Announcements

- Reading for this lecture: Chapter 8.
- Friday’s topic, Minimum Spanning Trees
- Wednesday / Thursday, NP Completeness
Disjoint Set ADT

• Data: set of pairwise **disjoint sets**.

• Required operations
  – **Union** – merge two sets to create their union
  – **Find** – determine which set an item appears in
Disjoint Sets and Naming

• Maintain a set of pairwise disjoint sets.
  – \{3,5,7\} , \{4,2,8\}, \{9\}, \{1,6\}

• Each set has a unique name: one of its members (for convenience)
  – \{3,5,7\} , \{4,2,8\}, \{9\}, \{1,6\}
Union / Find

• Union(x,y) – take the union of two sets named x and y
  – \{3,5,7\}, \{4,2,8\}, \{9\}, \{1,6\}
  – Union(5,1)
    \{3,5,7,1,6\}, \{4,2,8\}, \{9\},
• Find(x) – return the name of the set containing x.
  – \{3,5,7,1,6\}, \{4,2,8\}, \{9\},
  – Find(1) = 5
  – Find(4) = 8
Union/Find Trade-off

• Known result:
  – Find and Union cannot both be done in worst-case $O(1)$ time with any data structure.

• We will instead aim for good amortized complexity.

• For $m$ operations on $n$ elements:
  – Target complexity: $O(m)$ i.e. $O(1)$ amortized
Up-Tree for DS Union/Find

**Observation:** we will only traverse these trees upward from any given node to find the root.

**Idea:** reverse the pointers (make them point up from child to parent). The result is an **up-tree**.

Initial state

Intermediate state

Roots are the names of each set.
Operations

Find(x) follow x to the root and return the root.

Union(i, j) - assuming i and j roots, point j to i.
Simple Implementation

• Array of indices

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

`up[x] = -1` means `x` is a root.
A Bad Case

1 2 3 \cdots n

Union(1,2)

Union(2,3)

\vdots

Union(n-1,n)

Find(1) \ n \text{ steps!!}
Amortized Cost

• Cost of \( n \) Union operations followed by \( n \) Find operations is \( n^2 \)
• \( \Theta(n) \) per operation
Two Big Improvements

Can we do better? Yes!

1. **Union-by-size**
   - Improve Union so that Find only takes worst case time of $\Theta(\log n)$.

2. **Path compression**
   - Improve Find so that, with Union-by-size, Find takes amortized time of almost $\Theta(1)$. 
Union-by-Size

Union-by-size
– Always point the smaller tree to the root of the larger tree

S-Union(7,1)
Example Again

\[ S \cup \{1,2\} \]
\[ S \cup \{2,3\} \]
\[ \vdots \]
\[ S \cup \{n-1,n\} \]

\[ \text{Find}(1) \text{ constant time} \]
Analysis of Union-by-Size

• Theorem: With union-by-size an up-tree of height $h$ has size at least $2^h$.
• Proof by induction
  – Base case: $h = 0$. The up-tree has one node, $2^0 = 1$
  – Inductive hypothesis: Assume true for $h-1$
  – Observation: tree gets taller only as a result of a union.
Analysis of Union-by-Size

• What is worst case complexity of Find(x) in an up-tree forest of \( n \) nodes?

• (Amortized complexity is no better.)
Worst Case for Union-by-Size

n/2 Unions-by-size

n/4 Unions-by-size
Example of Worst Cast (cont’)

After $n-1 = n/2 + n/4 + \ldots + 1$ Unions-by-size

If there are $n = 2^k$ nodes then the longest path from leaf to root has length $k$. 
Array Implementation

Can store separate size array:

<table>
<thead>
<tr>
<th>up size</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-1</td>
<td>1</td>
<td>-1</td>
<td>7</td>
<td>7</td>
<td>5</td>
<td>-1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>
Better, store sizes in the up array:

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

Negative up-values correspond to sizes of roots.
Code for Union-by-Size

S-Union(i,j) {
  // Collect sizes
  si = -up[i];
  sj = -up[j];

  // verify i and j are roots
  assert(si >= 0 && sj >= 0)
  // point smaller sized tree to
  // root of larger, update size
  if (si < sj) {
    up[i] = j;
    up[j] = -(si + sj);
  } else {
    up[j] = i;
    up[i] = -(si + sj);
  }
}
Path Compression

- To improve the amortized complexity, we’ll borrow an idea from splay trees:
  - When going up the tree, *improve nodes on the path*!
- On a Find operation point all the nodes on the search path directly to the root. This is called “path compression.”

---

```
PC-Find(3)
```
Self-Adjustment Works

PC-Find(x)
Draw the result of Find(5):
Code for Path Compression Find

```c
PC-Find(i) {
    //find root
    j = i;
    while (up[j] >= 0) {
        j = up[j];
        root = j;
    }
    //compress path
    if (i != root) {
        parent = up[i];
        while (parent != root) {
            up[i] = root;
            i = parent;
            parent = up[parent];
        }
    }
    return(root);
}
```
Complexity of Union-by-Size + Path Compression

• Worst case time complexity for…
  – …a single Union-by-size is:
  – …a single PC-Find is:

• Time complexity for $m \geq n$ operations on $n$ elements has been shown to be $O(m \log^* n)$. [See Weiss for proof.]
  – Amortized complexity is then $O(\log^* n)$
  – What is $\log^*$ ?
log* $n$

$log^* n = \text{number of times you need to apply }
\log \text{ to bring value down to at most } 1$

$log^* 2 = 1$
$log^* 4 = log^* 2^2 = 2$
$log^* 16 = log^* 2^{2^2} = 3 \quad \text{(log log log 16 = 1)}$
$log^* 65536 = log^* 2^{2^{2^2}} = 4 \quad \text{(log log log log 65536 = 1)}$
$log^* 2^{65536} = \ldots \ldots \approx \log^* (2 \times 10^{19,728}) = 5$

$log^* n \leq 5 \text{ for all reasonable } n$. 
The Tight Bound

In fact, Tarjan showed the time complexity for \( m \geq n \) operations on \( n \) elements is:

\[
\Theta(m \alpha(m, n))
\]

Amortized complexity is then \( \Theta(\alpha(m, n)) \).

What is \( \alpha(m, n) \)?

- Inverse of Ackermann’s function.

- For reasonable values of \( m, n \), grows even slower than \( \log^* n \). So, it’s even “more constant.”

Proof is beyond scope of this class. A simple algorithm can lead to incredibly hardcore analysis!