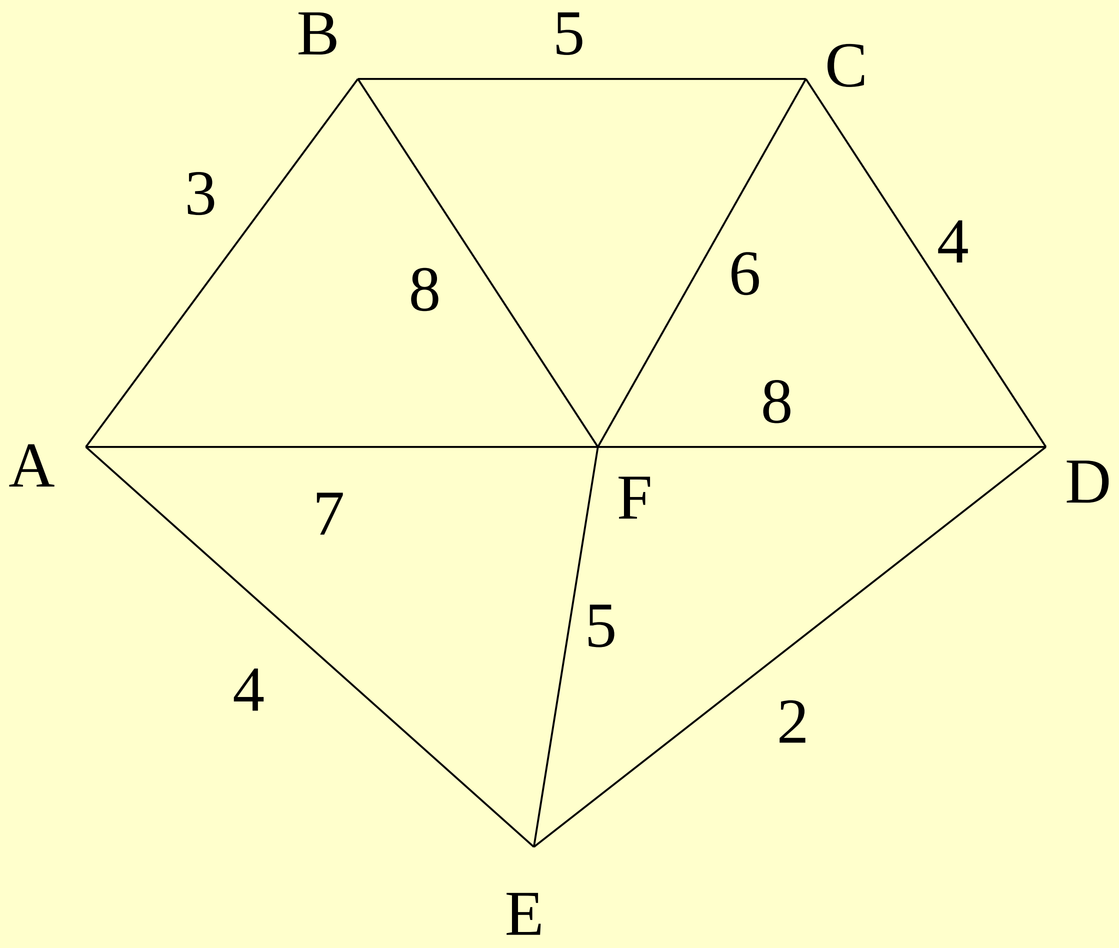
## Kruskal's algorithm

```
1    Sort the edges in an increasing order
2    T:={}
3    **while** E is not empty **do {**
3        take an edge (u, v) that is shortest in E
4        **if**  adding (u,v) does not form a cycle
                 add (u, v) to T
         }
```

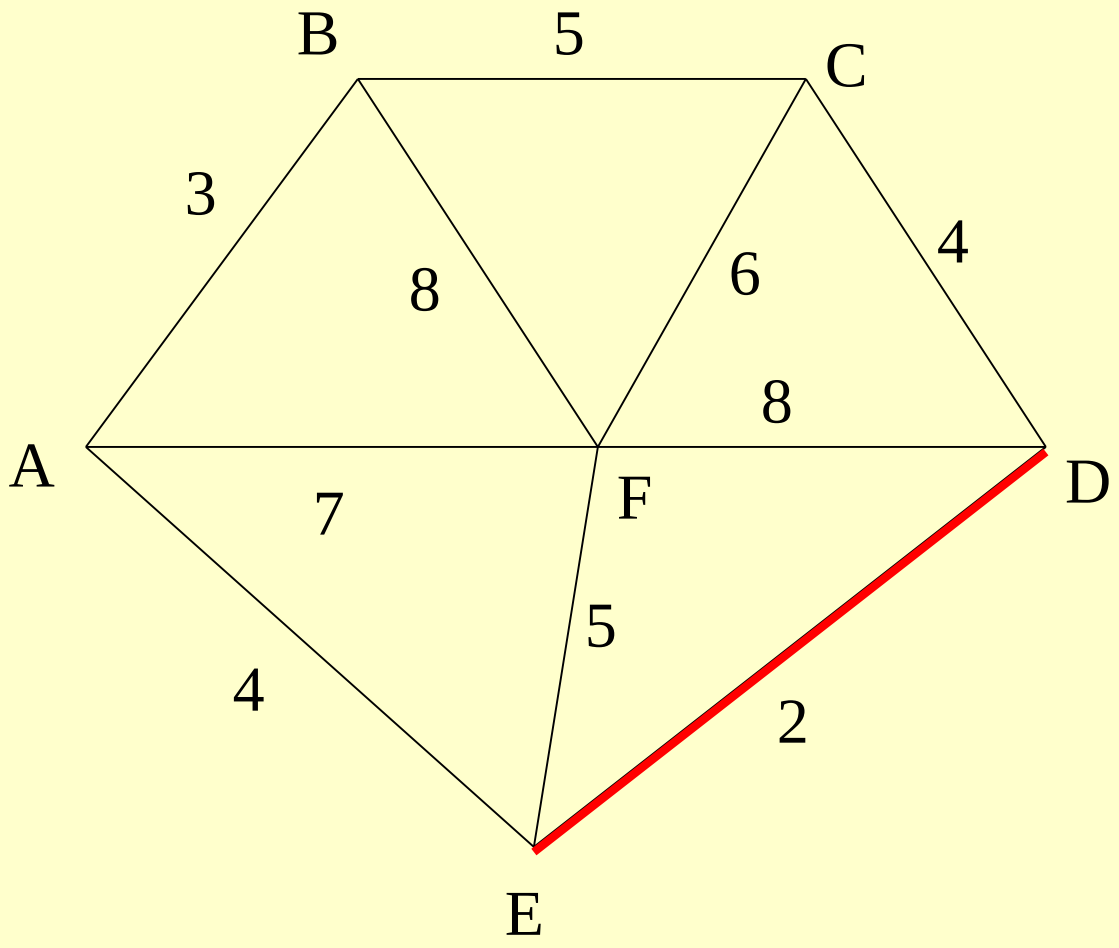
# Kruskal’s Algorithm



List the edges in order of size:

| | |
|---|---|
| ED | 2 |
| AB | 3 |
| AE | 4 |
| CD | 4 |
| BC | 5 |
| EF | 5 |
| CF | 6 |
| AF | 7 |
| BF | 8 |
| CF | 8 |

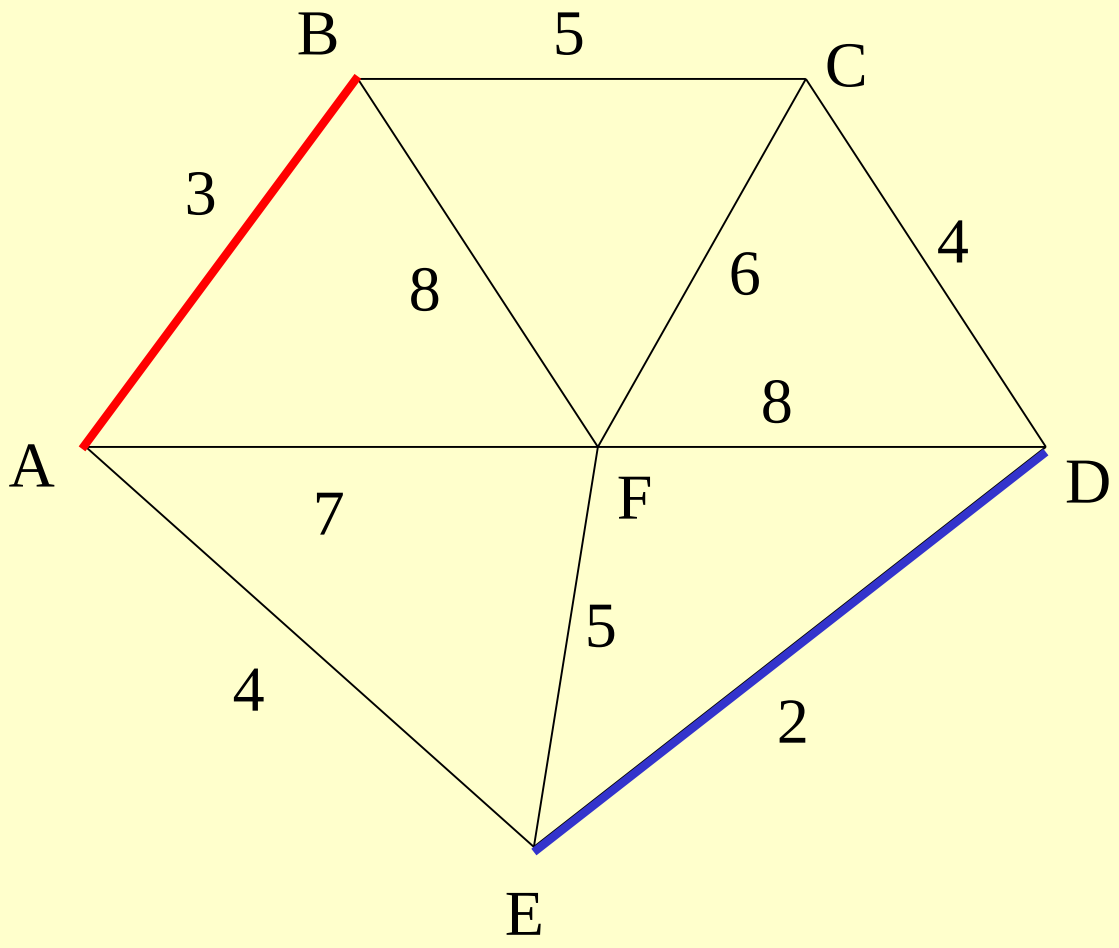
## Kruskal’s Algorithm



Select the shortest edge in the network

**ED 2**

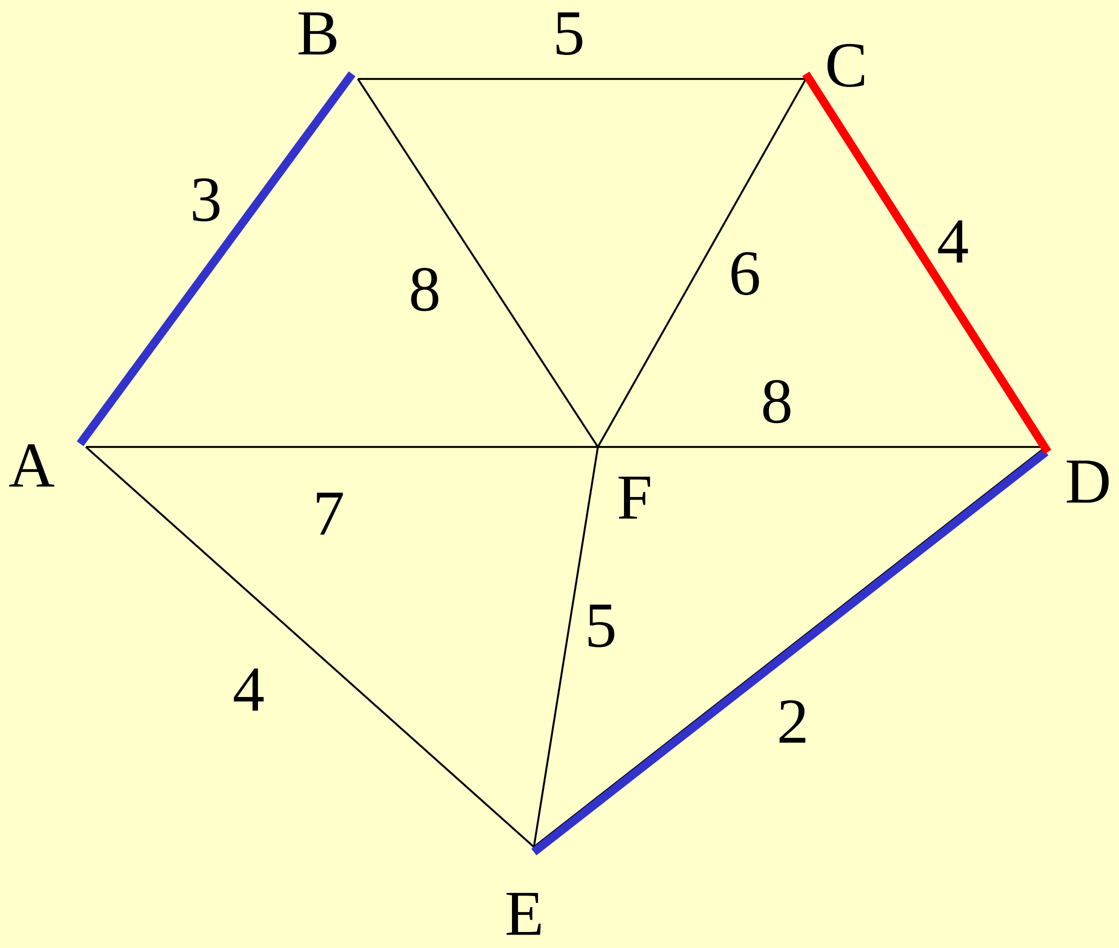
# Kruskal's Algorithm



Select the next shortest edge which does not create a cycle

**ED 2**
**AB 3**

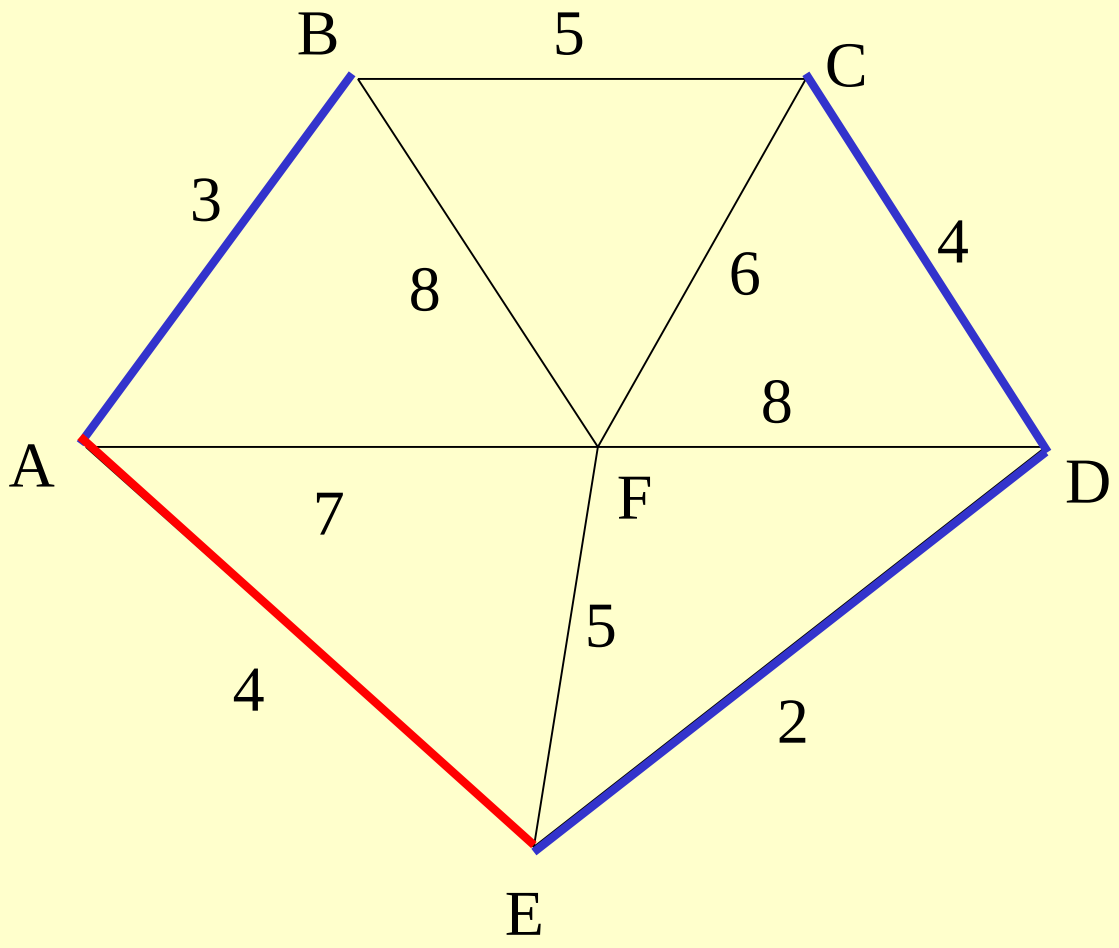
# Kruskal's Algorithm



Select the next shortest edge which does not create a cycle

**ED 2**
**AB 3**
**CD 4 (or AE 4)**

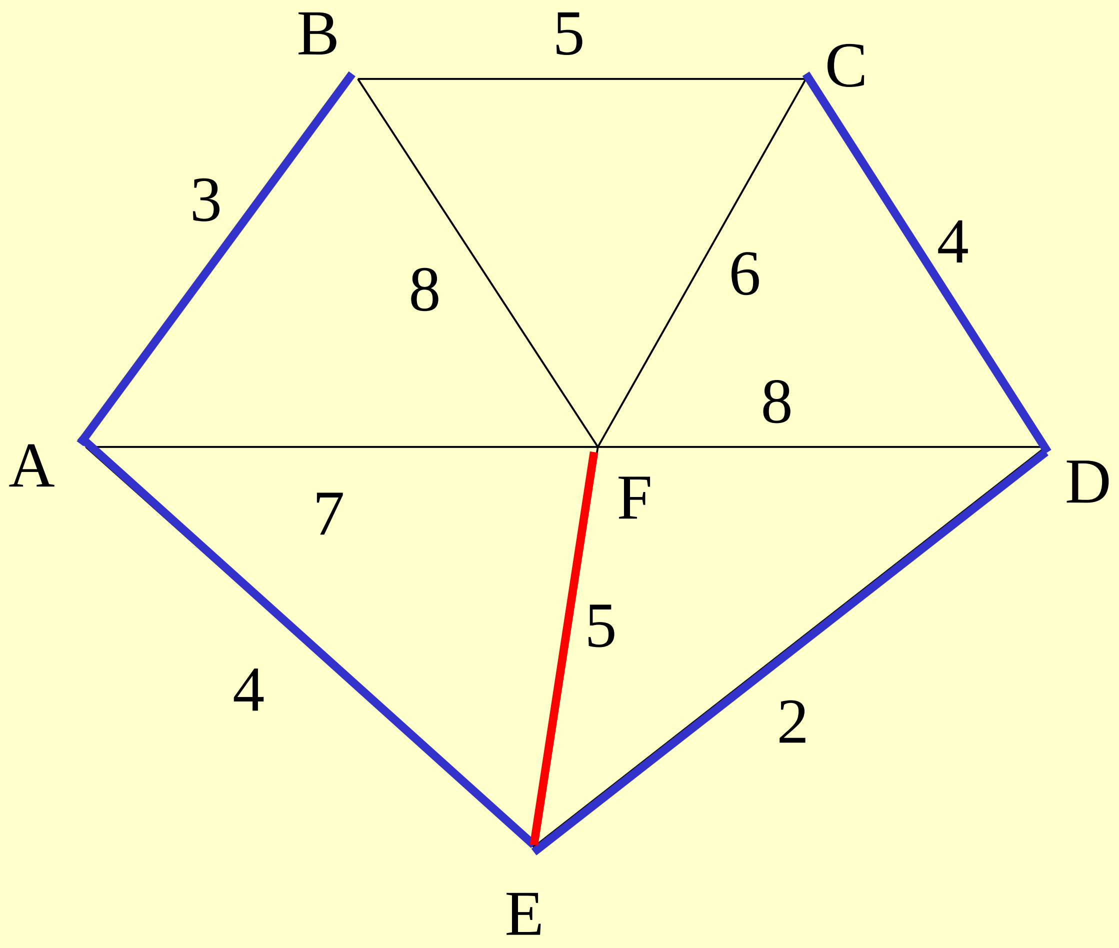
# Kruskal’s Algorithm



Select the next shortest edge which does not create a cycle

**ED 2**
**AB 3**
**CD 4**
**AE 4**

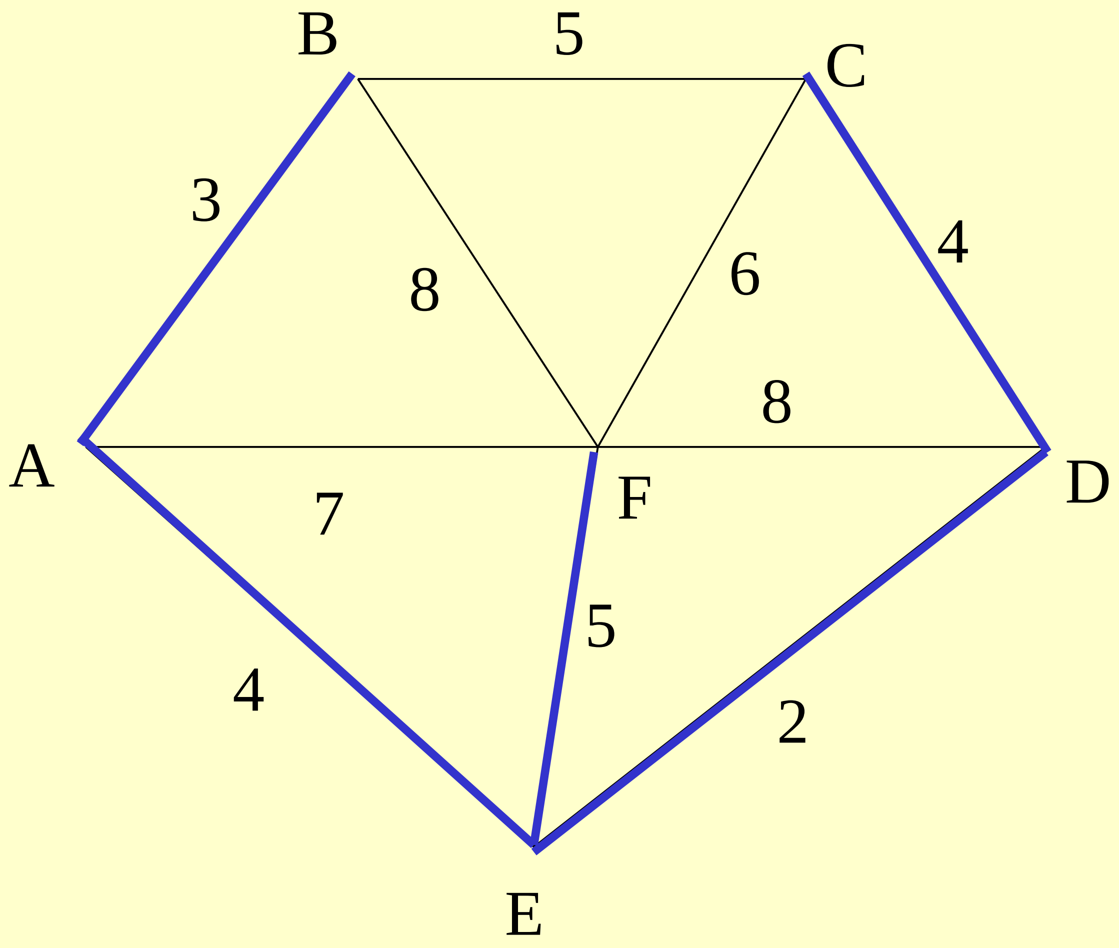
# Kruskal's Algorithm



Select the next shortest edge which does not create a cycle

**ED 2**
**AB 3**
**CD 4**
**AE 4**
**BC 5 – forms a cycle**
**EF 5**

## Kruskal’s Algorithm



All vertices have been connected.

The solution is

**ED 2**
**AB 3**
**CD 4**
**AE 4**
**EF 5**

Total weight of tree: 18

## Proof

Let T be the MST found by Kruskal.
Let T* be a MST in the graph.

Let e be an edge present in T* but not in T (since T and T* differ.)
T U {e} will contain a cycle C.
In cycle C, wt(e) > wt(k) for all edges k in the cycle.
(since else kruskal would have picked e)
Also, there exists an edge f in T that is not present in T*
(else T* would contain a cycle)
Let T1=T U {e} –{f}
Then, wt(T1) >=wt(T)
Similarly pick the next edge e1 present in T* not in T and f1 present in the cycle formed by T U{e1} and not in T*.
Let T2= T1 U {e1} – {f1}
Wt(T2)>= wt (T1) >= wt(T)
Wt(T*)>=.........>=Wt(T2)>= wt (T1) >= wt(T)
Thus, wt(T*) >= wt(T)

## Some points to note

- Both algorithms will always give solutions with the same length.
- They will usually select edges in a different order – you must show this in your workings.
- Occasionally they will use different edges – this may happen when you have to choose between edges with the same length. In this case there is more than one minimum connector for the network.

How do you check whether the edge added by kruskal
does not create a cycle?

Will color vertices used so far as grey and checking
for the end points of the new selected edge
being grey solve the problem?
No. Say you have so far created two components
C1 and C2 each without any cycles hence all points on C1 and C2
are grey. Now if we add an edge between a point in C1 and one in
c2 we do not form a cycle but are joining two grey egdes. So
joining grey edges does not imply that there is a cycle.

You might need to join two or more subtrees into a tree !!

The union find data structure will allow us to identify if the Two end vertices of an edge belong to the same connected component or not.

An edge is discarded if the two end points are already in the same connected component and included otherwise.

# CSE 326: Data Structu
# Part 7:
# The Dynamic (Equivalence) Duo: Weighted Union & Path Compression

Henry Kautz

Autumn Quarter 2002

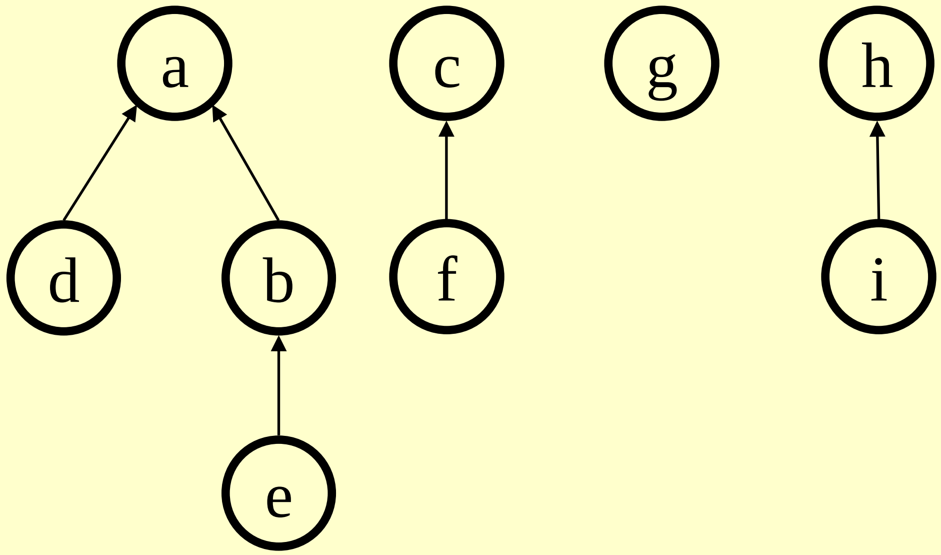
## Up-Tree Intuition

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

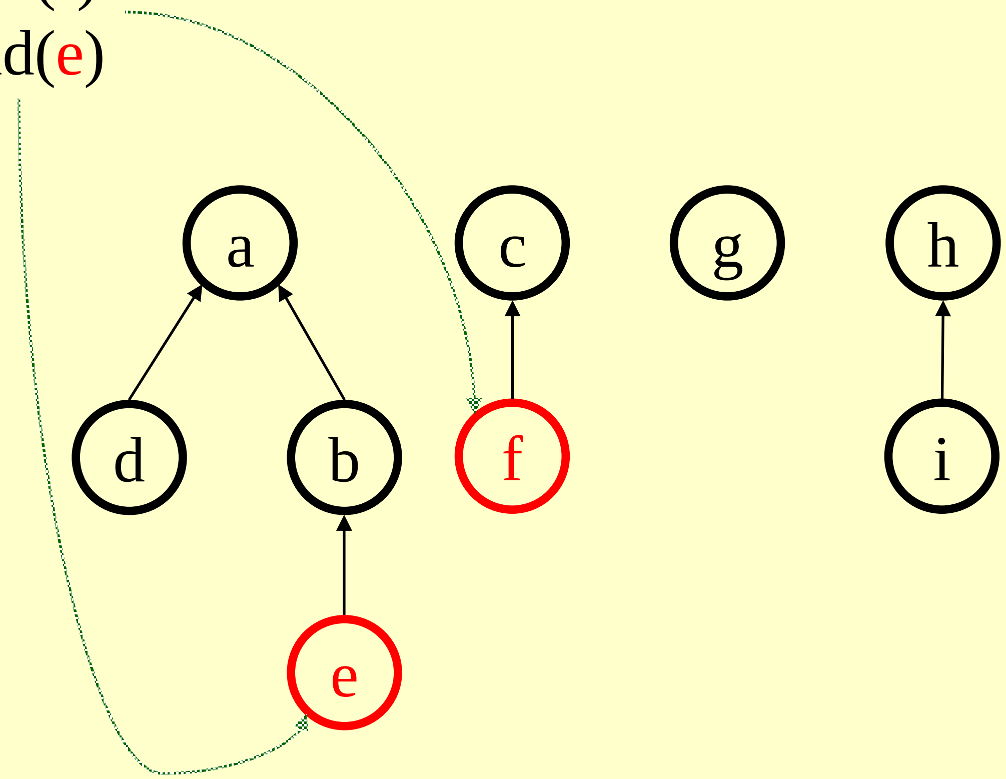
- Each subset is an up-tree with its root as its representative member
- All members of a given set are nodes in that set’s up-tree
- Hash table maps input data to the node associated with



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

# Find

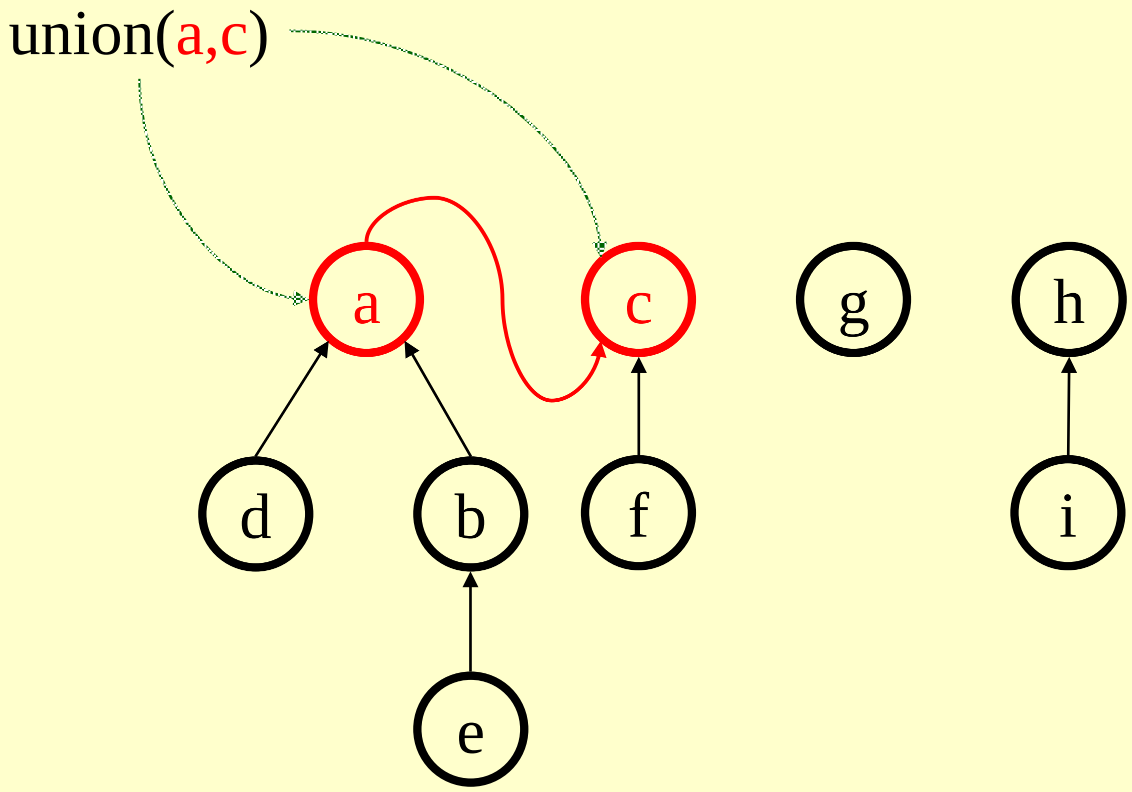
find(f)
find(e)



runtime:

Just traverse to the root!

# Union

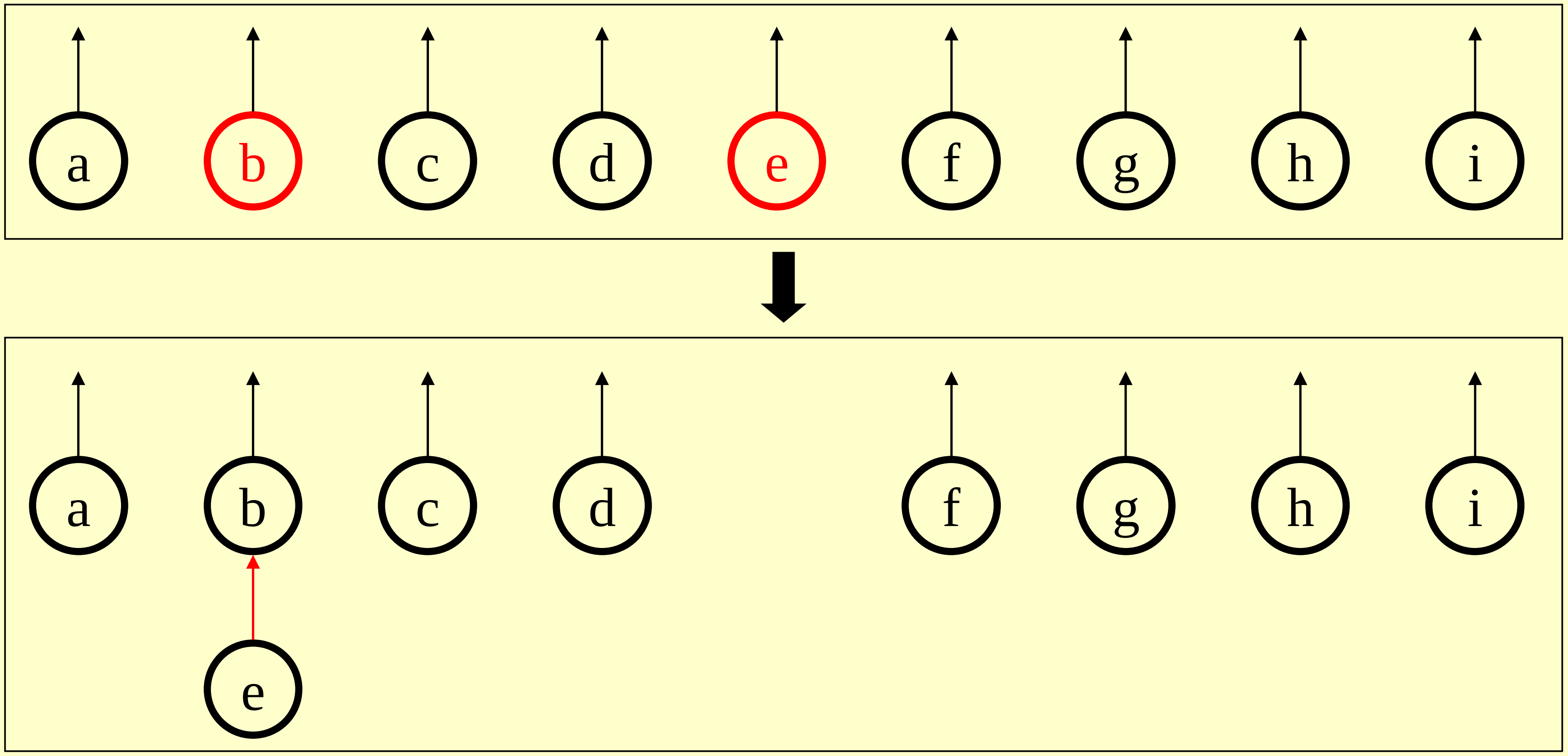


runtime:

Just hang one root from the other!

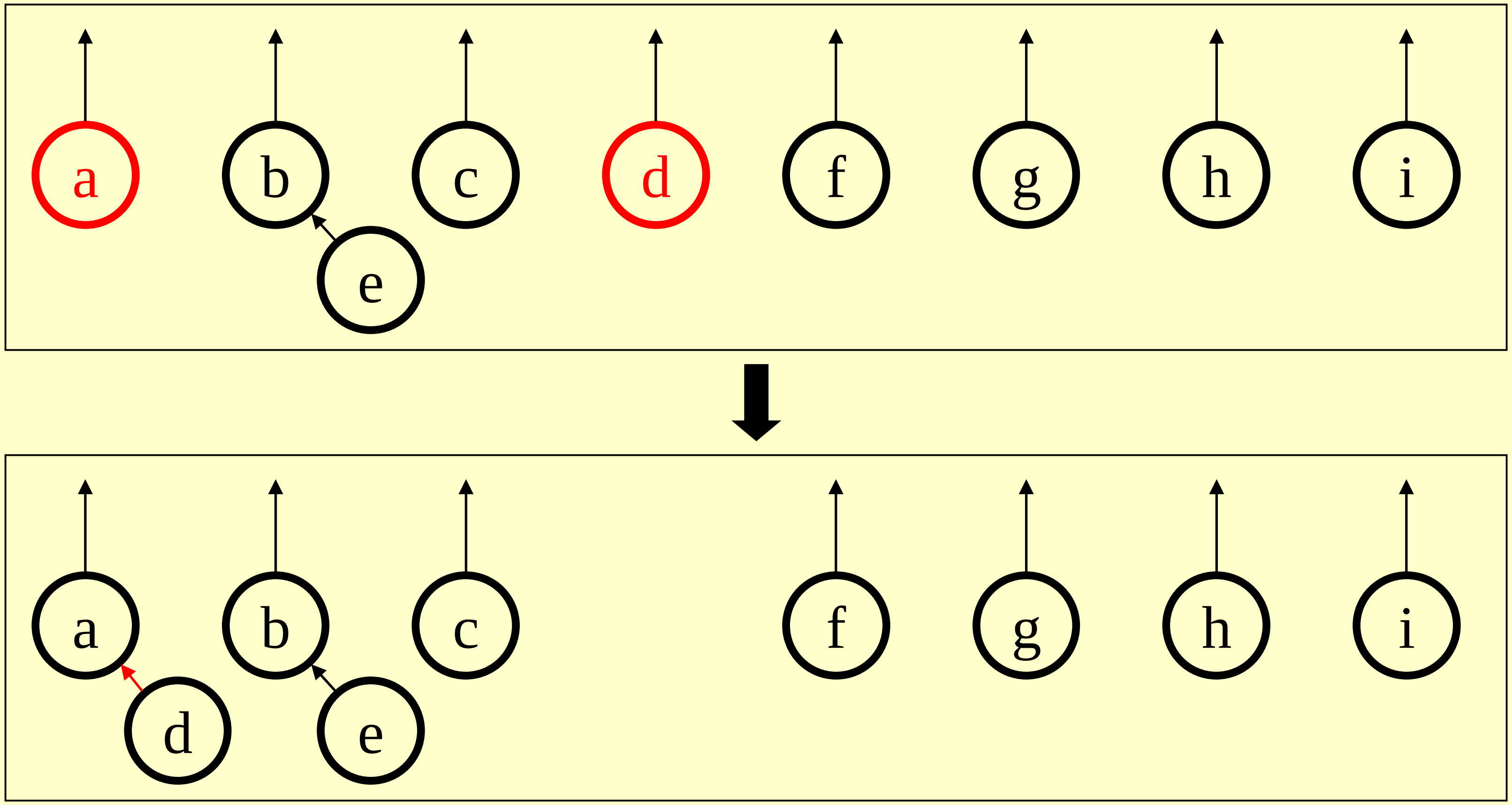
# The Whole Example (1/11)

union(b,e)



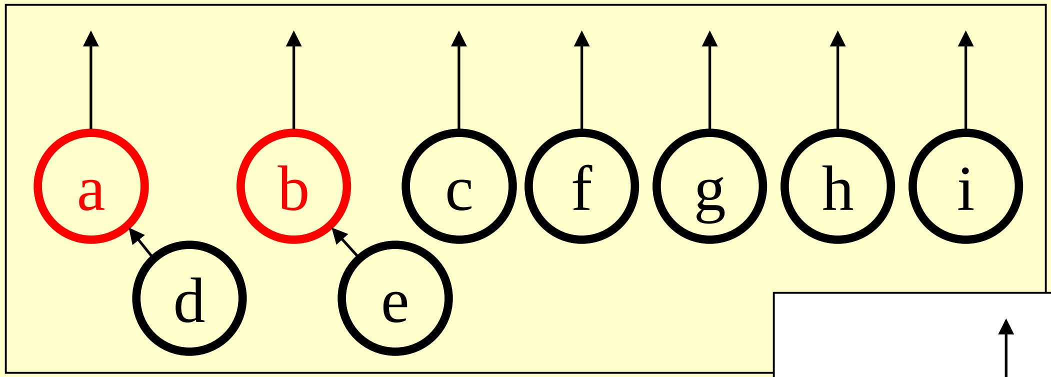
# The Whole Example (2/11)

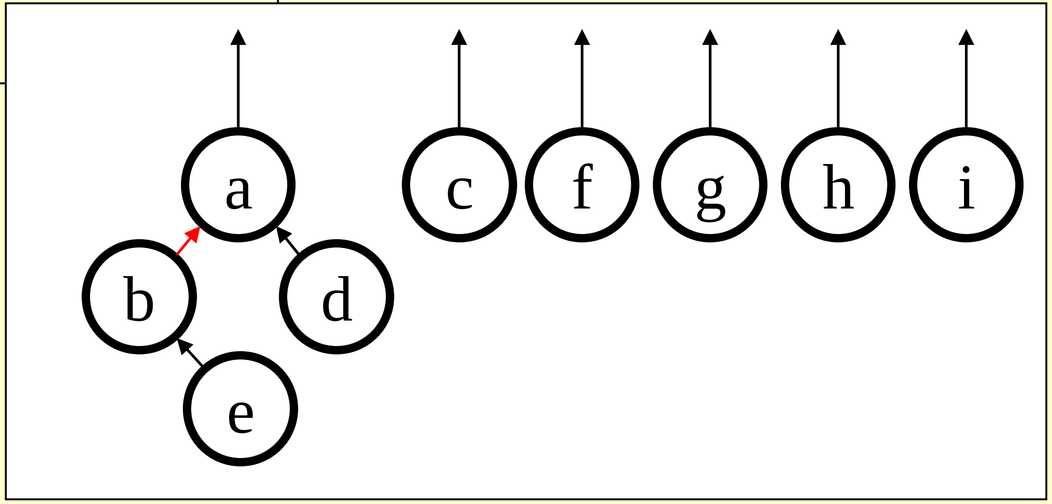
union(a,d)



# The Whole Example (3/11)

union(a,b)

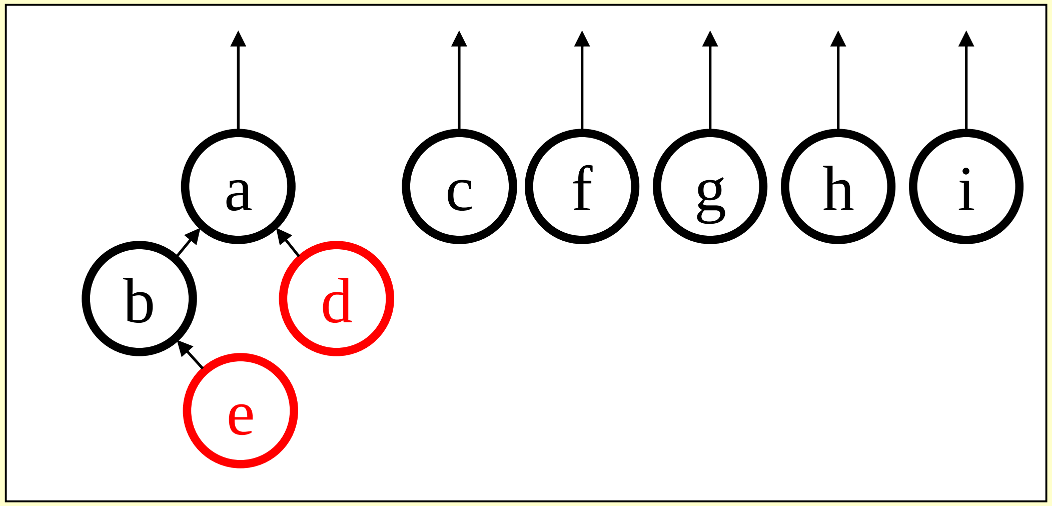




# The Whole Example (4/11)

find(d) = find(e)

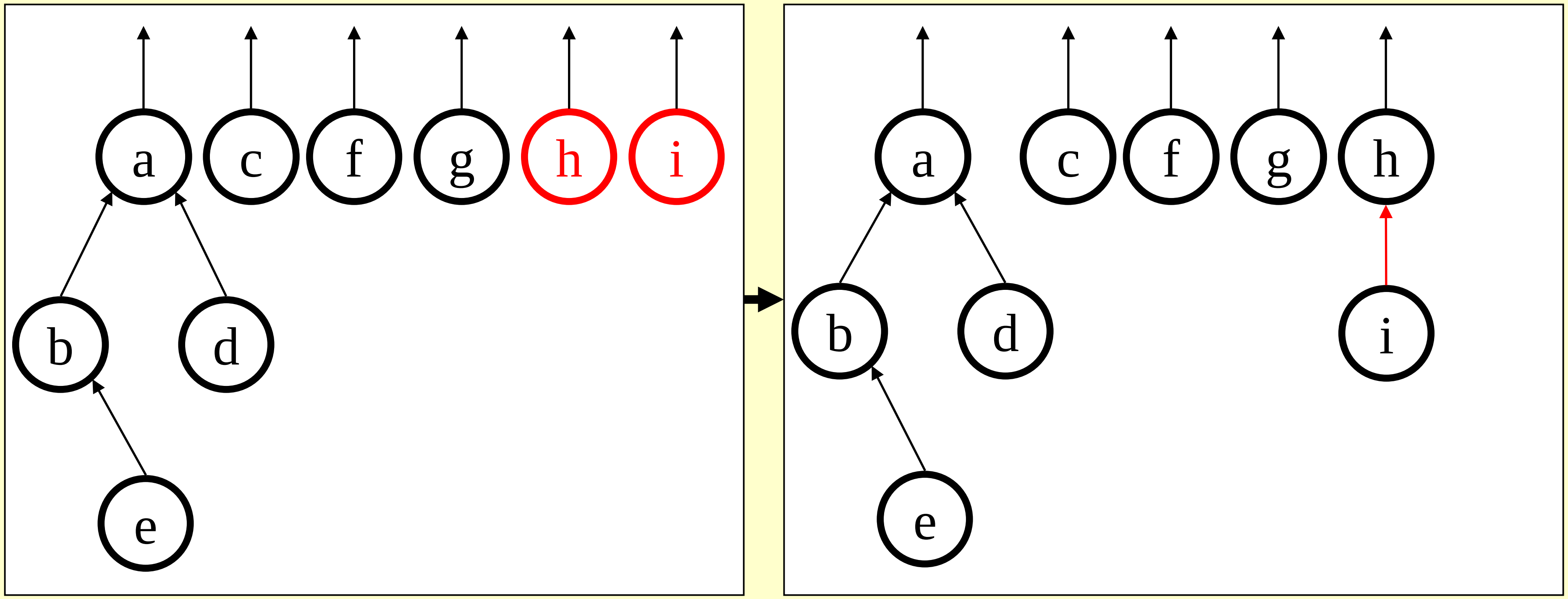
No union!



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

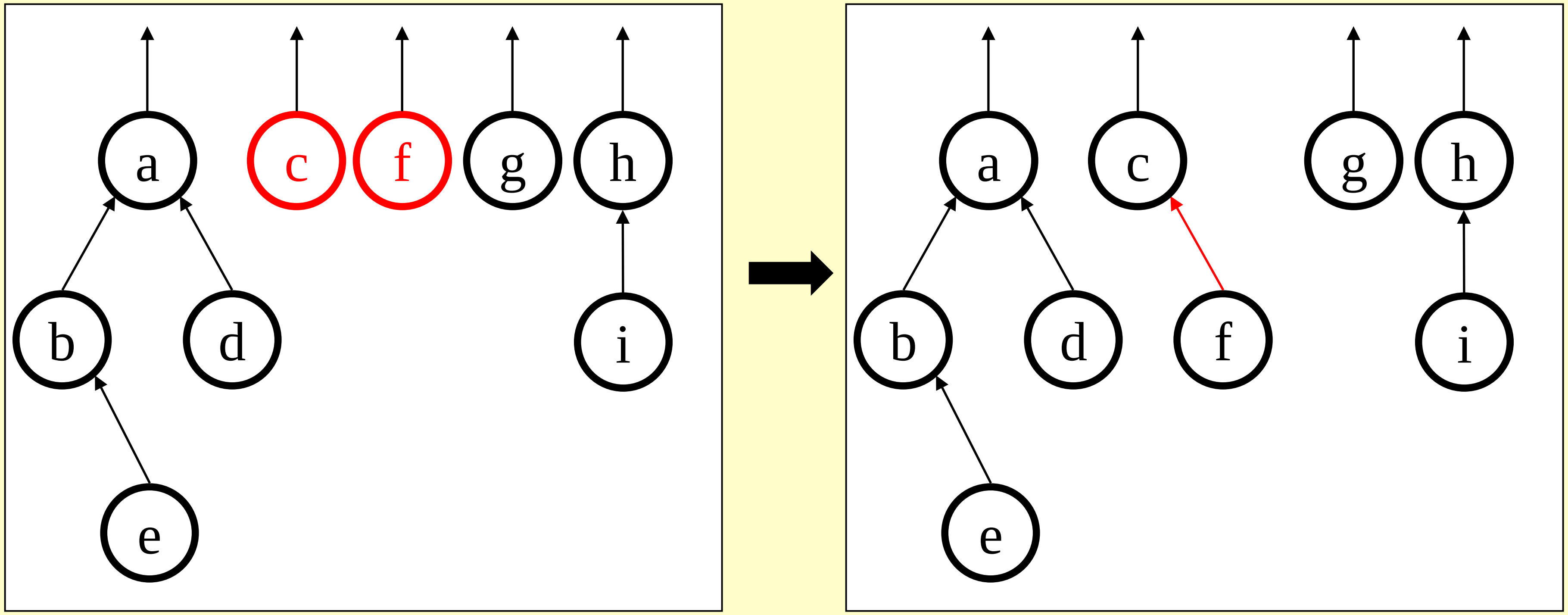
# The Whole Example (5/11)

union(h,i)



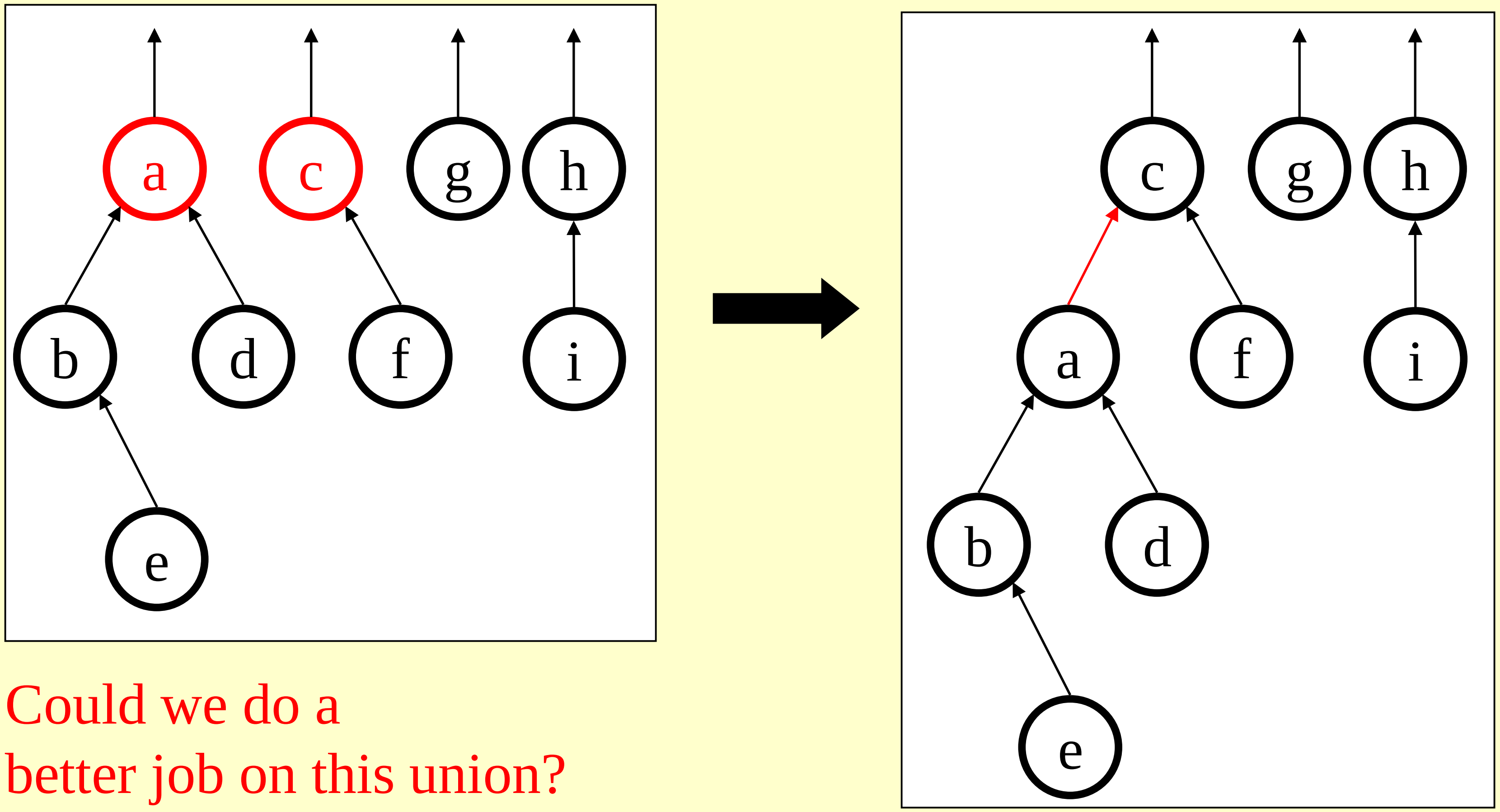
# The Whole Example (6/11)

union(c,f)



# The Whole Example (7/11)

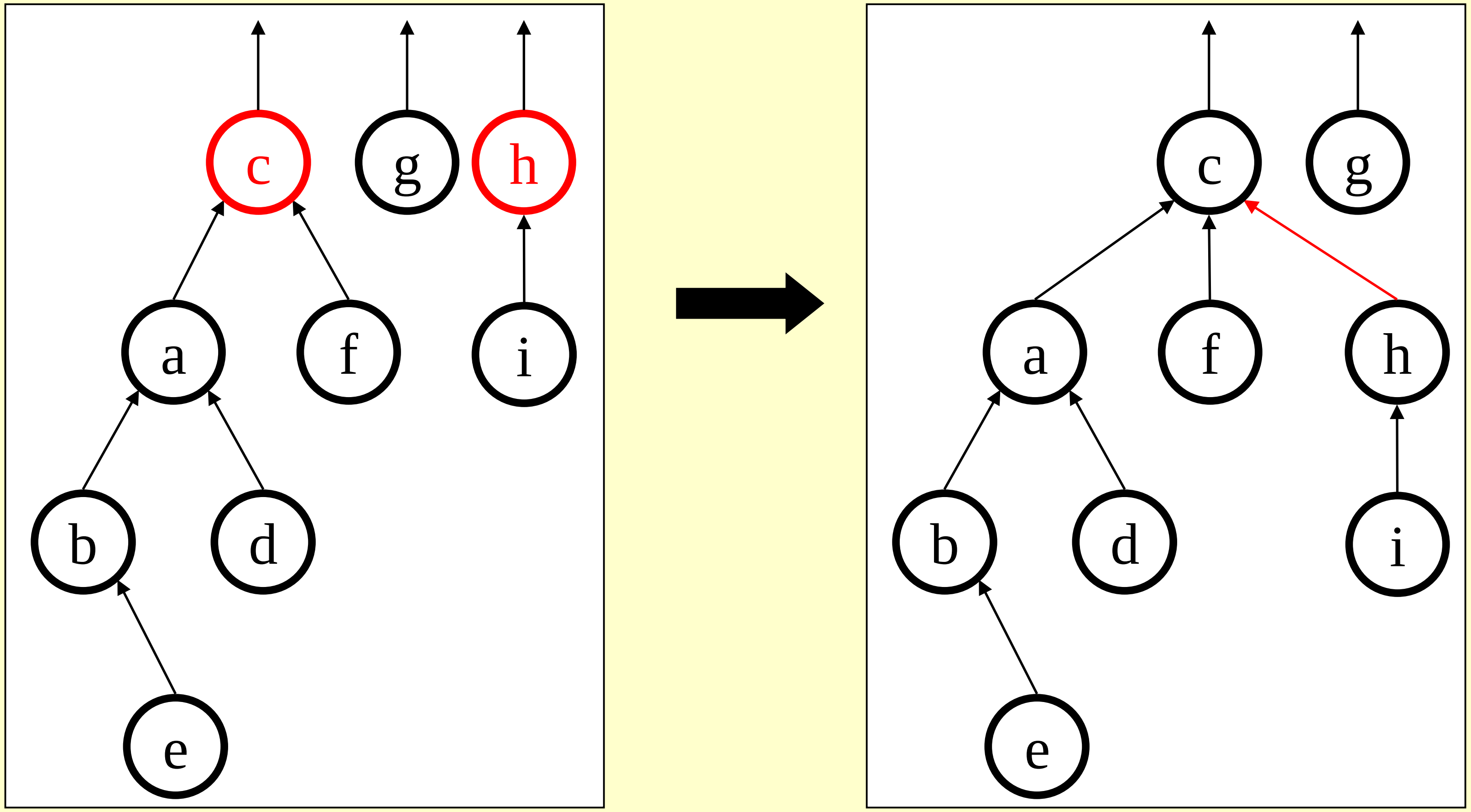
find(e)
find(f)
union(c,a)



Could we do a
better job on this union?

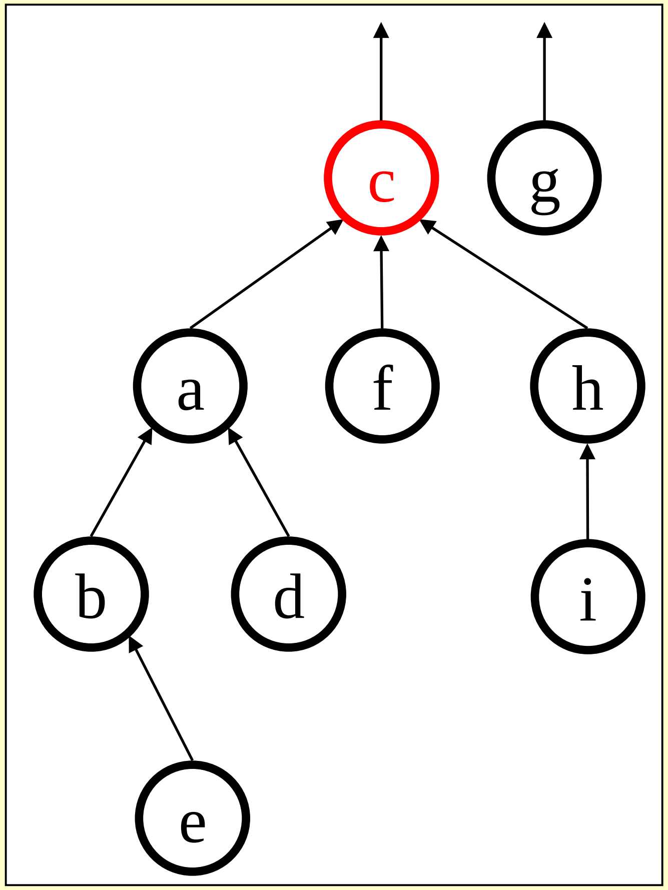
# The Whole Example (8/11)

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.

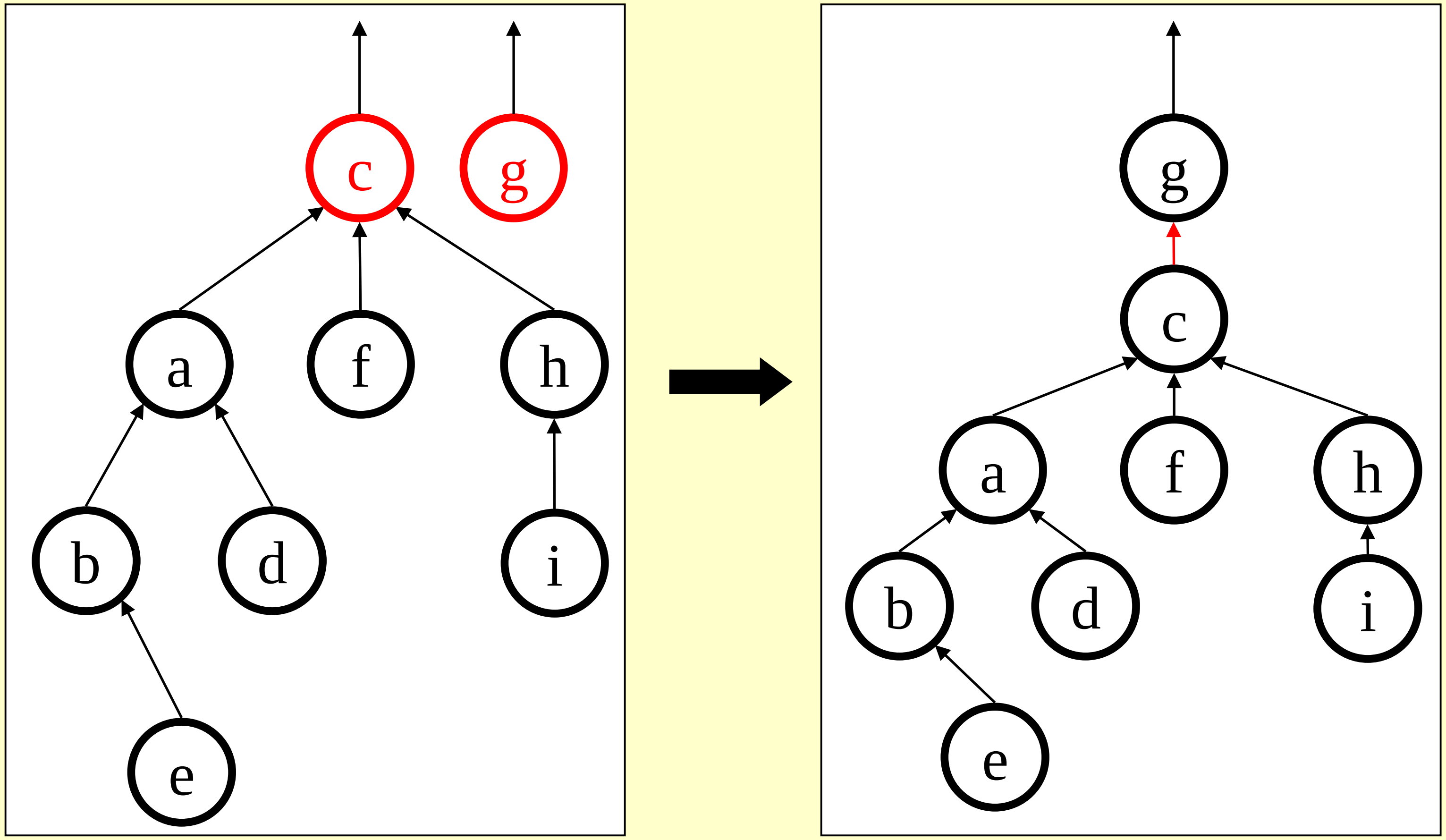


# The Whole Example (10/11)

find(d)

find(g)

union(g, c)

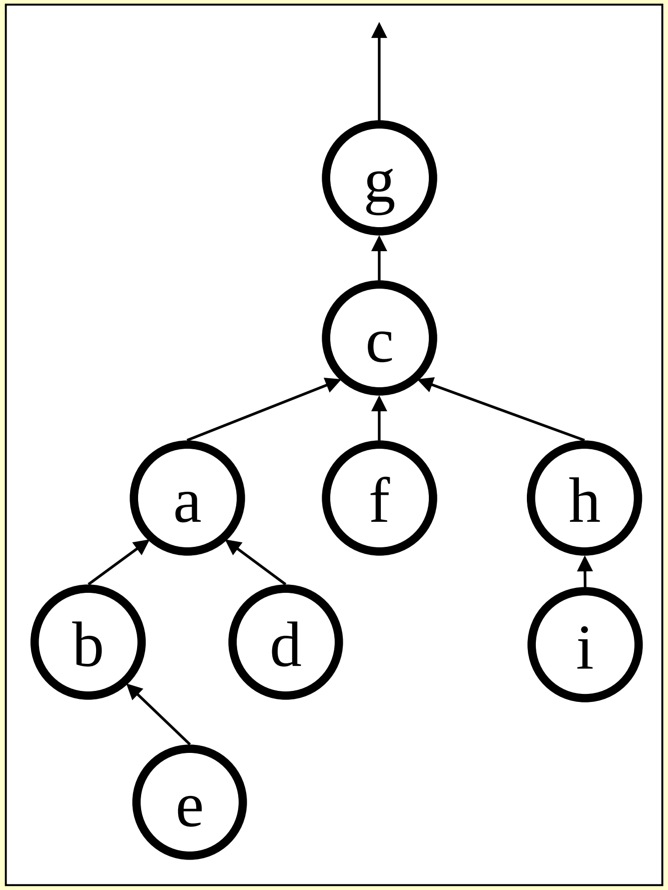


# The Whole Example (11/11)

find(g) = find(h)

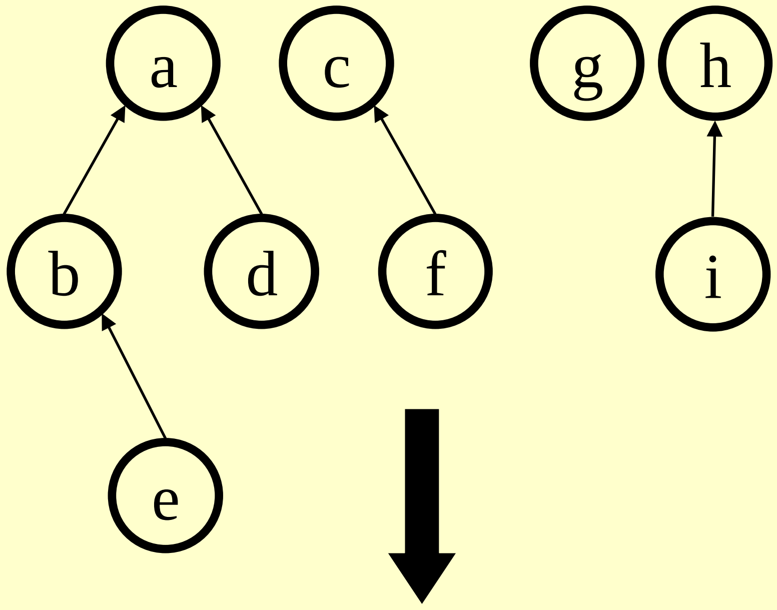
So, no union.

And, we're done!



# 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



| | 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

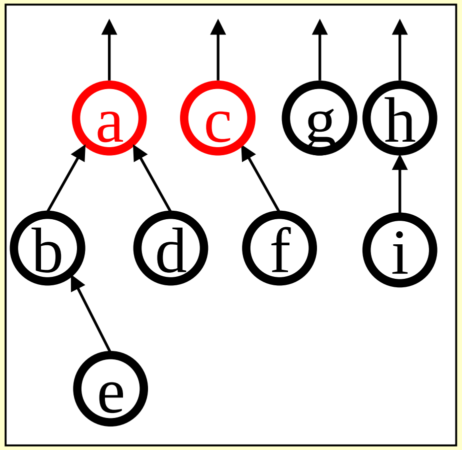
```
typedef ID int;
ID up[10000];

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)

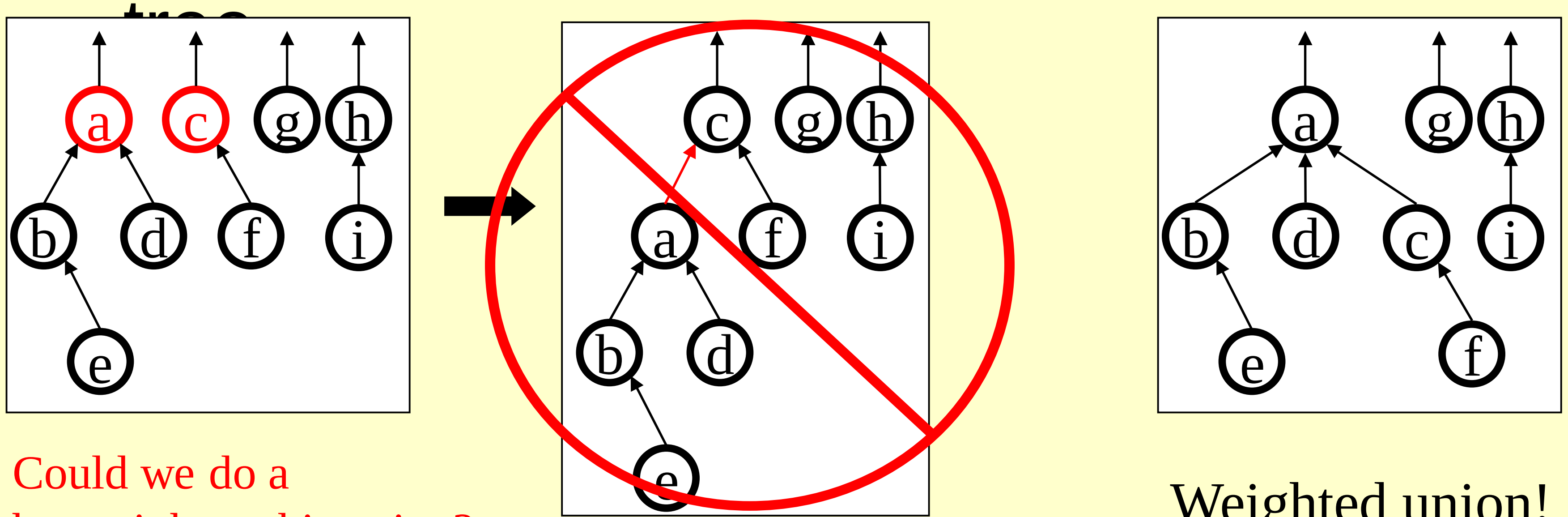
# Room for Improvement?



Find → O(n)
Union → O(1)
Could we do a better job ?

# 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

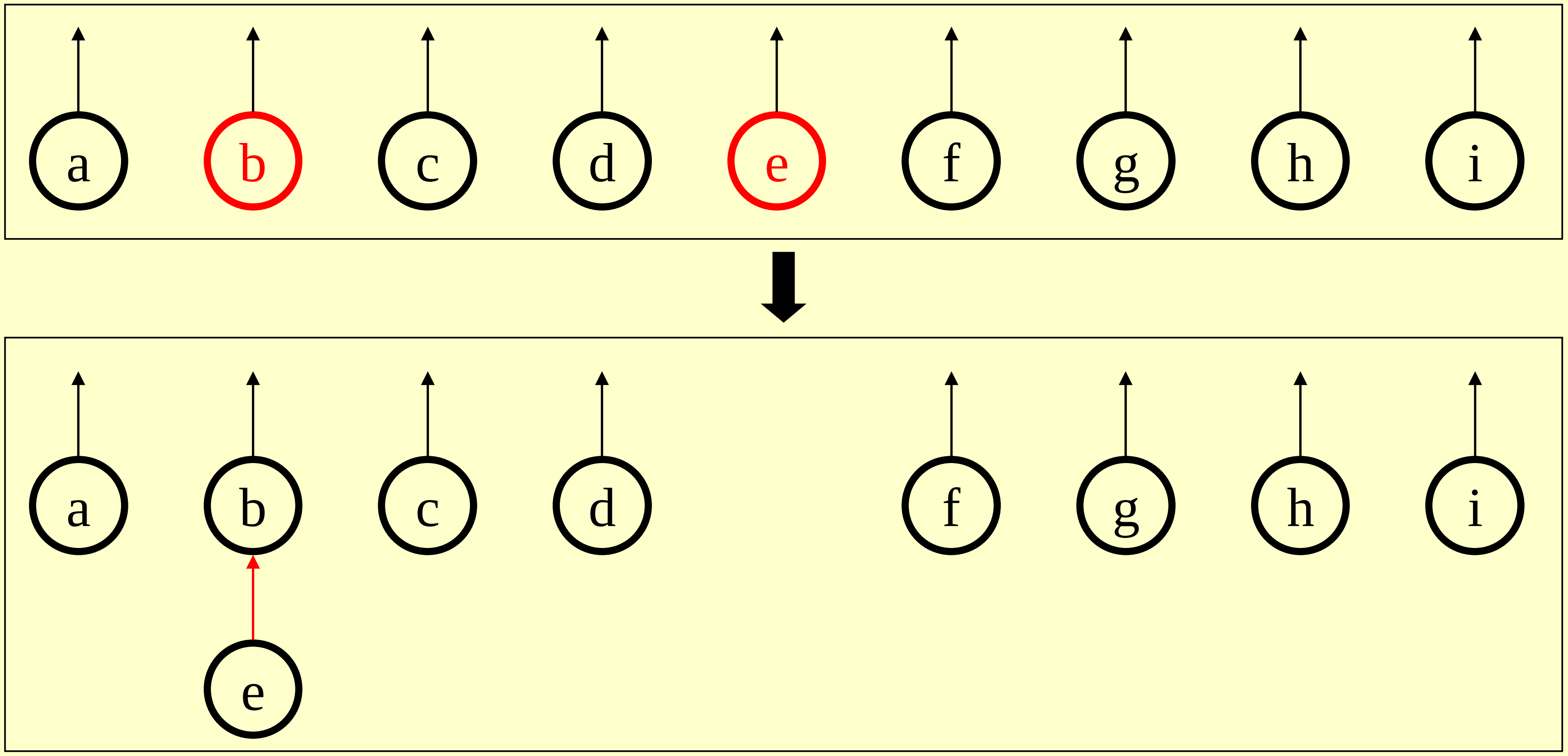


Could we do a better job on this union?

Weighted union!

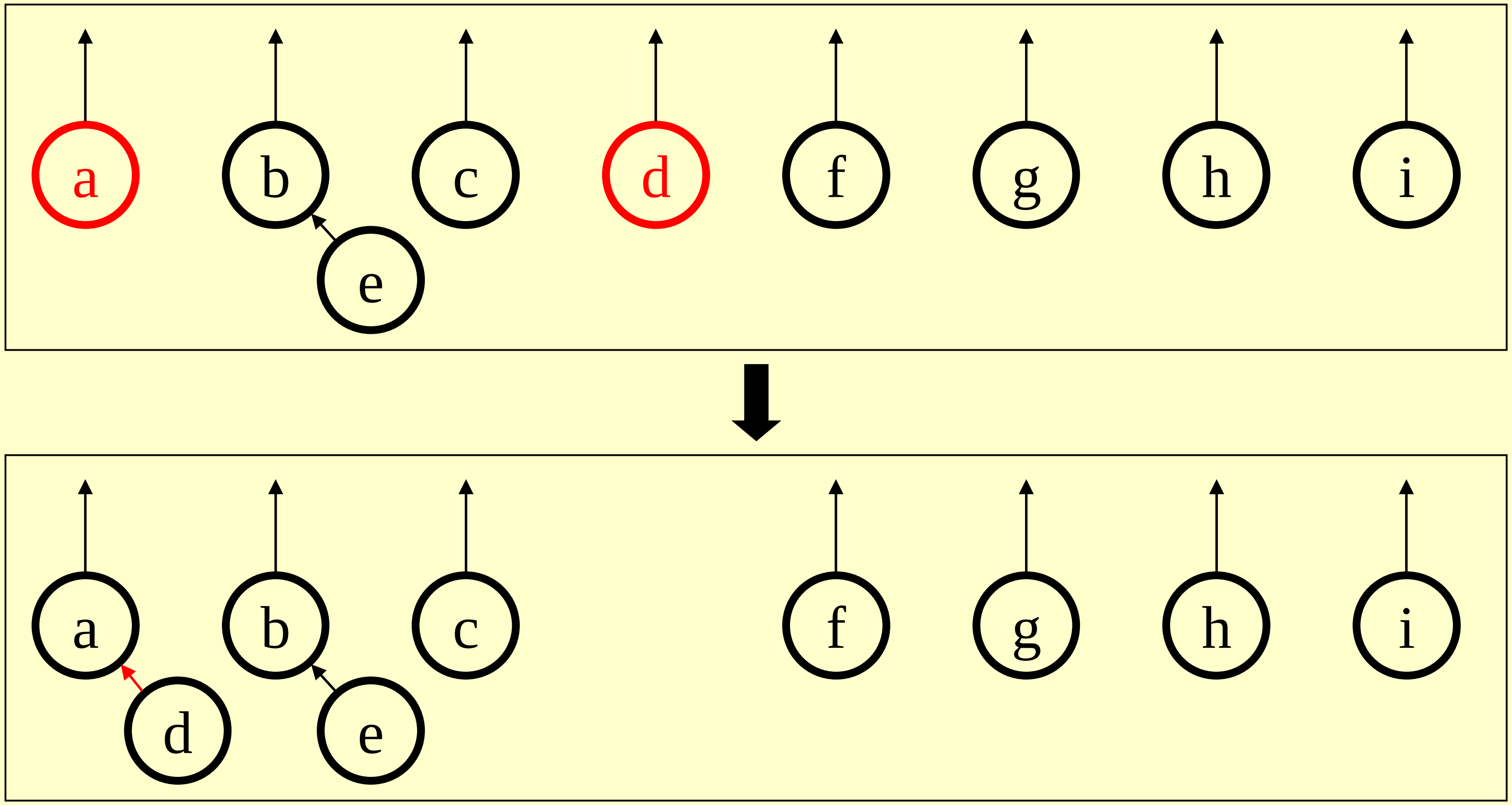
# The Whole Example (1/11)

union(b,e)



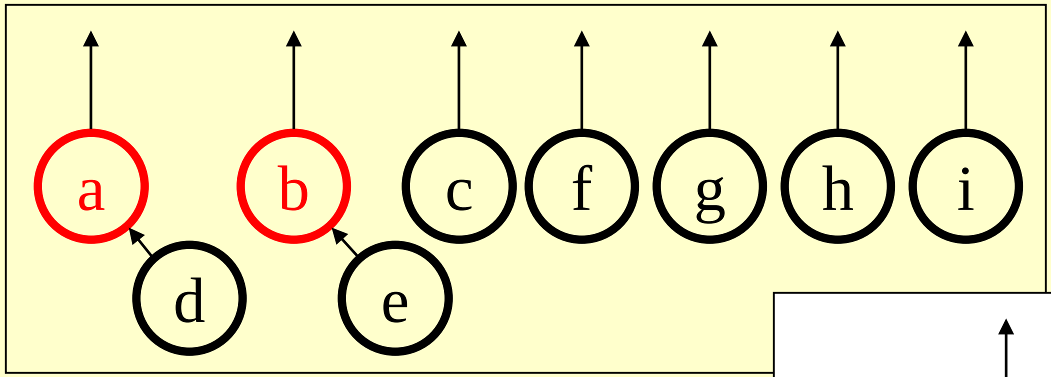
# The Whole Example (2/11)

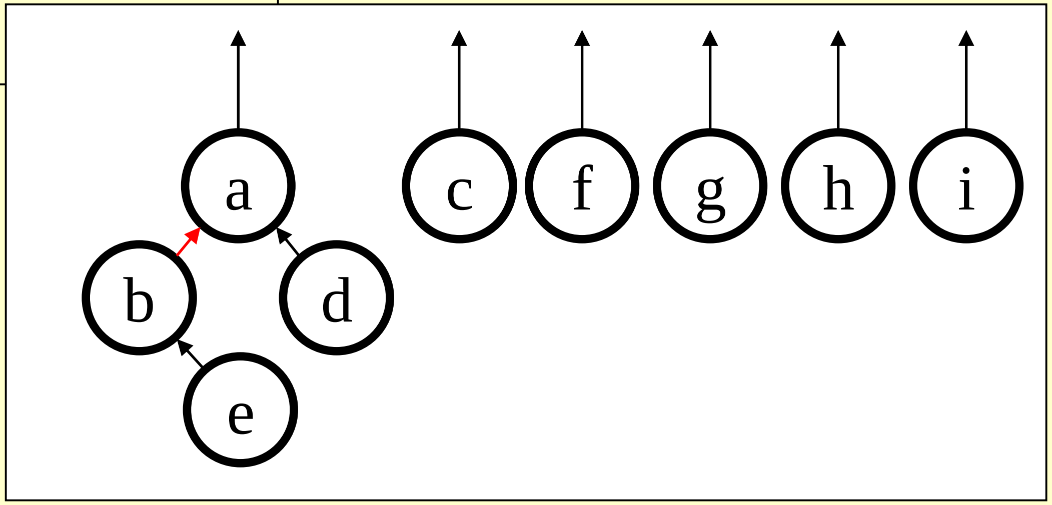
union(a,d)



# The Whole Example (3/11)

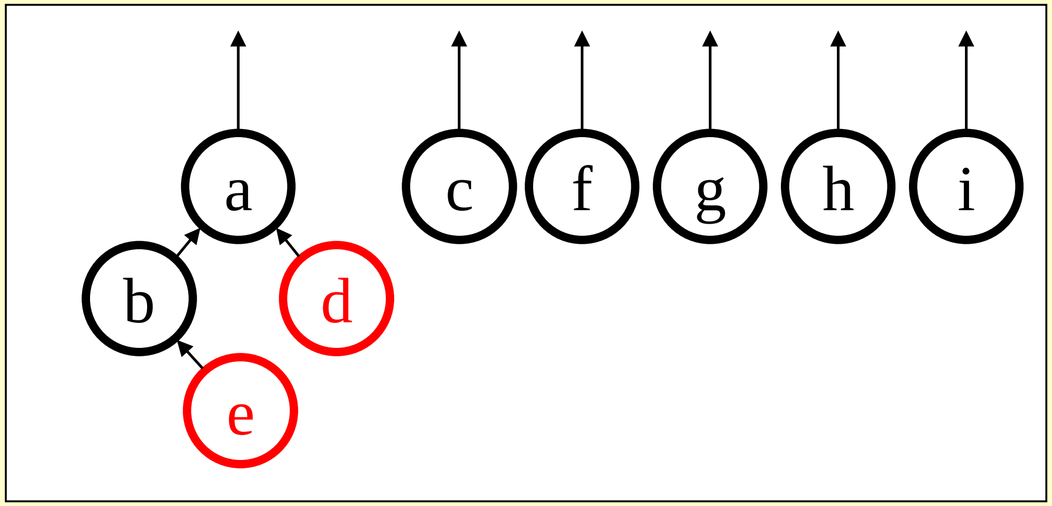
union(a,b)





# The Whole Example (4/11)

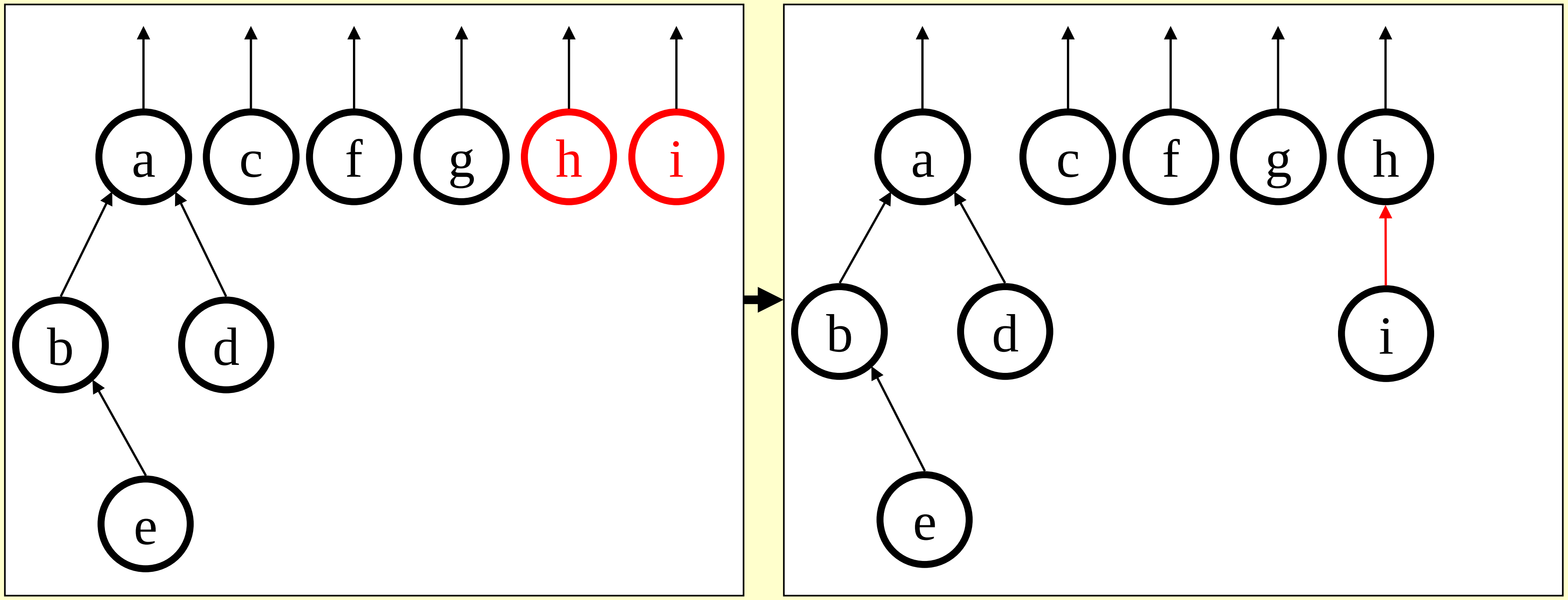
find(d) = find(e)
No union!



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

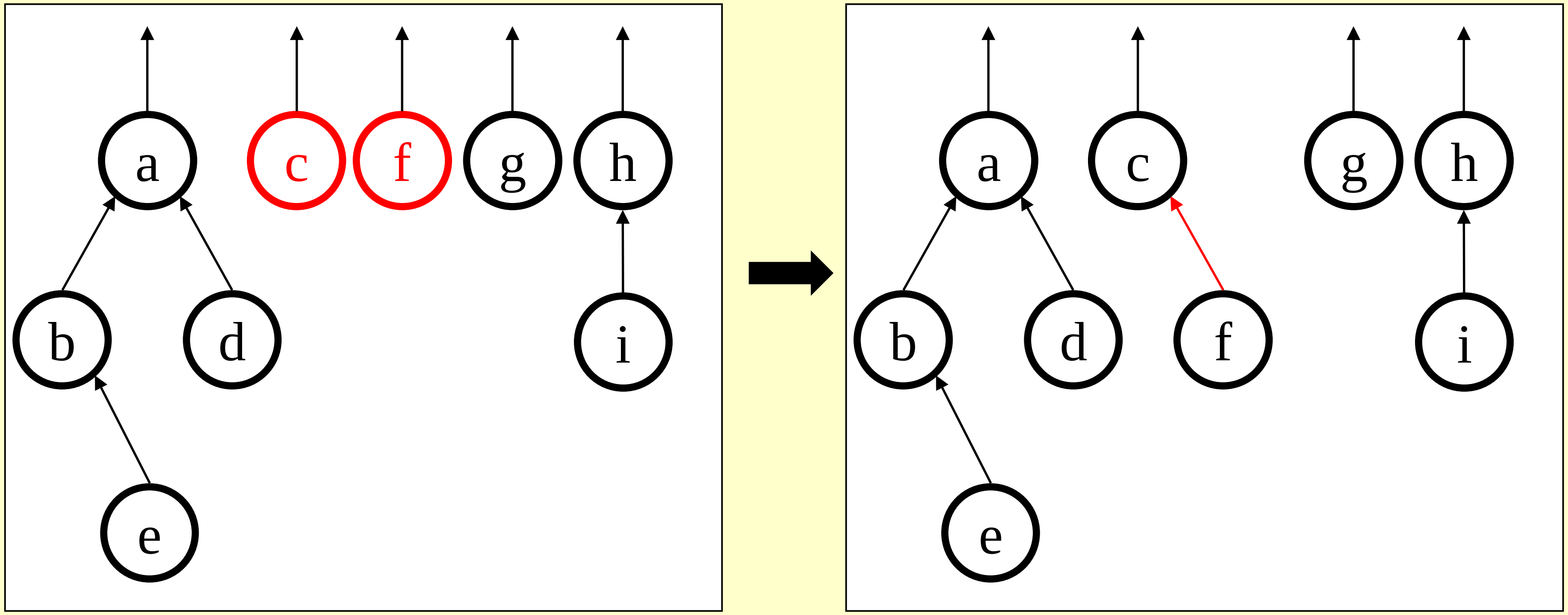
# The Whole Example (5/11)

union(h,i)



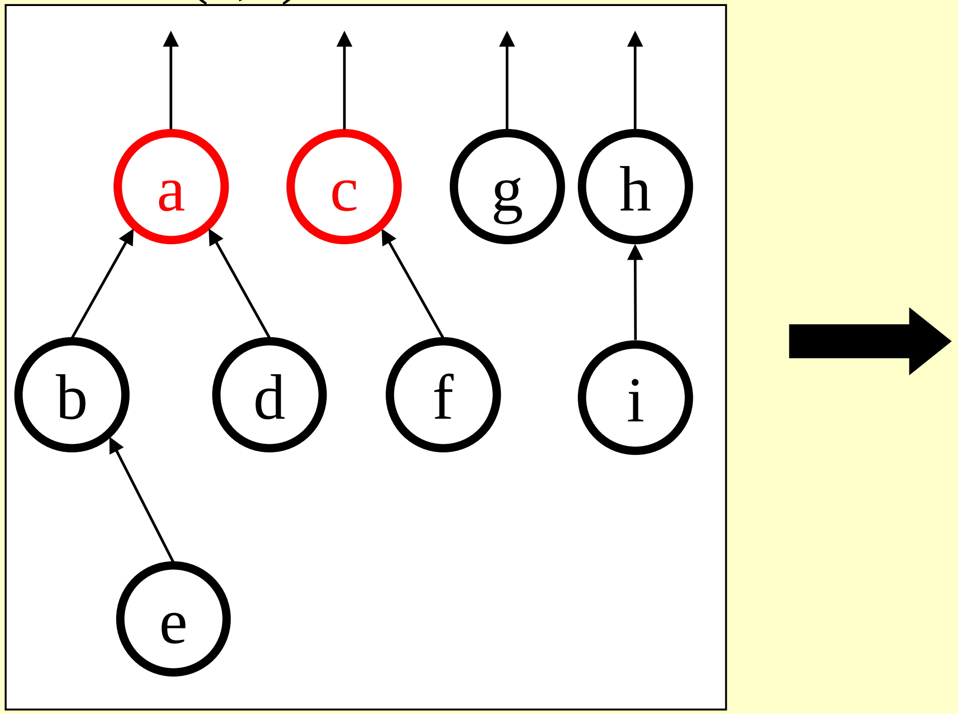
# The Whole Example (6/11)

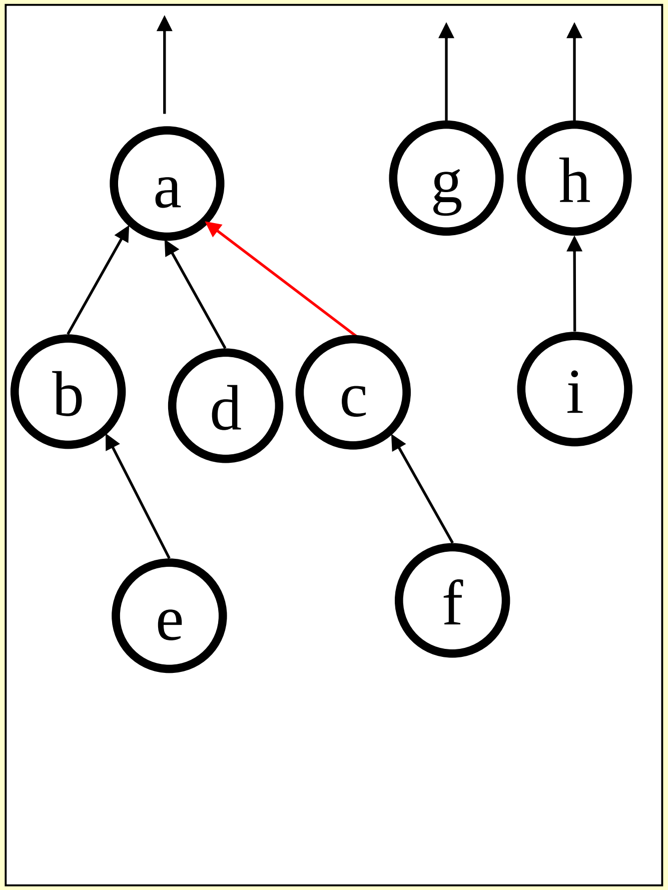
union(c,f)



# The Whole Example (7/11)

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

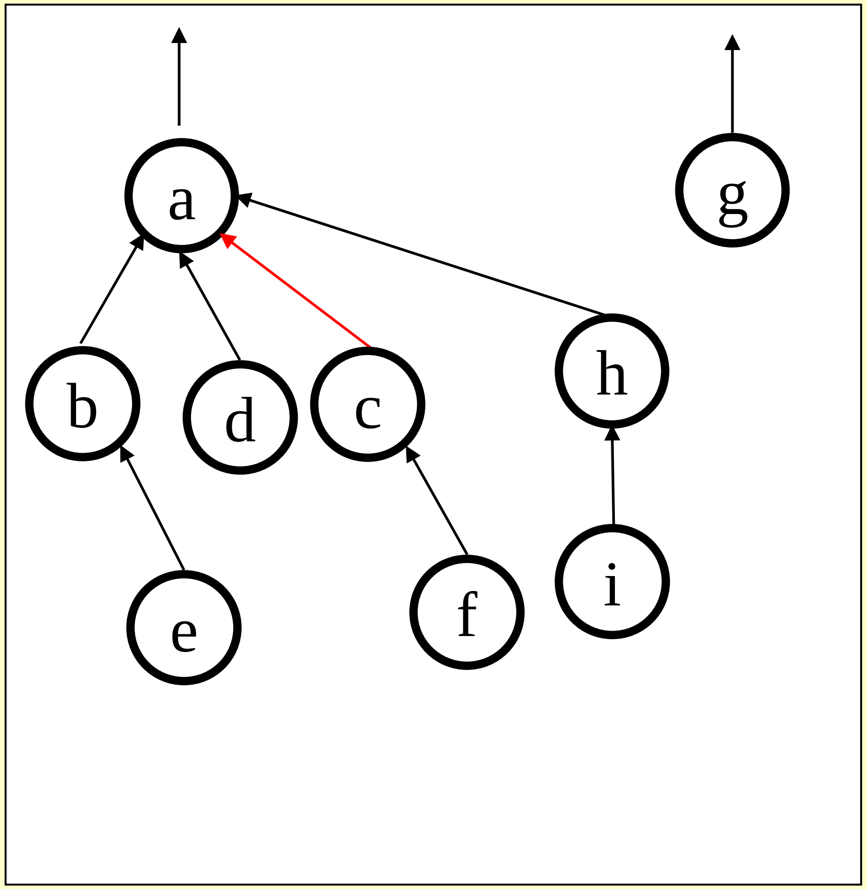




Could we do a
better job on this union?

# The Whole Example (8/11)

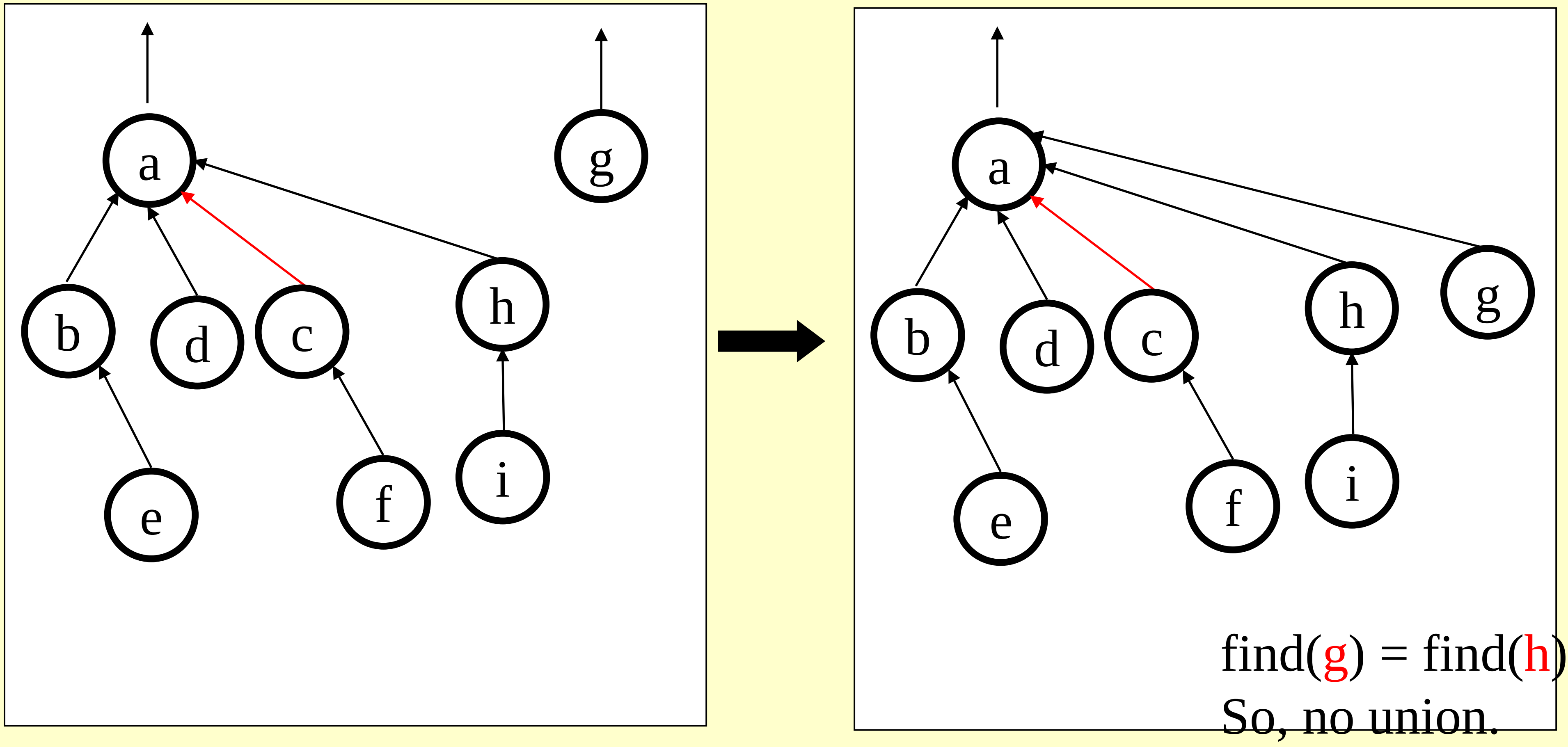
find(f)
find(i)
union(c,h)



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

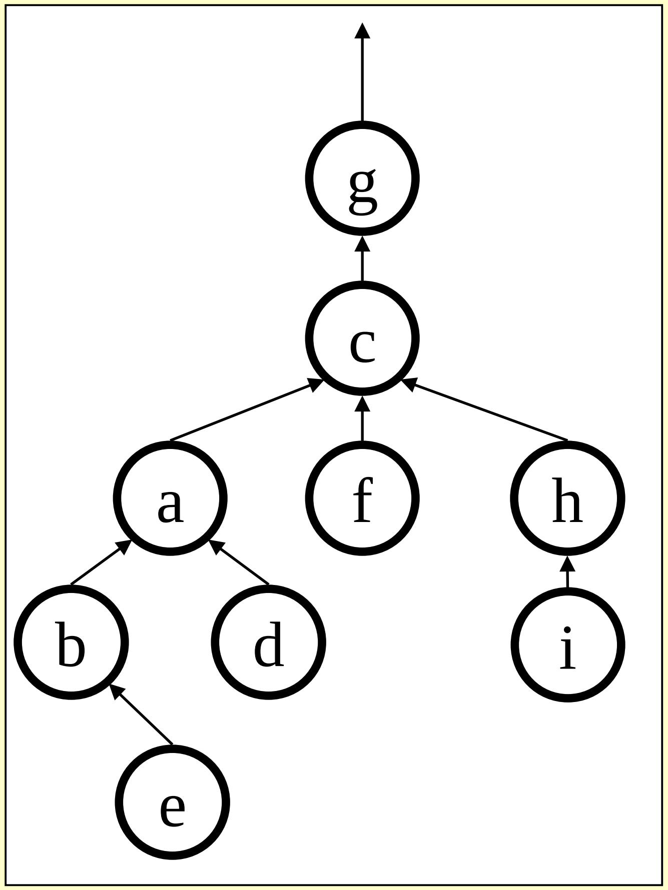
# The Whole Example (9/11)

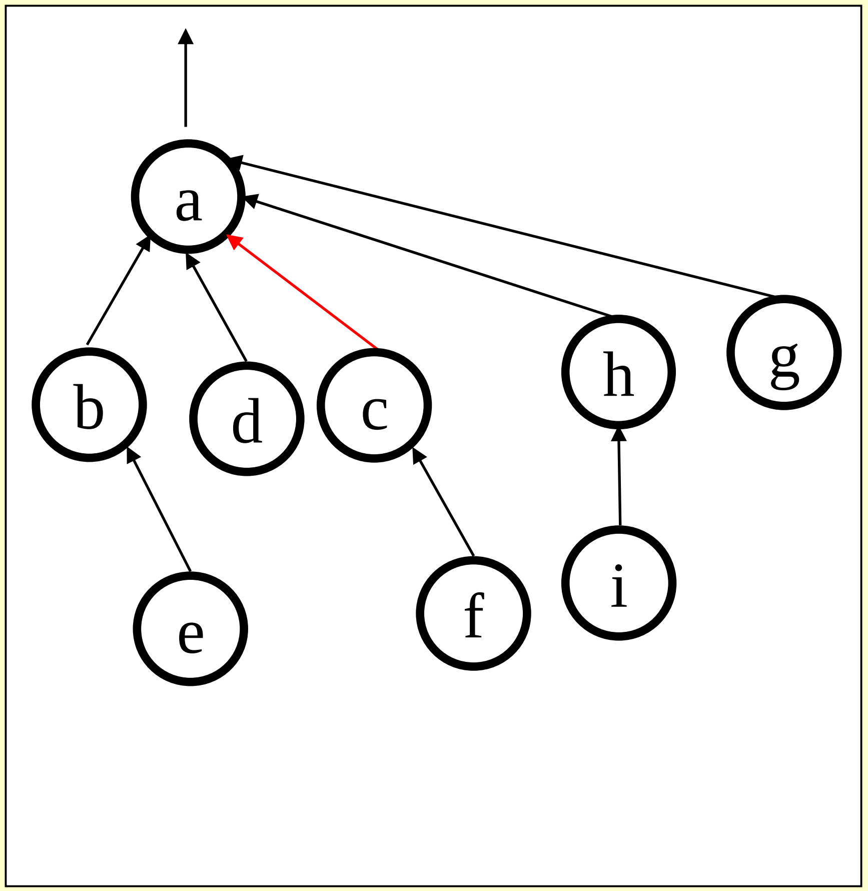
find(d)
find(g)
union(c, g)



find(g) = find(h)
So, no union.

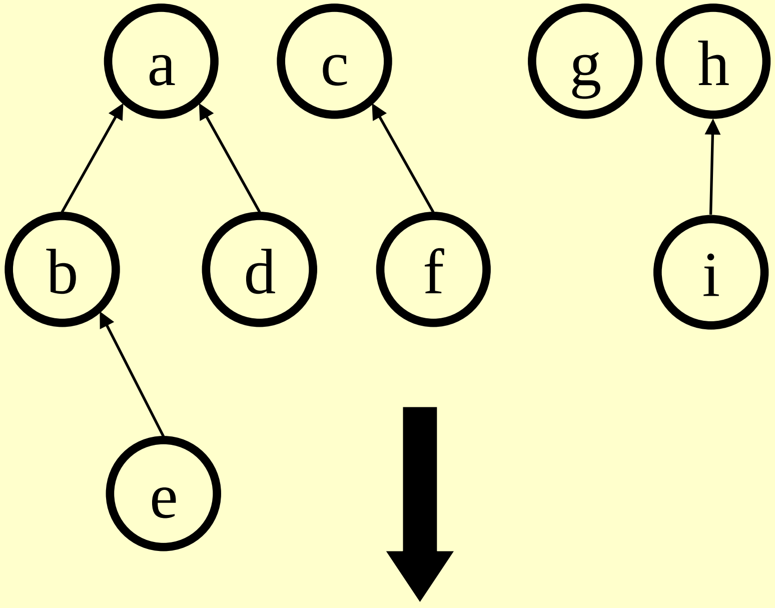
# Earlier vs New Tree





# Nifty storage trick

The root nodes now store sizes of the tree it is root of



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

# 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];
  }
}
```

# 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

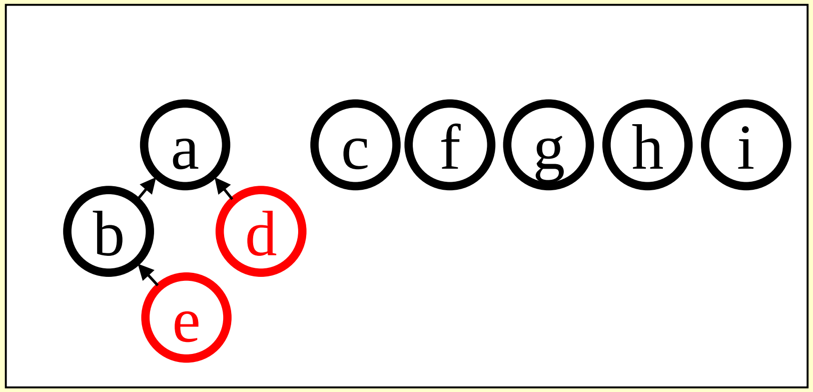
- Union by height
- Ranked union (cheaper approximation to union by height)
- See Weiss chapter 8.

Weighted Union:
Find : O(n) $\rightarrow$ O(log n)
Union: O(1)
Could we do a better job ?

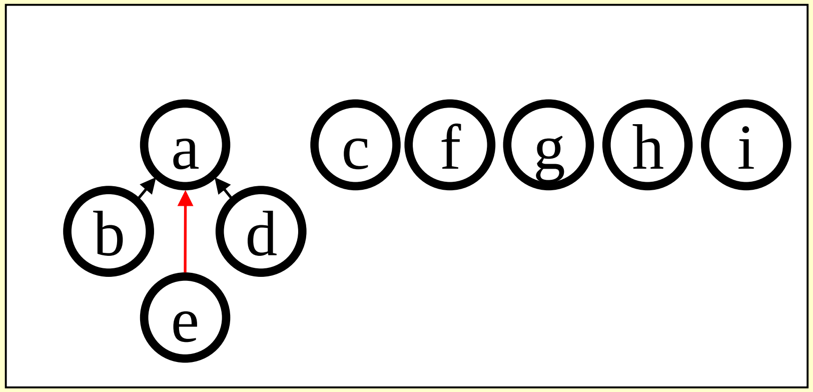
- In the array it would have been desirable to store the root node of the subtree instead of storing the parent node.
- Such a storage would have made find also O(1).
- Can this be achieved?

# 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



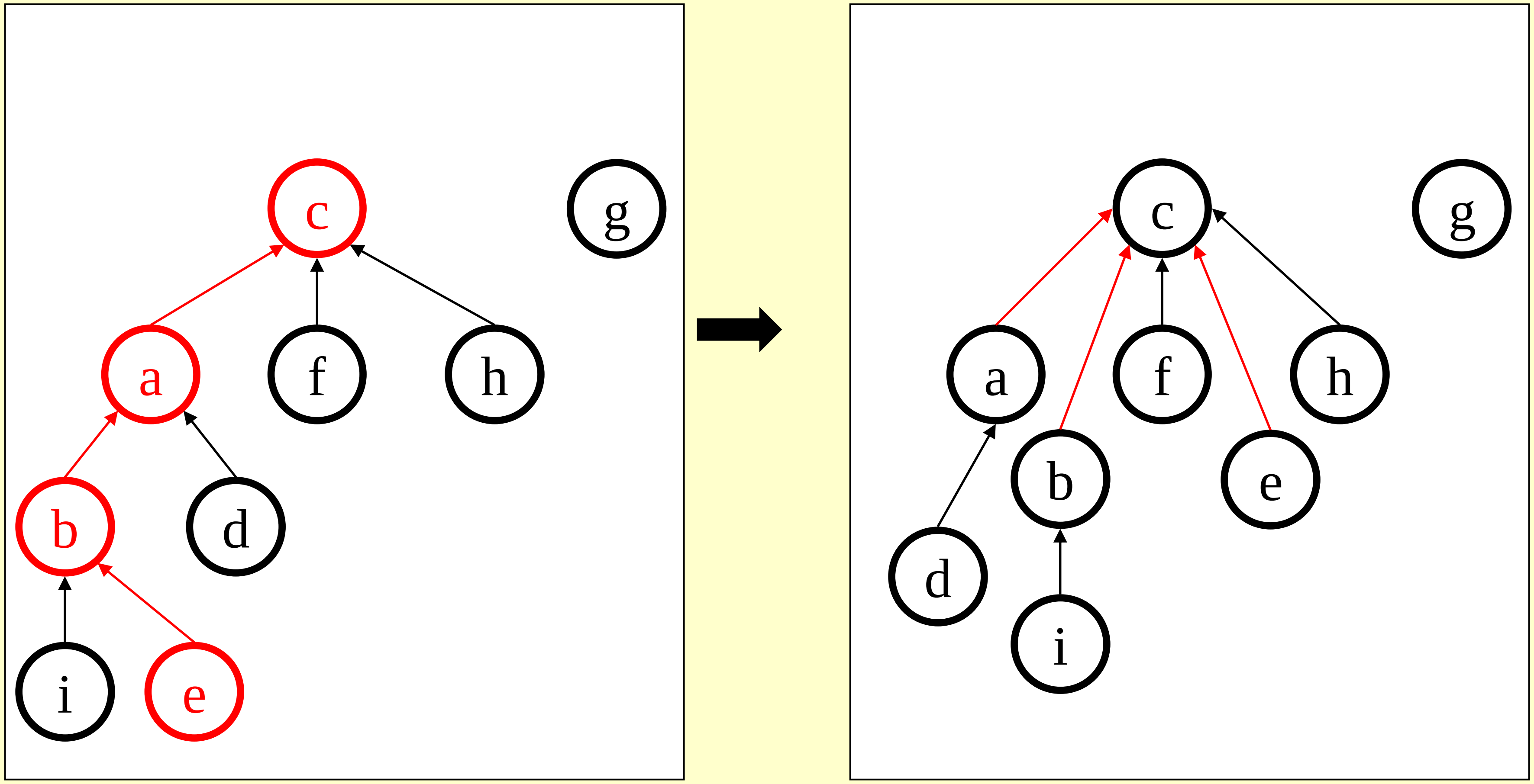
While we're finding ***e***, could we do anything else?



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

Journey so far
For E edges in a graph of n nodes

Find : E.O(n) → O(E.log n) with weighted union → O(E log*n) with path compression

Union: O(1)

Total time: E log E [//to sort edges] + E [// constant time to discard or include an edge]