Disjoint-sets data type

**Goal.** Support three operations on a collection of disjoint sets.
- **MAKE-SET(x):** create a new set containing only element x.
- **FIND(x):** return a canonical element in the set containing x.
- **UNION(x,y):** replace the sets containing x and y with their union.

**Performance parameters.**
- $m =$ number of calls to MAKE-SET, FIND, and UNION.
- $n =$ number of elements = number of calls to MAKE-SET.

**Dynamic connectivity.** Given an initially empty graph $G$, support three operations.
- **ADD-NODE(u):** add node u.
- **ADD-EDGE(u,v):** add an edge between nodes u and v.
- **IS-CONNECTED(u,v):** is there a path between u and v?

---

**Original motivation.** Compiling EQUIVALENCE, DIMENSION, and COMMON statements in Fortran.

An Improved Equivalence Algorithm

EDWARD A. GALEK and MICHAEL J. FISHER
University of Michigan, Ann Arbor, Michigan

An algorithm for assigning storage on the basis of EQUIVALENCE, DIMENSION and COMMON declarations is presented. The algorithm is based on a tree structure, and has reduced computation time by 40 percent over a previously published algorithm by identifying all equivalence classes with one scan of the EQUIVALENCE declarations. The method is applicable in any problem in which it is necessary to identify equivalence classes, given the element pairs defining the equivalence relation.

**Note.** This 1964 paper also introduced key data structure for problem.
Disjoint-sets data structure

**Parent-link representation.** Represent each set as a tree of elements.
- Each element has an explicit parent pointer in the tree.
- The root serves as the canonical element (and points to itself).
- \( \text{FIND}(x) \): find the root of the tree containing \( x \).
- \( \text{UNION}(x, y) \): merge trees containing \( x \) and \( y \).

**Array representation.** Represent each set as a tree of elements.
- Allocate an array \( \text{parent}[\_] \) of length \( n \).
- \( \text{parent}[i] = j \) means parent of element \( i \) is element \( j \).

\begin{verbatim}
parent[] = [7 5 7 8 8 7 5 7 8 8]
\end{verbatim}

**Note.** For brevity, we suppress arrows and self loops in figures.
Naïve linking

Naïve linking. Link root of first tree to root of second tree.

\textbf{Naïve linking: analysis}

\textbf{Theorem.} Using naïve linking, a UNION or FIND operation can take $\Theta(n)$ time in the worst case, where $n$ is the number of elements.

\textbf{Pf.}

- In the worst case, FIND takes time proportional to the height of the tree.
- Height of the tree is $n - 1$ after the sequence of union operations: $\text{UNION}(1, 2), \text{UNION}(2, 3), \ldots, \text{UNION}(n - 1, n)$. 

\begin{align*}
\text{MAKE-SET}(x) & \quad \text{parent}[x] \leftarrow x. \\
\text{FIND}(x) & \quad \text{WHILE} \; (x \neq \text{parent}[x]) \\
& \quad \quad x \leftarrow \text{parent}[x]. \\
& \quad \text{RETURN} \; x. \\
\text{UNION}(x, y) & \quad r \leftarrow \text{FIND}(x). \\
& \quad s \leftarrow \text{FIND}(y). \\
& \quad \text{parent}[r] \leftarrow s.
\end{align*}
**UNION–FIND**

- naïve linking
- link-by-size
- link-by-rank
- path compression
- link-by-rank with path compression
- context

---

**Link-by-size**

**Link-by-size.** Maintain a tree size (number of nodes) for each root node. Link root of smaller tree to root of larger tree (breaking ties arbitrarily).

**UNION(5, 3)**

- size = 4
- size = 6

---

**MAKE-SET(x)**

- parent[x] ← x.
- size[x] ← 1.

**FIND(x)**

- WHILE (x ≠ parent[x])
  - x ← parent[x].
- RETURN x.

**UNION(x, y)**

- r ← FIND(x).
- s ← FIND(y).
- IF (r = s) RETURN.
- ELSE IF (size[r] > size[s])
  - parent[s] ← r.
  - size[r] ← size[r] + size[s].
- ELSE
  - parent[r] ← s.
  - size[r] ← size[r] + size[s].

---

**Link-by-size**

**Link-by-size.** Maintain a tree size (number of nodes) for each root node. Link root of smaller tree to root of larger tree (breaking ties arbitrarily).
**Link-by-size: analysis**

**Property.** Using link-by-size, for every root node $r$:
\[ \text{size}[r] \geq 2^{\text{height}(r)}. \]

**Pf.** (by induction on number of links)
- Base case: singleton tree has size 1 and height 0.
- Inductive hypothesis: assume true after first $i$ links.
- Tree rooted at $r$ changes only when a smaller (or equal) size tree rooted at $s$ is linked into $r$.
- Case 1. $\text{height}(r) > \text{height}(s)$
  \[ \text{size}'[r] > \text{size}[r] \]
  \[ \geq 2^{\text{height}(r)} \quad \text{inductive hypothesis} \]
  \[ = 2^{\text{height}(r)}. \]
- Case 2. $\text{height}(r) \leq \text{height}(s)$
  \[ \text{size}'[r] = \text{size}[r] + \text{size}[s] \]
  \[ \geq 2 \cdot \text{size}[s] \quad \text{link-by-size} \]
  \[ \geq 2 \cdot 2^{\text{height}(s)} \quad \text{inductive hypothesis} \]
  \[ = 2^{\text{height}(s) + 1} \]
  \[ = 2^{\text{height}(r)}. \quad \]

**Link-by-size: analysis**

**Theorem.** Using link-by-size, any UNION or FIND operation takes $O(\log n)$ time in the worst case, where $n$ is the number of elements.

**Pf.**
- The running time of each operation is bounded by the tree height.
- By the previous property, the height is $\leq \lfloor \log n \rfloor$.

\[ \log n = \log_2 n \]

**Note.** The UNION operation takes $O(1)$ time except for its two calls to FIND.

**Link-by-size: analysis**

**Property.** Using link-by-size, for every root node $r$:
\[ \text{size}[r] \geq 2^{\text{height}(r)}. \]

**Pf.** (by induction on number of links)
- Base case: singleton tree has size 1 and height 0.
- Inductive hypothesis: assume true after first $i$ links.
- Tree rooted at $r$ changes only when a smaller (or equal) size tree rooted at $s$ is linked into $r$.
- Case 1. $\text{height}(r) > \text{height}(s)$
  \[ \text{size}'[r] > \text{size}[r] \]
  \[ \geq 2^{\text{height}(r)} \quad \text{inductive hypothesis} \]
  \[ = 2^{\text{height}(r)}. \]
- Case 2. $\text{height}(r) \leq \text{height}(s)$
  \[ \text{size}'[r] = \text{size}[r] + \text{size}[s] \]
  \[ \geq 2 \cdot \text{size}[s] \quad \text{link-by-size} \]
  \[ \geq 2 \cdot 2^{\text{height}(s)} \quad \text{inductive hypothesis} \]
  \[ = 2^{\text{height}(s) + 1} \]
  \[ = 2^{\text{height}(r)}. \quad \]

**A tight upper bound**

**Theorem.** Using link-by-size, a tree with $n$ nodes can have height $= \log n$.

**Pf.**
- Arrange $2^k - 1$ calls to UNION to form a binomial tree of order $k$.
- An order-$k$ binomial tree has $2^k$ nodes and height $k$. 

\[ B_0 \quad B_1 \quad B_2 \quad B_3 \quad B_4 \]
**SECTION 5.1.4**

**UNION–FIND**

- naïve linking
- link-by-size
- link-by-rank
- path compression
- link-by-rank with path compression
- context

**Link-by-rank**

Maintain an integer rank for each node, initially 0. Link root of smaller rank to root of larger rank; if tie, increase rank of new root by 1.

**UNION(7, 3)**

Note. For now, rank = height.

```plaintext
MAKE-SET(x)
parent[x] ← x.
rank[x] ← 0.
```

```plaintext
UNION(x, y)
r ← FIND(x).
s ← FIND(y).

IF (r = s) RETURN.
ELSE IF (rank[r] > rank[s])
    parent[s] ← r.
ELSE IF (rank[r] < rank[s])
    parent[r] ← s.
ELSE
    parent[r] ← s.
    rank[s] ← rank[s] + 1.
```

Note. For now, rank = height.

---

**Link-by-rank**

Maintain an integer rank for each node, initially 0. Link root of smaller rank to root of larger rank; if tie, increase rank of new root by 1.

```
Link-by-rank

Link-by-rank. Maintain an integer rank for each node, initially 0. Link root of smaller rank to root of larger rank; if tie, increase rank of new root by 1.

UNION(7, 3)

Note. For now, rank = height.

MAKE-SET(x)
parent[x] ← x.
rank[x] ← 0.

UNION(x, y)
r ← FIND(x).
s ← FIND(y).

IF (r = s) RETURN.
ELSE IF (rank[r] > rank[s])
    parent[s] ← r.
ELSE IF (rank[r] < rank[s])
    parent[r] ← s.
ELSE
    parent[r] ← s.
    rank[s] ← rank[s] + 1.

Note. For now, rank = height.

---

Link-by-rank

Link-by-rank. Maintain an integer rank for each node, initially 0. Link root of smaller rank to root of larger rank; if tie, increase rank of new root by 1.

```

```
```
Link-by-rank: properties

**PROPERTY 1.** If \( x \) is not a root node, then \( \text{rank}[x] < \text{rank}[\text{parent}[x]] \).

*Pf.* A node of rank \( k \) is created only by linking two roots of rank \( k - 1 \). •

**PROPERTY 2.** If \( x \) is not a root node, then \( \text{rank}[x] \) will never change again.

*Pf.* Rank changes only for roots; a nonroot never becomes a root. •

**PROPERTY 3.** If \( \text{parent}[x] \) changes, then \( \text{rank}[\text{parent}[x]] \) strictly increases.

*Pf.* The parent can change only for a root, so before linking \( \text{parent}[x] = x \).

After \( x \) is linked-by-rank to new root \( r \) we have \( \text{rank}[r] > \text{rank}[x] \). •

![Diagram](image1.png)

Link-by-rank: analysis

**Theorem.** Using link-by-rank, any UNION or FIND operation takes \( O(\log n) \) time in the worst case, where \( n \) is the number of elements.

*Pf.*

- The running time of UNION and FIND is bounded by the tree height.
- By PROPERTY 5, the height is \( \leq \lfloor \log n \rfloor \). •

![Diagram](image2.png)
Path compression

Path compression. When finding the root \( r \) of the tree containing \( x \), change the parent pointer of all nodes along the path to point directly to \( r \).
Path compression. When finding the root $r$ of the tree containing $x$, change the parent pointer of all nodes along the path to point directly to $r$.

Path compression. When finding the root $r$ of the tree containing $x$, change the parent pointer of all nodes along the path to point directly to $r$.

Path compression. When finding the root $r$ of the tree containing $x$, change the parent pointer of all nodes along the path to point directly to $r$.

Path compression. When finding the root $r$ of the tree containing $x$, change the parent pointer of all nodes along the path to point directly to $r$.

### FIND($x$)

If ($x \neq \text{parent}[x]$)

\[
\text{parent}[x] \leftarrow \text{FIND(parent}[x]).
\]

RETURN \text{parent}[x].

This FIND implementation changes the tree structure (!)

Note. Path compression does not change the rank of a node; so $\text{height}(x) \leq \text{rank}[x]$ but they are not necessarily equal.
Path compression

Fact. Path compression with naïve linking can require $\Omega(n)$ time to perform a single UNION or FIND operation, where $n$ is the number of elements.

Pf. The height of the tree is $n - 1$ after the sequence of union operations: UNION(1, 2), UNION(2, 3), ..., UNION(n – 1, n).

naïve linking: link root of first tree to root of second tree

Theorem. [Tarjan–van Leeuwen 1984] Starting from an empty data structure, path compression with naïve linking performs any intermixed sequence of $m \geq n$ MAKE-SET, UNION, and FIND operations on a set of $n$ elements in $O(m \log n)$ time.

Pf. Nontrivial (but omitted).

Link-by-rank with path compression: properties

PROPERTY. The tree roots, node ranks, and elements within a tree are the same with or without path compression.

Pf. Path compression does not create new roots, change ranks, or move elements from one tree to another.

Link-by-rank with path compression: properties

PROPERTY. The tree roots, node ranks, and elements within a tree are the same with or without path compression.

COROLLARY. PROPERTY 2, 4–6 hold for link-by-rank with path compression.

PROPERTY 1. If $x$ is not a root node, then $\text{rank}[x] < \text{rank}[	ext{parent}[x]]$.
PROPERTY 2. If $x$ is not a root node, then $\text{rank}[x]$ will never change again.
PROPERTY 3. If $\text{parent}[x]$ changes, then $\text{rank}[	ext{parent}[x]]$ strictly increases.
PROPERTY 4. Any root node of rank $k$ has $\geq 2^k$ nodes in its tree.
PROPERTY 5. The highest rank of a node is $\leq \lceil \log n \rceil$.
PROPERTY 6. For any integer $k \geq 0$, there are $\leq n / 2^k$ nodes with rank $k$.

Bottom line. PROPERTY 1–6 hold for link-by-rank with path compression (but we need to recheck PROPERTY 1 and PROPERTY 3).
Link-by-rank with path compression: properties

**Property 3.** If \( \text{parent}[x] \) changes, then \( \text{rank}[\text{parent}[x]] \) strictly increases.

**Pf.** Path compression can make \( x \) point to only an ancestor of \( \text{parent}[x] \).

**Property 1.** If \( x \) is not a root node, then \( \text{rank}[x] < \text{rank}[\text{parent}[x]] \).

**Pf.** Path compression doesn’t change any ranks, but it can change parents. If \( \text{parent}[x] \) doesn’t change during a path compression, the inequality continues to hold; if \( \text{parent}[x] \) changes, then \( \text{rank}[\text{parent}[x]] \) strictly increases.

Iterated logarithm function

**Def.** The iterated logarithm function is defined by:

\[
\lg^* n = \begin{cases} 
0 & \text{if } n \leq 1 \\
1 + \lg^* (\lg n) & \text{otherwise}
\end{cases}
\]

- \( \begin{array}{|c|c|}
| n & \lg^* n | \\
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1 &amp; 0</td>
<td></td>
</tr>
<tr>
<td>2 &amp; 1</td>
<td></td>
</tr>
<tr>
<td>[3, 4] &amp; 2</td>
<td></td>
</tr>
<tr>
<td>[5, 16] &amp; 3</td>
<td></td>
</tr>
<tr>
<td>[17, 65536] &amp; 4</td>
<td></td>
</tr>
<tr>
<td>[65537, 2^{65536}] &amp; 5</td>
<td></td>
</tr>
</tbody>
</table>
\end{array} \)

**Note.** We have \( \lg^* n \leq 5 \) unless \( n \) exceeds the # atoms in the universe.

Creative accounting

**Credits.** A node receives credits as soon as it ceases to be a root. If its rank is in the interval \( (k + 1, k + 2, \ldots, 2^k) \), we give it \( 2^k \) credits.

**Proposition.** Number of credits disbursed to all nodes is \( \leq n \lg^* n \).

**Pf.**

- By **Property 6**, the number of nodes with rank \( \geq k + 1 \) is at most

\[
\frac{n}{2^{k+1}} + \frac{n}{2^{k+2}} + \cdots \leq \frac{n}{2^k}
\]

- Thus, nodes in group \( k \) need at most \( n \) credits in total.
- There are \( \leq \lg^* n \) groups. [**Property 7**]
Running time of \textsc{Find}

Running time of \textsc{Find}. Bounded by number of parent pointers followed.

- Recall: the rank strictly increases as you go up a tree. \textbf{[PROPERTY 1]}
- Case 0: \( \text{parent}[x] \) is a root \( \Rightarrow \) only happens for one link per \textsc{Find}.
- Case 1: \( \text{rank}[	ext{parent}[x]] \) is in a higher group than \( \text{rank}[x] \).
- Case 2: \( \text{rank}[	ext{parent}[x]] \) is in the same group as \( \text{rank}[x] \).

Case 1. At most \( \lg n \) nodes on path can be in a higher group. \textbf{[PROPERTY 7]}

Case 2. These nodes are charged 1 credit to follow parent pointer.

- Each time \( x \) pays 1 credit, \( \text{rank}[	ext{parent}[x]] \) strictly increases. \textbf{[PROPERTY 1]}
- Therefore, if \( \text{rank}[x] \) is in the group \( \{k+1, \ldots, 2^k\} \), the rank of its parent will be in a higher group before \( x \) pays \( 2^k \) credits.
- Once \( \text{rank}[	ext{parent}[x]] \) is in a higher group than \( \text{rank}[x] \), it remains so because:
  - \( \text{rank}[x] \) does not change once it ceases to be a root. \textbf{[PROPERTY 2]}
  - \( \text{rank}[	ext{parent}[x]] \) does not decrease. \textbf{[PROPERTY 3]}
  - thus, \( x \) has enough credits to pay until it becomes a Case 1 node. \( \blacksquare \)

\textbf{Union–Find}

- naive linking
- link-by-size
- link-by-rank
- path compression
- link-by-rank with path compression
- context

Link-by-rank with path compression

\textbf{Theorem.} Starting from an empty data structure, link-by-rank with path compression performs any intermixed sequence of \( m \geq n \) \textsc{Make-Set}, \textsc{Union}, and \textsc{Find} operations on a set of \( n \) elements in \( O(m \log^* n) \) time.

Link-by-size with path compression

\textbf{Theorem.} [Fischer 1972] Starting from an empty data structure, link-by-size with path compression performs any intermixed sequence of \( m \geq n \) \textsc{Make-Set}, \textsc{Union}, and \textsc{Find} operations on a set of \( n \) elements in \( O(m \log \log n) \) time.

\textbf{Context}

\begin{center}
\textit{Introduction}

The equivalence problem is to determine the finest partition on a set that is consistent with a sequence of assertions of the form \( \alpha \equiv \beta \). A strategy for doing this on a computer prevents the assertions merely, maintaining always in storage a representation of the partition refined by the assertions so far encountered. To process the command \( \alpha \equiv \beta \), the equivalence classes of \( \alpha \) and \( \beta \) are determined. If they are the same, nothing further is done; otherwise the two classes are merged together.
\end{center}
**Ackermann function.** [Ackermann 1928] A computable function that is not primitive recursive.

\[
A(m, n) = \begin{cases} 
  n + 1 & \text{if } m = 0 \\
  A(m - 1, 1) & \text{if } m > 0 \text{ and } n = 0 \\
  A(m - 1, A(m, n - 1)) & \text{if } m > 0 \text{ and } n > 0 
\end{cases}
\]

**Inverse Ackermann function.**

\[
\alpha(m, n) = \min\{i \geq 1 : A(i, \lfloor m/n \rfloor) \geq \log_2 n\}
\]

"I am not smart enough to understand this easily."

— Raymond Seidel
Inverse Ackermann function

Definition. \[ \alpha_k(n) = \begin{cases} \lfloor n/2 \rfloor & \text{if } k = 1 \\ 0 & \text{if } n = 1 \text{ and } k \geq 2 \\ 1 + \alpha_k(\alpha_{k-1}(n)) & \text{otherwise} \end{cases} \]

Ex.
- \( \alpha_0(n) = \lfloor n/2 \rfloor \).
- \( \alpha_1(n) = \lfloor \log n \rfloor = \# \text{ of times we divide } n \text{ by } 2 \), until we reach 1.
- \( \alpha_2(n) = \lfloor \log \log n \rfloor = \# \text{ of times we apply the } \log \text{ function to } n \), until we reach 1.
- \( \alpha_4(n) = \# \text{ of times we apply the iterated } \log \text{ function to } n \), until we reach 1.

\[ 2 \uparrow 65536 = 2 \uparrow \uparrow 3 \]

| \( \alpha(0) \) | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 216 | 265536 | 2 \uparrow 65536 |
| \( \alpha(1) \) | 1 | 1 | 2 | 3 | 3 | 4 | 4 | 5 | 5 | 6 | 6 | 7 | 7 | 8 | 8 | 8 | 16 | 65536 | large |
| \( \alpha(2) \) | 0 | 1 | 2 | 3 | 3 | 3 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 16 | 65536 | 2 \uparrow 65536 |
| \( \alpha(3) \) | 0 | 1 | 2 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| \( \alpha(4) \) | 0 | 1 | 2 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |

A tight lower bound

Theorem. [Fredman-Saks 1989] In the worst case, any CELL-PROBE(\log n) algorithm requires \( \Omega(m \alpha(m, n)) \) time to perform an interleaved sequence of \( m \text{ MAKE-SET, UNION, and FIND operations on a set of } n \text{ elements.} \)

Cell-probe model. [Yao 1981] Count only number of words of memory accessed; all other accesses are free.

Path compaction variants

Path splitting. Make every node on path point to its grandparent.
**Path compaction variants**

**Path halving.** Make every other node on path point to its grandparent.

**Disjoint-sets data structures**

**Theorem.** [Tarjan–van Leeuwen 1984] Starting from an empty data structure, link-by-\{size, rank\} combined with \{path compression, path splitting, path halving\} performs any intermixed sequence of $m \geq n$ MAKE-SET, UNION, and FIND operations on a set of $n$ elements in $O(m \alpha(m, n))$ time.