第3章 栈与队列

特殊线性表或受限制的线性表

栈和队列

通常称，栈和队列是限定插入和删除只能在表的“端点”进行的线性表。

线性表	栈	队列
Insert(L, i, x)	Insert(S, n+1, x)	Insert(Q, n+1, x)
1≤i≤n+1		
Delete(L, i)	Delete(S, n)	Delete(Q, 1)
1≤i≤n		

栈和队列是两种常用的数据类型

本章重点与难点

- **重点：**

(1)栈、队列的定义、特点、性质和应用；

(2)ADT栈、ADT队列的设计和实现以及基本操作及相关算法。

- **难点：**

(1)循环队列中对边界条件的处理；

(2)利用栈和队列解决实际问题的应用水平。

第三章 栈和队列

第三章 栈和队列

3.1 栈

3.1.1 抽象数据类型栈的定义

3.1.2 栈的表示和实现

3.2 栈的应用举例

3.2.1 数制转换

3.2.2 括号匹配的检验

3.2.3 行编辑程序问题

3.2.4 表达式求值

3.3 栈与递归的实现

3.4 队列

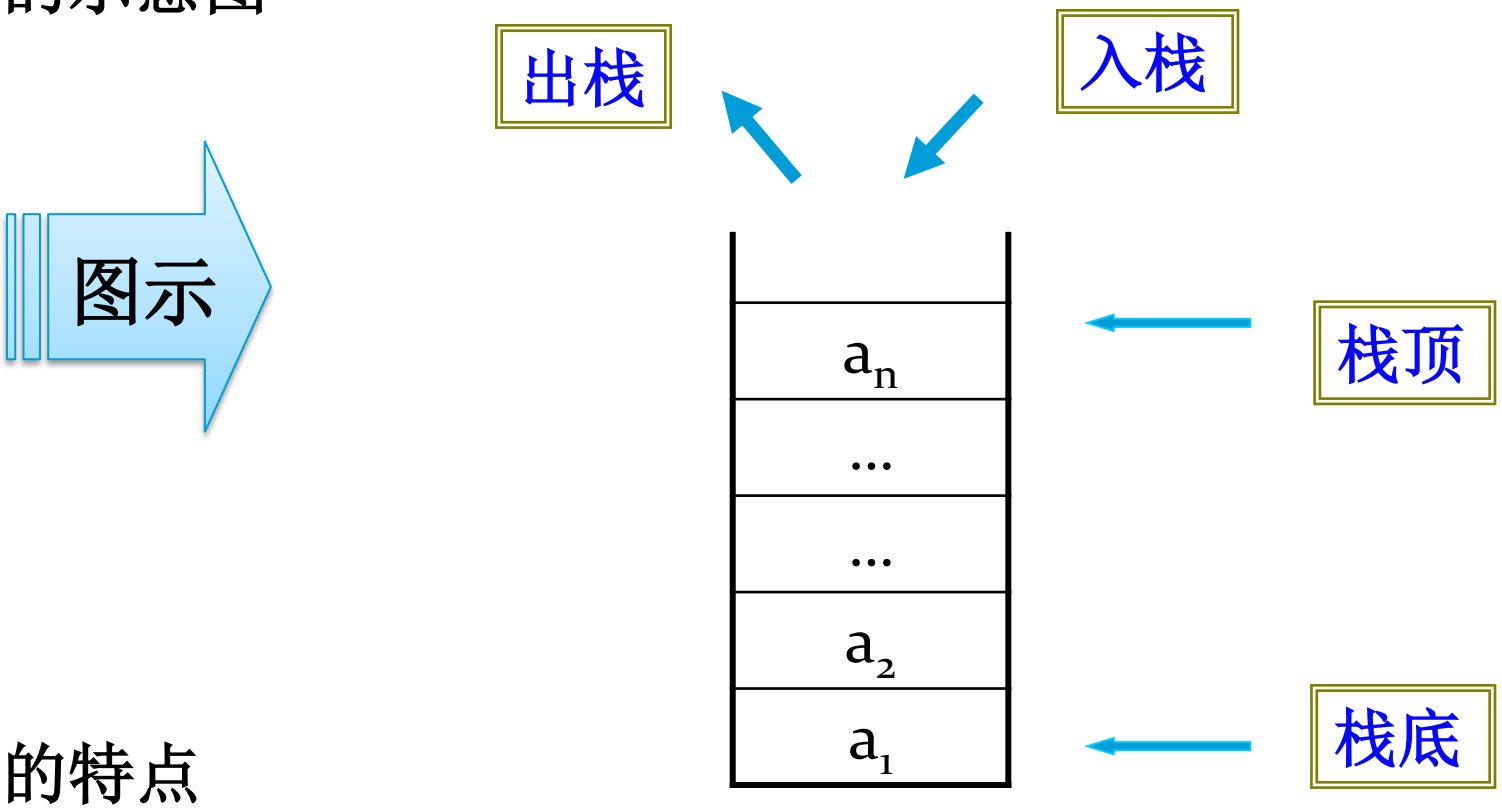
3.1.1 抽象数据类型栈的定义

- 栈的定义
  - 栈(Stack)是一种特殊的线性表，其插入和删除操作均在表的一端进行，是一种运算受限的线性表。

- 栈的术语
  - 栈顶(top)是栈中允许插入和删除的一端。
  - 栈底(bottom)是栈顶的另一端。

3.1.1 抽象数据类型栈的定义

- 栈的示意图



- 栈的特点
  - 后进先出(Last In First Out，简称LIFO)。又称栈为后进先出表(简称LIFO结构)。

3.1.1 抽象数据类型栈的定义

- 抽象数据类型栈

ADT Stack {

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

数据关系：$R1=\{ <a_{i-1}, a_i> \mid a_{i-1}, a_i \in D, i=2,...,n \}$

约定a_n 端为栈顶，a_1 端为栈底。

基本操作：见下页

} ADT Stack

3.1.1 抽象数据类型栈的定义

- 栈的基本操作

InitStack(&S)	//初始化栈
DestroyStack(&S)	//销毁栈
ClearStack(&S)	//清空栈
StackEmpty(S)	//判栈空
StackLength(S)	//求栈长度
GetTop(S, &e)	//取栈顶元素
Push(&S, e)	//入栈
Pop(&S, &e)	//出栈
StackTravers(S, visit())	//遍历栈

3.1.2 栈的表示和实现

- Recall线性表的顺序存储与实现

顺序表的类型定义

```
#define MAXSIZE 1000    // 数组容量
typedef  int  DataType;
Type struct
 { DataType  data[MAXSIZE]; // 数组域
    int  last;          // 线性表长域
 } Seqlist;             // 结构体类型名
```

3.1.2 栈的表示和实现

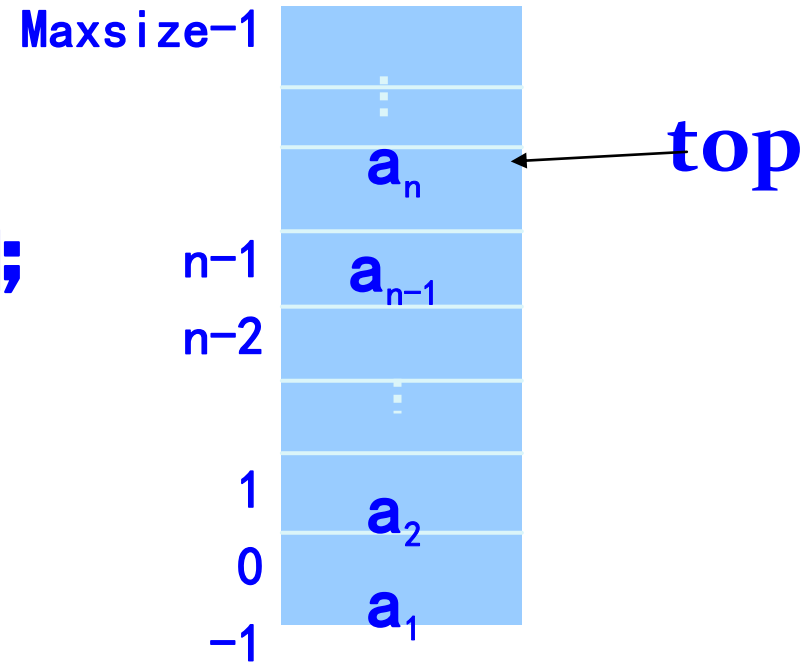
- 顺序栈的C语言实现---栈的数组实现

```
#define Maxsize 1000   // 数组容量

typedef  int  DataType;
```

结构类型：

```
    typedef   struct {
       DataType data[Maxsize];
      int  top ;//栈顶指针
    } SqStack ;
```



3.1.2 栈的表示和实现

(1)置空栈

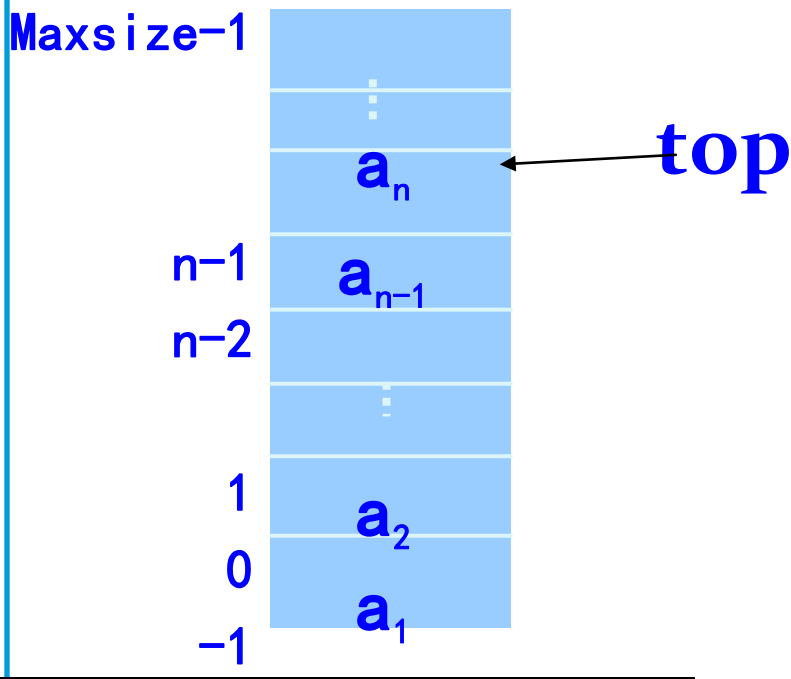
```
void InitStack(SeqStack *s)
{s->top=-1;}
```

(2)判栈空

```
int StackEmpty(SeqStack *s)
{return s->top==-1;}
```

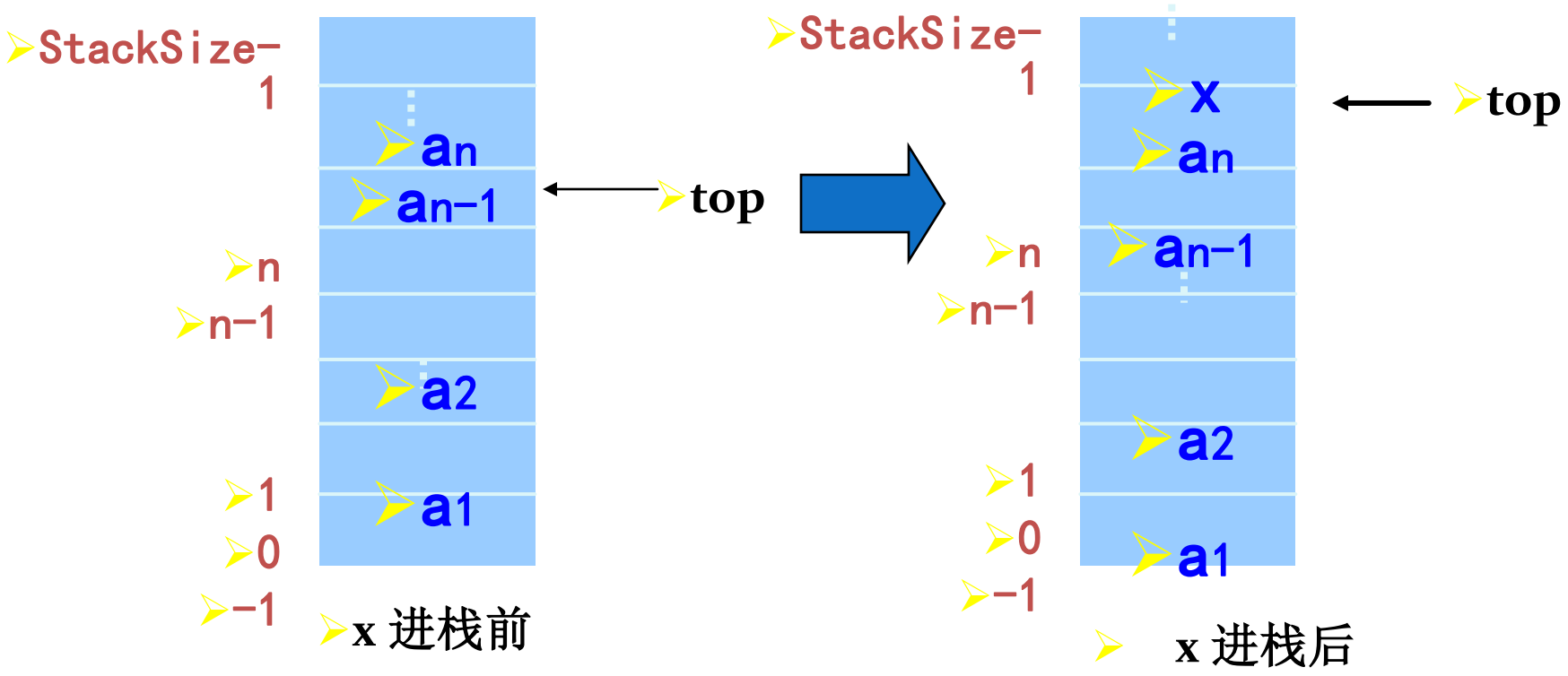
(3)判栈满

```
int StackFull(SeqStack *s)
{return s->top==StackSize-1;}
```



3.1.2 栈的表示和实现

- （4）进栈操作

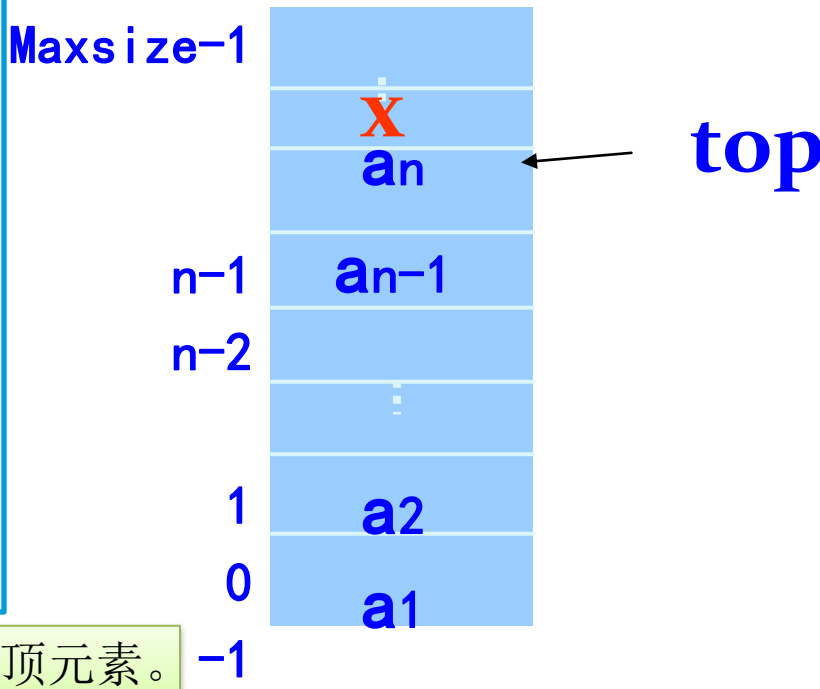


3.1.2 栈的表示和实现

- （4）进栈操作

4) 进栈操作

```
viod Push(SeqStack *s,DataType x )
{if (StackFull(s)   Error(“ overflow” );
   s->top ++;
   s->data[s->top]=x; }
}
```



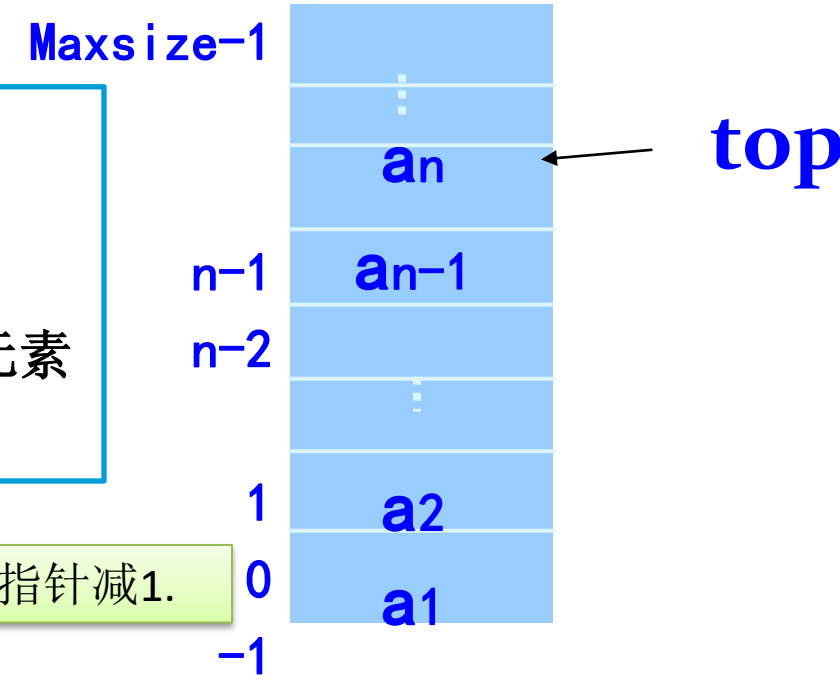
进栈操作：栈不满时，栈顶指针先加1，再送值到栈顶元素。

3.1.2 栈的表示和实现

- （5）出栈操作

```
5）出栈操作
DataType  Pop(SeqStack  *s)
  {if (Stackempty(s))  Error(“underflow”);
  return   s->data[s->top--];//返回当前栈顶元素
}
```

出栈操作：栈非空时，先取栈顶元素值，再将栈顶指针减1.



3.1.2 栈的表示和实现

- 顺序栈的C语言实现

```
//----- 栈的顺序存储表示 -----
#define  STACK_INIT_SIZE  100; //栈容量
#define  STACKINCREMENT   10;  //栈增量
typedef struct {
  SElemType  *base;    //基地址
  SElemType  *top;     //栈顶指针
  int  stacksize;      //栈容量
} SqStack;
```

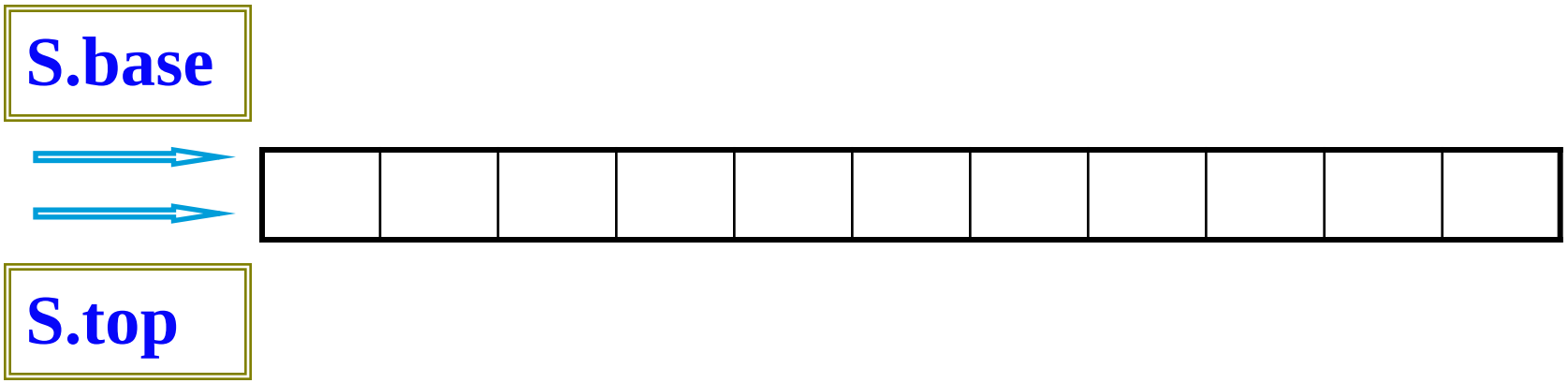
3.1.2 栈的表示和实现

- 栈初始化过程演示

(1) 给栈S申请栈空间

(2) 设置基地址S.base和栈顶地址S.top

(3) 设置栈容量S.stacksize=STACK_INIT_SIZE



3.1.2 栈的表示和实现

- 栈初始化算法

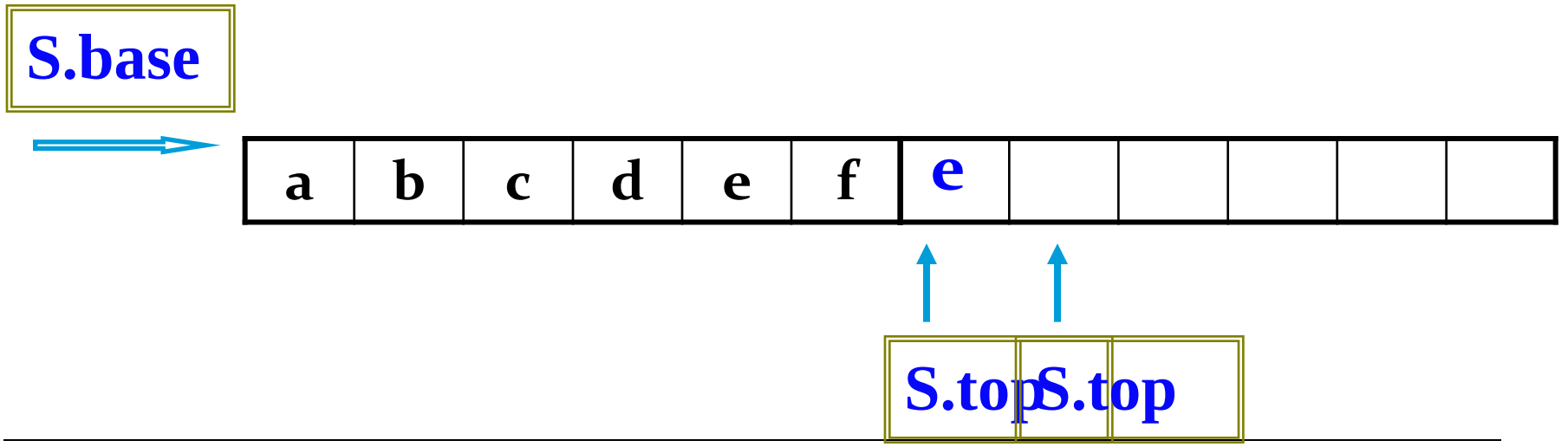
```
Status InitStack (SqStack &S) // 构造一个空栈S
{
  S.base=(ElemType*)malloc(STACK_INIT_SIZE*
                           sizeof(ElemType));
  if (!S.base) exit (OVERFLOW); //存储分配失败
  S.top = S.base;
  S.stacksize = STACK_INIT_SIZE;
  return OK;
}
```

3.1.2 栈的表示和实现

- 入栈操作演示

(1) 如果栈满，给栈增加容量

(2) 将数据存入栈顶位置，栈顶后移一位



3.1.2 栈的表示和实现

- 入栈操作演示

```
Status Push (SqStack &S, SElemType e) {
  if (S.top - S.base >= S.stacksize) //栈满，追加存储空间
    { S.base = (ElemType *) realloc ( S.base,
       (S.stacksize + STACKINCREMENT) *
                                    sizeof (ElemType));
      if (!S.base) exit (OVERFLOW); //存储分配失败
      S.top = S.base + S.stacksize;
      S.stacksize += STACKINCREMENT;
    }
  *S.top++ = e; //先传数据再移动指针
return OK;
}
```

3.1.2 栈的表示和实现

- 其它栈操作讨论

DestroyStack(&S)	//销毁栈
ClearStack(&S)	//清空栈
StackEmpty(S)	//判栈空
StackLength(S)	//求栈长度
GetTop(S, &e)	//取栈顶元素
Pop(&S, &e)	//出栈
StackTravers(S, visit())	//遍历栈

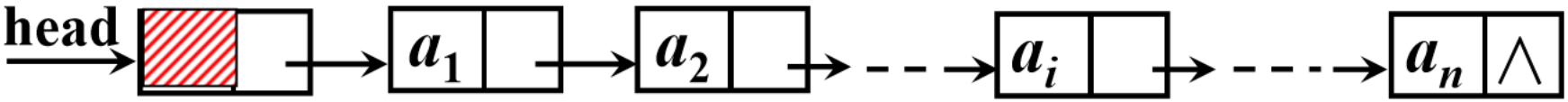
3.1.2 栈的表示和实现

- 栈的指针实现---链栈

✓ 栈的链接存储结构及实现

■ 链栈：栈的链接存储结构

■ 如何改造链表实现栈的链接存储？



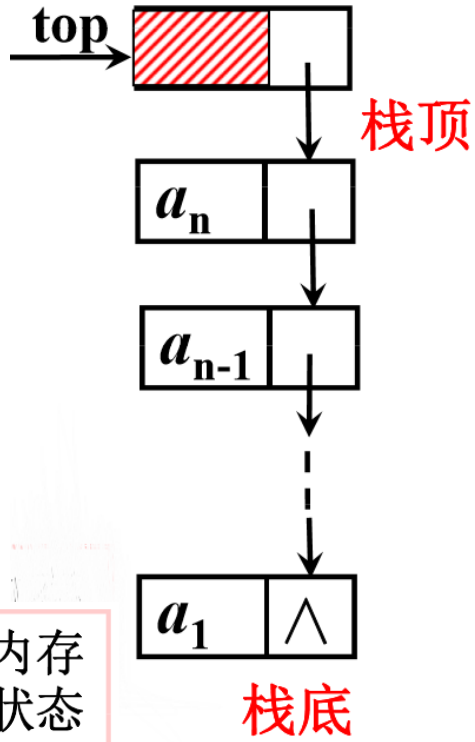
■ 将哪一端作为栈顶？将链表首端作为栈顶，方便操作。

■ 链栈需要加头结点吗？通常采用单链表实现，并规定所有操作都是在单链表的表头进行的。如果链栈没有头结点，Lhead指向栈顶元素。

3.1.2 栈的表示和实现

- 栈的指针实现---链栈

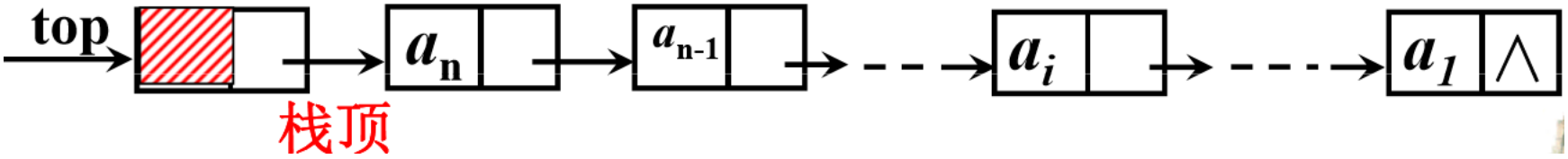
链栈的实现与链表的实现基本相同，头结点作为栈顶位置。在此不做详细讨论。需要注意的是，对于带头结点和不带头结点的链栈，在具体的实现方面有所不同。



链栈的C语言类型定义

```
Typedef struct  SNode {
    ElemType    data;      //数据域
    struct Snode *next;    //链域
}SNode, *LinkStack;
```

两种示意图在内存中对应同一种状态



3.1.2 栈的表示和实现

- 讨论链栈基本操作的实现

InitStack(&S)	//初始化栈
DestroyStack(&S)	//销毁栈
ClearStack(&S)	//清空栈
StackEmpty(S)	//判栈空
StackLength(S)	//求栈长度
GetTop(S, &e)	//取栈顶元素
Push(&S, e)	//入栈
Pop(&S, &e)	//出栈
StackTravers(S, visit())	//遍历栈

第三章 栈和队列

3.2.1 数制转换

- 数制转换----是计算机实现计算的基本问题。

例 将10进制1346转换成8进制

✓方法：除留余数法。

过程如下：

计算顺序

N	N div 8	N mod 8
1348	168	4
168	21	0
21	2	5
2	0	2

输出顺序

输出的时候是从高位到低位进行，恰好和计算相反。

转换的过程满足栈的后进先出的特点

3.2.1 数制转换

- 10进制数N转换成8进制的算法

```
void conversion () {
    InitStack(S); scanf (“%d”,N);//输入一个任意非负十进制整数
    while (N) {
      Push(S, N % 8); N = N/8;          //余数入栈
    }
    while (!StackEmpty(S)) {
      Pop(S,e);//若栈不空，则删除S的栈顶元素，用e返回其值，并返回ok，否则error
      printf ( "%d", e );
    }
} // conversion
```

3.2.2 括号匹配的检验

- 问题描述
  - 一个表达式中，包含三种括号“(”和“)”，“[”和“]”和“{”和“}”，这三种括号可以按任意的合法次序使用。设计算法检验表达式中所使用括号的合法性。
- 问题讨论

讨论：如果第一次遇到的右括号是“]”，那么前面出现的左括号有什么特点。

结论：如果第一次遇到的右括号是“]”，那么前面出现的左括号最后一个必然是“[”，否则不合法。

3.2.2 括号匹配的检验

例在表达式中正确的格式：

（[]（））或[（[] []）]

例在表达式中不正确的格式

- [（]）----左右不匹配
- （[（））---左括号多了
- （（）]）----右括号多了

检验括号是否匹配的方法可用“期待的急迫程度”这个概念来描述。

3.2.2 括号匹配的检验

- 算法过程

[([ ] [ ])]

1）凡出现左括弧，则进栈；

2）凡出现右括弧，首先检查栈是否空

若栈空，则表明该“右括弧”多余，

否则和栈顶元素比较，

若相匹配，则“左括弧出栈”，

否则表明不匹配。

3）表达式检验结束时，

若栈空，则表明表达式中匹配正确，

否则表明“左括弧”有余。

3.2.2 括号匹配的检验

- 括号匹配检验算法

```
Status  check( ) {
   char ch;  InitStack(S);
   while ((ch=getchar())!=‘#') {
      switch (ch) {
         case (ch=='('||ch=='['||ch=='{'):Push(S,ch);break;
         case (ch== ')'):
             if (StackEmpty(S)) return FALSE;
             else
                {Pop(S,e);if(e!= '(') return FALSE;}
             break;
```

3.2.2 括号匹配的检验

- 括号匹配检验算法

```
case (ch== ‘]'):
  if (StackEmpty(S)) retrun FALSE;
  else
  {Pop(S,e);if(e!= ‘[') return FALSE;}
   break;…….
 default:break;
}
}
if (StackEmpty(S))  return TRUE;
else return FALSE;
}
```

3.2.3 行编辑程序问题

- **问题描述**
  - 在用户输入一行的过程中，允许用户输入出差错，并在发现有误时可以及时更正。

- **解决方法**

设立一个输入缓冲区，用以接受用户输入的一行字符，然后逐行存入用户数据区，并假设“#”为退格符（当用户发现刚刚键入的一个字符是错的，可补进一个退格符，以表示前一个字符无效），“@”为退行符（当用户发现当前键入的行内差错较多或难以补救时）。

3.2.3 行编辑程序问题

例 假设从终端接受了这样两行字符：

```
whli##ilr#e（s#*s)
  outcha@putchar(*s=#++);
```

则实际有效的是下列两行：

```
while (*s)
  putchar(*s++);
```

3.2.3 行编辑程序问题

- 行编辑问题算法

```
void LineEdit(){
  //利用字符栈S，从终端接收一行并传送至调
  //用过程的数据区
    InitStack(S);
    ch=getchar();
    while (ch != EOF) { //EOF为全文结束符
      while (ch != EOF && ch != '\n') {……}
     ……
    DestroyStack(S);
}
```

3.2.3 行编辑程序问题

- 行编辑问题算法

```
switch (ch) {
   case '#' : Pop(S, c); break;
   case '@': ClearStack(S); break;// 重置S为空栈
   default : Push(S, ch);  break;
  }
  ch = getchar();          // 从终端接收下一个字符
```

将从栈底到栈顶的栈内字符传送至调用过程的数据区；

```
ClearStack(S);          // 重置S为空栈
if (ch != EOF)  ch = getchar();
```

3.2.4 表达式求值

限于二元运算符的表达式定义：

表达式 ::= (操作数) + (运算符) + (操作数)

操作数::= 简单变量|表达式

简单变量::=标识符|无符号整数

3.2.4 表达式求值

例 求表达式3*(2+3*5)+6的值

表达式 ::= (操作数) + (运算符) + (操作数)

表达式的三种标识方法：

设 Exp = S1 + OP + S2

则称 OP + S1 + S2 为前缀表示法

S1 + OP + S2 为中缀表示法

S1 + S2 + OP 为后缀表示法

它以运算符的位置来命名。

四则运算规则：

（1）先乘除，后加减；

（2）从左算到右；

（3）先括号内，后括号外。

例如： Exp = a × b + (c – d / e) × f

前缀式： + × a b × – c / d e f

中缀式： a × b + c – d / e × f

后缀式： a b × c d e / – f × +

3.2.4 表达式求值

- 结论

1）失去了括号，操作数之间的相对次序不变（a，b，c，d，e，f）；

2）运算符的相对次序不同；（中缀不变）

3）中缀式丢失了括弧信息，致使运算的次序不确定；

4) 前缀式的运算规则为：　　+ × a b × – c / d e f

□连续出现的两个操作数和在它们之前且紧靠它们的运算符构成一个最小表达式；

□前缀式唯一确定了运算顺序；

5) 后缀式的运算规则为：　　a b × c d e / – f × +

□每个运算符和在它之前出现且紧靠它的两个操作数构成一个最小表达式。

□先找运算符，再找操作数（后缀式求值算法基本依据）。

□操作数的顺序不变。

□运算符在式中出现的顺序恰为表达式的运算顺序。

3.2.4 表达式求值

- 如何从原表达式求得后缀式？

分析“原表达式”和“后缀式”中的运算符:

原表达式: a + b × c – d / e × f

后缀式: a b c × + d e / f × –

每个运算符的运算次序要由它之后的一个运算符来定，在后缀式中，优先数高的运算符领先于优先数低的运算符。

3.2.4 表达式求值

- 如何从原表达式求得后缀式？

分析“原表达式”和“后缀式”中的运算符：

原表达式：a + b × c – d / e × f

后缀式：a b c × + d e / f × –

运算符栈：　+/×　←比较优先级　-×

后缀式：a　b　c　　d　e　f

3.2.4 表达式求值

- 如何从原表达式求得后缀式？

分析“原表达式”和“后缀式”中的运算符：

原表达式：a + b × c – d / e × f

后缀式：a b c × + d e / f × –

运算符栈：- ×

比较优先级

后缀式：a b c× + d e/f

3.2.4 表达式求值

- 从原表达式求得后缀式的规律

1) 设立运算符栈；

2) 设表达式的结束符为“#”，予设运算符栈的栈底为“#”；

3) 若当前字符是操作数，则直接发送给后缀式。

4) 若当前运算符的优先数高于栈顶运算符，则进栈；

5) 否则，退出栈顶运算符发送给后缀式；

6) “(”对它之前后的运算符起隔离作用，“)”可视为自相应左括弧开始的表达式的结束符。

3.2.4 表达式求值

- 算法求解过程：

设置两个栈，一个存操作数，栈名为OPND，一个存运算符，栈名为OPTR栈。

(1) 首先置操作数栈为空，表达式起始符#为运算符栈的栈底元素；

(2)依次读入表达式中每个字符，若是操作数则进OPND栈，若是运算符则和OPTR栈的栈顶运算符比较优先权后作相应操作，直到整个表达式操作完毕。

3.2.4 表达式求值

- 算法求解过程：

(3)若栈顶运算符小于输入运算符，输入运算符进栈；

(4)若栈顶运算符等于输入运算符（只有栈顶是“（”，输入是“）”，或者栈顶是“#”，输入是“#）”两种情况，分别去除一对括号，或结束。

(5)若栈顶运算符大于输入运算符，弹出栈顶运算符，从OPND中弹出两个操作数与弹出运算符计算后再存入OPND栈，继续。

第三章 栈和队列

3.3 栈与递归的实现

- 递归调用的定义：
  - 子程序（或函数）直接调用自己或通过一系列调用语句间接调用自己。是一种描述问题和解决问题的基本方法。
- 递归的基本思想：
  - 把一个不能或不好求解的大问题转化为一个或几个小问题，再把这些小问题进一步分解成更小的小问题，直至每个小问题都可以直接求解。
- 递归的要素：
  - 递归边界条件：确定递归到何时终止，也称为递归出口；
  - 递归模式：大问题是如何分解为小问题的，也称为递归体
- 递归的精髓在于：能否将原始问题转换为属性相同但规模较小的问题。

3.3 栈与递归的实现

- 什么问题可以应用递归：
  - 一类是递归函数；
  - 第二类是，有的数据结构，如二叉树，广义表等，由于结构本身固有的递归特性，则它们的操作可递归地描述；
  - 还有一类问题，虽然问题本身没有明显的递归结构，但用递归求解比迭代求解更简单，如八皇后问题、Hanoi塔问题等。

3.3 栈与递归的实现

- 实现递归：
  - 当在一个函数的运行期间调用另一个函数时，在运行该被调用函数之前，需先完成三件事：
    - 将所有的实在参数、返回地址等信息传递给被调用函数保存；
    - 为被调用函数的局部变量分配存储区；
    - 将控制转移到被调函数的入口。

3.3 栈与递归的实现

- 实现递归：
  - 从被调用函数返回调用函数之前，应该完成：
    - 保持被调用函数的计算结果；
    - 释放被调用函数的数据区；
    - 依照被调用函数保存的返回地址将控制转移到调用函数。
  - 多个函数嵌套调用的规则是：
    - 后调用先返回；
    - 此时的内存管理实行“栈式管理”

3.3 栈与递归的实现

- 递归调用举例：
  - 求阶乘的函数

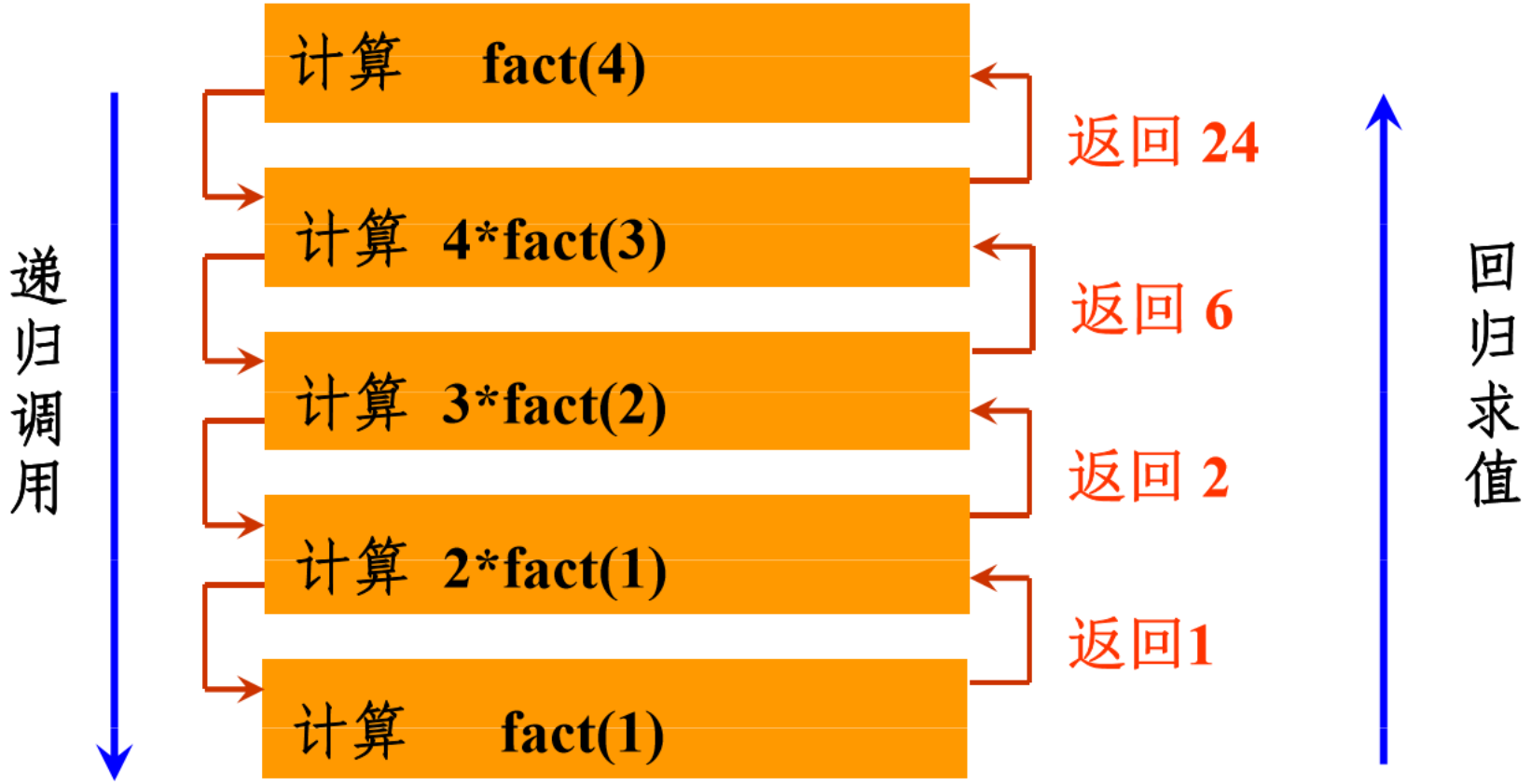
$$n! = \begin{cases} 1 & \text{当 } n=1 \text{ 时} \\ n * (n-1)! & \text{当 } n \geq 1 \text{ 时} \end{cases}$$

  - 递归算法

```
long fact  ( int n )
{
    if ( n == 0 ) return 1;
    else   return  n * fact (n-1);
}
```

3.3 栈与递归的实现

- 递归调用举例：
  - 求解阶乘n！的过程



3.3 栈与递归的实现

- 递归过程与递归工作栈：
  - 递归过程在实现时，需要自己直接或间接调用自己。
  - 层层向下递归，返回次序正好相反：

3.3 栈与递归的实现

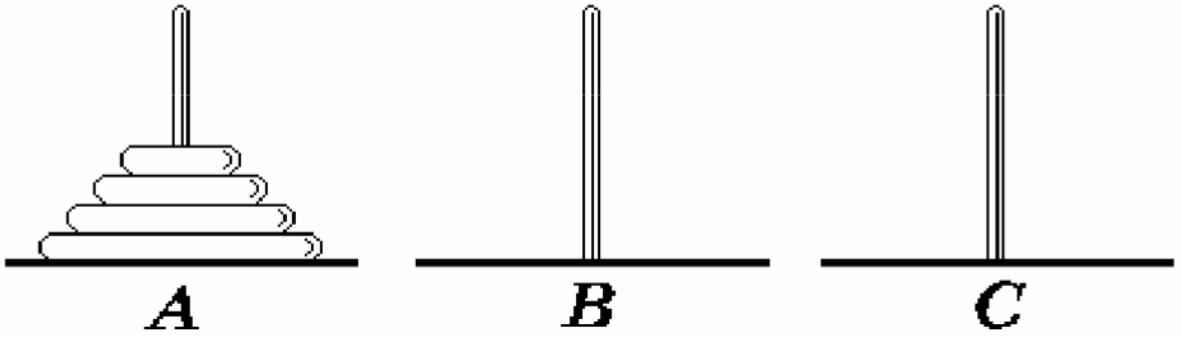
- 递归过程与递归工作记录：
  - 每一次递归调用时，需要为过程中使用的参数、局部变量和返回地址等另外分配存储空间。
  - 每层递归调用需分配的空间形成递归工作记录，按后进先出的栈组织。

3.3 栈与递归的实现

- 递归函数的内部执行过程：
  - (1)运行开始时，首先为递归调用建立一个工作栈，其结构包括值参、局部变量和返回地址。
  - (2)每次执行递归调用之前，把递归函数的值参和局部变量的当前值以及调用后的返回地址压栈。
  - (3)每次递归调用结束后，将栈顶元素出栈，使相应的值参和局部变量恢复为调用前的值，然后转向返回地址指定的 位置继续执行。

3.3 栈与递归的实现

- 汉诺塔问题——递归的经典问题
  - 在世界刚被创建的时候有一座钻石宝塔（塔A），其上有 64个金碟。所有碟子按从大到小的次序从塔底堆放至塔顶 。紧挨着这座塔有另外两个钻石宝塔（塔B和塔C）。从 世界创始之日起，婆罗门的牧师们就一直在试图把塔A上 的碟子移动到塔C上去，其间借助于塔B的帮助。每次只能移动一个碟子，任何时候都不能把一个碟子放在比它小的碟子上面。当牧师们完成任务时，世界末日也就到了。



3.3 栈与递归的实现

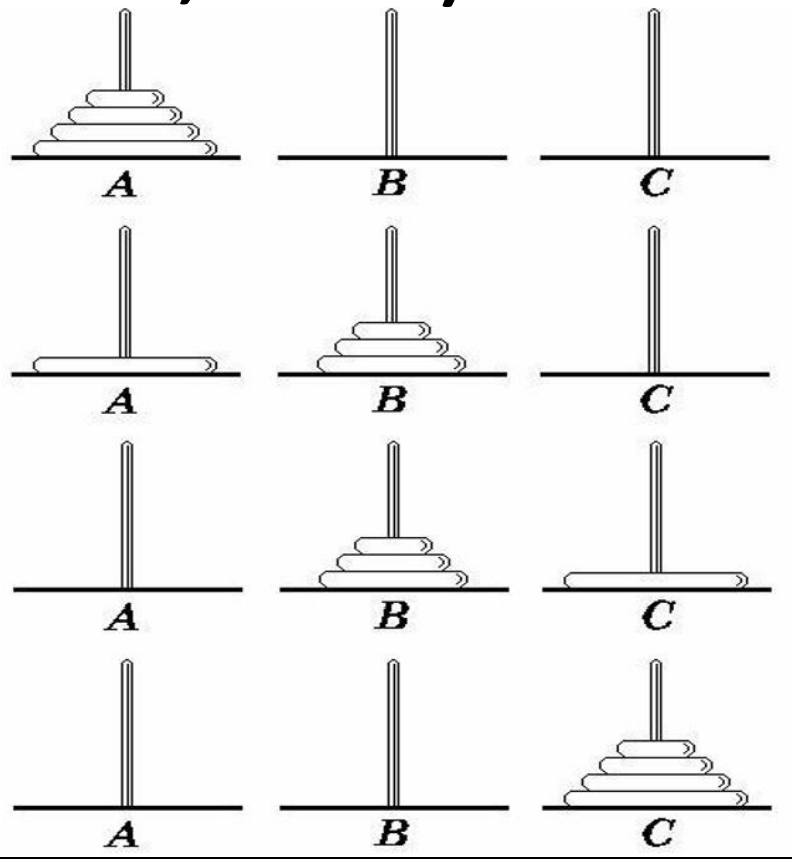
- 汉诺塔问题的递归求解:
  - 如果n=1，则将这一个盘子直接从塔A移到塔C上。
  - 否则，执行以下三步：
    - 将塔A上的n-1个碟子借助塔C先移到塔B上
    - 把塔A上剩下的一个碟子移到塔C上；
    - 将n-1个碟子从塔B借助于塔A移到塔C上。

说明：n个直径大小各不相同、依小到大编号为1，2，…，n的圆盘。

3.3 栈与递归的实现

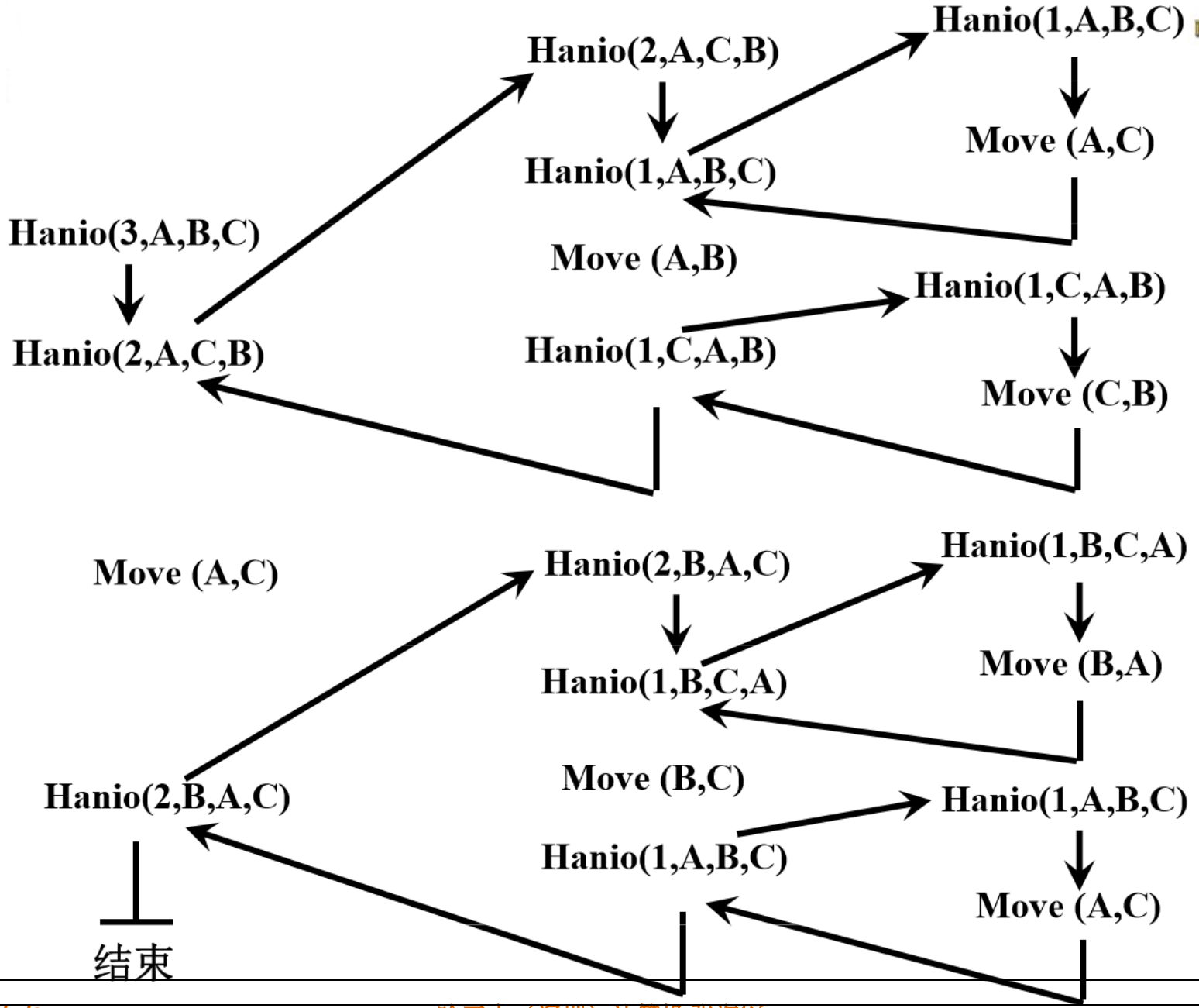
- 汉诺塔问题的递归求解:

```
void Hanoi(int n, char A, char B, char C)
{
    if (n==1)  Move(A, C);
     else {
            Hanoi(n-1, A, C, B);
            Move(A, C);
            Hanoi(n-1, B, A, C);
     }
}
```



3.3 栈与递归的实现

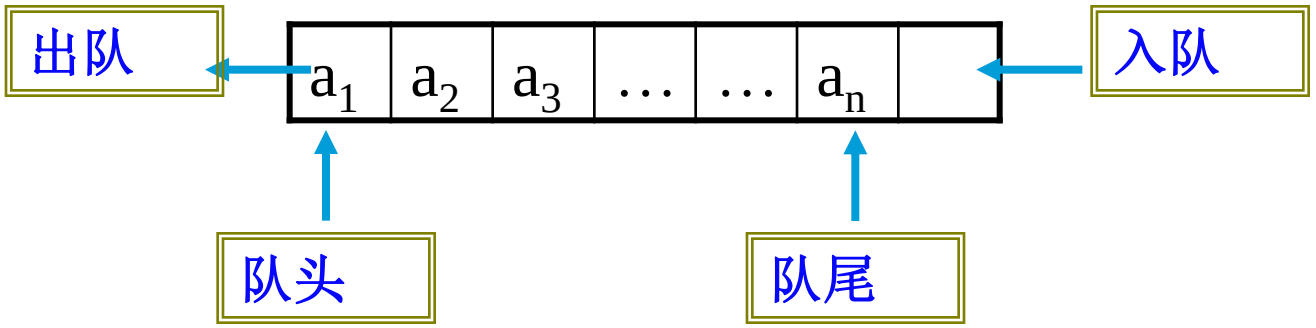
- 递归函数的运行轨迹
  - 写出函数当前调用层执行的各语句，并用有向弧表示语句的执行次序;
  - 对函数的每个递归调用，写出对应的函数调用，从调用处画一条有向弧指向被调用函数入口，表示调用路线，从被调用函数末尾处画一条有向弧指向调用语句的下面，表示返回路线;
  - 在返回路线上标出本层调用所得的函数值。


Hanio(1,A,B,C)
Hanio(2,A,C,B)
Move (A,C)
Hanio(1,A,B,C)
Hanio(3,A,B,C)
Move (A,B)
Hanio(1,C,A,B)
Hanio(2,A,C,B)
Hanio(1,C,A,B)
Move (C,B)
Hanio(1,B,C,A)
Move (A,C)
Hanio(2,B,A,C)
Move (B,A)
Hanio(1,B,C,A)
Move (B,C)
Hanio(1,A,B,C)
Hanio(2,B,A,C)
Hanio(1,A,B,C)
Move (A,C)
结束

第三章 栈和队列

3.4 特殊的线性表---队列

- 队列的定义：
  - 队列(Queue)——是一种运算受限的特殊的线性表，它只允许在表的一端进行插入，而在表的另一端进行删除。
- 队列的术语：
  - 队尾(rear)是队列中允许插入的一端。
  - 队头(front)是队列中允许删除的一端。
- 队列示意图：



- 队列的特点：
  - 先进先出(First In First Out，简称FIFO)。又称队列为先进先出表。

3.4.1 抽象数据类型队列的定义

- 抽象数据类型队列：

ADT Queue {

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

数据关系：$R1=\{ <a_{i-1},a_i> \mid a_{i-1}, a_i \in D,\ i=2,...,n\}$

约定其中a_1端为队列头，a_n端为队列尾

基本操作：

见下页

} ADT Queue

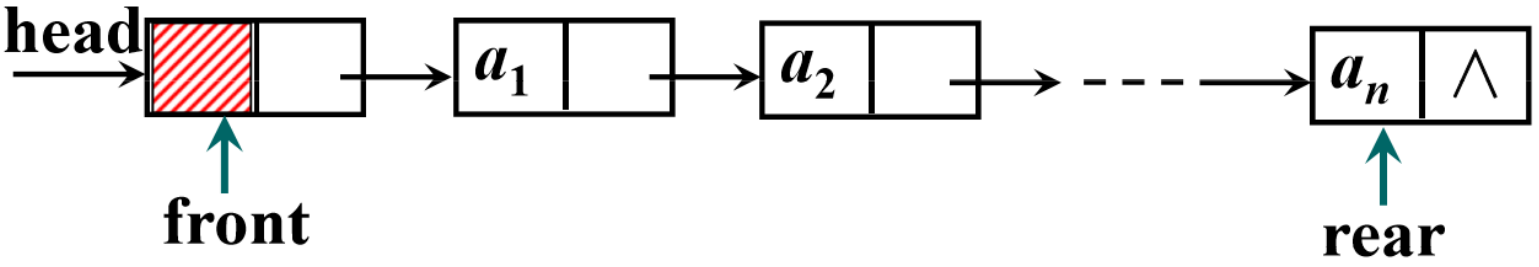
3.4.1 抽象数据类型队列的定义

- 队列的基本操作：

```
InitQueue(&Q)                  //初始化队列
DestroyQueue(&Q)               //销毁队列
QueueEmpty(Q)                  //判队列是否空
QueueLength(Q)                 //求队列长度
GetHead(Q, &e)                 //取队头元素
ClearQueue(&Q)                 //清空队列
EnQueue(&Q, e)                 //入队列
DeQueue(&Q, &e)                //出队列
QueueTravers(Q, visit())       //遍历队列
```

3.4.2 队列的指针实现---链队列

- 队列的链接存储结构及实现：
  - 链队列：队列的链接存储结构
  - 如何改造单链表实现队列的链接存储？



  - 队首指针即为链表的头结点指针
  - 增加一个指向队尾结点的指针

3.4.2 队列的指针实现---链队列

- 链队列的结点实现：

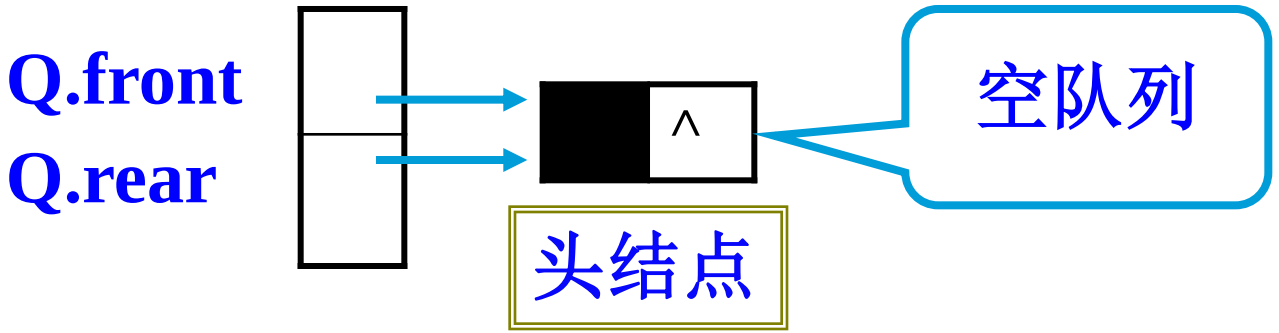
```
typedef struct QNode { // 结点类型
  QElemType    data;
  struct QNode  *next;
} QNode, *QueuePtr;
```

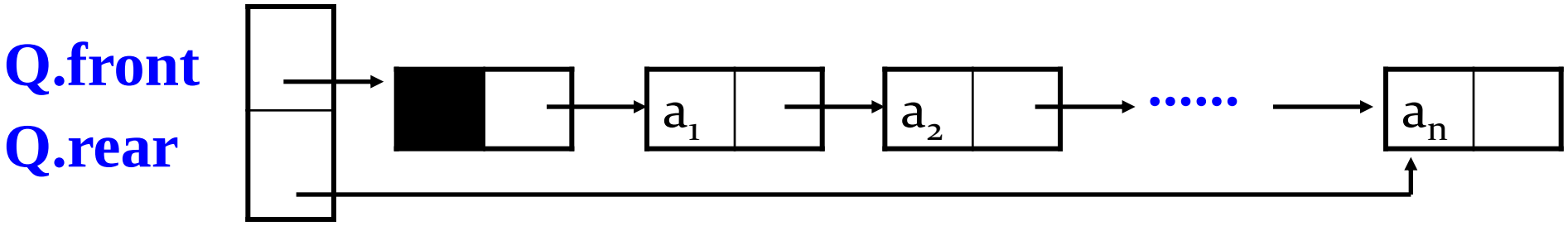
- 链队列数据类型实现：

```
typedef struct {          // 链队列类型
   QueuePtr  front;   // 队头指针
   QueuePtr  rear;    // 队尾指针
} LinkQueue;
```

3.4.2 队列的指针实现---链队列

- 带头结点的链队列示意图：





3.4.2 队列的指针实现---链队列

- 带头结点的链队列初始化：

```
Status InitQueue (LinkQueue &Q) {
                                    // 构造一个空队列Q
  Q.front = Q.rear =
          (QueuePtr)malloc(sizeof(QNode));
  if (!Q.front) exit (OVERFLOW);
                                    //存储分配失败
  Q.front->next = NULL;
  return OK;
}
```

3.4.2 队列的指针实现---链队列

- 带头结点的链队列入队算法：

```
Status EnQueue (LinkQueue &Q, QElemType e)
{   // 插入元素e为Q的新的队尾元素
    p = (QueuePtr) malloc (sizeof (QNode));
    if (!p)  exit (OVERFLOW);   //存储分配失败
    p->data = e;   p->next = NULL;
    Q.rear->next = p;    Q.rear = p;
    return OK;
}
```

3.4.2 队列的指针实现---链队列

- 带头结点的链队列出队算法：

```
Status DeQueue (LinkQueue &Q, QElemType &e)
{ // 若队列不空，则删除Q的队头元素，
  //用 e 返回其值，并返回OK；否则返回ERROR
  if (Q.front == Q.rear)    return ERROR;
  p = Q.front->next;   e = p->data;
  Q.front->next = p->next;
  if (Q.rear == p)  Q.rear = Q.front;
  free (p);      return OK;
}//注意当只有一个结点时，记得队尾指针要指向队头
```

3.4.2 队列的指针实现---链队列

- 带头结点的链队列其它操作讨论：

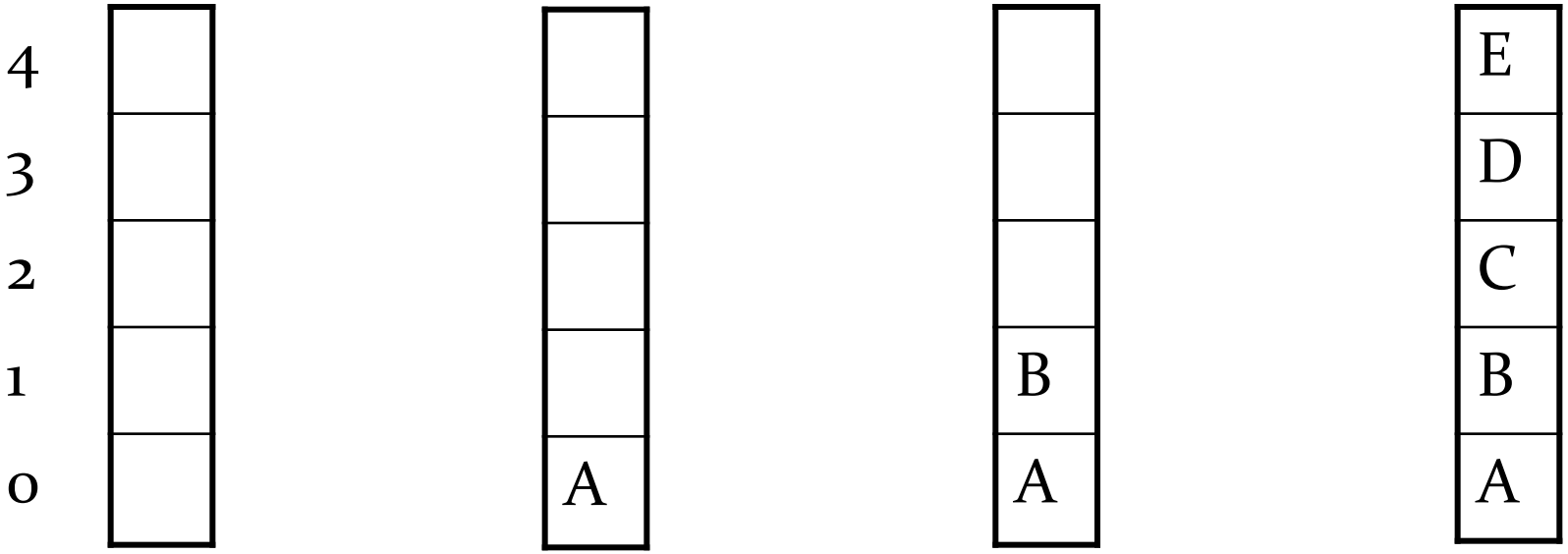
DestroyQueue(&Q)	//销毁队列
QueueEmpty(Q)	//判队列是否空
QueueLength(Q)	//求队列长度
GetHead(Q, &e)	//取队头元素
ClearQueue(&Q)	//清空队列
QueueTravers(Q, visit())	//遍历队列

3.4.3 队列的顺序表示及实现

- 顺序队列讨论：

循环队列属于顺序队列的一种，讨论在采用一般顺序队列时出现的问题。

我们约定初始化建空队列时，令front=rear=0，每当插入新的队列尾元素时，尾指针增加1；每当删除队列头元素时，头指针增加1.因此，在非空队列中，头指针始终指向队列头元素，而尾指针始终指向队列尾元素的下一个位置。



Sq.rear=0
Sq.front=0

Sq.rear=1
Sq.front=0

Sq.rear=2
Sq.front=0

Sq.rear=5
Sq.front=0

3.4.3 队列的顺序表示及实现---循环队列

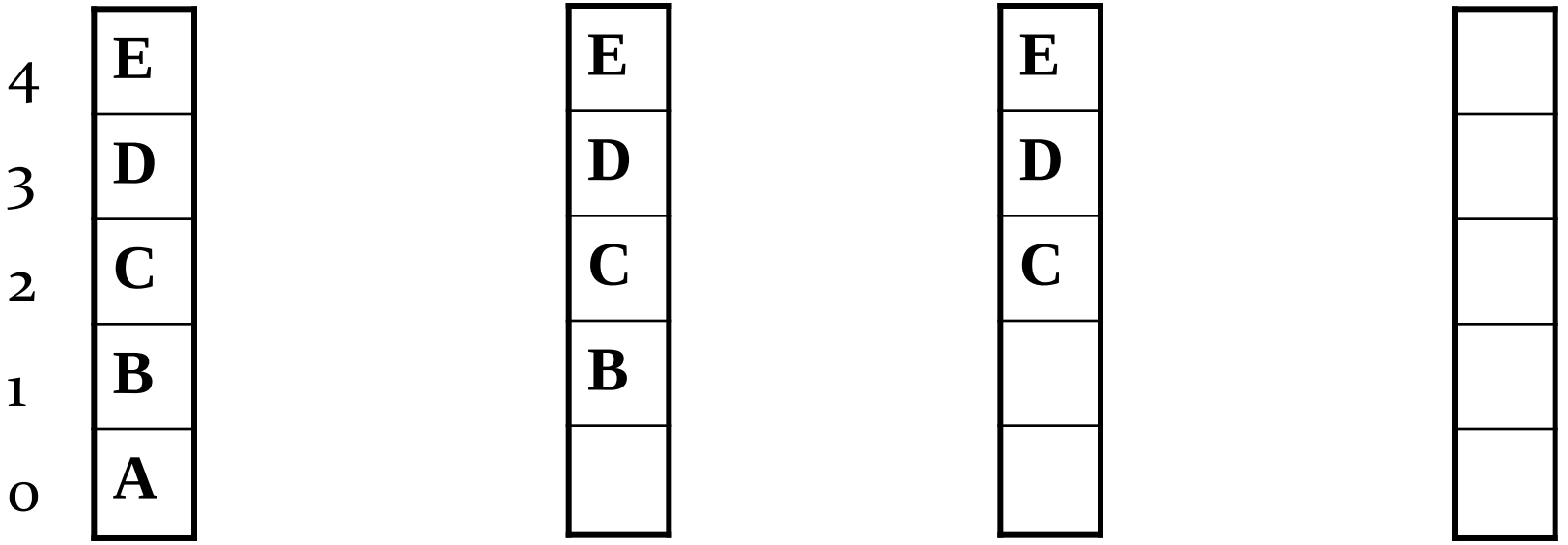
- 顺序队列数据类型实现：

```
#define MAXQSIZE  100  //最大队列长度
  typedef struct {
    QElemType  *base;  //初始化的 动态分配存储空间
    int  front;     // 头指针，若队列不空，
                    //  指向队列头元素
    int  rear;     // 尾指针，若队列不空，指向
                      队列尾元素 的下一个位置
  } SqQueue;
```

这里和栈相比，没有一个值表明最大长度；栈是可以在空间不够时动态再分配的，但是顺序队列不能再分配。

3.4.3 队列的顺序表示及实现

- 顺序队列讨论：



Sq.rear=5
sq.front=0

Sq.rear=5
Sq.front=1

Sq.rear=5
Sq.front=2

Sq.rear=5
Sq.front=5

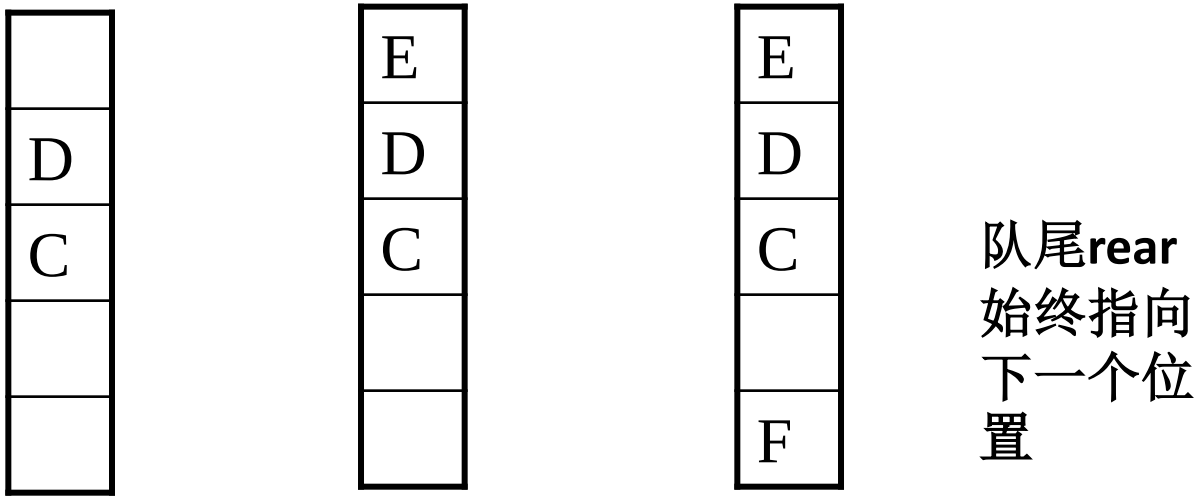
讨论结论：在采用一般顺序队列时出现假上溢现象？

当元素被插入到数组中下标最大的位置上之后，队列的空间就用尽了，但此时数组的低端还有空闲空间，这种现象叫做假溢出。

3.4.3 队列的顺序表示及实现---循环队列

- 循环队列的定义：

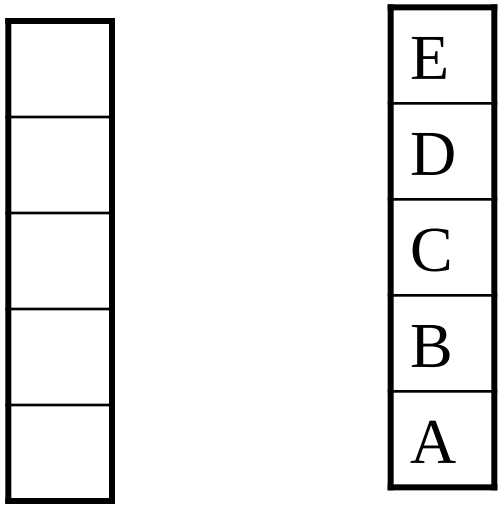
循环队列是顺序队列的一种特例，它是把顺序队列构造成一个首尾相连的循环表。指针和队列元素之间关系不变。



3.4.3 队列的顺序表示及实现---循环队列

- 循环队列空状态和满状态的讨论：

空状态　　满状态



讨论结论：循环队列空状态和满状态都满足
Q.front=Q.rear

Sq.rear=0　　**Sq.rear=0**
Sq.front=0　　**Sq.front=0**

3.4.3 队列的顺序表示及实现---循环队列

- 循环队列空状态和满状态的判别：

(1) 另设一个标志区别队列是空还是满；

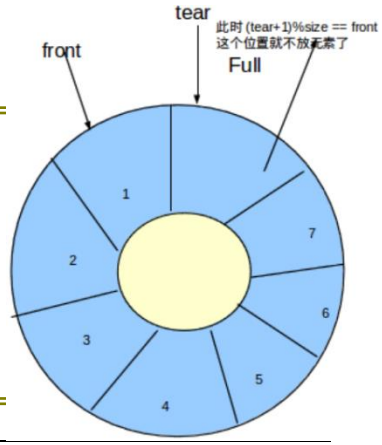
例如：设一个变量count用来记录队列中元素个数，当count==0时队列为空，当count= MAXQSIZE时队列为满。

(2)队满条件：以队头指针在队列尾指针的下一位置作为队列呈满状态的标志，牺牲一个存储空间；

队满条件为：

(sq.rear+1) mod maxsize==sq.front

队空条件为：sq.rear==sq.front



3.4.3 队列的顺序表示及实现---循环队列

- 队列初始化算法：

```
Status InitQueue (SqQueue &Q) {
                                    // 构造一个空队列Q
  Q.base = (ElemType *) malloc
          (MAXQSIZE *sizeof (ElemType));
  if (!Q.base) exit (OVERFLOW);
                                    // 存储分配失败
  Q.front = Q.rear = 0;
   return OK;
}
```

3.4.3 队列的顺序表示及实现---循环队列

- 入队列算法：

```
Status EnQueue (SqQueue &Q, ElemType e) {
                                  // 插入元素e为Q的新的队尾元素
    if ((Q.rear+1) % MAXQSIZE == Q.front)
        return ERROR; //队列满
    Q.base[Q.rear] = e;
    Q.rear = (Q.rear+1) % MAXQSIZE;
    return OK;
}
```

3.4.3 队列的顺序表示及实现---循环队列

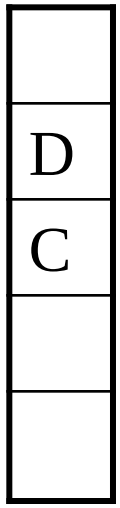
- 出队列算法：

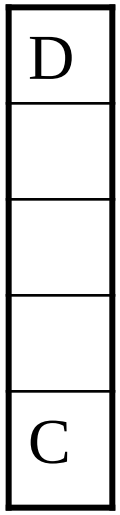
```
Status DeQueue (SqQueue &Q, ElemType &e) {
  // 若队列不空，则删除Q的队头元素，
  // 用e返回其值，并返回OK；否则返回ERROR
   if (Q.front == Q.rear)  return ERROR;
   e = Q.base[Q.front];
   Q.front = (Q.front+1) % MAXQSIZE;
   return OK;
}
```

3.4.3 队列的顺序表示及实现---循环队列

- 求队列长度：

分析以下两种状态如何求队列长度





sq->rear=4 **sq->rear=1**

sq->front=2 **sq->front=4**

3.4.3 队列的顺序表示及实现---循环队列

- 求队列长度算法：

```
int QueueLength(SqQueue Q)
{
    return (Q.rear-Q.front+MaxSize)%MaxSize;
}
```

3.4.4 队列的应用

- **队列使用的原则**：凡是符合先进先出原则的
  - 服务窗口和排号机、打印机的缓冲区、分时系统、树型 结构的层次遍历、图的广度优先搜索等等 结构的层次遍历、图的广度优先搜索等等
- **举例**
  - 约瑟夫出圈问题：n个人排成一圈，从第一个开始报数， 报到m的人出圈，剩下的人继续开始从1报数，直到所有 的人都出圈为止。
    - Josephus有过的故事：39 个犹太人与Josephus及他的朋友躲到一个洞中，39个犹太人决定宁愿死也不要被敌人抓。于是决定了自杀方式，41个人排成一个圆圈，由第1个人开始报数，每报数到第3人该人就必须自杀。然后下一个重新报数，直到所有人都自杀身亡为止。然而Josephus 和他的朋友并不想遵从，Josephus要他的朋友先假装遵从，他将朋友与自己安排在第16个与第31个位置，于是逃过了这场死亡游戏。
  - 舞伴问题：假设在周末舞会上，男士们和女士们进入舞厅 时，各自排成一队。跳舞开始时，依次从男队和女队的队 头上各出一人配成舞伴。若两队初始人数不相同，则较长 的那一队中未配对者等待下一轮舞曲。现要求写算法模拟 上述舞伴配对问题。

本章小结

- ✓ 熟练掌握：
  - (1)栈、队列的定义、特点和性质；
  - (2)ADT栈、ADT队列的设计和实现以及基本操作及相关算法。
- ✓ 重点学习：
  - ADT栈和队列在递归、表达式求值、括号匹配、数制转换、迷宫求解中的应用，提高利用栈和队列解决实际问题的应用水平。