小白专场：
File Transfer

浙江大学 陈 越

浙江大学计算机科学与技术学院

集合的简化表示

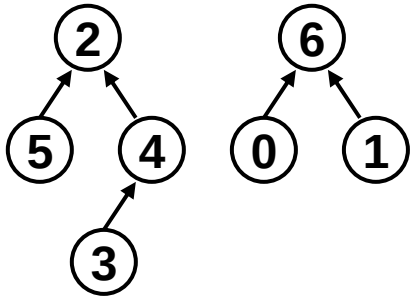
```
typedef struct {
    ElementType Data;
    int Parent;
} SetType;
```

任何有限集合的（N个）元素都可以被一一映射为整数 0 ~ N–1

可以理解为数组的下标。可以简化为数组里的每个元素直接用下标来表示

```
int Find( SetType S[], ElementType X )
{  int i;
   for ( i=0; i<MaxSize && S[i].Data!=X; i++ ) ;
   if ( i>=MaxSize ) return -1;
   for( ; S[i].Parent>=0; i=S[i].Parent ) ;
   return i;
}
```

☹ 有点慢



下标	[0]	[1]	[2]	[3]	[4]	[5]	[6]
S	6	6	-1	4	2	2	-1

下标表示这个数，s里面的值表示这个数的父节点。则有几个-1，就有几个独立的集合

集合的简化表示

```
typedef int ElementType; /*默认元素可以用非负整数表示*/
typedef int SetName;     /*默认用根结点的下标作为集合名称*/
typedef ElementType SetType[MaxSize];
```

```
SetName Find( SetType S, ElementType X )
{ /* 默认集合元素全部初始化为-1 */
    for ( ; S[X]>=0; X=S[X] ) ;
    return X;
}
```

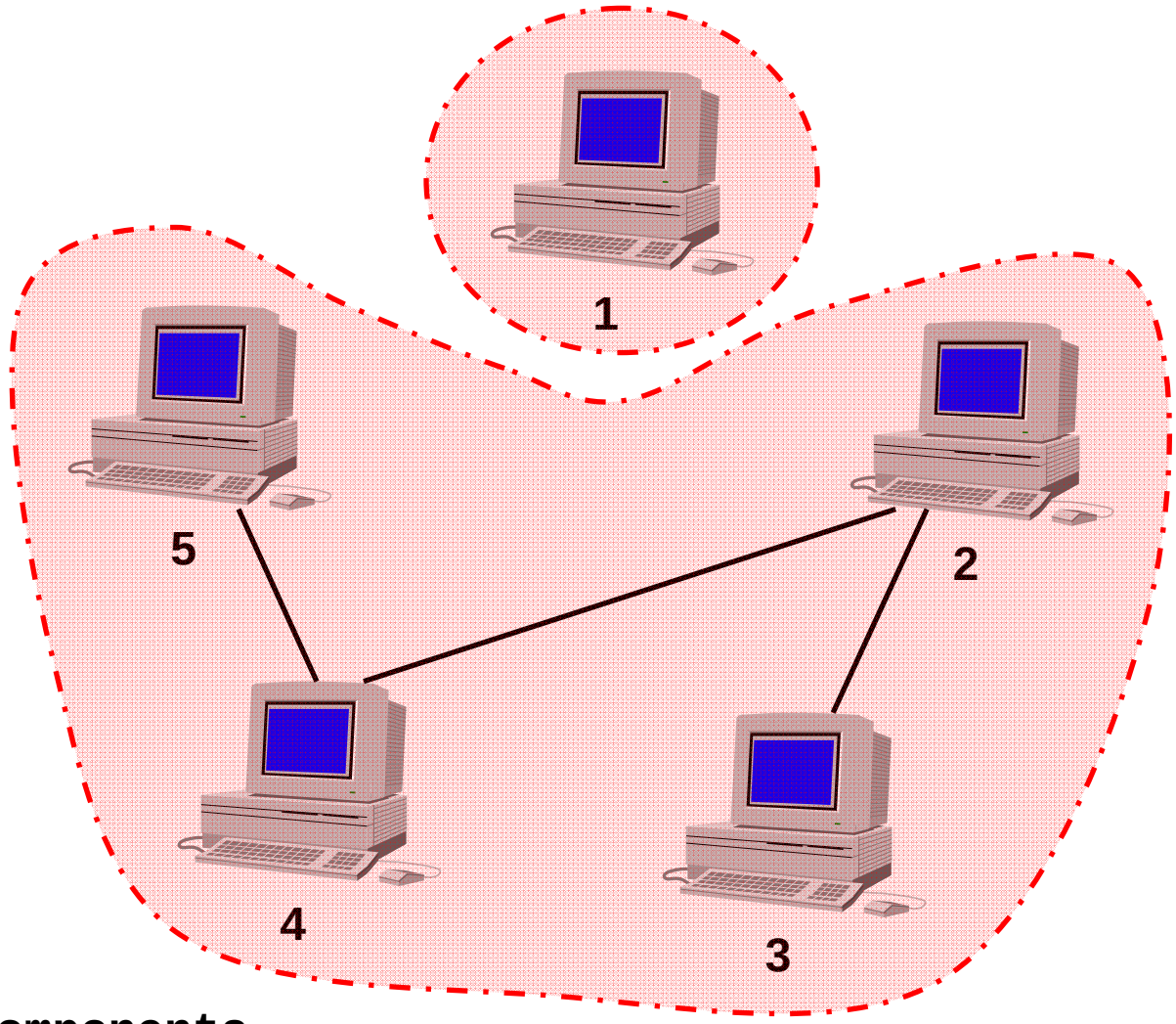
此时X就直接代表的是数组的下标

```
void Union( SetType S, SetName Root1, SetName Root2 )
{ /* 这里默认Root1和Root2是不同集合的根结点 */
    S[Root2] = Root1;
}
```

题意理解

输入样例1:

```
5
C 3 2  ☞ no
I 3 2
C 1 5  ☞ no
I 4 5
I 2 4
C 3 5  ☞ yes
S
```

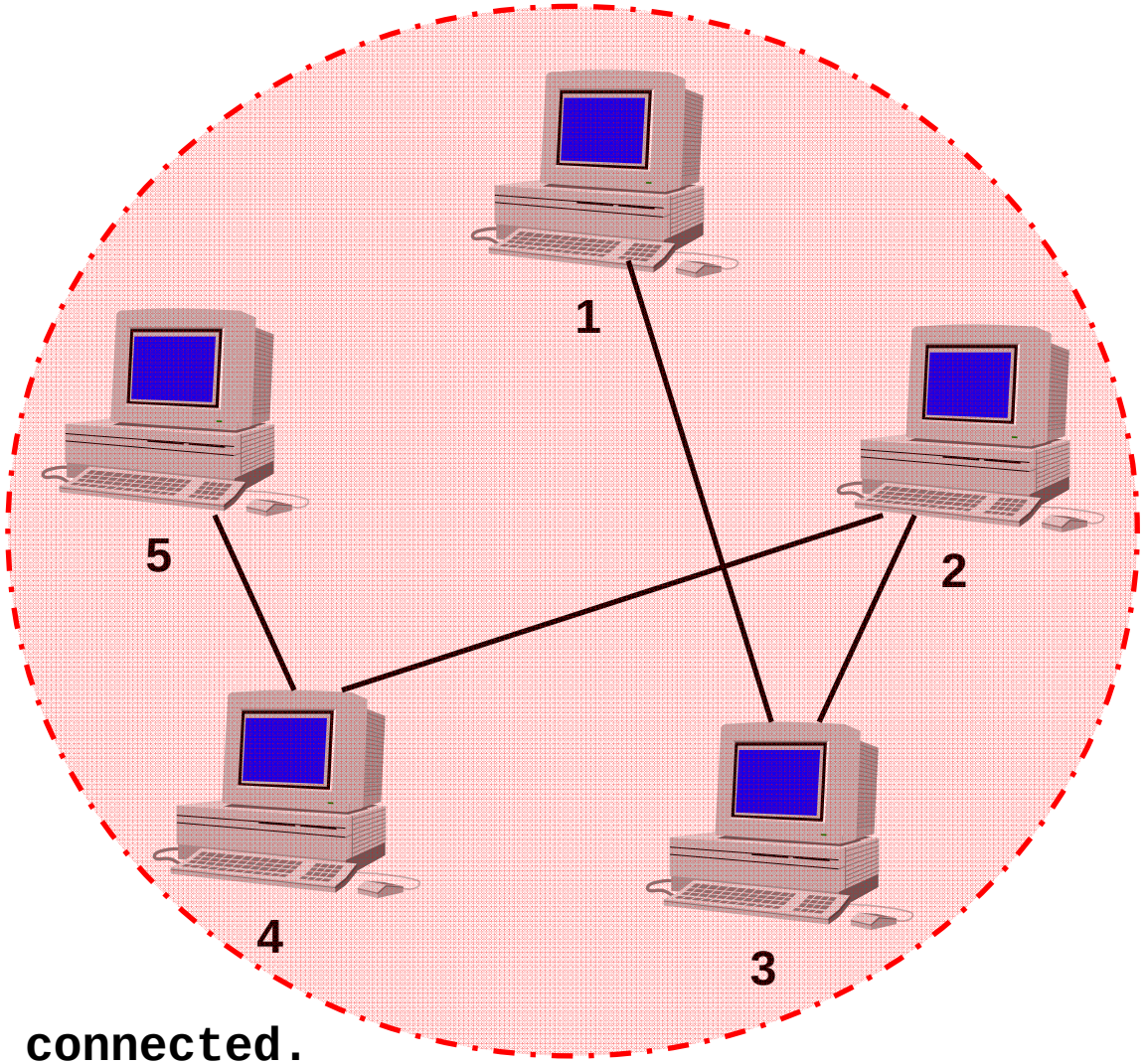



```
There are 2 components.
```

题意理解

输入样例**2:**

```
5
C 3 2  ☞ no
I 3 2
C 1 5  ☞ no
I 4 5
I 2 4
C 3 5  ☞ yes
I 1 3
C 1 5  ☞ yes
S
```




```
The network is connected.
```

程序框架搭建

```
int main()
{
    初始化集合;
    do {
        读入一条指令;
        处理指令;
    } while (没结束);
    return 0;
}
```

```
int main()
{ SetType S;
  int n;
  char in;
  scanf("%d\n", &n);
  Initialization( S, n );
  do {
    scanf("%c", &in);
    switch (in) {
     case 'I': Input_connection( S ); break;
     case 'C': Check_connection( S ); break;
     case 'S': Check_network( S, n ); break;
    }
  } while ( in != 'S');
  return 0;
}
```

Union(Find)

Find

数集合的根

程序框架搭建

```
void Input_connection( SetType S )
{ ElementType u, v;
  SetName Root1, Root2;
  scanf("%d %d\n", &u, &v);
  Root1 = Find(S, u-1);
  Root2 = Find(S, v-1);
  if ( Root1 != Root2 )
    Union( S, Root1, Root2 );
}
```

```
void Check_connection( SetType S )
{ ElementType u, v;
  SetName Root1, Root2;
  scanf("%d %d\n", &u, &v);
  Root1 = Find(S, u-1);
  Root2 = Find(S, v-1);
  if ( Root1 == Root2 )
        printf("yes\n");
  else  printf("no\n");
}
```

```
void Check_network( SetType S, int n )
{ int i, counter = 0;
  for (i=0; i<n; i++) {
    if ( S[i] < 0 )  counter++;
  }
  if ( counter == 1 )
    printf("The network is connected.\n");
  else
    printf("There are %d components.\n", counter);
}
```

TSSN的实现

```
SetName Find( SetType S, ElementType X )
{ /* 默认集合元素全部初始化为-1 */
    for ( ; S[X]>=0; X=S[X] ) ;
    return X;
}
```

```
void Union( SetType S, SetName Root1, SetName Root2 )
{ /* 这里默认Root1和Root2是不同集合的根结点 */
    S[Root1] = Root2;
}
```

测试点	结果
测试点1	答案正确
测试点2	答案正确
测试点3	答案正确
测试点4	答案正确
测试点5	答案正确
测试点6	运行超时
测试点7	答案正确

测试点	结果
测试点1	答案正确
测试点2	答案正确
测试点3	答案正确
测试点4	答案正确
测试点5	运行超时
测试点6	答案正确
测试点7	答案正确

TSSN的实现

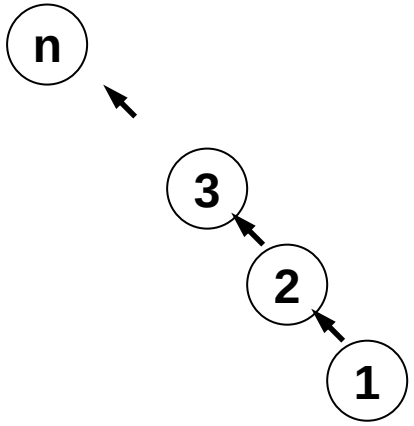
```
int Find( SetType S[], ElementType X )
{  int i;
   for ( i=0; i<MaxSize && S[i].Data!=X; i++ ) ;
   if ( i>=MaxSize ) return -1;
   for( ; S[i].Parent>=0; i=S[i].Parent ) ;
   return i;
}
```

测试点	结果
测试点1	答案正确
测试点2	答案正确
测试点3	答案正确
测试点4	答案正确
测试点5	运行超时
测试点6	运行超时
测试点7	运行超时

```
void Union( SetType S[], ElementType X1, ElementType X2 )
{  int Root1, Root2;
   Root1 = Find(S, X1);
   Root2 = Find(S, X2);
   if (Root1!=Root2) S[Root2].Parent = Root1;
}
```

按秩归并

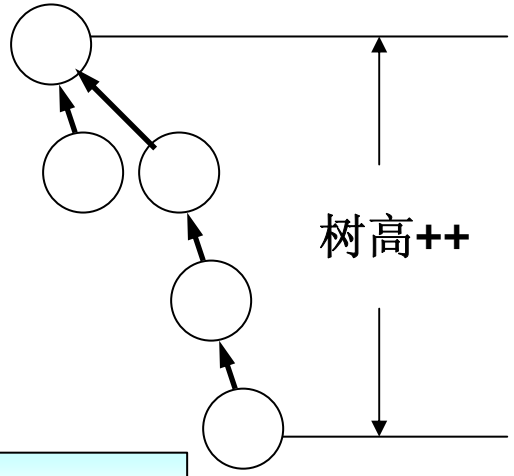
- 为什么需要按秩归并？
  - `Union(Find(2), Find(1))`
  - `Union(Find(3), Find(1))`
  - ……
  - `Union(Find(n), Find(1))`



$$T(n) = O(n^2)$$

按秩归并

- 为什么树会越来越高？
  - 应该把矮树贴到高树上！
- 树的高度存哪里？



S[Root] = -树高;

仍然初始化为-1

```
if ( Root2高度 > Root1高度 )
   S[Root1] = Root2;
else {
   if ( 两者等高 ) 树高++;
   S[Root2] = Root1;
}
```

```
if ( S[Root2] < S[Root1] )
   S[Root1] = Root2;
else {
   if ( S[Root1]==S[Root2] ) S[Root1]--;
   S[Root2] = Root1;
}
```

按秩归并

最坏情况下的树高 = $O(\log N)$

- 另一种做法：比规模
  - 把小树贴到大树上　S[Root] = -元素个数;

```
void Union( SetType S, SetName Root1, SetName Root2 )
{  if ( S[Root2]<S[Root1] ){
       S[Root2] += S[Root1];
       S[Root1] = Root2;
   }
   else {
       S[Root1] += S[Root2];
       S[Root2] = Root1;
   }
}
```

两种方法统称“按秩归并”

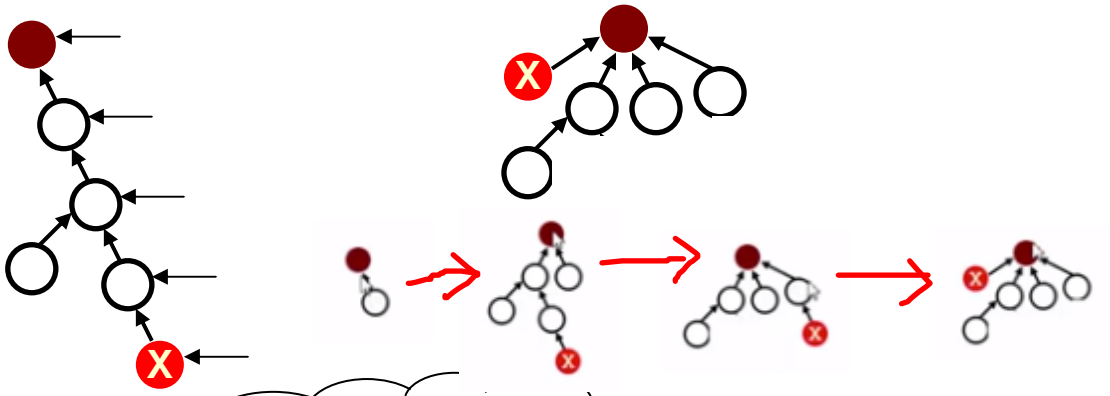
路径压缩

```
SetName Find( SetType S, ElementType X )
{  if ( S[X] < 0 ) /* 找到集合的根 */
      return X;
   else
      return S[X] = Find( S, S[X] );
}
```

这个递归函数实际上是一个尾递归。然而尾递归是非常容易被转换为循环的。
因此编译器会直接把他变成循环。

先找到根；
把根变成 X 的父结点；
再返回根。

每次都更新，把子节点的父节点更新为跟节点



时间复杂度

【引理 (Tarjan)】令 $T(M, N)$ 为交错执行 $M \geq N$ 次带路径压缩的查找和 $N-1$ 次按秩归并的最坏情况时间。则存在正常数 k_1 和 k_2 使得:

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

Ackermann 函数和 $\alpha(M, N)$

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

$A(4, 1) = 2^{65536}$ 36

$\log^* 2^{65536} = 5$
因为
$\log\log\log\log\log 2^{65536} = 1$

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

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

$\log^* N$ (Ackermann 反函数) = 对 N 求对数直到结果≤ 1的次数