第四章 串

机电工程与自动化学院 L栋301

任卫红 助理教授

renweihong@hit.edu.cn

http://faculty.hitsz.edu.cn/renweihong

4.1串

第一节　字符串

一、字符串（string）

- 字符串是n(≥0)个字符的有限序列，记作：

  S ＝ '$a_1a_2a_3...a_n$'

- 其中，S 是串名字

  '$a_1a_2a_3...a_n$'是串值

  a_i 是串中字符

  n 是串的长度(串中字符的个数)

- 例如，S = "Harbin Institute of Technology"

第一节　字符串

二、字符串术语

- **空串**：不含任何字符的串，串长度=0
- **空格串**：仅由一个或多个空格组成的串
- **子串**：由串中任意个连续的字符组成的子序列。
- **主串**：包含子串的串。
- 如：A=“Harbin Institute of Technology”
B=“Harbin” A为**主串**，B为**子串**。

第一节　字符串

二、字符串术语

- **位置**：字符在主串中的序号。**子串在主串中的位置**以子串**第一个字符**在主串中的位置来表示
- **串相等的条件**：当两个串的长度相等且各个对应位置的字符都相等时才相等。
- **模式匹配**：确定子串在主串中**首次**出现的位置的运算

第一节　字符串

三、字符串与线性表的关系

串的逻辑结构和线性表极为相似

- 它们都是线性结构
- 串中的每个字符都仅有一个前驱和一个后继

第一节　字符串

三、字符串与线性表的关系（内容受限）

串与线性表又有区别，主要表现为：

- 串的数据对象约定是字符集
- 在线性表的基本操作中，以“单个元素”作为操作对象
- 在串的基本操作中，通常以“串的整体”作为操作对象，如：在串中查找某个子串、在串的某个位置上插入一个子串等。

第一节 字符串

四、字符串的操作

13种操作中的最小操作子集:

最小操作集:

这些操作不可能利用其他串操作来实现,反之,其他串操作(除串清除ClearString和串销毁DestroyString外)可在这个最小操作子集上实现。

第一节　字符串

四、字符串的操作

13种操作中的最小操作子集（五种）：	C语言中的字符串操作中的最小操作集：
1. 串赋值StrAssign	1. strcpy 串拷贝
2. 串比较StrCompare	2. strcmp 串比较
3. 求串长StrLength	3. strlen 求串长
4. 串联接Concat	4. strcat 串联接
5. 求子串SubString	5. strchr 定位函数

四、字符串的操作（定位）

例如，可利用串比较（2）、求串长（3）和求子串（1）等操作实现定位函数Index(S, T, pos)。

算法的基本思想为：

StrCompare(SubString(S, i, StrLength(T)),T) = ? 0

即依次取出S的长度与T的长度相等的子串并与T进行比较。直到相等为止

第一节　字符串

四、字符串的操作（定位）

串匹配(查找)的定义：INDEX（S，T，pos）

初始条件：串S和T存在，T是非空串，
$1 \leqslant pos \leqslant StrLength(S)$。

操作结果：若主串S中存在和串T值相同的子串返回它在主串S中第pos个字符之后第一次出现的位置；否则函数值为0。

C++中string容器

string 模板类的定义在<string>头文件中。

定义string 对象的示例代码如下：

string str;

string str = “abcde”；

sring 的基本操作有：

赋值，如例：str.assign(6, ‘s’)；

交换，如例：str.swap(str1)；

顶部插入：str.push_back(‘s’)；//str=abc，str=abcs

删除：str.pop_back()；//str=abc，str=ab

长度，如例：str.length()；

插入，str.insert(2, “123”)；//str=abc，str=ab123c

子串查找,查找字符，大小写转换，链接。。。

4.2 串的表示和实现

第二节　串的表示和实现

一、定长顺序存储表示（静态存储分配）

- 用一组地址连续的存储单元存储字符序列
- 如C语言中的字符串定义(以“\0”为串结束标志)

```
char Str[MAXSTRLEN+1];
```

- 定义了长度为MAXSTRLEN字符存储空间
- 字符串长度可以是小于MAXSTRLEN的任何值（最长串长度有限制，多余部分将被截断）

第二节　串的表示和实现

二、串长的两种表示（隐含）

一般可使用一个不会出现在串中的特殊字符在串值的尾部来表示串的结束。

优点:便于系统自动实现

缺点:不利于某些操作(如合并).

例如，C语言中以字符‘\0’表示串值的终结，这就是为什么在上述定义中，串空间最大值maxstrlen为256，但最多只能存放255个字符的原因。

第二节　串的表示和实现

二、串长的两种表示（显式）

若不设终结符，可用一个整数来表示串的长度，那么该长度减1的位置就是串值的最后一个字符的位置（下标）。

此时顺序串的类型定义和顺序表类似：

```
class sstring{
    char ch[maxstrlen];
    int length;
};
```

优点:便于在算法中用长度参数控制循环过程

第二节　串的表示和实现

二、堆分配存储表示

- 在程序执行过程中，**动态分配（malloc）**一组地址连续的存储单元存储字符序列
- 在C++语言中，由new和delete动态分配与回收的存储空间称为堆
- 堆分配存储结构的串**既有**顺序存储结构的特点，处理方便,操作中对串长**又没有**限制,更显灵活

第二节　串的表示和实现

二、堆分配存储表示

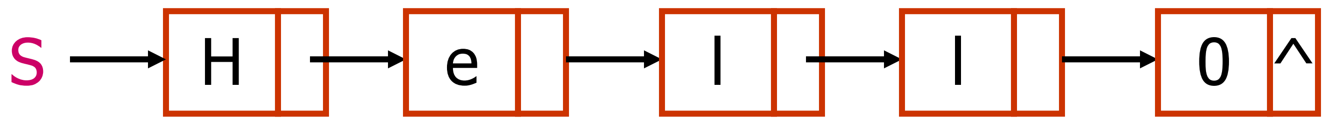
```
class HString{
   char *ch;
       // 若是非空串，则按串长分配存储区，
       //  否则ch为NULL
   int  length;   // 串长度
};           //其实质为动态数组。
```

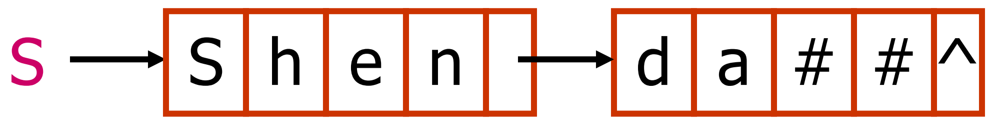
第二节　串的表示和实现

二、链存储表示

- 采用链表方式存储串值
- 每个结点中，可以存放一个字符，也可以存放多个字符

$$\textbf{存储密度} = \frac{\text{数据元素所占存储位}}{\text{实际分配的存储位}}$$





4.3 串的匹配算法

第三节　串的匹配算法[了解]

一、求子串位置函数Index()

- 子串的定位操作通常称做串的模式匹配
- 算法（穷举法，朴素算法，BF(Brute-Force)算法）：

  从主串的指定位置开始，将主串与模式（要查找的子串）的第一个字符比较，

  1. 若相等，则继续逐个比较后续字符；

  2. 若不等，从主串的下一个字符起再重新和模式的字符比较

朴素的模式匹配

第1趟

i=2

S a b b a b a

T a b a

j=2

第2趟

i=1

S a b b a b a

T a b a

j=0

第3趟

i=2

S a b b a b a

T a b a

j=0

第4趟

i=6

S a b b a b a

T a b a

j=3

朴素算法

以显示串长的定长的顺序串类型作为存储结构，串匹配算法实现为：

```
int Index(SString S, SString T, int pos) {
   // 返回子串T在主串S中第pos个字符之后的位置。若不存在，
    // 则函数值为-1。其中，T非空，0≤pos≤S.length-1)。
    i = pos;   j = 0;
    while (i <= S.length-1 && j <= T.length-1) {
      if (S.ch[i] == T.ch[j]) { ++i;  ++j; }   // 继续比较后继字符
      else { i = i-j+1;   j = 0; }     // 指针后退重新开始匹配
    }
   if (j ==T.length)  return  i-T.length;
   else return -1;
} // Index
```

时间复杂度?

第三节　串的匹配算法

一、求子串位置函数Index()

- 在**最好的情况**下，除比较成功的位置外，其余位置仅需比较一次（模式第一个字符），其时间复杂度为：

  O(n+m)　（n，m分别为主串和模式的长度）

- 但在**最坏的情况**下，如模式为‘00000001’，主串为‘00000000000000000000000000000000001’，则每次模式的前7个0都要与主串逐一比较，因此，其时间复杂度为：

  O(n*m)

第三节　串的匹配算法

二、KMP算法（时间复杂度n+m）

- 是index函数的一种改进，由D. E. Knuth(克努特)—J. H. Morris(莫里斯)—V. R. Pratt(普拉特)发现
- 当一趟匹配过程中出现字符比较不等(失配)时

  1. 不需回溯i指针

  2. 利用已经得到的“部分匹配”的结果

  3. 将模式向右“滑动”尽可能远的一段距(next[j])后，继续进行比较

第三节　串的匹配算法

二、KMP算法(举例)

- 假设主串ababcabcacbab，模式abcac，改进算法的匹配过程如下

第一趟匹配

```
    ↓ i=2
a b a b c a b c a c b a b
a b c a c
    ↑ j=2
```

第二趟匹配

```
    ↓ i=2
a b a b c a b c a c b a b
    a b c a c
    ↑ j=0
```

第三趟匹配

```
            ↓ i=6------9
a b a b c a b c a c b a b
          a b c a c
            ↑ j=1
```

第三节　串的匹配算法

二、KMP算法

- 假设主串为‘$s_1s_2s_3\ldots s_n$’, 模式串为‘$p_1p_2p_3\ldots p_m$’
- 若主串中第i个字符与模式串中第j个字符“失配”（$s_i != p_j$）, 说明，模式串中前面j-1个字符与主串中对应位置的字符相等，即：

$$‘p_1p_2\ldots p_{j-k}p_{j-k+1}p_{j-k+2}\ldots p_{j-1}’$$

$$= \quad ‘s_{i-j+1}s_{i-j+2}\ldots s_{i-k}s_{i-k+1}s_{i-k+2}\ldots s_{i-1}’$$

第三节 串的匹配算法

二、KMP算法

- 现假定主串中第i个字符需要与模式串中第k(k<j)个字符比较, 则说明，模式串中前k-1个字符与主串中对应位置的字符相等，即有以下关系成立：

$$‘p_1p_2\ldots p_{k-1}’ = ‘s_{i-k+1}s_{i-k+2}\ldots s_{i-1}’$$

第三节　串的匹配算法

二、KMP算法

- 比较：

$$'p_1p_2\cdots p_{j-k}p_{j-k+1}p_{j-k+2}\cdots p_{j-1}' = 's_{i-j+1}s_{i-j+2}\cdots s_{i-k}s_{i-k+1}s_{i-k+2}\cdots s_{i-1}'$$

$$'p_1p_2\ldots p_{k-1}' = 's_{i-k+1}s_{i-k+2}\cdots s_{i-1}'$$

- 由以上两式，有下式成立：

$$'p_1p_2\ldots p_{k-1}' = 'p_{j-k+1}p_{j-k+2}\cdots p_{j-1}'$$

	s_{i-j+1}	…	s_{i-k+1}	…	s_{i-2}	s_{i-1}	s_i	…
	p_1	…	p_{j-k+1}	…	p_{j-2}	p_{j-2}	p_j	…
			p_1	…	p_{k-2}	p_{k-1}	p_k	…

表中黑色字体表示对应列字符相等，红色表示$s_i \neq p_j$, s_i与p_k进行比较。

$$'p_1p_2\ldots p_{k-1}' = 'p_{j-k+1}p_{j-k+2}\ldots p_{j-1}'$$

$2 \leqslant k < j$

二、KMP算法

$$‘p_1p_2\ldots p_{k-1}’ = ‘p_{j-k+1}p_{j-k+2}\ldots p_{j-1}’$$

- 上式是只依赖于模式串的关系式
- 上式说明，在主串中第i个字符与模式串中第j个字符“失配”时，仅需与模式串中的第k个字符再开始比较（主串不需要回溯）

第三节 串的匹配算法

二、KMP算法

- 或者换言之，在模式串中第j个字符“失配”时，模式串第k个字符再同主串中对应的失配位置（i）的字符继续进行比较

$$‘p_1p_2\ldots p_{k-1}’ = ‘p_{j-k+1}p_{j-k+2}\ldots p_{j-1}’$$

- k值可以在做串的匹配之前求出
- 本质上是查找相等的真前缀和真后缀
- 一般用next函数求取k值

第三节　串的匹配算法

二、KMP算法(next函数，下标从1开始)

- next函数定义为：

$$
next[j] = \begin{cases} 0 & \text{当} j=1\text{时} \\ \max\{k \mid 0<k<j \text{且} `p_1\ldots p_{k-1}\text{'}=`p_{j-k+1}\ldots p_{j-1}\text{'}\} \\ 1 & \text{其它情况} \end{cases}
$$

- 如k=2，则$p_1=p_{j-1}$　(有1个字符相同)[除j=2外]；
- 如k=3，则$p_1p_2=p_{j-2}p_{j-1}$(有2个字符相同)；

第三节　串的匹配算法

二、KMP算法(next函数，下标从0开始)

- next函数定义为：

$$next[j]=\begin{cases}-1 & \text{当}j=0\text{时}\\ \max\{k \mid 0<k<j\text{且}‘p_0\ldots p_{k-1}’=‘p_{j-k}\ldots p_{j-1}’\} & \\ 0 & \text{其它情况}\end{cases}$$

- 如k=1，则$p_0=p_{j-1}$　　(有1个字符相同)[除j=1外]；
- 如k=2，则$p_0p_1=p_{j-2}p_{j-1}$(有2个字符相同)；

举列

j	0	1	2	3	4	5	6	7
模式	a	b	a	a	b	c	a	c
next[j]								

第三节　串的匹配算法

二、KMP算法(next函数举例)

- 现有模式串abaababaca, 求其next值， 下标从0开始

j	1	2	3	4	5	6	7	8	9	10
模式串	a	b	a	a	b	a	b	a	c	a
next[j]	-1	0	0	1	1	2	3	2	3	0

第三节　串的匹配算法

二、KMP算法(举例)

- 假设主串ababcabcacbab，模式abcac，改进算法的匹配过程如下

第一趟匹配

```
    ↓i=2
a b a b c a b c a c b a b
a b c a c
    ↑j=2
```

第二趟匹配

```
    ↓i=2
a b a b c a b c a c b a b
    a b c a c
    ↑ j=0
```

第三趟匹配

```
            ↓i=6------9
a b a b c a b c a c b a b
          a b c a c
            ↑j=1
```

第三节　串的匹配算法

为什么*i*可以不进行回溯

主串（S）	a	b	c	a	b	f	?	?	…	?
模式串（T）	a	b	c	a	b	d				

第三节　串的匹配算法

为什么*i*可以不进行回溯

主串（S）	a	b	c	a	b	f	?	?	…	?
模式串（T）	a	b	c	a	b	d				
对于BF算法	a	b	c	a	b	d				

第三节　串的匹配算法

为什么*i*可以不进行回溯

主串（S）	a	b	c	a	b	f	?	?	…	?
模式串（T）	a	b	c	a	b	d				
对于BF算法	a	b	c	a	b	d				

第三节　串的匹配算法

为什么*i*可以不进行回溯

主串（S）	a	b	c	a	b	f	?	?	…	?

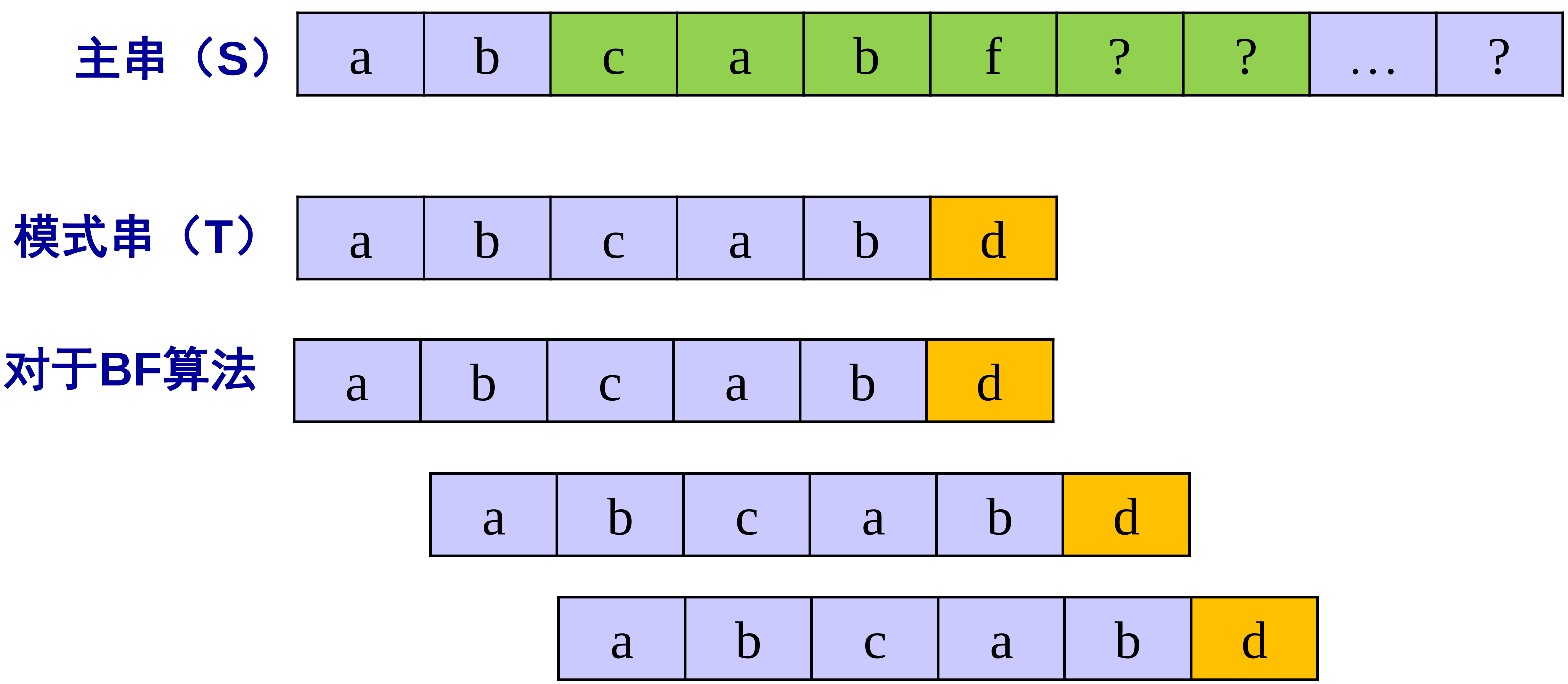
模式串（T）	a	b	c	a	b	d

对于BF算法	a	b	c	a	b	d

a	b	c	a	b	d

第三节　串的匹配算法

为什么*i*可以不进行回溯



第三节　串的匹配算法

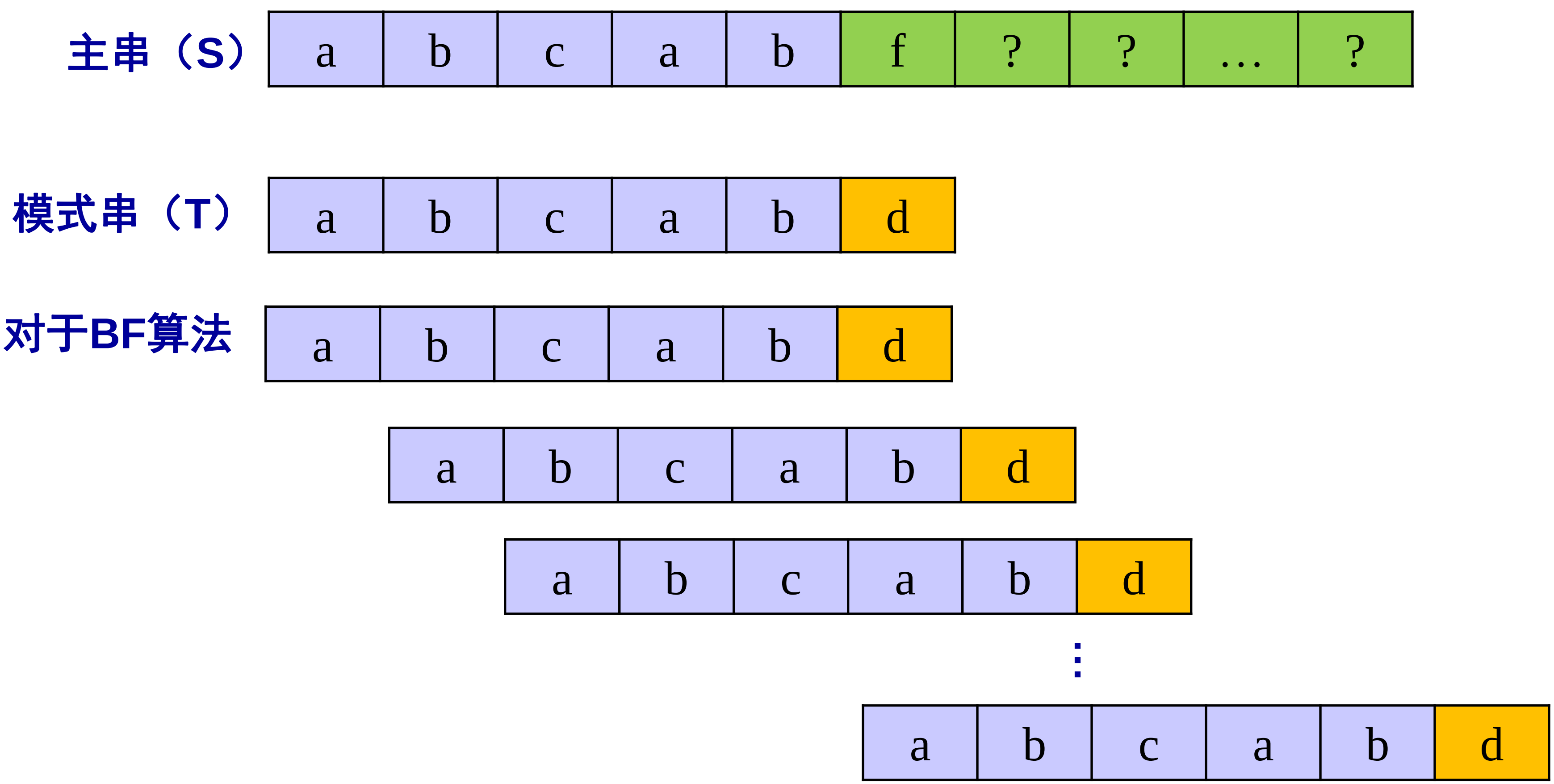
为什么*i*可以不进行回溯

主串（S）	a	b	c	a	b	f	?	?	…	?
模式串（T）	a	b	c	a	b	d				
对于BF算法	a	b	c	a	b	d				
		a	b	c	a	b	d			
			a	b	c	a	b	d		

⋮

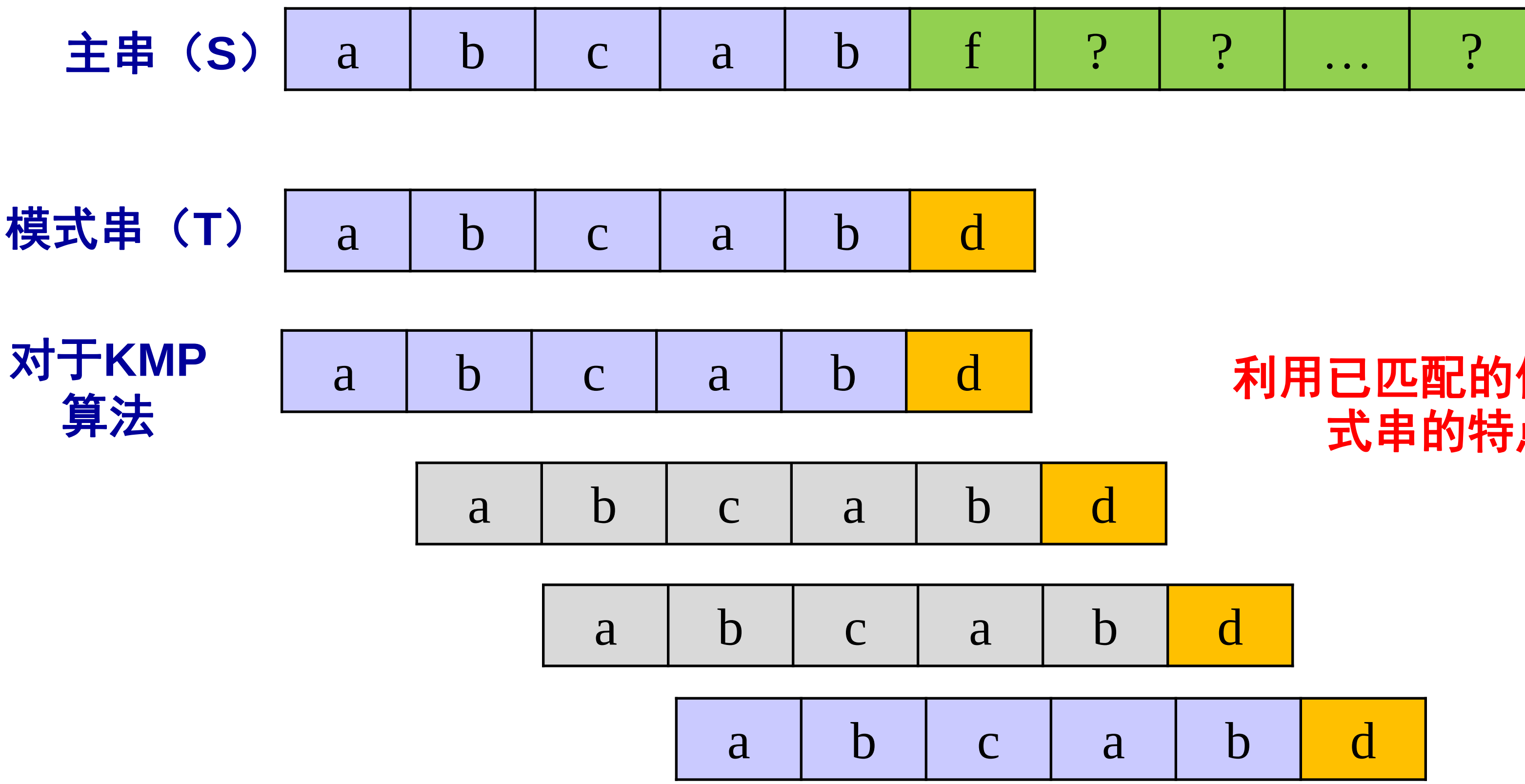
第三节　串的匹配算法

为什么*i*可以不进行回溯



第三节　串的匹配算法

为什么*i*可以不进行回溯



利用已匹配的信息和模式串的特点。

求*next*函数值的过程是一个递推过程，分析如下（实际求解）：

已知： next[0] = -1;

假设： next[j] = k；又 P[j] = P[k]

则： next[j+1] = k+1

若： P[j] ≠ P[k]

则需往前回朔，检查 P[j] = P[?]

这实际上也是一个匹配的过程，

不同在于：主串和模式串是同一个串

假设next[j]=k

	p_0	…	p_{j-k+1}	…	p_{j-2}	P_{j-1}	p_j	p_{j+1}
			p_0	…	P_{k-2}	P_{k-1}	p_k	p_{k+1}

若$p_j=p_k$,

即： $p_0p_2\ldots p_k=p_{j-k+1}\ldots p_j$

所以: next[j+1] = k+1

假设next[j]=k。

p_{j-k+1}	…	$P_{j-next[k]+1}$	…	p_{j-2}	P_{j-1}	p_j
p_0	…	…	…	P_{k-2}	P_{k-1}	p_k
		P_0	…	$P_{next[k]-2}$	$P_{next[k]-1}$	$P_{next[k]}$

若$p_j \neq p_k$

假设next[j]=k。

p_{j-k+1}	…	$P_{j-next[k]+1}$	…	p_{j-2}	P_{j-1}	p_j
		P_0	…	$P_{next[k]-2}$	$P_{next[k]-1}$	$P_{next[k]}$

若$p_j \neq p_k$，

令k=next[k]，若$p_j = p_{next[k]}$，则next[j+1] = next[k]+1

若$p_j \neq p_{next[k]}$，令k=next[next[k]]，重复上述过程

若k=next[0]=-1(下标从0开始，是-1），则next[j+1] = -1+1 = 0

求子串的next[]（自动）

Next[0]=-1,next[j]=k， next[j+1]=？

- 如果 $P_k=P_j$，则 next[j+1]=k+1
- 如果 $P_k!=P_j$，则 设 k’=next[k]
- 如果 $P_{k'=}P_j$， 则 next[j+1]=k’+1
  - 如果 $P_{k'}!=P_j$， 则 设 k’=next[k’]
    - ……
    - 如果 K^n=-1 next[j+1]=0

第三节　串的匹配算法

二、KMP算法(next函数)

- 求next[j]值的算法

1. j的初值为0， next[0]=-1， k=-1
2. While(j<模式串长度-1) {

   (1).若k=-1或者T_j=T_k,则j++,k++,next[j]=k

   (2).否则,k=next[k]

   }

第三节　串的匹配算法

二、KMP算法(next函数 C++实现)

```
int *GetNext(string T)   //计算T串的next值，并返回
{
    int     j,k;
    int     *next = new int[T.size()];  //next分配空间

    j=0,k=-1;
    next[j] = k;                 //next[0]=-1

    while(j<T.size()-1)
    {
        if(k==-1||T[j]==T[k]) //递推
            next[++j] = ++k;
        else
            k = next[k];      //回溯
    }

    return next;
}
```

二、KMP算法(利用next函数)

- 利用next函数,可写出KMP算法如下:

1. 令i的初值为pos,j的初值为0
2. While((i<主串长度)且(j<模式串长度)) {

   (1).若j=—1或者$s_i=p_j$,则i++, j++

   (2).否则,j=next[j]

   }

j=—1表示第一个字符失配

第三节　串的匹配算法

二、KMP算法(C++实现)

```
int KMP(string S,string T)    //在S串查找T串的位置，并返回
{
    int i,j;
    int *next = GetNext(T);  //计算T串的next值

    for(i=0,j=0; i<S.size() && j<(int)T.size();)
    {
        if(j==-1||S[i]==T[j])   //对应字符相等，或S[i]<>T[0]
            i++,j++;
        else                    //S[i]<>T[j],回溯
            j = next[j];
    }

    delete []next;              //释放next空间
    if(j==T.size())             //找到
        return i-j;
    return -1;                  //未找到
}
```

第三节　串的匹配算法

二、KMP算法(时间复杂度)

- KMP()函数的时间复杂度为O(n)
- 为了求模式串的next值,其算法与KMP很相似,其时间复杂度为O(m)
- 因此, KMP算法的时间复杂度为O(n+m)

回顾BF的最恶劣情况：S与T之间存在大量的部分匹配,

比较总次数为： $(n-m+1)*m = O(n*m)$

next求值的改进[了解]

例如：

$S = '\mathbf{aaabaaabaaabaaab}'$

$T = '\mathbf{aaaab}'$

next[j]=-1 0 1 2 3

nextval[j]=-1-1-1-1 3

当s3 ≠ t3时，s3依次与t2, t1, t0 比较，事实上t3=t2=t1=t0,这些比较不是必要的，为了避免这些不必要的比较，只需要让next[3]=next[2]=next[1]=next[0]，

Nextval的手工计算方法：

- 首先计算next
- 比较当前字符t.ch[j]与其next值k所指字符t.ch[k]
  - 不等： nextval[j]=next[j]（即维持不变）
  - 相等： nextval[j]=nextval[k]

Next计算举例

j	0	1	2	3	4	5	6	7
模式	a	b	a	a	b	c	a	c
next	-1	0	0	1	1	2	0	1
nextval	-1	0	-1	1	0	2	-1	1

j=0 nextval[0]=-1

j=1 next[1]=0 p[1]≠p[0] nextval[1]=next[1] = 0

j=2 next[2]=0 p[2]=p[0] nextval[2]=nextval[0]=-1

j=3 next[3]=1 p[3]≠p[1] nextval[3]=next[3]=1

j=4 next[4]=0 p[4]≠p[0] nextval[4]=0

j=5 next[5]=2 p[5]≠p[2] nextval[5]=2

j=6 next[6]=0 p[6]=p[0] nextval[6]=nextval[0]=-1

j=7 next[7]=1 p[7]≠p[1] nextval[7]=1

练习

求串eefegeef的next值。写出计算过程。

假设主串为eefeefegeebeefegeeb，写出KMP算法查找串eefegeef的过程。