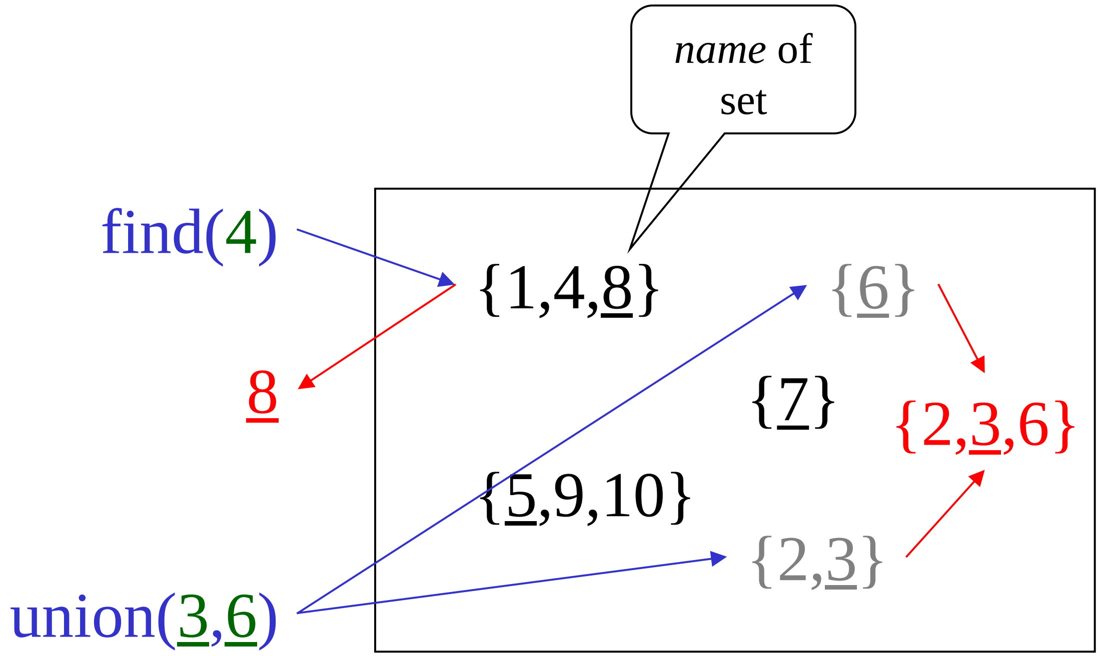
Disjoint Set Union/Find ADT

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



- *Disjoint set partition property*: every element of a DS U/F structure belongs to *exactly one set* with a *unique name*
- *Dynamic equivalence property*: Union(a, b) creates a new set which is the union of the sets containing a and b

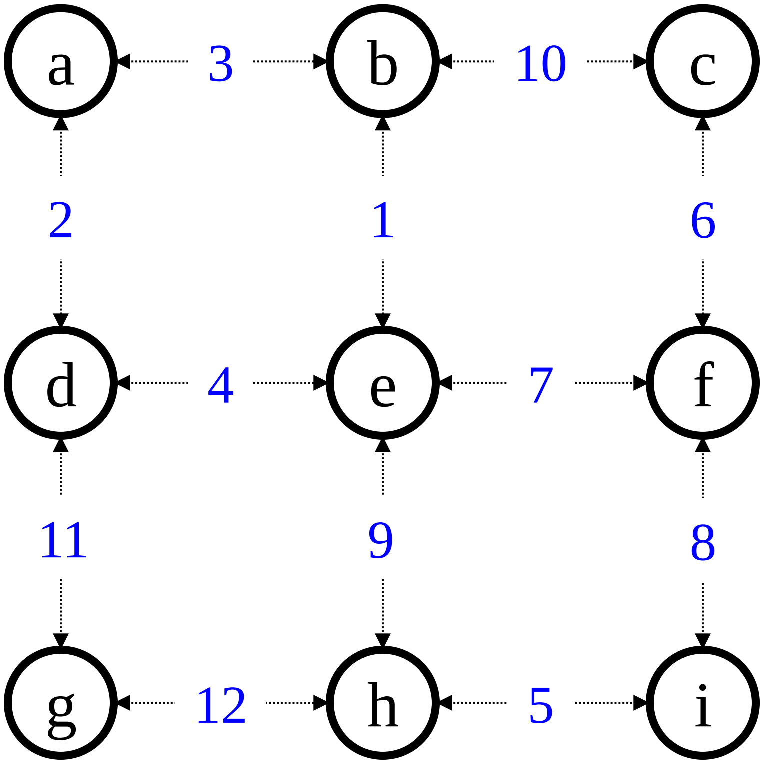
Example

Construct the maze on the right

Initial (the name of each set is underlined):

{<u>a</u>}{<u>b</u>}{<u>c</u>}{<u>d</u>}{<u>e</u>}{<u>f</u>}{<u>g</u>}{<u>h</u>}{<u>i</u>}

Randomly select edge 1



Order of edges in blue

Example, First Step

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

find(b) $\Rightarrow$ b

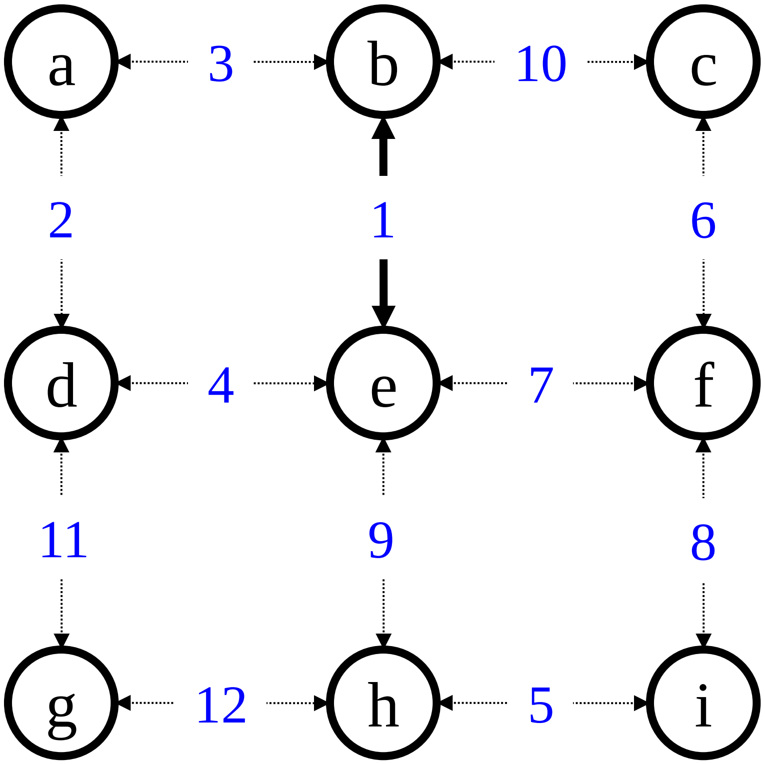
find(e) $\Rightarrow$ e

find(b) $\neq$ find(e) so:

add 1 to **E′**

union(b, e)

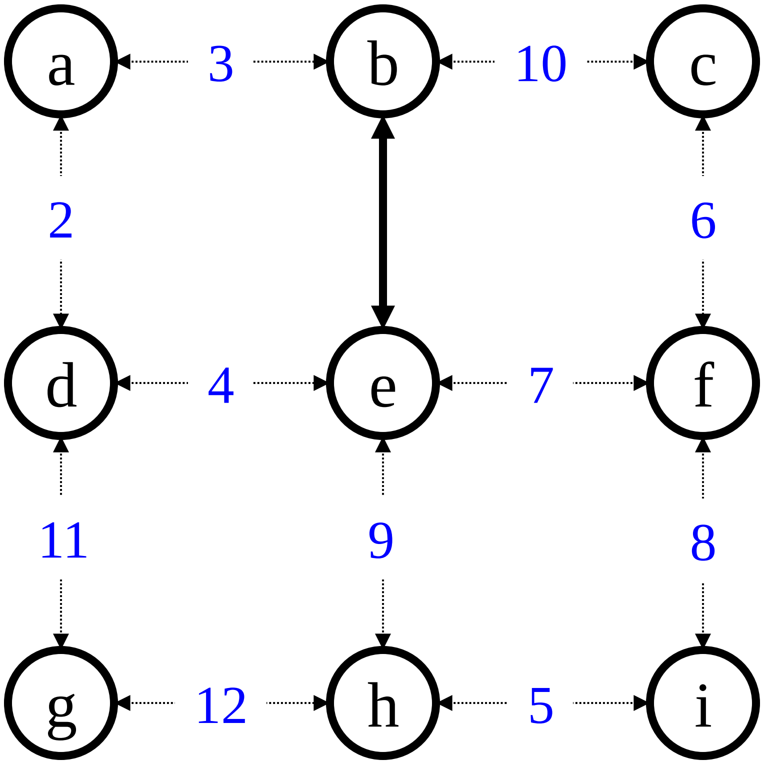
{a}{b,e}{c}{d}{f}{g}{h}{i}



Order of edges in blue

Example, Continued

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



Order of edges in blue

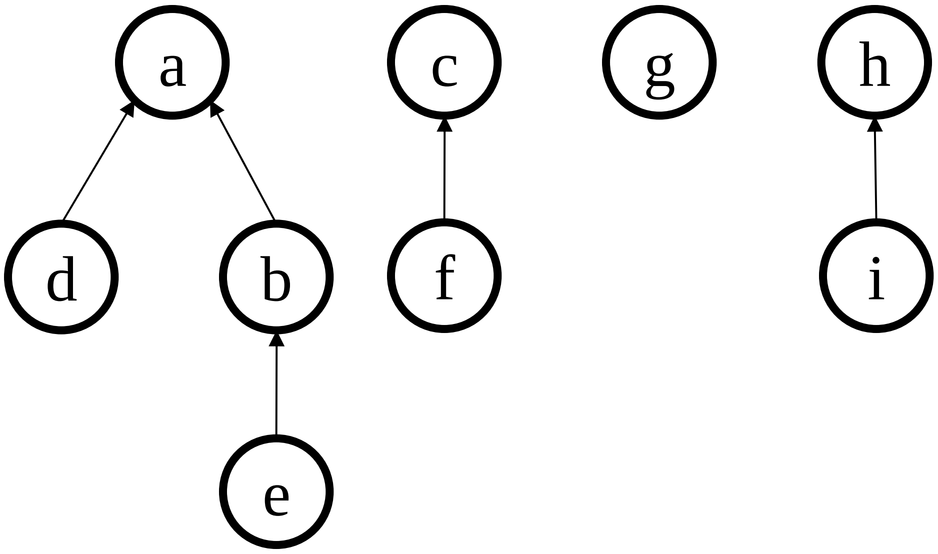
Up-Tree Intuition

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

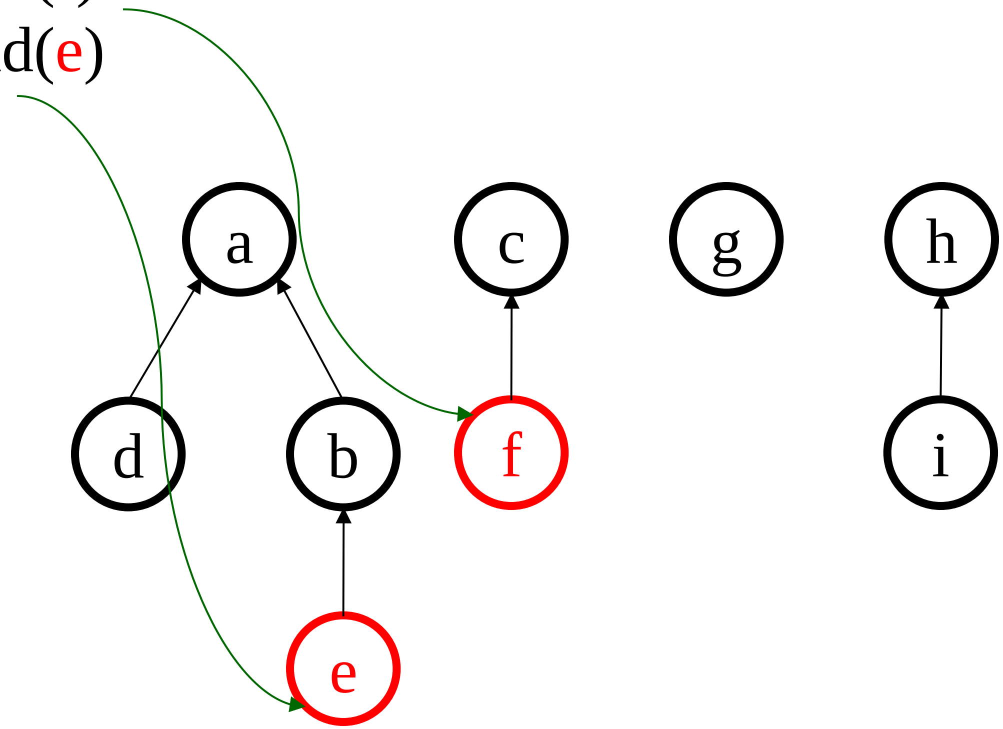
- Each subset is an up-tree with its root as its representative member
- All members of a given set are nodes in that set's up-tree
- Hash table maps input data to the node associated with that data

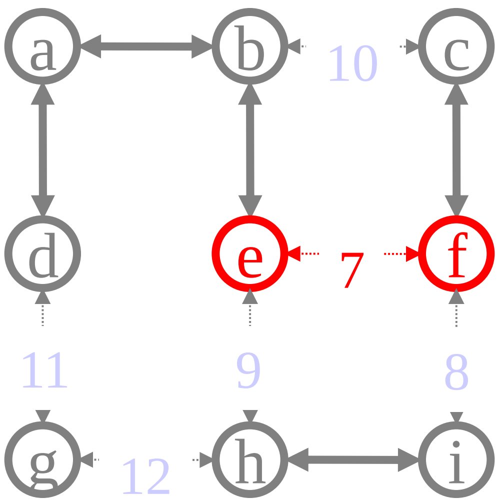


Up-trees are **not** necessarily binary!

Find

find(f)
find(e)

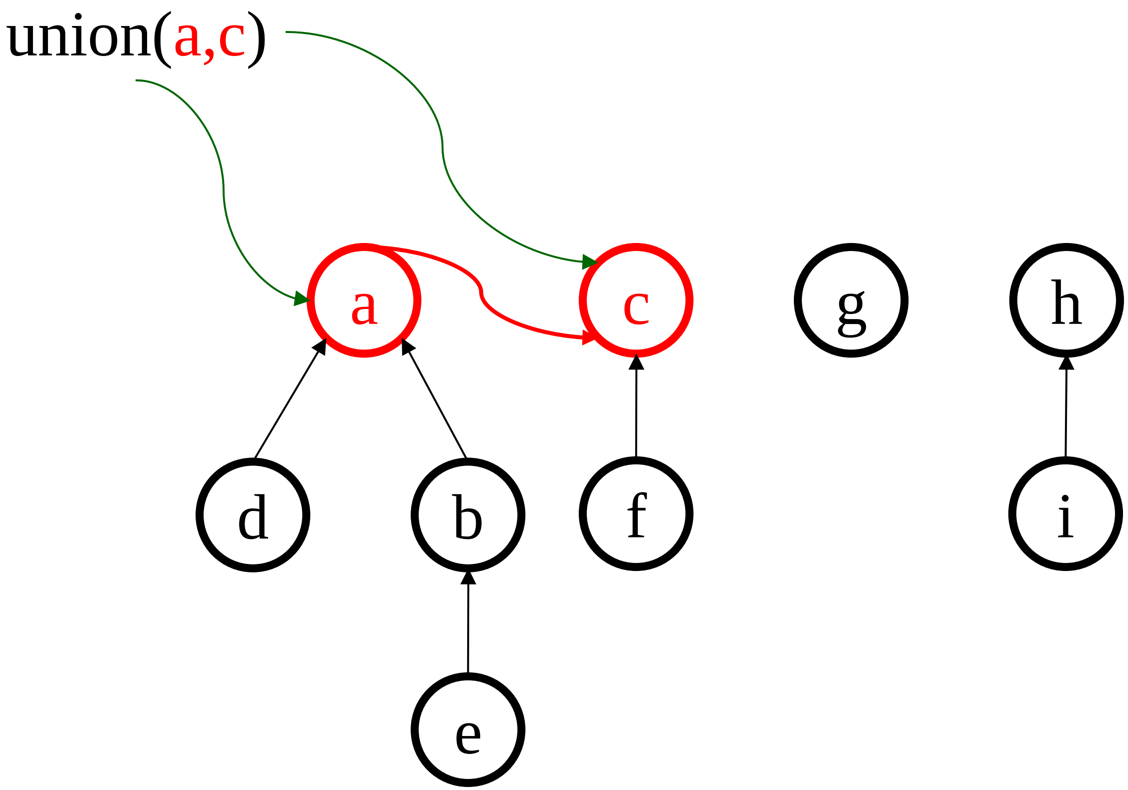


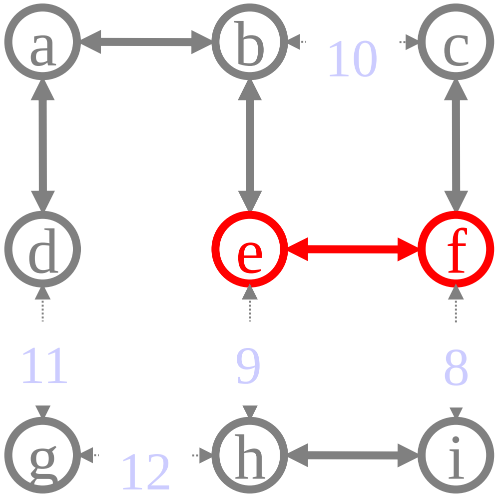


runtime:

Just traverse to the root!

Union

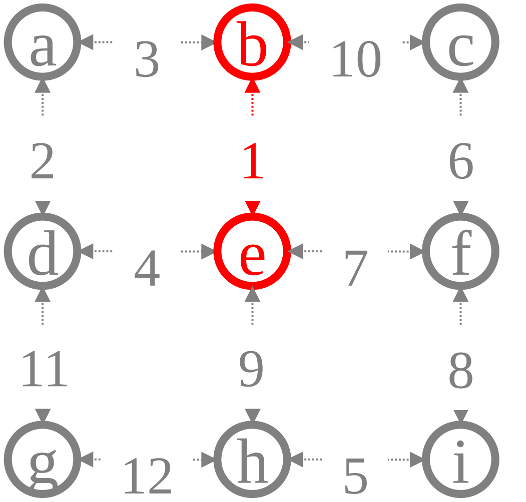




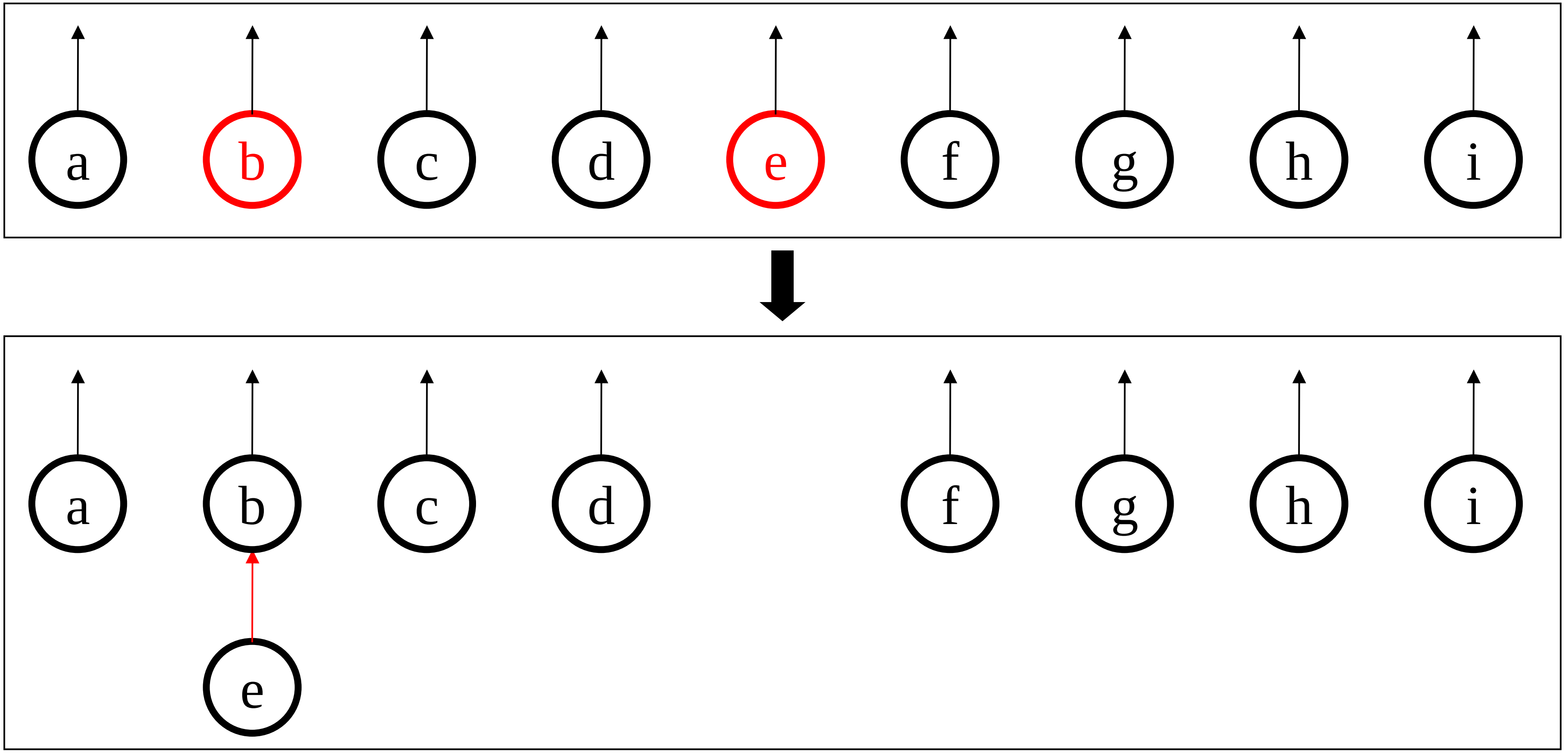
runtime:

Just hang one root from the other!

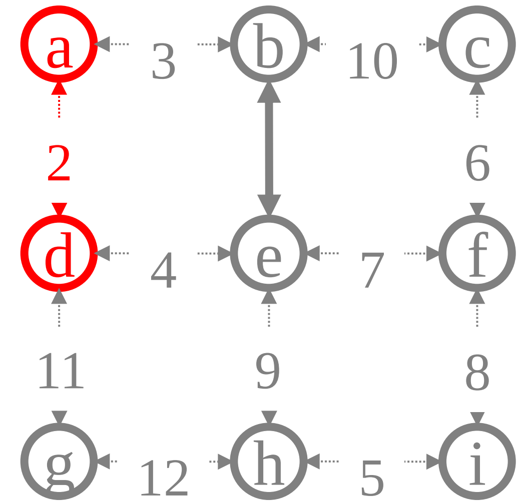
The Whole Example (1/11)



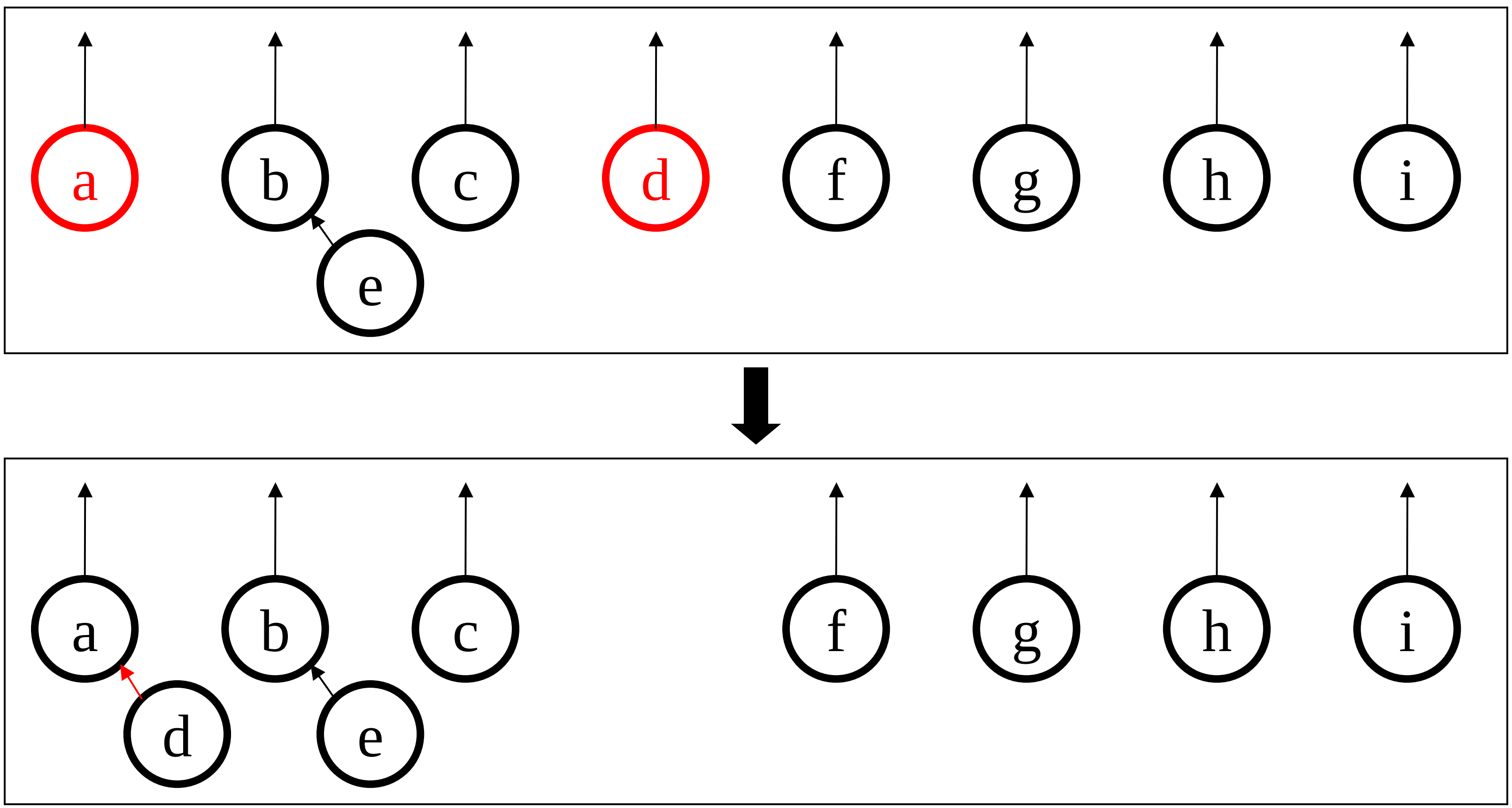
union(b,e)



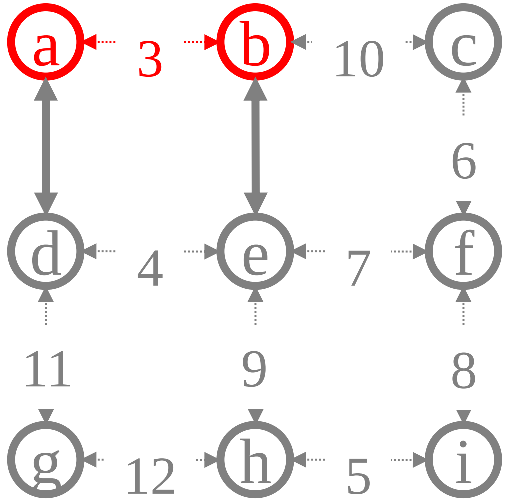
The Whole Example (2/11)



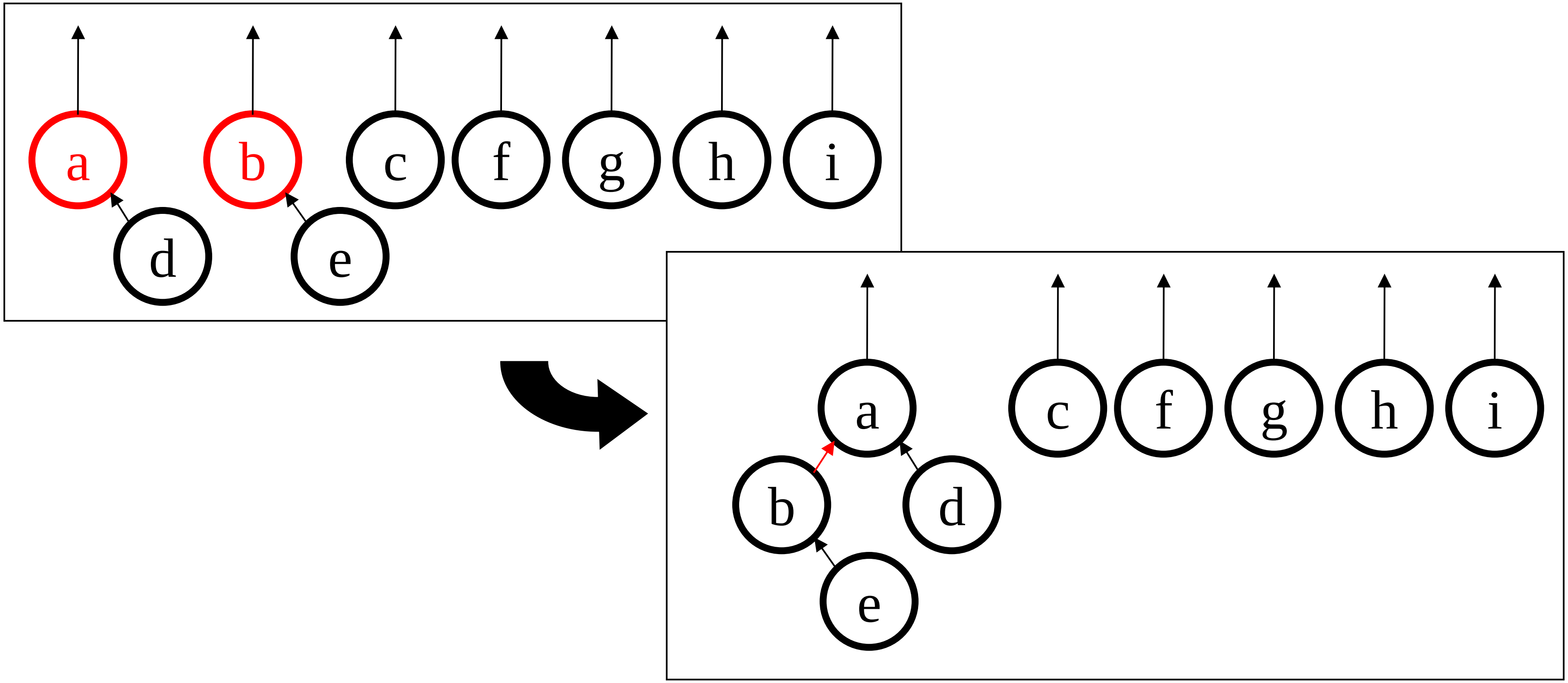
union(a,d)



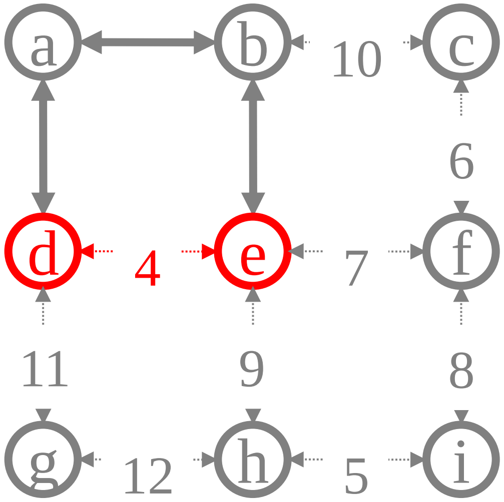
The Whole Example (3/11)



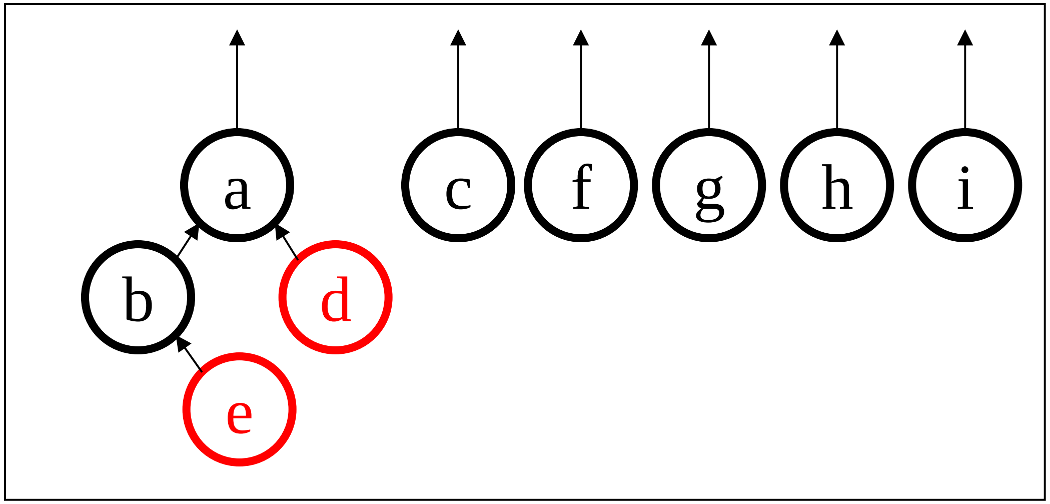
union(a,b)



The Whole Example (4/11)

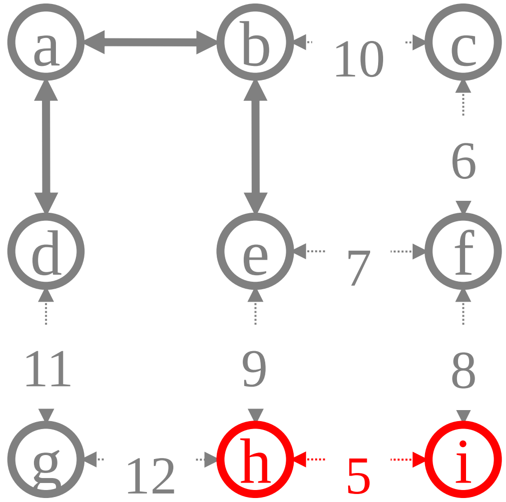


find(d) = find(e)
No union!

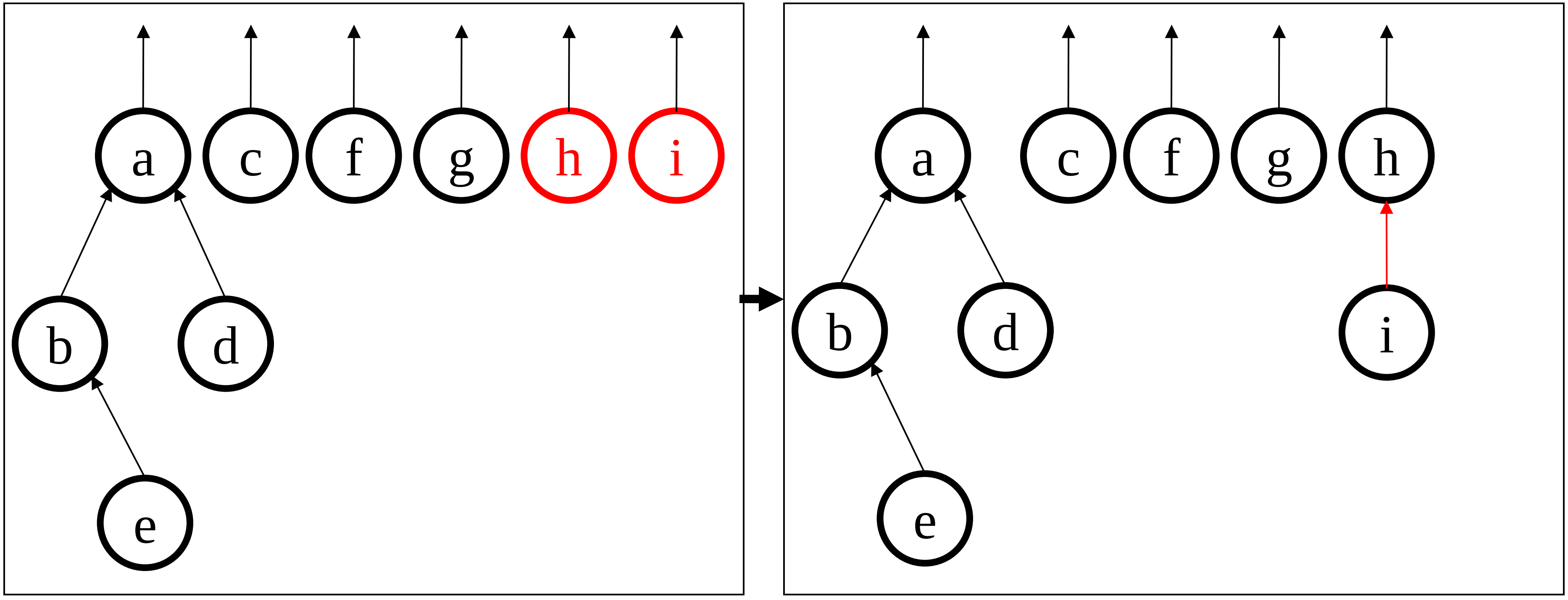


While we’re finding ***e***,
could we do anything else?

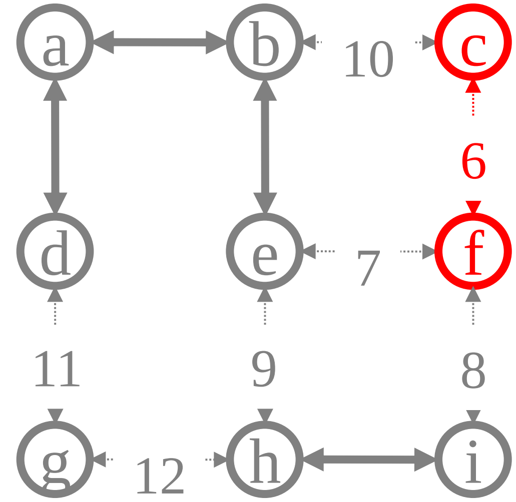
The Whole Example (5/11)



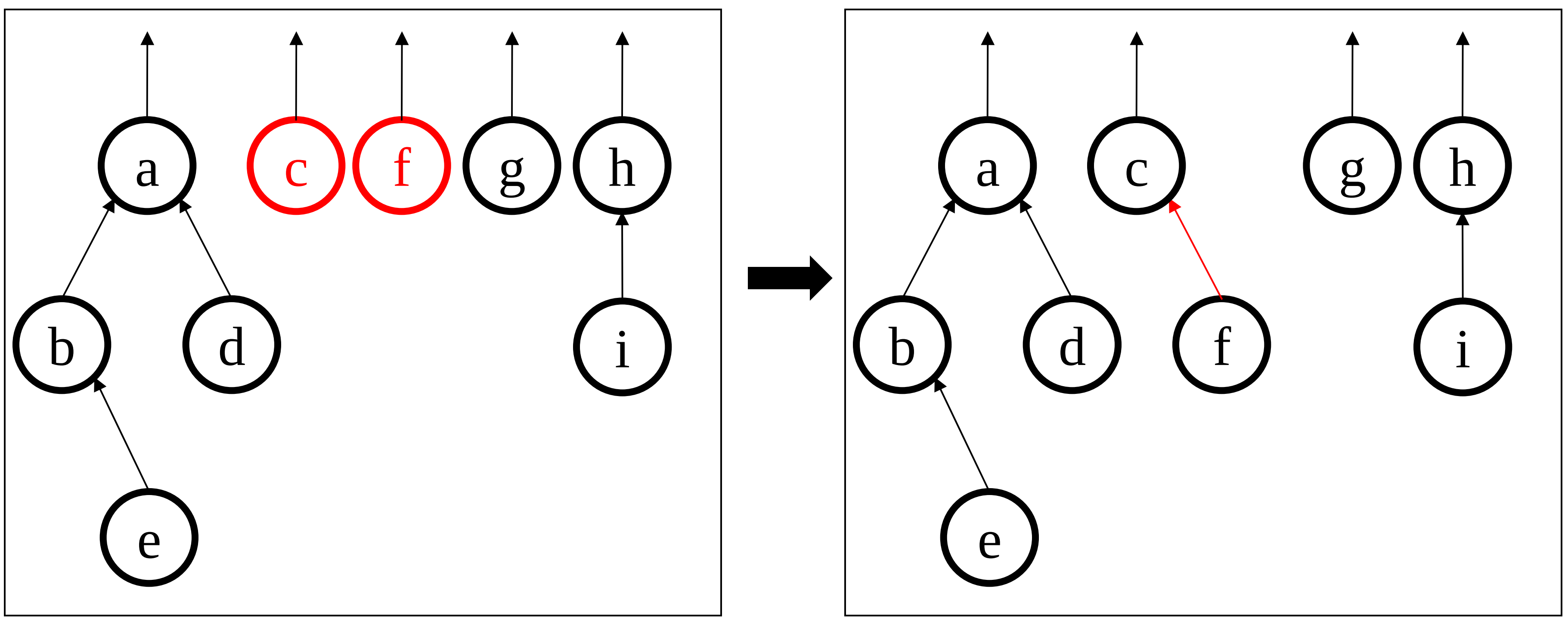
union(h,i)



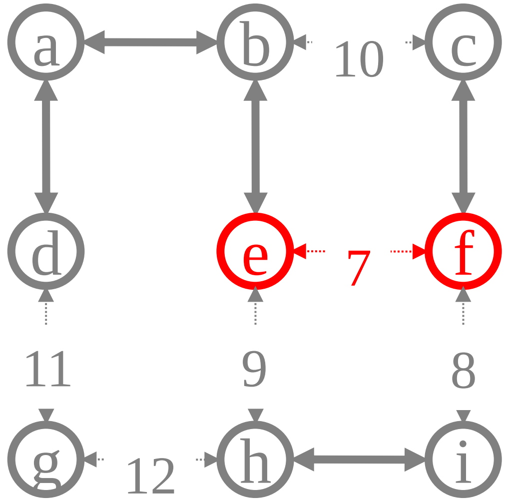
The Whole Example (6/11)



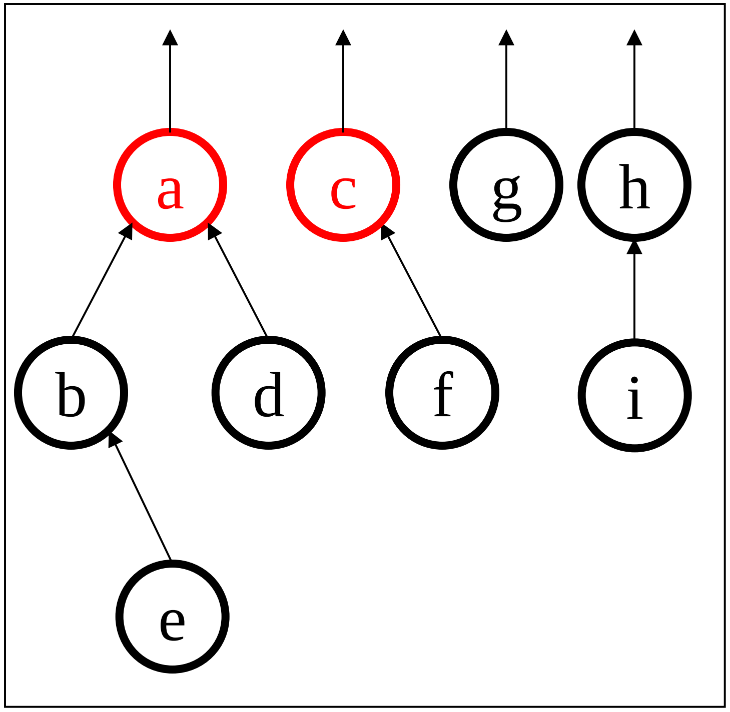
union(c,f)

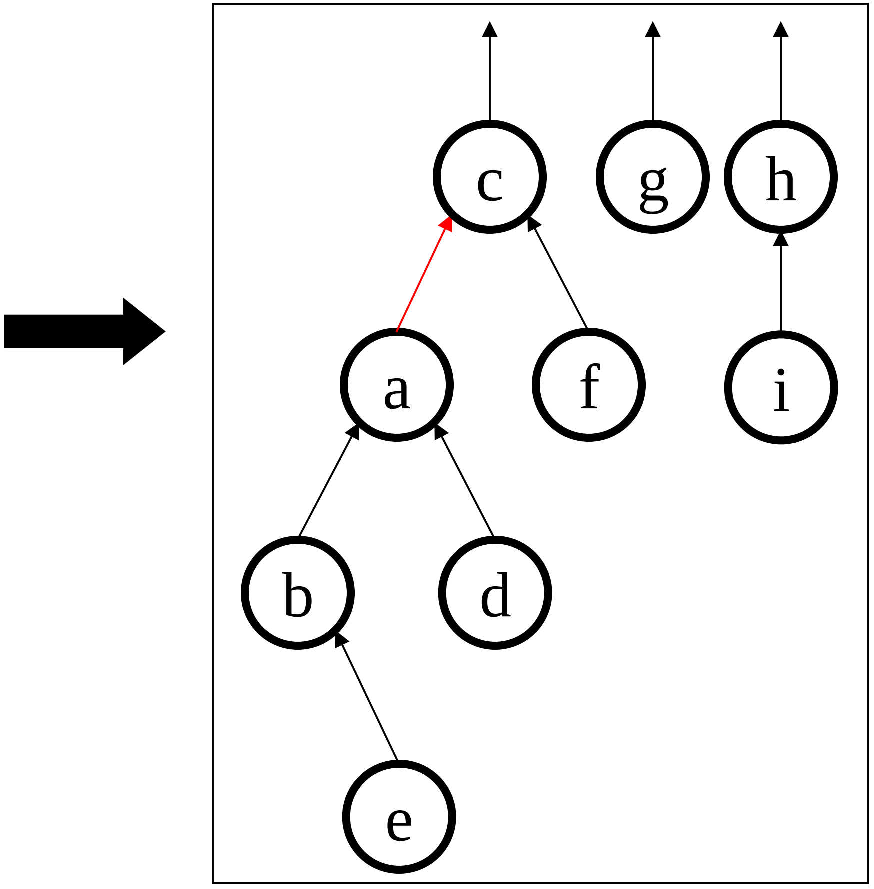


The Whole Example (7/11)



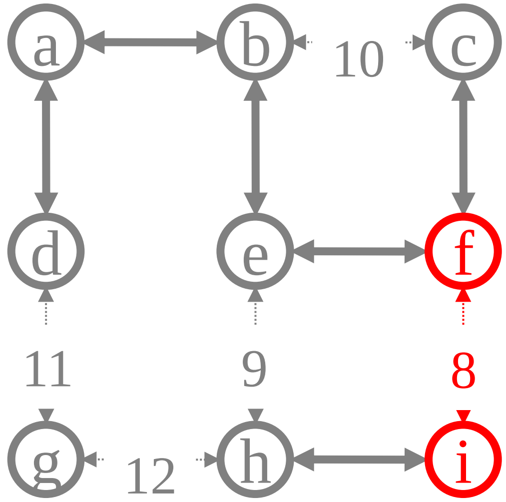
find(e)
find(f)
union(a,c)



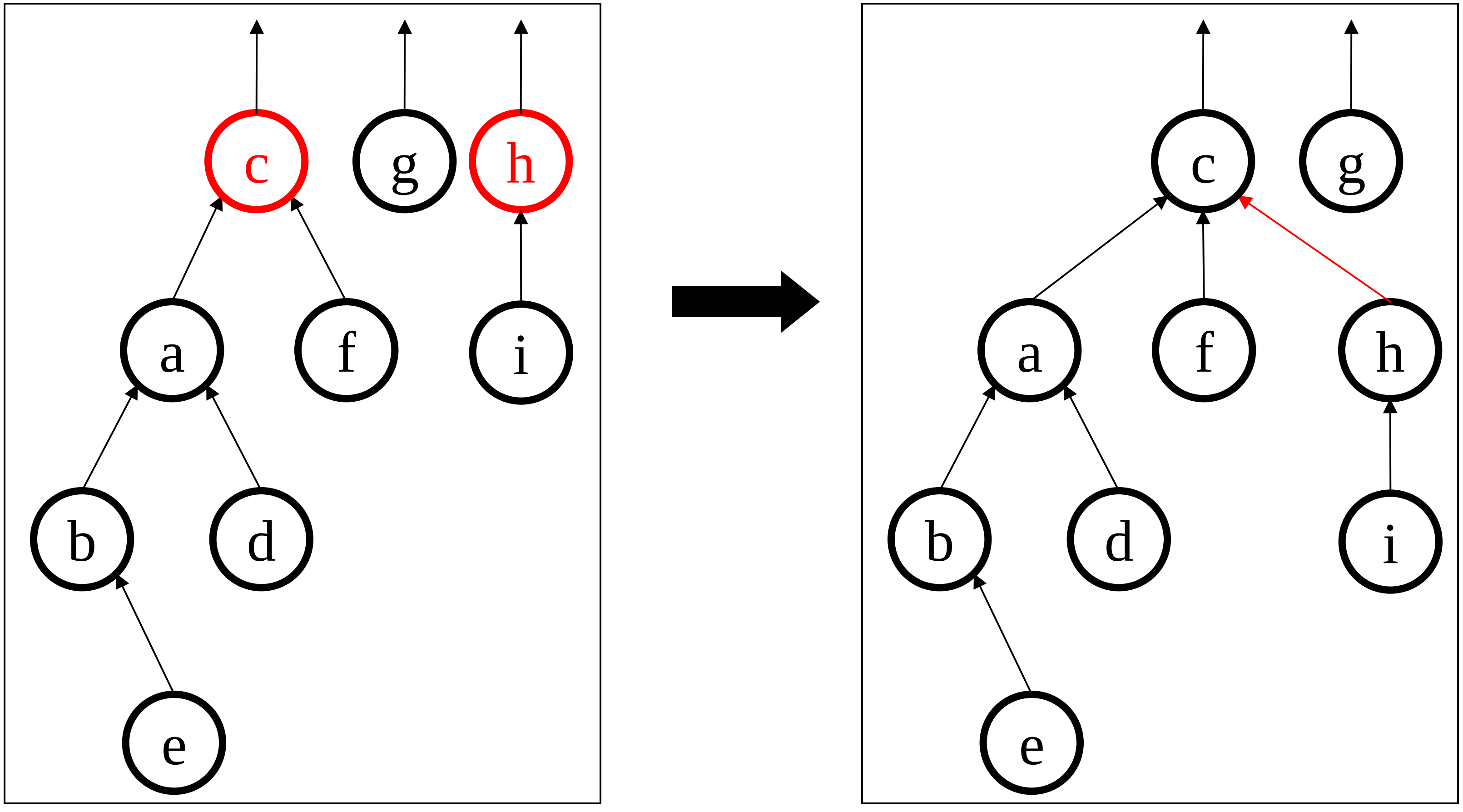


Could we do a
better job on this union?

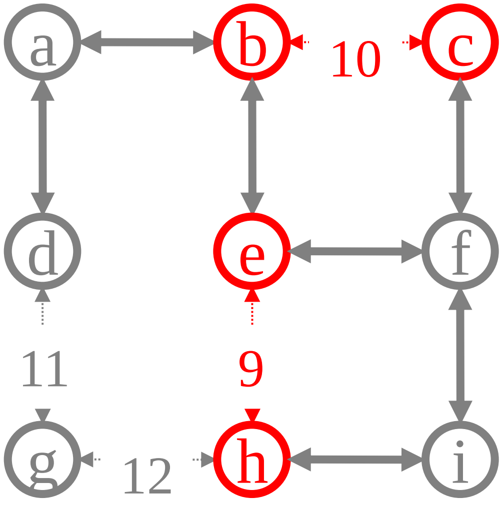
The Whole Example (8/11)



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

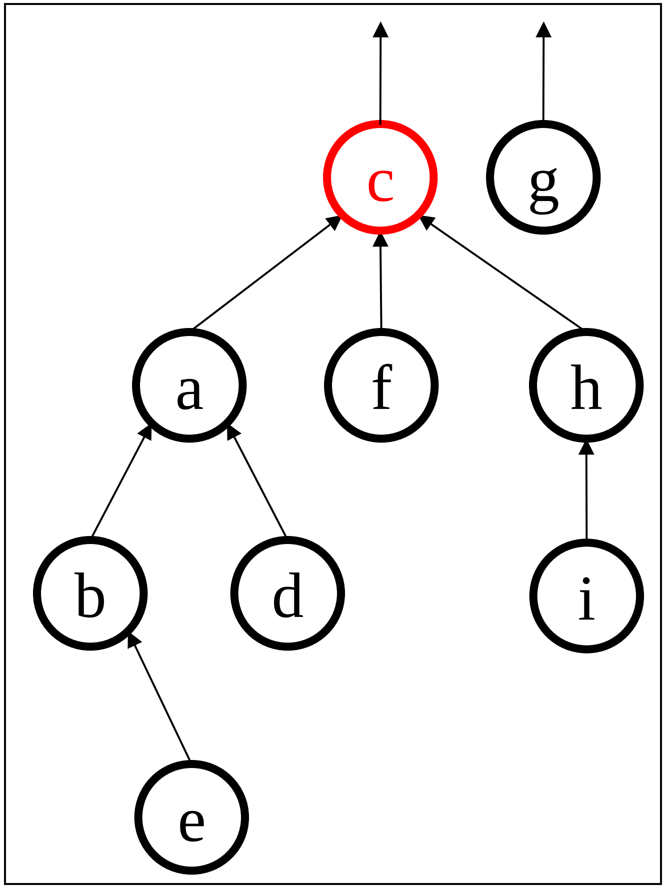


The Whole Example (9/11)

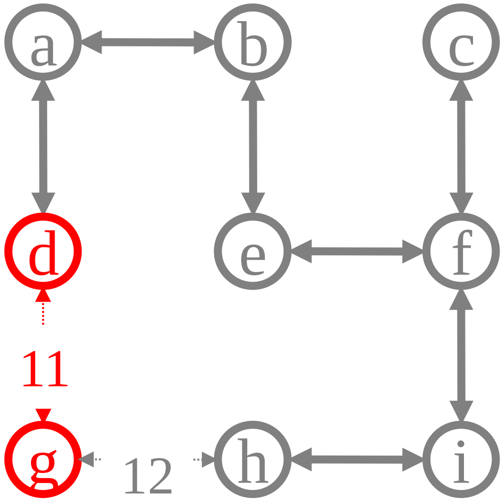


find(e) = find(h) and find(b) = find(c)

So, no unions for either of these.



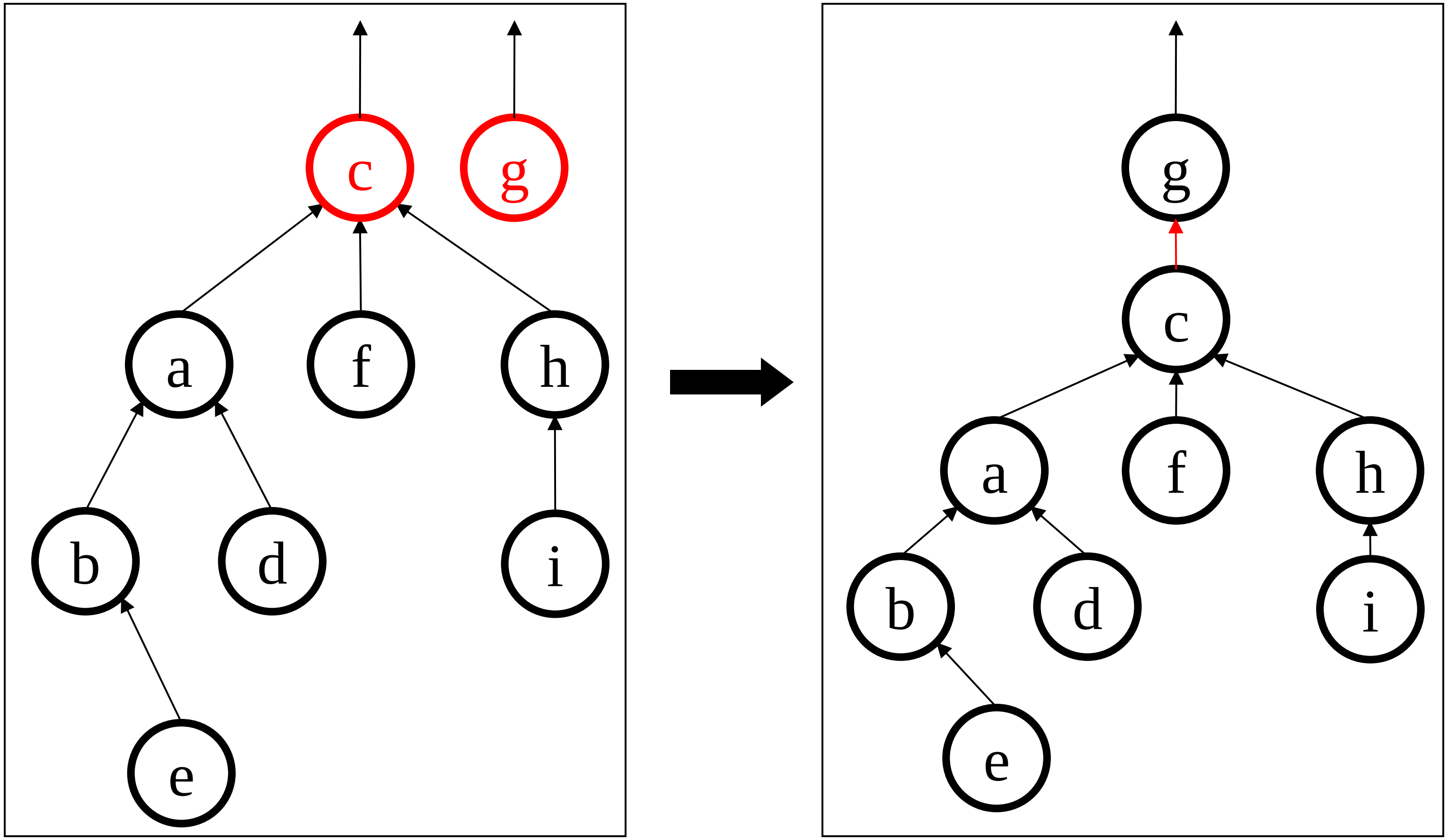
The Whole Example (10/11)



find(d)

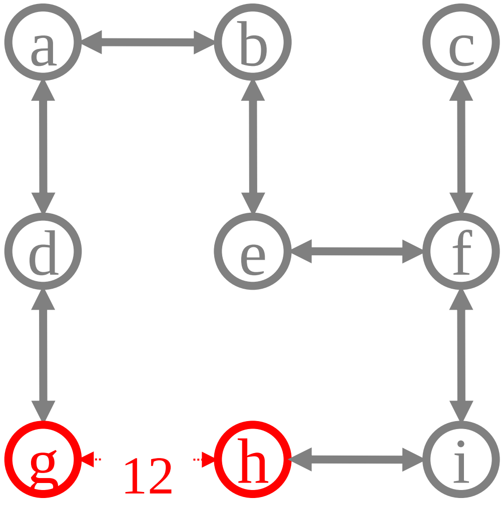
find(g)

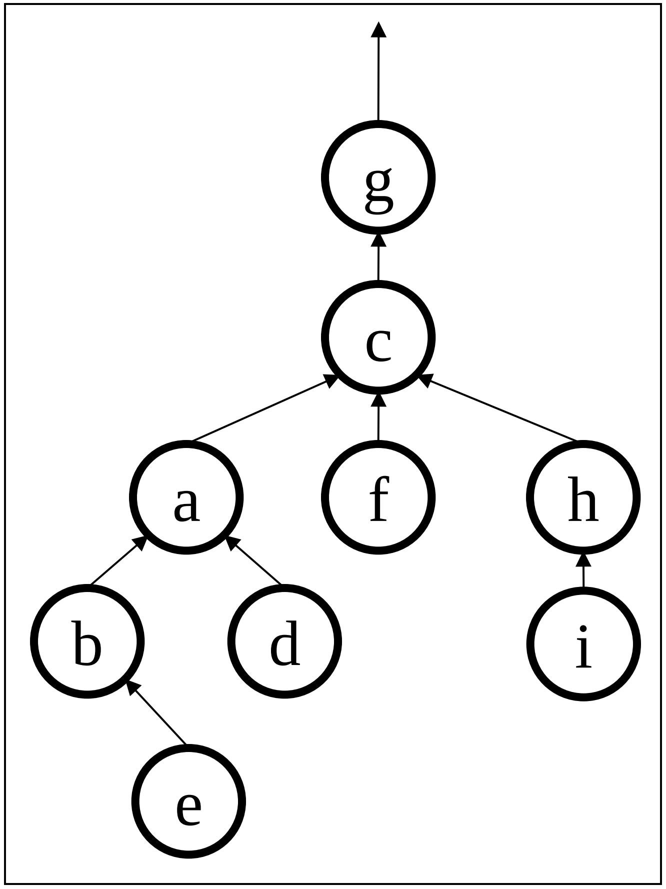
union(c, g)

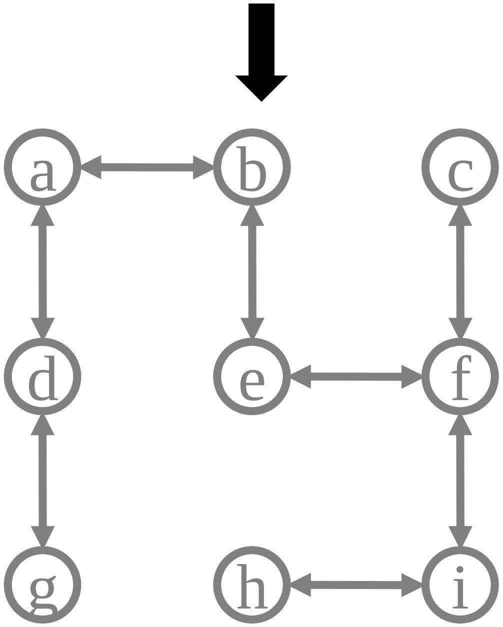


The Whole Example (11/11)

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





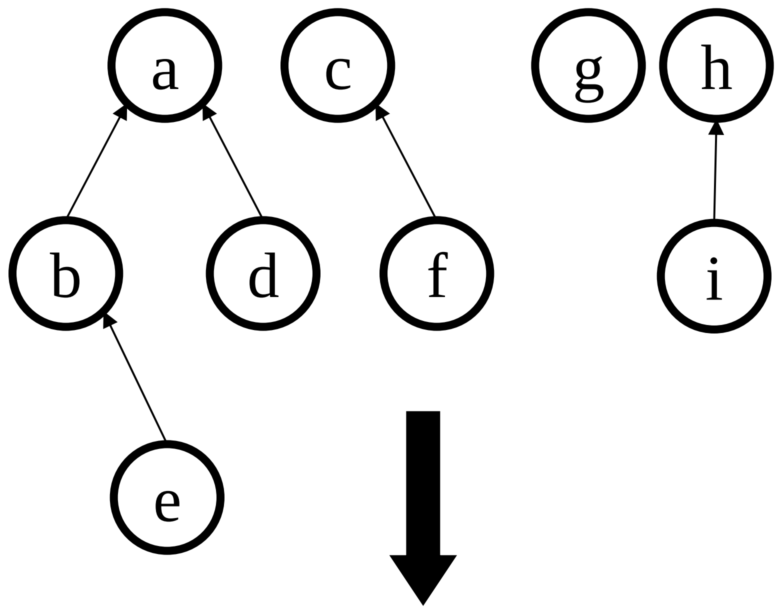


Ooh… scary!
Such a hard maze!

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 up[10000];

ID find(Object x)
{
  assert(HashTable.contains(x));
  ID xID = HashTable[x];
  while(up[xID] != -1) {
    xID = up[xID];
  }
  return xID;
}
```

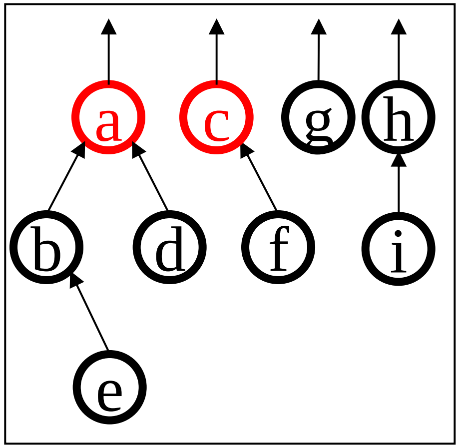
```
ID union(Object x, Object y)
{
  ID rootx = find(x);
  ID rooty = find(y);
  assert(rootx != rooty);
  up[y] = x;
}
```

runtime: O(depth) or …

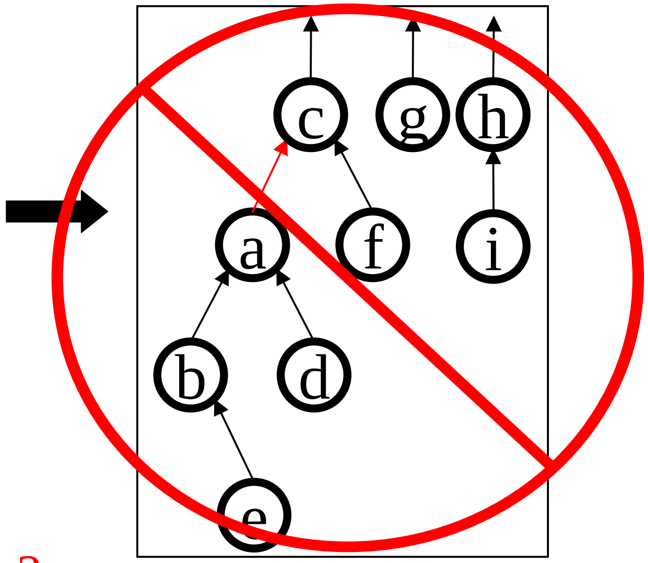
runtime: O(1)

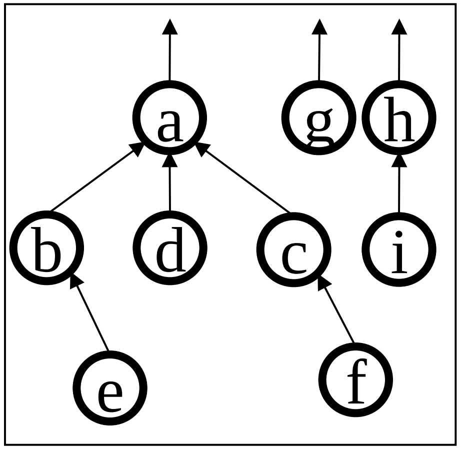
Room for Improvement: Weighted Union

- 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(Object x, Object y) {
  rx = Find(x);
  ry = Find(y);
  assert(rx != ry);
  if (weight[rx] > weight[ry]) {
    up[ry] = rx;
    weight[rx] += weight[ry];
  }
  else {
    up[rx] = ry;
    weight[ry] += weight[rx];
  }
}
```

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 $h < h'$ and consider the sequence of unions.
Case 1: Union does not increase max height. Resulting tree still has $\geq 2^h$ nodes.
Case 2: Union has height $h' = 1+h$, where h = height of each of the input trees. By induction hypothesis each tree has $\geq 2^{h'-1}$ nodes, so the merged tree has at least $2^{h'}$ nodes. QED.

∀ $\therefore$, $2^{\text{max height}} \leq n$ and max height $\leq \log n$

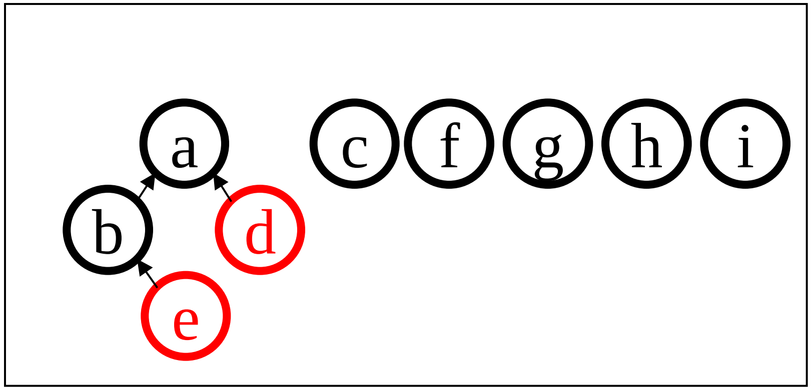
- So, find takes O(log n)

Alternatives to Weighted Union

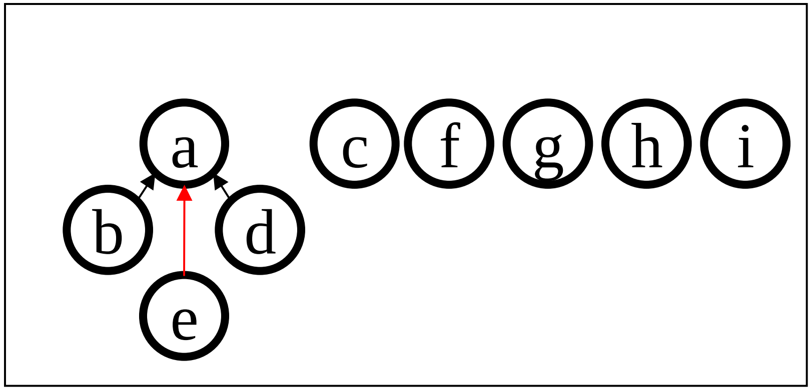
- Union by height
- Ranked union (cheaper approximation to union by height)
- See Weiss chapter 8.

Room for Improvement: Path Compression

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



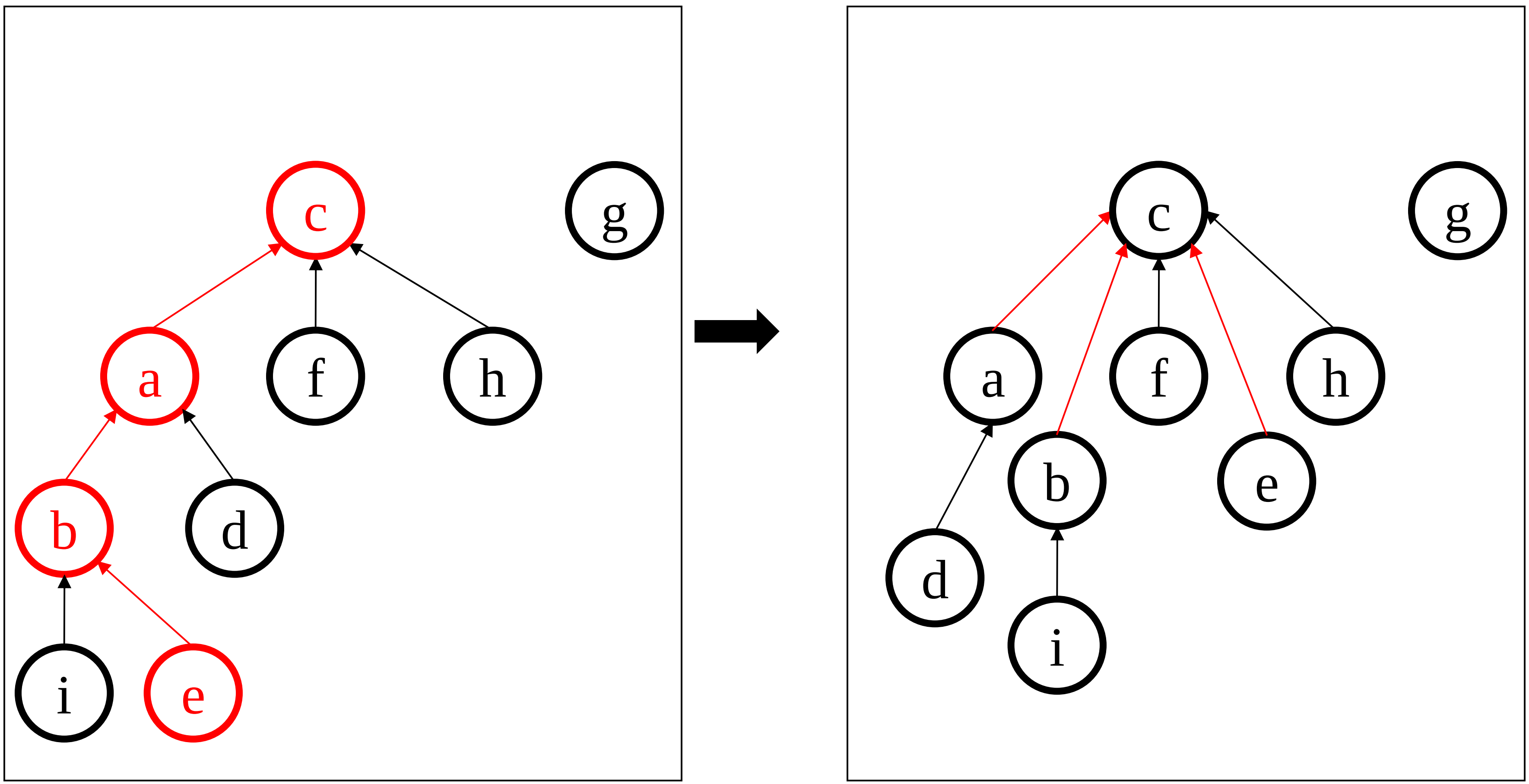
While we're finding ***e***,
could we do anything else?



Path compression!

Path Compression Example

find(e)



Path Compression Code

```
ID find(Object x) {
  assert(HashTable.contains(x));
  ID xID = HashTable[x];
  ID hold = xID;

  while(up[xID] != -1) {
    xID = up[xID];
  }
  while(up[hold] != -1) {
    temp = up[hold];
    up[hold] = xID;
    hold = temp;
  }
  return xID;
}
```

runtime:

Digression: Inverse Ackermann's

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

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

How fast does $\log^* n$ grow?

$\log^*(2) = 1$

$\log^*(4) = 2$

$\log^*(16) = 3$

$\log^*(65536) = 4$

$\log^*(2^{65536}) = 5$ (a 20,000 digit number!)

$\log^*(2^{2^{65536}}) = 6$

Complex Complexity of Weighted Union + Path Compression

- Tarjan (1984) proved that *m* weighted union and find operations with path commpression on a set of *n* elements have worst case complexity
  $O(m \cdot \log^*(n))$

  actually even a little better!

- For **all** practical purposes this is amortized constant time