第四章 串

4.1 串的类型定义

4.2 串的表示与实现

4.3 串的模式匹配算法

4.1 串的概念

一. 字符串

- 串即字符串，是n(≥0)个字符组成的有限序列，记作：

  S = ‘$a_1a_2a_3\cdots a_n$’

  - S 是串名字
  - ‘$a_1a_2a_3\cdots a_n$’ 是串值
  - a_i 是串中第i个字符
  - n 是串的长度（串中字符的个数）

  例如：串S = ‘Shen Zhen’ 的长度是9，其中有3个字符h、e、n是重复出现，并且有1个空格符

4.1 串的概念

二. 串的术语

- 空串：不含任何字符的串，串长度为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（含空串）。

例如，‘a b c d e’的子串有：

‘’ 、‘a’、 ‘a b’ 、 ‘a b c’ 、 ‘a b c d’

和 ‘a b c d e’ 等等

- 真子串： 非空且不包含自身的所有子串。

4.1 串的概念

二. 串的术语

- **串相等**：当且仅当两个串的长度相等且各个对应位置的字符都相等时，这两个串才相等。

例如：

‘1234’ ≠ ‘123’

‘abcd’ ≠ ‘abcde’

4.1 串的概念

二. 串的术语

- 主串：包含子串的串。
- 位置：字符在串中的序号。

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。

例如，串‘cde’在串‘abcde’中的位置是3。

- 模式匹配：确定子串在主串中首次出现的位置的运算。

例如：

对子串‘bcd’在主串‘abcde’中进行模式匹配，得到结果为2。

4.1 串的概念

三. 串与线性表的关系

- 相同点：
  - 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
- 区别
  - 串的数据对象约定是字符集，线性表可以是任意类型
  - 线性表的基本操作中，以“单个元素”作为操作对象；

  串的基本操作中，通常以“串的整体”作为操作对象，例如，查找子串、插入子串等

4.1 串的概念

四. 串的基本操作

- StrAssign（&S, chars ）：将字符串常量chars赋给串S
- StrCppy(&S,T) ：串复制
- StrEqual(S,T) ：判断串S、T是否相等
- StrLength(S) ：求串的长度
- Concat(S,T) ：串连接
- SubStr(S,i,j) ：求子串
- Index(S,T,pos)：返回串T在串S中pos个字符之后第一次出现的位置
- 。。。。。。

4.2 串的表示与实现

■串是一种特殊的线性表，其存储表示和线性表类似，但又不完全相同。串的存储方式取决于将要对串所进行的操作。串在计算机中有3种表示方式：

- 定长顺序存储表示
- 堆分配存储方式
- 块链存储方式

4.2 串的表示与实现

一. 定长顺序存储表示

- 将串定义成字符数组，利用串名可以直接访问串值。用这种表示方式，串的存储空间在编译时确定，其大小不能改变。

  例如，C语言中的字符串定义

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

  - 定义了长度为MAXSTRLEN的字符存储空间
  - 字符串长度可以是不大于于MAXSTRLEN的任何值（最长串长度有限制，多余部分将被**截断**）

- 串的定义

  ```
  #define MAX_STRLEN  255
  typedef  unsigned char  str[MAX_STRLEN+1] ;
  ```

4.2 串的表示与实现

一. 定长顺序存储表示

- 串的联接

```
Status Concat(SString &T, SString S1, SString S2) {
  int i;
  Status uncut;
  if ( S1[0]+S2[0] <= MAXSTRLEN ) {  // S1 S2均未截断
    for (i=1; i<=S1[0]; i++) T[i] = S1[i];
    for (i=1; i<=S2[0]; i++) T[i+S1[0]] = S2[i];
    T[0] = S1[0]+S2[0];
    uncut = TRUE;
  } else if ( S1[0] < MAXSTRLEN ) {  // 截断 S2
    for (i=1; i<=S1[0]; i++) T[i] = S1[i];
    for (i=S1[0]+1; i<=MAXSTRLEN; i++) T[i] = S2[i-S1[0]];
    T[0] = MAXSTRLEN;
    uncut = FALSE;
  } else {                       // 截断S1
    for (i=0; i<=MAXSTRLEN; i++) T[i] = S1[i];
    uncut = FALSE;
  }
  return uncut;
} // Concat
```

4.2 串的表示与实现

一. 定长顺序存储表示

- 求子串

```
Status SubString(SString &Sub, SString S, int pos, int len)
{ // 用Sub返回串S的第pos个字符起长度为len的子串
   int i;
   if (pos < 1 || pos > S[0] || len < 0 || len > S[0]-pos+1)
      return ERROR;
   for(i=1; i<=len; i++)
      Sub[i] = S[pos+i-1];
   Sub[0] = len;
   return OK;
} // SubString
```

4.2 串的表示与实现

二. 堆分配存储表示

- 仍然用一组地址连续的存储单元来依次存储串中的字符序列，但串的存储空间是在程序运行时根据串的实际长度动态分配的。

例如，C语言，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。

由malloc()和free()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。

4.2 串的表示与实现

二. 堆分配存储表示

- 堆分配的串定义

```
typedef  struct
{
    char *ch;    /* 若非空，按长度分配，否则为NULL */
    int length;     /* 串的长度 */
} HString ;
```

4.2 串的表示与实现

二. 堆分配存储表示

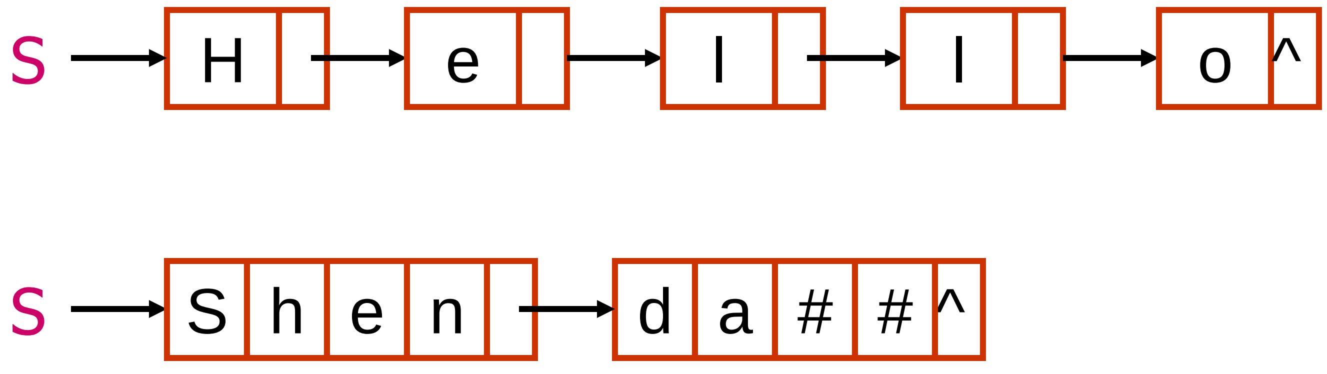
- 堆分配的串插入

```
Status StrInsert(HString &S, int pos, HString T)
{// 1 ≤ pos ≤ StrLength(S)＋1。在串S的第pos个字符之前插入串T
  int i;
  if (pos < 1 || pos > S.length+1)  // pos不合法
       return ERROR;
  if ( T.length ) {    // T非空,则重新分配空间,插入T
       if ( !(S.ch = (char *)realloc(S.ch, (S.length+T.length+1) * sizeof(char))) )
          return ERROR;
       for ( i = S.length-1; i>=pos-1; --i)  // 为插入T而腾出位置
          S.ch[i+T.length] = S.ch[i];
      for (i=0; i<T.length; i++)        // 插入T
          S.ch[pos-1+i] = T.ch[i];
      S.length += T.length;
  }
  return OK;
} // StrInsert
```

4.2 串的表示与实现

三. 块链存储表示

- 是一种链式存储结构表示，采用链表方式存储串值，每个结点中，可以存放一个字符（结点大小为1），也可以存放n个字符（结点大小为n）。



4.2 串的表示与实现

三. 块链存储表示

- 块链存储的串定义

```
#define CHUNKSIZE  80
typedef  struct  Chunk
{
        char  ch[CHUNKSIZE] ;
        struct  Chunk  *next;
}Chunk;

typedef  struct
{       Chunk  *head, *tail;       /*  头尾指针  */
        int  curlength ;        /*  当前长度  */
} LString ;
```

4.2 串的表示与实现

三. 块链存储表示

- 因为带了next指针，存储密度小于1

$$\text{存储密度} = \frac{\text{串值所占存储位}}{\text{实际分配的存储位}}$$

- 在这种存储结构下，结点的分配总是以完整的结点为单位，因此，为使一个串能存放在整数个结点中，在串的末尾填上不属于串值的特殊字符#，以表示串的终结。

S → | S | h | e | n | | → | d | a | # | # | ^ |

- 当一个块(结点)内存放多个字符时，往往会使操作过程变得较为复杂，如在串中插入或删除字符操作时通常需要在块间移动字符。

4.3 串的匹配

- **模式匹配**：子串在主串中的定位称为模式匹配或串匹配。

主串s

是子串吗?

模式串t

模式匹配

- 成功是指在主串s中找到一个模式串t——t是s的子串，返回t在s中的位置。
- 不成功则指主串s中不存在模式串t——t不是s的子串，返回0。

4.3 串的匹配

- 简单匹配算法

朴素算法（ BF(Brute-Force)算法)

采用穷举的思想进行匹配

① 从主串的指定位置开始，将主串与模式串（要查找的子串）的第一个字符比较；

② 若相等，则继续逐个比较后续字符；

③ 若不等，从主串的下一个字符起再重新和模式串的第一个字符比较。

4.3 串的匹配

- BF(Brute-Force)算法**穷举模式的匹配过程：**

```
              1 2 3 4 5 6
第1趟   S     a b b a b a
        T     a b a

第2趟   S     a b b a b a
        T       a b a

第3趟   S     a b b a b a
        T         a b a

第4趟   S     a b b a b a
        T           a b a        匹配成功！
```

4.3 串的匹配

- BF算法实现函数Index

```
Int Index(Sstring S, Sstring T, int pos) {
//S为主串，T为模式，串的第0位置存放串长度；串采用顺序存储结构
    i = pos;  j = 1;                  // 从第一个位置开始比较
    while (i<= StrLength(S) && j<= StrLength(T)) //当两串未检测完
    {
      if (S[i] == T[j]) {  ++i;  ++j;  } // 继续比较后续字符
      else { i = i – j + 2;    j = 1;}   // 主串和模式串的指针分
                                         别后退，重新开始匹配
    }
    if (j > StrLength(T)) return i-StrLength(T); // 返回与模式首
    字符相等的位置
    else return 0;   // 匹配不成功
}
```

4.3 串的匹配

- BF算法

设主串S = '$s_1s_2...s_n$'　模式串T = '$t_1t_2...t_m$'

i为指向S中字符的指针，j为指向T中字符的指针

匹配失败：当 $s_i \neq t_j$ 时，

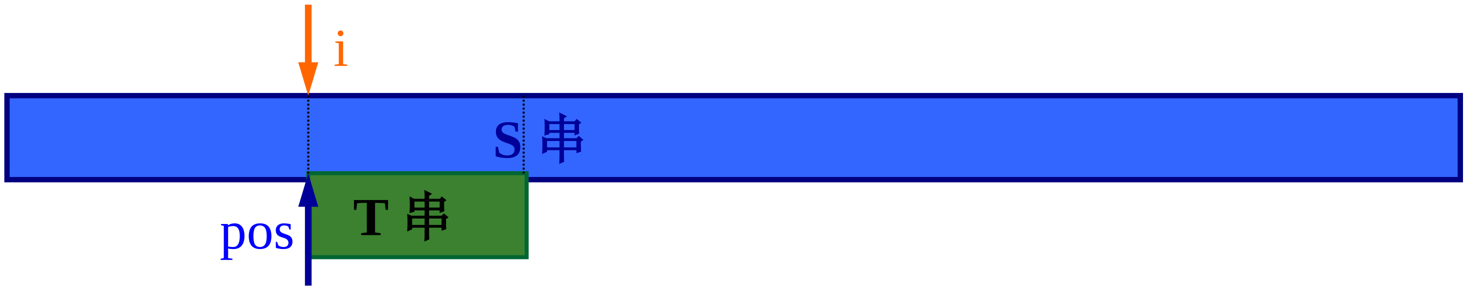
虽然已经判断出 '$s_{i-j+1} \ldots s_{i-1}$' = '$t_1 \ldots t_{j-1}$'，还是

要分别回退指针 i、j：　i = i - j+2；　j=1

主串指针i重复回溯！

4.3 串的匹配

- BF算法性能分析：



最好的情况下，在第一个对齐位置只经过一轮比对之后，就能确定整体匹配，其时间复杂度为：O(m)（m为模式串的长度）。

4.3 串的匹配

- BF算法性能分析：

最坏情况下，需要比较n-m+1趟，每趟比较m次，总比较次数达(n-m+1)*m，因此，其时间复杂度为O(n*m)

例如：主串为‘000000000000000000000000000001’，模式串为‘000001’，则每次模式串的前5个0都要与主串逐一比较.

4.3 串的匹配

- BF算法分析：

在字符比较不相等，需要回溯 ：即退到s 中的下一个字符开始进行继续匹配。

- 最好情况下的时间复杂度为O(m) 。
- 最坏情况下的时间复杂度为O(n×m) 。

有性能更好些的算法吗？

4.3 串的匹配

- KMP算法是由D. E. Knuth(克努特)－J. H. Morris(莫里斯)－V. R. Pratt(普拉特)提出。

  该算法较BF算法有较大改进，主要是消除了主串指针的回溯。

4.3 串的匹配

- KMP算法思路
  - 当一趟匹配过程中出现字符比较不等(失配)时，不需回溯主串指针i
  - 利用已经得到的“部分匹配”的结果，将模式串向右“滑动”尽可能远的一段距离后，继续进行比较。

4.3 串的匹配

- KMP算法举例

假设主串ababcabcacbab，模式串abcac，KMP算法的匹配过程如下：

↓ i=3

第一趟匹配 a b $\boxed{a}$ b c a b c a c b a b

a b $\boxed{c}$ a c

↑ j=3

↓ i=3 —→7

第二趟匹配 a b a b c a $\boxed{b}$ c a c b a b

a b c a $\boxed{c}$

↑ j=1

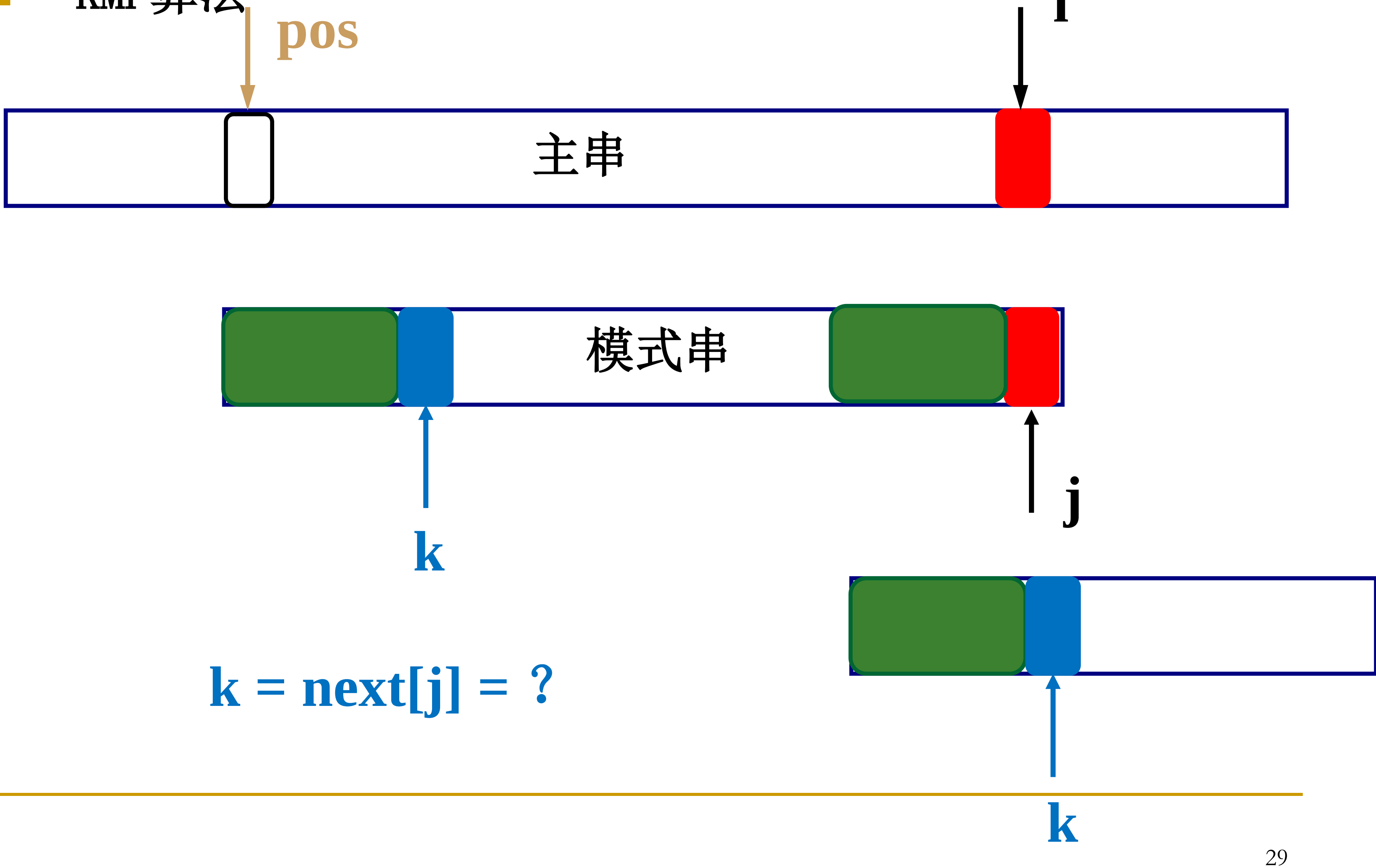
↓ i=7-->10

第三趟匹配 a b a b c a b c a c b a b

a b c a c 匹配成功！

↑ j=2

- KMP算法



k = next[j] = ?

4.3 串的匹配

- KMP算法说明

  - 假设主串为‘$s_1s_2s_3\cdots s_n$’，模式串为‘$t_1t_2t_3\cdots t_m$’，若主串中第i个字符与模式串中第j个字符“失配”(s_i!=t_j)，这说明，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：

	s_{i-j+1}	…	s_{i-k+1}	…	s_{i-2}	s_{i-1}	s_i	…
	t_1	…	t_{j-k+1}	…	t_{j-2}	t_{j-1}	t_j	…

那么，主串中第i个字符将与模式串中第几个字符再比较？

- KMP算法说明
  - 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

	s_{i-j+1}	…	s_{i-k+1}	…	s_{i-2}	s_{i-1}	s_i	…
	t_1	…	t_{j-k+1}	…	t_{j-2}	t_{j-1}	t_j	…
			t_1	…	t_{k-2}	t_{k-1}	t_k	…

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：$t_1t_2\ldots t_{k-1} = s_{i-k+1}s_{i-k+2}\ldots s_{i-1}$

4.3 串的匹配

	s_{i-j+1}	…	s_{i-k+1}	…	s_{i-2}	s_{i-1}	s_i	…
	t_1	…	t_{j-k+1}	…	t_{j-2}	t_{j-1}	t_j	…
			t_1	…	t_{k-2}	t_{k-1}	t_k	…

表中黑色字体表示对应列字符相等，红色表示 $s_i \neq t_j$

- 由以下两个表达式

① $t_1 t_2 \ldots t_{j-k+1} t_{j-k+2} \ldots t_{j-1} = s_{i-j+1} s_{i-j+2} \ldots s_{i-k+1} s_{i-k+2} \ldots s_{i-1}$

② $t_1 t_2 \ldots t_{k-1} = s_{i-k+1} s_{i-k+2} \ldots s_{i-1}$

可以得到

$$'t_1 t_2 \ldots t_{k-1}' = 't_{j-k+1} t_{j-k+2} \ldots t_{j-1}'$$

4.3 串的匹配

- 主串中第i个字符与模式串中第k个字符再比较

  换言之,在模式串中第j个字符“失配”时,如果有 ‘$t_1t_2\ldots t_{k-1}$’ = ‘$t_{j-k+1}t_{j-k+2}\ldots t_{j-1}$’, 那么就可以用模式串第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。

- 如何确定k?

  k值可以在做模式匹配之前求出，一般用next函数求取k值。

- 注意：next函数只和模式串有关，和主串无关

4.3 串的匹配

- next函数定义为：

$$
next[j]=\begin{cases} 0 & \text{当} j=1\text{时} \\ Max\{\, k \mid 1<k<j \text{且} 't_1\ldots t_{k-1}'='t_{j-k+1}\ldots t_{j-1}'\} & \text{当此集合非空时} \\ 1 & \text{其它情况} \end{cases}
$$

若$t_1 = t_{j-1}$，则k=2　　(有1个字符相同)[除j=2外]；

若$t_1t_2 = t_{j-2}t_{j-1}$，则k=3（有2个字符相同）；

※ 用数组next[]存放“部分匹配”信息。当模式串的字符j与主串中的当前字符i不相等时，j应回退的位置是0或1或k。

4.3 串的匹配

next[j] = Max{ k | 1<k<j且 '$t_1 \ldots t_{k-1}$'='$t_{j-k+1} \ldots t_{j-1}$'}当此集合非空时

模式串t中存在某个k（1<k<j），使得以下成立:

$$\underbrace{'t_1\ t_2\ \ldots\ t_{k-1}'}_{\text{以}t[1]\text{开始的}k\text{-}1\text{个字符}} = \underbrace{'t_{j-k+1}\ t_{j-k+2}\ \ldots\ t_{j-1}'}_{\text{以}t_{j-1}\text{结尾的}k\text{-}1\text{个字符}}$$

※利用模式串t中隐藏的信息来提高模式匹配的效率。

4.3 串的匹配

1 2 3 4 5

例如： t= ‘a b a b c ’ 考虑next[5] = ?

↓

有t_1t_2 = t_3t_4 = '*ab*' 所以 k-1=2， 所以next[5] = k = 3。

4.3 串的匹配

- next函数举例
  - 手工计算next函数例1

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

4.3 串的匹配

- next函数举例
  - 手工计算next函数例2

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]	0	1	1	2	2	3	4	3	4	1

4.3 串的匹配

- **KMP思想：**

　　有主串s与模式串t，i、j分别代表主串和模式串中正待比较的字符下标，令i和j的初值为1。

　　若有s[i]=t[j]，则i和j分别增加1，继续比较后续字符；否则，即s[i]≠ t[j]时，进行下列操作：

① i 不 变， j 退 回 到 某 个 j=next[[…next[j]…]] 位 置 时 有 s[i]=t[j]，则i和j分别增加1后继续比较后续字符；

② 当j退回到j=0（即模式的第一个字符“失配”），此时i、j指针分别增加1，从s[i+1]和t[1]开始继续比较。

4.3 串的匹配

- **KMP算法实现的详细步骤：**

  1. 在主串s中和模式串t中，设比较的起始下标分别是i和j;

  2. 循环下面步骤直到 s 和 t 的所有字符均比较完

  ① 如果是s[i]=t[j]或者j=0，i++，j++;

  ② 否则，i不变，j回溯到next[j]，准备下一趟比较;

  3. 如果t中所有字符均比较完，则匹配成功，返回t的位置; 否则，匹配失败，返回0。

4.3 串的匹配

- KMP算法实现

```
int Index_KMP(Sstring S, Sstring T, int pos) {
//S为主串，T为模式，串的第0位置存放串长度；串采用顺序存储结构
      i = pos;   j = 1;                                    // 从第一个位置开始比较
      get_next( T,  next );
      while (i<= StrLength（ S）  && j<=  StrLength（T ）) {
         if ( (j==0) || S[i] == T[j]) )  {  ++i;   ++j;}  // 继续比较后继字符
         else    j = next[j];                            // 模式串向右移
      }
      if (j > StrLength（T ）)
         return i-T[0];    // 匹配成功,返回模式串的位置
      else
         return 0;         // 匹配不成功
}
```

4.3 串的匹配

- 求next[j]值的算法

求*next*函数值的过程是一个递推过程。已知next [0, ..., j]，即对于t的前j个序列字符：有next[1]=0, …， next[j] = k （$1<k<j$），如何求出next [j + 1]呢？

$$\underbrace{'t_1\ t_2 \ldots\ t_{k-1}'}_{\text{以t[1]开始的}k-1\text{个字符}} \quad = \quad \underbrace{'\ t_{j-k+1}\ t_{j-k+2} \ldots\ t_{j-1}\ '}_{\text{以t}[j-1]\text{结尾的}k-1\text{个字符}}$$

实质上是判断 t[k] 和 t[j] 是否相等。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。

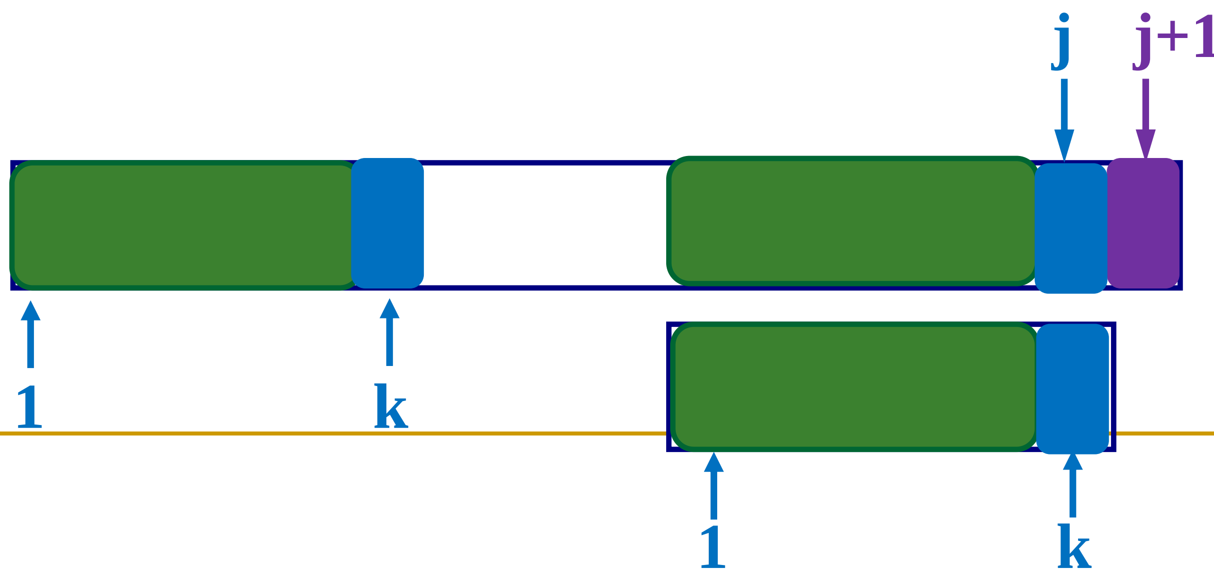
4.3 串的匹配

- 求next[j]值的算法

✓ 第1种情况： **t[k] = t[j]，也就是有**

$$'\underbrace{t_1\ t_2 \ldots\ t_{k-1}\ {\color{red}t_k}}_{\text{以t[1]开头的}k\text{个字符}}' \quad = \quad '\underbrace{t_{j-k+1}\ t_{j-k+2} \ldots\ t_{j-1}\ {\color{red}t_j}}_{\text{t[}j+1\text{]前面的}k\text{个字符}}'$$

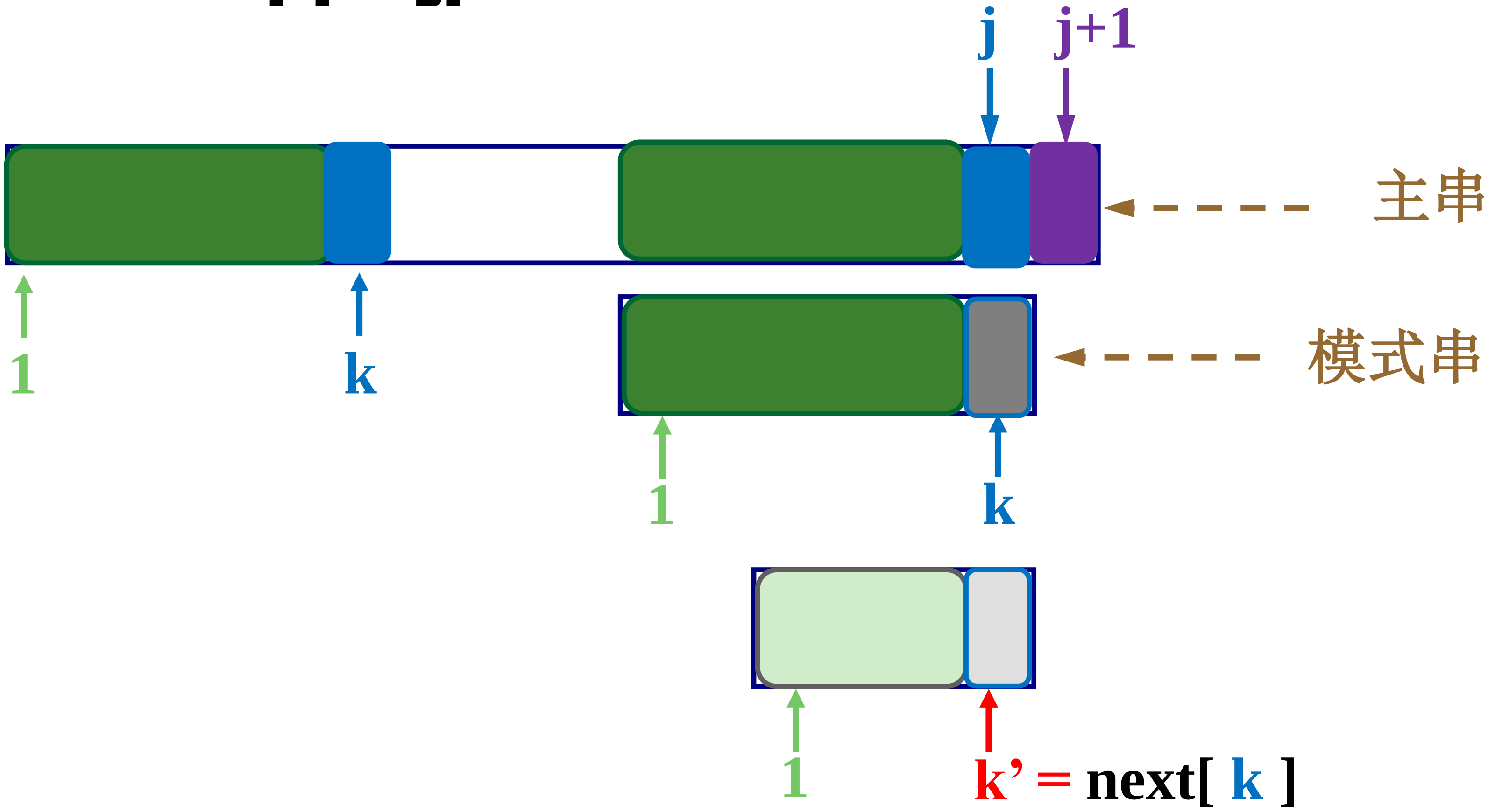
则 next[j + 1] = k + 1 = next [j] + 1



4.3 串的匹配

- 求next[j]值的算法

✓ 第2种情况： **t[k] ≠ t[j]**



4.3 串的匹配

- 求next[j]值的算法

✓ 第2种情况：若t[k] ≠ t[j]，如果此时next[k]= k'，并且t[k'] = t[j]，则next[j + 1] = k'+1=next[k] + 1，否则继续递归k'。

也就是说：当字符t[j+1]之前不存在 't_1, …, t_k '='t_{j-k+1}, …, t_j' 时，那么是否可能存在另一个值k'(1<k'<k<j)，可以有 't_1, …,$t_{k'}$' = '$t_{j-k'+1}$, …, t_j '呢？如果存在，那么就有next[j+1] =k'+1；如果不存在k'，那么就有next[j+1]=1。

这个过程相当于利用已经求得的next []值（next [0, ..., k, ..., j]）进行t串自身的匹配。

4.3 串的匹配

- 求next[j]值的两种算法
  - 传统算法 get_next( )

    根据$1<k<j$，穷举k, 检查是否满足

'$t_1 \ldots t_{k-1}$' = '$t_{j-k+1} \ldots t_{j-1}$', 若满足条件的k有多个，取**最大值**

While(j<模式串长度) {

1. 若j=0或者T_i=T_j, 则i++, j++, next[i]=j, 实质就是 next[j+1]= next[j] + 1
2. 否则, j=next[j], 即next[j+1]= next[k] + 1, 注意 next[k]要回溯直到满足前面的相等条件

}

4.3 串的匹配

- 求next[j]值的传统算法

```
void get_next(Sstring T,  int next[ ])
{ //求模式串T的next函数值并存入数组next[]
      i = 1;    next[1] = 0; j = 0;
      while ( j <  StrLength（T）)
      {
            if ( j == 0 || T[i] == T[j])
            {
                        ++i;      ++j;
                        next[i] = j;
            }
            else
                        j = next[j];
      }
}
```

4.3 串的匹配

- 求next[j]的 改进值 nextval [j]的算法

例如：

　　1 2 3 4 5

T = 'a a a a b'

next[j]=0 1 2 3 4

当s[i] ≠ t[3]时，s[i]要依次与t[2]（即t[next[3]]）、t[1]比较，事实上t[3]=t[2]=t[1]=‘a’，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]，所以有：

nextval[j]= 0 0 0 0 4

4.3 串的匹配

- 求next[j]的 改进值 nextval [j]的算法

- **nextval[1] = 0**
- 当**t[j] = t[next[j]]** 时：

  nextval [j] = nextval[next[j]]

- 否则： **nextval[j] = next[j]**

用**nextval [ ]**取代**next[ ]**，得到 改进的**KMP** 算法。

4.3 串的匹配

- 求next[j]的 改进值 nextval [j]的算法

```
void get_nextValue(Sstring T, int nextval[ ])
{ //求模式串T的next函数值并存入数组nextval[]
    i = 1;        nextval[1] = 0;             j = 0;
    while ( j <  StrLength（T ） )
    {
        if ( j == 0 || T[i] == T[j])
        {
            ++i;      ++j;
            if( T[i] == T[j]  )
                nextval[i] = nextval[j] ;
            else
                nextval[i] = j;
        }
        else
            j = nextval[j];
    }
}
```

4.3 串的匹配

- KMP算法性能分析
  - KMP算法的时间复杂度为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)。
- KMP算法的特点

匹配时，主串的指针不需要回溯，整个匹配过程中，对主串只需要从头至尾扫描一遍，对于处理从外设输入的庞大文件很有效，可以边读入边匹配，而无需回头重读。

第4章总结

- 串即字符串，是n(≥0)个字符的有限序列
- 串的属性：字符、位置、长度
- 串的术语：空串、空格串、主串、子串、位置、匹配
- 串相等的条件：长度相等且每个位置上的字符相等
- 模式匹配：确定子串在主串中首次出现的位置
- 串的特点，往往以串的整体作为操作对象，例如复制、合并、匹配都是以整个字符串来操作，而不是单个字符
- 串的存储表示
  - 定长顺序存储表示、堆分配存储表示、块链存储表示
- 串匹配算法：KMP算法
  - 匹配不成功时，i不动，模式串滑动到位置k开始比较，k即next[j]，求k的方法

练习

一．现有模式串eefegeef，写出每个字符的next函数值。

j	1	2	3	4	5	6	7	8
模式串	e	e	f	e	g	e	e	f
next[j]	0	1	2	1	2	1	2	3

练习

二. 假设主串abcabadabaabc，模式串abaabc，说明KMP算法的匹配过程。

next[j] = 0 1 1 2 2 3

第**1**次匹配：从头开始比较到**i=3 j=3**不等，模式滑动到**next[3]=1**比较**i=3 j=1**

第**2**次匹配：**i=3 j=1**失配，查到**next[1]=0**，则**i++**，**j++**变为**i=4 j=1**

第**3**次匹配：从**i=4 j=1**开始匹配到**i=7 j=4**失配，**i**不动，模式滑动到**next[4]=2**比较

第**4**次匹配：一开始**i=7 j=2**不等，查到**next[2]=1**，**i**不动，模式滑动选择**j=1**比较

第**5**次匹配：一开始**i=7 j=1**不等，查到**next[1]=0**，则**i++ j++**，变为**i=8 j=1**

第**6**次匹配：从**i=8 j=1**开始逐个字符匹配直到结束，匹配成功

练习

j	1	2	3	4	5	6	7	8	9	10
模式	a	b	a	c	a	b	a	a	a	d
Next［j］	0	1	1	2	1	2	3	4	2	2
Nextval［j］	0	1	0	2	0	1	0	4	2	2

北邮2000考研真题