Disjoint sets
A disjoint sets ADT is a structure that stores a set $C$ of disjoint sets. Each set $S \in C$ is identified by a representative element of the set. The structure supports the following operations.

- **MakeSet($x$)**: Create a new set $\{x\}$ and add it to $C$.
- **Union($x, y$)**: Join the sets containing $x$ and $y$ to a new set. The representative of the new set can be any element of the set.
- **FindSet($x$)**: Return a pointer to the representative of the unique set containing $x$.

The arguments to Union and FindSet are pointers to the elements.
Example

1. MakeSet(1) \{1\}
2. MakeSet(2) \{1\} \{2\}
3. MakeSet(3) \{1\} \{2\} \{3\}
4. MakeSet(4) \{1\} \{2\} \{3\} \{4\}
5. Union(1,3) \{1,3\} \{2\} \{4\}
6. FindSet(3) → 1
7. Union(3,4) \{1,3,4\} \{2\}
8. FindSet(3) → 4
9. MakeSet(5) \{1,3,4\} \{2\}\{5\}
10. Union(2,5) \{1,3,4\} \{2,5\}
11. FindSet(3) → 4
**Example — connected components**

**ConnectedComponents** $(V, E)$

1. `foreach v ∈ V`
2. `MakeSet(v)`
3. `foreach (u, v) ∈ E`
4. `if FindSet(u) ≠ FindSet(v)`
5. `Union(u, v)`

**SameComponent** $(u, v)$

1. `if FindSet(u) = FindSet(v) then return TRUE`
2. `else return FALSE`

Disjoint sets

$$\{1\} \{2\} \{3\} \{4\} \{5\} \{6\} \{7\} \{8\} \{9\}$$
Example — connected components

**ConnectedComponents**($V, E$)

(1) **foreach** $v \in V$
(2) **MakeSet**($v$)
(3) **foreach** $(u, v) \in E$
(4) **if** **FindSet**($u$) $\neq$ **FindSet**($v$)
(5) **Union**($u$, $v$)

**SameComponent**($u$, $v$)

(1) **if** **FindSet**($u$) = **FindSet**($v$) **then return** TRUE
(2) **else return** FALSE

Disjoint sets

\[
\begin{align*}
&\{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}, \{7\}, \{8\}, \{9\}
\end{align*}
\]
Example — connected components

**ConnectedComponents**($V$, $E$)

1. **foreach** $v \in V$
2. **MakeSet**($v$)
3. **foreach** $(u, v) \in E$
4. **if** **FindSet**($u$) $\neq$ **FindSet**($v$)
5. **Union**($u$, $v$)

**SameComponent**($u$, $v$)

1. **if** **FindSet**($u$) $=$ **FindSet**($v$) then **return** TRUE
2. **else** **return** FALSE

Disjoint sets

```
{1} {2,4} {3} {5} {6} {7} {8} {9}
```
Example — connected components

**ConnectedComponents** \((V, E)\)

1. **foreach** \(v \in V\)
2. **MakeSet**(\(v\))
3. **foreach** \((u, v) \in E\)
4. **if** **FindSet**(\(u\)) \(\neq\) **FindSet**(\(v\))
5. **Union**(\(u, v\))

**SameComponent**(\(u, v\))

1. **if** **FindSet**(\(u\)) = **FindSet**(\(v\)) then return **TRUE**
2. else return **FALSE**

\[
\begin{align*}
1 & \rightarrow 2 \\
3 & \rightarrow 4 \\
5 & \rightarrow 6 \\
7 & \rightarrow 8 \\
9 &
\end{align*}
\]

\[
\{1\} \{2,4\} \{3\} \{5\} \{6\} \{7\} \{8\} \{9\}
\]
Connected components

\[\text{ConnectedComponents}(V, E)\]

1. \(\text{foreach } v \in V\)
2. \(\text{MakeSet}(v)\)
3. \(\text{foreach } (u, v) \in E\)
4. \(\text{if } \text{FindSet}(u) \neq \text{FindSet}(v)\)
5. \(\text{Union}(u, v)\)

SameComponent \((u, v)\)

1. \(\text{if } \text{FindSet}(u) = \text{FindSet}(v) \text{ then return TRUE}\)
2. \(\text{else return FALSE}\)

Disjoint sets

\[\{1\}\{2,4\}\{3\}\{5,7\}\{6\}\{8\}\{9\}\]
Example — connected components

**ConnectedComponents**($V, E$)

(1) **foreach** $v \in V$
(2) **MakeSet**(v)
(3) **foreach** $(u, v) \in E$
(4) **if** **FindSet**(u) $\neq$ **FindSet**(v)
(5) **Union**(u, v)

**SameComponent**(u, v)

(1) **if** **FindSet**(u) = **FindSet**(v) **then return** TRUE
(2) **else return** FALSE

---

![Disjoint sets diagram]

{1}{2,4}{3}{5,7}{6}{8}{9}
Example — connected components

**CONNECTEDCOMPONENTS**(*V*, *E*)

1. foreach \( v \in V \)
2. **MAKESET**(\( v \))
3. foreach \((u, v) \in E\)
4. if **FINDSET**(\( u \)) \(\neq\) **FINDSET**(\( v \))
5. **UNION**(\( u, v \))

**SAMECOMPONENT**(*u, v*)

1. if **FINDSET**(\( u \)) = **FINDSET**(\( v \)) then return TRUE
2. else return FALSE

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

\{1,3\} \{2,4\} \{5,7\} \{6\} \{8\} \{9\}

Disjoint sets
Example — connected components

**ConnectedComponents**\((V, E)\)

1. **foreach** \(v \in V\)
2. **MakeSet**\((v)\)
3. **foreach** \((u, v) \in E\)
4. **if** **FindSet**\((u) \neq \text{FindSet}(v)\)
5. **Union**\((u, v)\)

**SameComponent**\((u, v)\)

1. **if** **FindSet**\((u) = \text{FindSet}(v)\) **then return** TRUE
2. else return FALSE

\[
\begin{array}{cccc}
1 & 2 & 5 & 6 \\
3 & 4 & 7 & 8 \\
\end{array}
\]

\[
\{1,3\} \{2,4\} \{5,7\} \{6\} \{8\} \{9\}
\]
Example — connected components

**ConnectedComponents**($V, E$)

1. foreach $v \in V$
2. MakeSet($v$)
3. foreach $(u, v) \in E$
4. if FindSet($u$) $\neq$ FindSet($v$)
5. Union($u$, $v$)

**SameComponent**(u, v)

1. if FindSet($u$) = FindSet($v$) then return TRUE
2. else return FALSE

Disjoint sets

{1,3}{2,4}{5,7}{6}{8,9}
Example — connected components

**ConnectedComponents** $(V, E)$

1. foreach $v \in V$
2. `MakeSet(v)`
3. foreach $(u, v) \in E$
4. if `FindSet(u) \neq FindSet(v)`
5. `Union(u, v)`

**SameComponent** $(u, v)$

1. if `FindSet(u) = FindSet(v)` then return `TRUE`
2. else return `FALSE`

Disjoint sets

![Graph with disjoint sets](image)
Example — connected components

**ConnectedComponents**\((V, E)\)

1. `foreach v ∈ V`
2. `MakeSet(v)`
3. `foreach (u, v) ∈ E`
4. `if FindSet(u) ≠ FindSet(v)`
5. `Union(u, v)`

**SameComponent**\((u, v)\)

1. `if FindSet(u) = FindSet(v) then return TRUE`
2. `else return FALSE`

Disjoint sets

\[
\begin{align*}
\{1,2,3,4\} & \{5,7\} \{6\} \{8,9\} \\
\end{align*}
\]
Example — connected components

**CONNECTEDCOMPONENTS**(\(V, E\))

1. \(\text{foreach } v \in V\)
2. \(\text{MAKESET}(v)\)
3. \(\text{foreach } (u, v) \in E\)
4. \(\text{if } \text{FINDSET}(u) \neq \text{FINDSET}(v)\)
5. \(\text{UNION}(u, v)\)

**SAMECOMPONENT**\((u, v)\)

1. \(\text{if } \text{FINDSET}(u) = \text{FINDSET}(v) \text{ then return } \text{TRUE}\)
2. \(\text{else return } \text{FALSE}\)

\begin{align*}
\{1, 2, 3, 4\} & \{5, 7\} \{6\} \{8, 9\} \\
\text{Disjoint sets}
\end{align*}
**Example — connected components**

**CONNECTEDCOMPONENTS**(\(V, E\))

1. foreach \(v \in V\)
2.\hspace{1cm}**MAKESET**(\(v\))
3. foreach \((u, v) \in E\)
4.\hspace{1cm}if **FINDSET**(\(u\)) \(\neq \) **FINDSET**(\(v\))
5.\hspace{1cm}**UNION**(\(u, v\))

**SAMECOMPONENT**(\(u, v\))

1. if **FINDSET**(\(u\)) = **FINDSET**(\(v\)) then return TRUE
2. else return FALSE

\[\{1,2,3,4\}\{5,6,7\}\{8,9\}\]

Disjoint sets
Example — connected components

**CONNECTEDCOMPONENTS**(V, E)

1. foreach \( v \in V \)
2. MakeSet(\( v \))
3. foreach \((u, v) \in E\)
4. if FindSet(\( u \)) \(\neq\) FindSet(\( v \))
5. Union(\( u, v \))

**SAMECOMPONENT**(\( u, v \))

1. if FindSet(\( u \)) = FindSet(\( v \)) then return TRUE
2. else return FALSE

\[
\begin{array}{c}
\{1, 2, 3, 4\} \\
\{5, 6, 7\} \\
\{8, 9\}
\end{array}
\]
Implementation using linked lists

- Each set is stored using a list and a set object.
- The set object contains pointers to the first and last elements in the list.
- Each element in the list contains a pointer to the set object.
- The representative is the first element in the list.

\[
\begin{align*}
\text{head} & \quad 5 & \quad 2 & \quad \text{set} \\
\text{tail} & & & \\
\{2,5\} & \quad \{1,3,4\}
\end{align*}
\]
Implementation using linked lists

MakeSet(\(x\))  Create a new list containing \(x\).  \(\Theta(1)\) time.

FindSet(\(x\))  Return  \(x.set.head\).  \(\Theta(1)\) time.

Union(\(x, y\))  Append the list of \(y\) to the list of \(x\).  Update pointers.
Implementation using linked lists

\textbf{MakeSet}(x) \quad \text{Create a new list containing } x. \Theta(1) \text{ time.}

\textbf{FindSet}(x) \quad \text{Return } x.set.head. \Theta(1) \text{ time.}

\textbf{Union}(x, y) \quad \text{Append the list of } y \text{ to the list of } x. \text{ Update pointers.}

\begin{figure}
\centering
\includegraphics[width=\textwidth]{disjoint_sets.png}
\end{figure}
The time of a Union operation is $\Theta(n)$ in the worst case.

In the worst case, the amortized time is also $\Theta(n)$:

<table>
<thead>
<tr>
<th>Operation</th>
<th># list objects updated</th>
</tr>
</thead>
<tbody>
<tr>
<td>MakeSet(1)</td>
<td>1</td>
</tr>
<tr>
<td>MakeSet(2)</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>:</td>
</tr>
<tr>
<td>MakeSet($n$)</td>
<td>1</td>
</tr>
<tr>
<td>Union(2,1)</td>
<td>2</td>
</tr>
<tr>
<td>Union(3,2)</td>
<td>3</td>
</tr>
<tr>
<td>Union(4,3)</td>
<td>4</td>
</tr>
<tr>
<td>...</td>
<td>:</td>
</tr>
<tr>
<td>Union($n$, $n-1$)</td>
<td>$n$</td>
</tr>
</tbody>
</table>

There are $2n - 1$ operations, and the total time is $\Theta(n^2)$, so the amortized time of an operation is $\Theta(n)$. 

Disjoint sets
Cost of Union

- The time of a Union operation is $\Theta(n)$ in the worst case.
- In the worst case, the amortized time is also $\Theta(n)$:

<table>
<thead>
<tr>
<th>Operation</th>
<th># list objects updated</th>
</tr>
</thead>
<tbody>
<tr>
<td>MakeSet(1)</td>
<td>1</td>
</tr>
<tr>
<td>MakeSet(2)</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>MakeSet($n$)</td>
<td>1</td>
</tr>
<tr>
<td>Union(2,1)</td>
<td>2</td>
</tr>
<tr>
<td>Union(3,2)</td>
<td>3</td>
</tr>
<tr>
<td>Union(4,3)</td>
<td>4</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Union($n$, $n-1$)</td>
<td>$n$</td>
</tr>
</tbody>
</table>

- There are $2n - 1$ operations, and the total time is $\Theta(n^2)$, so the amortized time of an operation is $\Theta(n)$.
Weighted union heuristic

- Append the smaller list to the larger list.
- Worst case time is still $\Theta(n)$.
- The time of $m \leq n - 1$ Union operations is $O(m \log m)$.
- Consider an element $x$.
- After the 1st time the set pointer of $x$ is changed, the size of the list is $\geq 2$.
- After the 2nd time the set pointer of $x$ is changed, the size of the list is $\geq 2 + 2 = 4$.
- After the $k$th time the set pointer of $x$ is changed, the size of the list is $\geq 2^k$.
- The set pointer of $x$ is updated at most $\log(m + 1)$ times.
Weighted union heuristic

- Append the smaller list to the larger list.
- Worst case time is still $\Theta(n)$.
- The time of $m \leq n - 1$ Union operations is $O(m \log m)$.
- Consider an element $x$.
- After the 1st time the set pointer of $x$ is changed, the size of the list is $\geq 2$.
- After the 2nd time the set pointer of $x$ is changed, the size of the list is $\geq 2 + 2 = 4$.
- After the $k$th time the set pointer of $x$ is changed, the size of the list is $\geq 2^k$.
- The set pointer of $x$ is updated at most $\log(m + 1)$ times.
Weighted union heuristic

- Append the smaller list to the larger list.
- Worst case time is still $\Theta(n)$.
- The time of $m \leq n - 1$ Union operations is $O(m \log m)$.
- Consider an element $x$.
- After the 1st time the set pointer of $x$ is changed, the size of the list is $\geq 2$.
- After the 2nd time the set pointer of $x$ is changed, the size of the list is $\geq 2 + 2 = 4$.
- After the $k$th time the set pointer of $x$ is changed, the size of the list is $\geq 2^k$.
- The set pointer of $x$ is updated at most $\log(m + 1)$ times.

\[ \begin{array}{c}
\begin{array}{c}
\includegraphics[width=0.3\textwidth]{disjoint_sets.png}
\end{array}
\end{array} \]
Weighted union heuristic

- Append the smaller list to the larger list.
- Worst case time is still \( \Theta(n) \).
- The time of \( m \leq n - 1 \) Union operations is \( O(m \log m) \).
- Consider an element \( x \).
- After the 1st time the set pointer of \( x \) is changed, the size of the list is \( \geq 2 \).
- After the 2nd time the set pointer of \( x \) is changed, the size of the list is \( \geq 2 + 2 = 4 \).
- After the \( k \)th time the set pointer of \( x \) is changed, the size of the list is \( \geq 2^k \).
- The set pointer of \( x \) is updated at most \( \log(m + 1) \) times.

![Diagram](image-url)
Weighted union heuristic

- Append the smaller list to the larger list.
- Worst case time is still $\Theta(n)$.
- The time of $m \leq n - 1$ Union operations is $O(m \log m)$.
- Consider an element $x$.
- After the 1st time the set pointer of $x$ is changed, the size of the list is $\geq 2$.
- After the 2nd time the set pointer of $x$ is changed, the size of the list is $\geq 2 + 2 = 4$.
- After the $k$th time the set pointer of $x$ is changed, the size of the list is $\geq 2^k$.
- The set pointer of $x$ is updated at most $\log(m + 1)$ times.

$$\geq 4$$

Disjoint sets
Each set is stored in a tree.

Each node in the tree stores a pointer to its parent (it does not store pointers to its children).

The representative of a set is the element in the root of the tree.

For convenience, a root stores a pointer to itself.
Implementation using a forest

**MakeSet**($x$) Create a new tree containing $x$. $\Theta(1)$ time.

**FindSet**($x$) Follow the parent pointers until reaching the root of the tree. Return the root. $\Theta(n)$ time.

**Union**($x$, $y$) Make the root of the tree of $x$ point to the root of the tree of $y$. $\Theta(n)$ time.

![Disjoint sets diagram]
Implementation using a forest

**MakeSet**\( (x) \) Create a new tree containing \( x \). \( \Theta(1) \) time.

**FindSet**\( (x) \) Follow the parent pointers until reaching the root of the tree. Return the root. \( \Theta(n) \) time.

**Union**\( (x, y) \) Make the root of the tree of \( x \) point to the root of the tree of \( y \). \( \Theta(n) \) time.

![Diagram showing the effect of union operation](image-url)

union\( (3,7) \)

Disjoint sets
Union by height

- Each root stores the height of the tree.
- To merge two trees, make the root with smaller height point to the root with larger height.
- A tree with height $h$ contains at least $2^h$ nodes:
  - If we hang a tree $T'$ on a tree $T$, the height of $T$ increases if and only if $\text{height}(T') = \text{height}(T)$.

The height of each tree is $O(\log n)$ and therefore FindSet and Union take $O(\log n)$ time.
Union by height

- Each root stores the height of the tree.
- To merge two trees, make the root with smaller height point to the root with larger height.
- A tree with height $h$ contains at least $2^h$ nodes:
  - If we hang a tree $T'$ on a tree $T$, the height of $T$ increases if and only if $\text{height}(T') = \text{height}(T)$.

- The height of each tree is $O(\log n)$ and therefore FindSet and Union take $O(\log n)$ time.
Union by height

- Each root stores the height of the tree.
- To merge two trees, make the root with smaller height point to the root with larger height.
- A tree with height $h$ contains at least $2^h$ nodes:
  - If we hang a tree $T'$ on a tree $T$, the height of $T$ increases if and only if $\text{height}(T') = \text{height}(T)$.

The height of each tree is $O(\log n)$ and therefore FindSet and Union take $O(\log n)$ time.
Path compression

During FindSet($x$), change the parent pointers of all ancestors of $x$ (except the root) to point to the root.

FindSet(1)
If path compression is used, we cannot maintain the height of the tree in the root.

Instead, we store in each root a rank value, which is an upper bound on the height of the tree.

To merge two trees, make the root with smaller rank point to the root with larger rank.
Pseudo code

**MakeSet**(x)

x.parent ← x
x.rank ← 0

**FindSet**(x)

if x.parent ≠ x
  x.parent ← **FindSet**(x.parent)
return x.parent

**Union**(x, y)

x ← **FindSet**(x)
y ← **FindSet**(y)
if x.rank > y.rank
  y.parent ← x
else
  x.parent ← y
if x.rank = y.rank
  y.rank ← y.rank + 1

Disjoint sets
If both path compression and union by rank are used, the time of a single Union or FindSet is $O(\log n)$.

If both path compression and union by rank are used, the time of $m$ operations is $O(m\alpha(n))$ in the worst case, where $\alpha$ is the inverse Ackermann function.

$\alpha(n)$ grows very slowly.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$\alpha(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0–2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4–7</td>
<td>2</td>
</tr>
<tr>
<td>8–2047</td>
<td>3</td>
</tr>
<tr>
<td>$2048–A_4(1)$</td>
<td>4</td>
</tr>
</tbody>
</table>

$A_4(1) \gg 2^{2048}$