Amortized Analysis: DISJOINT SETS
(CLRS chpts 17 & 21)

Problem: Maintain a dynamic collection of pairwise disjoint sets $\underline{S} = \{S_1, S_2, S_3, \ldots, S_r\}$ in which each set S_i has one representative element, rep $[S_i]$

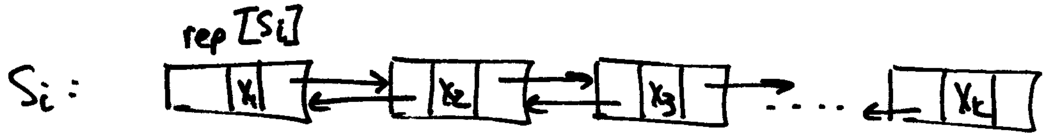
Supported Operations

Make-Set (x): Creates a new set containing the single element x. X must not be a member of any pre-existing set; X is the represent.

Union (x, y): Replace S_x, S_y with $S_x \cup S_y$ in $\underline{S}$ for any x, y in distinct sets S_x, S_y. Update representative.

Find-Set (x): Returns representative rep $[S_x]$ of set S_x containing x.

Solution ① : Doubly-linked list (unordered)



Ops:

Make-Set (x) initializes x as lone node : $\Theta(1)$

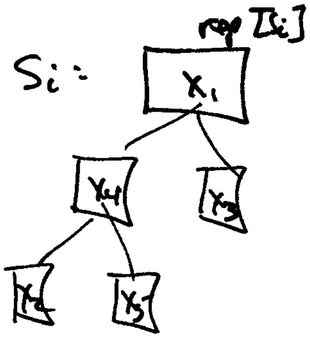
Find-Set (x) walks "left" from x to rep of head : $\Theta(n)$

Union (x,y) concatenates lists containing x and y, leaving front as rep : $\Theta(n)$

Walk to tail of one end and head of other

→ We can improve on this ←

Solution ② : Simple Balanced Tree → Forest



Make-Set (x) : initializes new tree with root node x : $\Theta(1)$

FIND-SET (x) : Walks up tree from x to root : $\Theta(h) \overset{\text{best}}{=} \Theta(\lg n)$

UNION (x,y) : Concatenates trees containing x and y, with overall root as rep : $\Theta(h) \overset{\text{best}}{=} \Theta(\lg n)$

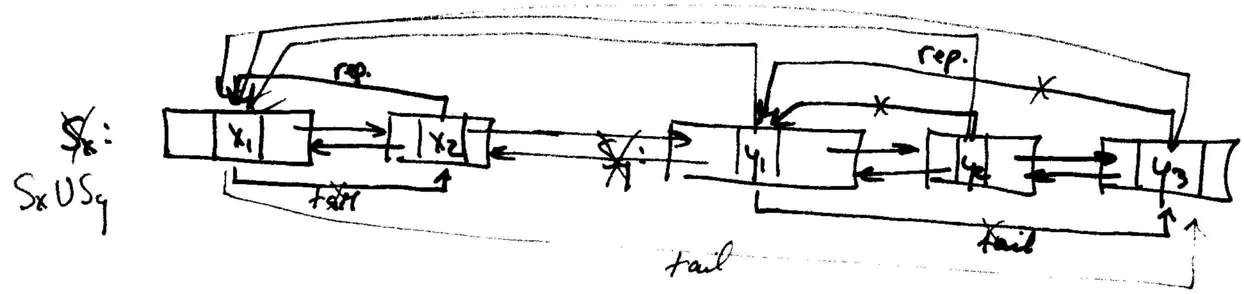
$\Theta(h_x + h_y + 1)$

This data structure supports so few ops
 – no search, no delete, no comparisons
that these running times are much poorer than necessary.
We will make improvements and find we can do substantially better than $\Theta(\lg n)$ or even $\Theta(\lg \lg \lg \cdots \lg n)$, but not quite $\Theta(1)$

Improvements

① Linked-List Improvements   problem here is that need to walk along linked list to find ends

ⓐ Augment the linked list to include pointer from each element to rep. (and from rep. to tail)



Time Analysis

FIND-SET(x)   returns rep[x] : $\Theta(1)$ had been $\Theta(n)$

UNION(x, y)   concatenates lists and moves rep pointers for list containing y : $\Theta(n)$ same as before

→ Θ(length of list in tail position)

↓ AMORTIZED ANALYSIS: Find cost per operation by considering cost of large number of operations and divide out to find cost per operation. Especially useful when some operations (infrequent) occur large overhead costs that wish to spread over other operations (e.g., allocating extra space when table grows → recitation)

For m union operations

→ initial sets $\{1\}, \{2\}, \ldots, \{m\}$

→ union($m-1, m$), union($m-2, m-1$), ..., union($1, 2$)

→ union($m-i, m-i+1$) modifies a list of length i

→ Total cost $= \sum_{i=1}^{m} i = \Theta(m^2)$ → order m per operation

(b) Further improvement: always add smaller list onto larger (fewer pointers to move)
[Must augment data structure to include weights (# elements)]

New amortized cost:

- Bound on cost of m Union operations and n Make-Set ops
- For each list node x, need to bound the number of times its pointer to rep. is updated
- Each time x has pointer updated, length of list containing x grows by a factor ≥ 2
- Thus, the update can happen at most $\lg n$ times
- Additionally, each union takes $\Theta(1)$ time
- Total cost is $\Theta(n \lg n + m)$

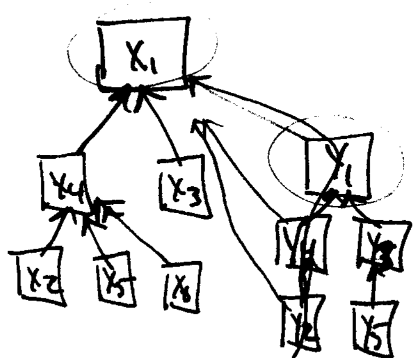
n elements each of which can have no more than $\lg n$ pointer updates (notice that number of merges doesn't enter here)

plus number of merges

② Forest of trees Improvements

- unordered
- possibly unbalanced
- not necessarily binary

- rep[S_x] is root
- tree only stores elements and parental pointers



Union(x, y)

Step 1: FIND-SET (x)

Step 2: FIND-SET (y)

Step 3: Connect root of y to root of x.

ⓐ Always attach smaller tree to larger tree (size is # of nodes)

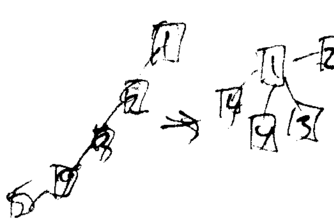
→ depth $= \Theta(\lg n)$

→ proof is as before: each time the depth of x increases (by at most 1) the size of the tree grows by ≥ 2

ⓑ Path Compression

update picture from above

→ After finding the root of T_x, make all traversed nodes (on path from x to root) point to root



→ Extreme example: A very ~~shallow~~ deep tree can become very shallow

→ Can prove that n Make-Set ops and m Find-Set ops and arbitrary number of Unions cost $\Theta(n + m(1 + \log_{2+\frac{m}{n}} n))$ using this improvement alone.

⇒ Really Impressive Result ⇐

Both improvements at once (small onto big & path compression) lead to spectacular time behavior.

Define $A_k(j) = \begin{cases} j+1 & \text{for } k=0 \\ A_{k-1}^{j+1}(j) & \text{for } k \geq 1 \end{cases}$

$A_k(1)$ grows very fast! $(A_4(1) \gg 2^{2048})$

This is Ackermann's function A

Now Define $\alpha(n) = \min\{k : A_k(1) \geq n\}$

This function α grows extremely slowly with n.

<u>Theorem:</u> Any sequence of n operations of the forest data structure with both improvements costs $\Theta(n\alpha(n))$.

<u>Just barely superlinear.</u>

$\boxed{\text{Proof in CLRS §21.4}}$

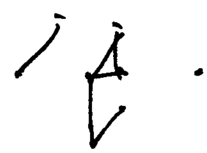
Any given operation barely depends on size of data structure

Why is this important?

Maintaining Dynamic Connectivity Information

Suppose a graph is given incrementally as

add-vertex (v)
add-edge (u,v)



We need to support connectivity queries

connected(u,v) ← are u & v in the same connected component?

add-vertex(v) ⟷ make-set(v)
~~Make-Set(v)~~
add-edge (u,v) ⟷ Union (u, v)

```
connected(u,v)
    if Find-Set(u) = Find-Set(v)
        then return TRUE
        else return FALSE
```

// FAST!!