算法基础
Foundation of Algorithms

主讲人 徐云

Fall 2018, USTC

Chapter 21 Data Structures for Disjoint Sets

21.1 Overview and Ops

- Disjoint-set Data Structures
- Operations on Disjoint-set
- Application

Disjoint-set Data Structures

- Maintain collection $S=\{S_1, S_2, \ldots, S_k\}$ of *disjoint sets* with dynamic (changing over time).
  - where any S_i and S_j are no any common members.
- Each set is identified by a *representative(rep. later)*.
  - which is some member of the set.
- Remark:
  - Doesn't matter which member is the rep, we get the same answer as long as if we ask for the rep.

Operations on Disjoint-set

- Make-Set(x): make a new set $S_i = \{x\}$.
- Union(x, y): if $x \in S_x$, $y \in S_y$, then

  $$S = S - S_x - S_y \cup \{S_x \cup S_y\}$$

  - Rep. of new set is any member of $S_x \cup S_y$
  - Destroys S_x and S_y.
- Find-Set(x): return rep. of set containing x.

- Analysis in terms of:
  - *n* = # of elements = # of *Make-Set operations*.
  - *m* = total # of operations.

Application: Dynamic connected components

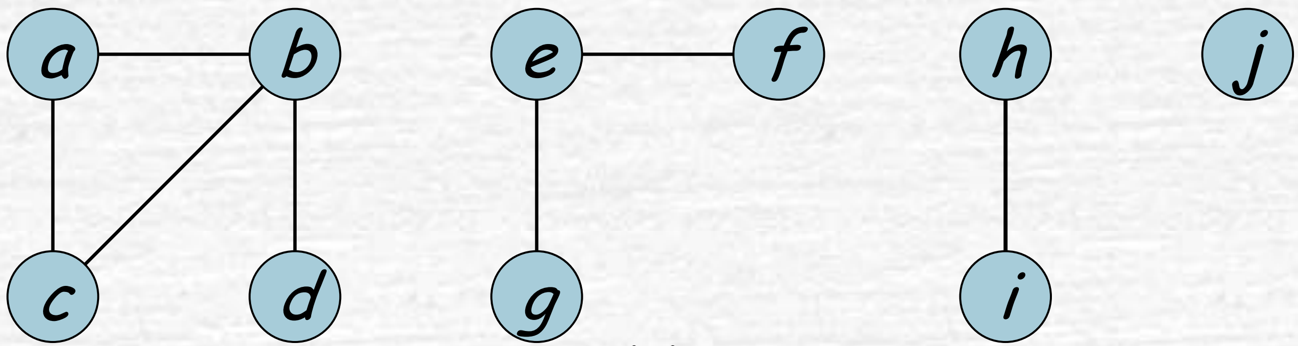
- Definition: For a graph G=(V, E), vertices u,v are *in same connected component* if and only if there's a path between them.
- Goal: Connected components partition vertices into equivalence classes.

```
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
```

- Remark: actually implementing,
  - each vertex needs a handle (指针) to its rep.,
  - Each rep. needs a handle to its vertex.

Application: an Instance



(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)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$

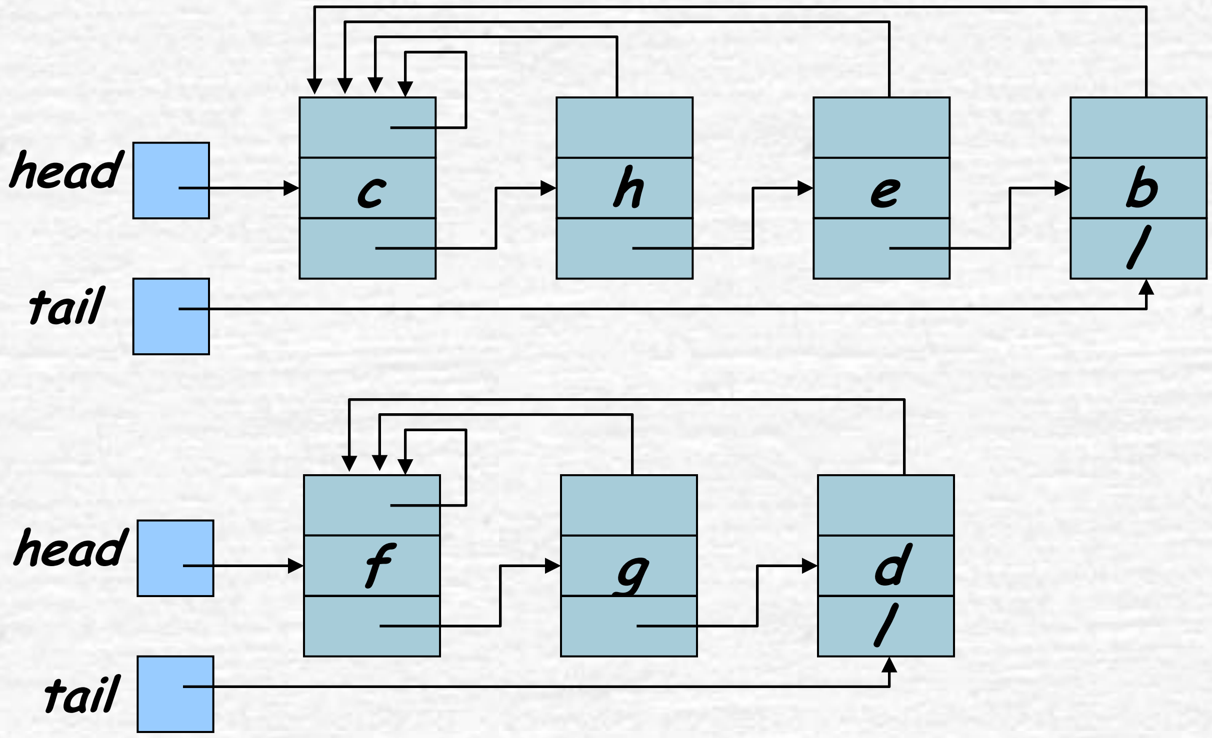
(b)

Chapter 21 Data Structures for Disjoint Sets

21.2 Linked List Representation

- Data Structure Design
- Simple Implementation of Union
- Weighted-Union Heuristic
- Theorem and its Proof

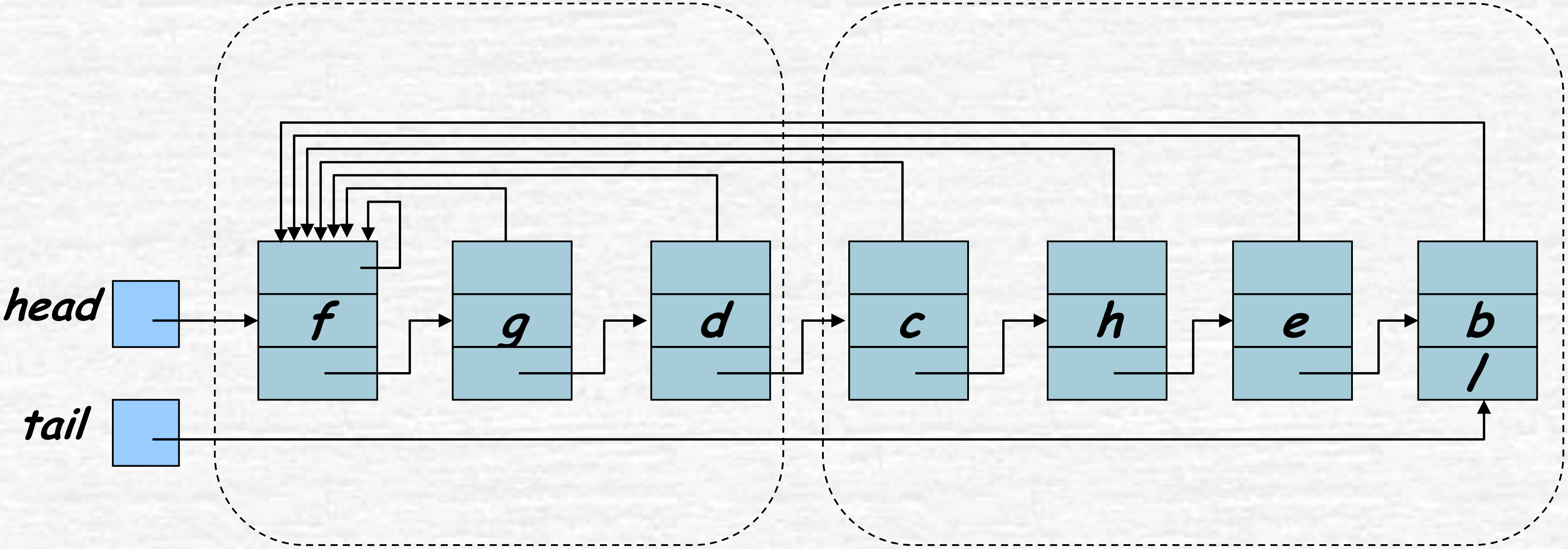
Data Structure Design



- Take the first element as the rep. in a list.
- Make-Set, Find-Set only need O(1).

Simple Implementation of Union (1)

- Union(x, y): *append y's list onto end of x's list*. Use x's tail pointer to find the end.
  - Need to update the pointer to the set rep. for every node on y's list.



Simple Implementation of Union (2)

- If appending a large list onto a small list, it can take a while.

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

- Amortized time per operation θ(n).

Weighted-Union Heuristic

- Always append the smaller list to the larger list.
- For any rep. stores the length (i.e. weight) of its list.

- Theorem

  With weighted union, a sequence of m operations on n elements takes O(m+nlogn) time.
  - m is total # of operations of Make-Set, Union, and Find-Set.

Proof of Theorem

- Each Make-Set() and Find-Set() still takes O(1).
- Lets consider the cost of Union():
  - Union cost is mainly the # of pointer updated for any x in smaller set.
  - The times updated of any x have

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

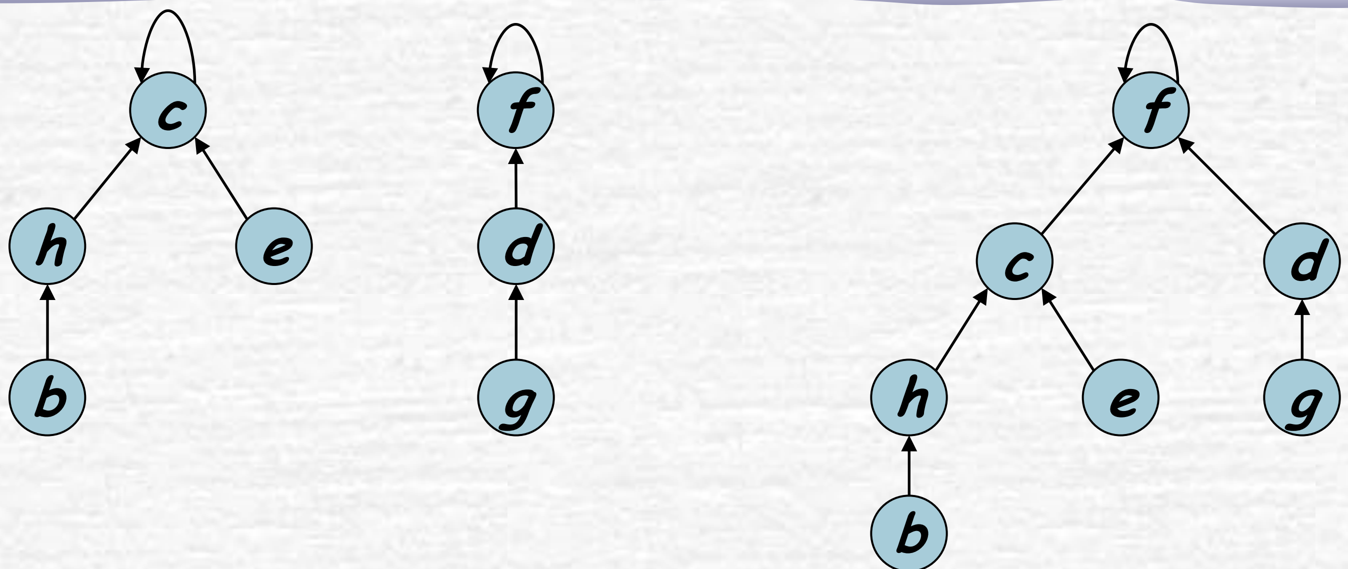
  - So, the total time spent updating object pointers O(nlogn).
- Because there are O(m) for all ops, therefore, The total time for the entire sequence is O(m+nlogn)

Chapter 21 Data Structures for Disjoint Sets

21.3 Disjoint-set Forest

- Forest Trees
- Some Heuristic Tricks
- Implementation

Forest Trees



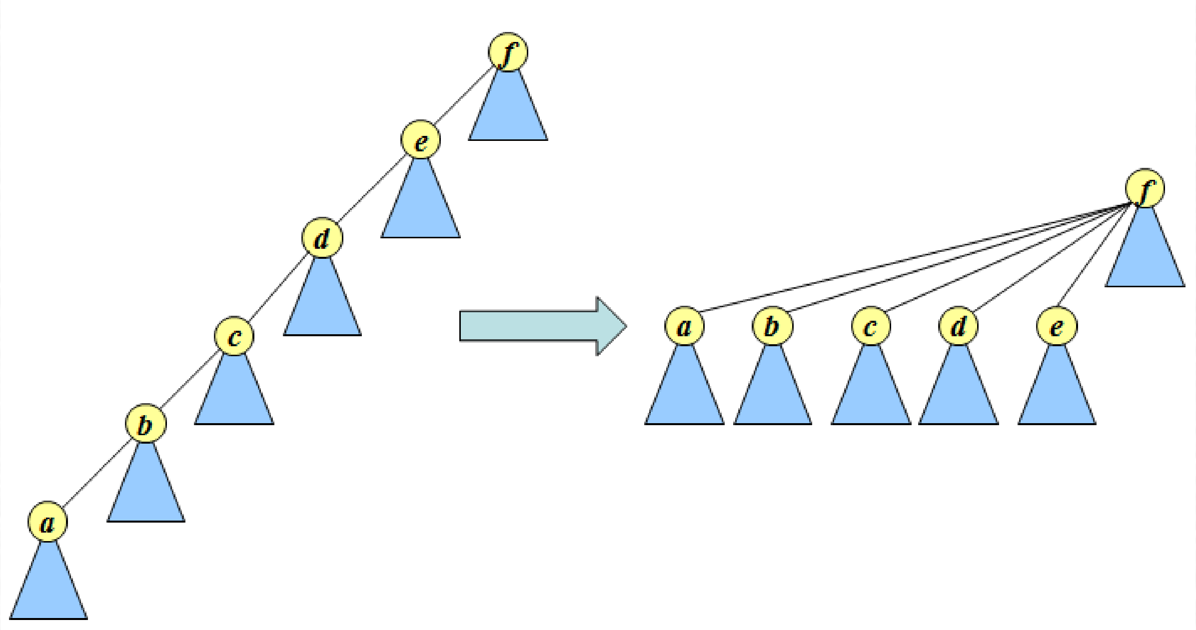
- 1 tree per set. And root is representative.
- Each node points only to its parent.
- We known that
  - Make-Set(x): O(1).
  - Find-Set(x): O(h), where h is the height of tree including x.
  - Union(x, y): the root of the tree including y is pointed to that of x.

Heuristics 1: Union by Rank

- Background: no any good heuristic, it could get a linear chain of nodes.
- Idea: Make the root of the smaller tree (fewer nodes or lower height) into a child of the root of the larger tree.
- Remak:
  - Don't actually use size.
  - Use rank, which is an upper bound on height of node.
  - Make the root with the smaller rank as a child of the root with the larger rank.

Heuristics 2: Path Compression

- Idea:
  - Find path = nodes visited during Find-Set on the trip to the root.
  - Make all nodes on the find path direct children of root.



each node has two attributes, p (parent) and rank

Implementation

MAKE-SET(x)

$x.p = x$

$x.rank = 0$

UNION(x, y)

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

FIND-SET(x)

if $x \neq x.p$

$x.p =$ FIND-SET($x.p$)

return $x.p$

a pass up to find the root, and a pass down as recursion, such as each node on find path to point directly to root.

LINK(x, y)

if $x.rank > y.rank$

$y.p = x$

else $x.p = y$

// If equal ranks, choose y as parent and increment its rank.

if $x.rank == y.rank$

$y.rank = y.rank + 1$

- Running time (proof in 21.4)
  - If use both union by rank and path compression, O(mα(n)).
  - This bound is tight, pls see right.
  - How about using one alone?

n	$\alpha(n)$
0–2	0
3	1
4–7	2
8–2047	3
2048–$A_4(1)$	4

End of Ch21