CHAPTER 8

THE DISJOINT SET ADT

§1 Equivalence Relations

【Definition】 A ***relation R*** is defined on a set ***S*** if for every pair of elements (a, b), $a, b \in S$, $a\ R\ b$ is either true or false. If $a\ R\ b$ is true, then we say that **a is related to b**.

【Definition】 A relation, ~, over a set, ***S***, is said to be an ***equivalence relation*** over ***S*** iff it is **symmetric**, **reflexive**, and **transitive** over ***S***.

【Definition】 Two members x and y of a set ***S*** are said to be in the same ***equivalence class*** iff $x \sim y$.

§2 The Dynamic Equivalence Problem

Given an equivalence relation ~, decide for any *a* and *b* if *a* ~ *b*.

〖Example〗 Given *S* = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 } and 9 relations: 12≡4, 3≡1, 6≡10, 8≡9, 7≡4, 6≡8, 3≡5, 2≡11, 11≡12.

The equivalence classes are { 2, 4, 7, 11, 12 }, { 1, 3, 5 }, { 6, 8, 9, 10 }

Algorithm: (Union / Find)

```
{   /* step 1: read the relations in */
    Initialize N disjoint sets;
    while ( read in a ~ b ) {
        if ( ! (Find(a) == Find(b)) )
            Union the two sets;
    } /* end-while */
    /* step 2: decide if a ~ b */
    while ( read in a and b )
        if ( Find(a) == Find(b) )   output( true );
        else   output( false );
}
```

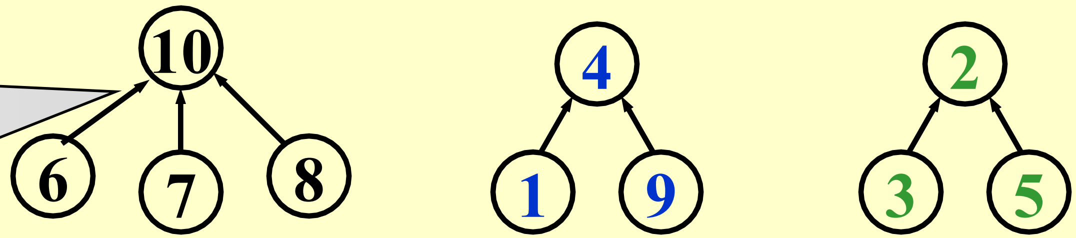
Dynamic (on-line)

- **Elements** of the sets: $1, 2, 3, \ldots, N$

- **Sets** : $S_1, S_2, \ldots \ldots$ and $S_i \cap S_j = \phi$ (if $i \neq j$) —— disjoint

〖Example〗 $S_1 = \{ 6, 7, 8, 10 \}, S_2 = \{ 1, 4, 9 \}, S_3 = \{ 2, 3, 5 \}$

Note:
Pointers are
from children
to parents



A possible forest representation of these sets

- **Operations** :

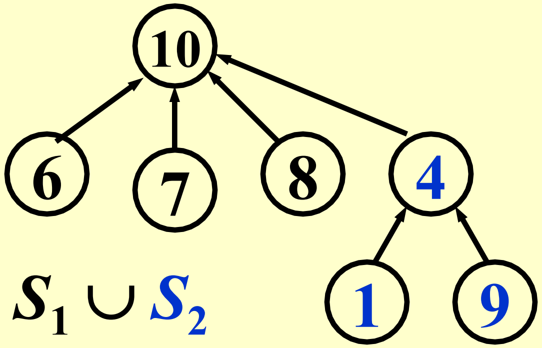
(1) Union(i, j) ::= Replace S_i and S_j by $S = S_i \cup S_j$

(2) Find(i) ::= Find the set S_k which contains the element i.

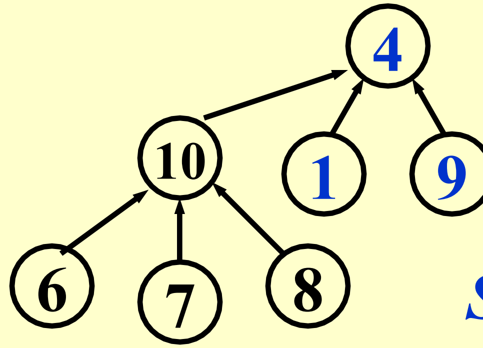
§3 Basic Data Structure

❖ Union (i, j)

Idea: Make S_i a subtree of S_j , or vice versa. That is, we can set the parent pointer of one of the roots to the other root.

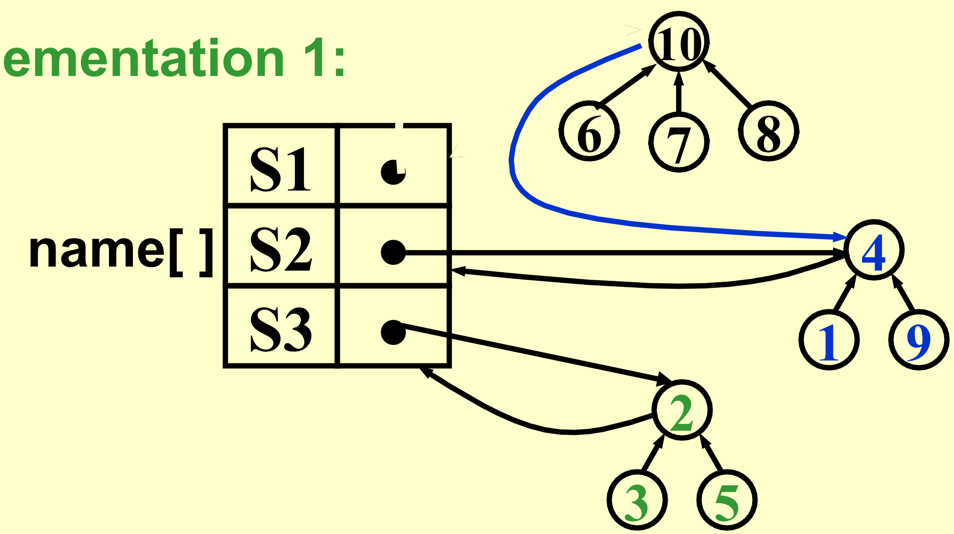


$S_1 \cup S_2$



$S_2 \cup S_1$

Implementation 1:



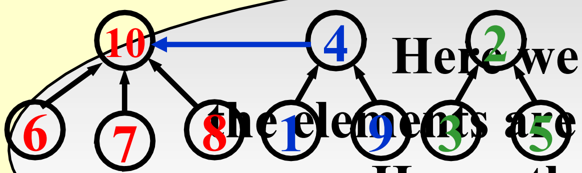
$S_2 \cup S_1$

$\Rightarrow \; S_2$

Implementation 2: S [element] = the element's parent.

Note: S [root] = 0 and set name = root index.

〖Example〗 The array representation of the three sets is



Here we use the fact that the elements are numbered from 1 to N. Hence they can be used as indices of an arr

S

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
4	0	2	10	2	10	10	10	4	0

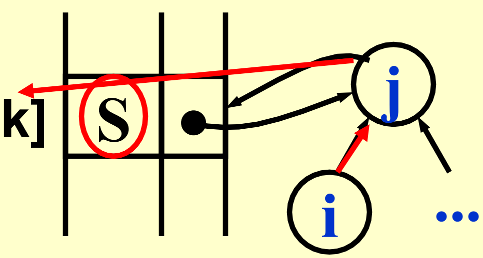
($S_1 \cup S_2 \Rightarrow S_1$) ⇔ S [4] = 10

```
void  SetUnion ( DisjSet S,
                 SetType Rt1,
                 SetType Rt2 )
{   S [ Rt2 ] = Rt1 ;    }
```

❖ Find (*i*)

Implementation 1:

name[k]



find (i) = 'S'

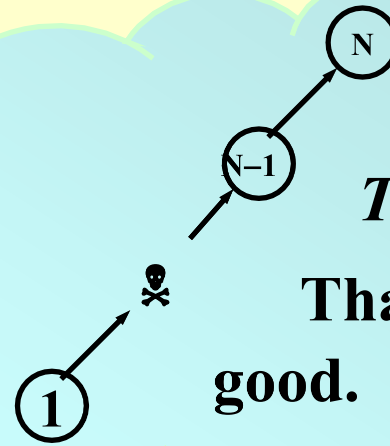
Implementation 2:

```
SetType  Find ( ElementType X,
                DisjSet S )
{   for ( ; S[X] > 0; X = S[X] )   ;
    return  X ;
}
```

❖ Analysis

Practically speaking, union and ... rmanc...

Sure. Try this one:
union(2, 1), find(1);
union(3, 2), find(1);
..., ... ;
union(N, N –1), find(1).



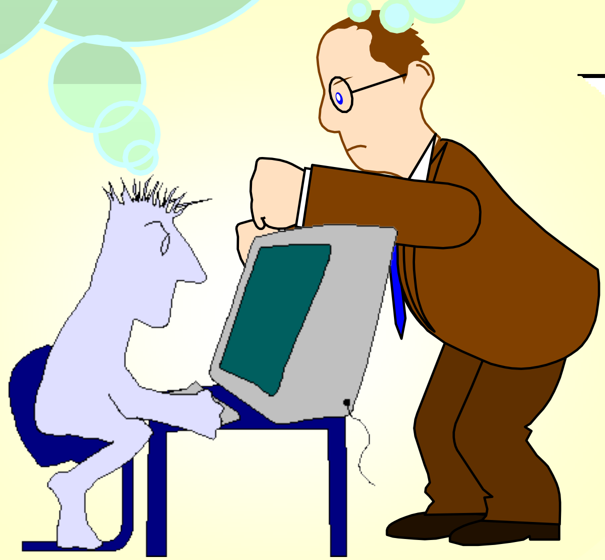
$T = \Theta(N^2)$!

That's not good.

2 ... val ... 7, 11,
12 }, { 1,

Algorithm usin...

{ Initialize S...
  for (k = 1 ... ≡ j */
    if (Find(...
      SetUni...
  }
}



§4 Smart Union Algorithms

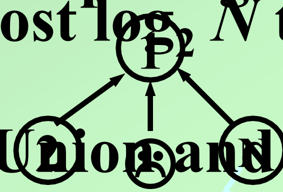
❖ **Union-by-Size** -- Always change the smaller tree

S [Root] = – size; /* initialized to be –1 */

【Lemma】Let T be a tree created by union-by-size with N nodes, then

$$height(T) \le \lfloor \log_2 N \rfloor + 1$$

Now $T = O(N)$ for the worst case example I gave.



Proof: By induction. (Each element can have its set name changed at most $\log_2 N$ times.)

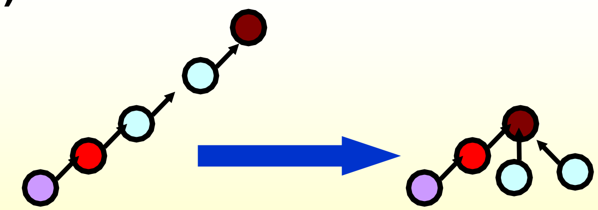
Time complexity of N Union and M Find operations is now $O(N + M \log_2 N)$.

❖ **Union-by-Height** -- Always change the shallow tree

Please read Figure 8.13 on p.273 for detailed implementation.

§5 Path Compression

```
SetType  Find ( ElementType  X, DisjSet  S )
{
    if ( S[ X ] <= 0 )    return  X;
    else    return  S[ X ] = Find( S[ X ], S );
}
```



```
SetType  Find ( ElementType  X, DisjSet  S )
{   ElementType  root,  trail,  lead;
    for ( root = X; S[ root ] > 0; root = S[ r
        ;  /* find the root */
    for ( trail = X; trail != root; trail = lead ) {
        lead = S[ trail ] ;
        S[ trail ] = root ;
    }  /* collapsing */
    return  root ;
}
```

Slower for a single find, but faster for a sequence of find operations.

Note: Not compatible with union-by-height since it changes the heights. Just take "height" as an estimated *rank*.

§6 Worst Case for Union-by-Rank and Path Compression

【Lemma (Tarjan)】 Let $T(M, N)$ be the maximum time required to process an intermixed sequence of $M \geq N$ finds and $N - 1$ unions. Then:

$$k_1 M \alpha(M, N) \leq T(M, N) \leq k_2 M \alpha(M, N)$$

for some positive constants k_1 and k_2.

log* 2^{65536} = 5
since
logloglogloglog
(2^{65536})= 1

☞ **Ackermann's Function and $\alpha(M, N)$**

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

http://mathworld.wolfram.com/AckermannFunction.html

$\alpha(M, N) = \min\{\, i \geq 1 \mid A(i, \lfloor M/N \rfloor) > \log N \,\}$ $\leq O(\log^* N) \leq 4$

$\log^* N$ (inverse Ackermann function)

= # of times the logarithm is applied to N until the result ≤ 1.