



第4章 串

- 内容：
 - 串的定义、相关术语
 - 串的三种存储表示和基本操作实现
 - 串的模式匹配算法
 - 串的应用实例
- 重难点：
 - 串的存储表示和基本操作实现
 - 串的模式匹配算法



4.1 串的基本概念

- **串**：由0或 n ($n \geq 0$) 个字符组成的有限序列，记为 $s = 'a_1a_2a_3 \dots a_n'$
 - 串名： s
 - 串值： $a_1a_2a_3 \dots a_n$
 - 串长： n
- **子串**：串中任意连续字符组成的子序列。
 - **任意串是其自身的子串**
- **主串**：包含子串的串。
- **位置**：字符在串中的序号称为该字符在串中的位置。
- **空格串**：由一个或多个空格组成的串，长度为空格个数。
- **空串**：零个字符的串，用 Φ 表示。
 - **空串是任意串的子串**
- **串相等**：两个串长度相等且各个对应位置的字符也相等。



举例

- 设 $S_1='BEI'$, $S_2='JING'$, $S_3='BEIJING'$, $S_4='BEI JING'$ 四个串, 请回答如下问题:
 - S_1 的串值为 BEI, 长度为 3。
 - S_1, S_2 均是 S_3, S_4 的子串 ✓
 - S_1 在 S_3, S_4 的位置相同 ✓
 - S_2 在 S_3, S_4 的位置相同 ✗



串的抽象数据类型定义

ADT String {

数据对象: $D = \{a_i \mid a_i \in \text{CharacterSet}, i = 1, 2, \dots, n, n \geq 0\}$

数据关系: $R_1 = \{ \langle a_{i-1}, a_i \rangle \mid a_{i-1}, a_i \in D, i = 1, 2, \dots, n \}$

基本操作:

StrAssign(&T, chars), DestroyString(&S)

StrEmpty(S), ClearString(&S),

StrInsert(&S, pos, T), StrDelete(&S, pos, len)

StrLength(S),

StrCopy(&T, S), StrCompare(S, T)

Index(S, T, pos), Substring(&Sub, S, pos, len)

Concat(&T, S1, S2), Replace(&S, T, V)

} ADT String



串与线性表的共同点和区别

- 共同点：串的逻辑结构与线性表相似
- 区别：
 - 串的数据对象约束为字符集，线性表的数据对象不受限
 - 串以整体为操作对象，线性表以单个元素为操作对象

- 串比较StrCompare
- 求串长StrLength
- 串拷贝StrCopy
- 串判空StrEmpty
- 求子串SubString
- 串定位Index

- 串清空ClearString
- 串联接Concat
- 串替换Replace
- 串插入StrInsert
- 串删除StrDelete

- 串生成StrAssign
- 串销毁DestroyString



串操作的实现-定位函数Index(S,T,pos)

- 初始条件: S、T存在, T不空, $1 \leq \text{pos} \leq \text{StrLength}(S)$
- 操作结果: 如果S中有与T相同的子串, 返回其在S中第pos个位置后第一次出现的位置, 否则返回0。
- 思想: 求 $\text{StrCompare}(\text{SubString}(\&\text{sub}, S, i, \text{StrLength}(T)), T) == 0$ 的i。
- 步骤:
 - 1) 在S中取从第i($\text{pos} \leq i \leq \text{StrLength}(S) - \text{StrLength}(T) + 1$)个字符起、**长度**和T相等的**子串**sub
 - 2) sub和T**比较**, 若相等, 则返回i, 否则i++继续比较直至S中不存在和T相等的子串。
 - **求串长**StrLength()
 - **求子串**SubString()
 - **判等**StrCompare()



串操作的实现-定位函数Index(S,T,pos)

```
int index(String S,String T,int pos)
{ //返回主串S中第pos个字符之后与T相等的子串的位置， 没有返回0
    if(pos>0)
    {   n=StrLength(S);   m=StrLength(T);   i=pos;
        while(i<=n-m+1)
        {   SubString(sub,S,i,m); //sub返回S中第i个位置起长m的子串
            if(StrCompare(sub,T)!=0) ++i; //串不等,位置后移
            else return i; //返回子串在T中的位置
        } //while
    } //if
    return 0; //如S中无与T相等的子串
} //index
```



4.2 串的实现

4.2.1 定长存储结构 ★★

4.2.2 堆分配存储结构 ★★★★★

4.2.3 块链存储结构 ★





4.2.1 串的定长存储表示

- 特点：用一组地址连续的存储单元存储串值的字符序列。
- 用C语言实现串的定长存储结构

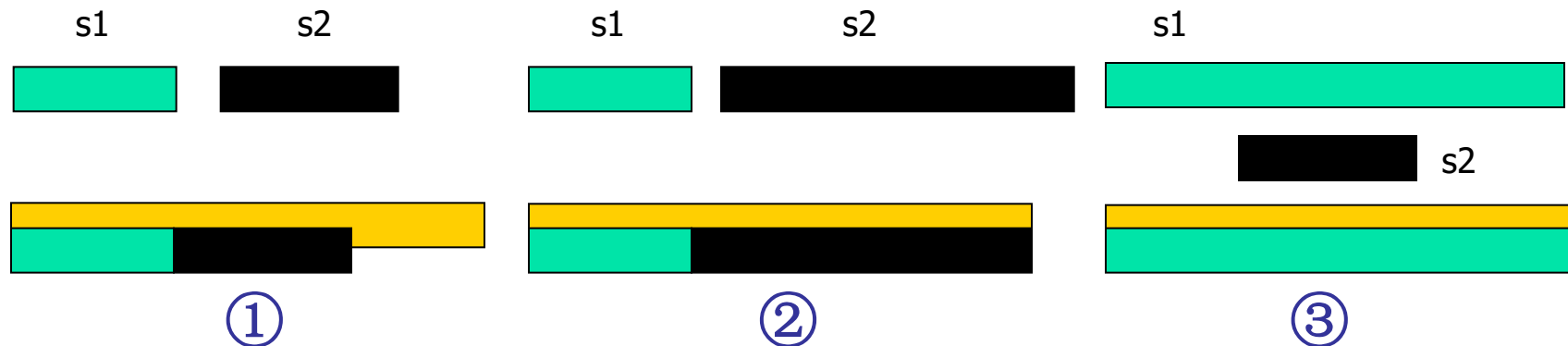
```
#define MAXSTRLEN 255 //最大串长
```

```
typedef unsigned char SString[MAXSTRLEN+1];
```

- 串截断
- 串长的表示方法
 - 以下标为 0 的数组分量存放串长 - PASCAL
 - 串值后加一个不计入串长的结束标记字符 - C中用'\0'
- 串的定长存储表示下各基本操作的实现

例 串联接Concat(&T,S1,S2)

- 要求：T,S1,S2都是SString型的串变量，用T返回S1和S2联接的新串。
- 算法思想：串T值产生有三种情况：
 - ① $S1[0]+S2[0] \leq \text{MAXSTRLEN}$
 - ② $S1[0] < \text{MAXSTRLEN}$ 但 $S1[0]+S2[0] > \text{MAXSTRLEN}$
 - ③ $S1[0] = \text{MAXSTRLEN}$



Status Concat(SString &T , SString S1, SString S2)

{//用T返回S1和S2联接的串。若未截断, 返回TRUE, 否则返回FALSE。

if (S1[0]+S2[0] <= MAXSTRLEN)

{//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] < MAXSTRSIZE)

{ //S1[0] <MAXSTRLEN, S1[0]+S2[0]> MAXSTRLEN截断

T[1..S1[0]] = S1[1..S1[0]];

T[S1[0]+1..MAXSTRLEN] = S2[1..MAXSTRLEN - S1[0]];

T[0] = MAXSTRLEN; uncut = FALSE;

}

else// S1[0] = MAXSTRSIZE截断(仅取S1)

{ T[0..MAXSTRLEN] = S1[0..MAXSTRLEN];//T[0]=S1[0]= MAXSTRLEN

uncut = FALSE;

}

return uncut;

} // Concat



求子串SubString(Sub, S, pos, len)

- 要求：Sub,S是SString型的串变量，求S中从第pos个字符起长度为len的子串存入Sub。

```
Status SubString(SString &Sub,SString S,int pos,int len)
```

```
{//用Sub返回S的第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;//pos或len的位置不合法
```

```
    Sub[1..len]=S[pos..pos+len-1];//串复制
```

```
    Sub[0]=len;
```

```
    return OK;
```

```
}//SubString
```



串赋值StrAssign(&T,chars)

```
Status StrAssign(SString &T, const char *chars)
{
    len=strlen(chars);
    if (len>MAXSTRLEN)
    {
        T[0]=MAXSTRLEN;
        uncut=FALSE;
    }
    else
    {
        T[0]=len;
        uncut=TRUE;
    }
    for (int i=1;i<=T[0];i++)
        T[i]=chars[i-1];
    return uncut;
}
```

串复制StrCopy(&T,S)

Status StrCopy(SString &T, SString S)

{**//S是源串, T是目的串**

for(i=0;i <=S[0];++i) **//字符序列的复制**

T[i] = S[i];

return OK;

}

串比较StrCompare(S,T)

int StrCompare (SString S, SString T)

{

for (int i=1;i<=S[0]&& i<=T[0];i++)

if (S[i]!=T[i])

return S[i]-T[i];

return S[0]-T[0];

}

串插入StrInsert(&S, pos, T)

Status StrInsert(SString S, int pos,SString T)

```
{  if(pos<1||pos>S[0]+1)
    return ERROR;
  if(S[0]+T[0]<=MAXSTRLEN)
  {    for(i=S[0];i>=pos;i--)
        S[i+T[0]]=S[i];
      for(i=pos;i<pos+T[0];i++)
        S[i]=T[i-pos+1];
      S[0]=S[0]+T[0];
      return TRUE;
  }
  else
  {    for(i=MAXSTRLEN;i>=pos;i--)
        S[i]=S[i-T[0]];
      for(i=pos;i<pos+T[0];i++)
        S[i]=T[i-pos+1];
      S[0]=MAXSTRLEN;
      return FALSE;
  }
}
```

串删除StrDelete(&S,pos,len)

```
Status StrDelete(SString S,int pos,int len)
{
    if(pos<1||pos>S[0]-len+1||len<0)
        return ERROR;
    for(i=pos+len;i<=S[0];i++)
        S[i-len]=S[i];
    S[0]-=len;
    return OK;
}
```

销毁DestroyString(&S)

```
Status DestroyString(SString S)
{ //定长分配的存储空间无法销毁 }
```




串替换Replace(S, T, V)

```
Status Replace(SString S, SString T, SString V)
{ //用V替换S中出现的所有与T相等的不重叠子串
    int i=1;
    if(StrEmpty(T))    return ERROR;
    do
    {
        i=index(S,T,i);
        if(i)
        {
            StrDelete(S,i,StrLength(T));
            StrInsert(S,i,V);
            i+=StrLength(V);
        }
    }while(i);
    return OK;
}
```



其他几个操作

串判空StrEmpty(T)

```
Status StrEmpty(SString T)
{ return (T[0]==0); }
```

求串长StrLength(T)

```
int StrLength(SString T)
{ return T[0]; }
```

串清空ClearString(S)

```
Status ClearString(SString &T)
{ T[0]=0; //StrDelete(T,1,T[0])
  return OK;
}
```



串的定长存储表示的小结

- 实现串操作的方法：字符序列的复制
- 操作的算法时间复杂度：基于串长
- 缺点：“截尾”

4.2.2 串的堆分配存储表示

- 特点：以一组地址连续的存储单元（在程序执行过程中动态分配）存放串值字符序列。
- 串的堆分配存储（C语言实现）

```
typedef struct {  
    char *ch;    //非空串按串长分配存储区，否则为NULL  
    int length;  // 串长度  
} HString;
```

- 串操作的基本思想：
 - 为新生成的串**分配**存储空间
 - 串值**复制**。





串插入StrInsert(&S,pos,T)

```
Status StrInsert(HString &S,int pos,HString T)
{  if(pos<1 || pos>S.length+1) 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)*sizeof(char)))
       exit(overflow);
      for(i=S.length-1;i>pos-1;--i) //为插入T腾出位置
        S.ch[i+T.length]=S.ch[i];
      S.ch[pos-1..pos+T.length-2]=T.ch[0..T.length-1];
      S.length+=T.length;
    }
   return OK;
}
```



串复制StrCopy(&T,S)

```
Status StrCopy(HString &T, HString S) //将串S复制到串T
{
    if(T.ch) free(T.ch);
    if(!S.length)//若S为空串
    { T.ch=NULL; T.length=0;}
    else
    {
        if(!(T.ch=(char*)malloc(S.length*sizeof(char)))
            exit(OVERFLOW);//为T分配S长度的空间
        for(i=0;i <S.length;++i)//字符序列的复制
            T.ch[i] = S.ch[i];
        T.length=S.length;
    }
    return OK;
}
```



串判空StrEmpty(HString s)

```
Status StringEmpty(HString S)
```

```
{
```

```
    if (!S.length) //if(!StrLength(s))
```

```
        return TRUE;
```

```
    else
```

```
        return FALSE;
```

```
}
```



串删除StrDelete(&S,pos,len)

```
Status StrDelete(HString &S,int pos,int len)
{  if(pos<1||pos>S.length||len<0||len>S.length-pos+1)
    return ERROR;
    for(i=pos-1;i<=S.length-len;i++)
        S.ch[i]=S.ch[i+len];
    S.length-=len;
    S.ch=(char*)realloc(S.ch,S.length*sizeof(char));
    return OK;
}
```




串替换Replace(S, T, V)

Status Replace(HString &S,HString T,HString V)

{ if(StrEmpty(T))

return ERROR;

do

{ i=index(S,T, i);

if(i)

{ StrDelete(S,i,StrLength(T));

StrInsert(S,i,V);

i+=StrLength(V);

}

}while(i);

return OK;

}

其它操作的实现见书上P76-77

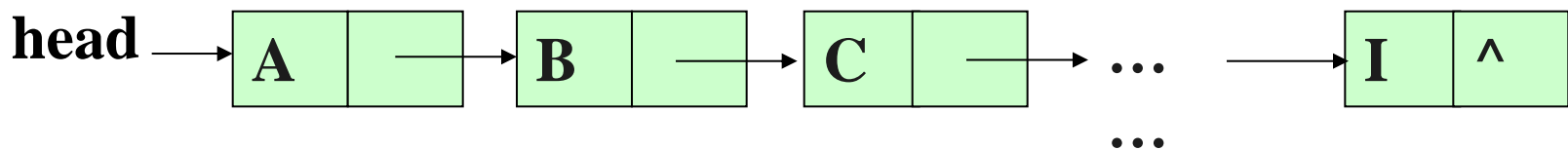


串的堆分配存储表示小结

- 存储空间动态分配
- 实现串操作的方法：字符序列的复制
- 操作的算法时间复杂度：基于串长
- 优点：
 - 有顺序存储结构的特点；
 - 处理方便；
 - 操作中对串长没有限制。

4.2.3 串的块链存储表示

- 特点：以一组存储单元（在程序执行过程中动态分配）存放串值字符序列-串的链表表示。



结点大小为1的链表



结点大小为4的链表



4.2.3 串的块链存储表示

- 串的块链存储表示 (C语言描述)

```
#define CHUNKSIZE 80 // 可由用户定义的块大小
typedef struct Chunk { // 结点结构
    char ch[CHUNKSIZE];
    struct Chunk *next;
} Chunk;
typedef struct { // 串的链表结构
    Chunk *head, *tail; // 串的头和尾指针, 便于联结操作
    int curlen; // 串的当前长度
} LString;
```



4.2.3 串的块链存储表示小结

- 串的块链存储结构-串的链表表示。
- 除了特定操作如联接比较方便，不如其他存储结构。
- 结点大小为1
 - 优点: 操作方便;
 - 缺点: 存储密度较低，占用存储量大。
- 结点大小为4
 - 优点: 存储密度高;
 - 缺点: 插入、删除字符时，可能会引起结点之间字符的移动，算法实现比较复杂。

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



4.3 串的模式匹配算法

- **模式匹配：**子串T在主串S中的定位操作。
- **目标：**主串S
- **模式：**子串T(模式串)
- **匹配结果：**
 - **匹配成功：**S中有T时，返回T在S中的位置，当S中有多个T，通常只找出第一个。
 - **匹配失败：**若S中无T，返回值为0。
- **方法：**
 - **BF(Brute-Force)算法：**经典、朴素、暴力穷举的。
 - **KMP算法：**速度快。



BF算法匹配过程

第一趟 主串 $\overset{i=1}{\downarrow}$ a c a b a a b a a b c a c a a b c

模式串 a b a a b c a c \uparrow j=1



BF算法匹配过程

$i=2$
↓
第一趟 主串 **a** c a b a a b a a b c a c a a b c
模式串 **a** b a a b c a c
↑
 $j=2$



BF算法匹配过程

$i=2$
↓
第二趟 主串 a c a b a a b a a b c a c a a b c
模式串 a b a a b c a c
 ↑
 j=1



BF算法匹配过程

第三趟 主串 $i=3$
 ↓
 a c a b a a b a a b c a c a a b c
模式串 a b a a b c a c
 ↑
 $j=1$



BF算法匹配过程

第三趟 主串 a c **a** b a a b a a b c a c a a b c
模式串 **a** b a a b c a c

$i=4$
↓
 $j=2$
↑



BF算法匹配过程

第三趟 主串 a c **a b** a a b a a b c a c a a b c
模式串 **a b** a a b c a c

$i=5$
↓
 $j=3$
↑



BF算法匹配过程

第三趟 主串 a c **a b a** a b a a b c a c a a b c
模式串 **a b a** a b c a c

$i=6$
↓
 $j=4$ ↑



BF算法匹配过程

第三趟 主串 a c a b a a b a a b c a c a a b c
模式串 a b a a b c a c

$i=7$
↓
 $j=5$
↑



BF算法匹配过程

第三趟 主串 a c a b a a b a a b c a c a a b c
模式串 a b a a b c a c

$i=8$
↓
 $j=6$
↑

a b a a b c a c

第四趟

主串 a c a b a a b a a b c a c a a b c
 ↓
i=4

模式串 a b a a b c a c
 ↑
j=1

第五趟 主串 a c a b a a b a a b c a c a a b c
模式串 a b a a b c a c
 ↑
 j=1



BF算法匹配过程

第五趟 主串 a c a b **a** b a a b c a c a a b c
模式串 **a** b a a b c a c

$i=6$
↓
 $j=2$ ↑

第六趟 主串 a c a b a a b a a b c a c a a b c
模式串 a b a a b c a c
 ↑
 j=1



BF算法匹配过程

第六趟 主串 a c a b a **a** b a a b c a c a a b c
模式串 **a** b a a b c a c
 ↑
 j=2

i=7



BF算法匹配过程

第六趟 主串 a c a b a **a b** a a b c a c a a b c
模式串 **a b** a a b c a c

$i=8$
↓
 $j=3$ ↑



BF算法匹配过程

第六趟 主串 a c a b a **a b a** a b c a c a a b c
模式串 **a b a** a b c a c

$i=9$
↓
 $j=4$
↑



BF算法匹配过程

第六趟 主串 a c a b a **a b a a** b c a c a a b c
模式串 **a b a a** b c a c

$i=10$
↓
 $j=5$ ↑



BF算法匹配过程

第六趟 主串 a c a b a a b a a b c a c a a b c
模式串 a b a a b c a c

$i=11$
↓
 $j=6$ ↑



BF算法匹配过程

第六趟 主串 a c a b a a b a a b c a c a a b c
模式串 a b a a b c a c

i=12
↓
j=7
↑



BF算法匹配过程

第六趟 主串 a c a b a **a b a a b c a** c a a b c
模式串 **a b a a b c a c**

$i=13$
↓
 $j=8$ ↑



BF算法匹配过程

第六趟 主串 a c a b a a b a a b c a c a a b c
模式串 a b a a b c a c

i=14
↓
j=9
↑

BF算法匹配情况

设主串是ababcabcacbab，模式串是abcac，指针i和j分别指向主串和模式串中正在比较的字符。

第一趟匹配

a b a b c a b c a c b a b
a b c a c

第二趟匹配

a b a b c a b c a c b a b
a b c a c

第三趟匹配

a b a b c a b c a c b a b
a b c a c

第四趟匹配

a b a b c a b c a c b a b
a b c a c

第五趟匹配

a b a b c a b c a c b a b
a b c a c

第六趟匹配

a b a b c a b c a c b a b
a b c a c

int Index_BF(S, T, pos)

Status 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; }           // 匹配时，继续比较后继字符
        else
        { i = i - j + 2 ; j = 1; } // 失配时，修改主串和模式串指针重新匹配
    }
    if ( j > T[0] )
        return i - T[0];       // 匹配成功返回第一个匹配字符位置
    else
        return 0;              // 失败返回 0
}
```

BF匹配算法的时间复杂度分析

- 若主串长度 n ，模式串长度 m ，BF算法的时间复杂度：

- 对子串中少有重复字符的情况，例：

S: a string searching example consisting of simple text

T: sting

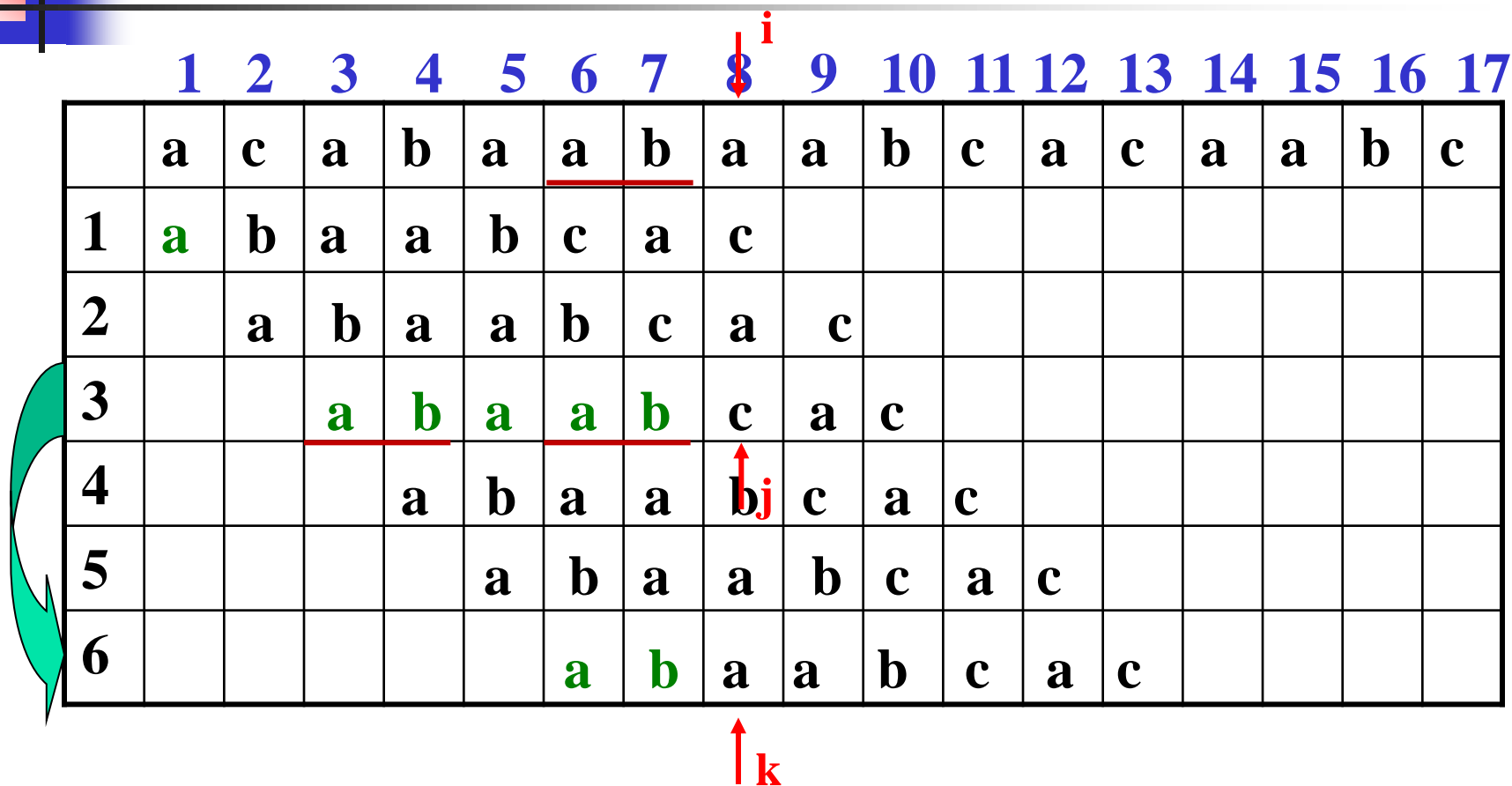
效率较高，一般情况下时间复杂度为 $O(n+m)$ 。

- 当模式串重复字符较多时，如：

S:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
T:	0	0	0	0	0	0	1										

- 最坏情况下需要比较字符的总次数为 $(n-m+1)*m$ ，时间复杂度 $O(n*m)$ 。

分析BF算法效率低的原因



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
	a	c	a	b	a	<u>a</u>	<u>b</u>	a	a	b	c	a	c	a	a	b	c
1	<u>a</u>	b	a	a	b	c	a	c									
2		a	b	a	a	b	c	a	c								
3			<u>a</u>	<u>b</u>	<u>a</u>	<u>a</u>	<u>b</u>	c	a	c							
4				a	b	a	a	<u>b</u>	c	a	c						
5					a	b	a	a	b	c	a	c					
6						<u>a</u>	<u>b</u>	a	a	b	c	a	c				

当主串与模式串失配时，模式串指针应尽量少回溯，不回溯



KMP算法

- D.E.Knuth, J.H.Morris 和 V.R.Pratt 发现。简称 KMP 算法。
- KMP 算法的思想
 - 利用已经得到的“部分匹配”结果将模式串向右“滑动”尽可能远，再重新进行新一趟的匹配。主串指针 i 不回溯，效率高，速度快。

主串与模式串匹配转化成模式串自身匹配

- 设 $s_i \neq t_j$ 时, s_i 应与模式串第 $k(k < j)$ 个字符比较, 根据 $s_i \neq t_j$:

$s_1 s_2 s_3 \dots s_{i-k+1} s_{i-k+2} \dots s_{i-1} s_i$
 $t_1 t_2 \dots t_{j-k+1} t_{j-k+2} \dots t_{j-1} t_j$

$\downarrow i$
 $\uparrow j$

$'t_{j-k+1}t_{j-k+2}\dots t_{j-1}' = 's_{i-k+1}s_{i-k+2}\dots s_{i-1}'$

- 根据 s_i 应与模式串第 $k(k < j)$ 个字符比较

$s_1 s_2 s_3 \dots s_{i-k+1} s_{i-k+2} \dots s_{i-1} s_i$
 $t_1 t_2 \dots t_{j-k+1} \dots t_{k-1} t_k \dots t_{j-1} t_j$

$\downarrow i$
 $\downarrow i$

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

- 主串与模式串的比较变换为模式串自身的比较:

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

当主串与模式串失配时，模式串指针应尽量少回溯

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
	a	c	a	b	a	<u>a</u>	<u>b</u>	a	a	b	c	a	c	a	a	b	c
			<u>a</u>	<u>b</u>	a	<u>a</u>	<u>b</u>	c	a	c							
								↑ j									
						a	b	a	a	b	c	a	c				
								↑ k									

应保持i不动，将j指向模式串第3个字符继续匹配

next函数定义

- 令 $next[j] = k$, 表示当模式中第 j 个字符与主串中相应字符“失配”时, 在模式串中需重新和主串中该字符进行比较的字符位置, $next$ 函数的定义:

当模式串第一个字符与主串对应字符失配时, 模式串不动

$$next[j] = \begin{cases} 0 & \text{当 } j=1 \text{ 时} \\ \max \{k \mid 0 < k < j, t_1 t_2 \dots t_{k-1} = t_{j-k+1} t_{j-k+2} \dots t_{j-1}\} & \text{当此集合不空} \\ 1 & \text{其他情况} \end{cases}$$

并非模式串第一个字符与主串对应字符失配, 也找不到 k , 从模式串第一个字符起重新匹配

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

KMP算法的比较过程

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

第一趟 主串 a c a b a a b a a b c a c a a b c
 ↓
 i=1

模式串 a b a a b c a c
 ↑
 j=1

KMP算法的比较过程

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

第一趟 主串 **a** c a b a a b a a b c a c a a b c
 模式串 **a** b a a b c a c
 j=2 next[2]=1

KMP算法的比较过程

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

第一趟 主串 a c a b a a b a a b c a c a a b c
 ↓
 i=2

模式串 a b a a b c a c
 ↑
 j=1 next[1]=0

KMP算法的比较过程

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

第一趟 主串 a c a b a a b a a b c a c a a b c
 ↓
 i=3

模式串 a b a a b c a c
 ↑
 j=1

KMP算法的比较过程

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

第一趟 主串 a c **a** b a a b a a b c a c a a b c
 ↓
 i=4

模式串 **a** b a a b c a c
 ↑
 j=2

KMP算法的比较过程

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

第一趟 主串 a c **a b** a a b a a b c a c a a b c
模式串 **a b** a a b c a c
i=5
j=3

KMP算法的比较过程

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

第一趟 主串 a c **a b a** a b a a b c a c a a b c
模式串 **a b a** a b c a c

$i=6$
↓
 $j=4$ ↑

KMP算法的比较过程

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

第一趟 主串 a c a b a a b a a b c a c a a b c
模式串 a b a a b c a c

i=7
↓
j=5
↑

KMP算法的比较过程

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

第一趟 主串 a c a b a a b a a b c a c a a b c

模式串 a b a a b c a c

$i=8$

$j=6$ next[6]=3

KMP算法的比较过程

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

第一趟 主串 a c a b a **a b** a a b c a c a a b c
模式串 **a b** a a b c a c

$i=8$
↓
 $j=3$ ↑

KMP算法的比较过程

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

第一趟 主串 a c a b a **a b a** a b c a c a a b c
模式串 **a b a** a b c a c

$i=9$
↓
 $j=4$ ↑

KMP算法的比较过程

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

i=10



第一趟 主串 a c a b a a b a a b c a c a a b c

模式串

a b a a b c a c



j=5

KMP算法的比较过程

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

i=11

第一趟 主串 a c a b a a b a a b c a c a a b c

模式串

a b a a b c a c

j=6

KMP算法的比较过程

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

i=12



第一趟 主串 a c a b a a b a a b c a c a a b c

模式串

a b a a b c a c



j=7

KMP算法的比较过程

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

i=13



第一趟 主串 a c a b a a b a a b c a c a a b c

模式串

a b a a b c a c



j=8

KMP算法的比较过程

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

i=14



第一趟 主串 a c a b a a b a a b c a c a a b c

模式串

a b a a b c a c



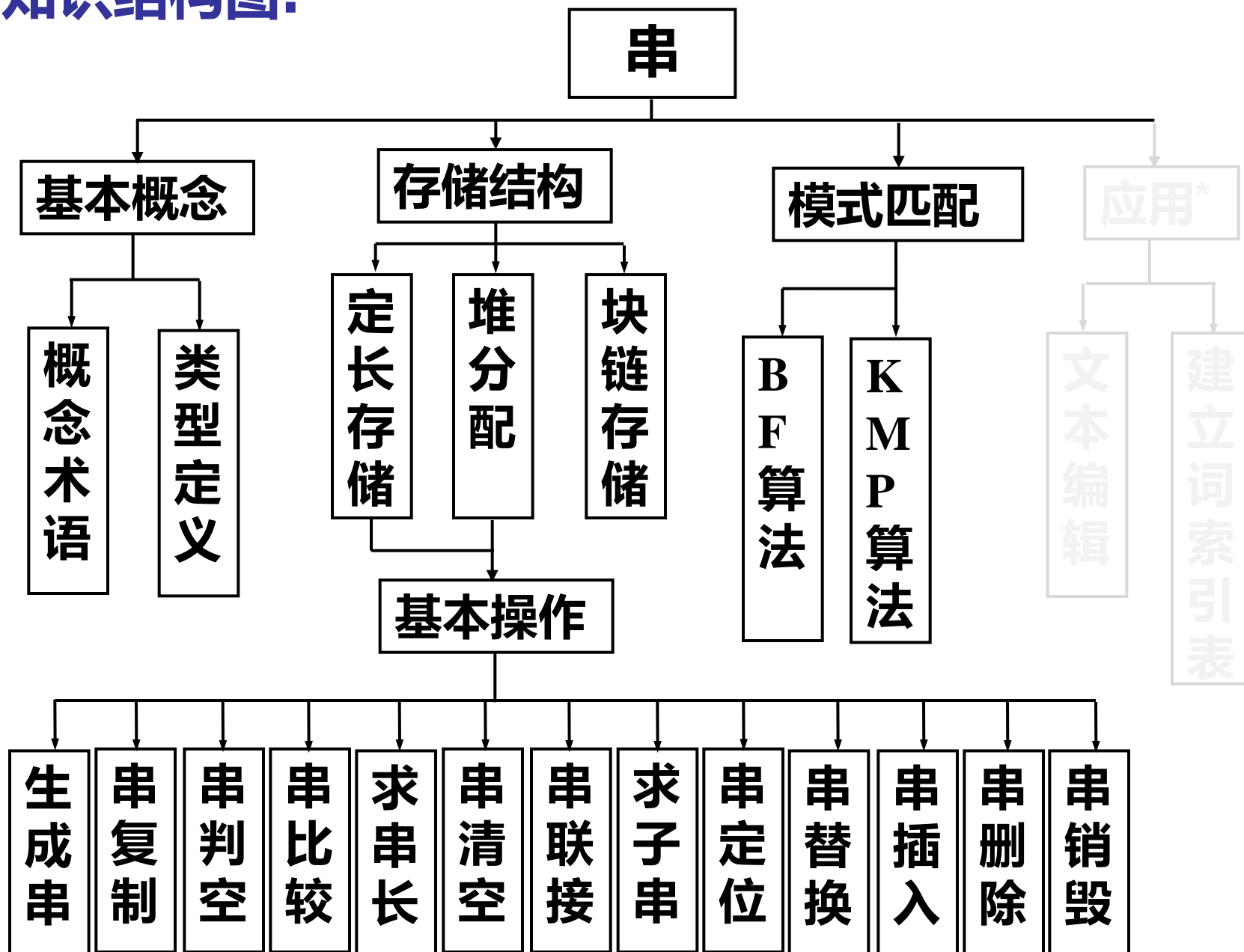
j=9



KMP算法时间复杂度分析

- 特点：在整个匹配过程中，对主串仅需从头到尾扫描一遍，主串指针不再回溯，因此可以边读入边匹配，对处理从外设读取的大文本匹配尤其有效。
- 时间复杂度： **$O(m+n)$** ，其中， **n** 是主串长度， **m** 为模式串长度。
- 简单模式匹配算法的时间复杂度是 **$O(m \times n)$** ，一般情况下，其实际的执行时间近似 **$O(m+n)$** 。当模式串与主串之间存在很多“部分匹配”时，**KMP**算法才比**BF**算法高效得多。

知识结构图:



2019年408真题

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

09. 设主串 $T = \text{"abaabaabcabaabc"}$ ，模式串 $S = \text{"abaabc"}$ ，采用 KMP 算法进行模式匹配，到匹配成功时为止，在匹配过程中进行的单个字符间的比较次数是（ ）。

A. 9

B. 10

C. 12

D. 15

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
主串	a	b	a	a	b	a	a	b	c	a	b	a	a	b	c
1	a	b	a	a	b	c									
2				a	b	a	a	b	c						

答案：B

2015年408真题

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

08. 已知字符串 s 为“abaabaabacacaabaabcc”，模式串 t 为“abaabc”。采用 KMP 算法进行匹配，第一次出现“失配” ($s[i] \neq t[j]$) 时， $i = j = 5$ ，下次开始匹配时， i 和 j 的值分别是 ()。
- A. $i = 1, j = 0$ B. $i = 5, j = 0$ C. $i = 5, j = 2$ D. $i = 6, j = 2$

答案：C



练习

- KMP算法中，`next[j]`函数的计算不仅与模式串本身有关，还与相匹配的主串有关。（对 / 错）
- 串S='software'的子串有多少个？
- 串T='database'的子串有多少个？
- 设S为一个长度为n的字符串，其中的字符各不相同，则S中的互异的非平凡子串（非空且不同于S本身）的个数为。