字 符 串

学习目标

了解字符串的定义包括几个概念

熟知字符串的基本操作

熟知字符串的抽象数据类型定义

掌握字符串的存储结构及基本操作实现算法

串的定义

- 线性表（表）：具有相同类型的数据元素的有限序列

⇩ 将元素类型限制为字符

- 字符串（串）：零个或多个字符组成的有限序列

$$(a_1,\ a_2,\ \dots,\ a_i,\ \dots,\ a_n)$$

⇩

串名 — S = " $s_1\ s_2\ \dots\dots\ s_n$ " — 定界符

串的基本概念

- 串长：串中所包含的字符个数
- 空串：长度为 0 的串
- 子串：串中任意个连续的字符组成的子序列,空串是任意串的子串
  主串：包含子串的串
  子串的位置：子串的第一个字符在主串中的序号

$S1$ = "$ab12cd$ "
$S2$ = "$ab12$"
$S3$ = "$ab12$␣"
$S4$ = "$ab13$"
$S5$ = " "
$S6$ = "␣␣␣"

字符串相关概念

前缀：从长度为n的字符串第0位开始，第i位（$0 \le i < n-1$）结束的任意子串。对字符串s，其前缀可表示为s[0...i](0≤i<n-1)。字符串的所有前缀构成的集合称为前缀集合。

后缀：从长度为n的字符串第i位（$0 < i \le n-1$）开始，最后一位结束的任意子串。对字符串s，其后缀可表示为s[i...n-1](0<i≤n-1)。字符串的所有后缀构成的集合称为后缀集合。

公共前后缀：字符串的前缀集合与后缀集合中相同的子串。

最长公共前后缀：字符串的前缀集合与后缀集合中相同的长度最长的子串。

例：字符串" aabaa "

前缀集合为:

{"a","aa","aab","aaba"}

后缀集合为:

{"a","aa","baa","abaa"}

其公共前后缀:

"a"和"aa“

最长公共前缀:

"aa"

思考

设S 为一个长度为n 的字符串，其中的字符各不相同，则S 中的互异的非平凡子串（非空且不同于S本身）的个数为 _____

A．2n-1

B．n^2

C．$(n^2/2)+(n/2)$

D．$(n^2/2)+(n/2)-1$

***ab12c*　　*n=5*（长度）**

a开头的子串：
a
ab
ab1
ab12
ab12c(不能含自己)

n-1

b开头的子串：
b
b1
b12
b12c

n-1

1开头的子串：
1
12
12c

n-2

2开头的子串：
2
2c

c开头的子串：
c

***n* +n-1+…+1 -1**

单选题 5分

设置

1. 空串与空格串是相同的。

A 正确

B 错误

提交

字符串的抽象数据类型定义

为了与C++中字符串区别

ADT **String** {

数据对象：

$D=\{ s_i \mid s_i \in \text{CharacterSet}, i=1, 2, \ldots, n, n>0 \}$ 或 $\varnothing$表示空字符串

数据关系：

$R=\{ \langle s_{i-1}, s_i \rangle \mid s_{i-1}, s_i \in D, i=2, \ldots, n, n>0 \}$

基本操作:

……

}

字符串的抽象数据类型定义

ADT **String** {　…..

基本操作:

InitStr(s)：初始化一个空的字符串s，字符串最大长度为kMaxSize。

StrCopy(s)：返回复制字符串s得到的字符串。

StrIsEmpty(s)：判断字符串s是否是空串。返回一个布尔值，若字符串s是空串，返回true；否则返回false。

StrLength(s)：返回字符串s的长度。

StrInsert(s, pos, t)：在字符串s的pos位置处插入字符串t，并返回插入后的字符串s。

StrRemove(s, pos, len)：删除字符串s中从pos位置开始的长度为len的子串，返回删除后的字符串s。

SubString(s, pos, len)：返回字符串s从pos位置开始长度为len的子串。

StrConcat(s,t)：返回字符串s和t联接而成的新串s。

Replace(s, sub_s, t)：将字符串s中的所有子串sub_s替换为字符串t。

StrCompare(s,t)：返回字符串s和t的大小关系。若s>t，返回+1；若s=t，返回0；若s<t，返回-1。

PatternMatch(s,t)：返回字符串s中首次出现字符串t的位置，若字符串s没有出现字符串t，则返回NIL。

}

C++中的字符串

在C++中的字符串是经过封装的

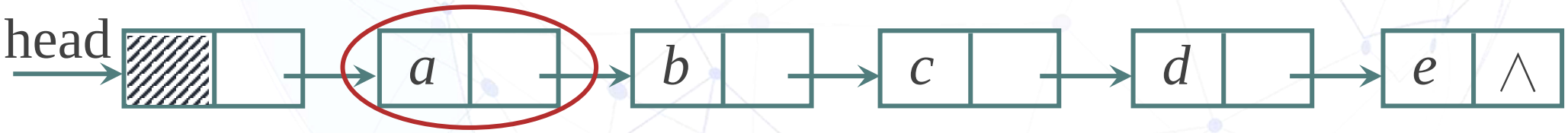
在标准模板库文件String中定义，这个文件实现了一个std::string类，比C语言串更方便、安全和高效。

```
#include <string>
using namespace std;
```

字符串的存储实现

字符串存储表示

- **顺序存储结构**：将所有的数据元素存放在一段连续的存储空间中，数据元素的存储位置反应了它们之间的逻辑关系
  - 字符串通常采用顺序存储，即用数组存储
- **链式存储结构**：逻辑上相邻的数据元素不需要在物理位置上也相邻，数据元素的存储位置可以是任意的

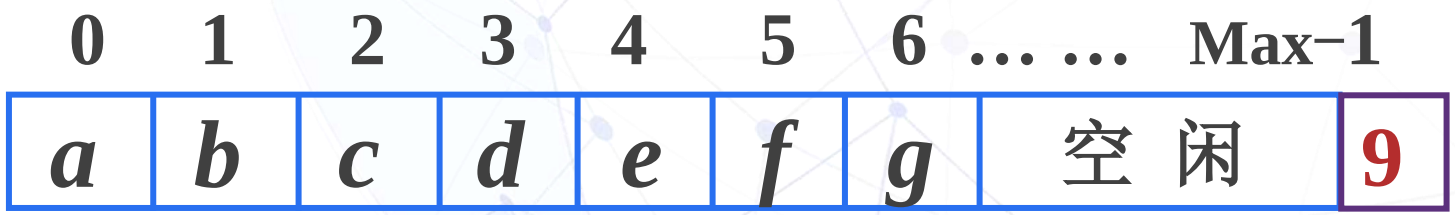


- 指针的结构性开销降低空间性能

顺序字符串（数组存储）

如何表示串的长度?

方案 1：用一个变量来表示串的实际长度

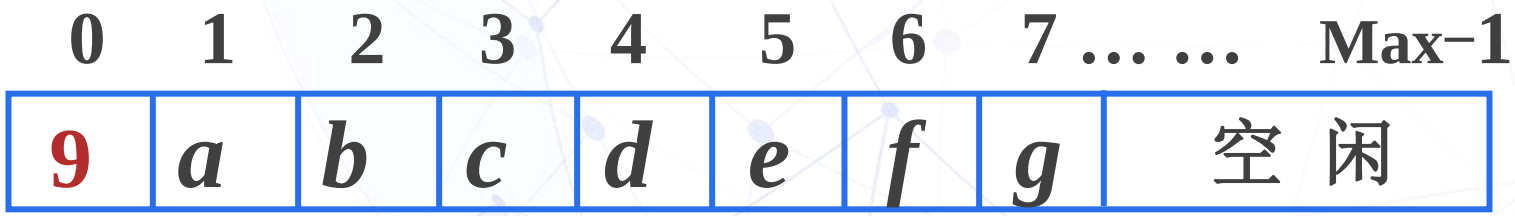


顺序字符串（数组存储）

如何表示串的长度?

方案 1：用一个变量来表示串的实际长度

方案 2：用数组的 0 号单元存放串的长度，从 1 号单元开始存放串值



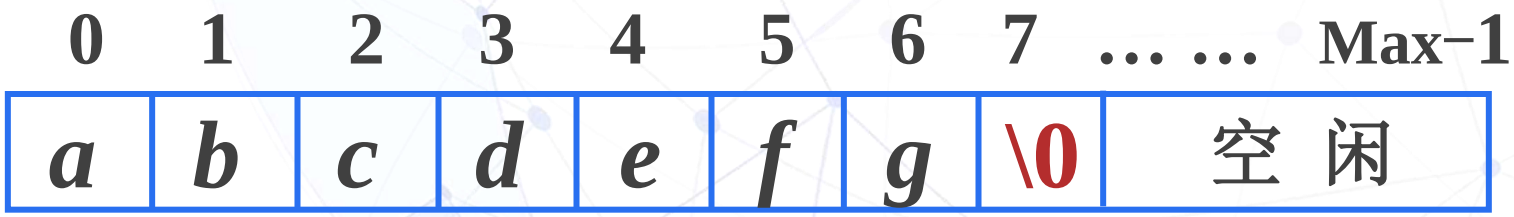
顺序字符串（数组存储）

如何表示串的长度?

方案 1：用一个变量来表示串的实际长度

方案 2：用数组的 0 号单元存放串的长度，从 1 号单元开始存放串值

方案 3：在串尾存储一个不会在串中出现的特殊字符作为串的终结符，表示串的结尾



一般在程序设计语言中，字符串会有结束符，如C语言和C++语言中字符串的结束符为'\0'，结束符不计入字符串长度，但要占存储空间。

顺序字符串的结构

data	a_1	a_2	……	a_i	a_{i+1}	……	a_n	……	-
i	0	1	……	*i*-1	*i*	……	*n*-1	……	kMaxSize-1


```
const int kMaxSize = 100;

class String
{
    public:
        [methods]
    private:
        char data[kMaxSize];
        int  length;
};
```

顺序字符串的操作

1.字符串的插入

2.字符串的删除

3.字符串的截取

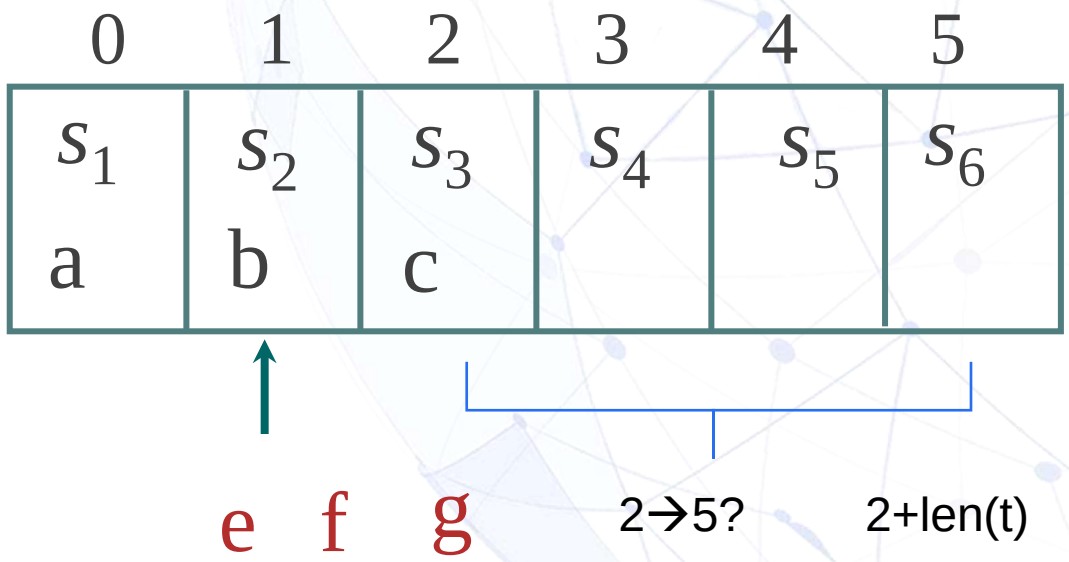
4.字符串的连接

5.字符串的比较

1. 字符串插入 strInsert(s, pos, t)

从字符串s的pos位置插入字符串t

举例对于字符串s=“abc”，在 $i = 2$的位置上插入字符串t=“efg”



StrInsert(s, pos, t) 的核心步骤：

(1) 首先将字符串s从pos位置开始整体后移len(t)个距离，为字符串t的插入让出插入空间；

(2) 然后将字符串t中的字符逐一插入到字符串s从pos开始的空间中，得到插入后的字符串s；

(3) 最后更新字符串s的length属性。

字符串插入算法

```
算法4-1  字符串插入操作StrInsert(s, pos, t)
输入：字符串s ，要插入的位置pos≥1，需要插入的字符串t
输出：完成插入后的字符串s。若插入后的字符串长度大于kMaxSize，则直接退出
n ← s.length
m ← t.length
if n+m≤kMaxSize then
| for i←n-1 downto pos-1 do
| | s.data[i+m] ← s.data[i]
      //将数组下标pos-1开始的子串后移，给t 留出空位
| end
| for i←0 to m-1 do
| | s.data[pos-1+i] ← t.data[i]  //将t 插入s
| end
| s.length ← n + m  //更新s的长度
else
| 长度超限，退出
end
```

边界处理：

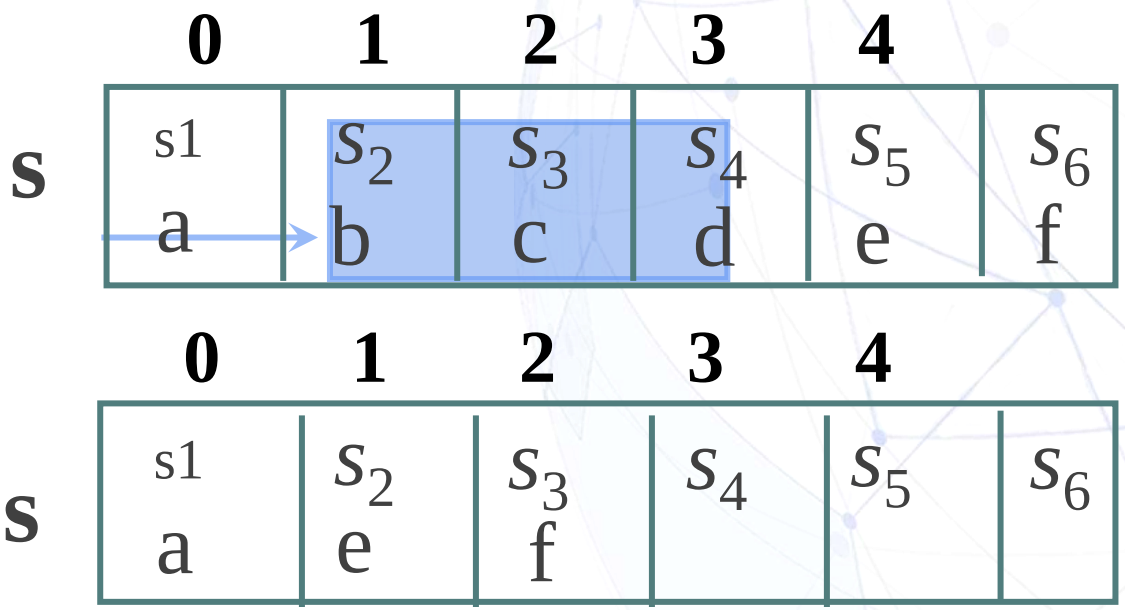
n+m≤kMaxSize，其中n和m分别是两个字符串的长度。

时间复杂度：

最坏情况是将t插在s的头，时间复杂度是**O(n+m)**

2. 字符串删除 StrRemove(s, pos, len)

对于s=“abcde”删除位置 $i = 2$，len=3



(1)将字符串s从pos位置开始删除长度为len的子串;

(2)将字符串s中pos+len后面的部分逐位向前移动;

(3)最后更新字符串s的length属性。

2. 字符串删除

算法4-2 字符串删除操作StrRemove(s, pos, len)

输入：字符串s，要删除的位置pos≥1，删除的字符个数len
输出：删除子串后的字符串s

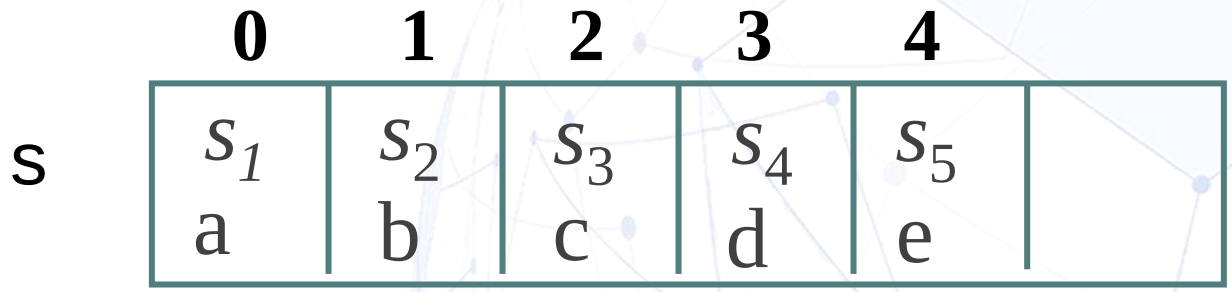
```
n ← s.length
if pos+len-1<n then
|  for i ← pos+len-1 to n-1 do
|  |  s.data[i-len] ← s.data[i]
|  end
|  s.length ← n – len
else  //从数组下标pos-1开始的所有字符都删掉
|  s.length ← pos-1
end
```

?

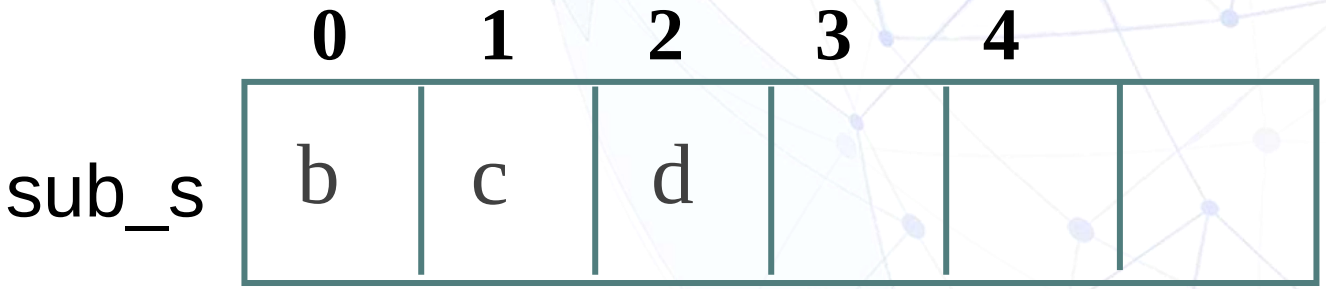
时间复杂度：最坏情况是将s开头的len个字符删除，时间复杂度是**O(n)**，其中n是s的长度。

3. 字符串截取 SubString(s, pos, len)

返回字符串s从pos位置开始长度为len的子串:



SubString(s,2,3)?



(1) 构造新串sub_s;
(2) 将字符串s从pos开始的len个字符一一赋值给sub_s;
(3) 同时维护sub_s的长度。

3. 字符串截取　SubString(s, pos, len)

算法4-3　字符串截取子串操作SubString(s, pos, len)

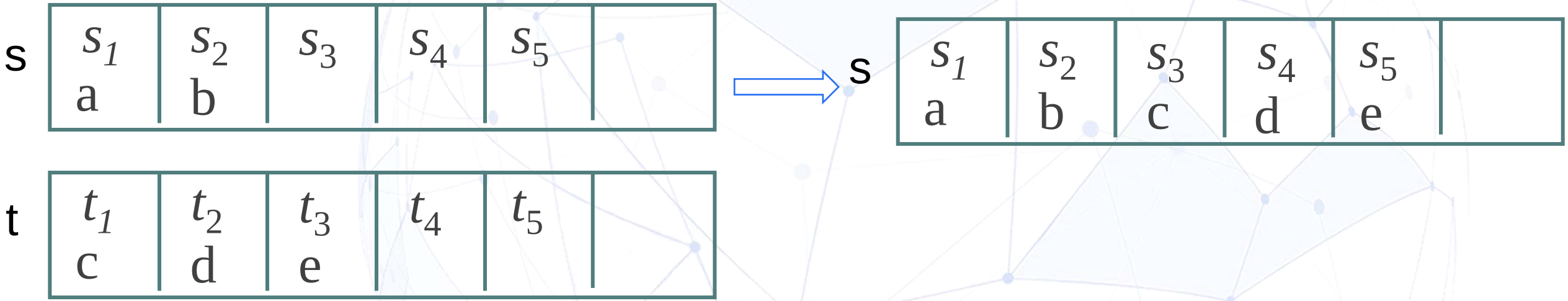
```
输入：字符串s，开始截取的位置pos≥1，截取的字符个数len
输出：截取的子串
InitStr(sub_s)  //初始化新串sub_s
n ← s.length
i ← 0
while pos-1+i<n 且 i<len do
   //若s从pos-1开始不到len个字符，就截取到s的末尾为止
|  sub_s.data[i] ← s.data[pos-1+i]
   //将s串从pos-1之后的len个字符复制到sub_s
|  sub_s.length ← sub_s.length + 1
|  i ← i+1
end
return sub_s
```

- 为啥不用sub_s.length=len?

时间复杂度：该操作仅与子串长度有关，为**O(len)**。

4. 字符串连接 StrConcat(s,t)

将字符串t连接在字符串s末尾



(1) 将字符串t中的字符一一赋值到s末尾;

(2) 并更新s的长度。

4. 字符串连接 StrConcat(s,t)

算法4-4 字符串连接操作StrConcat(s,t)

输入：字符串s，字符串t
输出：返回字符串s后面联接t而成的新串。若结果长度大于kMaxSize, 则退出。

```
n ← s.length
m ← t.length
if n+m ≤ kMaxSize then
|  for i ← 0 to m-1 do
|  |  s.data[n+i] ← t.data[i]
|  end
|  s.length ← n+m
else
|  长度超限，退出
end
```

注意边界处理

时间复杂度

该操作仅与t的长度有关，为**O(m)**。

5. 字符串比较

字符串比较操作 StrCompare(s,t)

对两个字符串s和t做比较:

若s>t返回+1;
若s=t返回0;
若s<t返回-1。

单选题 5分

设置

字符串"abc"（ ）字符串"aabbcc"。

A 大于

B 等于

C 小于

D 无法比较

提交

5. 字符串比较

StrCompare(s,t)将这两个字符串s和t从左到右逐个字符按照其ASCII码值进行比较：

- 如果两个字符串**长度相等**，且每一个相应位置上的字符都相同，则这两个**字符串相等**，如"abc"与"abc"相等。
- 如果两个字符串**长度不相等**，但较短字符串所有对应位置上的字符都与较长字符串相同，则字符串长度长的字符串大于长度短的字符串。如"abc"<"abcdef"。
- 如果两个字符串在某一相应位置上的字符不相同，则以第一个不相同的位置上的字符比较结果作为两个字符串的比较结果。如"abh">"abf"。

5. 字符串比较算法

```
算法4-5  字符串比较操作StrCompare(s,t)
输入：字符串s ， 字符串t
输出：s>t输出+1；s=t输出0；s<t输出-1
len ← Min(s.length, t.length)
i ← 0
while i<len 且s.data[i] = t.data[i] do
   i ← i+1
end
if i=len then
|  if s.length > len then
|  |  ret ← 1
|  else if t.length > len then
|  |  ret ← -1
|  else  //s=t
|  |  ret ← 0
|  end
else if s.data[i] > t.data[i] then
|  |  ret ← 1
else
|  |  ret ← -1
end
return ret
```

时间复杂度：

仅与s和t的最小长度有关

O(min(n, m))

串（链式）的操作

字符串的插入

字符串的删除

字符串的截取

字符串的连接

串的存储（链式）

非紧缩链接存储结构: 将每个结点存放一个字符的字符串链接存储方式，也称为稀疏存储；

紧缩链接存储结构: 一个结点存放多个字符的字符串链接存储方式，也称为块链存储结构。

例：字符串"abcdef"

注意：此后的算法以稀疏存储为例，且不带头结点

字符串的链式存储实现

```
class String{
public:
        String( ); //无参构造函数，建立只有头结点的空链表

        ~ String( ); //析构函数
        InitString(); //初始化
        String SubString(s, pos, len);      //截取子串
        String StrInsert(s, pos, t);     //插入操作
        String StrRemove(s, pos, len);     //删除操作，
        String StrConcat(s,t)                    //连接操作
        int StrCompare(s,t)                      //字符串比较
        int StrLength();                         //获得实际长度
        ……
private:
        Node<char> *head;                  //单链表的头指针
};
```

```
template <typename DataType>
struct  Node
{
        DataType   data;
        Node *next;
} Node;
```

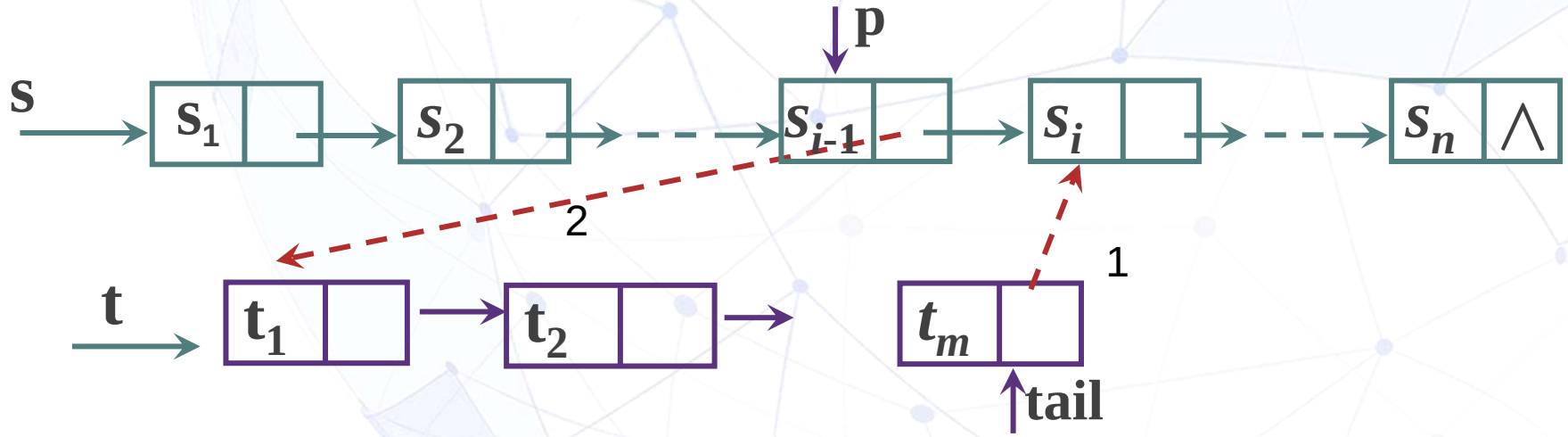
char

1. 字符串的链式存储-字符串插入

相对于顺序存储结构，使用链接存储结构的字符串插入操作较为简单，不需要担心字符串长度超限的问题。

（1）需要找到字符串s中位于pos位置的链表元素，

（2）需要找到插入字符串t的末尾



1. 字符串插入 StrInsert(s, pos, t)

```
算法4-6 字符串插入操作StrInsert(s, pos, t)
输入：字符串s，要插入的位置pos≥1，需要插入的字符串t
输出：完成插入后的字符串s。若不存在位置pos，则s不变
1       flag ← NormalCode
2       if t.length>0 then //若t不是空串
3       | tail ← t.head
4       | while tail.next ≠ NIL do
            //找到t的最后一个元素
5       | | tail ← tail.next
6       | end
7       | if s.length>0 then //若s不是空串
8       | | p ← s.head
9       | | count ← 1
10      | | while p ≠ NIL 且 count<pos-1 do
        //找第pos个元素的前一个元素
11      | | | count ← count + 1
12      | | | p ← p.next
13      | | end
14      | | if count=(pos-1) then //将t插在p的后面
15      | | | tail.next ← p.next
16      | | | p.next ← t.head
17      | | else if pos = 1 then //t插在s的头
18      | | | tail.next ← s.head
19      | | | s.head ← t.head
20      | | else
21      | | | flag ← ErrorCode //输入错误：不存在位
置pos
22      | | end
23      | else //若s是空串
24      | | s ← t
25      | end
26      end
27      if flag ≠ ErrorCode then  //正常完成插入
28      | s.length ← s.length + t.length
29      end
```

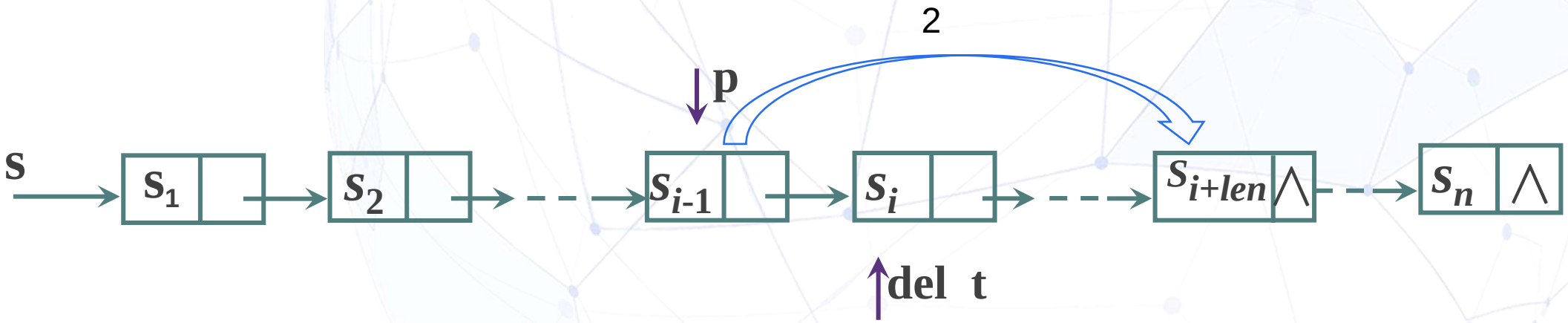
时间复杂度：该操作时间复杂度是**O(n+m)**，其中n和m分别是两个字符串的长度。

2. 字符串的链式存储-字符串删除

返回字符串s从pos位置开始删除长度为len的子串后的字符串。

（1）将pos-1位置上的链表元素的next指针指向原字符串s中的第pos+len个元素，

（2）释放被删除的结点空间。



t=del->next
delete del

2. 字符串删除StrRemove(s, pos, len)

算法4-7 字符串删除操作StrRemove(s, pos, len)

输入：字符串s，要删除的位置pos≥1，删除的字符个数len
输出：删除子串后的字符串s。若删除位置不存在，则s不变

```
1.  if s.length>0 then //若s不是空串
2.  | p ← s.head
3.  | count ← 1
4.  | while p ≠ NIL 且 count<pos-1 do
        //找第pos个元素的前一个元素
5.  | | count ← count + 1
6.  | | p ← p.next
7.  | end
8.  | if pos=1 或 (p ≠ NIL 且count=pos-1) then
        //将p后面的len个结点删除
9.  | | if pos=1 then
10. | | | deleted ← s.head
11. | | else
12. | | | deleted ← p.next
13. | | end
14. | | count ← 0
15. | | while deleted≠ NIL 且 count < len do
        //不足len个则一直删到末尾
16. | | | t ← deleted.next
17. | | | delete deleted
18. | | | count ← count + 1
19. | | | s.length ← s.length - 1
20. | | | deleted ← t
21. | | end
22. | | if pos = 1 then
23. | | | s.head ← deleted
24. | | else
25. | | | p.next ← deleted
26. | | end
27. | end
28. end
```

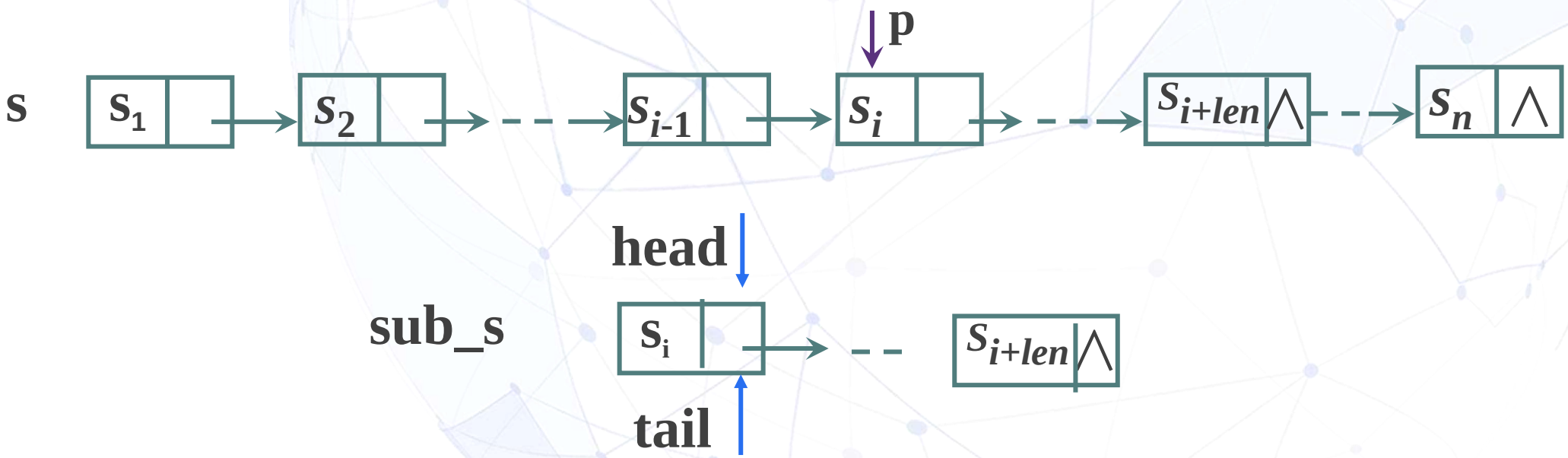
时间复杂度：这个操作的时间复杂度与s的长度有关，为**O(n)**

3. 字符串的链式存储-字符串截取

核心操作：

（1）返回字符串s从pos位置开始的长度为len的子串。

（2）构造新串，将字符串s从pos之后len个字符一一赋值给新串。



3. 字符串截取SubString(s, pos, len)

算法4-8 字符串截取子串操作SubString(s, pos, len)

输入：字符串s，开始截取的位置pos≥1，截取字符个数len
输出：截取的子串

```
1.  InitStr(sub_s)  //初始化新串sub_s
2.  if s.length>0 then //若s不是空串
3.  | p ← s.head
4.  | count ← 1
5.  | while p ≠ NIL 且 count<pos do
        //找第pos个元素
6.  | | count ← count + 1
7.  | | p ← p.next
8.  | end
9.  | if p ≠ NIL 且count=pos then
        //将s串从pos之后的len个字符复制到sub_s
10. | | count ← 0
11. | | sub_s.head ← new StringNode()
        //创建临时空头结点
12. | | tail ← sub_s.head
13. | | while p ≠ NIL 且 count<len do
        //若从pos开始不到len个字符，就截取到s的末尾
14. | | | t ← new StringNode(p.data, NIL)
        //复制一个新结点
15. | | | tail.next ← t  //将新结点接到sub_s的末尾
16. | | | tail ← tail.next
17. | | | sub_s.length ← sub_s.length + 1
        //sub_s 长度加1
18. | | | p ← p.next
19. | | | count ← count + 1
20. | | end
21. | end
22. end
23. tail ← sub_s.head
24. sub_s.head ← tail.next
25. delete tail  //删除临时空头结点
26. return sub_s
```

时间复杂度：与顺序存储不同，这个操作必须首先找到截取的起始位置，时间复杂度就不仅与要截取的子串长度len有关了，最坏时间复杂度为O(n)。

4. 字符串连接 StrConcat(s,t)

将字符串t连接在字符串s末尾：

与顺序存储不同，这里不需要将字符串t中的字符复制到字符串s末尾，只需要找到字符串s末尾并将字符串t接在后面即可。

时间复杂度：这个操作的时间复杂度与字符串t的长度无关，仅与字符串s的长度成正比，为**O(n)**。

算法4-9 字符串连接操作StrConcat(s,t)

```
输入：字符串s ， 字符串t
输出：返回后面联接t而成的字符串s
1.      if s.length > 0 then //若s非空串
2.      | p ← s.head
3.      | while p.next ≠ NIL do  //找到s的最后一个结点
4.      | | p ← p.next
5.      | end
6.      | p.next ← t.head
7.      else  //若s是空串
8.      | s.head ← t.head
9.      end
10.     s.length ← s.length + t.length
```

5.字符串的链式存储-字符串比较

返回字符串s和字符串t的大小关系，与顺序存储的字符串比较规则一致:

- 若s>t返回+1;
- 若s=t返回0;
- 若s<t返回-1。

时间复杂度：这个操作的时间复杂度也是仅与s和t的最小长度有关，为**O(min(n, m))**

算法4-10 字符串比较操作StrCompare(s,t)

输入：字符串s，字符串t
输出：s>t输出+1；s=t输出0；s<t输出-1

```
1.    sp ← s.head
2.    tp ← t.head
3.    while sp≠NIL 且 tp≠NIL且 sp.data = tp.data do
4.    | sp ← sp.next
5.    | p ← tp.next
6.    end
7.    if sp≠NIL且tp=NILthen
8.    | ret ← 1
9.    else if sp=NIL且tp≠NIL then
10.   | ret ← -1
11.   else if sp=NIL且tp=NIL then //s=t
12.   | ret ← 0
13.   else if sp.data > tp.data then
14.   | | ret ← 1
15.   else //sp.data < tp.data
16.   | | ret ← -1
17.   end
18.   return ret
```

课堂小结

字符串的定义和逻辑结构

字符串的实现(顺序和链式)