The Union-Find Problem

The Problem: Given a set $X$ of $n$ elements $x_1, x_2, \ldots, x_n$. We would like to maintain a collection of disjoint subsets (groups) of $X$.

Initially, the collection is empty.

There are three operations on the elements and the subsets.

**Make_set(i):** makes $x_i$ a subset and assigns a name for the subset.

**Find(i):** returns the name of the subset that contains $x_i$.

**Union(i, j):** combines subsets that contain $x_i$ and $x_j$, say $S_i$ and $S_j$, into a new subset with a unique name. (Any name distinct from other names will do.)

**The goal:** Design a data structure that will support any sequence of these three operations as efficient as possible.

*Note:* We assume the types for elements are subrange type. Therefore we can use elements name to index into array (e.g. integer 1, \ldots, $n$)
A simple (naive) solution

Store the name of the subset containing the $i$’th element $x_i$ in $A[i]$.

- **Make_set(i)**: we just set $A[i]$ to $i$.
- **Find(i)**: we just look at $A[i]$ and find out the name for the subset.
- **Union(i, j)**: (Assume the name of the resulting subset is $S_i$’s name) Change the subset name for all elements in $S_j$.

**Example:**

<table>
<thead>
<tr>
<th>Make_set(1 ... 7)</th>
<th>1 2 3 4 5 6 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Union(1, 2)</td>
<td>1 1 3 4 5 6 7</td>
</tr>
<tr>
<td>Union(5, 6)</td>
<td>1 1 3 4 5 5 7</td>
</tr>
<tr>
<td>Find(6)</td>
<td>5</td>
</tr>
<tr>
<td>Union(1, 5)</td>
<td>1 1 3 4 1 1 7</td>
</tr>
<tr>
<td>Union(3, 1)</td>
<td>3 3 3 4 3 3 7</td>
</tr>
</tbody>
</table>

_Time:_ $n$ union operations may need $O(n^2)$ time.
An improved implementation

- Each set is represented by a linked list.
- The first node in each list serves as its set’s representative.
- Each node of the list contains a set member, a pointer to the next node, and a pointer back to the representative.
- Each list maintains a pointer, head, to the first node and a pointer, tail, to the last node.
- Make_set(i) and Find(i) are easy to implement.
- For the Union(i,j), we will append the smaller list onto the longer list and update representative pointers of the smaller list.

Time: with a sequence of \( m \) operations, \( n \) of which are Make_set operations, it takes \( O(m + n \log n) \) time.

Why: how many times a pointer to its representative can be changed?
Example:

```
4 2 7 6 3 5 1
```

```
4
```

```
7 2 4
```

```
1 5 3 6 7 2 4
```

```
head
```

```
tail
```
Another implementation

- Instead of making *Find* operation simple, we make *Union* operation simple.
- Each set is a tree and each node in a tree is a record: one field for element name (not really necessary), one field for a pointer (parent pointer) to another node.

† **Find**(*i*): from entry *i*, follow parent pointer until we find a node with a nil pointer, or a pointer pointing to itself (root). Return the name in that node.

† **Union**(*i, j*): we change the pointer of the root of set *S_j* to pointing to the root of set *S_i*, or vice-versa.
Example:

- Make_set(1 ... 7)
  - Union(1, 2)
  - Union(5, 6)
  - Find(6)
  - Union(1, 5)
  - Union(3, 1)
We can consider the whole structure as a forest.

Make_set(1 ... 7)

Union(1, 2)

Union(5, 6)

Find(6)

Union(1, 5)

Union(3, 1)
Efficient *Union-Find*

• *Idea:* balance and collapse the trees.

• Balancing: when union operation is performed, the root pointer of the smaller tree is set to point to the root of larger tree.

  † Rather than explicitly keeping the size of the subtree rooted at each node, we use another approach.

  † For each node, we maintain a rank that is an upper bound on the height of that node.

  † In union by rank, the root with smaller rank is made to point to the root with larger rank during an Union operation.

    – If two roots have equal ranks, we arbitrarily choose one of the roots as the parent, increase its rank by 1, and reset the other root.

    – With Make_set(), the rank is set to 0.
• (1) If union by rank is used, then for any node, its height is bounded by its rank.
• (2) If union by rank is used, then for any node $i$, its rank is bounded by $\log(\text{size}(i))$.

Proof: (of (1)) Induction on the number of Make_set and Union operations.

Base case: the first operation must be Make_set, and (1) is true since we have one node with height 0 and rank 0.

Induction step: consider an $\text{Union}(i, j)$ operation and let $r_i$ and $r_j$ be the roots of the trees containing $i$ and $j$. We assume that $\text{height}(r_i) \leq \text{rank}[r_i]$ and $\text{height}(r_j) \leq \text{rank}[r_j]$.

† If $\text{rank}[r_i] > \text{rank}[r_j]$, \[ \text{height(union}(i, j)) = \max\{\text{height}(r_i), \text{height}(r_j) + 1\} \leq \text{rank}[r_i] = \text{rank}[\text{root}(\text{union}(i, j))]. \]

† If $\text{rank}[r_i] < \text{rank}[r_j]$, \[ \text{height(union}(i, j)) = \max\{\text{height}(r_i) + 1, \text{height}(r_j)\} \leq \text{rank}[r_j] = \text{rank}[\text{root}(\text{union}(i, j))]. \]

† If $\text{rank}[r_i] = \text{rank}[r_j]$, \[ \text{height(union}(i, j)) \leq \max\{\text{height}(r_i) + 1, \text{height}(r_j) + 1\} \leq \text{rank}[r_i] + 1 = \text{rank}[\text{root}(\text{union}(i, j))]. \]  
\[ \Box \]
Proof: (of (2)) Induction on the number of Make_set and Union operations.
• With balancing (union by rank), for a sequence of $m$ operations, $n$ of which are Make_set operations, the height of any tree is less than or equal to $\log n$, since we only have $n$ elements.

• Any find operation is at most $O(\log n)$

• Any sequence of $m \geq n$ operations will be bounded by $O(m \log n)$.

*Union*: constant time.

*Find*: $O(\log n)$ time.
Path compression (collapse the tree)

In the operation of $\text{Find}(i)$, do following:

**first pass**: follow parent pointer to find the root

**second pass**: follow parent pointer and change each of the pointers in the path to point to root.

With path compression alone, for a sequence of $m$ operations, $n$ of which are Make_set operations, the time complexity is $O(m \log n)$. 
**Theorem.** If both balancing and path comparisons are used, then the total number of steps in the worst case for any sequence of $m \geq n$ operations, $n$ of which are Make_set operations, is $O(m \log^* n)$.

**Proof:** Omitted.

\[
\log^*(1) = 0, \quad \log^*(2) = 1.
\]

\[
\log^*(n) = 1 + \log^*(\lceil \log_2 n \rceil), \quad n \geq 2.
\]

\[
\begin{align*}
\log^*(2) &= 1, \quad 2 = 2 \\
\log^*(2^2) &= 2, \quad 2^2 = 4 \\
\log^*(2^{2^2}) &= 3, \quad 2^{2^2} = 2^4 = 16 \\
\log^*(2^{2^{2^2}}) &= 4, \quad 2^{2^{2^2}} = 2^{16} = 65536 \\
\log^*(2^{2^{2^{2^2}}}) &= 5, \quad 2^{2^{2^{2^2}}} = 2^{65536}
\end{align*}
\]

The number of atoms in the observable universe is estimated to be about $10^{80}$ which is MUCH SMALLER than $2^{65536}$!!

In practice, above union-find algorithm is linear time.
An implementation of efficient union-find data structure.

Make-set($x$)

\[ \text{parent}[x] = x; \quad \text{rank}[x] = 0; \]

Union($x, y$)

\[ \text{Link}(\text{Find-set}(x), \text{Find-set}(y)); \]

Link($x, y$)

\[
\begin{align*}
& \text{if} \ (\text{rank}[x] > \text{rank}[y]) \\
& \quad \text{parent}[y] = x; \\
& \text{else if} \ (\text{rank}[x] < \text{rank}[y]) \\
& \quad \text{parent}[x] = y; \\
& \text{else if} \ (x \neq y) \\
& \quad \text{parent}[y] = x; \quad \text{rank}[x] = \text{rank}[x] + 1;
\end{align*}
\]

Find-set($x$)

\[
\begin{align*}
& \text{if} \ x \neq \text{parent}[x] \\
& \quad \text{parent}[x] = \text{Find-set}(\text{parent}[x]) \\
& \text{return}(\text{parent}[x])
\end{align*}
\]