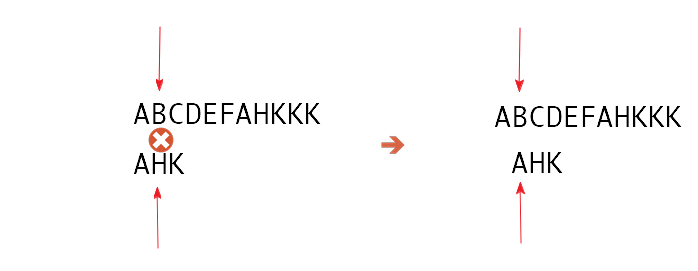
最简单的字符串匹配算法是暴力法，它的基本流程是这样的。

首先让搜索词和字符串对齐，然后从左到右比较上下两组字符：



如果发现两组字符不匹配，那么搜索词就要相对于字符串向前移动一位,然后继续按照上面的算法匹配下去。



它基本的框架如下：

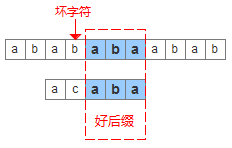
|  |
| --- |
| int m = strlen(pattern), n = strlen(text);  int j = 0, i = 0;  while (i <= n - m) {  for (j = 0; j < m && pattern[j] == text[i + j]; ++j);  if (j == m) {  Match; /\* 匹配成功 \*/  break;  }  else  ++i; /\* 搜素串相对于字符串向前移动一个字符 \*/  } |

而BM算法和暴力法稍微有一点不同，首先，它是从右到左进行匹配，其次，匹配时发现搜索词和文本串不匹配时，搜索词相对于文本串移动的位数不仅仅是1，在很多时候，位数远远超过1，这也是BM比暴力法快的关键所在。

BM算法的框架如下：

|  |
| --- |
| int n = strlen(text), m = strlen(pattern);  int j = 0, i = 0;  while (i <= n - m) {  /\* 从右到左进行匹配 \*/  for (j = m - 1; j >= 0 && pattern[j] == text[i + j]; --j);  if (j < 0) {  Match;  break;  }  else  i += BM(); /\* 每次模式串移动的距离不仅仅是1 \*/  } |

为了实现更快移动模式串，BM算法定义了两个规则，**好后缀规则**和**坏字符规则**，如下图可以清晰的看出他们的含义。利用好后缀和坏字符可以大大加快模式串的移动距离，不是简单的++j，而是**j+=max (shift(好后缀), shift(坏字符))**



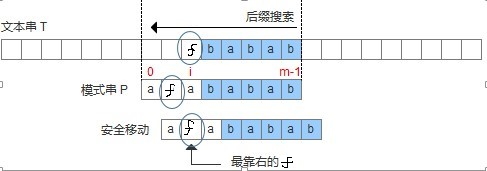
1. 计算坏字符规则表

先来看如何根据坏字符来移动模式串，**shift(坏字符)**分为两种情况：

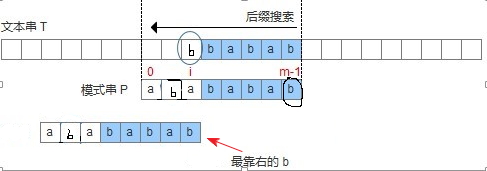
1. **坏字符没出现在字符串中**，这时可以把模式串移动到坏字符的下一个字符，继续比较，如下图：



2.**坏字符出现在模式串中**，这时可以把模式串中出现在最右侧的坏字符和母串的坏字符对齐，如下图：

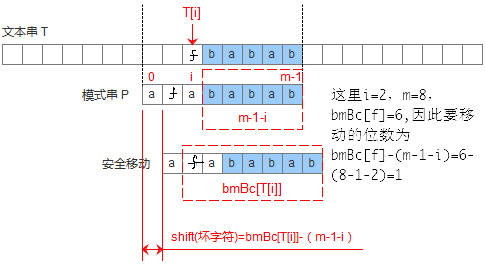


也许你会有这样的疑问，会不会出现模式串后退的情况？比如说下图：

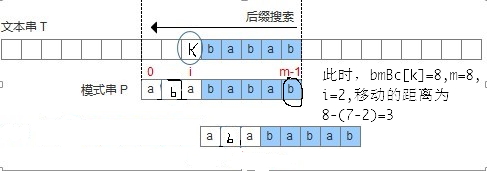


如果BM算法仅有坏字符规则的话，是会出现这种情况的，但是BM算法同时还有好后缀规则，并且好字符规则的各种情况的移动距离都是大于0的,所以模式串不会后退。

为了用代码来描述上述的两种情况，设计一个数组**bmBc['k']**，表示**坏字符‘k’在模式串中出现的位置距离模式串末尾的最大长度**，那么当遇到坏字符的时候，模式串可以移动的位数为： **shift(坏字符) = bmBc[T[i]]-(m-1-i)**。如下图, **m-1-i表示已经成功匹配的字符的长度，T[i]表示文本串(字符串)的第i个字符**。



上面的公式对模式串中不存在坏字符的情况同样有效：



写成代码如下：

|  |
| --- |
| /\*\*  \* @brief 预处理，计算模式串中每个字符最靠右的位置距离模式串末尾的长度.  \*  \* @param[in] pattern 模式串  \* @param[in] m 模式串的长度  \* @param[out] bmBc 用于记录每个字符在模式串中最靠右的位置距离模式串末尾的长度.  \*/  void preBmBc(char \*pattern, int m, int bmBc[])  {  int i;  for (i = 0; i < 256; i++) { /\* 英文的ASCII码最多256个字符 \*/  bmBc[i] = m;  }  for (i = 0; i < m - 1; i++) {  bmBc[pattern[i]] = m - 1 - i; /\* pattern[i]是模式串的第i个字符 \*/  }  } |

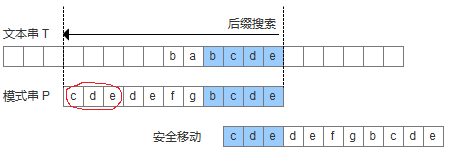
1. 计算好后缀规则表

再来看如何根据好后缀规则移动模式串，**shift（好后缀）**分为三种情况：

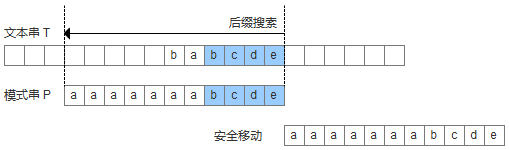
模式串中有子串匹配上好后缀，此时移动模式串，让该子串和好后缀对齐即可，如果超过一个子串匹配上好后缀，则选择最靠左边的子串对齐。



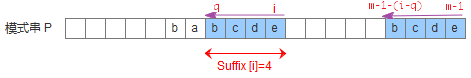
模式串中没有子串匹配上后后缀，此时需要寻找模式串的一个最长前缀，并让该前缀等于好后缀的后缀，寻找到该前缀后，让该前缀和好后缀对齐即可。



模式串中没有子串匹配上好后缀，并且在模式串中找不到最长前缀，让该前缀等于好后缀的后缀。此时，直接移动模式到好后缀的下一个字符。



为了实现好后缀规则，需要定义一个数组**suffix[]**，其中**suffix[i] = s** 表示从后往前看，以位置**i**为起点的子串与模式串后缀所能匹配的字符最大长度，如下图所示，用公式可以描述：满足**P[i-s, i] == P[m-s, m]**的最大长度**s**。

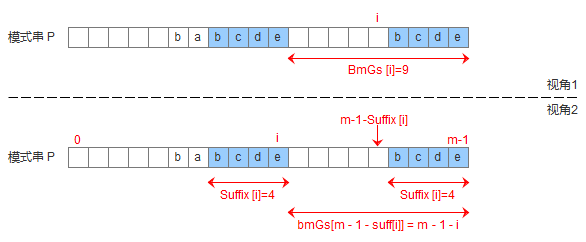


写成代码如下：

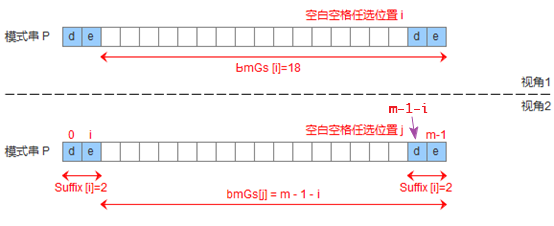
|  |
| --- |
| /\*\*  \* @brief 预处理，计算从模式串位置i开始的子串，从后往前，和后缀相匹配的最大长度.  \*  \* @param[in] pattern 模式字符串  \* @param[in] m 模式串的长度  \* @param[out] suff 用于记录结果的数组  \*/  void suffixes(char \*pattern, int m, int \*suff)  {  suff[m - 1] = m;  for (int i = m - 2; i >= 0; --i) {  int q = i;  /\* pattern[q]属于子串部分，pattern[m-1-i+q]属于后缀部分 \*/  while (q >= 0 && pattern[q] == pattern[m - 1 - i + q])  --q;  suff[i] = i - q;  }  } |

有了**suffix**数组，就可以定义**bmGs[]**数组，**bmGs[i]** 表示遇到好后缀时，模式串应该移动的距离，其中**i表示好后缀前面一个字符的位置（也就是坏字符的位置）**，构建**bmGs**数组分为三种情况，分别对应上述的移动模式串的三种情况。

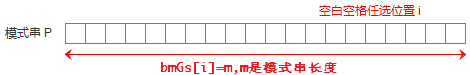
**1.模式串中有子串匹配上好后缀：**



**2.模式串中没有子串匹配上好后缀，但找到一个最大前缀:**



**3.模式串中没有子串匹配上好后缀，也找不到一个最大前缀:**



|  |
| --- |
| /\*\*  \* @brief 预处理,构建好后缀规则表.  \*  \* @param[in] pattern 模式串  \* @param[in] m 模式串的长度  \* @param[out] bmGs bmGs[i]表示如果在模式串的i位置遭遇不匹配,那么采用好后缀规则,模式串应当相对于文本串向后移动的位数.  \*/  void preBmGs(char \*x, int m, int bmGs[])  {  int i, j, suff[256];  suffixes(x, m, suff);  for (i = 0; i < m; ++i) /\* 既无子串匹配好后缀，也没有最大前缀 \*/  bmGs[i] = m;  j = 0;  for (i = m - 1; i >= 0; --i)  /\*  当**suff[i]==i+1**,根据**suff**的定义,有**pattern[i+1-suff[i]~i]==pattern[m-1-suff[i]~m-1]**,由**suff[i]==i+1**,有**pattern[0~i]==pattern[m-1-(i+1)~m-1]**,即存在长度为**i+1**的前缀和后缀匹配.这里特别要注意一点,就是这里的**i**是从后往前遍历的,假定我们有长度为**5**的前缀**aaaaa**和长度为**5**的后缀相匹配,那么这一轮修改完**bmGs[j]**后,**--i**,我们此时还有长度为**4**的前缀**aaaa**和后缀相匹配.所以你看到了下面的条件**bmGs[j]==m**,这个条件是为了保证,我们只会修改一次**bmGs[j]**.  至于**i**从后往前遍历的原因,是这样的,由于我们的目的是获得精确的**bmGs[i]**,故若一个字符同时符合上述三种情况中的几种,那么我们应当选取其中最小的**bmGs[i]**,因为**步子越小越谨慎,步子大了可能会漏掉某些匹配项**.比如当模式串中既有子串可以匹配上好后缀,又有前缀可以匹配好后缀,那么此时我们应该选择让子串对齐好后缀,也就是取**bmGs[i]**较小的那种情况.每次修改**bmGs[i]**都应该使其变小.**i**从后向前遍历的话,**i**越来越小,而**m-1-i**是越来越大,所以一旦修改过一次**bmGs[j]**之后,后面都不允许被修改,因为它们都会使得**bmGs[j]**越来越大.  \*/  if (suff[i] == i + 1)  for (; j < m - 1 - i; ++j)  if (bmGs[j] == m) /\* 保证只会修改一次 \*/  bmGs[j] = m - 1 - i;  /\*  这里对应第一种情况,有可能存在多个子串和好后缀相匹配,正如前面提到的,我们要选择步子迈得比较小的那种,显然,**i**越大,**m-1-i**就越小,所以**i**从小到大进行遍历.下面是一个典型的例子:  **0 1 2 3 4 5 6 7 8 9 10 11 12 13**  **+--+--+--+--+--+--+--+--+--+--+--+--+--+--+**  **|a |b | | |a |b | | | |a |b | |a |b |**  **+--+--+--+--+--+--+--+--+--+--+--+--+--+--+**  假定**suff[1] == suff[5] == suff[10] == 2**,我们应当如何选择**bmGs[11]**,根据**bmGs[m-1-suff[i]]=m-1-i**,则:  **m=14,i=1 时,bmGs[14-1-2]=bmGs[11]=14-1-1=12;**  **m=14,i=5 时,bmGs[14-1-2]=bmGs[11]=14-1-5= 8;**  **m=14,i=10时,bmGs[14-1-2]=bmGs[11]=14-1-10=3;**  我们看得到,自然**bmGs[11]**要选择**3**.  这里还有一点需要注意,那就是倘若**suff[i]==0**,则有**bmGs[m-1]=m-1-i**,这里实际表示的含义是,倘若模式串第**i**个字符和对应的文本串不匹配,而且**suff[i]==0**,那么**要将i指向的模式串字符移动到当前模式串末尾的位置**,**i**最大为**m-2**,则**m-1-(m-2)=1**,也就是说**bmGs[m-1]**至少为**1**,**事实上,如果最后一个字符都和模式串不匹配,根据坏字符规则,直接移动m个位置就可以了**.  **i**从小到大遍历,所以**bmGs[m-1]**一直被这样的位置**i**更新,**suff[i]==0**,遍历完成后,**bmGs[m-1]**的值为**m-1-k**,位置**k**在模式串的所有满足**suff[i]==0**的位置中处在最右侧.**(i=0, 1, ..., m-1)**  举个栗子:  **0 1 2 3**  **文本串 --> a b c e x x x x**  **模式串 --> a b c d**  **i == 0 --> a b c d**  **i == 1 --> a b c d**  **i == 2 --> a b c d**  **suff[3]=4, suff[2]==suff[1]==suff[0]==0**  **bmGs[0]==bmGs[1]==bmGs[2]==4**  执行下面的代码,有:  **i = 0 i = 1 i = 2**  **bmGs[3]==3 -> bmGs[3]==2 -> bmGs[3] = 1**  \*/  for (i = 0; i <= m - 2; ++i) /\* 存在子串匹配好后缀 \*/  bmGs[m - 1 - suff[i]] = m - 1 - i;  } |