

数据结构

北京邮电大学 网络安全学院

武 斌



上章内容

上一章（线性表）内容：

- 了解线性表的概念及其逻辑结构特性
- 理解顺序存储结构和链式存储结构的描述方法
- 掌握线性表的基本操作及算法实现
- 重点掌握线性链表的存储结构及算法实现
- 分析两种存储结构的时间和空间复杂度





本章课程学习目标

学习完本章课程，您应该能够：

- 掌握栈和队列这两种抽象数据类型的特点
- 熟练掌握栈类型的两种实现方法
- 理解递归算法执行过程中栈的状态变化过程





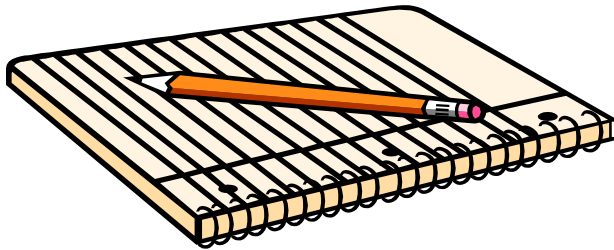
本章课程内容（第三章 栈和队列）

● 3.1 栈

● 3.2 栈的应用举例

● 3.3 队列

● 3.4 队列应用举例





第三章 栈和队列

- **栈**和**队列**是在程序设计中被广泛使用的两种**线性数据结构**。
- 从数据结构角度看，栈和队列是操作受限的线性表。
- 从数据类型角度看，栈和队列是与线性表不同的两类重要的抽象数据类型。
- 它们的**特点**在于基本操作的特殊性，栈必须按“**后进先出**”的规则进行操作，而队列必须按“**先进先出**”的规则进行操作。和线性表相比，它们的**插入**和**删除**操作受**更多的约束和限定**，故又称为限定性的线性表结构。



第三章 栈和队列

- 线性表与栈及队列的插入和删除操作对比如下：



插入

删除

线性表：	$\text{Insert}(L, i, x)$ $(1 \leq i \leq n+1)$	$\text{Delete}(L, i)$ $(1 \leq i \leq n)$	线性表允许在表内任一位置进行插入和删除
栈：	$\text{Insert}(L, n+1, x)$	$\text{Delete}(L, n)$	栈只允许在表尾一端进行插入和删除
队列：	$\text{Insert}(L, n+1, x)$	$\text{Delete}(L, 1)$	队列只允许在表尾一端进行插入，在表头一端进行删除



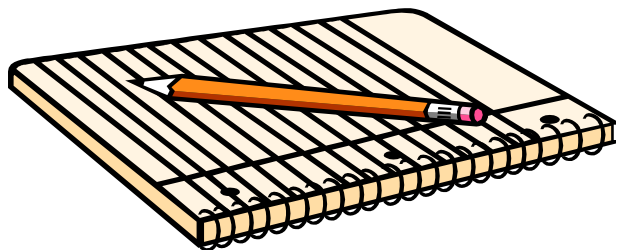
栈的类型定义

● 3.1 栈的类型定义

● 3.2 栈的应用举例

● 3.3 队列的类型定义

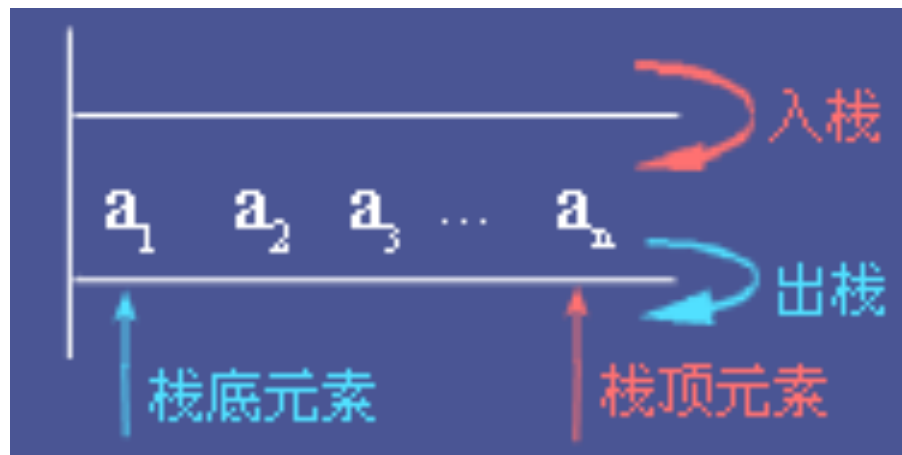
● 3.4 队列应用举例





栈的类型定义

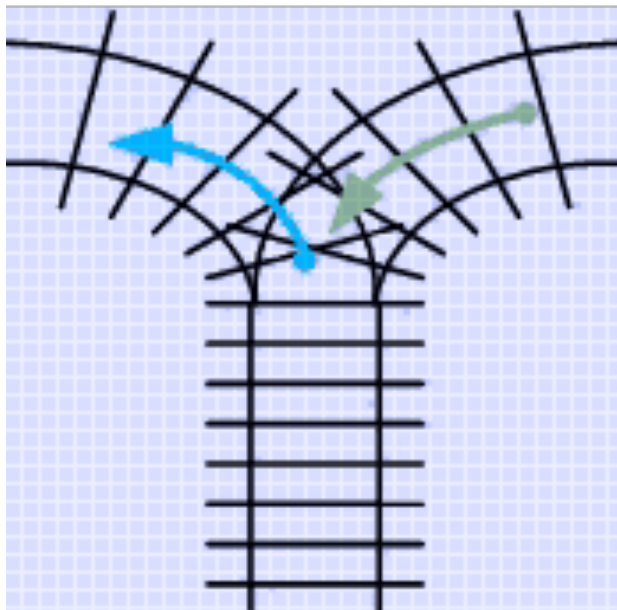
- **栈(Stack)**是限定只能在表的一端进行插入和删除操作的线性表。在表中，允许插入和删除的一端称作“**栈顶(top)**”，不允许插入和删除的另一端称作“**栈底(bottom)**”。当表中没有元素时称为**空栈**。
- 假设栈 $S=(a_1, a_2, a_3, \dots, a_n)$ ，则 a_1 称为栈底元素， a_n 为栈顶元素。栈中元素按 $a_1, a_2, a_3, \dots, a_n$ 的次序进栈，退栈的第一个元素应为栈顶元素。换句话说，栈的修改是按后进先出的原则进行的。因此，栈称为**后进先出表 (LIFO)**。





栈的类型定义

- 通常称往栈顶插入元素的操作为“**入栈**”，称删除栈顶元素的操作为“**出栈**”。因为后入栈的元素先于先入栈的元素出栈，故被称为是一种“**后进先出**”的结构，因此又称**LIFO**（Last In First Out的缩写）表。
- 下图所示的铁路调度站形象地表示栈的这个特点。





栈的类型定义

- 栈的类型定义如下：

ADT Stack {

数据对象： $D = \{ a_i \mid a_i \in \text{ElemSet}, i=1,2,\dots,n, n \geq 0 \}$

数据关系： $R1 = \{ \langle a_{i-1}, a_i \rangle \mid a_{i-1}, a_i \in D, i=2,\dots,n \}$

约定 a_n 端为栈顶， a_1 端为栈底。

基本操作：

InitStack(&S)

操作结果：构造一个空栈 S。

DestroyStack(&S)

初始条件：栈 S 已存在。

操作结果：栈 S 被销毁。

ClearStack(&S)

初始条件：栈 S 已存在。

操作结果：将 S 清为空栈。



栈的类型定义

StackEmpty(S)

初始条件：栈 **S** 已存在。

操作结果：若栈 **S** 为空栈，则返回**TRUE**，否则返回**FALSE**。

StackLength(S)

初始条件：栈 **S** 已存在。

操作结果：返回栈 **S** 中元素个数，即栈的长度。

GetTop(S, &e)

初始条件：栈 **S** 已存在且非空。

操作结果：用 **e** 返回**S**的栈顶元素。

Push(&S, e)

初始条件：栈 **S** 已存在。

操作结果：插入元素 **e** 为新的栈顶元素。



栈的类型定义

Pop(&S, &e)

初始条件：栈 S 已存在且非空。

操作结果：删除 S 的栈顶元素，并用 e 返回其值。

StackTraverse(S, visit())

初始条件：栈 S 已存在且非空，visit()为元素的访问函数。

操作结果：从栈底到栈顶依次对S的每个元素调用函数visit()，一旦 visit()失败，则操作失败。

} ADT Stack



栈的存储表示和操作的实现

- 和线性表类似，栈也有顺序存储和链式存储两种存储表示，其顺序存储结构简称为顺序栈。
 - ➔ 和顺序表类似，对顺序栈也需要事先为它分配一个可以容纳最多元素的存储空间，**base**为这个存储空间的基地址，也即一维数组的地址。
 - ➔ 从名称来讲，“栈顶指针”意为指示栈顶元素在栈中的位置，但它的值实际是栈中元素的个数，和顺序表中的length 值的意义相同。



栈的存储表示和操作的实现

- 顺序栈类型的定义:

→ // 结构定义:

```
typedef struct {  
    SElemType *base; // 存储空间基址  
    SElemType *top;   // 栈顶指针  
    int stacksize;    // 允许的最大存储空间以元素为单位  
} SqStack;
```



栈的存储表示和操作的实现

→ // 基本操作接口(函数声明):

void InitStack (SqStack &S, **int** maxsize);

// 构造一个最大存储容量为 maxsize 的空栈S。

void DestroyStack (SqStack &S);

// 销毁栈S，S 不再存在。

void ClearStack (SqStack &S);

// 将 S 置为空栈。

bool StackEmpty (SqStack S);

// 若栈 S 为空栈，则返回 TRUE，否则返回 FALSE。

int StackLength (SqStack S);

// 返回S的元素个数，即栈的长度。



栈的存储表示和操作的实现

→ // 基本操作接口(函数声明续):

bool GetTop (SqStack S, ElemType &e);

// 若栈不空, 则用 **e** 返回**S**的栈顶元素, 并返回**TRUE**;

// 否则返回 **FALSE**。

bool Push (SqStack &S, ElemType e);

// 若栈的存储空间不满, 则插入元素 **e** 为新的栈顶元素,

// 并返回 **TRUE**; 否则返回**FALSE**。

bool Pop (SqStack &S, ElemType &e);

// 若栈不空, 则删除**S**的栈顶元素, 用**e**返回其值, 并返回**TRUE**;

// 否则返回**FALSE**。

void StackTraverse(SqStack S, **void** (*visit(ElemType))

// 依次对**S**的每个元素调用函数 **visit()**, 一旦 **visit()**失败,

// 则操作失败。



栈的存储表示和操作的实现

- 以下给出其中4个函数的定义：

```
→ void InitStack (SqStack &S, int maxsize)
{
    // 构造一个最大存储容量为 maxsize 的空栈 S
    if (maxsize == 0)
        maxsize = MAXLISTSIZE;
    S.base = new SElemType[maxsize];
    if (!S.base) exit(1);    // 存储分配失败
    S.stacksize = maxsize;
    S.top = 0;               // 空栈中元素个数为0
}
```



栈的存储表示和操作的实现

```
→ bool GetTop (SqStack S, ElemType &e)
{
    // 若栈不空，则用 e 返回S的栈顶元素，并返回TRUE;
    // 否则返回FALSE
    if (S.top == 0) return FALSE;
    e = *(S.base + S.top-1);    // 返回非空栈中栈顶元素
    return TRUE;
}
```

※ 注意：*(S.base + S.top-1)是S.base[S.top-1] 的另一种写法，其实质相同。



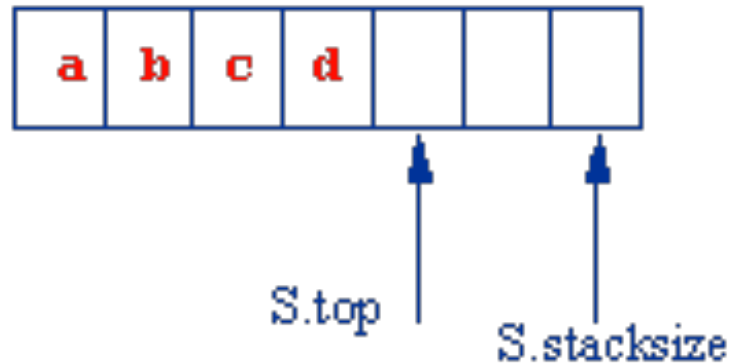
栈的存储表示和操作的实现

```
→ bool Push (SqStack &S, ElemType e)
{
    // 若栈的存储空间不满，则插入元素 e 为新的栈顶元素，
    // 并返回 TRUE；否则返回 FALSE
    if (S.top == S.stacksize)    // 栈已满，无法进行插入
        return FALSE;
    *(S.base + S.top) = e;        // 插入新的元素
    ++S.top;                      // 栈顶指针后移
    return TRUE;
}
```



栈的存储表示和操作的实现

- 用图表示顺序栈如下：



→图中的顺序栈的最大容量为7，当前栈中元素个数为4，因此，我们也可认为栈顶指针总是指向在栈顶元素的后面一个位置上。



栈的存储表示和操作的实现

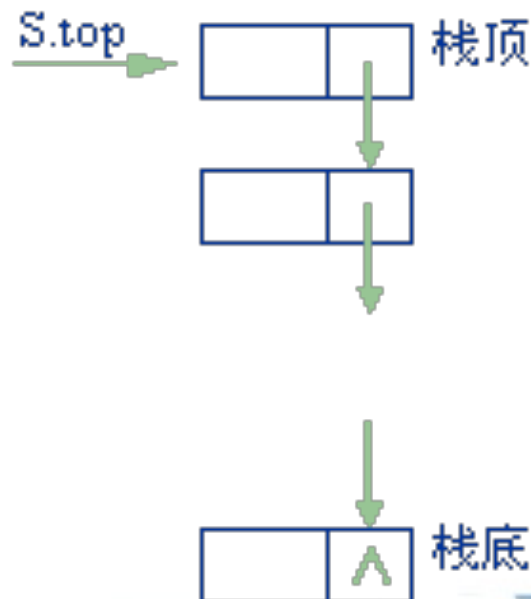
```
→ bool Pop (SqStack &S, ElemType &e)
{
    // 若栈不空，则删除S的栈顶元素，用 e 返回其值，
    // 并返回 TRUE；否则返回 FALSE
    if (S.top == 0) return FALSE;
    e = *(S.base + S.top-1);    // 返回非空栈中栈顶元素
    --S.top;                    // 栈顶指针前移
    return TRUE;
}
```

- 显然，顺序栈的基本操作的时间复杂度，除"遍历"之外，均为常量级的，即 **$O(1)$** 。
- **bo3-1.cpp main3-1.cpp**



栈的存储表示和操作的实现

- **链栈**即为栈的**链式存储结构**。
- 链栈的定义更简单，结点结构和单链表中的结点结构相同，无须重复定义。由于栈只在栈顶作插入和删除操作，因此**链栈中不需要头结点**，但要**注意**链栈中指针的方向是从栈顶指向栈底的，这正好和单链表是**相反**的。





栈的存储表示和操作的实现

- **链栈**的结构定义:

```
typedef struct {  
    SLink top;    // 栈顶指针  
    int length;   // 栈中元素个数  
} Stack;
```

- 此时只需要对顺序栈的基本操作接口作两处改动，便可作为链栈的基本操作接口。
 - ➔ 其一，初始化时不需要“maxsize”的参数，因为它不需要事先分配空间；
 - ➔ 其二，在进行入栈操作时不需要顾忌栈的空间是否已经被填满。



栈的存储表示和操作的实现

→ **void** InitStack (Stack &S)

{

 // 构造一个空栈 S

 S.top = NULL; // 设栈顶指针的初值为"空"

 S.length = 0; // 空栈中元素个数为0

} // InitStack



栈的存储表示和操作的实现

```
→ void Push ( Stack &S, ElemType e )  
{  
    // 在栈顶之上插入元素 e 为新的栈顶元素  
    p = new LNode;    // 建新的结点  
    if(!p) exit(1);    // 存储分配失败  
    p -> data = e;  
    p -> next = S.top; // 链接到原来的栈顶  
    S.top = p;         // 移动栈顶指针  
    ++S.length;        // 栈的长度增1  
} // Push
```



栈的存储表示和操作的实现

```
→ bool Pop ( Stack &S, SElemType &e )
{ // 若栈不空，则删除S的栈顶元素，用 e 返回其值，
  // 并返回 TRUE；否则返回 FALSE
  if ( !S.top ) return FALSE;
  else
  {   e = S.top -> data;    // 返回栈顶元素
      q = S.top;
      S.top = S.top -> next; // 修改栈顶指针
      --S.length;           // 栈的长度减1
      delete q;             // 释放被删除的结点空间
      return TRUE;
  }
} // Pop
```

● **bo3-5.cpp main3-5.cpp**



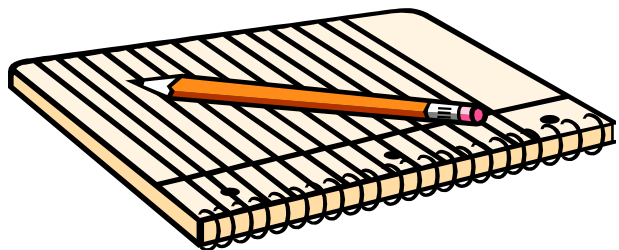
栈的应用举例

3.1 栈的类型定义

3.2 栈的应用举例

3.3 队列的类型定义

3.4 队列的应用举例





栈的应用举例

- 由于栈结构具有的后进先出的固有特性，致使栈成为程序设计中常用的工具。

- **例3-1 数制转换**

- 十进制N和其它进制数的转换是计算机实现计算的基本问题，其解决方法很多，其中一个简单算法基于下列原理：

$$N = (N \text{ div } d) \times d + n \text{ mod } d$$

(其中:div为整除运算,mod为求余运算)

- 例如: $(1348)_{10} = (2504)_8$

- 演示 3-2-1



栈的应用举例

- 从动画演示的计算过程可见，这八进制的各个数位产生的顺序是**从低位到高位**的，而打印输出的顺序，一般来说**应从高位到低位**，这恰好和计算过程**相反**。
- 因此，需要先保存在计算过程中得到的八进制数的各位，然后逆序输出，因为它是按“后进先出”的规律进行的，所以用栈最合适。
- 具体算法如下页**算法3.1**所示。
- algo3-1.cpp**



栈的应用举例

● 算法3.1

void conversion ()

{// 对于输入的任意一个非负十进制整数，打印输出与其等值的八进制数

 InitStack(S); // 构造空栈

 scanf ("%",N); // 输入一个十进制数

while(N)

 {

 Push(S,N % 8); // “余数” 入栈

 N = N/8; // 非零 “商” 继续运算

 } // while

while (!StackEmpty)

 { // 和 “求余” 所得相逆的顺序输出八进制的各位数

 Pop(S,e);

 printf("%d",e);

 } // while

} // conversion



栈的应用举例

- **例3-2** 假设表达式中允许包含两种括号：圆括号和方括号，其嵌套的顺序随意，如 $([] ())$ 或 $[([] [])]$ 等为正确的匹配， $[(])$ 或 $([] ()$ 或 $(()))$ 均为错误的匹配。
- 检验括号是否匹配的方法可用“**期待的急迫程度**”这个概念来描述。即后出现的“左括弧”，它等待与其匹配的“右括弧”出现的“急迫”心情要比先出现的左括弧高。换句话说，对“左括弧”来说，后出现的比先出现的“**优先**”等待检验，对“右括弧”来说，每个出现的右括弧要去找在它之前“**最后**”出现的那个左括弧去**匹配**。
- 显然，必须将先后出现的左括弧**依次保存**，为了反映这个优先程度，保存左括弧的结构**用栈最合适**。这样对出现的右括弧来说，只要“**栈顶元素**”**相匹配**即可。如果在栈顶的那个左括弧正好和它匹配，就可将它从栈顶**删除**。



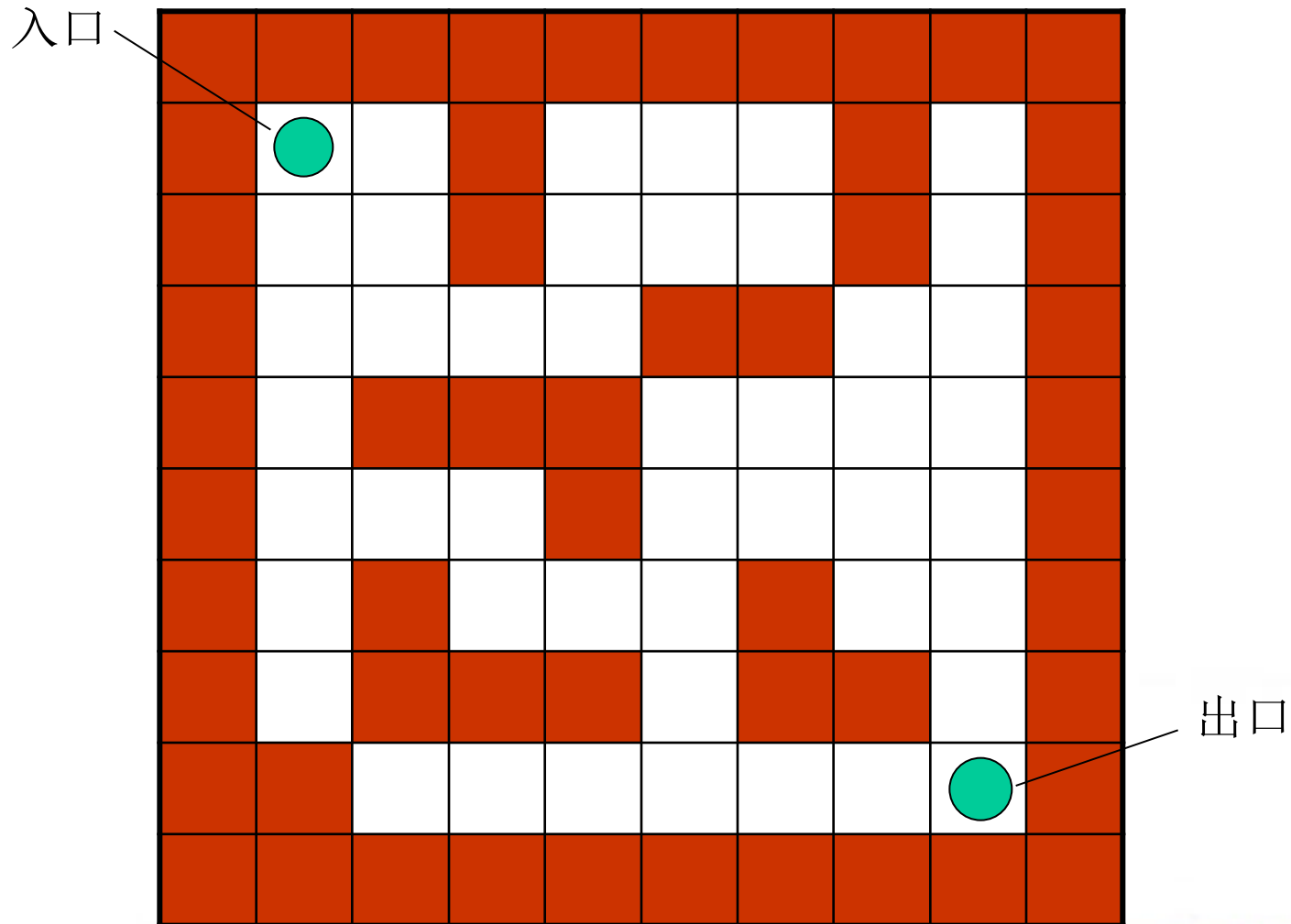
栈的应用举例

- 那么，什么样的情况是“**不匹配**”的情况呢？上面列举的三种错误匹配从“期待匹配”的角度描述即为：
 - ➔ 1. **大失所望**：来的右括弧非是所“期待”的；
 - ➔ 2. **不请自来**：来的是“不速之客”；
 - ➔ 3. **望眼欲穿**：直到结束，也没有到来所“期待”的。
- 这三种情况**对应**到栈的操作即为：
 - ➔ 1. 和栈顶的左括弧不相匹配；
 - ➔ 2. 栈中并没有左括弧等在哪里；
 - ➔ 3. 栈中还有左括弧没有等到和它相匹配的右括弧。



栈的应用举例

●例3-3 迷宫求解问题





栈的应用举例

- 计算机解迷宫时，通常用的是“**穷举求解**”的方法，即从入口出发，顺某一方向向前探索，若**能走通**，则**继续往前走**；**否则沿原路退回**，**换一个方向**再继续探索，直至**所有可能**的通路都探索到为止，如果所有可能的通路都试探过，还是不能走到终点，那就说明该迷宫不存在从起点到终点的通道。
- 演示一 3-3-1
- 演示二 3-3-2



栈的应用举例

- 求迷宫中一条路径的算法的**基本思想**是：
 - ➔ 若当前位置“**可通**”，则**纳入**“当前路径”，并**继续**朝“下一位置”探索；
 - ➔ 若当前位置“**不可通**”，则应顺着“来的方向”**退回**到“**前一通道块**”，然后朝着除“**来向**”之外的其他方向继续探索；
 - ➔ 若该通道块的四周四个方块均“**不可通**”，则应从“当前路径”上**删除**该通道块。
 - ➔ 假设以**栈S**记录“当前路径”，则**栈顶**中存放的是“当前路径上最后一个通道块”。 “纳入路径”的操作即为“当前位置入栈”； “从当前路径上删除前一通道块”的操作即为“出栈”。



栈的应用举例

- 求迷宫中一条从入口到出口的路径的伪码算法如下：

- 设定当前位置的初值为入口位置；

```
do{  
    若当前位置可通，  
        则{ 将当前位置插入栈顶；                // 纳入路径  
            若该位置是出口位置，则算法结束； // 此时栈中存放的是一条从入口位置到出口位置的路径  
            否则切换当前位置的可通方块为新的当前位置；  
        } 否则  
        {  
            若栈不空且栈顶位置尚有其他方向未被探索，  
                则设定新的当前位置为: 沿顺时针方向旋转找到的栈顶位置的下一相邻块；  
            若栈不空但栈顶位置的四周均不可通，  
                则{ 删去栈顶位置；                // 从路径中删去该通道块  
                    若栈不空，则重新测试新的栈顶位置，  
                    直至找到一个可通的相邻块或出栈至栈空； }  
        }  
} while (栈不空);
```



栈的应用举例

●例3-4 表达式求值问题

- ➔ 任何一个表达式都是由操作数(operand)、运算符(operator)和界限符(delimiter)组成
 - 操作数可以是常数也可以是被说明为变量或常量的标识符;
 - 运算符可以分为算术运算符、关系运算符和逻辑运算符等三类;
 - 基本界限符有左右括弧和表达式结束符等。
- ➔ 为了叙述简洁, 在此仅限于讨论只含二元运算符的算术表达式。可将这种表达式定义为:
 - 表达式::= 操作数 运算符 操作数
 - 操作数::= 简单变量 | 表达式
 - 简单变量::= 标识符 | 无符号整数



栈的应用举例

- 由于算术运算的规则是：先乘除后加减、先左后右和先括弧内后括弧外，则对表达式进行运算不能按其中运算符出现的先后次序进行。

→ 在计算机中，对这种二元表达式可以有三种不同的标识方法。

假设 $\text{Exp} = \text{S1} + \text{OP} + \text{S2}$

则称 $\text{OP} + \text{S1} + \text{S2}$ 为表达式的前缀表示法（简称前缀式）

称 $\text{S1} + \text{OP} + \text{S2}$ 为表达式的中缀表示法（简称中缀式）

称 $\text{S1} + \text{S2} + \text{OP}$ 为表达式的后缀表示法（简称后缀式）

可见，它以运算符所在不同位置命名的。



栈的应用举例

● 例如：若 $\text{Exp} = \underline{a \times b} + \underline{(c - d / e) \times f}$ 则它的

→ 前缀式为： $+\underline{\times a b} \underline{\times -c / d e f}$

→ 中缀式为： $\underline{a \times b} + \underline{c - d / e \times f}$

→ 后缀式为： $\underline{a b \times c d e / - f \times} +$ 演示 3-3-4

● 综合比较它们之间的关系可得下列结论：

- 1. 三式中的 “操作数之间的相对次序相同”；
- 2. 三式中的 “运算符之间的的相对次序不同”；
- 3. 中缀式丢失了括弧信息，致使运算的次序不确定；
- 4. 前缀式的运算规则为：连续出现的两个操作数和在它们之前且紧靠它们的运算符构成一个最小表达式；
- 5. 后缀式的运算规则为：
 - 运算符在式中出现顺序恰为表达式的运算顺序；
 - 每个运算符和在它之前出现且紧靠它的两个操作数构成一个最小表达式。



栈的应用举例

- 以下就分“**如何按后缀式进行运算**”和“**如何将原表达式转换成后缀式**”两个问题进行讨论。

→ 如何按**后缀式**进行运算？

可以用两句话来归纳它的求值规则：“**先找运算符，后找操作数**”。

演示 3-3-5

- 从这个例子的运算过程可见，运算过程为：对后缀式**从左向右**“扫描”，遇见操作数则**暂时保存**，遇见运算符即可进行运算；此时参加运算的两个操作数应该是在它之前刚刚碰到的两个操作数，并且**先出现**的是**第一**操作数，**后出现**的是**第二**操作数。由此可见，在运算过程中保存操作数的结构应该是**栈**。

➤ 实例

演示 3-3-6



栈的应用举例

● 如何由原表达式转换成后缀式？

→ 先分析一下“原表达式”和“后缀式”两者中运算符出现的次序有什么不同。

➤ 例一 原表达式： $a \times b / c \times d - e + f$

后 缀 式： $ab \times c / d \times e - f +$

➤ 例二 原表达式： $a + b \times c - d / e \times f$

后 缀 式： $abc \times + de / f \times -$

- 例一原表达式中运算符出现的先后次序恰为运算的顺序，自然在后缀式中它们出现的次序和原表达式相同。
- 例二中运算符出现的先后次序不应该是它的运算顺序。按照算术运算规则，先出现的“加法”应在在它之后出现的“乘法”完成之后进行，而应该在后面出现的“减法”之前进行；同理，后面一个“乘法”应后于在它之前出现的“除法”进行，而先于在它之前的“减法”进行。



栈的应用举例

- 对原表达式中出现的每一个运算符是否即刻进行运算**取决于在它后面出现的运算符**，如果它的优先级“高或等于”后面的运算，则它的运算先进行，否则就得等待在它之后出现的所有优先级高于它的“运算”都完成之后再行进行。

→ “优先数”的概念。给每个运算符赋以一个优先数的值，如下所列：

运算符 **#** (+ - **X** / **

优先数 **-1** **0** **1** **1** **2** **2** **3**

- “**”为乘幂运算，“#”为结束符。容易看出，优先数反映了算术运算中的优先关系，即优先数“高”的运算符应优先于优先数低的运算符进行运算。
- 显然，保存运算符的结构应该是**栈**，**从栈底到栈顶的运算符的优先级是从低到高的**，因此，它们**运算的先后应是从栈顶到栈底的**。



栈的应用举例

- 因此，从原表达式求得后缀式的**规则**为：
 - ➔ 1) 设立**运算符栈**；
 - ➔ 2) 设表达式的结束符为“#”，预设运算符栈的栈底为“#”；
 - ➔ 3) 若当前字符是操作数，则直接发送给后缀式；
 - ➔ 4) 若当前字符为运算符且优先数大于栈顶运算符，则进栈，否则退出栈顶运算符发送给后缀式；
 - ➔ 5) 若当前字符是结束符，则自栈顶至栈底依次将栈中所有运算符发送给后缀式；
 - ➔ 6) “(”对它前后的运算符起隔离作用，若当前运算符为“(”时进栈；
 - ➔ 7) “)”可视为自相应左括弧开始的表达式的结束符，则从栈顶起，依次退出栈顶运算符发送给后缀式，直至栈顶字符为“(”止。
- ➔ 演示 3-3-7



栈的应用举例

```
void transform(char suffix[], char exp[] ) { // 从合法的表达式字符串 exp 求得其相应的后缀式
    //suffix

    InitStack(S); Push(S, # );    p = exp; ch = *p;
    while (!StackEmpty(S)) {
        if (!IN(ch, OP)) Pass( suffix, ch); // 直接发送给后缀式
    else {
        switch (ch) {
            case ( : Push(S, ch); break;
            case ) : {
                Pop(S, c);
                while (c!= ( )
                    { Pass( suffix, c); Pop(S, c) }
                break; }
            default : { while(Gettop(S, c) && ( precede(c,ch))) { Pass( suffix, c); Pop(S, c); }
                        if ( ch!= # ) Push( S, ch);    break; } // default
        } // switch    } // else
        if ( ch!= # ) { p++; ch = *p; }
    } // while    } // transform
```



栈的应用举例

● 例3-5 递归函数的实现

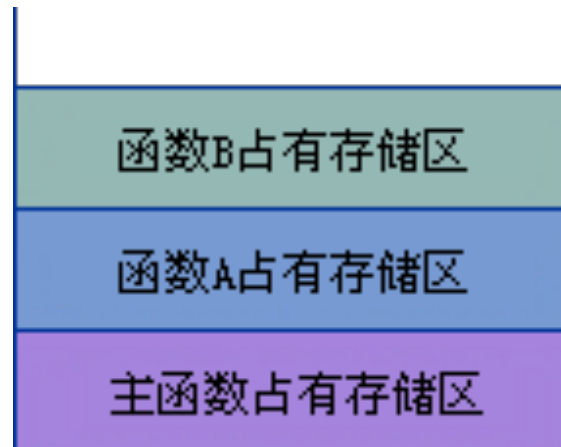
- 在程序设计中，经常会碰到多个函数的嵌套调用。和汇编程序设计中主程序和子程序之间的链接和信息交换相类似，在高级语言编制的程序中，调用函数和被调用函数之间的链接和信息交换也是由编译程序通过栈来实施的。

- ➔ 当一个函数在运行期间调用另一个函数时，在运行该被调用函数之前，需先完成三件事：
 - 1) 将所有的实在参数、返回地址等信息传递给被调用函数保存；
 - 2) 为被调用函数的局部变量分配存储区；
 - 3) 将控制转移到被调用函数的入口。
- ➔ 而从被调用函数返回调用函数之前，应该完成：
 - 1) 保存被调函数的计算结果；
 - 2) 释放被调函数的数据区；
 - 3) 依照被调函数保存的返回地址将控制转移到调用函数。



栈的应用举例

- 当多个函数嵌套调用时，由于函数的运行规则是：后调用先返回，因此各函数占有的存储管理应实行“**栈式管理**”。
- 假设主函数调用函数A，函数A又调用函数B，显然，在函数B运行期间主函数和函数A占用的存储都不能被覆盖，反之，当函数B运行结束，它所占用的存储区便可释放，同时因为程序控制转移到函数A，当前程序访问的数据自然就是函数A占用的存储区了。





栈的应用举例

- 一个递归函数的运行过程类似于多个函数的嵌套调用，**差别仅在于“调用函数和被调用函数是同一个函数”**。为了保证“每一层的递归调用”都是对“本层”的数据进行操作，在执行递归函数的过程中需要一个“**递归工作栈**”。

→ 它的**作用**是：

- 一、将递归调用时的实参和函数返回地址传递给下一层执行的递归函数；
- 二、保存本层的参数和局部变量，以便从下一层返回时重新使用它们。



栈的应用举例

- 递归过程执行过程中所占用的数据区，称之为**递归工作栈**。
 - 每一层的递归参数等数据合成一个记录，称之为**递归工作记录**。
 - 栈顶记录指示当前层的执行情况，称之为**当前活动记录**。
 - 递归工作栈的栈顶指针，称之为**当前环境指针**。
- 递归函数执行过程中递归工作栈的工作情况可用大家熟悉的“**梵塔函数**”为例。
 - 演示 3-3-9



栈的应用举例

● 算法3.3

void hanoi (**int** n, **char** x, **char** y, **char** z, **int** &i)

// 将塔座 **x** 上按直径由小到大且至上而下编号为**1**至 **n**

// 的 **n** 个圆盘按规则搬到塔座 **z** 上, **y** 可用作辅助塔座。

```
{
    if (n==1)
    {
        move(x, 1, z);      // 将编号为1的圆盘从 x 移到 z
        i++;
    }
    else {
        hanoi(n-1, x, z, y); // 将 x 上编号为1至 n-1 的圆盘移到 y,z 作辅助塔
        move(x, n, z);       // 将编号为 n 的圆盘从 x 移到 z
        i++;
        hanoi(n-1, y, x, z); // 将 y 上编号为1至 n-1 的圆盘移到 z,x 作辅助塔
    }
}
```



栈的应用举例

- 在此，称调用递归函数的**主函数**为“**第0层**”。从主函数调用递归函数被称为进入递归函数的“**第1层**”，从递归函数的“**第 i 层**”递归调用本函数被称为进入递归函数的“**第 $i+1$ 层**”。
- 显然，当递归函数**执行到第 i 层**时，从**第1层到第 $i-1$ 层**的数据都**必须被保存下来**，以便一层一层**退回**时继续使用。递归函数执行过程中每一层所占用的内存数据区合起来就是一个“**递归工作栈**”。
- algo3-10.cpp**



本章小结

- 在第三章（上）我们学习了**栈**这种抽象数据类型。在学习过程中大家已经了解到，栈都属**线性结构**，因此它的存储结构和线性表非常类似，同时由于它的基本操作要比线性表简单得多，因此它在相应的存储结构中实现的算法都比较简单，相信对大家来说都不是难点。
- 第三章（上）的**重点**则在于栈的**应用**。通过第三章（上）所举的例子学习分析应用问题的特点，在算法中适时应用栈。



本章知识点与重点

● 知识点

顺序栈、链栈

● 重点和难点

栈是在程序设计中被广泛使用的线性数据结构，因此本章的学习重点在于掌握这种结构的特点，以便能在应用问题中正确使用。