Sets

Disjoint Sets

- Disjoint Sets
  - We have a fixed set **U** of Elements $\mathbf{X_i}$
  - **U** is divided into a number of disjoint subsets $S_1, S_2, S_3, \ldots S_k$
  - $S_i \cap S_j$ is empty $\forall\ i \neq j$
  - $S_1 \cup S_2 \cup S_3 \cup \ldots S_k = \mathbf{U}$

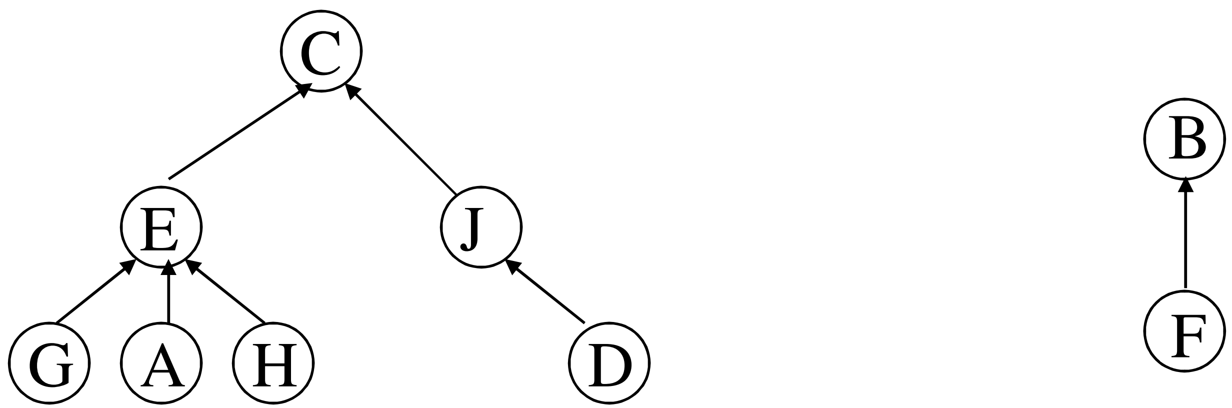
Disjoint Sets

- Operations on Disjoint Sets
  - **MakeSet(X):** Return a new set consisting of the single item **X**
  - **Union(S,T):** Return the set **S** $\cup$ **T**, which replaces **S** and **T** in the data base
  - **Find(X):** Return that set **S** such that **X** $\in$ **S**

- If each element can belong to only one set (definition of disjoint), a tree structure known as an **up-tree** can be used to represent disjoint sets
- Up-trees have pointers up the tree from children to parents

Up-Trees

- Properties
  - Each node has a single pointer field to point to its parent; at the root this field is empty
  - A node can have any number of children
  - The sets are identified by their root nodes



Disjoint Sets: {A,C,D,E,G,H,J} and {B,F}

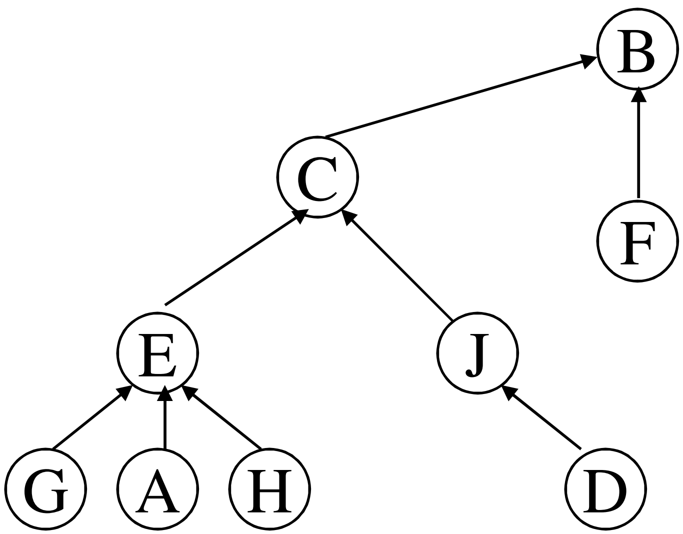
Up-Trees: **Union**

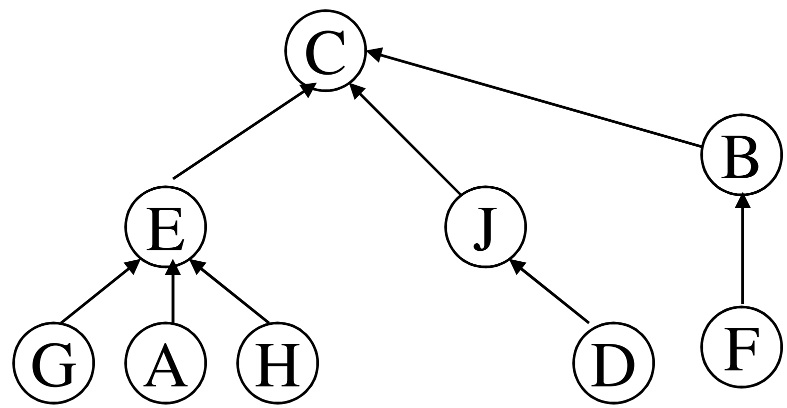
- **Union(S, T):** Just make the root of one tree point to the tree of the other. If we make root of **S** point to the root of **T**, we say we are merging **S** into **T**
- **Optimization**: Prevent the linear growth of the height of the tree – always merge the "smaller" tree into the larger one
  - By height – worst case optimization
  - By size (# of nodes) – avg case optimization
    - Increases depth of fewer nodes → minimizes expected depth of a node

Up-Trees: **Union**

- Called the **balanced merging strategy**
- Why avoid linear growth of the tree's height?
  - **Find** takes time proportional to the height of the tree in the worst case
- Implementation: Each node has an additional **Count** field that is used, if the node is the root, to hold the count of the number of nodes in the tree (or height)
- Running time: **O(1)** if we assume **S** and **T** are roots, else running time of Find(X)

Up-Trees: **Union by Size**





(a)

Incorrect Way

(b)

Correct Way

Up-Trees: **Find**

- **Find(X):** which set does an element belong to? Follow the pointers up the tree to the root
  - But where is X?

- LookUp(X): Get the location of the node **X**
  - If we assume we know location of X → constant time
    - Then Running time of Find(X): **O(log n)**
  - If we cannot directly access the node → logarithmic
    - E.g. use a balanced tree to index nodes
    - Then Running time of Find(X): still **O(log n)**
      - log time to LookUp node + log time to search up the up-tree

Up-Trees: **Find**

- **Optimization: Path Compression**
  - **Find(X)** would take less time in a shallow, bushy tree than it would in a tall, skinny tree
  - Use balanced merging strategy to prevent the growth in the tree's height → ensures height is at worst logarithmic in size
  - However, since any number of nodes can have the same parent, we can **restructure** our up-tree to make it bushier…

Up-Trees: **Find**

- **Path Compression:** After doing a **Find**, make any node on that path to the root point directly to the root.
- Any subsequent **Find** on any one of these nodes or their descendants will take less time since the node is now closer to the root.
- Minor "problem" when combined with Union-by-height
  - Treat height as an estimate → Union-by-rank

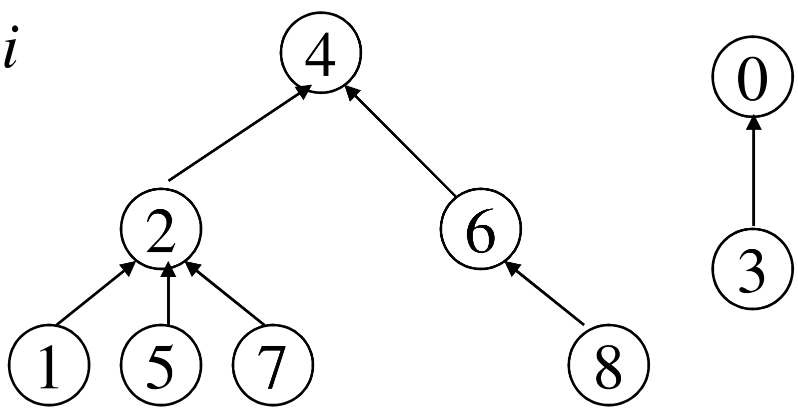
Up-Trees: **Array Implementation**

- If we assume all elements of the universe to be integers from 0 to N, then we can represent the Up-Trees as one Array of size N

	0	1	2	3	4	5	6	7	8
a	-1	2	4	0	-1	2	4	2	6

a[i] = parent of i

a[root] = -1



Summary

- Disjoint Sets
  - Operations on Disjoint Sets
  - Up-Tree as a Disjoint Set
    - Array Implementation
    - Union: O(log*n), Find: O(log*n)