第六章 集合与字典

- 集合及其表示
- 并查集与等价类
- 字典（略）
- 跳表（略）
- 散列
- 本章小结

6.1 集合及其表示

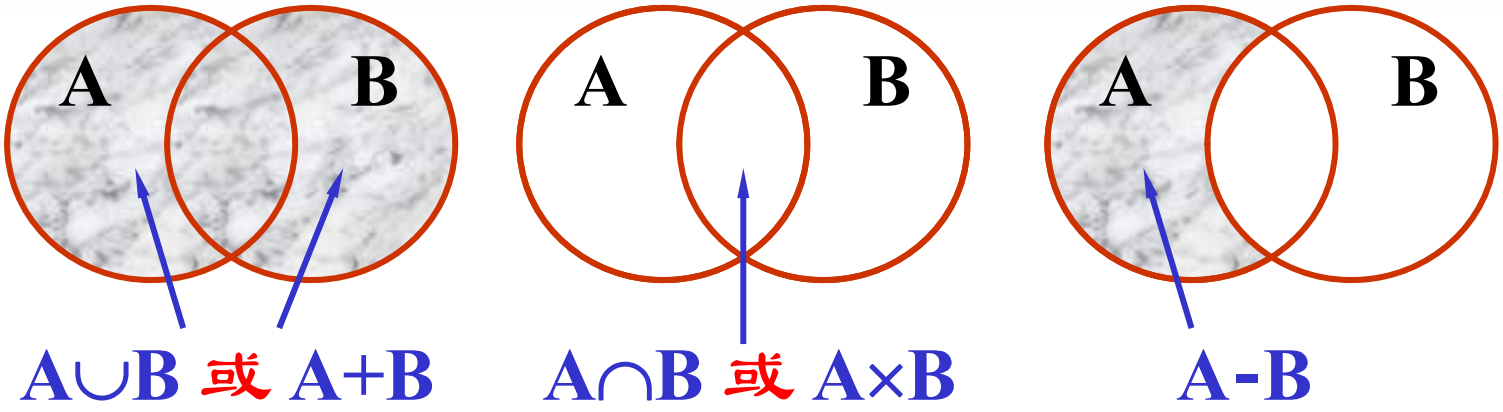
集合基本概念

- 集合是成员（对象或元素）的一个群集。集合中的成员可以是原子（单元素），也可以是集合。
- 集合的成员必须互不相同。
- 在算法与数据结构中所遇到的集合，其单元素通常是整数、字符、字符串或指针，且同一集合中所有成员具有相同的数据类型。

colour = { red, orange, yellow, green, black, blue, purple, white }

name = { “An”, “Cao”, “Liu”, “Ma”, “Peng”, “Wang”, “Zhang” }

- 集合中的成员一般是无序的，但在表示它时，常写在一个序列里。
- 常设定集合中的单元素具有线性有序关系，此关系可记作“<”，表示“优先于”。
- 整数、字符和字符串都有一个自然线性顺序，指针也可依据其在序列中安排的位置给予一个线性顺序。



集合(*Set*)的抽象数据类型

```
template <class Type> class Set {
publich:
    virtual Set ( ) = 0;
    virtual void MakeEmpty ( ) = 0;
    virtual bool AddMember ( const Type x ) = 0;
    virtual bool DelMember ( const Type x ) = 0;
```

```
    virtual  Set <Type> UnionWith
                        ( const Set <Type> &R ) = 0;
    virtual  Set <Type> IntersectWith
                        ( const Set <Type> &R ) = 0;
    virtual  Set <Type> DifferenceFrom
                        ( const Set <Type> &R ) = 0;
    virtual bool Contains ( const Type x ) = 0;
    virtual  bool <Type> SubSet
                        ( const Set <Type> &R ) = 0;
    virtual  bool <Type> operator ==
                        ( const Set <Type> &R ) = 0;
};
```

用位向量实现集合抽象数据类型

- 当集合是全集合 { 0, 1, 2, …, *n* } 的一个子集，且 *n* 是不大的整数时，可用位 (0, 1) 向量来实现集合。
- 当全集合是由有限的可枚举的成员组成的集合时，可建立全集合成员与整数 0, 1, 2, … 的一一对应关系，用位向量来表示该集合的子集。

集合的位向量(bitVector)类的定义

```
const int DefaultSize = 100;
template <class Type> class BitSet {
private:
    unsigned short *bitVector;  //集合的位向量数组
    int setSize;
public:
    BitSet ( int sz = DefaultSize );
    BitSet ( const BitSet <Type> &R );
    ~BitSet ( ) { delete [ ] bitVector; }
     void MakeEmpty ( ) {  //置空
        for ( int i = 0; i < MaxSize; i++ )
              bitVector[i] = 0;    }
```

```
    int GetMember ( const Type x );
    void PutMember ( const Type x, int v );
    bool AddMember ( const Type x );  //加
    bool DelMember ( const Type x );  //删
    BitSet <Type> operator = ( const BitSet <Type> &R );
    BitSet <Type> operator + ( const BitSet <Type> &R );
    BitSet <Type> operator * ( const BitSet <Type> &R );
    BitSet <Type> operator - ( const BitSet <Type> &R );
    bool Contains ( const Type x );  //判存在
    bool SubSet ( BitSet <Type> &R );  //判子集
    bool operator == ( BitSet <Type> &R );  //判相等
};
```

使用示例

```
Set s1, s2, s3, s4, s5;  int index, equal;
for ( int k = 0; k < 10; k++ )  //集合赋值
{ s1.AddMember (k);  s2.AddMember (k +7); }
// s1 : { 0, 1, 2, …, 9 },   s2 : { 7, 8, 9, …, 16 }
s3 = s1+s2;  //求 s1 与 s2 的并 { 0, 1, …, 16 }
s4 = s1*s2;  //求 s1 与 s2 的交 { 7, 8, 9 }
s5 = s1-s2;  //求 s1 与 s2 的差 { 0, 1, …, 6 }
index = s1.SubSet (s4);  //判断 s1 是否为 s4 子集
cout << index << endl;  //结果，index = 0
// s4 : { 7, 8, 9 }
equal = s1 == s2;  //集合 s1 与 s2 比较相等
cout << equal << endl;  //为 0，两集合不等
```

用位向量实现集合时部分操作的实现

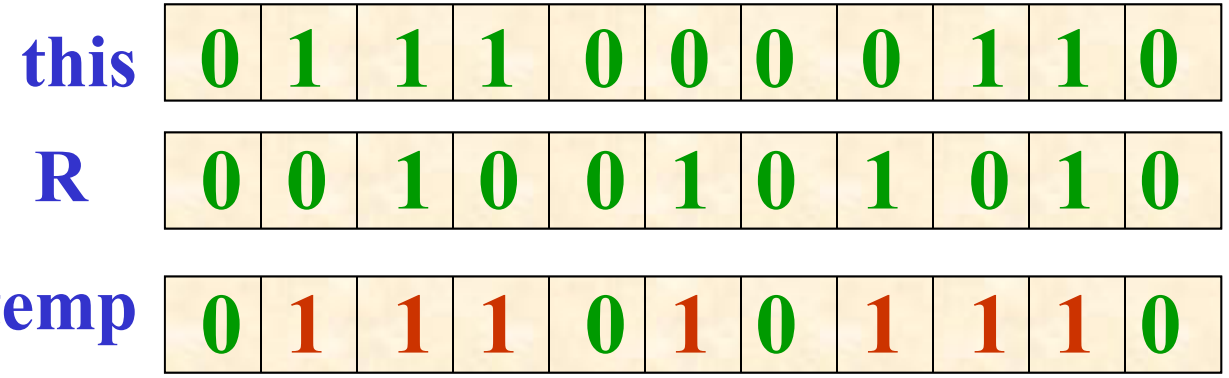
```
template <class Type> BitSet <Type> ::
BitSet ( int sz ) : setSize (sz) {
    assert ( setSize > 0 );  //判参数合理否
    bitVector = new int [setSize];  //分配空间
    assert ( bitVector != NULL );
    for ( int i = 0; i < setSize; i++ )  //置空集合
        bitVector[i] = 0;
}
```

```
template <class Type> BitSet <Type> ::
BitSet ( BitSet <Type> &R ) {  //复制构造函数
    setSize = R.setSize;  //位向量大小
    bitVector = new int [setSize];  //分配空间
    assert ( bitVector != NULL );
    for ( int i = 0; i < setSize; i++ )  //传送集合
        bitVector[i] = R.bitVector[i];
}
```

```
template <class Type> bool BitSet <Type> ::
Addmember ( const int x ) {
    if ( x >= 0 && x < setSize )
       { bitVector[x] = 1;  return true; }
    return false;
}

template <class Type> bool BitSet <Type> ::
DelMember ( const int x ) {
    if ( x >= 0 && x < setSize )
       { bitVector[x] = 0;  return true; }
    return false;
}
```

```
template <class Type> BitSet <Type> BitSet <Type> ::
operator = ( BitSet <Type> &R ) {  //赋值
    assert ( setSize == R.setSize );
    for ( int i = 0; i < setSize; i++ )  //逐位传送
        bitVector[i] = R.bitVector[i];
}

template <class Type> bool BitSet <Type> ::
Contains ( const int x ) {  //判存在
    assert ( x >= 0 && x < setSize );
    return ( bitVector[x] == 1 ) ? true : false;

}
```

```
template <class Type> BitSet <Type> BitSet <Type> ::
operator + ( BitSet <Type> &R ) {  //求并
    assert ( setSize == R.setSize );
    BitSet temp (setSize);
    for ( int i = 0; i < setSize; i++ )
    temp.bitVector[i] = bitVector[i] || R.bitVector[i];
    return temp;
}
```



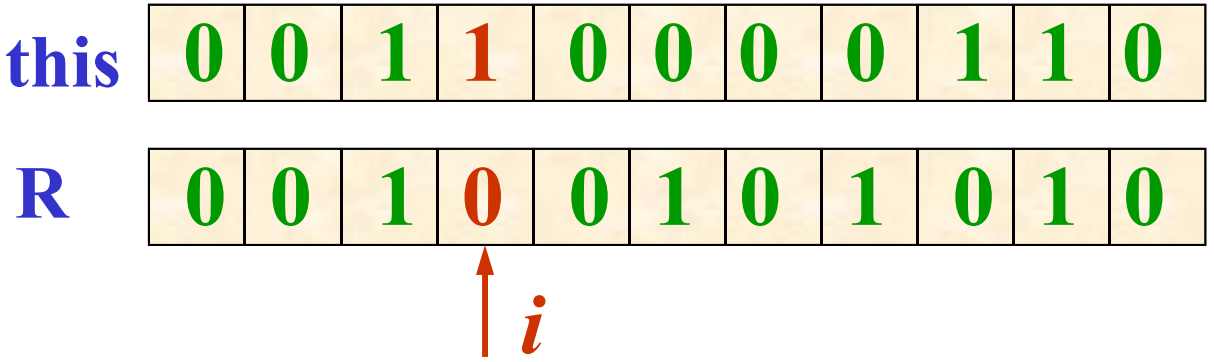
```
template <class Type> BitSet <Type> BitSet <Type> ::
operator * ( BitSet <Type> &R ) {  //求交
    assert ( setSize == R.setSize );
    BitSet temp (setSize);
    for ( int i = 0; i < setSize; i++)
        temp.bitVector[i] =
                bitVector[i] && R.bitVector[i];
    return temp;
}
```

this	0	1	1	1	0	0	0	0	1	1	0
R	0	0	1	0	0	1	0	1	0	1	0
temp	0	0	1	0	0	0	0	0	0	1	0

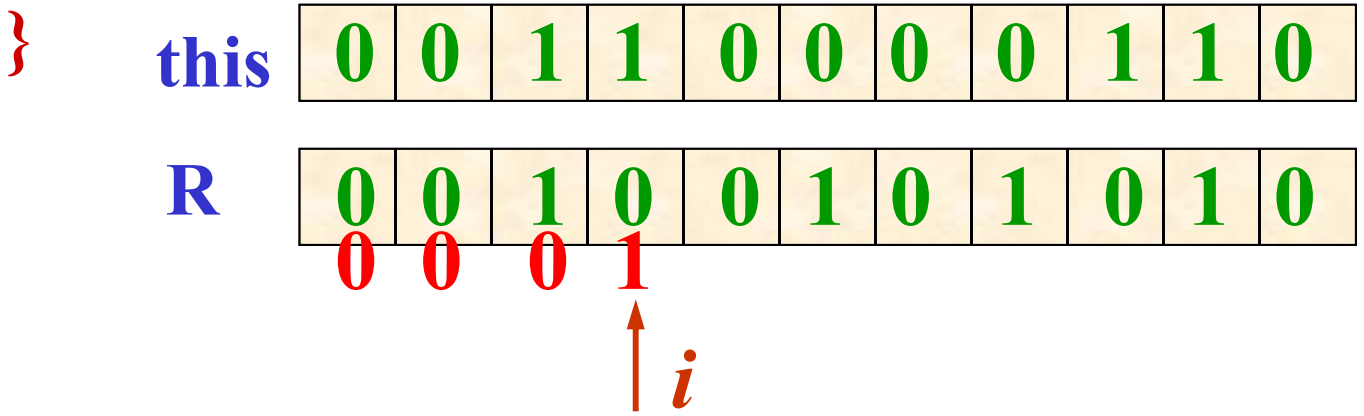
```
template <class Type> BitSet <Type> BitSet <Type> ::
operator - ( BitSet <Type> &R ) {  //求差
    assert ( setSize == R.setSize );
    BitSet temp (setSize);
    for ( int i = 0; i < setSize; i++ )
        temp. bitVector[i] =
                bitVector[i] && !R.bitVector[i];
    return temp;
}
```

this	0	1	1	1	0	0	0	0	1	1	0
R	0	0	1	0	0	1	0	1	0	1	0
temp	0	1	0	1	0	0	0	0	1	0	0

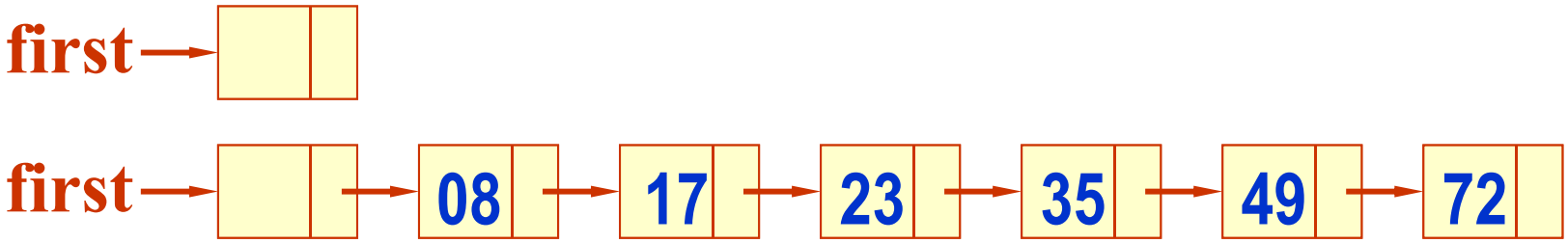
```
template <class Type> bool BitSet <Type> ::
operator == ( BitSet <Type> &R ) {  //判等
   assert ( setSize == R.setSize );
   for ( int i = 0; i < setSize; i++ )
      if ( bitVector[i] != R.bitVector[i] )
         return false;
   return true;
}
```



```
template <class Type> bool BitSet <Type> ::
SubSet ( BitSet <Type> &R ) {
//判断 this 是否是 R 的子集
   assert ( setSize == R.setSize );
   for ( int i = 0; i < setSize; i++ )
      if ( bitVector[i] && !R.bitVector[i] )
         return false;
   return true;
}
```



用有序链表实现集合的抽象数据类型



用带表头结点的有序链表表示集合

- 用有序链表来表示集合时，链表中的每个结点表示集合的一个成员。
- 各结点所表示的成员 $e_0, e_1, \ldots, e_n$ 在链表中按升序排列，即 $e_0 < e_1 < \ldots < e_n$。
- 集合成员可以无限增加，因此用有序链表可以表示无穷全集合的子集。

集合的有序链表类的定义

```
template <class Type> class LinkedSet;
template <class Type> class SetNode {
friend class LinkedSet <Type>;
public:
    SetNode ( ) : link(NULL) { }
    SetNode ( Type item ) :
                data(item), link(NULL) { }
private:
    Type data;
    SetNode <Type> *link;
};
```

```
template <class Type> class LinkedSet {
private:
    SetNode <Type> *first, *last;
public:
    LinkedSet ( )   //构造函数
      { first = last = new SetNode <Type>; }
    LinkedSet ( LinkedSet <Type> &R );
    ~LinkedSet ( ) { MakeEmpty( );  delete first; }
    void MakeEmpty ( );  //置空集合
    bool AddMember ( const Type &x );
    bool DelMember ( const Type &x );
    void  operator = ( LinkedSet <Type> &R );
    //将集合 R 赋给集合 this
```

```
    LinkedSet <Type>  operator + ( LinkedSet <Type> &R );
    //求集合 this 与集合 R 的并
    LinkedSet <Type>  operator * ( LinkedSet <Type> &R );
    //求集合 this 与集合 R 的交
    LinkedSet <Type>  operator - ( LinkedSet <Type> &R );
    //求集合 this 与集合 R 的差
    bool Contains ( const Type &x );
    //判 x 是否集合的成员
    bool operator == ( LinkedSet <Type> &R );
    //判集合 this 与集合 R相等
    bool Min ( Type &x );  //返回集合最小元素值
    bool Max ( Type &x );  //返回集合最大元素值
    bool SubSet ( LinkedSet <Type> &R );
};
```

集合的复制构造函数

```
template <class Type> LinkedSet <Type> ::
LinkedSet ( LinkedSet <Type> &R ) {
    SetNode <Type> *srcptr = R.first->link;
    first = last = new SetNode <Type>;
    while ( srcptr != NULL ) {
        last = last->link =
                new SetNode <Type> (srcptr->data);
        srcptr = srcptr->link;
    }
    last->link = NULL;
}
```

集合的搜索、加入和删除操作

```
template <class Type> bool LinkedSet <Type> ::
Contains ( const Type &x ) {
//若 x 是集合成员，则函数返回 true，否则返回 false
    SetNode <Type> *temp = first->link;
    while ( temp != NULL && temp->data < x )
        temp = temp->link; //循链搜索
     if ( temp != NULL && temp->data == x )
        return true; //找到，返回 true
    else return false; //未找到，返回 false
}
```

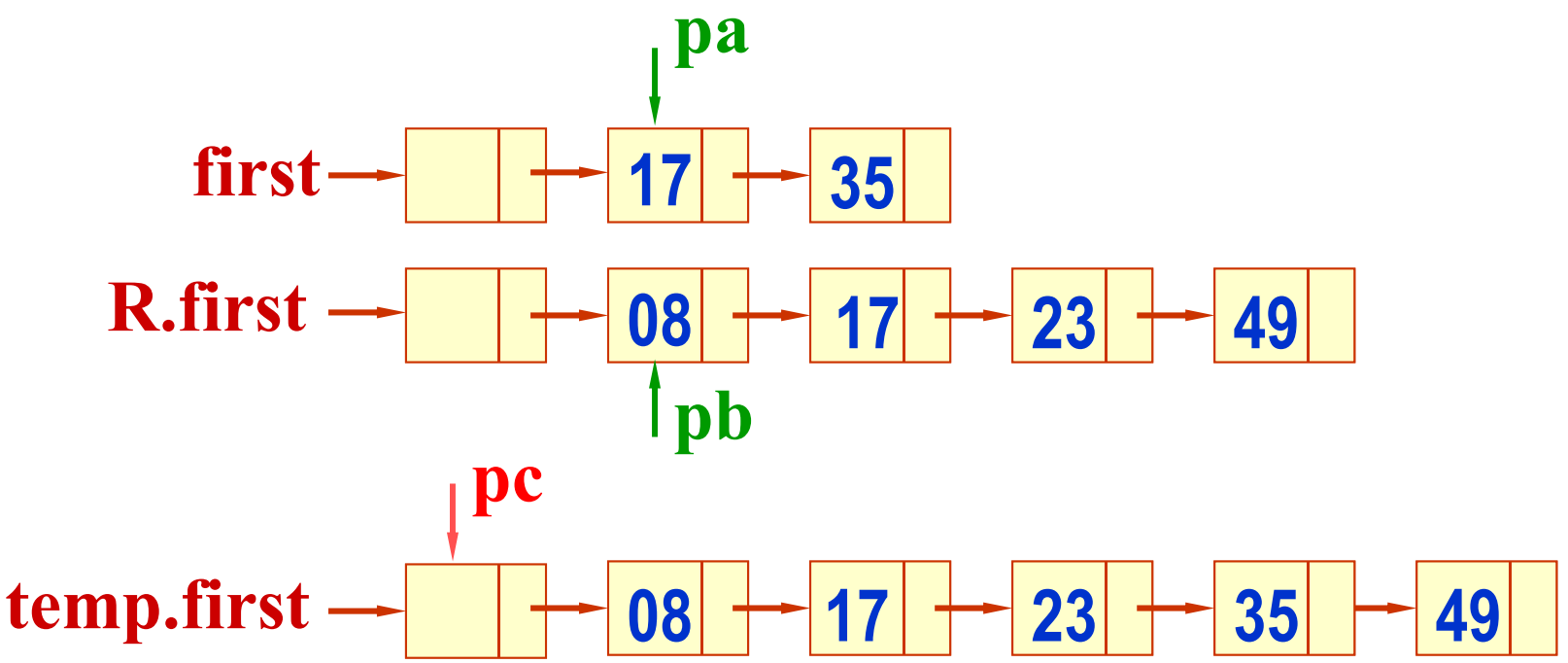
```
template <class Type> bool LinkedSet <Type> ::
AddMember ( const Type &x ) {
    SetNode <Type> *p = first->link, *q = first;
    while ( p != NULL && p->data < x )
        { q = p;  p = p->link; }  //循链扫描
    if ( p != NULL && p->data = = x ) return false;
    //集合中已有此元素
    SetNode <Type> *s = new SetNode (x);
    s->link = p;  q->link = s;  //链入
    if ( p == NULL ) last = s;
    //链到链尾时改链尾指针
    return true;
}
```

```
template <class Type> bool LinkedSet <Type> ::
DelMember ( const Type &x ) {
    SetNode <Type> *p = first->link, *q = first;
    while ( p != NULL && p->data < x )
        { q = p;  p = p->link; }  //循链扫描
     if ( p != NULL && p->data = = x ) {  //找到
        q->link = p->link;  //重新链接
        if ( p == last ) last = q;
        //删去链尾结点时改链尾指针
        delete p;  return true;  //删除含 x 结点  }
    else return false;  //集合中无此元素
}
```

用有序链表表示集合时的几个重载函数

```
template <class Type> LinkedSet <Type>
LinkedSet <Type> ::
operator = ( LinkedSet <Type> &R ) {
//复制集合 R 到 this
    SetNode <Type> *pb = R.first->link;  //源
    SetNode <Type> *pa = first
                       = new SetNode <Type>;  //目标
    while ( pb != NULL ) {  //逐个结点复制
      pa->link = new SetNode <Type> ( pb->data );
      pa = pa->link;  pb = pb->link;  }
    pa->link = NULL;  last = pa;  //目标链表收尾
}
```

```
//求集合 this 与集合 R 的并
//结果通过临时集合 temp 返回
// this 集合与 R 集合不变
```



```
template <class Type> LinkedSet <Type>
LinkedSet <Type> ::
operator + ( LinkedSet <Type> &R ) {
    SetNode <Type> *pb = R.first->link;
    SetNode <Type> *pa = first->link;
    LinkedSet <Type> temp;
    SetNode <Type> *pc = temp.first;
    while ( pa != NULL && pb != NULL ) {
        if ( pa->data == pb->data )   { //共有元素
            pc->link = pa;  pa = pa->link;  pb = pb->link;
        }
        else if ( pa->data < pb->data ) {
            pc->link = pa;  pa = pa->link;
        }
```

```
        else {  // pa->data > pb->data
          pc->link=new SetNode <Type> (pb->data);
          pb = pb->link;
        }
        pc = pc->link;
    }
    if ( pa != NULL ) pb = pa;  // pb 指未扫完集合
    while ( pb != NULL ) {  //向结果链逐个复制
      pc->link = new SetNode <Type> (pb->data);
      pc = pc->link;  pb = pb->link;
    }
    pc->link = NULL;  temp.last = pc;  //链表收尾
    return temp;
}
```

```
template <class Type> bool LinkedSet <Type> ::
operator == ( LinkedSet <Type> &R ) {
    SetNode <Type> *pb = R.first->link;
    SetNode <Type> *pa = first->link;
    while ( pa != NULL && pb != NULL )
        if ( pa->data == pb->data )  //相等，继续
          { pa = pa->link;  pb = pb->link; }
        else return false;  //不等时中途退出，返回 false
    if ( pa != NULL || pb != NULL ) return flase;
    //链不等长时，返回 false
    return true;
}
```

6.2 并查集与等价类

等价关系与等价类(Equivalence Class)

- 在求解实际应用问题时常会遇到等价类问题。
- 从数学上看，等价类是一个对象（或成员）的集合，在此集合中所有对象应满足等价关系。
- 若用符号“≡”表示集合上的等价关系，那么对于该集合中的任意对象x, y, z，下列性质成立：
  - 自反性：$x \equiv x$ （即等于自身）。
  - 对称性：若$x \equiv y$，则$y \equiv x$。
  - 传递性：若$x \equiv y$且$y \equiv z$，则$x \equiv z$。
- 因此，等价关系是集合上的一个自反、对称、传递的关系。

- “相等”(=)就是一种等价关系，它满足上述的三个特性。
- 一个集合 S 中的所有对象可以通过等价关系划分为若干个互不相交的子集 $S_1, S_2, S_3, \cdots$，它们的并就是 S，这些子集即为等价类。

确定等价类的方法

**分两步走——

第一步，读入并存储所有的等价对(i, j)；

第二步，标记和输出所有的等价类。

```
void equivalence ( ) {
    初始化;
    while 等价对未处理完
        { 读入下一个等价对 ( i, j );
          存储这个等价对 ; }
    输出初始化;
    for ( 尚未输出的每个对象 )
      输出包含这个对象的等价类;
}
```

给定集合 $S = \{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 \}$,

及如下等价对: $0 \equiv 4, 3 \equiv 1, 6 \equiv 10, 8 \equiv 9, 7 \equiv 4,$

$6 \equiv 8, 3 \equiv 5, 2 \equiv 11, 11 \equiv 0$ 。

初始 {0}, {1}, {2}, {3}, {4}, {5}, {6}, {7}, {8}, {9}, {10}, {11}

0 ≡ 4 {0, 4}, {1}, {2}, {3}, {5}, {6}, {7}, {8}, {9}, {10}, {11}

3 ≡ 1 {0, 4}, {1, 3}, {2}, {5}, {6}, {7}, {8}, {9}, {10}, {11}

6 ≡ 10 {0, 4}, {1, 3}, {2}, {5}, {6, 10}, {7}, {8}, {9}, {11}

8 ≡ 9 {0, 4}, {1, 3}, {2}, {5}, {6, 10}, {7}, {8, 9}, {11}

7 ≡ 4 {0, 4, 7}, {1, 3}, {2}, {5}, {6, 10}, {8, 9}, {11}

6 ≡ 8 {0, 4, 7}, {1, 3}, {2}, {5}, {6, 8, 9, 10}, {11}

3 ≡ 5 {0, 4, 7}, {1, 3, 5}, {2}, {6, 8, 9, 10}, {11}

2 ≡ 11 {0, 4, 7}, {1, 3, 5}, {2, 11}, {6, 8, 9, 10}

11 ≡ 0 {0, 2, 4, 7, 11}, {1, 3, 5}, {6, 8, 9, 10}

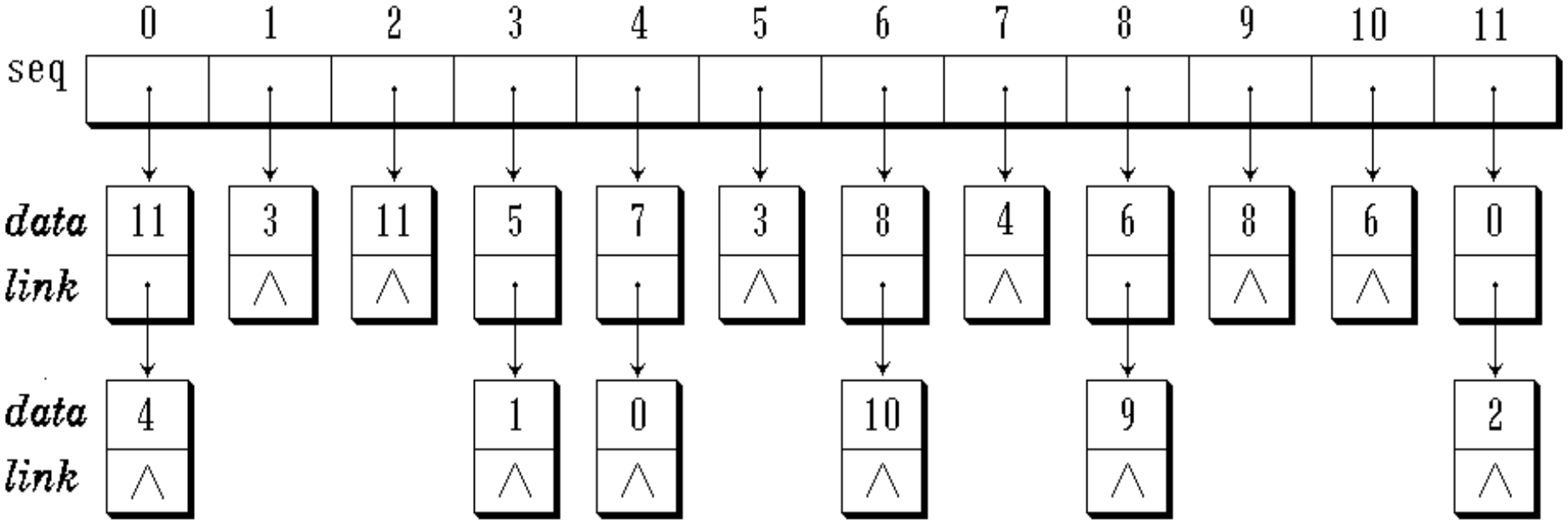
确定等价类的链表方法

设等价对个数为 m，对象个数为 n，一种可选的存储表示为单链表。

可为集合的每一对象建立一个带表头结点的单链表，并建立一个一维的指针数组 seq[n] 作为各单链表的表头结点向量。seq[i] 是第 i 个单链表的表头结点，第 i 个单链表中所有结点的 data 域存放在等价对中与 i 等价的对象编号。

当输入一个等价对 (i, j) 后，就将集合元素 i 链入第 j 个单链表，且将集合元素 j 链入第 i 个单链表。

在输出时，设置一个布尔数组 out[n] ，用 out[i] 标记第 i 个单链表是否已经输出。



- 算法的输出从编号 i = 0 的对象开始，对所有的对象进行检测。
- 在 i = 0 时，循第 0 个单链表先找出形式为(0, j)的等价对，把 0 和 j 作为同一个等价类输出。再根据等价关系的传递性，找所有形式为(j, k)的等价对，把 k 也纳入包含 0 的等价类中输出。如此继续，直到包含 0 的等价类完全输出为止。
- 接着再找一个未被标记的编号，如 i = 1，该对象将属于一个新的等价类，再用上述方法划分、标记和输出这个等价类。
- 在算法中使用了一个栈，每次输出一个对象编号时，都要把这个编号进栈，记下以后还要检测输出的等价对象的单链表。

建立等价类算法（输入等价对并输出等价类）

每当一个对象的单链表检测完，就需要从栈中退出一个指针，以便继续输出等价类中的其它对象。如果栈空，说明该等价类所有对象编号都已输出，再找一个使得 out[i] == False 的最小的 i，标记并输出下一个等价类。

给定集合：

$S = \{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 \}$

及如下等价对：

$0 \equiv 4, 3 \equiv 1, 6 \equiv 10, 8 \equiv 9, 7 \equiv 4,$

$6 \equiv 8, 3 \equiv 5, \ 2 \equiv 11, 11 \equiv 0$

链序号	等价对	OUT初态	输出	OUT终态	栈
0		False	0	True	
0	11	False	11	True	11
0	4	False	4	True	11,4
4	7	False	7	True	11,7
4	0	True	—	True	11,7

链序号	等价对	OUT初态	输出	OUT终态	栈
7	4	True	—	True	11
11	0	True	—	True	
11	2	False	2	True	2
2	11	True	—	True	

含0的等价类输出的对象顺序

等价类链表的定义

```
enum Boolean { False, True };
class ListNode {  //定义链表结点类
friend void Equivalence ( );
private:
    int data;  //结点数据
    ListNode *link;  //结点链指针
    ListNode ( int d ) { data = d;  link = NULL; }
};
typedef ListNode *ListNodePtr;
```

```
void Equivalence ( ) {
   ifstream inFile ( "equiv.in", ios::in );  //输入文件
    if ( !inFile ) {
       cerr << "不能打开输入文件" << endl;
       exit (1);
    }
    int i, j, n;
    inFile >> n;  //读入对象个数
    seq = new ListNodePtr[n];
    out = new Boolean[n];  //初始化 seq 和 out
    for ( i = 0; i < n; i++ )
       { seq[i] = 0;  out[i] = False; }
    inFile >> i >> j;  //输入等价对 ( i, j )
```

```
while ( inFile.good ( ) ) {  //文件结束出循环
    x = new ListNode ( j );  //创建结点 j
    x->link = seq[i];  seq[i] = x;  //链入第 i 个链表
    y = new ListNode ( i );  //创建结点 i
    y->link = seq[j];  seq[j] = y;  //链入第 j 个链表
    inFile >> i >> j;  //输入下一个等价对
}
for ( i = 0; i < n; i++ )
  if ( out[i] = = False ) {  //未输出，需要输出
    cout<< endl << "A new class: " << i;  //输出
    out[i] = True;  //作输出标记
    ListNode *x = seq[i];  //取第 i 链表头指针
```

```
ListNode *top = NULL;  //栈初始化
while (1) {  //找类的其它成员
  while ( x ) {  //处理链表，直到 x=0
    j = x->data;  //成员 j
    if ( out[j] == False ) {  //未输出，输出
      cout << "," << j;  out[j]=True;
      ListNode *y = x->link;
      x->link = top;  top = x;  //结点 x 进栈
      x = y;  // x 进到链表下一个结点
    }
    else x = x->link;  //已输出过，跳过
  }
```

```
            if ( top == NULL ) break;
            //栈空退出循环
            else {
              x = seq[top->data];
              top = top->link;
            }
            //栈不空，退栈
            // x 是根据结点编号回溯的另一个链表的头指针
        }
    }
    delete [ ] seq;  delete [ ] out;
}
```

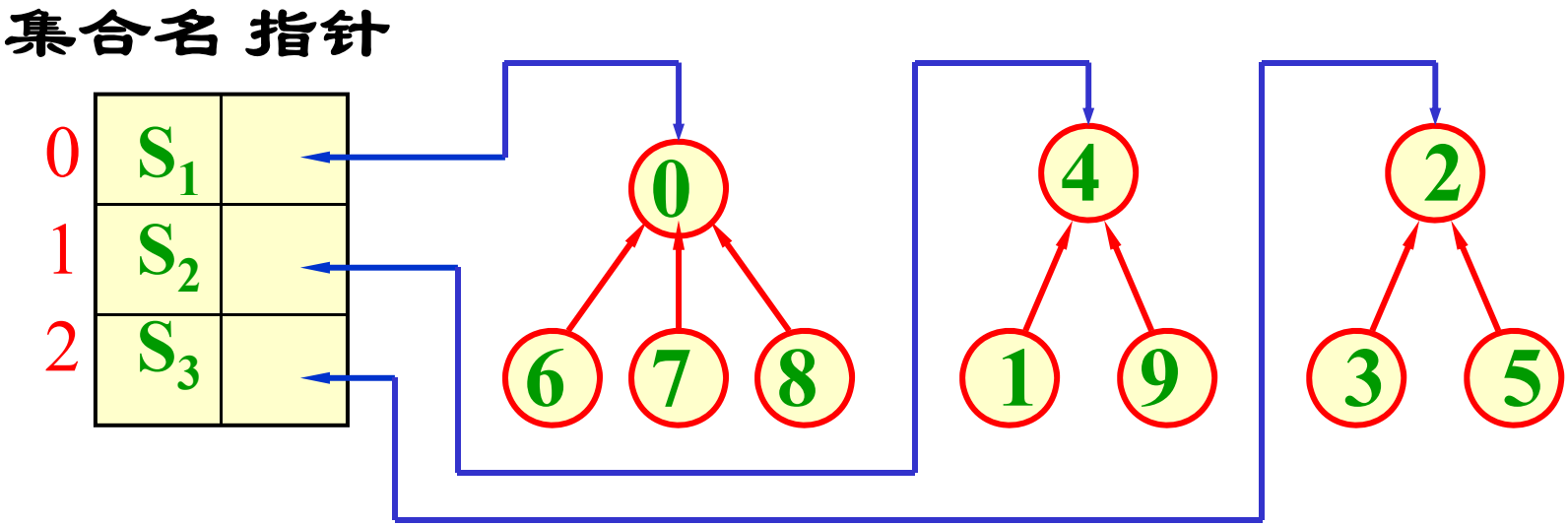
并查集 (Union-Find Sets)

- 建立等价类的另一种解决方案是先把每一个对象看作是一个单元素集合，然后按一定顺序将属于同一等价类的元素所在的集合合并。
- 在此过程中将反复地使用一个搜索运算，确定一个元素在哪一个集合中。

- 能够完成这种功能的集合就是并查集，支持以下三种操作：
  - Union (Root1, Root2) //并操作
    - 把子集合 Root2 并入集合 Root1 中，要求 Root1 与 Root2 互不相交，否则无结果。
  - Find (x) //搜索操作
    - 搜索单元素 x 所在的集合，并返回该集合名字。
  - UFSets (s) //构造函数
    - 将并查集中的 s 个元素初始化为 s 个只有一个单元素的子集合，根结点的 parent 值等于-1。

- 对于并查集来说，每个集合用一棵树表示。
- 为此，采用树的双亲表示作为集合存储表示。集合元素的编号从 0 到 n-1，其中 n 是最大元素个数。
- 在双亲表示中，第 i 个数组元素代表包含集合元素 i 的树结点。根结点的双亲为 -1，表示集合中的元素个数。
- 在同一棵树上所有结点所代表的集合元素在同一个子集合中。
- 为此，需要有两个映射：
  - 集合元素到存放该元素名的树结点间的对应；
  - 集合名到表示该集合的树的根结点间的对应。

- 设 $S_1 = \{ 0, 6, 7, 8 \}, S_2 = \{ 1, 4, 9 \}, S_3 = \{ 2, 3, 5 \}$



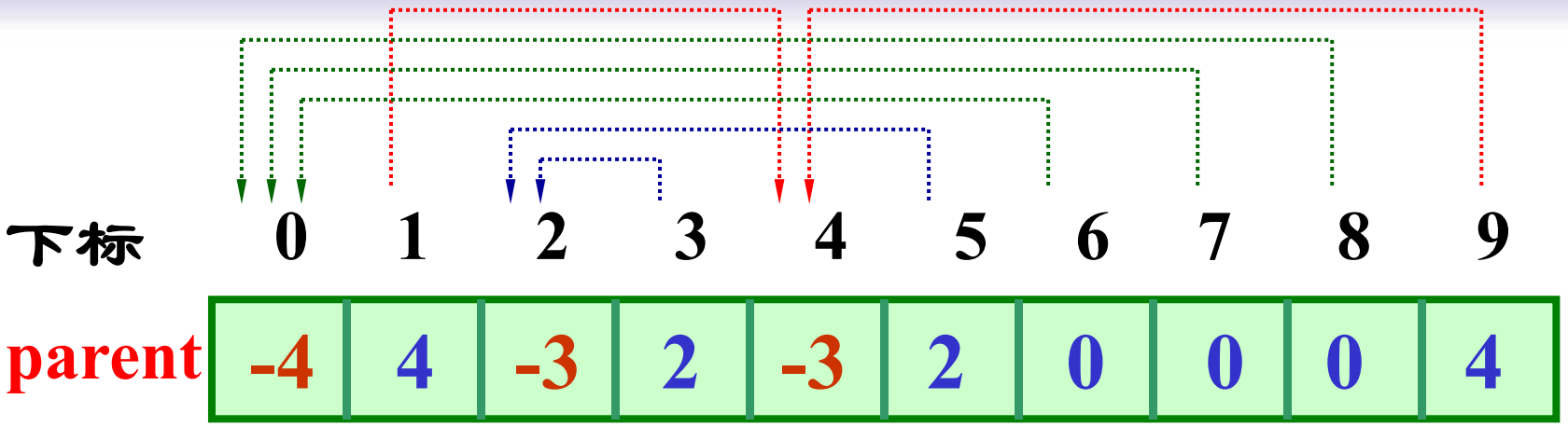
- 为简化讨论，忽略实际的集合名，仅用表示集合的树的根来标识集合。

利用并查集来解决等价问题的步骤

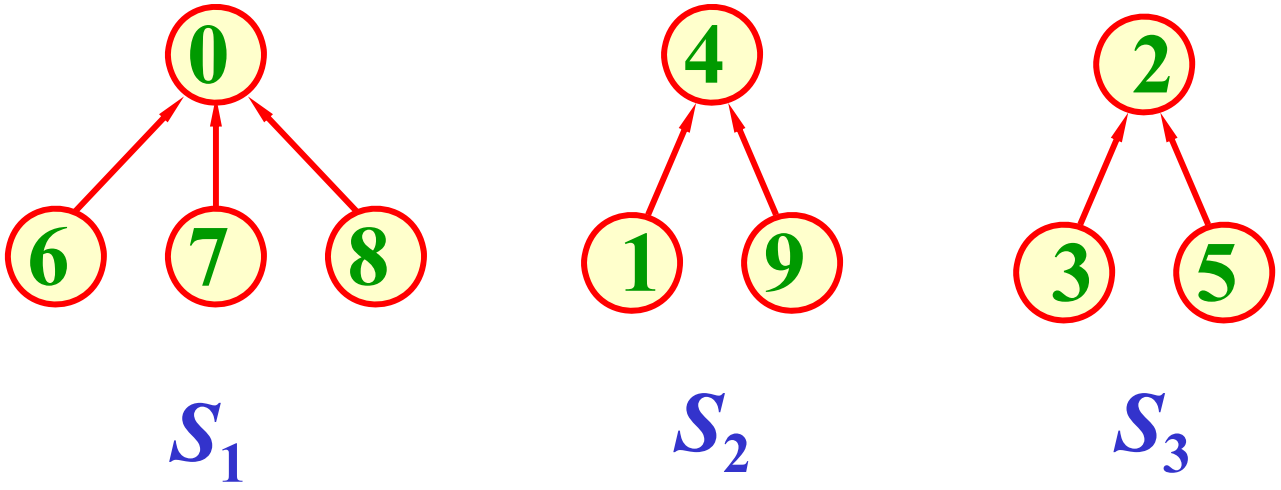
- 初始时，用构造函数 UFSets(s) 构造一个森林，每棵树只有一个结点，表示集合中各元素自成一个子集。

下标	0	1	2	3	4	5	6	7	8	9
parent	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

- 用 Find(i) 寻找集合元素 i 的根，如果有两个集合元素 i 和 j，Find(i) == Find(j)，表明这两个元素在同一个集合中。
- 如果两个集合元素 i 和 j 不在同一集合中，可用 Union(i, j) 将其合并到一集合中。

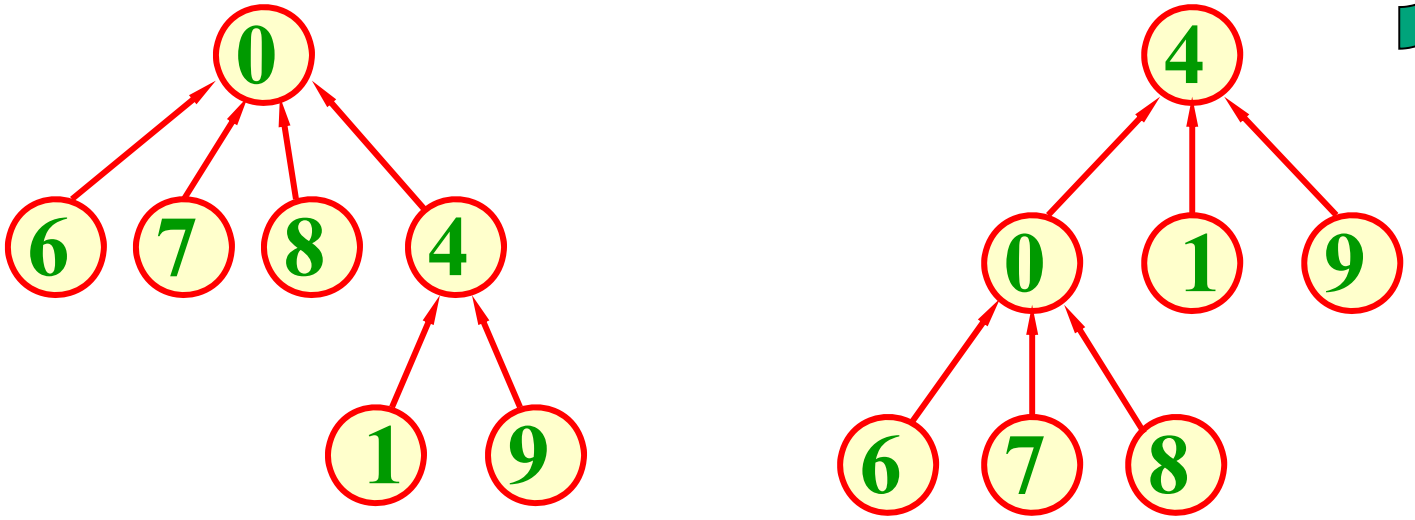


集合 S_1、S_2 和 S_3 的双亲表示



下标	0	1	2	3	4	5	6	7	8	9
parent	-7	4	-3	2	0	2	0	0	0	4

集合 $S_1 \cup S_2$ 和 S_3 的双亲表示



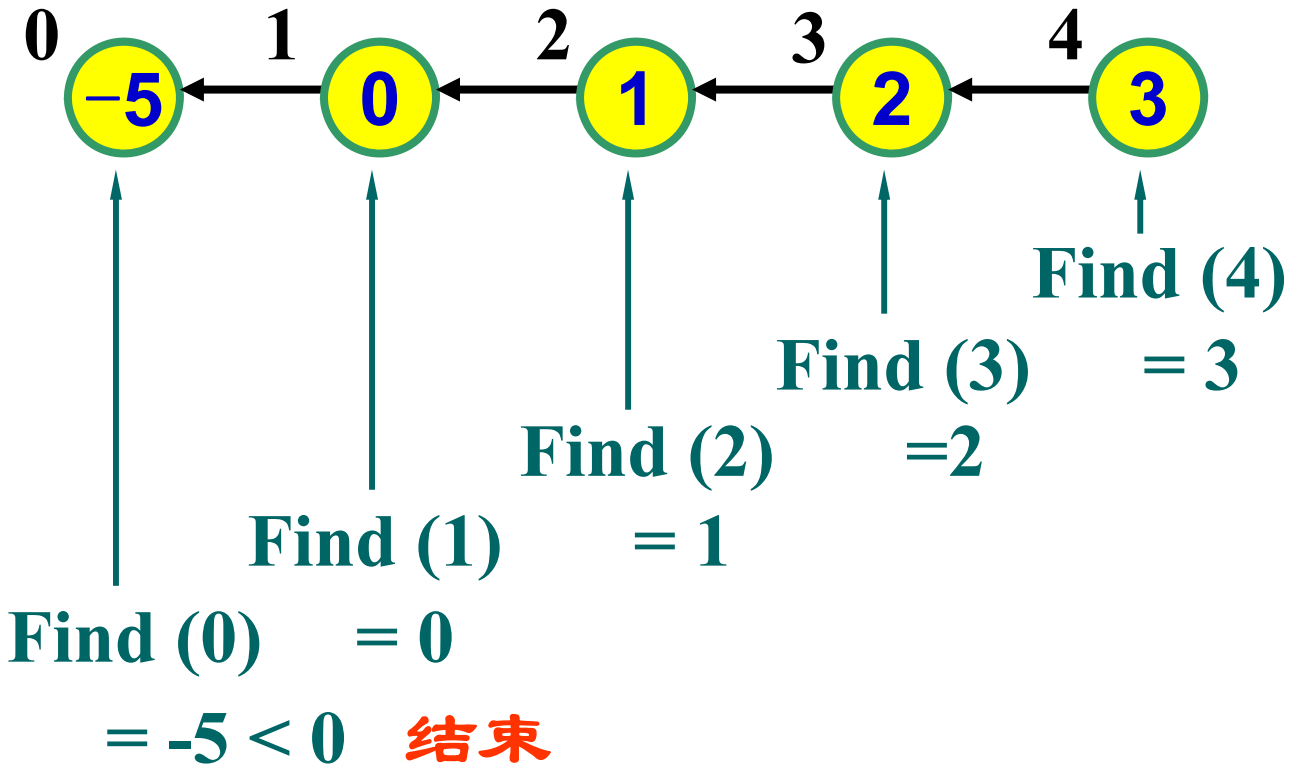
$S_1 \cup S_2$ 的可能的表示方法

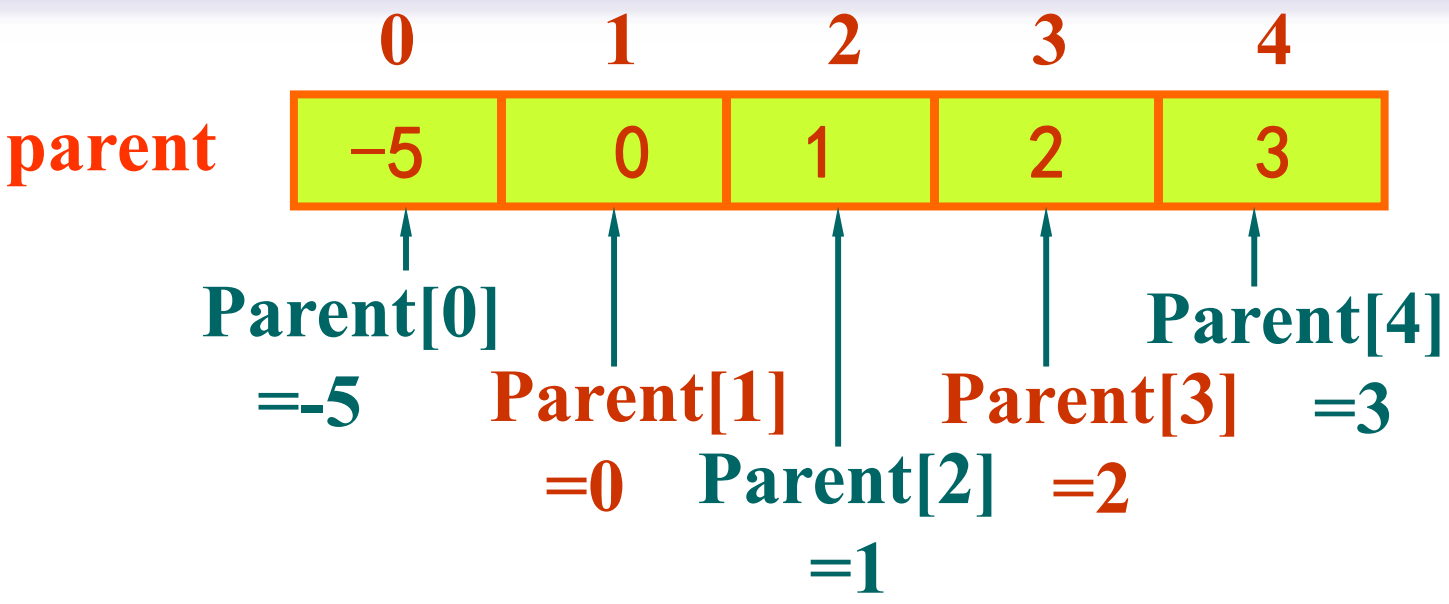
```
const int DefaultSize = 10;
class UFSets {  //并查集类定义
private:
    int *parent;  //集合元素数组
    int size;  //集合元素的数目
public:
    UFSets ( int s = DefaultSize );  //构造函数
    ~UFSets ( ) { delete [ ] parent; } //析构函数
    const UFSets & operator = ( UFSets &R );
    //重载函数：集合赋值
    void Union ( int Root1, int Root2 );
    //基本例程：两个子集合合并
```

```
    int Find ( int x );
    //基本例程：搜寻集合x的根
    void UnionByHeight ( int Root1, int Root2 );
    //改进例程：压缩高度的合并算法
};

UFSets :: UFSets ( int s ) {  //构造函数
    size = s;  //集合元素个数
    parent = new int [size];  //双亲指针数组
    for ( int i = 0; i < size; i++ ) parent[i] = -1;
    //每一个自成一个单元素集合
}
```

并查集操作的算法

- 查找



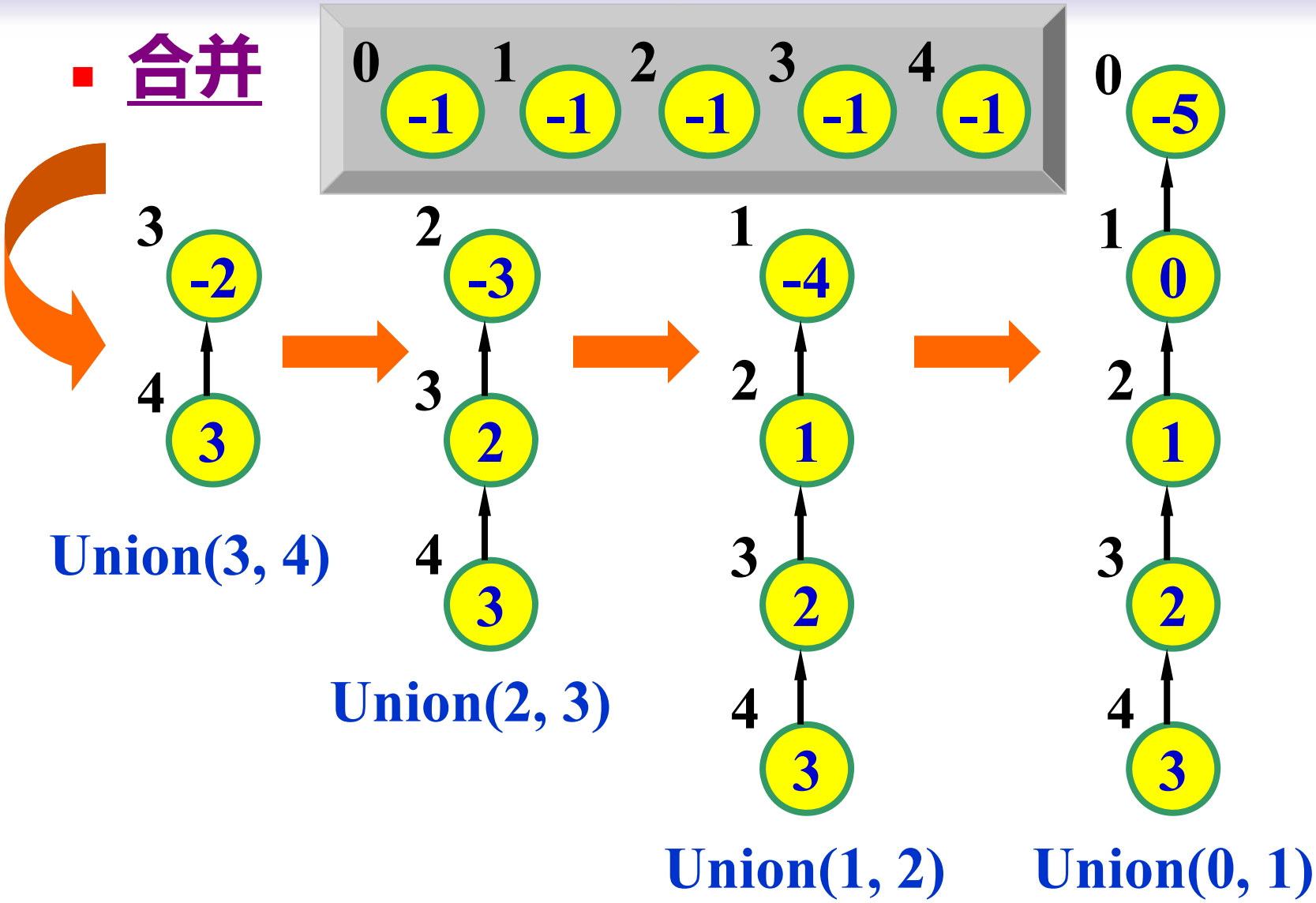



```
int UFSets :: Find ( int x ) {
    if ( parent [x] < 0 ) return x;
    else return Find ( parent [x] );
}
```

```
void UFSets :: Union ( int Root1, int Root2 ) {
//求两个不相交集合 Root1 与 Root2 的并
    parent[Root1] += parent[Root2];
    parent[Root2] = Root1;
    //将 Root2 连接到 Root1 下面
}
```

- **Find和Union操作性能不好。**
  - **假设最初 n 个元素构成 n 棵树组成的森林，parent[i] = -1。做处理Union(n-2, n-1), …, Union(1, 2), Union(0, 1)后，将产生退化的树。**

- **合并**

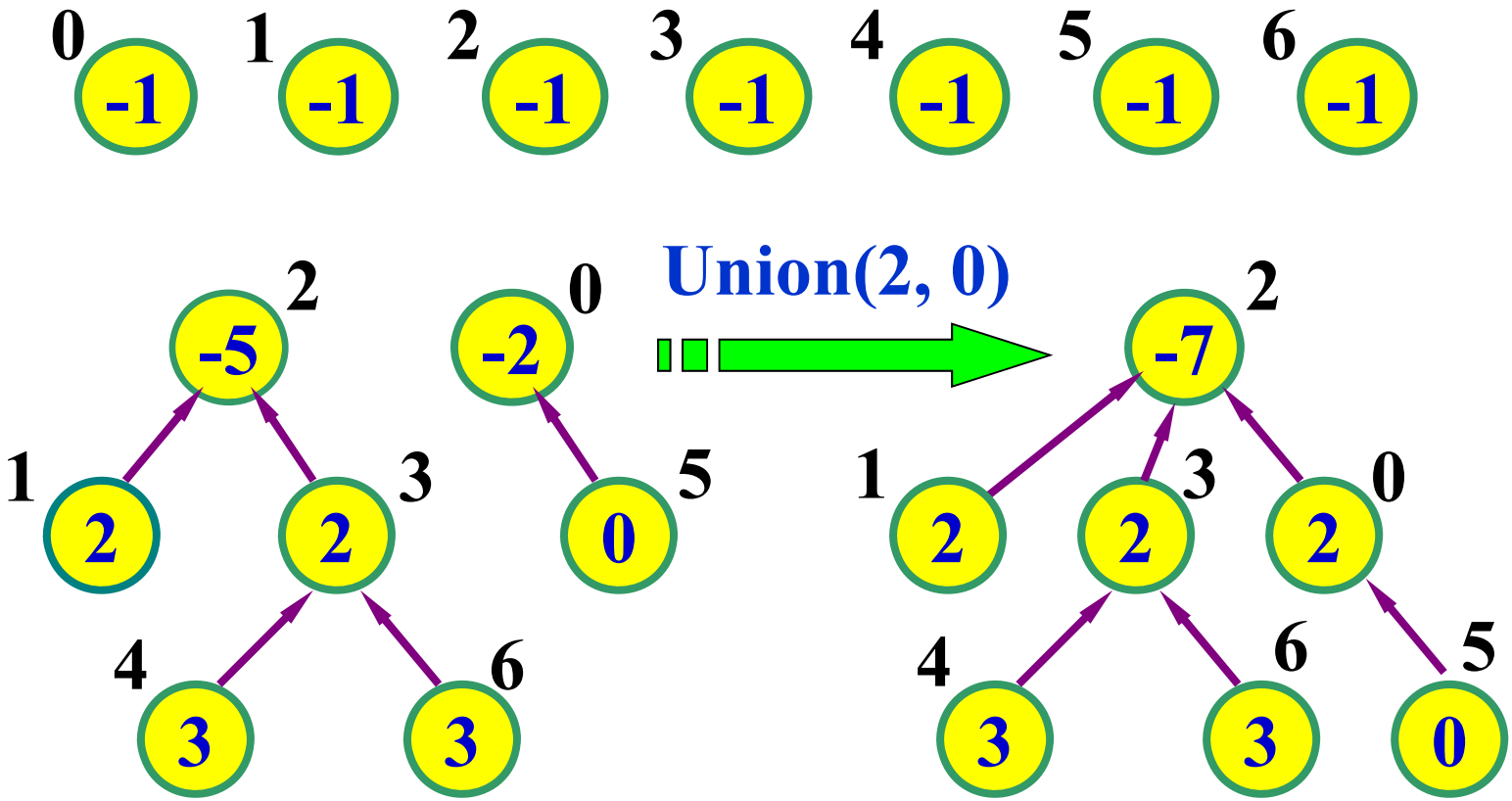

0 1 2 3 4
-1 -1 -1 -1 -1
3 -2
4 3
Union(3, 4)
2 -3
3 2
4 3
Union(2, 3)
1 -4
2 1
3 2
4 3
Union(1, 2)
0 -5
1 0
2 1
3 2
4 3
Union(0, 1)

- 执行一次Union操作所需时间是O(1)，n-1次Union操作所需时间是O(n)。
- 若再执行Find(0), Find(1), …, Find(n-1)，若被搜索的元素为i，完成Find(i)操作需要时间为O(i)，完成n次搜索需要的总时间将达到

$$O(\sum_{i=1}^{n} i) = O(n^2)$$

- 改进方法
  - 按树的结点个数合并
  - 按树的高度合并
  - 压缩元素的路径长度

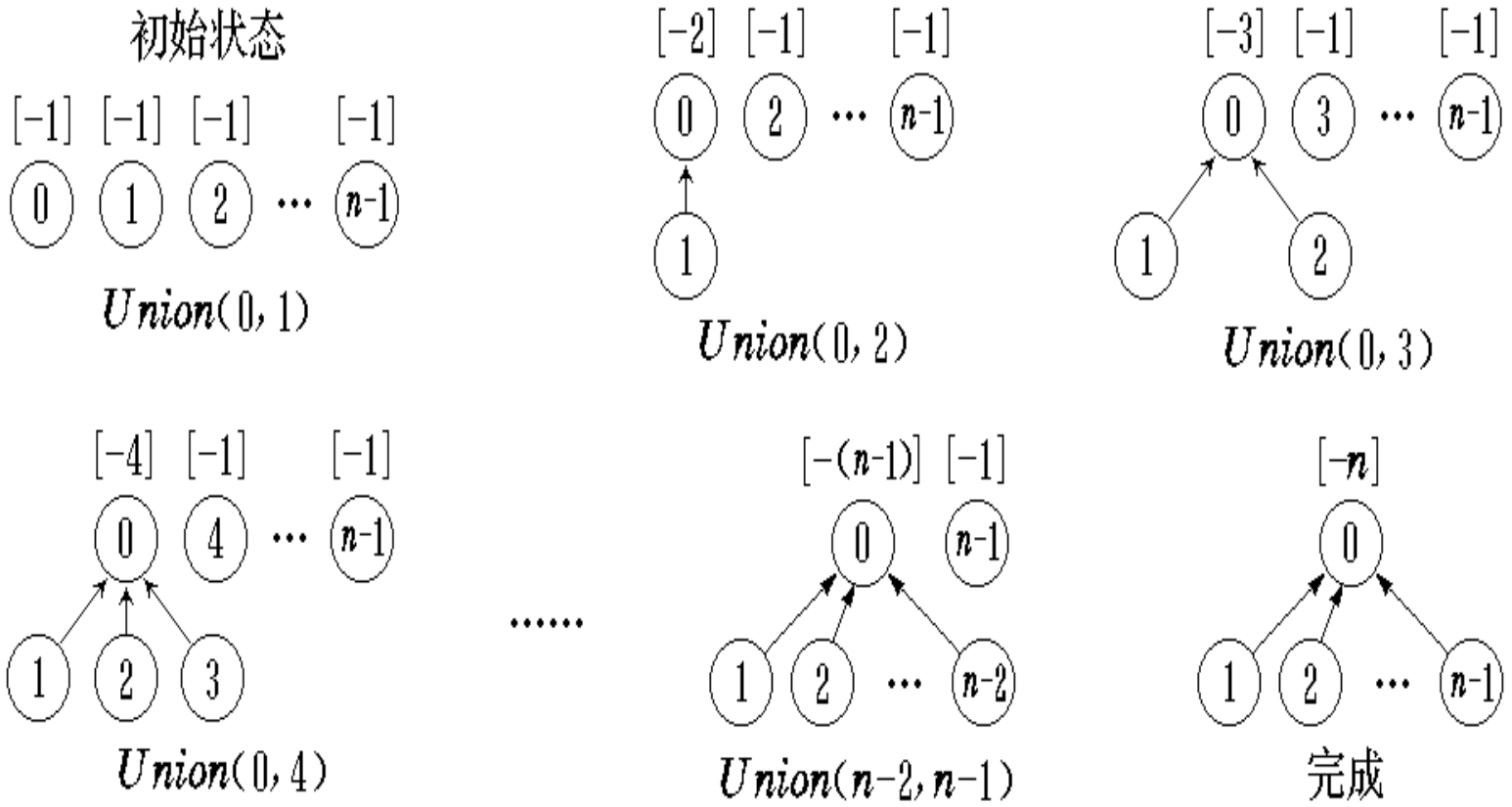
- **按树结点个数合并**
  - 结点个数多的树的根结点作根。



*Union*操作的加权规则

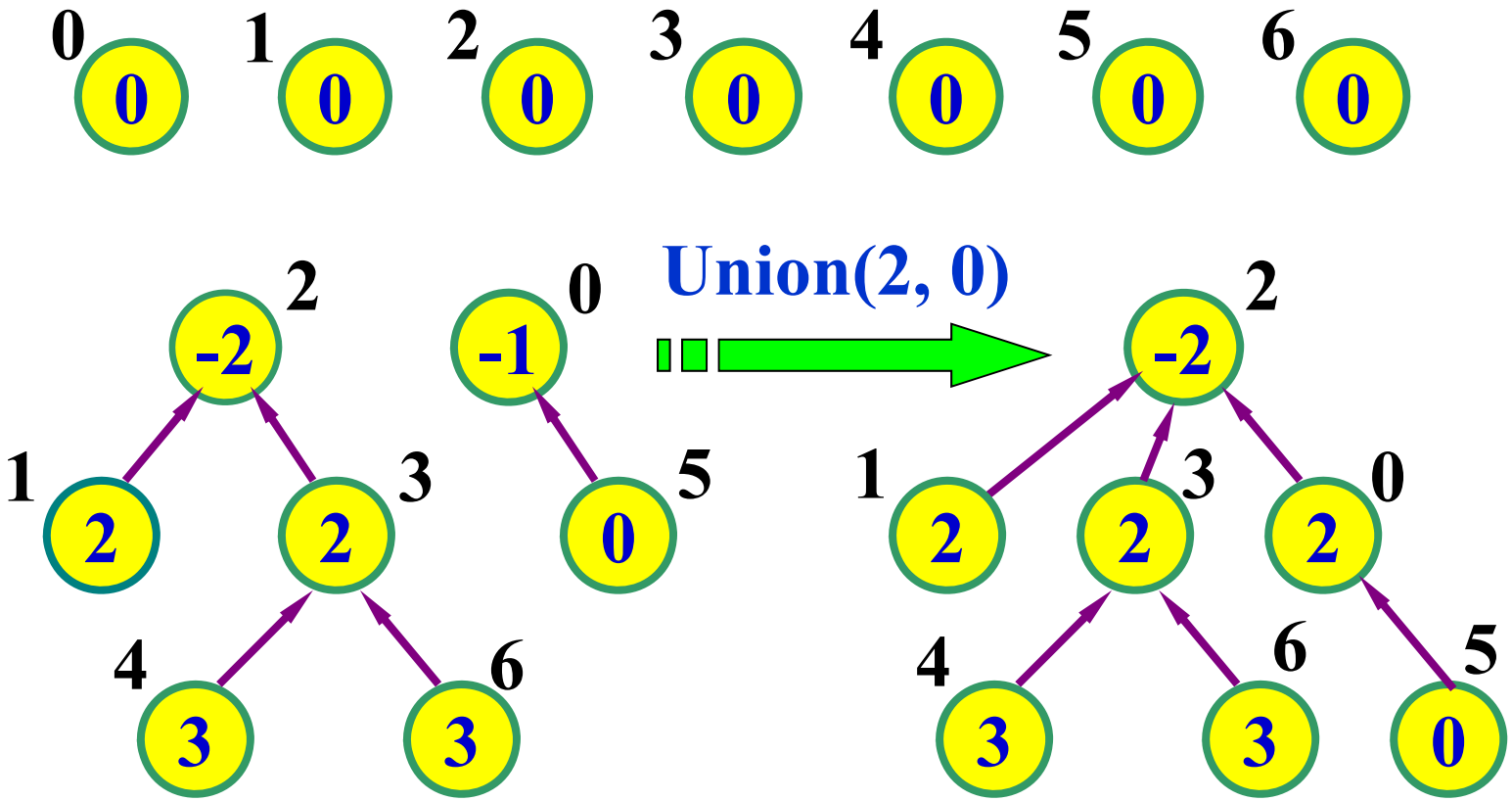
- 为避免产生退化的树，改进方法是先判断两集合中元素的个数。
  - 如果以 i 为根的树中的结点个数少于以 j 为根的树中的结点个数，即 parent[i] > parent[j]，则让 j 成为 i 的双亲；
  - 否则，让 i 成为 j 的双亲。

```
void UFSets :: WeightedUnion ( int Root1, int Root2 ) {
//按 Union 的加权规则改进的算法
    int temp = parent[Root1] + parent[Root2];
    if ( parent[Root2] < parent[Root1] ) {
      parent[Root1] = Root2;  // Root2 中结点数多
      parent[Root2] = temp;  // Root1 指向 Root2
    }
    else {
      parent[Root2] = Root1;  // Root1 中结点数多
      parent[Root1] = temp;  // Root2 指向 Root1
    }
}
```


初始状态
[-1] [-1] [-1] [-1]
0 1 2 … n-1
Union(0, 1)
[-2] [-1] [-1]
0 2 … n-1
1
Union(0, 2)
[-3] [-1] [-1]
0 3 … n-1
1 2
Union(0, 3)
[-4] [-1] [-1]
0 4 … n-1
1 2 3
Union(0, 4)
……
[-(n-1)] [-1]
0 n-1
1 2 … n-2
Union(n-2, n-1)
[-n]
0
1 2 … n-1
完成

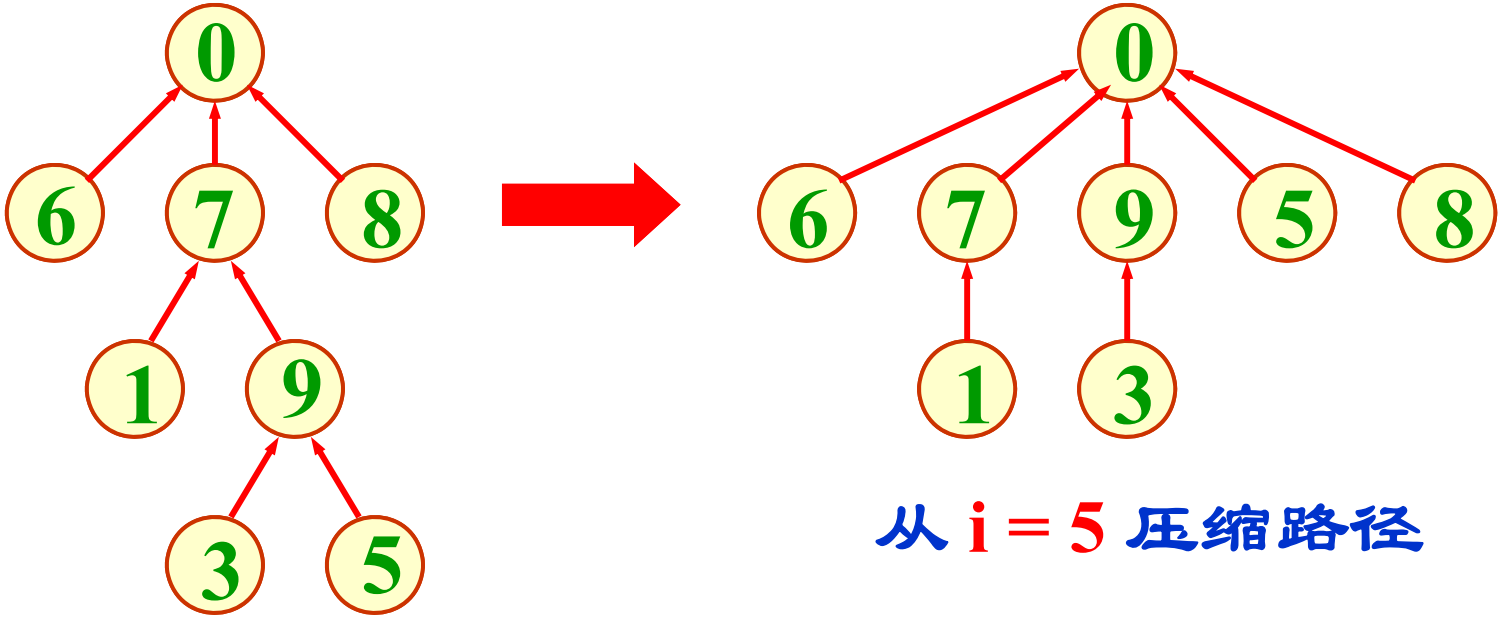
使用加权规则得到的树

- **按树高度合并**
  - 高度高的树的根结点作根。



Union操作的折叠规则

- 为进一步改进树的性能，可以使用如下折叠规则来“压缩路径”。即：如果 j 是从 i 到根的路径上的一个结点，且 parent[j]≠root[i]，则把 parent[j] 置为 root[i]。

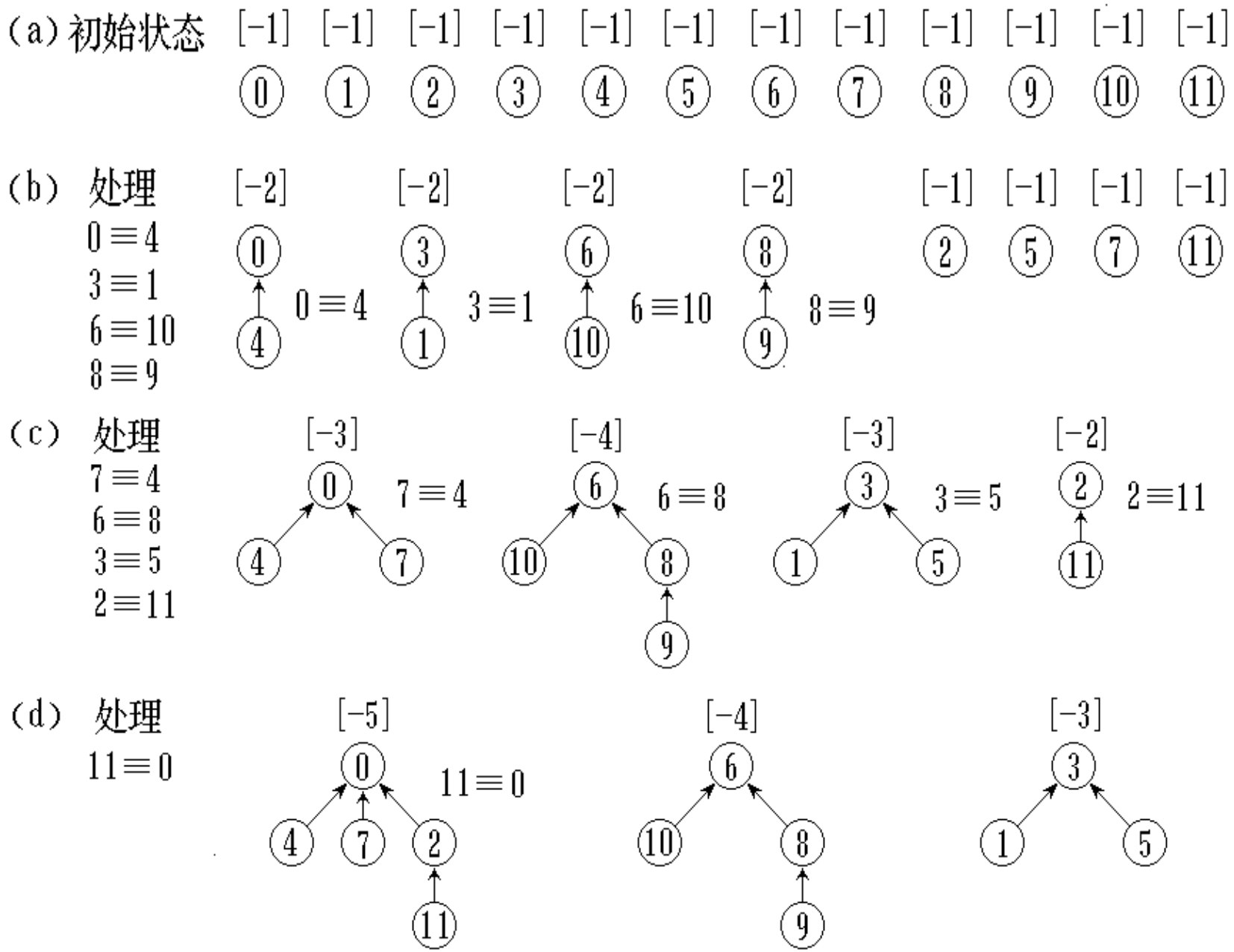


```
int UFSets :: CollapsingFind ( int i ) {
//使用折叠规则的搜索算法
    for ( int j = i; parent[j] >= 0; j = parent[j] );
    //让 j 循双亲指针走到根
    while ( parent[i] != j ) {  //换 parent[i] 到 j
        int temp = parent[i];
        parent[i] = j;  i = temp;    }
    return j;
}
```

使用折叠规则完成单个搜索，所需时间大约增加一倍。但是，能减少在最坏情况下完成一系列搜索操作所需的时间。

使用并查集处理等价对形成等价类的过程

- 最初，全部 n 个元素各自在自己的等价类中，parent[i] = -1；
- 以后，每处理一个等价对(i ≡ j)，先要确定 i 和 j 所在的集合，如果两个集合不同，则取其并集，否则不做任何事情。


(a) 初始状态
[-1] [-1] [-1] [-1] [-1] [-1] [-1] [-1] [-1] [-1] [-1] [-1]
0 1 2 3 4 5 6 7 8 9 10 11
(b) 处理
0≡4
3≡1
6≡10
8≡9
[-2] [-2] [-2] [-2] [-1] [-1] [-1] [-1]
0 3 6 8 2 5 7 11
4 1 10 9
0≡4 3≡1 6≡10 8≡9
(c) 处理
7≡4
6≡8
3≡5
2≡11
[-3] [-4] [-3] [-2]
0 6 3 2
4 7 10 8 1 5 11 9
7≡4 6≡8 3≡5 2≡11
(d) 处理
11≡0
[-5] [-4] [-3]
0 6 3
4 7 2 10 8 1 5 11 9
11≡0

6.5 散列 (Hashing)

- 在现实中经常遇到按给定的值进行查询的事例。为此，必须考虑在记录的存放位置和用以标识它的数据项（称为关键码）之间的对应关系，选择适当的数据结构，很方便地根据记录的关键码检索到对应记录的信息。
- 表项的存放位置及其关键码之间的对应关系可以用一个二元组表示：

  （关键码*key*，表项位置指针*addr*）
- 这个二元组构成搜索某一指定表项的索引项。
- 考虑到搜索效率，可以考虑散列表结构。

静态散列方法

- 散列方法在表项存储位置与其关键码之间建立一个确定的对应函数关系*Hash*( )，使每个关键码与结构中一个唯一存储位置相对应：

$$Address = Hash(Rec.key)$$

- 在搜索时，先对表项的关键码进行函数计算，把函数值当做表项的存储位置，在结构中按此位置取表项比较。若关键码相等，则搜索成功。在存放表项时，依相同函数计算存储位置，并按此位置存放。此方法称为散列方法。

- 在散列方法中使用的转换函数叫做**散列函数**。按此方法构造出来的表或结构就叫做**散列表**。
- 使用散列方法进行搜索不必进行多次关键码的比较，搜索速度比较快，可以直接到达或逼近具有此关键码的表项的实际存放地址。
- **散列函数是一个压缩映象函数**。关键码集合比散列表地址集合大得多，因此有可能经过散列函数的计算，把不同的关键码映射到同一个散列地址上，这就产生了**冲突**。
- **示例：有一组表项，其关键码分别是**

  12361，07251，03309，30976

采用的散列函数是：

$$hash(x) = x \% 73 + 13420$$

其中，“%”是除法取余操作。

则有：$hash(12361) = hash(07250) = hash(03309) = hash(30976) = 13444$。就是说，对不同的关键码，通过散列函数的计算，得到同一散列地址。我们称这些产生冲突的散列地址相同的不同关键码为同义词。

- 由于关键码集合比地址集合大得多，冲突很难避免。所以对于散列方法，需要讨论以下两个问题：

- 对于给定的一个关键码集合，选择一个计算 简单且地址分布比较均匀的散列函数，避免或尽量减少冲突；
- 拟订解决冲突的方案。

散列函数

构造散列函数时的几点要求：

- 散列函数应是简单的，能在较短的时间内计算出结果。
- 散列函数的定义域必须包括需要存储的全部关键码，如果散列表允许有 m 个地址时，其值域必须在 0 到 m-1 之间。

- 散列函数计算出来的地址应能均匀分布在整个地址空间中：若 *key* 是从关键码集合中随机抽取的一个关键码，散列函数应能以同等概率取 0 到 *m*-1 中的每一个值。

① 直接定址法

此类函数取关键码的某个线性函数值作为散列地址：

$$Hash(key) = a * key + b \quad \{a, b\text{为常数}\}$$

这类散列函数是一对一的映射，一般不会产生冲突。但是，它要求散列地址空间的大小与关键码集合的大小相同。

示例——有一组关键码如下：{ 942148, 941269, 940527, 941630, 941805, 941558, 942047, 940001 }。

散列函数为

$$Hash(key) = key - 940000$$

Hash (942148) = 2148　　*Hash* (941269) = 1269

Hash (940527) = 527　　*Hash* (941630) = 1630

Hash (941805) = 1805　　*Hash* (941558) = 1558

Hash (942047) = 2047　　*Hash* (940001) = 1

可以按计算出的地址存放记录。

② 数字分析法

- 设有 n 个 d 位数，每一位可能有 r 种不同的符号，这 r 种不同符号在各位上出现的频率不一定相同。可根据散列表的大小， 选取其中各种符号分布均匀的若干位作为散列地址。
- 计算各位数字中符号分布均匀度 λ_k 的公式：

$$\lambda_k = \sum_{i=1}^{r} (\alpha_i^k - n/r)^2$$

其中，α_i^k 表示第 i 个符号在第 k 位上出现的次数，n/r 表示各种符号在 n 个数中均匀出现的期望值。计算出的 λ_k 值越小，表明在该位（第 k 位）各种符号分布得越均匀。

9	4	2	1	4	8
9	4	1	2	6	9
9	4	0	5	2	7
9	4	1	6	3	0
9	4	1	8	0	5
9	4	1	5	5	8
9	4	2	0	4	7
9	4	0	0	0	1
①	②	③	④	⑤	⑥

①位，$\lambda_1 = 57.60$

②位，$\lambda_2 = 57.60$

③位，$\lambda_3 = 17.60$

④位，$\lambda_4 = 5.60$

⑤位，$\lambda_5 = 5.60$

⑥位，$\lambda_6 = 5.60$

若散列表地址范围有 3 位数字，取各关键码的 ④⑤⑥ 位做为记录的散列地址。

- 数字分析法仅适用于事先明确知道表中所有关键码每一位数值的分布情况，它完全依赖于关键码集合。如果换一个关键码集合，选择哪几位要重新决定。

③ 除留余数法

- 设散列表中允许地址数为 m，取一个不大于 m，但最接近于或等于 m 的质数 p 作为除数，利用以下函数把关键码转换成散列地址：

$$hash(key) = key \% p \quad p \le m$$

其中，“%”是整数除法取余的运算，要求这时的质数 p 不是接近 2 的幂。

- 示例：有一个关键码 $key = 962148$， 散列表大小 $m = 25$，即 $HT[25]$。取质数 $p = 23$。散列函数 $hash\ (key) = key\ \%\ p$，则散列地址为：

  $hash\ (962148) = 962148\ \%\ 23 = 12$

  可以按计算出的地址存放记录。需要注意的是，使用上面的散列函数计算出来的地址范围是 0 到 22。因此，从 23 到 24 这几个散列地址实际上在一开始是不可能用散列函数计算出来的，只可能在处理冲突时达到这些地址。

④ 平方取中法

- 此方法在词典处理中使用十分广泛。
- 它先计算构成关键码的标识符的内码的平方，然后按照散列表的大小取中间的若干位作为散列地址。
- 设标识符可以用一个计算机字长的内码表示。因为内码平方数的中间几位一般是由标识符所有字符决定，所以对不同的标识符计算出的散列地址大多不相同。
- 在平方取中法中，一般取散列地址为 2 的某次幂。例如，若散列地址总数取为 $m = 8^r$，则对内码的平方数取中间的 r 位。如果 $r = 3$，所取得的散列地址参看图的最右一列。

标识符	内码	内码的平方	散列地址
A	01	01	001
*A*1	0134	20420	042
*A*9	0144	23420	342
B	02	4	004
DMAX	04150130	215264436171 00	443
*DMAX*1	0415013034	526447352215 1420	352
AMAX	01150130	135423617100	236
*AMAX*1	0115013034	345424652215 1420	652

标识符的八进制内码表示及其平方值

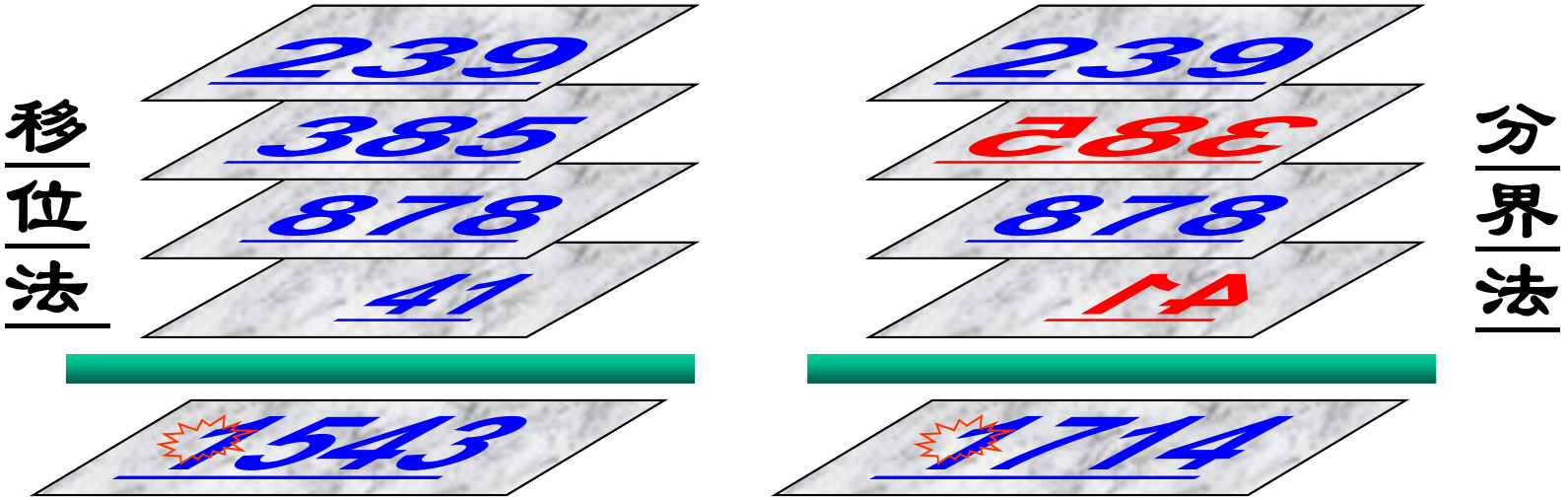
⑤ 折叠法

- 此方法把关键码自左到右分成位数相等的几部分，每一部分的位数应与散列表地址位数相同，只有最后一部分的位数可以短一些。
- 把这些部分的数据叠加起来，就可以得到具有该关键码的记录的散列地址。
- 有两种叠加方法：
  - ❖ 移位法 — 把各部分的最后一位对齐相加；
  - ❖ 分界法 — 各部分不折断，沿各部分的分界来回折叠，然后对齐相加，将相加的结果当做散列地址。

- 示例：设给定的关键码为 $key = 23938587841$，若存储空间限定 3 位，则划分结果为每段 3 位。 上述关键码可划分为 4 段：

  239　　385　　878　　41

- 把超出地址位数的最高位删去，仅保留最低的3 位，做为可用的散列地址。



- 一般当关键码的位数很多，而且关键码每一位上数字的分布大致比较均匀时，可用这种方法得到散列地址。
- 以上介绍了几种常用的散列函数。在实际工作中应根据关键码的特点，选用适当的方法。有人曾用“轮盘赌”的统计分析方法对它们进行了模拟分析，结论是平方取中法最接近于“随机化”。

处理冲突的闭散列方法

因为任一种散列函数也不能避免产生冲突，因此选择好的解决冲突的方法十分重要。

为了减少冲突，对散列表加以改造。若设散列表 HT 有 m 个地址，将其改为 m 个桶。其桶号与散列地址一一对应，第 i ($0 \le i < m$) 个桶的桶号即为第 i 个散列地址。

每个桶可存放 s 个表项，这些表项的关键码互为同义词。如果对两个不同表项的关键码用散列函数计算得到同一个散列地址，就产生了冲突，它们可以放在同一个桶内的不同位置。

- 只有当桶内所有 s 个表项位置都放满表项后再加入表项才会产生溢出。
- 通常桶的大小 s 取的比较小，因此在桶内大多采用顺序搜索。
- 闭散列也叫做开地址法。在闭散列情形，所有的桶都直接放在散列表数组中。因此，每个桶只有一个表项 ($s = 1$)。
- 若设散列表中各桶的编址为 0 到 m-1，当要加入一个表项 R_2 时，用它的关键码 $R_2.key$ ，通过散列函数 $hash\ (\ R_2.key\ )$ 的计算，得到它的存放桶号 j 。

- 但在存放时发现此桶已被另一个表项 R_1 占据，发生了冲突，必须处理冲突。为此，需把 R_2 存放到表中“下一个”空桶中。 如果表未被装满，则在允许的范围内必定还有空桶。

(1) 线性探查法 (Linear Probing)

假设给出一组表项，它们的关键码为 Burke, Ekers, Broad, Blum, Attlee, Alton, Hecht, Ederly。采用的散列函数是：取其第一个字母在字母表中的位置。

Hash (*x*) = *ord* (*x*) - *ord* (‘*A*’)

//*ord* ( ) 是求字符内码的函数

- 可得 *Hash* (Burke) = 1　　*Hash* (Ekers) = 4
  Hash (Broad) = 1　　*Hash* (Blum) = 1
  Hash (Attlee) = 0　　*Hash* (Hecht) = 7
  Hash (Alton) = 0　　*Hash* (Ederly) = 4
- 设散列表 *HT*[26], *m* = 26。采用线性探查法处理冲突，则散列结果如图所示。

0	1	2	3	4
Attlee	Burke	Broad	Blum	Ekers
(1)	(1)	(2)	(3)	(1)

5	6	7	8	9
Alton	Ederly	Hecht		
(6)	(3)	(1)		

- 需要搜索或加入一个表项时，使用散列函数计算桶号：

$$H_0 = hash(key)$$

- 一旦发生冲突，在表中顺次向后寻找“下一个”空桶 H_i 的递推公式为：

$$H_i = (H_{i-1} + 1) \% m, \quad i = 1, 2, \ldots, m-1$$

即用以下的线性探查序列在表中寻找“下一个”空桶的桶号：

$$H_0 + 1, H_0 + 2, \ldots, m-1, 0, 1, 2, \ldots, H_0-1$$

亦可写成如下的通项公式：

$$H_i = (H_0 + i) \% m, \quad i = 1, 2, \ldots, m-1$$

- 当发生冲突时，探查下一个桶。当循环 *m*-1 次后就会回到开始探查时的位置，说明待查关键码不在表内，且表已满，不能再插入新关键码。
- 用平均搜索长度*ASL* (*Average Search Length*) 衡量散列方法的搜索性能。
- 根据搜索成功与否，又有搜索成功的平均搜索长度*ASLsucc*和搜索不成功的平均搜索长度*ASLunsucc*之分。
- 搜索成功的平均搜索长度 *ASLsucc* 是指搜索到表中已有表项的平均探查次数，是找到表中各个已有表项的探查次数的平均值。

- 搜索不成功的平均搜索长度 *ASLunsucc* 是指在表中搜索不到待查的表项，但找到插入位置的平均探查次数。它是表中所有可能散列到的位置上要插入新元素时为找到空桶的探查次数的平均值。
- 在使用线性探查法对示例进行搜索时，搜索成功的平均搜索长度为：

$$ASL_{succ}=\frac{1}{8}\sum_{i=1}^{8}C_i=\frac{1}{8}(1+1+2+3+1+6+3+1)=\frac{18}{8}$$

- 搜索不成功的平均搜索长度为：

$$ASL_{unsucc}=\frac{9+8+7+6+5+4+3+2+18}{26}=\frac{62}{26}$$

- 下面是用线性探查法在散列表 *HT* 中搜索给定值 *x* 的算法。如果查到某一个 *j*，使得

  ht[j].info == Active && ht[j].Element == x

  则搜索成功；否则搜索失败。造成失败的原因可能是表已满，或者是原来有此表项但已被删去，或者是无此表项且找到空桶。

```
class HashTable {
//用线性探查法处理冲突时散列表类的定义
public:
    enum KindOfEntry { Active, Empty, Deleted };
    HashTable ( ) : TableSize ( DefaultSize )
                    { AllocateHt ( );  CurrentSize = 0; }
```

```
    ~HashTable ( ) { delete [ ] ht; }  //析构函数
    const HashTable & operator =  //表复制
                          ( const HashTable &ht2 );
    int Find ( const Type &x );  //搜索
    int Insert ( const Type &x );  //插入
    int Remove ( const Type &x );  //删除
    int IsIn ( const Type &x )  //判存在
        { return ( i =  Find (x) ) >= 0 ? 1 : 0; }
    void MakeEmpty ( );  //置空
private:
    struct HashEntry {  //散列表表项
      Type Element;  //表项关键码
      KindOfEntry info;  //三种状态
```

```
    int operator == ( HashEntry & );  //判相等
    int operator != ( HashEntry & );  //判不等
    HashEntry ( ) : info (Empty) { }  //构造函数
    HashEntry ( const Type &E, KindOfEntry
          i = Empty ) : Element (E), info (i) { }
    enum { DefualtSize = 11 };
    HashEntry *ht;  //散列表数组
    int CurrentSize, TableSize;  //当前及最大桶数
    void AllocateHt ( )  //分配空间
          { ht = new HashEntry [TableSize]; }
    int FindPos ( const Type &x ) const;
};  //散列函数
```

```
template <class Type> int HashTable <Type> ::
Find ( const Type &x ) {
//线性探查法的搜索算法，函数返回找到位置
//若返回负数可能是空位，若为 -TableSize 则失败
  int i =  FindPos ( x ),  j = i;  //计算散列地址
  while ( ht[j].info != Empty &&
                        ht[j].Element != x ) {
     j = ( j + 1 ) % TableSize;  //冲突，找空桶
     if ( j == i ) return -TableSize;  //失败，表满
  }
  if ( ht[j].info == Active ) return j;  //成功
  else -j;  //失败
}
```

- 在利用散列表进行各种处理之前，必须首先将散列表中原有的内容清掉，只需将表中所有表项的 info 域置为 Empty 即可。
- 散列表存放的表项不应有重复的关键码。在插入新表项时，如果发现表中已经有关键码相同的表项，则不再插入。
- 在闭散列情形下不能真正删除表中已有表项，删除表项会影响其他表项的搜索。若把关键码为 Broad 的表项真正删除，把它所在位置的info 域置为 Empty ，以后在搜索关键码为 Blum 和 Alton 的表项时就查不下去，会错误地判断表中没有关键码为 Blum 和 Alton 的表项。

- 若想删除一个表项，只能给它做一个删除标记 deleted 进行逻辑删除，不能把它真正删去。
- 逻辑删除的副作用是：在执行多次删除后，表面上看起来散列表很满，实际上有许多位置没有利用。

```
template <class Type>
void HashTabe <Type> :: MakeEmpty ( ) {
//置表中所有表项为空
    for ( int i = 0; i < TableSize; i++)
        ht[i].info = Empty;
    CurrentSize = 0;
}
```

```
template <class Type> const HashTable <Type> &
HashTable <Type> ::
operator = ( const HashTable <Type> &ht2 ) {
//重载函数：从散列表 ht2 复制到当前散列表
    if ( this != &ht2 ) {
       delete [ ] ht;
       TableSize = ht2.TableSize;  AllocateHt ( );
       for ( int i = 0; i < TableSize; i++ )
            ht[i] = ht2.ht[i];
       CurrentSize = ht2.CurrentSize;
    }
    return *this;
}
```

```
template <class Type> int HashTable <Type> ::
Insert ( const Type &x ) {
//将新表项 x 插入到当前的散列表中
    if ( ( int i = Find (x) ) >= 0 ) return 0;  //不插入
    else if ( i != -TableSize && ht[-i].info != Active )
          {  //在 -i 处插入 x
              ht[-i].Element = x;
               ht[-i].info = Active;
               CurrentSize++;
               return 1;
          }
          else return 0;
}
```

```
template <class Type> int HashTable <Type> ::
Remove ( const Type &x ) {
//在当前散列表中删除表项 x
   if ( ( int i =  Find (x) ) >= 0 ) {  //找到，删除
      ht[i].info = deleted;   //做删除标记
      CurrentSize--;
      return 1;
   }
   else return 0;
}
```

线性探查方法容易产生“堆积”，不同探查序列的关键码占据可用的空桶，为寻找某一关键码需要经历不同的探查序列，导致搜索时间增加。

算法分析

- 设散列表的**装填因子**为 $\alpha = n/(s*m)$，其中 n 是表中已有的表项个数，s 是每个桶中最多可容纳表项个数，m 是表中的桶数。
- 可用 α **表明散列表的装满程度**。α 越大，表中表项数越多，表装得越满，发生冲突可能性越大。
- 通过对线性探查法的分析可知，为搜索一个关键码所需进行的探查次数的期望值 P 大约是 $(2-\alpha)/(2-2\alpha)$ 。虽然平均探查次数较小，但在最坏情况下的探查次数会相当大。

(2) 二次探查法 (Quadratic Probing)

- 为改善“堆积”问题，减少为完成搜索所需的平均探查次数，可使用二次探查法。
- 通过某一个散列函数对表项的关键码 x 进行计算，得到桶号，它是一个非负整数。

$$H_0 = hash(x)$$

二次探查法在表中寻找“下一个”空桶的公式：

$$H_i = (H_0 + i^2) \% m,$$

$$H_i = (H_0 - i^2) \% m, \quad i = 1, 2, \ldots, (m-1)/2$$

式中的 m 是表的大小，它应是一个值为 $4k+3$ 的质数，其中 k 是一个整数。如 3, 7, 11, 19, 23, 31, 43, 59, 127, 251, 503, ….

- 探查序列如 $H_0, H_0+1, H_0-1, H_0+4, H_0-4, \ldots$。
  在做 $(H_0 - i^2) \% m$ 的运算时，当 $(H_0 - i^2) < 0$ 时，运算结果也是负数。实际算式可改为

  $j = (H_0 - i^2) \% m$, if $(j < 0)\ j += m$
- 示例：给出一组关键码 { Burke, Ekers, Broad, Blum, Attlee, Alton, Hecht, Ederly }。

  散列函数为：$Hash(x) = ord(x) - ord('A')$

  用它计算可得

  $Hash$ (Burke) = 1　　$Hash$ (Ekers) = 4

  $Hash$ (Broad) = 1　　$Hash$ (Blum) = 1

  $Hash$ (Attlee) = 0　　$Hash$ (Hecht) = 7

  $Hash$ (Alton) = 0　　$Hash$ (Ederly) = 4

- 因为可能桶号是 0 ~ 25，取满足 $4k+3$ 的质数，表的长度为 $TableSize = 31$，利用二次探查法得到的散列结果如图所示。

0	1	2	3	4	5
Blum	Burke	Broad		Ekers	Ederly
(3)	(1)	(2)		(1)	(2)

6	7	8	9	10	11
	Hecht				
	(1)				

25	26	27	28	29	30
		Alton			Attlee
		(5)			(3)

利用二次探查法处理溢出

- 使用二次探查法处理冲突时的搜索成功的平均搜索长度为：

$$ASL_{succ}=\frac{1}{8}\sum_{i=1}^{8}C_i=\frac{1}{8}(3+1+2+1+2+1+5+3)=\frac{18}{8}$$

- 搜索不成功的平均搜索长度为：

$$ASL_{unsucc}=\frac{1}{26}(6+5+2+3+2+2+20)=\frac{40}{26}$$

设散列表桶数为 m，待查关键码为 x，第一次通过散列函数计算出的桶号为 $H_0=hash(x)$。当发生冲突时，第 i-1 次和第 i 次计算出来的“下一个”桶号分别为：

$$H_{i-1}^{(0)} = (H_0 + (i-1)^2) \% m,$$
$$H_{i-1}^{(1)} = (H_0 - (i-1)^2) \% m.$$
$$H_i^{(0)} = (H_0 + i^2) \% m,$$
$$H_i^{(1)} = (H_0 - i^2) \% m.$$

相减，可以得到：

$$H_i^{(0)} - H_{i-1}^{(0)} = (2*i-1) \% m,$$
$$H_i^{(1)} - H_{i-1}^{(1)} = (-2*i+1) \% m.$$

从而

$$H_i^{(0)} = (H_{i-1}^{(0)} + 2*i-1) \% m,$$
$$H_i^{(1)} = (H_{i-1}^{(1)} - 2*i+1) \% m.$$

- 只要知道上一次的桶号$H_{i-1}^{(0)}$和$H_{i-1}^{(1)}$，当 i 增加 1 时可以从 $H_{i-1}^{(0)}$和$H_{i-1}^{(1)}$简单地导出$H_i^{(1)}$ 和$H_i^{(1)}$，不需要每次计算 i 的平方。
- 在冲突处理算法 Find 中,首先求出 H_0 作为当前桶号 CurrentPos ，当发生冲突时求“下一个”桶号，$i=1$ 。
- 此时用一个标志 odd 控制是加 i^2 还是减 i^2 。
  - 若 $odd==0$ 加 i^2，并置 $odd=1$ ；
  - 若 $odd==1$ 减 i^2，并置 $odd=0$ 。
- 下次 i 进一后，又可由 odd 控制先加后减。

处理冲突的算法

```
template <class Type> int HashTable <Type> ::
Find ( const Type &x ) {
    int pos0, pos1, i = 0, odd = 0;
    int CurrentPos = pos0 = pos1 = HashPos (x);
    //初始桶号
    while ( ht[CurrentPos].info != Empty &&
                ht[CurrentPos].Element != x ) {  //冲突
       if ( !odd ) {  // odd == 0 加 i²
          pos0 = ( pos0 + 2*i-1) % TableSize;
         CurrentPos = pos0;  odd = 1;  }
       else {  // odd == 1 减 i²
```

```
            pos1 = ( pos1-2*i+1 ) % TableSize;
            CurrentPos = pos1;  odd = 0;
        }
    }
    LastFindOK = ( ht[CurrentPos].info == Active );
    return CurrentPos;
}
```

可以证明，当表的长度TableSize为质数且表的装填因子 α 不超过 0.5 时，新的表项 x 一定能够插入，且任何一个位置不会被探查两次。只要表中至少有一半空，就不会有表满问题。

- 在搜索时可以不考虑表满的情况，但在插入时必须确保表的装填因子 α 不超过 0.5 。如果超出，必须将表长度扩充一倍，进行表的分裂。
- 在删除一个表项时，为确保搜索链不致中断，也只能做表项的逻辑删除，即将被删表项的标记info改为Deleted。

(3) 双散列法

- 使用双散列方法时，需要两个散列函数。
- 第一个散列函数 *Hash* ( ) 按表项的关键码 *key* 计算表项所在的桶号 $H_0 = Hash\ (key)$ 。

- 一旦冲突，利用第二个散列函数 *ReHash* ( ) 计算该表项到达“下一个”桶的移位量。它的取值与 *key* 的值有关，要求它的取值应是小于地址空间大小 TableSize，且与 TableSize 互质的正整数。
- 若设表的长度为 m = TableSize，则在表中寻找“下一个”桶的公式为：

$$j = H_0 = Hash\ (key),\ p = ReHash\ (key);$$

$$j = (j + p)\ \%\ m;$$

p 是小于 m 且与 m 互质的整数

- 利用双散列法，按一定的距离，跳跃式地寻找“下一个”桶，减少了“堆积”的机会。
- 双散列法的探查序列也可写成：

$$H_i = (H_0 + i * ReHash(key)) \% m,$$

$$i = 1, 2, \ldots, m-1$$

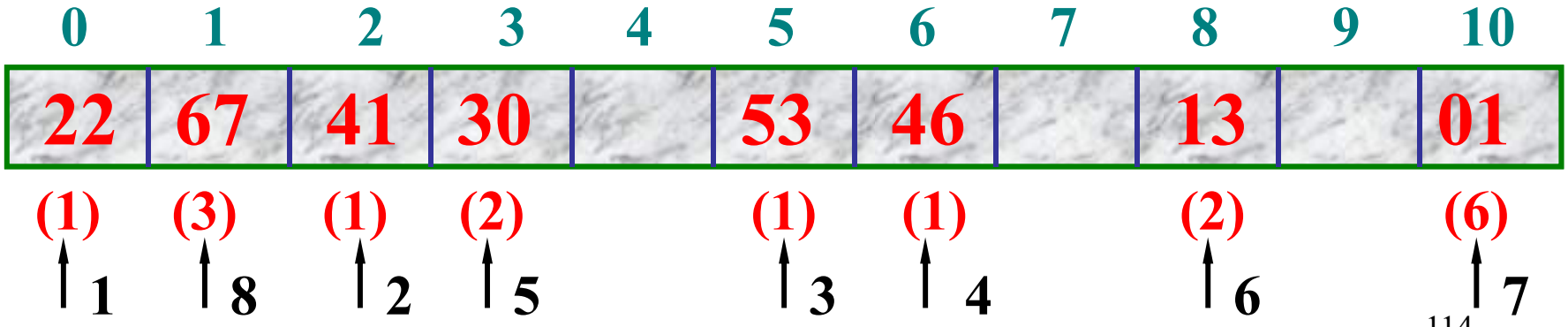
- 最多经过 $m-1$ 次探查，它会遍历表中所有位置，回到 H_0 位置。
- 示例：给出一组表项关键码{ 22, 41, 53, 46, 30, 13, 01, 67 }。散列函数为：

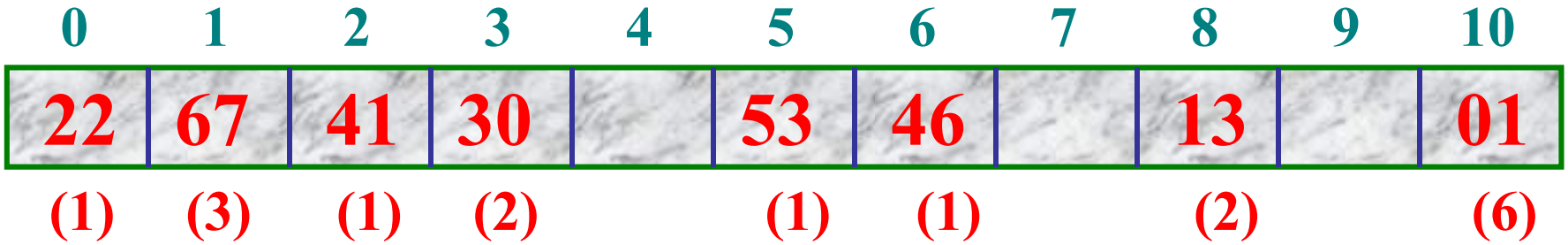
$$Hash(x) = (3x) \% 11。$$

- 散列表为 $HT[0..10]$，$m = 11$。因此，再散列函数为 $ReHash(x) = (7x) \% 10 + 1$。

$H_i = (H_{i-1} + (7x) \% 10 + 1) \% 11, \ i = 1, 2, \ldots$

- $H_0(22) = 0$ $H_0(41) = 2$ $H_0(53) = 5$
  $H_0(46) = 6$ $H_0(30) = 2$ 冲突 $H_1 = (2+1) = 3$
  $H_0(13) = 6$ 冲突 $H_1 = (6+2) = 8$
  $H_0(01) = 3$ 冲突 $H_1 = (3+8) = 0$ 冲突 $H_2 = (0+8) = 8$ 冲突 $H_3 = (8+8) = 5$ 冲突 $H_4 = (5+8) = 2$ 冲突 $H_5 = (2+8) = 10$
  $H_0(67) = 3$ 冲突 $H_1 = (3+10) = 2$ 冲突 $H_2 = (2+10) = 1$





- 搜索成功的平均搜索长度

$$ASL_{succ} = \frac{1}{8}(1+3+1+2+1+1+2+6) = \frac{17}{8}$$

- 搜索不成功的平均搜索长度
  - 每一散列位置的移位量有 10 种：1, 2, …, 10 。先计算每一散列位置各种移位量情形下找到下一个空位的比较次数，求出平均值；
  - 再计算各个位置的平均比较次数的总平均值。

- *Rehash* ( )的取法很多。例如，当 *m* 是质数时，可定义
  - $ReHash(key) = key \% (m-2) + 1$
  - $ReHash(key) = \lfloor key / m \rfloor \% (m-2) + 1$
- 当 *m* 是 2 的方幂时，*ReHash*(*key*) 可取从 0 到 *m*-1 中的任意一个奇数。

处理冲突的开散列方法 — 链地址法

- 开散列方法首先对关键码集合用某一个散列函数计算它们的存放位置。
- 若设散列表地址空间的所有位置是从 0 到 m-1，则关键码集合中的所有关键码被划分为 m 个子集，具有相同地址的关键码归于同一子集。我们称同一子集中的关键码互为同义词，每一个子集称为一个桶。
- 通常各个桶中的表项通过一个单链表链接起来，称之为同义词子表。所有桶号相同的表项都链接在同一个同义词子表中，各链表的表头结点组成一个向量。

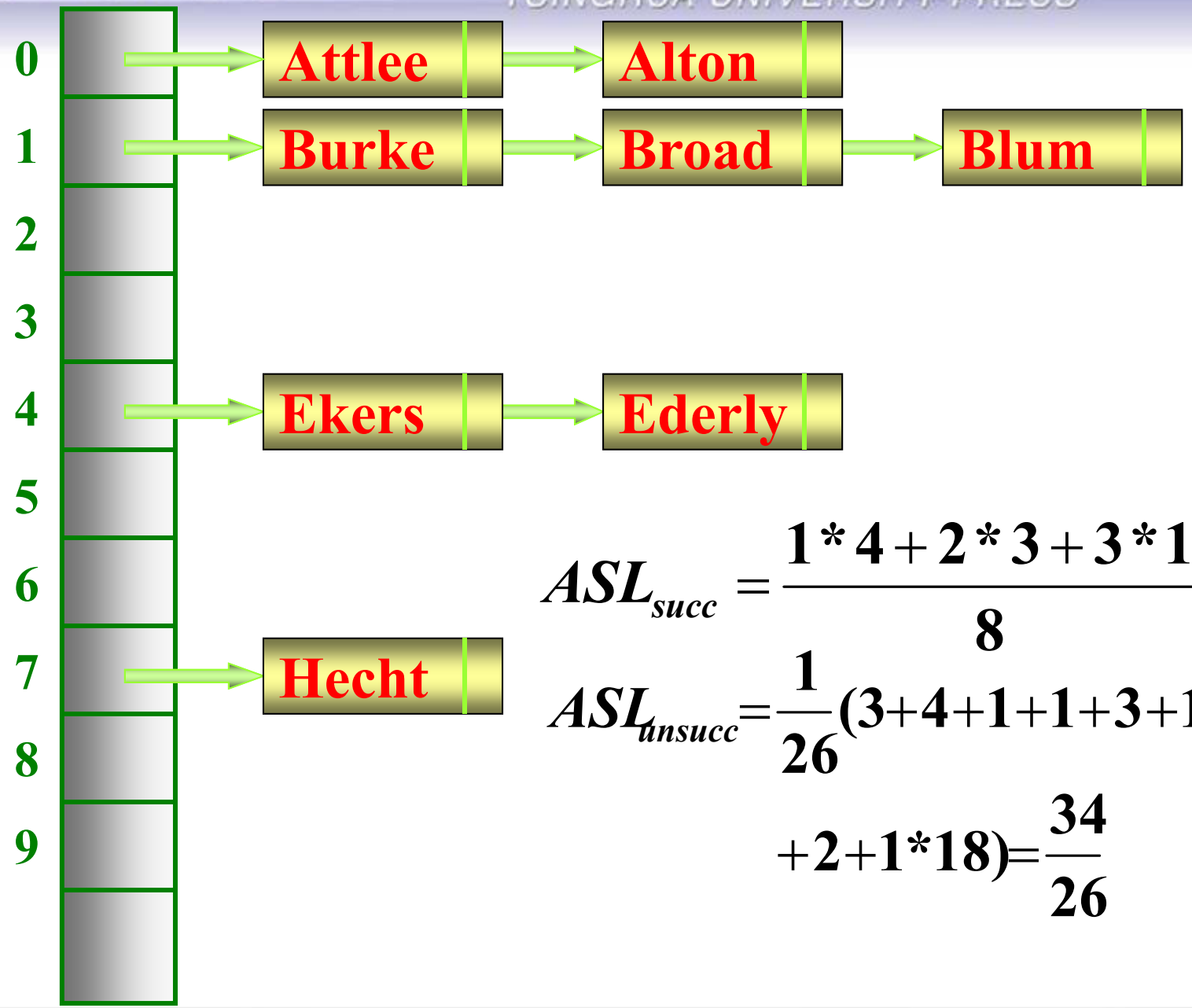
- 向量的元素个数与桶数一致。桶号为 *i* 的同义词子表的表头结点是向量中的第 *i* 个元素。
- 示例：给出一组表项关键码 { Burke, Ekers, Broad, Blum, Attlee, Alton, Hecht, Ederly }。散列函数为：*Hash* (*x*) = *ord* (*x*) - *ord* ('*A*')。
- 用它计算可得：

  Hash (Burke) = 1　　*Hash* (Ekers) = 4

  Hash (Broad) = 1　　*Hash* (Blum) = 1

  Hash (Attlee) = 0　　*Hash* (Hecht) = 7

  Hash (Alton) = 0　　*Hash* (Ederly) = 4
- 散列表为 *HT*[0..25]，*m* = 26。



$$ASL_{succ} = \frac{1*4+2*3+3*1}{8} = \frac{13}{8}$$

$$ASL_{unsucc} = \frac{1}{26}(3+4+1+1+3+1+1+$$

$$+2+1*18) = \frac{34}{26}$$

- 通常，每个桶中的同义词子表都很短，设有 n 个关键码通过某一个散列函数，存放到散列表中的 m 个桶中。那么每一个桶中的同义词子表的平均长度为 n/m。以搜索平均长度为 n/m 的同义词子表代替了搜索长度为 n 的顺序表，搜索速度快得多。

利用链地址法处理溢出时的类定义

```
template <class Type> class ListNode {  //链结点
friend HashTable;
private:
    Type key;  //关键码
    ListNode *link;  //链指针  };
```

```
typedef ListNode <Type> *ListPtr;

class HashTable {  //散列表的类定义
public:
    HashTable( int size = defaultsize )  //构造函数
      { buckets = size;  ht = new ListPtr [buckets]; }
private:
    int buckets;  //桶数
    ListPtr <Type> *ht;  //散列表数组的头指针
};
```

循链搜索的算法

```
template <class Type> Type * HashTable <Type> ::
Find ( const Type &x ) {
    int j = HashFunc ( x, buckets );  //初始桶号
    ListPtr <Type> *p = ht[j];  //桶地址
    while ( p != NULL )
        if ( p->key == x ) return &p->key;
        else p = p->link;
    return 0;
}
```

- 其它如插入、删除操作可参照单链表的插入、删除等算法来实现。
- 应用开散列法处理冲突，需要增设链接指针，似乎增加了存储开销。事实上，由于闭散列法必须保持大量的空闲空间以确保搜索效率，如二次探查法要求装填因子 $\alpha \leq 0.5$ ，而表项所占空间又比指针大得多，所以使用开散列法反而比闭散列法节省存储空间。

散列表分析

- 散列表是一种直接计算记录存放地址的方法，它在关键码与存储位置之间直接建立了映象。

- 散列表是一种直接计算记录存放地址的方法，它在关键码与存储位置之间直接建立了映象。
- 当选择的散列函数能够得到均匀的地址分布时，在搜索过程中可以不做多次探查。
- 由于很难避免冲突，增加了搜索时间。冲突的出现，与散列函数的选取（地址分布是否均匀），处理冲突的方法（是否产生堆积）有关。
- 在实际应用中使用关键码进行散列时，如在用作关键码的许多标识符具有相同的前缀或后缀，或者是相同字符的不同排列的场合，不同的散列函数往往导致散列表具有不同的搜索性能。

下图给出一些实验结果

$\alpha = n / m$	0.50		0.75		0.90		0.95	
散列函数种类	开散列法	闭散列法	开散列法	闭散列法	开散列法	闭散列法	开散列法	闭散列法
平方取中	1.26	1.73	1.40	9.75	1.45	310.14	1.47	310.53
除留余数	1.19	4.52	1.31	10.20	1.38	22.42	1.41	25.79
移位折叠	1.33	21.75	1.48	65.10	1.40	710.01	1.51	118.57
分界折叠	1.39	22.97	1.57	48.70	1.55	69.63	1.51	910.56
数字分析	1.35	4.55	1.49	30.62	1.52	89.20	1.52	125.59
理论值	1.25	1.50	1.37	2.50	1.45	5.50	1.48	10.50

搜索关键码时所需对桶的平均访问次数

从图中可以看出，开散列法优于闭散列法；在散列函数中，用除留余数法作散列函数优于其它类型的散列函数，最差的是折叠法。

- 当装填因子 α 较高时，选择的散列函数不同，散列表的搜索性能差别很大。在一般情况下多选用除留余数法，其中的除数在实用上应选择不含有 20 以下的质因数的质数。
- 对散列表技术进行的实验评估表明，它具有很好的平均性能，优于一些传统的技术，如平衡树。但散列表在最坏情况下性能很不好，如果对一 个有 n 个关键码的散列表执行一次搜索或插入操作，最坏情况下需要 $O(n)$ 的时间。
- Knuth对不同的冲突处理方法进行了概率分析。

- 若设 α 是散列表的装填因子：

$$\alpha = \frac{\text{表中已装有记录数}}{\text{表中预设的最大记录数}} = \frac{n}{m}$$

- 用地址分布均匀的散列函数 $Hash(\ )$ 计算桶号。
- S_n 是搜索一个随机选择的关键码 x_i $(1 \le i \le n)$ 所需的关键码比较次数的期望值。
- U_n 是在长度为 m 的散列表中 n 个桶已装入表项的情况下，装入第 $n+1$ 项所需执行的关键码比较次数期望值。
- 前者称为在 $\alpha = n/m$ 时的搜索成功的平均搜索长度，后者称为在 $\alpha = n/m$ 时的搜索不成功的平均搜索长度。

- **用不同的方法溢出处理冲突时散列表的平均搜索长度如图所示。**

处理冲突的方法		平均搜索长度 *ASL*	
		搜索成功 S_n	搜索不成功 U_n（登入新记录）
闭散列法	线性探查法	$\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$	$\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$
	伪随机探查法 二次探查法 双散列法	$-\left(\frac{1}{\alpha}\right)\log_e(1-\alpha)$	$\frac{1}{1-\alpha}$
开散列法（同义词子表法）		$1+\frac{\alpha}{2}$	$\alpha+e^{-\alpha}\approx\alpha$

- 散列表的装填因子 α 表明了表中的装满程度。越大，说明表越满，再插入新元素时发生冲突的可能性就越大。
- 散列表的搜索性能，即平均搜索长度依赖于散列表的装填因子，不直接依赖于 n 或 m 。
- 不论表的长度有多大，我们总能选择一个合适的装填因子，以把平均搜索长度限制在一定范围内。

例：求散列表大小并设计散列函数。

- 设有一个含 200 个表项的散列表，用二次探查法解决冲突，按关键码查询时找到一个新表项插入位置的平均探查次数不超过 1.5，则散列表项应能够至少容纳多少个表项。再设计散列函数（设搜索不成功的平均搜索长度为 $U_n = 1/(1-\alpha)$，其中 α 为装填因子）。
- 解答：设表中表项个数 $n = 200$，搜索不成功的平均搜索长度

$$U_n = 1/(1-\alpha) \le 1.5 \Rightarrow \alpha \le 1/3$$

$$\therefore \quad n/m = 200/m = \alpha \le 1/3, \ m \ge 600$$

随堂练习

例1：使用散列函数hash f(x)=x%11，把一个整数值转换成散列表下标，现要把数据1, 13, 12, 34, 38, 33, 27, 22插入到散列表中。

(1) 使用线性探查再散列法来构造散列表。

(2) 使用链地址法来构造散列表。

(3) 针对这种情况，确定其查找成功所需的平均查找次数以及查找不成功所需的平均探查次数。

(1) 首先计算出每个数据的 hash 地址如下：

f(1)=1, f(13)=2, f(12)=1, f(34)=1, f(38)=5, f(33)=0, f(27)=5, f(22)=0

使用线性探查再散列法构造的散列表：

地址	0	1	2	3	4	5	6	7	8	9	10
数据	33	1	13	12	34	38	27	22			
探查成功次数	1	1	1	3	4	1	2	8			
探查不成功次数	9	8	7	6	5	4	3	2	1	1	1

(2) hash 表长度为 m，填入表中的数据个数为 n，则装填因子 α=n/m；因此有：α=8/11。

在等概率情况下，

使用线性探查再散列法查找成功情况时所需的平均查找次数为：

$ASL_{成功}=(1+1/(1-\alpha))/2=(1+1(1-8/11))/2=7/3$

使用线性探查再散列法查找不成功情况时所需的平均查找次数为：

$ASL_{不成功}=(1+1/(1-\alpha)^2)/2=(1+1(1-8/11)^2)/2=65/9$

使用链地址法查找成功情况时所需的平均查找次数为：

$ASL_{成功}=1+\alpha/2=1+8/22=15/11$

使用链地址法查找不成功情况时所需的平均查找次数为：

$ASL_{不成功}=\alpha+e^{-\alpha}=8/11+e^{-8/11}$

***注意，利用装填因子所计算的平均查找长度为近似值。

也可直接计算如下：

线性探查再散列法——

$ASL_{成功}=1/8*(1+1+1+3+4+1+2+8)=21/8$

$ASL_{不成功}=1/11(9+8+7+6+5+4+3+2+1+1+1)=47/11$

链地址法查找成功时为链表中找到该结点时的比较次数，查找不成功时必须比较到链尾的空指针处，因此有：

$ASL_{成功}=1/8*(1+2+1+3+1+1+2)=13/8$

$ASL_{不成功}=1/11(3+4+2+1+1+3+1+1+1+1+1)=19/11$

本章小结

- 知识点
  - 集合
  - 等价类与并查集
  - 散列表及其分析

- **课程习题**

  - **笔做题——6.5, 6.6, 6.9**
    (以作业形式提交)
  - **上机题——6.14, 6.15, 6.16**
  - **思考题——6.11, 6.12, 6.13, 6.21**