Disjoint Sets

Key Ideas:
• Each element exists in exactly one set.
• Find returns a representative element
  • Programmatically: find(4) == find(8)
Implementation #1

Find(k):

Union(k1, k2):
Implementation #2

• We will continue to use an array where the index is the key

• The value of the array is:
  • -1, if we have found the representative element
  • The index of the parent, if we haven’t found the rep. element

• We will call these UpTrees:

```plaintext
0  1  2  3
-1 -1 -1 -1
```
# UpTrees

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
Disjoint Sets

```
0  1  2  3  4  5  6  7  8  9
4  8  5  6  -1 -1 -1 -1  4  5
```
Disjoint Sets Find

int DisjointSets::find() {
    if ( s[i] < 0 ) { return i; }
    else { return find( s[i] ); }
}

Running time?

What is the ideal UpTree?
Disjoint Sets Union

```cpp
void DisjointSets::union(int r1, int r2) {
}
```
Disjoint Sets – Union

<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>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>8</td>
<td>-1</td>
<td>10</td>
<td>7</td>
<td>-1</td>
<td>7</td>
<td>7</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>
Disjoint Sets – Smart Union

Union by height

<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>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Idea: Keep the height of the tree as small as possible.
Disjoint Sets – Smart Union

**Idea:** Keep the height of the tree as small as possible.

<table>
<thead>
<tr>
<th>Union by height</th>
<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>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

**Idea:** Minimize the number of nodes that increase in height

<table>
<thead>
<tr>
<th>Union by size</th>
<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>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

Both guarantee the height of the tree is: ______________.
Disjoint Sets Find

```cpp
int DisjointSets::find(int i) {
    if ( s[i] < 0 ) { return i; }
    else { return find( s[i] ); }
}
```

```cpp
void DisjointSets::unionBySize(int root1, int root2) {
    int newSize = arr_[root1] + arr_[root2];
    // If arr_[root1] is less than (more negative), it is the larger set;
    // we union the smaller set, root2, with root1.
    if ( arr_[root1] < arr_[root2] ) {
        arr_[root2] = root1;
        arr_[root1] = newSize;
    }
    // Otherwise, do the opposite:
    else {
        arr_[root1] = root2;
        arr_[root2] = newSize;
    }
}
```
Iterators Redux

• Still reading survey responses:
  • https://forms.gle/qbWkYeuPpzDHE3ZD9

• Will give additional opportunity on iterator problem:
  • Will put out 2 (updated) versions of the problem for practice
  • Optional 50 minute exam next week to do one of the other (updated) versions
    • Will average your scores between the two exams
Path Compression
Disjoint Sets Analysis

The *iterated log* function:

The number of times you can take a log of a number.

\[
\log^*(n) =
\begin{align*}
0, & \quad n \leq 1 \\
1 + \log^*(\log(n)), & \quad n > 1
\end{align*}
\]

What is \( \log^*(2^{65536}) \)?
Disjoint Sets Analysis

In an Disjoint Sets implemented with smart unions and path compression on find:

Any sequence of $m$ union and find operations result in the worse case running time of $O(\_\_\_\_\_\_\_\_\_\_\_\_\_\_)$, where $n$ is the number of items in the Disjoint Sets.