第3章 栈和队列

CHAPTER

栈和队列是两种特殊的线性结构。从数据结构的角度看，栈和队列也是一种线性表。它们的特殊性在于栈和队列的操作相比于线性表受到了一定的限制。因此，栈和队列也称为操作受限的线性表。

生活中栈和队列的使用十分广泛。例如枪膛的子弹与叠起的餐盆，只在一端进一端出，即为栈；而排队，一端进另一端出，则为队列。栈和队列因操作受限而有了特殊的特性，在解决问题中起到特殊的作用。

本章除了讨论栈和队列的定义和存储结构外，还将侧重介绍栈的“先进后出”原则和队列的“先进先出”原则，给出一些应用实例，以体会两种特殊线性表的应用场景。

本章主要知识点

- 栈和队列的结构特性。
- 栈和队列的顺序存储实现。
- 栈和队列的链式存储实现。

本章教学目标

- 掌握栈和队列的结构特性与操作特性。
- 灵活运用栈和队列解决应用问题。

3.1 栈

3.1.1 栈的定义和特点


微课视频

栈(stack)是限定只能在一端进行插入或删除操作的线性表。表中允许进行插入和删除操作的一端称为**栈顶**(top)，另一端相应地称为**栈底**(bottom)。栈中没有数据元素时称为**空栈**。栈的插入操作称为**入栈**/进栈/压栈(push)，栈的删除操作称为**出栈**/弹栈(pop)。栈结构如图 3-1 所示。

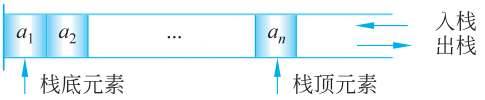


图 3-1　栈结构示意图

在栈中，无论是存数据还是取数据，都必须遵循“先进后出”原则，即最先入栈的元素最后出栈。因此，栈又称为**先进后出**（first in last out，FILO）或后进先出的线性表，简称为 **FILO 线性表**。如图 3-1 所示的栈，从当前数据的存储状态可知，元素 a_1 是最先入栈的。因此，当要从栈中取出元素 a_1 时，需要提前将元素 a_n，a_{n-1}，…，a_2 从栈中取出，然后才能成功取出 a_1。

图 3-2 是一个栈的操作示意图，图中箭头表示当前栈顶元素位置。图 3-2(a)表示一个空栈；图 3-2(b)表示插入一个元素 a 之后栈的状态；图 3-2(c)表示插入元素 b、c、d 之后栈的状态；图 3-2(d)表示删除一个元素 d 之后栈的状态。

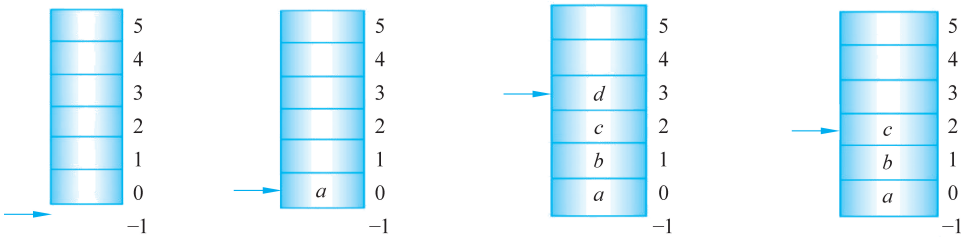


图 3-2　栈操作示意图

由于栈是一种只能从表的一端存取数据且遵循“先进后出”原则的线性存储结构，因此线性表的操作特性它都具备，只是栈的插入和删除操作改名为入栈和出栈。除此之外，栈的基本操作还包括初始化、判栈空、取栈顶元素等。

下面是栈的抽象数据类型 Stack 的定义。

```
ADT Stack {
数据对象：D={a_i | a_i ∈ ElementSet, i=1, 2, …, n, n≥0}。
数据关系：R={<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)                    //初始化栈
        操作功能：创建一个空的栈 S。
        操作输出：创建成功，返回 true；不成功，退出。
    DestroyStack(&S)                 //销毁栈
        操作功能：释放栈 S 所占的存储空间。
        操作输出：无。
    GetTop(S,&e)                     //取栈顶元素
        操作功能：读取栈 S 的栈顶元素。
        操作输出：如果栈不空，输出栈顶元素，不修改栈顶指针。
    Push(&S,e)                       //入栈
        操作功能：在栈顶插入新元素 e。
        操作输出：插入元素 e 为新的栈顶元素。
    Pop(&S,&e)                       //出栈
```

```
        操作功能：删除栈顶元素。
        操作输出：删除栈 S 的栈顶元素并用 e 返回其值。
    StackLength(S)                    //测栈长
        操作功能：求栈 S 的长度。
        操作输出：返回栈 S 的元素个数。
    StackEmpty(S)                     //判栈空
        操作功能：判断栈 S 是否为空。
        操作输出：如果 S 是空栈，返回 true；否则，返回 false。
    ClearStack(&S)                    //清空栈
        操作功能：删除栈 S 中的所有元素。
        操作输出：栈 S 被置为空栈。
}ADT Stack
```

【例 3-1】　设有 4 个元素 a、b、c、d 进栈，给出它们所有可能的出栈次序。

【解】　栈对线性表的插入和删除位置进行了限制，但并没有对元素进出的时间进行限制。也就是说，在不是所有元素都进栈的情况下，先进栈的元素也可以出栈，只要保证是栈顶元素出栈即可。因此，所有可能的出栈次序如下：$abcd$、$abdc$、$acbd$、$acdb$、$adcb$、$bacd$、$badc$、$bcad$、$bcda$、$bdca$、$cbad$、$cbda$、$cdba$、$dcba$。

【例 3-2】　设 n 个元素进栈序列是 1，2，…，n，其输出序列是 p_1，p_2，…，p_n，若 $p_1=3$，则 p_2 的值_____。

A. 一定是 2　　B. 一定是 1　　C. 不可能是 1　　D. 以上都不对

【解】　当 $p_1=3$ 时，说明 1，2，3 先进栈，立即出栈 3，然后可能出栈的为 2，也可能为 4 或后面的元素进栈，再出栈。因此，p_2 可能是 2，也可能是 4，…，n，但一定不能是 1。因此本题答案为 C。

与线性表类似，栈也有两种存储结构，分别称为顺序栈和链栈。

3.1.2　顺序栈


微课视频

顺序栈(sequential stack)**是用顺序表实现栈的存储结构**，即利用一组地址连续的存储单元依次存放自栈底到栈顶的数据元素。

连续内存的使用是基于起始地址的，设指针 base 为这个存储空间的基地址。栈的操作只能在栈顶进行，为方便操作，定义一个 top 变量指示栈顶元素在顺序栈中的位置①。另设栈的容量为 stacksize。顺序栈的存储示意图如图 3-3 所示，存储结构定义如下。

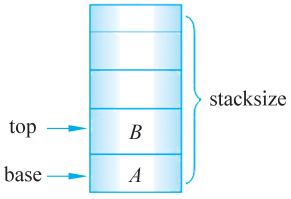


图 3-3　顺序栈存储示意图

```
template<class DT>
struct SqStack
{ DT * base;              //栈底指针
  int top;                //栈顶
  int stacksize;          //栈可用的最大容量
}
```

由上述定义可知下列语句的含义。

① 也有将栈顶指针设在栈顶元素的上方，此种情况下空栈 top=1。解题时，需要了解清楚题目中的设置。

SqStack S表示声明一个顺序栈变量S。

S.base[S.top]表示顺序栈S的栈顶元素。

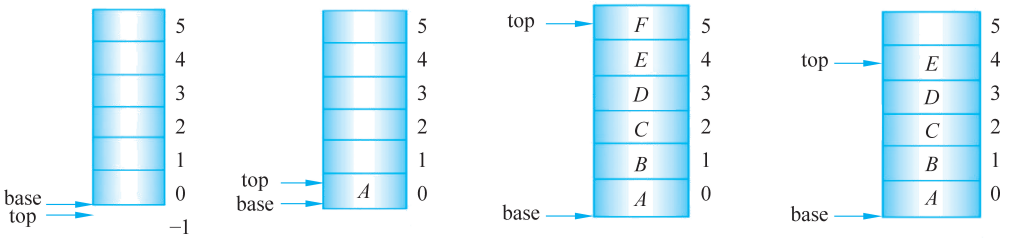
S.top表示栈顶元素位置。当S.top==-1，表示栈空，无法出栈。

S.stacksize表示顺序栈S的容量。当S.top==S.stacksize-1，表示栈满，无法入栈。

这里约定base为栈底指针，若base的值为NULL，表示栈结构不存在；元素e进栈时，先将top增1，然后将其放在栈顶指针处；元素要出栈时，先将栈顶指针处的元素取出，然后top减1。因此，top+1的值实际上反映了栈中元素的个数，与顺序表中length值的意义相同。

【思考】 因为连续内存单元容量受限，所以在顺序表中设置了表长属性length(线性表中元素的个数)，以指示连续内存中已被用掉的单元并由它判断表空和表满。顺序栈中没有设置栈的长度，如何判断栈空和栈满？

图3-4所示为栈空、栈满、入栈以及出栈时top的特征及变化。



(a) 空栈　(b) 入栈　(c) 栈满　(d) 出栈

图3-4　栈顶指针与栈中元素的关系

由于顺序栈的插入和删除操作限制在栈顶进行，因此对应栈的基本操作比顺序表要简单得多。根据ADT Stack中的基本操作定义设置操作参数类型和返回值后，顺序栈的基本操作的函数定义简述如表3-1所示。

表3-1　顺序栈的基本操作的函数定义

序号	函数定义	功能说明
1	//初始化顺序栈 bool InitStack(SqStack<DT> &S,int m)	创建一个容量为 m 的空栈S；创建成功，返回true；否则，退出
2	//销毁顺序栈 void DestroyStack(SqStack<DT> &S)	释放顺序栈S所占内存空间
3	//取栈顶元素 bool GetTop(SqStack<DT> S,DT &e)	栈S空返回false；否则取栈顶元素赋给e，返回true
4	//入栈 bool Push(SqStack<DT> &S,DT e)	栈S满返回false；否则在栈顶插入一个值为e的元素，返回true
5	//出栈 bool Pop(SqStack<DT> &S,DT &e)	栈S空返回false；否则删除栈顶元素，返回true
6	//测栈长 int StackLength(SqStack<DT> S)	求顺序栈S的长度，即栈中元素个数

续表

序号	函 数 定 义	功 能 说 明
7	//判栈空 bool StackEmpty(SqStack<DT> S)	判断顺序栈 S 是否为空栈。如果是空栈，返回 true；否则，返回 false
8	//清空栈 void ClearStack(SqStack<DT> &S)	把顺序栈变成空栈

顺序栈部分基本操作的实现算法如下。

1. 初始化顺序栈

【算法思想】　申请一组连续的内存空间，用来存放顺序栈，使 base 指向这段空间的基址；新栈为空栈，栈顶 top 设为－1；申请的容量为栈最大可用容量 m。

算法描述与算法步骤如算法 3.1 所示。

算法 3.1　【算法描述】　　【算法步骤】

```
0  template<class DT>
1  bool InitStack(SqStack<DT>&S,int m)          //创建容量为 m 的空栈
2  { S.base=new DT[m];                          //1.申请一组连续的内存空间
3    if(!S.base) exit(1);                       //申请失败，退出
4    S.top=-1;                                  //2.申请成功，为栈属性赋值
5    S.stacksize=m;
6    return true;                               //3.返回 true
7  }
```

2. 销毁顺序栈

【算法思想】　对应 new 申请内存空间命令，用 delete 释放顺序栈 S 所占内存空间；置栈顶 top 为－1；栈最大可用容量 StackSize 为 0。

算法描述与算法步骤如算法 3.2 所示。

算法 3.2　【算法描述】　　【算法步骤】

```
0  template<class DT>
1  void DestroyStack(SqStack<DT>&S)             //销毁顺序栈
2  {
3     delete [] S.base;                         //1.释放顺序栈占用的内存空间
4     S.top=-1;                                 //2.为栈属性赋值
5     S.stacksize=0;
6  }
```

3. 入栈

【算法思想】　判断栈是否满，满则返回 false；否则将栈顶指针增 1，新元素插入栈顶指针位置。

算法描述与算法步骤如算法 3.3 所示。

算法 3.3 【算法描述】　　【算法步骤】

```
0  template<class DT>
1  bool Push(SqStack<DT>&S, DT e)                  //在栈顶插入一个新元素
2  {
3    if(S.top==S.stacksize-1)                      //1.栈满的情况,即栈上溢出
4       return false;                              //无法入栈,返回 false;
5    S.top++;                                      //2.栈顶指针增 1
6    S.base[S.top]=e;                              //元素 e 放在栈顶指针处
7    return true;                                  //3.返回 true
8  }
```

4. 出栈

【算法思想】 判断栈是否空，空则返回 false；否则取栈顶元素赋给 e，然后将栈顶指针减 1。

算法描述与算法步骤如算法 3.4 所示。

算法 3.4 【算法描述】　　【算法步骤】

```
0  template<class DT>
1  bool Pop(SqStack<DT>&S,DT &e)                   //删除栈顶元素
2  {
3    if(S.top==-1)                                 //1.栈空的情况,即栈下溢出
4      return false;                               //无法出栈,返回 false
5    e=S.base[S.top];                              //2.取栈顶元素,赋值给 e
6    S.top--;                                      //栈顶指针减 1
7    return true;                                  //3.返回 true
8  }
```

5. 取栈顶元素

【算法思想】 判断栈是否空，空则返回 false；否则取栈顶元素赋给 e。此操作栈顶指针保持不变。

算法描述与算法步骤如算法 3.5 所示。

算法 3.5 【算法描述】　　【算法步骤】

```
0  template<class DT>
1  bool GetTop(SqStack<DT>S,DT &e)                 //取栈顶元素
2  {
3    if(S.top==-1)                                 //1.栈空的情况,即栈下溢出
4       return false;                              //无法出栈,返回 false
5    e=S.base[S.top];                              //2.取栈顶元素,赋值给 e
6    return true;                                  //3.返回 true
7  }
```

顺序栈与顺序表一样，操作时会受最大空间容量的限制，虽然在出现“满”的情况时可以通过重新分配存储空间来扩大容量，但此项工作量大，应尽量避免。如果在一个程序中

需要同时使用具有相同数据类型的两个栈，那么可以为它们各自开辟数组空间，也可以用一个数组来存储两个栈，具体做法如下。

让一个栈的栈底在数组的始端，即下标为 0 处；让另一个栈的栈底在数组的末端，即下标为数组长度 $n-1$ 处。这样，两个栈如果增加元素，每个栈从各自的端点向中间延伸。假设 top1 和 top2 分别是栈 1 和栈 2 的栈顶指针，如图 3-5 所示。可以想象，只要它们不“见面”，两个栈就可以一直使用。从这里也就可以分析出来，当 top1 等于 -1 时，栈 1 为空；当 top2 等于 n 时，栈 2 为空。那么，什么时候栈满呢？

图 3-5　栈顶指针与栈中元素的关系

可以思考极端的情况：若栈 2 是空栈，栈 1 的 top1 等于 $n-1$ 时，为栈 1 满；反之，当栈 1 为空栈，top2 等于 0 时，为栈 2 满。但更多的情况是在两个栈“见面”之时，也就是两个指针之间相差 1(即 top1+1==top2)时栈满。

此时，只有当整个存储空间都被两个栈占满，才会发生上溢。使用这样的数据结构通常都是当两个栈的空间需求有相反关系时，也就是一个栈增长另一个栈在缩短的情况。这样使用两栈共享存储空间才有较大意义。两栈共享一个长度为 n 的存储空间与两个栈分别占用两个长度为 $\lfloor n/2 \rfloor$ 和 $\lceil n/2 \rceil$[①] 的存储空间相比较，前者发生上溢的概率要比后者小得多。

在实际应用中，如果应用程序无法预先估计栈可能达到的最大容量，建议使用链栈。


微课视频

3.1.3　链栈

链栈(linked stack)**是用链表实现栈的存储结构**，通常用单链表表示。

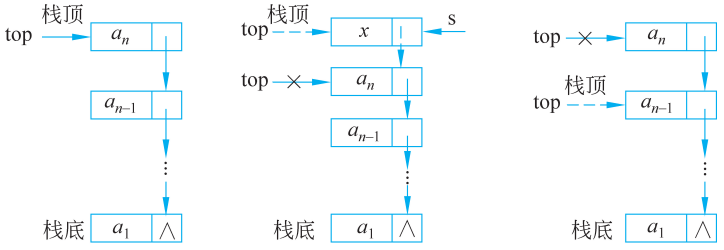
对于链栈来说，一般不考虑栈满上溢的情况，除非内存已经没有可以使用的空间。由于链栈的主要操作是在栈顶进行插入和删除，因此把栈顶放在单链表的头部是最方便的，这样可以避免实现数据“入栈”和“出栈”时做大量遍历链表的耗时操作。而且，在链栈中没有必要像单链表那样为操作方便附加一个头结点。因此，链栈实际上是一个只能采用头插法插入或删除数据的单链表。

链栈的存储及操作示意图如图 3-6 所示。

链栈的结点结构与单链表的结点结构相同，其存储结构定义如下：

```
template<class DT>
struct SNode            //结点类型名
{ DT data;              //数据域,存储数据元素
  SNode * next;         //指针域,指向后继结点
};
```

① $\lfloor n/2 \rfloor$ 表示下取整，即不大于 $n/2$ 的最大整数；$\lceil n/2 \rceil$ 表示上取整，即不小于 $n/2$ 的最小整数。



(a) 链栈存储示意图　(b) 入栈　(c) 出栈

图 3-6　链栈的存储及操作示意图

标识链栈的是指向栈顶元素结点的头指针。

根据定义，声明一个链栈变量 S 的语句为 SNode * S。

根据 ADT Stack 中的基本操作定义设置操作参数类型和返回值后，链栈的基本操作的函数定义简述如表 3-2 所示。

表 3-2　链栈的基本操作的函数定义

序号	函 数 定 义	功 能 说 明
1	//初始化链栈 bool InitStack(SNode<DT> * &S)	创建一个空栈 S，返回 true
2	//销毁链栈 void DestroyStack(SNode<DT> * &S)	释放链栈 S 所占内存空间
3	//取栈顶元素 bool GetTop(SNode<DT> * S,DT &e)	栈 S 空返回 false；否则取栈顶元素赋给 e，返回 true
4	//入栈 bool Push(SNode<DT> * &S,DT e)	在栈 S 的栈顶插入一个值为 e 的元素，返回 true
5	//出栈 bool Pop(SNode<DT> * &S,DT &e)	栈 S 空返回 false；否则删除栈顶元素，返回 true
6	//测栈长 int StackLength(SNode<DT> * S)	求链栈 S 的长度，即栈中元素个数
7	//判栈空 bool StackEmpty(SNode<DT> * S)	判断链栈 S 是否为空栈。如果是空栈，返回 true；否则，返回 false
8	//清空栈 void ClearStack(SNode<DT> * &S)	把链栈变成空栈

下面给出链栈中部分基本操作的实现算法。

1. 初始化链栈

【算法思想】　构造一个空栈，直接将栈顶指针置空即可。

算法描述与算法步骤如算法 3.6 所示。

算法 3.6　【算法描述】　　【算法步骤】

```
template<class DT>
bool InitStack(SNode<DT> * &S)                    //构造一个空栈
{
  S=NULL;                                         //1.栈顶指针置空
  return true;                                    //2.返回 true
}
```

2. 销毁链栈

【算法思想】 从栈顶结点开始，将栈 S 中的结点逐个销毁。

算法描述与算法步骤如算法 3.7 所示。

算法 3.7　【算法描述】　　【算法步骤】

```
template<class DT>
void DestroyStack(SNode<DT> * &S)                 //销毁链栈 S
{
  while(S)                                        //栈非空
  {
    p=S;                                          //1.取第一个结点
    S=S->next;                                    //2.栈顶指针后移
    delete p;                                     //3.释放原第一个结点
  }
}
```

3. 入栈

【算法思想】 为入栈元素动态分配一个结点空间 p，插入栈顶并修改栈顶指针为 p。

算法描述与算法步骤如算法 3.8 所示。

算法 3.8　【算法描述】　　【算法步骤】

```
template<class DT>
bool Push(SNode<DT> * &S,DT e)                    //插入元素 e 为新的栈顶元素
{
    p=new SNode<DT>;                              //1.生成新结点
    if(!p) return false;                          //创建失败，返回 false
    p->data=e;                                    //2.新结点数据域置为 e
    p->next=S;                                    //将新结点插入栈顶
    S=p;                                          //修改栈顶指针为 p
    return true;                                  //3.返回 true
}
```

4. 出栈

【算法思想】 判断栈是否空，空则返回 false；否则取栈顶元素赋给 e，然后修改栈顶指针并释放原栈顶元素的空间。

算法描述与算法步骤如算法 3.9 所示。

算法 3.9 【算法描述】　　【算法步骤】

```
0  template<class DT>
1  bool Pop(SNode<DT> * &S,DT &e)              //删除 S 栈顶元素,e 返回其值
2  {
3    if(S==NULL)                               //1.栈空的情况,即栈下溢出
4      return false;                           //无法出栈,返回 false
5    p=S;                                      //2.p 暂存栈顶结点
6    e=p->data;                                //栈顶元素赋值给 e
7    S=S->next;                                //修改栈顶指针
8    delete p;                                 //释放原栈顶元素的空间
9    return true;                              //3.返回 true
10 }
```

5. 取栈顶元素

【算法思想】 判断栈是否空，空则返回 false；否则取栈顶元素赋给 e。此操作栈顶指针保持不变。

算法描述与算法步骤如算法 3.10 所示。

算法 3.10 【算法描述】　　【算法步骤】

```
0  template<class DT>
1  bool GetTop(SNode<DT> * S,DT &e)            //取栈顶元素
2  {
3    if(S==NULL)                               //1.栈空的情况,即栈下溢出
4       return false;                          //无法出栈,返回 false
5    p=S;                                      //2.取栈顶元素
6    e=p->data;                                //赋值给 e
7    return true;                              //3.返回 true
8  }
```

3.1.4 顺序栈和链栈的比较

顺序栈和链栈基本操作的实现在时间上都是一致的，都是常数级 $O(1)$。在空间上，初始化一个顺序栈必须先声明一个固定长度，这样在栈不满时，就浪费了一部分存储空间，并且存在栈满溢出的问题；链栈没有栈满的问题，只有当内存没有可用空间时才会出现栈满，但是每个元素都需要一个指针域，从而产生了结构性开销。因此，当栈的使用过程中元素个数变化较大时，用链栈是适宜的，反之应该采用顺序栈。


微课视频

3.1.5 栈的应用

日常生活中栈的应用很常见。例如，洗干净的盘子总是逐个叠放在已经洗好的盘子上面，使用时从上往下逐个取出；在软件使用中，很多软件都有“撤销”或“后退”操作，可以像时间倒退一样，返回到之前的某个操作或某个页面。栈的操作特点正是这些实际应用的抽象。

栈是一种非常重要的数据结构，用途十分广泛。在程序设计中，如果需要对数据存取采用“先进后出”的特点，则可以利用栈来实现。在后续二叉树的各种算法中会大量使用栈。下面通过括号匹配和算术表达式求值的例子说明栈的应用。

【应用 3-1】 括号匹配的校验。

在表达式中经常会用到两种括号：圆括号和方括号。不管使用哪种括号，表达式没有问题的一个重要因素就是所使用的括号是否能够匹配上。括号可以嵌套，([()])或([]())等为正确格式，但([)、([]或(()]都不符合要求。

括号匹配要求“就近匹配”，后面的先匹配，一层层由内而外。因此，可以使用栈的“先进后出”原则来校验括号是否匹配。每当读入一个左括号，直接入栈，等待相匹配的同类右括号。每当读入一个右括号，若栈不空且与当前栈顶元素匹配，则将栈顶元素出栈，继续进行比较；否则返回。

【算法思想】 首先初始化一个空栈 S，设置 flag 标志位，用来标志匹配结果以控制循环以及返回结果。1 表示匹配成功，0 表示匹配失败，flag 初值置为 1。然后从左往右扫描表达式，依次读入字符 ch，如果表达式没有扫描结束或 flag 非零，则循环执行下列操作。

若 ch 是左括号“(”或“[”，ch 入栈；若 ch 是右括号“)”，如栈不空且栈顶元素是“(”，则匹配成功，栈顶元素出栈后继续扫描，否则匹配失败，flag 置为 0；若 ch 是右括号“]”，如栈不空且栈顶元素是“[”，则匹配成功，栈顶元素出栈后继续扫描，否则匹配失败，flag 置为 0。

循环结束后，如果栈空并且 flag 的值为 1，则匹配成功，返回 true；否则返回 false。

算法描述与算法步骤如算法 3.11 所示。

算法 3.11　【算法描述】　　【算法步骤】

```
0  bool match(string exp)                          //括号匹配校验
1  {
2    InitStack<char>(S);                           //1.初始化空栈
3    flag=1; i=0;                                  //2.标志匹配结果以控制循环以
4    ch=exp[i++];                                  //及返回结果
5    while(ch!='#'&& flag==1)                      //假设表达式以#结尾
6    {
7      switch(ch)
8      {
9        case '(':
10       case '[':                                 //3.左括号入栈
11         Push(S,ch);
12         break;
13       case ')':                                 //4.右括号入栈
14         GetTop(S,e);
15         if(!StackEmpty(S) && e=='(')            //4.1 栈非空且栈顶是'(',匹
16           Pop(S,x);                             //配正确,出栈
17         else                                    //4.2 栈空或栈顶不是'(',匹
18           flag=0;                               //配失败
```

```
19          break;
20        case ']':
21          GetTop(S,e);
22          if(!StackEmpty(S) && e=='[')          //4.3 栈非空且栈顶是'[',匹
23            Pop(S,x);                           //配正确
24          else                                  //4.4 栈空或栈顶不是'[',匹
25            flag=0;                             //配失败
26          break;
27      }
28      ch=exp[i++];                              //5.继续读入下一个字符
29    }
30    if(StackEmpty(S) && flag)
31        return true;                            //6.匹配成功,返回 true
32    else
33        return false;                           //匹配失败,返回 false
34  }
```

【算法分析】 算法执行过程中需要从头到尾扫描表达式中的每个字符，设表达式对应的字符串长度为 n，则算法的时间复杂度为 $O(n)$。算法运行时所占用的辅助存储空间主要取决于栈 S 的大小，显然栈 S 的空间大小不会超过 n，因此算法的空间复杂度也为 $O(n)$。

【应用 3-2】 算术表达式求值。

表达式求值是数学中的一个基本问题，也是程序设计中的一个基本问题。这里仅讨论简单算术表达式的求值问题。表达式包含数字和符号，表达式中处理的符号包括＋、－、＊、/、(、)。

根据运算符在操作数中的位置，表达式分为 3 种形式：前缀表达式（prefix expression）、中缀表达式（infix expression）和后缀表达式（postfix expression）。中缀表达式就是平常用的标准四则运算表达式，运算符在双目操作数中间且带有括号，如表达式 1＋2＊(7－4)/3。

(1) 中缀表达式求值。表达式求值的一个常用方法是算符优先法，即从左到右扫描表达式，按运算符的优先级高低进行计算。算术四则运算的规则可概括为：从左到右，先括号内后括号外，先乘除后加减。在表达式计算的每一步中，任意两个相邻出现的运算符 θ_1 和 θ_2 存在如下 3 种关系之一。

- $\theta_1 < \theta_2$：θ_1 的优先级低于 θ_2；
- $\theta_1 = \theta_2$：θ_1 的优先级等于 θ_2；
- $\theta_1 > \theta_2$：θ_1 的优先级高于 θ_2。

图 3-7 定义了运算符的优先级关系。

这里假设所求表达式不会出现语法错误，即不考虑 error 的情况。

为实现算符优先算法，设置两个工作栈：运算符栈 OP，用于存放暂不进行运算的运算符；操作数栈 OD，用于存放操作数或运算结果。一个运算符是否进行运算取决于其后出现的运算符，对应的操作分为 3 种：一是直接入栈（$\theta_1 < \theta_2$）；二是直接出栈（$\theta_1 = \theta_2$）；三

θ_1 \ θ_2	+	-	*	/	(	)	#
+	>	>	<	<	<	>	>
-	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	=	error
)	>	>	>	>	error	>	>
#	<	<	<	<	<	error	=

图 3-7　运算符的优先级关系

是将当前栈顶符号出栈($\theta_1 > \theta_2$)并计算，然后根据新的栈顶符号与当前符号的优先级关系重复操作类型的判断。

【算法思想】　初始化 OP 栈和 OD 栈，将表达式起始符“=”入 OP 栈。扫描表达式，读入第一个字符 ch。当读到的字符不是表达式结束符“=”或 OP 的栈顶元素不是“=”时，循环执行以下操作。

① 若 ch 是操作数，则入 OD 栈，读入下一个字符 ch；

② 若 ch 是运算符，则根据 OP 的栈顶元素(θ_1)和 ch(θ_2)的优先级比较结果，进行相应处理。

- 如果 $\theta_1 < \theta_2$，则 ch 入 OP 栈，读入下一个字符 ch；
- 如果 $\theta_1 = \theta_2$，则 OP 栈顶元素为'('且 ch 为')'时，将 OP 栈顶元素弹出，相当于括号匹配成功，消去括号，读入下一个字符 ch；
- 如果 $\theta_1 > \theta_2$，则弹出 OP 栈顶元素，并且从 OD 栈弹出两个操作数，进行相应运算并将运算结果入 OD 栈。

最后操作数栈的栈顶元素即为表达式求值的结果，输出即可。

【例 3-3】　给出中缀表达式 1+2*(7-4)/3 计算中栈的变化。

【解】　根据上述步骤，对表达式 1+2*(7-4)/3 的计算过程如表 3-3 所示。

表 3-3　1+2*(7-4)/3 计算过程中栈的变化

步骤	OP 栈	OD 栈	读 入 字 符	主 要 操 作
1				Push(**OP**,'=')
2	=		**1**+2*(7-4)/3=	Push(**OD**,'1')
3	=	1	1**+**2*(7-4)/3=	Push(**OP**,'+')
4	=+	1	1+**2***(7-4)/3=	Push(**OD**,'2')
5	=+	1 2	1+2*****(7-4)/3=	Push(**OP**,'*')
6	=+*	1 2	1+2*(7-4)/3=	Push(**OP**,'(')
7	=+*(	1 2	1+2*(**7**-4)/3=	Push(**OD**,'7')
8	=+*(	1 2 7	1+2*(7**-**4)/3=	Push(**OP**,'-')

续表

步骤	OP栈	OD栈	读入字符	主要操作
9	=＋＊(－	1 2 7	1＋2＊(7－**4**)/3=	Push(**OD**,'4')
10	=＋＊(－	1 2 7 4	1＋2＊(7－4**)**/3=	Pop(**OP**,'－'),Pop(**OD**,**4**),Pop(**OD**,**7**),Push(**OD**,7－4)
11	=＋＊(	1 2 3	1＋2＊(7－4**)**/3=	Pop(**OP**,'(')
12	=＋＊	1 2 3	1＋2＊(7－4)**/**3=	Pop(**OP**,'＊'),Pop (**OD**,**3**),Pop(**OD**,**2**),Push(**OD**,2＊3)
13	=＋	1 6	1＋2＊(7－4)**/**3=	Push(**OP**,'/')
14	=＋/	1 6	1＋2＊(7－4)/**3**=	Push(**OD**,'3')
15	=＋/	1 6 3	1＋2＊(7－4)/3**=**	Pop(**OP**,'/'),Pop(**OD**,3),Pop (**OD**,**6**),Push(**OD**,6/3)
16	=＋	1 2	1＋2＊(7－4)/3**=**	Pop(**OP**,'＋'),Pop (**OD**,**2**),Pop(**OD**,**1**),Push(**OD**,1＋2)
17	=	3	1＋2＊(7－4)/3**=**	PoP(**OP**, '=')

算法描述与算法步骤如算法3.12所示。

算法3.12 【算法描述】 【算法步骤】

```
0  float valExp(char * exp)                    //表达式求值
1  {
2    InitStack<char>(OP);                      //1.初始化 OP 栈
3    InitStack<int>(OD);                       //2.初始化 OD 栈
4    Push(OP,'=');                             //3.表达式起始符"="入 OP 栈
5    ch= * exp++;                              //4.输入表达式,边输入边处理
6    GetTop(OP,e);                             //5.获取 OP 栈顶元素
7    while(ch!='='||e!='=')                    //6.表达式没有扫描完或 OP 的栈顶元
8    {                                         //素不是'='
9      if(!In(ch))                             //6.1 ch 不是运算符
10     {
11       Push(OD,ch);                          //6.1.1 入 OD 栈
12       ch= * exp++;   }                      //6.1.2 读入下一个字符
13     else                                    //6.2 ch 是运算符
14       switch(Precede(e,ch))                 //比较 OP 栈顶元素和 ch 的优先级
15       {
16         case '<':                           //6.2.1 栈顶运算符级别低
17           Push(OP,ch);                      //ch 入 OP 栈
18           ch= * exp++;                      //读入下一个字符 ch
19           break;
20         case '=':                           //6.2.2 优先级相等
21           Pop(OP,x);                        //出栈
22           ch= * exp++;                      //读入下一个字符 ch
```

```
23            break;
24          case '>':                          //6.2.3 栈顶运算符优先级高
25            Pop(OP,x);                       //6.2.3.1 运算符出栈
26            Pop(OD,b);                       //6.2.3.2 弹出 OD 栈顶两个操作数
27            Pop(OD,a);                       //6.2.3.3 运算并将运算结果入 OD 栈
28            Push(OD,Operate(a,x,b));
29            break;
30          }
31          GetTop(OP,e);                      //6.2.3.4 获取栈顶运算符
32      }
33      GetTop(OD,result);                     //7.OD 栈顶元素为表达式求值的结果
34      return result;
35  }
```

【算法分析】 同样地，算法执行过程中需要从头到尾扫描表达式中的每个字符，若表达式对应的字符串长度为 n，则算法的时间复杂度为 $O(n)$。算法运行时所占用的辅助存储空间主要取决于 OP 栈和 OD 栈的大小，显然它们的空间大小之和不会超过 n，所以算法的空间复杂度也为 $O(n)$。

(2) 中缀表达式转换为后缀表达式。后缀表达式是一种把所有运算符都放在操作数后面的式子，因此被称为后缀表达式，这样就解决了运算优先级和括号的问题。计算机在计算一个标准四则运算表达式时，一般都是先将中缀表达式转换为后缀表达式，然后进行计算，如将中缀表达式 1+2*(7−4)/3 转换为后缀表达式 1 2 7 4 − * 3 /+。

这是因为计算机处理后缀表达式求值问题是比较方便的。首先扫描后缀表达式，将遇到的操作数暂存于一个操作数栈中，凡是遇到运算符，便从栈中弹出两个操作数并将运算结果存于操作数栈中，直到对后缀表达式中最后一个操作数处理完，最后压入栈中的数就是最后表达式的计算结果。

利用运算符的优先级，可以把中缀表达式转换为后缀表达式，其过程如下。

Step 1. 创建一个运算符栈，结束符入栈。

Step 2. 从左到右扫描读取表达式，执行下列运算，直至表达式结束符。

2.1　如果是操作数，则直接输出，读入下一个字符。

2.2　如果是运算符 θ_2，则把运算符栈栈顶运算符 θ_1 与 θ_2 进行比较。

2.2.1　若 $\theta_1<\theta_2$，则 θ_2 入运算符栈，读入下一个字符。

2.2.2　若 $\theta_1=\theta_2$，则退栈不输出；若退出的是右括号“)”，则读入下一个字符。

2.2.3　若 $\theta_1>\theta_2$，则退栈并输出。

这里给出一个手动将中缀表达式转换为后缀表达式的方法，如表 3-4 所示。

表 3-4　中缀表达式转换为后缀表达式

步　骤	方 法 描 述	示 例 说 明
1	写出中缀表达式	1+2*(7−4)/3

续表

步　骤	方 法 描 述	示 例 说 明
2	按运算先后把每一次运算用括号括起	(1＋((2＊(7－4))/3))
3	把运算符移至对应的括号的后面	(1((2(7 4)－) ＊ 3)/)＋
4	去除括号	1 2 7 4 － ＊ 3 /＋

（3）后缀表达式求值。将中缀表达式转换成对应的后缀表达式后，对表达式求值时不需要再考虑运算符的优先级，只需要从左到右扫描一遍后缀表达式即可。后缀表达式的求值过程为：从左到右读入后缀表达式，若读入的是一个操作数，则将其入操作数栈；若读入的是一个运算符 θ，则从操作数栈中连续出栈两个元素 a、b（两个操作数），进行运算 bθa 并把运算结果入操作数栈。重复上述过程直到表达式结束，操作数栈的栈顶元素即为该后缀表达式的计算结果。

算法描述与算法步骤如算法 3.13 所示。

算法 3.13　【算法描述】　　【算法步骤】

```
0  float valPostExp(char *postexp)          //后缀表达式求值
1  {
2    InitStack(OD);                         //1.初始化栈
3    ch=*postexp++;                         //2. 从左到右读入后缀表达式
4    while(ch!='#')
5    {
6      if(ch 是操作数)                      //2.1 操作数,入栈
7        Push(OD,ch);
8      else
9        if(ch 是运算符)                    //2.2 运算符
10       {
11         Pop(OD,a);Pop(OD,b);             //2.2.1 出栈两个元素
12         Push(OD,Operate(b,ch,a));        //2.2.2 运算后的结果入栈
13       }
14     ch=*postexp++;                       //2.2.3 读入表达式
15   }
16   GetTop(OD,result);                     //3.栈顶元素为后缀表达式的值
17   return result;
18 }
```

例如，后缀表达式 1 2 7 4 － ＊ 3 /＋的求值过程如表 3-5 所示。

表 3-5　后缀表达式 1 2 7 4 － ＊ 3 /＋的求值过程

步　骤	操 作 数 栈	说　明
1	1	1 进栈
2	1 2	2 进栈
3	1 2 7	7 进栈

续表

步　骤	操 作 数 栈	说　明
4	1 2 7 4	4 进栈
5	1 2	遇－,4、7 出栈
6	1 2 3	7－4＝3,结果入栈
7	1	遇 * ,3、2 出栈
8	1 6	2 * 3＝6,6 入栈
9	1 6 3	3 入栈
10	1	遇 /,3、6 出栈
11	1 2	6 / 3＝2,2 入栈
12		遇＋,2、1 出栈
13	3	1＋2＝3,3 入栈
14	3	扫描结束

最后求得后缀表达式 1 2 7 4 － * 3 /＋的结果为 3,与用中缀表达式求得的结果一致,显然后缀表达式的求值要简单得多。

3.2　队列


微课视频

3.2.1　队列的定义和特点

队列(queue)和栈一样,也是一种操作受限的线性表。但与栈不同的是,队列是一种**先进先出**的线性表。它只允许在一端进行插入操作,而在另一端进行删除操作。这里,允许插入的一端称为**队尾**(rear),允许删除的一端称为**队头**或**队首**(front)。向队列中插入新元素称为**进队**或**入队**(enqueue),新元素入队后就成为新的队尾元素。从队列中删除元素称为**出队**(dequeue)或**离队**,元素出队后,其直接后继元素就成为新的队首元素。队列结构如图 3-8 所示。

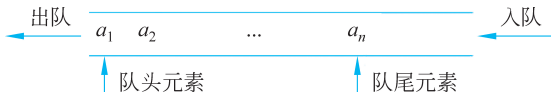
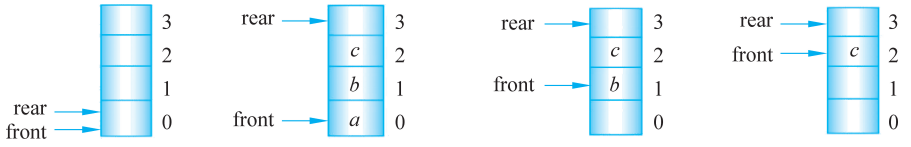


图 3-8　队列结构示意图

图 3-9 是一个队列的操作示意图,图中 front 指向队首元素位置,rear 指向队尾元素的下一个位置。图 3-9(a)表示一个空队;图 3-9(b)表示插入 3 个元素之后队列的状态;图 3-9(c)表示进行一次出队操作之后队列的状态;图 3-9(d)表示再出队一次之后队列的状态。

队列的操作与栈的操作类似,不同的是插入数据只能在队尾进行,删除数据只能在队头进行。下面是队列的抽象数据类型 Queue 的定义。



(a) 空队　(b) 元素a、b、c入队　(c) 出队一次　(d) 再出队一次

图 3-9　队列的操作示意图

```
ADT Queue {
数据对象：D={ai | ai∈ElementSet, i=1, 2, …, n, n≥0}。
数据关系：R={<ai-1, ai> | ai-1, ai∈D, i=2, …, n}，约定 an 端为队尾，a1 端为队头。
基本操作：
    InitQueue(&Q)                          //初始化队列
        操作功能：创建一个空的队列 Q。
        操作输出：创建成功，返回 true；不成功，返回 false。
    DestroyQueue(&Q)                       //销毁队列
        操作功能：释放队列 Q 所占的存储空间。
        操作输出：无。
    GetHead(Q)                             //取队头元素
        操作功能：读取队列 Q 的队头元素。
        操作输出：如果队列不空，输出队头元素。
    EnQueue(&Q,e)                          //入队
        操作功能：在队尾插入新元素 e。
        操作输出：插入元素 e 为队列 Q 新的队尾元素。
    DeQueue(&Q,&e)                         //出队
        操作功能：删除队头元素。
        操作输出：删除队列 Q 的队头元素并用 e 返回其值。
    QueueLength(Q)                         //测队长
        操作功能：求队列 Q 的长度。
        操作输出：返回队列 Q 的元素个数。
    QueueEmpty(Q)                          //判队空
        操作功能：判断队列 Q 是否为空。
        操作输出：如果 Q 是空队，返回 true；否则，返回 false。
    ClearQueue(&Q)                         //清空队列
        操作功能：删除队列 Q 中的所有元素。
        操作输出：队列 Q 被置为空队。
}ADT Queue
```

【例 3-4】　若元素入队顺序为 1234，能否得到 3142 的出队顺序？

【解】　入队顺序为 1234，那么由队列的先进先出原则，出队顺序也为 1234。因此不能得到 3142 的出队顺序。


微课视频

3.2.2　循环队列

队列作为一种特殊线性表，同样也存在顺序存储和链式存储两种方式。首先了解队列的顺序存储结构——顺序队列。

和顺序栈类似，**顺序队列**(sequential queue)**是利用一组连续的存储单元存放从队头至队尾的数据元素**。为操作方便，附设两个整型变量 front 和 rear，分别指示队头和队尾元素的位置。顺序队列定义如下。

```
template<class DT>
struct SqQueue
{ DT * base;          //存储空间基地址
  int front;          //队头指针,指向队首元素①
  int rear;           //队尾指针,指向队尾元素的后面
  int queuesize;      //队列容量
}
```

由上述定义可知下列语句的含义。

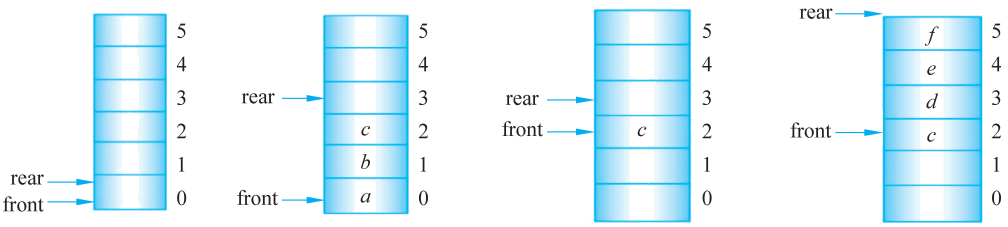
SqQueue＜DT＞ Q;表示声明一个顺序队列变量 Q。

Q.front 表示队头指针,Q.base[Q.front]表示队首元素,队不空时,即出队该元素。

Q.rear 表示队尾指针,如果元素入队,队不满时,存入 Q.base[Q.rear]处。

这里约定初始化创建空队列时,Q.front＝Q.rear＝0;元素入队时,若队列不满,新入队元素送入 Q.rear 所指单元,队尾指针 Q.rear 增 1;元素出队时,若队列不空,从 Q.front 所指单元取出队头元素,队头指针 Q.front 增 1(注意,这与现实生活中队列的队头元素出队操作不同)。因此,在非空队列中,队头指针始终指向队首元素位置,队尾指针始终指向队尾元素的下一个位置。

顺序队列结构及操作示意图如图 3-10 所示。



(a) 空队　(b) a、b、c入队　(c) a、b出队　(d) d、e、f入队

图 3-10　顺序队列结构及操作示意图

在图 3-10(d)中,由于当前队列分配的最大存储空间为 6,如有新元素插入,则无法继续入队,会产生数组越界引起程序非法操作的错误,此现象称为溢出。而实际上,此时队列中还有可用空闲空间,因此这种现象称为假溢出。

为解决“假溢出”现象,使得数组中的存储空间可以充分利用起来,一种比较巧妙的方法是假设存储队列的连续存储空间是头尾相接的环状结构,形成一个环形的顺序表,称为**循环队列**(circular queue)。

在循环队列中,头、尾指针以及队列元素之间的关系不变,只是头、尾指针“依环增 1”的操作需要通过模运算来实现。通过取模,循环队列的头指针和尾指针可以在顺序表空间内以头尾相接的方式“循环”移动。循环队列头、尾指针的调整方法如下。

- 队尾指针增 1:Q.rear＝(Q.rear＋1)％Q.queuesize。
- 队头指针增 1:Q.front＝(Q.front＋1)％Q.queuesize。

① 也有将队头元素指向队首元素前和将队尾元素指向队尾元素的,那出队元素与入队元素在位置上是有区别的。

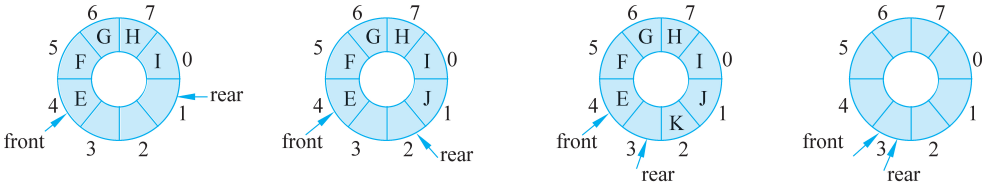
如图 3-10(d)所示，在元素 f 入队前，rear 的值为 5。当元素 f 入队后，通过模运算，Q.rear=(Q.rear+1)%6，因此得到 rear 的值为 0，不会出现“假溢出”现象。若此后有元素 g 和 h 相继入队，则队列空间满，此时 Q.front=Q.rear。

如图 3-10(c)所示，元素 c 出队，则出现队空的状态，此时 Q.front=Q.rear。

由此可见，引入循环队列后，出现了队空与队满的条件一样的情况。那么，如何区分某一状态是队空还是队满呢？为解决这一问题，采用的方法之一是牺牲一个存储单元，即当队尾指针加 1 等于队首指针时判定为队满。因此，本书约定循环队列中队空和队满的条件如下。

- 队空：Q.front==Q.rear。
- 队满：(Q.rear+1)%Q.queuesize==Q.front。

循环队列结构及操作示意图如图 3-11 所示。



(a) 一般结构　(b) J 入队后　(c) 队满　(d) 队空

图 3-11　循环队列结构及操作示意图

类似地，根据 ADT Queue 中的基本操作定义设置操作参数类型和返回值后，循环队列的基本操作的函数定义简述如表 3-6 所示。

表 3-6　循环队列的基本操作的函数定义

序号	函数定义	功能说明
1	//初始化队列 bool InitQueue(SqQueue<DT> &Q,int m)	创建容量为 m 的空队列 Q；创建成功，返回 true；否则，结束运行
2	//销毁队列 void DestroyQueue(SqQueue<DT> &Q)	释放队列 Q 所占内存空间
3	//取队头元素 bool GetHead(SqQueue<DT> Q,DT &e)	队列 Q 空，返回 false；否则取队头元素赋给 e，返回 true
4	//入队 bool EnQueue(SqQueue<DT> &Q,DT e)	队列 Q 满，返回 false；否则在队尾插入一个值为 e 的元素，返回 true
5	//出队 bool DeQueue(SqQueue<DT> &Q,DT &e)	队列 Q 空，返回 false；否则删除队头元素，返回 true
6	//测队长 int QueueLength(SqQueue<DT> Q)	求队列 Q 的长度，即队中元素个数
7	//判队空 bool QueueEmpty(SqQueue<DT> Q)	判断队列 Q 是否为空。如果是空队列，返回 true；否则，返回 false
8	//清空队 void ClearQueue(SqQueue<DT> &Q)	把队列 Q 变成空队列

循环队列的类型定义与前面给出的顺序队列的类型定义相同。循环队列部分基本操