数据结构

Ch4 串

计算机学院（国家示范性软件学院）

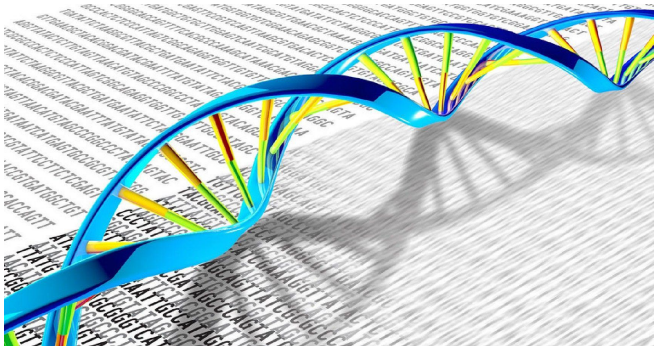
第4章 串

- 4.1 串类型的定义
- 4.2 串的表示和实现
  - 4.2.1 定长顺序存储表示
  - 4.2.2 堆分配存储表示
  - 4.2.3 块链存储表示
- 4.3 串的模式匹配算法
- 4.4 串应用示例–行文本编辑
- 4.5 本章知识点小结

4.1 串类型的定义

4.1.1 串的概念

串（**字符串**）：是由零个或多个字符组成的有限序列。 是特殊的线性表，数据元素是单个字符。



相对于一般的线性结构的特点：

- 结构简单，规模庞大，元素重复率高
  - ASCII字符组成的字符串可以表达近乎无限的内容
  - 4种碱基构成了DNA序列
- 模式匹配（查找）操作：模式检测、模式定位、模式计数等
  - 文本搜索、邮件过滤、热词统计……

4.1 串类型的定义

4.1.1 串的概念

s = “$a_1a_2\cdots a_n$” (n≥0)，其中ai∈Σ，Σ是所有可用字符的集合，称为字符表（Alphabet）

- s是串名，引号本身不属于串的内容。a_i(1<=i<=n)是一个任意字符，称为串的元素，i是它在整个串中的序号。
- n为串的长度，表示串中所包含的字符个数，当n=0时，称为空串,通常记为ϕ。

基本术语

子串和**主串**：串中任意个连续的字符组成的*子序列*称为该串的*子串*。包含子串的串称为*主串*。

子串的位置：子串的第一个字符在主串中的序号称为子串的位置。

例：abcdpqrst　cdp子串在主串中的位置为3

串相等：两个串长度相等，且对应位置的字符都相等。

空串和**空白串**：*空串*不包含任何字符,表示为ϕ；*空白串*由一个或多个空格组成，如‘ ’。

串的基本操作

(1) StrAssign(&T, chars) ： 生成一个值等于chars的串T

(2) StrCopy(&T, S)： 由串S复制得到串T

(3) StrEmpty(S)： L判断S是否为空串

(4) StrLength(S)：返回串S的元素个数，即求串的长度

(5) Concat(&T, S1, S2)：用T返回由S1和S2连接而成的新串两串

(6) SubString(&Sub, S, pos, len)：

用Sub返回串S第pos个字符起长度为len的子串

(7) StrCompare(S, T)： 两串比较

若S>T, 返回值>0，若S=T，返回值=0，若S<T，返回值<0

串的基本操作

(8) Index(S, T, pos)

若主串S中从pos开始的部分存在值和T相同的子串，则返回pos开始后第一次出现的位置，否则返回0

(9) Replace(&S, T, V)

用V替换主串S中出现的所有与T相等的不重叠子串

(10) StrInsert(&S, pos, T)

在串S的第pos个字符之前插入串T

(11) StrDelete(&S, pos, len)

从串S中删除第pos个字符起长度为len的子串

(12) ClearString(&S)：将S清为空串

(13) DestroyString(&S) 销毁串S

[例]

设 s = ‘I am a student.’

t = ‘OK!’

p = ‘student’

q = ‘nurse’

r = ‘good ’

(1) Concat (l, s, t)

l = ‘I am a student. OK!’

(2) Replace(s, p , q);

s = ‘I am a nurse.’

(3) StrInsert(s, 8, r)

s = ‘I am a good nurse.’

4.2 串的表示和实现

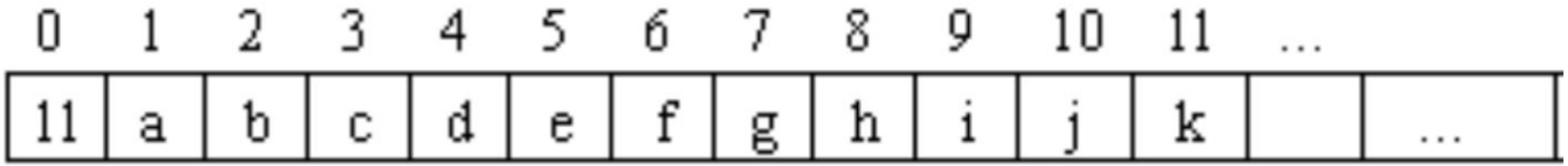
4.2.1 定长顺序存储表示

用一组地址连续的存储单元存储串值中的字符序列，按照预定义的大小，为每个定义的串变量分配一个固定长度的存储区，串的实际长度超过预定义的串值会被舍去，称之为“截断”。

```
#define MAXSTRLEN  255
typedef unsigned char  SString[MAXSTRLEN  + 1]。
```

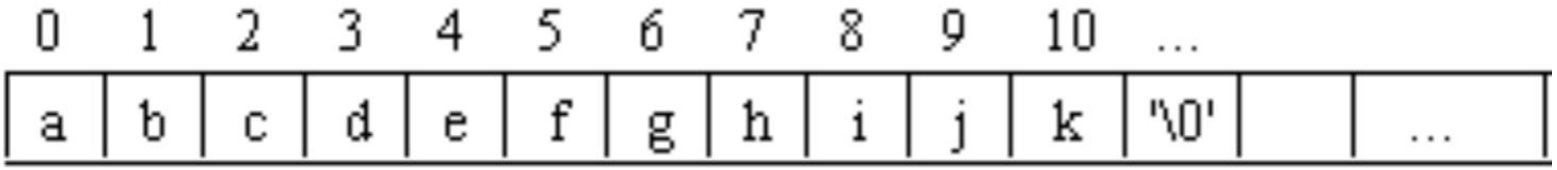
标识串实际长度的方法

方法1：用s[0]存放串的实际长度，串值存放在s[1]~s[MAXSTRLEN]

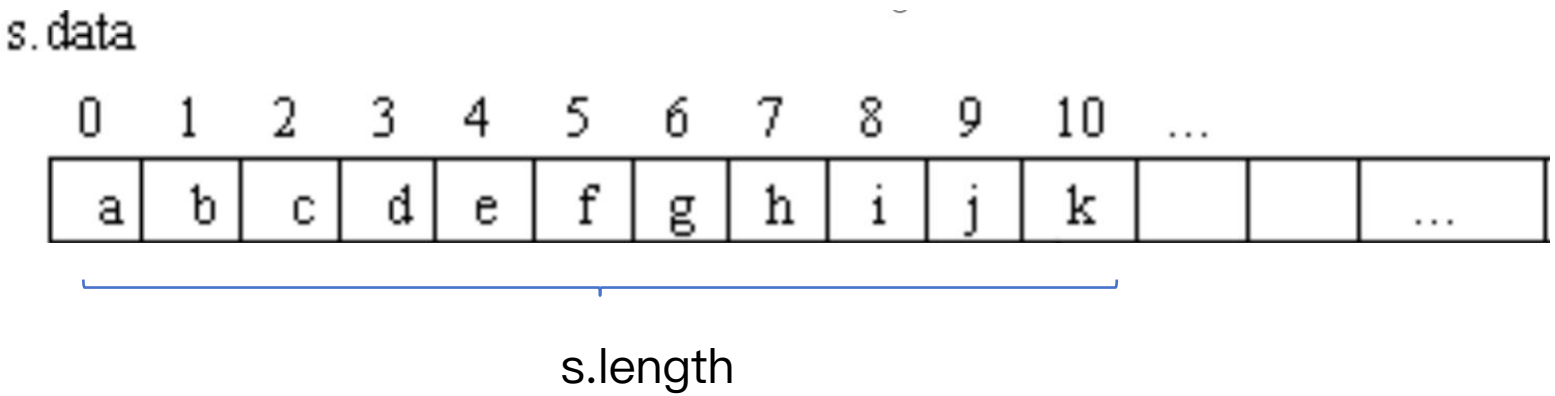


标识串实际长度的方法

方法2：在串尾存储一个特殊字符来作为终结符

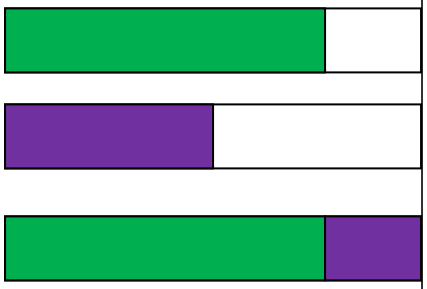


方法3：类似顺序表，用一个变量length来表示串的长度。

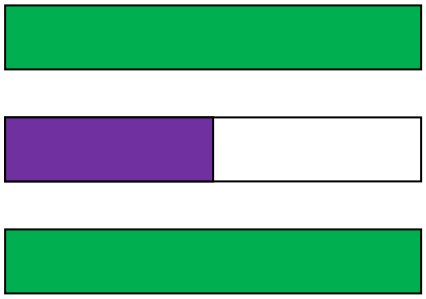


```
Status Concat(SString  &T,  SString S1, SString S2)
//用T返回串s1和s2联接而成的新串。
//uncut表示是否截断，  未截断TRUE，  截断为FALSE
{   if  ( S1[0]+S2[0] <= MAXSTRLEN ) {
          T[1..S1[0]] = S1[1..S1[0]];
          T[s1[0]+1..S1[0]+S2[0]] = S2[1..S2[0]];
          T[0]= S1[0]+S2[0];  uncut=TRUE;
    }
    else if  (S1[0]<MAXSTRLEN) {
        T[1..S1[0]] = S1[1..S1[0]];
        T[s1[0]+1..MAXSTRLEN] = S2[1..MAXSTRLEN-S1[0]];
        T[0]= MAXSTRLEN;  uncut=FALSE;
    }
     else  {
          T[0..MAXSTRLEN]= S1[0..MAXSTRLEN];
          uncut= FALSE;
   }
     return uncut;
} // Concat
```

if： 长度和<=容量

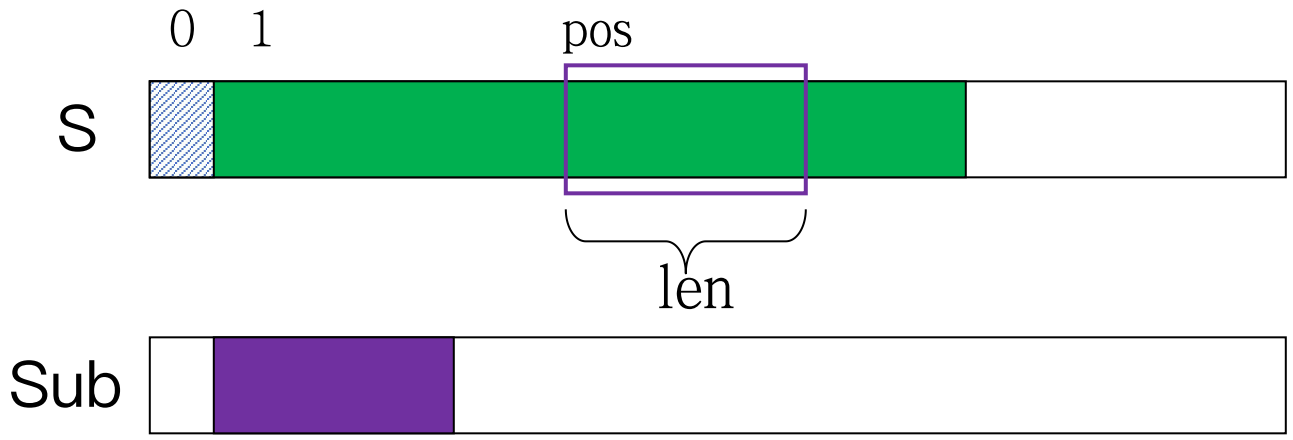


else if： S1长度<容量

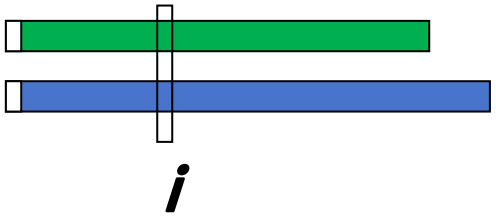


else ： S1长度>=容量

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



```
int StrCompare(SString S, SString T)
// S>T,返回值>0；  S=T，  返回0；  S<T，  返回值< 0
{    for  (i=1；i<=S[0] && i<=T[0]；i++){//逐个字符进行比较
          if  (S[i] != T[i] )
                return(S[i] - T[i] );
     }
      return S[0]-T[0]
} // StrCompare
```



4.2.2 堆分配存储表示

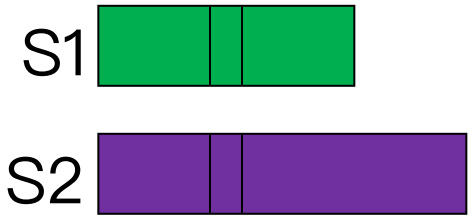
动态分配串值存储空间，避免定长结构的截断现象。

[存储定义]

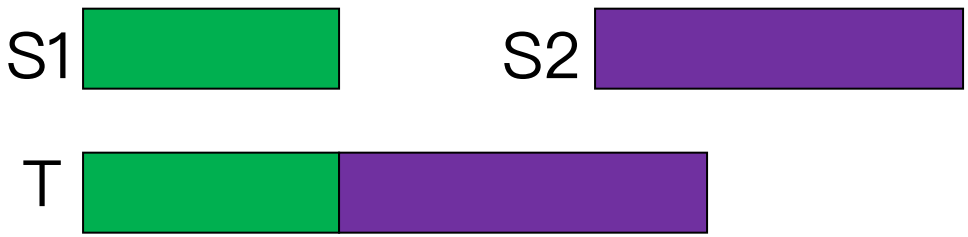
```
typedef  struct {
            char   *ch;    //串空间基址，按串长申请
            int   length;   //串长度
}HString;
```

[基本操作实现示例]

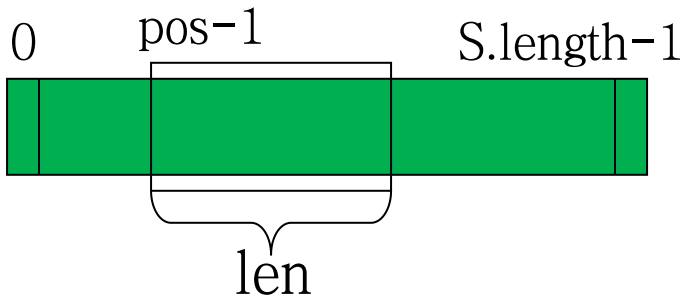
```
int StrCompare(HString  S, HString T)
 //S>T,返回值>0；  S=T，  返回0；  S<T，  返回值< 0
{    for  (i=0；i<S.length && i<T.length；i++){
                if  (S.ch[i] != T.ch[i] )
                        return  S.ch[i] - T.ch[i];
     }
      return  S.length-T.length ；//如果前面的对应位都相等，  则看长度
} // StrCompare
```



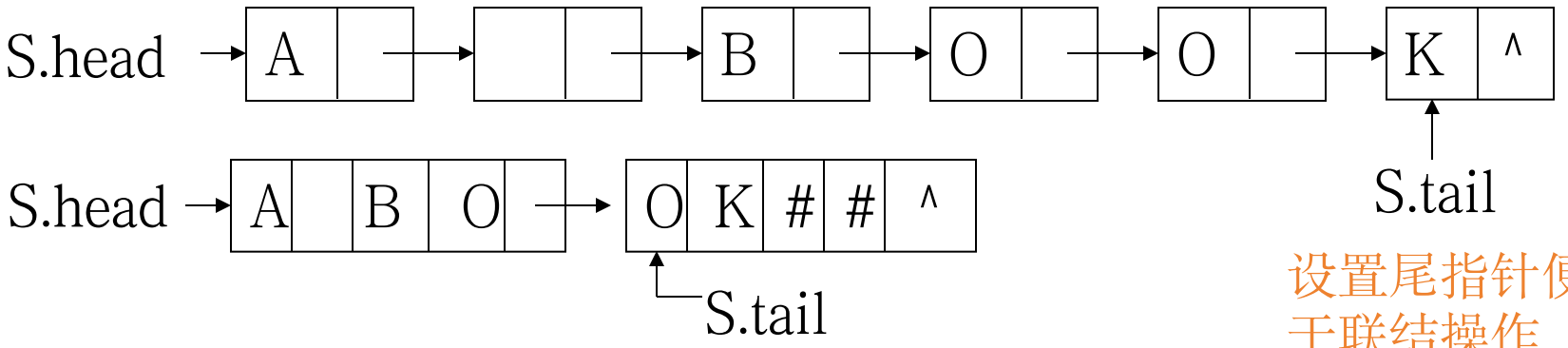
```
Status Concat(HString &T,  HString  S1, HString S2)
//返回串S1和S2联接而成的新串T
{   if  (T.ch)   free(T.ch);   //释放T原有空间
    if  ( ! (T.ch=(char *)malloc((S1.length+S2.length)*
                                 sizeof(char))))
          exit(OVERFLOW);
    T.length=S1.length+S2.length;
    T.ch[0..s1.length-1]=S1.ch[0..s1.length-1];
    T.ch[S1.length.. T.length-1]=S2.ch[0..S2.length-1];
    return OK;
 } // Concat
```



```
Status SubString(HString &Sub, HString S, int pos, int len)
//求串S从第pos个字符起长度为len的子串Sub
{   if ( (pos<1 || pos >S.length  || len<0  || len > S.length-pos+1 )
                                  return ERROR;
    if  (Sub.ch)   free(Sub.ch);   //释放旧空间
    if  ( !len ) {  Sub.ch=NULL;  Sub.length=0;  } //空子串
    else  {
          if  ( !(Sub.ch=(char *)malloc(len* sizeof(char))))
                exit(OVERFLOW);
          Sub.ch[0..len-1]=S.ch[pos-1..pos+len-2];
          Sub.length=len;
     }
     return OK;
 } // SubString
```



4.2.2 串的块链存储结构



设置尾指针便于联结操作

```
#define CHUNKSIZE  4          //由用户定义块大小
typedef  struct  Chunk{
        char  ch[CHUNKSIZE];
        struct Chunk * next;
}Chunk;
typedef  struct{
    Chunk * head,    * tail;    //串的头、尾指针
     int    curlen;              //串的当前长度
}LString;
```

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

4.3 串的模式匹配算法

- 定义

如何在字符串数据中，检测和提取以字符串形式给出的某一局部特征，是各种串处理系统中最重要的操作之一。常见形式有：

- 模式检测：只关心是否存在匹配而不关心具体位置，如垃圾邮件检测。
- 模式定位：需具体具体的匹配位置。
- 模式计数等：关心匹配子串的总数，比如热词统计等。

- 模式匹配函数
  - Index(S,T,pos)

  返回子串T在主串S中第pos个字符之后第一次出现的位置，不存在，返回0.

简单模式匹配算法 (Brute Force)

- 如何匹配?
- 例子
  - ✓ 主串

    ababcabcacbab
  - ✓ 模式

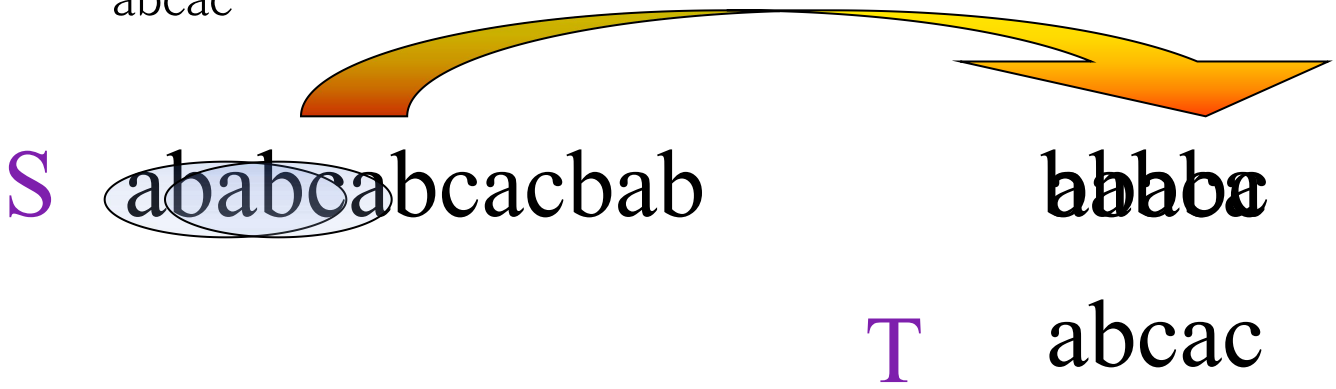
    abcac

简单模式匹配算法 (Brute Force)

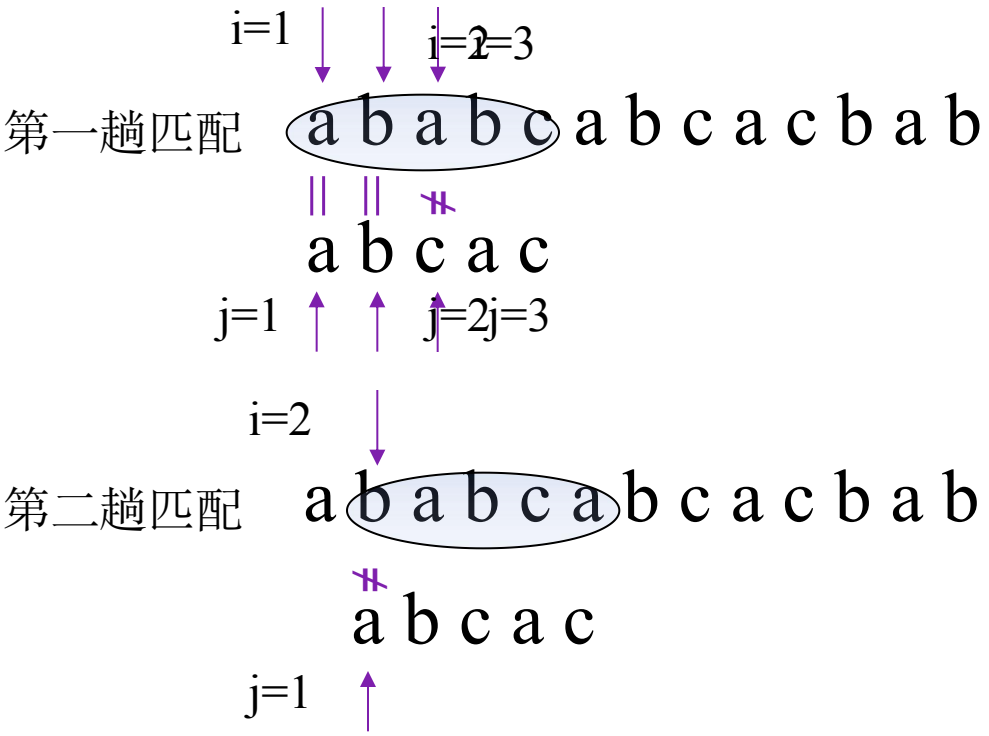
- 如何匹配?
- 例子
  - ✓ 主串

    ababcabcacbab
  - ✓ 模式

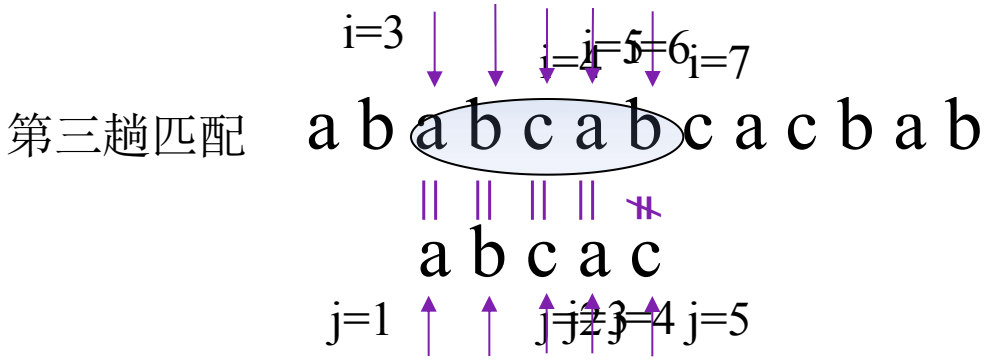
    abcac



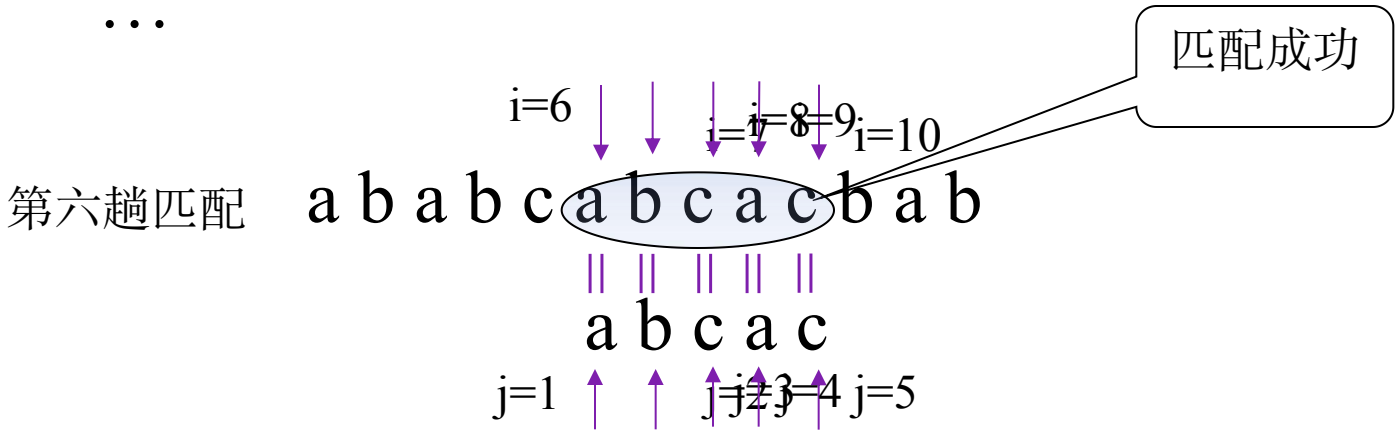
简单模式匹配算法 (Brute Force)



简单模式匹配算法（Brute Force)



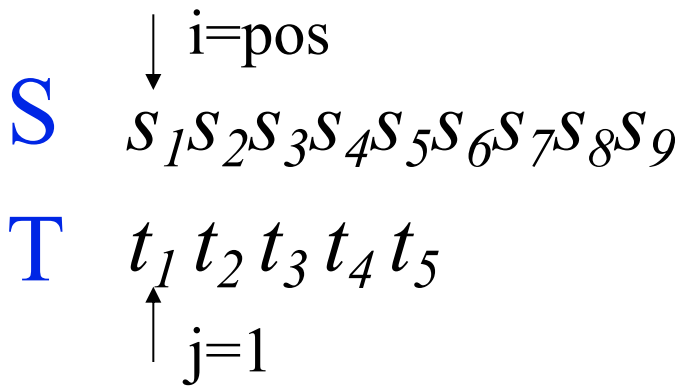
…



简单模式匹配算法 (Brute Force)

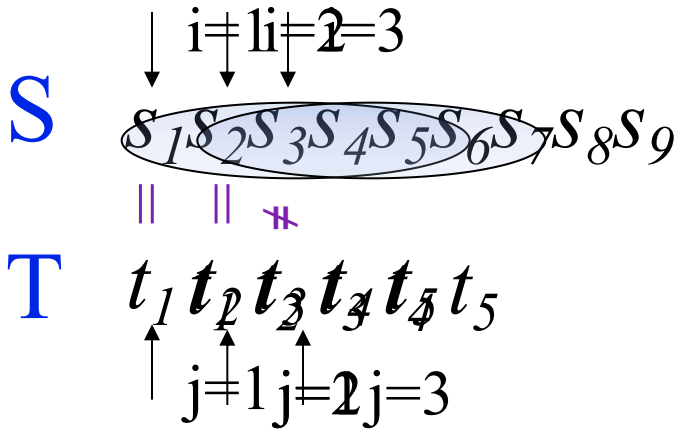
```
int Index(SString S, SString T, int pos) {
i = pos;   j = 1;
```

假定采用定长顺序存储，
即第0个位置存储长度



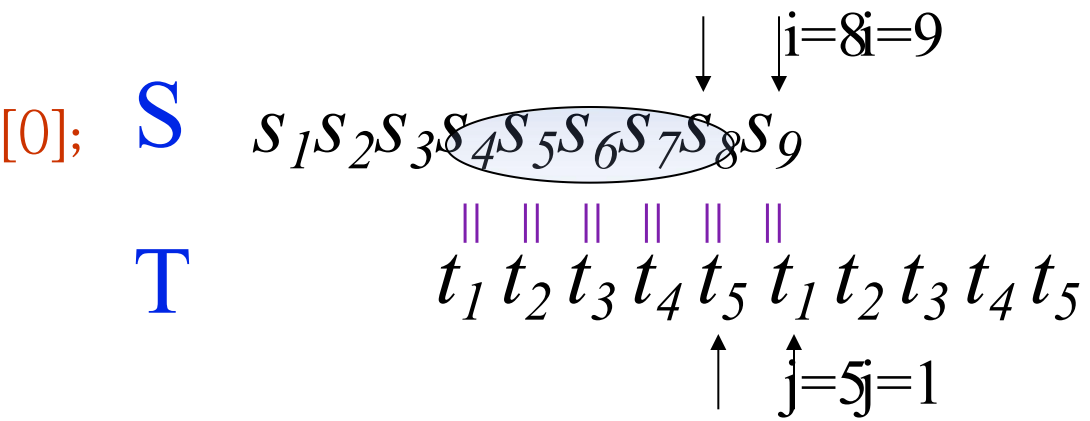
简单模式匹配算法 (Brute Force)

```
int Index(SString S, SString T, int pos) {
i = pos;   j = 1;
while ( 循环条件 ) {
  if (S[i] == T[j]) { ++i;  ++j; }// 继续比较后继字符
  else { i = i-j+2;   j = 1; } // 指针后退重新开始匹配
}
```



简单模式匹配算法 (Brute Force)

```
int Index(SString S, SString T, int pos) {
 i = pos;   j = 1;
 while (i <= S[0] && j <= T[0]) {
  if (S[i] == T[j]) { ++i;  ++j; }// 继续比较后继字符
  else { i = i-j+2;   j = 1; } // 指针后退重新开始匹配
 }
 if (j > T[0])  return  i-T[0];
 else return 0;
} // Index
```



算法性能分析

- 该匹配过程易于理解，且在某些应用场合，效率也较高
- 设串s长度为n，串t长度为m。
- 在好的情况下，每趟不成功的匹配都发生在第一对字符比较时
  - 例如： s =“aaaaaaaaaabc” t=“bc”

- 分析

  设匹配成功发生在s_i处，则在前面i-1趟匹配中共比较i-1次，第i趟成功匹配时比较了m次，所以总共比较i-1+m次。

算法性能分析

- 分析　“在好的情况下”

所有匹配成功的可能共有n-m+1种，假设是等概率的，那么在s_i匹配成功的概率是p_i=1/(n-m+1)。因此好的情况下的平均比较次数是：

$$\sum_{i=1}^{n-m+1} p_i \times (i-1+m) = \sum_{i=1}^{n-m+1} \frac{1}{n-m+1} \times (i-1+m) = \frac{(n+m)}{2}$$

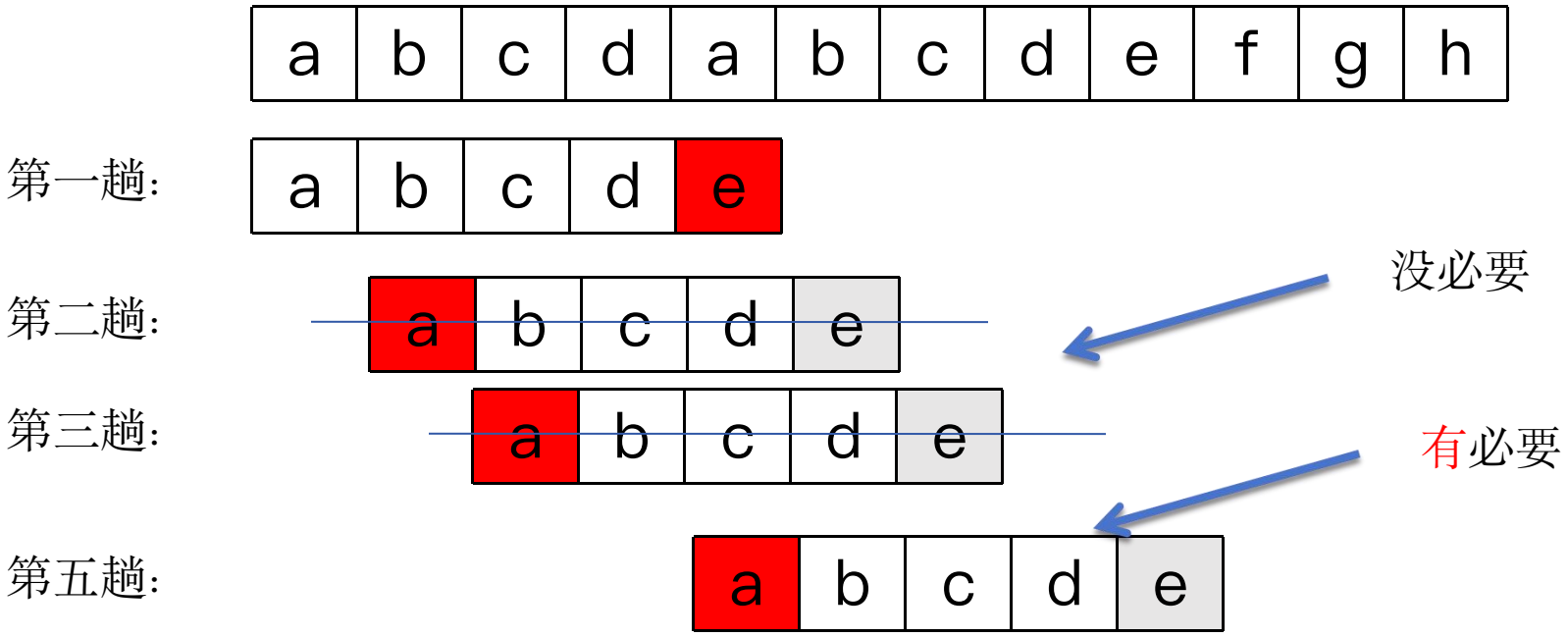
即匹配成功的好的情况的算法的时间复杂度为O(n+m)。

- 在坏的情况下，每趟不成功的匹配都发生在t的最后一个字符。
- 例如 s：“aaaaaaaaaaab”，t：“aaab”时
- 分析
  - 设匹配成功发生在s_i处，则在前面i-1趟匹配中共比较（i-1）*m次，到第i趟成功匹配共比较i*m次。所有匹配成功的可能共有n-m+1种，假设是等概率的。

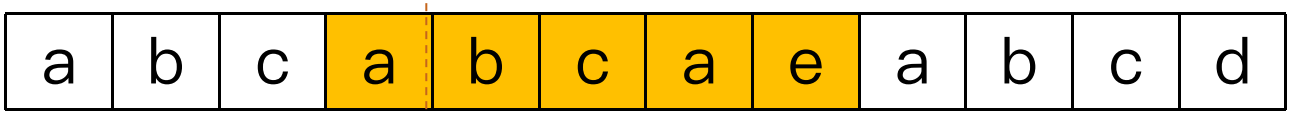
$$\sum_{i=1}^{n-m+1} p_i \times (i \times m) = \sum_{i=1}^{n-m+1} \frac{1}{n-m+1} \times (i \times m) = \frac{m \times (n-m+2)}{2}$$

  - 可见算法在坏的情况下的时间复杂度为O(n*m)。

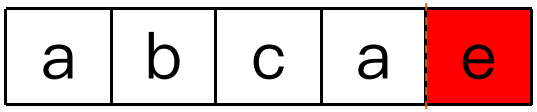
- 时间复杂度高的原因
  - 在主串中可能存在多个和模式串“部分匹配”的子串，因而引起指针i的多次回溯。
- 改进方法：
  - 指针i不回溯，模式向右滑动尽量远
  - 例如：s=“abcdabcdefgh”，t=“abcde”



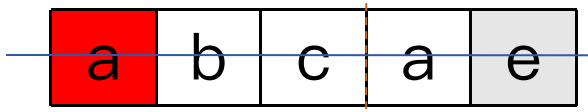
例如： s=“abcabcaeabcd”， t=“abcae”



第一趟：

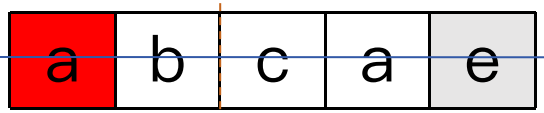


第二趟：

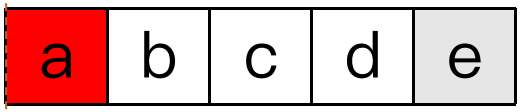


没必要

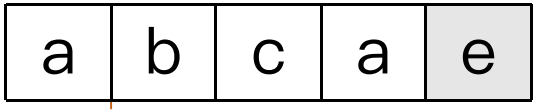
第三趟：



直接跳到第五趟???



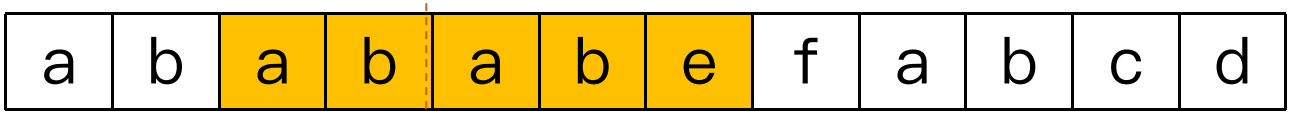
应该直接跳到这一步



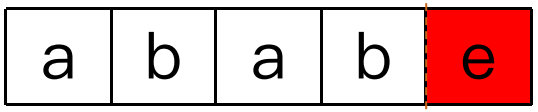
问题的关键：看看模式串中元素e前面有没有最后几个字符是和模式串开头的几个字符是重复的

e前面最后一个字符为“a”等于模式串开头的第一个字符“a”

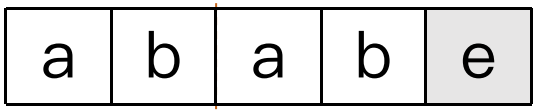
例如： s=“abababefabcd”， t=“ababe”



第一趟：

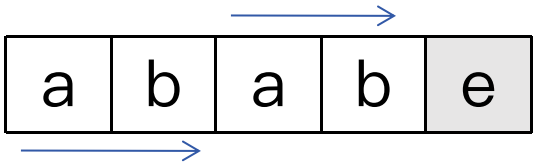


下一次从哪儿开始比？

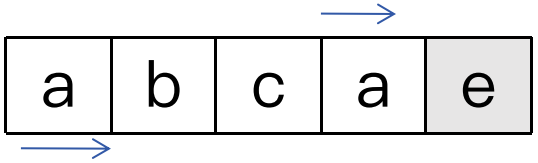


j=3

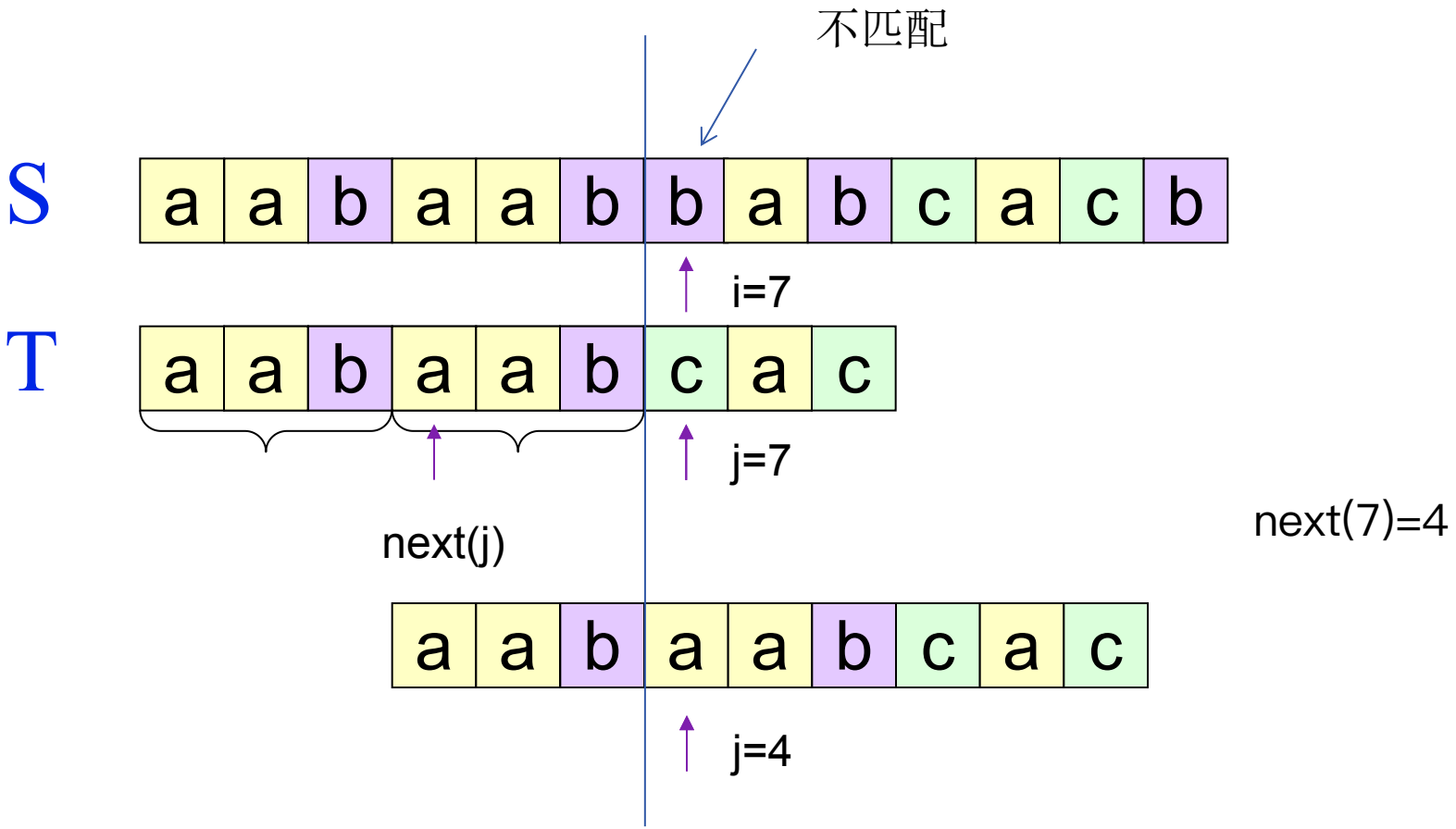
问题的关键：看看模式串中元素e前面有没有**最后几个字符**是和模式串**开头的几个字符**是重复的



e前面最后两个字符为“ab”等于模式串开头的两个字符“ab”



改进算法-KMP


不匹配
S
a a b a a b b a b c a c b
i=7
T
a a b a a b c a c
j=7
next(j)
next(7)=4
a a b a a b c a c
j=4

引入next数组

next[j]=k：k是当模式(子串)中第j个字符与主串中相应字符“失配”时，在模式中需重新和主串中该字符进行比较的字符的位置。

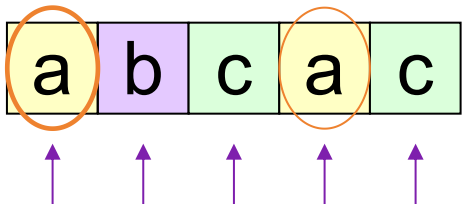
$$
\text{next}[j]=\begin{cases} 0 & \text{当 } j=1 \text{ 时 } \{\text{代表下一趟比较i=i+1，j=1}\} \\ \max\{k \mid 1<k<j \text{ 且 } `p_1 \ldots p_{k-1}`=`p_{j-k+1} \ldots p_{j-1}`\} & \{\text{下一趟比较i不变，j=k}\} \\ 1 & \text{其它情况(即}j\neq 1\text{且上述集合为空)} \{\text{下一趟比较i不变，j=1}\} \end{cases}
$$

前k−1个和后k−1个元素一致

举例

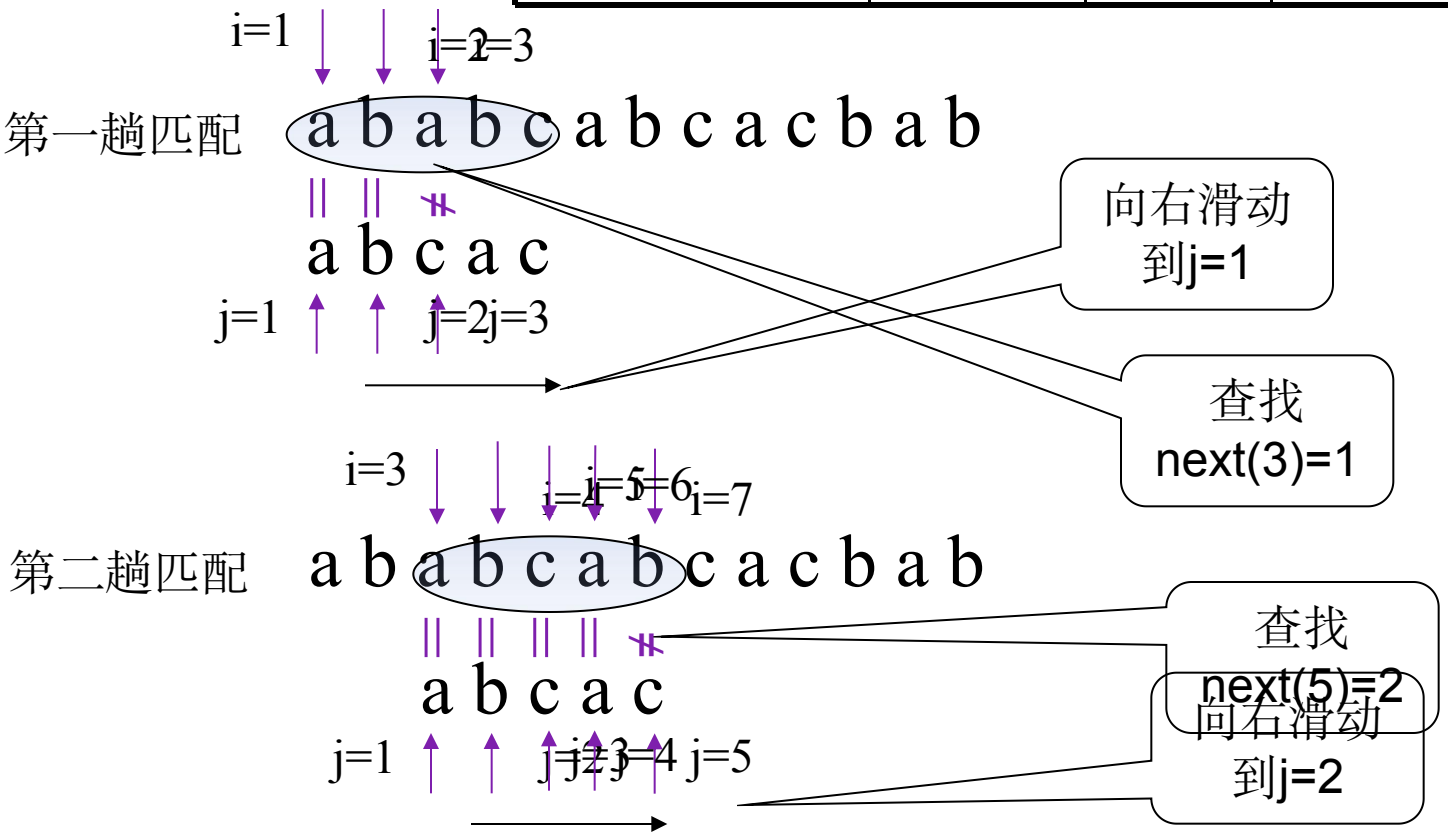
$$next[j]=\begin{cases}0, & \text{当}j=1\text{时}\\ \max\{k \mid 1<k<j \text{ 且前}k-1\text{个和后}k-1\text{个元素一致}\}\\ 1, & \text{其它情况}\end{cases}$$

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



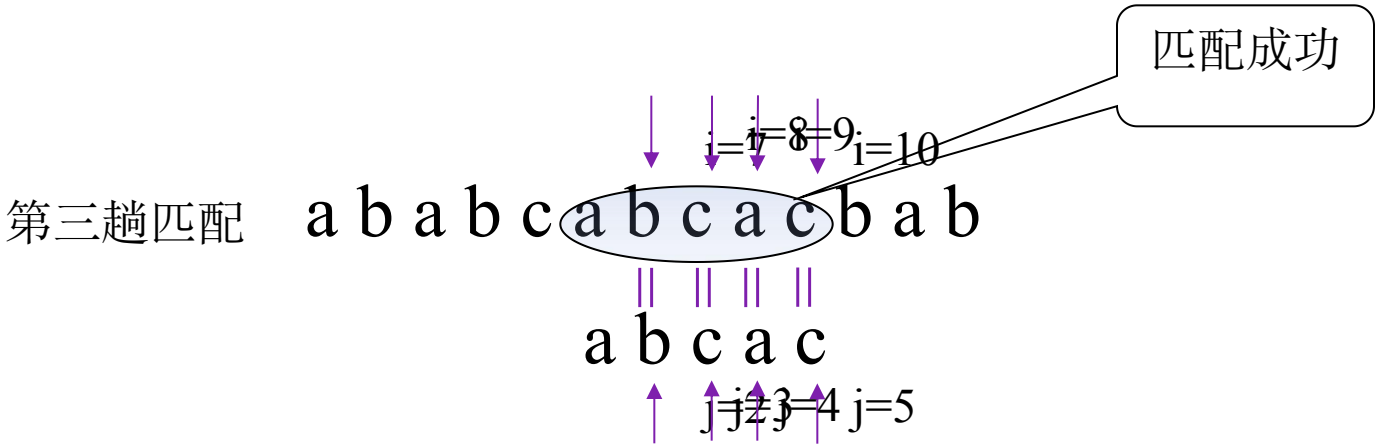
举例

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

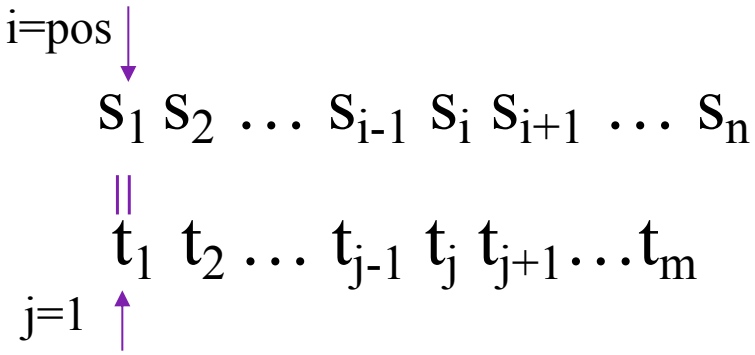


举例

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

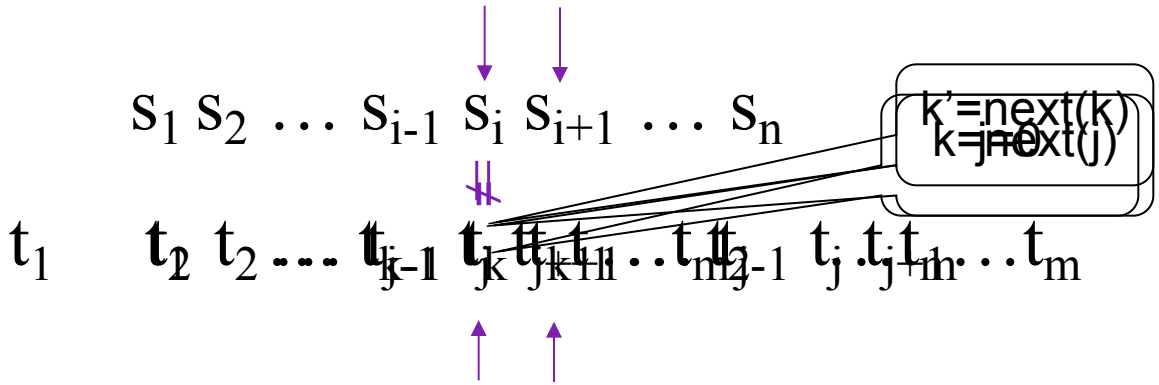


- 求得模式的next函数后，匹配可如下进行：
  - 假设以指针i和j分别指示主串和模式中正待比较的字符；
  - 令i的初值为pos，j的初值为1。



如果匹配继续向后比较

如果不匹配，则i不变，j退到k=next(j)位置再比较



```
int Index_KMP(SString S, SString T, int pos) {
  i = pos;   j = 1;
 while (i <= S[0] && j <= T[0]) {
      if (j = =  0 || S[i] == T[j]) { ++i;  ++j; } // 继续比较后继字符
      else  j = next[j];        // 模式串向右移动
  }//while
  if (j > T[0])  return  i-T[0];    // 匹配成功
  else return 0;
} // Index_KMP
```

改进算法-KMP

- KMP算法
  - 这种改进算法是D.E.Knuth、V.R.Pratt和J.H.Morris 同时发现的，因此人们称它为克努特—莫里斯—普拉特操作（简称为KMP算法）。此算法可以在O（n+m)的时间数量级上完成串的模式匹配操作。
- 改进
  - 每当一趟匹配过程中出现字符比较不等时，不需回溯i指针，而是利用已经得到的“部分匹配”的结果将模式向右“滑动”尽可能远的一段距离后，继续进行比较。

KMP-小结

(1) 简单模式匹配算法性能较低的根源

比到不相等时主串指针 i=i−j+2, 模式串指针j=1

主串指针回溯是不必要的

(2) 改进思路

主串指针i不回溯，模式串指针j也尽量不从1开始，尽量多跳过一些不必要的比较，设next[j] = k

(3) k的计算

当j等于1时，k=0；否则分析j前面的部分，看有没有正序x位和倒序x位相同的情况，如果没有这种情况k=1，如果有多于一种，则k等于x的最大值+1

KMP-小结

(4) **next函数的修正**

设主串S为'aaabaaaab' ,模式串T为'aaaab',

根据上一条规则，计算next[j]

j	1	2	3	4	5
next[j]	0	1	2	3	4

当i=4, j=4时，比较S[4] != T[4]，此时查next[j]，应该让j=3，即比较S[4]和T[3], 但是因为T[3]和T[4]是相等的，所以T[3]肯定也是不等于S[4]的，即这次比较也是不必要的

修正策略：当T[j]=T[next[j]]时需要，需要将next[j] 修正为next[next[j]]， 依次类推

KMP-小结

(4) next函数的修正

设主串S为'aaabaaaab' ,模式串T为'aaaab',

修正策略：当T[j]=T[next[j]]时需要，需要将next[j] 修正为next[next[j]]，依次类推

j	1	2	3	4	5
next[j]	0	1	2	3	4
nextval[j]	0	0	0	0	4

总结和思考

- 模式匹配定义
- 简单模式匹配算法
- 改进算法-KMP
- 思考
  - ✓ 模式匹配中，如模式带有通配符该如何匹配？比如用aaa??*b作为模式，其中？是任何字符，*是任何长度的字符串，可能为空。
  - ✓ Boyer-Moore算法，坏字符、好后缀，最好时间复杂度O(n/m)
  - ✓ Karp-Rabin算法：求子串的“指纹”特征，然后进行指纹比较。
  - ✓ 多模式匹配：Trie树（字典树）、AC自动机

4.4 串应用示例——文本编辑

[分析]

- 操作对象

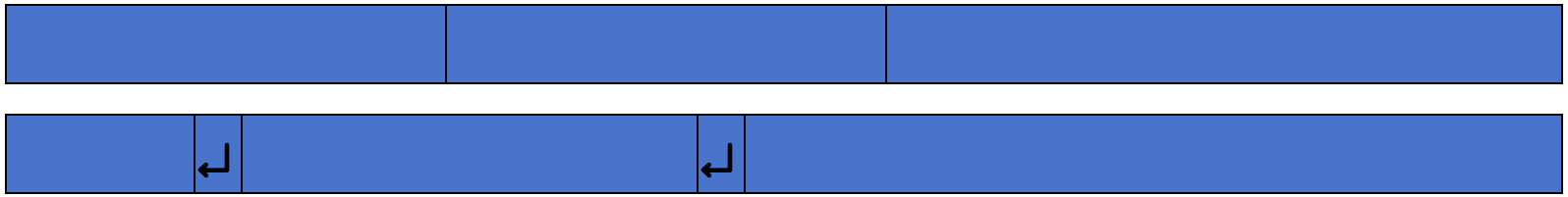
  ——文本串

  (行是文本串的子串)
- 基本操作
  - 查找
  - 插入
  - 删除

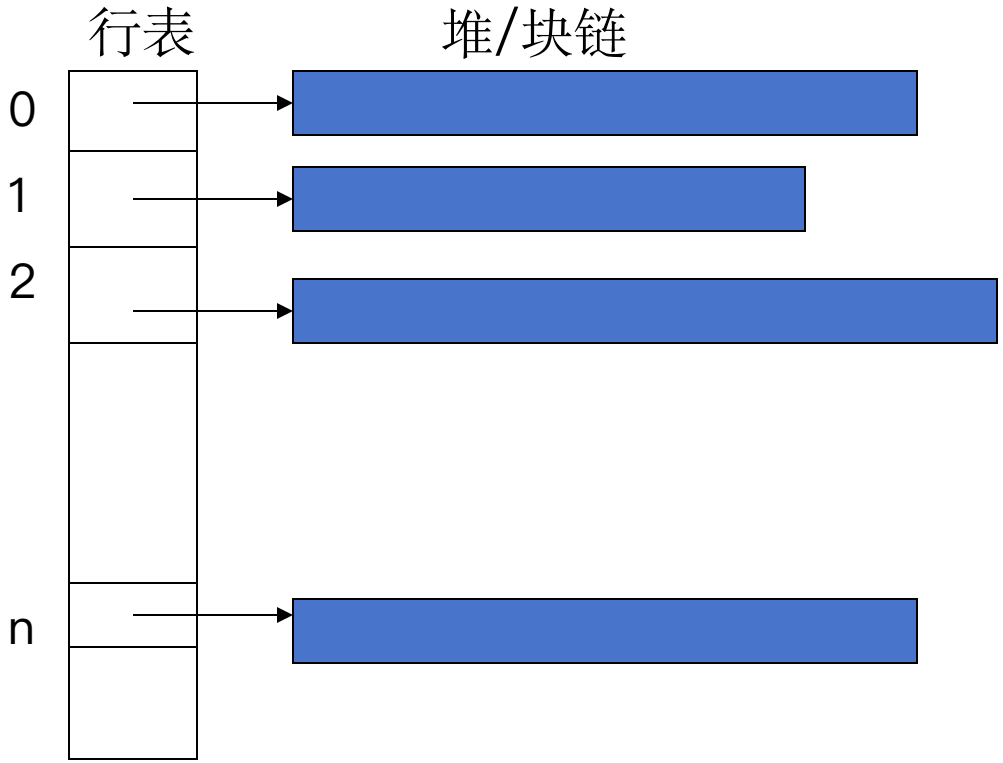
```
#include <stdio.h>
int main (void)
{
    printf("Hello world!\n");
    return 0;
}
```

[存储结构选择]

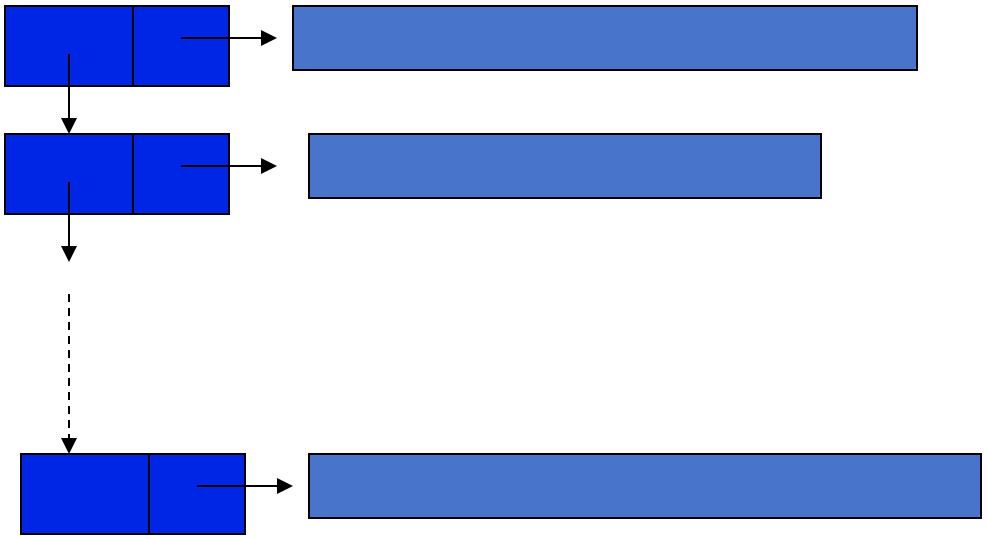
方案一　简单顺序存储

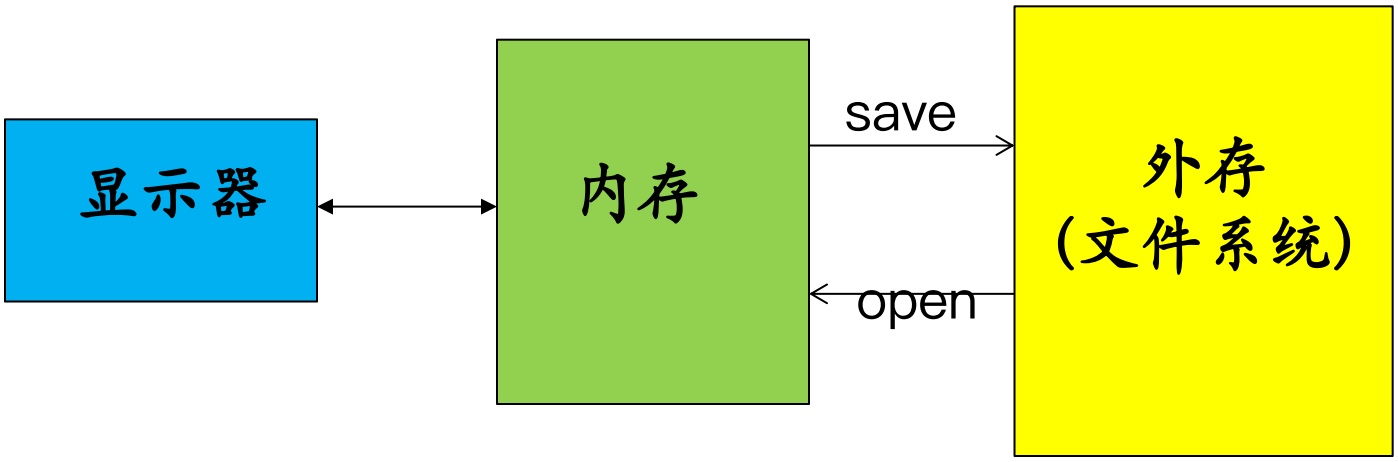


方案二



- 方案三





4.5 本章知识点小结

- 字符串线性结构的特点
- 字符串的顺序存储
- 字符串的链式存储
- 字符串的基本操作
- KMP模式匹配