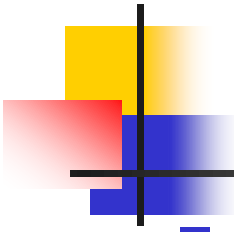
Stacks and Queues

- 栈(Stack)
  - 基本概念
  - 顺序存储结构
  - 链式存储结构
  - 应用

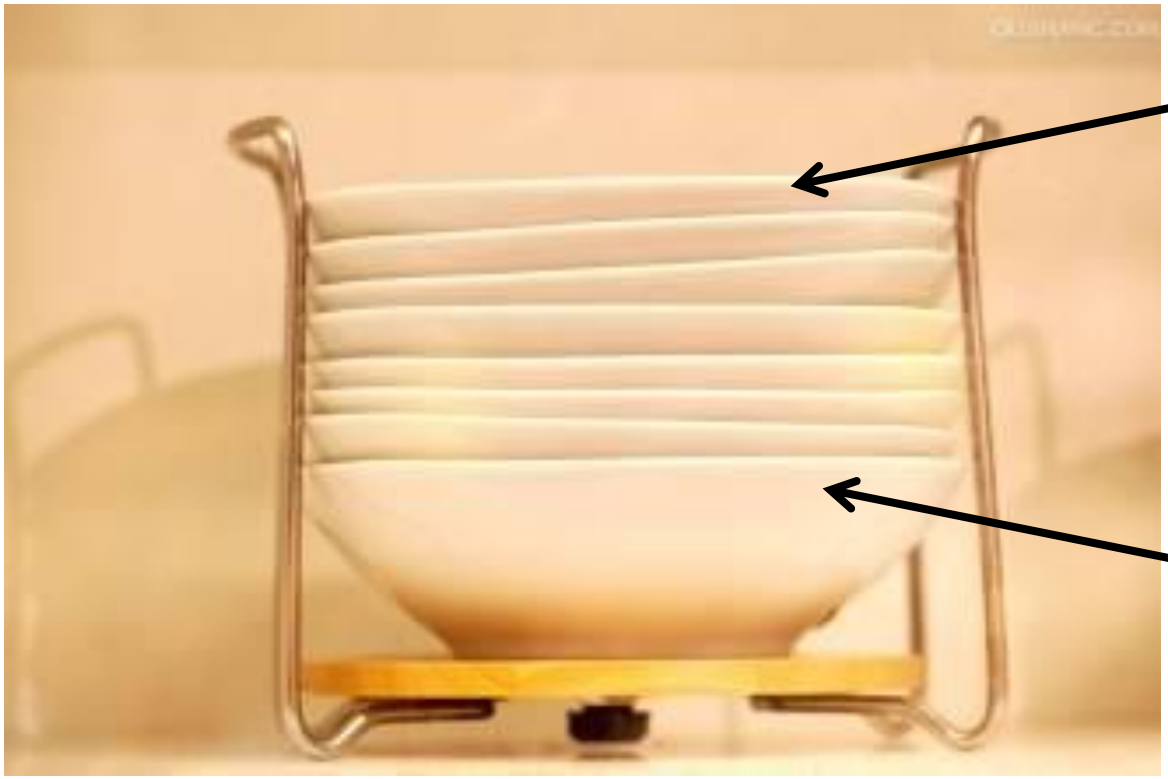
- 队列（Queue）
  - -- 基本概念
  - – 顺序存储结构
  - – 链式存储结构
  - – 应用



Stack

- Stack is
  - an ordered list.
  - in insertion known as push.
  - in remove known as pop.
  - made at one end called the top.
- Stack is also known as a LIFO list
  - LIFO: Last-In-First-Out
  - Additions to and removals from the top end only
- Give a stack $S=(a_0,\ldots,a_{n-1})$
  - a_0: is the bottom element, a_{n-1}: is the top element

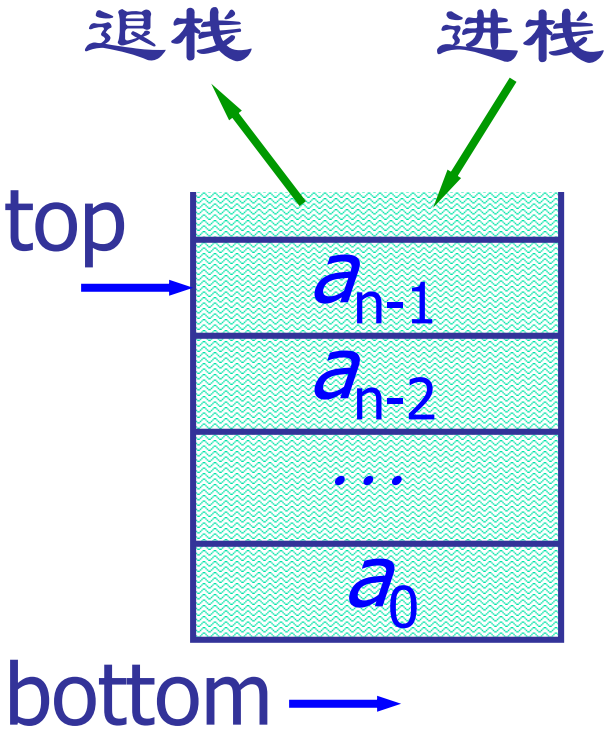
Stack Of Cups



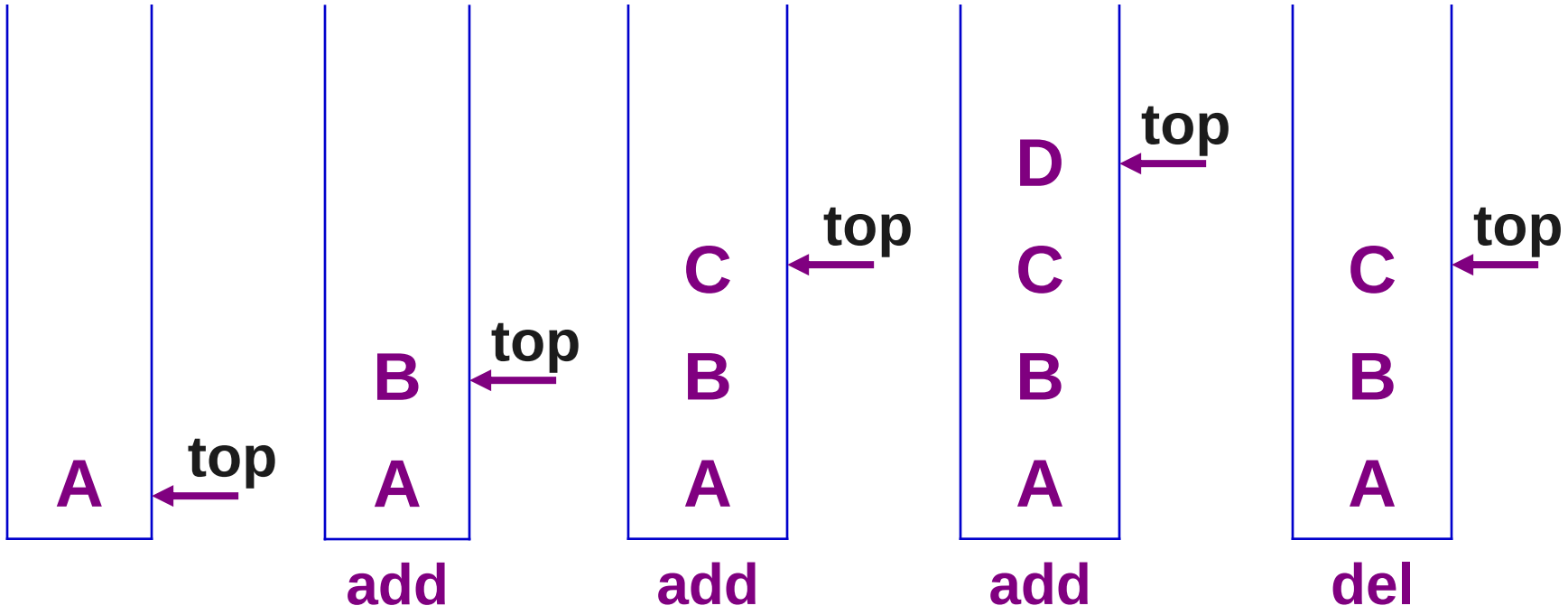
Stack

- 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)



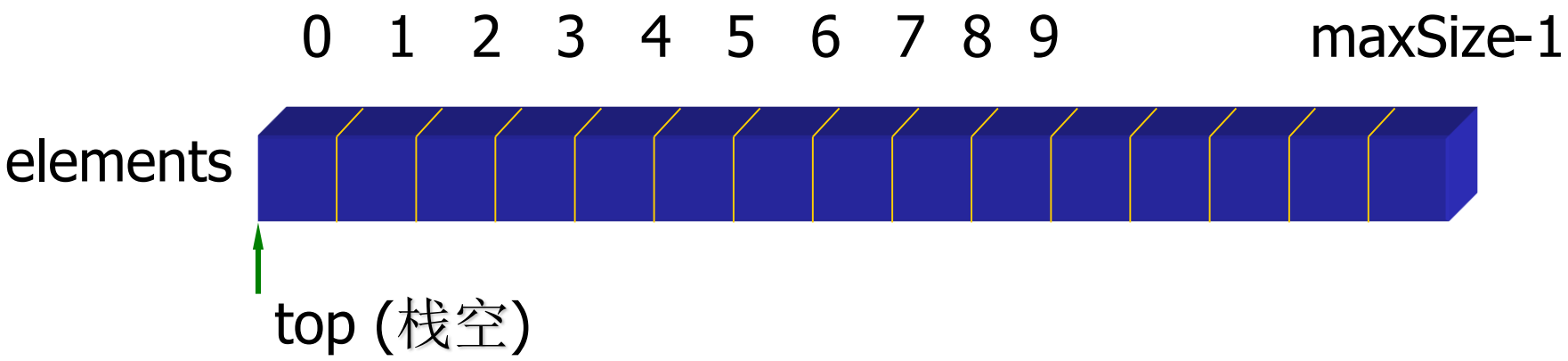
Inserting and deleting elements in a stack

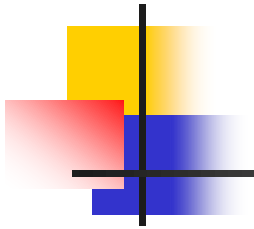


ADT of Stack

```
template <class E>
class Stack {                                  //栈的类定义
public:
    Stack(){ };                                //构造函数
    virtual void Push(E x) = 0;                //进栈
    virtual bool Pop(E& x) = 0;                //出栈
    virtual bool getTop(E& x) = 0;             //取栈顶
    virtual bool IsEmpty() = 0;                //判栈空
    virtual bool IsFull() = 0;                 //判栈满
};
```

栈的数组存储表示 —— 顺序栈





To implement STACK ADT, we can use

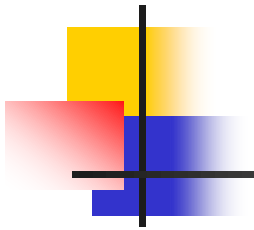
- **an array**
- **a variable top**

Initially top is set to

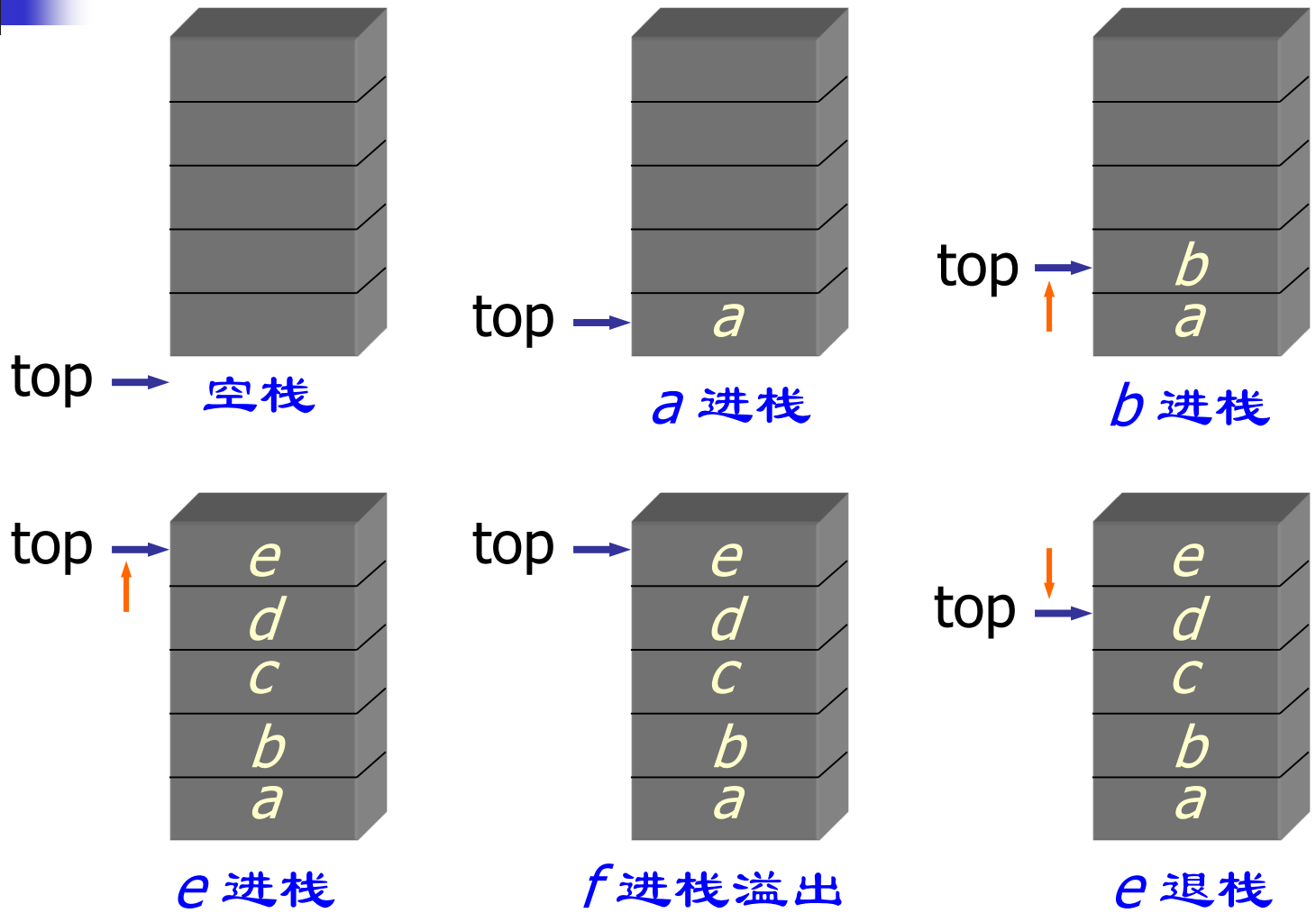
–1.

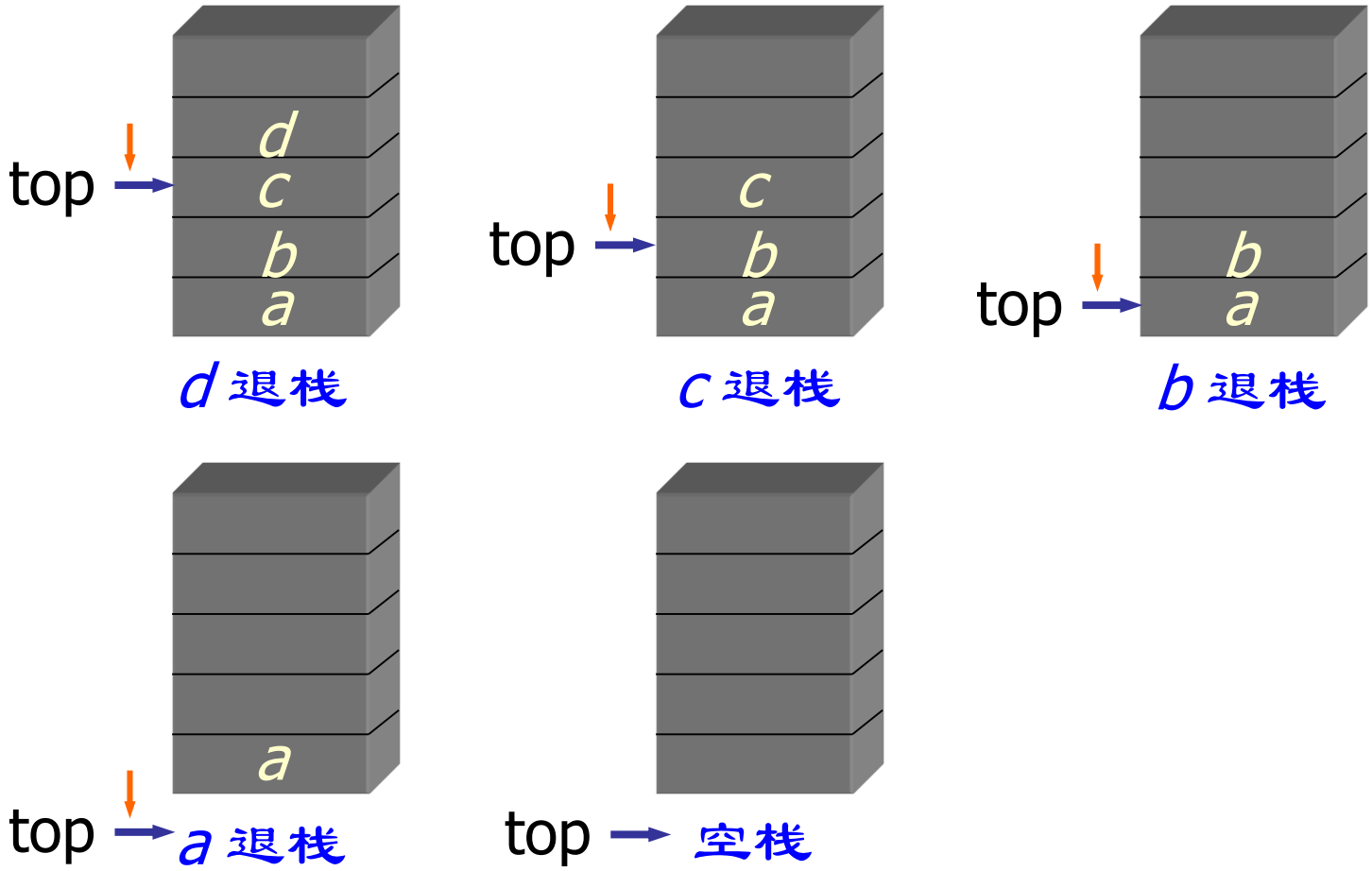
顺序栈类的定义

```
#include <assert.h>
#include <iostream.h>
#include "stack.h"

template <class E>
class SeqStack : public Stack<E>
{   //顺序栈类定义
private:
    E *elements;                    //栈元素存放数组
    int top;                        //栈顶指针
    int maxSize;                    //栈最大容量
    void overflowProcess(); //栈的溢出处理
```



```
public:
    SeqStack(int sz =50);              //构造函数
    ~SeqStack() { delete []elements; }  //析构函数

    void Push(E x);                          //进栈
    bool Pop(E& x);           //出栈
    bool getTop(E& x);                 //取栈顶内容
    bool IsEmpty() const { return top == -1; }
    bool IsFull() const { return top == maxSize-1; }
};
```


top
空栈
top
a
a 进栈
top
b
a
b 进栈
top
e
d
c
b
a
e 进栈
top
e
d
c
b
a
f 进栈溢出
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

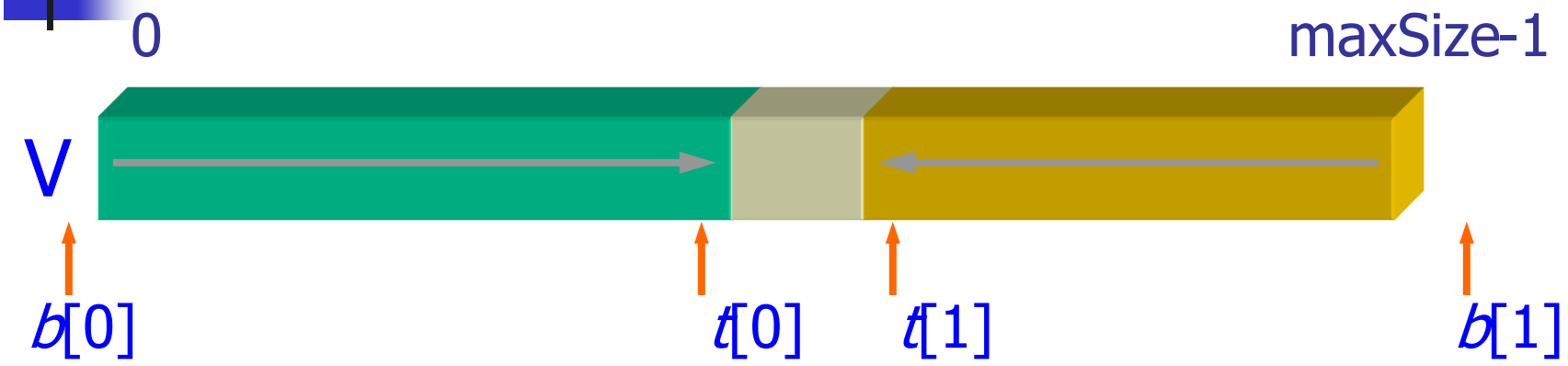
```
template <class E>
void SeqStack<E>::Push(E x) {
//若栈不满, 则将元素x插入该栈栈顶, 否则溢出处理
    if (IsFull() == true) overflowProcess;          //栈满
    elements[++top] = x;          //栈顶指针先加1, 再进栈
};

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
};
```

栈满操作

```
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>
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;
};
```

双栈共享一个栈空间



两个栈共享一个数组空间V[maxSize]

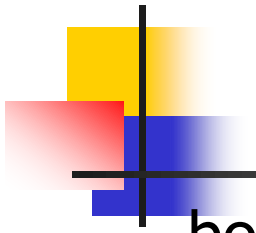
设立栈顶指针数组 t[2] 和栈底指针数组 b[2]

t[i]和b[i]分别指示第 i 个栈的栈顶(top)与栈底(bottom)

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] //退到栈底

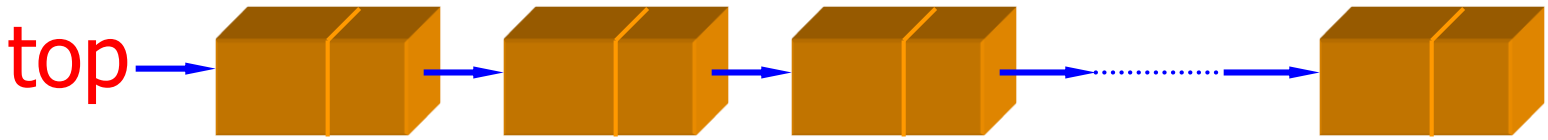


```
bool push(DualStack& DS, Type x, int i) {
    if (DS.t[0]+1 == DS.t[1]) return false;
    if (i == 0) DS.t[0]++; else DS.t[1]--;
    DS.V[DS.t[i]] = x;
    return true;
}

bool Pop(DualStack& DS, Type& 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;
}
```

共享栈的好处

- 节省存储空间
  - 共享存储空间
- 降低发生上溢的可能
  - 插入和删除都在栈顶操作，
  - 只可能发生栈顶指针超出最大范围的上溢问题
- 不存在减少存取时间的问题
  - 获取栈中元素的Tn=O(1)

栈的链接存储表示 — 链式栈

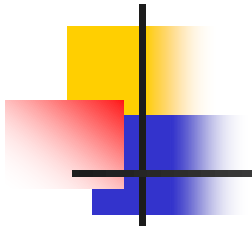


- 链式栈无栈满问题，空间可扩充
- 链式栈的栈顶在链头
- 插入与删除仅在栈顶处执行
- 适合于多栈操作

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
    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(); }   //析构函数
    void Push(E x);                             //进栈
    bool Pop(E& x);                         //退栈
```



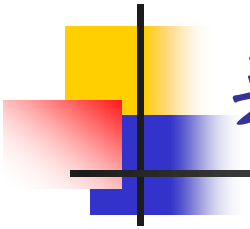
```
    bool getTop(E& x) const;                    //取栈顶
    bool IsEmpty() const { return top == NULL; }
    void makeEmpty();                           //清空栈的内容

    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 <<
};
```

```
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;
};
```

栈空操作

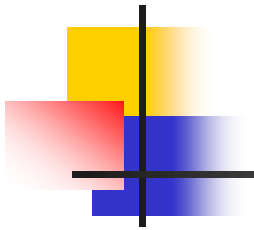
```
template <class E>
LinkedStack<E>::makeEmpty() {
//逐次删去链式栈中的元素直至栈顶指针为空。
    StackNode<E> *p;
    while (top != NULL)              //逐个结点释放
        { p = top;  top = top->link;  delete p; }
};
```

```
template <class E>
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
      }
};
```



关于栈的进一步讨论

- 问题：当进栈元素的编号为1, 2, …, n时，可能的出栈序列有多少种？



- 设进栈元素数为n，可能出栈序列数为m_n:
  - n = 0，m_0 = 1: 出栈序列{}。
  - n = 1，m_1 = 1: 出栈序列{1}。
  - n = 2，$m_2 = 2 = m_0*m_1+m_1*m_0$
  - 出栈序列中1在第1位。1进 1出 2进 2出，出栈序列为{1, 2}。 $m_0*m_1= 1$
  - 出栈序列中1在第2位。1进 2进 2出 1出，出栈序列为{2, 1}。 $m_1*m_0 = 1$

- n = 3，$m_3 = 5 = m_0*m_2+m_1*m_1+m_2*m_0$

a) 出栈序列中1在第1位。后面2个元素有$m_2 = 2$个出栈序列：{1, 2, 3}, {1, 3, 2}。

$m_0*m_2 = 2$

b) 出栈序列中1在第2位。1前有2后有3，出栈序列为{2, 1, 3}。 $m_1*m_1 = 1$

c) 出栈序列中1在第3位。前面2个元素有$m_2 = 2$个出栈序列：{2, 3, 1}, {3, 2, 1}。

$m_2*m_0 = 2$

- ✓ n = 4，$m_4 = 14 = m_0*m_3+m_1*m_2+m_2*m_1+m_3*m_0$

a) 出栈序列中1在第1位。后面3个元素有$m_3 = 5$个出栈序列：{1, 2, 3, 4}, {1, 2, 4, 3}, {1, 3, 2, 4}, {1, 3, 4, 2}, {1, 4, 3, 2}。 $m_0*m_3 = 5$

b) 出栈序列中1在第2位。前面有2，后面3、4有$m_2 = 2$个出栈序列: {2, 1, 3, 4}, {2, 1, 4, 3}。 $m_1*m_2 = 2$

c) 出栈序列中1在第3位。前面2、3有$m_2 = 2$个出栈序列，后面有4: {1, 4, 3, 2}, {2, 4, 3, 1}。 $m_2*m_1 = 2$

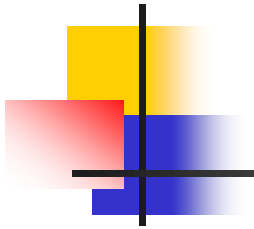
d) 出栈序列中1在第4位。前面3个元素有$m_3 = 5$个出栈序列：{2, 3, 4, 1}, {2, 4, 3, 1}, {3, 2, 4, 1}, {3, 4, 2, 1}, {4, 3, 2, 1}。 $m_3*m_0 = 5$

- 一般地，设有 n 个元素按序号1, 2, …, n 进栈，轮流让 1在出栈序列的第1, 第2, …第n位，则可能的出栈序列数为：

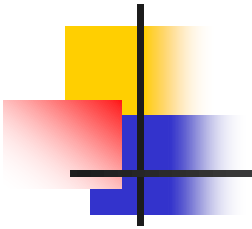
$$\sum_{i=0}^{n-1} m_i * m_{n-i-1} = m_0 * m_{n-1} + m_1 * m_{n-2} + \cdots + m_{n-1} * m_0$$

推导结果为：

$$\sum_{i=0}^{n-1} m_i * m_{n-i-1} = \frac{1}{n+1} C_{2n}^{n}$$



栈的应用：
表达式求值
Evaluation of Expressions

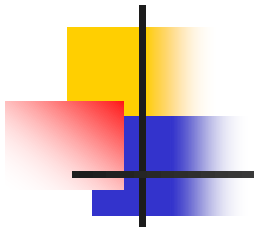


- 一个表达式由操作数（亦称运算对象）、操作符和分界符组成。
- 算术表达式有三种表示：
  - 中缀(infix)表示

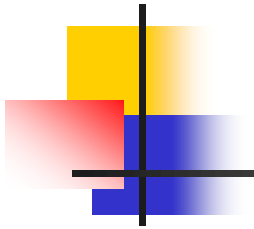
    <操作数> <操作符> <操作数>，如 A+B；
  - 前缀(prefix)表示

    <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+；
- 例如：
  - 中缀表达式　a + b * (c – d) – e / f
  - 后缀表达式　a b c d – * + e f / –
  - 前缀表达式　– + 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

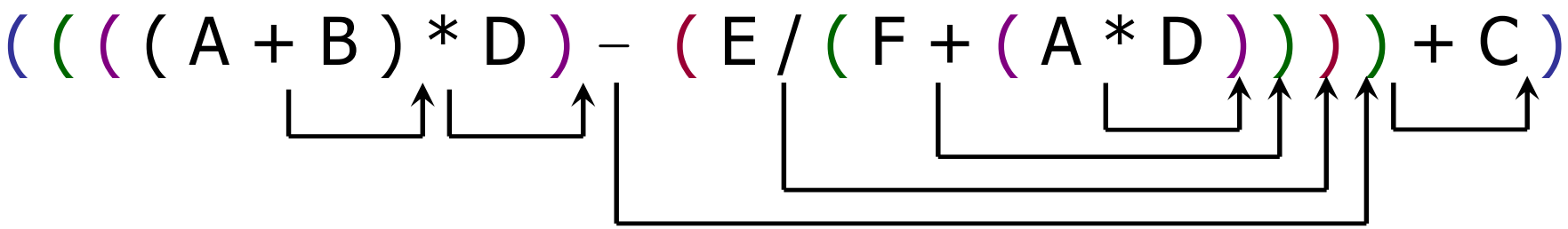


- 一般表达式的操作符有4种类型:
  - 算术操作符: 如双目操作符（+、–、*、/ 和%）以及单目操作符（-）。
  - 关系操作符: 包括<、<=、==、!=、>=、> 。这些操作符主要用于比较。
  - 逻辑操作符: 如与(&&)、或(||)、非(!)
  - 括号‘(’和‘)’: 它们的作用是改变运算顺序

中缀表示转后缀表示

- 对中缀表达式按运算优先次序加上括号
- 以就近移动为原则,把操作符后移到右括号的后面
- 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：

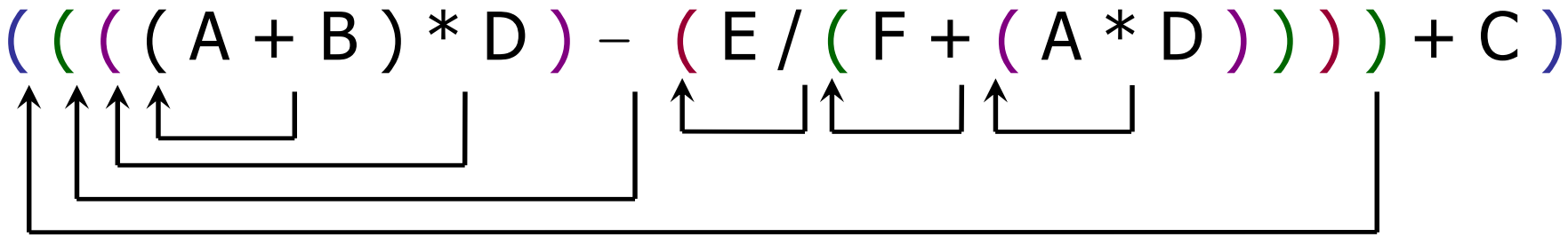


后缀表示 A B + D * E F A D * + / - C +

中缀表示转前缀表示

- 对中缀表达式按运算优先次序通统加上括号
- 以就近移动为原则,把操作符前移到左括号前
- 将所有括号消去

- 如中缀表示 (A+B)*D-E/(F+A*D)+C，其转换为前缀表达式的过程如下：



前缀表示 + - * + A B D / E + F * A D C