第3章　栈和队列

栈和队列是两种常用的数据结构，广泛应用在操作系统、编译程序等各种软件系统中。从数据结构角度看，栈和队列也是线性表，但栈和队列的基本操作是线性表操作的子集，它们是操作受限的线性表。从数据类型角度看，栈和队列是两种不同于线性表的重要的抽象数据类型。本章将讨论栈和队列的基本概念、存储结构和相关运算，以及栈和队列的应用实例。

3.1　栈

栈是一种常用而且重要的数据结构，它在计算机科学中应用非常广泛。例如，编译器对表达式的计算和表达式括号的匹配、计算机系统处理函数调用时参数的传递和函数值的返回等，都需要使用到栈。

3.1.1　栈的定义

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

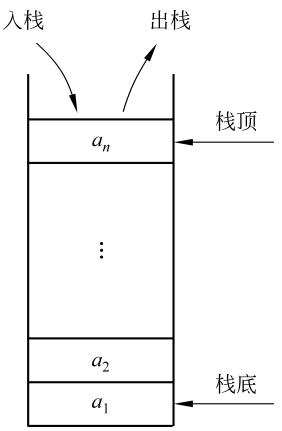


图 3-1　栈的示意图

设有栈 $S=(a_1,a_2,\cdots,a_n)$，则称 a_1 为栈底元素，称 a_n 为栈顶元素。如图 3-1 所示，栈中元素按 $a_1,a_2,\cdots,a_n$ 的次序进栈，出栈的第一个元素为栈顶元素，也就是说，栈的修改是按后进先出的原则进行的。因此，栈又称为后进先出(Last In First Out，LIFO)的线性表。

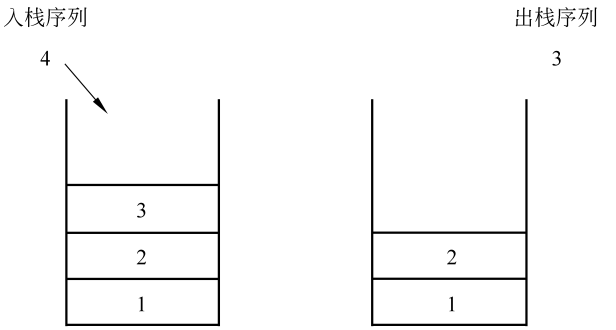
在日常生活中，有很多类似栈的例子。例如，洗干净的盘子总是逐个叠放在已经洗好的盘子上面，而使用时从上往下逐个取。栈的操作特点正是上述实际应用的抽象。在程序设计中，如果需要按照与保存数据时相反的顺序使用数据，则可以利用栈实现。例如，将浏览过的网址用栈进行存储，每访问一个网页，将其地址存放到栈顶，按“后退”按钮即可沿相反次序返回此前刚访问过的页面。

【例 3.1】 若元素的入栈序列为 1234，能否得到 3142 的出栈序列？

解析：为了让 3 作为第一个出栈元素，将 1、2、3 依次入栈，之后 3 出栈，如图 3-2 所示，此时要么 2 出栈，要么 4 入栈后出栈，出栈的第 2 个元素只能是 2 或 4，不可能是 1，因此得不到 3142 的出栈序列。

【例 3.2】 假设用 I 和 O 分别表示入栈和出栈操作，若元素的入栈顺序为 1234，为了得到 1342 的出栈序列，试给出相应的 I 和 O 操作串。

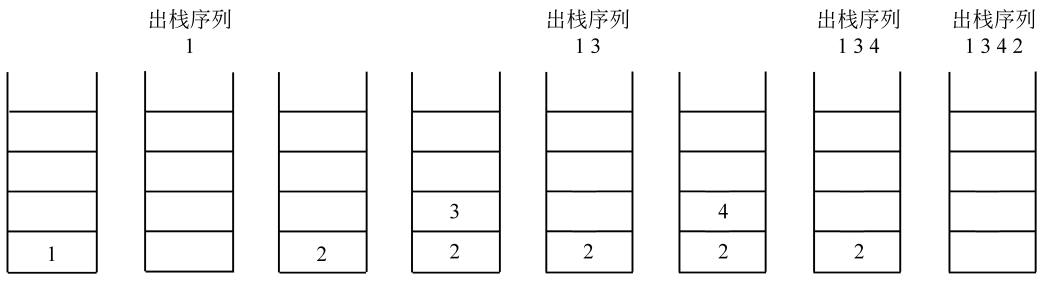
解析：为了得到 1342 的出栈序列，其操作过程是：1 入栈，1 出栈，2 入栈，3 入栈，3 出



（a）元素1、2、3入栈　　（b）元素3出栈

图 3-2　栈操作示意图

栈，4 入栈，4 出栈，2 出栈，如图 3-3 所示。因此，相应的 I 和 O 操作串为 IOIIOIOO。



（a）1入栈　（b）1出栈　（c）2入栈　（d）3入栈　（e）3出栈　（f）4入栈　（g）4出栈　（h）2出栈

图 3-3　栈操作示意图

【例 3.3】　一个栈的入栈序列为 1，2，…，n，出栈序列为 p_1，p_2，…，p_n（p_1，p_2，…，p_n 是 1，2，…，n 的一种排列）。若 $p_1=3$，则 p_2 可能的取值有多少个？

解析：为了让 3 作为第一个出栈元素，将 1、2、3 依次入栈，之后 3 出栈，此时栈如图 3-4 所示。之后可以让 2 出栈，$p_2=2$，也可以让 4 进栈再出栈，$p_2=4$，也可以让 4、5 进栈再出栈，$p_2=5$，…，所以 p_2 可以是 2，4，5，…，n，不可能是 1 和 3，即 p_2 可能的取值有 $n-2$ 个。

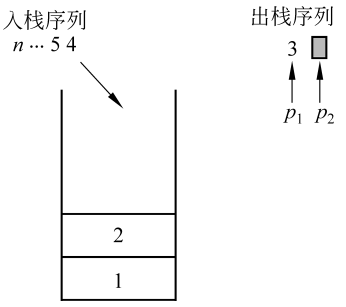


图 3-4　栈操作的一个时刻

在实际应用中，栈的基本操作除了入栈和出栈外，还有栈空的判定、取栈顶元素等。下面给出栈的抽象数据类型的定义：

```
ADT Stack
{
    数据对象：
        D={ai | ai∈ElemSet, i=1,2,…,n, n≥0}
    数据关系：
        R={<ai,ai+1> | ai,ai+1∈D, i=1,2,…,n-1}
        约定 an 端为栈顶，a1 端为栈底
    基本操作：
```

```
        InitStack()
        操作结果：构造一个空栈。
        DestroyStack()
        初始条件：栈已存在。
        操作结果：栈被销毁。
        Length()
        初始条件：栈已存在。
        操作结果：返回栈元素个数。
        Empty()
        初始条件：栈已存在。
        操作结果：若栈为空，则返回 true，否则返回 false。
        Clear()
        初始条件：栈已存在。
        操作结果：清空栈。
        Push(e)
        初始条件：栈已存在。
        操作结果：元素 e 入栈，成为新的栈顶元素。
        Pop(&e)
        初始条件：栈已存在，且非空。
        操作结果：弹出栈顶元素，并用 e 返回栈顶元素的值。
        GetTop(&e)
        初始条件：栈已存在，且非空。
        操作结果：用 e 返回栈顶元素。
}
```

栈中数据元素的逻辑关系呈线性关系，所以栈和线性表一样，也有两种存储表示方法，分别称为顺序栈和链栈。

3.1.2 顺序栈的表示和实现

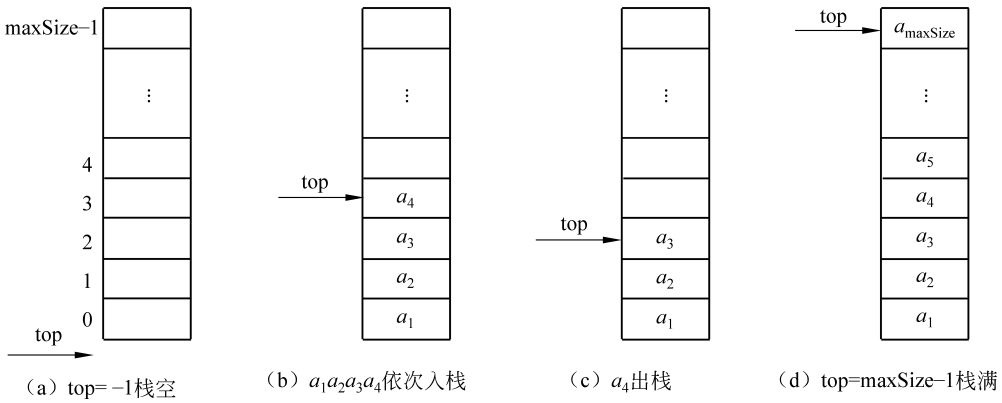
顺序栈是指利用顺序存储结构实现的栈，即利用一组地址连续的存储单元依次存放自栈底到栈顶的数据元素。

同顺序表一样，顺序栈也可以用一维数组实现。通常把数组中下标为 0 的一端作为栈底，同时附设变量 top 指示栈顶元素在数组中的位置。设存储栈的数组长度为 maxSize，则栈空时栈顶位置 top＝－1，栈满时栈顶位置 top＝maxSize－1。入栈时，栈顶位置 top 加 1；出栈时，栈顶位置 top－1。栈操作的示意图如图 3-5 所示。

顺序栈的类模板定义如下所示，其中成员变量实现顺序栈的存储结构，成员函数实现顺序栈的基本操作。

```
template <typename T>
class SqStack
{
private:
    int top;                                    //栈顶元素在数组中的下标
    int maxSize;                                //栈可用的最大容量
```

```
    T * elem;                                   //存储空间的基地址
public:
    SqStack(int size=DEFAULT_SIZE);             //构造函数
    SqStack(const SqStack<T>&st);               //复制构造函数
    ~SqStack();                                 //析构函数
    int Length();                               //返回栈中元素的个数
    bool Empty();                               //判断栈是否为空
    void Clear();                               //清空栈
    bool Push(T e);                             //入栈
    bool Pop( T &e);                            //出栈
    bool GetTop(T &e);                          //返回栈顶元素
};
```



（a）top=−1栈空　（b）$a_1a_2a_3a_4$依次入栈　（c）a_4出栈　（d）top=maxSize−1栈满

图 3-5　栈操作的示意图

下面讨论基本操作的算法及实现。

1. 构造函数——建立一个空栈

构造函数用于完成初始化工作，即为顺序栈动态分配一块连续的存储空间。

【算法实现】

```
template <typename T>
SqStack <T>::SqStack(int size)                  //构造一个最大容量为 size 的空栈
{
    elem=new T[size];                           //分配存储空间
    maxSize=size;                               //设置栈的最大容量
    top=-1;                                     //设置 top 的初值
}
```

2. 复制构造函数

复制构造函数是根据一个已经存在的顺序栈构造一个新的顺序栈。

【算法实现】

```
template <typename T>
SqStack <T>::SqStack(const SqStack<T>&st)
```

```
{
    maxSize=st.maxSize;
    elem=new T[maxSize];
    top=st.top;
    for (int i=0; i<=top; i++)
        elem[i]=st.elem[i];
}
```

3. 析构函数——销毁顺序栈

析构函数用于顺序栈对象离开作用域时释放其占用的存储空间。

【算法实现】

```
template <typename T>
SqStack <T>::~SqStack()
{
    delete []elem;                                    //释放存储空间
}
```

4. 返回栈中的元素个数

在栈的类模板定义中，变量 top 用于指示栈顶元素在数组中的位置，C++ 中，数组的下标是从 0 开始的，所以栈中的元素个数为 top+1。

【算法实现】

```
template <typename T>
int SqStack<T>::Length()
{
    return top+1;
}
```

5. 判断栈是否为空

顺序栈为空时，top=－1。因此，判断顺序栈是否为空，只需判断 top 的值是否为－1。

【算法实现】

```
template <typename T>
bool SqStack<T>::Empty()
{
    return top==-1;
}
```

6. 清空栈

栈空时，栈顶位置 top 的值为－1。因此，清空顺序栈只需将 top 的值设置为－1。

【算法实现】

```
template <typename T>
void SqStack<T>::Clear()
{
    top=-1;
```

```
}
```

7. 入栈

入栈操作是指在栈顶插入一个新的元素。

在顺序栈中插入元素,需要判断栈是否已满。若栈满,则操作失败,返回 false;若栈未满,则将栈顶位置 top 加 1,然后在该位置上填入待插入元素,并返回 true。

【算法实现】

```
template <typename T>
bool SqStack<T>::Push(T e)
{
    if (top==maxSize-1)
        return false;
    elem[++top]=e;
    return true;
}
```

8. 出栈

出栈操作是将栈顶元素删除。

删除栈顶元素,需要判断栈是否为空。若栈空,则操作失败,返回 false;若栈非空,则保存栈顶元素的值,栈顶位置 top 减 1,返回 true。

【算法实现】

```
template <typename T>
bool SqStack<T>::Pop(T &e)
{
    if (top==-1)
        return false;
    e=elem[top--];
    return true;
}
```

9. 返回栈顶元素

此操作是将 top 位置的栈顶元素取出,并不修改栈顶位置。

若栈空,则操作失败,返回 false;若栈非空,则将栈顶元素的值赋值给 e 并返回 true。

【算法实现】

```
template <typename T>
bool SqStack<T>::GetTop(T &e)
{
    if (top==-1)
        return false;
    e=elem[top];
    return true;
}
```

除复制构造函数外,上述关于顺序栈的基本操作的算法时间复杂度均为 $O(1)$。

3.1.3 链栈的表示和实现

顺序栈和顺序表一样，受到最大空间容量的限制，虽然可以在“满员”时重新分配空间扩大容量，但工作量较大，应该尽量避免。因此，在应用程序无法预先估计栈可能达到的最大容量时，还是应该使用链栈。

链栈是指采用链式存储结构实现的栈，通常用单链表表示，如图 3-6 所示。由于栈的插入和删除操作只能在栈顶执行，因此，以单链表的头部作为栈顶最方便，而且没有必要像单链表那样为了运算方便而附加一个头结点。

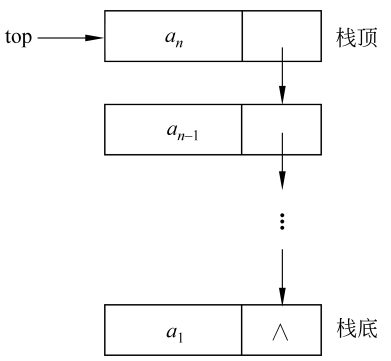


图 3-6　链栈示意图

链栈的结点结构与单链表的结点结构相同，定义如下：

```
template <typename T>
struct StNode
{
    T data;
    StNode<T>*next;
};
```

链栈的类模板定义如下所示，其中成员变量 top 为栈顶指针，成员函数实现链栈的基本操作。

```
template <typename T>
class LinkStack
{
private:
    StNode<T>*top;                          //栈顶指针
public:
    LinkStack();                            //构造函数
    LinkStack(LinkStack<T>&lst);            //复制构造函数
    ~LinkStack();                           //析构函数
    int Length();                           //返回栈中的元素个数
    bool Empty();                           //判断栈是否为空
    void Clear();                           //清空栈
    bool Push(T e);                         //入栈
    bool Pop(T &e);                         //出栈
    bool GetTop(T &e);                      //返回栈顶元素
};
```

下面讨论基本操作的算法及实现。

1. 构造函数——初始化栈

链栈的初始化操作就是构造一个空栈，因为没有必要附设头结点，所以直接将栈顶指针

置空即可。

【算法实现】

```
template <typename T>
LinkStack<T>::LinkStack()
{
    top=NULL;
}
```

本算法的时间复杂度为 $O(1)$。

2. 复制构造函数

复制构造函数是根据一个已经存在的链栈构造一个新的链栈。

【算法实现】

```
LinkStack<T>::LinkStack(LinkStack<T>&lst)
{
    if (lst.Empty())
    {
        top=NULL;
    }
    else
    {
        top=new StNode<T>;
        top->data=lst.top->data;
        StNode<T> * pre=top;
        StNode<T> * p=lst.top->next;
        while (p!=NULL)
        {
            StNode<T> * q=new StNode<T>;
            q->data=p->data;
            pre->next=q;
            pre=q;
            p=p->next;
        }
        pre->next=NULL;
    }
}
```

本算法的时间复杂度为 $O(n)$,其中 n 为链栈中数据结点的个数。

3. 析构函数——销毁链栈

析构函数用于链栈对象离开作用域时释放其占用的全部结点空间。

【算法实现】

```
template <typename T>
LinkStack<T>::～LinkStack()
{
```

```
    StNode<T>* p;
    while (top!=NULL)
    {
        p=top;
        top=p->next;
        delete p;
    }
}
```

本算法的时间复杂度为 $O(n)$，其中 n 为链栈中数据结点的个数。

4. 返回栈中的元素个数

可以采用遍历的方法求链栈中元素的个数，设置一个指针变量 p，初始指向栈顶结点，即 p＝top，依次遍历栈中的结点，直到栈底。

【算法实现】

```
template <typename T>
int LinkStack<T>::Length()
{
    int len=0;
    StNode<T>*p=top;
    while (p!=NULL)
    {
        len++;
        p=p->next;
    }
    return len;
}
```

本算法的时间复杂度为 $O(n)$，其中 n 为链栈中数据结点的个数。

5. 判断栈是否为空

链栈为空时，top 指针为空。因此，判断链栈是否为空，只需判断 top 的值是否为 NULL。

【算法实现】

```
template <typename T>
bool LinkStack<T>::Empty()
{
    return top==NULL;
}
```

本算法的时间复杂度为 $O(1)$。

6. 清空栈

清空栈就是从栈顶开始，依次删除栈中所有的结点，直到栈空为止。

【算法实现】

```
template <typename T>
```

```
void LinkStack<T>::Clear()
{
    StNode<T>* p;
    while (top!=NULL)
    {
        p=top;
        top=p->next;
        delete p;
    }
}
```

本算法的时间复杂度为 $O(n)$，其中 n 为链栈中数据结点的个数。

7. 入栈

和顺序栈的入栈操作不同的是，链栈在入栈前不需要判断栈是否已满，只需要为入栈元素动态分配一个结点空间，之后将新结点插入栈顶，并修改栈顶指针，使其指向该结点，如图 3-7 所示。

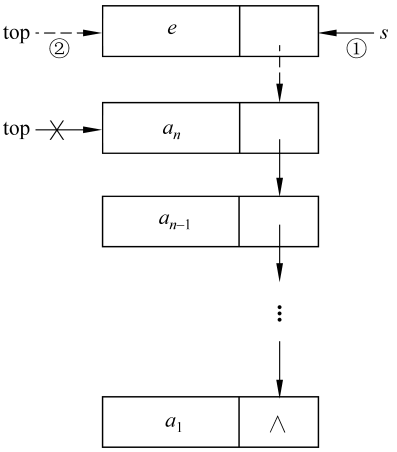


图 3-7　链栈插入操作示意图

【算法实现】

```
template <typename T>
bool LinkStack<T>::Push(T e)
{
    StNode<T>*p=new StNode<T>;                    //生成新结点
    if (p==NULL)                                  //动态内存耗尽
        return false;
    p->data=e;                                    //将新结点的数据域置为 e
    p->next=top;                                  //将新结点插入栈顶
    top=p;                                        //修改栈顶指针
    return true;
}
```

本算法的时间复杂度为 $O(1)$。

8. 出栈

和顺序栈一样，链栈在出栈前也需要判断栈是否为空，不同的是，链栈在出栈后需要释放出栈元素占用的存储空间，如图 3-8 所示。

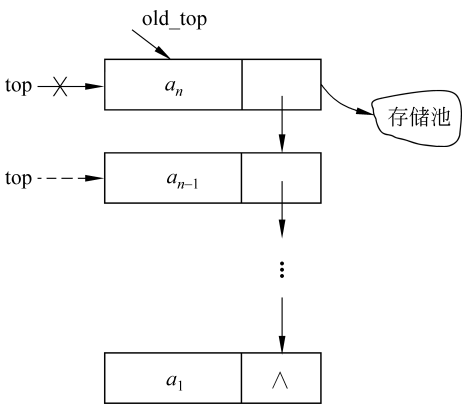


图 3-8　链栈删除操作示意图

【算法实现】

```
template <typename T>
bool LinkStack<T>::Pop(T &e)
{
    if (top==NULL)                          //栈空
        return false;
    StNode<T> * old_top=top;
    e=old_top->data;                        //用 e 保存栈顶元素的值
    top=old_top->next;                      //修改栈顶指针
    delete old_top;                         //释放原栈顶元素的空间
    return true;
}
```

本算法的时间复杂度为 $O(1)$。

9. 返回栈顶元素

和顺序栈一样，当栈非空时，此操作返回当前栈顶元素的值，栈顶指针 top 保持不变。

【算法实现】

```
bool LinkStack<T>::GetTop(T &e)
{
    if (top==NULL)
        return false;
    e=top->data;
    return true;
}
```

本算法的时间复杂度为 $O(1)$。

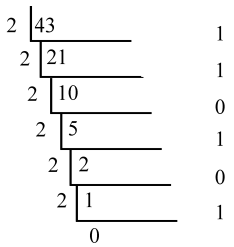
3.1.4 栈的应用

【例 3.4】 数制转换问题。对于输入的任意一个非负十进制整数，输出与其对应的二进制数。

【问题分析】


栈的应用

将十进制数转换为二进制数的方法有很多，其中一个常用的算法是“除 2 取余法”。例如，将十进制数 43 转换为二进制数，运算过程如下：



先取的余数作为低位，后取的余数作为高位，因此$(43)_{10}=(101011)_2$。

上述计算过程是从低位到高位顺序产生二进制数的各个数位，而输出过程应从高位到低位进行，恰好和计算过程相反，因此可以使用栈解决这个问题。在计算过程中依次将得到的余数压入栈中，计算完毕后，将栈中的余数依次出栈输出，输出结果就是转换后的二进制数。

【算法步骤】

(1) 当 $n\neq 0$ 时，重复执行以下操作：

① 计算 n 除以 2 的余数并将其压入栈中。

② 用 $n/2$ 代替 n。

(2) 重复执行以下操作，直到栈为空：弹出栈顶元素，并输出被弹出元素的值。

【程序实现】

```
#include<iostream>
#include"SqStack.h"                          //用顺序栈实现
using namespace std;
void Conversion(int n)
{
    SqStack<int>st1(32);
    int e;
    if (n>0)
    {
        while (n!=0)
        {
            e=n%2;                           //求余数
            st1.Push(e);                     //将余数压入栈中
            n=n/2;
        }
        while (!st1.Empty())
```

```
        {
            st1.Pop(e);                          //弹出栈顶元素,用 e 保存元素的值
            cout<<e;
        }
    }
    else if (n==0)
        cout<<0;
    else
        cout<<"参数错误!";
    cout<<endl;
}
int main()
{
    int n;
    cout<<"输入一个非负十进制整数: ";
    cin>>n;
    Conversion(n);
    return 0;
}
```

【例 3.5】 括号匹配的检测。

C/C++ 语言里,算术表达式中的括号只有小括号。设计算法,判断一个表达式中的括号是否正确配对。

【问题分析】

给定一个算术表达式,目测怎么判断括号是否匹配呢? 可以从左向右看这个表达式中的括号,看到一个“(”就记住它,如果下一个括号是“)”,则划掉这两个括号,如果下一个括号还是“(”,则暂时不管前一个“(”,先把它放在那里,等后面的“(”处理完再处理它,这正好符合栈的后进先出特性,因此可以使用栈解决这个问题。

从左向右依次扫描表达式字符串中的各字符,若读入的是“(”,则进栈;若读入的是“)”,且栈不为空,则出栈,如果栈空,则说明没有与之匹配的左括号,匹配失败。当表达式扫描结束时,若栈为空,则匹配成功;否则,说明没有与栈中的“(”相匹配的“)”,匹配失败。

【算法步骤】

(1) 初始化一个空栈。

(2) 扫描表达式,依次读入字符,直到表达式扫描完毕。

① 如果读入的字符是'(',则将其压入栈。

② 如果读入的字符是')',且栈非空,则弹出栈顶元素;如果栈为空,则括号不匹配,返回 false。

(3) 如果栈为空,则左右括号匹配,返回 true;否则括号不匹配,返回 false。

【程序实现】

```
#include<iostream>
#include<string>
#include"LinkStack.h"                            //用链栈实现
```

```
using namespace std;
bool Match(char * str)                    //表达式存放在字符数组中
{
    LinkStack<char>st1;
    char e;
    for (int i=0; i<strlen(str); i++)  //依次扫描各字符
    {
        if (str[i]=='(')                  //若遇到左括号,则将其入栈
            st1.Push(str[i]);
        else if (str[i]==')')             //遇到右括号,若栈空,则不匹配;若栈不空,则出栈
        {
            if (st1.Empty())
                return false;
            else
                st1.Pop(e);
        }
    }
    if (st1.Empty())                      //扫描完所有字符,若栈空,则匹配,否则不匹配
        return true;
    else
        return false;
}
int main()
{
    char s[50];
    cout<<"请输入表达式: ";
    cin>>s;
    if (Match(s))
        cout<<"括号匹配!"<<endl;
    else
        cout<<"括号不匹配!"<<endl;
    return 0;
}
```

3.2 队　　列

3.2.1 队列的定义

和栈相反,队列是一种先进先出(First In First Out,FIFO)的线性表。它只允许在表的一端进行插入操作,而在另一端进行删除操作。这和日常生活中的排队是一致的,最早进入队列的元素最早离开。在队列中,允许插入的一端称为队尾,允许删除的一端称为队头或队首。向队列中插入新元素称为入队,新元素入队后就成为新的队尾元素;从队列中删除元素称为出队,元素出队后,其直接后继元素就成为队首元素。

设队列 $Q=(a_1,a_2,a_3,\cdots,a_n)$,则称 a_1 为队首元素,称 a_n 为队尾元素。如图 3-9 所示,

队列中的元素是按照 $a_1,a_2,a_3,\cdots,a_n$ 的顺序进入的，退出队列也只能按照这个顺序依次退出，也就是说，只有在 $a_1,a_2,a_3,\cdots,a_{n-1}$ 都离开队列之后，a_n 才能退出队列。

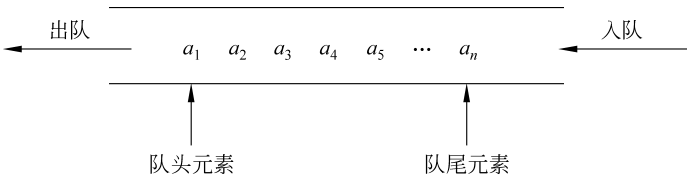


图 3-9　队列的示意图

队列在程序设计中也经常出现。一个最典型的例子就是操作系统中的作业排队。在允许多道程序运行的计算机系统中，同时有几个作业运行。如果运行的结果都需要通过通道输出，那就按请求的先后次序排队。每当通道传输完毕可以接受新的输出任务时，队头的作业先从队列中退出做输出操作。所有申请输出的作业都从队尾进入队列。

队列的操作与栈的操作类似，不同的是，队列的删除操作是在队头进行。下面给出队列的抽象数据类型的定义：

```
ADT Queue
{
    数据对象：
        D={aᵢ | aᵢ∈ElemSet, i=1,2,…,n, n≥0}
    数据关系：
        R={<aᵢ, aᵢ₊₁> | aᵢ, aᵢ₊₁∈D, i=1,2,…,n-1}
        约定 a₁ 端为队头，aₙ 端为队尾
    基本操作：
        InitQueue()
        操作结果：构造一个空队列。
        DestroyQueue ()
        初始条件：队列已存在。
        操作结果：队列被销毁。
        Length()
        初始条件：队列已存在。
        操作结果：返回队列长度。
        Empty()
        初始条件：队列已存在。
        操作结果：若队列为空，则返回 true，否则返回 false。
        Clear()
        初始条件：队列已存在。
        操作结果：清空队列。
        InQueue(e)
        初始条件：队列已存在。
        操作结果：插入元素 e 为新的队尾。
        OutQueue(&e)
        初始条件：队列已存在。
        操作结果：删除队头元素，并用 e 返回其值。
```

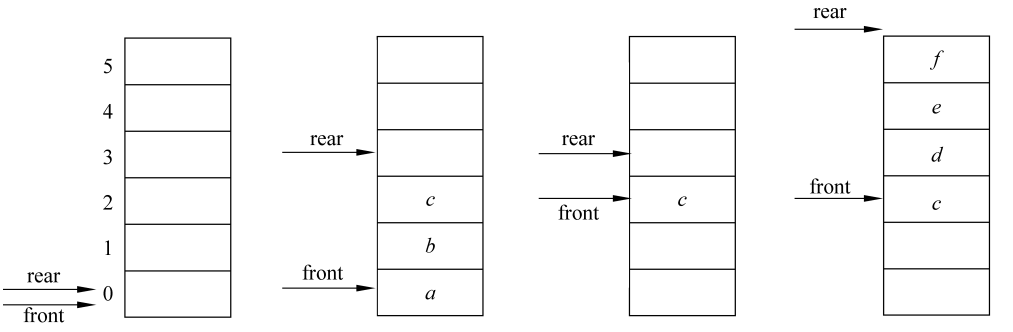
```
        GetHead(&e)
        初始条件：队列已存在。
        操作结果：用 e 返回队头元素。
}
```

3.2.2 循环队列——队列的顺序表示和实现

队列也有两种存储表示：顺序表示和链式表示。

队列的顺序表示就是用一组地址连续的存储单元依次存放从队头到队尾的元素。可以用数组 elem[maxSize]作为队列的顺序存储空间，maxSize 为队列的容量，队头元素放在下标为 0 的一端，则入队操作相当于追加，不需要移动元素。由于队列的队头和队尾都是活动的，因此需要附设两个整型变量 front 和 rear，分别指示队头元素和队尾元素的位置。

为了在 C++ 语言中描述方便，在此约定：初始化创建空队列时，令 front＝rear＝0，每当插入新的队尾元素时，rear 加 1；每当删除队头元素时，front 加 1。因此，在非空队列中，front 始终指向队头元素，而 rear 始终指向队尾元素的下一个位置，如图 3-10 所示。



（a）空队列　（b）a、b 和 c 依次入队　（c）a 和 b 依次出队　（d）d、e 和 f 依次入队

图 3-10　队列操作示意图

假设当前队列分配的最大空间为 6，则当队列处于图 3-10(d)所示的状态时，不可再继续插入新的队尾元素，否则会出现溢出现象，即因数组越界而导致的非法操作错误。事实上，此时队列的实际可用空间并未占满，所以这种现象称为“假溢出”。这是由“队尾入队，队头出队”这种受限制的操作造成的。

解决假溢出的方法是，将队列从逻辑上看成一个环，如图 3-11 所示，通常称之为循环队列。

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，可用取模运算实现。

队头进 1：front＝(front＋1)％maxSize

队尾进 1：rear＝(rear＋1)％maxSize

在图 3-12(a)中，队头元素是 e，在元素 f 入队之前，rear 的值为 5，当元素 f 入队之后，rear 的值为 0。

在图 3-12(b)中，g、h、i、j 相继入队，队列空间均被占

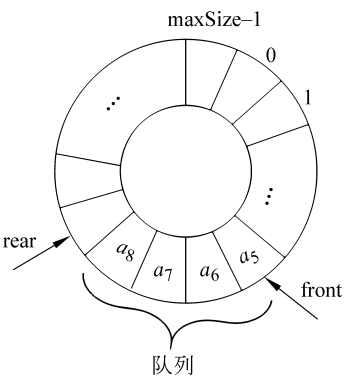
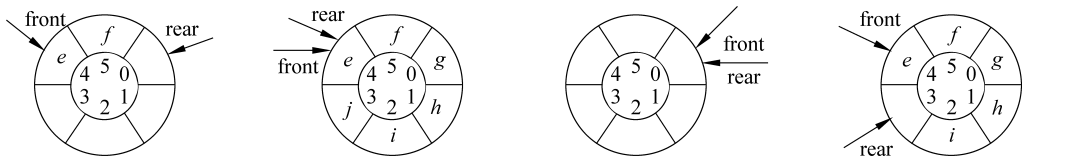


图 3-11　循环队列示意图



（a）一般情况　（b）队列空间被占满　（c）空队　（d）“满”状态的循环队列

图 3-12　循环队列中队头、队尾和元素之间的关系

满，此时 front 和 rear 的值相同。

在图 3-12(c)中，若 e 和 f 从图 3-12(a)所示的队列中出队，则队列呈空的状态，front 和 rear 的值也是相同的。

由此可见，对于循环队列，不能以 front 和 rear 的值是否相等判断队列空间是空，还是满。在这种情况下，如何区分是队满，还是队空呢？

通常有以下两种处理方法。

(1) 少用一个元素空间，即队列空间大小为 n 时，若有 $n-1$ 个元素，就认为是队满。这样，判断队空的条件不变，即当 front 和 rear 的值相同时，则认为队空；而当 rear 在循环意义上加 1 后等于 front，则认为队满。因此，在循环队列中，队空和队满的条件分别是

队空的条件：front==rear

队满的条件：(rear+1)%maxSize==front

如图 3-12(d)所示，当 g、h、i 进入图 3-12(a)所示的队列后，(rear+1)%maxSize 的值等于 front，此时认为队满。

(2) 另设一个标志位，以区别队列是空，还是满。

本书采用第一种方法处理循环队列，下面是循环队列的类模板声明。

```
template <typename T>
class SqQueue
{
pravite:
    int front, rear;                    //指示队头和队尾的位置
    int maxSize;                        //队列的最大容量
    T *elem;                            //存储空间的基地址
public:
    SqQueue(int size=MAXSIZE);          //构造函数
    SqQueue(SqQueue<T>&sq);             //复制构造函数
    ~SqQueue();                         //析构函数
    int Length();                       //返回队列的长度
    bool Empty();                       //判断队列是否为空
    void Clear();                       //清空队列
    bool InQueue(T e);                  //入队
    bool OutQueue(T &e);                //出队
    bool GetHead(T &e);                 //返回队头元素
};
```

下面讨论基本操作的算法及实现。

1. 构造函数

循环队列的初始化操作就是动态分配一块连续的存储空间，并将指示队头和队尾位置的变量 front 和 rear 置为 0，表示队列为空。

【算法实现】

```
SqQueue<T>::SqQueue(int size)          //构造一个空的循环队列
{
    elem=new T[maxSize];               //分配存储空间
    maxSize=size;                      //设置循环队列的最大容量
    rear=front=0;                      //设置 rear 和 front 的初值
}
```

2. 复制构造函数

复制构造函数是根据一个已经存在的循环队列构造一个新的循环队列。

【算法实现】

```
template<typename T>
SqQueue<T>::SqQueue(SqQueue<T>&sq)
{
    maxSize=sq.maxSize;
    elem=new T[maxSize];
    front=sq.front;
    rear=sq.rear;
    for (int i=front; i!=rear; i=(i+1)%maxSize)
        elem[i]=sq.elem[i];
}
```

3. 析构函数

析构函数用于循环队列对象离开作用域时释放其占用的存储空间。

【算法实现】

```
template <typename T>
SqQueue<T>::~SqQueue()
{
    delete []elem;                     //释放存储空间
}
```

4. 返回队列长度

对于非循环队列，rear 和 front 的差值便是队列长度，而对于循环队列，差值可能为负数，所以需要将差值加上 maxSize，然后对 maxSize 取余。

【算法实现】

```
template <typename T>
int SqQueue<T>::Length()
{
    return (rear-front+maxSize)%maxSize;
}
```

5. 判断队列是否为空

循环队列的判空操作只需判断 front 和 rear 的值是否相等。

【算法实现】

```
template <typename T>
bool SqQueue<T>::Empty()
{
    return front==rear;
}
```

6. 清空队列

循环队列为空时，队头和队尾位置均为 0，因此，清空队列只需将 front 和 rear 的值设置为 0。

【算法实现】

```
template <typename T>
void SqQueue<T>::Clear()
{
    rear=front=0;
}
```

7. 入队

入队操作是指在队尾插入一个新的元素。

插入元素前需要判断队列是否已满。若队列已满，则操作失败，返回 false；若队列未满，则将待插入元素写入队尾位置，rear 加 1，并返回 true。

【算法实现】

```
template <typename T>
bool SqQueue<T>::InQueue(T e)
{
    if ((rear+1)%maxSize==front)
        return false;
    elem[rear]=e;
    rear=(rear+1)%maxSize;
    return true;
}
```

8. 出队

出队操作是就将队头元素删除。

删除元素前需要判断队列是否为空。若队列为空，则操作失败，返回 false；若队列不为空，则保存队头元素的值，队头位置 front 加 1，返回 true。

【算法实现】

```
template <typename T>
bool SqQueue<T>::OutQueue(T &e)
{
```

```
    if (rear==front)
        return false;
    e=elem[front];
    front=(front+1)%maxSize;
    return true;
}
```

9. 返回队头元素

此操作是将 front 位置的队头元素取出，并不修改队头位置。

若队列为空，则操作失败，返回 false；若队列不为空，则将队头元素的值赋值给 e 并返回 true。

【算法实现】

```
template <typename T>
bool SqQueue<T>::GetHead(T &e)
{
    if (rear==front)
        return false;
    e=elem[front];
    return true;
}
```

上述关于循环队列的基本操作，除复制构造函数外，算法时间复杂度均为 $O(1)$。

3.2.3 链队——队列的链式表示和实现

队列的链式存储结构称为链队，通常用单链表表示，其结点结构与单链表的结点结构相同，定义如下：

```
template <typename T>
struct QuNode
{
    T data;
    QuNode<T> * next;
}
```

为了使空队列和非空队列的操作一致，链队也增设一个头结点。由于队列只允许在队头进行删除操作，在队尾进行插入操作，为操作方便，设置两个指针分别指向队头和队尾，队头指针 front 指向头结点，队尾指针 rear 指向终端结点，如图 3-13 所示。

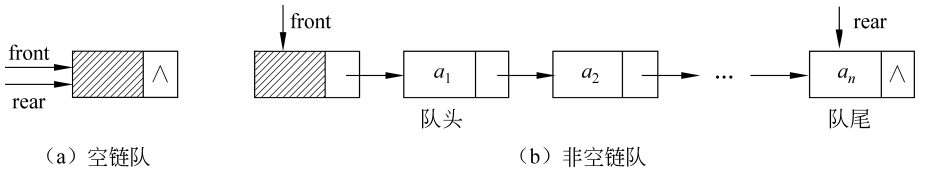


图 3-13 链队示意图