CS 225
Data Structures

April 5 – Disjoint Sets Finale + Graphs
Wade Fagen-Ulmschneider, Craig Zilles
Disjoint Sets Find

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

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 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 \( O(\underline{\text{_____________}}) \), 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**
- Singly Linked List
- Doubly Linked 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 to 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; $O(n)$ 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

Array

Unsorted Array

Stack (LIFO)
In Review: Data Structures

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

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

<table>
<thead>
<tr>
<th>Array</th>
<th>List</th>
<th>Graphs</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Sorted Array</td>
<td>- Doubly Linked List</td>
<td>- Trees</td>
</tr>
<tr>
<td>- Unsorted Array</td>
<td>- Skip List</td>
<td></td>
</tr>
<tr>
<td>- Stacks</td>
<td>- BTree</td>
<td>- Huffman Encoding</td>
</tr>
<tr>
<td>- Queues</td>
<td>- Binary Tree</td>
<td>- kd-Tree</td>
</tr>
<tr>
<td>- Hashing</td>
<td></td>
<td>- AVL Tree</td>
</tr>
<tr>
<td>- Heaps</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Priority Queues</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- UpTrees</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Disjoint Sets</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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|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/
MP Collaborations in CS 225

Unknown Source
Presented by Cinda Heeren, 2016
“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
Graph Vocabulary

G = (V, E)
|V| = n
|E| = m

Incident Edges:
I(v) = \{ (x, v) in E \}

Degree(v):
|I|

Adjacent Vertices:
A(v) = \{ x : (x, v) in E \}

Path(G₂): Sequence of vertices connected by edges

Cycle(G₁): Path with a common begin and end vertex.

Simple Graph(G): A graph with no self loops or multi-edges.
Graph Vocabulary

\[ G = (V, E) \]

\[ |V| = n \]

\[ |E| = m \]

Subgraph(G):
\[ G' = (V', E') : \]
\[ V' \in V, E' \in E, \text{ and } (u, v) \in E \Rightarrow u \in V', v \in V' \]

Complete subgraph(G)
Connected subgraph(G)
Connected component(G)
Acyclic subgraph(G)
Spanning tree(G)
Running times are often reported by $n$, the number of vertices, but often depend on $m$, the number of edges.

How many edges? **Minimum edges:**
- Not Connected:
  - Connected*:

**Maximum edges:**
- Simple:
- Not simple:

$$\sum_{v \in V} \deg(v) =$$
Connected Graphs