第 10 章

跳表和散列
（Skip List and Hashing）

本章内容

- **10.1 字典**
- **10.2 抽象数据类型**
- **10.3 线性表描述**
- 10.4 跳表描述
- **10.5 散列表描述**
- 10.6 应用

10.1 字典

- **字典**（dictionary）是一些形如**(*k*,*v*)**的数对(元素/记录)所组成的集合。其中*k*是**关键字**，*v*是关键字*k*对应的**值**。任意两个数对，其关键字各不相同。
- 有关字典的操作有：
  - 确定字典是否为空。
  - 确定字典有多少数对。
  - 在字典中寻找/搜索具有给定关键字的数对。
  - 插入一个数对。
  - 删除一个具有给定关键字的数对。

●**多重字典**：两个或多个数对有相同的关键字 。

10.2 抽象数据类型

抽象数据类型 *Dictionary*{

实例

关键字互不相同的数对(元素)集合

操作

empty(): 字典为空时返回true,否则返回false

size(): 返回字典中的数对个数

find(*k*): 返回关键字为*k*的数对

insert(*p*): 插入数对p

erase(*k*): 删除关键字为*k*的数对

}

C++抽象类dictionary

```
template <class K, class E>
class dictionary
{
  public:
    virtual ~ dictionary() {}
    virtual bool empty() const = 0;
          //字典为空时返回true，否则返回false
    virtual int size() const = 0;
          //返回字典中数对的个数
    virtual pair<const K,E>* find (const K&) const = 0;
          //返回匹配数对中的指针;
    virtual void erase(const K&) = 0;
          //删除匹配的数对
    virtual void insert(const pair<const K,E>&) = 0;
          //在字典中插入一个数对
}
```

访问字典数对

- 随机访问(Random Access)
  - 按照给定的一个关键字来访问字典中的数对 。
- 顺序访问（ Sequential Access)
  - 按照关键字的递增顺序逐个访问字典中的数对。
  - 顺序访问需要操作:
    - *Begin* –用来返回关键字最小的数对
    - *Next* –用来返回下一个数对

字典的线性结构描述

- 字典的线性结构描述：
  - **线性表描述**
  - 跳表描述
  - **散列表描述**

10.3 线性表描述

- 字典用有序线性表: $(p_0, p_1, p_2, \ldots, p_{n-1})$,表示
  - 关键字从左到右依次增大

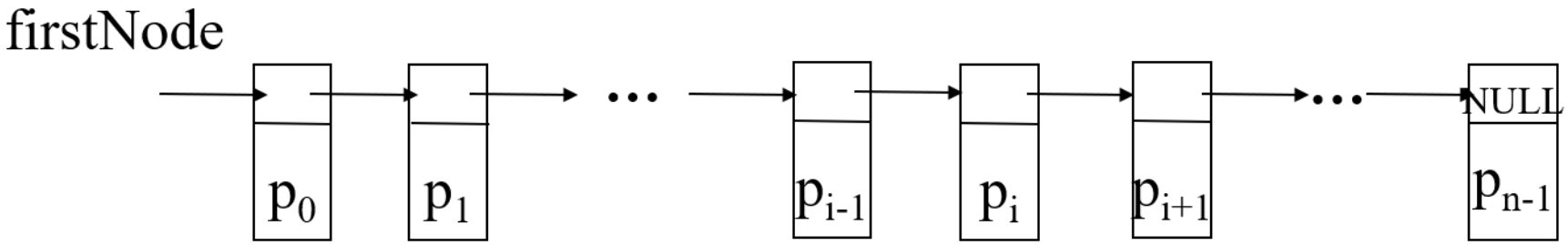
- 使用线性表的描述方法
  - 数组描述： sortedArrayList
  - 链表描述：sortedChain

数组描述的有序线性表sortedArrayList

- 用数组描述的有序线性表类sortedArrayList
  - 搜索:折半搜索，时间为O(logn)。
  - 插入:首先确认该字典中是否存在相同关键字的数对,然后进行插入,插入操作的时间是O(n)。
  - 删除:首先要找到欲删除的数对，然后再进行删除,删除的时间复杂性为O(n)。

链表描述的有序线性表sortedChain

- 有序链表表示字典




```
template<class K, class E>
struct pairNode{
      pair<const K,E>   element;
      pairNode<const K,E> *next;
     ……
} ;
```

类sortedChain

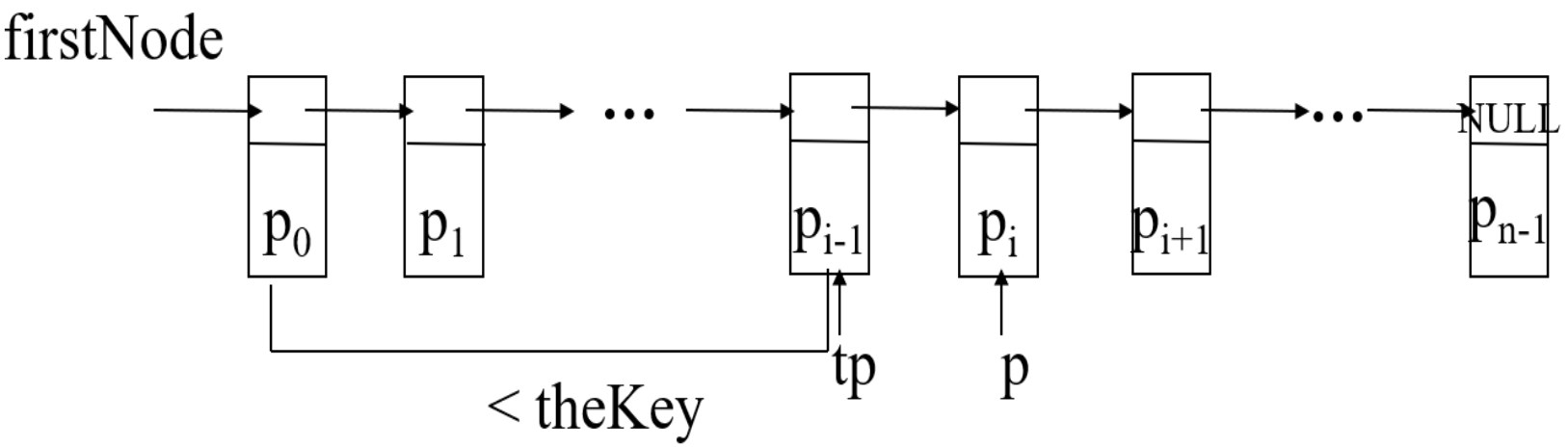
```
template<class K, class E>
class sortedChain
{public :
    ……
    pair<const K,E> * find(const K& theKey) const;
    //返回关键字theKey匹配的数对的指针，若不存在匹配的数对，
      则返回NULL
    void insert(const pair<const K,E>& thePair);
    //插入一个数对thePair,覆盖已经存在的数对
    void erase(const K& theKey);
    //删除关键字theKey匹配的数对
  protected:
    pairNode<const K,E> *firstNode; //指向链表第一个节点的指针
    int  dSize;        //表中的数对个数
}
```

方法 ‘find’

```
template<class E, class K>
pair<const K,E> * find(const K& theKey) const;
{//返回关键字theKey匹配的数对的指针，若不存在匹配的数对，
    则返回NULL
  pairNode<K,E> *currentNode = firstNode;
   // 搜索与theKey相匹配的数对
   while (currentNode !=NULL &&
            currentNode->element.first < theKey)
     currentNode = currentNode->next);

  // 判断是否与theKey匹配
   if  (currentNode !=NULL &&
        currentNode->element.first = = theKey) // 与theKey相匹配
          return &currentNode->element;
   return NULL; // 不存在相匹配的数对
}
```

方法 ‘insert’

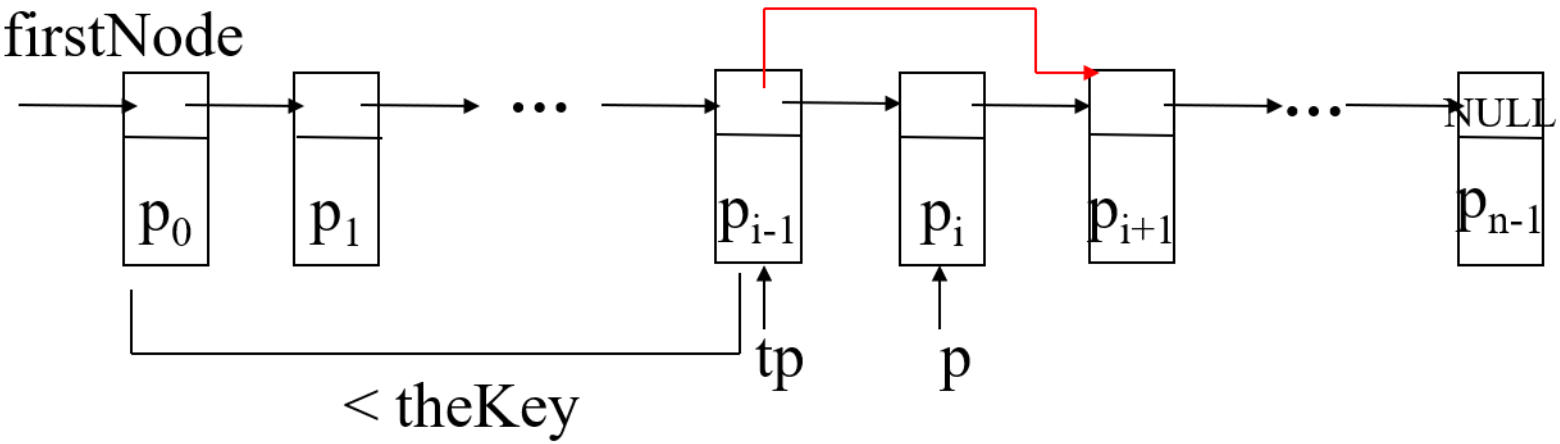
- 在字典中插入一个数对thePair,覆盖已经存在的数对



```
template<class E, class K>
void sortedChain<K,E>::insert(const pair<const K, E>& thePair)
{//在字典中插入一个数对thePair,覆盖已经存在的数对
  pairNode<K,E> *p = firstNode, *tp = NULL; // 跟踪p
  // 移动tp, 使得thePair在tp之后
  while (p!=NULL && p->element.first < thePair.first)
      { tp = p; p = p->next};

  if (p !=NULL&& p->element.first = = thePair.first)
     {// 有匹配的数对，覆盖数对中的值
       p->element.second= thePair.second; return; }
   // 若无匹配的数对，则为thePair建立一个新节点
  pairNode<K,E> *newNode = new pairNode<K,E>(thePair, p);
  if (tp==NULL) firstNode = newNode;
  else tp->next = newNode; // 将新节点插入到tp之后
  dSize++;
  return;
}
```

方法 ‘erase’

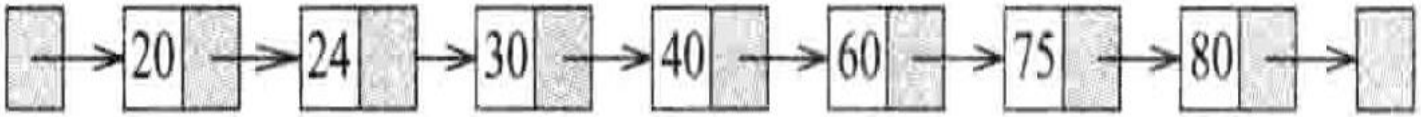
1. firstNode



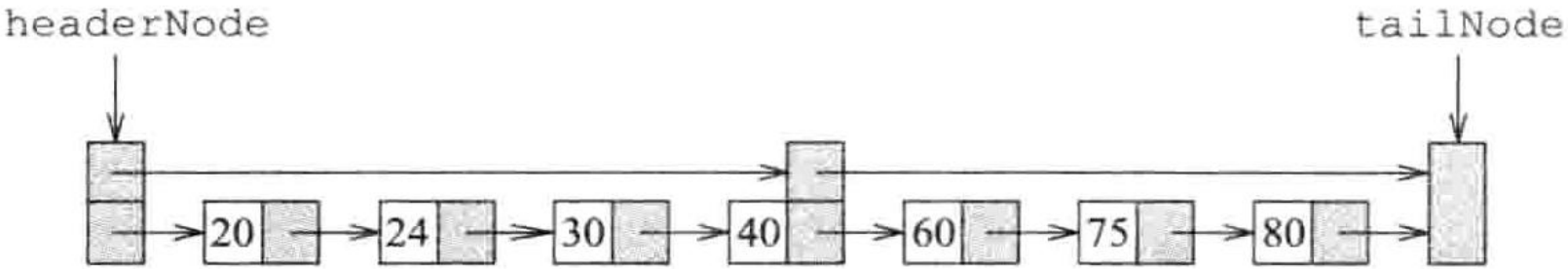
```
template<class E, class K>
void sortedChain<K,E>:: erase(const K& theKey)
{// 删除与theKey相匹配的数对
  pairNode<K,E> *p = firstNode,
                *tp = NULL; //跟踪p

 // 搜索与theKey相匹配的数对
  while(p!=NULL && p->element.first < theKey)
        {tp = p; p = p->next; }

  // 确定是否与theKey匹配
  if (p!=NULL && p->element.first = = theKey)
     {// 找到一个相匹配的数对
      if (tp==NULL) firstNode = p->next; // p是链首节点
      else tp->next = p->next;

       delete p;
       dSize--;}
}
```

10.4 跳表表示

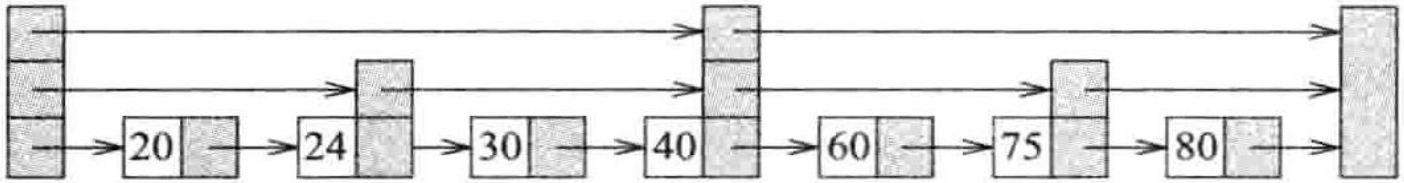
理想情况



搜索：　最多比较次数 = n



搜索：最多比较次数 = n/2+1



理想情况

- 0 级链表: n 个数对（元素/记录）
- 1 级链表: ½ n 个数对
- 2 级链表 ¼ n 个数对
- ……
- i 级链表: $n/2^i$ 个数对

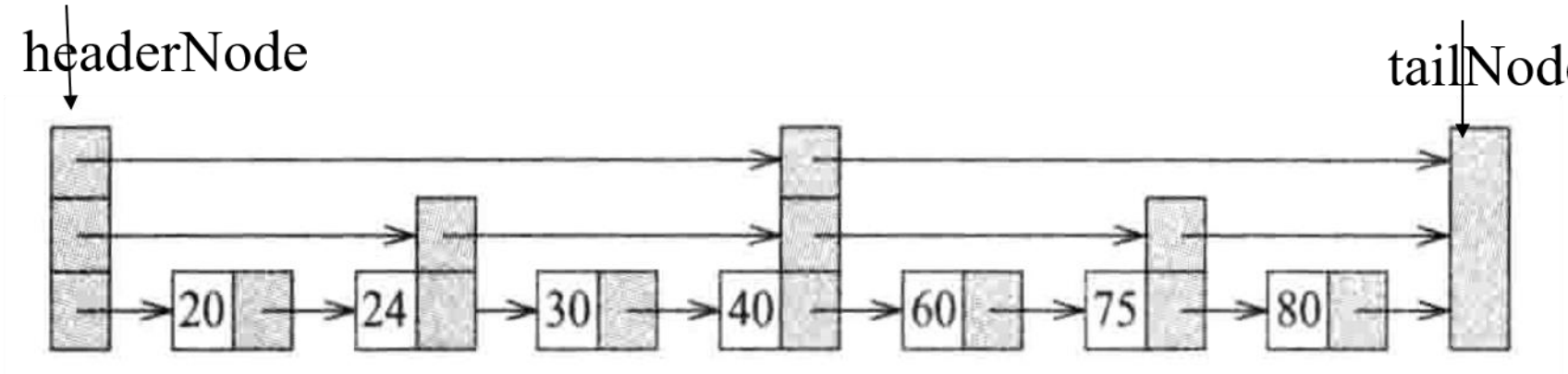
- 一个数对是 ***i*** 级链数对

  当且仅当 它属于 **0 ~ *i*** 级链表

  但不属于***i+1***级链表（若该链表存在）

- ***i*** 级链所包含的数对是***i*-1**级链所有数对的子集

搜索



- 从最高级链表开始，在每一级链中搜索，遇到最后一个小于theKey的数对，跳到下一级链表中搜索，直到0级链表

插入和删除

- 当插入和删除时，要保持跳表的规则结构
- 跳表的规则结构：
- *i* 级链表： $n/2^i$ 个数对
  - ***i-1***级数对属于***i***级链的概率是½
  - 一个数对属于i级链概率是$1/2^i=(1/2)^i$)
  - 链的级数是：$\lfloor \log_2 n \rfloor + 1$

插入和删除

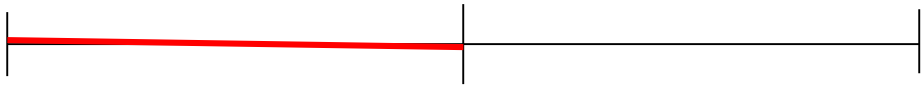
- 跳表的规则结构：
  - ***i -1*** 级链中的数对属于***i***级的概率为***p***
  - 每***1/p***个***i -1*** 级链中的数对有一个在***i***级

- 插入一个新数对时，新数对 被插入到***i*** 的概率是 p^i

- 链的级数：$\lfloor \log_{1/p} n \rfloor + 1$

级的分配

- 应用一个随机数产生器，产生0 到RAND_MAX之间的实数



0　　cutoff　　RAND_MAX

=p* RAND_MAX

●产生的随机数≤ cutoff (= p* RAND_MAX)的概率是 *p*

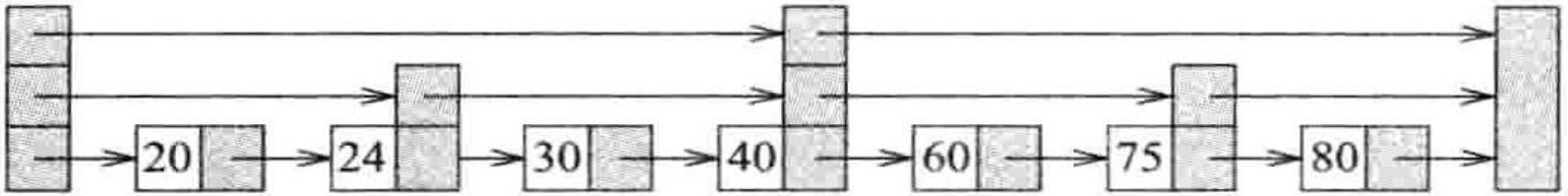
● 分配级数：

```
int lev = 0
while (rand() ≤ CutOff) lev++;
```

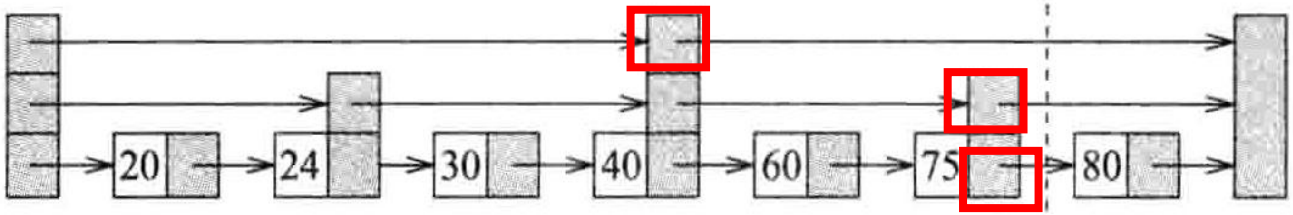
级的分配

- 缺点1：可能为某些数对分配特别大的级，从而导致一些数对的级远远超过$\log_{1/p}N$(N为字典中预期的最大数目 ）
  - 在有N个数对的跳表中，级的最大值 MaxLevel=$\lceil \log_{1/p}N \rceil -1$，以此值作为上限。
- 缺点2：可能存在下面的情况，如在插入一个新数对前有三条链，而在插入之后就有了10条链。这时，新插入数对的级为9级，……。
  - 我们可以把新数对的级调整为3。即把新数对的级调整为**最大级数**+1。

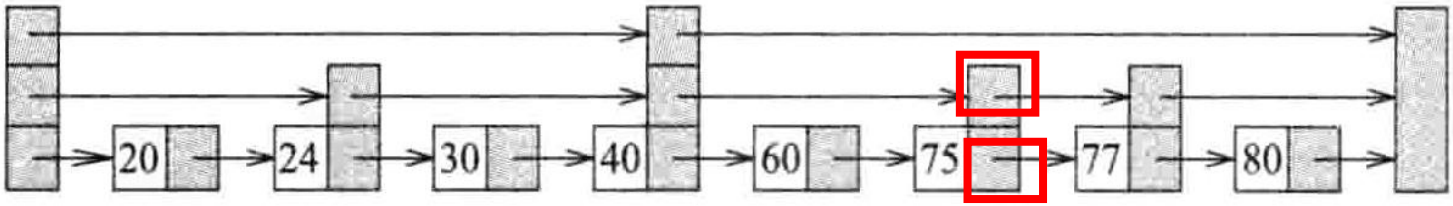
插入



- 插入77
- 搜索77

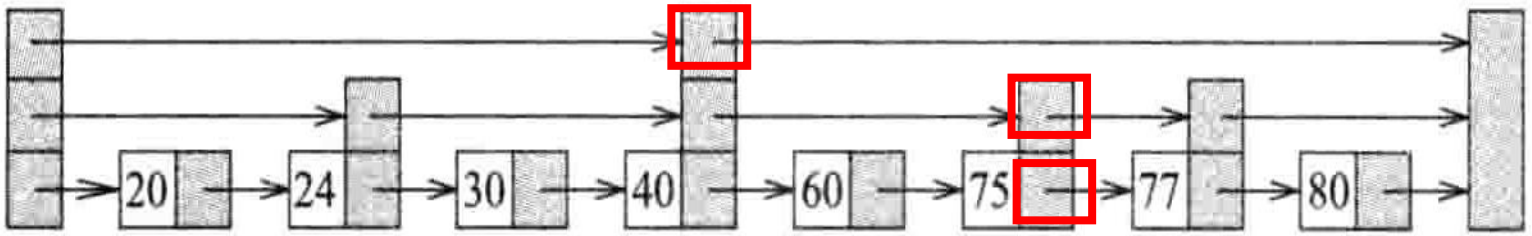


d）查找77时遇到的最后一个指针

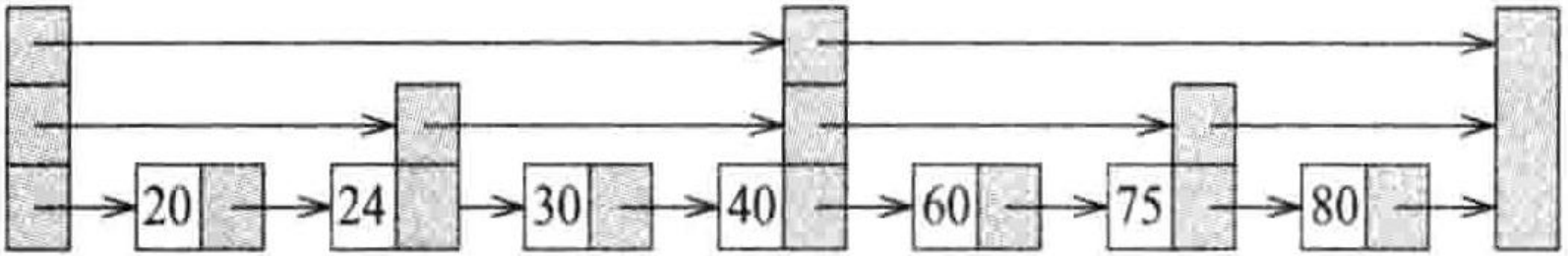


e）插入77

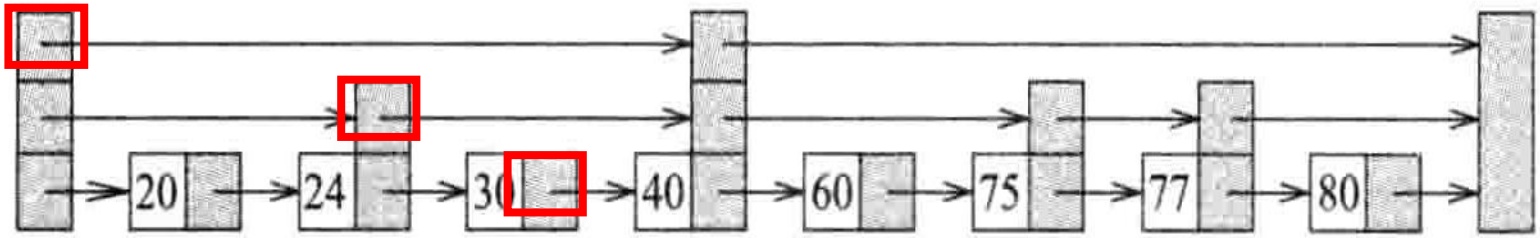
删除



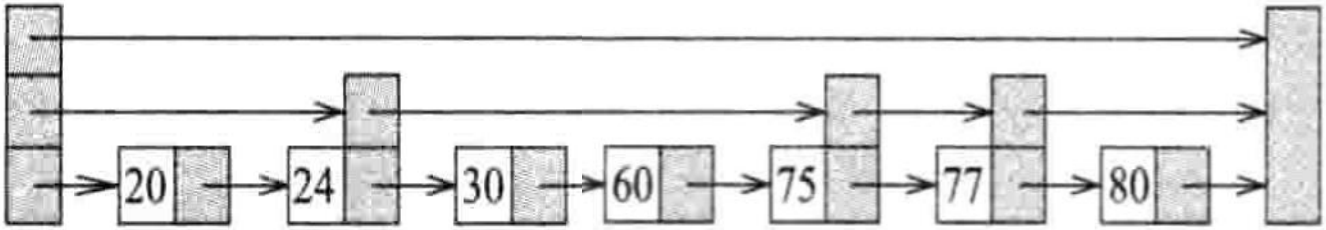
- 删除77



删除



- 删除40



- 删除操作，无法控制跳表结构

10.5 散列表描述

- 10.5.1 理想散列
- 10.5.2 散列函数和散列表
- 10.5.3 线性探查
- 10.5.4 链式散列

10.5.1 理想散列

- **散列(Hashing)** :字典的另一种描述方法就是散列.
- 散列方法：是用一个**散列函数(hash function又称哈希函数)**把字典的数对映射到**散列表/哈希表(hash table)**中的具体位置。
- 理想情况 :
  - 数对p 的关键字为k
  - f 是散列函数

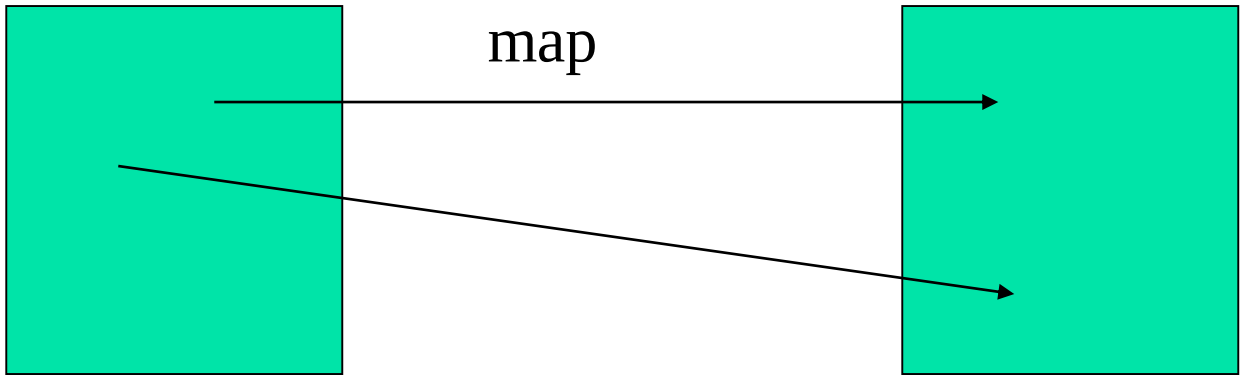
→ 数对p在散列表中的位置为$f(k)$

理想散列

- 假定散列表中的每一个位置最多只能存放一个数对（记录）
- 散列表操作:
  - **查找（find）**:计算出$f(k)$，然后看表中$f(k)$处是否有关键字为k的数对。
  - **插入（insert）**:计算出$f(k)$，把数对放在$f(k)$位置
  - **删除（erase）**:计算出$f(k)$， 把表中$f(k)$位置置为空
- 在理想情况下, 散列表操作时间复杂性： $\Theta(1)$

理想散列

- 例：
  - Key =学生的学号(12个数字)
- key: f(key):



关键字的范围：
20170000000~
20179999999

f(key)的范围：
0~99999999

•关键字的范围太大，因此不能像理想散列方法那样建立数组．

理想散列

- 当关键字的范围太大，不能用理想方法表示时，可以采用比关键字范围小的散列表。

⇨散列函数：多个关键字映射到同一位置。

10.5.2 散列函数和散列表

- **桶(bucket)**：散列表中的每一个位置
- **起始桶（home bucket ）**：对关键字为k的数对，位置 f(k)是起始桶
- **散列表的长度或大小**：桶的数量
-
- 最常用的散列函数:
- **除余散列(Hashing by division)**
  - **f(k) = k % D**　　（%为求模操作符）
  - D: 是散列表的大小(即位置数)
  - 散列表的位置索引：**0~D-1**

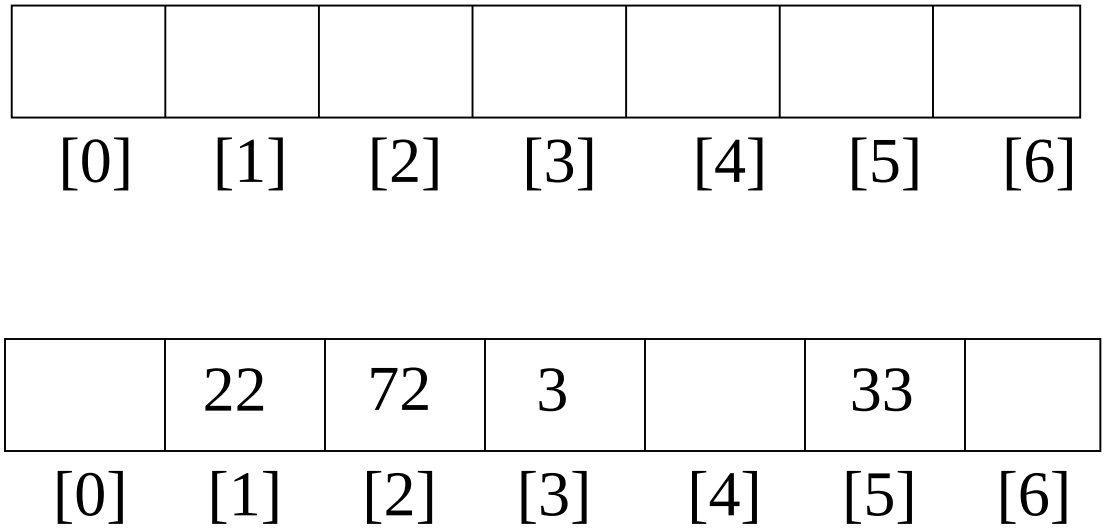
好的散列函数

- 好的散列函数：
  - 均匀散列函数：映射到一个桶里的关键字大致相同
- 良好的散列函数：性能较好的均匀散列函数

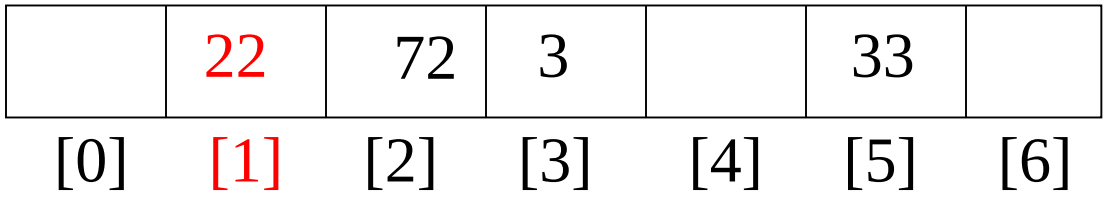
- 除余散列：f(k) = k % D
  - D的选择对于散列的性能有着重大的影响(D等于桶的个数b)。
  - 当D为**素数或D没有小于20的素数因子**时，可以使性能达到最佳 。

例

- 散列表：　table[0:6]
- 散列函数: ***f* (k)= k % 7**
- 依次插入关键字: 22,33,3,72



冲突和溢出（overflow）



- 85 放入什么地方?
- f(85)=1
- 85 的起始桶已经被另一个数对占用

→两个不同的关键字起始桶号相同时——**冲突（ collision）**

- 存储桶中若没有空间时就发生**溢出（overflow）**
- 当每个桶只能存储一个数对时，碰撞和溢出会同时发生.

溢出（overflow）

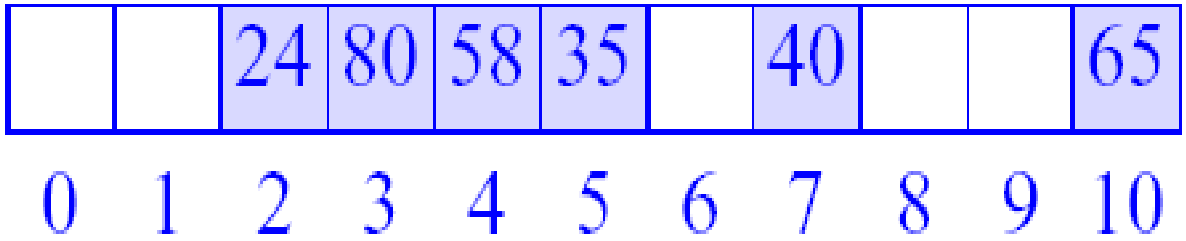
- 解决溢出的方法：
  - 线性探查
  - 链式散列

10.5.2 线性探查

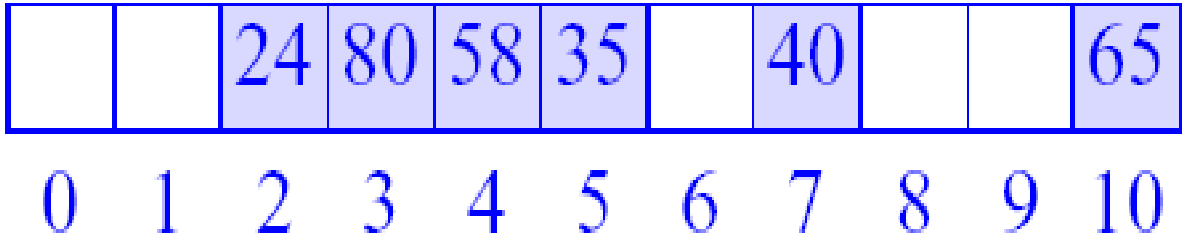
- 当溢出发生时，将元素插入下一个可用桶中。在寻找下一个可用桶时，表被视为环形的。
- 例
  - 散列表的大小b = 11
  - f(k) = k % b
  - 在插入 80, 40, 65后

线性探查—“插入”

- 再依次插入 58，24，35
  - f(58)=3 (冲突)； f(24)= 2； f(35)= 2 (冲突)



线性探查—“搜索”



- 例：搜索24、35、29
- 搜索（search）操作
  - 首先搜索关键字为k的起始桶*f(k)*,接着对表中后继桶进行搜索，直到发生以下情况：**(1)** 存有关键字为k的桶已找到，即**找到**了要搜索的数对。**(2)** 到达一个空桶；**(3)** 又回到起始桶*f(k)*。(2) 和 (3)说明表中**没有关键字为k的数对**。

线性探查—“删除”

- 删除操作
  - 执行搜索操作，找到关键字为k的桶.
  - 在完成一次删除操作后，必须能保证上述的搜索过程仍能够正常进行。不能仅仅把表中相应位置置为空。

- 删除:58; 然后 搜索:35 （f(35)= 2）

–采取办法:

●从欲删除的数对开始逐个检查每个桶以确定要移动的数对，直到到达一个空桶或返回删除操作所对应的桶为止。

●为每个桶增加一个**NeverUsed**域。

线性探查散列表的类

```
template<class K, class E>
class hashTable <K,E>{
public:
    hashTable(int theDivisor = 11);
     ……//~hashTable()、empty()； size()；output

    pair<const K, E>* find(const K&) const;
       //返回关键字theKey匹配的数对的指针，若不
    存在匹配的数对，则返回NULL
    void insert(const pair<const K, E>&);
       //在字典中插入一个数对thePair,若存在关键字
相同的数对,则覆盖
protected:
    int search(const K& ) const;
    pair<const K, E> **table; // 散列表
    int divisor; // 散列函数的除数
    hash<K> hash;//把类型k映射到一个非负整数
    int dSize; // 字典中数对的个数
} ;
```

hashTable的构造函数

```
template<class E, class K>
hashTable<K,E>::hashTable(int theDivisor)
{// 构造函数
    divisor = theDivisor;
    dSize=0;
    // 分配和初始化散列表数组
    table = new pair<const K, E>* [divisor];

    for (int i = 0; i < divisor; i++) //将所有桶置空
        table[i] = NULL;
}
```

hashTable<K,E>::search

```
template<class E, class K>
int hashTable<K,E>::search(const K& theKey) const
{// 搜索一个公开地址散列表，查找关键字为theKey的数对
 // 如果匹配的数对存在，则返回它的位置
 // 否则返回关键字为theKey的数对可以插入的位置
    int i =(int) hash(theKey) % divisor; // 起始桶
    int j = i; // 从起始桶处开始
    do {
        if (table[j] ==NULL || table[j]->first = = theKey) return j;
        j = (j + 1) % divisor; // 下一个桶
        } while (j != i); // 又返回起始桶?
    return j; // 表已经满
}
```

hashTable<K,E>:: find

```
template<class E, class K>
pair<const K, E> * hashTable<K,E>::find(const K& theKey) const
    {//返回关键字theKey匹配的数对的指针
     //若不存在匹配的数对，则返回NULL
    //搜索散列表
    int b = search(theKey);
    //判断table[b]是否是匹配数对
    if (table[b] ==NULL || table[b] != theKey)
        return NULL;   //未找到匹配数对

    return table[b]; //找到匹配数对
}
```

```
template<class E, class K>
void hashTable<K,E>::insert(const pair<const K, E>& thePair)
{//在字典中插入一个数对thePair,若存在关键字相同的数对，
 //则覆盖，若表已满，则抛出异常
    int b = search(thepair.first);
    // 检查匹配的数对是否存在
    //没有匹配的数对，且表不满,则插入数对
    if (table[b]==NULL)
                    {table[b] = new pair<const K, E> (thePair);
                     dSize++; }
    // // 有重复的关键字，则覆盖数对中的值
    if (table[b]->first = = thepair.first)
          table[b]->second=thepair. second;
    else   throw hashTableFull(); // 表满
}
```

性能分析

- 时间复杂性:
- b=散列表中桶的个数
- f(k)=k % D ;　b=D
- 初始化: Θ(b)
- 最坏情况:
  - 所有*n*个关键字值都在同一个桶中
  - 插入和搜索时间均为：Θ(*n*), *n为表中*元素个数
- 散列在最坏情况下的复杂性与线性表在最坏情况下的复杂性是完全相同。

性能分析

- 平均性能:
- 搜索因子(loading factor/装填因子、负载因子):
  - $\alpha = n/b$ ； $0 \le \alpha \le 1$
- S_n = 一次成功的搜索中平均搜索桶的个数
- U_n = 一次不成功的搜索中平均搜索桶的个数
- $S_n \approx \frac{1}{2}(1 + 1/(1-\alpha))$
- $U_n \approx \frac{1}{2}(1+1/(1-\alpha)^2)$

性能分析

α	S_n (桶数)	U_n (桶数)
0.50	1.5	2.5
0.75	2.5	8.5
0.90	5.5	50.5

$\alpha \downarrow \rightarrow S_n, U_n \downarrow$

除余散列中除数D的选择

- *D*的选择对于散列的性能有着重大的影响(*D*等于桶的个数*b*)。

方法 1:

- **确定** U_n**和** S_n
- **使用** U_n 和 S_n**的公式,确定最大的**α**值-----** α_{max}

- n
- α_{max}

$\alpha = n / b$ → b_{min}

●取 D为大于等于b_{min} 最小的整数，该整数要么是素数，要么没有小于20的素数因子。

除余散列中除数D的选择

例 10.12:

- n=1000; S_n<4; U_n<50.5;
- U_n<50.5 ⇒ α ≤0.9
- S_n<4 ⇒ α ≤6/7
- ⇒ α ≤min{0.9,6/7}; α_{max} =6/7
- α =n/b
- b_{min}=n/ α_{max}=7/6n=1167
- D=1171

除余散列中除数D的选择

方法 2:

- 根据散列表的最大空间确定b_{max} (b的最大可能值)
- 取*D* 为不大于这个最大值的整数，该整数要么是素数，要么没有小于20的素数因子。

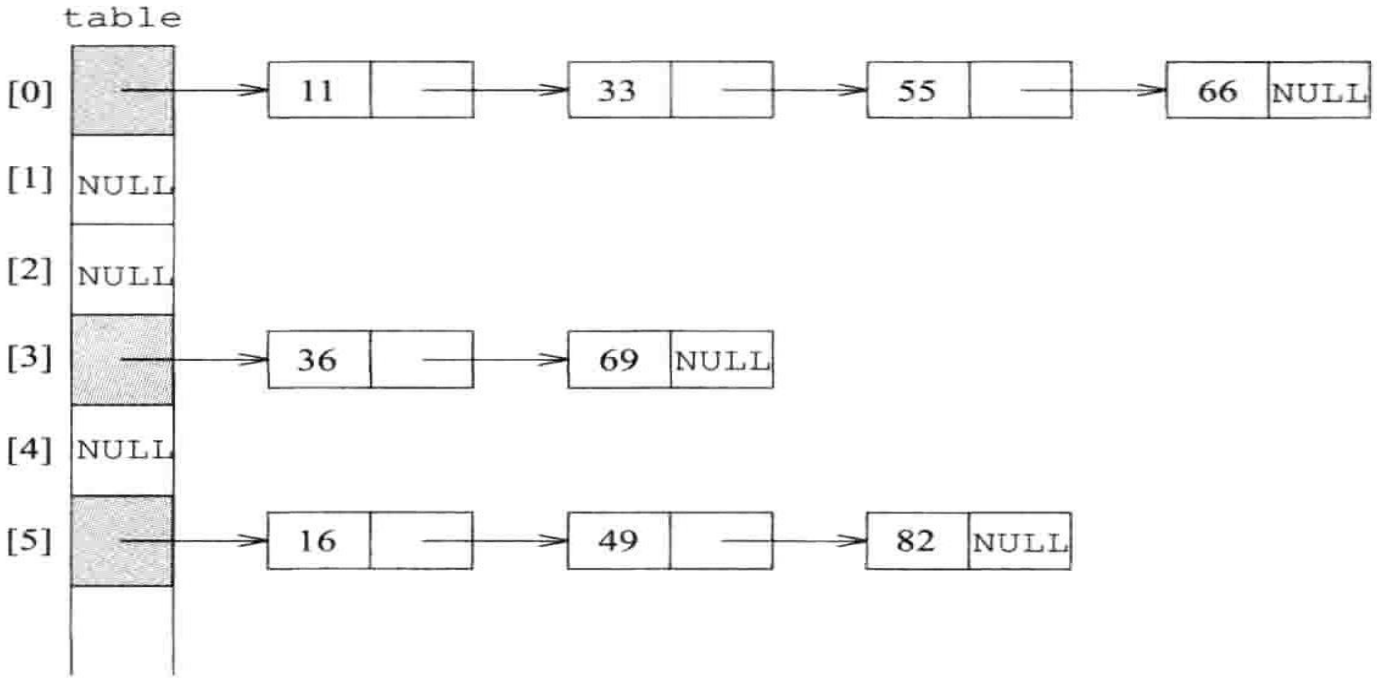
- 散列表的最大空间b_{max} =530
- 取D=23*23=529

练习

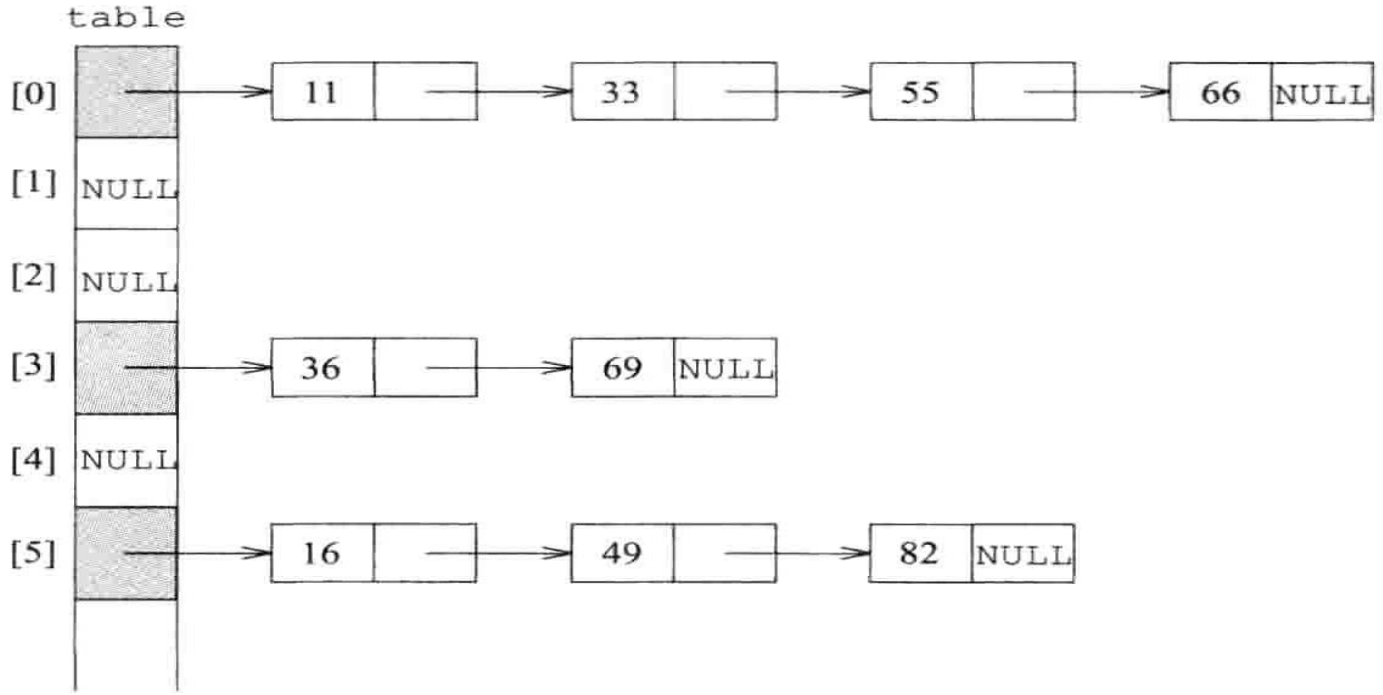
- 1.设有一组关键字：{19，14，23，01，68，20，84，27，55，11，10，79}，采用哈希函数：
- H（key）=key mod 13，采用线性开型寻址方法解决溢出。
- 要求：在0∽12的散列地址空间中对该关键字序列构造哈希表。搜索元素27，55所花的比较次数各是多少？

- 2.有关键字集合：{19，14，23，01，68，20，84，27，55，11，10，79}，哈希函数：H（key）=key mod 13，采用链地址散列方法解决溢出。要求：
- 构造哈希表；搜索元素27，55所花的比较次数各是多少？

10.5.4 链式散列



- 设一个桶可以放无限多个数对，散列表中的桶用一个**链表**来配置。
- 一个链表上的数对是具有同样起始桶的数对。



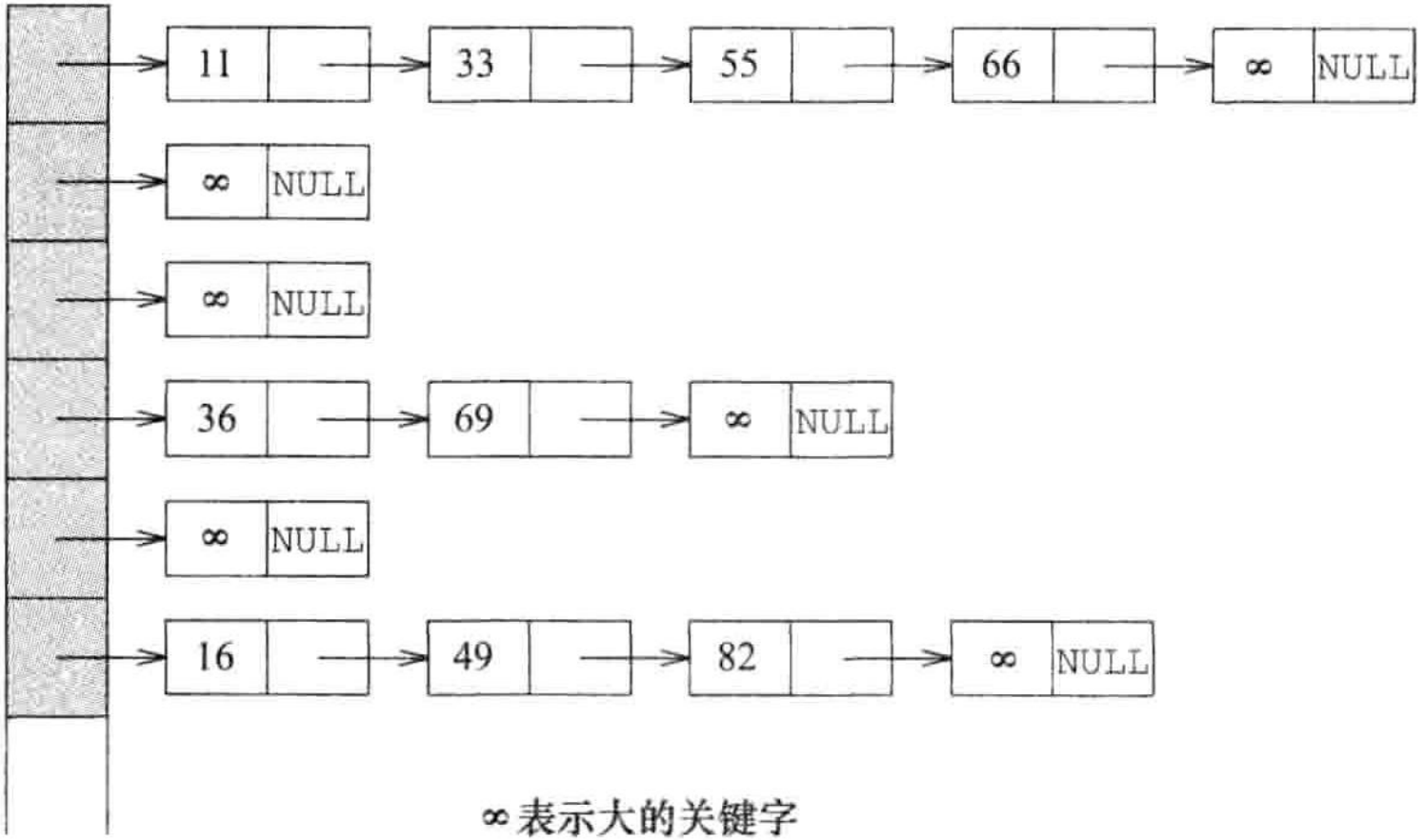
- 查找（find）：
  - 计算 f(k); 搜索该桶所对应的链表.
- 插入（insert）：
  - 计算 f(k); 搜索; 插入.
- 删除（erase）：
  - 计算 f(k); 搜索; 删除.

```
template<class K, class E>
class hashChains
{public:
    ……
    pairNode<const K,E> * find(const K& theKey) const;
    //返回关键字theKey匹配的数对的指针，若不存在匹配的数对，则返回NULL
     {return table[hash (& theKey)% divisor].find(theKey);}
    void insert(const pair<const K,E>& thePair);
    //在字典中插入一个数对thePair,若存在关键字相同的数对，则覆盖
    void erase(const K& theKey);
    //删除关键字theKey匹配的数对
    {table[hash (& theKey)% divisor].erase(theKey);}
Protected:
    sortedChain<K,E> *table; // 链表数组
    int divisor; // 位置数
    hash<K> hash;//把类型k映射到一个非负整数
    int dSize;  //数对的个数
}
```

hashChains <K,E>::insert

```
void hashChains <K,E>::insert(const pair<const K,E>& thePair);
{//插入一个数对thePair，若存在关键字相同的数对,则覆盖
    int homeBucket = (int) hash (thePair.first) )% divisor;
    int homeSize = table[homeBucket].size();
    table[homeBucket].insert(thePair);
    if ( table[homeBucket].size() > homeSize )
      dSize++;
}
```

一种改进的方法



∞表示大的关键字

与线性探查比较

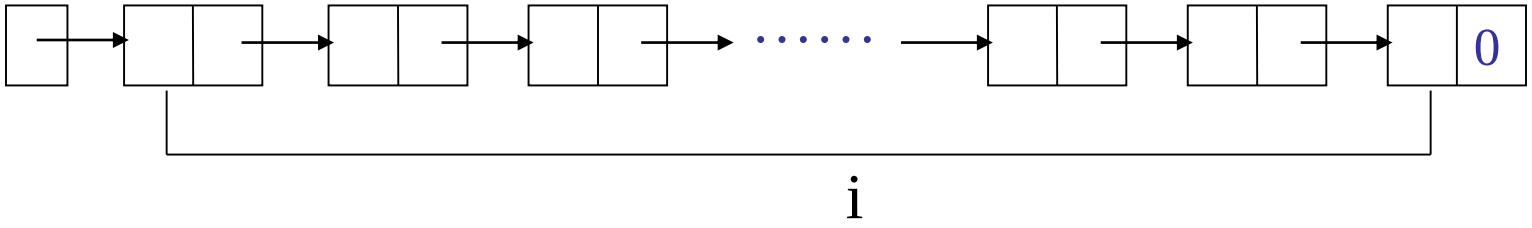
- 空间:
  - 线性探查 :2b+sn　(数组 empty: 2b)
  - 链式散列 :　2b(b个firstNode指针)+2b(b个dSize)+2n(n个next)+ns
  - 线性探查空间少

与线性探查比较

- 时间复杂性:
- 线性探查 :
  - 最坏情况: $\Theta(n)$
  - 平均性能:
    - $S_n \approx \frac{1}{2}(1 + 1/(1-\alpha))$
    - $U_n \approx \frac{1}{2}(1+1/(1-\alpha)^2)$
- 链式散列 :
  - 最坏情况: O(n)
  - 平均性能:

与线性开型寻址散列比较

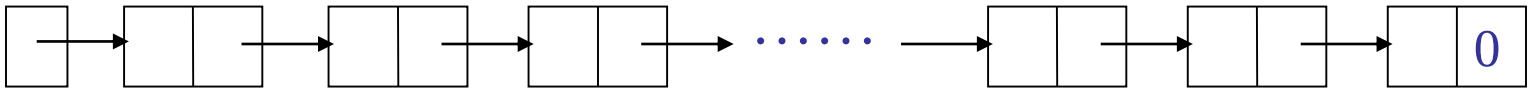
平均性能U_n：



- 一次不成功搜索所要搜索的节点数为：
  - $\frac{1}{i+1}(i+\sum_{j=1}^{i} j) = \frac{i\,(i+3)}{2(i+1)}$　　i>=1
  - 0　　i=0
- 链表的平均长度为: n/b= α
  - $U_n \sim (\alpha+1)/2$　　α >=1
  - $U_n <= \alpha$　　α <1

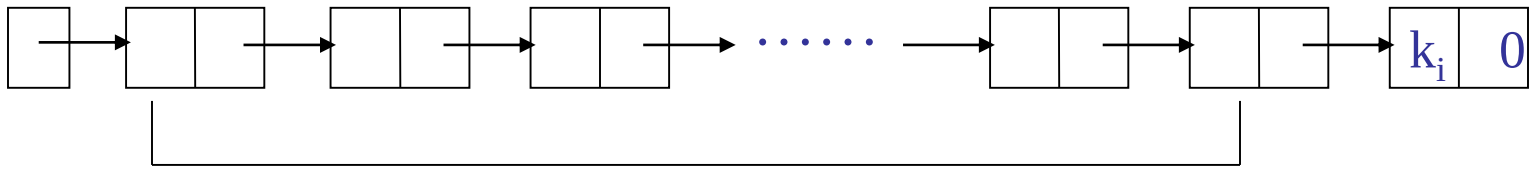
与线性开型寻址散列比较

平均性能S_n：



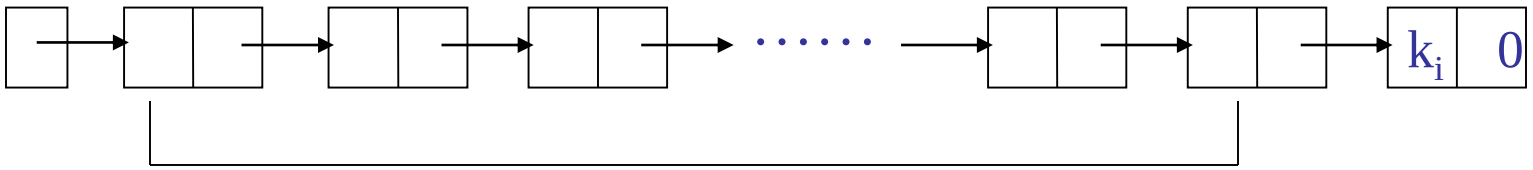
1次　　2次　　3次　　i次

- 假定 :各元素是按升序插入的。
- 当插入第*i* 个元素时



(i-1)/b

与线性开型寻址散列比较



- Search k_i: (i-1)/b+1

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

$$\sim 1+\alpha/2$$

- 使用链表时的平均性能要优于使用线性开型寻址。

与跳表比较

- 跳表和散列均使用了随机过程来提高字典操作的性能
- 时间复杂性:
  - 最坏情况:
    - 跳表: $\Theta(n+MaxLevel)$
    - 链表散列: $\Theta(n)$
  - 平均性能:
    - 跳表: $O(\log n)$
    - 链表散列: $O(1)$
- 空间 （指针所占用的空间 ）:
  - 跳表:
    - 平均: MaxLevel+n/(1-p)
    - 最坏情况: n*MaxLevel+1
  - 链表散列: D+n

与跳表比较

- 跳表比散列表更灵活.
  - 在线性时间内按升序输出所有的元素.而采用链表散列时，需要$(D+n)$ 时间去收集n 个元素，并且需要$O(n\log n)$时间进行排序，之后才能输出 。
  - 查找或删除最大或最小元素，散列可能要花费更多的时间（仅考虑平均复杂性）。

10.6 应用—文本压缩

- LZW（ Lempel、Ziv、Welch）压缩
- 压缩方法:
  - 基于原始数据，创建一个字典，字典中所存放的是文本中的字符串与其编码的映射。然后，用字典中的编码来代替原始数据中的相应字符串。
- 例:要压缩的文本文件为：
  - aaabbbbbbaabaaba
- LZW压缩字典:

code	0	1	2	3	4	5	6	7
key	a	b	aa	aab	bb	bbb	bbba	aaba

10.6 应用—文本压缩

- LZW（ Lempel、Ziv、Welch）压缩
- 压缩方法:
  - 基于原始数据，创建一个字典，字典中所存放的是文本中的字符串与其编码的映射。然后，用字典中的编码来代替原始数据中的相应字符串。
- 例:要压缩的文本文件为：
  - aaabbbbbbaabaaba
- LZW压缩字典:

code	0	1	2	3	4	5	6	7
key	a	b	aa	aab	bb	bbb	bbba	aaba

10.6 应用—文本压缩

- LZW规则：
  - 开始，为该文本文件中所有可能出现的字母分配一个代码，构成初始字典。
  - LZW压缩器不断地在输入文件的未编码部分中寻找在字典中出现的**最长的前缀p**，**输出**前缀p相应的代码，若输入文件中的下一个字符为c，则为**pc分配一个代码**,并**插入字典**。

LZW方法示例

- 用LZW方法来压缩字符串“aaabbbbbbaabaaba”。

code	0	1	2	3	4	5	6	7
key	a	b	aa	aab	bb	bbb	bbba	aaba

10.6 应用—文本压缩

- LZW压缩实现:
- 设当前前缀p为空，读取下一个字符c;

```
循环（当前字符串pc不为空）
 { if (当前字符串pc在字典中)
        当前前缀p=当前字符串pc;
     else { 将当前字符串pc插入到字典中;
            输出当前前缀p的编码;
            p=c;
          }
   读取下一个字符c;
 }
输出最后一个当前前缀p的编码;
```

LZW解压缩方法

1. 把分配给单一字母的代码插入字典中。
2. 输入第一个代码，用相应的文本（第一个代码一定对应于一个单一的字母）代替。
3. 设当前输入代码为p，q为p前面的代码。
   两种情况：1)在字典中；2)不在字典中。
   - 1)当p在字典中时，
     - 找到与p相关的文本text(p)并输出。
     - text(q)fc(p)插入字典
   - 2)当p不在字典中，此时只会是一种情况，
     - text(p)=text(q)fc(q)，　(p,text(p))插入字典
     - 代码串:qp　对应文本串:text(q)text(q)fc(q)

LZW解压缩实现：

1. 把分配给单一字母的代码插入字典中。
2. 输入第一个代码q，输出相应的文本（第一个代码一定对应于一个单一的字母）。
3. 循环：

   ①输入下一个代码p;

   ②If （p在字典中）

   {输出代码p对应的文本串text(p);

   将text(q)fc(p)及代码插入到字典中}

   Else{【此时只会是一种情况：

   text(p)=text(q)fc(q);

   即代码串qp对应文本串text(q)text(q)fc(q)】

   输出代码p对应的文本串text(q)fc(q);

   将text(q)fc(q)及代码插入到字典中}

   ③q=p;