数据结构

深圳技术大学
大数据与互联网学院

第四章 串

4.1 串的类型定义
4.2 串的表示与实现
4.3 串的模式匹配算法
4.4 串操作应用举例

4.1 串的概念

一. 字符串

- 串即字符串，是n(≥0)个字符的有限序列，记作：
  - S = ‘$a_1a_2a_3\cdots a_n$’
  - S 是串名字
  - ‘$a_1a_2a_3\cdots a_n$’是串值
  - a_i 是串中字符
  - n 是串的长度(串中字符的个数)
  - S = “Shen Zhen”,则共有9个字符，长度n是9，其中有3个字符h\e\n三个字符是重复，还有1个空格符

4.1 串的概念

二. 串的术语

- 空串：不含任何字符的串，串长度=0
- 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，例如
  - A='Shen Zhen' B='Zhen' A为主串，B为子串
- 位置：字符在序列中的序号。子串在主串中的位置以子串第一个字符在主串中的位置来表示。
- 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。
- 模式匹配：确定子串在主串中首次出现的位置的运算

4.1 串的概念

二. 串与线性表的关系

- 相同点：
  - 串的逻辑结构和线性表极为相似，它们都是线性结构
  - 串中的每个字符都仅有一个前驱和一个后继
- 区别
  - 串的数据对象约定是字符集，线性表可以是任意类型
  - 线性表的基本操作中，以“单个元素”作为操作对象
  - 串的基本操作中，通常以“串的整体”作为操作对象，例如，查找子串、插入子串等

4.2 串的表示与实现

串是一种特殊的线性表，其存储表示和线性表类似，但又不完全相同。串的存储方式取决于将要对串所进行的操作。串在计算机中有3种表示方式：

- 定长顺序存储表示：将串定义成字符数组，利用串名可以直接访问串值。用这种表示方式，串的存储空间在编译时确定，其大小不能改变。
- 堆分配存储方式：仍然用一组地址连续的存储单元来依次存储串中的字符序列，但串的存储空间是在程序运行时根据串的实际长度动态分配的。
- 块链存储方式：是一种链式存储结构表示。

4.2 串的表示与实现

一. 定长顺序存储表示

- 用一组地址连续的存储单元存储字符序列，如C语言中的字符串定义(以“\0”为串结束标志)

```
char Str[MAXSTRLEN+1];
```

  - 定义了长度为MAXSTRLEN字符存储空间
  - 字符串长度可以是小于MAXSTRLEN的任何值（最长串长度有限制，多余部分将被截断）

4.2 串的表示与实现

一. 定长顺序存储表示

- 串的定义

```
#define MAX_STRLEN  255
typedef  unsigned char  str[MAX_STRLEN+1]  SString;
```

4.2 串的表示与实现

二. 堆分配存储表示

- 在程序执行过程中，动态分配（动态分配）一组地址连续的存储单元存储字符序列
- 堆分配存储结构的串既有顺序存储结构的特点，处理方便, 操作中对串长又没有限制, 更显灵活

4.2 串的表示与实现

二. 堆分配存储表示

- 堆分配的串定义

```
class HString{
    char *ch;     //若非空，按长度分配，否则为NULL
    int length;   // 串的长度
      ... //其他成员
};
```

4.2 串的表示与实现

二. 堆分配存储表示

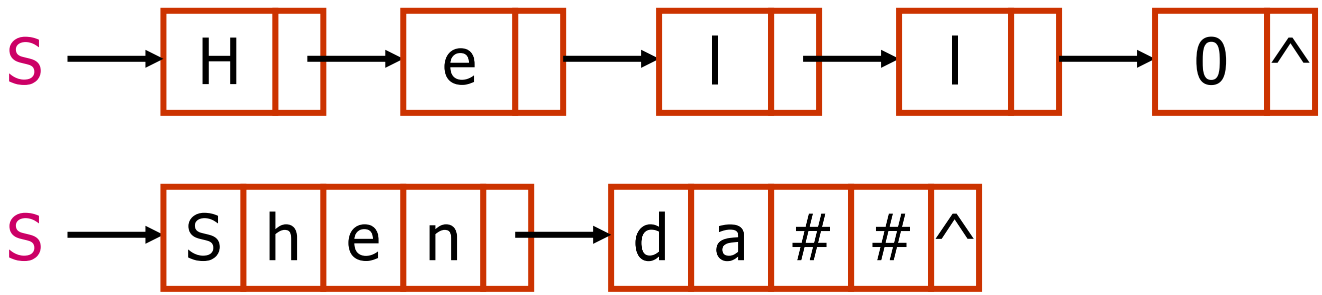
- 堆分配的串插入

```
Status HString::StrInsert(int pos, HString &T) {
    //在当前串的第pos个字符之前插入串T。
    int i;
    if (pos < 1 || pos > length+1)  // pos不合法
        return ERROR;
    if (!T.length) return OK; // T为空
    // T非空,则重新分配空间,插入T
    char *new_ch = new char[length + T.length + 1];
    if (!new_ch) return ERROR;
    for (i=0; i<pos; ++i)  // 复制pos之前的位置
        new_ch[i] = ch[i];
    for (i=0; i<T.length; i++) // 复制T
        new_ch[pos+i] = T.ch[i];
    for (i=pos; i<length; i++)
        new_ch[T.length+i] = ch[i];
    length += T.length; delete ch; ch = new_ch; //更新成员
    return OK;
}
```

4.2 串的表示与实现

三. 块链存储表示

- 采用链表方式存储串值，每个结点中，可以存放一个字符，也可以存放多个字符



4.2 串的表示与实现

三. 块链存储表示

- 因为带了next指针，存储密度不为1

- 在这种存储结构下，结点的分配总是完整的结点为单位，因此，为使一个串能存放在整数个结点中，在串的末尾填上不属于串值的特殊字符，以表示串的终结。

- 当一个块(结点)内存放多个字符时，往往会使操作过程变得较为复杂，如在串中插入或删除字符操作时通常需要在块间移动字符。

4.3 串的匹配

- 模式匹配(模范匹配)：子串在主串中的定位称为模式匹配或串匹配(字符串匹配)。模式匹配成功是指在主串S中能够找到模式串T，否则，称模式串T在主串S中不存在。
- 模式匹配的应用在非常广泛。例如，在文本编辑程序中，我们经常要查找某一特定单词在文本中出现的位置。显然，解此问题的有效算法能极大地提高文本编辑程序的响应性能。
- 模式匹配是一个较为复杂的串操作过程。迄今为止，人们对串的模式匹配提出了许多思想和效率各不相同的计算机算法。

4.3 串的匹配

- 算法（穷举法）：
  - 从主串的指定位置开始，将主串与模式（要查找的子串）的第一个字符比较，
  - 若相等，则继续逐个比较后续字符；
  - 若不等，从主串的下一个字符起再重新和模式的字符比较

4.3 串的匹配

- 实现函数Index

```
int Index(Sstring S, Sstring T, int pos) {
//S为主串，T为模式，串的第0位置存放串长度；串采用顺序存储结构
    i = pos;  j = 1;                    // 从第一个位置开始比较
    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];  // 返回与模式首字符相等的位置
    else return 0;   // 匹配不成功
}
```

⑩ 中途不匹配为什么是i - j + 2 ？？

4.3 串的匹配

- 在最好的情况下，除比较成功的位置外，其余位置仅需比较一次（模式第一个字符），其时间复杂度为：O(n+m)(n，m分别为主串和模式的长度)

- 在最坏的情况下，如模式为‘00000001’，主串为‘0000000000000000000000000000000001’,则每次模式的前7个0都要与主串逐一比较，因此，其时间复杂度为：O(n*m)

4.3 串的匹配

- KMP算法是index函数的一种改进,由D. E. Knuth(克努特)—J. H. Morris(莫里斯)—V. R. Pratt(普拉特)发现
- 算法思路
  - 当一趟匹配过程中出现字符比较不等(失配)时，不需回溯i指针
  - 利用已经得到的“部分匹配”的结果，将模式向右“滑动”尽可能远的一段距离(next[j])后，继续进行比较

4.3 串的匹配

- KMP算法举例

假设主串ababcabcacbab，模式abcac，改进算法的匹配过程如下

```
                    ↓ i=3
第一趟匹配    a b a b c a b c a c b a b
              a b c
                    ↑ j=3

                    ↓ i=3----7
第二趟匹配    a b a b c a b c a c b a b
                  a b c a c
                    ↑ j=1

                            ↓ i=7--10
第三趟匹配    a b a b c a b c a c b a b
                        a b c a c
                            ↑ j=2
```

4.3 串的匹配

- KMP算法说明
- 假设主串为‘$s_1s_2s_3\cdots s_n$’，模式串为‘$p_1p_2p_3\cdots p_m$’
- 若主串中第i个字符与模式串中第j个字符“失配”(s_i!=p_j)，说明，模式串中前面j-1个字符与主串中对应位置的字符相等，即：

$$p_1p_2...p_{j-k}p_{j-k+1}p_{j-k+2}...p_{j-1} = s_{i-j+1}s_{i-j+2}...s_{i-k}s_{i-k+1}s_{i-k+2}...s_{i-1}$$

- 现假定主串中第i个字符需要与模式串中第k(k<j)个字符比较，则说明，模式串中前k-1个字符与主串中对应位置的字符相等，即有以下关系成立：

$$p_1p_2...p_{k-1} = s_{i-k+1}s_{i-k+2}...s_{i-1}$$

4.3 串的匹配

- 从这两表达式

$p_1p_2\ldots p_{j-k}p_{j-k+1}p_{j-k+2}\ldots p_{j-1} = s_{i-j+1}s_{i-j+2}\ldots s_{i-k}s_{i-k+1}s_{i-k+2}\ldots s_{i-1}$

$p_1p_2\ldots p_{k-1} = s_{i-k+1}s_{i-k+2}\ldots s_{i-1}$

- 得到

  $‘p_1p_2\ldots p_{k-1}’ = ‘p_{j-k+1}p_{j-k+2}\ldots p_{j-1}’$

- 换言之,在模式串中第j个字符“失配”时,模式串第k个字符再同主串中对应的失配位置(i)的字符继续进行比较
- k值可以在作串的匹配之前求出，一般用next函数求取k值

4.3 串的匹配

- next函数定义为：

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

k=2，则$p_1=p_{j-1}$ （有1个字符相同）[除j=2外]；
k=3，则$p_1p_2=p_{j-2}p_{j-1}$（有2个字符相同）；

- 课本P82图4.5，结合P81的Next函数进行匹配

4.3 串的匹配

■ KMP算法实现

```
int Index_KMP(Sstring S, Sstring T, int pos) {
//S为主串，T为模式，串的第0位置存放串长度；串采用顺序存储结构
    i = pos; j = 1; // 从第一个位置开始比较
    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
    }
    if (j > T[0])
        return i-T[0];      // 返回与模式第1字符相等的字符在主串中的序号
    else
        return 0;      // 匹配不成功
}
```

4.3 串的匹配

- 求next[j]值的两种算法
  - 传统算法，根据1<k<j，穷举k检查是否满足$p_1 \ldots p_{k-1}$'='$p_{j-k+1} \ldots p_{j-1}$
    若满足条件的k有多个，取最大值　　子串　　主串
  - 改进算法

While(j<模式串长度) {

1. 若i=0或者T_i=T_j,则i++, j++, next[j]=i，实质就是
   next[j+1]= next[j] + 1

2. 否则, i=next[i]，即next[j+1]= next[k] + 1，注意next[k]要回溯直到满足前面的相等条件

}

4.3 串的匹配

- next函数推导举例

模式串: a b a a b c a c

初始状态: j = 1

i = 0

next[j] = i, 即next[1] = 0

j	1	2	3	4	5	6	7	8
模式串	a	b	a	a	b	c	a	c
next[j]	0							

- 注意：next函数只和模式串有关，和主串无关

4.3 串的匹配

- next函数推导举例

模式串: a b a a b c a c

i==0: j = 2

i = 1

next[j] = i, 即next[2] = 1

j	1	2	3	4	5	6	7	8
模式串	a	b	a	a	b	c	a	c
next[j]	0	1						

- 注意：next函数只和模式串有关，和主串无关

4.3 串的匹配

- next函数推导举例

模式串: a b a a b c a c

T1!=T2: j = 2

i = next[i]，即i = next[1] = 0

j	1	2	3	4	5	6	7	8
模式串	a	b	a	a	b	c	a	c
next[j]	0	1						

- 注意：next函数只和模式串有关，和主串无关

4.3 串的匹配

- next函数推导举例

模式串: a b a a b c a c

i==0: j = 3

i = 1

next[j] = i, 即next[3] = 1

j	1	2	3	4	5	6	7	8
模式串	a	b	a	a	b	c	a	c
next[j]	0	1	1					

- 注意：next函数只和模式串有关，和主串无关

4.3 串的匹配

- next函数推导举例

模式串: a b a a b c a c

T1==T3: **j = 4**

i = 2

next[j] = i, 即next[4] = 2

j	1	2	3	4	5	6	7	8
模式串	a	b	a	a	b	c	a	c
next[j]	0	1	1	2				

- 注意：next函数只和模式串有关，和主串无关

4.3 串的匹配

- next函数推导举例

模式串: a b a a b c a c

T2!=T4: j = 4

i = next[i], 即i=next[2]=1

j	1	2	3	4	5	6	7	8
模式串	a	b	a	a	b	c	a	c
next[j]	0	1	1	2				

- 注意：next函数只和模式串有关，和主串无关

4.3 串的匹配

- next函数推导举例

模式串: a b a a b c a c

T1==T4: j = 5

i = 2

next[j] = i, 即next[5] = 2

j	1	2	3	4	5	6	7	8
模式串	a	b	a	a	b	c	a	c
next[j]	0	1	1	2	2			

- 注意：next函数只和模式串有关，和主串无关

4.3 串的匹配

- next函数推导举例

模式串: a b a a b c a c

T2==T5: j = 6

i = 3

next[j] = i, 即next[6] = 3

j	1	2	3	4	5	6	7	8
模式串	a	b	a	a	b	c	a	c
next[j]	0	1	1	2	2	3		

- 注意：next函数只和模式串有关，和主串无关

4.3 串的匹配

- next函数推导举例

模式串: a b a a b c a c

T3!=T6: j = 6

i = next[i]，即i=next[3]=1

j	1	2	3	4	5	6	7	8
模式串	a	b	a	a	b	c	a	c
next[j]	0	1	1	2	2	3		

- 注意：next函数只和模式串有关，和主串无关

4.3 串的匹配

- next函数推导举例

模式串： a b a a b c a c

T1!=T6: j = 6

i = next[i]，即i=next[1]=0

j	1	2	3	4	5	6	7	8
模式串	a	b	a	a	b	c	a	c
next[j]	0	1	1	2	2	3		

- 注意：next函数只和模式串有关，和主串无关

4.3 串的匹配

- next函数推导举例

模式串: a b a a b c a c

i==0: j = 7

i = 1

next[j] = i, 即next[7] = 1

j	1	2	3	4	5	6	7	8
模式串	a	b	a	a	b	c	a	c
next[j]	0	1	1	2	2	3	1	

- 注意：next函数只和模式串有关，和主串无关

4.3 串的匹配

- next函数推导举例

模式串: a b a a b c a c

T1==T7: j = 8

i = 2

next[j] = i, 即next[8] = 2

j	1	2	3	4	5	6	7	8
模式串	a	b	a	a	b	c	a	c
next[j]	0	1	1	2	2	3	1	2

- 注意：next函数只和模式串有关，和主串无关

4.3 串的匹配

- KMP算法的时间复杂度为O(n)
- 为了求模式串的next值，其算法与Index_KMP很相似，其时间复杂度为O(m)
- 因此，KMP算法的时间复杂度为O(n+m)

练习

一. 现有模式abaabc，写出每个字母的next值。

- next[j] = 0 1 1 2 2 3

二. 假设主串abcabadabaabc，模式串如上，说明kmp算法的匹配过程。

- 第1次匹配：从头开始比较到i=3 j=3不等，模式滑动到i=3，选择j=1比较
- 第2次匹配：一开始i=3 j=1不等，查到nextj为0，则i++，j=1
- 第3次匹配：从i=4 j=1开始到i=7 j=4不能，模式滑动到i=7，选择j=2比较
- 第4次匹配：一开始i=7 j=2不等，查到nextj为1，模式滑动1位选择j=1比较
- 第5次匹配：一开始i=7 j=1不等，查到nextj为0，则i++，j=1
- 第6次匹配：从i=8 j=1开始到结束，匹配成功

Take Home Message

- 串：字符串
  - 成员类型为char类型的线性表

- 串的模式匹配
  - 查找子串在主串中出现的位置
  - 穷举法：O(nm)
  - KMP法
    - next矩阵记录子串的跳转信息
    - 活用以匹配的信息进行查找
    - 时间复杂度：O(n+m)