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 **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.
In Review: Data Structures

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

Linked
- Doubly Linked 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

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

Graphs
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:**

\[ \text{Degree}(v) = |I| \]

**Adjacent Vertices:**

\[ A(v) = \{ x : \{x, v\} \in E \} \]

**Path:**

\( \text{Path}(G_2) \): Sequence of vertices connected by edges

**Cycle:**

\( \text{Cycle}(G_1) \): Path with a common begin and end vertex with at least 3 vertices.

**Simple Graph:**

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

\[ G = (V, E) \]
\[ |V| = n \]
\[ |E| = m \]

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

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:

\[
\sum_{v \in V} \deg(v) =
\]
Graph ADT

Data:
- Vertices
- Edges
- Some data structure maintaining the structure between vertices and edges.

Functions:
- insertVertex(K key);
- insertEdge(Vertex v1, Vertex v2, K key);
- removeVertex(Vertex v);
- removeEdge(Vertex v1, Vertex v2);
- incidentEdges(Vertex v);
- areAdjacent(Vertex v1, Vertex v2);
- origin(Edge e);
- destination(Edge e);
Graph Implementation Idea

\[ u - a - v ~ c ~ b ~ w ~ d ~ z \]
Graph Implementation: Edge List

Vertex Collection:

Edge Collection:
Graph Implementation: Edge List

insertVertex(K key):

removeVertex(Vertex v):

\[
\begin{array}{c}
\text{u} \\
\text{v} \\
\text{w} \\
\text{z}
\end{array}
\quad
\begin{array}{ccc}
\text{u} & \text{v} & \text{a} \\
\text{v} & \text{w} & \text{b} \\
\text{u} & \text{w} & \text{c} \\
\text{w} & \text{z} & \text{d}
\end{array}
\]
Graph Implementation: Edge List

incidentEdges(Vertex v):

areAdjacent(Vertex v1, Vertex v2):

G.incidentEdges(v1).contains(v2)
Graph Implementation: Edge List

```
insertEdge(Vertex v1, Vertex v2, K key):
```

![Graph Diagram]
Graph Implementation: Adjacency Matrix

insertVertex(K key);
removeVertex(Vertex v);
areAdjacent(Vertex v1, Vertex v2);
incidentEdges(Vertex v);

<table>
<thead>
<tr>
<th></th>
<th>u</th>
<th>v</th>
<th>w</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>u</td>
<td></td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>v</td>
<td>b</td>
<td></td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>w</td>
<td>c</td>
<td></td>
<td></td>
<td>d</td>
</tr>
<tr>
<td>z</td>
<td>d</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>