Disjoint Set

Equivalence Relation

An equivalence relation R is defined on a set U that satisfies the following three properties:

- *Reflexive*: $(a\ R\ a)$ is true, $\forall\ a \in U$
- *Symmetric*: $(a\ R\ b) \implies (b\ R\ a), \forall\ a, b \in U$
- *Transitive*: $(a\ R\ b) \wedge (b\ R\ c) \implies (a\ R\ c)\ \forall\ a, b, c \in U$

An equivalence relation R divides all elements in U into *disjoint sets* of equivalent items. All items in a disjoint set belongs to the same *equivalence class*.

Dynamic Equivalence Problem

Given a set $U = \{a_1, a_2, ..., a_n\}$, dynamically maintain a partition S defined as $\{S_1, S_2, ..., S_k\}$ such that:

- $S_i \cap S_j = \varnothing$, for each pair of subset in S
- $U = \bigcup_{i=1}^{k} S_i$
- Each subset has a unique identifier (name).

Given a set of elements and an equivalence relation, the problem is to find all the disjoint sets. The problem has two parts to it:

- **Find** the disjoint set of an element.
- Add an element (**Union**) to its disjoint set.

Up-Trees

Up-tree data structure is a collection of nodes in which each node has a pointer to its parent. Root of an up-tree is the node with no parent. A disjoint set S_i is an up-tree with its root as the representative member and the partition S is a forest of up-trees.
The efficacy of this data structure comes from the fact that it can be implicitly stored in an array. If S is the array, then an element X is stored as,

$$S[X] = \begin{cases} parent\ of\ X & if\ X\ is\ not\ root \\ -1 & if\ X\ is\ root \end{cases}$$

In order to minimize height of the up-tree, a clever implementation trick is to store size or height of the up-tree in place of -1. Union thus performed is referred to as union-by-size and union-by-height respectively.

When Union-by-Size or Union-by-Height is used, the height of an up-tree is $O(logN)$

Consider using union-by-size. The analysis is pretty similar for union-by-height. The minimum number of nodes N in an up-tree of height h using union-by-size is 2^h. We prove the statement using method of induction.

- *Base case*: For $h = 0$, $2^0 = 1$ node is valid.
- *Induction Hypothesis*: Suppose it is true for $h < h'$
- *Induction Step*: An up-tree of height h' is formed with the union of two up-trees of height $h' - 1$. From induction hypothesis, each of them has at least $2^{h'-1}$ nodes. So the total number of nodes $\geq 2^{h'}$. Hence, it is true for all h.

Thus we have,

$$N \geq 2^h$$

$$h \leq logN$$

When Union-by-Size or Union-by-Height is used, the worst case run time for Find operation is $O(logN)$ and for Union operation is $O(1)$

Find = $O(logN)$
Union = $O(1)$

Find operation can further be improved using path compression strategy. The concept is similar to that of the Splay tree. After every find operations, all the nodes along the path of find is pointed to the root reducing height of the entire path to $O(1)$.

When both path compression and Union-by-Size are used, the worst case run time for a sequence of M operations (Unions or Finds) is $\Theta(M\alpha(M, N))$

$\alpha(M, N)$ is the inverse of Ackermann's function, a very very slow growing function. In fact, for all practical purposes its value is less than 4. Thus running time for M operations, $\Theta(M\alpha(M, N))$ is very close to being linear (but not linear since $\alpha(M, N)$ is not constant). Consequently, both operations (union and find) is nearly $O(1)$!