Lecture 18: The Dynamic Equivalence Problem (a.k.a. Disjoint Sets, Union/Find etc.)

- The Plot:
  - A new problem: Dynamic Equivalence
  - The setting:
    - Motivation and Definitions
  - The players:
    - Union and Find, two ADT operations
    - Up-tree data structure
  - Suspense-filled cliffhanger (to be continued…next time)

- Covered in Chapter 8 of the textbook

Motivation

- Consider the relation "=" between integers
  1. For any integer A, A = A
  2. For integers A and B, A = B means that B = A
  3. For integers A, B, and C, A = B and B = C means that A = C
- Consider cities connected by two-way roads
  1. A is trivially connected to itself
  2. A is connected to B means B is connected to A
  3. If A is connected to B and B is connected to C, then A is connected to C
- Consider electrical connections between components on a computer chip
  - 1, 2, and 3 are again satisfied

Equivalence Relations

- An equivalence relation R obeys three properties:
  1. reflexive: for any x, xRx is true
  2. symmetric: for any x and y, xRy implies yRx
  3. transitive: for any x, y, and z, xRy and yRz implies xRz
- Preceding relations are all examples of *equivalence relations*
- What are not equivalence relations?

Equivalence Relations

- An equivalence relation R obeys three properties:
  1. reflexive: for any x, xRx is true
  2. symmetric: for any x and y, xRy implies yRx
  3. transitive: for any x, y, and z, xRy and yRz implies xRz
- Preceding relations are all examples of *equivalence relations*
- What are not equivalence relations?
  - What about "<" on integers? (1 and 2 are violated)
  - What about "≤" on integers? (2 is violated)
  - What about "is having an affair with" in a soap opera?
    - Victor i.h.a.a.w. Ashley i.h.a.a.w. Brad does not imply Victor i.h.a.a.w. Brad (i.h.a.a.w. is not transitive)

Equivalence Classes and Disjoint Sets

- Any equivalence relation R divides all the elements into disjoint sets of "equivalent" items
- Let ~ be an equivalence relation. Then, if A~B, then A and B are in the same equivalence class.
- Examples:
  - On a computer chip, if ~ denotes "electrically connected," then sets of connected components form equivalence classes
  - On a map, cites that have two-way roads between them form equivalence classes
  - What are the equivalence classes for the relation "Modulo N" applied to all integers?

Equivalence Classes and Disjoint Sets

- Let ~ be an equivalence relation. Then, if A~B, then A and B are in the same equivalence class.
- Examples:
  - The relation "Modulo N" divides all integers in N equivalence classes (for the remainders 0, 1, …, N-1)

    E.g. Under Mod 5:

    0 ~ 5 ~ 10 ~ 15 …

    1 ~ 6 ~ 11 ~ 16 …

    2 ~ 7 ~ 12 ~ …

    3 ~ 8 ~ 13 ~ …

    4 ~ 9 ~ 14 ~ …

    (5 equivalence classes denoting remainders 0 through 4 when divided by 5)

Union and Find: Problem Definition

- Given a set of elements and some equivalence relation ~ between them, we want to figure out the equivalence classes
- Given an element, we want to find the equivalence class it belongs to
  - E.g. Under mod 5, 13 belongs to the equivalence class of 3
  - E.g. For the map example, want to find the equivalence class of Redmond (all the cities it is connected to)
- Given a new element, we want to add it to an equivalence class (union)
  - E.g. Under mod 5, since 18 ~ 13, perform a union of 18 with the equivalence class of 13
  - E.g. For the map example, Woodinville is connected to Redmond, so add Woodinville to equivalence class of Redmond

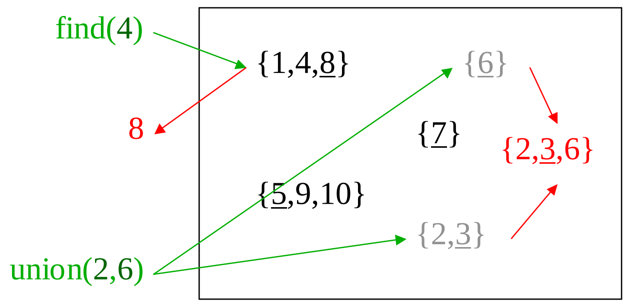
Disjoint Set ADT

- Stores N unique elements
- Two operations:
  - Find: Given an element, return the name of its equivalence class
  - Union: Given the names of two equivalence classes, merge them into one class (which may have a new name or one of the two old names)
- ADT divides elements into E equivalence classes, $1 \leq E \leq N$
  - Names of classes are arbitrary
  - E.g. 1 through N, as long as Find returns the same name for 2 elements in the same equivalence class

Disjoint Set ADT Properties

- Disjoint set equivalence property: every element of a DS ADT belongs to exactly one set (its equivalence class)
- *Dynamic* equivalence property: the set of an element can change after execution of a union

Example:
Initial Classes =
{1,4,8}, {2,3},
{6}, {7},
{5,9,10}
Name of equiv. class underlined



Formal Definition (for Math lovers' eyes only)

- Given a set $U = \{a_1, a_2, \dots, a_n\}$
- Maintain a *partition* of U, a set of subsets (or equivalence classes) of U denoted by $\{S_1, S_2, \dots, S_k\}$ such that:
  - each pair of subsets S_i and S_j are disjoint: $S_i \cap S_j = \emptyset$
  - together, the subsets cover U: $U = \bigcup_{i=1}^{k} S_i$
  - each subset has a unique name
- Union(a, b) creates a new subset which is the union of a's subset and b's subset
- Find(a) returns the unique name for a's subset

Implementation Ideas and Tradeoffs

- How about an array implementation?
  - N element array A: A[i] holds the class name for element i
  - E.g. if 18 ~ 3, pick 3 as class name and set A[18] = A[3] = 3
  - Running time for Find(i) = ?   (i = some element)
  - Running time for Union(i,j) = ? (i and j are class names)

Implementation Ideas and Tradeoffs

- How about an array implementation?
  - N element array A: A[i] holds the class name for element i
  - E.g. if 18 ~ 3, pick 3 as class name and set A[18] = A[3] = 3
  - Running time for Find(i) = **O(1)** (just return A[i])
  - Running time for Union(i,j) = **O(N)**
    If first N/2 elements have class name 1 and next N/2 have class name 2, Union(1,2) needs to change names of N/2 items
- How about linked lists?
  - One linked list for each equivalence class
  - Class name = head of list
  - Running time for Union(i,j) and Find(i) = ?

Implementation Ideas and Tradeoffs

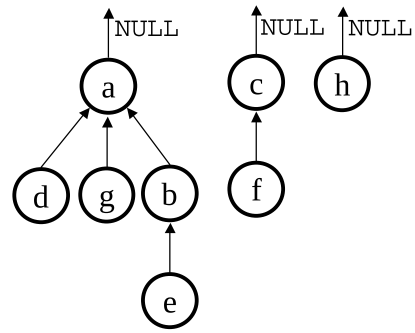
- How about linked lists?
  - One linked list for each class
  - Run time for Union(i,j) = **O(1)** (append one list to the other)
  - Run time for Find(i) = **O(N)** (must scan all lists in worst case)
- Tradeoff between Union-Find – can we do both in O(1) time?
  - N-1 Unions (the maximum possible) and M Finds = $O(N^2 + M)$ for array or O(N + MN) for linked lists implementation
  - Can we do this in O(M + N) time?

Let's find a new Data Structure

- Intuition: Finding the representative member (= class name) for an element is like the *opposite* of searching for a key in a given set
- So, instead of trees with pointers from each node to its children, let's use trees with a pointer from each node to its parent
- Such trees are known as Up-Trees

Up-Tree Data Structure

- Each equivalence class (or discrete set) is an up-tree with its root as its representative member
- All members of a given set are nodes in that set's up-tree
- Hash table maps input data to a node. E.g. input string to integer index

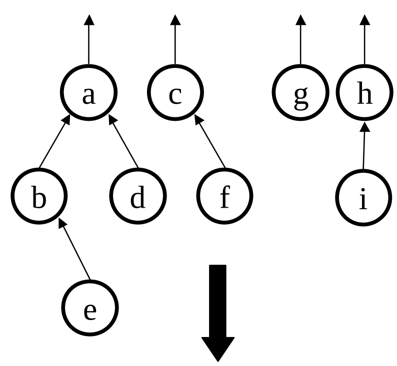


{a,d,g,b,e} {c,f} {h}

Up-trees are **not** necessarily binary!

A neat implementation trick for Up-Trees

- Forest of up-trees can easily be stored in an array (call it "`up`")
- If node names are integers or characters, can use a very simple, perfect hash function: Hash(X) = X
- `up [X]` = parent of X;
  = -1 if root



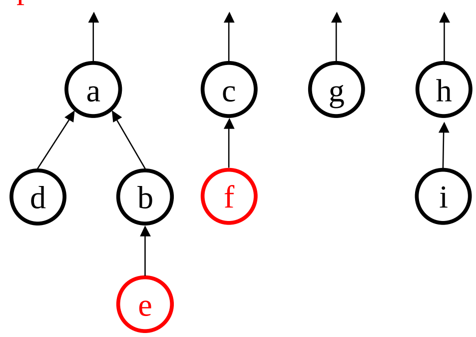
Array `up`:

0 (a)	1 (b)	2 (c)	3 (d)	4 (e)	5 (f)	6 (g)	7 (h)	8 (i)
-1	0	-1	0	1	2	-1	-1	7

Example of Find

Find: Just follow parent pointers to the root!

find(f) = c
find(e) = a



Runtime = ?

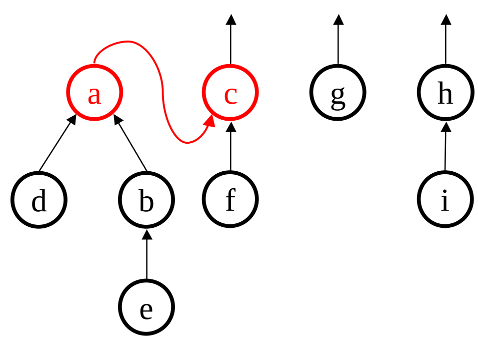
Array up:

0 (a)	1 (b)	2 (c)	3 (d)	4 (e)	5 (f)	6 (g)	7 (h)	8 (i)
-1	0	-1	0	1	2	-1	-1	7

Example of Union

Union: Just hang one root from the other!

union(c,a)



Runtime = ?

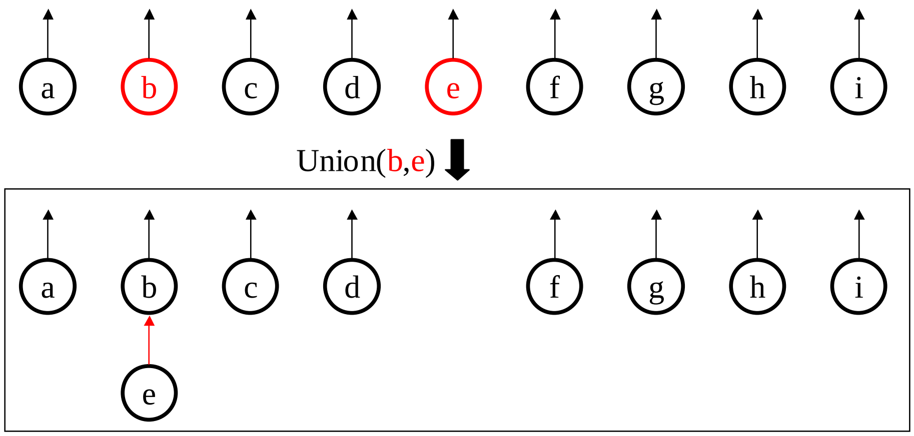
Array up:

0 (a)	1 (b)	2 (c)	3 (d)	4 (e)	5 (f)	6 (g)	7 (h)	8 (i)
2	0	-1	0	1	2	-1	-1	7

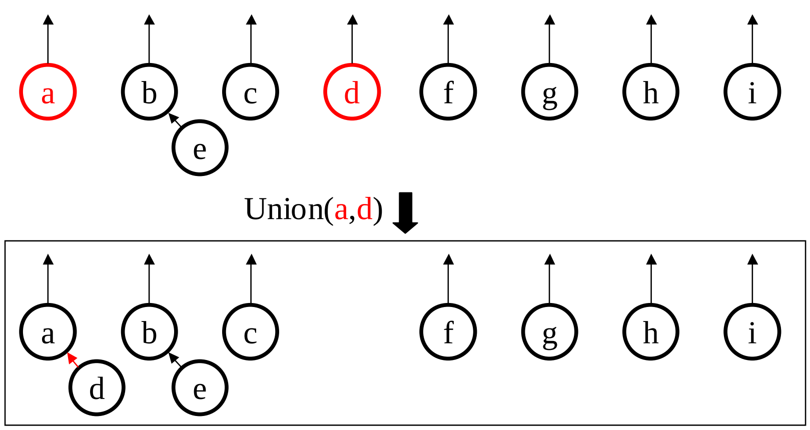
Change a (from -1) to c (= 2)

A more detailed example

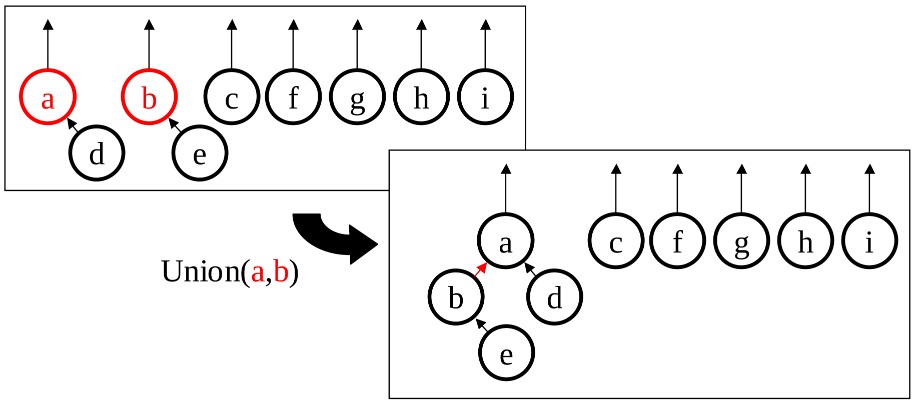
Initial Sets:



A more detailed example…

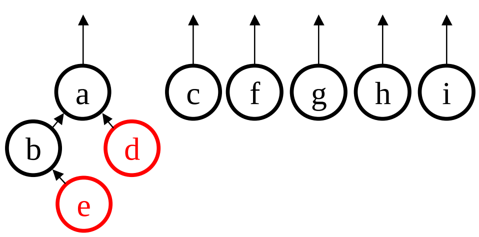


A more detailed example…



A more detailed example…

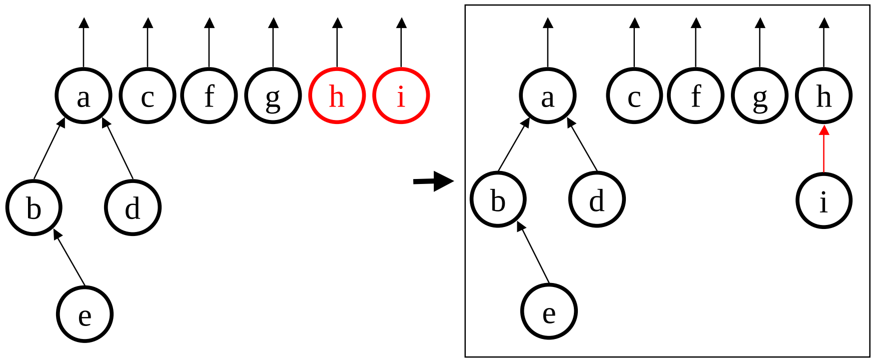
Union(d,e) – But (you say) d and e are not roots!
May be allowed in some implementations – do Find first to get roots
Since Find(d) = Find(e), union already done!



Thought-Provoking Question 1: While we're finding **e**, could we do something to speed up Find(e) next time? (hold that thought!)

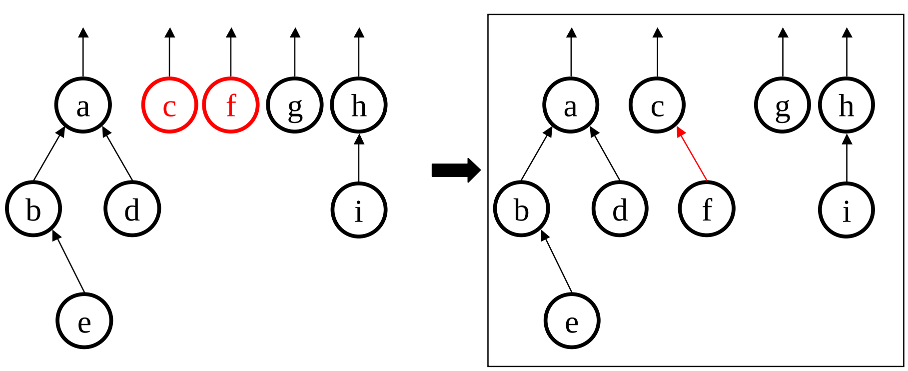
A more detailed example (continued)

Union(h,i)



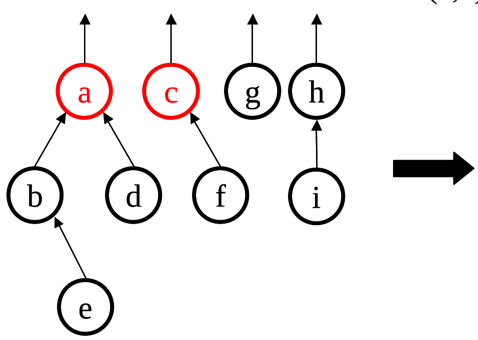
A more detailed example…

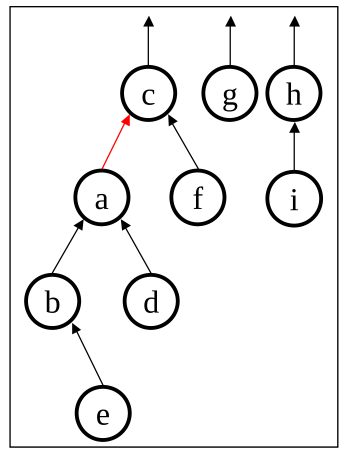
Union(c,f)



A more detailed example

Union(c,a)





TP Q2: Could we do a better job on this union for faster finds in future?

Implementation of Find and Union

```
public int Find(int X)
{ // Assumes X = Hash(X_Element)
  // X_Element could be str/char etc.

  if (up[X] < 0) // Root
   return X; //Return root = set name
  else
   //Find parent
   return Find(up[X]);
}
```

```
public void Union(int X,
   int Y) {
   //Make sure X, Y are
   //roots
  assert(up[X] < 0);
  assert(up[Y] < 0);

  up[Y] = X;
 }
```

Runtime of Find: <u>O(max height)</u>

Height depends on previous Unions

<u>Best case</u>: 1-2, 1-3, 1-4,… O(1)

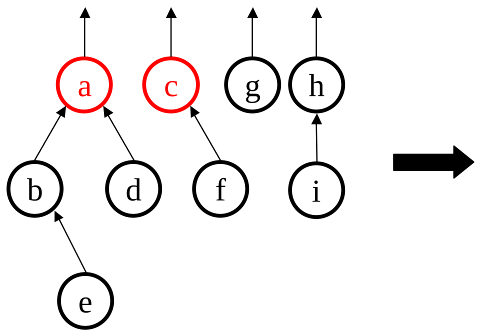
<u>Worst case</u>: 2-1, 3-2, 4-3,… <u>O(N)</u>

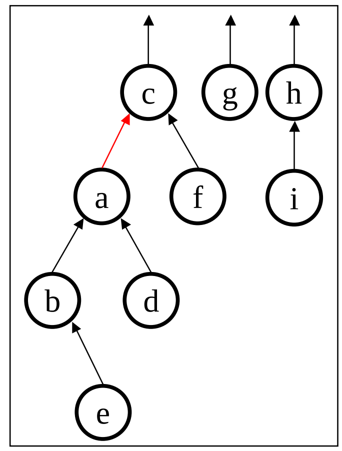
Runtime of Union: <u>O(1)</u>

Can we do better?

Let’s look back at our example…

Union(c,a)

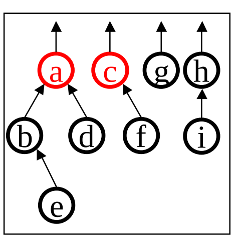




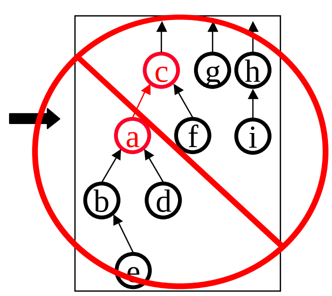
Could we do a better job on this Union? What happened to **e**?

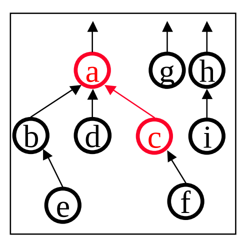
Speeding Up Union/Find: Union-by-Size

- For M Finds and N-1 Unions, worst case time is O(MN+N)
  - Can we speed things up by being clever about growing our up-trees?
- Idea: In Union, always make root of *larger* tree the new root
- Why? Minimizes height of the new up-tree



Union(c,a)

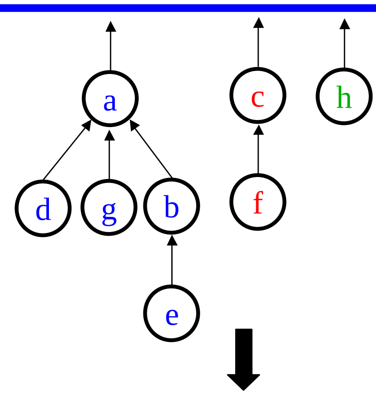




Union-by-Size!

Trick for Storing Size Information

- Instead of storing -1 in root, store up-tree size as negative value in root node



Array up:

0	1 (a)	2 (b)	3 (c)	4 (d)	5 (e)	6 (f)	7 (g)	8 (h)
-	**-5**	1	**-2**	1	2	3	1	**-1**

Union-by-Size Code

```
public void Union(int X, int Y) {
   //X, Y are root nodes
   //containing (-size) of up-trees
  assert(up[X] < 0);
  assert(up[Y] < 0);

  if (-up[X] > -up[Y]) {
  //update size of X and root of Y
     up[X] += up[Y];
     up[Y] = X;
   }
  else { //size of X <= size of Y
     up[Y] += up[X];
     up[X] = Y;
   }
}
```

New run time of Union = ?

New run time of Find = ?

Union-by-Size: Analysis

- Finds are O(max up-tree height) for a forest of up-trees containing N nodes
- Number of nodes in an up-tree of height h using union-by-size is $\geq 2^h$

- Pick up-tree with max height
- Then, $2^{\text{max height}} \leq N$
- max height $\leq \log N$
- Find takes **O(log N)**

Base case: $h = 0$, tree has $2^0 = 1$ node
Induction hypothesis: Assume true for $h < h'$

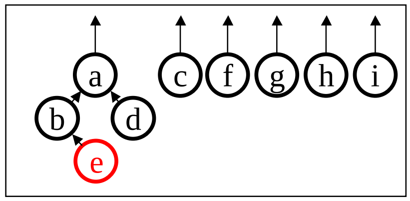
Induction Step: New tree of height h' was formed via union of two trees of height h'-1
Each tree then has $\geq 2^{h'-1}$ nodes by the induction hypothesis
So, total nodes $\geq 2^{h'-1} + 2^{h'-1} = 2^{h'}$
Therefore, True for all h

Union-by-Height

- Textbook describes alternative strategy of Union-by-height
- Keep track of height of each up-tree in the root nodes
- Union makes root of up-tree with greater height the new root
- Same results and similar implementation as Union-by-Size
  - Find is O(log N) and Union is O(1)

Suspense-filled questions to ponder over…

- While doing a find(e), can we do something to speed up future find(e) calls?
- How much speed-up can we get?
- What is the source of the dark matter in the universe?



To be continued next class…

(same place, same time)

Meanwhile…

Finish reading chapter 8