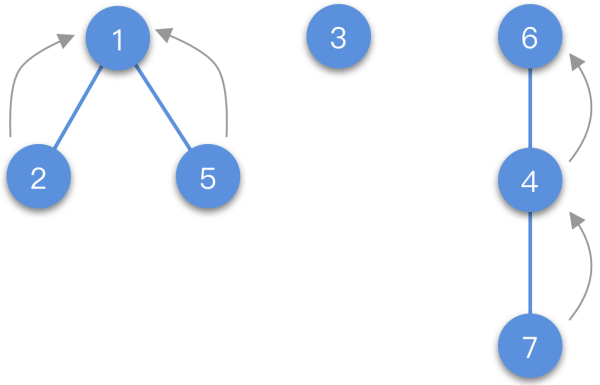
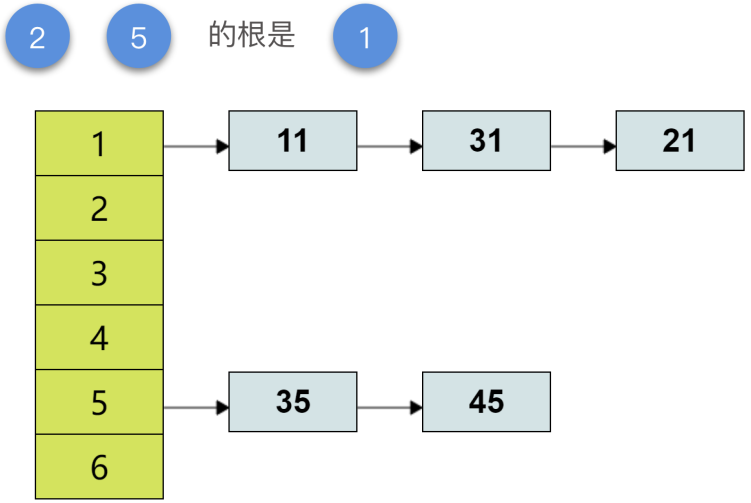


散列表、并查集

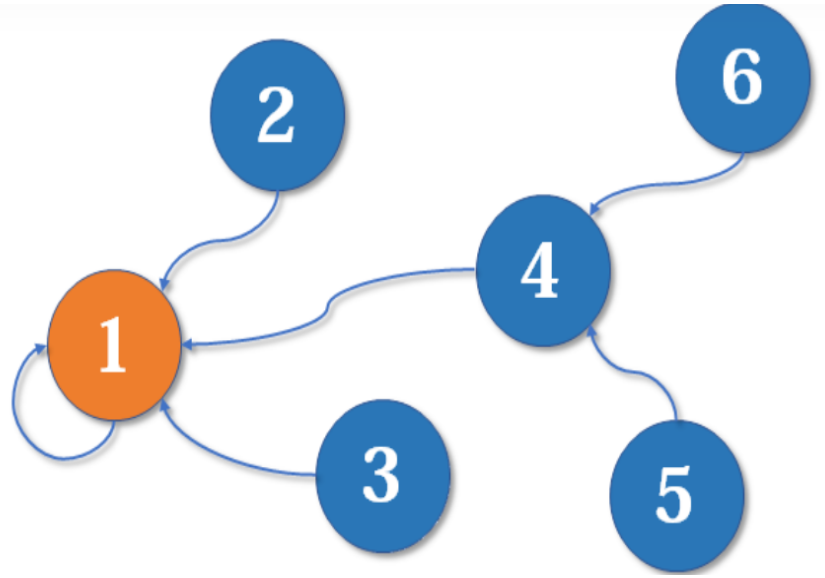


学培部分享
宣青卓
无16

并查集 → 课上遗留 → 哈希表 → 冲突

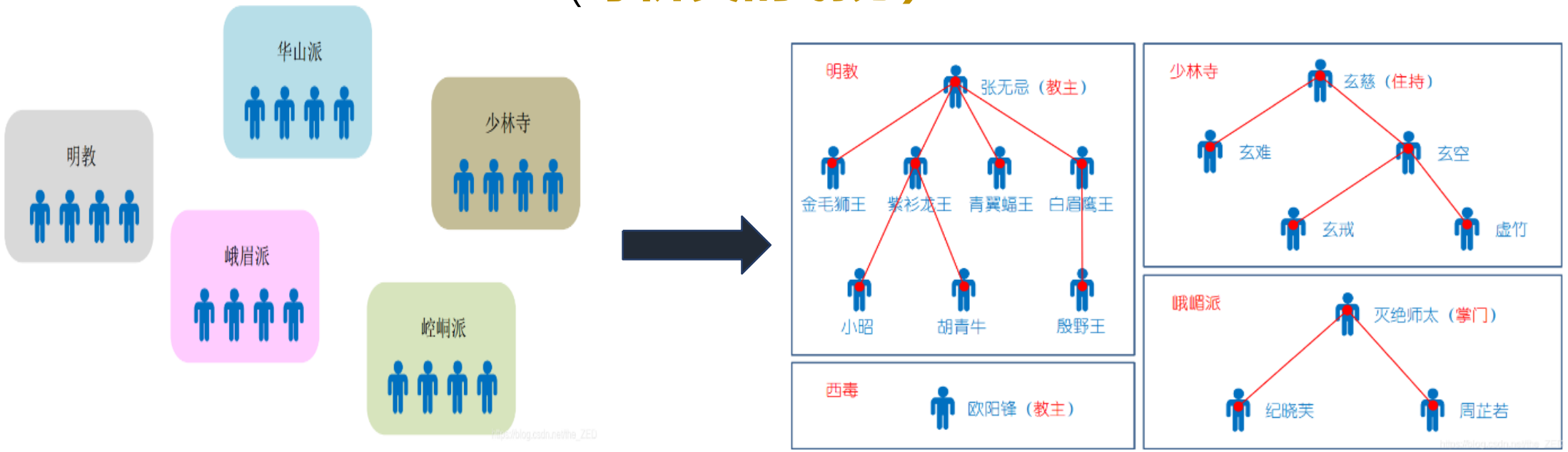
并查集

- 并查集被认为是最简洁而优雅的数据结构之一，主要用于解决一些**元素分组**的问题。它管理一系列**不相交的集合**，并支持两种操作:
- **合并**（Union）：把两个不相交的集合合并为一个集合。
- **查询**（Find）：查询两个元素是否在同一个集合中。



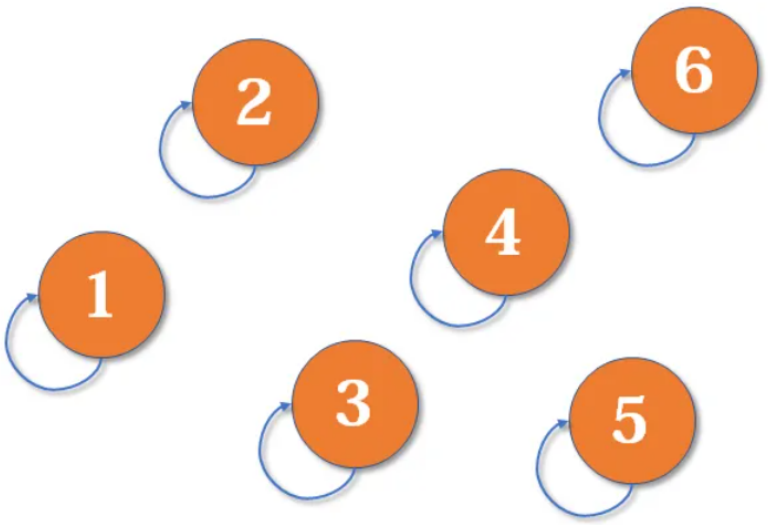
问题引入：江湖混战——如何辨别？

(等价类的划分)



并查集 → 课上遗留 → 哈希表 → 冲突

并查集——INITIALIZE

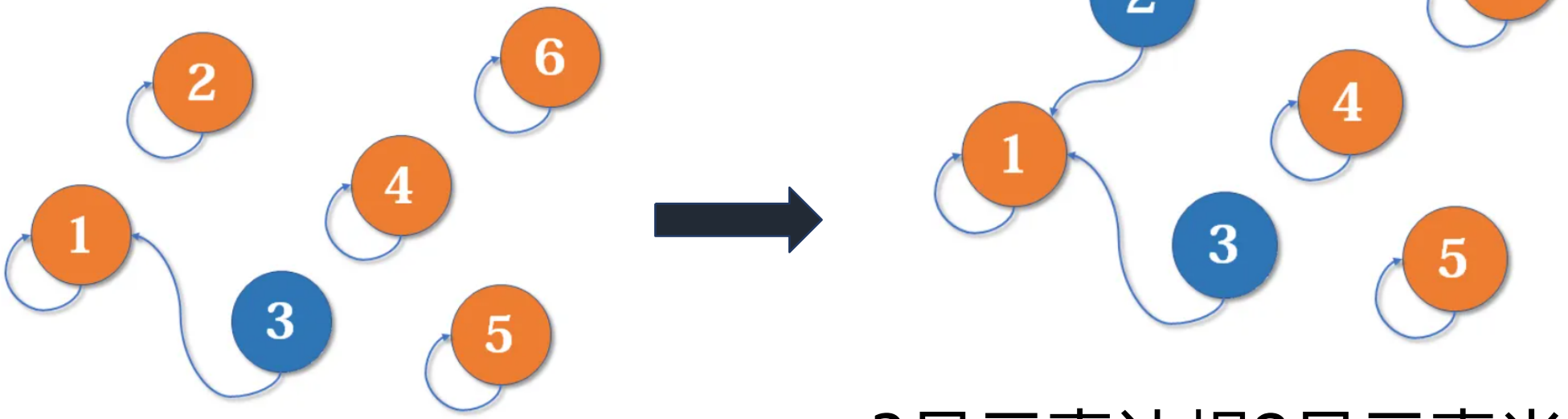


- 最开始，各位大侠们各自为战，则此时，他们就是自己的帮主——**即自己是自己的根节点**

```
int U_F[MAX_SIZE];

void initialize_U_F(int a[],int size
)
{
      for (int i = 1; i <= size; i++)
      a[i] = i;
}
```

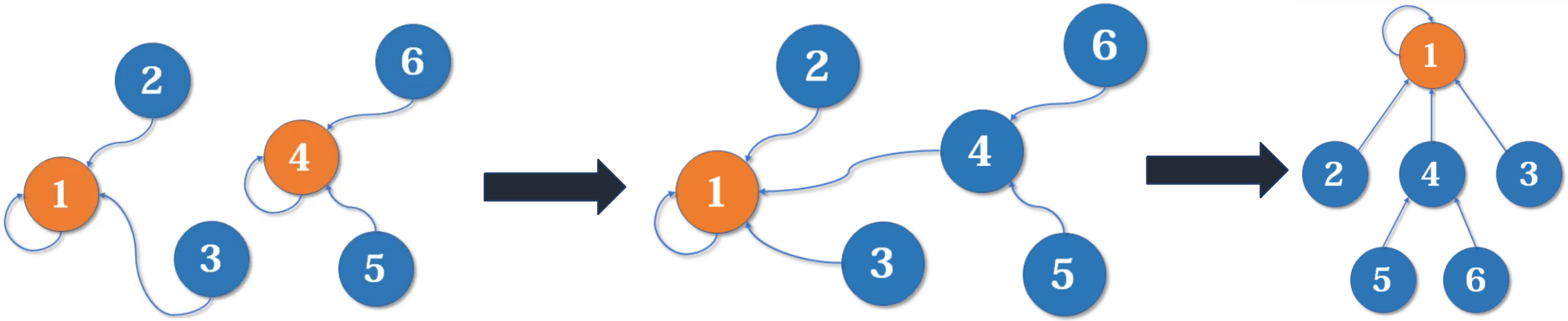
用数组元素作为根节点下标（因为并查集多用数组作为载体）

并查集——UNION



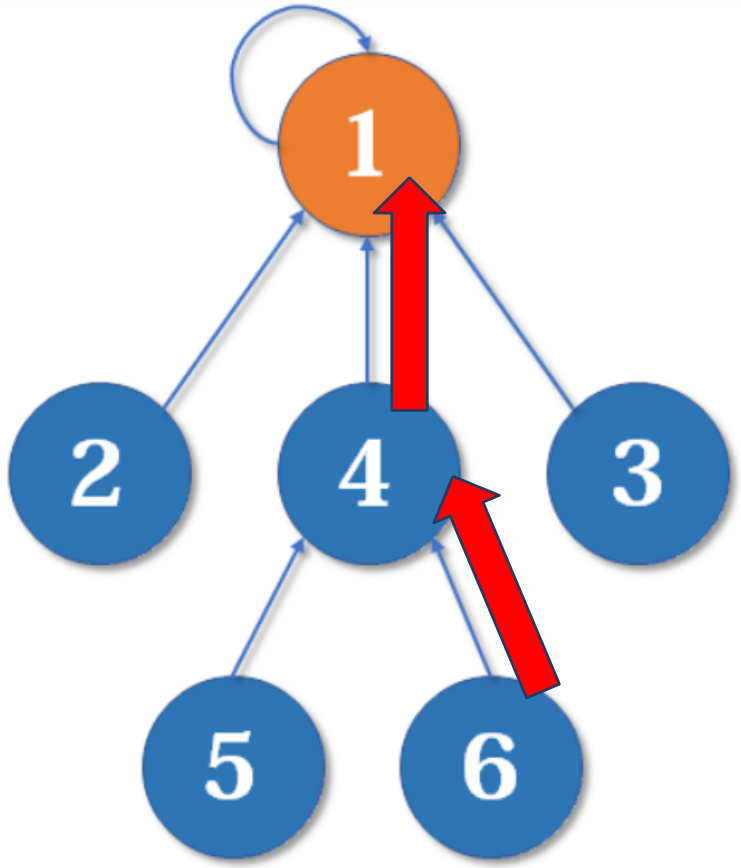
- 3号元素认1号元素当大哥
- 3号元素认想2号元素当大哥,但是1号元素是2的大哥，于是2认1做大哥（合并）

并查集——UNION



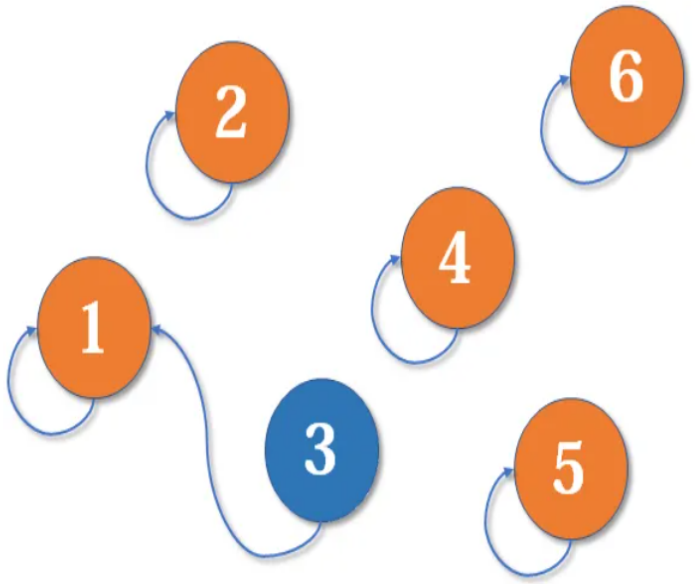
- 同理，4、5、6进行了帮派合并
- 4号元素又认1号为大哥，全体合并完成

并查集——FIND




```
int FIND(int a[], int x)//查找元素x的根;
{
      if (a[x] == x)
         return x;
      else return FIND(a, a[x]);//递归寻找;
}
```

并查集——merge函数 建立在FIND上）




```
void merge(int a[], int i, int j)
//把i和i的祖先并到j的根；
{
        a[FIND(a, i)] = FIND(a,j);
//找到i的根之后，i的根认j的根为“根”；
}
```

在一个包含n个元素的并查集中，进行m次查找或合并操作

最坏情况下所需的时间为 $O(m\alpha(n))$

- $\alpha(n)$是一个随着n的增大增长非常缓慢的函数
- $\alpha(n)$，对于n =宇宙中原子数之和(10^{80})，$\alpha(n) \leq 4$

并查集——路径压缩

最简单的并查集效率是很低下的！

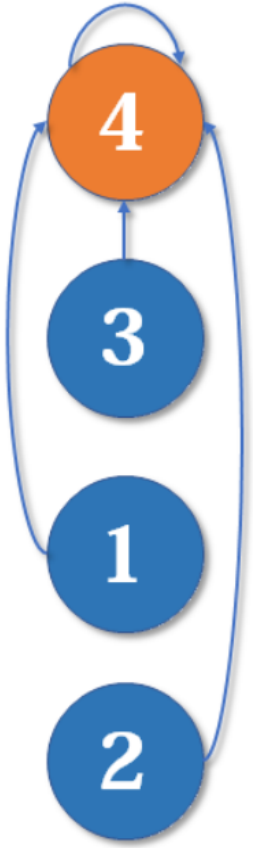
Merge（a[ ],2,3）

Merge（a[ ],2,4）

可能会形成一条长长的链，随着链越来越长，我们想要从底部找到根节点会变得越来越难。

怎么解决呢？我们可以使用路径压缩的方法。

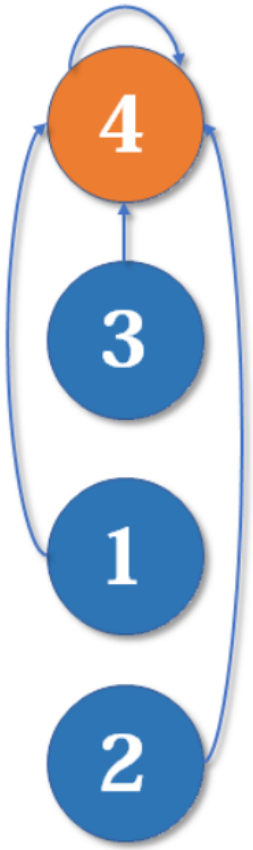
并查集——路径压缩



只要我们修改查询函数，即在查询的过程中，把沿途的每个节点的父节点都设为并查集的根节点即可。下一次再查询时，我们就可以省很多事。这用递归的写法很容易实现：压缩之后，查找复杂度O（1）

```
int FIND_AND_COMPRESSION(int a[],int x)
{
        if (x == a[x])
            return x;
        else
  {a[x] = FIND_AND_COMPRESSION(a,a[x]);
            return a[x];//层层返回根节点;
}
}
```

并查集——路径压缩




```
int FIND_AND_COMPRESSION(int a[],int x)
{
      if (x == a[x])
          return x;
      else
 {a[x] = FIND_AND_COMPRESSION(a,a[x]);
         return a[x];//层层返回根节点;
}
}
```

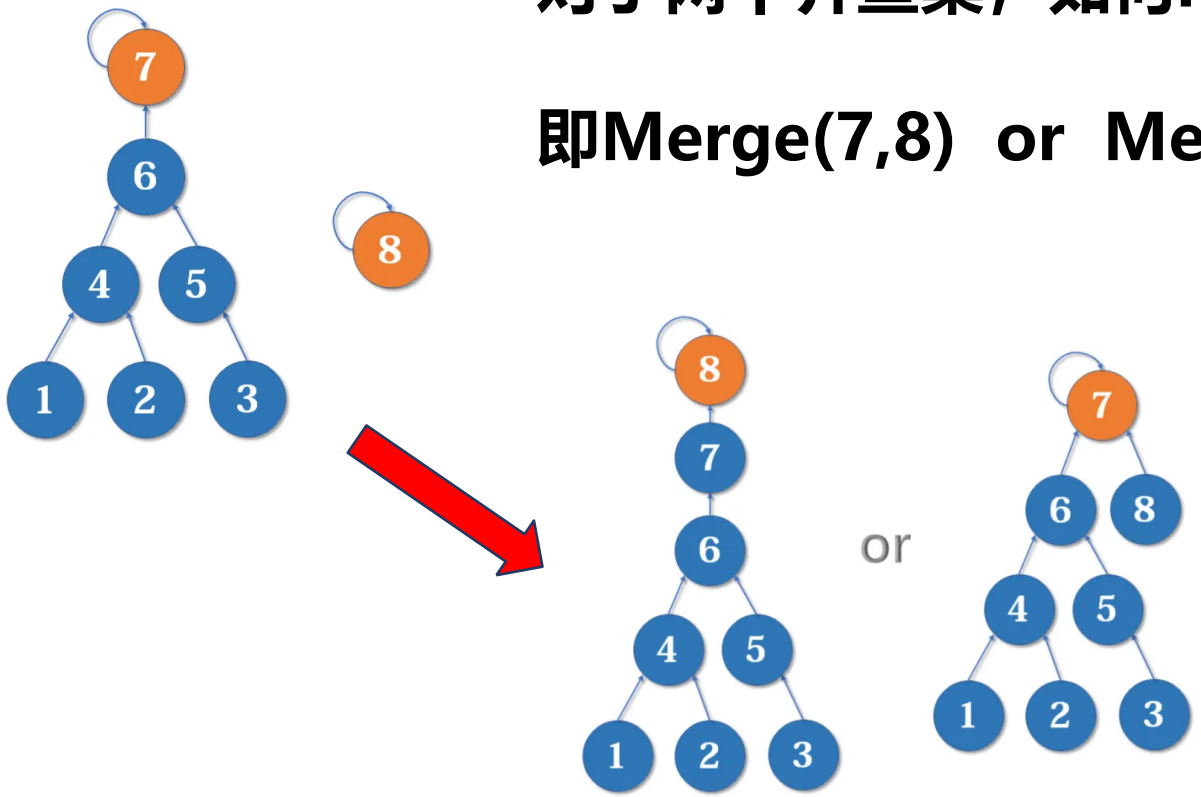
上一页代码写成如下一行:

```
int FIND_AND__COMPRESSION(int a[], int x)
{
   return x == a[x] ? x : (a[x] = FIND_AND__COMPRESSION(a,a[x]));
}
```

拓展——路径压缩_按秩压缩*

对于两个并查集，如何merge复杂度最小?

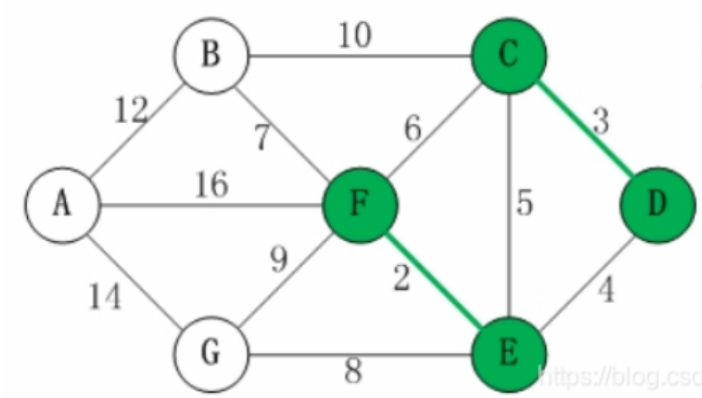
即Merge(7,8) or Merge（8,7）?



如果把7的父节点设为8，会使**深度**（最长链的长度）加深，并查集中每个元素到根节点的距离都变长了，我们寻找根节点的路径也就会相应变长。虽然我们有路径压缩，但路径压缩也是会消耗时间的。而把8的父节点设为7，则不会有这个问题——小规模并到大规模上，所谓的按秩（每个节点自树的深度）压缩；

并查集——Kruskal应用

【Kruskal】
1.将图中所有的边长权值按从小到大的顺序排列，从小的开始选取边：
①如果发现连上会形成环， 放弃这条边，继续寻找下一个边
②如果发现连上不会成环，连接这条边，并且将这条边两侧的点放入到“已连接”集合中，继续寻找下一个边



如何实现避圈？——并查集

并查集——Kruskal应用

在DLL老师ppt中对Kruskal避圈有如下描述：

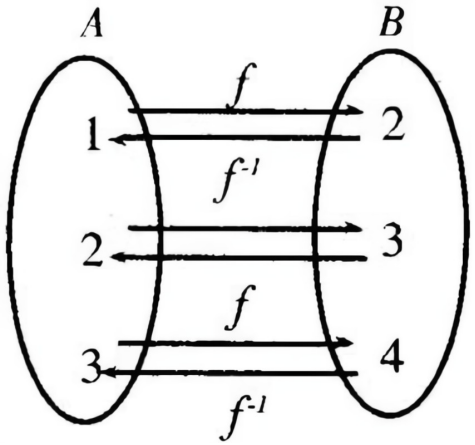
```
for (i=0; i<graph->Vcnt; i++) vtxSet[i] = i;

while(!IsEmpty(&hp) && edgeNum < (graph->Vcnt-1)) {
    edge = RemoveHeap(hp);
    if(vxtSet[edge.begin] == vtxSet[edge.end]) continue;        // 并查集Find操作
    mst[edgeCount++] = edge;
    edgeNum++;
    cntVtx = vtxSet[edge.end];
    for (i=0; i<graph->Vcnt; i++)
        if (vtxSet[i] == cntVtx) VtxSet[i]=vtxSet[edge.begin];  // 并查集Union操作
} }
ClearHeap[hp]; free(vtxSet); }
```

根节点相同
说明成环

总结：
在图的应用中，我们往往用并查集判断是否处于一个连通分支.....
在数据处理中，并查集是一个强有力的集合工具.....

哈希表—Hash table

- 散列表（Hash）将一组关键字映射到一个有限的、地址连续的区间上，并以关键字在地址集中的“像”作为相应记录在表中的存储位置




```
class hashtable
{
private:
int* elem;//元素指针 多指向数组
int count;//元素个数
public:
hashtable() { elem = NULL;
count = 0; }
int init();
int Hash_func(int key);
void insert(int key);
int search(const int& key);

};
```

散列表（Hash）最重要的一点是如何构建哈希函数，即我们希望做到一个完全的双射，但这绝大多数情况下是困难的！当Hash表的规模远远大于元素数目的时候，冲突容易避免，但是空间复杂度高！

Hash function

- 简单，能在较短时间内计算出结果
- 定义域必须包含全部关键字
- 若散列表允许有m个地址，其值域必须在$0 \sim m-1$之间
- 理想的散列函数应近似随机
- 对每一个输入，输出在值域上等概率，利于减少冲突发生

```
int hashtable::Hash_func(int key,int k)
{
    return k * key;//k为常数;
}//乘法散列;
```

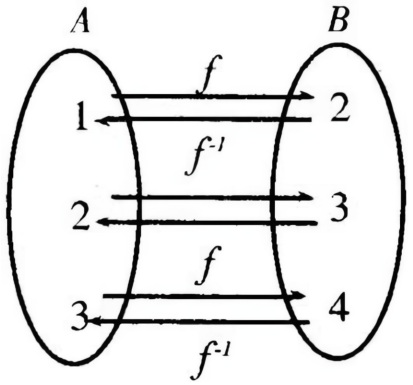
方法一： 乘法散列:对Key∈（0,1）,乘一个常数得到哈希表存贮地址;

Hash(Key)=M×Key;

对于更一般的情况，Key∈（s,t）,这个时候乘法散列函数适合为:

Hash(Key)=M×(k-s)/(t-s);

因为乘法散列是线性散列，故对Key要求为:
尽量均匀分布，不然容易出现储存地址聚集的现象



Hash function

方法二： 模散列:一般设定一个很大的质数M，用Key对M取模：(M<表长)

Hash(Key)=Key mod M;（1）

也可以把乘法和取模一起结合：

Hash(Key)=[Key×a] mod M; a常常为0.618；(2)

```
int hashtable::Hash_func(int key,int M)
{
        return (int)(0.618 * key)%M;
}
```

理论上来讲，只要素数相比较Key规模足够大，便可以很大程度上避免冲突; 但是对于大质数的寻找是一个比较困难的事情；因此往往我们会采用（2）的散列函数形式;

Hash function

方法三：数字分析法；n个d位数，每一位可能有r种不同符号，这r种符号出现的频率不尽相同，根据hash表的大小选取分布均匀的若干位作为散列地址——例如学生学号，前几位几乎相同，后面几位不同，可以取后面几位作为存储地址；

表 5.5　学生信息表

序　号	学　号	姓　名	班　级
1	2001010150	李定南	无 17
2	2001010151	王炳文	无 14
3	2001010152	张毅	无 11
4	2001010153	田长青	无 11
5	2001010154	高树	无 12
6	2001010155	陈卫	无 15
⋮	⋮	⋮	⋮
217	2001010366	林天波	无 11
218	2001010367	赵逸清	无 13
219	2001010368	王贵	无 13
220	2001010369	何敏	无 13

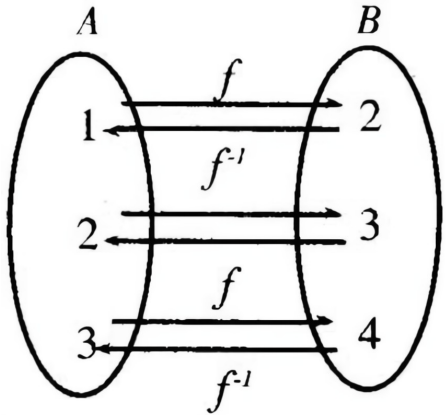
方法四：平方取中法：先计算构成关键字标志符内码的平方，然后按照哈希表的大小取平方后的若干位作为散列地址；其性能较好，因为“平方”的过程中，让中间取的位数卷入了整个数的性质，接近随机化；

表 5.8　平方取中法示例

标 识 符	内　码	内码的平方	散 列 地 址
A	01	1	1
A1	0134	20420	42
A9	0144	23420	342
B	02	4	4
DMAX	04150130	21526443617100	443
DMAX1	0415013034	5264473522151420	352
AMAX	01150130	135423617100	236
AMAX1	0115013034	3454246522151420	652

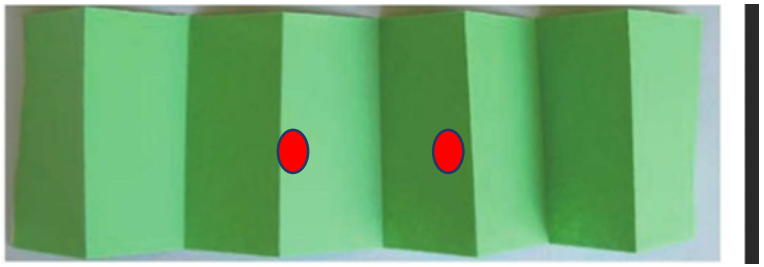
Hash function

方法五： 折叠法：把关键字自左到右分为和表长一样的部分，最后一部分可以短一点，然后把这些数加起来；



*** 移位法：把各个部分最后一位对齐相加，得到散列地址；**
Ex. Key：23938587841
239+385+878+41=1543 mod 1000=543；

*** 分界法：把各个部分数据沿各个部分来回折叠，然后对齐相加；**
例如如图的折纸，对折之后两个蓝点凑到了一起
Ex. Key：23938587841
239+583+878+41=1714 mod 1000=714；

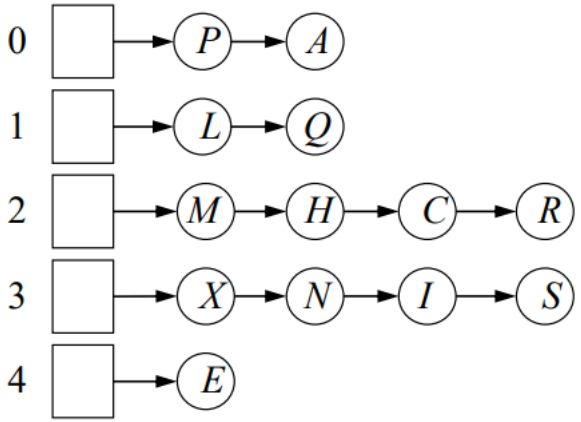


Hash Collisions——冲突

如果关键字散列到同一个地址，就是发生了冲突。无论怎样精心设计散列函数，都不能完全避免！

解决方法：开散列、闭散列法

开散列法(链地址法)：对每个散列地址建立一个链表，将散列到同一地址的关键字存入链表；



对于链地址法，装载因子α=N/M>1，需要额外的空间，平均查找时间为O（α），当空间充足的时候，一般常常选择M为N/5、N/10等等让搜索时间接近常数；

Hash Collisions——冲突

闭散列法（开放定址法）：依靠存贮空间余下的部分解决冲突，因为元素直接存再hash表空间中，故为闭散列方法；

探测：检查给定的表位置上是否存在一个与带搜关键词不同的元素称之为探测；最简单的是线性探测：

线性探测：当冲突发生的时候，即元素插入时候已经被其它元素占据，顺序检查下一个位置；如果碰到空位子，便是搜索失败，此时在结束的空位上面插入待插入的元素，若碰到表尾，则回到表头继续搜索；

```
bool Insert(const T& data)
{
size_t hashAddr = HashFunc(data);
size_t i = 0;//代表二次探测的探测次数
while (_vtable[hashAddr]._state != EMPTY)
{
if (_vtable[hashAddr]._state == EXIST &&
_vtable[hashAddr]._data == data)
{
return false;
}
//线性探测，依次往后遍历查找
if (isLine){}
//二次探测
else{}
}
//c.(循环结束，肯定已经找到空位置)插入元素
_vtable[hashAddr]._data = data;
_vtable[hashAddr]._state = EXIST;
_size++;
return true;
}
```

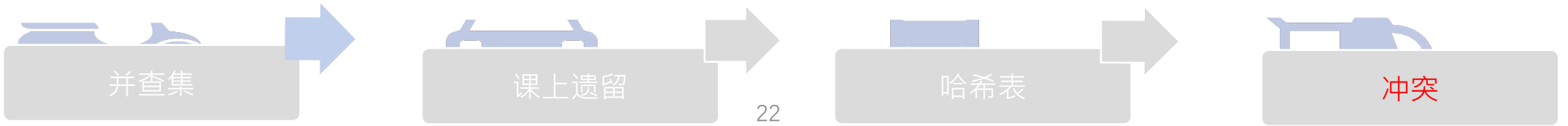
Hash Collisions——冲突

线性探测举例；

	H	D	J	W	I	R	L	U	X	M	A	V
ASCII	72	68	74	87	73	82	76	85	88	77	65	86
ADDR	7	3	9	9	8	4	11	7	10	12	0	8

$M = 13$

0	1	2	3	4	5	6	7	8	9	10	11	12
X	M	A	D	R	V		H	I	J	W	L	U
X	M	A	D	R			H	I	J	W	L	U
X	M		D	R			H	I	J	W	L	U
X			D	R			H	I	J	W	L	U
			D	R			H	I	J	W	L	U
			D	R			H	I	J	W	L	
			D	R			H	I	J	W		
			D				H	I	J	W		
			D				H		J	W		
			D				H		J			
			D				H					
							H					

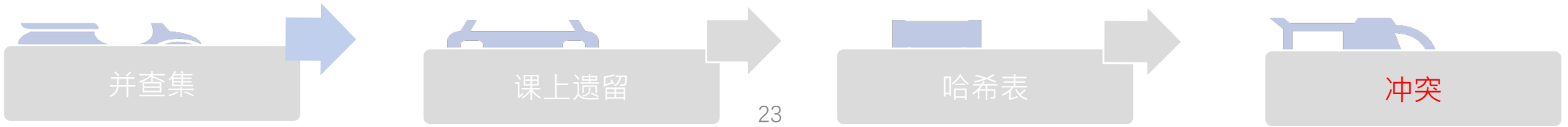


Hash Collisions——冲突

线性探测的缺点：随着hash表数目的增加，会出现元素聚集的现象（聚类），导致哈希表线性探测运行变慢；为此，我们引入新的方法：

双重散列，即不是检查冲突节点的下一个位置是否非空，而是采用第二个散列函数得到一个固定的增量序列确定探测序列;

开放定址法性能依赖于负载因子，α很小的时候，哈希表为稀疏表，此时大多数搜索只需要很少的探测次数便可以查询到，但是α接近1的时候，一次搜索需要相当多的搜索，性能不佳；但是相比较链接地址法，其还是具有较好的时间性能，但是空间利用率会降低；



散列的删除

对于链接地址法来说，删除操作很简单，只需进行链表的删除操作即可；

对于开放定址散列，是不可以直接删除的。其原因在于在开放定址散列中，元素除了携带自身的信息外还携带着链接元素的信息。故直接对元素删除是不行的！
解决方案1：元素加以标志留着hash表中；
解决方案2：重新散列；

Ex.比如下面删除L后查询U：

H	D	J	W	I	R	L	U	X	M	A	V
72	68	74	87	73	82	76	85	88	77	65	86
7	3	9	9	8	4	11	7	10	12	0	8

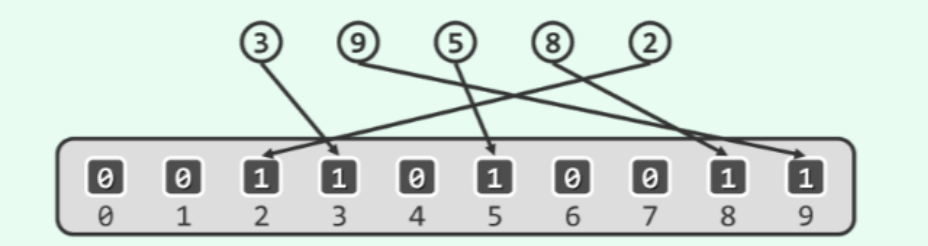
(a) 关键字及散列结果

	X	M	A	D	R	V		H	I	J	W	L	U
	X	M	A	D	R	V		H	I	J	W		U
0	1	2	3	4	5	6	7	8	9	10	11	12	

(b) 在哈希表中删除元素L前后的状态

Hash表——桶排序

- 基本排序方法在平均情况下的时间复杂度：$O(n^2)$
- 高级排序方法在平均情况下的时间复杂度：$O(n\log_2 n)$



利用hash表和最简单的函数hash（Key）=Key，我们便可以快速地排序，方法如下：

1、构造哈希表，初始化耗时O（M）；
2、把所有Key插入表中，耗时O（N）；
3、Key小到大，遍历hash表，读出表非空的元素，耗时O（M）；

整体耗时O（M+N）；

THANKS