Disjoint Sets with Arrays
Warm Up

Using the union-by-rank and path-compression optimized implementations of disjoint-sets draw the resulting forest caused by these calls:

1. `makeSet(a)`
2. `makeSet(b)`
3. `makeSet(c)`
4. `makeSet(d)`
5. `makeSet(e)`
6. `makeSet(f)`
7. `makeSet(h)`
8. `union(c, e)`
9. `union(d, e)`
10. `union(a, c)`
11. `union(g, h)`
12. `union(b, f)`
13. `union(g, f)`
14. `union(b, c)`

Reminders:
- **Union-by-rank**: make the tree with the larger rank the new root, absorbing the other tree. If ranks are equal pick one at random, increase rank by 1
- **Path-compression**: when running `findSet()` update parent pointers of all encountered nodes to point directly to overall root
- **Union(x, y)** internally calls `findSet(x)` and `findSet(y)`
Warm Up

Using the union-by-rank and path-compression optimized implementations of disjoint-sets draw the resulting forest caused by these calls:

1. `makeSet(a)`
2. `makeSet(b)`
3. `makeSet(c)`
4. `makeSet(d)`
5. `makeSet(e)`
6. `makeSet(f)`
7. `makeSet(g)`
8. `makeSet(h)`
9. `union(c, e)`
10. `union(d, e)`
11. `union(a, c)`
12. `union(g, h)`
13. `union(b, f)`
14. `union(g, f)`
15. `union(b, c)`

Reminders:
- **Union-by-rank**: make the tree with the larger rank the new root, absorbing the other tree. If ranks are equal pick one at random, increase rank by 1
- **Path-compression**: when running `findSet()` update parent pointers of all encountered nodes to point directly to overall root
- **Union(x, y)** internally calls `findSet(x)` and `findSet(y)`

## Administrivia

<table>
<thead>
<tr>
<th>Monday</th>
<th>Tuesday</th>
<th>Wednesday</th>
<th>Thursday</th>
<th>Friday</th>
</tr>
</thead>
<tbody>
<tr>
<td>5/21</td>
<td></td>
<td>5/23</td>
<td>5/24</td>
<td>5/25</td>
</tr>
<tr>
<td>Disjoint Sets</td>
<td></td>
<td>Implementing Disjoint Sets</td>
<td>Interview Prep</td>
<td>P vs NP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>HW 6 due</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>HW 7 out</td>
</tr>
<tr>
<td>5/28</td>
<td></td>
<td>5/30</td>
<td>5/31</td>
<td>6/1</td>
</tr>
<tr>
<td>Memorial Day</td>
<td></td>
<td>Final Review</td>
<td>Final Review</td>
<td>Tech Interview Prep</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>HW 7 due</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6/5 Final @ 8:30am</td>
</tr>
</tbody>
</table>

**CSE 373 SP 18 - KASEY CHAMPION**

Sorry, Kasey’s email is DEEP
Want a meeting? Email me this week for times next week
Have ANY grading questions/concerns, email Kasey by this weekend
TA lead review TBA
Alternative testing time TBA
# Optimized Disjoint Set Runtime

<table>
<thead>
<tr>
<th>Function</th>
<th>Without Optimizations</th>
<th>With Optimizations</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>makeSet(x)</code></td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td><code>findSet(x)</code></td>
<td>$O(n)$</td>
<td>Best case: $O(1)$ Worst case: $O(\log n)$</td>
</tr>
<tr>
<td><code>union(x, y)</code></td>
<td>$O(n)$</td>
<td>Best case: $O(1)$ Worst case: $O(\log n)$</td>
</tr>
</tbody>
</table>
**Kruskal’s**

\[ t_m = \text{time to make MSTs} \]
\[ t_f = \text{time to find connected components} \]
\[ t_u = \text{time to union} \]

**KruskalMST(Graph G)**

- initialize each vertex to be a connected component
- sort the edges by weight
- ```
```
  ```
  foreach(edge (u, v) in sorted order){
    if(u and v are in different components){
      add (u,v) to the MST
      Update u and v to be in the same component
    }
  }
  ```
- \[ t_m = O(1) \]
- \[ t_f = O(\log V) \]
- \[ t_u = O(\log V) \]

**KruskalMST(Graph G)**

- initialize a disjointSet, call makeSet() on each vertex
- sort the edges by weight
- ```
```
  ```
  foreach(edge (u, v) in sorted order){
    if(findSet(u) != findSet(v)){
      add (u,v) to the MST
      union(u, v)
    }
  }
  ```
- \[ O(V) \]
- \[ O(E) \]
- \[ O(\log V) \]

Aside: \[ O(V + E\log V + E) \] if you apply ackermann
KruskalMST(Graph G)
    initialize a disjointSet, call makeSet()
on each vertex
    sort the edges by weight
    foreach(edge (u, v) in sorted order){
        if(findSet(u) != findSet(v)){
            add (u,v) to the MST
            union(u, v)
        }
    }
KruskalMST(Graph G)
    initialize a disjointSet, call makeSet()
on each vertex
    sort the edges by weight
    foreach(edge (u, v) in sorted order){
        if(findSet(u) != findSet(v)){
            union(u, v)
        }
    }
Implementation

Use Nodes?

In modern Java (assuming 64-bit JDK) each object takes about 32 bytes
- int field takes 4 bytes
- Pointer takes 8 bytes
- Overhead ~ 16 bytes
- Adds up to 28, but we must partition in multiples of 8 => 32 bytes

Use arrays instead!
- Make index of the array be the vertex number
  - Either directly to store ints or representationally
  - We implement makeSet(x) so that we choose the representative
- Make element in the array the index of the parent
Array Implementation

rank = 0

rank = 3

rank = 3

Store \((\text{rank} \times -1) - 1\)

Each “node” now only takes 4 bytes of memory instead of 32
Practice

rank = 2

rank = 0

rank = 1

rank = 2

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>-3</td>
<td>3</td>
<td>-1</td>
<td>-2</td>
<td>6</td>
<td>12</td>
<td>13</td>
<td>13</td>
<td>0</td>
<td>13</td>
<td>-3</td>
<td>12</td>
<td>12</td>
<td>12</td>
</tr>
</tbody>
</table>
Array Method Implementation

`makeSet(x)`
add new value to array with a rank of -1

`findSet(x)`
Jump into array at index/value you’re looking for, jump to parent based on element at that index, continue until you hit negative number

`union(x, y)`
`findSet(x)` and `findSet(y)` to decide who has larger rank, update element to represent new parent as appropriate
Graph Review

Graph Definitions/Vocabulary
- Vertices, Edges
- Directed/undirected
- Weighted
- Etc...

Graph Traversals
- Breadth First Search
- Depth First Search

Finding Shortest Path
- Dijkstra’s

Topological Sort

Minimum Spanning Trees
- Primm’s
- Kruskal’s

Disjoint Sets
- Implementing Kruskal’s
Interview Prep

Treat it like a standardized test
- Cracking the Coding Interview
- Hackerrank.com
- Leetcode.com

Typically 2 rounds

Tech screen
“on site” interviews

4 general types of questions
- Strings/Arrays/Math
- Linked Lists
- Trees
- Hashing
- Optional: Design

It’s a conversation!
1. T – Talk
2. E – Examples
3. B – Brute Force
4. O – Optimize
5. W – Walk through
6. I – Implement
7. T – Test