



北京大学

第三章 栈和队列

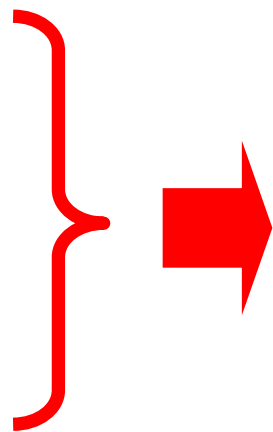
宋国杰

gjsong@pku.edu.cn

课程内容

1. 栈

2. 队列

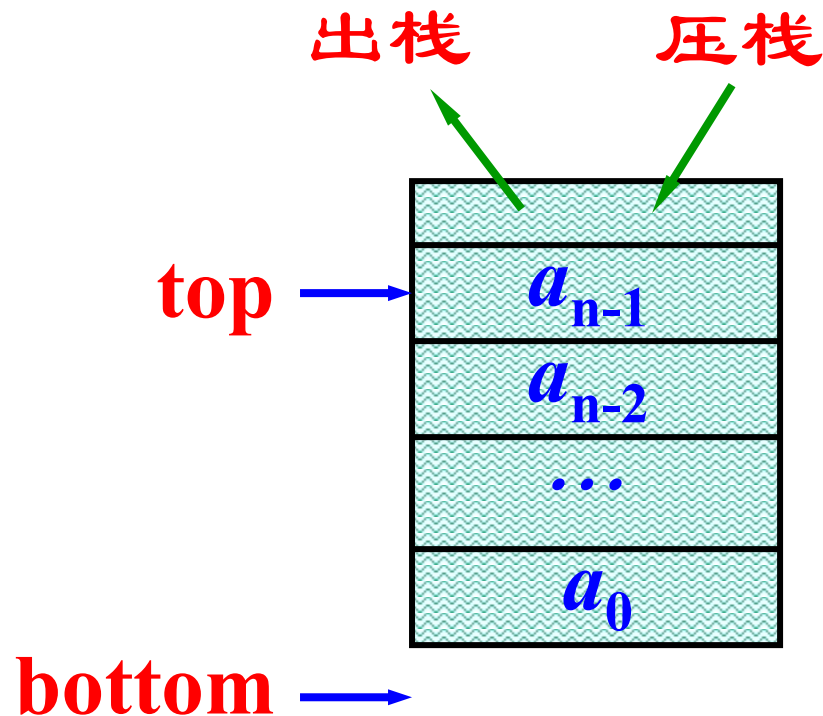


操作受限的线性结构

3. 栈和队列的讨论

3.1 栈

- 限制在一端访问的线性表
- 后进先出(Last-In First-Out, LIFO), 也称 “下推表”
- ‘压栈’ 与 ‘出栈’
- ‘栈顶’ 与 ‘栈底’



栈的主要操作

- 入栈 (**push**)
- 出栈 (**pop**)
- 取栈顶元素 (**top**)
- 判断栈是否为空 (**isEmpty**)
- 判断栈是否为满 (**isFull**)

3.1.1 栈的ADT

```
template <class T>                // 栈的元素类型为 T

class Stack {

    public:                        // 栈的运算集

        void clear();             // 变为空栈

        bool push(const T item);  // item入栈,成功则返回真,否则返回假

        bool pop(T & item);       // 返回栈顶内容并弹出,成功则真,否则为假

        bool top(T& item);        // 返回栈顶但不弹出,成功真,否则为假

        bool isEmpty();           // 若栈已空返回真

        bool isFull();            // 若栈已满返回真

};
```

➤ 若入栈顺序为1,2,3,4的话，则出栈的顺序可以有哪些？

➡ 1234

➡ 1243

➡ 1324

➡ 1342

➡ 1423

➡ 1432

➡ 2134

➡ 2143

.....

思考题

- 若元素a、b、c、d依次进栈，则可能的出栈序列（14）种；若有元素a、b、c、d、e依次进栈，则可能的出栈序列（42）种。

卡特兰数： $1/(n+1) * C(2n,n)$ 种

栈的存储结构

➤ 顺序栈 (Array-based Stack)

➡ 数组实现

➤ 链式栈 (Linked Stack)

➡ 链表实现

3.1.2 顺序栈

```
template <class T> class arrStack : public Stack <T> {  
    private:                                     // 栈的顺序存储  
        int mSize;                             // 栈中最多可存放的元素个数  
        int top;                               // 栈顶位置, 指向当前栈顶元素  
        T *st;                                 // 存放栈元素的数组  
    public:                                     // 栈的运算的顺序实现  
        arrStack(int size) {                  // 创建顺序栈实例  
            mSize = size;  
            st = new T[mSize];  
            top = -1;  
        }  
}
```



```
arrStack() {
```

```
    top = -1;
```

```
}
```

```
~arrStack() {
```

```
    delete [] st;
```

```
}
```

```
void clear() {
```

```
    top = -1;
```

```
}
```

```
}
```

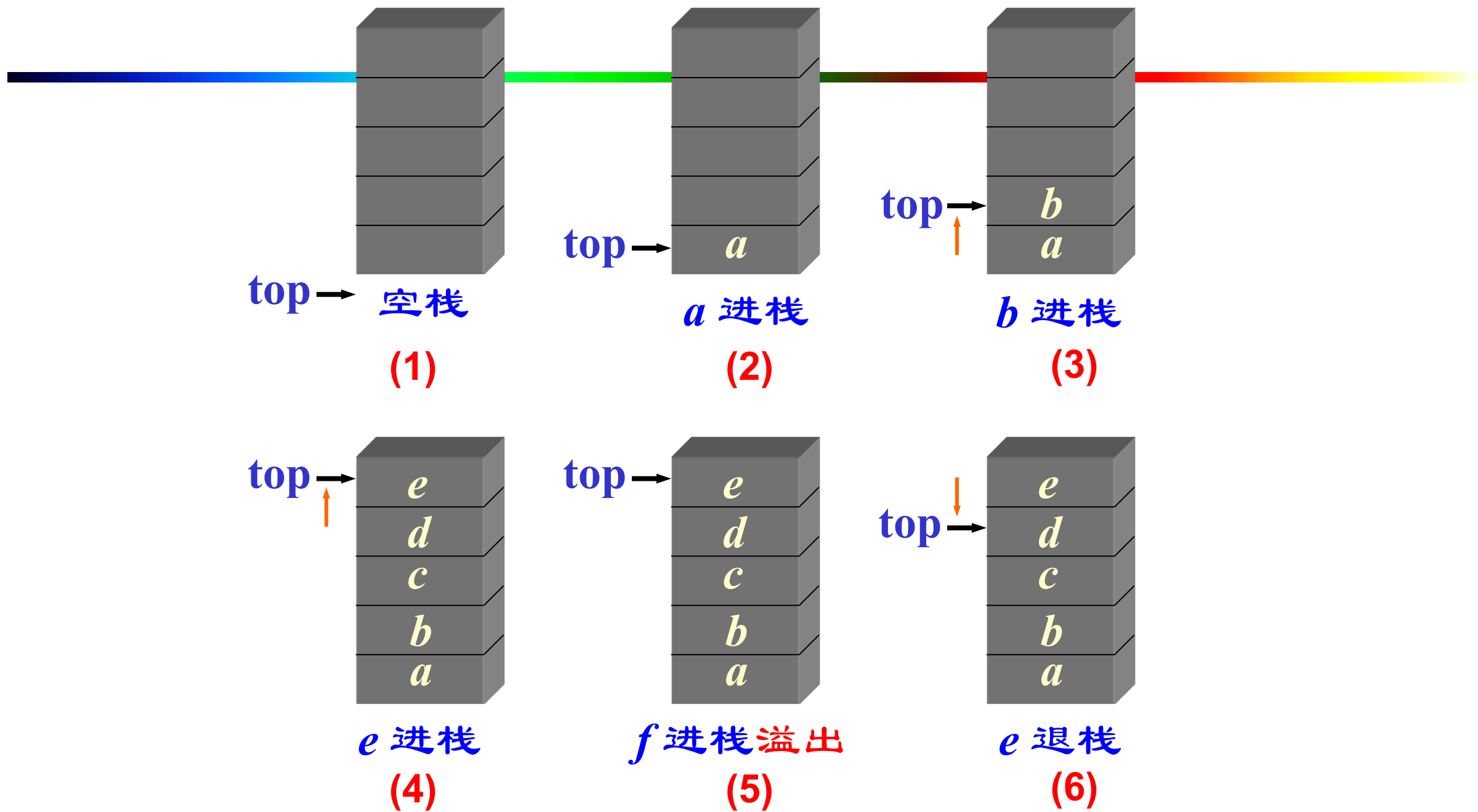
// 创建一个顺序栈的实例

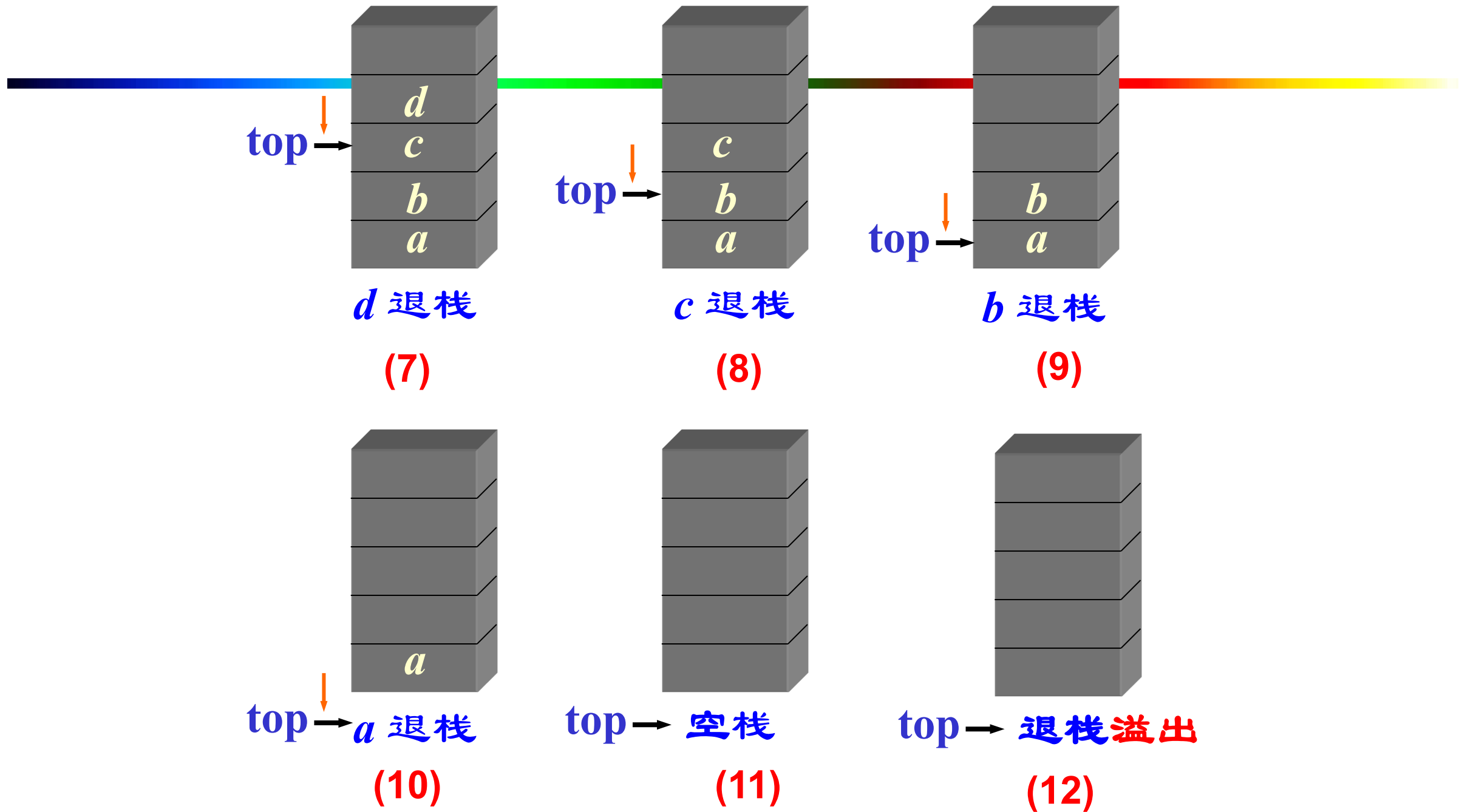
// 析构函数

// 清空栈内容

运算集

- 入栈
- 出栈
- 从栈顶读取，但不弹出
- 变空栈
- 栈满时，返回非零值





顺序栈的溢出

➤ 上溢 (Overflow)

- ➡ 当栈中已经有 maxsize 个元素时，如果再做进栈操作，所产生的“**无空间可用**”现象

➤ 下溢 (Underflow)

- ➡ 空栈进行出栈所产生的“**无元素可删**”现象

入栈

```
bool arrStack<T>::push(const T item) {  
    if (top == mSize-1) {                // 栈已满  
        cout << "栈满溢出" << endl;  
        return false;  
    }  
    else {                                // 新元素入栈并修改栈顶指针  
        st[++top] = item;                 "先执行++, 而后执行进栈操作!"  
        return true;  
    }  
}
```

出栈

```
bool arrStack<T>::pop(T & item) {           // 出栈的顺序实现
    if (top == -1) {                          // 栈为空
        cout << "栈为空，不能执行出栈操作" << endl;
        return false;
    }
    else {
        item = st[top--];                    // “先返回栈顶元素，后执行--”
        return true;
    }
}
```


从栈顶读取，但不弹出

```
bool arrStack<T>::top(T & item) {  
    // 返回栈顶内容，但不弹出  
  
    if (top == -1) {                                     // 栈空  
        cout << " 栈为空，不能读取栈顶元素" << endl;  
        return false;  
    }  
    else {  
        item = st[top];  
        return true;  
    }  
}
```

其他算法

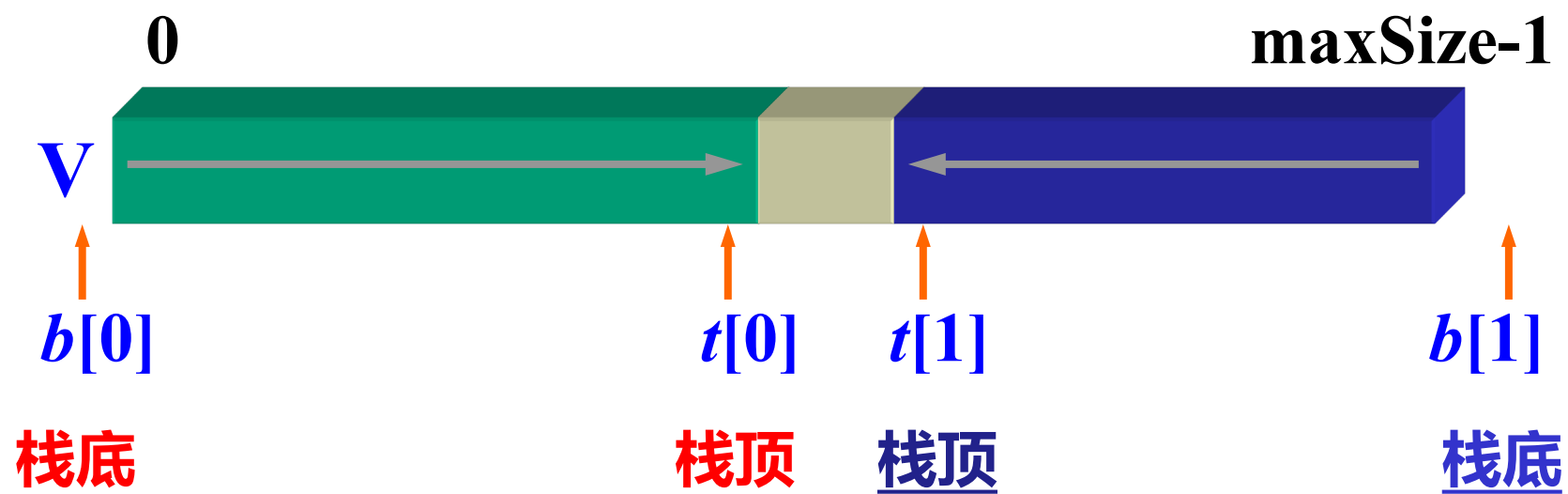
清空栈

```
void arrStack<T>::clear() {  
    top = -1;  
}
```

判断栈满：返回非零值（真值_{true}）

```
bool arrStack<T>::isFull() {  
    return (top == mSize-1) ;  
}
```

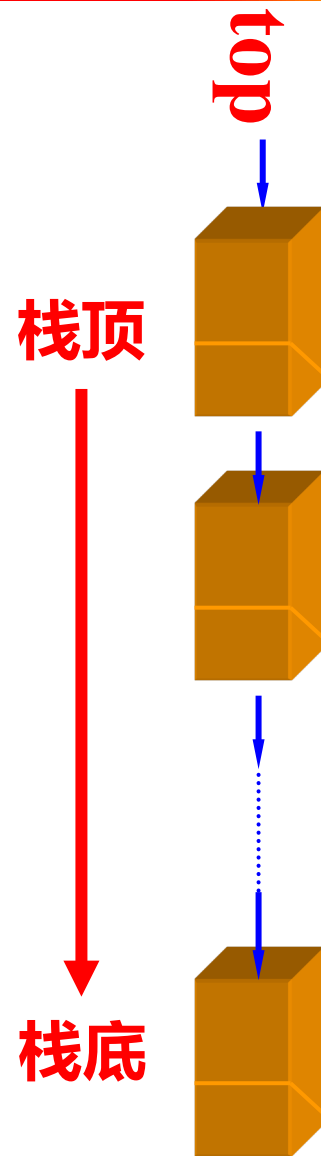
双栈共享一个栈空间



3.1.3 链式栈

➤ 链式栈

- ➡ 栈的链式存储，是运算受限的链表
- ➡ 指针方向：从栈顶向栈底
- ➡ 只在链表头部操作，不需附加头结点
- ➡ 栈顶指针就是链表的头指针（top）
- ➡ 无栈满问题（但存在栈空约束）



链式栈的创建

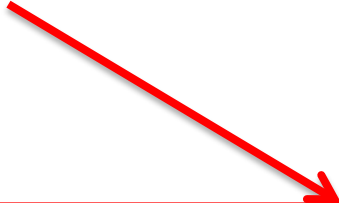
```
template <class T>
class InkStack : public Stack <T> {
    private:
        Link<T>*top;
        int      size;
    public:
        InkStack(int defSize) {
            top = NULL;
            size = 0;
        }
        ~InkStack() {
            clear();
        }
}

// 栈的链式存储
// 指向栈顶的指针
// 存放有效元素的个数
// 栈运算的链式实现
// 构造函数
// 析构函数
```

运算集：压栈

// 入栈操作的链式实现

```
bool InkStack<T>:: push(const T item) {  
    //创建一个新节点，并使其next域赋值top;  
    Link<T>* tmp = new Link<T> (item, top);  
    top = tmp;  
    size++;  
    return true;  
}
```



```
Link(const T info, Link* nextValue) {  
    data = info;  
    next = nextValue;  
}
```

运算集：出栈

bool InkStack<T>:: pop(T& item) { // 出栈操作的链式实现

```
Link <T> *tmp;  
    if (size == 0) {  
        cout << "栈为空，不能执行出栈操作"<< endl;  
        return false;  
    }  
    item = top->data;  
    tmp = top->next;  
    delete top;  
    top = tmp;  
    size--;  
    return true;  
}
```

顺序栈和链式栈的比较

➤ 时间效率

- ➡ 所有操作都只需常数时间
- ➡ 顺序栈和链式栈在时间效率上难分伯仲

➤ 空间效率

- ➡ 顺序栈须说明一个固定的长度
- ➡ 链式栈的长度可变，但增加结构性开销

3.1.4 栈的应用

- 满足后进先出特性，皆可使用栈
- 常用来处理具有**递归**结构的应用
 - ➡ 深度优先搜索
 - ➡ 子程序 / 函数调用的管理
 - ➡ (消除) 递归
- 应用举例
 - ➡ 递归
 - ➡ 表达式求值

1、栈与递归

➤ 递归概念

- ➡ 是指在调用一个函数的过程中又直接调用或间接调用了函数自身

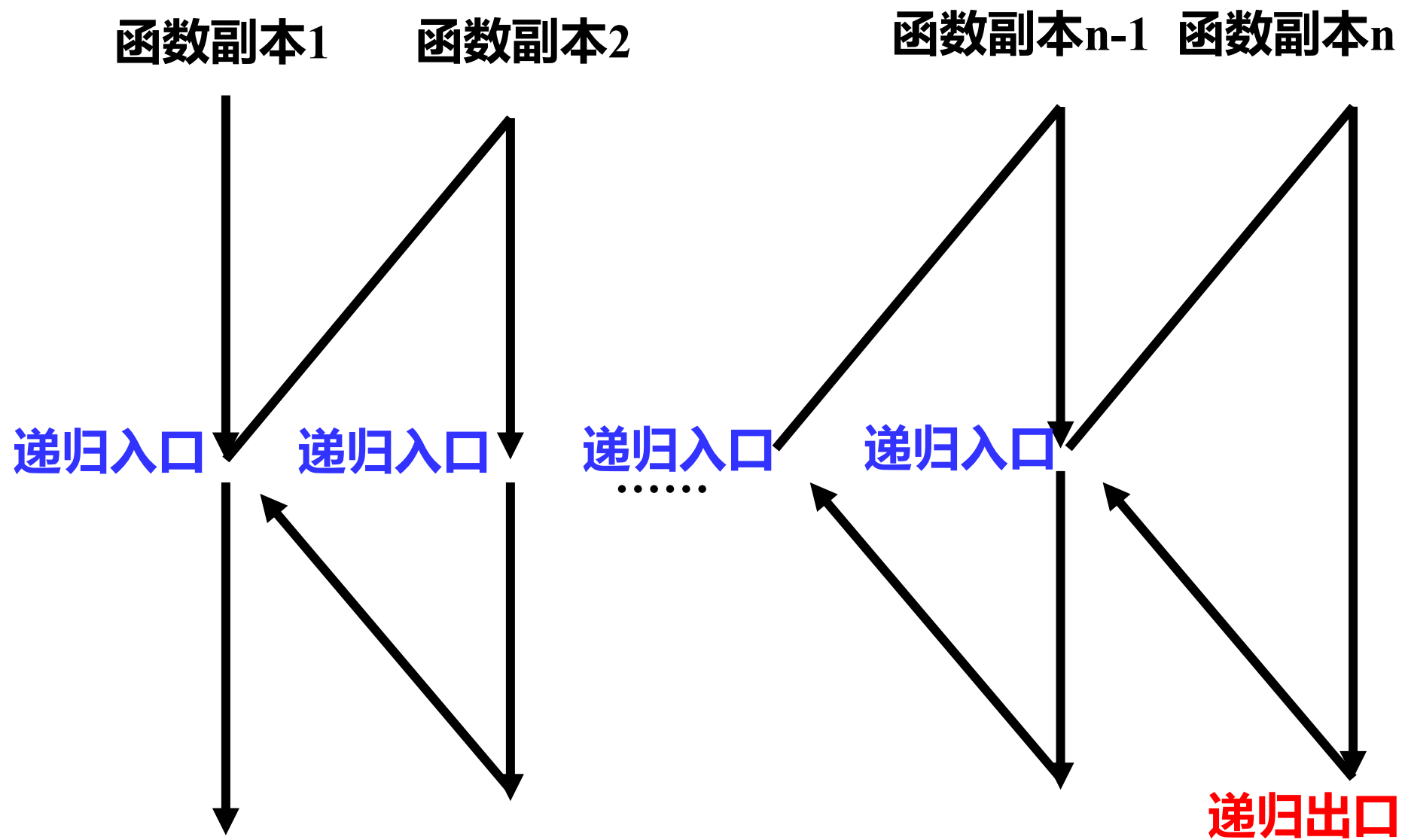
➤ 直接调用

```
long fact(int n) { //求阶乘
    if (n==1)
        return 1;
    return n*fact(n-1);
}
```

➤ 间接调用

```
int f1(int a)
{
    int b;
    b=f2(a+1);
}
int f2(int s)
{
    int c;
    c=f1(s-1);
}
```

递归调用过程示例



递归的实现

- 一个问题能否用递归实现，看其是否具有下面特点
 - ➡ 有递推公式（1个或多个）
 - ➡ 有递归结束条件（1个）
- 编写递归函数时，程序中必须有相应的语句
 - ➡ 一个（或者多个）递归调用语句
 - ➡ 测试结束语句
 - ➡ 先测试，后递归调用
- 递归程序的特点
 - ➡ 易读、易编，但占用额外内存空间。

函数运行时的存储分配

➤ 静态分配

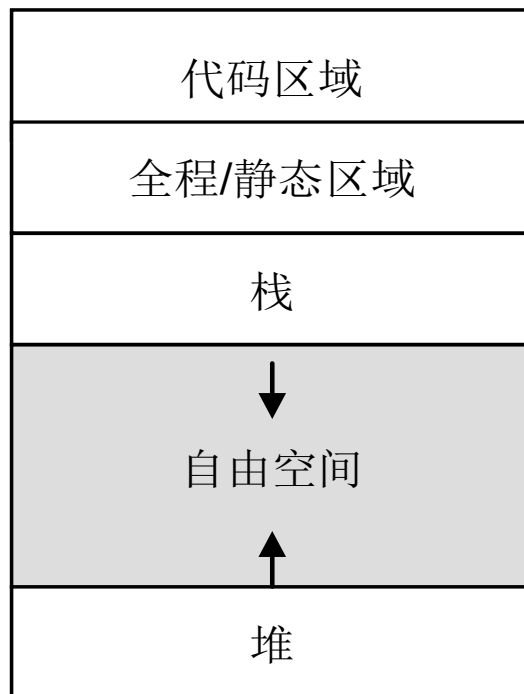
- ➡ 在非递归情况下，数据区的分配可以在程序运行前进行，一直到整个程序运行结束才释放，这种分配称为静态分配
- ➡ 采用静态分配时，函数的调用和返回处理比较简单，不需要每次分配和释放被调函数的数据区

➤ 动态分配

- ➡ 在递归（函数）调用的情况下，被调函数的局部变量不能静态地分配某些固定单元，而必须每调用一次就分配一份，以存放当前所使用的数据，当返回时随即释放。【大小不确定，值不确定】
- ➡ 动态分配在内存中开辟一个称为运行栈的足够大的动态区

动态存储分配

- 用作动态数据分配的存储区，分为堆（heap）和栈（stack）
 - ➡ 栈用于分配发生在后进先出LIFO风格中的数据（诸如函数的调用）
 - ➡ 堆区域则用于不符合LIFO（诸如指针的分配）的动态分配



运行栈中的活动记录

➤ 函数活动记录是动态存储分配中的基本单元

➡ 调用函数时，活动记录包含为局部数据分配的存储空间

自变量（参数）
用做记录信息 诸如返回地址
局部变量
用作局部 临时变量的空间

运行栈中的活动记录

- 运行栈随着程序执行时发生的调用链或生长或缩小
 - ➡ 每次调用执行进栈操作，把被调函数的活动信息压入栈顶
 - ➡ 函数返回执行出栈操作，恢复到上次调用所分配的数据区
- 一个函数在运行栈上可以有若干不同的活动记录，每个都代表了一个不同的调用
 - ➡ 递归深度决定运行栈中活动记录的数目
 - ➡ 同一局部变量在不同的递归层次被分配给不同的存储空间

举例1：阶乘 $n!$

➤ 阶乘 $n!$ 的递归定义

$$factorial(n) = \begin{cases} 1, & \text{if } n \leq 0 \\ n * factorial(n-1), & \text{if } n > 0 \end{cases}$$

➤ 递归入口

➡ $factorial(n-1)$

➤ 递归出口

➡ $n \leq 0$

程序实现

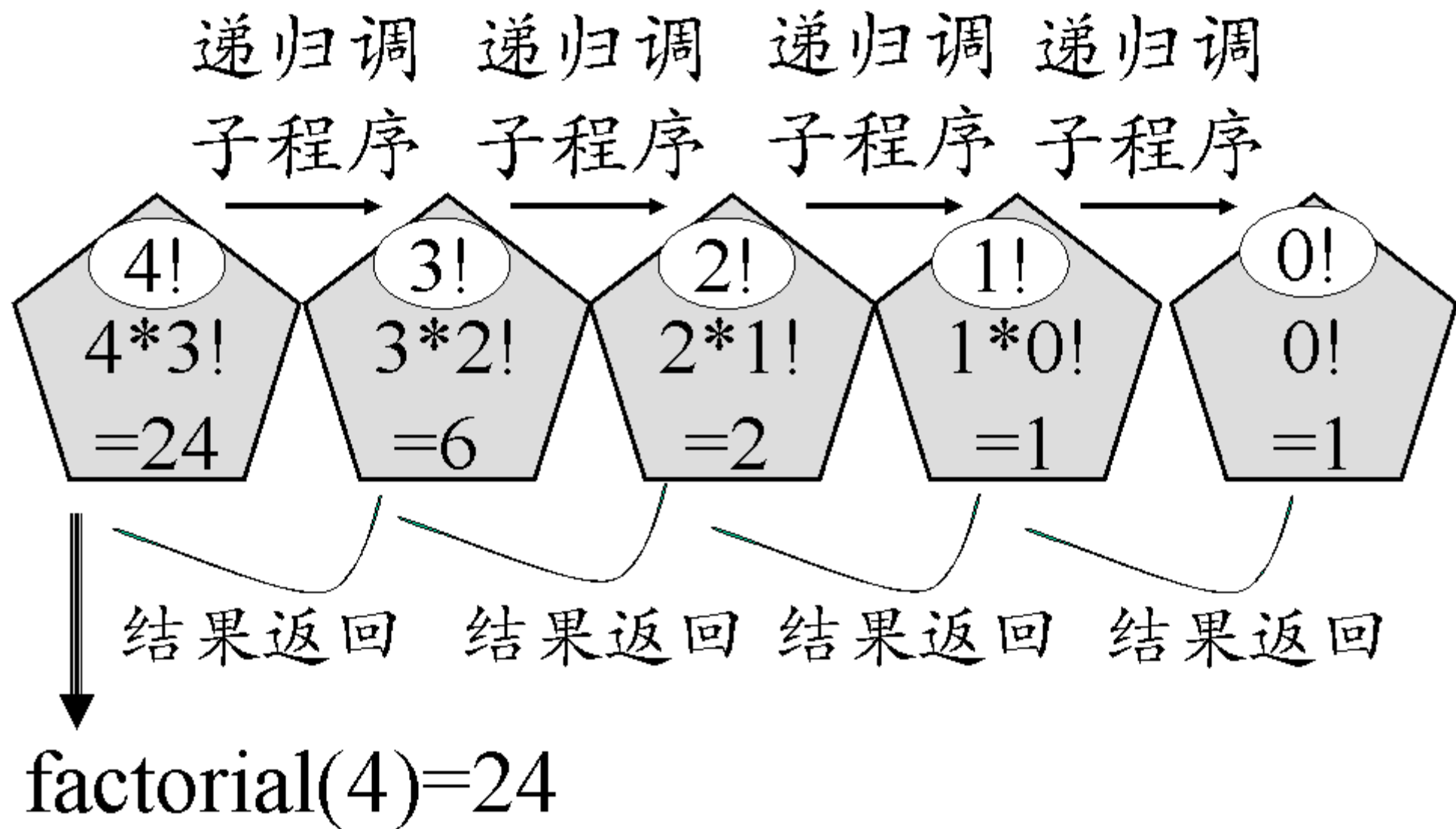
//递归函数

```
long factorial(long n)
{
    if (n==0)
        return 1;
    else
        //递归调用
        return n * factorial( n-1) ;
}
```

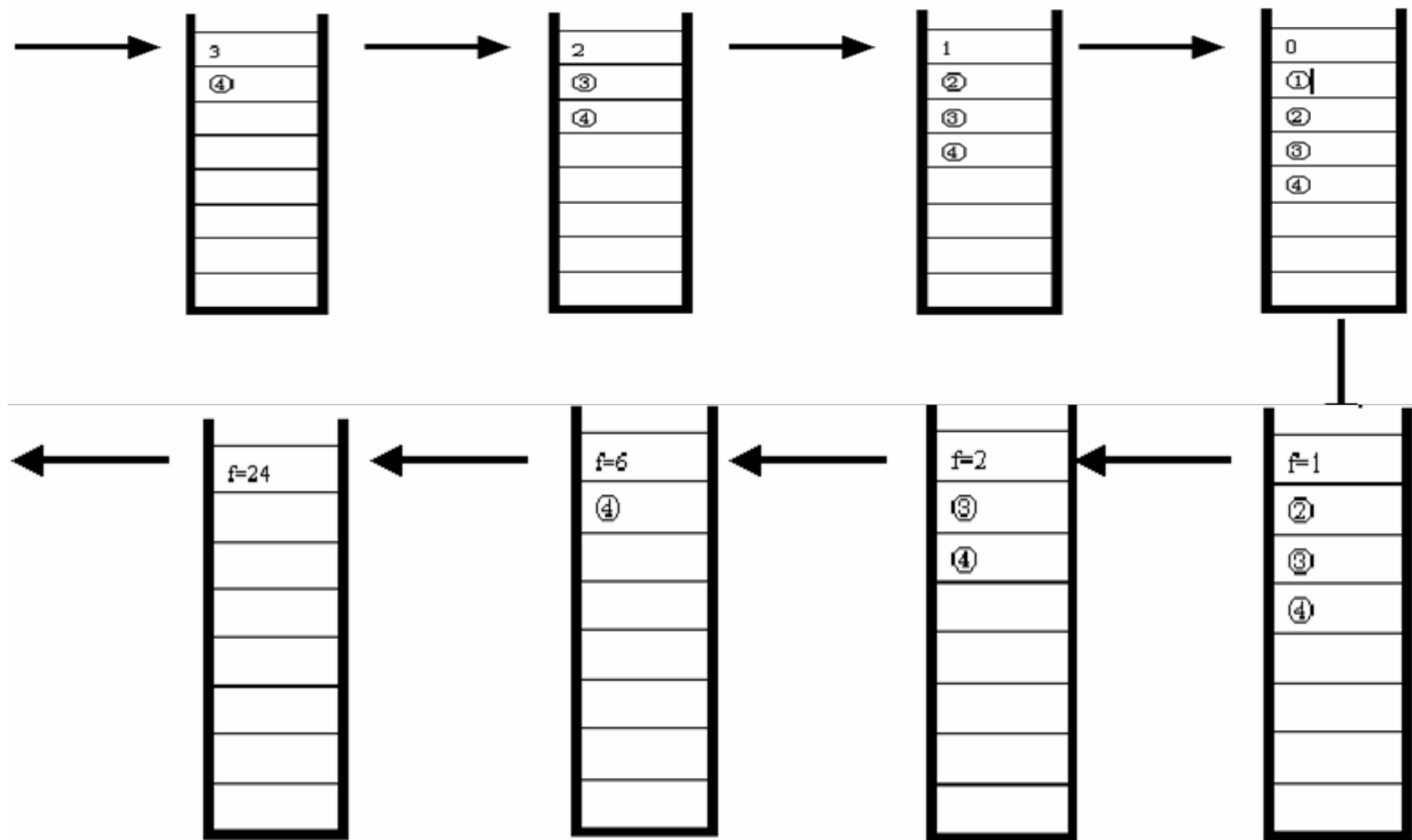
//非递归函数

```
long factorial(long n)
{
    int m = 1;
    int i ;
    if (n>0)
        for ( i = 1; i <= n; i++ )
            m = m * i ;
    return m ;
}
```

递归过程示意图(4!)



栈的变化



2、表达式的求值

➤ 表达式的组成

- 基本符号集：由{0, 1, ..., 9, +, -, *, /, (,)} 等16个基本符号组成
- 语法成分集：由{ <表达式>, <项>, <因子>, <常数>, <数字> } 5个语法成分组成
- 语法公式集：“::=”是规则定义符

➤ 一个表达式由操作数、操作符 和分界符组成

表达式的分类

➤ 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 $AB+$;

中缀表达式的递归定义

➤ 中缀计算表达式的递归定义

➡ $\langle \text{表达式} \rangle ::= \langle \text{项} \rangle + \langle \text{项} \rangle \mid \langle \text{项} \rangle - \langle \text{项} \rangle \mid \langle \text{项} \rangle$

➡ $\langle \text{项} \rangle ::= \langle \text{因子} \rangle * \langle \text{因子} \rangle \mid \langle \text{因子} \rangle / \langle \text{因子} \rangle \mid \langle \text{因子} \rangle$

➡ $\langle \text{因子} \rangle ::= \langle \text{常数} \rangle \mid (\langle \text{表达式} \rangle)$

➡ $\langle \text{常数} \rangle ::= \langle \text{数字} \rangle \mid \langle \text{数字} \rangle \langle \text{常数} \rangle$

➡ $\langle \text{数字} \rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

中缀表达式的计算次序

➤ **有括号时**：先括号内，后括号外

➤ 多层括号时，按层次反复脱括号

➤ 左右括号须配对

➤ **无括号时**：先乘($*$)、除($/$)，后加($+$)、减($-$)

➤ **同层次时**：若有多个乘除 ($*$ 、 $/$) 或加减 ($+$ 、 $-$) 时，自左至右顺序执行

后缀表达式（逆波兰式）

- $\langle \text{表达式} \rangle ::= \langle \text{项} \rangle \langle \text{项} \rangle + \mid \langle \text{项} \rangle \langle \text{项} \rangle - \mid \langle \text{项} \rangle$
- $\langle \text{项} \rangle ::= \langle \text{因子} \rangle \langle \text{因子} \rangle * \mid \langle \text{因子} \rangle \langle \text{因子} \rangle / \mid \langle \text{因子} \rangle$
- $\langle \text{因子} \rangle ::= \langle \text{常数} \rangle$
- $\langle \text{常数} \rangle ::= \langle \text{数字} \rangle \mid \langle \text{数字} \rangle \langle \text{常数} \rangle \mid$
- $\langle \text{数字} \rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

在高级语言程序设计中，使用后缀表达式进行算术表达式的求值计算！

后缀求值示例

➤ 中缀表达式

$$(23 + 34 * 45 / (5 + 6 + 7))$$

➤ 后缀表达式

23 34 45 * 5 6 + 7 + / +

The diagram illustrates the evaluation of the postfix expression. It shows the sequence of tokens: 23, 34, 45, *, 5, 6, +, 7, +, /, +. Below the tokens, there are four horizontal lines of different colors: a red line under '34', a green line under '5', a blue line under '6', and a red line under '7'. A thick black line is at the bottom.

后缀表达的计算规则

- 后缀表达式不含括号!
- 运算符放在两个参与运算的语法成分的后面
- 所有求值计算皆按运算符出现的顺序，严格从左向右进行。

如何转换？

➤ 在中缀和后缀表达式中

- ➡ 操作数的次序保持不变、操作符的次序发生改变

➤ 后缀表达式中相邻操作符的运算次序受下列因素影响

- ➡ 先括号内、后括号外

- ➡ 先乘除而后加减

- ➡ 同级别先左后右

➤ 后缀表达式中，操作符的次序要与中缀表达式中的计算次序保持一致！！

如何保持一致？借助栈进行排序

中缀到后缀表达式的转换

- 当输入是操作数，直接输出到后缀表达式序列
- 当输入的是左括号时，也把它压栈
- 当输入的是运算符时

■ While

- If (**栈非空 and 栈顶不是左括号 and 输入运算符的优先级 “ \leq ” 栈顶运算符的优先级**) 时
 - 弹栈，并放到后缀表达式序列中
- Else 把输入的运算符压栈 (**>当前栈顶运算符才压栈!**)

…转换(续)

➤ 当输入的是右括号时，先判断栈是否为空

➡ 若为空（括号不匹配），异常处理，清栈退出；

➡ 若非空，则把栈中的元素依次弹出

— 遇到第一个左括号为止，将弹出的元素输出到后缀表达式的序列中（弹出的开括号不放到序列中）

— 若没有遇到开括号，说明括号也不匹配，做异常处理，清栈退出

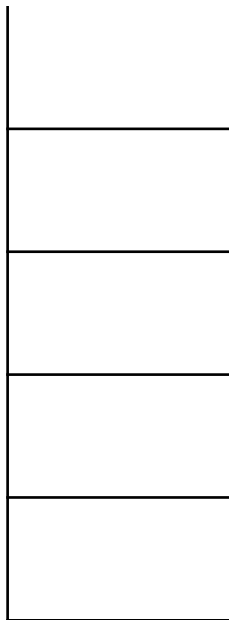
…转换(续)

- **最后，当中缀表达式的符号序列全部读入时，若栈内仍有元素，把它们全部依次弹出，都放到后缀表达式序列尾部**
 - ➡ **若弹出的元素遇到括号时，则说明括号不匹配，做错误异常处理，清栈退出**

中缀表达式→后缀表达式

输入中缀表达式: $23 + (34 * 45) / (5 + 6 + 7)$

栈的状态



输出后缀表达式:

后缀表达式求值

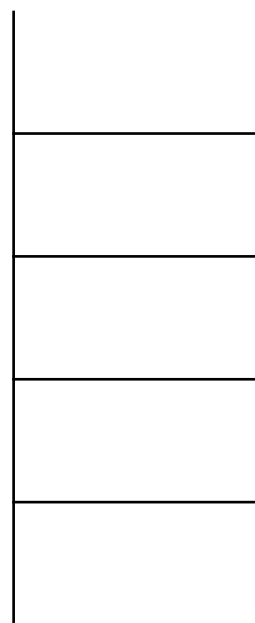
- **循环：依次顺序读用户键入的符号序列，组成并判别语法成分类别**
 - **1.当遇到的是操作数，则压入栈顶；**
 - **2.当遇到的是运算符，就从栈中两次取出栈顶，按照运算符对这两个操作数进行计算。然后将计算结果压入栈顶。**
- **如此继续，直到遇到符号 =，这时栈顶的值就是输入表达式的值。**

后缀表达式求值

待处理后缀表达式:

23 34 45 * 5 6 + 7 + / +

栈状态的变化



1530
11
18
85
108

计算器类的实现

// Class Declaration 类的说明

```
class Calculator {
```

```
private:
```

```
    Stack<double> s;    // 这个栈用于压入保存操作数
```

```
    bool GetTwoOperands(double& opd1,double& opd2); // 从弹出两个操作数
```

```
    void Compute(char op); // 取两个操作数，并按op对两个操作数进行计算
```

```
public:
```

```
    Calculator(void){} ; // 创建计算器实例，开辟一个空栈
```

```
    void Run(void);      // 读入后缀表达式，遇到符号"="时，求值计算结束
```

```
    void Clear(void);    // 计算器的清除，为随后的下一次计算做准备
```

```
};
```

部分成员函数的程序实现

```
bool Calculator::GetTwoOperands(double& opd1, double& opd2) {  
    if (s.isEmpty()) {  
        cerr << "Missing operand!" <<endl;  
        return false;  
    }  
    s.pop(&opd1);                                // 右操作数  
    if (s.isEmpty()) {  
        cerr << "Missing operand!" <<endl;  
        return false;  
    }  
    s.pop(&opd2)      ;                          // 左操作数  
    return true;  
}
```



```
void Calculator::Compute (char op)
```

```
{
```

```
    Boolean result;
```


```
    double operand1, operand1;
```

```
    result = GetTwoOperands(operand1,operand2);
```

```
    if(result == True)
```

```
        switch(op)
```

```
        {
```



```
case '+': S.Push(operand2 + operand1);
        break;
case '-': S.Push(operand2 - operand1);
        break;
case '*': S.Push(operand2 * operand1);
        break;
case '/': if(operand1 == 0.0)
{
    cerr<<"Divided by 0!"<<endl;
    S.ClearStack();
}
else
    S.Push(operand2 / operand1);
break;
}
else
    S.ClearStack();
}
```



void Calculator::Run(void)//读入表达式，遇到操作数则入栈，遇到操作符则从栈中连续弹出两个操作数计算，直至遇到 “=”结束。 {

char c;

double newoperand;

while(cin>>c, c != '=') {

case '+':

case '-':

case '*':

case '/':

Compute(c);

break;


default:

cin >> newoperand;

push(newoperand);

break;

}



```
    if (!S.IsEmpty())  
        cout << S.top() << endl; //印出求值的最后结果  
} //end calculator::run(void)
```

```
void Calculator::Clear(void)
```

```
{  
    S.ClearStack();  
}
```

[后缀计算器的类定义，结束]

3.2 队列(Queue)

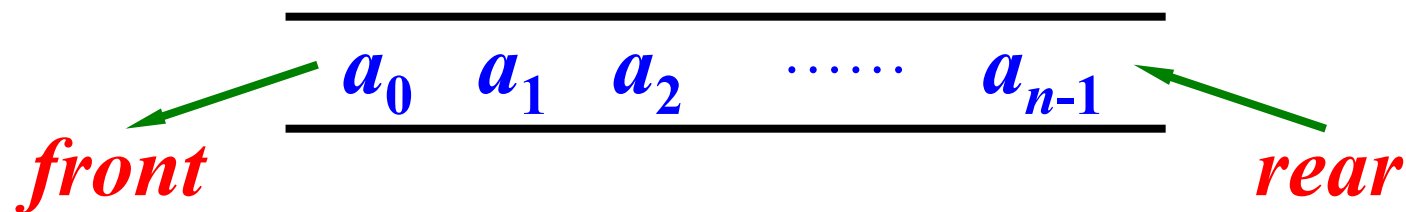
➤ 队列是只允许在一端删除，另一端插入的线性表

➡ 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**, **F**irst **I**n **F**irst **O**ut)



队列的主要操作

- 入队列 (enqueue)
- 出队列 (dequeue)
- 取队首元素 (getFront)
- 判断队列是否为空 (isEmpty)

抽象数据类型

```
template <class T>
class Queue {
public:                                // 队列的运算集
    void clear();                     // 变为空队列
    bool enqueueer (const T item);
        // 将item插入队尾, 成功返回真, 否则返回假
    bool dequeueer (T& item);
        // 返回队头元素并将其从队列中删除, 成功则返回真
    bool getFront (T& item);
        // 返回队头元素, 但不删除, 成功则返回真
    bool isEmpty ();                  // 返回真, 若队列已空
    bool isFull ();                   // 返回真, 若队列已满
};
```

队列的物理实现

➤ 顺序队列

➡ 顺序表实现

➤ 链式队列

➡ 单链表实现

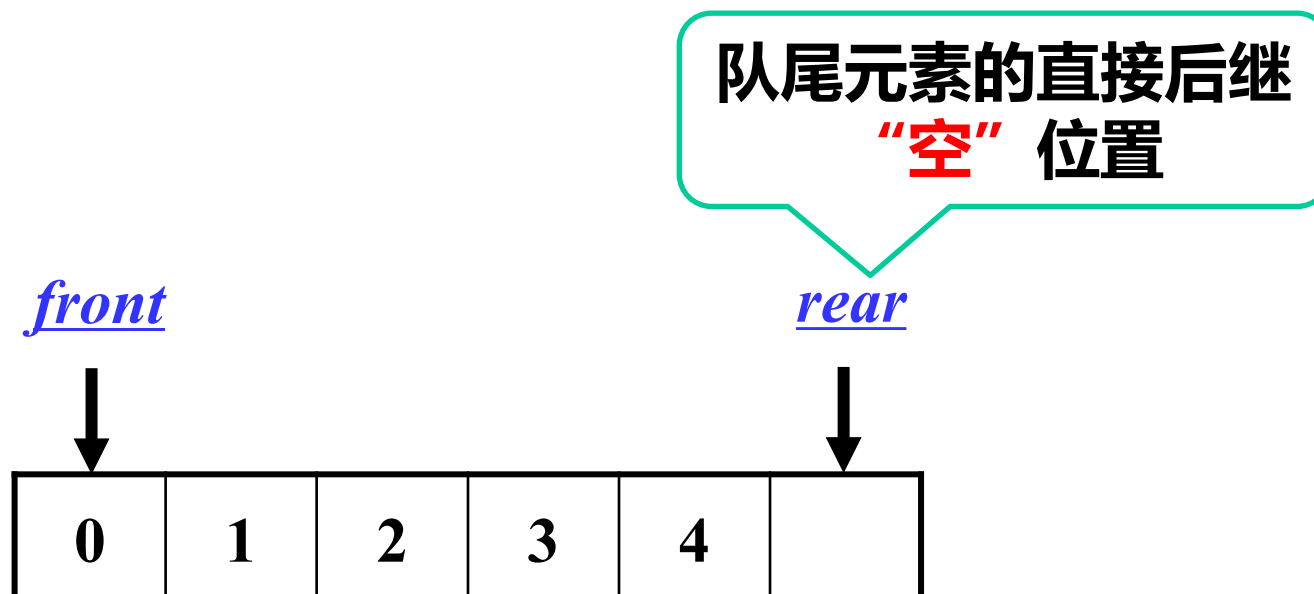
3.2.1 顺序队列

➤ 用向量存储队列，两个变量分别指向首、尾两端

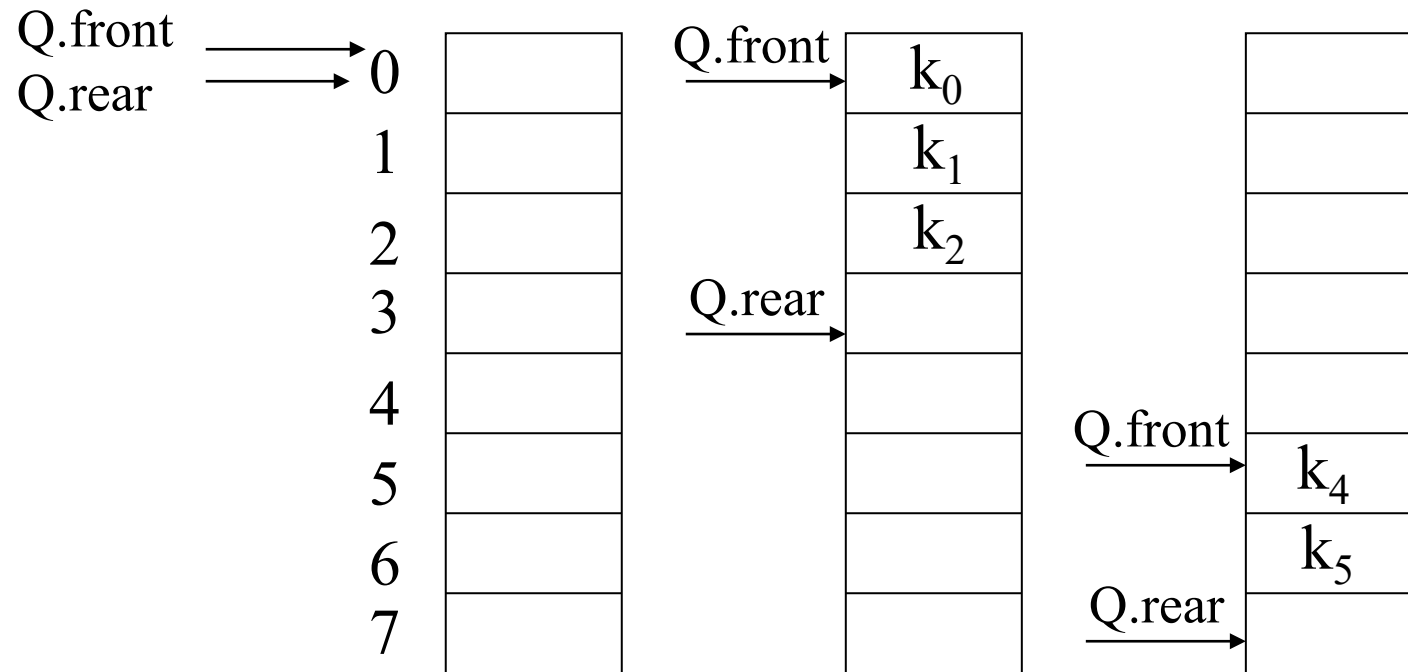
➡ **首 (front)**: 指向当前待出队元素位置 (实的!)

➡ **尾 (rear)**: 指向当前待入队元素位置 (虚的!)

— 队列中始终有一个空闲的虚位置



队列示意



队列空

再进队一个元素如何?

队列的溢出

➤ 上溢

- ➡ 当队列满时，再做进队操作，所出现的现象

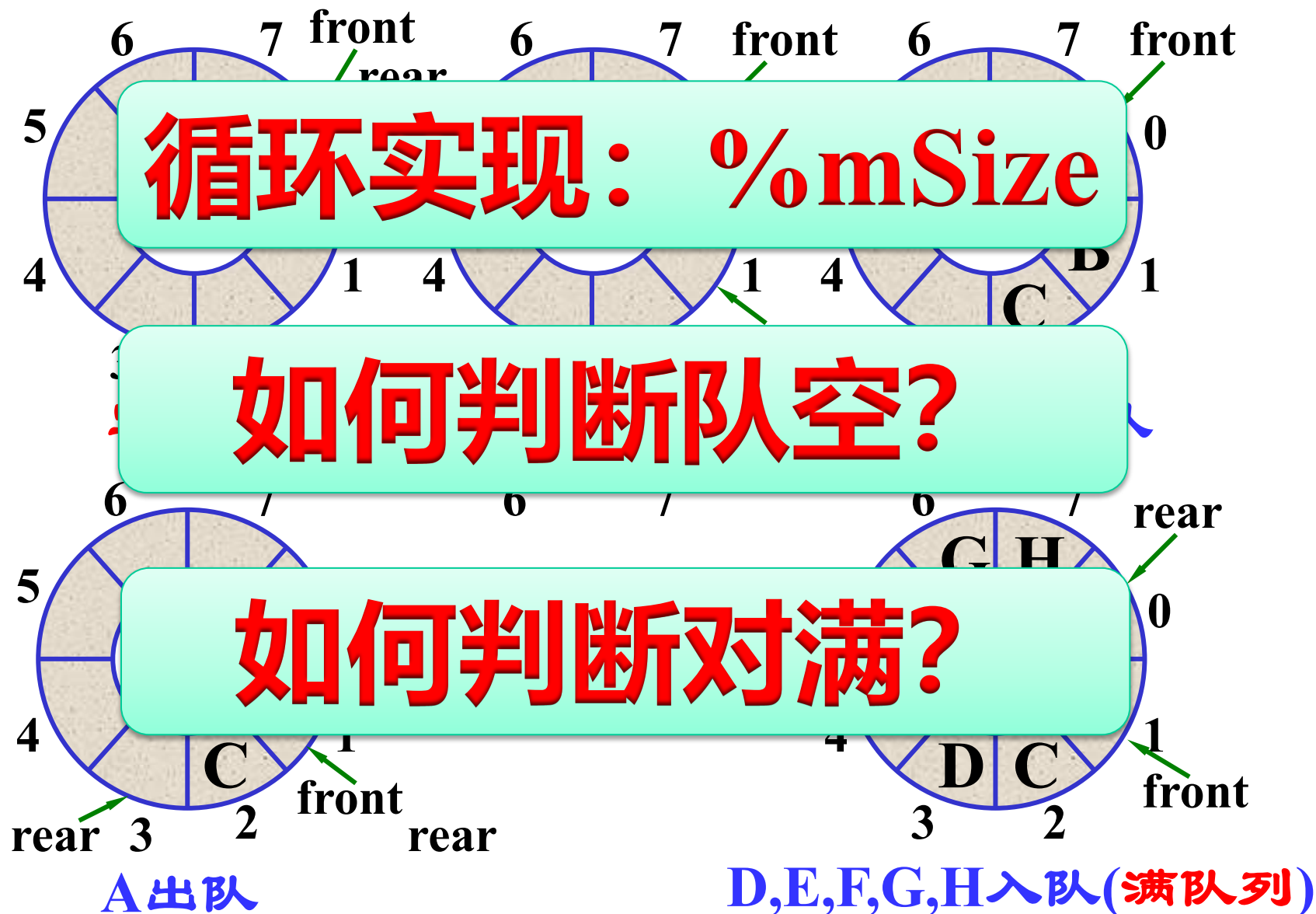
➤ 下溢

- ➡ 当队列空时，再做删除操作，所出现的现象

➤ 假溢出

- ➡ 当 $\text{rear} = \text{mSize}-1$ 时，再作插入运算就会产生溢出，如果这时队列的前端还有许多空位置，这种现象称为假溢出

循环队列运行示意图



队列的类定义

```
class arrQueue: public Queue<T> {  
    private:  
        int      mSize;           // 存放队列的数组的大小  
        int      front;          // 表示队头所在位置的下标  
        int      rear;           // 表示待入队元素所在位置的下标  
        T        *qu;            // 存放类型为T的队列元素的数组  
  
    public:                          // 队列的运算集  
        arrQueue(int size) {       // 创建队列的实例  
            mSize = size + 1;      // 多申请一个存储空间，以区别队空和队满  
            qu = new T [mSize];  
            front = rear = 0;  
        }  
        ~arrQueue() {              // 消除该实例，并释放其空间  
            delete [] qu;  
        }  
}
```

入队列操作

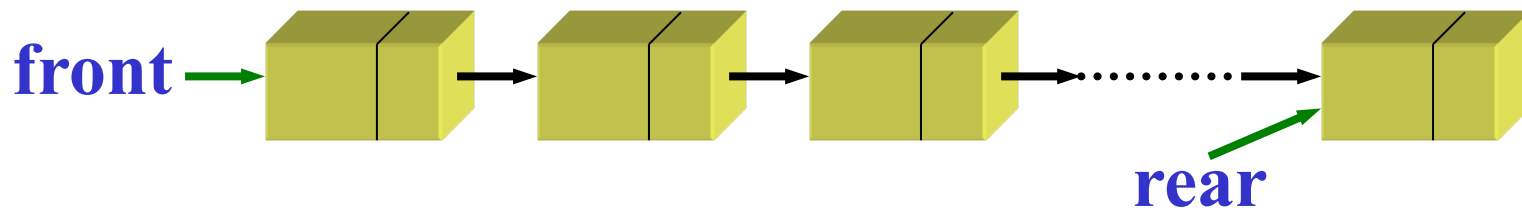
```
bool arrQueue<T> :: enqueue(const T item) {  
    // item入队, 插入队尾  
    if ((((rear + 1) % mSize) == front)) {  
        cout << "队列已满, 溢出" << endl;  
        return false;  
    }  
    qu[rear] = item;  
    rear = (rear + 1) % mSize;           // 循环后继  
    return true;  
}
```

出队列操作

```
bool arrQueue<T> :: deQueue(T& item) {  
    // 返回队头元素并从队列中删除  
    if ( front == rear) {  
        cout << "队列为空" << endl;  
        return false;  
    }  
    item = qu[front];  
    front = (front + 1) % mSize;  
    return true;  
}
```

3.2.2 链式队列

- 单链表队列：链接指针的方向是从队头指向队尾
- 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： $\text{front} == \text{rear} == \text{NULL}$



队列的类定义

```
template <class T>
class lnkQueue: public Queue <T> {
private:
    int    size;           // 队列中当前元素的个数
    Link<T>* front;        // 表示队头的指针
    Link<T>* rear;         // 表示队尾的指针
public:                    // 队列的运算集
    lnkQueue(int size) {   // 创建队列的实例
        size = 0;
        front = rear = NULL;
    }
    ~lnkQueue() {         // 消除该实例，并释放其空间
        clear();
    }
}
```

运算集： 入队

```
bool lnkQueue <T> :: enqueue(const T item) { // 入队插入队尾
    if (rear == NULL) { // 空队列
        front = rear = new Link<T> (item, NULL);
    }
    else { // 添加新的元素
        rear-> next = new Link<T> (item, NULL);
        rear = rear ->next;
    }
    size++;
    return true;
}
```

运算集：出队

// 返回队头元素并从队列中删除

```
bool lnkQueue <T> :: deQueue(T& item) {  
    Link<T> *tmp;  
    if (size == 0) {                // 队列为空, 没有元素可出队  
        cout << "队列为空" << endl;  
        return false;  
    }  
    &item = front->data;  
    tmp = front;  
    front = front -> next;  
    delete tmp;  
    if (front == NULL)  
        rear = NULL;  
    size--;  
    return true;  
}
```

队列的应用

- 只要满足先来先服务特性的应用均可采用队列作为其数据组织方式或中间数据结构。
- 调度或缓冲
 - ➡ 消息缓冲器
 - ➡ 邮件缓冲器
 - ➡ 计算机的硬设备之间的通信也需要队列作为数据缓冲
 - ➡ 操作系统的资源管理
- 宽度优先搜索

顺序队列与链式队列的比较

➤ 变种的栈或队列结构

- ➡ **双端队列**：在队首和队尾都可以插入、删除的队列
- ➡ **双栈**：两个底部相连的栈，共享一块数据空间
- ➡ **超队列**：是一种被限制的双端队列，删除操作只允许在一端进行，插入操作却可以在两端同时进行
- ➡ **超栈**：是一种插入受限的双端队列，即插入限制在一端而删除却允许在两端进行

思考题：栈和队列的互模拟

- 如何用两个栈模拟一个队列？
- 如何用两个队列模拟一个栈？



再见…

联系信息:

电子邮件: gjsong@pku.edu.cn

电 话: 62754785

办公地点: 理科2号楼2307室