Today

- A data structure for a new problem
  - “Union-Find”
- Amortized analysis
Dynamic Maintenance of Sets

• Assume, we have a collection of elements
• The elements are being clustered
• Initially, each element forms its own cluster/set
• We want to enable two operations:
  – FIND-SET(\(x\)): report the cluster containing \(x\)
  – UNION(\(C_1, C_2\)): merges the clusters \(C_1, C_2\)
Disjoint-set data structure (Union-Find)

Problem:

• Maintain a collection of *pairwise-disjoint* sets \( S = \{ S_1, S_2, \ldots, S_r \} \).
• Each \( S_i \) contains one “representative” element \( x = \text{rep}[S_i] \).
• Must support three operations:
  - **MAKE-SET(\( x \))**: adds new set \( \{ x \} \) to \( S \) with \( \text{rep}[\{ x \}] = x \) (for any \( x \not\in S_i \) for all \( i \)).
  - **UNION(\( x, y \))**: replaces sets \( S_x, S_y \) with \( S_x \cup S_y \) in \( S \) for any \( \text{rep}.x, y \) in distinct sets \( S_x, S_y \).
  - **FIND-SET(\( x \))**: returns representative \( \text{rep}[S_x] \) of set \( S_x \) containing element \( x \).
Quiz

• If we have a $\text{WeakUnion}(x, y)$ that works only if $x, y$ are representatives, how can we implement $\text{Union}$ that works for any $x, y$?

• $\text{Union}(x, y)$

  $= \text{WeakUnion}( \text{Find-Set}(x), \text{Find-Set}(y) )$
Applications

- Data clustering
- Killer App: Minimum Spanning Tree
- Amortized analysis
Ideas?

• How can we implement this data structure efficiently?
  – MAKE-SET
  – UNION
  – FIND-SET
Bad case for UNION or FIND
Simple linked-list solution

Store set $S_i = \{x_1, x_2, \ldots, x_k\}$ as an (unordered) doubly linked list. Define representative element $rep[S_i]$ to be the front of the list, $x_1$.

$S_i:$

![Diagram of linked list](image)

- **MAKE-SET($x$)** initializes $x$ as a lone node.
- **FIND-SET($x$)** walks left in the list containing $x$ until it reaches the front of the list.
- **UNION($x, y$)** concatenates the lists containing $x$ and $y$, leaving rep. as **FIND-SET**[$x$].
Store set $S_i = \{x_1, x_2, \ldots, x_k\}$ as an (unordered) doubly linked list. Define representative element $\text{rep}[S_i]$ to be the front of the list, $x_1$.

- $\text{MAKE-SET}(x)$ initializes $x$ as a lone node. $\Theta(1)$
- $\text{FIND-SET}(x)$ walks left in the list containing $x$ until it reaches the front of the list. $\Theta(n)$
- $\text{UNION}(x, y)$ concatenates the lists containing $x$ and $y$, leaving rep. as $\text{FIND-SET}[x]$. $\Theta(n)$
Augmented linked-list solution

Store set \( S_i = \{x_1, x_2, \ldots, x_k\} \) as unordered doubly linked list. Each \( x_j \) also stores pointer \( rep[x_j] \) to head.

\[
\begin{align*}
S_i : & \quad x_1 \quad x_2 \quad \cdots \quad x_k \\
rep[S_i] & \\
\end{align*}
\]

- **FIND-SET(\( x \))** returns \( rep[x] \).
- **UNION(\( x, y \))** concatenates the lists containing \( x \) and \( y \), and updates the \( rep \) pointers for all elements in the list containing \( y \).
Example of augmented linked-list solution

$S_x : \quad \text{rep} \quad x_1 \quad \text{rep}[S_x] \quad x_2$

$S_y : \quad \text{rep}[S_y] \quad y_1 \quad \text{rep} \quad y_2 \quad \text{rep} \quad y_3$
Example of augmented linked-list solution

\[ S_x \cup S_y : \]

\[ \text{rep}[S_x] \]

\[ \text{rep} \]

\[ \text{rep}[S_y] \]
Example of augmented linked-list solution

$S_x \cup S_y :$

$rep[S_x \cup S_y]$
Store set $S_i = \{x_1, x_2, \ldots, x_k\}$ as unordered doubly linked list. Each $x_j$ also stores pointer $rep[x_j]$ to head.

- **FIND-SET(x)** returns $rep[x]$. $\Theta(1)$
- **UNION(x, y)** concatenates the lists containing $x$ and $y$, and updates the $rep$ pointers for all elements in the list containing $y$. $\Theta(n)$
Amortized analysis

• So far, we focused on worst-case time of each operation.
  – E.g., UNION takes $\Theta(n)$ time for some operations
• Amortized analysis: count the total time spent by any sequence of operations
• Total time is always at most
  
  worst-case-time-per-operation * #operations
  
  but it can be much better!
  – E.g., if times are 1,1,1,…,1,n,1,…,1
• Can we modify the linked-list data structure so that any sequence of $m$ MAKE-SET, FIND-SET, UNION operations cost less than $m*\Theta(n)$ time?
Alternative

\textbf{UNION}(x, y) :

- concatenates the lists containing \(y\) and \(x\), and
- update the \textit{rep} pointers for all elements in the list containing \(y\) \(x\)
Alternative concatenation

\textsc{Union}(x, y) could instead

- concatenate the lists containing \textit{y} and \textit{x}, and
- update the \textit{rep} pointers for all elements in the list containing \textit{x}.

\[ S_x \cup S_y : \]

\[ rep[S_y] \]

\[ rep[S_x] \]
Alternative concatenation

$\text{UNION}(x, y)$ could instead

- concatenate the lists containing $y$ and $x$, and
- update the $\text{rep}$ pointers for all elements in the list containing $x$.

$S_x \cup S_y$:

$\text{rep}[S_x \cup S_y]$
Smaller into larger

- Concatenate smaller list onto the end of the larger list (each list stores its weight = # elements)

- Cost = $\Theta$(length of smaller list).

Let $n$ denote the overall number of elements (equivalently, the number of MAKE-SET operations). Let $m$ denote the total number of operations.

**Theorem:** Cost of all UNION’s is $O(n \lg n)$.

**Corollary:** Total cost is $O(m + n \lg n)$. 
Total UNION cost is $O(n \lg n)$

Proof:

- Monitor an element $x$ and set $S_x$ containing it
- After initial MAKE-SET($x$), $\text{weight}[S_x] = 1$
- Consider any time when $S_x$ is merged with set $S_y$
  - If $\text{weight}[S_y] \geq \text{weight}[S_x]$
    - pay 1 to update $\text{rep}[x]$
    - $\text{weight}[S_x]$ at least doubles (increasing by $\text{weight}[S_y]$)
  - Otherwise
    - pay nothing
    - $\text{weight}[S_x]$ only increases
- Thus:
  - Each time we pay 1, the weight doubles
  - Maximum possible weight is $n$
  - Maximum pay $\leq \lg n$ for $x$, or $O(n \log n)$ overall
Final Result

• We have a data structure for dynamic sets which supports:
  – **MAKE-SET**: $O(1)$ worst case
  – **FIND-SET**: $O(1)$ worst case
  – **UNION**:
    • Any sequence of $m$ operations* takes $O(m \log n)$ time, or
    • … the *amortized complexity* of the operations* is $O(\log n)$

* **MAKE-SET, FIND-SET** or **UNION**
Can we do better?

One can do:

- **MAKE-SET**: $O(1)$ worst case
- **FIND-SET**: $O(lg\ n)$ worst case
- **WEAK-UNION**: $O(1)$ worst case
- Thus, **UNION**: $O(lg\ n)$ worst case
Representing sets as trees

• Each set $S_i = \{x_1, x_2, \ldots, x_k\}$ stored as a tree
• $\text{rep}[S_i]$ is the tree root.

$S_1 = \{x_1, x_2, x_3, x_4, x_5, x_6\}$
$S_2 = \{x_7\}$

$\text{MAKE-SET}(x)$ initializes $x$ as a lone node.

$\text{FIND-SET}(x)$ walks up the tree containing $x$ until it reaches the root.

$\text{UNION}(x, y)$ concatenates the trees containing $x$ and $y$
Time?

- **MAKE-SET(x)** initializes $x$ as a lone node. $O(1)$
- **FIND-SET(x)** walks up the tree containing $x$ until it reaches the root. $O(\text{height}) = ?$
- **WEAK-UNION(x, y)** concatenates the trees containing $x$ and $y$. $O(1)$
**Trick 1: “Smaller into Larger”** (for trees)

**Algorithm:** Merge tree with smaller weight into tree with larger weight.

- Height of tree increases only when its weight doubles
- Height logarithmic in weight
“Smaller into Larger” in trees

Proof:

• Monitor the height of an element $z$
• Each time the height of $z$ increases, the weight of its tree doubles
• Maximum weight is $n$
• Thus, height of $z$ is $\leq \log n$
Tree implementation

- We have:
  - \texttt{MAKE-SET}: $O(1)$ worst case
  - \texttt{FIND-SET}: $O(\text{height}) = O(\lg n)$ worst case
  - \texttt{WEAK-UNION}: $O(1)$ worst case
- Can amortized analysis buy us anything?
- Need another trick…
**Trick 2: Path compression**

When we execute a `FIND-SET` operation and walk up a path to the root, we *know* the representative for *all* the nodes on the path.

*Path compression* makes all of those nodes direct children of the root.
Trick 2: Path compression

When we execute a FIND-SET operation and walk up a path to the root, we know the representative for all the nodes on the path.

Path compression makes all of those nodes direct children of the root.
**Trick 2: Path compression**

When we execute a `FIND-SET` operation and walk up a path \( p \) to the root, we know the representative for all the nodes on path \( p \).

*Path compression* makes all of those nodes direct children of the root.

Cost of `FIND-SET(x)` is still \( \Theta(\text{depth}[x]) \).
The Theorem

**Theorem:** In general, amortized cost is $O(\alpha(n))$, where $\alpha(n)$ grows really, really slow.

**Proof:** Really, really long (CLRS, p. 509)