第4章 串

4.1 串的基本概念

4.2 串的存储结构

4.3 串的模式匹配

4.1 串的基本概念

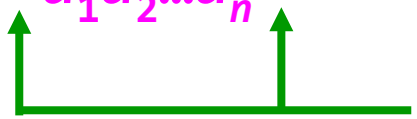
串（或字符串）是由零个或多个字符组成的有限序列。

串 ⊂ 线性表

串中所含字符的个数称为该串的长度（或串长），含零个字符的串称为空串，用Φ表示。

串的逻辑表示，a_i（$1 \le i \le n$）代表一个字符：

“$a_1a_2\ldots a_n$”



双引号不是串的内容，起标识作用

串相等：当且仅当两个串的长度相等并且各个对应位置上的字符都相同时，这两个串才是相等的。

如：

“abcd” ≠ “abc”

“abcd” ≠ “abcde”

所有空串是相等的。

子串：一个串中任意个连续字符组成的子序列（含空串）称为该串的子串。

例如，“*abcde*”的子串有：

“”、“*a*”、“*ab*”、“*abc*”、“*abcd*”和“*abcde*”等

真子串是指不包含自身的所有子串。

串抽象数据类型＝逻辑结构＋基本运算（运算描述）

串的基本运算如下：

❶ StrAssign(&s, cstr)：将字符串常量cstr赋给串s，即生成其值等于cstr的串s。

❷ StrCopy(&s, t)：串复制。将串t赋给串s。

❸ StrEqual(s, t)：判串相等。若两个串s与t相等则返回真；否则返回假。

❹ StrLength(s)：求串长。返回串s中字符个数。

❺ Concat(s, t)：串连接:返回由两个串s和t连接在一起形成的新串。

❻ SubStr(s, i, j)：求子串。返回串s中从第i（$1 \le i \le n$）个字符开始的、由连续j个字符组成的子串。

❼ InsStr(s1, i, s2)：插入。将串s2插入到串s1的第i（$1 \le i \le n+1$）个字符中，即将s2的第一个字符作为s1的第i个字符，并返回产生的新串。

❽ DelStr(s, i, j)：删除。从串s中删去从第i（$1 \le i \le n$）个字符开始的长度为j的子串，并返回产生的新串。

❾ RepStr(s, i, j, t)：替换。在串s中，将第i（$1 \le i \le n$）个字符开始的j个字符构成的子串用串t替换，并返回产生的新串。

❿ DispStr(s)：串输出。输出串s的所有元素值。

4.2 串的存储结构

串中元素逻辑关系与线性表的相同，串可以采用与线性表相同的存储结构。

4.2.1 串的顺序存储

【例】设计顺序串上实现串比较运算Strcmp(s, t)的算法。如：

"ab" < "abcd"　　"abcd" < "abd"

解：算法思路如下：

（1）比较s和t两个串共同长度范围内的对应字符：

① 若s的字符＞t的字符，返回1；

② 若s的字符＜t的字符，返回-1；

③ 若s的字符=t的字符，按上述规则继续比较。

（2）当（1）中对应字符均相同时，比较s和t的长度：

① 两者相等时，返回0；

② s的长度＞t的长度，返回1；

③ s的长度＜t的长度，返回-1。

```
int  Strcmp(SqString s, SqString t)
{  int i, comlen;
   if (s.length<t.length)  comlen=s.length; //求s和t的共同长度
   else comlen=t.length;
   for (i=0;i<comlen;i++)                   //在共同长度内逐个字符比较
      if (s.data[i]>t.data[i])
         return 1;
      else if (s.data[i]<t.data[i])
         return -1;

      if (s.length==t.length)               //s==t
         return 0;
      else if (s.length>t.length)           //s>t
         return 1;
      else  return -1;                      //s<t

}
```

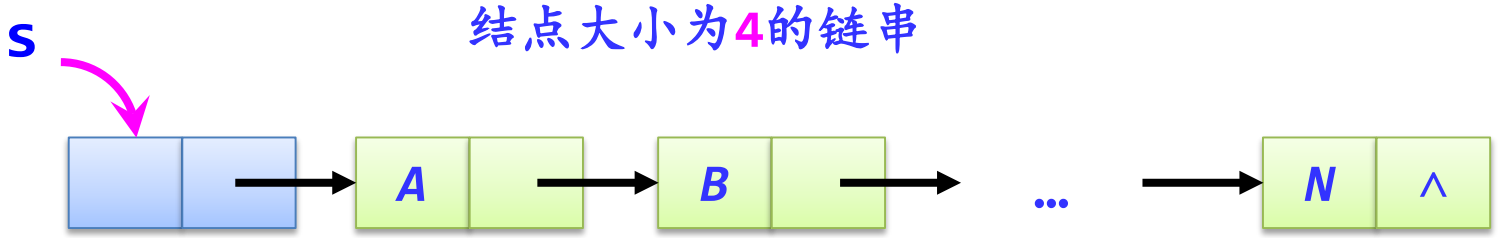
所有共同长度内的字符相同，哪个长哪个大

4.2.2 串的链式存储

链串的组织形式与一般的链表类似。

链串中的一个结点可以存储多个字符。通常将链串中每个结点所存储的字符个数称为结点大小。

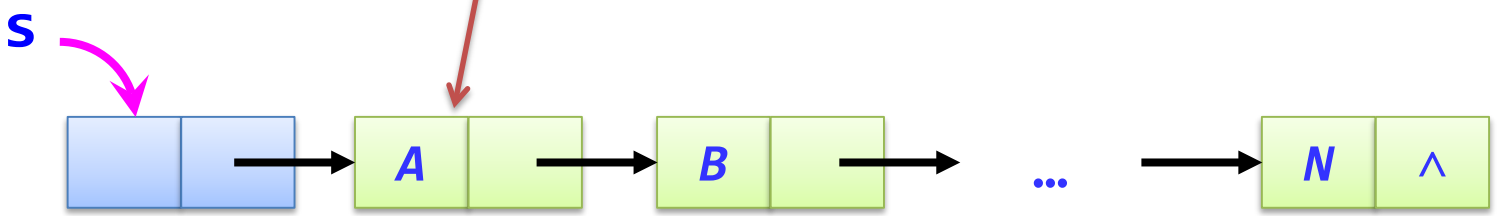
结点大小为4的链串



结点大小为1的链串

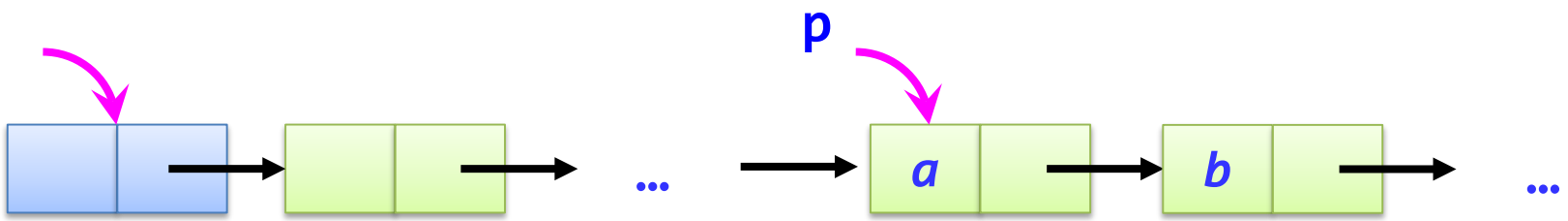
链串结点大小1时，链串的结点类型声明如下：

```
typedef struct snode
{    char data;
     struct snode *next;
}  LinkStrNode;
```

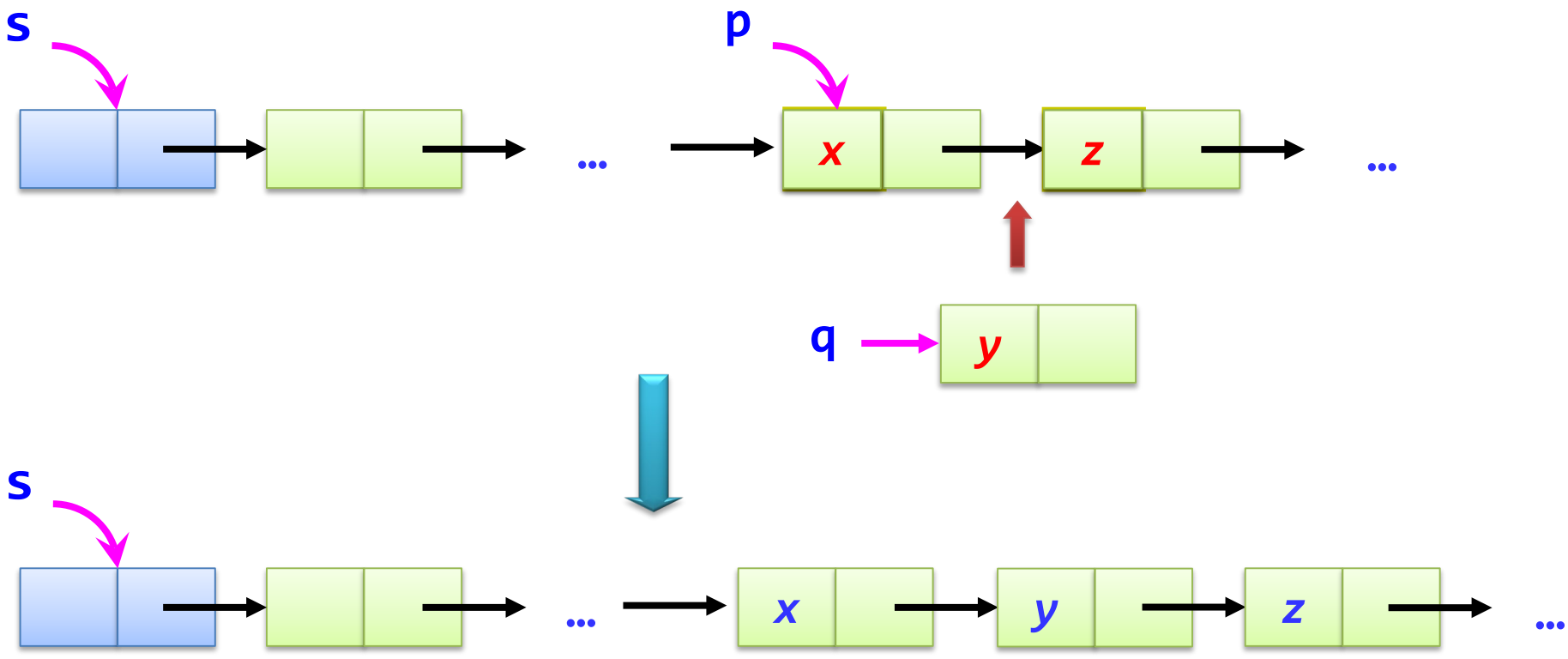


【例】在链串中，设计一个算法把最先出现的子串“*ab*”改为“*xyz*”。

❶ **查找**：p->data='a' && p->next->data='b'



❷ 替换

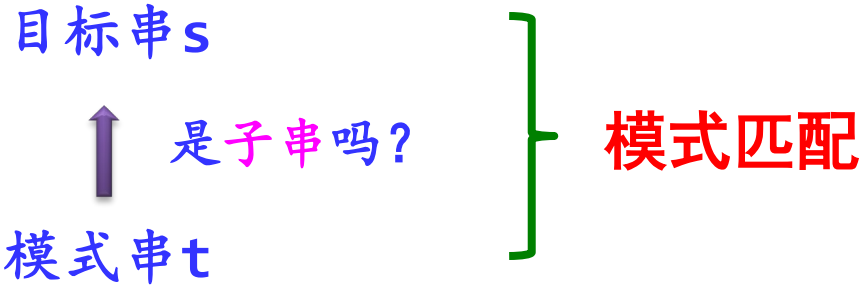


```
void Repl(LinkStrNode *&s)
{
    LinkStrNode *p=s->next, *q;
    int find=0;
    while (p->next!=NULL && find==0)          //查找ab子串
    {
        if (p->data==‘ a’ && p->next->data==‘b’)
        {  p->data=‘x’; p->next->data=‘z’;
           q=(LinkStrNode *)malloc(sizeof(LinkStrNode));
           q->data=‘y’;  q->next=p->next;  p->next=q;
           find=1;
         }
        else p=p->next;
     }
}
```

替换为xyz

算法的时间复杂度为$O(n)$。

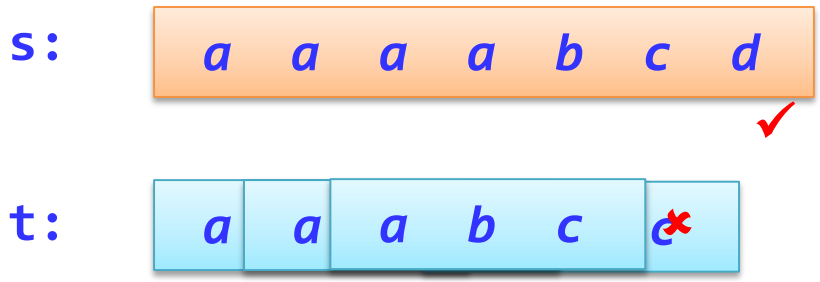
4.3 串的模式匹配



- 成功是指在目标串s中找到一个模式串t —— t是s的子串，返回t在s中的位置。
- 不成功则指目标串s中不存在模式串t —— t不是s的子串，返回-1。

4.4.1 Brute-Force算法

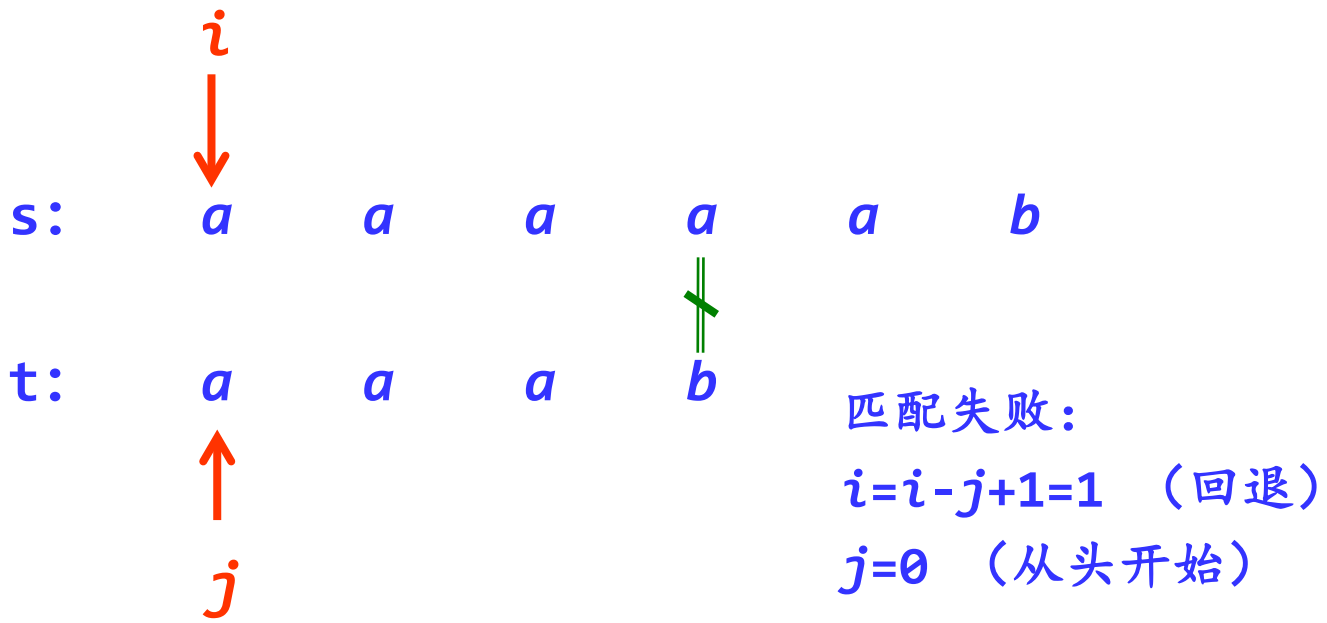
Brute-Force简称为BF算法，亦称简单匹配算法。采用穷举的思路。BF是指暴力的意思！



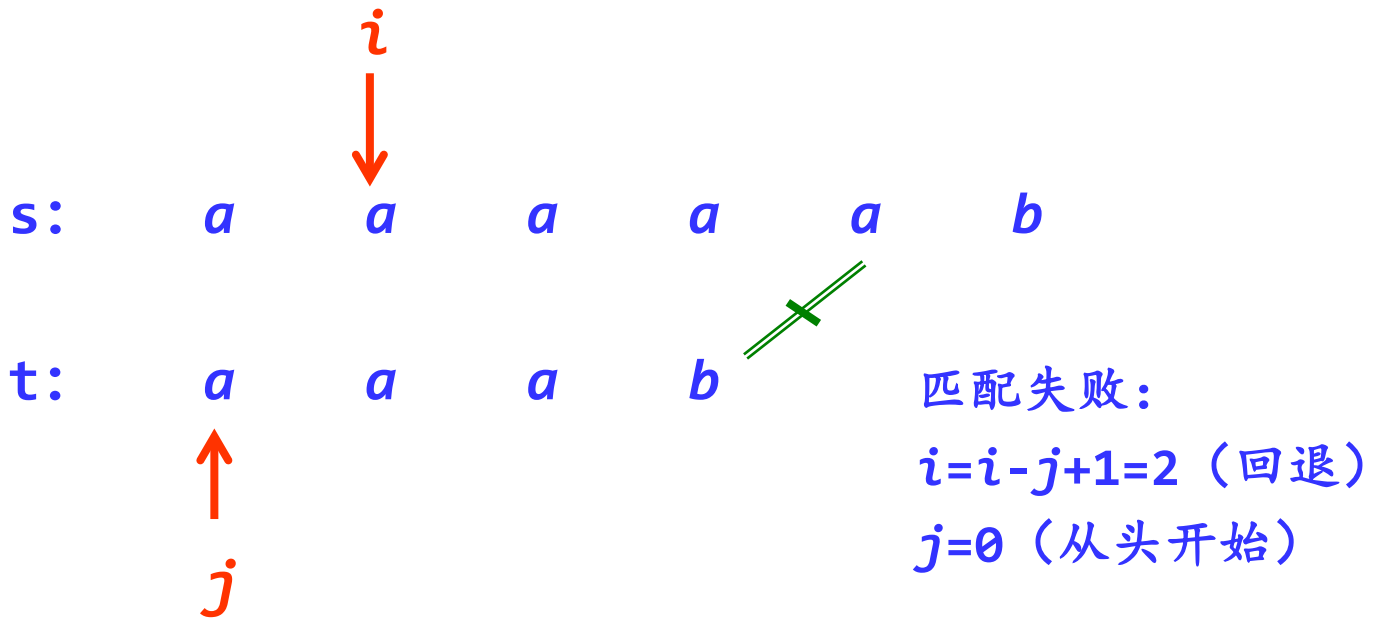
匹配成功

算法的思路是从s的每一个字符开始依次与t的字符进行匹配。

例如，设目标串s="aaaaab"，模式串t="aaab"。s的长度为n（n=6），t的长度为m（m=4）。BF算法的匹配过程如下。



$i=1,\ j=0$

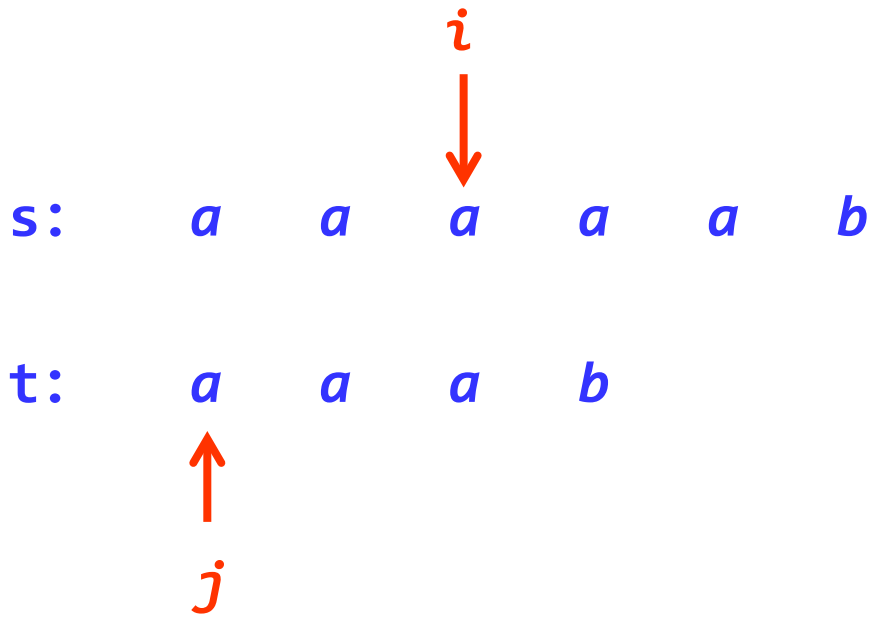


匹配失败：

$i=i-j+1=2$（回退）

$j=0$（从头开始）

i=2，j=0



匹配成功：

i=6，j=4

返回i-t.length=2

对应的BF算法如下：

```
int index(SqString s, SqString t)
{  int i=0, j=0;
   while (i<s.length && j<t.length)
   {  if (s.data[i]==t.data[j])      //继续匹配下一个字符
      {  i++;                        //主串和子串依次匹配下一个字符
         j++;
      }
      else                           //主串、子串指针回溯重新开始下一次匹配
      {  i=i-j+1;                    //主串从下一个位置开始匹配
         j=0;                        //子串从头开始匹配
      }
   }
   if (j>=t.length)
      return(i-t.length);            //返回匹配的第一个字符的下标
   else
      return(-1);                    //模式匹配不成功
}
```

BF算法分析：

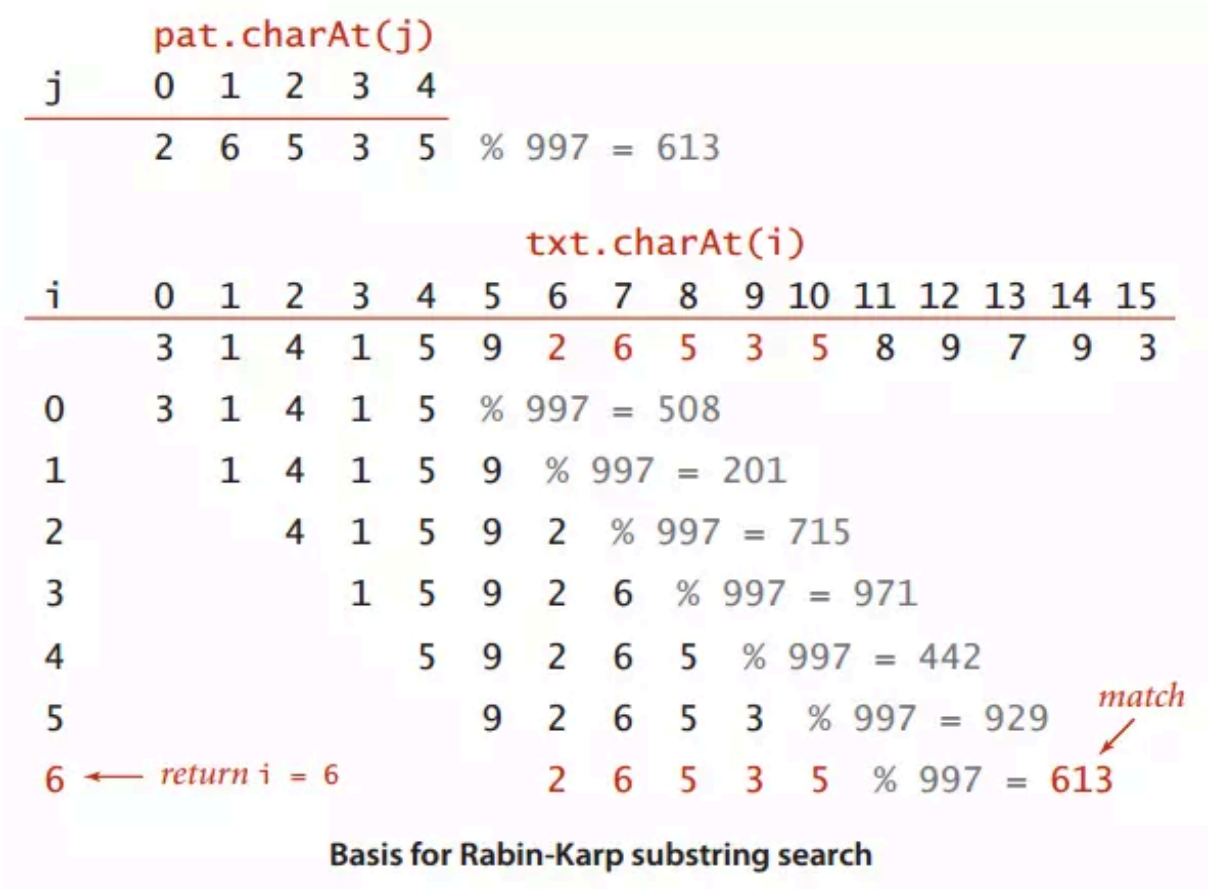
- 算法在字符比较不相等，需要回溯（即i=i-j+1）：即退到s中的下一个字符开始进行继续匹配。
- 最好情况下的时间复杂度为O(*m*)。
- 最坏情况下的时间复杂度为O(*n*×*m*)。
- 平均的时间复杂度为O(*n*×*m*)。

Rabin-Karp 算法

- 由 M.O. Rabin 和 R.A. Karp 发明
- RK 算法对“滑动窗口内容”逐一匹配
- 算法思想:
  - 将滑动窗口内 m 个字符的比较变为一个(哈希)值的比较
  - 通过对字符串进行(哈希)运算，然后比较子串哈希值与滑动窗口内子串的(哈希)值
  - 仅当这两个(哈希)值相等时再来比较窗口内的子串是否相等

Rabin-Karp 算法

- 从简单例子开始
  - 目标串为“3141592653589793”，模式串为“26535”



Basis for Rabin-Karp substring search

Rabin-Karp 算法

- 例子的算法执行过程:
  - 假设字符集全是由0到9的数字组成 $\sum = \{0,1,2\cdots9\}$
  - 给定长度为n的目标字符串 T[1…n]
  - 给定长度为m的模式字符串P[1…m]
  - 将P[1…m]映射为数值p；将T[s+1, … , s+m]映射为数值ts, 只有p与ts相等时候，才有可能发生匹配

    p = P[m] + 10(P[m-1] + 10(P[m-2]+ …. + (10(P[2]) + P[1]))

  - 滚动计算ts:

    $ts+1 = 10* (ts - 10^{m-1} * T[s+1]) + T[s+m+1],\ 0<=s <=n-m$

Rabin-Karp 算法

- 算法的一般执行过程
  - 使用 h(x) = x mod q 作为哈希函数
  - 滑动窗口的哈希值计算公式

    $hash(w[0..m-1])=(w[0]\times 2^{m-1}+ w[1]\times 2^{m-2}+\cdots+ w[m-1]\times 2^{0}) \bmod q$

    其中q是一个素数
  - 当滑动窗口右移时，需要对新窗口的子串重新利用哈希函数计算

    $rehash(a,b,h) = ((h-a\times 2^{m-1})\times 2 + b) \bmod q$

    其中h为上一个滑动窗口的hash值，b为新加入的字符

Rabin-Karp 算法

- 第一次匹配 h(P)=67*2^7+71*2^6+84*2^5+67*2^4+84*2^3+67+2^2+84*2^1+67 = 17858

C G T A G C G T C T C T C A T A T G T C A T G C

C G T C T C T C

- 第二次匹配

C G T A G C G T C T C T C A T A T G T C A T G C

C G T C T C T C

- 第三次匹配

C G T A G C G T C T C T C A T A T G T C A T G C

C G T C T C T C

- 第六次匹配

C G T A G C G T C T C T C A T A T G T C A T G C

C G T C T C T C

Rabin-Karp 算法

- 算法复杂度分析
  - 预处理时间 O(m)
  - 匹配时间最好是 O(m)；最坏是 O((n-m+1)m), 即 O(mn)
  - 算法的期望匹配时间计算比较复杂(涉及到哈希冲突)，但一般认为是O(m+n)

——本章完——