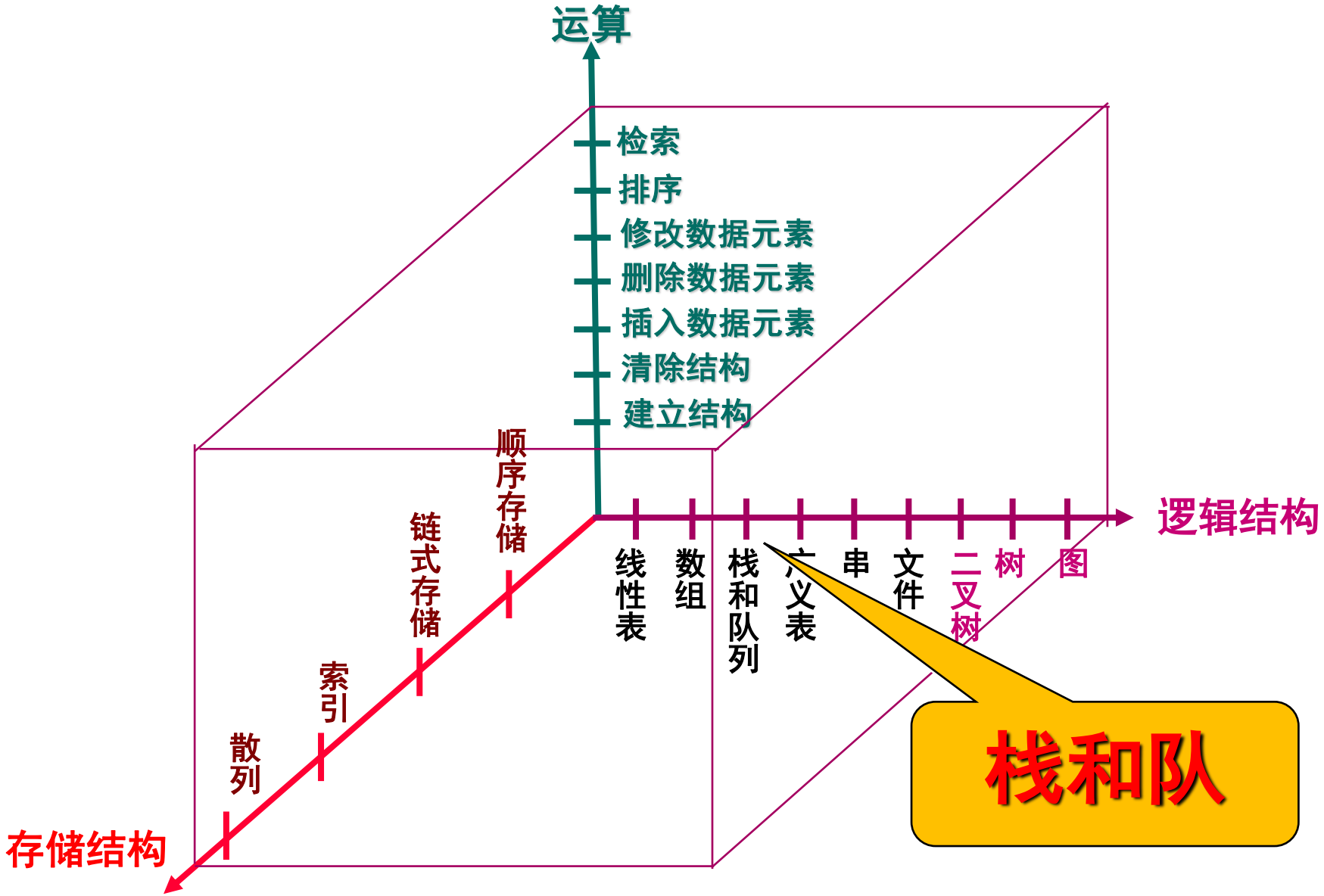
数据结构与程序设计
(Data Structure and Programming)

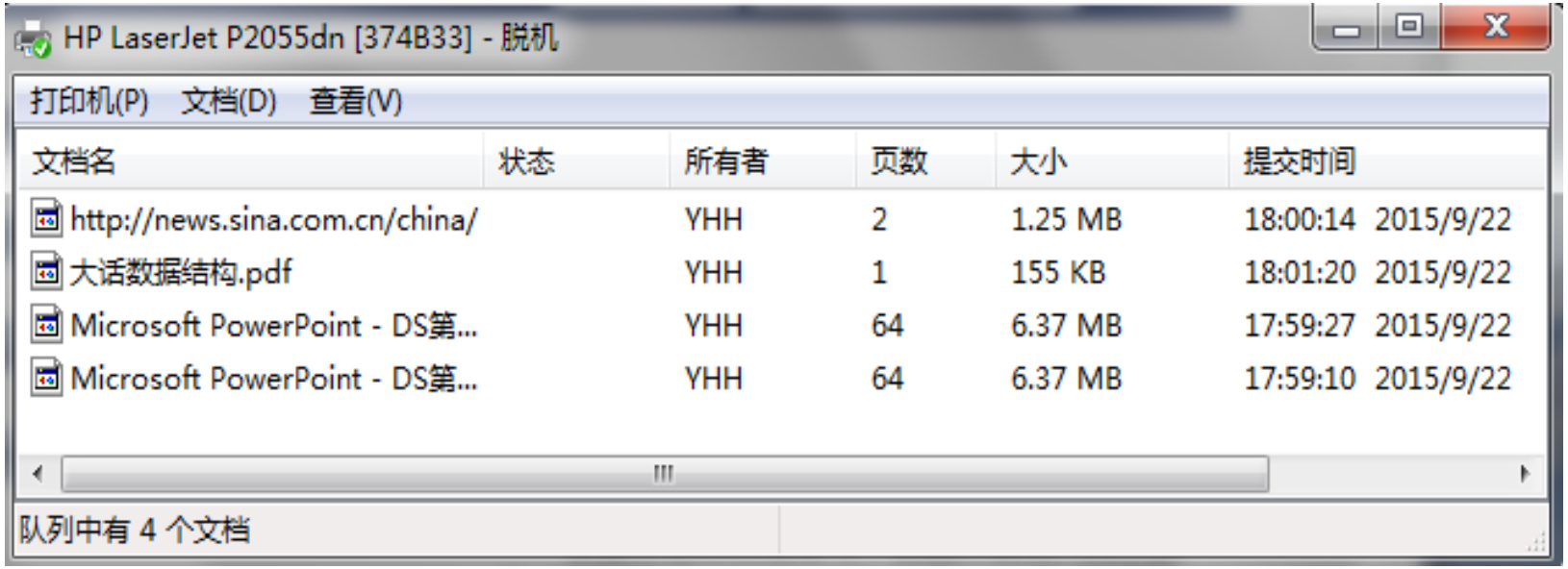
栈与队
(Stack and Queue)

北航计算机学院 林学练

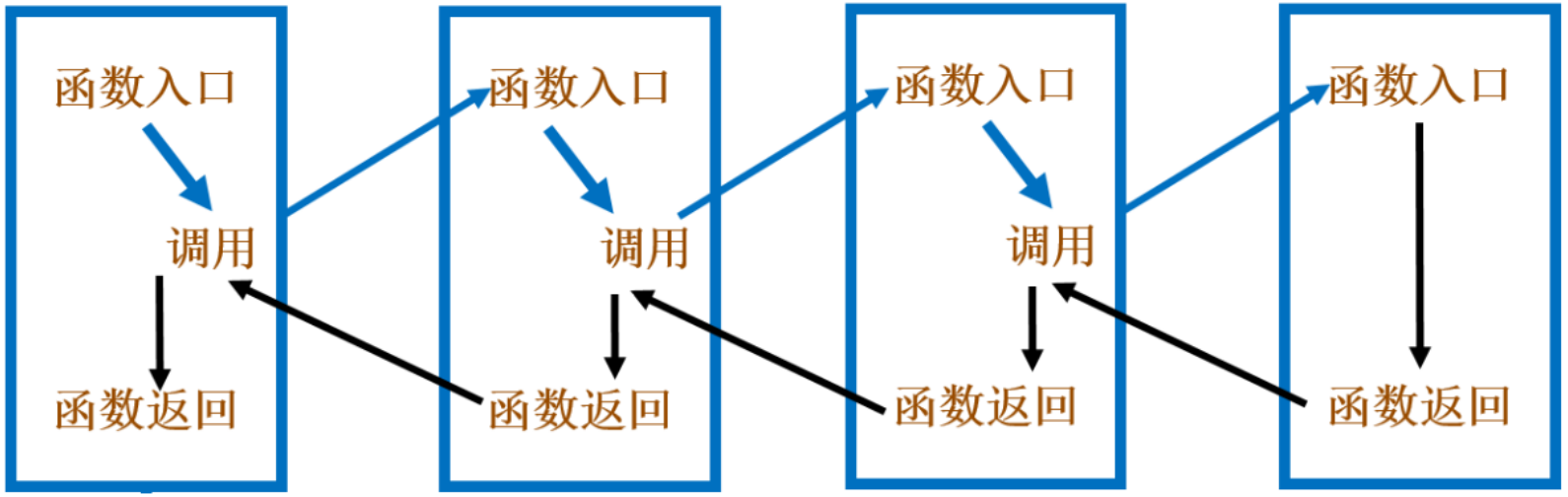
数据结构的基本问题空间



例1：


HP LaserJet P2055dn [374B33] - 脱机
打印机(P) 文档(D) 查看(V)
文档名 状态 所有者 页数 大小 提交时间
http://news.sina.com.cn/china/ YHH 2 1.25 MB 18:00:14 2015/9/22
大话数据结构.pdf YHH 1 155 KB 18:01:20 2015/9/22
Microsoft PowerPoint - DS第... YHH 64 6.37 MB 17:59:27 2015/9/22
Microsoft PowerPoint - DS第... YHH 64 6.37 MB 17:59:10 2015/9/22
队列中有 4 个文档

例2：


函数入口
调用
函数返回
函数入口
调用
函数返回
函数入口
调用
函数返回
函数入口
函数返回

在计算机领域

程序设计

深度优先搜索/回溯法

函数调用、递归程序的执行过程

编译程序

变量的存储空间的分配

表达式的翻译与求值计算

栈

操作系统

作业调度、进程调度

后续章节

二叉树的层次遍历……

队

本章内容

3.1 栈的基本概念

3.2 栈的顺序存储结构

3.3 栈的链式存储结构

3.4 队的基本概念

3.5 队的顺序存储结构

3.6 队的链式存储结构

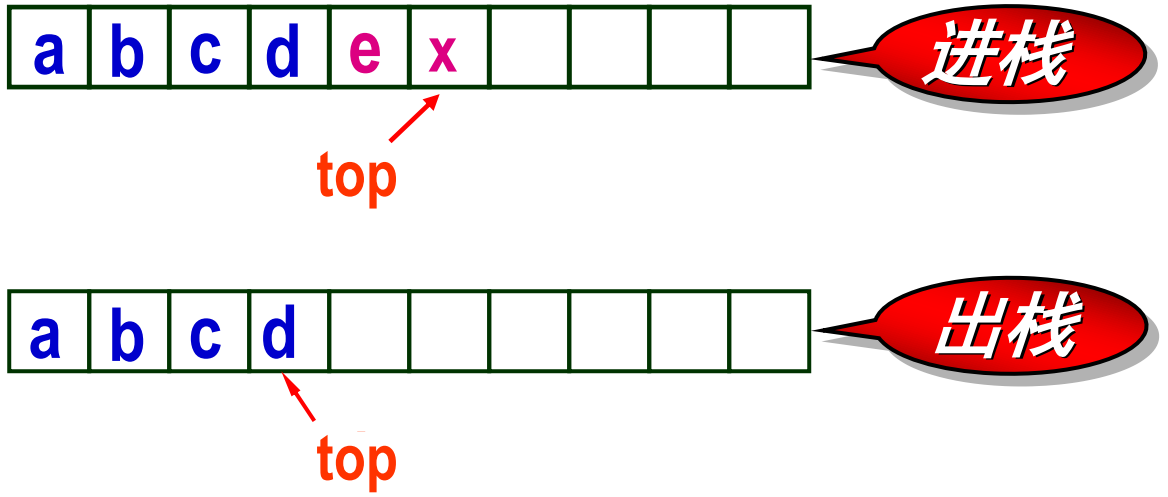
3.1 栈的基本概念

(一) 栈的定义

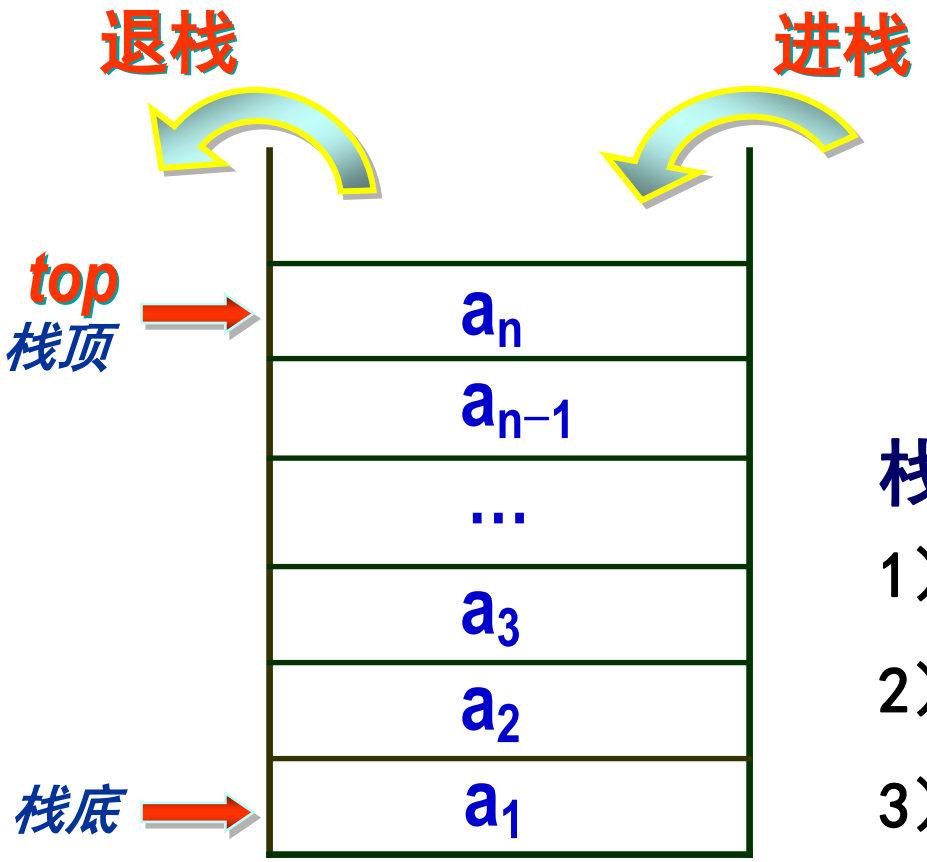
后进先出

栈(Stack)，也叫堆栈，是一种只允许在表的一端进行插入操作和删除操作的线性表。

允许操作的一端称为栈顶，栈顶元素的位置由一个称为**栈顶位置**的变量给出。当表中没有元素时，称之为**空栈**。



LIFO (Last-In-First-Out)



栈的示意图

栈的特点：

1）元素间呈线性关系

2）插入删除在一端进行

3）后进先出，先进后出

(二) 栈的基本操作

1. 插入（进栈、入栈、压栈） √
2. 删除（出栈、退栈、弹出） √
3. 测试栈是否为空 √
4. 测试栈是否已满
5. 检索当前栈顶元素

特殊性

1. 其操作仅仅是一般线性表的操作的一个子集。
2. 插入和删除操作的位置受到限制。

栈的基本操作

- void **push**(Stack s, ElemType e)　　//压一个元素进栈
- ElemType **pop**(Stack s)　　//从栈中弹出一个元素
- ElemType getTop(Stack s)　　//获取栈顶元素
- int isEmpty(Stack s)　　//判断栈是否为空
- int isFull(Stack s)　　//判断栈是否已满

练习1

若以符号PUSH和POP分别表示对堆栈进行1次进栈操作与1次出栈操作，则对进栈序列 a, b, c, d, e， 经过PUSH, PUSH, PUSH, POP, PUSH, POP, POP, PUSH以后, 栈中状态如何？得到的出栈序列是什么？

c , d , b

常见错误(多输出)：
c,d,b,e,a

注：有几个pop输出几个字符

练习2

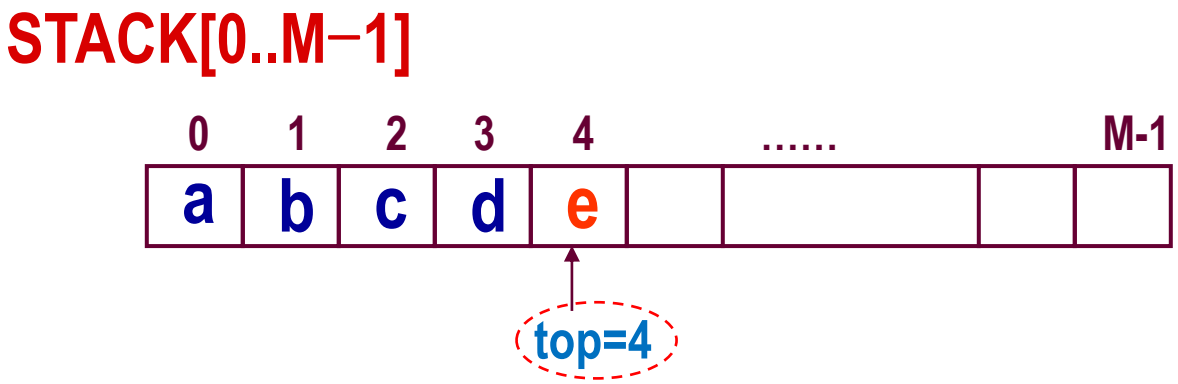
若符号PUSH(k)表示整数k进栈， POP表示栈顶元素出栈, 那么, 请画出依次执行PUSH(1), PUSH(2), POP, PUSH(5), PUSH(7), POP, PUSH(6)以后堆栈的状态。

2 , 7

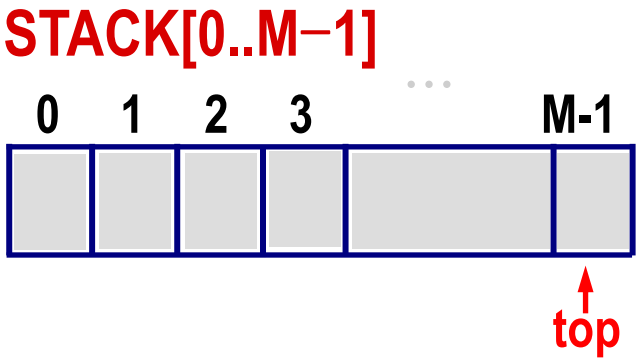
3.2 栈的顺序存储结构（顺序栈）

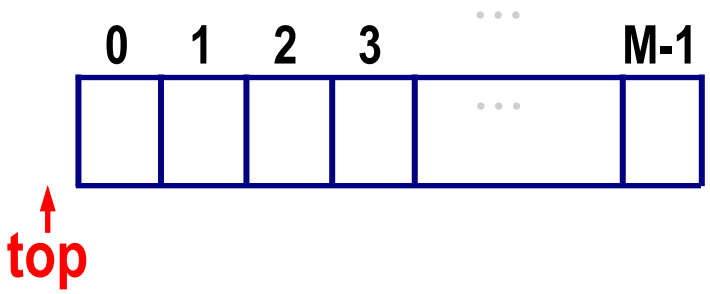
(一) 构造原理

描述栈的顺序存储结构最简单的方法是利用一维数组 STACK[0..M–1] 来表示，同时定义一个整型变量（不妨取名为top）给出栈顶元素的位置。



数组：静态结构
栈：动态结构





溢出

上溢 —— 当栈已满时做入栈操作。 (top=M–1)

下溢 —— 当栈为空时做出栈操作。 (top=–1)

类型定义

```
#define MAXSIZE    1000
ElemType  STACK[MAXSIZE];
int  Top;
```

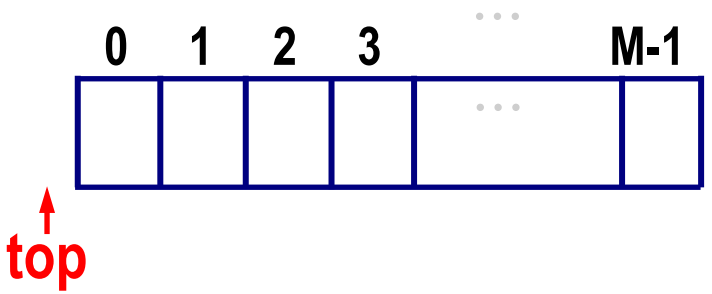
初始时,Top= –1

由于Top变量需要在多个函数间共享，为了保持函数接口简洁，在此定义为全局变量。

(二) 顺序栈的基本算法

1. 初始化堆栈

```
void  initStack( ){
        Top= –1;
}
```

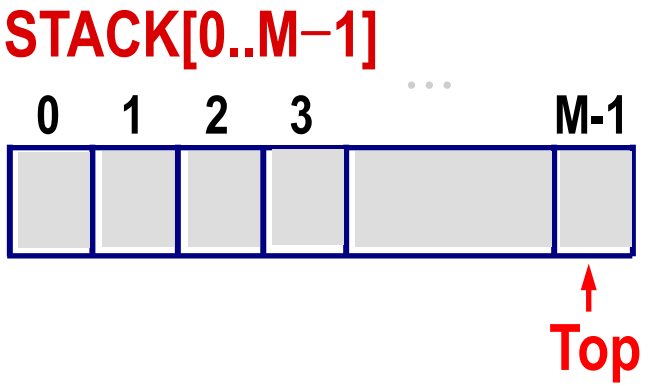


2. 测试堆栈是否为空

```
int  isEmpty( ){
       return Top== –1;
}
```

栈空,返回1,否则,返回0。

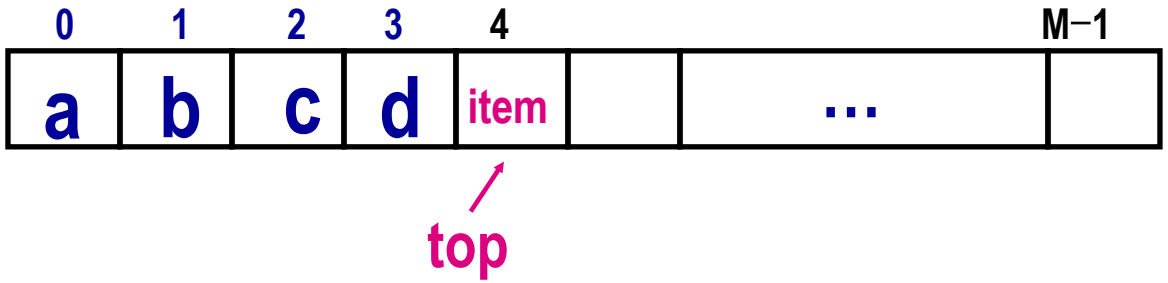
3. 测试堆栈是否已满




```
int isFull(  ){
      return Top==MAXSIZE–1;
}
```

栈满,返回1,否则,返回0。

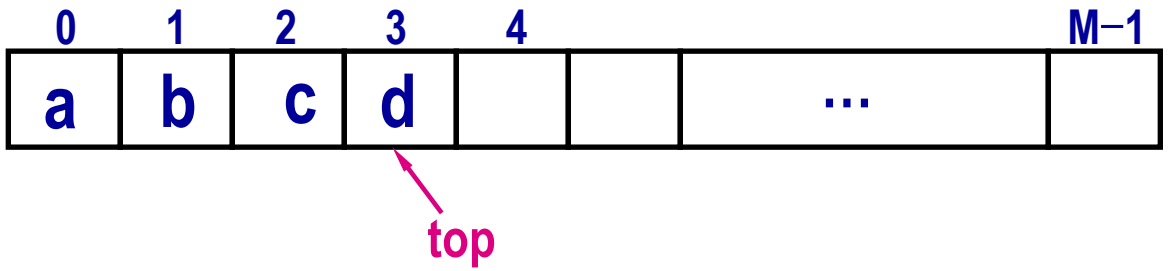
4. 进栈算法




```
void  push(ElmeType item ){
      if( isFull() )
            Error(“Full Stack!”);
      else
            STACK[++top]=item;
}
```

入栈成功

5. 出栈算法

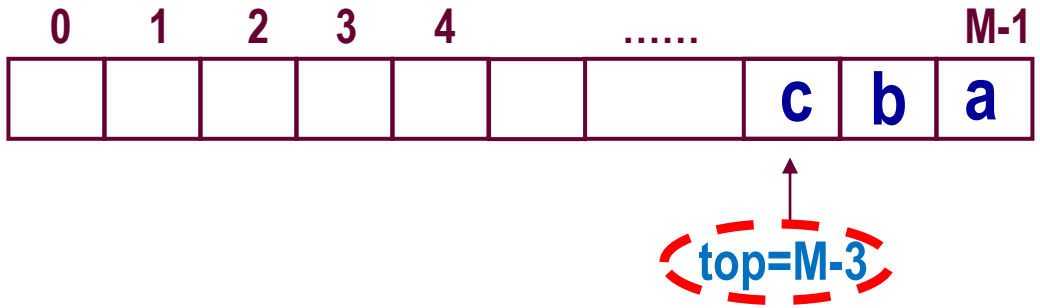



```
ElemType pop(){
    if(isEmpty())
        Error("Empty Stack!");
    else
        return STACK[top--];
}
```

出栈成功

高地址空间为栈底

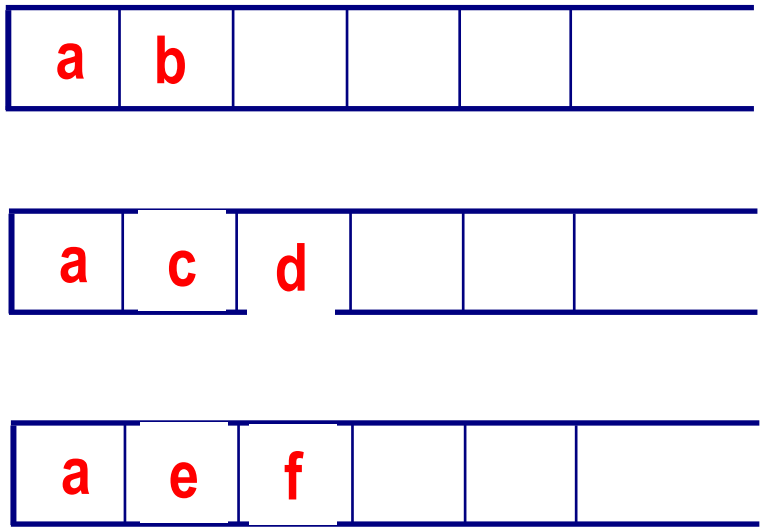
STACK[0..M−1]



练习4

设有一顺序栈S，元素a, b, c, d, e, f依次进栈，如果6个元素的合法出栈顺序是b, d, c, f, e, a，则栈的容量至少应该是（　　）

A、2　　B、3　　C、5　　D、6

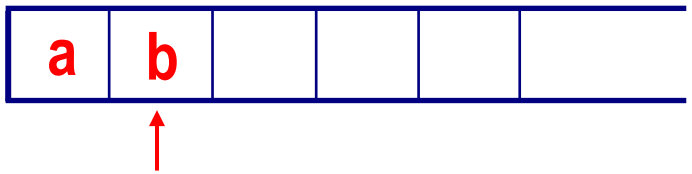


练习4扩展：设有一顺序栈S，n个不同的元素$a_1, a_2, a_3, \cdots, a_n$依次进栈，给出一个算法，判断上述元素的某个排列是否是合法的出栈序列，如果是，给出其出栈过程中所需的栈容量最小值。

例：

- 输入入栈序列abc，出栈序列cba，输出：3
- 输入入栈序列abc，出栈序列cab，输出：不合法
- 输入入栈序列abcdef，出栈序列bdcfea，输出：3

方法：用算法模拟出入栈过程



程序逻辑：

1. 元素依次入栈
2. 每入栈一个元素后，（重复）判断栈顶元素是否需要出栈

扩展知识：卡特兰数(Catalan Number) *

设有一顺序栈S，元素1,2,3,…,n 依次进栈，问这n个元素的合法的出栈序列个数总共有多少?

规律：假设k最后出栈，则在k入栈之前，比k小的值均出栈；
而之后比k大的值入栈，且都在k之前出栈
例：入栈序列abcdef，出栈序列bacfed

设 f(n)=“元素个数为n的出栈序列个数”。给定每个k：在k入栈之前，比k小的值均出栈，此处有f(k-1)种排列；而之后比k大的值入栈，且都在k之前出栈，计有f(n-k)种方式。以上共有f(k-1)* f(n-k)种。

k取值是相互独立的，故：

$$f(n) = f(0)f(n-1) + f(1)f(n-2) + \cdots\cdots + f(n-1)f(0)$$

$$f(0) = f(1) = 1$$

(三) 多栈共享连续空间问题

(以两个栈共享一个数组为例)

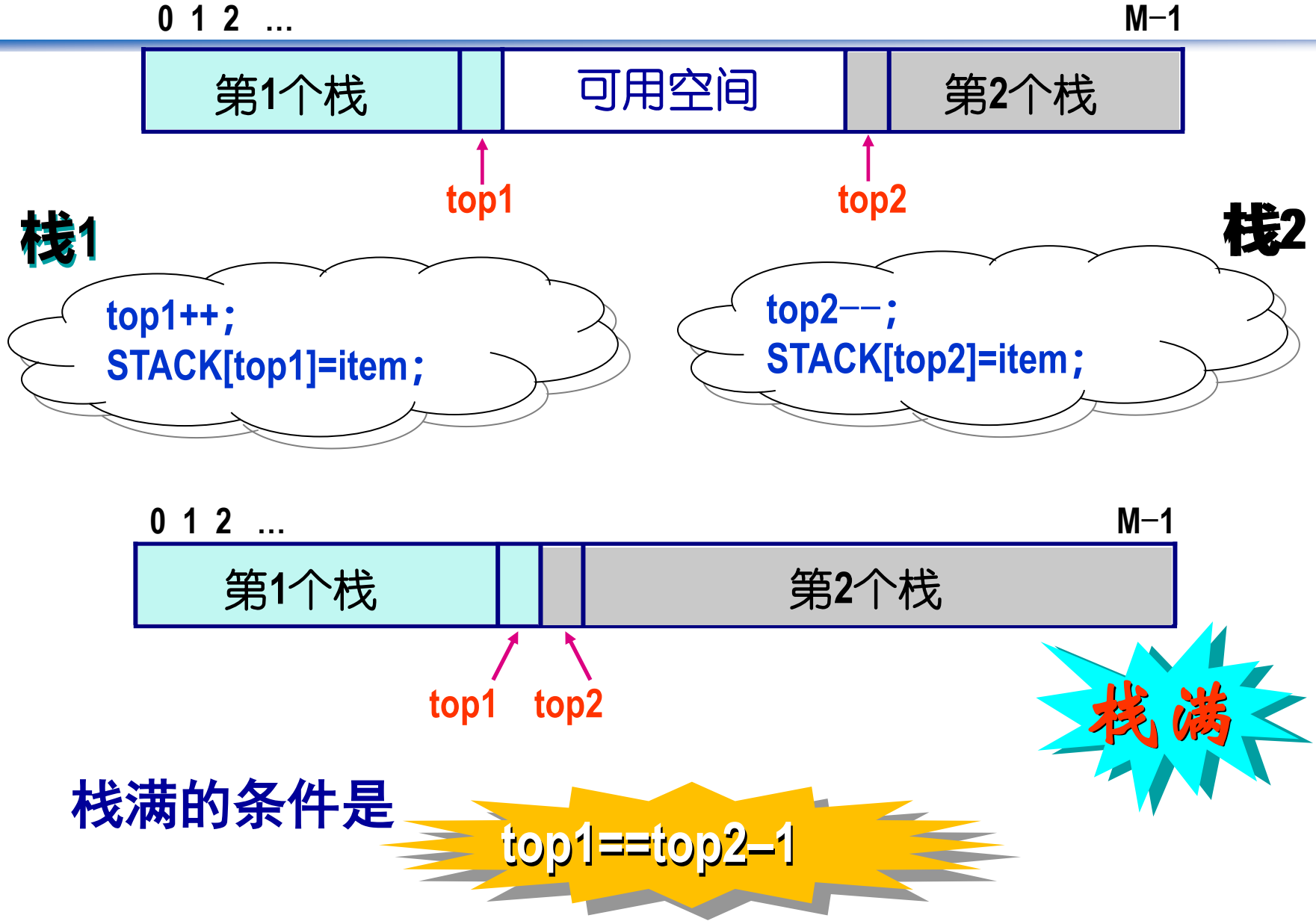
STACK[0..M−1]

Top1、Top2 分别给出第1个与第2个栈的栈顶元素的位置。
变量 i 指定使用哪个栈

void push(int i, ElemType item)

进栈

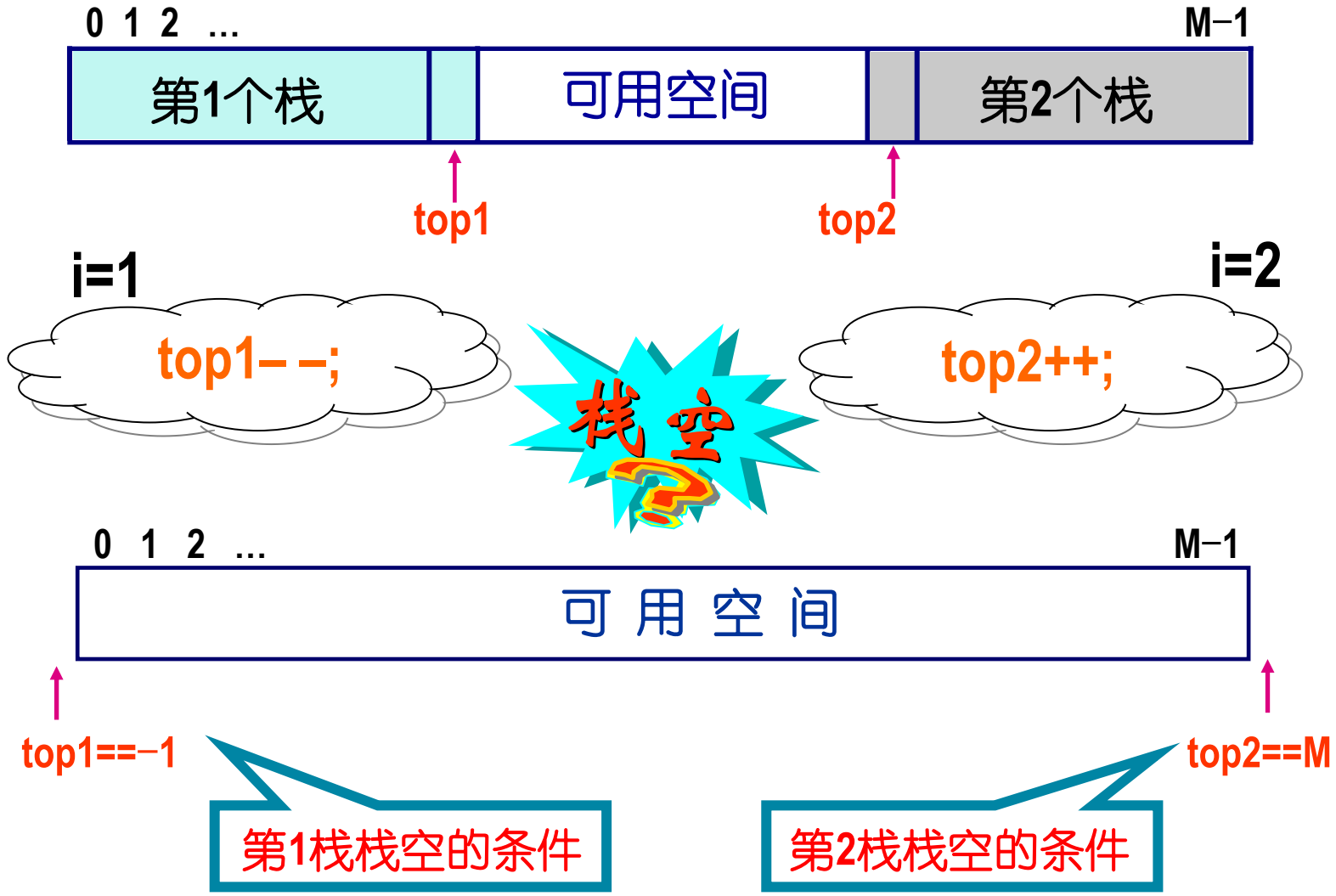
当i=1时，将item 插入第1个栈，
当i=2时，将item 插入第2个栈。


0 1 2 ...
M−1
第1个栈
可用空间
第2个栈
top1
top2
栈1
栈2
top1++;
STACK[top1]=item;
top2−−;
STACK[top2]=item;
0 1 2 ...
M−1
第1个栈
第2个栈
top1
top2
栈满
栈满的条件是
top1==top2−1

```
void  push(int i, ElemType item ){
    if(top1==top2–1)          /* 栈满 */
        Error(“Full Stack!”);
    else{
        if(i==1)                    /* 插入第1个栈 */
            STACK[++top1]=item;
        else                        /* 插入第2个栈 */
            STACK[– –top2]=item;
        return;
     }
}
```

出栈

当i=1时，删除第1个栈的栈顶元素，
当i=2时，删除第2个栈的栈顶元素。



```
EleType  pop(int i){
    if(i==1)
        if(top1==–1)
            Error(“Empty Stack1!”);
        else
            return STACK[top1– –];
    else
        if(top2==MAXSIZE)
            Error(“Empty Stack2!”);
        else
            return STACK[top2++];
}
```

对第一个栈进行操作

对第二个栈进行操作

3.3 栈的链式存储结构

(一) 构造原理

链接栈
链栈

链接栈就是用一个线性链表来实现栈结构，同时设置一个指针变量(这里不妨仍用top表示)指出当前栈顶元素所在链结点的位置。栈为空时，有top=NULL。

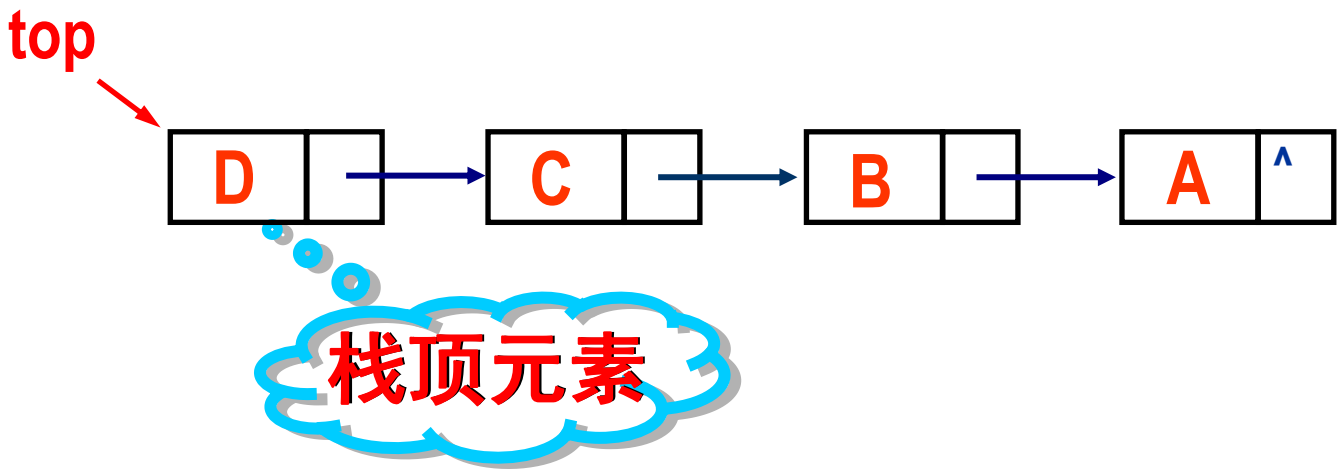
链栈是一种特殊的链表，其结点的插入（进栈）和删除（出栈）操作始终在链表的头。

例

在一个初始为空的链接栈中依次插入元素

A, B, C, D

以后，栈的状态为



类型定义

```
struct  node {
        SElmeType  data;
        struct  node  *link;
};
typedef  struct node  *Nodeptr;
typedef  struct node  Node;
Nodeptr Top; //即为链表的头结点指针
```

由于Top变量需要在多个函数间共享，为了简化操作在此定义为全局变量。

(二) 链栈的基本算法

1. 栈初始化

```
void initStack(  ){
        Top=NULL;
}
```

思考？

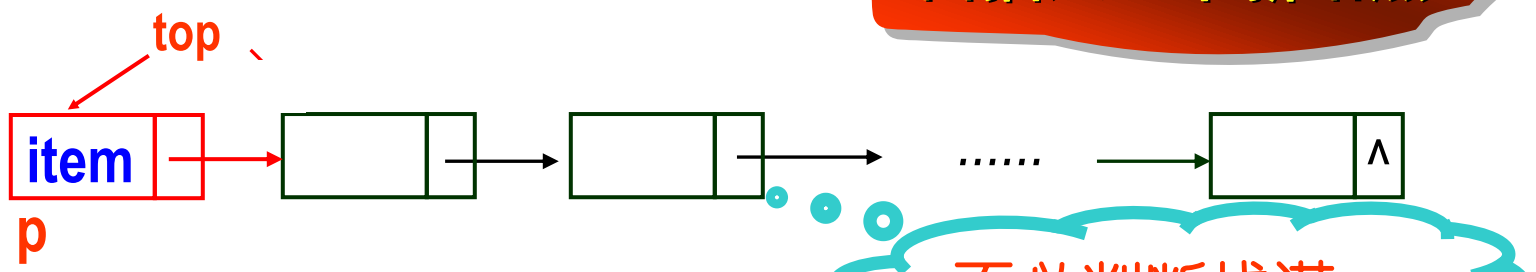
为什么不需要测试栈是否已满

2. 测试堆栈是否为空

```
int  isEmpty( ){
        return Top==NULL;
}
```

3. 进栈算法

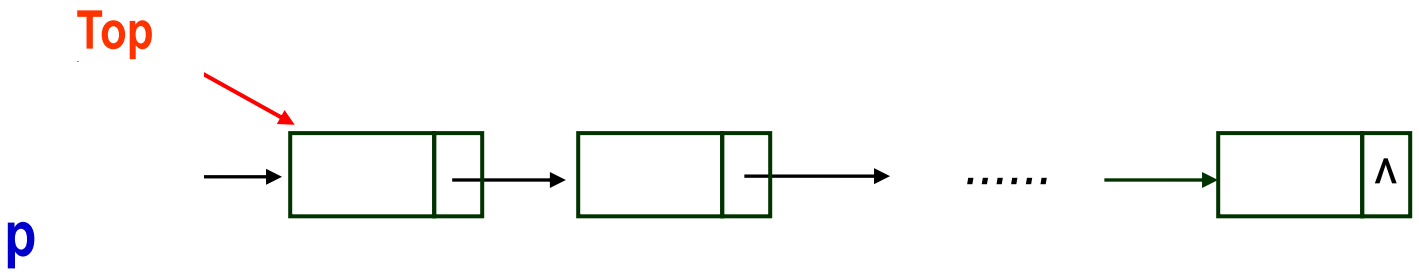
等效于在链表最前面插入一个新结点



不必判断栈满

```
void  push( ElemType item ){
      Nodeptr p;
      if( (p=(Nodeptr)malloc(sizeof(Node)))==NULL )
            Error("No memory!");
      else{
            p->data=item;          /*将item送新结点数据域*/
            p->link=Top;           /*将新结点插在链表最前面*/
            Top=p;                 /*修改栈顶指针的指向*/
      }
}
```

4. 出栈算法




```
ElemType  pop( ){
    Nodeptr p;
    ElemType item;
    if ( isEmpty() )
        Error("Empty Stack!");      /* 栈中无元素*/
    else{
        p=Top;                      /* 暂时保存栈顶结点的地址*/
        item=Top->data;             /* 保存被删栈顶的数据信息*/
        Top=Top->link;              /* 删除栈顶结点 */
        free(p);                    /* 释放被删除结点*/
         return item;               /* 返回出栈元素*/
    }
}
```

栈的应用：(1) 括号匹配问题

给定一个只包括 '('，')'，'{'，'}'，'['，']' 的字符串 s ， 判断字符串是否有效。

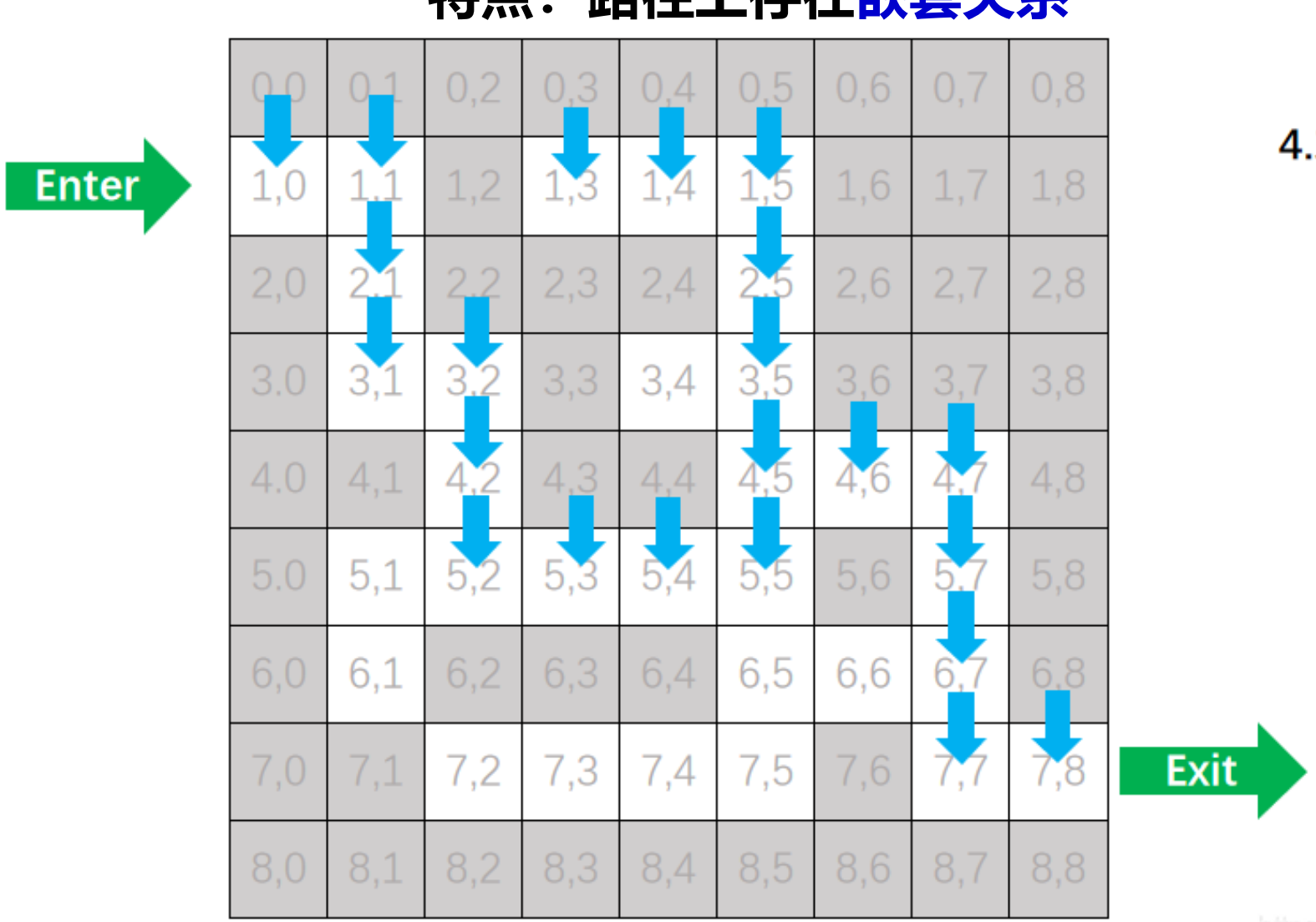
- **利用栈做匹配，依次读取一个字符，做如下判断：**
  - 遇到左括号 '(', '{', '[' 则压入栈
  - 遇到右括号 ')' , '}' , ']' 和栈顶元素匹配， 匹配到相应的左括号，则栈顶元素出栈, 否则匹配失败
  - 最后检查栈内没有其他元素则匹配成功.

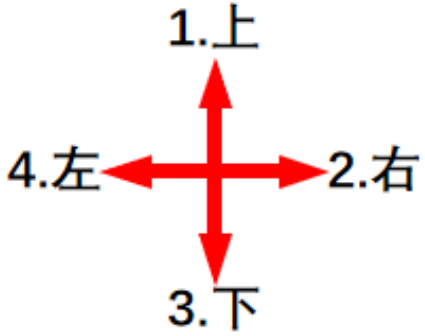
- 特点：处理嵌套关系

  a{b{cd},{[(e)],{(f)}}g}

栈的应用：(2) 迷宫问题/深度优先搜索

特点：路径上存在嵌套关系





回溯算法之迷宫问题（Maze）- CSDN博客

栈的应用：(2)深度优先搜索/迷宫问题

- **用二维数组表示迷宫maze[][]**
  - 0-通路；1障碍；记录入口和出口坐标
- **搜索方向数字化：dir[4][2]={{-1,0},{1,0},{0,-1},{0,1}};**

```
private static int[][] maze = {
        {1, 1, 1, 1, 1, 1, 1, 1, 1},
入口    {0, 0, 1, 0, 0, 0, 1, 1, 1},
        {1, 0, 1, 1, 1, 0, 1, 1, 1},
        {1, 0, 0, 1, 0, 0, 1, 1, 1},
        {1, 1, 0, 1, 1, 0, 0, 0, 1},
        {1, 0, 0, 0, 0, 0, 1, 0, 1},
        {1, 0, 1, 1, 1, 0, 0, 0, 1},
        {1, 1, 0, 0, 0, 0, 1, 0, 0},   出口
        {1, 1, 1, 1, 1, 1, 1, 1, 1}
};
```


回溯算法之迷宫问题（Maze）- CSDN博客

栈的应用：(2)深度优先搜索/迷宫问题

- **标记**走过的位置，避免重复搜索
- **用栈记录路径**：前进——进栈；回退——出栈

```
private static int[][] maze = {
        {1, 1, 1, 1, 1, 1, 1, 1, 1},
        {0, 0, 1, 0, 0, 0, 1, 1, 1},
        {1, 0, 1, 1, 1, 0, 1, 1, 1},
        {1, 0, 0, 1, 0, 0, 1, 1, 1},
        {1, 1, 0, 1, 1, 0, 0, 0, 1},
        {1, 0, 0, 0, 0, 0, 1, 0, 1},
        {1, 0, 1, 1, 1, 0, 0, 0, 1},
        {1, 1, 0, 0, 0, 0, 1, 0, 0},
        {1, 1, 1, 1, 1, 1, 1, 1, 1}
};
```

入口

出口

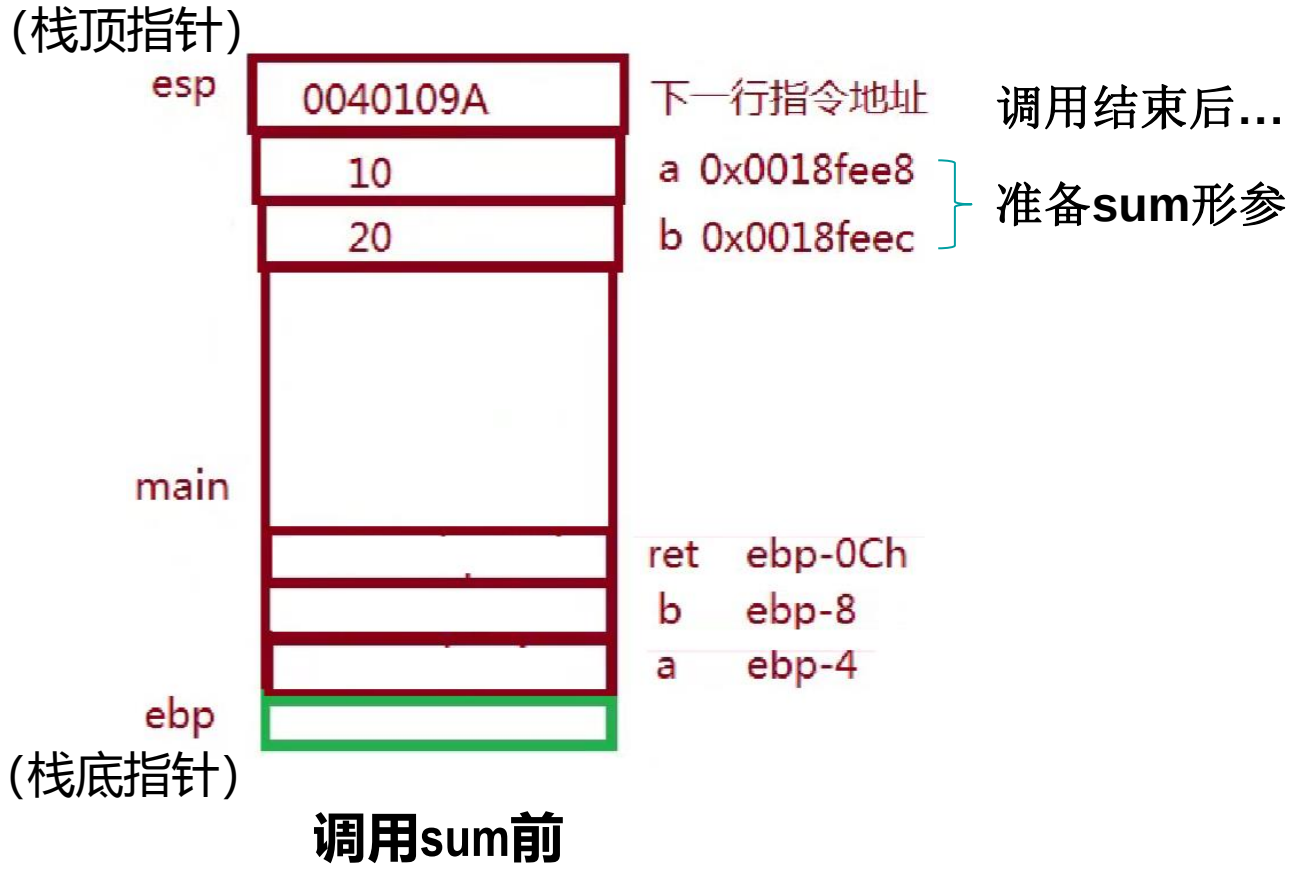
回溯算法之迷宫问题（Maze）- CSDN博客

栈的应用：(3)过程调用*

```
int sum(int a, int b)
{
    int temp = 0;
    temp = a+b;
    return temp;
}
int main()
{
    int a = 10;
    int b = 20;
    int ret = 0;

    ret = sum(a, b);
    printf("%d\n", ret);

    return 0;
}
```



调用sum前

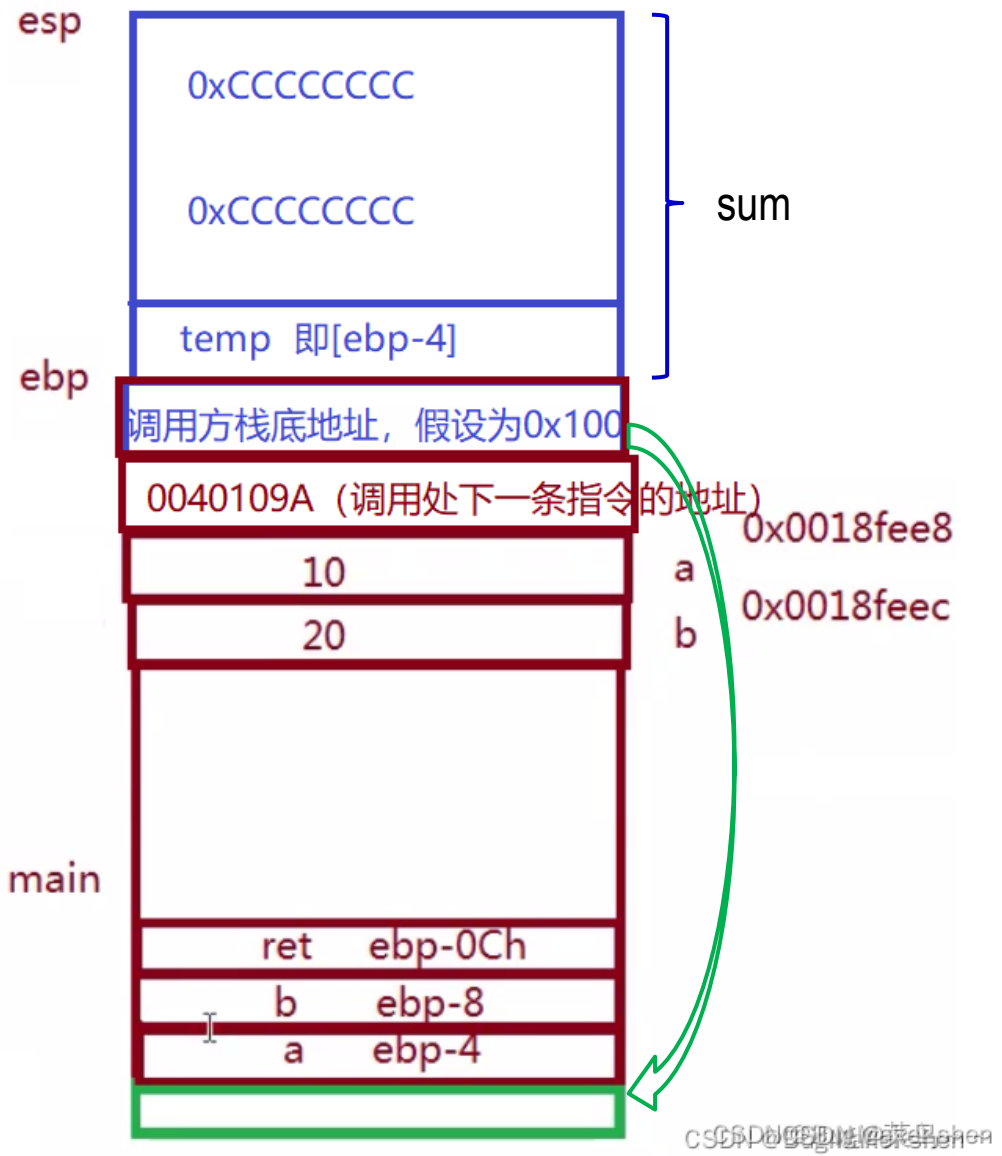
网络资源（https://blog.csdn.net/qq_41721746/article/details/124557537）

栈的应用：(3)过程调用*

```
int sum(int a, int b)
{
    int temp = 0;
    temp = a+b;
    return temp;
}
int main()
{
    int a = 10;
    int b = 20;
    int ret = 0;

    ret = sum(a, b);
    printf("%d\n", ret);

    return 0;
}
```

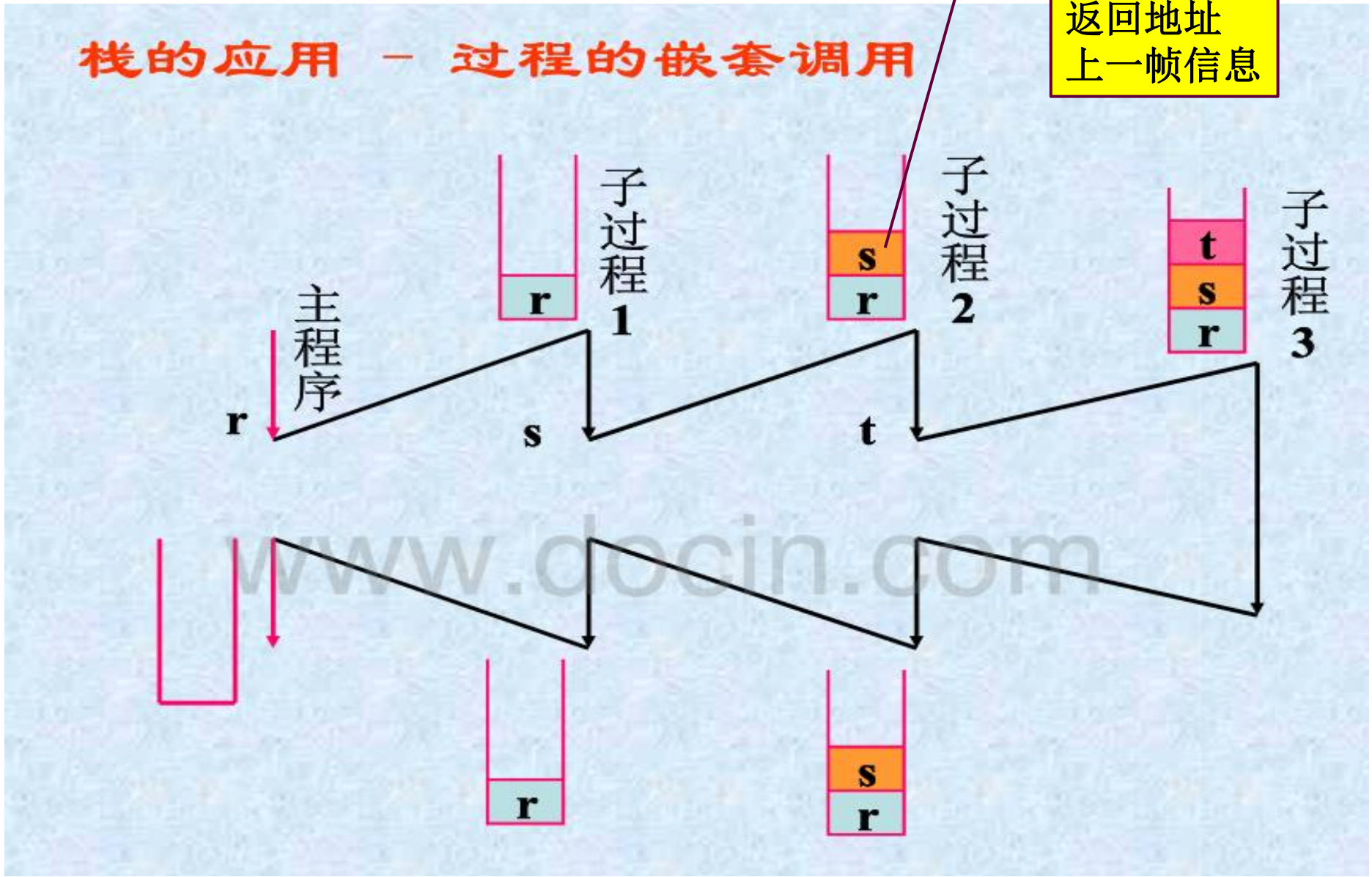


调用进入sum时

网络资源（https://blog.csdn.net/qq_41721746/article/details/124557537）

栈的应用：(3)过程调用

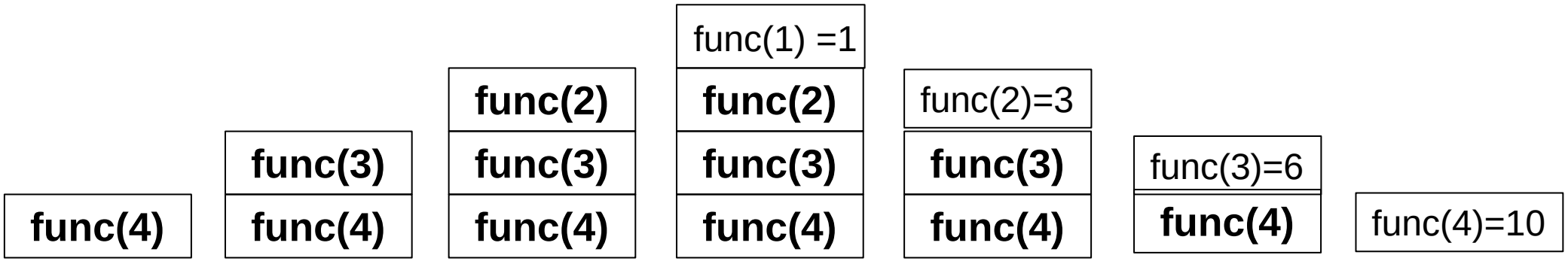
参数列表
局部变量
返回地址
上一帧信息



栈的应用：(3)过程调用

设有递归函数 f(n) = n + f(n-1), f(1)=1，求f(4).

```
int func(int n){
    if (n==1)
        return 1;
    else
        return n + func(n-1);
}
```



函数调用关系分析

第4套作业编程题2

【问题描述】

给定某能正常运行结束的用户函数调用栈信息（当一个函数被调用时将入栈，当调用返回时，将出栈）。编写程序，对函数调用栈信息进行分析，依据函数入栈和出栈信息，分析函数调用关系（当一个函数入栈时，它就是当前栈顶函数调用的一个函数），即一个函数调用了哪些不同函数。并按运行时调用序输出调用关系。

【调用关系】

main:input,mysqrt,findA,findB,findC,ouput
mysqrt:max
findB:area
area:mysin,mycos,mysqrt
findC:area,mysqrt

关键数据结构：

1. 调用栈：跟踪操作过程
2. 一维结构数组 + 链表：数组记录函数名，链表记录被调用函数

【操作序列】

```
push main
push input
pop
push mysqrt
pop
push findA
pop
push findB
push area
push mysin
pop
push mycos
pop
push mysqrt
pop
pop
pop
push findC
push area
push mysin
pop
pop
push mysqrt
push max
pop
pop
pop
push output
pop
pop
```

问题3.1a：表达式计算（中缀表达式计算）

一般形式的表达式通常称为中缀表达式(infix)：

$$a+b*c+(d*e+f)/g$$

【问题描述】从标准输入中读入一个整数算术运算表达式，如24/(1+2+36/6/2-2)*(12/2/2)=，计算表达式并输出结果。

1、表达式运算符只有+、-、*、/，表达式末尾的‘=’字符表示表达式输入结束，表达式中可能会出现空格；

2、表达式中会出现圆括号，括号可能嵌套，不会出现错误的表达式；

3、出现除号/时，以整数相除进行运算，结果仍为整数，例如：5/3结果为1。

问题3.1a：表达式计算（中缀表达式计算）

24/(1+2+36/6/2−2)*(12/2/2) =

计算机一般从左到右扫描并处理中缀表达式

计算中缀表达式的主要挑战依然是处理**嵌套关系**：

- 运算符有优先级，*,/优先于+,-；左边的+,-优先于右边

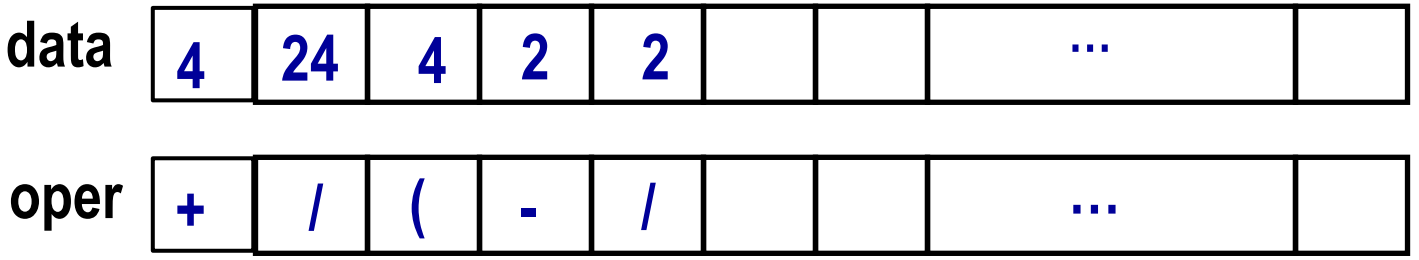
  例：1+2+3-4=; 1+2*3/4=;

- 括号会改变计算的次序（或者说括号也有优先级）

设两个栈：**数据栈**与**运算符栈**。

而后从左到右扫描表达式，遇数值或运算符而暂时不能计算时，则分别入栈；待定元素符合特定条件时则弹出并计算。

4+ 24/ (1+2+36/6/2−2) *(12/2/2) =



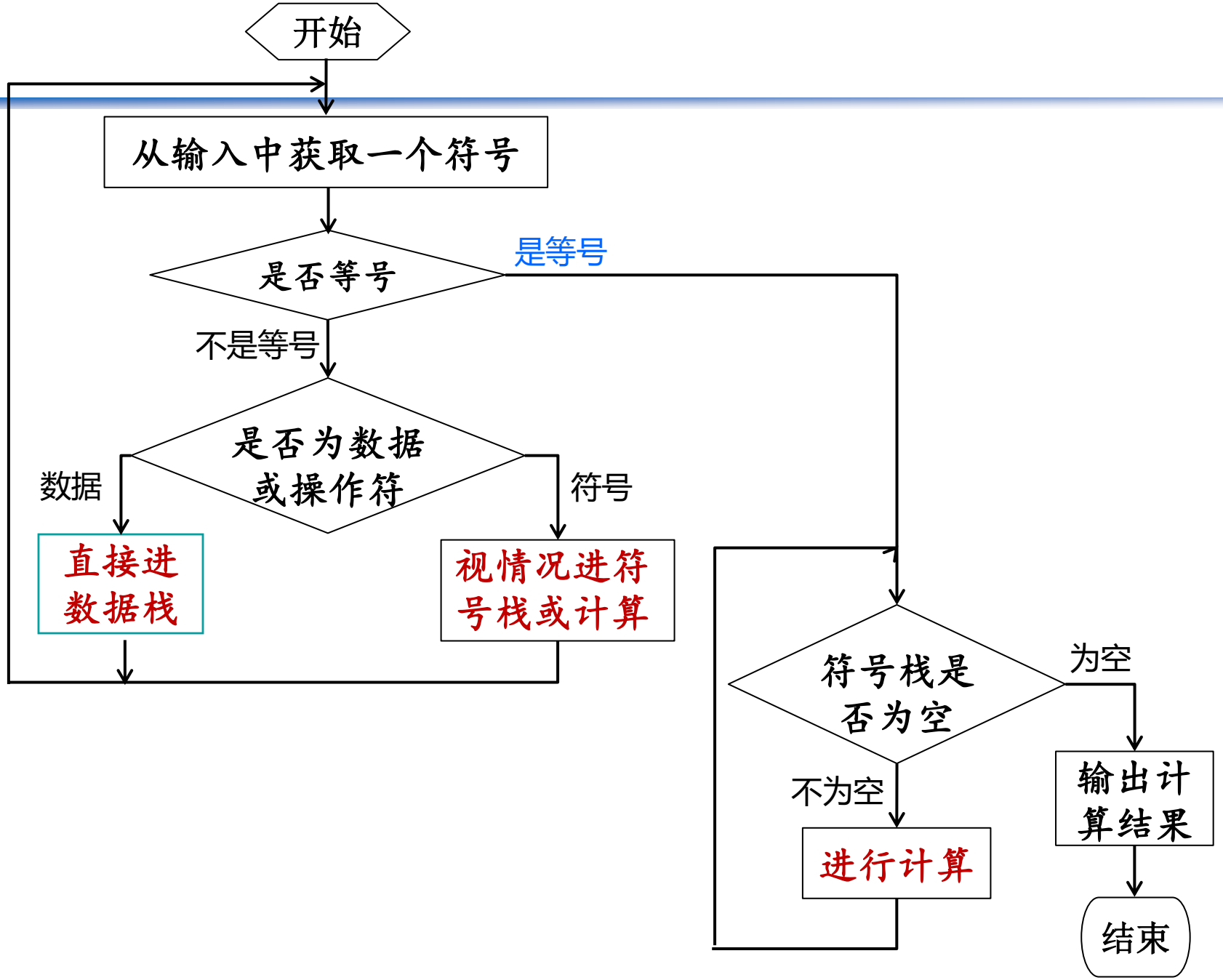
从左至右遍历表达式中的每个数字和符号，（根据表达式计算的语义）提炼出如下计算规则：

① 若是数字直接进数据栈；

② 若是符号则“判断”入栈或出栈操作：

- 若是+, −, *, /等运算符，则比较该运算符与栈顶运算符， 若当前运算符优先级高于栈顶运算符，则直接入栈；
  否则，弹出运算符并计算，结果入数据栈。继续比较。
- 若是(，则直接将其压栈，该操作符需特殊对待；
- 若是)，则将符号栈中元素弹出并计算，直到遇到“(”。 “(”弹出但无需计算，“)”不入栈；

③ 最后，将符号栈中元素依次弹出并计算，直到栈为空。


开始
从输入中获取一个符号
是否等号
是等号
不是等号
是否为数据或操作符
数据
符号
直接进数据栈
视情况进符号栈或计算
符号栈是否为空
为空
不为空
进行计算
输出计算结果
结束

问题3

参考实现

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<ctype.h>
#define  MAXSIZE 100
enum  SymbolType {NUM=0, OP=1, EQ=2, OTHER=3 };     //符号类型
enum  Oper {EPT=0, ADD, MIN, MUL, DIV, LEFT, RIGHT }; //运算符枚举名
char opCh[] = {' ', '+', '-', '*', '/', '(', ')'};   //运算符字符
int opPri[] = { -1,   0,  0,  1,  1,  2,  2 };   //运算符优先级
struct Symbol {        //从输入设备中读到的一个运算符、数据项或等号
    enum SymbolType symType;          //符号类型
    enum Oper opn;                    //若是运算符，保存其枚举名
    char str[10];                     //以字符串形式保存数值
};

struct Symbol getSym();               //读取符号
void compute(enum Oper op);           //执行算术运算

void pushNum(int num);
int popNum();

void pushOp(enum Oper op);
enum Oper  popOp();
enum Oper  topOp();
```

枚举类型（enum）

为了使程序具有更好的扩展性，
本问题实现中用到了**枚举**类型

- **定义形式：**

  enum 枚举名 { 值表 };

  如：enum Boolean { FALSE, TRUE };

  如：enum color { red, green, yellow, white, black };

- **C语言编译程序把值表中的标识符视为从0开始的连续整数。**

  如：

  enum color { red, green, yellow = 5, white, black };

  则：

  red=0, green=1, yellow=5, white=6, black=7

- **枚举类型用途：**
  - **枚举类型通常用来说明变量取值为有限的一组值之一，**
  - **用来定义常量，如：enum { PI = 3.14159 };**

枚举类型（续）

- **枚举变量定义**

```
enum color chair;
enum color suite[10];
```

- **在表达式中使用枚举变量**

```
chair = red;
suite[5] = yellow;
if( chair == green ) ...
```

注意：对枚举变量的赋值并不是将标识符字符串传给它，而是把名值表该标识符所对应的常数值赋与变量。

问题3.1：代码实现

非必须，可增加可读性和安全性

```
#define  MAXSIZE 100
enum  SymbolType {NUM=0, OP=1, EQ=2, OTHER=3 }; //符号类型

enum  Oper {EPT=0, ADD, MIN, MUL, DIV, LEFT, RIGHT }; //运算符枚举名
char opCh[] = {    ' ',   '+',    '-',    '*',    '/',     '(',      ')'};    //运算符字符
int opPri[] = {      -1,    0,      0,      1,     1,       2,        2 };    //运算符优先级

struct Symbol {        //从输入设备中读到的一个运算符、数据项或等号
    enum SymbolType symType;          //符号类型
    enum Oper opn;                    //若是运算符，保存其枚举名
    char str[10];                     //以字符串形式保存数值
};

struct Symbol getSym();               //读取下一个运算符/操作数
void compute(enum Oper op);           //执行算术运算

void pushNum(int num);
int popNum();

void pushOp(enum Oper op);
enum Oper  popOp();
enum Oper  topOp();
```

问题

```
struct Symbol getSym() {
    int  c, n;
    struct Symbol sym;
    while ((c = getchar()) != '=') {
        if (c >= '0' && c <= '9') {
                for (n = 0; c >= '0' && c <= '9'; c = getchar())
                    sym.str[n++] = c;
                ungetc(c, stdin);  sym.str[n] = '\0';  sym.symType = NUM;
                return sym;
        }
        sym.str[0] = c;       sym.str[1] = '\0';
        switch (c) {
            case ' ': case '\t': case '\n': break;
            case '+': sym.symType = OP; sym.opn = ADD;          return sym;
            case '-': sym.symType = OP; sym.opn = MIN;          return sym;
            case '*': sym.symType = OP; sym.opn = MUL;          return sym;
            case '/': sym.symType = OP; sym.opn = DIV;          return sym;
            case '(': sym.symType = OP; sym.opn = LEFT;         return sym;
            case ')': sym.symType = OP; sym.opn = RIGHT;        return sym;
            default: sym.symType = OTHER;                       return sym;
        }
    }
    sym.symType = EQ;     return sym;
}
```

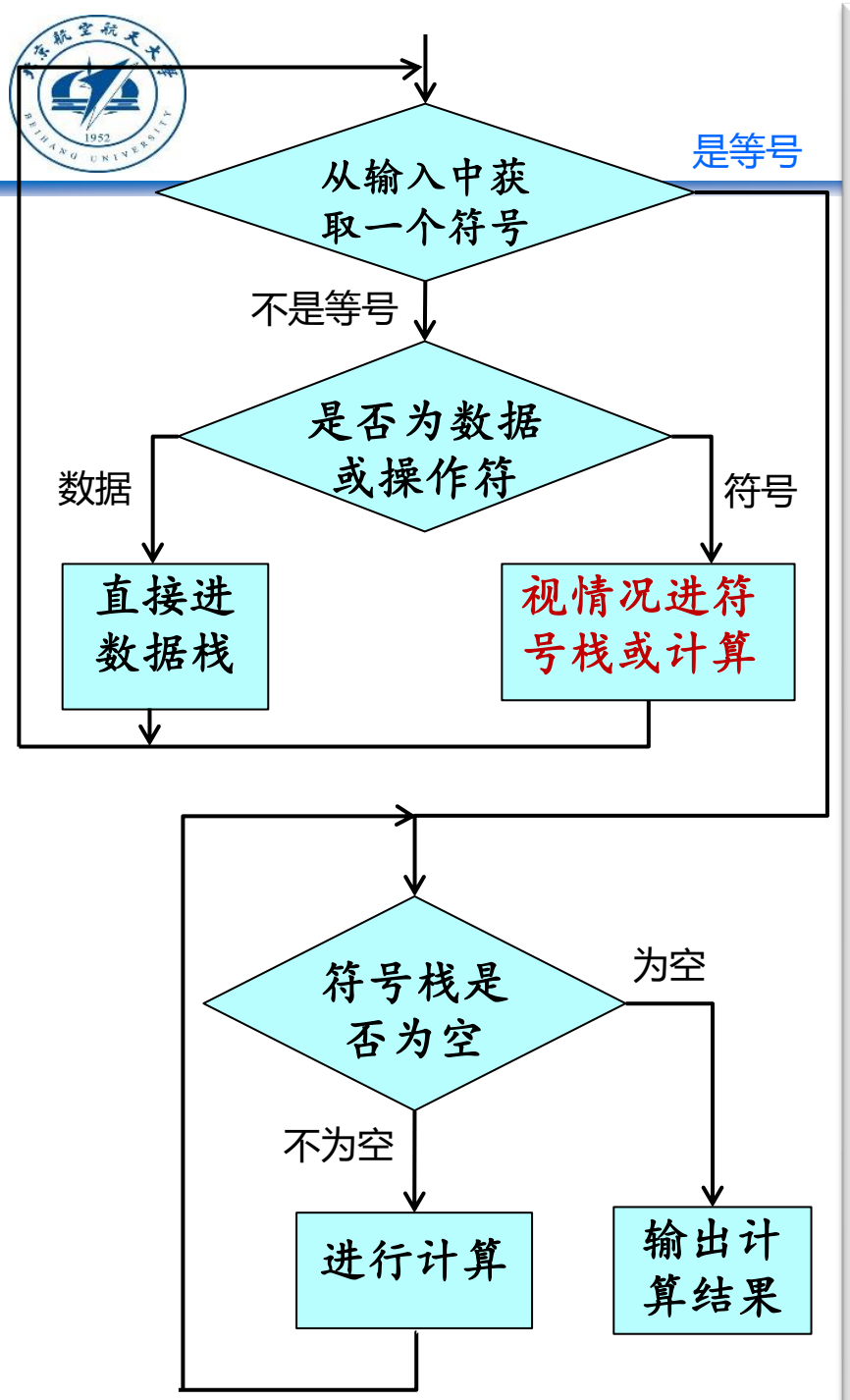
```
void compute(enum Oper op) {
        int  tmp;
        switch (op) {
        case ADD:
                pushNum(popNum() + popNum());
                break;
        case MIN:
                tmp = popNum();
                pushNum(popNum() - tmp);
                break;
        case  MUL:
                pushNum(popNum() * popNum());
                break;
        case DIV:
                tmp = popNum();
                pushNum(popNum() / tmp);
                break;
        }
}
```

```
//数据栈及操作
int Num_stack[MAXSIZE]; //数据栈
int Ntop = -1; //数据栈顶，栈初始为空
void pushNum(int num) {
    if (Ntop == MAXSIZE - 1) {
        printf("Data stack is full!\n");
        exit(1);
    }
    Num_stack[++Ntop] = num;
}


int popNum() {
    if (Ntop == -1) {
        printf("Error in the expression!\n");
        exit(1);
    }
    return Num_stack[Ntop--];
}
```

```
//运算符栈及操作
enum Oper Op_stack[MAXSIZE];//符号栈
int  Otop = -1; //运算符栈顶指示器
void pushOp(enum Oper op) {
    if (Ntop == MAXSIZE - 1) {
        printf("operator stack is full!\n");
        exit(1);
    }
    Op_stack[++Otop] = op;
}

enum Oper popOp() {
    if (Otop != -1) {
        return Op_stack[Otop--];
    }
    return EPT;
}
enum Oper topOp() {
    return Op_stack[Otop];
}
```




```
int executeExpr() {
    struct Symbol sym = getSym();
    while (sym.symType != EQ) {
        if (sym.symType == NUM) {                                    //case:数字
            pushNum(atoi(sym.str));
        }else if (sym.symType == OP && sym.opn != RIGHT) {  //case:非右括号
            while (opPri[sym.opn] <= opPri[topOp()] && topOp() != LEFT)
                compute(popOp());        //执行一次算术运算，计算结果入数据栈
            pushOp(sym.opn);
        }else if (sym.symType == OP && sym.opn == RIGHT) { //case:右括号
            enum Oper t;
            while ((t = popOp()) != LEFT)
                compute(t);
        }else {                                                      //case:其他异常情况
            printf("Error in the expression!\n");
            return 1;
        }
        sym = getSym();          //读取下一个运算符/操作数
    }
    while (Otop >= 0)            //将栈中所有运算符弹出计算或输出
        compute(popOp());
    printf("%d\n", popNum());
    return 0;
}
```

问题3.1b：表达式计算
（中缀表达式先转后缀表达式，再计算后缀表达式）

一般形式的表达式通常称为中缀表达式(infix)：

$a+b*c+(d*e+f)/g$

在（计算机）计算中缀表达式时面临的主要问题有：

- 运算符有优先级
- 括号会改变计算的次序

在编译阶段简化问题？

填空5，7

为了方便表达式的（计算机）计算，波兰数学家Lukasiewicz在20世纪50年代发明了一种将运算符写在操作数之后的表达式表示方式，称为后缀表达式(postfix)，或逆波兰表示(Reverse Polish Notation,RPN)

中缀表达式	后缀表达式（RPN）
a + b	a b +
a+ b * c	a b c * +
a + b * c + d	a b c * + d +

编译器通常先将数学表达式转换成后缀表达式，然后再将后缀表达式转换成机器代码。

为了方便表达式的（计算机）计算，波兰数学家Lukasiewicz在20世纪50年代发明了一种将运算符写在操作数之后的表达式表示方式，称为后缀表达式(postfix)，或逆波兰表示(Reverse Polish Notation,RPN)

中缀表达式	后缀表达式（RPN）
a+b	a b +
a+b*c	a b c * +
a+b*c+d	a b c * + d +
a+(b*c+d)	a b c * d + +
a+b*c+(d*e+f)/g	a b c * + d e * f + g / +

后缀表达式，无括号，无算符的优先级问题，便于机器处理

（1）后缀表达式计算（利用数据栈）

后缀表达式在程序执行时计算，此时无需考虑运算符优先级

中缀表达式	后缀表达式（RPN）
a+b*c+(d*e+f)/g	a b c * + d e * f + g / +

计算规则：从左至右遍历后缀表达式中每个数字和符号：

- 若是数字直接进栈；
- 若是运算符（+，-，*，/），则从栈中弹出两个元素进行计算（注意：后弹出的是左运算数），并将计算结果进栈。
- 遍历结束，将计算结果从栈中弹出（栈中应只有一个元素，否则表达式有错）。

（2）中缀到后缀的转换规则（利用符号栈）(类似于中缀表达式的计算)

规则：从左至右遍历中缀表达式中的每个数字和符号：

◆若是数字直接输出，即成为后缀表达式的一部分；

◆若是符号：

➢若是+，*等运算符，则比较该运算符与栈顶运算符，从栈中弹出并输出优先级高于当前的符号，直到遇到一个优先级低的符号；然后将当前符号压入栈中。

➢若是(，则直接将其压栈，该操作符需特殊对待；

➢若是)，则将栈中元素弹出并输出，直到遇到“(”，“(”弹出但不输出；

◆遍历结束，将栈中所有元素依次弹出并输出，直到栈为空。

中缀表达式	后缀表达式（RPN）
a+b*c+(d*e+f)/g	abc*+de*f+g/+

```
int toSuffixExpr() {
    struct Symbol sym = getSym();
    while (sym.symType != EQ) {
        if (sym.symType == NUM) {                                   //case:数字
            printf("%s ", sym.str);
        }else if (sym.symType == OP && sym.opn != RIGHT) {          //case:非右括号
            while (opPri[sym.opn] <= opPri[topOp()] && topOp() != LEFT)
                printf("%c ", opCh[popOp()]);
            pushOp(sym.opn);
        }else if (sym.symType == OP && sym.opn == RIGHT) {          //case:右括号
            enum Oper t;
            while ((t = popOp()) != LEFT)
                printf("%c ", opCh[t]);
        }else {                                                     //case:其他异常情况
            printf("Error in the expression!\n");
            return 1;
        }
        sym = getSym();             //读取下一个符号/数值
    }
    while (Otop >= 0)               //将栈中所有运算符弹出计算或输出
        printf("%c ", opCh[popOp()]);
    return 0;
}
```

问题3.1 计算器/表达式处理（小结）

从本例中看出由于使用了**栈**这种数据结构，一方面简化了算法**复杂性**；另一方面程序具有很好的**可扩展性**（如增加新的优先级运算符非常方便）。

思考：修改该表达式计算程序，为其增加：**%(求余)，>(大于)，<(小于)**等运算符。运算符优先级照C语言中定义。

2021-2022学年期末考题： min解释语言A

【问题描述】

有一min解释语言，其只有整型常量、变量、赋值语句、算术表达式语句及打印语句组成。编写一程序，实现该系统。规则：

1. 变量仅由单个小写字母组成。

2. 只有三种语句：print、exit和赋值语句。一行只有一个语句，语句中的字符个数不会超过200，每条语句的末尾都有换行符。语句格式为：

   a) 赋值语句：<变量> = <算术表达式>。<算术表达式>是由十进制整型常量（末尾不带后缀）、变量、算术运算符（+, -, *, /）及小括号组成；赋值语句中没有空白符。

   b) print语句：print <变量列表>。<变量列表>为由空格分隔的变量序列，print和变量间由空格分隔。功能是输出变量列表中各变量的值。

   c) exit语句：exit。退出解释系统。

3. 在进行赋值运算、算术运算、打印输出时，相应变量都会有具体值，不会出现语法错误。在计算过程中结果数据类型为浮点型，输出时保留小数点后2位。

2021-2022学年期末考题：min解释语言A

【输入形式】

从标准输入读入解释系统的待执行语句，每条语句独占一行；最后一条语句为exit。

注意：所有输入中出现的常量都是不带后缀的十进制整数常量。

【输出形式】

在解释系统执行过程中，每条print语句执行后，print后各变量的值将会输出到标准输出，输出时保留小数点后2位，各数据间以一个空格分隔，最后一个数据后没有空格，有换行符。

【样例输入和输出】

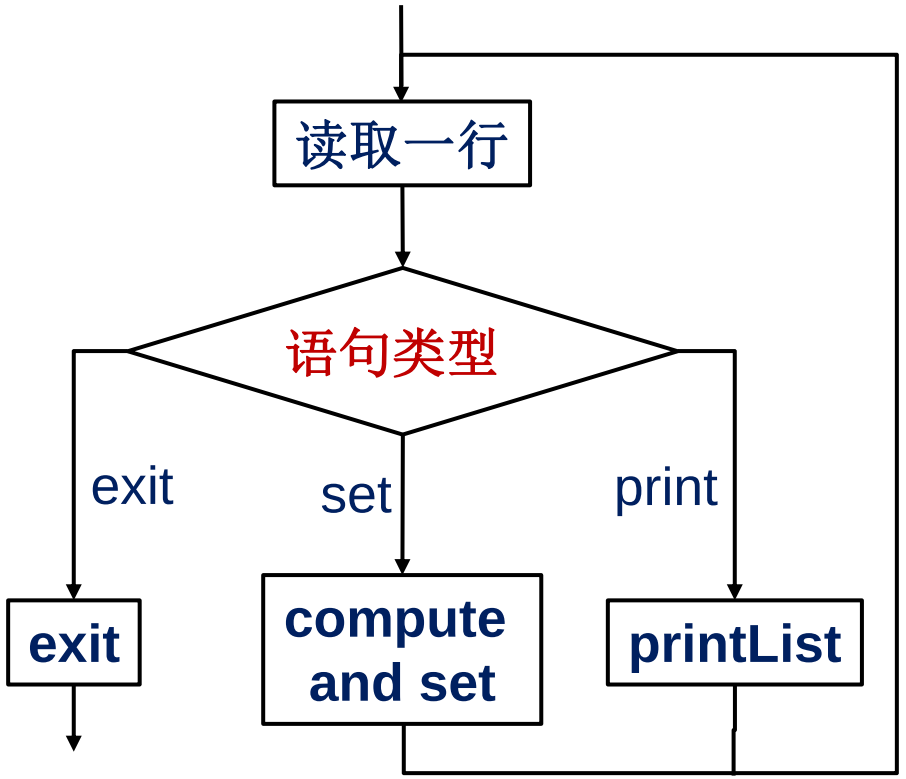
```
a=10
b=20
c=(a+b)/4
print a b c
10.00 20.00 7.50
d=a*(b-c)
print d
125.00
exit
```

【样例说明】

第一条语句将10赋值给变量a；
第二条语句将20赋值给变量b；
第三条语句需要先计算表达式(a+b)/4的值，然后将结果赋值给变量c，这时c的值为7.50；
第四条语句要打印变量a、b和c的值，于是下一行便会输出当前a、b和c的值，分别为10.00、20.00和7.50；
第五条语句需要先计算表达式a*(b-c)的值，然后将结果赋值给变量d，这时d的值为125.00；
第六条语句要打印变量d的值，于是在下一行输出125.00；
第七条语句为exit，执行后退出解释系统。

2021-2022学年期末考题：min解释语言A

- **名值对列表**：记录变量的名字和数值
- **辅助函数：**
  - getLine()：读取一行
  - get()：读取变量值
  - set(…)：变量赋值
  - **compute(…)：表达式计算**
  - **printList(…)：输出**
- **主程序：**



栈、队

- **栈、队的基本概念**
  - 栈、队的定义
  - 栈、队的基本操作

    栈、队列是特殊线性表(特殊性)
- **栈、队的顺序存储结构**
  - 构造原理、特点
  - 对应的插入、删除操作的算法设计

    (循环队列)
- **栈、队的链式存储结构**
  - 构造原理、特点
  - 对应的插入、删除操作的算法设计
- 栈、队的应用举例

The End!