第3章 栈和队列

CHAPTER

栈和队列是两种常用的数据结构，广泛应用在操作系统、编译程序等各种软件系统中。从数据结构角度看，栈和队列是操作受限的线性表，栈和队列的数据元素具有单一的前驱和后继的线性关系；从抽象数据类型角度看，栈和队列又是两种重要的抽象数据类型。

【学习重点】

① 栈和队列的操作特性；

② 基于顺序栈和链栈的基本操作的实现；

③ 基于循环队列和链队列的基本操作的实现。

【学习难点】

① 两栈共享空间的实现；

② 循环队列的组织及队空和队满的判定条件。

3.1 栈

3.1.1 栈的逻辑结构

1. 栈的定义

栈(stack)是限定仅在表尾进行插入和删除操作的线性表，允许插入和删除的一端称为栈顶，另一端称为栈底，不含任何数据元素的栈称为空栈。

如图 3-1 所示，栈中有三个元素，插入元素(也称进栈、压栈)的顺序是 a_1、a_2、a_3，当需要删除元素(也称出栈、弹栈)时只能删除 a_3。换言之，在任何时候出栈的元素都只能是栈顶元素，即最后入栈者最先出栈。所以栈中元素除了具有线性

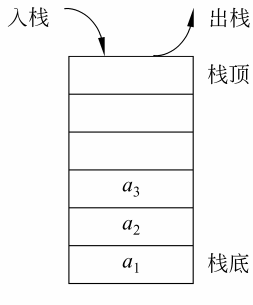


图 3-1 栈的示意图

关系外，还具有**后进先出**(last in first out)[①]的特性。

在日常生活中，有很多栈的例子。例如，一叠摞在一起的盘子，要从这叠盘子中取出或放入一个盘子，只有在其顶部操作才是最方便的。早在计算机出现之前，会计就使用栈来记账；火车扳道站、单车道死胡同等也使用栈。在程序设计语言中，也有很多栈应用的例子，例如，在对高级语言编写的源程序进行编译时类似于表达式括号匹配问题就是用栈来解决的；计算机系统在处理子程序之间的调用关系时，用栈来保存处理执行过程中的调用次序，等等。

2. 栈的抽象数据类型定义

虽然对插入和删除操作的位置限制减少了栈操作的灵活性，但同时也使得栈的操作更有效更容易实现。其抽象数据类型定义为：

```
ADT Stack
Data
  栈中元素具有相同类型及后进先出特性，相邻元素具有前驱和后继关系
Operation
  InitStack
      前置条件：栈不存在
      输入：无
      功能：栈的初始化
      输出：无
      后置条件：构造一个空栈
  DestroyStack
      前置条件：栈已存在
      输入：无
      功能：销毁栈
      输出：无
      后置条件：释放栈所占用的存储空间
  Push
      前置条件：栈已存在
      输入：元素值 x
      功能：入栈操作，在栈顶插入一个元素 x
      输出：如果插入不成功，则抛出异常
      后置条件：如果插入成功，则栈顶增加了一个元素
  Pop
      前置条件：栈已存在
      输入：无
      功能：出栈操作，删除栈顶元素
      输出：如果删除成功，返回被删元素值，否则，抛出异常
      后置条件：如果删除成功，则栈顶减少了一个元素
```

① 需要注意的是，栈只是对线性表的插入和删除操作的位置进行了限制，并没有限定插入和删除操作进行的时间，也就是说，出栈可随时进行，只要某个元素位于栈顶就可以出栈。例如，假定有三个元素 a、b、c 按 a、b、c 的次序依次进栈，且每个元素只允许进一次栈，则可能的出栈序列有 abc、acb、bac、bca、cba 五种。

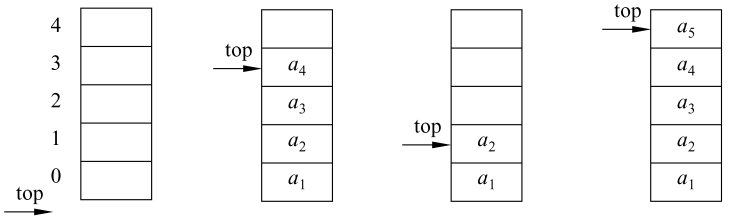
```
GetTop
    前置条件：栈已存在
    输入：无
    功能：取栈顶元素，读取当前的栈顶元素
    输出：若栈不空，返回当前的栈顶元素值
    后置条件：栈不变
Empty
    前置条件：栈已存在
    输入：无
    功能：判空操作，判断栈是否为空
    输出：如果栈为空，返回 1，否则返回 0
    后置条件：栈不变
endADT
```

3.1.2　栈的顺序存储结构及实现

1. 栈的顺序存储结构——顺序栈

栈的顺序存储结构称为**顺序栈**(sequential stack)。

顺序栈本质上是顺序表的简化，唯一需要确定的是用数组的哪一端表示栈底。通常把数组中下标为 0 的一端作为栈底，同时附设指针 top 指示栈顶元素在数组中的位置[①]。设存储栈元素的数组长度为 StackSize，则栈空时栈顶指针 top＝－1；栈满时栈顶指针 top＝StackSize－1。入栈时，栈顶指针 top 加 1；出栈时，栈顶指针 top 减 1。图 3-2 是栈操作的示意图。



(a) top=–1栈空　(b) $a_1a_2a_3a_4$依次入栈　(c) a_4a_3依次出栈　(d)top=4栈满

图 3-2　栈的操作示意图

2. 顺序栈的实现

将栈的抽象数据类型定义在顺序栈存储结构下用C++ 中的类实现。由于栈元素的数据类型不确定，因此采用C++ 的模板机制。

```
const int StackSize=10;              //10 只是示例性的数据，可以根据实际问题具体定义
template <class DataType>            //定义模板类 SeqStack
class SeqStack
{
```

① 在有些参考书中，将 top 指向栈中的第一个空闲位置，如果这样的话，栈空应该表示为 top＝0。

```
public:
    SeqStack(){top=-1;}                    //构造函数,初始化一个空栈
    ~SeqStack(){}                          //析构函数为空
    void Push(DataType x);                 //入栈操作,将元素 x 入栈
    DataType Pop();                        //出栈操作,将栈顶元素弹出
    DataType GetTop(){if(top!=-1) return data[top];}      //取栈顶元素(并不删除)
    int Empty(){top==-1?return 1: return 0;}              //判断栈是否为空
private:
    DataType data[StackSize];              //存放栈元素的数组
    int top;                               //栈顶指针,为栈顶元素在数组中的下标
};
```

根据顺序栈的操作定义，很容易写出顺序栈的基本操作的算法，并且其时间复杂度均为 $O(1)$。

(1) 栈的初始化

初始化一个空栈只需将栈顶指针 top 置为－1。

(2) 入栈操作

在栈中插入一个元素 x 只需将栈顶指针 top 加 1，然后在 top 指向的位置填入元素 x，算法如下：

顺序栈入栈算法 Push

```
template <class DataType>
void SeqStack<DataType>::Push(DataType x)
{
  if(top==StackSize-1)throw "上溢";
  data[++top]=x;
}
```

(3) 出栈操作

删除栈顶元素只需取出栈顶元素，然后将栈顶指针 top 减 1，算法如下：

顺序栈出栈算法 Pop

```
template <class DataType>
DataType SeqStack<DataType>::Pop()
{
  if(top==-1)throw "下溢";
  x=data[top--];
  return x;
}
```

(4) 取栈顶元素

取栈顶元素只是将 top 指向的栈顶元素取出，并不修改栈顶指针。

(5) 判空操作

顺序栈的判空操作只需判断 top==-1 是否成立，如果成立，则栈为空，返回 1；如果不成立，则栈非空，返回 0。

3. 两栈共享空间

在一个程序中如果需要同时使用具有相同数据类型的两个栈时，最直接的方法是为每个栈开辟一个数组空间，不过这样做的结果可能出现一个栈的空间已被占满而无法再进行插入操作，同时另一个栈的空间仍有大量剩余而没有得到利用的情况，从而造成存储空间的浪费。一种可取的方法是充分利用顺序栈单向延伸的特性，使用一个数组来存储两个栈，让一个栈的栈底为该数组的始端，另一个栈的栈底为该数组的末端，每个栈从各自的端点向中间延伸，如图 3-3 所示。

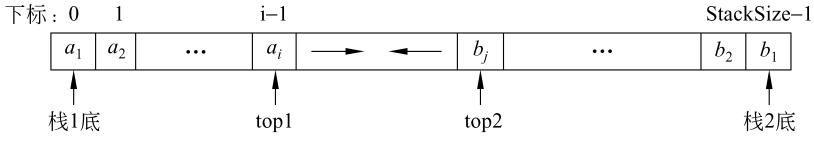


图 3-3　两栈共享空间示意图

其中，top1 和 top2 分别为栈 1 和栈 2 的栈顶指针，StackSize 为整个数组空间的大小，栈 1 的底固定在下标为 0 的一端；栈 2 的底固定在下标为 StackSize−1 的一端。

两栈共用一个数组空间的抽象数据类型可定义为：

```
ADT BothStack
Data
  共享的数组空间长度 StackSize
  存放栈元素的数组 data[StackSize]
  栈 1 的栈顶指针 top1
  栈 2 的栈顶指针 top2
Operation
  BothStack
    前置条件：共享数组空间不存在
    输入：无
    功能：创建两栈共享的数组空间
    输出：无
    后置条件：两栈均为空，top1 等于-1，top2 等于 StackSize
  ~BothStack
    前置条件：共享空间已存在
    输入：无
    功能：销毁两栈共享的数组空间
    输出：无
    后置条件：将两栈共享的数组空间释放
  GetTop
    前置条件：共享空间已存在
    输入：栈号 i
```

```
    功能：读取栈 i 当前的栈顶元素
    输出：若栈 i 不空,返回栈 i 当前的栈顶元素值
    后置条件：两栈均不变
  Push
    前置条件：共享空间已存在
    输入：栈号 i,元素值 x
    功能：入栈操作,在栈 i 插入一个元素 x
    输出：若插入不成功,则抛出插入异常
    后置条件：若插入成功,则栈 i 插入了一个栈顶元素
  Pop
    前置条件：共享空间已存在
    输入：栈号 i
    功能：出栈操作,在栈 i 中删除栈顶元素
    输出：若删除不成功,则抛出删除异常
    后置条件：若删除成功,则栈 i 中删除了栈顶元素
  Empty
    前置条件：共享空间已存在
    输入：栈号 i
    功能：判空操作,判断栈 i 是否为空栈
    输出：若栈 i 为空栈,返回 1;否则返回 0
    后置条件：两栈均不变
endADT
```

对应的C++类声明如下：

```
const int StackSize=100;                          //100只是示例数据,需根据具体问题定义
template <class DataType>
class BothStack
{
public:
    BothStack(){top1=-1; top2=StackSize;}   //构造函数,将两个栈分别初始化
    ~BothStack(){}                                //析构函数为空
    void Push(int i,DataType x);                  //入栈操作,将元素 x 压入栈 i
    DataType Pop(int i);                          //出栈操作,对栈 i 执行出栈操作
    DataType GetTop(int i);                       //取栈 i 的栈顶元素
    int Empty(int i);                             //判断栈 i 是否为空栈
private:
    DataType data[StackSize];                     //存放两个栈的数组
    int top1,top2;              //两个栈的栈顶指针,分别为各自栈顶元素在数组中的下标
};
```

设i表示整型数值,它只取1和2两个值。当i=1时,表示对栈1操作;当i=2时,表示对栈2操作。下面讨论两栈共享空间的入栈和出栈操作。

(1) 入栈操作

当存储栈的数组中没有空闲单元时为栈满,此时栈1的栈顶元素和栈2的栈顶元素

位于数组中的相邻位置，即 top1＝top2－1(或 top2＝top1＋1)。另外，当新元素插入栈 2 时，栈顶指针 top2 不是加 1 而是减 1，算法如下：

两栈共享空间入栈算法 Push

```
template <class DataType>
void BothStack<DataType>::Push(int i,DataType x)
{
    if(top1==top2-1)throw "上溢";          //判断是否栈满
    if(i==1)data[++top1]=x;                //在栈 1 插入
    if(i==2)data[--top2]=x;                //在栈 2 插入
}
```

(2) 出栈操作

当 top1＝－1 时栈 1 为空；当 top2＝StackSize 时栈 2 为空。另外，当从栈 2 删除元素时，栈顶指针 top2 不是减 1 而是加 1，算法如下：

两栈共享空间出栈算法 Pop

```
template <class DataType>
DataType BothStack<DataType>::Pop(int i)
{
    if(i==1){                                  //在栈 1 删除
      if(top1==-1)throw "下溢";                //判断栈 1 是否为空
      return data[top1--];
    }
    if(i==2){                                  //在栈 2 删除
      if(top2==StackSize)throw "下溢";         //判断栈 2 是否为空
      return data[top2++];
    }
}
```

在两栈共享空间中，由于两个栈相向增长，这样浪费的空间就会减少，同时发生上溢的概率也会减少。但是，只有当两个栈的空间需求有相反的关系时，这种方法才会奏效。也就是说，最好一个栈增长时另一个栈缩短。当三个或三个以上的栈共享一个数组空间时，只有相向增长的栈才能互补余缺，而背向增长或同向增长的栈之间无法自动互补，所以必须对某些栈作整体移动，这将使问题变得复杂而且效果也欠佳。

3.1.3　栈的链接存储结构及实现

1. 栈的链接存储结构——链栈

栈的链接存储结构称为**链栈**(linked stack)。

通常链栈用单链表表示，因此其结点结构与单链表的结点结构相同。因为只能在栈顶执行插入和删除操作，显然以单链表的头部做栈顶是最方便的，而且没有必要像单链表

那样为了运算方便附加一个头结点。通常将链栈表示成如图 3-4 的形式。

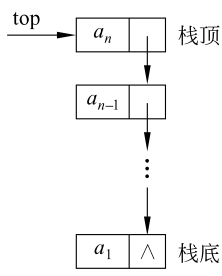


图 3-4 链栈示意图

2. 链栈的实现

链栈的结点结构可以复用单链表的结点结构，参见第 2.3.1 节。将栈的抽象数据类型定义在链栈存储结构下用C++ 中的类实现。

```
template <class DataType>
class LinkStack
{
public:
    LinkStack(){top=NULL;}                    //构造函数,初始化一个空链栈
    ~LinkStack();                             //析构函数,释放链栈中各结点的存储空间
    void Push(DataType x);                    //入栈操作,将元素 x 入栈
    DataType Pop();                           //出栈操作,将栈顶元素出栈
    DataType GetTop(){if(top!=NULL)return top->data;}     //取栈顶元素(并不删除)
    int Empty(){top==NULL?return 1: return 0;}       //判空操作,判断链栈是否为空栈
private:
    Node<DataType> * top;                             //栈顶指针即链栈的头指针
};
```

链栈基本操作的实现本质上是单链表基本操作的简化，并且除析构函数外，算法的时间复杂度均为 $O(1)$。

(1) 构造函数

构造函数的作用是初始化一个空链栈，由于链栈不带头结点，因此只需将栈顶指针 top 置为空。

(2) 入栈操作

链栈的插入操作只需处理栈顶即第一个位置的情况，而无需考虑其他位置的情况，其操作示意图如图 3-5 所示，算法如下：

链栈入栈算法 Push

```
template <class DataType>
void LinkStack<DataType>::Push(DataType x)
{
  s=new Node;s->data=x;              //申请一个数据域为 x 的结点 s
  s->next=top;top=s;                 //将结点 s 插在栈顶
}
```

(3) 出栈操作

链栈的删除操作只需处理栈顶即第一个位置的情况，而无需考虑其他位置的情况，其操作示意图如图 3-6 所示，算法如下：

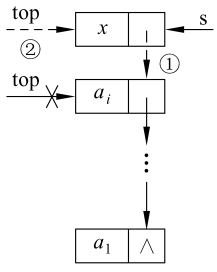


图 3-5　链栈插入操作示意图

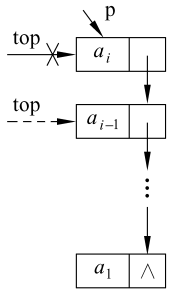


图 3-6　链栈删除操作示意图

链栈出栈算法 Pop

```
template <class DataType>
DataType LinkStack<DataType>::Pop()
{
  if(top==NULL)throw "下溢";
  x=top->data;p=top;                          //暂存栈顶元素
  top=top->next;                              //将栈顶结点摘链
  delete p;
  return x;
}
```

(4) 取栈顶元素

取栈顶元素只需返回栈顶指针 top 所指结点的数据域。

(5) 判空操作

链栈的判空操作只需判断 top==NULL 是否成立。如果成立,则栈为空,返回 1;如果不成立,则栈非空,返回 0。

(6) 析构函数

链栈的析构函数需要将链栈中所有结点的存储空间释放,算法与单链表类的析构函数类似,算法略。

3.1.4　顺序栈和链栈的比较

实现顺序栈和链栈的所有基本操作的算法都只需要常数时间,因此唯一可以比较的是空间性能。初始时顺序栈必须确定一个固定的长度,所以有存储元素个数的限制和空间浪费的问题。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。所以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。

3.2 队　　列

3.2.1 队列的逻辑结构

1. 队列的定义

队列(queue)是只允许在一端进行插入操作，在另一端进行删除操作的线性表。允许插入(也称入队、进队)的一端称为队尾，允许删除(也称出队)的一端称为队头。图3-7所示是一个有5个元素的队列，入队的顺序为a_1、a_2、a_3、a_4、a_5，出队的顺序依然是a_1、a_2、a_3、a_4、a_5，即最先入队者最先出队。所以队列中的元素除了具有线性关系外，还具有**先进先出**(first in first out)的特性。

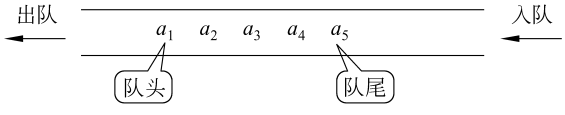


图3-7　队列的示意图

现实世界中有许多问题可以用队列描述。例如，对顾客服务部门(例如银行)的工作往往是按队列方式进行的，这类系统称作排队系统。在程序设计中，也经常使用队列记录需按先进先出方式处理的数据，例如键盘缓冲区、操作系统中的作业调度等。

2. 队列的抽象数据类型定义

```
ADT Queue
Data
  队列中的元素具有相同类型及先进先出特性，相邻元素具有前驱和后继关系
Operation
  InitQueue
    前置条件：队列不存在
    输入：无
    功能：初始化队列
    输出：无
    后置条件：创建一个空队列
  DestroyQueue
    前置条件：队列已存在
    输入：无
    功能：销毁队列
    输出：无
    后置条件：释放队列所占用的存储空间
  EnQueue
    前置条件：队列已存在
    输入：元素值 x
    功能：入队操作，在队尾插入一个元素
```