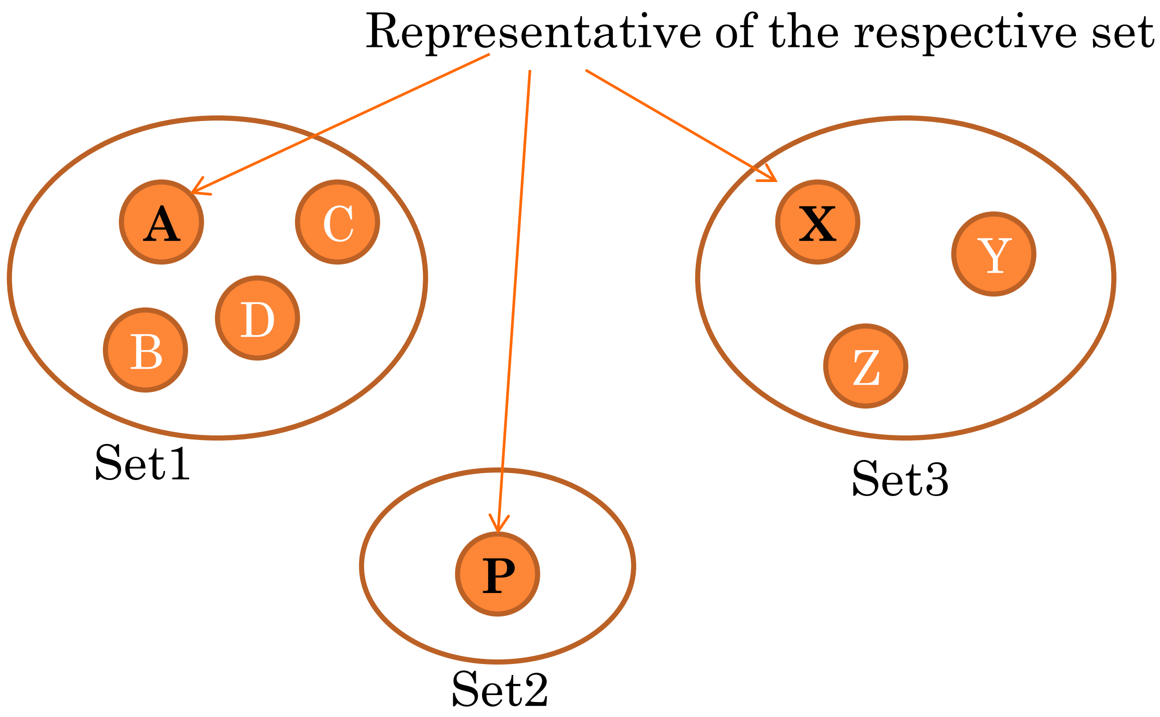
Disjoint Set

Tanjina Helaly

DISJOINT SET

- A data structure to keep track of set of elements partitioned into different disjoint (non overlapping) subset
- Each set is represent by a representative
  - In some applications, it doesn't matter which member is used as the representative
  - Other applications may require a pre-specified rule for choosing the representative, such as choosing the smallest member in the set (assuming, of course, that the elements can be ordered)

Disjoint set



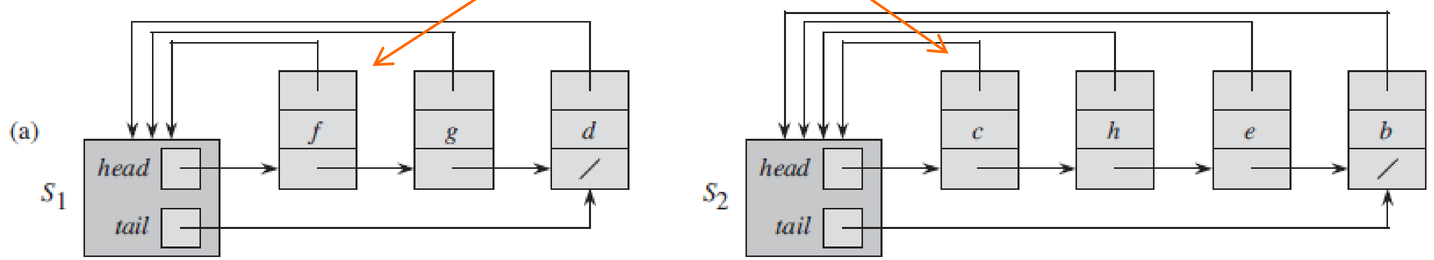
How to represent

- Can be represented by **Linked list**.
- But usually implemented as **tree like structure**.

How to represent

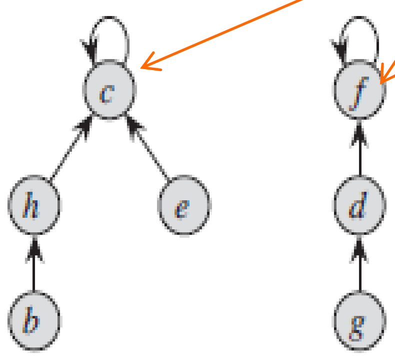
- Linked Link

Representative (The element head points to)



- Tree

Representative
- the node that is its own parent
- root node



3 OPERATIONS

- Make-Set(X)
  - Create a set with just one node x
    - condition – x can't be in any other set (as the sets are disjoint)
- Find-Set(X)
  - Find the representative of the set X belong to.
- Union(X,Y)
  - Combine the 2 sets x and y belongs to.

Create-Set

- Create a set with just one node x
  - condition – x can't be in any other set (as the sets are disjoint)
  - Complexity: O(1)

function *MakeSet*(x)

if x is not already present:

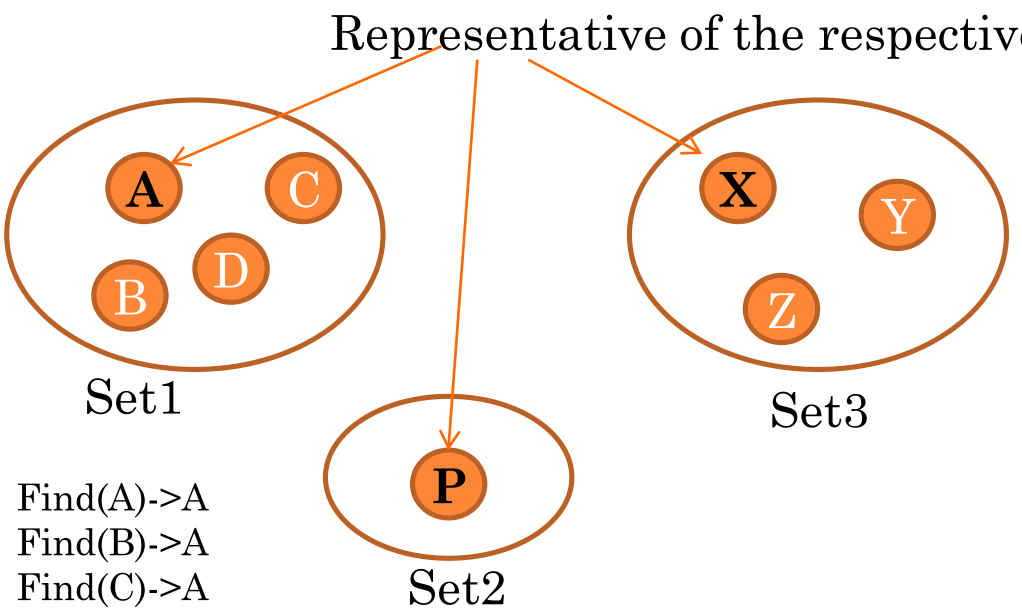
add x to the disjoint-set tree

x.parent := x

x.rank := 0

Find-Set

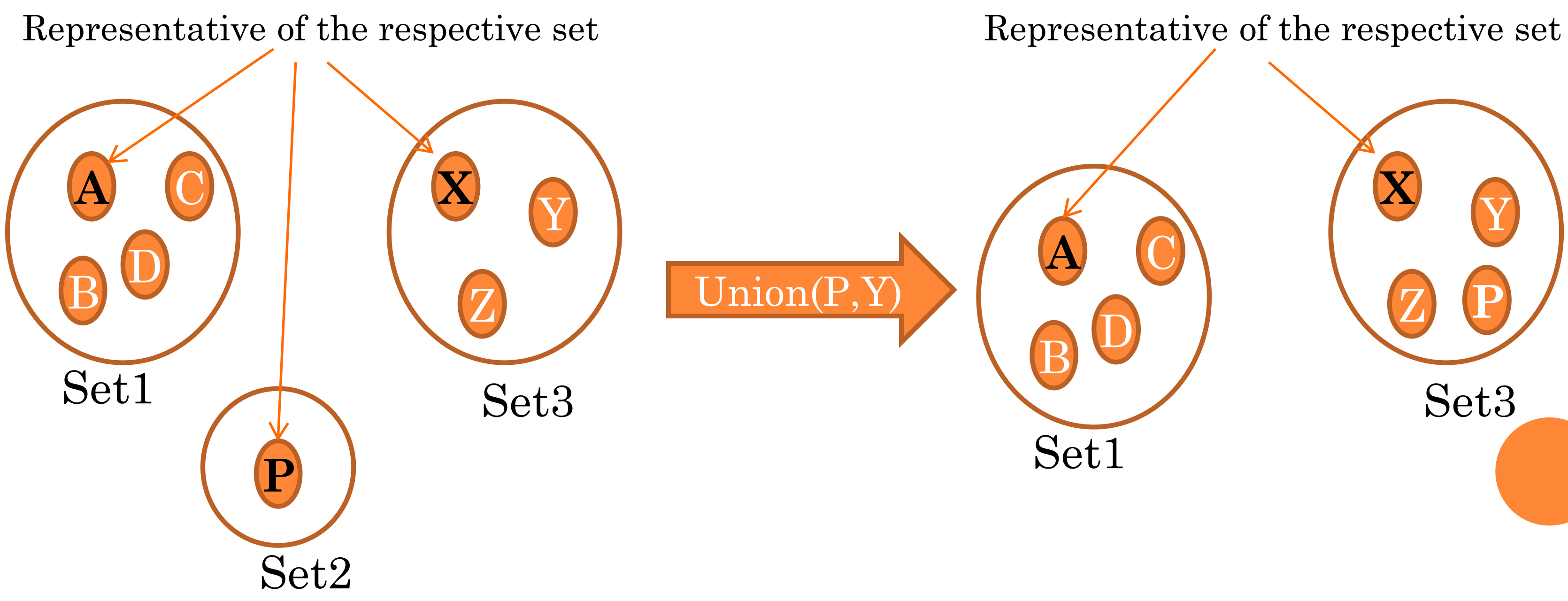
```
int find(int x)
{
    if (x.parent == x)
        return x;
    return find(x.parent);
}
```



Find(A)->A
Find(B)->A
Find(C)->A
Find(D)->A

UNION

- Combines the dynamic sets that contain x and y into a new set that is the union of these two sets.
- We assume that the two sets are disjoint prior to the operation.
- Choose representative from any set.
- Reduce the number of set by 1.



Union

```
Union(x,y)
   x-set = Find-Set(x)
   y-set = Find-Set(y)
   if(x-set !=y-set)
        x-set.Parent = y-set;
```

UNION USING TREE STRUCTURE

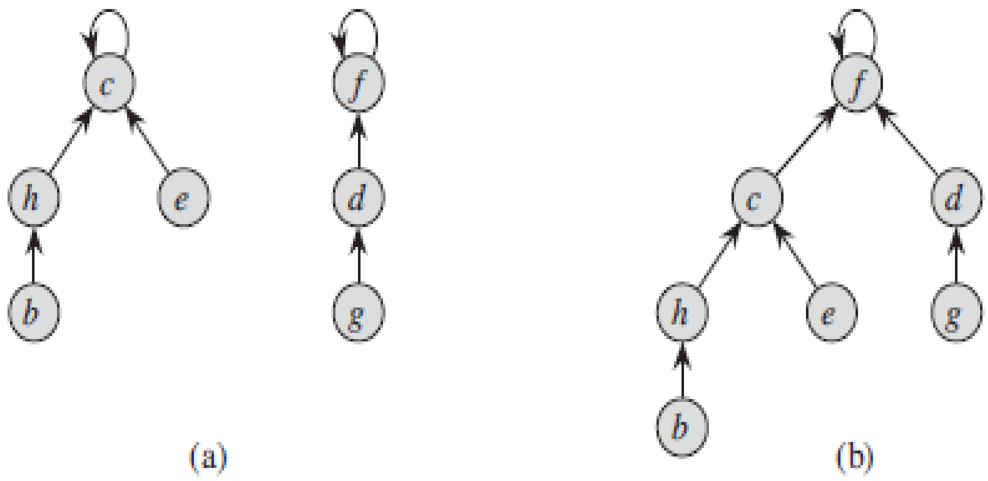


Figure 21.4 A disjoint-set forest. **(a)** Two trees representing the two sets of Figure 21.2. The tree on the left represents the set $\{b, c, e, h\}$, with c as the representative, and the tree on the right represents the set $\{d, f, g\}$, with f as the representative. **(b)** The result of UNION(e, g).

Union using Linked List

21.2 Linked-list representation of disjoint sets 565

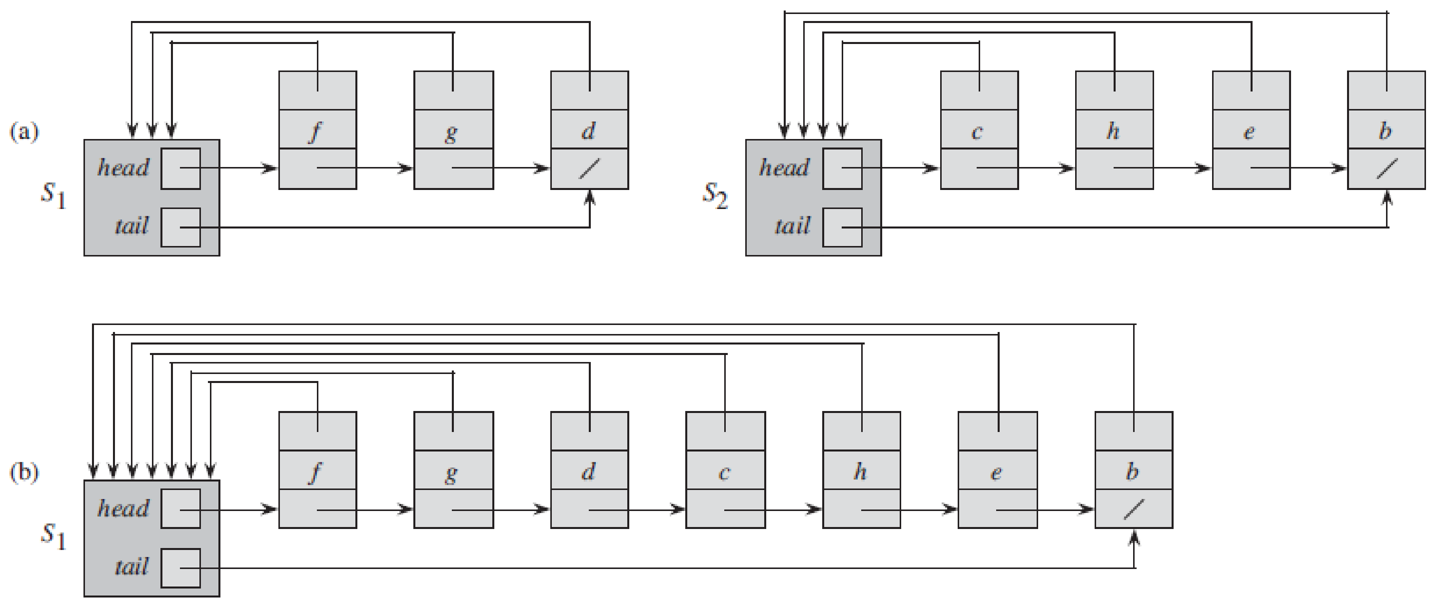


Figure 21.2 **(a)** Linked-list representations of two sets. Set S_1 contains members d, f, and g, with representative f, and set S_2 contains members b, c, e, and h, with representative c. Each object in the list contains a set member, a pointer to the next object in the list, and a pointer back to the set object. Each set object has pointers *head* and *tail* to the first and last objects, respectively. **(b)** The result of UNION(g, e), which appends the linked list containing e to the linked list containing g. The representative of the resulting set is f. The set object for e's list, S_2, is destroyed.

Union Problem

- For linked list
  - Need at most n-1 union operation
  - Each union will cause all members of the newly appended list to be updated with the new representative and also need to update the tail pointer.
    - $O(n^2)$
- For tree structure
  - Tree could get unbalance if we keep adding the tree with more depth as a leaf of smaller depth tree. Worst case would be creating a linear tree with one child at every level.
  - Operation for unbalanced tree is really slow.

How to solve

- Use heuristics
  - Union-by-rank
    - Add the lower rank set at the end of higher rank set so that minimum updates are required.
      - For linked list, rank is the length
      - For tree, rank is the depth of the tree.
  - Path compression
    - Flattening the structure of the tree.

Path Compression

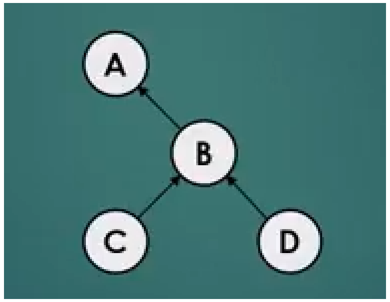
- Set every visited node to be connected to the root node directly
- This is modified version of the Find method

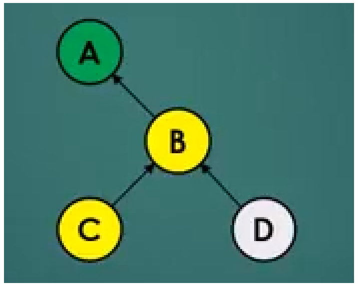
```
Find(x):
  If x.parent != x // x is not root node
          x.parent = Find(x.parent);
  return x.parent;
```

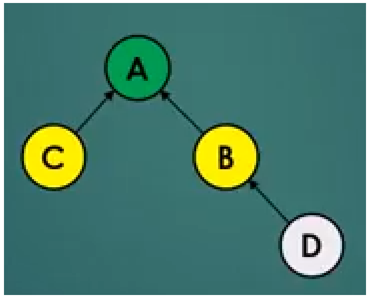
- Once the tree is flattened, all nodes will be directly connected to root/representative. And Find(any node) will take constant time complexity.

Path Compression - example

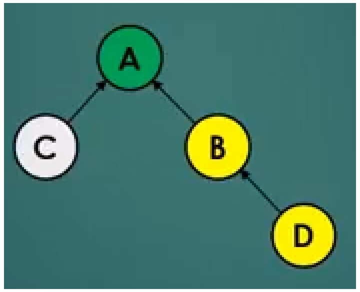
Find(C) – with path compression

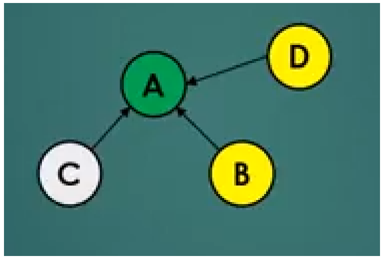






Find(D) – with path compression



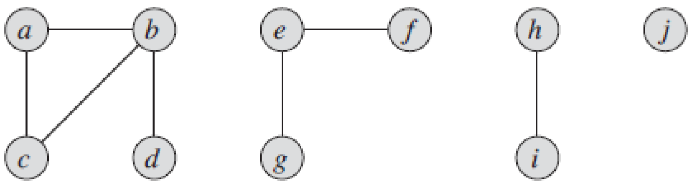


APPLICATIONS

Applications

- Main application is creating connected component.
  - Involve manipulating objects from wide range e.g.
    - Pixels in digital image
    - Computers in network
    - Friends in social network
    - Transistors in computer chip
    - .....
- Reachability checking
- Cycle checking
- Connected component counting
- Connected component size calculation,
- Component labeling for vertices,
- Minimum Spanning Tree

CONNECTED COMPONENT



(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)

Figure 21.1 (a) A graph with four connected components: $\{a, b, c, d\}$, $\{e, f, g\}$, $\{h, i\}$, and $\{j\}$. (b) The collection of disjoint sets after processing each edge.

```
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)
```

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

Cycle detection

- During the union process if 2 vertices belong to the same disjoint set -> there is a cycle

Union(X,Y)	Union(X,Y) – for cycle checking

x-root = Find-Set(x)
y-root = Find-Set(y)
if(x-root !=y-root)
    x-root.Parent = y-root;
	x-root = Find-Set(x)
	y-root = Find-Set(y)
	if(x-root==y-root)
	    there is a cycle
	else
	    x-root.Parent = y-root;

Reachability/Connectivity checking

- If 2 vertices belong to the same set then they are connected to each other.
- Pre-processing
  - Need to create the connected component before calling this method.

IsConnected(X,Y)

x-root = Find-Set(x)

y-root = Find-Set(y)

if(x-root==y-root)

y is reachable from x or vice versa.

Reference

- Chapter 21 (Cormen) -> 21.1, 21.3