第9章 查找

Search

主讲：顾为兵

第9章 查找/检索 (Search)

目录

查找概述

- 查找表(Search Table)：
  同类型数据元素(又称记录）的集合
- 数据元素/记录：

	关键字	其他项
ElemType	key	other

- 查找操作：
  给定一个值K，在查找表中找出键值等于K的记录

教材中几个带参宏定义作为比较函数：

对数值型关键字：

```
#define EQ(a,b)  ((a)==(b))
#define LT(a,b)  ((a)<(b))
#define LQ(a,b)  ((a)<=(b))
```

对字符串型关键字：

```
#define EQ(a, b)  (!strcmp((a),(b)))
#define LT(a, b)  (strcmp((a),(b))<0)
#define LQ(a, b)  (strcmp((a), (b))<= 0)
```


查找表的存储结构：

线性结构：顺序表，链表

树形结构：二叉排序树，B_树等

散列结构：哈希表（散列表）

查找性能的评价指标：

平均查找长度ASL： 在查找过程中，给定值K与关键字比较次数的期望值

$$ASL = \sum_{i=1}^{n} Ci \times Pi$$

n ---查找表中记录个数

Ci----查找到第i条记录时，K与关键字的比较次数

Pi----查找第i条记录的概率

- 查找表的基本运算：

创建表：	Create(&ST,n)	静态查找表
销毁表：	Destroy(&ST)	静态查找表
查找：	Search(ST,K)	静态查找表
读表元：	Get(ST,pos)	静态查找表
插入：	Insert(&ST,e)	动态查找表
删除：	Delete(&ST,K)	动态查找表

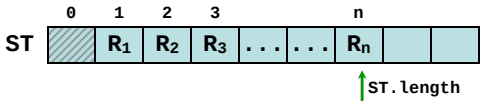
- 静态查找表：查找表被创建后，一般不做插入和删除的修改，即没有为其定义插入和删除操作；
- 动态查找表：既查找，又修改

§9.1 静态查找表

9.1.1 顺序查找

假定，查找表采用顺序存储结构：

```
typedef struct{
   ElemType   *elem;      //数组的基地址
   int        length;     //表中记录的个数
}SSTable;
```

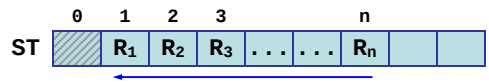


return

顺序查找思想：

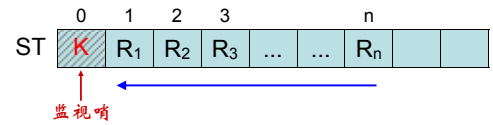
从表的一端开始，逐个将给定值K与关键字进行比较，直到找到记录或查找失败；

注意：这里“顺序”的含义不是查找表是顺序存储的，而是顺着表的自然次序逐个比较。




```
int Search(SSTable ST, KeyType K) {
 //从表尾开始，顺序查找，  查找成功返回元素的位置下标
 //查找失败，返回0
  int i=ST.length;
  while(i>0 && ST.elem[i].key!=K) i--;
  return  i ;
}//search
```

检测下标 i 是否越界




```
int Search(SSTable ST, KeyType K) {
   //带哨兵的检索，从表尾开始，顺序查找
   //查找成功返回元素的位置下标；查找失败，返回0
   int i=ST.length; ST.elem[0].key=K;
   while(ST.elem[i].key!=K) i--;
   return i;
}
```

静态查找表 • 顺序查找

查找效率：

- 查找成功时的ASL：

$$ASL=\sum_{i=1}^{n}Ci\times Pi=\sum_{i=1}^{n}(n-i+1)\times\frac{1}{n}=\frac{n+1}{2}$$

- 查找失败时的ASL： ASL=n+1
- 平均：ASL＝3(n+1)/4

静态查找表 • 顺序查找

顺序查找小结：

- 最朴素的查找方法
- 对表的限制最少
- 不仅适用于顺序存储结构，而且适用于链式存储结构
- 时间性能最差，O(n)
- 等概情况下查找成功时ASL＝(n+1)/2

查找

§9.1 静态查找表

静态查找表 • 折半查找

9.1.2 折半查找(二分查找)

这种查找方法对表有严格的限制：

1. 顺序存储；
2. 按关键字值有序排列

静态查找表 • 折半查找

折半查找方法：

设查找区间为R[low ..high]，

取mid = ⌊(low+high)/2⌋，则

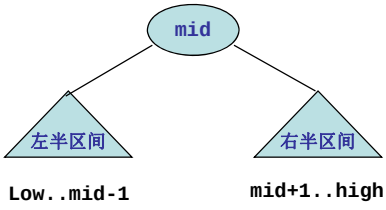
- 当K<R[mid].key，
  下一个查找区间为R[low..mid-1](左半区间)
- 当K>R[mid].key，
  下一个查找区间为R[mid+1..high](左半区间)
- 当K==R[mid].key，
  查找成功，结束
- 当low>high，
  查找失败，结束

静态查找表 • 折半查找

```
int Search_Bin(SSTable ST, KeyType K){
  int low=1, high=ST.length;
  int mid;
  while(low<=high){
     mid=(low+high)/2;
     if(K<ST[mid].key)
          high=mid-1;      //在左区间继续查找
     else if(K>ST[mid].key)
          low=mid+1;       //在右区间继续查找
     else  return mid;     //查找成功的出口
   }//while
   return 0;       //查找失败的出口
}//Search_Bin
```

静态查找表 • 折半查找

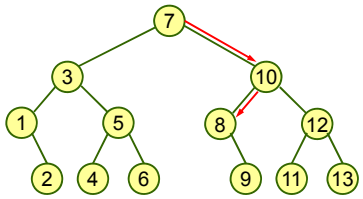
折半查找判定树：



静态查找表 • 折半查找

查找K＝39：

0	1	2	3	4	5	6	7	8	9	10	11	12	13
	02	11	16	22	25	27	33	39	42	56	77	79	81

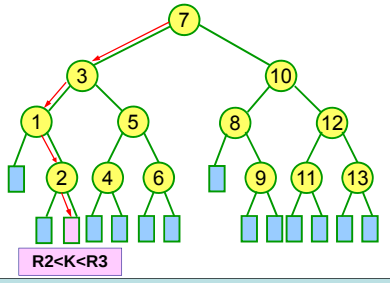


查找成功的过程是自树根沿树枝到达目标结点，K与关键字的最大比较次数不超过树高$\lfloor \log_2 n \rfloor+1$

静态查找表 • 折半查找

查找K＝13：

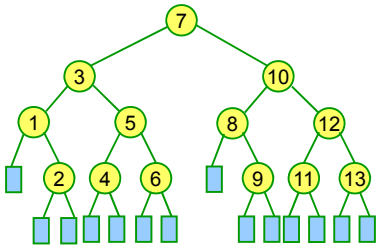
0	1	2	3	4	5	6	7	8	9	10	11	12	13
	02	11	16	22	25	27	33	39	42	56	77	79	81



查找失败的过程是从树根开始，沿树枝到达一个外部结点，K与关键字的最大比较次数也不超过树高

静态查找表 • 折半查找

n=13



查找成功：ASL=(1×1+2×2+3×4+4×6) /13=41/13

查找失败：ASL=(3×2 + 4×12) / 14 = 54/14

静态查找表 • 折半查找

折半查找的ASL

设: $n=2^h-1$----折半查找判定树是高为h的满二叉树

$h=\log_2(n+1)$

$$ASL=\sum_{i=1}^{n} Ci\times Pi=\frac{1}{n}\sum_{j=1}^{h} j\times 2^{j-1}$$

$$设: S=\sum_{j=1}^{h} j\times 2^{j-1}=1\times 2^0+2\times 2^1+\cdots+h\times 2^{h-1}$$

静态查找表 • 折半查找

$$S=2S-S$$
$$=(1\times 2^1+2\times 2^2+\cdots+h\times 2^h)-(1\times 2^0+2\times 2^1+\cdots+h\times 2^{h-1})$$
$$=h\times 2^h+(2^0+2^1+\cdots+2^{h-1})$$
$$=h\times 2^h+(2^h-1)$$
$$=(n+1)\log_2(n+1)+n$$

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

折半查找小结

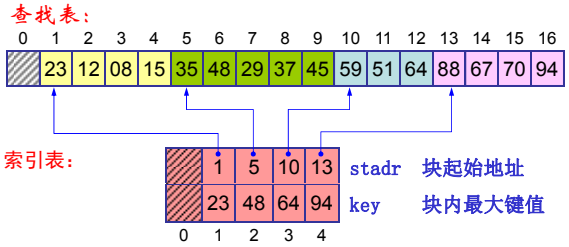
- 一种效率很高的查找方法，时间复杂度O($\log_2 n$)；
- 对表有严格的限制：有序的顺序表；
- 折半查找判定树是分析查找性能的有效工具，查找每个结点所需的比较次数等于该结点在树上的层次数；
- 折半查找判定树是一棵理想平衡二叉树，树的高度为$\lfloor \log_2 n \rfloor + 1$；
- 无论查找成功或失败，与关键字的最大比较次数都不会超过树的高度。

§9.1 静态查找表

9.1.4 索引顺序查找（分块查找）

- 查找性能介于顺序查找和折半查找之间；
- 对表的限制也是介于顺序查找和折半查找之间；
- 查找表的特点：分块有序，块内无序，块间有序。第i块的最大关键字小于第i＋1块的最小关键字；
- 除了查找表外，还需要建一个索引表，查找分两级进行。

查找表：



- 索引表是有序表
- 在索引表中查块号，采用顺序查找或折半查找均可
- 在查找表中找目标记录，只能采用顺序查找
- ASL＝索引表的ASL＋查找表的ASL

索引表的类型定义：

```
typedef struct{
     KeyType   key;       //块内最大键值
     int       stadr;     //块起始地址
}IndexItem;

typedef struct{
     IndexItem *elem;     //索引表基地址
     int       length;    //索引表长度
}IndexTable;
```


设：将n个元素均匀地分为b块，

前b-1块每块有s个元素，s=⌈n / b⌉

第b块有n-s×(b-1)个元素

对索引表采用用顺序查找：

$$ASL = \frac{b+1}{2} + \frac{s+1}{2} = \frac{s^2+2s+n}{2s}$$

当$s=\sqrt{n}$时，ASL取得极小值 $\sqrt{n}+1$

对索引表采用用折半查找查找：

$$ASL = [\frac{b+1}{b}\log_2(b+1)-1] + \frac{s+1}{2} \approx \log_2(\frac{n}{s}+1) + \frac{s}{2}$$

第九章 查找/检索 (Search)

目录

§ 9.2 动态查找表----二叉排序树

9.2.1二叉排序树的定义

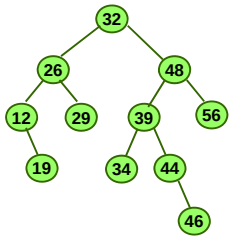
Binery Sort Tree定义(P141)

二叉排序树或者是一棵空树；或者是具有下列性质的二叉树：

(1) 若它的左子树不空，则左子树上所有结点的键值均小于根结点的键值；

(2) 若它的右子树不空，则右子树上所有结点的键值均大于根结点的键值；

(3) 它的左、右子树也分别是二叉排序树。

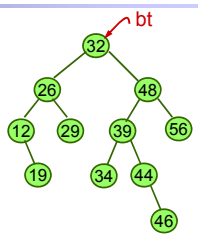


中序序列：12，19，26，29，32，34，39，44，46，48，56

二叉排序树的一个重要性质：中序序列是一个有序序列

§ 9.2 动态查找表----二叉排序树

9.2.2 二叉排序树的查找



给定值K，

当 K < bt->key: 在bt的左子树上继续查找

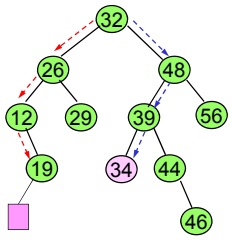
当 K > bt->key: 在bt的右子树上继续查找

当 K==bt->key: 查找成功

举例：

K＝34

K＝18



查找成功：自树根沿树枝到达目标结点

查找失败：自树根沿树枝到达一个外部结点

成功或失败的最大比较次数都不超过树高

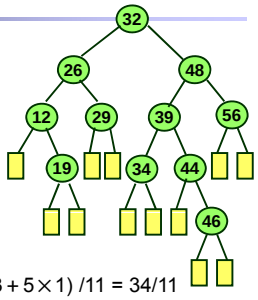
查找的递归算法：

查找成功，返回目标记录的指针；查找失败，返回空指针

```
BiTree  BSTSearch(BiTree bt, KeyType K) {
  //在根为bt的二叉排序树上查找键值等于K的记录
  if ( !bt || K==bt->key ) return bt;   //查找成功或失败而结束
  else if  (K < bt->key)
       return  BSTSearch (bt->lchild, K);  //在左子树上继续找
  else
       return BSTSearch (bt->rchild, K)  //在右子树上继续找
}
```


查找的非递归算法

```
Status BSTSearch(BiTree bt, KeyType K,
                 BiTree &p, BiTree &f){
  //在根为bt的二叉排序树上查找键值等于K的记录
  //查找成功,用指针p指向目标;f指向目标的双亲,f初值为NULL
  p=bt;
  while(p){
     if(K<p->key){f=p;p=p->lchild;} //左子树上继续找
      else if(K>p->key)
              {f=p; p=p->rchild;} //右子树上继续找
               else return TRUE; //查找成功
  }//end while
  return  FALSE;     //查找失败
}
```

查找分析：

查找成功时：

ASL = (1×1＋2×2＋3×4＋4×3＋5×1) /11 = 34/11

查找失败时：

ASL = (3×5＋4×5＋5×2) /12 = 45/12

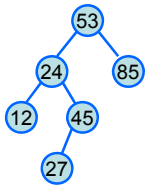
ASL值与树的形态有关

§9.2 动态查找表----二叉排序树

9.2.3 二叉排序树的插入与生成

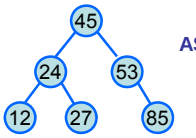
新结点作为一个叶子插到二叉排序树上

关键字序列： 53，24，45，85，27，12



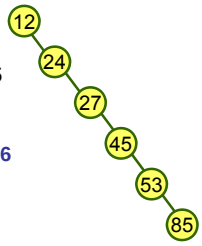
ASL=(1×1＋2×2＋3×2＋4×1) / 6 = 15/6

关键字序列 45，24，27，53，12 ，85



ASL=(1×1+2×2+3×3) / 6 = 14/6

关键字序列 12，24，27，45，53，85



ASL=(1+2+3+4+5+6) / 6 = 21/6

在二叉排序树上插入指针S所指的新结点递归算法：

```
void InsertBST ( BiTree  &bt,  BiTree S ) {
  //在根为bt的二叉排序树上插入指针S所指的新结点
  //新结点是二叉排序树上的一个新叶子
  if(!bt)bt=S;            //bt为空时，S成为树根
  else if(S->key<bt->key)
     InsertBST(bt->lchild, S);   //插在左子树上
  else if(S->key>bt->key)
     InsertBST(bt->rchild, S);   //插在右子树上
  else return; //键值已存在，不插
}
```


从键盘输入记录序列， 生成二叉排序树

```
void CreateBST (BiTree &bt){
  bt=NULL;;
  scanf(&R);
  while(R.key!=endmark){
      f=NULL;
      if(!BSTSearch(bt,R.key,p,f)){ //未找到R.key
         S=(BiTree)malloc(sizeof(BiNode));  //申请结点
         S->key=R.key; S->other=R->other;  //填写记录
         S->lchild=S->rchild=NULL; //填树叶的左、右空指针
         if(f==NULL)bt=S;   //输入的第一个成为树根
         else if(S->key<f->key)f->lchild=S;
              else f->rchild=S;}
      else printf("该结点已经存在!\n");
     scanf( &R);
    }//end while
}
```


§9.2 动态查找表----二叉排序树

9.2.4二叉排序树的删除操作

在下面的讨论中，设：

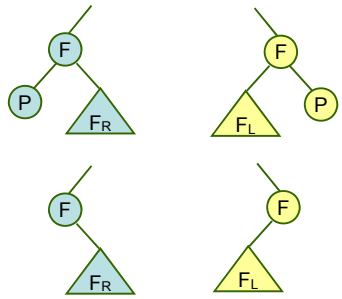
P是待删结点

F是P的双亲

S是P的中序前驱

Q是S的双亲

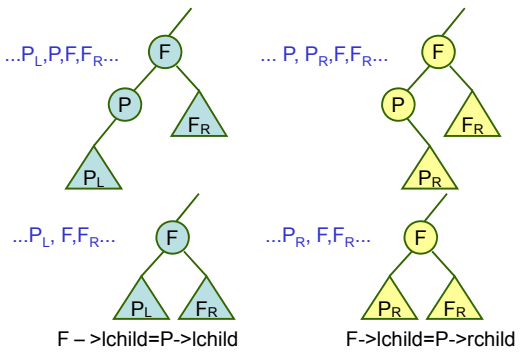
1. P的度＝0： 直接删



F->lchild=NULL　　F->rchild=NULL

动态查找表 • 二叉排序树 • 删除操作

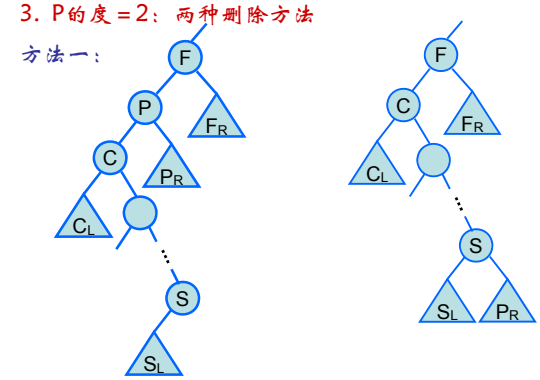
2. P的度＝1 ：将P的唯一的孩子交给双亲



动态查找表 • 二叉排序树 • 删除操作

3. P的度＝2：两种删除方法

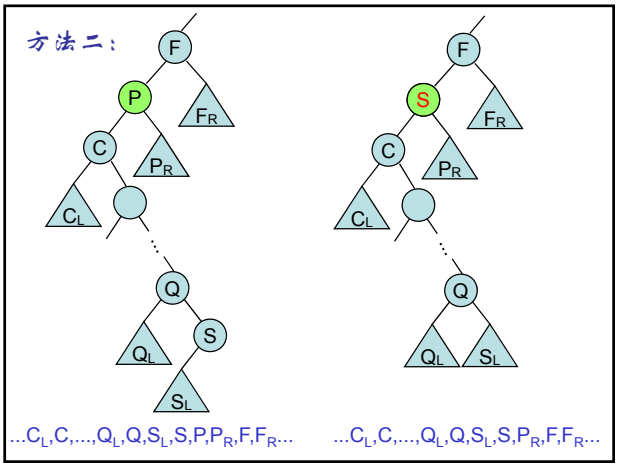
方法一：



动态查找表 • 二叉排序树 • 删除操作

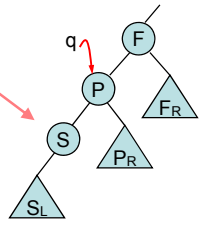
```
s=p->lchild;
while(s->rchild)   s=s->rchild ;   //找P的中序前驱S
s->rchild=p->rchild;                //P的右孩子交给S
F->lchild=p->lchild;                // P的左孩子交给F
free(p);
```

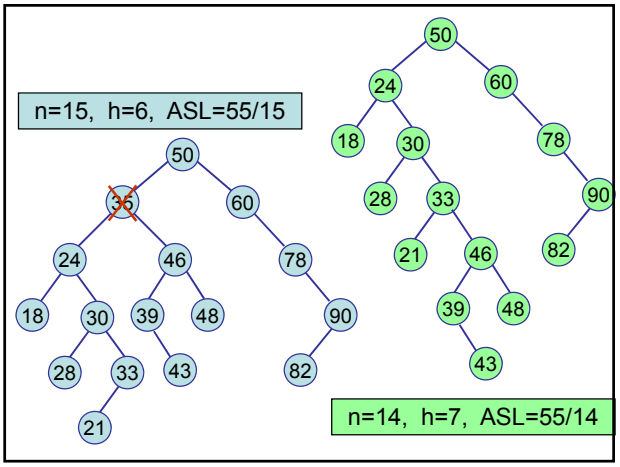
方法二：

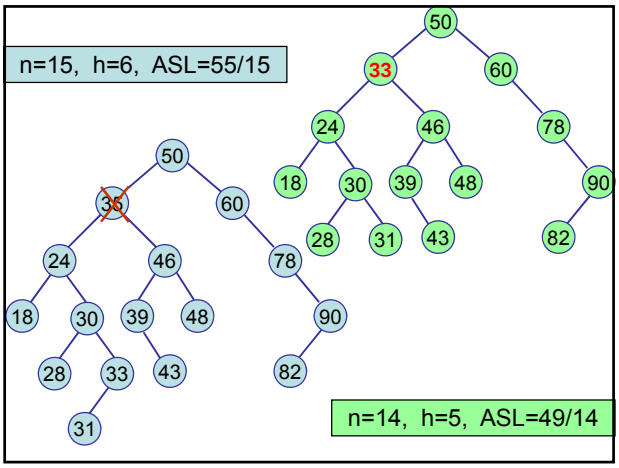


动态查找表 • 二叉排序树 • 删除操作

```
q=p; s=p->lchild;
whild(s->rchild)  { q=s; s=s->rchild ; }   //找P的中序前驱S
p->key=s->key;  p->other=s->other;
if (p != q)  q->rchild=s->lchild          //S的左孩子交给Q
else   q->lchild=s->lchild
free(s)
```







第九章 查找/检索 (Search)

目录

§ 9.3 散列表（哈希表）

9.3.1哈希表的概念(hash)

- 一种基于计算的存储技术和查找技术：

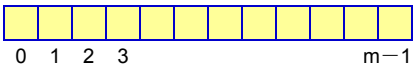
  `Address(Ri)=H(Ri.key)`

  其中，函数H( )称为哈希函数；

- 建立查找表时，每个元素按哈希函数算出的地址`H(Ri.key)`去存放；
- 检索时，按`H(K)`算出地址，找目标元素 。

哈希表的存储结构:

用一维数组`HT[m]`存放n个元素：



n：表长

m：哈希表的容量(m>n)，

哈希地址空间：`0..m-1`

$\alpha=n/m < 1$：装填因子，一般取为`0.65~0.9`

装填因子反映了哈希表的装满程度，是影响哈希表查找性能的重要参数 。

哈希表需要解决的两个问题：

1. 哈希函数H( )的构造：

   要求： (1) 值域是(0, m-1)
   (2) 计算简单
   (3) 函数值尽量分布均匀，以减少同义词

2. 解决冲突问题：

   冲突：$H(K_1)==H(K_2)$， 且$K_1 \neq K_2$

   不同的记录争夺同一个哈希地址；

   K_1与K_2称为同义词

§9.3 散列表（哈希表）

9.3.2 哈希函数的构造

1. 直接定址法
2. 数字分析法
3. 平方取中法
4. 折叠法
5. 除留余数法
6. 随机数法

一、直接定址法

用关键字的线性函数作为哈希函数：

H(k)= a×k+b　　a,b----常数

举例：学生档案，地址空间：000..999

H(k)=k－98015001

哈希地址	000	001	002	…	…
学号	98015001	98015002	99015003	…	…
姓名	王平	李凯	田爱玲	…	…
…	…	…	…		

特点：一般不会发生冲突，但适用面比较窄

二、数字选择法

- 可能出现的关键字都是已知的。
- 将关键字的若干位提取出来，组合成哈希地址。

			2		3	1		哈希地址
8	1	3	4	6	5	3	2	3 4 5
8	1	3	7	2	2	4	2	4 7 2
8	1	3	8	7	4	2	2	2 8 4
8	1	3	0	1	3	6	7	6 0 3
8	1	3	2	2	8	1	7	1 2 8
8	1	3	3	8	9	6	7	6 3 9
8	1	3	5	4	1	5	7	5 5 1
8	1	3	6	8	5	3	7	3 6 5
8	1	4	1	9	3	5	5	5 1 3
×	×	×	✓	✓	✓	✓	×	

三、平方取中法

- 对数字选择法的改进
- 将关键字平方，取中间的几位作为哈希地址。这样，中间几位与关键字的每一位都相关。

关键字	(关键字)²	哈希地址
0100	0010000	100
0110	0012100	121
1010	1020100	201
1001	1002001	020
0111	0012321	123

四、折叠法

- 适合于长关键字

K=582423241169

位移叠加：

582
423
241
+ 69
1315

H(k)=315

间界叠加：

582
324
241
+ 96
1243

H(k)=243

五、除留余数法

- 最简单、最常用的一种方法；

$$H(K) = K \bmod p \quad (p \leqslant m)$$

- 除数p的选择不当将会导致很多同义词出现；
- 当m较小时，取p为小于m的最大质数；
- 当m较大时，取p为质因子不小于20的合数 。

§9.3 散列表（哈希表）

9.3.2 处理冲突的方法

一、开放定址法

“开放”----哈希地址为h的单元不仅向哈希函数值等于h的同义词开放，而且向哈希函数值不等于h的记录开放。以“抢占”的方式争取哈希地址。

探测序列：

$$Hi=(H(key)+di) \bmod m \quad i=1,2,...,k$$

线性探测法：$di=1, 2, 3, ..., k$

二次探测法：$di=1^2, -1^2, 2^2, -2^2, ...$

举例，已知关键字序列：

26，36，41，38，44，15，68，12，06，51，25

用除留余数法构造哈希函数，线性探测法解决冲突。

①建哈希表；②求查找成功和失败的ASL。

解： n=11

取α = 0.75， $m = \lceil n/\alpha \rceil = 15$

哈希函数：H(key) = key mod 13

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

构造哈希表：

key	26	36	41	38	44	15	68	12	06	51	25
H(key)=key mod 13	0	10	2	12	5	2	3	12	6	12	12

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
HT	26	25	41	15	68	44	06	Ø	Ø	Ø	36	Ø	38	12	51

哈希表的查找方法：

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
HT	26	25	41	15	68	44	06	Ø	Ø	Ø	36	Ø	38	12	51

当K＝06， H(K)=6，与HT[6]进行 1 次比较查找成功。

当K＝51，H(K)=12，与HT[12]，HT[13]，HT[14]共进行3次比较查找成功。

当K＝20， H(K)=7，与HT[7]进行 1 次比较查找失败。

当K＝27， H(K)=1，与HT[1]、HT[2]、HT[3]、HT[4]、HT[5]，HT[6]、HT[7]共进行7次比较查找失败。

哈希表的查找性能分析：

查找成功时：

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
HT	26	25	41	15	68	44	06	Ø	Ø	Ø	36	Ø	38	12	51

Key=	26	36	41	38	44	15	68	12	06	51	25
H(key)=	0	10	2	12	5	2	3	12	6	12	12
比较次数：	1	1	1	1	1	2	2	2	1	3	5

等概下，查找成功时，

ASL = (1+1 + 1 + 1 + 1 + 2 + 2 + 2 + 1 + 3 + 5) /11 = **20 / 11**

哈希表的查找性能分析：

查找失败时：

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
HT	26	25	41	15	68	44	06	Ø	Ø	Ø	36	Ø	38	12	51

H(Key)=	0	1	2	3	4	5	6	7	8	9	10	11	12
比较次数：	8	7	6	5	4	3	2	1	1	1	2	1	11

等概下，查找失败时，

ASL = (8+7 + 6 + 5 + 4 + 3 + 2 + 1 + 1 + 1 + 2 + 1 + 11) /13 = 4

对哈希表做删除操作时应注意：

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
HT	26	25	41	15	68	44	06	Ø	Ø	Ø	36	Ø	38	Ø	51
														del	

注意：在哈希表中删除一个结点，正确的做法是用一个特殊标记的结点取代被删结点

解决冲突的方法：

一、 开放定址法

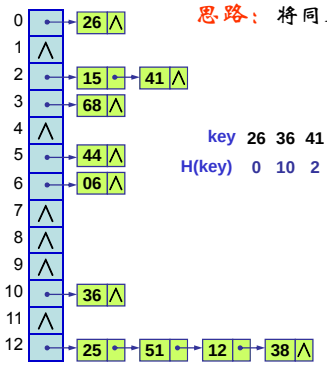
二、 拉链法

三、 再哈希法

四、 建公共溢出区法

二、 拉链法

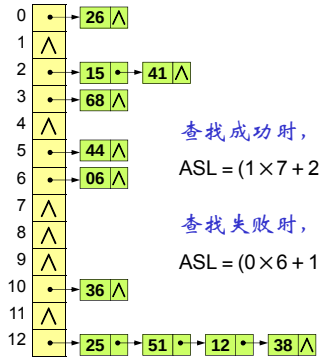
思路： 将同义词链接成单链表。



key	26	36	41	38	44	15	68	12	06	51	25
H(key)	0	10	2	12	5	2	3	12	6	12	12

哈希表 • 处理冲突的方法

拉链法的查找效率：



查找成功时，

ASL = (1×7 + 2×2 + 3 + 4) / 11 = 18 / 11

查找失败时，

ASL = (0×6 + 1×5 + 2 + 4) / 13 = 11 / 13