Cairo University
Faculty of Computers and Artificial Intelligence
Computer Science Department

Advanced Data Structures

Disjoint Sets

Dr. Amin Allam

[For more details, refer to "Introduction to Algorithms" by *Thomas Cormen*, et al.]

1 Disjoint sets

A *disjoint sets* data structure contains a number of *disjoint sets* where each *set* contains at least one *element*. *Sets* can be combined such that different *sets* have different *identifiers* and each *element* is a member of exactly one *set*.

A *disjoint sets* data structure is usually initialized by creating n *disjoint sets* where each *set* contains exactly one *element* (*singleton* sets). The *identifiers* of these *elements* range from 0 to $n-1$. The initial *identifier* of each *set* equals to the *identifier* of its *element*. *Set identifiers* can change during execution but *element identifiers* do not change.

A *disjoint sets* data structure allows two operations: $Find(a)$ and $Union(a, b)$.
$Find(a)$ returns the *set identifier* of the *set* containing the *element* whose *identifier* is a.
$Union(a, b)$ unions the two *sets* containing the two *elements* whose *identifiers* are a and b.

To implement a *disjoint sets*: Each *set* is represented by a tree, where each tree node stores the *identifier* of an *element* that belongs to this *set*. Each tree node also contains a link to its parent. Links to children are not required. The *set identifier* equals to the *element identifier* stored at the *root*. An array of integers $p[]$ where $p[i]$ is the index of the parent of *element* i is enough to represent all these trees.

According to the above implementation, $Find(a)$ is implemented by following the chain of parents starting from *element* a until we reach the *root* of this tree, then returning the *set identifier* which equals to the *element identifier* at the *root*.

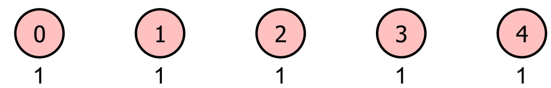
$Union(a, b)$ can be implemented by calling $sa = Find(a)$, $sb = Find(b)$, then letting sa to be the parent of sb or vice versa. That is, by letting the *root* of the tree containing a to be parent of the *root* of the tree containing b. Thus, sa becomes the *set identifier* of all *elements* in the two *sets* making it actually one *set*. This procedure is correct but the height of some trees may grow up to $O(n)$ causing $Find()$ calls to be very inefficient.

Fortunately, we can restrict the height of all trees to $O(\log n)$ if we implement $Union(a, b)$ exactly as above with a minor modification: Let the *root* of the tree containing *more elements* to be the parent of the *root* of the tree containing *less elements*. This is called the *weight rule*, and it requires storing the number of *elements* of each *set* in its tree *root*.

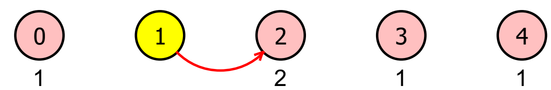
Thus, if the height of all trees is $O(\log n)$, the time complexity of $Find()$ is clearly $O(\log n)$. $Union()$ consists of two calls to $Find()$ plus $O(1)$ additional work. Therefore, $Union()$ time complexity is $O(\log n)$ as well. Now, it remains to prove that the height of any tree is $O(\log n)$ *.

In the following example, a *disjoint sets* structure is initialized by creating 5 *elements* whose *identifiers* are 0 to 4. Initially, there are 5 *sets*, each *set* corresponds to a *one-node tree*, contains exactly one *element*, and has the same *identifier* as the contained *element*. *Root nodes* are coloured red while *non-root nodes* are coloured yellow. The number of *elements* in each *set* is shown below its *tree root*. *Element identifiers* exist inside the circles. The following operations are performed: Union(1, 2), Union(3, 4), Union(0, 1), Union(1, 3). Note that we always link *roots*.

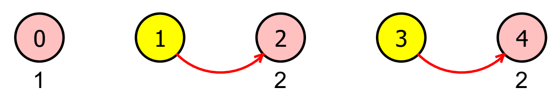
Initial configuration:
Find(0)=0 Find(1)=1 Find(2)=2
Find(3)=3 Find(4)=4



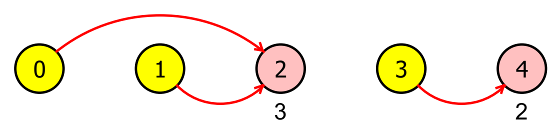
Union(1, 2)
Find(0)=0 Find(1)=2 Find(2)=2
Find(3)=3 Find(4)=4



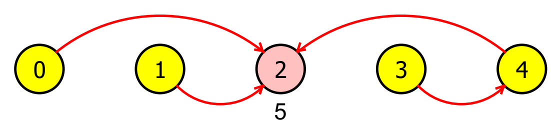
Union(3, 4)
Find(0)=0 Find(1)=2 Find(2)=2
Find(3)=4 Find(4)=4



Union(0, 1)
Find(0)=2 Find(1)=2 Find(2)=2
Find(3)=4 Find(4)=4



Union(1, 3)
Find(0)=2 Find(1)=2 Find(2)=2
Find(3)=2 Find(4)=2



$*$ Starting with *singleton sets* and performing $Union()$ operations using *weight rule*, a tree t containing $N(t)$ nodes has height $H(t) \leq \log_2 N(t)$. (height $=$ distance from *root* to farthest leaf)

Proof: By induction on the number of nodes $N(t)$ of a tree t:
Base step: When $N(t) = 1$, the tree contains one node so its height $H(t) = 0 \leq \log_2 N(t)$.
Induction step: When $N(t) \geq 2$, consider the last union operation performed to result the tree t from two trees a and b where $1 \leq N(a) \leq N(b) < N(t)$:

Since $N(a) \leq N(b)$ and $N(a) + N(b) = N(t)$ thus $N(a) + N(a) \leq N(t)$ so $N(a) \leq N(t)/2$.
Both $N(a)$ and $N(b)$ are less than $N(t)$, so we can apply induction to assume that:
- $H(a) \leq \log_2 N(a) \leq \log_2 N(t)/2 = \log_2 N(t) - \log_2 2 = \log_2 N(t) - 1$.
- $H(b) \leq \log_2 N(b) < \log_2 N(t)$. (because $\log$ is an increasing function)

After performing union based on the *weight rule*, $H(t) = \max(H(a) + 1, H(b))$.
Since $H(a) + 1 \leq \log_2 N(t)$ and $H(b) < \log_2 N(t)$, thus $H(t) \leq \log_2 N(t)$.