Implementation #1:

**Operation:** `find(k)`

**Operation:** `union(k1, k2)`

Implementation #2 - UpTrees:
- 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

Example using UpTrees:

```
  2 5 9
  7
  0 1 4 8
  3 6
```

...what is the error in this table?

```
<table>
<thead>
<tr>
<th>4</th>
<th>8</th>
<th>5</th>
<th>6</th>
<th>-1</th>
<th>-1</th>
<th>-1</th>
<th>-1</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>
```

What is the running time of `find`?

What is the ideal UpTree?

```
DisjointSets.cpp (partial)
1 int DisjointSets::find(int i) {
2   if ( s[i] < 0 ) { return i; }
3   else { return _find( s[i] ); }
4 }
```

How do we want to union the two UpTrees?

```
DisjointSets.cpp (partial)
1 void DisjointSets::union(int r1, int r2) {
2
3 }
4 }
```
Building a Smart Union Function

The implementation of this visual model is the following:

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<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>
</tr>
</tbody>
</table>

Metadata at Root:

After union(4, 7):

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<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>

Strategy #1: Union by Height

Idea: Keep the height of the tree as small as possible!

Metadata at Root:

After union(4, 7):

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<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>

Strategy #2: Union by Size

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

(Observe that the tree we union have all their nodes gain in height.)

Metadata at Root:

After union(4, 7):

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<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>

Smart Union Implementation:

```cpp
void DisjointSets::unionBySize(int root1, int root2) {
    int newSize = arr_[root1] + arr_[root2];
    if (arr_[root1] < arr_[root2]) {
        arr_[root2] = root1;
        arr_[root1] = newSize;
    } else {
        arr_[root1] = root2;
        arr_[root2] = newSize;
    }
}
```

Path Compression: