buildHeap

1. Sort the array – it’s a heap!

2. 

```cpp
1 template <class T>
2 void Heap<T>::buildHeap() {
3   for (unsigned i = 2; i <= size_; i++) {
4     heapifyUp(i);
5   }
6 }
```

3. 

```cpp
1 template <class T>
2 void Heap<T>::buildHeap() {
3   for (unsigned i = parent(size); i > 0; i--) {
4     heapifyDown(i);
5   }
6 }
```
Proving buildHeap Running Time

**Theorem:** The running time of buildHeap on array of size $n$ is: _________.

**Strategy:**
- 
- 
- 
-
Proving buildHeap Running Time

$S(h)$: Sum of the heights of all nodes in a complete tree of height $h$.

$S(0) =$

$S(1) =$

$S(h) =$
Proving buildHeap Running Time

Proof the recurrence:
Base Case:

General Case:
Proving buildHeap Running Time

From $S(h)$ to $\text{RunningTime}(n)$:

$S(h)$:

Since $h \leq \lg(n)$:

$\text{RunningTime}(n) \leq \ldots$
Heap Sort

Running Time?

Why do we care about another sort?
A(nother) throwback to CS 173...

Let $R$ be an equivalence relation on $us$ where $(s, t) \in R$ if $s$ and $t$ have the same favorite among:

\{___, ___, _____, __, ___, ___\}
Disjoint Sets

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

Operation: find(4)
Disjoint Sets

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

Operation:
if ( find(2) != find(7) ) {
    union( find(2), find(7) );
}
**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 makeSet(const T & t);
  void union(const T & k1, const T & k2);
  T & find(const T & 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 theses UpTrees:
UpTrees

0 1 2 3

0 1 2 3
-1 -1 -1 -1

0 1 2 3

0 1 2 3
Disjoint Sets

- Set 1: 5, 9, 2
- Set 2: 7
- Set 3: 0, 1, 4, 8
- Set 4: 3, 6

- Array:

<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>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>8</td>
<td>5</td>
<td>6</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>
Disjoint Sets Find

Running time?

What is the ideal UpTree?

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

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.

Union by size

<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: 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;
    }

    // Otherwise, do the opposite:
    else {
        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.*

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

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 $O(\text{___________})$, where $n$ is the number of items in the Disjoint Sets.