第3章

栈和队列

栈和队列是操作受限的线性表，是线性表的子集，因此栈和队列的操作也是线性表操作的子集。栈和队列在实际的工作和生活中应用非常广泛。栈的应用包括迷宫问题、表达式求值、括号匹配、进制转换等。队列的应用包括缓冲区、火车调度、航空机票的预订等。

3.1 栈

3.1.1 栈的逻辑结构

栈(stack)是一种运算受限的线性表。栈是只允许在表尾进行插入或删除操作的线性表。允许操作的一端称为栈顶，另一端称为栈底。栈允许为空，不包含任何元素的栈为空栈。

向栈中插入元素称为入栈、进栈、压栈，从栈中删除元素称为出栈、弹栈。任何时候进行出栈运算的只能是栈顶元素，任何时候进行入栈运算，入栈的元素都会成为新的栈顶。如图 3-1 所示，元素 a_1、a_2、a_3 分别入栈，如果此时出栈，则第一个出栈的元素是 a_3，最后一个入栈的元素第一个出栈，因此栈的操作特性是**后进先出**(last in first out)，也称为后进先出表。

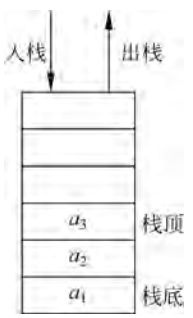


图 3-1　栈的示意图

栈作为一种特殊的线性表，具有非常广泛的应用。在日常生活中，随处可见栈的应用实例，例如一摞书，只有在顶端拿走一本书或者在顶端加入一本书最方便。在高级程序设计语言中，在各级函数之间调用时，需要使用栈来保存临时信息。此外，栈的应用还包括递归程序、函数调用、表达式求值及转换、括号匹配、迷宫问题求解等。

虽然对栈的操作位置做了限制，但是对栈的入栈或出栈的时间并没有限制，也就是说，只要栈里有元素，就可以出栈。只要栈里还有空间，就允许进栈。例如，假定三个元素按照 a、b、c 的次序进栈，则其出栈的次序可能是 abc、acb、bca、bac、cba 五种。假设有 n 个互不相同的元素依次进栈，则出栈的次序种数共为 $\frac{(2n)!}{(n+1)(n!)^2}$ 种。

栈的抽象数据类型定义为：

ADT Stack{

数据对象：

$D=\{a_i \mid a_i \in \text{ElemSet}, i=1,2,\cdots,n, n\geqslant 0\}$

数据关系：

$R=\{<a_i, a_{i+1}> \mid a_i \in D, a_{i+1} \in D, i=1,2,\cdots,n-1\}$

基本运算：

InitStack：初始化空栈；

DestroyStack：销毁栈；

Push：入栈，入栈成功会产生新的栈顶；

Pop：栈不空时出栈，同时返回出栈元素的值；

GetTop：栈不空时取栈顶元素；

Empty：判断栈是否为空栈；

}

可以采用顺序存储结构或者链式存储结构来存储栈。

3.1.2 栈的顺序存储结构

1. 顺序栈

栈的顺序存储结构称为**顺序栈**(sequential stack)。可以采用数组来描述顺序栈。一般将数组中下标为 0 的一端称为栈底，另一端称为栈顶，为了标识栈顶元素方便，附设一个 int 类型的栈顶指针 top。设顺序栈的存储容量为 StackSize，当栈为空时，top=−1。top 始终指向栈顶元素，当有元素入栈时，top 加 1；当有元素出栈时，top 减 1。当 top=StackSize−1 时，顺序栈满，不能再做入栈运算。图 3-2 所示分别是栈为空、入栈、出栈、栈满的各种情况。

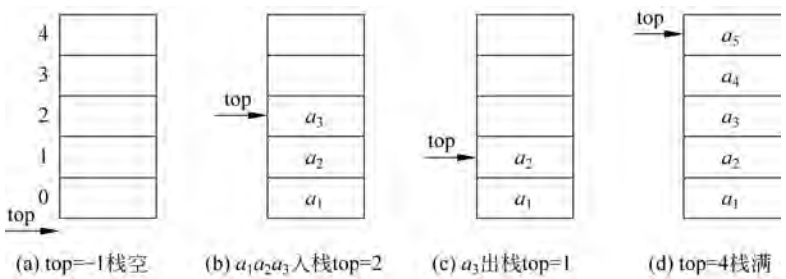


图 3-2　栈的操作示意图

2. 顺序栈的实现

可以使用 C++语言中的类模板描述顺序栈。

```
const int StackSize = 100;                    /*定义顺序栈的容量*/
template<class ElemType>                      /*定义类模板 SeqStack*/
class SeqStack{
public:
    SeqStack();                               /*构造函数,初始化空栈*/
    ~SeqStack();                              /*析构函数*/
    void Push(ElemType x);                    /*元素 x 入栈*/
    ElemType Pop();                           /*返回栈顶元素的值并出栈*/
    ElemType GetTop();                        /*返回栈顶元素*/
    int Empty();                              /*判断栈是否为空,若为空返回 1,否则返回 0*/
private:
    ElemType data[StackSize];                 /*存放栈元素的数组*/
    int top;                                  /*栈顶指针*/
};
```

1）构造函数

构造函数初始化空的顺序栈,只需要将 top 指针设为−1 即可。

2）入栈

如果顺序栈不满,则入栈时只需要将 top 加 1,再将 x 赋值给 data[top]。算法描述如下。

```
template<class ElemType>
void SeqStack<ElemType>::Push(ElemType x) {
    if(top == StackSize - 1)
        throw "顺序栈已满,上溢!";
    data[top++] = x;
}
```

入栈操作的时间复杂度为 $O(1)$。

3）出栈

如果顺序栈不为空,只需读取 top 位置处的元素,再将 top 减 1。

```
template<class ElemType>
ElemType SeqStack<ElemType>::Pop() {
    ElemType x;
    if(top == -1)
        throw "顺序栈为空,下溢!";
    x = data[top--];
    return x;
}
```

出栈操作的时间复杂度为 $O(1)$。

4）取栈顶元素

取栈顶元素与出栈操作类似,但不用修改 top 指针。

```
template <class ElemType>
ElemType SeqStack<ElemType>::GetTop() {
    if(top != -1)
        return data[top];
    else
        throw "顺序栈为空!";
}
```

取栈顶元素的时间复杂度为 $O(1)$。

顺序栈基本操作的实现详细代码可参照 ch03\SeqList 目录下的文件，此目录包含三个文件，SeqList.h 为声明 SeqList 类模板的头文件；SeqList.cpp 为类模板的实现，包含类中方法的定义；SeqListMain.cpp 为主文件，包含入口函数 main()。该实验的运行结果如图 3-3 所示。

图 3-3　顺序栈基本操作的运行结果

3. 共享栈

顺序栈要求静态分配空间，并且要求占用一片连续的存储空间。为了更灵活地利用空间，可以使两个栈共享一片存储空间。将两个栈的栈底分别设在共享空间的两端，每个栈从两端向中间延伸，称为**共享栈**。假设共享栈的空间容量为 StackSize，栈顶指针 top1 指向栈 1 的栈顶元素，栈顶指针 top2 指向栈 2 的栈顶元素，则共享栈如图 3-4 所示。

图 3-4　两栈共享空间示意图

初始化空的共享栈时，栈 1 为空，top1＝－1，栈 2 也为空，top2＝StackSize。

当两个栈的栈顶指针相遇时，共享栈满，此时 top1＋1＝top2，如图 3-5 所示。

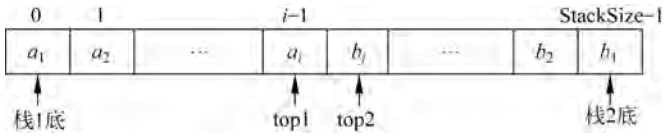


图 3-5　共享栈满的示意图

当在栈 1 入栈时，如果共享栈不满，则先将 top1 加 1，再赋值；

当在栈 2 入栈时，如果共享栈不满，则先将 top2 减 1，再赋值；

当在栈 1 出栈时，如果栈 1 不为空，则先返回 top1 处的元素值，再将 top1 减 1；

当在栈 2 出栈时，如果栈 2 不为空，则先返回 top2 处的元素值，再将 top2 加 1。

可以使用 C++语言中的类模板 SharedStack 来描述共享栈。

```
const int StackSize = 100;                    /*定义共享栈的容量*/
template <class ElemType>                     /*定义类模板 SharedStack*/
```

```
class SharedStack{
public:
    SharedStack();                          /*构造函数,共享栈的初始化*/
    ~SharedStack();                         /*析构函数*/
    void Push(int i, ElemType x);           /*元素 x 入栈*/
    ElemType Pop(int i);                    /*返回栈顶元素的值并出栈*/
    ElemType GetTop(int i);                 /*返回栈顶元素*/
    int Empty(int i);                       /*判断栈 i 是否为空,若为空返回 1,否则返回 0*/
private:
    ElemType data[StackSize];               /*存放栈元素的数组*/
    int top1;                               /*栈 1 栈顶指针*/
    int top2;                               /*栈 2 栈顶指针*/
};
```

1）构造函数

构造函数初始化空的共享栈,将栈顶指针赋初值。

```
template<class ElemType>
SharedStack<ElemType>::SharedStack() {
    top1 = -1;
    top2 = StackSize;
}
```

2）入栈

入栈运算时,使用参数 i 区分栈 1 和栈 2。

```
template<class ElemType>
void SharedStack<ElemType>::Push(int i, ElemType x) {
    if(top1 == top2 - 1)
        throw "共享栈已满,上溢!";
    if(i == 1)
        data[++top1] = x;                   /*栈 1 入栈*/
    if(i == 2)
        data[--top2] = x;                   /*栈 2 入栈*/
}
```

入栈操作的时间复杂度为 $O(1)$。

3）出栈

与入栈运算类似,使用参数 i 区分栈 1 和栈 2。

```
template<class ElemType>
ElemType SharedStack<ElemType>::Pop(int i) {
    if(i == 1) {
        if(top1 == -1)
            throw "栈 1 为空!";
        return data[top1--];
    }
    else if(i == 2) {
        if(top2 == StackSize)
```

```
            throw "栈 2 为空!";
        return data[top2++];
    }
    else
        throw "参数不合法,1 表示栈 1,2 表示栈 2!";
}
```

共享栈出栈操作的时间复杂度为 $O(1)$。

4）取栈顶

```
template < class ElemType >
ElemType SharedStack < ElemType >::GetTop(int i) {
    if(i == 1) {
        if(top1 == -1)
            throw "栈 1 为空!";
        return data[top1];
    }
    else if(i == 2) {
        if(top2 == StackSize)
            throw "栈 2 为空!";
        return data[top2];
    }
    else
        throw "参数不合法,1 表示栈 1,2 表示栈 2!";
}
```

取栈顶操作的时间复杂度为 $O(1)$。

5）判断栈空

```
template < class ElemType >
int SharedStack < ElemType >::Empty(int i) {
    if(i == 1) {
        if(top1 == -1)
            return 1;
        else
            return 0;
    }
    if(i == 2) {
        if(top2 == StackSize)
            return 1;
        else
            return 0;
    }
}
```

共享栈判空操作的时间复杂度为 $O(1)$。

使用两个栈共享一片空间，只有当一个栈增长，而另一个栈缩短时，才能真正充分地利用空间。如果两个栈总是同时增长，或者同时缩短，使用共享栈并不能减少空间的浪费，也不能降低发生溢出的概率。另外，共享栈总是使用两个栈而不是三个或更多栈，因为只有

相向增长的栈之间才能互补，如果栈太多，空间利用率不一定能够提高，反而会使问题变得更加复杂。

3.1.3　栈的链式存储结构

1. 链栈

使用链式存储结构存储的栈称为**链栈**(linked stack)。由于栈是操作受限的线性表，因此链栈也可以看成操作受限的单链表，链栈的操作为单链表操作的子集。

首先确定使用链表的表头还是表尾作为栈顶。如果使用带头指针的链表的表尾作为栈顶，如图 3-6 所示。入栈操作时，需要查找表尾的位置，因此时间复杂度是 $O(n)$；出栈操作时，由于需要查找表尾的前一个结点的位置，因此时间复杂度也是 $O(n)$。

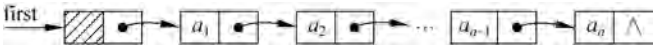


图 3-6　带头指针的单链表

如果使用带尾指针的单链表，采用表尾作为栈顶。如图 3-7 所示，则入栈操作时间复杂度为 $O(1)$，但是出栈操作的时间复杂度仍为 $O(n)$，因此，应该使用单链表的表头作为栈顶。

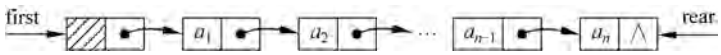


图 3-7　带尾指针的单链表

此外，由于单链表的任意一个合法的位置都可以操作，因此添加头结点以使首元结点的地址存放方法和其他元素结点一致。但是在链栈中，由于操作仅在栈顶处进行，即只在链表的表头进行，其他位置不进行插入或删除操作，因此，链栈没有必要设置头结点。按照习惯，链栈中的栈顶指针为 top 指针，当链栈非空时，如图 3-8 所示。当链栈为空时，top＝NULL。

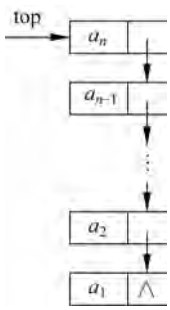


图 3-8　链栈示意图

2. 链栈的实现

可以使用 C++的类模板描述链栈。

```
template < class ElemType >
class LinkStack{
public:
    LinkStack();                        / * 构造函数,初始化空的链栈 * /
    ~LinkStack();                       / * 析构函数,释放链栈中所有结点 * /
    void Push(ElemType x);              / * x 入栈 * /
    ElemType Pop();                     / * 返回栈顶元素,并出栈 * /
    ElemType GetTop();                  / * 取栈顶 * /
    int Empty();                        / * 判断链栈是否为空 * /
private:
    Node < ElemType > * top;            / * 栈顶指针 * /
};
```

1）构造函数

初始化空栈，只需将 top 指针置为 NULL。

2）入栈

入栈操作是在栈顶的位置处插入一个新结点 s，如图 3-9 所示。算法描述如下。

```
template <class ElemType>
void LinkStack <ElemType>::Push(ElemType x) {
    s = new Node <ElemType>;
    s->data = x;
    s->next = top;
    top = s;
}
```

链栈入栈操作的时间复杂度为 $O(1)$。

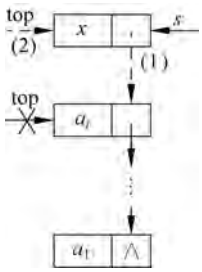


图 3-9　链栈入栈操作示意图

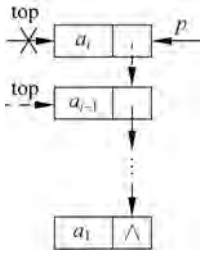


图 3-10　链栈出栈操作示意图

3）出栈

链栈出栈首先要判断栈是否为空。如果非空则在栈顶位置处删除一个结点，如图 3-10 所示。算法描述如下。

```
template <class ElemType>
ElemType LinkStack <ElemType>::Pop() {
    if(top == NULL)
        throw "链栈为空!";
    Node <ElemType> * q;
    x = top->data;
    q = top;
    top = top->next;
    delete q;
    return x;
}
```

链栈出栈操作的时间复杂度为 $O(1)$。

4）取栈顶

取栈顶的操作和出栈操作类似，但是不用修改 top 指针，也不用释放结点。算法描述如下。

```
template <class ElemType>
ElemType LinkStack <ElemType>::GetTop() {
```

```
    if(top != NULL)
        return top->data;
    else
        throw "栈为空!";
}
```

取栈顶的时间复杂度为 $O(1)$。

5）判空

算法描述如下。

```
template<class ElemType>
int LinkStack<ElemType>::Empty() {
    if(top == NULL) {
        return 1;
    }
    else {
        return 0;
    }
}
```

判空的时间复杂度为 $O(1)$。

6）析构函数

链栈的析构函数与单链表的析构函数类似，算法描述如下。

```
template<class ElemType>
LinkStack<ElemType>::~LinkStack() {
    while(top != NULL) {
        q = top;
        top = top->next;
        delete q;
    }
}
```

析构函数的时间复杂度为 $O(n)$。

链栈基本操作实现的详细代码可参照 ch03\LinkStack 目录下的文件，该实验的运行结果如图 3-11 所示。

图 3-11　链栈基本操作的运行结果

3.1.4　顺序栈和链栈的比较

除了链栈的析构函数以外，顺序栈和链栈的基本操作的时间复杂度均为 $O(1)$，因此顺序栈和链栈的比较主要在于空间性能方面。

顺序栈静态分配空间，需要预先估计栈的容量，但是不存在结构性开销。链栈动态分配空间，不存在栈满的问题，但是存在结构性开销，每个结点都需要指针域以保存后继的地址。

因此，如果栈的元素个数变化较大时，链栈较为合适；相反，如果栈的元素个数变化不大，顺序栈较为合适。

3.2 栈的应用

3.2.1 Hanoi 塔问题

Hanoi 塔问题来自一个古老的传说。据说在世界刚被创建的时候有一座钻石宝塔(塔 A)，其上有 64 个金碟，按照从大到小的次序从塔底堆至塔顶。另外有两座钻石宝塔(塔 B 和塔 C)。牧师们一直试图把塔 A 上的金碟借助塔 B 移动到塔 C 上去。要求每次只能移动一个金碟，并且任何时候都不能将较大的金碟放置到较小的金碟的上方。可采用分治法解决，将问题分解成三个小问题：

(1) 将塔 A 上的 $n-1$ 个金碟借助于塔 C 移动到塔 B 上；

(2) 将塔 A 上的一个金碟移动到塔 C 上；

(3) 将塔 B 上的 $n-1$ 个金碟借助于塔 A 移动到塔 C 上。

使用递归算法描述如下。

```
void Hanoi(int n, char A, char B, char C) {
    if(n == 1) {
        cout << A <<" 塔移到 "<< C <<" 塔"<< endl;
    }
    else {
        Hanoi(n - 1, A, C, B);
        cout << A <<" 塔移到 "<< C <<" 塔"<< endl;
        Hanoi(n - 1, B, A, C);
    }
}
```

Hanoi 塔的详细代码可参照 ch03\hanoi 目录下的文件，$n=3$ 的运行结果如图 3-12 所示，$n=4$ 的运行结果如图 3-13 所示。

图 3-12 Hanoi 塔 $n=3$ 的运行结果

图 3-13 Hanoi 塔 $n=4$ 的运行结果

3.2.2　利用顺序栈实现进制转换

利用栈可以实现进制之间的转换，例如十进制数转换成二进制、八进制到十六进制的数。把十进制数 N 转换成 n 进制数，则在 N 不为零时循环进行 N 整除以 n，$N\%n$ 入栈，$N=N/n$。然后在栈不为空时循环将栈中的元素出栈，得到的即是转换之后的结果。使用C++语言描述算法如下。

```
void convert(int N, int n) {
    while(N != 0) {
        S.Push(N % n);
        N = N / n;
    }
    while(!S.Empty()) {
        e = S.Pop();
        if(e > 9) {
            /* 当余数大于 9 时，用大写字母表示 */
            e = e + 55;
            cout <<(char)e;
        }
        else
            cout << e;
    }
}
```

详细代码可以参照 ch03\HexConversion 目录下的文件，运行结果如图 3-14 所示。

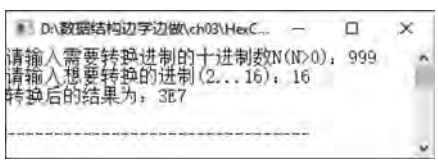


图 3-14　进制转换的运行结果

3.2.3　迷宫问题

迷宫问题的求解是实验心理学中的一个经典问题，心理学家把一只老鼠从一个没有顶盖的大盒子的入口赶进迷宫，在迷宫中通过设置很多障碍，在老鼠的前进方向上加入了多处障碍。心理学家在迷宫的唯一出口处放置了一块奶酪，吸引老鼠从入口寻找通路以达到出口。迷宫分为四方向迷宫或者八方向迷宫。四方向迷宫指的是当前位置的上、下、左、右可能存在通路。八方向迷宫指的是当前位置的上、下、左、右、左上、左下、右上、右下可能存在通路，此处按八方向迷宫处理。此类问题可以使用非递归方法或者递归方法解决。

1. 非递归方法

非递归方法使用栈保存走过的路径。利用栈将走过并且可以继续走下去的位置入栈，如果当前位置无路可走，则需要出栈。每次出栈后都要判断有无其他可以走的路径，如果没有，则应该继续出栈，直到所有可以走的路径走完。栈中的元素至少应该包括横坐标和纵坐标。使用C++语言描述的算法如下。

```
/*初始化迷宫*/
int maze[ROW][COL] = {
        {1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
        {1, 0, 1, 1, 1, 0, 1, 1, 1, 1},
        {1, 1, 0, 1, 0, 1, 1, 1, 1, 1},
        {1, 0, 1, 0, 0, 0, 0, 0, 1, 1},
        {1, 0, 1, 1, 1, 0, 1, 1, 1, 1},
        {1, 1, 0, 0, 1, 1, 0, 0, 0, 1},
        {1, 0, 1, 1, 0, 0, 1, 1, 0, 1},
        {1, 1, 1, 1, 1, 1, 1, 1, 1, 1}};
/*初始化标志位,0代表没走过,1代表走过*/
int mark[ROW][COL] = {0};
/*方向*/
typedef struct{
    short int vert;
    short int horiz;
}offsets;
/*当前位置的八个方向,右、下、左、上、左上、左下、右上、右下*/
offsets move[8] = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}, {-1, -1}, {1, -1}, {-1, 1}, {1, 1}};
/*迷宫类型*/
typedef struct element{
    short int row;
    short int col;
    short int dir;
}element;

/*打印迷宫*/
void printMaze(int maze[][COL], int row) {
    cout <<"迷宫为: "<< endl;
    for(i = 0; i < row; i++) {
        for(j = 0; j < COL; j++) {
            cout << maze[i][j]<<" ";
        }
        cout << endl;
    }
}

/*打印起点和终点*/
void printStartAndEnd() {
```

```
    cout <<"起点为: ("<< START_I <<","<< START_J <<")"<< endl;
    cout <<"终点为: ("<< END_I <<","<< END_J <<")"<< endl;
}

/* 迷宫函数 */
void path() {
    SeqStack < element > S;
    element position;
    found = 0;
    /* 从(1,1)开始,到(6,8)结束 */
    /* 初始化标志数组元素 */
    mark[START_I][START_J] = 1;
    start.row = START_I, start.col = START_J, start.dir = 0;
    /* 将起点入栈 */
    S.Push(start);
    while(!S.Empty() && !found) {
        position = S.Pop();              /* 将栈顶元素取出 */
        row = position.row;              /* 利用中间变量 row, col, dir 等候判断 */
        col = position.col;
        dir = position.dir;
        while(dir < 8 && !found) {
            next_row = row + move[dir].vert;
            next_col = col + move[dir].horiz;
            if(row == END_I && col == END_J)
                found = 1;
            /* 下一步可走并且没走过,则入栈 */
            else if(!maze[next_row][next_col] && !mark[next_row][next_col]) {
                mark[next_row][next_col] = 1;
                position.row = row;
                position.col = col;
                position.dir = ++dir;
                /* 合理则入栈 */
                S.Push(position);
                /* 继续向下走 */
                row = next_row;
                col = next_col;
                dir = 0;
            }
            else
                /* dir<8 时,改变方向 */
                dir++;
        }
        /* 判断是否有出口 */
        if(found) {
            SeqStack < element > R;
            /* 将终点入栈 */
            element end;
```

```
                end.row = END_I;
                end.col = END_J;
                end.dir = 0;
                S.Push(end);
                    cout <<"存在路径:"<< endl;
                    /* 利用栈 R 将栈 S 中的路径按从栈底到栈顶的顺序输出 */
                    while(!S.Empty()) {
                        element e = S.Pop();
                        R.Push(e);
                    }
                    while(!R.Empty()) {
                        element e = R.Pop();
                        cout <<"("<< e.row <<","<< e.col <<")";
                    }
            }
        }
        if(found == 0)
            cout <<"不存在路径!";
}
```

详细代码可参照 ch03\Maze 目录下的 MazeUseStackMain.cpp 文件，运行结果如图 3-15 所示。

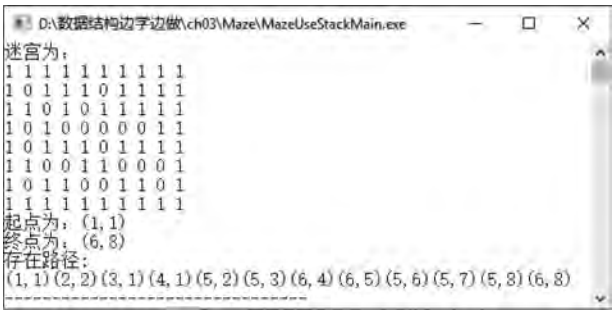


图 3-15　非递归求解迷宫问题的运行结果

2. 递归方法

也可以使用递归方法求解迷宫问题，使用 C++语言描述算法如下。

```
/* 初始化迷宫 */
int maze[ROW][COL] = {
        {1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
        {1, 0, 1, 1, 1, 0, 1, 1, 1, 1},
        {1, 1, 0, 1, 0, 1, 1, 1, 1, 1},
        {1, 0, 1, 0, 0, 0, 0, 0, 1, 1},
        {1, 0, 1, 1, 1, 0, 1, 1, 1, 1},
        {1, 1, 0, 0, 1, 1, 0, 0, 0, 1},
        {1, 0, 1, 1, 0, 0, 1, 1, 0, 1},
```

```
        {1, 1, 1, 1, 1, 1, 1, 1, 1, 1}};

/* 打印迷宫 */
void printMaze(int maze[][COL], int row) {
    cout <<"迷宫为: "<< endl;
    for(i = 0; i < row; i++) {
        for(j = 0; j < COL; j++) {
            cout << maze[i][j]<<" ";
        }
        cout << endl;
    }
}

/* 打印起点和终点 */
void printStartAndEnd() {
    cout <<"起点为: ("<< START_I <<","<< START_J <<")"<< endl;
    cout <<"终点为: ("<< END_I <<","<< END_J <<")"<< endl;
}

/* 走迷宫 */
int VisitMaze(int maze[][COL], int i, int j) {
    int end = 0;
    /* 假设能够走通,标记为 2,用来记录路径 */
    maze[i][j] = 2;
    /* 如果到达终点则将 end 置为 0 */
    if(i == END_I && j == END_J) {
        end = 1;
    }
    /* 如果没有到达终点则搜索当前所在位置的右、下、左、上、左上、左下、右上、右下方向是否能
够走通 */
    /* 右 */
    if (end != 1 && j + 1 <= END_J && maze[i][j + 1] == 0) {
        if(VisitMaze(maze, i, j + 1) == 1) {
            return 1;
        }
    }
    /* 下 */
    if (end != 1 && i + 1 <= END_I && maze[i + 1][j] == 0) {
        if(VisitMaze(maze, i + 1, j) == 1) {
            return 1;
        }
    }
    /* 左 */
    if (end != 1 && j - 1 >= START_J && maze[i][j - 1] == 0) {
        if(VisitMaze(maze, i, j - 1) == 1) {
            return 1;
        }
```

```
    }
    /* 上 */
    if (end != 1 && i-1 >= START_I && maze[i-1][j] == 0) {
        if(VisitMaze(maze, i-1, j) == 1) {
            return 1;
        }
    }
    /* 左上 */
    if (end != 1 && i - 1 >= START_I && j - 1 >= START_J && maze[i - 1][j - 1] == 0) {
        if(VisitMaze(maze, i - 1, j - 1) == 1) {
            return 1;
        }
    }
    /* 左下 */
    if (end != 1 && i + 1 <= END_I && j - 1 >= START_J && maze[i + 1][j - 1] == 0) {
        if(VisitMaze(maze, i + 1, j - 1) == 1) {
            return 1;
        }
    }
    /* 右上 */
    if (end != 1 && i - 1 >= START_I && j + 1 <= END_J && maze[i - 1][j + 1] == 0) {
        if(VisitMaze(maze, i - 1, j + 1) == 1) {
            return 1;
        }
    }
    /* 右下 */
    if (end != 1 && i + 1 <= END_I && j + 1 <= END_J &&
maze[i + 1][j + 1] == 0) {
        if(VisitMaze(maze, i + 1, j + 1) == 1) {
            return 1;
        }
    }
    /* 当四周都不通的时候将其置回 0 */
    if(end != 1) {
        maze[i][j] = 0;
    }
    return end;
}
```

详细代码可参照 ch03\Maze\MazeUseRecursionMain.cpp 文件，运行结果如图 3-16 所示。

图 3-16　递归求解迷宫问题的运行结果

3.2.4　八皇后问题

八皇后问题是一个古老而经典的问题，是回溯算法的典型案例。该问题是国际象棋棋手 Max Bazzel 提出的。八皇后问题是在 8×8 的国际象棋棋盘上，放置 8 个皇后，使任何两

个皇后都不能相互攻击，即任何两个皇后不能在同一行、同一列或者同一斜线上，图 3-17 所示为八皇后问题的一个解。八皇后问题提出后，吸引了许多著名的数学家包括 Gauss 的关注。Gauss 认为有 76 种方案，后来有人用图论的方法得到 92 种解。计算机发明后，有多种计算机语言可以解决此问题。

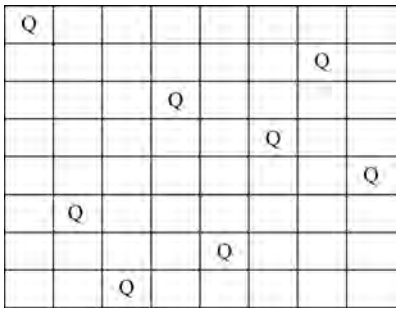


图 3-17　八皇后问题图示

求解八皇后问题，可以使用穷举法或者递归法。

1. 穷举法

穷举法是列举出所有可能的摆放位置，然后判断是否存在冲突，将不存在冲突的摆放位置打印出来。使用 C++语言描述算法如下。

```
/*打印棋盘和皇后*/
void ShowQueens(int queenArr[], int nlen, int nSolution) {
    /*解法数量*/
    cout <<"第"<< nSolution <<"种解法："<< endl;
    for(i = 0; i < nlen; i++) {
        for(j = 0; j < nlen; j++) {
            if(j == queenArr[i])
                cout <<"Q ";
            else
                cout <<"1 ";
        }
        cout << endl;
    }
    cout << endl;
}

/*判断是否符合规则*/
bool Rule(int queenArr[]) {
    for(i = 0; i <= 7; ++i) {
        for(j = 0; j <= i - 1; ++j) {
                /*判断皇后是否在同一列*/
                if(queenArr[i] == queenArr[j]) {
                    return true;
                }
                /*判断皇后是否在同一斜线上*/
                if(abs(queenArr[i] - queenArr[j]) == abs(i - j)) {
                    return true;
                }
        }
    }
    return false;
}

/*移动皇后*/
```

```
void EnumQueensPositon(int queenArr[], int &nSolution) {
    for(queenArr[0] = 0; queenArr[0] < 8; ++queenArr[0])
        for(queenArr[1] = 0; queenArr[1] < 8; ++queenArr[1])
            for(queenArr[2] = 0; queenArr[2] < 8; ++queenArr[2])
                for(queenArr[3] = 0; queenArr[3] < 8; ++queenArr[3])
                    for(queenArr[4] = 0; queenArr[4] < 8; ++queenArr[4])
                        for(queenArr[5] = 0; queenArr[5] < 8; ++queenArr[5])
                            for(queenArr[6] = 0; queenArr[6] < 8; ++queenArr[6])
                                for(queenArr[7] = 0; queenArr[7] < 8; ++queenArr[7]) {
                                    if(Rule(queenArr)) {
                                        continue;
                                    }
else {
                                        ++nSolution;
                                        ShowQueens(queenArr, 8, nSolution);
                                    }
                                }
}
```

详细代码可参照 ch03\EightQueens\ExhaustionMain.cpp 文件。运行结果罗列了 92 种解法，如图 3-18 所示。

图 3-18　穷举法求解八皇后问题的运行结果

2. 递归法

除了穷举法之外，还可以使用递归法对八皇后问题求解。使用 C++语言描述算法如下。

```
int queenArr[8], nlen = 8, nSolution = 0;
void ShowQueens() {
    /*解法数量*/
    cout <<"第"<< nSolution <<"种解法："<< endl;
    for(i = 0; i < nlen; i++) {
        for(j = 0; j < nlen; j++) {
            if(j == queenArr[i])
                cout <<"Q ";
            else
                cout <<"1 ";
        }
        cout << endl;
    }
    cout << endl;
}
void Search(int r) {
    /*8 个皇后已放置完毕*/
    if(r == nlen) {
        nSolution++;
        ShowQueens();
        return;
    }
```

```
        /* 寻找第 r 个皇后的位置 */
        for(i = 0; i < nlen; i++) {
            queenArr[r] = i;
            ok = 1;
            for(j = 0; j < r; j++) {
                if(queenArr[r] == queenArr[j] || abs(r - j) == abs(queenArr[r] - queenArr[j])) {
                    ok = 0;
                    break;
                }
            }
            /* 寻找第 r + 1 个皇后的位置 */
            if(ok)
                Search(r + 1);
        }
    }
```

详细代码可参照 ch03\EightQueens\RecursionMain.cpp，运行结果如图 3-18 所示。

3.2.5　火车调度问题

编号为 1,2,3,4 的四列火车通过一个栈式的列车调度站，可能的调度结果有哪些？如果有 n 列火车通过调度站，可能的调度结果有哪些？栈具有后进先出的特点，因此，任意一个调度结果都应该是 1,2,3,4 全排列中的一个。由于进栈的顺序是从小到大的，所以出栈的顺序应该满足以下条件：对于序列中的任意一个数，其后面的所有比它小的数都应该是倒序的。例如 4,3,2,1 是一个有效的出栈序列；而 1,4,2,3 不是一个有效的出栈序列，因为 4 后面的 2 和 3 不是倒序的。因此，只需要将 n 个数的全排列中符合出栈规则的序列输出即可。

求 n 个元素$\{r_1,r_2,\cdots,r_n\}$的全排列可以采用递归算法。设 $R=\{r_1,r_2,\cdots,r_n\}$是要进行全排列的 n 个元素，定义 $R_i=R-\{r_i\}$。集合 R 中的元素的全排列记为 perm(R)。$(r_i)\cdot$perm(R)表示在全排列 perm(R)的每一个排列前加上前缀 r_i 得到的排列。则 R 的全排列可以定义如下。

当 $n=1$ 时，perm(R)$=(r)$，此时 r 是集合 R 中唯一的元素；

当 $n>1$ 时，perm(R)由$(r_1)\cdot$perm(R_1)，$(r_2)\cdot$perm(R_2)，…，$(r_n)\cdot$perm(R_n)构成。即将集合 R 中所有的元素分别与第一个元素交换，也即总是处理后 $n-1$ 个元素的全排列。

例如，当 $n=3$，并且 $R=\{a,b,c\}$时，则：

$$\begin{aligned}\text{perm}(R)&=a\cdot\text{perm}\{b,c\}+b\cdot\text{perm}\{a,c\}+c\cdot\text{perm}\{a,b\}\\&=a\cdot b\cdot\text{perm}\{c\}+a\cdot c\cdot\text{perm}\{b\}+b\cdot a\cdot\text{perm}\{c\}\\&\quad+b\cdot c\cdot\text{perm}\{a\}+c\cdot a\cdot\text{perm}\{b\}+c\cdot b\cdot\text{perm}\{a\}\\&=\{abc,acb,bac,bca,cab,cba\}\end{aligned}$$

栈式列车调度求所有出栈序列的算法描述如下。

```
int count = 1;                          /* 满足出栈序列条件的序号 */
void Print(int array[], int n);         /* 判断 array 是否满足出栈序列条件,若满足,输出 array */
void Perm(int array[], int k, int n) {
    int i, temp;
    if(k == n - 1)
        Print(array, n);                /* k 和 n-1 相等,即一趟递归走完 */
    else {
        for(i = k; i < n; i++) {        /* 把当前结点元素与后续结点元素交换 */
            array[k] <--> array[i];     /* 交换 */
            Perm(array, k + 1, n);      /* 把下一个结点元素与后续结点元素交换 */
            arrray[i] <--> array[k];    /* 恢复原状 */
        }
    }
}

void Print(int array[], int n) {
    int i, j, k, l, m, flag = 1, b[2];
    /* 对每个 array[i] 判断其后比它小的数是否为降序 */
    for(i = 0; i < n; i++) {
        m = 0;
        for(j = i + 1; j < n && flag; j++) {
            if(array[i] > array[j]) {
                if(m == 0)
                    b[m++] = array[j];  /* 记录 array[i]后比它小的数 */
                else {
                    /* 如果之后出现的数比记录的数还大,则改变标记变量 */
                    if(array[j] > b[0])
                        flag = 0;
                    /* 否则记录这个更小的数 */
                    else
                        b[0] = array[j];
                }
              }
        }
    }
    /* 如果满足出栈规则,则输出 array */
    if(flag) {
        cout <<"第";
        cout.width(2);
        cout << count++;
        cout <<"种: ";
        for(i = 0; i < n; i++)
            cout << array[i]<<" ";
        cout << endl;
    }
}
```

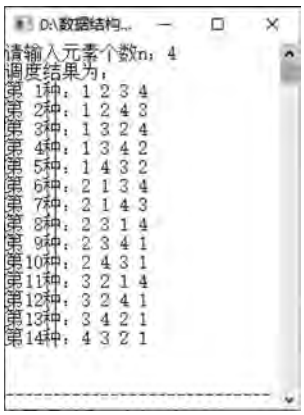


图 3-19　栈式列车调度的运行结果

详细代码可参照 ch03\TrainControl\TrainControlMain.cpp 文件，当 $n=4$ 时的运行结果如图 3-19 所示。

3.2.6　表达式括号匹配问题

假设表达式中允许的括号为()、[]、{},其嵌套顺序是任意的。可以借助栈来判断表达式中的各种括号是否匹配。使用伪代码描述的算法如下。

1. 栈S初始化为空栈;
2. 从左到右依次扫描表达式的每一个字符,执行以下操作:
   2.1 若当前字符是{、[、(,则字符入栈;
   2.2 若当前字符是}、]、),则栈顶出栈,如果与当前字符不匹配,则返回0;如果匹配,则处理下一个字符;
   2.3 若当前字符是其他字符,则处理下一个字符;
3. 若栈非空,则返回0,否则返回1;

使用C++语言描述算法如下。

```
/*中缀表达式中包括{}、[]、(),以'#'结束,判断各种括号是否匹配*/
int IsBracketMatching() {
    SeqStack<char> S;                    /*初始化空顺序栈*/
    i = 0;
    c = getchar();
    while(c != '#') {
        switch(c) {
            case '{':
            case '[':
            case '(':
                S.Push(c);
                break;
            case '}':
                if(S.Pop() != '{')
                    return 0;
                break;
            case ']':
                if(S.Pop() != '[')
                    return 0;
                break;
            case ')':
                if(S.Pop() != '(')
                    return 0;
                break;
            default:
                break;
        }
        c = getchar();
    }
```

```
        if(!S.Empty()) {
            return 0;
        }
        return 1;
    }
```

详细代码可参照 ch03\BracketMatching 目录下的文件，当输入的表达式为((22＋35＊7－{4－2}＋3]＃时的运行结果如图 3-20 所示。

当输入的表达式为[((22＋35)＊7－{4－2})＋3]＃时的运行结果如图 3-21 所示。

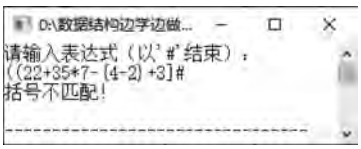


图 3-20　括号匹配的运行结果 1

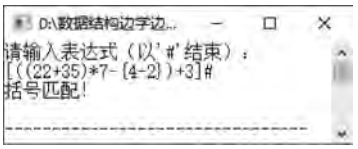


图 3-21　括号匹配的运行结果 2

3.2.7　后缀表达式求值

表达式由操作数(运算对象)、运算符(包括括号)组成。运算符分为单目运算符和双目运算符，表达式求值问题涉及的运算符一般为双目运算符。假设表达式为算术表达式，包括＋、－、＊、/、数字、()。根据运算符和操作数的位置可将表达式分为前缀表达式、中缀表达式和后缀表达式。中缀表达式指的是运算符位于两个操作数中间。例如操作数都是个位数的表达式 3＊(4＋2)/2－5。前缀表达式指的是运算符位于两个操作数之前，例如中缀表达式 3＊(4＋2)/2－5 对应的前缀表达式为 －/＊3＋4225。后缀表达式指的是运算符位于两个操作数之后，例如 342＋＊2/5－。

如果参与运算的数字是多位的，在前缀表达式和后缀表达式中为了不引起混淆，需要在相邻的两个操作数之间加特殊字符来加以区分。例如中缀表达式为 (89－60)＊(12－8)，则其对应的前缀表达式可表示为＊－89＃60＃－12＃8＃，后缀表达式可表示为 89＃60＃－12＃8＃－＊。

对于表达式求值来说，后缀表达式中已经考虑了运算的优先级，没有圆括号，只有操作数和运算符，左边的运算符的优先级高于右边运算符的优先级，即从左到右优先级依次降低。后缀表达式求值时需要用栈来保存操作数。后缀表达式求值的算法使用伪代码描述如下。

1. 初始化栈 S；
2. 从左到右依次扫描后缀表达式的每一个字符，执行下列操作：
   2.1 若当前字符是操作数，则从表达式中取连续的字符并转化成数值，入栈 S，处理下一个字符；
   2.2 若当前字符是运算符，则从栈 S 中出栈两个操作数，运算后将结果入栈；处理下一个字符；
3. 输出栈 S 的栈顶元素，即表达式的运算结果；

使用C++语言描述算法如下。

```
float PostExpression(char postexp[]) {
    SeqStack<float> S;                    /*初始化空顺序栈*/
    int i = 0;
    float a, b;
    while(postexp[i] != '\0') {
        switch(postexp[i]) {
            /*运算符+ */
            case '+':
                a = S.Pop();
                b = S.Pop();
                S.Push(a + b);
                break;
            /*运算符- */
            case '-':
                a = S.Pop();
                b = S.Pop();
                S.Push(b - a);
                break;
            /*运算符* */
            case '*':
                a = S.Pop();
                b = S.Pop();
                S.Push(a * b);
                break;
            /*运算符/ */
            case '/':
                a = S.Pop();
                b = S.Pop();
                if(a != 0) {
                    S.Push(b / a);
                }
                else {
                    throw "除零错误!";
                }
                break;
            default:
                /*处理数字字符*/
                float d = 0;
                while(postexp[i] >= '0' && postexp[i] <= '9') {
                    d = 10 * d + postexp[i] - '0';
                    i++;
                }
                S.Push(d);
                break;
        }
        i++;
    }
```

```
    return S.GetTop();
}
```

详细代码可参照 ch03\PostfixExpression 目录下的文件，当后缀表达式为 89＃60＃－12＃8＃－＊时，其对应的中缀表达式为(89－60)＊(12－8)，运行结果如图 3-22 所示。

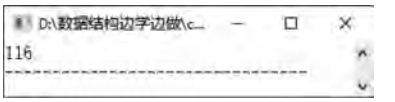


图 3-22　后缀表达式求值的运行结果

注意：前缀表达式求值的方法和后缀表达式求值的方法类似，只是其运算符的优先级与后缀表达式的顺序恰好相反，感兴趣的读者可以自行完成相应的算法。

3.2.8　中缀表达式求值

中缀表达式求值时因为要比较运算符的优先级，优先级较高的先运算，优先级较低的后运算，因此比后缀表达式求值复杂。其运算规则为：

(1) 运算符的优先级从高到低依次为()、＊、/、＋、－、＃；

(2) 有括号出现时先算括号内的，后算括号外的，多层括号由内向外进行。

中缀算术表达式求值时需要用到两个栈：操作数栈 OPND 和运算符栈 OPTR。使用伪代码描述算法如下。

1. 将栈 OPND 初始化为空，将栈 OPTR 初始化为表达式的定界符＃；
2. 从左至右扫描表达式，在没有遇到＃或者栈 OPTR 的栈顶不是＃时循环：
   2.1 若当前字符是操作数，则将连续的操作数转化成数值，入栈 OPND；
   2.2 若当前字符是运算符且优先级比栈 OPTR 的栈顶的优先级高，则入栈 OPTR，处理下一个字符；
   2.3 若当前字符是运算符且优先级比栈 OPTR 的栈顶的优先级低，则从栈 OPND 出栈两个操作数，从栈 OPTR 出栈一个运算符，将运算结果入栈 OPND，继续处理当前字符；
   2.4 若当前字符是运算符且优先级和栈 OPTR 的栈顶的优先级相同，则将栈 OPTR 的栈顶出栈，处理下一个字符；
3. 输出栈 OPND 的栈顶元素，即表达式的计算结果；

使用 C++语言描述算法如下。

```
char OperaterSet[OperaterSetSize] = {'+', '-', '*', '/', '(', ')', '#'};
/*运算符间的优先关系*/
unsigned char Prior[7][7] = {
    '>', '>', '<', '<', '<', '>', '>',
    '>', '>', '<', '<', '<', '>', '>',
    '>', '>', '>', '>', '<', '>', '>',
    '>', '>', '>', '>', '<', '>', '>',
    '<', '<', '<', '<', '<', '=', ' ',
```

```
    '>', '>', '>', '>', ' ', '>', '>',
    '<', '<', '<', '<', '<', ' ', '='
};

/* 判断 c 是否是运算符 */
int IsOperator(char c) {
    int flag = 0;
    for(i = 0; i < OperaterSetSize; i++) {
        if(c == OperaterSet[i]) {
            flag = 1;
            break;
        }
    }
    return flag;
}

/* 返回运算符 oper 在运算符数组中的序号 */
int ReturnOpOrd(char oper) {
    for(i = 0; i < OperaterSetSize; i++) {
        if (oper == OperaterSet[i]) {
        return i;
        }
    }
    return -1;
}

/* 比较两个运算符的优先级,返回字符>, <, = */
char Priority(char c1, char c2) {
    i = ReturnOpOrd(c1);
    j = ReturnOpOrd(c2);
    return Prior[i][j];
}

/* 符号运算函数,只有 +, -, *, / */
double Operate(double a, unsigned char c, double b) {
    switch(c) {
        case '+':
            return a + b;
        case '-':
            return a - b;
        case '*':
            return a * b;
        case '/':
            return a / b;
        default:
            return 0;
    }
```

```
}

/*算术表达式求值的算符优先算法*/
float EvaluateInExpression() {
    SeqStack<char> OPTR; /*运算符栈*/
    SeqStack<float> OPND; /*操作数栈*/
    char tmp[20]; /*临时数据,用于将数字字符串转化成整数数值*/
    float data, a, b;
    char oper, c, cton[2];
    OPTR.Push('#');
    /*将 tmp 置为空*/
    strcpy(tmp, "\0");
    c = getchar();
    while (c != '#' || OPTR.GetTop() != '#') {
        /*c 是操作数*/
        if(!IsOperator(c)){
            cton[0] = c;
            cton[1] = '\0'; /*存放单个数*/
            strcat(tmp, cton); /*将单个数连接到 tmp 中,形成字符串*/
            c = getchar();
            /*如果遇到运算符,则将字符串 tmp 转换成实数,入栈,并重新置空*/
            if(IsOperator(c)){
                data = (float)atof(tmp);
                OPND.Push(data);
                strcpy(tmp, "\0");
            }
        }
        /*c 是运算符*/
        else{
            switch(Priority(OPTR.GetTop(), c)) {
            case '<': /*栈顶元素优先权低*/
                OPTR.Push(c);
                c = getchar();
                break;
            case '=': /*脱括号并接收下一字符*/
                OPTR.Pop();
                c = getchar();
                break;
            case '>': /*退栈并将运算结果入栈*/
                oper = OPTR.Pop();
                b = OPND.Pop();
                a = OPND.Pop();
                OPND.Push(Operate(a, oper, b));
                break;
            default:
                break;
            }
```

```
            }
        }
        return OPND.GetTop();
}
```

详细代码可参照 ch03\InExpression 目录下的文件，当中缀表达式为(89－60)＊(12－8)＃时的运行结果如图 3-23 所示。

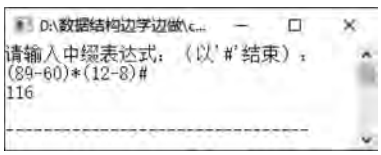


图 3-23　中缀表达式求值的运行结果

3.2.9　中缀表达式转换为后缀表达式

为了处理问题方便，编译程序常将中缀表达式转换成后缀表达式。中缀表达式求值时也可先将其转换成后缀表达式，然后按后缀表达式求值。中缀表达式转换成后缀表达式的算法使用伪代码描述如下。

1. 栈 S 初始化为空；
2. 从左到右依次扫描表达式的每一个字符，执行以下操作：
   2.1 若当前字符是操作数，则处理连续的操作数并输出，用＃分隔，处理下一个字符；
   2.2 若当前字符是运算符并且优先级比栈 S 的栈顶运算符的优先级高，则将该字符入栈 S，处理下一个字符；
   2.3 若当前字符是运算符并且优先级比栈 S 的栈顶运算符的优先级低，则将栈 S 的栈顶元素出栈并输出，继续处理当前字符；
   2.4 若当前字符是运算符并且优先级和栈 S 的栈顶运算符的优先级相等，则将栈 S 的栈顶元素出栈，处理下一个字符；

使用 C++语言描述算法如下。

```
char OperaterSet[OperaterSetSize] = {'+', '-', '*', '/', '(', ')', '#'};
/*运算符间的优先关系*/
unsigned char Prior[7][7] = {
    '>', '>', '<', '<', '<', '>', '>',
    '>', '>', '<', '<', '<', '>', '>',
    '>', '>', '>', '>', '<', '>', '>',
    '>', '>', '>', '>', '<', '>', '>',
    '<', '<', '<', '<', '<', '=', ' ',
    '>', '>', '>', '>', ' ', '>', '>',
    '<', '<', '<', '<', '<', ' ', '='
};

/*判断 c 是否是运算符*/
int IsOperator(char c) {
    int flag = 0;
```

```
    for(i = 0; i < OperaterSetSize; i++) {
        if(c == OperaterSet[i]) {
            flag = 1;
            break;
        }
    }
    return flag;
}

/* 返回运算符 oper 在运算符数组中的序号 */
int ReturnOpOrd(char oper) {
    for(i = 0; i < OperaterSetSize; i++) {
          if (oper == OperaterSet[i]) {
          return i;
        }
    }
    return -1;
}

/* 比较两个运算符的优先级,返回字符>,<,= */
char Priority(char c1, char c2) {
    i = ReturnOpOrd(c1);
    j = ReturnOpOrd(c2);
    return Prior[i][j];
}

/* 符号运算函数,只有+,-,*,/ */
double Operate(double a, unsigned char c, double b) {
    switch (c) {
        case '+':
            return a + b;
        case '-':
            return a - b;
        case '*':
            return a * b;
        case '/':
            return a / b;
        default:
            return 0;
    }
}

/* 算术表达式求值的运算符优先算法 */
void InToPostfix() {
    SeqStack<char> OPTR;
    OPTR.Push('#');
    char c, oper, cton[2];
    char tmp[10];
    float data;
```

```
        c = getchar();
        /* 将 tmp 置为空 */
        strcpy(tmp, "\0");
        while (c != '#' || OPTR.GetTop() != '#') {
            /* c 是操作数,将操作数用#分隔并输出 */
            if(!IsOperator(c)) {
                cton[0] = c;
                cton[1] = '\0'; /* 存放单个数 */
                strcat(tmp, cton); /* 将单个数连接到 tmp 中,形成字符串 */
                c = getchar();
                /* 如果遇到运算符,则将字符串 tmp 转换成浮点数,入栈,并重新置空 */
                if(IsOperator(c)) {
                    data = (float)atof(tmp);
                    cout << data;
                    cout <<"#";
                    strcpy(tmp, "\0");
                }
            }
            /* c 是运算符 */
        else {
                switch(Priority(OPTR.GetTop(), c)) {
                    case '<': /* 栈顶元素优先权低 */
                        OPTR.Push(c);
                        c = getchar();
                        break;
                    case '=': /* 脱括号并接收下一字符 */
                        OPTR.Pop();
                        c = getchar();
                        break;
                    case '>': /* 退栈并输出,继续处理当前字符 */
                        oper = OPTR.Pop();
                        cout << oper;
                        break;
                    default:
                        break;
                }
            }
        }
    }
```

详细代码可参照 ch03\InToPostfix 目录下的文件,当输入的表达式为 3*(4+2)/2-5# 时,运行结果如图 3-24 所示。

当输入的表达式为(89-60)*(12-8)#时,运行结果如图 3-25 所示。

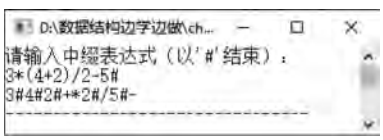


图 3-24　中缀表达式转换为后缀表达式的运行结果 1

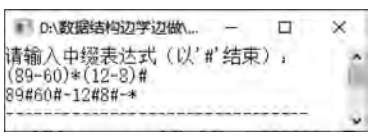


图 3-25　中缀表达式转换为后缀表达式的运行结果 2

3.3 队列

3.3.1 队列的逻辑结构

队列是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。允许插入的一端称为队尾，允许删除的一端称为队头。插入操作也称为入队或进队，删除操作也称为出队。如图 3-26 所示，元素 a_1、a_2、a_3、a_4 依次入队，此时，表头为 a_1，表尾为 a_4。如果要出队，则出队次序也是 a_1、a_2、a_3、a_4，即先入队的元素先出队，队列具有先进先出(first in first out)的特性，也称为先进先出表。

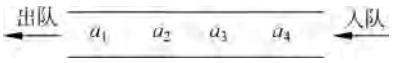


图 3-26 队列示意图

队列在实际生活和工作中具有广泛的应用，例如银行排队、售票排队、计算机的 CPU 和外设之间的缓冲区、操作系统中的作业调度等。而火车调度中的车厢重组，需要同时用到栈和队列两种数据结构。队列的抽象数据类型定义为：

ADT Queue{

数据对象：

$D=\{a_i \mid a_i \in \text{ElemSet}, i=1,2,\cdots,n, n\geqslant 0\}$

数据关系：

$R=\{<a_i, a_{i+1}> \mid a_i \in D, a_{i+1} \in D, i=1,2,\cdots,n-1\}$

基本运算：

InitQueue：初始化空队列；

DestroyQueue：销毁队列；

EnQueue：入队，入队成功会产生新的队尾；

DeQueue：队列不空时出队，同时返回出队元素的值；

GetQueue：队列不空时取队头元素；

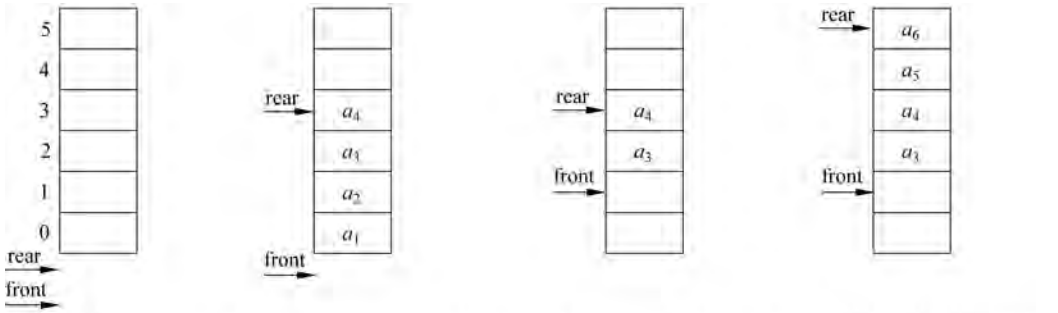
Empty：判断队列是否为空；

}

3.3.2 顺序队列

可以采用顺序存储结构或者链式存储结构存储队列。使用顺序存储结构存储的队列称为**顺序队列**(sequential queue)。可采用一维数组描述顺序队列。按照习惯，将数组的低端设为队头，为了便于访问队头元素和队尾元素，设置两个指针 front 和 rear。front 指向队头元素的前一个位置，rear 指向队尾元素，如图 3-27 所示。假设顺序队列的容量为 6，其中，图 3-27(a)为顺序队列为空，front=rear=−1；图 3-27(b)为元素 a_1、a_2、a_3、a_4 依次入队，front=−1，rear=3；图 3-27(c)为元素 a_1、a_2 依次出队，front=1，rear=3。由图可见，当入

队时，rear 加 1，front 不变；当出队时，front 加 1，rear 不变。随着入队和出队操作的进行，队列中的元素会逐渐向着下标较大的一方移动，如果到达图 3-27(d)的状态时，rear 指向了数组中最大的下标，此时顺序队列已满，无法再进行入队操作。即使此时数组的低端仍有空闲空间也无法再利用，此种现象称为**假溢出**。为了解决顺序队列假溢出的问题，引入了循环队列。



(a) 顺序队列为空　(b) 元素a_1、a_2、a_3、a_4依次入队　(c) 元素a_1、a_2依次入队　(d) rear指向数组中最大的下标

图 3-27　顺序队列操作示意图

3.3.3　循环队列

1. 循环队列概述

循环队列(circular queue)是将队列想象成一个首尾相接的循环结构，即将数组中的 0 号单元看成是数组中最大的下标单元的下一个单元，如图 3-28 所示。

实际中并不存在循环的内存结构，循环队列只是借助软件方法实现数组最大的下标到 0 之间的过渡。可以利用取模运算实现，假设队列的容量为 QueueSize，则入队时 rear＝(rear＋1) % QueueSize，出队时 front＝(front＋1) % QueueSize。要使用循环队列解决实际问题，还需要区分循环队列空和循环队列满的问题。因为当循环队列空时，不能出队和取队头；当循环队列满时，不能入队。如图 3-29 所示，图 3-29(a)为队列空的情况，此时 front＝rear，图 3-29(c)为图 3-29(b)继续入队两个元素以后的队列满的状态，也满足 front＝rear，因此普通的循环队列出现了队空和队满条件相同的问题。

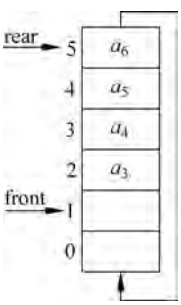
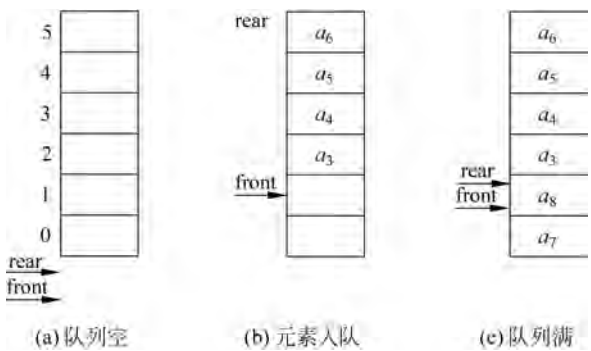


图 3-28　循环队列示意图



(a) 队列空　(b) 元素入队　(c) 队列满

图 3-29　普通循环队列空和队列满条件相同示意图

要区分循环队列空和队列满的条件只需要浪费一个存储单元，即当循环队列中还剩余一个空闲的存储单元时就认为队列已经满了。当循环队列的容量为 QueueSize 时，实际只能存储 QueueSize－1 个元素。如图 3-30 所示，图 3-30(a)为循环队列空的情况，此时满足的条件仍是 front＝rear。图 3-30(b)和图 3-30(c)都是循环队列满的情况，此时满足的条件为(rear＋1) % QueueSize＝front。循环队列中包括的元素个数可表示为(rear－front＋QueueSize)%QueueSize。

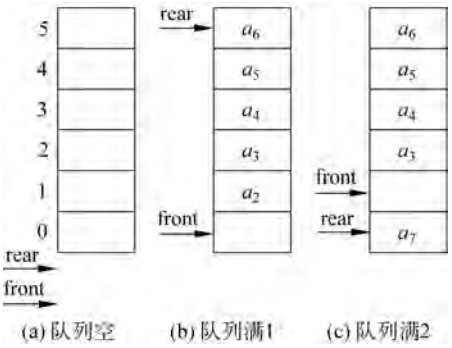


图 3-30　循环队列空和队列满的情况

2. 循环队列的实现

可以使用 C++语言的类模板 CirQueue 描述循环队列。

```
const int QueueSize = 100;              /*定义循环队列的容量*/
template<class ElemType>                /*定义类模板 CirQueue*/
class CirQueue{
public:
    CirQueue();                         /*构造函数,初始化循环队列*/
    ~CirQueue();                        /*析构函数*/
    void EnQueue(ElemType x);           /*入队操作*/
    ElemType DeQueue();                 /*出队操作,返回出队的元素*/
    ElemType GetQueue();                /*取队头操作*/
    int Length();                       /*返回循环队列的元素个数*/
    int Empty();                        /*判断循环队列是否为空,若为空则返回 1,否则返回 0*/
private:
    ElemType data[QueueSize];           /*存放循环队列元素的数组*/
    int front,rear;                     /*队头,队尾指针*/
};
```

1) 构造函数

构造函数初始化空的循环队列，只需要将 front 和 rear 同时指向一个位置，可以指向 0～QueueSize－1 的任意值，通常设为 QueueSize－1。算法描述如下。

```
template<class ElemType>
CirQueue<ElemType>::CirQueue() {
    front = QueueSize - 1;
    rear = QueueSize - 1;
}
```

2) 入队

当队列不满时，将队尾指针循环后移一个位置，然后给新元素赋值。算法描述如下。

```
template<class ElemType>
void CirQueue<ElemType>::EnQueue(ElemType x) {
```

```
    if((rear + 1) % QueueSize == front)
        throw "循环队列已满,上溢!";
    rear = (rear + 1) % QueueSize;
    data[rear] = x;
}
```

3）出队

当循环队列不为空时,先将 front 指针循环后移一个位置,然后返回 front 指针处的元素值。算法描述如下。

```
template <class ElemType>
ElemType CirQueue<ElemType>::DeQueue() {
    if(rear == front)
        throw "循环队列为空!";
    front = (front + 1) % QueueSize;
    return data[front];
}
```

4）取队头

取队头算法与出队算法类似,区别在于 front 指针不变化。算法描述如下。

```
template <class ElemType>
ElemType CirQueue<ElemType>::GetQueue() {
    if(rear == front)
        throw "循环队列为空!";
    return data[(front + 1) % QueueSize];
}
```

5）返回循环队列中元素的个数

```
template <class ElemType>
int CirQueue<ElemType>::Length() {
    return (rear - front + QueueSize) % QueueSize;
}
```

6）判断循环队列是否为空

```
template <class ElemType>
int CirQueue<ElemType>::Empty() {
    if(front == rear)
        return 1;
    else
        return 0;
}
```

循环队列的以上算法的时间复杂度都为 $O(1)$。循环队列基本操作实现的详细源代码可参照 ch03\CirQueue 目录下的文件,运行结果如图 3-31 所示。

图 3-31　循环队列基本操作的运行结果

3.3.4 双端队列

1. 双端队列概述

双端队列(double ended queue,简称 deque)是限定在线性表的两端(left 端和 right 端)都可以进行插入和删除操作的线性表,如图 3-32 所示。

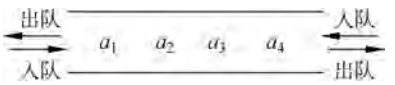


图 3-32 双端队列示意图

可以采用顺序存储结构存储双端队列。双端队列的实现和循环队列类似,可以附设 front 和 rear 两个指针记录队列中队头和队尾的位置。front 指向队头元素的前一个位置,rear 指向队尾元素。为了区分队列满和队列空的条件,同样浪费一个存储单元。当队列空时,front＝rear;当队列满时(rear＋1) % QueueSize＝front。与循环队列不同的是,双端队列既可以在左端入队,也可以在右端入队;类似地,双端队列既可以在左端出队,也可以在右端出队。

2. 双端队列的实现

可采用 C++语言的类模板 Deque 描述双端队列。

```
const int QueueSize = 100;              /*定义双端队列的容量*/
/*定义类模板 Deque*/
template<class ElemType>
class Deque{
public:
    Deque();                            /*构造函数,双端队列的初始化*/
    ~Deque();                           /*析构函数*/
    void EnQueue(int i, ElemType x);    /*入队操作,i=0 表示左端,i=1 表示右端*/
    ElemType DeQueue(int i);            /*出队操作,i=0 表示左端,i=1 表示右端*/
    ElemType GetQueue(int i);           /*取队头操作,i=0 表示左端,i=1 表示右端*/
    int Length();                       /*返回双端队列中包含的元素个数*/
    int Empty();                        /*判断双端队列是否为空,若为空则返回 1,否则返回 0*/
    void Print();                       /*打印双端队列中的元素*/
private:
    ElemType data[QueueSize];           /*存放双端队列元素的数组*/
    int front, rear;                    /*队头,队尾指针,设定与循环队列相同*/
};
```

1) 构造函数

双端队列的构造函数与循环队列构造函数类似,将 front 和 rear 同时指向 QueueSize－1。

2) 入队

当双端队列不满时,类似于循环队列,在左端入队时,front 循环前移,front＝(front－1＋QueueSize) % QueueSize;在右端入队时,rear 循环后移,rear＝(rear＋1) % QueueSize。算法描述如下。

```
template<class ElemType>
```

```
void Deque<ElemType>::EnQueue(int i,ElemType x) {
    if((rear + 1) % QueueSize == front)
        throw "上溢";
    /*左端入队*/
    if(i == 0) {
        data[front] = x;
        front = (front - 1 + QueueSize) % QueueSize;
    }
    /*右端入队*/
    if(i == 1) {
        rear = (rear + 1) % QueueSize;
        data[rear] = x;
    }
}
```

3）出队

当双端队列不空时，在左端出队时，front 循环后移；在右端出队时，rear 循环前移。算法描述如下。

```
template<class ElemType>
ElemType Deque<ElemType>::DeQueue(int i) {
    if(rear == front)
        throw "下溢";
    /*左端出队*/
    if(i == 0) {
        front = (front + 1) % QueueSize;
        x = data[front];
    }
    /*右端出队*/
    if(i == 1) {
        x = data[rear];
        rear = (rear-1 + QueueSize) % QueueSize;
    }
    return x;
}
```

4）取队头

```
template<class ElemType>
ElemType Deque<ElemType>::GetQueue(int i) {
    if(front == rear)
        throw "下溢";
    /*左端*/
    if(i == 0) {
        x = data[(front + 1) % QueueSize];
    }
    /*右端*/
    if(i == 1) {
        x = data[rear];
    }
```

```
    return x;
}
```

5）求双端队列中的元素个数

```
template <class ElemType>
int Deque<ElemType>::Length() {
    return (rear - front + QueueSize) % QueueSize;
}
```

6）判空

```
template <class ElemType>
int Deque<ElemType>::Empty() {
    if(front == rear)
        return 1;
    else
        return 0;
}
```

7）遍历双端队列

```
template <class ElemType>
void Deque<ElemType>::Print() {
    if(front == rear)
        cout <<"双端队列为空!"<< endl;
    else {
        i = (front + 1) % QueueSize;
        j = (rear + 1) % QueueSize;
        while(i != j) {
            cout << data[i]<<" ";
            i = (i + 1) % QueueSize;
        }
        cout << endl;
    }
}
```

该算法的时间复杂度为 $O(n)$。

由以上算法可知，除了遍历双端队列以外，其他操作的时间复杂度都为 $O(1)$。双端队列基本操作实现的详细代码可参照 ch03\Deque 目录下的文件，运行结果如图 3-33 所示。

图 3-33　双端队列基本操作的运行结果

除了普通的双端队列之外，还有操作受限的双端队列。操作受限的双端队列包括输入受限的双端队列和输出受限的双端队列。输入受限的双端队列指只能在队列的一端进行输入，输出受限的双端队列指只能在队列的一端进行输出。

3.3.5　链队列

1. 链队列概述

采用链式存储结构实现的队列称为**链队列**(linked queue)。可以对单链表进行改造得到链队列,表头对应队头,表尾对应队尾。为了更方便操作,头指针为 front,指向头结点。附设指向尾结点的尾指针 rear,空链队列和非空链队列如图 3-34 所示。

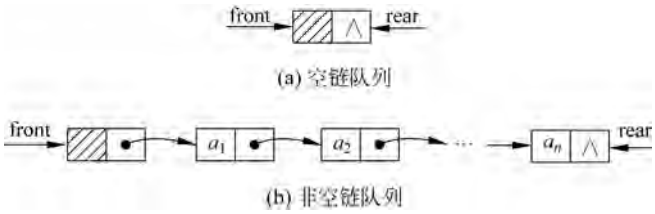


图 3-34　链队列示意图

2. 链队列的实现

可以使用 C++语言的类模板 LinkQueue 描述链队列。由于队列是操作受限的线性表,因此链队列也可以看作是操作受限的单链表。

```
template < class ElemType >
class LinkQueue{
public:
    LinkQueue();                    /* 创建只包含一个头结点的链队列 */
    ~LinkQueue();                   /* 释放链队列中所有的结点 */
    void EnQueue(ElemType x);       /* 入队操作 */
    ElemType DeQueue();             /* 出队操作 */
    ElemType GetQueue();            /* 取队头 */
    int Empty();                    /* 判断链队列是否为空 */
private:
    Node < ElemType > * front, * rear; /* 链队列的头指针和尾指针 */
};
```

1) 构造函数

构造函数初始化空的链队列,只需要申请头结点,将 front 和 rear 赋初值。算法描述如下。

```
template < class ElemType >
LinkQueue < ElemType >::LinkQueue() {
    front = new Node < ElemType >;
    front -> next = NULL;
    rear = front;
}
```

2) 入队

入队操作是在链表的表尾插入新结点 s,如图 3-35 所示。

图 3-35　链队列入队示意图

修改指针的关键语句为“rear-> next＝s；rear＝s;”，当链队列为空时，操作语句不变。算法描述如下。

```
template < class ElemType >
void LinkQueue < ElemType >::EnQueue(ElemType x) {
    s = new Node < ElemType >;
    s -> data = x;
    s -> next = NULL;
    rear -> next = s;
    rear = s;
}
```

3）出队

链队列的出队是在链表的表头删除一个结点，如果队列中包括的元素结点个数大于 1，则不会影响到 rear 指针，因为队尾元素不变。但是如果队列中只包括一个元素结点，出队以后，链队列为空，此时需要将 rear 指针指向头结点，即 rear＝front。链队列出队操作如图 3-36 所示。

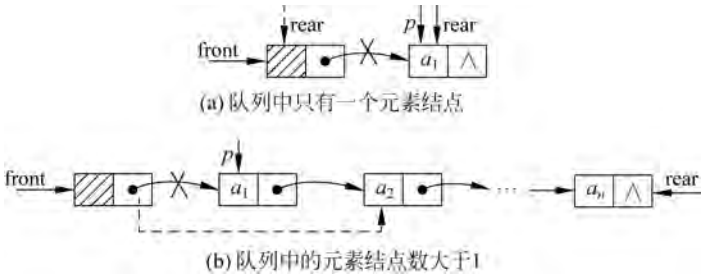


图 3-36　链队列出队示意图

算法描述如下。

```
template < class ElemType >
ElemType LinkQueue < ElemType >::DeQueue() {
    if(rear == front)
        throw "链队列为空!";
    /* p 指向队头 */
    p = front -> next;
    x = p -> data;
    front -> next = p -> next;
    /* 队列长度为 1 时,需要更改 rear 指针 */
    if(p -> next == NULL)
        rear = front;
    delete p;
```

```
    return x;
}
```

4）取队头

```
template < class ElemType >
ElemType LinkQueue < ElemType >::GetQueue() {
    if(front == rear)
        throw "链队列为空!";
    return front -> next -> data;
}
```

5）判空

```
template < class ElemType >
int LinkQueue < ElemType >::Empty() {
    if(front == rear)
        return 1;
    else
        return 0;
}
```

6）析构函数

析构函数将链队列中包括头结点在内的所有结点释放掉，执行完析构函数以后，front指针和 rear 指针都为 NULL。算法描述如下。

```
template < class ElemType >
LinkQueue < ElemType >::~LinkQueue() {
    while(front != NULL) {
        p = front;
        front = front -> next;
        delete p;
    }
    rear = NULL;
}
```

析构函数的时间复杂度为 $O(n)$，链队列的其他运算的时间复杂度都为 $O(1)$。

3.4 小结

- 栈和队列是操作受限的线性表，栈和队列的操作是线性表操作的子集。
- 由于栈和队列操作的特殊性，所以其基本运算的时间复杂度一般为 $O(1)$，例如栈的入栈、出栈，队列的入队、出队等。
- 栈是一种重要的数据结构，用途十分广泛。在递归算法中要用到系统工作栈，将递归算法改写成非递归算法时有时也需要用到栈。

- 采用顺序存储结构存储的栈为顺序栈，采用链式存储结构存储的栈为链栈。
- 栈的应用主要包括 Hanio 塔问题、进制转换、迷宫问题、八皇后问题、火车调度、表达式括号匹配、表达式求解及表达式转换等。
- 队列也是一种常用的数据结构，例如操作系统的资源分配和排队、主机和外设之间的缓冲区等。
- 循环队列的引入是为了解决顺序队列假溢出的问题。
- 双端队列是允许在线性表的两端进行插入或删除的队列。另外，还存在输入受限的双端队列和输出受限的双端队列。

习题

1. 选择题

（1）一个栈的入栈序列为 A,B,C,D,E，则栈的不可能的输出序列是（　　）。

A. $E\ D\ C\ B\ A$　　B. $D\ E\ C\ B\ A$　　C. $D\ C\ E\ A\ B$　　D. $A\ B\ C\ D\ E$

（2）若已知一个栈的入栈序列是 $1,2,3,\cdots,n$，若出栈的第一个元素为 n，则第 i 个出栈的元素是（　　）。

A. i　　B. $n-i$　　C. $n-i+1$　　D. 无法确定

（3）一个队列的入队序列是 1,2,3,4，则队列的出队序列是（　　）。

A. 4,3,2,1　　B. 1,2,3,4　　C. 1,4,3,2　　D. 3,2,4,1

（4）设计算法判断表达式的括号是否匹配，使用（　　）数据结构最好。

A. 栈　　B. 线性表　　C. 队列　　D. 数组

（5）在解决计算机的 CPU 和打印机之间速度不匹配的问题时通常设置一个缓冲区，缓冲区的数据结构是（　　）。

A. 栈　　B. 队列　　C. 数组　　D. 线性表

（6）循环队列用数组 $A[0,m-1]$ 存放其元素值，已知其头尾指针分别是 front 和 rear，则当前队列中的元素个数是（　　）。

A.（rear－front＋m）％ m　　B. rear－front＋1

C. rear－front－1　　D. rear－front

（7）栈与队列都是（　　）。

A. 链式存储的线性结构　　B. 链式存储的非线性结构

C. 限制存取点的线性结构　　D. 限制存取点的非线性结构

（8）判定一个循环队列 Q[0…QueueSize－1]为满的条件是（　　）。

A. rear＝front　　B. rear＝front＋1

C. front＝（rear＋1）％ QueueSize　　D. front＝（rear－1）％ QueueSize

（9）判定一个循环队列 Q[0…QueueSize－1]为空的条件是（　　）。

A. rear＝front　　B. rear＝front＋1

C. front=(rear+1) % QueueSize　　D. front=(rear−1) % QueueSize

(10) 在一个链队列中,假定 front 和 rear 分别为头指针和尾指针,则插入新结点 s 的操作是(　　)。

A. front=front-> next;　　B. s-> next=rear; rear=s;

C. rear-> next=s; rear=s;　　D. s-> next=front; front=s;

(11) 表达式 $a*(b+c)-d$ 的后缀表达式是(　　)。

A. $abcd*+-$　　B. $abc+*d-$　　C. $abc*+d-$　　D. $-+*abcd$

(12) 若用一个大小为 6 的数组来实现循环队列,且当前 rear 和 front 的值分别为 0 和 3,当从队列中删除一个元素,再加入两个元素后,rear 和 front 的值分别为(　　)。

A. 1 5　　B. 2 4　　C. 4 2　　D. 5 1

(13) 设栈 S 和队列 Q 的初始状态为空,元素 e_1,e_2,e_3,e_4,e_5,e_6 依次通过栈 S,一个元素出栈后即进入队列 Q,若 6 个元素出队的序列是 e_2,e_4,e_3,e_6,e_5,e_1,则栈 S 的容量至少应为(　　)。

A. 6　　B. 4　　C. 3　　D. 2

(14) 若以 1、2、3、4 作为双端队列的输入序列,则既不能由输入受限的双端队列得到,也不能由输出受限的双端队列得到的输出序列是(　　)。

A. 1234　　B. 4132　　C. 4231　　D. 4213

2. 填空题

(1) 栈的运算规则是(　　)。

(2) 循环队列的引入是为了克服(　　)。

(3) 对于栈和队列,无论它们采用顺序存储结构还是链式存储结构,进行插入和删除操作的时间复杂度都是(　　)。

(4) 共享栈满的条件是(　　)。

(5) 设 $a=6,b=4,c=2,d=3,e=2$,则后缀表达式 $abc-/de*+$ 的值为(　　)。

(6) 用带头结点的循环链表表示的队列长度为 n,若只设头指针,则出队和入队的时间复杂度为(　　)和(　　);若只设尾指针,则出队和入队的时间复杂度为(　　)和(　　)。

(7) 假设有一个空栈,栈顶指针为 1000H,现有输入序列 1,2,3,4,5,经过 PUSH,PUSH,POP,PUSH,POP,PUSH,PUSH 之后,输出序列是(　　),如果栈为顺序栈,每个元素占 4 字节,则此时栈顶指针的值是(　　)。

3. 判断题

(1) 在栈满的情况下不能做进栈操作,否则将产生“上溢”。(　　)

(2) 循环队列中至少有一个数组单元是空闲的,否则无法区分队列空和队列满的条件。(　　)

(3) 循环队列也存在空间溢出问题。(　　)

(4) 队列元素进队的顺序和出队的顺序总是一致的。(　　)

(5) 循环队列中的 front 值一定小于 rear。(　　)

(6) 栈和队列都是运算受限的线性表。(　　)

(7) 链队列出队时一定不会修改队尾指针。(　　)

（8）两个栈共享一片连续的内存空间时，为了提高内存利用率，减少溢出的可能，应把两个栈的栈底分别设在这片内存空间的两端。（　　）

（9）栈是实现过程和函数等子程序所必需的结构。（　　）

（10）栈和队列的存储方式，既可以是顺序存储方式，也可以是链式存储方式。（　　）

4. 问答题

（1）顺序队列为什么会产出假溢出？有几种解决方法？

（2）什么是递归程序？递归程序的优点和缺点是什么？递归程序在执行时，应借助什么来完成？递归程序的入口语句、出口语句一般使用什么语句来完成？

（3）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，并且第二个出栈元素为 D 的出栈序列有哪些？

（4）队列可以使用循环单链表实现，可以只设置头指针或者尾指针，哪种方案更合适？

5. 算法设计题

（1）设计求整型数组 $r[1\cdots n]$ 最大值的递归算法。

（2）已知 Ackerman 函数定义如下：

$$\mathrm{akm}(m,n)\begin{cases} n+1 & m=0 \\ \mathrm{akm}(m-1,1) & m\neq 0,n=0 \\ \mathrm{akm}(m-1,\mathrm{akm}(m,n-1)) & m\neq 0,n\neq 0 \end{cases}$$

根据定义，写出递归算法。