November 5 – Heap Analysis and Disjoint Sets
Wade Fagen-Ulmschneider
1. Sort the array – it’s a heap!

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

2. ```cpp
    template <class T>
    void Heap<T>::buildHeap() {
        for (unsigned i = parent(size); i > 0; i--) {
            heapifyDown(i);
        }
    }
```
$O(h) = 2$
$O(h) = 3$
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 $h$.

$S(0) = \quad$ 

$S(1) = \quad$ 

$S(h) = \quad$
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 $h \leq \lg(n)$:

$\text{RunningTime}(n) \leq$
Mattox Monday
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
- 7
- 0 1 4 8
- 3 6
Disjoint Sets

Operation: find(4)
Disjoint Sets

Operation: find(4) == 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 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 these UpTrees:
UpTrees
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

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) {
}
```

```
0
  1
   8
    4
  4
```
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

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

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>
Disjoint Sets – Smart Union

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

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

Both guarantee the height of the tree is: _____________.

<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>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>

<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>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>
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.

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

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.