Disjoint sets

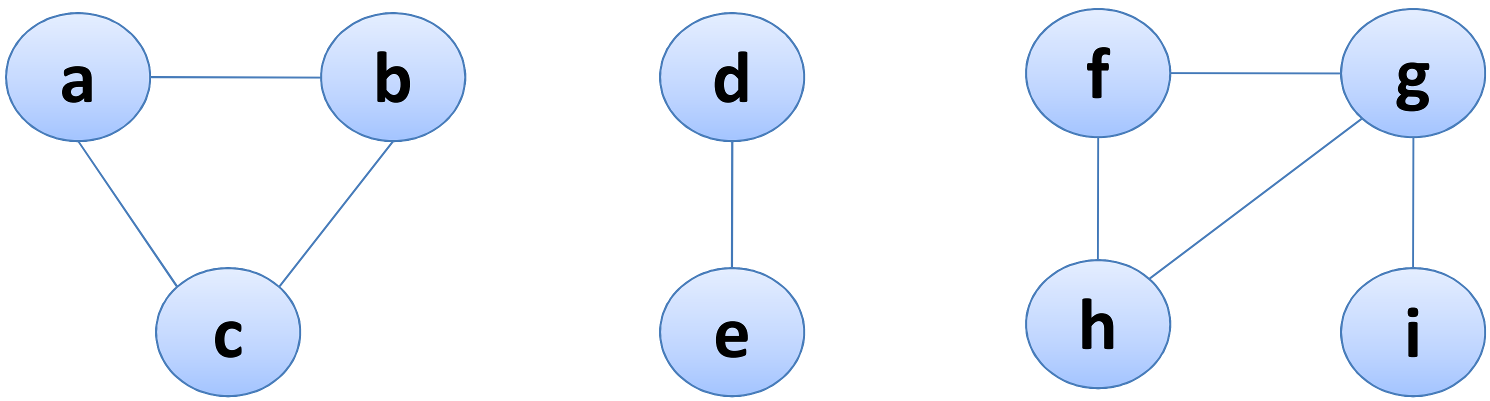
Disjoint set ADT

- Maintains a collection $\mathcal{S} = \{S_1, \ldots, S_k\}$ of disjoint sets
- Each set is identified by a representative, which is an element of the set
- **Operations**:
  - MAKE-SET(x): creates a new set containing only x, and makes x the representative
  - FIND-SET(x): returns the representative of x's set
  - UNION(x, y): merges the sets containing x and y, and chooses a new representative
- Note: No duplicate elements are allowed!

Disjoint set application

- **Example:** Determine whether two nodes are in the same connected component of an undirected graph
- **Connected component:** a maximal subgraph such that any two vertices are connected to each other by a path

Disjoint sets for connected components



- How do you use disjoint sets to solve this problem?

Disjoint sets for connected components

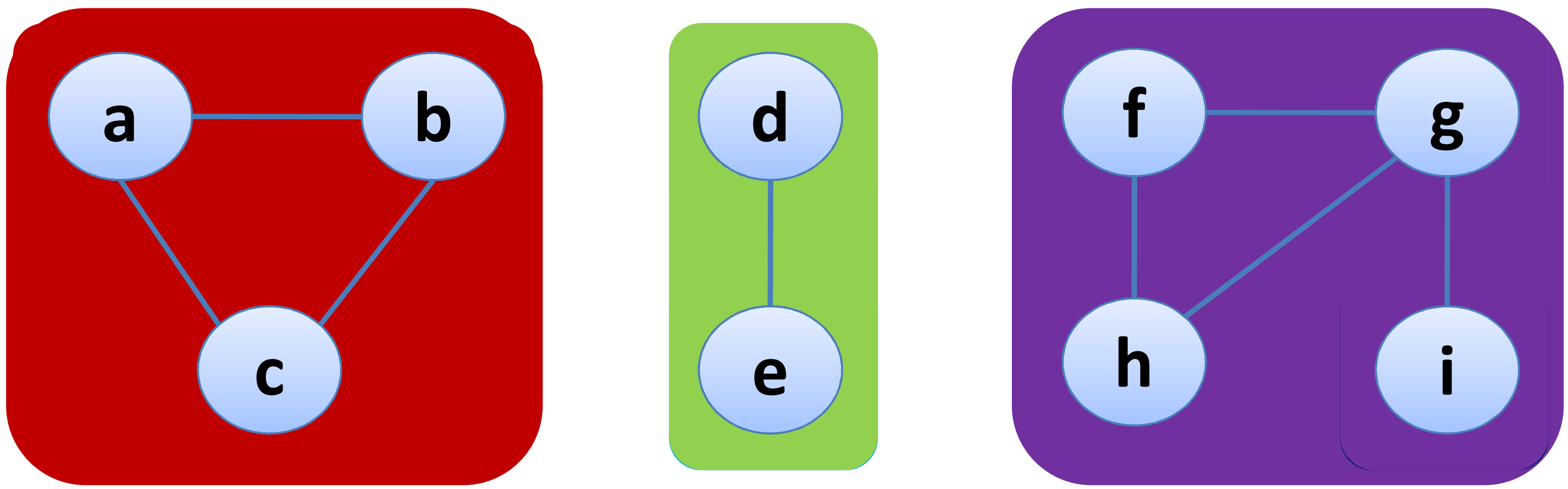
Connected-Components(G):

for each vertex v ∈ V[G] do
    MAKE-SET(v)

for each edge (u,v) ∈ E[G] do
    if FIND-SET(u) ≠ FIND_SET(v) then
        UNION(u,v)

Disjoint sets for connected components

Connected components:



Process the edges:

(a, b) (f, g) (g, i) (d, e) (c, b) (a, c) (f, h) (h, g)

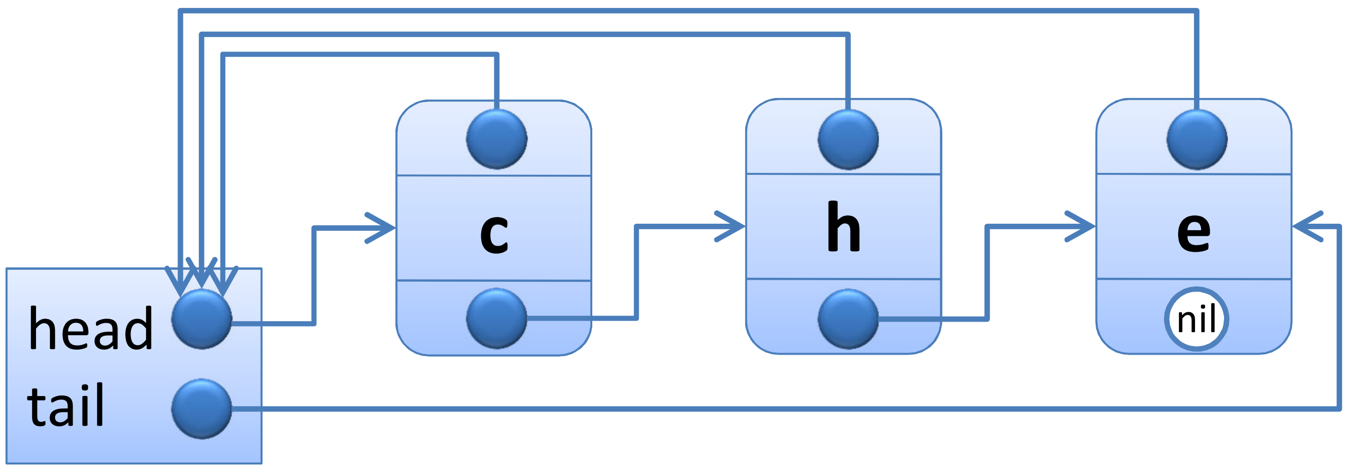
Disjoint sets for connected components

Same-Component(u,v):

```
if FIND-SET(u) = FIND-SET(v) then
    return True
else
    return False
```

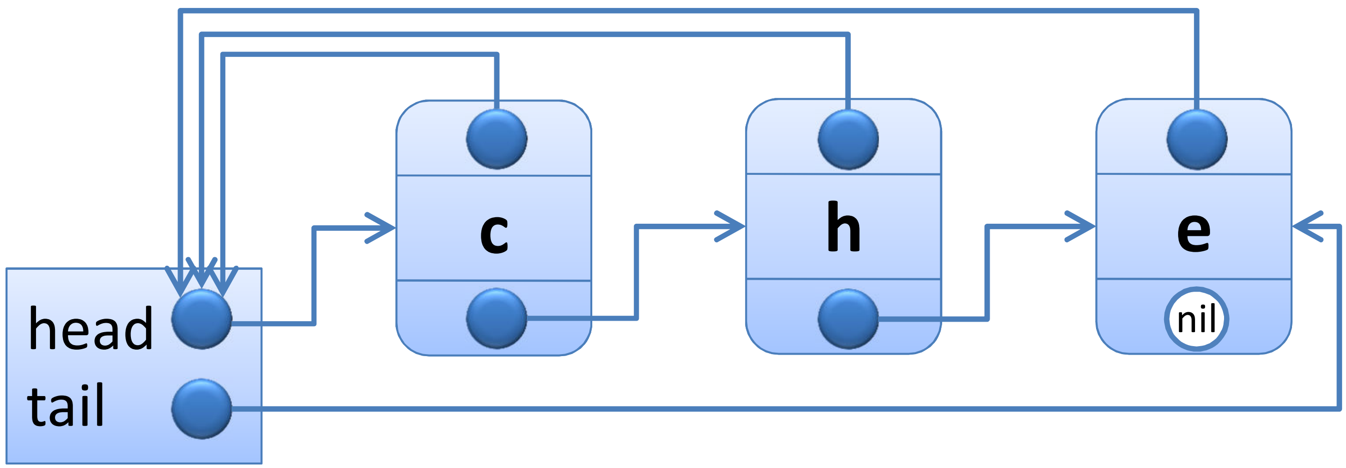
Linked list implementation of Disjoint Sets

Implementing a single set



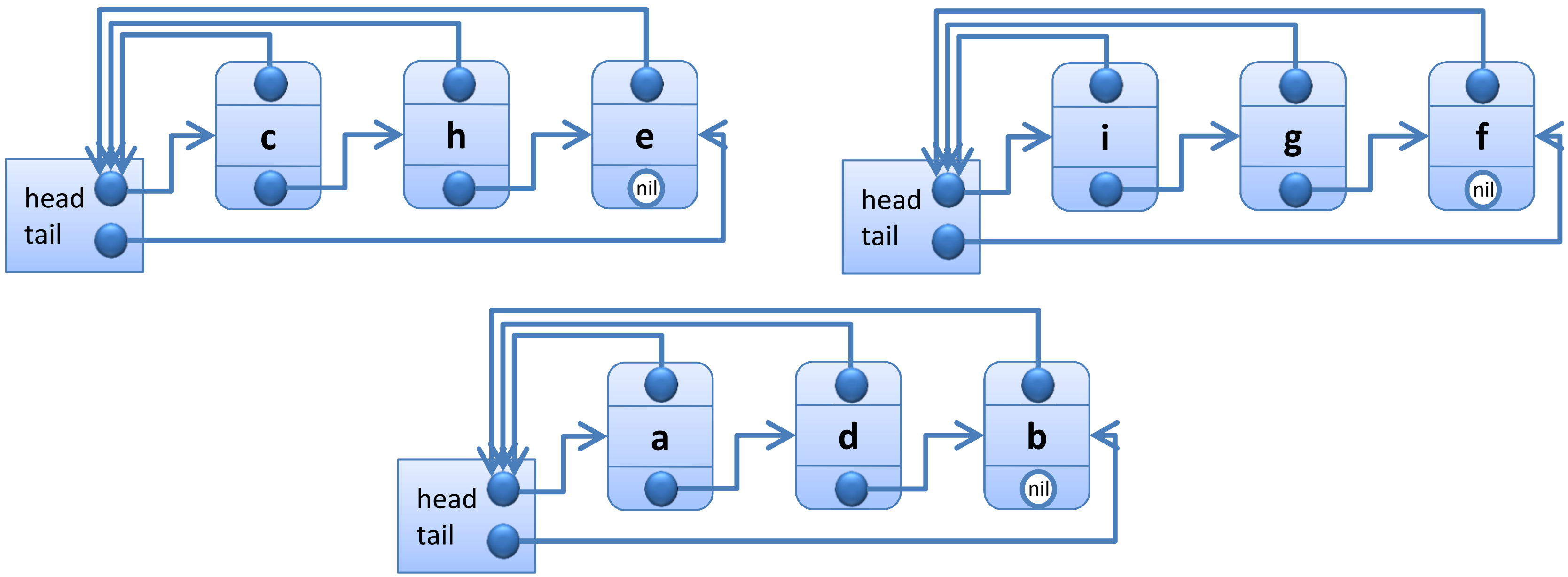
- The **representative** of the set **=** the **first element** in the list
- Other elements may appear in any order in the list

Implementing a single set



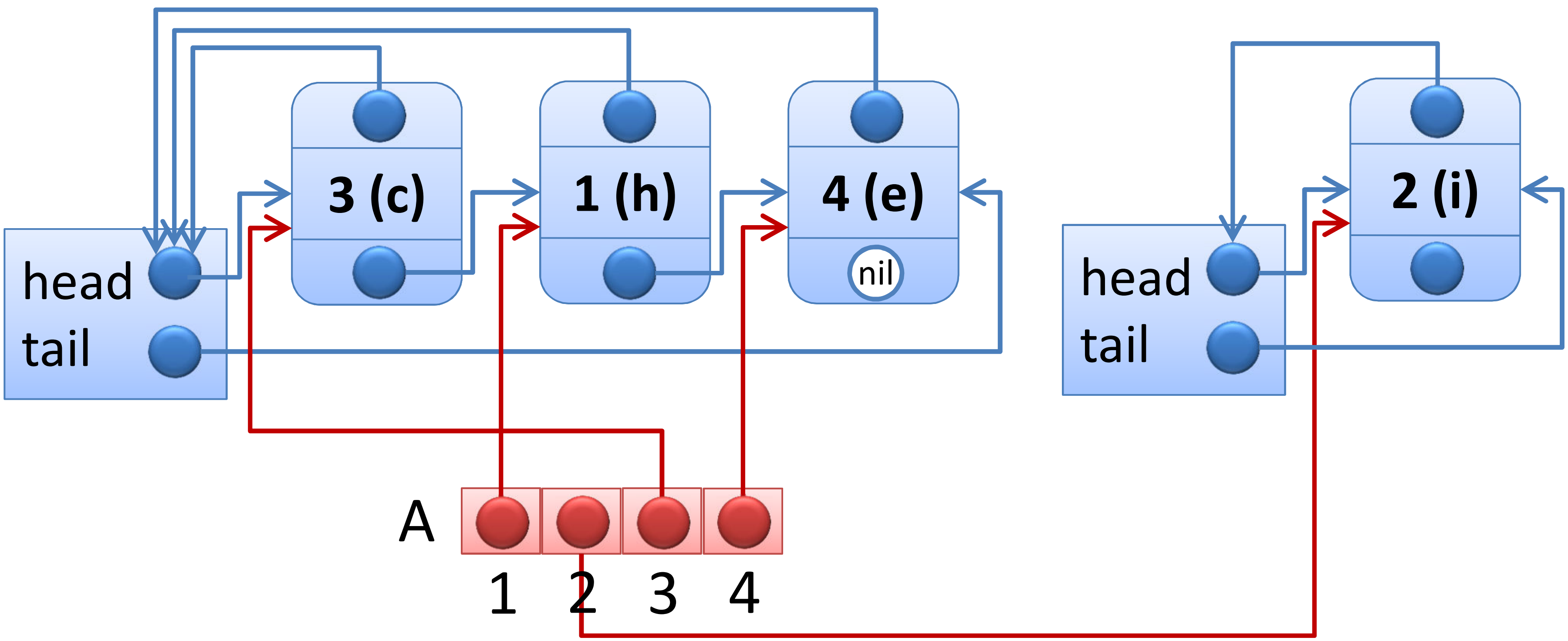
- A node contains pointers to:
  - The next element
  - Its representative
- + each set has pointer to **head** and **tail** of its list

Implementing the data structure



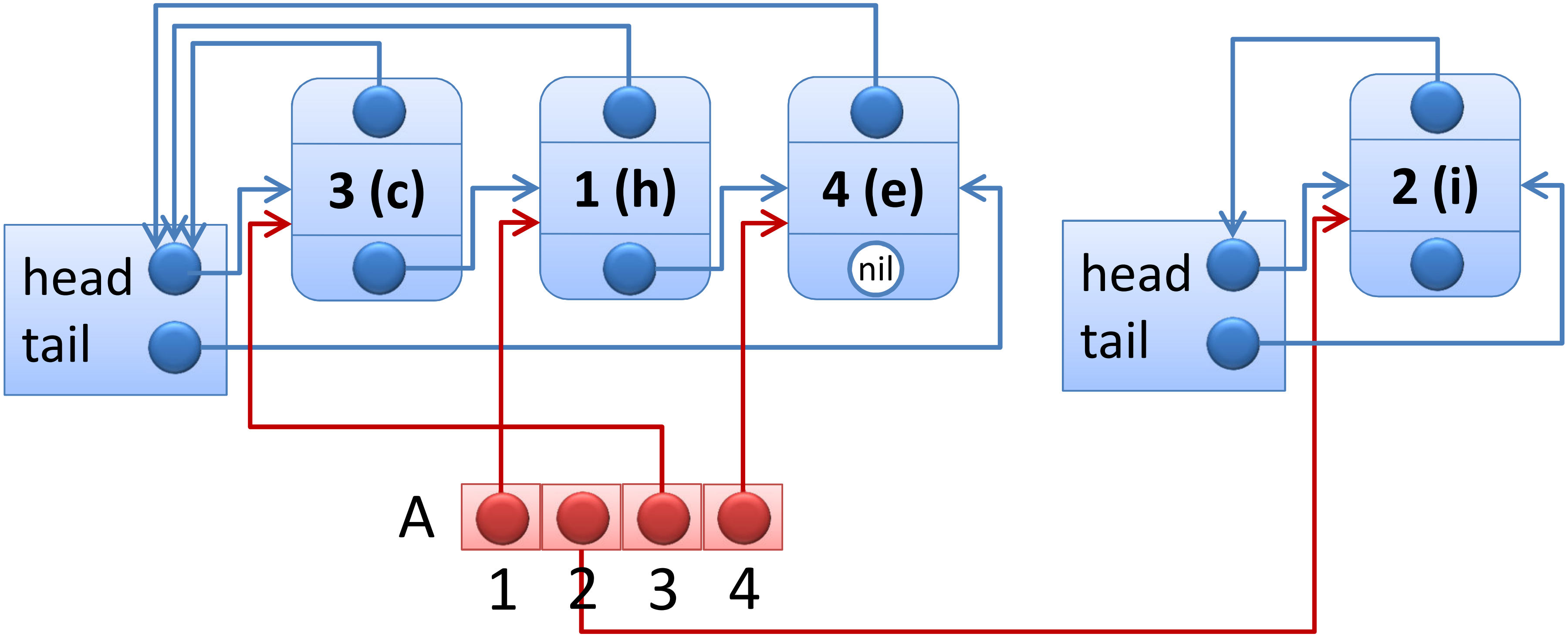
- Collection of several sets, each a linked list
- How do we do FIND-SET(h)?
  - Do we have to search through every list?

Implementing the data structure



- In practice, we rename the elements to **1..n**, and maintain an array **A** where **A[i]** points to the list element that represents **i.**
- Now, how do we do FIND-SET(**3**)?

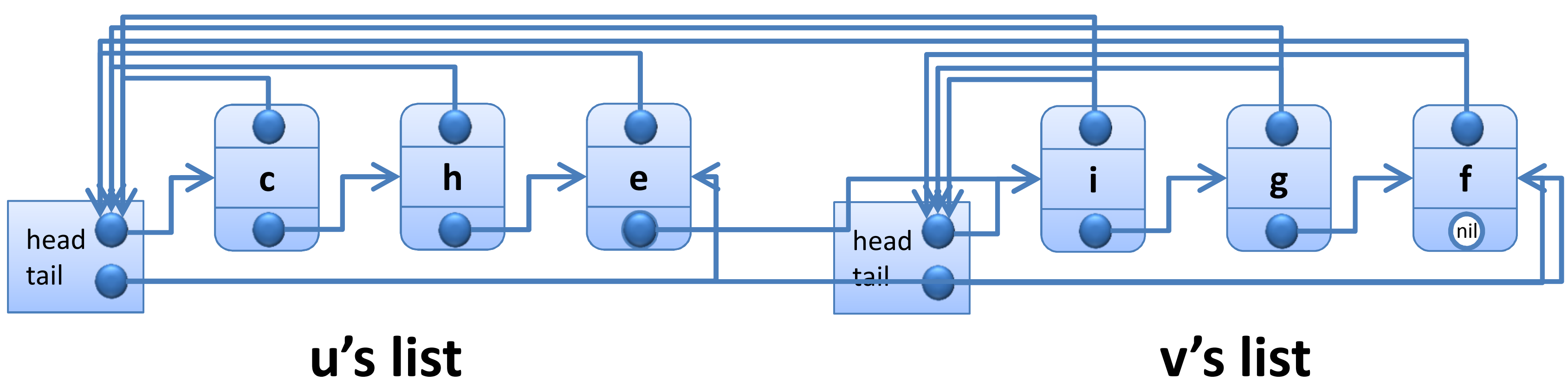
Implementing the data structure



- Harder question: how about FIND-SET(**e**)?
  - When you rename h->1, i->2, c->3, e->4 you store these mappings in a *dictionary D*.
  - Later, you can call D.get(e) to retrieve the value 4.
  - So, you call FIND-SET(D(e)), which becomes FIND-SET(4).

Naïve implementation of Union(u,v)

- Append v's list onto the end of u's list:
  - Change u's tail pointer to the tail of v's list = $\boldsymbol{\theta}$**(1)**
  - Update representative pointers for all elements in the v's list = $\boldsymbol{\theta}$**(|v's list|)**
    - Can be a long time if |v's list| is large!
    - In fact, **n-1** Unions can take $\boldsymbol{\theta(n^2)}$



Weighted-union heuristic for Union(u,v)

- Similar to the naïve Union but uses the following rule/heuristic for joining lists:
- **Append the smaller list onto the longer one** (and break ties arbitrarily)

- Does this help us do better than $O(n^2)$?
- Worst-case time for a **single** Union(u,v) – **NO**
- Worst-case time for a **sequence** of **n** Union operations – **YES**

Weighted-union running time analysis

- We will analyze the running times of disjoint-set data structures in terms of two parameters:
  - **n** = the number UNION operations
  - **m** = the number of FIND-SET operations

Weighted-union running time analysis

- **Theorem:**
  - Suppose a disjoint set implemented using linked-lists and the **weighted-union heuristic** initially contains **n singleton sets**.
  - Performing a sequence of **n** UNIONs and **m** FIND-SETs takes **O(m + n lg n)** time.

- **Compare:** for the naïve Union implementation, **n** UNIONs and **m** FIND-SETs takes **$O(m + n^2)$** time.

Weighted-union running time analysis

- Let's prove the easy part first
- **FIND-SET** operations:
  - each FIND-SET operations takes **O(1)** time
  - so **m** FIND-SET operations takes **O(m)** time

Weighted-union running time analysis

- Now the harder part – **UNION** operations:
- What takes time in a UNION operation?
  - Update **head** and **tail pointers,** a single **next pointer**, and a bunch of **representative pointers**.
  - Representative pointers take time.
  - Everything else is O(1).
- How many times can an element's representative pointer be updated?

Weighted-union running time analysis

- Fix an element **x**.
- If **x** is in a set **S** and its representative pointer changes, then **S** is being attached to another set with size at least **|S|**.
- After the union, x's set contains at least 2|S| elements.
  - Initially, x's set contains 1 element (itself).
  - After x's set is UNIONed once, it has size at least 2.
  - After x's set is UNIONed twice, it has size at least 4.
  - After x's set is UNIONed thrice, it has size at least 8.
  - …
  - After x's set is UNIONed **k** times, it has size at least $\mathbf{2^k}$.

Weighted-union running time analysis

⇨ The total update time for all **n** elements is **O(n lg n)**

*Updating the head and tail pointers takes $\boldsymbol{\theta}$**(1)** per operation, thus total time to update the pointers over at most n UNION operations is $\boldsymbol{\theta}$**(n)**

$$\mathbf{2^k} \le \mathbf{n}$$

apply $\log_2$

$$\mathbf{k} \le \lceil \lg \mathbf{n} \rceil$$

- ⇒ **x**'s representative is updated at most **k** = ⌈lg **n**⌉ times

Weighted-union running time analysis

- Summary:
  - **m FIND-SET** operations take **O(m)**
  - **n UNION** operations take **O(n lg n)**

$\Rightarrow$ The total time of **n UNION**s and **m FIND-SET** operations is **O(m + n log n)**