CHAPTER 3

第3章 栈和队列

栈和队列是两种常用的数据结构，它们的数据元素的逻辑关系也是线性关系，但在运算上不同于线性表。

本章的学习要点如下：

（1）栈、队列和线性表的异同，栈和队列抽象数据类型的描述方法。

（2）顺序栈的基本运算算法设计方法。

（3）链栈的基本运算算法设计方法。

（4）顺序队的基本运算算法设计方法。

（5）链队的基本运算算法设计方法。

（6）STL 中 stack(栈)、queue(队列)、deque(双端队列)和 priority_queue(优先队列)容器的应用。

（7）单调栈和单调队列的基本应用。

（8）综合运用栈和队列解决一些复杂的实际问题。

3.1 栈

本节先介绍栈的定义，然后讨论栈的存储结构和基本运算算法设计，最后通过两个综合实例说明栈的应用。

视频讲解

3.1.1 栈的定义

栈是一种只能在同一端进行插入或删除操作的线性表。在表中允许进行插入、删除操作的一端称为**栈顶**。栈顶的当前位置是动态的，可以用一个称为栈顶指针的位置指示器来指示。表的另一端称为**栈底**。当栈中没有数据元素时称为**空栈**。栈的插入操作通常称为**进栈**或**入栈**，栈的删除操作通常称为**退栈**或**出栈**。

说明：对于线性表，可以在中间和两端的任何地方插入和删除元素，而栈只能在同一端插入和删除元素。

栈的主要特点是“后进先出”，即后进栈的元素先出栈。每次进栈的元素

都放在原当前栈顶元素的前面成为新的栈顶元素，每次出栈的元素都是当前栈顶元素，栈顶元素出栈后次栈顶元素变成新的栈顶元素。栈也称为**后进先出表**。

抽象数据类型栈的定义如下：

ADT Stack

{

数据对象：

$D=\{a_i \mid 0\leqslant i\leqslant n-1, n\geqslant 0\}$

数据关系：

$R=\{r\}$

$r=\{<a_i, a_{i+1}> \mid a_i, a_{i+1}\in D, i=0, \cdots, n-2\}$

基本运算：

empty()：判断栈是否为空，若栈为空则返回真，否则返回假。

push(T e)：进栈操作，将元素 e 插入栈中作为栈顶元素。

pop(T& e)：出栈操作，取栈顶元素并退出该元素。

gettop(T&e)：取栈顶操作，取出当前的栈顶元素。

}

【例 3.1】 若元素的进栈顺序为 1234，能否得到 3142 的出栈序列？

解：为了让 3 作为第一个出栈元素，1、2、3 依次进栈，再出栈 3，接着要么 2 出栈，要么 4 进栈后出栈，第 2 次出栈的元素不可能是 1，所以得不到 3142 的出栈序列。

【例 3.2】 用 S 表示进栈操作，用 X 表示出栈操作，若元素的进栈顺序为 1234，为了得到 1342 的出栈顺序，给出相应的 S 和 X 操作串。

解：为了得到 1342 的出栈顺序，其操作过程是 1 进栈，1 出栈，2 进栈，3 进栈，3 出栈，4 进栈，4 出栈，2 出栈，因此相应的 S 和 X 操作串为 SXSSXSXX。

【例 3.3】 设 n 个元素的进栈序列是 1，2，3，…，n，通过一个栈得到的出栈序列是 p_1，p_2，p_3，…，p_n，若 $p_1=n$，则 p_i（$2\leqslant i\leqslant n$）的值是什么？

解：当 $p_1=n$ 时，说明进栈序列的最后一个元素最先出栈，此时出栈序列只有一种，即 n，$n-1$，…，2，1，或 $p_1=n$，$p_2=n-1$，…，$p_{n-1}=2$，$p_n=1$，也就是说 $p_i+i=n+1$，推出 $p_i=n-i+1$。

视频讲解

3.1.2　栈的顺序存储结构及其基本运算算法的实现

视频讲解

由于栈中元素的逻辑关系与线性表中的相同，因此可以借鉴线性表的两种存储结构来存储栈。在采用顺序存储结构存储时，用一维数组 data 来存放栈中的元素，称为**顺序栈**。顺序栈存储结构如图 3.1 所示，由于栈顶是动态变化的，为此设置一个栈顶指针 top 以反映栈的状态，约定 top 总是指向栈顶元素。为了简单，这里的 data 数组采用固定容量（容量为 MaxSize）分配方式，并且置 data[0]端作为栈底，另一端作为栈顶，其中的元素个数恰好为 top+1。

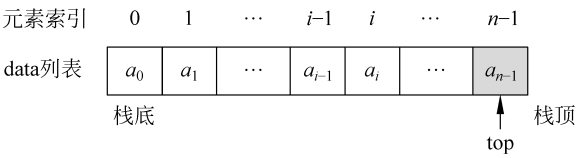


图 3.1　顺序栈的示意图

如图3.2所示为栈操作示意图,这里MaxSize=5,图3.2(a)表示一个空栈,图3.2(b)表示元素a进栈以后的状态,图3.2(c)表示数据元素b、c、d进栈以后的状态,图3.2(d)表示出栈一个元素d以后的状态。

从中看到,初始时置top=-1,对应的顺序栈的四要素如下。

① 栈空条件:top=-1。

② 栈满条件:top=MaxSize-1。

③ 元素e进栈操作:top++,data[top]=e。

④ 元素e出栈操作:e=data[top],top--。

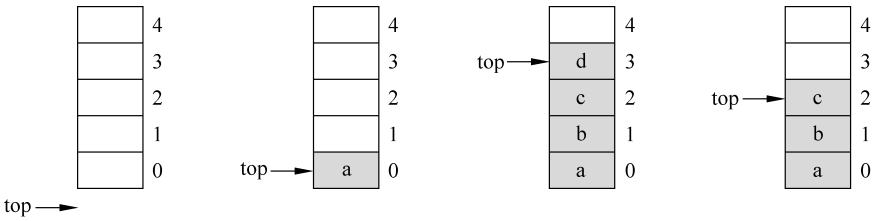


图3.2 栈操作示意图

顺序栈类模板SqStack设计如下:

```
template < typename T >
class SqStack                    //顺序栈类模板
{
    T * data;                    //存放栈中的元素
    int top;                     //栈顶指针
    //栈的基本运算算法
};
```

顺序栈的基本运算算法如下:

1) 顺序栈的初始化和销毁

通过构造函数实现初始化,即创建一个空的顺序栈;通过析构函数实现销毁,即释放顺序栈占用的空间。对应的构造函数如下:

```
SqStack()                        //构造函数
{
    data=new T[MaxSize];         //为data分配容量为MaxSize的空间
    top=-1;                      //栈顶指针初始化
}
```

对应的析构函数如下:

```
~SqStack()                       //析构函数
{
    delete [] data;              //释放data指向的空间
}
```

2) 判断栈是否为空:empty()

若top=-1成立,则表示为空栈。对应的算法如下:

```
bool empty()                    //判断栈是否为空
{
    return top==-1;
}
```

3）进栈：push(T *e*)

元素进栈只能从栈顶进入，不能从栈底或中间位置进入。在进栈中，当栈满出现上溢出时返回 false，否则先递增 top，再将元素 *e* 放置在 top 位置上，并且返回 true。对应的算法如下：

```
bool push(T e)                  //进栈算法
{
    if (top==MaxSize-1)         //栈满时返回 false
        return false;
    top++;                      //栈顶指针增 1
    data[top]=e;                //将 e 进栈
    return true;
}
```

4）出栈：pop(T& *e*)

元素出栈只能从栈顶出，不能从栈底或中间位置出栈。在出栈中，当栈空出现下溢出时返回 false，否则取出 top 位置的元素 *e*，递减 top 并且返回 *e*。对应的算法如下：

```
bool pop(T& e)                  //出栈算法
{
    if (empty())                //栈为空的情况，即栈下溢出
        return false;
    e=data[top];                //取栈顶指针位置的元素
    top--;                      //栈顶指针减 1
    return true;
}
```

5）取栈顶元素：gettop(T& *e*)

在栈不为空的条件下返回栈顶元素 *e*，不移动栈顶指针 top。对应的算法如下：

```
bool gettop(T& e)               //取栈顶元素算法
{
    if (empty())                //栈为空的情况，即栈下溢出
        return false;
    e=data[top];                //取栈顶指针位置的元素
    return true;
}
```

从以上看出，栈的各种基本运算算法的时间复杂度均为 $O(1)$。

3.1.3　顺序栈的应用算法设计示例


视频讲解

【例 3.4】 设计一个算法，利用顺序栈判断用户输入的表达式中的括号是否配对（假设表达式中可能含有圆括号、中括号和大括号），并用相关数据进行测试。

解： 因为各种括号的匹配过程遵循这样的原则，任何一个右括号与前面最靠近的未匹

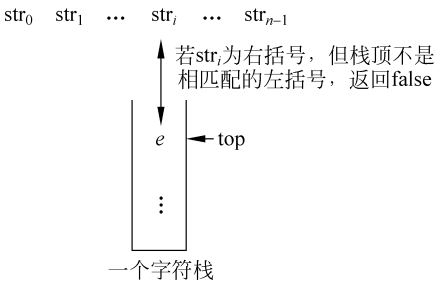


图 3.3 用一个栈判断 str 中的括号是否匹配

配的同类左括号进行匹配，所以采用一个栈来实现匹配过程。

用 str 字符串存放含有各种括号的表达式，建立一个字符顺序栈 st，用 i 遍历 str，当遇到各种类型的左括号时进栈，当遇到右括号时，若栈空或者栈顶元素不是匹配的左括号时返回 false(中途就可以确定括号不匹配)，如图 3.3 所示，否则退栈一次继续判断。当 str 遍历完毕，栈 st 为空返回 true，否则返回 false。

对应的完整程序如下：

```
#include "SqStack.cpp"                          //包含顺序栈类模板的定义
bool isMatch(string str)                        //判断表达式中的各种括号是否匹配的算法
{
    SqStack<char> st;                           //建立一个顺序栈
    int i=0;
    char e;
    while (i<str.length())
    {
        if (str[i]=='('||str[i]=='['||str[i]=='{')
            st.push(str[i]);                    //遇到左括号均进栈
        else
        {
            if (str[i]==')')                    //遇到')'
            {
                if (st.empty())return false;    //栈空时返回 false
                st.pop(e);                      //出栈元素 e
                if (e!='(') return false;       //栈顶不是匹配的'(',返回 false
            }
            if (str[i]==']')                    //遇到']'
            {
                if (st.empty()) return false;   //栈空时返回 false
                st.pop(e);                      //出栈元素 e
                if (e!='[') return false;       //栈顶不是匹配的'[',返回 false
            }
            if (str[i]=='}')                    //遇到'}'
            {
                if (st.empty())return false;    //栈空时返回 false
                st.pop(e);                      //出栈元素 e
                if (e!='{') return false;       //栈顶不是匹配的'{',返回 false
            }
        }
        i++;                                    //继续遍历 str
    }
    return st.empty();
}
int main()
{
    cout << "测试 1: ";
    string str="([)]";
```

```
    if (isMatch(str))
        cout << str << "中括号是匹配的" << endl;
    else
        cout << str << "中括号是不匹配的" << endl;
    cout << "测试 2:";
    str="([])";
    if (isMatch(str))
        cout << str << "中括号是匹配的" << endl;
    else
        cout << str << "中括号是不匹配的" << endl;
    return 0;
}
```

上述程序的执行结果如下：

```
测试 1:([)]中括号是不匹配的
测试 2:([])中括号是匹配的
```

【例 3.5】 设计一个算法，利用顺序栈判断用户输入的字符串表达式是否为回文，并用相关数据进行测试。

解：用 str 存放表达式，其中含 n 个字符，建立一个顺序栈 st，可以将 str 中的 n 个字符 $str_0, str_1, \cdots, str_{n-1}$ 依次进栈再连续出栈，得到反向序列 $str_{n-1}, \cdots, str_1, str_0$，若 str 与该反向序列相同，则是回文，否则不是回文。可以改为更高效的方法，若 str 的前半部分的反向序列与 str 的后半部分相同，则是回文，否则不是回文。判断过程如下：

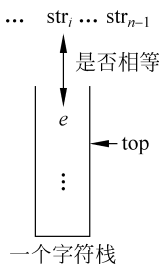


图 3.4　用一个栈判断 str 是否为回文

① 用 i 从头开始遍历 str，将前半部分字符依次进栈。

② 若 n 为奇数，i 增 1 跳过中间的字符。

③ i 继续遍历其他后半部分字符，每访问一个字符，则出栈一个字符，两者进行比较，如图 3.4 所示，若不相等返回 false。

④ 当 str 遍历完毕返回 true。

对应的完整程序如下：

```
#include "SqStack.cpp"                    //包含顺序栈类模板的定义
bool isPalindrome(string str)             //判断是否为回文的算法
{
    SqStack<char> st;                     //建立一个顺序栈
    char e;
    int i=0;
    while (i<str.length()/2)              //将 str 的前半部分字符进栈
    {
        st.push(str[i]);
        i++;                              //继续遍历 str
    }
    if (str.length()%2==1)                //str 的长度为奇数时
        i++;                              //跳过中间的字符
    while (i<str.length())                //遍历 str 的后半部分字符
    {
        if (st.empty()) false;            //栈空时返回 false
```

```
        st.pop(e);                          //出栈元素 e
        if (e!=str[i]) return false;        //若 str[i]不等于出栈字符则返回 false
        i++;
    }
    return true;                            //是回文则返回 true
}
int main()
{
    cout << "测试 1: ";
    string str="abcba";
    if (isPalindrome(str))
        cout << str << "是回文" << endl;
    else
        cout << str << "不是回文" << endl;
    cout << "测试 2: ";
    str="1221";
    if (isPalindrome(str))
        cout << str << "是回文" << endl;
    else
        cout << str << "不是回文" << endl;
    return 0;
}
```

上述程序的执行结果如下：

```
测试 1: abcba 是回文
测试 2: 1221 是回文
```

【例 3.6】 设计最小栈。定义一个栈数据结构 STACK，添加一个 Getmin()运算用于直接返回栈中的最小元素(假设栈不空)，要求 Getmin()、push()以及 pop()的时间复杂度都是 $O(1)$。例如：


视频讲解

```
push(5);   #栈元素: (5)          最小元素: 5
push(6);   #栈元素: (6,5)        最小元素: 5
push(3);   #栈元素: (3,6,5)      最小元素: 3
push(7);   #栈元素: (7,3,6,5)    最小元素: 3
pop();     #栈元素: (3,6,5)      最小元素: 3
pop();     #栈元素: (6,5)        最小元素: 5
```

解：由于可能有连续的进栈和出栈操作，并且栈中的元素可能重复，所以仅保存栈中的一个最小元素会得不到正确的结果，为此设计满足题目要求的顺序栈类为 STACK，它包含 data 和 mindata 两个数组，data 数组表示 data 栈(主栈)，mindata 数组表示 mindata 栈，后者作为存放当前最小元素的辅助栈。

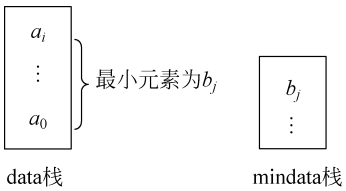


图 3.5 data 栈和 mindata 栈

当元素 $a_0, a_1, \cdots, a_i$ ($i \geqslant 1$) 进栈到 data 栈后，mindata 栈的栈顶元素 b_j 为 $a_0, a_1, \cdots, a_i$ 中的最小元素(含后进栈的重复最小元素)，如图 3.5 所示。

例如，前面的栈操作中 data 和 mindata 栈的变化如图 3.6 所示。STACK 类的主要运算算法设计如下：

① Getmin()函数用于返回栈中的最小元素，其操作

是取 mindata 栈的栈顶元素。

② 进栈函数 push(x)的操作是，当 data 栈空或者进栈元素 x 小于/等于当前栈中的最小元素(即 $x \leqslant$Getmin())时，则将 x 进 mindata 栈。最后将 x 进 data 栈。

③ 出栈函数 pop()的操作是，当 data 栈不空时从 data 栈出栈元素 x，若 mindata 栈的栈顶元素等于 x，则同时从 mindata 栈出栈 x。最后返回 x。

④ 取栈顶函数 gettop()的操作是，当 data 栈不空时返回 data 栈的栈顶元素。

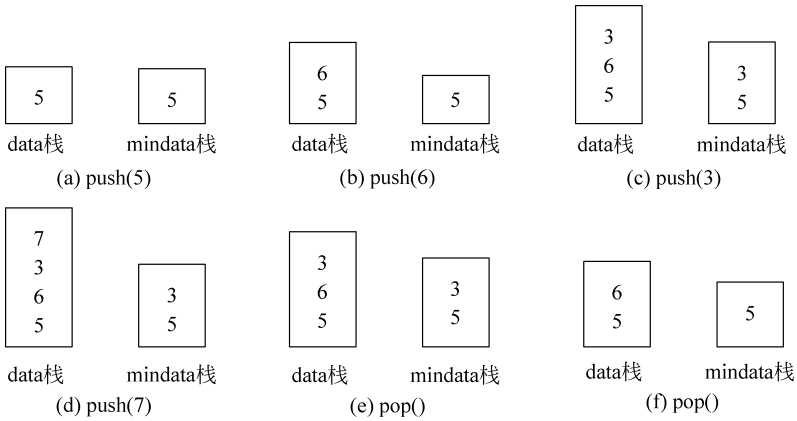


图 3.6　栈操作中 data 栈和 mindata 栈的变化情况

对应的完整程序如下：

```
#include <iostream>
using namespace std;
const int MaxSize=100;                    //栈中的最多元素个数
template <typename T>
class STACK                               //含 Getmin()的栈类
{
    T data[MaxSize];                      //存放主栈中的元素,初始为空
    T mindata[MaxSize];                   //存放 mindata 栈中的元素,初始为空
    int top;
    int mintop;
public:
    STACK():top(-1),mintop(-1) {}         //构造函数
private:                                  //mindata 栈简化的基本运算算法,设为私有的
    bool minempty()                       //判断 mindata 栈是否为空
    {
        return mintop==-1;
    }
    void minpush(T e)                     //元素 e 进 mindata 栈
    {
        mintop++;
        mindata[mintop]=e;
    }
    T minpop()                            //元素出 mindata 栈
    {
        T x=mindata[mintop];
```

```
        mintop--;
        return x;
    }
    T mingettop()                          //取 mindata 栈的栈顶元素
    {
        return mindata[mintop];
    }
public:                                    //主栈的基本运算算法,设为公有的
    bool empty()                           //判断主栈是否为空
    {
        return top==-1;
    }
    bool push(T x)                         //元素 x 进主栈
    {
        if (top==MaxSize-1)  return false; //主栈满返回 false
        if (empty() || x<=Getmin())
            minpush(x);                    //栈空或者 x<=mindata 栈顶元素时进 mindata 栈
        top++;
        data[top]=x;                       //将 x 进主栈
        return true;
    }
    bool pop(T& x)                         //元素 x 出主栈
    {
        if (empty())  return false;        //栈为空的情况,即栈下溢出
        x=data[top];                       //从主栈出栈 x
        top--;
        if (x==mingettop())                //若栈顶元素为最小元素
            minpop();                      //mindata 栈出栈一次
        return true;
    }
    bool gettop(T& e)                      //取主栈的栈顶元素
    {
        if (empty()) return false;         //栈为空的情况,即栈下溢出
        e=data[top];                       //取栈顶指针位置的元素
        return true;
    }
    T Getmin()                             //获取栈中的最小元素
    {
        return mingettop();                //返回 mindata 栈的栈顶元素,即主栈中的最小元素
    }
};
int main()
{
    STACK<int> st;                         //定义栈对象
    int e;
    cout <<"元素 5,6,3,7 依次进栈" << endl;
    st.push(5);
    st.push(6);
    st.push(3);
    st.push(7);
    cout << "  求最小元素并出栈" << endl;
```

```
    while (!st.empty())
    {
        cout << "      最小元素: " << st.Getmin() << endl;
        st.pop(e);
        cout << "      出栈元素: " << e << endl;
    }
    return 0;
}
```

上述程序的执行结果如下：

```
元素 5,6,3,7 依次进栈
  求最小元素并出栈
    最小元素:3
    出栈元素:7
    最小元素:3
    出栈元素:3
    最小元素:5
    出栈元素:6
    最小元素:5
    出栈元素:5
```

【例 3.7】 设有两个栈 $S1$ 和 $S2$，它们都采用顺序栈存储，并且共享一个固定容量的存储区 $s[0..M-1]$，为了尽量利用空间，减少溢出的可能，请设计这两个栈的存储方式。

解：为了尽量利用空间，减少溢出的可能，可以让两个栈的栈顶相向，即采用进栈元素迎面增长的存储方式，为此设置两个栈的栈顶指针分别为 top1 和 top2（均指向对应栈的栈顶元素），如图 3.7 所示。

图 3.7　两个顺序栈的存储结构

栈 S1 空的条件是 top1＝－1；栈 $S1$ 满的条件是 top1＝top2－1；元素 e 进栈 $S1$（栈不满时）的操作是“top1＋＋；s[top1]＝e”；元素 e 出栈 S1（栈不空时）的操作是“e＝s[top1]；top1－－”。

栈 S2 空的条件是 top2＝M；栈 $S2$ 满的条件是 top2＝top1＋1；元素 e 进栈 $S2$（栈不满时）的操作是“top2－－；s[top2]＝e”；元素 e 出栈 $S2$（栈不空时）的操作是“e＝s[top2]；top2＋＋”。

说明：本例的共享栈主要适合将固定容量的空间用作两个栈，不适合 3 个或者更多栈共享，因为超过两个栈共享时栈的运算性能较低。

视频讲解

3.1.4　栈的链式存储结构及其基本运算算法的实现

视频讲解

采用链式存储的栈称为**链栈**，这里采用单链表实现。链栈的优点是不需要考虑栈满上溢出的情况。这里用带头结点的单链表 head 表示的链栈如图 3.8 所示，首结点是栈顶结

点,尾结点是栈底结点,栈中的元素自栈底到栈顶依次是 a_0、a_1,…,a_{n-1}。

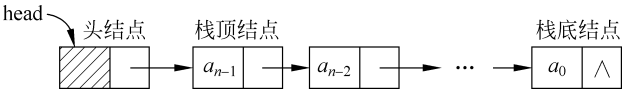


图 3.8 链栈的存储结构

从该链栈存储结构看到,初始时只含有一个头结点 head 并置 head->next 为 NULL,这样链栈的四要素如下。

① 栈空条件:head->next==NULL。

② 栈满条件:由于只有在内存溢出时才会出现栈满,因此通常不考虑这种情况。

③ 元素 e 进栈操作:将包含该元素的结点 s 插入作为首结点。

④ 出栈操作:返回首结点值并且删除该结点。

和普通单链表一样,链栈中每个结点的类型 LinkNode 定义如下:

```
template<typename T>
struct LinkNode                          //链栈结点类型
{
    T data;                              //数据域
    LinkNode* next;                      //指针域
    LinkNode():next(NULL){}              //构造函数
    LinkNode(T d):data(d),next(NULL){}   //重载构造函数
};
```

链栈类模板 LinkStack 的设计如下:

```
template<typename T>
class LinkStack                          //链栈类模板
{
public:
    LinkNode<T>* head;                   //链栈的头结点
    #栈的基本运算算法
};
```

在链栈中实现栈的基本运算的算法如下。

1) 链栈的初始化和销毁

链栈类的构造函数和析构函数与 2.3.2 节中单链表的完全相同:

```
LinkStack()                              //构造函数
{
    head=new LinkNode<T>();
}
~LinkStack()                             //析构函数
{
    LinkNode<T>* pre=head, *p=pre->next;
    while (p!=NULL)
    {
        delete pre;
        pre=p; p=p->next;                //pre、p同步后移
    }
```

```
    delete pre;
}
```

2）判断栈是否为空：empty()

若头结点的 next 域为空表示空栈，即单链表中没有任何数据结点。对应的算法如下：

```
bool empty()                                    //判栈空算法
{
    return head->next==NULL;
}
```

3）进栈：push(T e)

新建包含数据元素 e 的结点 p，将 p 结点插入头结点的后面。链栈不考虑栈满的情况，所以链栈的进栈运算总是返回 true。对应的算法如下：

```
bool push(T e)                                  //进栈算法
{
    LinkNode<T>* p=new LinkNode<T>(e);          //新建结点 p
    p->next=head->next;                         //插入结点 p 作为首结点
    head->next=p;
    return true;
}
```

4）出栈：pop(T& e)

在链栈空时返回 false，否则让 p 指向首结点，取结点 p 的值并删除它，同时返回 true。对应的算法如下：

```
bool pop(T& e)                                  //出栈算法
{
    LinkNode<T>* p;
    if (head->next==NULL)   return false;       //栈空的情况
    p=head->next;                               //p 指向开始结点
    e=p->data;
    head->next=p->next;                         //删除结点 p
    delete p;                                   //释放结点 p
    return true;
}
```

5）取栈顶元素：gettop(T& e)

在链栈空时返回 false，否则取首结点的值并返回 true。对应的算法如下：

```
bool gettop(T& e)                               //取栈顶元素
{
    LinkNode<T>* p;
    if (head->next==NULL)   return false;       //栈空的情况
    p=head->next;                               //p 指向开始结点
    e=p->data;
    return true;
}
```

3.1.5 链栈的应用算法设计示例

【例 3.8】 设计一个算法,利用栈的基本运算将一个整数链栈中的所有元素逆置。例如链栈 st 中的元素从栈底到栈顶为(1,2,3,4),逆置后为(4,3,2,1)。


视频讲解

解:这里要求利用栈的基本运算来设计算法,所以不能直接采用单链表逆置方法。先出栈 st 中的所有元素并保存在一个数组 a 中,再将 a 中的所有元素依次进栈。对应的算法如下:

```
#include "LinkStack.cpp"                    //包含链栈类模板的定义
void Reverse(LinkStack<int> & st)           //逆置栈 st
{
    int a[MaxSize];                         //定义一个辅助数组
    int i=0,e;
    while (!st.empty())                     //将出栈的元素放到数组 a 中
    {
        st.pop(e);
        a[i++]=e;
    }
    for (int j=0;j<i;j++)                   //将数组 a 中的所有元素进栈
        st.push(a[j]);
}
```

【例 3.9】 定义一个栈数据结构 STACK,添加一个 Getbottom()运算用于直接返回栈底元素(假设栈不空),要求采用链表实现,并且函数 Getbottom()、push()以及 pop()的时间复杂度都是 $O(1)$。


视频讲解

解:如果采用普通单链表实现,以前端为栈顶,以后端为栈底,那么找到尾结点(存放栈底元素)的时间复杂度为 $O(n)$,不满足题目要求。改为不带头结点仅有尾结点指针的循环单链表 rear 作为链栈,如图 3.9 所示。初始时 rear=NULL,栈的四要素如下。

① 栈空条件:rear=NULL。

② 栈满条件:不考虑。

③ 元素 e 进栈操作:建立含 e 元素的结点 p,将结点 p 插入 rear 结点的后面。

④ 元素 e 出栈操作:取 rear 结点后面的结点值 e,删除该结点。

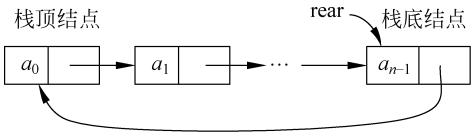


图 3.9 STACK 链栈的存储结构

这样,Getbottom()函数就返回了 rear 结点值,即 rear->data(栈不空时)。满足题目要求的 STACK 类如下:

```
template<typename T>
struct LinkNode                             //链栈结点类型
{
    T data;                                 //数据域
```

```
    LinkNode * next;                                    //指针域
    LinkNode():next(NULL) {}                            //构造函数
    LinkNode(T d):data(d),next(NULL) {}                 //重载构造函数
};
template < typename T >
class STACK                                             //链栈类模板
{
public:
    LinkNode < T > * rear;                              //链栈的尾结点指针
    STACK():rear(NULL) {}                               //构造函数
    ~STACK()                                            //析构函数
    {
        if (rear==NULL) return;                         //空链表直接返回
        LinkNode < T > * pre=rear, * p=pre-> next;
        while (p!=rear)
        {
            delete pre;
            pre=p; p=p-> next;                          //pre、p 同步后移
        }
        delete pre;
    }
    bool empty()                                        //判栈空算法
    {
        return rear==NULL;
    }
    bool push(T e)                                      //进栈算法
    {
        LinkNode < T > * p=new LinkNode < T >(e);       //新建结点 p
        if (empty())                                    //栈为空的情况
        {
            rear=p;
            rear-> next=rear;
        }
        else                                            //栈不空的情况
        {
            p-> next=rear-> next;                       //将结点 p 插入结点 rear 的后面
            rear-> next=p;
        }
        return true;
    }
    bool pop(T& e)                                      //出栈算法
    {
        LinkNode < T > * p;
        if (empty())  return false;                     //栈空的情况
        if (rear-> next==rear)                          //栈中只有一个结点的情况
        {
            p=rear;
            rear=NULL;
        }
        else                                            //栈中有两个及以上结点的情况
        {
            p=rear-> next;
            rear-> next=p-> next;
```

```
        }
        e=p->data;
        delete p;                                    //释放结点 p
        return true;
    }
    bool gettop(T& e)                                //取栈顶元素
    {
        if (empty())   return false;                 //栈空的情况
        e=rear->next->data;
        return true;
    }
    T Getbottom()                                    //取栈底元素
    {
        if (empty()) throw "栈空";
        return rear->data;
    }
};
```


视频讲解

3.1.6 STL 中的 stack 栈容器

STL 中的 stack 栈容器是前面介绍的栈数据结构的一种实现,具有后进先出的特点。stack 容器只有一个进出口,即栈顶,可以在栈顶插入(进栈)和删除(出栈)元素,而不允许像数组那样从前向后或者从后向前顺序遍历,所以 stack 容器没有 begin()/end()和 rbegin()/rend()这样的用于迭代器的成员函数。

stack 是一种适配器容器,即使用一个特定容器类的封装对象作为它的底层容器。简单地说,stack 的数据存放在底层容器中,并且利用底层容器提供的成员函数(如 back()、push_back()、pop_back())实现 stack 的功能。如果未特别指定 stack 的底层容器,默认使用双端队列 deque 容器作为底层容器,也可以指定 vector 或者 list 作为底层容器。

例如,以下语句用于定义 4 个 stack 对象:

```
stack<int> st1;                                  //定义一个整数栈 st1
stack<int> st2(st1);                             //由 st1 栈复制产生 st2 栈
stack<int, vector<int>> st3;                     //定义整数栈 st3,以 vector 作为底层容器
stack<int, list<int>> st4;                       //定义整数栈 st4,以 list 作为底层容器
```

stack 容器的主要成员函数如表 3.1 所示,与前面讨论的栈运算相比,stack 容器的成员函数的使用更加简单、方便。需要注意的是 stack 容器具有空间动态扩展功能,push()不会出现上溢出的情况,另外在使用 top()和 pop()之前应保证栈不空。

表 3.1 stack 容器的主要成员函数及其说明

成员函数	说明
empty()	判断栈是否为空
size()	返回栈中的实际元素个数
push(*e*)	元素 *e* 进栈
top()	返回栈顶元素,当栈空时抛出异常
pop()	出栈一个元素,并不返回出栈的元素,当栈空时抛出异常

例如有以下程序：

```
#include <iostream>
#include <stack>
using namespace std;
int main()
{
    stack<int> st;
    st.push(1); st.push(2); st.push(3);
    printf("栈顶元素：%d\n",st.top());
    printf("出栈顺序：");
    while (!st.empty())                          //栈不空时出栈所有元素
    {
        printf("%d ",st.top());
        st.pop() ;
    }
    printf("\n");
    return 0;
}
```

在上述程序中建立了一个整数栈 st，进栈 3 个元素，取栈顶元素，然后出栈所有元素并输出。程序的执行结果如下：

```
栈顶元素：3
出栈顺序：3 2 1
```

说明：由于 vector 向量容器提供了高效的尾端插入(push_back)和删除(pop_back)运算，并且可以顺序遍历元素，所以有时直接用 vector 代替 stack。

【实战 3.1】　POJ1363——铁轨问题

时间限制：1000ms；内存限制：65 536KB。

问题描述：A 市有一个著名的火车站，那里的山非常多。该站建于 20 世纪，不乐观的是该火车站是一个死胡同，并且只有一条铁轨，如图 3.10 所示。

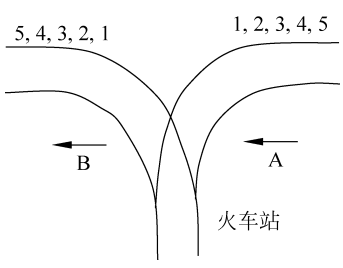


图 3.10　铁轨问题示意图


视频讲解

当地的传统是从 A 方向到达的每列火车都继续沿 B 方向行驶，需要以某种方式进行车厢重组。假设从 A 方向到达的火车有 n($n \leqslant 1000$)个车厢，按照递增的顺序 1，2，…，n 编号。火车站负责人必须知道是否可以通过重组得到 B 方向的车厢序列。

输入格式：输入由若干块组成。除了最后一个块外，每个块描述了一个列车以及可能更多的车厢重组要求。在每个块的第一行中为上述的整数 n，下一行是 1，2，…，n 的车厢重组序列。每个块的最后一行仅包含 0。最后一个块只包含一行 0。

输出格式：输出包含与输入中具有车厢重组序列对应的行。如果可以得到车厢对应的重组序列，输出一行“Yes”，否则输出一行“No”。此外，在输入的每个块后面有一个空行。


视频讲解

【实战 3.2】 POJ1208——箱子操作

问题描述参见第 2 章中的实战 2.4,这里改为用栈求解。

3.1.7 栈的综合应用

本节通过利用栈求简单算术表达式值和求解迷宫问题两个示例来说明栈的应用。


视频讲解

1. 用栈求简单算术表达式值

❑ 问题描述

这里限定的简单算术表达式(简称为表达式)求值问题是,用户输入一个仅包含+、-、*、/、正整数和小括号的合法算术表达式,计算该表达式的运算结果。

❑ 数据组织

表达式采用字符串 exp 表示,其中仅含运算符(operator)和运算数(operand)。在设计的相关算法中用到两个栈,一个是运算符栈 opor,另一个是运算数栈 opand,均采用 stack 栈容器表示,其定义如下:

```
stack<char> opor;                    //运算符栈
stack<double> opand;                 //运算数栈
```

❑ 设计运算算法

运算符位于两个运算数中间的表达式称为**中缀表达式**,例如 exp="1+2*(4+12)"就是一个中缀表达式,中缀表达式是最常用的一种表达式。计算中缀表达式一般遵循"从左到右,先乘除,后加减,有括号时先括号内,后括号外"的规则,因此,中缀表达式求值不仅要依赖运算符的优先级,还要处理括号。

所谓**后缀表达式**,就是运算符放在运算数的后面。后缀表达式有这样的特点:已经考虑了运算符的优先级,不包含括号,只含运算数和运算符。这里后缀表达式采用 postexp 字符串存放,每个整数字符串以"#"结尾,例如前面 exp 对应的 postexp 为"1#2#4#12#+*+"。

后缀表达式的求值十分简单,其过程是从左到右遍历后缀表达式,若遇到一个运算数,就将它进运算数栈;若遇到一个运算符 op,就从运算数栈中连续出栈两个运算数,假设为 a 和 b,计算 b op a 之值,并将计算结果进运算数栈;对整个后缀表达式遍历结束后,栈顶元素就是计算结果。

假设给定的简单表达式 exp 是正确的,其求值过程分为两步,先将中缀表达式 exp 转换成后缀表达式 postexp,然后对后缀表达式求值。设计求表达式值的类 Express 如下:

```
class Express                        //求表达式值的类
{
    string exp;                      //存放中缀表达式
    string postexp;                  //存放后缀表达式
public:
    Express(string str)              //构造函数
    {
        exp=str;
        postexp="";
    }
    string getpostexp()              //返回 postexp
    {
```

```
        return postexp;
    }
    void Trans() { … }                    //将算术表达式 exp 转换成后缀表达式 postexp
    double GetValue() { … }               //计算后缀表达式 postexp 的值
};
```

1）中缀表达式转换成后缀表达式

将正确的中缀表达式 exp 转换成后缀表达式 postexp 时仅用到运算符栈 opor，其转换过程是遍历 exp，遇到数字符，将连续的数字符末尾加上'#'后添加到 postexp；遇到'('，将其进栈；遇到')'，退栈运算符并添加到 postexp，直到退栈的是'('为止（该左括号不添加到 postexp 中）；遇到运算符 op_2，将其跟栈顶运算符 op_1 的优先级进行比较，只有当 op_2 的优先级高于 op_1 的优先级时才直接将 op_2 进栈，否则将栈中'('（如果有）之前的优先级等于或高于 op_2 的运算符均退栈并添加到 postexp，如图 3.11 所示，再将 op_2 进栈。

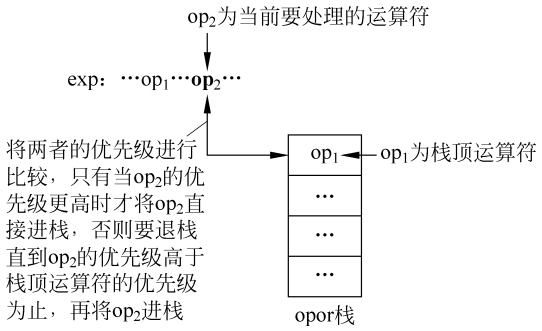


图 3.11　当前运算符的操作

上述过程说明如下：在遍历 exp 的任何运算符 op_2 时，除非遍历结束，都不能确定是否立即执行 op_2，所以将其暂时保存在 opor 栈中。假设 exp 中只有 op_1 和 op_2 运算符，op_2 的处理过程如下：

① 当 op_2 和 op_1 的优先级相同时，op_1 先进栈，说明 exp 中 op_1 在 op_2 的前面，按中缀表达式的运算规则，先做 op_1，即出栈 op_1 并添加到 postexp 中（按后缀表达式的求值过程，先添加的先执行），再将 op_2 进栈。

② 当 op_2 低于 op_1 的优先级时，显然先做 op_1，也就是出栈 op_1 并添加到 postexp 中，再将 op_2 进栈。

③ 当 op_2 高于 op_1 的优先级时，按中缀表达式的运算规则，op_2 应该在 op_1 之前做，此时直接将 op_2 进栈，以后 op_2 一定先于 op_1 出栈，从而满足该运算规则。

④ 当 op_2 为'('时，表示开始处理“(…)”，此时要么遇到 exp 开头的'('，要么遇到一个子表达式，所以无论栈中有什么运算符，都直接将'('进栈。

⑤ 当 op_2 为')'时，表示一个表达式或者子表达式处理结束，由于假设表达式中的括号是匹配的，所以栈中一定存在'('。设栈顶到栈底方向的第一个'('的位置为 p，如图 3.12 所示，将栈顶到 p 位置前的所有运算符出栈并添加到 postexp 中，再出栈'('，该'('不需要添加到 postexp。

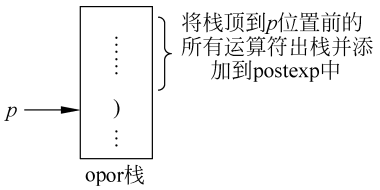


图 3.12　遇到')'的处理方式

由于中缀和后缀表达式中所有运算数的相对次序相同,所以遇到每个运算数都直接添加到 postexp。

针对这里的简单算术表达式,只有'*'和'/'运算符的优先级高于'+'和'-'运算符的优先级,所以上述过程简化如下:

```
while (若 exp 未读完)
{
    从 exp 读取字符 ch
    ch 为数字符: 将连续的数字符末尾加上'#'后添加到 postexp 中
    ch 为左括号'(': 将'('进栈
    ch 为右括号')': 将栈中'('后进栈的运算符依次出栈并添加到 postexp 中,再将'('退栈
    ch 为'+'或'-':将 opor 栈中'('后进栈的(如果有'(')所有运算符出栈并添加到 postexp,再将 ch 进栈
    ch 为'*'或'/':将 opor 栈中'('后进栈的(如果有'(')所有'*'或'/'运算符出栈并添加到 postexp,
                  再将 ch 进栈
}
若字符串 exp 扫描完毕,则退栈所有运算符并添加到 postexp
```

例如,对于 exp="(56-20)/(4+2)",其转换成后缀表达式的过程如表 3.2 所示。

表 3.2 表达式"(56-20)/(4+2)"转换成后缀表达式的过程

ch	操　作	postexp	opor 栈
(	将'('进栈		(
56	将 56 存入 postexp 中	56#	(
-	由于栈顶为'(',直接将'-'进栈	56#	(-
20	将 20 添加到 postexp	56#20#	(-
)	将栈中'('后进栈的运算符出栈并添加到 postexp,再将'('出栈	56#20#-	
/	将'/'进栈	56#20#-	/
(	将'('进栈	56#20#-	/(
4	将 4 添加到 postexp	56#20#-4#	/(
+	由于栈顶为'(',直接将'+'进栈	56#20#-4#	/(+
2	将 2 添加到 postexp	56#20#-4#2#	/(+
)	将栈中'('后进栈的运算符出栈并添加到 postexp,再将'('出栈	56#20#-4#2#+	/
	exp 扫描完毕,将栈中的所有运算符依次出栈并添加到 postexp,得到最后的后缀表达式	56#20#-4#2#+/	

根据上述原理得到中缀表达式 exp 转换为后缀表达式 postexp 的算法如下:

```
void Trans()                          //将中缀表达式 exp 转换成后缀表达式 postexp
{
    stack<char> opor;                 //运算符栈
    int i=0;                          //i 为 exp 的下标
    char ch,e;
    while (i<exp.length())            //exp 表达式未扫描完时循环
    {
        ch=exp[i];
        if (ch=='(')                  //遇到左括号
            opor.push(ch);            //将左括号直接进栈
        else if (ch==')')             //遇到右括号
```

```
        {
            while (!opor.empty() && opor.top()!='(')
            {
                e=opor.top();                    //将栈中'('前面的运算符退栈并存入 postexp
                opor.pop();
                postexp+=e;
            }
            opor.pop();                          //将'('退栈
        }
        else if (ch=='+' || ch=='-')             //遇到加或减号
        {
            while (!opor.empty() && opor.top()!='(')
            {
                e=opor.top();                    //将栈中'('前面的所有运算符退栈并存入 postexp
                opor.pop();
                postexp+=e;
            }
            opor.push(ch);                       //再将'+'或'-'进栈
        }
        else if (ch=='*' || ch=='/')             //遇到'*'或'/'号
        {
            while (!opor.empty() && opor.top()!='(' && (opor.top()=='*' || opor.top()=='/'))
            {
                e=opor.top();                    //将栈中'('前面的所有'*'或'/'依次出栈并存入 postexp
                opor.pop();
                postexp+=e;
            }
            opor.push(ch);                       //再将'*'或'/'进栈
        }
        else                                     //遇到数字字符
        {
            string d="";
            while (ch>='0' && ch<='9')           //遇到数字
            {
                d+=ch;                           //提取所有连续的数字字符
                i++;
                if (i<exp.length())              //exp 没有遍历完时取下一个字符 ch
                    ch=exp[i];
                else                             //exp 遍历完毕时退出数字判断
                    break;
            }
            i--;                                 //退一个字符
            postexp+=d;                          //将数字串存入 postexp
            postexp+="#";                        //用"#"标识一个数字串结束
        }
        i++;                                     //继续处理其他字符
    }
    while (!opor.empty())                        //此时 exp 扫描完毕,栈不空时循环
    {
        e=opor.top();
        opor.pop();                              //将栈中的所有运算符退栈并放入 postexp
```

```
            postexp+=e;
        }
    }
```

2) 后缀表达式的求值

在后缀表达式求值中仅用到运算数栈 opand。对后缀表达式 postexp 求值的过程如下：

```
while (若 postexp 未读完)
{
    从 postexp 读取一个元素 ch
    ch 为'+': 出栈两个数值 a 和 b,计算 c=b+a,再将 c 进栈
    ch 为'-': 出栈两个数值 a 和 b,计算 c=b-a,再将 c 进栈
    ch 为'*': 出栈两个数值 a 和 b,计算 c=b*a,再将 c 进栈
    ch 为'/': 出栈两个数值 a 和 b,若 a 不为零,计算 c=b/a,再将 c 进栈
    ch 为数值: 将该数值进栈
}
opand 栈中唯一的数值即为表达式值
```

例如，postexp="56#20#-4#2#+/"的求值过程如表 3.3 所示。

表 3.3 后缀表达式"56#20#-4#2#+/"的求值过程

ch 序列	说　明	opand 栈
56#	遇到 56#，转换为整数 56 并进栈	56
20#	遇到 20#，转换为整数 20 并进栈	56,20
-	遇到-，出栈两次，将 56-20=36 进栈	36
4#	遇到 4#，转换为整数 4 并进栈	36,4
2#	遇到 2#，转换为整数 2 并进栈	36,4,2
+	遇到+，出栈两次，将 4+2=6 进栈	36,6
/	遇到/，出栈两次，将 36/6=6 进栈	6
	postexp 遍历完毕，算法结束，栈顶数值 6 即为所求	

根据上述计算原理得到计算后缀表达式值的算法如下：

```
double GetValue()                          //计算后缀表达式 postexp 的值
{
    stack<double> opand;                   //定义运算数栈 opand
    double a,b,c,d;
    char ch;
    int i=0;
    while (i<postexp.length())             //postexp 字符串未扫描完时循环
    {
        ch=postexp[i];
        switch (ch)
        {
        case '+':                          //遇到+
            a=opand.top(); opand.pop();    //退栈运算数 a
            b=opand.top(); opand.pop();    //退栈运算数 b
            c=b+a;                         //计算 c
            opand.push(c);                 //将计算结果进栈
            break;
```

```
        case '-':                          //遇到-
            a=opand.top(); opand.pop();    //退栈运算数 a
            b=opand.top(); opand.pop();    //退栈运算数 b
            c=b-a;                         //计算 c
            opand.push(c);                 //将计算结果进栈
            break;
        case '*':                          //遇到 *
            a=opand.top(); opand.pop();    //退栈运算数 a
            b=opand.top(); opand.pop();    //退栈运算数 b
            c=b*a;                         //计算 c
            opand.push(c);                 //将计算结果进栈
            break;
        case '/':                          //遇到/
            a=opand.top(); opand.pop();    //退栈运算数 a
            b=opand.top(); opand.pop();    //退栈运算数 b
            c=b/a;                         //计算 c
            opand.push(c);                 //将计算结果进栈
            break;
        default:                           //遇到数字字符
            d=0;                           //将连续的数字字符转换成数值存放到 d 中
            while (ch>='0' && ch<='9')
            {
                d=10*d+(ch-'0');
                i++;
                ch=postexp[i];
            }
            opand.push(d);                 //将数值 d 进栈
            break;
        }
        i++;                               //继续处理其他字符
    }
    return opand.top();                    //栈顶元素即为求值结果
}
```

❑ 设计主程序

设计以下主程序求简单算术表达式“(56-20)/(4+2)”的值：

```
int main()
{
    string str="(56-20)/(4+2)";
    Express obj(str);
    cout << "中缀表达式: " << str << endl;
    cout << "中缀转换为后缀" << endl;
    obj.Trans();
    cout << "后缀表达式: " << obj.getpostexp() << endl;
    cout << "求后缀表达式值" << endl;
    cout << "求值结果:    " << obj.GetValue() << endl;
    return 0;
}
```

❑ 程序执行结果

本程序的执行结果如下：

```
中缀表达式：  (56－20)/(4＋2)
中缀转换为后缀
后缀表达式：  56＃20＃－4＃ 2＃＋/
求后缀表达式值
求值结果：    6
```

上述先转换为后缀表达式再对后缀表达式求值的两步可以合并起来，同样需要设置运算符栈 opor 和运算数栈 opand，合并过程中遍历表达式 exp：

① 遇到数字字符，将后续的所有数字字符合起来转换为数值，进栈到 opand。

② 遇到左括号，进栈到 opor。

③ 遇到右括号，将 opor 栈中'('后进栈的运算符依次出栈并添加到 postexp 中，再将'('退栈。

④ 遇到运算符，只有优先级高于 opor 栈顶运算符的才直接进栈 opor，否则出栈 op 并执行 op 计算。

若简单表达式遍历完毕，退栈 opor 的所有运算符并执行 op 计算。

其中，执行 op 计算的过程是：出栈 opand 两次得到运算数 a 和 b，执行 $c=b$ op a，然后 c 进栈 opand。最后 opand 栈的栈顶运算数就是简单表达式的值。


视频讲解

2. 用栈求解迷宫问题

❑ 问题描述

给定一个 $m\times n$ 的迷宫图，求一条从指定入口到出口的路径。假设迷宫图如图 3.13 所示(其中 $m=4$，$n=4$)，迷宫由方块构成，空白方块表示可以走的通道，阴影方块表示不可走的障碍物。要求所求路径必须是简单路径，即在求得的路径上不能重复出现同一空白方块，而且从每个方块出发只能走向上、下、左、右 4 个相邻的空白方块。

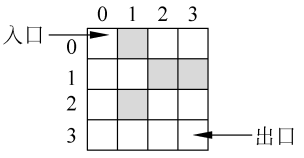


图 3.13　一个迷宫示意图

❑ 迷宫的数据组织

为了表示迷宫，设置一个数组 mg，其中每个元素表示一个方块的状态，为 0 时表示对应方块是通道，为 1 时表示对应方块是不可走的障碍物。图 3.13 所示的迷宫对应的迷宫数组 mg 如下：

```
int mg[MAX][MAX]={{0,1,0,0},{0,0,1,1},{0,1,0,0},{0,0,0,0}};
int m=4,n=4;                                    //一个 4 行 4 列的迷宫图
```

❑ 设计运算算法

求迷宫问题就是在一个指定的迷宫中求出从入口到出口的一条路径。在求解时，通常用的方法是穷举法，即从入口出发，沿着某个方位向前试探，若能走通，则继续往前走；否则进入死胡同，沿原路退回，换一个方位再继续试探，直至所有可能的通路都试探完为止。

对于迷宫中的每个方块，有上、下、左、右 4 个方块相邻，如图 3.14 所示，第 i 行第 j 列的方块的位置记为(i,j)，规定上方方块为方位 0，并按顺时针方向递增编号。对应的方位偏移量如下：

```
int dx[]={-1,0,1,0};                    //x 方向的偏移量
int dy[]={0,1,0,-1};                    //y 方向的偏移量
```

为了保证在任何位置上都能沿原路退回(称为回溯)，需要用一个后进先出的栈保存从入口到当前方块的路径，也就是说每个可走的方块都要进栈，栈中保存的每个方块除了位置信息外，还有走向信息，即从该方块走到相邻方块的方位 di。st 栈采用 stack 容器表示，每个方块的 Box 类型定义如下：

```
struct Box                              //方块类型
{
    int i;                              //方块的行号
    int j;                              //方块的列号
    int di;                             //di 是下一可走相邻方块的方位号
    Box() {}                            //构造函数
    Box(int i1,int j1,int d1):i(i1),j(j1),di(d1) {}     //重载构造函数
};
```

说明：栈是一种具有记忆功能的数据结构，在应用中重点是确定栈元素保存哪些信息。这里看一个日常生活中的例子，如图 3.15 所示。假设小明住在 A 地，想到 C 地去看望好朋友，但他不熟悉路线。他从 A 地出发，走到了 B 地，有两条道路，于是他习惯性地走了上方的道路，结果遇到一条小河，他过不去，只好回到 B 地。如果他不记下前面走过的路线，他会在这条路线上陷入死循环，永远见不到好朋友。小明是个聪明的孩子，他会记下前面走过的路线，于是在 B 地走另外一条(下方的)道路，结果很快找到了 C 地，高兴地见到了好朋友。在这个例子中，小明要记忆走过的每个地点以及所走方向，小明的记忆功能可以用栈来实现。所以在求解迷宫问题中，用栈保存每个走过的方块以及所走方位。

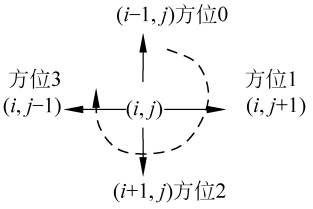


图 3.14　方位图

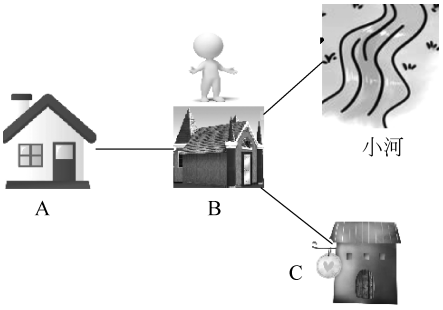


图 3.15　小明找好朋友的过程

求解入口(xi,yi)到出口(xe,ye)迷宫路径的过程是先将入口进栈(其初始方位设置为-1)，在栈不空时进行如下循环：

① 取栈顶方块 b(不退栈)。

② 若 b 方块是出口，则输出栈中的所有方块即为一条迷宫路径，返回 true。

③ 否则从 b 方块的新方位 di=b.di+1 开始试探相邻方块是否可走。

④ 若找到 b 方块的 di 方位的相邻方块(i,j)可走,则走到相邻方块(i,j),操作是修改栈顶 b 方块的 di 域为该 di 值,并将(i,j)方块(对应 b1)进栈(其初始方位设置为-1)。

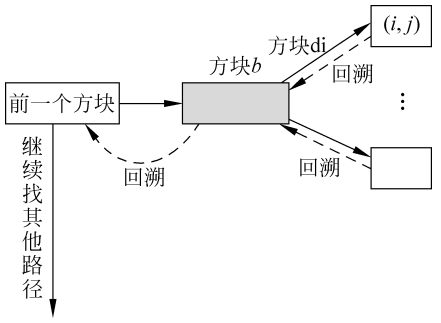


图 3.16 求迷宫路径的回溯过程

⑤ 若 b 方块找不到相邻可走方块,说明当前路径不可能走通(进入死胡同),b 方块不会是迷宫路径上的方块,则原路回退(即回溯),操作是将 b 方块出栈,从次栈顶方块(试探路径上 b 方块的前一个方块)做相同的试探,如图 3.16 所示(图中的虚线表示回退)。如果一直回退到出口,而出口也没有未试探过的相邻可走方块,说明不存在迷宫路径,返回 false。

为了保证试探的可走相邻方块不是已走路径上的方块,如(i,j)已进栈,在试探$(i+1,j)$的下一可走方块时又试探到(i,j),这样可能会引起死循环,为此在一个方块进栈后将对应的 mg 数组元素值改为-1(变为不可走的相邻方块),当退栈时(表示该栈顶方块没有可走相邻方块),将其恢复为 0。在图 3.13 所示的迷宫中,求入口(0,0)到出口(3,3)迷宫路径的搜索过程如图 3.17 所示,图中带"×"的方块是死胡同方块,走到这样的方块后需要回溯,找到出口后,栈中的方块对应迷宫路径。

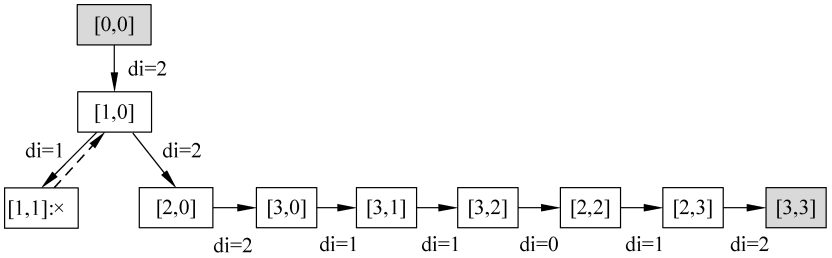


图 3.17 用栈求(0,0)到(3,3)迷宫路径的搜索过程

说明:这里的迷宫数组 mg 除了表示一个迷宫外,还通过将元素值设置为-1记忆路径,当找到出口后,恰好该迷宫路径上所有方块的 mg 元素值均为-1,这样可以在出口处继续回退查找所有的迷宫路径。如果在一个方块出栈时不将其 mg 元素值恢复为 0,那么尽管可以找到一条迷宫路径(当存在迷宫路径时),但会将所有试探方块的 mg 元素值均置为-1,这样不能找到其他可能存在的迷宫路径。

求解迷宫问题的 mgpath 算法如下:

```
bool mgpath(int xi,int yi,int xe,int ye)        //求一条从(xi,yi)到(xe,ye)的迷宫路径
{
    int i,j,di,i1,j1;
    bool find;
    Box b,b1;
    stack<Box> st;                              //建立一个栈
    b=Box(xi,yi,-1);
    st.push(b);                                 //入口方块进栈
    mg[xi][yi]=-1;                              //为避免来回找相邻方块,置 mg 值为-1
```

```
    while (!st.empty())                    //栈不空时循环
    {
        b=st.top();                        //取栈顶方块,称为当前方块
        if (b.i==xe && b.j==ye)            //找到出口,输出栈中的所有方块构成一条路径
        {
            disppath(st);
            return true;                   //找到一条路径后返回 true
        }
        find=false;                        //否则继续找路径
        di=b.di;
        while (di<3 && find==false)        //找 b 的一个相邻可走方块
        {
            di++;                          //找下一个方位的相邻方块
            i=b.i+dx[di];                  //找 b 的 di 方位的相邻方块(i,j)
            j=b.j+dy[di];
            if (i>=0 && i<m && j>=0 && j<n && mg[i][j]==0)
                find=true;                 //(i,j)方块有效且可走
        }
        if (find)                          //栈顶方块找到一个相邻可走方块(i,j)
        {
            st.top().di=di;                //修改栈顶方块的 di 为新值
            b1=Box(i,j,-1);                //建立相邻可走方块(i,j)的对象 b1
            st.push(b1);                   //b1 进栈
            mg[i][j]=-1;                   //为避免来回找相邻方块,置 mg 值为-1
        }
        else                               //栈顶方块没有找到任何相邻可走方块
        {
            mg[b.i][b.j]=0;                //恢复栈顶方块的迷宫值
            st.pop();                      //将栈顶方块退栈
        }
    }
    return false;                          //没有找到迷宫路径,返回 false
}
```

当成功找到出口后,栈 st 中从栈底到栈顶恰好是一条从入口到出口的迷宫路径,输出该迷宫路径并返回 true,否则说明找不到迷宫路径,返回 false。通过 st 栈输出一条迷宫路径的算法如下:

```
void disppath(stack<Box> & st)             //输出栈中的所有方块构成一条迷宫路径
{
    Box b;
    vector<Box> apath;                     //存放一条迷宫路径
    while (!st.empty())                    //出栈所有的方块
    {
        b=st.top(); st.pop();
        apath.push_back(b);
    }
    reverse(apath.begin(),apath.end());    //逆置 apath(也可以直接反向输出 apath)
    cout << "一条迷宫路径: ";
    for (int i=0;i<apath.size();i++)
        cout << "[" << apath[i].i << "," << apath[i].j << "]   ";
```

```
    cout << endl;
}
```

❑ 设计主程序

设计以下主程序求图 3.13 所示的迷宫图中从(0,0)到(3,3)的一条迷宫路径:

```
int main()
{
    int xi=0,yi=0,xe=3,ye=3;
    printf("求(%d,%d)到(%d,%d)的迷宫路径\n",xi,yi,xe,ye);
    if (!mgpath(xi,yi,xe,ye))
        cout << "不存在迷宫路径\n";
    return 0;
}
```

❑ 程序执行结果

本程序的执行结果如下:

```
求(0,0)到(3,3)的迷宫路径
一条迷宫路径:[0,0]  [1,0]  [2,0]  [3,0]  [3,1]  [3,2]
[2,2]  [2,3]  [3,3]
```

该路径如图 3.18 所示(迷宫路径方块上的箭头指示路径中下一个方块的方位),显然这个解不是最优解(即不是最短路径)。在后面使用队列求解时可以找出最短路径。

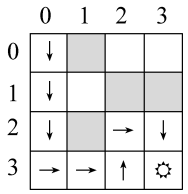


图 3.18 用栈找到的一条迷宫路径

3.2 队列

本节先介绍队列的定义,然后讨论队列存储结构和基本运算算法设计,最后通过迷宫问题的求解说明队列的应用。


视频讲解

3.2.1 队列的定义

队列(简称为队)是一种操作受限的线性表,其限制为仅允许在表的一端进行插入,而在表的另一端进行删除。把进行插入的一端称为**队尾**(rear),把进行删除的一端称为**队头**或**队首**(front)。向队列中插入新元素称为**进队**或**入队**,新元素进队后就成为新的队尾元素;从队列中删除元素称为**出队**或**离队**,元素出队后,其直接后继元素就成为队首元素。

由于队列的插入和删除操作分别是在表的各自一端进行的,每个元素必然按照进入的次序出队,所以又把队列称为**先进先出表**。

抽象数据类型队列的定义如下:

ADT Queue
{
数据对象:
$D=\{a_i \mid 0\leqslant i\leqslant n-1, n\geqslant 0\}$
数据关系:
$R=\{r\}$

$r=\{<a_i,a_{i+1}> \mid a_i,a_{i+1}\in D,i=0,\cdots,n-2\}$

基本运算：

empty()：判断队列是否为空，若队列为空返回真，否则返回假。

push(T e)：进队操作，将元素 e 进队作为队尾元素。

pop(T& e)：出队操作，从队头出队一个元素。

gethead(T& e)：取队头操作，取出队头的元素。

}

【例 3.10】　若元素进队的顺序为 1234，能否得到 3142 的出队序列？

解：进队的顺序为 1234，则出队的顺序只能是 1234(先进先出)，所以不能得到 3142 的出队序列。

3.2.2　队列的顺序存储结构及其基本运算算法的实现

由于队列中元素的逻辑关系与线性表中的相同，因此可以借鉴线性表的两种存储结构来存储队列。当队列采用顺序存储结构存储时，用 data 数组来存放队列中的元素，由于队列的两端都是动态变化的，为了表示队列的状态需要设置两个指针，队头指针为 front(实际上是队头元素的前一个位置)，队尾指针为 rear(正好是队尾元素的位置)。

为了简单，这里使用固定容量的数组 data(容量为常量 MaxSize)，如图 3.19 所示，队列中从队头到队尾为 $a_0,a_1,\cdots,a_{n-1}$。采用顺序存储结构的队列称为**顺序队**。

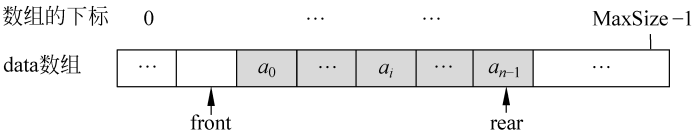


图 3.19　顺序队示意图

顺序队分为非循环队列和循环队列两种方式，这里先讨论非循环队列，并通过说明该类型队列的缺点引出循环队列。

1. 非循环队列

如图 3.20 所示为一个非循环队列的动态变化示意图(MaxSize=5)。图 3.20(a)表示一个空队；图 3.20(b)表示进队 5 个元素后的状态；图 3.20(c)表示出队一次后的状态；图 3.20(d)表示再出队 4 次后的状态。

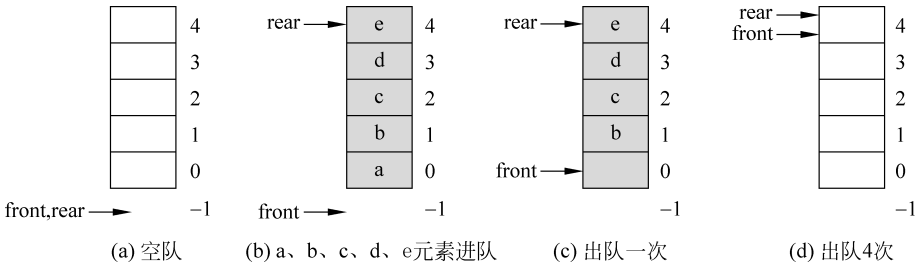


图 3.20　队列操作的示意图

从图 3.20 中看到，初始时置 front 和 rear 均为 -1(满足 front==rear 的条件)。非循环队列的四要素如下。

① 队空条件：front==rear。图3.20(a)和(d)满足该条件。

② 队满(队上溢出)条件：rear==MaxSize-1(因为每个元素进队都让rear增1，当rear达到最大下标时不能再增加)。图3.20(d)满足该条件。

③ 元素e进队操作。先将队尾指针rear增1，然后将元素e放在该位置(进队的元素总是在尾部插入的)。

④ 出队操作：先将队头指针front增1，然后取出该位置的元素(出队的元素总是从头部出来的)。

非循环队列类SqQueue的定义如下：

```
const int MaxSize=100;                    //队列的容量
template <typename T>
class SqQueue                             //非循环队列类模板
{
public:
    T * data;                             //存放队中的元素
    int front, rear;                      //队头和队尾指针
    //队列的基本运算算法
};
```

在非循环队列中实现队列的基本运算的算法如下：

1) 非循环队列的初始化和销毁

通过构造函数实现初始化，即创建一个空队；通过析构函数实现销毁，即释放队列占用的空间。对应的构造函数如下：

```
SqQueue()                                 //构造函数
{
    data=new T[MaxSize];                  //为data分配容量为MaxSize的空间
    front=rear=-1;                        //队头和队尾指针置初值
}
~SqQueue()                                //析构函数
{
    delete [] data;
}
```

2) 判断队列是否为空：empty()

若满足front==rear条件，则返回true，否则返回false。对应的算法如下：

```
bool empty()                              //判队空运算
{
    return (front==rear);
}
```

3) 进队运算：push(T e)

元素e进队只能从队尾插入，不能从队头或中间位置进队，仅改变队尾指针。进队操作是在队满时返回false，否则将队尾指针rear增1，然后将元素e放到该位置，并且返回true。对应的算法如下：

```
bool push(T e)                            //进队列运算
```

```
{
    if (rear==MaxSize-1)                    //队满上溢出
        return false;
    rear++;
    data[rear]=e;
    return true;
}
```

4）出队：pop(T &e)

元素出队只能从队头删除，不能从队头或中间位置出队，仅改变队头指针。出队操作是在队列空时返回 false，否则将队头指针 front 增 1，取出该位置的元素值，并返回 true。对应的算法如下：

```
bool pop(T& e)                              //出队列运算
{
    if (front==rear)                        //队空下溢出
        return false;
    front++;
    e=data[front];
    return true;
}
```

5）取队头元素：gethead(T &e)

与出队类似，但不需要移动队头指针 front。对应的算法如下：

```
bool gethead(T& e)                          //取队头运算
{
    if (front==rear)                        //队空下溢出
        return false;
    int head=front+1;
    e=data[head];
    return true;
}
```

上述算法的时间复杂度均为 $O(1)$。

2. 循环队列


视频讲解

在前面的非循环队列中，元素进队时队尾指针 rear 增 1，元素出队时队头指针 front 增 1，当进队 MaxSize 个元素后，队满条件 rear==MaxSize-1 成立，此时即使出队若干元素，队满条件仍成立（实际上队列中有空位置），这种队列中有空位置但仍然满足队满条件的上溢出称为**假溢出**。也就是说，非循环队列存在假溢出现象。为了克服非循环队列的假溢出，充分使用数组中的存储空间，可以把 data 数组的前端和后端连接起来，形成一个循环数组，即把存储队列元素的表从逻辑上看成一个环，称为**循环队列**（也称为**环形队列**）。

循环队列首尾相连，当队尾指针 rear=MaxSize-1 时，再前进一个位置就应该到达 0 位置，这可以利用数学上的求余运算（%）实现。

① 队首指针循环进 1：front=(front+1) % MaxSize。

② 队尾指针循环进 1：rear=(rear+1) % MaxSize。

循环队列的队头指针和队尾指针初始化为0,即front=rear=0。在进队元素和出队元素时,队头和队尾指针都循环前进一个位置。

那么,循环队列的队满和队空的判断条件是什么呢?若设置队空条件是rear==front,如果进队元素的速度快于出队元素的速度,队尾指针很快就赶上了队头指针,此时可以看出循环队列队满时也满足rear==front,所以这种设置无法区分队空和队满。

实际上循环队列的结构与非循环队列相同,也需要通过front和rear标识队列的状态,一般是采用它们的相对值(即|front-rear|)实现的,若data数组的容量为m,则队列的状态有$m+1$种,分别是队空、队中有一个元素、队中有两个元素、……、队中有m个元素(队满)。front和rear的取值范围均为0~$m-1$,这样|front-rear|只有m个值,显然$m+1$种状态不能直接用|front-rear|区分,因为必定有两种状态不能区分。为此让队列中最多只有$m-1$个元素,这样队列恰好只有m种状态,就可以通过front和rear的相对值区分所有状态了。

在规定队列中最多只有$m-1$个元素时,设置队空条件仍然是rear==front。当队列中有$m-1$个元素时一定满足(rear+1)%MaxSize==front。这样,循环队列在初始时置front=rear=0,其四要素如下。

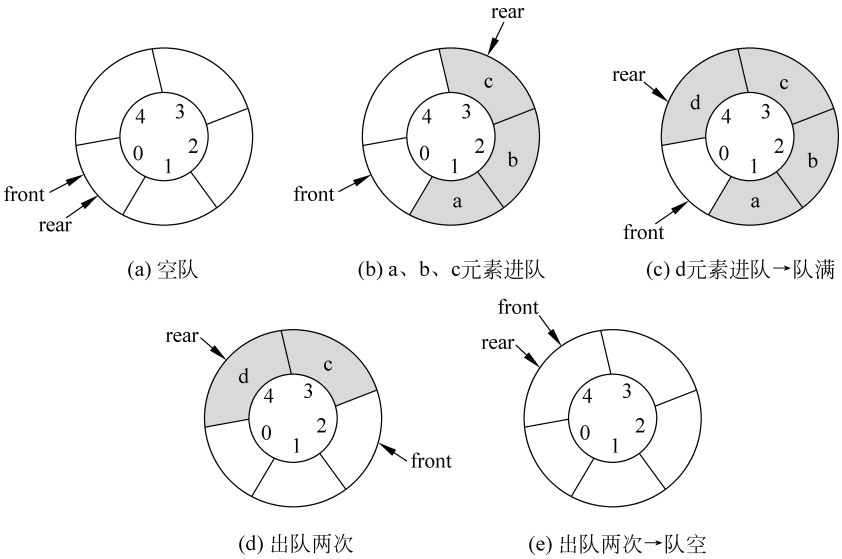
① 队空条件:rear==front。

② 队满条件:(rear+1)%MaxSize==front(相当于试探进队一次,若rear达到front,则认为队满了)。

③ 元素e进队操作:rear=(rear+1)%MaxSize,将元素e放置在该位置。

④ 元素e出队操作:front=(front+1)%MaxSize,取出该位置的元素e。

图3.21说明了循环队列的几种状态,这里假设MaxSize=5。图3.21(a)为空队,此时front=rear=0;图3.21(b)的队列中有3个元素,当进队元素d后队中有4个元素,此时满足队满的条件。



(a) 空队　(b) a、b、c元素进队　(c) d元素进队→队满

(d) 出队两次　(e) 出队两次→队空

图3.21　循环队列进队和出队操作示意图

循环队列类模板 CSqQueue 的定义如下：

```
const int MaxSize=100;                    //队列的容量
template < typename T >
class CSqQueue                            //循环队列类模板
{
public:
    T * data;                             //存放队中元素
    int front, rear;                      //队头和队尾指针
    //队列的基本运算算法
};
```

在这样的循环队列中，实现队列的基本运算算法如下。

1）循环队列的初始化和销毁

通过构造函数实现初始化，即创建一个空队；通过析构函数实现销毁，即释放队列占用的空间。对应的构造函数如下：

```
CSqQueue()                                //构造函数
{
    data=new T[MaxSize];                  //为 data 分配容量为 MaxSize 的空间
    front=rear=0;                         //队头和队尾指针置初值
}

~CSqQueue()                               //析构函数
{
    delete [] data;
}
```

2）判断队列是否为空：empty()

若满足 front==rear 条件，返回 true，否则返回 false。对应的算法如下：

```
bool empty()                              //判队空运算
{
    return (front==rear);
}
```

3）进队：push(T e)

在队列满时返回 false，否则先将队尾指针 rear 循环增 1，然后将元素 e 放到该位置，并返回 true。对应的算法如下：

```
bool push(T e)                            //进队列运算
{
    if ((rear+1)%MaxSize==front)          //队满上溢出
        return false;
    rear=(rear+1)%MaxSize;
    data[rear]=e;
    return true;
}
```

4）出队：pop(T &e)

在队列空时返回 false，否则将队头指针 front 循环增 1，取出该位置的元素值，并返回

true。对应的算法如下：

```
bool pop(T& e)                          //出队列运算
{
    if (front==rear) return false;      //队空下溢出
    front=(front+1)%MaxSize;
    e=data[front];
    return true;
}
```

5) 取队头元素：gethead(T &e)

与出队类似，但不需要移动队头指针 front。对应的算法如下：

```
bool gethead(T& e)                      //取队头运算
{
    if (front==rear) return false;      //队空下溢出
    int head=(front+1)%MaxSize;
    e=data[head];
    return true;
}
```

上述算法的时间复杂度均为 $O(1)$。

3.2.3 循环队列的应用算法设计示例

【例 3.11】 在 CSqQueue 循环队列类中增加一个求元素个数的算法 getlength()。对于一个整数循环队列 qu，利用队列基本运算和 getlength()算法设计进队和出队第 k($k\geqslant 1$，队头元素的序号为 1)个元素的算法。


视频讲解

解：在前面的循环队列中，队头指针 front 指向队中队头元素的前一个位置，队尾指针 rear 指向队中的队尾元素，可以求出队中元素的个数=(rear−front+MaxSize)%MaxSize。为此在 CSqQueue 循环队列类中增加 getlength()算法如下：

```
int getlength()                         //返回队中元素的个数
{
    return (rear-front+MaxSize)%MaxSize;
}
```

队列中并没有直接取 k($k\geqslant 1$)个元素的基本运算，进队第 k 个元素 e 的算法思路是出队前 $k-1$ 个元素，边出边进，再将元素 e 进队，将剩下的元素边出边进。该算法如下：

```
bool pushk(CSqQueue<int> & qu,int k,int e)   //进队第 k 个元素 e
{
    int x;
    int n=qu.getlength();
    if (k<1 || k>n+1) return false;         //参数 k 错误返回 false
    if (k<=n)
    {
        for (int i=1;i<=n;i++)              //循环处理队中的所有元素
        {
            if (i==k)
```

```
            qu.push(e);                    //将 e 元素进队到第 k 个位置
            qu.pop(x);                     //出队元素 x
            qu.push(x);                    //进队元素 x
        }
    }
    else  qu.push(e);                      //k=n+1 时直接将 e 进队
    return true;
}
```

出队第 $k(k\geqslant 1)$个元素 e 的算法思路是出队前 $k-1$ 个元素，边出边进，再出队第 k 个元素 e，e 不进队，最后将剩下的元素边出边进。该算法如下：

```
bool popk(CSqQueue<int> & qu,int k,int& e)   //出队第 k 个元素
{
    int x;
    int n=qu.getlength();
    if (k<=1 || k>n) return false;         //参数 k 错误返回 false
    for (int i=1;i<=n;i++)                 //循环处理队中的所有元素
    {
        qu.pop(x);                         //出队元素 x
        if (i!=k)
            qu.push(x);                    //将非第 k 个元素进队
        else
            e=x;                           //取第 k 个出队的元素
    }
    return true;
}
```

【例 3.12】 对于循环队列来说，如果知道队头指针和队列中元素的个数，则可以计算出队尾指针。也就是说，可以用队列中元素的个数代替队尾指针。设计出这种循环队列的判队空、进队、出队和取队头元素的算法。

解：本例的循环队列包含 data 数组、队头指针 front 和队中元素个数 count 域，可以由 front 和 count 求出队尾位置。公式如下：


视频讲解

```
rear1=(front+count)%MaxSize
```

初始时 front 和 count 均置为 0。队空的条件为 count==0；队满的条件为 count==MaxSize；元素 e 进队的操作是先根据上述公式求出队尾指针 rear1，将 rear1 循环增 1，然后将元素 e 放置在 rear1 处；出队的操作是先将队头指针循环增 1，然后取出该位置的元素，进队和出队中应维护队中元素个数 count 的正确性。设计本题的循环队列类 CSqQueue1 如下：

```
const int MaxSize=100;                     //队列的容量
template<typename T>
class CSqQueue1                            //循环队列类模板
{
public:
    T* data;                               //存放队中的元素
    int front;                             //队头指针
    int count;                             //队中元素的个数
    CSqQueue1()                            //构造函数
```

```
    {
        data=new T[MaxSize];                //为 data 分配容量为 MaxSize 的空间
        front=0;                            //队头指针置初值
        count=0;                            //元素个数置初值
    }
    ~CSqQueue1()                            //析构函数
    {
        delete [] data;
    }
    //————————————循环队列基本运算算法————————————————————
    bool empty()                            //判队空运算
    {
        return count==0;
    }
    bool push(T e)                          //进队列运算
    {
        if (count==MaxSize)  return false;  //队满上溢出
        int rear1=(front+count)%MaxSize;    //求队尾(rear1 为局部变量)
        rear1=(rear1+1) % MaxSize;
        data[rear1]=e;
        count++;                            //元素个数增 1
        return true;
    }
    bool pop(T& e)                          //出队列运算
    {
        if (count==0) return false;         //队空下溢出
        front=(front+1)%MaxSize;
        e=data[front];
        count--;                            //元素个数减少 1
        return true;
    }
    bool gethead(T& e)                      //取队头运算
    {
        if (count==0) return false;         //队空下溢出
        int head=(front+1)%MaxSize;
        e=data[head];
        return true;
    }
};
```

说明：本例设计的循环队列中最多可保存 MaxSize 个元素。

从上述循环队列的设计看出，如果将 data 数组的容量改为可以扩展的，在队满时新建更大容量的数组 newdata 后，不能像顺序表、顺序栈那样简单地将 data 中的元素复制到 newdata 中，而需要按队列操作，将 data 中的所有元素出队后进队到 newdata 中，这里不再详述。


视频讲解

3.2.4 队列的链式存储结构及其基本运算算法的实现

队列的链式存储结构也是通过由结点构成的单链表实现的，此时只允许在单链表的表首进行删除操作(出队)和在单链表的表尾进行插入操作(进队)，这里的单链表是不带头结

点的，需要使用两个指针（即队首指针 front 和队尾指针 rear）标识，front 指向队首结点，rear 指向队尾结点。用于存储队列的单链表简称为**链队**。

链队的存储结构如图 3.22 所示，其中链队中存放元素的结点类型 LinkNode 定义如下：

```
template < typename T >
struct LinkNode                          //链队数据结点类型
{
    T data;                              //结点数据域
    LinkNode * next;                     //指向下一个结点
    LinkNode():next(NULL) {}             //构造函数
    LinkNode(T d):data(d),next(NULL) {}  //重载构造函数
};
```

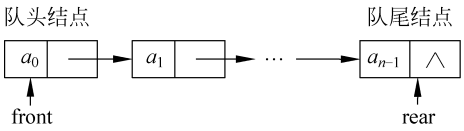


图 3.22　链队存储结构的示意图

设计链队类模板 LinkQueue 如下：

```
template < typename T >
class LinkQueue                          //链队类模板
{
public:
    LinkNode < T > * front;              //队头指针
    LinkNode < T > * rear;               //队尾指针
    //队列的基本运算算法
};
```

图 3.23 说明了一个链队的动态变化过程。图 3.23(a)是链队的初始状态，图 3.23(b)是进队 3 个元素后的状态，图 3.23(c)是出队两个元素后的状态。

归纳起来，初始时置 front＝rear＝NULL 的链队的四要素如下。

① 队空条件：front＝rear＝＝NULL，不妨仅以 front＝＝NULL 作为队空条件。

② 队满条件：由于只有内存溢出时才出现队满，因此通常不考虑这样的情况。

③ 元素 e 进队操作：在单链表的尾部插入一个存放 e 的 s 结点，并让队尾指针指向它。

④ 出队操作：取出队首结点的 data 值并将其从链队中删除。

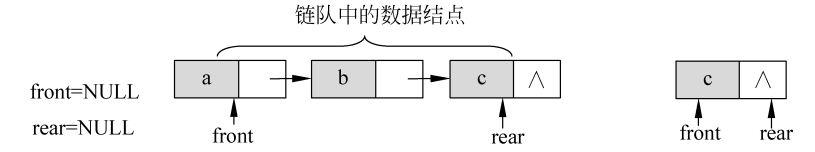


图 3.23　一个链队的动态变化过程

对应队列的基本运算算法如下：

1）链队的初始化和销毁

链队类的构造函数和析构函数如下：

```
LinkQueue()                                    //构造函数
{
    front=NULL;                                //置为不带头结点的空单链表
    rear=NULL;
}
~LinkQueue()                                   //析构函数
{
    LinkNode<T>* pre=front, *p;
    if (pre!=NULL)                             //非空队的情况
    {
        if (pre==rear)                         //只有一个数据结点的情况
            delete pre;                        //释放 pre 结点
        else                                   //有两个或多个数据结点的情况
        {
            p=pre->next;
            while (p!=NULL)
            {
                delete pre;                    //释放 pre 结点
                pre=p; p=p->next;              //pre、p 同步后移
            }
            delete pre;                        //释放尾结点
        }
    }
}
```

2) 判断队列是否为空：empty()

链队的 front 为空表示队列为空，返回 true，否则返回 false。对应的算法如下：

```
bool empty()                                   //判队空运算
{
    return rear==NULL;
}
```

3) 进队：push(T e)

创建存放元素 e 的结点 p。若原队列为空，则将 front 和 rear 均指向 p 结点，否则将 p 结点链接到单链表的末尾，并让 rear 指向它。对应的算法如下：

```
bool push(T e)                                 //进队运算
{
    LinkNode<T>* p=new LinkNode<T>(e);
    if (rear==NULL)                            //链队为空的情况
        front=rear=p;                          //新结点既是队首结点又是队尾结点
    else                                       //链队不空的情况
    {
        rear->next=p;                          //将 p 结点链接到队尾，并将 rear 指向它
        rear=p;
    }
    return true;
}
```

4）出队：pop(T& e)

原队为空时返回 false，若队中只有一个结点（此时 front 和 rear 都指向该结点），则取首结点 p 的 data 值赋给 e，删除结点 p 并置为空队，否则说明链队中有多个结点，取首结点 p 的 data 值赋给 e 并删除之，最后返回 true。对应的算法如下：

```
bool pop(T& e)                          //出队运算
{
    if (rear==NULL)    return false;    //队列为空
    LinkNode<T>* p=front;               //p 指向首结点
    if (front==rear)                    //队列中只有一个结点时
        front=rear=NULL;
    else                                //队列中有多个结点时
        front=front->next;
    e=p->data;
    delete p;                           //释放出队结点
    return true;
}
```

5）取队头元素：gethead(T& e)

与出队类似，但不需要删除首结点。对应的算法如下：

```
bool gethead(T& e)                      //取队头运算
{
    if (rear==NULL)    return false;    //队列为空
    e=front->data;                      //取首结点的值
    return true;
}
```

上述算法的时间复杂度均为 $O(1)$。

3.2.5　链队的应用算法设计示例

【例 3.13】　采用链队求解第 2 章中例 2.16 的约瑟夫问题。


视频讲解

解：先定义一个链队 qu，对于(n,m)约瑟夫问题，依次将 1～n 进队。循环 n 次出列 n 个小孩：依次出队 $m-1$ 次，将所有出队的元素立即进队（将他们从队头出队后插入队尾），再出队第 m 个元素并且输出（出队第 m 个小孩）。对应的程序如下：

```
#include "LinkQueue.cpp"                //包含链队类模板的定义
void Jsequence(int n,int m)             //输出约瑟夫序列
{
    int x;
    LinkQueue<int> qu;                  //定义一个链队
    for (int i=1;i<=n;i++)              //进队编号为 1 到 n 的 n 个小孩
        qu.push(i);
    for (int i=1;i<=n;i++)              //共出队 n 个小孩
    {
        int j=1;
        while (j<=m-1)                  //出队 m-1 个小孩，并将他们进队
        {
            qu.pop(x);
            qu.push(x);
            j++;
```

```
            }
            qu.pop(x);                          //出队第 m 个小孩,只出不进
            cout << x << " ";
        }
        cout << endl;
    }
    int main()
    {
        printf("测试 1: n=6,m=3\n");
        printf("   出列顺序:");
        Jsequence(6,3);
        printf("测试 2: n=8,m=4\n");
        printf("   出列顺序:");
        Jsequence(8,4);
        return 0;
    }
```

上述程序的执行结果如下:

```
测试 1: n=6,m=3
  出列顺序: 3 6 4 2 5 1
测试 2: n=8,m=4
  出列顺序: 4 8 5 2 1 3 7 6
```

说明: 与第 2 章中的例 2.16 相比,这里用带首尾结点指针的链队代替了循环单链表。


视频讲解

3.2.6 STL 中的 queue 队列容器

STL 中的 queue 队列容器是前面介绍的队列数据结构的一种实现,具有先进先出的特点。queue 容器只有一个进口(即队尾(back))和一个出口(即队头(front)),可以在队尾插入(进队)元素,在队头删除(出栈)元素,不允许像数组那样从前向后或者从后向前顺序遍历,所以 queue 容器没有 begin()/end()和 rbegin()/rend()这样的用于迭代器的成员函数。

与 stack 栈容器一样,queue 队列容器也是一种适配器容器,其底层容器必须提供 front()、back()、push_back()和 pop_front()等操作,因此 queue 的底层容器可以是 deque(默认)或者 list,不能是 vector,因为 vector 容器没有提供头部操作函数(如 pop_front())。

例如,以下语句用于定义 3 个 queue 对象:

```
queue<int> qu1;                         //定义一个整数队列 qu1
queue<int> qu2(qu1);                    //由 qu1 队列复制产生 qu2 队列
queue<int, list<int>> qu3;              //定义整数队列 qu3,以 list 作为底层容器
```

queue 的主要成员函数及其说明如表 3.4 所示。

表 3.4 queue 容器的主要成员函数及其说明

成员函数	说明
empty()	判断队列容器是否为空
size()	返回队列容器中的实际元素个数
front()	返回队头元素
back()	返回队尾元素
push(*e*)	元素 *e* 进队
pop()	元素出队

例如有以下程序：

```
#include <iostream>
#include <queue>
using namespace std;
int main()
{
    queue<int> qu;
    qu.push(1); qu.push(2); qu.push(3);
    printf("队头元素：%d\n",qu.front());
    printf("队尾元素：%d\n",qu.back());
    printf("出队顺序：");
    while (!qu.empty())                    //出队所有元素
    {
        printf("%d ",qu.front());
        qu.pop();
    }
    printf("\n");
    return 0;
}
```

在上述程序中建立了一个整数队列 qu，进队 3 个元素，取队头和队尾元素，然后出队所有元素并输出。程序的执行结果如下：

```
队头元素：1
队尾元素：3
出队顺序：1 2 3
```

【实战 3.3】　LeetCode225——用队列实现栈


视频讲解

问题描述：使用队列实现栈的下列操作。

void push(int x)：元素 x 进栈。

int pop()：删除并且返回栈顶元素。

int top()：返回栈顶元素。

bool empty()：返回栈是否为空。

【实战 3.4】　HDU1276——士兵队列训练问题


视频讲解

时间限制：1000ms；内存限制：32 768KB。

问题描述：某部队进行新兵队列训练，将新兵从 1 开始按顺序依次编号，并排成一行横队。训练的规则如下：从头开始按 1～2 报数，凡报到 2 的出列，剩下的向小序号方向靠拢；再从头开始按 1～3 报数，凡报到 3 的出列，剩下的向小序号方向靠拢；继续从头开始按 1～2 报数，……；之后从头开始轮流按 1～2 报数、按 1～3 报数，直到剩下的人数不超过 3 个为止。

输入格式：本题有多个测试数据组，第一行为组数 n，接着为 n 行新兵人数，新兵人数不超过 5000。

输出格式：共有 n 行，分别对应输入的新兵人数，每行输出剩下的新兵的最初编号，编号之间有一个空格。

3.2.7 队列的综合应用

本节通过用队列求解迷宫问题来讨论队列的应用。


视频讲解

❑ 问题描述

参见3.1.7节。

❑ 迷宫的数据组织

参见3.1.7节。

❑ 设计运算算法

用队列求迷宫路径的思路是从入口开始试探,当试探到一个方块 b 时,若该方块是出口,则输出对应的迷宫路径并返回,否则找其所有相邻可走方块,假设找到的方块的顺序为 $b_1,b_2,\cdots,b_k$,称 b 方块为这些方块的前驱方块,这些方块称为 b 方块的后继方块,按 b_1,$b_2,\cdots,b_k$ 的顺序试探每个方块。

为此采用一个队列存放这些方块,那么当找到出口后如何求出迷宫路径呢?由于试探的每个方块可能有多个后继方块,但一定只有唯一的前驱方块(除了入口方块外),为此设置队列中方块元素的类型如下:

```
struct Box                          //队列中方块元素的类型
{
    int i,j;                        //方块的行、列号
    Box * pre;                      //本路径中上一方块的地址
    Box() {}                        //构造函数
    Box(int i1,int j1)              //重载构造函数
    {
        i=i1; j=j1;
        pre=NULL;
    }
};
```

假设当前试探的方块位置是(i,j),对应的队列元素(Box对象)为 b,如图3.24所示,一次性地找它所有的相邻可走方块。假设有4个相邻可走方块(一般情况下最多3个),则这4个相邻可走方块均进队,同时置每个 b_i 的pre域(即前驱方块)为 b,即bi.pre=b。所以找到出口后,可以从出口通过pre域回推到入口,即找到一条迷宫路径。

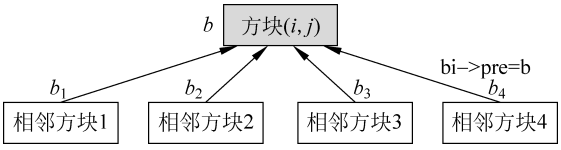


图3.24 当前方块 b 和相邻方块

查找一条从(xi,yi)到(xe,ye)的迷宫路径的过程是首先建立入口方块(xi,yi)的Box对象 b,将 b 进队,在队列qu不为空时循环:出队一次,称该出队的方块 b 为当前方块,做如下处理。

① 如果 b 是出口,则从 b 出发沿着pre域回推到出口,找到一条迷宫逆路径path,反向

输出该路径后返回 true。

② 否则，按顺时针方向一次性查找方块 b 的 4 个方位中的相邻可走方块，每个相邻可走方块均建立一个 Box 对象 b1，置 b1.pre＝b，将 b1 进 qu 队列。与用栈求解一样，一个方块进队后，将其迷宫值置为－1，以避免重复搜索。

如果到队空都没有找到出口，表示不存在迷宫路径，返回 false。

在图 3.13 所示的迷宫图中求入口(0,0)到出口(3,3)迷宫路径的搜索过程如图 3.25 所示，图中带“×”的方块表示没有相邻可走方块，每个方块旁的数字表示搜索顺序，找到出口后，通过虚线(即 pre)找到一条迷宫逆路径。

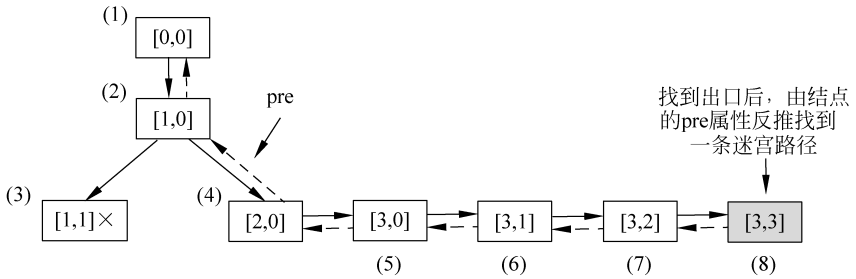


图 3.25　用队列求(0,0)到(3,3)迷宫路径的搜索过程

这里采用 queue 容器作为队列，但设计中存在一个问题，即在搜索迷宫路径中，队列中不再保存已经出队的方块，当找到出口时通过队列沿着 pre 反向推到迷宫路径中，由于某些前驱方块已经丢失无法找到完整的迷宫路径，为此采用指针方式，即 queue 中仅保存 Box 对象的指针，这样出队时仅出队指针，而指向的方块对象仍然存在，可以通过这些对象找到迷宫路径。用队列求解迷宫问题的 mgpath()算法如下：

```
bool mgpath(int xi,int yi,int xe,int ye)        //求一条从(xi,yi)到(xe,ye)的迷宫路径
{
    Box * b, * b1;
    queue<Box * > qu;                           //定义一个队列 qu
    b=new Box(xi,yi);                           //建立入口的对象 b
    qu.push(b);                                 //入口对象 b 进队，其 pre 置为 NULL
    mg[xi][yi]=-1;                              //为避免来回找相邻方块，置 mg 值为-1
    while (!qu.empty())                         //队不空时循环
    {
        b=qu.front();                           //取队头方块 b
        if (b->i==xe && b->j==ye)               //找到了出口，输出路径
        {
            disppath(qu);                       //输出一条迷宫路径
            return true;                        //找到一条迷宫路径后返回 true
        }
        qu.pop();                               //出队方块 b
        for (int di=0;di<4;di++)                //循环扫描每个方位，把每个可走的方块进队
        {
            int i=b->i+dx[di];                  //找 b 的 di 方位的相邻方块(i,j)
            int j=b->j+dy[di];
            if (i>=0 && i<m && j>=0 && j<n && mg[i][j]==0)
            {
```

```
                b1=new Box(i,j);          //(i,j)方块有效且可走,建立其队列对象 b1
                b1->pre=b;                //将该相邻方块进队,并置 pre 指向前驱方块
                qu.push(b1);
                mg[i][j]=-1;              //为避免来回找相邻方块,置 mg 值为-1
            }
        }
    }
    return false;                         //未找到任何路径时返回 false
}
```

当找到出口后,队头恰好为出口方块元素,通过队列反向搜索并输出一条迷宫路径的算法如下:

```
void disppath(queue<Box*>& qu)            //输出一条迷宫路径
{
    vector<Box> apath;                    //存放一条迷宫路径
    Box* b;
    b=qu.front();                         //从队头开始向入口方向搜索
    while (b!=NULL)
    {
        apath.push_back(Box(b->i,b->j));  //将搜索的方块添加到 apath 中
        b=b->pre;
    }
    cout << "一条迷宫路径: ";
    for (int i=apath.size()-1;i>=0;i--)   //反向输出构成一条正向迷宫路径
        cout << "[" << apath[i].i << "," << apath[i].j << "]  ";
    cout << endl;
}
```

❑ 设计主程序

设计以下主程序用于求图3.13所示的迷宫图中从(0,0)到(3,3)的一条迷宫路径:

```
int main()
{
    int xi=0,yi=0,xe=3,ye=3;
    printf("求(%d,%d)到(%d,%d)的迷宫路径\n",xi,yi,xe,ye);
    if (!mgpath(xi,yi,xe,ye))
        cout << "不存在迷宫路径\n";
    return 0;
}
```

❑ 程序执行结果

本程序的执行结果如下:

```
一条迷宫路径:[0,0] [1,0] [2,0] [3,0] [3,1] [3,2] [3,3]
```

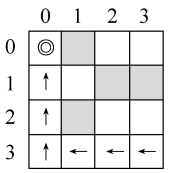


图3.26 用队列求出的一条迷宫路径

该路径如图3.26所示,迷宫路径上方块的箭头表示其前驱方块的方位。显然这个解是最优解,也就是最短路径。至于为什么用栈求出的迷宫路径不一定是最短路径,而用队列求出的迷宫路

径一定是最短路径，将在第 8 章中说明。

在上述 mgpath()算法中定义的 qu 队列存放的是 Box 对象指针，通过 pre 成员表示前驱关系（指向路径中前驱 Box 对象的指针），也可以不在队列中表示前驱关系，专门设置一个 pre[m][n]二维数组来表示前驱关系，或者采用非循环队列，用数组模拟，出队的元素仍然在数组中，可以用来找迷宫路径。


视频讲解

3.2.8　STL 中的双端队列和优先队列

在 STL 中还提供了另外两种非常有用的队列容器，即双端队列容器和优先队列容器。

1. 双端队列容器 deque


视频讲解

双端队列是在队列的基础上扩展而来的，其示意图如图 3.27 所示。双端队列与队列一样，元素的逻辑关系也是线性关系，但队列只能在一端进队，在另一端出队，而双端队列可以在两端进行进队和出队操作，具有队列和栈的特性，因此使用更加灵活。

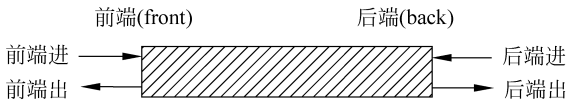


图 3.27　双端队列示意图

STL 中的双端队列 deque 是一种顺序容器，采用双向开口的连续线性空间，由若干个缓冲区块构成，每个块中元素的地址是连续的，块之间的地址是不连续的，如图 3.28 所示为 deque 容器的一般存储结构，系统有一个特定的机制维护这些块构成一个整体。

deque 的头、尾均能以常数时间插入和删除（vector 只能在尾端进行插入和删除），支持随机元素访问，但性能没有 vector 好；可以在中间位置插入和删除元素，但性能不及 list。与 vector 相比，deque 的容量大小是可以自动伸缩的，而 vector 的容量大小只会自动伸长，另外 deque 的空间重新分配比 vector 快。

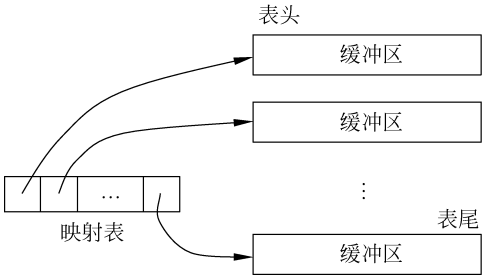


图 3.28　deque 容器的存储结构

定义 deque 双端队列容器的几种方式如下：

```
deque<int> dq1;                          //定义元素为 int 的双端队列 dq1
deque<int> dq2(10);                      //指定 dq2 的初始大小为 10 个 int 元素
deque<double> dq3(10,1.23);              //指定 dq3 的 10 个初始元素的初值为 1.23
deque<int> dq4(dq2.begin(),dq2.end());   //用 dq2 的所有元素初始化 dq4
```

deque 的主要成员函数及其说明如表 3.5 所示。

表 3.5　deque 容器的主要成员函数及其说明

成员函数	说　明
empty()	判断双端队列容器是否为空队
size()	返回双端队列容器中元素的个数
front()	返回队头元素
back()	返回队尾元素
push_front(*e*)	在队头插入元素 *e*
push_back(*e*)	在队尾插入元素 *e*
pop_front()	删除队头元素
pop_back()	删除队尾元素
erase()	从双端队列容器中删除一个或几个元素
clear()	删除双端队列容器中的所有元素
begin()	该函数的两个版本返回 iterator 或 const_iterator,引用容器首元素
end()	该函数的两个版本返回 iterator 或 const_iterator,引用容器尾元素的后一个位置
rbegin()	该函数的两个版本返回 reverse_iterator 或 const_reverse_iterator,引用容器尾元素
rend()	该函数的两个版本返回 reverse_iterator 或 const_reverse_iterator,引用容器首元素的前一个位置

例如有以下程序:

```
#include <iostream>
#include <deque>
using namespace std;
int disp(deque<int> & dq)                    //输出 dq 的所有元素
{
    deque<int>::iterator iter;               //定义迭代器 iter
    for (iter=dq.begin();iter!=dq.end();iter++)
        printf("%d ", *iter);
    printf("\n");
}
int main()
{
    deque<int> dq;                           //建立一个双端队列 dq
    dq.push_front(1);                        //队头插入 1
    dq.push_back(2);                         //队尾插入 2
    dq.push_front(3);                        //队头插入 3
    dq.push_back(4);                         //队尾插入 4
    printf("dq: "); disp(dq);
    dq.pop_front();                          //删除队头元素
    dq.pop_back();                           //删除队尾元素
    printf("dq: "); disp(dq);
    return 0;
}
```

在上述程序中定义了字符串双端队列 dq,利用插入和删除成员函数进行操作。程序的执行结果如下:

```
dq: 3 1 2 4
dq: 1 2
```

说明：如果仅使用双端队列的在同一端插入和删除的成员函数，该双端队列就变成了一个栈；如果仅使用双端队列的在不同端插入和删除的成员函数，该双端队列就变成了一个普通队列。


视频讲解

2. 优先队列

所谓优先队列，就是这样的一种队列，队列中的每个元素都有一个优先级，优先级越高越优先出队。优先级与元素值的大小不是一回事，需要专门指定，如果指定元素值越大优先级越高，即元素值越大越优先出队，则称这样的优先队列为大根堆；如果指定元素值越小优先级越高，即元素值越小越优先出队，则称这样的优先队列为小根堆。实际上，普通队列就是按元素进队的先后时间作为优先级，越先进队的元素越优先出队。

STL 中的优先队列是 priority_queue 容器，它和 stack/queue 一样，也是一种适配器容器，其底层容器必须是用数组实现的，可以是 vector(默认)或者 deque，不能是 list。priority_queue 对象的一般定义格式如下：

```
priority_queue < type, container, functional >
```

其中，type 参数指出数据类型，container 参数指出底层容器，functional 参数指出比较函数。在默认情况下(不加后面两个参数)是以 vector 为底层容器，以 less＜T＞为比较函数(即＜运算符对应的仿函数)。所以在定义优先队列时只使用第一个参数，该优先队列默认的是元素值越大越优先出队(大根堆)。

priority_queue 容器的主要成员函数如表 3.6 所示，它们与 queue 容器的成员函数类似。与 STL 中的 sort()算法一样，下面按 type 参数的数据类型分两种情况讨论 priority_queue 容器的使用。

表 3.6　priority_queue 容器的主要成员函数及其说明

成 员 函 数	说　　明
empty()	判断优先队列容器是否为空
size()	返回优先队列容器中实际元素的个数
push(*e*)	元素 *e* 进队
top()	获取队头元素
pop()	元素出队

1) type 为内置数据类型

对于 C/C++内置数据类型，默认的 functional 是 less＜T＞(小于比较函数)，即建立的是大根堆(元素值越大越优先出队)，可以改为 greater＜T＞(大于比较函数)，这样元素值越小优先级越高(称为小根堆)。例如，建立大根堆：

```
priority_queue < int > big_heap;                              //默认方式
priority_queue < int, vector < int >, less < int > > big_heap2;  //使用 less < T >比较函数
```

建立小根堆：

```
priority_queue<int,vector<int>,greater<int>> small_heap;    //使用 greater<T>比较函数
```

2) type 为自定义类型

对于自定义数据类型(如类或者结构体),在建立优先队列(堆)时需要比较两个元素的大小,即设置元素的比较方式,在元素的比较方式中指出按哪个成员值做比较以及大小关系(是越大越优先还是越小越优先)。

同样默认的比较函数是 less<T>(即小于比较函数),但需要重载该函数。另外,还可以重载函数调用运算符()。通过这些重载函数来设置元素的比较方式。

归纳起来,假设 Stud 类中包含学号 no 和姓名 name 数据成员,设计各种优先队列的3种方式如下。

方式1:在定义类或者结构体类型中重载<运算符(operator <),以指定元素的比较方式,例如 priority_queue<Stud> pq1 语句会调用默认的<运算符创建堆 pq1(是大根堆还是小根堆由<重载函数体确定)。

方式2:在定义类或者结构体中重载>运算符(operator >),以指定元素的比较方式,例如 priority_queue<Stud,vector<Stud>,greater<Stud> > pq2 语句会调用重载>运算符创建堆 pq2,此时需要指定优先队列的底层容器(这里为 vector,也可以是 deque)。

方式3:在单独定义的类或者结构体中重载函数调用运算符()(operator ()),以指定元素的比较方式,例如 priority_queue<Stud,vector<Stud>,StudCmp > pq3 语句会调用 StudCmp 的()运算符创建堆 pq3,此时需要指定优先队列的底层容器(这里为 vector,也可以是 deque)。

例如,以下程序分别采用上述3种方式创建3个优先队列:

```
#include <iostream>
#include <queue>
#include <string>
using namespace std;
class Stud                                          //定义类 Stud
{
public:
    int no;                                         //学号
    string name;                                    //姓名
    Stud(int n,string na)                           //构造函数
    {
        no=n;
        name=na;
    }
    bool operator<(const Stud& s) const             //重载<比较函数
    {
        return no<s.no;                             //按 no 越大越优先出队
    }
    bool operator>(const Stud& s) const             //重载>比较函数
    {
        return no>s.no;                             //按 no 越小越优先出队
    }
};
```

```
//类的比较函数,改写 operator()
class StudCmp                                          //含重载()成员函数的类
{
public:
    bool operator()(const Stud& s,const Stud& t) const
    {
        return s.name<t.name;                          //按 name 越大越优先出队
    }
};
int main()
{
    Stud a[]={Stud(2,"Mary"),Stud(1,"John"),Stud(5,"Smith")};
    int n=sizeof(a)/sizeof(a[0]);
    //*************************************************
    //方式 1: 使用 Stud 类的<比较函数定义大根堆 pq1
    //*************************************************
    priority_queue<Stud> pq1(a,a+n);
    cout << "pq1 出队顺序: ";
    while (!pq1.empty())                               //按 no 递减输出
    {
        cout << "[" << pq1.top().no << "," << pq1.top().name << "]\t";
        pq1.pop();
    }
    cout << endl;
    //*************************************************
    //方式 2: 使用 Stud 类的>比较函数定义小根堆 pq2
    //*************************************************
    priority_queue<Stud,deque<Stud>,greater<Stud>> pq2(a,a+n);
    cout << "pq2 出队顺序: ";
    while (!pq2.empty())                               //按 no 递增输出
    {
        cout << "[" << pq2.top().no << "," << pq2.top().name << "]\t";
        pq2.pop();
    }
    cout << endl;
    //*************************************************
    //方式 3: 使用 StudCmp 类的比较函数定义大根堆 pq3
    //*************************************************
    priority_queue<Stud,deque<Stud>,StudCmp> pq3(a,a+n);
    cout << "pq3 出队顺序: ";
    while (!pq3.empty())                               //按 name 递减输出
    {
        cout << "[" << pq3.top().no << "," << pq3.top().name << "]\t";
        pq3.pop();
    }
    cout << endl;
    return 0;
}
```

上述程序的执行结果如下：

```
pq1 出队顺序：[5,Smith]  [2,Mary]  [1,John]
pq2 出队顺序：[1,John]  [2,Mary]  [5,Smith]
pq3 出队顺序：[5,Smith]  [2,Mary]  [1,John]
```

说明：priority_queue 容器的实现原理将在第 10 章介绍，在此之前读者仅需要掌握该容器的基本应用。

3.3* 栈和队列的扩展——单调栈和单调队列


视频讲解

3.3.1 单调栈

若在应用栈时始终维护从栈底到栈顶的元素是有序的，称这样的栈为**单调栈**。也就是说，单调栈中的元素是有序的，单调栈分为单调递增栈和单调递减栈两种，前者从栈底到栈顶的元素是递增的，后者从栈底到栈顶的元素是递减的，如图 3.29 所示。单调栈可以直接采用 STL 的 stack 容器来实现。

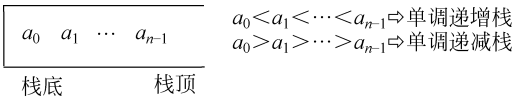


图 3.29 单调栈示意图

单调栈的简单应用是由输入序列产生一个单调栈。这里以单调递增栈为例，元素 e 进栈的过程是从栈顶开始，把大于 e 的元素出栈，直到遇到一个小于或等于 e 的元素或者栈为空时为止，然后把 e 进栈。对于单调递减栈，则每次弹出的是小于 e 的元素。

例如有一个整数序列 $a=\{3,4,2,6,4,5,2,3\}$，用 i 遍历 a，产生一个递减栈 st，过程如下：

① $i=0$，$a[0]=3$。栈 st 为空，将 $a[0]$进栈⇨st=(3)。

② $i=1$，$a[1]=4$。栈 st 非空，$a[1]<$栈顶元素不成立，出栈 3，栈为空，再将 $a[1]$进栈⇨st=(4)。

③ $i=2$，$a[2]=2$。栈 st 非空，$a[2]<$栈顶元素成立，将 $a[2]$进栈⇨st=(4,2)。

④ $i=3$，$a[3]=6$。栈 st 非空，$a[3]<$栈顶元素不成立，出栈 2，再出栈 4，栈为空，再将 $a[3]$进栈⇨st=(6)。

⑤ $i=4$，$a[4]=4$。栈 st 非空，$a[4]<$栈顶元素成立，将 $a[4]$进栈⇨st=(6,4)。

⑥ $i=5$，$a[5]=5$。栈 st 非空，$a[5]<$栈顶元素不成立，出栈 4，再将 $a[5]$进栈⇨st=(6,5)。

⑦ $i=6$，$a[6]=2$。栈 st 非空，$a[6]<$栈顶元素成立，将 $a[6]$进栈⇨st=(6,5,2)。

⑧ $i=7$，$a[7]=3$。栈 st 非空，$a[7]<$栈顶元素不成立，出栈 2，再将 $a[7]$进栈⇨st=(6,5,3)。

【例 3.14】 一个栈 st 中有若干个整数，设计一个算法利用一个辅助栈将该栈改变为单调递减栈。

解：设辅助栈为 st1，第一步是将 st 中的所有整数进入 st1 并使 st1 为一个单调递增栈，

第二步是依次将 st1 中的所有整数出栈并进栈 st，则 st 栈改变为单调递减栈。

实现第一步的操作是，当 st 栈不空时循环：st 栈出栈元素 e，将 st1 栈顶大于或等于 e 的元素退栈并进到 st 栈中，再将元素 e 进 st1 栈。最后将 st1 中的所有元素退栈并进到 st 栈中。

对应的算法如下：

```
void Sortst(stack<int>& st)
{
    stack<int> st1;                          //定义临时栈 st1
    int e;
    while (!st.empty())                      //将 st1 变为单调递增栈
    {
        e=st.top();                          //st 出栈元素 e
        st.pop();
        while (!st1.empty() &&e<st1.top())   //e 相对 st1 是反序时
        {
            st.push(st1.top());              //退栈 st1 中大于 e 的元素并进栈 st
            st1.pop();
        }
        st1.push(e);                         //将 e 进 st1 栈
    }
    while (!st1.empty())                     //将 st1 中的所有元素退栈并进到 st 栈中
    {
        e=st1.top();
        st1.pop();
        st.push(e);
    }
}
```

单调栈更复杂的应用是求一个序列中以某个元素为最值(最大值或者最小值)的最大延伸区间，比当前元素更大的前一个元素和后一个元素，比当前元素更小的前一个元素和后一个元素等。例如，$a[]=\{2,3,2,5,1,4\}$，从左到右遍历每一个整数 $a[i](0\leqslant i\leqslant n-1)$，并求出以 $a[i]$为最小值所能延伸的最大区间(即这个区间中的所有整数均大于或等于 $a[i]$)。例如 $a[0]=2$，以它为最小值所能延伸的最大区间是$\{2,3,2,5\}$；对于 $a[1]=3$，以它为最小值所能延伸的最大区间是$\{3\}$，以此类推。

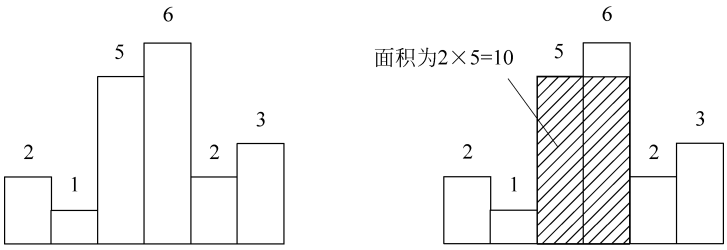
【实战 3.5】　LeetCode84——柱状图中最大的矩形

问题描述：给定 n 个非负整数，用来表示柱状图中各个柱子的高度。每个柱子彼此相邻，且宽度为 1，求在该柱状图中能够勾勒出来的矩形的最大面积。例如，如图 3.30(a)所示的柱状图，其中每个柱子的宽度为 1，给定的高度 heights[]=$\{2,1,5,6,2,3\}$，求出的最大面积的矩形如图 3.30(b)所示，其面积为 10。


视频讲解

要求设计相应的 largestRectangleArea()函数：

```
class Solution
{
public:
    int largestRectangleArea(vector<int>& heights)
    {  ...  }
};
```



(a) 6个柱子　　(b) 最大面积的矩形

图 3.30　一个柱状图及其求解结果


视频讲解

3.3.2　单调队列

若在应用队列时始终维护从队头到队尾的元素是有序的,称这样的队列为**单调队列**。也就是说,单调队列中的元素是有序的,单调队列分为单调递增队列和单调递减队列,前者从队头到队尾的元素是递增的,后者从队头到队尾的元素是递减的,如图 3.31 所示。

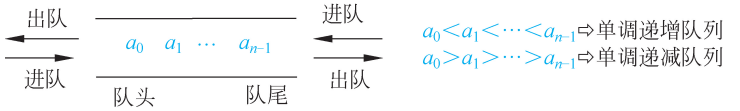


图 3.31　单调队列示意图

单调队列通常用于维护一个输入序列中指定区间的最值(即指定区间中的最大值或者最小值),假设区间的长度为 k,其基本操作是队头出队、队尾进队和出队:

① 队中元素的顺序(从队头到队尾)是输入序列中的元素从前向后的相对顺序。

② 保持队列的单调性,队头为最值,求最大值时采用单调递减队列 minqu,求最小值时采用单调递增队列 maxqu。

③ 元素 x 进队总是从队尾进,如果是单调递减队列,若 $x\leqslant$队尾元素,直接将 x 进队,否则从队尾出队元素,直到 $x\leqslant$队尾元素(称为去尾操作),再将 x 进队;如果是单调递增队列,若 $x\geqslant$队尾元素,直接将 x 进队,否则从队尾出队元素,直到 $x\geqslant$队尾元素,再将 x 进队。

④ 从队头出队超过区间的元素(过期元素),如果当前位置是 i,若 i-队头元素序号$\geqslant k$,则说明队头元素超过当前区间,需要出队(称为掐头操作)。所以在单调队列中需要保存元素的时间戳以便确定是否过期。

通常单调队列采用 STL 中的 deque 容器来实现,由于输入序列中的每个元素只会进队和出队一次,所以单调队列的时间复杂度是 $O(n)$,是非常高效的。

例如,一个整数序列 $a=\{3,1,2,5,4\}$,输出所有长度为 3 的区间的最小值,用 i 遍历 a,采用单调递增队列 qu,$k=3$,求解过程如下:

① $i=0$,$a[0]=3$。qu 为空,将 $a[0]$从队尾进队⇨qu=(3)。

② $i=1$,$a[1]=1$。$a[1]\geqslant$队尾元素不成立,做去尾操作,出队尾 3,将 1 从队尾进队⇨qu=(1)。

③ $i=2$,$a[2]=2$。$a[2]\geqslant$队尾元素成立,将 2 从队尾进队⇨qu=(1,2)。求得{3,1,2}区间的最小值为队头 1。

④ $i=3$，$a[3]=5$。$a[3]\geqslant$队尾元素成立，将 5 从队尾进队⇨qu=(1,2,5)。求得{1,2,5}区间的最小值为队头 1。

⑤ $i=4$，$a[4]=4$。$a[4]\geqslant$队尾元素不成立，做去尾操作，出队尾 5，将 4 从队尾进队⇨qu=(1,2,4)。再做掐头操作，出队头 1⇨qu=(2,4)。求得{2,5,4}区间的最小值为队头 2。

所以 $a=\{3,1,2,5,4\}$时长度为 3 的区间的最小值序列为 1,1,2。求区间的最大值序列与之相似。

【实战 3.6】　POJ2823——滑动窗口


视频讲解

时间限制：12 000ms；内存限制：65 536KB。

问题描述：给出一个长度为 $n\leqslant 10^6$ 的整数数组。有一个大小为 k 的滑动窗口，它从数组的最左边移动到最右边，用户只能在窗口中看到 k 个整数。每次滑动窗口向右移动一个位置。在如表 3.7 所示的例子中，该数组为{1,3,−1,−3,5,3,6,7}，k 为 3。

表 3.7　一个 $k=3$ 的滑动窗口例子

窗口位置	最小值	最大值
[1 3 −1] −3 5 3 6 7	−1	3
1 [3 −1 −3] 5 3 6 7	−3	3
1 3 [−1 −3 5] 3 6 7	−3	5
1 3 −1 [−3 5 3] 6 7	−3	5
1 3 −1 −3 [5 3 6] 7	3	6
1 3 −1 −3 5 [3 6 7]	3	7

确定每个位置的滑动窗口中的最大值和最小值。

输入格式：输入包含两行，第一行包含两个整数 n 和 k，它们是数组和滑动窗口的长度，第二行有 n 个整数。

输出格式：输出中有两行，第一行按从左到右的顺序给出每个位置的窗口中的最小值，第二行为对应的最大值。

3.4　练习题

3.4.1　问答题

1. 简述线性表、栈和队列的异同。

2. 设输入元素为 1、2、3、P 和 A，进栈次序为 123PA，元素经过栈后到达输出序列，当所有元素均到达输出序列后，有哪些序列可以作为高级语言的变量名？

3. 假设以 I 和 O 分别表示进栈和出栈操作，则初态和终态为栈空的进栈和出栈操作序列可以表示为仅由 I 和 O 组成的序列，称可以实现的栈操作序列为合法序列（例如 IIOO 为合法序列，IOOI 为非法序列）。试给出区分给定序列为合法序列或非法序列的一般准则。

4. 有 n 个不同元素的序列经过一个栈产生的出栈序列个数是多少？

5. 若一个栈的存储空间是 data[0..$n-1$]，则对该栈的进栈和出栈操作最多只能执行 n 次。这句话正确吗？为什么？

6. 若采用数组 data[0..$m-1$]存放栈元素,回答以下问题:

(1) 只能以 data[0]端作为栈底吗?

(2) 为什么不能以 data 数组的中间位置作为栈底?

7. 链栈只能顺序存取,而顺序栈不仅可以顺序存取,还能够随机存取。这句话正确吗?为什么?

8. 什么叫队列的"假溢出"?如何解决假溢出?

9. 假设循环队列的元素存储空间为 data[0..$m-1$],队头指针 f 指向队头元素,队尾指针 r 指向队尾元素的下一个位置(例如 data[0..5],队头元素为 data[2],则 front=2,队尾元素为 data[3],则 rear=4),则在少用一个元素空间的前提下,表示队空和队满的条件各是什么?

10. 在算法设计中有时需要保存一系列临时数据元素,如果先保存的后处理,应该采用什么数据结构存放这些元素?如果先保存的先处理,应该采用什么数据结构存放这些元素?

3.4.2 算法设计题

1. 给定一个字符串 str,设计一个算法,采用顺序栈判断 str 是否为形如"序列 1@序列 2"的合法字符串,其中序列 2 是序列 1 的逆序,在 str 中恰好只有一个@字符。

2. 假设有一个链栈 st,设计一个算法,出栈从栈顶开始的第 k 个结点。

3. 设计一个算法,利用顺序栈将一个十进制正整数 d 转换为 r($2\leqslant r\leqslant 16$)进制的数,要求 r 进制数采用字符串 string 表示。

4. 用于列车编组的铁路转轨网络是一种栈结构,如图 3.32 所示。其中,右边轨道是输入端,左边轨道是输出端。当右边轨道上的车皮编号顺序为 1、2、3、4 时,如果执行操作进栈、进栈、出栈、进栈、进栈、出栈、出栈、出栈,则在左边轨道上的车皮编号顺序为 2、4、3、1。设计一个算法,给定 n 个整数序列 a 表示右边轨道上的车皮编号顺序,用上述转轨栈对这些车皮重新编号,使得编号为奇数的车皮都排在编号为偶数的车皮的前面,要求产生所有操作的字符串 op 和最终结果字符串 ans。

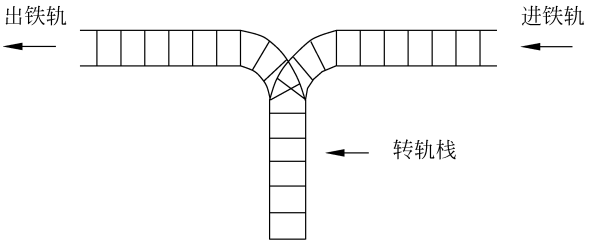
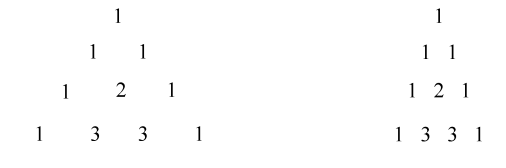


图 3.32 铁路转轨网络

5. 设计一个算法,利用一个顺序栈将一个循环队列中的所有元素倒过来,队头变队尾,队尾变队头。

6. 对于给定的正整数 n($n>2$),利用一个队列输出 n 阶杨辉三角形。5 阶杨辉三角形如图 3.33(a)所示,其输出结果如图 3.33(b)所示。

7. 有一个整数数组 a,设计一个算法,将所有偶数位的元素移动到所有奇数位的元素的前面,要求它们的相对次序不变。例如,$a=\{1,2,3,4,5,6,7,8\}$,移动后 $a=\{2,4,6,8,1,3,5,7\}$。



(a) n=5时的杨辉三角形　　(b) 输出结果

图 3.33　5 阶杨辉三角形及其生成过程

8. 设计一个循环队列 QUEUE<T>，用 data[0..MaxSize−1]存放队列元素，用 front 和 rear 分别作为队头和队尾指针，另外用一个标志 tag 标识队列可能空(false)或可能满(true)，这样加上 front==rear 可以作为队空或队满的条件。要求设计队列的相关基本运算算法。

3.5　上机实验题

3.5.1　基础实验题

1. 设计整数顺序栈的基本运算程序，并用相关数据进行测试。
2. 设计整数链栈的基本运算程序，并用相关数据进行测试。
3. 设计整数循环队列的基本运算程序，并用相关数据进行测试。
4. 设计整数链队的基本运算程序，并用相关数据进行测试。

3.5.2　应用实验题

1. 改进用栈求解迷宫问题的算法，累计如图 3.34 所示的迷宫的路径条数，并输出所有的迷宫路径。

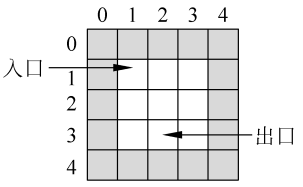


图 3.34　迷宫示意图

2. 括号匹配问题。在某个字符串(长度不超过 100)中有左括号、右括号和大小写字母，规定(与常见的算术表达式一样)任何一个左括号都从内到外与在它右边且距离最近的右括号匹配。编写一个实验程序，找到无法匹配的左括号和右括号，输出原来的字符串，并在下一行标出不能匹配的括号。不能匹配的左括号用"$"标出，不能匹配的右括号用"?"标出。例如，输出样例如下：

```
( (ABCD(x)
$ $
)(rttyy())sss)(
?             ?$
```

3. 修改第 3 章 3.2 节中的循环队列算法，增加数据成员 length 表示长度，并且其容量可以动态扩展，在进队元素时若容量满按两倍扩大容量，在出队元素时若当前容量大于初始容量并且元素的个数只有当前容量的 1/4，则缩小当前容量为一半。通过测试数据说明队列容量变化的情况。

4. 采用一个不带头结点、只有一个尾结点指针 rear 的循环单链表存储队列，设计出这

种链队的进队、出队、判队空和求队中元素个数的算法。

5. 设计一个队列类 QUEUE,其包含判断队列是否为空、进队和出队运算。要求用两个栈 st1、st2 模拟队列,其中栈用 stack<T>容器表示。

6. 设计一个栈类 STACK,其包含判断栈是否为空、进栈和出栈运算。要求用两个队列 qu1、qu2 模拟栈,其中队列用 queue<T>容器表示。

3.6 在线编程题

1. LeetCode150——逆波兰表达式求值
2. LeetCode622——设计循环队列
3. HDU5818——合并栈操作
4. HDU6215——暴力排序
5. HDU4699——编辑器
6. HDU6375——度度熊学队列
7. HDU4393——扔钉子
8. POJ3032——纸牌戏法
9. POJ2259——团队队列
10. POJ2559——最大矩形面积
11. POJ3984——迷宫问题
12. POJ1686——算术式子是否等效