POW

ZING

CSE 326: Data Structures
Lecture #15
The Dynamic (Equivalence) Duo: Weighted Union & Path Compression

Bart Niswonger

Summer Quarter 2001

Whack!!

Zap!

Today's Outline

- Project
  - Rules of competition
- Making a "good" maze
- Disjoint Set Union/Find ADT
- Up-trees
- Weighted Unions
- Path Compression

Unix Tutorial!!

- Tuesday, July 31st
  - 10:50am, Sieg 322

Printing worksheet

Shell
- different shell quotes : ' `
- scripting, #!
- alias
- variables / environment
- redirection, piping

Useful tools
- grep, egrep/grep -e
- sort
- cut
- file
- tr
- find, xargs
- diff, patch
- which, locate, whereis

Finding info
- Techniques
- Resources (ACM webpage, web, internal docs)

Process management

File management/permissions

Filesystem layout

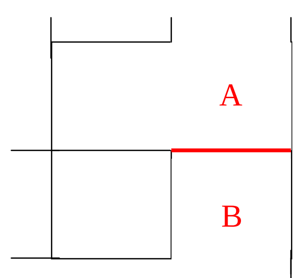
What's a Good Maze?

The Maze Construction Problem

- Given:
  - collection of rooms: **V**
  - connections between rooms (initially all closed): **E**
- Construct a maze:
  - collection of rooms: **V′ = V**
  - designated rooms in, $\mathbf{i} \in \mathbf{V}$, and out, $\mathbf{o} \in \mathbf{V}$
  - collection of connections to knock down: $\mathbf{E'} \subseteq \mathbf{E}$ such that one unique path connects every two rooms

The Middle of the Maze

- So far, a number of walls have been knocked down while others remain.
- Now, we consider the wall between A and B.
- Should we knock it down?
  - if A and B **are** otherwise connected
  - if A and B are **not** otherwise connected



Maze Construction Algorithm

While edges remain in **E**

❶ Remove a random edge **e** = (**u**, **v**) from **E**

❷ If **u** and **v** have not yet been connected

- add **e** to **E′**
- mark **u** and **v** as connected

Mysterious note:
We'll see this algorithm again!

Equivalence Relations

An equivalence relation $\mathcal{R}$ must have three properties

– reflexive: for any *x*, $x\mathcal{R}x$ is true

– symmetric: for any *x* and *y*, $x\mathcal{R}y$ implies $y\mathcal{R}x$

– transitive: for any *x*, *y*, and *z*, $x\mathcal{R}y$ and $y\mathcal{R}z$ implies $x\mathcal{R}z$

Connection between rooms is an equivalence relation

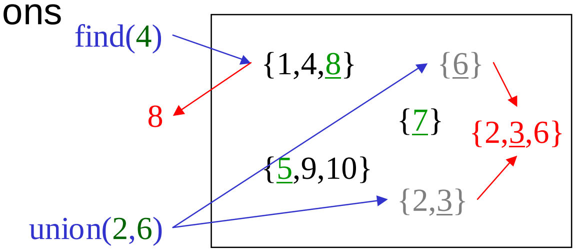
– any room is connected to itself

– if room **a** is connected to room **b**, then room **b** is connected to room **a**

– if room **a** is connected to room **b** and room **b** is connected to room **c**, then room **a** is connected to room **c**

Disjoint Set Union/Find ADT

- Union/Find operations
  - create
  - destroy
  - union
  - find



- Disjoint set equivalence property: every element of a DS U/F structure belongs to exactly one set
- *Dynamic* equivalence property: the set of an element can change after execution of a union

Disjoint Set Union/Find† (More Formally)

- Given a set $U = \{a_1, a_2, \dots, a_n\}$
- Maintain a *partition* of U, a set of subsets of U $\{S_1, S_2, \dots, S_k\}$ such that:
  - each pair of subsets S_i and S_j are disjoint: $S_i \cap S_j = \varnothing$
  - 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 a unique name for a's subset

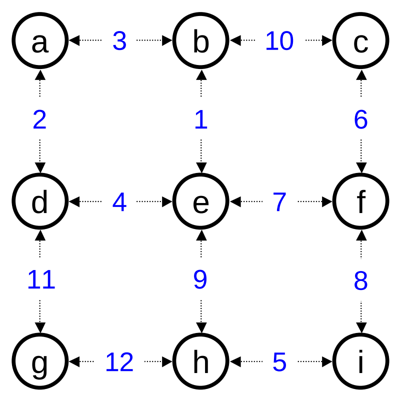
†AKA the dynamic equivalence problem

Example

Construct the maze on the right

Initial (the name of each set is underlined):

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



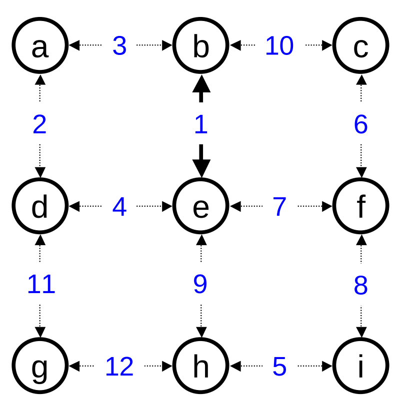
Order of edges in blue

Example, First Step

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

find(b) $\Rightarrow$ b
find(e) $\Rightarrow$ e
find(b) $\neq$ find(e) so:
add 1 to **E′**
union(b, e)

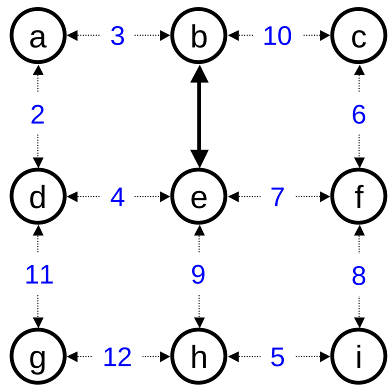
{a}{b,e}{c}{d}{f}{g}{h}{i}



Order of edges in blue

Example, Continued

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



Order of edges in blue

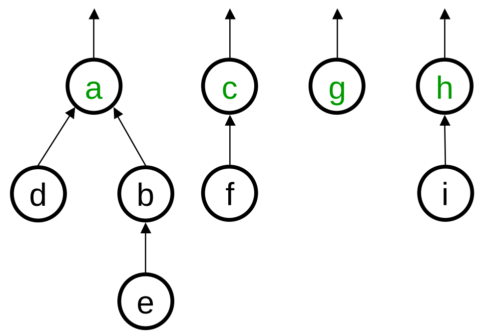
Up-Tree Intuition

Finding the representative member of a set is somewhat like the *opposite* of finding whether a given key exists in a set.

So, instead of using trees with pointers from each node to its children; let's use trees with a pointer from each node to its parent.

Up-Tree Union-Find Data Structure

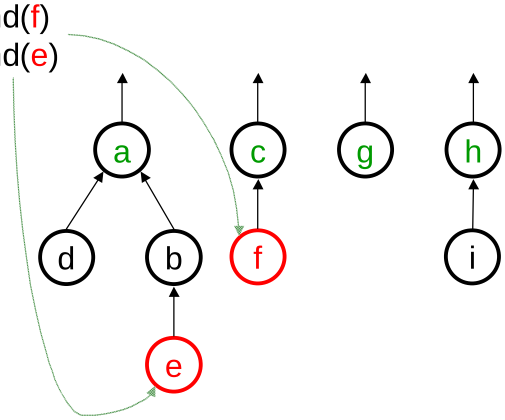
- Each subset is an up-tree with its root as its representative member
- All members of a given set are nodes in that set's up-tree
- Hash table maps input data to the node associated with that data

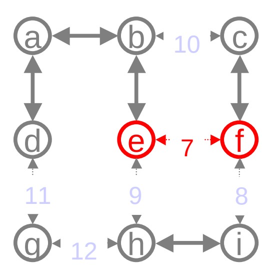


Up-trees are **not** necessarily binary!

Find

find(f)
find(e)



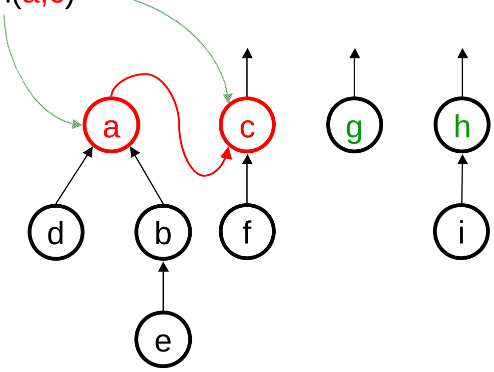


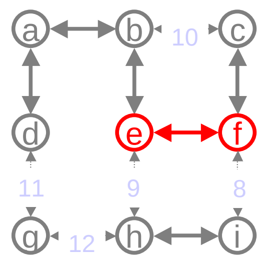
runtime:

Just traverse to the root!

Union

union(a,c)

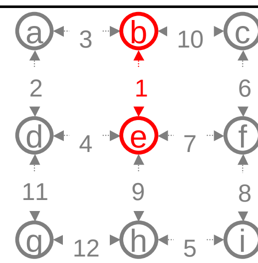




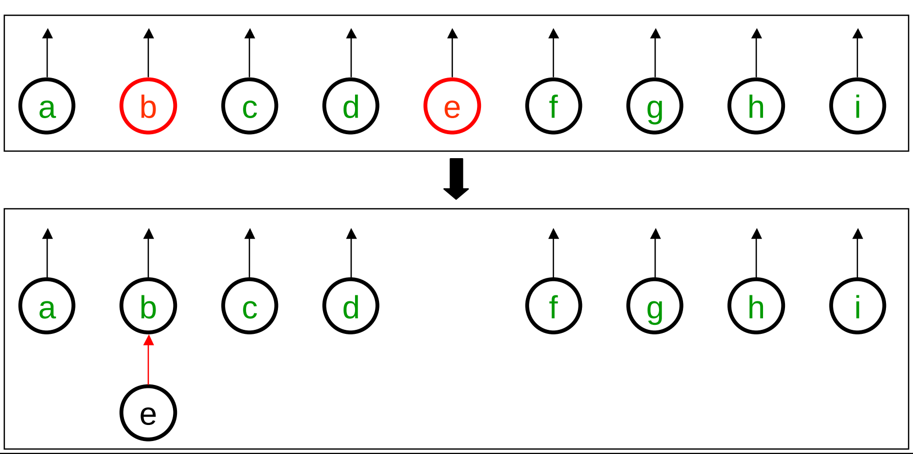
runtime:

Just hang one root from the other!

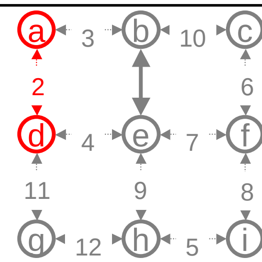
The Whole Example (1/11)



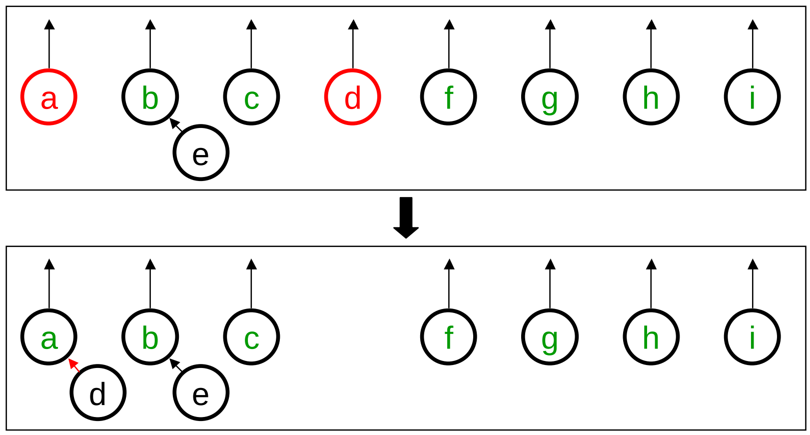
union(b,e)



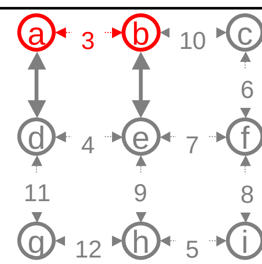
The Whole Example (2/11)



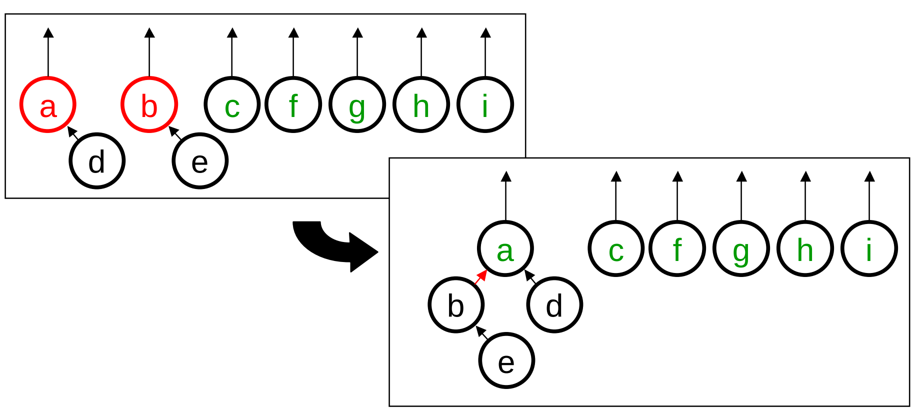
union(a,d)



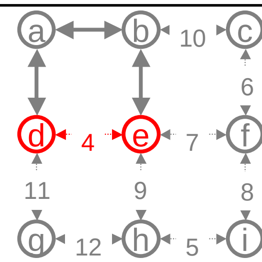
The Whole Example (3/11)



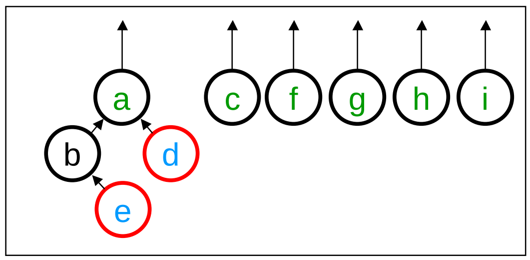
union(a,b)



The Whole Example (4/11)

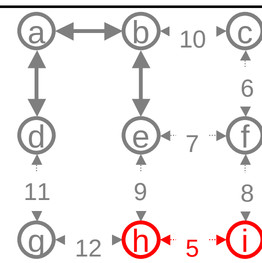


find(d) = find(e)
No union!

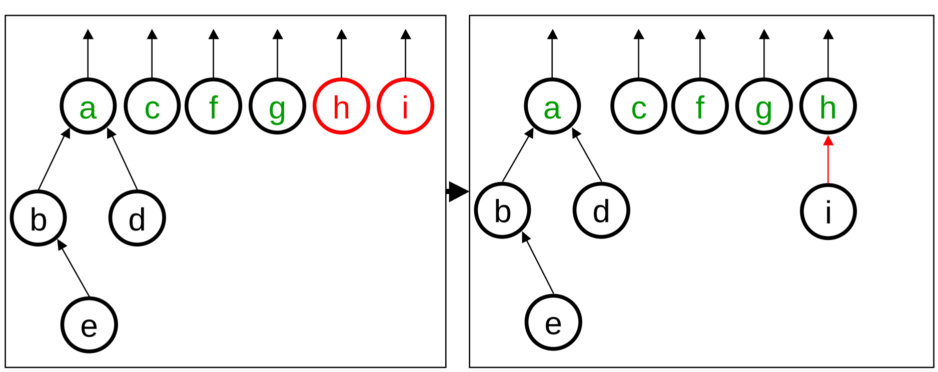


While we're finding ***e***,
could we do anything else?

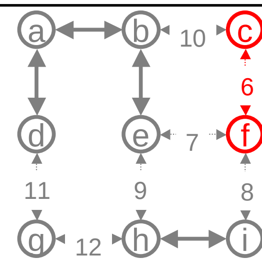
The Whole Example (5/11)



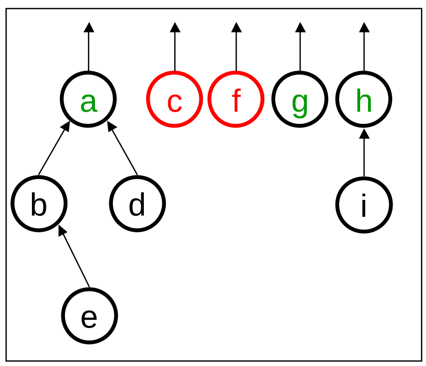
union(h,i)

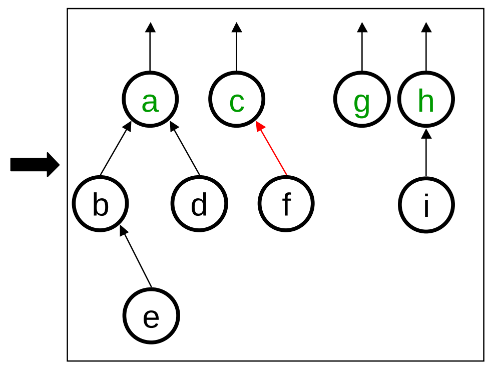


The Whole Example (6/11)

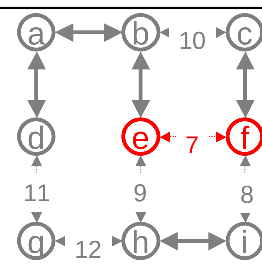


union(c,f)

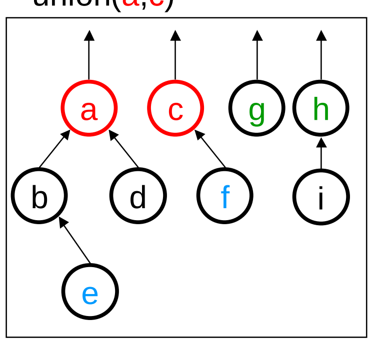


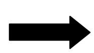


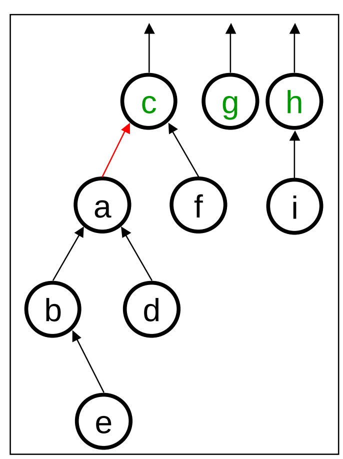
The Whole Example (7/11)



find(e)
find(f)
union(a,c)

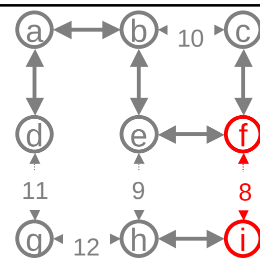




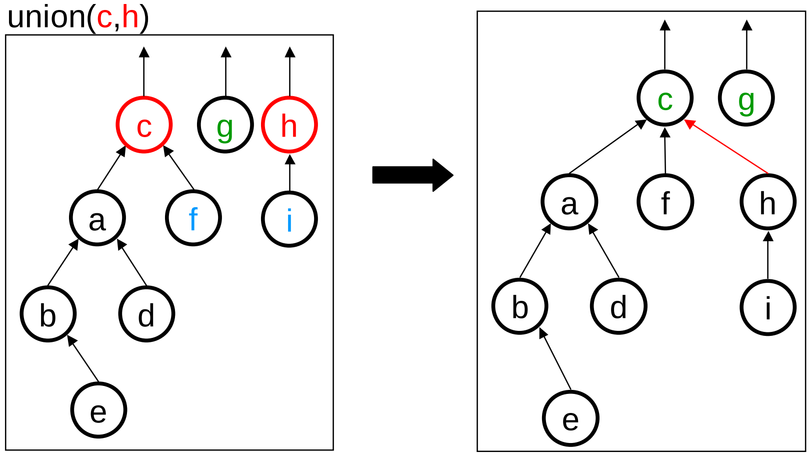


Could we do a
better job on this union?

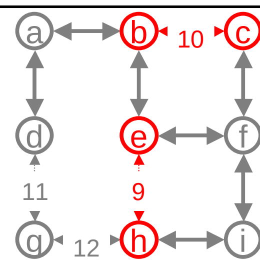
The Whole Example (8/11)



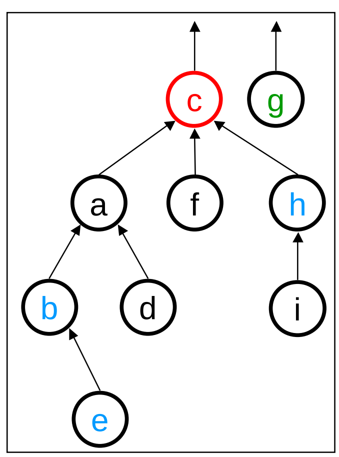
find(f)
find(i)
union(c,h)



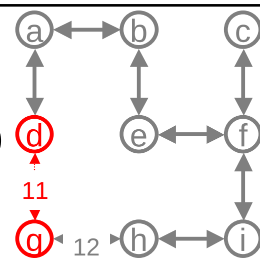
The Whole Example (9/11)



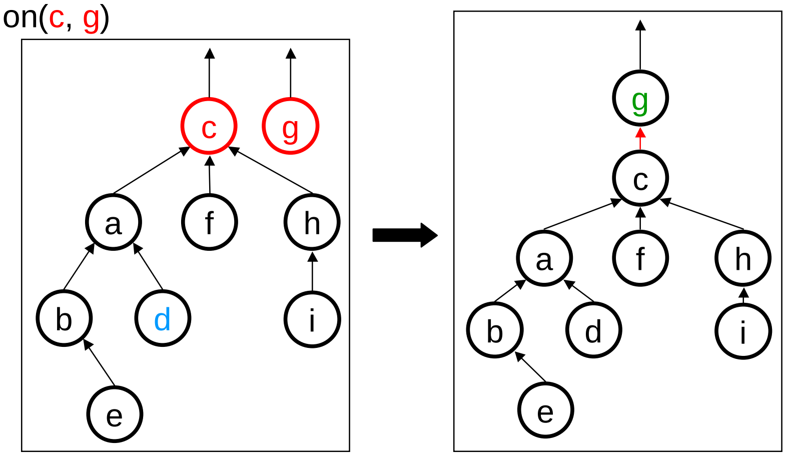
find(e) = find(h) and find(b) = find(c)
So, no unions for either of these.



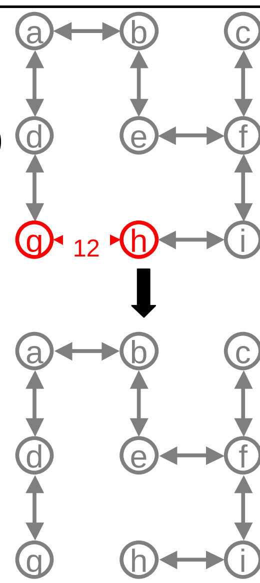
The Whole Example (10/11)



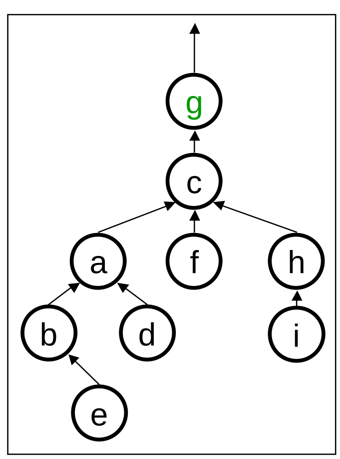
find(d)
find(g)
union(c, g)



The Whole Example (11/11)



find(g) = find(h)
So, no union.
And, we're done!

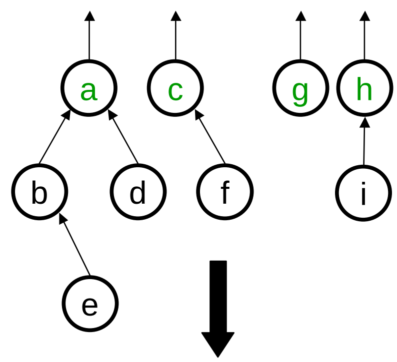


Ooh… scary!
Such a hard maze!

Nifty storage trick

A forest of up-trees can easily be stored in an array.

Also, if the node names are integers or characters, we can use a very simple, perfect hash.



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

Implementation

```
typedef ID int;

ID find(Object x) {
  assert(hTable.contains(x));
  ID parentID = hTable[x];

  while(up[parentID] != -1) {
    parentID = up[xID];
  }

  return parentID;
}
```

runtime: O(depth) or …

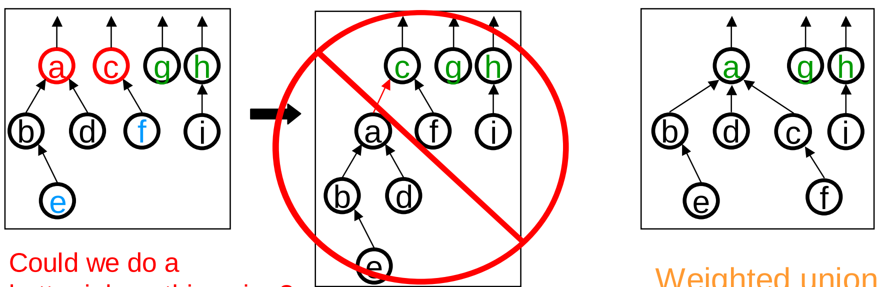
```
ID union(ID x, ID y) {
  assert(up[x] == -1);
  assert(up[y] == -1);

  up[y] = x;
}
```

runtime: O(1)

Improvement: Weighted Union

- Always makes the root of the larger tree the new root
- Often cuts down on height of the new up-tree



Could we do a better job on this union?

Weighted union!

Weighted Union Code

```
typedef ID int;
ID union(ID x, ID y) {
  assert(up[x] == -1);
  assert(up[y] == -1);

  if (weight[x] > weight[y]) {
    up[y] = x;
    weight[x] += weight[y];
  }
  else {
    up[x] = y;
    weight[y] += weight[x];
  }
}
```

new runtime of union:

new runtime of find:

Weighted Union Find Analysis

- Finds with weighted union are O(max up-tree height)
- But, an up-tree of height h with weighted union must have at least 2^h nodes

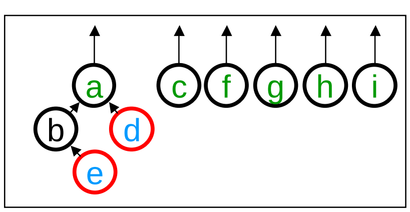
Base case: $h = 0$, tree has $2^0 = 1$ node
Induction hypothesis: assume true for $h < h'$

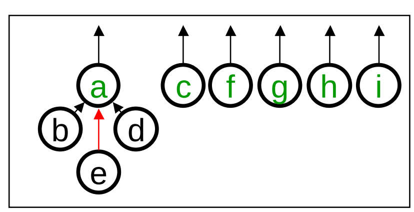
A merge can only increase tree height by one over the smaller tree. So, a tree of height h'-1 was merged with a larger tree to form the new tree. Each tree then has $\geq 2^{h'-1}$ nodes by the induction hypotheses for a total of at least $2^{h'}$ nodes. QED.

- $\therefore$, $2^{\text{max height}} = n$ and max height = log n
- So, find takes O(log n)

Improvement: Path Compression

- Points everything along the path of a find to the root
- Reduces the height of the entire access path to 1



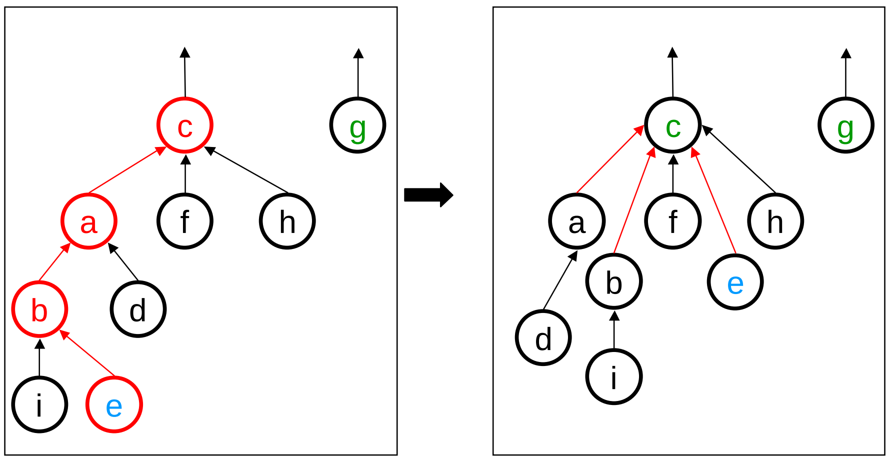


While we're finding ***e***,
could we do anything else?

Path compression!

Path Compression Example

find(e)



Path Compression Code

```
typedef ID int;
ID find(Object x) {
  assert(hTable.contains(x));
  ID parentID = hTable[x];
  ID hold = parentID;
  while(up[parentID] != -1) {
    parentID = up[parentID];
  }
  ID rootID = parentID;
  while(up[hold] != -1) {
    ID oldParentID = up[hold];
    up[hold] = rootID;
    hold = oldParentID;
  }
  return rootID;
}
```

runtime:

Digression: Doping at the Silicon Downs

How fast does log n grow?   log n = 4 for n = 16

Let $\log^{(k)} n = \underbrace{\log (\log (\log \ldots (\log}_{k \text{ logs}} n)))$

Then, let $\log^* n$ = minimum k such that $\log^{(k)} n \leq 1$

How fast does $\log^*$ n grow? $\log^*$ n = 4 for n = 65536

Ackermann created a really big function A(x, y) with the inverse α(x, y) which is really small

How fast does α(x, y) grow? α(x, y) = 4 for n **far** larger than the number of atoms in the universe (2^{300})

Complex Complexity of Weighted Union + Path Compression

- Tarjan proved that *m* weighted union and find operations on a set of *n* elements have worst case complexity $O(m \cdot \alpha(m, n))$
- For **all** practical purposes this is amortized constant time
- In some practical cases, one or both is unnecessary because trees do not naturally get very deep.

To Do

- Start Project III (only 5 days!)
- Read chapter 8 in the book
- Start reading chapter 7

Coming Up

- **Algorithms**
- Sorting (Chapter 7)
- Project III due (next Wednesday)

- Unix Tutorial (next Tuesday)