第3章　栈 和 队 列

3.1　栈

3.1.1　栈的概念

1. 栈的定义与基本运算

栈（Stack）是定义为只允许在末端进行插入和删除的线性表。允许插入和删除的一端称为栈顶（top），而不允许插入和删除的另一端称为栈底（bottom）。当栈中没有任何元素时则称为空栈。

栈只能通过在它的栈顶进行访问，故栈称为后进先出（Last In First Out，LIFO）或先进后出（First In Last Out，FILO）的线性表。

栈的基本运算包括：

- 栈初始化 void initStack(Stack& S)：创建一个空栈 S 并使之初始化。
- 判栈空否 int StackEmpty(Stack& S)：当栈 S 为空时函数返回 1，否则返回 0。
- 进栈 int Push(Stack& S, type x)：当栈 S 未满时，函数将元素 x 加入并使之成为新的栈顶，若操作成功，则函数返回 1；否则函数返回 0。
- 出栈 int Pop(Stack& S, type& x)：当栈 S 非空时，函数将栈顶元素从栈中退出并通过引用参数 x 返回退出元素的值。若操作成功，则函数返回 1；否则函数返回 0。
- 读栈顶元素 int getTop (Stack& S, type& x)：当栈 S 非空时，函数将通过引用参数 x 返回栈顶元素的值（但不退出）。若操作成功，则函数返回 1；否则函数返回 0。

2. 使用栈的实例

3-1　简述以下算法的功能（栈的元素类型为 int）。

算法一：

```
void algo1(SeqStack& S) {
     int i, d, n = 0, A[255];
     while(!stackEmpty(S)) { Pop(S, d);  A[n++] = d; }
     for(i = 0; i < n; i++) Push(S, A[i]);
}
```

算法二：

```
int algo2(SeqStack& S, int e) {
     SeqStack T;  initStack(T);
     int d;
     while(!stackEmpty(S)) {
          Pop(S, d);
          if(d != e) Push(T, d);
     }
```

```
        while(!stackEmpty(T))
             { Pop(T, d);  Push(S, d); }
        return stackSize(S);
    }
```

【解答】 注意栈的先进后出的特性。

算法一　将栈 S 中数据元素次序颠倒。

算法二　利用栈 T 辅助过滤掉栈 S 中所有值为 e 的数据元素，函数返回栈内元素个数。

3-2　设一个栈的输入序列为 1, 2,…, n，编写一个算法，判断一个序列 $p_1, p_2,\cdots, p_n$ 是否是一个合理的栈输出序列。

【解答】 首先举例说明。设一个输入序列为 1、2、3，预期输出序列为 2、3、1，可能的进栈/出栈动作如表 3-1 所示：如果预期输出序列为 3、1、2，这是一个不合理的输出序列，则可能的进栈/出栈动作如表 3-2 所示。从表中可见当预期的输出序列的数据比当前栈顶的元素小时，一定出现了输出不合理的情形，可以报错并结束检查处理，如表 3-2 所示。

表 3-1　一种进栈方式

输入序列当前数据	1	2		3			
栈顶数据	栈空	1	2	1	3	1	栈空
预期输出序列		2	2	3	3	1	
栈顶与预期输出序列比较		1 < 2	2 = 2	1 < 3	3 = 3	1 = 1	
动作	1 进栈	2 进栈	2 出栈	3 进栈	3 出栈	1 出栈	结束

表 3-2　另一种进栈方式

输入序列当前数据	1	2	3				
栈顶数据	栈空	1	2	3	2		
预期输出序列		3	3	3	1		
栈顶与预期输出数据比较		1 < 3	2 < 3	3 = 3	2 > 1		
动作	1 进栈	2 进栈	3 进栈	3 出栈	报错	结束	

算法中用 i = 1, 2,…, n 作为栈的输入数据序列，用 p[0], p[1],…, p[n-1]作为预期的栈的输出序列。算法的实现如下。

```
void Decision(int p[], int n) {
//算法调用方式 Decision(p, n)。输入: 存放预期出栈序列的数组 p, 序列的元素个
//数 n; 输出: 算法判断序列 p[0], p[1], .., p[n-1]是否合理的出栈序列
     SeqStack S;  InitStack(S);
     int i = 0, k = 0, d;  bool succ = true;
     do {
          if(stackEmpty(S)) Push(S, ++i);
          else {
               getTop(S, d);
               if(d < p[k]) Push(S, ++i);
               else if(d == p[k]) { Pop(S, d);  k++; }
               else { succ = false;  break; }
```

```
            }
        } while(k < n);
        for(int j = 0; j < n; j++) printf("%d ", p[j]);
        if(succ) printf(" 是合理的出栈序列! \n");
        else printf(" 是不合理的出栈序列! \n");
    }
```

设序列有 n 个元素，算法的时间复杂度为 O(n)，空间复杂度为 O(n)。

3.1.2 顺序栈

1. 顺序栈的概念

顺序栈是栈的顺序存储表示，是指用一组地址连续的存储单元依次存储栈中元素，同时附设指针 top 指向栈顶元素。用 C 语言描述，就是用一个一维数组来存储栈中的元素。

2. 顺序栈的存储表示

（1）静态存储分配，它的存储数组采用静态方式定义。

```
#define maxSize 100
typedef char SElemType;
typedef struct {
    SElemType elem[maxSize];
    int top;
} SeqStack;
```

顺序栈的静态存储结构预先定义或申请栈的存储空间，一旦装满不能扩充。因此在顺序栈中，当一个元素进栈之前，需要判断是否栈满，若栈满，则元素进栈会发生上溢现象。

（2）动态存储分配，它的存储数组在使用时动态存储分配，其好处是一旦栈满可以扩充。顺序栈的动态存储结构定义如下（保存于头文件 SeqStack.h 中）。

```
#define initSize 100                //栈的初始最大容量
typedef char SElemType;
typedef struct {                    //顺序栈的结构定义
     SElemType *elem;               //存储数组指针
     int maxSize, top;              //栈的最大容量和栈顶指针
} SeqStack;
```

在此头文件中还有一个对动态存储分配的顺序栈做初始化的算法，它的实现如下。

```
void initStack(SeqStack& S) {
//算法调用方式 initStack(S)。输入: 已声明的顺序栈 S; 输出: 算法创建一个最大
//尺寸为 initSize 的空栈, 若分配不成功则错误处理
     S.elem =(SElemType*) malloc(initSize*sizeof(SElemType));//创建栈空间
     if(S.elem == NULL) { printf("存储分配失败! \n");  exit(1); }
     S.maxSize = initSize;  S.top = -1;
}
```

算法的时间复杂度为 O(1)，空间复杂度为 O(1)。

栈的结构定义中给出的是一个指向栈的存储空间的指针，但这个存储空间不是自动分

配的，而是要通过初始化函数动态分配的。这样处理还有一个好处，一旦栈存储空间被耗尽，还可以扩充。我们可以申请一个新的更大的连续的存储空间取代原来的存储空间，把原来存储空间内存放的所有栈元素转移到新的存储空间后释放原来的存储空间。

3. 顺序栈基本运算的实现

以下 7 个算法都存放在程序文件 SeqStack.cpp 内，可直接链接使用。

3-3　设计一个算法，将新元素 x 进入顺序栈 S。

【解答】 算法的主要步骤是：首先判断栈是否已满，若栈已满，新元素 x 进栈将发生栈溢出；若栈不满，先让栈顶指针进 1，指到当前可加入新元素的位置，再按栈顶指针所指位置将新元素 x 插入。这个新插入的元素将成为新的栈顶元素。算法的实现如下。

```
bool Push(SeqStack& S, SElemType x) {
//进栈算法调用方式 bool succ = Push(S, x)。输入：已初始化的顺序栈 S，进栈元素
//值 x；输出：x 插入栈顶后的栈 S。若原栈不满，则进栈成功，函数返回 true
//否则函数返回 false
     if(stackFull(S)) {printf("栈满！\n");  return false;}    //栈满
     S.elem[++S.top] = x;                          //栈顶指针先加 1，再进栈
     return true;
}
```

算法的时间复杂度为 O(1)，空间复杂度为 O(1)。

3-4　设计一个算法，从顺序栈 S 退出栈顶的元素。

【解答】 算法的处理步骤是：先判断是否栈空。若在退栈时发现栈空，则不执行退栈处理，若栈不空，可先将栈顶元素取出，再让栈顶指针减 1，让栈顶退回到次栈顶位置，退栈成功。算法的实现如下。

```
bool Pop(SeqStack& S, SElemType& x) {
//退栈算法调用方式 bool succ = Pop(S, x)。输入：顺序栈 S；输出：若栈不空则算
//法退出栈顶元素的值，并通过引用参数 x 返回该元素的值，同时函数返回 true
//否则函数返回 false，且 x 的值不可引用
     if(S.top == -1) return false;          //若栈空则函数返回 false
     x = S.elem[S.top--];  return true;     //先保存栈顶的值，再让栈顶指针退 1
}
```

算法的时间复杂度为 O(1)，空间复杂度为 O(1)。

3-5　设计一个算法，读取顺序栈 S 的栈顶元素的值而不修改栈顶指针。

【解答】 算法的处理步骤是：先判断是否栈空。若发现栈空，则不执行读取栈顶元素值的处理；若栈不空，可将栈顶元素取出返回即可。算法的实现如下。

```
bool getTop(SeqStack& S, SElemType& x) {
//读取栈顶元素算法的调用方式 bool succ = getTop(S, x)。输入：顺序栈 S
//输出：若栈不空，通过引用参数 x 获取函数值栈顶元素的值，同时函数返回 true，否则
//函数返回 false
     if(S.top == -1) return false;      //判栈空否，若栈空则函数返回 false
     x = S.elem[S.top];  return true;   //返回栈顶元素的值
}
```

算法的时间复杂度为 O(1)，空间复杂度为 O(1)。

3-6　设计一个算法，判断顺序栈 S 是否为空。

【解答】 若栈顶指针 top 退到-1 则栈空。算法的实现如下。

```
bool stackEmpty(SeqStack& S) {
//测试栈 S 空否算法的调用方式 bool succ = stackEmpty(S)。输入: 顺序栈 S
//输出: 若栈空, 则函数返回 true; 否则函数返回 false
     return S.top == -1;
}
```

算法的时间复杂度为 O(1)，空间复杂度为 O(1)。

3-7　设计一个算法，判断顺序栈 S 是否栈满。

【解答】 若栈顶指针等于栈存储空间的最后位置 S.maxSize−1，则栈满。算法的实现如下。

```
bool stackFull(SeqStack& S) {
//测试栈 S 满否算法的调用方式 bool succ = stackFull(S)。输入: 顺序栈 S
//输出: 若栈满, 则函数返回 true; 否则函数返回 false
     return S.top == S.maxSize-1;
}
```

算法的时间复杂度为 O(1)，空间复杂度为 O(1)。

3-8　设计一个算法，计算顺序栈 S 的元素个数。

【解答】 栈顶指针 top 指向最终元素加入位置，栈顶指针加 1 就是栈内元素个数。算法的实现如下。

```
int StackSize(SeqStack& S) {
//算法调用方式 int k = StackSize(S)。输入: 顺序栈 S; 输出: 函数返回栈 S 的长度,
//即栈 S 中元素个数
     return S.top+1;
}
```

算法的时间复杂度为 O(1)，空间复杂度为 O(1)。

3-9　设计一个算法，将顺序栈 S 复制给另一个顺序栈 S1。要求操作前栈 S1 已存在并已初始化（栈存储空间已分配且栈已置空）。

【解答】 将栈 S 的元素顺序传送给 S1，并将栈 S 的栈顶指针的值传送给 S1。算法的实现如下。

```
void stackCopy(SeqStack& S, SeqStack& S1) {
//算法调用方式 stackCopy(S, S1)。输入: 顺序栈 S, 空顺序栈 S1;
//输出: 栈 S 的元素全部传送给栈 S1, 栈 S 的栈顶指针的值也传送给栈 S1
     for(int i = 0; i <= S.top; i++) S1.elem[i] = S.elem[i];
     S1.top = S.top;
}
```

设栈 S 有 n 个元素，则算法的时间复杂度为 O(n)，空间复杂度为 O(1)。

4. 顺序栈相关的算法

3-10　改写顺序栈的进栈成员函数 Push (x)，要求当栈满时执行一个 stackFull( )操作进行溢出处理。其功能是：动态创建一个比原来的栈数组大一倍的新数组，代替原来的栈数组，原来栈数组中的元素占据新数组的前 maxSize 位置。

【解答】 按照题意，算法创建一个与原栈数组同类型但大一倍的新数组，把原栈数组的元素全部复制到新数组的前 maxSize 个位置，再释放原栈数组，用新数组代替原栈数组并修改 maxSize。top 可以不变，因为它表示的是数组下标。算法的实现如下。

```
void StackFull(SeqStack& S) {
     SElemType *temp =(SElemType*) malloc(2*S.maxSize*sizeof(SElemType));
     if(temp == NULL) { printf("存储分配失败! \n");  exit(1); }
     for(int i = 0; i <= S.top; i++) temp[i] = S.elem[i];//传送原栈内的数据
     free(S.elem);                                        //删去原数组
     S.maxSize = 2*S.maxSize;  S.elem = temp;             //数组最大空间增大一倍
}
void Push_1(SeqStack& S, SElemType x) {
//算法调用方式 Push(S, x)。输入: 顺序栈 S, 进栈元素的值 x; 输出: 进栈后更新
//的顺序栈 S
     if(S.top == S.maxSize-1) StackFull(S);          //栈满, 做溢出处理
     S.elem[++S.top] = x;                            //进栈
}
```

设原栈空间的大小为 n，由于调用了 StackFull 操作，算法的时间复杂度为 O(n)，空间复杂度为 O(n)。

3-11　设有一个非空顺序栈 S 存放的整数都大于 0，设计一个算法，借助另一个栈（初始为空）对 S 中的整数排序，使得栈中的整数自栈顶到栈底有序。

【解答】 假设栈中有 n 个整数，可以仿照选择排序的方法实现栈排序，辅助栈为 S1，用以帮助遍历 S 中的元素及存放部分排序结果。具体过程如下。

（1）做 n-1 趟选择，每趟选出最大值整数并放入栈 S1 中，其步骤为

① 将栈 S 中整数逐个移动到 S1，在此过程中选出最小值 x，x 不进入 S1 中；

② 将 S1 中除已排序的整数之外的所有整数移动回栈 S；

③ 将 x 推入到栈 S1。

（2）将 S 中的栈顶元素出栈并推入 S1 栈。

（3）将 S1 栈中所有整数移动到栈 S。

算法的实现如下。

```
void SortStack2(SeqStack& S) {
//算法调用方式 SortStack2(S)。输入: 整数非空栈 S; 输出: S 中所有整数按递增顺
//序排好序
     SeqStack S1;  initStack(S1);
     int x, min, count;
     while(! stackEmpty(S)) {
          Pop(S, min);  count = 0;
          while(! stackEmpty(S) {
```

```
                Pop(S, x);  count++;
                if(x >= min) Push(S1, x);        //比 min 大的整数进栈 S1
                else {Push(S1, min);  min = x;}  //存储新 min, 原 min 进栈 S1
            }
            while(!stackEmpty(S1) && count != 0) {
                Pop(S1, x);  Push(S, x);  count--;
            }
            Push(S1, min);                       //min 进 S1 栈
        }
        while(!stackEmpty(S1)) { Pop(S1, x);  Push(S, x); }
    }
```

设顺序栈中有 n 个元素，算法的时间复杂度为 $O(n^2)$，空间复杂度为 O(n)。

3-12　若有一个非空顺序栈 S 存放的整数都大于 0，设计一个算法，不借助任何辅助数据结构对 S 中的整数排序，使得栈中的整数自栈顶到栈底有序。

【解答】 因为不能借助任何辅助数据结构，题 3-11 的方法不能使用，需要使用递归方法来实现。算法的思路如下：若栈中只有一个元素，排序完成，这是递归的结束条件；否则，先退出栈顶元素存入 x，再递归地对栈中剩余元素排序，使得最小元素上浮到栈顶，然后比较 x 与栈顶元素，若 x 小，直接进栈，若 x 大，将栈顶元素退出存于 y，x 进栈，再递归地对栈中元素排序，最后 y 进栈，排序完成。算法的实现如下。

```
    void SortStack_recur(SeqStack& S) {
    //算法调用方式 SortStack_recur(S)。输入: 非空顺序栈 S; 输出: S 中元素自栈顶到
    //栈底有序排列
        if(!stackEmpty(S)) {                     //栈空, 递归结束
            int x, y;
            Pop(S, x);
            if(!stackEmpty(S)) {
                SortStack_recur(S);              //对栈 S 递归排序
                Pop(S, y);                       //排好序的子栈栈顶 y
                if(x <= y) { Push(S, y);  Push(S, x); }
                else {
                    Push(S, x);  SortStack_recur(S);
                    Push(S, y);
                }
            }
            else Push(S, x);                     //仅一个元素, 递归结束
        }
    }
```

设顺序栈中有 n 个元素，算法的时间复杂度为 $O(n^2)$，时间复杂度为 O(n)。

3-13　设一个顺序栈有 n 个元素，设计一个算法，翻转栈中的所有元素。例如，顺序栈中原来存放的元素是 [1, 2, 3, 4,5]，翻转后栈中元素的排列是 [5, 4, 3, 2, 1]。

【解答】 最常用的办法是利用一个辅助的队列，先把栈中元素依次出栈放到队列里，然后再把队列里的元素依次出队顺序进栈，就可以实现栈的翻转。下面给出的解法是一个

递归的方法。递归过程是先将当前栈的栈底元素移到栈顶，其他元素下移一位，然后对不包含原栈顶的子栈进行同样的操作，递归的结束条件是栈空。算法的实现如下。

```
void move_bot_to_top(SeqStack& S) {
//算法调用方式 move_bot_to_top(S)。输入: 顺序栈 S; 输出: 将当前栈的栈底元素
//移到栈顶, 其他元素下移一位
     if(!stackEmpty(S)) {
          SElemType x, y;
          Pop(S, x);                                  //暂存栈顶元素到 x
          if(!stackEmpty(S)) {
               move_bot_to_top(S);                    //对子栈递归翻转
               Pop(S, y);                             //交换栈顶与子栈栈顶
               Push(S, x);  Push(S, y);
          }
          else Push(S, x);
     }
}
void Reverse_stack(SeqStack& S) {
//算法调用方式 Reverse_stack(S)。输入: 顺序栈 S; 输出: 翻转后的栈 S
     if(!stackEmpty(S)) {
          SElemType x;
          move_bot_to_top(S);
          Pop(S, x);
          Reverse_stack(S);
          Push(S, x);
     }
}
```

设顺序栈中有 n 个元素，算法的时间复杂度为 O(n)，空间复杂度为 O(n)。

3-14　输入两个等长的整数序列，若第一个序列是栈的进栈序列（不一定是 1, 2, 3, …），设计一个算法，判断第二个序列是否是该栈的合理出栈序列。

【解答】 设用两个指针 i 和 j 分别指向 ins 和 outs 当前处理位置，初始位于序列开始位置。然后让 ins 第一个整数进 S 栈，i 加 1，指向 ins 下一个可取位置。然后执行：

（1）循环，判断：当位于 S 栈顶的整数与 outs 中 j 所指整数不等时，在 ins 中依次取下一个整数进 S 栈，直到位于 S 栈顶的整数与 outs 中 j 所指整数相等为止。

（2）退出 S 栈顶元素，在 outs 中让 j 加 1。当 S 栈不空时且在 ins 中 i 未到底时返回到（1），继续执行判断和进栈，当 S 栈空时转到（3）。

（3）若 ins 中 i 未到头，让 i 加 1 再按 i 所指位置取下一整数进 S 栈，返回到（1）。若 ins 中所有整数已处理完且 S 栈空，则 outs 是合理的出栈序列，否则 outs 是不合理的出栈序列。

算法的实现如下。

```
bool isOutStackSeq(int ins[], int outs[], int n) {
//算法调用方式 bool yes = isOutStackSeq(ins, outs, n)。输入: 进栈序列 ins,
//用于比较的另一序列 outs, 序列中整数个位数 n; 输出: 若 outs 是合理的出栈序列, 函数
```

```
//返回 true, 否则函数返回 false
     SeqStack S;  initStack(S);
     int i = 0, j = 0;  int x;              //i 是 ins 指针, j 是 outs 指针
     Push(S, ins[i++]);
     while(i < n && j < n) {
          getTop(S, x);
          if(x == outs[j]) { j++;  Pop(S, x); }
          else {
               if(i < n) Push(S, ins[i]);
               i++;
          }
     }
     if(i == n && j == n-1) return true;
     else return false;
}
```

设进栈序列或出栈序列有 n 个整数，算法的时间复杂度为 O(n)，空间复杂度为 O(n)。

3-15　设顺序栈 S 里有 n 个互不相等的整数，设计一个算法，返回栈中值最小的整数。

【解答】 同样采用递归方法求解。先从栈中退出栈顶元素暂存于 x，如果栈中仅剩一个元素，暂定它就是最小值整数，否则递归地在除已退出的栈顶元素外的其他元素构成的子栈中查找出最小值 y，再比较 x 与 y，若 x＜y，则得到栈中所有元素的最小值 x，否则 y 就是最小值。算法的实现如下。

```
int Min(SeqStack& S) {
//算法调用方式 int value = Min(S)。输入: 整数顺序栈 S; 输出: 函数返回栈内所
//有整数中的最小值
     SElemType x, y;  Pop(S, x);
     if(stackEmpty(S)) { Push(S, x);  return x; }
     else {
         y = Min(S);
         if(x < y) { Push(S, x);  return x; }
         else { Push(S, x);  return y; }
     }
}
```

设栈有 n 个整数，因为使用递归，算法的时间复杂度为 O(n)，空间复杂度为 O(n)。

3-16　若顺序栈 S 中有 n 个互不相等的正整数，设计一个算法，返回栈内值最小的整数，要求算法的时间复杂度为 O(1)。

【解答】 为使算法的时间复杂度达到 O(1)，可以空间换取时间，即设置一个同样大小的栈作为额外的辅助存储，保存当前进栈元素中的最小值。例如，进栈序列为{3, 5, 2, 1, 4}，进栈过程如图 3-1 所示，其中原栈 S，辅助栈 mS。

如果栈空，则当前进栈元素的值 x 成为栈中已有元素的最小值，x 同时进 S 栈和 mS 栈；否则，比较当前进栈元素的值 x 和栈中原有元素的最小值 y（mS 栈的栈顶）：若 x＜y，则 x 将成为栈中最小值，x 同时进 S 栈和 mS 栈；否则，x 进 S 栈，y 进 mS 栈。这样，每次需要取得栈 S 中元素的最小值和 mS 栈的栈顶即可，时间复杂度为 O(1)，空间复杂度仍

为 O(n)。适合此题的进、出栈算法，以及求最小值的算法实现如下。

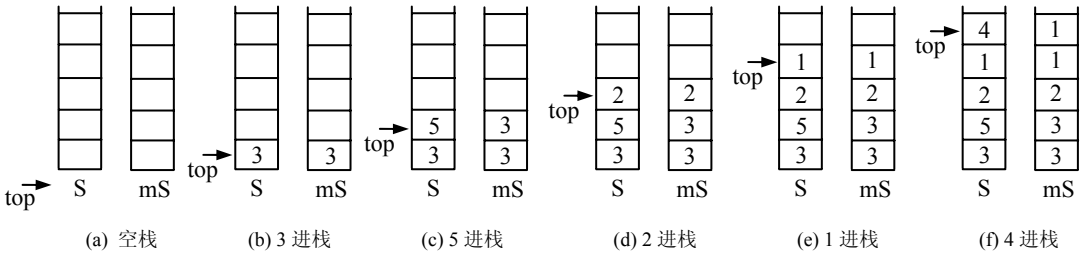


图 3-1　题 3-16 的图

```
bool stackEmpty(SeqStack& S) {
     return S.top == -1;
}
bool Push(SeqStack& S, SeqStack& mS, SElemType x) {
//进栈算法调用方式 bool succ = Push(S, mS, x)。输入: 原栈 S, 存放当前最小值的
//辅助栈 mS, 当前进栈元素 x; 输出: 元素 x 进 S 栈, 当前最小值进 mS 栈, 若进栈成功, 函数
//返回 true; 否则返回 false
     if(S.top == S.maxSize-1) {printf("栈满! \n"); return false; } //栈满
     if(S.top == -1) {S.elem[++S.top] = x;  mS.elem[++mS.top] = x;}//空栈
     else {                                                    //非空栈
          SElemType y = mS.elem[mS.top];
          if(x < y) { S.elem[++S.top] = x;  mS.elem[++mS.top] = x; }
          else { S.elem[++S.top] = x;  mS.elem[++mS.top] = y; }
     }
     return true;
}
bool Pop(SeqStack& S, SeqStack& mS, SElemType& x, SElemType& y) {
     if(S.top == -1) { printf("栈空! \n");  return false;}   //栈空
     x = S.elem[S.top--];  y = mS.elem[mS.top--];
     return true;
}
bool getTop(SeqStack& S, SeqStack& mS, SElemType& x, SElemType& y) {
     if(S.top == -1) { printf("栈空! \n");  return false;}   //栈空
     x = S.elem[S.top];  y = mS.elem[mS.top];
     return true;
}
int Min(SeqStack& S, SeqStack& mS) {
//算法调用方式 int value = Min(S, mS)。输入: 整数顺序栈 S, 辅助栈 mS;
//输出: 函数返回栈内所有整数中的最小值
     if(stackEmpty(S)) return 0;                 //0 是序列中不可能有的数
     else { SElemType x, y;  getTop(S, mS, x, y);  return y; }
}
```

3-17　若进栈序列有 n 个互不相等的整数，设计一个算法，输出所有可能的出栈序列。

【解答】 同样采用递归方法求解，但需要使用辅助栈来保存生成的出栈序列。假设 n =

3，若进栈标记为 I，出栈标记为 O，可用如图 3-2 所示的状态树来表示进栈与出栈情形。

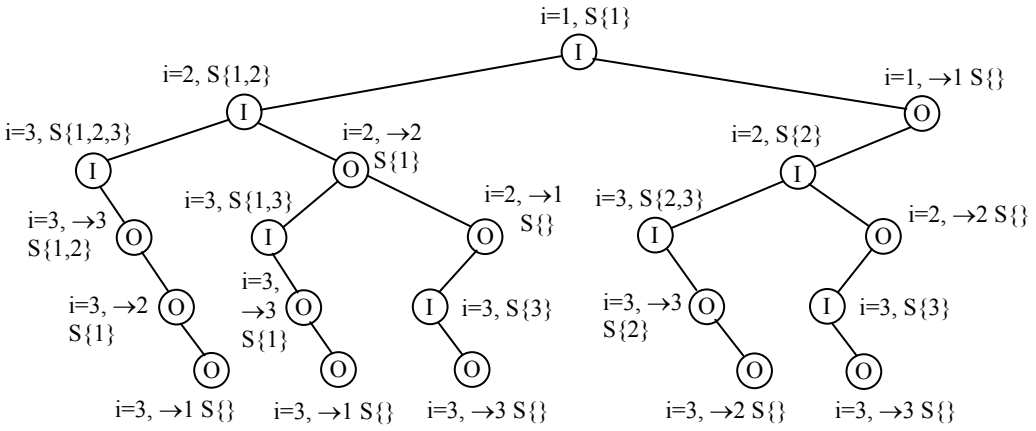


图 3-2　题 3-17 的图

依据进栈操作（I）在任何情况下都比它右边的出栈操作（O）多的要求，不合理的情形都被剪枝，在树中没有画出来，树中从左向右的状态分别是 IIIOOO（输出 321），IIOIOO（输出 231），IIOOIO（输出 213），IOIIOO（输出 132），IOIOIO（输出 123）。算法的思路就是遍历此状态树，按向左进栈，向右出栈，出栈时输出的原则进行处理。每个分支代表一个输出序列。算法的实现如下。

```
#define N 3
void allOutSTK(int A[], int C[], SeqStack& S, int i, int k, int& count){
//算法调用方式 allOutSTK(A, C, S, i, k, count)。输入: 进栈序列 C, 出栈序列 A,
//由于是递归算法, k 是 A 中当前形成出现序列元素数, 初值为 0, i 是 C 中当时取元素指针,
//初值为 0, count 用于统计出栈序列个数, 初值为 0; 输出: 算法输出可能的出栈序列, 栈 S
//存放已进栈序列
     int x, j;
     if(i == N && stackEmpty(S)) {
          printf("  ");
          for(j = 0; j < k; j++) printf("%d ", C[A[j]]);
          printf("\n");
          count++;
     }
     if(i < N) {
          Push(S, i);
          allOutSTK(A, C, S, i+1, k, count);
          Pop(S, x);
     }
     if(!stackEmpty(S)) {
          Pop(S, x);  A[k] = x;
          allOutSTK(A, C, S, i, k+1, count);
          Push(S, x);
     }
}
```

若进栈序列有 n 个元素，算法的时间复杂度为 $O(n^2)$，空间复杂度为 O(n)。

3-18　将编号为 0 和 1 的两个栈存放于一个一维数组空间 elem[maxSize]中，栈底分别处于数组的两端。当第 0 号栈的栈顶指针 top[0]等于−1 时该栈为空，当第 1 号栈的栈顶指针 top[1]等于 maxSize 时该栈为空。两个栈均从两端向中间增长，如图 3-3 所示。

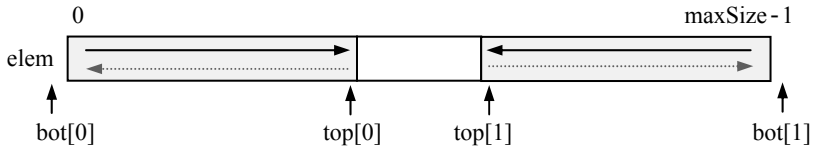


图 3-3　题 3-18 的图

给出这种双栈结构的结构定义，并实现判栈空、判栈满、插入、删除运算算法。

【解答】 双栈的结构定义如下。

```
#define maxSize 20                          //栈存储数组的大小
typedef int SElemType
typedef struct {                            //双栈的结构定义
     int top[2], bot[2];                    //双栈的栈顶指针和栈底指针
     SElemType elem[maxSize];               //栈数组
} DblStack;
```

当向第 0 号栈插入一个新元素时，使 top[0]增 1 得到新的栈顶位置，当向第 1 号栈插入一个新元素时，使 top[1]减 1 得到新的栈顶位置。当 top[0]+1 == top[1]或 top[0] == top[1]−1 时，栈满，此时不能再向任一栈加入新的元素。而双栈的其他操作的实现可以类似定义，相应算法的实现如下。

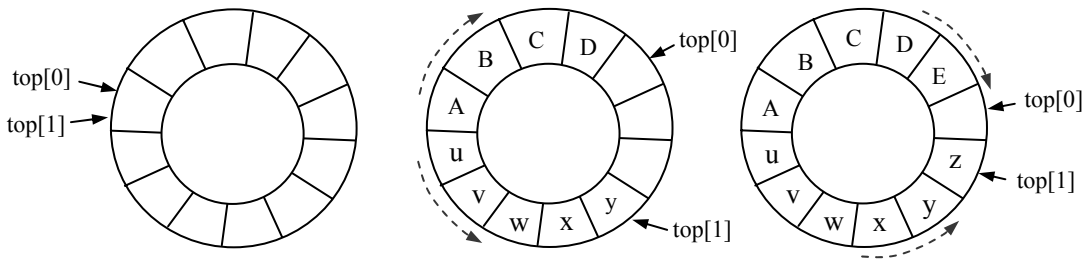
```
void initStack(DblStack& S) {
//初始化函数: 创建一个尺寸为 maxSize 的空栈, 若分配不成功则错误处理
     S.top[0] = S.bot[0] = -1;  S.top[1] = S.bot[1] = maxSize;
}
bool stackEmpty(DblStack& S, int i) {          //判断栈 i 空否
     return S.top[i] == S.bot[i];
}
bool stackFull(DblStack& S) {                  //判断栈满否
     return S.top[0]+1 == S.top[1];
}
bool Push(DblStack& S, SElemType x, int i) {   //进栈运算
     if(stackFull(S)) return false;            //栈满则返回 false
     if(i == 0) S.elem[++S.top[0]] = x;        //栈 0: 栈顶指针先加 1 再进栈
     else S.elem[--S.top[1]] = x;              //栈 1: 栈顶指针先减 1 再进栈
     return true;
}
bool Pop(DblStack& S, int i, SElemType& x) {
//函数通过引用参数 x 返回退出栈 i 栈顶元素的元素值, 前提是栈不为空
     if(stackEmpty(S, i)) return false;        //第 i 个栈栈空, 不能退栈
     if(i == 0) x = S.elem[S.top[0]--];        //栈 0: 先出栈, 栈顶指针减 1
```

```
        else x = S.elem[S.top[1]++];                    //栈 1: 先出栈, 栈顶指针加 1
        return true;
    }
    bool getTop(DblStack& S, int i, SElemType& x) {
    //函数通过引用参数 x 返回栈 i 栈顶元素的元素值, 前提是栈不为空
        if(stackEmpty(S, i)) return false;
        x = S.elem[S.top[i]];
        return true;
    }
    void printStack(DblStack& S, int i) {
        int j;
        printf("第%d 号栈(top=%d): ", i, S.top[i]);
        if(i == 0) for(j = 0; j <= S.top[0]; j++) printf("%d ", S.elem[j]);
        else for(j = maxSize-1; j >= S.top[1]; j--) printf("%d ", S.elem[j]);
        printf("\n");
    }
```

算法的调用方式与顺序栈的基本运算类似，只是在参数表中多了一个 i，表示操作对象是哪一个栈：i= 0 是 0 号栈，i = 1 是 1 号栈。

3-19　将编号为 0 和 1 的两个栈存放于一个数组 elem[0..maxSize-1]中，top[0]和 top[1]分别是它们的栈顶指针。设这两个栈具有共同的栈底，它们的存储数组可视为一个首尾相接的环形数组，0 号栈的栈顶指针顺时针增长，1 号栈的栈顶指针逆时针增长。试问：各栈的栈顶指针指向的位置是否是实际栈顶元素位置？各栈的栈空条件和栈满条件是什么？给出各栈进栈、出栈、判栈满和判栈空、计算栈中元素个数的实现算法。

【解答】 本题的双栈是底靠底的结构，初始时 top[0] = 0，top[1] = maxSize-1，将两个栈的栈顶指针置于数组下标为 0 和 maxSize-1 的位置，表示将两个栈置空，如图 3-4（a）所示，因此，两个栈的栈空条件是(top[1]+1) % maxSize = top[0]。



(a) 双栈的空栈状态　(b) 加入数据之后的状态　(c) 双栈的栈满状态

图 3-4　题 3-19 的图

由于两个栈从同一起点往相反方向增长，相应进栈与出栈策略也相反。

- 对于 0 号栈，进栈时先存后进：S.elem[S.top[0]] = x；S.top[0]++。栈顶指针 top[0]指向实际栈顶的后一位置。对于 1 号栈，进栈时是先进后存：S.top[1]--；A.elem [S.top[1]] = x，栈顶指针指向实际栈顶位置。
- 对于 0 号栈，出栈时先退后取：S.top[0]--；x = S.elem[S.top[0]]。对于 1 号栈，出栈时先取后退：x = S.elem[S.top[1]]；S.top[1]++，如图 3-4(b)所示。栈空条件是

top[0] = top[1]，栈满条件是 S.top[0]+1 = S.top[1]，如图 3-4（c）所示。双栈的结构定义如下。

```
#include<stdio.h>
#include<stdlib.h>
#define maxSize 20                          //栈存储数组的大小
typedef int SElemType;
typedef struct {                            //双栈的结构定义
     int top[2], bot[2];                    //双栈的栈顶和栈底指针
     SElemType elem[maxSize];               //栈数组
} DblStack;
```

双栈的主要操作的实现如下。

```
void initStack(DblStack& S) {
//初始化函数: 两个栈的栈顶指针共享一个位置 0
     S.top[0] = 0;  S.top[1] = 0;  S.bot[0] = 0;  S.bot[1] = 0;
}
bool stackEmpty(DblStack& S , int i) {              //判断栈空否
     return(S.top[i] == S.bot[i]);
}
bool stackFull(DblStack& S) {                       //判断栈满否
     return S.top[0]+1 == S.top[1];
}
bool Push(DblStack& S, SElemType x, int i) {
//进栈运算。若栈满则函数返回 false; 否则元素 x 进栈, 函数返回 true
     if(stackFull(S)) return false;
     if(i == 0) {                                   //0 号栈先存后加
          S.elem[S.top[0]] = x;
          S.top[0] = (S.top[0]+1) % maxSize;
     }
     else {                                         //1 号栈先减后存
          S.top[1] = (S.top[1]-1+maxSize) % maxSize;
          S.elem[S.top[1]] = x;
     }
     return true;
}
bool Pop(DblStack& S, int i, SElemType& x) {
//退栈运算。若栈空则函数返回 false, 否则退栈元素保存到 x, 函数返回 true
     if(stackEmpty(S, i)) return false;             //i 号栈空则返回 false
     if(i == 0) {                                   //0 号栈先减后取
          S.top[0] =(S.top[0]-1+maxSize) % maxSize;
          x = S.elem[S.top[0]];
     }
     else {                                         //1 号栈先取后加
          x = S.elem[S.top[1]];
          S.top[1]=(S.top[1]+1) % maxSize;
     }
```

```
        return true;
    }
    bool getTop(DblStack& S, int i, SElemType& x) {
    //若栈不空则函数通过 x 返回该栈栈顶元素的内容
        if(stackEmpty(S, i)) return false;          //i 号栈空则返回 false
        if(i == 0) x = S.elem[(S.top[0]-1+maxSize) % maxSize];
        else x = S.elem[S.top[1]];
        return true;
    }
```

算法的调用方式与顺序栈类似，只是多了对 0 号栈和 1 号栈的控制。

3-20　设计一个算法，识别依次读入的一个以'\0'为结束符的字符序列是否为形如"序列 1&序列 2"的字符序列。其中序列 1 和序列 2 中都不含字符'&'，且序列 2 是序列 1 的逆序列。例如，"a+b&b+a"是属于该模式的字符序列，而"1+3&3-1"则不是。

【解答】 对于给定的字符序列，从头逐个读字符，边读边入栈，直到“&”为止，然后再逐个读“&”之后的字符，边读边与退栈字符做比较，若不相等，则序列 2 不是序列 1 的逆序列，若相等继续进行读字符、退栈、比较。算法的实现如下。

```
    bool isReverse(char A[]) {
    //算法调用方式 bool succ = isReverse(A)。输入: 输入字符序列存储数组 A;
    //输出: 若输入字符串 A 中'&'前和'&'后部分是逆串, 则函数返回 true, 否则返回 false
        char S[maxSize];  int top = -1;
        char ch;  int i = 0;
        while(A[i] != '&' && A[i] != '\0')     //将字符序列中'&'的前半部分进栈
            { S[++top] = A[i];  i++; }
        if(A[i] == '\0') return false;         //若没有遇到'&'则返回 false
        while(A[++i] != '\0' && top >= 0){     //栈中字符与 A 剩余部分继续比较
            ch = S[top--];
            if(A[i] != ch) return false;
        }
        if((A[i] == '\0' && top == -1)) return true;
        else return false;
    }
```

若字符串中字符个数为 n，算法的时间复杂度为 O(n)，空间复杂度为 O(n)。

3.1.3　链式栈

1. 链式栈的概念和结构类型的定义

用单链表作为存储结构的栈称为链式栈。链式栈的表示如图 3-5 所示。

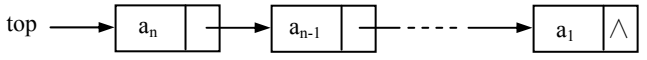


图 3-5　链式栈

从图 3-5 可知，链式栈的栈顶在链表的表头。因此，新结点的插入和栈顶结点的删除都在链表的表头，即栈顶进行。通常情况下，作为链式栈使用的链表不需要头结点，链表

的表头指针就是栈顶指针。链式栈的类型定义如下（保存于头文件 LinkStack.h 中）：

```
#include<stdio.h>
#include<stdlib.h>
#define maxSize 20
typedef int SElemType;
typedef struct node {
     SElemType data;
     struct node *link;
} LinkNode, *LinkList, *LinkStack;
```

虽然链式栈不带头结点，但也需要初始化运算，把将要投入使用的栈置空。

```
void initStack(LinkStack& S) {
     S = NULL;
}
```

采用链式栈来表示一个栈，便于结点的插入与删除。在程序中同时使用多个栈的情况下，用链接表示不仅能够提高效率，还可以达到共享存储空间的目的。

2. 链式栈基本运算的实现

以下 5 个算法都存放在程序文件 LinkStack.cpp 中，可直接链接使用。

3-21　设计一个算法，将元素 x 压入链式栈 S 中。

【解答】 链式栈的栈顶在链头，只要把新 x 结点链入链头，使其成为新的首元结点即可。算法的实现如下。

```
bool Push(LinkStack& S, SElemType x) {
//进栈算法调用方式 bool succ = Push(S, x)。输入: 链式栈 S, 进栈元素 x; 输出:
//进栈后更新的链式栈 S。函数返回 true
     LinkNode *p =(LinkNode*) malloc(sizeof(LinkNode)); //创建新结点
     p->data = x;  p->link = S;  S = p;                 //新结点插入在链头
     return true;
}
```

算法的时间复杂度为 O(1)，空间复杂度为 O(1)。

3-22　设计一个算法，从链式栈 S 退出栈顶的元素，其值通过引用参数返回。

【解答】 链式栈的栈顶在链头，退栈时把链表的首元结点摘下即可。算法的实现如下。

```
bool Pop(LinkStack& S, SElemType& x) {
//退栈算法调用方式 bool succ = Pop(S, x)。输入: 链式栈 S; 输出: 栈顶元素出栈
//后更新的链式栈 S, 若退栈成功, 函数返回 true, 同时引用参数 x 返回被删栈顶元
//素的值; 若栈空则函数返回 false, 引用参数 x 的值不可用
     if(S == NULL) return false;                //栈空函数返回 0
     LinkNode *p = S;  x = p->data;             //存栈顶元素
     S = p->link;  free(p);  return true;       //栈顶指针退到新栈顶位置
}
```

算法的时间复杂度为 O(1)，空间复杂度为 O(1)。

3-23　设计一个算法，读取链式栈 S 的栈顶元素。

【解答】 在读取栈顶元素的值时不对链式栈做任何改变。算法的实现如下。

```
bool getTop(LinkStack& S, SElemType& x) {
//读取栈顶算法调用的方式 bool succ = getTop(S, x)。输入：链式栈 S；输出：若栈
//不空，则操作成功，函数返回 true，引用参数 x 返回栈顶元素的值；若栈空，函数返回
//false，引用参数 x 的值不可用
     if(S == NULL) return false;                    //栈空函数返回 false
     x = S->data;  return true;                     //栈不空则返回 true
}
```

算法的时间复杂度为 O(1)，空间复杂度为 O(1)。

3-24　设计一个算法，判断链式栈是否为空，是则函数返回 true；否则函数返回 false。

【解答】 由于链式栈没有头结点，栈顶指针直接指向首元结点，如果表头指针为空，说明链表中一个结点也没有，链表为空。因此要判断栈空，可直接判断链表表头指针是否为空。算法的实现如下。

```
bool stackEmpty(LinkStack& S) {
//判断栈是否为空算法的调用方式 bool succ = stackEmpty(S)。输入：链式栈 S；
//输出：若栈空，则函数返回 true；否则函数返回 false
     return S == NULL;
}
```

算法的时间复杂度为 O(1)，空间复杂度为 O(1)。

3-25　设计一个算法，求链式栈的长度，即链表的结点个数。

【解答】 从栈顶开始顺序扫描链表，同时统计经过的结点数即可。算法的实现如下。

```
int stackSize(LinkStack& S) {
//求栈长度算法的调用方式 int k = stackSize(S)。输入：链式栈 S；
//输出：函数返回求得的栈元素个数
     LinkNode *p = S;  int count = 0;
     while(p != NULL) { p = p->link;  count++; }    //逐个结点计数
     return count;
}
```

若栈中有 n 个元素，算法的时间复杂度为 O(n)，空间复杂度为 O(1)。

3. 链式栈相关的算法

3-26　设计一个算法，借助栈判断存储在单链表中的数据是否中心对称。例如，单链表中的数据序列{12, 21, 27, 21, 12}或{13, 20, 38, 38, 20, 13}即为中心对称。

【解答】 设置一个栈 S，算法首先遍历一次单链表，把所有结点数据顺序进栈；然后同时做两件事：再次从头遍历单链表和退栈。每次访问一个链表结点并与退栈元素比较，若比较相等，则继续比较链表下一结点并退栈；若比较不等则不是中心对称，返回 false。若单链表遍历完则表示链表是中心对称，返回 true。算法的实现如下。

```
bool centreSym(LinkList& L) {
//算法调用方式 bool succ = centreSym(L)。输入：单链表 L；输出：
```

```
//若链表是中心对称，函数返回true，否则函数返回false
     LinkStack S;  initStack(S);  SElemType x;
     for(LinkNode *p = L->link; p != NULL; p = p->link)
          Push(S, p->data);
     p = L->link;
     while(p != NULL) {
          Pop(S, x);
          if(p->data != x) return false;
          else p = p->link;
     }
     return true;
}
```

若栈中有 n 个元素，算法的时间复杂度为 O(n)，空间复杂度为 O(1)。

3-27　设计一个算法，借助栈实现单链表上链接顺序的逆转。

【解答】 算法首先遍历单链表，逐个结点删除并把被删结点存入栈中，再从栈中取出存放的结点把它们依次链入单链表中。通过栈把结点的次序颠倒过来。算法的描述如下。

本题在函数体内直接定义了一个栈 S，因为栈元素的数据类型是链表结点。

```
void Reverse(LinkList& L) {
//算法调用方式Reverse(L)。输入：单链表L；输出：经逆转后的单链表L
     LinkNode *S = NULL;  LinkNode *p, *q;
     while(L->link != NULL) {                //检测原链表
         p = L->link;  L->link = p->link;
         p->link = S;  S = p;                //结点p从原链表摘下，进栈
     }
     p = L;
     while(S != NULL) {                      //当栈不空时
         q = S;  S = S->link;                //退栈，退出元素由q指向
         p->link = q;  p = q;                //链入结果链尾
     }
     p->link = NULL;                         //链收尾
}
```

若栈中有 n 个元素，算法的时间复杂度为 O(n)，空间复杂度为 O(1)。

3-28　设计一个算法，利用栈逆向输出一个单链表的所有数据。

【解答】 算法首先遍历单链表，逐个访问单链表的结点，把结点数据存入栈中；然后再从栈中顺序取出存放的数据并输出它们。算法的实现如下。

```
void reverseOut(LinkList& L) {
//算法调用方式reverseOut(L)。输入：单链表L；输出：算法逆向输出单链表L
     LinkStack S;  initStack(S);
     for(LinkNode *p = L->link; p != NULL; p = p->link)
          Push(S, p->data);                  //所有数据进栈
     SElemType x;  int first = 1;
     while(! stackEmpty(S)) {
          Pop(S, x);
```

```
            if(first) { printf("%d", x);  first = 0; }
            else printf(", %d", x);
        }
        printf("\n");
    }
```

若栈中有 n 个元素，算法的时间复杂度为 O(n)，空间复杂度为 O(n)。

3-29　若有一个元素类型为整型的栈 S，设计一个算法，借助另一个栈实现把该栈的所有元素从栈顶到栈底按从小到大的次序排列起来。

【解答】 设待排序的栈为 S，辅助排序的栈为 T。算法实现的步骤如下。

（1）循环执行从栈 S 的栈顶退出一个整数，置于 k：

- 若栈 T 不为空，则从栈 T 退出所有大于 k 的元素，送回 S 栈。
- 若栈 T 已空，或栈 T 的栈顶元素的值小于或等于 k，则把 k 压入栈 T。

（2）执行完步骤（1）后，所有栈 T 中元素逐个弹出，压入栈 S，算法结束。

算法的实现如下。

```
    void StackSort(LinkStack& S) {
    //算法调用方式 StackSort(S)。输入：链式栈 S；输出：排好序的链式栈 S
        LinkStack T;  initStack(T);     int i, k;
        while(!stackEmpty(S)) {
            Pop(S, k);                              //从栈 S 弹出一个整数 k
            while(!stackEmpty(T)) {
                getTop(T, i);                       //读取栈 T 的栈顶 i
                if(i > k){Pop(T, i);  Push(S, i);}  //k 比 i 小，i 退出栈 T 回栈 S
                else break;                         //k 比 i 大，停止循环
            }
            Push(T, k);                             //k 压入栈
        }
        while(! stackEmpty(T)) { Pop(T, k);  Push(S, k); }
    }
```

若栈中有 n 个元素，算法的时间复杂度为 $O(n^2)$，空间复杂度为 O(1)。

3-30　若有一个元素类型为整型的栈 S，设计一个算法，返回栈中所有元素的中位值。例如，栈中元素为{3, 2, 4, 5, 7, 6, 1}，元素个数 n 为奇数，中位值为 4，即栈中第(n+1)/2 小的元素；若栈中元素为{3, 2, 7, 5, 8, 6, 1, 4}，n 为偶数，中位值为 4，即栈中第 n/2 小的元素。

【解答】 一种解决方案是利用题 3-29 的算法先把栈中所有元素排序，再按栈中元素个数 n 取第 n/2（n 是偶数）或第(n+1)/2（n 是奇数）个元素作为栈中所有元素的中位值。另一种解决方案是借助一个辅助数组 a[n]，在进栈的过程中把进栈元素 x 放到 a 中适当位置，使得 a 中元素按值从小到大有序，这样就可以在数组 a 中取中位值了。不过这样做就必须在进栈、出栈过程中重排数组 a 的元素，算法的实现如下。

```
    #define N 16
    void Push_1(LinkStack& S, int x, int a[], int& top) {
    //算法调用方式 Push_1(S, x, a, top)。输入：链式栈 S，进栈元素 x，排序数组 a，a
```

```
//中最末元素位置top, 引用参数top初值为-1; 输出: 元素x进栈, 同时插入a中适
//当位置, 使a中元素保持有序, 引用参数top加1
     Push(S, x);
     int i = top;
     while(i >= 0 && a[i] > x) { a[i+1] = a[i]; i--; }
     a[i+1] = x;  top++;                          //若有相等元素, x加在后面
}
bool Pop_1(LinkStack& S, int& x, int a[], int& top) {
//算法调用方式Pop_1(S, x, a, top)。输入: 链式栈S, 排序数组a, a中最末元素位置
//top; 输出: 引用参数x返回出栈元素的值, 同时在a中删除该元素, 并使a中元素保持有序,
//引用参数top减1
     if(stackEmpty(S)) { printf("栈空不能退栈! \n");  return false; }
     Pop(S, x);
     int i = 0, j;                                //调整a[i-1]并保持a有序
     while(i <= top && a[i] <= x) i++;            //找到首个a[i] > x的元素
     for(j = i; j <= top; j++) a[j-1] = a[j];   //后面元素前移填补a[i-1]
     top--;  return true;
}
int getMedian(LinkStack& S, int a[]) {
//算法调用方式int v = getMedian(S, a)。输入: 非空链式栈S, 辅助排序数组a;
//输出: 函数返回中位值
     if(stackEmpty(S)) {printf("未找到中位值! \n");  return 0;}//空栈
     int i, n = stackSize(S);                     //n是当前栈中元素个数
     i =(n % 2 == 0) ? n/2-1 :(n+1)/2-1;          //取中位值在a中下标
     return a[i];
}
```

进栈和出栈算法的时间复杂度与空间复杂度均为 O(n)；取中位值算法的时间复杂度为 O(1)，空间复杂度为 O(n)，n 是栈中元素个数。

3.2　队　　列

3.2.1　队列的定义及基本运算

队列（Queue）是一种限定存取位置的线性表。它只允许从表的一端插入元素，从另一端删除元素。允许插入的一端称为队尾（rear），允许删除的一端称为队头（front）。

新元素每次都在队尾插入，且最早进入队列的元素最先退出队列。队列所具有的这种特性就称为先进先出（First In First Out，FIFO）。

队列的基本运算如下。

- 队列初始化 void InitQueue(Queue& Q)：创建一个空的队列 Q 并初始化。
- 判队列空否 bool QueueEmpty(Queue& Q)：队列 Q 为空则函数返回 true，否则返回 false。
- 进队列 bool EnQueue(Queue& Q, type x)：当队列 Q 未满时函数将元素 x 插入并成为新的队尾，若操作成功，则函数返回 true，否则返回 false。