




Data Structures and Algorithms

Strings

- 
- A faint, light gray world map is visible in the background of the slide, centered behind the main content.
- 1 Definition**
 - 2 Implementation**
 - 3 Application**

- ◆ 基本概念：
- ◆ 字符串，由0个或多个字符的顺序排列所组成的复合数据结构，简称“串”。
- ◆ 串的长度：一个字符串所包含的字符个数。
空串：长度为零的串，它不包含任何字符内容。
- ◆ 字符(char)：组成字符串的基本单位。
- ◆ 子串：串中任意个连续的字符组成的子序列称为该串的**子串**。
例，串 "ej" 是串 "beijing" 的子串，"beijing" 称为主串
- ◆ 串的位置：字符在序列中的序号称为该字符在串中的**位置**。**子串在主串中的位置**定义为子串的第一个字符在主串中的位置。
例，字符 'n' 在串 "beijing" 中的位置为 6。
例，子串 "ej" 在串 "beijing" 中的位置为 2。
- ◆ 其他事项：两个串**相等**，当且仅当这两个串的**值相等**。
例，串 "bei jing" 与串 "beijing" 不相等。
串值必须用一对单引号括起来，但单引号本身不属于串，只起界定作用。
由一个或多个空格组成的串称为**空格串**，**空格串不是空串**。

◆ 串与线性表的差别：

- 串的逻辑结构和线性表极为相似，区别仅在于串的数据对象约束为**字符集**。
- 串的**基本操作**与线性表差别
 1. 线性表的基本操作中，大多以“**单个元素**”作为操作对象，如查找某个元素、在某个位置上插入一个元素和删除一个元素。
 2. 串的基本操作中，通常以“**串的整体**”作为操作对象。如在串中查找某个子串、在串的某个位置上插入一个子串以及删除一个子串。

Definition



ADT String{

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

数据关系: $R = \{ \langle a_{i-1}, a_i \rangle | a_{i-1}, a_i \in D, i=2,3,\dots,n \}$

基本操作:

StrAssign(t, chars)

初始条件: **chars**是一个字符串常量。

操作结果: 生成一个值为**chars**的串**t**。

StrConcat(s, t)

初始条件: 串**s, t**已存在。

操作结果: 将串**t**联结到串**s**后形成新串存放到**s**中。

}

Definition



ADT String{

StrLength(t)

初始条件：字符串t已存在。

操作结果：返回串t中的元素个数，称为串长。

SubString (s, pos, len, sub)

初始条件：串s, 已存在, $1 \leq \text{pos} \leq \text{StrLength}(s)$ 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。

.....

} ADT String

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。（或存放一个特殊的空字符，例如C语言中的字符串‘\0’）用这种表示方式，是使用一个预先定义的常数作为数组的大小，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。这种表示方法更适用于经常发生插入和删除操作的字符串，因为不需要移动整个字符串，只需要改变指针。

◆ 串的定长顺序存储表示:

这种存储结构又称为串的顺序存储结构。是用一组连续的存储单元来存放串中的字符序列。所谓定长顺序存储结构，是直接使用定长的字符数组来定义，数组的上界预先确定。

```
4  
6  ✓ #define NAX_STRLEN 256  
7    typedef struct {  
8        char str[NAX_STRLEN];  
9        int length;  
    } StringType;
```


◆ 串的联结操作:

```
10
17 Status StrContract(StringType s, StringType t) {
18     if((s.length + t.length) > MAX_STRLEN) {
19         return ERROR;
20     }
21     for (int i=0; i< t.length; i++) {
22         s.str[s.length + i] = t.str[i];
23     }
24     s.length = s.length + t.length;
25     return OK;
26 }
```

◆ 求子串操作

```
28  ✓ Status SubString(StringType s, int pos, int len, StringType *sub) {  
29  ✓  if (pos < 1 || pos > s.length || len < 0 || len > (s.length - pos + 1)) {  
30      return ERROR;  
31  }  
32      sub->length = len - pos + 1;  
33  ✓  for (int j = 0, k = pos; k <= len; k++, j++) {  
34      sub->str[j] = s.str[k];  
35  }  
36  
37      return OK;  
38  }
```

◆ 串的堆分配存储表示:

实现方法：系统提供一个空间足够大且地址连续的存储空间(称为“堆”)供串使用。

可使用C语言的动态存储分配函数`malloc()`和`free()`来管理。

特点是：仍然以一组地址连续的存储空间来存储字符串值，但其所需的存储空间是在程序执行过程中动态分配，故是动态的，变长的。

```
4  
5  ✓ typedef struct {  
    char *ch;  
7    int length;  
8 } HeapString;  
9
```

◆ 串的联结操作:

```
18 // 指针t指向s1 和 s2 contract的新的子串
19 Status str_contract(HeapString *t, HeapString *s1, HeapString *s2) {
20     if (t->ch) {
21         free(t->ch);
22         t->ch = NULL;
23     }
24
25     t->length = s1->length + s2->length;
26     if ((t->ch = (char *)malloc( size: sizeof(char) * t->length)) == NULL) {
27         printf("系统空间不够, 申请空间失败 ! \n");
28         return ERROR;
29     }
30     for (int j = 0; j < s1->length; j++) {
31         t->ch[j] = s1->ch[j];
32     }
33     for (int k = s1->length, j = 0; j < s2->length; k++, j++) {
34         t->ch[k] = s2->ch[j];
35     }
36
37     return OK;
38 }
```

Implementation

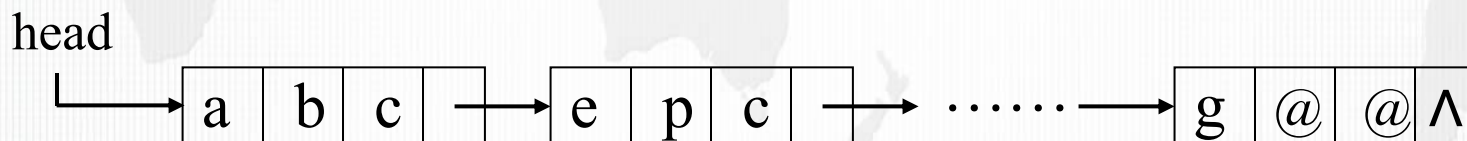


◆ 串的链式存储表示:

串的链式存储结构和线性表的串的链式存储结构类似，采用单链表来存储串，结点的构成是：

- ◆ **data**域：存放字符，**data**域可存放的字符个数称为结点的大小；
- ◆ **next**域：存放指向下一结点的指针。

若每个结点仅存放一个字符，则结点的指针域就非常多，造成系统空间浪费，为节省存储空间，考虑串结构的特殊性，使每个结点存放若干个字符，这种结构称为块链结构。如图是块大小为3的串的块链式存储结构示意图。

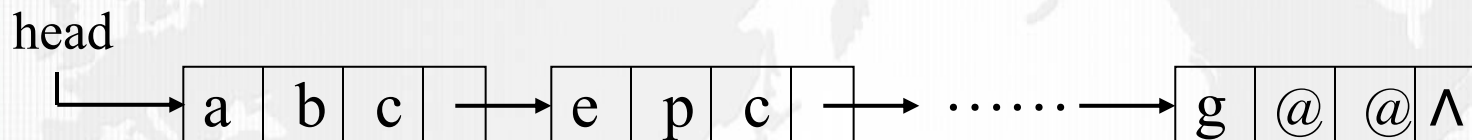


Implementation



◆ 串的链式存储表示:

串的块链式存储的类型定义包括:

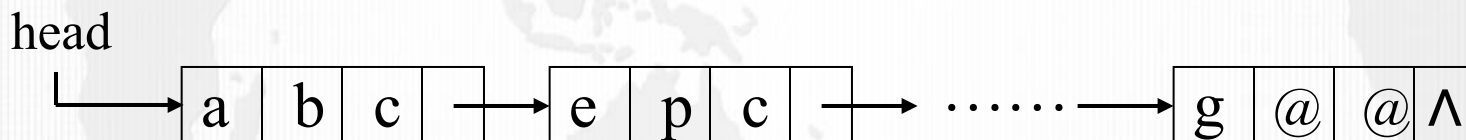


```
5  # define CHUNK_SIZE 80
6
7  typedef struct Chunk {
8      char ch[CHUNK_SIZE];
9      struct Chunk* next
10 } Chunk;
11
12 typedef struct {
13     Chunk *head, *tail;
14     int length;
15 } LinkString;
```

◆ 串的链式存储表示:

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。



◆ 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。介绍两种主要的模式匹配算法。

◆ Brute-Force模式匹配算法

基本思想：

将主串S的第pos个字符和模式T的第1个字符比较，

- 若相等，继续逐个比较后续字符；

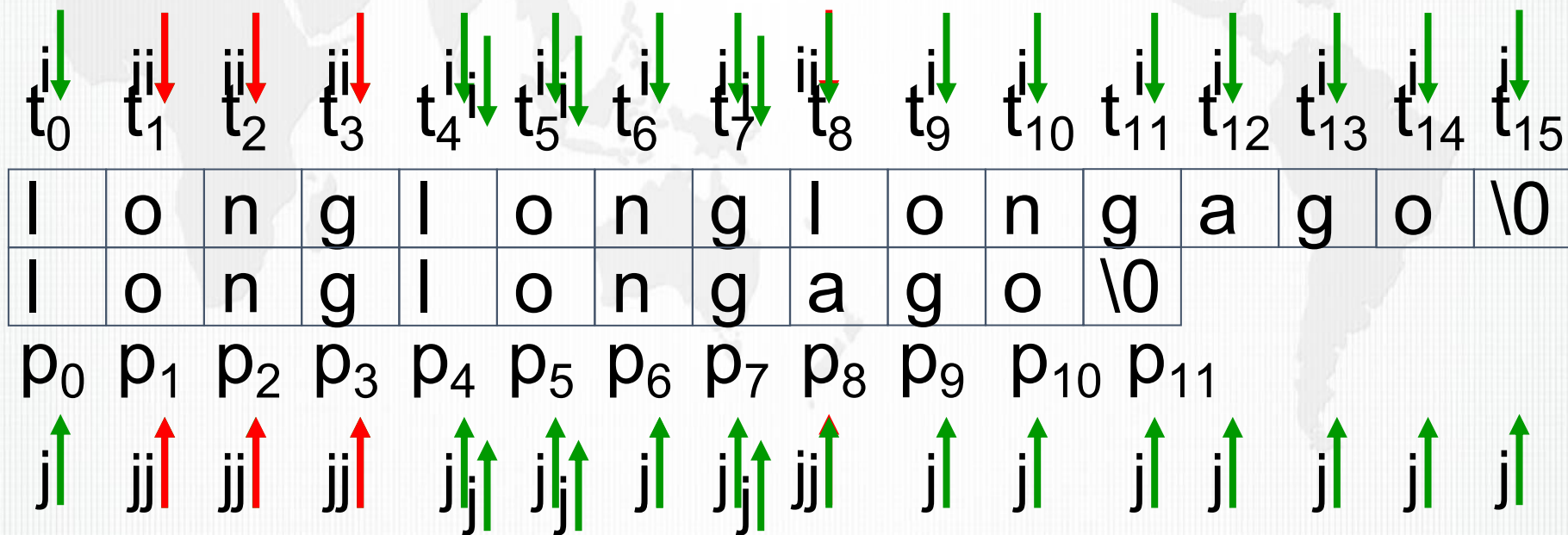
- 若不等，从主串S的下一字符（pos+1）起，重新与T第一个字符比较。

直到主串S的一个连续子串字符序列与模式T相等。返回值为S中与T匹配的子序列第一个字符的序号，即匹配成功。否则，匹配失败，返回值0。

◆ Brute-Force模式匹配算法

基本过程：

第五趟比较：



◆ Brute-Force模式匹配算法

```
int index_string(SequentialString source, SequentialString target, int pos) {
    if (pos < 0 || pos >= source.length) {
        return ERROR;
    }

    char *source_pointer = source.str + pos;
    char *target_pointer = target.str;
    int source_index = pos;
    int target_index = 0;

    while (source_index < source.length && target_index < target.length) {
        if (*source_pointer == *target_pointer) {
            source_pointer++;
            target_pointer++;
            source_index++;
            target_index++;
        } else {
            source_index = source_index - target_index + 1;
            target_index = 0;
            target_pointer = target.str;
            source_pointer = source.str + source_index;
        }
    }

    if (target_index == target.length) {
        return source_index - target.length;
    }
    return ERROR;
}
```

◆ Brute-Force模式匹配算法

- 设主串的长度为 n , 子串的长度为 m , 在最坏情况下, 比较次数:

$$(n-m+1)*m$$

- 在多数情况下, m 远小于 n , 因此算法的最坏的时间复杂性为 $O(n*m)$
- 复杂度高, 效率低