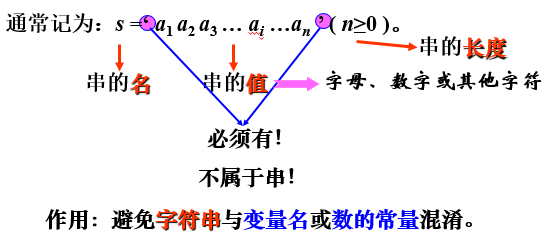
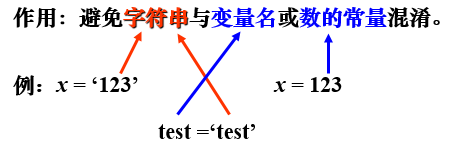
# 串的概念、ADT定义

## 串的概念

**串（字符串）**：是**由 0 个或多个字符组成的有限序列**。





**空串：**不含任何字符的串，长度 = 0，用符号 *φ* 表示。

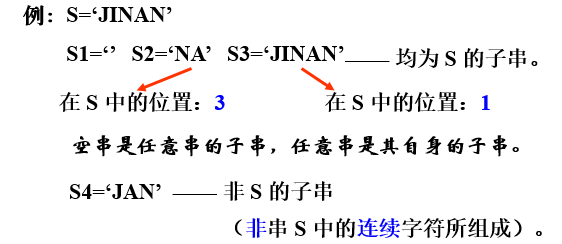
**空格串：**仅由一个或**多个**空格组成的串。

**子串：**由串中**任意个连续**的字符组成的子序列。**（子串可以是空串和它本身）**

**主串：**包含子串的串。

**位置：**字符在序列中的序号。**（位置的序号是从1开始数的）**

**子串在主串中的位置：**子串的首字符在主串中的位置。



**串相等的条件：**当两个串的**长度**相等且**各个对应位置的字符都相等**时才相等。

**例：** S=‘JINAN’ S1=‘JI NAN’ **S≠S1**

**串的逻辑结构：**和线性表极为相似。 **区别：**串的数据对象约定是**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：大多以“**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。

## 串的抽象数据类型的定义

**ADT String {**

**数据对象：**D＝{ *ai* |*ai*∈CharacterSet, *i* = 1, 2, ..., *n*, *n*≥0 }

**数据关系：**R1＝{ < *ai*-1, *ai* > | *ai*-1, *ai* ∈D, *i* = 2, ..., *n* }

**基本操作：**

**StrAssign (&T, chars)**

初始条件：chars 是字符串常量。

操作结果：把 chars 赋为 T 的值。

**StrCopy (&T, S)**

初始条件：串 S 存在。

操作结果：由串 S 复制得串 T。

**DestroyString (&S)**

初始条件：串 S 存在。

操作结果：串 S 被销毁。

**StrEmpty (S)**

初始条件：串 S 存在。

操作结果：若 S 为空串，则返回 TRUE，否则返回 FALSE。

**StrCompare (S, T)**

初始条件：串 S 和 T 存在。

操作结果：若 S > T，则返回值 > 0；

若 S = T，则返回值 = 0；

若 S < T，则返回值 < 0。

**说明：“串值大小” 是按 “词典次序” 进行比较的，如：**StrCompare(“**d**ata”, “**s**tru”)<0 （由于对于第一个字符来说d<s）  
StrCompare(“ca**t**”, “ca**s**e”)>0 （对于第三个字符来说s>t）

**StrLength (S)**

初始条件：串 S 存在。

操作结果：返回 S 的元素个数，称为串的长度。

**Concat (&T, S1, S2)**

初始条件：串 S1 和 S2 存在。

操作结果：用 T 返回由 S1 和 S2 **联接**而成的新串。

**SubString (&Sub, S, pos, len)**

初始条件：串 S 存在，1≤pos≤StrLength(S)

且 0≤len≤**StrLength(S) – pos + 1**。

操作结果：用 Sub 返回串 S 的第 pos 个字符起长度为 len 的子串。

**Index (S, T, pos)**

初始条件：串 S 和 T 存在，T 是非空串，1≤pos≤StrLength(S)。

操作结果：若主串 S 中存在和串 T 值相同的子串，

则**返回它在主串 S 中第 pos 个字符之后第一次出现的位置**；

否则函数值为 0。

**Replace (&S, T, V)**

初始条件：串 S、T 和 V 存在，T 是非空串。

操作结果：用 V 替换主串 S 中出现的所有与 T 相等的**不重叠**的子串。

**假设：**

**S=“abcacabcaca”， T=“abca”**  **V=“ab”，**

**则置换之后的 S=“abcabca”， 而不是 “abbcaca”。（不重叠的子串）**

**StrInsert (&S, pos, T)** 初始条件：串 S 和 T 存在，1≤pos≤**StrLength(S)+1**。   
 操作结果：在串 S 的第 pos 个字符之前插入串 T。

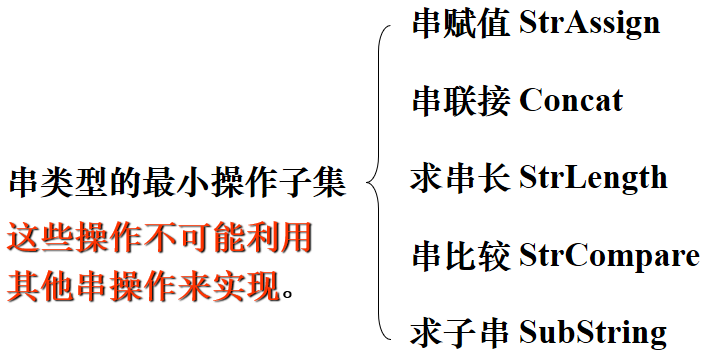
**StrDelete (&S, pos, len)** 初始条件：串 S 存在，1≤pos≤**StrLength(S)-len+1**。   
 操作结果：从串 S 中删除第 pos 个字符起长度为 len 的子串。

**ClearString (&S)**

初始条件：串 S 存在。

操作结果：将 S 清为空串。

**} ADT String**



例如，可利用**求串长、求子串和串比较**等操作 实现定位函数 **Index(S, T, pos) 。**

int Index (String S, String T, int pos)

{

if (pos > 0)

{ n = StrLength(S); m = StrLength(T); // 求串长

　　　i = pos;

　　while ( i <= n-m+1)

{ SubString (sub, S, i, m);

　　　　if (StrCompare(sub,T) != 0) ++i ;

　　　　else return i ;　 // 找到和 T 相等的子串

　　　} // while

　} // if

　 return 0;　 　　　　// S 中不存在满足条件的子串

} // Index

# 串的定长表示、基本操作实现

因为串是特殊的线性表，故其存储结构与线性表的存储结构类似，只不过组成串的结点是**单个字符**。

## 定长顺序存储表示

定长顺序存储表示，也称为**静态存储分配的顺序串**。即用一组地址连续的存储单元依次存放串中的字符序列。

**“定长”、“静态”**的意思可简单地理解为一个**确定的存储空间**，它的**长度是不变。**

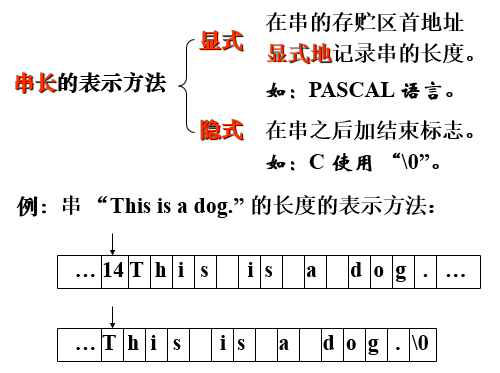
可直接使用定长的字符数组来定义一个串，数组的上界预先给出：

#define maxstrlen 255 // 可在 255 以内定义最大串长。

typedef unsigned char sstring[maxstrlen+1]; // 0 号单元存放串的长度。

sstring a; // sstring代表的是一个一维数组类型，里面存放的类型是unsigned char

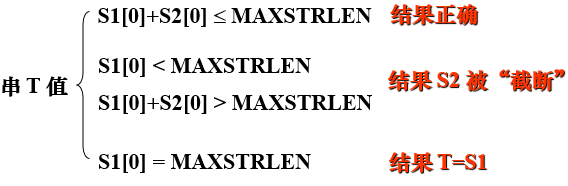
串的实际长度可在这个预定义长度的范围内随意设定，超过预定义长度的串值则被舍去，称之为“截断”。



## 定长顺序存储表示时串的操作的实现

**1、串联接 Concat(&T, S1, S2)**

假设串 T 是由串 S1 联结串 S2 得到的，则只要进行相应的“**串值复制**”操作即可，需要时进行“截断”。



Status Concat(SString &T, SString S1, SString S2) {

if (S1[0]+S2[0] <= MAXSTRLEN) // 未截断

{ T[1...S1[0]] = S1[1...S1[0]];

T[S1[0]+1...S1[0]+S2[0]] = S2[1...S2[0]];

T[0] = S1[0]+S2[0]; uncut = TRUE; }

else

if (S1[0] < MAXSTRSIZE) // 截断

{ T[1...S1[0]] = S1[1...S1[0]];

T[S1[0]+1...MAXSTRLEN] = S2[1...MAXSTRLEN－S1[0]];

T[0] = MAXSTRLEN; uncut = FALSE; }

else // 截断(仅取S1)

{ T[0...MAXSTRLEN] = S1[0...MAXSTRLEN]; T[0] = MAXSTRLEN; uncut = FALSE; }

return uncut;

} // Concat

**2、求子串 SubString(&Sub, S, pos, len)**

**求子串的过程即为复制字符序列的过程，将串 S 中的第 pos 个字符开始的长度为 len 的字符串复制到串 Sub 中。**

**注：1)、不会出现“截断”的情况。**

**2)、可能出现“参数非法”的情况，应返回 ERROR。**

Status SubString(SString &Sub, SString S, int pos, int len)

{ if (pos < 1 || pos > S[0] || len < 0 || len > S[0]-pos+1)

return ERROR;

Sub[1…len] = S[pos…pos+len-1];

Sub[0]=len;

return OK;

} // SubString

**3、求置换Replace (&S, T, V)**

定长顺序存储表示时串操作的缺点 ：

1、需事先预定义串的最大长度，这在程序运行前是很难估计的。

2、由于定义了串的最大长度，使得串的某些操作受限（截尾），如串的联接、插入、置换等运算。

克服办法：不限定最大长度 ——动态分配串值的存储空间。

#include<stdio.h>

#include<string.h>

#include<stdlib.h>

#define MAXSTRLEN 255

typedef unsigned char sstring[MAXSTRLEN+1];

/\* 字符串赋值,当赋值字符串长度超过被赋值字符串时候截断,未超过时先将长度存入T[0],再逐位赋值 \*/

Status StrAssign(sstring T, char \*chars)

{

int i;

if (strlen(chars) > MAXSTRLEN)

{

for (i = 1; i <= MAXSTRLEN; ++i)

T[i] = chars[i-1];

T[0] = MAXSTRLEN;

}

else

{

T[0] = strlen(chars); //p.s.此时T[0]存入的是int类型的数据,打印%s时无法显示

for (i = 1; i <= T[0]; ++i)

T[i] = chars[i-1];

return OK;

}

}

/\* 字符串拷贝,逐个字符拷贝(仅拷贝长度范围内的) \*/

Status StrCopy(sstring T, sstring S)

{

int i;

//p.s.此处要拷贝长度+1,将S1的\0同时拷贝进去,否则T中没有\0

for (i = 1; i <= S[0]; ++i)

T[i] = S[i];

T[0] = S[0];

return OK;

}

/\* 字符串判空,当S[0]==0时为空 \*/

Status StrEmpty(sstring S)

{

if (S[0] == 0)

{

printf("String is Empty\n");

return TRUE;

}

else

{

printf("String is not Empty\n");

return FALSE;

}

}

/\* 返回子串长度 \*/

Status StrLength (sstring S)

{

return S[0];

}

/\* 打印字符串 \*/

Status StrPrint(sstring T)

{

int i;

for (i = 1; i <= T[0]; ++i)

{

printf("%c", T[i]);

}

printf("\n");

return OK;

}

/\* 清空字符串 \*/

Status ClearString(sstring S)

{

S[0] = 0;

return OK;

}

/\* 字符串联接,通过T来存储结果,S2连接而成的新串,若未截断则返回TRUE,截断返回FAlSE,注意字符串数组越界问题 \*/

Status Concat(sstring T, sstring S1, sstring S2)

{

int i;

if (S1[0] + S2[0] <= MAXSTRLEN)

{

for (i = 1; i <= S1[0]; ++i)

T[i] = S1[i];

for (i = 1; i <= S2[0]; ++i)

T[i + S1[0]] = S2[i];

T[0] = S1[0] + S2[0];

return TRUE;

}

else //先存完S1中的,再存S2中的内容,条件是不能越界

{

for (i = 1; i <= S1[0] && i <= MAXSTRLEN - 1; ++i)

T[i] = S1[i];

for (i = 1; i <= MAXSTRLEN - S1[0]; ++i)

T[i + S1[0]] = S2[i];

T[0] = MAXSTRLEN;

return FALSE;

}

}

/\* 索引子串 \*/

Status Index(sstring T, sstring S)

{

int i = 1, j = 1;

while(j <= T[0] && i <= S[0])

{

if (T[j] == S[i])

{

++i;

++j;

}

else

{

i = i - j + 2;

j = 1;

}

}

if (j > T[0])

{

printf("Position:%d\n", i - j + 1);

return OK;

}

else

{

printf("substring is not exist\n");

return ERROR;

}

}

/\* 在位置pos处插入子串T \*/

Status StrInsert(sstring T, sstring S, int pos)

{

int i;

//判断pos的合法性

if (S[0] + T[0] <= MAXSTRLEN)

{

for (i = S[0] + T[0]; i >= pos + T[0]; --i)

S[i] = S[i - T[0]];

for (i = 1; i <= T[0]; ++i)

S[pos + i - 1] = T[i];

S[0] = S[0] + T[0];

return OK;

}

else

{

for (i = MAXSTRLEN; i >= pos + T[0]; --i)

S[i] = S[i - T[0]];

for (i = 1; i <= T[0]; ++i)

S[pos + i - 1] = T[i];

S[0] = MAXSTRLEN;

return ERROR;

}

}

/\*从串 S 中删除第 pos 个字符起长度为 len 的子串。\*/

Status StrDelete(sstring S, int pos, int len)

{

int i;

if (pos > S[0])

return ERROR;

else if (pos + len > S[0])

len = S[0] - pos + 1;

S[0] -= len;

}

/\* 字符串比较。比较字符串是否完全相同,如果不同则判断子串所在的位置 \*/

Status StrCompare(sstring S, sstring T)

{

int i, j;

for (i = 1; i <= T[0]&&i<=S[0]; ++i)

{

if(S[i]!=T[i])

return S[i]-T[i]

}

return S[0]-T[0];

}

Status SubString(sstring &Sub, sstring S, int pos, int len)

{ if (pos < 1 || pos > S[0] || len < 0 || len > S[0]-pos+1)

return ERROR;

Sub[1…len] = S[pos…pos+len-1];

Sub[0]=len;

return OK;

} // SubString

# 串的堆分配表示、基本操作实现

## 堆分配存储表示

**堆存储结构的特点：**仍以一组空间足够大的、地址连续的存储单元依次存放串值字符序列，但它们的存储空间是在程序执行过程中**动态分配**的。

通常，C 语言中提供的串类型就是以这种存储方式实现的。由动态分配函数 malloc() 分配一块实际串长所需要的存储空间（“堆”），如果分配成功，则返回此空间的起始地址，作为串的基址。由 free( ) 释放串不再需要的空间。

**用堆存放字符串时，其结构用 C 语言定义如下：**

typedef struct {

char \*ch; // 若非空则按串长分配存储区，

// 否则 ch 为 NULL

int length; //串长度

} HString;

这类串操作的实现算法为：

1、为新生成的串分配一个存储空间；

2、进行串值的复制。

## 堆分配基本操作实现

#include<stdio.h>

#include<stdlib.h>

typedef struct {

char \*ch; // 若非空则按串长分配存储区， 否则 ch 为 NULL

int length; //串长度

} HString;

//初始化字符串

Status InitString(HString \*T)

{

//指针指向NULL,长度为0即可

//p.s.申请内存空间的过程在赋值中完成

T->ch = NULL;

T->length = 0;

return OK;

}

//字符串赋值

Status StrAssign(HString \*T, char \*p)

{

//1.判断T是否已有内容,有则释放

//2.判断赋值的内容是否为空,为空则不赋值

//3.根据长度向内存申请空间,遍历赋值给T,长度等于字符串长度

//p.s.在这里赋值不赋\0,在打印时通过长度来判断字符串结尾

int i,j;

char \*c = p;

if (T->ch)

free(T->ch);

for (i = 0,c = p;\*c;++i,++c)//求p的长度

if (!i)

{

T->ch = NULL;

T->length = 0;

return ERROR;

}

else

{

T->ch = (char \*)malloc(i \* sizeof(char));

if(!T->ch)

exit(OVERFLOW);

for (j = 0; j < len; ++j)

T->ch[j] = p[j];

T->length = len;

return OK;

}

}

//字符串长度

Status StrLength(HString T)

{

return T.length;

}

//字符串判空

Status StrEmpty(HString T)

{

if (T.length == 0)

return TRUE;

else

return FALSE;

}

//字符串联接

Status Concat(HString \*T, HString S1, HString S2)

{

//1.申请长度为S1和S2之和的字符串空间

//2.先将S1的元素逐个赋值到T中

//3.再将S2的元素逐个赋值到T中

int i;

if (T->ch)

free(T->ch);

T->ch = (char \*)malloc((S1.length + S2.length) \* sizeof(char));

if (!T->ch)

exit(OVERFLOW);

for (i = 0; i < S1.length; ++i)

T->ch[i] = S1.ch[i];

for (i = 0; i < S2.length; ++i)

T->ch[i + S1.length] = S2.ch[i];

T->length = S1.length + S2.length;

/\*T->length = S1.length + S2.length;

T.ch[0…S1.length-1] = S1.ch[0…S1.length-1];

T.ch[S1.length…T.length-1] = S2.ch[0…S2.length-1]; \*/

return OK;

}

//删除字符串中某个位置固定长度的子串

Status StrDelete(HString \*T, int pos, int len)

{

//pos是字符串中的位置,删除包括pos的len长度

int i;

if (pos >= T->length)

return ERROR;

else if(pos + len > T->length)

len = T->length - pos + 1;

for (i = pos - 1; i < T->length - len; ++i)

T->ch[i] = T->ch[i + len];

T->length -= len;

T->ch = (char \*)realloc(T->ch, T->length \* sizeof(char));

if (!T->ch)

exit(OVERFLOW);

return OK;

}

//插入操作

Status StrInsert (HString &S, int pos, HString T)

{

//1、为串 S 重新分配大小等于串 S 和串 T 长度之和 的存储空间；

// 2、进行串值的复制。

if (pos<1 || pos > S.length +1) return ERROR; // 插入位置不合法

if (T.length) // T 非空，则为 S 重新分配空间并插入 T

{ if (!(S.ch=(char \*) realloc(S.ch,(S.length+T.length)\*sizeof(char))))

exit(OVERFLOW);

for (i=S.length-1; i>=pos-1; --i) //为插入 T 而腾出位置

S.ch[i+T.length]=S.ch[i];

S.ch[pos-1…pos+T.length-2]=T.ch[0…T.length-1]; //插入 T

S.length+=T.length;

}

return OK;

}

//在字符串S中索引位置pos之后的子串t

Status Index(HString S, HString T, int pos)

{

//同定长顺序存储表示法

//p.s.传入的pos是字符串的位置,从1开始

//p.s.初始状态下T为非空串

if (StrEmpty(T))

return ERROR;

int i = pos - 1, j = 0;

while(i < S.length && j < T.length)

{

if (S.ch[i] == T.ch[j])

{

++i;

++j;

}

else

{

i = i - j + 1;

j = 0;

}

}

if (j >= T.length)

return i - j + 1;

else

return 0;

}

Status Replace(HString \*T, HString S1, HString S2) //将字符串T中等于S1的子串替换成为S2

{

//循环索引子串S1在字符串T中的位置(每次的位置从上一次位置后开始查找)

//从查找到的位置-1开始替换

//p.s.初始状态下S1为非空串

int pos = 0;

if (StrEmpty(S1))

return ERROR;

//当pos存在时循环,当全部索引完毕后pos为0

//将索引到的该位置对应的子串删除后再插入新的子串

do

{

pos = Index(\*T, S1, pos);

if (pos)

{

StrDelete(T, pos, StrLength(S1));

StrInsert(T, pos, S2);

}

}

while(pos);

return OK;

}

Status SubString(HString \*Sub, HString S, int pos, int len)

{

int i;

if (pos < 1 || len > S.length || len < 0 || len > S.length - pos + 1)

exit(OVERFLOW);

if (Sub->ch)

free(Sub->ch);

//如果查询的长度为0,则子串置空

if (len == 0)

{

Sub->ch = NULL;

Sub->length = 0;

}

else

{

Sub->ch = (char \*)malloc(len \* sizeof(char));

for (i = 0; i < len; ++i)

Sub->ch[i] = S.ch[pos + i - 1];

/\*Sub.ch = (char \*)malloc(len\*sizeof(char));

Sub.ch[0…len-1] = S[pos-1…pos+len-2];

Sub.length = len; \*/

Sub->length = len;

}

return OK;

}

// StrCompare

int StrCompare(HString S, HString T)

{

for (i=0; i<S.length && i<T.length; ++i)

if (S.ch[i]!=T.ch[i]) return S.ch[i]-T.ch[i];

return S.length-T.length;

}

//串复制 Strcopy 算法

status Strcopy(HString &T, HString S)

{

if (T.ch) free(T.ch);

n=S.length;

if (n!=0)

{ T.ch=(char \*)malloc(n\*sizeof(char));

T.ch[0..n-1]=S.ch[0..n-1];

T. length=S.length; }

return OK;

}

//清空串 ClearString 算法

status ClearString(HString &S)

{

//将串 S 清为空串

if (S.ch) { free(S.ch); S.ch=NULL; }

S.length=0;

return OK;

}

**堆存储结构的优点：**堆存储结构既有顺序存储结构的特点，处理（随机取子串）方便，操作中对串长又没有任何限制，更显灵活，因此在串处理的应用程序中常被采用。

定长顺序存储表示和堆分配存储表示通常为高级程序设计语言所采用。

# 串的块链存储及优缺点

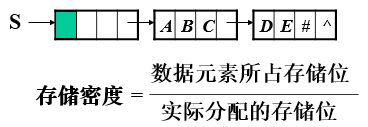
## 串的块链存储表示

串值也可用单链表存储，简称为链串。 链串与单链表的差异只是它的结点数据域为单个字符。



对于字符来说，char是1个字节，指针是4个字节，因此空间利用率极低。

为了提高**空间利用率**，可使每个结点存放**多个字符**（这是顺序串和链串的综合 (折衷) ），称为**块链结构**。



**实际应用时，可以根据问题所需来设置结点的大小。例如：在编辑系统中，整个文本编辑区可以看成是一个串，每一行是一个子串，构成一个结点。即：同一行的串用定长结构（80个字符），行和行之间用指针相联接。**

**结点结构用 C 语言定义如下：**

#define CHUNKSIZE 80 // 可由用户定义的块大小

typedef struct Chunk // 结点结构

{

char ch[CHUNKSIZE];

struct Chunk \*next;

} Chunk;

/\*为了便于进行串的操作（联接），当以块链存储串值时，除头指针外还可附设一个尾指针指示链表中

的最后一个结点，并给出当前串的长度。其结构用C语言定义如下： \*/

typedef struct // 串的链表结构\_

{

Chunk \*head, \*tail; // 串的头和尾指针\_

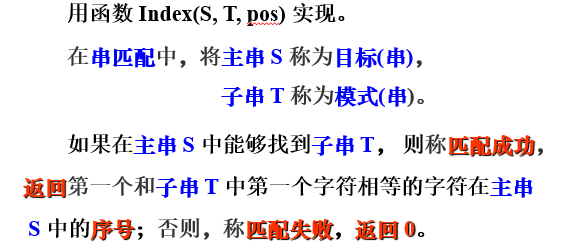
int curlen; // 串的当前长度\_

} LString;

# 串的模式匹配算法

**模式匹配 ：**子串定位运算。

(串匹配)就是在主串中找出子串出现的位置。



**例如：**

**主串：ss=‘abcabcabdabcdef’;**

**子串：tt1=‘abc’;子串：tt2=‘abd’;**

**Index(ss, tt1, 2) ; //4**

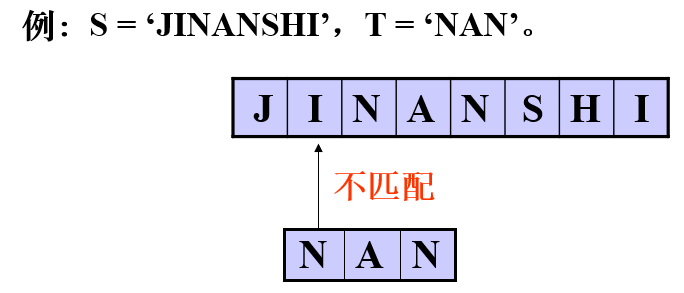
**Index(ss, tt1, 5) ; //10**

**Index(ss, tt2, 8) ; //0**

**模式匹配**是各种处理系统中最重要的操作之一，也是一个比较复杂的串操作。模式匹配的算法不同，效率将有很大差别。同一算法应用不同，效率亦有很大差别。

## 朴素的模式匹配算法

**算法思想：**从主串 S 的第 pos 个字符起和模式 T 的第一个字符比较之，若相同，则继续比较后续字符；否则从主串 S 的下一个字符起再重新和模式 T 的字符比较之。



**当采用定长顺序存储结构时，实现此操作的算法如下：**

int Index(SString S, SString T, int pos)

{ i = pos; j = 1;

while (i <= S[0] && j <= T[0])

{ if (S[i] == T[j])

{ ++ i; ++ j; } // 继续比较后继字符

else

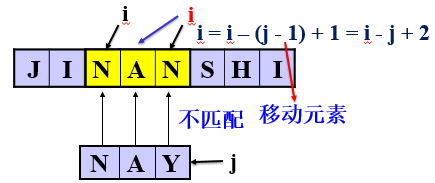
{ i = i – j + 2; j = 1; } // 指针后退重新开始匹配

}

if ( j >T[0]) return i -T[0];

else return 0;

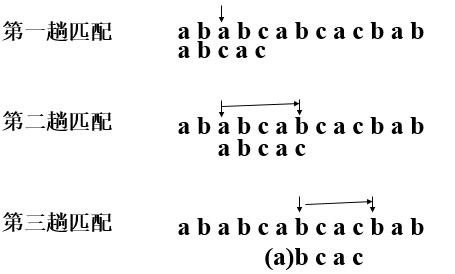
} // Index



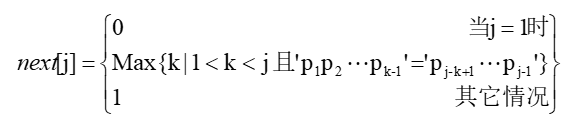
朴素的模式匹配算法评价

* 设计思想简单、易于理解。
* 通常情况下，效率比较高。经常被程序员选用。此时算法的时间复杂度为：O(n+m)。m、n分别为主串和子串的长度。
* 某些特殊的情况下，效率比较低。此时算法的时间复杂度为：O(n\*m)m、n分别为主串和子串的长度。（**比如计算机中00001 和00000000000000001模式匹配就是效率相当低的一种例子。**）
* 因此我们学习另一种高效的匹配算法----KMP算法。

## KMP算法



若令next[j] = k, 则next[j]表明当模式中第j个字符与主串中相应字符“失配”时，在模式串中需重新和主串中该字符进行比较的字符的位置(k)。由此可以得出next函数的定义：



由此定义可以推出下列模式串的next函数值：

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| j | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 模式串 | a | b | a | a | b | c | a | c |
| next[j] | 0 | 1 | 1 | 2 | 2 | 3 | 1 | 2 |

int Index\_KMP(SString S, SString T, int pos) {

i = pos; j = 1;

while (i <= S[0] && j <= T[0]) {

if (j = = 0 || S[i] = = T[j])

{ ++i; ++j; } // 继续比较后继字符

else j = next[j]; // 模式串向右移动

}

if (j > T[0]) return i-T[0]; // 匹配成功

else return 0;

}