第六章 集合与字典

- 集合及其表示
- 并查集与等价类
- 字典
- 散列

集合及其表示

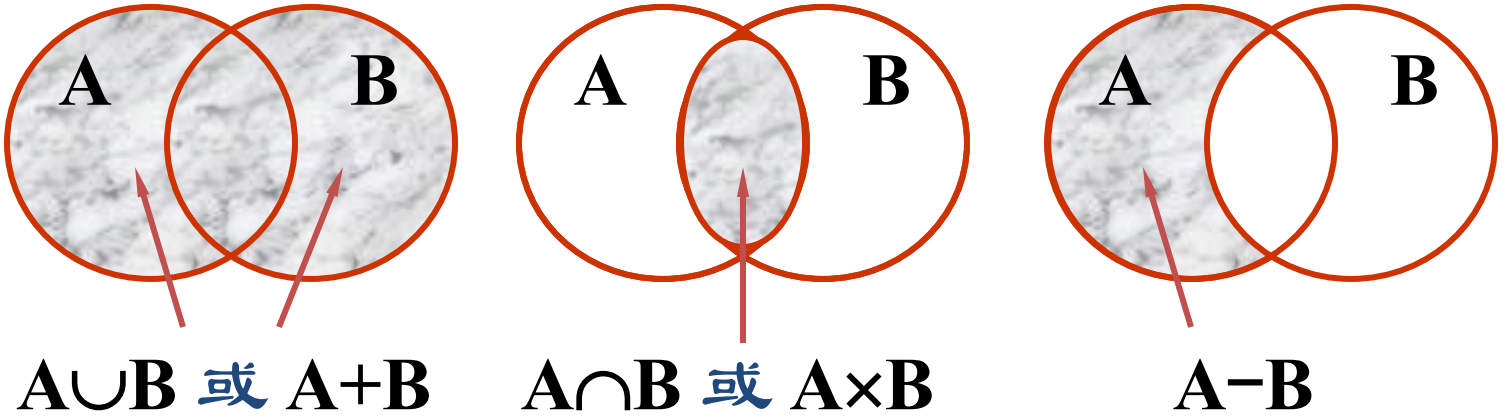
集合基本概念

- 集合是成员(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。
- 例： **colour = { red, orange, yellow, green, black, blue, purple, white }**

- 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 T>
class Set {
public:
    virtual Set() = 0;                    //构造函数
    virtual makeEmpty() = 0;              //置空集合
    virtual bool addMember (const T x) = 0;
```

```
    virtual bool delMember (const T x) = 0;
    virtual Set<T> intersectWith (Set<T>& R) = 0;
                        //集合的交运算
    virtual Set<T> unionWith (Set<T>& R) = 0;
                        //集合的并运算
    virtual Set<T> differenceFrom (Set<T>& R) = 0;
                        //集合的差运算
    virtual bool Contains (T x) = 0;
    virtual bool subSet (Set<T>& R) = 0;
    virtual bool operator == (Set<T>& R) = 0;
                        //判集合是否相等
};
```

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。
- 当全集合是由有限个可枚举的成员组成时，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。
- 一个二进位有两个取值1或0，分别表示在集合与不在集合。如果采用16位无符号短整数数组 bitVector[ ]作为集合的存储，就要考虑如何求出元素 i 在bitVector数组中的相应位置。

集合的位向量(bit Vector)类的定义

```
#include <assert.h>
#include <iostream.h>
const int DefaultSize = 50;
template <class T>
class bitSet {
//用位向量来存储集合元素, 集合元素的范围在0到
//setSize-1之间。 数组采用16位无符号短整数实现
public:
    bitSet (int sz = DefaultSize);          //构造函数
    bitSet (const bitSet<T>& R);            //复制构造函数
    ～bitSet () { delete [ ]bitVector; }    //析构函数
    unsigned short getMember (const T x);  //读取集合
元素x
```

```
void putMember (const T x, unsigned short v);  //将值v送入集合元素x
void makeEmpty () {                             //置空集合
    for (int i = 0; i < vectorSize; i++)
        bitVector[i] = 0;
}
bool addMember (const T x);          //加入新成员x
bool delMember (const T x);          //删除老成员x
bitSet<T>&  operator = (const bitSet<T>& R);
bitSet<T>&  operator + (const bitSet<T>& R);
bitSet<T>&  operator * (const bitSet<T>& R);
bitSet<T>&  operator - (const bitSet<T>& R);
bool Contains (const T x);
                                    //判x是否集合this的成员
```

```
    bool subSet (bitSet<T>& R);   //判this是否R的子集
    bool operator == (bitSet<T>& R);
                                  //判集合this与R相等
    friend istream& operator >> (istream& in,
        bitSet<T>& R);            //输入
    friend ostream& operator << (ostream& out,
        bitSet<T>& R);            //输出
private:
    int setSize;                  //集合大小
    int vectorSize;               //位数组大小
    unsigned short *bitVector;
                                  //存储集合元素的位数组
};
```

使用示例

```
bitSet<int> s1, s2, s3, s4, s5;  bool 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}
```

```
//s1: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
//s4: {7, 8, 9}

    index = s1.subSet (s4);   //判断s1是否为s4子集
    cout << index << endl;   //结果，  index = 0

    equal = (s1 == s2);              //集合s1与s2比较相等
    cout << equal << endl;   //为0, 两集合不等
```

构造函数的实现

```
template <class T>
bitSet<T>::bitSet (int sz) : setSize (sz) {    //构造函数
    assert (setSize > 0);                      //检查参数的合理性
    vectorSize = (setSize+15) >> 4;            //存储数组大小
    bitVector = new unsigned short[vectorSize];    //分配空间
    assert (bitVector != NULL);
                    //检查存储分配是否成功
    for (int i = 0; i < vectorSize; i++)
        bitVector[i] = 0;                      //初始化
};
```

```
template<class T>
 unsigned short bitSet<T>:: getMember (const T x) {
    //读取集合元素x，x从0开始
     int ad = x/16;
     int id = x%16;
     unsigned short elem = bitVector[ad];
     return ((elem >> (15 - id)) %2);
};
```

```
template<class T>
 void bitSet<T>:: putMember (const T x, unsigned
    short v) {  //将值v送入集合元素x
      int ad = x/16;
      int id = x%16;
      unsigned short elem = bitVector[ad];
      unsigned short temp = elem >> (15-id);
      elem = elem << (id+1);
      if (temp%2==0 && v ==1) temp = temp +1;
      else if (temp%2==1 && v==0) temp = temp -1;
      bitVector[ad] = (temp<<(15-id)) | (elem >> (id+1));
};
```

```
template<class T>
bitSet<T>&  bitSet<T>::          //求集合this与R的并
operator + (const bitSet<T>& R) {
assert (vectorSize == R.vectorSize);
    bitSet temp (vectorSize);

for (int i = 0; i < vectorSize; i++)
        temp.bitVector[i] = bitVector[i] | R.bitVector[i];

return temp;  //按位求"或", 由第三个集合返回
};
```

```
template <class T>
bitSet<T>&  bitSet<T>::                    //求集合this与R的交
operator * (const bitSet<T>& R) {
    assert (vectorSize == R.vectorSize);
    bitSet temp (setSize);

    for (int i = 0; i < vectorSize; i++)
        temp.bitVector[i] = bitVector[i] & R.bitVector[i];

    return temp;  //按位求“与”, 由temp返回
};
```

```
template <class T>
bitSet<T>&  bitSet<T>::          //求集合this与R的差
operator - (const bitSet<T>& R) {
    assert (vectorSize == R.vectorSize);
    bitSet temp (setSize);

    for (int i = 0; i < vectorSize; i++)
        temp.bitVector[i] =  bitVector[i] & (~R.bitVector[i]);

    return temp;                 //由第三个集合返回
};
```

集合的并

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	1	1	0	1	0	1	1	1	0

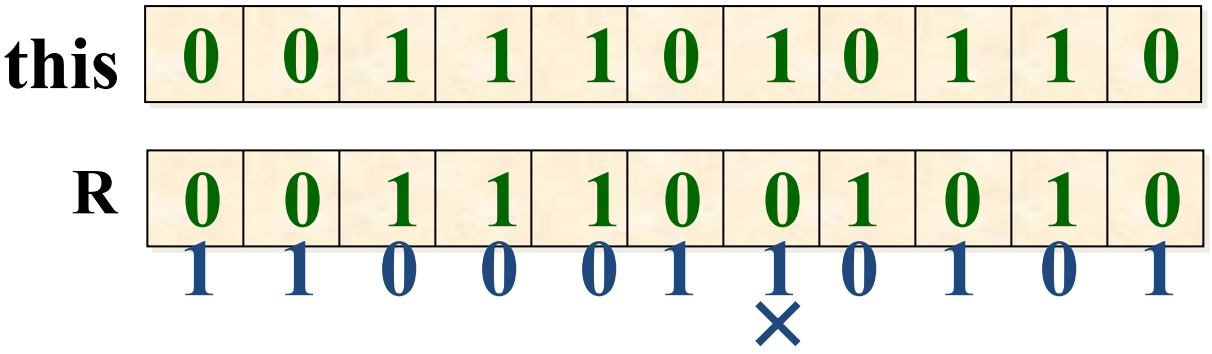
集合的交

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

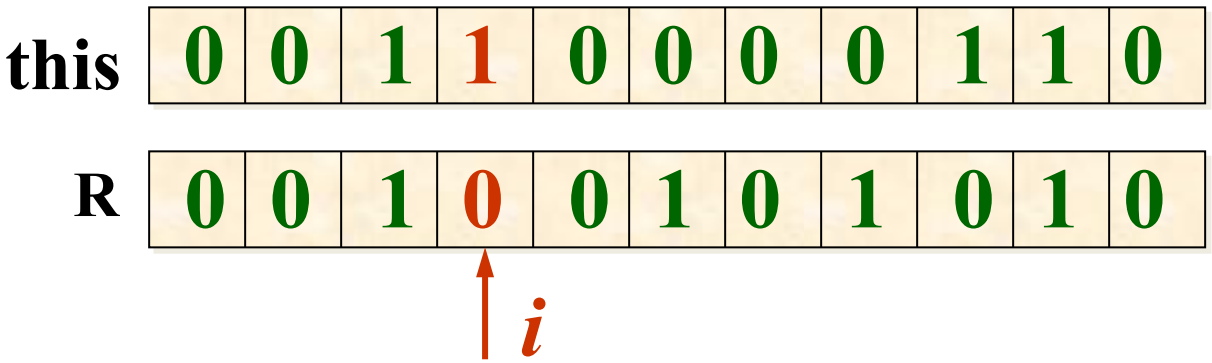
集合的差

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

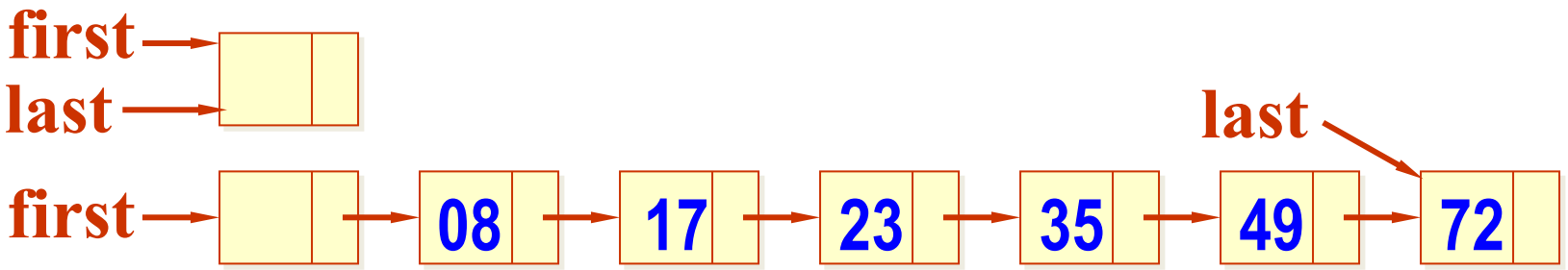
```
template <class T>
bool bitSet<T>::subSet (bitSet<T>& R) {
//判this是否R的子集
    assert (setSize == R.setSize);
    for (int i = 0; i < vectorSize; i++)   //按位判断
        if (bitVector[i] & (~R.bitVector[i])) return false;

   return true;
};
```



```
template <class T>
bool bitSet<T>::operator == (bitSet<T>& R) {
//判集合this与R相等
    if (vectorSize != R.vectorSize) return false;
    for (int i = 0; i < vectorSize; i++)
        if (bitVector[i] != R.bitVector[i])
            return false;
    return true;                          //对应位全部相等
};
```



用有序链表实现集合抽象数据类型



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。

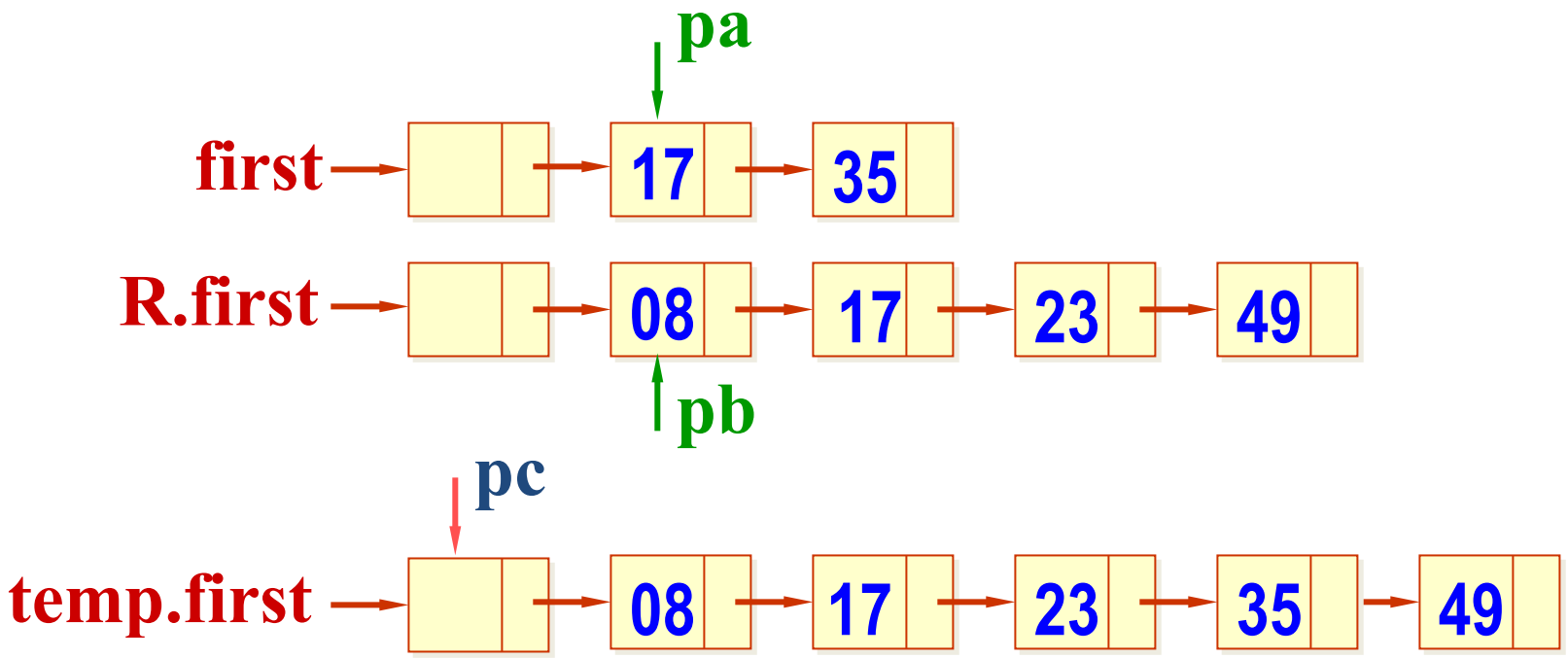
集合的有序链表类的定义

```
template <class T>
struct SetNode {                    //集合的结点类定义
    T data;                         //每个成员的数据
    SetNode<T> *link;               //链接指针
    SetNode() : link (NULL);                    //构造函数
    SetNode (const T& x, SetNode<T> *next = NULL)
        : data (x), link (next);                //构造函数
};
```

```
template <class T>
class LinkedSet {              //集合的类定义
  private:
      SetNode<T> *first, *last;
      //有序链表表头指针, 表尾指针
  public:
      LinkedSet () { first = last = new SetNode<T>; }
      LinkedSet (LinkedSet<T>& R);  //复制构造函数
      ～LinkedSet () { makeEmpty();  delete first; }
      void makeEmpty();                    //置空集合
      bool addMember (const T& x);
      bool delMember (const T& x);
      LinkedSet<T>& operator = (LinkedSet<T>& R);
      LinkedSet<T>& operator + (LinkedSet<T>& R);
```

```
    LinkedSet<T>& operator * (LinkedSet<T>& R);
    LinkedSet<T>& operator – (LinkedSet<T>& R);
    bool Contains (const T x);    //判x是否集合的成员
    bool operator == (LinkedSet<T>& R);
                                  //判集合this与R相等
    bool Min (T& x);        //返回集合最小元素的值
    bool Max (T& x);        //返回集合最大元素的值
    bool subSet (LinkedSet<T >& R);
                                  //判this是否R的子集
};
```

表示集合的几个重载函数




```
template <class T>
LinkedSet<T>&  LinkedSet<T>::
operator + (LinkedSet<T>& R) {
//求集合this与集合R的并
```

```
SetNode<T> *pb = R.first->link;   //R扫描指针
SetNode<T> *pa = first->link;     //this扫描指针
LinkedSet<T> temp;                //创建空链表
SetNode<T> *p, *pc = temp.first;  //结果存放指针
while (pa != NULL && pb != NULL) {
    if (pa->data == pb->data) {       //两集合共有
        pc->link = new SetNode<T>(pa->data);
        pa = pa->link;  pb = pb->link;
    }
    else if (pa->data < pb->data) {  //this元素值小
        pc->link = new SetNode<T>(pa->data);
        pa = pa->link;
    }
```

```
        else {                          //R集合元素值小
            pc->link = new SetNode<T>(pb->data);
            pb = pb->link;
        }
        pc = pc->link;
    }
    if (pa != NULL) p = pa;         //this集合未扫完
    else p = pb;                    //或R集合未扫完
    while (p != NULL) {             //向结果链逐个复制
        pc->link = new SetNode<T>(p->data);
        pc = pc->link;  p = p->link;
    }
    pc->link = NULL;  temp.last = pc;       //链表收尾
    return temp;
};
```

等价类与并查集

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，则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, …，它们的并就是 S。这些子集即为等价类。

确定等价类的方法

确定等价类的方法分两步走:

1. 读入并存储所有的等价对(i, j);

2. 标记和输出所有的等价类。

- 给定集合 $S = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$，及如下等价对：$0 \equiv 4$, $3 \equiv 1$, $6 \equiv 10$, $8 \equiv 9$, $7 \equiv 4$, $6 \equiv 8$, $3 \equiv 5$, $2 \equiv 11$, $11 \equiv 0$

- 进行归并的过程为：

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

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

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

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

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

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

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

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

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

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

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

并查集 (Union-Find Sets)

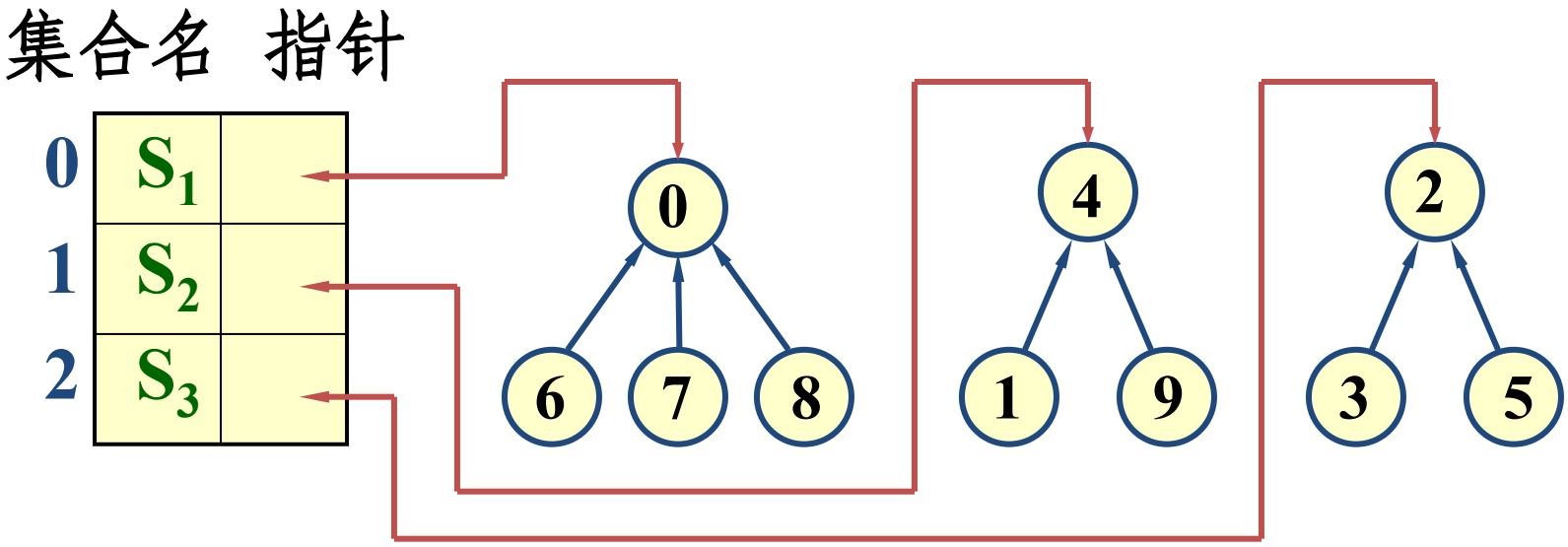
- 并查集支持以下三种操作:
  - **Union (Root1, Root2)** //合并操作
  - **Find (x)** //查找操作
  - **UFSets (s)** //构造函数
- 对于并查集来说，每个集合用一棵树表示。
- 为此，采用树的双亲表示作为集合存储表示。集合元素的编号从0到 $n-1$。其中 n 是最大元素个数。

- 在双亲表示中，第 *i* 个数组元素下标代表包含集合元素 *i* 的树结点。根结点的双亲为 **-k**，表示集合中的元素个数为**k**。
- 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:

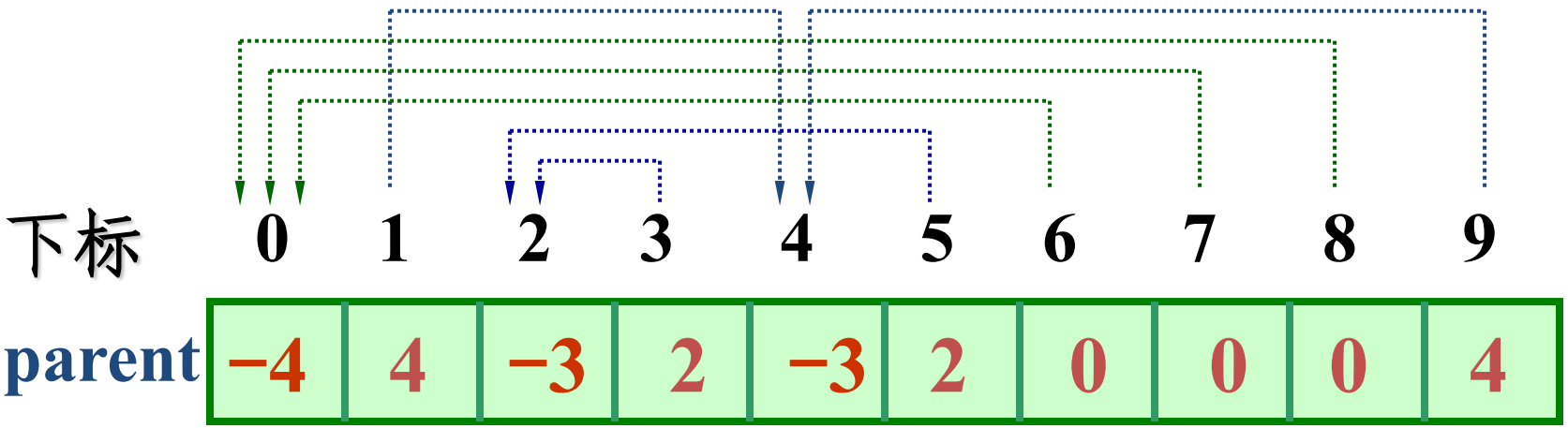
a) 集合元素到存放该元素名的树结点间的对应;

b) 集合名到表示该集合的树的根结点间的对应。

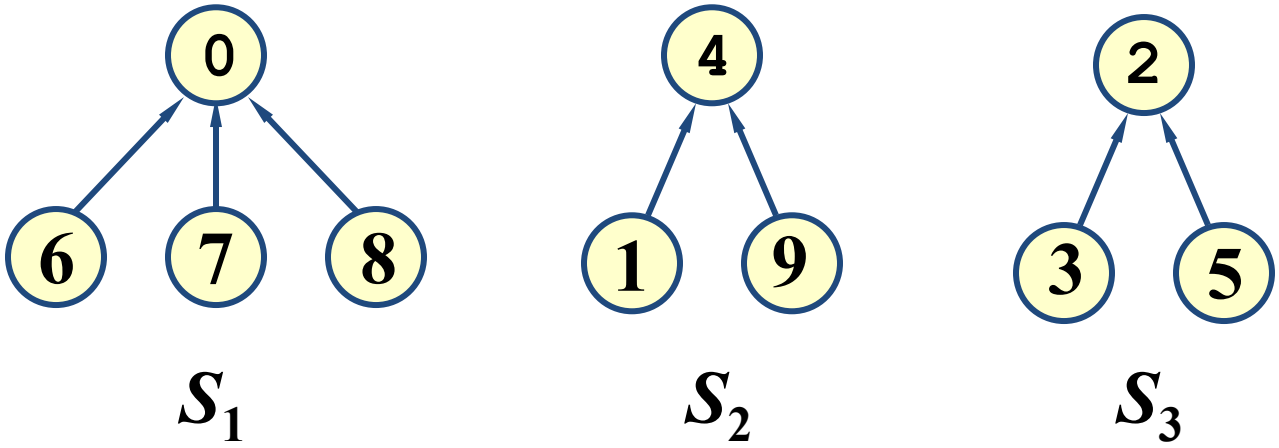
- 设 S_1= {0, 6, 7, 8}, S_2= {1, 4, 9}, S_3= {2, 3, 5}



- 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。



集合 S_1, S_2 和 S_3 的双亲表示



S_1 S_2 S_3

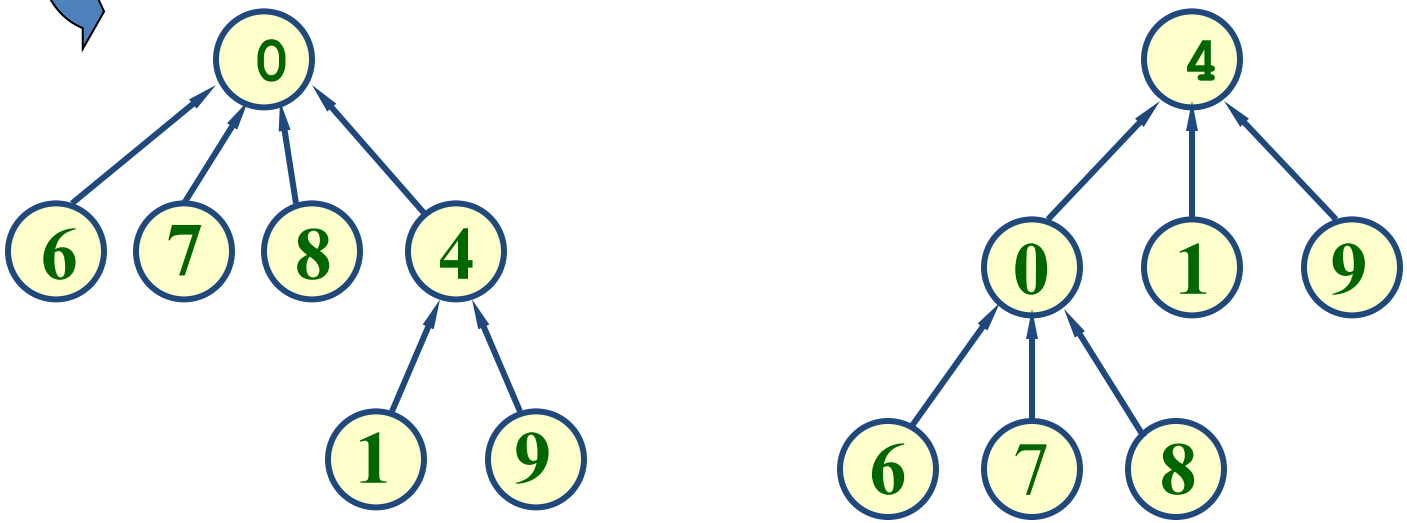
- 初始时，用构造函数 **UFSets(s)** 构造一个森林，每棵树只有一个结点，表示集合中各元素自成一个子集合。

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** 不在同一个集合中，可用 **Union(i, j)** 将它们合并到一个集合中。

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



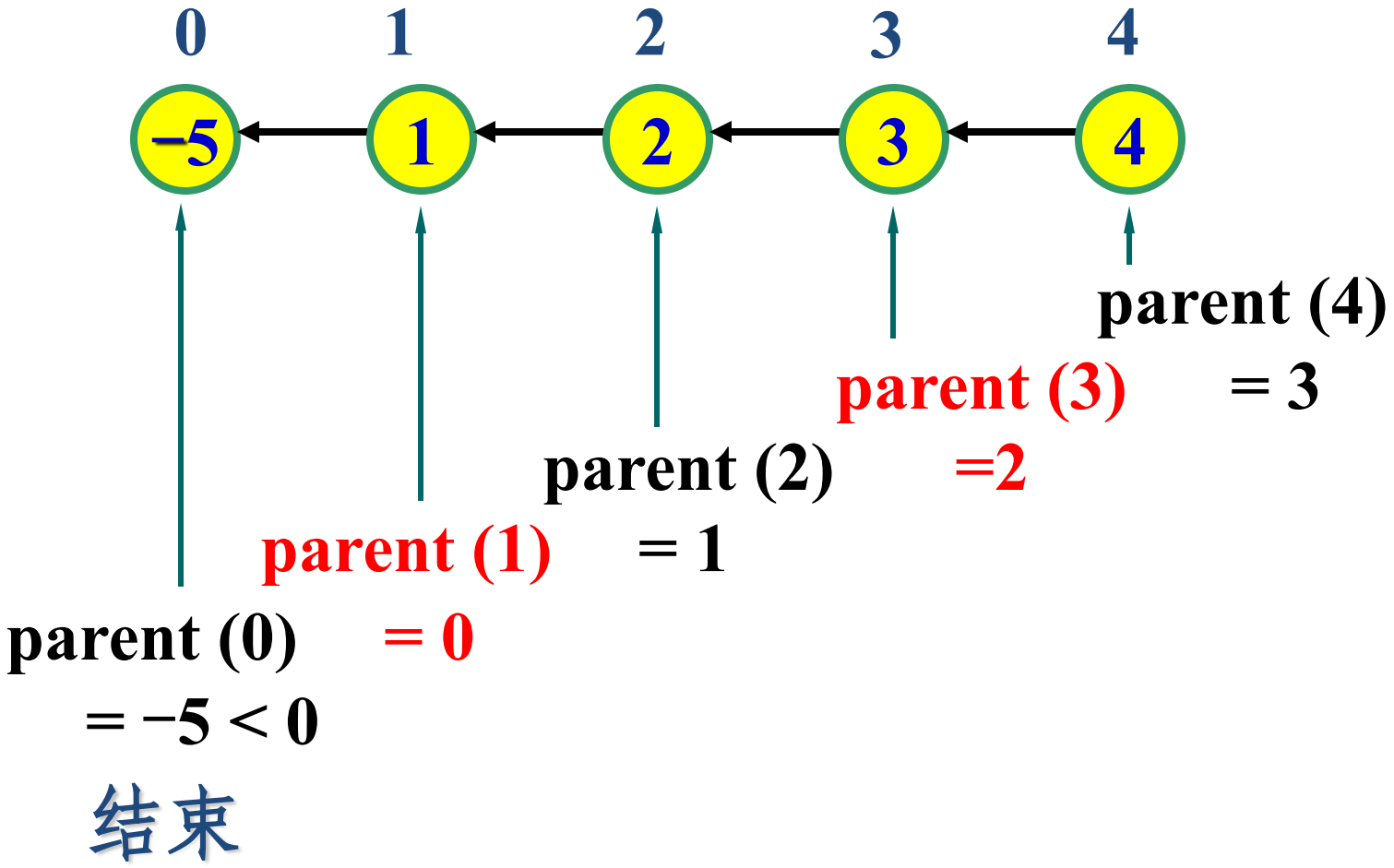
$S_1 \cup S_2$的可能的表示方法

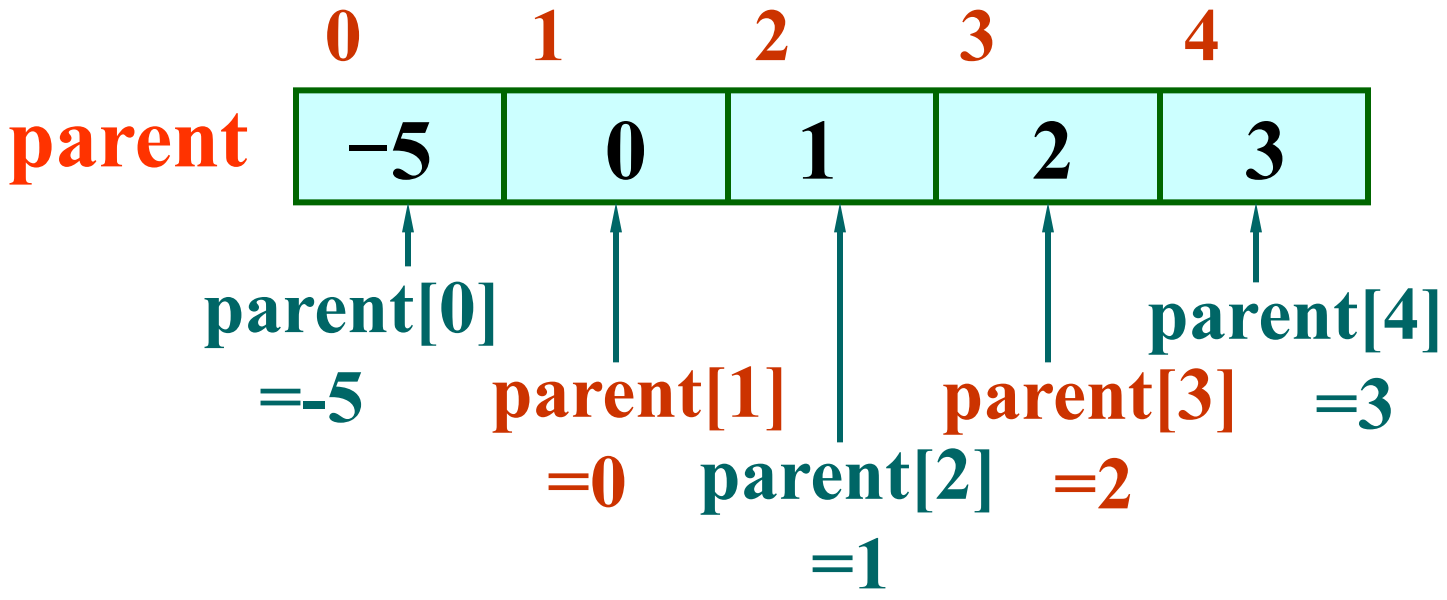
用双亲表示实现并查集的类定义

```
const int DefaultSize = 10;
class UFSets {        //集合中的各个子集合互不相交
public:
    UFSets (int sz = DefaultSize);        //构造函数
    ~UFSets() { delete []parent; }        //析构函数
    UFSets& operator = (UFSets& R);    //集合赋值
    void Union (int Root1, int Root2);    //子集合并
    int Find (int x);                     //查找x的根
    void WeightedUnion (int Root1, int Root2);
        //改进例程:加权的合并算法
private:
```

```
    int *parent;                    //集合元素数组(双亲表示)
    int size;                       //集合元素的数目
};

UFSets::UFSets (int sz) {
//构造函数：sz 是集合元素个数，双亲数组的范围
//为parent[0]～parent[size-1]。
    size = sz;                      //集合元素个数
    parent = new int[size];  //创建双亲数组
    for (int i = 0; i < size; i++) parent[i] = -1;
                                    //每个自成单元素集合
};
```

- 并查集操作的算法
  - 查找


0
1
2
3
4
−5
1
2
3
4
parent (4) = 3
parent (3) =2
parent (2) = 1
parent (1) = 0
parent (0) = −5 < 0
结束

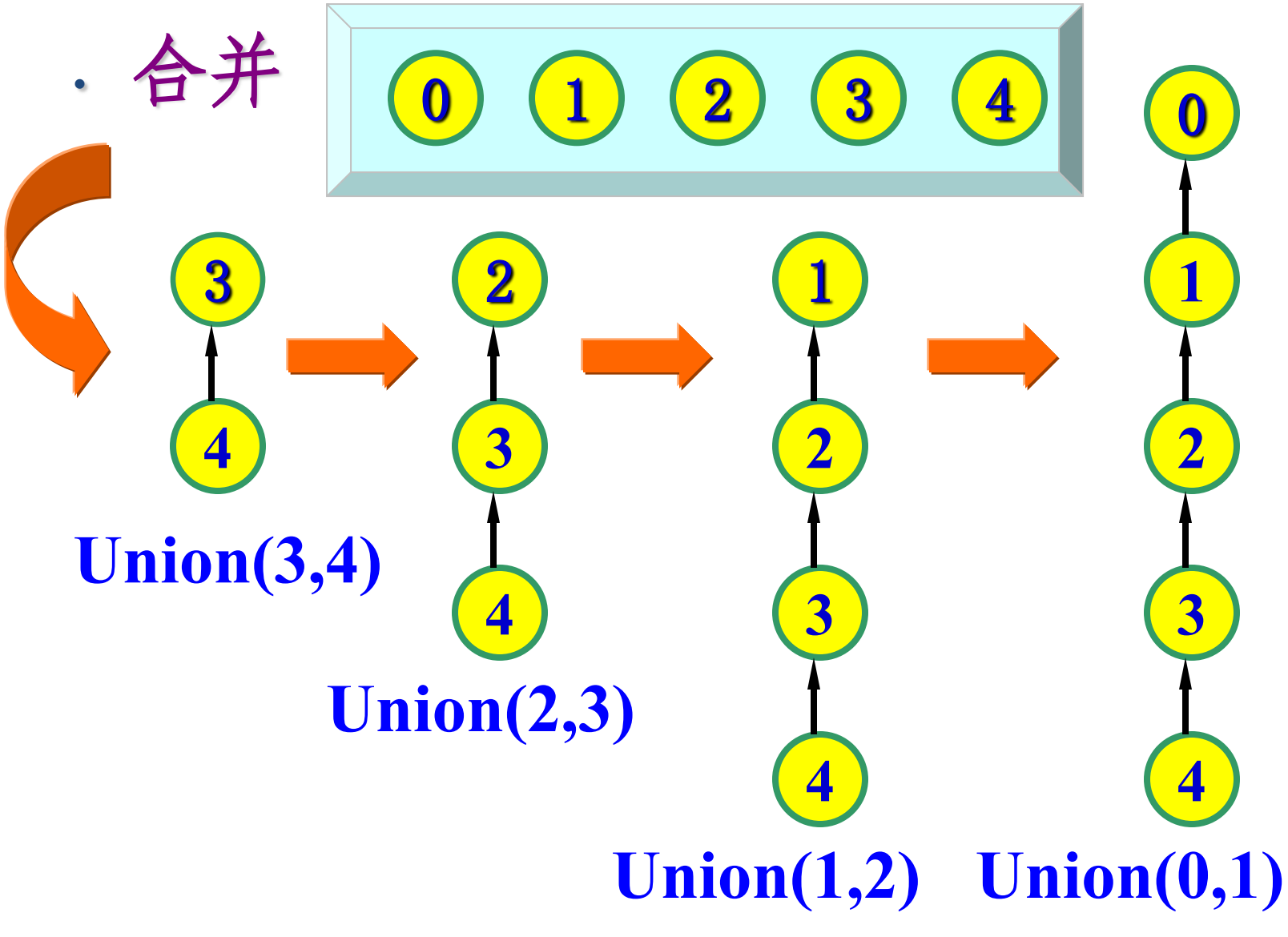



```
int UFSets::Find (int x) {
//函数搜索并返回包含元素x的树的根。
    if (parent[x] < 0) return x;   //根的parent[]值小于0
    else return (Find (parent[x]));
};
```

```
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

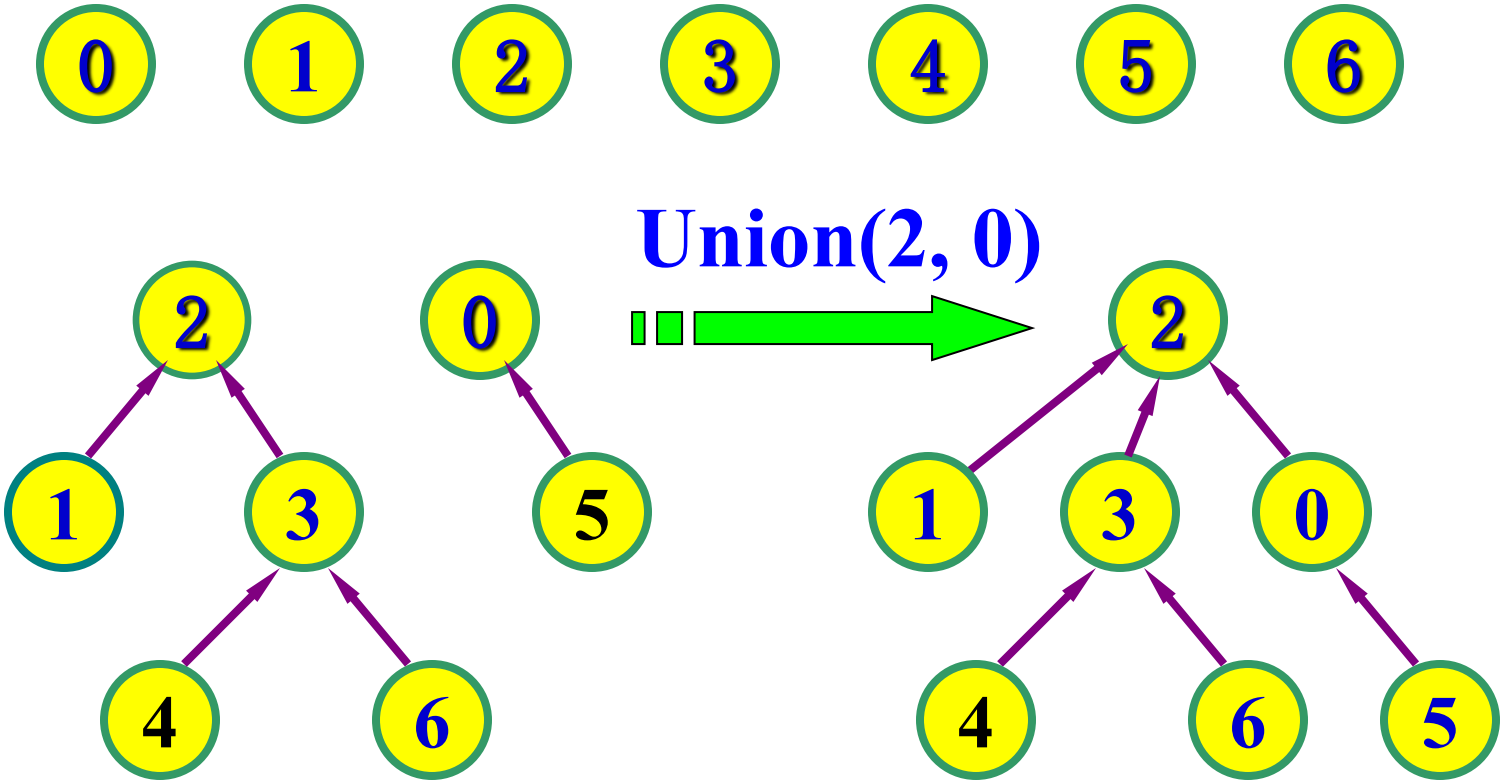


- 执行一次Union操作所需时间是 O(1)，$n-1$次Union操作所需时间是O(n)。
- 若再执行Find(0), Find(1), …, Find($n-1$), 若被搜索的元素为 i，完成 Find(i) 操作需要时间为O($i+1$)，完成 n 次搜索需要的总时间将达到

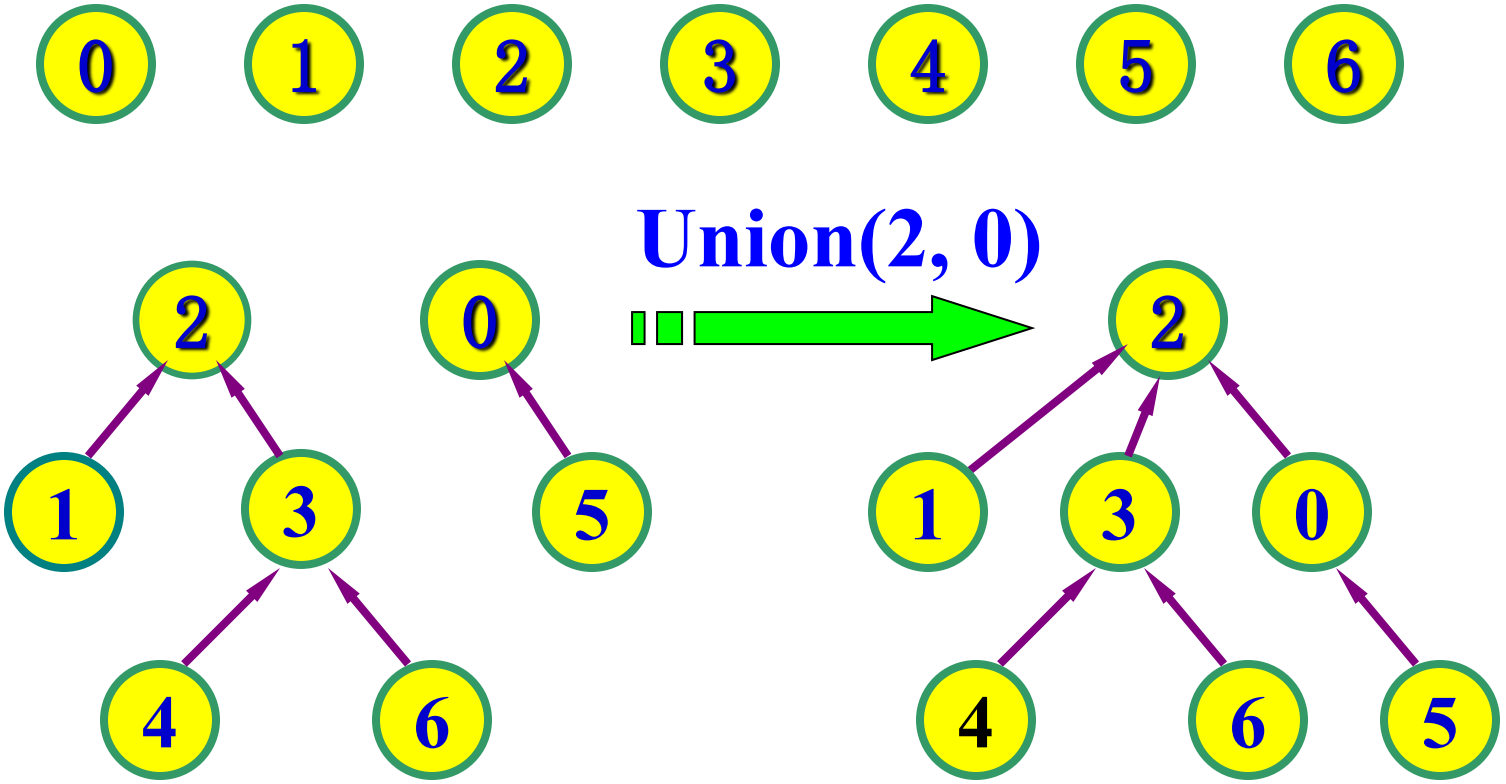
$$\mathrm{O}\left(\sum_{i=0}^{n-1}(i+1)\right)=\mathrm{O}(n^2)$$

- 改进的方法
  - 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

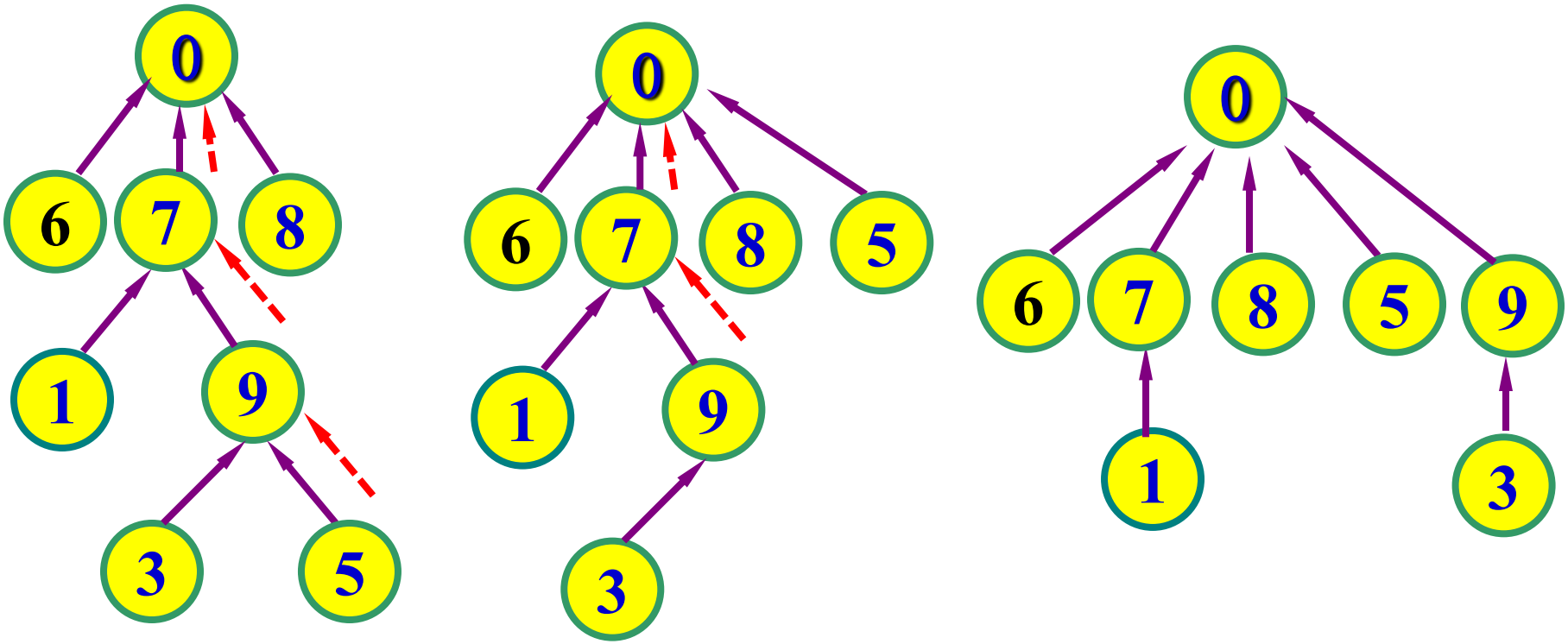
- 按树结点个数合并
  - 结点个数多的树的根结点作根



- 按树高度合并
  - 高度高的树的根结点作根



- 压缩元素的路径长度



字典（Dictionary）

- 字典是一些元素的集合，每个元素有一个称作关键码（key）的域，不同元素的关键码互不相同。
  - ✓ 文件（File）
  - ✓ 表格（Table）
- 在讨论字典抽象数据类型时，把字典定义为<名字-属性>对的集合。
- 根据问题的不同，可以为名字和属性赋予不同的含义。

- 例如，在图书馆检索目录中，名字是书名，属性是索书号及作者等信息；在计算机活动文件表中，名字是文件名，属性是文件地址、大小等信息。
- 一般来说，有关字典的操作有如下几种：

1. 确定一个指定的名字是否在字典中；
2. 搜索出该名字的属性；
3. 修改该名字的属性；
4. 插入一个新的名字及其属性；
5. 删除一个名字及其属性。

字典的抽象数据类型

```
const int DefaultSize = 26;
template <class Name, class Attribute>
class Dictionary {
//对象: 一组<名字-属性>对, 其中, 名字是唯一的
public:
    Dictionary (int size = DefaultSize);      //构造函数
    bool Member (Name name);
        //判name是否在字典中
    Attribute *Search (Name name);
        //在字典中搜索关键码与name匹配的表项
```

```
    void Insert (Name name, Attribute attr);
      //若name在字典中, 则修改相应<name, attr>对
      //的attr项; 否则插入<name, attr>到字典中
    void Remove (Name name);
      //若name在字典中, 则在字典中删除相应的
      //<name, attr>对
};
```

- 用文件记录（**record**）或表格的表项（**entry**）来表示单个元素时，用:

  （关键码**key**，记录或表项位置指针**adr**）

  构成搜索某一指定记录或表项的索引项。

字典的线性表描述

- 字典可以保存在线性序列 $(e_1,e_2,\ldots)$ 中，其中 e_i 是字典中的元素，其关键码从左到右依次增大。为了适应这种描述方式，可以定义有序顺序表和有序链表。
- 用有序链表来表示字典时，链表中的每个结点表示字典中的一个元素，各个结点按照结点中保存的数据值非递减链接，即 $e_1 \le e_2 \le \ldots$。因此，在一个有序链表中寻找一个指定元素时，一般不用搜索整个链表。

有序顺序表的类定义

```
#include <iostream.h>
#include "SeqList.h"
const int defaultSize = 50;
template <class K, class E>
class SortedList : public SeqList {
public:
    int Search (K k1) const;                    //搜索
    void Insert (const K k1, E& e1);            //插入
    bool Remove (const K k1, E& e1);            //删除
};
```

基于有序顺序表的顺序搜索算法

```
template <class K, class E>
int SortedList<K, E>::Search (K k1) const {
//顺序搜索关键码为k1的数据对象
    int n = last+1;
    for (int i = 1;  i <= n;  i++)
        if (data[i-1] == k1) return i;          //成功
        else if (data[i-1] > k1) break;
    return 0;       //顺序搜索失败,返回失败信息
};
```

- 算法中的“==”和“>”都是重载函数，在定义K时定义它们的实现。

有序顺序表顺序搜索的时间代价

- 衡量一个搜索算法的时间效率的标准是：在搜索过程中关键码平均比较次数，也称为平均搜索长度**ASL (Average Search Length)**，通常它是字典中元素总数 n 的函数。
- 设搜索第 i 个元素的概率为 p_i，搜索到第 i 个元素所需比较次数为 c_i，则搜索成功的平均搜索长度：

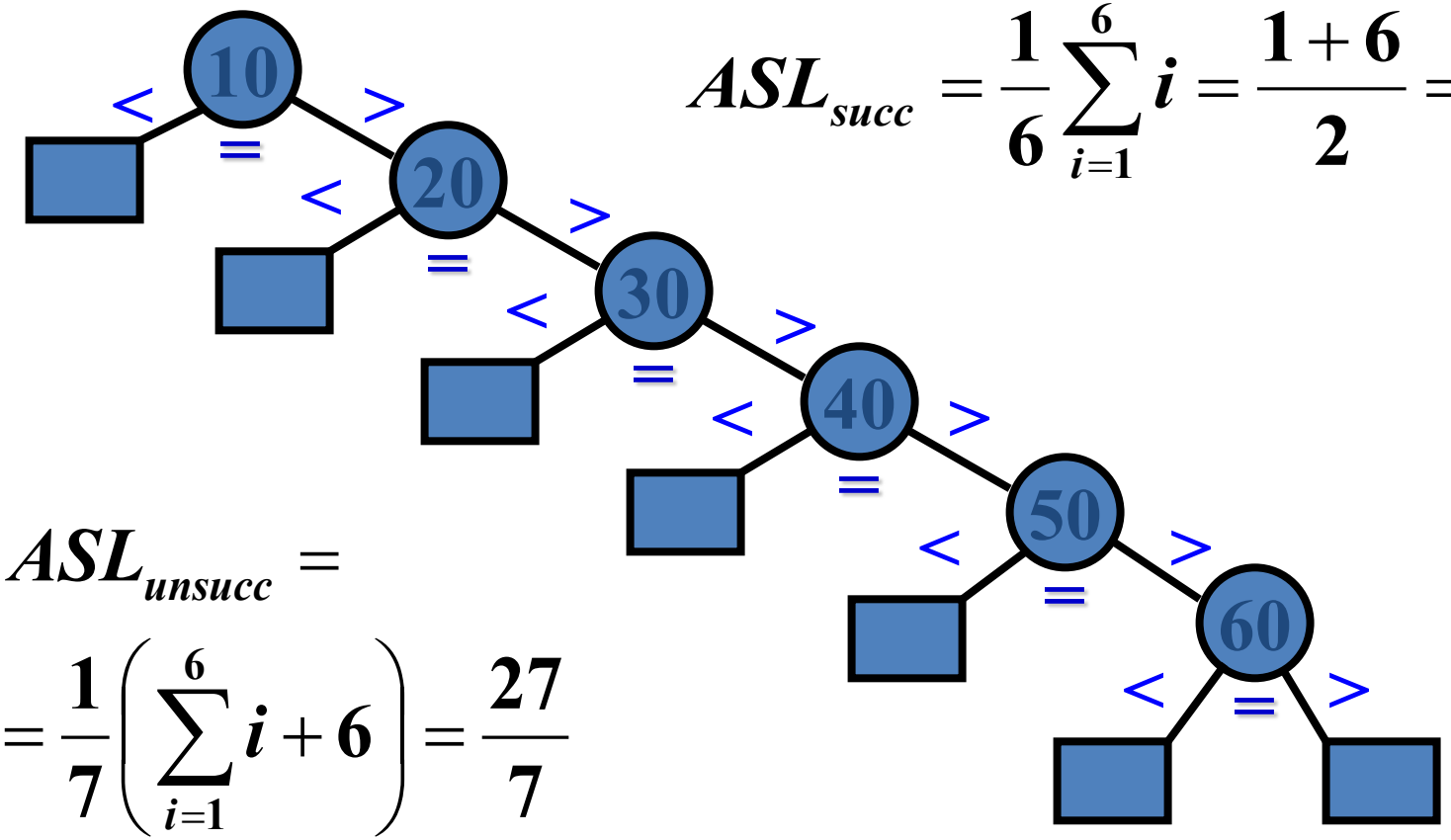
$$ASL_{succ} = \sum_{i=1}^{n} p_i \cdot c_i. \qquad (\sum_{i=1}^{n} p_i = 1)$$

- 在顺序搜索情形，搜索第 i $(1\le i\le n)$ 个元素需要比较 i 次，假定按等概率搜索各元素：

$$ASL_{succ}=\sum_{i=1}^{n}p_i\times c_i=\frac{1}{n}\sum_{i=1}^{n}i=\frac{1}{n}\frac{(n+1)n}{2}=\frac{n+1}{2}$$

- 这与一般顺序表情形相同。但搜索不成功时不需一直比较到表尾，只要发现下一个元素的值比给定值大，就可断定搜索不成功。
- 设一个有 n 个表项的表，查找失败的位置有 $n+1$ 个，可以用判定树加以描述。搜索成功时停在内结点，搜索失败时停在外结点。

- 例如，有序顺序表 (10, 20, 30, 40, 50, 60) 的顺序搜索的分析（使用判定树）



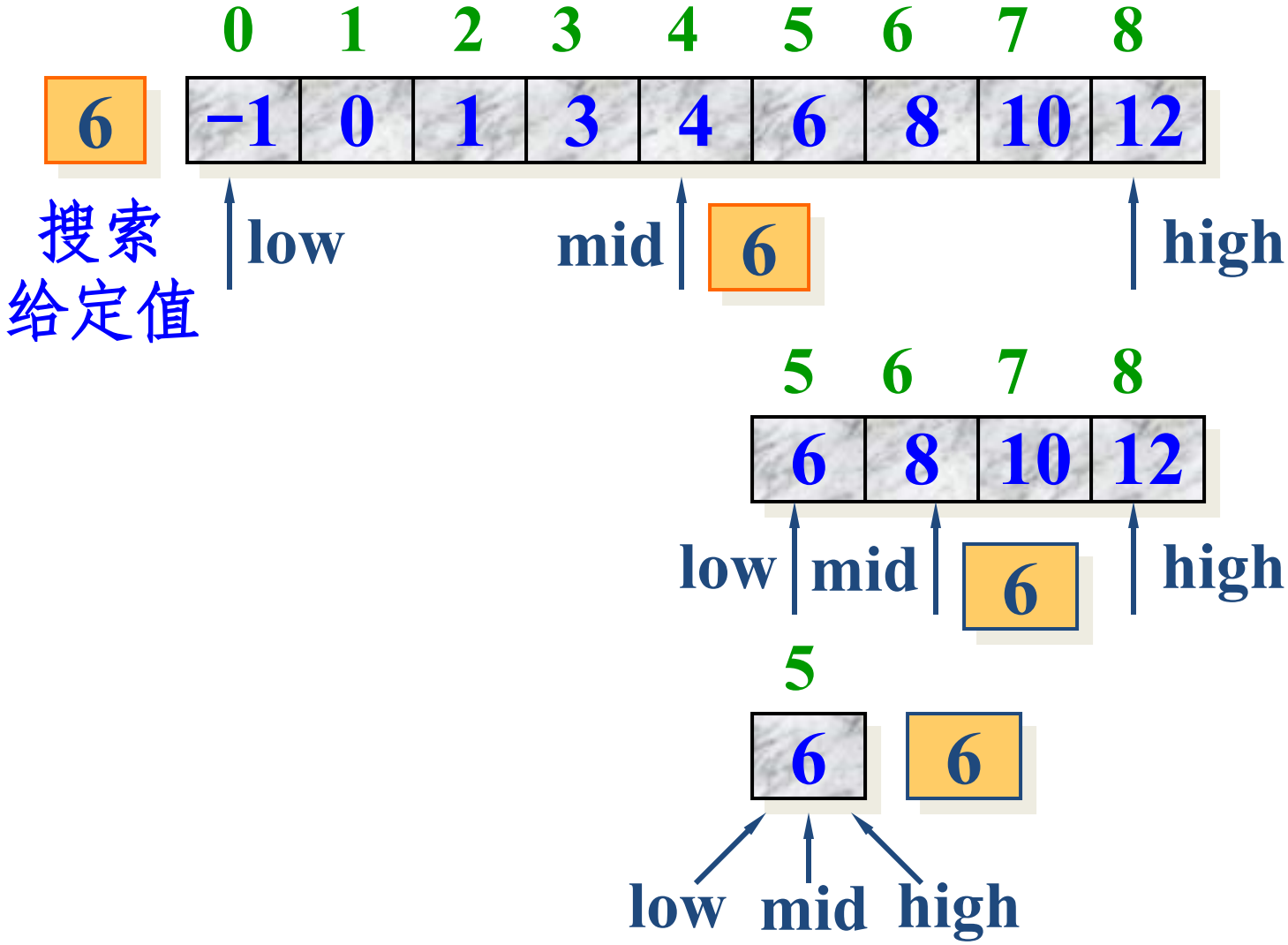
- 判定树是一种扩充二叉树。内结点代表顺序表中已有的元素，外结点代表失败结点，它表示在两个相邻已有元素值之间的值。
- 假定表中所有失败位置的搜索概率相同，则搜索不成功的平均搜索长度：

$$ASL_{unsucc} = \frac{1}{n+1}\left(\sum_{i=1}^{n} i + n\right)$$

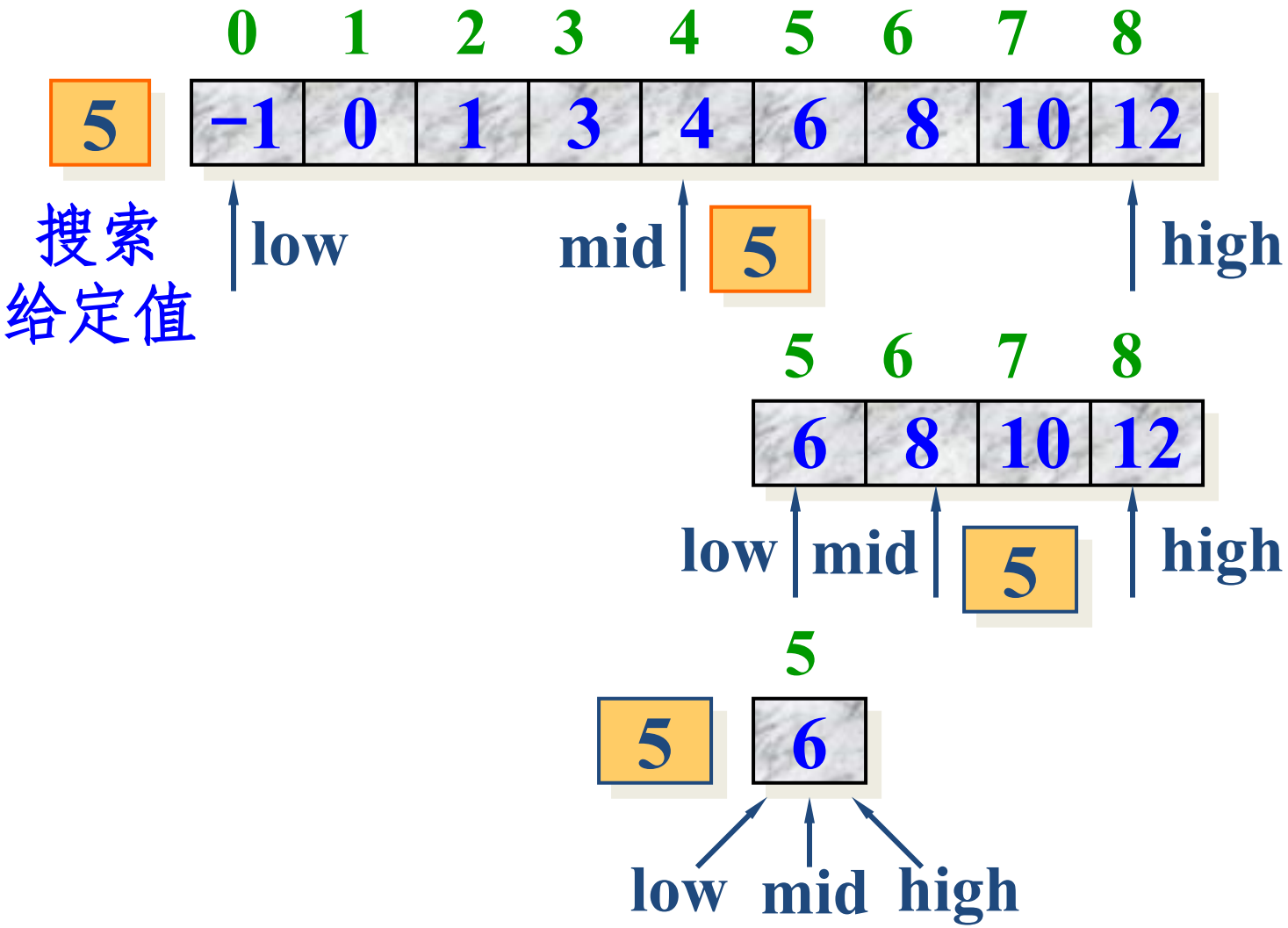
- 时间代价为$O(n)$。为了加速搜索，在有序顺序表的情形，可以采用折半搜索，它也称二分搜索，时间代价可减小到$O(\log_2 n)$。

基于有序顺序表的折半搜索

- 设 *n* 个元素存放在一个有序顺序表中。
- 折半搜索时，先求位于搜索区间正中的元素的下标**mid**，用其关键码与给定值 **x** 比较：
  - **data[mid].key == x**，搜索成功；
  - **data[mid].key > x**，把搜索区间缩小到表的前半部分，继续折半搜索；
  - **data[mid].key < x**，把搜索区间缩小到表的后半部分，继续折半搜索。
- 如果搜索区间已缩小到一个对象，仍未找到想要搜索的对象，则搜索失败。


0 1 2 3 4 5 6 7 8
6
-1 0 1 3 4 6 8 10 12
搜索
给定值
low
mid
6
high
5 6 7 8
6 8 10 12
low
mid
6
high
5
6
6
low mid high

搜索成功的例子


0 1 2 3 4 5 6 7 8
5
-1 0 1 3 4 6 8 10 12
搜索
给定值
low
mid
5
high
5 6 7 8
6 8 10 12
low
mid
5
high
5
5
6
low
mid
high

搜索失败的例子

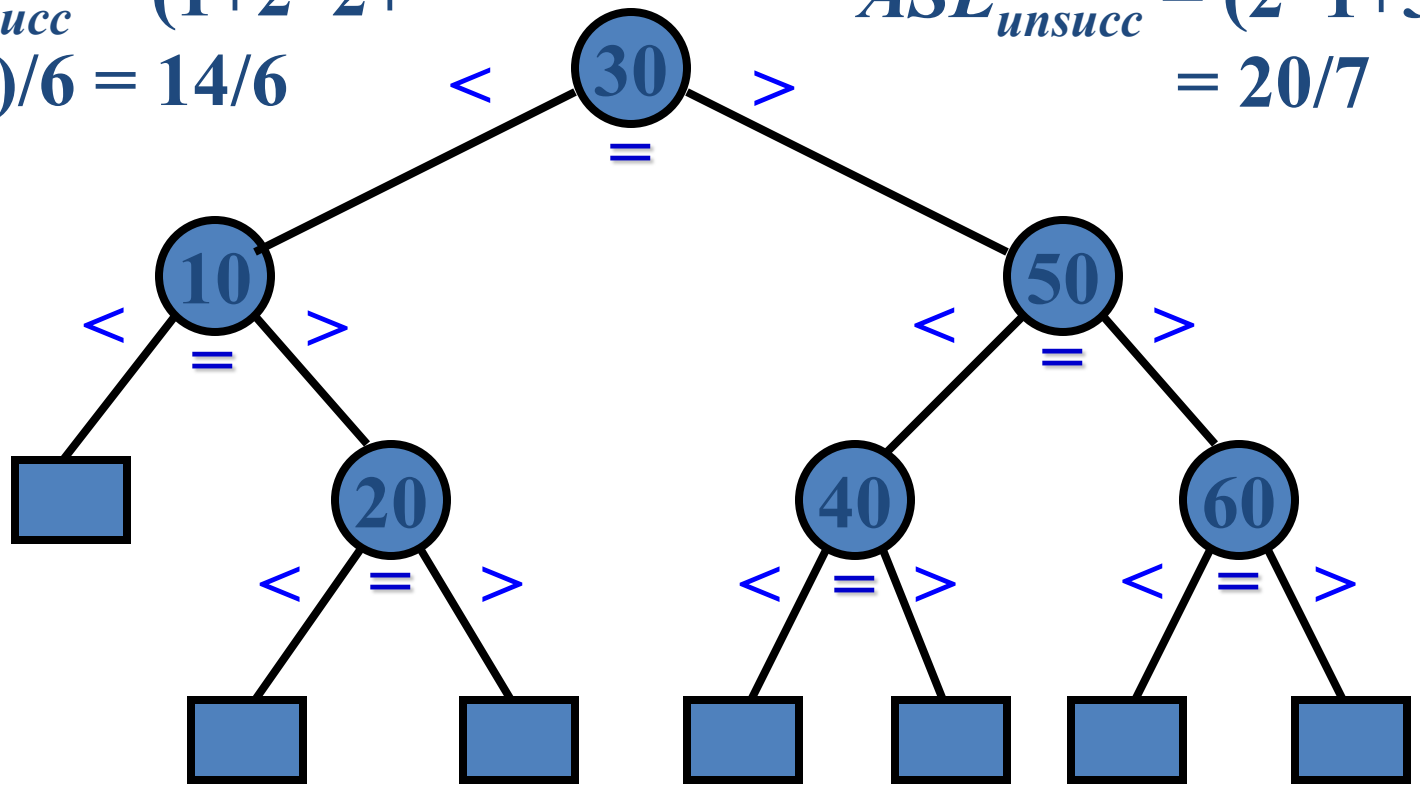
```
template<class K, class E>
int SortedList<K, E>::BinarySearch
    (K k1, const int low, const int high ) const {
//折半搜索的递归算法，用到E的重载操作“<”和“>”
    int mid = 0;                          //元素序号从0开始
    if (low <= high) {
        mid = (low + high) / 2;
        if (data[mid] < k1)
            mid = BinarySearch (k1, mid +1, high);
        else if (data[mid] > k1)
            mid = BinarySearch (k1, low, mid −1);
        else return mid;
    }
    return 0;
};
```

```
template<class K, class E>
int SortedList <K, E>::BinarySearch (K k1) const {
//折半搜索的迭代算法，用到K的重载操作“<”和“>”
    int high = n-1,  low = 0,  mid;    //元素序号从0开始
    while (low <= high) {
        mid = (low + high) / 2;
        if (data[mid] < k1) low = mid+1;
                                        //右缩搜索区间
        else if (data[mid] > k1) high = mid-1;
                                        //左缩搜索区间
        else return mid;                //搜索成功
    }
   return 0;                            //搜索失败
}
```

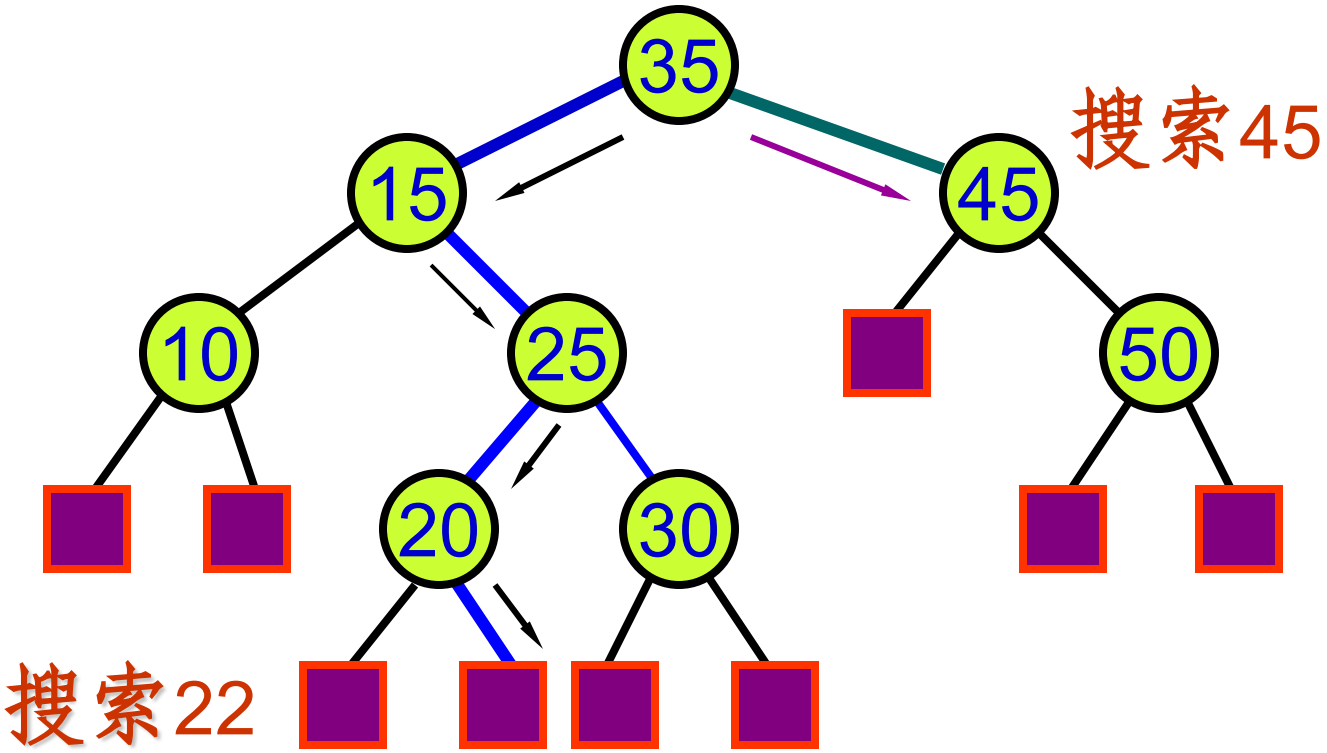
- 分析有序顺序表 (10, 20, 30, 40, 50, 60) 的折半搜索算法性能的判定树:

ASL_{succ} = (1+2*2+3*3)/6 = 14/6

ASL_{unsucc} = (2*1+3*6)/7 = 20/7



- 判定树也是扩充二叉树，搜索成功时检测指针停留在树中某个内结点上。搜索不成功时检测指针停留在某个外结点（失败结点）上。



折半搜索算法的性能分析

- 若设 $n = 2^h - 1$，则描述折半搜索的判定树是高度为 h 的满二叉树 (加上失败结点高度为 $h+1$)。

$$2^h = n+1,\ h = \log_2(n+1)$$

- 第1层结点有1个, 搜索第1层结点要比较1次; 第2层结点有2个, 搜索第2层结点要比较2次; …, 第 i $(1 \leqslant i \leqslant h)$ 层结点有 2^{i-1} 个, 搜索第 i 层结点要比较 i 次，….
- 假定每个结点的搜索概率相等，即 $p_i = 1/n$，则搜索成功的平均搜索长度为

$$ASL_{succ} = \frac{1}{n}(1\times 2^0 + 2\times 2^1 + 3\times 2^2 + \cdots + h\times 2^{h\text{-}1})$$

可以用归纳法证明

$$1\times 2^0 + 2\times 2^1 + 3\times 2^2 + \cdots + (h-1)\times 2^{h-2} + h\times 2^{h-1}$$

$$= (h-1)\times 2^h + 1$$

这样，由 $2^h = n+1$, $h = \log_2(n+1)$

$$ASL_{succ} = \frac{1}{n}((h-1)\times 2^h + 1) = \frac{1}{n}((n+1)\log_2(n+1) - n)$$

$$= \frac{n+1}{n}\log_2(n+1) - 1 \approx \log_2(n+1) - 1$$

有序链表的类定义

```
#include <assert.h>
template <class K, class E>
struct ChainNode {                    //链表结点类定义
    K key;
    E attr;
    ChainNode<K, E> *link;
    ChainNode() : link (NULL) { };      //构造函数
    ChainNode (K k1, E& e1,                    //构造函数
          ChainNode<K,E> *next = NULL)
           : key (k1), attr (e1), link (next) { };
};
```

```
template <class K, class E>
class SortedChain {                          //有序链表类定义
public:
    SortedChain () {                         //构造函数
        first = new ChainNode<K, E>;
        assert (first != NULL);
    };
    ～SortedChain ();                        //析构函数
    ChainNode<K, E> *Search (K k1);                    //搜索
    void Insert (const K k1, E& e1);                   //插入
    bool Remove (const K k1, E& e1);                   //删除
```

```
    ChainNode<K, E> *Begin () { return first->link; }
                                        //定位第一个
    ChainNode<K, E> *Next (ChainNode<K, E>
            *current) const {       //定位下一个
        if (current != NULL) return current->link;
        else return NULL;
    }
private:
    ChainNode<K, E> *first;     //链表的头指针
};
```

搜索、插入与删除算法

```
template <class K, class E>
ChainNode<K, E> *SortedChain<T>::
Search (K k1) const {
    ChainNode<K, E> *p = first->link;
    while (p != NULL && p->key< k1)
        p = p->link;                    //重载：元素判小于
    if ( p != NULL && p->key== k1) return p;
                                        //重载：元素判等于
    else return NULL;
};
```

```
template <class K, class E>
void SortedChain<K, E>::
Insert (const K k1,  E& e1) {
    ChainNode<K, E> *p = first->link, *pre = first;
    ChainNode<K, E> *newNode;
    while (p != NULL && p->key<= k1)
                        //重载：元素判小于等于
        { pre = p;  p = p->link; }      //寻找插入位置
    if (p != NULL && p->key== k1) p->attr= e1;
    else {
        newNode = new ChainNode<K, E>(k1,e1);
        if (newNode == NULL) {
```

```
            cerr <<“存储分配失败！” << endl;
            exit (1);
        }
        newNode->link = p;   pre->link = newNode;
    }
};
```

```
template <class K, class E>
bool SortedChain<K, E>::Remove (const K k1,  E& e1) {
    ChainNode<K, E> *p = first->link, *pre = first;
    while (p != NULL && p->key< k1)
       { pre = p;  p = p->link; }          //寻找删除位置
    if (p != NULL && p->key== k1) {
                  //重载：元素关键码判等于
        pre->link = p->link;
        e1 = p->attr;
        delete p;
        return true;
    }
    else return false;                     //未找到删除结点
};
```

散列表（Hash Table）

- 理想的搜索方法是可以不经过比较，一次直接从字典中得到要搜索的元素。

- 如果在元素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()，使得每个关键码与唯一的存储位置相对应：

  Address = Hash(key)

- 在插入时依此函数计算存储位置并按此位置存放。在搜索时对元素的关键码进行同样的计算，把求得的函数值当做元素存储位置，然后按此位置搜索。这就是散列方法。

- 在散列方法中所用转换函数叫做**散列函数**(又叫**哈希函数**)。按此方法构造出来的表叫做**散列表**(又叫**哈希表**)。

- 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

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

  则有 $hash(12361) = hash(07251) = hash(03309) = hash(30976) = 13444$。

- 就是说，对不同的关键码，通过散列函数的计算，得到了同一散列地址。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；

  - 设计解决冲突的方案。

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。

❶ 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。

❷ 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。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	①位， $\lambda_1 = 57.60$
9	4	1	2	6	9	②位， $\lambda_2 = 57.60$
9	4	0	5	2	7	③位， $\lambda_3 = 17.60$
9	4	1	6	3	0	④位， $\lambda_4 = 5.60$
9	4	1	8	0	5	⑤位， $\lambda_5 = 5.60$
9	4	1	5	5	8	⑥位， $\lambda_6 = 5.60$
9	4	2	0	4	7	
9	4	0	0	0	1	
①	②	③	④	⑤	⑥	

- 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。

❸ 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，用以下函数把关键码转换成散列地址:

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

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

  $hash(962148) = 962148 \% 23 = 12$。

- 可按计算出的地址存放记录。注意，使用散列函数计算出的地址范围是 0 到 22，而 23、24 这几个地址实际上不能用散列函数计算出来，只能在处理冲突时达到这些地址。

❹平方取中法

它首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8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。

标识符	内码	内码平方	散列地址
A	01	01	001
*A*1	0134	20420	042
*A*9	0144	23420	342
B	02	04	004
DMAX	04150130	21526443617100	443
*DMAX*1	0415013034	526447352151420	352
AMAX	01150130	135423617100	236
*AMAX*1	0115013034	345424652151420	652

标识符的八进制内码表示及其平方值和散列地址

❺ 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。

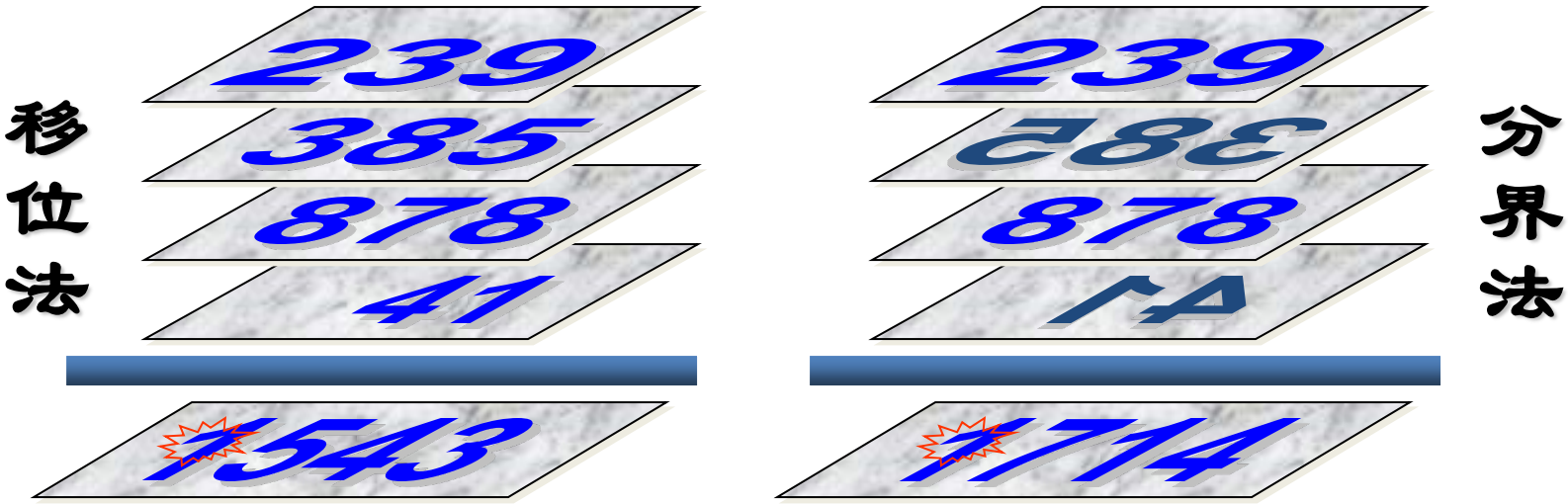
把这些部分的数据叠加起来，就可以得到具有该关键码的记录的散列地址。

- 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。

- 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

- 叠加，然后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。

- 假设地址空间为**HT[400]**，利用以上函数计算，取其中**3**位，取值范围在**0～999**，可能超出地址空间范围，为此必须将**0～999**压缩到**0～399**。可将计算出的地址乘以一个压缩因子**0.4**，把计算出的地址压缩到允许范围。

(1)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。

线性探查法（Linear Probing）

例1：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，使用散列函数计算关键字在表中的位置：

$$H_0 = hash(key)$$

- 一旦发生冲突，在表中顺次向后寻找下一个位置 H_i：

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

即用以下的线性探查序列在表中寻找下一个位置：

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

亦可写成如下的通项公式：

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

例1中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:

$$ASLsucc = \frac{1}{8}\sum_{i=1}^{8} Ci = \frac{1}{8}(1+1+2+3+1+6+3+1) = \frac{18}{8}.$$

- 搜索不成功的平均搜索长度为:

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

- 例2：已知散列表A[0..11]，选取除留余数法设计散列函数H(K)= K mod 11，关键字集合为{10，20，15，17，21，6，8，25，40，35，27}，采用线性探查法处理冲突，请将关键字填入下面的哈希表中，计算成功和失败时的平均查找长度ASL。

0	1	2	3	4	5	6	7	8	9	10	11
		35	25	15	27	17	40	8	20	21	
		1	1	1	1	2	6	1	1	2	
失败	2	1	12	11	10	9	8	7	6	5	4

例2中使用线性探查法对示例进行搜索时:

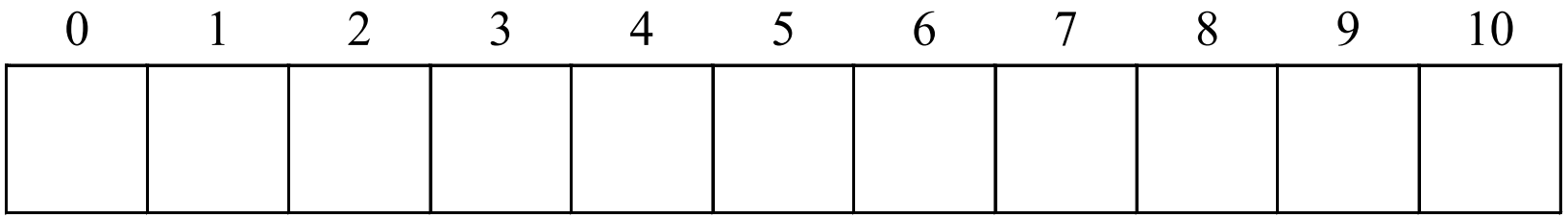
- 搜索成功的平均搜索长度为:

$$ASLsucc = \frac{1}{11}\sum_{i=1}^{11} Ci = \frac{1}{11}(8 * 1 + 2 + 2 + 6) = \frac{18}{11}.$$

- 搜索不成功的平均搜索长度为:

$$ASLunsucc = \frac{2 + 1 + 12 + 11 + \ldots + 4}{11} = \frac{75}{11}.$$

- 例3：已知关键字集合为{ 47，7，29，11，16，92，22，8 }。假设哈希函数 H(key) = key mod 11 (表长=11)，采用线性探查法处理冲突，请将关键字填入下面的哈希表中，计算成功和失败时的ASL。



(2)处理冲突的开散列方法（链地址法）

- 开散列方法（链地址法）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位置从 $0 \sim m-1$, 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*’)**。

用散列函数*Hash* (*x*) = *ord* (*x*) − *ord* (‘*A*’)计算可得:

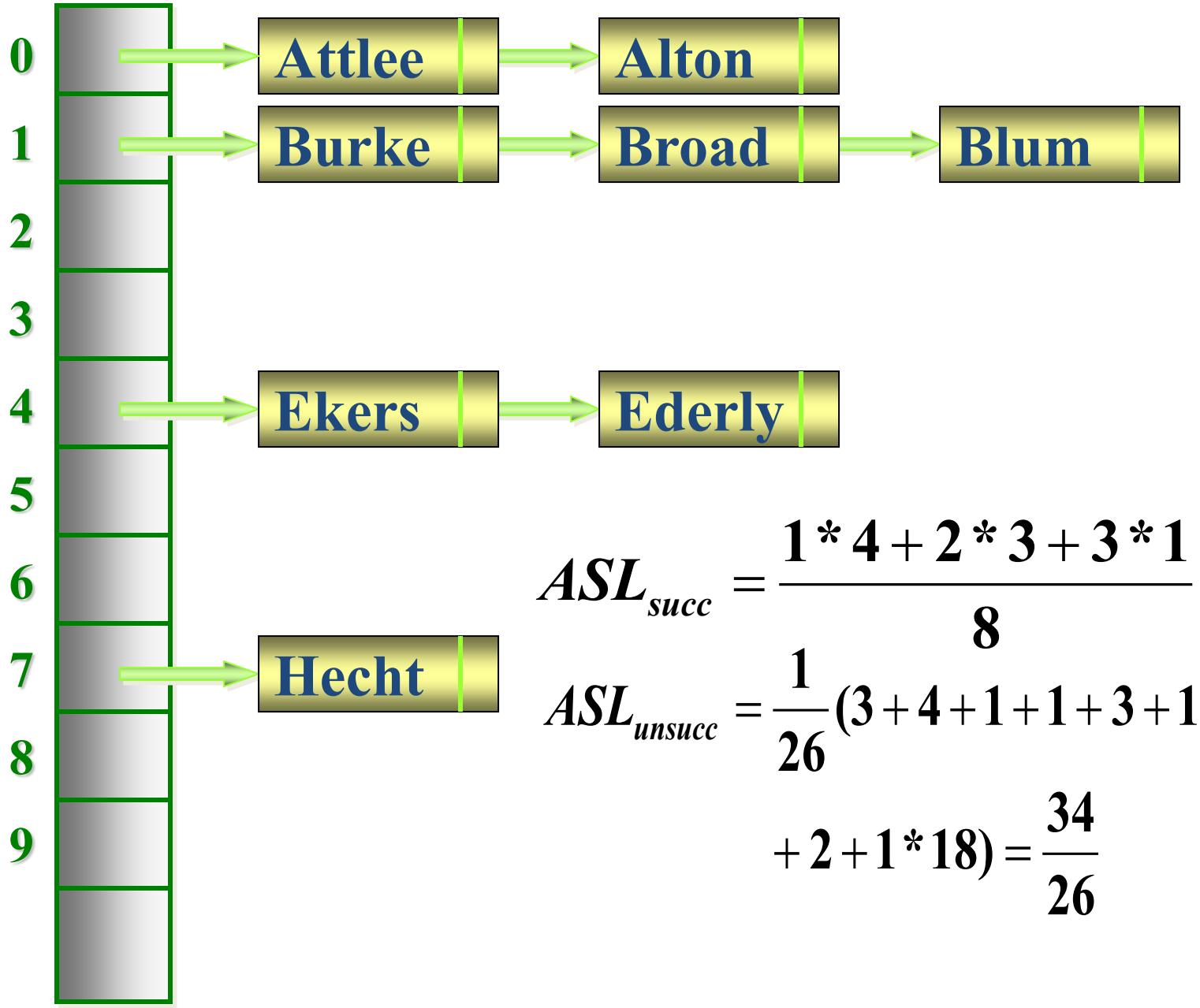
Hash (Burke) = 1　　*Hash* (Ekers) = 4

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。