Union-Find

Union-find stores a bunch of disjoint sets. Supports:

- from a member, **find** its owning set
- **union** two sets

Basic idea:

- Each set is a directed tree. Members are vertices.
- Use root to stand for the set.
- Edges go from children to parents. (So, only parent pointers.)
Union-Find: MakeSet

Initially, sets begin as single-element sets.

makeSet(a): create a set that has just one element a.
Union-Find: Union

To union two sets: have smaller set’s root point to larger set’s root.

union(b, g):

This requires storing a “size” field with each root, and updating at each union.
Union-Find: Find

To find set from element: follow pointers until root.

\[
\begin{array}{cccc}
  u & v & w & x \\
  \text{etc.} & \text{etc.} & \text{etc.} & \text{etc.}
\end{array}
\]

\text{find(x)} \text{ returns } u.

Moreover, path compression: have every node along the path point to root directly.

\[
\begin{array}{cccc}
  u & v & w & x \\
  \text{etc.} & \text{etc.} & \text{etc.} & \text{etc.}
\end{array}
\]
One More Point on Union

What happens to union(x,y), if x or y is not a root?

Option 1: Disallow.

Option 2: Implicitly convert to union(find(x), find(y)).

$n$ operations become at most $3 \cdot n$ basic operations.
Size and Rank

Each root has a size field = tree size.

What if a root becomes non-root after union? Imagine keeping it around, don’t change it anymore.

Note that due to path compression, non-roots lose descendents gradually.

So each non-root has a size field ≥ subtree size.

Define \( rank(v) = \lfloor \log(v.size) \rfloor \).
rank(parent) > rank(child)

\[ \text{rank}(v) = \lfloor \log(v.\text{size}) \rfloor. \]

If \( v \) is non-root and \( v \)'s parent is \( w \):

\( w \) became \( v \)'s parent by a union in the past.
We always merge smaller trees into larger trees.

\[
\text{rank}(w) &= \log[\text{w.size}] \\
&\geq \log[\text{(old w.size) + v.size}] \\
&\geq \log[2 \cdot v.\text{size}] \\
&\geq 1 + \log[\text{v.size}] \\
&> \text{rank}(v)
\]
How Many Nodes Have Same Rank

Parent rank > child rank. Ancestor rank > descendent rank.

(Assume \( v \neq w \))
If \( v, w \) have the same rank \( s \), then their descendents are disjoint.

\( v.size \) and \( w.size \) talk about two disjoint groups.

\[ \log(v.size) \geq s \]
\[ v.size \geq 2^s \]

Globally \( n \) elements, divide them into groups of at least \( 2^s \), you get at most \( n/2^s \) groups.

Each node of rank \( s \) claims one group.

You can have at most \( n/2^s \) such nodes.
Tower of 2 and Iterated Log

Define

\[ \text{tower}(0) = 1 \]
\[ \text{tower}(i + 1) = 2^{\text{tower}(i)} \]

Inverse function in a sense:

\[ \log^*(n) = \min\{i \in \mathbb{N} : n \leq \text{tower}(i)\} \]

\[ \log^*(n) \leq i \iff n \leq \text{tower}(i) \]

E.g., \( \log^*(n) \leq 5 \) iff \( n \leq \text{tower}(5) = 2^{65536} \)

“practically \( O(5) \)”
## Tower of 2 and Iterated Log

<table>
<thead>
<tr>
<th>$n$</th>
<th>$\log^*(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 to 1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3 to 4</td>
<td>2</td>
</tr>
<tr>
<td>5 to 16</td>
<td>3</td>
</tr>
<tr>
<td>17 to 65536</td>
<td>4</td>
</tr>
<tr>
<td>65537 to $2^{65536}$</td>
<td>5</td>
</tr>
</tbody>
</table>
Define: \( v \) and \( w \) are in the same “rank group” iff:

\[
\log^* (\text{rank}(v)) = \log^* (\text{rank}(w))
\]

Use \( \log^* (\text{rank}(v)) \) as the name of \( v \)'s rank group.

If globally at most \( n \) elements, largest possible rank group name:

\[
\log^* [\log n] \leq \log^* (n) - 1
\]

Rank group \( g > 0 \) has \( \text{tower}(g) - \text{tower}(g - 1) \) ranks.
How Many Nodes in Same Rank Group

Assume $g > 0$.

# of nodes in rank group $g$

\[
\leq \sum_{s=\text{tower}(g-1)+1}^{\text{tower}(g)} \frac{n}{2^s}
\]

\[
< \sum_{s=\text{tower}(g-1)+1}^{\infty} \frac{n}{2^s}
\]

\[
= 2 \cdot \frac{n}{2^{\text{tower}(g-1)+1}}
\]

\[
= \frac{n}{\text{tower}(g)}
\]
Amortized Time Analysis: Outline

- union(root1, root2) is $O(1)$. Just need to figure out find.
- A mix of the accounting method and an aggregate argument.
- If find(x) goes through $k$ edges, it receives $k$ dollars and spends it all. But . . .
- Classify those $k$ dollars into two categories.
- For one category, get an upper bound per find.
- For another category, get a total upper bound over all operations. (Aggregate argument.)
Dollars for Find

$1$ for each edge $\text{find}(x)$ goes through. But there are two cases.

Let $(v, w)$ be an edge. ($w$ parent, $v$ child.)

- $w$ is root or $w$’s rank group $\neq v$’s
  - At most $\log^*(n)$ rank groups.
  - At most $\log^*(n) - 1$ edges crossing rank groups.
  - At most $\log^*(n)$ edges in this category.
  - At most $\log^*(n)$ dollars per $\text{find}(x)$ under this category.

- $w$ is not root, and $v$ and $w$ same rank group
  - Call this “$1$ charged to node $v$”
  - Will use an aggregate argument for dollars in this category.
Dollars Charged to A Node, Ever

$1$ charged to $v$ iff $v$'s parent is not root and they are in the same rank group.

Every time $1$ is charged to $v$, find($x$) performs path compression, $v$ gets new parent, higher rank than (ancestor of) old parent.

Eventually $v$ gets a parent in a different rank group. Then no more dollars charged to $v$.

If $v$ is in rank group $g > 0$, then at most $\text{tower}(g) - \text{tower}(g - 1)$ dollars charged to $v$, ever.

(If $v$ is in rank group $0$, then parent is root or parent is in different rank group.)
Dollars Charged to All Nodes, Ever

Upper bound:

\[
\log^*(n) - 1 \leq \sum_{g=1}^{\log^*(n) - 1} (\text{# nodes in rank group } g) \cdot (\text{tower}(g) - \text{tower}(g - 1)) \\
\leq \sum_{g=1}^{\log^*(n) - 1} \left( \frac{n}{\text{tower}(g)} \right) \cdot (\text{tower}(g) - \text{tower}(g - 1)) \\
\leq \sum_{g=1}^{\log^*(n) - 1} \left( \frac{n}{\text{tower}(g)} \right) \cdot (\text{tower}(g)) \\
= \sum_{g=1}^{\log^*(n) - 1} n \\
\leq n \cdot \log^*(n)
\]
Amortized Time: Conclusion

In a sequence of $n$ operations:

- at most $n$ elements globally
- makeSet takes $O(1)$ time
- union(root1, root2) takes $O(1)$ time
- find(x) first category takes $O(\log^* (n))$ time each
- find(x) second category takes $O(n \cdot \log^* (n))$ time total over all $n$ operations

Total time $O(n \cdot \log^* (n))$.

Amortized time $O(\log^* (n))$. “Practically $O(5)$.”