第四章、串

4.1. 串的定义和实现

4.1.1、串的定义

定义：

即字符串（String）是由零个或多个字符组成的有限序列。

基本术语：

- 子串：串中任意个连续的字符组成的子序列。Eg：'iPhone'，'Pro M' 是串T 的子串
- 主串：包含子串的串。Eg：T 是子串'iPhone'的主串
- 字符在主串中的位置：字符在串中的序号。Eg：'1'在T中的位置是8(第一次出现)
- 子串在主串中的位置：子串的第一个字符在主串中的位置 。Eg：'11 Pro'在 T 中的位置为8(`注意：位序从1开始而不是从0开始`)

串是一种特殊的线性表，数据元素之间呈线性关系

4.1.2、串的存储结构

顺序存储

```c
#define MAXLEN 255      // 预定义最大串长为255
typedef struct{
    char ch[MAXLEN];    // 每个分量存储一个字符
    int length;         // 串的实际长度
}SString;               // 静态数组实现

typedef struct {
    char *ch;           // 按串长分配存储区
    int length;
}HString;               // 动态分配
HString S;
S.ch = (char *)malloc(MAXLEN *sizeof(char));
S.length = 0;
```

链式存储

```c
typedef struct StringNode{
    char ch;
    struct StringNode * next;
}StringNode,* String;
```


```
typedef struct StringNode{
    char ch[4];
    struct StringNode * next;
}StringNode,* String;
```

基于顺序存储实现成操作

切一段子串：

```
/*
    功能:用Sub返回主串S的第pos个字符起长度为len的子串
    参数:主串S,返回的子串Sub,位置pos,长度len
    返回值:
*/
bool SubString(SString &Sub,SString S,int pos,int len) {
    if(pos+len-1 > S.length)
        return false;
    for(int i = pos; i<pos+len; i++)
        Sub.ch[i-pos+1] = S.ch[i];
    Sub.length = len;
    return true;
}
```

比较操作：

```
/*
    功能:若S>T，则返回值>0；若S=T，则返回值=0；若S<T，则返回值<0。
    参数:字符串S,字符串T
    返回值:
*/
int StrCompare(SString S,SString T) {
    for(int i = 0;i <= S.length && i <= T.length; i++){
        if(S.chp[i] != T.ch[i])
            return S.ch[i]-T.ch[i];
    }
    // 扫描通过的所有字符都相同，则长度长的串更大
    return S.length-T.length;
}
```

定位操作：

```
/*
```

```
    功能:定位操作。若主串S中存在与子串T值相同的子串，则返回它在主串S中第一次出现的位置；否则返回值为0。
    参数:字符串S,字符串T。
    返回值:
*/
int Index(SString S,SString T) {
    int i = 0 ,n = StrLength(S), m=StrLeng(T);
    SString sub;
    while(i <= n-m+1) {
        SubString(sub,S,i,m);
        if(StrCompare(Sub,T)!=0) ++i;
        else return i;  // 返回子串在主串中的位置
    }
    return 0;
}
```

4.2. KMP 算法

朴树模式匹配算法

最坏时间复杂度：O(nm)

```
int StrCompare(SString S,SString T) {
    for(int 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;
}
```

KMP 算法

```
int Index_KMP(SString S,SString T,int next[]) {
    int i = 1,j = 1;
    while (i<=S.length && j <= T.length) {
        if(j == 0||S.ch[i]==T.ch[i]) {
            ++i;
            ++j;            // 继续比较后续字符
        }else {
            j = next[j];    // 模式串向右移动
            // j=nextval[j];  KMP算法优化,当子串和模式串不匹配时
        }
    }
    if(j > T.length)
        return i-T.length;  // 匹配成功
    else
```

```
15          return 0;
16  }
```

next数组的求法

nextval数组的求法

> next数组手算方法：当第j个字符匹配失败，由前 1~j-1 个字符组成的串记为S，则： next[j]=S的最长相等前后缀长度+1
>
> 特别地，next[1]=0

```
nextval数组的求法：
先算出next数组
先令nextval[1]=0
for (int j=2; j<=T.length; j++) {
        if(T.ch[next[j]]==T.ch[j])
                nextval[j]=nextval[next[j]];
        else
                nextval[j]=next[j];
}
```

序号j	1	2	3	4	5	6
模式串	a	b	a	b	a	a
next[j]	0	1	1	2	3	4
nextval[j]	0	1	0	1	0	4