数据结构

第三讲　字符串

孙猛

http://www.math.pku.edu.cn/teachers/sunm

2017年9月21日

李冕《两相思》

枯眼望遥山隔水，往来曾见几心知？
壶空怕酌一杯酒，笔下难成和韵诗。
途路阻人离别久，讯音无雁寄回迟。
孤灯夜守长寥寂，夫忆妻兮父忆儿。

吴绛雪《四肘山水诗》

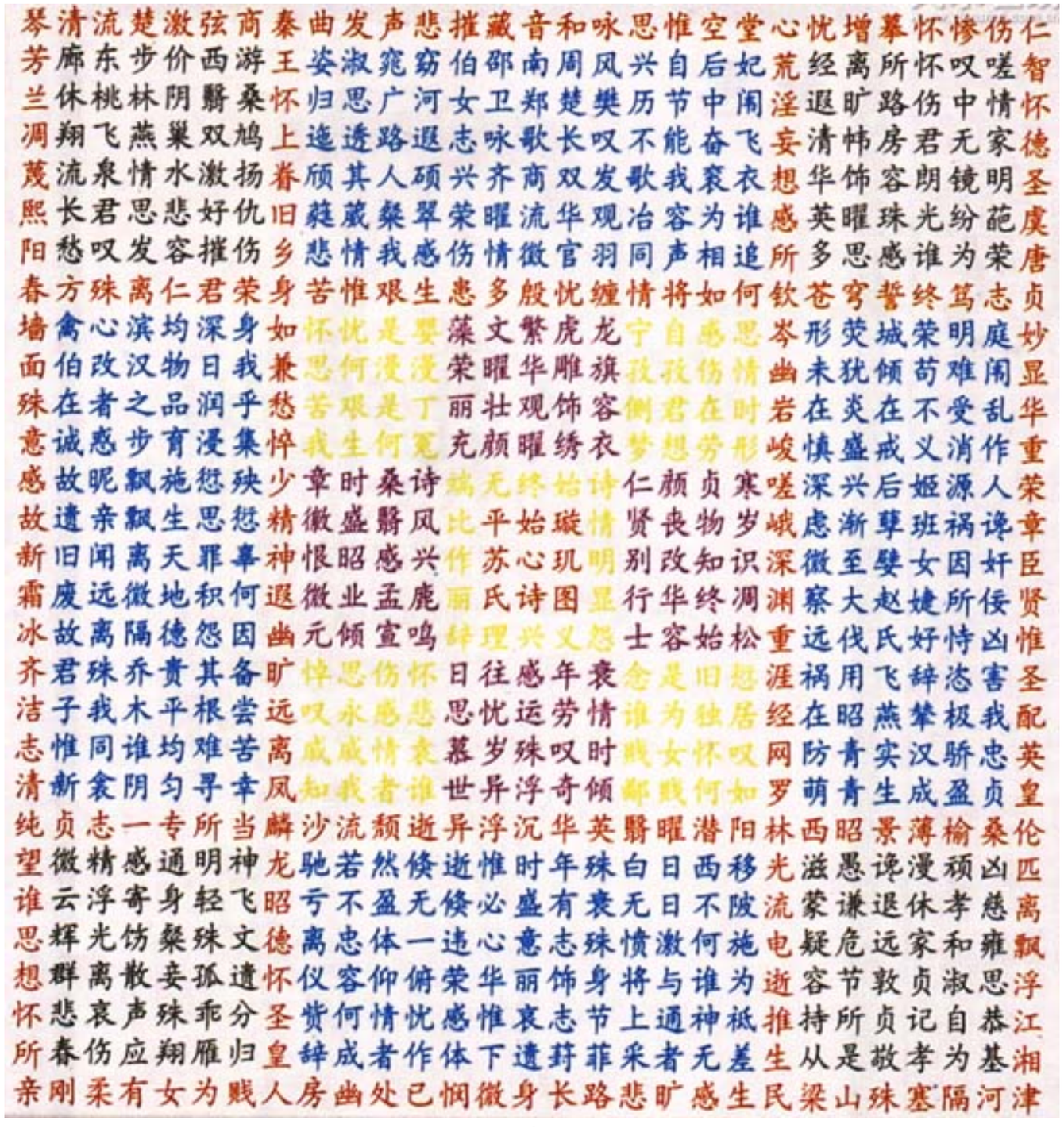
莺啼岸柳弄春晴夜月明

香莲碧水动风凉夏日长

秋江楚雁宿沙洲浅水流

红炉透炭炙寒风御隆冬

苏蕙《璇玑图》



课程内容

- 字符串及其抽象数据类型
- 字符串的实现
- 模式匹配

字符串及其抽象数据类型

- 基本概念
  - 字符串，简称串，一种特殊的线性表，表中的每个元素都是一个字符。
- 一个串可以记为$s = "s_0\ s_1 \ldots s_{n-1}"\ (n \geq 0)$
  - s是串的名字，
  - 字符序列$s_0\ s_1 \ldots s_{n-1}$是串的值，
  - 字符个数称为该串的长度。
- 长度为0的串称为空串，写成s=　""，注意与空白字符构成的串s=" "相区分。

子串、主串与子串位置

- 子串
  - 字符串s1中任意个连续的字符组成的子序列s2被称为是s1的子串，而称s1是s2的主串。
    - 空串是任意串的子串
    - 除s外，s的其他子串称为s的真子串
  - 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。

A= "PEKINGUNIVERSITY"
　　1　　7

B= "UNIVERSITY"

字符串的相等与字典序关系

- 两个字符串**相等**
  - 两个字符串的长度相等，
  - 且各个对应位置上的字符都相同。

- 如果整个字符集上有全（线）序关系，则两个字符串之间有如下**字典序关系**：
  - 设$A = a_0 a_1 \ldots a_{n-1}$, $B = b_0 b_1 \ldots b_{m-1}$, 则$A < B$：
    - 若存在 k使 $a_i = b_i$ $(i = 0,1, \ldots k\text{-}1)$ ， 但是 $a_k < b_k$ ；
    - 或者$n < m$， 且$a_i = b_i$ $(i = 0,1, \ldots n\text{-}1)$。

字符串的拼接

- 串与串的最重要运算是拼接（concatenate）。
- 字符串$s_1 = a_0a_1\cdots a_{n-1}$，$s_2 = b_0b_1\cdots b_{m-1}$的拼接为串

$$s = concat(s_1, s_2) = a_0a_1\cdots a_{n-1}b_0b_1\cdots b_{m-1}$$

- 显然s的长度为s_1与s_2的长度之和。

子串的多次出现与重叠

- 如果 s_1 是 s_2 的子串，s_2 里可能出现多个与 s_1 相同的段，这时说 s_1 在 s_2 里多次出现。
- 注意： s_1 在 s_2 中的多个出现可能不独立，相互重叠。例如
  - babb 在 babbabbbbabb 里有三个出现，前两个有重叠

两种特殊子串

- 如果存在 s' 使 $s_2 = s_1 + s'$，称 s_1 为 s_2 的一个**前缀**。
- 如果存在 s 使得 $s_2 = s + s_1$，称 s_1 为 s_2 的一个**后缀**。
- 直观上，一个串的前缀就是该串开头的一段字符构成的子串，后缀就是该串最后的一段字符构成的子串。
- 显然，空串和 s 既是 s 的前缀，也是 s 的后缀。

其它有用的串运算

- 串 s 的 n 次幂 s^n 是连续 n 个 s 拼接而成的串。
- 串替换，指将一个串里的一些（互不重叠的）子串代换为另一些串得到的结果（由于可能重叠，需规定代换的顺序，如从左到右）。
- 还有许多有用的串运算，可以参考C语言或其他语言的字符串类型。

字符串的理论

- 字符串集合和拼接操作构成了一种代数结构
  - 空串是拼接操作的“单位元”（幺元）
    有结合律，无交换律
  - 串集合加上拼接操作，构成一个半群
    一种典型的非交换半群
  - 有单位元，因此是一个幺半群
- 关于串的理论有许多研究工作
  - 基于串和串替换，研究者提出了 post 系统
    这是一种与图灵机等价的计算模型
  - （串）重写系统（rewriting system）是计算机理论的一个研究领域，一直非常活跃，有许多重要结果和应用，例如现在应用非常广泛的Maude语言。

抽象数据类型

ADT String **is**
operations
String createNullStr (void)
创建一个空串。
int IsNullStr (String s)
判断串s是否为空串，若为空串，则返回1，否则返回0。
int length (String s)
返回串s的长度。
String concat (String s1, Sting s2)
返回将串s1和s2拼接在一起构成的一个新串。
String subStr (String s, **int** i, **int** j)
在串s中，求从串的第i个字符开始连续j个字符所构成的子串。
int index (String s1, String s2)
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。
String subst(String s1, String s2, String s3)
做出将串s1里的子串s2都替换为s3的结果串。
end ADT String

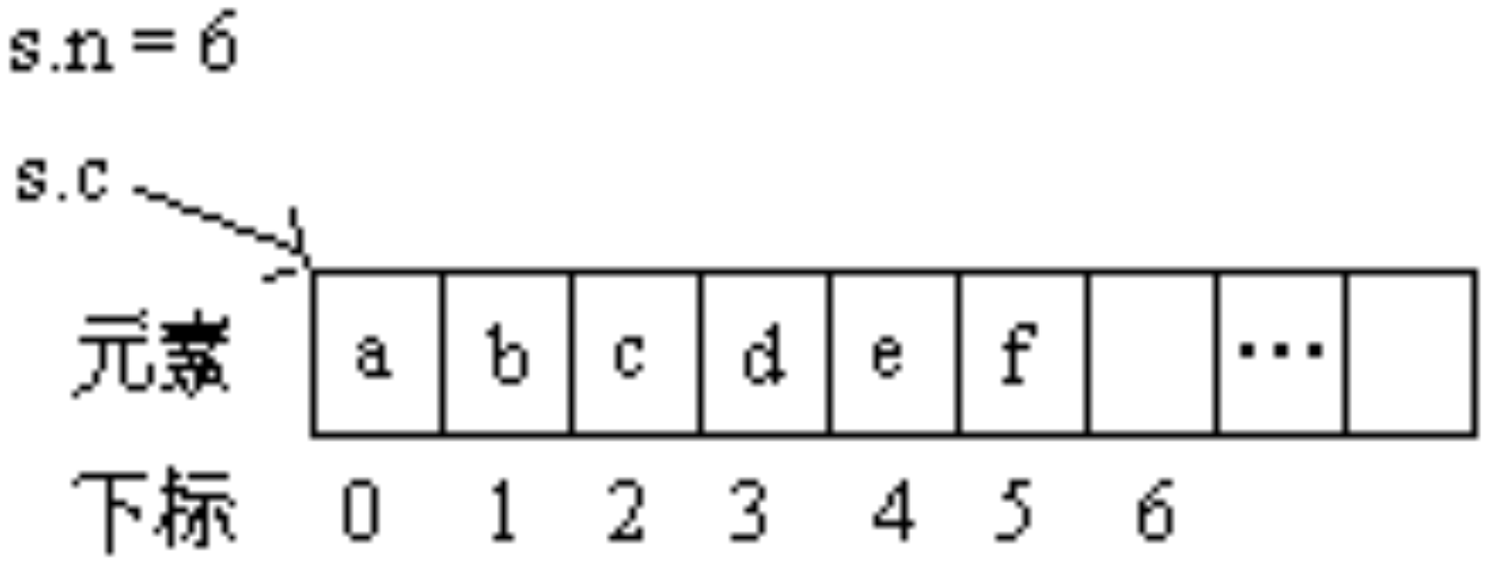
字符串的顺序表示

- 字符串的顺序表示，就是把串中的字符，顺序地存储在一组地址连续的存储单元中。其类型定义为：

```
struct SeqString {          /* 顺序串的类型 */
    int   MAXNUM;           /* 串允许的最大字符个数 */
    int   n;                /* 串的长度，n≤MAXNUM */
    char  *c;
};
typedef  struct SeqString  *PSeqString;
```

顺序表示示例

- 字符串s=“abcdef”，用顺序表示方式，假设s是struct　SeqString类型的变量，那么它的元素在数组中的存放方式如下图所示：



创建空顺序串

```
PSeqString createNullStr_seq( int m ) {
PSeqString pstr = (PSeqString)malloc(sizeof(struct SeqString));
  if (pstr!=NULL){
     pstr->c= (char*)malloc(sizeof (char)*m);
     if(pstr->c){
     pstr->n=0;  pstr->MAXNUM=m;
     return  pstr;
     }
     else free(pstr);
     }
  printf("Out of space!!\n");
  return NULL;
}
```

求顺序串的子串

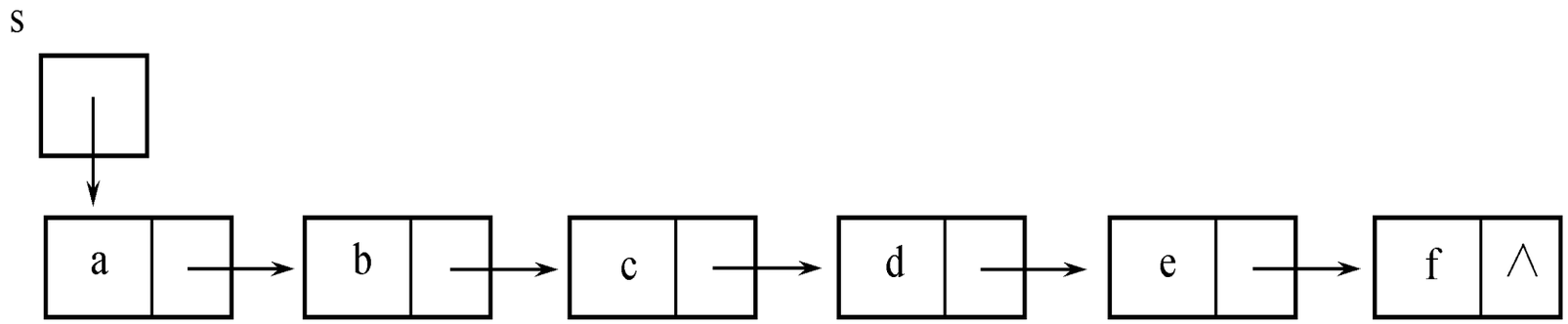
```
PSeqString subStr_seq(PSeqString s,int i,int j) {
PSeqString s1;
  int k;
  s1 = createNullStr_seq(j);      /* 创建一空串 */
  if (s1==NULL) return  NULL ;
  if ( i>0 && i<=s->n && j>0 ) {
    if ( s->n<i+j-1 )  j = s->n-i+1;
      /*若从i开始取不了j个字符,则能取几个就取几个*/
    for (k=0;k<j;k++)  s1->c[k]=s->c[i+k-1];
   s1->n=j;
  }
  return s1;
}
```

字符串的链接表示

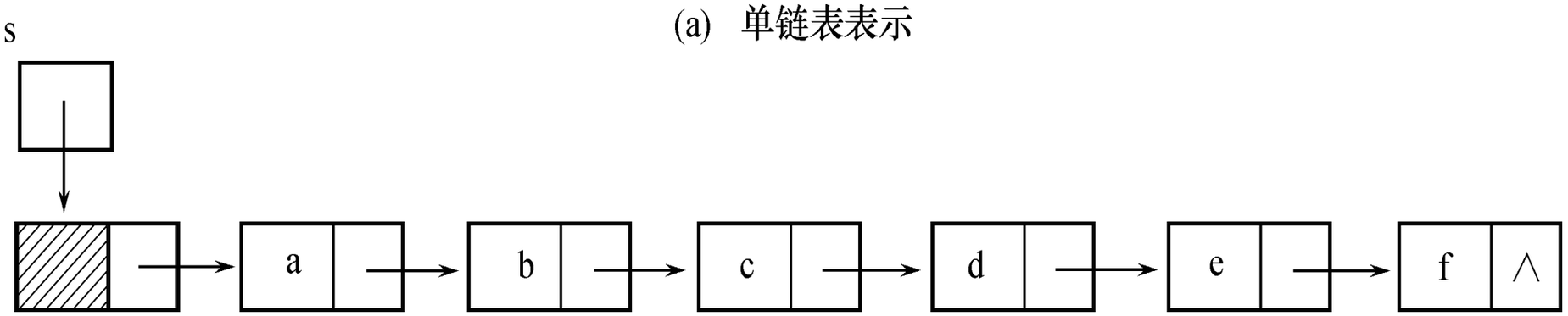
- 字符串的链接表示中，每个结点包含两个字段：字符和指针，分别用于存放字符和指向下一个结点的指针。这样一个串就可用一个单链表来表示，其类型定义为：

```
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  *LinkString; /* 链串的类型 */
```

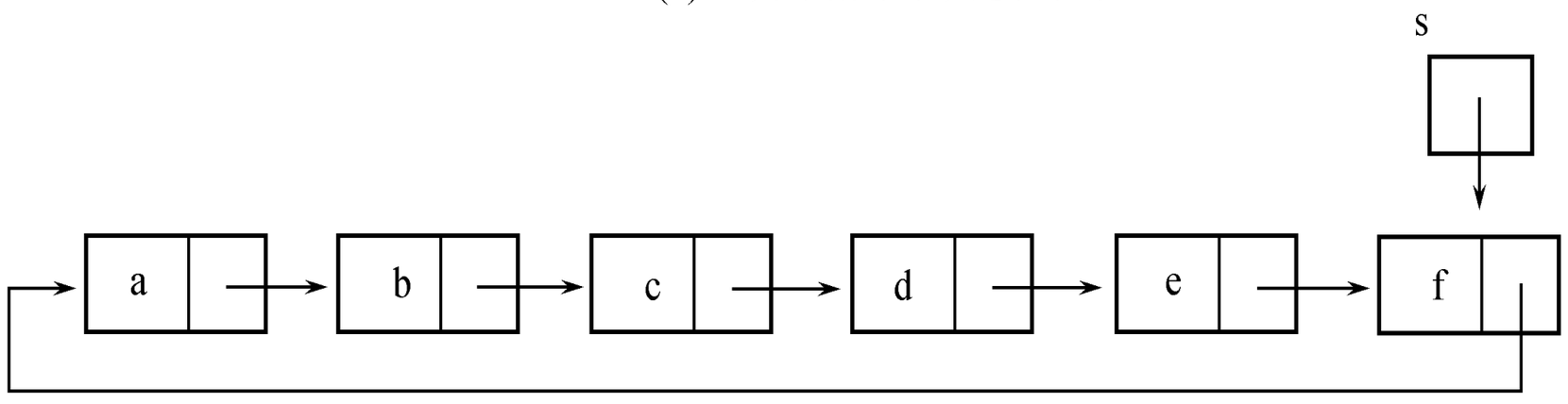
链接表示示例s=“abcdef”



(a) 单链表表示



(b) 带头结点的单链表表示



(c) 循环表表示

创建带头结点的空链接串

```
LinkString  createNullStr_link( void ) {
  /*创建带头结点的空链串*/
  LinkString pst;
  pst = (LinkString)malloc( sizeof(struct StrNode) );
  if (pst!=NULL)  pst->link = NULL;
  else  printf("Out of space! \n");  /*创建失败*/
  return  pst;
}
```

求单链表示的串的子串

LinkString subStr_link(LinkString s,int i,int j)

- **求从 s 所指的带头结点的链串中第 i (i>0) 个字符开始连续取 j 个字符所构成的子串。**
- **这里首先要为链串结构和头结点申请空间，创建一个空链表，这由前面的算法可以实现。**
- **然后判断所给参数i、j的值是否合理，i、j的取值应满足i>0，j>0。**
- **接着从s -> head开始找第 i 个结点，找到后，就从该结点开始，为子串中的结点申请空间，并将元素值复制过去。**

求单链表示的串的子串

LinkString subStr_link(LinkString s, int i, int j)

```
{
 LinkString s1;
 PStrNode p,q,t;
 int k;
    /* 创建一空串 */
 s1 = createNullStr_link( );
 if( s1 == NULL ) {
   printf( "Out of space!\n" );
    return  NULL ;
   }
 if (i<1 || j<1 ) return s1 ;
    /* i,j值不合适，返回空串 */
 p = s; /* p为主串*/
 for (k=1;k<=i;k++)
     /*找第i个结点*/
   if ( p != NULL) p = p->link;
    else  return s1 ;
 if (p==NULL) return s1 ;
 t = s1; /* t为子串尾*/
 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) {
        printf( "Out of space!\n" );
        return s1 ;
      }
     q->c = p->c;
     q->link = NULL;
     t->link = q;
      /* 结点放入子链串中 */
     t = q;
     p = p->link;
   }
 return s1 ;
}
```

字符串的实现

- 无论是顺序表示还是链接表示,都可以看作特殊的线性表的实现方式
- C语言中的字符串直接有字符数组表示，以"\0"作为串结束的符号，也可以看作一种实现方式
- 许多语言提供了标准字符串库，如C语言标准库有一组字符串函数（string.h）
- 支持不同的字符串操作，可能需要不同的实现

模式匹配（子串查找）

假设有两个串

$t=t_0\ t_1\ t_2 \ldots\ t_{n-1}$　　目标(串)

$p=p_0\ p_1\ p_2 \cdots\ p_{m-1}$　　模式(串)

通常$m << n$。

模式匹配就是在目标串 t 中查找与模式串 p 相同的子串的过程。

如前所述，串匹配是最重要的字符串操作，也是其他许多重要字符串操作的基础。实际中 n 可能非常大，m 也可以有一定规模，也可能需要做许多模式串和/或许多目标串的匹配，有关算法的效率非常重要。

字符串模式匹配

- 许多计算机应用的最基本操作是字符串匹配。如
  - 用编辑器或字处理系统工作时，在文本中查找单词或句子（中文字或词语），在程序里找拼写错误的标识符等
  - email 程序的垃圾邮件过滤器，google 等网络搜索引擎
  - 各种防病毒软件，主要靠在文件里检索病毒模式串
- 在分子生物学领域：DNA 细胞核里的一类长分子，在遗传中起着核心作用。DNA 内有四种碱基：腺嘌呤(adenine)，胞嘧啶(cytosine)，鸟嘌呤(guanine)，胸腺嘧啶(thymine)。它们的不同组合形成氨基酸、蛋白质和其他更高级的生命结构
  - DNA 片段可看作是a,c,g,t构成的模式，如 acgatactagacagt
  - 考查在蛋白质中是否出现某个 DNA 片段，可看成与该 DNA 片段的串匹配问题。DNA 分子可以切断和拼接，切断动作由各种酶完成，酶也是采用特定的模式确定剪切位置

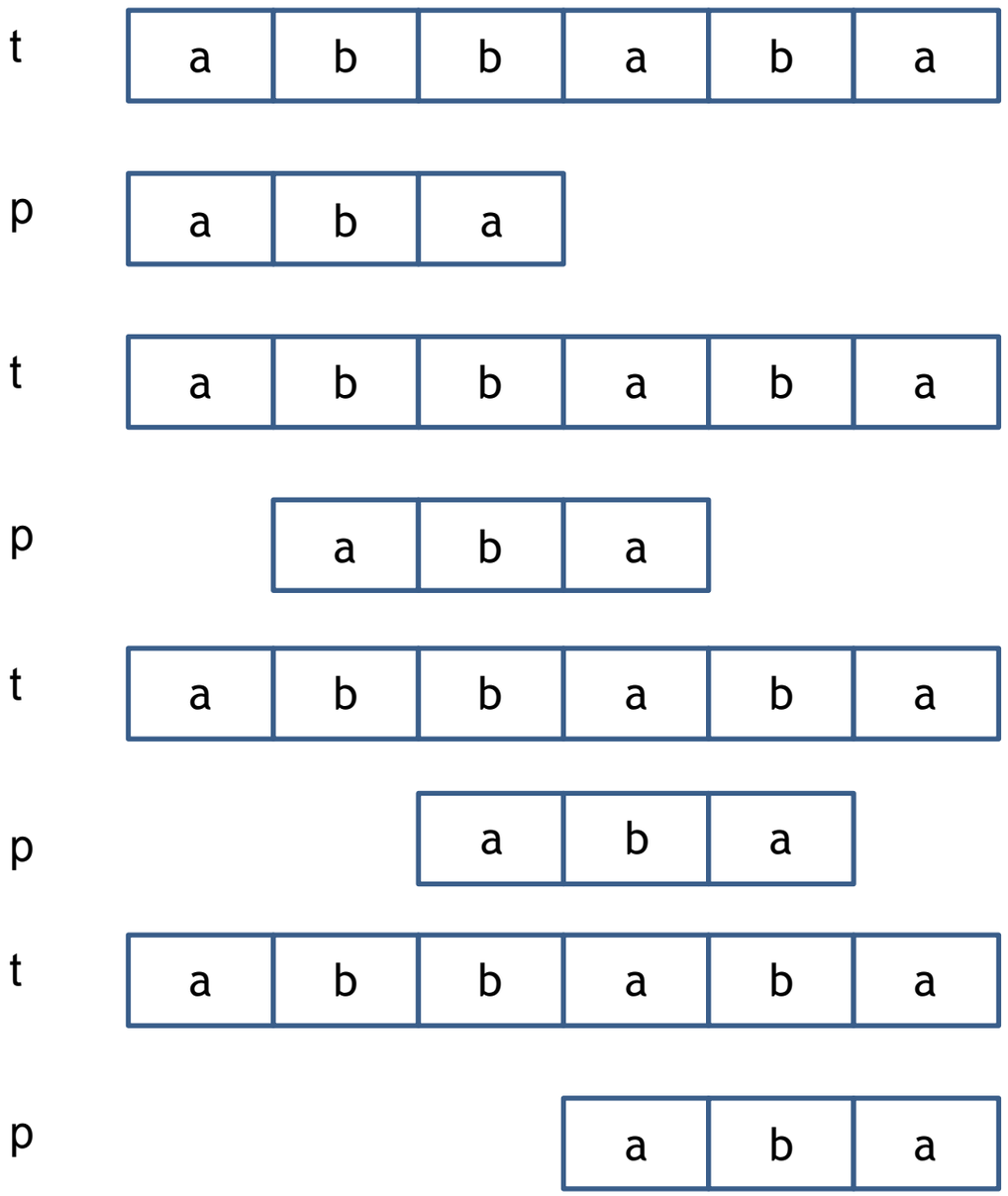
字符串模式匹配

- 实际中模式匹配的规模n和m可能非常大，而且有时间要求
  - 被检索的文本可能很大
  - 网络搜索需要处理亿万的网页
  - 防病毒软件要在合理时间内检查数以十万计的文件（以 GB 计）
  - 运行在服务器上的邮件过滤程序，可能需要在一秒钟的时间内扫描数以万计的邮件和附件
  - 为疾病/药物研究/新作物培养等生物学工程应用，需要用大量 DNA模式与大量 DNA 样本（都是 DNA 序列）匹配
- 由于在诸多领域的重要应用，串模式匹配问题已成为一个极端重要的计算问题。高效的串匹配算法非常重要
  - 有几个集中关注字符串匹配问题的国际学术会议，有过专门的国际竞赛（wiki页http://en.wikipedia.org/wiki/String_searching_algorithm）
  - 目前全世界一大半的计算能力是在做串模式匹配

字符串模式匹配算法

- 还需注意不同的实际需要，如
  - 用一个模式在很长的目标串里反复匹配（确定出现位置）
  - 一组（可能很多）模式，在一个或一组目标串里确定是否有匹配
- 不同算法在处理不同实际情况时可能有不同的表现
  - 人们已经开发出一批有意义的算法（进一步情况见 wiki）
- 粗看，字符串匹配是一个很简单的问题
  - 字符串是简单数据（字符）的简单序列，结构也最简单（顺序）
  - 很容易想到最简单而直接的算法
  - 但事实是：直接而简单的算法并不是高效的算法
    - 因为它可能没有很好利用问题的内在结构
  - 字符串匹配“貌似简单”，但人们已开发出许多“大相径庭”的算法

朴素的模式匹配



朴素的模式匹配算法

求串p在串t中第一次出现的位置，即p的第一个元素在t中的序号(下标+1)

```
int index( PSeqString t,  PSeqString p) {
    int i = 0, j = 0; /*初始化*/
    while ( i < p->n && j < t->n )    /*反复比较*/
        if (p->c[i] == t->c[j]) {          /* 继续匹配下一个字符 */
            ++ i; ++ j;
        }
        else {                             /* i, j值回溯，再开始下一位置的匹配 */
            j =j- i + 1; i = 0;
        }
    if (i >= p->n)  return( j - p->n + 1);  /* 匹配成功*/
    else return 0;                                  /* 匹配失败 */
}
```

算法分析

- 朴素的模式匹配算法直观简单, 易理解
- 最坏的情况下，每趟比较都在最后出现不等，最多比较$n-m+1$趟, 总比较次数为$m \times (n-m+1) = O(m \times n)$
  - 例如：t=00000000000000000000000001, p=001
  - 匹配过程有回溯
- 算法效率低的原因：把每次字符比较看作完全独立的操作
  - 完全没有利用字符串本身的特点（每个字符串都是特殊的）
  - 没有利用前面已做过的字符比较得到的信息
- 从数学上看，这样做相当于认为目标串和模式串里的字符都是随机量，而且有无穷多可能取值，两次字符比较相互无关也不可借鉴
  - 实际字符串取值来自一个有穷集合
  - 每个串都有穷。特别是模式串通常不太长，而且可能反复使用

· 设有主串 *t* =“abababbababaaba”，

模式串*p* =“**ababa**”。

第一趟匹配

abababbababaaba
ababa

能否跳到第六趟?

第二趟匹配

abababbababaaba
 ababa

第三趟匹配

abababbababaaba
 ababa

第四趟匹配

abababbababaaba
 ababa

第五趟匹配

abababbababaaba
 ababa

第六趟匹配

ababbabbababa
ababa

第七趟匹配

ababbabbababa
ababa

第八趟匹配

ababbabbababa
ababa

第九趟匹配

ababbabbababa
ababa

朴素的模式匹配算法效率不高

影响效率的关键因素是有不必要的回溯

算法中没有利用前面已进行的字符比较得到的信息

提高匹配速度

- 如果在匹配过程中一旦p_i和t_j不相等，即:

$$p_0 = t_{j-i}, p_1 = t_{j-i+1}, \cdots, p_{i-1} = t_{j-1}, p_i \neq t_j$$

- 希望能找到一个大于等于1的右移的位数, 确定 p 和 t 继续比较的字符
- 希望匹配过程对于 t 是无回溯的:
  - 右移若干位后, 立即用 p 中一个新的字符 p_k 和 $t_j\left(t_{j+1}\right)$ 继续进行比较
- 最好能通过对p 的分析得到右移位置（p_k）

关键问题: 如何找到这个k

无回溯的匹配模式：KMP算法

- 由 D.E.Knuth 和 V.R. Pratt 提出, J.H.Morris 几乎同时发现这一算法。因此又称为 **KMP 算法**。
- 这是本课程中第一个非平凡的算法： 基于对问题的深入分析和理解。这个算法并不太复杂，但非常巧妙，效率较高。

KMP算法的next数组

- 与p_i对应的k值与被匹配的目标串无关。通过对模式串p的预先分析，可以得到每个i对应的k值。
- 假设p的长度为m，现在需要对每个$i(0 \leq i < m)$算出一个对应的k值并保存起来，以便在匹配中使用。
- 把这m个值存入数组next，用next[i]表示与i对应的k值。
- 一种特殊情况：当p_i匹配失败时，用它之前任何字符与t_j比较都无意义，这时应该用p_0从头开始与t_{j+1}比较。我们在next[i]里保存 -1 表示这种特殊情况。
- 显然，对于任何模式都有：next [0] = -1。

假设next已经建立好

- 无回溯的模式匹配算法的基本思想：匹配中 $p_i \neq t_j$ 时，通过next取得应与目标串当前字符匹配的模式串里的字符下标：
  - 若next [i] ≥ 0，右移 i - next[i] 个字符（也就是说，让 i 取 next[i] 的值）， 下一步用 $p_{next[i]}$ 与 t_j 比较
  - 若next [i] = -1，下一步用 p_0 与 t_{j+1} 比较

模式匹配的核心循环

```
while ( i < p->n && j < t->n )
   if (p->c[i] == t->c[j]) { ++ i; ++ j;   }
   else { j =j- i + 1; i = 0;  }
```

- **假设数组next已经建好：**

```
while (i < p->n && j < t->n) {              /*i, j 是两串的当前位置*/
   if (i == -1) {  i++; j++; }
   else if (p->c[i] == t->c[j]) {  i++; j++; }
   else  i = next[i] ;                      /*与朴素的{j = j-i+1; i = 0;}对应*/
}
```

- **前两个条件(蓝色)可以用 II 合并（执行的操作一样）**

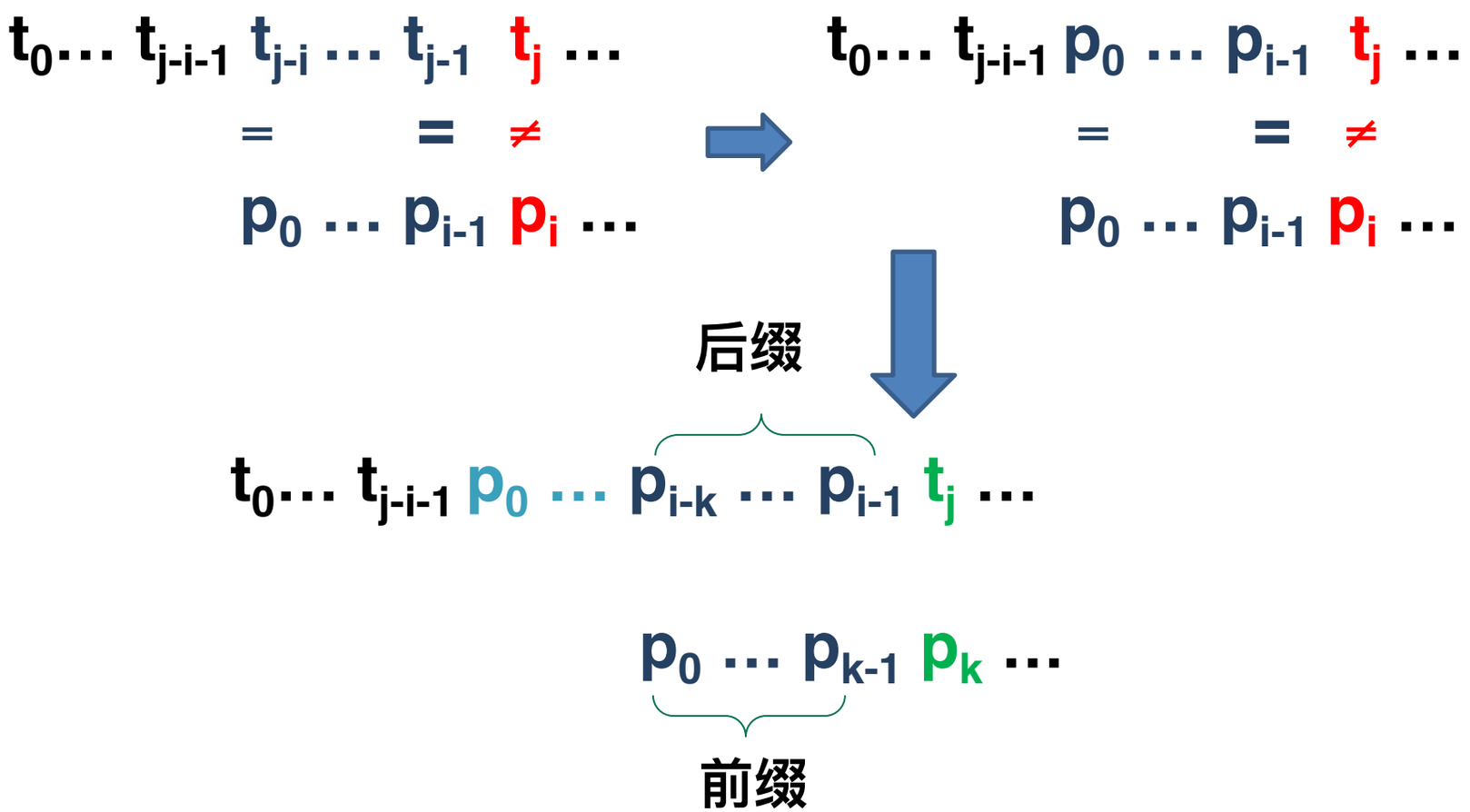
```
while (i < p->n && j < t->n) {
   if (i == -1|| p->c[i] == t->c[j]) {  i++; j++; }
   else  i = next[i] ;
}
```

无回溯的模式匹配函数

```
int  pMatch (PSeqString t, PSeqString p, int *next) {
    int i = 0, j = 0;                                  /*初始化*/
    while (i < p->n && j < t->n) {                     /*反复比较*/
        if (i==-1 || p->c[i] == t->c[j]) {
            i++; j++;
        }
        else  i = next[i] ;
    }
    if (i >= p->n)
        return (j - p->n + 1);
                        /*匹配成功，返回p中第一个字符在t中的序号*/
    return   0;                                        /*匹配失败*/
}
```

next数组的性质

匹配中出现$p_i \neq t_j$时:



next的存在

- 需要考虑的是模式串 p 的子串 $p_0 \cdots p_{i-1}$ 里相同的前缀与后缀：
  - 把与当时后缀相同的前缀移来，前面一段保证匹配
  - 如果把最大的相同前缀移来，就可保证不遗漏可能的匹配
  - 若$p_0 \cdots p_{i-1}$中最大的相同前缀与后缀(不包括 $p_0 \cdots p_{i-1}$本身，但允许为空串)的长度为 k ($0 \le k \le i-1$). 当 $p_i \neq t_j$ 时, p 就应右移 i - k 位, 随后应比较 p_k 与 t_j . 即next[i] 应该为k.

next的计算

- 求 next 的问题现在变成：对每个 i，求出 p 的子串 $p_0 \cdots p_{i-1}$中最大的相同前缀与后缀的长度
- KMP 提出了一种巧妙的递推算法

$p_0 \cdots$	$p_{i-k} \cdots p_{i-1}$	p_i	$p_{i+1} \cdots$
	= =	?	
	$p_0 \cdots p_{k-1}$	p_k	$p_{k+1} \cdots$

前一步计算时最大相同的前缀长度为k

比较

检查长度为k+1的前后缀是否相等

next的计算

- 求 next 的问题现在变成：对每个 i，求出 p 的子串 $p_0 \cdots p_{i-1}$中最大的相同前缀与后缀的长度
- 初值：对于任何模式都有next[0] = -1

如果next[0]到next[i]都已经计算出来，如何计算next[i+1]的值?

- 假设next[i]=k，首先判断next[i+1]的值是否等于k+1

$$\begin{array}{cccccccc} p_0 & \cdots & p_{i-k} & \cdots & p_{i-1} & p_i & p_{i+1} & \cdots \\ & & = & & = & = & & \\ p_0 & \cdots & p_{k-1} & & p_k & p_{k+1} & \cdots & \end{array}$$

$p_i=p_k$，直接得到结果next[i+1]=k+1 (无回溯！)

next的计算

$p_0 \cdots p_{i-k} \cdots p_{i-1} \quad p_i \quad p_{i+1} \cdots$

$= \quad = \quad \neq$

$p_0 \cdots p_{k-1} \quad p_k \quad p_{k+1} \cdots$

$?$

$p_0 \cdots p_{next[k]} \cdots$

- 使用next[k]，无回溯的匹配方法，直接考虑已有比较短的前后缀。
- 跳到比较$p_{next[k]}$和p_i，检查长度为next[k+1]的前后缀是否相等（回到前面处理）。

递推计算next数组

- 利用next[0]= -1, …, next[i] 求next[i+1] 的算法：

  1）假设 next [i]=k，若$p_k = p_i$，则 $p_0 \cdots p_{i-k} \cdots p_i$ 中最大相同前后缀长度为next[i+1] = k+1。

  2）若$p_k \neq p_i$ 置 k 为next[k]，然后转到 1。（设 k = next[k]，就是考虑前一个更短的匹配前缀，从那里继续向下检查）

  3）若 k 值（来自next）为-1，就得到$p_0 \cdots p_{i-k} \cdots p_i$ 中最大相同前后缀的长度为k = 0。（即next [i+1] = 0）

计算next的算法

```
/*next是指向next数组的指针参数。*/
void makeNext (PSeqString p, int *next) {
    int i = 0, k = -1;
     next[0]= -1;                                   /* 初始化 */
    while (i < p->n-1) {                            /* 计算next[i+1] */
      while (k >= 0 && p->c[i] != p->c[k])
         k = next[k];
      i++; k++;
      next[i] = k;                                  /*？？*/
    }
}
```

对算法的进一步改进

- 当 $p_i \neq t_j$ 时,若$p_i = p_k$, 那么一定有 $p_k \neq t_j$.所以模式串应再向右移 k-next[k]位,下一步用 $p_{next[k]}$ 与t_j比较
- 对于next[i]=k 的改进：

  if (p_k== p_i) next[i] = next[k];

  else next[i]=k;
- 这一改进可以避免一些不必要的操作

计算next数组（改进后）

```
/* next是指向next数组的指针参数 */
 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];
          i++;k++;
          if(p->c[i] == p->c[k]) next[i] = next[k];
          else  next[i] = k;
      }
}
```

算法分析

- 关于算法复杂性，设模式串长m两重循环, 貌似 $O(m^2)$?
  - 外层循环每次将 i 加1,循环体总共执行m-1次
  - 外层循环的 k ＝ k+1 正好执行m-1次.k值从 -1 递增
  - 内层循环的 k ＝ next[k] 至少使 k 值减少1,但 k 值不可能小于 -1, 因此内层循环最多总共执行 m-1 次
  - 因此构造 next 表的代价是 O(m)
- KMP算法的一个重要优点是执行中不回溯。
  - 在处理从外部设备读入的庞大文件时，这种特性很有价值，因为可以一边读入一边匹配，不需要回头重读，因此不需要保存被匹配串
  - 做好next表之后，无回溯匹配时间复杂性是O(n)
- 如果需要多次使用一个模式串，相应的 next 数组只需建立一次（如在大文件里反复找一个单词）
- 这种情况下，可以考虑将 next 数组作为模式串的一个成分（另外定义一个模式串类型）

例子

- 考虑 t=″aabcbabcaabcaababc″
  p=″abcaababc″

下标	0	1	2	3	4	5	6	7	8
p	a	b	c	a	a	b	a	b	c
k	-1	0	0	0	1	1	2	1	2
p_k与p_i比较		≠	≠	=	≠	=	≠	=	=
next[i]	-1	0	0	-1	1	0	2	0	0

第一趟匹配

aabcbabcaabcaababc
abcaababc

第二趟匹配

aabcbabcaabcaababc
abcaababc

第三趟匹配

aabcbabcaabcaababc
abcaababc

第四趟匹配

aabcbabcaabcaababc
abcaababc

小结

- 字符串是由字符作元素组成的线性表。但是作为一种抽象数据类型，有它自己的操作，在对串处理时，要抓住它的特殊性。
- 模式匹配是子串在主串中的定位操作，是一个比较常用的操作。朴素的模式匹配算法比较直观，易于理解；但是效率比较低。
- 无回溯的模式匹配算法的技巧性很强，实现无回溯的模式匹配的基础是依靠next表支持。计算next的算法实质上还是一个无回溯的模式匹配算法。