Ch 4　字符串

- **串(string)的基本概念**

串由**n(n≥0)**个字符组成的有限*序列*, 记作 **S = “$c_1c_2\cdots c_n$”**
其中: **S—**串名, **n—**串长, **n = 0** 时是空串**(empty string).**
注: ①空串和由一个或多个空格字符组成的空白串**(blank string)**不同.
②串是一种线性表, 只是组成串的每个数据元素均是一个字符.

串中任意个连续**(consecutive)**字符组成的子序列称为该串的子串**(substring),**
包含子串的串相应地称为主串.
注**1:** ①空串是任意串的子串, 任意串是其自身的子串.
空串是任意串的子序列, 任意串是其自身的子序列.
②注意区别子串与子序列**(subsequence).** A subsequence is a sequence that appears in the same relative order in the string, but not necessarily consecutive.
例如 **“abc”, “abg”, “bdf”, “aeg”, ‘”acefg”, ...** 是字符串**“abcdefg”**的子序列.
③ 串的前缀**(prefix)** & 后缀**(suffix)　vs.**
串的真前缀**(proper prefix)** & 真后缀**(proper suffix)**

注**2:** 在**C/C++**中,
①空字符**‘\0’(ascii** 值为**0)**不是空格字符**(space, ascii32),**
且空字符不算作字符串的长度
②一个字符和含一个字符的串不同, **‘x’ vs. “x”(**含两个字符**‘x’**和**‘\0’)**

- 串的的存储结构

串的顺序存储: 有两种方式

- 用一维数组(array)的静态存储结构
- 用堆(heap)结构动态存储方式

存储空间的大小在编译时刻就已确定

串的顺序存储 -- ①用一维数组表示的静态分配的顺序存储结构

```
#define MaxLength 256
typedef char SeqString[MaxLength];
//SeqString是一个可容纳255个字符的顺序存储结构类型.
//在说明字符数组时必须比它要存放的最长字符串多1个字符,
//因需留一个位置来存放空字符即终结符
```

若不设终结符'\0', 上述数据结构可改为:

```
#define MaxLength 256
typedef struct string {
   char ch[MaxLength];
   int length;  //记录当前串的实际长度
} SeqString;  //SeqString是一个可容纳256个字符的顺序存储结构类型
```

- 串的的存储结构

串的顺序存储 -- ②运用堆动态地进行顺序存储分配

```
typedef struct string {
   char *ch;  //若是非空串, 则按实际串长分配存储空间, 否则ch = Null
   int length; //记录当前串的实际长度
} HString;  //HString是串的一种动态分配的顺序存储结构
```

Hstring本质上是数组和指针的结合. 对上述结构还可以再简单定义成:

```
typedef char *String;  //String实际上就是一个字符数组类型[数组指针]
```

注: C/C++用堆来对存储空间进行动态的管理. 即字符串被定义为一个以空字符(Null character)结尾的字符数组, 其实就是一个数组指针.事实上, C/C++的类库<string.h>中所有对串的运算均是使用以此类型定义的串. 例如:

```
char str[80] = "This is an example.\n", *p, *q;  //String p, q;
 { p = str;  //相当于 p = &str[0], 即把字符数组str的第一个字符的地址赋给指针p.
   p[3] = 's'; //assign using index
   str[5] = *(p+2);  //assign using pointer arithmetic
   strcpy(q, p);  //实现了串复制 ~ while ( ( *q++ = *p++ ) != '\0' );
}
```

又例如:

```
{ char str[ ] = "HELLO!  WORLD.";
  char *p = str;
  printf("%s", p + p[3] - p[1]); } //?输出结果
```

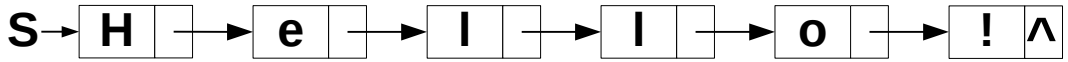
串的的存储结构

串的链式存储

```
typedef struct node {
   char ch;
   struct node *next;
} Node, *LinkedString;
```

```
#define NodeSize 4
typedef struct node {
   char ch[NodeSize];
   struct node *next;
} Node, *LinkedString;
```

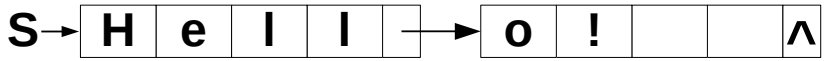
设 LinkedString S; //Node *S;



//串S的内容: Hello!

$$\text{存储密度(storage density)} = \frac{\text{结点(node)数据本身所占的存储空间}}{\text{结点结构所占的存储空间(即还包含指针域)}}$$

显然, 顺序结构的存储密度为1, 而链式结构则小于1
这里串的链式存储, 其存储空间利用率太低(若指针占用4个字节, 则20%)



虽然提高了存储空间利用率, 但在做插入或删除运算时引起的大量字符移动给算法带来了很大不便.
--> 串一般不采用链式存储结构.

串的基本运算

大多高级语言中有现成的库函数支持串的基本运算.
以C/C++语言为例, 从<string.h>选用几个最常用的串操作函数:

基本运算	*函数名称*	*功能*
copy	strcpy(s1, s2)	将s2拷贝到s1
concatenate	strcat(s1, s2)	将s2连接到s1的末尾
get the length	strlen(s1)	返回s1的长度
compare	strcmp(s1, s2)	若s1 == s2, 则返回值为0
		若s1 < s2, 则返回值小于0
		若s1 > s2, 则返回值大于0
index-字符定位	strchr(s1, ch)	返回一个指针指向s1中第一次出现字符ch的位置
index-子串定位 (串配匹/模式配匹)	strstr(s1, s2)	返回一个指针指向s1中第一次出现串s2的位置

如何运用串的存储结构来实现这些基本运算即算法.

- **子串定位(或称为字符串匹配/模式匹配(Pattern Match))的算法**

模式匹配的定义　假设t(text/正文串或主串、目标串)和p(pattern/模式)是两个给定的串(且len(t)≥len(p)), 在t中寻找与p相同子串的过程称为~. 若找到, 则匹配成功, 返回串p在串t中第一次出现的位置; 否则匹配失败, 返回0.

串的匹配有两种算法 { 简单算法 / 朴素算法(Naive pattern searching algorithm)
KMP算法(Knuth Morris Pratt Pattern Searching)

模式匹配简单算法的基本思路

对i = 1, 2, ⋯ , n-m+1, 依次进行以下匹配步骤(最多进行n-m+1趟), 其中n和m分别为串t和串p的长度(m≤n): 用p[1], p[2], ⋯ , p[m]依次和t[i], t[i+1], ⋯ , t[i+m-1]进行比较, 如果p[1] = t[i], p[2] = t[i+1], ⋯ , p[m] = t[i+m-1], 则匹配成功(此次的位移称为有效位移), 返回i值(i即为有效位移值, 也就是从此位置开始的匹配成功), 算法结束; 否则, 一定存在着某个整数k(1≤k≤m), 使得p[k] ≠ t[i+k-1], 这时则中止该趟的匹配(此次位移称为无效位移), 而执行下一趟的匹配步骤, 直至i > n-m+1.

模式匹配简单算法的算法描述

假设算法描述所用的串以静态的顺序存储结构表示

```
P73 //串的顺序存储表示
#define MaxLen 256
typedef unsigned char SString[MaxLen+1];
//0#存放串的长度
```

或

```
#define MaxLen 256
typedef struct string {
   char ch[MaxLen];
   int length;
} SeqString;
```

- 子串定位(或称为字符串匹配/模式匹配(Pattern Match))的算法

模式匹配简单算法的算法描述 P79/A.4.5

一般说来, 在执行简单匹配算法的过程中, 若p[j]≠t[i](1≤i≤n, 1≤j≤m), 则下一趟匹配一定是从模式p的第1个(即j=1)字符开始与正文t中第i-j+2个(即i=i-(j-1)+1=i-j+2)字符开始依次进行比较. 即:

position(index)

i → t[1] t[2] … t[i-j+1] t[i-j+2] … t[i-1] t[i] …

‖ ‖ ‖ ≠

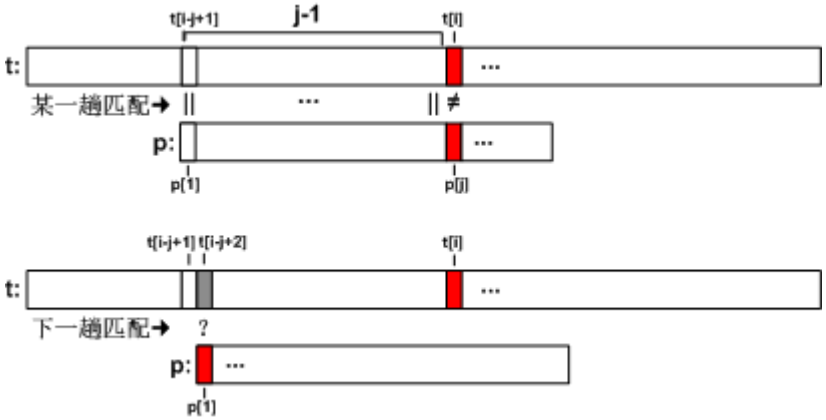
j → p[1] p[2] … p[j-1] p[j] …

失配, 下一趟 ↴

i → t[1] t[2] … t[i-j+1] t[i-j+2] t[i-j+3] … t[i-1] t[i] t[i+1] …

↕ ↕ ↕ ↕ ↕ ↕

j → p[1] p[2] … p[j-2] p[j-1] p[j] …



- 子串定位(或称为字符串匹配/模式匹配(Pattern Match))的算法

模式匹配简单算法的算法描述 **P79/A.4.5**

```
int NaiveIndex(SString t, SString p, int pos)
{ i = pos; j = 1;
  while ( i <= (t[0]-p[0]+1) && j <= p[0] ) {
       if ( p[j++] == t[i++] );
       else { i = i-j+2; j = 1; } }  //失配后准备进行下一趟
   if ( j > p[0] ) return i - p[0]; //有效位移值, 即从该位置起匹配成功
   else return 0;  //匹配失败
} //其中t[0]、p[0]分别为正文t、模式p的长度
```

该算法的(最坏)时间复杂性:O(m×(n-m+1)) = O(m×n) (若n>>m)

- 子串定位(或称为字符串匹配/模式匹配(Pattern Match))的算法

KMP对简单匹配算法的改进 --- KMP算法

例1: 设 t = "A**BAAAAB**AAAAAAAA"　p = "**BAAAAAA**A", **i = 2、j = 1**
某一趟匹配至第**7**个字符时失配**(dismatch)**, 即 **p[6] ≠ t[7]**, 这时**i = 7、j = 6**

对**KMP**算法而言, 下一趟匹配时, 只需从 **i = 7、j = 1**位置处开始匹配,
即**p[1] ? t[7]:** t = "ABAAAA**B**AAAAAAAA"　p = "**B**AAAAAAA".

例2: 设 t = "**AAAA**BAAABA"　p = "**AAAB**", **i = 1、j = 1**
第一趟匹配至第**4**个字符时失配, 即 **p[4] ≠ t[4]**, 这时**i = 4、j = 4**

对**KMP**算法而言, 下一趟匹配时, 只需从 **i = 4、j = 3**位置处开始匹配,
即**p[3] ? t[4]:** t = "A**AA**BAAABA"　p = "**AA**B"

可见, **KMP**算法进行下一趟匹配时, **i**没有"回溯"(即**i**不变), **j**也不一定从**1**再开始.
这就是**KMP**算法的改进之处.

其原理是: 失配时, 利用该趟已得到的"部分匹配"的结果将模式**p**向右"滑动"尽可能远的一段距离后继续进行下一趟匹配**(P80**例图**)**.

问题提出: 若**p[j] ≠ t[i]**, 如何去找到模式**p**中的位置**k**去继续匹配**? (**即**p[k] ? t[i])**

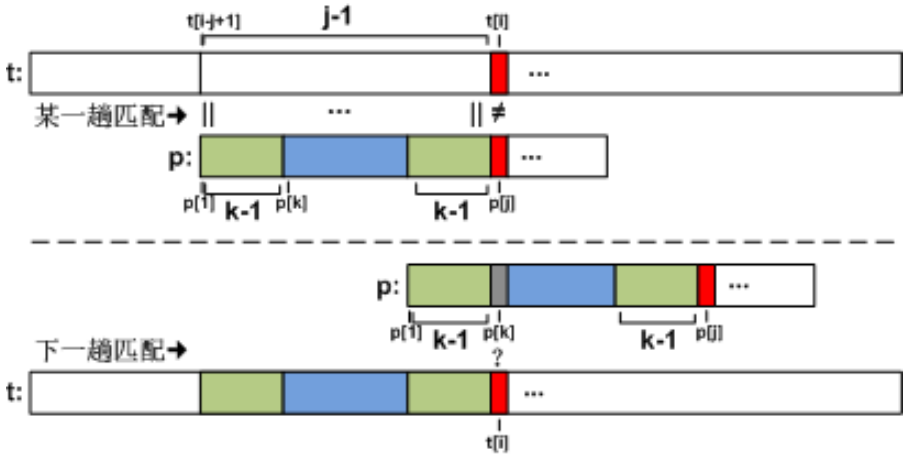
- 子串定位(或称为字符串匹配/模式匹配(Pattern Match))的算法

KMP对简单匹配算法的改进 --- KMP算法

设p[j]≠t[i], 若存在着一个整数k(1<k<j), 使得在模式p中开头*k-1*个字符依次与p[j]之前*k-1*个字符相同(最长公共前后缀).

t[1] t[2] … t[i-j+1] … t[i-k+1] … t[i-1] t[i] t[i+1] …
‖ ‖ ‖ ≠
p[1] … p[j-k+1] … p[j-1] p[j] p[j+1] …
‖ ‖ ?
p[1] … p[k-1] p[k] p[k+1] …

因此当p[j]≠t[i]时, 只要从模式p中的p[k]与正文t中的t[i]开始依次进行比较. i没有回溯, 同时也省去前面的k-1次比较.



- 子串定位(或称为字符串匹配/模式匹配(Pattern Match))的算法

KMP对简单匹配算法的改进 --- KMP算法

这里的k称为模式p[j]的失配链接值或模式p[j]的next函数值(即: next[j]=k), 其作用是: 当模式p中的第j个字符与正文t中的相应的字符(假设第i个) “失配”时(p[j] ≠ t[i]), 在模式p中需重新和该字符(t[i])进行比较的字符的位置(p[k])(即p[k]?t[i]).

KMP算法是建立在模式p的next函数值基础上的, 因此该算法的关键是如何计算出模式p中每个字符的失配链接值k(=next[j])!

由上述对k值的分析可知, 此函数值只依赖于模式p本身, 而与正文t无关.
P81给出了next函数的定义:

$$next[j] = \begin{cases} 0 & \text{当j=1时(约定: 模式p的首字符的失配链接值为0, next[1]=0)} \\ Max\{ k \mid 1<k<j \text{ 且 } `p_1 \cdots p_{k-1}' = `p_{j-k+1} \cdots p_{j-1}' \} & \\ 1 & \text{其它情形(如: next[2]=1; j>2时,不满足上述条件)} \end{cases}$$

例如:

j	1	2	3	4	5	6	7	8
模式p	a	b	a	a	b	c	a	c
next[j](=k)	0	1	1	2	2	3	1	2

又例如:

j	1	2	3	4	5	6	7	8
DNA模式p	G	C	A	G	A	G	A	G
KMPnext[j]	0	1	1	1	2	1	2	1

- 子串定位(或称为字符串匹配/模式匹配(Pattern Match))的算法

KMP对简单匹配算法的改进 --- KMP算法

依据next[j]的定义, KMP算法基本思路:

P82: 由next[j]的定义可知 next[1]=0, next[2]=1

设next[j]=k, 即表明在模式p中存在如下关系: '$p_1 \cdots p_{k-1}$' = '$p_{j-k+1} \cdots p_{j-1}$'

那么如何计算下一个字符的失配链接值, 即next[j+1]=?

- 若$p_k=p_j$, 则有 '$p_1 \cdots p_{k-1}\ p_k$' = '$p_{j-k+1} \cdots p_{j-1}\ p_j$'

  即next[j+1] = k+1 ➔ next[j+1] = next[j]+1 (1)

- 若$p_k \neq p_j$, 则有 '$p_1 \cdots p_{k-1}\ p_k$' ≠ '$p_{j-k+1} \cdots p_{j-1}\ p_j$'

  此时把求next函数值的问题又可看成是一个模式匹配的问题(This is tricky), 即当前模式p既是正文串又是模式串(也就是看成, 模式的p_k与"正文"的p_j失配后的下一趟匹配). 这时将模式p移动至第next[k](设next[k]=k')个字符和"正文" p中的第j个字符相比较, 若$p_j=p_{k'}$, 则说明在"正文" p中第j+1个字符之前存在一个长度为k'(即next[k])的最长子串和模式p从头字符起长度为k'的子串相等: '$p_1 \cdots p_{k'}$' = '$p_{j-k'+1} \cdots p_j$'

  即next[j+1] = k'+1 ➔ next[j+1] = next[k]+1 (2)

- 同理, 若$p_j \neq p_{k'}$, 则将模式p继续移动至第next[k'](=k")个字符和"正文" 的p_j相比较(即p_j?$p_{k''}$), 如此类推, 直至p_j和模式中某个字符匹配成功或者不存在任何k'(即k<1)满足等式: '$p_1 \cdots p_{k'}$' = '$p_{j-k+1} \cdots p_j$' ➔ next[j+1] = 1 (3)

- 子串定位(或称为字符串匹配/模式匹配(Pattern Match))的算法

KMP对简单匹配算法的改进 --- KMP算法

几个计算模式串next函数值的例子: 已知next[1]~next[j](↓(x)-情况), 计算next[j+1]

(1)
1. ‘aaaaaa’
012345

(1)(2)(1)(2)
2. ‘abcaabac’
011122323

(2)(1)(3)
3. ‘abaabcac’
01122312

(3)
4. ‘ababacb’
0112341

据此, P83/A.4.7给出了计算函数值next[j]的算法:

```
// i指示“正文串”, j指示模式串.
// 根据计算next数组的算法思路, 从第2个字符起,依次算出“正文串”中每个字符的失配链接值
{ next[1] = 0; i = 1; j = 0;
   while ( i < p[0] ) { // 通过模式串来匹配“正文串”, 找当前“正文串”的最长公共前后缀
      if ( j == 0 || p[i] == p[j] ) { i++; j++; next[i] = j; }
      else j = next[j];
    }
} // 时间复杂度O(m)
```

- 子串定位(或称为字符串匹配/模式匹配(Pattern Match))的算法

KMP对简单匹配算法的改进 --- KMP算法

利用模式串next函数值进行匹配的算法(KMP算法): P82/A.4.6

```
{ i = pos; j = 1;
  while ( i <= t[0] && j <= p[0] ) {
      if ( j = 0 || p[j] == t[i] ) { i++; j++; };
      else j = next[j]; } //失配时, 正文 i不变、而模式j向右滑动至k即next[j]
                          //注意: 当next[j]等于0(即j=0)时, 则正文右移1位(即i++)、模式j置为1(即j++)
   if ( j > p[0] ) return i - p[0];
   else return 0;
} //KMP算法时间复杂度为: O(n+m), 其中O(m)为计算模式串next[j]时间开销
```

- 串操作应用举例

  - 文本编辑 **P84~85**

文本编辑器中包括了大量的串插入、串删除、串查找等操作 ➔ 实习题

  - 建立词汇索引表 **P86~89**

目标: 实际应用中, 因很多内容相似的书籍其书名不一定相同, 一种较好的解决方法是根据书名建立“关键词索引表”.

书号	*书名*
005	Computer Data Structures
010	Introduction to Data Structures
023	Fundamentals of Data Structures
034	The Design and Analysis of Computer Algorithms
050	Introduction to Numerical Analysis
067	Numerical Analysis

书目文件↴, 关键词索引表➔

关键词	*书号索引*
algorithms	034
analysis	034, 050, 067
computer	005, 034
data	005, 010, 023
design	034
fundamentals	023
introduction	010, 050
numerical	050, 067
structures	005, 010, 023

建表一般思路: ①从书目文件中读入一个书目串
②从书目串中提取除常用词外的所有关键词(a/an/the/of/for/to…)
③对每一关键词在索引表中进行查找, 若无则插入该索引项(关键词+书号), 若有则在该索引项中插入其书号

- 串操作应用举例
  - 建立词汇索引表

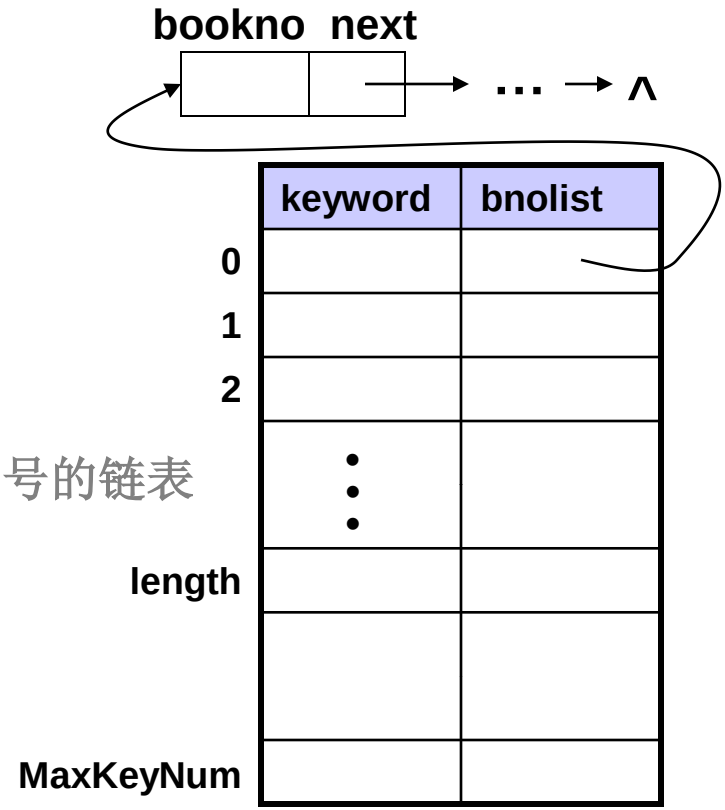
算法实现: 这里仅给出关键词索引表的数据结构, 建立该表的各项操作的具体描述参见P84~89.

```
#define MaxKeywordNum 1000  //关键词索引表的最大容量
typedef char *BookNo  //定义书号类型

typedef struct bnonode {
   BookNo  bookno;  //书号
   struct bnonode *next;
} BNoNode;  //存放一个书号的结点

typedef struct indexentry {
   char *keyword;   //关键词
   BNoNode *bnolist;  //存放具有同一关键词的书号的链表
} IndexEntry;  //关键词索引表的一个索引项

typedef struct indexlist {
   IndexEntry entry[MaxKeywordNum];
   int length;  //关键词索引表的实际项数
} IndexList;  //关键词索引表
```



Exercises (After class):

1. 利用C/C++中有关串操作的函数写一算法: void StringDelete(char *s, int i, int m) 删去串s中从位置i开始的连续 m个字符. 若i≥strlen(s), 则没有字符被删除; 若 i+m≥strlen(s), 则将s中从位置i开始直至末尾的字符均删去.
2. 若S和T是用结点大小为1的单链表存储的两个串, 设计一个算法找出S中第一个不在T中出现的字符.
3. 说明在改进的模式匹配算法(KMP)中, next数组的作用? 并求模式 “ABCABDABEABCABDABF”的相应的next[j].
4. 设正文t=“abcaabbabcabaacbacba”, 模式p=“abcabaa”.

   1) 计算模式p的next函数值;

   2) 不写算法, 只写出利用KMP算法进行模式匹配时每一趟的匹配过程.