数据结构

北京邮电大学 信息安全中心
武 斌 杨 榆

上章内容

上一章（线性表）内容：

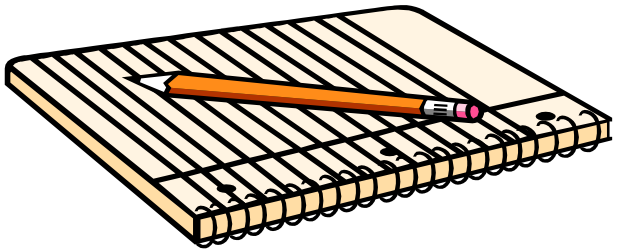
- 了解线性表的概念及其逻辑结构特性
- 理解顺序存储结构和链式存储结构的描述方法
- 掌握线性表的基本操作及算法实现
- 重点掌握线性链表的存储结构及算法实现
- 分析两种存储结构的时间和空间复杂度

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

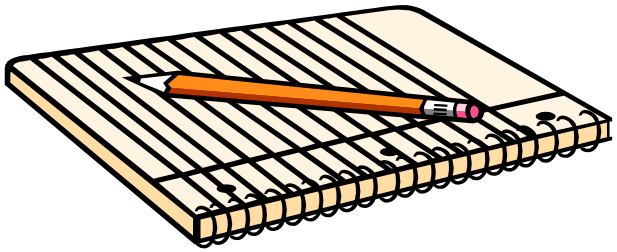
- 栈和队列是在程序设计中被广泛使用的两种线性数据结构。
- 从数据结构角度看，栈和队列是操作受限的线性表。
- 从数据类型角度看，栈和队列是与线性表不同的两类重要的抽象数据类型。
- 它们的特点在于基本操作的特殊性，栈必须按“后进先出”的规则进行操作，而队列必须按“先进先出”的规则进行操作。和线性表相比，它们的插入和删除操作受更多的约束和限定，故又称为限定性的线性表结构。

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：	Insert(L,i,x) (1≤i≤n+1)	Delete(L,i) (1≤i≤n)	线性表允许在表内任一位置进行插入和删除
栈：	Insert(L,n+1,x)	Delete(L,n)	栈只允许在表尾一端进行插入和删除
队列：	Insert(L,n+1,x)	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

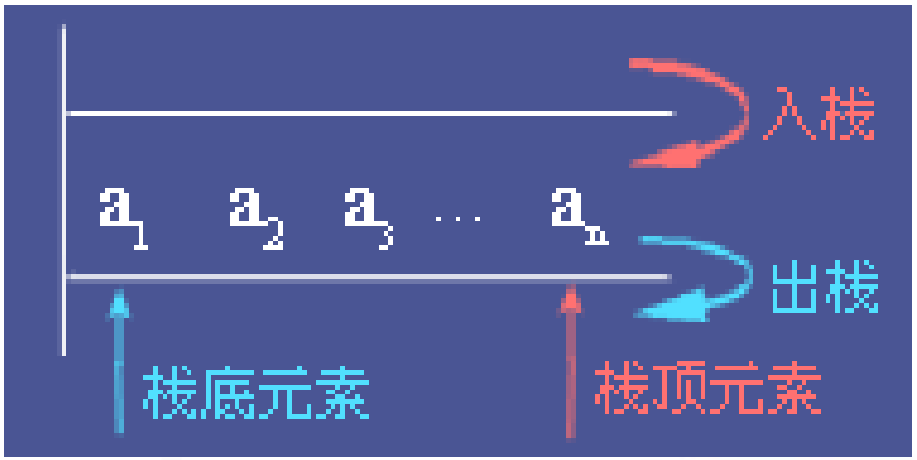
栈的类型定义



- 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

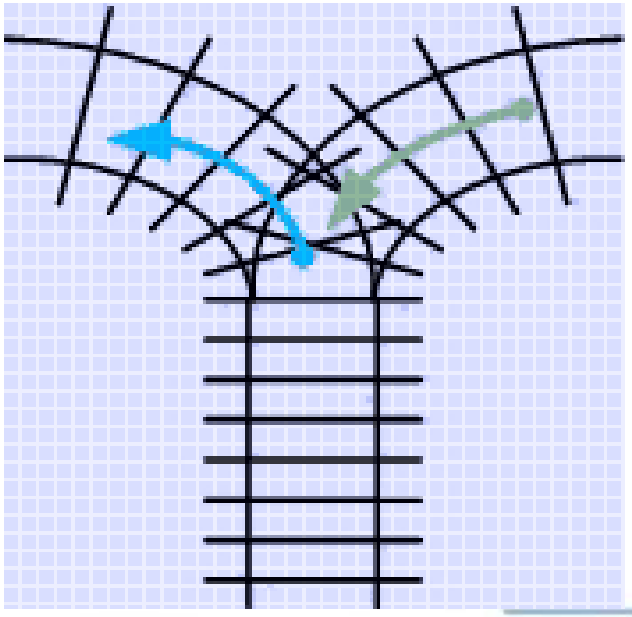
栈的类型定义

- **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，…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，…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 | a_i ∈ElemSet, i=1,2,...,n, n≥0 }

数据关系： R1＝{ < a_{i-1} , a_i >| a_{i-1} , a_i ∈D, i=2,...,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

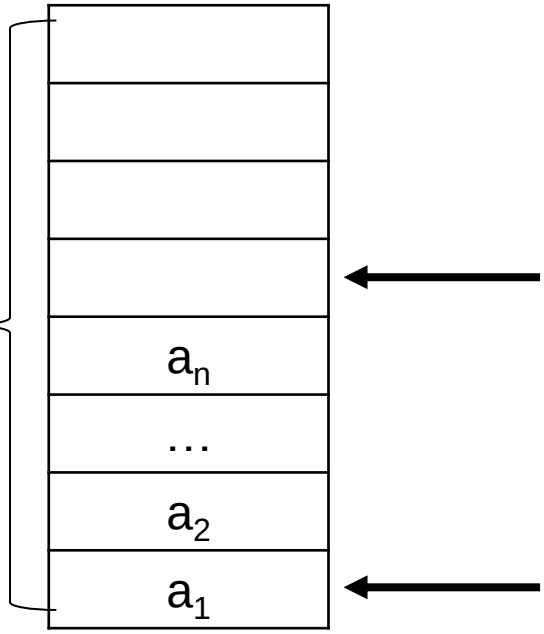
- 和线性表类似，栈也有顺序存储和链式存储两种存储表示，其顺序存储结构简称为顺序栈。

```
#define STACK_INIT_SIZE 100 //栈空间初始容量，以元素为单位
#define STACK_INCREMENT //栈空间扩容增量，以元素为单位
typedef struct {
    SElemType *base; // 栈底指针：存储空间基址
    SElemType *top;          // 栈顶指针
    int stacksize;    // 允许最大容量，以元素为单位
} SqStack;
```

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**。

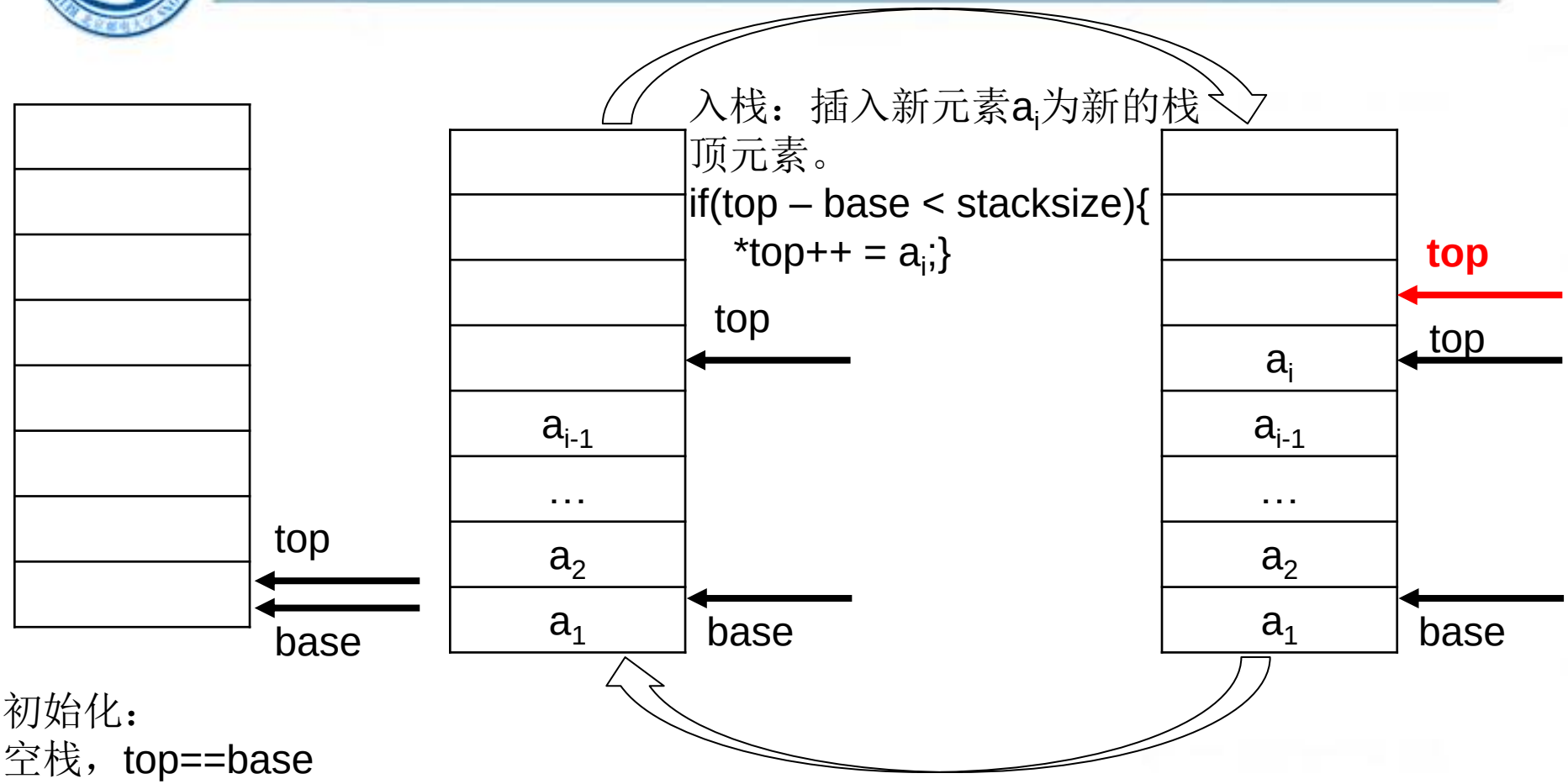
stacksize:
最大容量，即可容纳的最大元素个数。
初始化时，栈最多可容纳STACK_INIT_SIZE个元素。
栈满以后，每次扩容增加STACK_INCREMENT个元素。



栈顶指针：
top指向栈顶元素的下一个地址。因此，当前栈元素个数等于top-base。

栈底指针：
base为这个存储空间的基地址，也即一维数组的地址。

栈的存储表示和操作的实现



初始化：
空栈，top==base

出栈：删除栈顶a_i。
if(top > base){
e = *--top;}

栈的存储表示和操作的实现

➔ // 基本操作接口(函数声明)：

StatusInitStack (SqStack **&**S);

// 构造一个空栈S。

Status DestroyStack (SqStack **&**S);

// 销毁栈S，S 不再存在。栈存储空间全部释放

Status ClearStack (SqStack **&**S);

// 将 S 置为空栈。栈的存储空间仍然保留，原有元素被清空。

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)：

Status GetTop (SqStack S, ElemType &e);

// 若栈不空，则用 e 返回S的栈顶元素，并返回OK；

// 否则返回 错误码。（和**Pop**操作不同，栈顶指针不变）

Status Push (SqStack **&**S, ElemType e);

// 插入元素 e 为新的栈顶元素，并返回 OK；若失败返回错误码。

Status 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OK；

// 否则返回错误码。

Status 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：

```
Status InitStack (SqStack &S)
{
    S.base=(SElemType*)malloc(STACK_INIT_SIZE * sizeof(SElemType));
    if(S.base == NULL)
      exit(OVERFLOW);          // 存储分配失败
    S.stacksize = STACK_INIT_SIZE;
    S.top = S.base;            // 空栈
    return OK;
}
```

栈的存储表示和操作的实现

➔ **Status** GetTop (SqStack S, ElemType **&**e)（课堂）

```
{
//   若栈不空，则用 e 返回S的栈顶元素，并返回OK；
//   否则返回 错误码。（和Pop操作不同，栈顶指针不变）
     if(S.base == S.top)//空栈
        return ERROR;
     e = *(S.top - 1);//仅返回值，不删除栈顶元素；栈顶指针不变
                      //栈顶指针指向栈顶元素下一个位置
     return OK;
 }
```

栈的存储表示和操作的实现

➔ **Status** Push (SqStack **&**S, ElemType e)　（随堂）

```
{
//   插入元素 e 为新的栈顶元素，并返回 OK；若失败返回错误码。
     if(S.top - S.base >= S.stacksize) {//栈满了
        S.base = (SElemType *)realloc(S.base,
                    (S.stacksize + STACK_INCREMENT)*sizeof(SElemType));
        if(!S.base)  exit(OVERFLOW);
        S.top = S.base + S.stacksize; //基地址可能变化，修正栈顶指针
        S.stacksize+=STACK_INCREMENT; //更新栈最大容量
      } //满栈，处理完毕
      *(S.top++)=e; //等价于：*S.top=e; S.top++
      return OK;
}
```

栈的存储表示和操作的实现

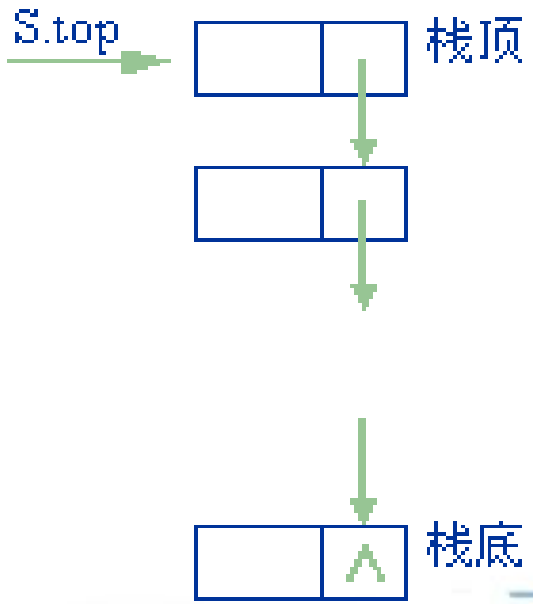
➔ **Status** Pop (SqStack **&**S, ElemType **&**e)

```
{ //   若栈不空，则删除S的栈顶元素，用e返回其值，并返回OK；
  //   否则返回错误码。
      if(S.base == S.top)//空栈，无元素可删除
        return ERROR;
      e = *(--S.top); //等价于：S.top--; e=*S.top;
      return OK;
}
```

- 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)**。
- 思考：请写出栈遍历操作。

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 _Node{
    ElemType data;      // 节点数据域
    struct _Node *next;     // 节点指针域，指向前驱元素节点
} Node, *Link;
typedef struct {
    Link top;       // 栈顶指针
    int length;         // 栈中元素个数
} LnkStack;
```

栈的存储表示和操作的实现

- **Status** InitStack (LnkStack **&**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

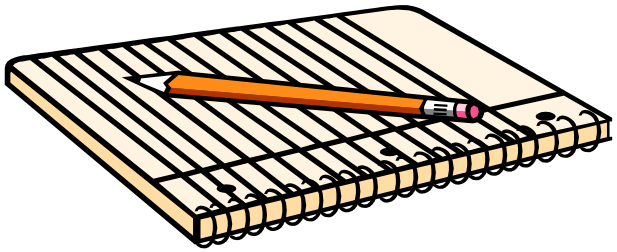
➔ **Status** Push (Stack **&**S, ElemType e)

```
{// 在栈顶之上插入元素 e 为新的栈顶元素
    p = new Node;       // 建新的结点
    if(!p) exit(OVERFLOW);          // 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

```
Status Pop ( LnkStack &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 OK；否则返回 错误码
    if ( !S.top ) return ERROR; // 空栈，没有栈顶元素可删除
    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 OK;
    }
} // Pop
```

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:

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

    (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.swf

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。

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 (“%d”,&N);            // 输入一个十进制数
    while(N)
    {
        Push(S,N % 8);   // “余数”入栈
        N = N/8;             // 整除，非零“商”继续运算
    } // while
    while (!StackEmpty(S))
    {    // 按“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

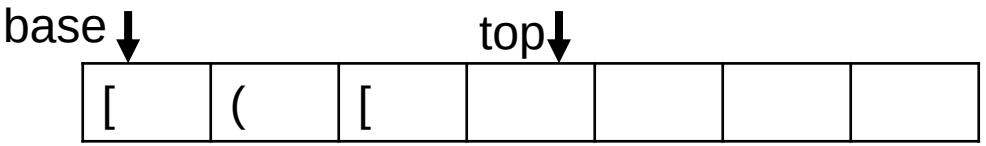
- **例3-2** 假设表达式中允许包含两种括号：圆括号和方括号，其嵌套的顺序随意，如（［ ］（））或［（［ ］［ ］）］等为正确的匹配，［（ ］）或（［ ］（ ）或（（ ））均为错误的匹配。
- 检验括号是否匹配的方法可用“**期待的急迫程度**”这个概念来描述。即后出现的“左括弧”，它等待与其匹配的“右括弧”出现的“急迫”心情要比先出现的左括弧高。换句话说，对“左括弧”来说，后出现的比先出现的**“优先”等待检验**，对“右括弧”来说，每个出现的右括弧要去找在它之前**“最后”出现**的那个左括弧去**匹配**。
- 显然，必须将先后出现的左括弧**依次保存**，为了反映这个优先程度，保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

- 例3-2以表达式“[（[ ] [ ]）]”和“[（]）”分析算法执行过程中栈的变化。

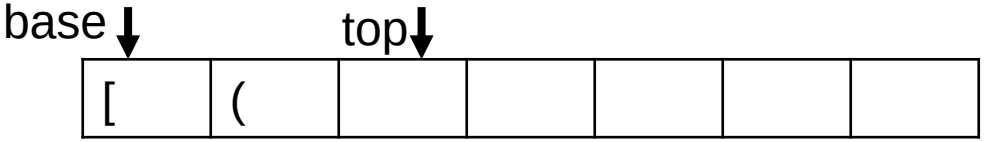
1	2	3	4	5	6	7	8
[	(	[	]	[	]	)	]

1.分析表达式字符1,2,3，它们都是左括号，是否能够得到匹配，现在还未知，因此应该暂存、结合后续出现的字符来判断。“待处理”的顺序，考虑到越后出现的括号，越需要优先匹配，所以按照栈的顺序暂存这些待判决的括号。

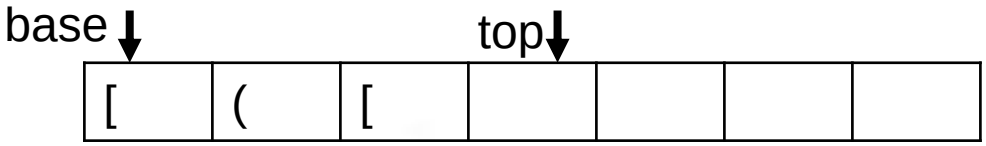


1	2	3	4	5	6	7	8
[	(	[	]	[	]	)	]

2.分析表达式字符4，它是右括号，理应与最近一次出现的左括号配对，否则表达式错误。而最近一次出现的左括号为栈顶元素。因此，“弹出”栈顶元素，检查其是否与字符4配对。本例中，左右方括号配对。可以继续分析表达式。



1	2	3	4	5	6	7	8
[	(	[	]	[	]	)	]



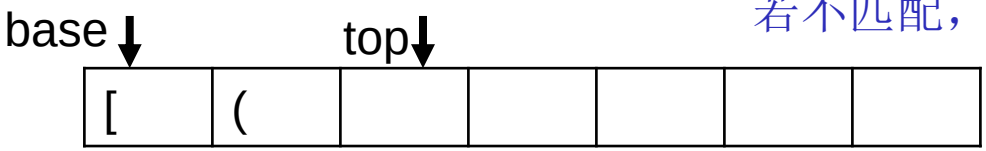
1	2	3	4	5	6	7	8
[	(	[	]	[	]	)	]

栈的应用举例

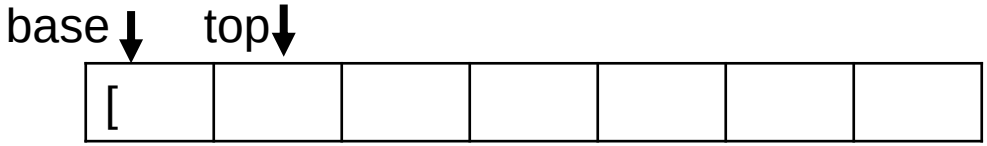
- 例3-2以表达式“[（[ ] [ ]）]”和“[（]）”分析算法执行过程中栈的变化。

1	2	3	4	5	6	7	8
[	(	[	]	[	]	)	]

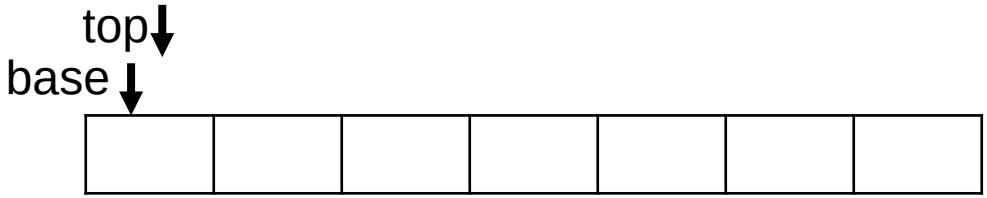
3.同样的方法继续分析后续字符。若当前字符是左括号，则压入栈；若是右括号，则弹出栈顶元素，检查两者是否匹配。
若匹配，继续检查下一字符，直到表达式结束。
若不匹配，表达式不合法，返回这一判决结果。



1	2	3	4	5	6	7	8
[	(	[	]	[	]	)	]



1	2	3	4	5	6	7	8
[	(	[	]	[	]	)	]



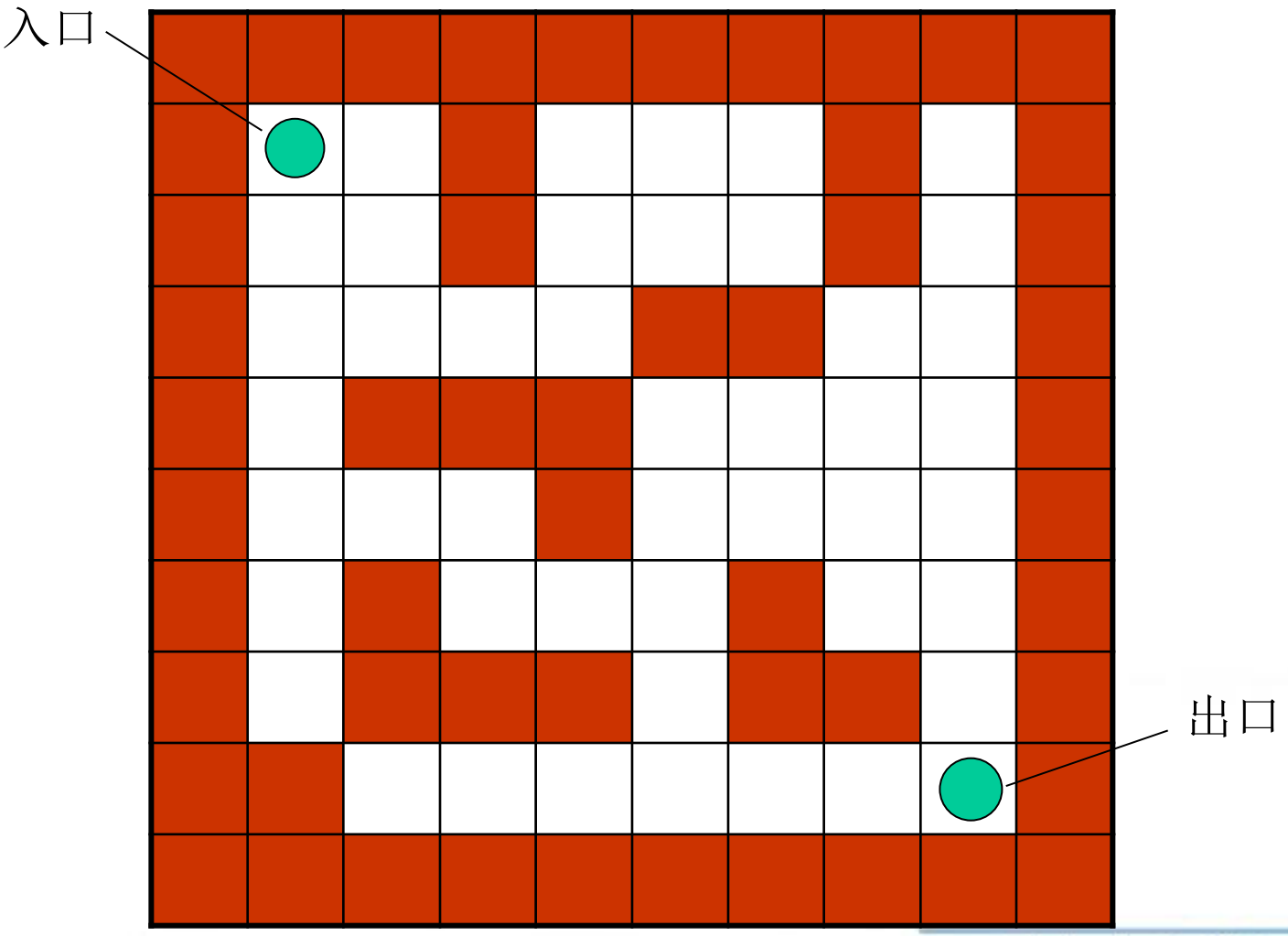
1	2	3	4	5	6	7	8
[	(	[	]	[	]	)	]

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．来的右括弧非是所“期待”的；
  - 2．来的是“不速之客”；
  - 3．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；例如：[()[)….
  - 2．栈中并没有左括弧等在哪里；例如：[()])….
  - 3．栈中还有左括弧没有等到和它相匹配的右括弧。例如：[()[([])]….

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.swf（迷宫存在从起点到终点的通道）
- 演示二 3-3-2.swf（迷宫不存在从起点到终点的通道）

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东邻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

```
bool MazePath(MazeType name, PosType start, PosType end){
 //若在迷宫name中存在从start到end的路径，保存在栈中，并返回TRUE；
 InitStack(S);   curpos = start;   curstep = 1;
 do{
  if(Pass(curpos)){//当前位置可行，不是砖、没走过、不是不可行块
    FootPrint(curpos);  //留下足迹，标记为“经过块”
    e.seat = curpos; e.ord = curstep; e.di = 1;//{1,2,3,4}->{east,south,west,north}
    Push(e);
    if(seat == end){return TRUE;}
    curpos = NextPos(curpos,1); curstep++;  //得到下一个待检测的位置，从东方开始
  }else{//当前位置不可行
     if(!StackEmpty(S)){
          Pop(S, e);
          while(e.di == 4 && !StackEmpty(S)){
             MarkPrint(e.seat); Pop(S, e);/*标记为不可行点*/}
          if(e.di < 4){e.di++; Push(S,e); curpos = NextPos(e.seat, e.di);} //下一个待检测的位置
    }//end of 栈不空
  }//end of 当前位置不可行
 }while(!StackEmpty(S));  return FALSE;
}
```

栈的应用举例

通过实例分析迷宫算法堆栈：

	0	1	2	3	4
0	#	#	#	#	#
1	#	E			#
2	#	#		#	#
3	#	S			#
4	#	#	#	#	#

1.(3,4)是不可行位置，后退，尝试下一个位置。

{3,(3,3),1}
{2,(3,2),1}
{1,(3,1),1}

2.(4,3)是不可行位置，后退，尝试下一个位置。

{3,(3,3),2}
{2,(3,2),1}
{1,(3,1),1}

3.(3,2)已走过，(2,3)，是不可行位置。(3,3)已无可尝试位置，应标识为不能通行。后退，尝试下一个位置。

{3,(3,3),4}
{2,(3,2),1}
{1,(3,1),1}

	0	1	2	3	4
0	#	#	#	#	#
1	#	E			#
2	#	#		#	#
3	#	S			#
4	#	#	#	#	#

4.同样步骤测试可行路径，下一位置为(1,4)。

{5,(1,3),1}
{4,(1,2),1}
{3,(2,2),4}
{2,(3,2),4}
{1,(3,1),1}

5.(1,3) 不能通行，标记。下一步，检测(1,2)的南方邻块。

{4,(1,2),2}
{3,(2,2),4}
{2,(3,2),4}
{1,(3,1),1}

3.(2,2)已走过。后退，检测(1,2)的西方邻块。(1,1)是终点，路径搜索成功。

{5,(1,1),1}
{4,(1,2),3}
{3,(2,2),4}
{2,(3,2),4}
{1,(3,1),1}

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 Exp = S1 + **OP** + S2

  则称 **OP** + S1 + S2 为表达式的**前缀表示法**（简称**前缀式**）

  称 S1 + **OP** + S2 为表达式的**中缀表示法**（简称**中缀式**）

  称 S1 + S2 + **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Exp = a×b+(c−d / e)×f 则它的
  - 前缀式为： +×a b ×−c / d e f
  - 中缀式为：a×b +c−d / e×f
  - 后缀式为：a b×c d e / −f ×+ 后缀式的形成见演示（3-3-4.swf）
- 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：
    - 运算符在式中出现的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.swf
    - ➢ 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，后缀式“3,5,*,6,8,4,/,-,7,*,+,#”的计算

      演示3-3-6.swf

栈的应用举例

- 练习“如何按后缀式进行运算”
  - ➔ 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”。

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。

已知后缀式为：3,4,×,2, / ,8,4, / ,5, -, 3, ×, +，请计算

解：

1.操作数3,4入栈，遇到操作符×后，弹出栈，计算3×4，结果再入栈。

2.此时栈内操作数为3×4，操作数2入栈，遇到操作符 / 后，弹出栈，计算3×4 / 2，结果再入栈。

3.依次类推，结果为-3

4.原表达式为： 3×4 / 2+(8 / 4-5) ×3

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×b / c×d－e＋f
      后 缀 式：ab×c / d×e－f＋
    - 例二　原表达式：a＋b×c－d / e×f
      后 缀 式：abc×＋de / f×－
  - 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.swf

栈的应用举例

$a \times b + (c - d / e) \times f$

```
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);   // 若当前字符是操作数，直接发送给后缀式
    else {
       switch (ch) {
          case '(' : Push(S, ch); break;
          case ')': Pop(S, c);//发送'('和')'间操作符到后缀式
             while (c!= '(' ) { Pass( suffix, c); Pop(S, c) ;}      break;
          default : while(Gettop(S, c) && ( precede(c,ch)))//持续传递栈顶元素，直到优先级低于当前操作符
                    { Pass( suffix, c); Pop(S, c); }
                     if ( ch!= '# ') Push( S, ch);    break;   } // 把当前操作符压入栈
       } /*end of switch*/   } /* end of else*/
    if ( ch!='# ') { p++; ch = *p; }
} /* end of while*/   } /* end of transform */
```

While循环中，使用Getop获取栈顶操作符c（此时c仍在栈中）。若c的优先级大于或等于当前操作符ch，则对应操作应该优先执行，所以用Pop弹出该栈顶元素c，否则并不弹出。因此，循环结束时，栈顶元素c的优先级小于ch，需要将ch压入栈。

栈的应用举例

通过实例分析后缀式转换算法堆栈：

原表达式	a×b+(c−d / e)×f
后缀式	ab
栈	#×

表达式当前字符ch为操作符"+"，操作符栈栈顶c为"×"，应该先弹出栈顶元素，发往后缀式。

原表达式	a×b+(c−d / e)×f
后缀式	ab×cde
栈	#+(- /

表达式当前字符ch为界限符")"，应该先弹出栈内"("后所有元素，发往后缀式。

原表达式	a×b+(c−d / e)×f
后缀式	ab×cde / -f
栈	#+×

表达式当前字符ch为界限符"#"，应该先弹出栈内"#"后所有元素，发往后缀式。

原表达式	a×b+(c−d / e)×f
后缀式	ab×cde / -f×+
栈	

表达式所有字符处理完毕，后缀式转换操作完成。

栈的应用举例

后缀式转换算法练习

原表达式	a×b / c×d－e＋f
后缀式	ab×c / d×e－f＋

原表达式	a＋b×c－d / e×f
后缀式	abc×＋de / f×－

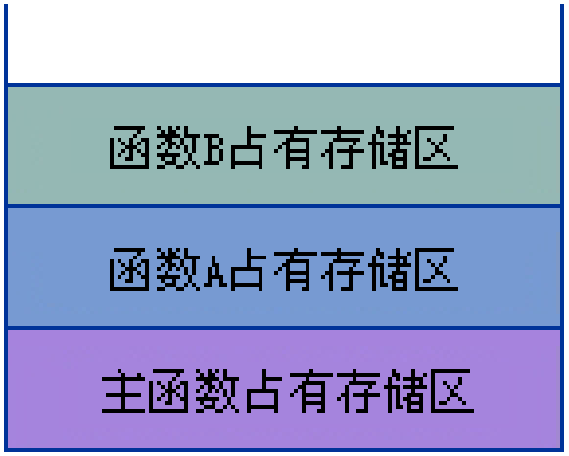
原表达式	a＋b / c×(d－e＋f×g)
后缀式	abc / de－fg×＋×＋

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“阶乘计算”为例。

栈的应用举例

- 阶乘的递归实现。已知阶乘递归计算公式如下：

- $f(n)=\begin{cases}1, & n=0\\ n\cdot f(n-1), & n>0\end{cases}$

```
1 int Fact(int n){
2   if(n==0)//0的阶乘为1
3     return 1;
4   else //n的阶乘等于n乘以(n-1)的阶乘
5     return n*Fact(n-1);
6 }
```

栈的应用举例

- 分析阶乘递归实现的执行，以n等于3为例：

递归运行层次	递归工作栈状态 返回地址，n值	执行语句
		Fact(3);
1	0,3	return 3*Fact(2);
2	5,2	return 2*Fact(1);
3	5,1	return 1*Fact(0);
4	5,0	return 1;

```
1 int Fact(int n){
2   if(n==0)//0的阶乘为1
3     return 1;
4   else //n的阶乘等于n乘以(n-1)的阶乘
5     return n*Fact(n-1);
6 }
```

栈的应用举例

- 分析阶乘递归实现的执行，以n等于3为例：

递归运行层次	递归工作栈状态 返回地址，n值	执行语句
		Fact(3);
1	0,3	return 3*Fact(2);
2	5,2	return 2*Fact(1);
3	5,1	return 1*Fact(0);
4	**5,0**	**return 1;**

函数执行完成，返回值为1;
而后，表达式1*Fact(0)求解为1*1,并返回上一级。后续调用执行流程相同。

```
1 int Fact(int n){
2   if(n==0)//0的阶乘为1
3     return 1;
4   else //n的阶乘等于n乘以(n-1)的阶乘
5     return n*Fact(n-1);
6 }
```

栈的应用举例

- n阶Hanoi塔问题。对于3个分别命名为X,Y,和Z的塔座，在塔座上插有n个直接大小各不相同、依小到大编号为1,2，…，n的圆盘。现要求将塔座X上的n个圆盘移至塔座Z，且保持原顺序不变。移动时必须遵循下列规则：
- 每次只能移动一个圆盘；
- 圆盘可以插在X、Y和Z中的任一塔座上；
- 任何时刻都不能将较大的圆盘压在较小的圆盘之上；
- 演示3-3-9.swf

栈的应用举例

问题的分析： 要求：将n个汉诺塔从塔座X，经由塔座Y辅助，移动到塔座Z， 求解：n阶汉诺塔问题的求解可以视为：	
步骤	伪码
若n为1，直接将塔盘从塔座X移动到塔座Z	move(x, 1, z)
按汉诺塔的移动规则，将n-1个汉诺塔盘，从塔座X，经由塔座Z辅助，移动到塔座Y，	hanoi(n-1, x, z, y)
将塔盘n，从塔座X，移到塔座Z，	move(x, n, z)
按汉诺塔的移动规则，将n-1个汉诺塔盘，从塔座Y，经由塔座X辅助，移动到塔座Z	hanoi(n-1, y, x, z)

以上分析中，n阶问题规模下降为n-1阶问题。n-1阶汉诺塔问题可用类似步骤解决。也就是说，n-1阶汉诺塔问题转换为求解n-2阶问题。依次类推，2阶汉诺塔问题转化为1阶问题。1阶问题的解决方法是明确的。而一旦低阶问题解决了，高阶问题也就解决了。

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

用3阶汉诺塔问题求解过程中的函数栈分析递归

本层参数				嵌套调用/主要步骤	返回后执行
n	源	辅	目		
				hanoi(3,a,b,c)	
3	a	b	c	hanoi(2,a,c,b)	move(a,3,c); hanoi(2,b,a,c)
2	a	c	b	hanoi(1,a,b,c)	move(a,2,b); hanoi(1,c,a,b)
1	a	b	c	move(a,1,c)	
输出结果				1.将塔盘1从塔座a移动到c 2.将塔盘2从塔座a移动到b	

至此，问题规模降至1，将塔盘1从塔座a移动到c。然而，函数调用栈未空，应逐层返回，执行下一条语句。这里，从调用hanoi(1,a,b,c)返回后，执行move(a,2,b)，又发起了嵌套调用hanoi(1,c,a,b)。

栈的应用举例

用3阶汉诺塔问题求解过程中的函数栈分析递归

本层参数				嵌套调用/主要步骤	返回后执行
n	源	辅	目		
				hanoi(3,a,b,c)	
3	a	b	c	hanoi(2,a,c,b)	move(a,3,c); hanoi(2,b,a,c)
2	a	c	b	hanoi(1,c,a,b)	返回上层
1	c	a	b	move(c,1,b)	
输出结果				1.将塔盘1从塔座a移动到c 2.将塔盘2从塔座a移动到b 3.将塔盘1从塔座c移动到b 4.将塔盘3从塔座a移动到c	

问题规模再次降至1，将塔盘1从塔座c移动到b。从调用hanoi(1,c,a,b)返回后，hanoi(2,a,c,b)执行完毕，返回到上一级，执行move(a,3,c)，又发起了嵌套调用hanoi(2,b,a,c)。

栈的应用举例

用3阶汉诺塔问题求解过程中的函数栈分析递归

本层参数				嵌套调用/主要步骤	返回后执行
n	源	辅	目		
				hanoi(3,a,b,c)	
3	a	b	c	hanoi(2,b,a,c)	返回上层
2	b	a	c	hanoi(1,b,c,a)	move(b,2,c); hanoi(1,a,b,c)
1	b	c	a	move(b,1,a)	
输出结果				1.将塔盘1从塔座a移动到c;2.将塔盘2从塔座a移动到b 3.将塔盘1从塔座c移动到b;4.将塔盘3从塔座a移动到c 5.将塔盘1从塔座b移动到a;6.将塔盘2从塔座b移动到c	

问题规模再次降至1，将塔盘1从塔座b移动到a。从调用hanoi(1,b,c,a)返回后，执行move(b,2,c)，又发起了嵌套调用hanoi(1,a,b,c)。

栈的应用举例

用3阶汉诺塔问题求解过程中的函数栈分析递归

本层参数				嵌套调用/主要步骤	返回后执行
n	源	辅	目		
				hanoi(3,a,b,c)	
3	a	b	c	hanoi(2,b,a,c)	返回上层
2	b	a	c	hanoi(1,a,b,c)	返回上层
1	a	b	c	move(a,1,c)	
输出结果				1.将塔盘1从塔座a移动到c;2.将塔盘2从塔座a移动到b 3.将塔盘1从塔座c移动到b;4.将塔盘3从塔座a移动到c 5.将塔盘1从塔座b移动到a;6.将塔盘2从塔座b移动到c 7.将塔盘1从塔座a移动到c;	

问题规模再次降至1，将塔盘1从塔座a移动到c。逐层返回，函数调用栈空，问题得解。

栈的应用举例

main:
hanoi(3,a,b,c)
按规则将3个盘从柱a经由柱b移到柱c

hanoi(3,a,b,c):
1.hanoi(2,a,c,b)
2.a->c
3.hanoi(2,b,a,c)

hanoi(2,a,c,b):
1.hanoi(1,a,b,c)
2.a->b
3.hanoi(1,c,a,b)

hanoi(1,a,b,c):
a->c

hanoi(1,c,a,b):
c->b

hanoi(2,b,a,c):
1.hanoi(1,b,c,a)
2.b->c
3.hanoi(1,a,b,c)

hanoi(1,b,c,a):
b->a

hanoi(1,a,b,c):
a->c

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。