Disjoint Sets Data Structure (Chap. 21)

- A disjoint-set is a collection ©={S_1, S_2,…, S_k} of distinct dynamic sets.
- Each set is identified by a member of the set, called *representative*.
- Disjoint set operations:
  - MAKE-SET(*x*): create a new set with only *x*. assume *x* is not already in some other set.
  - UNION(*x*,*y*): combine the two sets containing *x* and *y* into one new set. A new representative is selected.
  - FIND-SET(*x*): return the representative of the set containing *x*.

Multiple Operations

- Suppose multiple operations:
  - n: #MAKE-SET operations (executed at beginning).
  - m: #MAKE-SET, UNION, FIND-SET operations.
  - $m \geq n$, #UNION operation is at most n-1.

An Application of Disjoint-Set

- Determine the connected components of an undirected graph.

```
CONNECTED-COMPONENTS(G)
1.  for each vertex v ∈ V[G]
2.      do MAKE-SET(v)
3.  for each edge (u,v) ∈ E[G]
4.      do if FIND-SET(u) ≠ FIND-SET(v)
5.          then UNION(u,v)
```

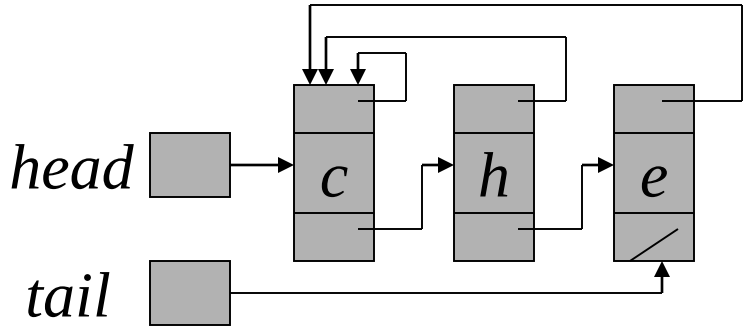
```
SAME-COMPONENT(u,v)
1.  if FIND-SET(u)=FIND-SET(v)
2.      then return TRUE
3.      else return FALSE
```

Linked-List Implementation

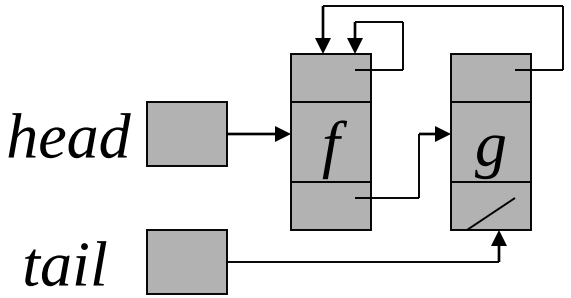
- Each set as a linked-list, with head and tail, and each node contains value, next node pointer and back-to-representative pointer.
- Example:
- MAKE-SET costs $O(1)$: just create a single element list.
- FIND-SET costs $O(1)$: just return back-to-representative pointer.

Linked-lists for two sets

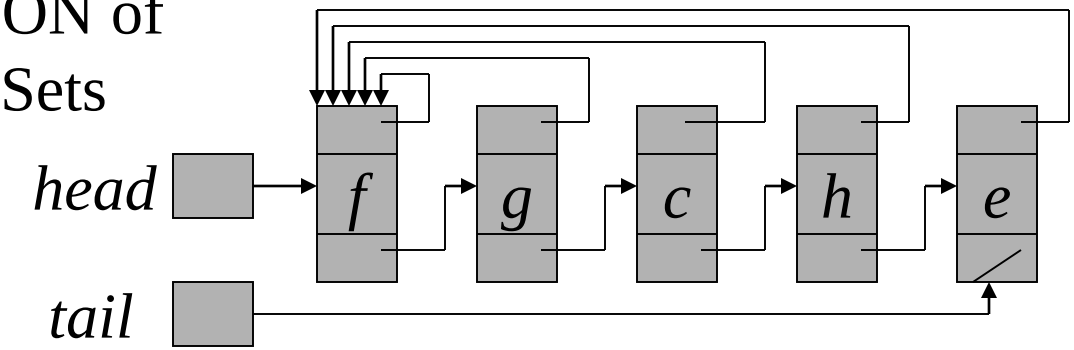
Set {*c*,*h*,*e*}



Set {*f*, *g*}



UNION of two Sets



UNION Implementation

- A simple implementation: UNION(x,y) just appends x to the end of y, updates all back-to-representative pointers in x to the head of y.
- Each UNION takes time linear in the x's length.
- Suppose n MAKE-SET(x_i) operations ($O(1)$ each) followed by n-1 UNION
  - UNION(x_1, x_2), $O(1)$,
  - UNION(x_2, x_3), $O(2)$,
  - …..
  - UNION(x_{n-1}, x_n), $O(n-1)$
- The UNIONs cost $1+2+\ldots+n-1=\Theta(n^2)$
- So $2n-1$ operations cost $\Theta(n^2)$, average $\Theta(n)$ each.
- Not good!! How to solve it ???

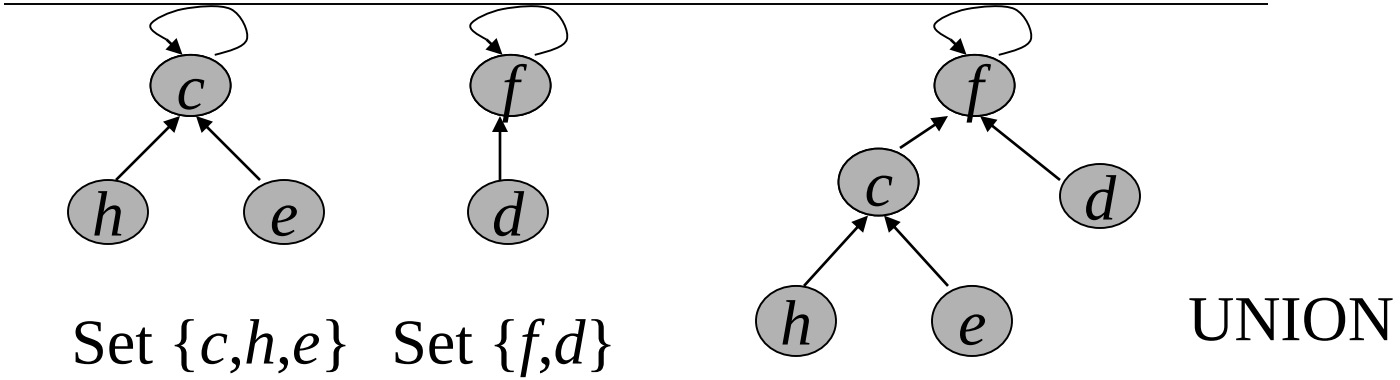
Weighted-Union Heuristic

- Instead appending x to y, appending the shorter list to the longer list.
- Associated a length with each list, which indicates how many elements in the list.
- Result: a sequence of m MAKE-SET, UNION, FIND-SET operations, n of which are MAKE-SET operations, the running time is $O(m+n\lg n)$. Why???

•Hints: Count the number of updates to back-to-representative pointer for any x in a set of n elements. Consider that each time, the UNION will at least double the length of united set, it will take at most lg n UNIONS to unite n elements. So each x's back-to-representative pointer can be updated at most lg n times.

Disjoint-set Implementation: Forests

- Rooted trees, each tree is a set, root is the representative. Each node points to its parent. Root points to itself.



Straightforward Solution

- Three operations
  - MAKE-SET(x): create a tree containing x. $O(1)$
  - FIND-SET(x): follow the chain of parent pointers until to the root. O(height of x's tree)
  - UNION(x,y): let the root of one tree point to the root of the other. $O(1)$
- It is possible that n-1 UNIONs results in a tree of height n-1. (just a linear chain of n nodes).
- So n FIND-SET operations will cost $O(n^2)$.

Union by Rank & Path Compression

- Union by Rank: Each node is associated with a rank, which is the upper bound on the height of the node (i.e., the height of subtree rooted at the node), then when UNION, let the root with smaller rank point to the root with larger rank.
- Path Compression: used in FIND-SET(x) operation, make each node in the path from x to the root directly point to the root. Thus reduce the tree height.

Algorithm for Disjoint-Set Forest

MAKE-SET(x)
1. $p[x] \leftarrow x$
2. $rank[x] \leftarrow 0$

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

LINK(x,y)
1. **if** $rank[x] > rank[y]$
2. **then** $p[y] \leftarrow x$
3. **else** $p[x] \leftarrow y$
4. **if** $rank[x] = rank[y]$
5. **then** $rank[y]$++

FIND-SET(x)
1. **if** $x \neq p[x]$
2. **then** $p[x] \leftarrow$FIND-SET($p[x]$)
3. **return** $p[x]$

Worst case running time for m MAKE-SET, UNION, FIND-SET operations is: $O(m\alpha(n))$ where $\alpha(n) \leq 4$. So nearly linear in m.

Summary

- Disjoint set
  - Three operations
  - Different implementations and different costs
- Forest implementation:
  - Union by rank and path compression
  - Properties: rank, level, iter.
  - Amortized analysis of the operations:
    - Potential function.

Minimum Spanning Tree -Prim's Algorithm,

- Idea
  - Begin from any node, each time add a new node which is closest to the existing subtree.
- Code -- MIN-SPT(G, w, r): key: key value and pi: parent
  - For each vertex u
    - key[u]= ∞
    - pi[u]=NIL
  - key[r]=0;
  - Q =all vertices of G (formed as a priority queue)
  - while (Q!=∅)
    - u=Extract-MIN(Q)
    - for each v ∈ adjacent[u]
      - If (v ∈Q and w(u,v)<key[v])
        - » pi[v]=u;
        - » key[v]=w(u,v) //update the priority queue

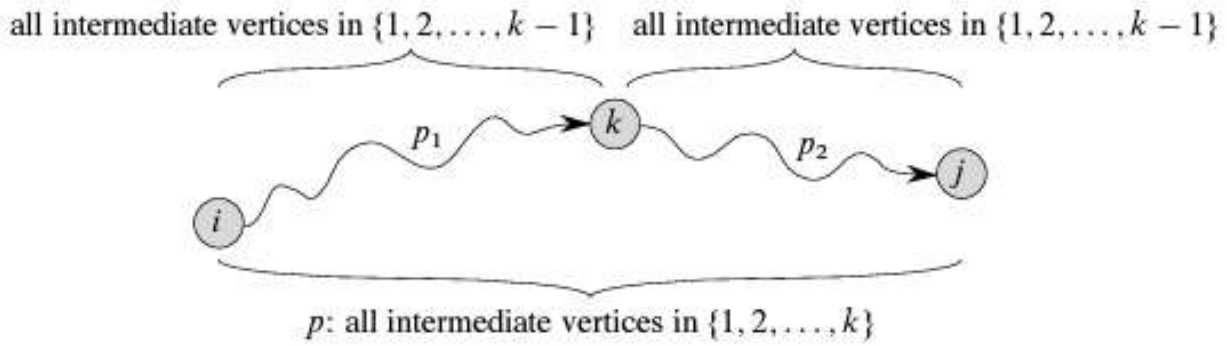
- Single source shortest path: Dijkstra's algorithm
  - w(u,v)<key[v] ➔ key[u]+w(u,v)<key[v]
  - key[v]=w(u,v) ➔ key[v]=key[u]+w(u,v)

Shortest paths among all pairs— Floyd-Warshall algorithm

- Given G=(V, E, w)
  - suppose V={1,2,…,n} are vertices and w is weight matrix on edges
- Consider a subset of vertices {1,2,…, k} for some k, for any pair of i and j, consider all the paths from i to j including only vertices from 1,2,…,to k and let p be one of the minimum such paths (also a simple path).

Floyd-Warshall algorithm (cont.)

- There are two cases:
  - K is not on path p. In this case, path p will be a minimum path including only vertices of 1,2,…, k-1.
  - k is on path p. In this case, the portion of i to k on path p will be a minimum path from i to k including only vertices 1,2, …, k-1, similarly for the pair of k and j.



Floyd-Warshall algorithm (cont.)

- Let D(i,j,k) denote the (length of) minimum path between i and j including only vertices 1,2,…, k.
- D(i,j,n) will be the answer for any pair i and j.
- Recursive definition:

$$D(i,j,k) = \begin{cases} w(i,j): & \text{if } k = 0, (i,j) \in E \\ \infty: & \text{if } k = 0, (i,j) \notin E \\ min\{D(i,j,k-1), D(i,k,k-1) + D(k,j,k-1)\}: & \text{if } k > 0 \end{cases}$$

- Algorithm:
  - Initialize for k=0
  - For (k=1 to n)
    - For (i=1 to n) and for (j=1 to n) D(i,j,k)=min{D(i,j,k-1), D(i,k,k-1)+D(k,j,k-1)}