

程序设计方法与实践

——时空权衡

时空权衡

- 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

时空权衡

- 7.1 计数排序
- 7.2 串匹配中的输入增强技术
- 7.3 散列法
- 7.4 B树

7.1 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”

7.1 计数排序

- 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
----	----	----	----	----	----

初始

Count []	0	0	0	0	0
----------	---	---	---	---	---

$i = 0$ 遍之后

Count []	3	0	1	1	0
----------	---	---	---	---	---

$i = 1$ 遍之后

Count []		1	2	2	0
----------	--	---	---	---	---

$i = 2$ 遍之后

Count []			4	3	0
----------	--	--	---	---	---

$i = 3$ 遍之后

Count []				5	0
----------	--	--	--	---	---

$i = 4$ 遍之后

Count []					0
----------	--	--	--	--	---

最终状态

Count []	3	1	4	5	0
----------	---	---	---	---	---

数组 S[0..5]

19	31	47	62	84	96
----	----	----	----	----	----

7.1 计数排序

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;

}

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

$$= \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1]$$

$$= \sum_{i=0}^{n-2} (n-1-i)$$

$$= \frac{n(n-1)}{2}$$

7.1 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）
 - 那扫描列表中1和2的数目计入 $F[0]$ 和 $F[1]$, ($F[0] \sim F[u-l]$)
 - 有序列表中前 $F[0]$ 个就是1，接下来 $F[1]$ 个是2。

7.1 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”。

7.1 计数排序

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){ //开始放入数据
    j ← A[i]-L;
    S[D[j]-1] ← A[i]; //地址在S[D[j]-1]
    D[j] ← D[j]-1;
}
return S;

```

$A[5] = 12$
 $A[4] = 12$
 $A[3] = 13$
 $A[2] = 12$
 $A[1] = 11$
 $A[0] = 13$

$D[0..2]$

1	4	6
1	3	6
1	2	6
1	2	5
1	1	5
0	1	5

$S[0..5]$

			12		
		12			
					13
	12				
11					
				13	

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 ;++j) 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];//分布值

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，分布计数排序的这种高效是因为利用了输入列表独特的自然属性！

时空权衡

- 7.1 计数排序
- 7.2 串匹配中的输入增强技术
- 7.3 散列法
- 7.4 B树

7.2 串匹配中的输入增强技术

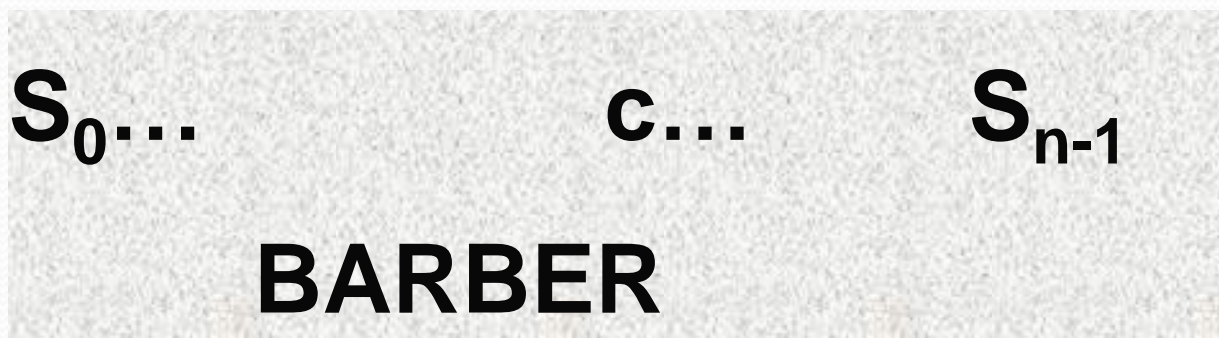
- 字符串匹配问题：要求在一个较长的 n 个字符的串（称为**文本**）中，寻找一个给定的 m 个字符的串（称为**模式**）。
- 蛮力法：简单地**从左到右**比较模式和文本中每一个对应的字符，如果不匹配，把文本向右移动一格，再进行下一轮尝试，**最差效率为 $O(nm)$** ，随机文本的平均效率 $O(n)$
- 输入增强技术：对模式进行预处理以得到它的一些信息，把这些信息存储在表中，然后在给定文本中实际查找模式的时候使用这些信息——Knuth-Morris_Pratt(KMP)算法和Boyer-Moore(BM)算法

7.2 串匹配中的输入增强技术

- Knuth-Morris_Pratt算法和Boyer-Moore算法的主要区别在于：**前者是从左到右，后者是从右到左**
- 但因为后者更简单，所以**我们只考虑Boyer-Moore算法**：
 - 开始的时候把模式和文本的开头字符对齐。如果第一次尝试失败了，把模式向右移。
 - 只是每次尝试过程中的**比较是从右向左的**，即从模式的最后一个字符开始
 - Horspool算法和Boyer-Moore算法的简化版

7.2 Horspool算法

- 从模式的最后一个字符开始从右向左，比较模式和文本的相应字符
 - 如果模式中所有的字符都匹配成功，就找到了一个匹配子串，就可以停止了
 - 如果遇到一对不匹配的，把模式右移



$S_0 \dots C \dots S_{n-1}$

BARBER

7.2 Horspool算法

- 根据对齐模式的**最后一个字符c**的不同情况确定移动的距离：
 - 情况1：模式不存在c，模式安全移动的幅度就是它的**全部长度m**
 - 情况2：模式存在c，且不是模式末尾字符，移动时把模式中**最右的c与文本中的c对齐**
 - 情况3：c是模式的最后一个字符，且模式中不包含其他c，移动幅度等于**模式长度m**
 - 情况4：c是模式的最后一个字符，而模式前m-1个字符中包含c，把模式中**前m-1个字符中的c和文本中的c对齐**

7.2 Horspool算法

考虑在某些文本中查找模式BARBER

$s_0 s_1 \dots$ $c \dots$ s_{n-1}

BARBER

情况1: 模式中无 c ,
移动幅度等于模式长度

$s_0 s_1 \dots$ $s \dots$ s_{n-1}

BARBER

BARBER

$s_0 s_1 \dots$ $B \dots$ s_{n-1}

BARBER

BARBER

情况2: 模式有 c 且末尾不是 c
模式最右 c 与文本 c 对齐

$s_0 s_1 \dots$ $MER \dots$ s_{n-1}

LEADER

LEADER

情况3: 模式有 c 且仅末尾是 c
移动幅度等于模式长度

$s_0 s_1 \dots$ $OR \dots$ s_{n-1}

REORDER

REORDER

情况4: 模式有多' R '且末尾是' R '
模式除末尾外最右 c 与文本 c 对齐

7.2 Horspool算法

- 比起蛮力法每次只移动一个位置，该算法移动的更远
- 但如果为了移动的更远就需要每次都检查模式中的每个字符，它的优势也会丧失
 - **时空权衡**：预先算出每次移动的距离并把它们存在表中，将距离填入表中的单元格中
 - 这个表是**以文本中所有可能遇到的字符为索引**的
 - 对于每个字符c可用公式算出移动距离：

$$t(c) = \begin{cases} \text{模式的长度}m(\text{如果}c\text{不包含在模式前}m-1\text{个字符中}) \\ \text{模式前}m-1\text{个字符中最右边的}c\text{到模式最后一个字符的距离} \end{cases}$$

7.2 Horspool算法

- 例如模式为 “BARBER”，那么文本中除了E,B,R, A的单元格分别为1,2,3,4外，其他的都为6
- 一个简单的算法用来计算移动表中每个单元格的值：
 - 初始时，把所有的单元格都置为模式的长度 m
 - 然后从左到右扫描模式，将下列步骤重复 $m-1$ 次
 - 对于模式中的第 j 个字符，将他在表中的单元格改写为 $m-1-j$ ，这是该字符到模式右端的距离

7.2 Horspool算法

- 第一步：对于给定的长度为 m 的模式和在模式及文本中用到的字母表，按照上面的描述构造移动表
- 第二步：将模式与文本的开始处对齐
- 第三步：重复下面的过程，直到发现了一个匹配子串或者模式到达了文本的最后一个字符以外。
 - 从模式的最后一个字符开始，比较模式和文本中的相应字符，
 - 直到：要么所有 m 个字符都匹配（然后停止），要么遇到了一对不匹配的字符。
 - 后者，如果 c 是当前文本中的和模式的最后一个字符相对齐的字符，从移动表的第 c 列中取出单元格 $t(c)$ 的值，然后将模式沿着文本向右移动 $t(c)$ 个字符的距离

7.2 Horspool算法

算法 *HorspoolMatching* ($P[0..m-1], T[0..n-1]$)

//实现 Horspool 串匹配算法

//输入: 模式 $P[0..m-1]$ 和文本 $T[0..n-1]$

//输出: 第一个匹配子串最左端字符的下标, 但如果没有匹配子串, 则返回-1

ShiftTable($P[0..m-1]$) // 生成移动表

$i \leftarrow m-1$ // 模式最右端的位置

while $i \leq n-1$ **do**

$k \leftarrow 0$ // 匹配字符的个数

while $k \leq m-1$ **and** $P[m-1-k] = T[i-k]$

$k \leftarrow k+1$

if $k = m$

return $i-m+1$

else $i \leftarrow i + \text{Table}[T[i]]$

return -1

算法 *ShiftTable*($P[0..m-1]$)

//用Horspool算法和Boyer-Moore算法填充移动表

//输入: 模式 $p[0..m-1]$ 以及一个可能出现字符的字符表

//输出: 以字母表中字符为索引的数组 $\text{table}[0..\text{size}-1]$

把Table中的所有元素初始化为m;

for($j \leftarrow 0$ to $m-2$) **do**

$\text{Table}[P[j]] \leftarrow m-1-j;$

return Table;

7.2 Horspool算法

字符 c	A	B	C	D	E	F	...	R	...	Z	-
移动距离 $t(c)$	4	2	6	6	1	6	6	3	6	6	6

在特定文本中的实际查找是像下面这样的：

J I M _ S **A** W _ M **E** _ I N _ A _ **B** A **R** B E R S H O P
B A R B E R B A R B E R
 B A R B E R B A R B E R
 B A R B E R B A R B E R

- Horspool算法的最差效率 $\Theta(mn)$ why? 习题7.2-4
- 对于随机文本，它的效率为 $\Theta(n)$

P202 习题7.2-4

4. 用Horspool算法在一个长度为 n 的文本中查找一个长度为 m 的模式,请分别给出下面两种例子.

a.最差输入 b.最优输入

a. 在 n 个“0”组成的文本中查找“100...0”(长度为 m),

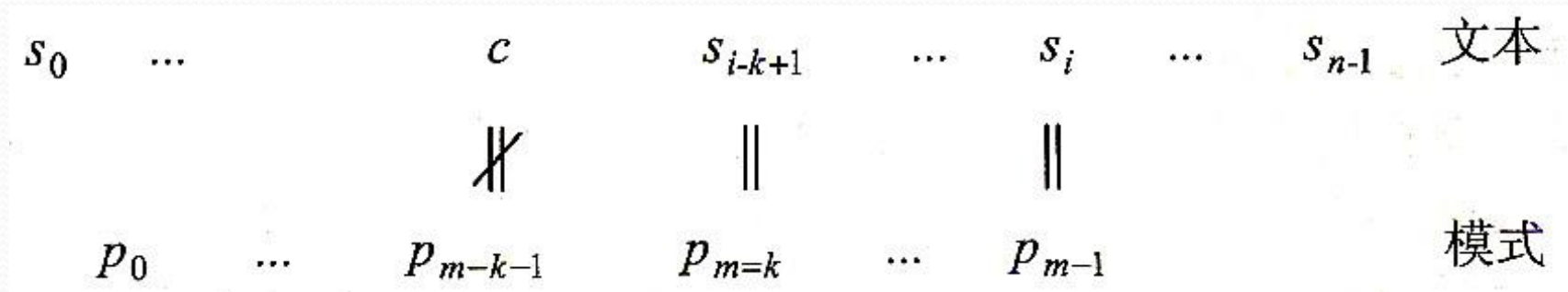
查找次数 $C(\text{worst})=m(n-m+1)$

b. 在 n 个“0”组成的文本中查找由 m 个“0”组成的模式,

查找次数 $C(\text{best})=m$

7.2 Boyer-Moore算法

- 如果在遇到一个不匹配字符之前，如果已经有 $k(0 < k < m)$ 个字符匹配成功，则Boyer-Moore算法与Horspool算法处理不同。



在这种情况下，Boyer-Moore算法参考两个数值来确定移动距离。第一个数值是有文本中的第一个坏字符 c 所确定，用公式 $t_1(c) - k$ 来计算其中 $t_1(c)$ 是Horspool算法用到的预先算好的值， k 是成功匹配的字符个数

7.2 Boyer-Moore算法

s_0 ... S E R ... s_{n-1}
~~||~~ || ||
 B A R B E R
 B A R B E R

$$t_1(S) - 2 = 6 - 2 = 4$$

s_0 ... A E R ... s_{n-1}
~~||~~ || ||
 B A R B E R
 B A R B E R

$$t_1(A) - 2 = 4 - 2 = 2$$

$$d_1 = \max\{t_1(c) - k, 1\}$$

坏符号移动

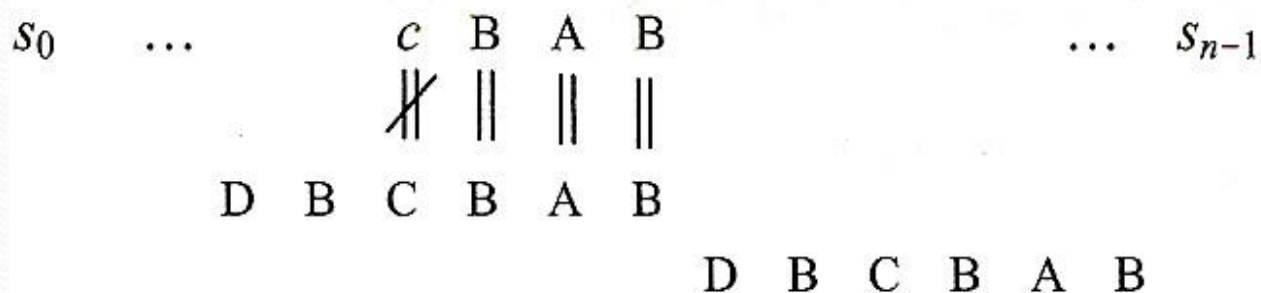
Boyer-Moore算法

- 第二个数值是由模式中最后 $k > 0$ 个成功匹配的字符所确定。
称为好后缀移动
- 把模式的结尾部分叫做模式的长度为 k 的后缀，记作 $\text{suff}(k)$
- 情况1**：当模式中存在两个以上 $\text{suff}(k)$ 的情况时，移动距离 d_2 就是从右数第二个 $\text{suff}(k)$ 到最右边的 $\text{suff}(k)$ 之间的距离。

k	模式	d_2
1	ABC <u>B</u> AB	2
2	AB <u>CD</u> AB	4

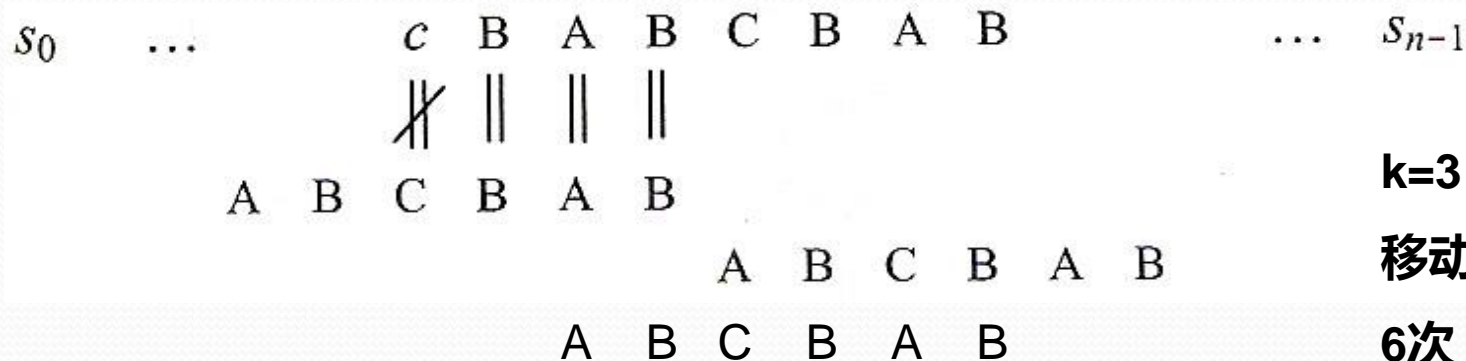
7.2 Boyer-Moore算法

- 情况2: 当模式中存在1个 $\text{suff}(k)$ 的情况时:



$k=3$

移动6



$k=3$

移动? 次

6次 or 4次

7.2 Boyer-Moore算法

- 为了避免情况2的出现，我们需要找出长度为 $l < k$ 的最长前缀，它能够和长度同样为 l 的后缀完全匹配。
- 如果存在这样的前缀，我们通过求出这样的前缀和后缀之间的距离来作为移动距离 d_2 的值，否则移动距离就是 m

k	模式	d_2
1	ABC <u>B</u> A <u>B</u>	2
2	<u>AB</u> CB <u>AB</u>	4
3	<u>ABC</u> B <u>AB</u>	4
4	<u>ABCB</u> A <u>B</u>	4
5	<u>ABCBAB</u>	4

7.2 Boyer-Moore算法

Boyer-Moore 算法

- 第一步：对于给定的模式和在模式及文本中用到的字母表，按照给出的描述构造坏符号移动表。
- 第二步：按照给出的描述，利用模式来构造好后缀移动表。
- 第三步：将模式与文本的开始处对齐。
- 第四步：重复下面的过程，直到发现了一个匹配子串或者模式到达了文本的最后一个字符以外。从模式的最后一个字符开始，比较模式和文本中的相应字符，直到：要么所有 m 个字符都匹配（然后停止），要么在 $k \geq 0$ 对字符成功匹配以后，遇到了一对不匹配的字符。在第二种情况下，如果 c 是文本中的不匹配字符，我们从坏符号移动表的第 c 列中取出单元格 $t_1(c)$ 的值。如果 $k > 0$ ，还要从好后缀移动表中取出相应的 d_2 的值。然后将模式沿着文本向右移动 d 个字符的距离， d 是按照这个公式计算出来的：

$$d = \begin{cases} d_1 & \text{如果 } k = 0 \\ \max\{d_1, d_2\} & \text{如果 } k > 0 \end{cases} \quad (7.3)$$

其中， $d_1 = \max\{t_1(c) - k, 1\}$ 。

7.2 Boyer-Moore算法

- 在一个由英文字母和空格构成的文本中查找BAOBAB

坏符号

c	A	B	C	D	...	O	...	Z	-
$t_1(c)$	1	2	6	6	6	3	6	6	6

移动表

好后缀

移动表

k	模式	d_2
1	BAO <u>B</u> AB	2
2	<u>B</u> AOBAB	5
3	<u>B</u> AOBAB	5
4	<u>B</u> AOBAB	5
5	<u>B</u> AOBAB	5

B E S S _ **K** N E W _ A B O U T _ B A O B A B S
 B A O B A B

$$d_1 = t_1(K) - 0 = 6$$

B A O B A B

$$d_1 = t_1(-) - 2 = 4$$

B A O B A B

$$d_2 = 5$$

$$d_1 = t_1(-) - 1 = 5$$

$$d = \max\{4, 5\} = 5$$

$$d_2 = 2$$

$$d = \max\{5, 2\} = 5$$

B A O B A B

时空权衡

- 7.1 计数排序
- 7.2 串匹配中的输入增强技术
- **7.3 散列法**
- 7.4 B树

7.3 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
 - 集合的元素可以是任意类型的，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 $H[0, \dots, m-1]$ 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

7.3 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)

7.3 散列法——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；

例：

元素键值为：

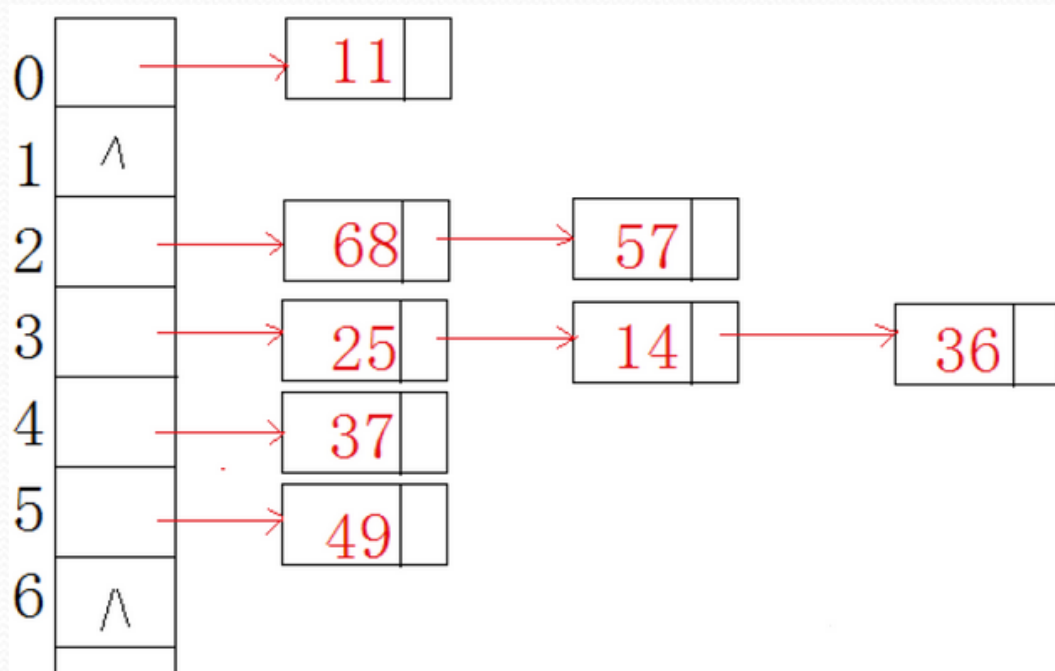
37、25、14、36、

49、68、57、11

散列表为HT[11]

散列函数为：

$\text{Hash}(x) = x \% 11$



7.3 散列法——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

➤ 成功查找和不成功查找中平均需检查的个数S和U:

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

➤ 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 $\Theta(1)$

7.3 散列法——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）
- 解决碰撞：有多种方法，例如线性探测
- 插入和查找：简单而直接
- 删除操作：“延迟删除”，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用过的位置区别开来

7.3 散列法——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

闭散列（开式寻址）

- 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：

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

- 散列表的规模越大，该近似值越精确

时空权衡

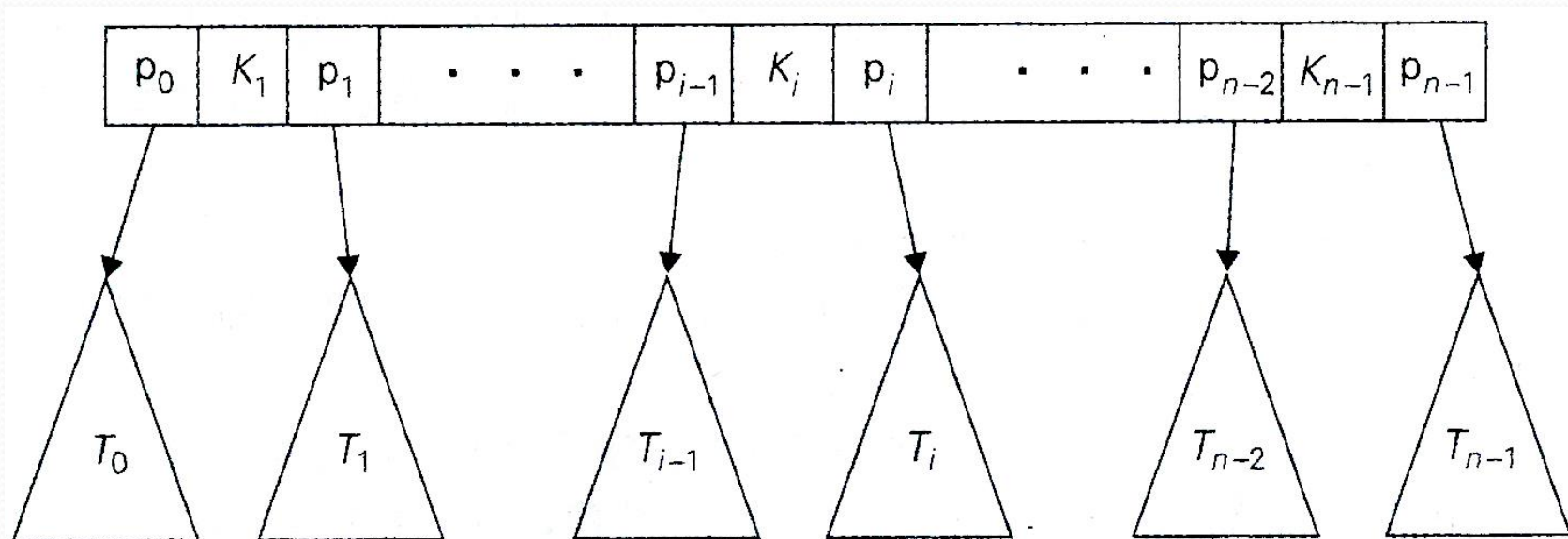
- 7.1 计数排序
- 7.2 串匹配中的输入增强技术
- 7.3 散列法
- **7.4 B树**

7.4 B树

- B树：所有的数据记录（或者键）都按照键的升序存储在叶子中；它们的父母节点作为索引
 - 每个父母节点包含 $n-1$ 个有序的键 $K_1 < \dots < K_{n-1}$
 - 这些键之间有 n 个指向子女的指针，使得子树 T_0 中的所有键都小于 K_1 ，子树 T_1 中的大于等于 K_1 小于 K_2 ，以此类推

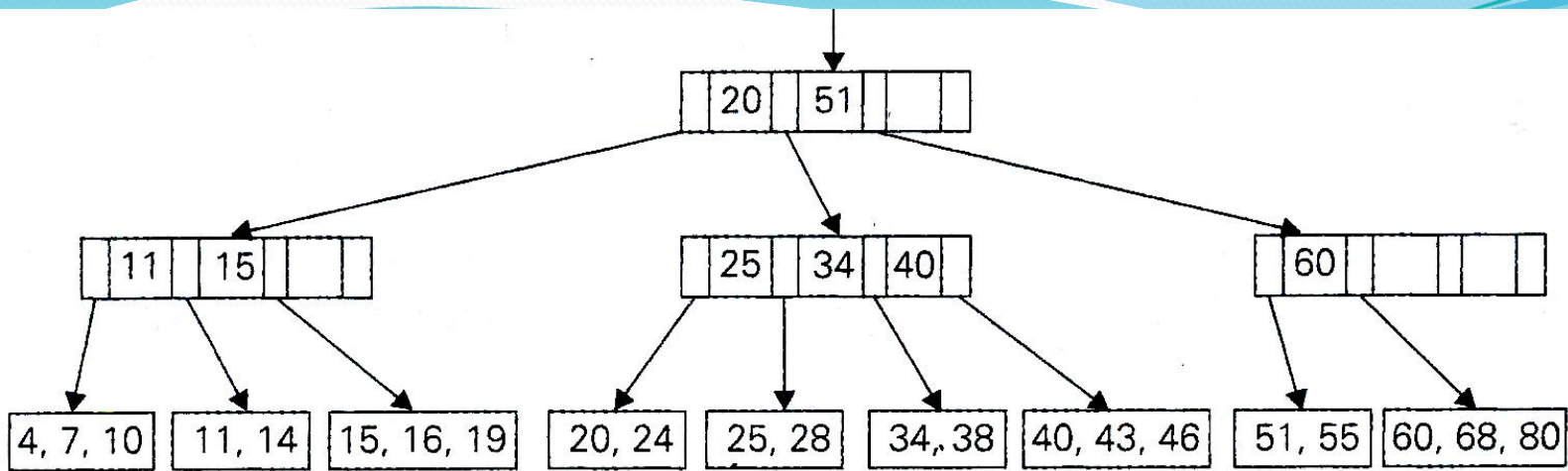
7.4 B树

- 在B树中，所有的数据记录都按照键的增序存储在叶子中，它们的父节点作为索引。



7.4 B树

- 一棵次数为 $m \geq 2$ 的B树必须满足下面这些特性：
 - 它的根要么是一个叶子，要么具有2到 m 个子女
 - 除了根和叶子以外的每个节点，具有 $m/2$ 到 m 个子女
 - 这棵树是（完美）平衡的，也就是说，它的所有叶子都是在同一层上



次数为4的B树

- 在查找键给定的某条记录中，需要访问多少个B树的节点？

对于任何包含n个key值、次数为m、高度为h>0的B树来说，有：

$$n \geq 1 + \sum_{i=1}^{h-1} 2 \lceil m/2 \rceil^{i-1} (\lceil m/2 \rceil - 1) + 2 \lceil m/2 \rceil^{h-1}$$

$$n \geq 4 \lceil m/2 \rceil^{h-1} - 1$$

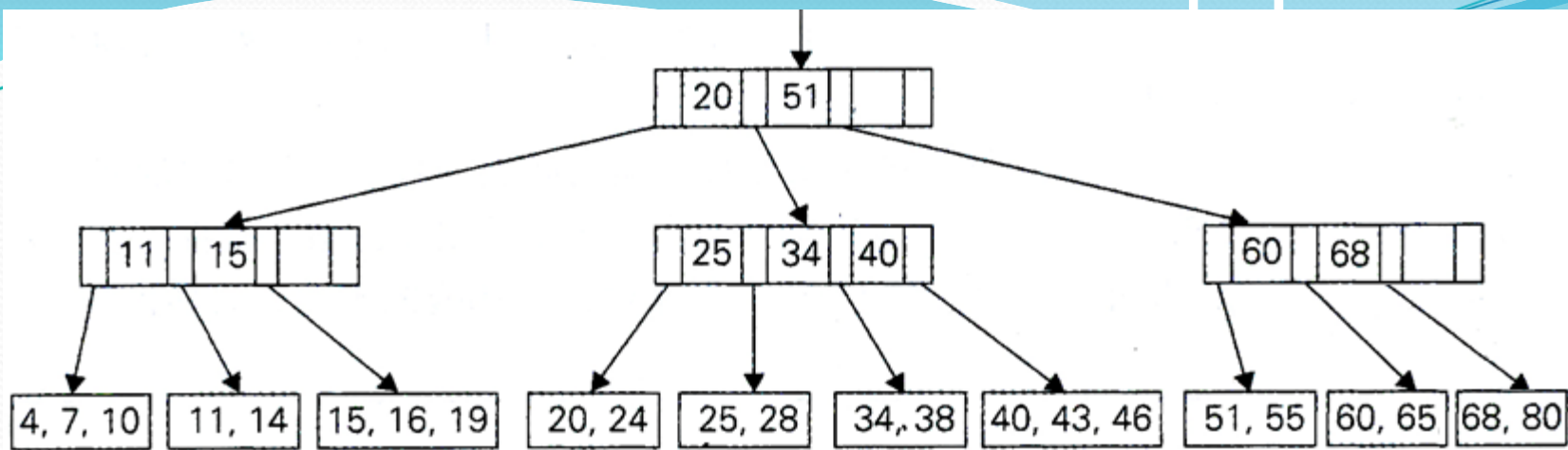
$$h \leq \left\lceil \log_{\lceil m/2 \rceil} \frac{n+1}{4} \right\rceil + 1$$

7.4 B树

应用B树：磁盘访问

- 当一个文件包含 1亿条记录时：

次数	50	100	250
h 上界	6	5	4



- 插入数据

- 查找到叶节点。
- 如果叶节点有空隙，插入数据。
- 如果叶节点没有空隙，叶子分裂。后面叶子的最小key值插入父节点。
- 向上回溯，分裂树枝，直到树根，如果也满了，树根分裂。

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。