Linked-list & Tree of Disjoint Sets

Kuan-Yu Chen (陳冠宇)

2019/05/08 @ TR-310-1, NTUST

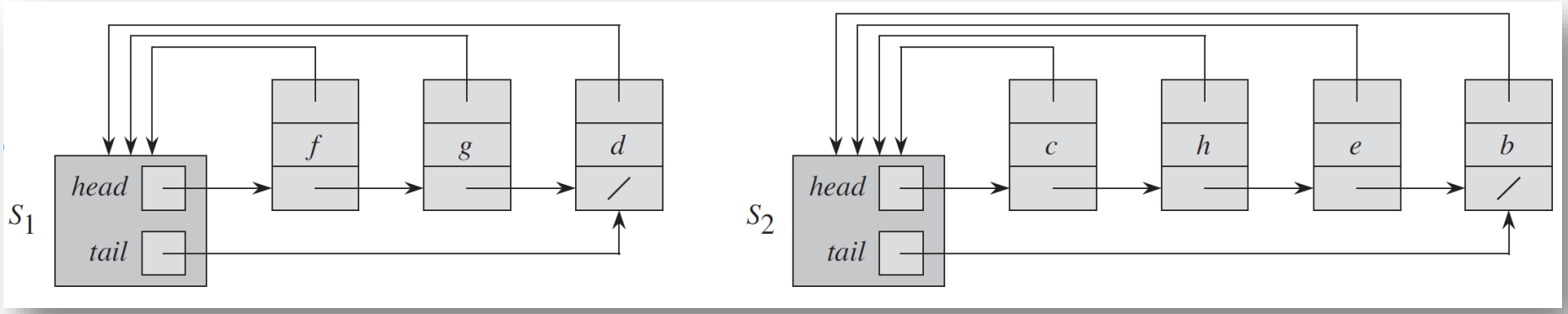
Review

- Some applications involve grouping n distinct elements into a collection of disjoint sets
  - $\{1, 2, 3\}$
  - $\{4, 5\}$
  - $\{6\}$
  - $\{1, 3, 5\}$
  - $\{1, 2, 3\}, \{4, 5\}, \{6\}$ are disjoint sets
  - $\{1, 2, 3\}, \{4, 5\}, \{6\}, \{1, 3, 5\}$ are not disjoint sets

- Letting x denote an object, an element in a set, we wish to support the following operations
  - MAKE-SET(x)
  - UNION(x, y)
  - FIND(x)

Linked-list of Disjoint Sets.

- A simple way to implement a disjoint-set data structure is using linked list
  - Each set is represented by its own linked list

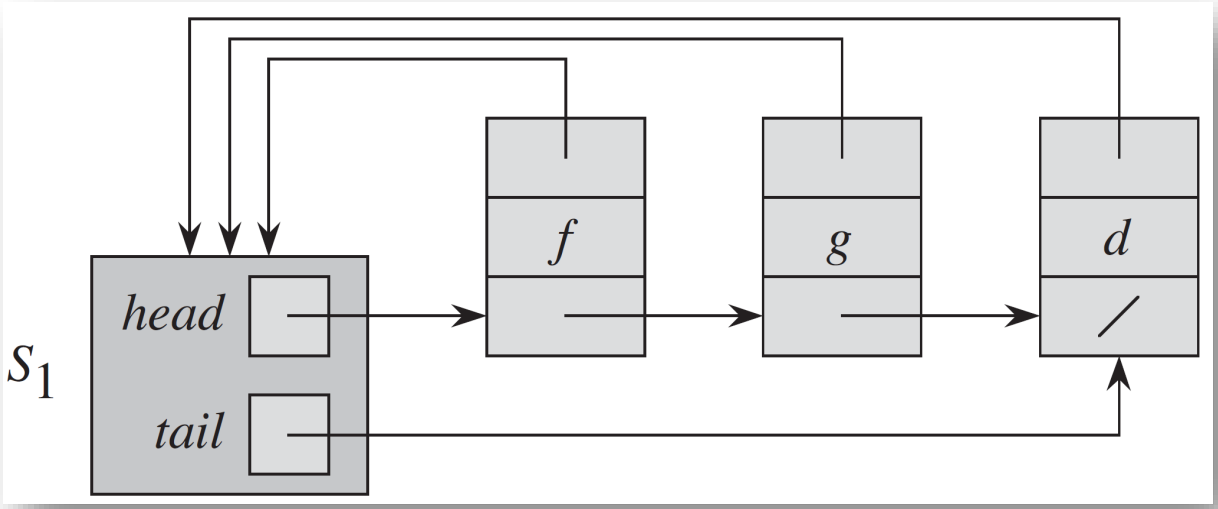


  - The object for each set has attributes ***head***, pointing to the first object in the list, and ***tail***, pointing to the last object
    - Each object in the list contains a set member, a pointer to the next object in the list, and a pointer back to the set object
    - The representative is the set member in the first object in the list

Linked-list of Disjoint Sets..

- With this linked-list representation, both MAKE-SET and FIND-SET are easy, requiring $O(1)$ time
  - For MAKE-SET(x), we create a new linked list whose only object is x
  - For FIND-SET(x), we just follow the pointer from x back to its set object and then return the member in the object that ***head*** points to

    FIND-SET(g) would return f

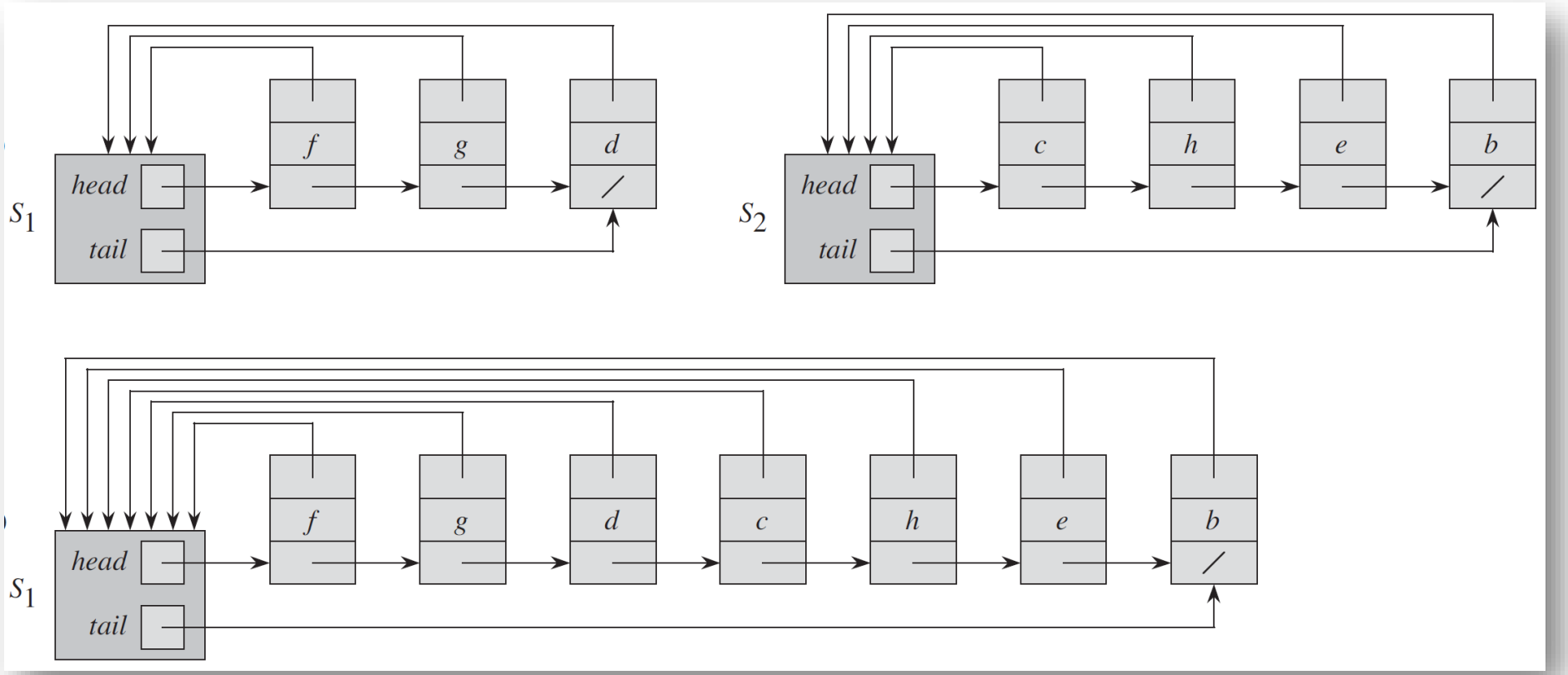


Linked-list of Disjoint Sets…

- For UNION(x, y), we append list of y onto the end of list of x

  We must update the pointer to the set object for each object originally on y's list

  For example, the operation UNION(g, e), causes pointers to be updated in the objects for b, c, e, and h



Linked-list of Disjoint Sets….

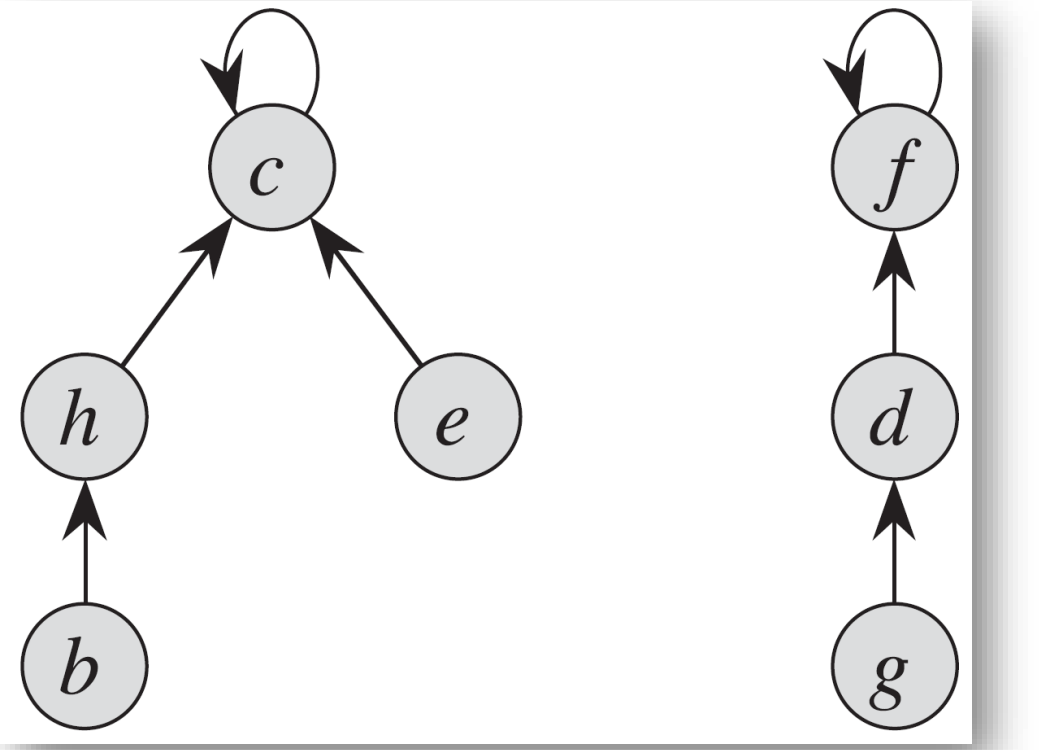
- Suppose that we have objects $x_1, x_2, \cdots, x_n$
  - If we execute the sequence of n MAKE-SET operations followed by $n-1$ UNION operations incrementally
  - We should take $\Theta(n^2)$
    - Because we append a longer list onto a shorter list

Operation	Number of objects updated
MAKE-SET(x_1)	1
MAKE-SET(x_2)	1
⋮	⋮
MAKE-SET(x_n)	1
UNION(x_2, x_1)	1
UNION(x_3, x_2)	2
UNION(x_4, x_3)	3
⋮	⋮
UNION(x_n, x_{n-1})	$n-1$

  - ***Weighted-union heuristic:*** we always append the shorter list onto the longer list!

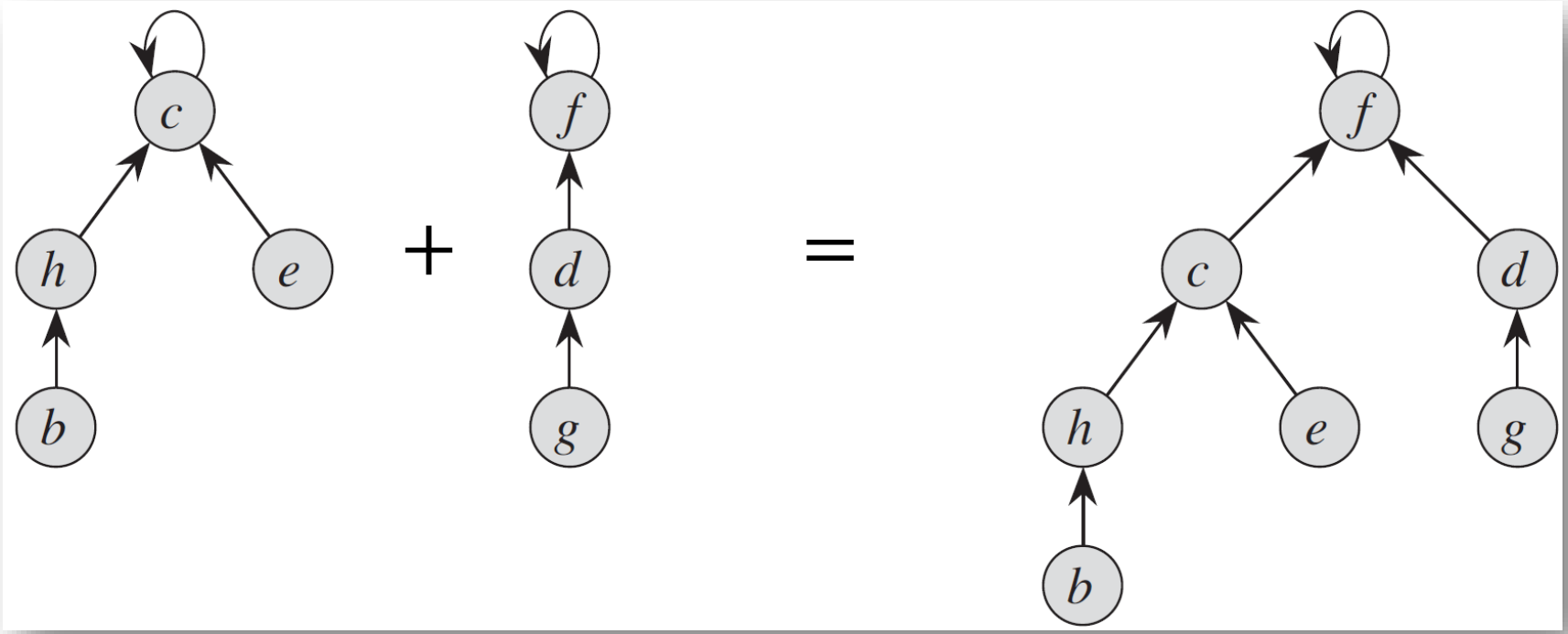
Disjoint-set Forests.

- In a faster implementation of disjoint sets, we represent sets by rooted trees
  - Each node containing one member and each tree representing one set
  - The root of each tree is the representative of the set



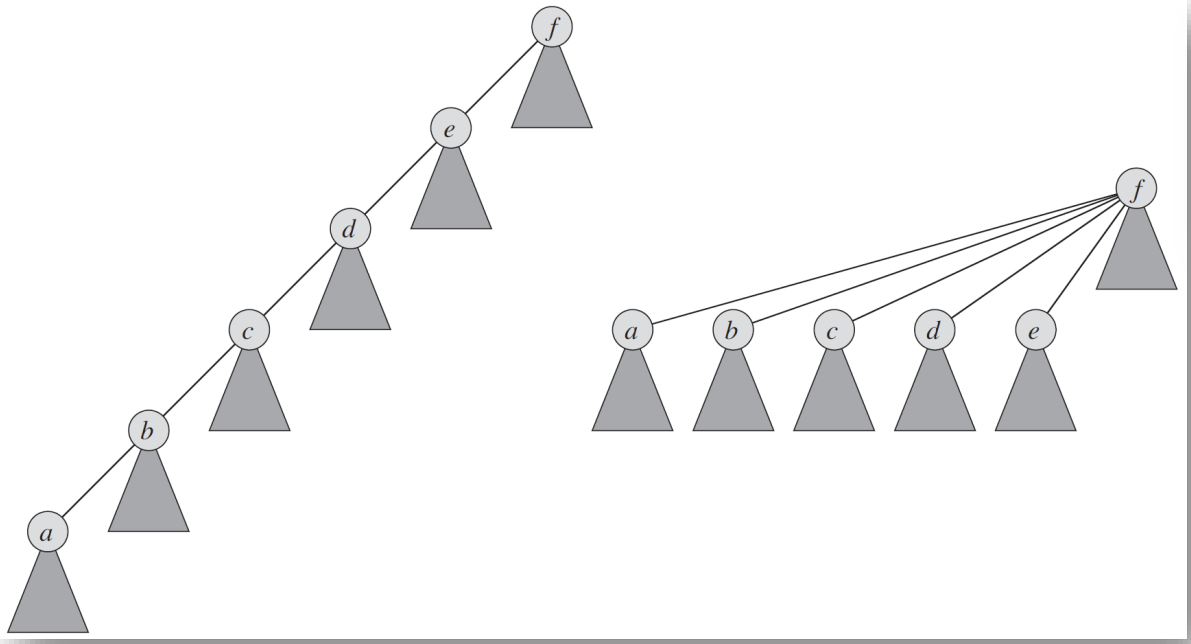
Disjoint-set Forests..

- MAKE-SET: simply creates a tree with just one node
- FIND-SET: performs a FIND-SET operation by following parent pointers until we find the root of the tree
- UNION: causes the root of one tree to point to the root of the other



Disjoint-set Forests…

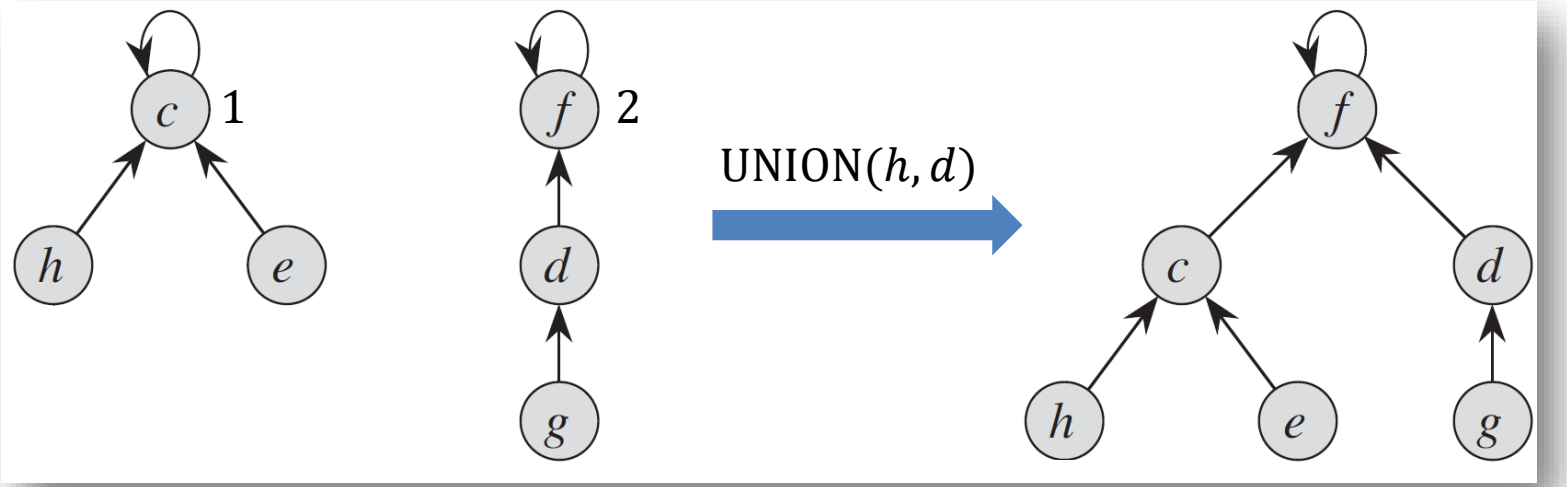
- The ***path compression*** is quite simple and highly effective
  - We use it during FIND-SET operations to make each node on the find path point directly to the root
    - If we perform FIND-SET(a) on the left tree, the path compression will also perform
    - Thus, after executing FIND-SET(a), each node on the find path now points directly to the root




```
FIND-SET(x)
1  if x ≠ x.p
2      x.p = FIND-SET(x.p)
3  return x.p
```

Disjoint-set Forests….

- The **union-by-rank** strategy
  - Each node keeps track of its rank
    - An upper bound on the height of the node
  - To perform UNION, we link root with smaller rank to root with larger rank
    - When UNION is performed, only the rank of the roots may change



Disjoint-set Forests…..

- The union-by-rank strategy algorithm

```
MAKE-SET(x)
1   x.p = x
2   x.rank = 0

UNION(x, y)
1   LINK(FIND-SET(x), FIND-SET(y))

LINK(x, y)
1   if x.rank > y.rank
2         y.p = x
3   else x.p = y
4         if x.rank == y.rank
5               y.rank = y.rank + 1
```

Questions?

kychen@mail.ntust.edu.tw