- Have $n \gg 0$ items and need to extract them from smallest to largest

- Have $n \gg 0$ items and need to extract them from smallest to largest, but sometimes need to add new elements in

- Have $n = 10$ items and need to extract them from smallest to largest, but sometimes need to add new elements in
Overview

⇒ Disjoint Set Operations
• Linked-List Representation
• Disjoint Forests
Disjoint Set

• A bunch of elements
  - Each can be only one set ⇒ disjoint sets
• Dynamic set
  - New elements added to set
• Identify a set by an element in the set: representative
  - Doesn’t matter which element it is
  - Needs to be same element while set is not modified
Operations

• Make-Set(x)
  - Creates a new set whose only member is x
  - $x$ can not already be in some other set

• Union(x,y)
  - Unites the sets represented by x and y
  - Assume set are disjoint beforehand

• Find-Set(x)
  - Returns a pointer to the representative of the set containing x

• No other operations
  - No delete

*What can it be used for?
Analysis

• Analyze performance via:
  - $n$ number of Make-Set operations
  - $m$ number of Make-Set, Union, and Find-Set operations

• Each union reduces number of sets by 1
  - Can be at most $n - 1$ unions
  - Assume $n$ Make-set operations occur first
Application of Disjoint Sets

- Connected components in a Graph

![Graph with labeled nodes and connecting edges](image)

<table>
<thead>
<tr>
<th>Edge processed</th>
<th>Collection of disjoint sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial sets</td>
<td>{a} {b} {c} {d} {e} {f} {g} {h} {i} {j}</td>
</tr>
<tr>
<td>(b,d)</td>
<td>{a} {b,d} {c} {e} {f} {g} {h} {i} {j}</td>
</tr>
<tr>
<td>(e,g)</td>
<td>{a} {b,d} {c} {e,g} {f} {h} {i} {j}</td>
</tr>
<tr>
<td>(a,c)</td>
<td>{a,c} {b,d} {e,g} {f} {h} {i} {j}</td>
</tr>
<tr>
<td>(h,i)</td>
<td>{a,c} {b,d} {e,g} {f} {h,i} {j}</td>
</tr>
<tr>
<td>(a,b)</td>
<td>{a,b,c,d} {e,g} {f} {h,i} {j}</td>
</tr>
<tr>
<td>(e,f)</td>
<td>{a,b,c,d} {e,f,g} {h,i} {j}</td>
</tr>
<tr>
<td>(b,c)</td>
<td>{a,b,c,d} {e,f,g} {h,i} {j}</td>
</tr>
</tbody>
</table>

**Figure 21.1** (a) A graph with four connected components: \{a, b, c, d\}, \{e, f, g\}, \{h, i\}, and \{j\}. (b) The collection of disjoint sets after processing each edge.
CONNECTED-COMPONENTS \((G)\)

1. \textbf{for} each vertex \(v \in G.V\)
2. \hspace{1em} \textbf{MAKE-SET}(v)
3. \textbf{for} each edge \((u, v) \in G.E\)
4. \hspace{1em} \textbf{if} \textbf{FIND-SET}(u) \neq \textbf{FIND-SET}(v)
5. \hspace{2em} \textbf{UNION}(u, v)

SAME-COMPONENT \((u, v)\)

1. \textbf{if} \textbf{FIND-SET}(u) == \textbf{FIND-SET}(v)
2. \hspace{1em} \textbf{return} TRUE
3. \hspace{2em} \textbf{else return} FALSE
Proof

Question 21.1-2
After all edges are processed by Connected-Components, show that two vertices are in the same connected component if and only if they are in the same set.

Proof:
\( x \) and \( y \) in same component \( \Rightarrow \) \( x \) and \( y \) in same set

\( x \) and \( y \) in same set \( \Rightarrow \) \( x \) and \( y \) in same component
Overview

• Disjoint Set Operations
  ⇒ Linked-List Representation
• Disjoint Forests
Figure 21.2  (a) Linked-list representations of two sets. Set $S_1$ contains members $d$, $f$, and $g$, with representative $f$, and set $S_2$ contains members $b$, $c$, $e$, and $h$, with representative $c$. Each object in the list contains a set member, a pointer to the next object in the list, and a pointer back to the set object. Each set object has pointers head and tail to the first and last objects, respectively. (b) The result of UNION($g$, $e$), which appends the linked list containing $e$ to the linked list containing $g$. The representative of the resulting set is $f$. The set object for $e$’s list, $S_2$, is destroyed.
## Time Analysis

- **Make-Set(x) and Find-Set(x)**
  - $O(1)$ time

- **Union**
  - $O(1)$ to find end of 1st list and point it to start of 2nd list
  - But must change children of second list to point to first list
  - Worst case: join bigger to smaller
  - Worst case running time?

- Best case: join bigger to smaller
- Best case running time?

<table>
<thead>
<tr>
<th>Operation</th>
<th>Number of objects updated</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAKE-SET($x_1$)</td>
<td>1</td>
</tr>
<tr>
<td>MAKE-SET($x_2$)</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>MAKE-SET($x_n$)</td>
<td>1</td>
</tr>
<tr>
<td>UNION($x_2$, $x_1$)</td>
<td>1</td>
</tr>
<tr>
<td>UNION($x_3$, $x_2$)</td>
<td>2</td>
</tr>
<tr>
<td>UNION($x_4$, $x_3$)</td>
<td>3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>UNION($x_n$, $x_{n-1}$)</td>
<td>$n - 1$</td>
</tr>
</tbody>
</table>
Weighted-Union Heuristic

• Worst case since appending a longer list onto a shorter list
• Improvement:
  - Check which list is shorter, append it onto longer list
  - **Weighted-Union Heuristic**
  - How do we know which is shorter in $O(1)$ time?

**Theorem 21.1**
Using linked-list and weighted-union heuristic, a sequence of $m$ Make-set, Union, and Find-Set, $n$ of which are Make-set, will take $O(m + n \lg n)$

- What is maximum number of unions?
- What is running time of non-unions?
- Just need to prove that $n - 1$ unions take $O(n \lg n)$
Proof

Running time is same as number of times an element’s pointer is changed to point to another set. Each time it is changed, put into a set at least twice as large. So most changes are: \( \lg n \)
Overview

• Disjoint Set Operations
• Linked-List Representation

⇒ Disjoint Forests
Faster Implementation: Disjoint-set Forests

• Use rooted trees (Chapter 11)
  - Each node contains a member, each tree represents a set
  - Forest for the sets
  - Just use parent pointers

• Running time?
Heuristics to Improve Running Time

• Union by rank
  - Similar to weighted-union heuristic for linked lists
  - Make root of tree with fewer nodes the child node
  - How can we keep track of number of nodes without adding extra time?
    + Rank: upper bound on height of node (max number of edges to get to a leaf)
  - When unioning, just root’s rank might need to be updated
Heuristics

• Path Compression
  - During find-set’s traversal to root of tree, make each node point to root
  - Don’t update rank: rank is an upper bound anyways, and no way to do it
**Code**

- Where are two heuristics?  
  + Union by Rank  
  + Path Compression

- When is rank changed?  
  + When might rank just be upper bound?

- How does Find-Set work?  
  + Tail-end recursion?

```plaintext
MAKE-SET(x)
1 x.p = x
2 x.rank = 0

UNION(x, y)
1 LINK(FIND-SET(x), FIND-SET(y))

LINK(x, y)
1 if x.rank > y.rank
2     y.p = x
3 else x.p = y
4     if x.rank == y.rank
5         y.rank = y.rank + 1

The FIND-SET procedure with path compression is quite simple:

FIND-SET(x)
1 if x != x.p
2     x.p = FIND-SET(x.p)
3 return x.p
```

© P. Heeman, 2017