

字符串 (String)

字符串是 n (≥ 0) 个字符的有限序列，

记作 $S: "c_1c_2c_3 \dots c_n"$

其中， S 是串名字

$"c_1c_2c_3 \dots c_n"$ 是串值

c_i 是串中字符

n 是串的长度。

例如， $S = \text{"Tsinghua University"}$

字符串抽象数据类型定义

ADT String {

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

数据关系: $R1 = \{ \langle a_{i-1}, a_i \rangle | a_{i-1}, a_i \in D, i=2, \dots, n \}$

基本操作:

StrAssign(&T, chars) 生成字符串T

StrCopy(&T, S) 复制字符串S

StrEmpty(S) 判断S是否为空串

StrCompare(S, T) 比较字符串S和T

StrLength(S) 求字符串S长度

ClearString(&S) 清空字符串S

Concat(&T, S1, S2) 连接字符串S1和S2

SubString(&Sub, S, pos, len) 求S长度为len位置为pos的子串

Index(S, T, pos) 求子串在主串中的位置

Replace(&S, T, V) 在S中用子串V替换子串T

StrInsert(&S, pos, T) 在S中插入子串T

StrDelete(&S, pos, len) 在S中删除长度为len的子串

DestroyString(&S) 销毁串S

}

```
int Index ( String S, String T, int pos ) {  
    if (pos > 0) {  
        n=StrLength(S); m=StrLength(T); i=pos;  
        while (i<=n-m+1) {  
            SubString (sub, S, i, m);  
            if (StrCompare(sub,T)!=0) ++i;  
            else return i;  
        }//while  
    }//if  
    return 0;  
}//Index
```

串的实现和表示

1.定长顺序存储表示

用一组地址连续的存储单元存储串值的字符序列。

```
#define MAXSTRLEN 255
```

```
typedef unsigned char
```

```
    SString[MAXSTRLEN+1]; //0号单元存串长
```

(1)串连接

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

//如果连接后的串过长则截断

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

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]<MAXSTRLEN) {

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

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

T[0]=MAXSTRLEN; uncut=FALSE;

}

else {

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

}

return uncut;

}//Concat

(2)求子串

```
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
```

使用顺序存储结构过程中可能出现串长度超过数组上限，经过截断的串已经不完整，克服这个问题可使用动态分配串值的存储空间。

2. 堆分配存储表示

堆是操作系统中为进程分配的自由存储空间，在C语言中用malloc()和free()来管理。

```
typedef struct {
    char *ch;
    int length;
}HString;
Status StrInsert(HString &S, int pos, HString T){
//在串S的第pos个字符之前插入串T
    if (pos<1||pos>S.length+1) return ERROR;
    if(T.length){
        if(!(S.ch=(char *)realloc(S.ch,(S.length+T.length)*sizeof(char))))
            exit(OVERFLOW);
        for (i=S.length-1;i>=pos-1;--i)
            S.ch[i+T.length]=S.ch[i];
        S.ch[pos-1..pos+T.length-2]=T.ch[0..T.length-1];
        S.length+=T.length;
    }
    return OK;
}//StrInsert
```

```
Status StrAssign(HString &T,char *chars){  
    //生成值为chars的串T  
    if(T.ch) free(T.ch);  
    for(i=0, c=chars;c;++i,++c);//求chars长度i  
    if(!i) {T.ch=NULL; T.length=0;}  
    else {  
        if(!(T.ch=(char *)malloc(i*sizeof(char))))  
            exit (OVERFLOW);  
        T.ch[0..i-1]=chars[0..i-1];  
        T.length=i;  
    }  
    return OK;  
} //StrAssign
```



```
int StrLength(HString S) {
```

```
//求串长度
```

```
    return S.length;
```

```
}//StrLength
```

```
int StrCompare(HString S, HString T){
```

```
//比较串S和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;
```

```
}//StrCompare
```

```
Status ClearString(HString &S){
```

```
//清空串S
```

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

```
    S.length=0;
```

```
    return OK;
```

```
}//ClearString
```

```
Status Concat(HString &T,HString S1,HString  
S2) {
```

```
//连接串S1和S2
```

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

```
if(!(T.ch=(char  
*)malloc((S1.length+S2.length)*sizeof(char))))
```

```
exit(OVERFLOW);
```

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

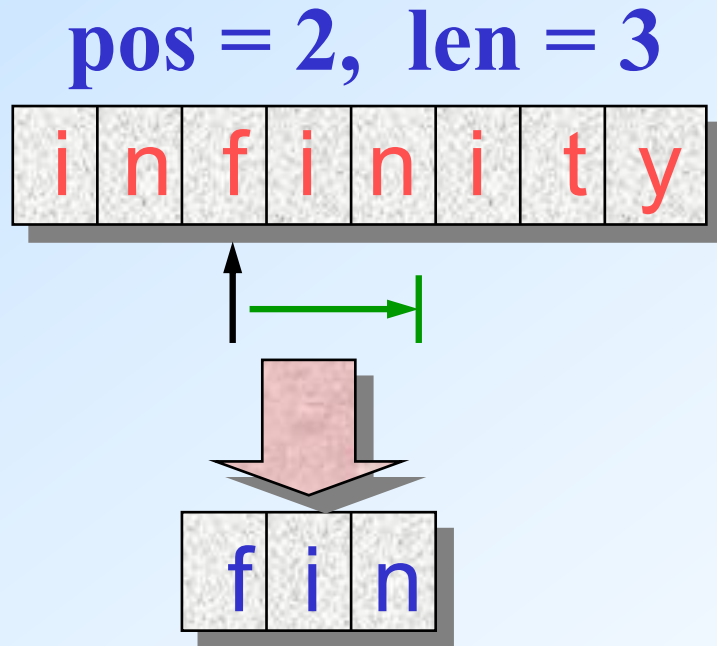
```
T.length=S1.length+S2.length;
```

```
T.ch[S1.length..T.length-1]=S2.ch[0..S2.length-  
1];
```

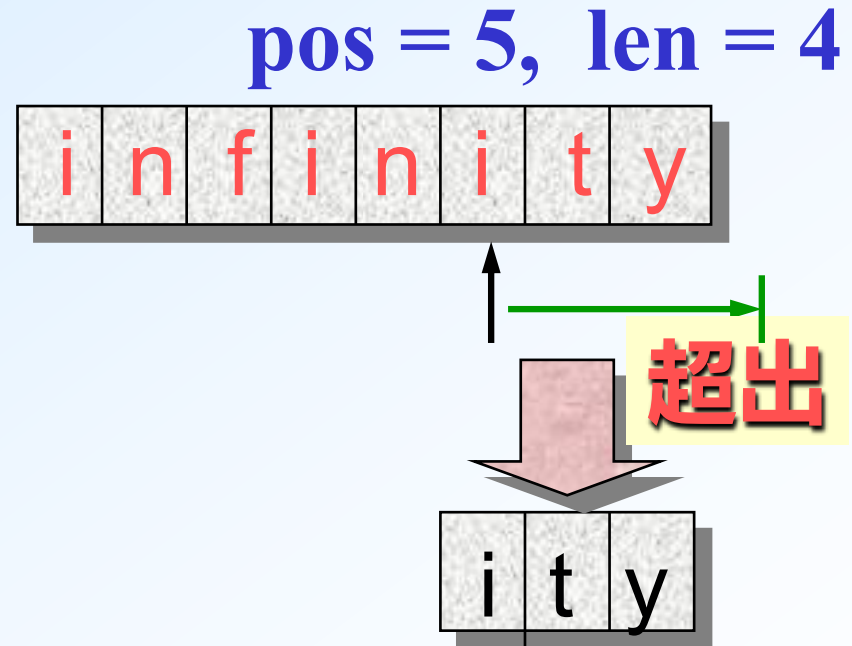
```
return OK;
```

```
}//Concat
```

提取子串的算法示例



$\text{pos} + \text{len} - 1$
 $\leq \text{length} - 1$



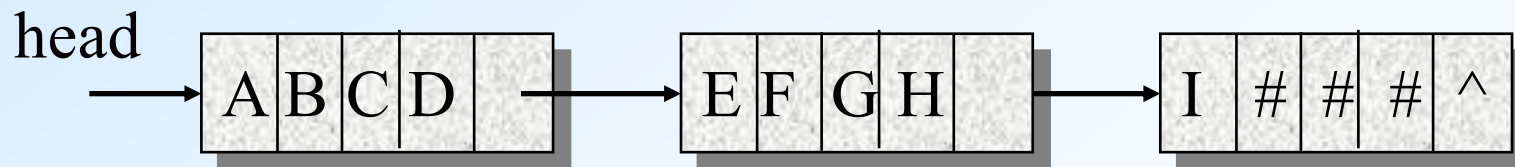
$\text{pos} + \text{len} - 1$
 $\geq \text{length}$

```
Status SubString(HString &Sub,HString S,int pos,int
len){
//返回串S中从pos位置起长度为len的子串
    if (pos<1||pos>S.length||len<0||len>S.length-pos+1)
        return ERROR;
    if(Sub.ch) free(Sub.ch);
    if(!len){Sub.ch=NULL; Sub.length=0;}
    else{
        Sub.ch=(char *)malloc(len*sizeof(char));
        Sub.ch[0..len-1]=S[pos-1..pos+len-2];
        Sub.length=len;
    }
    return OK;
}//SubString
```

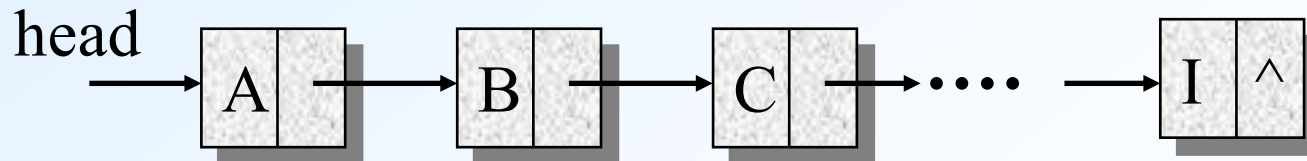
3.串的块链存储表示

用链表存储串，每个结点存储串的 n 个字符，
当串长不是 n 的整数倍时最后一个结点剩
余位置用空字符#补齐。

如串ABCDEFGHI当 $n=4$ 时：



当 $n=1$ 时：



串的块链存储表示

```
#define CHUNKSIZE 80
typedef struct Chunk{
    char ch[CHUNKSIZE];
    struct Chunk *next;
}Chunk;
```

```
typedef struct {
    Chunk *head, *tail;//为指针便于连接操作
    int curlen;
}LString;
```

块链存储方式便于连接操作，但占用存储量大，操作复杂。

串的模式匹配

定义 在串中寻找子串（第一个字符）
在串中的位置

词汇 在模式匹配中，子串称为模式，
串称为目标。

示例 目标 T：“Beijing”

模式 P：“jin”

匹配结果 = 3

第1趟

T

P

$i=1$
a b **b** a b a
 $j=1$
a b **a**

穷举的模式
匹配过程

第2趟

T

P

$i=2$
a **b** b a b a
 $j=1$
a b a

匹配模式 : aba

第3趟

T

P

$i=3$
a b **b** a b a
 $j=1$
a b a

第4趟

T

P

$i=4$
a b b **a** b a
 $j=1$
a b a

✓

匹配算法:

```
int Index(SString S, SString T, int pos){  
    //从串S中pos位置开始搜索模式T  
    i=pos; j=1;  
    while (i<=S[0] && j<=T[0]){  
        if(S[i]==T[j]) {++i; ++j;}  
        else {i=i-j+2; j=1;}//i从初始位置后移, j复位  
    }  
    if (j>T[0]) return i-T[0];  
    else return 0;  
} //Index
```

模式匹配的改进算法：KMP算法

匹配模式abcac

第1趟

T	a	b	a	b	c	a	b	c	a	c	b	a	b
P	a	b	c										

i=3 j=3

第2趟

T	a	b	a	b	c	a	b	c	a	c	b	a	b
P			a	b	c	a	c						

i=3 i=7 j=5

第3趟

T	a	b	a	b	c	a	b	c	a	c	b	a	b
P						a	b	c	a	c			

i=7 j=2

√

第1趟

T a b **a** b c a b c a c b a b

P a b **c**

在第一趟中 $i=3, j=3$ 时比较失败，按穷举法 i 应变为初始值加1，即 $i=2$ ； j 每次重新变为1，再开始下一轮比较，但事实上 $i=2$ 时已经作过比较不需要再比较一次，所以 i 值不变，只需 j 变为1再开始比较即可。

第2趟

T a b a b c a b c a c b a b

P a b c a c

在第二趟中 $i=7, j=5$ 时比较失败，按穷举法 i 应变为初始值加1，即 $i=4; j$ 变为1，再开始下一轮比较，但事实上 $i=4, 5, 6$ 时已经作过比较不需要再比较一次，所以 i 值不变，而 j 值也不需变为1再开始比较，因为字符 c 前的'a'在P的开始也有，而且也比较过了， j 从2开始即可。

总之，利用已经比较完的结果，可以减少下次比较的次数，不需要移动指针i的位置，只需移动指针j的位置，而且按照模式中的包含关系，指针j也不需每次从1开始。

由模式串的包含关系找到j的移动位置：例

模式串	a	b	a	a	b	c	a	c
当前j	1	2	3	4	5	6	7	8
下一j	0	1	1	2	2	3	1	2

说明：next[j]=0指的是第一个字符比较就不成功，i指针需要后移，j指针变为1；其它值表示i指针不动，j指针移动的位置。

KMP算法：

```
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}  
        else j=next[j];  
    }  
    if (j>T[0]) return i-T[0];  
    else return 0;  
} //Index_KMP
```

计算next函数值算法：

```
void get_next(SString S, int next[ ]) {  
    i=1; next[1]=0; j=0;  
    while(i<T[0]) {  
        if (j==0 || T[i]==T[j]) {++i; ++j; next[i]=j; }  
        else j=next[j];  
    }  
} //get_next
```