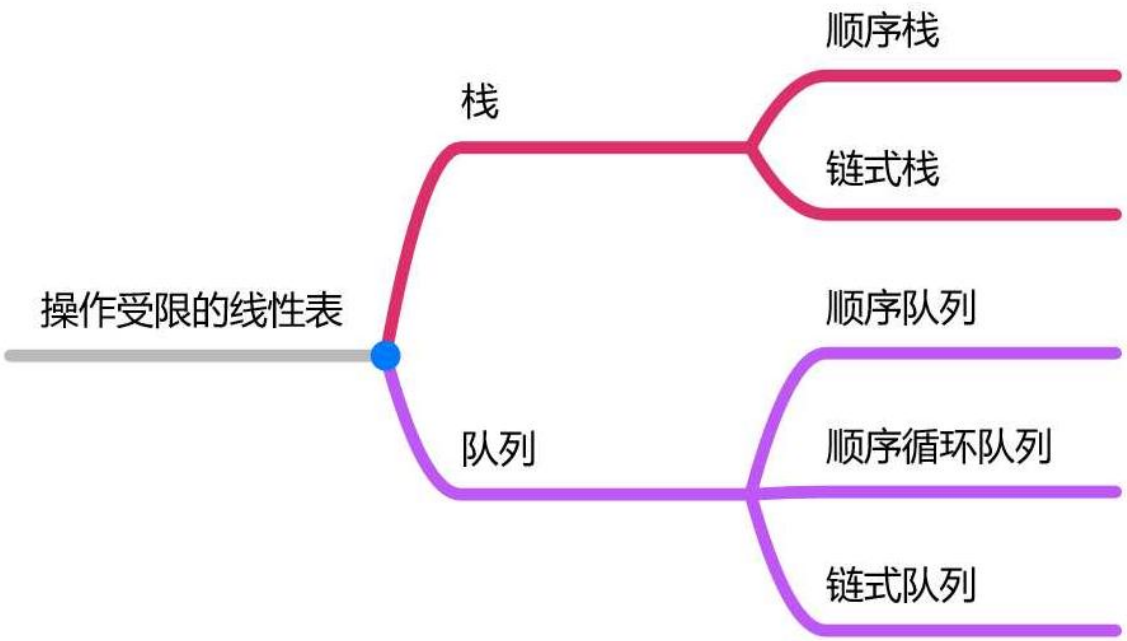
数据结构:栈和队列

Data Structure

主讲教师：杨晓波

E-mail:　248133074@qq.com

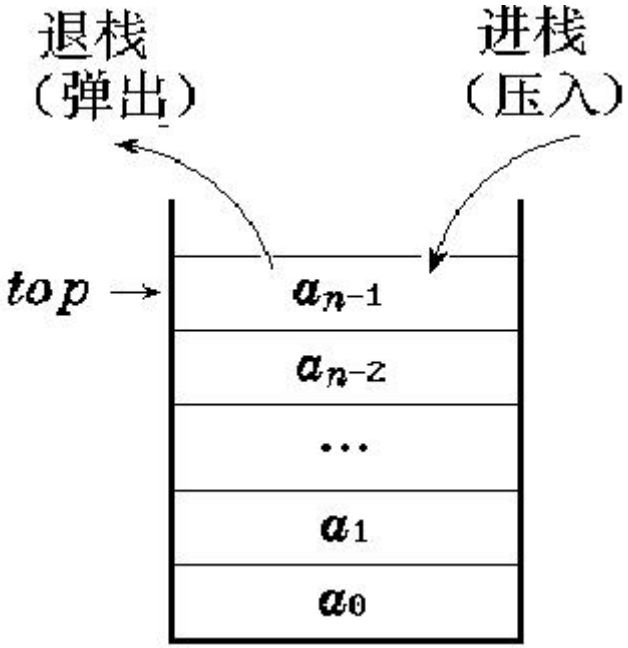
学习要点——栈和队列思维导图



栈(stack)

- 只允许在一端插入和删除的线性表
- 允许插入和删除的一端称为栈顶(***top***)，另一端称为栈底(***bottom***)
- 特点

  后进先出（***LIFO***）
- 主要操作
  - 入栈(push) 、出栈(pop)
  - 取栈顶元素(topValue)
  - 判栈空(isEmpty)



栈的ADT

```
// Stack abtract class
template <typename E> class Stack {
private:
  void operator =(const Stack&) {}     // Protect assignment
  Stack(const Stack&) {}               // Protect copy constructor

public:
  Stack() {}                           // Default constructor
  virtual ~Stack() {}                  // Base destructor

  // Reinitialize the stack.  The user is responsible for
  // reclaiming the storage used by the stack elements.
  virtual void clear() = 0;

  // Push an element onto the top of the stack.
  // it: The element being pushed onto the stack.
  virtual void push(const E& it) = 0;

  // Remove the element at the top of the stack.
  // Return: The element at the top of the stack.
  virtual E pop() = 0;

  // Return: A copy of the top element.
  virtual const E& topValue() const = 0;

  // Return: The number of elements in the stack.
  virtual int length() const = 0;
};
```

Figure 4.17 The stack ADT.

基于数组的栈——顺序栈

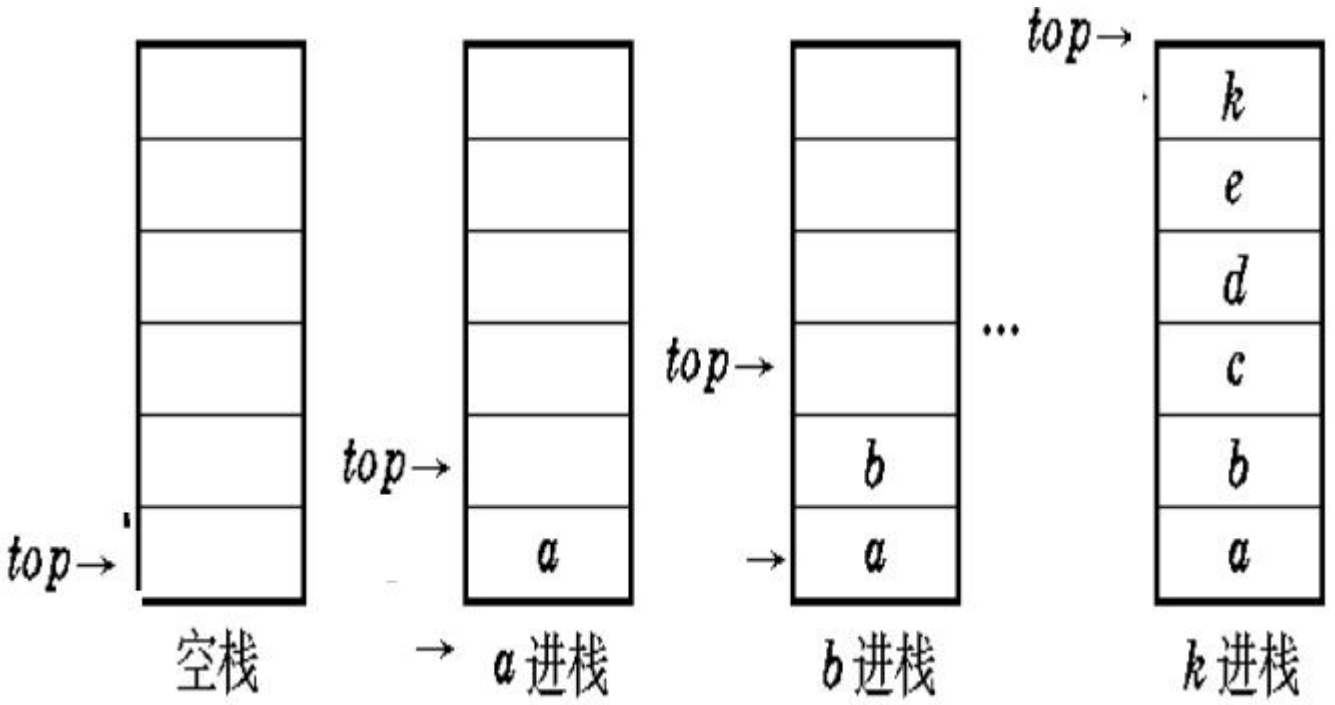
```
template <typename E> class AStack :public Stack<E> {
private:
    int maxsize;
    int top;
    E *listarray;
public:
AStack(int size =DefaultListSize)
{ maxsize =size; top =0; listarray =new E [size]; }
~AStack() {delete [] listarray; }
void clear() {top = 0; }
int length() const {return top;}
```

此初始化选择**top**指示的是下一次压栈元素存储的位置；实际栈顶元素位置为**top-1**

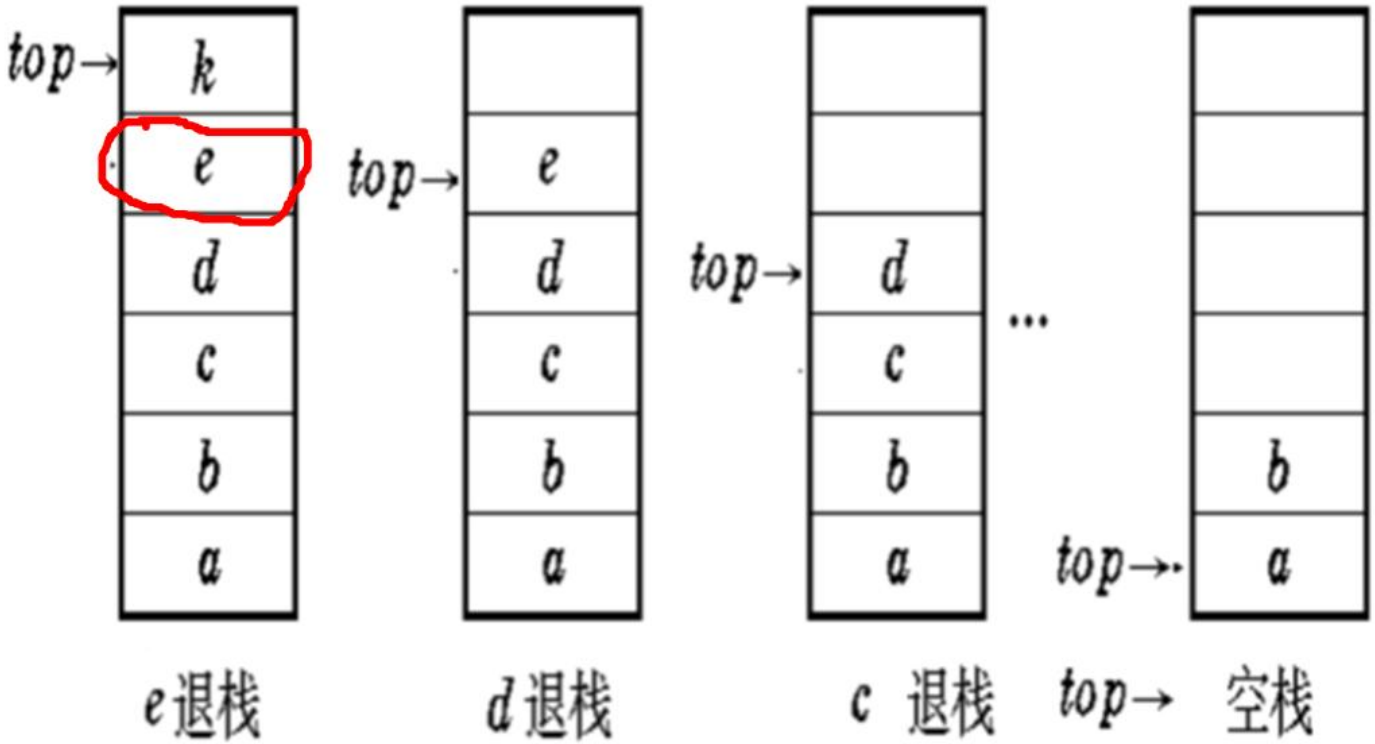
进栈、出栈算法

```
void push(const E& it) //压栈
{ Assert (top!=maxsize, “Stack is full”);//判断是否栈满
   listarray[top++] =it;
}
E pop() {//弹栈
  Assert (top!=0, “Stack is empty”);//判断是否栈空
  return listarray[--top];
}
Const E& topValue() const//取栈顶元素
{Assert (top!=0, “Stack is empty”);
  return listarray[top-1];
 }
```

进栈示例

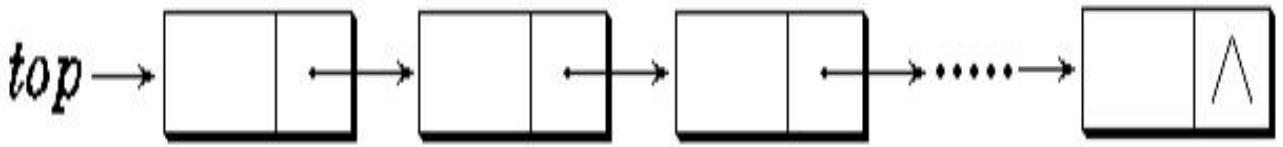


出栈示例



链式栈

```
template < typename E > class LStack :public Stack<E> {
private:
link<E> * top; //栈顶指针
int size;
public:
LStack(int sz =DefaultListSize){top = NULL; size=0;}
~LStack() { clear(); }
void clear(){
  while (top!=NULL)//从头至尾删除每个结点
   {Link<E>*temp=top;top=top->next; delete temp;}
   size=0;
}
```



进栈、出栈算法

```
void push(const E& it) //压栈
{top =new Link<E>(it, top);size++;}

E pop(){//弹栈
  Assert (top!=NULL, “Stack is empty”);
  E it=top->element;  Link<E>* ltemp=top->next;
  delete top; top=ltemp; size--;return it;
}

Const E& topValue() const {
  Assert (top!=NULL, “Stack is empty”);
  return top->element;}

int length() const {return size;}}
```

顺序栈和链式栈的比较

- 操作时间都是常数时间
- 空间开销类似一般线性表
  - 顺序栈
    - 初始化时分配了一个固定长度的空间
    - 当栈不够满时，有空间浪费
  - 链式栈
    - 长度可变，空间按需分配
    - 每个元素的链接域带来结构性开销

共享空间的两个顺序栈

- 当需要实现多个栈时，可利用顺序栈单向延伸的特性，在两个栈之间共享一个数组空间，两个栈从数组的两端向中间延伸，从而减少空间浪费

- 适用于两个栈的空间需求具有相反关系的情况，即一个栈增长时，另一个栈缩短

Figure 4.20 Two stacks implemented within in a single array, both growing toward the middle.

栈的应用——过程调用

- 目标程序的代码放置在代码区
- 静态区、堆区、栈区分别放置不同类型生命期的数据值

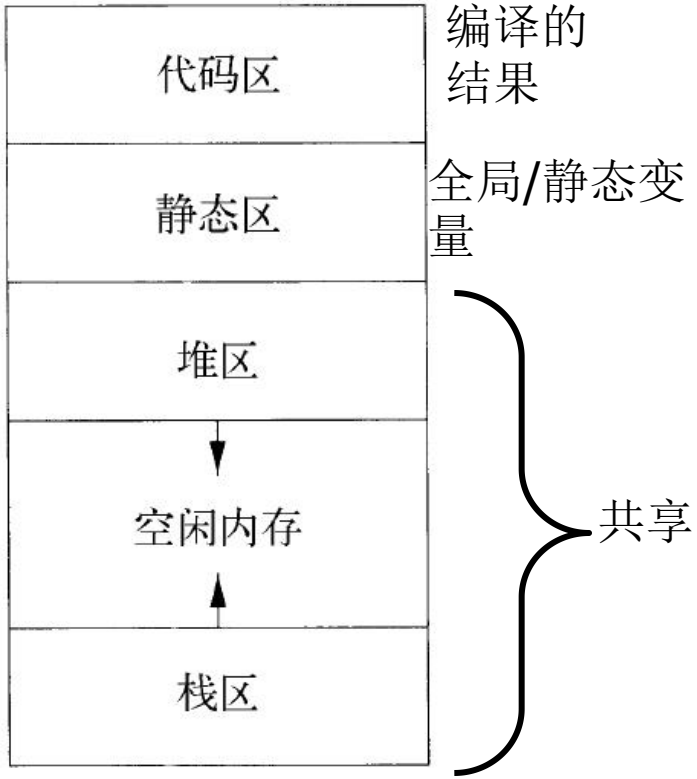


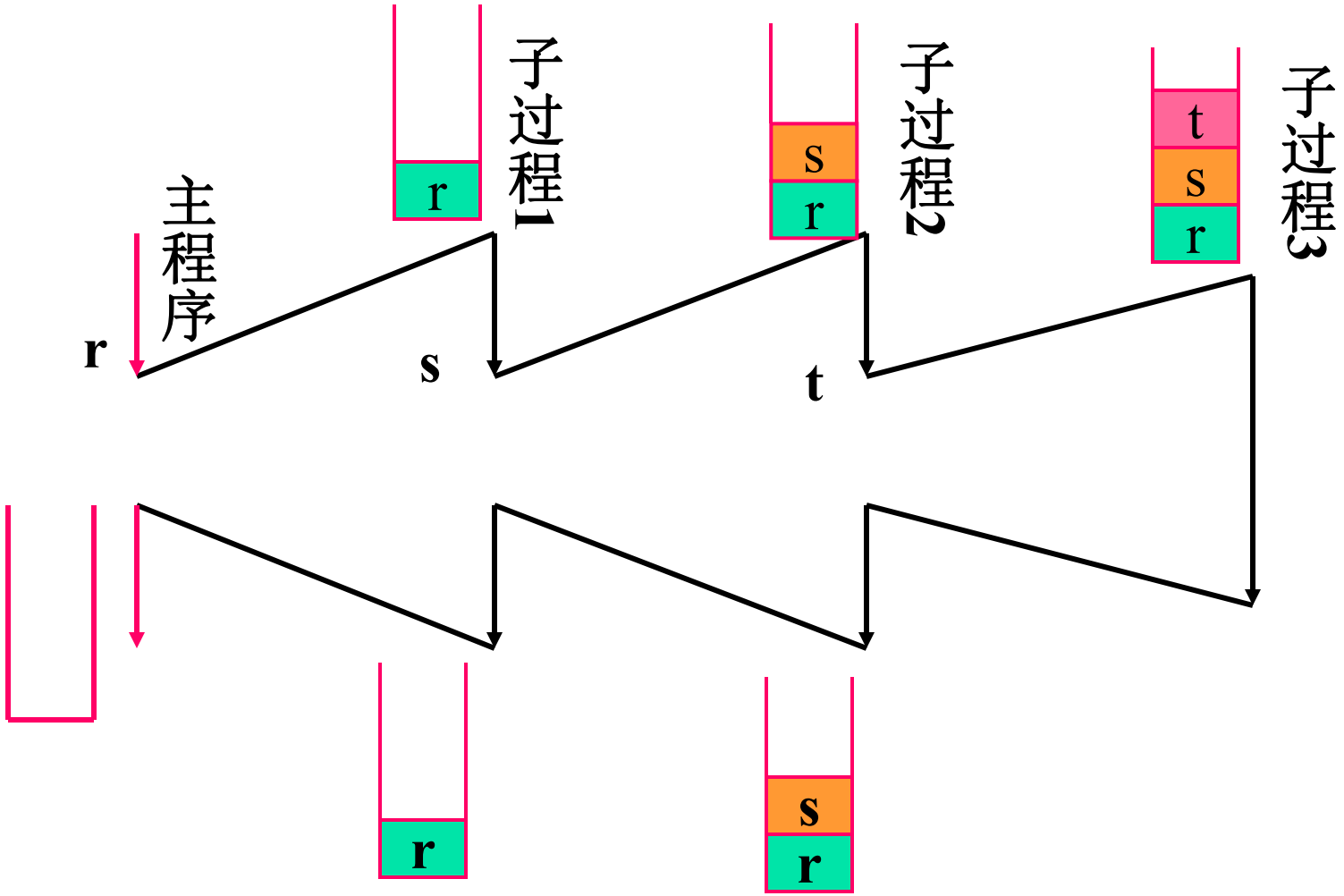
图 典型的运行时环境

活动记录

- 过程调用和返回由控制栈进行管理
- 过程活动记录：当调用过程或函数时，为其局部数据动态分配的存储区
- 活动记录按照活动的开始时间，从栈底到栈顶排列

活动记录框架

堆栈的应用——过程的嵌套调用



实例1——递归过程及其实现

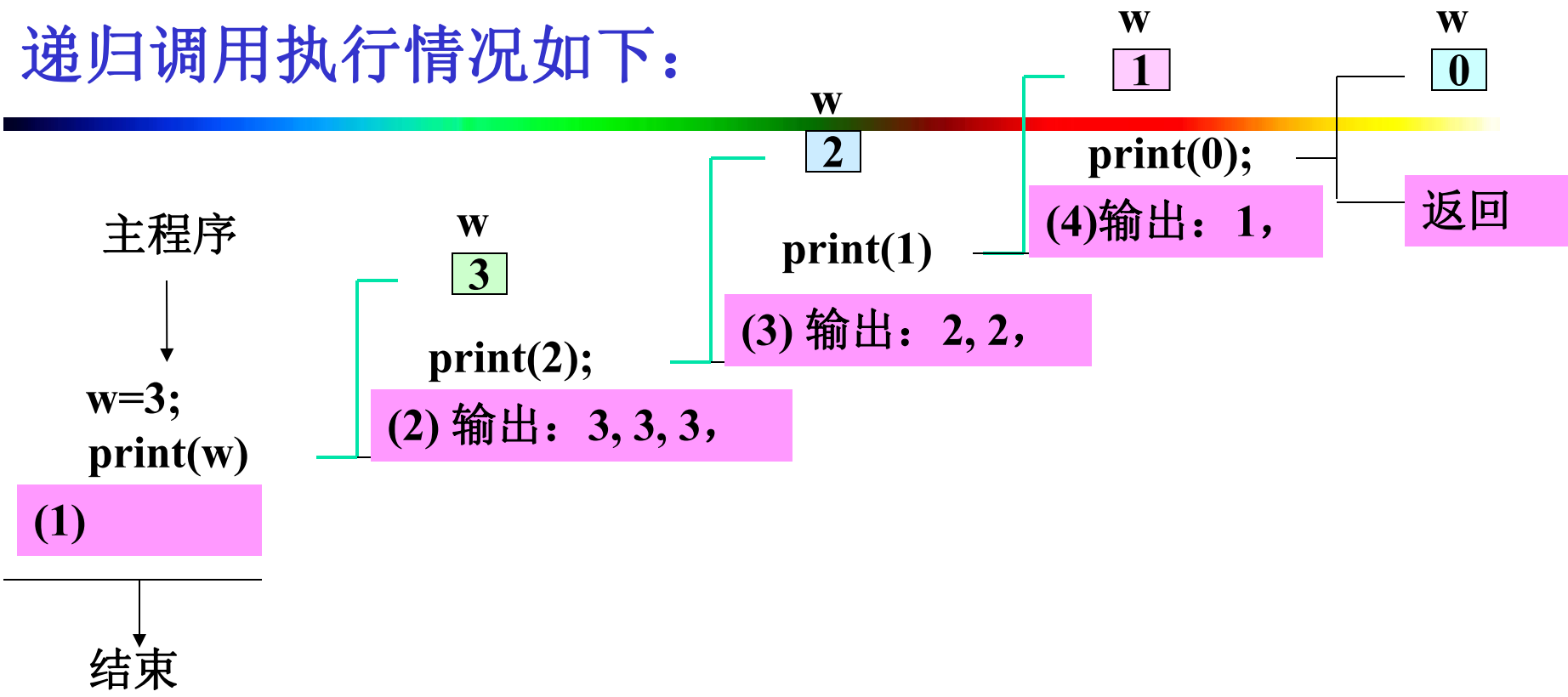
例 递归的执行情况分析

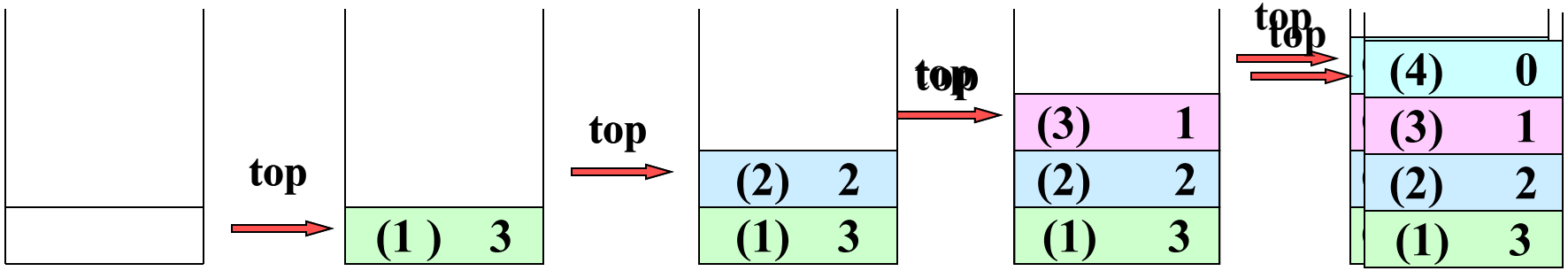
```
void    print(int  w)
{      int  i;
       if ( w!=0)
       {     print(w-1);
             for(i=1;i<=w;++i)
                  printf("%3d,",w);
             printf("/n");
       }
}
```

若调用print（3）

运行结果：
1，
2，2，
3，3，3，

递归调用执行情况如下：


主程序
w=3;
print(w)
(1)
结束
w
3
print(2);
(2) 输出：3, 3, 3，
w
2
print(1)
(3) 输出：2, 2，
w
1
print(0);
(4)输出：1，
w
0
返回


top
(1) 3
top
(2) 2
(1) 3
top
(3) 1
(2) 2
(1) 3
top
top
(4) 0
(3) 1
(2) 2
(1) 3

实例2——Hanoi塔问题

起源（樊塔问题）：印度传说：在贝拿勒斯的圣庙里，一块黄铜板上插着三根宝石针。印度教的主神梵天在创造世界的时候，在其中一根针上从下到上地穿好了由大到小的64片金片。不论白天黑夜，总有一个僧侣在按照下面的法则移动这些金片：一次只移动一片，不管在哪根针上，小片必须在大片上面。僧侣们预言，当所有的金片都从梵天穿好的那根针上移到另外一根针上时，世界就将在一声霹雳中消灭，而梵塔、庙宇和众生也都将同归于尽。

Hanoi塔问题——樊塔问题的抽象

问题描述：

有A,B,C三个塔座，A上套有n个直径不同的圆盘，按直径从小到大叠放，形如宝塔,编号1,2,3.....n。

要求将n个圆盘从A移到C，叠放顺序不变，移动过程中遵循下列原则：

- 每次只能移一个圆盘
- 圆盘可在三个塔座上任意移动
- 任何时刻，每个塔座上不能将大盘压到小盘上

Tower of Hanoi问题

解决方法：（问题分解和递归）

n=1时，直接把圆盘从A移到C。（基础情况，可直接求解）

n>1时，先把上面n-1个圆盘从A移到B,然后将n号盘从A移到C,再将n-1个盘从B移到C。（问题分解和递归）即把求解n个圆盘的Hanoi问题转化为求解n-1个圆盘的Hanoi问题，依次类推，直至转化成只有一个圆盘的Hanoi问题。

Tower of Hanoi算法

```
enum TOHop {DOMOVE,DOTOH};//移动和生成汉诺伊塔
class TOHobj{//汉诺伊塔对象
public:
  TOHop op;
  int num;//盘子总数
  Pole start,goal,tmp;//初始杆、目标杆和中转杆
  TOHobj(int n,Pole s,Pole g,Pole t) {
    op=DOTOH;num=n;
    start=s;goal=g;tmp=t;
  }
  TOPobj(Pole s,Pole g)
    {op=DOMOVE;start=s;goal=g;}
}
```

基于递归的Tower of Hanoi算法

```
void TOH(int n,Pole start,Pole goal,Pole temp)
{  if (n==0) return;
   else {
     TOH(n-1,start,temp,goal);
     move(start,goal);
     TOH(n-1,temp,goal,start);
        }
}
```

课后练习：请编程尝试用基于递归的方法能解决几阶樊塔问题？

基于栈的Tower of Hanoi算法

```
void TOH(int n,Pole start,Pole goal,Pole tmp,Stack<TOHobj*>& S)
{ S.push(new TOHobj(n,start,goal,tmp));
  TOHobj* t;
  while (S.length()>0) { t=S.pop();
    if (t->op==DOMOVE) move(t->start,t->goal);
    else if (t->num>0) {
      int num=t->num;Pole tmp=t->tmp;Pole goal=t->goal;
      Pole start=t->start;
      S.push(new TOHobj(num-1,tmp,start,goal));
      S.push(new TOHobj(start,goal));
      S.push(new TOHobj(num-1,start,tmp,goal)); }
    delete t; }
}
```

把分解的子问题按照处理反顺序入栈

栈的应用举例

例 数制转换

算法基于原理：

$$N = (N \text{ div } d) \times d + N \bmod d$$

数制转换

例如：$(1348)_{10} = (2504)_{8}$，其运算过程如下：

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

```
void conversion () {
    InitStack(S);
    scanf ("%d",N);
    while (N) {
      Push(S, N % 8);
      N = N/8;
    }
    while (!StackEmpty(S)) {
      Pop(S,e);
      printf ( "%d", e );
    }
} // conversion
```

栈的应用举例

例 括号匹配的检验

假设在表达式中

（[] （））或[（[ ] [ ]）]

等为正确的格式，

[（]）或（[（ ））或 （（）]）

均为不正确的格式。

则 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。

分析

例如：考虑下列括号序列：

[([ ] [ ])]

1 2 3 4 5 6 7 8

分析可能出现的不匹配的情况：

- 到来的右括弧并非是所“期待”的；
- 到来的是“不速之客”；
- 直到结束，也没有到来所“期待”的括弧。

算法的设计思想

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，

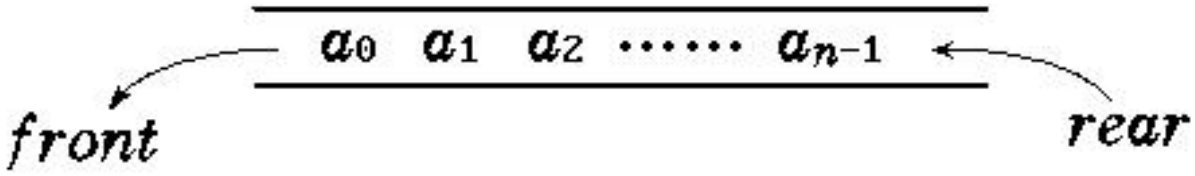
否则表明“左括弧”有余。

```
Status matching(string& exp) {
  int state = 1;//匹配状态标志
  while (i<=Length(exp) && state) {
    switch of exp[i] {
      case 左括弧:{Push(S,exp[i]); i++; break;}
      case”)”: {
        if(NOT StackEmpty(S)&&GetTop(S)=“(“
          {Pop(S,e);  i++;}
        else {state = 0;}
        break;  }   … …
  }
  if (StackEmpty(S)&&state) return OK; ……
```

队列 (Queue)

- 只允许在一端插入，在另一端删除的线性表
- 允许插入一端称为**队尾(rear)**，另一端称为**队首(front)**
- 特点

  先进先出（***FIFO***）
- 主要操作
  - 入队(enqueue) 、出队(dequeue)
  - 取队首元素(frontValue)



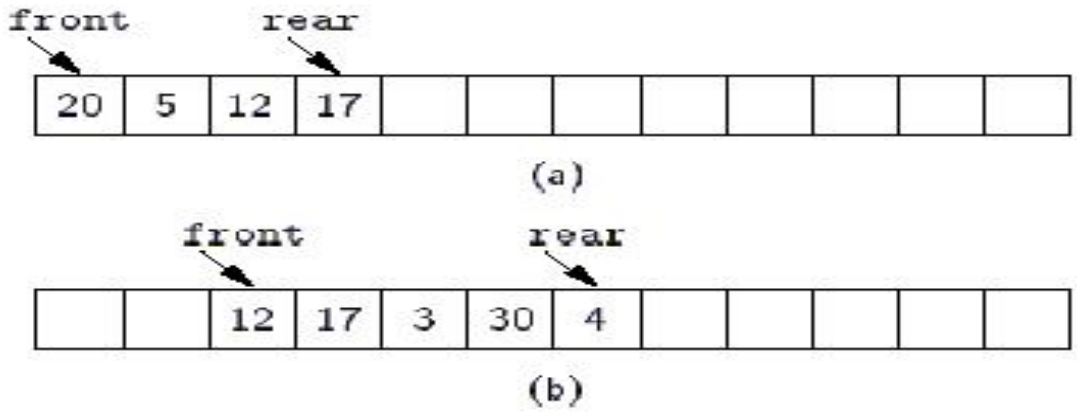
队列的ADT（图4.23）

```
// Abstract queue class
template <typename E> class Queue {
private:
  void operator =(const Queue&) {}     // Protect assignment
  Queue(const Queue&) {}          // Protect copy constructor

public:
  Queue() {}           // Default
  virtual ~Queue() {} // Base destructor

  // Reinitialize the queue.  The user is responsible for
  // reclaiming the storage used by the queue elements.
  virtual void clear() = 0;

  // Place an element at the rear of the queue.
  // it: The element being enqueued.
  virtual void enqueue(const E&) = 0;

  // Remove and return element at the front of the queue.
  // Return: The element at the front of the queue.
  virtual E dequeue() = 0;

  // Return: A copy of the front element.
  virtual const E& frontValue() const = 0;

  // Return: The number of elements in the queue.
  virtual int length() const = 0;
};
```

队列的物理实现

队列的实现——顺序队列 (Queue)

顺序队列



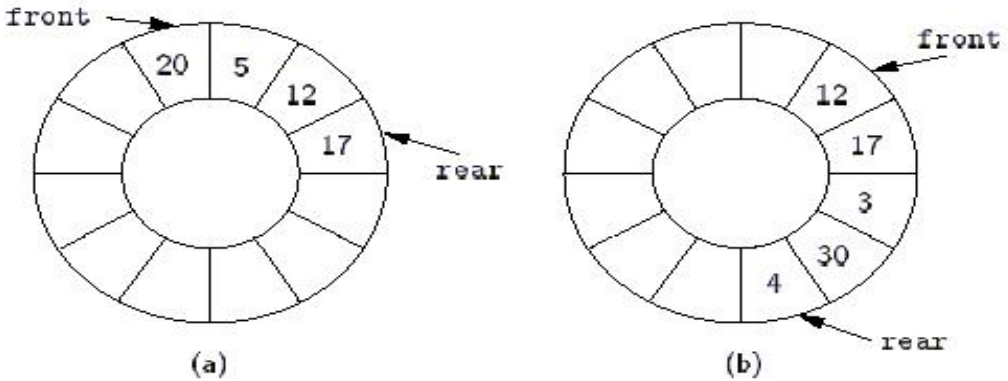
（b)为在（a）基础上
执行3，30，4入队列
和2次出队列操作

顺序队列的问题：

front=rear=n时，队列为空，如何插入元素？

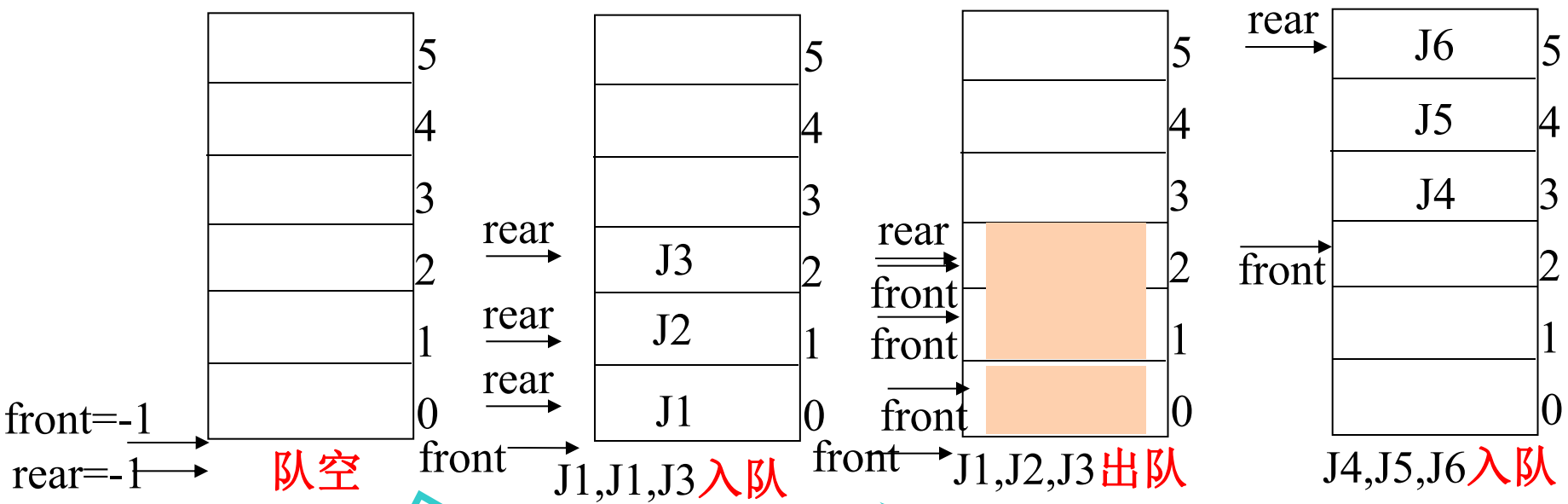
队列的实现——顺序队列 (Queue)

顺序循环队列（常用形式）



首尾相连，有利于提高空间利用率

实现：用一维数组实现sq[M]



设两个指针front,rear,约定：
rear指示队尾元素；
front指示队头元素前一位置
初值front=rear=-1

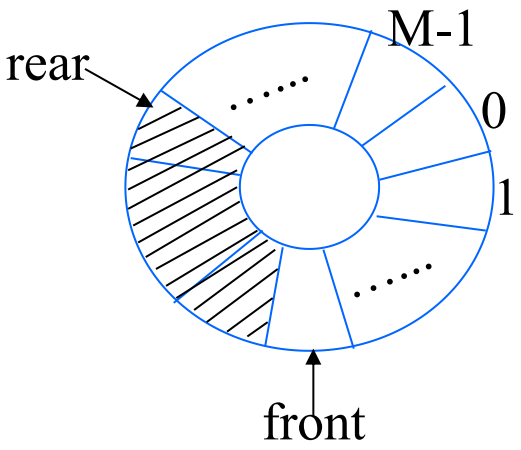
空队列条件：front==rear
入队列：sq[++rear]=x;
出队列：x=sq[++front];

存在问题

设数组维数为M，则：

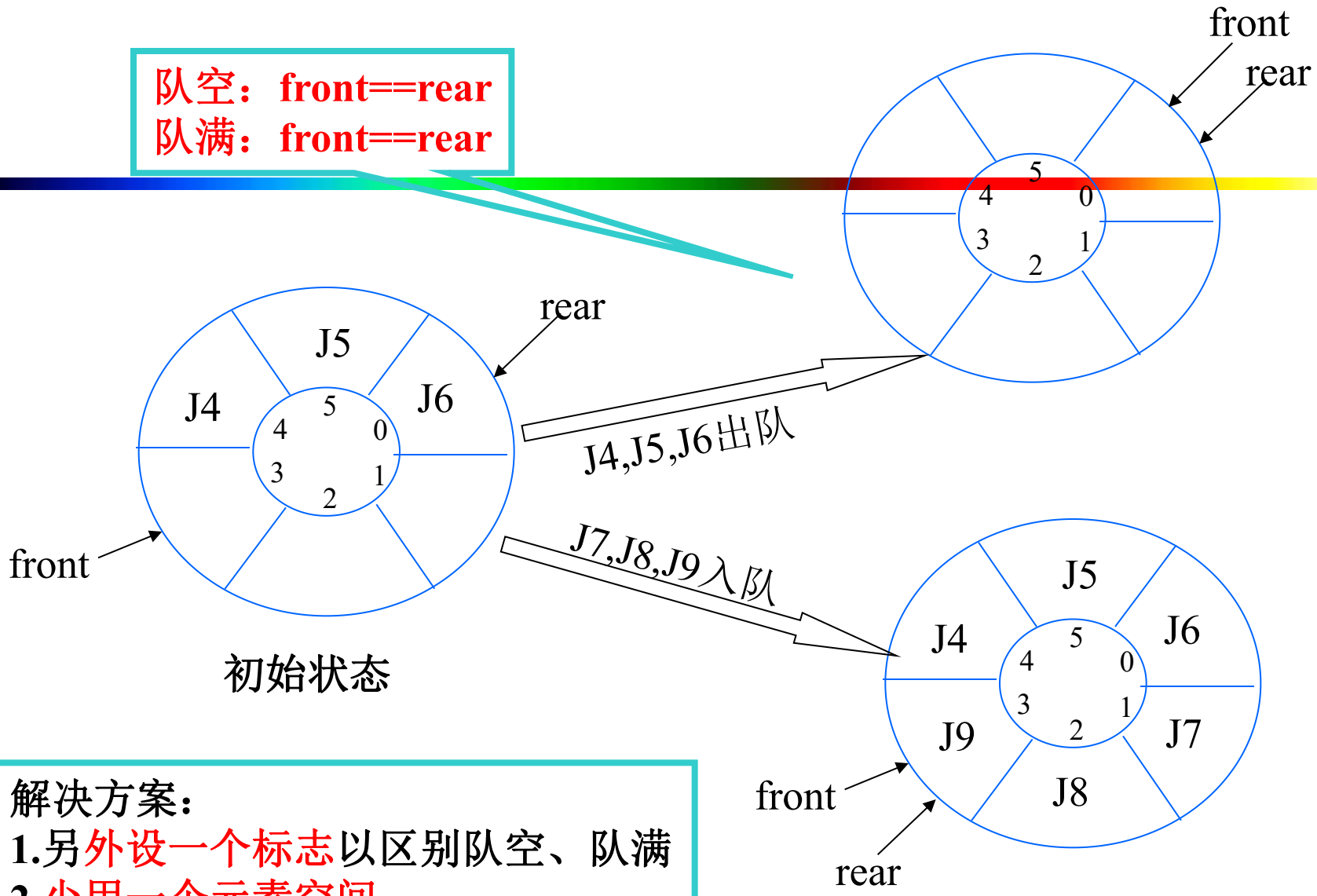
- 当front=-1,rear=M-1时，再有元素入队发生溢出——真溢出
- 当front≠-1,rear=M-1时，再有元素入队发生溢出——假溢出

- 解决方案
  - 队首固定，每次出队剩余元素向下移动——浪费时间
  - 循环队列
    - 基本思想：把队列设想成环形，让sq[0]接在sq[M-1]之后，若rear+1==M,则令rear=0;



- 实现：利用“模”运算
- 入队： rear=(rear+1)%M; sq[rear]=x;
- 出队： front=(front+1)%M; x=sq[front];
- 队满、队空判定条件

队空：front==rear
队满：front==rear



解决方案：
1.另外设一个标志以区别队空、队满
2.少用一个元素空间：
队空：front==rear
队满：(rear+1)%M==front

顺序队列类的实现

```
template <typename E> class Aqueue:public Queue<E> {
private:
  int maxsize;
  int front;
  int rear;
  E *listArray;
public:
  AQueue(int size =DefaultListSize) {
   maxsize = size+1;  front =1; rear = 0;//为区分队列空或满加了一个节点的额外空间
      listArray = new E [maxsize];
  }
  ~AQueue() { delete [] listArray; }
  void clear() {front =1; rear = 0; }
```

顺序队列类的实现

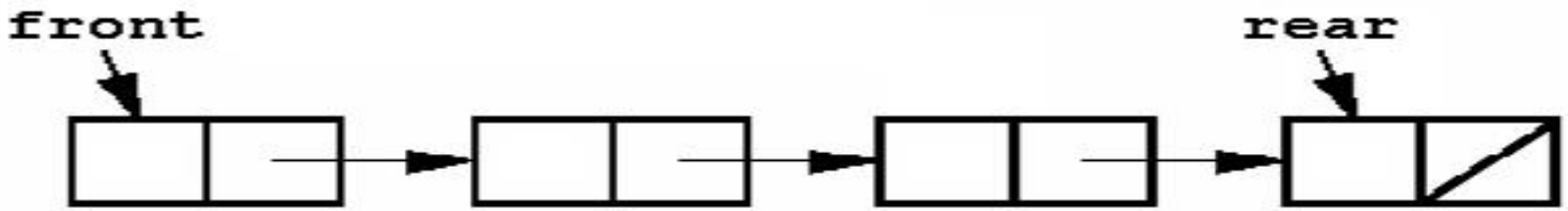
```
void enqueue(const E& it) {
  Assert (((rear+2)%maxsize)!=front, “Queue is full”);
  rear=(rear+1)%maxsize;
  listArray[rear]=it;
}
E dequeue(){
  Assert (length()!=0, “Queue is empty”);
  E it=listArray[front];
  front=(front+1)%maxsize;
  return it;
}
```

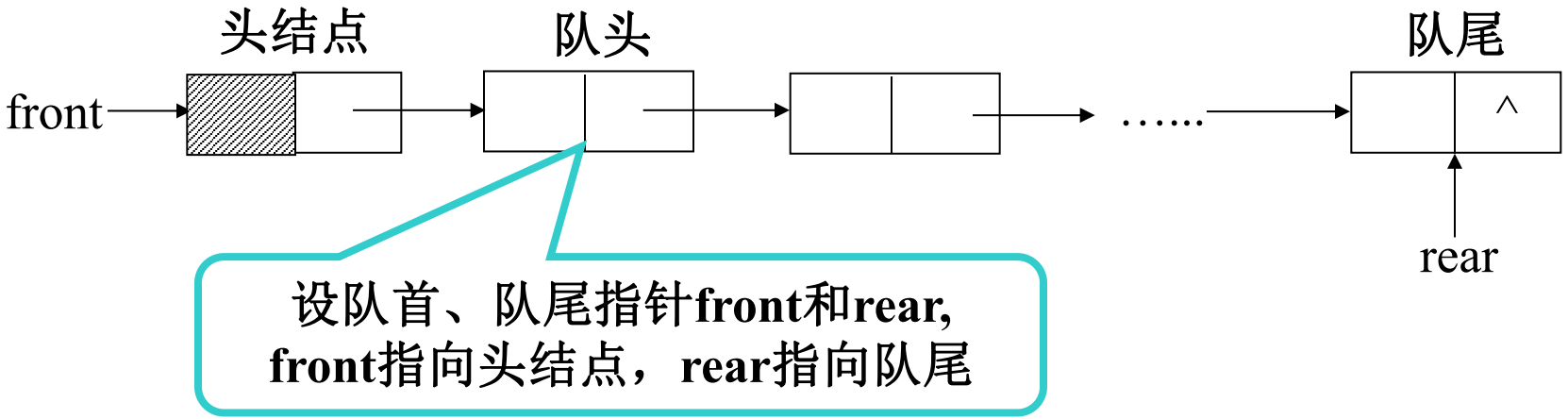
队列：先进先出

顺序队列类的实现

```
  const E& frontValue() const {
    Assert (length()!=0, “Queue is empty”);
     return listArray[front];
   }
 virtual int length() const
 { return ((rear+maxsize)-front+1)%maxsize;}
};
```

队列的实现——链式队列


front
rear


头结点
队头
队尾
front
……
^
rear
设队首、队尾指针front和rear,
front指向头结点，rear指向队尾

链式队列类的实现

```
template <typename E> class LQueue:public Queue<E> {
private:
  Link<E> *front;
  Link<E> *rear;
  int size;
public:
  LQueue(int sz=DefaultListSize)
   { front = rear = new Link<E>(), size=0; }
 ~LQueue() { clear(); delete front;}
```

链式队列类的实现

```
void clear() {    //清空队列
 while (front ->next!= NULL) {
  rear = front;front = front->next;delete rear; }
 rear = front;size=0;
}

void enqueue(const E& it) {  //入队列
 rear->next=new Link<E>(it, NULL);
 rear = rear->next;
  size++;
}
```

链式队列类的实现

```
E dequeue() {   //带头结点的链队列的删除操作
 Assert (size!=0, “Queue is empty”);
 E it=front->next->element;
 Link<E> *ltemp=front->next;//记录删除结点位置
 front ->next= ltemp->next;
 if (rear == ltemp) rear = front;//删除队尾结点
 delete ltemp;
 size--;
 return it;
}
```

链式队列类的实现

```
  const E& frontValue() const {
    Assert (size!=0, “Queue is empty”);
    return front->next->element;
}

 virtual int length() const {return size;}
};
```

队列的应用——识别图元

- 数字化图像是一个m*m的像素矩阵。
- 单色图像中，每个 像素值为0（表示为背景），或为1（表示图元上的一个点），称为图元像素。
- 如果一个像素在另一个像素的左侧、上侧、右侧、下侧，则这两个像素为相邻像素。
- 识别图元就是对图元像素进行标记，当且仅当两个像素属于同一图元时，它们的标号相同。
- 通过逐行扫描像素来识别图元。当遇到一个没有标记的图元像素时，就给它指定一个图元标号（使用数字2, 3, …作为图元编号），该像素就成为一个新图元的种子。通过识别和标记与种子相邻的所有图元像素，可以确定图元中的其他像素。

实例说明

		1				
		1	1			
				1		
			1	1		
	1			1		1
1	1	1				1
1	1	1			1	1

		2				
		2	2			
				3		
			3	3		
	4			3		5
4	4	4				5
4	4	4			5	5

空白代表背景像素，标记为1代表图元像素.
如：（1,3）和（2,3）属于同一图元，（2,3）和（2,4）属于同一图元。因此，（1,3）、（2,3）和（2,4）属于同一图元。属于同一图元的像素被编上相同的标号。

算法说明

- 首先在图像周围包上一圈背景图像（即**0**像素），并对数组**offset**初始化。
- 然后，两个**for** 循环通过扫描图像来寻找下一个图元的种子。种子应是一个无标记的图元像素，有**pixel[r][c]=1**。
- 将**pixel[r][c]**从**1**变成**id**（新的图元编号），即可把图元编号设置为种子的标号。
- 接下来，借助于链表队列的帮助可以识别出该图元中的其余像素。当函数**Label**结束时，所有的图元像素都已经获得了一个标号。

```
void Label()
{//识别图元
    //初始化“围墙”
    for( int i=0; i<=m+1; i++ ){
            pixel[0][i] = pixel[m+1][i]=0;  //底和顶
            pixel[i][0] = pixel[i][m+1]=0;  //左和右
            }

    //初始化offset，相邻像素的行列偏移量
    Position offset[4];
    offset[0].row = 0;  offset[0].col = 1;   //右
    offset[1].row = 1;  offset[1].col = 0;   //下
    offset[2].row = 0;  offset[2].col = -1;  //左
    offset[3].row = -1; offset[3].col = 0;   //上

    int NumOfNbrs = 4;  //一个像素的相邻像素个数
    LinkedQueue<Position> Q;
    int id = 1;  //图元id
    Position here, nbr;

    //扫描所有像素
    for（ int r = 1; r<=m; r++ ）  //图像的第r行
            for ( int c=1; c<=m; c++ )  //图像的第c列
            if（pixel[r][c] == 1 ) {  //新图元
                pixel[r][c] = ++id;  //得到下一个id
                    here.row = r;  here.col = c;
                do{   //寻找其余图元
            for ( int i = 0; i<NumOfNbrs; i++ ){
                     //检查当前像素的所有相邻像素
                    nbr.row = here.row + offset[i].row;
                     nbr.col = here.col + offset[i].col;
                     if( pixel[nbr.row][nbr.col] == 1 ){
                        pixel[nbr.row][nbr.col] = id;
                        Q.Add(nbr);
                     }}
                     //end of if and for
                    //还有未探索的像素吗？
                    if( Q.IsEmpty() ) break;
                    Q.Delete( here ); //一个图元像素
                    }while(true);

            }//结束if和for
    }
```

课堂练习

- 线性结构随堂练习（10分钟）