Disjoint Sets

**2 5 9**

![Diagram of disjoint sets for 2, 5, and 9]

**7**

![Diagram of disjoint sets for 7]

**0 1 4 8**

![Diagram of disjoint sets for 0, 1, 4, and 8]

**3 6**

![Diagram of disjoint sets for 3 and 6]

---

<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>-1</td>
<td>-1</td>
<td>-1</td>
<td>3</td>
<td>-1</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>
Disjoint Sets Find

Running time?
Structure: A structure similar to a linked list
Running time: $O(h) = O(n)$

What is the ideal UpTree?
Structure: One root node with every other node as it’s child
Running Time: $O(1)$

```cpp
int DisjointSets::find() {
    if ( s[i] < 0 ) { return i; }
    else { return _find( s[i] ); }
}
```
void DisjointSets::union(int r1, int r2) {
}

Disjoint Sets – Union
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></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>7</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

**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: ______________.
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.*

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

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

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

Any sequence of \textbf{m union} and \textbf{find} operations result in the worse case running time of $O(\text{___________})$, where $n$ is the number of items in the Disjoint Sets.
In Review: Data Structures

Array
- Sorted Array
- Unsorted Array
  - Stacks
  - Queues
  - Hashing
  - Heaps
    - Priority Queues
- UpTrees
  - Disjoint Sets

List
- Doubly Linked List
- Skip List
- Trees
  - BTree
  - Binary Tree
    - Huffman Encoding
    - kd-Tree
    - AVL Tree
- Constant time access to any element, given an index $a[k]$ is accessed in $O(1)$ time, no matter how large the array grows.

- Cache-optimized
  
  Many modern systems cache or pre-fetch nearby memory values due the “Principle of Locality”. Therefore, arrays often perform faster than lists in identical operations.
• Efficient general search structure
  Searches on the sort property run in $O(\lg(n))$ with Binary Search

• Inefficient insert/remove
  Elements must be inserted and removed at the location dictated by the sort property, resulting shifting the array in memory – an $O(n)$ operation
• Constant time add/remove at the beginning/end
  Amortized O(1) insert and remove from the front and of the array
  **Idea:** Double on resize

• Inefficient global search structure
  With no sort property, all searches must iterate the entire array; O(1) time
First In First Out (FIFO) ordering of data
Maintains an arrival ordering of tasks, jobs, or data

All ADT operations are constant time operations
enqueue() and dequeue() both run in $O(1)$ time
Last In First Out (LIFO) ordering of data
Maintains a “most recently added” list of data

All ADT operations are constant time operations
push() and pop() both run in O(1) time
In Review: Data Structures

Array
- Sorted Array
- Unsorted Array
  - Stacks
  - Queues
- Hashing
- Heaps
  - Priority Queues
- UpTrees
  - Disjoint Sets

List
- Doubly Linked List
- Skip List
- Trees
  - BTree
  - Binary Tree
    - Huffman Encoding
    - kd-Tree
    - AVL Tree
In Review: Data Structures

**Array**
- Sorted Array
- Unsorted Array
  - Stacks
  - Queues
  - Hashing
  - Heaps
    - Priority Queues
  - UpTrees
    - Disjoint Sets

**List**
- Doubly Linked List
- Skip List
- Trees
  - BTree
  - Binary Tree
    - Huffman Encoding
    - kd-Tree
    - AVL Tree

**Graphs**
The Internet 2003
The OPTE Project (2003)
Map of the entire internet; nodes are routers; edges are connections.
Who’s the real main character in Shakespearean tragedies?

Martin Grandjean (2016)

“Rush Hour” Solution
Unknown Source
Presented by Cinda Heeren, 2016
Wolfram\textregistered\textsc{Alpha}'s "Personal Analytics" for Facebook

Generated: April 2013 using Wade Fagen-Ulmschneider's Profile Data
This graph can be used to quickly calculate whether a given number is divisible by 7.

1. Start at the circle node at the top.
2. For each digit $d$ in the given number, follow $d$ blue (solid) edges in succession. As you move from one digit to the next, follow 1 red (dashed) edge.
3. If you end up back at the circle node, your number is divisible by 7.

3703
Conflict-Free Final Exam Scheduling Graph

Unknown Source
Presented by Cinda Heeren, 2016
Class Hierarchy At University of Illinois Urbana-Champaign
A. Mori, W. Fagen-Ulmschneider, C. Heeren

Graph of every course at UIUC; nodes are courses, edges are prerequisites

http://waf.cs.illinois.edu/discovery/class_hierarchy_at_illinois/
“Stanford Bunny”
Greg Turk and Mark Levoy (1994)
Graphs

To study all of these structures:
1. A common vocabulary
2. Graph implementations
3. Graph traversals
4. Graph algorithms