Disjoint Sets

1. Definition:

   - Disjoint sets are a collection of sets with no elements in common.
     I.e. Disjoint sets are sets whose intersection is the empty set.

   - E.g.
     $\{1, 2, 3\}$, $\{4, 5, 6\}$ and $\{7, 8, 9\}$ are disjoint sets.

- A disjoint-set data structure maintains a collection $S = \{S_1, S_2, \ldots S_k\}$ of disjoint dynamic sets.

- We can identify each set by a representative, which is a member of the set.

2. Operations:

   1. make-set(x): Creates a new singleton set containing x.

   2. find(x): Find the set x is in

   3. Union(S, S'): Merges sets S and S'.
      OR
      Union(x, y): Merges x's owner and y's owner. We assume that they are in 2 diff sets.

Note: A set is anything that supports the following:

1. find(x) = find(y)

2. union(find(x), find(y))

3. Implementation:

1. Linked List:

- We will use one linked list per set.
- Each element in the linked list will have a pointer to the rep of the set, which is usually the head of the list.
- Given a seq of m operations, including n make-sets, we can calculate the amortized time, aggregate method, from the following:
  - Given a seq of m operations, including n make-sets, each time we union 2 disjoint sets we have to update the rep field of all the new elements.
  - To save time, we always add the elements from the smaller linked list to the bigger linked list.
  - Each time x.rep is updated, x lands in a list at least twice as long as before. Since the max length of the linked list is

Hilroy

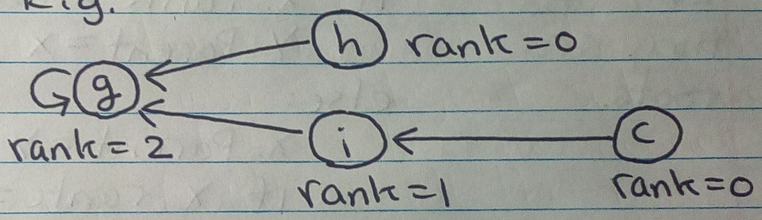
n, at most, we will update each rep field $\lceil \log n \rceil$ times. Since there are n owner fields, so in total, we have at most $n\lceil \log n \rceil$ updates. The other steps cost O(1) per operation, so O(m) time total. Altogether, it is $O(m+n\log n)$ or $O(m\log n)$. This gives an amortized time of $O(\log n)$.

- Pseudo-Code for Union:

```
union(x,y):
# Assume x.rep ≠ y.rep

if x.rep is shorter:
    # transfer x's list content
    # to y's list

    for each item in x.rep:
        item.rep = y.rep

else:
    for each item in y.rep:
        item.rep = x.rep
```

2. Tree:

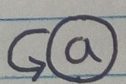
- Each set is a directed tree where directed edges go from children to parents. Furthermore, elements are nodes.
- We use the root to be the representative of the set. I.e. find(x) returns the root of x.
- We will store a rank field at every node. The rank represents the max height at that node.
- E.g.



- Initally, sets begin as single element trees.

I.e.

```
make-set(a):
    a.rep=a
    a.rank=0
```



Hibroy

- If we are unioning 2 disjoint sets using the tree version, we always attach the tree with the lower rank to the tree with the higher rank. This is done to save time. If the ranks are equal, we update the new root's rank by adding 1.

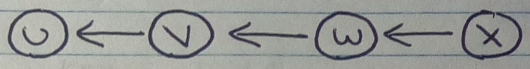
Pseudo Code For Union

```
union(a,b):
Finds the root link(find(a), find(b))

Assume x≠y and they are roots.
link(x,y):
    if x.rank > y.rank:
        y.Parent = x
    else:
        x.Parent = y
        if x.rank == y.rank:
            y.rank++
```

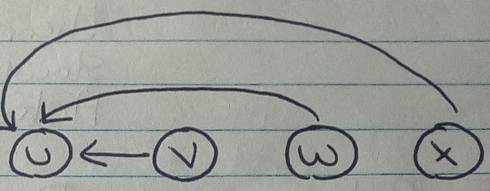
- Note: m calls to {make-set, union, find} becomes up to 3m calls to {make-set, link, find}.

- To find the root from any element, using the tree method, we can start from the element and go up to the root. However, a better way would be to use a path compression. We can update every node along the path to the root directly.

E.g. Instead of doing



we can do



4. Complexity:

- The best disjoint set implementation is trees using union with rank and find with path compression.

- The worst-case time for a seq of m operations, where there are n make-sets is $O(m \log^* n)$.

Note: $\log^*$ is the number of times that you need to apply log to n until the answer is less than 1. Hilroy

E.g. Let $n=40$.

$5 < \log(40) < 6$

$2 < \log(\log(40)) < 3$

$1 < \log(\log(\log(40))) < 2$

$0 < \log(\log(\log(\log(40)))) < 1$

$\therefore \log^* 40 = 4$

- Note: Higher bases give smaller $\log^*$, also called iterated logs.

- Note: The only commonly used function in complexity theory that grows more slowly is the inverse Ackermann function.

5. Ackermann's Function and Inverse

- Let $k \geq 1$, $j \geq 1$.
  We define Ackermann's function to be:

$$\begin{cases} A_0(j) = j+1 \\ A_k(j) = \underbrace{A_{k-1}(A_{k-1}(\dots(j)\dots)}_{\text{we do this } j+1 \text{ times.}} \end{cases}$$

- $A_k(j)$ increases when j or k increases.

- E.g. Let $j=1$

$A_0(1) = 1+1$
$= 2$

$A_1(1) = A_0(A_0(1))$
$= A_0(2)$
$= 2+1$
$= 3$

$A_2(1) = A_1(A_1(1))$
$= A_1(3)$
$= A_0(A_0(A_0(A_0(3))))$
$= A_0(A_0(A_0(4)))$
$= A_0(A_0(5))$
$= A_0(6)$
$= 7$

$A_3(1) = A_2(A_2(1))$
$= A_2(7)$
$= \underbrace{A_1(\ldots A_1(7)\ldots)}_{8 \text{ times}}$
$= 2047$

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

- Note that the Ackermann's Function grows very quickly.

- We can define the inverse Ackermann's Function, denoted by $\alpha(n)$, as:

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

  E.g. $\alpha(10^{80}) = 4$

- Note: The inverse Ackermann's Function grows very slowly. $\alpha(n)$ is practically a constant.

- The amortized times of link and find are $O(\alpha(n))$ if there are n elements.

- $\alpha(|V|) \in O(\lg|V|)$

- Recall that Kruskal's method used disjoint sets. Kruskal's alg had $O(|E|\lg|E| + |E|\alpha(|V|))$ time. Since $O(|E|\lg|E|) \in O(|E|\lg|V|)$, Kruskal's alg takes $O(m\lg n)$ time.