北京航空航天大学 BEIHANG UNIVERSITY


数据结构与程序设计（信息类）

Data Structure & Programming

北京航空航天大学 数据结构课程组
软件学院 林广艳
2023年春

提纲：栈和队

栈和队是解决问题时常用的一种线性结构

这些红圈所标注的功能所涉及的数据是如何组织的？

栈和队是解决问题时常用的一种线性结构

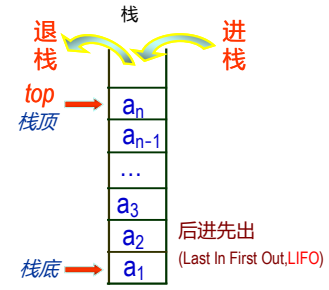
- 程序设计
  - 回溯法
  - 递归程序的执行过程
- 编译程序
  - 变量的存储空间的分配
  - 表达式的翻译与求值计算

} 栈

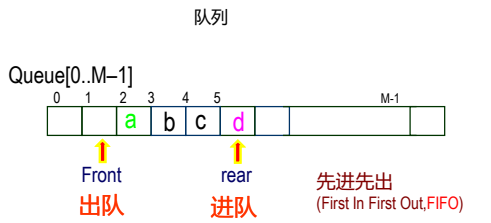
- 操作系统
  - 作业调度、进程调度
- 后续章节
  - 二叉树的层次遍历……

} 队

栈和队的整体印象



适于处理具有递归结构的数据



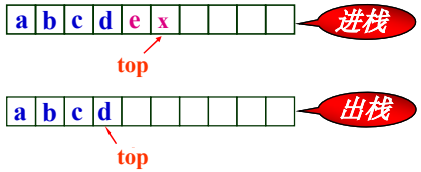
适于存放需要按一定次序依次处理但尚未处理的数据

逻辑结构：线性结构

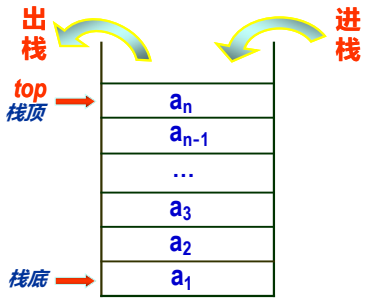
特　　点：操作仅允许在线性表一端或两端进行，是一般线性表操作的子集

3.1 栈的基本概念

- 栈（Stack，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



进栈和出栈



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

LIFO（Last-In-First-Out）

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. 测试是否为空
4. 测试是否满
5. 检索当前栈顶元素

```
void push(Stack s, ElemType e);
ElemType pos(Stack s);
int isEmpty(Stack s);
int isFull(Stack s);
ElemType getTop(Stack s);
```

特殊性

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

单选题 1分

考虑有5个元素a, b, c, d, e依次进栈，则不可能出现的出栈序列是

A **a, b, c, d, e**

B **e, d, c, b, a**

C **a, c, b, e, d**

D **a, c, e, b, d**

9

单选题 1分

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

A **2**

B **3**

C **5**

D **6**

10

单选题 1分

一个栈的入栈序列为1，2，3，…，n，其出栈序列是p1，p2，p3，…，pn。若p2=3，则p3可能取值的个数是（ ）多少？(全国考研题)

A n - 3

B n - 2

C n - 1

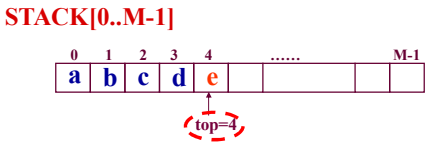
D 不确定

11

3.2 栈的顺序存储

(一) 构造原理

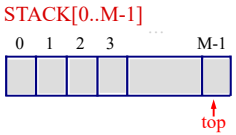
描述栈的顺序存储结构最简单的方法是利用一维数组 STACK[0..M–1] 来表示，同时定义一个整型变量(不妨取名为top) 给出栈顶元素的位置

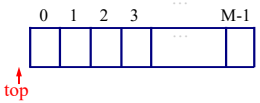


12

利用数组实现顺序栈

```
//typedef int ElemType;
#define MAXSIZE 1000   //栈的最大长度
ElemType Stack[MAXSIZE];//栈，使用数组存储
int top = -1;    //栈顶位置，-1表示栈空
```





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)

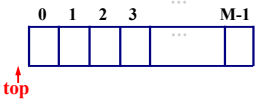
(二) 顺序栈的基本算法

1. 初始化栈

```
void initStack()
{
    top = -1;
}
```

2. 测试栈是否为空

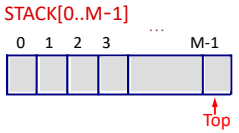
```
int isEmpty()
{
    return top == -1;
}
```



栈空,返回1,否则,返回0

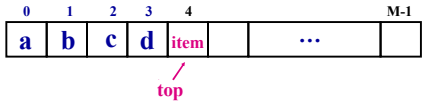
3. 测试栈是否已满

```
int isFull()
{
    return top == MAXSIZE - 1;
}
```



栈满,返回1,否则,返回0

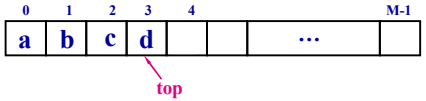
(二) 顺序栈的基本算法：进栈




```
void push(ElemType s[], ElmeType item)
{
    if (isFull())
        Error("Full Stack!");
    else
        s[++top] = item;
}
```

```
void Error(char s[])
{
    puts(s);
    exit(-1);
}
```


(二) 顺序栈的基本算法：出栈




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

(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]

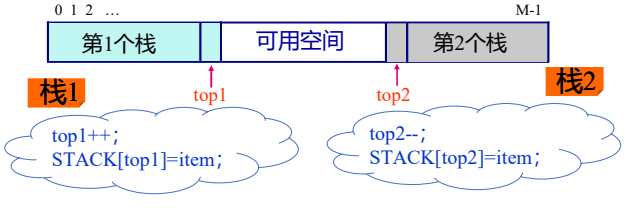
top1、top2 分别给出第1个与第2个栈的栈顶元素的位置。

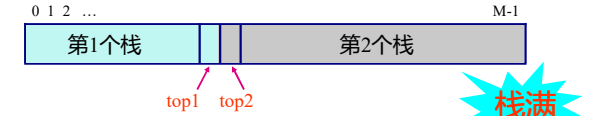
进栈

当i=1时，将item 插入第1个栈，

当i=2时，将item 插入第2个栈。

多栈共享连续空间：进栈





栈满的条件是 top1==top2–1

多栈共享连续空间：进栈（续）

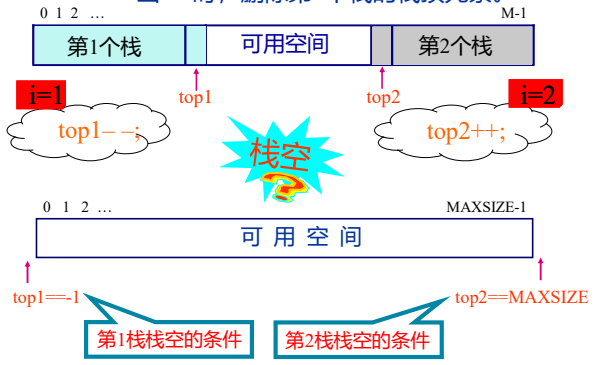
```
void push(ElemType s[], int i, ElemType item)
{
    if (top1 == top2-1) /* 栈满 */
        Error("Full Stack !");
    else
    {
        if (i == 1) /* 插入第1个栈 */
            s[++top1] = item;
        else /* 插入第2个栈 */
            s[--top2] = item;
        return;
    }
}
```


多栈共享连续空间：出栈

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。



多栈共享连续空间：出栈（续）

```
ElemType pop(ElemType s[], int i)
{
    if (i == 1)
        if (top1 == -1)
            Error("Empty Stack1 !");
        else
            return s[top1--];

    else if (top2 == MAXSIZE)
        Error("Empty Stack2 !");
    else
        return s[top2++];
}
```

对第一个栈进行操作

对第二个栈进行操作

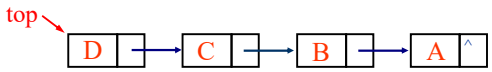
3.3 栈的链式存储：链栈

(一) 构造原理

链栈就是用一个线性链表来实现一个栈结构, 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



链栈的实现

类型定义

```
struct node
{
    ElemType 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;
}
```

2.测试栈是否为空

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

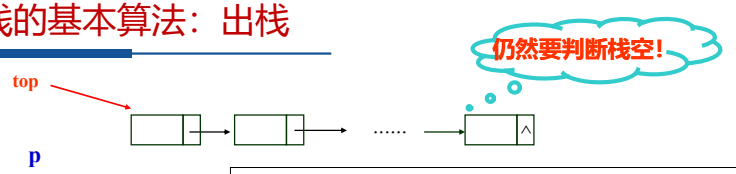
思考：是否需要定义栈满的算法？为什么？

链栈的基本算法：进栈


```
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
    }
}
```

25

链栈的基本算法：出栈




```
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
    }
```

栈的综合应用D03-01：括号匹配

- **问题**
  - ✓编写一个算法判断输入的表达式中括号是否配对（假设只考虑左、右小括号）
- **实现思路**
  - ✓ 一个表达式中的左右括号是按**最近位置配对**的，所以利用一个栈来进行求解

27

问题分析：括号匹配的情况分析

- 表达式括号匹配的情况

如：exp= “(( ))”

① ‘(‘进栈

② ‘(‘进栈

③ 遇到’)’，栈顶为’ (‘，退栈

④ 遇到’)’，栈顶为’ (‘，退栈

栈空且exp扫描完，返回真

28

问题分析：括号匹配的情况分析

◆ 表达式括号不匹配的情况

如：exp= “(( )))”

① ‘(’进栈
② ‘(’进栈
③ 遇到’)’，栈顶为’ (’，退栈
④ 遇到’)’，栈顶为’ (’，退栈
⑤ 遇到’)’，栈为空，返回假

D03-01：代码实现

```c
int matchBrackets(char *exp){
    int i = 0, match = 1; //默认匹配
    char ch, stack[MAXSIZE];//直接用数组实现栈
    int top = -1;   //栈顶指针
    //如果字符串结束，或者中途不匹配，则循环结束
    while (exp[i] != '\0' && match) {
        if (exp[i] == '(') //左括号进栈
            stack[++top] = exp[i]; // 进栈
        else if (exp[i] == ')') {//如果是右括号
            if (top >= 0)  {  //判断栈非空
                ch = stack[top--];//元素出栈
                if (ch != '(') //栈顶元素不是左括号，则不匹配
                    match = 0;
            } else match = 0; //栈空，则也不匹配
        }//其他非括号不做处理
        i++;
    }
    return match;
}
```

```c
#include <stdio.h>
#define MAXSIZE 100
int matchBrackets(char *exp);
int main()
{
    char *exp = "(()))";
    printf("%d", matchBrackets(exp));
    return 0;
}
```


综合应用D03-02：计算器（表达式计算）

【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)= ，计算表达式结果，并输出
- ✓要求
  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结果应为1

【输入形式】
从键盘输入一个以’ =’结尾的整数算术运算表达式

【输出形式】
在屏幕上输出计算结果（为整数）。

【样例1输入】
24 / (1 + 2 + 36 / 6 / 2 - 2) * (12 / 2 / 2)=

【样例1输出】
18

综合应用D03-02：问题分析

对于一般形式的表达式（通常称为中缀表达式(infix)）：

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

在（计算机）计算表达式的值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

后缀表达式的最大好处是没有括号，也不用考虑运算符的优先级！

后缀表达式(postfix)，或逆波兰表示，Reverse Polish Notation,RPN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）

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

单选题 1分

后缀表达式ABC-D/+E*对应的中缀表达式是

A **A/B-C+D*E**

B **(A+(B-C)/D)*E**

C **(A/(B-C)+D)*E**

D **(A+(B-C))/D*E**

33

中缀到后缀的转换规则

- 规则：从左至右遍历中缀表达式中每个数字和符号
- 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“）”，则将栈中元素弹出并输出，直到遇到“(”，“(”弹出但不输出
  - ✓若是“(”，“+”，“*”等符号，则从栈中弹出并输出优先级高于（或等于）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
  - ✓优先级：“+”，“-”最低，“*”，“/”次之，“(”最高
- 遍历结束，将栈中所有元素依次弹出，直到栈为空

34

后缀表达式的计算

- 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）

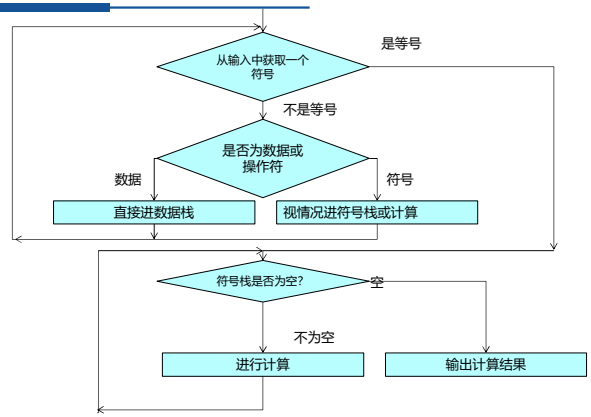
35

D03-02：算法设计

- 算法
  - ✓对于本题，我们没有必要象编译程序那样先将中缀表达式转换为后缀表达式，然后再进行计算
  - ✓为此，可设两个栈，一个为数据栈，另一个为运算符栈，在转换中缀表达的同进行表达式的计算
  - ✓主要思路为：当一个运算符出栈时，即与数据栈中的数据进行相应计算，计算结果仍存至数据栈中

36

算法设计



新的自定义类型：枚举类型

- 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
- 枚举型变量的取值仅限于**规定的一组值**之一
- 定义形式：
  - ✓enum 枚举名 { 值表 };
  - ✓例：
    - ➢enum color { red, green, yellow, white, black }; /* 枚举值是标识符 */
- 枚举变量说明
  - ✓enum color chair;
  - ✓enum color suite[10];

使用枚举类型

- **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，而是把该标识符所对应的各值表中常数值赋与变量
  - ✓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
  - ✓另外，枚举类型变量的作用范围与一般变量的定义相同
  - ✓ 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，如：enum Boolean { FALSE, TRUE };

综合应用D03-02：代码

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#define MAXSIZE 100
typedef int DataType;
enum symbol //符号类型
{NUM,OP,EQ,OTHER};
enum oper //运算类型
{EPT,ADD,MIN,MUL,DIV,LEFT,RIGHT};
//运算符优先级
int Pri[] = {-1, 0, 0, 1, 1, 2, 2};
union sym //符号值
{DataType num; enum oper op;};
enum symbol getSym(union sym *item);
void operate(enum oper op);//操作运算符
void compute(enum oper op);//进行运算
```

```
int main(){
    union sym item;
    enum symbol s;
    while ((s = getSym(&item)) ! = EQ) {
        if (s == NUM) pushNum(item.num);
        else if (s == OP) operate(item.op);
        else{
           printf("Error in the expression!\n");
           return 1;
        }
    }
    while (Otop >= 0) //将栈中所有运算符弹出计算
        compute(popOp());
    if (Ntop == 0) //输出计算结果（在数据栈）
        printf("%d\n", popNum());
    else
        printf("Error in the expression!\n");
    return 0;
}
```

```
void pushNum(DataType num);
DataType popNum();
void pushOp(enum oper op);
enum oper popOp();
enum oper topOp();
```

综合应用D03-02：代码实现（续）

```
enum symbol getSym( union sym *item){
  int  c, n;
  while((c = getchar()) != '=') {
    if(c >= '0' && c <= '9'){
     for(n=0;c>='0'&& c<='9';c=getchar())
       n = n*10 + c-'0';
       ungetc(c, stdin);
       item->num = n;
       return NUM;
     }else switch(c) {
        case '+': item->op = ADD; return OP;
        case '-': item->op = MIN; return OP;
        case '*': item->op = MUL; return OP;
        case '/': item->op = DIV; return OP;
        case '(': item->op = LEFT; return OP;
        case ')': item->op = RIGHT; return OP;
        case ' ': case '\t': case '\n': break;
        default: return OTHER;
     }
  }
  return EQ;
}
```

```
void operate(enum oper op){
  enum oper t;
  if (op != RIGHT)  {
    while(Pri[op]<=Pri[topOp()]&&topOp()!=LEFT)
      compute(popOp());
    pushOp(op);
  }else
    while ((t = popOp()) != LEFT) compute(t);
}
```

```
void compute(enum oper op){
    DataType 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;
    }
```

综合应用D03-02：代码实现（续）

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

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

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

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

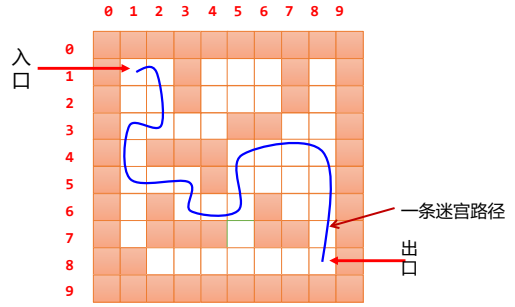
综合应用D03-02: 思考

- 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

综合应用D03-03：走迷宫

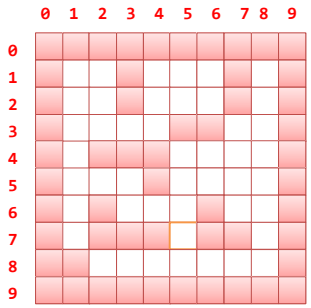
- 问题描述：给定一个N×M的迷宫图、入口与出口、行走规则，求一条从指定入口到出口的路径，所求路径应为简单路径（即路径不重复）

✓如图：N=8，M=8，空白表示通道，阴影表示障碍物；为了算法方便，一般在迷宫的外围加上一条**围墙**



综合应用D03-03：走迷宫：数据结构

◆ 利用数组来表示迷宫结构：0表示通道，1表示障碍物




```
int maze[M+2][N+2]=
{
    {1,1,1,1,1,1,1,1,1,1},
    {1,0,0,1,0,0,0,1,0,1},
    {1,0,0,1,0,0,0,1,0,1},
    {1,0,0,0,0,1,1,0,0,1},
    {1,0,1,1,1,0,0,0,0,1},
    {1,0,0,0,1,0,0,0,0,1},
    {1,0,1,0,0,0,1,0,0,1},
    {1,0,1,1,1,0,1,1,0,1},
    {1,1,0,0,0,0,0,0,0,1},
    {1,1,1,1,1,1,1,1,1,1}
};
```

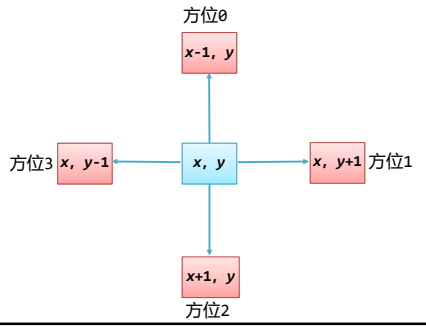
M×N

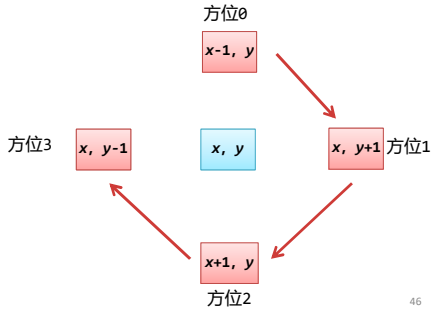
综合应用D03-03：走迷宫：寻路规则

◆ 寻路规则：上、下、左、右相邻方块行走

✓左图为(x，y)对应下一步4个方位坐标（教材为8个方位，此处简化为4个）

✓右图为迷宫行走选择顺序（从方位0开始，顺时针）





综合应用D03-03：走迷宫：寻路规则（续）

◆ 对于每一个位置，从方位0开始尝试

✓如果试不通，则**顺时针**试下一个位置

✓当选定某个可通行的位置后，把**目前所在位置以及所选的方位号**记录下来，以便当往下走不通时可以**依次一点点地退回来**，每回退一步以后，接着试**在该点上尚未试过**的下一方位

✓此外，为了避免走回到**已经进入过**的点，已经进入过的点可以设置为非0和1的值（如设置为：-1）

◆为方便**回退**，需要利用栈记录**当前位置和所选的方位号**

综合应用D03-03：走迷宫-代码实现（头部定义）

```
#include <stdio.h>
#define LEN 50
#define DIR 4
typedef struct _Box
{
    int x, y; // 当前位置的行号、列号
    int di;   // 下一个可走相邻方位的方位号
} Box;
Box stack[LEN * LEN]; // 记录位置的堆栈
// 方位计算规则
int next[DIR][2] = {{-1, 0}, {0, 1}, {1, 0}, {0, -1}};
int top = -1;                       // 栈顶指针
int push(int x, int y, int pos); // 进栈
Box pop();                          // 出栈
// 走迷宫，约定（1，1）为入口，（ex,ey）为出口
int mazePath(int maze[][LEN], int ex, int ey);
```

综合应用D03-03：走迷宫-代码实现（主函数）

```c
int main(){
    int maze[LEN][LEN];
    int n, m, i, j;
    scanf("%d%d", &n, &m); // 读入迷宫的大小
    // 输入迷宫各位置情况，0表示可通行，1表示障碍，保存在第1~n、1~m列中
    // 约定（1，1）为入口，（n,m）为出口
    for (i = 1; i <= n; i++)
        for (j = 1; j <= m; j++)
            scanf("%d", &maze[i][j]);
    // 将迷宫的最外圈设置为障碍物
    for (j = 0; j <= m; j++) { // 将第0行、第n+1行设置为障碍物
        maze[0][j] = 1;
        maze[n + 1][j] = 1;
    }
    for (i = 0; i <= n; i++) { // 第0列、第m+1列设置为障碍物
        maze[i][0] = 1;
        maze[i][m + 1] = 1;
    }
    mazePath(maze, n, m);
    return 0;
}
```


综合应用D03-03：走迷宫-代码实现（栈操作）

```c
int push(int x, int y, int pos){
    ++top;
    stack[top].x = x;
    stack[top].y = y;
    stack[top].di = pos;
}
Box pop(){
    if (top == -1)
    {
        puts("stack is empty");
        exit(1);
    }
    return stack[top--];
}
int isEmpty(){
    return top == -1;
}
```


综合应用D03-03：走迷宫-代码实现（走迷宫）

```c
int mazePath(int maze[][LEN], int ex, int ey) {
    int x, y, di, nx, ny;//当前位置和方位、下一个位置
    int i, k, found;
    Box pos, path[LEN * LEN];//当前位置、迷宫路径
    push(1, 1, -1);  // 入口位置进栈
    maze[1][1] = -1; // 入口位置设置为已访问
    while (!isEmpty()) {
        pos = pop();//出栈
        x = pos.x; y = pos.y; di = pos.di;
        if (x == ex && y == ey) { // 找到出口，输出该路径
            k = 0;
            path[k++] = pos;
            while (!isEmpty()) path[k++] = pop(); // 弹出栈里的路径
            for (i = k - 1; i >= 0; i--) { // 输出路径信息
                printf("(%d, %d) ", path[i].x, path[i].y);
                if ((i + 2) % 5 == 0)  putchar('\n');
            } //endfor
            return 1;
        }  //endif
```

综合应用D03-03：走迷宫-代码实现（走迷宫）

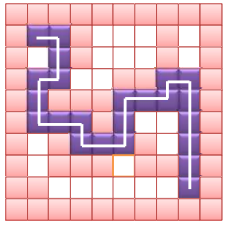
```c
        found = 0; // 没到出口，找下一个位置
        while (di < 3 && !found) { // 找下一个相邻可走位置
            di++;
            nx = x + next[di][0];
            ny = y + next[di][1];
            if (maze[nx][ny] == 0) found = 1;
        } //endwhile
        if (found) {//如果找到一个位置，则继续
            pos.di = di;
            push(x, y, di);
            push(nx, ny, -1);
            maze[nx][ny] = -1;
        }
        else maze[x][y] = 0; // 没有合适的路径，退栈，往回走
    } //endwhile
    return 0;
}
```

综合应用D03-03：走迷宫

迷宫路径如下：

(1,1) (1,2) (2,2) (3,2) (3,1)
(4,1) (5,1) (5,2) (5,3) (6,3)
(6,4) (6,5) (5,5) (4,5) (4,6)
(4,7) (3,7) (3,8) (4,8) (5,8)
(6,8) (7,8) (8,8)

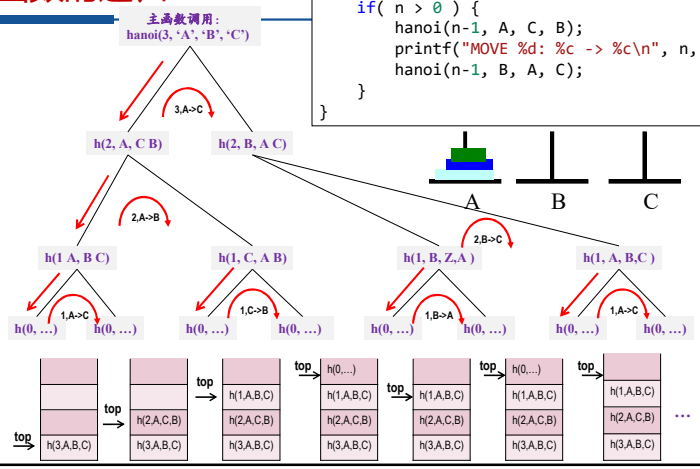
```
8 8
0 0 1 0 0 0 1 0
0 0 1 0 0 0 1 0
0 0 0 0 1 1 0 0
0 1 1 1 0 0 0 0
0 0 0 1 0 0 0 0
0 1 0 0 0 1 0 0
0 1 1 1 0 1 1 0
1 0 0 0 0 0 0 0
```



显然，这个解不是最优解，即不是最短路径。为什么？

栈与函数的递归

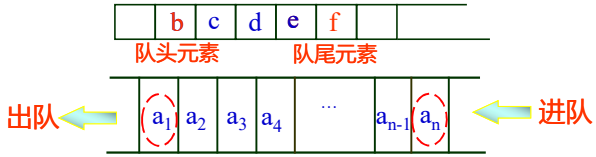
```
//汉诺塔(hanoi tower)游戏
void hanoi( int n, char A, char B, char C){
    if( n > 0 ) {
        hanoi(n-1, A, C, B);
        printf("MOVE %d: %c -> %c\n", n, A, C);
        hanoi(n-1, B, A, C);
    }
}
```



提纲：栈和队

3.2.1 队的基本概念

- 队（Queue，队列）是一种只允许在表的一端进行插入操作，而在表的另一端进行删除操作的线性表
  - ✓允许插入的一端称为队尾，队尾元素的位置由rear指出
  - ✓允许删除的一端称为队头，队头元素的位置由front指出



FIFO(First-In-First-Out)

队的基本操作

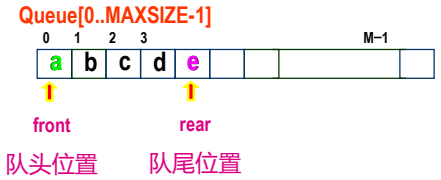
1. 队头插入：进队、入队　　`void enQueue(Queue q, ElemType);`
2. 队尾删除：出队、退队　　`ElemType deQueue(Queue q);`
3. 测试队是否满　　`isFull(Queue q);`
4. 测试队是否空　　`isEmpty(Queue q);`
5. 创建一个空队　　`Queue initQueue();`

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

3.2.2 队的顺序存储结构

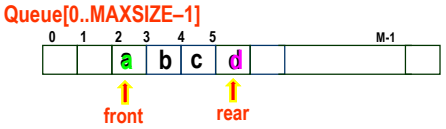
(一)构造原理

在实际程序设计过程中，通常借助一个一维数组Queue[0..M-1]来描述队的顺序存储结构，同时，设置两个变量 front与rear分别指出当前队头元素与队尾元素的位置



队的顺序存储结构

rear 指出实际队尾元素所在的位置
front 指出实际队头元素所在位置
count 指出实际队中元素个数



初始时, 队为空, 有

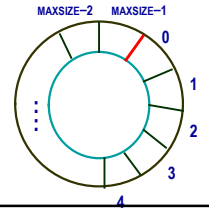
front= 0　rear= –1　count=0

测试队为空的条件是

count==0

（二）循环队列

- 在实际应用中，由于队元素需要频繁进出，上述结构很容易溢出
  - ✓即rear到达数组尾，而实际队中元素并没有超出数组大小
  - ✓因此，在实际应用中通常将队设计成一个循环队列，从而提高空间利用率
- 循环队列：把队列(数组)设想成头尾相连的循环表，使得数组前部由于删除操作而导致的无用空间尽可能得到重复利用



（三）循环队列的基本操作

1. 初始化队列

```
typedef int ElemType;
#define MAXSIZE 1000
//利用数组实现队列的顺序存储
ElemType QUEUE[MAXSIZE];
//定义队头、队尾和元素个数变量
int Front, Rear, Count;
```

```
void initQueue()
{
    Front = 0;
    Rear = MAXSIZE - 1;
    Count = 0;
}
```

2. 测试队列是否为空或满

```
int isEmpty()
{
    return Count == 0;
}
```

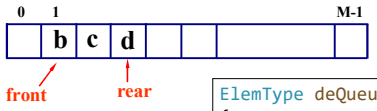
```
int isFull()
{
    return Count == MAXSIZE;
}
```


3.插入（进队）算法


```
void enQueue(ElemType queue[], ElemType item)
{
    if (isFull()) // 队满，插入失败
        Error("Full queue !");
    else
    {
        Rear = (Rear + 1) % MAXSIZE;

        queue[Rear] = item;
        Count++; // 队未满，插入成功
    }
}
```


4.删除（出队）算法




```
ElemType deQueue(ElemType queue[])
{
    ElemType e;
    if (isEmpty())
        Error("Empty queue !"); // 队空，删除失败
    else
    {
        e = queue[Front];
        Count--; // 队非空，删除成功

        Front = (Front + 1) % MAXSIZE;

        return e;
    }
}
```


3.2.2 队的链式存储结构

(一)构造原理

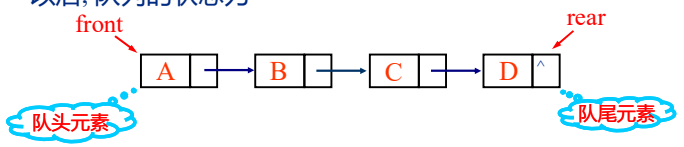
队列的链式存储结构是用一个线性链表表示一个队列，指针front与rear分别指向实际队头元素与实际队尾元素所在的链结点

空队的标志是：front == NULL

例 在一个初始为空的链接队列中依次插入数据元素

A, B, C, D

以后, 队列的状态为



（二）循环队列的基本操作

1. 初始化队列

```
void initQueue()
{
    Front = NULL;
    Rear = NULL;
}
```

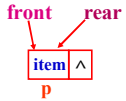
2. 测试队列是否

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

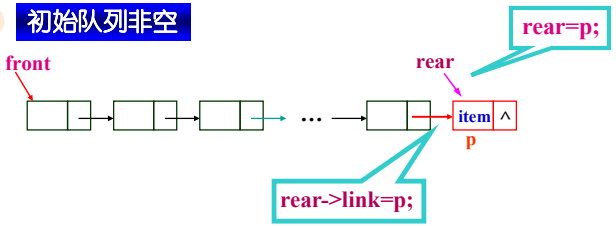
```
#include <stdio.h>
//链式队列定义
typedef int ElemType;
typedef struct _QNode
{
    ElemType data;
    struct _QNode *link;
} QNode, *QNodeptr;
//队头队尾指针
QNodeptr Front, Rear;
```


3. 插入（进队）

1 初始队列为空 front=rear=NULL



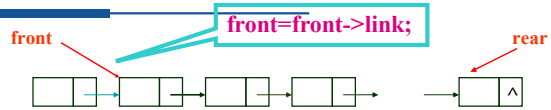
2 初始队列非空



进队：代码实现

```
void enLQueue(ElemType item)
{
    QNodeptr p;
   p=(QNodeptr)malloc(sizeof(QNode))); // 申请链结点
   p->data = item;
   p->link = NULL;
   if (Front == NULL) // 插入空队的情况
       Front = p;
   else // 插入非空队的情况
       Rear->link = p;
   Rear = p;
}
```

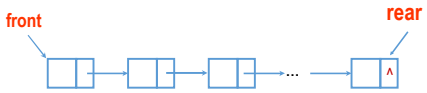

4.删除（出队）




```
ElemType deLQueue(){
   QNodeptr p;
   ElemType item;
   if (isEmpty()) // 队为空，删除失败
       Error("Empty queue!");
   else
   {
       p = Front;
       Front = Front->link;
       item = p->data;
       free(p);
       return item; // 队非空，删除成功
   }
}
```

5.销毁一个队

- 所谓销毁一个队是指将队列所对应的链表中所有结点都删除，并且释放其存储空间，使队成为一个空队(空链表)
  - ✓归结为线性链表的删除




```
void destroyLQueue()
{
   while (Front != NULL)
   { //队非空时
       Rear = Front->link;
       free(Front); //释放一个结点空间
       Front = Rear;
   }
}
```

队的综合应用D03-04：银行排队模拟（Simulation）

【问题描述】某银行网点有五个服务窗口，分别为三个对私、一个对公和一个外币窗口。通常对私业务人很多，其它窗口人则较少，可临时改为对私服务。假设当对私窗口客户平均排队人数超过7人时，客户将有抱怨，此时银行可临时将其它窗口中一个或两个改为对私服务，当客户少于7人时，将恢复原有业务。设计一个程序用来模拟银行服务

【输入】首先输入一个整数表示时间周期数，然后再依次输入每个时间周期中因私业务的到达客户数。

注：一个时间周期指的是银行处理一笔业务的平均处理时间，可以是一分钟、三分钟或其它。例如：

- ✓ 6
- ✓ 2 5 13 11 15 9
- ✓ 说明：表明在6个时间周期内，第1个周期来了2个（ID分别为1,2），第2个周期来了5人（ID分别为3,4,5,6,7），以此类推

【输出】每个客户等待服务的时间周期数

70

模拟（仿真）问题

- 一个系统模仿另一个系统行为的技术称为模拟（Simulation，仿真），如飞行模拟器
- 模拟可以用来进行方案认证、人员培训和改进服务。计算机技术常用于模拟系统中
  - ✓ 生产者-消费者（Server-Custom）是常见的应用模式，见于银行、食堂、打印机、医院、超市…提供服务和使用服务的应用中
  - ✓ 这类应用的主要问题是消费者如果等待（排队）时间过长，会引发用户抱怨，影响服务质量
  - ✓ 如果提供服务者（服务窗口）过多，将提高运管商成本（排队论-queuing theory）

71

问题3.2：问题分析及算法设计

在**生产者-消费者**应用中消费者显然是先来先得到服务。在此，可用一个**队列**来存放等待服务的**客户**队列。

每个客户有二个基本属性：排队序号和等待时间（时间周期数）：

```
struct  cust {
    int id;        //客户排队序号
    int wtime;     //客户等待服务的时间（时间周期数）
};
Struct cust Cqueue[MAXSIZE]; //等待服务的客户队列，一个循环队列
```

为了简化问题，可用一个变量来表示银行当前提供服务的**窗口数**：

```
int  snum;
```

在本问题中，该变量的取值范围为**3<= snum <= 5**

72

问题3.2：问题分析及算法设计（续）

主要算法：

```
for(clock=1; ; clock++) //在每个时间周期内
{
   1. If 客户等待队列非空
      将每个客户的等待时间增加一个时间单元；
   2. If(clock <= simulationtime)
      2.1 如果有新客户到来（从输入中读入本周期内新来客户数），将其入队；
      2.2 根据等待服务客户数重新计算服务窗口数；----------可能增加窗口
   3. If 客户等待队列非空
      3.1 从客户队列中取（出队）相应数目（按实际服务窗口数）客户获得服务；
      3.2 然后根据等待服务客户数重新计算服务窗口数；------可能减少窗口
   4. 如果客户等待队列为空 && 银行服务时间周期到，则结束模拟
}
```

代码实现

```
#include <stdio.h>
#include  <stdlib.h>
#define  MAXSIZE 200  //队列容量
#define THRESHOLD  7  //窗口增加阈值
#define MAXSVR  5   //最大服务窗口数
#define MINSVR  3    //最小服务窗口数
typedef struct {
    int id;
    int wtime;
} CustType;
int Winnum=MINSVR; //提供服务的窗口数
//更新等待队列中客户等待时间
void  updateCustqueue();
void  enCustqueue(CustType c); //客户入等待队列
CustType deCustqueue(); //客户出队
int  getCustnum();   //获取队中等待客户人数
int  isFull();
int  isEmpty();
void arriveCust(); //获取新客户，并加至等待队列中
int  service(); //银行从队列中获取客户进行服务
```

```
int main()
{
    int clock, simulationtime;
    scanf("%d",&simulationtime);
    for(clock=1; ; clock++) {
        updateCustqueue();//更新客户等待时间
        if(clock <= simulationtime )
            arriveCust();  //新客户加入队列
                    //并根据需要调整服务窗口数量
        if(service()==0 && clock>simulationtime)
            break;//等待队列为空且服务时间到
                    //不会有新客户
    }
    return 0;
}
```


代码实现（续）

内部静态变量，作用域为当前函数，生存周期同全局变量。它只初始化一次，每次函数调用时，上次的值仍在

```
void arriveCust(){
    int i,n;
    static int count=1;
    CustType c;
    scanf("%d", &n);
    for(i=0; i<n; i++){
        c.id = count++; c.wtime = 0;
        enCustqueue(c);
    }
    while((getCustnum() / Winnum) >= THRESHOLD && Winnum<MAXSVR)
        Winnum++; //增加服务窗口
}
int service(){
    int i;
    CustType c;
    for(i=0; i<Winnum; i++)
       if(isEmpty() ) return 0;
       else {
           c = deCustqueue();printf("%d :%d\n", c.id, c.wtime);}
     if((getCustnum() / Winnum) < THRESHOLD && Winnum>MINSVR)
          Winnum--;
    return 1;
}
```

代码实现（续）

```
#include <stdio.h>
#include  <stdlib.h>
#define  MAXSIZE 200  //队列容量
typedef struct {
       int id;
       int wtime;
} CustType;
//客户等待队列
static  CustType Cqueue[MAXSIZE];
//队头队尾指示器
static  int Cfront=0;
//队尾
static  int  Crear = -1;
//队中元素个数
static  int Cnum=0;
```

```
void enCustqueue(CustType c){
    if (isFull()) { /* 队满，插入失败 */
        printf("Full queue!"); exit(-1);
    } else{ /* 队未满，插入成功 */
        Crear = (Crear + 1) % MAXSIZE;
        Cqueue[Crear] = c;  Cnum++;
    }}
CustType deCustqueue(){
    CustType c;
    if (isEmpty())
    { printf("Empty queue!"); exit(-1);}
    else{ /* 队非空，删除成功 */
        c = Cqueue[Cfront];  Cnum--;
        Cfront = (Cfront + 1) % MAXSIZE;
        return c;
}}
void updateCustqueue(){
    int i;
    for (i = 0; i < Cnum; i++)
        Cqueue[(Cfront + i) % MAXSIZE].wtime++;
}
int isEmpty() { return Cnum == 0; }
int isFull() { return Cnum == MAXSIZE; }
int getCustnum() { return Cnum; }
```

问题3.2：思考

- 在本问题中，当前服务窗口平均排队等待服务的客户人员数小于某个阈值时，临时窗口将不再提供服务，一来该策略不是最优，二来也不符合实际情况
- 现增加如下规则：
  - ✓外币和对公窗口应优先处理本业务，即当有对应业务（有客户等待时）时应优先处理，只有当本业务没有排队客户时，才能处理对私业务
  - ✓当外币和对公窗口没有等待客户同时对私窗口有等待客户排队时，将处理对私业务（资源利用最大化）

优先队列（Priority Queue）

- 在实际应用时，前述简单队列结构是不够的，先入先出机制需要使用某些优先规则来完善。如：
  - ✓在服务行业，通常有残疾人、老人优先
  - ✓在公路上某些特殊车辆（如救护车、消防车）优先
  - ✓在操作系统进程调度中，具有高优先级的进程优先执行
- 优先队列（Priority Queue）
  - ✓根据元素的优先级及在队列中的当前位置决定出队的顺序

优先队列的实现

方法一：使用两种变种链表实现
- ✓一种链表是所有元素都按进入顺序排列（队），取元素效率为O(n)
- ✓另一种链表是根据元素的优先级决定新增位置（按优先级排序），新增元素效率为O(n)

方法二：使用一个链表和一个指针数组
- ✓链表用于存放元素，一个指向链表的指针数组用于确定新加入的元素应该在哪个范围中（按优先级），算法的时间复杂度为$O(\sqrt{n})$。（J.O.Hendriksen提出）

方法三：用一个堆（Heap）结构实现
- ✓这是常用的一种高效实现优先队列的方法（原理将在树中讲解），算法的时间复杂度为$O(\log_2 N)$

小结

- 栈和队的基本概念
- 栈和队的顺序存储
- 栈和队的链式存储
- 栈的综合应用：回溯问题
- 队的综合应用：排队问题