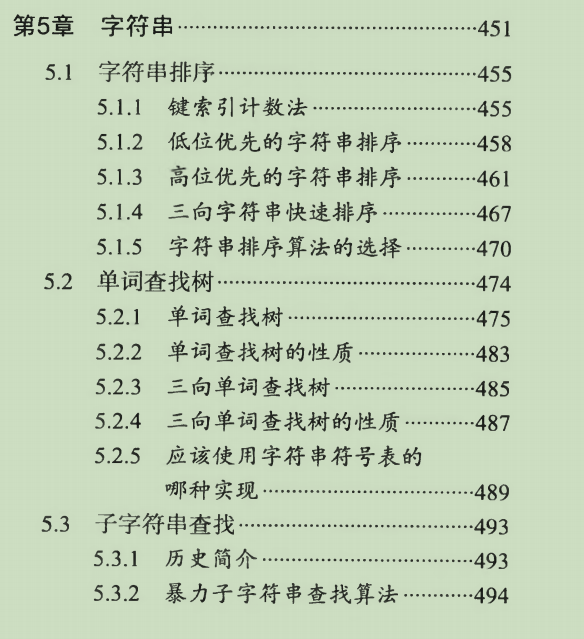
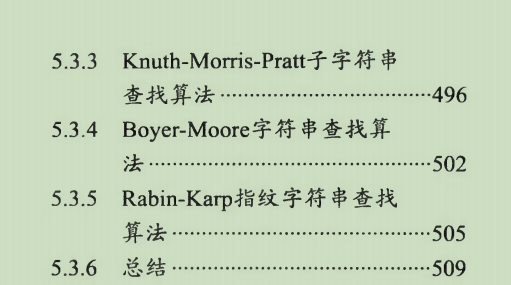
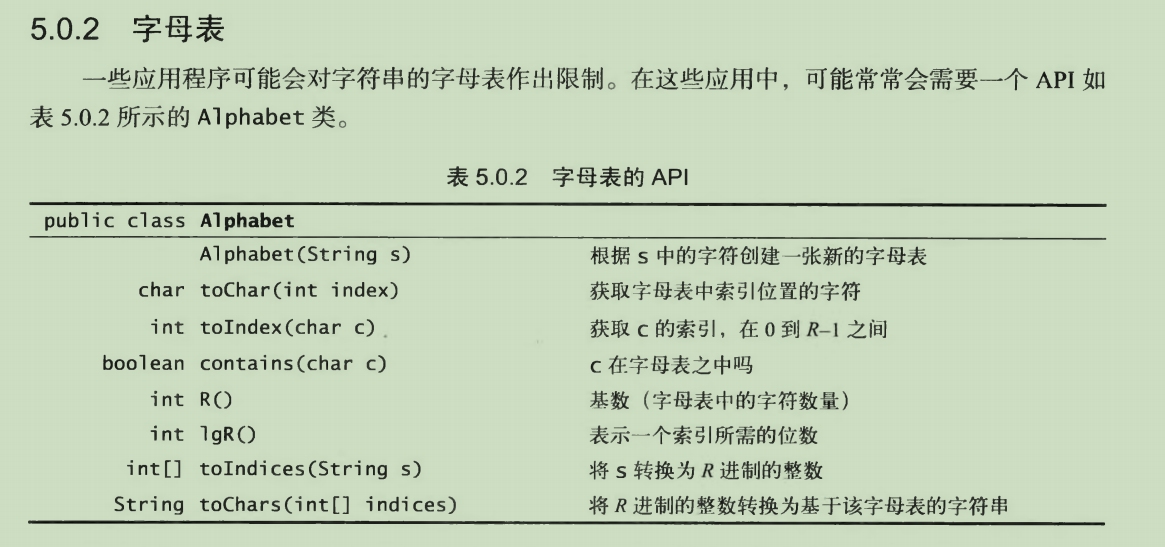
**字符串**

**目录**

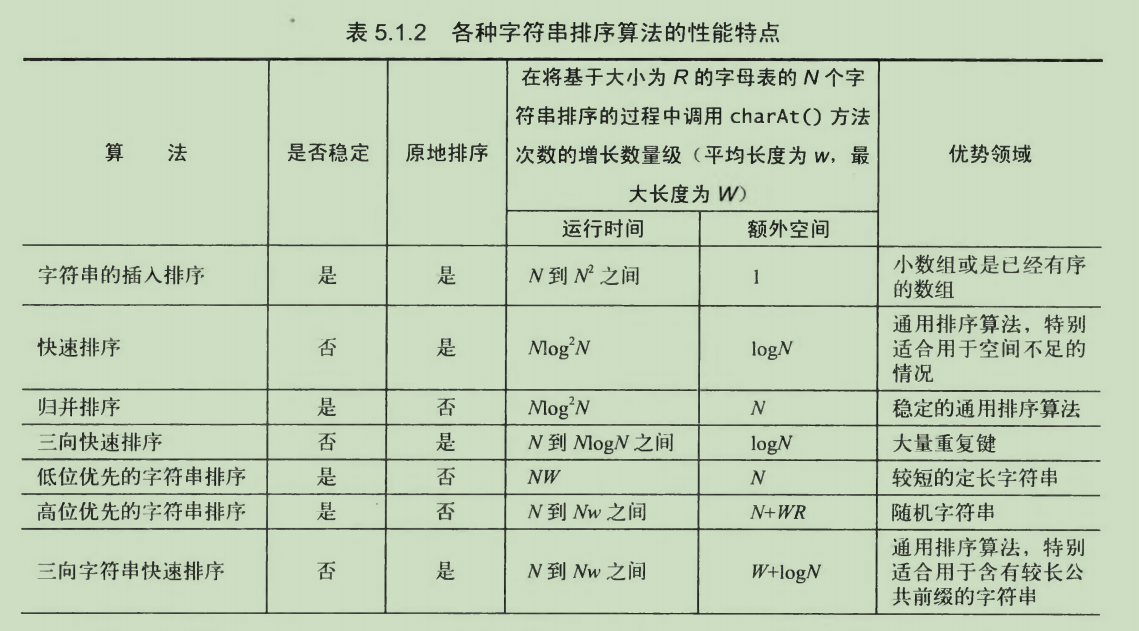




**基础**



**总结性能**

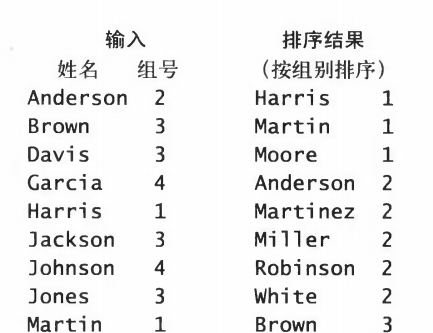




1.字符串排序

**No.1键索引计数法**

对于小整数键排序非常有效却常常被忽略的排序方法。（下图的数据分类）

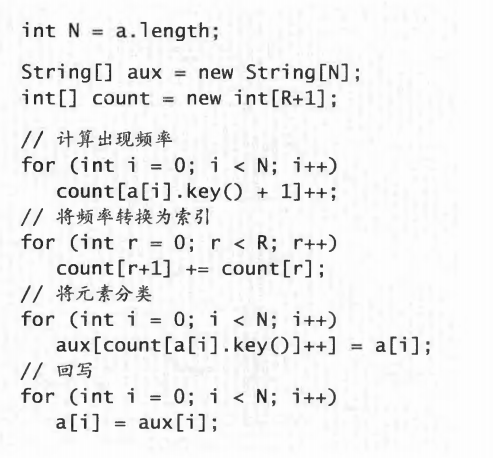


**思路**

先统计每组的个数，然后把C组的前面所以组的个数加起来存放，这是为了在排序数组中，知道C组的起始位置。

（这里有一个小技巧方便我们存放该组前面所有组的个数，就是我们对于 上面图片姓名 Andersong 这个第二组的人 在count中加一存放到count【3】中，count【3】代表第二组的人，对于count【2】来说，存放的就是第2组之前的所有人，因为count【3】存放的是第二组的人）

**代码展示**



**意义**

意味着键索引计数法突破了 MogTV的排序算法运行时间下限

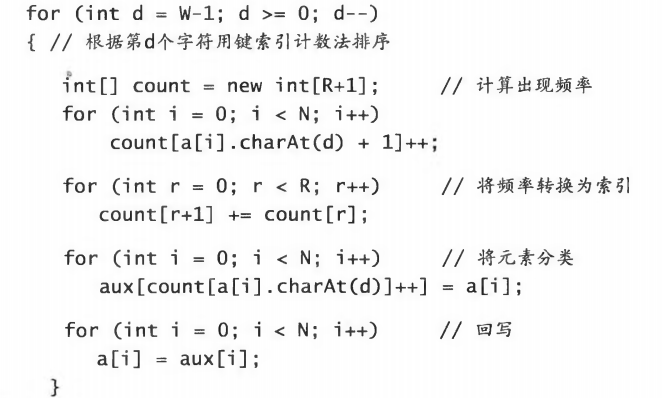
**No.2 低位优先算法（从右向左字符串长度一样）**

**思路**

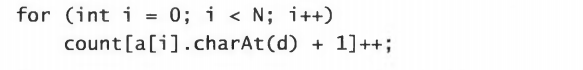
针对ip地址这种字符长度一样的字符串，核心思路基于No.1键索引计数法。

如果字符串的长度均为W那就从右向左以每个位置的字符作为键，用键索引计数法将字符串排序F遍。

**代码展示**



**更改部分**



**No.3 高位优先算法（这个算法主要针对字符串长度不一的问题）**

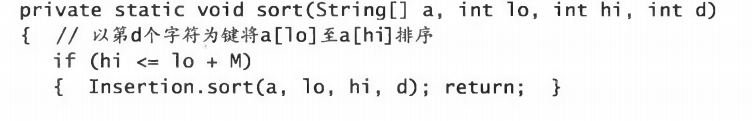
**算法存在问题**

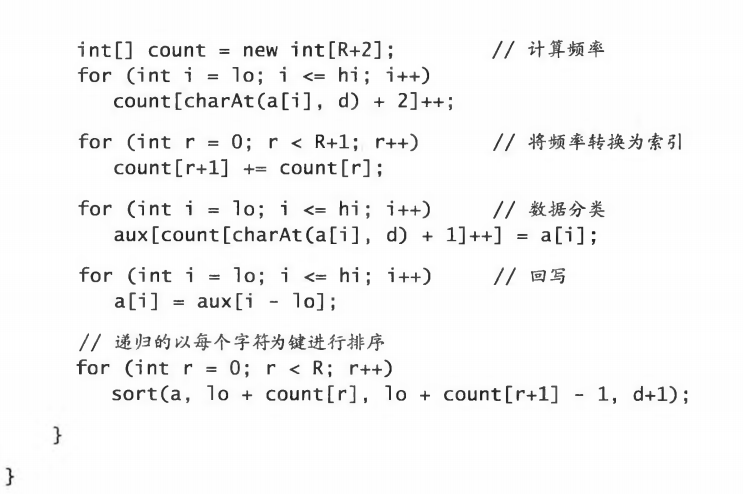
键索 代表字符的种类，例如 a ，b ，c等等，我们算法是根据字符串前缀字母种类来将字符串进行归类

1. 从左往右，不断执行No.1键索引计数法，如果给出的字符串的前缀都相似，这样算法的效率极其缓慢，

2. 这个算法的键索也是一个问题，ASCII字 符 串 （只有256 个键索)，Unicode时 （有655 36 个键索)，这样将耗费大量空间，同时许多使用排序但考虑不周的程序在从ASCII切换到Unicode后运行时间从几分钟暴涨到几个小时。

**代码展示**

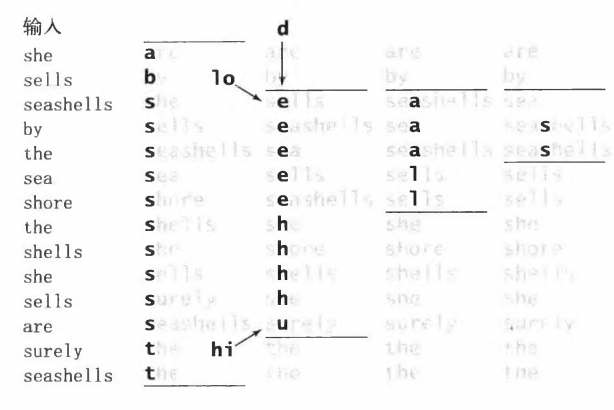




**解释**

count[ charAt(a[i ] ,d) +2 ]++ ，每次调用sort 我们都会传进去 d，d代表现在进行到第几个字符位置了，如果这个字符串在这个位置上已经结束，那么count[ charAt(a[i ] ,d) +2 ]++ 中的 charAt(a[i ] ,d) 返回0，代表这个没有意义，不在考虑范围内。其实本质上还是No.1键索引计数法的多次执行。只不过是从左往右。

lo和hi的关系。

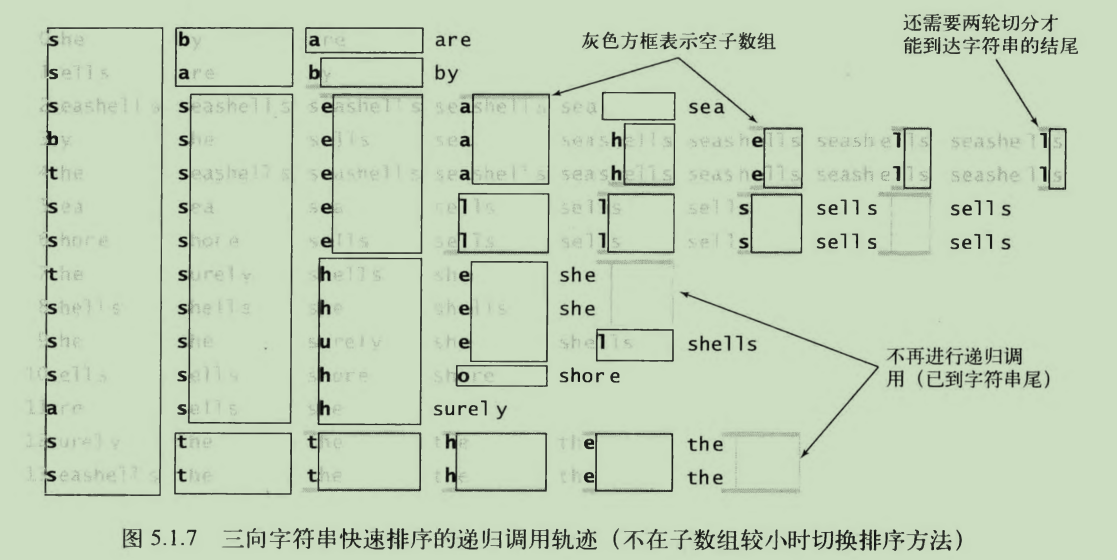


**三向字符串快速排序**

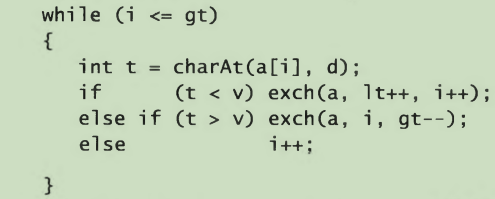
**核心思路**

结合快速排序的思路，从每个单词首字母开始，进行快速排序，以第一个字符串的首字母为中间为中间值然后拆分出三个数组，然后再对三个数组以此递归。

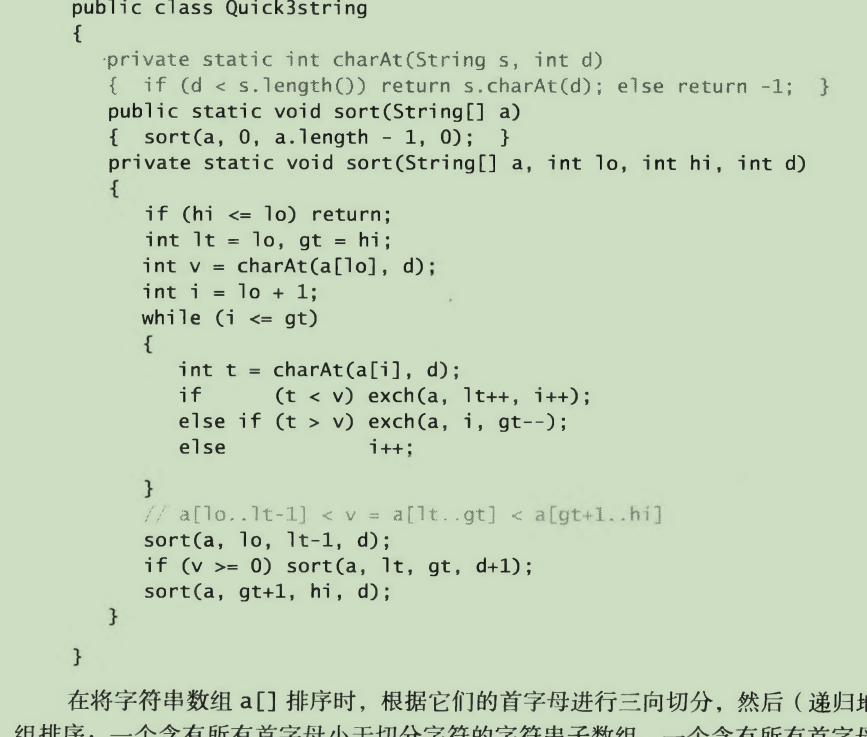
**思路演示）**



**核心代码（如何拆分出三个数组，然后索引是如何在数组中移动的）**



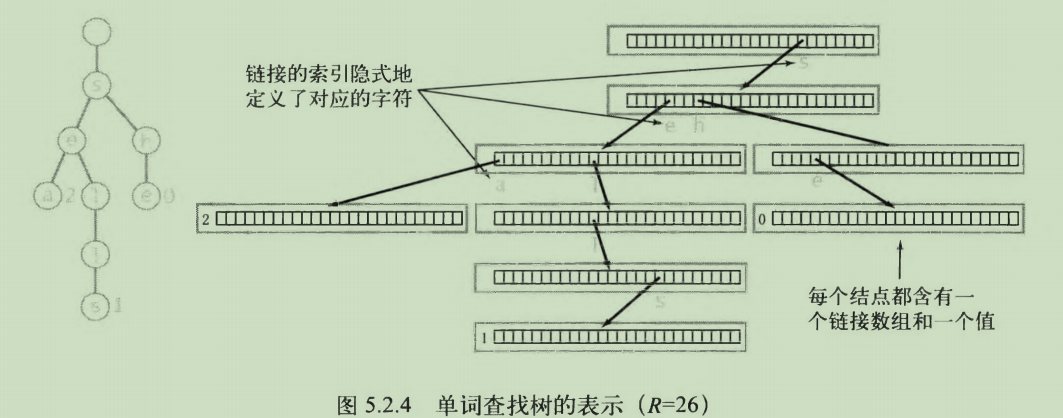
**全部代码**



**2. 单词查找树**

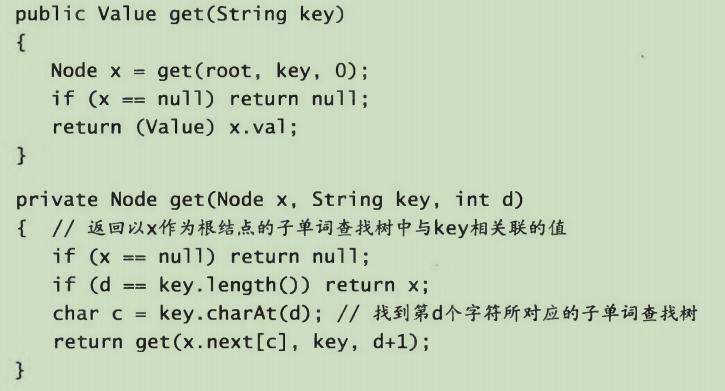
**（不要使用大量长键，如果你能负担得起他的空间，性能是无敌的）**





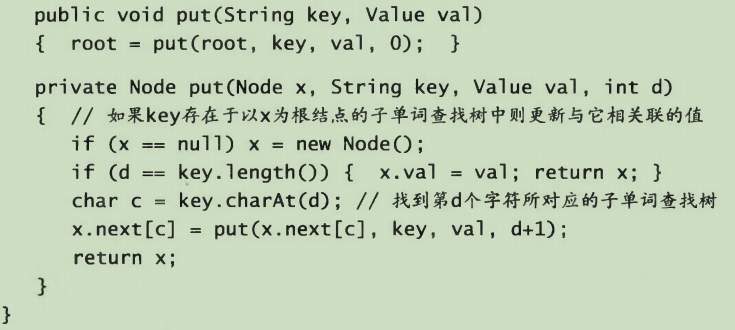
实际存储是每个单词代表一个数组来存储，每个数组前面的数组代表结束同时也表示这段字符串背后代表的数字。

**查找思路（依照上面树的结构，一个一个查找）**



1. d代表现在查找到第几个了，不断递归查找

**存放思路（与查找思路类似）、**

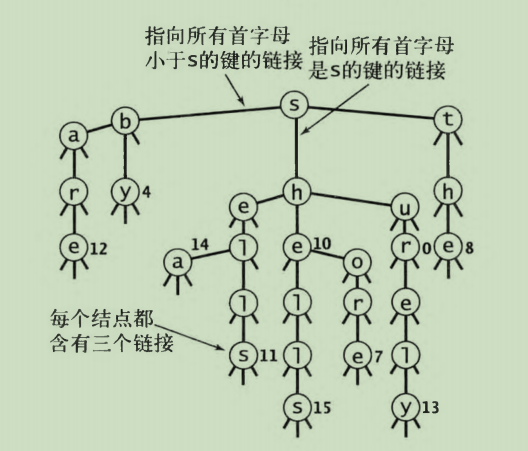


**删除思路**



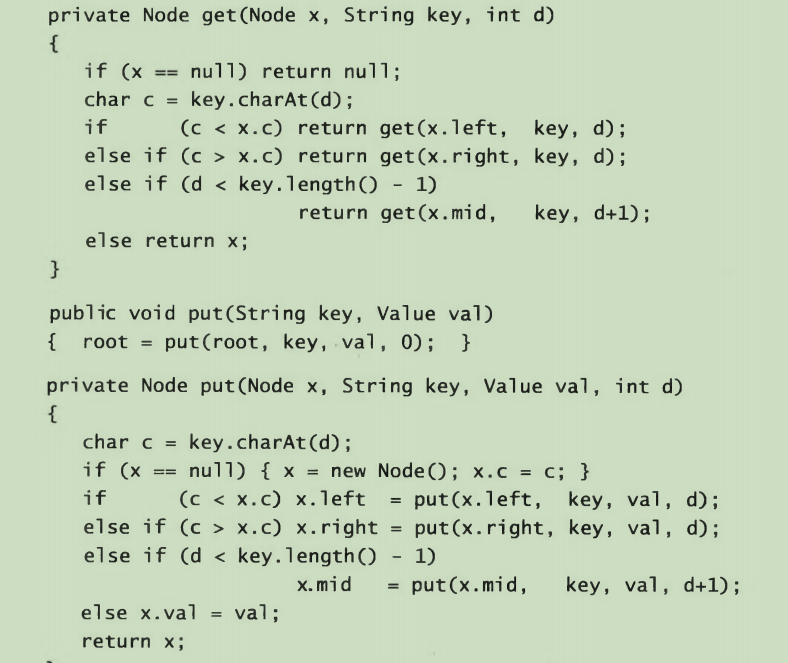
**三向单词查找树**

**（适用于任何情况，不像单词查找树大量空间浪费，但是性能下降。）**



这个查找树与前面单词查找树一样，存储 查找思路一样。但是他结合二叉树思想，所以与前面单词查找树不一样的是，每个节点仅仅包含三个，大于 小于 等于，这样的操作可以大幅度减少单词查找树的空间浪费。

**代码展示**



**3. 子字符串查找**

**性能**



**暴力子字符查找方法**

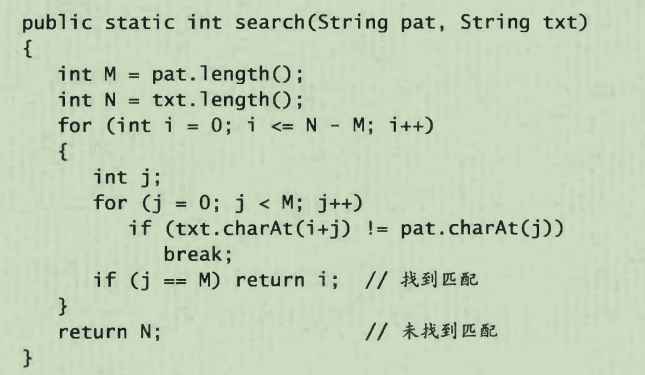
子字符串查找有一个简单而使用广泛的暴力算法。虽然它在最坏情况下的运行时间与MN（M代表文本字节大小 N代表匹配子字符串字节大小）成正比，但是在处理许多应用程序中的字符串时（除了一些变态的情况之外），它的实际运行时间一般M+N与 成 正 比

**算法思路**

这段程序使用了一个指针 i 跟踪文本，一个指针j 跟踪模式（代指匹配的子字符串）。对于每个i ，代码首先将j重置为0 并不断将它增大，直至找到了一个不匹配的字符或是模式结束（j==M)为止。

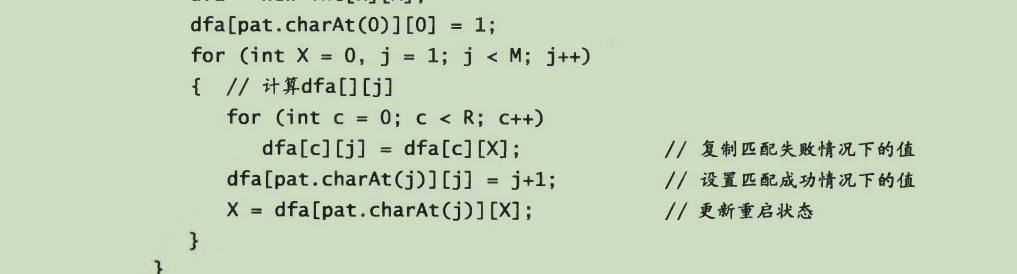
一旦发生（j==M)说明已经找到匹配的了，如果没有找到，j会一直被重置为0.

**代码展示**



**Kmp算法**

核心这段代码用于计算如何找出在模式字符串下一个比较的位置，所以说这个是基于已经存在的dfa数组来构造的，dfa如何构造书上没有写出。（就是字符串前后腏最大数 参考下面链接<https://www.cnblogs.com/SYCstudio/p/7194315.html>）



代码理解

dfa[text.charAt(i)][j]是指当文本字符串的字符s[i]与模式字符串的字符p[j]比较后下一次与文本字符串的字符s[i+1]比较的模式字符串的字符位。

当文本字符串的第i位字符与模式字符串的第j位进行匹配时，如果此两位字符匹配，则说明下一步应该为i++ j++之后再比较，也就是说当txt.charAt(i)==pat.charAt(j)时，有dfa[pat.charAt(j)][j]=j+1。

当文本字符串的第i位字符与模式字符串的第j位检测到不匹配时，设txt.charAt(i)==c，c!=pat.charAt(j)，但文本字符串的s[i-j...i-1]这部分与模式字符串的p[0...j-1]这部分是匹配的。这时从文本字符串的i-j位置起已不可能出现匹配字符串，现在已不用管s[i-j]字符，现在的问题是依次输入s[i-j+1...i-1]c后会进入什么状态，由于s[i-j...i-1]这部分与模式字符串的p[0...j-1]这部分是匹配的，也就是说现在的问题是依次输入p[1...j-1]c后会进入什么状态。引入一个状态指示数组X，X[j]是指正确输入p[1...j-1]后进入的状态，输入p[1...j-1]c后进入的状态就是dfa[c][X[j]]（在X[j]状态时输入c），即dfa[c][j]=dfa[c][x[j]]。

而计算X[]数组的方法为递推方法：X[j+1]为正确输入p[1...j]后进入的状态，即正确输入p[1...j-1]p[j]后进入的状态，也就是在X[j]状态时输入p[j]时进入的状态，就是dfa[pat.charAt[j]][X[j]]，即递推公式为：X[j+1]=dfa[pat.charAt[j]][X[j]]，而X[0]手动初始化为0。

由于X[]数组为辅助数据，且为递推的，所以书中仅使用了一个变量X来指示当前的X[j]，我觉得理解起来很不方便，所以这里由数组代替。

希望能帮到你~

代码展示（c++版本）

#include<memory>

#include<vector>

#include<iostream>

#include<string>

using namespace std;

class KMP {

public:

KMP(string pattern);

int search(string text);

private:

shared\_ptr<vector<vector<int>>> dfa;

int R;

int M;

};

KMP::KMP(string pattern) :R(256), M(pattern.size()){

dfa = make\_shared<vector<vector<int>>>(R);

for (int i = 0; i < R; i++) {

(\*dfa)[i].resize(M);

}

(\*dfa)[pattern[0]][0] = 1;

for (int X = 0, j = 1; j <M; j++)

{

for (int i = 0; i < R; i++)

(\*dfa)[i][j] = (\*dfa)[i][X];

(\*dfa)[pattern[j]][j] = j + 1;

X = (\*dfa)[j][X];

}

}

int KMP::search(string text) {

int N = text.size();

int i, j;

for (i = 0, j = 0; i < N&&j < M; i++)

j = (\*dfa)[text[i]][j];

if (j == M)

return i - j;

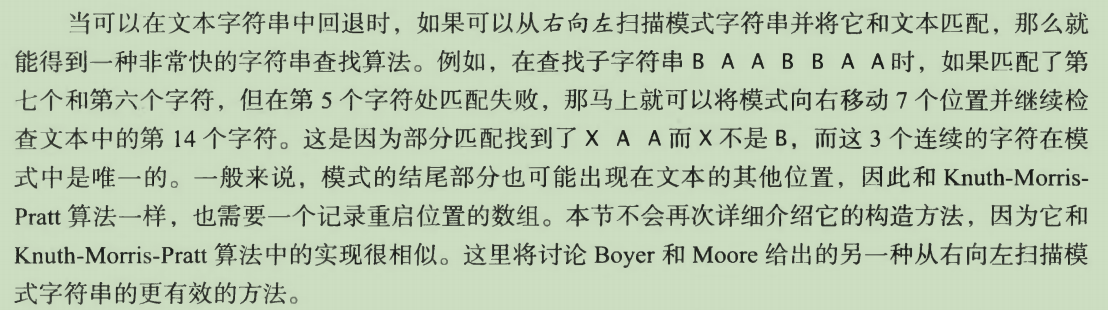
else

return N;

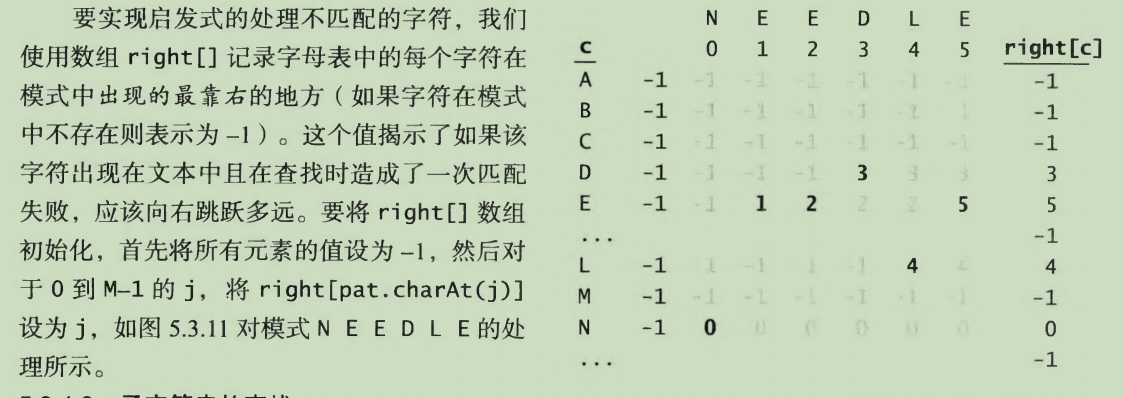
}

**boyer-moore 字符串查找算法**

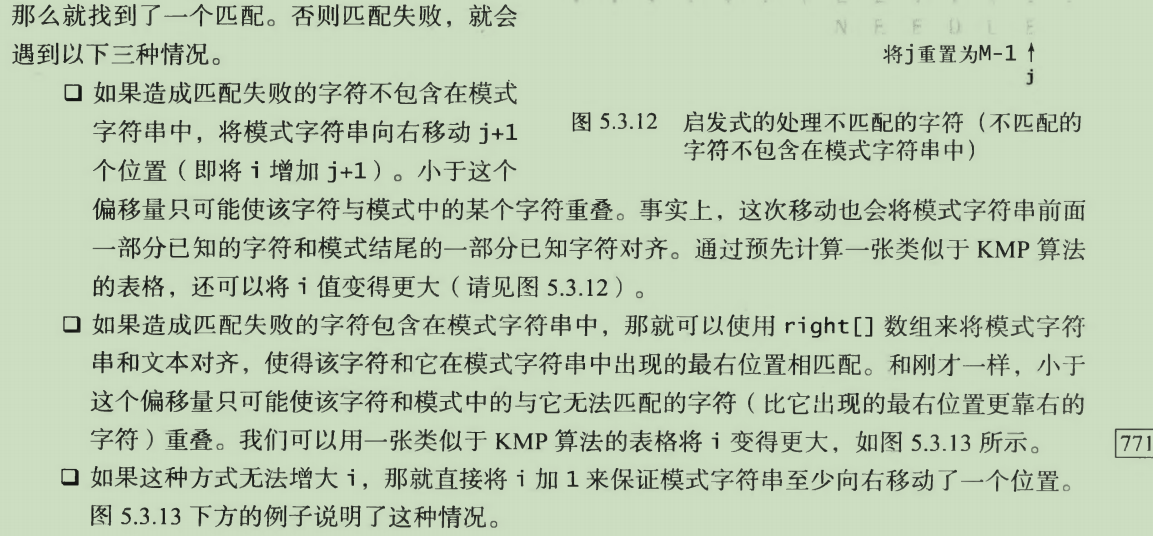
**思路**

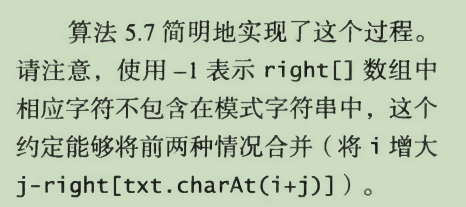


**right 数组初始化**



**字符没有匹配的时候**



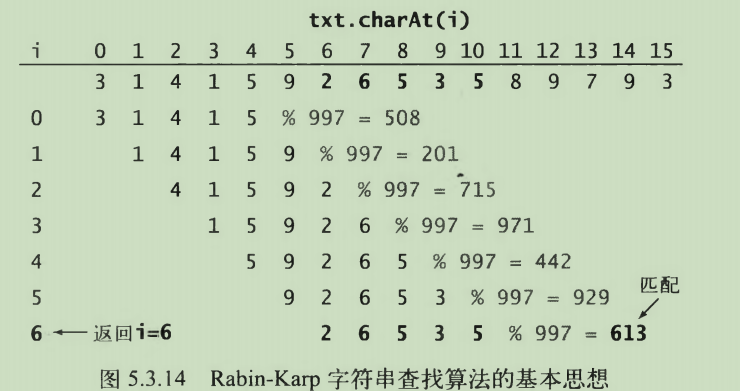


**代码**



**Rabin-Karp指纹字符串查找算法**

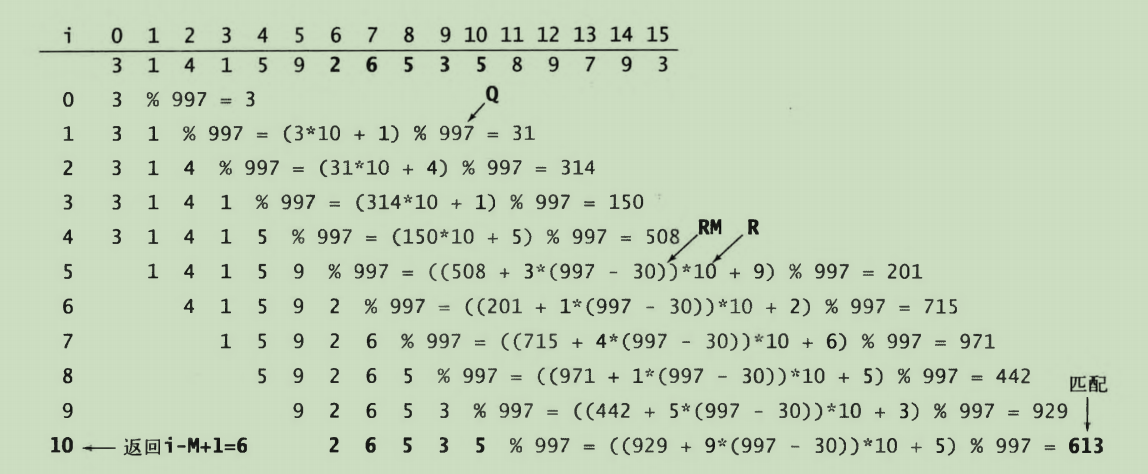
**核心思路**



**关键思路**

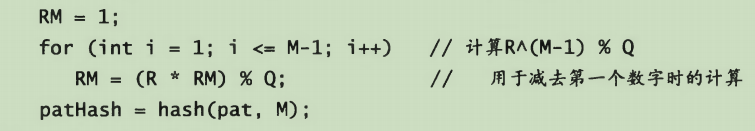
如何快速计算需要文本产生的子字符串的散列表数值，减去子字符串第一个字符，加上后面一个字符，这样十分高效计算了。

**具体计算方法**



我们注意 减去 3的计算 是3\*（997-30）\*10（这里的道理没有弄明白）

Rm的计算方法 997代表Q



**代码**

