Given a finite set $S = \{x_1, x_2, \ldots, x_n\}$, $n \geq 1$. A partition $P$ of $S$ is a collection of $k$ subsets $\{S_i : S_i \subseteq S, 1 \leq i \leq k \leq n\}$ such that (1) $S_i \cap S_j = \emptyset$, $\forall \ i \neq j$, and (2) $\bigcup_{i=1}^{k} S_i = S$.

Observations:
1. A trivial partition $P$ of $S$ can be obtained by simply defining $S_i = \{x_i\}$, for all $i$, $1 \leq i \leq n$.
2. Given a partition $P$ of $S$ with $k$ sets, $k \geq 2$, a new partition of $S$ can be formed by taking some unions of the sets in $P$.
3. One can perform at most $n-1$ union operations on the sets in $P$ before $S$ is re-generated.

Questions:
For any given partition $P$ of $S$,
1. How do we represent/naming a set?
2. What kind of data structure should we use to implement a set so as to support a large number of the following basic operations:
   (i) $\text{find}(x)$, $x \in S$: Return the (unique) set containing $x$.
   (ii) $\text{union}(x,y)$: Return the union of the two sets with representative $x$, and $y$, respectively.
A Simple Approach:
- Given any set $S_i$. Use an element $x \in S_i$ as the representative/name of the set.
- Use a tree to implement a set with the root of the tree being the representative of the set.

Implementing a Set using a Tree Structure:
Each set $S_i$ will be implemented as a (rooted) tree $T_i$ such that each element $x \in S_i$ will be represented by a node with a parent pointer pointing to its parent in $T_i$.

Example:

Remark: The array of pointers will be used to access the nodes of the tree.
Set Union and Trees Merging:
Given two sets $S_i$ and $S_j$ with representatives $x_i$ and $x_j$, the operation $\text{union}(x_i, x_j)$ is equivalent to merging two trees with roots $x_i$ and $x_j$ together. To merge two trees together, one can simply set the parent pointer of $x_i$ to point at $x_j$ with $x_j$ being the representative of the new set. Observe that, independent of the sizes of the two sets, $\text{union}(x_i, x_j)$ can now be executed in $O(1)$ time.

Union and Find Operations in Disjoint Sets:
If each element $x \in S$ is being used to represent a data object, then one of the most important and relevant operations in a partition of $S$ will be the $\text{find}$ operation, $\text{find}(x)$, which will return the representative of the unique set that contains $x$.

Basic find Operation:
- Using the array of pointers to locate the element $x$, and the tree containing $x$, among the trees in a partition.
- Follow the parent pointer of $x$ to the root of the tree that contains $x$.
- Return the root of the tree that contains $x$.

Complexity of find Operations:
Recall that if one performs $O(n)$ union operations on the sets of a given partition using the above tree structures and trees merging algorithm, a tree with height $O(n)$ can be formed. Hence, in order to find the root of the tree that contains $x$, one may have to follow the parent pointers of a leaf to the root, resulting in $T(n) = O(n)$.
Amortized Analysis of Disjoint Set Operations:
Given a sequence of $O(n)$ union operations intermixed with a sequence of $O(m)$ find operations, where $m \gg n$, what is a good data structure and its complexity $T(m,n)$ in performing these $O(n)$ union and $O(m)$ find operations?

Simple Approach:
Using the above implementation, we have $T(m,n) = O(mn)$.

Questions:
Can we do it better?

Two Possible Approaches:
1. Minimize the height of the resulting tree during union operation.
2. Modify the structure of a tree during find operations so that subsequent find operations performed on the same tree can be speeded up.

Union-by-Height Heuristic:
For each node $x$ in a tree $T$, recall that the height of $x$, $h(x)$, is the length of a longest path from $x$ to a leaf node in $T$. In performing union($x_i,x_j$) operation, the parent pointer of $x_i$ is set to point at $x_j$ iff $h(x_i) \leq h(x_j)$. Otherwise, we will set the parent pointer of $x_j$ to point at $x_i$ instead. Observe that, by using this union-by-height heuristic, the height of the resulting tree will increase by 1 iff both trees have the same height.
Example: Performing union(x,y) using union-by-height with h(x) \geq h(y).

Other Union Heuristics:
1. Union-by-Rank Heuristic:
   For each node x in S, define rank(x) as followed:
   Initially, when x is in a tree by itself, rank(x) = 0. When performing union(x_i,x_j) operation, the parent pointer of x_i is set to point at x_j iff rank(x_i) \leq rank(x_j). And if rank(x_i) = rank(x_j), then rank(x_j) = rank(x_j) + 1. If rank(x_i) > rank(x_j), then we will set the parent pointer of x_j to point at x_i instead. Observe that, by using this union-by-rank heuristic, the rank of the resulting tree will increase by 1 iff both trees have the same rank. Also, the rank of a root can only be changed during union operation. Hence, the rank(x) \geq h(x), \forall x \in S.

2. Union-by-Weight/Size Heuristic:
   For each node x in S, define w(x) be the number of nodes, including x, in the tree rooted at x. When performing union(x_i,x_j) operation, the parent pointer of x_i is set to point at x_j iff w(x_i) \leq w(x_j). Otherwise, we will set the parent pointer of x_j to point at x_i instead. Observe that, by using this union-by-weight heuristic, the weight of the resulting tree will always be increased by the size of the additional tree.
Path Compression Heuristic:
In performing find(x) operation, after the tree T that contains x and the root r of T is identified, every node on the path from x to r will be made a new child of r.

Example: Performing find(x) using path compression.

Theorem (Tarjan): By using union-by-rank and path compression heuristics, $T(m,n) = O(m \alpha(m,n))$, where $\alpha(m,n)$ is the inverse Ackerman’s function $A(m,n)$. 
**Review: Ackermann’s Function**

Define the **Ackermann’s function** $A : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ by

- $A(0, n) = n + 1,$
- $A(m, 0) = A(m-1, 1),$ if $m > 0,$
- $A(m, n) = A(m-1, A(m, n-1)),$ if $m, n > 0.$

**Example:**

- $A(0, 0) = 1,$
- $A(0, 1) = 2,$
- $A(0, 2) = 3,$
- $A(0, 3) = 4,$

…

- $A(1, 0) = A(0, 1) = 2,$
- $A(1, 1) = A(0, A(1, 0)) = A(0, 2) = 3,$
- $A(1, 2) = A(0, A(1, 1)) = A(0, 3) = 4,$
- $A(1, 3) = 5,$

…

- $A(2, 0) = A(1, 1) = 3,$
- $A(2, 1) = A(1, A(2, 0)) = A(1, 3) = 5,$
- $A(2, 2) = A(1, A(2, 1)) = A(1, 5) = 7,$
- $A(2, 3) = 9,$

…

- $A(3, 0) = A(2, 1) = 5,$
\[ A(3, 1) = A(2, A(3, 0)) = A(2, 5) = 13, \]
\[ A(3, 2) = A(2, A(3, 1)) = A(2, 13) = 29, \]
\[ A(3, 3) = A(2, A(3, 2)) = A(2, 29) = 61, \]

\[ \ldots \]

\[ A(4,0) = A(3,1) = 13, \]
\[ A(4,1) = A(3,A(4,0)) = A(3,13), \]
\[ A(4,2) = ? \]

**Remark:** \( A(m,n) \) is an extremely fast growing function and, hence, its inverse function \( \alpha(m,n) \), which often appears in data structures analyses and counting, grows extremely slow. For all practical purpose, \( \alpha(m,n) \) can be treated as a constant.