Practice

Given the following disjoint-set what would be the result of the following calls on union if we add the “union-by-weight” optimization. Draw the forest at each stage with corresponding ranks for each tree.

union(2, 13)
union(4, 12)
union(2, 8)
Given the following disjoint-set what would be the result of the following calls on union if we add the “union-by-weight” optimization. Draw the forest at each stage with corresponding ranks for each tree.

union(2, 13)
Practice

Given the following disjoint-set what would be the result of the following calls on union if we add the “union-by-weight” optimization. Draw the forest at each stage with corresponding ranks for each tree.

union(2, 13)
union(4, 12)
Given the following disjoint-set what would be the result of the following calls on union if we add the “union-by-weight” optimization. Draw the forest at each stage with corresponding ranks for each tree.

union(2, 13)
union(4, 12)
union(2, 8)
Practice

Given the following disjoint-set what would be the result of the following calls on union if we add the “union-by-weight” optimization. Draw the forest at each stage with corresponding ranks for each tree.

union(2, 13)
union(4, 12)
union(2, 8)

Does this improve the worst case runtimes?

findSet is more likely to be O(log(n)) than O(n)
Midpoint survey

Thank you all so much for filling out the lecture and section midpoint surveys! We appreciate the feedback and are working on incorporating it :)

90.4% MIDPOINT SURVEY COMPLETION RATE

GOT THE EXTRA CREDIT
Announcements

P3 due Today

P4 comes out today due in 3 weeks on Wednesday June 2^{nd}
- last project!
- ~2 weeks of work
- extra credit spec quiz on gradescope!

E3 came out on Friday – due this Friday May 14^{th}
- two more exercises coming

Upcoming meme competition

Reminders:
- Tons of extra practice on section hand outs
- section slides and videos are also available
- always always feel free to reach out to Tas 😊
# New ADT

## Set ADT

### state
- Set of elements
  - Elements must be unique!
  - No required order

### Count of Elements

### behavior
- **create(x)** - creates a new set with a single member, x
- **add(x)** - adds x into set if it is unique, otherwise add is ignored
- **remove(x)** – removes x from set
- **size()** – returns current number of elements in set

## Disjoint-Set ADT

### state
- Set of Sets
  - **Disjoint**: Elements must be unique across sets
  - No required order
  - Each set has representative

### Count of Sets

### behavior
- **makeSet(x)** – creates a new set within the disjoint set where the only member is x. Picks representative for set
- **findSet(x)** – looks up the set containing element x, returns representative of that set
- **union(x, y)** – looks up set containing x and set containing y, combines two sets into one. Picks new representative for resulting set
Implementation

Disjoint-Set ADT

**state**
- Set of Sets
  - Disjoint: Elements must be unique across sets
  - No required order
  - Each set has representative

**behavior**
- `makeSet(x)` – creates a new set within the disjoint set where the only member is x. Picks representative for set
- `findSet(x)` – looks up the set containing element x, returns representative of that set
- `union(x, y)` – looks up set containing x and set containing y, combines two sets into one. Picks new representative for resulting set

TreeDisjointSet<E>

**state**
- `Collection<TreeSet> forest`
- `Dictionary<NodeValues, NodeLocations> nodeInventory`

**behavior**
- `makeSet(x)` – create a new tree of size 1 and add to our forest
- `findSet(x)` – locates node with x and moves up tree to find root
- `union(x, y)` – append tree with y as a child of tree with x

TreeSet<E>

**state**
- `SetNode overallRoot`

**behavior**
- `TreeSet(x)`
- `add(x)`
- `remove(x, y)`
- `getRep()` – returns data of overallRoot

SetNode<E>

**state**
- `E data`
- `Collection<SetNode> children`

**behavior**
- `SetNode(x)`
- `addChild(x)`
- `removeChild(x, y)`
Review QuickFind vs. QuickUnion

**DISJOINT SETS ADT**

**QuickFind**
- map from value to representative ID
- Joyce, Sam, Ken, Alex
- Aileen, Santino
- Paul

**QuickUnion**
- trees of values with representative ID at each root
- Joyce (2)
- Sam
- Ken
- Santino
- Aileen (1)
- Paul (3)

Could also use one element from each set (e.g. the root) as its representative: only uniqueness matters

<table>
<thead>
<tr>
<th></th>
<th>(Baseline)</th>
<th>QuickFind</th>
<th>QuickUnion</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>makeSet(value)</code></td>
<td>(\Theta(1))</td>
<td>(\Theta(1))</td>
<td>(\Theta(1))</td>
</tr>
<tr>
<td><code>find(value)</code></td>
<td>(\Theta(n))</td>
<td>(\Theta(1))</td>
<td>(\Theta(n))</td>
</tr>
<tr>
<td><code>union(x, y)</code></td>
<td>(\Theta(n))</td>
<td>(\Theta(n))</td>
<td>(\Theta(1))</td>
</tr>
</tbody>
</table>
Review: QuickUnion: Why Use Both Roots?

Example: result of union(Ken, Santino) on these Disjoint Sets given three possible implementations:

**Correct**: Everything in Ken’s set now connected to everything in Santino’s set!

**Incorrect**: Ken and Joyce were connected before; the union operation shouldn’t remove connections.

**Inefficient**: Technically correct, but increases height of the up-tree so makes
**Goal:** Always pick the smaller tree to go under the larger tree

**Implementation:** Store the number of nodes (or “weight”) of each tree in the root
- Constant-time lookup instead of having to traverse the entire tree to count

**union(A, B):**
- `rootA = find(A)`
- `rootB = find(B)`
- put lighter root under heavier root

**union(A, B)**
**union(B, C)**
**union(C, D)**
**find(A)**

Now what happens?

```
Perfect! Best runtime we can get.
```
**Review** WeightedQuickUnion: Performance

Consider the worst case where the tree height grows as fast as possible

<table>
<thead>
<tr>
<th>N</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Consider the worst case where the tree height grows as fast as possible.

<table>
<thead>
<tr>
<th>N</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
Consider the worst case where the tree height grows as fast as possible:

<table>
<thead>
<tr>
<th>N</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>?</td>
</tr>
</tbody>
</table>
Review WeightedQuickUnion: Performance

Consider the worst case where the tree height grows as fast as possible

<table>
<thead>
<tr>
<th>N</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>
Review WeightedQuickUnion: Performance

Consider the worst case where the tree height grows as fast as possible

<table>
<thead>
<tr>
<th>N</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>?</td>
</tr>
</tbody>
</table>
Consider the worst case where the tree height grows as fast as possible.
WeightedQuickUnion: Performance

- Consider the worst case where the tree height grows as fast as possible
- Worst case tree height is $\Theta(\log N)$
Review Why Weights Instead of Heights?

We used the number of items in a tree to decide upon the root

Why not use the height of the tree?
- HeightedQuickUnion’s runtime is asymptotically the same: $\Theta(\log(n))$
- It’s easier to track weights than heights, even though WeightedQuickUnion can lead to some suboptimal structures like this one:
This is pretty good! But there’s one final optimization we can make: **path compression**

<table>
<thead>
<tr>
<th>Operation</th>
<th>(Baseline)</th>
<th>QuickFind</th>
<th>QuickUnion</th>
<th>WeightedQuickUnion</th>
</tr>
</thead>
<tbody>
<tr>
<td>makeSet(value)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
</tr>
<tr>
<td>find(value)</td>
<td>Θ(n)</td>
<td>Θ(1)</td>
<td>Θ(n)</td>
<td>Θ(log n)</td>
</tr>
<tr>
<td>union(x, y)</td>
<td>Θ(n)</td>
<td>Θ(n)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
</tr>
<tr>
<td>union(x, y) assuming root args</td>
<td>Θ(n)</td>
<td>Θ(n)</td>
<td>Θ(n)</td>
<td>Θ(log n)</td>
</tr>
</tbody>
</table>
Thus far, the modifications we’ve studied are designed to *preserve invariants*
- E.g. Performing rotations to preserve the AVL invariant
- We rely on those invariants always being true so every call is fast

Path compression is entirely different: we are modifying the tree structure to *improve future performance*
- Not adhering to a specific invariant
- The first call may be slow, but will optimize so future calls can be fast
Path Compression: Idea

This is the worst-case topology if we use WeightedQuickUnion

Idea: When we do find(15), move all visited nodes under the root

- Additional cost is insignificant (we already have to visit those nodes, just constant time work to point to root too)
Path Compression: Idea

This is the worst-case topology if we use WeightedQuickUnion

Idea: When we do find(15), move all *visited nodes* under the root
- Additional cost is insignificant (we already have to visit those nodes, just constant time work to point to root too)

• Perform Path Compression on every find(), so future calls to find() are faster!
Path Compression: Details and Runtime

Run path compression on every find()!
- Including the find()s that are invoked as part of a union()

Understanding the performance of many operations requires **amortized analysis**
- Effectively averaging out rare events over many common ones
- Typically used for “In-Practice” case
  - E.g. when we assume an array doesn’t resize “in practice”, we can do that because the rare resizing calls are amortized over many faster calls
- In 373 we don’t go in-depth on amortized analysis
Path Compression: Runtime

M find()s on WeightedQuickUnion requires takes $\Theta(M \log N)$

... but M find()s on WeightedQuickUnionWithPathCompression takes $O(M \log^* N)!$

- $\log^* n$ is the "iterated log": the number of times you need to apply log to $n$ before it’s $\leq 1$
- Note: $\log^*$ is a loose bound
Path Compression: Runtime

Path compression results in find()s and union()s that are very very close to (amortized) constant time

- $\log^*$ is less than 5 for any realistic input
- If $M$ find()s/union()s on $N$ nodes is $O(M \log^* N)$ and $\log^* N \approx 5$, then find()\slash union()s amortizes to $O(1)!$

<table>
<thead>
<tr>
<th>$N$</th>
<th>$\log^* N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>16</td>
<td>3</td>
</tr>
<tr>
<td>65536</td>
<td>4</td>
</tr>
<tr>
<td>$2^{65536}$</td>
<td>5</td>
</tr>
</tbody>
</table>

Number of atoms in the known universe is $2^{256}$ish
WQU + Path Compression Runtime

In-Practice Runtimes:

<table>
<thead>
<tr>
<th>Operation</th>
<th>(Baseline)</th>
<th>QuickFind</th>
<th>QuickUnion</th>
<th>WeightedQuickUnion</th>
<th>WQU + Path Compression</th>
</tr>
</thead>
<tbody>
<tr>
<td>makeSet(value)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
</tr>
<tr>
<td>find(value)</td>
<td>Θ(n)</td>
<td>Θ(1)</td>
<td>Θ(n)</td>
<td>Θ(log n)</td>
<td>O(log* n)</td>
</tr>
<tr>
<td>union(x, y) assuming root args</td>
<td>Θ(n)</td>
<td>Θ(n)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
</tr>
<tr>
<td>union(x, y)</td>
<td>Θ(n)</td>
<td>Θ(n)</td>
<td>Θ(n)</td>
<td>Θ(log n)</td>
<td>O(log* n)</td>
</tr>
</tbody>
</table>

And if log* n <= 5 for any reasonable input...
- We’ve just witnessed an incredible feat of data structure engineering: every operation is constant!*
- *Caveat: amortized constant, in the “in-practice” case; still logarithmic in the worst case!
Disjoint Sets Implementation

In-Practice Runtimes:

<table>
<thead>
<tr>
<th></th>
<th>(Baseline)</th>
<th>QuickFind</th>
<th>QuickUnion</th>
<th>WeightedQuickUnion</th>
<th>WQU + Path Compression</th>
</tr>
</thead>
<tbody>
<tr>
<td>makeSet(value)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
</tr>
<tr>
<td>find(value)</td>
<td>Θ(n)</td>
<td>Θ(1)</td>
<td>Θ(n)</td>
<td>Θ(log n)</td>
<td>O(log* n)</td>
</tr>
<tr>
<td>union(x, y) assuming root args</td>
<td>Θ(n)</td>
<td>Θ(n)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
</tr>
<tr>
<td>union(x, y)</td>
<td>Θ(n)</td>
<td>Θ(n)</td>
<td>Θ(n)</td>
<td>Θ(log n)</td>
<td>O(log* n)</td>
</tr>
</tbody>
</table>
Kruskal’s Runtime

$\Theta(|E| \log |E|)$

$\Theta(|E| \log |V|)$

$\Theta(|V| \log |V|)$

find and union are $\log |V|$ in worst case, but amortized constant “in practice”

Either way, dominated by time to sort the edges 😊

- For an MST to exist, $E$ can’t be smaller than $V$, so assume it dominates
- Note: some people write $|E| \log |V|$, which is the same (within a constant factor)

For an MST to exist, $E$ can’t be smaller than $V$, so assume it dominates

Note: some people write $|E| \log |V|$, which is the same (within a constant factor)
Using Arrays for Up-Trees

Since every node can have at most one parent, what if we use an array to store the parent relationships?

Proposal: each node corresponds to an index, where we store the index of the parent (or –1 for roots). Use the root index as the representative ID!

Just like with heaps, tree picture still conceptually correct, but exists in our minds!

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td>6</td>
<td>-1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Joyce</td>
<td>Sam</td>
<td>Aileen</td>
<td>Alex</td>
<td>Paul</td>
<td>Santino</td>
<td>Ken</td>
</tr>
</tbody>
</table>
Using Arrays: Find

Initial jump to element still done with extra Map

But traversing up the tree can be done purely within the array!

- Can still do path compression by setting all indices along the way to the root index!

```
find(A):
    index = jump to A node’s index
    while array[index] > 0:
        index = array[index]
    path compression
    return index
```

```
find(Alex) = 0
```

```
Joyce  Sam  Aileen  Alex  Paul  Santino  Ken
-1     0     -1     0     -1     2     0
```

```
Joyce (0)  Sam  Ken  Aileen (2)  Santino  Paul (4)
```

Aileen  Alex  Santino  Paul  Sam  Ken
Using Arrays: Union

For WeightedQuickUnion, we need to store the number of nodes in each tree (the weight).

Instead of just storing -1 to indicate a root, we can store -1 * weight!

union(A, B):
rootA = find(A)
rootB = find(B)
use -1 * array[rootA] and -1 * array[rootB] to determine weights
put lighter root under heavier root

union(Ken, Santino)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-4</td>
<td>0</td>
<td>-2</td>
<td>6</td>
<td>-1</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

Joyce | Sam  | Aileen | Alex | Paul | Santino | Ken

weight 4
Joyce (0)
Sam
Alex
Ken
Santino

weight 2
Aileen (2)

weight 1
Paul (4)
Using Arrays: Union

For WeightedQuickUnion, we need to store the number of nodes in each tree (the weight).

Instead of just storing -1 to indicate a root, we can store -1 * weight!

union(Ken, Santino)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>-6</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>-1</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

Joyce  Sam  Aileen  Alex  Paul  Santino  Ken

union(A, B):
rootA = find(A)
rootB = find(B)
use -1 * array[rootA] and -1 * array[rootB] to determine weights
put lighter root under heavier root

weight 6

weight 1
Array Implementation

rank = 0

\[
\begin{array}{c}
\text{0}
\end{array}
\]

rank = 3

\[
\begin{array}{c}
\text{1} \\
\text{2} \\
\text{3} \quad \text{4} \\
\text{5} \quad \text{6} \quad \text{7} \\
\text{8} \quad \text{9} \\
\text{10}
\end{array}
\]

rank = 3

\[
\begin{array}{c}
\text{11} \\
\text{12} \\
\text{13} \quad \text{14} \\
\text{15} \quad \text{16} \quad \text{17} \\
\text{18}
\end{array}
\]

\[
\begin{array}{r|rrrrrrrrrrrrrrrrrrr}
\end{array}
\]

Store \((\text{rank} \times -1) - 1\)

Each “node” now only takes 4 bytes of memory instead of 32
Practice

rank = 2

rank = 0

rank = 1

rank = 2

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>-3</td>
<td>3</td>
<td>-1</td>
<td>-2</td>
<td>6</td>
<td>12</td>
<td>13</td>
<td>13</td>
<td>0</td>
<td>13</td>
<td>-3</td>
<td>12</td>
<td>12</td>
<td>12</td>
</tr>
</tbody>
</table>
Using Arrays for WQU+PC

Same asymptotic runtime as using tree nodes, but check out all these other benefits:
- More compact in memory
- Better spatial locality, leading to better constant factors from cache usage
- Simplify the implementation!

<table>
<thead>
<tr>
<th></th>
<th>(Baseline)</th>
<th>QuickFind</th>
<th>QuickUnion</th>
<th>WeightedQuickUnion</th>
<th>WQU + Path Compression</th>
<th>ArrayWQU+PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>makeSet(value)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
</tr>
<tr>
<td>find(value)</td>
<td>Θ(n)</td>
<td>Θ(1)</td>
<td>Θ(n)</td>
<td>Θ(log n)</td>
<td>O(log* n)</td>
<td>O(log* n)</td>
</tr>
<tr>
<td>union(x, y)</td>
<td></td>
<td></td>
<td>Θ(n)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
</tr>
<tr>
<td>assuming root args</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>union(x, y)</td>
<td>Θ(n)</td>
<td>Θ(n)</td>
<td>Θ(n)</td>
<td>Θ(log n)</td>
<td>O(log* n)</td>
<td>O(log* n)</td>
</tr>
</tbody>
</table>
Appendix