= Data Structures for Disjoint Sets =

• some applications involve grouping n distinct elements into a collection of disjoint sets. Two important operations are

FIND: finding which set an element belongs to
UNION: uniting two sets

• disjoint set data structure: collection $S = \{s_1, \ldots, s_k\}$ of disjoint dynamic sets.

•Each set is identified by a representative, which is some member of the set.

Example:

- Each set represents cities between which network routes exist in a transportation network.

- Every time a new network route is added, two sets should be united.

- One would like to know which routing network a city belongs to.

• we represent each set $\underline{s_i}$ by a pointer to its representative.

MAKE-SET(x): creates a new set whose only member (and thus representative) is $\underline{x}$; returns a pointer to $\underline{x}$.

UNION(x, y): unites the dynamic sets containing $\underline{x}$ and $\underline{y}$, say S_x and S_y, into a new set $S_x \cup S_y$. Since S_x and S_y are disjoint, $S_x \cup S_y$ is disjoint of all other sets in the collection of sets (note that Sx and Sy no longer belong to this collection, they have been united into—and replaced by—$S_x \cup S_y$). The representative of $S_x \cup S_y$ can be chosen to be the representative of either S_x or S_y.

FIND(x): returns a pointer to the representative of the (unique) set containing $\underline{x}$.

• If n is the total number of elements in the sets of our collection of sets, then:

- n is also the number of MAKE-SET operations done so far. (A)

- n-1 is the maximum number of UNION operations done so far. Why? Each UNION reduces the number of sets in the collection by 1, and we created n (single element) sets so far.

• Let m be the total number of MAKE-SET, UNION, and FIND operations ($m \geq n$, since (A)).

• We analyze the running times of disjoint-set data structure in terms of both n and m.

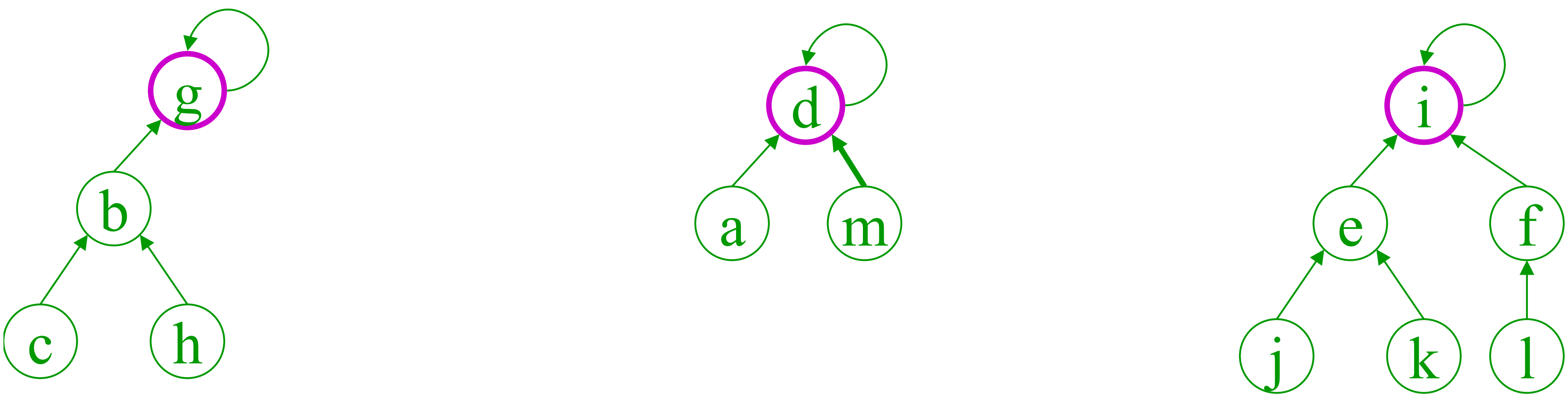
- What would be a good way of representing disjoint sets?

Idea: represent sets by rooted trees, where each tree represents a set, each node of a tree represents a member of the set, and the representative of a set is the root of the associated tree.

We call this representation a disjoint-set forest.

Example:

Sets {b, c, g, h}, {a, d, m}, {e, f, i, j, k, l}



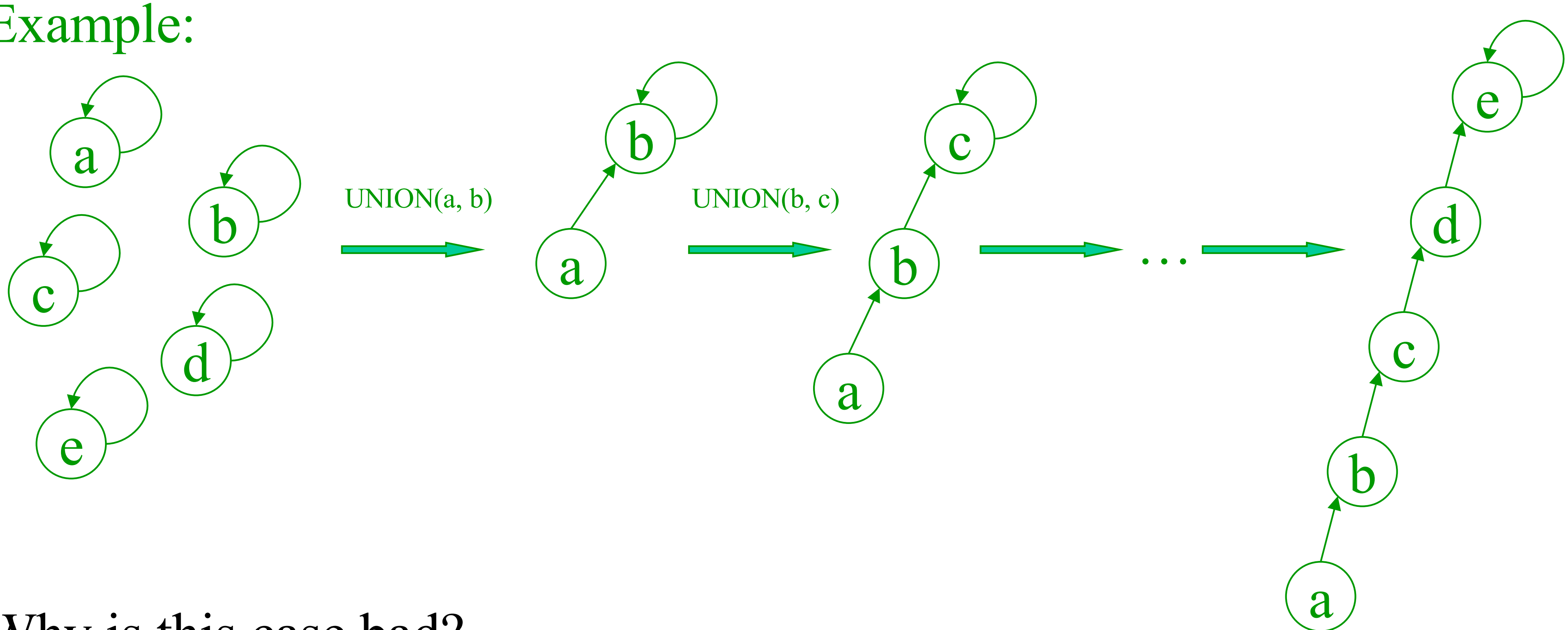
= root nodes = representatives of the sets.

- Disjoint set procedures using union by rank:

```
MAKE-SET(x)
        x.parent = x
        x.rank = 0

UNION(x, y)
        LINK( FIND(x), FIND(y) )          /* FIND(x) and FIND(y) will
                                          /* return the roots of the trees
                                          /* x and y belong to.

LINK(p, q)
        if p.rank > q.rank then           /* LINK(p, q) uses union by
                q.parent = p              /* rank to unite the trees rooted
        else                              /* at p and q
                p.parent = q
                if p.rank = q.rank then
                        q.rank = q.rank + 1

FIND(x)
        while x ≠ x.parent do
                x = x.parent
        return x
```

• Problem: A sequence of n-1 UNIONS may create a tree that is just a linear chain of n nodes.

Example:



Why is this case bad?

A FIND operation may take θ(n) time.

• Idea: keep the trees (associated with the sets) balanced. How?

more general definition of balanced tree: the height of the tree is O(log n).

- Union by rank:

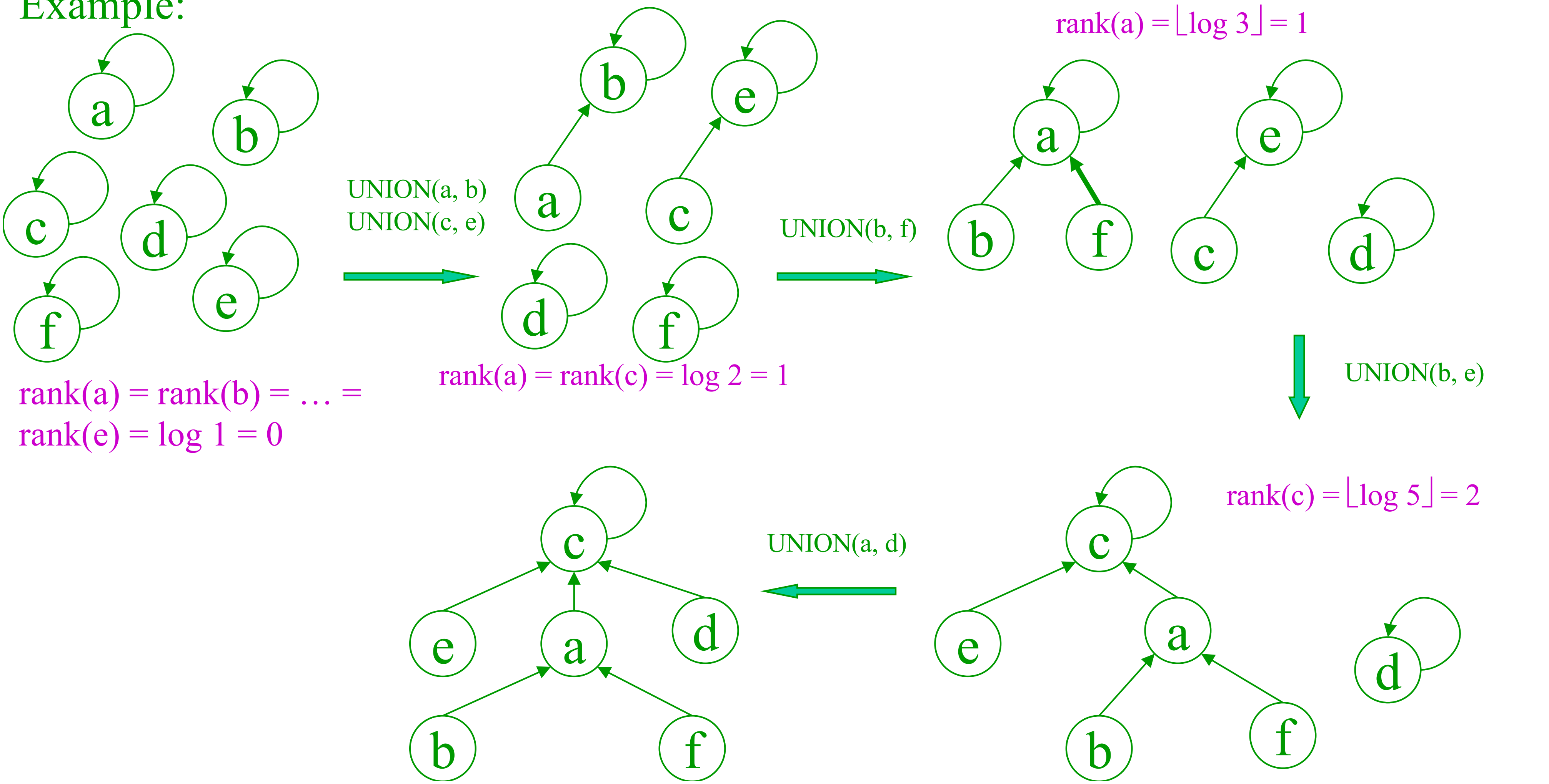
  Always make the tree with fewer nodes point to the tree with more nodes.

- Maintain the rank of each root

  upper bounds height of the tree: $\leq \lfloor \log(\#\text{ of nodes in tree}) \rfloor$

  $\therefore$ In union by rank, the root with smaller rank is made to point to the root with larger rank during a union operation.

Example:



- disjoint-set procedures using union by rank:

```
MAKE-SET(x)
    x.parent = x
    rank[x] = 0

UNION(x, y)
    LINK( FIND(x), FIND(y) )        /* FIND(x) and FIND(y) will
                                    /* return the roots of the trees
                                    /* x and y belong to.

LINK(p, q)
    if rank[p] > rank[q] then       /* LINK(p, q) uses union by
        q.parent = p                /* rank to unite the trees rooted
    else                            /* at p and q
        p.parent = q
        if rank[p] = rank[q] then
            rank[q] = rank[q] + 1

FIND(x)
    while x ≠ x.parent do
        x = x.parent
    return x
```

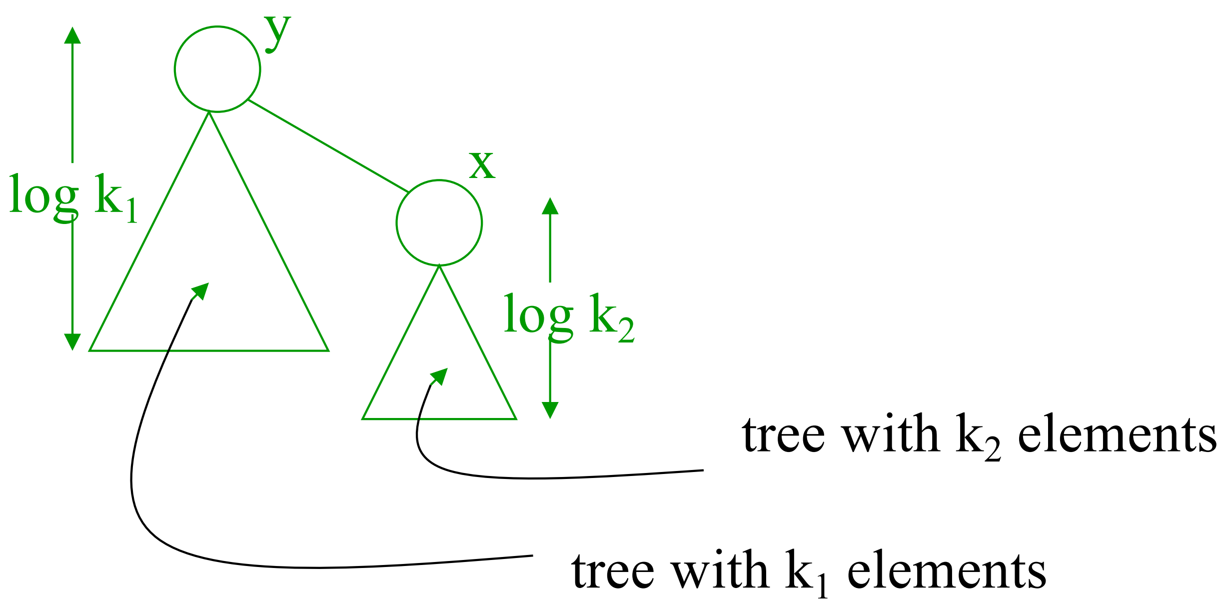
- m = total number of operations (MAKE-SET, UNION, & FIND)
  n = total number of MAKE-SET operations

- Union by rank heuristic: worst case total running time (for all m operations) is $\theta(m \log n)$. Why?

It is not hard to see that the total running time is $O(m \log n)$. We just need to show that a tree representing a set with k elements has height $O(\log k)$, since the maximum height of a tree is an upper bound on the running time of any of the three operations.

How do we show that the height of a tree with k elements is $O(\log k)$?

Use induction on number of elements in a tree.



- Base case: a tree with one element has height of $\log 1 = 0$.

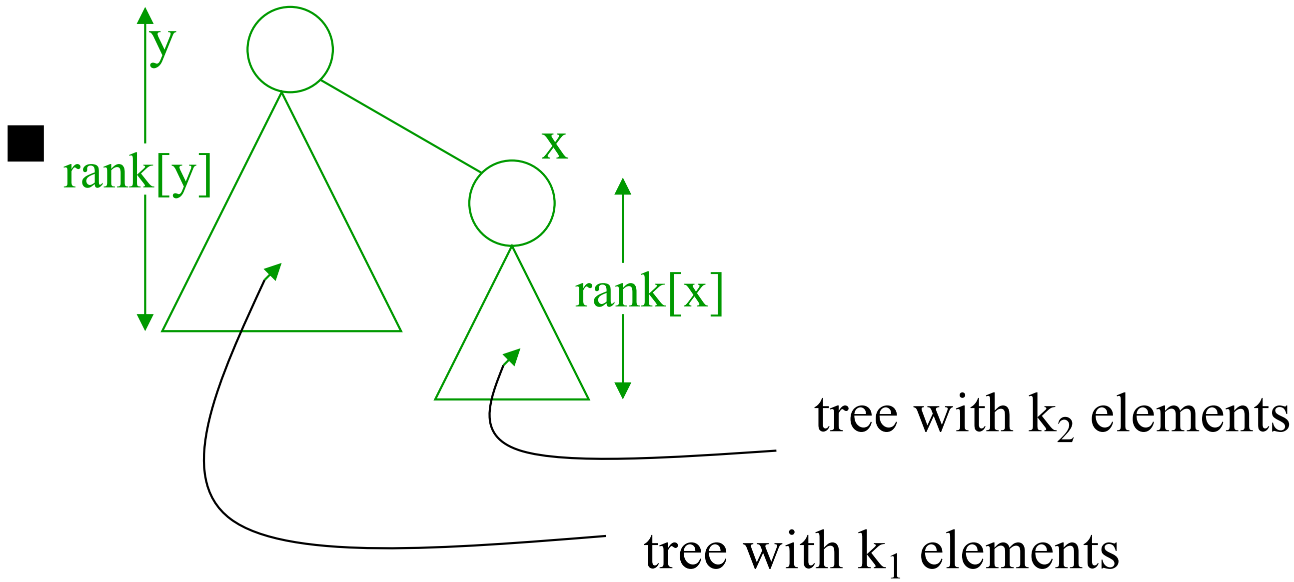
- Resulting tree (after UNION operation) has $k_1 + k_2$ elements.

- Height of resulting tree is

$$\max \{ \log k_2, (\log k_2) + 1 \} \leq \log (k_1 + k_2)$$

since we are using union by rank.

Use induction on number of elements in a tree.



- Base case: a tree with one element x has height $\leq$ rank[x] (= 0)

- Resulting tree (after UNION operation) has $k_1 + k_2$ elements.

- Height of resulting tree is

max { rank[y] before UNION, rank[x] + 1 } $\leq$ rank[y] in the resulting tree

since we are using union by rank

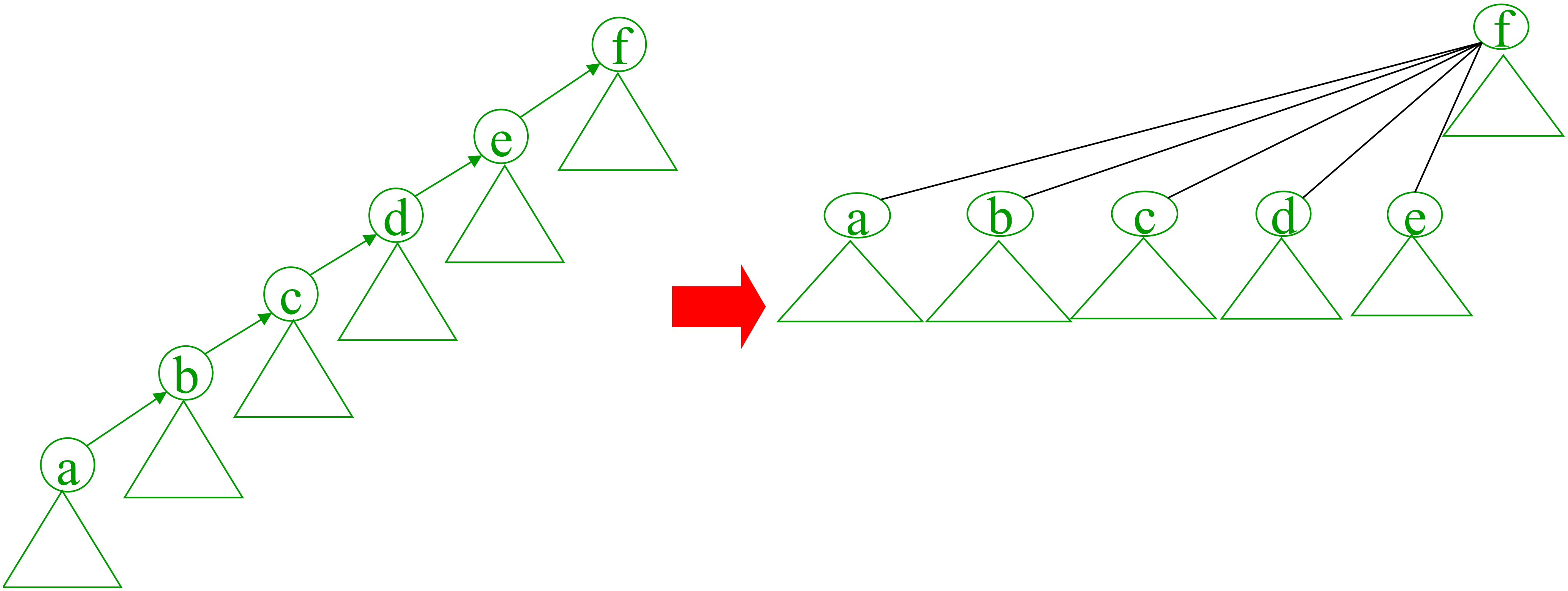
- Actually, we need to be more precise and show that the above argument works for the ranks assigned to each root node (and it does, see the   marks), and that rank[y] = $O(\log(k_1 + k_2))$ after UNION (if rank[x] = $O(\log k_2)$ and rank[y] = $O(\log k_1)$ before UNION).

• We can show that $rank[y] = O(\log (k_1 + k_2))$ again by induction, or

$\therefore$ the average time (or amortized time) to perform each of the m operations using union by rank is $O(\log n)$.

• We can do much better still…

• another heuristic: path compression!



• Path compression: use it during FIND operations, to make each node on the find path point directly to the root. Path compression does not change any ranks.

• FIND procedure with path compression:

```
FIND(x)
    if x ≠ x.parent then
        x.parent = FIND(x.parent)
    return x.parent
```

• FIND(x) procedure is a two-pass method: it makes one pass up the find path to find the root, and a second pass back down the find path to update each node so that it points directly to the root.

- As with union by rank, the path compression heuristic alone gives a worst-case running time of $O(m \log n)$.

- Idea: use both union by rank and path compression heuristics, to obtain a worst-case running time for any sequence of m operations of

$$O(m * \alpha(m, n))$$

where $\alpha(m, n)$ is a very slowly growing function—for any practical purposes, $\alpha(m, n) \leq 4$.

$\therefore$ Amortized running time of each operation is constant in all practical situations.