第三章　　栈和队列

栈
队列
优先级队列

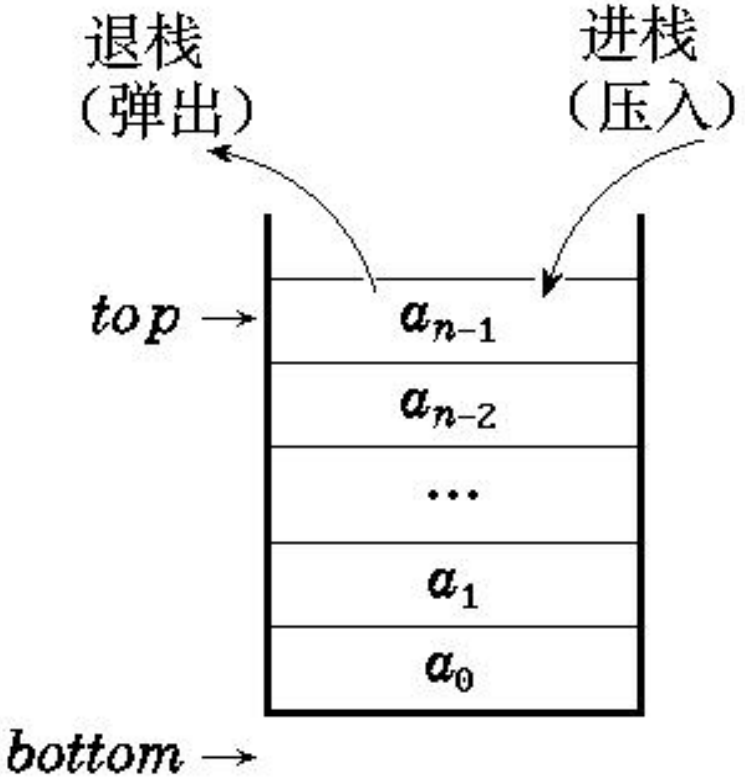
} 线性结构

与表不同的是，它们都是限制存取位置的线性结构

§3.1 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

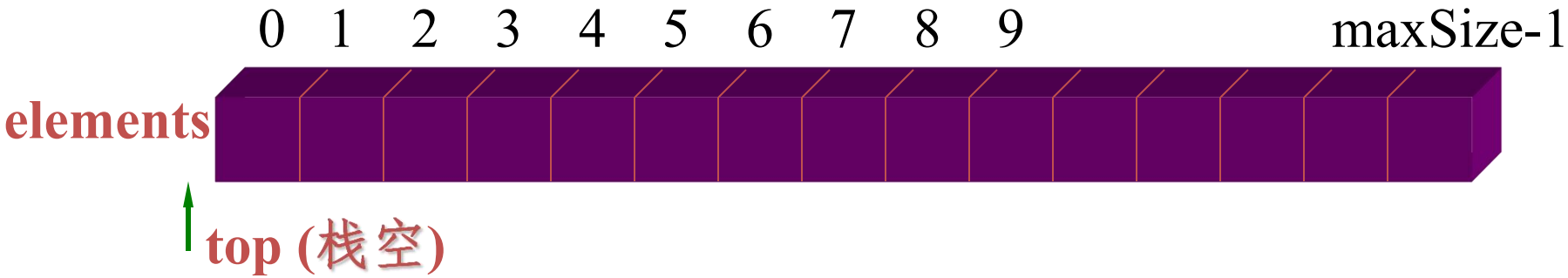
  后进先出(*LIFO*)



栈的抽象数据类型

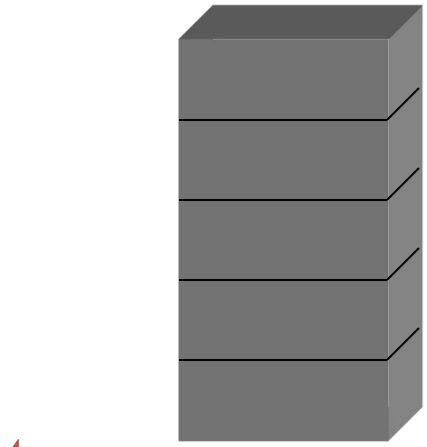
```
template <class E> class Stack {
public:
    Stack ( int=10 );                    //构造函数
    void Push ( const E & item);  //进栈
    E Pop ( );                           //出栈
    E getTop ( );                        //取栈顶元素
    void makeEmpty ( );                  //置空栈
    int IsEmpty ( ) const;               //判栈空否
    int IsFull ( ) const;                //判栈满否
}
```

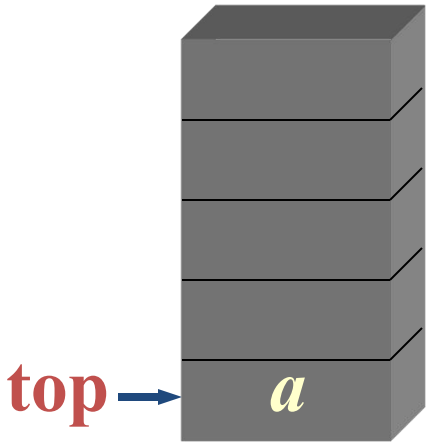
栈的数组表示 — 顺序栈

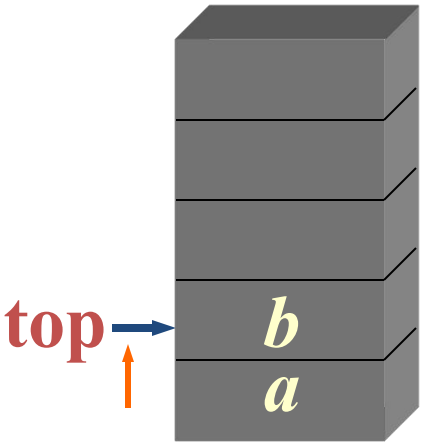


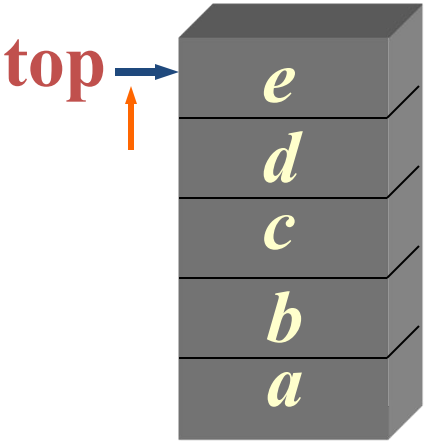

```
#include <assert.h>
template <class E>
class SeqStack : public Stack<E> {
private:
    int top;                        //栈顶指针
    E *elements;                    //栈元素数组
    int maxSize;                    //栈最大容量
    void overflowProcess();         //栈的溢出处理
```

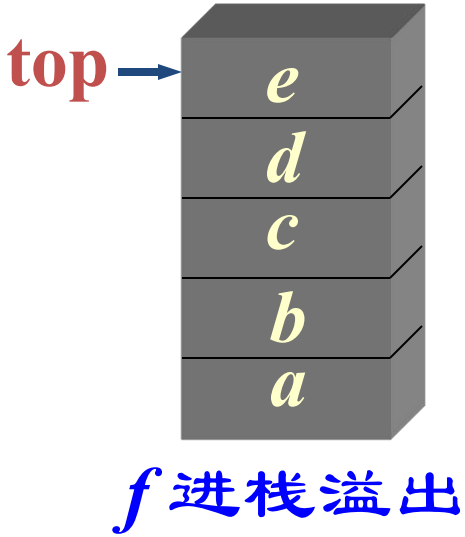
```
public:
    Stack (int sz = 10);                    //构造函数
    ~Stack ( ) { delete [ ] elements; }
    void Push (E x);                        //进栈
    int Pop (E& x);                         //出栈
    int getTop (E& x);                      //取栈顶
    void makeEmpty ( ) { top = -1; }        //置空栈
    int IsEmpty ( ) const { return top == -1; }
    int IsFull ( ) const
        { return top == maxSize-1; }
}
```



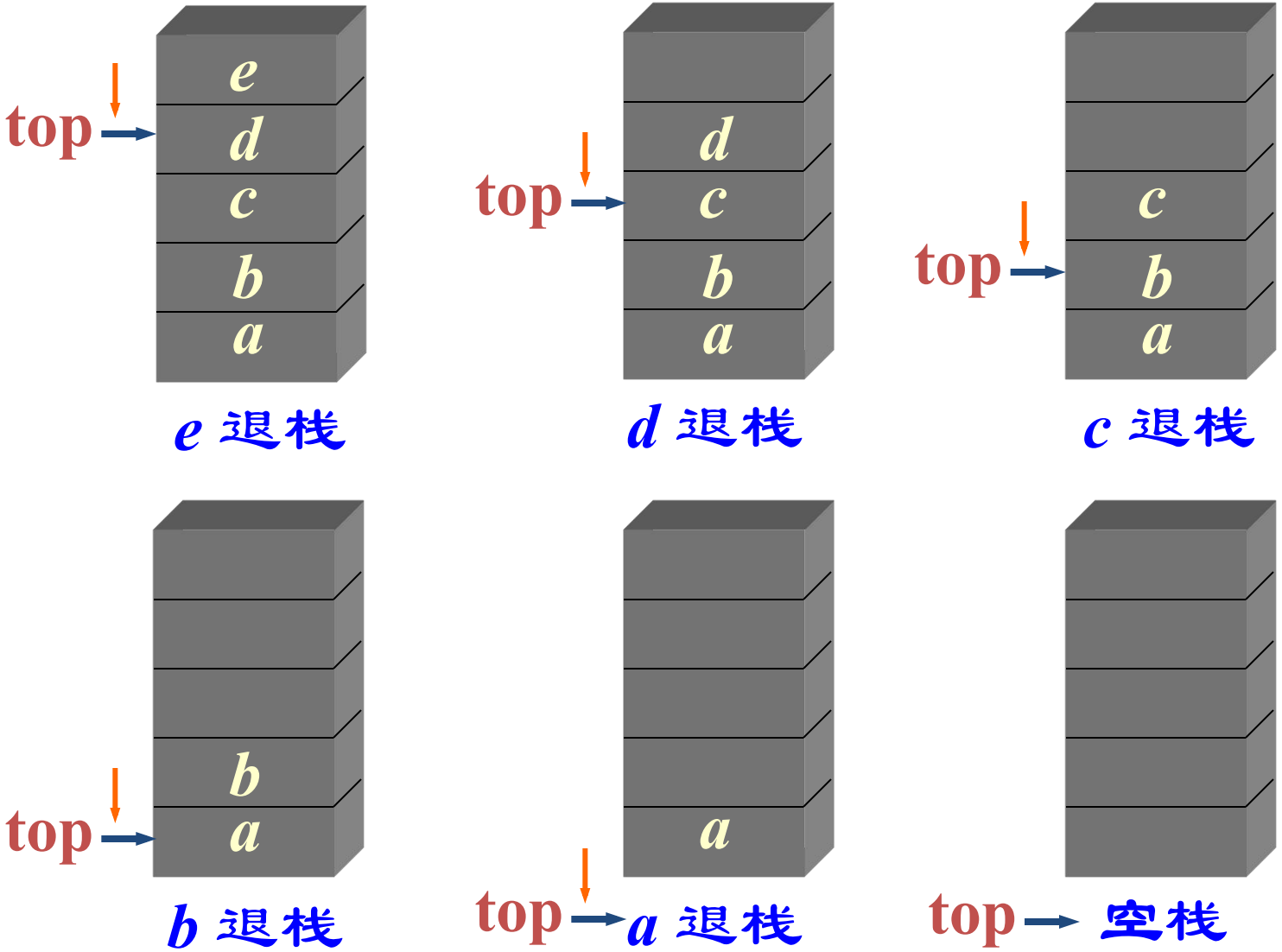








进栈示例


top
e
d
c
b
a
e 退栈
top
d
c
b
a
d 退栈
top
c
b
a
c 退栈
top
b
a
b 退栈
top
a
a 退栈
top
空栈

退栈示例

顺序栈的操作

```
template <class E>
void SeqStack<E>::overflowProcess() {
//私有函数：当栈满则执行扩充栈存储空间处理
    E *newArray = new E[2*maxSize];
                //创建更大的存储数组
    for (int i = 0; i <= top; i++)
        newArray[i] = elements[i];
    maxSize += maxSize;
    delete [ ]elements;
    elements = newArray;    //改变elements指针
};
```

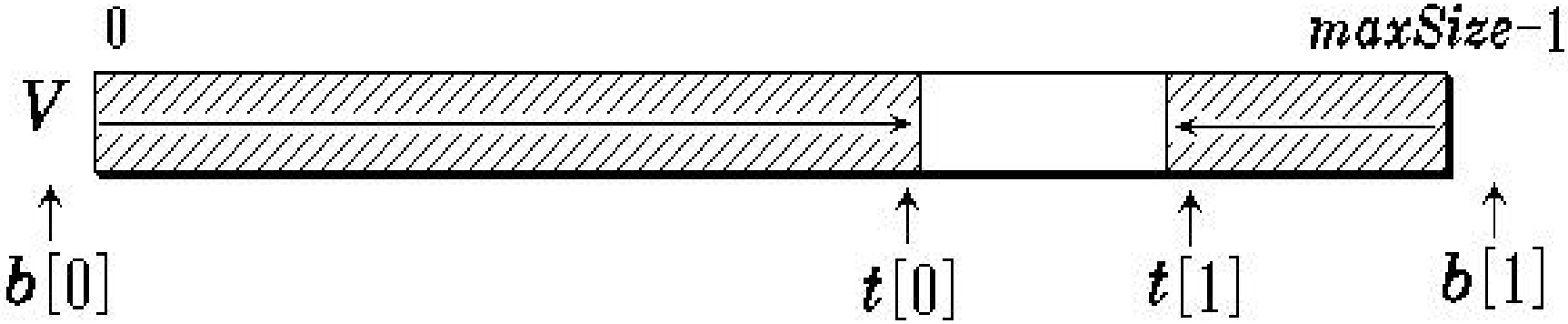
```
template <class E>
void SeqStack<E>::Push(E x) {
//若栈不满,则将元素x插入该栈栈顶,否则溢出处理
    if (IsFull() == true) overflowProcess();        //栈满
    elements[++top] = x;      //栈顶指针先加1,再进栈
};

template <class E>
bool SeqStack<E>::Pop(E& x) {
//函数退出栈顶元素并返回栈顶元素的值
    if (IsEmpty() == true) return false;
    x = elements[top--];      //栈顶指针退1
    return true;                    //退栈成功
};
```

```
template <class E>
bool Seqstack<E>::getTop(E& x) {
//若栈不空则函数返回该栈栈顶元素的地址
    if (IsEmpty() == true) return false;
    x = elements[top];
    return true;
};
```

如何合理进行栈空间分配，以避免栈溢出或空间的浪费？

- 双栈共享一个栈空间（多栈共享栈空间）
- 栈的链接存储方式——链式栈

双栈共享一个栈空间

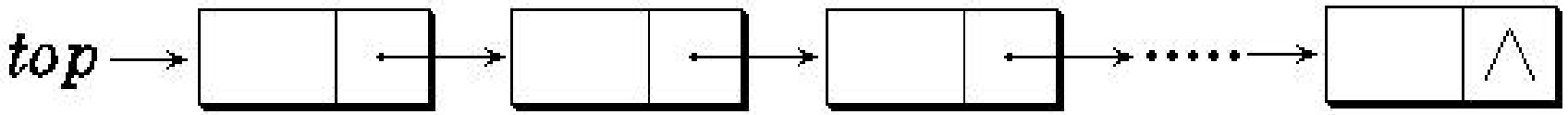


- 两个栈共享一个数组空间V[maxSize]
- 设立栈顶指针数组 t[2] 和栈底指针数组 b[2]
  t[i]和b[i]分别指示第 i 个栈的栈顶与栈底
- 初始 t[0] = b[0] = −1， t[1] = b[1] = maxSize
- 栈满条件：t[0]+1 == t[1]　　//栈顶指针相遇
- 栈空条件：t[0] = b[0]或t[1] = b[1]

//栈顶指针退到栈底

```
bool Push(DualStack& DS, E x, int i) {
    if (DS.t[0]+1 == DS.t[1]) return false;
    if (i == 0) DS.t[0]++; else DS.t[1]−−;
    DS.V[DS.t[i]] = x;
    return true;
}

bool Pop(DualStack& DS, E & x, int i) {
    if (DS.t[i] == DS.b[i]) return false;
    x = DS.V[DS.t[i]];
    if (i == 0) DS.t[0]−−; else DS.t[1]++;
    return true;
}
```

栈的链接表示 — 链式栈



- 链式栈无栈满问题，空间可扩充
- 插入与删除仅在栈顶处执行
- 链式栈的栈顶在链头

链式栈 (LinkedStack) 类的定义

```
#include <iostream.h>
#include "stack.h"
template <class E>
struct StackNode {                          //栈结点类定义
private:
    E data;                                 //栈结点数据
    StackNode<E> *link;                     //结点链指针
public:
    StackNode(E d = 0, StackNode<E> *next = NULL)
        : data(d), link(next) { }
};
```

```
template <class E>
class LinkedStack : public Stack<E> { //链式栈类定义
private:
    StackNode<E> *top;                    //栈顶指针

public:
    LinkedStack() : top(NULL) {}          //构造函数
    ~LinkedStack() { makeEmpty(); }       //析构函数
    void Push(E x);                       //进栈
    bool Pop(E& x);                       //退栈
```

```
    bool getTop(E& x) const;              //取栈顶
    bool IsEmpty() const { return top == NULL; }
    void makeEmpty();                     //清空栈的内容
     friend ostream& operator << (ostream& os,
        LinkedStack<E> & s) ;
     //输出栈元素的重载操作 <<
};
```

链式栈类操作的实现

```
template <class E>
LinkedStack<E>::makeEmpty() {
//逐次删去链式栈中的元素直至栈顶指针为空。
    StackNode<E> *p;
    while (top != NULL)            //逐个结点释放
       { p = top;  top = top->link;  delete p; }
};

template <class E>
void LinkedStack<E>::Push(E x) {
//将元素值x插入到链式栈的栈顶,即链头
 top = new StackNode<E> (x, top);     //创建新结点
     assert (top != NULL);                //创建失败退出
};
```

```
template <class E>
bool LinkedStack<E>::Pop(E& x) {
//删除栈顶结点，返回被删栈顶元素的值。
    if (IsEmpty() == true) return false;   //栈空返回
     StackNode<E> *p = top;                //暂存栈顶元素
     top = top->link;                      //退栈顶指针
     x = p->data;
    delete p;              //释放结点
    return true;
};
```

```
template <class E>
bool LinkedStack<E>::getTop(E& x) const {
     if (IsEmpty() == true) return false;   //栈空返回
     x = top->data;                         //返回栈顶元素的值
    return true;
   };
```

```
template <class E>
ostream& operator << (ostream& os,
        LinkedStack<E> & s) {
    //输出栈中元素的重载操作<<
    os << "栈中元素个数="<<s.getSize() << endl;
    LinkNode<E> * p = s.top; int i = 0;
    while (p != NULL) {
        os << ++i << ":" <<p->data << endl;
         p = p-> link;
     }
     return os;
};
```

思考：当进栈元素的编号为1, 2, …, n时，可能的出栈序列有多少种？

当进栈元素为 1, 2, …, n

输入顺序固定 **1, 2, 3, …, n** 

n = 1	n = 2	n = 3
1 种	**2 种**	**？ 种**
1	12，21	??

一般的 n：可能的出栈序列有多少种?

问题的关键不是排列，而是每一步出栈都必须合法。

用 P/O 序列描述栈操作

P **push：将下一个输入元素压栈**

O **pop：弹出当前栈顶元素**

例：n = 3 时的一种合法操作　　P P O P O O

整个序列恰好含 n 个 P 和 n 个 O

任意前缀中，P 的个数不少于 O 的个数

前缀差 P−O 就是当前栈中元素个数

合法条件：扫描过程中，栈高度始终不小于 0。

n = 3 时的合法出栈序列

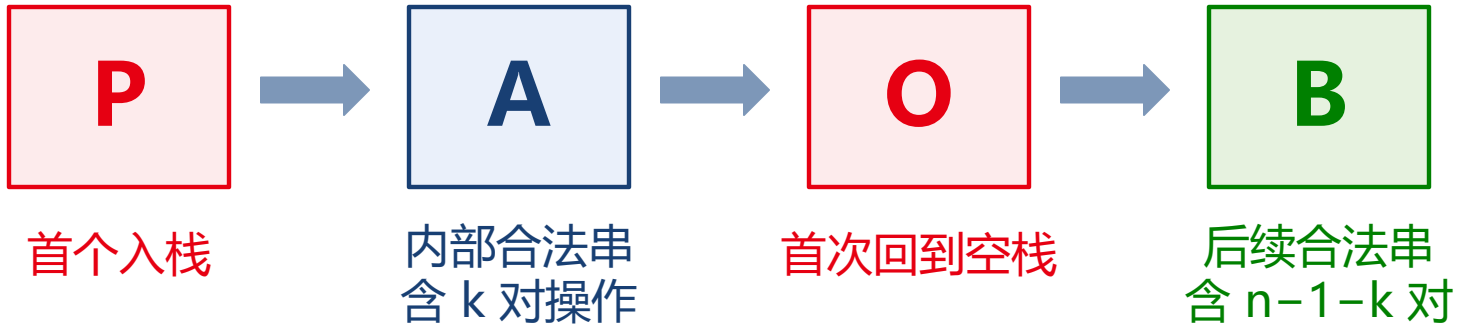
出栈序列	一种对应的 P/O 操作	结论
123	P O P O P O	**合法**
132	P O P P O O	**合法**
213	P P O O P O	**合法**
231	P P O P O O	**合法**
321	P P P O O O	**合法**

反例：312 先压入 1、2、3 并弹出 3，此时栈顶为 2，不能越过 2 直接弹出 1。

因此 $C_3 = 5$。

合法操作串的首次匹配分解

任一非空合法串中，首个 P 都有一个与它首次匹配的 O。



A 有 C_k 种选择，B 有 C_{n-1-k}种选择

固定 k 时：$C_k \times C_{n-1-k}$

k 可以取 0, 1, …, n−1。

卡特兰数递推式

$C_0 = 1$

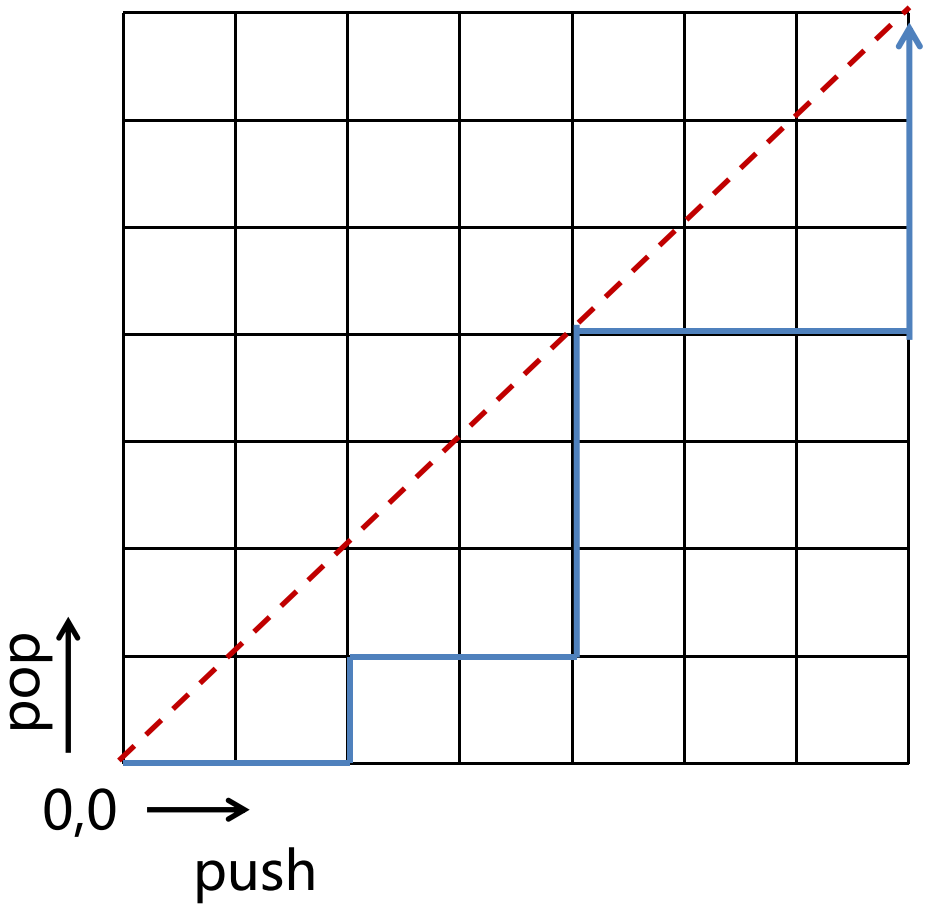
$$C_n = \sum_{k=0}^{n-1} C_k \times C_{n-1-k}$$

k	A 中操作对数	B 中操作对数	组合数
0	0	n−1	C_0 C_{n-1}
1	1	n−2	C_1 C_{n-2}
⋮	⋮	⋮	⋮
n−1	n−1	0	C_{n-1} C_0

不同 k 对应互不重叠的情况，全部相加得到 C_n 。

这正是卡特兰数的标准递推。

卡特兰数计算



计算一条从0,0 到 n,n的路径

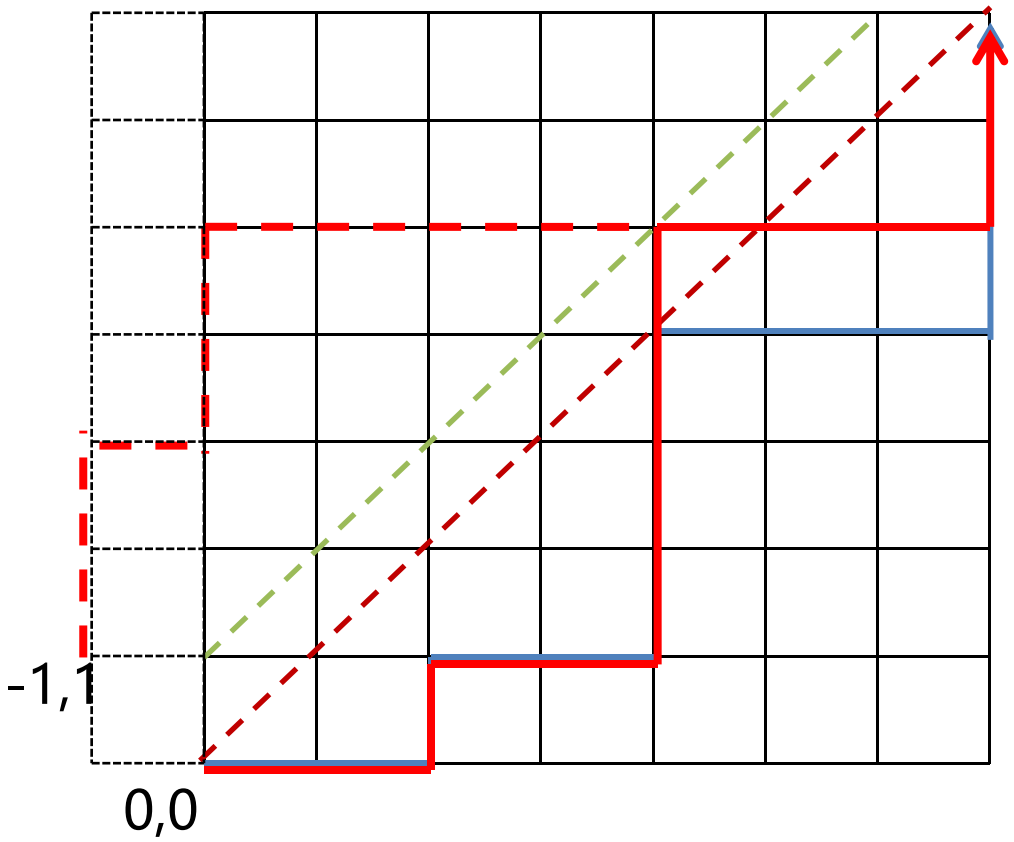
只能向右和向上移动

且需满足路径全部在y=x的对角线一下

从0,0 到 n,n的路径$\binom{2n}{n}$

非法路径有多少条呢？

卡特兰数计算



非法路径为从-1,1 到 n,n的路径条数

$$\binom{2n}{n-1} = \frac{(2n)!}{(n+1)!\,(n-1)!}$$

$$= \frac{n}{n+1} \cdot \frac{(2n)!}{(n)!\,(n)!} = \frac{n}{n+1}\binom{2n}{n}$$

$$C_n = \binom{2n}{n} - \binom{2n}{n-1} = \frac{1}{n+1}\binom{2n}{n}$$

出栈序列总数

$$C_n = \frac{1}{n+1} \cdot \frac{(2n)!}{(n!\,n!)}$$

即第 n 个卡特兰数

n	0	1	2	3	4	5	6
C_n	**1**	**1**	**2**	**5**	**14**	**42**	**132**

答案 **当进栈顺序固定为 1, 2, …, n 时，可能的出栈序列共有 C_n 种。**

增长很快：C_{10} = 16796，实际计算时要注意整数范围。

卡特兰数的动态规划计算

```
unsigned long long countPopOrders(int n) {
    vector<unsigned long long> c(n + 1, 0);
    c[0] = 1;
    for (int i = 1; i <= n; ++i) {
        for (int k = 0; k < i; ++k) {
            c[i] += c[k] * c[i - 1 - k];
        }
    }
    return c[n];
}
```

时间复杂度：$O(n^2)$　　空间复杂度：$O(n)$

unsigned long long 仅能安全保存到 C_{36}；更大的 n 需要高精度整数。

容量为 6 的顺序栈采用 `top=-1` 表示空栈。依次执行 `Push(8)`、`Push(5)`、`Pop(x)`、`Push(2)`、`Push(9)` 后，`top` 与从栈底到栈顶的元素分别是什么？

数组下标范围为 `0..9`，左栈顶 `t0=3`，右栈顶 `t1=5`。此时还能成功压入几个元素？下一次发生满栈的判定条件是什么？

“链式栈不设置固定容量，所以任何情况下 Push 都不会失败。”判断正误并说明理由。

容量为 6 的顺序栈采用 `top=-1` 表示空栈。依次执行 `Push(8)`、`Push(5)`、`Pop(x)`、`Push(2)`、`Push(9)` 后，`top` 与从栈底到栈顶的元素分别是什么？

`top=2`，元素为 `8、2、9`，`x=5`。

数组下标范围为 `0..9`，左栈顶 `t0=3`，右栈顶 `t1=5`。此时还能成功压入几个元素？下一次发生满栈的判定条件是什么？

还能压入 1 个元素，即下标 4。满栈条件为 `t0+1==t1`。

“链式栈不设置固定容量，所以任何情况下 Push 都不会失败。”判断正误并说明理由。

错。链式栈不会因预设数组容量而满，但动态内存分配仍可能失败。

栈的应用：表达式的计算

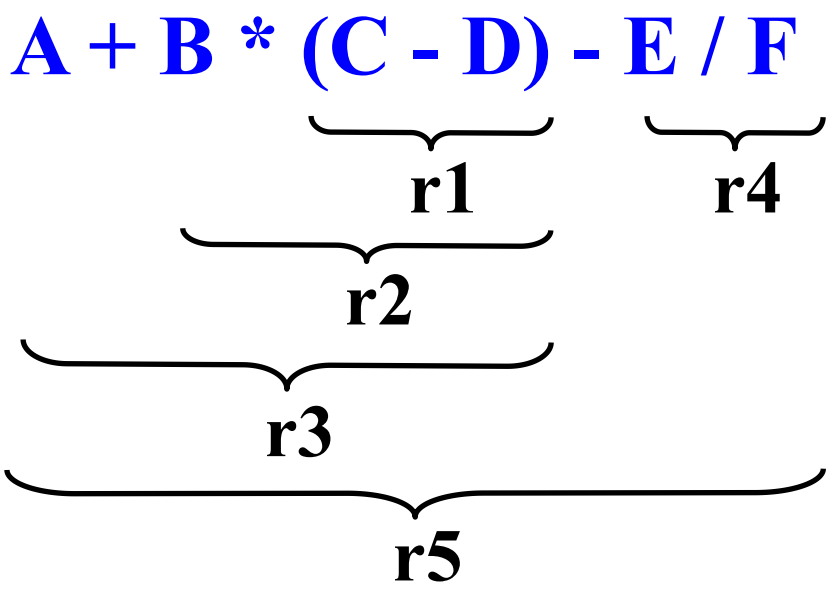
- 算术表达式有三种表示：
  - 中缀(infix)表示

    <操作数> <操作符> <操作数>，如 A+B；
  - 前缀(prefix)表示

    <操作符> <操作数> <操作数>，如 +AB；
  - 后缀(postfix)表示

    <操作数> <操作数> <操作符>，如 AB+；

表达式示例



中缀表达式 → 后缀表达式

A + B * (C - D) - E / F　　ABCD-*+EF/-

中缀表达式　A+B*(C-D)-E/F

- 表达式中相邻两个操作符的计算次序为：
  - 优先级高的先计算
  - 优先级相同的自左向右计算
  - 当使用括号时从最内层括号开始计算

后缀表达式　ABCD-*+EF/-

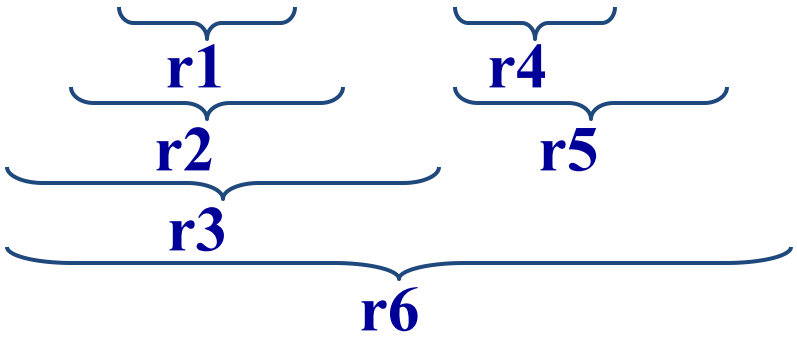
- 在后缀表达式的计算顺序中已隐含了加括号的优先次序，括号在后缀表达式中不出现。

应用后缀表示计算表达式的值

idea：

- 从左向右顺序地扫描表达式，并用一个栈暂存扫描到的操作数或计算结果。
- 扫描中遇操作数则压栈；遇操作符则从栈中退出两个操作数，计算后将结果压入栈
- 最后计算结果在栈顶

- 计算例　A B C D − * + E F ^ G / −



计算后缀表达式 A B C D − * + E F ^ G / −

步	输入	类 型	动 作	栈内容
1			置空栈	空
2	A	操作数	进栈	A
3	B	操作数	进栈	AB
4	C	操作数	进栈	ABC
5	D	操作数	进栈	ABCD
6	−	操作符	D、C 退栈，计算 C−D，结果 r1 进栈	ABr1
7	*	操作符	r1、B 退栈，计算 B*r1，结果 r2 进栈	Ar2
8	+	操作符	r2、A 退栈，计算 A+r2，结果 r3 进栈	r3

计算后缀表达式 AB C D − * + E F ^ G / −

步	输入	类 型	动 作	栈内容
9	E	操作数	进栈	r3E
10	F	操作数	进栈	r3EF
11	^	操作符	F、E 退栈，计算 E^F，结果 r4 进栈	r3r4
12	G	操作数	进栈	r3r4G
13	/	操作符	G、r4 退栈，计算 r4/G，结果 r5 进栈	r3r5
14	−	操作符	r5、r3 退栈，计算 r3−r5，结果 r6 进栈	r6

```
void Calculator :: Run ( ) {
   char ch;   double newoperand;
   while ( cin >> ch,  ch != ';' ) {
      switch ( ch ) {
        case '+' :  case '-' :   case '*' :   case '/' :
         case '^' :  DoOperator ( ch );  break;
                                        //计算
        default :  cin.putback ( ch );
                            //将字符放回输入流
                       cin >> newoperand;  //读操作数
                       S.Push( newoperand );
      }
   }
}
```

```
void Calculator :: DoOperator ( char op ) {
//从栈S中取两个操作数，形成运算指令并计算进栈
   double left, right;   bool result;
   result = Get2Operands(left, right); //退出两个操作数
   if ( !result ) return;
   switch ( op ) {
      case '+': S.Push ( left + right);  break;     //加
      case '−': S.Push ( left −  right);  break;     //减
      case '*': S.Push ( left * right);  break;     //乘
      case '/': if ( right != 0.0 ) {S.Push ( left / right); break; }
                else { cout << "除数为0!\n" );  exit(1); } //除
      case '^': S.Push ( Power(left,right) ); //乘幂
   }
}
```

```
bool Calculator :: Get2Operands(double& left,
                                double& right ) {
//从栈S中取两个操作数
    if (S.IsEmpty( ) == true)
        {cerr << "缺少右操作数！ "<< endl; return false;}
    S.Pop(right);

    if (S.IsEmpty( ) == true)
        {cerr << "缺少左操作数！ "<< endl; return false;}
    S.Pop(left);

    return true;
}
```

- 一般表达式的操作符有4种类型：

1 算术操作符　如双目操作符（+、-、*、/和%）以及单目操作符（-）。

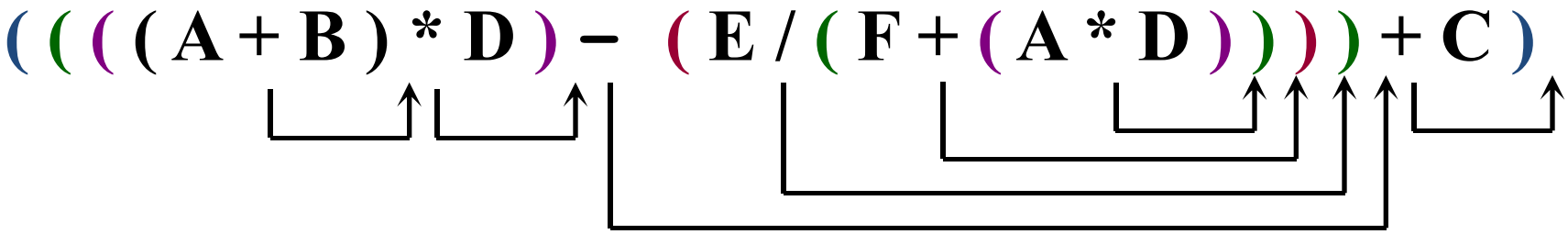
2 关系操作符　包括<、<=、==、!=、>=、>。这些操作符主要用于比较。

3 逻辑操作符　如与(&&)、或(||)、非(!)。

4 括号‘(’和‘)’，它们的作用是改变运算顺序。

中缀表示→ 后缀表示

- 先对中缀表达式按运算优先次序加上括号，再把操作符后移到右括号的后面并以就近移动为原则，最后将所有括号消去。
- 如中缀表示 (A+B)*D−E/(F+A*D)+C，其转换为后缀表达式的过程如下：



后缀表示 A B + D * E F A D * + / − C +

利用栈将中缀表示转换为后缀表示

- 使用栈可将表达式的中缀表示转换成它的后缀表示。
- 为了实现这种转换，需要考虑各操作符的优先级。

各个算术操作符的优先级

操作符 ch	;	(	*, /, %	+, −	)
isp (栈内)	0	1	5	3	6
icp (栈外)	0	6	4	2	1

- isp叫做栈内(in stack priority)优先级
- icp叫做栈外(in coming priority)优先级。
- 操作符优先级相等的情况只出现在括号配对或栈底的“;”号与输入流最后的“;”号配对时。书上是#

中缀表达式转换为后缀表达式的算法

- 操作符栈初始化，将结束符‘;’进栈。然后读入中缀表达式字符流的首字符**ch**。
- 重复执行以下步骤，直到**ch** = ‘;’，同时栈顶的操作符也是‘;’，停止循环。
  - 若**ch**是操作数直接输出，读入下一个字符**ch**。
  - 若**ch**是操作符，判断**ch**的优先级**icp**和位于栈顶的操作符**op**的优先级**isp**：

- 若 icp(ch) > isp(op)，令ch进栈，读入下一个字符ch。
- 若 icp(ch) < isp(op)，退栈并输出。
- 若 icp(ch) == isp(op)，退栈但不输出，若退出的是“(”号读入下一个字符ch。

算法结束，输出序列即为所需的后缀表达式

步	输入	栈内容	语义	输出	动作
1		;			栈初始化
2	A	;		A	操作数A输出，读字符
3	+	;	+ > ;		操作符+进栈，读字符
4	B	;+		B	操作数B输出，读字符
5	*	;+	* > +		操作符*进栈，读字符
6	(	;+*	(> *		操作符(进栈，读字符
7	C	; +*(		C	操作数C输出，读字符
8	–	; +*(	– > (		操作符–进栈，读字符
9	D	; +*(–		D	操作数D输出，读字符
10	)	; +*(–	) < –	–	操作符–退栈输出
11		; +*(	) = (		(退栈，消括号，读字符

步	输入	栈内容	语义	输出	动作
12	–	;+*	– < *	*	操作符*退栈输出
13		;+	– < +	+	操作符+退栈输出
14		;	– > ;		操作符–进栈，读字符
15	E	;–		E	操作数E输出，读字符
16	/	;–	/ > –		操作符/进栈，读字符
17	F	;–/		F	操作数F输出，读字符
18	;	;–/	; < /	/	操作符/退栈输出
19		;–	; < –	–	操作符–退栈输出
20		;	; = ;		;配对，转换结束

在中缀转后缀过程中，）在什么情况下会进栈？何时会出现栈外优先级与栈内优先级一样，退栈但不输出。

在常见的“栈内/栈外优先级”中缀转后缀算法中：

- 右括号）永远不会真正进栈。
- 当扫描到）时，依次弹出栈顶运算符并输出，直到遇到栈内的左括号（。
- 此时栈外的）与栈内的（优先级相同，于是将（退栈，但（和）都不输出。

计算后缀表达式：`15 3 2 + / 7 4 - *`。要求列出每次遇到操作符后的栈内容。

把中缀表达式 `(M-N*P)+(Q/R)` 转换为后缀表达式，并写出出栈入栈情况。

后缀表达式 `6 2 - *` 在何处失败？属于哪一类错误？

计算后缀表达式：`15 3 2 + / 7 4 - *`。要求列出每次遇到操作符后的栈内容。

- 执行 `+` 后：`15，5`
- 执行 `/` 后：`3`
- 执行 `-` 后：`3，3`
- 执行 `*` 后：`9`

把中缀表达式 `(M-N*P)+(Q/R)` 转换为后缀表达式，并写出出栈入栈情况。 `M N P * - Q R / +`

后缀表达式 `6 2 - *` 在何处失败？属于哪一类错误？

执行 `-` 后栈中只有结果 4；继续执行 `*` 时只能取得一个操作数，缺少左操作数。

单调栈

单调栈仍然遵守“后进先出”，同时维持栈中关键字的单调性。

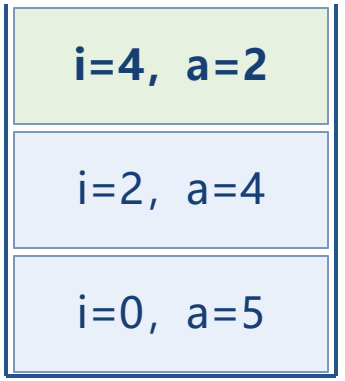
普通栈

只规定访问顺序

栈内元素没有大小关系

关注压栈、弹栈和栈顶

单调栈



数值自栈底到栈顶不增

通常保存下标

栈中元素都是尚未找到答案的候选位置。

右侧第一个严格更大元素

对每个位置 i，寻找最小的 j > i，使 a[j] > a[i]。

逐个向右扫描

每个位置最多检查 n−1 个元素

最坏比较次数约为 n(n−1)/2

时间复杂度 $O(n^2)$

单调栈处理

新元素到来时集中解决较小元素

未解决的下标继续留在栈中

每个下标至多压栈、弹栈各一次

目标答案（下标）：[2, 2, 4, 4, −1]

单调栈过程：处理下标 0～2

i	a[i]	处理前的栈（栈底→栈顶）	操作	确定的答案
0	3	空	压入 0	无
1	1	0:3	压入 1	无
2	4	0:3，1:1	弹出 1、0；压入 2	ans[1]=2，ans[0]=2

每次处理完成后的栈

值为 1 和 3 的两个下标都第一次遇到更大元素 4。

单调栈过程：处理下标 3～4

阶段	当前值	处理前的栈（栈底→栈顶）	操作	确定的答案
i=3	2	2:4	压入 3	无
i=4	5	2:4，3:2	弹出 3、2；压入 4	ans[3]=4，ans[2]=4
扫描结束	—	4:5	弹出剩余下标	ans[4]=−1

最终结果

答案下标：[2, 2, 4, 4, −1]

对应数值：[4, 4, 5, 5, −1]

栈中仍然存在的下标，右侧没有严格更大的元素。

右侧第一个严格更大元素：C++ 模板

```
vector<int> nextGreaterRight(const vector<int>& a) {
    vector<int> ans(a.size(), -1);
    stack<int> st;                  // 保存下标
    for (int i = 0; i < (int)a.size(); ++i) {
        while (!st.empty() && a[i] > a[st.top()]) {
            ans[st.top()] = i;
            st.pop();
        }
        st.push(i);
    }
    return ans;
}
```

保存下标：既能比较 a[st.top()]，也能写入 ans[st.top()]。

严格更大使用 “>”；相等元素暂时保留在栈中。

正确性与均摊复杂度

正确性　弹出 j 时，i 是扫描到的第一个严格更大元素，因此 ans[j] = i。

操作计数　每个下标压栈 1 次，最多弹栈 1 次，栈操作总数不超过 2n。

$$T(n) = O(n)$$

while 循环虽然嵌套，但同一个下标不会被反复弹出。

单调栈常见变体

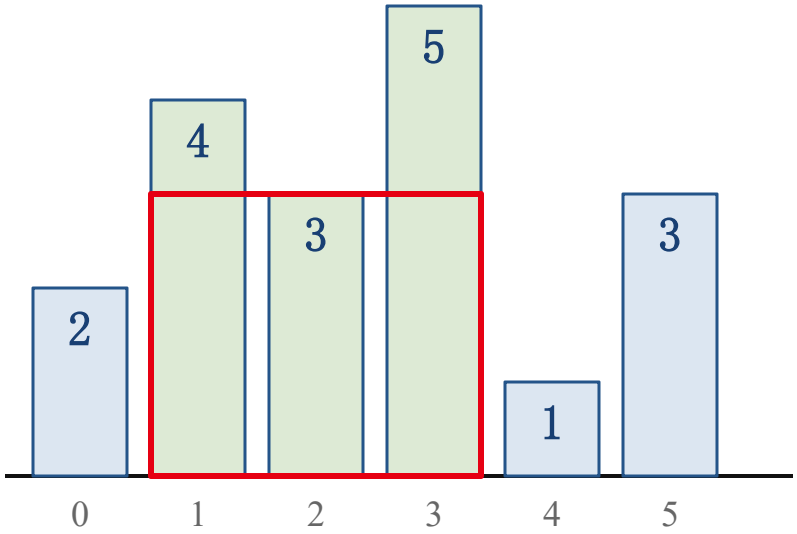
问题	扫描方式	核心条件	相等元素
右侧第一个严格更大	从左向右，解决旧下标	**a[i] > a[top]**	保留
右侧第一个大于等于	从左向右，解决旧下标	**a[i] >= a[top]**	弹出并解决
右侧第一个严格更小	从左向右，解决旧下标	**a[i] < a[top]**	保留
左侧第一个严格更大			

比较符决定重复元素如何处理。

先明确方向、大小关系和是否严格，再写弹栈条件。

柱状图中的最大矩形

高度：[2, 4, 3, 5, 1, 3]



高 = 3，宽 = 3，面积 = 9

柱状图最大矩形：C++ 实现

```
long long largestRectangleArea(vector<int> h) {
    h.push_back(0);              // 末尾哨兵
    stack<int> st;
    long long ans = 0;
    for (int i = 0; i < (int)h.size(); ++i) {
        while (!st.empty() && h[st.top()] > h[i]) {
            int height = h[st.top()];
            st.pop();
            int left = st.empty() ? -1 : st.top();
            int width = i - left - 1;
            ans = max(ans, 1LL * height * width);
        }
        st.push(i);
    }
    return ans;
}
```

示例结果

[2,4,3,5,1,3]
最大面积 = 9

识别信号

寻找最近边界

新元素使多个候选同时失效

希望每个下标只进出一次

单调栈把“向两边反复查找”转化为一次线性扫描。

§3.2 栈与递归

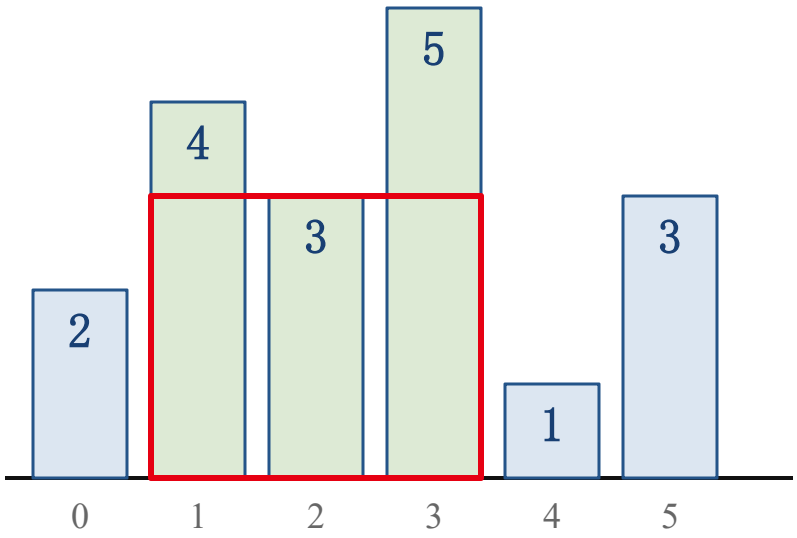
- 递归的定义

  若一个对象部分地包含它自己，或用它自己给自己定义，则称这个对象是递归的；若一个过程直接地或间接地调用自己，则称这个过程是递归的过程。

- 以下三种情况常常用到递归方法。
  - 定义是递归的
  - 数据结构是递归的
  - 问题的解法是递归的

柱状图中的最大矩形

高度：[2, 4, 3, 5, 1, 3]



高 = 3，宽 = 3，面积 = 9

当前 i	弹出下标	宽度	面积
2	1	1	4
4	3	1	5
4	2	3	9
4	0	4	8
6（哨兵）	5、4	1、6	3、6

弹出高度 h 时

左边界 = 弹栈后的新栈顶

右边界 = 当前下标 i

宽度 = 右边界 − 左边界 − 1

定义是递归的

例如，阶乘函数

$$n!=\begin{cases}1, & \text{当} n=0\text{时}\\ n*(n-1)!, & \text{当} n\geq 1\text{时}\end{cases}$$

求解阶乘函数的递归算法

```
long Factorial(long n) {
    if (n == 0) return 1;
    else return n*Factorial(n−1);
}
```

求解阶乘 $n!$ 的过程

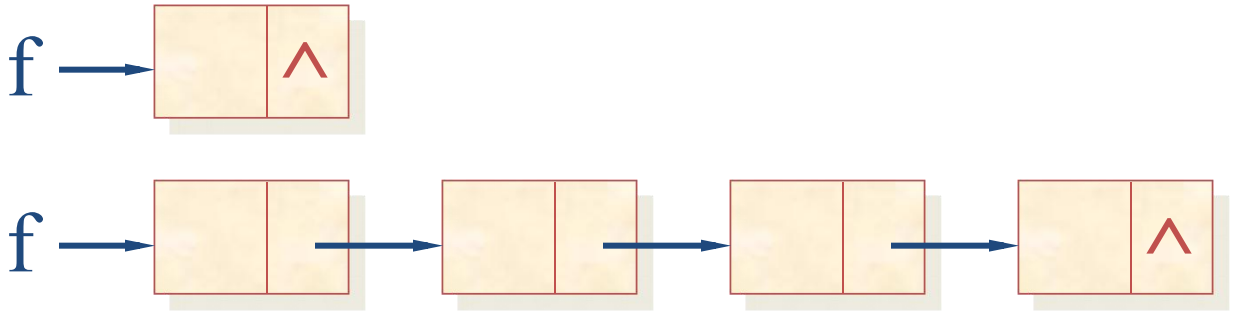

主程序 main : factorial(4)
参数 4　计算 4*factorial (3)　返回 24
参数 3　计算 3*factorial (2)　返回 6
参数 2　计算 2*factorial (1)　返回 2
参数 1　计算 1*factorial (0)　返回 1
参数 0　直接定值 = 1　返回 1
递归调用
参数传递
结果返回
返回求值

三点认识

- 对于一个较为复杂的问题，如果能够分解成几个相对简单的且解法相同或类似的子问题时，只要解决了这些子问题，那么原来的问题就迎刃而解了。（分治法）
- 当分解后的子问题可以直接解决时，停止分解。
- 递归定义的函数可以用递归过程来编程求解。递归过程直接反映了定义的结构

数据结构是递归的

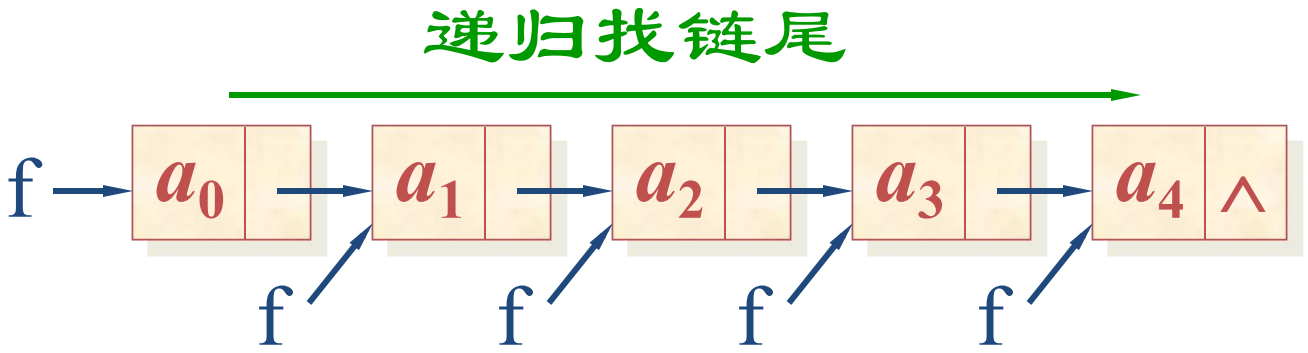
- 例如，单链表结构



- 一个结点，它的指针域为NULL，是一个单链表；
- 一个结点，它的指针域指向单链表，仍是一个单链表。

搜索链表最后一个结点并打印其数值

```
template <class E>
void Print(ListNode<E> *f) {
    if (f->link == NULL)
        cout << f->data << endl;
    else Print(f->link);
}
```



问题的解法是递归的

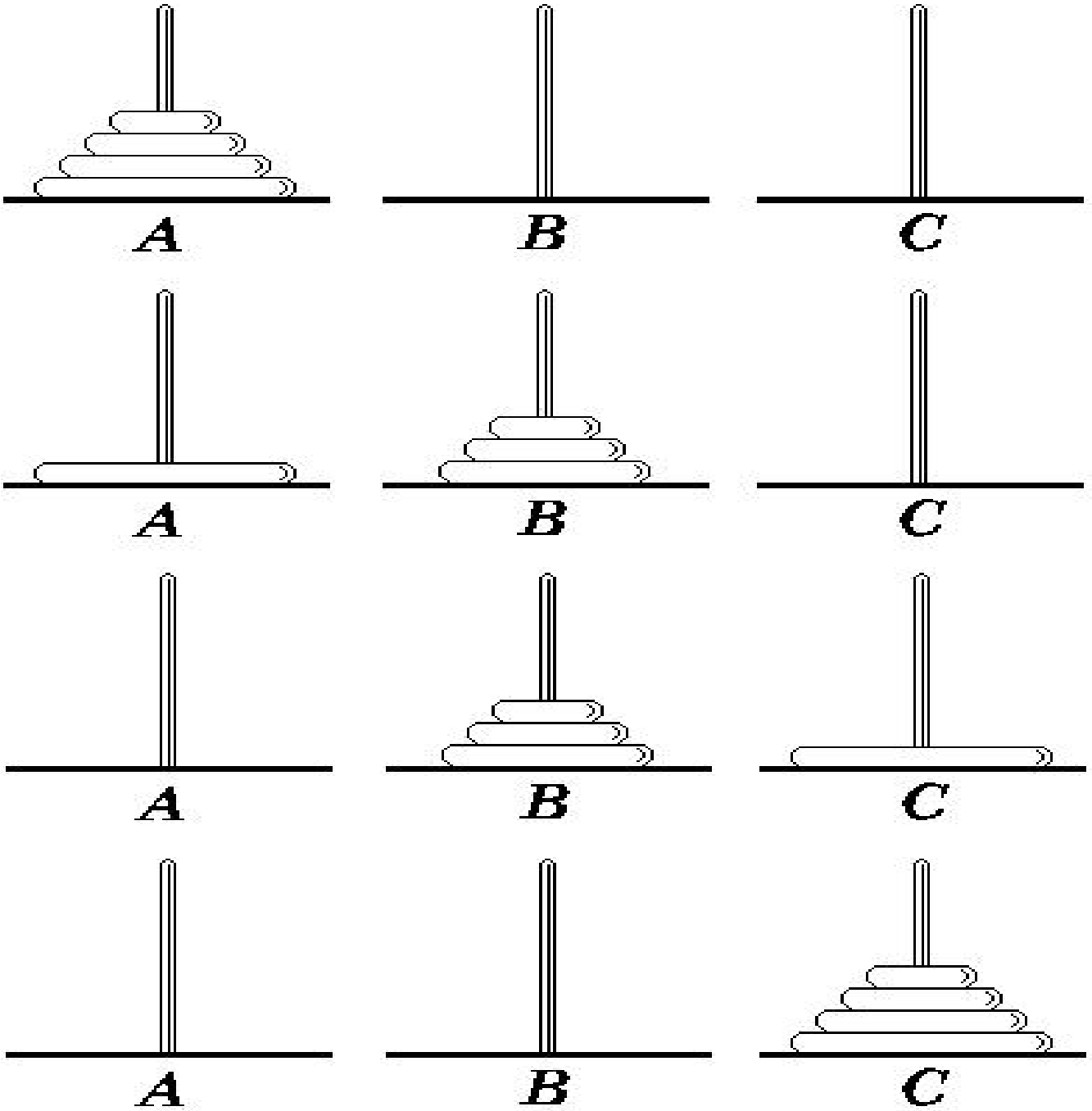
- 例，汉诺塔(Tower of Hanoi)问题的解法：

  如果 n = 1，则将这一个盘子直接从 A 柱移到 C 柱上。否则，执行以下三步：

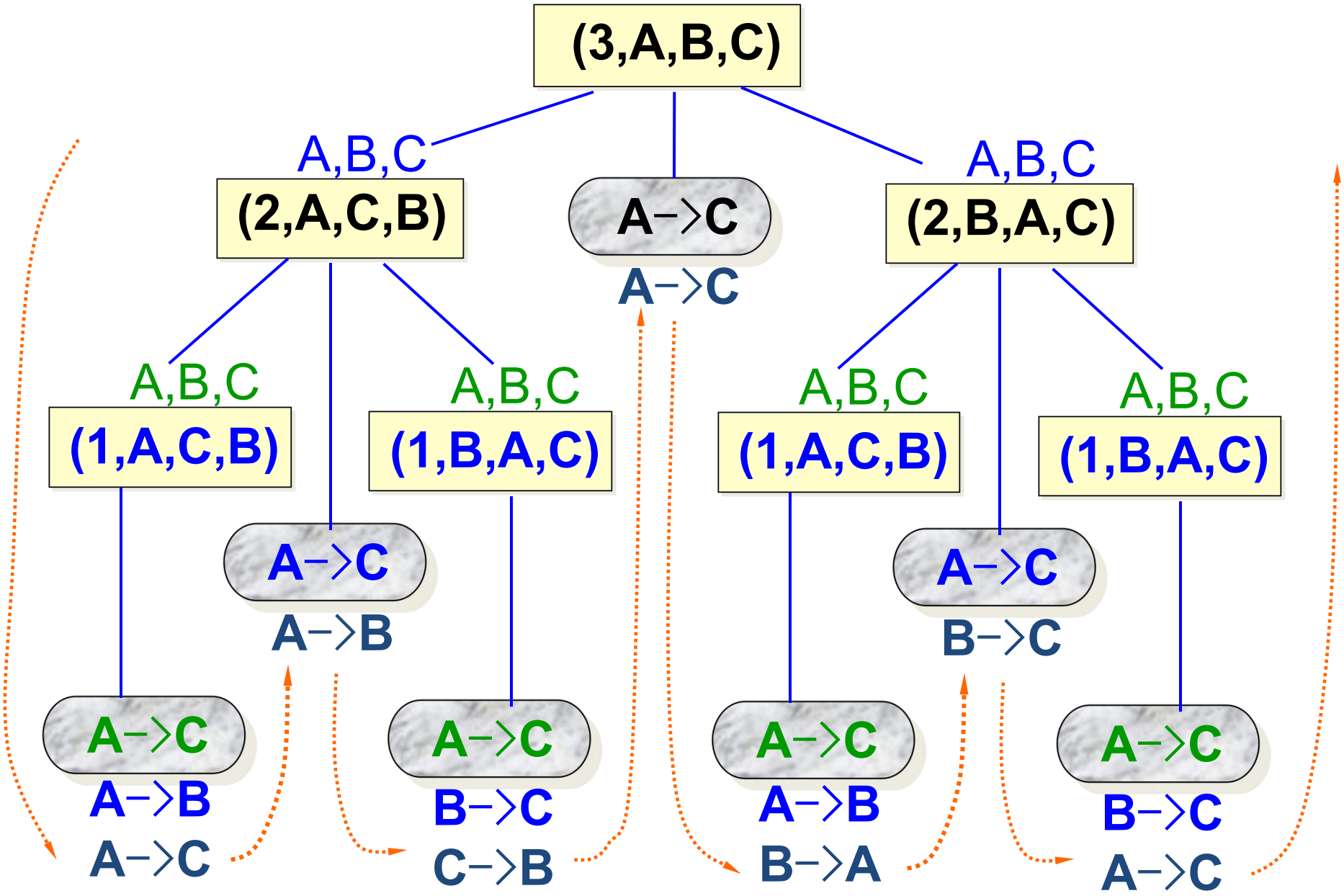
  ① 用 C 柱做过渡，将 A 柱上的 (n-1) 个盘子移到 B 柱上；

  ② 将 A 柱上最后一个盘子直接移到 C 柱上；

  ③ 用 A 柱做过渡，将 B 柱上的 (n-1) 个盘子移到 C 柱上。


A
B
C
A
B
C
A
B
C
A
B
C

```
#include <iostream.h>
void Hanoi (int n, char A, char B, char C) {
//解决汉诺塔问题的算法
    if (n == 1) cout << " move " << A << " to  "
          << C << endl;
    else {  Hanoi(n-1, A, C, B);
            cout << " move " << A << " to " << C
                  << endl;
            Hanoi(n-1, B, A, C);
          }
}
```


(3,A,B,C)
A,B,C
(2,A,C,B)
A->C
A->C
A,B,C
(2,B,A,C)
A,B,C
(1,A,C,B)
A,B,C
(1,B,A,C)
A->C
A->B
A,B,C
(1,A,C,B)
A,B,C
(1,B,A,C)
A->C
B->C
A->C
A->B
A->C
B->C
C->B
A->C
A->B
B->A
A->C
B->C
A->C

什么时候运用递归？

- 子问题应与原问题做同样的事情，且更为简单；
- 把一个规模比较大的问题分解为一个或若干规模比较小的问题，分别对这些比较小的问题求解，再综合它们的结果，从而得到原问题的解。

—分而治之策略（分治法）

- 这些比较小的问题的求解方法与原来问题的求解方法一样。

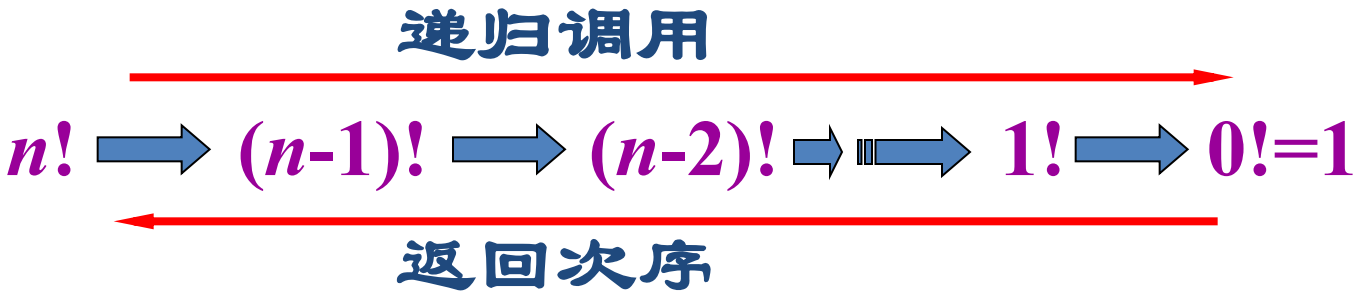
构成递归的条件

- 不能无限制地调用本身，必须有一个出口，化简为非递归状况直接处理。

```
Procedure <name> ( <parameter list> ) {
  if ( < initial condition> )    //递归结束条件
    return ( initial value );
  else                           //递归
    return (<name> ( parameter exchange ));
}
```

递归过程与递归工作栈

- 递归过程在实现时，需要自己调用自己。
- 层层向下递归，退出时的次序正好相反：

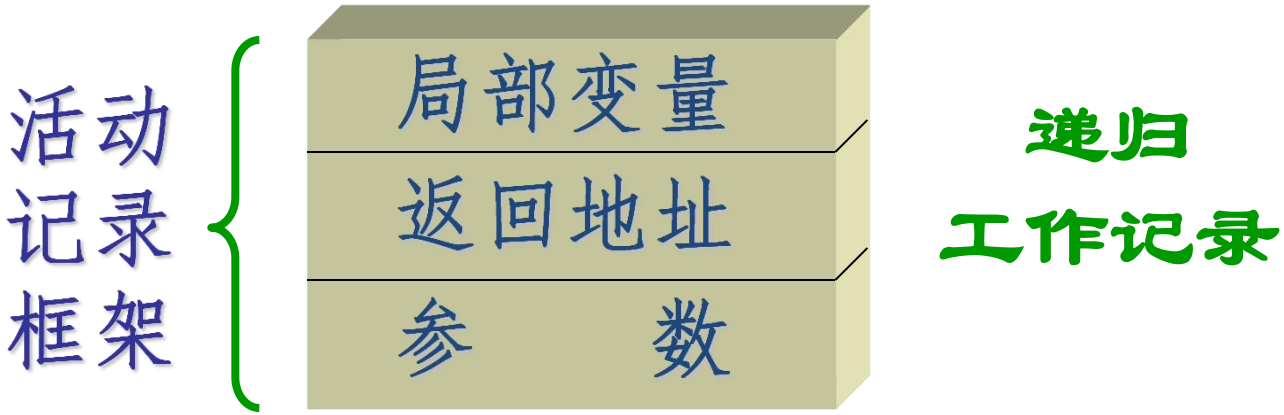


- 主程序第一次调用递归过程为外部调用；
- 递归过程每次递归调用自己为内部调用。
- 它们返回调用它的过程的地址不同。

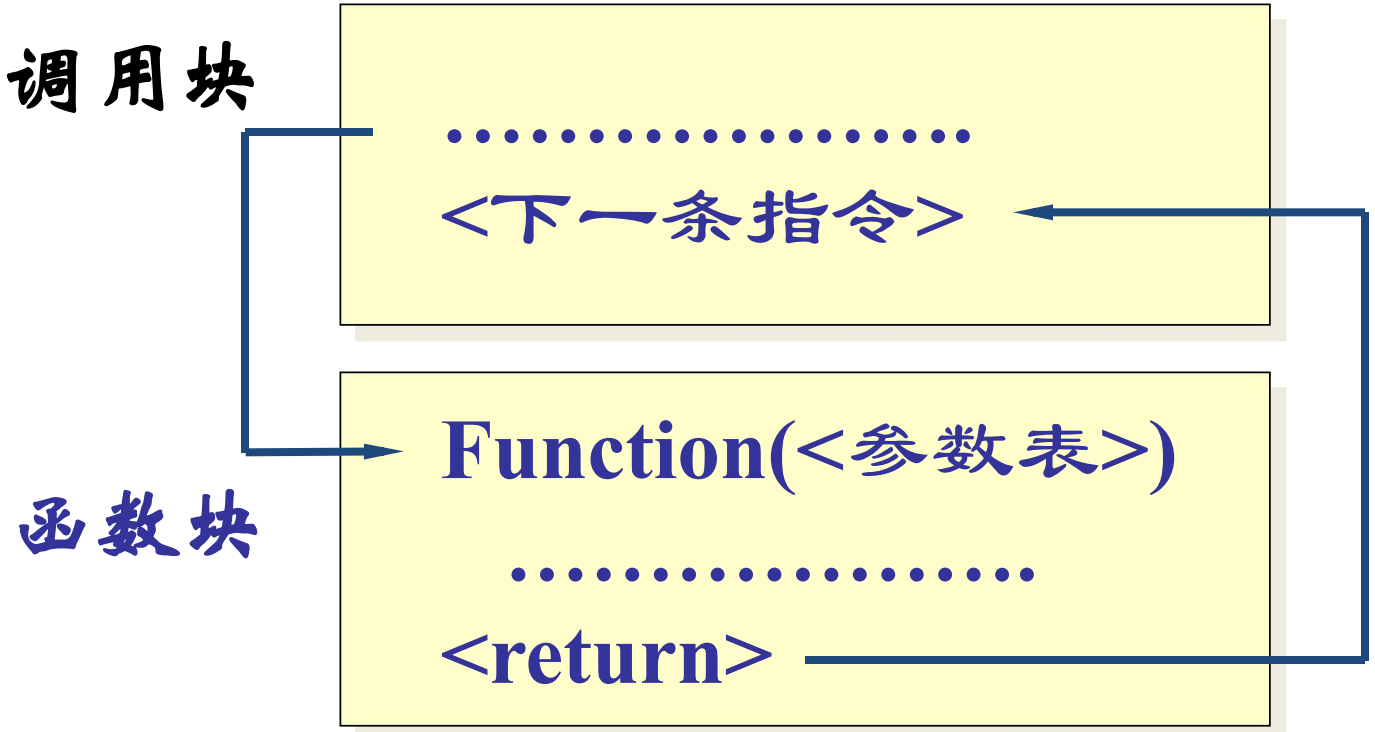
```
long Factorial(long n) {
int temp;
if (n == 0) return 1;
else temp = n * Factorial(n−1);
return temp;
}

void main() {
int n;
n = Factorial(4);
}
```

递归工作栈

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



递归工作栈



函数递归时的活动记录

计算Factorial时活动记录的内容

递归过程改为非递归过程

- 递归过程简洁、易编、易懂
- 递归过程效率低，重复计算多
- 改为非递归过程的目的是提高效率
- 单向递归和尾递归可直接用迭代实现其非递归过程
- 其他情形必须借助栈实现非递归过程

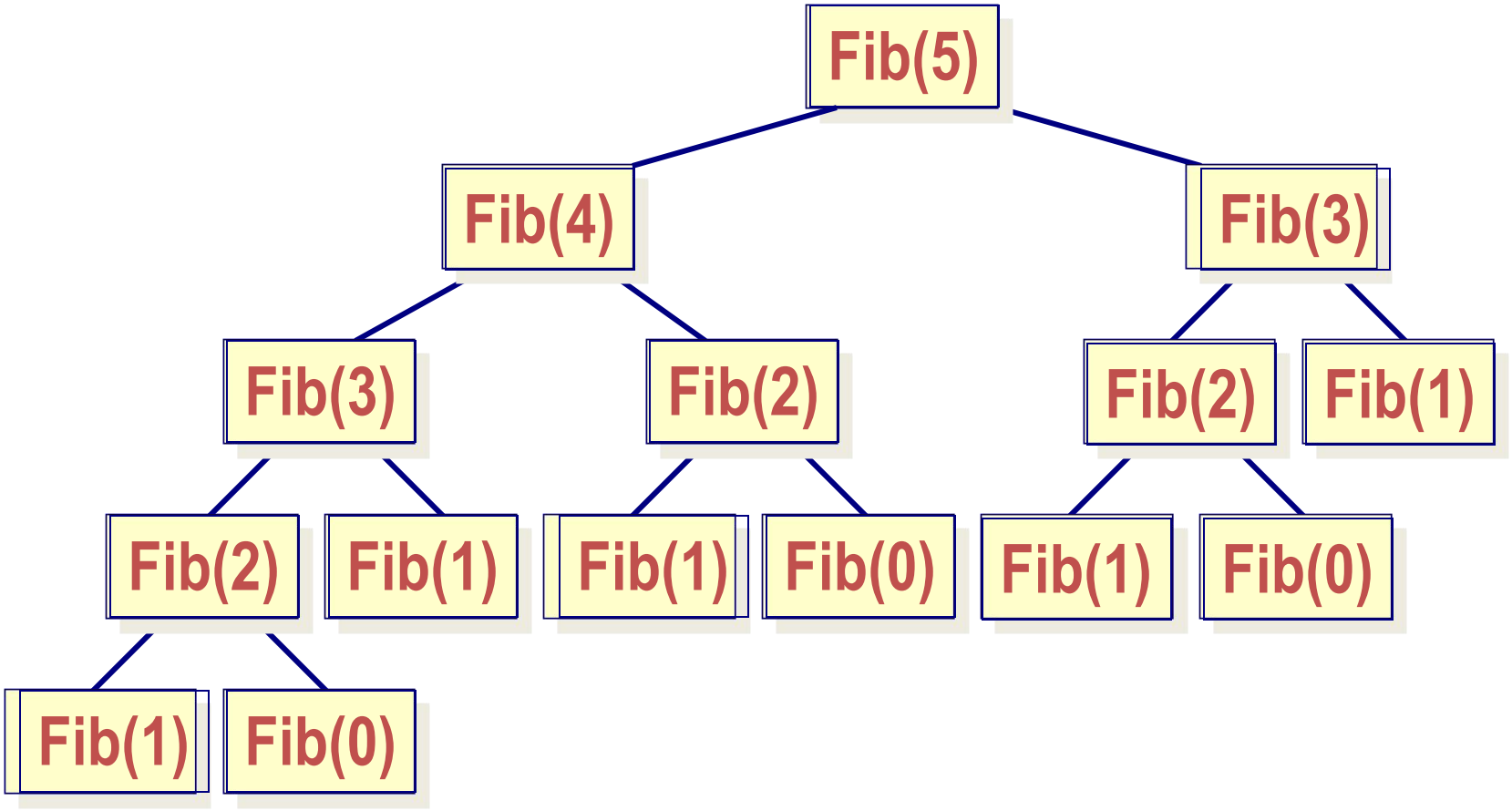
计算斐波那契数列的函数Fib(n)的定义

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

如 $\mathbf{F_0=0, F_1=1, F_2=1, F_3=2, F_4=3, F_5=5}$

求解斐波那契数列的递归算法

```
long Fib(long n) {
    if (n <= 1) return n;
    else return Fib(n-1)+Fib(n-2);
}
```



斐波那契数列的递归调用树

调用次数 **NumCall(k) = 2*Fib(k+1)−1**

```
struct Node{
long n;     //记忆走过的n
int tag;    //区分左右
};

 long Fabnacci(long n)
 {
     Stack<Node> S; Node * w; long sum = 0;
     do{
         while(n>1){w->n=n;w->tag=1;
         S.push(w);n--;}//左侧进栈
     sum = sum + n;  //n=1或0
```

```
        while(S.IsEmpty() == false){
            S.Pop(w);
            if(w->tag == 1)
            {w->tag=2;S.push(w);
            n=w->n-2;break;}//右侧进栈
        }
    }while(S.IsEmpty()==false);
    return sum;
}
```

单向递归用迭代法实现

```
long FibIter(long n) {
    if (n <= 1) return n;
    long twoback = 0,  oneback = 1,  Current;
    for (int i = 2; i <= n; i++) {
        Current = twoback + oneback;
        twoback = oneback;   oneback = Current;
    }
    return Current;
}
```

单向递归用迭代法实现

```
long FibIter(long n) {
    if (n <= 1) return n;
    long * F= new long [n+1]; F[0] = 0; F[1] = 1;
    for (int i = 2; i <= n; i++) {
        F[i]= F[i-2] + F[i-1];
        twoback = oneback;  oneback = Current;
    }
    return F[n-1];
}
```

尾递归用迭代法实现

25	36	72	18	99	49	54	63

```
void recfunc(int A[ ],  int n) {
    if (n >= 0) {
        cout << A[n] << “  ”;
        n--;
        recfunc(A, n);
    }
}
```

```
void sterfunc(int A[ ], int n) {
//消除了尾递归的非递归函数
    while (n >= 0) {
        cout << "value    " << A[n] << endl;
        n--;
    }
}
```

函数 `f(n)` 在 `n<=1` 时直接返回，否则调用 `f(n-3)`。从 `f(11)` 开始，最大同时存在多少个活动记录？

分析下面函数在 `printDown(4)` 时的输出，并说明输出发生在递归调用之前还是之后。

```
void printDown(int n) {
    if (n <= 0) return;
    printDown(n - 2);
    cout << n << " ";
}
```

将下列尾递归改为循环。

```
void visit(int a[], int i) {
    if (i < 0) return;
    cout << a[i] << " ";
    visit(a, i - 1);
}
```

函数 `f(n)` 在 `n<=1` 时直接返回，否则调用 `f(n-3)`。从 `f(11)` 开始，最大同时存在多少个活动记录？

5 个，对应参数 `11、8、5、2、-1`。计数包括基例调用。

分析下面函数在 `printDown(4)` 时的输出，并说明输出发生在递归调用之前还是之后。

```
void printDown(int n) {
    if (n <= 0) return;
    printDown(n - 2);
    cout << n << " ";
}
```

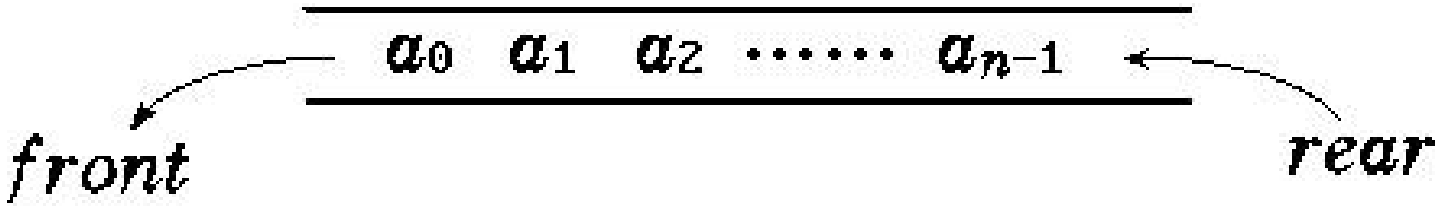
输出 `2 4`。输出语句位于递归调用之后，因此在返回阶段执行。

将下列尾递归改为循环。

```
void visit(int a[], int i) {
    if (i < 0) return;
    cout << a[i] << " ";
    visit(a, i - 1);
}
```

```
void visit(int a[], int i) {
    while (i >= 0) {
        cout << a[i] << " ";
        --i;
    }
}
```

§3.3 队列（queue）



- 定义
  - 队列是只允许在一端删除，在另一端插入的线性表
  - 允许删除的一端叫做队头(*front*)，允许插入的一端叫做队尾(*rear*)。
- 特性
  - 先进先出(*FIFO, First In First Out*)

队列的抽象数据类型

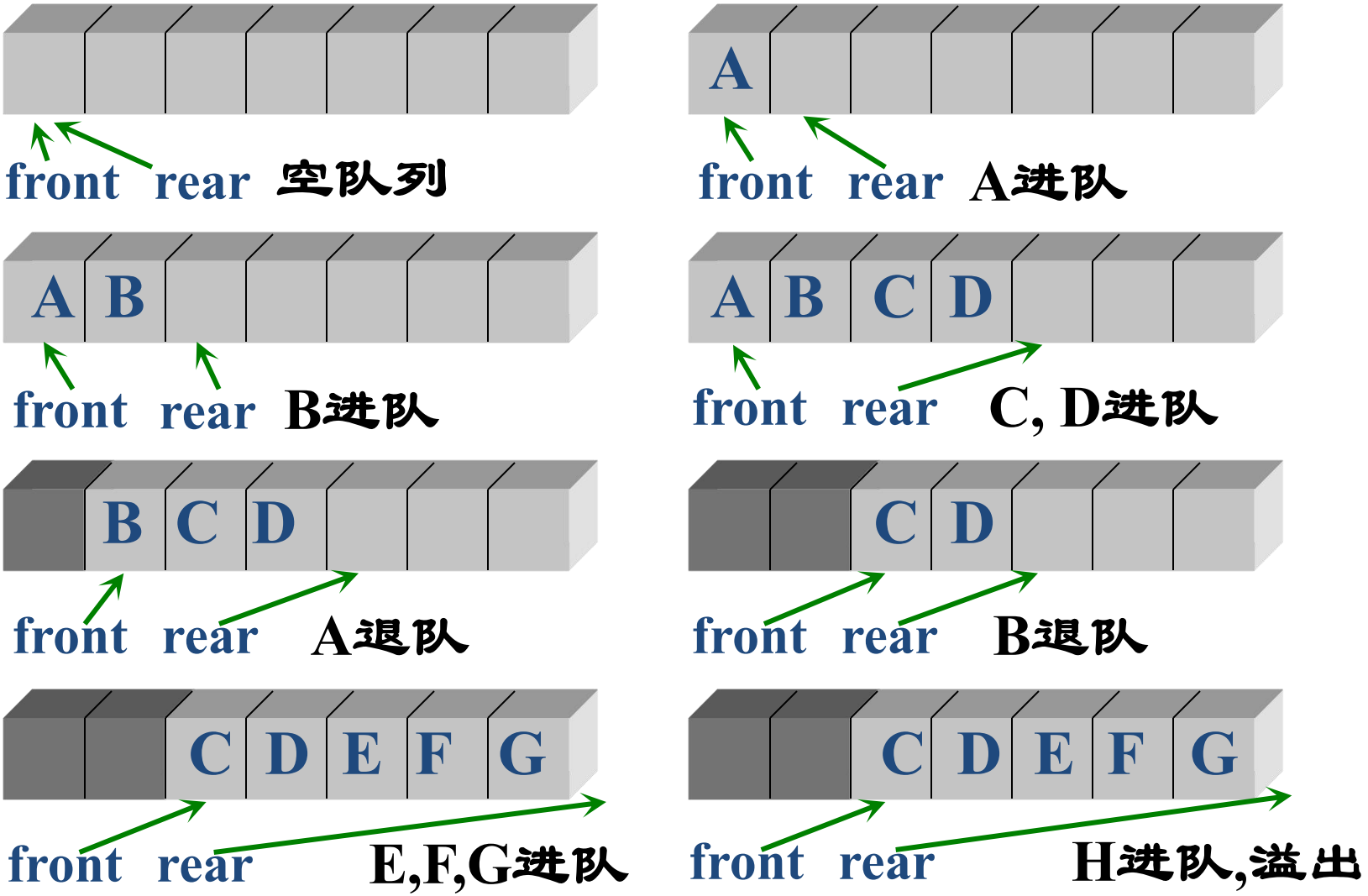
```
template <class E>
class Queue {
public:
    Queue() { };                    //构造函数
    ~Queue() { };                   //析构函数
    virtual bool EnQueue(E x) = 0;          //进队列
    virtual bool DeQueue(E& x) = 0;         //出队列
    virtual bool getFront(E& x) = 0;        //取队头
    virtual bool IsEmpty() const = 0;       //判队列空
    virtual bool IsFull() const = 0;        //判队列满
};
```

队列的数组存储表示——顺序队列

```
#include <assert.h>
#include <iostream.h>
#include "Queue.h"
template <class E>
class SeqQueue : public Queue<E> {    //队列类定义
protected:
    int rear, front;                  //队尾与队头指针
    E *elements;                      //队列存放数组
    int maxSize;                      //队列最大容量
public:
   SeqQueue(int sz = 10);    //构造函数
```

```
    ~SeqQueue() { delete[ ] elements; }  //析构函数
    bool EnQueue(E x);          //新元素进队列
    bool DeQueue(E& x);         //退出队头元素
    bool getFront(E& x);        //取队头元素值
    void makeEmpty() { front = rear = 0; }
    bool IsEmpty() const { return front == rear; }
    bool IsFull() const
        { return rear ==maxSize; }
    int getSize() const
        { return rear-front; }
};
```

队列的进队和出队（数组方式）

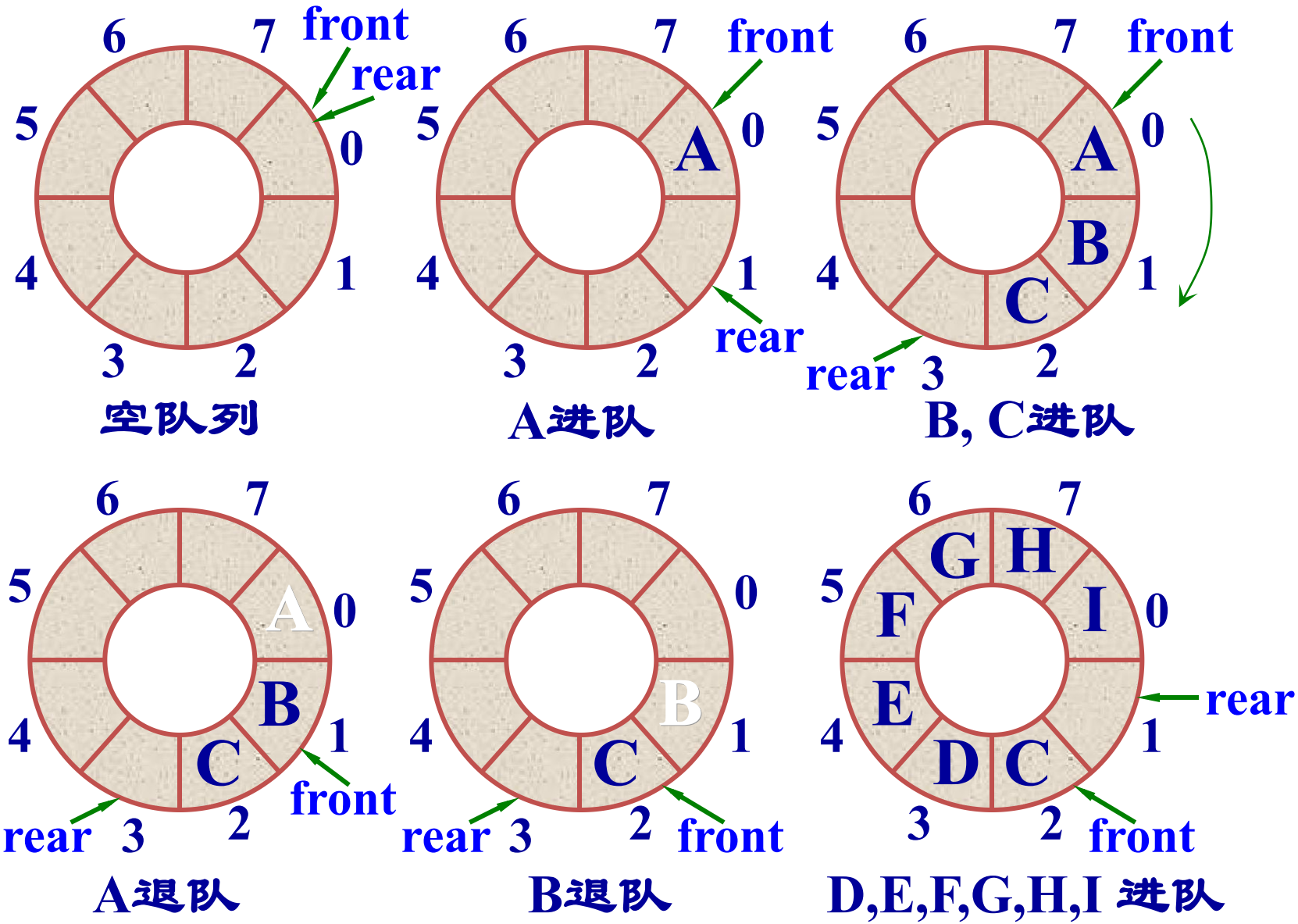


队列的进队和出队

- 进队:新元素在*rear*处加入，*rear* = *rear* + 1。
- 出队:取出下标为 *front* 的元素，*front* = *front* + 1
- 队空时 *rear* == *front*
- 队满时 *rear* == *maxSize* （假溢出）
- 解决假溢出的办法之一：将队列元素存放数组首尾相接，形成循环（环形）队列。

循环队列 (Circular Queue)

- 队列存放数组被当作首尾相连的表处理。
- 队头、队尾指针加**1**时从**maxSize-1**直接进到**0**, 可用语言的取模(余数)运算实现。
- 队头指针进**1**：front = (front+1) % maxSize;
- 队尾指针进**1**：rear = (rear+1) % maxSize;
- 队列初始化：front = rear = 0;
- 队空条件：front == rear;
- 队满条件：(rear+1) % maxSize == front


6
7
front
rear
5
0
4
1
3
2
空队列
A
A进队
B
C
B, C进队
A退队
B
B退队
G
H
F
I
E
D
C
D,E,F,G,H,I 进队

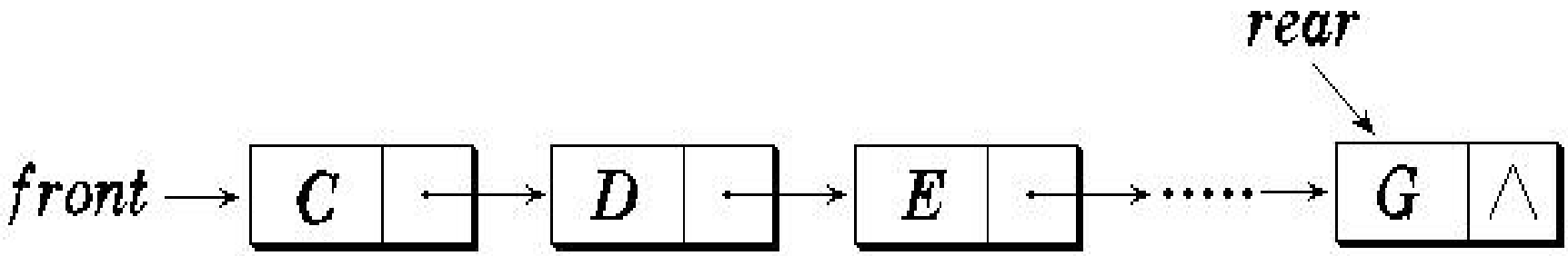
循环队列操作的定义

```
void MakeEmpty() { front = rear = 0; }
int IsEmpty() const { return front == rear; }
int IsFull() const
    { return (rear+1) % maxSize == front; }

template <class E>
SeqQueue<E>::SeqQueue(int sz)
    : front(0), rear(0), maxSize(sz) {     //构造函数
      elements = new E[maxSize];
      assert ( elements != NULL );
};
```

```
template <class E>
bool SeqQueue<E>::EnQueue(E x) {
//若队列不满, 则将x插入到该队列队尾, 否则返回
  if (IsFull() == true) return false;
     elements[rear] = x;                    //先存入
     rear = (rear+1) % maxSize;             //尾指针加一
     return true;
};
```

```
template <class E>
bool SeqQueue<E>::DeQueue(E& x) {
//若队列不空则函数退队头元素并返回其值
    if (IsEmpty() == true) return false;
    x = elements[front];                  //先取队头
    front = (front+1) % maxSize;   //再队头指针加一
    return true;
};

template <class E>
bool SeqQueue<E>::getFront(E& x) const {
//若队列不空则函数返回该队列队头元素的值
    if (IsEmpty() == true) return false;    //队列空
    x = elements[front];                  //返回队头元素
    return true;
};
```

队列的链接表示 — 链式队列



- 队头在链头，队尾在链尾。
- 链式队列在进队时无队满问题，但有队空问题。
- 队空条件为 ***front == NULL***

链式队列类的定义

```
#include <iostream.h>
#include "Queue.h"
template <class E>
struct QueueNode {                          //队列结点类定义
private:
    E data;                                 //队列结点数据
    QueueNode<E> *link;                     //结点链指针
public:
    QueueNode(E x = 0, QueueNode<E>
      *next = NULL) : data(x), link(next) { }
};
```

```
template <class E>
class LinkedQueue {
private:
    QueueNode<E> *front, *rear;   //队头、队尾指针
public:
   LinkedQueue() : rear(NULL), front(NULL) { }
   ~LinkedQueue();
   bool EnQueue(E x);
   bool DeQueue(E& x);
   bool getFront(E& x);
   void makeEmpty();          //实现与~Queue()同
   bool IsEmpty() const { return front == NULL; }
};
```

```
template <class E>
LinkedQueue<E>::~LinkedQueue() {      //析构函数
    QueueNode<E> *p;
    while (front != NULL) {           //逐个结点释放
        p = front;  front = front->link;  delete p;
    }
};
```

```
template <class E>
bool LinkedQueue<E>::EnQueue(E x) {
//将新元素x插入到队列的队尾

    if (front == NULL)  {              //创建第一个结点
        front = rear = new QueueNode<E> (x);
        if (front == NULL) return false; }      //分配失败
    else {                                  //队列不空, 插入
        rear->link = new QueueNode<E> (x);
        if (rear->link == NULL) return false;
        rear = rear->link;
    }
    return true;
};
```

```
template <class E>
//如果队列不空，将队头结点从链式队列中删去
bool LinkedQueue<E>::DeQueue(E& x) {
    if (IsEmpty() == true) return false;        //判队空
    QueueNode<E> *p = front;
    x = front->data;  front = front->link;
    delete p;   return true;
};

template <class E>
bool LinkedQueue<E>::getFront(E& x) {
//若队列不空，则函数返回队头元素的值
    if (IsEmpty() == true) return false;
    x = front->data;  return true;
};
```

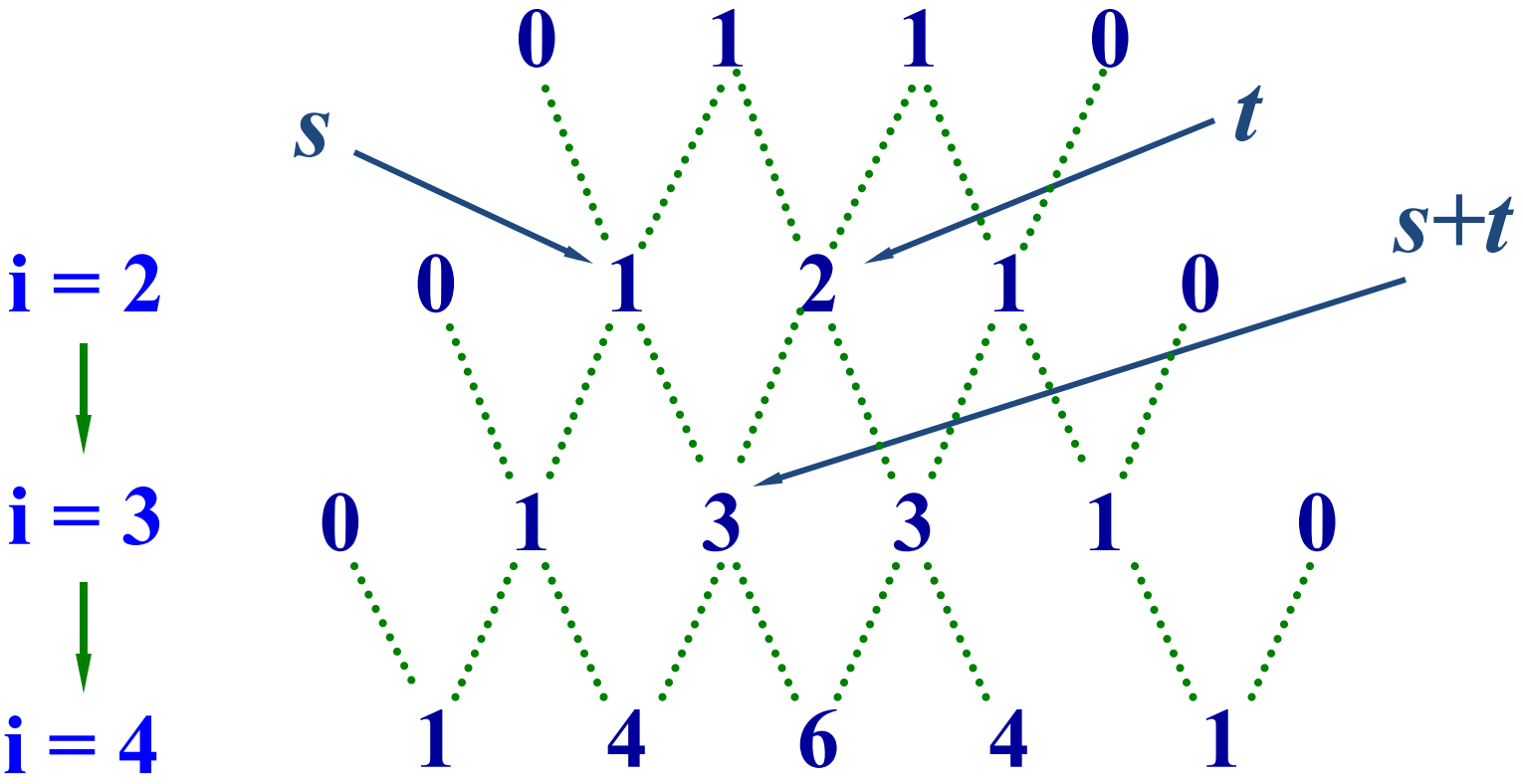
队列的应用：

逐行打印二项展开式 $(a+b)^i$ 的系数

杨辉三角形 (Pascal's triangle)

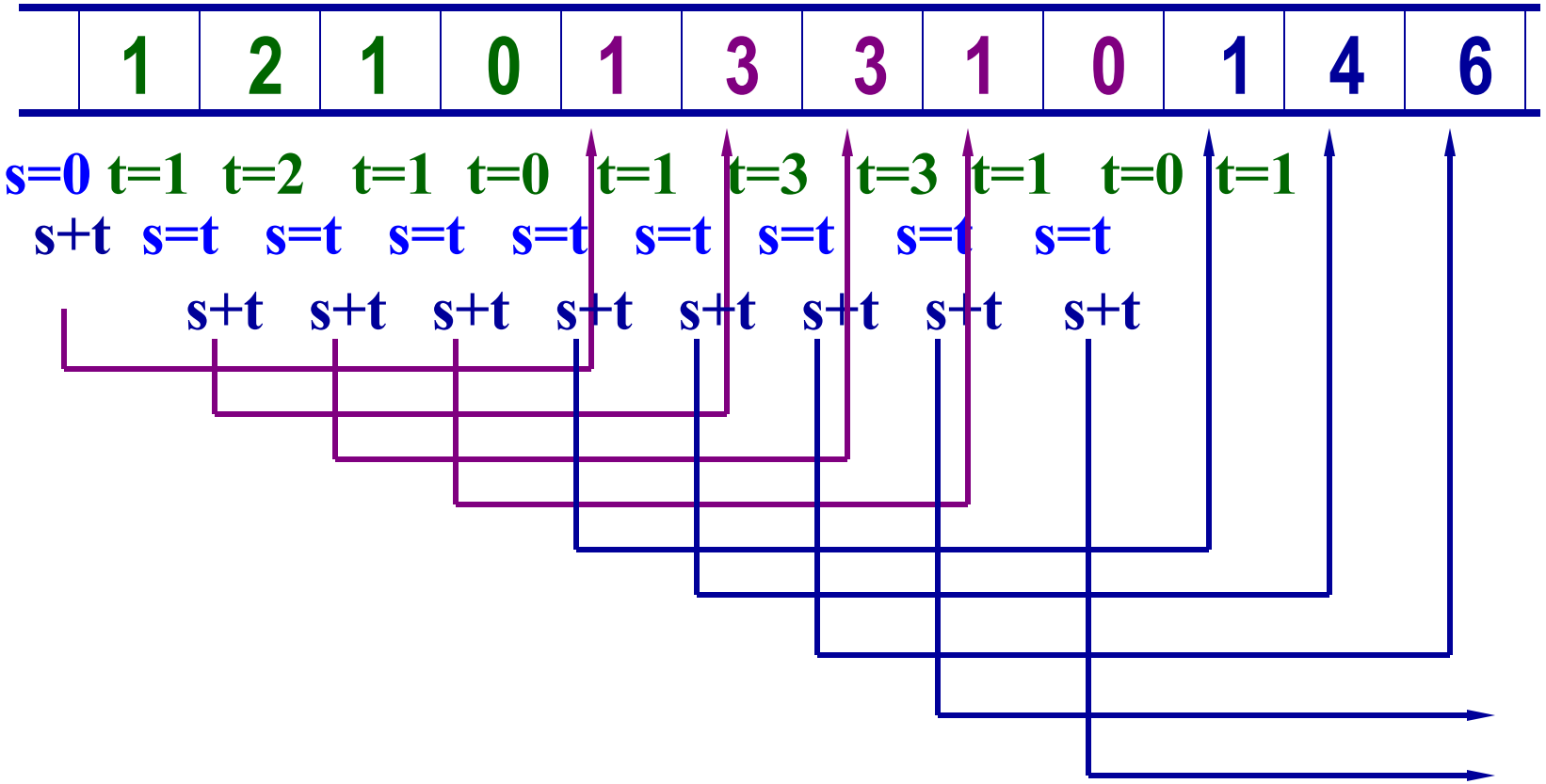
				1		1						$i=1$
			1		2		1					2
		1		3		3		1				3
	1		4		6		4		1			4
1		5		10		10		5		1		5
	6		15		20		15		6		1	6

分析第 i 行元素与第 i+1行元素的关系



从前一行的数据可以计算下一行的数据

从第 i 行数据计算并存放第 i+1 行数据



利用队列打印二项展开式系数的算法

```
#include <stdio.h>
#include <iostream.h>
#include "queue.h"
void YANGHVI(int n) {
    Queue q(n+3);                    //队列初始化
    q.makeEmpty();
    q.EnQueue(1);  q.EnQueue(1);
    int s = 0, t;
```

```
    for (int i = 1; i <= n; i++) {          //逐行计算
        cout << endl;
        q.EnQueue(0);
        for (int j = 1; j <= i+2; j++) {    //下一行
            q.DeQueue(t);
            q.EnQueue(s + t);
            s = t;
            if (j != i+2) cout << s << ' ';
        }
    }
}
```

§3.4 优先级队列（priority queue）

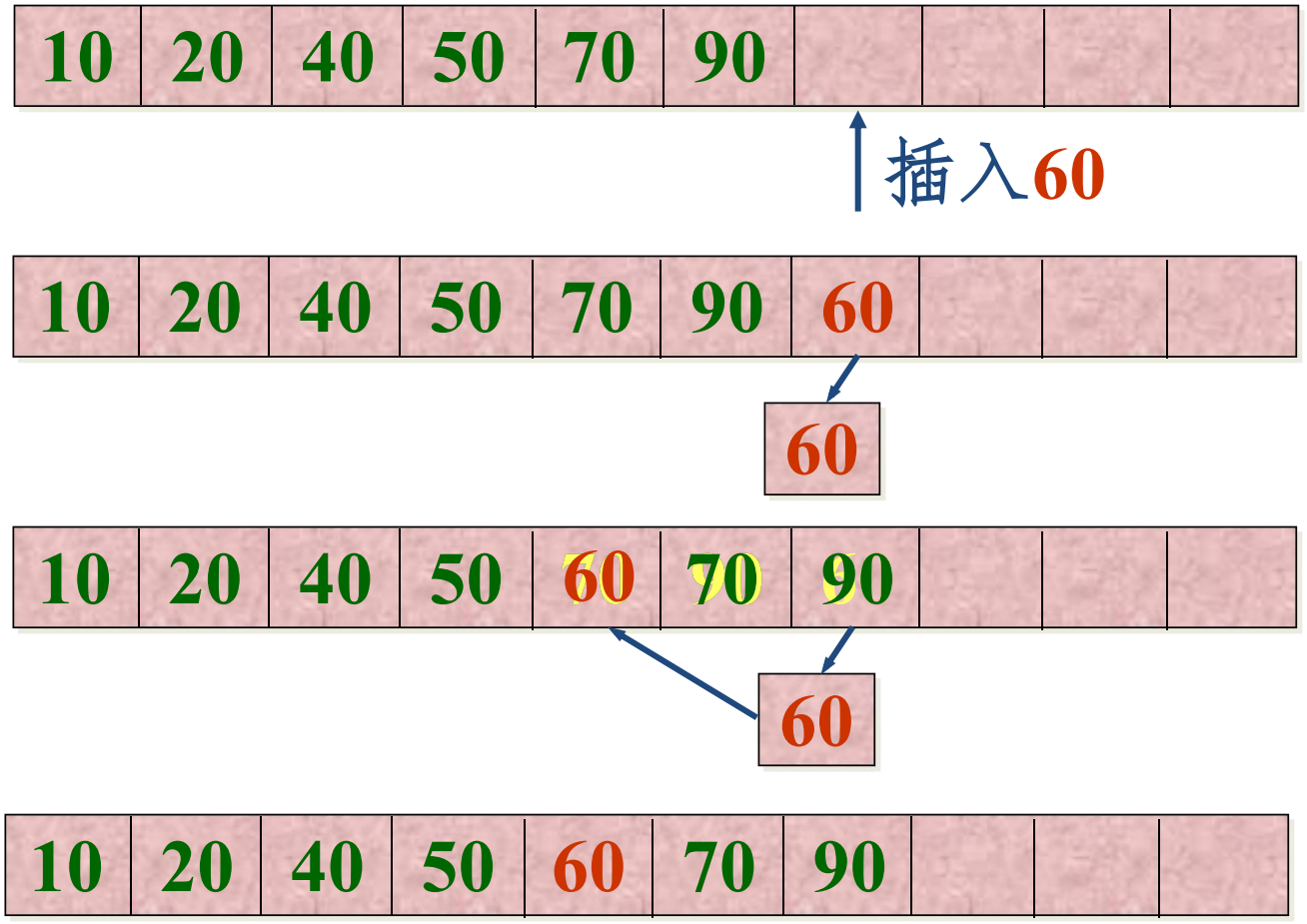
任务编号	1	2	3	4	5
优先权	20	0	40	30	10
执行顺序	3	1	5	4	2

数字越小，优先权越高

- 优先级队列： 是不同于先进先出队列的另一种队列。每次从队列中取出的是具有最高优先权(优先级)的元素

- 优先权是根据问题而定的

- 出现相同的优先权的元素时，按***FIFO***的方式处理


10 20 40 50 70 90
插入60
10 20 40 50 70 90 60
60
10 20 40 50 60 70 90
60
10 20 40 50 60 70 90

优先级队列的类定义

```
#include <assert.h>
#include <iostream.h>
#include <stdlib.h>

template <class E>
class PQueue {
private:
    E *pqelements;          //存放数组
    int count;              //队列元素计数
    int maxPQSize;          //最大元素个数
    void adjust();          //调整
```

```
public:
    PQueue(int sz = 50);
   ~PQueue() { delete [ ] pqelements; }
    bool Insert(E x);
    bool removeMin(E& x);
    bool getFront(E& x);
    void makeEmpty() { count = 0; }
    bool IsEmpty() const { return count == 0; }
    bool IsFull() const
          { return count == maxPQSize; }
    int Length() const { return count; }
};
```

优先级队列部分成员函数的实现

```
template <class E>
PQueue<E>::PQueue(int sz) {
    maxPQSize = sz;  count = 0;
    pqelements = new E[maxPQSize];
    assert (pqelements != NULL);
}

template <class E>
bool PQueue<E>::Insert(E x) {
    if (IsFull() == true) return false;   //判队满断言
    pqelements[count++] = x;              //插入
    adjust();   return true;
}
```

```
template <class E>
void PQueue<E>::adjust() {
    E temp = pqelements[count-1];
    //将最后元素暂存再从后向前找插入位置
    for (int j = count-2;  j >= 0;  j--)
        if (pqelements[j] <= temp) break;
        else pqelements[j+1] = pqelements[j];
    pqelements[j+1] = temp;
}
```

```
template <class E>
bool PQueue<E>::removeMin(E& x) {
    if (IsEmpty()) return false;
    x = pqelements[0];        //取出0号元素
    for (int i = 1; i < count; i++)
        pqelements[i-1] = pqelements[i];
        //从后向前逐个移动元素填补空位
    count--;
    return true;
}
```

```
template <class E>
bool PQueue<E>::getFront (E& x) {
    if (IsEmpty() == true) return false;
    x = pqelements[0];
    return true;
}
```

循环队列数组容量 `M=7`，采用牺牲一个单元的方案。初始 `front=rear=4`，依次入队 `A、B、C`，出队两次，再入队 `D、E、F`。求最终 `front`、`rear` 及队头到队尾的元素序列。

链式队列只有一个结点 `K`。执行出队后，只写 `front=front->link` 而没有修改 `rear`。程序留下了什么隐患？

任务按 `(编号, 优先级)` 依次到达：`(J1,2)、(J2,1)、(J3,2)、(J4,3)、(J5,1)`。规定数字越小优先级越高，同优先级按到达顺序处理。连续删除全部任务时的顺序是什么？

循环队列数组容量 `M=7`，采用牺牲一个单元的方案。初始 `front=rear=4`，依次入队 `A、B、C`，出队两次，再入队 `D、E、F`。求最终 `front`、`rear` 及队头到队尾的元素序列。

答案： 最终 `front=6`，`rear=3`；队列元素依次为 `C、D、E、F`。

链式队列只有一个结点 `K`。执行出队后，只写 `front=front->link` 而没有修改 `rear`。程序留下了什么隐患？

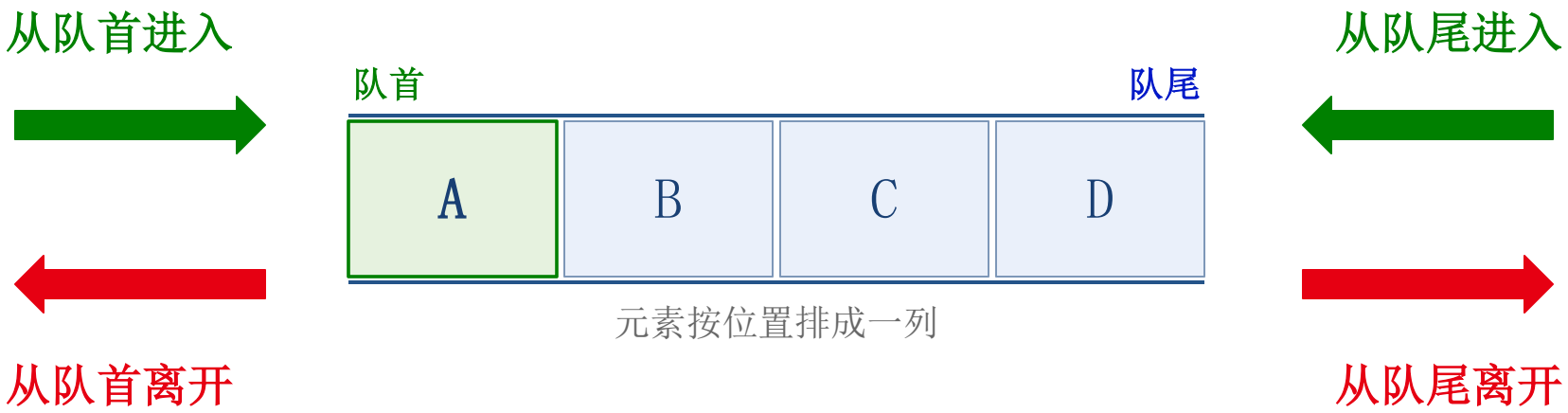
答案： `front` 变为 `NULL`，但 `rear` 仍指向已经释放的结点，形成悬空指针。应在出队后检测 `front==NULL`，并令 `rear=NULL`。

任务按 `(编号, 优先级)` 依次到达：`(J1,2)、(J2,1)、(J3,2)、(J4,3)、(J5,1)`。规定数字越小优先级越高，同优先级按到达顺序处理。连续删除全部任务时的顺序是什么？

答案： `J2、J5、J1、J3、J4`。

双端队列：两端都能进出

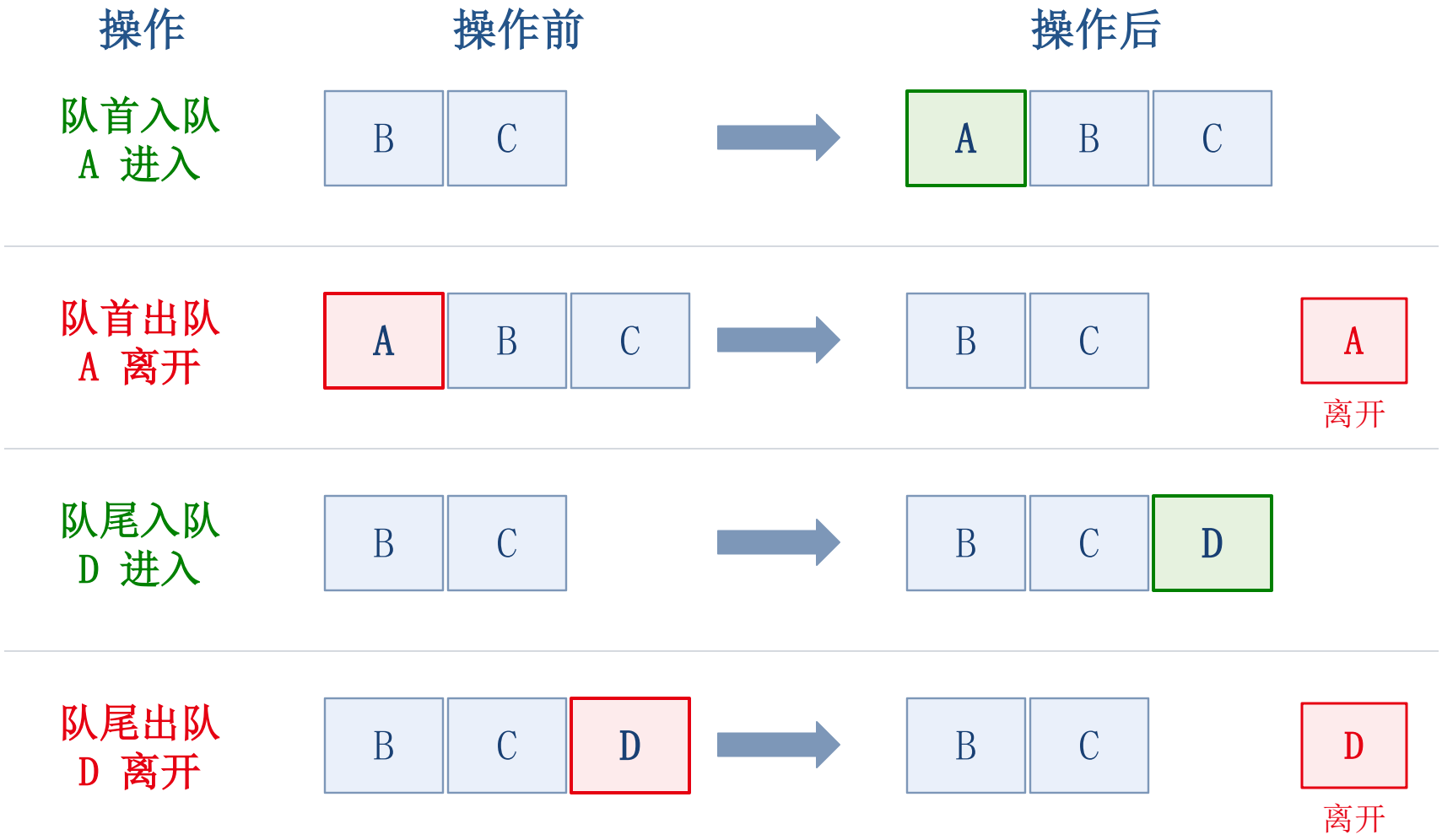
把双端队列想象成一条两头都开门的通道。



普通队列固定从一端进入、另一端离开。

双端队列允许队首和队尾各自进入或离开。

双端队列的四种基本操作



“双端”描述的是访问位置，不表示元素按大小排列。

双端队列与单调队列

单调队列使用双端队列保存候选下标，两端承担不同职责。

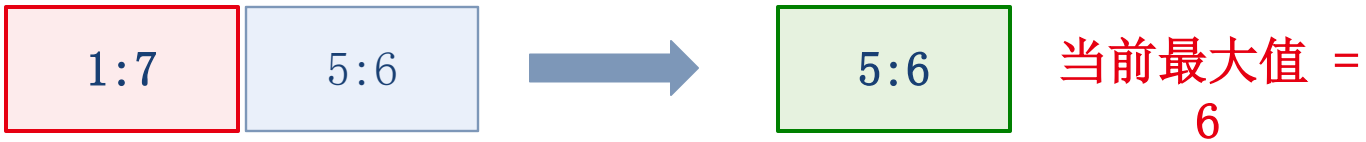
候选写法：下标:数值

① 队尾维护候选

3:5 和 4:2 的值更小，并且更早离开窗口，因此失去竞争力。

② 队首删除过期候选

窗口左边界越过下标 1



队尾保持候选值的单调性，队首给出当前最优值并清理过期下标。

单调队列

单调队列通常用双端队列实现：队首给出当前最优值，队尾负责维护单调性。

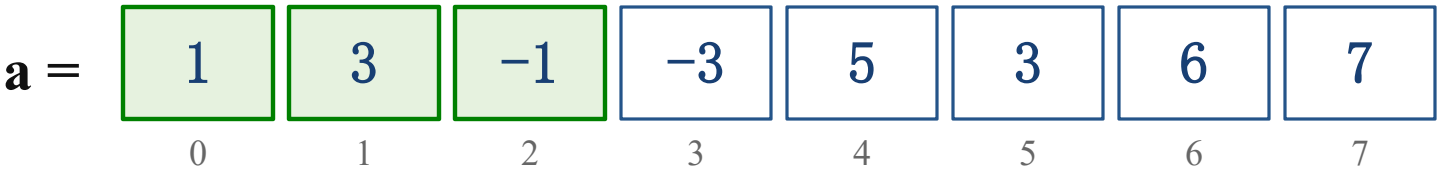
比较	单调栈	单调队列
容器操作	一端压入、弹出	队首和队尾均可弹出
典型问题	最近更大或更小边界	滑动区间最大值或最小值
候选失效	新元素破坏栈顶单调性	队尾被支配，队首过期

最大值队列的不变量

下标从队首到队尾递增；对应数值单调不增；队首就是当前窗口最大值。

仍然保存下标，因为窗口移动时必须判断候选是否过期。

滑动窗口最大值



窗口长度 k = 3。每移动一步，输出当前窗口的最大值。

窗口下标	[0,2]	[1,3]	[2,4]	[3,5]	[4,6]	[5,7]
窗口最大值	**3**	**3**	**5**	**5**	**6**	**7**

逐个扫描窗口

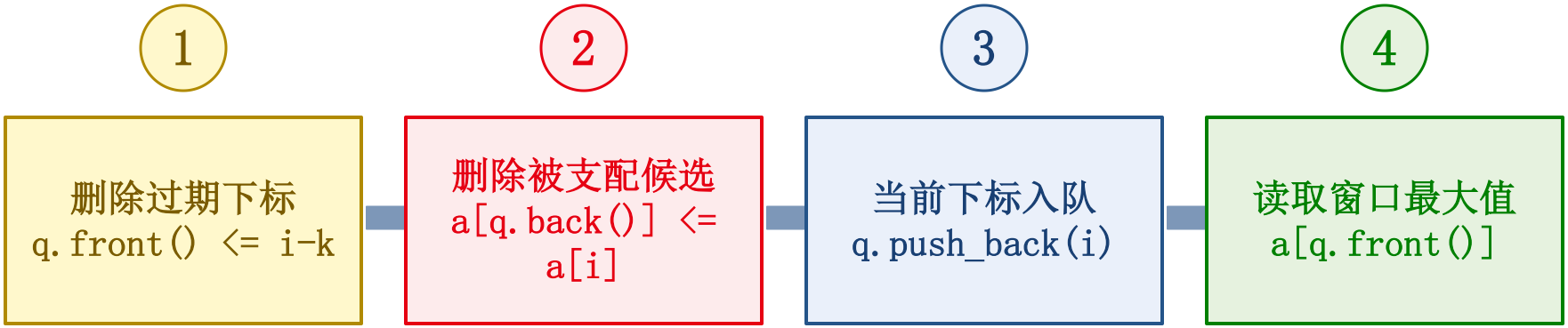
每个窗口检查 k 个元素
时间复杂度 O(nk)

维护候选队列

每个下标至多进、出队列一次
时间复杂度 O(n)

目标输出：[3, 3, 5, 5, 6, 7]

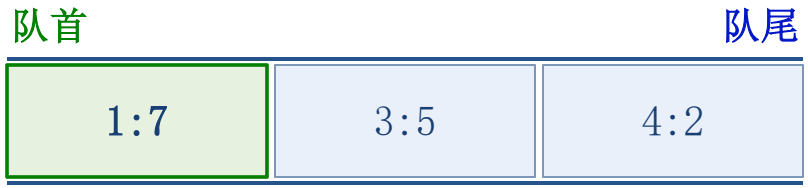
最大值队列的维护步骤



操作顺序　先移动左边界并清理队首，再用新元素清理队尾

形成完整窗口后，队首下标一定属于当前窗口

队列内部始终满足



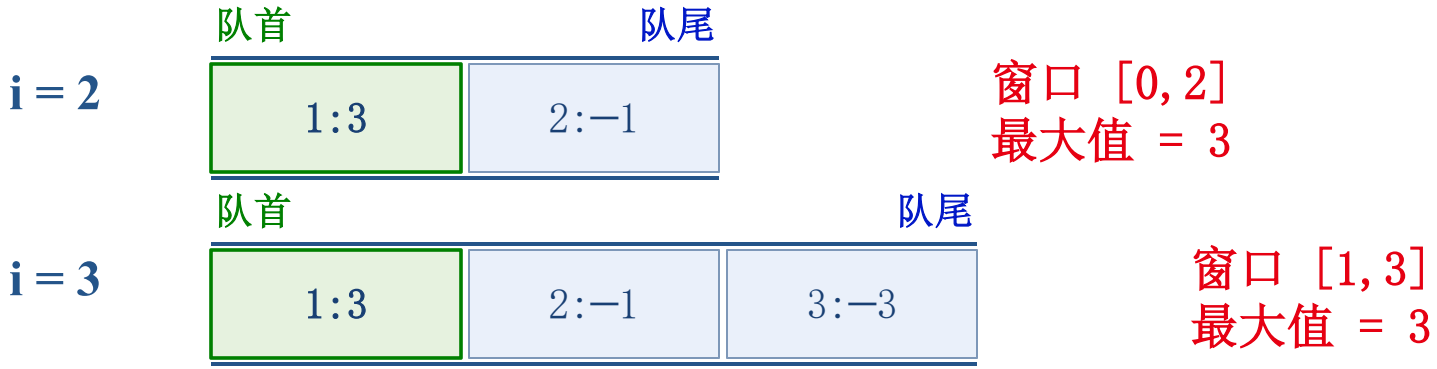
队首最大

下标递增，数值不增

新元素比队尾大或相等时，旧队尾的值不大于新元素，并且更早过期。

单调队列过程：形成前两个窗口

i	a[i]	队首过期	队尾删除	处理后的队列（队首→队尾）	输出
0	1	无	无	0:1	—
1	3	无	0:1	1:3	—
2	−1	无	无	1:3，2:−1	3
3	−3	无	无	1:3，2:−1，3:−3	3



单调队列过程：完成其余窗口

i	a[i]	队首过期	队尾删除	处理后的队列	输出
4	5	1:3	3:−3，2:−1	4:5	**5**
5	3	无	无	4:5，5:3	**5**
6	6	无	5:3，4:5	6:6	**6**
7	7	无	6:6	7:7	**7**

关键步骤（i = 4）　新窗口为 [2, 4]，下标 1 先从队首过期；数值 5 再从队尾删除 −3 和 −1。

最终结果　**[3, 3, 5, 5, 6, 7]**

队列中从不保留已过期或已被更新元素支配的下标。

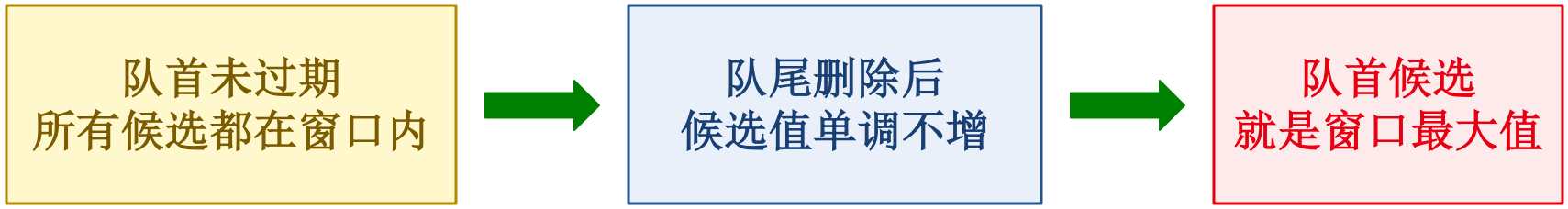
滑动窗口最大值：C++ 模板

```
vector<int> maxSlidingWindow(const vector<int>& a, int k) {
    if (k <= 0 || k > (int)a.size()) return {};
    vector<int> ans;
    deque<int> q;                // 保存下标
    for (int i = 0; i < (int)a.size(); ++i) {
        while (!q.empty() && q.front() <= i - k)
            q.pop_front();       // 队首过期
        while (!q.empty() && a[q.back()] <= a[i])
            q.pop_back();        // 队尾被支配
        q.push_back(i);
        if (i >= k - 1) ans.push_back(a[q.front()]);
    }
    return ans;
}
```

q.front() 控制窗口左边界；q.back() 维护候选值的单调性。

使用 <= 会删除较早出现的相等元素。

正确性与均摊复杂度



为什么可以删除队尾 j

若 a[i] ⩾ a[j] 且 i > j，则 a[i] 不小于 a[j]，并且 i 更晚离开窗口。j 不可能再次成为最大值。

操作计数　每个下标入队一次，之后只会从队首或队尾离开一次。全部队列操作为 O(n)。

时间复杂度 O(n)　　**空间复杂度 O(k)**

双重 while 不会让同一个下标被重复删除。

重复元素与常见变体

目标	队尾删除条件	相等元素处理	队首含义
窗口最大值	a[q.back()] <= a[i]	删除较早的相等值	最大值
窗口最大值	a[q.back()] < a[i]	保留所有相等值	最早的最大值
窗口最小值	a[q.back()] >= a[i]	删除较早的相等值	最小值
窗口最小值	a[q.back()] > a[i]	保留所有相等值	最早的最小值

重复元素示例：[5, 5, 5]，k = 2

使用 <=：队列只保留较新的 5

使用 <：保留窗口内的两个 5

两种写法都输出 5，但队列内容不同。
若题目要求最早位置，需要保留相等元素。

扩展问题：和至少为 K 的最短子数组

数组含负数时，普通双指针无法保证窗口和随边界单调变化。

前缀和 $P[j] - P[i] \geq K$ 寻找最小的 j − i

原数组 a	—	2	-1	2
前缀下标	0	1	2	3
前缀和 P	**0**	**2**	**1**	**3**

单调队列保存前缀下标，保证 **P[q.front()] < … < P[q.back()]**。

若 P[j]−P[q.front()] ≥ K，记录长度并弹出队首，继续寻找更短答案

若 P[q.back()] ≥ P[j]，弹出队尾。新前缀和不大于旧值，且下标更大

示例 [2, −1, 2]，K = 3，最短长度为 3。

最短子数组：单调队列实现

```
int shortestSubarray(const vector<int>& a, long long K) {
    int n = a.size(), ans = n + 1;
    vector<long long> p(n + 1, 0);
    for (int i = 0; i < n; ++i) p[i + 1] = p[i] + a[i];
    deque<int> q;
    for (int j = 0; j <= n; ++j) {
        while (!q.empty() && p[j] - p[q.front()] >= K) {
            ans = min(ans, j - q.front());
            q.pop_front();
        }
        while (!q.empty() && p[q.back()] >= p[j])
            q.pop_back();
        q.push_back(j);
    }
    return ans <= n ? ans : -1;
}
```

识别信号

窗口持续右移

反复查询区间最值

队首候选会过期

队尾候选会被支配

目标复杂度为 O(n)

队首处理边界
队尾维护单调性

单调队列把区间内的全部元素压缩为仍有机会成为答案的候选下标。