5.7 Disjoint Set Structures

Assume we have N objects numbered from 1 to N. We wish to group them is to disjoint sets. At any given time each object is in exactly one set. In each subset designate one member as a label. for example: refer to $\{2, 5, 7, 10\}$ as "set 2"
Initially, the N objects are in N different sets (each set contains one element). Thereafter, we execute a sequence of operations of 2 kinds:

1. Given some object, we "find" which set contains it and returns the label of this set; and
2. Given 2 different labels, we merge the contents of the two corresponding sets, and choose a label for the combined set.

Our problem is to represent this situation efficiently.
One obvious representation is obvious.
Suppose we decide to choose the smallest element as the label. If we declare an array set $[1..N]$, it suffices to put the label of the set corresponding to each object in the appropriate array element. The 2 operations we want to perform can be implemented by the following procedures.

```
function find1(x)      { finds the label of set containing x }
    return set[x]

procedure merge1 (a, b)   { merges set[a] & set[b]; a ≠ b }
    i ← min {a, b}
    j ← max {a, b}
    for k ← 1 to N do
        if set[k] = j then set[k] ← i
```

Now suppose we execute an arbitrary sequence of operations, of types find and merge, starting from the initial situation. We don't know in

which order these operations will occur. There will be "n" of type find and not more than N-1 of type merge. For many applications n is comparable to N.

We assume that consulting or modifying one element of an array counts as an elementary operation. It is clear that find1 takes a constant time, and merge takes a time in $\theta(N)$. The n "find" operations therefore take time in $\theta(n)$, while N-1 "merge" operations take a time in $\theta(N^2)$. If n and N are comparable, the whole sequence of operations takes a time in $\theta(n^2)$. Let's do better than this. Still using a single array, we can represent each set as a rooted tree, where each node contains a single pointer to its parent.

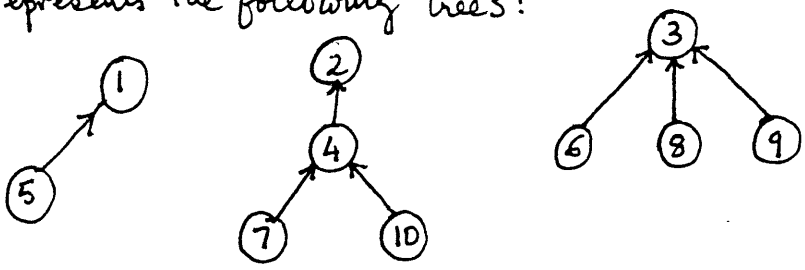
if set[i] = i then i is the root and label of the tree.

if set[i] = j ≠ i then j is the parent of i in some tree.

The array:

1	2	3	2	1	3	4	3	3	4

represents the following trees:



which in turn represent the sets {1,5}, {2,4,7,10}, {3,6,8,9}

To merge 2 sets, we need only to change one single value in the array; on the other hand, it is harder to find the set which an object belongs to.

```
function find2 (x)
    r ← x
    while set[r] ≠ r do r ← set[r]
    return r
```

```
procedure merge2 (a, b)
    if a < b then set[b] ← a
             else set[a] ← b
```

obviously merge takes a time that is constant. The time needed to find is $\theta(N)$ in the worst case. Executing n finds 2 and N-1 merge 2 starting from the initial situation can take time in $\theta(nN)$ like previously $\theta(n^2)$, and thus no gain. The problem obviously is: after k calls of merge 2 we find ourselves confronted by a tree of height k, so each subsequent call on find 2 may take a time proportional to k. To avoid this we must find a way to limit the height of the trees produced.

So far we considered the smallest element of a set to serve as a label. It would be better to arrange matters so it is always the tree whose height is less (lesser) that becomes a subtree of the other. Using this technique:

Suppose we want to merge 2 trees of height h_1 and h_2
The resulting merging tree will be $\max(h_1, h_2)$ if $h_1 \neq h_2$
or $h_1 + 1$ if $h_1 = h_2$

As the following theorem shows, the height of the trees does not grows as rapidly.

Theorem:

Using the technique outlined above, after an arbitrary sequence of merge operations starting from the initial situation, a tree containing k nodes has a height at most $\lfloor \log k \rfloor$.

Proof can be derived easily by induction.

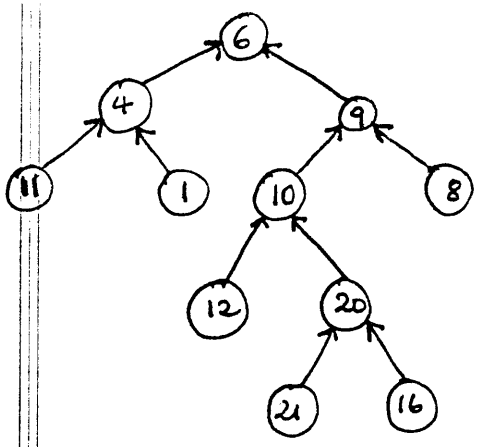
The height of the trees can be maintained in an additional array "height[1..N]" so that height[i] gives the height of i in its current tree. Whenever a is the label of a set, height[a] therefore gives the height of the corresponding tree, and in fact these are the only heights that concern us.
Initially, height of i is set to zero for each i. The procedure find2 is unchanged, but we must modify merge appropriately.

```
procedure merge3(a, b)        {merges sets labelled a & b}
                                  a ≠ b
  if height[a] = height[b]
  then
    height[a] ← height[a] + 1
    set[b] ← a
  else
    if height[a] > height[b]
    then set[b] ← a
    else set[a] ← b
```

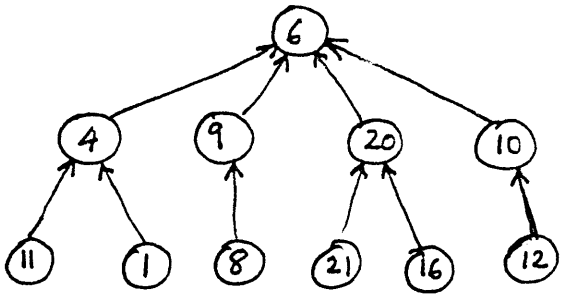
Starting with the initial conditions, the time needed to execute n find2 and $N-1$ merge3 operations is: $\Theta(N + n \log N)$ in the worst case; assuming n and N are comparable, this becomes of the order of $n \log n$.
By modifying find2, we can make the operations even faster. When determining the set that contains x, we traverse the edges of the tree leading up to the root. Once we know the root we traverse the same edges again, this time modifying each node on the way so its pointer now indicates the root directly. This technique is called "path compression".

Example:

Let's execute the operation find(20) on figure (a). The result is in figure (b). Nodes 20, 10, 9 point directly to the root.



(a)



(b)

This technique reduce the height of a tree, and subsequently accelerates "find" operations. On the other hand, this new find traverses the path from the root to the node in question twice, and therefore takes twice as long as before. This technique may not be worthwhile if only a small number of find operations are executed. However if many find operations are executed then after a while (roughly speaking) we may expect that all the nodes involved will be attached directly to the roots of their respective trees, so the subsequent searches take a <u>constant</u> time. A merge operation will perturb the situation only slightly, and not for long, and therefore find and merge operations require a time in $O(1)$. A sequence of n of the former and $N-1$ of the latter (merge) will take nearly $O(n)$ when N and n are comparable.

One small point remains to be cleared. Using compression it is no longer true that the height of a tree whose root is a is given by height[a]. To avoid confusion we call this value the "rank" of the tree

The find function is now as follows.

```
function  find3 (x)
    r ← x
    while set[r] ≠ r  do r ← set[r]
    {r is the root of the tree}
    i ← x
    while i ≠ r  do
        j ← set[i]
        set[i] ← r
        i ← j
    return r
```

From now on, when we use this combination of 2 arrays and of procedures find3 and merge3 to deal with disjoint sets of objects, we say we are using a <u>disjoint set structure</u>.

It is a little hard to analyze the time needed for an arbitrary sequence of "find" and "merge" operations when path compression is used.

Let's digress for a moment and introduce the following 2 new functions $A(i,j)$ and $\alpha(i,j)$.

$A(i,j)$ is a slight variant of Ackermann's function

$$A(i,j) = \begin{cases} 2j & \text{if } i = 0 \\ 2^j & \text{if } j = 1 \\ A(i-1, A(i,j-1)) & \text{otherwise} \end{cases}$$

From the definition it appears that $A(1,j) = 2^j \ \forall j$

and $A(2,j) = 2^{\cdots^2}\}j$ twos.

$A(3,1) = 2$, $A(3,2) = 4$, $A(3,3) = 65536 = 2^{16}$

$A(3,4) = 2^{\cdots^2}\}65536$ twos and so on.

This function grows extremely fast.

Now the function $\alpha(i,j)$ is defined as a kind of inverse of $A(i,j)$.

$$\alpha(i,j) = \min\{k \mid k \geq 1 \text{ and } A(k, 4\lceil i/j \rceil) > \log_2 j\}$$

Notice $\alpha(i,j) > 3$ only when

$$\log_2 j \geq A(3,4) = 2^{\cdot^{\cdot^{\cdot^{2}}}} \Big\} 2^{16} \text{ twos.}$$

Thus for all except "astronomical" values of j, $\alpha(i,j) \leq 3$

With a universe of N objects and the given initial situation, consider an arbitrary sequence of n calls of find3 and $m \leq N-1$ calls of merge3. Let $c = n+m$, using the function above, it was proven (Tarjan) that such a sequence can be executed in a time in $\theta(c\alpha(c,N))$ in the worst case. For all practical purposes suppose $\alpha(c,N) \leq 3$, thus the time is in the order of c.