Disjoint-Sets

(Most of the images and text are excerpted from Cormen et al.)

Where are we?

- Data Structures
  - Stacks & Queues
  - Linked Lists
  - Hash Tables
  - Binary Search Trees & Red Black Trees
  - Etc.

- Design&Analysis Techniques
  - Divide & Conquer
  - Dynamic Programming
  - Greedy Algorithms
  - Amortized Analysis

We return to studying data structures

- Advance Data Structures
  - Disjoint-sets
  - Graphs
  - Etc.

Outline

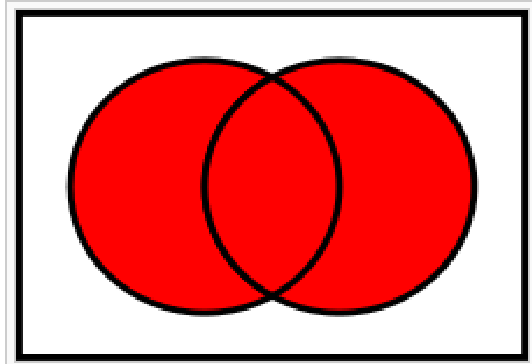
- Background info about Sets
- Disjoint-set data structure & its operations
- Representation of Disjoint-sets
  - Using Linked-Lists
  - Using Rooted Trees

Sets

- **Set:** A set is a collection of distinct objects.
  - A = { 1, 2, 3 }
  - B = { 3, 4, 5}

- **Basic operations:**
  - **Union:** Two sets can be added together.

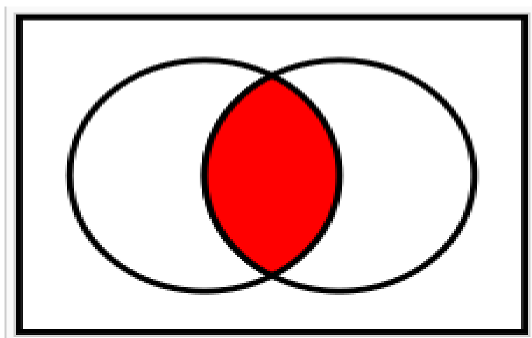
    A U B = {1, 2, 3} U {3, 4, 5} = {1, 2, 3, 4, 5}



The **union** of *A* and *B*, denoted $A \cup B$

  - **Intersection:** Common members of both sets

    $A \cap B$ = {1, 2, 3} $\cap$ {3, 4, 5} = { 3 }



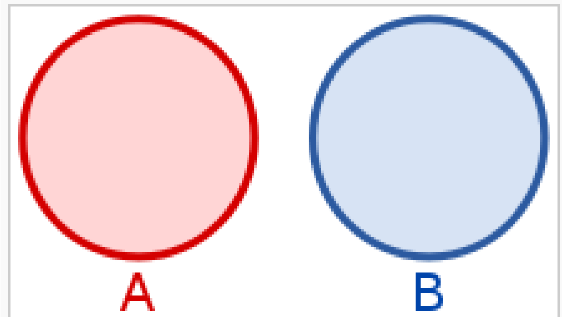
The **intersection** of *A* and *B*, denoted $A \cap B$.

Image source: wikipedia

Disjoint-Sets

- **Disjoint Sets:** Two sets, A and B, are disjoint if they have no element in common.
  - A = {1, 2, 3}
  - B = {7, 8, 9}

$A \cap B = \emptyset$

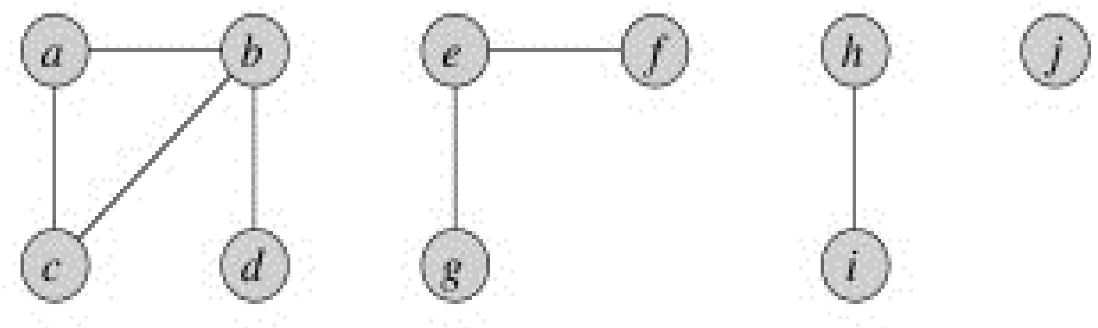


Two disjoint sets.

- Sometimes, we need to group n distinct elements into a collection of disjoint sets.
- Let's first understand the problem, then we will go through the solution.

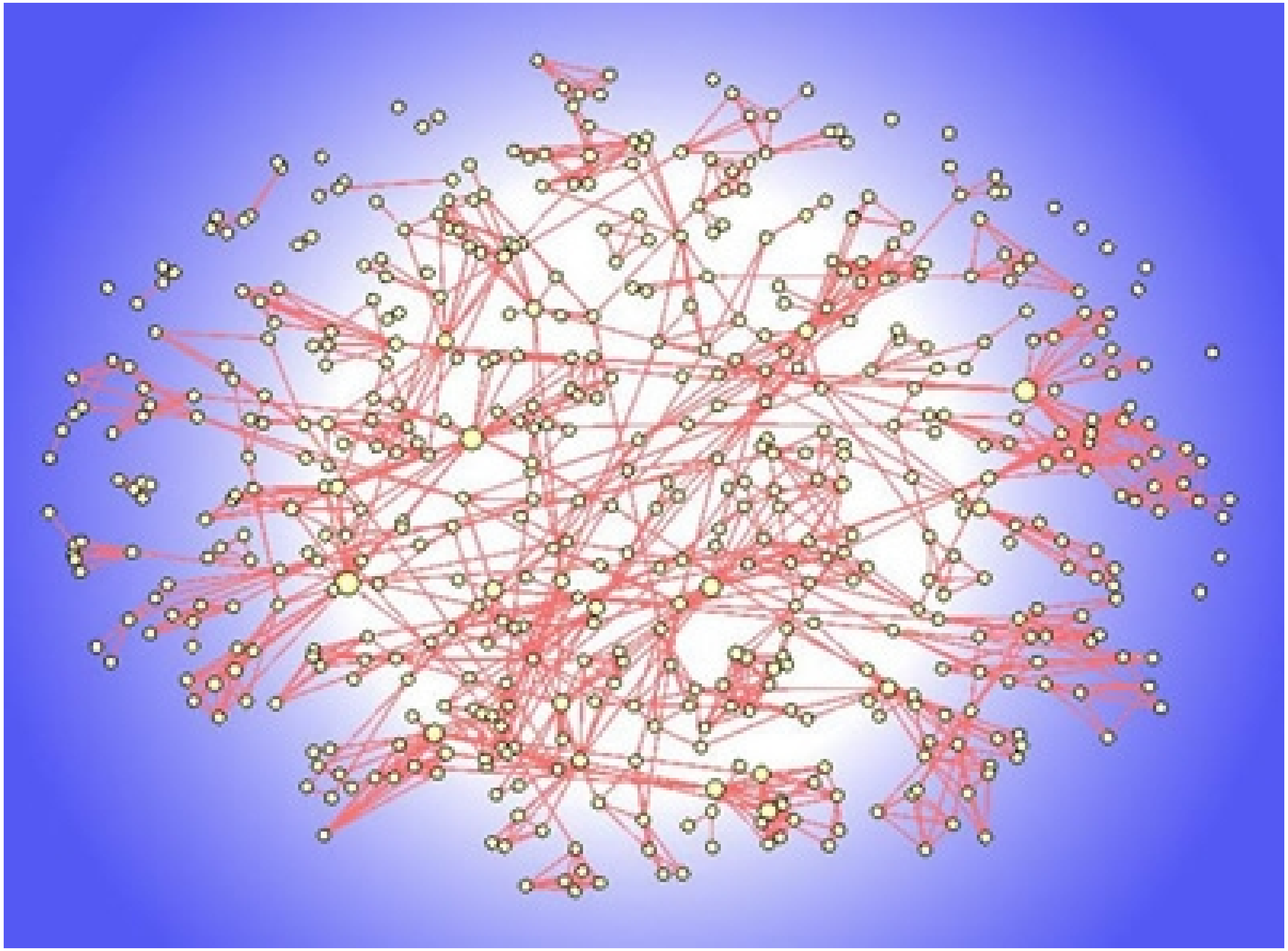
Image source: wikipedia

Problem



- This is an undirected graph with 10 nodes and 7 edges.
- Questions:
  - How many connected-components are there?
  - And what are the members of each component?

Problem

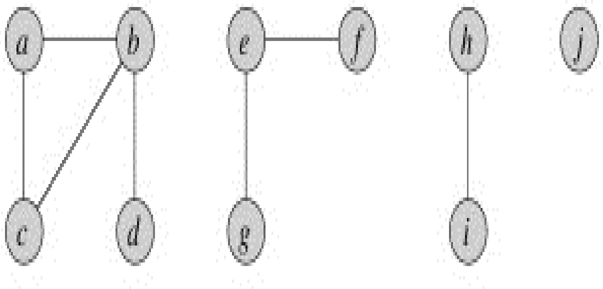


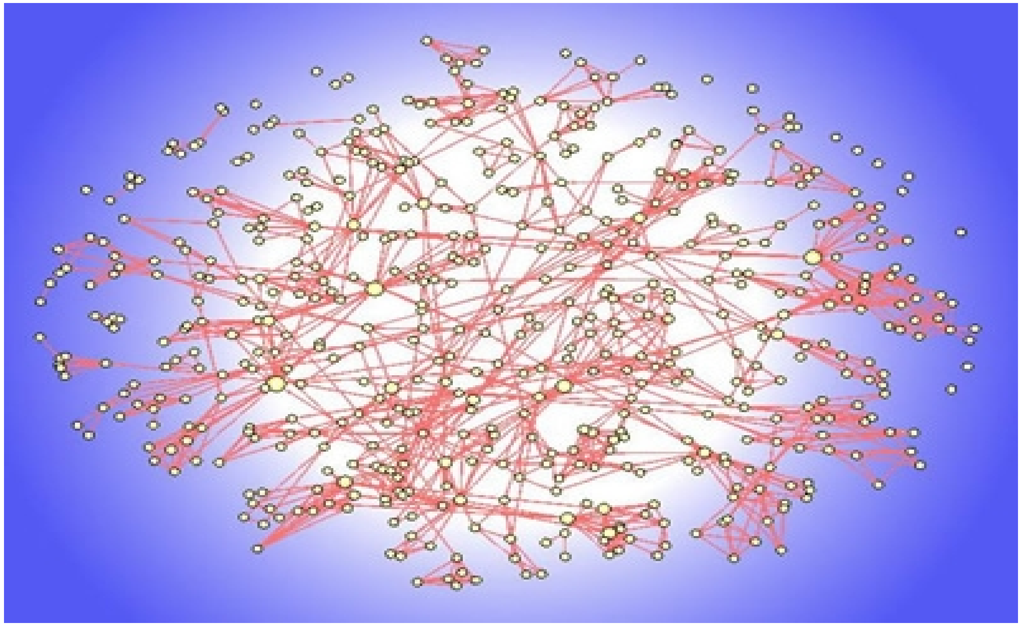
Co-authorship network of 555 scientists

- This is an undirected graph with 555 nodes and ~5000 edges.
- How many connected-components are there? And what are the members of each component?

Image source: http://www.mpi-fg-koeln.mpg.de/~lk/netvis/Huge.html

Problem



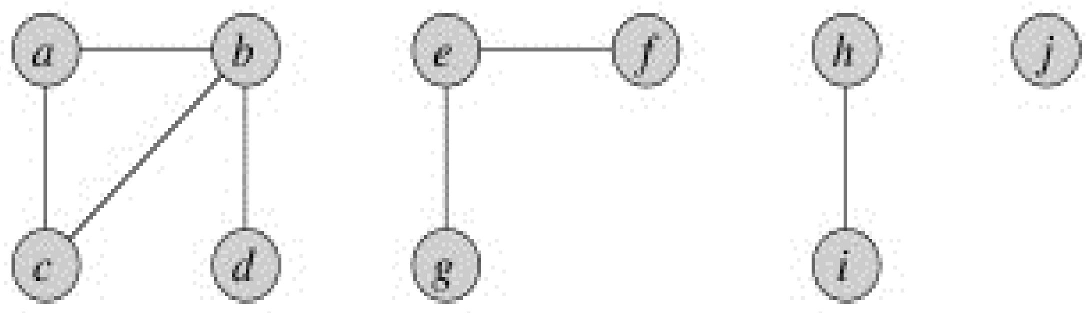


- We need a data structure and operations that can find connected-components on both (simple and complex) graphs in linear time.

- "a list of well-defined instructions", definition of algorithm.

Image source: http://www.mpi-fg-koeln.mpg.de/~lk/netvis/Huge.html

Problem Definition



- We need an algorithm which takes the graph as input and produces the following output.
- Input:

  Vertices: G.V = {a}, {b}, {c}, {d}, {e}, {f}, {g}, {h}, {i}, {j}

  Edges: G.E ={b,d}, {e,g}, {a,c}, {h,i}, {a,b}, {e,f}, {b,c}
- Output: There are 4 connected-components:
  - {a, b, c, d}
  - {e, f, g}
  - {h, i}
  - { j }

Disjoint-Set Data Structure

- A **disjoint-set data structure** maintains a collection of disjoint dynamic sets.
  - $S = \{S_1, S_2, ..., S_k\}$
    - $S_1 = \{1, 2, 3\}$
    - $S_2 = \{5, 6\}$
    - ...
    - $S_k = \{88, 89\}$
- Each set is identified by a representative, a member of the set.
  - It doesn't matter which member is used as the representative.
  - But, if we ask for the representative of a set twice without modifying the set between requests, we should get the same answer.
  - You can specify a rule or you can use the smallest member as the representative.
    - $S_1 = \{1, 2, 3\}$ Representative of S_1 may be 1.
    - $S_2 = \{5, 6\}$ Representative of S_2 may be 5.

Operations of Disjoint-Sets

- Operations of Disjoint-set data structure:
  - MAKE-SET(x): creates a new set whose only member is x. Make a new set $S_i = \{x\}$, and add S_i to S.
    - Initial vertices: 1, 2, 4, 5
    - $S_1 = \{1\}$ $S_2 = \{2\}$ $S_4 = \{4\}$ $S_5 = \{5\}$
    - $S = \{S_1, S_2, S_4, S_5\}$
  - UNION(x, y): unites the dynamic sets that contain x and y into a new set.
    - $S_1 = \{1, 2, 3\}$ $S_3 = \{4\}$ and $S_5 = \{5, 6\}$
    - UNION(2, 6) = S_1 U S_5 = {1, 2, 3, 5, 6}
  - FIND-SET(x): finds the set which contains x, and returns a pointer to the representative of that set.
    - $S_1 = \{1, 2, 3\}$ and $S_5 = \{5, 6\}$
    - FIND-SET(2) = 1 (rep of S_1) FIND-SET(6) = 5 (rep. of S_5)

Analogy

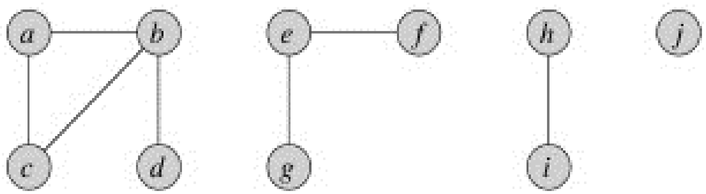
- Disjoint-set is a data structure too. We can make analogy with other data structures.

DATA STRUCTURE	**OPERATIONS**
Stacks	Push(x) Pop()
Queue	Enqueu(x) Dequeu()
Linked-List	Search(x) Prev() Next()
Disjoint-Set	Make-Set(x) Union(x, y) Find-Set(x)

- Disjoint-set data structure is also known as:
  - Union-find data structure or
  - Merge-find set

An application of disjoint-set data structure

- Let's write a "list of well-defined instructions" to find the connected components.



(a)

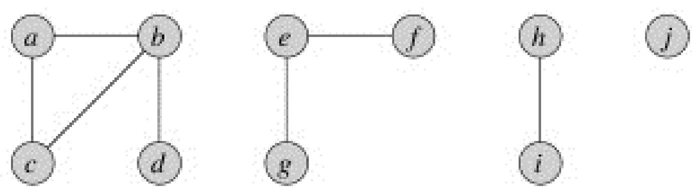
Edge processed	Collection of disjoint sets									
initial sets	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$

```
CONNECTED-COMPONENTS(G)
for each vertex v ∈ G.V
    MAKE-SET(v)
for each edge (u, v) ∈ G.E
    if FIND-SET(u) ≠ FIND-SET(v)
        UNION(u, v)
```

- CONNECTED-COMPONENTS initially places each vertex v in its own set.

An application of disjoint-set data structure

- Let's write a "list of well-defined instructions" to find the connected components.



(a)

Edge processed	Collection of disjoint sets									
initial sets	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b,d)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e,g)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(a,c)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(h,i)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(a,b)	$\{a,b,c,d\}$				$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(e,f)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$
(b,c)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$

(b)

CONNECTED-COMPONENTS(G)

```
for each vertex v ∈ G.V
    MAKE-SET(v)
for each edge (u, v) ∈ G.E
    if FIND-SET(u) ≠ FIND-SET(v)
        UNION(u, v)
```

- Then, for each edge (u,v), it unites the sets containing u and v.
- Also, we can determine whether two vertices are in the same component or not using FIND-SET operation.

SAME-COMPONENT(u, v)

```
if FIND-SET(u) == FIND-SET(v)
    return TRUE
else return FALSE
```

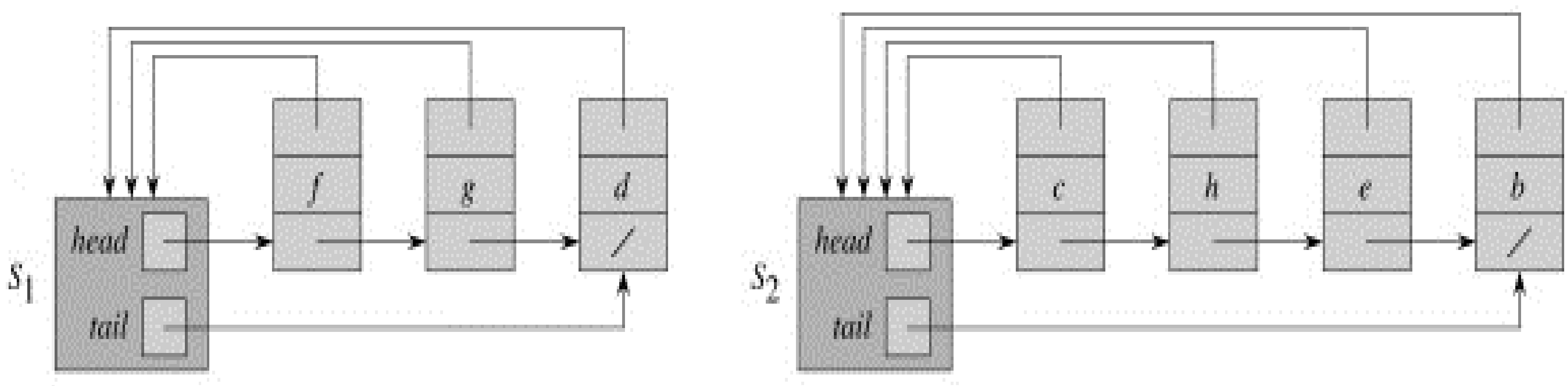
Implementation of Disjoint-Sets

- We have see the Abstract Data Type of disjoint-sets.

- We can implement a disjoint-set data structure in many ways, but here are two approaches:
  - Using link-lists
  - Using rooted-trees

Link-list representation of disjoint sets

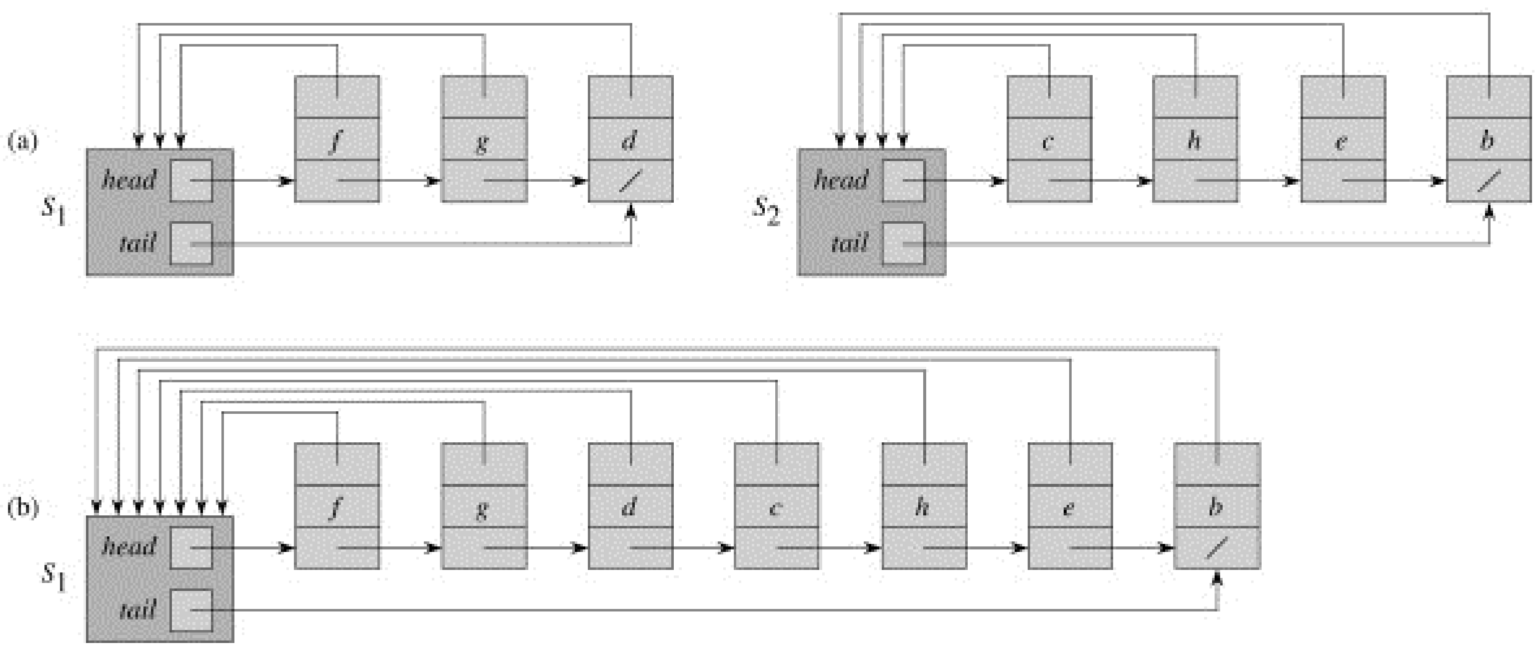
- A simple way to implement a disjoint-set.
- Each set is represented by its own linked-list. S = {S_1, S_2}



- Disjoint-Set has following 2 attributes:
  - *head*: pointing to the first object in the list
  - *tail*: pointing to the last object in the list
- Each object(node) has following 3 attributes:
  - *parent*: a pointer back to the set object.
  - *member*: data part of the node.
  - *next*: a pointer to the next object

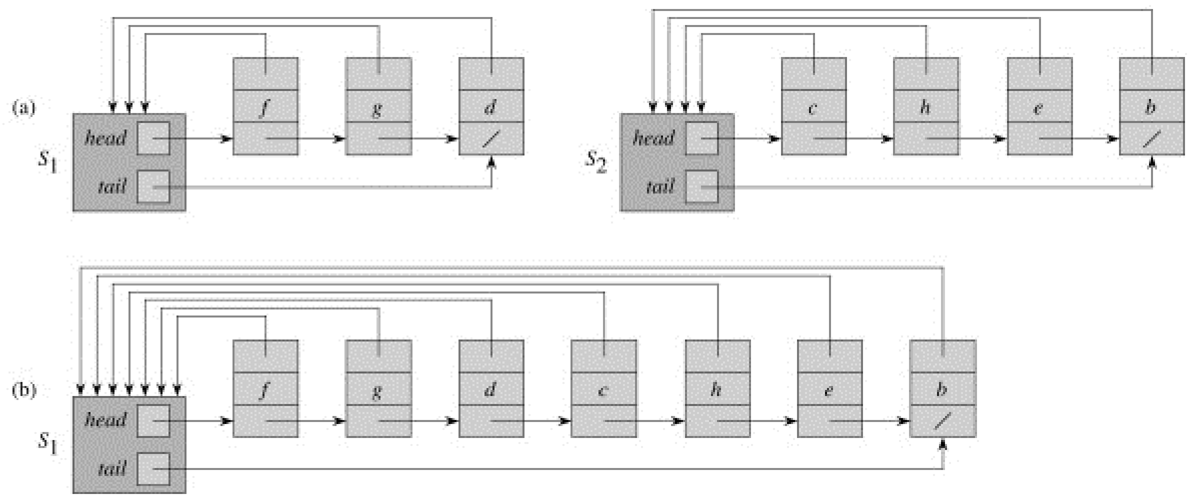
Link-list representation of disjoint sets

- Analysis of disjoint-set operations with this link-list representation
  - MAKE-SET(x): We create a new linked-list whose only object is x. So it is easy and requires O(1) time.
  - FIND-SET(x): Follow the pointer from x back to its set object(parent). It requires a constant O(1) time too.
  - UNION(x, y): Append y's list at the end of the x's list. It looks like O(1) too but it is not, because we need to update the parent(pointer to the set) of each object in y.



Implementation of Union

- UNION(x, y): Append y's list at the end of the x's list. It looks like O(1) too but it is not, because we need to update the parent(pointer to the set) of each object in y.



- Assume that we have objects $x_1, x_2, \ldots, x_n$. And, after MAKE-SET operation, we have n separate linked-lists. What is the worst-case running-time to get the union of all sets?

{1} {2} .... {n}

If we append the larger list to the smaller list.

Operation	# objects updated
UNION(x_2, x_1)	1
UNION(x_3, x_2)	2
UNION(x_4, x_3)	3
UNION(x_5, x_4)	4
$\vdots$	$\vdots$
UNION(x_n, x_{n-1})	$n-1$
	$\Theta(n^2)$ total

$1+2+3+\ldots\ n = n(n+1)/2 = (n^2+n)/2 = O(n^2)$

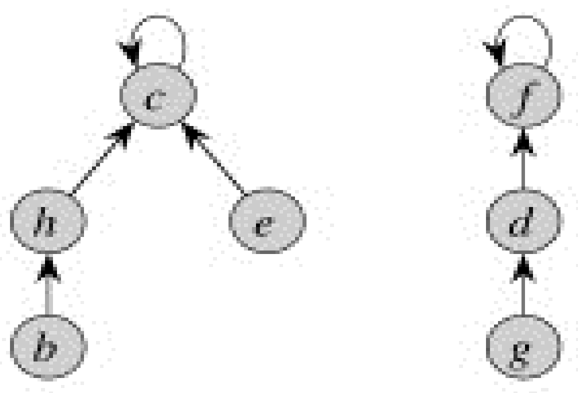
If we append the smaller list to the larger list.

times updated	size of resulting set
1	≥ 2
2	≥ 4
3	≥ 8
$\vdots$	$\vdots$
k	$\geq 2^k$
$\vdots$	$\vdots$
$\lg n$	$\geq n$

O(n logn)

Disjoint-set forests

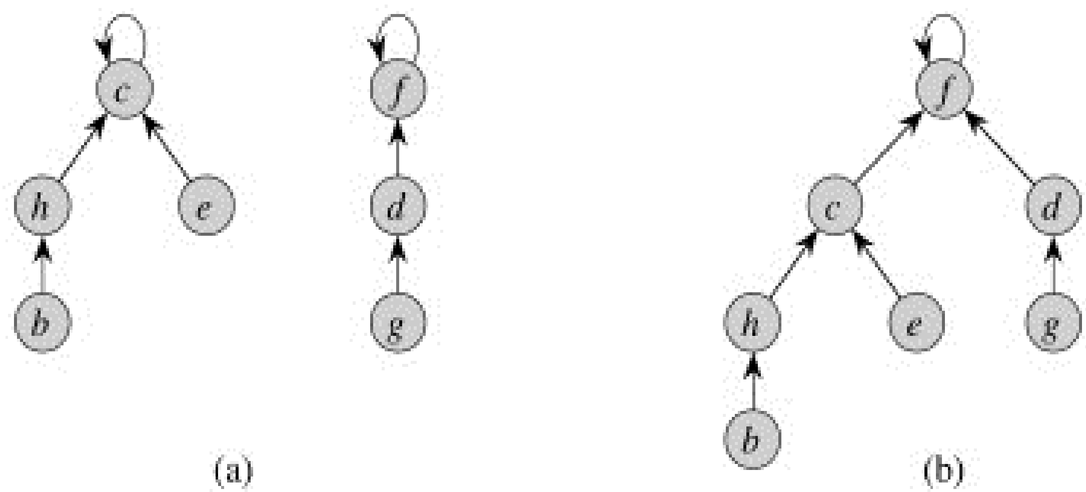
- Alternatively, we can represent sets by rooted trees, with
  - Each node containing one member
  - Each tree representing one set



- In a disjoint-set forest:
  - Each member points only to its parent.
  - The root of tree contains the representative and is its own parent.

Disjoint-set forests

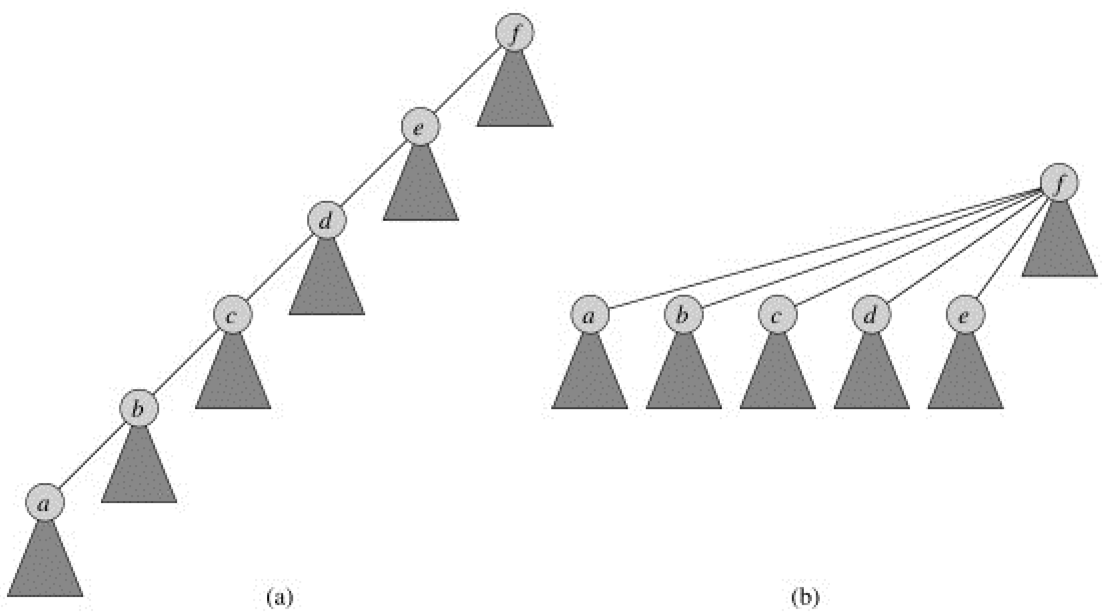
- We can perform the 3 disjoint-operations:
  - MAKE-SET(x): creates a tree with just one node. And its running-time is constant.
  - FIND-SET(x): follows parent pointers until we find the root of the tree.
  - UNION(x, y): makes the root of one tree to point to the root of the other.



Heuristics to improve running time

- *Union by rank*: This heuristic improves the running time of UNION operation.
  - While making union, choose the root of fewer nodes point to the root of the tree with more nodes. (It is similar to the linked-list heuristic)

- *Path compression*: This heuristic improves the running time of FIND-SET operation.
  - Make each node on the find path point directly to the root.



a-) Triangles represent subtrees, whose roots are the nodes shown. It is a nested-tree.

b-) After path-compression, each node on the path points directly to the point.

Heuristics: experience-based techniques that are used to speed-up some process.

Future Usage

- Disjoint-sets is simple, but useful.
- Next-week, we will use disjoint-sets and its operations for the minimum-spanning tree algorithm.
  - MAKE-SET(x)
  - UNION(x, y)
  - FIND-SET(x)