第3章 栈和队列

栈和队列是两种重要的数据结构，在操作系统、编译程序等各种软件系统中都有应用。从数据结构角度看，栈和队列也是线性表，其基本操作是线性表操作的子集，是操作受限的线性表；从数据类型角度看，栈和队列是和线性表大不相同的两类重要的抽象数据类型，在很多复杂问题的求解中，往往采用栈或队列作为辅助数据结构，如树和图的遍历、关键路径等。

【学习重点】

◆ 栈的定义与特征；

◆ 栈的顺序存储与链接存储结构；

◆ 队列的定义与特征；

◆ 队列的顺序存储与链接存储结构；

◆ 栈和队列的应用。

【学习难点】

◆ 函数的递归调用；

◆ 表达式求值。

3.1 引　　言

你玩过正话反说的游戏吗？我说“新年好”，你要说“好年新”。随着字数的增加难度越来越大，这种倒背如流的能力对人类来说难度比较大，但对于栈来说就非常简单了，栈的特性就是“后进先出”，类似货车或者集装箱装卸货物时需要遵循的“后进先出”规则——最后装进去的货物需要最先卸货。

算术表达式中一般会出现三种括号“()”“[]”“{}”，每种括号都是左括号与对应的右括号匹配。从左到右顺序扫描表达式，当遇到一个右括号时，查找已经遍历过的最后一个尚未配对的左括号，如果与该右括号匹配，则该左右括号匹配成功。对于左括号来说，具有最后遍历的最先匹配的特点，通常用栈这种数据结构描述具有后到先处理特征的问题。当然，栈应用的例子还有很多，如数制转换、函数的调用与递归调用、表达式求值、拓扑排序和迷宫问题求解等。

为了保证公平，在生活中经常需要排队，食堂吃饭要排队、医院挂号要排队、乘坐公交车要排队、登机要排队……队列具有“先到先得”的特性，生活中的这些事情都可以用队列模拟。

队列在计算机系统中的应用也非常广泛，它可以解决主机与外部设备之间速度不匹配

的问题和由多用户引起的资源竞争问题。打印机的打印速度远远小于计算机处理数据的速度,若将数据直接送到打印机,则会导致计算机处理完一批数据就要等待打印机打印。所以为提高计算机的处理效率,设置一个打印数据缓冲区,计算机把要打印的数据依次写入缓冲区,打印机从缓冲区按照先进先出的原则依次取出数据打印,由此打印数据缓冲区中所存储的数据就是一个队列。在一个带有多终端的计算机系统上,多个用户需要 CPU 运行自己的程序,它们分别通过各自终端向操作系统提出占用 CPU 的请求。操作系统通常按照每个请求在时间上的先后顺序,把它们排成一个队列,每次把 CPU 分配给队头请求的用户使用。这样既相对公平地满足了每个用户的请求,又使 CPU 能够正常运行。

3.2 栈

本节主要讨论栈的定义及其实现。

3.2.1 栈的定义

栈本义是存储货物或供旅客住宿的地方,可引申为仓库、中转站,因此引入到计算机领域就是指数据暂时存储的地方。栈(stack)是一种限定仅在表的一端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(stack top),另一端称为栈底(stack bottom),如图 3.1 所示。栈顶的当前位置是动态的,当栈中没有数据时称为空栈,在栈顶插入一个数据元素通常称为进栈或入栈(push),从栈顶删除一个数据元素通常称为出栈、退栈或弹栈(pop)。

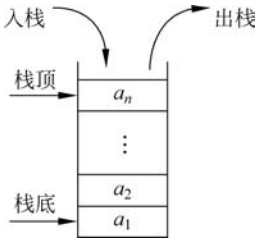


图 3.1 栈的示意图

栈的显著特点是“后进先出”(last in first out,LIFO),即最后进栈的元素最先出栈。每次进栈的元素都放在原栈顶元素之上成为新的栈顶元素,每次出栈的元素都是当前栈顶元素。例如华山的长空栈道,入选全球十大恐怖悬崖步道,是个断头路,必须原路折返,假设栈道上人和人之间不能错身,那么一队人按顺序走入长空栈道,则必须按照其反序走出长空栈道。

栈的抽象数据类型定义如下:

```
ADT 名:Stack
    Data:
        数据对象:D = {a_i | 1 ≤ i ≤ n,n ≥ 0,a_i 是 Element 类型数据}
        数据关系:R = {< a_i,a_i+1 >| 1 ≤ i ≤ n - 1}
    Operation:
        InitStack(&S):初始化栈,构造一个空栈 S。
        DestroyStack (&S):销毁栈,释放栈 S 所占用的存储空间。
        Empty(S):判断栈是否为空,若栈 S 为空,则返回真值;否则返回假。
        Push(&S,x):入栈,将元素 x 插入到栈 S 中作为新栈顶元素。
        Pop(&S,&x):出栈,从栈 S 中删除栈顶元素,并用 x 返回其值。
        GetTop(S,&x):取栈顶元素,用 x 返回当前的栈顶元素。
    End ADT
```

【例 3.1】 用 S 表示进栈操作,用 X 表示出栈操作,若元素的进栈顺序是 1234,为了得

到 1432 的出栈顺序，给出相应的 S 和 X 的操作串。

解：为了得到 1432 的出栈顺序，其操作过程应该是 1 进栈、1 出栈、2 进栈、3 进栈、4 进栈、4 出栈、3 出栈、2 出栈，相应的操作串是 $SXSSSXXX$。

说明：n 个不同的元素通过一个栈可以产生的出栈序列的个数是 $\frac{1}{n+1}C_{2n}^{n}$，如有 3 个元素(a、b、c)时，可能的出栈序列个数为 5，大家可以试试是哪 5 个序列。

【例 3.2】 若栈 $S1$ 中保存整数，栈 $S2$ 中保存运算符，函数 $F()$依次执行下述各步操作：

(1) 从 $S1$ 中依次弹出两个操作数 a 和 b；

(2) 从 $S2$ 中弹出一个运算符 op；

(3) 执行相应的运算 b op a；

(4) 将运算结果压入 $S1$ 中。

假定 $S1$ 中的操作数依次是 5、8、3、2(2 在栈顶)，$S2$ 中的运算符依次是 *、－、＋(＋在栈顶)。调用 3 次 $F()$后，$S1$ 栈顶保存的值是多少？

解：第一次调用：①从 $S1$ 中弹出 2 和 3；②从 $S2$ 中弹出＋；③执行 3＋2＝5；④将运算结果 5 压入 $S1$ 中。第一次调用结束后 $S1$ 中剩余 5、8、5(5 在栈顶)，$S2$ 中剩余 *、－(－在栈顶)。

第二次调用：①从 $S1$ 中弹出 5 和 8；②从 $S2$ 中弹出－；③执行 8－5＝3；④将运算结果 3 压入 $S1$ 中。第二次调用结束后 $S1$ 中剩余 5、3(3 在栈顶)，$S2$ 中剩余 *。

第三次调用：①从 $S1$ 中弹出 3 和 5；②从 $S2$ 中弹出 *；③执行 5 * 3＝15；④将运算结果 15 压入 $S1$ 中。第三次调用结束后 $S1$ 中仅剩余 15(栈顶)，$S2$ 为空。所以，调用 3 次 $F()$后，$S1$ 栈顶保存的值是 15。

3.2.2 栈的顺序存储结构及其实现

栈是一种操作受限的线性表，数据元素之间的关系与线性表相同，所以也可以像线性表一样采用顺序存储结构进行存储，即分配一块连续的存储空间来存放栈中元素，在 C++语言中用数组实现。采用顺序存储结构的栈称为顺序栈(sequential stack)。通常把数组中下标为 0 的一端作为栈底，附设变量 top 指向栈顶元素在数组中的位置。假设栈的元素个数最大不超过正整数 StackSize，则栈空时栈顶位置变量 top＝－1，栈满时 top＝StackSize－1，入栈时 top＋1，出栈时 top－1，如图 3.2 所示。因为这里 top 的值是栈顶元素在数组中的下标，也就是 top 始终指向栈顶元素，所以也将 top 称为栈顶指针，但这与指针不同，请读者注意。

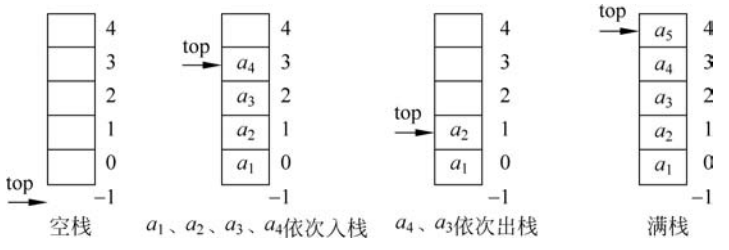


图 3.2 栈的操作

在顺序存储结构下用C++语言中的类实现栈的抽象数据类型定义,见代码3.1。

代码3.1 顺序栈的类定义

```
const int StackSize = 10;          //根据实际问题具体定义
template < typename Element >
class SStack
{
public:
    SStack();                      //构造函数,构造一个空栈
    ~SStack();                     //析构函数,销毁顺序栈
    void push(Element x);          //入栈操作,将元素 x 入栈
    Element pop();                 //出栈操作,将栈顶元素返回
    Element getTop();              //取栈顶元素
    int isEmpty();                 //判断栈是否为空
private:
    Element data[StackSize];       //存放栈元素的数组
    int top;                       //栈顶元素在数组中的下标,即栈顶指针
};
```

下面按照顺序栈的类定义逐个讨论每个成员函数的实现。

1. 构造函数,构造一个空栈

当栈顶指针top=-1时,栈为空;那么构造一个空栈只需将top置-1即可。

2. 析构函数,销毁顺序栈

顺序栈是静态存储分配,在顺序栈变量退出作用域时系统会自动释放顺序栈所占存储空间,因此顺序栈无须销毁,析构函数为空函数。

3. 入栈操作,将元素 x 入栈

该操作首先要判断栈是否已满,若已满则抛出上溢异常;否则先将栈顶指针top+1,再将元素 x 插入在该位置上。顺序栈入栈函数实现见代码3.2。

代码3.2 顺序栈入栈函数

```
template < typename Element >
void SStack < Element > :: push(Element x)
{
    if (top == StackSize - 1) throw "上溢";
    data[++top] = x;
}
```

4. 出栈操作,将栈顶元素返回

该操作首先要判断栈是否为空,若为空则抛出下溢异常;否则先将栈顶指针top所指元素赋给 x,然后top-1,并将 x 返回即可。顺序栈出栈函数实现见代码3.3。

代码3.3 顺序栈出栈函数

```
template < typename Element >
Element SStack < Element > :: pop()
{
```

```
    if (top == -1) throw "下溢";
    x = data[top--];
    return x;
}
```

5. 取栈顶元素

该操作与出栈操作类似，不同之处就是不删除栈顶元素，即栈顶指针 top 不改变。顺序栈取栈顶元素算法示例见代码 3.4。

代码 3.4 顺序栈取栈顶元素

```
template < typename Element >
Element SStack < Element > :: getTop()
{
    if (top == -1) throw "下溢";
    x = data[top];
    return x;
}
```

6. 判断栈是否为空

该操作只需要判断栈顶指针 top 是否为－1 即可。顺序栈判断栈空算法示例见代码 3.5。

代码 3.5 顺序栈判断栈空

```
template < typename Element >
Element SStack < Element > :: isEmpty ()
{
    if (top == -1) return 1;
    else return 0;
}
```

3.2.3 两栈共享空间

对于顺序栈来说，最大的缺点就是必须事先确定栈的存储空间的大小。对于一个栈，只能尽量考虑周全，设计大小适合的数组来处理。但对于两个数据类型相同的栈，可以最大限度地利用事先开辟的存储空间进行操作，这就是两栈共享空间，如图 3.3 所示。数组有两个端点，两个栈有两个栈底，让一个栈的栈底在数组的起始端，即下标 0 处，另一个栈底为数组的末端；两个栈顶指针 top1 和 top2 分别指向两个栈的栈顶元素。也就是说，两个栈在数组的两端，两栈入栈时向中间靠拢，出栈时向两边远离。只要两个栈顶不相遇，两个栈就可以一直使用。

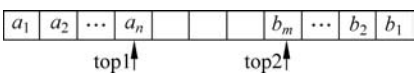


图 3.3 两栈共享空间

假设数组的长度为 MaxSize。那么,栈 1 为空时,top1=-1;栈 2 为空时,top2=MaxSize。那什么时候栈满呢?极端的情况,若栈 2 为空,则栈 1 的栈顶指针 top1=MaxSize-1 时,栈就满了;反之,当栈 1 为空栈时,栈 2 的栈顶指针 top2=0 时,栈就满了。一般情况下,当两个栈顶指针相遇时,栈就满了,即 top1+1=top2 为栈满。

与顺序栈不同,两栈共享空间在实现时,需要定义两个栈顶指针,入栈、出栈、取栈顶元素、判栈空时都需要指定是对哪个栈进行的操作,栈的类定义见代码 3.6。

代码 3.6 两栈共享存储空间的类定义

```
const int MaxSize = 100;
template < typename Element >
class DoubleSStack
{
public:
    DoubleSStack ();
    ~DoubleSStack ();
    void push(Element x, int stackNumber);          //入栈操作,根据栈号将元素 x 入栈
    Element pop(int stackNumber);                   //根据栈号出栈,并返回栈顶元素
    Element getTop(int stackNumber);                //取栈顶元素
    int empty(int stackNumber);                     //判断栈空
private:
    Element data[MaxSize];
    int top1, top2;
}
```

两栈共享空间的构造函数与栈类似,top1=-1,top2= MaxSize 即可。析构函数与栈相同,为空函数。入栈操作时需要判断元素是要入哪个栈,其算法示例见代码 3.7。

代码 3.7 两栈共享存储空间的入栈函数

```
template < typename Element >
void DoubleSStack < Element > :: push(Element x, int stackNumber)
{
    if (top1 + 1 == top2) throw "上溢";
    if(stackNumber == 1)
        data[++top1] = x;
    else if (stackNumber == 2)
        data[ --top2] = x;
}
```

算法中第一个 if 语句判断栈是否已满,后面的++top1 和--top2 就不用担心溢出的问题了。

对于两栈共享空间的出栈操作,只需要增加判断是栈 1 或栈 2 的参数 stackNumber,算法示例见代码 3.8。

代码 3.8 两栈共享存储空间的出栈函数

```
template < typename Element >
Element DoubleSStack < Element > :: pop(int stackNumber)
{
    if (stackNumber == 1){
        if (top1 == -1) throw "下溢";
        x = data[top1--];
    }
    else (stackNumber == 2){
        if (top2 == MaxSize) throw "下溢";
        x = data[top2++];
    }
    return x;
}
```

对于两栈共享空间的使用,需要满足两个条件:①两栈数据类型相同,如果不同,使用这种方法不但不能更好地处理问题,反而会使问题变得更复杂;②两个栈的空间需求相反,就是一个栈增长时另一个栈缩短,这样两栈共享空间的存储方法才有较大的意义,否则若两个栈都不停地增长,那很快就会因栈满而溢出了。

3.2.4 栈的链接存储结构及其实现

栈中数据元素之间呈现线性关系,也可以像线性表一样采用链接存储结构,采用链接存储结构的栈称为链栈(linked stack),通常采用单链表实现。不失一般性,可用单链表的头部作栈顶,尾部作栈底;因为栈只在栈顶插入和删除元素,因此设头指针即栈顶指针 top,不需要像单链表那样为了运算方便增加头结点,如图 3.4 所示。

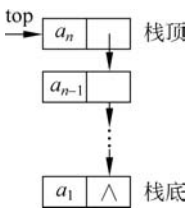


图 3.4 链栈示意图

将栈的抽象数据类型定义在链接存储结构下用 C++语言中的类实现,如代码 3.9 所示,其中成员变量 top 为栈顶指针。

代码 3.9 链栈的类定义

```
template < typename Element >
class LStack
{
public:
    LStack();                    //构造函数,构造一个空栈
    ~LStack();                   //析构函数,销毁栈
    void push(Element x);        //入栈操作,将元素 x 入栈
    Element pop();               //出栈操作,将栈顶元素返回
    Element getTop();            //取栈顶元素
    int empty();                 //判断栈空
private:
    Node < Element > * top;      //栈顶指针
};
```

下面按照链栈的类实现逐个讨论每个成员函数的实现。

1. 构造函数,构造一个空栈

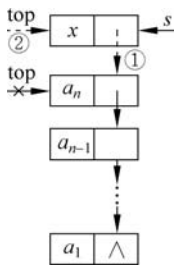


图 3.5 链栈入栈示意图

当栈顶指针 top 为空时,栈为空;那么构造一个空栈只需置 top=nullptr 即可。

2. 析构函数,销毁链栈

链栈是动态存储分配,其析构函数需要释放链栈的所有存储空间,与单链表的析构函数类似。

3. 入栈操作,将元素 x 入栈

对于链栈来说,不存在栈满的情况,除非内存没有可以使用的空间了。假设要入栈的元素是 x,新结点是 s,则将结点 s 入栈的操作如图 3.5 所示,算法示例见代码 3.10。

代码 3.10 链栈的入栈函数

```
template < typename Element >
void LStack < Element > :: push(Element x)
{
    s = new Node < Element >;
    s -> data = x;          //申请结点 s 数据域为 x
    s -> next = top;        //将结点 s 插在栈顶
    top = s;
}
```

4. 出栈操作,将栈顶元素返回

链栈的出栈操作只在栈顶进行:先判断是否为空栈,若非空,用 p 存储要删除的栈顶结点,将栈顶指针下移一位,如图 3.6 所示,最后释放结点 p 的存储空间即可。算法示例见代码 3.11。

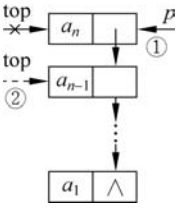


图 3.6 链栈出栈示意图

代码 3.11 链栈的出栈函数

```
template < typename Element >
Element LStack < Element > :: pop()
{
    if (top == nullptr) 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;
}
```

链栈的入栈和出栈操作都很简单,没有任何循环操作,时间复杂度均为 $O(1)$。

5. 取栈顶元素

该操作与出栈操作类似,不同之处就是不删除栈顶元素,即栈顶指针 top 不改变,只是将栈顶数据元素返回。

6. 判断栈是否为空

该操作只需要判断栈顶指针 top 是否为空即可。

顺序栈与链栈的时间复杂度一样，都是 $O(1)$。对于空间性能，顺序栈需要事先确定一个固定的长度，存在存储元素个数限制和空间浪费的问题，但它的优势是存取时定位非常方便；而链栈要求每个元素都有指针域，从而产生了结构性开销，但不存在栈满的情况，对于栈的长度无限制。所以它们的区别和线性表中的讨论一样，如果在栈的使用过程中元素个数变化不可预料或变化较大，最好使用链栈；反之，如果元素个数在可控范围内，建议使用顺序栈更方便些。

3.3 队　　列

队列有广泛的应用，如生活中的排队和运筹学中的排队论。本节主要讨论队列的定义及其实现。

3.3.1 队列的定义

队列(queue)也是一种操作受限的线性表，只允许在表的一端进行插入操作，在表的另一端进行删除操作。队列是一种先进先出(first in first out，FIFO)的线性表，允许插入的一端称为队尾(rear)，允许删除的一端称为队头(front)，如图 3.7 所示。这个队列中有五个元素，a_1 是队头元素，a_5 是队尾元素；此时如果要出队，则 a_1 出队，a_2 就成为新的队头元素；此时如果要入队一个元素 x，则在 a_5 这一端入队，入队后 x 就成为新的队尾元素。

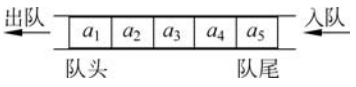


图 3.7　队列的示意图

队列的抽象数据类型定义如下：

```
ADT 名:Queue
    Data:
        数据对象:D = {a_i | 1 ≤ i ≤ n,n ≥ 0,a_i 是 Element 类型数据}
        数据关系:R = {< a_i,a_i+1 > | 1 ≤ i ≤ n - 1}
    Operation:
        InitQueue(&Q):初始化队列,构造一个空队列 Q。
        DestroyQueue (&Q):销毁队列,释放队列 Q 所占用的存储空间。
        Empty(Q):判断队列是否为空,若队列 Q 为空,则返回真值;否则返回假。
        EnQueue(&Q,x):入队,将元素 x 插入队列 Q 中作为新队尾元素。
        DeQueue(&Q,&x):出队,从队列 Q 中删除队头元素,并用 x 返回其值。
        GetQueue(Q,&x):取队头元素,用 x 返回当前的队头元素。
    End ADT
```

【例 3.3】 若元素的进队顺序为 12345，能否得到 32145 的出队顺序？

解：队列不同于栈，若进队顺序为 12345，出队顺序只能有一种，即 12345，所以不能得到 32145 的出队顺序。

3.3.2 队列的顺序存储结构及其实现

队列中的数据元素的逻辑关系呈线性关系,所以也可以像线性表一样采用顺序存储结构进行存储,即分配一块连续的存储空间来存放队列中的元素,在C++语言中用数组实现;同时,用两个整型变量反映队列中元素的变化,它们分别存储队头元素和队尾元素在数组中的下标,分别称为队头指针(front)和队尾指针(rear)。采用顺序存储结构存储的队列称为顺序队列(sequential queue)。一个空的顺序队列如图3.8(a)所示。不失一般性,从数组下标为0的一端开始入队,所以此时front和rear都是-1。每入队一个元素,队尾指针rear+1,rear总是指向队尾元素;每出队一个元素,队头指针front+1,front总是指向队头元素的前一个位置。那么a_1、a_2、a_3依次入队后队列的状态如图3.8(b)所示。此时如果a_4、a_5、a_6继续入队,则队列状态如图3.8(c)所示,队列已满。如果a_1、a_2、a_3依次出队,队列的状态如图3.8(d)所示,此时队列中虽然只有三个元素,但若想继续入队,数组高下标端已经没有存储空间了,表示队列已满,但实际上数组低下标端还有空闲位置,这种状态称为假溢出。随着顺序队列的入队和出队不断进行,队列中的元素在数组中呈现"单向移动性",也就出现了这种假溢出的现象。

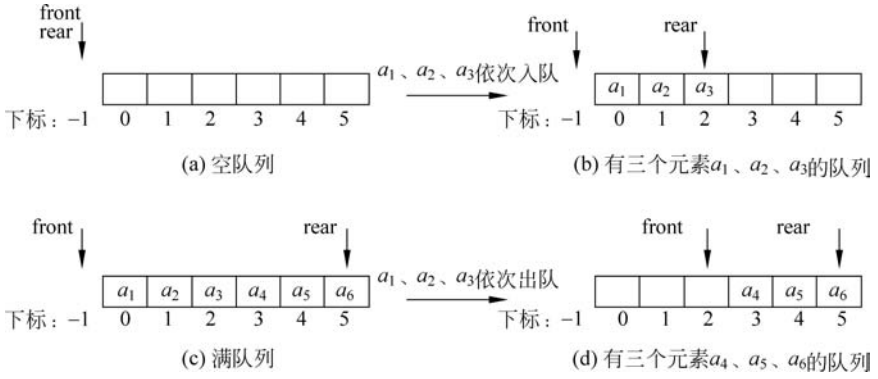


图3.8 顺序队列入队和出队

那么如何解决假溢出的问题呢?假溢出问题是因为元素从队头出队后的存储单元不能被重新利用而产生的,可以在元素出队时让队列整体前移,始终保持队头元素在下标为0的位置。这样处理虽然解决了假溢出的问题,但出队操作的时间复杂度从$O(1)$提高为$O(n)$。那还有其他解决办法吗?

摩天轮是大朋友小朋友都非常喜欢的一种游乐设施,只要摩天轮上还有空的吊篮,就可以让游客乘坐,不会出现这种假溢出的情况,那顺序队列是否也可以像摩天轮一样让存储单元形成一个环形以避免假溢出呢?答案是肯定的,可以将一维数组从逻辑上首尾相接,形成一个环状存储结构,如图3.9所示。

图3.9(a)是一个空队列,front和rear都是0;a_1、a_2、a_3依次入队,rear为3;a_4、a_5、a_6继续入队,rear在循环状态下经过三次加1后又变为0,这时可以发现:队列为空时,front=rear;队列为满时,front=rear。那么如何区分队空队满呢?有多种方法可以采用:①增加一个队列长度的成员变量,通过对队列长度的判断可以区分队空队满;②增加一个布尔型变量,有元素入队时将其置为true,有元素出队时将其置为false,此变量与front=

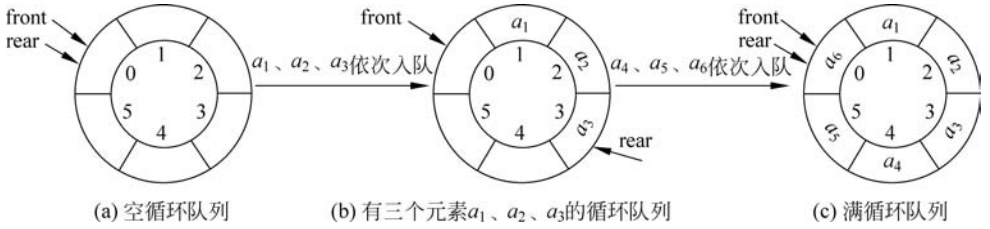


图 3.9　循环队列

rear 结合使用可判断队空队满；③浪费一个单元的存储空间，即当 rear 和 front 的位置正好差 1 时，认为队满，如图 3.10(a)所示。本书采取方法③。

需要注意的是，在循环队列中入队和出队时，rear 和 front 不能单纯加 1，需要在循环意义下加 1。出队一个元素时：front＝(front＋1)％QueueSize，如图 3.10(b)所示。入队一个元素时：rear＝(rear＋1)％QueueSize，如图 3.10(c)所示。同样判断队列是否已满时：(rear＋1)％QueueSize ＝＝ front。这里 QueueSize 是队列的容量，即数组的长度，％是 C++语言中的求余符号。

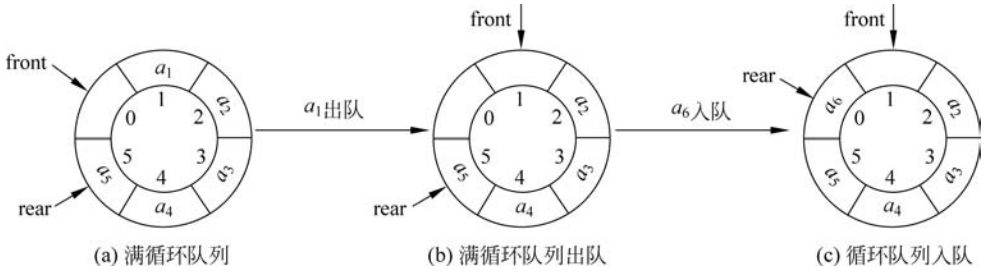


图 3.10　满循环队列与出队入队

按这个思路设计的循环队列存储结构用 C++语言中的类实现，类定义见代码 3.12。

代码 3.12　循环队列的类定义

```
const int QueueSize = 10;          //根据实际问题具体定义
template < typename Element >
class CQueue
{
public:
    CQueue();                      //构造函数,构造一个空循环队列
    ~ CQueue();                    //析构函数
    void enQueue(Element x);       //入队操作,将元素 x 入队
    Element deQueue();             //出队操作,将队头元素返回
    Element getQueue();            //取队头元素
    int empty();                   //判断循环队列是否为空
private:
    Element data[QueueSize];       //存放队列元素的数组
    int front, rear;               //队头和队尾指针
};
```

下面按照循环队列的类定义逐个讨论每个成员函数的实现。

1. 构造函数,构造一个空循环队列

当 front=rear 时,循环队列为空。那么初始化一个空的循环队列,只需要将 front 和 rear 初始化为同一个值即可,不失一般性,可以令 front=rear=-1。

2. 析构函数,销毁循环队列

循环队列是静态存储分配,在循环队列变量退出作用域时系统会自动释放队列所占存储空间,因此循环队列无须销毁,析构函数为空函数。

3. 入队操作,将元素 x 入队

该操作首先要判断队列是否已满,若已满则抛出上溢异常;否则先将队尾指针在循环意义下加 1,再将元素 x 插入在该位置上。入队函数实现见代码 3.13。

代码 3.13 循环队列的入队函数

```
template < typename Element >
void CQueue < Element > :: enQueue(Element x)
{
    if ((rear + 1) % QueueSize == front) throw "上溢";
    rear = (rear + 1) % QueueSize;          //队尾指针在循环意义下加 1
    data[rear] = x;                         //在队尾处插入元素
}
```

4. 出队操作,将队头元素返回

该操作首先要判断队列是否为空,若为空则抛出下溢异常;否则先将队头指针在循环意义下加 1,然后将队头指针所指元素返回即可。出队函数实现见代码 3.14。

代码 3.14 循环队列的出队函数

```
template < typename Element >
Element CQueue < Element > :: deQueue()
{
    if (rear == front) throw "下溢";
    front = (front + 1) % QueueSize;        //队头指针在循环意义下加 1
    return data[front];                     //读取并返回出队前的队头元素
}
```

5. 取队头元素

该操作与出队操作类似,不同之处就是不出队,即队头指针 front 不改变。

6. 判断队列是否为空

该操作只需要判断队头指针和队尾指针是否相等即可。

代码 3.15 循环队列判断队空函数

```
template < typename Element >
int CQueue < Element > :: empty()
{
    if (rear == front) return 1;
    else return 0;
}
```

上述基本操作均不包含循环结构，时间复杂度均为 $O(1)$。在循环队列中，队头指针 front 始终指向队头元素的前一个位置，队尾指针 rear 指向队尾元素，队列中的元素个数为 (rear－front＋ QueueSize)％QueueSize。

3.3.3 双端队列

双端队列(double-ended queue)是队列的扩展，指两端都可以进行入队和出队操作的队列，如图 3.11 所示。队列的两端分别称为队头和队尾，两端都可以入队和出队，队列中元素的逻辑关系仍是线性关系。从图 3.11 还可以看出，从队尾入队，队头出队或者从队头入队，队尾出队体现出先进先出的特点，从队头入队，队头出队，或者从队尾入队，队尾出队，体现出后进先出的特点。

如果允许在队列的两端入队，但只允许在一端出队，则称为两进一出队列；如果允许在队列的两端出队，但只允许在一端入队，则称为两出一进队列；如果限定双端队列中从某端进队的元素只能从该端出队，则该双端队列就退变为两个栈底相邻的栈了。

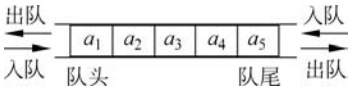


图 3.11 双端队列示意图

双端队列和普通队列类似，都有入队、出队等基本操作，只是操作时需指明操作的位置。双端队列在循环队列这种存储结构下可用 C++语言中的类实现，其类定义见代码 3.16。

代码 3.16 双端队列的类定义

```
const int QueueSize = 10;
template < typename Element >
class DEQueue
{
public:
    DEQueue();                          //构造函数,构造一个空的双端队列
    ～ DEQueue();                       //析构函数
    void enQueueFront(Element x);       //入队操作,将元素 x 从队头入队
    void enQueueRear(Element x);        //入队操作,将元素 x 从队尾入队
    Element deQueueFront();             //出队操作,将队头元素出队并返回
    Element deQueueRear();              //出队操作,将队尾元素出队并返回
    Element getQueueFront();            //取队头元素,将队头元素返回
    Element getQueueRear();             //取队尾元素,将队尾元素返回
    int empty();                        //判断循环队列是否为空
private:
    Element data[QueueSize];            //存放队列元素的数组
    int front, rear;                    //队头和队尾指针
};
```

双端队列的基本操作都与循环队列类似，这里只对队头入队和队尾出队两个操作进行分析实现，其他操作请读者参照循环队列自行实现。

1. 入队操作，将元素 *x* 从队头入队

循环队列中，队尾指针指向队尾元素，从队尾入队，队尾指针在循环意义下加 1；队头元素指向队头元素的前一个位置，从队头出队时，队头指针在循环意义下加 1。那么对于双端

队列,从队头入队,需要先将入队元素 x 存入队头指针所指位置,然后队头指针在循环意义下减 1,如图 3.12 所示。在算法实现时首先判断队列是否已满,具体算法示例见代码 3.17。

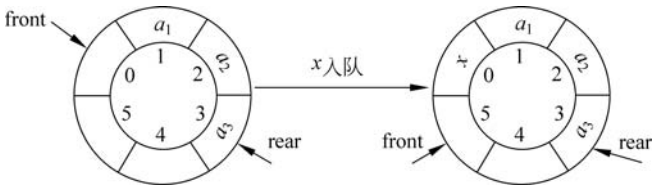


图 3.12　双端队列队头入队示意图

代码 3.17　双端队列的队头入队函数

```
template < typename Element >
void DEQueue < Element > :: enQueueFront(Element x)
{
    if ((rear + 1) % QueueSize == front) throw "上溢";
    data[front] = x;
    front = (front - 1 + QueueSize) % QueueSize;
}
```

2. 出队操作,将队尾元素出队并返回

队尾指针指向队尾元素,首先需要将队尾元素暂存,然后队尾指针在循环意义下减 1,最后把暂存的队尾元素返回,如图 3.13 所示。在算法实现时首先应判断队列是否为空,具体算法示例见代码 3.18。

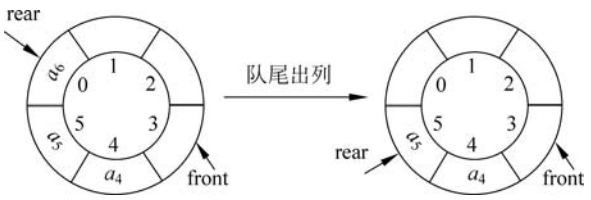


图 3.13　双端队列队尾出队示意图

代码 3.18　双端队列的队尾出队函数

```
template < typename Element >
Element DEQueue < Element > :: deQueueRear()
{
    if (rear == front) throw "下溢";
    Element x = data[rear];
    rear = (rear - 1 + QueueSize) % QueueSize;
    return x;
}
```

3.3.4 队列的链接存储结构及其实现

队列也可以用链接存储方式存储，与单链表类似。采用链接存储结构存储的队列称为链队列(linked queue)。使用单链表实现链队列，队头指针指向单链表的头结点，队尾指针指向单链表的尾结点，如图 3.14 所示。

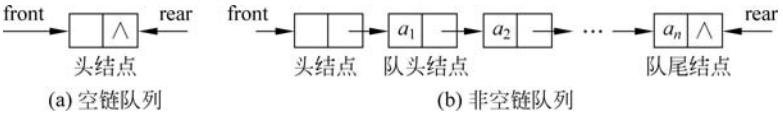


图 3.14 链队列示意图

这里增加头结点的目的是使对空队列和非空队列的操作一致。

将队列的抽象数据类型定义在链队列存储结构下用 C++语言中的类实现，其类定义见代码 3.19。

代码 3.19 链队列的类定义

```
template < typename Element >
class LQueue
{
public:
    LQueue();                              //构造函数,构造一个空链队列
    ~LQueue();                             //析构函数
    void enQueue(Element x);               //入队操作,将元素 x 入队
    Element deQueue();                     //出队操作,将队头元素返回
    Element getQueue();                    //取队头元素
    int empty();                           //判断队列是否为空
private:
    Node < Element > * front, * rear;      //队头和队尾指针
};
```

下面按照链队列的类定义逐个讨论每个成员函数的实现。

1. 构造函数，构造一个空链队列

空链队列包含头指针、尾指针和头结点，构造函数就是要初始化这些参量。函数实现见代码 3.20。

代码 3.20 链队列的构造函数

```
template < typename Element >
LQueue < Element > :: LQueue()
{
    s = new Node < Element >;          //申请头结点,并让其指针域为空
    s -> next = nullptr;
    front = rear = s;                  //初始化队头指针和队尾指针指向头结点
}
```

2. 析构函数,销毁链队列

链队列是动态存储分配,在链队列变量退出作用域时需要释放所有结点的存储空间。链队列的析构函数与单链表的析构函数类似,读者可自行实现。

3. 入队操作,将元素 x 入队

该操作首先申请一个结点,将元素 x 存入结点数据域,然后将该结点插入尾结点之后,队尾指针需要更新指向该结点。入队函数实现见代码 3.21。

代码 3.21 链队列的入队函数

```
template < typename Element >
void LQueue < Element > :: enQueue(Element x)
{
    s = new Node < Element >;          //申请结点并初始化
    s -> data = x;
    s -> next = nullptr;
    rear -> next = s;                  //插入队尾
    rear = s;
}
```

4. 出队操作,将队头元素返回

该操作首先要判断队列是否为空,若为空则抛出下溢异常;否则先将队头元素暂存,再将队头结点摘链,最后返回暂存的被删元素。出队函数实现见代码 3.22。

代码 3.22 链队列的出队函数

```
template < typename Element >
Element LQueue < Element > :: deQueue()
{
    if (rear == front) throw "下溢";          //判断若栈空,抛出下溢异常
    p = front -> next;                        //p 指向队头元素所在结点
    x = p -> data;                            //暂存队头元素
    front -> next = p -> next;                //摘链
    if (p -> next == nullptr) rear = front;   //若队列中唯一的元素出队,需更新队尾指针
    delete p;
    return x;
}
```

5. 取队头元素

该操作与出队操作类似,不同之处就是不删除队头结点。

6. 判断队列是否为空

该操作只需要判断队头指针和队尾指针是否相等即可。判断队空函数实现见代码 3.23。

代码 3.23 链队列的判断队空函数

```
template < typename Element >
int LQueue < Element > :: empty()
```

```
{
    if (rear == front) return 1;
    else return 0;
}
```

上述基本操作均不包含循环结构，时间复杂度均为 $O(1)$。

可以从时间和空间两方面比较循环队列和链队列。循环队列和链队列基本操作的时间复杂度都是常量阶的，不过循环队列是提前申请好存储空间，使用期间不释放，而链队列每次申请和释放结点都会存在一些时间开销，如果入队出队频繁，则两者还是有细微差别的。从空间上来说，循环队列必须预先确定一个固定的长度，所以就有了存储元素个数有限和空间浪费的问题。而链队列没有这个问题，但是每个元素都需要一个指针域，从而产生了结构性开销。总之，如果可以确定队列的最大长度，建议使用循环队列，反之使用链队列。

3.4 栈和队列的应用

栈和队列都是操作受限的线性表，其基本操作用线性表也完全可以实现，那为什么要引入栈和队列这两种数据结构呢？这样简化了程序设计的问题，划分了不同的关注层次，使得解决问题时需要思考的范围缩小，更能聚焦于所要解决的问题核心。反之，像数组或单链表，要分散精力去考虑数组下标增减或链表指针移动等细节问题，反而掩盖了问题的核心。现在的高级语言，如 C++、Java、C＃等都有对栈和队列的封装，可以直接使用而无须关注其实现细节。

3.4.1 数制转换

十进制数和其他进制数的转换是计算机实现计算的基本问题，解决方法很多，其中一个简单的整数转换算法原理如下：

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

其中，N 是十进制数，d 是 d 进制，div 是整除运算，mod 是求余运算。

例如：$(2892)_{10} = (5514)_8$，运算过程如下：

N	N div 8	N mod 8
2892	361	4
361	45	1
45	5	5
5	0	5

那么现在要编写一个满足下列要求的程序：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。观察上述运算过程，是从低位到高位的顺序依次产生八进制数的各个数位，而打印输出时，需要从高位到低位进行，这与运算过程恰好相反，与栈的后进先出的特性一致。因此，将运算过程中得到的八进制数按顺序入栈，然后按出栈顺序逐个打印输出即为与所输入数值对应的八进制数。C++实现程序见代码 3.24。

代码 3.24 数制转换的 C++实现

```
#include <iostream>
#include "SStack.cpp"
using namespace std;
int main()
{
    int n;
    SStack<int> stack;          //定义一个顺序栈对象
    cin >> N;                   //输入十进制数 N
    while (N)                   //N 不为 0 时循环
    {
      stack.Push(N % 8);        //将余数依次入栈
      N = N / 8;
    }
    while (!stack.empty())      //栈非空时循环
    {
      cout << stack.Pop();      //元素依次出栈并打印
    }
}
```

程序中使用了 3.2.2 节实现的顺序栈类,编程时需要使用“#include "SStack.cpp""将此类包含进来。

3.4.2 函数的调用与递归

用高级语言编写的程序中,调用函数和被调用函数之间的链接及信息交换需要通过栈来进行。通常,在一个函数 A 的运行期间调用另一个函数 B 时,在运行被调用函数 B 之前,系统需要完成三项工作:①将所有的实参、返回地址等信息传递给被调用函数 B 保存;②为被调用函数 B 的局部变量分配存储区;③将控制转移到被调函数的入口。而从被调用函数 B 返回调用函数 A 之前,系统也要完成三项工作:①保存被调用函数 B 的计算结果;②释放被调函数 B 的数据区;③按照被调函数 B 保存的返回地址将控制转移到调用函数 A。当有多个函数构成嵌套调用时,按照“先调用后返回”的原则,上述函数之间的信息传递和控制转移需要通过“栈”来实现,即系统将整个程序运行时所需的数据空间安排在一个栈中。每当调用一个函数时,就为它在栈顶分配一个存储区;每当从一个函数退出时,就释放它的存储区,则当前正在运行的函数的数据区就在栈顶。例如,在图 3.15(a)中,主函数 main 中调用了函数 A,而在函数 A 中又调用了函数 B,图 3.15(b)展示了当前正在执行函数 B 中某个语句时栈的状态,而图 3.15(c)展示了从函数 B 返回之后正在执行函数 A 中某个语句时栈的状态(图 3.15 中以语句标号表示返回地址)。

直接调用自己或通过一系列调用语句间接调用自己的函数,称作递归函数。递归函数的运行过程类似多个函数的嵌套调用,只是调用函数和被调用函数是同一个函数。下面以斐波那契数列为例说明递归调用的过程。

斐波那契数列(Fibonacci sequence),又称黄金分割数列,因数学家莱昂纳多·斐波那契(Leonardoda Fibonacci)以兔子繁殖为例子而引入,故又称为“兔子数列”。

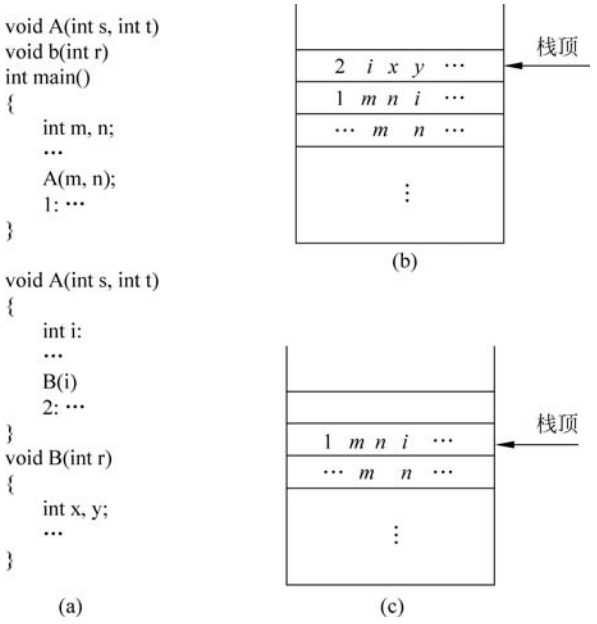


图 3.15　函数嵌套调用时运行栈的状态

如果兔子在出生两个月后就有繁殖能力，一对兔子每个月能生出一对小兔子。假设所有兔子都不死，那么一年以后可以繁殖多少对兔子呢？

不妨拿新出生的一对幼兔 AA 分析一下：第一个月幼兔 AA 没有繁殖能力，所以还是一对；两个月后，幼兔 AA 长成成兔，生下一对幼兔 BB，对数共有两对；三个月以后，成兔 AA 又生下一对，因为幼兔 BB 还没有繁殖能力，所以一共是三对……依此类推可以列出表 3.1。

表 3.1　兔子数列

经过月数	幼兔对数	成兔对数	总对数
0	1	0	1
1	0	1	1
2	1	1	2
3	1	2	3
4	2	3	5
5	3	5	8
6	5	8	13
7	8	13	21
8	13	21	34
9	21	34	55
10	34	55	89
11	55	89	144
12	89	144	233
…	…	…	…

幼兔对数＝前月成兔对数

成兔对数＝前月成兔对数＋前月幼兔对数

总对数＝本月成兔对数＋本月幼兔对数

可以看出,总体对数构成了一个数列,这个数列有十分明显的特点:前面相邻两项之和,构成了后一项。列式如下:

$$F(0)=0$$

$$F(1)=1$$

$$F(n)=F(n-1)+F(n-2)(n\geqslant 2)$$

用C++语言编程打印前20位的斐波那契数列,程序见代码3.25。

代码3.25 斐波那契数列的C++实现

```
#include <iostream>
using namespace std;
int F(int i)
{
    if (i < 2)
      return i == 0 ? 0 : 1;
    return F(i - 1) + F(i - 2);          //递归调用
}
int main()
{
    int n;
    for (n = 0;n < 20;n++)
    {
      cout << F(n) << "\t";
    }
}
```

递归调用中一个重要的概念是递归函数运行的“层次”,假设调用该递归函数的主函数为第0层,则从主函数调用递归函数为进入第1层。从第i层递归调用本函数为进入“下一层”,即为第$i+1$层;反之,从第i层返回至“上一层”,即为第$i-1$层。为了保证递归函数正确执行,系统设立一个“工作栈”作为整个递归函数运行期间使用的数据存储区。每一层递归所需信息构成一个“工作记录”,其中包括所有的实参、局部变量以及上一层的返回地址。每进入一层递归,就产生一个新的工作记录压入栈顶。每退出一层递归,就从栈顶弹出一个工作记录,则当前执行层的工作记录就是工作栈栈顶的工作记录。图3.16详细展示了求斐波那契数列中第五位$F(5)$的递归调用过程,图中的“1:”和“2:”分别对应代码3.25中函数int F(int i)中的两条return语句。从图3.16中也可以看到,求$F(5)$需要递归调用五层,最后调用的函数最先返回,最先调用的函数最后返回,这也是栈的特性。

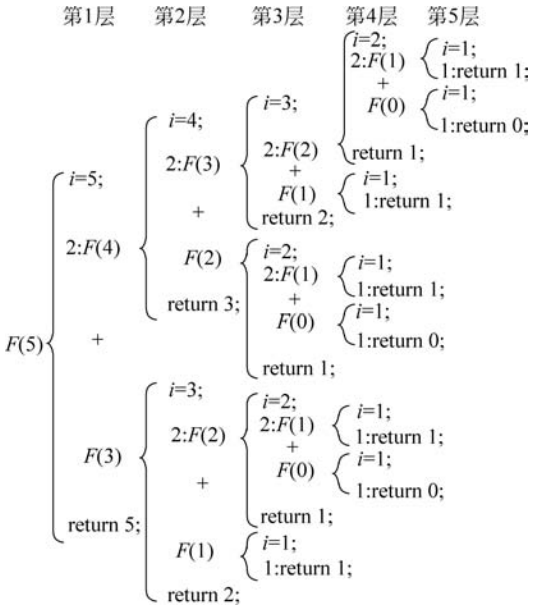


图 3.16　$F(5)$的递归调用过程

3.4.3　表达式求值

表达式求值是小学生学习数学经常遇到的问题，也是程序设计语言编译中的一个最基本问题，它的实现是栈应用的一个典型例子。这里以简单的只带小括号的四则算术表达式求值为例。

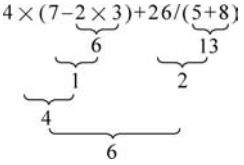
小学数学中学习过，四则运算的规则是：

(1) 先乘除，后加减；

(2) 从左算到右；

(3) 先括号内，后括号外。

例如：4×(7－2×3)＋26/(5＋8)，它的计算顺序如下：



这个表达式非常简单，通过口算很容易就能得到答案，那计算机如何按照这个计算优先顺序实现呢？其中的困难就在于求值时不仅要考虑运算符的优先级，还要处理括号。波兰逻辑学家 J. 卢卡西维兹(J. Lukasiewicz)于 1929 年首先提出了一种表达式的表示方法——逆波兰式，也称为后缀表达式(postfix expression)，就是所有的运算符都在对应的操作数后面出现；日常生活中经常使用的表达式形式，也就是运算符在两个操作数的中间，称为中缀表达式(infix expression)。中缀表达式“4＊(7－2＊3)＋26/(5＋8)”的后缀表达式为“4＃7＃2＃3＃＊－＊26＃5＃8＃＋/＋”。为了在后缀表达式中区分相邻的操作数，在每个操作数末尾添加一个字符“＃”。后缀表达式中没有括号，只有操作数和运算符，越放在前面的运

算符优先级越高。计算机就是先将中缀表达式转换为后缀表达式,然后对后缀表达式求值以得到中缀表达式的运算结果。

1. 后缀表达式求值

为了进一步体会后缀表达式的好处,首先需要了解计算机如何对后缀表达式求值。设有后缀表达式为"4#7#2#3#*-*26#5#8#+/+",基本的求值规则:从左到右遍历表达式的每个数字和符号,遇到数字就进栈,遇到运算符就将处于栈顶的两个数字弹出并进行运算,然后将运算结果进栈,一直到最终获得结果。

求值步骤如下:

(1) 初始化一个用于操作数的空栈,表达式的前四个都是数字,依次入栈,如图 3.17(b)所示。

(2) 接下来是"*",将栈中的"3"和"2"依次出栈作为乘数,计算"2*3"得到"6",再将"6"进栈,如图 3.17 (c)所示。

(3) 紧接着是"-",将栈中的"6"和"7"依次出栈作为减数和被减数,计算"7-6"得到"1",再将"1"进栈,如图 3.17 (d)所示。

(4) 接下来又是"*",将栈中的"1"和"4"依次出栈作为乘数,计算"4*1"得到"4",再将"4"进栈,如图 3.17 (e)所示。

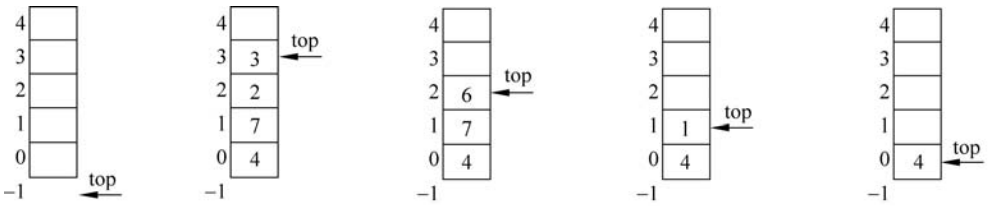


图 3.17 后缀表达式求值过程(1)

(5) 接下来将三个数字"26""5"和"8"依次入栈,如图 3.18(a)所示。

(6) 接着是"+",将栈中的"8"和"5"依次出栈作为加数,计算"5+8"得到"13",再将"13"进栈,如图 3.18(b)所示。

(7) 紧接着是"/",将栈中的"13"和"26"依次出栈作为除数和被除数,计算"26/13"得到"2",再将"2"进栈,如图 3.18(c)所示。

(8) 最后一个符号是"+",将栈中的"2"和"4"依次出栈作为加数,计算"4+2"得到"6",再将"6"进栈,如图 3.18(d)所示,求值结果就是栈顶元素"6"。

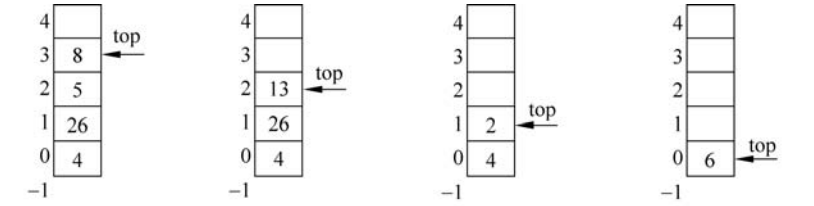


图 3.18 后缀表达式求值过程(2)

后缀表达式求值程序如代码 3.26 所示，这里用栈存储数字，假定使用顺序栈。

代码 3.26 后缀表达式求值的 C++实现

```
int compute(string str)                                    //str 是后缀表达式,返回求值结果
{
    SStack < int > stack;                                  //定义存储数字的顺序栈
    int i = 0, a, b, c, d;
    while (str[i] != '\0')                                 //遍历后缀表达式
    {
        switch (str[i])
        {
        case '+':
            a = stack.Pop();
            b = stack.Pop();
            c = b + a;
            stack.Push(c);
            break;
        case '-':
            a = stack.Pop();
            b = stack.Pop();
            c = b - a;
            stack.Push(c);
            break;
        case '*':
            a = stack.Pop();
            b = stack.Pop();
            c = b * a;
            stack.Push(c);
            break;
        case '/':
            a = stack.Pop();
            b = stack.Pop();
            if (a != 0)
            {
                c = b / a;
                stack.Push(c);
                break;
            }
            else throw "除数不能为 0!";
            break;
        default:
            d = 0;
            while (str[i] >= '0'&&str[i] <= '9')           //处理数字
            {
                d = 10 * d + str[i] - '0';
                i++;
            }
            stack.Push(d);                                 //数字入栈
            break;
```

```
            }
            i++;
        }
        return stack.Pop();
    }
```

可见,计算机使用后缀表达式求值非常方便,那怎么才能把中缀表达式转换为后缀表达式呢?下面就来学习这个知识点。

2. 中缀表达式转后缀表达式

中缀表达式"4*(7-2*3)+26/(5+8)"转化为后缀表达式"4#7#2#3#*-*26#5#8#+/+",规则如下:

(1) 从左到右遍历中缀表达式,如果是数字就直接输出,如果是运算符,则执行步骤(2)。

(2) 判断其与栈顶运算符的优先级,若优先级相等,则肯定是")",栈顶的"("直接出栈即可;若优先级高,则直接入栈;若优先级低,则栈顶元素依次出栈并输出,直到栈顶元素优先级低于当前运算符,当前运算符入栈。

(3) 执行步骤(1)、步骤(2),一直遍历到表达式结束,栈中运算符依次出栈到栈空为止。

根据四则运算的规则,在运算的每一步中,任意两个相继出现的运算符 θ1 和 θ2 的优先级如表 3.2 所示。

表 3.2 运算符之间的优先级关系

θ1 / θ2	+	-	*	/	(	)
+	>	>	<	<	<	>
-	>	>	<	<	<	>
*	>	>	>	>	<	>
/	>	>	>	>	<	>
(	<	<	<	<	<	=
)	>	>	>	>	=	>

中缀表达式转化为后缀表达式的过程如下:

(1) 初始化一个用于运算符的空栈,第一个字符是数字"4",直接输出;第二个字符是"*",此时栈空,直接入栈,如图 3.19(a)所示。

(2) 第三个字符是"(",优先级高于栈顶的"*",直接入栈;第四个字符是数字"7",直接输出,接下来是"-",优先级高于栈顶的"(",直接入栈,如图 3.19 (b)所示。

(3) 第六个字符是数字"2",直接输出;接下来是"*",优先级高于栈顶的"-",直接入栈,如图 3.19 (c)所示。

(4) 第八个字符是数字"3",直接输出;接下来是")",优先级低于栈顶的"*","*"输出;此时")"优先级仍低于栈顶的"-",继续输出"-";之后")"与栈顶的"("优先级相同,栈顶的"("直接出栈即可,如图 3.19 (d)所示。

(5) 第十个字符是"+",优先级低于栈顶的"*",此时输出"*";这时栈空,"+"入栈,如图 3.20(a)所示。

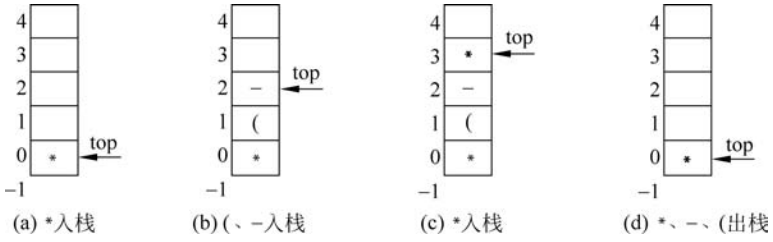


图 3.19　中缀表达式转后缀表达式示意图(1)

(6) 之后是数字“26”,直接输出,接下来是“/”,优先级高于栈顶的“+”,直接入栈;之后是“(”,优先级高,直接入栈;接下来是数字“5”输出,“+”优先级高于“(”,直接入栈,如图 3.20(b)所示。

(7) 数字“8”直接输出,“)”优先级低于栈顶的“+”,“+”出栈输出;之后“)”优先级与栈顶的“(”优先级相等,“(”出栈,如图 3.20(c)所示;表达式遍历完毕,栈内的运算符依次出栈输出即可,如图 3.20(d)所示。

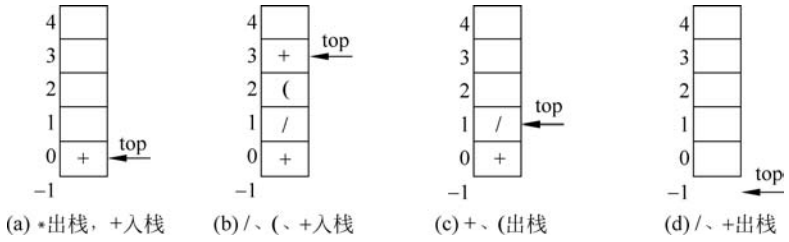


图 3.20　中缀表达式转后缀表达式示意图(2)

中缀表达式转后缀表达式的程序如代码 3.27 所示,其中用抽象数据类型栈存储运算符,这里同样使用顺序栈。

代码 3.27　中缀表达式转后缀表达式的 C++实现

```
string trans(string str1)                                    //str1 是中缀表达式,返回转后结果
{
    SStack < char > stack;                                   //定义存储运算符的顺序栈
    string str2;
    int i = 0, k;
    stack.Push('#');                                         //为避免重复判栈空,增加 #
    while (str1[i] != '\0')
    {
        if(str1[i] < '0'|| str1[i] > '9')                    //若非数字
        {
            k = compare(stack.GetTop(), str1[i]);            //比较优先级
            switch (k)
            {
            case 0:stack.Pop();                              //优先级相等
                i++;
                break;
            case -1:stack.Push(str1[i]);                     //str1[i]优先级高,直接入栈
```

```
                i++;
                break;
            case 1:do                                   //str1[i]优先级低,则依次出栈
                {
                    str2 = str2 + stack.Pop(); //直到 str1[i]优先级高于栈顶运算符
                }
            while (compare(stack.GetTop(), str1[i]) == 1);
                break;
            default:
                break;
            }
        }
        else                                            //若为数字
        {
            while (str1[i] >= '0'&&str1[i] <= '9')   //处理数字串
            {
                str2 = str2 + str1[i++];
            }
            str2 = str2 + '#';                          //标识一个数字串结束
        }

    }
    while(stack.GetTop()!= '#')
        str2 = str2 + stack.Pop();
    return str2;
}
```

中缀表达式转后缀表达式的算法中,若遍历中缀表达式时遇到运算符,需要与栈顶运算符比较优先级,而若栈为空,则认为该运算符优先级高,直接入栈。编程时为避免重复判断栈是否为空,先入栈一个运算符“#”,并且其他所有运算符的优先级都高于“#”。

程序中还用到了优先级比较函数 int compare(char theta1, char theta2),用于比较两个相继出现的运算符 theta1 和 theta2 的优先级,返回值为 1 表示 theta1 优先级高,0 表示相等,−1 表示 theta1 优先级低,其 C++语言实现见代码 3.28。

代码 3.28 优先级比较函数的 C++实现

```
int compare(char theta1, char theta2)
{
    switch (theta1)
    {
    case '+':case '-':if (theta2 == '+'|| theta2 == '-'|| theta2 == ')')return 1;
            else return -1;
        break;
    case '*':case '/':if (theta2 == '(')return -1;
            else return 1;
        break;
    case '(':if (theta2 == ')')return 0;
```

```
                else return -1;
            break;
        case ')':if (theta2 == '(')
            {
                throw "表达式不正确!";
                exit(0);
            }
                  else return 1;
            break;
        case'#': return -1;       //其他运算符优先级都高于"#"
            break;
        default:
            break;
        }
    }
```

3.4.4 魔王语言翻译官

传说有一个魔王总是使用自己的一种非常精炼而抽象的语言讲话，没人能听得懂。后来有一位智者发现魔王的语言是由以下两种形式的规则由人的语言逐步抽象上去的。

(1) $\alpha \rightarrow \beta_1\beta_2\beta_3\cdots\beta_n$

(2) $(\theta\delta_1\delta_2\delta_3\cdots\delta_n) \rightarrow \theta\delta_n\theta\delta_{n-1}\cdots\theta\delta_1\theta$

上面的规则(1)和规则(2)中，从左到右表示将魔王语言翻译成人类的语言。魔王语言和人类的语言按照该语法的规则进行转换。

设大写字母表示魔王语言词汇，小写字母表示人类语言词汇，魔王语言可以包含人类的词汇。假设把规则(1)具体化为以下两条规则：①$B \rightarrow tAdA$；②$A \rightarrow sae$。

则 $B(ehnxgz)B = tsaedsaeezegexenehetsaedsae$，若将小写字母与汉字建立以下对应关系进行翻译，则魔王说的话是："天上一只鹅地上一只鹅鹅追鹅赶鹅下鹅蛋鹅恨鹅天上一只鹅地上一只鹅。"那么怎样设计一个魔王语言的翻译系统，把魔王的话翻译成人能听懂的话呢？

t	*d*	*s*	*a*	*e*	*z*	*g*	*x*	*n*	*h*
天	地	上	一只	鹅	追	赶	下	蛋	恨

规则(1)和规则(2)不同。规则(1)是直接替换，先输入就先替换，与队列的特性一致，因此可以用队列来实现。规则(2)需要按照输入顺序的反序输出，也就是最后输入的需要先输出，与栈的特性一致，因此使用栈来实现。

魔王的语言中包含括号，如果括号不匹配，即左括号或右括号多余，则不能翻译。那么如何实现括号匹配呢？观察下列括号序列：

```
(  (  (  )  )  )
1  2  3  1  2  3
```

当输入第1个左括号时,它期待着与它匹配的右括号的出现,而等来的却是第2个左括号;这时第1个左括号只能靠边站,迫切期待与第2个左括号匹配的右括号的出现;这时出现的却是第3个左括号,第2个左括号也只能暂时靠边站;等到第1个右括号出现时,它与第3个左括号匹配成功;这时第2个右括号出现,第2个左括号的期待才能得到满足。对左括号来说,总是最后出现的能最先获得匹配,这与栈的特性相同,所以括号匹配检验可以用栈来实现。在算法中设置一个栈,每读入一个左括号就入栈,读入右括号时或者使得栈顶的左括号的期待得以满足,或者是不匹配;当算法结束时,栈若空就匹配成功,否则栈中的左括号就是多余的。

魔王语言翻译官的活动图如图3.21所示。

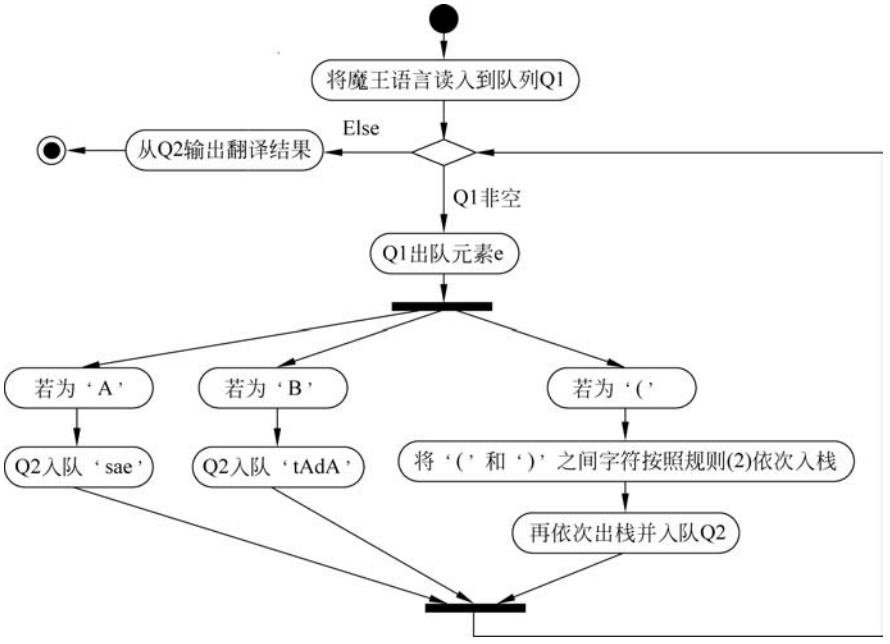


图3.21　魔王语言翻译官算法活动图

不失一般性,程序中使用顺序栈和链队列,对应的顺序栈类和链队列类的实现读者可参考3.1节和3.2节内容,魔王语言翻译官程序见代码3.29。

代码3.29　魔王语言翻译官的C++实现

```
#include<iostream>
#include<string>
using namespace std;
#include "SStack.cpp"                    //包含顺序栈类
#include "LQueue.cpp"                    //包含链队列类

//将魔王语言读入到队列
bool read(LQueue<char> &Q)
{
    int m,i;
    string str;
```

```
    SStack<char> stack;                    //用于括号匹配检验的栈
    cout << "请输入魔王语言:" << endl;
    cin >> str;
    m = str.length();
    for (i = 0;i < m;i++)                  //括号匹配检验
    {
        if (str[i] == '(') stack.Push(str[i]);
        else if (str[i] == ')')
        {
            if (stack.empty())             //若栈空,则右括号多余
                return false;
            else stack.Pop();
        }
    }
    if (!stack.empty())                    //若栈非空,则左括号多余
        return false;
    for (i = 0;i < m;i++)                  //读入队列
    {
        Q.enQueue(str[i]);
    }
    return true;
}
//字母'A'用'sae'代替
void enQueue_A(LQueue<char> &Q)
{
    Q.enQueue('s');
    Q.enQueue('a');
    Q.enQueue('e');
}
//将魔王语言转换为汉字输出
void show(LQueue<char> &Q)
{
    char e;
    cout << endl << "若将小写字母与汉字建立以下对应关系:" << endl;
    cout << "t  d  s a   e z g x  n  h" << endl;
    cout << "天 地 上一只鹅追赶下蛋恨" << endl;
    cout << "则魔王说的话是:" << endl;
    while (!Q.empty())
    {
        e = Q.deQueue();
        switch (e)
        {
        case 't':cout << "天";break;
        case 'd':cout << "地";break;
        case 's':cout << "上";break;
        case 'a':cout << "一只";break;
        case 'e':cout << "鹅";break;
        case 'z':cout << "追";break;
        case 'g':cout << "赶";break;
```

```
            case 'x':cout << "下";break;
            case 'n':cout << "蛋";break;
            case 'h':cout << "恨";break;
            }
        }
    }
    int main()
    {
        LQueue < char > q1,q2;              //魔王语言读入到 q1,翻译后存入 q2
        SStack < char > s;                  //规则(2)用栈来实现
        if (!read(q1))                      //读入魔王语言
        {
            cout << "括号不匹配,无法解释!" << endl;
            return 0;
        }
        char e,theta;
        while(!q1.empty())                  //翻译魔王语言
        {
            e = q1.deQueue();
            switch (e)
            {
            case 'A':enQueue_A(q2);         //字母'A'直接用'sae'代替
                break;
            case 'B':q2.enQueue('t');       //字母'B'用'tAdA'代替
                enQueue_A(q2);
                q2.enQueue('d');
                enQueue_A(q2);
                break;
            case'(':theta = q1.deQueue();     //'('开始规则(2)
                q2.enQueue(theta);
                e = q1.deQueue();
                while (e != ')')            //将')'之前部分入栈
                {
                    s.Push(e);
                    e = q1.deQueue();
                }
                while (!s.empty())          //逐个出栈,并按照规则(2)入队
                {
                    q2.enQueue(s.Pop());
                    q2.enQueue(theta);
                }
            }
        }
        cout << "魔王语言可以解释为:" << endl;
        LQueue < char > q3;                 //用于汉字输出显示的队列
        while (!q2.empty())                 //输出翻译结果
        {
            q3.enQueue(q2.GetQueue());
            cout << q2.deQueue();
        }
        show(q3);                           //显示为汉字
    }
```

3.5 本章小结

本章介绍了栈和队列两种数据结构的定义、顺序存储结构和链接存储结构的实现、两栈共享空间和双端队列两种特殊情况下的存储结构，以及栈和队列的典型应用。重点内容包括栈和队列的两种数据结构的特性、顺序和链接存储结构的实现，难点是栈在函数递归调用中作用的理解、表达式求值、魔王语言翻译官的实现。

本章习题

一、选择题

1. 若栈 $S1$ 中保存整数，栈 $S2$ 中保存运算符，函数 $F()$ 依次执行下述各步操作：

(1) 从 $S1$ 中依次弹出两个操作数 a 和 b；

(2) 从 $S2$ 中弹出一个运算符 op；

(3) 执行相应的运算 a op b；

(4) 将运算结果压入 $S1$ 中。

假定 $S1$ 中的操作数依次是 5,8,3,2(2 在栈顶)，$S2$ 中的运算符依次是 *，－，＋(＋在栈顶)。调用 3 次 $F()$ 后，S 栈顶保存的值是(　　)。

A. －15　　B. 15　　C. －20　　D. 20

2. 现有队列 Q 与栈 S，初始时 Q 中的元素依次是 1,2,3,4,5,6(1 在队头)，S 为空。若仅允许下列 3 种操作：①出队并输出出队元素；②出队并将出队元素入栈；③出栈并输出出栈元素，则不能得到的输出序列是(　　)。

A. 1,2,5,6,4,3　　B. 2,3,4,5,6,1

C. 3,4,5,6,1,2　　D. 6,5,4,3,2,1

3. 为解决计算机主机与打印机之间的速度不匹配问题，通常设置一个打印数据缓冲区，主机将要输出的数据依次写入该缓冲区，而打印机则依次从该缓冲区中取出数据。该缓冲区的逻辑结构应该是(　　)。

A. 栈　　B. 队列　　C. 树　　D. 图

4. 某队列允许在其两端进行入队操作，但仅允许在一端进行出队操作。若元素 a,b,c,d,e 依次入此队列后再进行出队操作，则不可能得到的出队序列是(　　)。

A. b,a,c,d,e　　B. d,b,a,c,e

C. d,b,c,a,e　　D. e,c,b,a,d

5. 中缀表达式$(A+B)*(C-D)/(E-F*G)$的后缀表达式是(　　)。

A. $A+B*C-D/E-F*G$　　B. $AB+CD-*EFG*-/$

C. $AB+C*D-E/F-G*$　　D. $ABCDEFG+*-/-*$

6. 与中缀表达式 $a*b+c/d-e$ 等价的前缀表达式是(　　)。

A. $-+*ab/cde$　　B. $*+/-abcde$　　C. $abcde*+/-$　　D. $+*ab-/cde$

7. 有六个元素按 6,5,4,3,2,1 的顺序进栈，下列(　　)不是合法的出栈序列。

A. 5 4 3 6 1 2　　B. 4 5 3 1 2 6　　C. 3 4 6 5 2 1　　D. 2 3 4 1 5 6

8. 向一个栈顶指针为 h 的带头结点的链栈中插入指针 s 所指的结点时,应执行(　　)。

A. h->next=s;　　B. s->next=h;

C. s->next=h; h->next=s;　　D. s->next=h->next; h->next=s;

9. 设顺序队列的容量为 MaxSize,其头指针为 front,尾指针为 rear,空队列的条件为(　　)。

A. front=rear　　B. front=MaxSize　　C. front+1=rear　　D. rear=0

10. 已知一个算术表达式的中缀形式为 $A+B*C-D/E$,后缀形式为 $ABC*+DE/-$,则其前缀形式为(　　)。

A. $-A+B*C/DE$　　B. $-A+B*CD/E$

C. $-+*ABC/DE$　　D. $-+A*BC/DE$

二、填空题

1. 若某堆栈初始为空,PUSH 与 POP 分别表示对栈进行一次进栈与出栈操作,那么,对于输入序列 a,b,c,d,e,经过 PUSH,PUSH,POP,PUSH,POP,PUSH,PUSH 以后,输出序列是_____。

2. 栈是_____的线性表,其运算遵循_____的原则。

3. 堆栈是一种操作受限的线性表,它只能在线性表的_____进行插入和删除操作,对栈的访问是按照_____的原则进行的。

4. 当两个栈共享一个存储区时,栈利用一维数组 stack(1,n)表示,两栈顶指针为 top[1]与 top[2],则当栈 1 空时,top[1]为_____,当栈 2 空时,top[2]为_____,栈满时为_____。

5. 表达式 23+((12*3-2)/4+34*5/7)+108/9 的后缀表达式是_____。

6. 用 S 表示入栈操作,X 表示出栈操作,若元素入栈顺序为 1,2,3,4,则为了得到 1,3,4,2 的出栈顺序,相应的 S 和 X 操作串为_____。

7. 在具有 n 个元素的非空队列中插入一个元素或者删除一个元素的操作的时间复杂度采用大 O 形式表示为_____。

8. 循环队列是队列的一种_____存储结构。

9. 栈和队列的共同特点是只允许在端点处进行_____和_____。

10. 用带头结点的单链表表示栈,则栈空的标志是_____。

三、应用题

1. 有 5 个元素,其入栈次序为 A,B,C,D,E,在各种可能的出栈序列中,第一个出栈元素为 C 且第二个出栈元素为 D 的出栈序列有哪几个?

2. 用栈实现将中缀表达式 8-(3+5)*(5-6/2)转换成后缀表达式,画出栈的变化过程图。

3. 请设计一个队列,满足以下要求:①初始时队列为空;②入队时,允许增加队列占用空间;③出队后,出队元素占用的空间可重复使用,即队列所占用的空间只增不减;④入队和出队操作时间复杂度始终保持 $O(1)$。

(1) 该队列应使用链式存储还是顺序存储结构?

(2) 画出该队列的初始状态,并给出队空和队满的判断条件。

(3) 画出第一个元素入队后的队列状态。

(4) 画出入队和出队操作的基本过程。

4. 以栈为工具，将十进制 9027 转化为八进制数，画出运算过程中栈中元素的变化过程。

四、算法设计题

1. 如果用一个循环数组 $q[0\cdots m-1]$表示队列，该队列只有一个队列头指针 front，不设队列尾指针 rear，而设置计数器 count 用以记录队列中结点的个数。

(1) 编写实现队列的三个基本运算：判空、入队、出队。

(2) 队列中能容纳元素的最多个数是多少？

2. 设有两个栈 $S1$、$S2$ 都采用顺序栈方式，并且共享一个存储区[O…maxsize−1]。为了尽量利用空间，减少溢出的可能，可采用栈顶相向迎面增长的存储方式。试设计 $S1$、$S2$ 有关入栈和出栈的操作算法。

3. 假设称正读和反读都相同的字符序列为“回文”，例如，'abcba'是回文，'abcde'和'ababab'则不是回文。试写一个算法判别读入的一个以“@”为结束符的字符序列是否是“回文”。

4. 已知有 n 个元素存放在向量 $\boldsymbol{S}[1..n]$中，其值各不相同，请写一递归算法，生成并输出 n 个元素的全排列。

扩展阅读：排队论

本章学习了队列这种特殊的线性结构，它可以解决许多实际问题。日常生活中许多地方都能见到排队现象，例如在超市的收银台、高速公路收费站等。排队现象对人们的生活有重要的影响，因此在数学上，有一个专门的分支来研究各种有形和无形的排队现象，称为排队论(queuing theory)。

排队论起源于 20 世纪初，丹麦数学家和电气工程师埃尔朗(A. K. Erlang)用概率论方法研究电话通话问题时提出了排队问题。20 世纪 30 年代中期，费勒(W. Feller)引进了生灭过程，排队论也被数学界承认是一门重要的学科，逐渐成为运筹学领域中的一项重要内容。

排队论是通过对服务对象到来及服务时间的统计研究，得出这些数量指标(等待时间、排队长度、忙期长短等)的统计规律，然后根据这些规律来改进服务系统的结构或重新组织被服务对象，使得服务系统既能满足服务对象的需要，又能使机构的费用最经济或某些指标最优。它是数学运筹学的分支学科，也是研究服务系统中排队现象随机规律的学科。

从数学上，一个简单的排队系统可以用符号 $X/Y/Z$ 描述。其中，X 表示顾客到来的时间间隔分布；Y 表示服务台时间的分布；Z 表示服务台个数。例如一个超市收银台，就可以用 $M/M/1$ 表示。其中，第一个 M 表示顾客到来的时间间隔服从参数为 λ 的负指数分布；第二个 M 表示收银员的收银时间也服从参数为 μ 的负指数分布；1 表示只有一个收银员在工作。基于这些假设，从概率角度可以计算出乘客的平均等待时间$\frac{\lambda}{\mu(\mu-\lambda)}$，平均逗留时间$\frac{1}{\mu-\lambda}$，平均队长$\frac{\lambda}{\mu-\lambda}$等重要的统计数据。另外，还可以在给定服务水平(例如乘客平均等待时间小于给定值)条件下，计算出应有多少个收银员同时工作等问题。

排队论同样可以用仿真方法求解,属于离散事件系统仿真的一个分支,具体的实现就是使用本章中介绍的队列,根据服务台的个数建立若干队列,再将到来的乘客分配到各个队列中,就可以模拟排队的生成和消散过程,并计算相应的统计数据值。正如第2章所介绍的,与数学方法相比,排队系统仿真虽然不具备优秀直观的解析性,但具有更好的灵活性。例如,大学中的理发店,顾客到来后发现等待人数较多,有可能会暂时离开,先去上课,下课以后再来排队。解析方法难以解决类似的灵活问题,而用仿真方法就很容易实现。