

DS—第四章

串-- String

内容回顾

- 栈的定义、ADT定义、特点
- 栈的表示和实现
- 栈的应用
- 栈和递归的关系
- 队列的定义、ADT定义、特点
- 队列的表示和实现
- 队列的应用



串的概念、ADT定义

串的定长表示、基本操作实现

串的堆分配表示、基本操作实现

串的块链存储及优缺点

串的模式匹配算法



串的概念、ADT定义

串的定长表示、基本操作实现

串的堆分配表示、基本操作实现

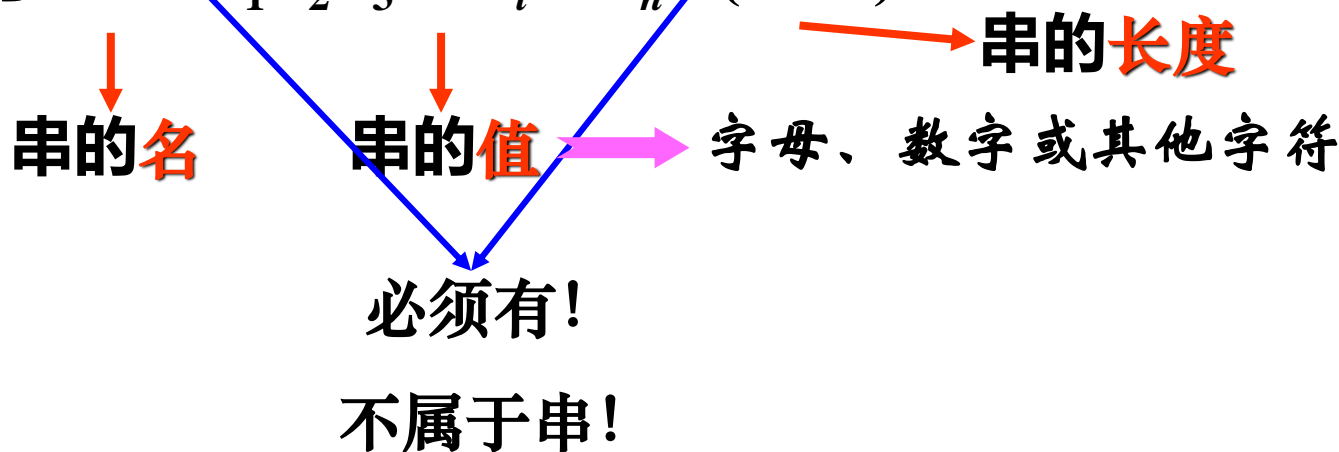
串的块链存储及优缺点

串的模式匹配算法

● 基本概念

串（字符串）：是由 0 个或多个字符组成的有限序列。

通常记为： $s = 'a_1 a_2 a_3 \dots a_i \dots a_n'$ ($n \geq 0$)。



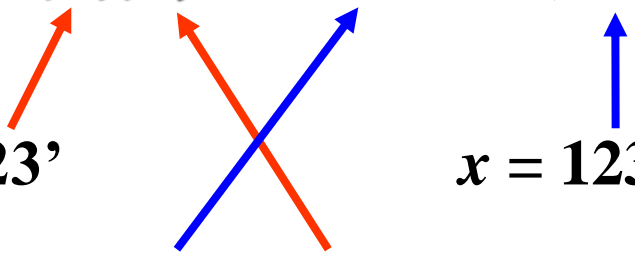
作用：避免**字符串**与**变量名**或**数的常量**混淆。

作用：避免**字符串**与**变量名**或**数的常量**混淆。

例： $x = '123'$

$x = 123$

$test = 'test'$



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。

主串：包含子串的串。

位置：字符在序列中的序号。

子串在主串中的位置：子串的**首字符**在主串中的位置。

例：S='JINAN'

S1='' S2='NA' S3='JINAN' —— 均为 S 的子串。

在 S 中的位置：3

在 S 中的位置：1

空串是任意串的子串，任意串是其自身的子串。

S4='JAN' —— 非 S 的子串

(非串 S 中的连续字符所组成)。

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。

例：在串中查找某个子串；

在串的某个位置上插入一个子串；

删除一个子串等。

串的抽象数据类型的定义

ADT String {

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

数据关系: $R_1 = \{ \langle a_{i-1}, a_i \rangle \mid a_{i-1}, a_i \in D, i = 2, \dots, n \}$

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

操作结果：串 S

StrEmpty (S)

初始条件：串 S

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 。

“串值大小”是按“词典次序”
进行比较的，如：

$\text{StrCompare}(\text{"data"}, \text{"stru"}) < 0$

$\text{StrCompare}(\text{"cat"}, \text{"case"}) > 0$

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， $1 \leq \text{pos} \leq \text{StrLength}(S)$

且 $0 \leq \text{len} \leq \text{StrLength}(S) - \text{pos} + 1$ 。

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
 $1 \leq \text{pos} \leq \text{S}$

操作结果：若主串 S
 则返回它
 第一次出

假设

S="abcacabcaca",

T="abca"

V="ab",

则置换之后的

S="abcabca",

而不是 "abbcaca"。

子串，
 符之后
 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。

StrInsert (&S, pos, T)

初始条件：串 S 和 T 存在， $1 \leq \text{pos} \leq \text{StrLength}(S) + 1$ 。

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， $1 \leq \text{pos} \leq \text{StrLength}(S) - \text{len} + 1$ 。

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

串类型的最小操作子集
**这些操作不可能利用
其他串操作来实现。**

串赋值 StrAssign

串联接 Concat

求串长 StrLength

串比较 StrCompare

求子串 SubString

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
```



串的概念、ADT定义

串的定长表示、基本操作实现 >>>

串的堆分配表示、基本操作实现

串的块链存储及优缺点

串的模式匹配算法

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**。

4.2.1 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，它的长度是不变的。

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。

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**”。

串长的表示方法

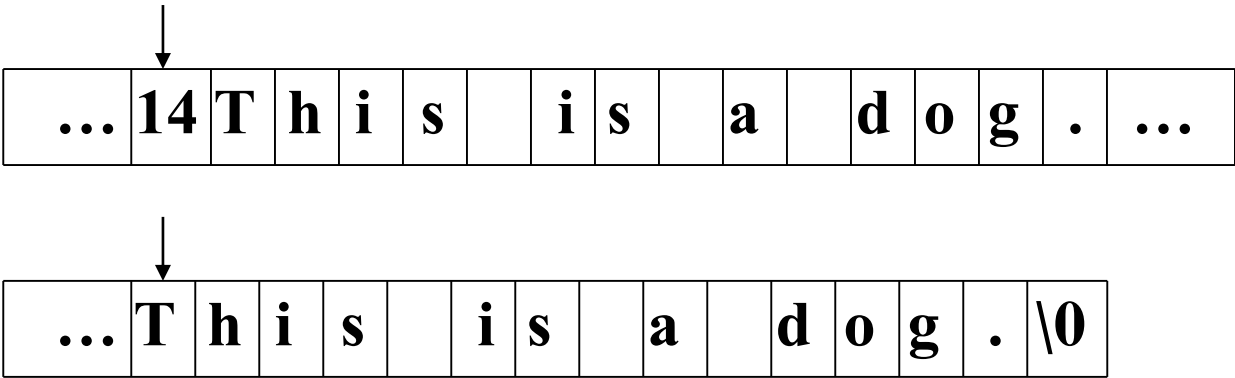
显式

在串的存贮区首地址
显式地记录串的长度。
如：PASCAL 语言。

隐式

在串之后加结束标志。
如：C 使用 “\0”。

例：串 “This is a dog.” 的长度的表示方法：



● 定长顺序存储表示时串的操作的实现

1、串联接 $\text{Concat}(\&T, S1, S2)$

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”。

串 T 值	$S1[0]+S2[0] \leq \text{MAXSTRLEN}$	结果正确
	$S1[0] < \text{MAXSTRLEN}$	结果 $S2$ 被“截断”
	$S1[0]+S2[0] > \text{MAXSTRLEN}$	
	$S1[0] = \text{MAXSTRLEN}$	结果 $T=S1$

串联接 Concat 算法描述

```
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]; 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。

求子串 SubString 算法描述

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

● 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。



串的概念、ADT定义

串的定长表示、基本操作实现

串的堆分配表示、基本操作实现



串的块链存储及优缺点

串的模式匹配算法

4.2.2 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。

串插入操作 $\text{StrInsert}(\&S, \text{pos}, T)$ 的**实现算法**为：

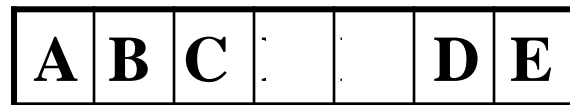
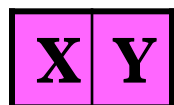
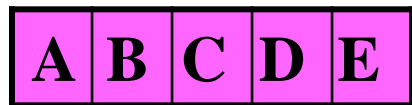
- 1、为串 S 重新分配大小等于串 S 和串 T 长度之和的存储空间；
- 2、进行串值的复制。

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

```
{ 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; 例 : S='ABCDE' T='XY'

//StrInsert



↑
S.ch

↑
i

Pos -1

```
typedef struct{
  char *ch;
  int length;
} HString;
```

串联接 Concat 算法描述

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

{ if (T.ch) free(T.ch); // 释放旧空间

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

if (!T.ch) 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

求子串 SubString 算法描述

```
Status SubString(HString &Sub, HString S, int pos, int len)
{ if (pos < 1 || pos > S.length || len < 0 || len > S.length-pos+1)
    return ERROR;
  if (Sub.ch) free (Sub.ch);          // 释放旧空间
  if (!len)
    { Sub.ch = NULL; Sub.length = 0; } // 空子串
  else
    { Sub.ch = (char *)malloc(len*sizeof(char));
      Sub.ch[0...len-1] = S[pos-1...pos+len-2];
      Sub.length = len;
    } // 完整子串
  return OK;
} // SubString
```

串复制 Strcopy 算法描述：

```
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;
} Strcopy
```


串赋值 StrAssign 算法描述：

```
status StrAssign(HString &T, char *chars)
{ if (T.ch) free(T.ch); //释放 T 原有的空间
  for (i=0 , c=chars; c; ++i, ++c); //求 chars 的长度 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
```

串比较 StrCompare 算法描述:

```
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;
```

```
}// StrCompare
```

清空串 ClearString 算法描述：

```
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;
```

```
} ClearString
```

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。



串的概念、ADT定义

串的定长表示、基本操作实现

串的堆分配表示、基本操作实现

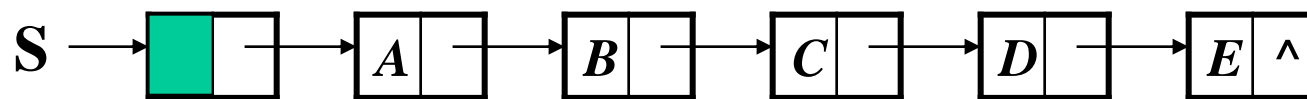
串的块链存储及优缺点

串的模式匹配算法

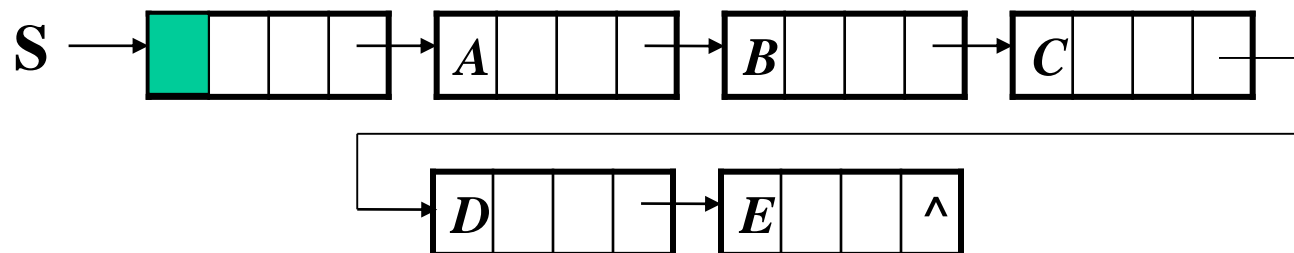


4.2.3 串的块链存储表示

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**。



优点：便于插入和删除 **缺点：**空间利用率低



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**。



$$\text{存储密度} = \frac{\text{数据元素所占存储位}}{\text{实际分配的存储位}}$$

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;
```



串的概念、ADT定义

串的定长表示、基本操作实现

串的堆分配表示、基本操作实现

串的块链存储及优缺点

串的模式匹配算法



4.3 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。

用函数 $\text{Index}(S, T, \text{pos})$ 实现。

在串匹配中，将主串 S 称为**目标(串)**，
子串 T 称为**模式(串)**。

如果在主串 S 中能够找到子串 T ，则称**匹配成功**，
返回第一个和子串 T 中第一个字符相等的字符在**主串**
 S 中的**序号**；否则，称**匹配失败**，**返回 0**。

例如：

主串： ss='abccababdabcdef';

子串： tt1='abc'; **子串：** tt2='abd';

Index(ss, tt1, 2) ; Index(ss, tt1, 5) ; Index(ss, tt2, 8) ;

4

10

0

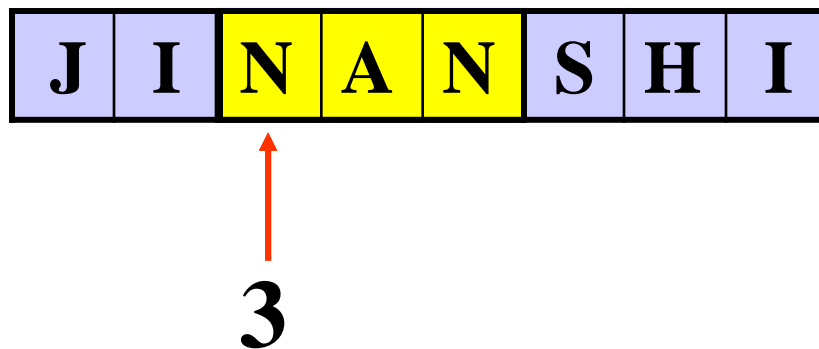
模式匹配是各种处理系统中最重要的操作之一，也是一个比较复杂的串操作。模式匹配的算法不同，效率将有很大差别。同一算法应用不同，效率亦有很大差别。

- 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。

例：S = 'JINANSHI' , T = 'NAN'。



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
```

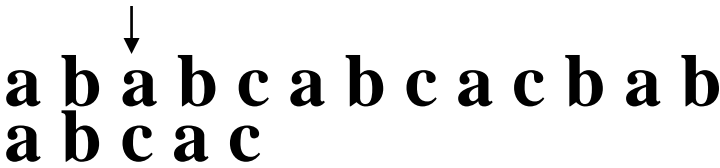
朴素的模式匹配算法评价

- 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。
- **因此我们学习另一种高效的匹配算法----KMP算法。**

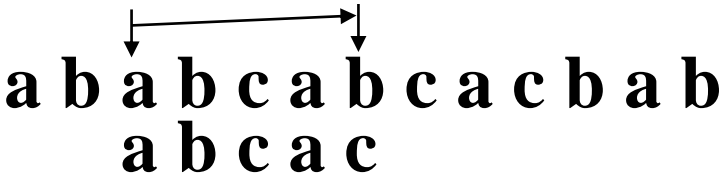
● KMP算法

例子：

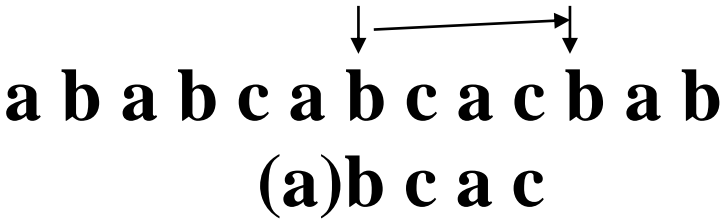
第一趟匹配



第二趟匹配



第三趟匹配



当主串中第 i 个字符与模式串中第 j 个字符不等时，仅需将模式串向右滑动至第 k 个字符和主串中第 i 个字符比较即可。

失配时的位置：

$$s_1 s_2 \cdots s_{i-k+1} s_{i-k+2} \cdots s_{i-1} \mathbf{s_i} s_{i+1} \cdots$$

$$p_1 p_2 p_{k-1} p_k \cdots p_{j-1} p_j$$

应滑动的位置：

$$s_1 s_2 \cdots s_{i-k+1} s_{i-k+2} \cdots s_{i-1} \mathbf{s_i} s_{i+1} \cdots$$

$$p_1 p_2 \cdots p_{k-1} p_k \cdots p_{j-1} p_j$$

问题：到底是模式串的哪个字符与主串的第 i 个字符比较

---算法的优化过程

$$“s_{i-k+1}s_{i-k+2}\cdots s_{i-1}” = “p_{j-k+1}\cdots p_{j-2}p_{j-1}”$$

当在

$$\begin{cases} s_i \neq p_j \\ “s_{i-j+1}s_{i-j+2}\cdots s_{i-1}” = “p_1p_2\cdots p_{j-1}” \end{cases}$$

要能立即确定模式**右移的位数**，即确定 s_i （ i 指针不回溯）应与模式串的哪一个字符继续进行比较？

假设此时 s_i 应与模式串的第 k （ $k < j$ ）个字符比较，则应有如下关系式存在：

$$“s_{i-k+1}s_{i-k+2}\cdots s_{i-1}” = “p_1p_2\cdots p_{k-1}”$$

于是有：

$$“p_1p_2\cdots p_{k-1}” = “p_{j-k+1}\cdots p_{j-2}p_{j-1}”$$

结论：如果子串满足这个关系，则当子串的第 i 个字符与主串的第 j 个字符失配后，直接用子串的第 k 个字符与主串的第 i 个字符比较。

- 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：

$$next[j] = \begin{cases} 0 & \text{当 } j = 1 \text{ 时} \\ \text{Max}\{k \mid 1 < k < j \text{ 且 } 'p_1p_2 \wedge p_{k-1}' = 'p_{j-k+1} \wedge p_{j-1}'\} & \\ 1 & \text{其它情况} \end{cases}$$

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; }           // 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;  
}
```

模式串的next数组的生成？

- ✓ 从前面的讨论可知，next函数值仅取决于模式串本身，而与主串无关。
- ✓ next[j]的值等于在“ $p_1 p_2 \dots p_{k-1} p_k \dots p_{j-1}$ ”这个模式串中，相同的前缀子串和后缀子串的最大长度加1。因此要计算next[j]就要在“ $p_1 p_2 \dots p_{k-1} p_k \dots p_{j-1}$ ”找出前缀和后缀相同的最大子串。这个查找过程实际上仍然是模式匹配，只是匹配的模式与目标在这里是同一个串P。

✓ 求next数组值可采用递推的方法，分析如下：

已知： $\text{next}[1] = 0$ ；

假设： $\text{next}[j] = k$ ，现在要求 $\text{next}[j+1]$ 的值。这时有两种情况：

(1) 若 $p_k = p_j$ ，则表明在模式串中有：

$$p_1 p_2 \cdots p_{k-1} p_k = p_{j-k+1} p_{j-k+2} \cdots p_{j-1} p_j$$

这时有 $\text{next}[j+1] = \text{next}[j] + 1 = k + 1$

(2) 若 $p_k \neq p_j$ ，则表明在模式串中有

$$p_1 p_2 \cdots p_{k-1} p_k \neq p_{j-k+1} p_{j-k+2} \cdots p_{j-1} p_j$$

则需往前回溯，检查 $P[j] = P[?]$

这实际上也是一个匹配的过程，

最终： $\text{next}[j+1] = \text{next}[\text{next} \dots [j]] + 1$ 或 $\text{next}[j+1] = 1$

next数组的生成算法

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

next函数的讨论：

j	1	2	3	4	5
模式串	a	a	a	a	b
next	0	1	2	3	4

主串： *'aaabaaab'*;

当子串中的第四个字符'a'与主串中的第四个字符'b'失配后，如果用子串中的第3个字符'a'继续与主串中的第四个字符'b'比较，将是做无用功。以此类推。

结论： next函数仍有改进的地方。

改进next函数：

当子串中的第 j 个字符与主串中的第 i 个字符失配后，如果有 $\text{next}[j]=k$ 且在子串中有 $p_j=p_k$ ；那么 p_k 肯定也与主串中的第 i 个字符不等，所以，直接让 $\text{next}[j]=\text{next}[k]$ ；直到他们不等或 $\text{next}[j]=0$ 为止。

j	1	2	3	4	5
模式串	a	a	a	a	b
next	0	1	2	3	4
nextval	0	0	0	0	4

主串：*'aaabaaab'*；

当子串中的第四个字符'a'与主串中的第四个字符'b'失配后，直接让 i, j 同时加1。

next数组的生成算法

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

本章小结

- 串的相关概念
- 串的ADT定义
- 串的实现与表示
 - I.定长顺序表示
 - II.堆分配存储表示
 - III.块链存储表示
- 模式匹配算法
 - I.基本匹配算法
 - II.改进匹配算法(KMP)
 - III.模式串的next[]。
 - IV.改进的模式串的nextval[]。