



第三章 栈和队列

本章内容：

3.1 栈

3.2 栈与递归

3.3 队列

3.4 优先级队列



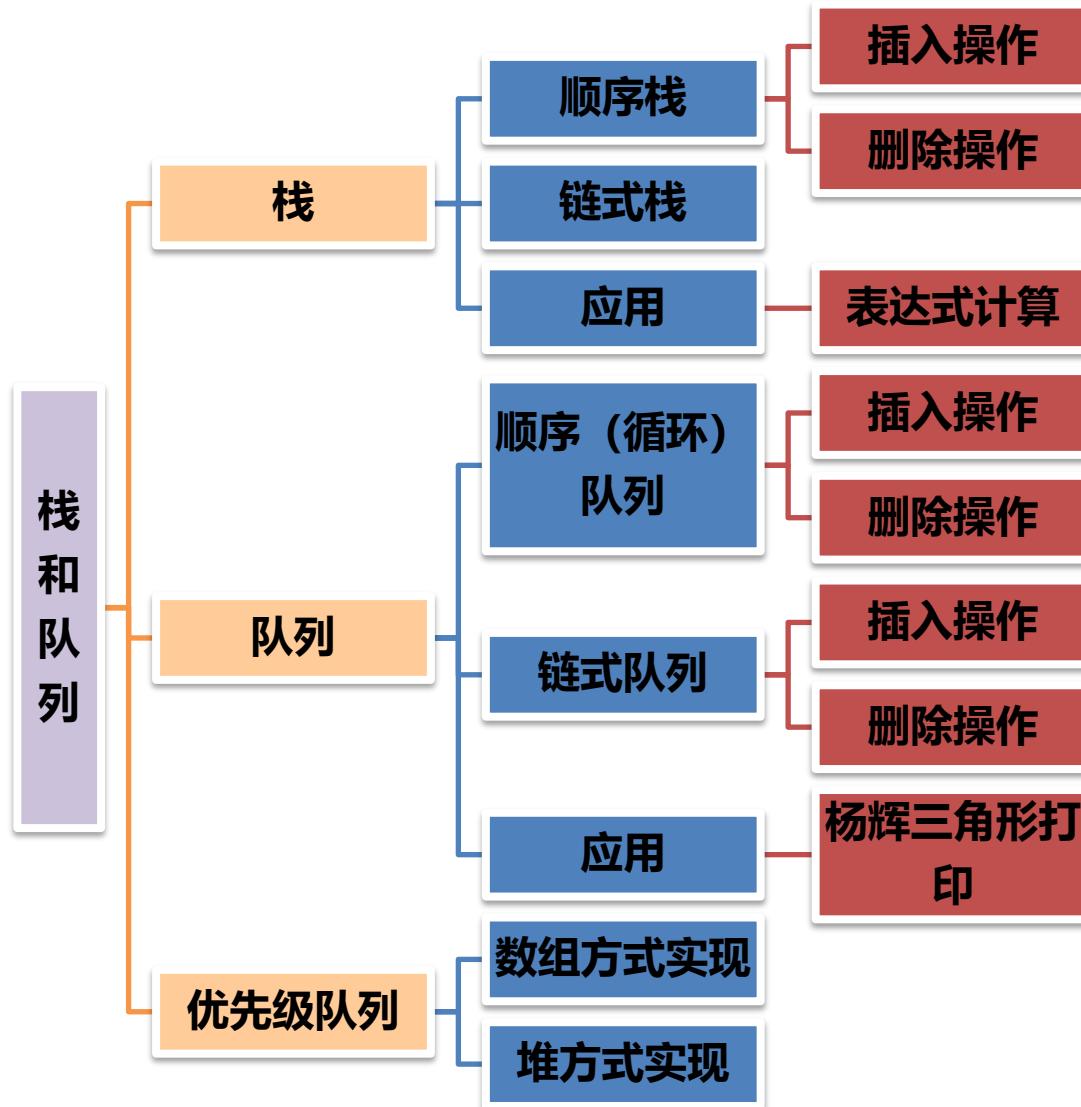
第三章 栈和队列

栈
队列
优先级队列 } → 线性结构

与线性表不同的是，它们都是
限制存取位置的线性结构



知识导图





第三章 栈和队列

3.1 栈

3.2 栈与递归

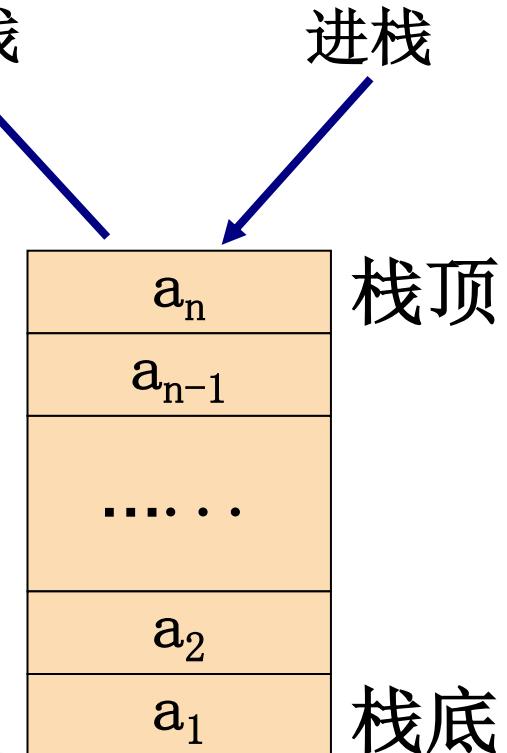
3.3 队列

3.4 优先级队列



栈 (stack)

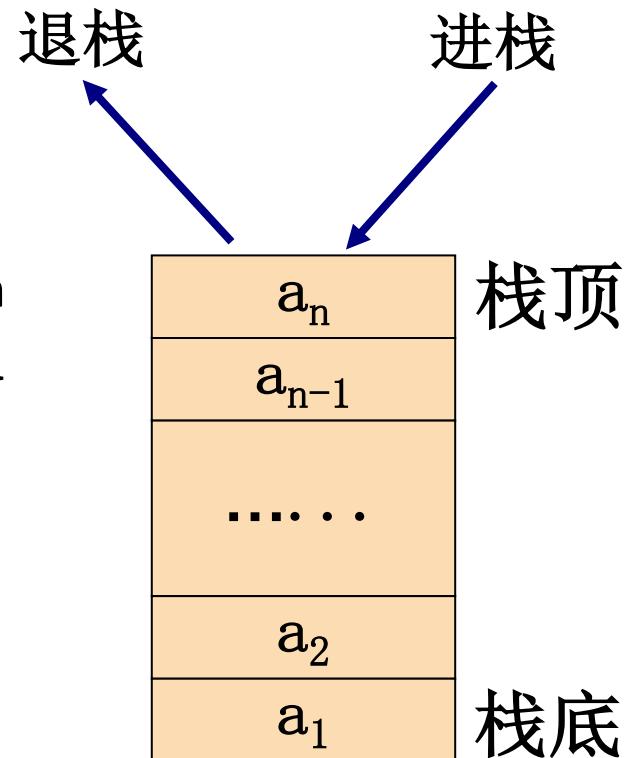
- 只允许在一端插入和删除的线性表
- 允许插入和删除的一端称为**栈顶**(*top*)，另一端称为**栈底**(*bottom*)
- 当表中没有元素时称为空栈。
- 进栈 (Push)：在栈顶位置插入元素的操作。
- 出栈 (Pop)：删除栈顶元素的操作





栈

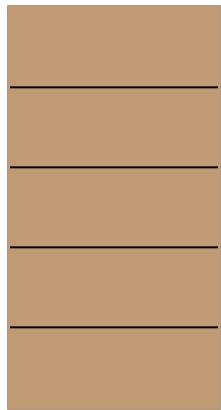
- 假设栈 $S = (a_1, a_2, a_3, \dots, a_n)$,
则 a_1 称为栈底元素, a_n 为栈顶元素。
- 栈中元素按 $a_1, a_2, a_3, \dots, a_n$ 的次序进栈, 退栈的第一个元素应为栈顶元素。
- 特点: 后进先出 (LIFO, Last In First Out) 或者先进后出 (FILO, First In Last Out)。



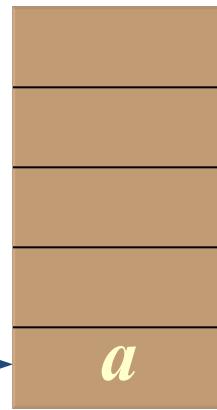


进栈示例

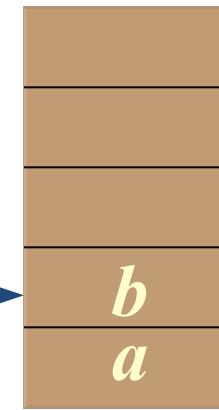
top → 空栈



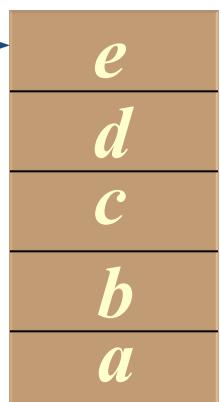
top → **a** 进栈



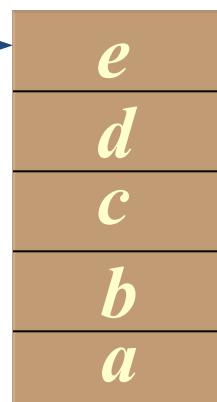
top → **b** 进栈



top → **e** 进栈

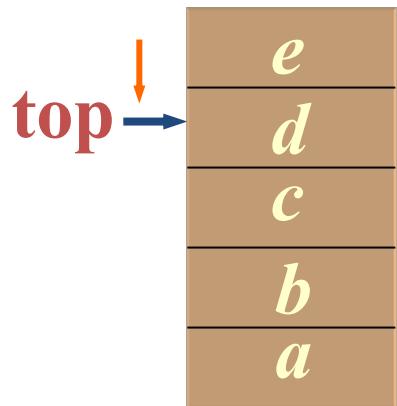


top → **f** 进栈溢出

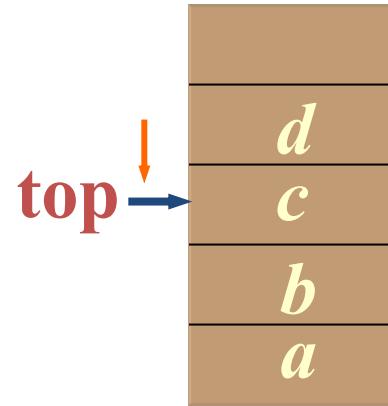




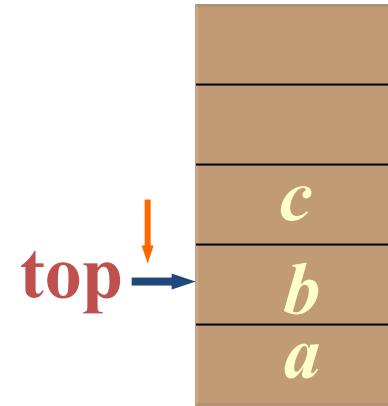
退栈示例



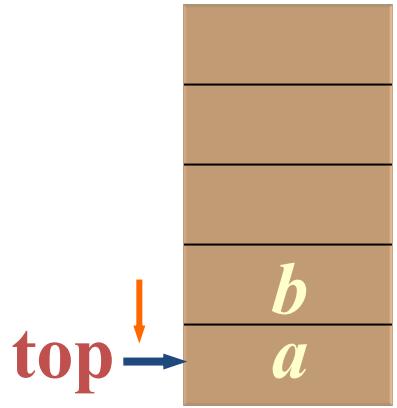
e 退栈



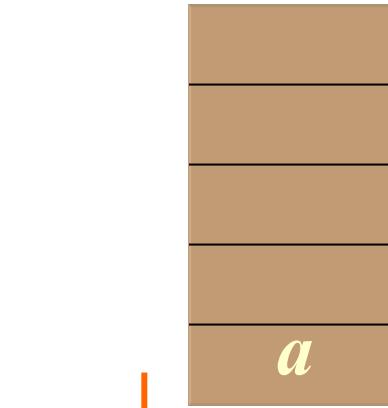
d 退栈



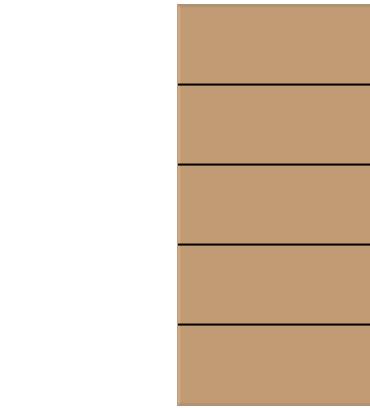
c 退栈



b 退栈



a 退栈



空栈



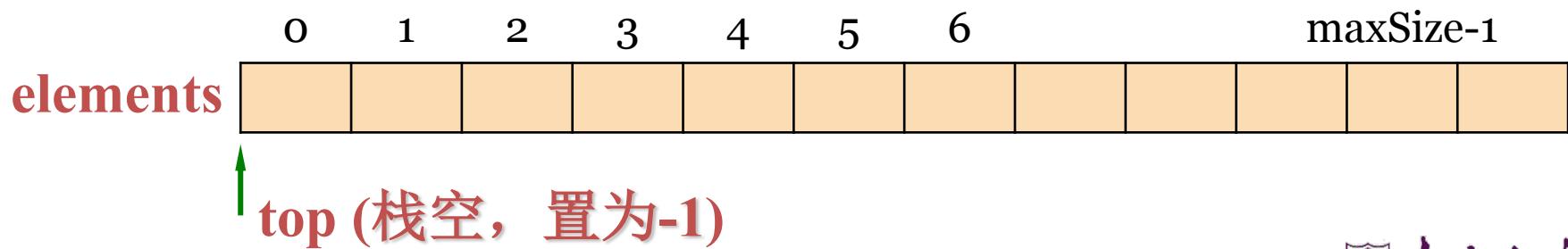
栈的抽象数据类型

```
template <class Type> class Stack {  
public:  
    Stack ( int=10 );           //构造函数  
    void Push ( const Type & item); //进栈  
    Type Pop ();                //出栈  
    Type 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  
}
```



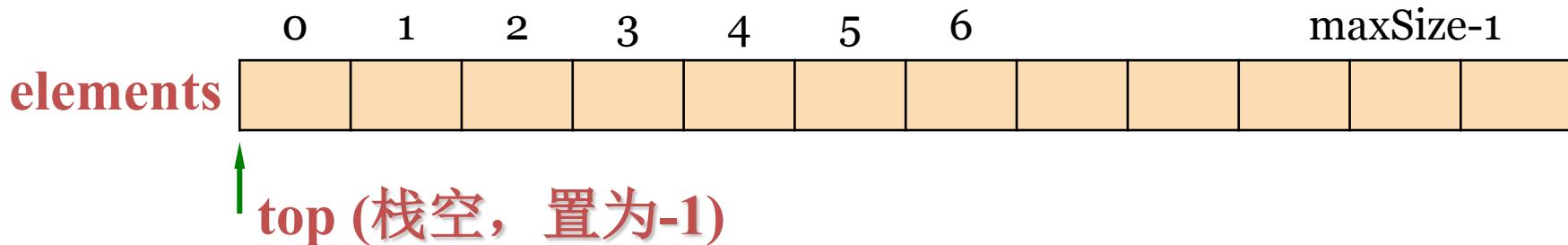
栈的数组表示 — 顺序栈

- 栈的顺序存储结构简称为**顺序栈**，可用**数组**来实现顺序栈。
- 栈底位置是固定不变的，可以将栈底位置设置在数组的两端的任何一个端点；
- 栈顶位置随着进栈和退栈操作而变化，用一个整型变量**top**（栈顶指针）来指出栈顶。





栈的数组表示 – 顺序栈



```
#include <assert.h>
template <class Type> class Stack {
private:
    int top;                                // 栈顶指针
    Type *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 (Type x); //进栈

int Pop (Type& x); //出栈

int GetTop (Type& x); //取栈顶

void MakeEmpty () { top = -1; } //置空栈

int IsEmpty () const { return top == -1; }

int IsFull () const

{ return top == maxSize-1; }

}



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]; //复制原有数组
    maxSize += maxSize;
    delete [ ]elements;
    elements = newArray;      //改变elements指针
};
```



顺序栈的操作

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

};



顺序栈的操作

```
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;
};
```



顺序栈的溢出

- 上溢（Overflow）

当栈中已经有 maxSize 个元素时，如果再做进栈运算，所产生的现象

- 下溢（Underflow）

对空栈进行出栈运算时所产生的现象

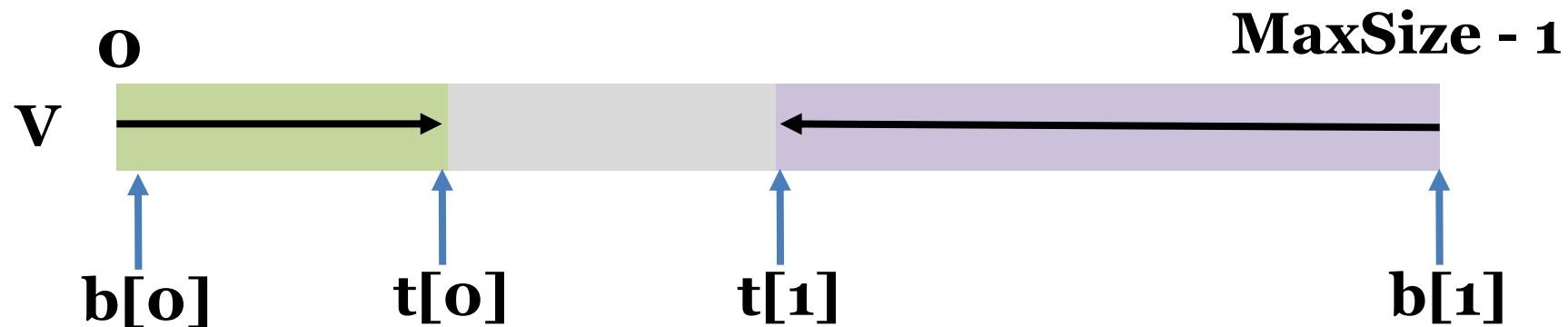


如何合理进行栈空间分配，以避免栈溢出或空间的浪费？

- 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 $V[\text{maxSize}]$
- 该空间的两端分别设置为两个栈的栈底
- $b[0] = -1, b[1] = \text{maxSize}$
- 两个栈的栈顶都向中间延伸，直到相遇
- 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



双栈共享一个栈空间

```
bool push(DualStack& DS, Type x, int i) {  
    if (DS.t[0]+1 == DS.t[1]) return false; //栈满  
    if (i == 0) DS.t[0]++; else DS.t[1]--;  
    DS.V[DS.t[i]] = x;  
    return true;  
}
```

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



双栈共享一个栈空间

超过两个栈怎么办？

- 处理十分复杂
- 尤其是在存储空间即将充满时
- 需要大量的移动



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
- 适合于多栈操作



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          数据成员
    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

void output(ostream& os,

```
StackNode <E> *ptr, int& i);
```

//递归输出栈的所有元素

public:

```
LinkedStack() : top(NULL) {} //构造函数
```

```
~LinkedStack() { makeEmpty(); }
```

void Push(E x);

bool Pop(E& x);

//构造函数

//析构函数

//进栈

//退栈





```
bool getTop(E& x) const;           //取栈顶
bool IsEmpty() const { return top == NULL; }
void makeEmpty();                  //清空栈的内容
int k = 1;
friend ostream& operator << (ostream& os,
    LinkedStack<E>& s) { output(os, s.top, k); }
                                //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;                  //释放结点  
    delete p;  
    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>
void LinkedStack<E>::output(ostream& os,
    StackNode<E> *ptr, int& i) {
    //递归输出栈中所有元素（沿链逆向输出）
    if (ptr != NULL) {
```



```
if (ptr->link != NULL)
    output(os, ptr->link, i++);
os << i << ":" << p->data << endl;
        //逐个输出栈中元素的值
}
};

};
```



栈的应用：括号匹配的检验

假设表达式中允许括号嵌套，则检验括号是否匹配的方法可用栈。

例：

(() () (()))



栈的应用：括号匹配的检验

示例 1:

- 输入: "()" 输出: true

示例 2:

- 输入: "()[]{}" 输出: true

示例 3:

- 输入: "[]" 输出: false

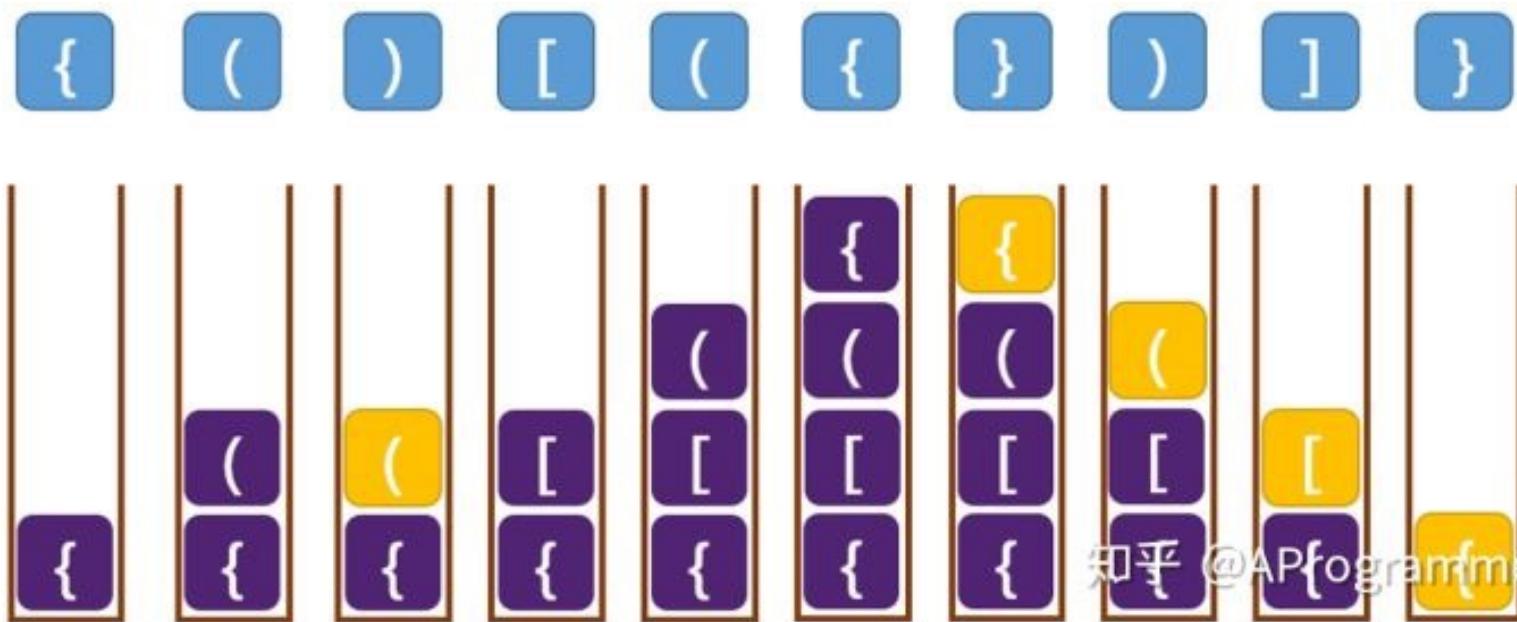
示例 4:

- 输入: "[][]" 输出: false



栈的应用：括号匹配的检验

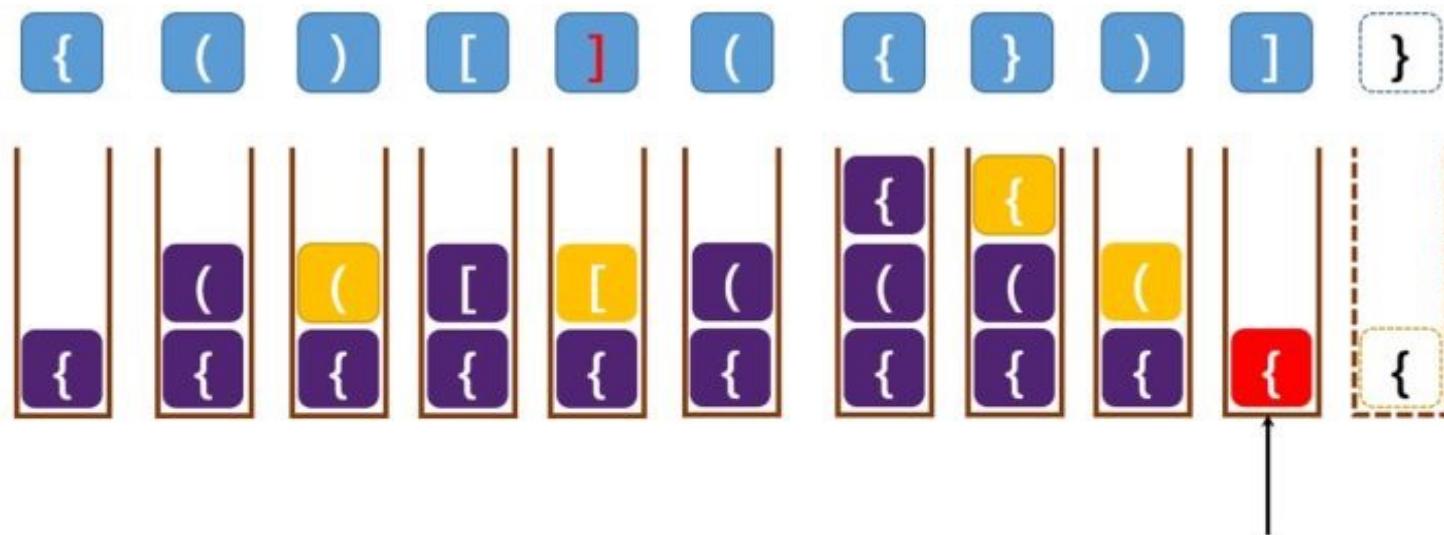
例 1: { () [({ })] }





栈的应用：括号匹配的检验

例 2: { () []({ }) }



不匹配!

这一步方括号与栈顶的花括号不匹配，程序会退出，不再检查后面的字符
知乎 ZHIDU.com @corner



栈的应用：括号匹配的检验

```
bool paren(const char exp[], int low, int high) {  
    Stack<char> S;  
    while (low <= high) { //自左向右逐一检查各字符  
        switch (exp[low]) {  
            case '(': case '[': case '{': //当遇到左括号时  
                S.push(exp[low]); break; //将字符入栈  
            case ')': //当遇到右括号时，判断栈是否为空，  
若不为空，则判断是否与栈顶符号相匹配  
                if ((S.empty() || ('(' != S.pop()))) return  
false;  
                break;  
        }  
    }  
}
```



栈的应用：括号匹配的检验

case ']':

```
if ((S.empty()) || ('[' != S.pop())) return false;  
break;
```

case '}':

```
if ((S.empty()) || ('{' != S.pop())) return false;  
break;
```

default: break; //非括号字符一律忽略

}

low++;

}

return S.empty();

}



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+;



表达式例子

$$A + B * \underbrace{(C - D)}_{R_1} - \underbrace{E / F}_{R_4}$$
$$\underbrace{\qquad\qquad\qquad}_{R_2}$$
$$\underbrace{\qquad\qquad\qquad}_{R_3}$$
$$\underbrace{\qquad\qquad\qquad}_{R_5}$$

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 $a\ b\ c\ d\ -\ *\ +\ e\ f\ / \ -$

- 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：表达式求值

思考：表达式求值，采用中缀表达式？后缀表达式？

中缀表达式 $a + b * (c - d) - e / f$

后缀表达式 $a\ b\ c\ d\ -\ *\ +\ e\ f\ / \ -$



应用：中缀表示计算表达式的值

- 例 中缀表达式求值

假定表达式中只有运算符和操作数两类成分，
运算符有加+，减-，乘*，除/，以及圆括号
()。

- 设置两个栈：运算符栈(**optr**)和操作数栈
(opnd)



应用：中缀表示计算表达式的值

算法思想：首先置操作数栈**opnd**为空，将表达式起始符‘=’压入运算符栈**optr**作为栈底元素，然后从左向右扫描用于存储中缀表达式的**infix**数组，读入字符**infix[i]**，直至表达式结束，

1. 若**infix[i]**是数字字符就拼成数字，然后压入**opnd**栈，
2. 若**infix[i]**为运算符，则按以下规则进行：
 - (1) 若**infix[i]**的优先级高于**optr**栈顶的运算符，**infix[i]**入**optr**栈；
 - (2) 若**infix[i]**的优先级低于**optr**栈顶的运算符，则弹出**optr**栈顶的运算符，并从**opnd**栈弹出两个操作数，进行相应算数运算后，结果压入**opnd**栈； **a + b * (c - d) - e / f**

中缀表达式 $12*(6-3.5)=$ 的求值过程



应用：中缀表示计算表达式的值

注：

- (1) 若 $\text{infix}[i]$ 是'('， 其优先级最高， 入 optr 栈；
- (2) 若 $\text{infix}[i]$ 是')'， 并且 optr 栈顶是'('， 则 optr 执行退栈操作，从而消去了左右括号； 否则')'被解释为优先级低于（除了'=’之外的）其它运算符， 要按上面规则(2)进行， 直到碰到'('， optr 栈弹出'('与栈外的')'抵消；
- (3) 若 $\text{infix}[i]$ 是'=’， 我们认为'=’的优先级是最低的， 因此 optr 退栈， 用弹出的运算符与从 opnd 栈弹出的两个操作数进行相应算术运算， 结果压入 opnd 栈。这个过程一直持续到 optr 栈只剩'=’为止， 运算结束。



应用：中缀表示计算表达式的值

下表中定义了运算符在栈内外的优先关系，数值越大优先级越高。

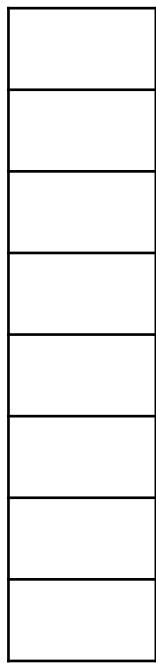
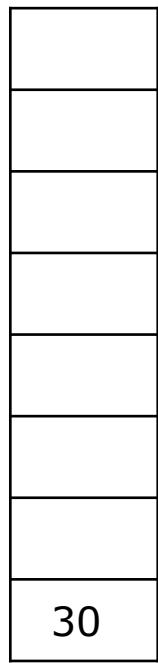
表 运算符之间的优先关系

运算符	=	(*， /	+， -)
栈内优先级	0	1	5	3	6
栈外优先级	0	6	4	2	1



应用：中缀表示计算表达式的值

例：中缀表达式 $12*(6-3.5)=$ 的求值过程



1	2	*	(6	-	3	.	5)	=
---	---	---	---	---	---	---	---	---	---	---



弹出运算符：theta =-

弹出运算数2：b=3.5

弹出运算数1：a=6

压入数字栈 $a-b=2.5$

弹出 ‘(’

弹出运算符：theta =*

弹出运算数2：b=2.5

弹出运算数1：a=12

压入数字栈 $a*b=30$

弹出 ‘=’



应用：中缀表示计算表达式的值

例：中缀表达式 $12*(6-3.5)=$ 的求值过程

步骤	中缀表达式	opnd 栈	optr 栈	主要操作
初始	$12*(6-3.5)=$		=	optr. push(' =') ;
1	<u>12*(6-3.5)=</u>	12	=	opnd. push(12) ;
2	<u>*</u> (6-3.5)=	12	= *	optr. push(' *') ;
3	<u>(6-3.5)=</u>	12	= * (optr. push(' ()') ;
4	<u>6-3.5)=</u>	12 6	= * (opnd. push(6) ;
5	<u>-3.5)=</u>	12 6	= * (-	optr. push(' -') ;
6	<u>3.5)=</u>	12 6 3.5	= * (-	opnd. push(3.5) ;
7	<u>)=</u>	12 2.5	= * (opnd. push(operate(6, ' -', 3.5)) ;
8	<u>)=</u>	12 2.5	= *	item=optr. pop() ; // 消去一对括号
9	<u>=</u>	30	= *	opnd. push(operate(12, ' *', 2.5)) ;
10	<u>=</u>	30	=	item=optr. pop() ; // 消去' ='
11		30		result=opnd. pop() ;



应用：后缀表示计算表达式的值

- 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



应用：后缀表示计算表达式的值

例：后缀表达式 $12\ 6\ 2\ / 0.5\ -\ *$ 的求值过程

postfix

1	2	6	2	/	0	.	5	-	*
---	---	---	---	---	---	---	---	---	---



opnd

30

b=2.5

$$a=12$$

$$a^*b=30$$



应用：后缀表示计算表达式的值

例：后缀表达式 $12\ 6\ 2\ / \ 0.5\ -\ *\ 1$ 的求值过程

步骤	后缀表达式 postfix	主要操作	栈 opnd
1	<u>12</u> 6 2 / 0.5 - *	opnd. push(12) ;	12
2	<u>6</u> 2 / 0.5 - *	opnd. push(6) ;	12 6
3	<u>2</u> / 0.5 - *	opnd. push(2) ;	12 6 2
4	<u>/</u> 0.5 - *	b=opnd. pop() ; a=opnd. pop() ; opnd. push(a/b) ;	12 3
5	<u>0.5</u> - *	opnd. push(0.5) ;	12 3 0.5
6	<u>-</u> *	b=opnd. pop() ; a=opnd. pop() ; opnd. push(a-b) ;	12 2.5
7	<u>*</u>	b=opnd. pop() ; a=opnd. pop() ; opnd. push(a*b) ;	30



应用：后缀表示计算表达式的值

```
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 );  
        }  
    }  
}
```



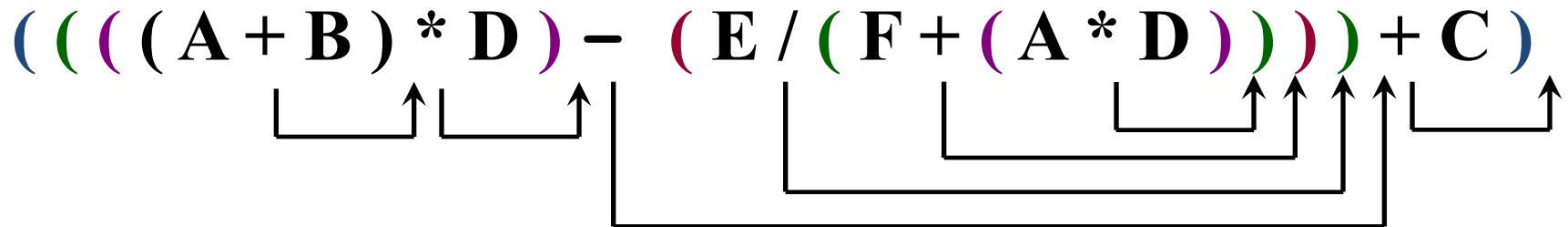
应用：后缀表示计算表达式的值

```
void 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; Boolean 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  
    }  
}
```



思考：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 $A\ B\ +\ D\ *\ E\ F\ A\ D\ *\ +\ /\ -\ C\ +$



第三章 栈和队列

3.1 栈

3.2 栈与递归

3.3 队列

3.4 优先级队列



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



定义是递归的

例如，阶乘函数

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

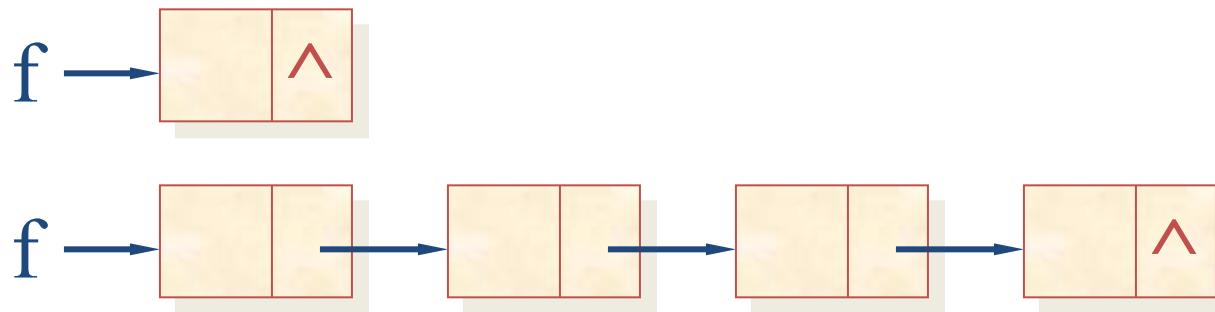
求解阶乘函数的递归算法

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



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。

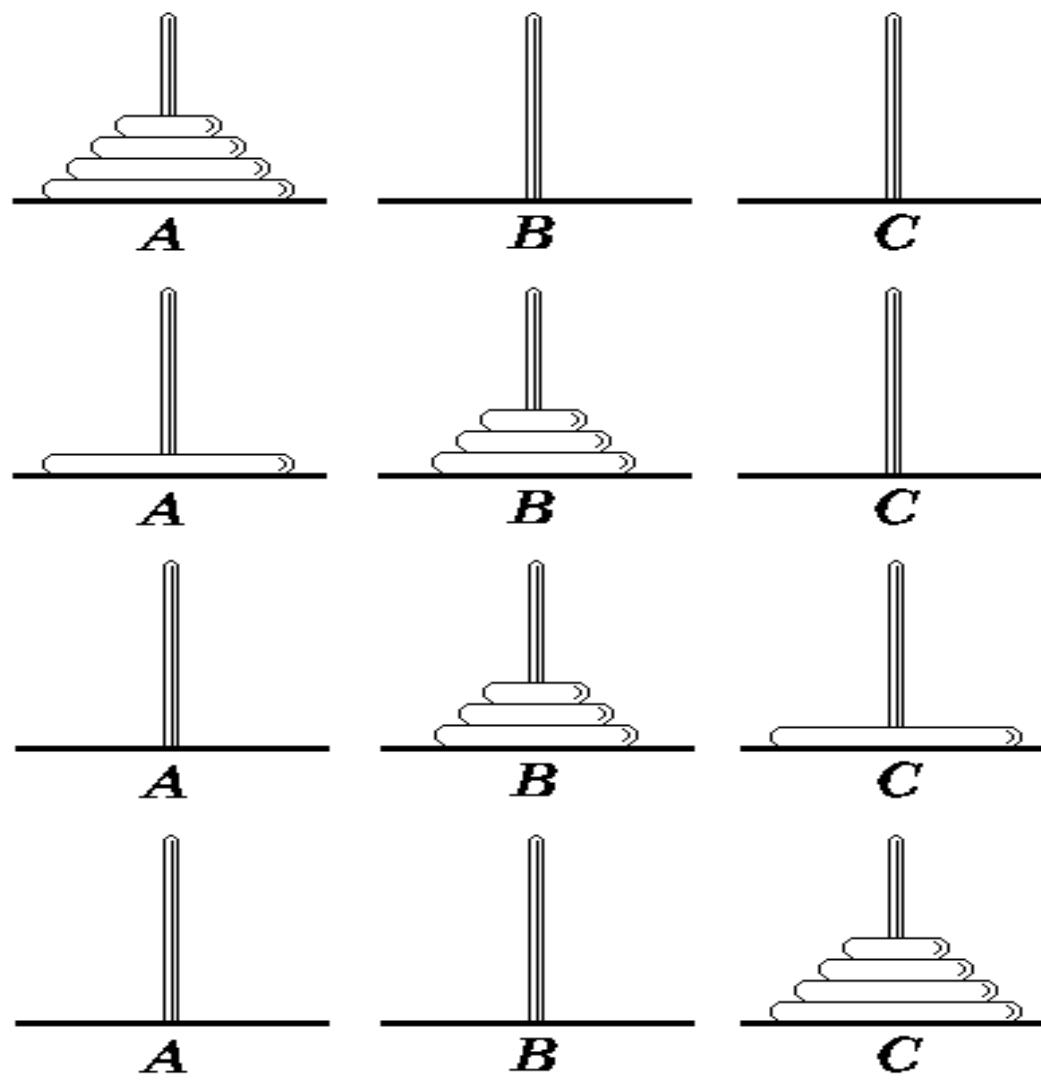


问题的解法是递归的

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。





```
#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);
    }
}
```



n=3时

(3,A,B,C)

A,B,C

(2,A,C,B)

A->C

A,B,C

(2,B,A,C)

A,B,C

(1,A,C,B)

A->C

A->B

A->C

A->B

A->C

A,B,C

(1,B,A,C)

A->C

A->C

B->C

C->B

A,B,C

(1,A,C,B)

A->C

B->C

A->C

A->B

B->A

A,B,C

(1,B,A,C)

A->C

B->C

A->C



南京大學
NANJING UNIVERSITY



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）

使用递归的前提：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。

```
Function <name> ( <parameter list> ) {  
    if ( < initial condition> ) //递归结束条件  
        return ( initial value );  
    else //递归调用  
        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**。



```
void main() {  
    int n;  
    n = Factorial(4);  
  
    RetLoc1 }  
(外部)  
  
long Factorial(long n) {  
    int temp;  
    if (n == 0) return 1;  
    else temp = n * Factorial(n-1);  
  
    RetLoc2 }  
(内部)      return temp;
```



求解阶乘 $n!$ 的过程





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



调用块

函数块

活动记录



F (.....) ;
<下一条指令>

Function F(<参数表>)

.....
<return>

函数递归时的活动记录

返回地址(下一条指令)

局部变量

参数



计算Fact时活动记录的内容

参数调用

参数 返回值 返回地址

4	24	RetLoc1
---	----	---------

3	6	RetLoc2
---	---	---------

2	2	RetLoc2
---	---	---------

1	1	RetLoc2
---	---	---------

0	1	RetLoc2
---	---	---------

返回时的指令

return 4*6 //返回 24

return 3*2 //返回 6

return 2*1 //返回 2

return 1*1 //返回 1

return 1 //返回 1

结果返回



递归过程到非递归过程

- 递归过程简洁、易编、易懂
- 但有的时候递归过程效率低，重复计算多（例如Fib数列的计算）
- 改为非递归过程的目的是提高效率

递归改非递归的方法：

- 通过迭代实现非递归过程（高效）
- 借助栈实现非递归过程（功能等价）



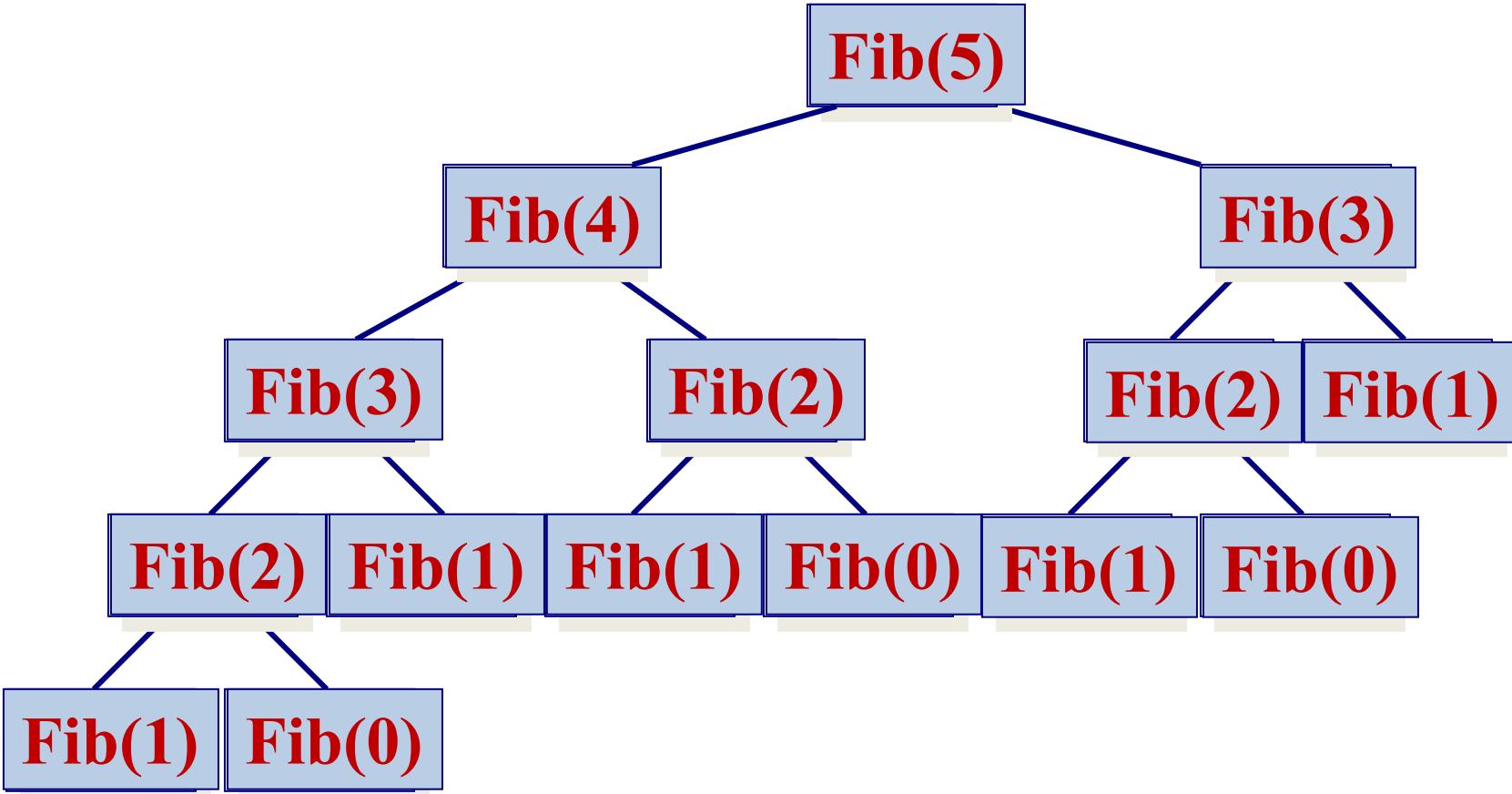
计算斐波那契数列的函数**Fib(n)**的定义

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

如 $F_0 = 0, F_1 = 1, F_2 = 1, F_3 = 2, F_4 = 3, F_5 = 5$

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 $\text{NumCall}(k) = 2 * \text{Fib}(k+1) - 1$



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;  
}
```

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



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); //递归调用语句只有一句，并且放在过程的最后  
    }  
}
```



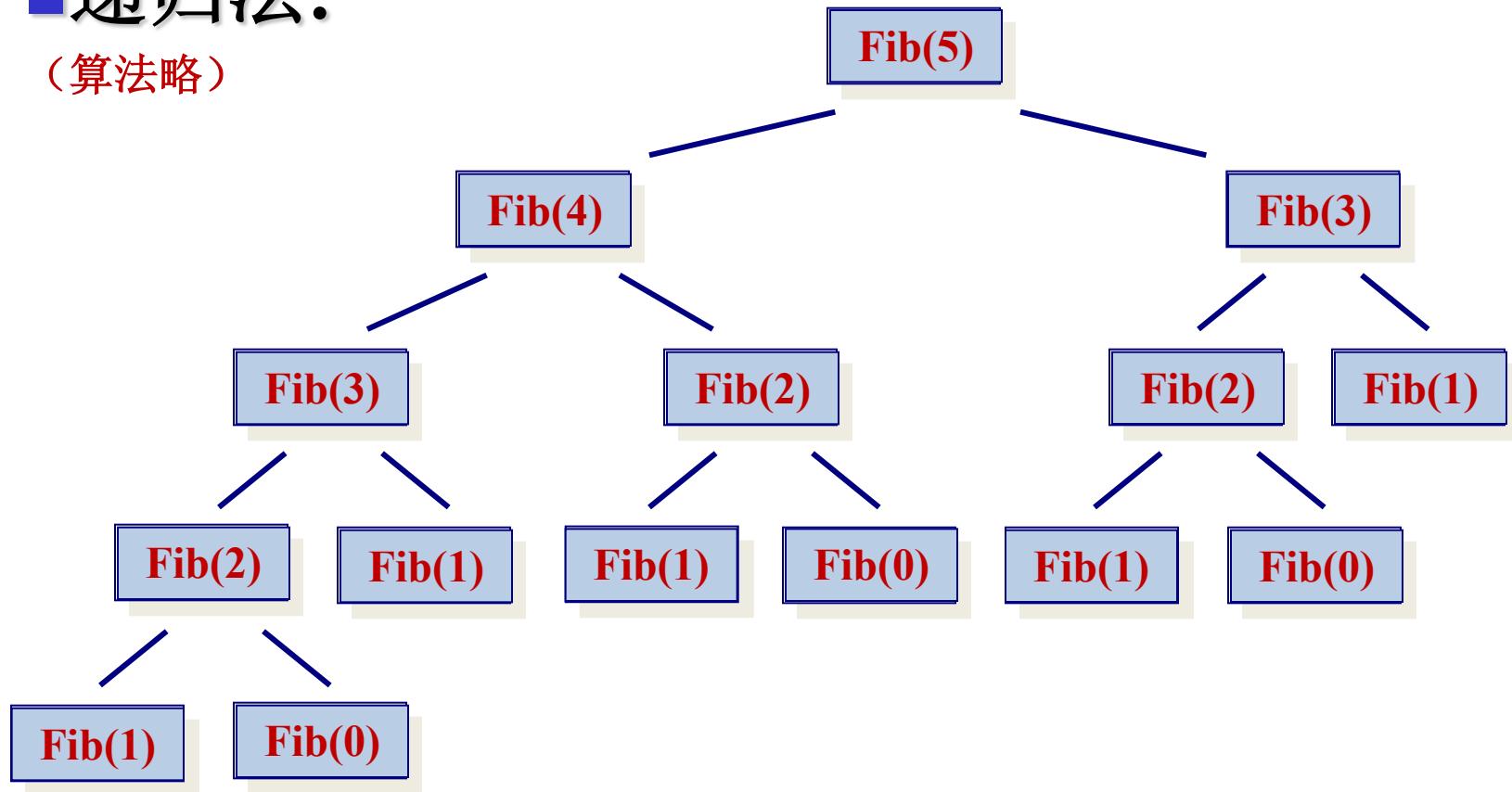
```
void recfunc(int A[ ], int n) {  
    //消除了尾递归的非递归函数，逆向打印数组  
    while (n >= 0) {  
        cout << "value " << A[n] << endl;  
        n --;  
    }  
}
```



斐波那契数列的几种解法

■ 递归法：

(算法略)



调用次数 $\text{NumCall}(k) = 2 * \text{Fib}(k+1) - 1$



迭代法：

可以将数列写成如下形式：

$\text{Fibonacci}[0] = 0$, $\text{Fibonacci}[1] = 1$

当($n \geq 2$)时，有

$\text{Fibonacci}[n] = \text{Fibonacci}[n-1] + \text{Fibonacci}[n-2]$

动态规划： 动态规划的实质是分治和消除冗余，是一种将问题实例分解为更小的、相似的子问题，并**存储子问题的解以避免计算重复的子问题**，来解决最优化问题的算法策略。



公式解法

利用通项公式求 $F(n)$:

$$F(n) = \frac{1}{\sqrt{5}} \times \left(\left[\frac{1 + \sqrt{5}}{2} \right]^n - \left[\frac{1 - \sqrt{5}}{2} \right]^n \right)$$

注意：其中 $[x]$ 是取整符号，表示取距离 x 最近的整数。



作业：小行星碰撞问题

给定一个整数数组 asteroids，表示在同一行的小行星。对于数组中的每一个元素，其**绝对值表示小行星的大小，正负表示小行星的移动方向**（正表示向右移动，负表示向左移动）。每一颗小行星以相同的速度移动。找出碰撞后剩下的所有小行星。

碰撞规则：两个小行星相互碰撞，较小的小行星会爆炸。如果两颗小行星大小相同，则两颗小行星都会爆炸。两颗移动方向相同的小行星，永远不会发生碰撞。

例：

输入：asteroids = [5,10,-5]

输出：[5,10]

解释：10 和 -5 碰撞后只剩下 10。5 和 10 永远不会发生碰撞。



第三章 栈和队列

3.1 栈

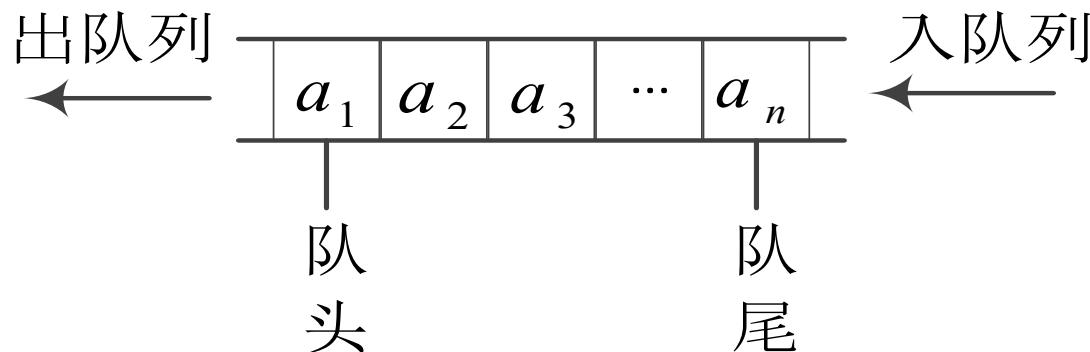
3.2 栈与递归

3.3 队列

3.4 优先级队列



队列 (queue)



● 定义

- 队列是只允许在一端删除，在另一端插入的顺序表
- 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
```

数据成员

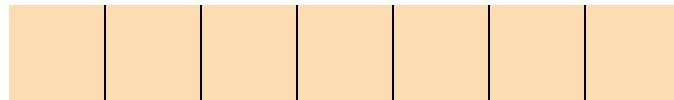


```
~SeqQueue() { delete[ ] elements; } //析构函数
bool EnQueue(E x);           //新元素进队列
bool DeQueue(E& x);         //退出队头元素
bool getFront(E& x);         //取队头元素值
void makeEmpty() { front = rear = 0; }
bool IsEmpty() const { return front == rear; }
bool IsFull() const
{ return ((rear+1)% maxSize == front); }
                           //判满(循环队列的实现)

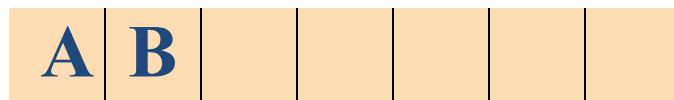
int getSize() const
{ return (rear-front+maxSize) % maxSize; }
                           //计算队列元素个数(循环队列的实现);
```



队列的进队和出队（数组方式—非循环）



front rear 空队列



front rear B进队



front rear A退队



front rear E,F,G进队



front rear A进队



front rear C, D进队



front rear B退队



front rear H进队,溢出



队列的进队和出队

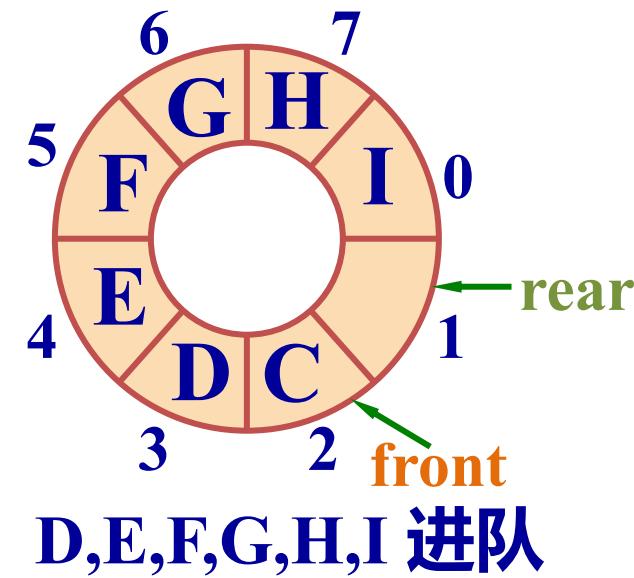
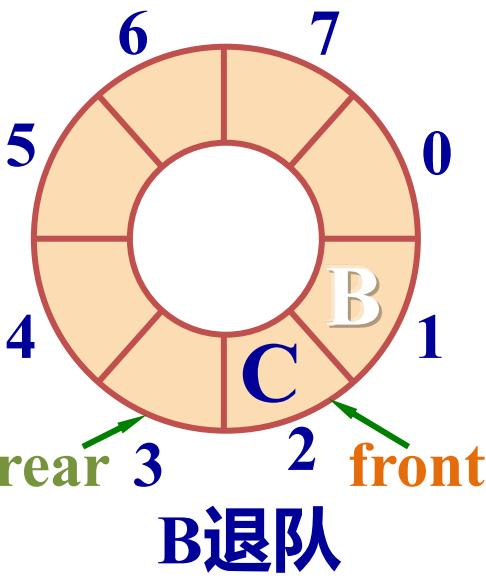
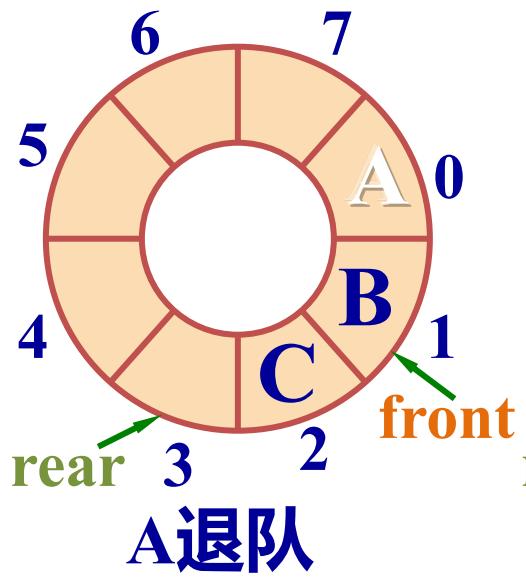
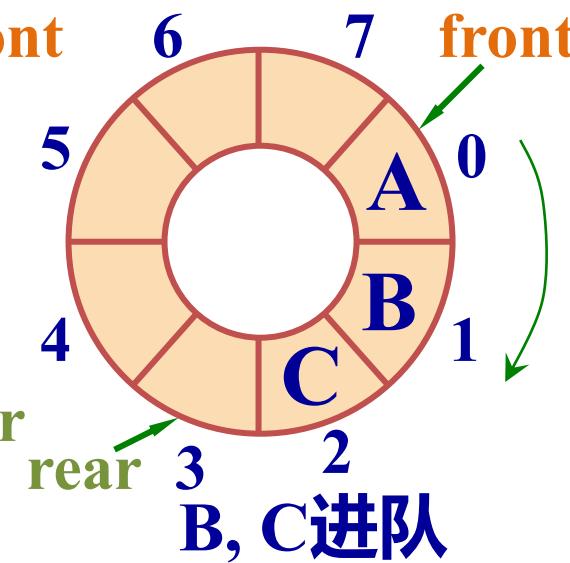
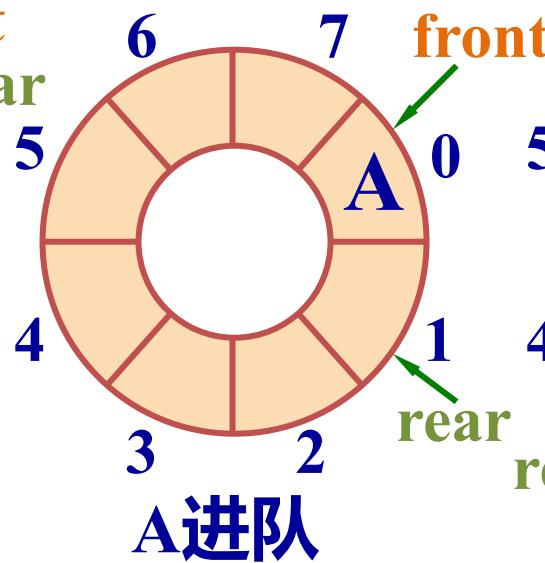
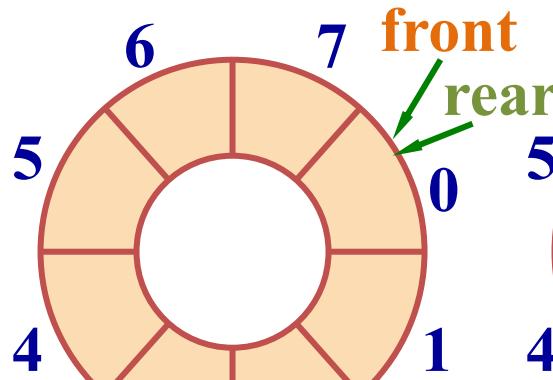
- 进队: $rear = rear + 1$, 新元素在 $rear$ 处加入。
- 出队: $front = front + 1$, 取出下标为 $front$ 的元素
- 队空时: $rear == front$
- 队满时: $rear == maxSize - 1$ (假溢出)

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接的表处理。
- 队头、队尾指针加1时从 $\text{maxSize}-1$ 直接进到0，可用语言的取模(余数)运算实现。
- 队头指针进1: $\text{front} = (\text{front}+1) \% \text{maxSize};$
- 队尾指针进1: $\text{rear} = (\text{rear}+1) \% \text{maxSize};$
- 队列初始化 : $\text{front} = \text{rear} = 0;$
- 队空条件 : $\text{front} == \text{rear};$
- 队满条件 : $(\text{rear}+1) \% \text{maxSize} == \text{front};$





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;
```

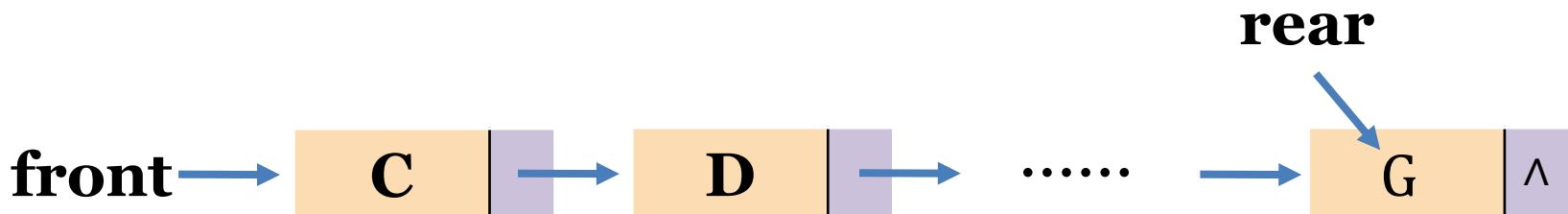


```
x = elements[front];           //先取队头
front = (front+1) % maxSize;   //再队头指针加一
return true;
};
```

```
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 $\text{front} == \text{NULL}$



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 d = 0, QueueNode<E>
              *next = NULL) : data(d), link(next) { }
};
```

数据成员



```
template <class E>
class LinkedQueue {
private:
    QueueNode<E> *front, *rear; //队头、队尾指针 数据成员
public:
    LinkedQueue() : rear(NULL), front(NULL) { }
    ~LinkedQueue();
    bool EnQueue(E x);
    bool DeQueue(E& x);
    bool GetFront(E& x);
    void MakeEmpty();           //实现与~LinkedQueue()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

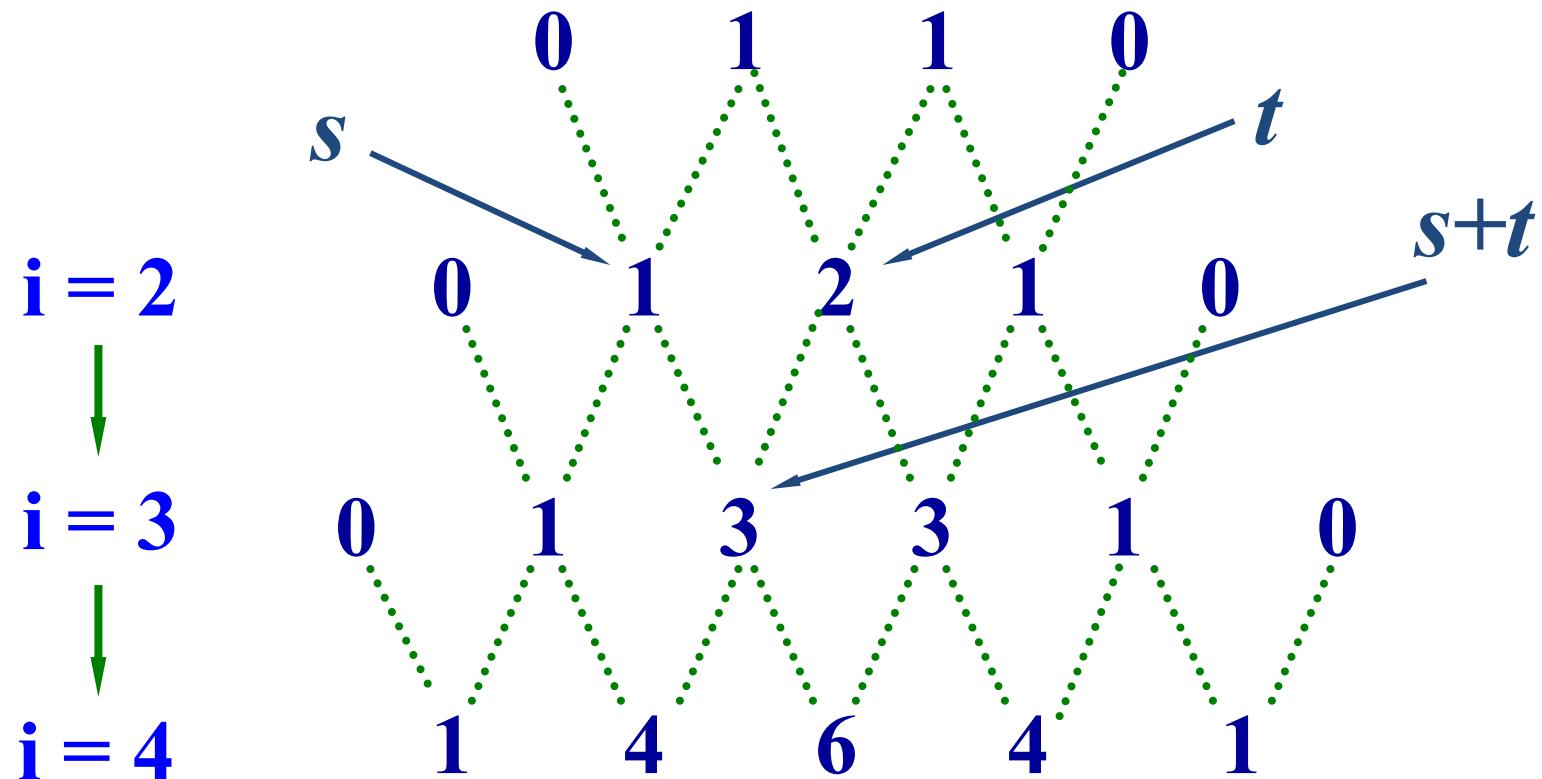
逐行打印二项展开式 $(a + b)^i$ 的系数

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
		1	6	15	20	15	6



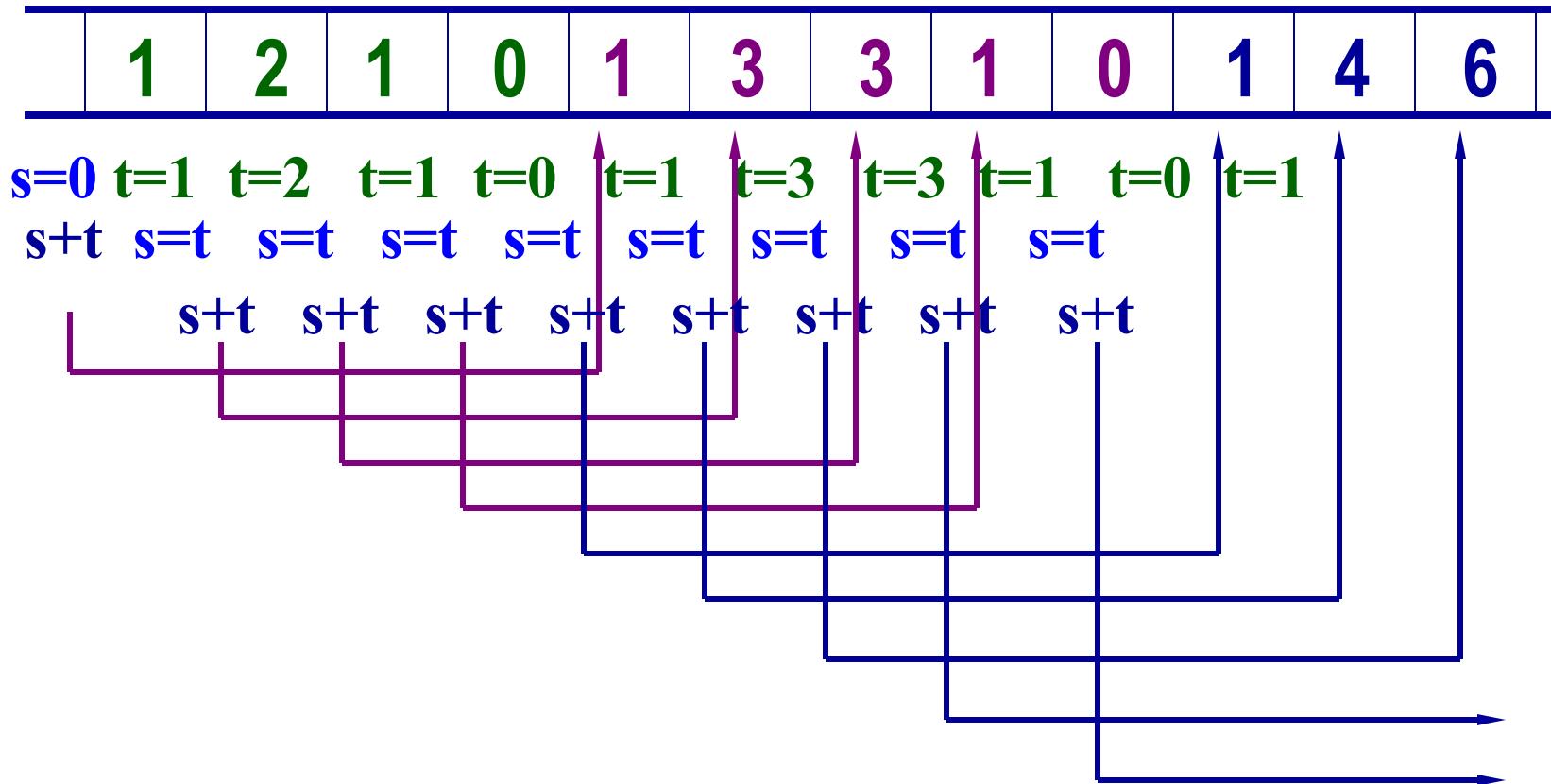
分析第 i 行元素与第 $i+1$ 行元素的关系



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;                  //换行
    q.Enqueue(0);
    for (int j = 1; j <= i+2; j++) { //处理第i行的系数
        q.DeQueue(t);
        q.Enqueue(s + t);
        s = t;
        if (j != i+2) cout << s << ' ' ; //输出i行的系数 ,
                                         //第i+2个是0
    }
}
```



第三章 栈和队列

3.1 栈

3.2 栈与递归

3.3 队列

3.4 优先级队列



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的元素



- 优先权是根据问题而定的
- 出现相同的优先级的元素时，按**FIFO**的方式处理

应用场景：任务调度。如果有多个任务待处理，则首先处理优先级高的任务，使效率最大化。



优先队列的存储表示与实现

- 数组实现（本章）
- 堆（第五章）



例子

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

数据成员

```
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 [ ] pquelements; }
bool Insert(E x);           // 入队操作
bool RemoveMin(E& x); // 出队操作，优先权
                           // 值最小的出队
bool GetFront(E& x);     // 取队头元素
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  
    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++)
        //1~count-1号元素移动
        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;
}
```



本章小结

- 栈的两种物理实现及应用
 - 顺序栈（数组实现的栈）
 - 链式栈（单链表实现的栈）
 - 应用：后缀表达式的求值
 - 递归过程和递归工作栈
- 队列的两种物理实现及应用
 - 顺序队列（循环队列）
 - 链式队列（单链表实现的队列）
 - 应用：杨辉三角形的打印
- 优先级队列的定义及实现
 - 顺序优先级队列（数组实现的优先级队列）