## Heap Operations & Complexities

<table>
<thead>
<tr>
<th>Operation</th>
<th>LinkedList</th>
<th>Binary Heap</th>
<th>Binomial Heap</th>
<th>Fibonacci Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>MakeHeap</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Insert</td>
<td>1</td>
<td>Log n</td>
<td>Log n</td>
<td>1</td>
</tr>
<tr>
<td>findMin</td>
<td>n</td>
<td>1</td>
<td>Log n</td>
<td>1</td>
</tr>
<tr>
<td>deleteMin</td>
<td>n</td>
<td>Log n</td>
<td>Log n</td>
<td>Log n</td>
</tr>
<tr>
<td>decreaseKey</td>
<td>1</td>
<td>Log n</td>
<td>Log n</td>
<td>1</td>
</tr>
<tr>
<td>Delete</td>
<td>n</td>
<td>Log n</td>
<td>Log n</td>
<td>Log n</td>
</tr>
<tr>
<td>Union /Merge</td>
<td>1</td>
<td>n</td>
<td>Log n</td>
<td>1</td>
</tr>
</tbody>
</table>
Static Sets

Search: Is x in S?

Implement set as a List

- Array
- Linked List
- Bit Maps
Dynamic Set operations

Need to maintain a collection of dynamic Sets

- SEARCH: Find-Set(u)
- INSERT: Union(u,v)
- Make-Set(u)
- Implementations
  - Lists, BitMaps, …
Connected Components

- Given a graph, compute all connected components
  - DFS or BFS
  - $O(m+n)$ time
Connected Components w/ Sets

![Graph example](image)

<table>
<thead>
<tr>
<th>Edge processed</th>
<th>Collection of disjoint sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial sets</td>
<td>{a}</td>
</tr>
<tr>
<td>(b,d)</td>
<td>{b,d}</td>
</tr>
<tr>
<td>(e,g)</td>
<td>{a}</td>
</tr>
<tr>
<td>(a,c)</td>
<td>{a,c}</td>
</tr>
<tr>
<td>(h,i)</td>
<td>{a,c}</td>
</tr>
<tr>
<td>(a,b)</td>
<td>{a,b,c,d}</td>
</tr>
<tr>
<td>(e,f)</td>
<td>{a,b,c,d}</td>
</tr>
<tr>
<td>(b,c)</td>
<td>{a,b,c,d}</td>
</tr>
</tbody>
</table>
Connected Components w/ Sets

```
CONNECTED-COMPONENTS(G)
1 for each vertex \( v \in G.V \)
2 MAKE-SET( )
3 for each edge \( (u, v) \in G.E \)
4 if FIND-SET(u) \( \neq \) FIND-SET( )
5 UNION(u, )

SAME-COMPONENT(u, )
1 if FIND-SET(u) == FIND-SET( )
2 return TRUE
3 else return FALSE
```
Dynamic **Set** operations

Need to maintain a collection of dynamic **Sets**

- **SEARCH:** Find-Set(u)
- **INSERT:** Union(u,v)
- **Make-Set**(u)
- **Implementations**
  - Lists, BitMaps, …

Enough to deal with **Disjoint Sets**
Implementation Challenges

- **Array Implementations**
  - Series of unions become very expensive
    - $O(n^2)$

- **Linked List Implementations**
  - Series of unions become very expensive $O(n^2)$

- **Bit Implementations**
  - Can be $O(n^2)$ bit operations
Union Operation

FindSet is $O(1)$ time.
Union is $O(n)$ time.

How do we do FindSet?
How do we do FindSet?

- FindSet is O(h) time.
- Union is O(1) time.
- Union is O(1) + 2h time.
Trees in forest need not be binary trees
Even better Union operations

- Problem with FindSet
  - Height determines time complexity
  - Height determined by order of operations
- Always attach smaller tree to larger tree
  - Why is this better?
  - Guarantees height = $O(\log n)$

- FindSet is $O(\log n)$ time.
- Union is $O(1)$ time.
- Union is $O(\log n)$ time
Even better \text{FindSet} \text{ operation}

\text{FindSet}(a)
Example of Path Compression
UnionFind Data Structure

MAKE-SET(x)
1 x.p = x
2 x.rank = 0

FIND-SET(x)
1 if x ≠ x.p
2 x.p = FIND-SET(x.p)
3 return x.p

UNION(x, y)
1 LINK(FIND-SET(x), FIND-SET(y))

LINK(x, y)
1 if x.rank > y.rank
2 y.p = x
3 else x.p = y
4 if x.rank == y.rank
5 y.rank = y.rank + 1
Union-FInd w/ Path Compression

Given m operations on n elements,

Time Complexity = $O(m \alpha(n))$

For integers $k \geq 0$ and $j \geq 1$, we define the function $A_k \cdot j / D$ as:

$$A_k \cdot j / D = \begin{cases} j \cdot \lfloor 1 \rfloor & \text{if } k = 0 \\ A_k \cdot j / C & \text{if } k > 0 \\ A_k \cdot j / 1 & \text{if } k = 1 \end{cases}$$

$$\alpha(n) = \min \{k : A_k(1) = n\}$$
More about Ackermann function

\[ A_1(j) = 2j + 1 \]
\[ A_2(j) = 2^{j+1}(j + 1) \]