Disjoint Sets

S = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}
Partition of S = \{\{2, 5, 9\}, \{7\}, \{0, 1, 4, 8\}, \{3, 6\}\}
Disjoint Sets

Operation: find(4)
Disjoint Sets

Operation: \( \text{find}(4) == \text{find}(8) \)
Disjoint Sets

Operation:

```java
if ( find(2) != find(7) ) {
    union( find(2), find(7) );
}
```
Disjoint Sets

Key Ideas:
• Each element exists in exactly one set.
• Every set is an equitant representation.
  • Mathematically: $4 \in [0]_R \Rightarrow 8 \in [0]_R$
  • Programmatically: find(4) == find(8)
Disjoint Sets ADT

• Maintain a collection $S = \{s_0, s_1, \ldots, s_k\}$

• Each set has a representative member.

• API:  
  void addElements(int number);
  void union(int k1, int k2);
  int find(int k);
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:
UpTrees
Disjoint Sets

2 5 9

0 1 4 8

3 6

0 1 2 3 4 5 6 7 8 9

4 8 5 -1 -1 -1 3 -1 4 5
Disjoint Sets Find

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

Running time?

What is the ideal UpTree?
void DisjointSets::union(int r1, int r2) {
}

Disjoint Sets Union
Disjoint Sets – Union

```
0 1 2 3 4 5 6 7 8 9 10 11
6 6 6 8 -1 10 7 -1 7 7 4 5
```
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>5</td>
<td>7</td>
</tr>
</tbody>
</table>

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

Union by height

<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>
</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>
<td></td>
</tr>
</tbody>
</table>

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

Union by size

<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>
</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>
<td></td>
</tr>
</tbody>
</table>

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

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;
    } else { // Otherwise, do the opposite:
        arr_[root1] = root2;
        arr_[root2] = newSize;
    }
}
```
Path Compression
Disjoint Sets Analysis

The **iterated log** function:  
*The number of times you can take a log of a number.*

\[ \log^*(n) = \]

\[ 0 \quad , \quad n \leq 1 \]

\[ 1 + \log^*(\log(n)) \quad , \quad n > 1 \]

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

In a 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.