Disjoint Sets
CSE 373 Winter 2020

Instructor: Hannah C. Tang

Teaching Assistants:
Aaron Johnston    Ethan Knutson    Nathan Lipiarski
Amanda Park      Farrell Fileas    Sam Long
Anish Velagapudi Howard Xiao    Yifan Bai
Brian Chan       Jade Watkins     Yuma Tou
Elena Spasova    Lea Quan
Announcements

- HW7 released; due Fri, Feb 28
  - HW7 exercises a different set of skills: reading/understanding a large codebase and figuring out where to plug in
  - **Read the spec carefully;** HW7 substantially longer than last quarter’s because we added a ton of hints
Feedback from Reading Quiz

❖ When do we use Disjoint Sets?

❖ Can we make union() constant time?

❖ How did you choose the values for the ids?
Lecture Outline

❖ Disjoint Set ADT

❖ QuickFind Data Structure

❖ QuickUnion Data Structure

❖ WeightedQuickUnion with Path Compression
Disjoint Sets ADT

The Disjoint Sets ADT has two operations:
- `find(e)`: gets the id of the element’s set
- `union(e1, e2)`: combines the set containing `e1` with the set containing `e2`

Example: ability to travel to drive to a country
- `union(france, germany)`
- `union(spanish, france)`
- `find(spanish) == find(germany)`?
- `union(england, france)`

**Disjoint Sets ADT.** A collection of elements and sets of those elements.

- An element can only belong to a single set.
- Each set is identified by a unique id.
- Sets can be combined/connected/unioned.
Disjoint Sets ADT

- The Disjoint Sets ADT has two operations:
  - `find(e)`: gets the id of the element’s set
  - `union(e1, e2)`: combines the set containing `e1` with the set containing `e2`

- Applications include percolation theory (computational chemistry) and .... Kruskal’s algorithm

- Simplifying assumptions
  - We can map elements to indices quickly (see reading)
  - We know all the items in advance; they’re all disconnected initially
Lecture Outline

❖ Disjoint Set ADT

❖ QuickFind Data Structure

❖ QuickUnion Data Structure

❖ WeightedQuickUnion with Path Compression
QuickFind (review)

find(A) == 123
find(B) == 123
find(A) == find(B)
find(C) != find(D)
union(C, D)
QuickFind (review)

\[
\begin{align*}
\text{find}(A) &= 123 \\
\text{find}(B) &= 123 \\
\text{find}(A) &= \text{find}(B) \\
\text{find}(C) &\neq \text{find}(D) \\
\text{union}(C, D)
\end{align*}
\]
Disjoint Sets: Runtime

- Feedback from reading quiz: “can we make union() constant time?”

<table>
<thead>
<tr>
<th></th>
<th>find</th>
<th>union</th>
</tr>
</thead>
<tbody>
<tr>
<td>QuickFind</td>
<td>$O(1)$</td>
<td>$O(N)$</td>
</tr>
</tbody>
</table>
Lecture Outline

❖ Disjoint Set ADT

❖ QuickFind Data Structure

❖ **QuickUnion Data Structure**

❖ WeightedQuickUnion with Path Compression
QuickUnion Data Structure

- Fundamental idea:
  - QuickFind tracks each element’s ID
  - QuickFind tracks each element’s parent. Only the root has an ID

```
find(A) == 123
find(B) == 123
find(A) == find(B)
find(C) != find(D)
```
QuickUnion: Representation

- Like the binary heap, we can represent this as an array
  - Note: we represent ids as negative numbers to clarify that they’re not indices

A B C E

D F

G

\[
\begin{array}{ccccccc}
\text{int[]} & \text{parents} \\
-123 & 0 & 0 & -456 & 2 & 4 & -789 \\
0 & 1 & 2 & 3 & 4 & 5 & 6
\end{array}
\]
QuickUnion: Union

- How does this help us with the union operation?

![Diagram of QuickUnion]

```
int[] parents = {-123, 0, 0, -456, 2, 4, 4, -789};
```
What changed when we implemented QuickUnion?

What are QuickUnion’s runtimes?

<table>
<thead>
<tr>
<th></th>
<th>find</th>
<th>union</th>
</tr>
</thead>
<tbody>
<tr>
<td>QuickFind</td>
<td>Θ(1)</td>
<td>Θ(N)</td>
</tr>
<tr>
<td>QuickUnion</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A. Θ(N) / Θ(1)  
B. Θ(N) / O(1)   
C. O(N) / Θ(1)   
D. O(N) / O(1)   
E. I’m not sure ...
Worst-case QuickUnion

union(A, B)
union(B, C)
union(C, D)
union(D, E)
union(E, F)
union(F, G)
...

🤔 If only I could keep these trees (semi-?)balanced
Lecture Outline

- Disjoint Set ADT
- QuickFind Data Structure
- QuickUnion Data Structure
- WeightedQuickUnion with Path Compression
WeightedQuickUnion

- We always picked the same side to become the child in the unioned structure

- Instead, let’s pick the smaller tree to be the child

union(A, B)
union(B, C)
union(C, D)
union(D, E)
union(E, F)
union(F, G)
...
**WeightedQuickUnion**

- Let’s pick the smaller tree to be the child

```
union(A, B)
union(B, C)
union(C, D)
union(D, E)
union(E, F)
union(F, G)
...  
```
WeightedQuickUnion

- Let’s pick the smaller tree to be the child

```
union(A, B)
union(B, C)
union(C, D)
union(D, E)
union(E, F)
union(F, G)
...
```
We need to store the sizes of each tree
  - We don’t need to store the root’s ID; we can just hash the element as needed
  - Which means we can store the weight there instead!
  - (we still use the convention of negative values to indicate they’re not indices, however)
Weighted Quick Union Performance

Let’s 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>
Weighted Quick Union Performance

Let’s 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>
Weighted Quick Union Performance

Let’s 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>
Weighted Quick Union Performance

Let’s 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>
Weighted Quick Union Performance

Let’s 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>
Weighted Quick Union Performance

Let’s 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>
Weighted Quick Union Performance

Let’s 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>
Weighted Quick Union Performance

Let’s 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>3</td>
</tr>
</tbody>
</table>
Weighted Quick Union Performance

Let’s consider the worst case where the tree height grows as fast as possible.

Worst case tree height is $\Theta(\log N)$. 

<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>3</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
</tr>
</tbody>
</table>
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.
WeightedQuickUnion Runtime

<table>
<thead>
<tr>
<th></th>
<th>find</th>
<th>union</th>
</tr>
</thead>
<tbody>
<tr>
<td>QuickFind</td>
<td>Θ(1)</td>
<td>Θ(N)</td>
</tr>
<tr>
<td>QuickUnion</td>
<td>O(N)</td>
<td>Θ(1)</td>
</tr>
<tr>
<td>WeightedQuickUnion</td>
<td>Θ(log N)</td>
<td>Θ(1)</td>
</tr>
</tbody>
</table>

- There’s one final optimization we can make: path compression
Path Compression

- Below is the topology of the worst case if we use WeightedQuickUnion
  - Clever idea: When we do find(15, 10), tie all nodes seen to the root
  - Additional cost is insignificant (same order of growth).
Path Compression

- Below is the topology of the worst case if we use WeightedQuickUnion
  - Clever idea: When we do find(15, 10), tie all nodes seen to the root
  - Additional cost is insignificant (same order of growth).
Path Compression

- Below is the topology of the worst case if we use WeightedQuickUnion
  - Clever idea: When we do find(15, 10), tie all nodes seen to the root
  - Additional cost is insignificant (same order of growth).
Path Compression: Runtime

- Path compression results in a union operations that are very very close to amortized constant time
  - M operations on N nodes is $O(N + M \lg^* N)$
  - $\lg^*$ is less than 5 for any realistic input.

<table>
<thead>
<tr>
<th>N</th>
<th>$\lg^* N$</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>$2^{64}$</td>
<td>&gt;5</td>
</tr>
<tr>
<td>$2^{65536}$</td>
<td>5</td>
</tr>
</tbody>
</table>
Runtime

- Disjoint Sets ADT implementations:

<table>
<thead>
<tr>
<th></th>
<th>find</th>
<th>union</th>
</tr>
</thead>
<tbody>
<tr>
<td>QuickFind</td>
<td>$\Theta(1)$</td>
<td>$\Theta(N)$</td>
</tr>
<tr>
<td>QuickUnion</td>
<td>$O(N)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>WeightedQuickUnion</td>
<td>$\Theta(\log N)$</td>
<td>$\Theta(1)$</td>
</tr>
</tbody>
</table>

- Kruskal’s Algorithm: $O(V \times \text{union} + E \times \text{find}) = O(V + E \log V)$
  - Assuming $E > V$, Kruskal’s $O(E \log V)$