第4章 串、数组和广义表

- 第2章　线性表
- 第3章　栈和队列
- 第4章　串、数组和广义表

线性结构

可表示为：　$(a_1, a_2, \ldots\ldots, a_n)$

2020年4月26日

补充：C语言中常用的串运算

调用标准库函数 #include<string.h>

串比较，strcmp(char s1,char s2)
串复制，strcpy(char to,char from)
串连接，strcat(char to,char from)
求串长，strlen(char s)
……

2020年4月26日

第4章　串、数组和广义表

教学内容

4.1　串

4.2　数组

4.3　广义表

2020年4月26日

教学目标

1. 了解串的存储方法，理解串的两种模式匹配算法，重点掌握BF算法。
2. 明确数组和广义表这两种数据结构的特点，掌握数组地址计算方法，了解几种特殊矩阵的压缩存储方法。
3. 掌握广义表的定义、性质及其GetHead和GetTail的操作。

2020年4月26日

4.1 串的定义

串(String)----零个或多个字符组成的有限序列

$$s = 'a_1 a_2 \cdots a_n'$$

串名

串值

串长 → n

空串 ← n=0

2020年4月26日

a=‘BEI’,
b=‘JING’
c=‘BEIJING’
d=‘BEI JING’

Substring(c,4,3)=‘JIN’

子串

主串

字符位置

子串位置

串相等

空格串

2020年4月26日

4.2 案例引入

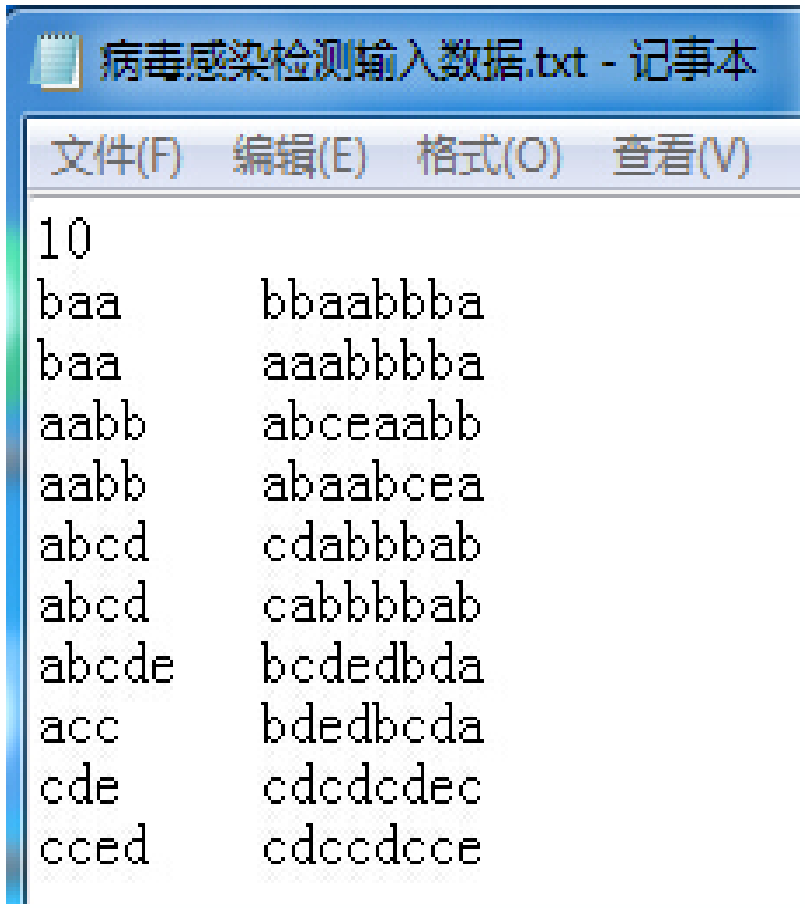
案例4.1： 病毒感染检测

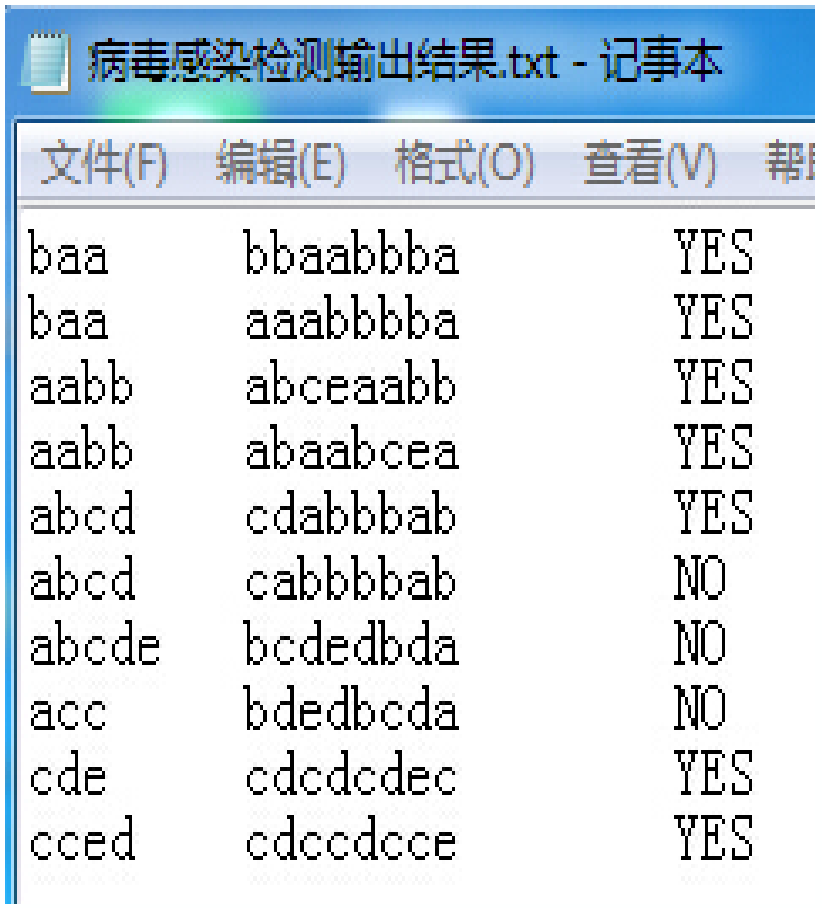
研究者将人的DNA和病毒DNA均表示成由一些字母组成的字符串序列。

然后检测某种病毒DNA序列是否在患者的DNA序列中出现过，如果出现过，则此人感染了该病毒，否则没有感染。

例如，假设病毒的DNA序列为baa，患者1的DNA序列为aaabbba，则感染，患者2的DNA序列为babbba，则未感染。

（注意，人的DNA序列是线性的，而病毒的DNA序列是环状的)


病毒感染检测输入数据.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V)
10
baa bbaabbba
baa aaabbbba
aabb abceaabb
aabb abaabcea
abcd cdabbbab
abcd cabbbbab
abcde bcdedbda
acc bdedbcda
cde cdcdcdec
cced cdccdcce


病毒感染检测输出结果.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V)
baa bbaabbba YES
baa aaabbbba YES
aabb abceaabb YES
aabb abaabcea YES
abcd cdabbbab YES
abcd cabbbbab NO
abcde bcdedbda NO
acc bdedbcda NO
cde cdcdcdec YES
cced cdccdcce YES

4.3 串的类型定义、存储结构及运算

ADT String {

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

数据关系: $R_1=\{<a_{i-1},a_i> \mid a_{i-1},a_i \in D, i=1,2,\cdots,n\}$

基本操作:

(1) StrAssign (&T,chars) //串赋值

(2) StrCompare (S,T) //串比较

(3) StrLength (S) //求串长

(4) Concat(&T,S1,S2) //串联

(5) SubString(&Sub,S,pos,len)　　//求子串

(6) StrCopy(&T,S)　　//串拷贝

(7) StrEmpty(S)　　//串判空

(8) ClearString (&S)　　//清空串

(9)　Index(S,T,pos)　　//子串的位置

(11) Replace(&S,T,V)　　//串替换

(12) StrInsert(&S,pos,T)　　//子串插入

(12) StrDelete(&S,pos,len)　　//子串删除

(13) DestroyString(&S)　　//串销毁

}ADT String

串的存储结构

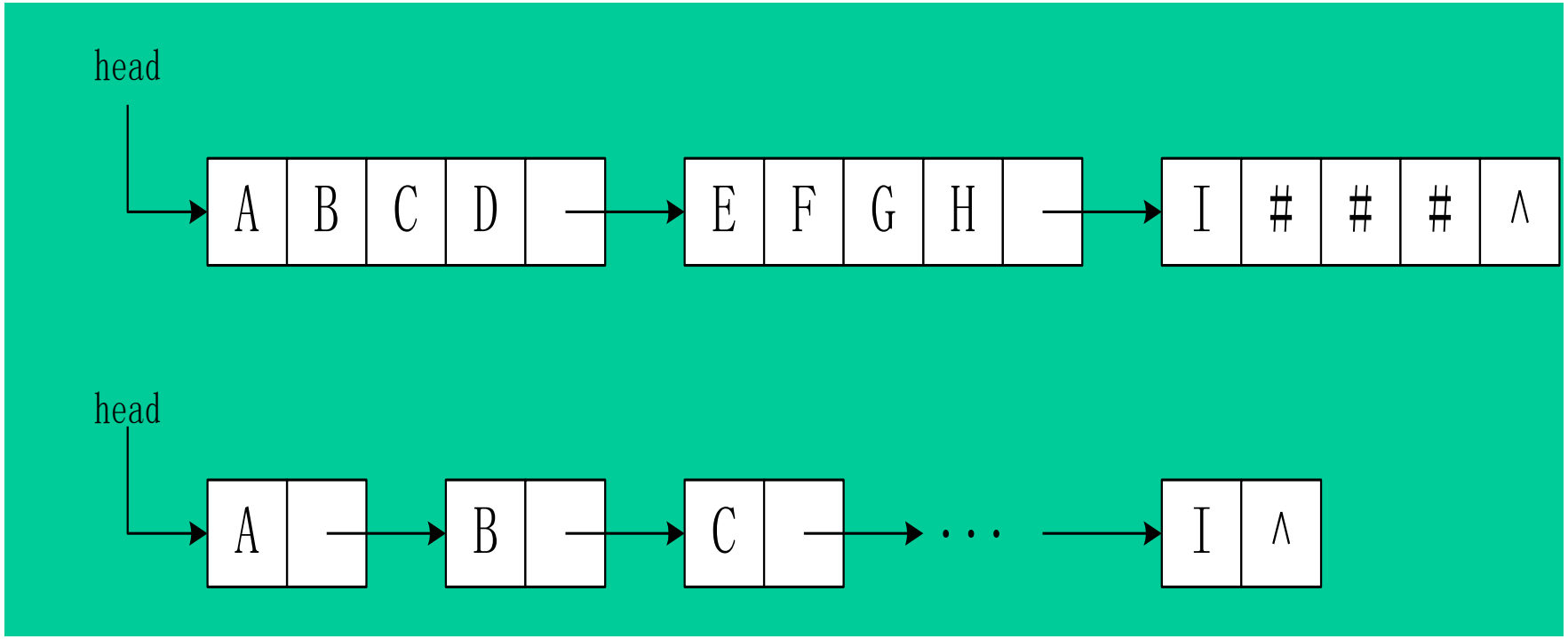
- 顺序存储
- 链式存储

2020年4月26日

顺序存储表示

```
typedef struct{
    char *ch;        //若串非空,则按串长分配存储区,
                     //否则ch为NULL
    int  length;     //串长度
}HString;
```

链式存储表示



链式存储表示

```
#define CHUNKSIZE 80       //可由用户定义的块大小
typedef struct Chunk{
   char  ch[CHUNKSIZE];
   struct Chunk *next;
}Chunk;

typedef struct{
   Chunk *head,*tail;      //串的头指针和尾指针
   int curlen;             //串的当前长度
}LString;
```

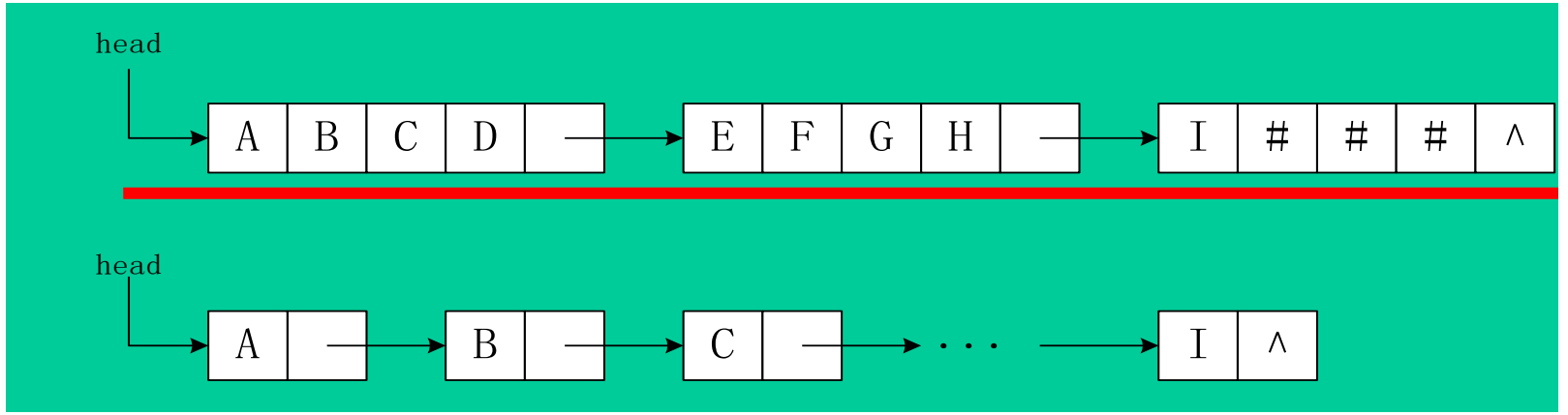
2020年4月26日

链式存储表示

优点：操作方便
缺点：存储密度较低

$$\text{存储密度} = \frac{\text{串值所占的存储位}}{\text{实际分配的存储位}}$$

可将多个字符存放在一个结点中，以克服其缺点



2020年4月26日

串的模式匹配算法

算法目的：

确定主串中所含子串第一次出现的位置（定位）

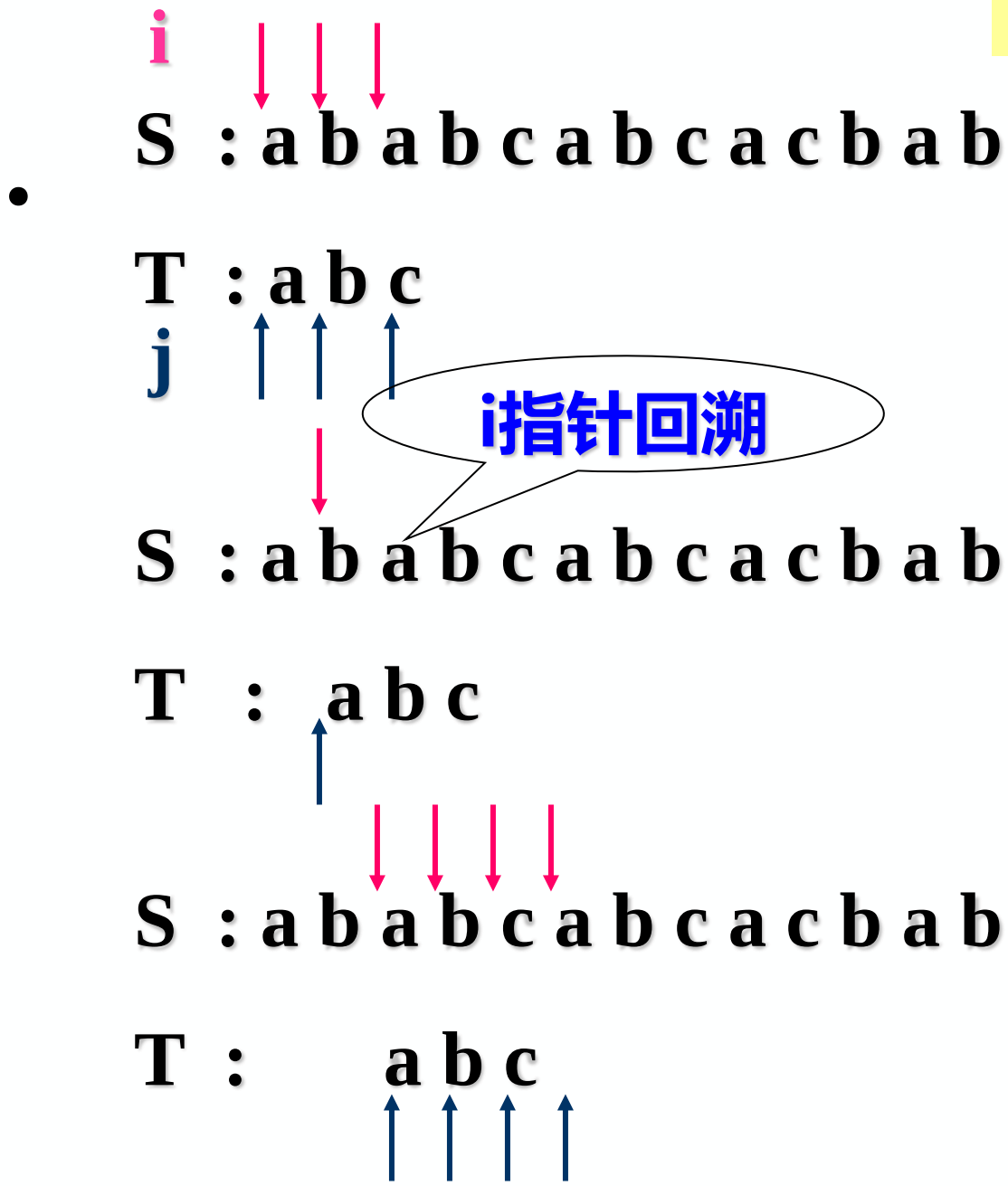
即如何实现教材P72 Index(S,T,pos)函数

算法种类：

- BF算法（又称古典的、经典的、朴素的、穷举的）
- KMP算法（特点：速度快）

2020年4月26日

BF算法设计思想


i
S : a b a b c a b c a c b a b
T : a b c
j
i指针回溯
S : a b a b c a b c a c b a b
T : a b c
S : a b a b c a b c a c b a b
T : a b c

BF算法设计思想

Index(S,T,pos)

- 将主串的第pos个字符和模式的第一个字符比较，
  若**相等**，继续逐个比较后续字符；
  若**不等**，从主串的下一字符起，重新与模式的第一个字符比较。

- 直到主串的一个连续子串字符序列与模式相等 。
  返回值为S中与T匹配的子序列**第一个字符的序号**，
  即匹配成功。
- 否则，匹配失败，返回值 0

BF算法描述（算法4.1）

```
int  Index(SString S,SString T,int pos){
    i=pos;   j=1;
   while (i<=S.length && j <=T.length){
      if ( S[ i ]==T[ j ]) {++i;  ++j; }
     else{ i=i-j+2;    j=1; }
   if ( j> T.length)   return   i－ T.length;
   else return 0;
}
```