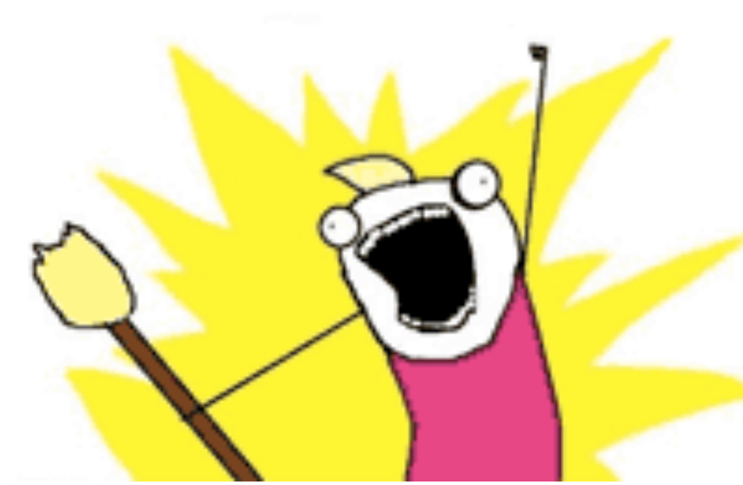
CSC263 Week 11

Announcements

Problem Set 5 (the last one!!) is due this Tuesday (Dec 1)

Problem Set 4 is graded. Average=77%



ADT: Disjoint Sets

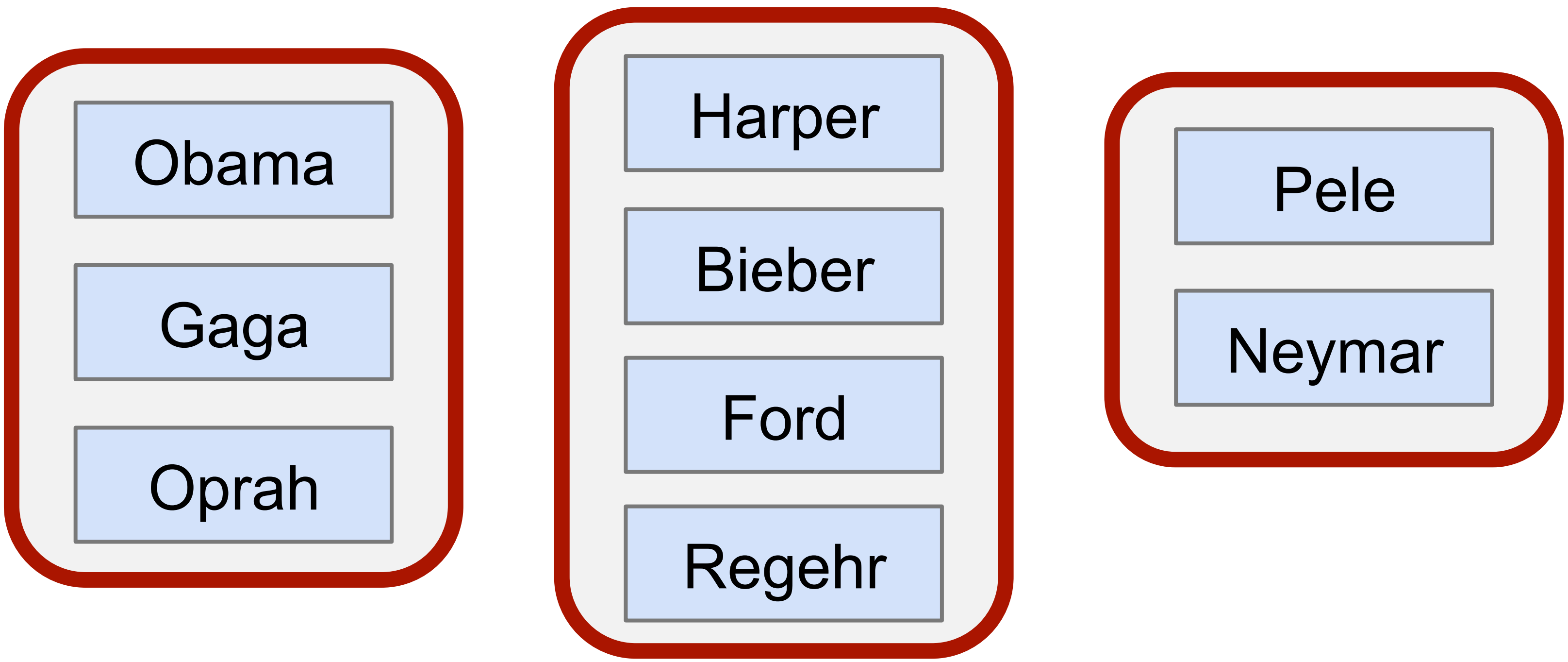
- ➔ What does it store?
- ➔ What operations are supported?

What does it store?

The elements in the sets can change dynamically.

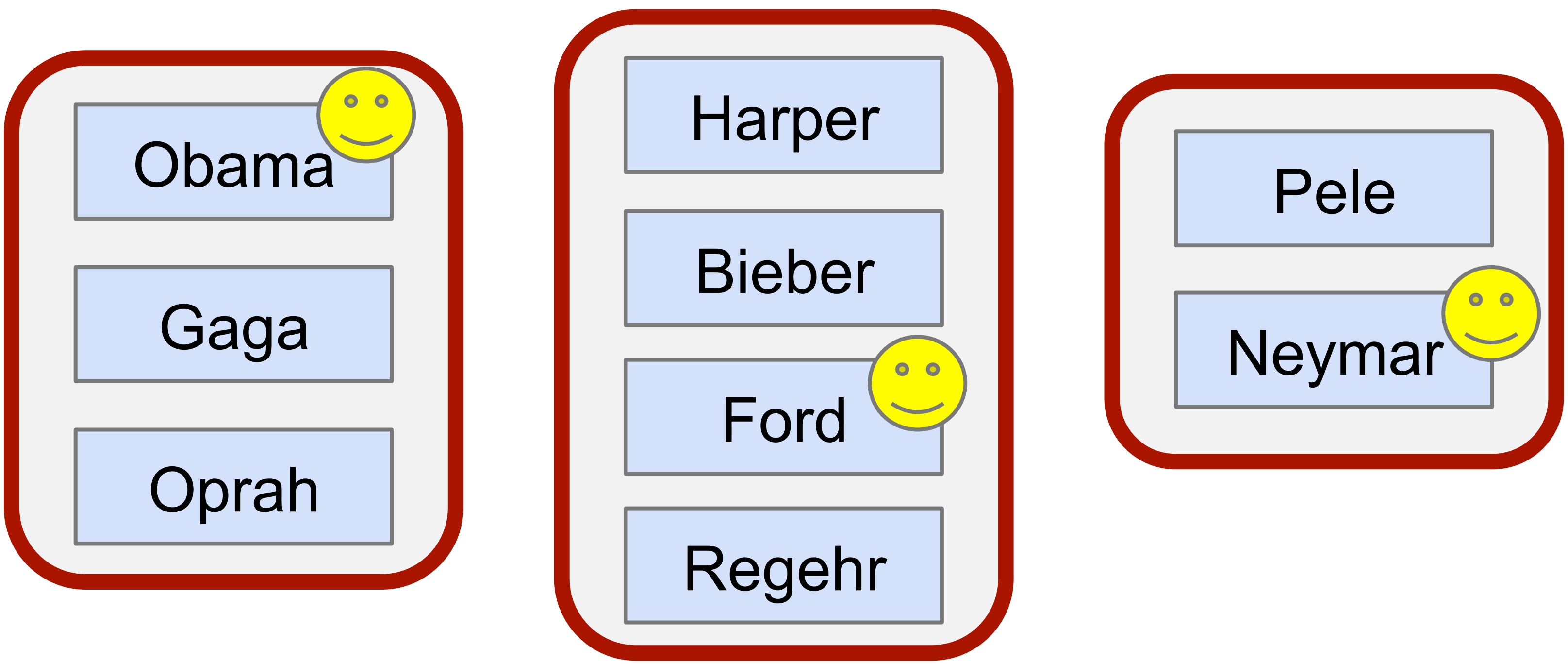
It stores a collection of (**dynamic**) **sets** of elements, which are **disjoint** from each other.

Each element belongs to **only one** set.



Each set has a **representative**

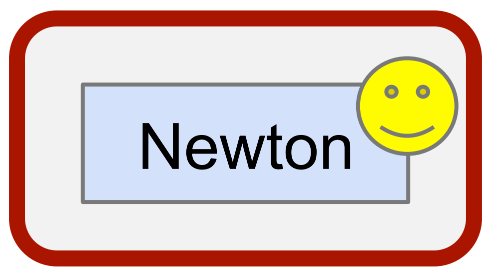
A set is **identified** by its representative.



Operations

MakeSet(x): Given an element **x** that does NOT belong to any set, create a new set **{x}**, that contains only **x**, and assign **x** as the representative.

MakeSet("Newton")



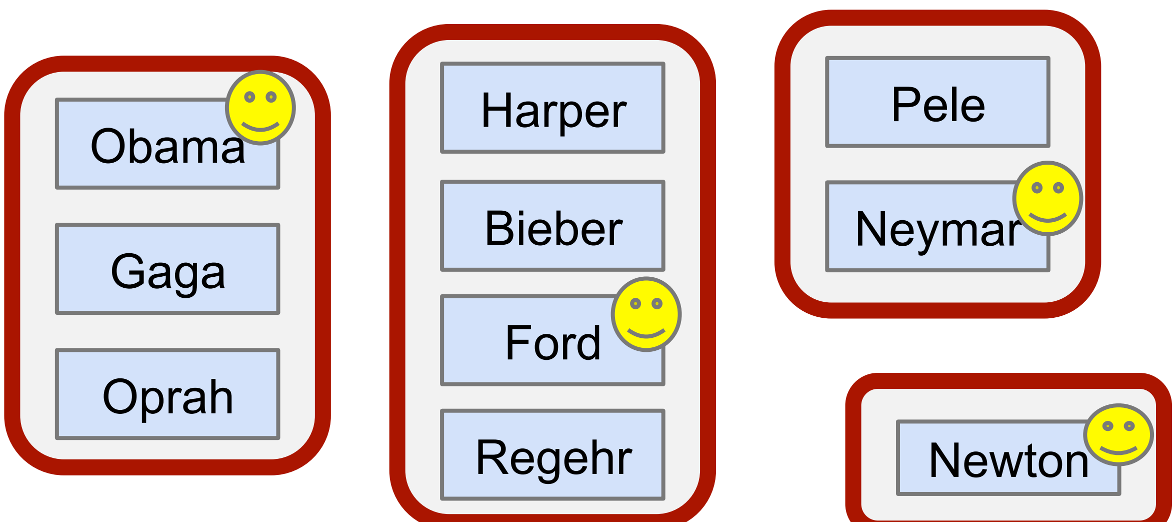
Operations

FindSet(x): return the representative of the set that contains **x**.

FindSet("Bieber") returns: **Ford**

FindSet("Oprah") returns: **Obama**

FindSet("Newton") returns: **Newton**

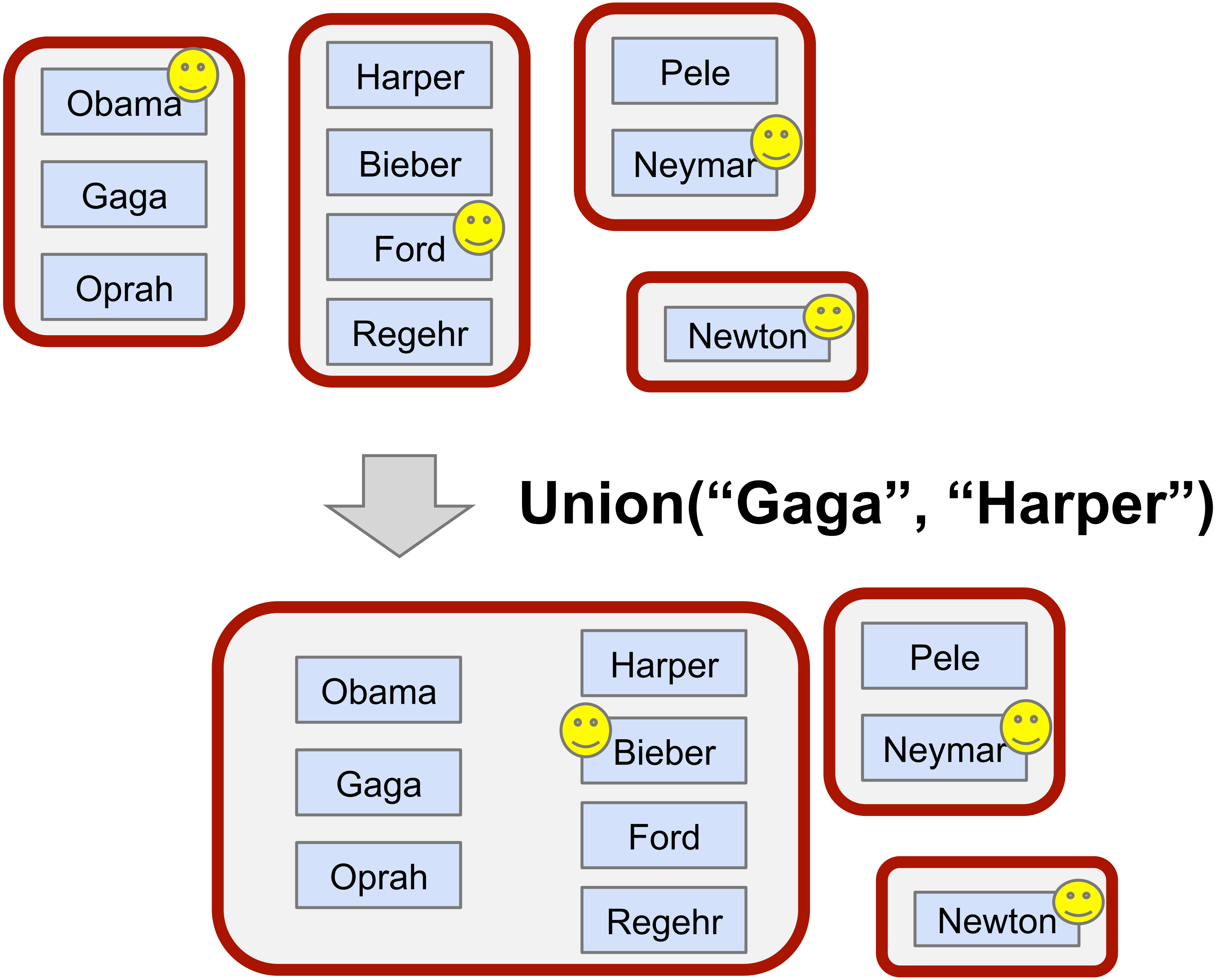


If **x** and **y** are already in the **same** set, then nothing happens.

Operations

Union(x, y): given two elements **x** and **y**, create a **new set** which is the **union** of the two sets that contain **x** and **y**, **delete** the original sets that contains x and y.

Pick a **representative** of the new set, usually (but not necessarily) one of the representatives of the two original sets.


Obama
Gaga
Oprah
Harper
Bieber
Ford
Regehr
Pele
Neymar
Newton
Union("Gaga", "Harper")
Obama
Gaga
Oprah
Harper
Bieber
Ford
Regehr
Pele
Neymar
Newton

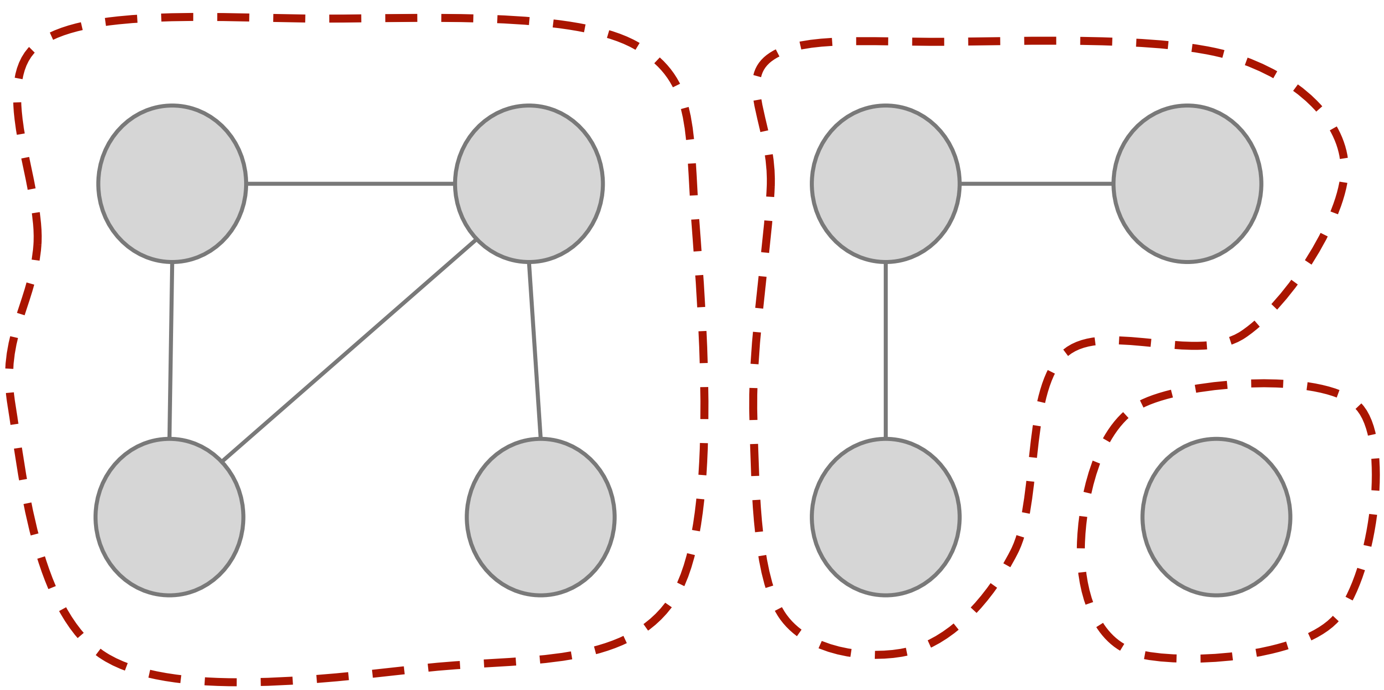
Applications

```
KRUSKAL-MST(G(V, E, w)):
    T ← {}
    sort edges so that w(e1)≤w(e2)≤...≤w(em)
    for each v in V:
        MakeSet(v)
    for i ← 1 to m:
        # let (ui, vi) = ei
        if FindSet(ui) != FindSet(vi):
            Union(ui, vi)
            T ← T ∪ {ei}
```

Other applications

For each edge (u, v)
if FindSet(u) != FindSet(v),
then Union(u, v)

Finding connected components of a graph



Summary: the ADT

- ➔Stores a collection of disjoint sets
- ➔Supported operations
  - ◆MakeSet(x)
  - ◆FindSet(x)
  - ◆Union(x, y)

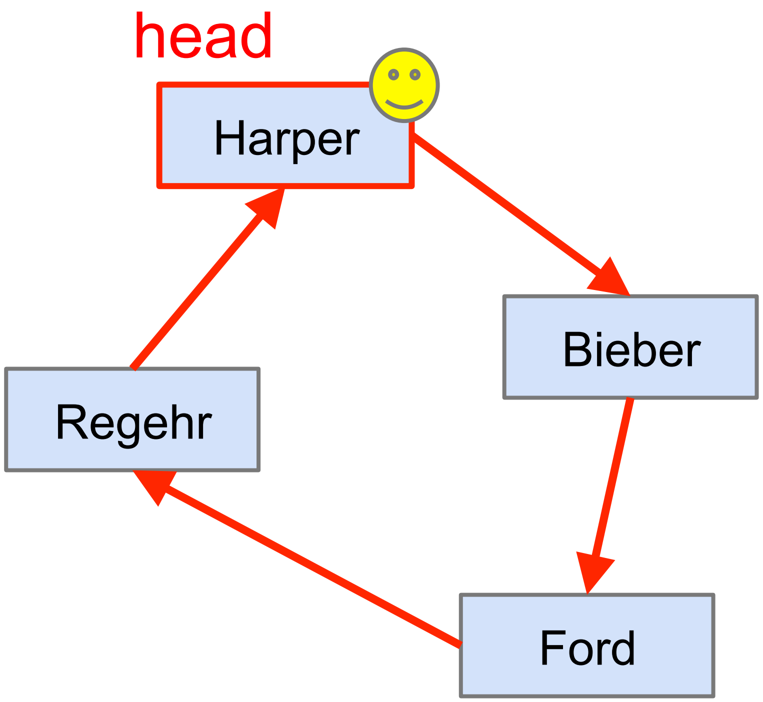
How to **implement** the Disjoint Sets ADT (efficiently) ?

Ways of implementations

1. Circularly-linked lists
2. Linked lists with extra pointer
3. Linked lists with extra pointer and with union-by-weight
4. Trees
5. Trees with union-by-rank
6. Trees with path-compression
7. Trees with union-by-weight and path-compression

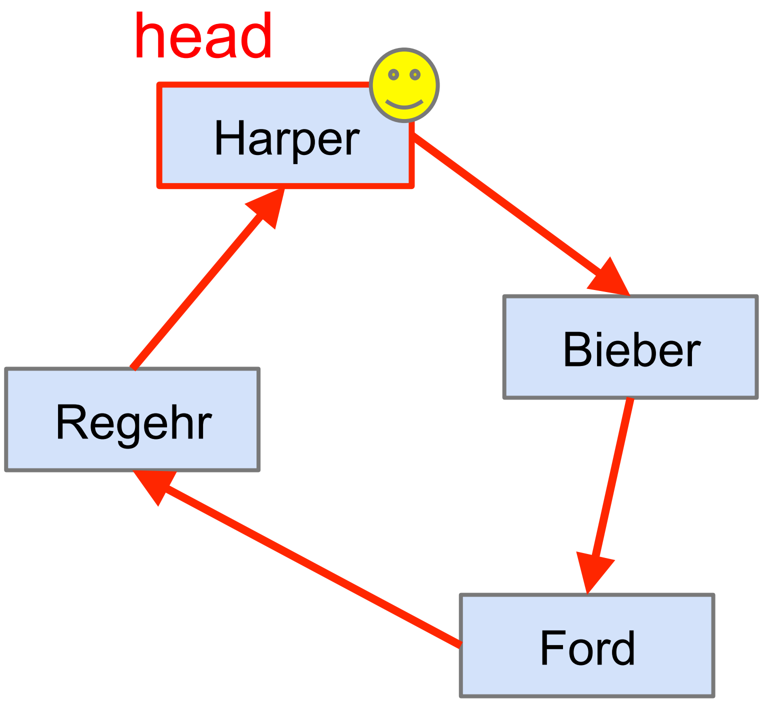
Circularly-linked list

Circularly-linked list



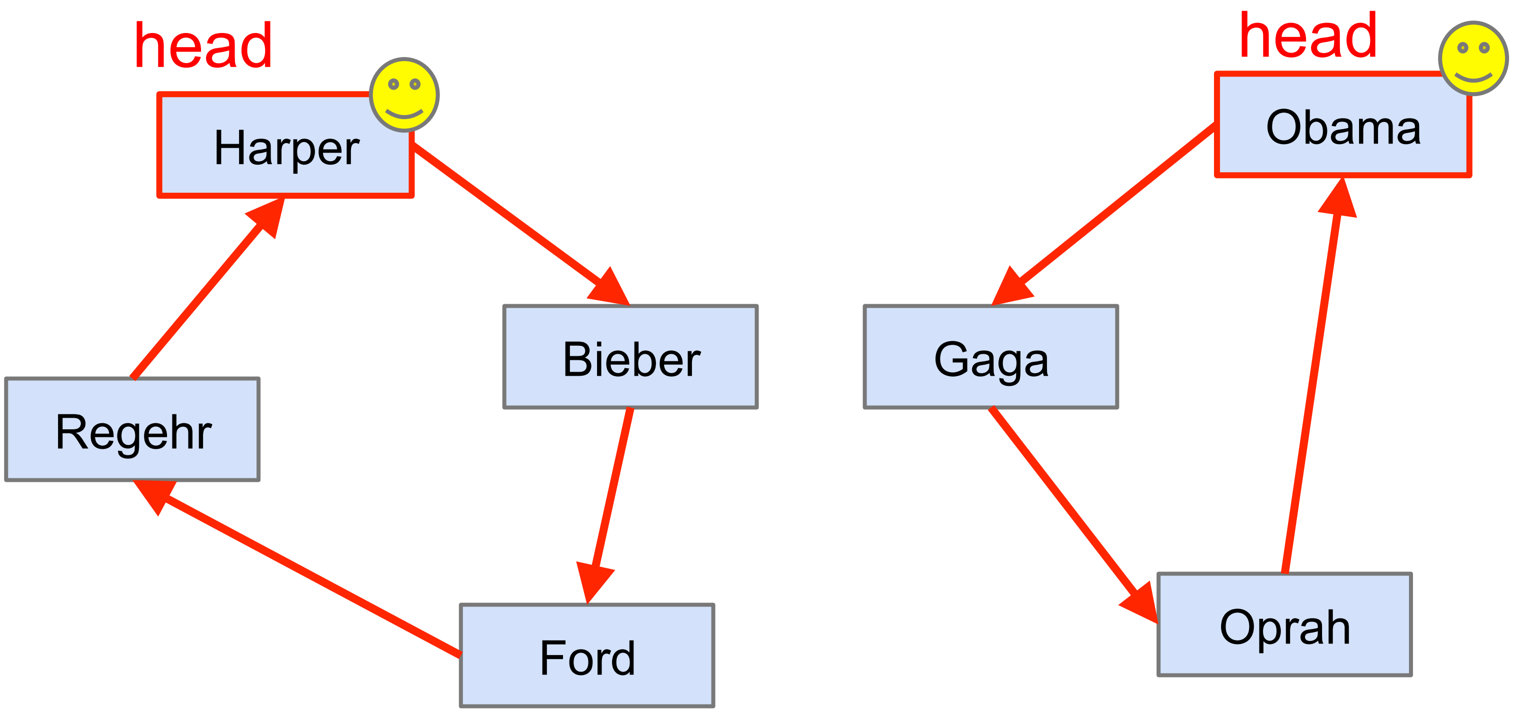
- One circularly-linked list per set
- Head of the linked list also serves as the representative.

Circularly-linked list



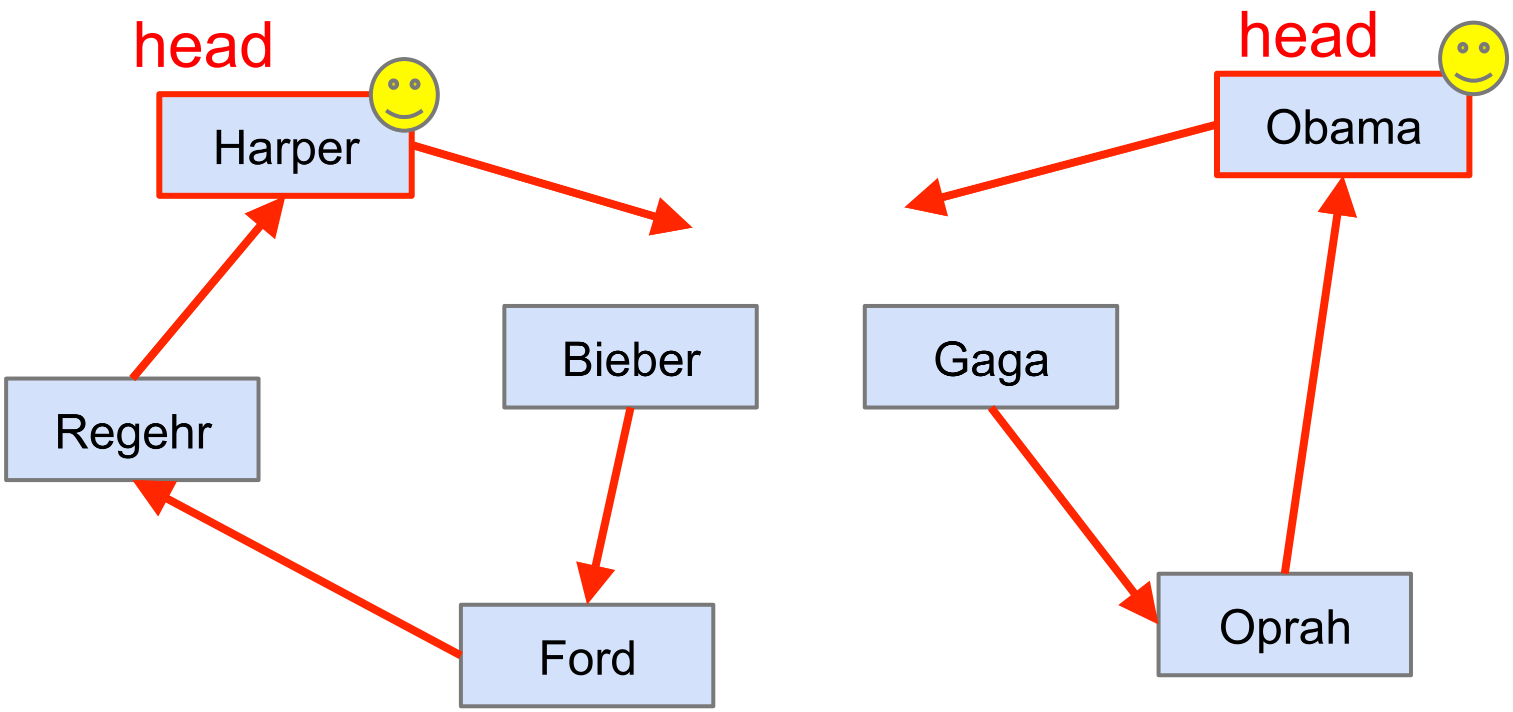
- **MakeSet(x):** just a new linked list with a single element x
  - worst-case: **O(1)**
- **FindSet(x):** follow the links until reaching the head
  - **Θ(Length of list)**
- **Union(x, y): ...**

Circularly-linked list: Union(Bieber, Gaga)

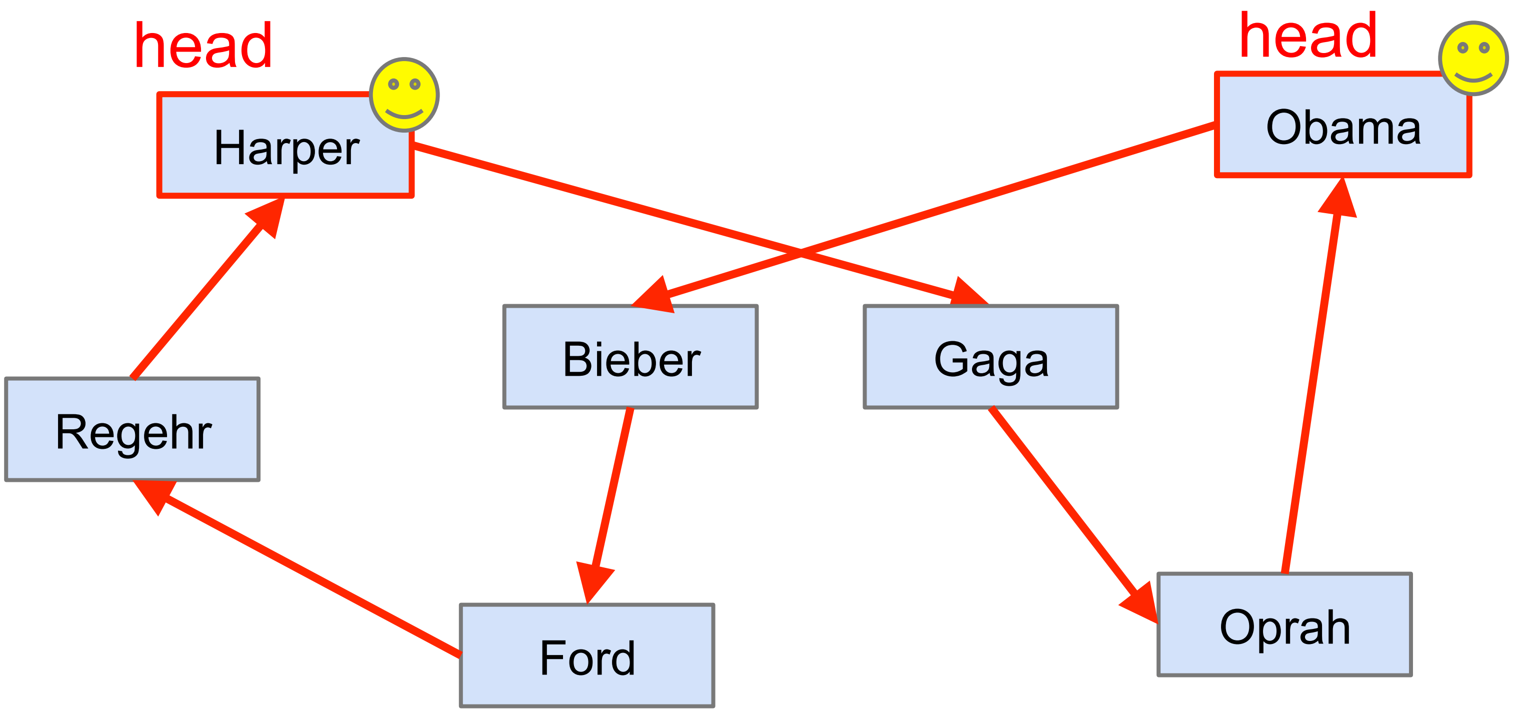


First, locate the head of each linked-list by calling FindSet, takes **Θ(L)**

Circularly-linked list: Union… 1

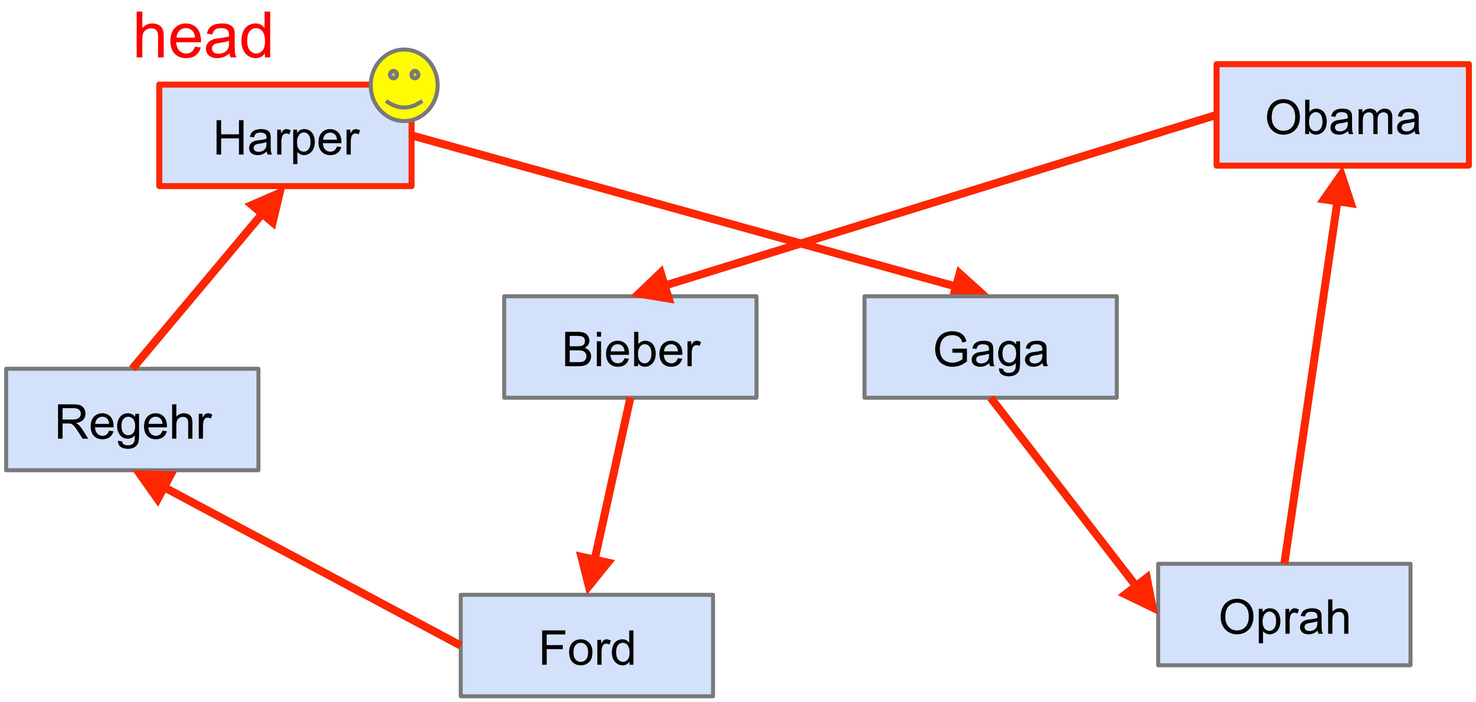


Circularly-linked list: Union… 2



Exchange the two heads' "next" pointers, **O(1)**

Circularly-linked list: Union... 3



Keep only one representative for the new set.

Circularly-linked list: runtime

FindSet is the time consuming operation

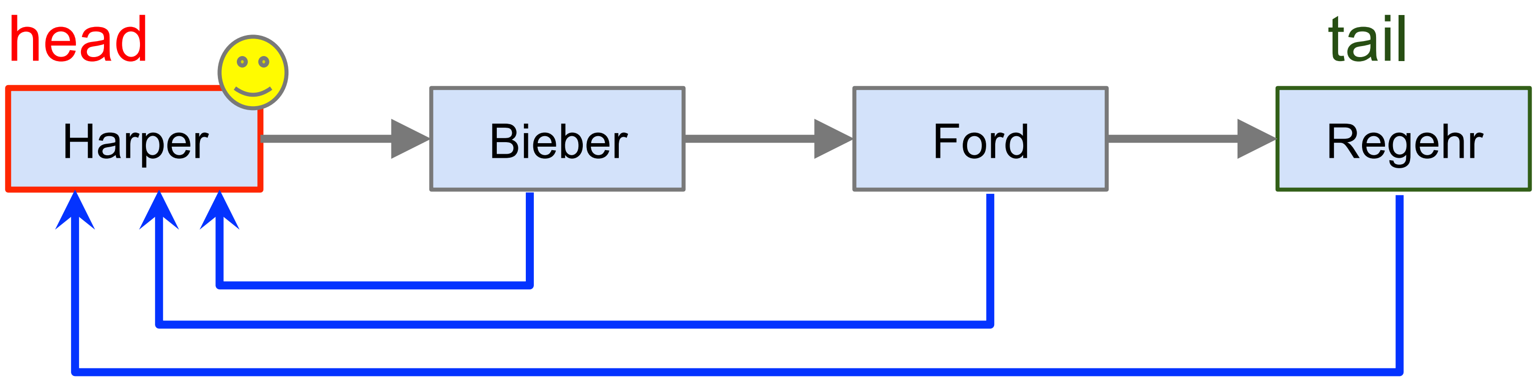
Amortized analysis: How about the **total cost** of a sequence of **m** operations (MakeSet, FindSet, Union)?

- ➔ A bad sequence: **m/4** MakeSet, then **m/4 - 1** Union, then **m/2 +1** FindSet
  - ◆ why it's bad: because many FindSet on a large set (of size m/4)

- ➔ Total cost: **$\Theta(m^2)$**
  - ◆ each of the **m/2 + 1** FindSet takes **$\Theta(m/4)$**

Linked list
with extra pointer
(to head)

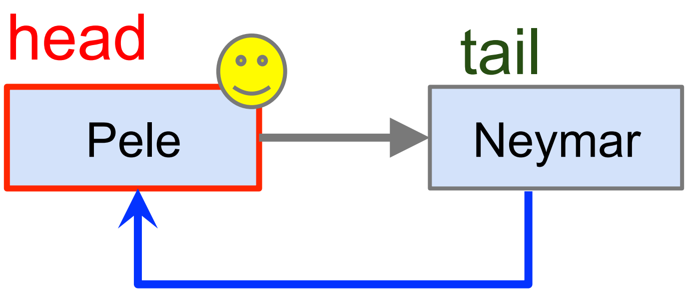
Linked list with pointer to head



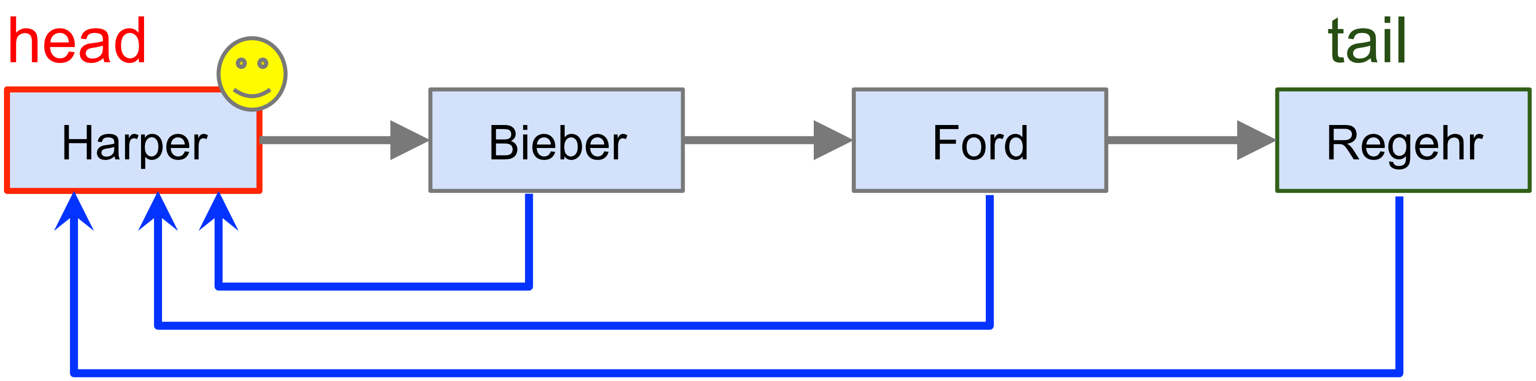
- ➔ **MakeSet** takes **O(1)**
- ➔ **FindSet** now takes **O(1)**, since we can go to head in 1 step, better than circular linked list
- ➔ **Union…**

Linked list with pointer to head

Union(Bieber, Pele)

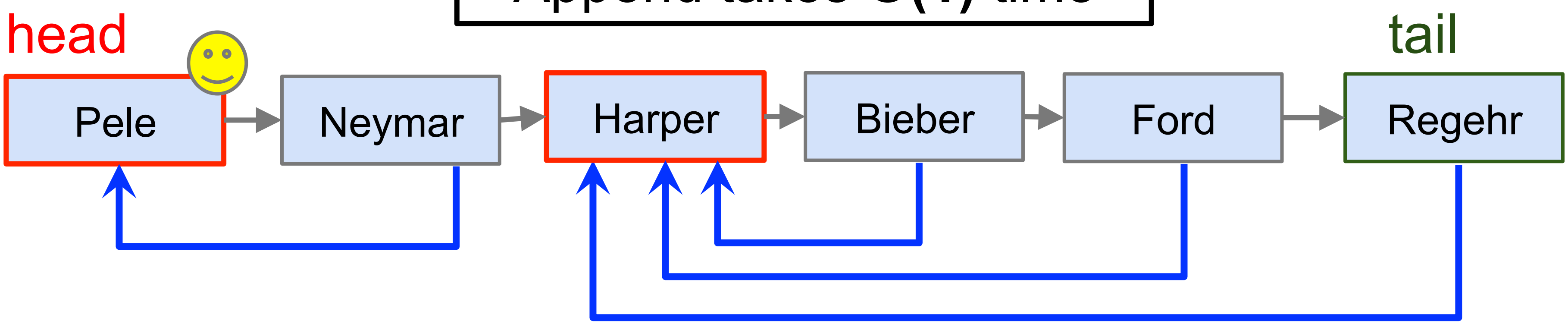


Idea:
Append one list to the other, then **update** the pointers to head

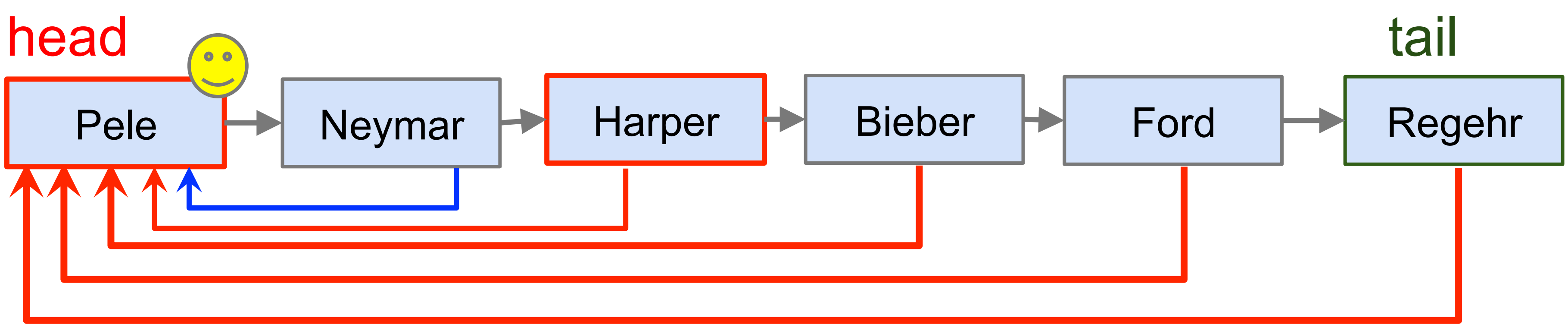


Linked list with pointer to head

Append takes **O(1)** time



Update pointers take **O(L of appending list)**



Linked list with pointer to head

MakeSet and **FindSet** are fast, **Union** now becomes the time-consuming one, especially if appending a long list.

Amortized analysis: The total cost of a sequence of **m** operations.

- ➔ Bad sequence: **m/2** MakeSet, then **m/2 - 1** Union, then 1 whatever.
  - Always let the longer list append, like 1 appd 1, 2 appd 1, 3 appd 1, ..., m/2 -1 appd 1.

➔ Total cost: $\Theta(1+2+3+...+m/2 - 1) = \boldsymbol{\Theta(m^2)}$

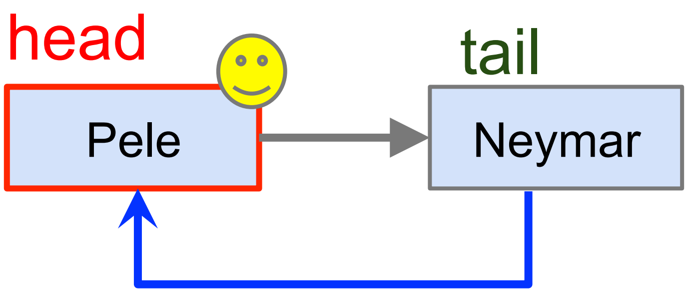
Linked list

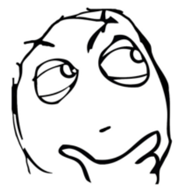
with extra pointer to head

with union-by-weight

Linked list with union-by-weight

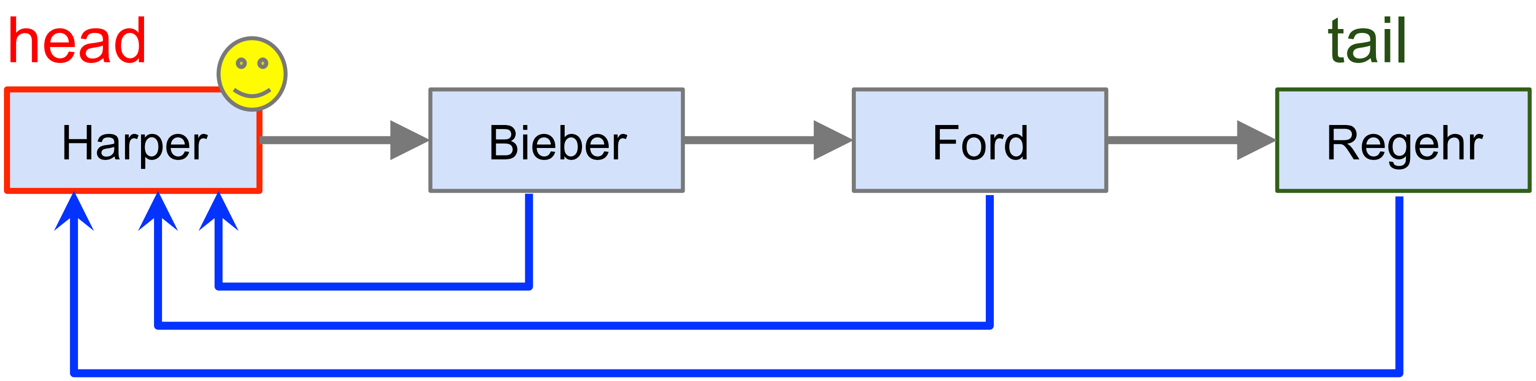
Union(Bieber, Pele)



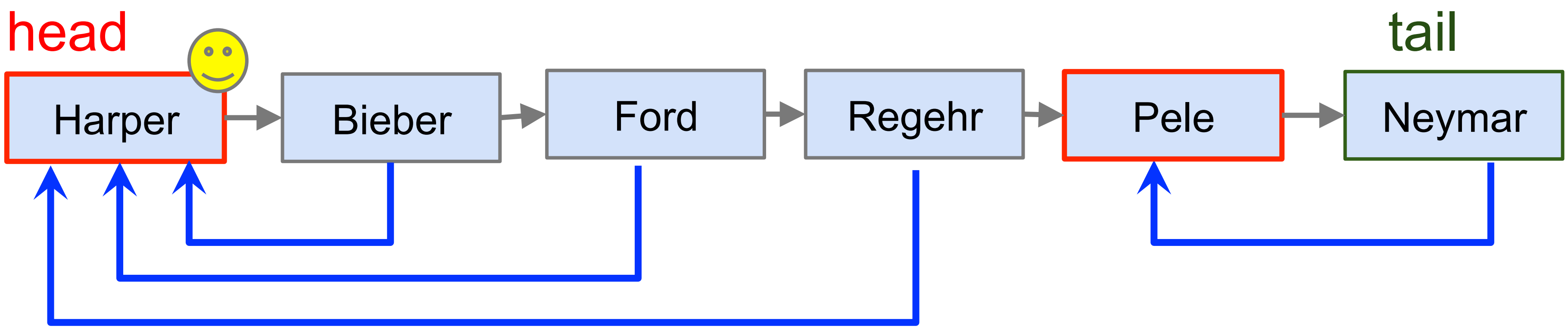


Here we have a choice, let's be a bit smart about it…

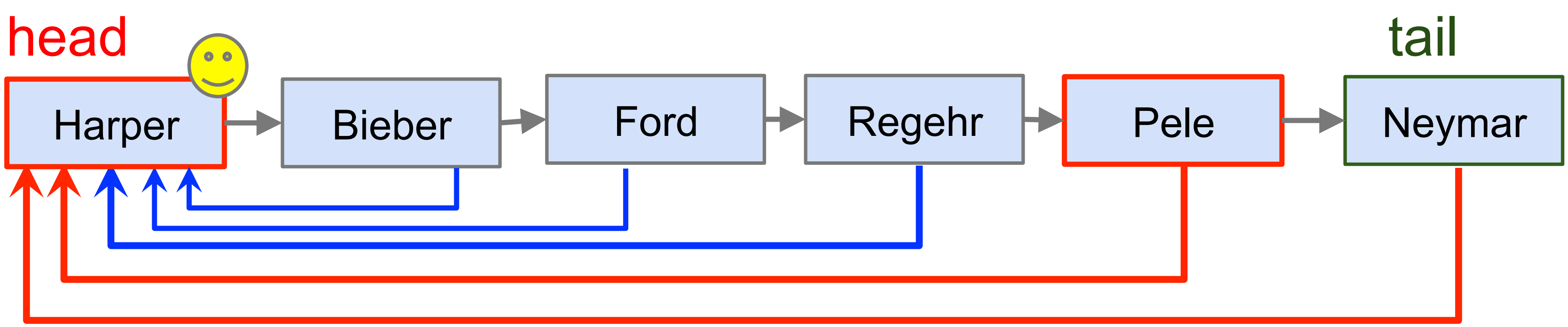
Append the shorter one to the longer one



Linked list with union-by-weight



Need to keep track of the **size** (**weight**) of each list, therefore called **union-by-weight**



Linked list with union-by-weight

Union-by-weight sounds like a simple heuristic, but it actually provides significant improvement.

For a sequence of **m** operations which includes **n** MakeSet operations, i.e., **n** elements in total,
the total cost is $\mathbf{O(m + n \log n)}$

i.e., for the previous sequence with m/2 MakeSet and m/2 - 1 Union, the total cost would be $\mathbf{O(m \log m)}$, as opposed to $\mathbf{\Theta(m^2)}$ when without union-by-weight.

Linked list with union-by-weight

Proof: (assume there are n elements in total)

- ➔ Consider an arbitrary element **x**, how many times does its head pointer need to be updated?
- ➔ Because **union-by-weight**, when **x** is updated, it must be in the smaller list of the two. In other words, after **union**, the size of list at least **doubles**.
- ➔ That is, every time x is **updated**, set size **doubles**.
- ➔ There are only n elements in total, so we can double at most **O(log n)** times, i.e., **x** can be updated at most **O(log n)**.
- ➔ Same for all **n** elements, so total updates **O(n log n)**

Ways of implementing Disjoint Sets

1. Circularly-linked lists $\Theta(m^2)$
2. Linked lists with extra pointer $\Theta(m^2)$
3. Linked lists with extra pointer and with union-by-weight $\Theta(m \log m)$
4. Trees
5. Trees with union-by-rank
6. Trees with path-compression
7. Trees with union-by-weight and path-compression

Benchmark:

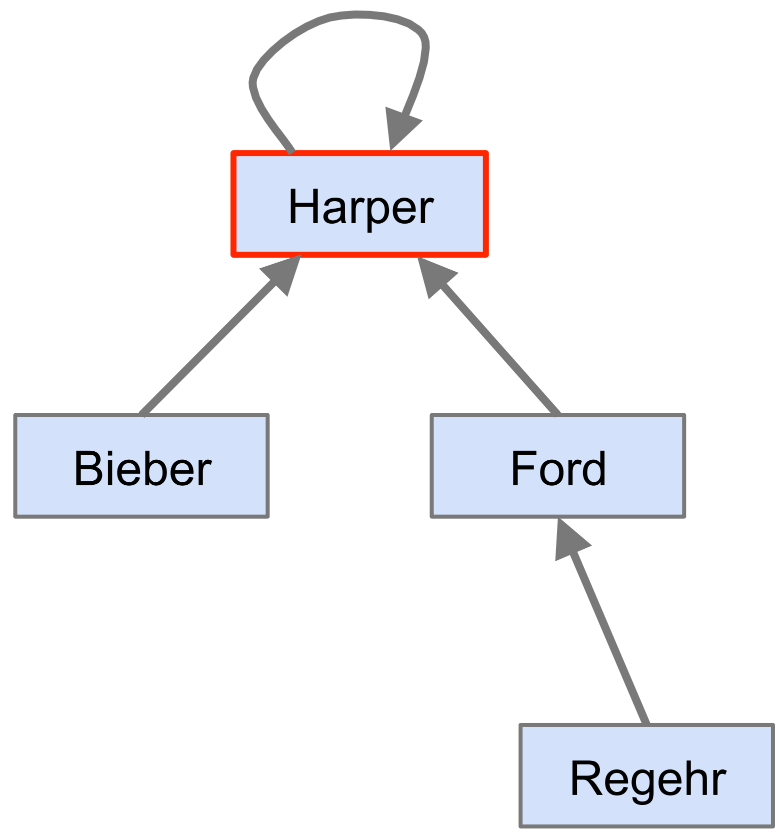
Worst-case total cost of a sequence of **m** operations (MakeSet or FindSet or Union)

Trees

a.k.a. disjoint set forest

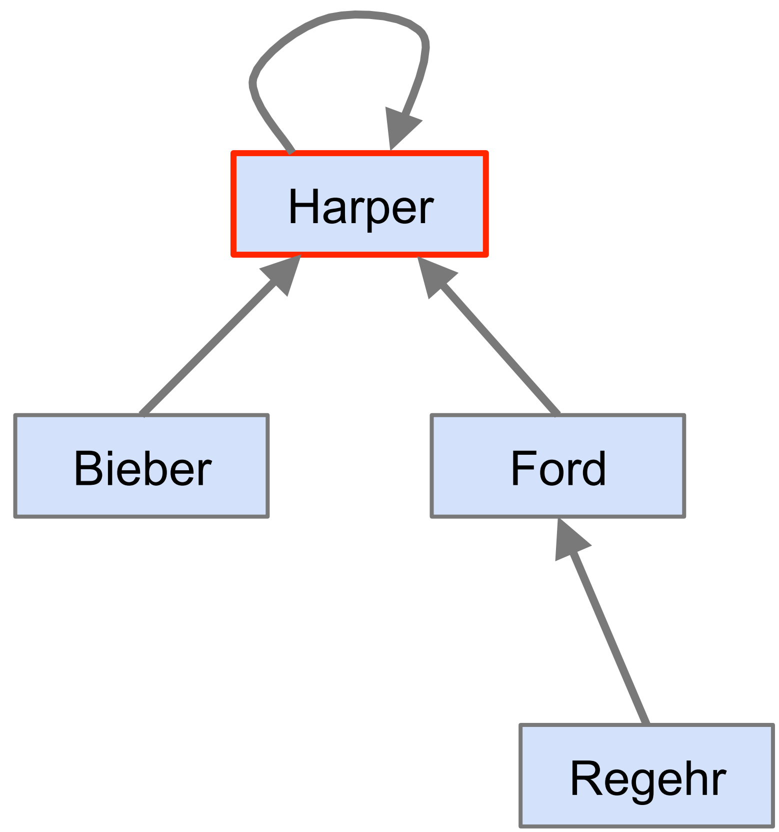
Each set is an “inverted” tree

- ➔ Each element keeps a pointer to its **parent** in the tree
- ➔ The root points to **itself** (test root by **x.p = x**)
- ➔ The **representative** is the root
- ➔ NOT necessarily a binary tree or balanced tree



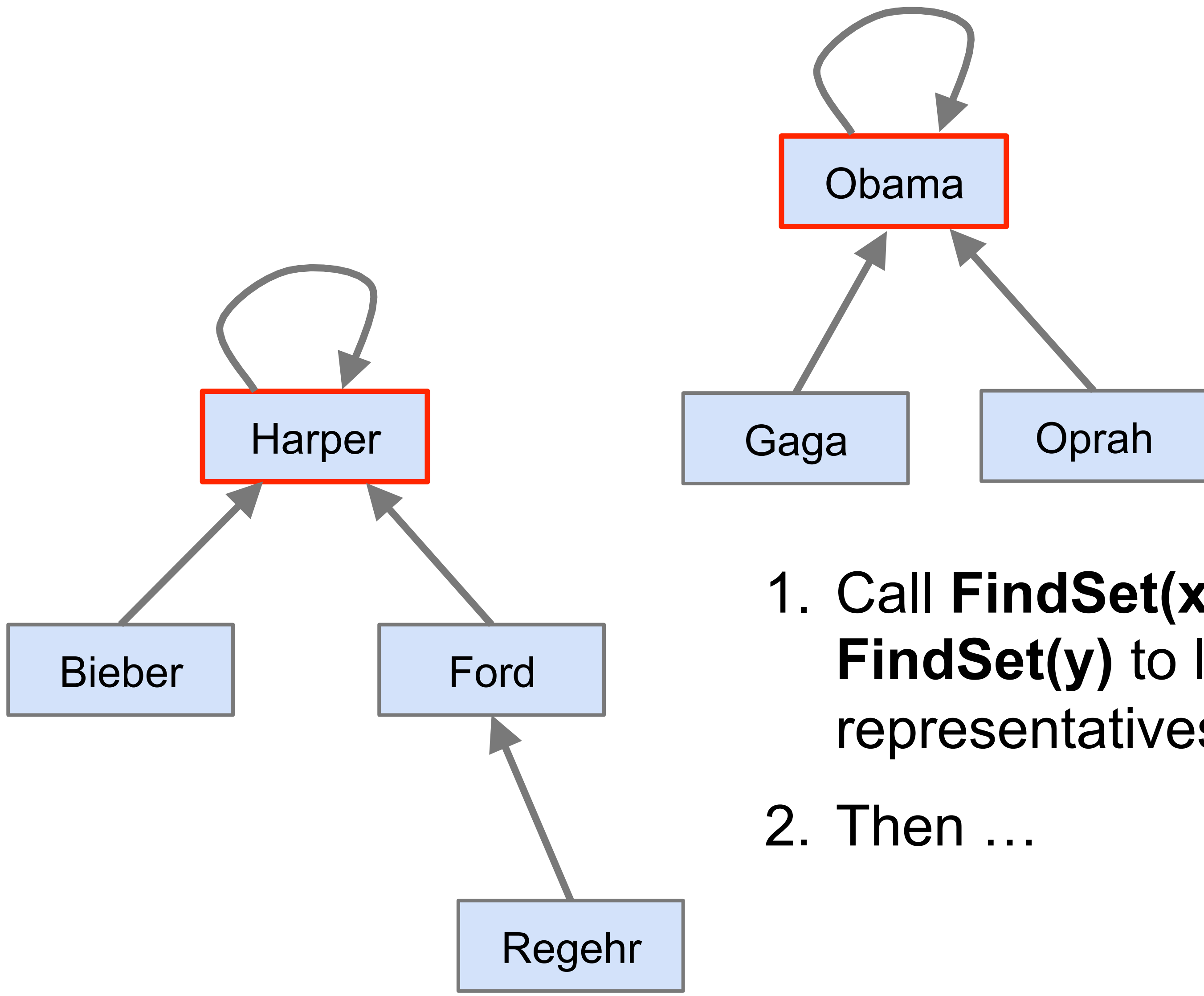
Operations

- **MakeSet(x):** create a single-node tree with root x
  - **O(1)**
- **FindSet(x):** Trace up the parent pointer until the root is reached
  - **O(height of tree)**
- **Union(x, y)...**



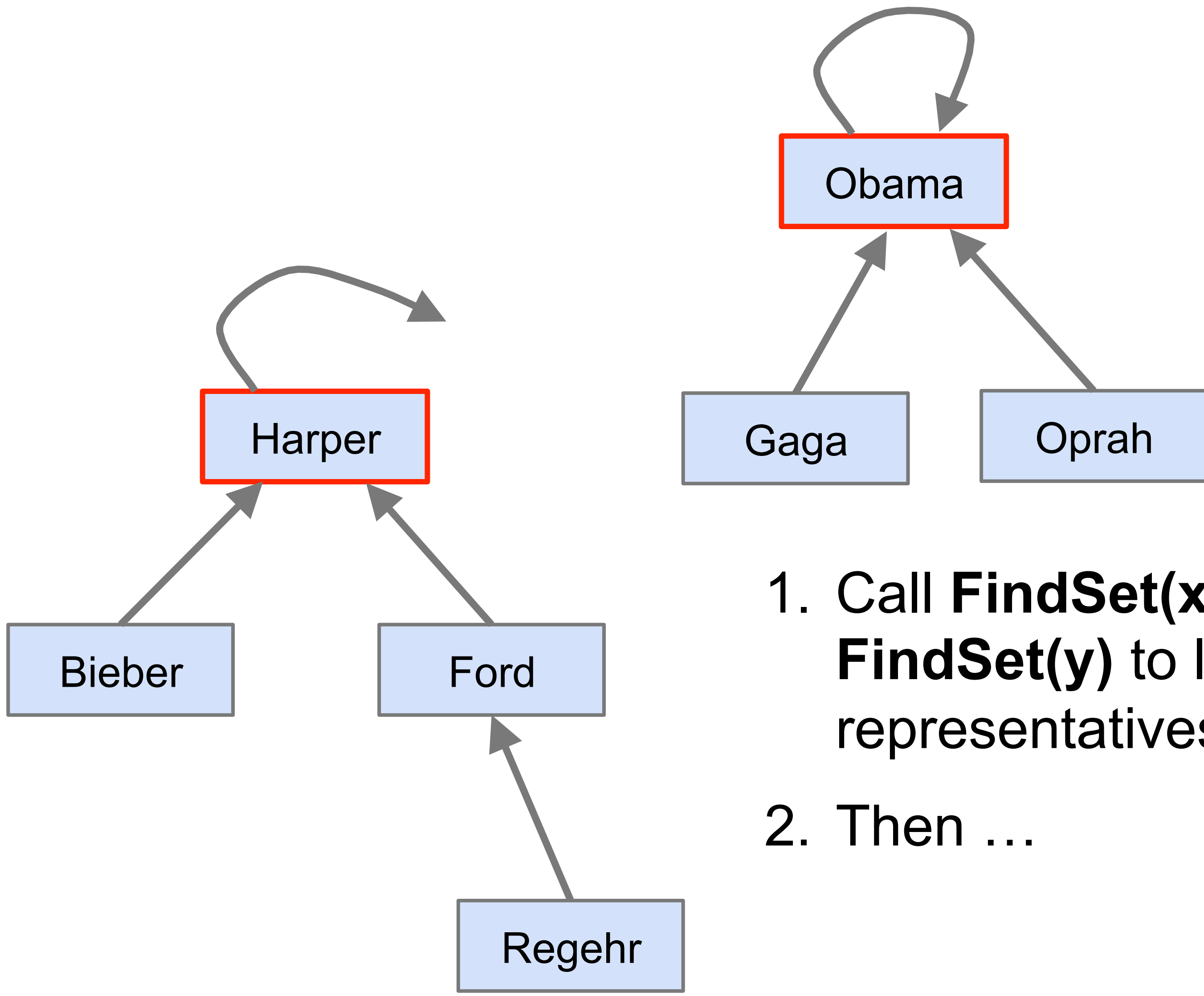
Trees with small heights would be nice.

Union(Bieber, Gaga)



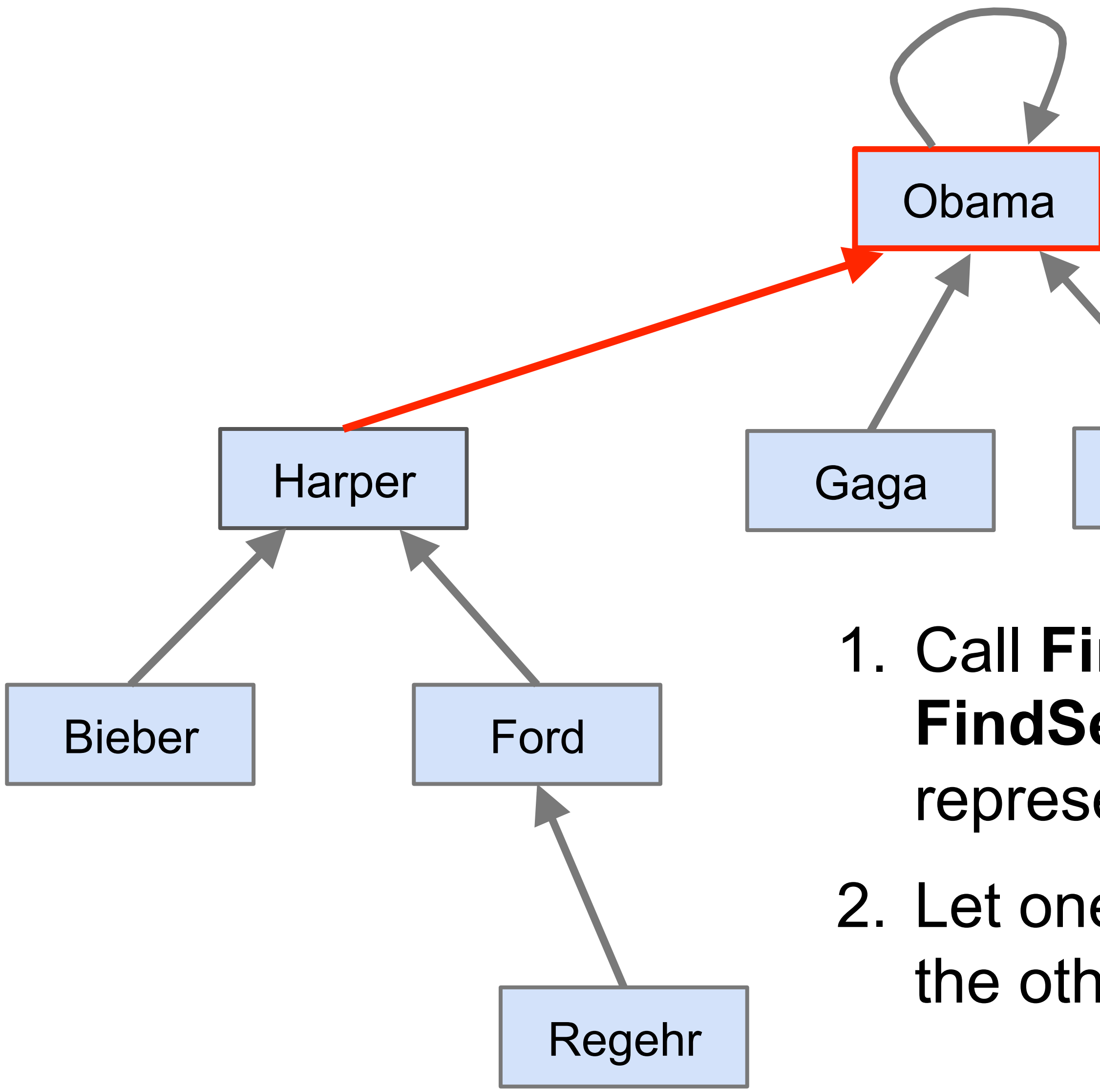
1. Call **FindSet(x)** and **FindSet(y)** to locate the representatives, **O(h)**
2. Then …

Union(Bieber, Gaga)

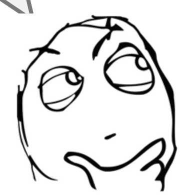


1. Call **FindSet(x)** and **FindSet(y)** to locate the representatives, **O(h)**
2. Then …

Union(Bieber, Gaga)



Could we have been smarter about this?

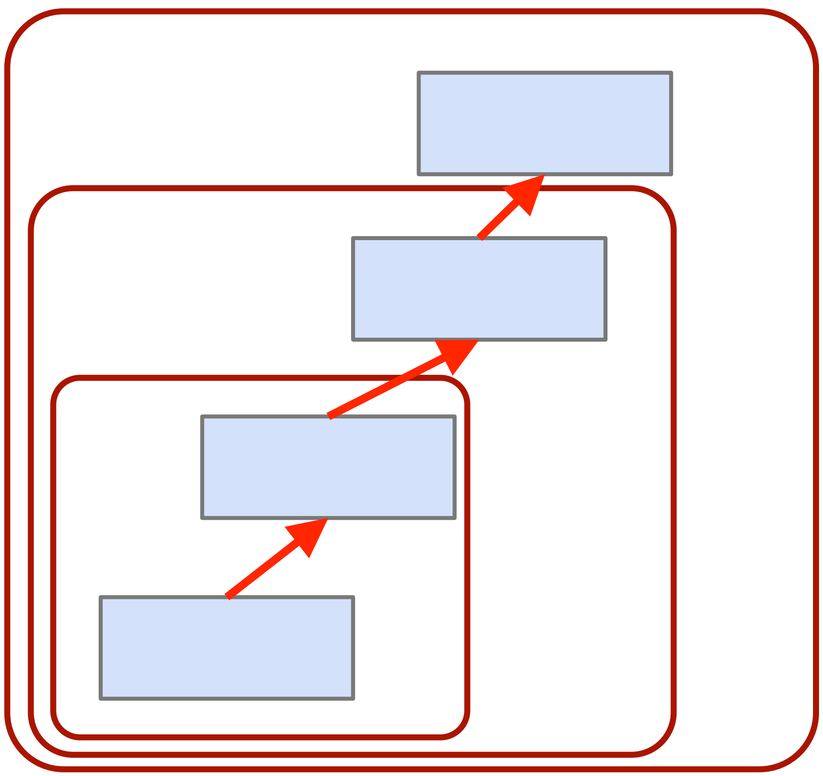


1. Call **FindSet(x)** and **FindSet(y)** to locate the representatives, **O(h)**
2. Let one tree's root point to the other tree's root, **O(1)**

Benchmarking: runtime

The worst-case sequence of **m** operations.
(with **FindSet** being the bottleneck)

m/4 MakeSets, **m/4 - 1** Union, **m/2 + 1** FindSet



Total cost in worst-case sequence :

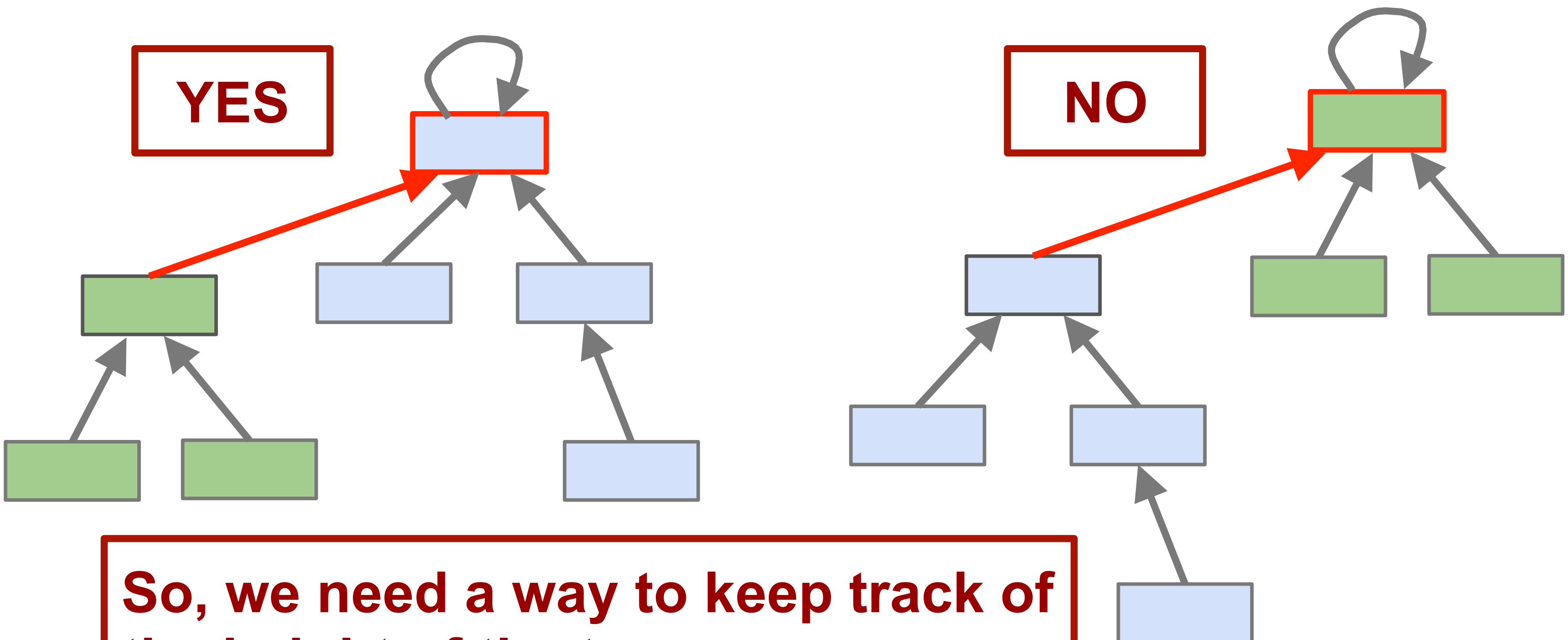
$\Theta(m^2)$

(each FindSet would take up to m/4 steps)

Trees with union-by-rank

Intuition

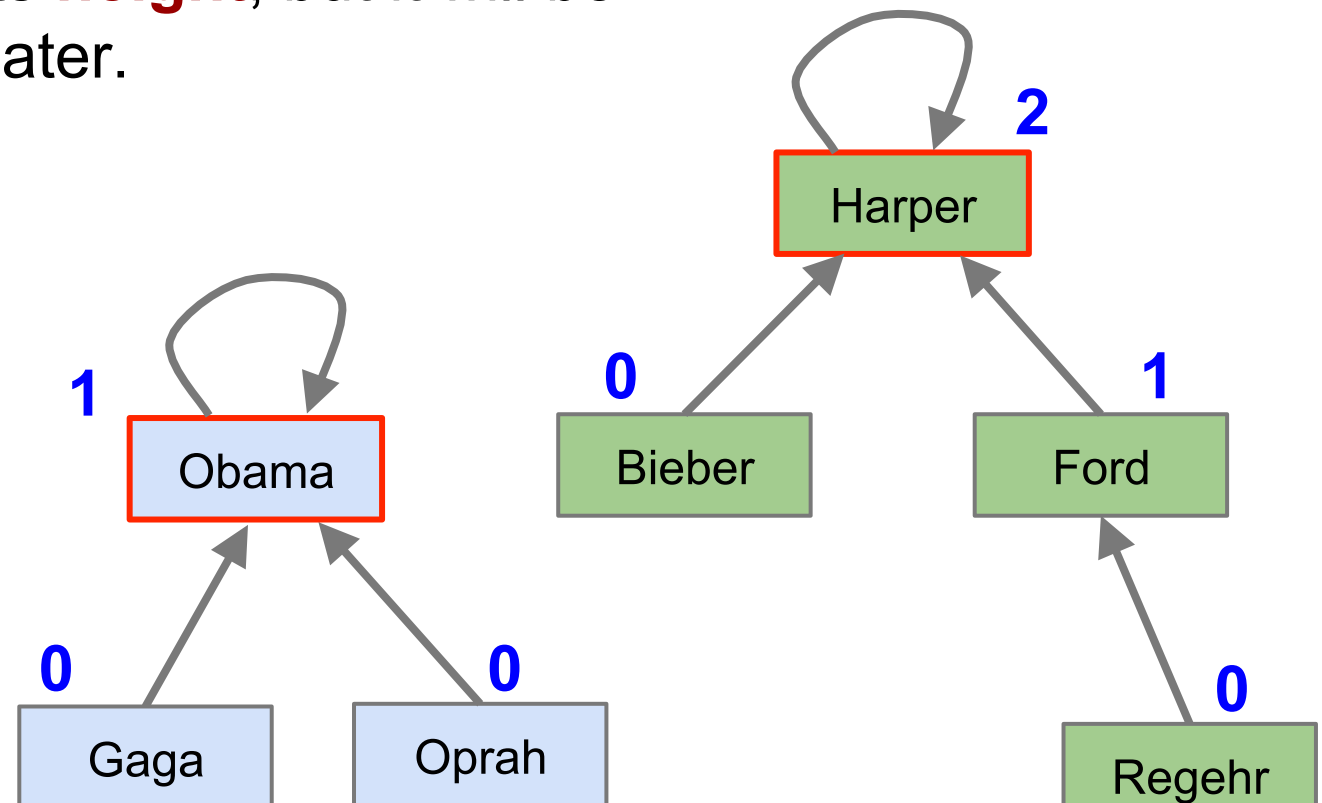
- ➔ FindSet takes **O(h)**, so the **height** of tree matters
- ➔ To keep the unioned tree's height small, we should let the **taller** tree's root be the root of the unioned tree



So, we need a way to keep track of the height of the tree

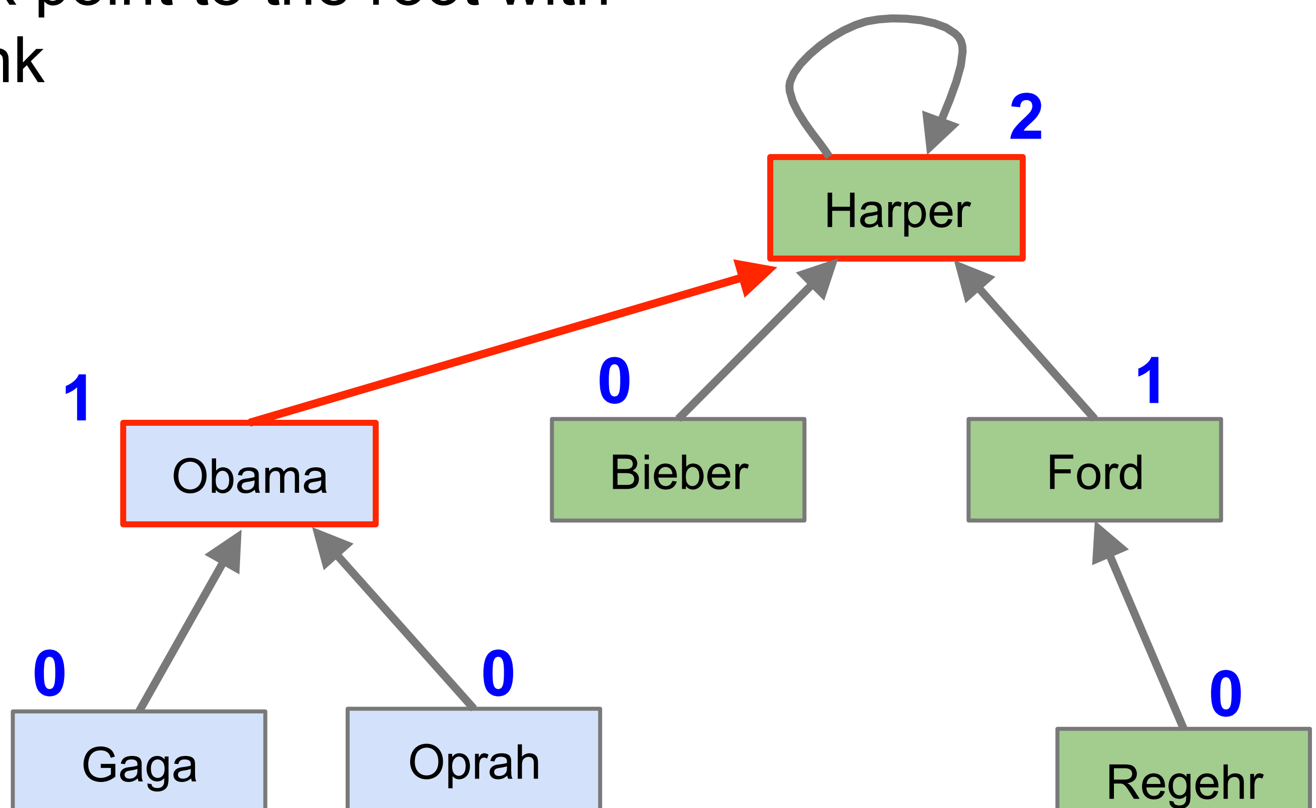
Each node keeps a rank

For now, a node's **rank** is the same as its **height**, but it will be **different** later.



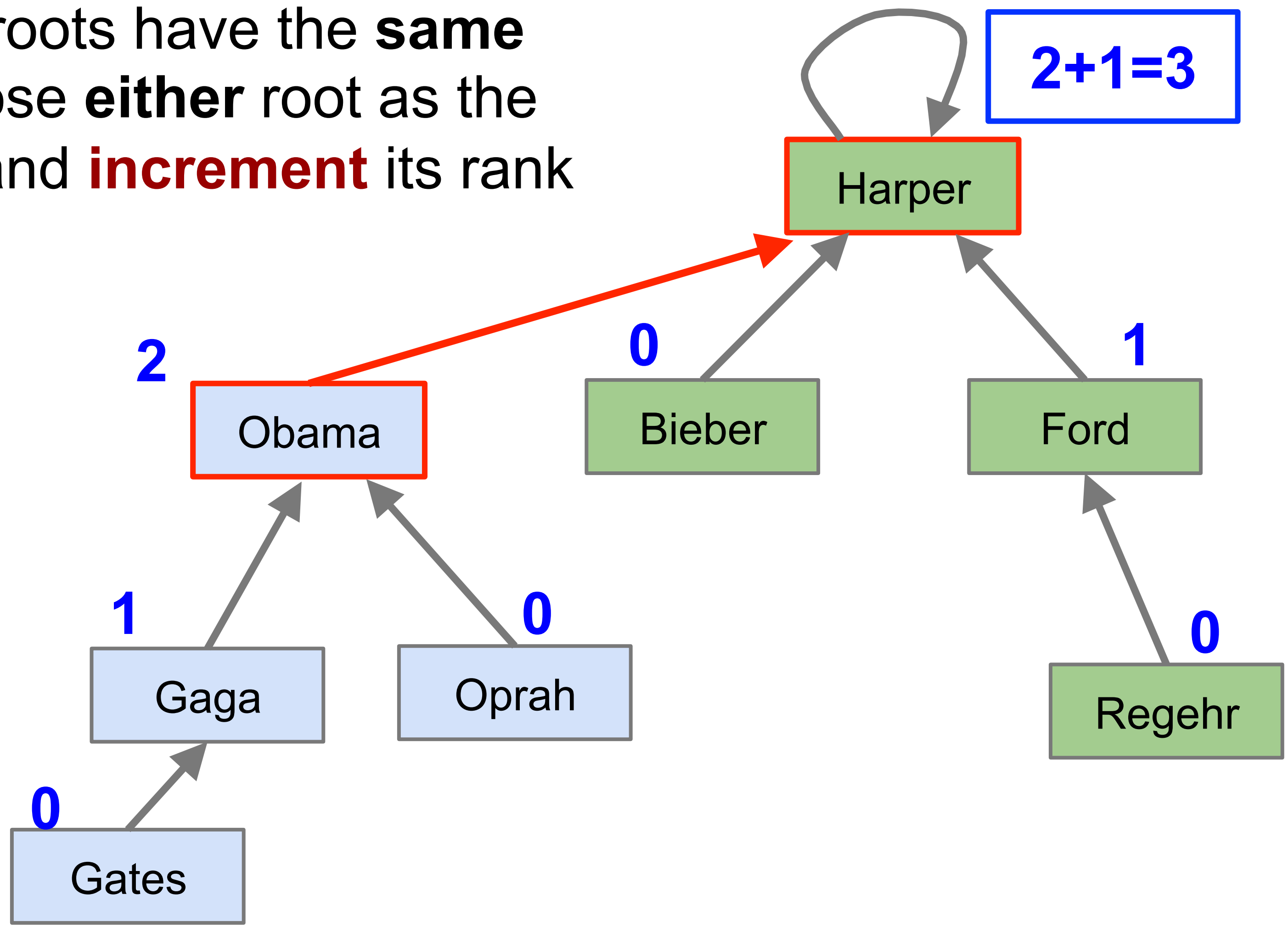
Each node keeps a rank

When **Union**, let the root with **lower** rank point to the root with **higher** rank



Each node keeps a rank

If the two roots have the same rank, choose either root as the new root and increment its rank



Benchmarking: runtime

It can be proven that, a tree of **n** nodes formed by **union-by-rank** has height at most **log n,**
which means **FindSet** takes **O(log n)**

So for a sequence of **m/4** MakeSets, **m/4 - 1** Union, **m/2 + 1** FindSet operations,
the total cost is **O(m log m)**

Rank of a tree with **n** nodes is at most **log n**, i.e., **$r(n) \leq \log n$**

Proof:

Equivalently, prove **$n(r) \geq 2^r$**

Use **induction** on **r**

Base step: if r = 0 (single node), n(0) = 1, TRUE

Inductive step: assume $n(r) \geq 2^r$

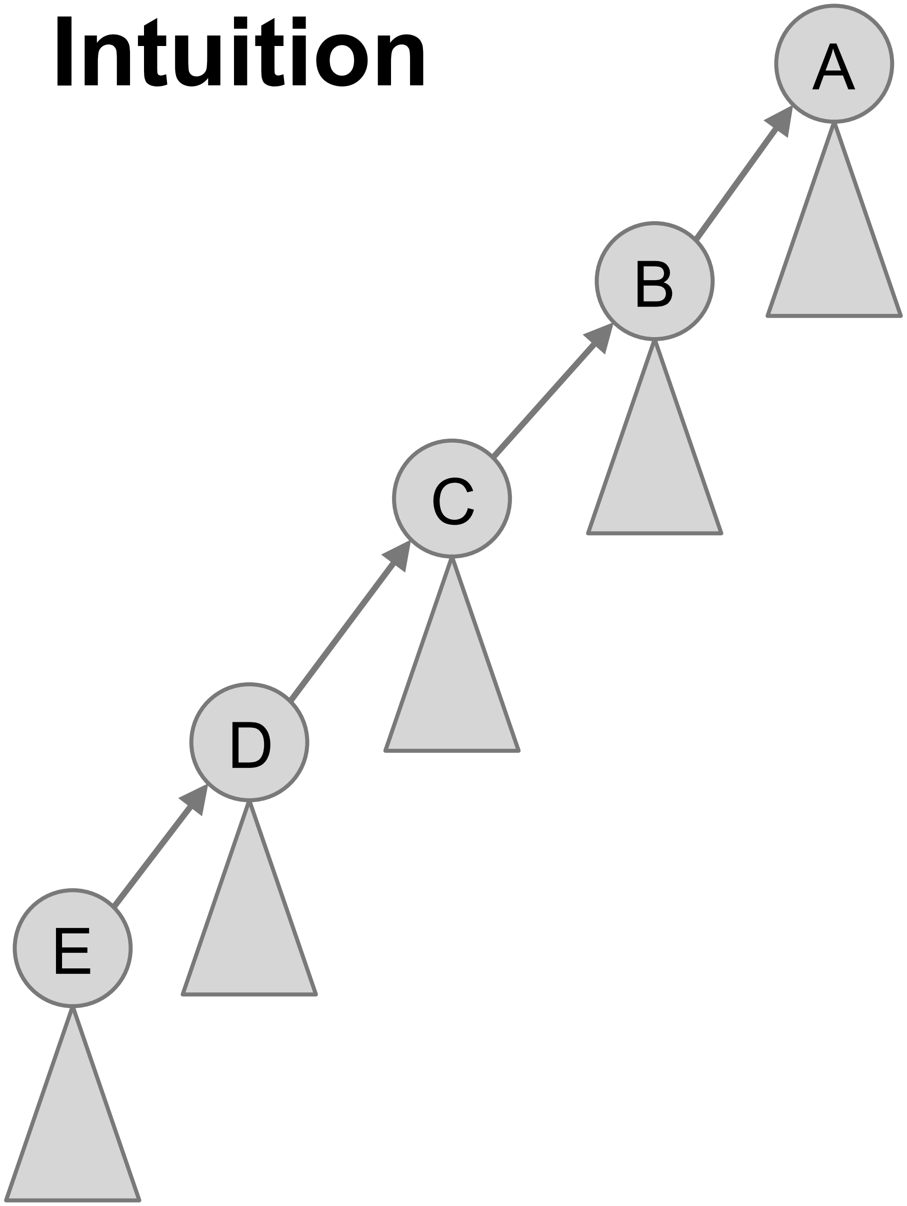
➔ a tree with root rank r+1 is a result of unioning two trees with root rank r, so

➔ **$n(r+1) = n(r) + n(r) \geq 2 \times 2^r = 2^{r+1}$**

➔ Done.

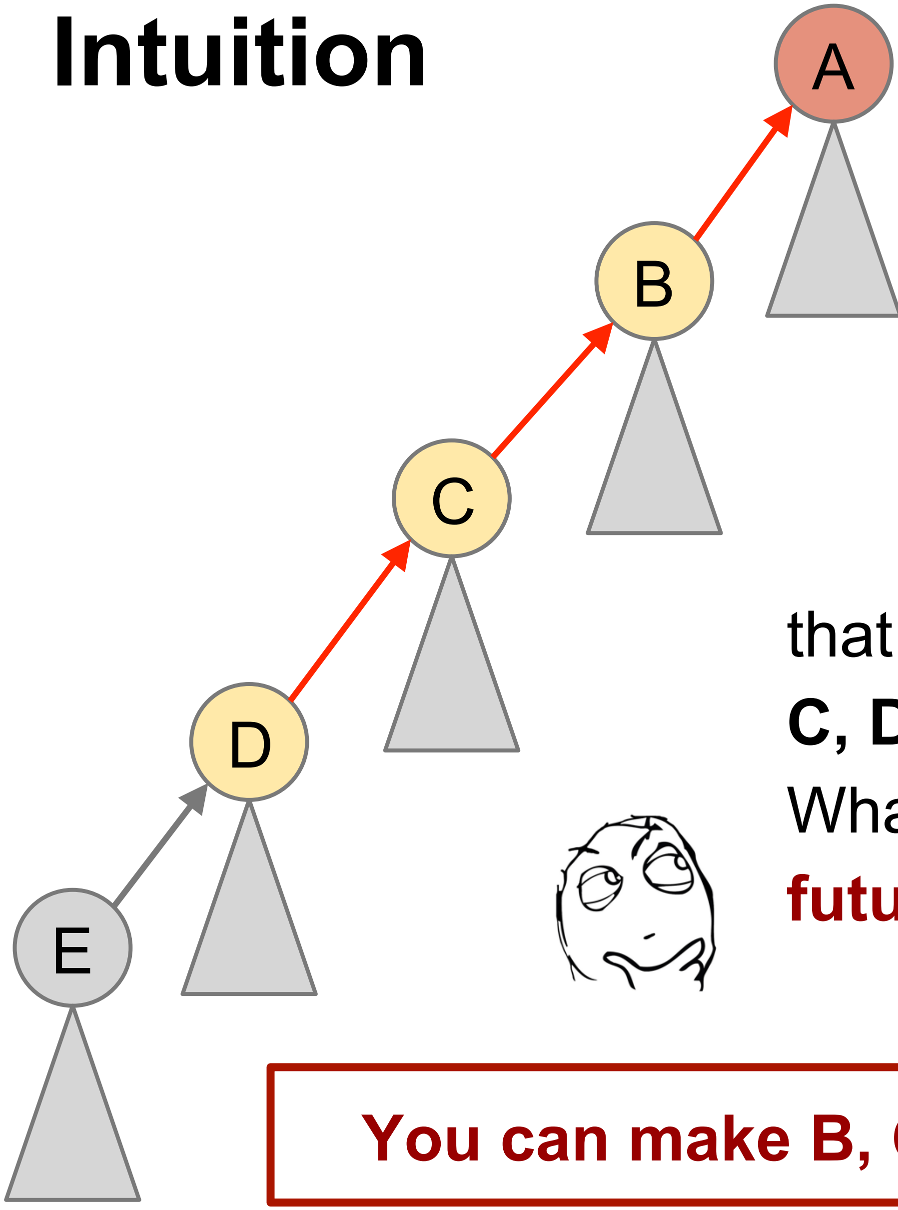
Trees with
path compression

Intuition



Now I do a
FindSet(D)

Intuition



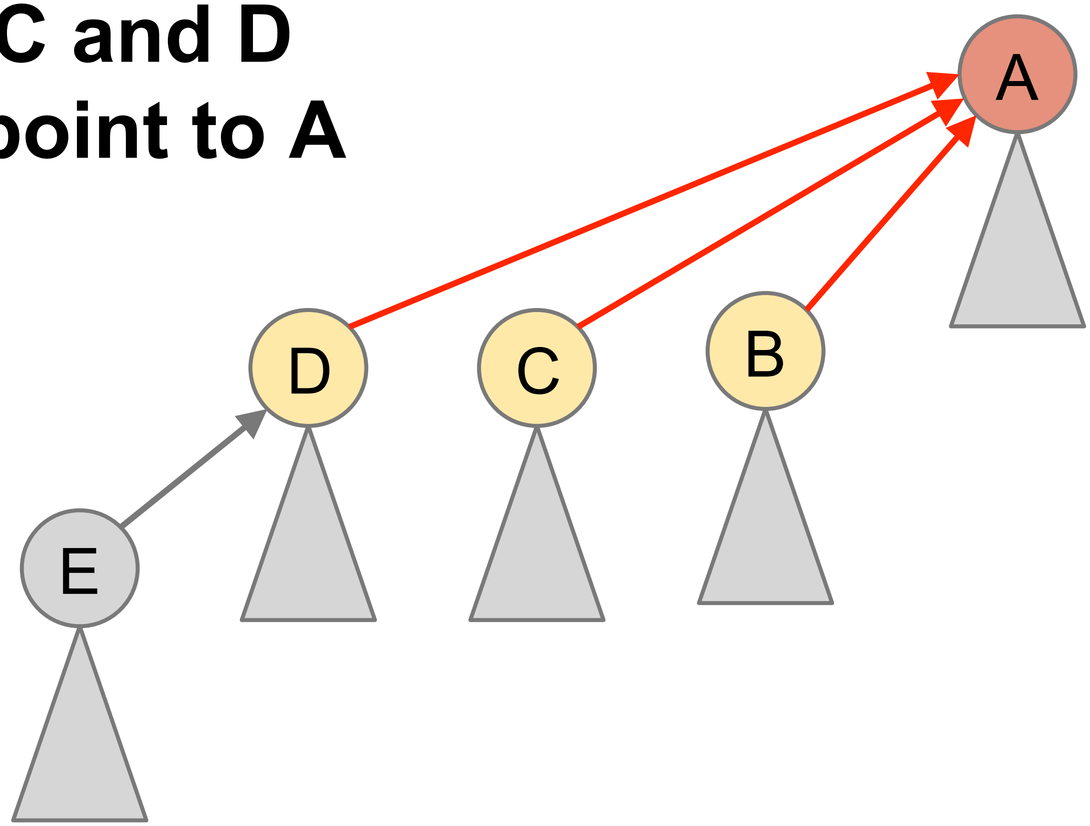
Now I do a
FindSet(D)

On the way of finding **A**, you visit **D**, **C**, **B** and **A**.

that is, now you have access to **B**, **C, D** and the root **A**.
What **nice** things can you do for **future** **FindSet** operations?

You can make B, C and D super close to A!

Make B, C and D
directly point to A



In other words, the path D→C→B→A is "compressed".

Extra cost to FindSet: at most **twice** the cost, so does not affect the order of complexity

Benchmark: runtime

Can be prove: for a sequence of operations with **n** MakeSet (so at most **n-1** Union), and **k** FindSet, the worst-case total cost of the sequence is in

$$\Theta\left(n + k \cdot \left(1 + \log_{2+\frac{k}{n}} n\right)\right)$$

So for a sequence of **m/4** MakeSets, **m/4 - 1** Union, **m/2 + 1** FindSet, the worst-case total cost is in **Θ(m log m)**

Ways of implementing Disjoint Sets

1. Circularly-linked lists $\Theta(m^2)$
2. Linked lists with extra pointer $\Theta(m^2)$
3. Linked lists with extra pointer and with union-by-weight $\Theta(m \log m)$
4. Trees $\Theta(m^2)$
5. Trees with union-by-rank $\Theta(m \log m)$
6. Trees with path-compression $\Theta(m \log m)$

Benchmark:

Worst-case total cost of a sequence of **m** operations (MakeSet or FindSet or Union)

Can we do better than $\Theta(m \log m)$?

U. B. R.

P. C.

Trees with
union-by-rank
and
path compression

How to combine union-by-rank and path compression?

- ➔**Path compression** happens in the **FindSet** operation
- ➔**Union-by-rank** happens in the **Union** operation (outside **FindSet**)
- ➔So they don't really interfere with each other, simply use them both!

Pseudocodes

Complete code using both union-by-rank and path compression

```
MakeSet(x):
  x.p ← x
  x.rank ← 0
```

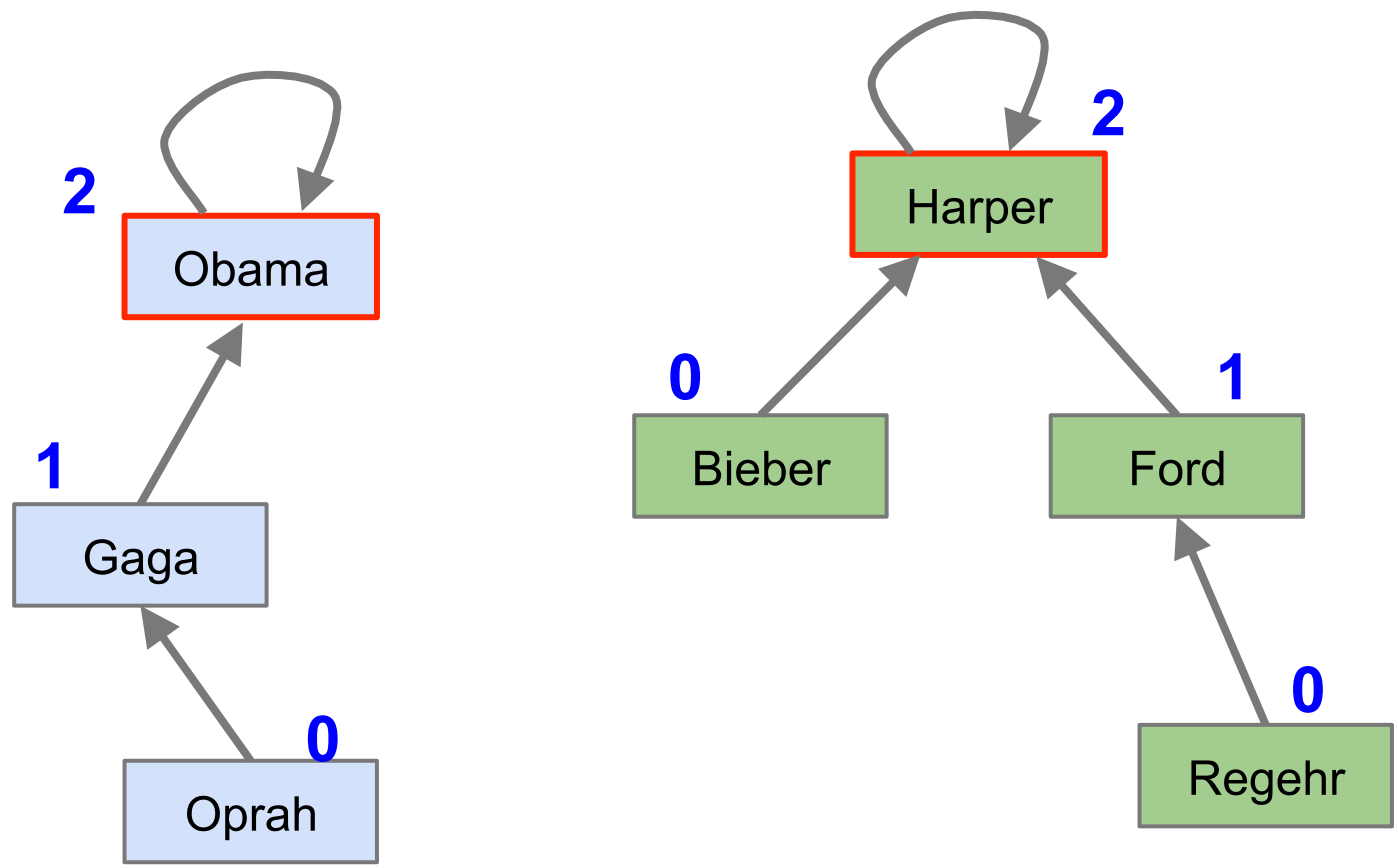
```
FindSet(x):
  if x ≠ x.p:  # if not root
     x.p ← FindSet(x.p)
  return x.p
```

```
Union(x, y):
  Link(FindSet(x), \
       FindSet(y))
```

```
Link(x, y):
  if x.rank > y.rank:
    y.p ← x
  else:
    x.p ← y
    if x.rank = y.rank:
      y.rank += 1
```

Exercise

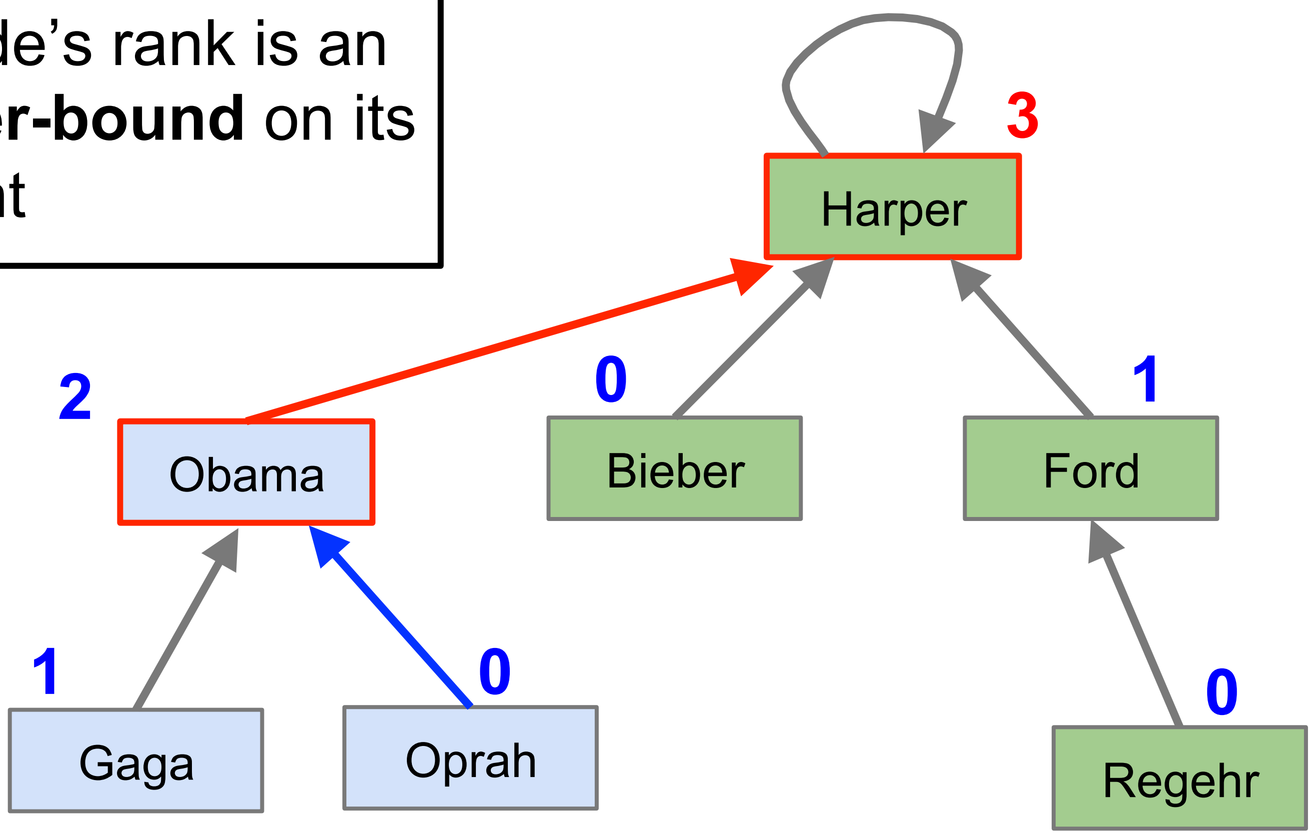
Draw the result after **Union(Oprah, Ford).**
using both union-by-rank and path compression



Note: **rank ≠ height**

because path compression does NOT maintain height info

a node's rank is an **upper-bound** on its height



Benchmark: runtime

Can be proven: for a sequence of **m** operations with **n** MakeSet (so at most **n-1** Union), worst-case total cost of the sequence is $O(m \log^* n)$

Note: $\log^* n$ is equal to the number of times the log function must be iteratively applied so that the result is at most 1

Example: $\log_2(2^{256}) = 256$

$\log_2(256) = 8$

$\log_2(8) = 3$

$\log_2(3) < 1.6$

$\log_2(1.6) < 1$

So $\log^*(2^{256}) = 5$, and $\log^*(2^m) = 6$, where $m=2^{256}$

Since $\log^*$ n is so slowly growing it is like a constant.

Sketch of Analysis

Lemma: A node v which is the root of a subtree of rank r has at least 2^r nodes

(We already proved this.)

Lemma: If there are n nodes, the maximum number of nodes of rank r is $n/2^r$

Each node which is the root of a subtree with rank r has at least 2^r nodes. So maximum is $n/2^r$ rank r root notes, each with 2^r children

Sketch of Analysis

Group the nodes into at most $\log^* n$ buckets:

Bucket 0: nodes of rank 0
Bucket 1: nodes of rank 1
Bucket 2: nodes of rank 2-3
Bucket 3: nodes of rank 4-16
...
Bucket B: nodes of rank $[r, 2^r -1] = [r, R-1]$
Bucket B+1: nodes of rank $[R, 2^R -1]$

Note: the maximum number of elements in bucket containing nodes of rank $[R, 2^R -1]$ is at most $n/2^R + n/2^{R+1} + \ldots + n/2^{2^\wedge R -1} \leq 2n/2^R$

Sketch of Analysis

Let F be the list of all m FindSet operations performed

Then total cost of m finds is $T_1 + T_2 + T_3$
Where T_1 = links pointing to root that are traversed
T_2 = links traversed between nodes in different buckets
T_3 = links traversed between nodes in same bucket

- $T_1 \leq m$ since each FindSet traverses one link to root
- $T_2 \leq m \log^* n$ since there are only $\log^* n$ buckets
- It is left to bound T_3

Sketch of Analysis

It is left to bound T_3

Suppose we are traversing from u to v, where u,v are both in the bucket of nodes with rank $[B, 2^B -1]$
Since the rank is always increasing as we follow a path to a root, the number of links going from u to v is at most $2^B -1 - B \leq 2^B$

Thus $T_3 \leq \Sigma_B\, 2^B\; 2n/2^B \leq 2n \log^* n$

Thus $T_1 + T_2 + T_3 = O(m \log^* n)$

Summary of worst case runtime for m operations, n elements)

1. Circularly-linked lists $\Theta(m^2)$
2. Linked lists with extra pointer $\Theta(m^2)$
3. Linked lists with extra pointer and with union-by-weight $\Theta(m \log m)$
4. Trees $\Theta(m^2)$
5. Trees with union-by-rank $\Theta(m \log m)$
6. Trees with path compression $\Theta(m \log m)$
7. Trees with union-by-rank and path compression $O(m \log^* n)$

Next week

- ➔Lower bounds
- ➔Review for final exam