第7章 时空权衡

本章主要内容

- 时空权衡的方法
- 计数排序
- 串匹配中的输入增强技术
- 散列法
- B树
- 要求
  - 掌握时空权衡的概念及基本方法，掌握时空权衡的方法在常见问题中的应用。

时空权衡算法思想

- 时空权衡在算法设计中是一个众所周知的问题
  - 对问题的部分或全部输入做预处理，然后对获得的额外信息进行存储，以加速后面问题的求解——输入增强
  - 使用额外空间来实现更快和（或）更方便的数据存取——预构造

时空权衡

时空权衡是指在算法的设计中，对算法的时间和空间作出权衡。

常见的以空间换取时间的方法有：

- 输入增强
  - **计数排序**
  - **字符串匹配中的输入增强技术**
- 预构造
  - **散列法**
  - **B树**

计数排序

- 针对待排序列表中的每个元素，算出列表中小于该元素的元素个数，并把结果记录在一张表中。
  - 这个“个数”指出了元素在有序列表中的位置
  - 可以用这个信息对列表的元素排序，这个算法称为“比较计数”

- 思路：针对待排序列表中的每一个元素，算出列表中小于该元素的元素个数，把结果记录在一张表中。

数组 A[0..5]

62	31	84	96	19	47

	Count []					
初始	0	0	0	0	0	0
$i=0$ 遍之后	3	0	1	1	0	0
$i=1$ 遍之后		1	2	2	0	1
$i=2$ 遍之后			4	3	0	1
$i=3$ 遍之后				5	0	1
$i=4$ 遍之后					0	2
最终状态	3	1	4	5	0	2

数组 S[0..5]

19	31	47	62	84	96

```
算法 Comparison(A[0…n-1])
{  //用比较计数法对数组排序
   for(i=0;i < n;i++)    Count[i]=0;
   for(i=0;i < n-1;i++)
       for(j=i+1;j < n; j++)
                  if(A[i]<A[j]) Count[j]++;
                  else  Count[i]++;
  for(i=0;i < n ; i ++) S[Count[i]] = A[i];
   return  S;
}
```

$$
\begin{aligned}
C(n) &= \sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1} 1 \\
&= \sum_{i=0}^{n-2}[(n-1)-(i+1)+1] \\
&= \sum_{i=0}^{n-2}(n-1-i) \\
&= \frac{n(n-1)}{2}
\end{aligned}
$$

计数排序

- 该算法执行的键值比较次数和选择排序一样多，并且还占用了线性数量的额外空间，所以几乎不能来做实际的应用
- 但在一种情况下还是卓有成效的——待排序的元素值来自一个已知的小集合
  - 如待排序集合只有多个1，2元素（更一般：元素位于下界 l 和上界u之间的整数）
  - 那么我们可以使用计数排序方法，扫描列表中1和2的数目，然后重排列就可以了（只有我们可以改写给定的元素时才成立）

计数排序

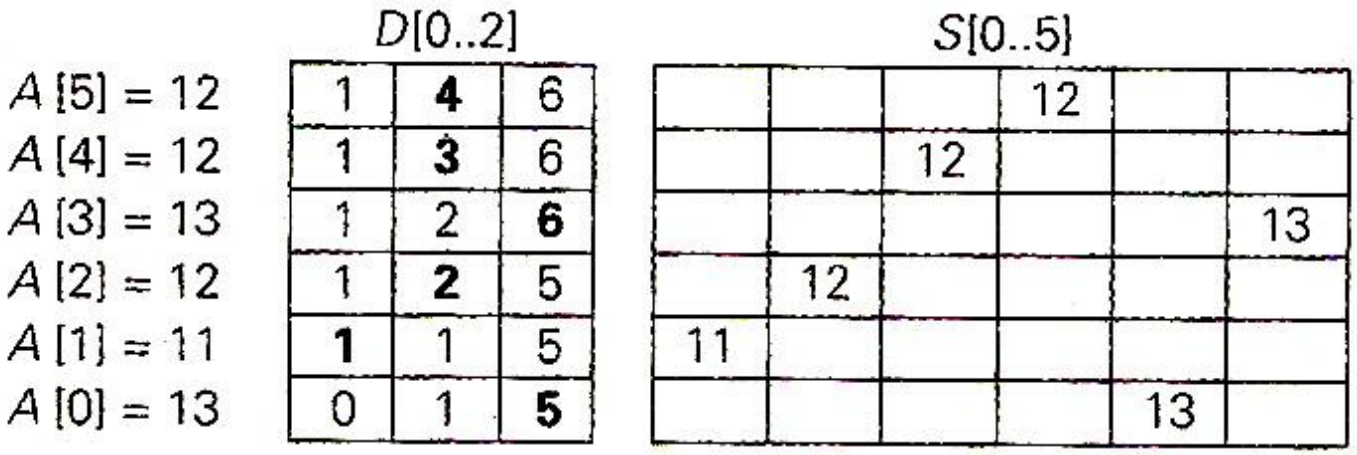
- 另一种更现实的情况：待排序的数组元素有一些其他信息和键值相关（不能改写列表的元素）
  - **将A数组元素复制到一个新数组S[0…n-1]中**
  - **A中元素的值如果等于最小的值*l*，就被复制到S的前F[0]个元素中，即位置0到F[0]-1中**
  - **值等于*l+1*的元素被复制到位置F[0]至(F[0]+F[1])-1，以此类推。**
- 因为这种频率的累积和在统计中称为**分布**，这个方法也称为“**分布计数**”。

计数排序算法分析实例

13	11	12	13	12	12

数组值	11	12	13
频率	1	3	2
分布值	1	4	6

```
算法 DistributionCountingt(A[0..n-1],L，U)
for(j←0 to u-l)  D[j]←0;
for(i←0 to n-1) D[A[i]-l]←D[A[i]-l]+1;
for(j←1 to U-L) D[j]←D[j-1]+D[j];
for(i←n-1 downto 0){
        j←A[i]-L;
        S[D[j]-1]←A[i];
        D[j]←D[j]-1;
}
return S;
```



```
算法 DistributionCounting(A[0…n-1],l,u)
{  //分布计数法对有限范围整数的数组排序
   for(j=0;j <= u-l;++i)  D[j]=0;//初始化频率数组
   for(i=0;i < n; ++i)      D[A[i]-l]++;//计算频率值
   for(j=1;j <= u-l; ++j)  D[j]+=D[j-1];//重用分布
  for(i=n-1;i>=0;--i){
        j = A[i] – l;
        S[D[j]-1] = A[i];
        D[j]--;
   }
   return  S;
}
```

- 假设数组值的范围是固定的，那么这是一个线性效率的算法
- 但重点是：除了空间换时间之外，分布技术排序的这种高效是因为利用了输入列表独特的自然属性！

散列法

- 考虑一种非常高效的实现字典的方法
  - **字典是一种抽象数据类型，即一个在其元素上定义了查找、插入和删除操作的元素集合**
  - **集合的元素可以是容易类型的，一般为记录**
- 散列法的基本思想是：把键分布在一个称为散列表的一维数组H[0，…，m-1]中。
  - **可以通过对每个键计算某些被称为“散列函数”的预定义函数h的值，来完成这种发布**
  - **该函数为每个键指定一个称为“散列地址”的位于0到m-1之间的整数**

散列法

- 散列函数需要满足两个要求：
  - **散列函数需要把键在散列表的单元格中尽可能均匀地分布（所以m常被选定为质数，甚至必须考虑键的所有比特位）**
  - **散列函数必须容易计算**
- 散列的主要版本：
  - **开散列（分离链）**
  - **闭散列（开式寻址）**

开散列（分离链）

- 键被存储在附着于散列表单元格上的链表中，散列地址相同的记录存放于同一单链表中
- 查找时：首先根据键值求出散列地址，然后在该地址所在的单链表中搜索；

开散列（分离链）

- 查找效率取决于链表的长度，而这个长度又取决于字典和散列表的长度以及散列函数的质量
  - **若散列函数大致均匀地将n个键分布在散列表的m个单元格中，每个链表的长度大约相当于n/m个**

➢成功查找和不成功查找中平均需检查的个数S和U：

$$S \approx 1 + \frac{\alpha}{2} \qquad U = \alpha$$

➢之所以能得到这样卓越的效率，不仅是因为这个方法本身就非常精巧，而且也是以额外的空间为代价的

➢插入和删除在平均情况下都是属于Θ(1)

闭散列（开式寻址）

- 所有的键值都存储在散列表本身中，而没有使用链表（这表示表的长度m至少必须和键的数量一样大）

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

- 删除操作：[illegible]延迟删除[illegible]，用一个特殊的符号来标记曾被占用过的位置，以把它们和那些从未被只用过的位置区别开来

闭散列（开式寻址）

- 所有键都存储在散列表本身，采用线性探查解决冲突，即碰撞发生时，如果下一个单元格空，则放下一个单元格，如果不空，则继续找到下一个空的单元格，如果到了表尾，则返回到表首继续。

键	A	FOOL	AND	HIS	MONEY	ARE	SOON	PARTED
散列地址	1	9	6	10	7	11	11	12

0	1	2	3	4	5	6	7	8	9	10	11	12
	A								FOOL			
	A								FOOL			
	A					AND			FOOL	HIS		
	A					AND			FOOL	HIS		
	A					AND	MONEY		FOOL	HIS		
	A					AND	MONEY		FOOL	HIS	ARE	
	A					AND	MONEY		FOOL	HIS	ARE	SOON
PAETED	A					AND	MONEY		FOOL	HIS	ARE	SOON

- 散列法的基本思想：
  - **把键分布在一个称为散列表的一维数组H[0..m-1]中。**
  - **可以利用*散列函数*来计算每个键的值，该函数为每个键指定一个称为*散列地址*的值，该值是位于0到m-1之间的整数。**
  - **如果键是一个非负整数，则h(K)=K mod m**
  - **如果键是某个字母表中的字母，则可以把该字母在字母表中的位置指定个键，记为ord(K)**
  - **如果键是一个字符串$c_0c_1...c_{s-1}$，则定义**
    - $h(K)=(\sum ord(c_i)) \bmod m$
    - 或者h←0; for i←0 to s-1 do h←$(h*C+ord(c_i)) \bmod m$

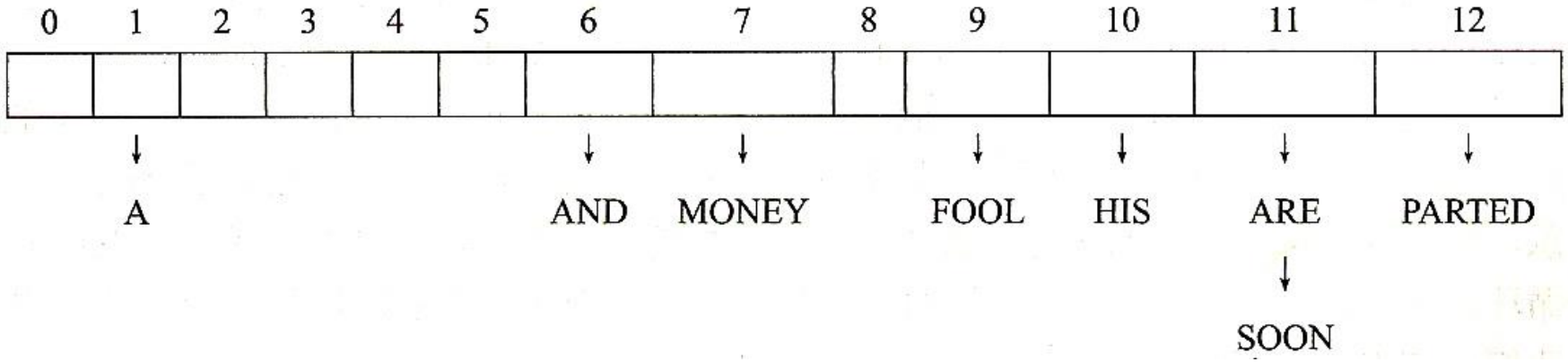
- 一个散列函数必须满足的两个要求：
  - **需要把键在散列表的单元中尽可能的均匀分布**
  - **必须是容易计算的**
- 碰撞
  - **当三列表的长度m小于键的数量n时，会有两个或多个键被三列到同一个单元中**
  - **即时m相对于n足够大，碰撞还是会发生**
- 散列法的两个版本
  - **开散列（分离链）**
  - **闭散列（开式寻址）**

开散列（分离链）

- 在开散列中，键被存放于散列表单元的链表中。

A,FOOL,AND,HIS,MONEY,ARE,SOON,PARTED

键	A	FOOL	AND	HIS	MONEY	ARE	SOON	PARTED
散列地址	1	9	6	10	7	11	11	12



- 一般来说，查找的效率取决于链表的长度，而这个长度有取决于字典和散列表的长度以及散列函数的质量。
- 如果该散列函数大致均匀地将n个键分布在散列表的m个单元中，每个链表的长度大约相当于n/m，其α =n/m称为散列表的***负载因子***。
- 成功查找中平均需检查的指针个数S=1+ α /2
- 不成功查找中平均需检查的指针个数U= α
- 通常情况下，我们希望负载因子和1不要相差太大。

闭散列（开式寻址）

- 所有键都存储在三列表本身，采用线性探查解决冲突，即碰撞发生时，如果下一个单元格空，则放下一个单元格，如果不空，则继续找到下一个空的单元格，如果到了表尾，则返回到表首继续。

键	A	FOOL	AND	HIS	MONEY	ARE	SOON	PARTED
散列地址	1	9	6	10	7	11	11	12

0	1	2	3	4	5	6	7	8	9	10	11	12
	A								FOOL			
	A								FOOL			
	A					AND			FOOL	HIS		
	A					AND			FOOL	HIS		
	A					AND	MONEY		FOOL	HIS		
	A					AND	MONEY		FOOL	HIS	ARE	
	A					AND	MONEY		FOOL	HIS	ARE	SOON
PAETED	A					AND	MONEY		FOOL	HIS	ARE	SOON

- 闭散列的查找和插入操作是简单而直接的，但是删除操作则会带来不利的后果。
- 比起分离链，现行探查的数学分析是一复杂的多的问题。
- 对于复杂因子为α的散列表，成功查找和不成功查找必须要访问的次数分别为：
  - **$S \approx (1+1/(1-\alpha))/2$　　$U \approx (1+1/(1-\alpha)^2)/2$**
  - **散列表的规模越大，该近似值越精确**

本章小结

- 空间换时间技术有两种主要的类型：**输入增强**和**预构造**。
- **分布计数**是一种特殊方法，用来对元素取值来自于一个小集合的列表排序。
- 串匹配的**Horspool算法**是**Boyer-Moore算法**的简化，都以**输入增强**技术为基础，且从右向左比较模式中的字符。
- **散列**是一种非常高效的实现字典的方法，分为开散列和闭散列，其中必须采用碰撞解决机制。
- **B树**是一棵平衡查找树。

. 给定字符串str1和str2，求str1子串中包含str2所有字符的最小子串长度。