

第三章 字符串

张史梁

slzhang.jdl@pku.edu.cn

内容提要

- 字符串及其运算
- 字符串的存储表示
 - 顺序存储
 - 链式存储
- 模式匹配*
- 练习

字符串示例

- $s1 = "123"$
- $s2 = "ABBABBC"$
- $s3 = "BB"$
- $s4 = "BB "$
- $s5 = ""$

- **字符串**：简称为串，是零个或多个字符组成的有限序列。一般记为： $s = "s_0s_1 \dots s_{n-1}"$ ($n \geq 0$)
- s 为串名
 - 每个字符 s_i ($0 \leq i \leq n-1$)可以是字母、数字或其它字符
 - 一般用一对双引号 “ ” 将串括起来，避免和其他变量混淆

字符串是一种特殊的线性结构

□ 字符串与一般线性表的区别

- 串的数据对象约束为**字符集**，其每个结点由一个字符组成；
- 线性表的基本操作大多以“单个元素”为操作对象，而串的基本操作通常以“**串的整体**”作为操作对象；
- **线性表的存储方法同样适用于字符串**，在选择存储结构时，应根据不同情况选择合适的存储表示。

□ C的标准函数库中的字符串：<string.h>

- 例如：char s1[7] = “value”;
- 串的结束标记：**‘\0’ (‘\0’是ASCII码中8位BIT全0码，又称为NULL符)**
- 字符串占用的实际长度为 $7-1=6$

字符型变量的赋值

□ 基本方法：通过赋值语句

```
int main() {  
    char b, c, d;  
  
    b = 'x' ;  
    c = b;  
    d = 120;  
    return 0;  
}
```

b = c = d = 'x'

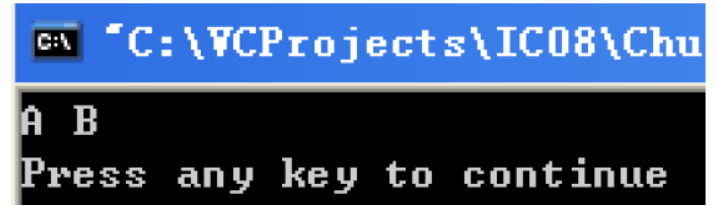
ASCII 编码表

ASCII 字符代码表 一

高四位 低四位		ASCII非打印控制字符										ASCII 打印字符											
		0000					0001					0010	0011		0100		0101		0110		0111		
		0					1					2	3		4		5		6		7		
		+进制	字符	ctrl	代码	字符解释	+进制	字符	ctrl	代码	字符解释	+进制	字符	+进制	字符	+进制	字符	+进制	字符	+进制	字符	+进制	字符
0000	0	0	BLANK NULL	^@ NUL	空	16	▶	^P	DLE	数据链路转意	32		48	0	64	@	80	P	96	`	112	p	
0001	1	1	☺	^A SOH	头标开始	17	◀	^Q	DC1	设备控制 1	33	!	49	1	65	A	81	Q	97	a	113	q	
0010	2	2	☹	^B STX	正文开始	18	↕	^R	DC2	设备控制 2	34	"	50	2	66	B	82	R	98	b	114	r	
0011	3	3	♥	^C ETX	正文结束	19	!!	^S	DC3	设备控制 3	35	#	51	3	67	C	83	S	99	c	115	s	
0100	4	4	♦	^D EOT	传输结束	20	¶	^T	DC4	设备控制 4	36	\$	52	4	68	D	84	T	100	d	116	t	
0101	5	5	♣	^E ENQ	查询	21	♢	^U	NAK	反确认	37	%	53	5	69	E	85	U	101	e	117	u	
0110	6	6	♠	^F ACK	确认	22	■	^V	SYN	同步空闲	38	&	54	6	70	F	86	V	102	f	118	v	
0111	7	7	●	^G BEL	震铃	23	↑↓	^W	ETB	传输块结束	39	'	55	7	71	G	87	w	103	g	119	w	
1000	8	8	◼	^H BS	退格	24	↑	^X	CAN	取消	40	(56	8	72	H	88	X	104	h	120	x	
1001	9	9	○	^I TAB	水平制表符	25	↓	^Y	EM	媒体结束	41)	57	9	73	I	89	Y	105	i	121	y	
1010	A	10	◻	^J LF	换行/新行	26	→	^Z	SUB	替换	42	*	58	:	74	J	90	Z	106	j	122	z	
1011	B	11	♂	^K VT	竖直制表符	27	←	^[ESC	转意	43	+	59	;	75	K	91	[107	k	123	{	
1100	C	12	♀	^L FF	换页/新页	28	└	^\ FS	文件分隔符	44	,	60	<	76	L	92	\	108	l	124			
1101	D	13	♪	^M CR	回车	29	↔	^] GS	组分隔符	45	-	61	=	77	M	93]	109	m	125	}		
1110	E	14	🎵	^N SO	移出	30	▲	^6 RS	记录分隔符	46	.	62	>	78	N	94	^	110	n	126	~		
1111	F	15	☼	^O SI	移入	31	▼	^- US	单元分隔符	47	/	63	?	79	O	95	_	111	o	127	Δ	Back space	

Examples

```
int main(){  
    char a;  
    a = 0;  
    printf("A%cB\n", a);  
    return 0;  
}
```



A screenshot of a Windows command prompt window. The title bar is blue and contains the text "C:\VCProjects\IC08\Chu". The command prompt itself has a black background with white text. It shows the output of the program: "A B" on the first line and "Press any key to continue" on the second line.

编码值为0的字符

```
int main(){  
    char a;  
    a = 0;  
    .....  
    return 0;  
}
```



```
int main(){  
    char a;  
    a = ' ';  
    .....  
    return 0;  
}
```

```
int main(){  
    char a;  
    a = 0;  
    .....  
    return 0;  
}
```



```
int main(){  
    char a;  
    a = '0';  
    .....  
    return 0;  
}
```


ASCII 编码表

ASCII 字符代码表 一

高四位 低四位		ASCII非打印控制字符										ASCII 打印字符											
		0000					0001					0010	0011	0100	0101	0110	0111						
		0					1					2	3	4	5	6	7						
		十进制	字符	ctrl	代码	字符解释	十进制	字符	ctrl	代码	字符解释	十进制	字符	十进制	字符	十进制	字符	十进制	字符	十进制	字符	ctrl	
0000	0	0	BLANK NULL	^@	NUL	空	16	►	^P	DLE	数据链路转意	32		48	0	64	@	80	P	96	`	112	p
0001	1	1	☺	^A	SOH	头标开始	17	◄	^Q	DC1	设备控制 1	33	!	49	1	65	A	81	Q	97	a	113	q
0010	2	2	☻	^B	STX	正文开始	18	↕	^R	DC2	设备控制 2	34	"	50	2	66	B	82	R	98	b	114	r
0011	3	3	♥	^C	ETX	正文结束	19	!!	^S	DC3	设备控制 3	35	#	51	3	67	C	83	S	99	c	115	s
0100	4	4	◆	^D	EOT	传输结束	20	¶	^T	DC4	设备控制 4	36	\$	52	4	68	D	84	T	100	d	116	t
0101	5	5	♣	^E	ENQ	查询	21	♢	^U	NAK	反确认	37	%	53	5	69	E	85	U	101	e	117	u
0110	6	6	♠	^F	ACK	确认	22	■	^V	SYN	同步空闲	38	&	54	6	70	F	86	V	102	f	118	v
0111	7	7	●	^G	BEL	震铃	23	↑↓	^W	ETB	传输块结束	39	'	55	7	71	G	87	w	103	g	119	w
1000	8	8	◼	^H	BS	退格	24	↑	^X	CAN	取消	40	(56	8	72	H	88	X	104	h	120	x
1001	9	9	○	^I	TAB	水平制表符	25	↓	^Y	EM	媒体结束	41)	57	9	73	I	89	Y	105	i	121	y
1010	A	10	◻	^J	LF	换行/新行	26	→	^Z	SUB	替换	42	*	58	:	74	J	90	Z	106	j	122	z
1011	B	11	♂	^K	VT	竖直制表符	27	←	^[ESC	转意	43	+	59	;	75	K	91	[107	k	123	{
1100	C	12	♀	^L	FF	换页/新页	28	└	^\ FS	文件分隔符	44	,	60	<	76	L	92	\	108	l	124		
1101	D	13	♪	^M	CR	回车	29	↔	^] GS	组分隔符	45	-	61	=	77	M	93]	109	m	125	}	
1110	E	14	🎵	^N	SO	移出	30	▲	^6 RS	记录分隔符	46	.	62	>	78	N	94	^	110	n	126	~	
1111	F	15	☼	^O	SI	移入	31	▼	^- US	单元分隔符	47	/	63	?	79	O	95	_	111	o	127	Δ	Back space

How about this one?

```
int main(){  
    char a;  
    a = 0;  
    .....  
    return 0;  
}
```

=

```
int main(){  
    char a;  
    a = '\0';  
    .....  
    return 0;  
}
```

```
Int main(){  
    char str[10] = {48, 0, };  
    printf ("%c\n",str[0]);  
    printf("%c\n", str[1]);  
    printf("%c\n", str[2]);  
    return 0;  
}
```

基本概念

- 串的长度：一个字符串中字符个数
- 空串：长度为零的字符串，记为 $s = ""$
 - [注意与空格串 “ ” 的区别]
- 主串与子串：字符串 s_1 中任意个连续字符组成的子序列 s_2 被称为 s_1 的子串，称 s_1 为 s_2 的主串。
 - 假设 S_1 和 S_2 是两个串 $S_1 = "a_1 a_2 \dots a_n"$ $S_2 = "b_1 b_2 \dots b_m"$ ，其中 a_i 和 b_j 代表字符， $0 \leq m \leq n$
 - 如果存在整数 i ($0 \leq i \leq n-m$)，使得 $a_{i+j} = b_j$ ($j=1 \dots m$)
 - 则称 S_2 是 S_1 的“子串”，又称 S_1 包含 S_2
 - 特别地，空串是任意串的子串，任意串 s 都是 s 本身的子串；除 s 本身之外， s 的其他子串称为 s 的真子串。

基本概念

- **字符在串中的位置**：该字符在串中第一次出现的位置。
- **子串在主串中的位置**：该子串的第一个字符在主串中的位置。
- **两个字符串相等的充分必要条件**：长度相等，且对应位置上字符相同。

字符串示例

□ A="123" 长度=3

□ B="ABBABBC" 长度=7

□ C="BB" 长度=2

□ D="BB " 长度=3

□ E="" 长度=0

字符串的基本运算

□ 定义抽象数据类型String的运算如下:

- ① 创建一个空串;
- ② 判断一个串是否为空串, 若为空串, 则返回1, 否则返回0;
- ③ 求一个串的长度;
- ④ 将两个串拼接在一起构成一个新串
- ⑤ 在串s中求从第i个字符开始连续j个字符所构成的子串
- ⑥ 如果串S2是S1的子串, 则可求串S2在串S1中第一次出现的位置

字符串的基本运算

C的标准函数库<string.h>中有关字符串的函数

—串长函数 `int strlen(char *s);`

—串复制 `char *strcpy(char *destin, char *source)`

—串拼接 `char *strcat(char *destin, char *source)`

—串比较 `char *strcmp(char *str1, char *str2)`

 //看Ascii码, `str1>str2`, 返回值 `> 0`; 两串相等, 返回0

—定位函数 `char *strchr(char *str, char c);`

 //首次出现字符c的位置

—子串 `char *strstr(const char *str1, const char *str2);`

—将串转换为double型 `double strtod(char *str, char **endptr);`

—

示例-1

```
①    #include <stdio.h>
②    #include <string.h>
③    int main(void)
④    {
⑤        char string[10];
⑥        char *str1 = "abcdefghi";
⑦        strcpy(string, str1);
⑧        printf("%s\n", string);
⑨        return 0;
⑩    }
```


示例-2

```
①    #include <string.h>
②    #include <stdio.h>
③    int main(void)
④    {
⑤    char destination[25];
⑥    char *blank = " ", *c = "C++", *Borland = "Borland";
⑦    strcpy(destination, Borland);
⑧    strcat(destination, blank);
⑨    strcat(destination, c);
⑩    printf("%s\n", destination);
⑪    return 0;
⑫    }
```

示例-3

```
①  #include <string.h>
②  #include <stdio.h>
③  int main(void)
④  {
⑤  char *buf1 = "aaa", *buf2 = "bbb", *buf3 = "ccc";
⑥  int ptr;
⑦  ptr = strcmp(buf2, buf1);
⑧  if (ptr > 0) printf("buffer 2 is greater than buffer 1\n");
⑨  else      printf("buffer 2 is less than buffer 1\n");
⑩  ptr = strcmp(buf2, buf3);
⑪  if (ptr > 0)
⑫      printf("buffer 2 is greater than buffer 3\n");
⑬  else  printf("buffer 2 is less than buffer 3\n");
⑭  return 0;
⑮  }
```

内容提要

□ 字符串及其运算

□ 字符串的存储表示

■ 顺序存储

■ 链式存储

线性表的存储方法同样适用于字符串，在选择存储结构时，应根据不同情况选择合适的存储表示。

□ 模式匹配

□ 练习

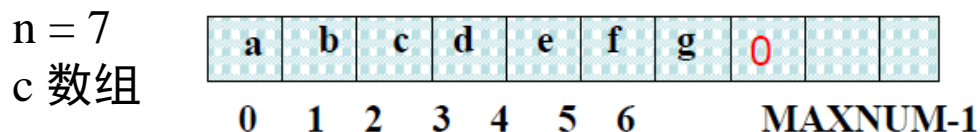
字符串的存储表示

- 字符串仍是一种特殊的线性表，
 - 其每个结点仅由一个字符组成，线性表的存储方法同样适用于字符串。
 - 在选择存储结构时，应根据不同情况选择合适的存储表示。
 - 例如，插入、删除时，采用顺序结构不方便，但访问字符时（尤其连续多个）非常容易。

字符串的顺序存储

- ```
#define MAXNUM 500 //串允许的最多字符个数
struct SeqString
{
 char c[MAXNUM]; /* 字符数组 */
 int n; /* 串长n<= MAXNUM */
};
typedef struct SeqString *PSeqString;
```

最后一个字符后面可加一个特定的结束标志' \0'，该结束标志占用存储空间但不计入串长。譬如PSeqStrings = “abcdefg”



# 顺序表-创建空串

---

//初始化，创建空的字符串

```
PSeqString CreateNULLStr_Seq(void)
{
 PSeqString pstr;
 pstr = (PSeqString)malloc(sizeof(struct SeqString));
 if (!pstr)
 {
 printf(" 内存溢出!\n");
 }
 else
 {
 pstr->n = 0; // 串长置0
 }
 return pstr;
}
```

# 顺序表 – 连接两个串

---

```
PSeqString strcat_Seq(struct SeqString s1, struct SeqString s2)
{
 int i;
 PSeqString s = NULL;
 printf(" s1 = %s, s2 = %s\n", s1.c, s2.c);
 if (s1.n+s2.n >= MAXNUM) return NULL; // 必须保证新字符串长度合法

 s = CreateNULLStr_Seq(); // 创建空串
 for (i = 0; i < s1.n; i++) s->c[i] = s1.c[i]; // 将s1传给s
 for (i = 0; i < s2.n; i++) s->c[s1.n+i] = s2.c[i]; // 将s2传给s
 s->n = s1.n+s2.n; // 联结后的长度
 s->c[s->n] = '\0'; // 最后结束标志

 return s;
}
```

# 顺序表 – 取子串

---

**//求从字符串S中第 i 个字符开始连续取 j 个字符所构成的子串**

**PSeqString substr\_Seq( struct SeqString s, int i, int j )**

```
{ PSeqString s1; int k;
 s1 = CreateNULLStr_Seq(); // 创建空串
 if (!s1) return NULL;
 if (i > 0 && i <= s.n && j > 0) // i, j 必须合法
 { if (s.n < i+j-1) j = s.n-i+1; // 从i开始的连续字符数不够j个时
 for (k = 0; k < j; k++) s1->c[k] = s.c[i+k-1];
 s1->c[j] = '\0'; // 最后结束标志
 s1->n = j; // 串长
 } else s1->c[0] = '\0'; // 空串
 return s1;
}
```



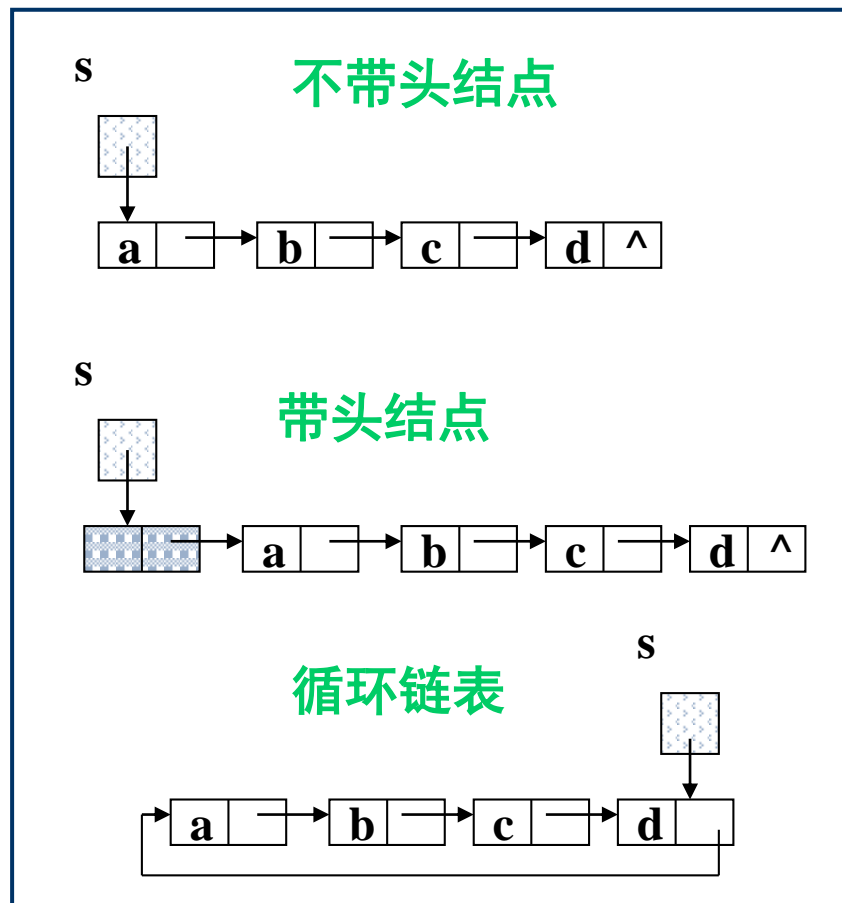
# 内容提要

---

- 字符串及其运算
- 字符串的存储表示
  - 顺序表示
  - 链接表示
- 模式匹配
- 练习

# 字符串的链接存储

```
struct StrNode ; /* 链串的结点 */
typedef struct StrNode *PStrNode;
/*结点指针类型*/
struct StrNode /*链串的结点结构*/
{
 char c;
 PStrNode link;
};
typedef struct StrNode *PLinkString;
/* 链串的类型 */
PLinkString s;
```



# 链表-取子串

---

//求从字符串S中第 i 个字符开始连续取 j 个字符所构成的子串

```
PLinkString SubStr_link(PLinkString s, int i, int j)
{
 PLinkString s1;
 PStrNode p, q, t;
 int k;
 s1 = CreateNULLStr_link(); //创建带头结点新串s1
 if (!s1) return NULL;
 if (i < 1 || j < 1) return s1; //如果i,j值不合法

 p = s; //p指向头结点
 for (k = 1; k <= i; k++) if (p!=NULL) p = p->link; //找第i个结点
 if (p==NULL) return s1;
```

# 链表-取子串

---

```
t = s1; //t指向s1的头结点
for (k = 1; k<=j; k++) //连续取j个字符
{
 if (p!=NULL)
 { q = (PStrNode)malloc(sizeof(struct StrNode));
 if (q==NULL) break;

 q->c = p->c; //将p指向的结点加入到s1中
 q->link = NULL;
 t->link = q;
 t = q;
 p = p->link;
 }
}
return s1;
```

# 链接存储结构的基本运算

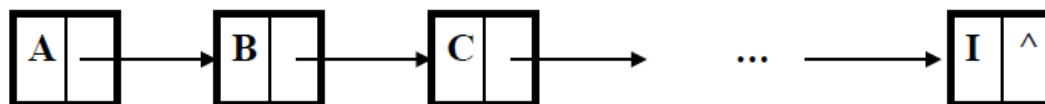
---

- ❑ PLinkStringCreateNULLStr\_link( void );
- ❑ intLenStr\_link( PLinkStrings );
- ❑ PLinkStringConcatStr\_link(PLinkStringr1,PLinkString r2);
- ❑ PLinkStringSubStr\_link(PLinkStrings, inti, intj );
- ❑ void DelSubString\_link( PLinkStringr, inti, intj );
- ❑ intInsertSubStr\_link( PLinkStringr, inti, PLinkStringr1 );
- ❑ intLocSubStr\_link( PLinkStringr1, PLinkStringr2 ); //strstr
- ❑ intprint (PLinkStringpst);
- ❑ .....

# 讨论

□ 上述链接存储结构虽然操作实现简单，但存储密度太低。

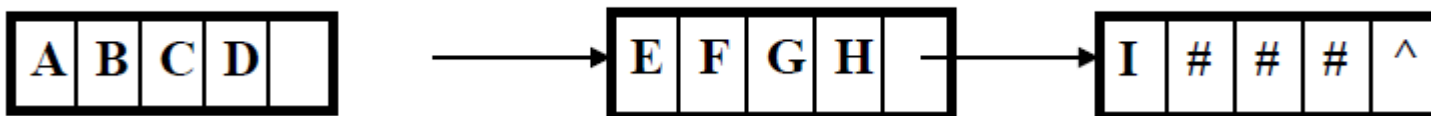
■ 存储密度 = 串值所占存储空间 / 为链表分配的存储空间



□ 改进办法：每个结点存放多个字符。

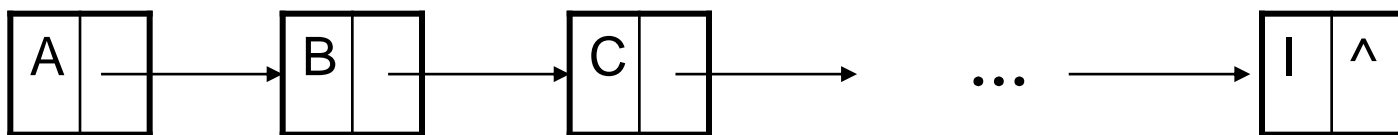
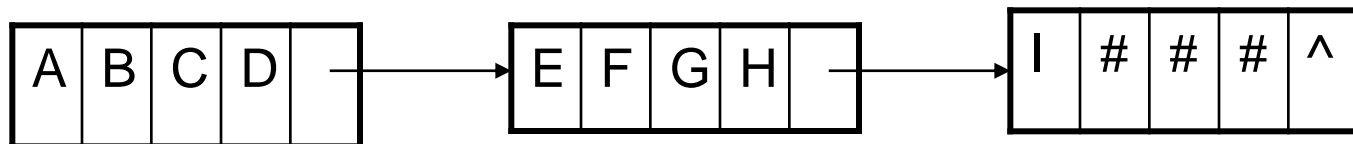
■ 改进后虽然提高了存储密度，但操作实现变的复杂。每个结点存放多少个字符？

■ 最后结点存放字符的个数与前面的不同，操作实现需要特殊处理。



# 串值的链式存储方式

---



存储密度小, 运算处理方便, 但是存储占用量大.

# 内容提要

---

- 字符串及其运算
- 字符串的存储表示
  - 顺序存储
  - 链式存储
- 模式匹配
- 练习



# 问题定义

---

□ 设有两个串t和p:

- $t = t_0t_1\dots t_{n-1}$ ,  $p = p_0p_1\dots p_{m-1}$ , 其中  $1 < m \leq n$
- 在t中找出和p相同的子串。此时, t称为“目标”, 而p称为“模式”。

□ `intLocSubStr_Seq( structSeqString t, structSeqString p )`:

- 匹配成功: t中存在等于p的子串, 返回子串在t中的位置
- 匹配失败: 返回一个特定的标志 (如-1)。

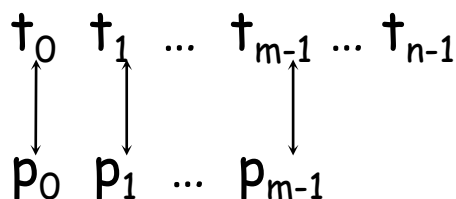
# 两种方法

---

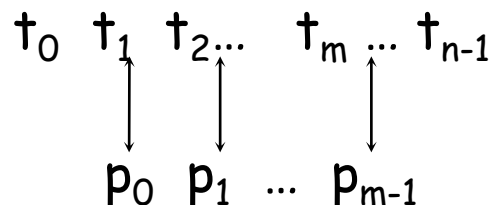
- 模式匹配是一个比较复杂的字符串操作，
  - 用于在大文本（诸如，句子、段落，或书本）中定位（查找）特定的模式
  - 对于大多数的算法而言，匹配的主要考虑在于其速度和效率
- 下面的讨论是基于字符串的顺序存储结构进行。
  - 朴素的模式匹配方法(Brute Force)
  - 无回溯的模式匹配方法 Knuth-Morris-Pratt (KMP)

# 朴素的模式匹配思想

□ 用 $p$ 中的字符依次与 $t$ 中的字符比较：



1, 如果 $t_0 = p_0$ ,  $t_1 = p_1$ , ...,  $t_{m-1} = p_{m-1}$ , 则匹配成功;



2, 否则, 必有某个 $i$  ( $0 \leq i \leq m-1$ ), 使得 $t_i \neq p_i$ , 将 $p$ 右移一个字符, 用 $p$ 中字符从头开始与 $t$ 中字符依次比较。

如此反复执行, 直到下面两种情况之一:

# 朴素的模式匹配(续)

- ① 到达某步时,  $t_i = p_0, t_{i+1} = p_1, \dots, t_{i+m-1} = p_{m-1}$ , 匹配成功,  $\text{subStr\_seq}(t, i, m)$  即是找到的第一个与模式  $p$  相同的子串;
- ② 将  $p$  移到无法与  $t$  继续比较为止 (剩余长度  $< m$ ), 匹配失败

```
t a b b a b a
 || |
p a b a
```

(a)  $p_2 \neq t_2$  将  $p$  右移一位

```
t a b b a b a
 |
p a b a
```

(b)  $p_0 \neq t_1$  将  $p$  右移一位

```
t a b b a b a
 |
p a b a
```

(c)  $p_0 \neq t_2$  将  $p$  右移一位

```
t a b b a b a
 |||
p a b a
```

(d) 匹配成功  $\text{subStr\_seq}(t, 4, 3) = p$

# 朴素的模式匹配 算法

---

```
int LocSubStr_Seq(struct SeqString t, struct SeqString p)
{
 int pi=0, tj=0;
 while (pi < p->n && tj < t->n) //反复比较
 {
 if (p->c[pi] == t->c[tj])
 { pi++; tj++; } //继续匹配下一个字符
 else
 { tj = tj-pi+1; pi = 0; } //主串、子串pi、tj值回溯，重新开始
 }
 if (pi >= p->n) return (tj-p->n+1); //匹配成功，返回位置
 else return 0; //匹配失败
}
```

# example-1

| 下标i      | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
|----------|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|
| 目标t      | a | a | b | c | b | a | b | c | a | a | b  | c  | a  | a  | b  | a  | b  | c  |
| 模式p      | a | b | c | a | a | b | a | b | c |   |    |    |    |    |    |    |    |    |
| i=3,j=4  |   | a | b | c | a | a | b | a | b | c |    |    |    |    |    |    |    |    |
| i=0,j=2  |   |   | a | b | c | a | a | b | a | b | c  |    |    |    |    |    |    |    |
| i=0,j=3  |   |   |   | a | b | c | a | a | b | a | b  | c  |    |    |    |    |    |    |
| i=0,j=4  |   |   |   |   | a | b | c | a | a | b | a  | b  | c  |    |    |    |    |    |
| i=6,j=11 |   |   |   |   |   | a | b | c | a | a | b  | a  | b  | c  |    |    |    |    |
| i=0,j=6  |   |   |   |   |   |   | a | b | c | a | a  | b  | a  | b  | c  |    |    |    |
| i=0,j=7  |   |   |   |   |   |   |   | a | b | c | a  | a  | b  | a  | b  | c  |    |    |
| i=1,j=9  |   |   |   |   |   |   |   |   | a | b | c  | a  | a  | b  | a  | b  | c  |    |
| i=9,j=17 |   |   |   |   |   |   |   |   |   | a | b  | c  | a  | a  | b  | a  | b  | c  |

# 算法时间效率分析

---

## □ 匹配失败

- 最坏情况：每趟匹配皆在最后一个字符不等，且有  $n-m+1$  趟匹配（每趟比较  $m$  个字符），
- 共比较  $m*(n-m+1)$  次，由于  $m \ll n$ ，因此最坏时间复杂度  $O(n*m)$ 。
- 最好情况：  $n-m+1$  次比较[每趟只比较第一个字符]。

## □ 匹配成功

- 最好情况：  $m$  次比较。
- 最坏情况： 与匹配失败的最坏情况相同。

## □ 综上讨论，朴素模式匹配算法的时间复杂度为 $O(m*n)$ 。

# 进一步的分析

- 朴素模式匹配算法简单，容易理解，但效率不高。
  - 主要原因是：一旦比较不等， $p$ 右移一个字符并且下次从 $p_0$ 开始重新进行比较，对于目标 $t$ ，存在回溯现象。
- 如何减少或避免回溯？
  - 某次匹配失败时，下次匹配时是否可以利用前面已经比较的结果？

|   |                                    |                                    |                                    |             |
|---|------------------------------------|------------------------------------|------------------------------------|-------------|
| t | a b b a b a                        | a b b a b a                        | a b b a b a                        | a b b a b a |
| p | a b a                              | a b a                              | a b a                              | a b a       |
|   | p <sub>2</sub> , t <sub>2</sub> 不等 | p <sub>0</sub> , t <sub>1</sub> 不等 | p <sub>0</sub> , t <sub>2</sub> 不等 | 匹配成功        |

- $p_0 \neq p_1$  可以推出  $p_0 \neq t_1$ ，所以  $p$  右移一位后的比较一定不等；
- $p_0 = p_2$ ，可以推出  $p_0 \neq t_2$
- 因此，可以由第1趟匹配直接跳过2、3趟匹配进入第4趟匹配，此时消除了回溯问题，提高了模式匹配的时间效率。



# 无回溯模式匹配的问题描述

- 在前例中，一旦 $p_i$ 与 $t_j$ 比较不等，有：

$$\text{SubStr\_Seq}(p, 0, i-1) = \text{SubStr\_Seq}(t, j-i, j-1)$$

$$p_i \neq t_j$$

$$\begin{array}{ccccccc} t_0 \dots t_{j-i-1} & t_{j-i} & \dots & t_{j-1} & t_j & \dots \\ & \parallel & & \parallel & \not\parallel & \\ & p_0 & \dots & p_{i-1} & p_i & \dots \end{array}$$

- 目标

- 要避免回溯  $j \rightarrow j-i+1, i \rightarrow 0$
- 要确定 $p$ 右移的位数和继续（无回溯）比较的字符，也就是说应该用 $p$ 中的哪个字符和 $t_j$ 进行比较？
- 设这个字符为 $p_k$ ，下标 $k$ 与 $p_i$ 密切相关，因此可表示为 $\text{next}[i]$ ，则无回溯的模式匹配算法可实现为：

# 无回溯的模式匹配算法

```
int LocSubStr_Seq(struct SeqString t, struct SeqString p, int *next)
```

```
{ int i=0, j=0;
 while (i < p->n && j < t->n) //反复比较
 { if (i == -1 || p->c[i] == t->c[j])
 { i++; j++; } //继续匹配下一个字符
```

```
 else
```

```
 i = next[i]; // j不变, i后退
```

```
 }
```

```
//匹配成功, 返回p中第一个字符在t中的位置
```

```
if (i >= p->n) return (j - p->n + 1)
```

```
else return 0;
```

```
}
```

若 $next[i] = -1$ , 则表示p中任何字符都不必在与 $t_j$ 进行比较, 下次比较从 $t_{j+1}$ 与 $p_0$ 开始。

|   |                                                           |                                                           |                                                           |                                                           |
|---|-----------------------------------------------------------|-----------------------------------------------------------|-----------------------------------------------------------|-----------------------------------------------------------|
| t | a b b a b a                                               | a b b a b a                                               | a b b a b a                                               | a b b a b a                                               |
| p | <div> <div>   </div> <div>   </div> <div>   </div> </div> | <div> <div>   </div> <div>   </div> <div>   </div> </div> | <div> <div>   </div> <div>   </div> <div>   </div> </div> | <div> <div>   </div> <div>   </div> <div>   </div> </div> |
|   | a b a                                                     | a b a                                                     | a b a                                                     | a b a                                                     |
|   | $p_2, t_2$ 不等                                             | $p_0, t_1$ 不等                                             | $p_0, t_2$ 不等                                             | 匹配成功                                                      |

# next[i]如何计算？


□ D.E.Knuth、J.H.Morris、V.R.Pratt 同时发现：

■  $P_k$ 的k值仅依赖于模式p本身前i个字符的组成，而与目标t无关！！

□ next[i]表示 $P_i$ 对应的k值：

■ 若next[i] ≥ 0，表示一旦匹配过程中 $p_i$ 与 $t_j$ 比较不等，可用p中以next[i]为下标的字符与 $t_j$ 进行比较（ $p_{\text{next}[i]}$ 与 $t_j$ 比较）。

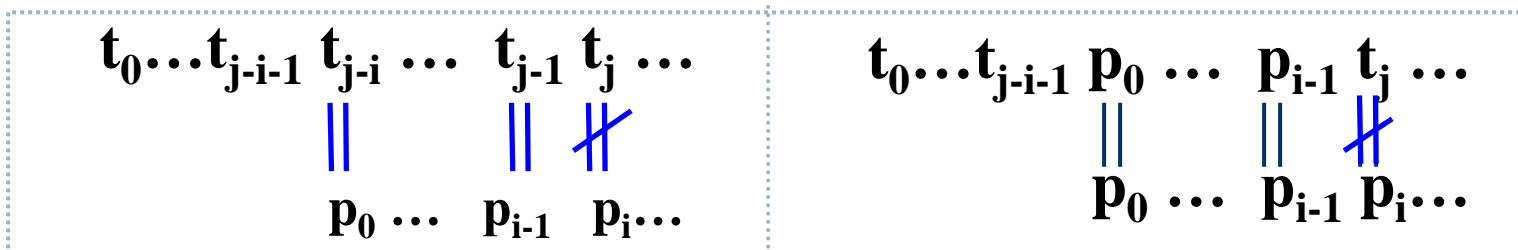
■ 若next[i] = -1，则表示p中任何字符都不必再与 $t_j$ 进行比较，下次比较从 $t_{j+1}$ 与 $p_0$ 开始。

|   |                                                                                     |                                                                                     |                                                                                       |                                                                                       |
|---|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------|
| t | a b b a b a                                                                         | a b b a b a                                                                         | a b b a b a                                                                           | a b b a b a                                                                           |
| p | a b a                                                                               | a b a                                                                               | a b a                                                                                 | a b a                                                                                 |
|   |  |  |  |  |
|   | $p_2, t_2$ 不等                                                                       | $p_0, t_1$ 不等                                                                       | $p_0, t_2$ 不等                                                                         | 匹配成功                                                                                  |

□ next[0]=-1, next[1]=0, next[2]=0

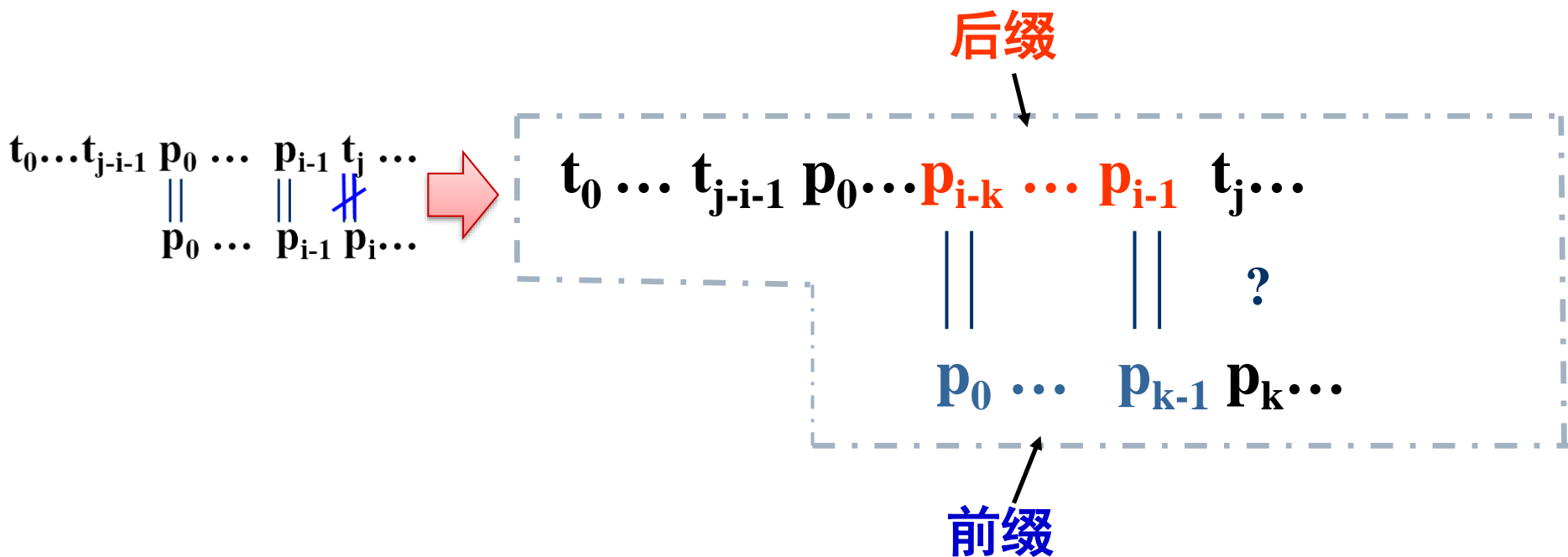
# next数组计算—1

- 在p与任意的目标串t匹配时，若发现 $t_j \neq p_i$ ，则意味着 $p_0$ 、 $p_1$ 、...、 $p_{i-1}$ 已经与t中对应的字符进行过比较，而且是相等的，否则轮不到 $t_j$ 与 $p_i$ 的比较，因此下面两个图是等价的。



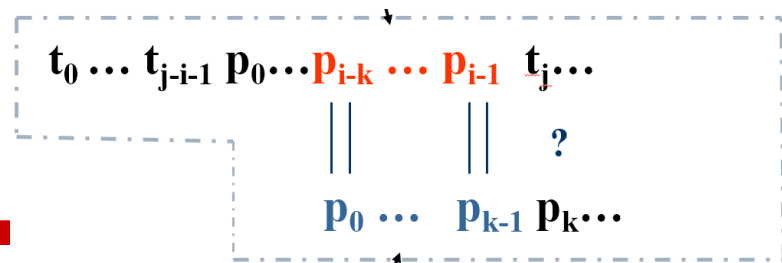
# next数组计算-2

- 然后把 $p$ 右移若干位,  $t_j$ 以前的比较工作相当于用 $p_0 \dots p_{i-1}$ 的一个前缀与 $p_0 \dots p_{i-1}$ 的一个长度相同的前缀进行对齐, 显然比较的结果由 $p$ 本身决定。



结论:  $p_i$  对应的  $k/\text{next}[i]$  等于  $p_0 \dots p_{i-1}$  中相同并且最大的前缀和后缀的长度 ( $0 \leq k < i-1$ )。

# next数组计算—3



□ 因此，可以在 $p_0 \dots p_{i-1}$ 中求出相同并且最大的前缀和后缀的长度  $k$  ( $0 \leq k < i$ )。

■ 此时，相同的前后缀恰好对齐， $p_k$ 与 $t_j$ 也对齐，可以从 $p_k$ 和 $t_j$ 开始向右比较。

■ 不用去比较这一对前后缀，因为它们相等的。

□ 通过上面分析，得到了初步的next计算方法：

(1)  $\text{next}[i] = p_0 \dots p_{i-1}$ 中最大相同的前缀和后缀的长度 $k$ ;

(2) 当 $i=0$ 时，令 $\text{next}[i] = -1$ ;

显然，对于任意 $i$  ( $0 \leq i < m$ )，有 $\text{next}[i] < i$ 。

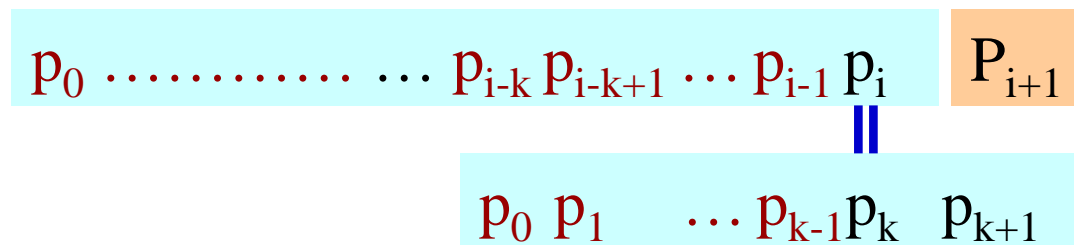
# next数组计算—4

□ 假定已经计算得到 $\text{next}[i]$ , 那么 $\text{next}[i+1] = ?$

对于 $\text{next}[i]=k$ , 有:  $p_0 \dots p_{k-1} = p_{i-k} \dots p_{i-1}$

(1) 如果 $p_k=p_i$ , 则有:  $p_0 \dots p_{k-1} p_k = p_{i-k} \dots p_{i-1} p_i$

$$\text{next}[i+1] = k+1 = \text{next}[i]+1$$



(2) 如果 $p_k \neq p_i$ , 此时有:  $p_0 \dots p_{k-1} = p_{i-k} \dots p_{i-1}$

与模式匹配的思路一样, 应当将模式串往右滑动到模式串的第 $\text{next}[k]$ 个字符与 $p_i$ 进行比较, 即  $k=\text{next}[k]$ 。如果 $p_k \neq p_i$ 则继续滑动。

# next数组的计算—算法

```
makeNext(PSeqString p, int *next)
```

```
{
 int i , k; next[0] = -1;
 for (i=0; i < p->n-1; i++) /* 计算next[i+1] */
 {
 k=next[i];
 while (k >= 0 && p->c[i] != p->c[k])
 k = next[k]; /* p0...pi中最大的相同的前后缀长度k */

 next[i+1] = k+1;
 }
}
```

|                |    |   |   |   |   |   |   |   |   |
|----------------|----|---|---|---|---|---|---|---|---|
| 下标i            | 0  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| p <sub>i</sub> | a  | b | c | a | a | b | a | b | c |
| k              | -1 | 0 | 0 | 0 | 1 | 1 | 2 | 1 | 2 |
| next[i]        | -1 | 0 | 0 | 0 | 1 | 1 | 2 | 1 | 2 |



# next数组的计算—算法'

```
makeNext(PSeqString p, int *next)
```

```
{
```

```
 int i = 0, k = -1; next[0] = -1;
```

```
 while (i < p->n-1) /* 计算next[i+1] */
```

```
 {
```

```
 while (k >= 0 && p->c[i] != p->c[k])
```

```
 k = next[k]; /* p0...pi中最大的相同的前后缀长度k */
```

```
 i++; k++;
```

```
 next[i] = k;
```

```
 }
```

```
}
```

| 下标i            | 0  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|----------------|----|---|---|---|---|---|---|---|---|
| p <sub>i</sub> | a  | b | c | a | a | b | a | b | c |
| k              | -1 | 0 | 0 | 0 | 1 | 1 | 2 | 1 | 2 |
| next[i]        | -1 | 0 | 0 | 0 | 1 | 1 | 2 | 1 | 2 |

# next数组计算 – 算法改进

- 按照上述方法求得的next数组已经可以用于无回溯模式匹配中，但是还可以进一步改进。
- 例如，若求得k后，有 $p_k = p_i$ ，则当 $p_i \neq t_j$ 时， $p_k$ 与 $t_j$ 的比较必然不等，应该继续右移，用 $p_{\text{next}[k]}$  ( $p_{\text{next}[\text{next}[i]]}$ )与 $t_j$ 比较。

$$\begin{array}{ccccccc}
 t_0 & \dots & t_{j-i-1} & p_0 & \dots & p_{i-1} & t_j & \dots \\
 & & & \parallel & & \parallel & \neq & \\
 & & & p_0 & \dots & p_{i-1} & p_i & \dots
 \end{array}$$

- 修改上面next计算的步骤如下：

(1) 求 $p_0 \dots p_{i-1}$ 中最大相同的前缀和后缀的长度k;

(2) if ( $p_k == p_i$ )

$\text{next}[i] = \text{next}[k];$

else  $\text{next}[i] = k;$

$$\begin{array}{ccccccc}
 p_0 & \dots & p_{i-k} & p_{i-k+1} & \dots & p_{i-1} & p_i & p_{i+1} \\
 & & & & & & \parallel & \\
 p_0 & p_1 & \dots & p_{k-1} & p_k & p_{k+1} & & 
 \end{array}$$

# next数组的计算—算法改进

```
makeNext(PSeqString p, int *next)
{
 int i , k; next[0] = -1;
 for(i=0;i < p->n-1;i++) /* 计算next[i+1] */
 {
 k=next[i];
 while (k >= 0 && p->c[i] != p->c[k])
 k = next[k]; /* p0...pi中最大的相同的前后缀长度k */
 if (p->c[i+1] == p->c[k+1]) next[i+1] = next[k+1];
 else next[i+1] = k+1;
 }
}
```

| 下标i                               | 0  | 1 | 2 | 3  | 4 | 5 | 6 | 7 | 8 |
|-----------------------------------|----|---|---|----|---|---|---|---|---|
| p <sub>i</sub>                    | a  | b | c | a  | a | b | a | b | c |
| 最大相同前后缀长度k                        | -1 | 0 | 0 | 0  | 1 | 1 | 2 | 1 | 2 |
| p <sub>k</sub> 与p <sub>i</sub> 比较 |    | ≠ | ≠ | =  | ≠ | = | ≠ | = | = |
| next[i]                           | -1 | 0 | 0 | -1 | 1 | 0 | 2 | 0 | 0 |

# next数组的计算—算法改进'

```
makeNext(PSeqString p, int *next)
{
 int i = 0, k = -1; next[0] = -1;
 while (i < p->n-1) /* 计算next[i+1] */
 {
 while (k >= 0 && p->c[i] != p->c[k])
 k = next[k]; /* p0...pi中最大的相同的前后缀长度k */
 i++; k++;
 if (p->c[i] == p->c[k]) next[i] = next[k];
 else next[i] = k;
 }
}
```

| 下标i                               | 0  | 1 | 2 | 3  | 4 | 5 | 6 | 7 | 8 |
|-----------------------------------|----|---|---|----|---|---|---|---|---|
| p <sub>i</sub>                    | a  | b | c | a  | a | b | a | b | c |
| 最大相同前后缀长度k                        | -1 | 0 | 0 | 0  | 1 | 1 | 2 | 1 | 2 |
| P <sub>k</sub> 与p <sub>i</sub> 比较 |    | ≠ | ≠ | =  | ≠ | = | ≠ | = | = |
| next[i]                           | -1 | 0 | 0 | -1 | 1 | 0 | 2 | 0 | 0 |

# example-2

- $t = \text{"aabcbabcaabcaababc"}$  ,  $n = 18$   
 $p = \text{"abcaababc"}$  ,  $m = 9$

|    |   |   |    |   |   |   |   |   |
|----|---|---|----|---|---|---|---|---|
| -1 | 0 | 0 | -1 | 1 | 0 | 2 | 0 | 0 |
|----|---|---|----|---|---|---|---|---|

| 下标j | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
|-----|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|
| 目标t | a | a | b | c | b | a | b | c | a | a | b  | c  | a  | a  | b  | a  | b  | c  |
| 模式p | a | b | c | a | a | b | a | b | c |   |    |    |    |    |    |    |    |    |
|     |   | a | b | c | a | a | b | a | b | c |    |    |    |    |    |    |    |    |
|     |   |   | a | b | c | a | a | b | a | b | c  |    |    |    |    |    |    |    |
|     |   |   |   | a | b | c | a | a | b | a | b  | c  |    |    |    |    |    |    |
|     |   |   |   |   | a | b | c | a | a | b | a  | b  | c  |    |    |    |    |    |
|     |   |   |   |   |   | a | b | c | a | a | b  | a  | b  | c  |    |    |    |    |
|     |   |   |   |   |   |   | a | b | c | a | a  | b  | a  | b  | c  |    |    |    |
|     |   |   |   |   |   |   |   | a | b | c | a  | a  | b  | a  | b  | c  |    |    |
|     |   |   |   |   |   |   |   |   | a | b | c  | a  | a  | b  | a  | b  | c  |    |
|     |   |   |   |   |   |   |   |   |   | a | b  | c  | a  | a  | b  | a  | b  | c  |

# 时间复杂性分析

---

## □ 在无回溯的模式匹配算法中

- $j$ 值只增不减，由于 $j$ 的初值为0，循环过程中保持 $j < n$ ，所以循环体中 $j++$ 语句的执行次数不超过 $n$ ，从而 $i++$ 的执行次数也不超过 $n$ 。
- $i$ 的初始值为0，唯一使 $i$ 减少的语句是 $i = \text{next}[i]$ 。计算 $\text{next}$ 数组的时间复杂度为 $O(m)$  [ $m$ 为模式串长度]

## □ 综合上述情况，无回溯模式匹配的时间复杂度为 $O(m+n)$ 。

# 小结

---

## □ 字符串的概念、存储表示以及基本运算的实现

- 字符串是零或多个字符组成的有限序列，它是一种线性结构；
- **重点掌握**字符串在顺序存储和链接存储结构中的实现

## □ 模式匹配算法

- **朴素模式匹配**算法简单，但算法效率低[ $O(n*m)$ ];
- **无回溯模式匹配**方法通过使用next数组避免匹配过程中的回溯操作，从而减少了时间复杂度[ $O(m+n)$ ]。但其空间效率比朴素模式匹配方法降低。

# 内容提要

---

- 字符串及其运算
- 字符串的存储表示
  - 顺序存储
  - 链式存储
- 模式匹配
- 练习



# 练习

---

- 设 $S_1, S_2$ 为串，请给出使 $S_1+S_2=S_2+S_1$ 成立的所有可能的条件（其中 $+$  为连接运算）；

$s_1$ 或 $s_2$ 至少一个为空； $s_1=s_2$ ； $s_1、s_2$ 分别为一个前缀的若干倍

- 设计一个算法来实现字符串逆序存储，要求不另设串存储空间；

利用字符串的结束符 $\backslash 0$ ，作为中间变量

利用 $a=a+b; b=a-b; a=a-b;$

利用 $a=a^b; b=a^b; a=a^b;$ （异或）

- 对 $t=\text{“ababbbaaaba”}$ ,  $p=\text{“aab”}$  进行KMP 快速模式匹配，请画出匹配过程的图示。

# 练习

```
makeNext(PSeqString p, int *next)
{
 int i = 0, k = -1; next[0] = -1;
 while (i < p->n-1) /* 计算next[i+1] */
 {
 while (k >= 0 && p->c[i] != p->c[k])
 k = next[k]; /* p0...pi中最大的相同的前后缀长度
 k */
 i++; k++;
 if (p->c[i] == p->c[k]) next[i] = next[k];
 else next[i] = k;
 }
}
```

|                                   |    |    |   |
|-----------------------------------|----|----|---|
| 下标i                               | 0  | 1  | 2 |
| p <sub>i</sub>                    | a  | a  | b |
| 最大相同前后缀长度k                        | -1 | 0  | 1 |
| P <sub>k</sub> 与p <sub>i</sub> 比较 |    | =  | ≠ |
| next[i]                           | -1 | -1 | 1 |

|   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|
| a | b | a | b | b | a | a | a | b | a |
| a | a |   |   |   |   |   |   |   |   |
|   |   | a | a |   |   |   |   |   |   |
|   |   |   |   | a | a | b |   |   |   |
|   |   |   |   |   | a | a | b |   |   |
|   |   |   |   |   |   | a | a | b |   |