

String

引言

串

- 逻辑结构：
 - $s = 'a_1a_2\dots a_n'$
- 存储结构：
 - 定长顺序存储结构
 - 堆存储结构
 - 块链存储结构
- 操作：
 - 若干函数
 - 模式匹配算法

模式匹配

- 即子串定位算法，如何实现 $\text{Index}(S, T, pos)$ 函数
- 精华：KMP算法——快速查找(用 $\text{next}[j]$ 或 $\text{nextval}[j]$)

串类型的定义

- 有限
- 数据元素为：单个字符
- 隐含结束符 '\0' 即 ASCII 码NULL
- 为何单独讨论串：
 - 比其他数据类型更复杂
 - 程序设计中处理对象很多是串类型

若干术语

- 串长：字符个数， $n=0$ 时为空串
- 空格（白）串：由 空格符 组成
- 空串与空格串：

问：空串和空格串有无区别？

答：有区别。

- 空串 (Null String) 是指长度为零的串；
而空格串 (Blank String), 是指包含一个或多个空格字符 ‘’ (空格键) 的字符串.

- 子串：给定串S中，任意连续的字符序列，S叫 **主串**
- 字串位置：子串的 **第一个** 字符在主串中的序号
- 字符位置：字符的序号
- 串相等：串长度相等，且对应位置字符相等

“空串是任意串的子串；任意串S都是S本身的子串，除S本身外，S的其他子串称为S的**真子串**。”

——《数据结构与算法》中山大学出版社

串的抽象数据类型定义

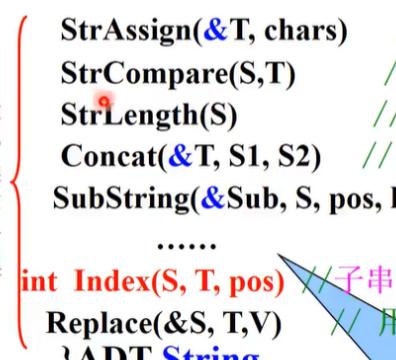
串的抽象数据类型定义 (参见教材P71)

ADT String{

Objects: $D = \{a_i \mid a_i \in \text{CharacterSet}, i=1, 2, \dots, n, n \geq 0\}$

Relations: $R1 = \{<a_{i-1}, a_i> \mid a_{i-1}, a_i \in D, i=2, \dots, n\}$

functions: //至少有13种基本操作

最
小
操
作
子
集

StrAssign(&T, chars) // 串赋值，生成值为chars的串T
StrCompare(S,T) // 串比较，若S>T，返回值大于0...
StrLength(S) // 求串长，即返回串S中的元素个数
Concat(&T, S1, S2) // 串连接，用T返回S1+S2的新串
SubString(&Sub, S, pos, len) //求S中pos起长度为len的子串
.....
int Index(S, T, pos) //子串定位函数（模式匹配），返回位置
Replace(&S, T, V) //用子串V替换子串T

}ADT String

C语言中已有类似的串运算函数！

复习：C语言中常用的串运算

注：用C处理字符串时，要调用标准库函数 #include<string.h>



- 习题

例1： (参见P71) 设 **s = 'I AM A STUDENT'**,
t = 'GOOD', **q = 'WORKER'**。求：

StrLength(s) =	14	Index(S, T, pos) // 返回子串T在pos之后的位置
StrLength(t) =	4	
SubString(&sub, s, 8, 7)=	'STUDENT'	
SubString(&sub, t, 2, 1)=	'O'	
Index(s, 'A')=	3	
Index(s, t)=	0	(s 中没有 t='GOOD')
Replace(&s, 'STUDENT', q)=	'I AM A WORKER'	

- 注意：

- t 的长度为4，不是5
- 但实验出来，`string s = "good"; cout << s.length() << endl;` 出来是5
- `Index(s, 'A')` 是3不是2，即第3个

串的表示和实现

- 与 线性表 不同，是以 串的整体 作为操作对象
- 三种机内表示方法
 - 顺序存储
 - 定长的顺序存储表示
 - 地址连续
 - 静态存储
 - 堆分配存储表示
 - 地址连续
 - 动态分配

- 链式存储

- 链式

顺序存储

数组•定长

```
#define MAXstrlen 255
typedef unsigned char SString[MAXstrlen + 1] SString;
SString s;
```

说明：

- 一般用 **SString[0]** 来存放 **串长** 信息
- C 语言约定：在串尾加结束符 '\0'，以利操作加速，但不计入串长
- （用 **首址和串长** 或 **首址和尾标记** 来描述串数组）
- 若字符串超过 MAXstrlen 则自动截断（静态数组存不进去嘛）

堆•动态分配

```
typedef struct{
    char *ch; //若 不是 空串, 按串长分配空间, 否则ch = NULL;
    //结合下面的 示例, ch 是指向这个连续物理空间的首地址的
    int length; //串长度
}HString
```

- 特点：仍用 **连续存储单元**，存储空间在执行过程中 **动态分配**
 - 动态分配的一个简单例子：
 - 原有1~10的10个地址，现在需要20个且第11个为已占用地址；于是重新分配一片连续的20个地址。

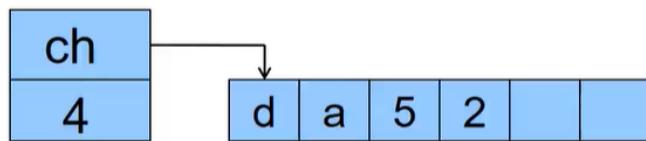
堆分配存储特点：仍用一组连续的存储单元来存放串，
但存储空间是在程序执行过程中**动态分配**而得。

堆T的存储结构描述：

```
Typedef struct {
    char *ch; //若非空串, 按串长分配空间; 否则ch=NULL
    int length; //串长度
}HString
```

各分量书写方式：
T.ch
T.length

堆分配存储示例：串 ‘da52’ 的堆分配存储结构
如下图所示：



思路：利用malloc函数合理预设串长空间。

特点：若在操作中串值改变，还可以利用realloc函数按新串长度增加空间（即动态数组概念）。

malloc：在内存中分配一段连续的存储空间

- 建堆

例1：编写建堆函数 (参见教材P76)

```
Status StrAssign(HString &T, char *chars)
{
    //生成一个串T, T值←串常量chars
    if (T.ch) free(T.ch); //释放T原有空间
    for (i=0, c=chars; c; ++i, ++c); //求chars的串长度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;
}
```

19

- 插入字符串

例2：用“堆”方式编写串插入函数（参见教材P75）

```
Status StrInsert ( HString &S, int pos, HString T )
{
    //在串S的第pos个字符之前（包括尾部）插入串T
    if (pos<1||pos>S.length+1) return ERROR; //pos不合法则告警
    if(T.length){ //只要串T不空，就需要重新分配S空间，以便插入T
        if( !(S.ch=(char*)realloc(S.ch,(S.length+T.length)*sizeof(char))) )
            exit(OVERFLOW); //若开不了新空间，则退出
        for ( i=S.length-1; i>=pos-1; --i ) S.ch[i+T.length] = S.ch[i];
        //为插入T而腾出pos之后的位置，即从S的pos位置起全部字符均后移
        S.ch[pos-1..pos+T.length-2] = T.ch[0..T.length-1]; //插入T，略/0
        S.length += T.length; //刷新S串长度
    }
    return OK;
} //StrInsert
```

20

注意：使用了 `realloc` 函数，重新分配了 (`S.length + T.length`) 个空间

链式存储

- 易于删除和插入
- 块链结构
 - 多个字符存入一个结点里

链式存储特点：用链表存储串值，易插入和删除。

法1：链表结点的数据分量长度取1（个字符）



法2：链表结点（数据域）大小取n（例如n=4）



讨论：法1存储密度为 $\frac{1}{2}$ ；法2存储密度为 $\frac{9}{15} = \frac{3}{5}$ ；

显然，若数据元素很多，用法2存储更优—称为**块链结构**

块链类型定义

```
#define CHUNKSIZE 80 // 每块的大小
// 定义结点类型
typedef struct Chunk{
    char ch[CHUNKSIZE]; // 数据域
    struct Chunk* next; // 指针域
}Chunk;

// 块链表的整体描述
```

```
//定义 串 类型
typedef struct{
    Chunk *head; // 头指针
    Chunk *tail; // 尾指针
    int curLen; // 结点个数
}LString;
```

块链类型定义：

对块链表的整体描述

```
typedef struct { //定义用链式存储的串类型
    Chunk *head; //头指针
    Chunk *tail; //尾指针
    int curLen; //结点个数
} LString; //串类型只用一次，前面可以不加结构名称
```

```
#define CHUNKSIZE 80 //每块大小，可由用户定义
typedef struct Chunk { //首先定义结点类型
    char ch[CHUNKSIZE]; //每个结点中的数据域
    struct Chunk * next; //每个结点中的指针域
}Chunk;
```

块链函数的编写略

对每个结点的描述

串的模式匹配算法

- Word 文档查找
- 算法目的：确定主串中所含子串第一次出现的位置
 - 典型函数：Index(S, T, pos)：从S的pos位置开始找
- 算法类型：
 - BF算法**：速度慢，带回溯（古典的、经典的、朴素的、穷举的）
 - KMP算法**：避免回溯，匹配速度快

BF的实现

```
//BS算法匹配
int Index(const SString& S, const SString& T, const int& pos)
{/*返回子串T在主串S中的第pos个字符之后的位置。若不存在，则函数值为0。其中T非空且
1<=pos<=Strlen(S) */
    int q = pos, p=1;
    while(q + T[0] <= S[0] && p <= T[0])
    {
        //如果i, j两指针在正常范围
        if(S[i]==T[i]){
            ++i;
            ++j;
        }else{
```

```

        i = i - j + 2; // 相当于是 i 从下一个开始
        j = 1;
    }
}

if(p > T[0]) return i - T[0]; // T子串指针p正常到队尾，匹配成功
return 0; //否则i先到尾就不正常
}

```

分析时间复杂度

主串长n，子串长m，可能匹配成功的位置 $i = (1 \sim n-m+1)$

- 最好的情况

- 在第 i 个位置匹配成功，且前 $i-1$ 个位置都仅匹配了 1 次
- 第 i 个位置总共比较了 $(i-1+m)$ 次
- 则有如下公式

$$\text{平均比较次数} = \sum_{i=1}^{n-m+1} p_i(i-1+m) = \frac{1}{n-m+1} \sum_{i=1}^{n-m+1} (i-1+m) = \frac{1}{2}(m+n)$$

最好的情况：平均时间复杂度 $= O(n+m)$

- 最坏的情况

- 在第 i 的位置匹配成功，且前 $i-1$ 个位置都匹配了 m 次
- 第 i 个位置总共比较了 $(i * m)$ 次
- 则有如下公式，

$$\text{平均比较次数} = \sum_{i=1}^{n-m+1} p_i(i * m) = \frac{m}{n-m+1} \sum_{i=1}^{n-m+1} i = \frac{1}{2}m(n-m+2)$$

设 $n >> m$ ，最坏的情况下，平均时间复杂度 $= O(n * m)$

KMP算法

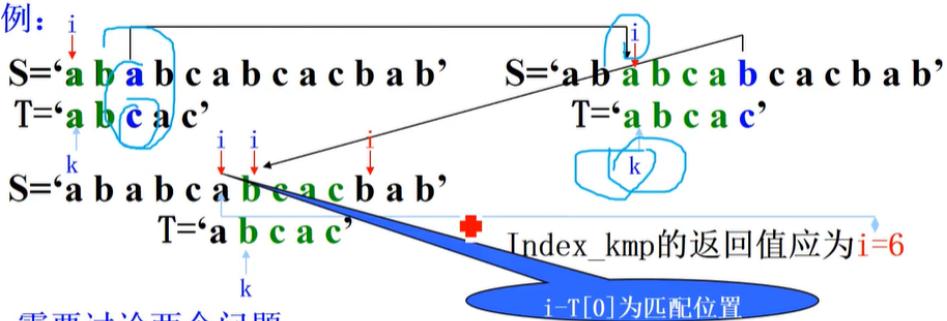
基本思想

- 失配时， i 不变， j 回溯到 k 的位置
- j 和 k 满足： $"T_1 T_2 \dots T_{k-1}" == "T_{j-(k-1)} \dots T_{j-1}"$
 - j 和 k 分别是匹配串 T 的 子串 中的 前后两个 相等 的 子串 的 尾结点 的下标 +1
 - i 是 S 的指针，不回溯

① KMP算法设计思想：（参见教材P80-84）

尽量利用已经部分匹配的结果信息，尽量让*i*不要回溯，加快模式串的滑动速度。

例：



- 建立 $k = \text{next}[j]$ 的函数关系
- 重点：对 $\text{next}[j]$ 的求解：KMP算法的实现

新起点 k 怎么求？

根据模式串T的规律： $'T_1 \dots T_{k-1}' = 'T_{j-(k-1)} \dots T_{j-1}'$
由当前失配位置j(已知)，可以归纳出计算新起点k的表达式。

令 $k = \text{next}[j]$ (k 与 j 显然具有函数关系)，则

$$\text{next}[j] = \begin{cases} 0 & \text{当 } j=1 \text{ 时} \\ \max \{ k \mid 1 < k < j \} & \text{且 } 'T_1 \dots T_{k-1}' = 'T_{j-(k-1)} \dots T_{j-1}' \\ 1 & \text{其他情况} \end{cases}$$

//不比较
取T首与T_j处最大的相同子串

讨论：

- (1) $\text{next}[j]$ 有何物理意义？
- (2) $\text{next}[j]$ 具体怎么求？—即KMP算法的实现

$\text{next}[j]$ 的物理意义

$\text{Next}[j]$ 函数表征 模式 T 中 最大的 相同前缀子串 和 后缀字串 (真子串) 的长度

备注：

1. 关于 k 的，前缀子串，一定是从下标 1 开始的
2. 模式中相似部分越多，则 $\text{next}[j]$ 函数越大，即模式 T 字符之间 相关度 越高，也表示第 j 个位置以前与主串部分匹配的字符数越多

(1) $\text{next}[j]$ 有何物理意义? $T = 'a\ b\ a\ a\ b\ c\ a\ c'$

$\text{next}[j] = \max \{ k \mid 1 < k < j \text{ 且 } T_1 \dots T_{k-1} = T_{j-(k-1)} \dots T_{j-1} \}$

模式串从第1位往右
直到 $k-1$ 位

模式串从 j 的前一位
往左经过 $k-1$ 位

$\text{next}[j]$ 函数表征着模式 T 中最大相同前缀子串和后缀子串(真子串)的长度。

可见, 模式中相似部分越多, 则 $\text{next}[j]$ 函数越大, 它既表示模式 T 字符之间的相关度越高, 也表示 j 位置以前与主串部分匹配的字符数越多。

$\text{next}[j]$ 如何求解

- * 当 $j=1$ 时, $\text{next}[j]=0$; // $\text{next}[j]=0$ 表示不进行字符比较
- * 当 $j>1$ 时, $\text{next}[j]$ 的值为: 模式串的位置从1到 $j-1$ 构成的串中, 所出现的, **首尾相同的子串** 的 **最大长度** 加1 // 无首尾相同的子串时 $\text{next}[j]=1$, 此时从头部开始比较

(2) $\text{next}[j]$ 具体怎么求?

计算 $\text{Next}[j]$ 的方法:

• 当 $j=1$ 时, $\text{Next}[j]=0$;

// $\text{Next}[j]=0$ 表示根本不进行字符比较

• 当 $j>1$ 时, $\text{Next}[j]$ 的值为: 模式串的位置从1到 $j-1$ 构成的串中所出现的首尾相同的子串的最大长度加1。

无首尾相同的子串时 $\text{Next}[j]$ 的值为1。

// $\text{Next}[j]=1$ 表示从模式串头部开始进行字符比较

怎样计算模式 T 所有可能的失配点 j 所对应的 $\text{next}[j]$?

例题: 求 $\text{next}[j]$

例：

模式串 T:	a b a a b c a c
可能失配位 j:	1 2 3 4 5 6 7 8
新匹配位 k=next[j]:	0 1 1 2 2 3 1 2

next[j]与s无关，
可以预先计算

刚才已归纳：
$$\text{next}[j] = \begin{cases} 0 & \text{当 } j=1 \text{ 时} \\ \max\{k \mid 1 < k < j \text{ 且 } 'T_1 \dots T_{k-1}' = 'T_{j-(k-1)} \dots T_{j-1}' \} \\ 1 & \text{其他情况} \end{cases}$$

讨论：

- j=1时, next[j]=0; // 属于“j=1”情况
j=2时, next[j]=1; // 找不到1<k<j的k, 属于“其他情况”
j=3时, k={2}, 只需查看 'T₁'='T₂' 成立否, No则属于其他情况
j=4时, k={2, 3}, 要查看 'T₁'='T₃' 及 'T₁T₂'='T₂T₃' 是否成立
j=5时, k={2, 3, 4}, 要查看 'T₁'='T₄', 'T₁T₂'='T₃T₄' 和
T₁T₂T₃'='T₂T₃T₄'
以此类推, 可得后续next[j]值。

从两头往中间比较

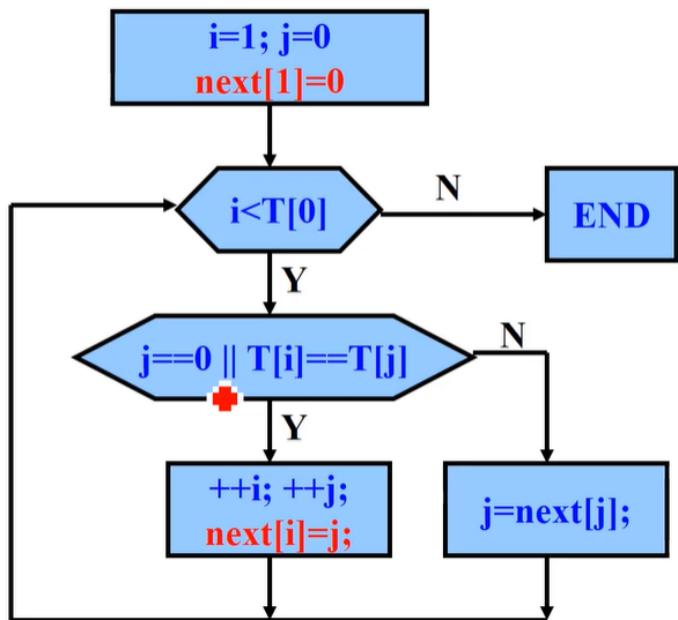
激活 W
转到“设置”
23

！！编程求解next[j]

- 递推方式

```
void get_next(SString T, int &next[]) {
    // 求模式串T的next函数并放入next中
    i = 1;
    j = 0;
    next[1] = 0;
    while (i < T[0]) { // T[0]为T串的长度
        if (j == 0 || T[i] == T[j])
        {
            ++i, ++j;
            next[i] = j;
            continue;
        }
        j = next[j];
    }
}
```

求解next[j]流程图（递推）



KMP算法的实现

```
Int Index_KMP(SString S, SString T, int pos)
{
    //KMP的Index算法
    i = pos;
    j = 1;
    while(i < S[0] && j < T[0])
    {
        if(S[i] == T[j] || j==0)
        { // 不失配则继续比较后续字符
            ++i, ++j;
            continue;
        }
        j = next[j]; // 特点: S的i指针不回溯, 而且从T的k位置开始匹配
    }
    if(j > T[0]) return i - T[0]; // i - j + 1; 也是可以的
    return 0;
} // Index_KMP
```

KMP的时间复杂度分析

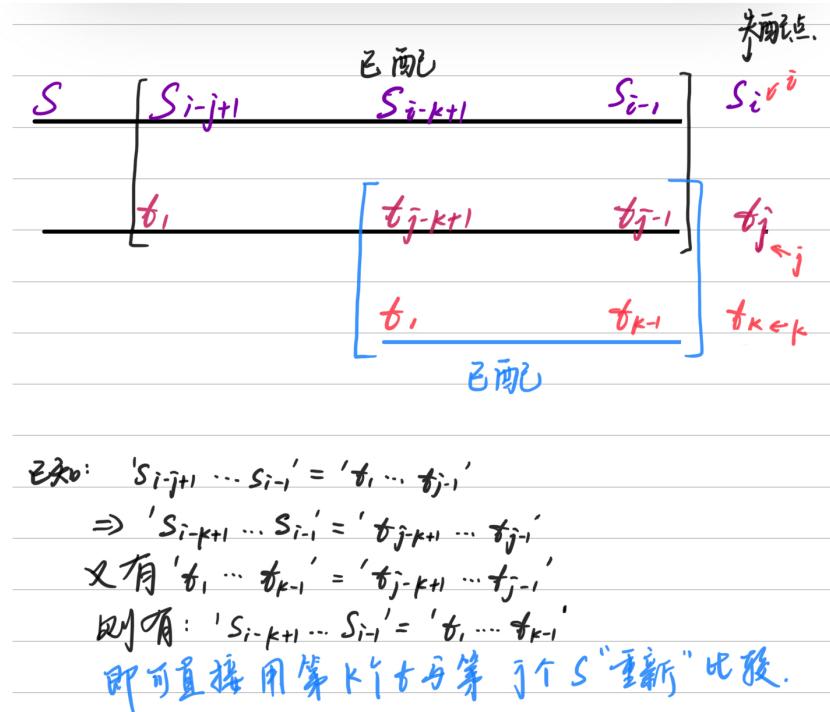
- 最恶劣的情况
- 由于指针 i 无须回溯，比较次数仅为 n ，算 $next[j]$ 时所用的比较次数 m ，比较总次数为 $n+m = O(n+m)$
- BF在一般情况下，时间复杂度也近似于 $O(n+m)$ ，仍被广泛使用

KMP算法的用途

- 主串指针 i 不必回溯，从外存输入文件时，可以边读入，边查找，流水作业

清华版 KMP

大体思路、示意图：



定义 next 函数

$$next[j] = \begin{cases} 0 & \text{当 } j=1 \text{ 时} \\ \max\{k \mid 1 < k < j \\ \text{且 } 'p_1 p_2 \cdots p_{k-1}' = 'p_{j-k+1} \cdots p_{j-1}'\} & \\ 1 & \text{其它情况} \end{cases}$$

一些含义

```
* next[7] = 2;  
    则 't_6 t_7' = 't_1 t_2'  
* next[8] = 3;  
    则 't_6 t_7 t_8' = 't_1 t_2 t_3'
```

求解 next 函数的思路

- 递推过程

```
* 已知: next[1] = 0;  
* 假设: next[j] = k;  
    且 T[j] = T[k];  
* 则: next[j+1] = k+1;  
* 若: T[j] != T[k];  
* 则: 使 j = next[j], 直到 T[j] = T[k], 相当于模板串的子串和自己进行匹配;
```

求 *next* 函数值的过程是一个
递推过程，分析如下：

已知: $\text{next}[1] = 0$;

假设: $\text{next}[j] = k$; 又 $T[j] = T[k]$

则: $\text{next}[j+1] = k+1$

若: $T[j] \neq T[k]$

则需往前回溯，检查 $T[j] = T[?]$

NEXT_VAL: next 的改良

还有一种特殊情况需要考虑：

例如：

$S = 'aaabaaaabaaabaaaabaaab'$

$T = 'aaaab'$

$\text{next}[j] = 01234$

```
next[j] = k; 当 T[j] != T[k] 时;  
next[j] = next[k]; 当 T[j] == T[k] 时;
```

* 说明: next 是用来当 $T[j] \neq S[i]$ 时，重新给 j 赋值的
所以当 $T[j] = T[\text{next}[j]=k]$ 时，浪费了一次对比时间
这样可以有效地提高效率

```
void get_next_new(SSString T, int &next[]){
```

```
//求模式串T的next函数并放入next中，求的是next_new
i = 1;
j = 0;
next[1]=0;
while(i < T[0]){//T[0]为T串的长度
    if(j==0 || T[i]==T[j])
    {
        ++i; ++j;
        if(T[j] == T[i]) next[i] = next[j];
        else next[i]=j;
    }
    else j=next[j];
}
}
```

一些关于next[]的感想

1. 从效率的角度上来看，next越小，移动越快，效率越高
2. 从保险、不漏的角度看，next越大，越保险