数据结构

Data Structure

张先宜

手机：18056307221 13909696718

邮箱：zxianyi@163.com

QQ：702190939

QQ群：XC数据结构交流群 275437164

第3章 栈、队列与递归

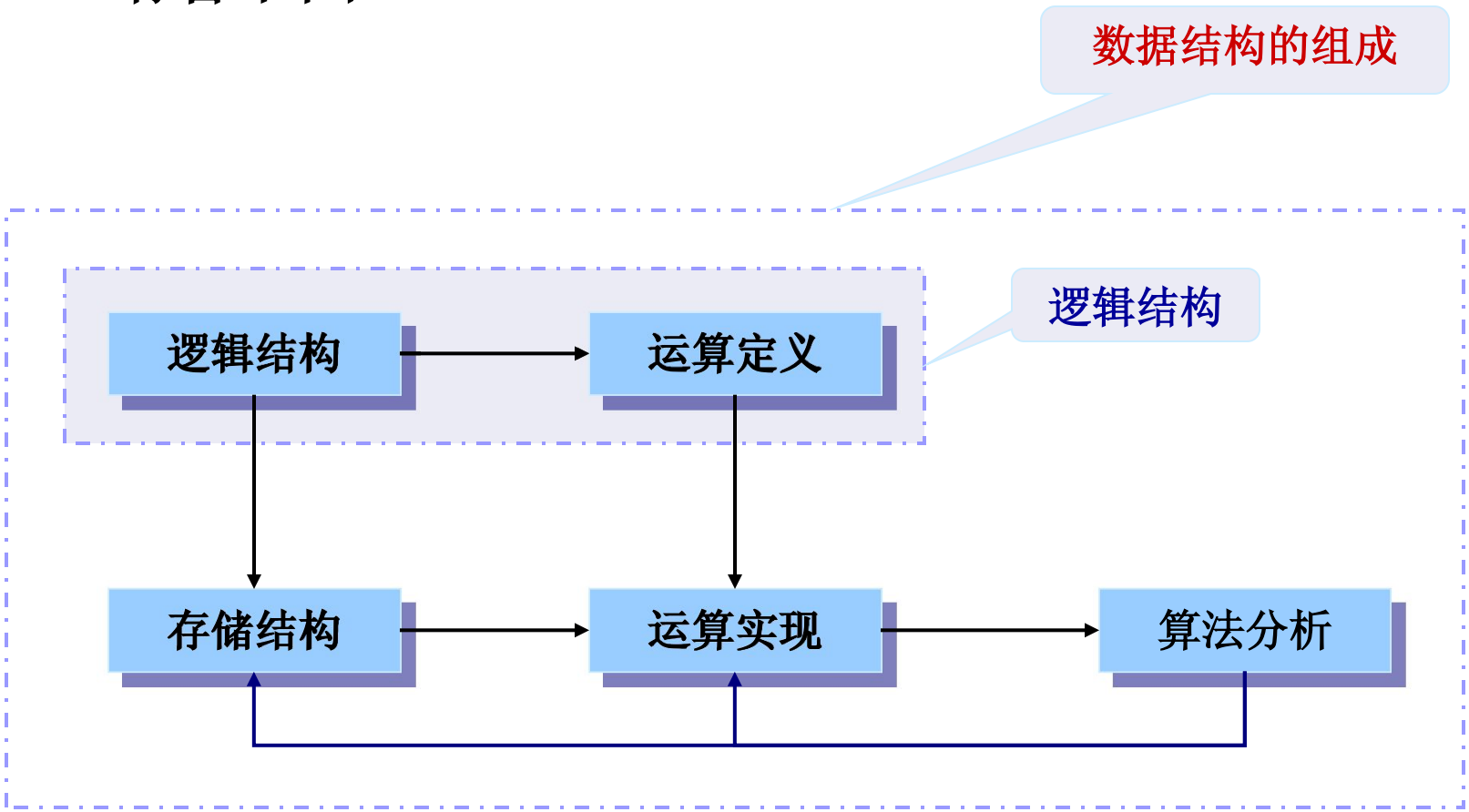
【本章内容】

- 栈和队列也是线性表，是操作受限的特殊的线性表。

- 线性表的插入、删除操作是不受限制的；而栈和队列的插入、删除操作是受限的。
  - 栈的插入、删除操作只能在表的同一端进行。
  - 队列限制在一端插入，另一端删除

- 但从数据类型角度看，栈和队列是与线性表不相同的两种重要的数据结构。在计算机科学和程序设计中有着广范的应用。

- 学习栈结构时，需要掌握哪些方面的内容?
  - 请看下图：



3.1 栈

【本章内容】

3.1.1 栈的定义与基本运算

3.1.2 顺序栈

3.1.3 链栈

3.1.4 栈的应用实例

3.1.1 栈的定义和基本运算

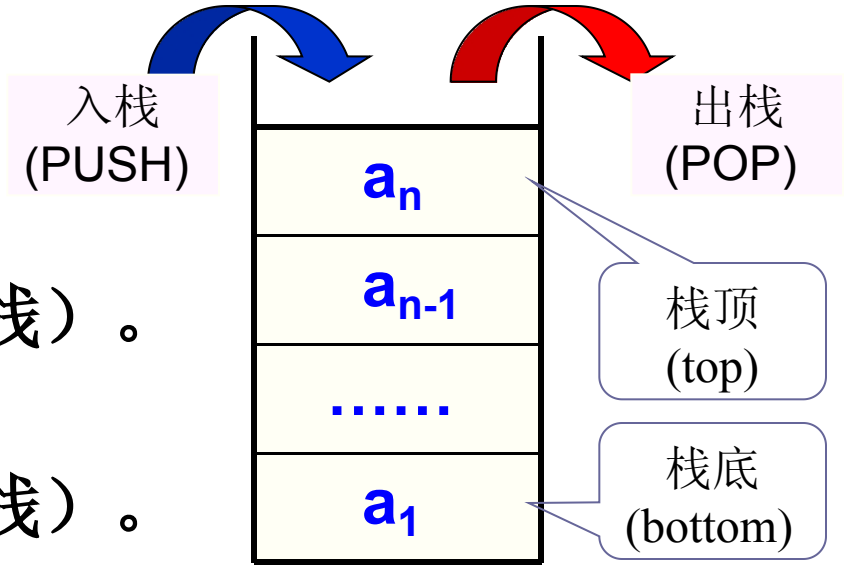
- 栈（Stack）是一种基本的、重要的、应用广泛的数据结构。
- 栈是按“先进后出”操作方式组织的数据结构。

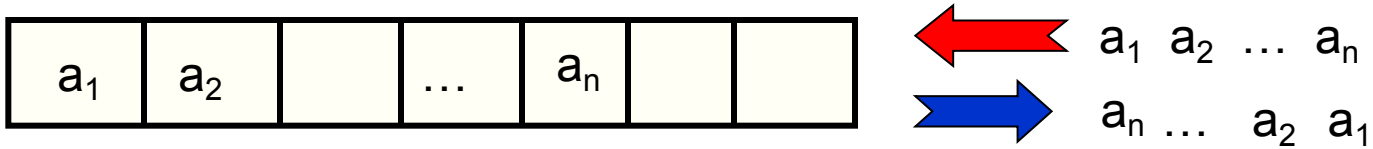
- 日常生活中栈操作方式实例:
  - 洗碗摞一堆；
  - 书摞成一堆；
  - 一些枪支子弹夹中子弹的操作方式等。

- 计算机中栈的应用实例:
  - 许多类型CPU的内部就构建了栈;
  - 在操作系统、编译器、虚拟机等系统软件中栈都有重要的应用，比如，函数调用、语法检查等。
  - 在应用软件设计和实际问题求解中更是经常会用到栈结构，比如，表达式求解、回溯法求解问题、递归实现、递归算法转换为非递归、树和图搜索等算法的非递归实现等。

1 栈的基本概念

- **栈（Stack）**
  - 是限定只能在一端插入和删除元素的线性表。
- **栈顶（top）**
  - 进行插入和删除操作的一端成为栈顶。
- **栈底（bottom）**
  - 另一端成为栈底。
- **入栈（push）**
  - 栈的插入操作叫入栈（压栈）。
- **出栈（pop），**
  - 栈的删除操作叫出栈（弹栈）。





- 栈的特性
  - 后进先出 (LIFO-- Last In First Out)，或
  - 先进后出 (FILO-- First In Last Out)

- 栈是操作受限的线性表。

2 栈的基本运算

- 下面逐个讨论栈的每个运算的情况

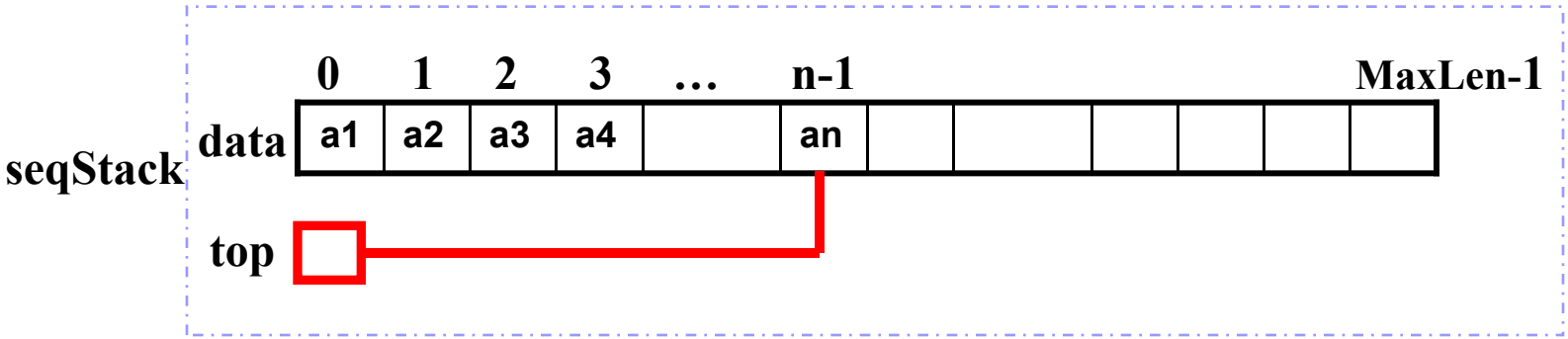
- **(1) 初始化**
  - **initialStack(S)**
- **(2)判断栈空**
  - **stackEmpty(S)**
  - **若栈为空，返回True，否则，返回False。**
- **(3)判断栈满**
  - **stackFull(S)**
  - **栈为满时，返回True，否则，返回False。**

- **(4)取栈顶元素值**
  - bool getTop(**S,x**)
  - 若栈不空，栈顶元素的值由变量x返回；
  - 取栈顶成功返回True；若为空栈，取栈顶失败，返回False。

- (5)入栈
  - push(S,x)
  - 将值为x的元素插入到栈顶。
  - 插入成功返回True；若插入前的栈已经满了，不能入栈时，插入失败，返回False。
- (6)出栈
  - pop(S)
  - 删除栈顶元素。
  - 删除成功返回True；若删除前为空栈，删除失败，返回False。

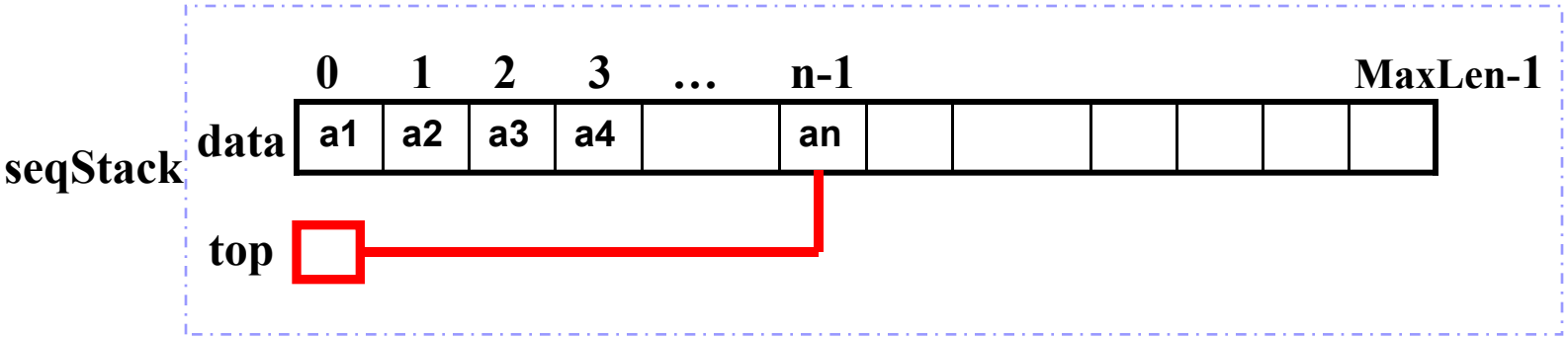
3.1.2 顺序栈

- **1 栈的顺序存储结构**
  - 假设有一个足够大的连续存储空间（数组）data, 用于存储栈的元素。
  - 将栈中的元素依次存储到数组中----顺序存储方式--顺序栈。
  - 如下图所示：



  - 其中，设置一个整型变量top，指示栈顶（栈顶指针），即栈顶元素的数组下标。也可用来计数元素个数。
  - 将数组data和top作为顺序栈的数据成员。

顺序栈存储结构 — 采用顺序表来存储的栈




```
#define MaxLen 100                    //最大容量
typedef   struct
{    elementType  data[MaxLen];   //存放栈元素
     int  top;                    //指示栈顶元素的位置
 }seqStack;
```

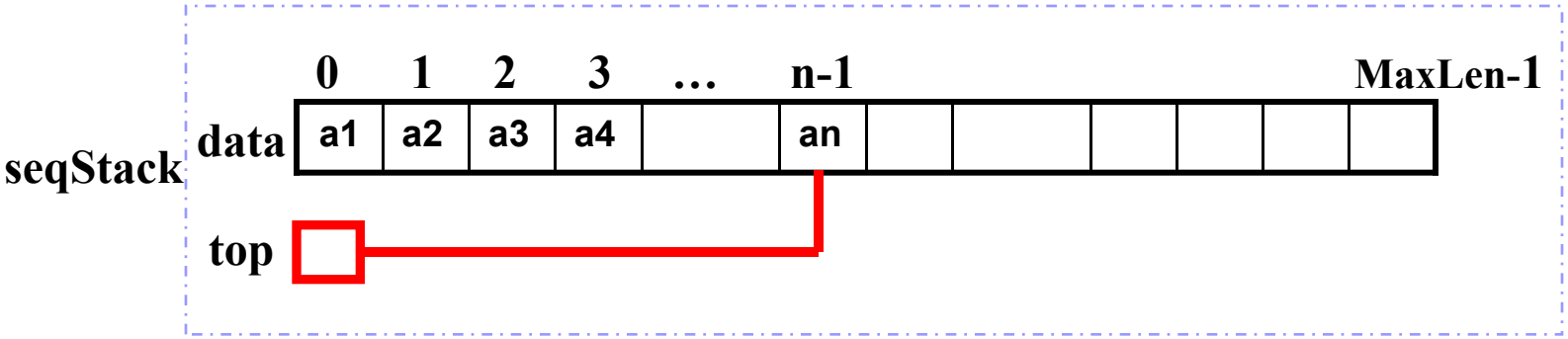
2 顺序栈类的C++描述--Stack类的完整描述

```
class Stack
{
 public:
    stack();                                  //初始化
    bool empty();                             //判断栈空
    bool full();                              //判断栈满
    bool getTop(elementType &x);              //取栈顶元素
    bool push(elementType x);                 //入栈
    bool pop();                               //出栈
    //其它运算（操作）;
private:
    elementType data[MaxLen];                 //存放栈元素
    int top;                                  //栈顶指针（数组下标）
    //其它数据成员。注意书中的count与top的区别
};
```

函数成员说明

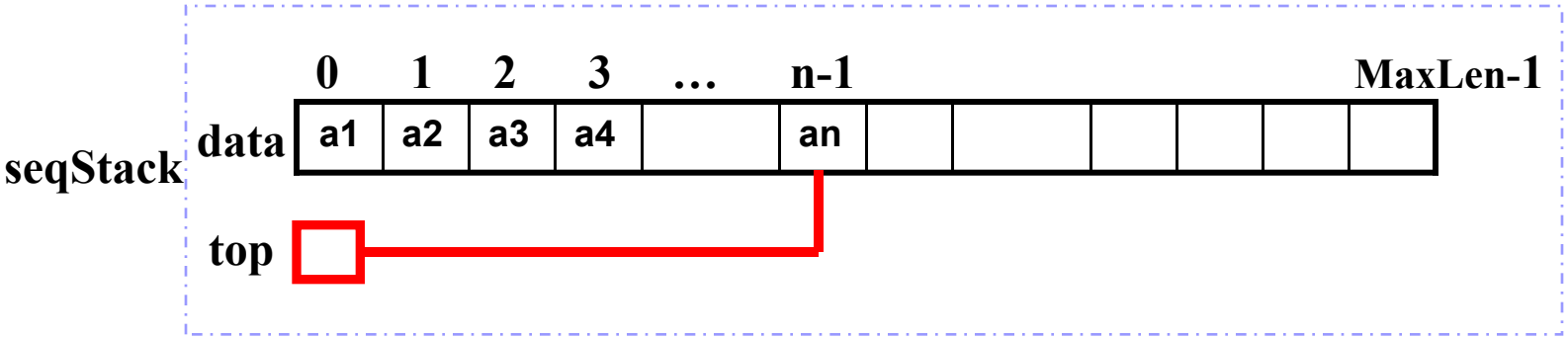
数据成员说明

3 顺序栈上运算的实现



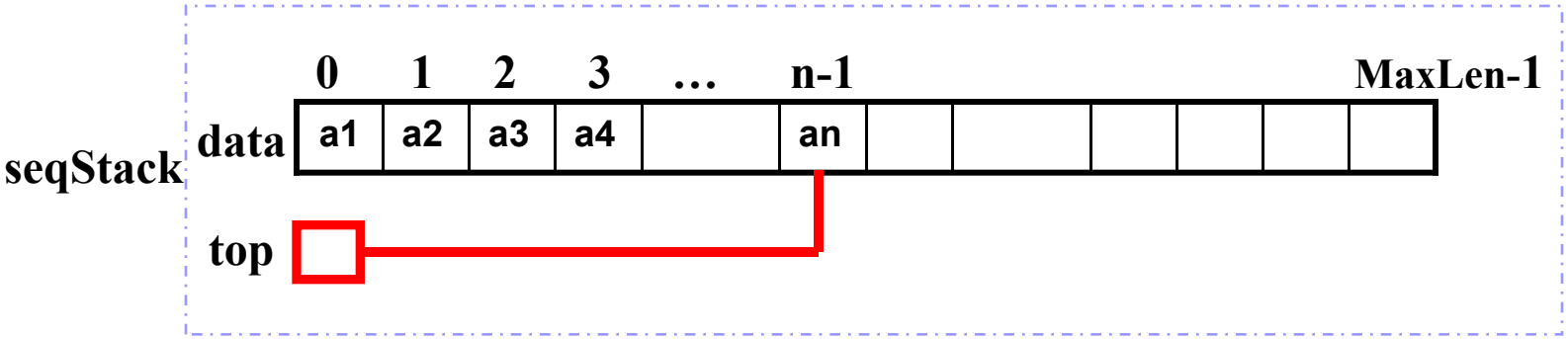
- **(1) 初始化**
  - 置栈顶top=-1。
  - void initStack(seqStack * S)

```
{
    S->top = -1;
}
```



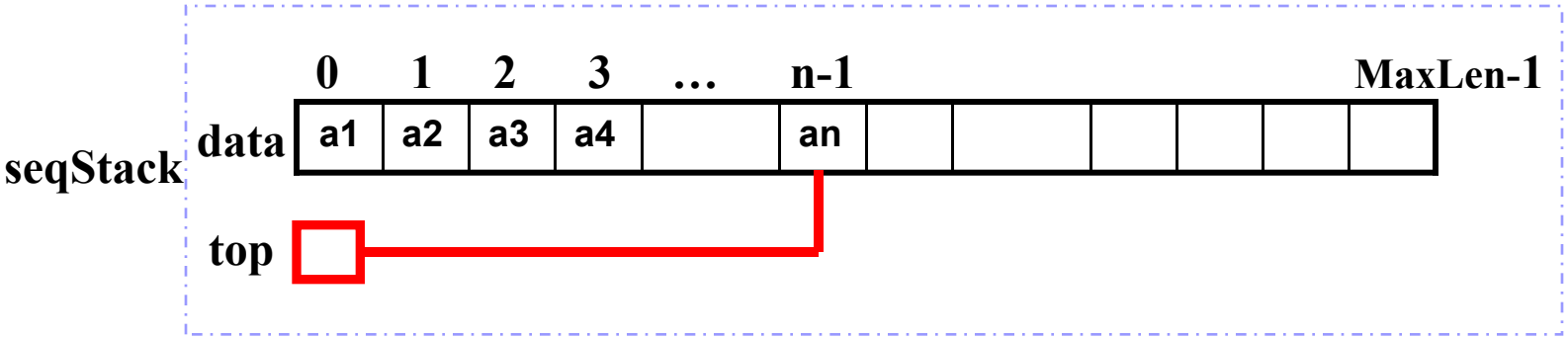
- **(2) 判栈空:**

```
bool stackEmpty(seqStack &S)
{
   if(S.top==-1)
   return true;
   else
   return false;
}     //简写：return (S.top == -1);
```



- **(3) 判栈满:**

```
BOOL  stackFull( seqStack S )
{   if ( S.top == MaxLen-1 )
        return TRUE;     //栈满，返回true
    else
        return FALSE;   //栈不满，返回false
}
```



- **(4) 取栈顶元素:**

```
void stackTop(seqStack * S, elementType &x)
{
   if ( S -> top == -1 )
         error("栈空" );
   else
         x=S->data[S->top];   //参数x返回栈顶元素
}
```

(5) 入栈:

```
void push( seqStack * S, elementType x )
{
    if ( S ->top == MaxLen-1 )
        error("栈满" );
    else
    {
        S -> top ++;                    //栈顶后移
        S -> data [S -> top] = x ; //元素x入栈
    }
}
```

- **(6) 出栈:**

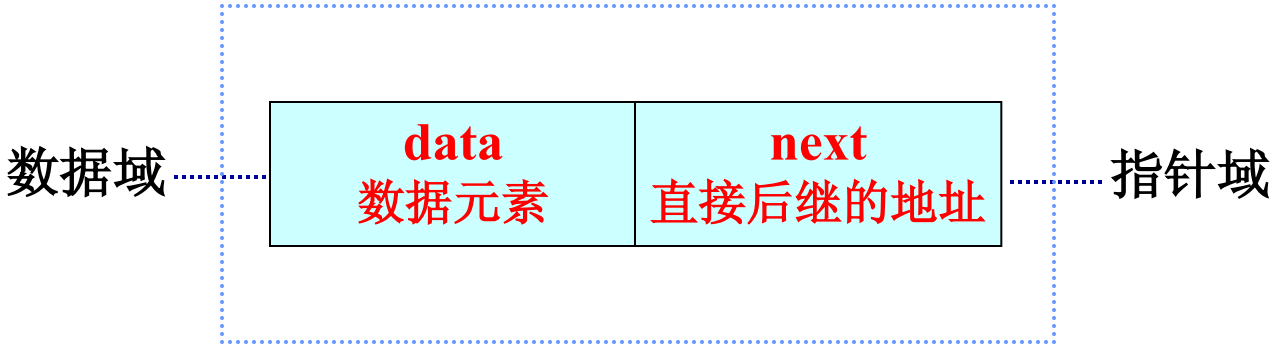
```
void pop(seqStack * S, elementType &x)
{
    if  (stackEmpty(*S))
        error("栈空,不能删除" );
    else
    {
        x=S->data［S->top］; //取栈顶元素至x
        S->top--;  //栈顶减1，即删除了栈顶元素
    } //可用一行代码： x=S->data［S->top --］;
}
```

- 顺序栈特点
  - 所有运算的时间复杂度均为O(1)；
  - 通常一次性申请空间，只能按最大空间需求分配，容易造成空间浪费。-- 使用链式栈。
  - 本教材使用数组实现简单易理解，也可以使用malloc()或new申请一块连续的空间实现，但处理和操作较复杂。
- 【思考问题】
  - 如果要让一套代码适用于所有数据类型，应该使用C++的什么技术？
  - -- template （模板）

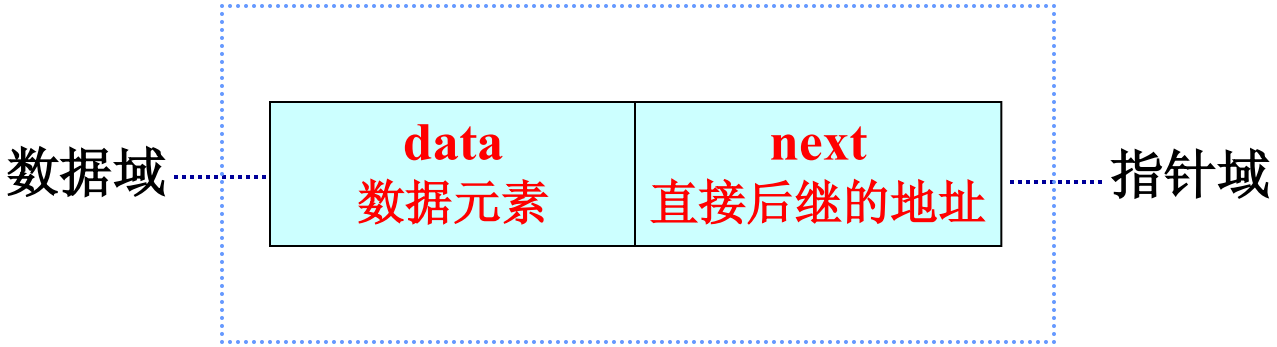
3.1.3 链栈

(1) 链栈的存储结构

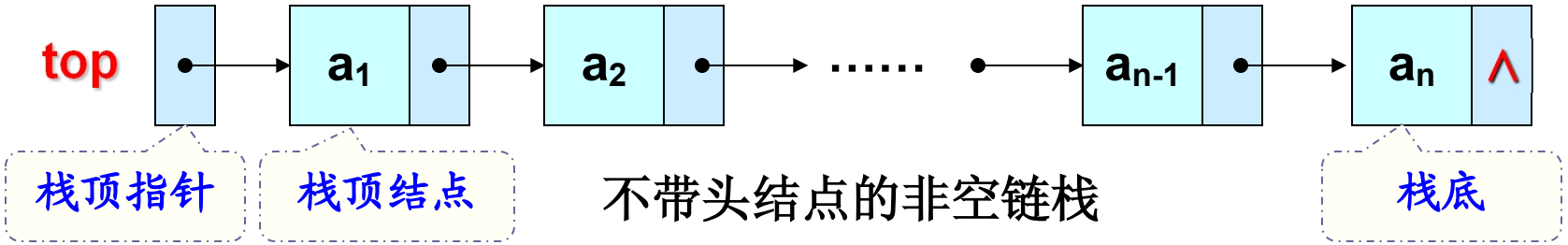
- 链栈即采用链式存储结构实现的的栈。
- 链栈可用单链表结构来表示。结点结构同单链表。
- 链栈既可以带头结点，也可以不带头结点。



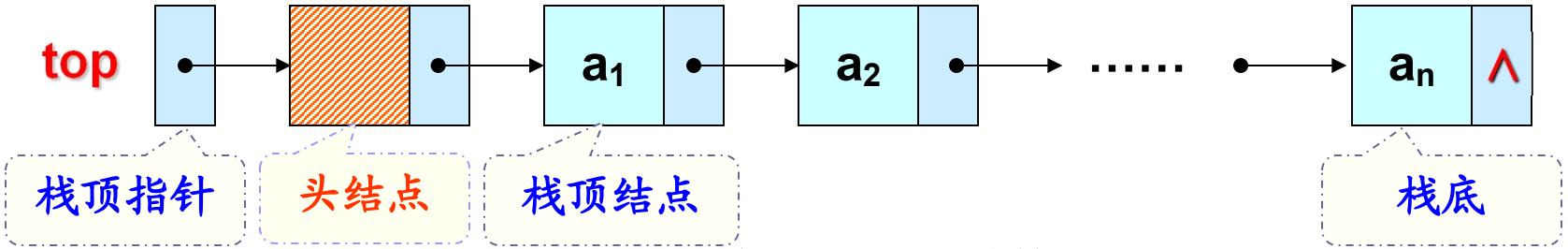
- 在这种表示中，同样要注意栈顶位置的选择。
  - 不带头结点，栈顶指针top指示首元素结点；
  - 带有头结点，栈顶指针top->next指示首元素结点；
  - 事实上，这里的栈顶指针top即单链表的头指针（只是叫法不同）。



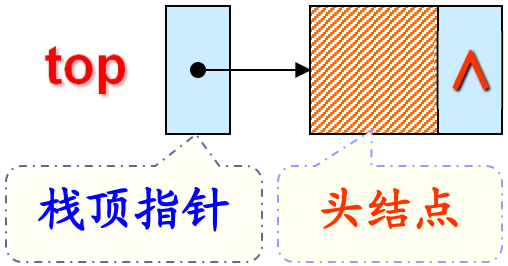
- 链栈既可以带头结点，也可以不带头结点。



不带头结点的非空链栈



带头结点的非空链栈



带头结点的空链栈

(2) 链栈的存储实现

【链栈的结点结构】

- 链栈的结点结构同单链表

```
typedef struct lsNode
{
   elementType data;        //链栈结点数据域
   struct lsNode *next;     //链栈结点指针域
} node;
```

【链栈类Stack描述】

```
class Stack
{
public:
    Stack();     //初始化空栈
    ~Stack();    //销毁链栈
    bool empty();                          //判定栈空
    bool getTop(elementType &x);  //取栈顶元素
    void push(elementType x);        //入栈
    bool pop();                             //出栈

    void print();       //从栈顶到栈底打印站内元素值
    void destroy();  //销毁链栈
private:
    node* top;   //栈顶指针
    int count;    //记录结点个数
};
```

2 链栈的运算实现

- 不带头结点链栈的基本运算

(1)初始化栈

- 由于不带头结点，初始化仅需将栈顶指针置为空，表示空栈。结点计数初始化为0。

【算法描述】

```
void initialStack(sNode *& top)
{
   top=NULL;
}
```

(2) 判定栈空

- 利用top==NULL判定。

【算法描述】

```
bool stackEmpty(node* top)
{
   if(top==NULL)
   return true;     //栈空，返回true
   else
   return false;   //栈不空，返回false
}
```

(3) 取栈顶元素

```
bool getTop(node * top, elementType &x)
{
   if(top==NULL)
   return false;    //栈空，返回false
   else
   {
   x=top->data;   //取栈顶元素，用参数x返回
   return true;    //取栈顶成功，返回true
   }
}
```

(4)入栈

【算法描述】

```
void push(node *& top, elementType x)
{
   node* s;

   s=new node;
   s->data=x;

   s->next=top;
   top=s;
}
```

(5)出栈

```
bool pop(node *& top, elementType &x)
{  node* u;
   if(top==NULL)
   return false;      //栈空，返回false
   else
   {
   x=top->data;      //取栈顶元素，由变量x返回
   u=top;            //栈顶指针保存到u
   top=top->next;    //栈顶指针后移一个元素结点
   delete u;         //释放原栈顶结点
   return true;      //出栈成功，返回true
   }}
```

(6) 释放链栈空间

```
void destroyStack(node *& top)
{
   node *p,*u;
   p=top;
   while(p)
   {
   u=p;
   p=p->next;
   delete(u);
   }
   top=NULL;
}
```

【算法分析】

- 销毁链栈运算的时间复杂度为O(n)，
- 其它运算的时间复杂度皆为O(1)。

【思考问题】

- 请您自行完成带头结点链栈的基本运算。

- 栈的链式存储结构（链式栈）特点：
  - 使用连续或不连续的存储空间；
  - 各数据元素独立存储，依靠指针链接建立逻辑相邻关系；
  - 对每个数据元素单独申请结点空间；
  - 由于动态申请空间，没有栈满溢出问题；
  - 栈顶指针 top 唯一确定一个链式栈；

- 链式栈也可以加同构头结点，这种情况下：
  - Top->next 指向栈顶元素；
  - 若 top->next 为 NULL，则为一空栈。
- 如果同时需要2个以上栈，最好使用链式栈。
- 销毁链栈运算的时间复杂度为O(n)，其它运算的时间复杂度皆为O(1)。

【例3.1】 将单链表就地逆置。即：利用原有结点，前驱变后继，后继变前驱。

【解题分析】

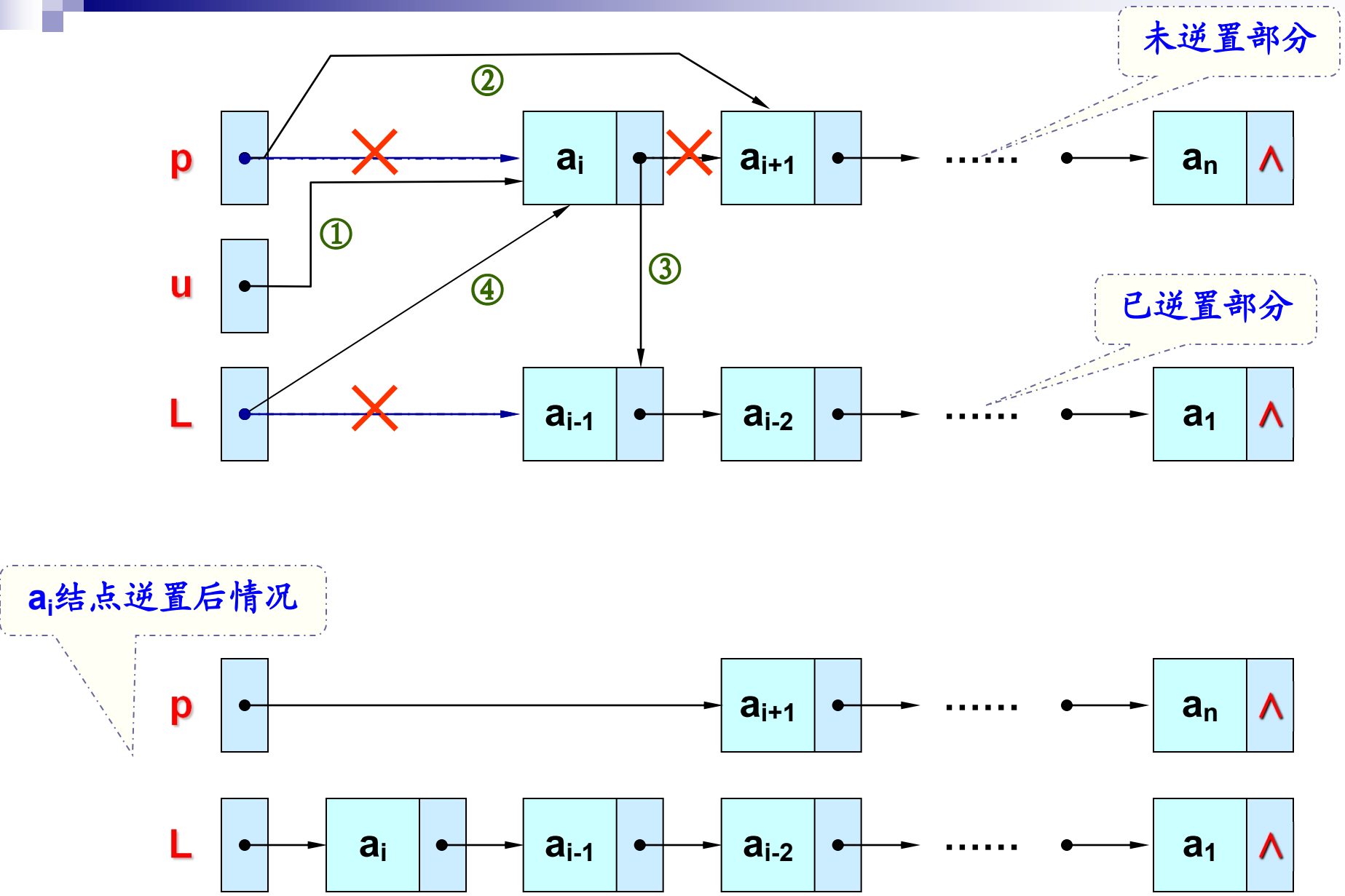
- 就地逆置指不申请新结点，利用原表结点重新链接完成逆置。

【解法一】基于第二章介绍的单链表技术求解。

- 方法是从原链表首元素结点开始，依次取出结点，采用头插法构建一个新表即可，只是这里不要申请新结点。这样操作的过程中，原链表结点就被分为两个部分：已逆置部分、未逆置部分。未逆置部分用指针p指向第一个结点；已逆置部分用指针L指示第一个结点。

【解法一实现】

- exam301ListReverse


未逆置部分
②
p
a_i
a_{i+1}
……
a_n
∧
①
u
③
④
已逆置部分
L
a_{i-1}
a_{i-2}
……
a_1
∧
a_i结点逆置后情况
p
a_{i+1}
……
a_n
∧
L
a_i
a_{i-1}
a_{i-2}
……
a_1
∧

【单链表就地逆置解法一】

```
void listReverse(linkedList &L)
{
   node *p,*s;
   p=L->next;
   L->next=NULL;
   while(p)
   {
   s=p;                          //未逆置部分取出一个结点
   p=p->next;
   s->next=L->next;   //头插接入已逆置部分
   L->next=s;
   }
}
```

【解法二】基于栈技术求解。

- 定义一个顺序栈，栈元素为单链表的结点指针node*，循环取出每个结点的指针，入栈。入栈后，原链表尾结点的指针存放在栈顶，首元素结点指针存放在栈底。构建逆置链表时，将栈内的指针依次弹出，按尾插法重建链表，这样就得到一个就地逆置的链表。栈的元素类型要定义为 typedef node* elementType。因为要采用尾插法，重建时要设置一个尾指针。

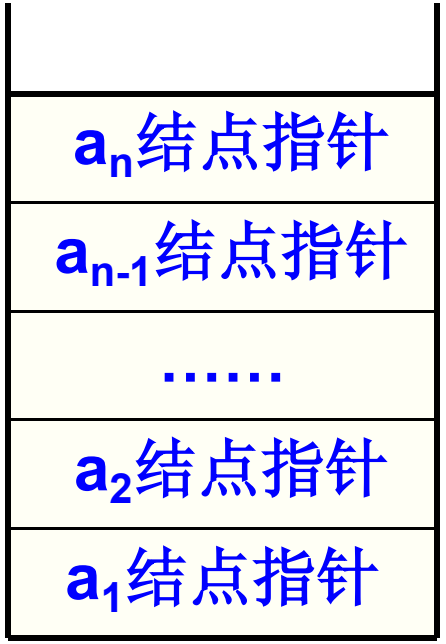
【解法二实现】

- exam301ListReverse

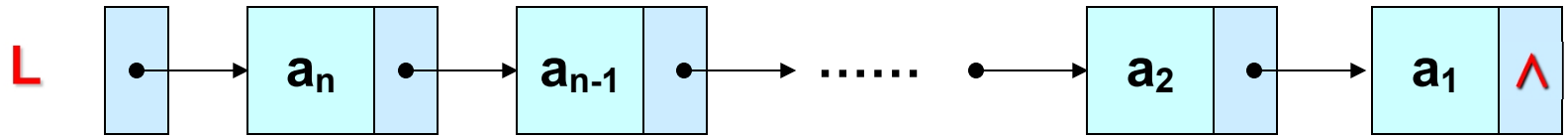
【思考问题】

- 用链栈可以实现单链表就地逆置吗？如何实现？

- a_n结点指针第一个出栈，设为u，它成为首元素结点，须单独做如下处理：
  - L=u; //头指针L（top）指向a_n结点
  - R=u;
  - L->next=NULL;
- 其它结点依次出栈，处理：
  - R->next=u; //尾插当前结点
  - u->next=NULL; //尾结点next为空
  - R=u; //移动尾指针
- 最后出栈的是a_1结点的指针。



指针全部入栈情况



3.1.4 栈的应用实例

1. 栈的基本应用实例

【例3.2】 输入n个整数，逆序输出。

【分析】利用栈结构的FILO特性，输入时采用循环入栈，输出采用循环出栈。

【实现代码】Exam302ReverseOut.cpp

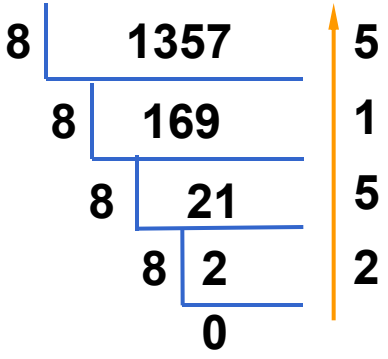
【例3.3】 十进制数转换为八进制数。

【分析】

“除8取余”方法。设十进制数为N，循环除8，取出余数，商的整数部分重新赋给N，直至N==0；

- 将最后的余数作为8进制数的最高位，第一个余数作为8进制的最低位，得到8进制数。即：余数逆（反）序输出。操作过程如下图：
- 利用栈，每次余数入栈，结束时余数全部出栈即可得到8进制数。

- 【问题】
  - 十进制转换为二进制数呢？
  - 十进制转换为十六进制数呢？
  - 十进制转换为任意进制数呢？

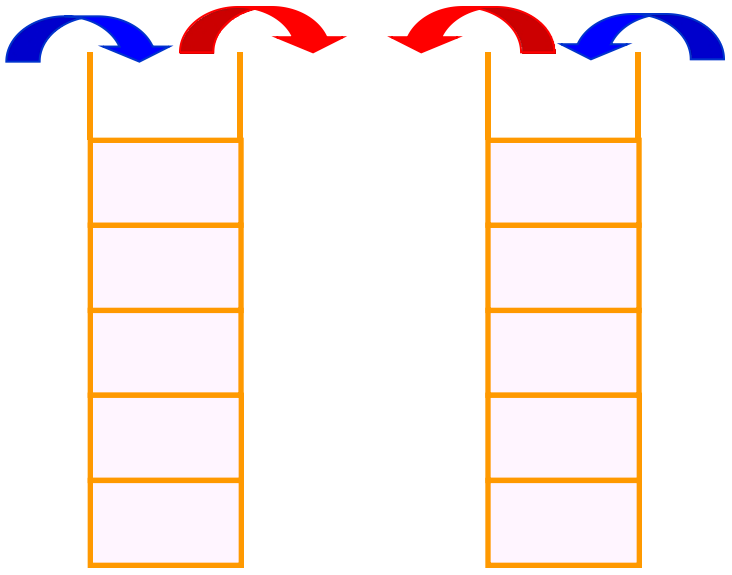


```
void Dec2Ocx(int N) {
   Stack S;  int Mod,x;
   while(N!=0)
    {    Mod=N % 8;          //除8取余数到Mod
      S.push( Mod  );       //余数入栈
      N=N/8;                //除8取商，回存到N
    }
   cout<<“八进制数为：”;      //出栈输出8进制数
   while(!S.empty())
   {   S.getTop(x);         //取栈顶元素到x
      cout<<x;              //输出栈顶元素
        S.pop();            //栈顶元素出栈
}}
```

- **2. 栈在表达式计算中的应用**
  - 表达式计算是计算机要解决的基本问题。
- **表达式构成**
  - 运算符—算术运算符、关系运算符、逻辑运算符
  - 操作数—常量、变量、包括中间计算结果
  - 定界符—成对出现的符号对，改变运算优先级

- 优先级规则
  - 算术运算符优先级不变；顺序出现的同级运算符级别按先后次序，先来的级别高；
  - 定界符，如括号，成对出现；一对定界符外部任何其它运算符优先级低于定界符；一对定界符内部所有运算符优先级高于定界符。
- 构建2个栈
  - 操作数栈—表达式中常量、变量、中间计算结果
  - 运算符栈—运算符、定界符

- 运算符出栈计算规则：
  - 栈顶运算符优先级高，待入栈运算符优先级低，栈顶运算符出栈执行运算；若是一对定界符释放；
  - 栈顶运算符优先级低，待入栈运算符优先级高，待入栈运算符直接入栈。



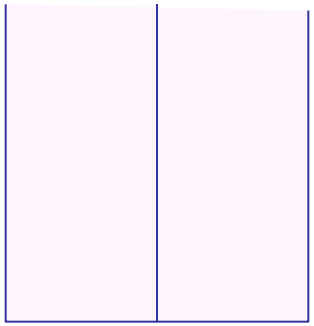
【例3.4】模拟表达式 12+5*(2+3)*6/2-4 的求解过程。

【分析】构建2个栈，操作数和运算符各自依次入栈。为处理方便给整个表达式加一对定界符“#”。

#12 + 5 × (2+3) × 6 ÷ 2 - 4#

#12+5*(2+3)*6/2-4#

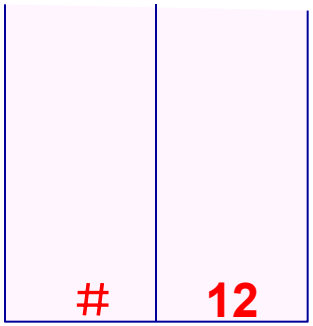
CurrentS='#'



开始时，指向第一个符号位置，准备读入，此时的两个栈均为空。

#12+5*(2+3)*6/2-4#

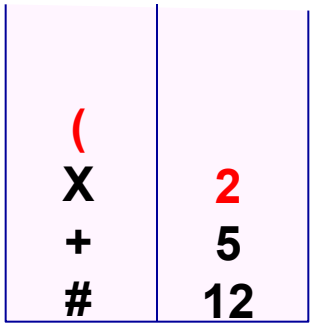
CurrentS='+'



■ '#' 入栈； '12'入栈；
■当前运算符 '+' 优先级高，入栈。

#12+5*(2+3)*6/2-4#

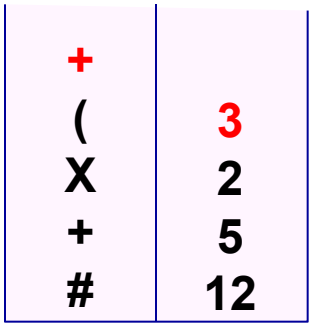
CurrentS='+'



■依次读入 '5'、 '×'、 '('、 '2'，入栈；
■当前读入 '+'，优先级高，入栈。

#12+5*(2+3)*6/2-4#

CurrentS=')'



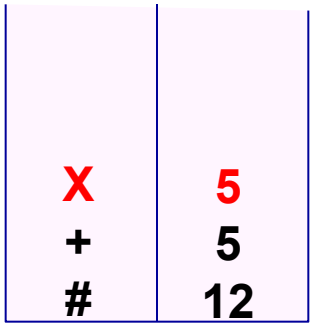
■ '+' 入栈； '3'入栈；

■当前运算符 ')' 优先级低，栈顶运算符 '+'出栈运算；

■操作数栈弹出3、2，执行2+3，结果5入栈。

#12+5*(2+3)*6/2-4#

CurrentS='X'

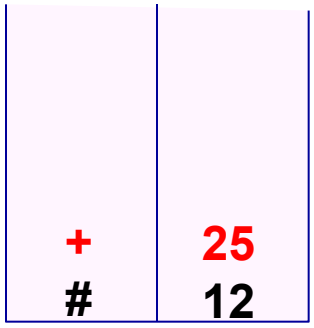


■栈顶算符 'X'、当前算符 'X'；

■但栈顶算符 'X'先，优先级高，出栈，执行计算：5×5

#12+5*(2+3)*6/2-4#

CurrentS='X'

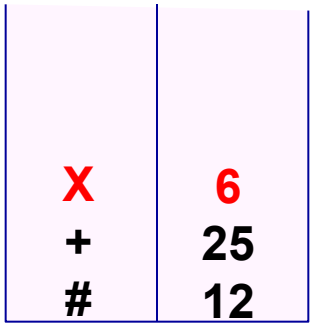


■栈顶算符'+'、当前算符'X'；

■当前算符优先级高，入栈；

#12+5*(2+3)*6/2-4#

CurrentS='/'

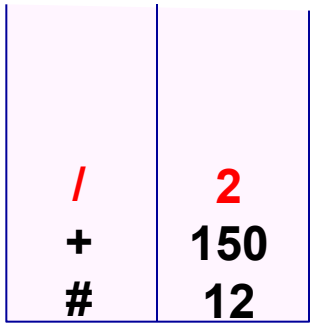


■栈顶算符'X'、当前算符'/'；

■但栈顶算符'X'先，优先级高，出栈，执行计算：6×25

#12+5*(2+3)*6/2-4#

CurrentS='/'



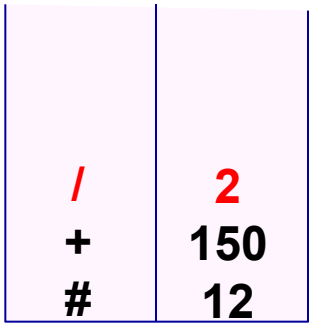
■栈顶算符‘+’、当前算符‘/’；

■当前算符‘/’优先级高，入栈；

■ ‘2’入栈。

#12+5*(2+3)*6/2-4#

CurrentS='-'

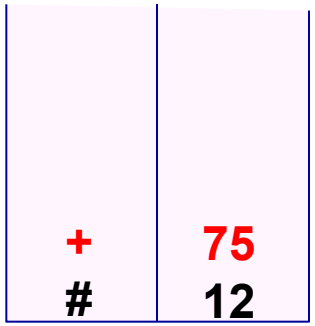


■栈顶算符‘/’、当前算符‘-’；

■栈顶算符‘/’优先级高，出栈，计算150/2=75

#12+5*(2+3)*6/2-4#

CurrentS='-'

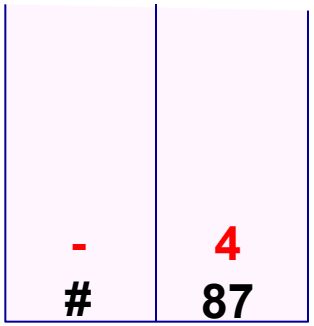


■栈顶算符‘+’、当前算符‘-’；

■栈顶算符‘+’先，优先级高，出栈，计算75+12=87

#12+5*(2+3)*6/2-4#

CurrentS='-'



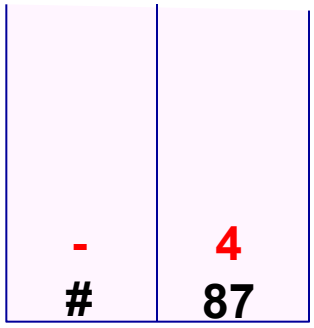
■栈顶算符‘#’、当前算符‘-’；

■当前算符‘-’优先级高，入栈；

■ ‘4’入栈

#12+5*(2+3)*6/2-4#

CurrentS='#'

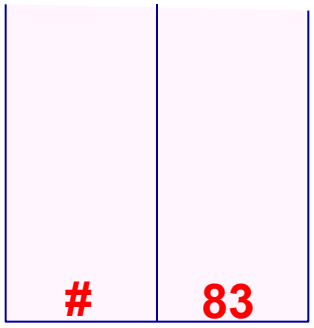


■栈顶算符‘-’、当前算符‘#’；

■栈顶算符‘#’优先级高，出栈，计算87-4=3，入栈。

#12+5*(2+3)*6/2-4#

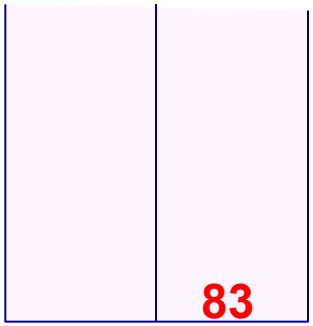
CurrentS='#'



■栈顶算符‘#’、当前算符‘#’；

■栈顶算符‘#’先，优先级高，出栈，与当前算符成对，共同释放。

#12+5*(2+3)*6/2-4#

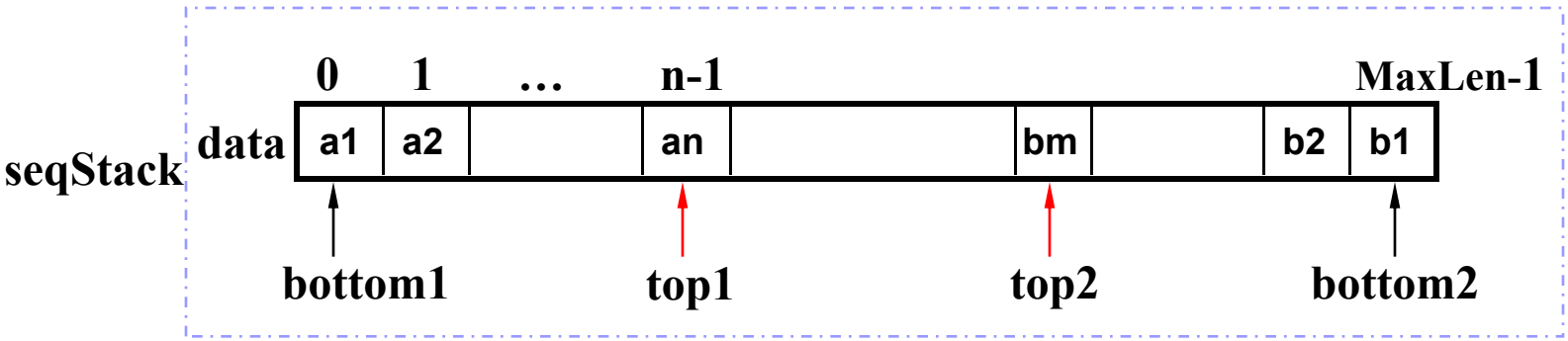


- 算符栈空；
- 操作数栈是最终结果83；
- 需要时83出栈，销毁2个栈。

3. 栈的其它应用

(1) 顺序栈的空间共用问题

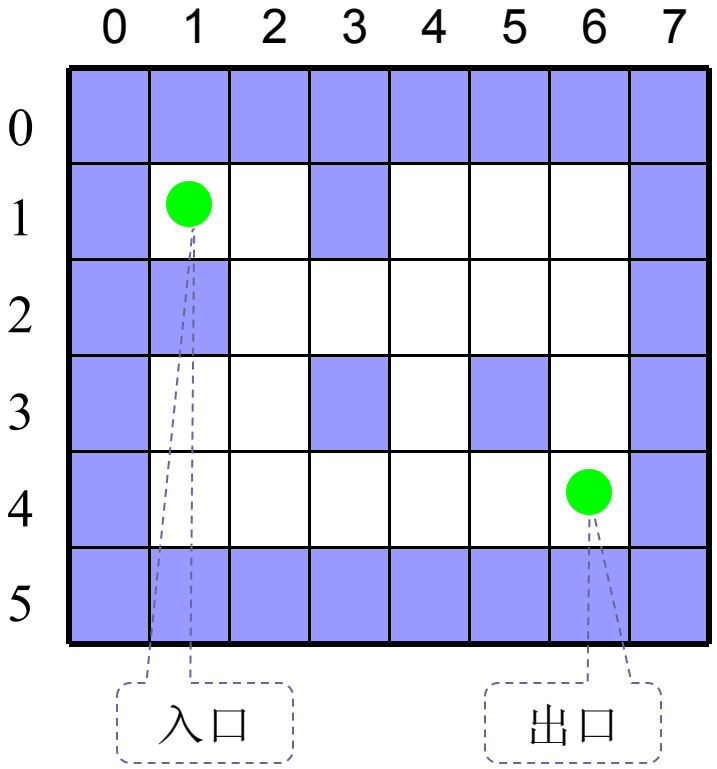
- 在一块连续的存储空间上构建2个共用空间的栈，提高空间利用率，如图所示：



- 如何定义存储结构和基本运算？

(2) 迷宫问题（maze problem）

- 一个迷宫的实例，如图所示：
- 图中紫色方块为障碍，不能通行；
- 白色方块可以通行；
- 行进方向4个或8个；
- 2维数组表示迷宫；
- 方块用2维数组坐标表示；
- 用栈如何找到路径？



栈小结

- 栈的概念
  - ☞什么是栈？栈顶、栈底、入栈、出栈
- 栈的特点—LIFO，或FILO
- 栈的6个基本运算
- 栈的顺序存储结构实现
- 栈的C++描述
- 栈的基本应用

【布置作业】

（p95-p96）

- 3.1
- 3.2
- 3.4
- 3.5

A man who does not plan long ahead will find trouble right at his door.

Confucius

子曰：“人无远虑，必有近忧”

3.2 队列

【本节内容】

3.2.1 队列的定义和运算

3.2.2 顺序队列和循环队列

3.2.3 链队列

- 队列也是线性表，是操作受限的特殊的线性表。线性表的插入、删除操作是不受限制的；队列的插入操作在表的一端进行，删除操作在表的另一端进行。

- 但从数据类型角度看，队列是和线性表不相同的两种重要的数据结构。在计算机科学和程序设计中有广范的应用。

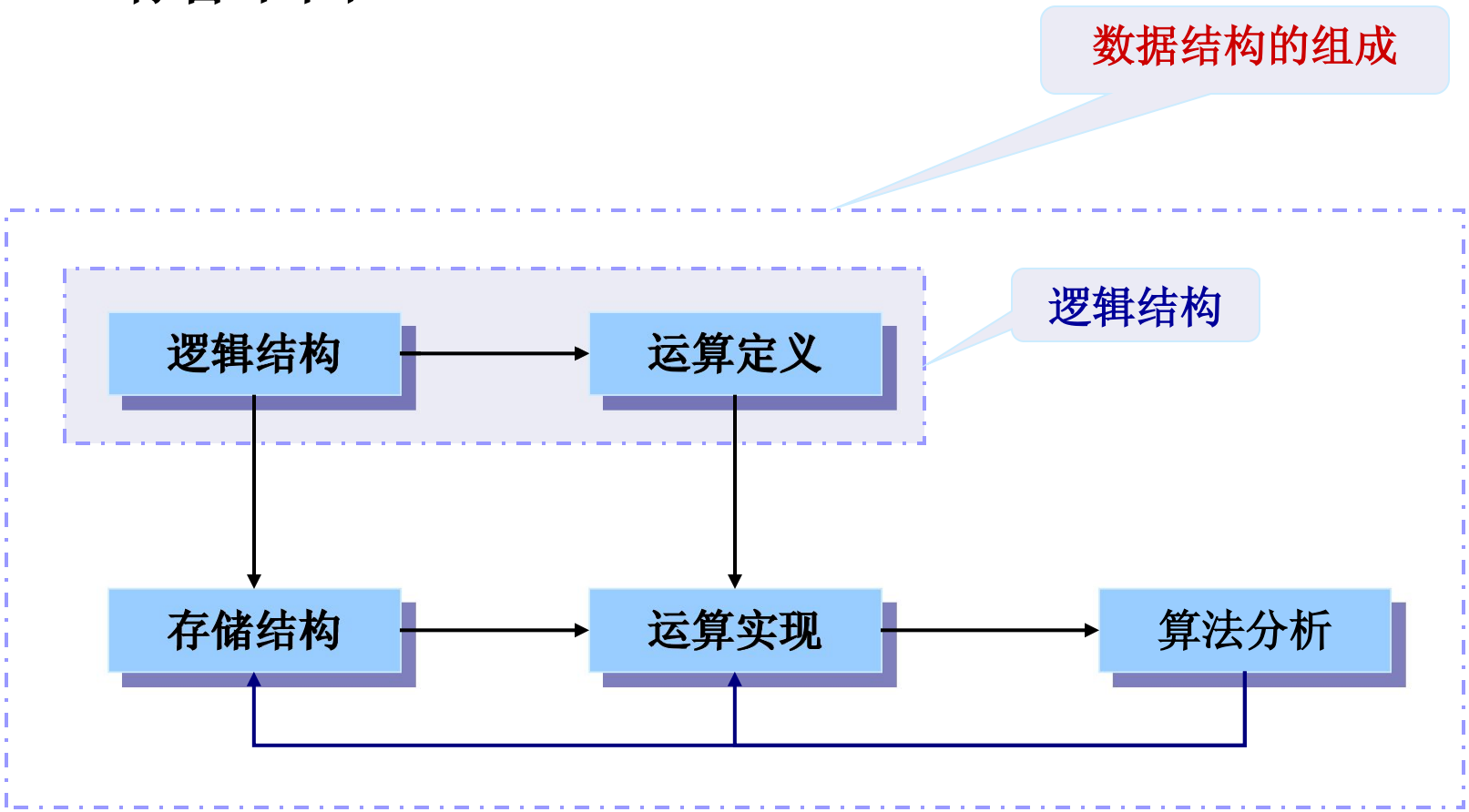
3.2.1 队列

- 队列（Queue）也是一种基本的、重要的、应用广泛的数据结构。
- 队列是按“先进先出”操作方式组织的数据结构。
- 日常生活中队列操作方式实例：
  - 人类在争用某些稀缺资源时，为体现公平，往往采取排队的解决办法，按“先来先服务”的原则使用这些资源。
  - 春节排队购买火车票；食堂排队买饭等。

- 计算机中队列的应用实例：
  - 早期Host+Terminal系统，终端争用CPU；
  - 许多类型CPU的内部就构建了取指令队列，加快指令执行速度；
  - 计算机主机向外设传输数据，比如打印机打印文档，当外设速度较慢，来不及处理主机传来的数据时要使用队列对数据进行缓冲；

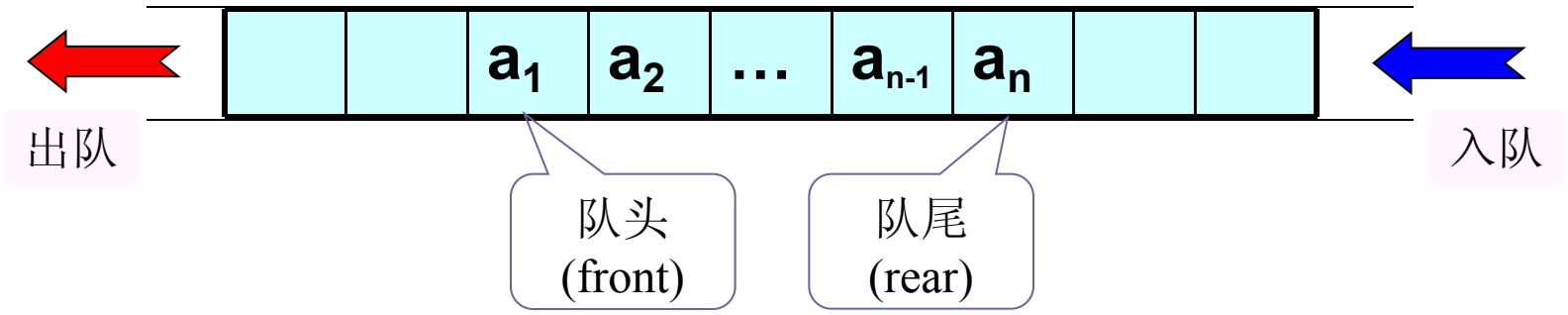
- 操作系统中的进程、线程调度等用到队列，Windows操作系统更是大量使用信息队列，利用消息驱动来调度和管理计算机系统，系统运行时构建众多的消息队列，将不同的消息放入不同的消息队列进行管理；
- 网络应用系统中队列也得到广泛的使用，比如C/S、B/S模式系统中，服务器为了处理众多客户端的并发访问，会用队列对客户端的服务请求进行管理；
- 此外，解决实际问题的应用系统中也会经常使用队列。

- 学习队列结构时，需要掌握哪些方面的内容？
  - 请看下图：

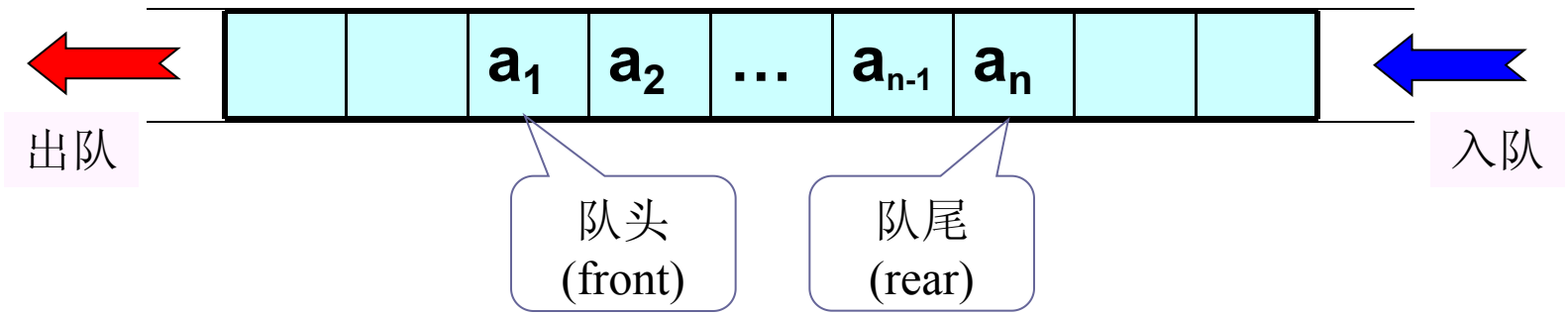


1 队列基本概念

- **队列（Queue）**
  - 是限定只能在一端插入元素，另一端删除元素的线性表，也称作先进先出表。
- **空队列**
  - 没有元素的队列。



- 队头（front）
  - 删除元素的一端。
- 队尾（rear）
  - 插入元素的一端。
- 入队
  - 插入操作，即队尾插入。
- 出队
  - 删除操作，即删除队头。



- 队列的特性
  - 先进先出（FIFO—First In, First Out）
- 队列也是操作受限的线性表。

2 队列的基本运算

- **1. 队列的基本运算定义**
  - **(1) 初始化： initialQueue(Q)**
    设置队列为空。
  - **(2) 判断队列为空：queueEmpty(Q);**
    若为空，则返回TRUE，否则返回FALSE.
  - **(3) 判断队列为满： queueFull(Q);**
    若为满，则返回TRUE，否则返回FALSE.

- **(4) 取队头元素： getFront(Q, x)**

  条件：队列不空。

  否则，应能明确给出标识，以便程序的处理。

- **(5) 入队： enQueue(Q, x)**

  将元素入队，即放到队列的尾部。

  如果队列满，怎样处理？

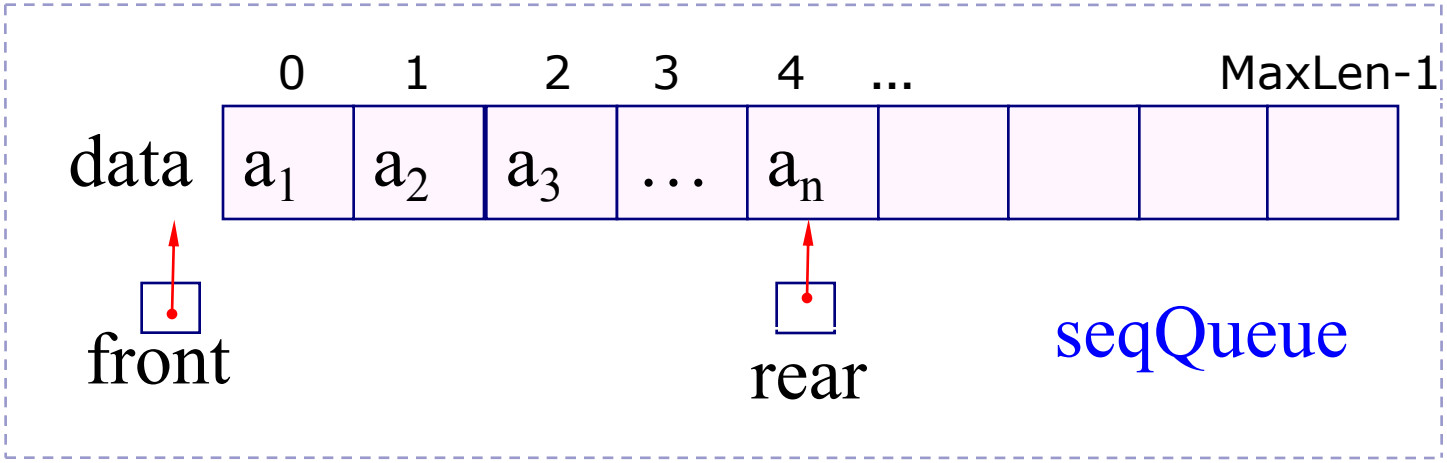
- **(6) 出队： outQueue(Q,x)**

  删除队头元素。

  如队列空而不能进行，应怎样处理？

3.2.2 顺序队列

- **1 顺序队列存储结构**
  - 用连续的存储空间（数组）**data**, 存储队列的元素。
  - 将队列中的元素依次存储到数组中----顺序存储方式--顺序队列。



- 【顺序队列存储描述】

```
typedef struct
{ elementType  data[MaxLen];  // 存放元素
   int  front;   //队头指针
   int  rear;    //队尾指针
}seqQueue;
```

顺序队列的C++描述--Queue类的完整描述

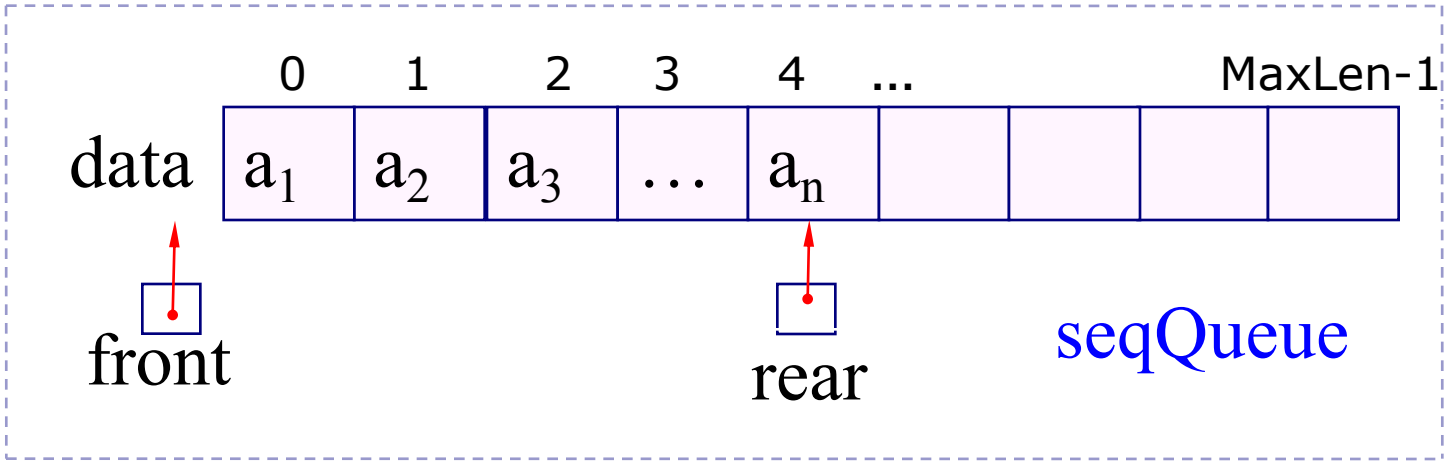
```
class Queue
{
 public:
    Queue();                                  //初始化
    bool empty();                             //判断队空
    bool full();                              //判断队满
    bool getFront(elementType &x);            //取队头元素
    bool enQueue(elementType x);              //入队
    bool outQueue();                          //出队
    //其它运算（操作）;
private:
    elementType data[MaxLen];    //存放队列元素
    int front, rear;     //队头和队尾指针（数组下标）
    //其它数据成员;书中的count成员是非必需的
};
```

函数成员说明

数据成员说明

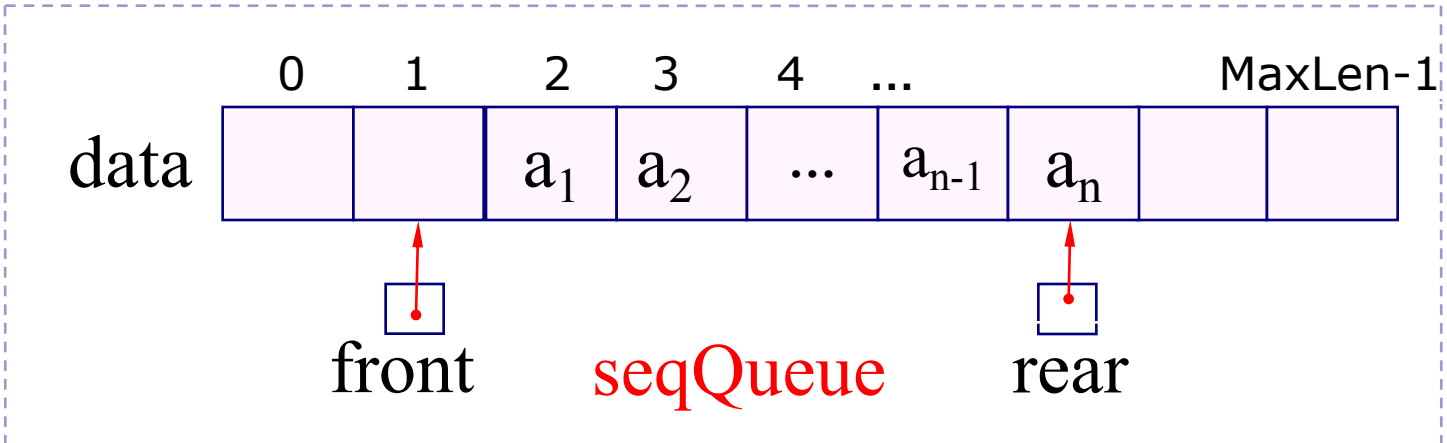
3.2.2 顺序队列讨论

- **(1) front和rear的指向**
  - 如下图所示:
  - **front** 指向队头元素的前一个位置。此在区分队空和队满时有用。**rear** 指向队尾元素。
  - 或者， **front**指队头元素，让**rear**指向队尾元素的下一个单元。



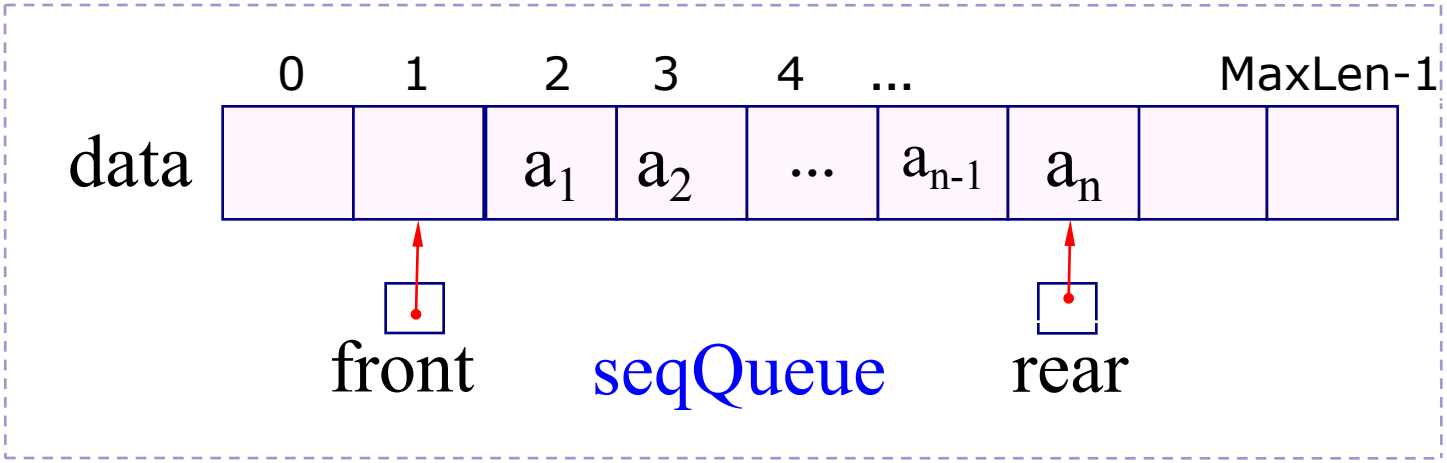
3.2.2 顺序队列讨论

- **(2) 哪端为队头？哪端队尾？**
  - **data[0]队头；data[n-1]队尾。**
- **(3) 如何解决出队移动元素问题？**
  - **出队时只做front++，不移动元素。**
- **(4) 初始化**
  - **front=-1; rear=-1;**



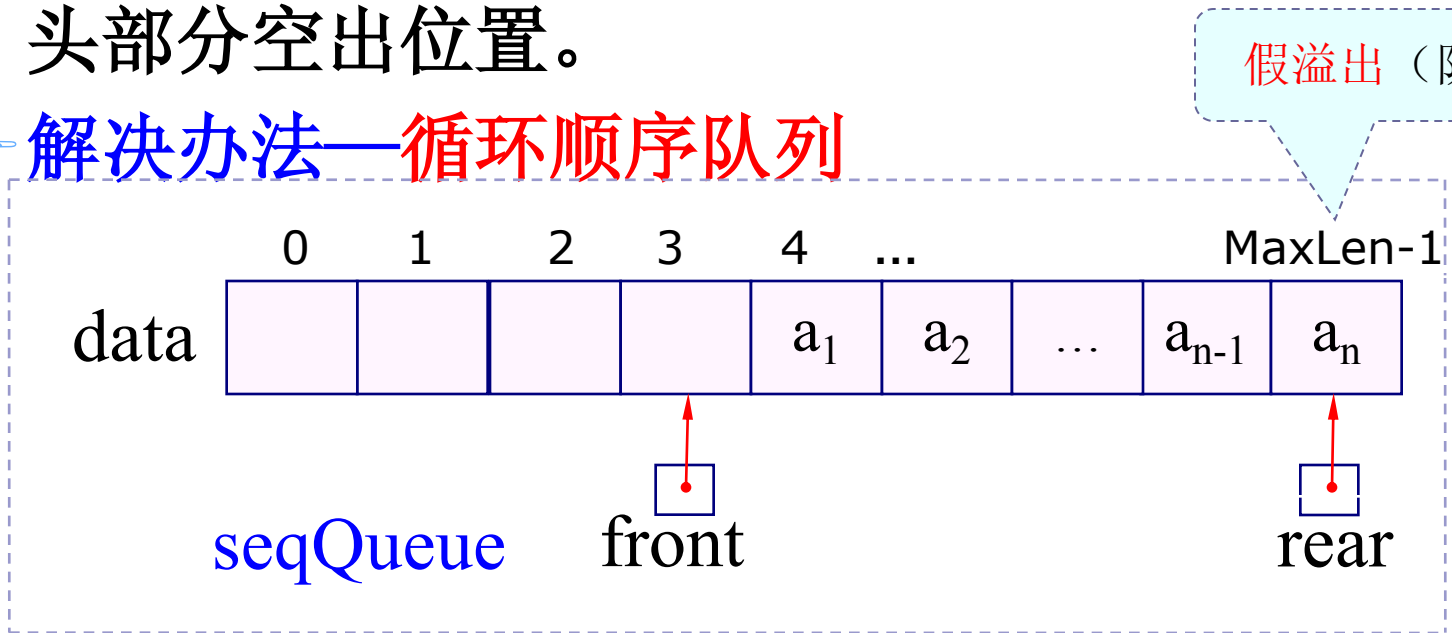
- (5) 队空：front == rear;
- (6) 入队：if(rear != MaxLen-1)
  { rear++; data[rear]=x; }
- (7) 出队：if(front != rear) { front++; }
- (8) 队满：rear == MaxLen-1;

假溢出—>循环队列



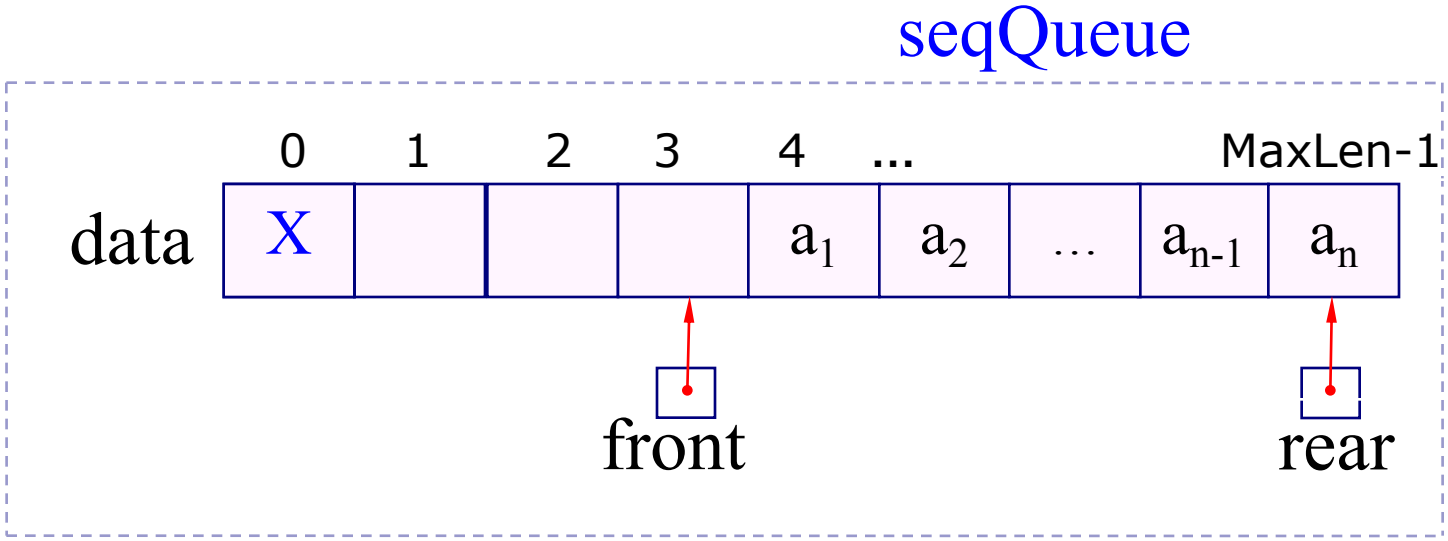
普通顺序队列存在的问题

- 队头不断删除元素，将使队列的头部空出单元；
- 随着删除、插入的进行，将很快队满—假溢出；
- 这一点不象人类排队，人类排队时，队头的人走掉后，后面的人自然会往前移动，不会让队头部分空出位置。
- 解决办法—循环顺序队列

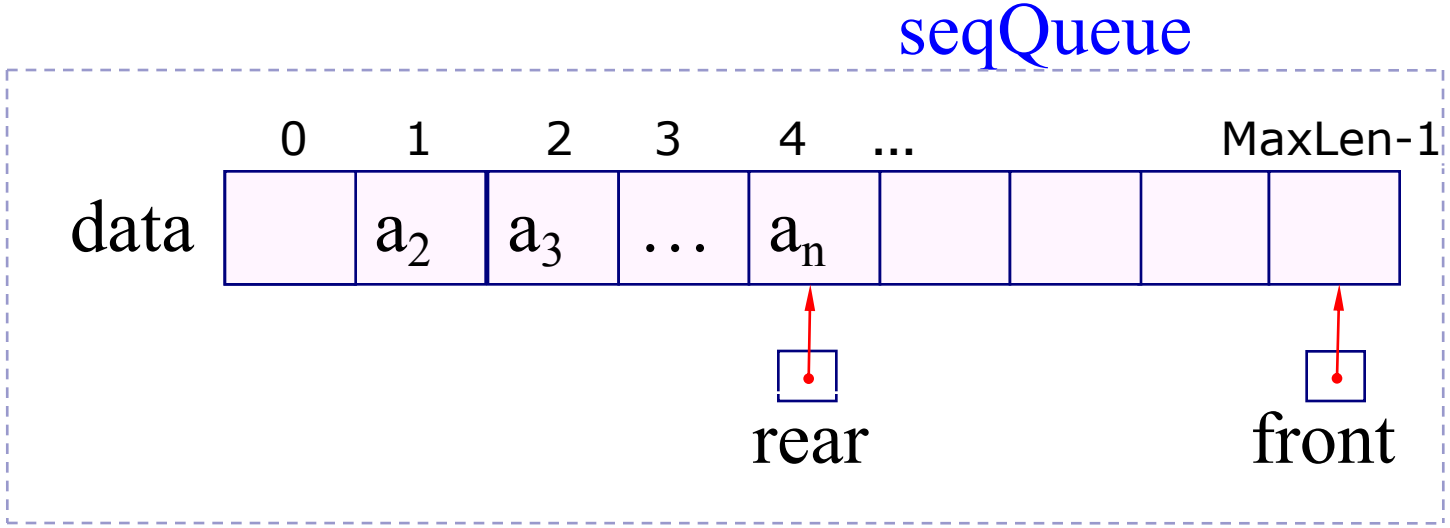


3.2.3 循环顺序队列

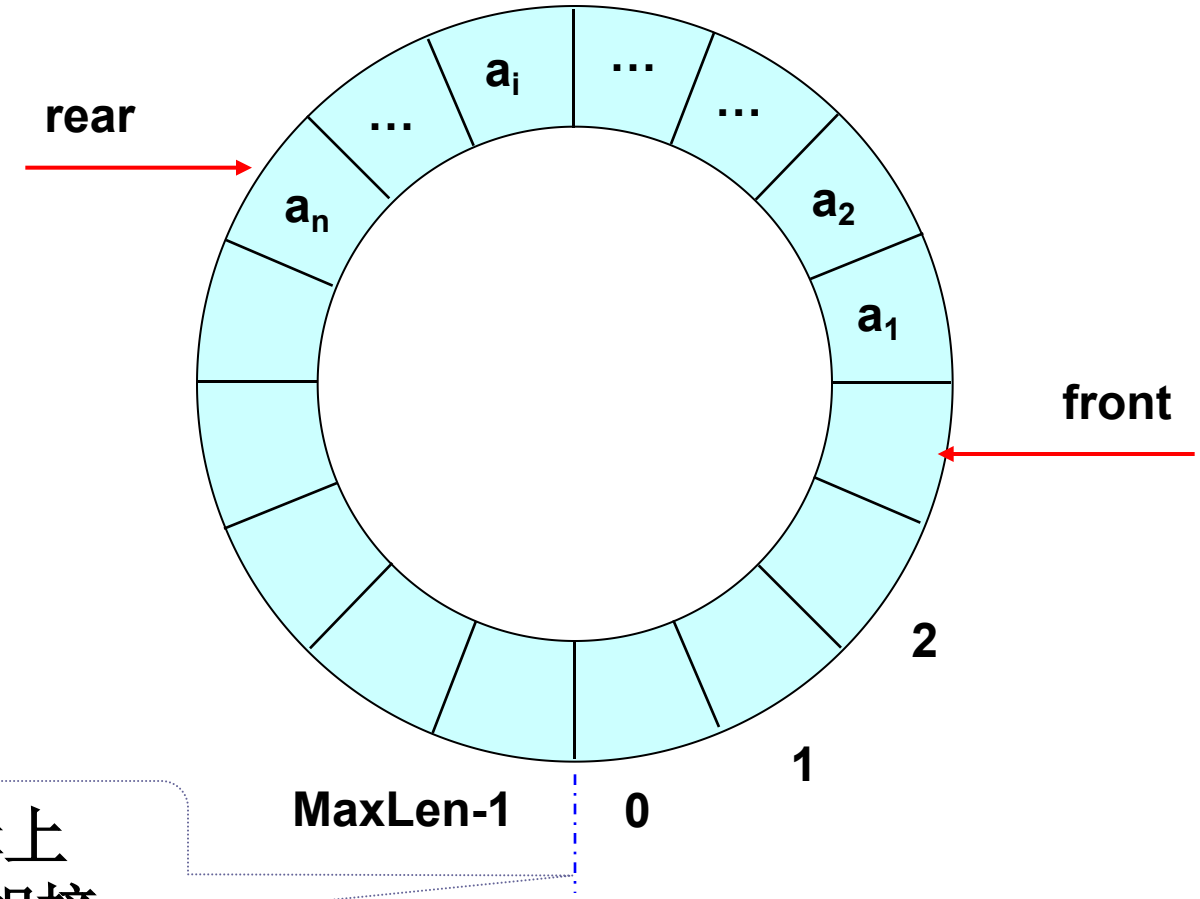
- 将data[0]和data[MaxLen-1]视为相邻单元，首尾相接，形成一个逻辑意义的环（非物理上的）。
  - 当rear=Maxlen-1时，再插入一个元素，让rear=0；



- 同样，**front=MaxLen-1**时，再删除一个元素，也让**front=0**；
- 形成逻辑上的循环。



循环顺序队列示意图



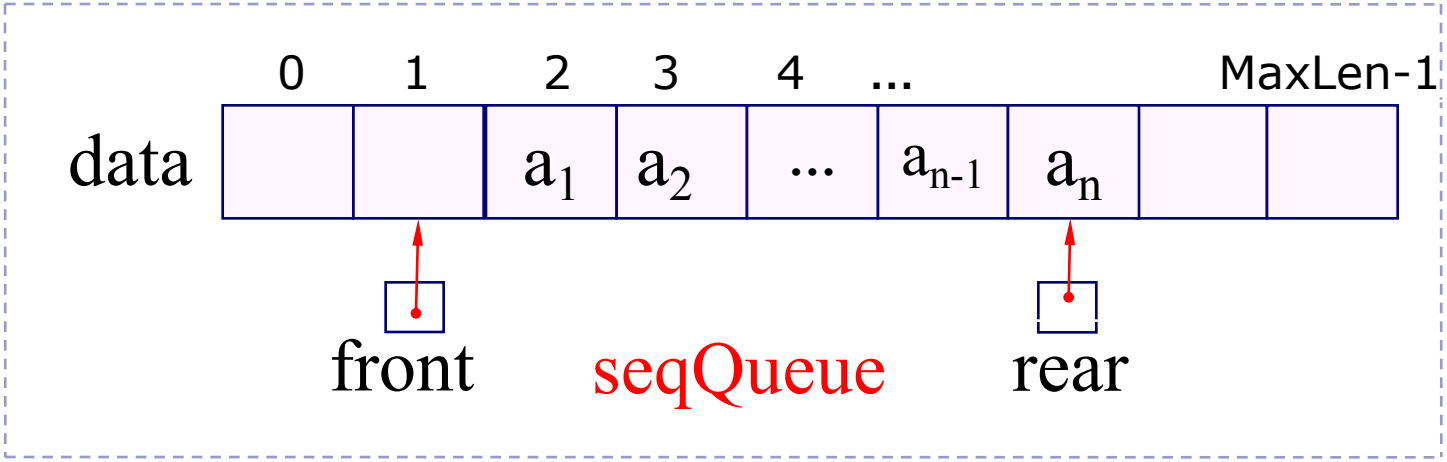
- 入队（插入）操作位置计算
  - 正常操作：rear++; data[rear]=x; 循环队列中，当rear=Maxlen-1时，再进行插入，此时要回到rear=0, 而不能是rear+1=MaxLen，如何实现呢？

(1)方法一：手工判断

```
if( rear+1==MaxLen )
    rear=0;
else
    rear++;
```

  - //简写：rear=(rear+1==MaxLen) ? 0 : rear ++ ;

- (2)方法二：模运算(取余数)
  - rear=(rear+1) % MaxLen;
  - 【例】MaxLen=100，rear=99时插入，则rear+1==100，模运算后rear=100 %100为0，即：rear=0；实现了逻辑循环。
  - rear=75时插入，rear+1=76，rear=76%100=76

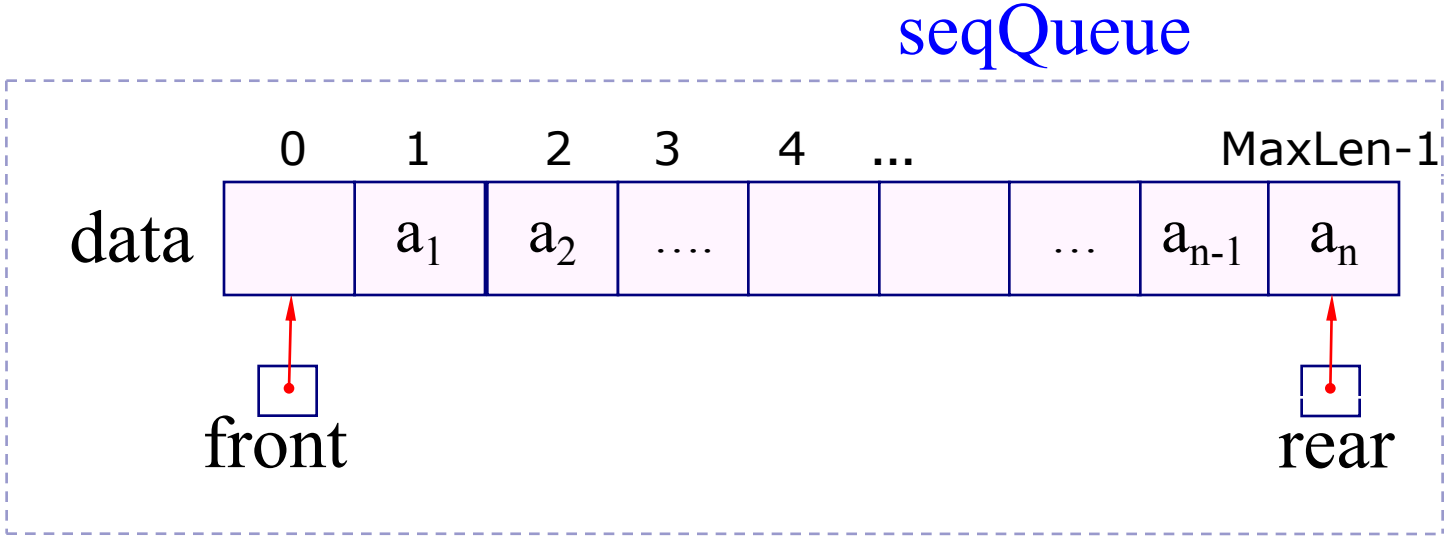


- 出队（删除）操作呢？
  - 出队操作位置计算与入队位置计算方式同样，也有两种方法，其中方法二为：
  - front=(front+1) % MaxLen;

- 如何判定队空和队满呢？
  - 由于循环，队空和队满时都有：rear==front，如何区分出队空和队满呢？--两种方法

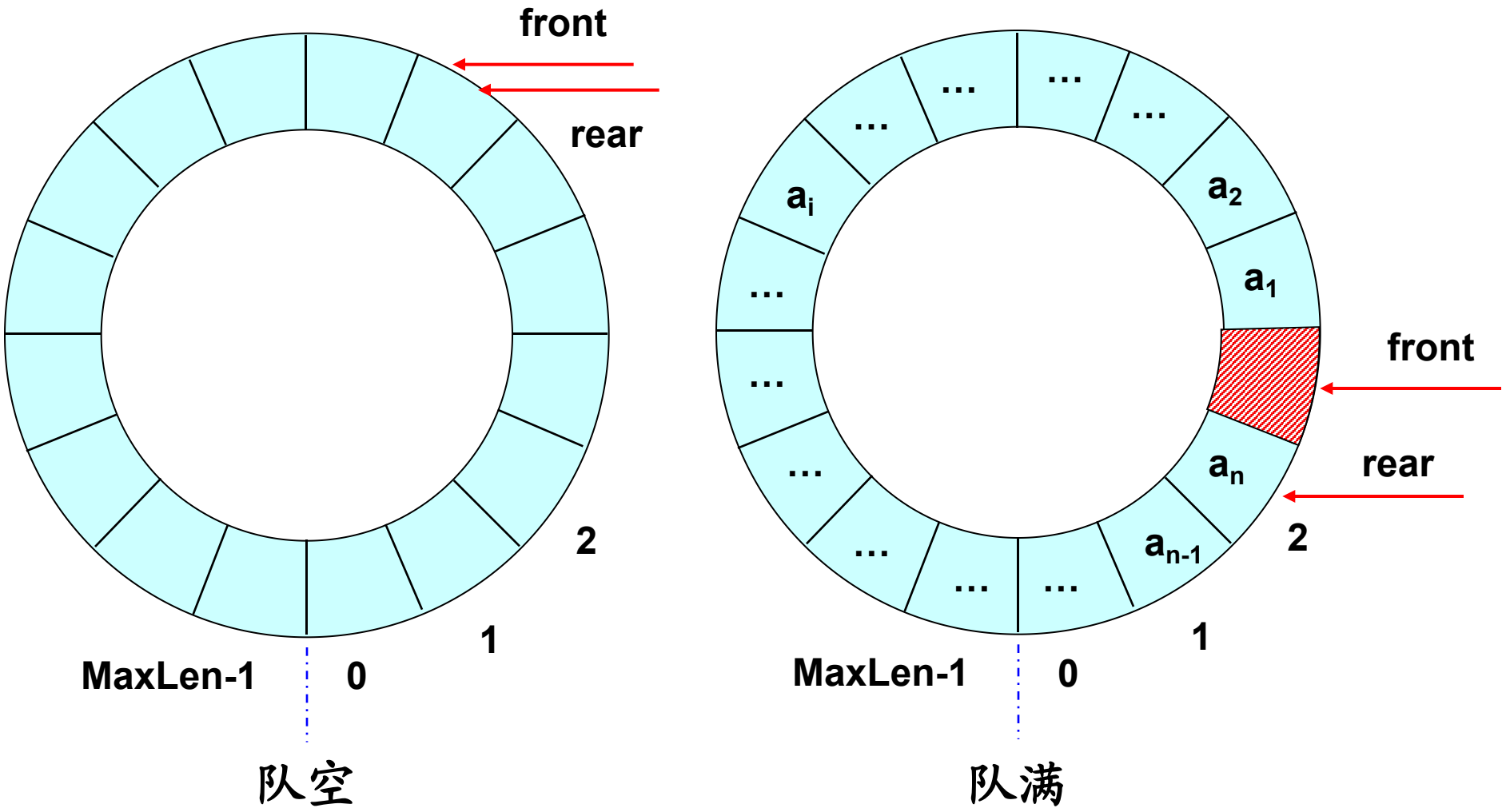
- 【方法一】约定保留一个元素单元:
  - 约定数组最多存放MaxLen-1个元素，front指队头元素的前一个单元；（前面已提到过）
  - 因此，在有数据元素时，front和rear至少相差一个单元，使得头指针front永远“赶不上”尾指针rear；
  - 因而不会出现上述的在满的情况下，头尾指针相等的情况。从而便于判断的实现。
  - 本书即采用这种方法讨论各运算的实现。

- 判定条件：
  - 队列空：front==rear
  - 队列满：front==(rear+1) % MaxLen
- 问题：判定队列满时，为何做模运算呢？
  - 考虑下图的情况



- 【方法二】增加标志
  - 设置一个记录最后的操作是插入还是删除的标志。比如：
    - 1：插入操作；
    - 0：删除操作。
  - 当出现rear==front时：
    - 如果标志是插入（1），则可断定当前队列是 -- ？
    - 如果标志是删除（0），则可断定当前队列是 -- ？

因为反复的入队、出队操作，队空、队满时，**front**和**Rear**指针可能指向**data[]**数组的任何位置，而不是固定在某个位置。



循环顺序队列的运算实现

- **(1)初始化队列:**
  - ☞创建空队列。

```
void initQueue( seqQueue  &Q )
{
      Q.front=0;   //初始首尾都指向0，便于循环处理
      Q.rear=0;
}
```

- **(2)判队空:**

```
BOOL queueEmpty(seqQueue  &Q)
{
    return (  Q.front == Q.rear  );  //简化写法
}
```

- (3)判队满:

```
BOOL  queueFull( seqQueue &Q )
{
    if((Q.rear+1) % MaxLen == Q.front)  //关键
        return TRUE;
    else
        return FALSE;
}
```

//简写: return ((rear+1)%MaxLen == front)

- (4) 取队头:

```
void queueFront(seqQueue &Q, elementType &x)
{
  if (Q.front==Q.rear)
        error("队空");
  else                          //下式为什么取模呢?
       x=Q.data[ (Q.front+1) % MaxLen ];
 }
  //【思考问题】取到的队头用指针变量能否返回?
```

- **(5) 入队**

```
void enQueue(seqQueue & Q, elementType x)
{
  if( (Q.rear+1)%MaxLen == Q.front )  //关键
        error("队满" );
  else
   {
        Q.rear=(Q.rear+1) % MaxLen;     //关键
        Q.data[Q.rear]=x;
   }}
```

(6) 出队

```
void outQueue(seqQueue &Q ,elementType &x)
{
    if ( Q.front == Q.rear )
        error("队空" );
    else
    {
        Q.front=(Q.front+1)%MaxLen;    //关键
        x=Q.data[Q.front];   //捎带取出队头，非必须
    }}
```

【算法分析】

- 上述队列的所有运算，时间复杂度皆为O(1)。

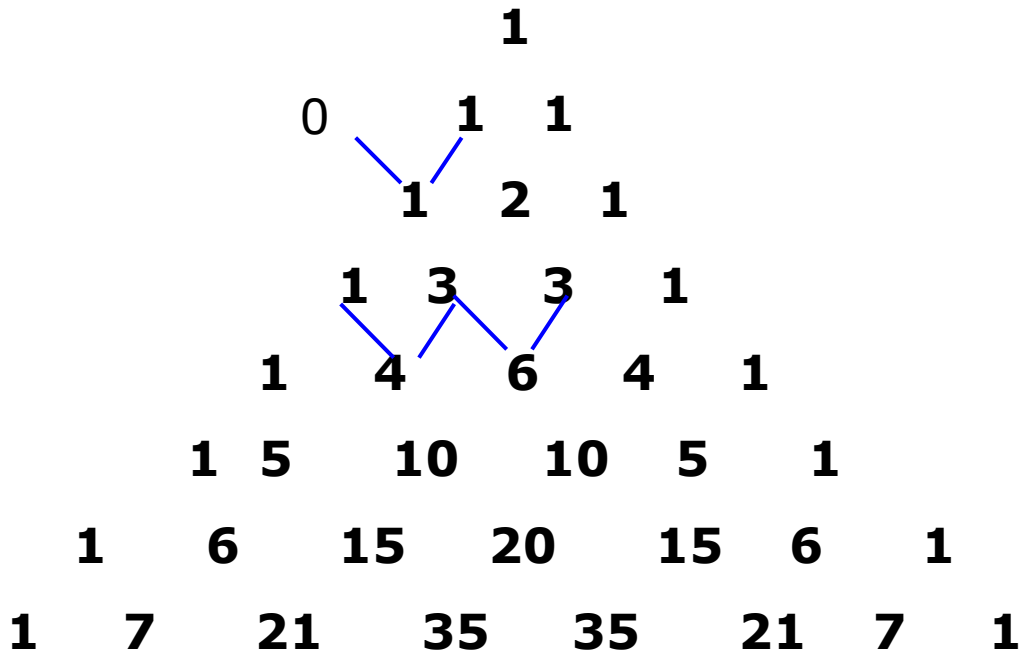
【思考问题】

① 如何求循环队列中的元素个数？

② 如何打印循环队列中的所有元素？

3.2.3 队列的应用

- **【例3.5】杨辉三角**
  - 【实现代码】**Exam401YangHuiTriangle**



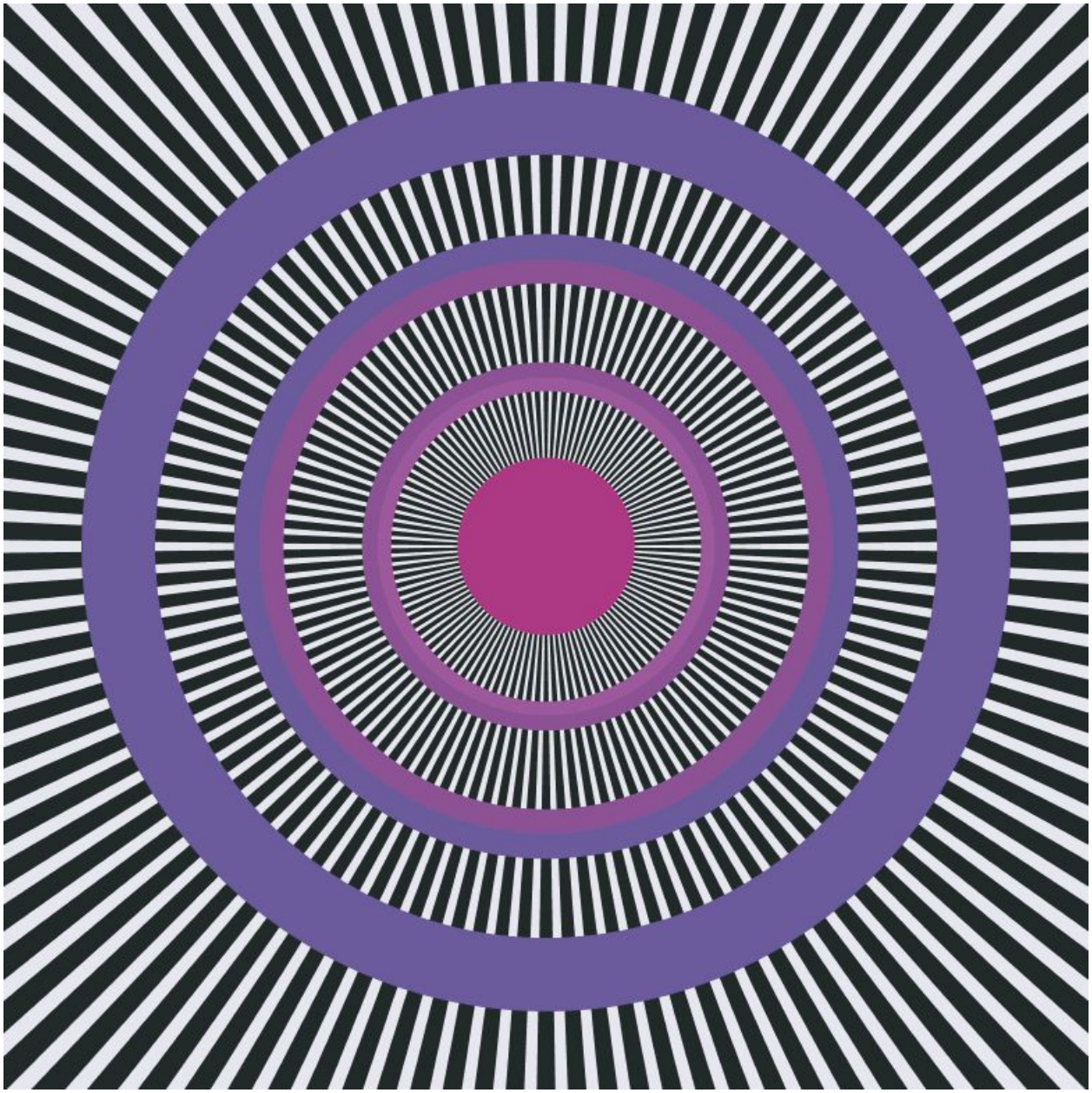
- 【分析】杨辉三角的规律是:
  - 第1行1个数字，…，第n行有n个数字；
  - 每行的第一和最后一个数是1；
  - 从第3行开始，除了首尾的1，其余的数是上一行对应位置的左、右两个数的和。
  - 例如，第7行的第3个数15是第6行中的第2和第3两个数5和10的和。
  - 由这一规律可知，我们可用上一行的数来求出对应位置的下一行的内容。
  - 为此，可用队列来保存上一行的内容。
  - 每当由上一行的两个数求出下一行的一个数时，其中的前一个数便需要出队（删除），而新求出的数就要入队（插入）。

- 优先级队列
  - 插队、特权

【布置作业】

（p96）

- 3.9
- 3.10
- 3.11

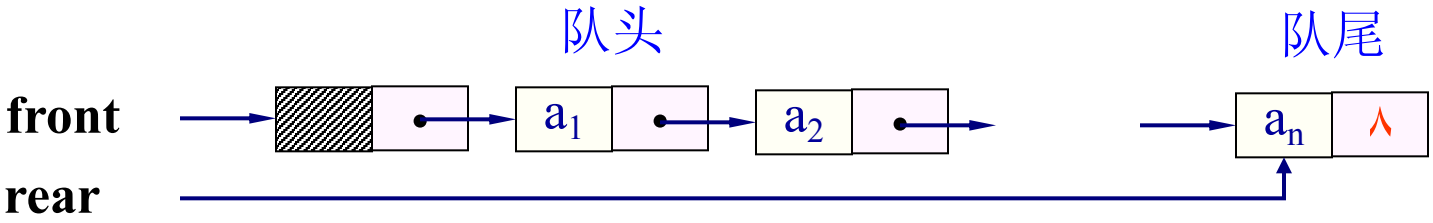


1 链队列的存储结构

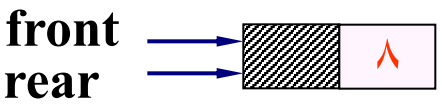
(1) 链队存储结构

- 结点结构同单链表；
- 可以使用连续的存储空间，也可以存储在内存的不同位置。依靠指针建立逻辑上的相邻关系；
- 因为入队、出队分别在队头、队尾进行，所以，需要 2 个指针分别指向队头(front)和队尾(rear)。
- 链式队列也可增加一个同构的头结点。

链式队列结点结构



非空链式队列（带头结点）



空链式队列（只有头结点）

队空的条件：**front==rear**

- 链队结点结构定义

```
typedef struct  LNode
{    elementType   data;   // 数据域
     struct LNode* next;   // 指针域
 } node;    //结点结构定义同单链表
```

- 链队结构定义

```
typedef struct  lQueue
{
     node  *front,  *rear;   //队头、队尾指针
 } linkQueue;
```

【链队列类Queue描述】

```
class Queue
{
public:
    Queue();
    ~Queue();
    bool empty();
    bool getFront(elementType &x);
    void enQueue(elementType x);
    bool outQueue();

    void print();
    int getCount();  //求结点（元素）个数
private:
    int count;
    node* front;      //队头指针
    node* rear;       //队尾指针
};
```

- 【思考问题】

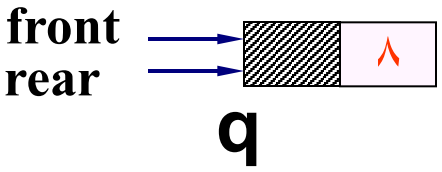
① front、rear和next指针是否类型相同？

2 链队列基本运算的实现

(1)初始化队列

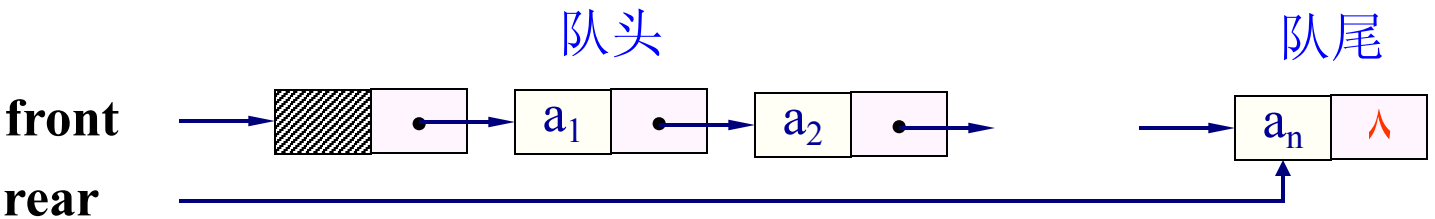
- 建立带头结点的空队列

```
void initQqueue(linkQueue  &q)
{
    q.front = new node;
    q.front -> next = NULL;
    q.rear = q.front;
}
```



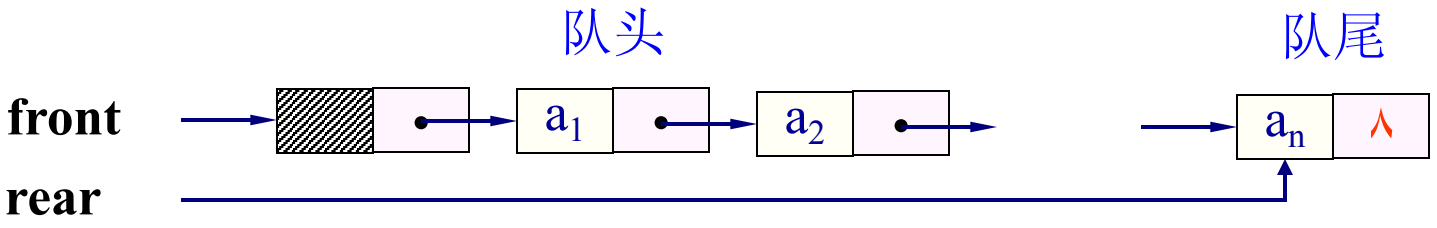
(2)判队空:

```
BOOL  queueEmpty( linkQueue  &q)
{
        return ( q.front == q.rear );
}                    //简略写法
//【简略写法】return ( front == rear );
```



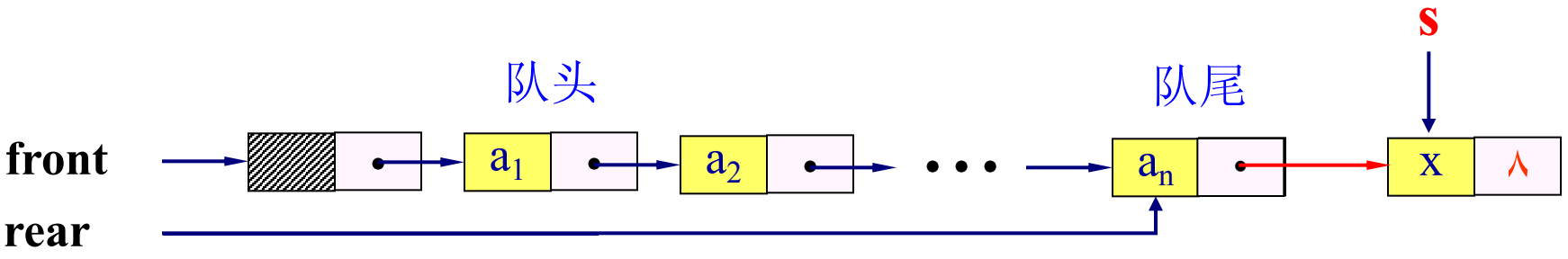
(3)取队头:

```
bool getFront(linkedQueue &Q, elementType &x)
{
   if(queueEmpty(Q))
   return false;     //空队列，无法取队头元素
   else
   {
   x=((Q.front)->next)->data;
   return true;
   }
}
```



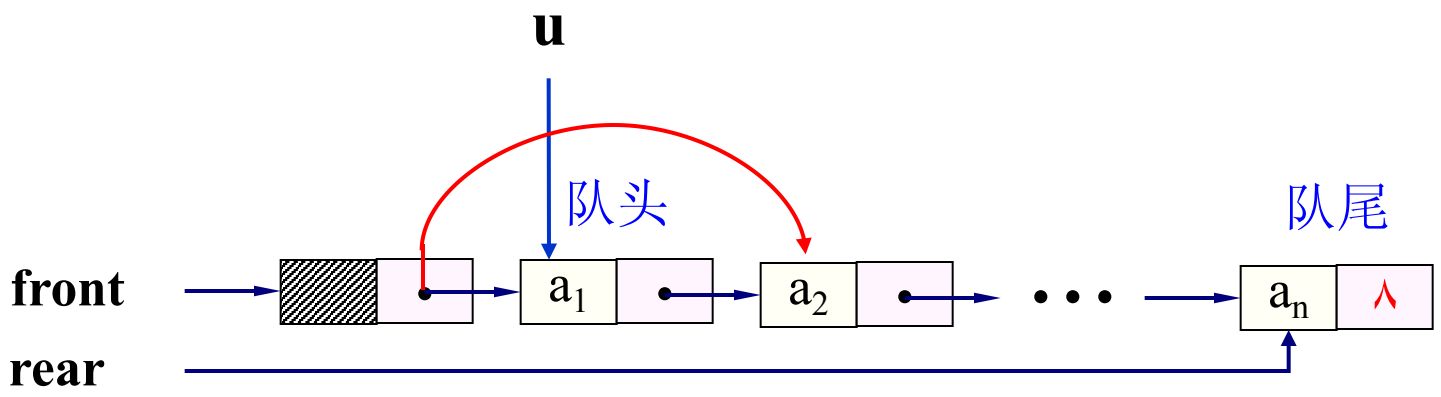
(4)入队—尾插新结点

```
void enQueue(linkedQueue &Q, elementType x)
{
   node* P=new node;  //申请内存，产生新节点
   P->data=x;
   P->next=NULL;
   Q.rear->next=P;
   Q.rear=P; //尾指针指向新的节点（新队尾）
}
```



- **(5)出队（核心代码）**

```
u = front -> next;                  //取队头指针
front -> next = u -> next;          //修改队头指针
delete u;    //删除原来队头结点
```



(5)出队（完整描述）

```
void outQueue( linkQueue &q ,elementType &x)
{ if ( q.front == q.rear )  error(“下溢出”);
  else{
        u = q.front -> next;    //队头
        q.front -> next = u -> next;  //修改队头
        x = u->data;   //取出队头元素值
         delete u;   //删除原来队头
        if ( q.front -> next == NULL )
           q.rear=q.front; //删除后成为空队列情况
       }
}
```

(6) 销毁队列

- 利用析构函数，删除元素结点和头结点。也可以自行设计函数进行销毁。

【算法描述】

```
void destroyQueue(linkedQueue &Q)
{
   node *p,*u;
   p=Q.front;
   while(p)
   {
   u=p;
   p=p->next;
   delete(u);
   }
   Q.front=NULL;
   Q.rear=NULL;
}
```

【算法分析】时间复杂度：O(n)

【算法分析】

- 销毁队列运算的时间复杂度为O(n)，其它运算时间复杂度皆为O(1)。

【思考问题】

① 不带头结点链队列的基本如何实现？

② 循环链式队列如何实现？（分带头结点、不带头结点两种情况）

- 队列的链式存储结构（链队列）特点：
  - 使用连续或不连续的存储空间；
  - 各数据元素独立存储，依靠指针链接建立逻辑相邻关系；
  - 对每个数据元素单独申请结点空间；
  - 由于动态申请空间，没有队满溢出问题；
  - 队头指针front和队尾指针rear 唯一确定一个链队列；
  -

- 链队列通常也加同构头结点，这种情况下：
  - front->next 指向队头元素结点；
  - 若 front->next 为 NULL，则为一空队列，此时，front==rear。
- 销毁队列运算的时间复杂度为O(n)，其它运算时间复杂度皆为O(1)。

- 优先级队列
  - 插队、特权

Do right things right and fast.

3.3 递归

【本节内容】

3.3.1 引言

- **递归(Recursion)**
  - 人类解决问题“分而治之”思想；
  - 递归是一种思考问题的方式；
  - 递归是一种解决问题的方法；
  - 递归是问题归约或分治法的一种。
  - 递归和数学归纳相关。

- 计算机中：
  - 递归是一种程序的形式，是一种特殊的函数调用，在函数体内调用函数自身；
  - 递归更是一种程序设计（算法设计）的技术；
  - 递归与栈技术的应用密切相关。

- 理解掌握递归技术的难点
  1. 递归的理解；
  2. 递归程序的阅读；
  3. 递归程序的验证和编写；
  4. 递归程序的运行时间及解决；

3.3.2 递归的定义及其一般形式

1 递归程序的定义

(1) 递归定义:

- 在函数（过程、子程序）体中直接或间接地引用自身，则称之为递归函数。
- 如果一个对象全部或部分地包含它自己，或者利用自己定义自己的方式来定义或表述，则称这个对象是递归的。

■ 递归可以分为（多种分类方法）:

- 直接递归—函数体中调用自身;
- 间接递归—本函数调用其他函数，其他函数又调用本函数。

2. 几个递归函数实例

$$n!=\begin{cases}1 & n=0\\ n(n-1)! & n>0\end{cases}$$

(1)阶乘

【算法描述】

$$Fact(n)=\begin{cases}1 & n=0\\ n*Fact(n-1) & n>0\end{cases}$$

```
int Fact( int n )
{
  if(n==0)
  return 1;   //终止条件
  else
  return n*Fact(n-1);  //递归调用
}
```

(2) Fibonacci数

【算法描述】

$$Fib(n)=\begin{cases}0 & n=0\\ 1 & n=1\\ Fib(n-1)+Fib(n-2) & n\geq 2\end{cases}$$

```
int Fib(int n)
{
  if( n==0 )
     return 0;   //终止条件
  else if(n==1)
     return 1;   //终止条件
  else
    return Fib(n-1)+Fib(n-2);  //递归调用
}
```

(3)两次递归调用

【算法描述】

```
void P(int n)
{
    if(n>0)   //终止条件：n<=0
    {
        P(n-1);    //递归调用
        cout<<n;
        P(n-2);    //递归调用
    }
}
```

(4) 间接递归

```
void P1(int n)
{
  if(n>0)
  {
  cout<<n;
  P2(n-1);
  }
}
```

```
void P2(int n)
{
  if(n>0)
  {
   P1(n-1);
   cout<<n;
  }
}
```

3.3.2 递归程序的一般形式

```
void  RecFunc(参数表)
{
    if( 递归出口条件 )
        简单操作;    //递归出口（终止条件）
    else
    {
       简单操作;
         RecFunc(参数表);        //递归调用
       [简单操作;]
         [RecFunc(参数表);]    //可能有多次递归调用
         [简单操作;]
       }
} //这里给出的函数形式没有返回值，但有的递归函数
  是需要返回值//的，且返回值还可能参与运算
```

- 递归调用需具备的条件

(1) 原问题能逐步分解为更简单的子问题，子问题与原问题定义、求解方法相同，只是参数不同；

(2) 正向递推分解子问题的过程必须有明确的结束条件，叫做递归终止条件，或递归出口。即把问题反复分解，直至产生一个可以直接求解的基本问题集合，正向递推结束。

☞ 如果没有明确的递归出口，正向分解过程将一直进行下去，对计算机程序来说，这相当于“死循环”，直至栈溢出错或系统崩溃。

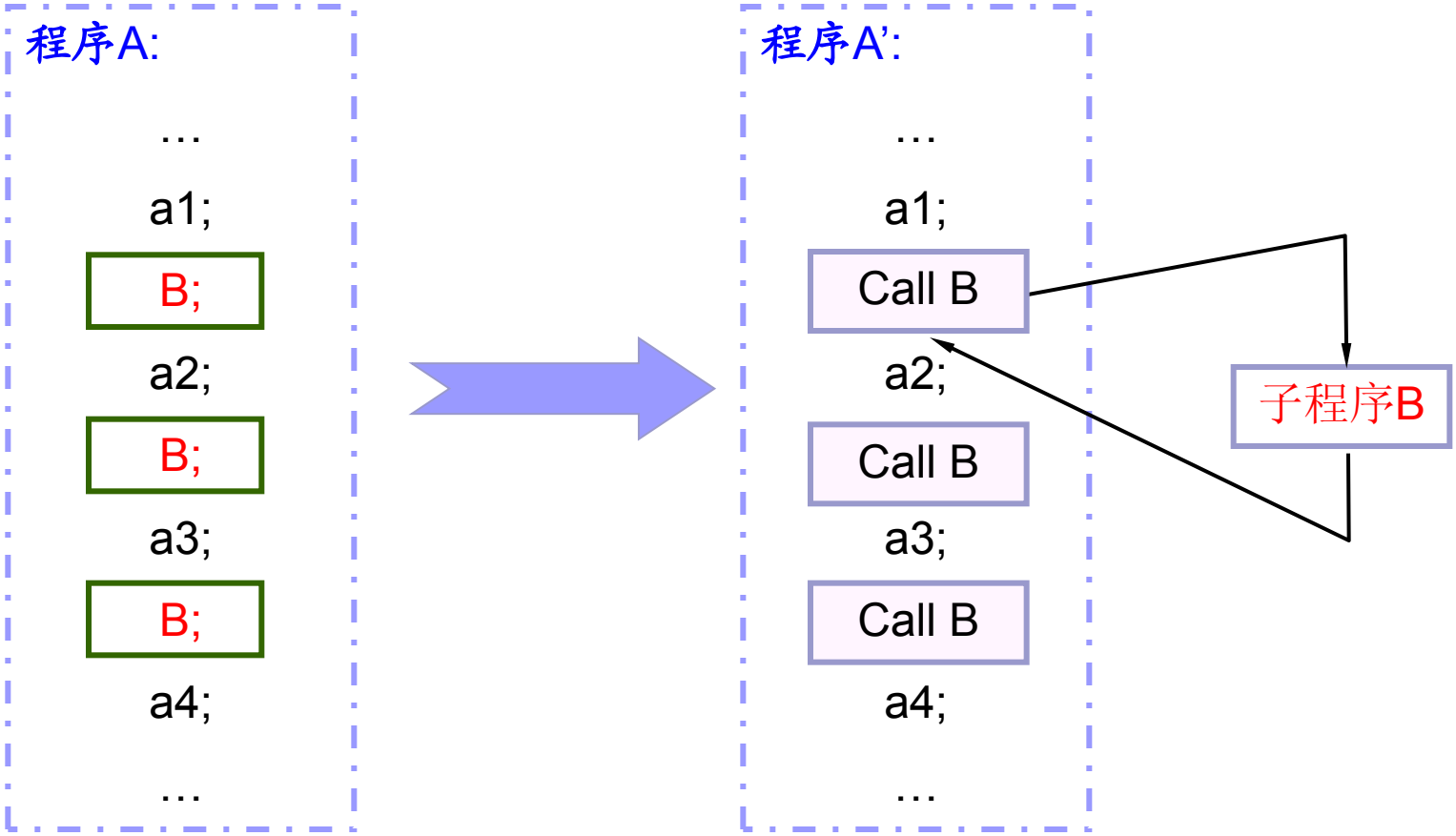
(3)通过基本问题的解可以反向回归合成出原问题的解。

- 可见，递归求解分为两个过程：
  - 一个是正向递推分解子问题的过程；
  - 另一个是通过基本问题的解反向回归合成原问题解的过程。

3.3.3 递归的内部实现原理

1 一般函数调用的实现

☞递归调用是一种特殊的函数调用形式。



1. 函数调用要解决的问题

①如何找到子函数（过程）的入口

- 我们知道每个函数都有函数名，编译成机器代码后，函数名就变为子函数第一条指令的存储地址，当调用此子函数时，比如上面的call B，CPU的程序指针指向这个地址（函数名），就找到了子函数的入口。

②如何向子函数传递参数

- 许多情况下函数带有参数，比如前面给出的实例（1）求阶乘的实例（2）求Fibonacci数，执行函数调用时，需要主调函数向子函数传递这些参数（实参），这个工作是用栈来完成，即在转入子函数执行之前，把实参入栈，详情见稍后的讨论。

③子函数执行完毕如何找到返回位置（返回地址）

- 子函数执行完毕，应该返回到主调函数继续执行，即返回到主调函数调用位置后面的第一条指令继续执行。
- 所以在转入子函数执行之前，需要把主调函数调用指令后面第一条指令的地址“告知”子函数，以便子函数执行结束能正确返回，这个工作也是通过将返回地址入栈完成的。

④子函数本地变量存储问题

- 子函数的本地变量也入栈保存，这样做对递归调用特别有用，参见下面的讨论。

⑤函数返回值问题

- 子函数可以通过函数参数和函数返回值两种方式往主调函数回传数据。数据回传是通过在主调函数和子函数间共享内存实现的。
- 函数参数回传数据时，先要在主调函数中定义一个变量，比如x，为变量x在内存中开辟一块存储空间，然后把变量x的地址传递到子函数，子函数通过x的地址操作这块存储空间改变x的值，主调函数也是操作这块空间改变x的值。这样主调函数和子函数改变x的值，操作的是同一块内存空间，所以子函数中改变了变量x的值，主调函数是能够“感知”到的，从而实现了数据的回传，C++中指针和引用回传数据就是这样实现的。

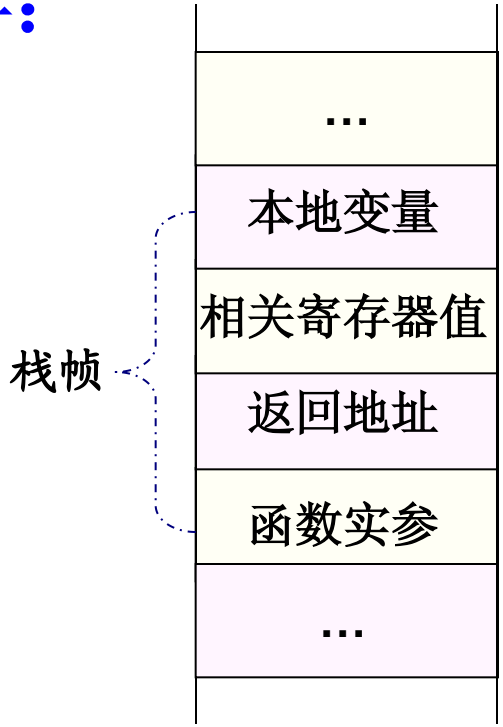
- 函数返回值回传数据情况稍微复杂一些，
- 对于字节数较少的返回值，直接通过CPU的寄存器回传，即子函数返回时，把要返回的值存放到CPU的寄存器中，返回后主调函数直接到CPU指定寄存器中读取这个值即可。比如对于32位机，4字节以下的返回值，用单个寄存器返回；5-8字节的返回值用2个寄存器返回。

- 对于较大的返回值，即长度超出2个寄存器长度的返回值，也是转换为参数进行传递的，只是参数是隐藏的。这个工作由编译器自动完成，编译器根据函数返回值类型，计算出需要的存储空间，如果超出2个寄存器长度，它就自动定义一个隐藏变量，开辟一块内存，调用子函数之前，把隐藏变量的地址传递给子函数，与②中讨论的实参传递一并完成，就是主调函数向子函数传递实参时，多了一个作为函数返回值的隐藏变量，可见这就是参数回传。

- 为了简化讨论，我们假设有一个“回传变量”，通过此回传变量把函数返回值从子函数返回到主调函数。

2. 函数调用栈

- 系统为每个线程的函数调用开辟一块存储区域，叫函数调用栈（call stak），如图所示：
- 每调用一个函数，就在调用栈中开辟一块区域，叫做栈帧（Stack Frame）保存：
  - 函数实参
  - 返回地址
  - 子函数的局部变量
  - 相关寄存器的值（细节不介绍）
  - 函数返回值（包含在实参中）



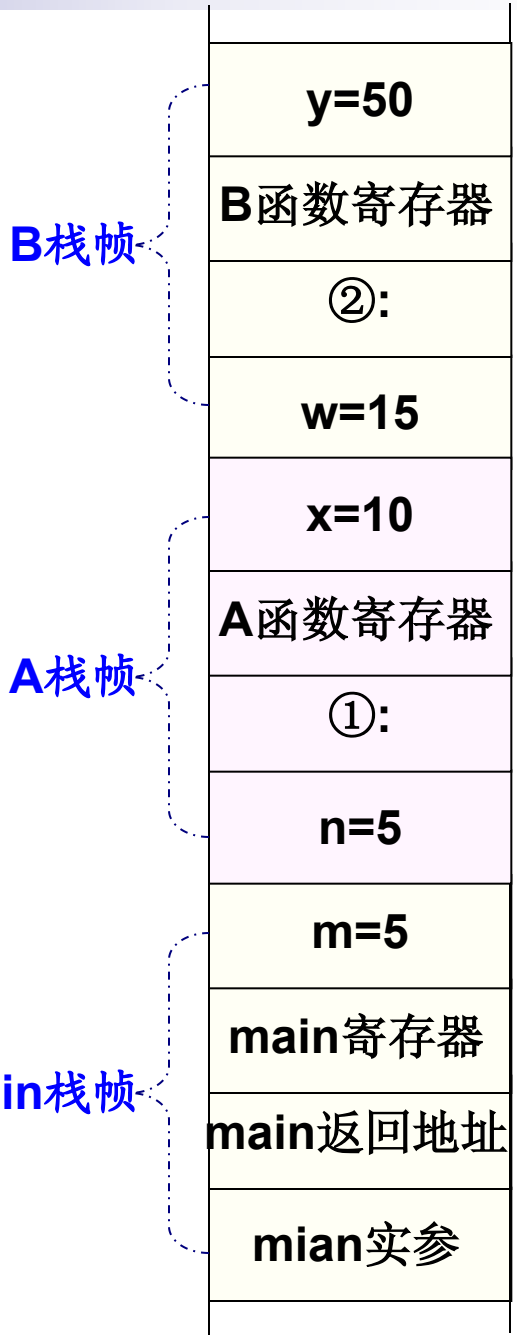
- 调用函数时，相关信息入栈，形成栈帧；子函数执行结束，相关信息出栈，对应的栈帧释放。
- 当函数有嵌套调用时，每调用一次函数就形成一个栈帧，调用栈中就会同时存在多个栈帧。按栈的先进后出原则，当前正在执行的函数的栈帧处于栈顶位置，叫做活动栈帧，也叫活动记录（AR—Active Record）。

■ 例:

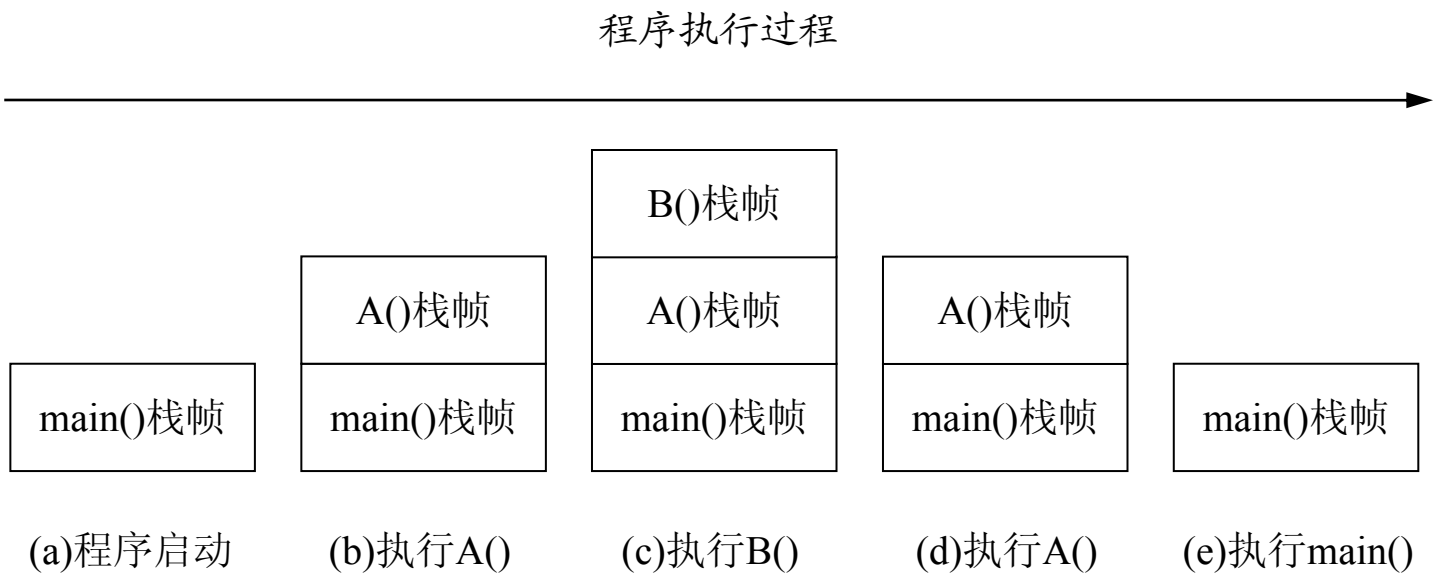
```
int main()
{
      int m=5;
①:   m=A(m);
     cout<<m;
     return 0;
}
```

```
int A(int n)
{
      int x=10;
②:    x=B(x+n);
      return x+n;
}
```

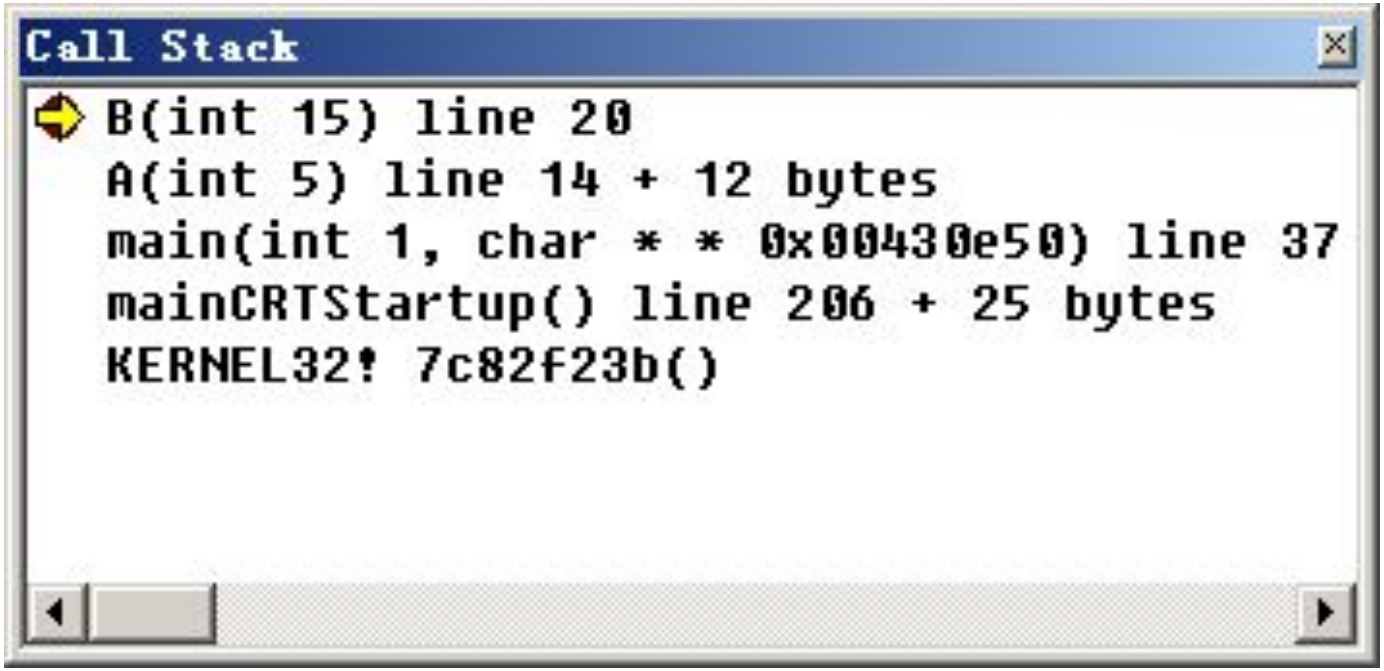
```
int B(int w)
{
     int y=50;
     return y+w;
}
```



实例中调用栈的使用过程



- **vc6.0查看上面实例的函数调用栈变化情况**
  - 设置断点，单步执行即可查看。



2 递归调用的内部实现原理

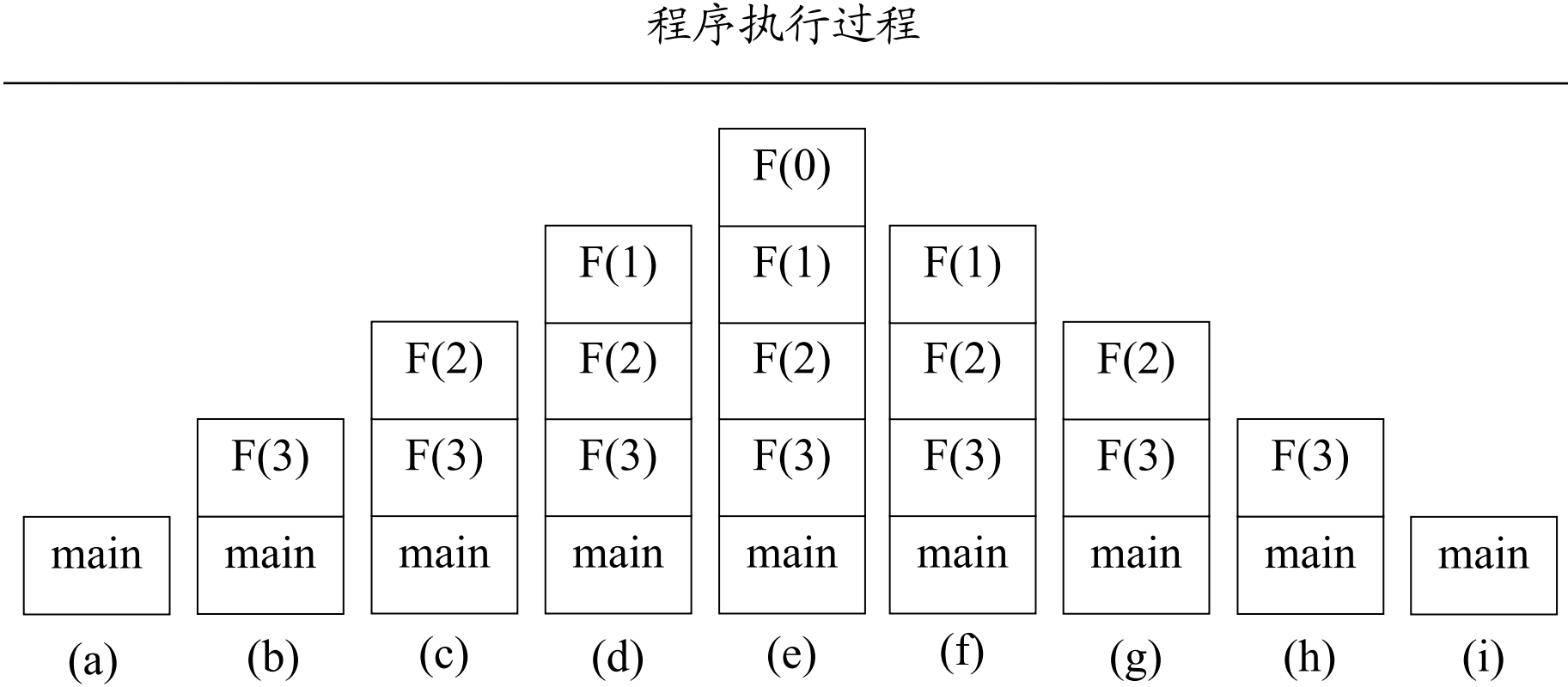
- 递归调用与一般函数调用的内部实现原理完全相同。
- 不同的只是递归调用时，每次调用的都是函数自身，即每次递归调用执行的代码都相同（调用自身），但每次调用时的函数参数都不同，函数的本地变量值也可能不同。

- 例：主函数中调用Fact(3)
  - 省略相关寄存器值

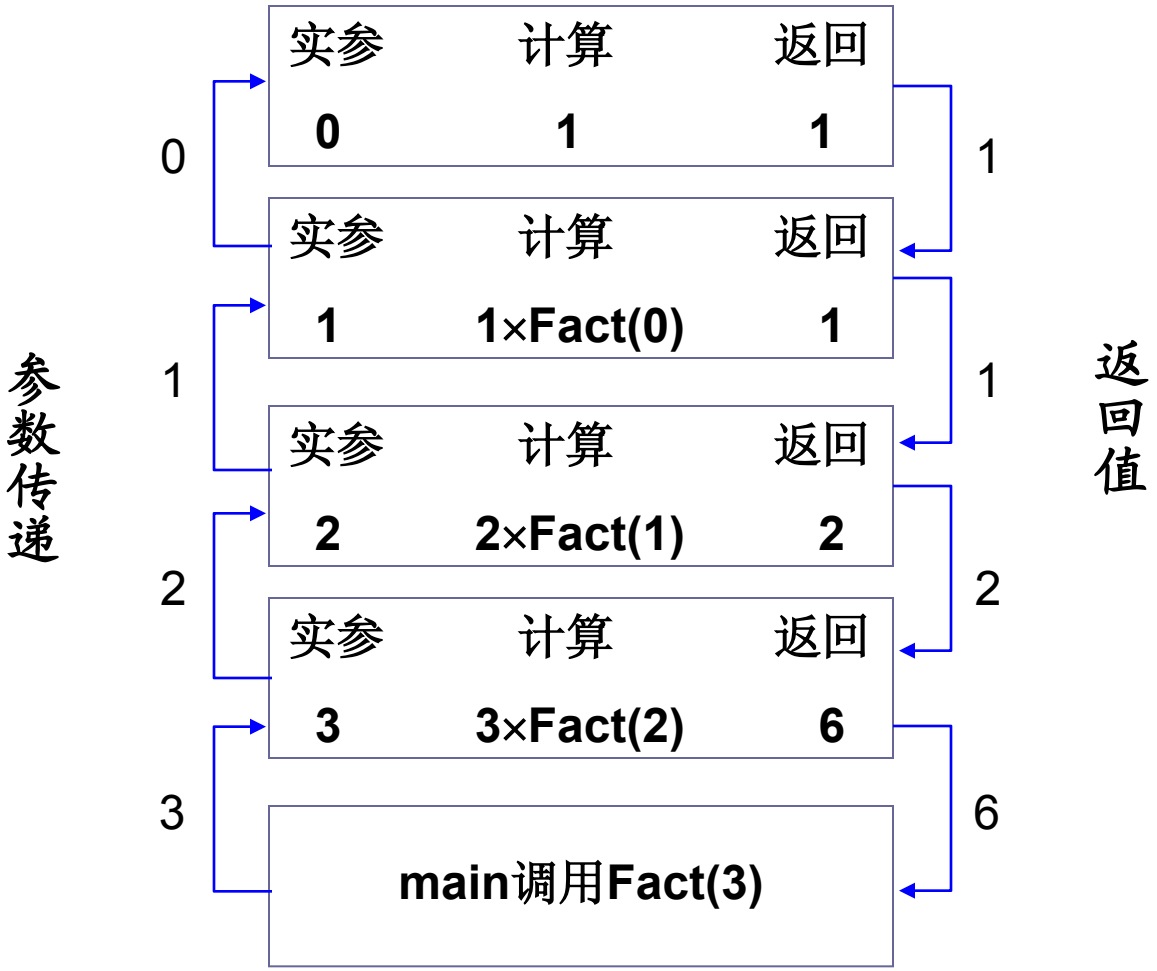
```
int main()
{
    Fact(3);
①: cout<<“OK”;
    return 0;
}
```

```
int Fact( int n )
{
   if(n==0)
      return 1;
   else
②:    return n*Fact(n-1);
}
```

实例中递归调用中调用栈的变化过程



Fact(3)调用过程


参数传递
返回值
实参 计算 返回
0 1 1
实参 计算 返回
1 1×Fact(0) 1
实参 计算 返回
2 2×Fact(1) 2
实参 计算 返回
3 3×Fact(2) 6
main调用Fact(3)
0
1
2
3
1
1
2
6

- VC6.0查看递归调用Fact(6)的调用栈情况

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

Call Stack

```
Fact(int 2) line 13
Fact(int 3) line 13 + 12 bytes
Fact(int 4) line 13 + 12 bytes
Fact(int 5) line 13 + 12 bytes
Fact(int 6) line 13 + 12 bytes
main(int 1, char * * 0x00440e50) line 41
mainCRTStartup() line 206 + 25 bytes
KERNEL32! 7c82f23b()
```

- 许多材料上会笼统的说递归调用会耗费内存空间，什么原因呢？
  - 从上面的讨论和例子中可以清楚看出这一点。系统为每个线程开辟的函数调用栈空间大小是有限的。每一次函数调用都要在调用栈上建立一个函数调用栈帧。栈帧的大小由函数参数的个数及大小、返回地址、相关寄存器值和本地变量的个数及大小共同确定。其中，返回地址和相关寄存器值占用空间相对固定。但函数参数和本地变量的个数和大小在不同的函数中都是不同的。

- 不管怎样每建立一个栈帧就会消耗掉调用栈上的部分存储空间。所以，递归调用中，正向递推的过程是不能无限加深的，递归每加深一次，至少会新建一个栈帧，消耗掉调用栈的部分空间，这样迟早会造成函数调用栈的溢出（调用栈满）。特别是栈帧较大时，更容易溢出。许多读者在平时运行一些软件时，所见到的“**stack overflow**”或“栈溢出错”就是因为递归调用或嵌套函数调用层次太深，造成函数调用栈满溢出的情况。

3.3.4 递归程序的阅读和理解

- 一种图形化的递归程序阅读方法。
- 以每次调用的函数为主要结点，因为每次递归调用的函数名相同，用实参加以区分。
- 两条主线
  - 正向递推线
  - 反向回归线（回溯线）
- 绘图步骤

（1）根据调用情况，画出每个调用函数作为图的主要结点，以实参进行区分，用有向边连接各个结点，表明调用层次和调用点；

（2）从递归出口点开始，逐次回归，返回线路也用有向边链接，直到递归起始函数，给出整条反向回归线。在绘制返回线时要特别注意函数的返回点；

（3）在正向递推线相关位置，画出函数执行的其它计算、打印等结果；在反向回归线的相关位置，画出函数的返回值或修改的返回参数；

（4）重绘上面得到的草图，使更加美观和直观。从这个图上就可以清晰的见到递归的运行轨迹和程序的执行结果。

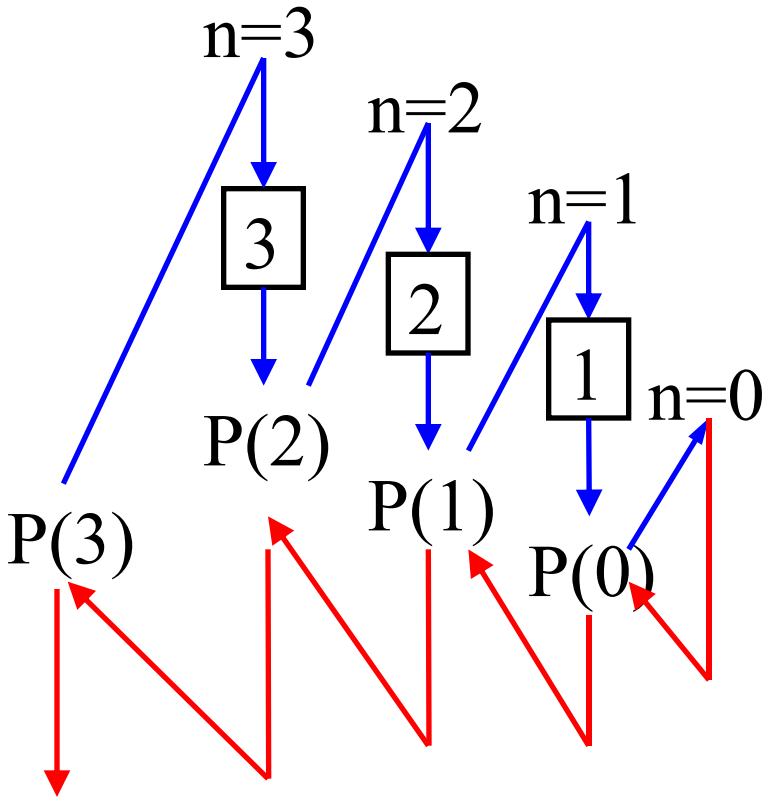
3.3.4 递归程序的阅读和理解

1. 线性递归及执行路线图

☞ — 函数体中只有一次递归调用，画起来相对简单。

【例p1】对下面函数，给出p(3)的执行过程和运行结果。

```
void p(int n)
{   if(n>0)
    {
      printf( “%d” ,n);
      p(n-1);
      //调用返回点
    }}
```

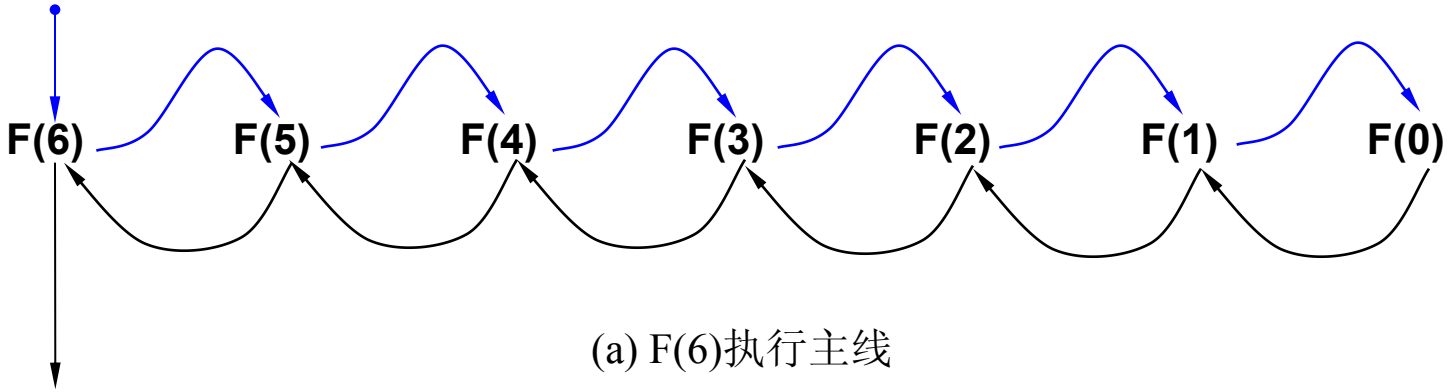


■ 运行结果：3 2 1

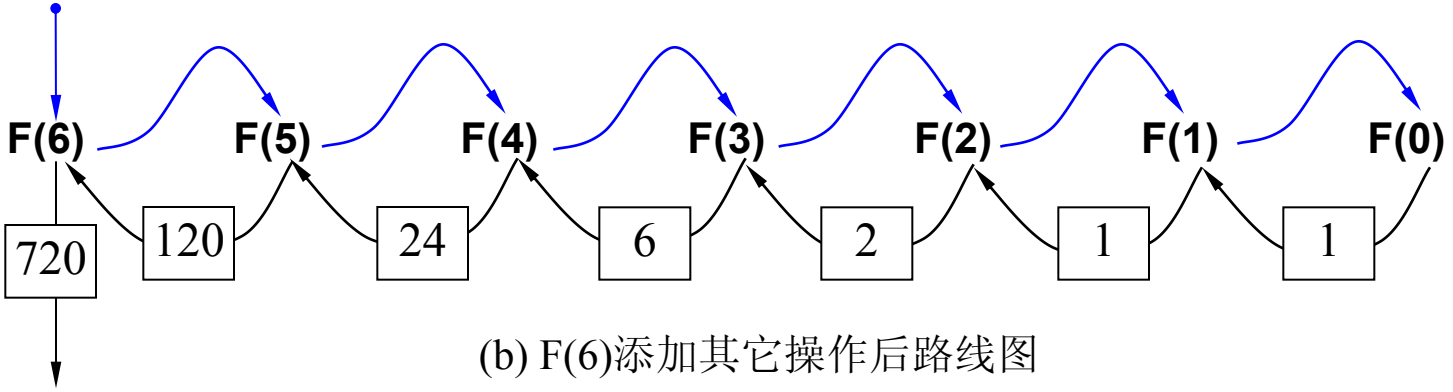
【例3.6】画出求阶乘函数F(6)的执行路线图，并给出执行结果。

【解】

先画“主线”，再补充数据和结果。



(a) F(6)执行主线



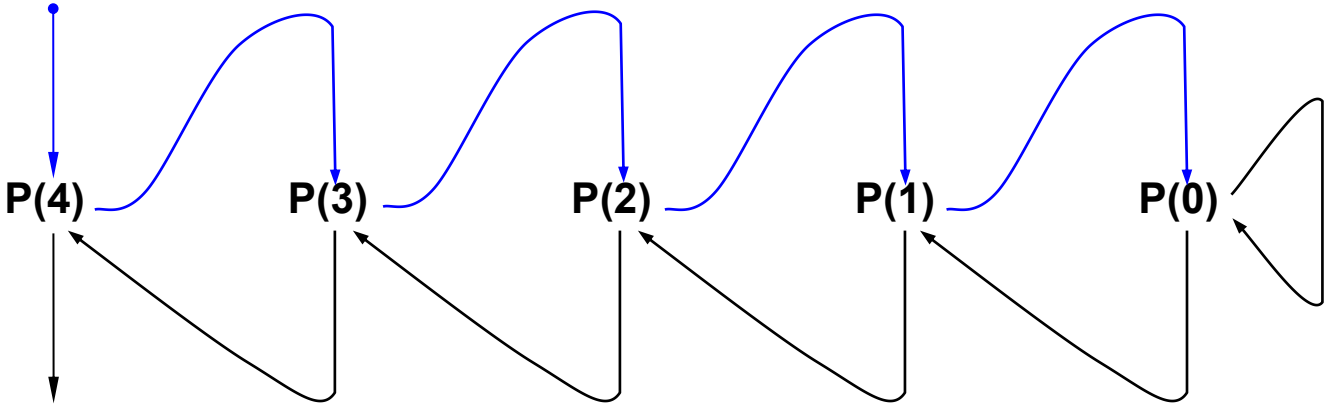
(b) F(6)添加其它操作后路线图

【例3.7】对下面的程序代码，画出P(4)的执行路线图，并给出执行结果。

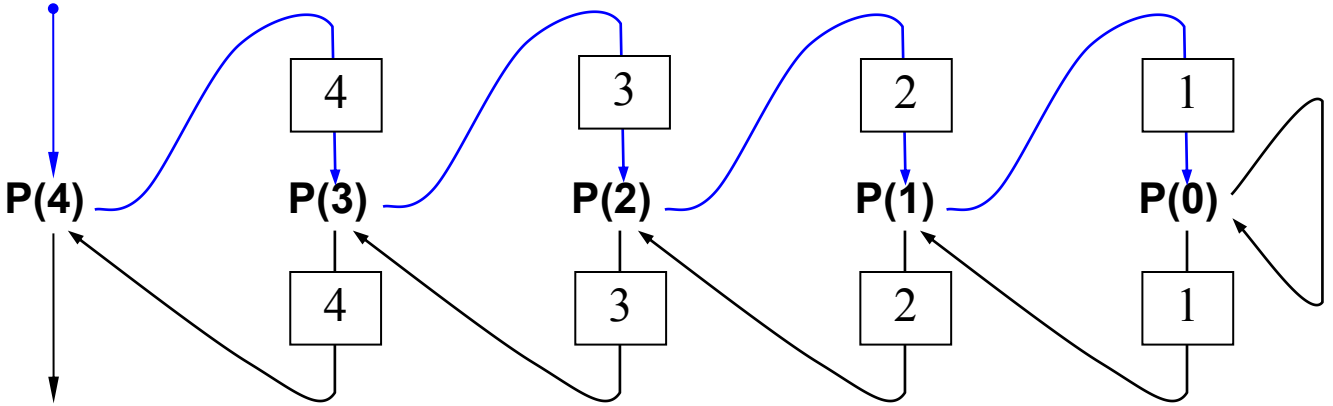
【解】

先画“主线”，再补充数据和结果。

```
void P(int n)
{
    if(n>0)
    {
        printf(“%d”,n);
        P(n-1);
        printf(“%d”,n);
     }
}    //【思考问题】一次调用中，
     //2个打印语句打印的变量n的值是否相同？
```

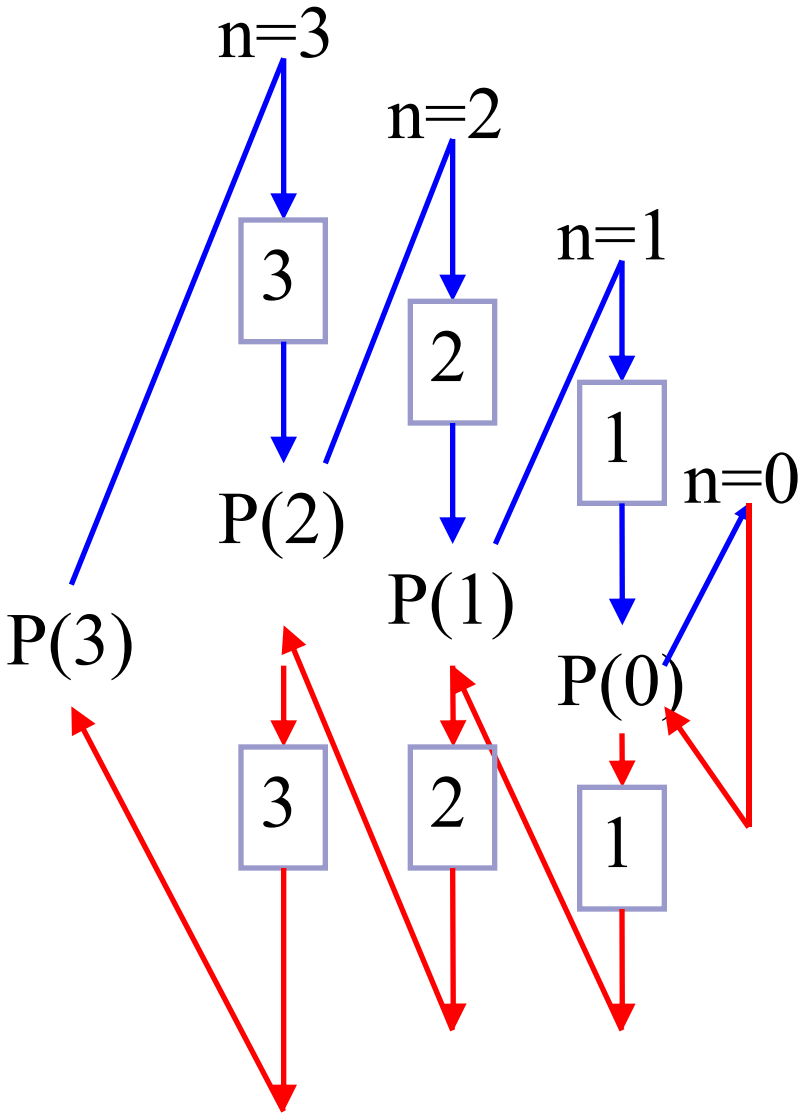


(a)P(4)执行主线



(b)P(4)添加其它操作后路线图

```
void p(int n)
{
    if(n>0)
    {
        printf( “%d” ,n);
        p(n-1);
        printf( “%d” ,n);
     }
}
```



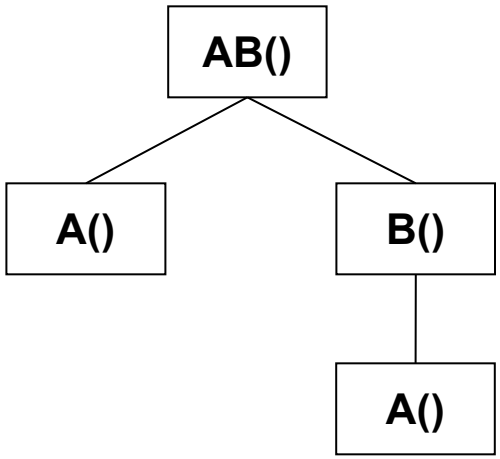
- 运行结果：3 2 1 1 2 3

2. 树形递归及其执行路线图

- 函数体内有两次或两次以上的递归调用，我们称这样的递归为树形递归。
- 我们在画这种递归的执行路线图时，如果只保留正向递推线路，省略其它所有东西，将得到一棵递归调用树，这也是其名称的由来。
- 在画完整的执行路线图之前，我们先画出其递归调用树，有助于我们理解递归执行过程，然后再在树上添加反向回归路线，以及其它计算和处理，这样比直接画出执行路线图要简单的多。

【例3.8】对下面的程序代码，画出调用AB()的执行路线图，并给出执行结果。

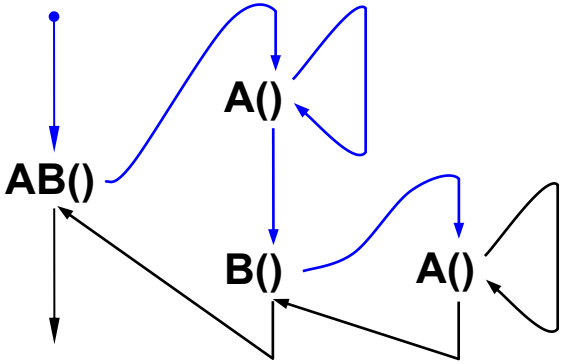
```
void A( )
{    cout<<'A';
}
void B( )
{
    cout<<'B';
    A( );
    cout<<'B';
}
void AB( )
{
    cout<<'AB';
    A( );
    B( );
    cout<<'AB';
}
```



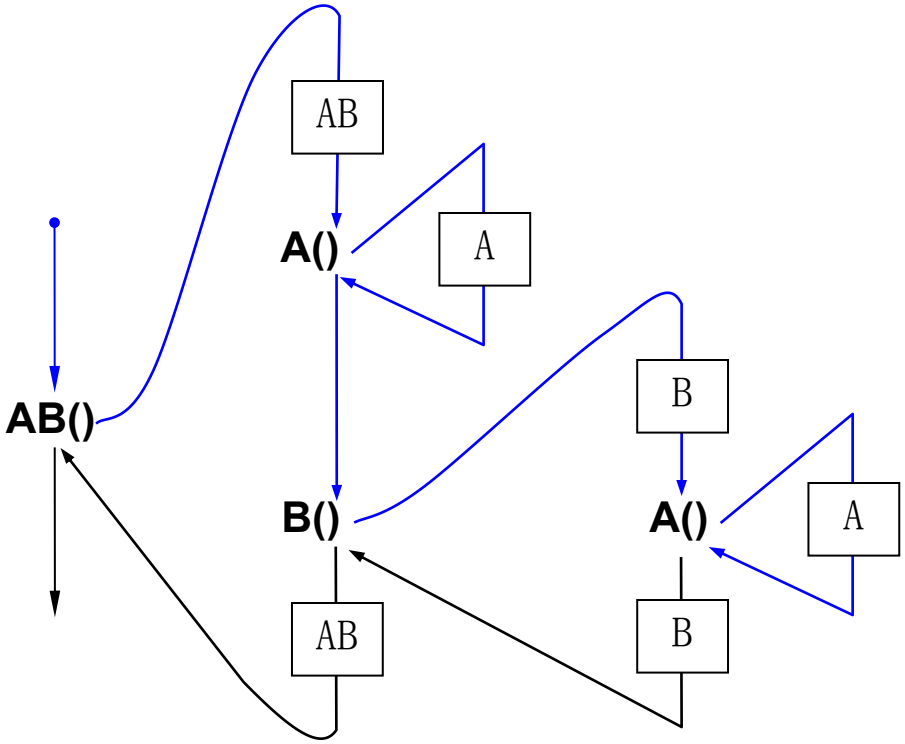
函数AB( )的调用树

执行结果

- AB A B A B AB



(a)AB()执行主线

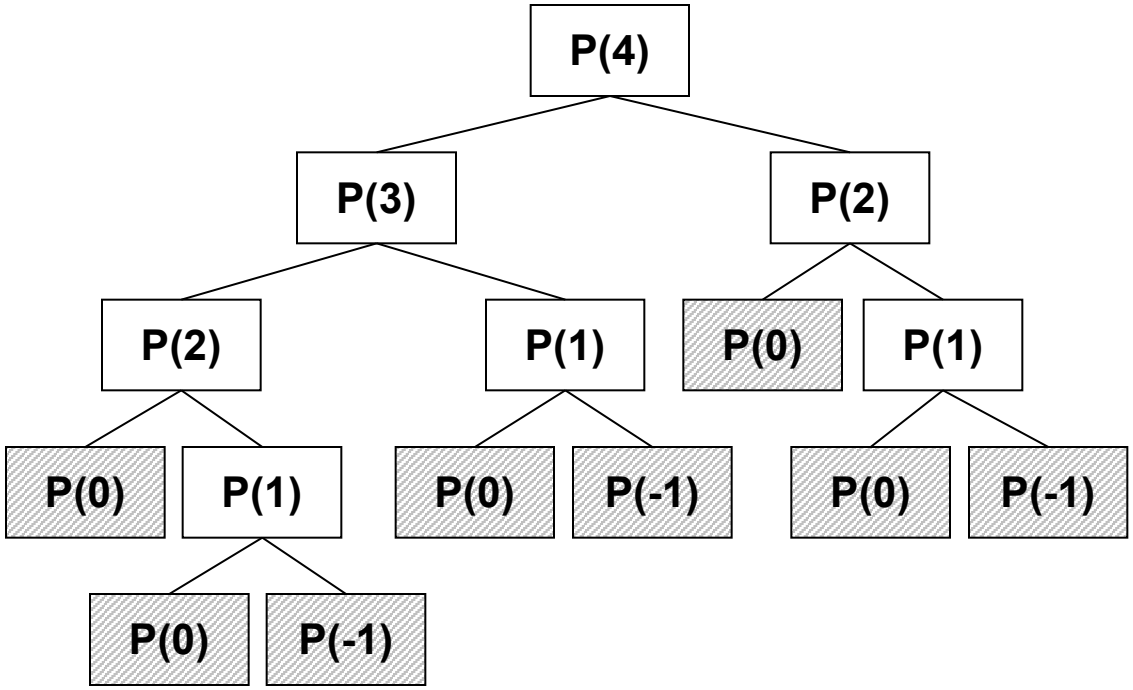


(b)AB()添加其它操作后路线图

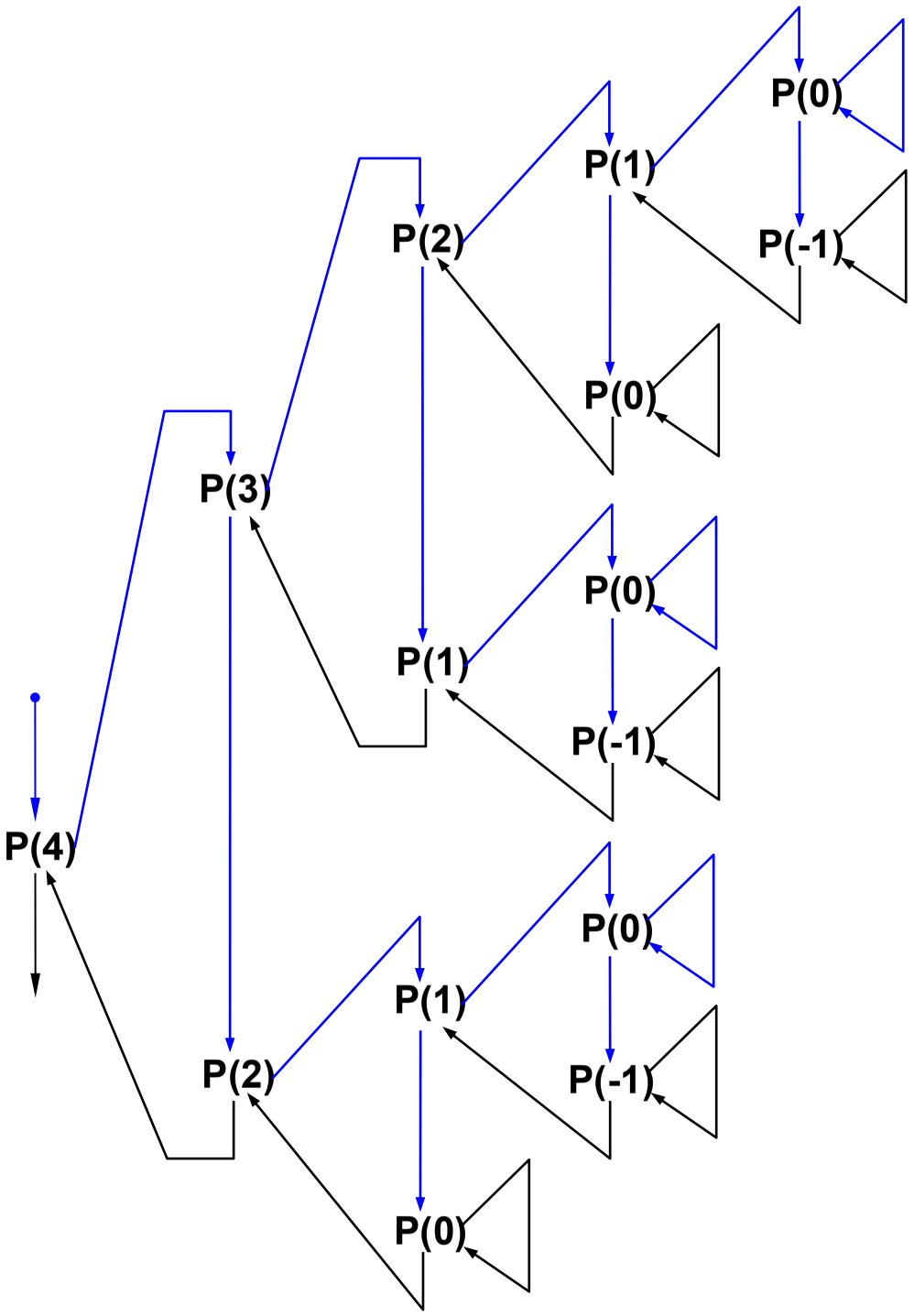
AB( )执行路线图

【例3.9】对下面的程序代码，画出调用P(4)的执行路线图，并给出执行结果。

```
void P( int w )
{
    if( w>0 )
    {
        P( w-1 );
        cout<<w;
        P(w-2);
    }
}
```

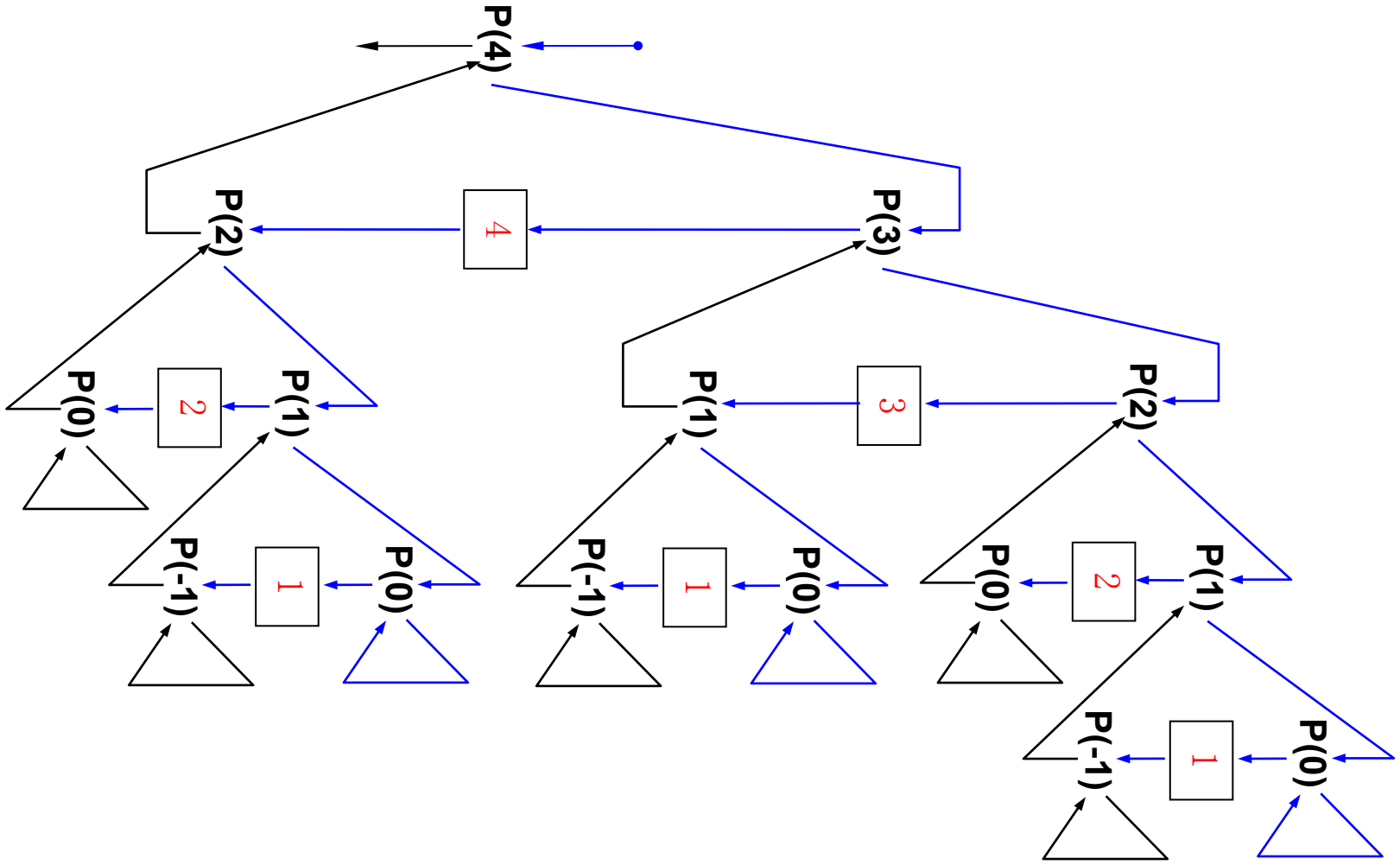


函数P(4)的调用树



P(4)执行主线

- 执行结果为：1 2 3 1 4 1 2。



P(4)执行路线图

- 本题类似二叉树的中序遍历算法

- 类似问题

```
if( w>0 )
{  //类似二叉树先序遍历
        cout<<w;
        P( w-1 );
        P(w-2);
}
```

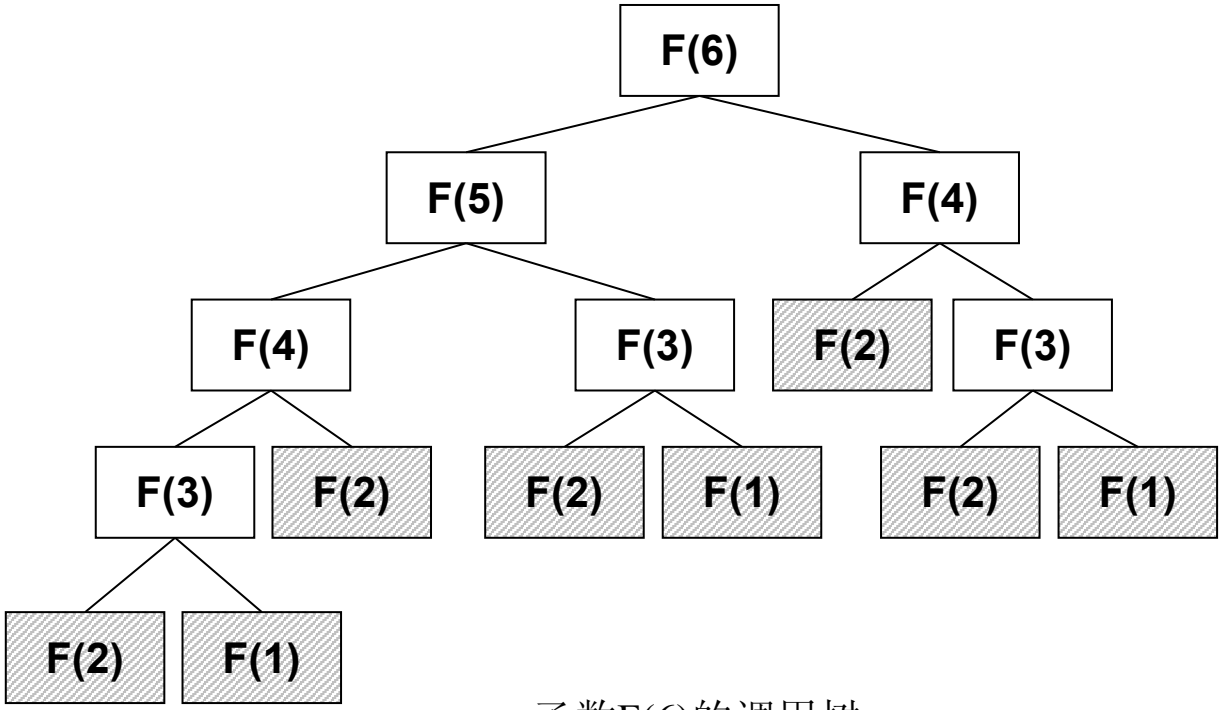
```
if( w>0 )
{  //类似二叉树后序遍历
        P( w-1 );
        P(w-2);
        cout<<w;
}
```

【例3.10】对下面的程序代码，画出调用F(5)的执行路线图，并给出执行结果。

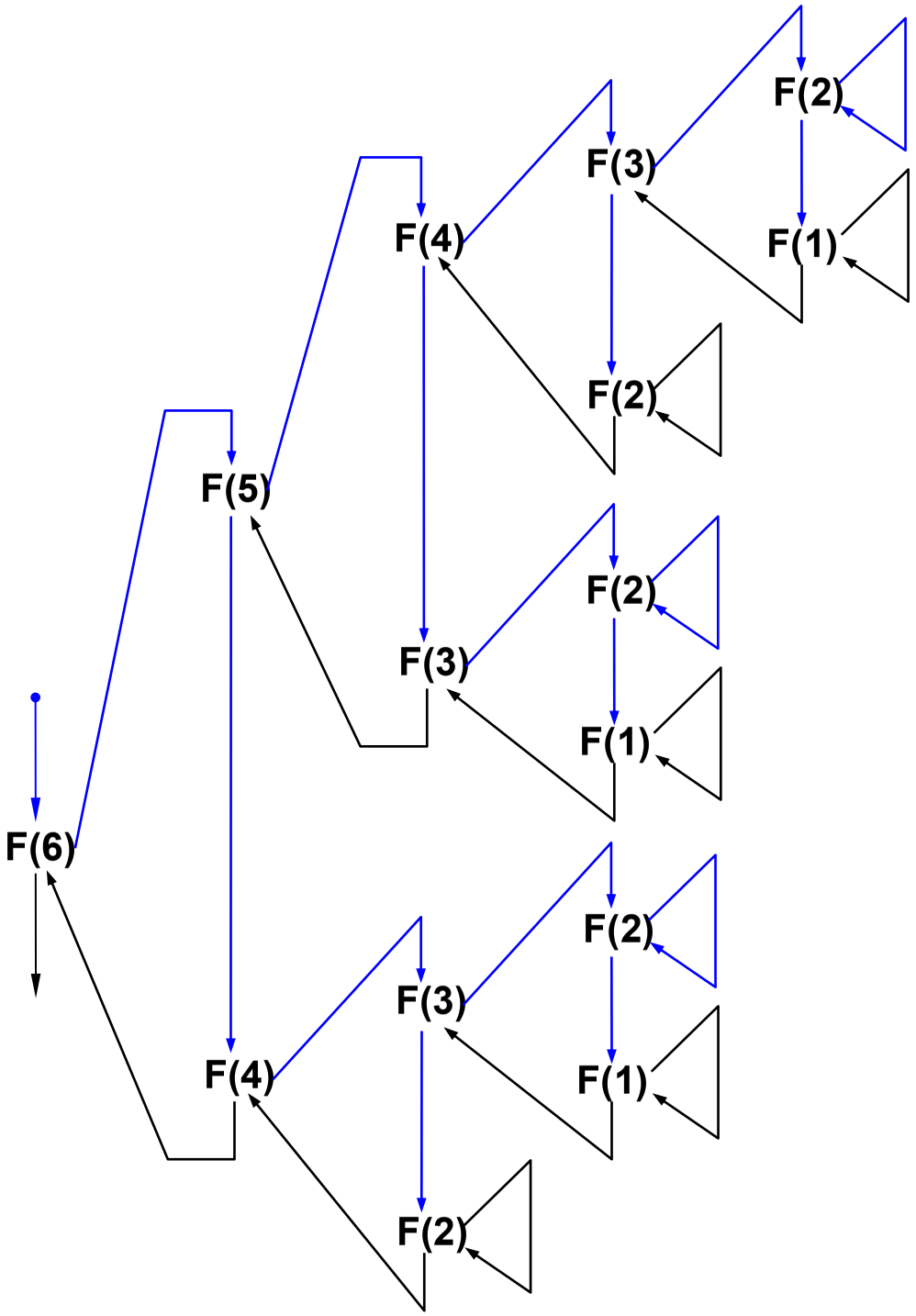
```
int F( int n )
{
    if( n<=2 )
        return 1;
    return 2*F(n-1) + 3*F(n-2);
}
```

【解题分析】

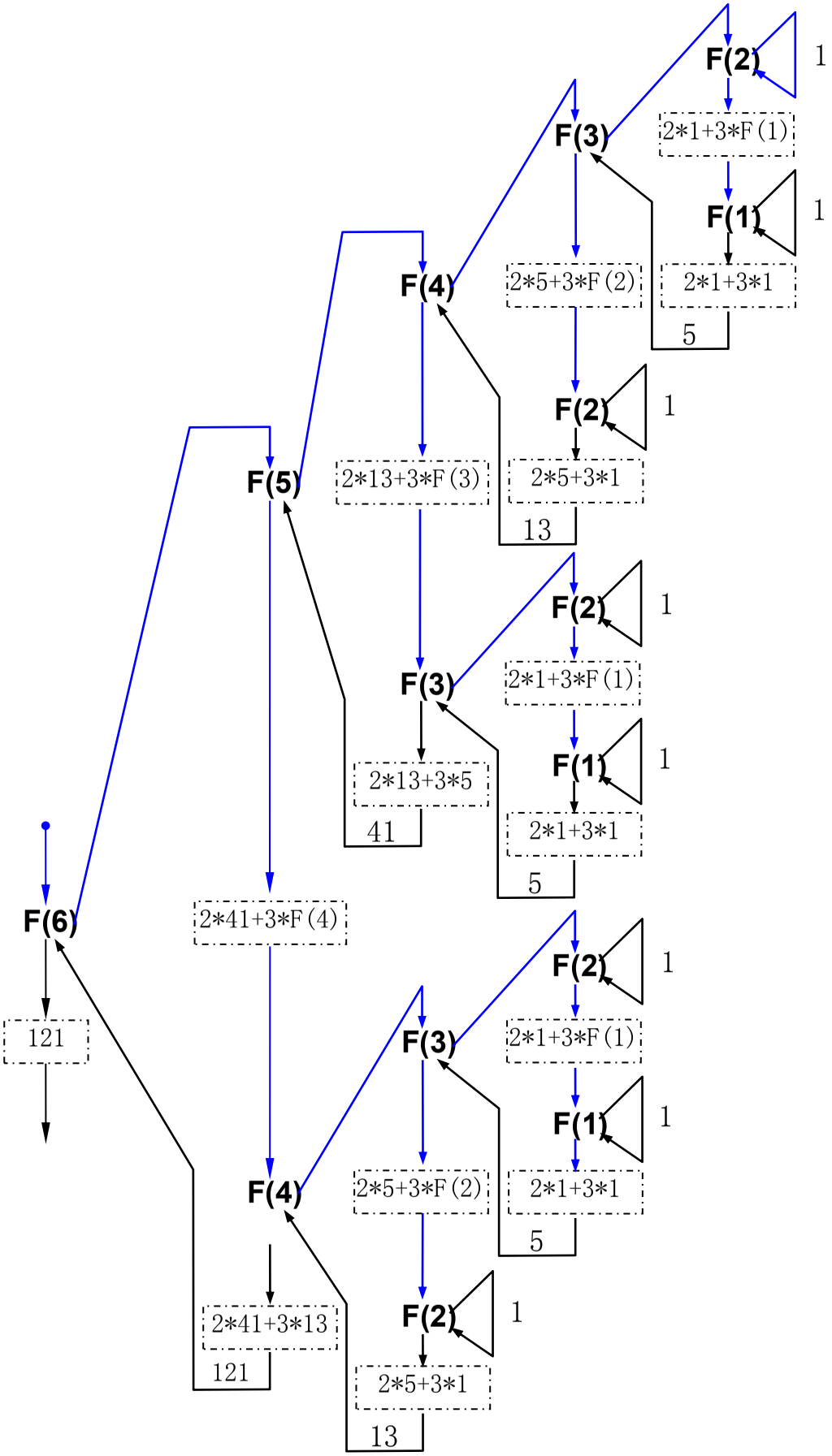
本题有点类似Fibonacci函数，在函数体中有两次递归调用，属于树形递归，且在返回线路上要做计算处理。递归出口条件为n<=2，所以基本问题为F(1)和F(2)，返回值都为1。



函数F(6)的调用树



F(6)执行主线


F(6)
121
2*41+3*F(4)
F(5)
2*13+3*F(3)
2*41+3*13
121
F(4)
2*5+3*F(2)
2*13+3*5
41
2*5+3*1
13
F(3)
2*1+3*F(1)
2*5+3*F(2)
2*1+3*1
5
F(2)
2*1+3*F(1)
2*5+3*1
13
F(1)
2*1+3*1
5
F(2)
1
F(1)
1
F(2)
1
F(1)
1
F(2)
1
F(1)
1
F(2)
1

F(6)执行路线图

3.3.5 递归程序的正确性证明和编写

- 递归程序的阅读理解
- 数学归纳

6.5.2 递归程序的编写

- 不是所有问题都可以用递归方法求解的，只有满足递归三个条件的问题，才可以用递归方法求解。
- 当我们面对一个问题并试图用递归求解时，首先就要对照递归的三个条件进行分析，判定递归求解是否可行。

- 一旦确定用递归求解，我们需要做好下面的工作：

  （1）分析出正向递推分解子问题的规律，并给出形式化的表示。

  - 比如求阶乘的递推公式为：n!=n*(n-1)!，求Fibonacci数的递推公式为:Fib(n)=Fib(n-1)+Fib(n-2)。
  - 对于较复杂的问题这一步可能是相对困难的工作。

（2）分析出递归出口条件和基本问题的解。有的递归只有一个出口，有的递归会有多个出口。

- 比如，求阶乘的出口条件为n=0，只有此一个出口，对应的基本问题为Fact(0)=1；
- 求Fibonacci数，就有两个出口，分别为n=0 和n=1，对应两个基本问题Fib(0)=0和Fib(1)=1。

（3）分析反向回归合成解的规律。有的递归只需简单地返回值；但有些递归不仅要返回值，而且要用返回值进行其它计算和处理；简单的递归甚至不要求返回值。

- 例如求阶乘需要根据Fact(n-1)的返回值做n*Fact(n-1)的计算;
- 求Fibonacci数要根据Fib(n-1)和F(n-2)的返回值，计算Fib(n-1)+Fib(n-2)。

【例3.11】

小猴子第一天摘下若干桃子，当即吃掉一半，又多吃一个，第二天早上又将剩下的桃子吃一半，又多吃一个。以后每天早上吃前一天剩下的一半再加一个。到第10天早上猴子想再吃时发现，只剩下一个桃子了。问第一天猴子共摘多少个桃子？

【分析】

- 假定求解函数名为：int MonkeyPeach(int t)，函数返回值为桃子数，t为天数。
- （1）分析递推公式。这个问题我们可以从第十天的桃子数，倒推求出第九天的桃子数，一直到第一天。假定今天的桃子数为n_2，前一天的桃子数为n_1，而今天的桃子数为昨天吃剩下的，即$n_2=n_1-(n_1/2+1)$，得到$n_1=2*(n_2+1)$。这样第十天为1，第九天为4，第八天为10，第一天为1534。递推公式写成函数形式即为：return 2*(MonkeyPeach (t+1)+1);

- （2）第十天为递归出口条件，对应的基本问题为n_2=1。代码为：**if(t==10) return 1;**
- （3）反向合成原问题解。对这个问题很简单，只要调用**MonkeyPeach (1)**，即求出第一天的桃子数。

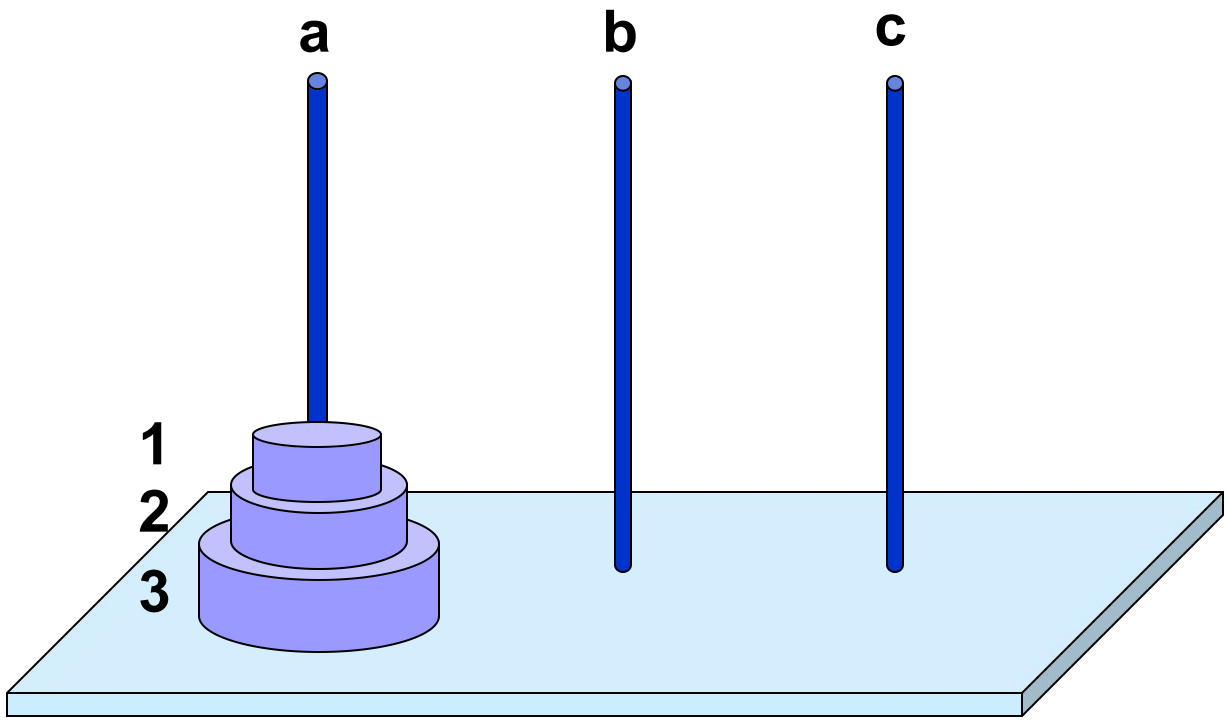
- 也有人把第一天看作10，第十天看作1，这样上面的公式、出口条件、原问题解的表示都要做相应变更。

- 【算法描述】

```
int MonkeyPeach(int t)
{
    if(t==10)
        return 1;
    else
        return 2*(MonkeyPeach(t+1)+1);
}
```

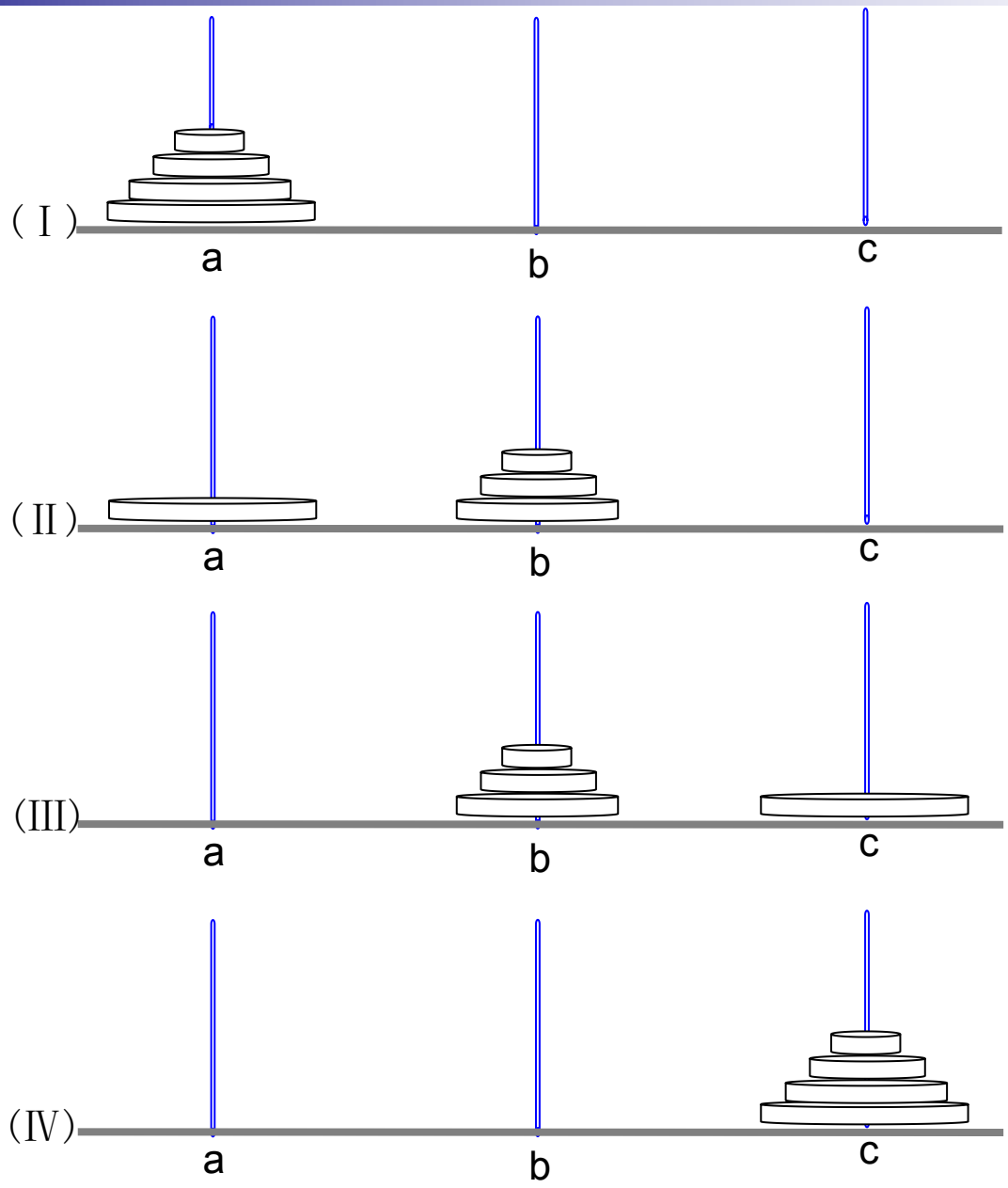
【例3.12】Hanoi塔问题

- 河内塔问题（Problem of Hanoi Tower）。一个底座上有a、b和c三根柱子，a柱子上放置有n片直径不同的圆盘，大盘放底下，小盘放上面。现在要求把所有圆盘全部从a移动到c柱子上。游戏规则：一次只能移动一片圆盘；移动过程中大盘不能压在小盘上面；移动过程可以借助第三根柱子。
- 游戏规则：
  - 一次只能移动一片圆盘;
  - 任何时候大盘不能压在小盘上;


a
b
c
1
2
3

- 【分析】
  - 假设我们用1到n对盘片进行编号，数字越大表示圆盘越大，1号盘最小，n号盘最大。我们以4片盘的移动为例分析问题的解法，如图所示。递归求解分析如下：
  - （1）若想把4片盘全部从a移到c，关键是要把4号盘先移到c柱；若要移动4号盘，必须把压在其上的1、2、3号盘移走，且最好放到b柱子上，如图(Ⅱ)。

- （2）此时，a柱上只有4号盘，c柱为空，4号盘可以从a移到c，如图(Ⅲ)。
- （3）最后，再设法将b柱上的3片盘移动到c柱，这样就实现了全部的移动，如图(Ⅳ)。



四片盘河内塔示意图

- 通过上面的分析可见，移动4片盘的问题可以分解为3个子问题来解决，即2个移动3片盘的子问题，和1个移动1片盘的子问题。这里1片盘的移动是直接可求解的基本问题，不需再分解。同样，3片盘的移动问题可按同样的方法分解为3个子问题解决：2个移动2片盘问题，和1个移动1片盘问题。显然，可以用递归分解，每次分解后问题求解难度降阶。
- 对于n片盘的移动，也是这样，现把a柱上n-1片盘移到b上；n号盘从a移到c上；再把其它n-1片盘从b移动c。n-1片盘采用同样方法分解，如此下去，直到1片盘移动的基本问题。

- 假设函数为void Hanoi(char a, char b, char c, int n)，其中a、b、c为柱子名，n为盘片数，含义为把n片盘从a柱移动到c柱，中间借用b柱；另定义一函数表示1片盘的移动，void Move(a,n,c)，表示将编号为n的1片盘子从a柱移到c柱。根据上面的分析得到:
  - （1）正向递推规律。n片盘移动分解3个子问题。用上面定义的函数表示就是：Hanoi(a,c,b,n-1)，Move(a,n,c)，Hanoi(b,a,c,n-1)。即现把较小的n-1片盘，经c柱，从a柱移动到c柱；n号盘从a柱移到c柱；b上的n-1片盘，经a柱，从b柱移到c柱。

- （2）递归出口条件。n=1，即1片盘子的移动，对应的基本问题Move(a,n,c)。
- （3）反向回归求解原问题。在主函数中调用Hanoi(a,b,c,n)即可，n用实际的盘片数。

【算法描述】

```
void Hanoi(char a, char b, char c, int n)
{
  if(n>=1)
  {
  Hanoi(a,c,b,n-1);
          //将a上的n-1片判移动到b，c为借用盘
  Move(a,n,c);  //最大盘n从a移到c，基本问题
  Hanoi(b,a,c,n-1);
          //将b上的n-1片判移动到c，a为借用盘
  }
}
```

- **Move()函数可以很简单，比如只是一条简单的打印语句：**

```
void Move(char a, int n, char c)
{
   cout<<"移动圆盘 "<<n<<"：  "<<a<<"-->"<<c<<endl;
        //表示移动一个圆盘n，从a到c。
}
```

- Hanoi塔传说
- n片盘需要移动的总次数为2^n-1次。64片盘的移动次数是18,446,744,073,709,551,616次。如果一个人每秒钟移动一片盘，大约需要584,942,417,355.07年，可见一个人有生之年是无法完成的。即使每秒中移动10亿片盘的计算机，也需要大约584.94年才能完成。
- 上面两个问题看上去有点复杂，但用递归编程求解只不过3行代码就解决了，这正是递归的魅力所在。

3.3.6 递归的转换和模拟

- 递归程序转换为非递归程序
- 递归方法的优点和缺点都非常明显，我们在解决实际问题时怎样来扬长避短呢？通常的做法是一开始用递归设计算法，并用递归方法编程实现，然后再寻求把递归程序转换为等价的非递归程序。当然，转换的前提是确保两者的等价性，且程序性能得到提升。
- 还有，除了递归的上述缺点外，有些程序设计语言不支持递归，比如Fortran语言，就必须把递归算法转换为非递归实现。

6.6.1 用循环（迭代）进行转换

- 有些类型的递归程序可以直接用循环（迭代）方法进行等价转换，这种转换会使得程序的时间和空间性能都得到极大提高，因为减少了递归调用中栈帧的建立和释放过程，节省了时间和空间。
- 通常说的非递归比递归好即指这类问题。

1. 尾递归

- 函数体中只有一次递归调用，且调用发生在函数的最后，即函数最后一条语句是递归调用，递归调用返回时，不需进行其它运算，这种递归叫尾递归（tail recursion）。
- 包括有些借助辅助函数和辅助参数可以转换为尾递归的函数。
  - 【例】求阶乘函数，本身不是尾递归函数，因为递归调用返回后，还要计算n*Fact(n-1)，但借助一个辅助参数可以转化为尾递归调用，代码如下：

【尾递归调用求阶乘代码】

```
int FactTail( int n, int result )
{
   if(n==0)
   return result;
   else
   return FactTail( n-1, n*result );
}
```

- 用辅助参数result保存累乘的结果，开始调用时result=1，即FacTail(n,1)。转换为循环方法也是要借助一个辅助变量来保存累乘的结果，代码如下：

【阶乘的循环方法代码】

```
int Fact1(int n)
{
   int result=1;  //保存累乘结果
   int i;
   for(i=n;i>=1;i--)
   result=i*result;
   return result;
}
```

2. 单向递归

- 单向递归指函数体中有两次以上递归调用，但是各递归调用的参数之间没有关系，且递归调用发生在函数的最后，称这种递归叫做单向递归。
- 单向递归借助辅助参数和辅助函数也可以转换为尾递归调用，只是可能要借助多个辅助参数。
- 上面讨论的尾递归是单向递归的一个特例。
  - 【例】求Fibonacci数函数就属于单向递归。可以借助参数转换为尾递归调用，代码如下：

【尾递归调用求Fibonacci数代码】

```
int FibTail(int n, int k, int f1, int f2)
{
   if(n==k)
   return f1;
   else
   return FibTail(n,k+1,f1+f2,f1);  //尾递归
}
```

- 我们引入了3个参数，k初始化为1，每次递归k加1，以实现从Fib(2)往Fib(n)方向累加；f1相当与Fib(n-1)，初始化为1；f2 相当于Fib(n-2)，初始化为0。这个尾递归函数初始调用形式为FibTail(n,1,1,0)。同样我们引入相关参数，可将其转换为循环实现，代码如下：

【循环方法求Fibonacci数代码】

```
int Fib1(int n)
{
   int f1=1,f2=0,tem,i;
   if(n==0)
   return 0;
   else if(n==1)
   return 1;
   else
   {
   for(i=2;i<=n;i++)
   {
   tem=f1;
   f1=f1+f2;  //fib(n-1)
   f2=tem;    //fib(n-2)
   }
   return f1;
   }
}
```

6.6.2 栈模拟进行转换

- 递归的栈模拟就是我们自己定义一个软件栈来模拟递归调用时系统使用函数调用栈建立栈帧和释放栈帧的过程。
- 理论上讲这种方法可以将任何递归函数转换为非递归函数。但是必须明确的是通过这种转换后，程序代码会变得“面目全非”，代码的可读性和可维护性都将很大程度上变差，且容易出错。
- 由于还是使用栈，还是有调用时的入栈操作（类似系统调用的建立栈帧）和返回时的出栈操作（类似系统调用时的释放栈帧），因而转换后程序的时空性能并没得到改善，甚至比系统调用时的性能更差。

- 好处在于每次入栈不要保留寄存器状态和部分程序不需保留返回地址（与系统调用栈比较）；可以开辟更大的软件栈空间，突破系统对函数调用栈的空间限制。
- 所以，使用这种方法转换时，一定要分析是否经过优化后，程序的时空性能的确有改善，否则不建议做这种转换。它的真正作用在于深入研究递归的实现机理。
- 下面仅给出这种转换的步骤：

（1）定义一个软件栈S，初始化为空栈；

（2）在调用函数的入口处设置一个标号地址（不妨设为L0）。系统调用时这一步是系统通过指令指针自动完成的，转为人工调用时，必须有一个地址，明确函数从哪开始执行。

（3）模拟递归的正向递归过程。即模拟系统调用时，建立栈帧的过程，可用以下等价操作替换：

①保留现场：在栈S的栈顶，将函数实参，返回地址，函数本地变量值入栈。

②转入函数入口点执行，即执行 goto L0；

③在函数原来的递归返回处设置标号地址L_i（i=1,2,3,…），即原递归调用的返回地址。有些递归只有一个返回点，有些递归有几个返回点，根据实际情况进行设置。

④根据需要，如果函数有返回值，需要人工定义一个回传变量，模拟函数返回时取出返回值，传送到相应位置。这个过程在系统调用中也是自动完成的。

（4）模拟反向回归过程。即模拟系统调用时，释放栈帧的过程，可以用以下等价操作替换：

如果栈S不空，则依次执行下列操作，否则结束函数执行，返回。

①回传数据：函数值返回的值保存到回传变量中；函数参数回传的，从相应参数取出。系统调用中此步自动完成。

②恢复现场：从栈顶取出返回地址和需要回传的参数。本次调用的相关信息退栈。这个操作相当于系统调用时释放一个栈帧。

③返回，按取出的返回地址，执行goto X，假定取出的返回地址放在X中。

（5）对其它非递归函数的调用和返回可以照搬上述步骤。

- 按照上述5个步骤我们就可以把任何递归程序机械地转换为非递归程序。
- 但上述方法在实现时，还有一些具体问题，比如函数参数、返回值、本地变量可能有不同的数据类型，再加上返回地址类型。如果只定义一个软件栈，那么就要先定义一个包含所有需要数据类型的结构体，将上面的各个数据变为结构的一个分量来处理；还有一种办法是定义几个栈，一个栈处理一种类型的数据。
- 不管那种方法都比系统调用来的复杂，都需要人工去处理和完成。前面已经分析过，这种方法并没太多实用价值。
- 我们后面在学习树和图的遍历算法时，会给出一些通过软件栈将递归遍历函数转换为非递归函数的例子。

【例3.13】栈模拟递归方法求阶乘

【分析】

- 用结构或类来模拟系统调用栈的栈帧，分量分别为：函数参数、函数返回值、本地变量、标记。
- 标记：1—表示已经求出具体的函数值；0—表示本次调用尚未求出具体的函数值。
- 仿照递归的系统调用栈建立和释放过程，即可求出解。
- 本题没有本地变量，结构体（也可以用类定义）定义如下：

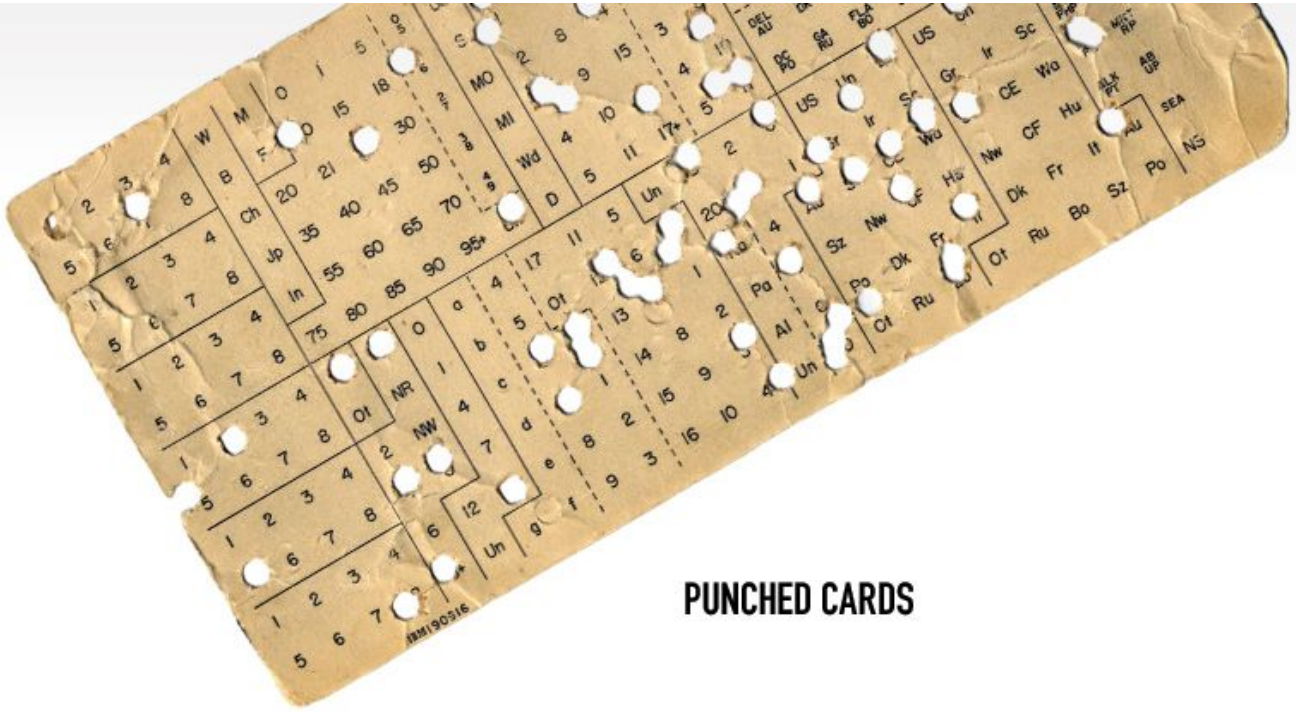
```
typedef struct elemType
{                  //模拟栈帧定义顺序栈元素结构类型
   int n;      //阶乘函数参数
   int fact;  //保存阶乘结果
   int tag;   //标记是否已经求出具体的fact值。
                  //1--已经求出；0--未求出具体的值
}elementType;
```

- 例如，求4!，过程如下：
- 入栈：
  - n=4，fact，tag=0
  - n=3，fact，tag=0
  - n=2，fact，tag=0
  - n=1，fact，tag=0
  - n=0，fact=1，tag=1　//此为原递归出口
- 出栈：
  - 如果栈顶元素的tag==1，说明本次调用结果已经求出，保留相关值，栈顶元素出栈。修改新的栈顶元素。
  - 本题中，n==0，先出栈，取得结果fact=1；此时n==1为新的栈顶，修改新栈顶使：fact=1*fact=1；tag=1。

- 栈顶为n=1，fact=1，tag=1；保留fact=1；出栈；修改新栈顶使：fact=2*fact=2，tag=1。
- 栈顶为n=2，fact=2，tag=1，保留fact=2；出栈；修改新栈顶使：fact=3*fact=3*2=6，tag=1。
- 栈顶为n=3，fact=6，tag=1，保留fact=6；出栈；修改新栈顶使：fact=4*fact=4*6=24，tag=1。
- 当栈中只有1个元素，且tag=1，此时fact即为最终函数值。

【算法描述】

- 见实现代码：Factor.cpp
- 本实现中如果修改栈的一些运算可使代码更简洁一点。



PUNCHED CARDS

6.7 递归的应用

- 递归在程序设计中应用非常广泛，下面只是列举递归应用的一些实例。

【例3.14】设计算法求出从(a_1, a_2, …, a_n)取出k个元素的所有组合。

【分析】

- 需满足n≥k，否则无解。

- 全部组合数为：$C_n^k = \dfrac{n!}{k! \times (n-k)!}$

- 递推分解：从n个数中取一个数后，剩下的问题是从剩下的n-1个数中取k-1个数的组合问题。
  - 递归原型：比如用函数comb(n,k)，则取出一个元素后，变为执行comb(n-1,k-1)。
- 递归终止条件：取满k个数。

算法实现考虑

- 用数组A[n]存放原始元素序列；
- 用数组P[k]存放取得的组合元素；
- 为防止重复取数据，对A[]使用一个指针i（下标），指示当前A[]中取元素的位置，即i之前的元素已经取用过。并用i++替代递归调用中的n-1。
- n，k保持不变，因其它地方用到；
- 用i和num变化控制递归执行。i+1相当原“n-1”；num-1相当原“k-1”

【算法描述】

```
void combination(elementType A[], elementType P[], int n,int
    k,int i, int num)
{
    int j;
    if(num==0)
    {
    cNum++;   //全局变量，记录组合数，获得一个组合，组合数加1.
    print(P,k); //外部函数调用，打印已经取得的组合
    }
    else
    {
    for(j=i;j<n-num+1;j++)   //循环取出num个元素
    {
    P[k-num]=A[j];  //取A中当前元素放至P中
    combination(A, P, n, k, j+1, num-1);   //递归调用
    }
    }
}
```

【程序代码】exam612Combination.cpp

- 初始调用方法：combination(A,P,n,k,0,k);
- 实现中如果把A[]、P[]、n、k都处理为全局变量，函数参数可以减少为2个（i和num）。

- 求解组合还有其它方法。
- 思考书上给出的问题。

【例3.15】背包问题(Knapsack Problem)：设有n个物品，其重量分别是$w_1,w_2,\ldots,w_n$，背包能容纳的重量为S，所有物品的重量之和大于等于S。设计算法从中挑选若干件物品，使其重量之和等于S，正好可以放进背包。

【解】

- 不失一般性，假定物品重量按序号递减排列：$w_1 \geq w_2 \geq \ldots \geq w_n$。
- 用数组w[]保存物品重量。
- 采用试探放置法求解，假设当前正试探第k个物品，重量为w[k]，出现一下3中情况：

① w[k]加入后，背包总重=S，输出结果。

② w[k]加入后，背包总重<S，可在剩下的物品中继续选择试探，这可用递归实现。

③ w[k]加入后，背包总重>S，说明w[k]不能放入，可能原因如下：

(1) 仅w[k]不可，余者皆可以。

(2) w[k]可放，但前面已经放入的物品中有的不合适。

(3) w[k]不可，其它已经放入的物品中也有不合适的。

☞ 暂时不清楚是那种情况，只能继续试探，暂时假设是上述第一种情况。

- 这样试探后，最终会出现以下2中情况：

① 包中物品总重=S，得到一个解决方案；

② 在试探第n个物品时，无论放入与否，均不符合要求，说明前面的放置方案不行，需要尝试其它放置方法。如何实现重新试探呢？

- **重新试探思想:**
  - 从背包中取出最后放置的物品，试探放入余下的物品；
  - 如果仍不可行，继续取出最后放入物品，再试探；
  - 背包的这个操作类似栈。
- **算法实现方法:**
  - 当前w[k]可以放置时，分别递归试探w[k]放与不放两种情况，直到找到解。
- **算法实现考虑**
  - W[]保存物品重量，从W[1]开始，W[0]不用；
  - P[]保存放入背包的物品编号，从P[1]开始，P[0]不用；
  - S背包容量；
  - n物品总数；
  - S1背包当前物品总重；
  - num背包当前放入物品个数，也是数组P[]的下标；
  - k当前试探物品编号，也是W[]的下标。

【算法描述】

```
void Knapsack(int S1,int num, int k)
{
   if(k<=n)
   {
   if(S1+W[k]==S)  //得到一个解，输出
   {
   cNum++;            //解决方案数增1
   P[num+1]=k;        //保存物品编号
   print(P,num+1);  //输出结果
   }
   if(S1+W[k]<S)
   {
   P[num+1]=k;         //保存物品编号
                                    //放入W[k]，继续试探
            Knapsack(S1+W[k],num+1,k+1);
   }
   Knapsack(S1, num, k+1);   //不放入W[k]，继续试探
   }
}
```

【程序代码】exam613Knapsack.cpp

- 初始调用方法：Knapsack(0,0,1);
- 实现中W[]、P[]、n、S皆为全局变量。

- 背包问题还有其它方法。
- 思考书上给出的问题。

- 递归技术的其它应用
  - 树、图遍历算法
  - 快速排序
  - 二分查找
  - 凡是能用循环（迭代）求解的问题都可以转为递归求解，但性能会变差，反之则不一定。
  - …

- 比如：
  - 一个楼梯有n个台阶，每次可以走1个台阶，或2个台阶，问上此楼梯共有多少种走法？
  - 求最大公因子（公约数）。
  - 8皇后问题。
  - 求单链表长度
  - 单链表就地逆置。
  - 输入一个非负整数，将其各个组成数字相加之和。例，n=2012，结果为2+0+1+2=5。
  - 回文判定。
  - 数制转换。
  - …

本章小结

- 递归调用
- 掌握递归技术的重要性
  - 数据结构中许多算法实现非常简单；
  - 甚至有些算法必须要用递归来实现；

【递归调用的缺点】

- 空间效率不高。每次递归调用都要在函数调用栈上建立一个调用栈帧，耗费一些内存空间，递归深度越深，栈帧就越多，占用的内存越多。容易导致“栈溢出”错误；
- 时间效率不高。树形递归调用，即在递归函数体内有多次递归调用时，会出现多次重复调用同样的函数（相同代码、相同参数、相同本地变量），而每次调用函数都有建立栈帧和释放栈帧的过程，花费一定的时间，重复调用造成相同的栈帧被重复的建立和释放，完成同一任务花费重复的时间。

【递归调用的优点】

- 递归调用有完善的数学理论支撑，其正确性可以用数学归纳法加以证明。
- 递归容易编程实现，我们只需按照数学定义就可以写出正确的递归程序，有时甚至还未弄清楚问题的求解机理。
- 递归编程求解问题，代码简短，便于阅读，容易理解。甚至可称之为“优美”。往往几行代码就可以解决很复杂的问题，前面我们已经看到这样的例子，在后面学习树、二叉树和图的遍历算法时还可以充分见证这一点。
- 递归代码健壮性和可维护行好。
- 递归通过让计算机做更多的事情，而让人做更少的事情。

本章小节

- 栈--FILO
- 队列--FIFO
- 链栈
- 链队列
- 递归

本章小节

- 队列是一种特殊的线性表。只能在一端删除（队头），另一端插入（队尾）；
- 队列特性—FIFO
- 顺序队列的存储结构
- 循环顺序队列
  - 判定队空、队满方法
  - 插入、删除的位置计算

【布置作业】

（p96-p97）

- 3.12
- 3.13-2、4、6
- 3.14
- 3.16

考研大纲

- 二、 栈、队列和数组
  - (一) 栈和队列的基本概念
  - (二) 栈和队列的顺序存储结构
  - (三) 栈和队列的链式存储结构
  - (四) 栈和队列的应用
  - (五) 特殊矩阵的压缩存储

考研真题

1. （2009）为解决计算机与打印机之间速度不匹配的问题，通常设置一个打印数据缓冲区，主机将要输出的数据依次写入该缓冲区，而打印机则依次从该缓冲区中取出数据。该缓冲区的逻辑结构应该是

   A. 栈 B. 队列 C. 树 D. 图

2. （2009）设栈S和队列Q的初始状态均为空，元素abcdefg依次进入栈S。若每个元素出栈后立即进入队列Q，且7个元素出队的顺序是bdcfeag，则栈S的容量至少是

   A.1 B.2 C.3 D.4

考研真题

2. （2010）某队列允许在其两端进行入队操作，但仅允许在一端进行出队操作，则不可能得到的顺序是

A. bacde B. dbace C. dbcae D. ecbad

42.（2010）（13 分）设将n(n,1)个整数存放到一维数组R中，试设计一个在时间和空间两方面尽可能有效的算法，将R中保有的序列循环左移P（0＜P＜n）个位置，即将R中的数据由（X0 X1 ……Xn-1）变换为（Xp Xp+1 ……Xn-1 X0 X1 ……Xp-1）要求:

（1）给出算法的基本设计思想。

（2）根据设计思想，采用C或C++或JAVA语言表述算法，关键之处给出注释。

（3）说明你所设计算法的时间复杂度和空间复杂度

考研真题

3. （2011）已知循环队列存储在一维数组A[0..n-1]中，且队列非空时front 和rear 分别指向队头元素和队尾元素。若初始时队列为空，且要求第1个进入队列的元素存储在A[0]处，则初始时front和rear 的值分别是

A. 0, 0 B. 0, n-1 C. n-1, 0 D. n-1, n-1

Thank you !