EECS 311: Data Structure and Data Management **Lecture 24**
Union-Find

Reading: Chapter 8.

Union-Find

"a data structure for maintaining **disjoint sets**. Supports operations **union** and **find**."

Recall: Kruskal's MST Algorithm
Input: Graph $G = (V, E)$, edge weights $w(\cdot)$

1. sort edges by weight.
2. for each edge $e = (u, v)$ (in sorted order)
   (a) if u and v already connected, discard edge.
   (b) otherwise, add (u, v) to MST.

Def: Union–Find ADT

- **create(n)**: initializes disjoint sets {1,...,n}
- **union(i,j)**: joins set containing i with set containing j.
- **find(i)**: gives unique identifier for set containing i.
  (Note: need find(i) == find(j) iff i and j in same set)

Claim: exists a union-find data structure that is almost amortized constant time.

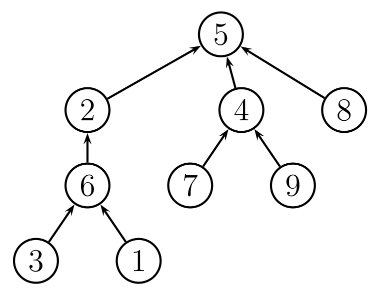
(really: each operation is amortized $f(m)$ time where $f(m) \leq 5$ if m is less than number of particles in the universe).

Idea: keep sets in trees, union merges trees, find returns root.

Note: only ever need to traverse *up* tree, so keep up-pointers.

Idea: tree with up arrows is easy to do in an array

Example:



1	2	3	4	5	6	7	8	9
6	5	6	5	0	2	4	5	4

Algorithm: create(n)

1. array = length n array.
2. array$[1, \ldots, n] = 0$.

Algorithm: find(i)

1. while array$[i] \geq 1$, $i =$ array$[i]$.
2. return i.

Algorithm: union(i, j)

1. $i' = \text{find}(i)$.
2. $j' = \text{find}(j)$.

3. array$[j'] = i'$

Example:

1. create(5)
2. union(1,2)
3. union(3,4)
4. find(4) $\Rightarrow 3$
5. union(5,2)
6. find(1) $\Rightarrow 5$
7. union(2,4)

Idea: union-by-size
"make smaller tree child of root of larger tree"

Claim: with union-by-size, any sequence of m operations after creation costs $\Theta(m \log n)$

Proof:

1. runtime $= m \times$ "worst case depth"
2. depth of any node = number of times its tree was the smaller of the trees in union.
3. if tree is smaller of trees in union, after union size of tree more than doubles.
4. can only double size of tree $\log n$ times before it contains all nodes.
5. maximum depth $= \log n$.

$\square$

Implementation Detail: store "negative size of tree" at root node in array.

Note: union-by-height also works

(trees only get taller when both trees are same height $\Rightarrow$ tree doubles in size.)

Idea: when doing a find, do "path compression"
(relink all nodes on path directly to root)

Example:

1. create(5)
2. union(1,2)
3. union(3,4)
4. union(5,2)
5. union(2,4)
6. find(5)
7. find(4)

Def: $\log^* n$ = the number of logs you can take of n before you're ≤ 1.

Example:

$$\begin{aligned}
\log^* 1 &= 0 \\
\log^*(2^1) &= \log^* 2 = 1 \\
\log^*(2^2) &= \log^* 4 = 2 \\
\log^*(2^4) &= \log^* 16 = 3 \\
\log^*(2^{16}) &= \log^* 65536 = 4 \\
\log^*(2^{65536}) &= 5 \\
\log^*(\text{hubungus!!}) &= 6
\end{aligned}$$

Claim: with path-compression and union-by-size/height m union and find operations from creation costs $O(m \log^* n)$

(actually, it is better, see "inverse Ackermann function")