Disjoint Sets Data Structure

Disjoint Sets

- A Disjoint set S is a collection of disjoint dynamic sets $S_1,...S_n$ where

  where

  $$\forall_{i \neq j} S_i \cap S_j = \phi$$

- Each set has a representative which is a member of the set (Usually the minimum if the elements are comparable)

Disjoint Set Operations

- Make-Set(x) – Creates a new set where x is it's only element (and therefore it is the representative of the set).

- Union(x,y) – Unites the dynamic sets that contains x and y, say S_x, S_y, into a new set that is the union of these two sets. The representative of the resulting set is any member of S_x U S_y .
- Find(x) – Returns the representative of the set containing x.

Analyzing Operations

- We analyze running times of disjoint-set data structures in terms of two parameters: n, the number of Make_Set operations and *m,* the total number of Make_Set, Find, and Union operations. Since the MAKE-SET operations are included in the total number of operations m, we have m >= n.

- Each union operations decreases the number of sets in the data structure, so the number of Union operations can be at most n-1. After n-1 Union operations only one set remains.

Linked List implementation

- Each set is represented by its own linked list.
- 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. Within each linked list, the objects may appear in any order.
- The representative is the set member in the first object in the list.

Linked List implementation

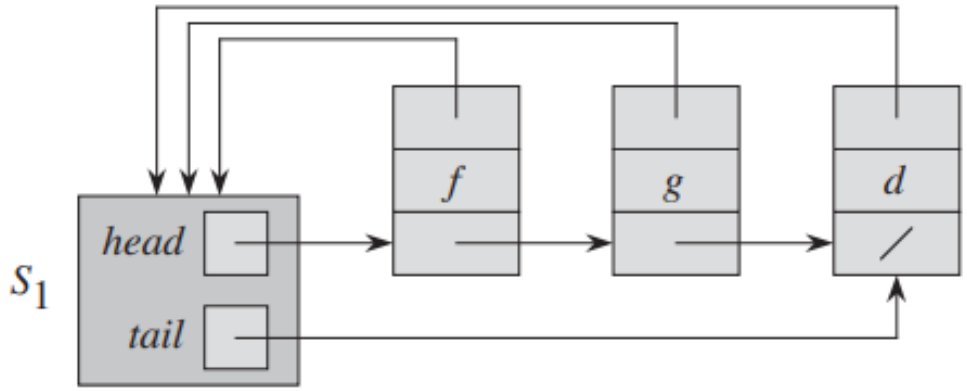
- With this linked-list representation, both MAKE-SET and FIND-SET requires O(1) time.

- To carry out 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.

Linked List implementation

- For example, in the following figure, the call FIND-Set(g) will return f.

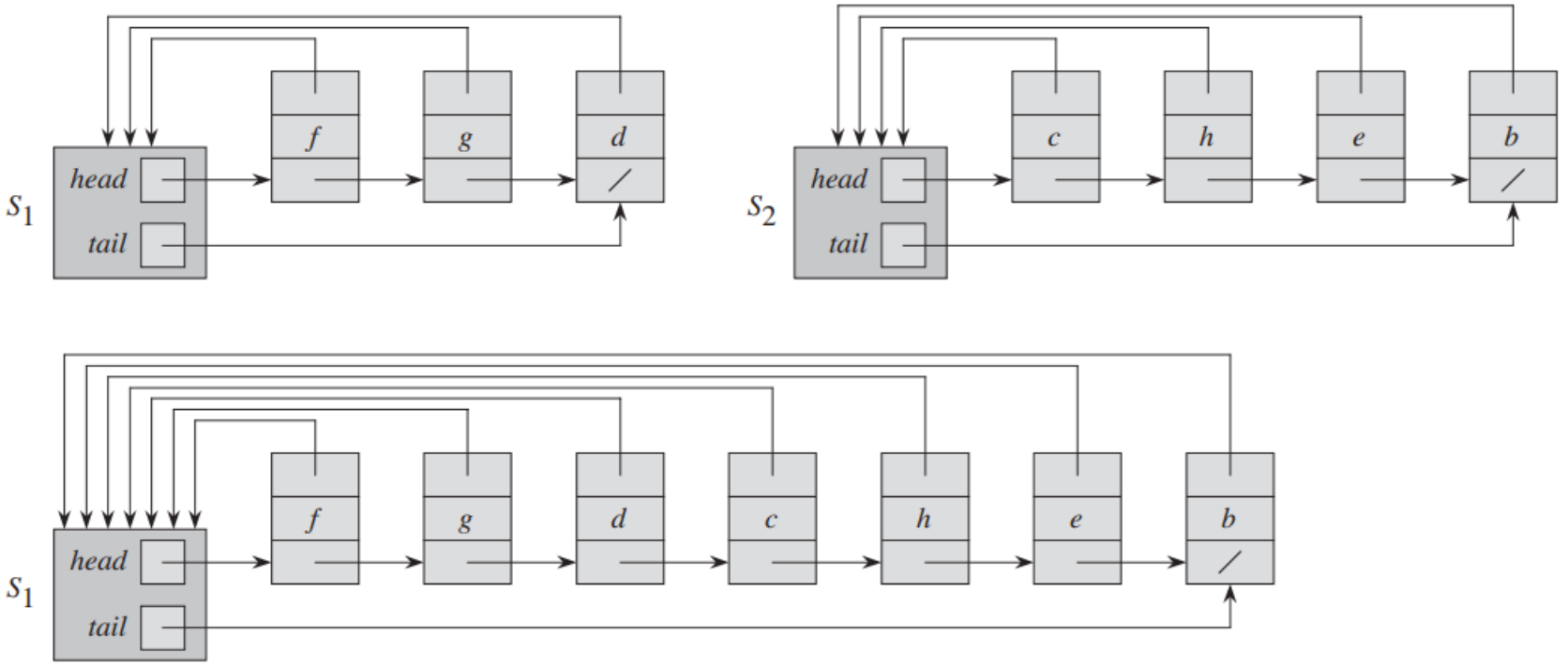


Simple Implementation of Union

- We perform UNION(x,y) by appending y's list onto the end of x's list. The representative of x's list becomes the representative of the resulting set.
- We use the tail pointer for x's list to quickly find where to append y's list. Because all members of y's list join x's list, we can destroy the set object for y's list.
- We must update the pointer to the set object for each object originally on y's list, which takes time linear in the length of y's list.

Simple Implementation of Union

- Result of Union(g,e) is shown in the following figure.



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 back to the set object for every node on y's list.
   - If appending a large list onto a small list, it can take a while.

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

   Amortized time per operation $= \Theta(n)$.

Weighted-Union Heuristic

Instead appending x to y, appending the shorter list to the longer list.

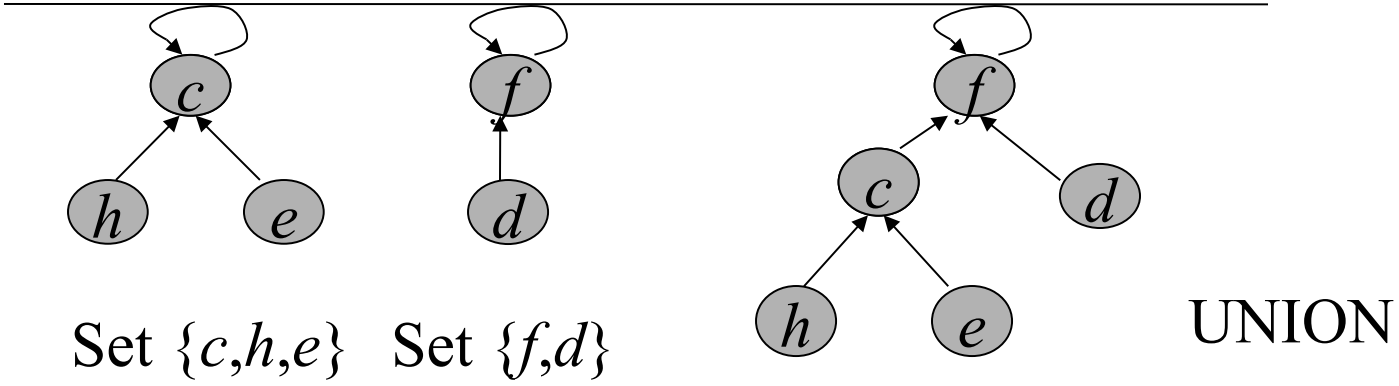
Associated a length with each list, which indicates how many elements in the list.

Result: a sequence of m MAKE-SET, UNION, FIND-SET operations, n of which are MAKE-SET operations, the running time is $O(m+n\lg n)$.

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

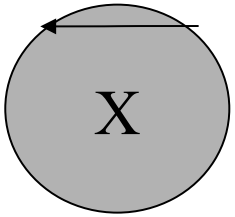
Rooted trees, each tree is a set, root is the representative. Each node points to its parent. Root points to itself.



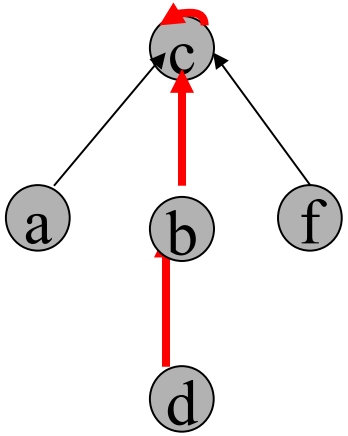
Straightforward Solution

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)



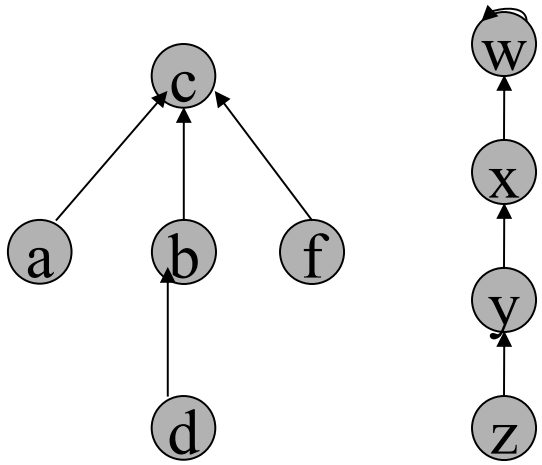
`FIND_SET(d)`

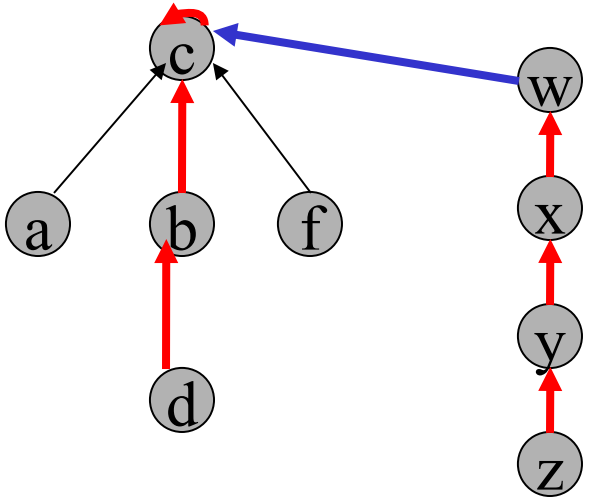
Straightforward Solution

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 $O(n^2)$.



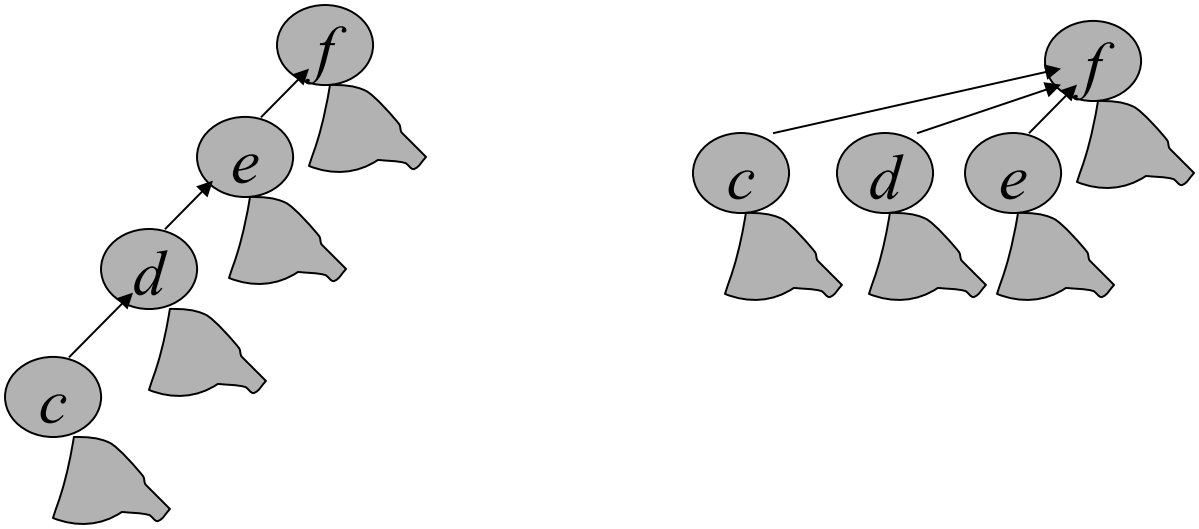


Union by Rank & Path Compression

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.

Path Compression



Algorithm for Disjoint-Set Forest

```
MAKE-SET(x)
1.  p[x]←x
2.  rank[x]←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] ←x
3.  else  p[x] ←y
4.          if rank[x]=rank[y]
5.          then rank[y]++
```

```
FIND-SET(x)
1.  if x≠ p[x]
2.     then p[x] ←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.

Analysis

- In Union we attach a smaller tree to the larger tree, results in logarithmic depth.

- Path compression can cause a very deep tree to become very shallow.
- Combining both ideas gives us (**without proof**) a sequence of m operations in O(m. $\alpha(n)$) where $\alpha(n)$ is very slowly growing function and $\alpha(n) <= 4$.