数据结构:线性表

Data Structure

主讲教师：屈卫兰

Office number：基地203

Email：5604293@qq.com

线性表、栈和队列

- 线性表
- 字典ADT
- 栈
- 队列

4.1 线性表

定义：

线性表L是n个数据元素$a_0,a_1,\ldots a_{n-1}$的有限序列,记作$L=(a_0,a_1,\ldots a_{n-1})$。其中元素个数$n(n\geq 0)$定义为表L的长度。当$n=0$时，L为空表，记作( )。

特性：

在表中,除第一个元素a_0外，其他每一个元素a_i有一个且仅有一个直接前驱a_{i-1}。除最后一个元素a_{n-1}外，其他每一个元素a_i有一个且仅有一个直接后继a_{i+1}。 a_0为第一个元素, 又称为表头元素; a_{n-1}为最后一个元素, 又称为表尾元素。

线性表的抽象数据类型(ADT)

```
template <typename E> class List {
public:
  List() {}
  virtual ~List() {}
  virtual void clear()=0;
  virtual void insert(const E& item)=0;
  virtual void append(const E& item)=0;
  virtual E remove()=0;
  virtual void moveToStart()=0;
  virtual void moveToEnd()=0;
  virtual void prev()=0;
  virtual void next()=0;
  virtual int  length() const=0;
  virtual int  currPos() const=0;
  virtual void moveToPos(int pos)=0;
  virtual bool getValue(Elem&) const=0;
  virtual const E& getValue() const=0;
};
```

4.1.1　顺序表的实现

采用连续的存储单元依次存储线性表中各元素。

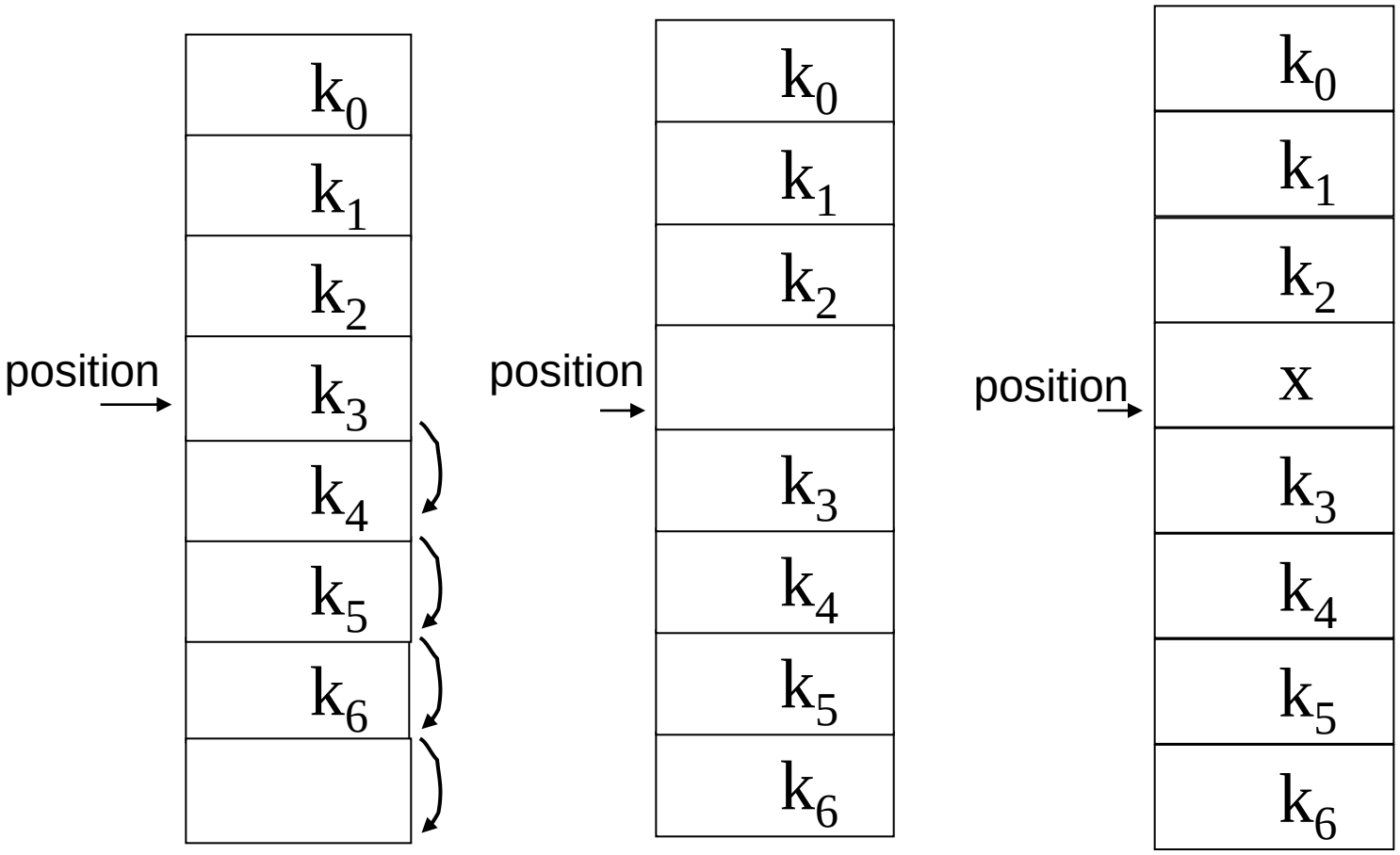
我们称这种存储方式为顺序存储方式，按这种存储方式所得到的线性表叫顺序表。

顺序表具有这样的一个特点：逻辑上相邻的元素在物理上一定相邻。

顺序表类的定义

```
template <typename E>
class AList :public List<E> {
private:
  int maxSize;
  int listSize;
  int curr;
  E* listArray;
public:
…
```

顺序表的插入图示

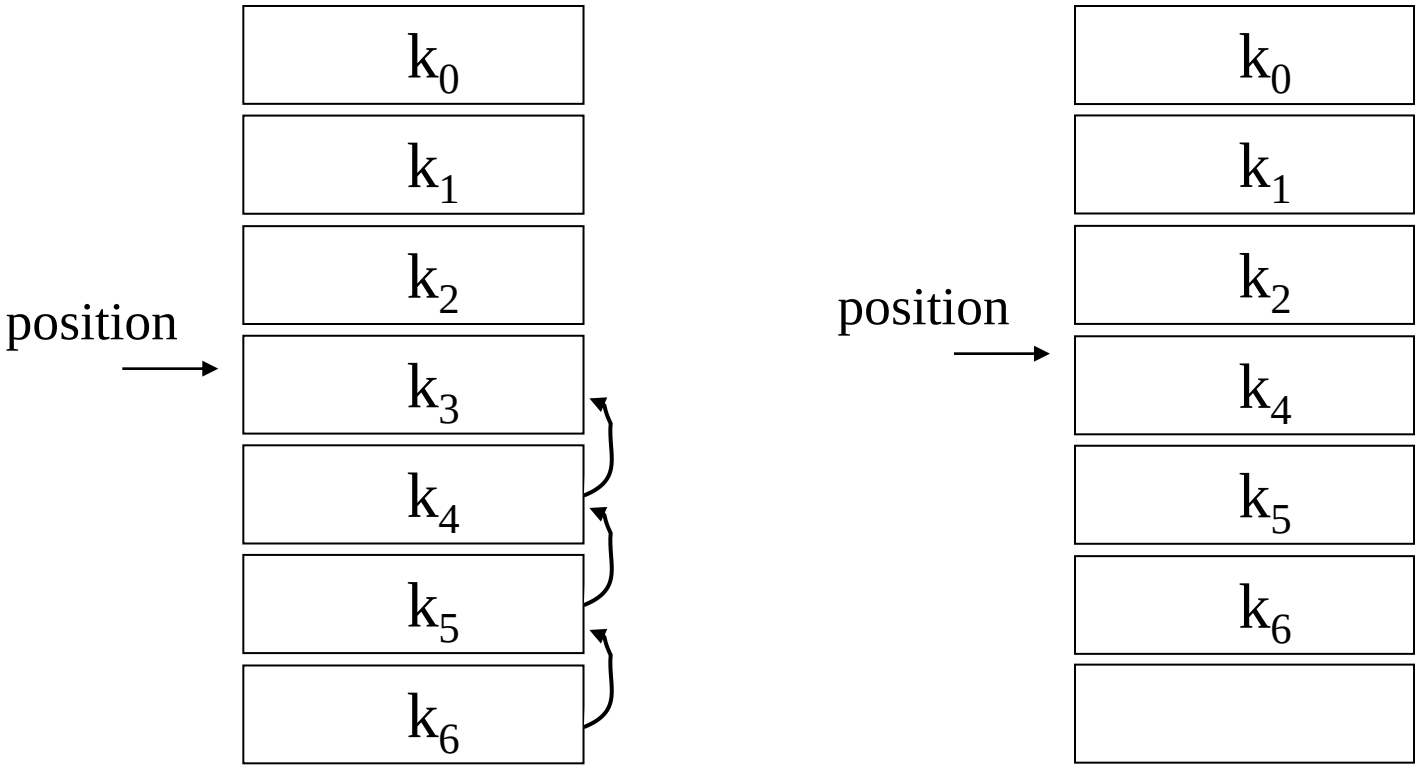

position
k_0
k_1
k_2
k_3
k_4
k_5
k_6
position
k_0
k_1
k_2
k_3
k_4
k_5
k_6
position
k_0
k_1
k_2
x
k_3
k_4
k_5
k_6

顺序表类成员函数的实现

- 顺序表结点插入操作

```
void insert(const E& it)
{  Assert (listSize<maxSize,"List capacity exceeded") ;//边界检查
   for (int i=listSize;i>curr;i--)    //移位
     listArray[i]=listArray[i-1];
   listArray[curr]=it;
   listSize++;
}
```

顺序表的删除图示



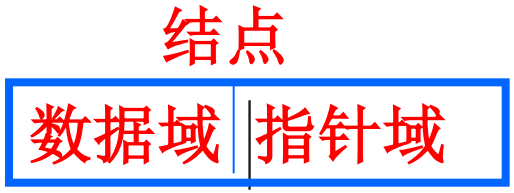
顺序表结点删除操作

```
E remove()
{  Assert ((curr>=0) &&(curr<listSize), “No element”);//边界
   检查
    E it=listArray[curr];
    for (int i=curr;i<listSize-1;i++)    //移位
      listArray[i]=listArray[i+1];
    listSize--;
    return it;
}
```

链表

特点：

- 用一组任意的存储单元存储线性表的数据元素
- 利用指针实现了用不相邻的存储单元存放逻辑上相邻的元素
- 每个数据元素ai，除存储本身信息外，还需存储其直接后继的信息
- 结点
  - 数据域：元素本身信息
  - 指针域：指示直接后继的存储位置



链表

```
template <typename E>
class Link {
public:
  E element; // value for this node
  Link *next; // Pointer to next node in list
  Link(const E& elemval, Link* nextval=NULL)
   { element=elemval; next=nextval; }
  Link(link* nextval=NULL) { next=nextval; }
}
```

单链表类

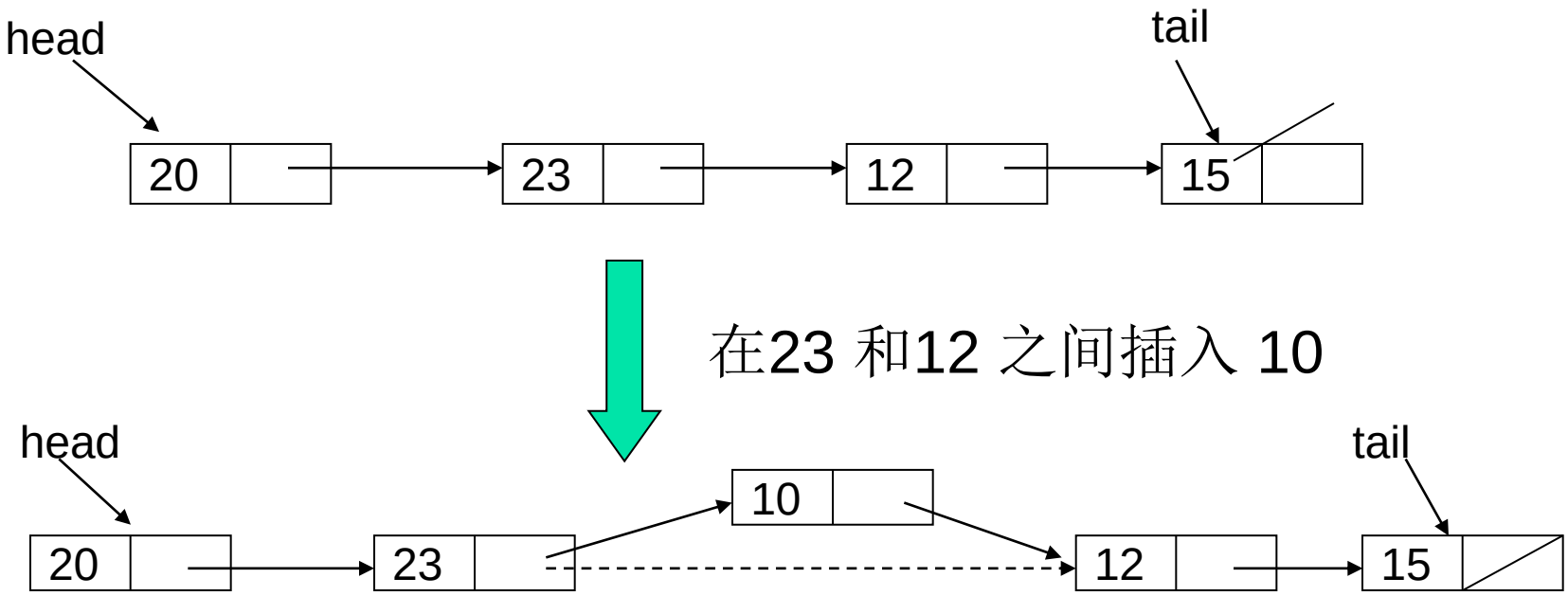
```
template <typename E>
class LList :public List<E> {
private:
  Link<E>* head;
  Link<E>* tail;
  Link<E>* curr;
  int cnt;
```

```
void init(){
  curr=tail=head=new
 Link<E>;
   cnt=0;
 }
 void removeall(){
  while (head!=NULL) {
    curr=head;
    head=head->next;
    delete curr;
   }
 }
```

单链表类

```
public:
  LLlist(int size=DefaultListSize){init();}
  ~LList(){removeall();}
  void clear(){removeall();init();}
  bool insert(const E& it);
  bool append(const E& it);
  E remove();
  void moveToStart() {curr=head;}
  void moveToEnd()  {curr=tail;}
  void prev();
  void next();
  int  length() const {return cnt;}
  int  currPos() const;
  void moveToPos(int pos);
  const E& getValue() const;
  }
```

单链表的插入



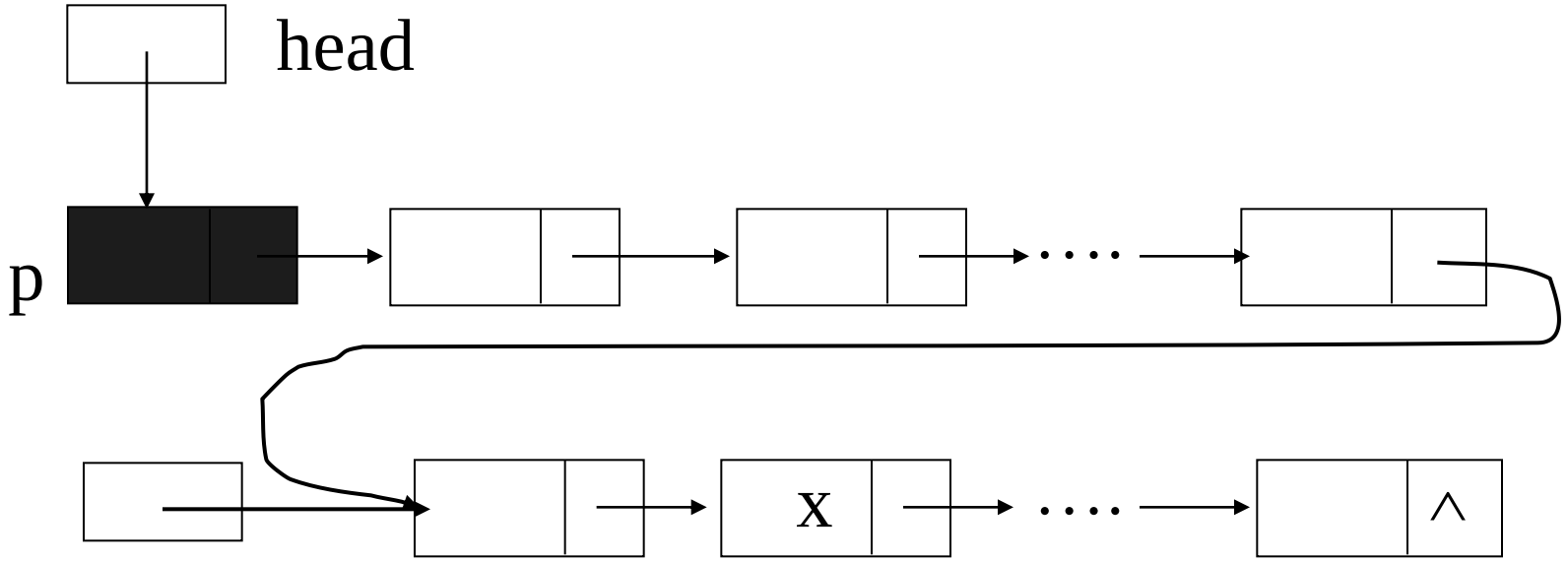
1. 创建新结点
2. 新结点指向右边的结点
3. 左边结点指向新结点

单链表的结点插入

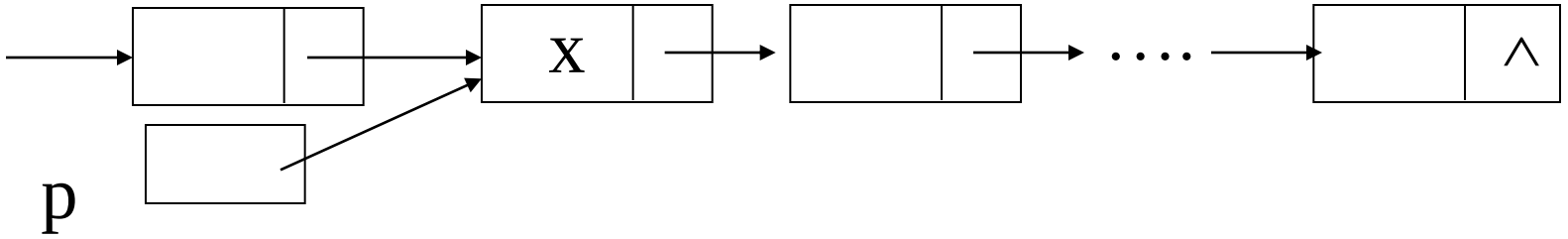
```
void insert(const E& it) {
  curr->next=new link<E>(it, curr->next);
  if (tail == curr) tail =curr->next;
  cnt++;
 }
```

- 创建新的结点并且赋给新值。
  - new link<E>(it, curr->next);
- 当前结点元素前驱的next 域要指向新插入的结点。
  - curr->next=new link<E>(it, curr->next);

单链表删除示意



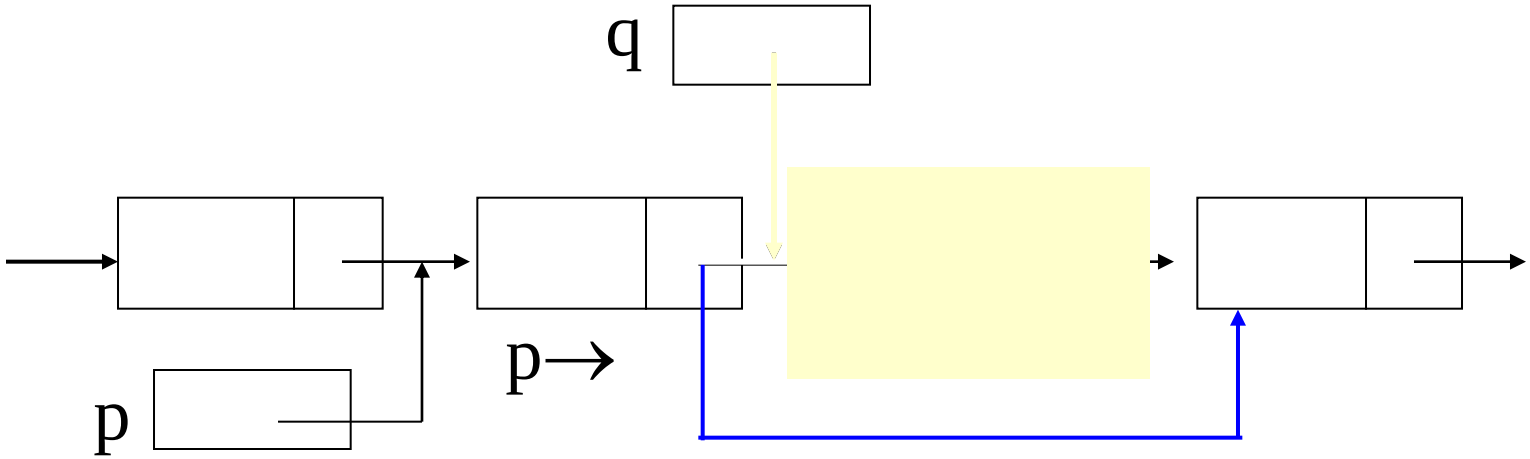
用p指向元素x的结点，可以吗?



单链表结点的删除

```
E remove() {
  Assert(curr->next!=NULL,”No element”);
  E it =curr->next->element;
  link<E>* ltemp =curr->next;
  if (tail == ltemp) tail =curr;
  curr->next =curr->next->next;
  delete ltemp;
  cnt--;
  return it;
}
```

删除值为 x 的结点



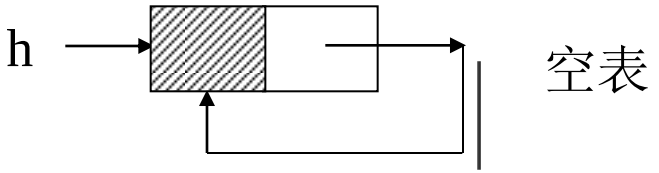
q = p→next;

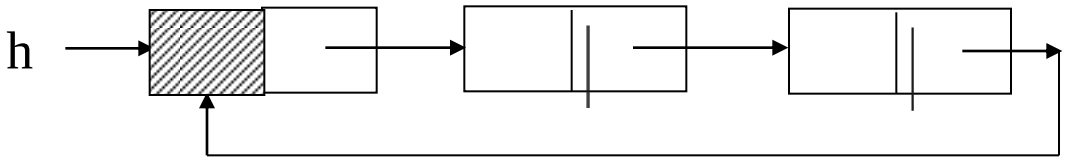
p→ next = q→ next;

free(q);

循环链表

- 循环链表是表中最后一个结点的指针指向头结点，使链表构成环状。
- 特点：从表中任一结点出发均可找到表中其他结点，提高查找效率。
- 操作与单链表基本一致,循环条件不同
  - 单链表p：p->link=NULL
  - 循环链表p：p->link=H





双链表

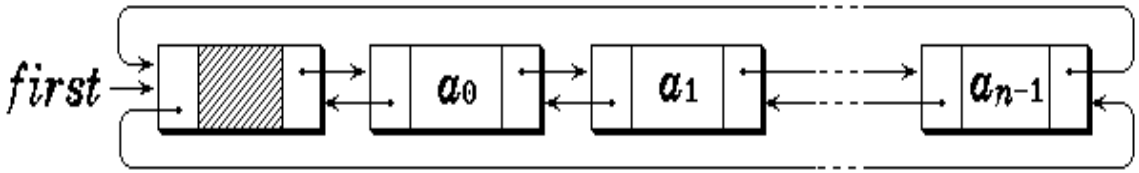
- 双向链表是指在前驱和后继方向都能遍历的线性链表。
- 双向链表每个结点结构：

前驱方向 ← → 后继方向

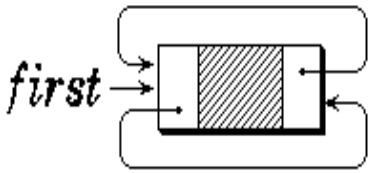
- 双向链表通常采用带表头结点的循环链表形式。

双链表

非空表

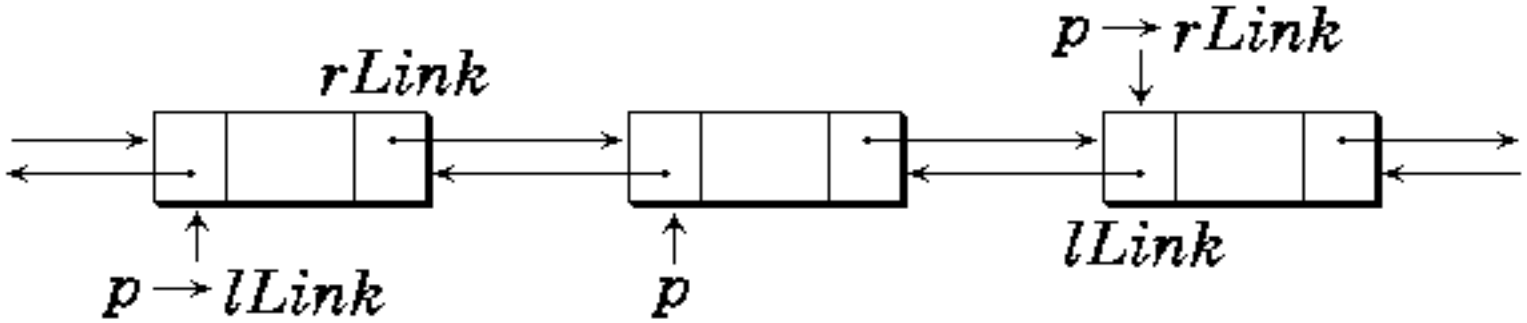


空表



结点指针的指向

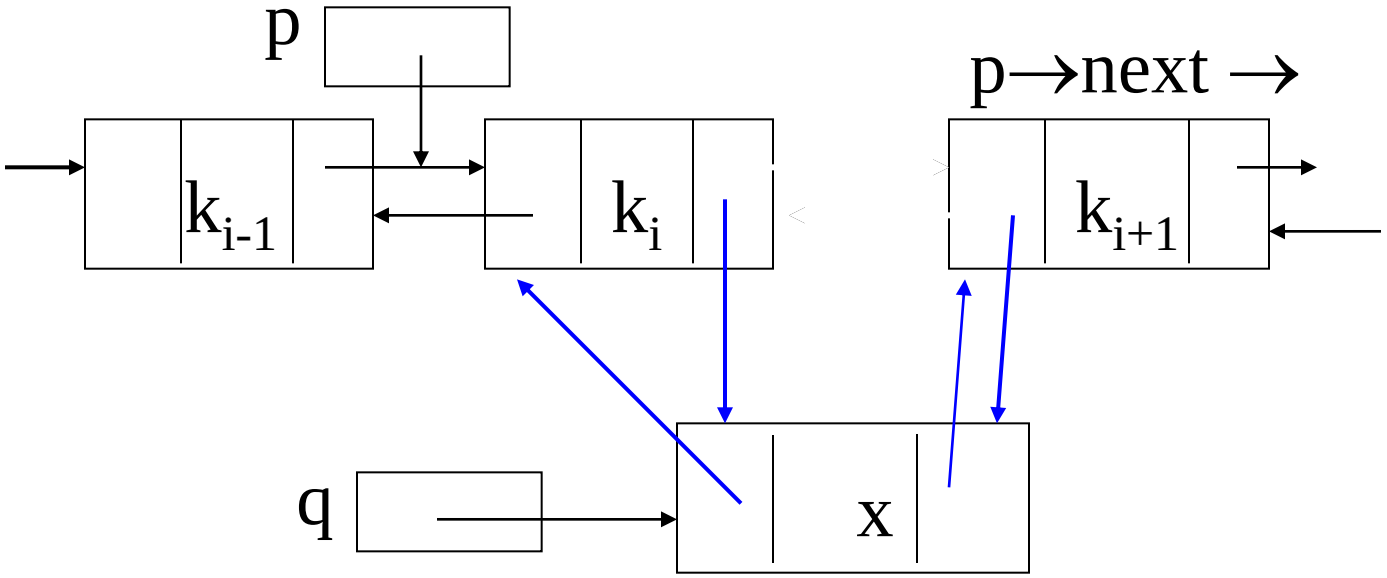
$p == p\rightarrow lLink\rightarrow rLink == p\rightarrow rLink\rightarrow lLink$



双链表结点

```
template <typename E> class link {
private:
  static Link<E> *freelist;
public:
  E element;
  Link* next;
  Link* prev;
  Link(const E& it, Link* prevp, Link* nextp)
  { element=it; prev=prevp; next=nextp; }
  Link(Link* prevp=NULL,Link* nextp=NULL)
  { prev=prevp;next=nextp; }
  void* operator new(size_t);
  void operator delete(void*);
}
```

双链表插入示意



q → prev = p;

q → next = p → next;

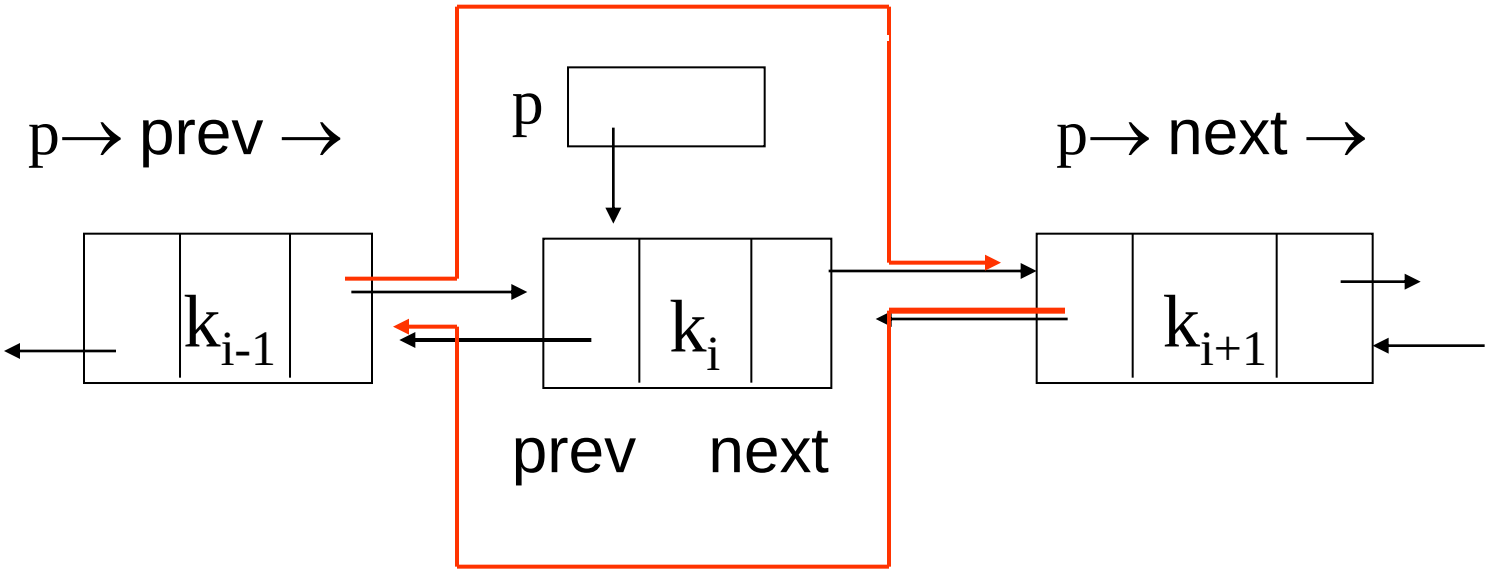
p →next → prev = q;

p → next = q;

双链表的插入

```
void insert(const E& it) {
  curr->next=curr->next->prev
          =new Link<E>(it,curr,curr->next);
  cnt++;
  }
```

双链表删除示意



$p \rightarrow prev \rightarrow next = p \rightarrow next$

$p \rightarrow next \rightarrow prev = p \rightarrow prev$

双链表的删除

```
E remove() {
  if (curr->next==tail) return NULL;
  E it =curr->next->element;
  link<E>* ltemp =curr->next;
  curr->next->next->prev=curr;
  curr->next= curr->next->next;
  delete ltemp;
  cnt--;
  return it;
}
```

线性表实现方法的比较

顺序表

- 插入、删除运算时间代价O(n)
- 预先申请固定长度的数组
- 如果整个数组元素很满，则没有结构性存储开销

链表

- 插入、删除运算时间代价O(1)但找第i个元素删除运算时间代价O(n)
- 存储利用指针， 动态地按照需要为表中新的元素分配存储空间
- 每个元素都有结构性存储开销

顺序表和链表存储密度的临界值

n表示线性表中当前元素的数目，
P表示指针的存储单元大小小(通常为4个字节)
E表示数据元素的存储单元大小
D表示可以在数组中存储的线性表元素的最大数目

- 空间需求
  - 顺序表的空间需求为DE
  - 链表的空间需求为n(P+E)
- n的临界值，即n>DE/(P+E)
  - n越大，顺序表的空间效率就更高
  - 如果P=E，则临界值为n=D/2

根据应用选择顺序表和链表

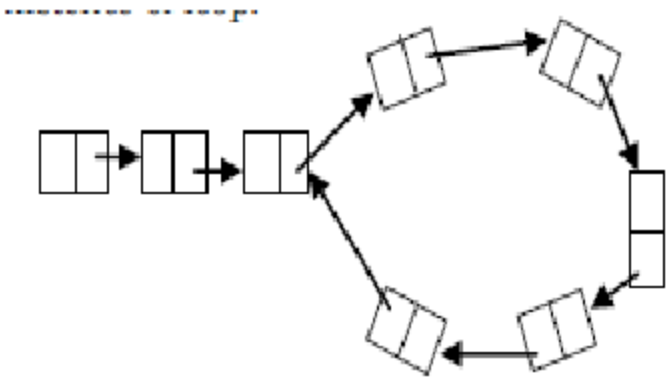
顺序表

- 结点总数目大概可以估计
- 线性表中结点比较稳定（插入删除操作少）
- n>DE/(P+E)

链表

- 结点数目无法预知
- 线性表中结点动态变化（插入删除多）
- n<DE/(P+E)

判定给定的链表是以NULL结尾，还是形成一个环。



蛮力法：例如考虑上面的链表，其中包含一个环。这个链表与常规链表的区别在于，其中有两个结点的后继结点是相同的。在常规链表中是不存在环的，每个结点的后继结点是唯一的。换言之，链表中若出现（多个结点的）后继指针重复，就表明存在环。

判定给定的链表是以NULL结尾，还是形成一个环。

- Floyd环判定算法：使用了两个在链表中具有不同移动速度的指针。一旦它们进入环便会相遇，即表示存在环。

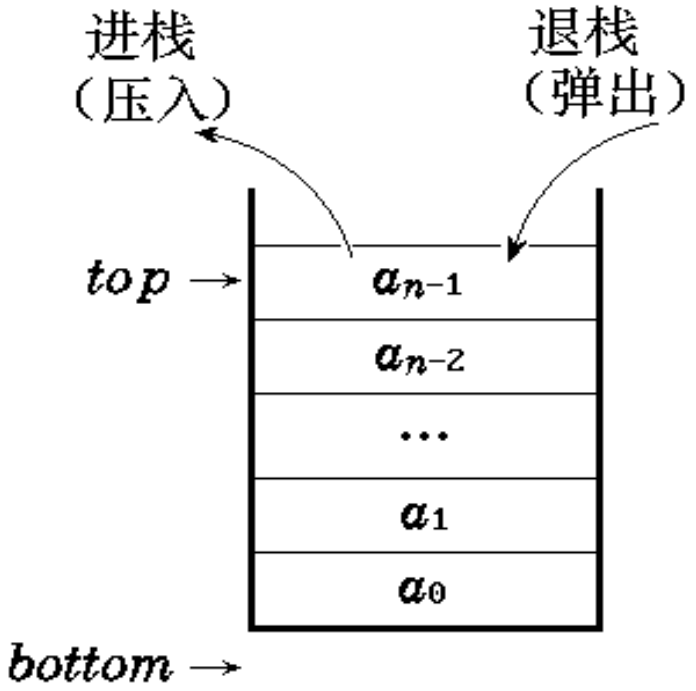
```
boolean DoesLinkedListContainsLoop(ListNode head) {
if (head == null ) return false;
ListNode slowPtr = head, fastPtr = head;
while (fastPtr.getNext()!= null && fastPtr.getNext().getNext()!= null )
    {  slowPtr = slowPtr.getNext();
       fastPtr = fastPtr.getNext().getNext();
       if ( slowPtr == fastPtr ) return true;
     }
return false;
}
```

时间复杂度为**O(*n*)**，空间复杂度为**O(1)**。

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)



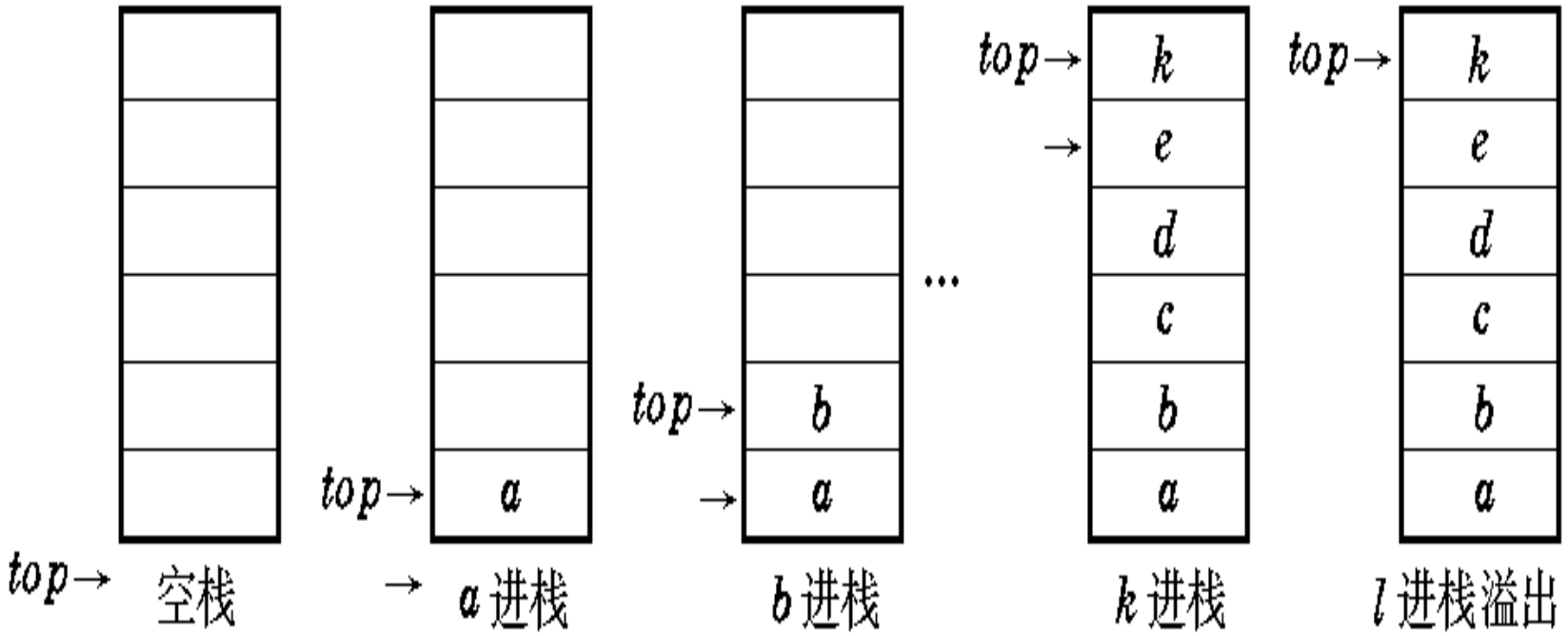
顺序栈

```
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;}
```

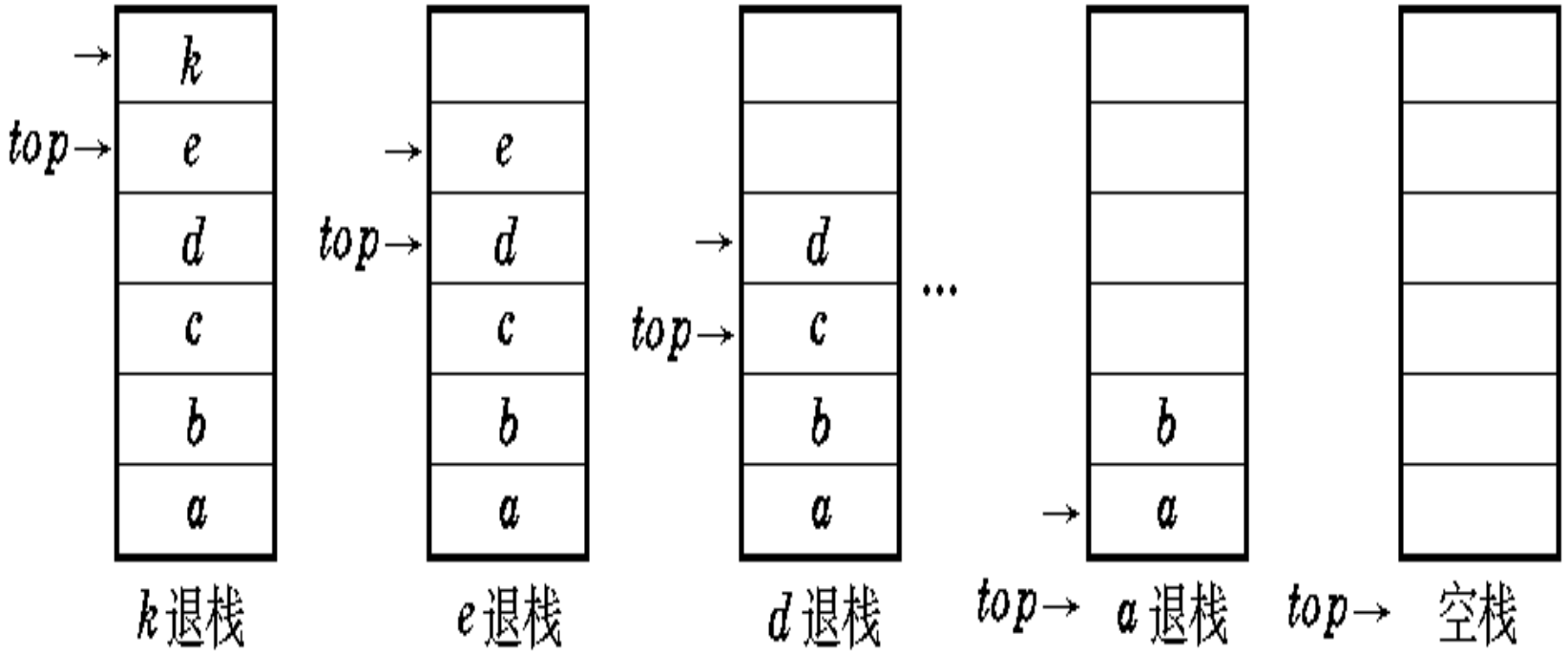
进栈、出栈算法

```
void push(const E& it)
{ Assert (top!=maxsize, “Stack is full”);
   listarray[top++] =it;
}
E pop() {
  Assert (top!=0, “Stack is empty”);
  return listarray[--top];
}
Const E& topValue() const
{Assert (top!=0, “Stack is empty”);
  return listarray[top-1];
 }
```

进栈示例

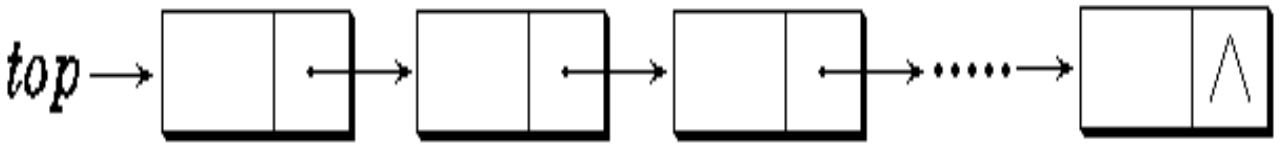


出栈示例



链式栈

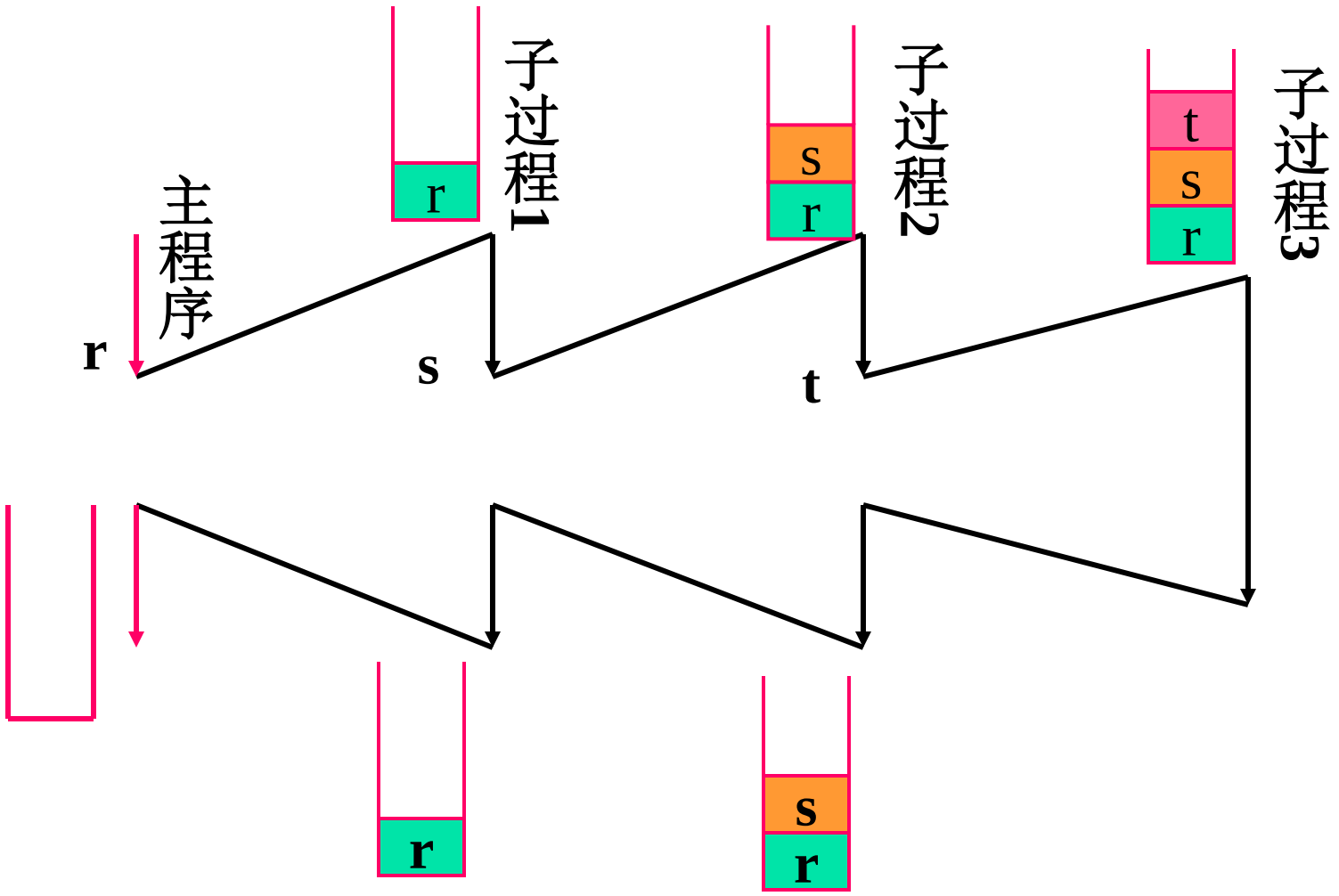
```
template < typename E > class LStack :public Stack<E> {
private:
link<E> * top;
int size;
public:
LStack(int sz =DefaultListSize){top = NULL; size=0;}
~LStack() { clear(); }
void clear(){
  while (top!=NULL)
   {Link<E>*temp=top;top=top->next; delete temp;}
   size=0;
}
```



进栈、出栈算法

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

过程的嵌套调用



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");
       }
}
```

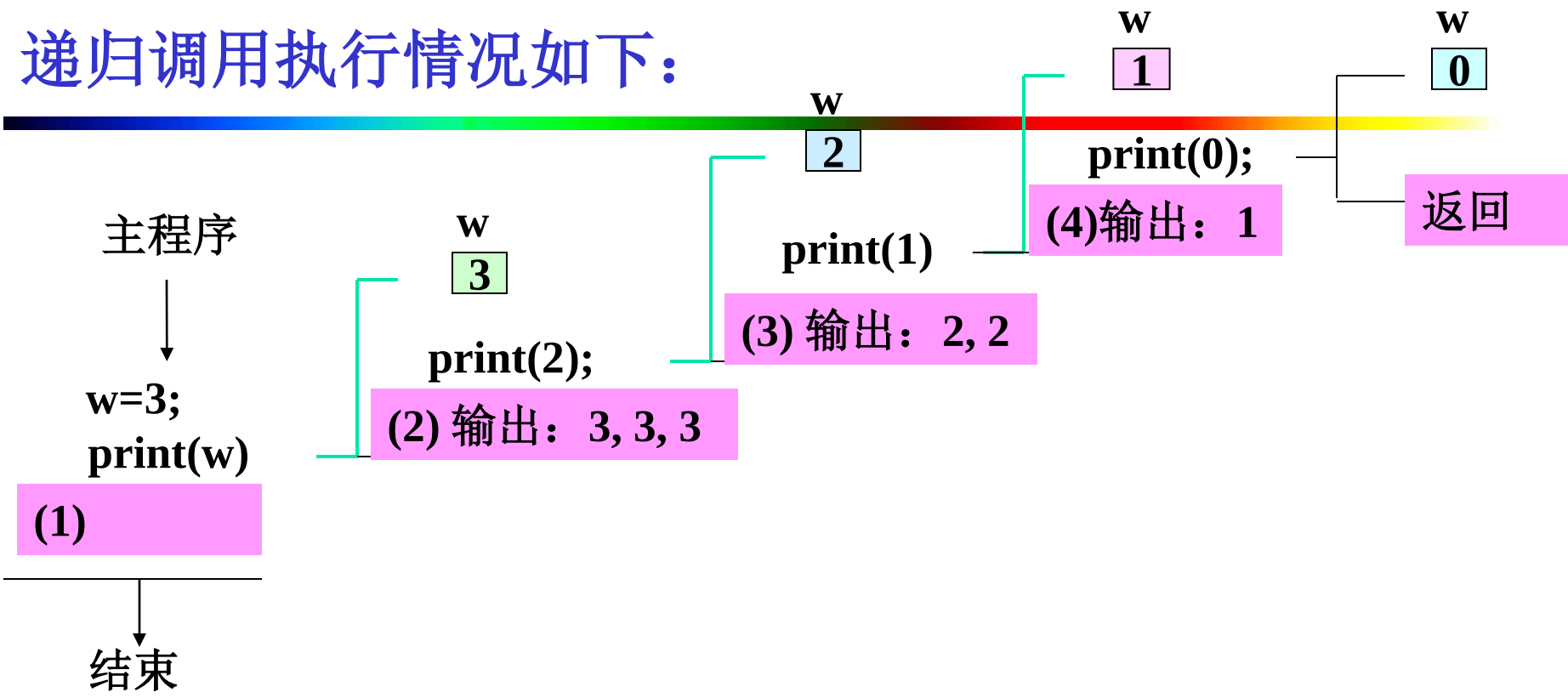
运行结果：

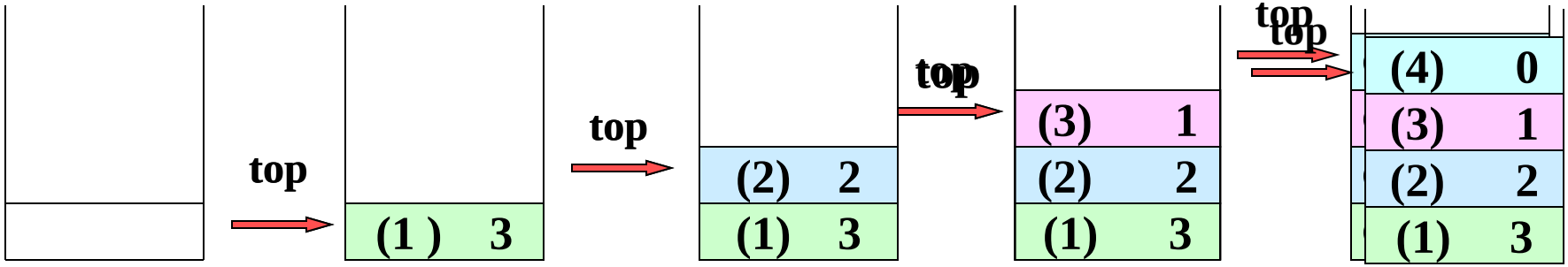
1，

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

Tower of Hanoi问题

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：

n=1时，直接把圆盘从A移到C。

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。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};
class TOHobj{
public:
  TOHop op;
  int num;
  Pole start,goal,tmp;
  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;}
}
```

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);
        }
}
```

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,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,goal,start));
      S.push(new TOHobj(start,goal));
      S.push(new TOHobj(num-1,start,tmp,goal)); }
    delete t; }
}
```

递归函数示例

```
void exmp(int n, int& f) {
   int u1, u2;
   if (n<2)
       f = n+1;
   else {
       exmp((int)(n/2), u1);
       exmp((int)(n/4), u2);
       f = u1*u2;
   }
}
```

数学公式

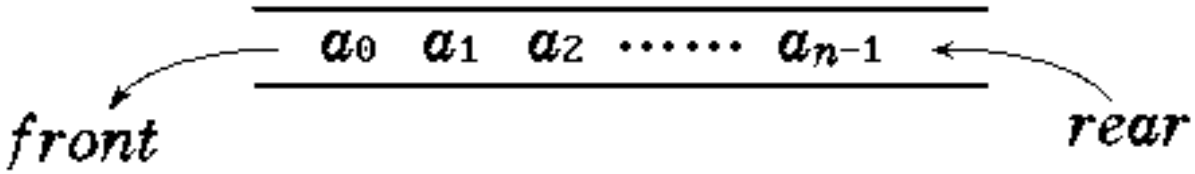
$$fu(n)=\begin{cases} n+1 & \text{当}n<2\text{时} \\ fu(\lfloor n/2 \rfloor)*fu(\lfloor n/4 \rfloor) & n\geq 2\text{时} \end{cases}$$

函数调用及返回的步骤

- 调用
  - 保存调用信息（参数，返回地址）
  - 分配数据区（局部变量）
  - 控制转移给被调函数的入口
- 返回
  - 保存返回信息
  - 释放数据区
  - 控制转移到上级函数（主调用函数）

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)**

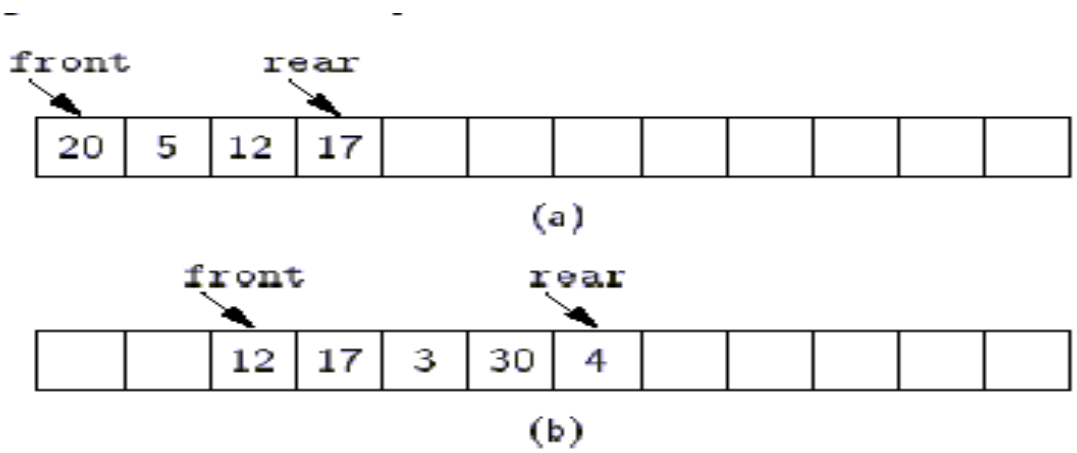


- 特点

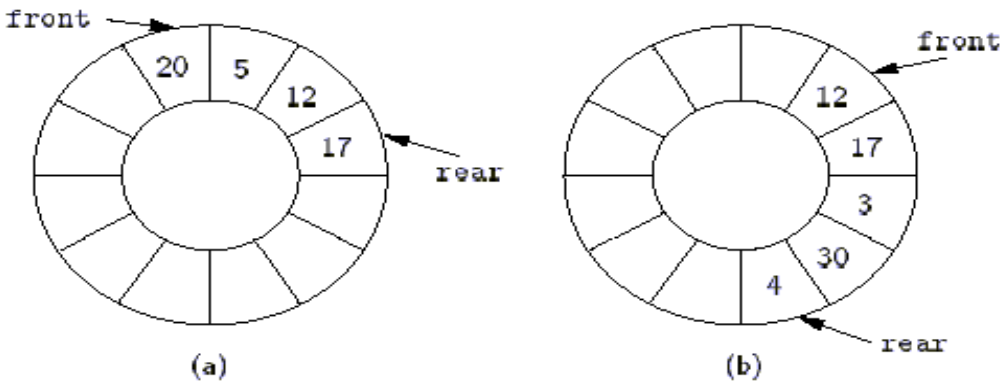
  先进先出（***FIFO***）
- 主要操作
  - –入队(enqueue) 、出队(dequeue)
  - –取队首元素(frontValue)

顺序队列 (Queue)

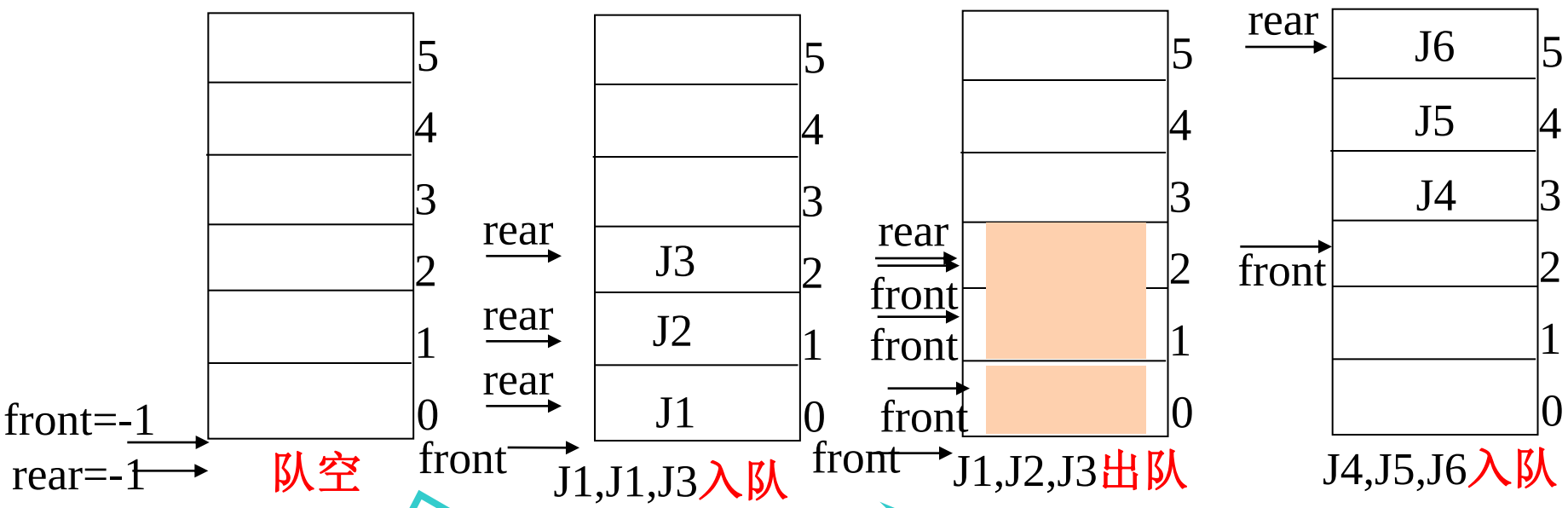
顺序队列



顺序循环队列



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

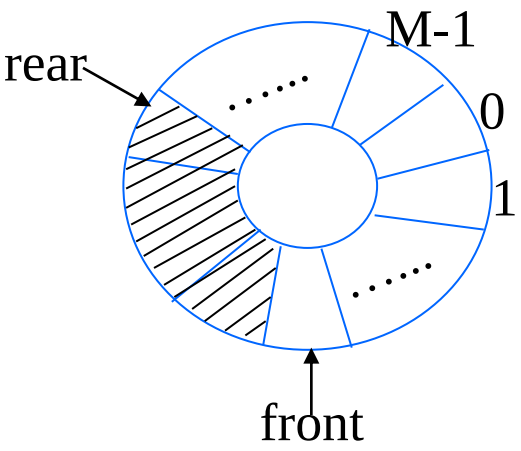
空队列条件：front==rear
入队列：sq[++rear]=x;
出队列：x=sq[++front];

存在问题

设数组维数为M，则：

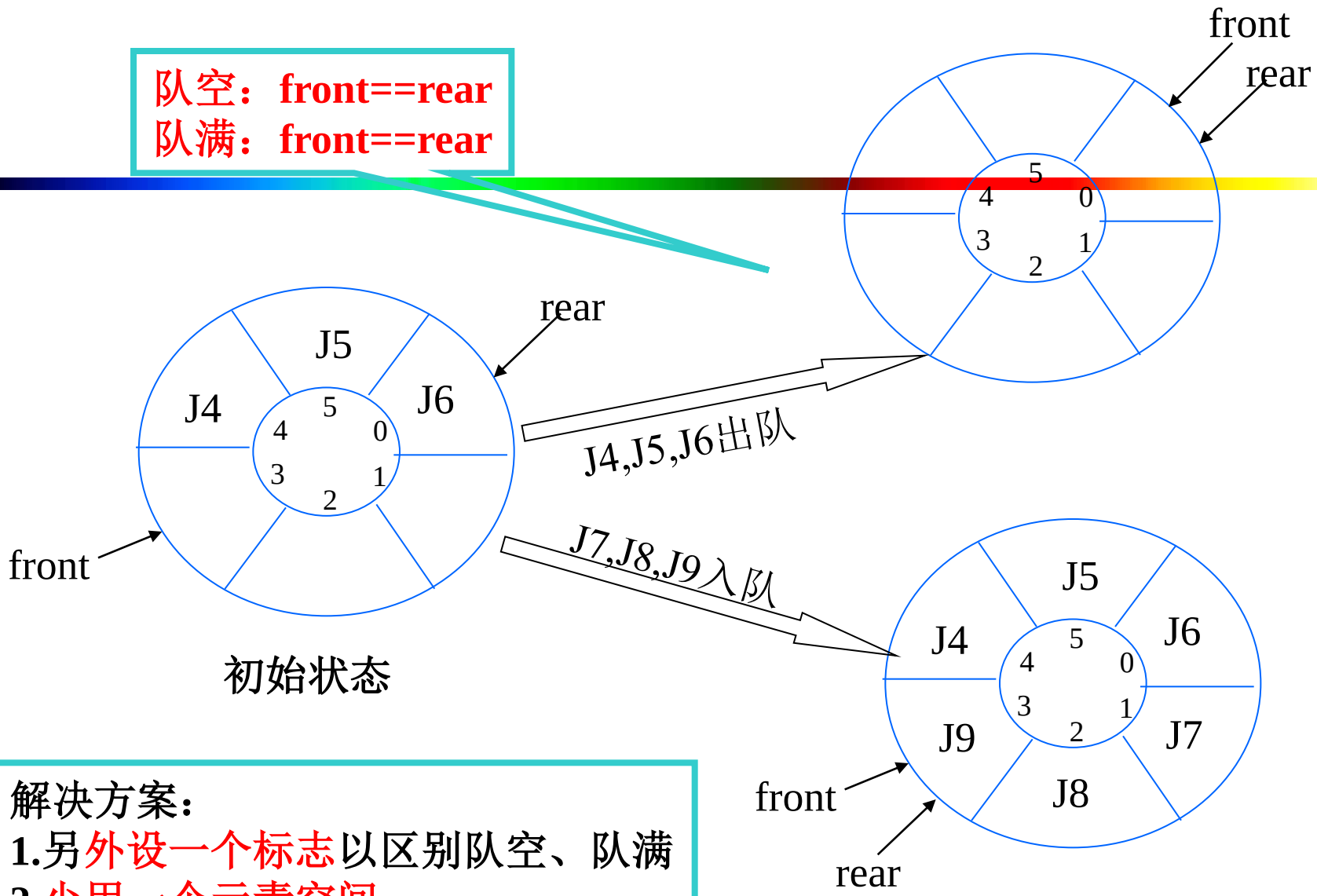
- 当front=-1,rear=M-1时，再有元素入队发生溢出——真溢出
- 当front≠-1,rear=M-1时，再有元素入队发生溢出——假溢出

- 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;
       listArray = new E [maxsize];
  }
  ~AQueue() { delete [] listArray; }
  void clear() {front =1; rear = 0; }
```

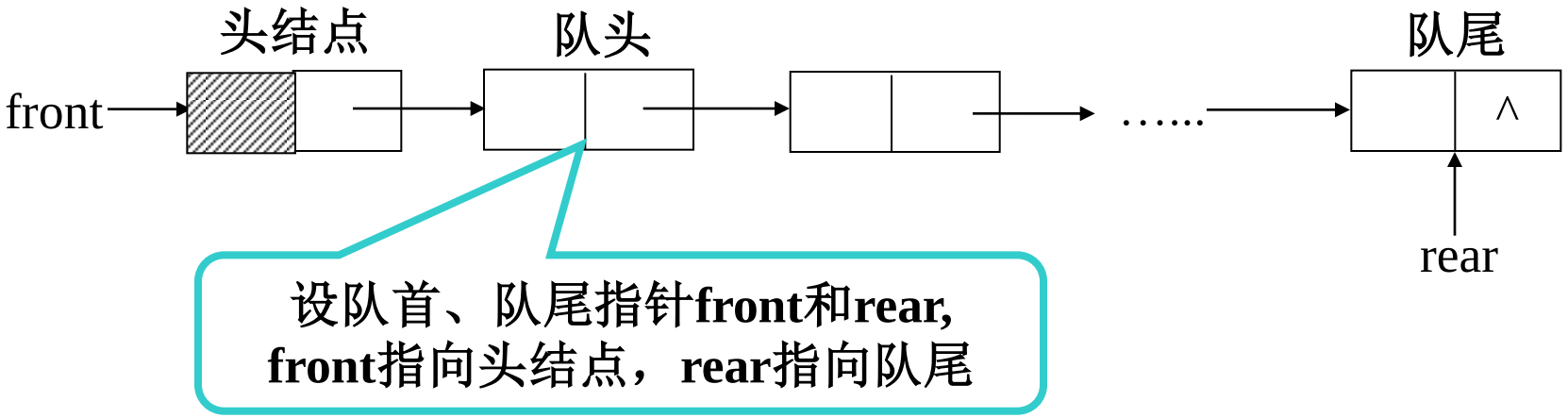
顺序队列类的实现

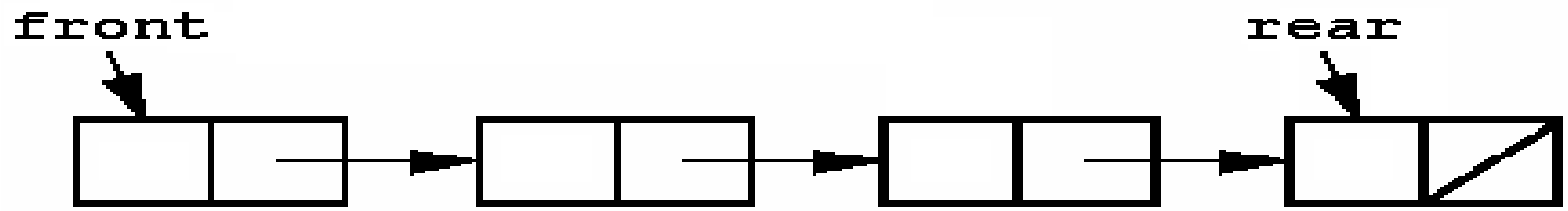
```
void enqueue(const E& it) {
  Assert (((rear+2)%maxsize)!=front, “Queue is full”);
  rear=(rear+1)%maxsize;
  listArray[rear]=it;
}
E dequeue(){
  Assert (length()!=0, “Queue is empty”);
  E it=listArray[front];
  front=(front+1)%maxsize;
  return it;
}
```

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;}
};
```

链式队列


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


front
rear

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() {
  while (front ->next!= NULL) {
   rear = front;front = front->next;delete rear; }
  rear = front;size=0;
}
void enqueue(const E& it) {
 rear->next=new Link<E>(it, NULL);
 rear = rear->next;
  size++;
}
```

链式队列类的实现

```
E dequeue() {
 Assert (size!=0, “Queue is empty”);
 E it=front->next->element;
 Link<E> *ltemp=front->next;
 front ->next= ltemp->next;
 if (rear == ltemp) rear = front;
 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;}
};
```

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，他们的标号相同。
- 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（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
    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
    }
```