

数据结构

Data Structure

2017年秋季学期
刘鹏远

1、队列的应用

2、串

队列的应用 — 逐行打印二项展开式 $(a + b)^i$ 的系数

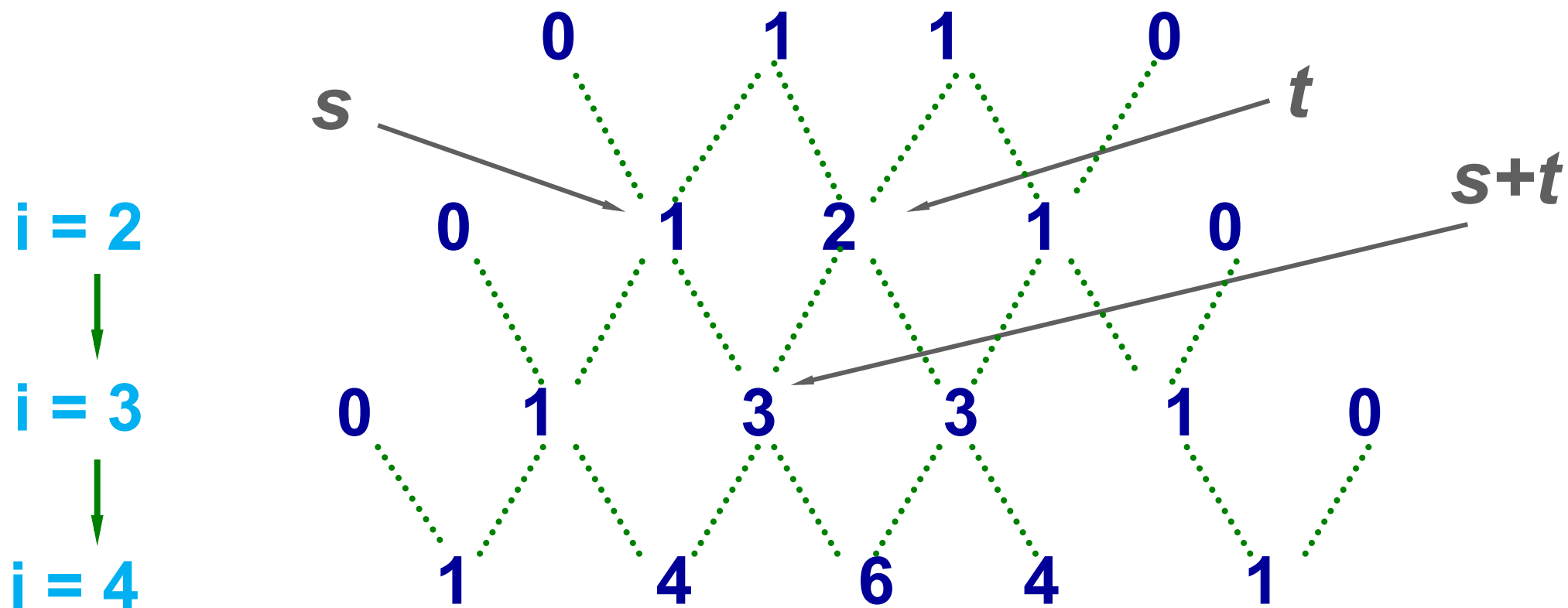
杨辉三角形

1

(Pascal' s triangle) i= 0

				1		1					$i = 1$
			1		2		1				2
		1		3		3		1			3
	1		4		6		4		1		4
1		5		10		10		5		1	5
	6		15		20		15		6		6

分析第 i 行元素与第 $i+1$ 行元素的关系



从前一行的数据可以计算下一行的数据

从第 i 行数据计算并存放第 $i+1$ 行数据

$s=0$

	1	0										
--	---	---	--	--	--	--	--	--	--	--	--	--

de_queue(Q,t), 入队s+t, $s=t$

		0	1									
--	--	---	---	--	--	--	--	--	--	--	--	--

de_queue(Q,t), 入队s+t, $s=t$

			1	1								
--	--	--	---	---	--	--	--	--	--	--	--	--

出栈元素为0时候，循环结束，。最后再enqueue(0)

			1	1	0							
--	--	--	---	---	---	--	--	--	--	--	--	--

从第 i 行数据计算并存放第 $i+1$ 行数据

$s=0$

	1	1	0									
--	---	---	---	--	--	--	--	--	--	--	--	--

de_queue(Q,t), 入队s+t, $s=t$

		1	0	1								
--	--	---	---	---	--	--	--	--	--	--	--	--

de_queue(Q,t), 入队s+t, $s=t$

			0	1	2							
--	--	--	---	---	---	--	--	--	--	--	--	--

de_queue(Q,t), 入队s+t, $s=t$. $t == 0$, 循环结束, enqueue(0)

				1	2		1	0				
--	--	--	--	---	---	--	---	---	--	--	--	--

实践时间

请同学们自行画出第四行的队列进出顺序

自己尝试写下实现打印杨辉三角的伪码/描述

伪码实现

```
Status print_yanghui(int n){  
    queue Q;init_queue(Q);enqueue(1);enqueue(0);  
    //完成准备，并把第0行的数据压入了  
    for(i=0;i<=n;i++){    s=0;  
        do{t = dequeue(Q,t);enqueue(Q,s+t);s=t;  
        }while(t)  
        //在do循环中找合适地方输出  
        enqueue(0);//注意把最后的0压入  
    }//此时栈内存有最后一行  
    return OK;  
}
```


优先（级）队列 (Priority Queue)

- 优先队列 每次从队列中取出的是具有最高优先权的元素
- 如下表：任务优先权及执行顺序的关系

任 务 编 号	1	2	3	4	5
优 先 权	20	0	40	30	10
执 行 顺 序	3	1	5	4	2

数字越小，优先权越高 通常用堆来实现

注意：优先队列是一种特殊的队列，说的是逻辑结构，其与循环(顺序)队列、链队列不是一个层面的东西，后两者是存储结构。

思考：有没有优先级栈呢？

银行模拟

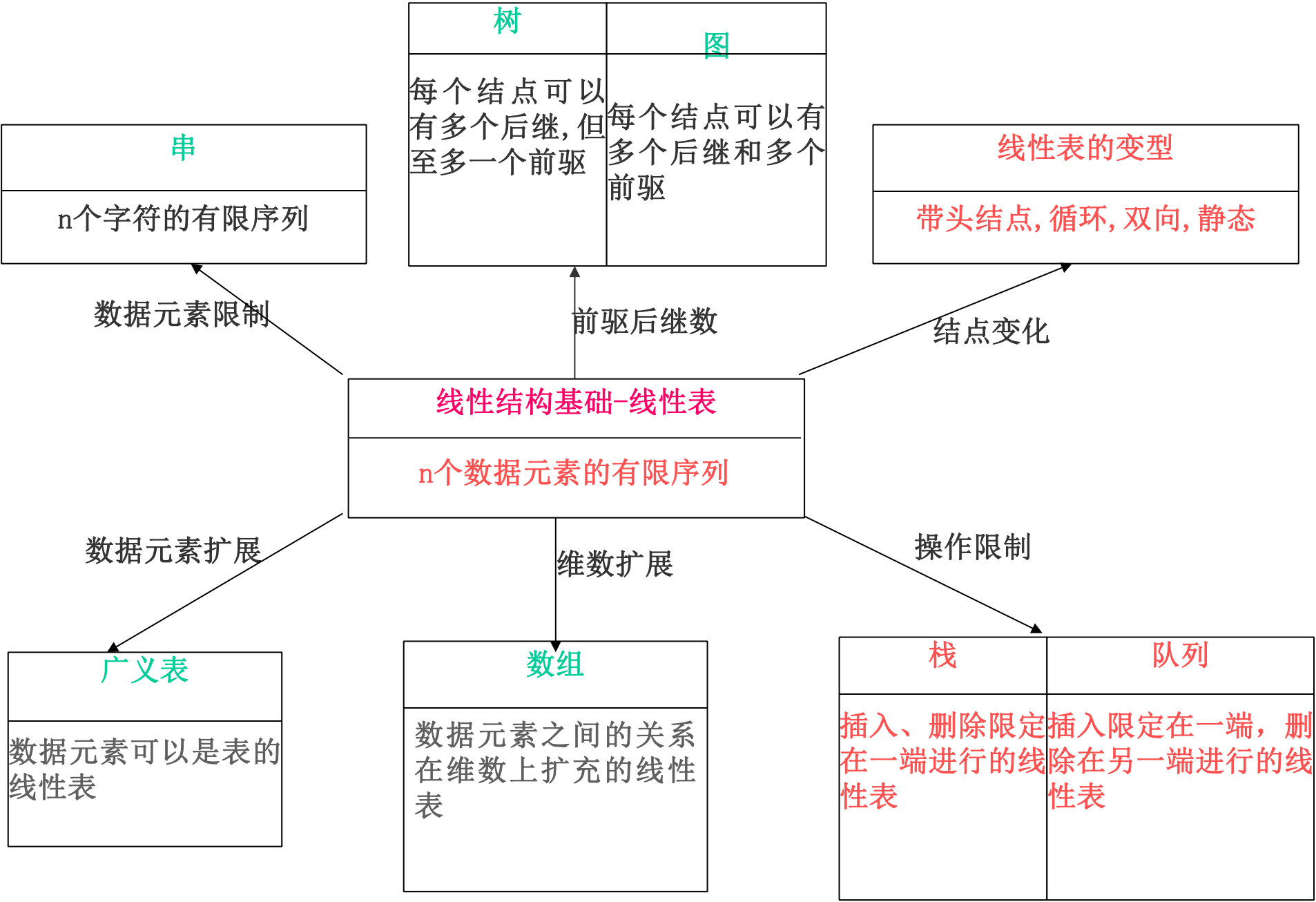
排队窗口为队列

事件（时间轴）为链表（有序）

自学

1、队列的应用

2、串(字符串)



- 1、串（字符串）的逻辑结构是**线性**的。
- 2、也是限定性线性表，是对线性表中每一个元素进行约束，使每一个**元素必须限定在字符集中**。

3、操作上有特殊性：

线性表中多是对单个元素进行操作。

串很多时候对**连起的多个元素作为整体操作**
（如求子串、插入/删除子串）。

1. 定义	}	1	<u>串类型的定义</u>
2. 逻辑结构			
3. 存储结构	}	2	串的实现
4. 操作实现		3	串的模式匹配算法
5. 应用(略)			

概念与逻辑结构

串即字符串，是由**零个或多个**字符组成的有限序列，
是数据元素为单个字符(限定在字符集)的**特殊线性表**。

记为： $s = "a_1, a_2, \dots, a_n"$ $(n \geq 0)$

串名

串值（用“ ”括起来）

隐含结束符
‘\0’，即
ASCII码NULL

若干术语：

串长n：串中字符个数（ $n \geq 0$ ）。 $n=0$ 时称为空串 \emptyset 。

空白串：由一个或多个**空格符**组成的串。

子串：串S中任意个连续的字符序列叫S的子串；S叫**主串**。

子串位置：子串的**第一个字符的在主串中的序号**。

字符位置：字符在串中的序号。

串相等：串长度相等，且对应位置上字符相等。

串的抽象数据类型定义（参见教材P71）

ADT String{

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

Relations: $R1 = \{ \langle a_{i-1}, a_i \rangle \mid a_{i-1}, a_i \in D, i=2, \dots, n \}$

functions:

最小操作子集 {

- `str_assign(&T, chars)` // 串赋值, 生成值为chars的串T
- `str_compare(S, T)` // 串比较, 若 $S > T$, 返回值大于0, 相等返回0, 否则返回小于0
- `str_length(S)` // 求串长, 即返回S的元素个数
- `concat(&T, S1, S2)` // 串连接, 用T返回S1+S2的新串
- `sub_string(&Sub, S, pos, len)` // 求S中pos起长度为len的子串
-
- `index(S, T, pos)` // 返回子串T在pos之后的位置（模式匹配）
- `replace(&S, T, V)` // 用子串V替换子串T

}ADT String

存储结构与操作实现

串的存储表示方法：

顺序
存储

- 定长顺序存储表示（静态）

——用一组地址连续的存储单元存储串值的字符序列。

- 堆分配存储表示（动态）

——用一组地址连续的存储单元存储串值的字符序列,但存储空间是在程序执行过程中动态分配而得。

链式
存储

- 串的块链存储表示

——链式

定长顺序存储（静态）：

用一组连续的存储单元来存放串，直接使用定长的字符数组来定义，数组的上界预先给出，故称为静态存储分配。

```
#define max_len 255 //用户可用的最大串长
```

```
char sstring[ max_len+1 ];
```

```
//sstring是一个可容纳255个字符的顺序串。
```

注:

- C语言约定在串尾加结束符 ‘\0’, 但不计入串长;
- 一些编程语言约定 `sstring[0]` 来存放串长信息;
- 缺陷: 若字符串超过 `Maxstrlen` 则自动截断

实现方式: 参见教材P73编程两例, 两串连接和求子串

该方式缺陷确实明显

因难以在运行前, 确定最终串的大小

堆分配存储（**其实就是动态分配存储**）
仍用一组连续的存储单元来存放串，但存储空间
是在程序执行过程中动态分配而得。

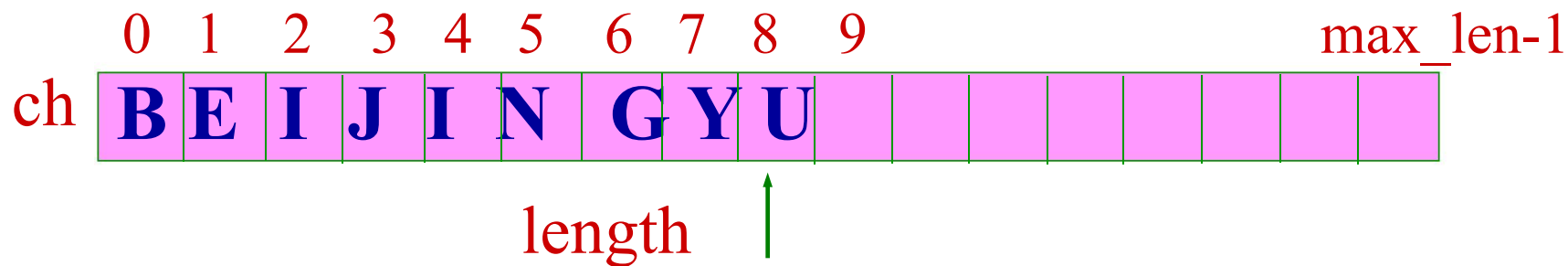
任何时候，均能利用malloc函数合理预设串长空间。

灵活性加强，不需要截断

（话说，一言不合就截断处理也太不负责了哈）

字符串的顺序方式(堆)定义

```
typedef struct {  
    char *ch;           //串的存储数组  
    int length;         //串的当前长度  
} hstring;              //疑问：为何没有容量size?
```



//清空字符串

```
Status clear ( hstring &S){  
    if ( S.ch ) { free(S.ch); S.ch = NULL; }  
    S.length = 0;  
    return OK;  
}
```

```
int str_compare(hstring S, hstring T) {  
    //比较字符串S与T: 若S = T, 返回值 = 0; 若S >  
    // T, 返回值>0; 若S < T, 返回值 <0。  
    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;  
    //或直接用如下C内置函数：  
    //return strcmp(S . ch, T.ch));  
}
```


联接两个串成新串

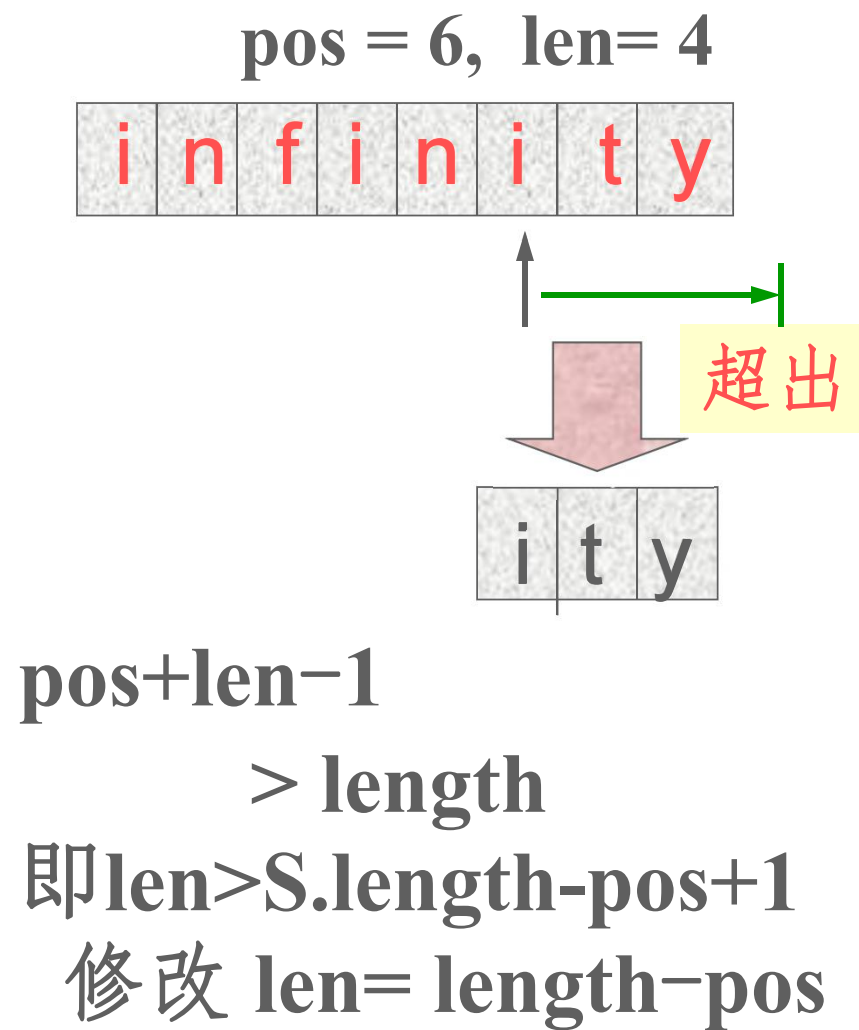
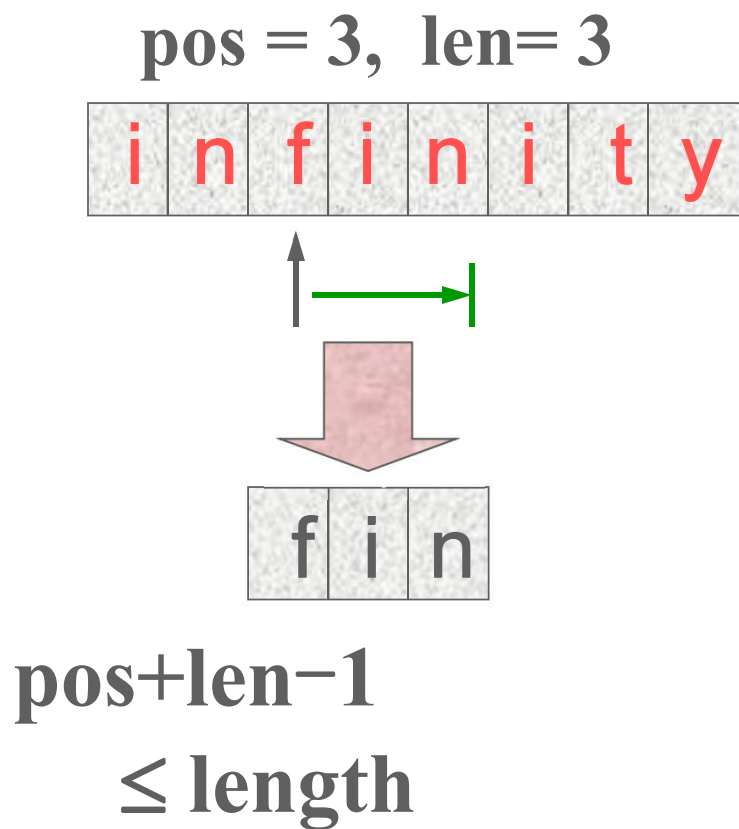
```
Status concat ( hstring &T, hstring S1, sstring S2 )
{ // 用T返回由S1和S2联接而成的新串。
    if (T.ch) free(T.ch);           // 释放旧空间
    if ( !(T.ch = (char *) malloc ((S1.length+S2.length) *
        sizeof (char) ) ) )        exit ( OVERFLOW);
    T.ch[0 .. S1.length-1] = S1.ch[0 .. S1.length-1];
    T.length = S1.length + S2.length ;
    T.ch[S1.length .. T.length-1] = S2.ch[0 .. S2.length-1];
    return OK;
}
```

```
Status sub_string ( hstring &Sub, hstring S,int len )  
{ 用Sub返回串S的第pos个字符起长度为len的子串。  
    边界（前置）条件是？？  
  
if ( pos<1 || pos>S.length || len<0 || len>S.length-pos+1)  
    return ERROR;    // 参数不合法
```

用Sub返回串S的第pos个字符起长度为len的子串。

```
if ( Sub.ch) free ( Sub.ch); // 释放旧空间
if (!len) { Sub.ch = NULL; Sub.length = 0; } //空子串
else {                                     // 完整子串
    Sub.ch = ( char *) malloc ( len *sizeof ( char ));
    Sub.ch[0..len-1] = S.ch [ pos-1.. Pos+len-2];
    Sub.length = len;
}
return OK;
```

提取子串(边界处理)



与顺序表同为线性结构，
串的顺序存储结构与顺序表区别不大（元素限定、操作不同而已）

下一页

链式存储：用链表存储串值，易插入和删除。

法1： 链表结点（数据域）大小取1



法2： 链表结点（数据域）大小取n(例如n=4)



法1存储密度为 $\frac{1}{2}$ ； 法2存储密度为 $\frac{9}{15} = \frac{3}{5}$ ；

显然，若数据元素很多，用法2存储更优，称为块链结构
法2的缺点呢？

块链类型定义:

```
#define CHUNKSIZE 80    //可由用户定义的块大小
typedef struct chunk {   //首先定义结点类型
    char  ch [ CHUNKSIZE ]; //结点中的数据域
    struct chunk * next ;    //结点中的指针域
}chunk;

typedef struct {          //其次定义用链式存储的串类型
    chunk *head;          //头指针
    chunk *tail;          //尾指针----便于链接操作
    int length;           //结点个数
} lstring;
```

再次强调：

串仍然是线性结构

串与线性表的运算有所不同，多以“串的整体”作为操作对象，例如查找某子串，在主串某位置上插入一个子串等。

在C中常用的字符串操作

字符串初始化

```
char name[12] = "BLCU";
```

```
char name[ ] = "BLCU";
```

```
char name[12] = {'B','L','C','U'};
```

```
char name[ ] = {'B','L','C','U','\0'}; //c语言中\0自动加到结尾
```

```
char *name = "BLCU";
```

```
char name[12];
```

```
name = "BLCU"; ?
```

× 因数组名是地址常量

单个字符串的输入函数 gets (str)

例 char name[12]; gets (name);

多个字符串的输入函数 scanf

例 char name1[12], name2[20];

scanf ("%s%s", name1, name2); //回车作为一个s结束

字符串输出函数 puts (str)

例 char name[12]; gets(name);

puts(name);

字符串求长度函数 strlen(str)//一些操作须include <string.h>

字符串长度不包括 “\0”，不包括分界符（引号）

○字符串连接函数 strcat (str1, str2)

例 str1 “BLC\0” //连接前 str2 “U\0” //连接前

str1 “BLCU\0” //连接后 str2 “U\0” //不变

○字符串比较函数 strcmp (str1, str2)

//从两个字符串第 1 个字符开始，逐个对应字符进行比较，全部字符相等则函数返回0，否则在不等字符处停止比较，函数返回其差值 — ASCII代码

例 str1 “University” i 的代码值105

str2 “Universal” a 的代码值97，差8

```
int index (String S, String T, int pos) {  
    //在串S中pos (>0) 位置开始寻找与串T相匹配的子串。若有,  
    //函数返回子串在串中位置 (从1开始), 若无, 返回0。  
    int LT = str_length (T), LS = str_length (S); int i = pos;  
    while (i <= LT-LP+1) {  
        sub_string(&sub, S, i, LT);  
        if (!compare(sub, T)) return i;  
        else i++;  
    }  
    return -1;  
} //用基本操作查找匹配的子串, 复杂度O ( M*N )
```

****串的模式匹配算法****

字符串很多操作中均涉及到**定位问题**，称为**串的模式匹配**。它是串处理系统中最重要的操作之一。

模式匹配(Pattern Matching) 即**子串定位运算**（Index函数）。

算法目的：确定主串中所含子串第一次出现的位置（定位）
——不利用其他操作，实现 $\text{index}(S, T, \text{pos})$ 函数

初始条件： 串S和T存在，T是非空串， $1 \leq \text{pos} \leq \text{length}(s)$

操作结果： 若主串S中存在和串T值相同的子串，则返回它的主串S中第pos个字符之后第一次出现的位置；否则返回值为0。

注： S称为**被匹配的串**，T称为**模式串**。若S包含串T，则称“**匹配成功**”，否则称“**匹配不成功**”。

大家先想想，如何**匹配**呢？

蛮力法/暴力法/朴素法(Brute Force)

S='a b a b c a b c a c b a b'

T= 'a b c a c'

① BF算法设计思想:

- 将主串的第pos个字符和模式的第1个字符比较，
若**相等**，继续逐个比较后续字符；
若**不等**，从主串的下一字符（pos+1）起，重新与子串第一个字符比较。
- 直到主串的一个连续子串字符序列与模式相等 。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 。

第1趟

S a b **b** a b a 3
T a b a 3

第2趟

S a b **b** a b a 2
T a b a 1

第3趟

S a b **b** a b a 3
T a b a 1

第4趟

S a b **b** a b a 4
T a b a

✓


```
int index(sstring S, sstring T, int pos) { //静态为例
```

```
    i=pos;    j=1;
```

```
    while ( i<=S[0] && j<=T[0] ) {
```

```
        if (S[i] == T[j] ) {++i, ++j} //继续比较后续字符
```

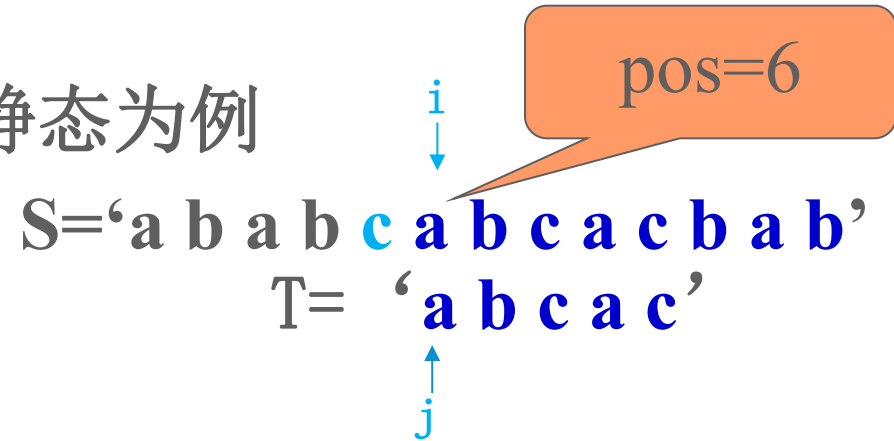
```
        else {i=i-j+2; j=1;} //指针回溯到 下一首位，重新开始匹配
```

```
    }
```

```
    if(j>T[0]) return i-T[0]; //子串结束，说明匹配成功
```

```
    else return 0;
```

```
} // P79  算法4.5
```



时间复杂度分析

$n-m+1$ 个可能匹配位置。至少需要 m 次比较才能知道是否匹配（因长度为 m ）。

最好情况：每次匹配比较，一次就发现不匹配。

假设第一个位置就成功，则比较次数为 m 次

假设第二个位置成功，则 $1+m$ 次比较

假设第三个位置成功，则 $2+m$ 次比较

假设第 i 个位置成功，则 $i+m-1$ 次比较

$i \in [1, n-m+1]$ 。求和，并令概率均等得： $(m+n)/2$ ，则时间复杂度为： $O(m+n)$

等差数列通项公式、求和公式

$$a_n = a_1 + (n-1) \times d$$

$$S_n = na_1 + \frac{n(n-1)}{2}d, n \in N^*$$

时间复杂度分析

最坏情况：若 n 为主串长度， m 为子串长度，则串的BF匹配算法**最坏的情况**下需要比较字符的总次数为：

m

$m+m$

$m+m+m$

...

$m*(n-m+1)$ 由于 $n \gg m$,时间复杂度为: $O(n*m)$

即用基本操作查找匹配的子串的复杂度

可否有更高效率的算法？

BF算法问题分析，指针回溯

改进策略----充分利用已知信息，我们能够已知什么？

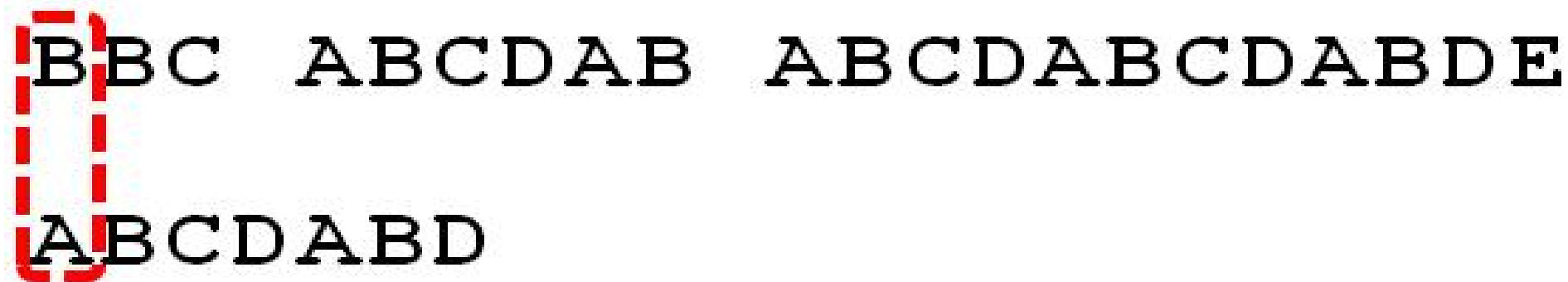
1、

2、

KMP算法为代表 Knuth-Morris-Pratt算法

K-----Donald Knuth

KMP算法简介（精髓：利用已知结构信息，缩短回溯距离）



BBC ABCDAB ABCDABCDABDE
ABCDABD



BBC ABCDAB ABCDABCDABDE
ABCDABD

不匹配的位置处，观察已匹配字符串(s)。

最佳情况：如果字符串s自身没有重复字母，则可以直接跳过已匹配字符串，从不匹配位置进行匹配。

KMP算法简介

“前缀”和“后缀”。“前缀”指除了最后一个字符以外，一个字符串的全部头部组合；“后缀”指除了第一个字符以外，一个字符串的全部尾部组合。

字符串： **“bread”**

前缀： **b , br , bre , brea**

后缀： **read , ead , ad , d**

KMP算法简介

部分匹配模式长度：“前缀”和“后缀”的最长共有元素长度

- “A”的前缀和后缀都为空集，共有元素的长度为0；
- “AB”的前缀为[A]，后缀为[B]，共有元素的长度为0；
- “ABC”的前缀为[A, AB]，后缀为[BC, C]，共有元素的长度0；
- “ABCD”的前缀为[A, AB, ABC]，后缀为[BCD, CD, D]，共有元素的长度为0；

KMP算法简介

- “ABCD A”的前缀为[A, AB, ABC, ABCD], 后缀为[BCDA, CDA, DA, A], 共有元素为“A”, 长度为1;
- “ABCDAB”的前缀为[A, AB, ABC, ABCD, ABCDA], 后缀为[BCDAB, CDAB, DAB, AB, B], 共有元素为“AB”, 长度为2;
- “ABCDABD” 的前缀为[A, AB, ABC, ABCD, ABCDA, ABCDAB], 后缀为[BCDABD, CDABD, DABD, ABD, BD, D], 共有元素的长度为0。

KMP算法简介

BBC ABCDAB ABCDABCDABDE
ABCDABD

搜索词	A	B	C	D	A	B	D
部分匹配值	0	0	0	0	1	2	0

下次子串移动位数 = 已匹配的字符数 - 对应的部分匹配值

BBC ABCDAB ABCDABCDABDE
 ABCDABD

搜索词	A	B	C	D	A	B	D
部分匹配值	0	0	0	0	1	2	0

移动位数4=已匹配的字符数6-对应的部分匹配值2

BBC ABCDAB ABCDABCDABDE
 ABCDABD

移动位数 = 2 - 0 = 2

BBC ABCDAB ABCDABCDABDE
 ABCDABD

每次主串比较位置i不动，只是子串第j位置与其比较

搜索词	A	B	C	D	A	B	D
部分匹配值	0	0	0	0	1	2	0

因此，算法核心是得到子串下一个比较位置j，也就是
 $j = \text{next}(j)$

next函数的生成可形成所有前缀后缀，形成上述匹配表后，
根据移动与j的比较位置间关系，再考虑边界条件后得到。

或者利用教材P83算法4.7

P84算法4.8是一种改进求next函数的方法

KMP算法简介（不作要求）

```
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} //继续比较后续字符  
        else {j=next[j];} //S的i指针不回溯，从T的j位置开  
始匹配}  
        if(j>T[0])    return i-T[0]; //子串结束，说明匹配成功  
        else return 0;  
    }O(m+n)
```

作业（周一晨4点交）：

用队列实现杨辉三角

$n=5$ ，打印出来，不要求形状好看

下周预期进度：递归与栈+迷宫
（含实现递归/非递归）

话说期中考试。。。。

考试范围：

线性结构：

- 1、线性表---顺序表， 链表(有头无头)
- 2、栈---顺序栈、链栈(有头无头)
- 3、队列---顺序栈（循环）、链栈(有头无头)
- 4、应用

```
typedef struct{
    ElemType *elem;    /* 存储空间的首址base */
    int      length;    /* 当前长度 */
    int      size;      /* 当前分配的存储空间 */
}SqList;
```

```
typedef struct {
    ElemType *elem;
    int front, rear;
} SqQueue;
```

```
typedef struct{
    ElemType *base;
    ElemType *top;
    int stacksize;
}SqStack;
```

动态分配, Destroy需要free分配空间

```
typedef struct Node{  
    ElemType data;          /*数据域，保存结点的值*/  
    struct Node *next;      /*指针域*/（双向需两个）  
}Node *LinkList, *LinkStack; /*结点的类型*/
```

```
typedef struct LinkQueue{  
    Node *front;  
    Node *rear;  
}LinkQueue;
```

```
typedef struct STNode{  
    ElemType data;  
    int cursor;  
}STList;  
STList mylist[MAXSIZE];
```

带头节点：初始化时候要分配一个头结点。操作实现简单。

无头节点：初始化无需分配节点。后续操作实现相对麻烦。

实际应用：

0、分析问题的逻辑结构

1、选择合适的逻辑结构

2、分析问题，设计算法

3、根据算法的操作，选择合适的存储结构

算法性能（时空复杂度）