Union-Find Data Structure (Disjoint Set)

Goal: create a data structure to track a collection of pair-wise disjoint sets.

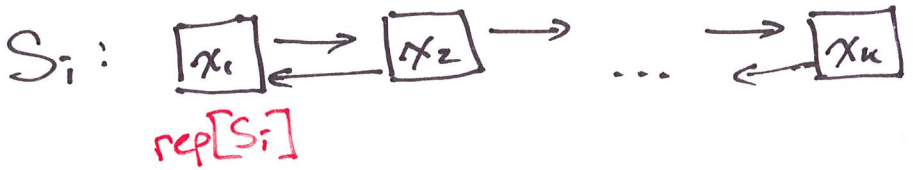
$$S = \{S_1, S_2, S_3, \ldots, S_r\}$$

containing n total elements

- Each S_i is represented by a single arbitrary element $rep[S_i]$
- Operations:
  - MakeSet(x) - creates a new set $\{x\}$, add it to S with $rep[\{x\}] = x$
  - FindSet(x) - determine which set $S_x \in S$ contains x and returns $rep[S_x]$
  - Union(x, y) - replaces S_x and S_y with $S_x \cup S_y$ in S

Linked-List Solution

- represent each set as a doubly-linked list

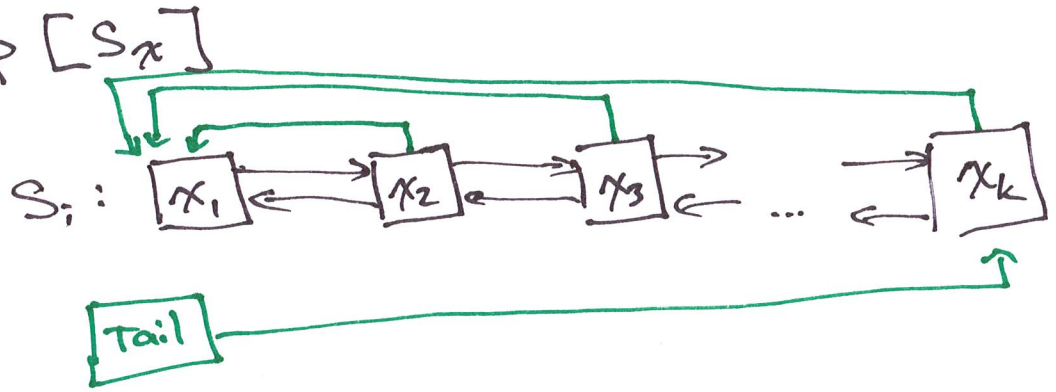


- MakeSet(x): initialize x as the head of a new list O(1)
- FindSet(x): walk left in the list containing x until we reach the head.

  O(n) time in the worst case
- Union(x, y): walk to the tail of S_x and head of S_y and concatenate the lists. O(n) time in the worst case

Augmented Linked List

- augment each item x with a pointer to rep $[S_x]$



- FindSet(x): $O(1)$
- Union(x,y): update the rep pointers for all elements in S_y

  $O(n)$ - time in the worst case

Consider: MakeSet on the integers $1 \ldots n$ and then call Union($n-1$,n), Union($n-2$,$n-1$), ...

Smaller into Larger

- add a weight to each list that tracks the number of elements and use that to always modify the smaller list.
- n calls to MakeSet followed by n calls to union has amortized running time of $\Theta(n \log n)$

Proof (aggregate method)

- suppose the cost to move a rep pointer is 1
- consider some element x and set S_x
- after makeSet(x) we have weigh(S_x) = 1

- when we call Union(x,y) one of the following will happen
  - if weight[S_x] > weight[S_y] then rep(x) is unchanged and weight[S_x] can only increase
  - If weight[S_x] ≤ weight[S_y] we pay 1 to update rep(x) and weight[S_x] at least doubles.
- The weight of S_x can double at most lg n times before S_x contains all n elements
- The rep(x) is updated at most lg n times
- Across all n elements we get an amortized running time of $O(n \log n)$

Forest Representation

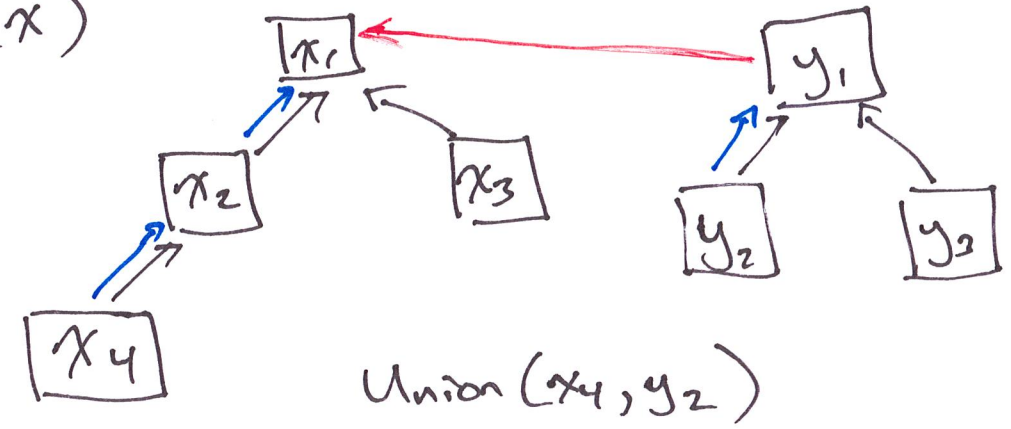
- a set S_x can be represented as a tree with rep[S_x] being the root of the tree containing x.

- Find Set(x): climb the tree to the root

  O(height)

- Union(x, y): climb the trees containing x and y, set the parent of rep(y) to rep(x)

  O(height)



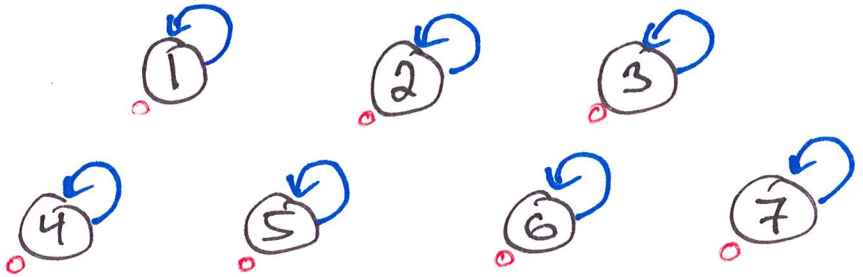
Smaller into Larger

- merge the ~~the~~ tree with smaller height into the taller one
- heights are bounded by $O(\log n)$

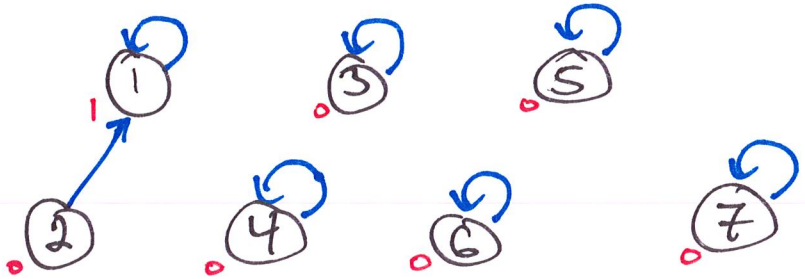
Path Compression

- Improving FindSet
- when we climb the tree we learn the ~~be~~ rep pointers for every intermediate element we pass
- redirect the rep pointers so that future calls to FindSet don't need to repeat the work.

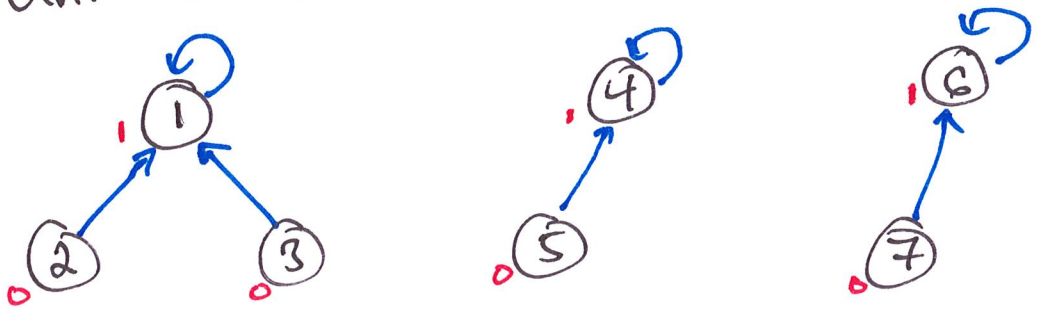
MakeSet on the Integers 1...7



Union(1, 2)



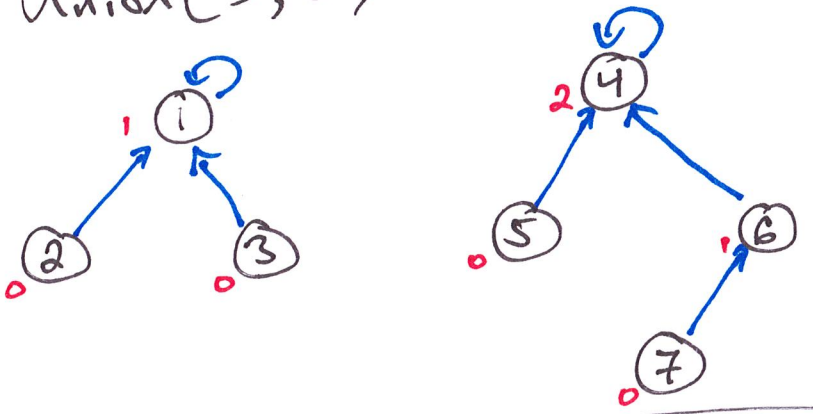
Union(2,3) Union(4,5) Union(6,7)



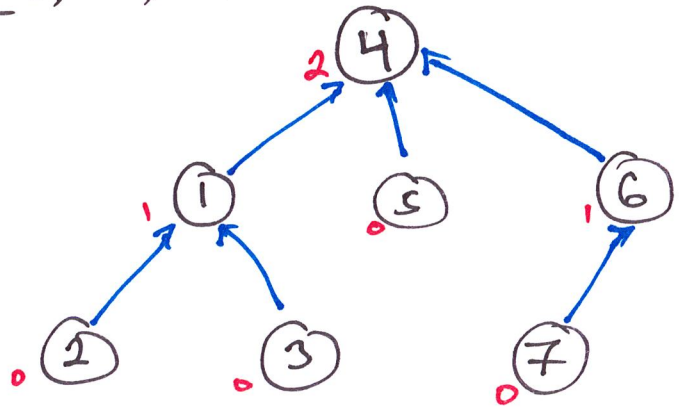

```
Node {
    int data
    int rank
    Node parent
}
```

rank is only used by root nodes.

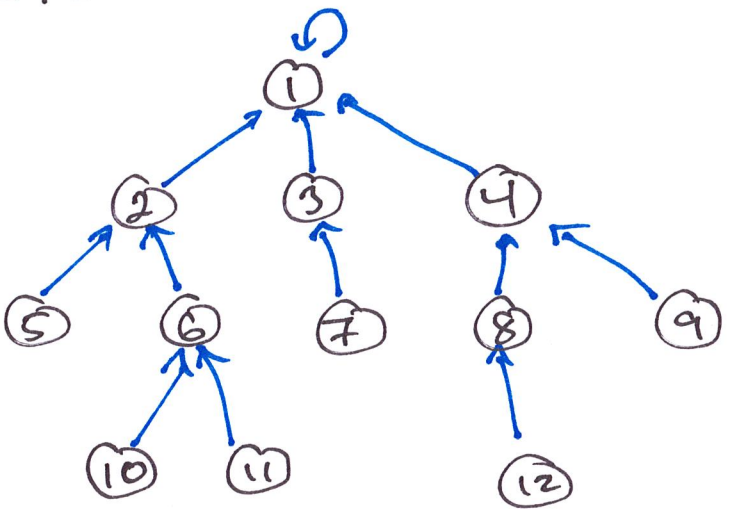
Union(5, 6)



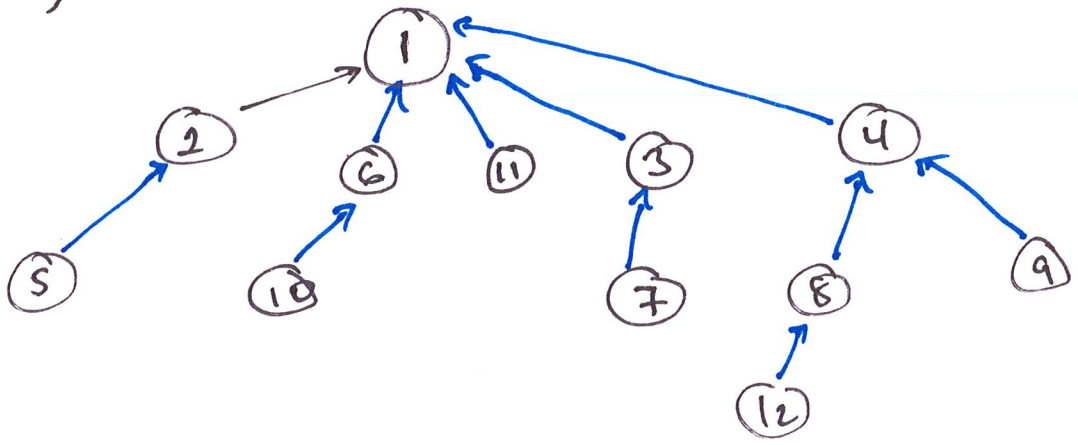
Union(3, 7)



Path Compression



Find Set (11)



Proof CLRS 21.4

A disjoint set with forest representation with union by rank and path compression. Any sequence of m operations, n of which are MakeSet has worst case amortized cost of

$$\Theta(m\,\alpha(n))$$

α is the inverse of Ackermann Function

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

The Ackermann Function A

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

$A_{k-1}^{(j+1)}(j)$: A_{k-1} iterated $j+1$ times

$A_1(1) = 3$

$A_2(1) = 7$

$A_3(1) = 2047$

$A_4(1) >> 10^{80}$