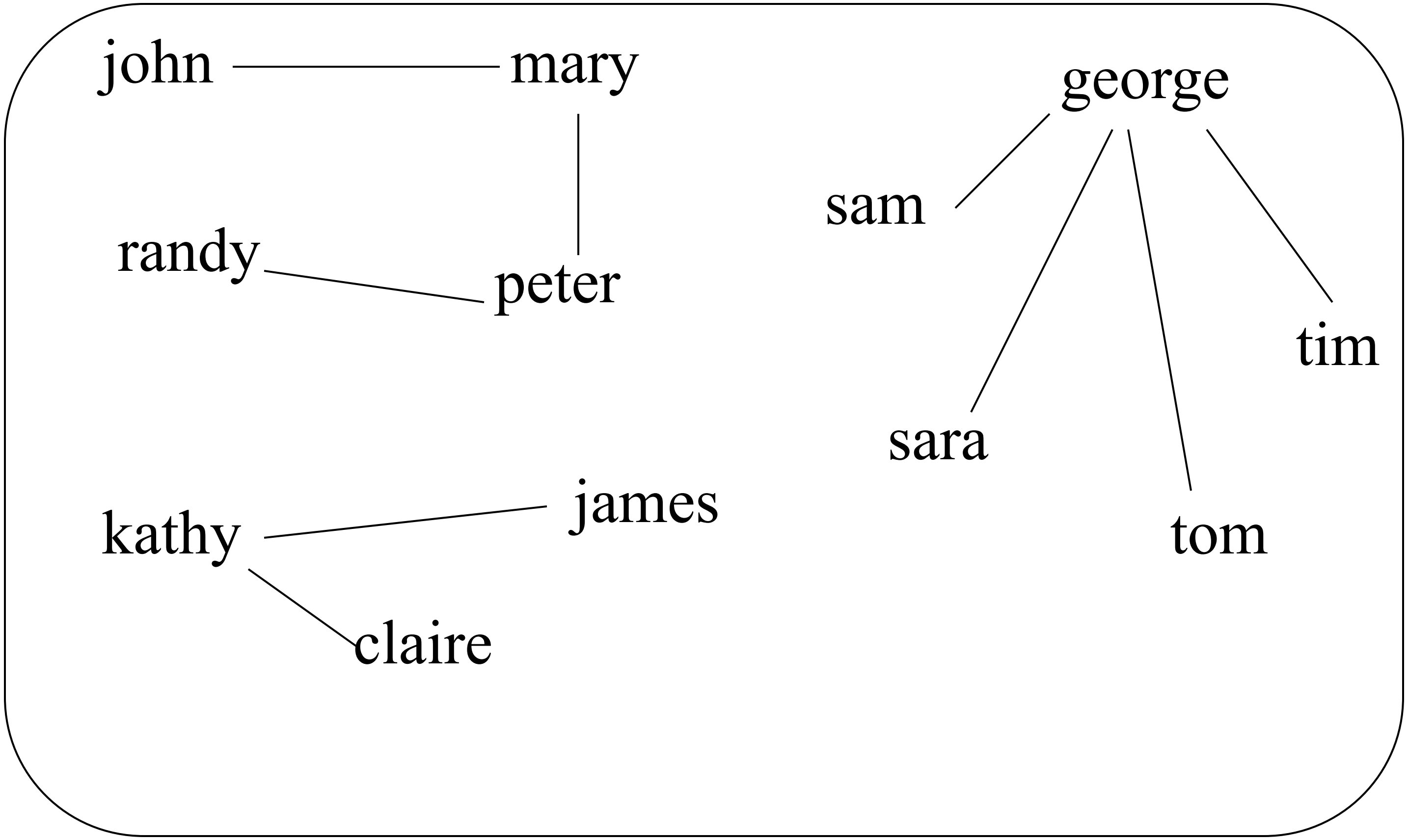
kathy

tom

claire

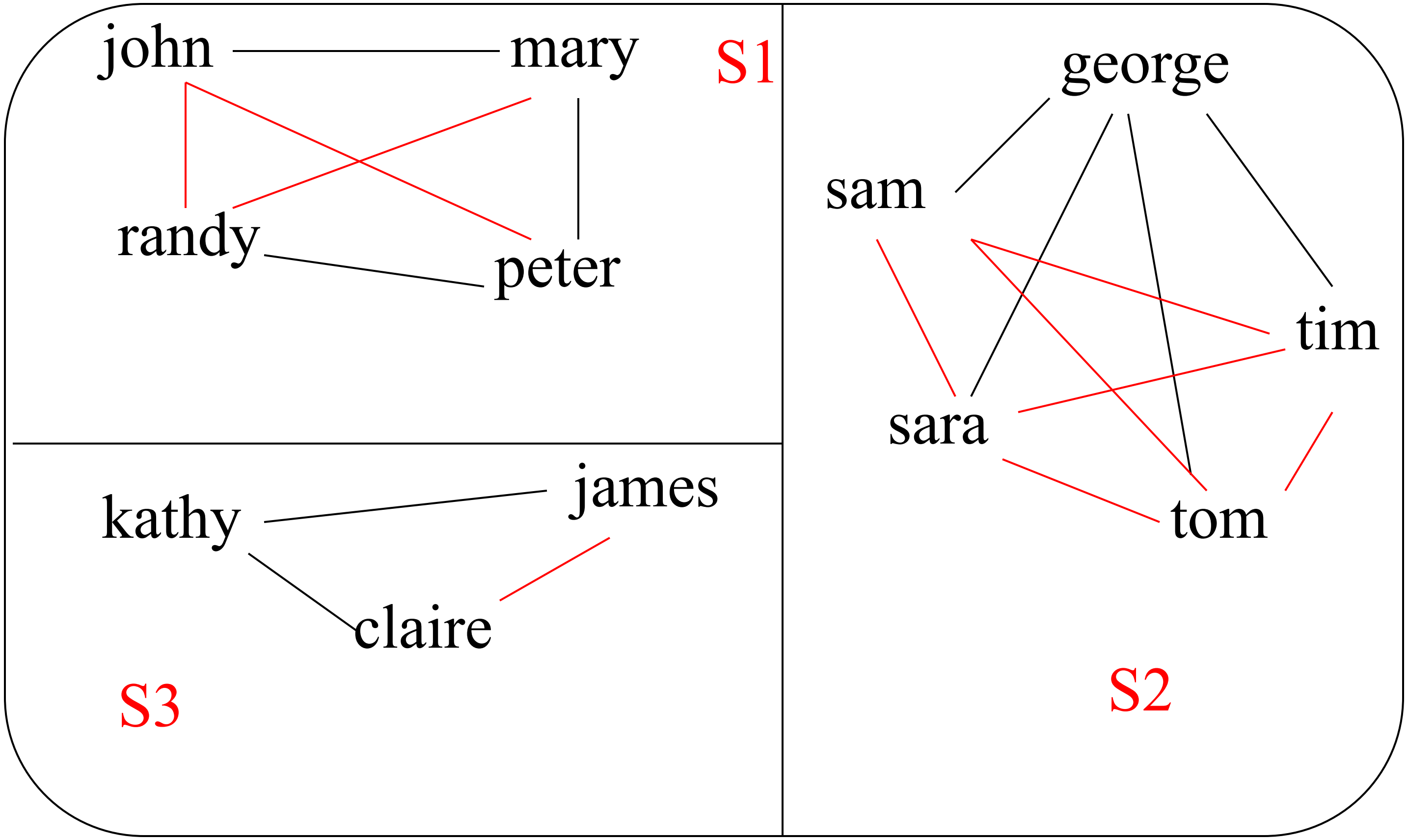
Example

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.



Insight!

- So, if we maintain information about all ECs in a set 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.
  - i.e., we have transformed the relation checking problem into a set membership problem!

- Similarly, we can assert aRb in the database by putting a and b in the same EC.
  - i.e., we have transformed the relation asserting problem into a set merging (union) problem!

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-Set(element) returns its disjoint set or EC
    - Union(set1, set2) merges the two sets

Implementation

- Data elements in the set are numbered 1…n (actual data may be anything)
- Disjoint sets are numbered 1…n also (using numbers of *representative* elements)
- Initially we assume that no element is related to any other, i.e. each is in a disjoint set by itself
- Two approaches to implementing such a database so that Find-Set and Union operations can be done very efficiently

Implementation Approach 1: Arrays

- Use an array A in which A[i] contains the “name” of the set 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-Set(i) & Union(i,j)? With what complexities?

Implementation Approach 1

- Find-Set(i) (where i is an element)
  - Returns A[i]
  - Find is Θ(1)
- Union(i,j) (where i and j are “names” of separate disjoint sets)
  - How can this be implemented?
  - Iterate 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 following array:

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

↓

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

Implementation Approach 2: Trees

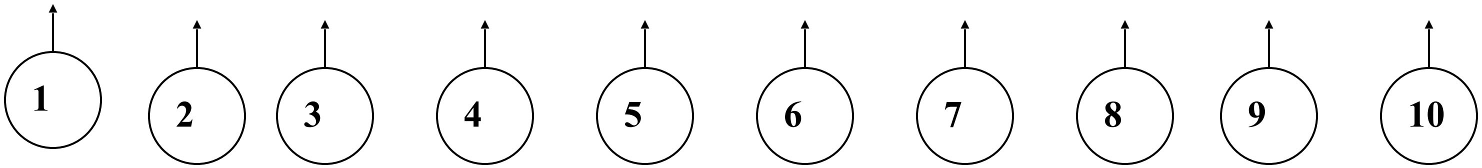
- Keep all elements in the same set in a tree.

- The element in the root provides the name of the set, i.e., it is the *representative* element.

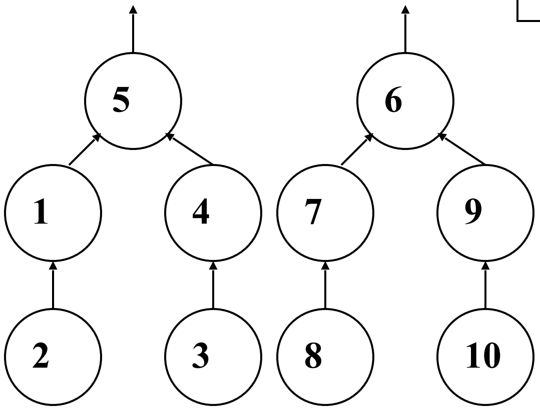
- This generates a forest of trees, which is implemented with an array P.

- 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

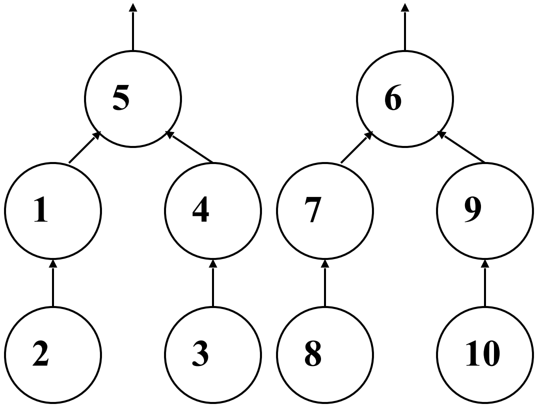


- How can Find-Set and Union be implemented?

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

Implementation Approach 2

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

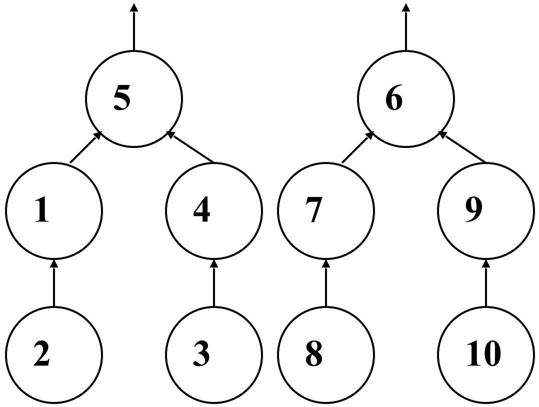


1	2	3	4	5	6	7	8	9	10
5	1	4	5	0	0	6	7	6	9

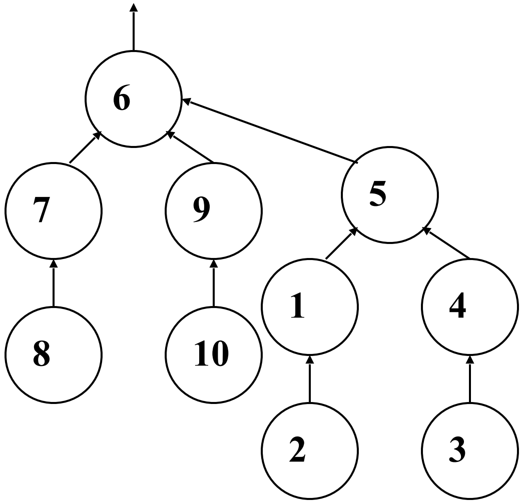
Implementation Approach 2

- Union(i,j) where i & j are roots of trees (therefore, names of the corresponding disjoint sets): 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

Make-Set (element)

- Builds a disjoint set containing a single data element
- Θ(1) under either implementation

Which approach is better?

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

Reading Assignment
DisjointSets.pdf on Canvas.

Disjoint Set Operations

- Now, let us take a closer look at the fundamental operations Find-Set 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.

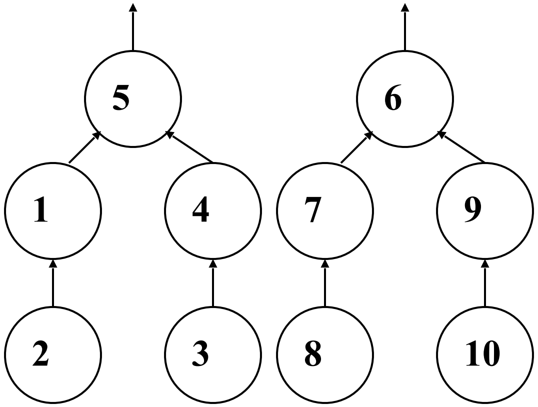
Disjoint Set Operations

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

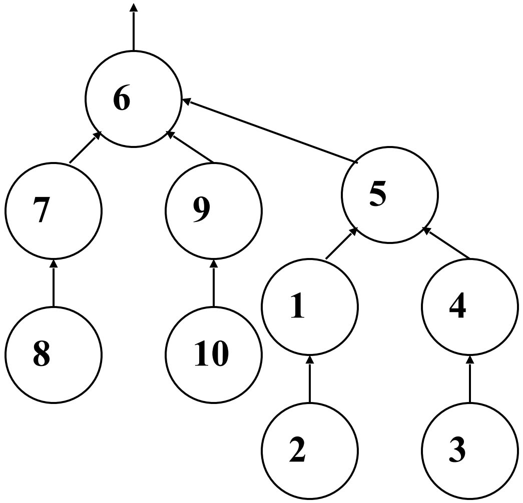
Arbitrary Union

- Union(i,j) where i & j are roots of trees (therefore, names of the corresponding disjoint sets): 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)

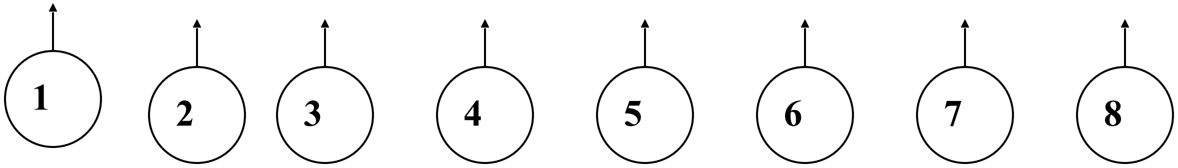


1	2	3	4	5	6	7	8	9	10
5	1	4	5	6	0	6	7	6	9

Arbitrary Union

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

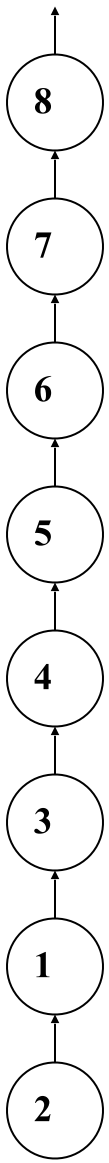
1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0



Example

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

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



“Smart” Union

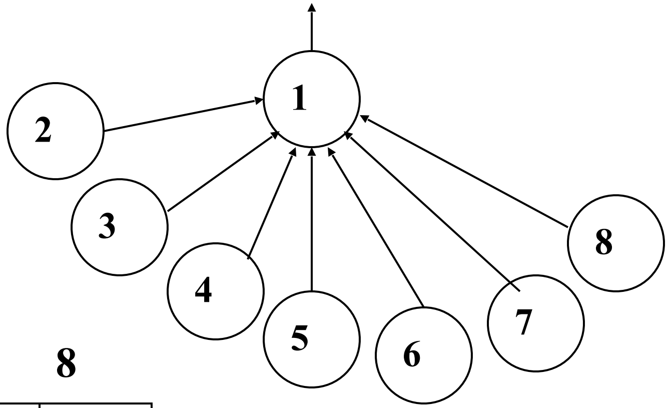
- Be “smart” in deciding how to merge two trees!
- Two ways: Union-by-Size and Union-by-Height

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 doing union

Union-by-Size

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

Union-by-Size

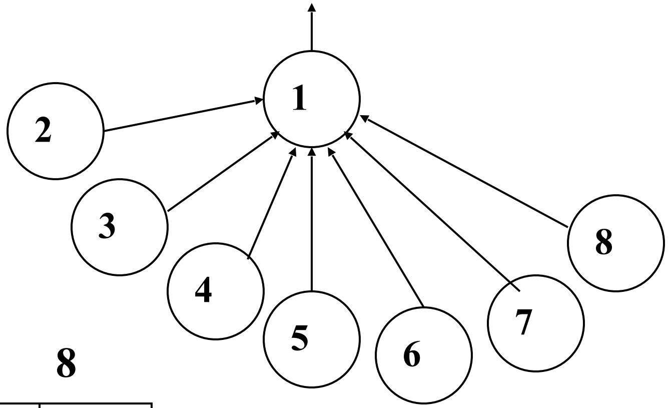
- Any tree resulting from Unions-by-size will have a depth ≤ logn
- This implies that Find-Set is O(logn)
  - (an improvement over O(n) with arbitrary Union!)
- A sequence of m operations on a data set of size n has complexity O(mlogn).

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 doing union of two trees of the same height

Union-by-Height

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



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

Union-by-Height

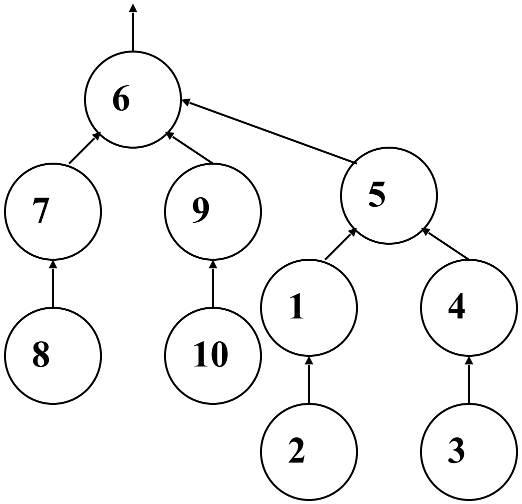
- If you start with n nodes in the set originally, any tree resulting from Unions-by-height will have a depth of at most logn.

Path Compression

- With smart unions we can bound the depth of any tree to logn.
- But Find-Set operations on leaf nodes will still take logn time.
- We can improve this situation by modifying the Find-Set operation.
- During Find-Set (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**

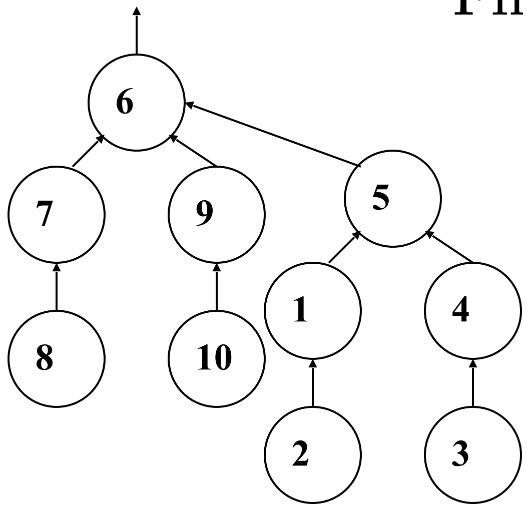
Find-Set without Path Compression

Find-Set (3)

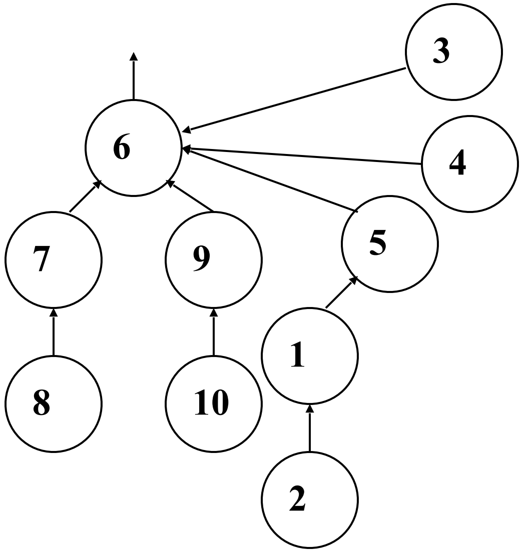


1	2	3	4	5	6	7	8	9	10
5	1	4	5	6	0	6	7	6	9

Find-Set with Path Compression



Find-Set (3)



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

Path Compression

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

Summary

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

Chapter 21
Reading Assignments

21.1

21.2

(Omit Theorem 21.1 and its proof)

21.3

(Omit "Effect of the heuristics on the running time")

Omit 21.4

Thinking Assignments

1. Using the array implementation of disjoint sets as trees, write an algorithm to go from a node in the tree to <u>any</u> leaf that can be reached from it.
2. Using the array implementation of disjoint sets as trees, write an algorithm to efficiently delete a node in the tree, which may be a root, a leaf or an interior node.
3. Modify the arbitrary union algorithm discussed in the class to obtain an algorithm for union-by-size. Assume the array implementation of disjoint sets as trees.
4. Modify the arbitrary union algorithm discussed in the last class to obtain an algorithm for union-by-height. Assume the array implementation of disjoint sets as trees.
5. Modify the find-set algorithm discussed in the last class to obtain an algorithm for find-set-with-path-compression. Hint: use a stack. Assume the array implementation of disjoint sets as trees.

Section 21.1: Operations

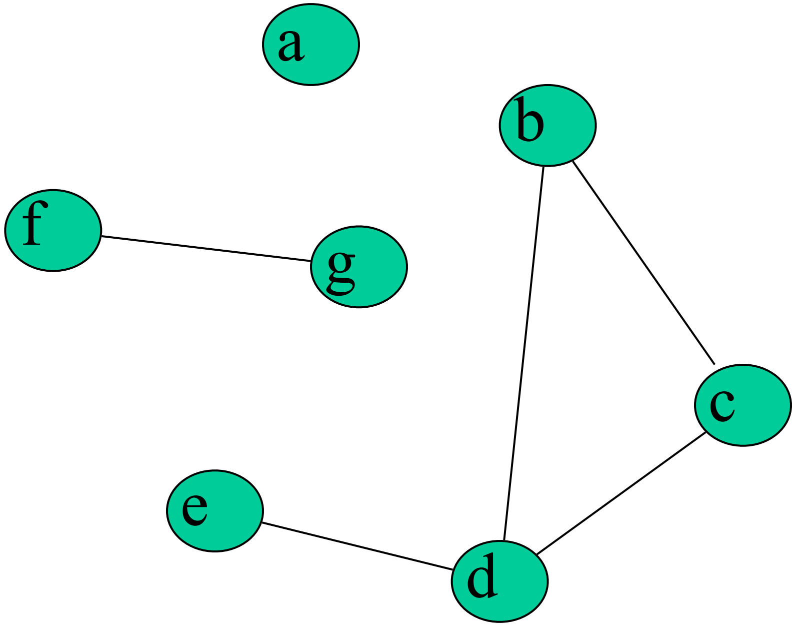
- Make-Set(x: element)
  - Make a set with data element x
- Union(x,y: elements)
  - Merge (union) the disjoint set containing x with the set containing y
- Find-Set(x: element)
  - Find the disjoint set containing x
- Initialization: n Make-Set operations
  - Create n disjoint sets containing one data element each

A more general view

Though we introduced the Disjoint Set data structure as an efficient means to assert and query equivalence relations that hold between pairs from a set of data items, this idea is applicable whenever a computational problem can be solved by maintaining a set of data objects separated into non-overlapping subsets, and querying the membership of a data element or change that membership.

Section 21.1: An Application

Connected Components of an Undirected Graph



21.1: An Application

```
Connected-Components(G)   {Preprocessing}
for each node v in G.V
    make-set(v)
for each edge (u,v) in G.Edges
    if find-set(u)≠find-set(v)
        union(u,v)
```

21.1: An Application

```
Same-Component(u,v)   {Querying}
if find-set(u)==find-set(v)
    return TRUE
else return FALSE
```

Thinking Assignments

1. What is the relation being maintained in the Connected Components application?
2. It has to be an equivalence relation? Explain why the relation you stated is an ER
3. Do problems 21.1-1, 21.1-2 and 21.1-3

Section 21.2: A Third Implementation of Disjoint Sets

Linked lists with head, tail and return pointers:

Make-Set: $\Theta(1)$

Find-Set: $O(n)$

Union: $O(n)$

The text assumes a linked list representation in with each list node is directly accessible by the name of the data object in that node. In this case:

Make-Set: $\Theta(1)$

Find-Set: $\Theta(1)$

Union: $O(n)$

The worst case for Union operation is when you are appending a linked list of size (n-1) to the end of a linked list with one element!

This situation can be somewhat improved by using the idea behind Union-by-size: keep the linked list size in its header and append the smaller list to the end of the larger list.

This will not improve the complexity order which is still O(n) – e.g., when unioning two sets of size n/2 to get a list of size n – but will speed up unions in practice.

This is called **weighted-union**.

Weighted Union

1. Store size of set at the header of the list
2. When merging two sets, update the smaller linked list – similar to Union-by-Size
3. A variation of the proof we used to show that Union-by-Size caps tree height to at most logn can be used to show that even though weighted union does not change the complexity order of any single union operation, a sequence of (n-1) unions on n data elements will cost O(nlogn) and therefore the average cost of a union is O(logn).

Optional Reading Assignment: if interested in how/why, read Theorem 21.1 and its proof, p.566-567.

Thinking Assignments

Do problems : 21.2-1, 21.2-2, 21.2-5, 21.2-6

Section 21.3: Same as Implementation #2

This section introduces a variation on Union-by-Height discussed earlier, calling it Union-by-Rank where Rank is not the actual height but an upper bound on the height of the tree.

So we can use it along with Find-with-Path-Compression, whereas path compression is not possible with Union-by-height.

```
Make-Set(x)
P[x]=—1

Union(x,y)      //by Rank
a=Find-Set(x)
b=Find-Set(y)
if |P[a]|>|P[b]|
    P[b]=a
else
    if |P[a]|==|P[b]| then P[b]=P[b] — 1
    P[a]=b

Find-Set(x)     //with Path Compression
if P[x] is not negative then
    P[x]=Find-Set[P[x]]
    return P[x]
else return x
```

These are algorithms from text p.571 modified to work for the array based approach (implementation #2) discussed in the last class.

Thinking Assignment: Read, simulate and fully understand these three algorithms.

Thinking Assignments

1. Draw the recursion tree of the Find-Set algorithm on the previous slide working on a specific example.
2. Can you write an iterative version of this find with path compression algorithm? (Hint: you need a stack).
3. Read the algorithms on text p. 571, understand them, and think about (and understand!) the differences between them and the algorithms on the previous slide.

Thinking Assignments

Do problems : 21.3-1, 21.3-2, 21.3-4

Comparison of the three Disjoint Set implementations

	#1: Arrays	**#2: Trees**	**#3: Lists**
Make-Set	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Find-Set	$\Theta(1)$	O(logn) (with PC)	$\Theta(1)$ (each linked list node is directly accessible by name)
Union	$\Theta(n)$	$\Theta(1)$ (by size or by rank)	O(logn) (weighted union)

If an application is expected to execute significantly more finds than unions, the list implementation is best.

If the opposite is true, the tree implementation is better.

In applications where the data objects need to be directly accessible (the array implementation of the forest stores only data object IDs, not the objects themselves), the list is the preferred implementation since pointers can be used to link complex data objects with linked list nodes, or the data objects themselves may be made into linked list nodes with additional pointer attributes added.