第10章　散列

----有序列表：基于高效访问的处理技术

计算机学院

主要内容

- **字典（有序表）的定义**
- **散列表**

字典ADT

抽象数据类型*Dictionary*{

实例

具有不同关键字的元素集合

操作

Create()：创建一个空字典

Search(*k*, *x*)：搜索关键字为*k*的元素，结果放入*x*；
如果没找到，则返回*false*，否则返回true

Insert(*x*)：向字典中插入元素*x*

Delete(*k*, *x*)：删除关键字为*k*的元素，并将其放入*x*

}

字典操作

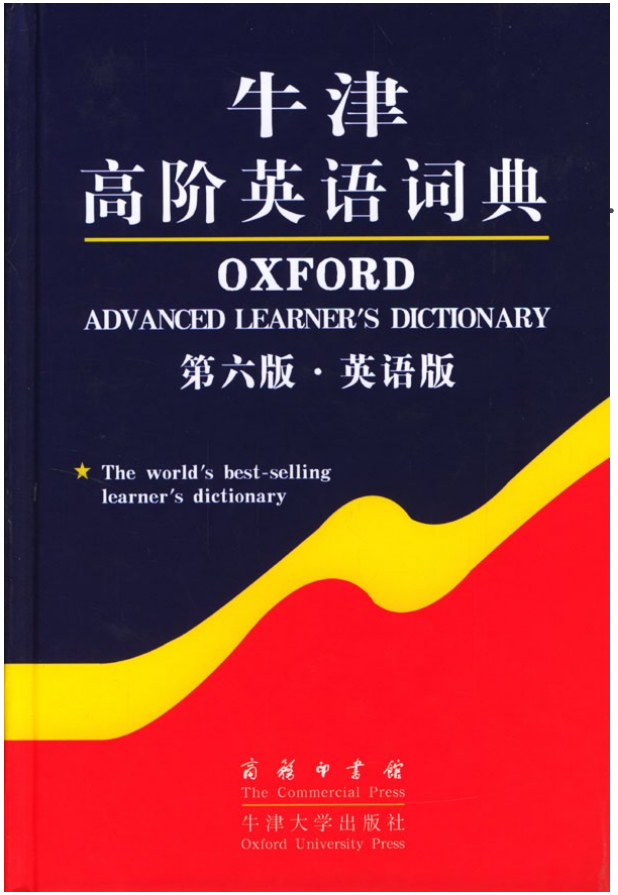
- 随机访问，random access
- 顺序访问，sequential access
  - Begin，Next
- 重要的操作方式是“按关键字访问”
- 需注意的一个问题：重复关键字
  - 如何消除歧义

字典例

- 一个班注册学习数据结构课程的学生
  - 新学生注册→在字典中插入相关的元素(记录)
  - 放弃这门课程→删除对应记录
  - 查询字典→获取特定学生相关的记录或修改记录
  - 学生的姓名域可作为关键字

字典例（续）

- 在编译器中定义用户标识符的符号表（symbol table）——重复元素字典
  - 定义标识符→建立一个记录并插入到符号表中
  - 同样的标识符名可以定义多次(在不同的程序块中)→相同关键字的记录
  - 搜索结果——最新插入的元素
  - 删除——程序块的结尾（标识符作用域结束）


牛津
高阶英语词典
OXFORD
ADVANCED LEARNER'S DICTIONARY
第六版·英语版
★ The world's best-selling learner's dictionary
商务印书馆
The Commercial Press
牛津大学出版社
Oxford University Press


com·mem·or·ation /kə,memə'reɪʃn
com·mem·ora·tive /kə'memərətɪv;
com·mence /kə'mens || kə'mɛns/

字典的线性表描述

- $(e_1, e_2, \ldots,)$
  - e_i：字典元素，关键字升序排列
- 公式化描述
  - 搜索操作：二分搜索，O(logn)
  - 插入、删除操作：需数据移动，O(n)
- 链表描述
  - 搜索、插入、删除均为O(n)

能否提高？如何提高？

SortedChain类

```
template<class E, class K>
class SortedChain {
  public:
    SortedChain() {first = 0;}
    ~SortedChain();
    bool IsEmpty() const {return first == 0;}
    int Length() const;
    bool Search(const K& k, E& e) const;
    SortedChain<E,K>& Delete(const K& k, E& e);
    SortedChain<E,K>& Insert(const E& e);
    SortedChain<E,K>& DistinctInsert(const E& e);
    void Output(ostream& out) const;
  private:
    SortedChainNode<E,K> *first;
};
```

搜索操作

```
template<class E, class K>
bool SortedChain<E,K>::Search(const K& k, E& e) const
{
   SortedChainNode<E,K> *p = first;
   while (p && p->data < k)
      p = p->link;

  // 判断是否匹配
  if (p && p->data == k) // 如果链表尚不为空且数据匹配
    {e = p->data; return true;}
  return false; // 链表已经为空，或当前数据大于k
}
```

等价于for(; p&&p->data<k ; p=p->k) ;

删除操作

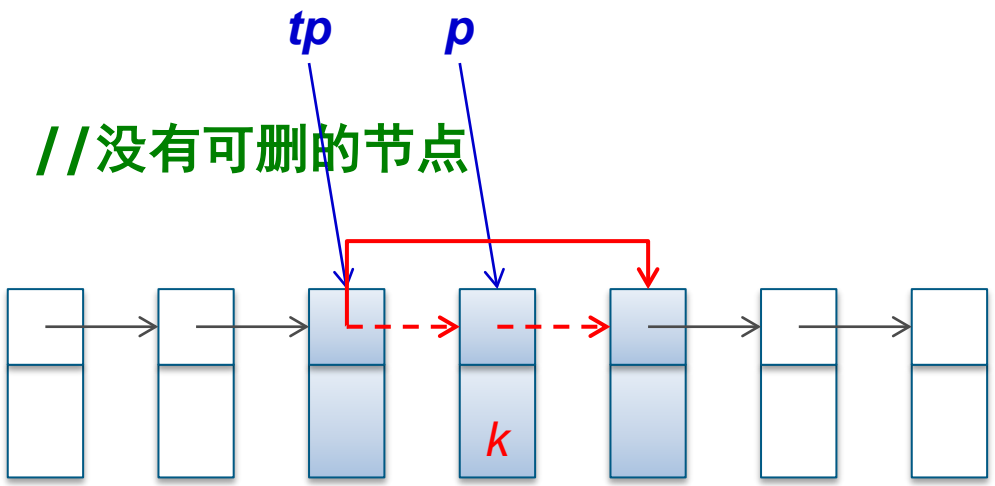
```
template<class E, class K>
SortedChain<E,K>& SortedChain<E,K>
                ::Delete(const K& k, E& e)
{
   SortedChainNode<E,K> *p = first, *tp = 0;

  // search for match with k
  while (p && p->data < k) {
      tp = p;
      p = p->link;
    }
```

用于移动

用于记录p的前一位置

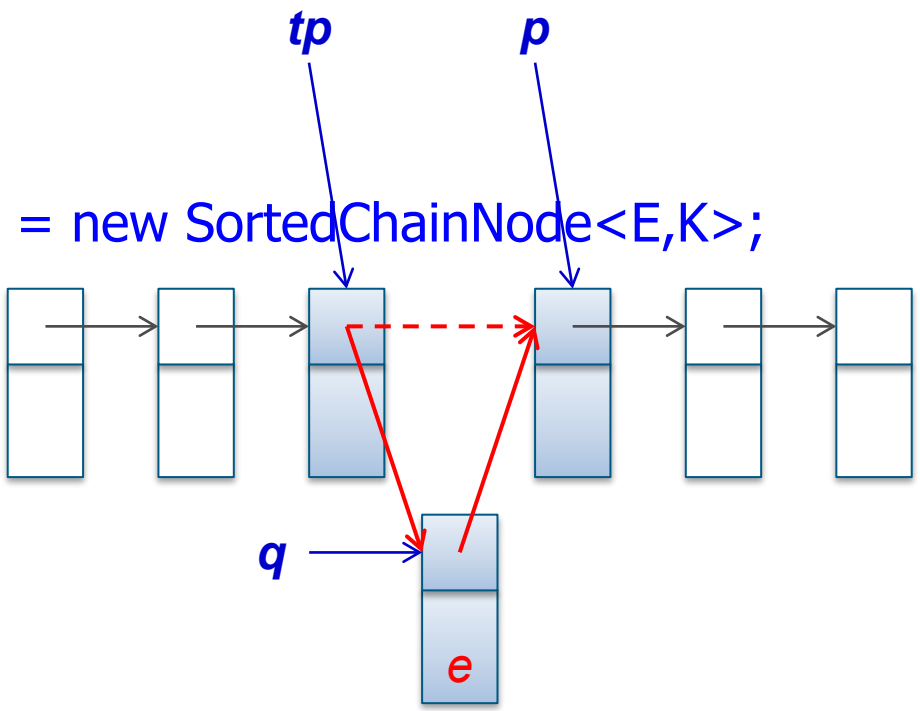
删除操作（续）

```
    if (p && p->data == k) //找到了要删的节点
    {
        e = p->data;
        if (tp)  tp->link = p->link; //p是普通节点
        else first = p->link;  //p是首节点
        delete p;
        return *this;
    }
    throw BadInput();  //没有可删的节点
    return *this;
}
```



插入操作

```
template<class E, class K>
SortedChain<E,K>& SortedChain<E,K>::Insert(const E& e)
{
  SortedChainNode<E,K> *p = first,  *tp = 0;
  while (p && p->data < e) {
    tp = p;
    p = p->link;
  }
  SortedChainNode<E,K> *q = new SortedChainNode<E,K>;
  q->data = e;
  q->link = p;
  if (tp) tp->link = q;
  else first = q;
  return *this;
}
```



不允许重复关键字的插入

```
template<class E, class K>
SortedChain<E,K>& SortedChain<E,K>
                ::DistinctInsert(const E& e)
{// Insert e only if no element with same key

  SortedChainNode<E,K> *p = first, *tp = 0;

  while (p && p->data < e)
  {
    tp = p;
    p = p->link;
  }
```

不允许重复关键字的插入(续)

```
    // check if duplicate
    if (p && p->data == e) throw BadInput();

    // not duplicate, set up node for e
    SortedChainNode<E,K> *q = new SortedChainNode<E,K>;
    q->data = e;

    // insert node just after tp
    q->link = p;
    if (tp) tp->link = q;
    else first = q;

    return *this;
}
```

主要内容

- 字典（有序表）的定义
- **散列表**
  - **定义**
  - **散列函数**
  - **解决冲突的方法**
    - **开地址法：线性、双散列**
    - **链表法**

H1. 散列

- 散列法（Hash）
  - 哈希法、杂凑法
  - 在表项的存储位置与表项关键字之间建立一个确定的对应函数关系Hash()，使每个关键字值与结构中的一个唯一的存储位置相对应

    Address = Hash(key)
  - **插入时**，依此函数计算存储位置并按此位置存放
  - **查找时**，对元素的关键字进行同样的函数计算，把求得的函数值当做元素的存储位置，在此结构中按此位置取元素比较，若关键字值相等，则查找成功

散列

- 在散列法中使用的转换函数叫做**散列函数**
- 按此种思想构造出来的表或结构叫做**散列表**
- 散列表的适用范围
  - key的取值范围比较宽泛
  - 待处理的key值不多
  - 存储空间有限
  - 特别适用于需要**快速查找**的问题

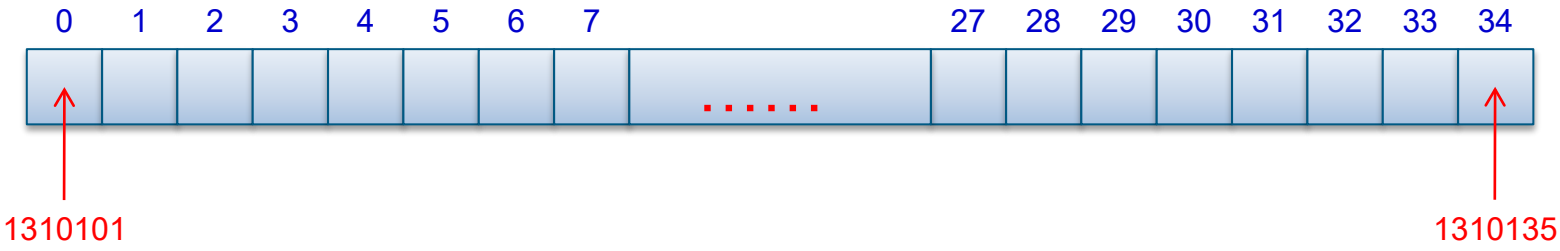
Hash VS 传统查找

- 传统查找
  - 记录在数据结构中的位置是**随机**的
  - 和记录的关键字之间不存在确定关系
  - 查找记录时需要进行一系列的比较，效率依赖于比较的次数
- Hash
  - **能否不经过比较，一次存取即得到所需数据？**
  - 必须在记录的存储位置和关键字之间建立一个确定的对应关系 f，使得每个关键字和一个唯一的存储位置对应。

最简单的散列

- 假设某班有学生35名
- 关键字：学号1310101-1310135

  h(k)=k-1310101 （线性函数、无冲突）



一个更一般的散列

- 我国有省级行政区30+个
- 关键字：拼音，可能的组合26^{12}+
  - BEIJING
  - TIANJIN
  - SHANDONG
  - HEILONGJIANG
  - HEBEI
  - ……

散列表的适用范围

- key的取值范围比较宽泛
- 待处理的key值不多
- 存储空间有限
- 特别适用于需要快速查找的问题

省级区划的Hash函数

1. 取关键字中第一个字母在字母表中的序号作为Hash函数
2. 先求关键字的第一个和最后一个字母在字母表中的序号之和，然后判别这个和值，若比30（假设为表长）大，则减去30
3. 先求每个汉字的第一个拼音字母的ASCII码之和的八进制形式，然后将这个八进制数看成是十进制再除以30取余数

省级区划的Hash结果

key	BEIJING 北京	TIANJIN 天津	HEBEI 河北	SHANXI 山西	SHANGHAI 上海	SHANDONG 山东	HENAN 河南	SICHUAN 四川
$f_1(key)$	02	20	08	19	19	19	08	19
$f_2(key)$	09	04	17	28	28	26	22	03
$f_3(key)$	04	26	02	13	23	17	16	16

Hash函数的设定很灵活，只要使得任何关键字由此所得的哈希值落在表长范围内即可

对不同关键字可能得到同一哈希地址，即key1≠key2，而f(key1)=f(key2)，这被称为冲突

Hash的两个关键问题

- 结论
  - 根据key的特性，选取合适的Hash函数可以尽量减少冲突
  - 一般来说冲突不能完全避免，必须有处理机制
- **Hash函数如何设计和选取？（如何建立从关键字到存储位置的映射关系？）**
- **如果经过Hash函数的运算，多个关键字被映射到同一个存储位置（发生冲突），该怎么办？**

关键问题一：构造Hash函数

- 好的Hash函数
  - 定义域必须包括所有关键字，值域必须在表长范围之内
  - 若对于关键字集合中的任一个关键字，经Hash函数映像到地址集合中任何一个地址的概率是相等的，则称此类Hash函数是均匀的。换句话说，就是使关键字经过Hash函数得到一个“随机地址”，以便使一组关键字的哈希地址均匀分布在整个地址区间中，从而减少冲突
  - 尽量简单，计算时间尽量少

直接定址法

- 对关键字做一个线性计算，把计算结果当做散列地址

```
Hash(key)= a * key + b
```

- 特点
  - 计算简单
  - 没有冲突发生
  - 太理想，很少有应用场景

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，可能在某些位上分布均匀些，在另一些位上不均匀。
- 则应根据已知关键字集合的特点，选取出那些分布均匀（冲突较少）的位进行哈希映射

数字分析法示例

80个数

			……				
8	1	3	4	6	5	3	2
8	1	3	7	2	2	4	2
8	1	3	8	7	4	2	2
8	1	3	0	1	3	6	7
8	1	3	2	2	8	1	7
8	1	3	3	8	9	6	7
8	1	3	5	4	1	5	7
8	1	4	6	8	5	3	7
			……				
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)

哈希表长度为100，即有100个存储地址。

可取第4、5两位组成一个两位数作为Hash地址

平方取中法

- 取关键字平方后的中间几位为哈希地址
  - 通常在选定哈希函数时不一定能知道关键字的全部情况，取其中哪几位也不一定合适
  - 而一个数的平方后的中间几位数和原数的每一位都相关，由此使随机分布的关键字得到的哈希地址也是随机的
  - 取的位数由表长决定

平方取中法示例

假设有此8进制编码

A	B	C	……	Z	0	1	2	……	9
01	02	03		32	60	61	62		71

记录	关键字	$(关键字)^2$	哈希地址
A	0100	0 010 000	010
I	1100	1 210 000	210
J	1200	1 440 000	440
I0	1160	1 370 400	370
P1	2061	4 310 541	310
P2	2062	4 314 704	314
Q1	2161	4 734 741	734
Q2	2162	4 741 304	741

折叠法

- 将关键字分割成位数相同的几部分（最后一部分的位数可以不同），然后取这几部分的叠加和（舍去进位）作为哈希地址
  - 可以从左向右分割，也可以从右向左分割
  - 一般分割出的位数将与散列表地址位数相同
  - 适用于关键字位数很多的情况（**显然此时数字分析和平方取中均不合适**）

折叠法的分类

- 移位叠加法
  - 把各部分的最后一位对齐相加
- 间界叠加法（分界叠加法）
  - 各部分不折断，沿各部分的分界来回折叠，然后对齐相加

折叠法示例

- 每一种西文图书都有一个国际标准图书编号（ISBN），是一个10进制数字（假设）
- 当图书馆藏书种类不到10 000时，可以构造一个四位数的哈希函数
- 如ISBN：0-442-20586-4

```
      5864
      4220
+）     04
------------
     10088

H(key) = 0088
```

```
      5864
      0224
+）     04
------------
      6092

H(key) = 6092
```

除留余数法

- 取关键字被某个不大于哈希表表长m的数p除后所得余数为哈希地址【最常用的方法】

$$H(key)=key \ \% \ p, \quad p<=m$$

- 其中，p一般取：
  - 最接近m的质数
  - 或者不包含小于20的质因数的合数

关键问题二：处理冲突

- 线性开型寻址法（线性探测法）
- 二次探测法
- 双散列法（再哈希法）
- 链表法（拉链法、链地址法）

线性开型寻址法

- 开地址法
  - 可存放新表项的空闲位置既向它的同义词表项开放，又向它的非同义词表项开放
  - 这里的同义词是指那些散列地址相同的不同关键字

线性开型寻址法

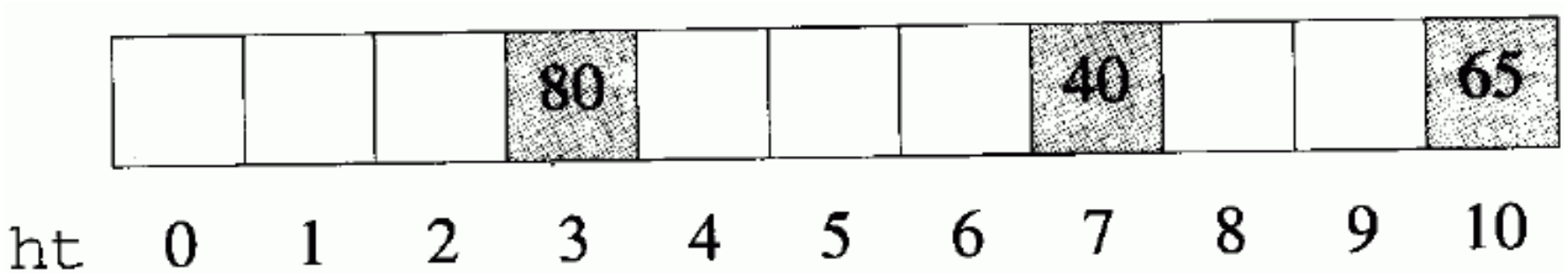
- 使用某一种散列函数计算出初始散列地址H0，一旦发生冲突，在表中顺次向后寻找“下一个”空闲位置Hi的公式为：

$$H_i=(H_{i-1}+d)\ \%\ m$$

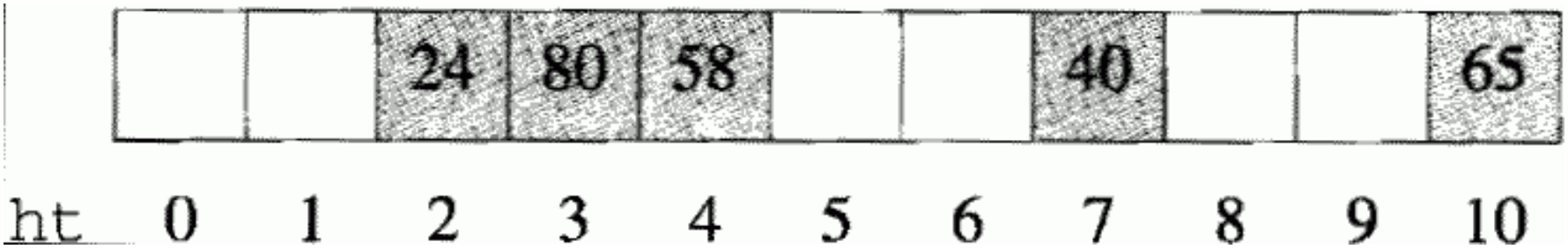
- 其中，d=1

线性探测法示例

- hash函数：除留余数法，h(k) = k % m
- 例：桶数m=11

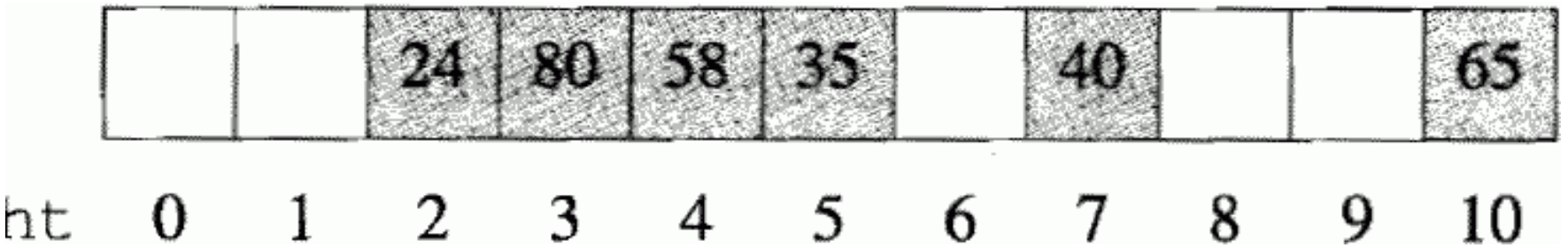


线性探测法实例



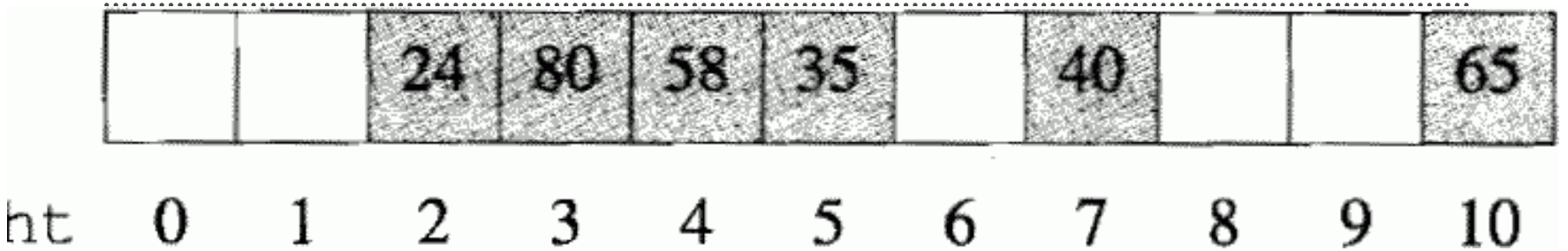
- 接上例，hash表中已保存了80和40
- 插入58，58%11=3，与80冲突，从4开始检测空桶，插入位置4

搜索操作



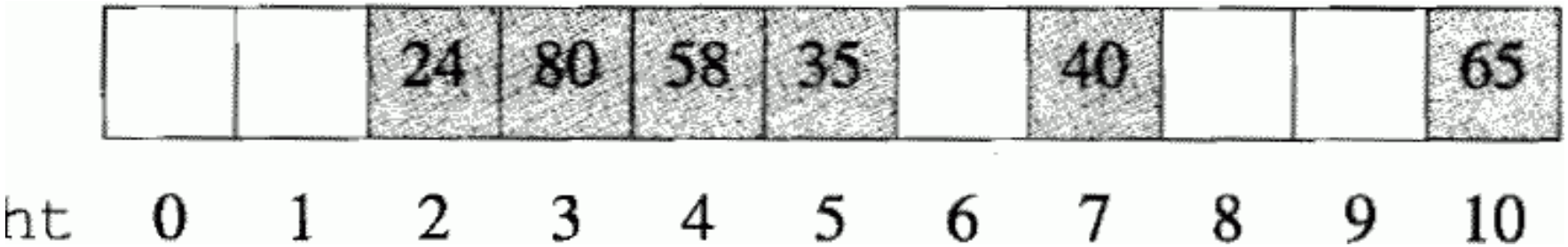
- 插入35，经过几次冲突，最终放置于哈希地址5
- 从h(k)开始顺序检查，直到某个桶满足：
  - 关键字与目标关键字相同，搜索成功
  - 空桶或回到h(k)，搜索失败

搜索操作



- 搜索35，h(35)=2
  - 2号桶，关键字不符；3号，不符；4号，不符；5号，成功
- 搜索46，h(46)=2
  - 2号-5号，不符；6号为空，失败

删除操作



- 不能简单删除，会影响后续搜索操作
  - 删除80，会造成58、35搜索失败
  - 删除58，会造成搜索35失败

HashTable类

```
template<class E, class K>
class HashTable {
  public:
    HashTable(int divisor = 11);
    ~HashTable() {delete [] ht;  delete [] empty;}
    bool Search(const K& k, E& e) const;
    HashTable<E,K>& Insert(const E& e);
    void Output();// output the hash table
```

HashTable类

```
private:
    int hSearch(const K& k) const;
    int m; // hash function divisor
    E *ht; // hash table array
    bool *empty; // 1D array
};
```

构造函数

```
template<class E, class K>
HashTable<E,K>::HashTable(int divisor)
{// Constructor.
   m = divisor;

   // allocate hash table arrays
   ht = new E [m];
   empty = new bool [m];

   // set all buckets to empty
   for (int i = 0; i < m; i++)
      empty[i] = true;
}
```

辅助函数hSearch

```
template<class E, class K>
int HashTable<E,K>::hSearch(const K& k) const
{// Search an open addressed table.
 // Return location of k if present.
 // Otherwise return insert point if there is space.
   int i = k % m; // home bucket
   int j = i;     // start at home bucket
   do {
      if (empty[j] || ht[j] == k) return j;
      j = (j + 1) % m;  // next bucket
      } while (j != i); // returned to home?

   return j;  // table full
}
```

三种返回情况：
1、empty[b]=true，可插入该位置
2、ht[j]=k，重复
3、ht[b]<>k，且empty[b]=false，表满

搜索函数Search

```
template<class E, class K>
bool HashTable<E,K>::Search(const K& k, E&
  e) const
{// Put element that matches k in e.
 // Return false if no match.
   int b = hSearch(k);
   if (empty[b] || ht[b] != k) return false;
   e = ht[b];
   return true;
}
```

插入操作

```
template<class E, class K>
HashTable<E,K>& HashTable<E,K>::Insert(const E& e)
{// Hash table insert.
  K k = e; // extract key
  int b = hSearch(k);

  // check if insert is to be done
  if (empty[b]) {empty[b] = false;
                 ht[b] = e;
                 return *this;}

  // no insert, check if duplicate or full
  if (ht[b] == k) throw BadInput(); // duplicate
  throw NoMem(); // table full
  return *this;  // Visual C++ needs this line
}
```

线性探测法的特点

- 优点
  - 简单
  - 只要表不满，总可以找到空位，插入成功

线性探测法的特点

- 缺点
  - 聚集问题
    $h(k_1)=i$，$h(k_2)=j$，k_1可能占据k_2的hash表位置，从而可能在局部造成严重的聚集，性能急剧下降，即便hash表还很空
  - 而当表较满时，性能几乎一定会很差

复杂性分析

- 初始化： $\Theta(m)$
- 搜索、插入最坏情况： $\Theta(n)$

平均情况

- U_n：一次不成功搜索平均检查的桶的数目
- S_n：一次成功搜索平均检查的桶的数目

$$U_n \approx \frac{1}{2}\left[1+\frac{1}{(1-\alpha)^2}\right] \qquad S_n \approx \frac{1}{2}\left[1+\frac{1}{1-\alpha}\right]$$

- α=n/m：负载因子——hash表满的程度
  - 0.5：S_n=1.5，U_n=2.5
  - 0.8：S_n=5.5，U_n=50.5

二次探测法

- 线性探测的缺点：聚集——h(k)不相同的（相近的）关键字发生冲突
- 平方探测法：$d=i^2$——
  探测h(k)、h(k)+1、h(k)+4、…

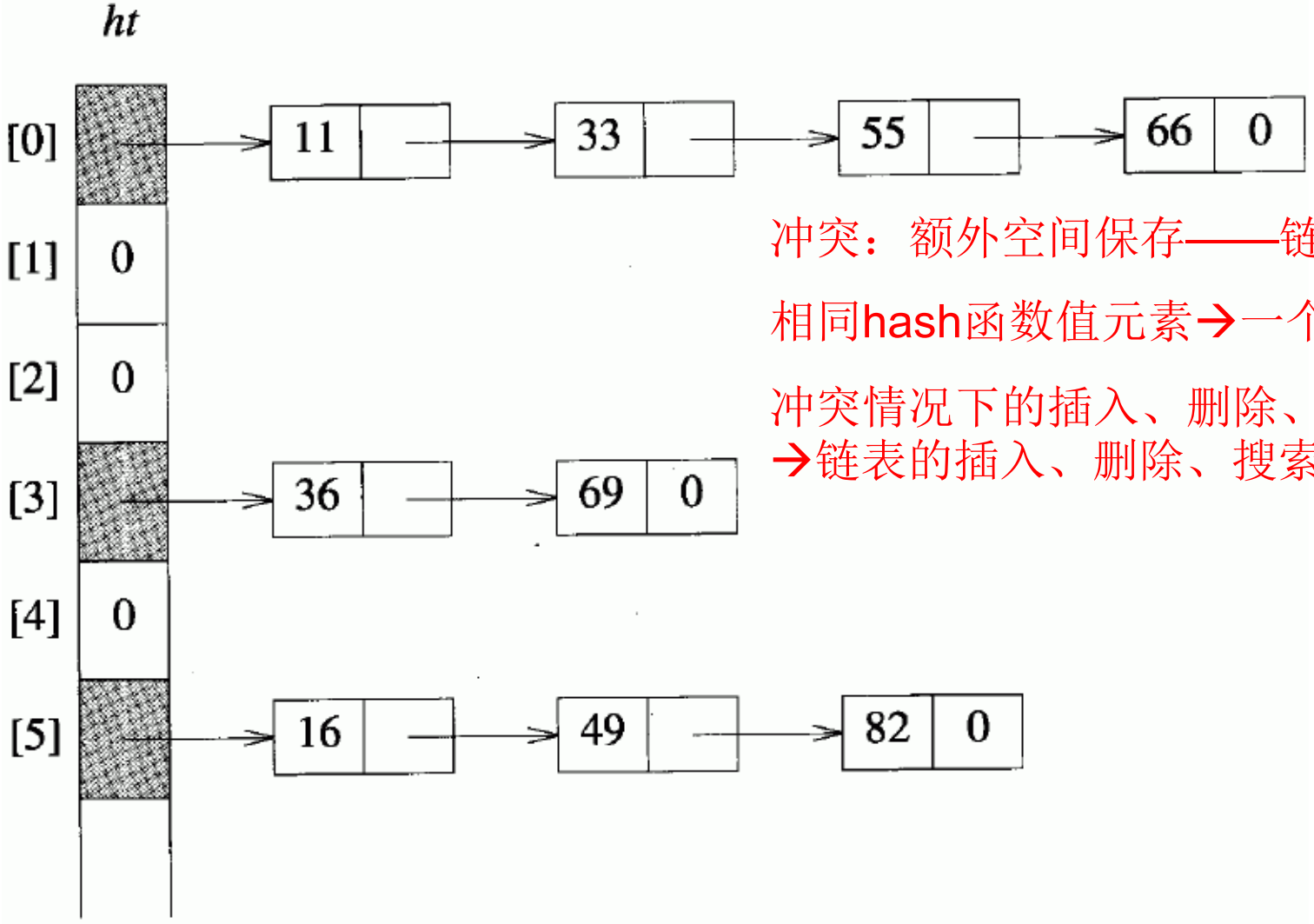
与线性探测的比较

- 解决了局部聚集问题
- 缺点：在表不满的情况下，也不能保证插入肯定成功

双散列法

- 需要两个散列函数
- 第一个散列函数计算关键字的首选地址
- 一旦发生冲突，用第二个散列函数计算到下一地址的增量；或者直接计算下一个地址
- 双散列法将冲突处理也“随机化”，避免了“聚集”

链表法



冲突：额外空间保存——链表

相同hash函数值元素→一个链表

冲突情况下的插入、删除、搜索→链表的插入、删除、搜索

链表法思想

- 通过哈希函数计算具有相同哈希地址的元素串在一个链表当中（归于一个子集合）
- 正常情况下，每个同义词链表长度都比较短，设有n个关键字存放到长度为m的散列表中，则每一个同义词链平均长度是n/m，效率可以接受
- 另一个优点是：除了解决冲突外，还可以解决溢出问题。

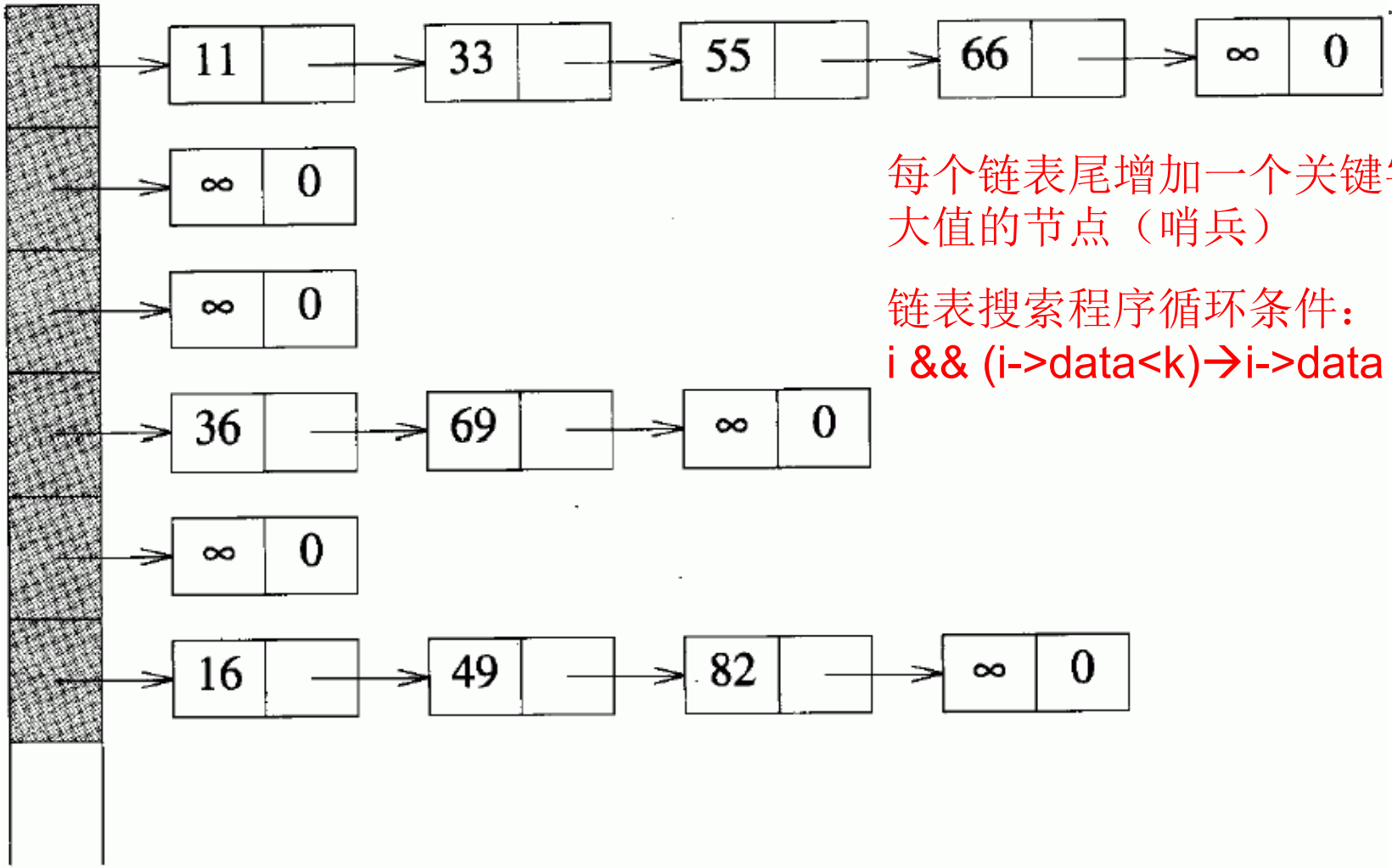
ChainHashTable类

```
template<class E, class K>
class ChainHashTable {
   public:
      ChainHashTable(int divisor = 11)
         {m = divisor; ht = new SortedChain<E,K>
     [m];}
      ~ChainHashTable() {delete [] ht;}
      bool Search(const K& k, E& e) const
          {return ht[k % m].Search(k, e);}
```

ChainHashTable类

```
ChainHashTable<E,K>& Insert(const E& e)
    {ht[e % m].DistinctInsert(e); return *this;}
ChainHashTable<E,K>& Delete(const K& k, E& e)
    {ht[k % m].Delete(k, e); return *this;}
void Output() const;   // output the table
private:
int m;                 // divisor
SortedChain<E,K> *ht;  // array of chains
};
```

一点改进



每个链表尾增加一个关键字为极大值的节点（哨兵）

链表搜索程序循环条件：

i && (i->data<k)→i->data < k

溢出链表法时间复杂性

- 链表长度为i，不成功搜索平均操作次数

$$\frac{1}{i}\sum_{j=1}^{i} j = \frac{i(i+1)}{2i} = \frac{i+1}{2}$$

- 平均链表长度n/m=α，

  代入上式 $U_n = \frac{\alpha+1}{2}, \alpha \geq 1$

  α<1时，$U_n \leqslant \alpha$

溢出链表法时间复杂性（续）

- 成功搜索——考虑关键字在链表中位置
  - 不妨假定关键字按升序插入
  - 插入第i个关键字时，链表平均长度(i-1)/b
  - 而此关键字插入某个链表末尾，其搜索代价为1+(i-1)/b，因此有

$$S_n = \frac{1}{n}\sum_{i=1}^{n}(1+(i-1)/m) = 1+\frac{n-1}{2m} \approx 1+\frac{\alpha}{2}$$

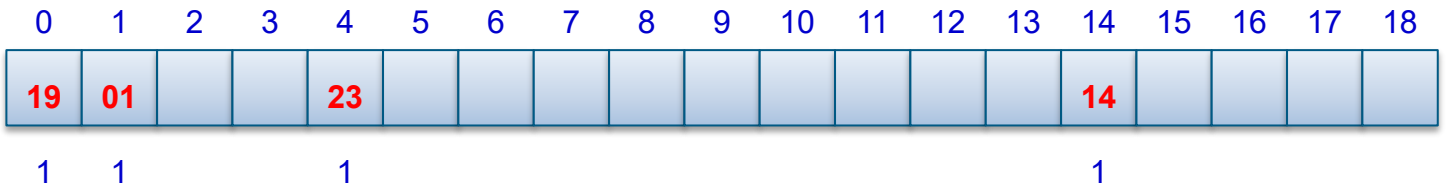
H1小结

- 散列表的两大关键
  - 散列函数

    $h(k) = k \% D$
  - 解决冲突的策略
    - 线性开型寻址：简单，但容易造成堆积
    - 双散列开型寻址：稍复杂，可部分解决堆积（更随机）
    - 链表法

例题

- 设一个哈希表的地址区间为0-18，哈希函数为H(K)=K mod 19。采用线性探测法处理冲突，请将关键字序列19，14，23，01，68，20，84，27，55，11，10，79，12依次存储到哈希表中，画出结果，并计算平均查找长度。



课堂练习

- 假定关键字K=2789465，允许存储地址为三位十进制数，现得到的散列地址为149，则所采用的构建哈希函数的方法是_____。

A. 除留余数法，模为23

B. 平方取中法

C. 移位叠加

D. 间界叠加

课堂练习

- 为提高散列（Hash）表的查找效率，可以采取的正确措施是_____。

  Ⅰ. 增大装填因子

  Ⅱ. 设计冲突少的散列函数

  Ⅲ. 处理冲突时避免产生聚集现象

A. 仅Ⅰ

B. 仅Ⅱ

C. 仅Ⅰ、Ⅱ

D. 仅Ⅱ、Ⅲ

本章结束

5-10章小结

- 五种数据结构
  - 线性表、矩阵、堆栈、队列、字典
- 六种排序算法
  - 计数、选择、冒泡、插入、箱子、基数
- 三种查找算法
  - 顺序、二分、哈希

5-10章小结

- 特别重要的知识点
  - 线性表（尤其是**链表**）的含义、描述、操作、性能分析
  - 特殊矩阵和稀疏矩阵的表示
  - 栈和队列的原理、操作
  - Hash过程