第六章 集合与字典

从逻辑结构上看，集合和字典都是最简单的数据结构，它们的元素之间没有任何确定的依赖关系。

- 集合具有简单结构，对元素没有任何要求
- 字典是关联（二元组）的集合

作为抽象数据类型，集合和字典之间的主要区别，在于它们的操作。

- 集合主要考虑集合之间的并、交和差操作（集合操作）
- 字典主要关心其元素的检索、插入和删除

6.1 集合及其抽象数据类型

6.1.1 基本概念

集合是数学中最基本的概念，也是一种基本数据结构

在数学中，集合是一些互不相同元素的无序汇集。这些元素称为该集合的成员。集合中的成员可以是一个原子（不可再分解）；也可以具有复杂的结构，甚至可以又是集合

我们关心的问题就是一个元素是否属于某个集合：

$$x \in A, \quad x \notin A$$

集合表示法：

列举法，用于描述有穷集，将成员放在一对花括号中，成员之间用逗号隔开。 例如｛1，2，4｝

谓词描述法，例如 $I = \{x | Z(x) \wedge x \geq 0\}$

相关概念：

集合的大小，即集合中所包含的元素的个数

空集 子集 超集

数据结构中讨论的集合，一般有以下限制：

1）只讨论有穷集

2）假定集合中所有元素属同一类型；并且假设元素之间存在一个小于关系“<”，也称为有序集。

6.1.2 主要运算

可定义测试一个元素是否存在于集合中、增加元素、删除元素等运算。但集合更加关心下面运算：

求并集： $A \cup B = \{x | x \in A \vee x \in B\}$

求交集： $A \cap B = \{x | x \in A \wedge x \in B\}$

求差集： $A - B = \{x | x \in A \wedge x \notin B\}$

子集： A是B的子集 $\forall x(x \in A \Rightarrow x \in B)$

如果集合A是B的子集，反过来也称集合B是A的超集。

相等： $A = B \Leftrightarrow A \subseteq B \wedge B \subseteq A$

例：若A＝｛a，b，c｝，B＝｛b，d｝，则有

A∪B＝｛a，b，c，d｝
A∩B＝｛b｝
A－B＝｛a，c｝

另外，A不等于B，同时A和B相互都不是子集关系。

集合间运算都可通过增加、删除元素和成员测试等实现，如：

- 已知集合A和B，求它们的并集，只要以集合A（或B）为基础，把集合B（或A）的元素逐个插入
- 要求两集合的交集，只要从A（或B）出发，检查各元素是否也在B（或A）中出现，也在另一个集合里出现的元素插入初态为空集的集合中即可
- 求A与B的差集A-B时，只要以A为基础，对每个B中的元素做删除运算即可

6.1.3 抽象数据类型

ADT Set is

operations

Set createEmptySet (void)

创建一个空集合。

int member（Set A，DataType x ）

当 x ∈ A时返回真值，否则取假值。

int insert（Set A，DataType x ）

使 x 成为 A 的一个成员,如果 x 本来就是A 的成员，则 A 不变

int delete（Set A，DataType x ）

将 x 从A 中删除，如果 x 本来就不在A 中，则不改变 A

int union（Set A，Set B，Set C ）

求集合A和B的并集，结果在集合C中

int intersection （Set A，Set B，Set C ）

求集合A和B的交集，结果在集合C中

int difference（Set A，Set B，Set C ）

求集合A和B的差集，结果在集合C中

int subset（Set A，Set B ）

当且仅当A是B的子集时，函数值才为真

end ADT Set

6.2 集合的实现

- 位向量表示
- 单链表表示

6.2.1 位向量表示

位向量是一种每个元素都是二进制位（即0/1值）的数组

当表示的集合存在不太大的公共超集时，采用位向量的方式表示这种集合往往十分有效

存储结构

假设要表示的集合的公共超集共有 n 个不同元素，为叙述方便，不妨假设这些元素就是整数0， 1， 2，…， n-1

每个集合可以采用一个有 n 位的位向量来表示。若整数 i 是这个集合的一个元素，则位向量中对应的位取 1（真），否则取0（假）

由于在C语言中无法直接定义位数组，为定义位向量需要借助其它方式。一种自然方式：是用长n/8的字符数组表示长n的位向量。一个字符有8位二进制编码，可表示8个二进制位

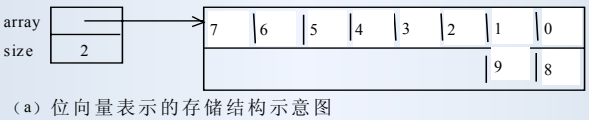
集合的位向量表示的存储结构：

```
typedef struct {
   int size;          /*字符数组的长度*/
   char * array; /*位向量空间。每一数组元素保存8位*/
} BitSet;
```

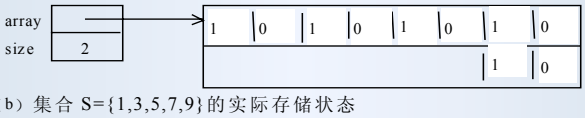

公共超集是0到9的整数集的位向量表示的存储结构示意图。

可以看出每个整数所对应的二进制位的位置

用位向量表示集合，所占空间与公共超集的大小 n 成正比，与要表示的集合的大小无关。



（a）位向量表示的存储结构示意图



（b）集合 S={1,3,5,7,9}的实际存储状态

算法实现

以位向量表示集合，只有两个集合都是某公共集合的子集时，互相运算才有效。

由于位向量里并没有关于集合元素的实际信息，只能要求参与运算的两个位向量长度相同。

C语言位运算：设 x 和 y 都是8位的字符，其值分别是：

x = 01010111

y = 11011010

对 x 和 y 做各种字位运算，得到的结果如下：

~x 10101000

x & y 01010010

x ^ y 10001101

x | y 11011111

x << 3 10111000

y >> 5 00000110

空集合的创建

与创建空顺序表类似，但这里需要给每个字节赋值0

```
BitSet * createEmptySet (int n) {   /*创建n位的位向量*/
   int i;    BitSet * s = (BitSet *)malloc(sizeof(BitSet));
   if (s ==NULL) return NULL;
   s->size = (n + 7) / 8;
   s->array = (char *)malloc(s->size * sizeof(char));
   if (s->array == NULL) {
      free s; return NULL;
   }
   for (i = 0; i < s->size; i++)  s->array[i] = '\0';
   return s;
}
```


将整数 index 对应的元素插入集合

```
int insert (BitSet * s, int index) {
   /*将位向量中下标为index的位置为1*/
   if (index >= 0 && index>>3 < s->size)  {
      s->array[index >> 3] |= (1 << (index & 07));
      return 1;
   }
   return 0;
}
```

index >> 3：取得第 index 位所在的数组元素的下标

1 << (index & 07)：做一个字节，其中一位为1，其余位都为0

将整数index的元素从集合中删除

```
int delete(BitSet * s, int index) {
   /*将位向量中下标为index的位，置为0*/
   if (index >= 0 && index >> 3 < s->size) {
      s->array[index >> 3] &= ~(1 << (index & 07));
      return 1;
   }
   return 0;
}
```


判断整数index的元素是否属于集合

```
int member(BitSet * s, int index) {
   /*检查位向量中下标为index的位置是否为1*/
   if (index >= 0 && index >> 3 < s->size&&
         (s->array[index >> 3] & (1 << (index & 07))))
      return 1;
   return 0;
}
```


集合与集合的并

将 s2 设置为 s0 和 s1 的并集。利用按位“或”运算实现：

```
int union (BitSet * s0, BitSet * s1, BitSet * s2) {
/*位向量等长时置s2为s0与s1的并集且返回1；否则返回0*/
   int i;
   if (s0->size != s1->size || s2->size != s1->size) return 0;
   for (i = 0; i < s1->size; i++)
      s2->array[i] = s0->array[i] | s1->array[i];
   return 1;
}
```

集合与集合的交

将 s2 设置为 s0 和 s1 的交集。通过按位“与”运算实现:

```
int intersection(BitSet * s0, BitSet * s1, BitSet * s2) {
/* 位向量等长时置s2为s0与s1的交集且返回1；否则返回0*/
   int i;
   if (s0->size != s1->size|| s2->size != s1->size) return 0;
   for (i = 0; i < s1->size; i++)
        s2->array[i]=s0->array[i] & s1->array[i];
   return 1;
}
```

19

集合与集合的差

将 s2 设为 s0 对 s1 的差集，只需求出 s0 与 s1 补集的交集

```
int difference(BitSet * s0, BitSet * s1, BitSet * s2) {
/*位向量等长时置s2为s0与s1的差集且返回1；否则返回0*/
   int i;
   if (s0->size != s1->size|| s2->size != s1->size) return 0;
   for (i = 0; i < s1->size; i++)
      s2->array[i]=s0->array[i] & ~s1->array[i];
   return 1;
}
```

20

6.2.2 单链表表示

链接表示方式下，在链表中存放的是集合中元素的实际数值，而不是元素是否属于集合的标记。

- 链表中的每个结点表示集合中的一个元素，具体方式与第二章单链表的结点struct Node类似
- 不同之处在于：线性表的单链表中，link字段表示线性表元素之间的逻辑后继关系，而在这里仅仅是把属于同一集合的所有元素链接成一个整体
- 因为讨论的是有序集，若将集合中的元素按“<”关系排序构造有序链表，给集合的某些运算会带来方便

具体实现就是链表上的操作，见书。不再仔细讨论

21

6.3 字典及其抽象数据类型

考虑的问题是数据的存储和检索（查询），就像在新华字典里字词的组织和查找一个字的相关解释等

- 存储和检索是计算过程中最重要的基本操作
- 本章主要讨论基于关键码的数据存储和检索，需要根据某些线索找出相关的数据
- 支持这种操作的数据结构，通常称为字典或查找表
- 不是介绍一种结构，而是介绍计算中最重要的一种问题的许多不同解决方式，以及各种相关性质
- 将用到前面讨论过的许多结构。包括各种线性结构、树性结构及其各种组合，涉及在这些结构上操作的许多算法

22

抽象模型

设有关键码集合 KEY 和值（或称属性）集合 $VALUE$

关联(Association)是二元组 $(k, v) \in KEY \times VALUE$

字典：以关联 为元素的有穷集合

$$dic \subseteq KEY \times VALUE$$

$$i \neq j \Rightarrow k_i \neq k_j$$

关键码到值的有穷函数： $dic : KEY \rightarrow VALUE$

所有字典的集合： $DIC \subseteq \mathcal{P}(KEY \times VALUE)$

主要字典操作:

- 检索 $search : DIC \times KEY \rightarrow VALUE$
- 插入元素 $insert : DIC \times KEY \times VALUE \rightarrow DIC$
- 删除元素 $delete : DIC \times KEY \rightarrow DIC$

23

最主要也是使用最频繁的操作是检索（也称查找）

检索效率是字典实现中最重要的考虑因素

静态字典：建立后保持不变的字典

动态字典：内容经常动态变动的字典

静态字典的基本操作就是检索。实现中主要考虑检索效率

动态字典除检索外还有插入和删除，实现中还需要考虑:

- 插入删除操作的效率
- 插入删除可能导致字典结构变化，动态变化中能否保证检索效率？（使字典性能不随操作的进行而逐渐恶化）

24

ADT Dictionary is

operations

Dictionary createEmptyDictionary (void)

创建一个空字典

int search(Dictionary dic, KeyType key, Position p)

在 dic 中检索关键码为 key 的关联，取得的位置放入 p

int insert(Dictionary dic, DicElement ele)

在 dic 中插入关联 ele

int delete(Dictionary dic, KeyType key)

在 dic 中删除关键码为 key 的关联

end ADT Dictionary

25

- 存储和检索是计算机信息处理中最基本的工作
- 字典就是实现存储和检索的结构
- 需要存储和检索的信息有许多具体情况，因此要考虑各种不同的字典实现技术。
- 这里的基本问题是空间利用率和操作效率

检索效率的评价标准：检索过程中关键码的平均比较次数，即平均检索长度ASL（average search length），定义为：

$$ASL = \sum_{i=1}^{n} p_i c_i$$

c_i 和 p_i 分别为元素 i 的检索长度和概率。若各元素的检索概率相等，就有 $p_i=1/n$。$ASL = 1/n \sum c_i$。

字典的组织方法很多，如：顺序、散列、二叉树和B树等。

26

6.4 线性表表示

线性表可以保存信息，因此可以作为字典的实现基础

关联在线性表里顺序排列，形成关联的序列

检索就是在线性表里查找关键码合适的元素

数据元素的插入删除都是很平常的线性表操作

- 顺序表表示
- 链表表示（也可以，不讨论了）

27

6.4.1 顺序表表示

把字典元素（关联）保存在顺序表里，可得到了一种实现

有关数据类型定义：

```
typedef  int KeyType;
typedef  int DataType;

typedef struct {
    KeyType key;       /* 字典元素的关键码字段 */
    DataType value;    /* 字典元素的属性字段 */
} DicElement;

typedef struct {
    int n;         /* n≤MAXNUM，字典中元素个数 */
    DicElement element[MAXNUM];
} SeqDic;
```

28

顺序检索算法

```
/*在字典中顺序检索关键码为key的元素*/

int seqSearch(SeqDic *pdic, KeyType key, int *position) {
   int i;
   for(i = 0; i < pdic->n; i++)   /* 从头开始向后扫描 */
      if(pdic->element[i].key == key) {
         *position = i;
         return TRUE;          /* 检索成功 */
      }
   *position = -1;
   return FALSE;                /* 检索失败 */
}
```

失败时由 position 设置一个特殊值。

29

平均检索长度分析

- $ASL = 1*p_1 + 2*p_2 + \ldots + n*p_n$
  $= (1+2+\ldots+n)/n$ ($p_i=1/n$ 时)
  $= (n+1)/2 = O(n)$
- 优点：数据结构和算法简单，元素是否有序均可使用
- 缺点：平均检索长度大，n很大时检索耗时太多

不适合动态变化频繁的字典

删除元素需顺序检索，O(n)，可能大量移动数据

插入元素（关联）时可保存在已有元素之后，O(1) 操作；若要防止出现重复关键码，就需要检查所有元素，O(n)

动态变化中字典的检索效率不变（已经是最低效的）

30

二分法检索算法

如果字典元素之间有关系，就可能利用它加快检索速度

最常见的关系是关键码有序，这时将元素按序排列（从小到大或从大到小）排列，就可以快速检索（二分法）

二分法检索基本过程（假设元素按关键码升序排列）：

1. 初始时，考虑范围是整个字典（顺序表）
2. 取考虑范围中位于中间的元素，用这个元素的关键码与检索关键码比较。如果相等则检索结束；否则
3. 如果检索关键码较大，把检索范围改为位置高的一半；如果检索关键码较小，把检索范围改为低一半
4. 如果范围里已经无元素，检索失败结束，否则回到2


```
/* 在字典中用二分法检索关键码为key的元素 */
int biSearch(SeqDic *pdic, KeyType key, int *position) {
   int low = 0, high = pdic->n-1, mid;
   while(low<=high) {
      mid = (low+high)/2;          /* 当前检索的中间位置 */
      if(pdic->element[mid].key == key) {  /* 检索成功 */
         *position = mid;  return TRUE;
      }
      else if (pdic->element[mid].key > key)
         high = mid-1;             /* 要检索的元素在左半区 */
      else low = mid+1;            /* 要检索的元素在右半区 */
   }
   *position= -1;
   return FALSE;                   /* 检索失败 */
}
```


性能分析示例:

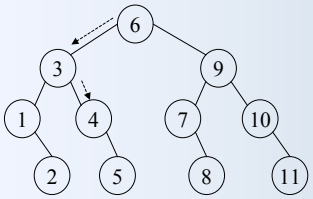
以下面11个数的检索为例:

5	13	19	21	37	56	64	75	80	88	92
1	2	3	4	5	6	7	8	9	10	11

从二分法检索可知：找到第6个数仅需比较1次，找到第3和第9个数需2次，找到第1，4，7，10个数需3次，找到第2，5，8和11个数需4次。

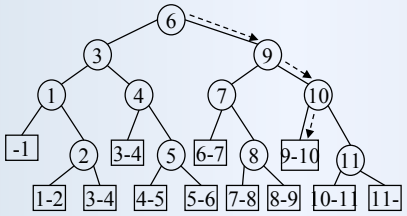
检索过程可用二叉树表示。检索某个结点的比较次数等于该结点的层数加1。

检索结点的过程就是从根结点到相应结点的路径。



检索成功时比较次数不超过树深度。n结点判定树深度至多为$\lfloor \log_2 n \rfloor$。二分法检索成功时的比较次数最多为$\lfloor \log_2 n \rfloor+1$

检索不成功时的判定树如下（方框表示检索不成功情况，扩充二叉树的外部结点）：



由此可见，检索不成功时，最大比较次数也就是$\lfloor \log_2 n \rfloor+1$。

分析

每次比较将范围缩小一半，第i次可能比较的元素个数：

比较次数	可能比较的元素个数
1	$1=2^0$
2	$2=2^1$
3	$4=2^2$
⋮	⋮
j	2^{j-1}

若元素个数n刚好为$2^0+2^1+\cdots\cdots+2^{j-1}=2^j-1$则最大检索长度为j；若$2^j-1<n\leqslant 2^{j+1}-1$，则最大检索长度为j+1。

所以，二分法的最大检索长度为 $\lceil \log_2(n+1) \rceil$

平均检索长度:

$$ASL = \frac{1}{n} * (\sum_{i=1}^{j} i * 2^{i-1})$$

$$= \frac{1}{n} * \sum_{i=1}^{j} \sum_{m=i}^{j} 2^{m-1}$$

$$= \frac{1}{n} * \sum_{i=1}^{j} (2^j - 2^{i-1})$$

$$= \frac{1}{n} * (j * 2^j - \sum_{i=1}^{j} 2^{i-1})$$

$$= \frac{1}{n} * (j * 2^j - 2^j + 1)$$

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

设 $n = 2^j - 1$，j 是搜索的“深度”。

二分法检索（排序顺序字典）：

- 优点是检索速度快，O(log n)
- 插入时需要维护元素顺序，O(n) 操作（检索插入位置可以用二分法，O(log n)）
- 删除可用二分法查找元素，但需移动后面元素，O(n)
- 只能用于元素按关键码排序的字典，只适合顺序存储结构

总结：排序表实现字典，检索效率高。为保持元素顺序，插入删除操作效率低。因此不适合用于实现大的动态字典

6.5 字典的散列表示

什么情况下检索元素的速度最快？

如果关键码是数组下标，可以直接找到元素（常量时间）！

- 一般说，关键码可能不是整数
- 即使是整数，也可能取值范围太大，不易直接作为数组下标。例如北大学生的学号：8位数，取值范围 0～1亿
- 散列表思想：用一个从关键码集合到适当整数区间的映射，把数据保存在映射值的那个位置

➢ 散列技术

➢ 散列函数

➢ 实现和碰撞处理

6.5.1 散列技术

散列函数：一个映射，把可能关键码映射到一段存储位置（数组下标，整数区间）。也称哈希（Hush）函数或杂凑函数。

关键码 $\xrightarrow{h}$ 存储地址

通过这种对应关系，把元素存入数组里由 h 确定的位置。

需要检索元素时，通过 h 求出对应位置后直接到数组里去找。

散列也称“杂凑”，散列表也称“杂凑表”

散列的思想在信息领域很有价值，应用极广，可能用在各种数据处理、存储、检索工作中。例如:

- 文件完整性检查（用散列函数把整个文件映射到一个数）
- 计算机安全领域中大量使用散列技术

通常可能的关键码个数远远大于存储区（数组）的位置数

因此 h 不可能是单射，必然会出现

key1≠key2　但　h(key1)=h(key2)

此时说出现碰撞（冲突），称key1和key2称为同义词。

散列表的实现必须解决碰撞问题。下面讨论实现技术时研究。

负载因子是考察散列表性质的一个重要参数。定义:

$$\alpha = \frac{\text{散列表中实际结点数}}{\text{基本区域能容纳的结点数}}$$

负载因子的大小与冲突出现的概率关系密切

注意：增大空间可以减小负载因子，也会减小出现冲突的概率

6.5.2 散列函数

散列函数的选择可能影响出现冲突的概率（显然）

- 对典型关键码分布情况，应尽可能将其映射到分散的值
- 关键码的散列值在地址区间均匀分布，可能减少冲突

常用散列函数实例（假设关键码是整数）：

1，除余法　　2，基数转换法

3，折叠法　　4，中平方法

5，数字分析法

只作为示例，实际中需要根据关键码情况和问题的情况设计

最常用的就是除余法和基数转换法。其余方法的应用很少，因为它们或者不方便，或太依赖于实际数据

1. 除余法：关键码是数，以关键码除以某个不大于散列表长m的数 p 后的余数为散列地址。数 p 一般选质数

h(key)=(int)key % p;

m = 128, 256, 512, 1024　p 选择 127, 251, 503, 1019

除余法使用最广。常用于将其他散列函数得到值归入所需区间

2. 基数转换法

把关键码看成基数为r1的数，将它转换成基数为r2的数，用数字分析法取中间几位作为散列地址。r1和r2最好互素。例

key=236075, r1=13, r2=10,

$(236075)_{13} = 2*13^5+3*13^4+6*13^3+7*13+5$

$= (841547)_{10}$

h(key) = 4154

3. 折叠法：将关键码分成几部分，以几部分的叠加和作为地址。

key = 582422241

58 | 2422 | 241　　58 | 2422 | 241

85　　58

142　　241

2422　　2422

11064　　2721

4. 中平方法

先求出关键码的平方，然后取中间几位作为地址。

key = 4731　　$(4731)^2 = 22382361$

h(key) = 382

5. 数字分析法。关键码是数，可能关键码已知，则可取关键码的若干数位作为散列地址。例：

Key	h(key)
00031942**6**	326
00071830**9**	709
00062944**3**	643
00075861**5**	715

- 对非整数关键码，常见的方式是先设计一种方式把它转换到整数，而后再用整数散列的方法
- 除余法常常用作最后一步，把关键码归结到一个范围中

例：字符串的散列

把字符串看作是基数为 29 的数字串（基数法），从一个字符串就可以算出一个整数值（用字符的编码计算）。计算中以 1019 为模，就可以把字符串散列到整数区间 [0, 1018]

采用散列方式实现字典，检索时间包括计算散列函数的时间和实际检索时间。需要根据实际情况考虑散列函数的问题

需要考虑的因素通常包括：

A. 保存散列表内容的数组的大小（根据实际需要确定）

B. 关键码的类型和长度

C. 关键码的分布情况（不同记录的检索频率）

D. 散列函数的设计

E. 计算散列函数所需时间

6.5.3 散列表的设计和实现

- 散列表的基本部分是个数组，应根据实际需要确定大小
- 数组有连续的一段下标范围（例如 0 到某个正整数）
- 散列表里存储的是字典元素（元素是关键码和值的关联）
- 根据关键码和其他因素的情况设计一个散列函数 h，其值域必须完全落在合法的数组下标范围中
- 冲突是必然会出现的事件（大集合到小集合的全函数，必然会出现两个元素的函数值相同的情况），设计散列表，必须同时设计一种冲突消解方案，处理元素冲突问题

人们提出了一些冲突消解方法，主要分为内消解方法（在基本存储区内解决）和外消解方法（在基本存储区外解决）

1，开地址法（内消解法）

基本想法：出现冲突时元素无法保存在散列函数确定的位置，需设法为它在数组里另外安排一个位置

为此要设计一种系统化的位置安排方式（探查方式）

开地址法就是生成基本区域内的一个位置探查序列

$H_i = (H(key) + d_i) \text{ MOD } m,$

$i = 1, 2, \ldots, k \ (k \le m-1)$

其中：m为表长，d_i为增量序列

增量序列可以有许多取法：

A. $d_i = 1, 2, 3, \ldots, m-1$, 称线性探查

B. 设计另一散列函数h2，令 $d_i = i * h2(key)$，称为双散列探查

当发生碰撞时，根据探查序列进行检索

- 插入时，根据探查序列向下找，直至找到一个空单元
- 检索时，根据探查序列向下找，直至遇到关键码为 key 的元素（成功）或者到达一个空单元（失败）

例：取 h(key)=key%13;

K = {18, 73, 10, 05, 68, 99, 22, 32, 46, 58, 25}

h(key) = { 5, 8, 10, 5, 3, 8, 9, 6, 7, 6, 12}

地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键码	58	25		68		18	05	32	73	99	10	22	46

线性探测容易出现堆积现象，即在处理同义词的过程中又添加了非同义词冲突。（其他探查法也可能有类似情况）

双散列探查序列

K = {18, 73, 10, 05, 68, 99, 22, 32, 46, 58, 25}

h1(key) = { 5, 8, 10, 5, 3, 8, 9, 6, 7, 6, 12}

h1(key)=key%13; h2(key)=key%11+1;

地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键码		58	25	68		18	32	46	73	99	10	05	22

随着负载因子增大到一定程度，冲突的概率明显增加。

开地址法散列表上的检索很像插入操作。对给定key值:

1. 根据散列函数求出散列地址;
2. 若该位置无记录，则检索失败结束;
3. 否则比较元素关键码，若与 key 相等则检索成功结束;
4. 否则根据本散列表设计的冲突处理方法找出“下一地址”，回到步骤 2 用这个位置检查

散列表的实现（开地址法）

数据表示

算法 散列表的检索

算法 散列表的插入


```
#define null   -1    /* null为空结点标记 */
#define RLEN …

typedef int KeyType;

typedef struct {
   KeyType key;              /* 字典元素的关键码字段    */
   /*DataType value;          /* 字典元素的属性字段    */
} DicElement;

typedef struct {
   int m;          /* m=RLEN，为基本区域长度 */
   DicElement elements[RLEN];
} HashDic;

int h(KeyType key){    return … }
```



```
int linearSearch(HashDic *phash, KeyType key, int *pos)  {
   int d = h(key), inc;  /* 设 h 为散列函数  */
   for (inc = 0; inc < phash->m; inc++) {
      if (phash->elements[d].key == key) {
         *pos = d;            /* 检索成功 */
         return TRUE;
      }
      else if (phash->elements[d].key == null) {
         *pos = d;            /* 检索失败，找到空位置 */
         return FALSE;
      }
      d = (d+1) % phash->m;
   }
   *pos = -1;                  /* 散列表溢出 */
   return FALSE;
}
```



```
int linearInsert(HashDic *phash, KeyType key) {
   int position;
   if (linearSearch(phash, key, &position) == TRUE )
      printf("Key exists \n"); /* 已有关键码为key的结点*/
   else if (position != -1) * 插入结点，应对value字段赋值 */
      phash->elements[position].key = key;
   else  return FALSE;                /* 散列表溢出 */

   return TRUE;
}
```

对于采用内冲突消解方式实现的散列表，如果希望支持元素的删除操作，散列表的各个操作都会受到影响。

作为思考题，请自己设计一种实现删除的技术

2，公共溢出区（外消解法）

另设立一个溢出表，把所有关键码冲突的记录都填入溢出表。

- 把公共溢出区作为一个线性表。
- 插入元素时遇到冲突，放入公共溢出区里下一空位
- 检索元素时先散列找到对应位置
  - 如果关键码匹配就返回
  - 如果关键码不匹配，就转到公共溢出区中顺序检索

容易看出：当散列表的元素很多，溢出区变得很大之后，元素插入和检索的工作量都会趋于线性

3，拉链法

基本思想：元素存在基本数组之外，为每个关键码建立一个链接表，所有关键字为同义词的记录存储在同一个链表中

所有元素可以统一处理（无论是否冲突）。允许任意的负载因子。最简单的方法是把新元素插在链接表头。如果要防止出现重复关键码，就需要检查整个链表

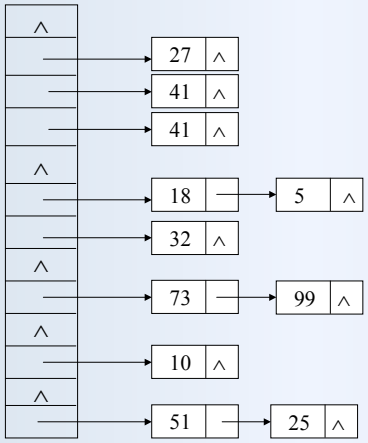
例:

K={18, 73, 10, 05, 68, 99, 27, 41, 51, 32, 25}

{ 5, 8, 10, 5, 3, 8, 1, 2, 12, 6, 12}

h(key)=key%13;

{ { }, {{27}, {41}, {68}, { }, {05, 18}, {32}, { },
{99, 73}, { }, {10}, { }, {25, 51} }



K={18, 73, 10, 05, 68, 99, 27, 41, 51, 32, 25}

h(key)=key%13

拉链法散列表

在实际应用中上述方法可以推广，同义词表可以采用顺序表或散列表，或者其他（字典）实现方式

有时把存放同义词的表称为“桶”，这种结构也称为“桶散列”。这种结构可以用于存储大型字典，用于组织文件

一般情况（采用内部消解时）

负载因子 $\alpha \leqslant 0.5 \sim 0.75$ 时，平均检索长度几乎为常数

采用桶散列，负载因子就是桶的平均大小（平均长度），因此可以容忍任意大的负载因子。但随着负载因子变大，检索时间也趋于线性（桶长 = 元素个数/桶数）

如果效率不能满足需要，可扩大散列桶的个数!

明显：时间与空间交换!

6.5.4 散列文件：桶散列的应用

使用外存文件时，为了节省外存存取时间，需尽量减少访外次数。减少访外次数的有效方法是精心设计文件结构，使每次访外能存取更多应用程序中需要的逻辑记录的信息

散列文件是存在外存的大型字典结构，把散列的思想用于外存。将散列函数作用到记录的关键码，确定记录在外存的存储地址，处理冲突常用适合外存分块存取的拉链法

实现散列文件的常见方法就是按桶散列，只是这里存储桶里的一个链表结点对应于外存的一个页块，各种链接都直接采用外存的页块地址

检索和插入、删除操作中涉及到访问外存。其余与前面讨论的桶散列的链接表一样

小　　结

集合是最简单的结构。本章介绍了集合的概念，主要讨论了集合的位向量实现和链表实现。

字典是一种特殊集合，最主要操作为检索本章介绍字典的各种存储表示及相应检索方法，包括

- 线性表示（顺序表表示，链表表示和目录表表示等）
- 散列表示（开地址法、拉链法和桶散列等）

其中顺序表示的线性表、开地址法处理碰撞的散列表和最佳二叉排序树主要使用于静态或基本静态的字典

链表表示的线性表、拉链法处理碰撞的散列表、桶散列结构较多考虑到元素的动态处理，适合于组织各种动态字典

桶散列结构常用于外存的大型字典表示