Union-Find Data Structure

In computer science, a union–find data structure, also called a disjoint-set data structure or merge–find set, is a data structure that keeps track of a set of elements partitioned into a number of disjoint (nonoverlapping) subsets. It supports two useful operations:

findSet(): Determine which subset a particular element is in. findSet typically returns an item from this set that serves as its "representative"; by comparing the result of two findSet operations, one can determine whether two elements are in the same subset.

union(): Join two subsets into a single subset.

The other important operation, **makeSet()**, which makes a set containing only a given element (a singleton), is generally trivial. With these three operations, many practical partitioning problems can be solved.

In order to define these operations more precisely, some way of representing the sets is needed. One common approach is to select a fixed element of each set, called its representative, to represent the set as a whole. Then, findSet(x) returns the representative of the set that x belongs to, and union takes two set representatives as its arguments.

Two standard implementations are: **Union-find linked lists** and **Union-find forest**. In linked list method, findSet() requires 1 step, whereas union() requires many steps. The opposite is holds for union-find forest method.

Union-find forest

In this approach each set is represented by a tree data structure. When trees are used to represent the disjoint union-find sets, the tree root is used to label the set. All the vertices on a tree are considered to be in the same set. In some applications, initially each vertex is in a singleton set and is the root of its own tree. In general a set a tree node points to its parent on the tree and the root node points to itself. This allow a traversal from a set element to the tree root in a simple manner. A simple array parent[V+1] is sufficient for this where parent[u] indicates the vertex that u is attached to in the tree. Note that: parent[root] == root.

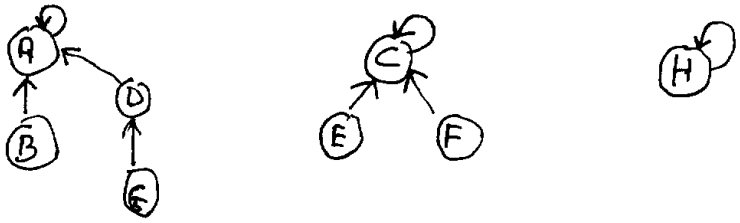
In this approach, findSet(u) means travelling back the tree from vertex u until the root is found and so may take more than 1 step. These operations are illustrated below in example below.

Union - Find Example

I will used trees to represent the sets and these trees can be stored in a simple 1-D array.

Consider the sets $\{A, B, D, G\}$, $\{C, E, F\}$ $\{H\}$

Could be stored as



Each set is identified by the root of its tree.

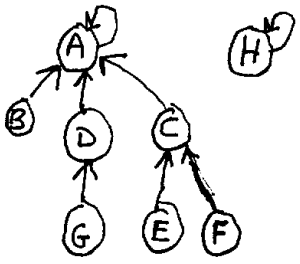
So findSet(G) = A and findSet(F) = C

A	B	C	D	E	F	G	H
A	A	C	A	C	C	D	H

$A \neq C$, so G and F are in 2 different sets.

union(set_G, set_F) = union(A, C) gives

A	B	C	D	E	F	G	H
A	A	A	A	C	C	D	H



Pseudocode

```
function makeSet(x)
     treeParent[x] := x

function findSet(x)
     if treeParent[x] == x
        return x
     else
        return findSet(treeParent[x])

 function union(xRoot, yRoot)
     treeParent[yRoot] := xRoot
```

Improvements

Union by rank

The first way, called union by rank, is to always attach the smaller tree to the root of the larger tree. Since it is the depth of the tree that affects the running time, the tree with smaller depth gets added under the root of the deeper tree, which only increases the depth if the depths were equal. In the context of this algorithm, the term rank is used instead of depth since it stops being equal to the depth if path compression (described below) is also used. One-element trees are defined to have a rank of zero, and whenever two trees of the same rank r are united, the rank of the result is r+1. Just applying this technique alone yields a worst-case running-time of O(log n) for the Union or Find operation.

Path compression

The second improvement, called path compression, is a way of flattening the structure of the tree whenever findSet() is used on it. The idea is that each node visited on the way to a root node may as well be attached directly to the root node; they all share the same representative. To effect this, as findSet() recursively traverses up the tree, it changes each node's parent reference to point to the root that it found. The resulting tree is much flatter, speeding up future operations not only on these elements but on those referencing them.

```
function makeSet(x)
     parent[x] := x
     rank[x]   := 0
```

```
 function Union(xRoot, yRoot)
     // x and y are not already in same set. Merge them.
     if rank[xRoot] < rank[yRoot]
         parent[xRoot] := yRoot
     else if rank[xRoot] > rank[yRoot]
         parent[yRoot] := xRoot
     else
         parent[yRoot] := xRoot
         rank[xRoot] := rank[xRoot] + 1
```

```
function findSet(x)
     if parent[x] != x
        parent[x] := findSet(parent[x])
     return parent[x]
```