第3章　栈和队列

栈和队列是两种特殊的线性表，从结构上讲，虽然它们都是线性表，但基本操作却是线性表操作的子集，是受限制的操作，在各种软件系统中有着广泛应用。本章讨论栈和队列的定义、表示方法、实现和一些典型的应用。

3.1　栈

3.1.1　栈的基本概念

栈(stack)是限定只在表头进行插入(入栈)与删除(出栈)操作的线性表，表头端称为栈顶，表尾端称为栈底。

设有栈 $S=(a_1,a_2,\cdots,a_n)$，则一般称 a_1 为栈底元素，a_n 为栈顶元素，按 $a_1,a_2,\cdots,a_n$ 的顺序依次进栈，出栈的第一个元素为栈顶元素，也就是说栈是按后进先出的原则进行进栈和出栈操作，如图 3.1 所示，所以栈可称为后进先出(last in first out，LIFO)的线性表，简称 LIFO 结构。

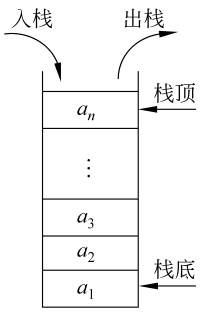


图 3.1　栈示意图

在实际应用中，栈包含了如下基本操作。

1) int Length() const

初始条件：栈已存在。

操作结果：返回栈元素个数。

2) bool Empty() const

初始条件：栈已存在。

操作结果：如栈为空，则返回 true，否则返回 false。

3) void Clear()

初始条件：栈已存在。

操作结果：清空栈。

4) void Traverse(void (* visit)(const ElemType &)) const

初始条件：栈已存在。

操作结果：从栈底到栈顶依次对栈的每个元素调用函数(* visit)。

5) bool Push(const ElemType &e)

初始条件：栈已存在。

操作结果：插入元素 e 为新的栈顶元素。

6) bool Top(ElemType &e) const

初始条件：栈已存在且非空。

操作结果：用 e 返回栈顶元素。

7) bool Pop(ElemType &e)

初始条件：栈已存在且非空。

操作结果：删除栈顶元素，并用 e 返回栈顶元素。

8) bool Pop()

初始条件：栈已存在且非空。

操作结果：删除栈顶元素。

3.1.2 顺序栈

与线性表类似，栈的实现也有两种方法：顺序栈和链栈。本节讨论顺序栈的实现，下一节讨论链栈的实现。

在顺序栈实现时，可利用一组地址连续的存储单元依次存放从栈底到栈顶的数据元素，将数据类型为 ElemType 的数据元素存储在数组中，并用 count 存储数组中存储的栈的实际元素个数。当 count=0 时表示栈为空，每当插入新的栈顶元素时，若栈未满，则操作成功，count 的值加 1；当删除栈顶元素时，若栈不空，则操作成功，count 的值减 1。具体类模板声明及实现如下：

```
//顺序栈类模板
template<class ElemType>
class SqStack
{
protected:
//数据成员
    ElemType *elems;                                          //元素存储空间
    int maxSize;                                              //栈最大元素个数
    int count;                                                //元素个数

public:
//抽象数据类型方法声明及重载编译系统默认方法声明
    SqStack(int size=DEFAULT_SIZE);                           //构造函数模板
    virtual ~SqStack();                                       //析构函数模板
    int Length() const;                                       //求栈长度
    bool Empty() const;                                       //判断栈是否为空
    void Clear();                                             //将栈清空
    void Traverse(void (*visit)(const ElemType &)) const;     //遍历栈
    bool Push(const ElemType &e);                             //入栈
    bool Top(ElemType &e) const;                              //返回栈顶元素
    bool Pop(ElemType &e);                                    //出栈
    bool Pop();                                               //出栈
    SqStack(const SqStack<ElemType>&source);                  //复制构造函数模板
    SqStack<ElemType>&operator=(const SqStack<ElemType>&source);
                                                              //重载赋值运算符
};


//顺序栈类模板的实现部分
```

```
template<class ElemType>
SqStack<ElemType>::SqStack(int size)
//操作结果:构造一个最大元素个数为 size 的空栈
{
    maxSize =size;                                          //最大元素个数
    elems =new ElemType[maxSize];                           //分配存储空间
    count =0;                                               //空栈元素个数为 0
}

template<class ElemType>
SqStack<ElemType>::~SqStack()
//操作结果:销毁栈
{
    delete []elems;                                         //释放存储空间
}

template <class ElemType>
int SqStack<ElemType>::Length() const
//操作结果:返回栈元素个数
{
    return count;                                           //count 表示栈元素个数
}

template<class ElemType>
bool SqStack<ElemType>::Empty() const
//操作结果:如栈为空,则返回 true,否则返回 false
{
    return count ==0;                                       //count ==0 表示栈为空
}

template<class ElemType>
void SqStack<ElemType>::Clear()
//操作结果:清空栈
{
    count =0;                                               //清空栈后元素个数为 0
}

template <class ElemType>
void SqStack<ElemType>::Traverse(void (*visit)(const ElemType &)) const
//操作结果:从栈底到栈顶依次对栈的每个元素调用函数(*visit)
{
    for (int temPos =1; temPos <=Length(); temPos++)
    {   //从栈底到栈顶对栈的每个元素调用函数(*visit)
        (*visit)(elems[temPos -1]);
    }
```

```
}

template<class ElemType>
bool SqStack<ElemType>::Push(const ElemType &e)
//操作结果:将元素 e 追加到栈顶,如成功则返加 true,如栈已满将返回 false
{
    if (count ==maxSize)
    {   //栈已满
        return false;                                   //入栈失败
    }
    else
    {   //操作成功
        elems[count] =e;                                //将元素 e 追加到栈顶
        count++;                                        //入栈成功后元素个数自加 1
        return true;                                    //入栈成功
    }
}

template<class ElemType>
bool SqStack<ElemType>::Top(ElemType &e) const
//操作结果:如栈非空,用 e 返回栈顶元素,返回 true,否则返回 false
{
    if(Empty())
    {   //栈空
        return false;                                   //失败
    }
    else
    {   //栈非空,操作成功
        e =elems[count -1];                             //用 e 返回栈顶元素
        return true;                                    //成功
    }
}

template<class ElemType>
bool SqStack<ElemType>::Pop(ElemType &e)
//操作结果:如栈非空,删除栈顶元素,并用 e 返回栈顶元素,返回 true,否则返回 false
{
    if (Empty())
    {   //栈空
        return false;                                   //失败
    }
    else
    {   //操作成功
        e =elems[count -1];                             //用 e 返回栈顶元素
        count--;                                        //出栈成功后元素个数自减 1
```

```
        return true;                                          //成功
    }
}

template<class ElemType>
bool SqStack<ElemType>::Pop()
//操作结果:如栈非空,删除栈顶元素,返回 true,否则返回 false
{
    if (Empty())
    {   //栈空
        return false;                                         //失败
    }
    else
    {   //操作成功
        count--;                                              //出栈成功后元素个数自减 1
        return true;                                          //成功
    }
}

template<class ElemType>
SqStack<ElemType>::SqStack(const SqStack<ElemType>&source)
//操作结果:由栈 source 构造新栈——复制构造函数模板
{
    maxSize =source.maxSize;                                  //最大元素个数
    elems =new ElemType[maxSize];                             //分配存储空间
    count =source.count;                                      //栈元素个数
    for (int temPos =1; temPos <=Length(); temPos++)
    {   //从栈底到栈顶对栈 source 的每个元素进行复制
        elems[temPos -1] =source.elems[temPos -1];
    }
}

template<class ElemType>
SqStack<ElemType>&SqStack<ElemType>::operator =
    (const SqStack<ElemType>&source)
//操作结果:将栈 source 赋值给当前栈——重载赋值运算符
{
    if (&source !=this)
    {
        maxSize =source.maxSize;                              //最大元素个数
        delete []elems;                                       //释放存储空间
        elems =new ElemType[maxSize];                         //分配存储空间
        count =source.count;                                  //复制栈元素个数
        for (int temPos =1; temPos <=Length(); temPos++)
        {   //从栈底到栈顶对栈 source 的每个元素进行复制
```

```
            elems[temPos - 1] = source.elems[temPos - 1];
        }
    }
    return * this;
}
```

例 3.1 读入一个整数 n 和 n 个整数，然后按相反的顺序输出这 n 个数。

本题可利用栈的后进先出的特性，在读取每个数据时将其入栈，然后再将数据出栈即可实现反序输出。

具体算法如下：




```
//文件路径名:s3_1\alg.h
void Reverse()
//初始条件:读入一个整数 n 和 n 个整数
//操作结果:按相反的顺序输出这 n 个整数
{
    int n, e;
    SqStack<int>temS;                                    //临时栈

    cout <<"输入一个整数:";
    cin >>n;

    while(n <=0 )
    {   //保证 n 为正整数
        cout <<"不能为负或 0,请重新输入 n:";
        cin >>n;
    }

    cout <<"请输入" <<n <<"个整数:" <<endl;
    for (int i =0; i <n; i++)
    {   //输入 n 个整数,并入栈
        cin >>e;
        temS.Push(e);
    }

    cout <<"按输入的相反顺序输出:" <<endl;
    while (!temS.Empty())
    {   //出栈,并输出
        temS.Pop(e);
        cout <<e <<" ";
    }
}
```

当栈满时将发生溢出，为避免这种情况的发生，需为栈设立一个足够大的存储空间，但如果空间过大，而栈中实际元素个数不多，则会浪费存储空间。当在程序中存在几个栈时，

在实际运行中，可能有的栈膨胀过快，很快产生溢出，而有的栈可能还有许多空余空间。为避免这种情况，比如只有两个栈时，可以定义一个足够大的栈空间，此存储空间的两端分别设为两个栈的栈底，两个栈的栈顶都向中间伸展，直到两个栈顶相遇才发生溢出，如图 3.2 所示。

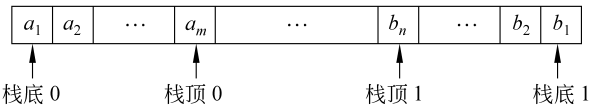


图 3.2　两个栈共享存储空间示意图

说明：对于 $n(n>2)$ 个栈的共享存储空间的情形，处理十分复杂，并且在插入和删除时可能会出现移动大量元素的情况，时间代价较高，解决的办法就是采用链式存储方式。

3.1.3　链式栈

在程序中同时使用多个栈的情形下，使用链式栈不但可以提高存储效率，同时还可达到共享存储空间的目的。

链式栈的结构如图 3.3 所示，入栈和出栈操作都非常简单，一般都不使用头节点直接实现，下面是链式栈的类模板声明和实现。

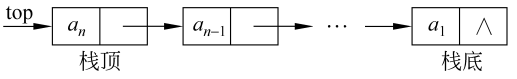


图 3.3　链式栈示意图

```
//链栈类模板
template<class ElemType>
class LinkStack
{
protected:
//数据成员
    Node<ElemType> * top;                                          //栈顶指针
    int count;                                                     //元素个数

public:
//抽象数据类型方法声明及重载编译系统默认方法声明
    LinkStack();                                                   //无参数的构造函数模板
    virtual ~LinkStack();                                          //析构函数模板
    int Length() const;                                            //求栈长度
    bool Empty() const;                                            //判断栈是否为空
    void Clear();                                                  //将栈清空
    void Traverse(void ( * visit)(const ElemType &)) const ;       //遍历栈
    bool Push(const ElemType &e);                                  //入栈
    bool Top(ElemType &e) const;                                   //返回栈顶元素
    bool Pop(ElemType &e);                                         //出栈
    bool Pop();                                                    //出栈
```

```
    LinkStack(const LinkStack<ElemType>&source);          //复制构造函数模板
    LinkStack<ElemType>&operator=(const LinkStack<ElemType>&source);
                                                          //重载赋值运算符
};


//链栈类模板的实现部分

template<class ElemType>
LinkStack<ElemType>::LinkStack()
//操作结果:构造一个空栈表
{
    top=NULL;                                             //构造栈顶指针
    count=0;                                              //初始化元素个数
}

template<class ElemType>
LinkStack<ElemType>::~LinkStack()
//操作结果:销毁栈
{
    Clear();                                              //清空栈
}

template <class ElemType>
int LinkStack<ElemType>::Length() const
//操作结果:返回栈元素个数
{
    return count;                                         //count 表示栈元素个数
}

template<class ElemType>
bool LinkStack<ElemType>::Empty() const
//操作结果:如栈为空,则返回 true,否则返回 false
{
    return count==0;                                      //count==0 表示栈为空
}

template<class ElemType>
void LinkStack<ElemType>::Clear()
//操作结果:清空栈
{
    while (!Empty())
    {   //表栈非空,则出栈
        Pop();                                            //出栈
    }
```

```
}

template <class ElemType>
void LinkStack<ElemType>::Traverse(void (*visit)(const ElemType &)) const
//操作结果:从栈底到栈顶依次对栈的每个元素调用函数(*visit)
{
    Node<ElemType> *temPtr;          //临时指针变量
    LinkStack<ElemType>temS;         //临时栈,temS 中元素顺序与当前栈元素顺序相反
    for (temPtr =top; temPtr !=NULL; temPtr =temPtr->next)
    {   //用 temPtr 依次指向当前栈的每个元素
        temS.Push(temPtr->data);     //对当前栈的每个元素入栈到 temS 中
    }

    for (temPtr =temS.top; temPtr !=NULL; temPtr =temPtr->next)
    {   //用 temPtr 从栈顶到栈底依次指向栈 temS 的每个元素
        (*visit)(temPtr->data);      //对栈 temS 的每个元素调用函数(*visit)
    }
}

template<class ElemType>
bool LinkStack<ElemType>::Push(const ElemType &e)
//操作结果:将元素 e 追加到栈顶,如成功则返加 true,否则如动态内存已耗尽将返回 false
{
    Node<ElemType> *newTop =new Node<ElemType>(e, top);
    if (newTop ==NULL)
    {   //动态内存耗尽
        return false;                //失败
    }
    else
    {   //操作成功
        top =newTop;
        count++;                     //入栈成功后元素个数加 1
        return true;                 //成功
    }
}

template<class ElemType>
bool LinkStack<ElemType>::Top(ElemType &e) const
//操作结果:如栈非空,用 e 返回栈顶元素,返回 true,否则返回 false
{
    if(Empty())
    {   //栈空
        return false;                //失败
    }
    else
```

```
    {   //栈非空,操作成功
        e =top->data;                   //用 e 返回栈顶元素
        return true;                    //成功
    }
}

template<class ElemType>
bool LinkStack<ElemType>::Pop(ElemType &e)
//操作结果:如栈非空,删除栈顶元素,并用 e 返回栈顶元素,返回 true,否则返回 false
{
    if (Empty())
    {   //栈空
        return false;                   //失败
    }
    else
    {   //操作成功
        Node<ElemType> * oldTop =top; //旧栈顶
        e =oldTop->data;                //用 e 返回栈顶元素
        top =oldTop->next;              //top 指向新栈顶
        delete oldTop;                  //删除旧栈顶
        count--;                        //出栈成功后元素个数自减 1
        return true;                    //功能
    }
}

template<class ElemType>
bool LinkStack<ElemType>::Pop()
//操作结果:如栈非空,删除栈顶元素,返回 true,否则返回 false
{
    if (Empty())
    {   //栈空
        return false;                   //失败
    }
    else
    {   //操作成功
        Node<ElemType> * oldTop =top; //旧栈顶
        top =oldTop->next;              //top 指向新栈顶
        delete oldTop;                  //删除旧栈顶
        count--;                        //出栈成功后元素个数自减 1
        return true;                    //功能
    }
}

template<class ElemType>
LinkStack<ElemType>::LinkStack(const LinkStack<ElemType>&source)
```

```
//操作结果:由栈 source 构造新栈——复制构造函数模板
{
    if (source.Empty())
    {   //source 为空
        top =NULL;                                          //构造栈顶指针
        count =0;                                           //初始化元素个数
    }
    else
    {   //source 非空,复制栈
        top =new Node<ElemType>(source.top->data);          //生成当前栈项
        count =source.count;                                //栈元素个数
        Node<ElemType> * buttomPtr =top;                    //当前栈底指针
        for (Node<ElemType> * temPtr =source.top->next; temPtr !=NULL;
            temPtr =temPtr->next)
        {   //用 temPtr 依次指向其余元素
            buttomPtr->next =new Node<ElemType>(temPtr->data);
                                                            //向栈底追加元素
            buttomPtr =buttomPtr->next;                     //buttomPtr 指向新栈底
        }
    }
}


template<class ElemType>
LinkStack<ElemType>&LinkStack<ElemType>::operator =
    (const LinkStack<ElemType>&source)
//操作结果:将栈 source 赋值给当前栈——重载赋值运算符
{
    if (&source !=this)
    {
        if (source.Empty())
        {   //source 为空
            top =NULL;                                      //构造栈顶指针
            count =0;                                       //初始化元素个数
        }
        else
        {   //source 非空,复制栈
            Clear();                                        //清空当前栈
            top =new Node<ElemType>(source.top->data);      //生成当前栈项
            count =source.count;                            //栈元素个数
            Node<ElemType> * buttomPtr =top;                //当前栈底指针
            for (Node<ElemType> * temPtr =source.top->next; temPtr !=NULL;
                temPtr =temPtr->next)
            {   //用 temPtr 依次指向其余元素
                buttomPtr->next =new Node<ElemType>(temPtr->data);
```

```
                                                    //向栈底追加元素
                buttomPtr =buttomPtr->next;         //buttomPtr 指向新栈底
            }
        }
    }
    return *this;
}
```

例 3.2 设计一个算法判别用字符串表示的表达式中括号“()”“[]”“{}”是否配对出现。

例如字符串

$$\{a*[c+d*(e+f)]$$

漏掉了大括号“}”。

字符串

$$\{a*[b+c*(e-f])\}$$

虽然有同等数量的左右大中小括号,但仍是不匹配的括号,第 1 个右中括号“]”和最近的左小括号“(”不匹配。

下面通过一个算法来实现检查一个输入的字符串中括号是否正确匹配,算法思路如下:

用一个字符串表示一个表达式,从左向右依次扫描各字符;

如果读入的字符为“(”“[”或“{”,则进栈;

若读入的字符为“)”,如栈空则左右括号不匹配,否则如栈顶的括号是“(”,则出栈,否则不匹配(这时栈顶为“[”或“{”);

若读入的字符为“]”,如栈空则左右括号不匹配,否则如栈顶的括号是“[”,则出栈,否则不匹配(这时栈顶为“(”或“{”);

若读入的字符为“}”,如栈空则左右括号不匹配,否则如栈顶的括号是“{”,则出栈,否则不匹配(这时栈顶为“(”或“[”);

若当前读入的字符是其他字符,则继续读入;

扫描完各字符后,如栈为空,则左右括号匹配,否则左右括号不匹配。

具体算法如下:

```
//文件路径名:s3_2\alg.h
bool Match(char *s)
//操作结果:判别用字符串 s 表示的表达式中大、中和小括号是否配对出现
{
    LinkStack<char>temS;                            //临时栈
    char temCh;                                     //临时字符

    for (int i =0; i <strlen(s); i++)
    {   //从左向右依次扫描各字符
        if (s[i] =='(' || s[i] =='[' || s[i] =='{')
        {   //如读入的字符为"("、"["或"{",则进栈
            temS.Push(s[i]);
        }
        else if (s[i] ==')')
```

```
        {   //读入的字符为')'
            if (temS.Empty())
            {   //如栈空则左右括号不匹配
                return false;
            }
            else if (temS.Top(temCh), temCh =='(')
            {   //如栈顶的括号是'(',则出栈
                temS.Pop();
            }
            else
            {   //否则不匹配(这时栈顶为'['或'{')
                return false;
            }
        }
        else if (s[i] ==']')
        {   //读入的字符为']'
            if (temS.Empty())
            {   //如栈空则左右括号不匹配
                return false;
            }
            else if (temS.Top(temCh), temCh  ==']')
            {   //如栈顶的括号是'[',则出栈
                temS.Pop();
            }
            else
            {   //否则不匹配(这时栈顶为'('或'{')
                return false;
            }
        }
        else if (s[i] =='}')
        {   //读入的字符为'}'
            if (temS.Empty())
            {   //如栈空则左右括号不匹配
                return false;
            }
            else if (temS.Top(temCh), temCh  =='{')
            {   //如栈顶的括号是'{',则出栈
                temS.Pop();
            }
            else
            {   //否则不匹配(这时栈顶为'('或'[')
                return false;
            }
        }
    }
```

```
    if (temS.Empty())
    {   //栈空则左右括号匹配
        return true;
    }
    else
    {   //栈不空则左右括号不匹配
        return false;
    }
}
```

3.2 队　　列

3.2.1 队列的基本概念

队列(queue)是一种先进先出(first in first out,FIFO)的线性表,只允许在一端进行插入(入队)操作,在另一端进行删除(出队)操作。

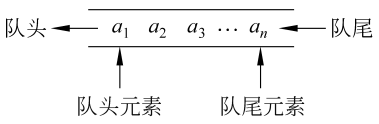


图 3.4　队列示意图

在队列中,允许入队操作的一端称为队尾,允许出队操作的一端称为队头,如图 3.4 所示。

设有队列 $q=(a_1,a_2,\cdots,a_n)$,则一般 a_1 称为队头元素,a_n 称为队尾元素,队列中元素按 $a_1,a_2,\cdots,a_n$ 的顺序入队,同时也按相同的顺序出队,队列的典型应用是操作系统中的作业排队,在实际应用中,队列包含了如下基本操作。

1) int Length() const

初始条件:队列已存在。

操作结果:返回队列长度。

2) bool Empty() const

初始条件:队列已存在。

操作结果:如队列为空,则返回 true,否则返回 false。

3) void Clear()

初始条件:队列已存在。

操作结果:清空队列。

4) void Traverse(void (*visit)(const ElemType &)) const

初始条件:队列已存在。

操作结果:依次对队列的每个元素调用函数(*visit)。

5) bool OutQueue(ElemType &e)

初始条件:队列非空。

操作结果:删除队头元素,并用 e 返回其值。

6) bool OutQueue()

初始条件:队列非空。

操作结果:删除队头元素。

7) bool GetHead(ElemType &e) const

初始条件：队列非空。

操作结果：用 e 返回队头元素。

8) bool InQueue(const ElemType &e)

初始条件：队列已存在。

操作结果：插入元素 e 为新的队尾。

除了上面定义的队列而外，还有一种限定性数据结构——双端队列(deque)，双端队列是插入和删除限定在线性表的两端进行的线性表，如图 3.5 所示。

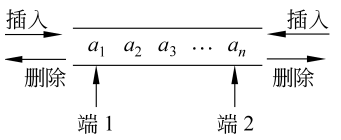
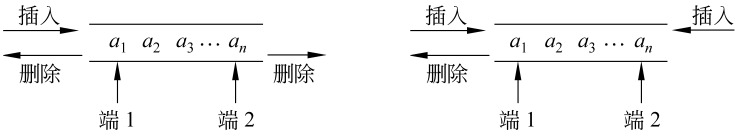


图 3.5　双端队列示意图

在实际应用中，可以有输入受限的双端队列，也就是允许在一端进行插入和删除操作，在另一端只允许进行删除操作，如图 3.6(a)所示。还有输出受限的双端队列，也就是允许在一端进行插入和删除操作，在另一端只允许进行插入操作，如图 3.6(b)所示。



(a) 输入受限的双端队列　　(b) 输出受限的双端队列

图 3.6　输入输出受限双端队列示意图

说明：虽然双端队列看起来具有更大的优越性，但在实际应用中主要还是使用栈和队列。所以对双端队列不进行详细讨论。

3.2.2　链队列

队列分为顺序存储结构和链式存储结构两种。

本节先讨论队列的链式存储结构，用链表表示的队列称为链队列，一个链队列应有两个分别指示队头与队尾的指针(分别称为头指针与尾指针)，如图 3.7 所示。

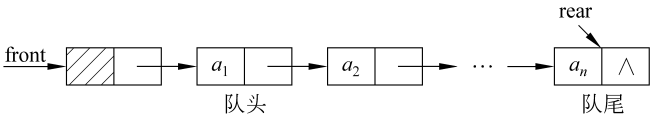


图 3.7　链队列示意图

如果要从队列中退出一个元素，必须从单链表的首元素节点中取出队头元素并删除此节点，而入队的新元素存放在队尾处，也就是单链表最后一个元素的后面，此节点将成为新的队尾。

说明：链队列适合于数据元素变动比较大的情形，一般不存在溢出的问题。如果程序中要使用多个队列，最好使用链队列，以避免出现存储分配的问题和数据元素的移动。

下面是链队列的类模板声明及相应成员函数模板的实现。

```
//链队列类模板

template<class ElemType>
class LinkQueue
{
protected:
//数据成员:
    Node<ElemType> * front, * rear;                          //队头队尾指针
    int count;                                               //元素个数

public:
//抽象数据类型方法声明及重载编译系统默认方法声明:
    LinkQueue();                                             //无参数的构造函数模板
    virtual ~LinkQueue();                                    //析构函数模板
    int Length() const;                                      //求队列长度
    bool Empty() const;                                      //判断队列是否为空
    void Clear();                                            //将队列清空
    void Traverse(void ( * visit)(const ElemType &)) const ; //遍历队列
    bool OutQueue(ElemType &e);                              //出队操作
    bool OutQueue();                                         //出队操作
    bool GetHead(ElemType &e) const;                         //取队头操作
    bool InQueue(const ElemType &e);                         //入队操作
    LinkQueue(const LinkQueue<ElemType>&source);             //复制构造函数模板
    LinkQueue<ElemType>&operator =(const LinkQueue<ElemType>&source);
                                                             //重载赋值运算符
};

//链队列类模板的实现部分

template<class ElemType>
LinkQueue<ElemType>::LinkQueue()
//操作结果:构造一个空队列
{
    rear =front =new Node<ElemType>;                         //生成头节点
    count =0;                                                //初始化元素个数
}

template<class ElemType>
LinkQueue<ElemType>::~LinkQueue()
//操作结果:销毁队列
{
    Clear();                                                 //清空队列
    delete front;                                            //释放头节点所占空间
}
```

```
template<class ElemType>
int LinkQueue<ElemType>::Length() const
//操作结果:返回队列长度
{
    return count;                                          //count 表示队列元素个数
}

template<class ElemType>
bool LinkQueue<ElemType>::Empty() const
//操作结果:如队列为空,则返回 true,否则返回 false
{
    return count ==0;                                      //count ==0 表示队列为空
}

template<class ElemType>
void LinkQueue<ElemType>::Clear()
//操作结果:清空队列
{
    while (!Empty())
    {   //队列非空,则出列
        OutQueue();                                        //出列
    }
}

template <class ElemType>
void LinkQueue<ElemType>::Traverse(void ( * visit)(const ElemType &)) const
//操作结果:依次对队列的每个元素调用函数( * visit)
{
    for (Node<ElemType> * temPtr =front->next; temPtr !=NULL;
        temPtr =temPtr->next)
    {   //对队列的每个元素调用函数( * visit)
        ( * visit)(temPtr->data);
    }
}


template<class ElemType>
bool LinkQueue<ElemType>::OutQueue(ElemType &e)
//操作结果:如果队列非空,那么删除队头元素,并用 e 返回其值,返回 true,否则返回 false
{
    if (!Empty())
    {   //队列非空
        Node<ElemType> * temPtr =front->next;          //指向队列头素
        e =temPtr->data;                               //用 e 返回队头元素
        front->next =temPtr->next;                     //front->next 指向下一元素
```

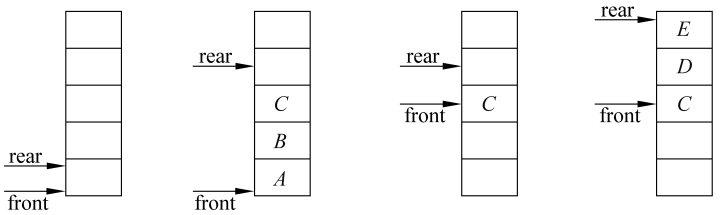
```
        if (rear ==temPtr)
        {   //表示出队前队列中只有一个元素,出队后为空队列
            rear =front;
        }
        delete temPtr;                                  //释放出队的节点
        count--;                                        //出队成功后元素个数自减 1
        return true;                                    //成功
    }
    else
    {   //队列为空
        return false;                                   //失败
    }
}

template<class ElemType>
bool LinkQueue<ElemType>::OutQueue()
//操作结果:如果队列非空,那么删除队头元素,返回 true,否则返回 false
{
    if (!Empty())
    {   //队列非空
        Node<ElemType> * temPtr =front->next;           //指向队列头素
        front->next =temPtr->next;                      //front->next 指向下一元素
        if (rear ==temPtr)
        {   //表示出队前队列中只有一个元素,出队后为空队列
            rear =front;
        }
        delete temPtr;                                  //释放出队的节点
        count--;                                        //出队成功后元素个数自减 1
        return true;                                    //成功
    }
    else
    {   //队列为空
        return false;                                   //失败
    }
}

template<class ElemType>
bool LinkQueue<ElemType>::GetHead(ElemType &e) const
//操作结果:如果队列非空,那么用 e 返回队头元素,返回 true,
//否则返回 false
{
    if (!Empty())
    {   //队列非空
        Node<ElemType> * temPtr =front->next;           //指向队列头素
        e =temPtr->data;                                //用 e 返回队头元素
```

```
        return true;                                        //成功
    }
    else
    {   //队列为空
        return false;                                       //失败
    }
}

template<class ElemType>
bool LinkQueue<ElemType>::InQueue(const ElemType &e)
//操作结果:插入元素 e 为新的队尾,插入成功 true,否则返回 false
{
    Node<ElemType> * temPtr =new Node<ElemType>(e);   //生成新节点
    if (temPtr ==NULL)
    {   //动态内存耗尽
        return false;                                       //失败
    }
    else
    {   //操作成功
        rear->next =temPtr;                                 //新节点追加在队尾
        rear =temPtr;                                       //rear 指向新队尾
        count++;                                            //入队成功后元素个数加 1
        return true;                                        //成功
    }
}

template<class ElemType>
LinkQueue<ElemType>::LinkQueue(const LinkQueue<ElemType>&source)
//操作结果:由队列 source 构造新队列——复制构造函数模板
{
    rear =front =new Node<ElemType>;                    //生成头节点
    count =0;                                           //初始化元素个数
    for (Node<ElemType> * temPtr =source.front->next; temPtr !=NULL;
        temPtr =temPtr->next)
    {   //对 source 队列的每个元素对当前队列作入队列操作
        InQueue(temPtr->data);
    }
}

template<class ElemType>
LinkQueue<ElemType>&LinkQueue<ElemType>::operator =(
    const LinkQueue<ElemType>&source)
//操作结果:将队列 source 赋值给当前队列——重载赋值运算符
{
    if (&source !=this)
```

```
    {
        Clear();                                          //清空当前队列
        for (Node<ElemType> * temPtr =source.front->next; temPtr !=NULL;
            temPtr =temPtr->next)
        {   //对 source 队列的每个元素对当前队列作入队列操作
            InQueue(temPtr->data);
        }
    }
    return * this;
}
```

3.2.3 循环队列——队列的顺序存储结构

用 C++ 描述队列的顺序存储结构，就是利用一个容量是 maxSize 的一维数组 elems 作为队列元素的存储结构，其中 front 和 rear 分别表示队头和队尾，maxSize 是队列的最大元素个数，如图 3.8 所示。



(a) 空队列　(b) A、B、C入列　(c) A、B出列　(d) D、E入列

图 3.8　顺序队列的入列和出列示意图

设队列分配的最大空间为 5，当队列处于图 3.8(d)所示状态时不可再继续插入新元素，否则会因为数组越界而出错。当队列实际可用的空间还没有使用完的情况下，插入一个新元素而产生溢出的现象称为假溢出。解决假溢出的一个技巧是将顺序队列从逻辑上看成一个环，形成循环队列。循环队列的首尾相接，当队头 front 和队尾 rear 进入 maxSize－1 时，再进一个位置就自动移动到 0。此操作可用取余运算(%)简单地实现。

队头进 1：front＝(front＋1)%maxSize。

队尾进 1：rear＝(rear＋1)%maxSize。

循环队列如图 3.9 所示，在图 3.9(a)中，元素 A、B、C 相继入列，在图 3.9(b)中，元素 D、E、F 相继入列，这时队满了。从图中可知，队满时

```
front==rear
```

如果图 3.9(a)中元素 A、B、C 相继出列，则可得到如图 3.9(c)所示的空队列，这时也有

```
front=rear
```

由此可知，仅从 front＝＝rear 是无法判断是队空还是队满的。可通过以下 3 种方法进行处理。

(1) 另设一个标志符来区别队列是空还是满。