数据结构与算法

主讲教师： 刘峤

第4章 串

第4章 内容提要

- **串的定义与抽象数据结构**
- **串的存储实现**
- **串的模式匹配**

串的定义与抽象数据结构

串的定义

- 串（String）
  - 是字符串的简称
  - 是由零个或多个字符组成的有限序列
  - 一般记为：**s=“a_1a_2 … a_n”**（n≥0）
  - 其中：s为**串名**；用双引号括起来的字符序列是**串值**；
    - **a_i** (0≤i≤n) 可以是字母、数字或其它字符
    - 双引号为串值的**定界符**，不是串的一部分
    - 串中字符的数目n称为串的**长度**

串的定义

- 空串（NULL string）：零个字符构成的串
  - 以两个相邻的双引号来表示空串：s=“”
  - 空串的长度为零
- 空格串：仅由空格组成的的串（例如：s=“␣”）
  - 若串中含有空格，在计算串长时，空格应计入串的长度中
  - 如：s=“I’m␣a␣boy” 的长度为9

主串与子串

- 如果一个串**s1**是另一个串**s2**中**连续的一段子序列**
  - 则称：串**s1**是**s2**的**子串**；称：**s2**是串**s1**的**主串**
- 定义：子串在主串中的位置为：
  - 子串在主串中第一次出现的第一个字符的位置
  - 例如： s1=“How␣are␣you”；s2=“you”
    - 则： s2是s1的子串；s2在s1中的位置 = 9
    - 若给出：s3=“Howare”；s4=“how”
    - 则：s3和s4不是s1的子串

C语言函数库中提供的串处理函数

- 头文件：#include <**stdio.h**>
  - **char *gets(char *string);**
    - 从输入流（缓冲区）中读取字符串，直到出现换行符或读到文件尾为止，最后加上NULL作为字符串结束符
  - **int puts(const char *string);**
    - 向标准输出设备（stdout）写字符串并换行
- 头文件：#include <**string.h**>
  - **extern unsigned int strlen(char *s)；**
    - 从内存的某个位置（可以是字符串开头，中间某个位置，甚至是某个不确定的内存区域）开始扫描，直到碰到第一个字符串结束符'\0'为止，然后返回字符计数值(长度不包含'\0')。

C语言函数库中提供的串处理函数

- 头文件：#include <**string.h**>
  - **extern char *strcat(char *dest, char *src);**
    - 字符串连接：将src指向的字符串添加到dest指向的字符串结尾处（覆盖dest结尾处的'\0'）
  - **extern char *strcpy(char *dest,char *src);**
    - 把从src地址开始且含有'\0'结束符的字符串复制到以dest开始的地址空间，返回指向dest的指针
  - **extern int strcmp(const char *s1,const char *s2);**
    - 两个字符串自左向右逐个字符相比（按ASCII值大小相比较），直到出现不同的字符或遇到'\0'为止。若s1==s2，则返回零；若s1>s2，则返回正数；若s1<s2，则返回负数。

串的存储实现

串与线性表的区别

- 串的逻辑结构和线性表极为相似
  - 区别仅在于串的数据对象约束为字符集
- 串的基本操作和线性表有很大差别
  - 线性表的基本操作以“单个元素”作为操作对象
  - 串的基本操作通常以“串的整体”作为操作对象

串的顺序存储结构

静态分配存储空间

```
#define MAXSIZE 256

char String[MAXSIZE];
```

动态分配存储空间

```
typedef struct{
    char *pch;          // 指针（指向字符数组）
    int   length;       // 串长
} String, *Pstr;
```

顺序串的基本操作：初始化

```
Pstr init_string (int n){
    Pstr pstr = (Pstr) malloc (sizeof(String));
    if(!pstr){ printf("为串分配内存失败!\n"); exit(0);}
    char * pch = (char * ) malloc (n);
    if (pch){
        pstr->pch = pch;
        pstr->length  = 0;
    }
    else{
        printf("为字符数组分配内存失败!\n");
        free(pstr);  pstr = NULL;
    }
    return pstr;
}
```

顺序串的基本操作：串的赋值

```
// 将串dest的值赋给串src，若src非空，则首先释放串src的内存
Pstr assign(Pstr dest, Pstr src){
    int n = src->length; dest->length = n; int i = 0;
    if(dest->pch) free(dest->pch);
    dest->pch = (char *)malloc(n+1);
    if(!dest->pch){printf("分配空间失败\n"); exit(0);}
    for(i = 0; i < n; i++){
        dest->pch[i] = src->pch[i];
    }
    dest->pch[i] = '\0';
    return dest;
}
```

顺序串的基本操作：串的连接

```
Pstr concate (Pstr str1, Pstr str2){ // 将str2连接到str1之后
    String tmp;  assign(&tmp,  str1);  int i, n;
    n = str1->length + str2->length;
    free(str1->pch);  str1->length =  n;
    str1->pch = (char *)malloc(n +1);
    if(!str1->pch){printf("分配空间失败\n"); exit(0);}
    for(i = 0; i < tmp.length; i++)
        str1->pch[i] = tmp.pch[i];
    for(i = 0; i < str2->length; i++)
        str1->pch[tmp.length + i] = str2->pch[i];
    dest->pch[i] = '\0';
    free(tmp.pch);   return str1;
}
```

串的链式存储结构

（请自学）

串的模式匹配

串的模式匹配

- 串的模式匹配是一种重要的串操作
  - 也称为子串定位操作
- 问题描述：
  - **T** 和 **P** 是给定的两个串（P称为模式）
  - 在目标串**T**中找到模式串**P**的过程称为模式匹配
  - 如果在目标串**T**中找到模式串**P**，则称匹配成功
    - 函数返回**P**在**T**中首次出现的位置
  - 否则称匹配不成功，函数返回0

串的简单模式匹配算法

T: b c d a b a a b a b c d a b d c

P: a b c d a b d

算法设计思想

- 从目标串T的第一个字母和模式串P的第一个字母开始比较
- 如果不匹配，则比较T的第二个字母与P的第一个字母
- 依次下去，直到匹配成功
- 或者已经到了T的最后一个字母，匹配失败

串的简单模式匹配算法

i=5　i = i-j+1

T: b c d a b a a b a b c d a b d c

P: a b c d a b d

j=2　j = 0

- 算法性能分析
  - 设： **T** 的长度为 **m**， **P** 的长度为**n** （n ≤ m）
  - 内层循环次数：**≤ n**； 外层循环次数：**≤ (m - n)**
  - 循环次数为：**(m-n+1)*n**
  - 时间复杂度：**O(n*m)**

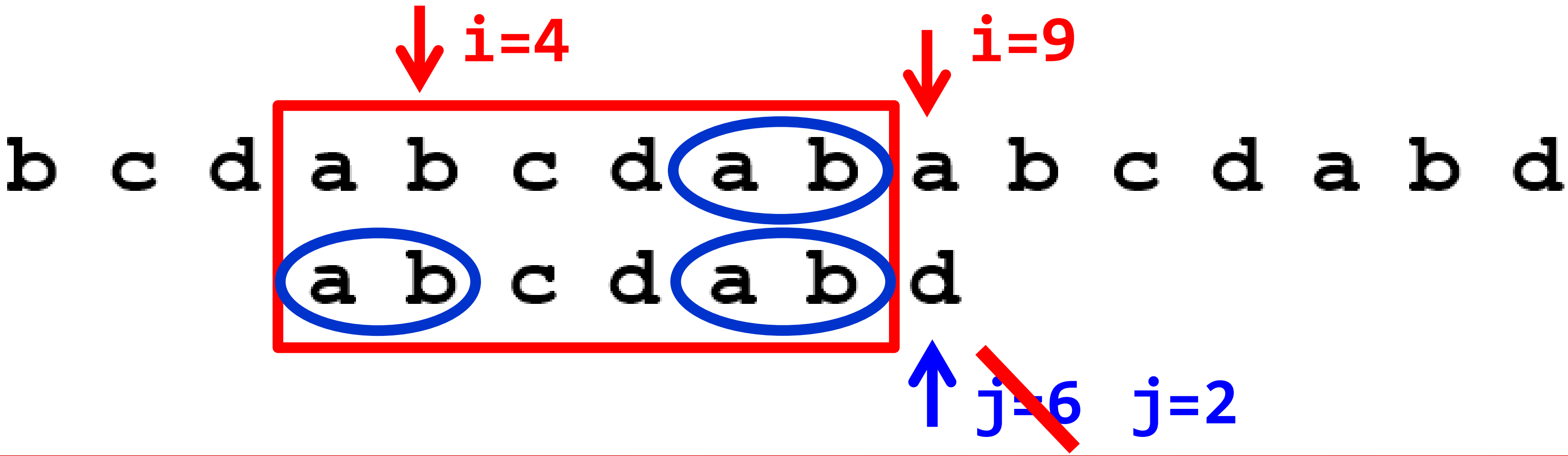
串的简单模式匹配算法

```
int str_match(String *T, String *P){
    int i = 0, j = 0;
    int m = T->length,  n = P->length;
    while (i <= (m - n)){
        j = 0;
        while(T->pch[i]==P->pch[j]){
            i++; j++;
            if(j == n) return (i - n);  // 返回匹配的位置
        }
        i = i-j+1;
    }
    return -1;   // 未找到匹配子串
}
```

算法复杂度?

$$T(n) = O(mn)$$

Knuth-Morris-Pratt算法（简称KMP）

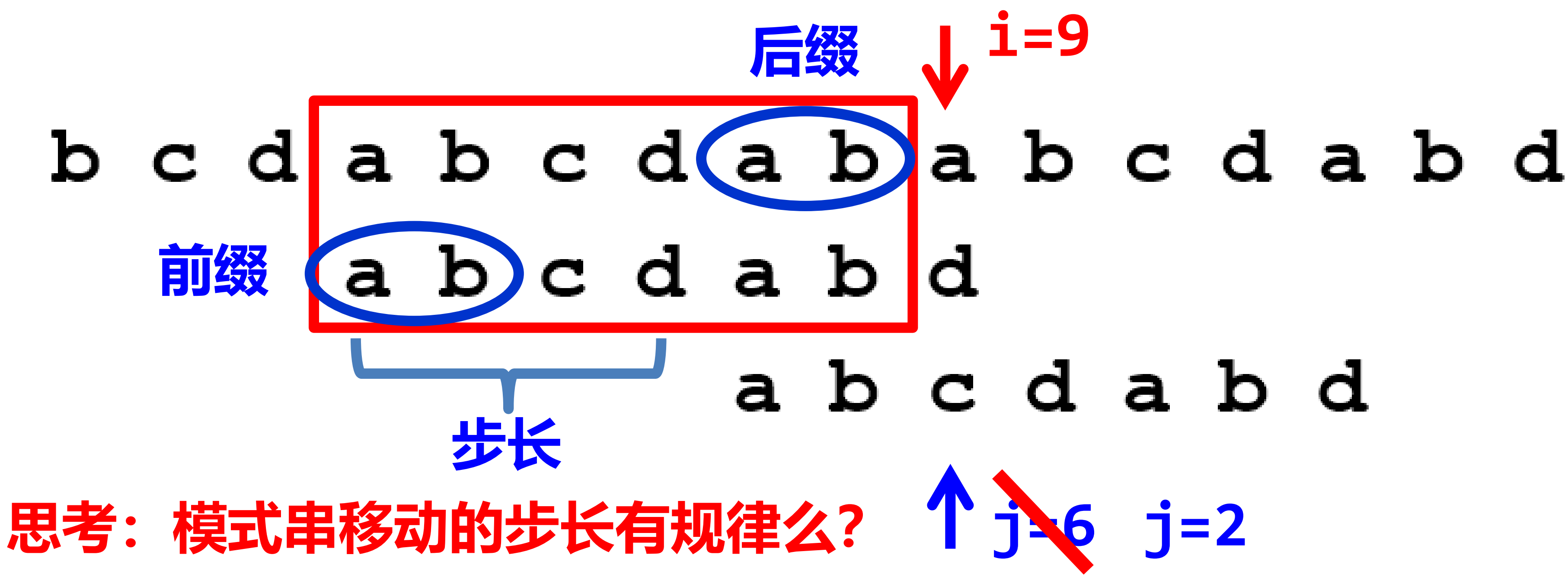


- 简单模式匹配算法存在的问题
  - 当 T->pch[i] != P->pch[j] 时
  - i 指针回溯（j-1）格，同时 j 回退到 0
  - 思考：i 指针有必要回退到 4 么?
    - 提示：匹配过程中可否不移动 i 指针?

KMP算法

- 问题：给定目标串 T 和模式串 P
  - 要求（1）：找出 P 在 T 中首次出现的位置
  - 要求（2）：若 T 不是 P 的子串，则返回零
- 符号约定
  - 设目标串T为"$\mathbf{t_0 t_1 \ldots\ldots t_{m-1}}$"，模式串P为"$\mathbf{p_0 p_1 \ldots\ldots p_{n-1}}$"
  - 将目标串简记为：**T[0:m-1]**，模式串P简记为：**P[0:n-1]**
  - 思考：当目标串中第**i**个字符与模式串中第**j**个字符失配时
    - 若希望不对目标串的下标 **i** 进行回溯操作
    - 目标串中的第**i**个字符下一步应与模式串中哪个字符比较？

KMP算法



- 思考：当T中第 **i** 个字符与P中第 **j** 个字符失配时
  - 若希望不对目标串的下标 i 进行回溯操作
  - **T** 中的第 **i** 个字符下一步应与 **P** 中哪个字符比较?

KMP算法

- 字符串的**真前缀**（**proper prefixes**）
  - 真前缀：除最后一个字符外，一个字符串的全部头部组合
  - 真前缀就是指不包含自身的前缀
  - 例如：字符串 T = ababc （不考虑空字符）
    - 所有的前缀包括：a，ab，aba，abab，ababc
    - 其中真前缀包括：a，ab，aba，abab
- 字符串的**真后缀**（**proper suffixes**）
  - 除第一个字符以外，一个字符串的全部尾部组合
  - 真后缀就是指不包含自身的后缀

KMP算法

- 字符串的前缀函数**next(n)**定义为满足如下条件的子串长度：
  - 长度为n的字符串的最长真前缀同时也是它的真后缀
- 例如：给定字符串 “abababca”
  - next(1)表示长度为1的字符串“a”的前缀函数
    - a没有真前缀 -> next(1) = 0
  - ab的真前缀为a，真后缀为b，a≠b
    - 所以：next(2) = 0
  - aba的真前缀为**a**和ab，真后缀为**a**和ba， “**a**”= “**a**”
    - 所以：next(3) = 1
  - 依此类推：next(4) = 2， next(5) = 3 ……

Partial Match Table

pattern	a	b	a	b	a	b	c	a
next(n)	0	0	1	2	3	4	0	1
index(n)	0	1	2	3	4	5	6	7

- 对于给定字符串 “abababca”
  - 可以根据其前缀函数next(n)构造局部子串匹配表（PMT）
- 思考： PMT对于解决我们的问题有什么作用?
  - 提示：假设模式串当前发生失配的位置为 **j = 5**
  - 提示：**next(n)**和**index(n)**分别表示什么意思?
  - 提示：**j = 5** 表示第6个字符失配，即：**n = 6**

Partial Match Table

j=5

pattern	a	b	a	b	a	b	c	a
next(n)	0	0	1	2	3	4	0	1
index(n)	0	1	2	3	4	5	6	7

- 设模式串当前发生失配的位置为 **j = 5 (n = 6)**
  - **index(6) = 5** 即：当前发生字符失配的位置下标
  - **next(6) = 4** 即：长度为6的子串中前后缀匹配的长度
  - 思考：next(6) 对解决问题有帮助吗？**j - next[j-1] = 2**
  - 思考： 此时模式串向右移动的字符个数应为多少？
  - 思考： 模式串平移后 j 指针位置？ **j = next[j-1] = 3**

Partial Match Table

				j=3		j=5			
pattern	a	b	a	b	a	b	c	a	
index(n)	0	1	2	3	4	5	6	7	8
next(n)	-1	0	0	1	2	3	4	0	1

- 为便于计算：将next数组的元素右移一位
  - 向PMT数组下标为零的位置填充-1
    - 则：模式串右移操作变为： j = next[j]
  - 模式串平移后： j 对应的下标值（3）是？
    - 长度为 j = 5 的子串中最长匹配前后缀的长度
    - 此时j指向指向原失配位置（i 位置不变！）

KMP算法

1. 首先构造部分匹配表（next数组）
   - next[j]的取值只与模式P的前j-1项有关，与目标串T无关
2. 在字符串匹配过程中若发生字符失配现象
   - 保持目标串指针（**i**）不变，根据模式串发生失配的位置（**j**）
   - 查next表得到模式串右移后指针 **j** 的位置：**next[j]**
3. 比较：**P[next[j]]** 与 **T[i]**
   - 若：**P[next[j]] == T[i]**；则继续比较后续元素
   - 若：**P[next[j]] ≠ T[i]**；则令 **j = next[j]**
   - 若：**next[j]==-1**；将P右移一格，重新开始比对
   - 重复以上步骤直至匹配成功或达到目标串的结尾

KMP算法

```
int KMP(Pstr T, Pstr P, int *next) {
    int i =0, j = 0, m = T->length, n = P->length;
    while(i <= m - n){
        while( j == -1 || (j < n && T->pch[i] == P->pch[j])){
            i++;  j++;
        }
        if (j == n)  return (i - n);  // 匹配成功
        j = next[j];
    }
    return -1;   // 匹配失败
}
```

求解PMT：填充next数组

i=7　j=5

b c a b a b a c b a b c d a b d

k=3　a b a b a b c a

next									
next	-1	0	0	1	2	3	4	0	1
	0	1	2	3	4	5	6	7	8

- 设目标串T为“$t_0t_1......t_{m-1}$”，模式串P为“$p_0p_1......p_{n-1}$”
- 将目标串简记为：T[0:m-1]，模式串P简记为：P[0:n-1]
- 若当前在 **T** 的下标位置 **i** 处发生失配
- 设：此时P中对应失配位置下标为 j > 0
- 设：k为满足条件 P[**0:k-1**] = T[**i-k:i-1**] 的最大值（1≤k<j）

求解PMT：填充next数组

i=7　j=5

b c a b a b a c b a b c d a b d

a b a b a b c a

next	-1	0	0	1	2	3	4	0	1
	0	1	2	3	4	5	6	7	8

next[j]=k=3

- 若当前在 **T** 的下标位置 **i** 处发生失配
- 设：此时P中对应失配位置下标为 j > 0
- 设：k为满足条件 P[**0:k-1**] = T[**i-k:i-1**] 的最大值（1≤k<j）
- 由于： P[**j-k:j-1**] = T[**i-k:i-1**]
- 所以：**P**[**0:k-1**] = **P**[**j-k:j-1**]　　**思考：有何意义?**

求解PMT：填充next数组

- next[j]的形式化定义

$$next[j]=\begin{cases} -1 & j=0 \\ 0 & j=1 \\ \max\{k \mid P[0:k-1]=P[j-k:j-1]\} & 1\le k<j \end{cases}$$

- 若：k为模式串的子串P[0:j-1]中相等的真前后缀的最大长度
  - 则有：P[0:k-1] = P[j-k:j-1]
- next[j]表示当 **P** 中第 **j** 个字符与 **T** 中第 **i** 个字符失配时
  - 在模式串中需重新与目标串第 **i** 个字符进行比较的字符位置
- 求next[j]的过程即填充PMT的过程

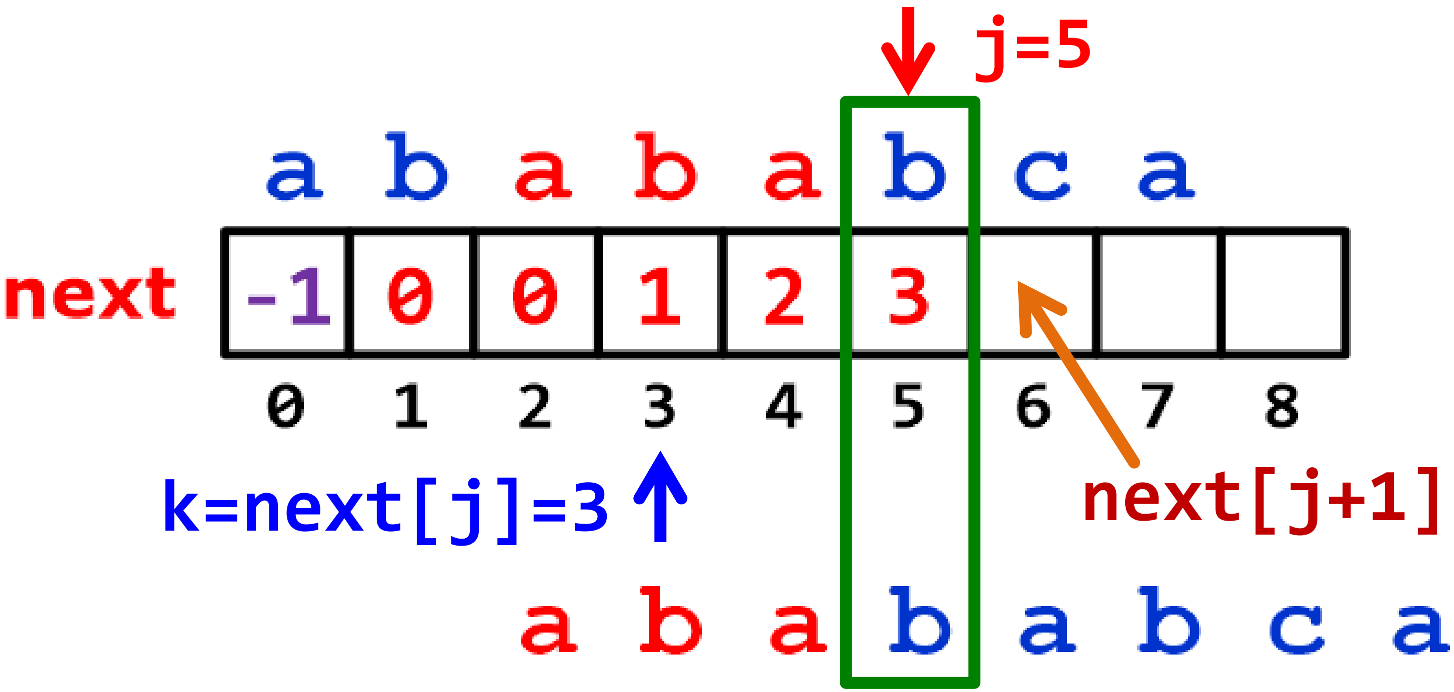
求解PMT：填充next数组

$$next[j]=\begin{cases} -1 & j=0 \\ 0 & j=1 \\ \max\{k \mid P[0:k-1]=P[j-k:j-1]\} & 1\le k<j \end{cases}$$

- 已知：next[0] = -1； next[1] = 0 **next[2] =?**
- 设：next[j] = k (j > 1) 注意：k是最优值
  - 由next[j]的定义可知：P[0 : k-1] = P[j-k : j-1]
- 若：P[j] = P[k]?
  - 则：next[j+1] = k+1 = next[j]+1

求解PMT：填充next数组

- 若：**P[j]=P[k]** 则：**next[j+1]=k+1=next[j]+1**



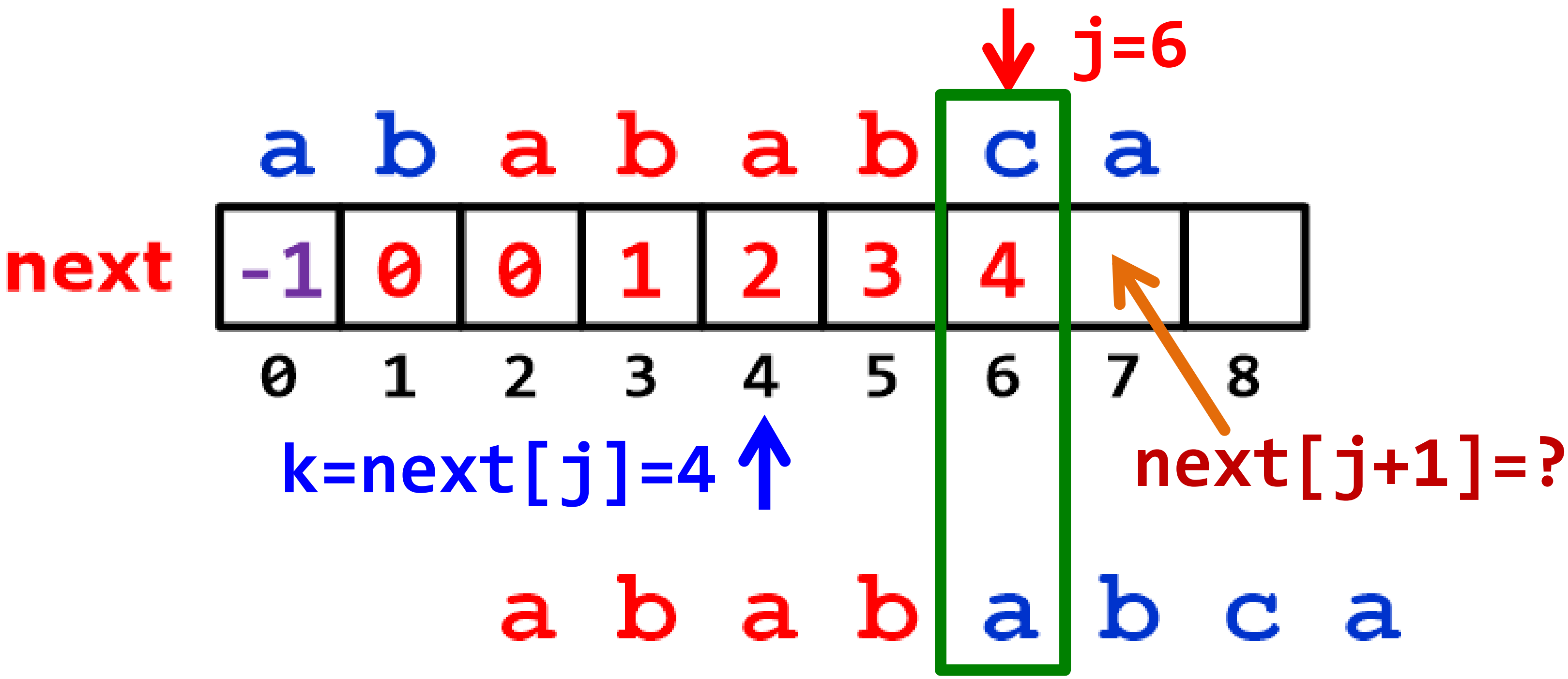
P[j]=P[5]=‘b’

P[k]=P[3]=‘b’

➔ P[j] = P[k]

求解PMT：填充next数组

若：P[j] ≠ P[k]?　则：P[0:k] ≠ P[j-k:j]



P[j]=P[6]='c'

P[k]=P[4]='a'

➔ P[j] ≠ P[k]

求解PMT：填充next数组

j=6

a b a b a b c a

	0	1	2	3	4	5	6	7	8
next	-1	0	0	1	2	3	4		

next[k]表示?

k=4　　$\boxed{\text{a b a b}}$ a b c a

- 若： P[j]≠P[k] 思考： next[j+1] =?
  - 需回溯检查是否存在 k′ < k 使得
    - P[0:k′-1] = P[j-k′:j-1]
  - 仍为串匹配问题：应将模式串右移多少位?
    - 令：k′=next[k]；使P[k′]与目标串中的P[j]对齐

求解PMT：填充next数组

- 小结：当 P[j] ≠ P[k] 时的处理流程
  - 首先令： k′ = next[k]
    - 意思是：将 P[k′] 与 P[j] 对齐
  - 若：P[j] = P[k′]
    - 则：next[j+1] = k′+1 =next[k′]+1
  - 若： P[j] ≠ P[k′]，则继续匹配直至
    - P[j]与某个字符k″匹配成功：next[j+1] = k″+1
    - 或确认不存在匹配对象： next[j+1] = 0
  - 思考：怎样确认不存在P[j]的匹配对象？ **k == -1**

KMP算法：构造部分匹配表（next数组）

```
void buid_table(Pstr P, int *next){
    int j = 0,  k = -1, n = P->length;  next[0] = -1;
    while(j <n){
        if (k == -1 || P->pch[j] == P->pch[k] ){
            next[j+1] = k + 1;
            j++; k++;
        }
        else{
            k = next[k];
        }
    }
}
```

思考：k=-1什么意思?

k=-1表示对当前字符 j

在模式串中未找到匹配字符

算法复杂度?

$$T(n) = O(n)$$

本章小结

- 串是由零个或多个字符组成的序列
- 串的存储方式：线性存储和链式存储
- 串的基本操作：串赋值、判相等、求串长、串连接、求子串
  - 其他的操作可以通过这五种操作来实现
- 串中任意一个连续字符组成的子序列称为该串的子串
  - 包含子串的串相应地称为主串
- 若两个串的串长相等且对应位置元素相同，则这两个串相等
- 空串是任意串的子串，空格串的长度等于其包含的空格个数

电子科技大学

UESTC

1956