第3章 堆栈和队列

本章介绍计算机程序设计中应用非常广泛的数据结构：堆栈和队列。逻辑上讲，堆栈和队列应属于线性表的范畴，只是与线性表相比，它们的运算受到了严格的限制（故也称为限定性线性表）。之所以将它们单独讨论，是由于它们在程序设计中具有重要性。

3.1 堆栈的定义

3.1.1 堆栈的逻辑结构

堆栈（简称栈）是一个线性表，其数据元素只能从这个有序集合的同一端插入或删除，这一端称为堆栈的栈顶（top），而另一端称为堆栈的栈底（bottom）。

用第 2 章学到的知识来理解，也可以说，堆栈是限定只能在表头（或表尾）进行插入和删除运算的线性表。表头（或表尾）是开放运算的栈顶，另一端是封闭运算的栈底。

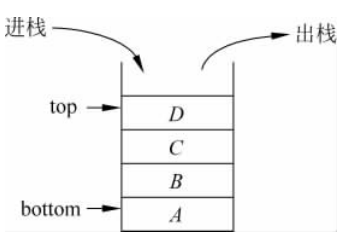

图 3.1 堆栈

现在用实例来说明这个定义的含义。如图 3.1 所示，它是一个栈，依次往栈中压入 4 个元素 A、B、C、D。则 A 在栈的最底下，B 在 A 的上面，C 在 B 的上面，D 又压在 C 的上面。若要访问这 4 个元素中除 D 外的元素只能先取出 D，才能取到 C，只有取出 C 之后，才能取出 B，最后才能取出 A。取出这 4 个元素的顺序是 D、C、B、A，与放入时的顺序恰好相反，故也称栈为后进先出表或先进后出表，简称 LIFO（Last In First Out）或 FILO（First In Last Out）表。

3.1.2 堆栈的抽象数据类型

堆栈的抽象数据类型如下：

```
ADT Stack
{
Data: 是一个只能从同一端插入或删除限定性的线性表(e0,e1,…,en-1)。
Relation: 堆栈的一端(en-1)称为栈顶(top),而另一端(e0)称为栈底(bottom)。
Operation:
CreatStack(MaxStackSize)          //构造大小为 MaxStackSize 的空堆栈
IsEmpty()                         //判断堆栈是否为空,如果为空返回 true,否则返回 false
```

```
IsFull()                    //判断堆栈是否为满,如果为满返回 true,否则返回 false
GetTop(result)              //返回栈顶元素
Push(newvalue)              //向堆栈中压入元素值 newvalue
Pop(result)                 //从堆栈中弹出元素
}
```

压入运算也称进栈(或入栈)操作。它是将数据元素插入到堆栈的栈顶,相当于线性表的插入运算(InsertElement),即在线性表的末端插入一个元素。

弹出运算也称为出栈操作。它是将堆栈的栈顶元素取出,相当于线性表的删除运算(DeleteElement),即删除线性表的最后一个元素。

判栈空运算的作用是判断堆栈是否为空。为空时,若再执行弹出运算操作就是一个错误,发生这种情况称为下溢(underflow)。

判栈满运算的作用是判断堆栈是否已满(即预留的空间已被元素充满),若满,再执行压入堆栈运算就是一个错误,称为上溢(overflow)。

返回栈顶数据元素,即取出堆栈的栈顶上的元素值,但不删除栈顶上的元素,相当于查找线性表的最后一个数据元素。

如同线性表一样,堆栈也有顺序存储和链式存储两种存储方式。在不同的存储方式下,上述运算的执行过程是不相同的,下面就两种不同的存储方式讨论堆栈运算的实现。

3.2　堆栈的顺序存储及操作

3.2.1　堆栈顺序存储

1. 堆栈顺序存储概念

堆栈顺序存储方式,是将堆栈中的数据元素依次地存放于存储器相邻的单元,来保证堆栈数据元素逻辑上的有序性。

堆栈结构中,element 是堆栈数据存放空间,其中的第一个存储单元就是堆栈中栈底元素(e_0)存放的位置。top(top=$n-1$)指向堆栈中所有进栈元素的栈顶元素,即 top 是栈顶元素的地址。MaxSpaceSize 记录堆栈可以存储元素的最大空间值。

由于定义堆栈空间时采用的是数组,所以一般约定下标为 0 的元素空间就是栈底,这样就不用另设一个变量再来记录栈底指针 bottom。

假设堆栈中每个数据元素占用 size 字节空间,由图 3.2 可见,仍然可以像线性表中一样利用公式

$$\text{location}(e_i) = \text{location}(e_0) + i \times \text{size}$$

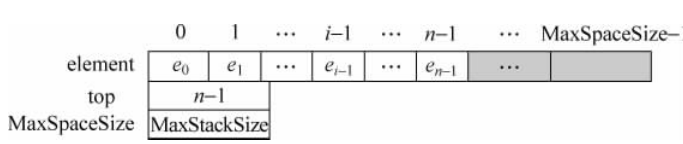

图 3.2　堆栈的顺序存储结构

求取堆栈中元素 e_i 的地址。但是,由于堆栈是一个受限的线性表,所以,一般情况下不做取中间数据元素的运算。

2. 顺序存储结构堆栈类定义

顺序堆栈类定义如下:

```
template<class StackType>
class Stack
{   //顺序存储结构堆栈模板类 Stack 的定义
   public:
    Stack(int MaxStackSize = 20);              //构造函数
    ~Stack() {delete[] element;}               //析构函数(释放空间)
    int GetTopAddress(){return top;};          //获取堆栈栈顶指针
    bool IsEmpty() {return top == -1;}         //判断堆栈空
    bool IsFull() {return top >= MaxSpaceSize;} //判断堆栈满
    bool GetTop(StackType& result) ;           //获取栈顶元素值,存放到 result
    bool Push(StackType& newvalue);            //进栈: newvalue 值进栈
    bool Pop(StackType& result);               //出栈: 出栈值存放到 result
   private:                                    //堆栈数据结构定义
    int top;                                   //堆栈栈顶指针
    int MaxSpaceSize;                          //堆栈空间大小
    StackType *element;                        //堆栈数据元素存放空间
};
```

Stack 模板类是一个顺序存储的堆栈,其中 element 是一个一维数组(静态存储空间),每个数组元素空间用于存放堆栈元素数据值,top 指向堆栈栈顶元素,MaxSpaceSize 记载堆栈可存储的最多数据元素个数。

3.2.2 顺序存储结构堆栈的运算实现

下面讨论顺序存储结构下堆栈的几种主要运算。

1. 构造空堆栈

构造空堆栈是 Stack 模板类的构造函数。所谓空堆栈是指堆栈中没有一个数据元素,但创建了数据元素的空间和堆栈结构,如图 3.3 所示。

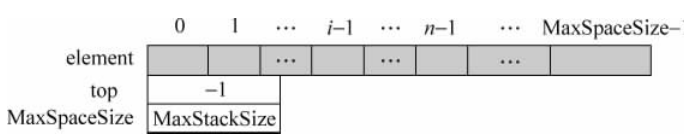

图 3.3 顺序存储堆栈的空栈

空堆栈产生后,就存在一个 ElementType 类型的数组,大小是 MaxSpaceSize,表中只有存放数据元素的空间,堆栈栈顶指针设为−1(用户约定),即堆栈为空时,栈顶指针指向栈底空间的前面。不难分析,算法的时间复杂性是 $O(1)$。

构造空堆栈算法(Stack 类构造函数)如下:

```
template<class StackType>
```

```
Stack < StackType >::
Stack(int MaxStackSize)
{   ///构造函数,堆栈数据元素存放于 element[0..MaxSpaceSize - 1]
    MaxSpaceSize = MaxStackSize;
    element = new StackType[MaxSpaceSize];
    top = -1;
}
```

2. 判断堆栈是否为空

所谓堆栈为空,是指堆栈中没有一个数据元素,即栈顶指针 top 指向数据空间的第一个位置(0 下标的空间)的前面(top 的值为－1)。如堆栈不空,top 总是指向栈顶元素(top 的值为非－1)。

判断堆栈是否为空算法 IsEmpty 如下:

```
bool IsEmpty()
{   //判断堆栈是否为空
    if (top == -1) return true;
    return false;
}
```

3. 判断堆栈是否为满

所谓堆栈为满,是指堆栈的数据空间已经全部用完,即栈顶指针 top 指向数据空间的最大下标位置(MaxSpaceSize－1 下标的空间)。如堆栈不满,top 总是指向栈顶元素(top 的值小于 MaxSpaceSize－1)。

判断堆栈是否为满算法 IsFull 如下:

```
bool IsFull()
{   //判断堆栈是否为满
    if (top >= MaxSpaceSize - 1) return true;
    return false;
}
```

4. 返回栈顶元素的值

返回堆栈栈顶元素的值,是指将 top 所指的堆栈元素的值取出,但是 top 指针不移动,当然,能够取得栈顶值的前提是栈中有元素存在。

返回栈顶元素值算法 GetTop 如下:

```
template < class StackType >
bool Stack < StackType >::
GetTop(StackType& result)
{   //获取栈顶元素值
    if (IsEmpty()) return false;
    result = element[top];
    return true;
}
```

5. 出栈运算

出栈(又称弹出)运算是将栈顶元素取出,且将栈顶指针 top 向下移动一个位置,即取出 top 所指的元素,然后,top 的值减 1。出栈时,首先要判断堆栈中是否存在元素可取,即先判断栈是否为空,不为空时可以出栈,否则出错。出栈后,top 指针要做相应的移动。注意,这个算法与取栈顶元素值的算法 GetTop 有所不同,两个算法都可以取得栈顶指针所指的元素值,但 GetTop 算法取值后不会移动 top 指针,即栈中元素的个数不发生改变。

出栈(弹出)算法 Pop 如下:

```
template<class StackType>
bool Stack<StackType>::
Pop(StackType& result)
{   //出栈
    if (IsEmpty()) return false;
        result = element[top--];
    return true;
}
```

6. 进栈运算

进栈(又称压入)运算是将一个新元素存储到当前 top 所指的空间的上一个位置,即 top+1 的元素空间中。进栈时,首先要判断堆栈中是否存在元素存放的空间,即先判断栈是否为满,不为满时,newvalue 可以进栈,否则出错。newvalue 进栈前,top 指针要先做相应的移动。

进栈(压入)算法 Push 如下:

```
template<class StackType>
bool Stack<StackType>::
Push(StackType& newvalue)
{   //进栈
    if (IsFull()) return false;
        element[++top] = newvalue;
    return true;
}
```

上面讨论的是堆栈的相关操作,可以看到,堆栈的顺序存储与线性表的结构基本上是一样的,而其操作比线性表的操作更简单。

3.3 堆栈的链式存储及操作

3.3.1 堆栈的链式存储

堆栈的链式存储结构的特点是用物理上不一定相邻的存储单元来存储堆栈的元素,为了保证堆栈元素之间逻辑上的连续性,存储元素时,除了存储它本身的数据内容以外,还附

加一个指针域(也叫链域)来指出相邻元素的存储地址。

在C++语言中,首先定义动态存储空间分配方式下堆栈的数据元素的类型:

```
template < class ElementType >
class ChainNode
{
public:
    ElementType              data;
    ChainNode < ElementType > * link;
};
```

在动态链式结构中,栈结点由两部分构成的,一是数据元素的数据域,二是数据元素的链接域,如图 3.4 所示。链接域指向更靠近栈底的相邻的栈元素存储空间的起始地址,链式栈中的第一个结点就是栈顶元素,最后一个结点就是栈底元素,最后一个结点的链接域的值为空。

图 3.5 给出了一个链式栈的结构,表头链接域 top 始终指向栈顶结点,即链表的第一个结点。对链式栈,一般不存在栈满问题,除非存储空间全部被耗尽;链式栈空表现为链式栈的 top 的值为空,即链表为空。

图 3.4 堆栈数据元素结点结构

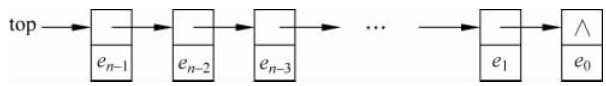

图 3.5 链式栈

3.3.2 链式栈类的定义

链式栈类的定义如下:

```
template < class ElementType >
class ChainStack
{
   public:
    ChainStack() {top = NULL;}                  //构造函数
    ~ChainStack();                              //析构函数
    bool IsEmpty() {return top == NULL;}        //判断堆栈空
    bool GetTop(ElementType& result);           //获取堆栈栈顶元素值
    bool Push( ElementType& newvalue);          //进栈
    bool Pop(ElementType& result);              //出栈
   private:
    ChainNode < ElementType > * top;            //top 指向栈顶结点
};
```

3.3.3 链式栈类运算的实现

1. 析构运算

所谓析构运算,是指删除堆栈中的所有结点,释放所有空间。

析构运算算法~ChainStack 如下：

```
template<class ElementType>
ChainStack<ElementType>::
~ChainStack()
{   //析构函数
   ChainNode<ElementType> *nextPtr;
   while (top)
   {
      nextPtr = top->link;
      delete top;
      top = nextPtr;
   }
}
```

2. 返回链式栈栈顶元素的值

返回链式栈栈顶元素的值，是指将 top 所指的堆栈元素的值取出，但是 top 指针不改变，当然，能够取得栈顶值的前提是栈中有元素存在。

返回链式栈栈顶元素值算法 GetTop 如下：

```
template<class ElementType>
bool ChainStack<ElementType>::
GetTop(ElementType &result)
{   //获取堆栈栈顶元素值
    if (IsEmpty()) return false;
    result = top->data;
    return true;
}
```

3. 链式栈出栈运算

链式栈的出栈运算是将 top 指针所指的结点值取出，且将栈顶指针 top 指向第一个结点的直接后继结点，并将原来的第一个结点空间释放。出栈时，首先要判断堆栈中是否存在元素结点可取，即先判断栈是否为空，不为空时可以出栈，否则出错。注意，这个算法与取栈顶元素值的算法 GetTop 有所不同，两个算法都可以取得栈顶指针所指的元素值，但 GetTop 算法取值后不会改变 top 指针，即栈中元素的个数不发生改变。

链式栈出栈算法 Pop 如下：

```
template<class ElementType>
bool ChainStack<ElementType>::
Pop(ElementType& result)
{   //出栈元素到 result
    if (IsEmpty()) return false;
    result = top->data;
    ChainNode<ElementType> *p = top;
    top = top->link;
    delete p;
```

```
    return true;
}
```

4. 链式栈进栈运算

进栈运算是将一个新元素 newvalue 的结点压入到链式栈中，作为链表的第一个结点。newvalue 结点进栈后，top 指针则指向它。

链式栈进栈算法 Push 如下：

```
template<class ElementType>
bool ChainStack<ElementType>::
Push( ElementType& newvalue)
{   //newvalue 元素进栈
    ChainNode<ElementType> *p = new ChainNode<ElementType>;
    p->data = newvalue;
    p->link = top;
    top = p;
    return true;
}
```

3.4 多个栈共享邻接空间

以上讨论了单个栈顺序存储结构和动态链式存储结构方式下的实现和运算，它能够有效地控制后进先出的顺序数据处理。但是，仔细分析一下就可以发现，在顺序存储结构方式下，栈内的元素的多少往往受到堆栈 MaxSpaceSize 的限制。单个堆栈或因进栈元素过多造成上溢，或因进栈元素太少造成栈多数情况下不满，剩余空间又得不到充分利用。

实际中，有时候需要同时建立多个栈，则可以将这多个栈巧妙地安排在一起，让多个栈共享同一块存储空间，这样可以节省空间并高效地使用这些存储空间。

现在以两个栈 S1、S2 共享一块连续存储空间为例来说明这个问题。如图 3.6 所示，在同时建立两个栈的情况下，可以在内存开辟一块连续存储空间（一个数组），存储空间的两头（数组的两端）分别为两个栈的栈底（bottom）。设空间的范围为（0..MaxSpaceSize－1），则一个栈 S1 从下标 0（S1 栈栈顶）的数组元素空间向 MaxSpaceSize－1 下标方向延伸，而另一个栈 S2 从下标 MaxSpaceSize－1（S2 栈栈顶）的数组元素空间向 0 下标方向延伸。

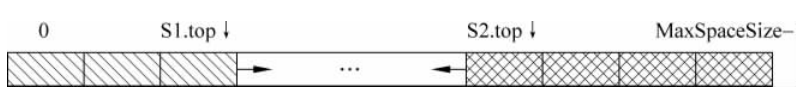

图 3.6 两栈共享空间

除非空间的总容量耗尽，否则两个栈都不会出现栈满的情况。这样，两个栈都可以独立地向中间区域延伸，也就是说，两个栈的大小不是固定不变，而是可以伸缩的，中间区域可以成为两个栈的共享存储区，互不影响，直到两个栈的栈顶相邻时，才出现栈满。

这种处理思想就如同两栈中间有一个“活塞”一样，而这个“活塞”的厚薄表示两栈还可利用的存储空间大小。

在实际中,若要求使用两个栈,而每个栈的各自容量都不能确定,但两个栈的容量之和是一定时,采用这种方法最有效,它能充分利用存储空间。

对于如图 3.6 所示的这样两个栈共享存储空间的情况,栈的运算基本与一个栈的运算方法是一样的,只需要注意 S2 栈顶的伸缩方向与前面讨论的栈的定义的方向相反。所以,S1 栈的栈空仍为 S1.top=-1,S2 栈的栈空应为 S2.top=MaxSpaceSize。而且 S2 进栈和出栈操作时,栈顶指针 S2.top 的修改正好反过来。元素进栈前 S2.top--,元素出栈后 S2.top++。

另外,在两个栈共享相邻区间的情况下,栈满的条件应是两个栈栈顶相碰。两个栈栈满的条件应定义为

S1.top=S2.top-1 或 S1.top1+1=S2.top2

需要说明的是,只有两个栈的元素类型相同,才能采用两个栈共享相邻的存储空间的办法。关于两栈共享空间的相关算法,本书不再详细讨论,留给读者自己完成。

3.5 堆栈的应用

3.5.1 检验表达式中括号的匹配

在将高级编译语言翻译成低级语言的过程中,编译程序或解释程序所做的第一件工作就是语法正确性检验,其中检验的主要内容之一就是程序中左右括号是否的匹配。如 C++语言中使用"{"和"}"作为一组语句的左右匹配符,表达式中"("和")"作为表达式运算时优先级的限定符等。下面就以表达式为例,来实现表达式中括号匹配的正确性检验算法。

为了问题的简化,假设表达式中只出现两种括号:"{"和"}","("和")"。被检验的表达式被看成一个字符串数组,如 expstr[]={(a+b*(c-d)-(x+y)+x*a)}。

在匹配性检验时,从左至右取出字符串中的每个字符。

(1) 如果取出的字符是一个左括号,就将它压入堆栈。

(2) 如果取出的字符不是括号字符,就再取表达式字符串中的下一个字符。

(3) 如果取出的字符是一个右括号,就从堆栈中退出一个左括号,并比较这两个括号:

- 如果取出的右括号是退出的栈顶左括号所对应的右括号,说明这两个括号匹配成功,继续取下一个字符。
- 如果取出的右括号不是退出的栈顶左括号所对应的右括号,说明这两个括号匹配失败,结束匹配过程,并报告出错。

将所有字符全部从字符串中取出(表达式的最后一个字符在实际中是回车符,这里约定为"#"字符)。

如果全部正确匹配,堆栈中应该是空状态,如果堆栈非空,则说明还有左括号未匹配,这时也应该报告出错。

如下面的例子:"{()()}{()#",最后,堆栈中还有一个括号"{"。

如果取出右括号后,堆栈中无出栈左括号与之比较,即堆栈已为空状态时,这时也应该报告出错。

如下面的例子:"{()()}}()#",在匹配过程中,当取出的是第二个"}"时,此时,堆栈已

没有可以出栈的左括号，说明这个“}”无匹配的字符。

下面给出括号的匹配算法 Matching：

```
#define BRACKET StackType
class BRACKET
{
    public:
    char bracket;
};

void Matching (char exp[])
{
    int                MaxStackSize = 30;
    Stack < StackType > S(MaxStackSize);
    StackType          temp;
    char               ch;
    ch = * exp++;
    temp.bracket = ch;
    while (ch != '#')
    {
        temp.bracket = ch;
        switch (ch)
        {
            case  '(':
                {S.Push(temp); break;}
            case  '[':
                {S.Push(temp); break;}
            case  ')':
            {
                if (!S.IsEmpty())
                {
                    S.Pop(temp);
                    ch = temp.bracket;
                    if (ch!= '(')
                        cout <<"ERROR11: 右边是),左边不是("<< endl;
                }
                else
                    cout <<"ERROR12: 右边是),左边没有任何括号!"<< endl;
                break;
            }
            case  '}':
            {
                if (!S.IsEmpty())
                {
                    S.Pop(temp);
                    ch = temp.bracket;
                    if (ch != '{')
                        cout <<"ERROR21: 右边是},左边不是{"<< endl;
                }
                else
```

```
                    cout<<"ERROR22: 右边是},左边没有任何括号!"<<endl;
                break;
            }
        }
        ch = *exp++;
    }
    if (!S.IsEmpty()) cout<<"ERROR3: 左边有括号,但右边没有括号了!"<<endl;
}
```

3.5.2 表达式的求值

在将高级编译语言翻译成低级语言的过程中,编译程序或解释程序要做的另一件工作就是表达式的转换。所谓表达式的转换,就是将在高级语言源程序中的表达式转换为另一种书写顺序,以后运算时按转换后的表达计算结果。

在高级语言中,表达式的书写格式只是其最初的表现形式,实际上,表达式的书写格式有多种方式。任何一个表达式都是由操作数(运算对象)、操作符(运算符)、界定符(括号或结束符)组成,其中,操作数和操作符是主要的两个部分,这两个部分可以有 3 种书写顺序:

中缀表达式:〖操作数〗〖操作符〗〖操作数〗。

前缀表达式:〖操作符〗〖操作数〗〖操作数〗。

后缀表达式:〖操作数〗〖操作数〗〖操作符〗。

例如代数式 X÷Y−(A+B×C)÷D 的表达式有以下 3 种书写顺序:

中缀表达式:X / Y − (A + B * C) / D。

前缀表达式:− / XY / + A * BCD。

后缀表达式:XY / ABC * + D / −。

这 3 种表达式的书写顺序不同,平常使用的就是中缀表达式。在中缀表达式中,有时为了限定运算次序,需要使用括号来实现,但是,在前缀表达式和后缀表达式中就不需要括号,而是以操作符和运算符的排列顺序来表示运算的顺序。

前缀表达式的运算规则是:连续出现的两个操作数与在它们前面且紧靠它们的运算符构成一个最基本的运算步骤。

后缀表达式的运算规则是:每个运算符与在它之前出现且紧靠它的两个操作数构成一个最基本的运算步骤。后缀表达式中,运算符出现的顺序就是表达式的运算顺序。

下面假设表达式已经被书写成后缀表达式形式,并将表达式存储在一个字符串中,然后从左至右地扫描这个字符串,并利用堆栈,最终求出表达式的结果。

表达式运算的算法思想:从左至右依次从字符串中取出每个字符。若取出的字符为字母,则是一个操作数,将其压入堆栈;否则是一个操作符,从堆栈中退出一个值(第一个操作数),再从堆栈中退出一个值(第二个操作数),将两个操作数与运算符做相应的运算。

假设 Operate(f1,op,f2)返回 f1 和 f2 进行 op 运算的结果,算法中,假设只有加、减、乘、除四种运算符。表达式的求值算法 Evalution 如下:

```
#define DIGITAL StackType
class DIGITAL
{
```

```
  public:
    float value;
};

StackType Operate(StackType f1,char ch,StackType f2)
{
    StackType result;
    switch(ch)
    {
        case '+':
        {
            result.value = f2.value + f1.value;
            break;
        }
        case '-':
        {
            result.value = f2.value - f1.value;
            break;
        }
        case '*':
        {
            result.value = f2.value * f1.value;
            break;
        }
        case '/':
        {
            result.value = f2.value/f1.value;
            break;
        }
    }
    return result;                              //返回计算结果
}
void Evalution(char *suffixexp, StackType &result)
{   //计算后缀表达式的值
    char     ch ;
    StackType x;
    StackType f1,f2;                            //两个进栈操作数定义

    int MaxStackSize = 10;
    Stack<StackType> S(MaxStackSize);           //创建 S 堆栈

    ch = *suffixexp++;                  //获取后缀表达式的第一个字符,并准备好获取下一个地址
    while (ch != '#')
    {
        if (!(ch=='+'||ch=='-'||ch=='*'||ch=='/'))
        {
            x.value = (int)ch - 48;             //获取的操作数转换为数值型数据
            S.Push(x);
        }
        else
        {
```

```
            S.Pop(f1);                          //第一个操作数出栈
            S.Pop(f2);                          //第二个操作数出栈
            x = Operate(f1,ch,f2);              //完成两个操作数的 ch 运算,并返回结构到 x 中
            S.Push(x);                          //表达式运算结果重新进栈
        }
        ch = * suffixexp++;
    }
    S.Pop(result);
}
```

3.5.3 背包问题求解

假设有一个能容纳货物总体积为 TotalVolume 的背包,另有 n 个体积分别是 w_1,w_2,…,w_n 的货物,现在要在 n 件货物中选出若干件货物恰好装满背包,求出满足要求的所有解。

实现背包问题的思想是利用尝试回逆法。首先将所有的货物从 0 到 $n-1$ 编号,每个货物的信息包括编号(number)、名称(name)、存放位置(place)、体积(weight),货物存储的信息存放在线性表的 element[]空间中,以后货物就用货物编号来表示。另外,算法实现时使用一个堆栈 S。

算法的思想如下:

从 0 号货物开始顺序地选取货物,如果可以装入背包(装入后不满),则将该货物的编号进栈(堆栈的数据是暂时确定装入背包的货物编号)。

如果当前选取的 k 编号的货物装不进去(如果装入,则总体积大于 TotalVolume),则选取下一个货物($k+1$ 编号的货物),尝试装入背包。

如果尚未求得解,又已无货物可选,则说明上一个装入的货物不合适,就将堆栈退出一个货物编号,再选取这个退出编号的下一个编号的货物尝试。

每求得一组解,就输出堆栈中的所有物品编号(输出不出栈,只是遍历所有堆栈中的数据),然后,退出栈顶数据,再选取当前退出编号的下一个编号的货物尝试,直到堆栈为空(无出栈数据),且达到最大编号的货物。

假设线性表中货物的体积存放在 w 数组成员中,背包的体积 $T=10$,下面讨论算法过程中各种值的变化状态,如图 3.7 所示。

背包问题求解算法由主算法 Knapsack 和输出一种组合结果算法 TraverseStack 构成。完成求解的算法是 Knapsack。每一种组合解的数据都存放在堆栈 S 中,每求得一个解,就调用 TraverseStack 算法输出所求的组合。

TraverseStack 算法输出时,先将组合数据逐个地从 S 堆栈中出栈,输出对应的货物信息,再将出栈的值进栈保存到 Skeep 堆栈中,出栈输出 S 堆栈的所有组合数据后,再从 Skeep 堆栈恢复数据到 S 堆栈中。完成一个组合方案的求解过程。

背包问题求解算法 Knapsack 如下:

```
#define PRODUCTS ElementType
class PRODUCTS
{   //线性表数据元素的结构,货物信息存放于线性表的 element[]空间
public:
    char number[10];                            //货物编号
```

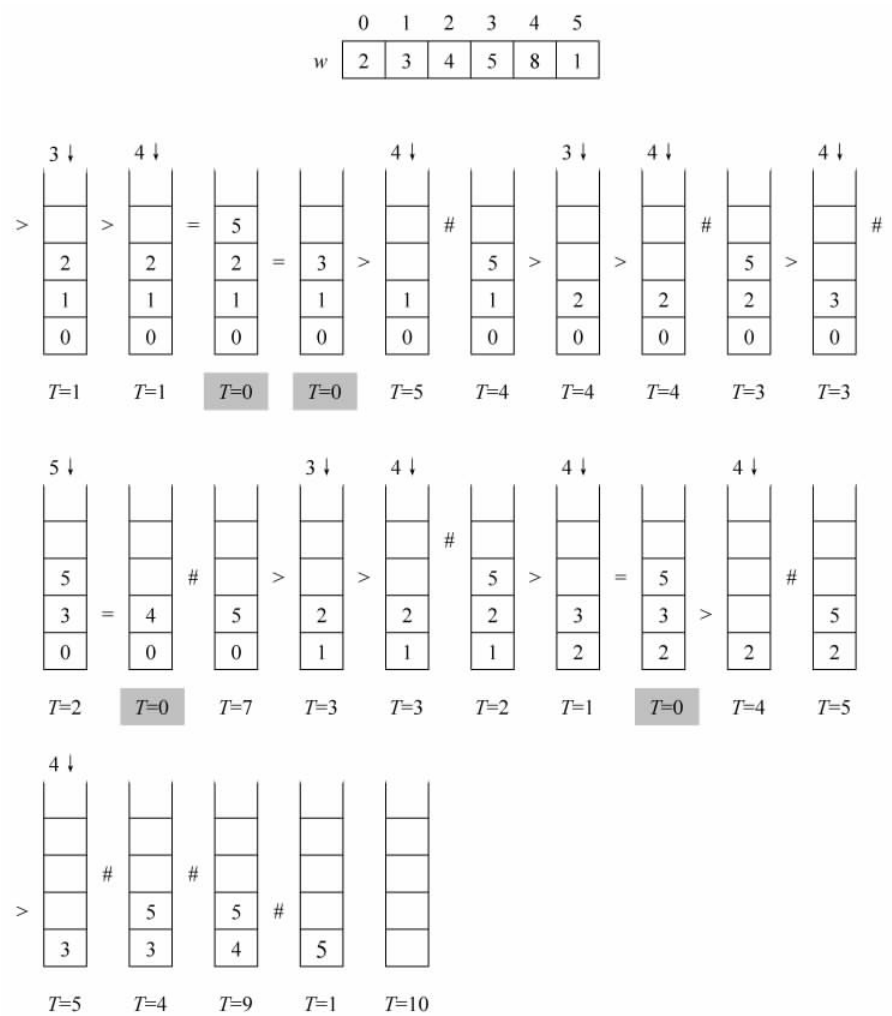

图 3.7　背包求解过程中栈及 T 的变化

```
    char name[10];                          //货物名称
    char place[20];                         //存放位置
    int  weight;                            //货物体积
};
class StackType
{
  public:
    int index;                              //货物编号
};
void TraverseStack(Stack<StackType>&S,ElementType element[])
{   //逐个地输出堆栈中的数据元素(一个货物组合方案).堆栈中的数据只输出,不改变
    //代入的堆栈 S 一定要以引用方式传地址,此输出算法会出栈输出
    //再进栈保存,用于寻找下一个组合。货物信息只输出,不改变
    int       k;
    int       topkeep;
    StackType temp;
```

```
    int       MaxStackSize = 50;
    Stack < StackType > Skeep(MaxStackSize);
//Skeep 堆栈用于暂存 S 堆栈的数据输出 S 堆栈的一个组合方案
//Skeep 堆栈用于恢复 S 堆栈的数据
    cout <<"          -------------- 运送方案货物信息 ------------- "<< endl;
    cout <<"           货物编号        货物名称     存放位置       货物体积"<< endl;
    topkeep = S.GetTopAddress();      //保存堆栈栈顶指针值,其值是堆栈数据元素个数减 1
    while(!S.IsEmpty())
    {
        S.Pop(temp);                  //出栈是为了输出一种货物组合方案,输出后还要还原堆栈
        k = temp.index;                       //出栈值是货物编号 k
        cout <<"              "               //输出货物编号 k 对应的货物信息
            << element[k].number <<"         "
            << element[k].name <<"      "
            << element[k].place <<"          "
            << element[k].weight << endl;
        Skeep.Push(temp);
        //输出后将堆栈出栈值(货物编号 k)先暂存于另外一个堆栈 Skeep 中
    }
    cout <<"          该方案有以上"<< topkeep + 1 <<"件货物"<< endl << endl;
    while(!Skeep.IsEmpty())
    {   //一种方案组合输出后,用暂存堆栈 Skeep 中的数据恢复原来堆栈 S 的数据
        Skeep.Pop(temp);
        S.Push(temp);
    }
}
void Knapsack(ElementType element[], int n,int TotalVolume)
{//货物信息存储在线性表 element[]中
 //背包体积为 TotalVolume,n 个物品的体积存储在 element[]中,求解装满背包的所有解
    int       MaxStackSize = 50;
    StackType temp;
    Stack < StackType > S(MaxStackSize);       //组合结构存放于 S 堆栈
    int k = 0;
    do
    {
        while (TotalVolume > 0 && k < n)
        {   //剩余总体积和货物没有尝试完,继续尝试新组合
            if (TotalVolume - element[k].weight >= 0)
            {   //剩余总体积与要进入当前组合的货物体积之差大于 0 时,货物进入当前组合
                temp.index = k;
                S.Push(temp);                  //k 编号的货物进入当前组合
                TotalVolume = TotalVolume - element[k].weight;
                //剩余总体积减去 k 编号的货物体积
            }
            k++;
        }
        if (TotalVolume == 0)
          TraverseStack(S,element);            //输出堆栈中已找到的一个货物组合方案的数据
        S.Pop(temp);                           //出栈,准备重新选择其他货物编号进栈
        k = temp.index;
        TotalVolume = TotalVolume + element[k].weight;
        //出栈货物体积还原剩余总体积
        k++;                                   //重新选择其他货物编号
```

```
    } while (!S.IsEmpty() || k != n);
        //堆栈中还原货物存在,或者货物没有试完,继续寻找新组合
}
```

图 3.8 是 TotalVolume＝100 时 6 件货物求解的结果，有 4 种方案，每一种方案都是上述算法得出的结果。

********** 背包问题求解 **********

********** 货物信息 **********

货物编号	货物名称	存放位置	货物重量
10000	AAAAA	wwww0	20
10001	BBBBB	wwww1	30
10002	CCCCC	wwww2	40
10003	DDDDD	wwww3	50
10004	EEEEE	wwww4	80
10005	FFFFF	wwww5	10

****

运送方案货物信息

货物编号	货物名称	存放位置	货物重量
10005	FFFFF	wwww5	10
10002	CCCCC	wwww2	40
10001	BBBBB	wwww1	30
10000	AAAAA	wwww0	20

该方案有以上4件货物

运送方案货物信息

货物编号	货物名称	存放位置	货物重量
10003	DDDDD	wwww3	50
10001	BBBBB	wwww1	30
10000	AAAAA	wwww0	20

该方案有以上3件货物

运送方案货物信息

货物编号	货物名称	存放位置	货物重量
10004	EEEEE	wwww4	80
10000	AAAAA	wwww0	20

该方案有以上2件货物

运送方案货物信息

货物编号	货物名称	存放位置	货物重量
10005	FFFFF	wwww5	10
10003	DDDDD	wwww3	50
10002	CCCCC	wwww2	40

该方案有以上3件货物

图 3.8　背包问题求解结果

3.5.4　地图四染色问题求解

用不多于 4 种颜色为地图染色，使相邻的行政区不重色，是计算机科学中著名的四色定理的典型应用，这个定理的思想是以回溯的算法对一幅给定的地图染色。这种解法不是按照固定的计算法则，而是通过尝试与纠正错误的过程来完成任务的。其典型特征是：尝试可能最终解决问题的各个步骤，并加以记录。而随后当发现某步骤进入“死胡同”、不能解决问题时，就把它取出来删掉，再取一个新的值尝试，直至所有数据都尝试过或者找到所求结果。这种尝试与纠正错误的过程本身就是进栈和出栈的一个典型应用的例子。

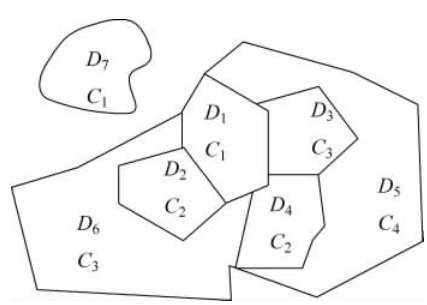

图 3.9　地图的染色

下面根据上述思想，结合实例来学习四染色问题的应用。假设已知地图的行政区如图 3.9 所示，

各行政区的名称以 $D_1 \sim D_7$ 来表示,同时用 $C_1 \sim C_4$ 来表示各区的颜色。

染色过程的思想如下:

设当前准备染色的行政区是 D_i,当前准备使用的颜色是 C_j。

从第一号行政区开始逐一染色,每一个区域逐次用颜色 C_1,C_2,C_3,C_4 进行试探,直到所有区域全部被染色。

若用 C_j 颜色为 D_i 行政区染色,则与周围已染色的行政区(进入堆栈的行政区为暂时确定了颜色的区域)不重色,则将该区域的 C_j 和 D_i 进栈(进栈内容为地区编号和颜色编号);否则依次用下一颜色进行试探。

若对当前处理的地区用 4 种颜色中的任意一种颜色染色,均与已染色的相邻区域发生重色,则需修改当前栈顶(上一个染色地区)的颜色,即出栈回溯,以另一种颜色对出栈的地区重新染色。

实现这个算法时用一个 $n \times n$ 的相邻关系矩阵 $\boldsymbol{r}$ 来描述各地区之间的边界关系,若第 i 号区域与第 j 号区域相邻,则 $r[i][j]=1$,否则 $r[i][j]=0$。关系矩阵如图 3.10 所示。

$$\boldsymbol{r}=\begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

图 3.10 区域相邻的关系矩阵

用栈 S 记载每个区域当前的染色结果,堆栈中的元素记载每个被染色的地区和对应的颜色。

染过色的区域(可能是暂时确定的结果)存放在 S 堆栈中,当前染色的区域要与堆栈中已染色的区域逐一比较是否发生冲突。逐一比较就是要将 S 堆栈中的已染色区域逐一出栈比较,而这些堆栈不是最终出栈,而是为了比较的临时出栈,无论是否发生冲突,临时出栈的数据都要重新再进栈。

在算法中,创建另一个堆栈 Skeep,保存 S 堆栈中临时出栈的数据,比较完成后,再将 Skeep 堆栈中的数据出栈,并进栈到 S 堆栈中,实现 S 堆栈数据的恢复。

图 3.11 为区域 D_1、D_2、D_3 的染色结果(暂时结果)。

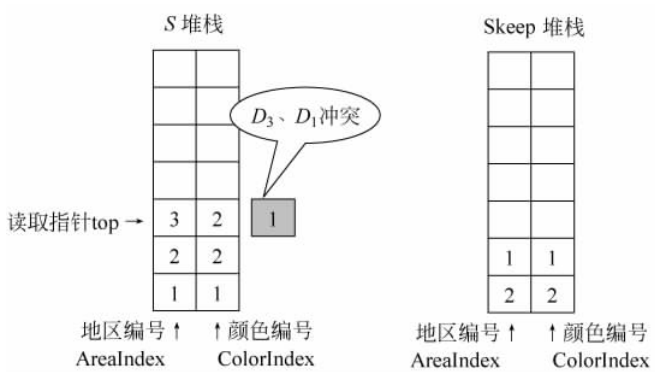

图 3.11 D_1、D_2、D_3 染色堆栈

D_1 首先用颜色 C_1 染色。

D_2 首先用颜色 C_1 染色,与进入堆栈中染过色的 D_1 进行比较,由于 D_2 与 D_1 相邻,所

以不能使用颜色 C_1，调整颜色为 C_2，再为 D_2 染色。

D_3 首先用颜色 C_1 染色，与进入堆栈中染过色的 D_2、D_1 进行比较，由于 D_3 与 D_1 相邻，且同为颜色 C_1，所以不能使用颜色 C_1；调整颜色为 C_2，再与进入堆栈中染过色的 D_2、D_1 进行比较，无冲突，所以为 D_3 使用 C_2 进行染色。

Skeep 的状态是最后一次恢复 S 堆栈前的状态。

图 3.12 为区域 D_4 的染色过程。

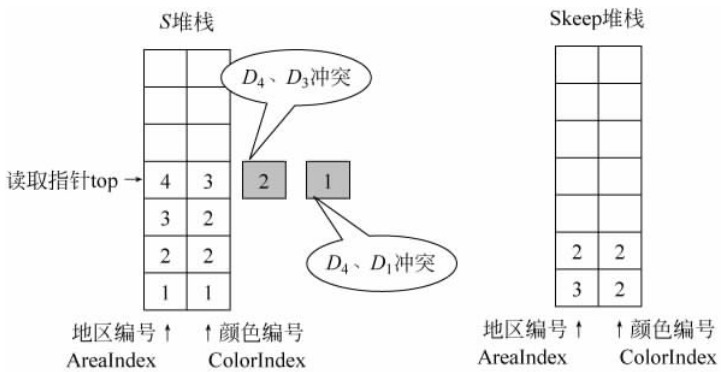

图 3.12　D_4 染色堆栈

D_4 首先用颜色 C_1 染色，与进入堆栈中染过色的 D_3、D_2、D1 进行比较，由于 D_4 与 D_1 相邻，且同颜色 C_1，所以不能使用颜色 C_1；调整颜色为 C_2，再与进入堆栈中染过色的 D_3、D_2、D_1 进行比较，由于 D_4 与 D_3 相邻，且同颜色 C_2，所以不能使用颜色 C_2；调整颜色为 C_3，再与进入堆栈中染过色的 D_3、D_2、D_1 进行比较，无冲突，所以为 D_4 使用 C_3 进行染色。

Skeep 的状态是最后一次恢复 S 堆栈前的状态。

图 3.13 为区域 D_5 的染色过程。

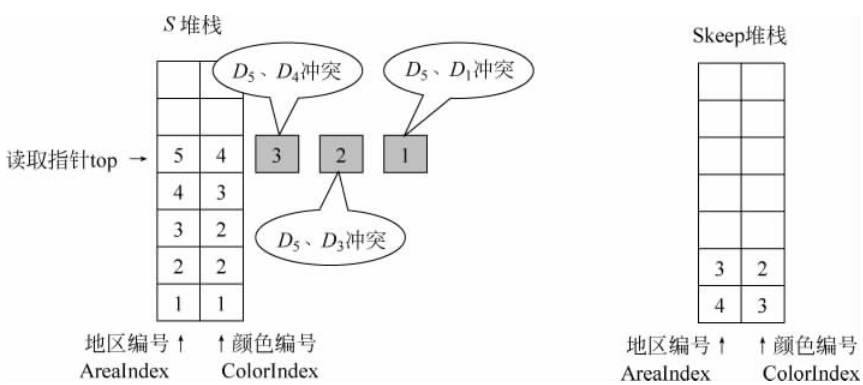

图 3.13　D_5 染色堆栈

D_5 首先用颜色 C_1 染色，与进入堆栈中染过色的 D_4、D_3、D_2、D_1 进行比较，由于 D_5 与 D_1 相邻，且同颜色 C_1，所以不能使用颜色 C_1；调整颜色为 C_2，再与进入堆栈中染过色的 D_4、D_3、D_2、D_1 进行比较，由于 D_5 与 D_3 相邻，且同颜色 C_2，所以不能使用颜色 C_2；调整颜色为 C_3，再与进入堆栈中染过色的 D_4、D_3、D_2、D_1 进行比较，由于 D_5 与 D_4 相邻，且同颜色

C_3,所以不能使用颜色 C_3;调整颜色为 C_4,再与进入堆栈中染过色的 D_4、D_3、D_2、D_1 进行比较,无冲突,所以为 D_5 使用 C_4 进行染色。

Skeep 的状态是最后一次恢复 S 堆栈前的状态。

图 3.14 和图 3.15 为区域 D_6 的染色过程。

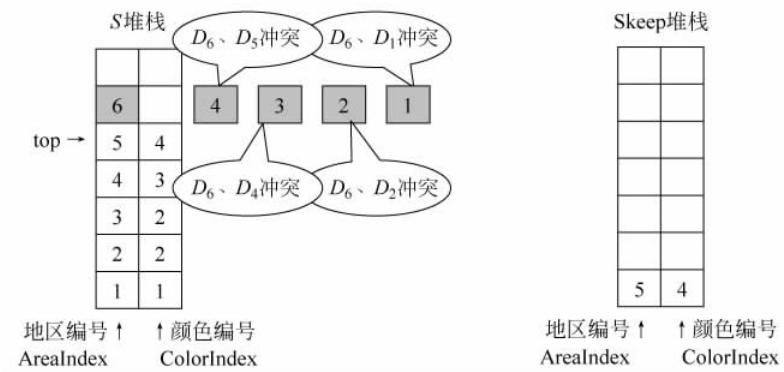

图 3.14 D_6 染色堆栈(一)

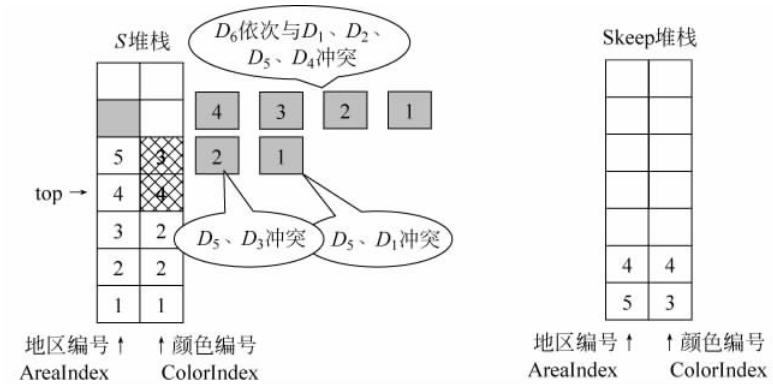

图 3.15 D_6 染色堆栈(二)

为 D_6 染色时,使用 C_1 与 D_1 冲突,使用 C_2 与 D_2 冲突,使用 C_3 与 D_4 冲突,使用 C_4 与 D_5 冲突。由于所有颜色全部使用,已无其他颜色可用,说明前面染色的区域有问题,只能通过出栈,对前面已染过色的区域重新调整颜色。

Skeep 的状态是最后一次恢复 S 堆栈前的状态。

所以,退出 D_5,D_5 也已经使用过所有的颜色,所以再退出 D_4,D_4 调整颜色为 C_4,重新染色。再对 D_5 使用 C_1 染色,与 D_1 冲突,再对 D_5 使用 C_2 染色,与 D_3 冲突,再对 D_5 使用 C_3 染色,不冲突。然后,再次对 D_6 进行染色。

为 D_6 再染色过程中,使用 C_1 与 D_1 冲突,使用 C_2 与 D_2 冲突,使用 C_3 与 D_5 冲突,使用 C_4 与 D_4 冲突。由于所有颜色全部使用,已无其他颜色可用,说明前面染色的区域仍然有问题,只能通过出栈,对前面已染过色的区域进行重新调整颜色。后面的染色过程的思想与前述过程相同。

下面给出四染色问题的数据结构定义和算法 MapColor 如下:

```
#define MAP       ElementType                    //实例数据元素句柄化
#define AreaIndex key                            //实例数据元素关键字句柄化
struct MAP
{
    int  AreaIndex;                              //地区编号
    char AreaName[20];                           //地区名称
    int  ColorIndex;                             //颜色编号
};
class StackType
{
  public:
    int AreaIndex;                               //地区编号
    int ColorIndex;                              //颜色编号
};
void MapColor(int r[8][8], int n ,Stack<StackType>&S)
{   //将地图用 4 种颜色染色,n 个地区间的相邻关系在 r 数组中表示
    int MaxStackSize=20;
    Stack<StackType> Skeep(MaxStackSize);        //创建堆栈对象 Skeep
//  Stack<StackType> S(MaxStackSize);            //创建堆栈对象 S
    //染色结果在 S 堆栈中
    //如果本算法在外部创建 S 堆栈,作为引用参数代入此算法中,结果会带出
    //可以在本算法中创建堆栈对象 S,用 return S 返回结果
    StackType x,temp;                            //StackType 是堆栈元素的数据类型
    bool      flag;

    int currentArea = 1;                         //当前准备染色的区域编号,从 1 号地区开始
    int currentColor = 1;                        //当前准备使用的颜色编号,从 1 号颜色开始
    x.AreaIndex = currentArea;
    x.ColorIndex = currentColor;
    S.Push(x);                                   //1 号地区以 1 号颜色染色,并进栈
    currentArea ++;                              //地区编号增 1,准备为新地区染色
    while (currentArea <= n)
    {
        flag = true;              //flag 为真时,表示与堆栈中已染色的区域比较时未发现重色
        while (!S.IsEmpty() && flag)
        {   //从栈顶至栈底与已染色区域逐个比较有无重色
            S.Pop(x);                            //读取栈中一个数据元素,比较冲突
            Skeep.Push(x);                       //出栈元素保存到 Skeep 堆栈,以便恢复
            if (x.ColorIndex == currentColor && r[currentArea][x.AreaIndex] )
            //栈中读出的区域与当前准备染色的区域要使用的颜色相同,且两个区域相邻
                flag = false;          //无法使用当前颜色染色,需改变颜色或出栈,结束比较
        }
        if (flag)
        {   //与已染色区域比较,无一同色
            //将当前区域号及使用颜色进栈,并准备下一个区域号,从颜色 1 开始尝试
            x.AreaIndex = currentArea;
            x.ColorIndex = currentColor;
            while(!Skeep.IsEmpty())
            {   //从 Skeep 堆栈中恢复 S 堆栈的数据(S 中为暂时染色的区域)
                //准备下一个区域染色时堆栈的初始数据
                Skeep.Pop(temp);
```

```
                S.Push(temp);
            }
            S.Push(x);                          //当前染色区域进栈
            cout<<"    【进栈】"<<endl;
                S.DisplayStack();               //这个输出可以给出整个进栈过程
            currentArea++;
            currentColor = 1;

        }
        else
        {
            currentColor++;                     //准备用下一种颜色重新尝试
            while(!Skeep.IsEmpty())
            {   //从 Skeep 堆栈中恢复 S 堆栈的数据(S 中为暂时染色的区域),
                //退出栈顶染色区域,重新染色
                Skeep.Pop(temp);
                S.Push(temp);
            }
            while (currentColor>4)
            {   //如果当前使用的颜色或退出的栈顶使用的颜色超过 4
                //说明栈顶染色不对,需出栈
                S.Pop(x);                       //退出栈顶染色区域,重新染色
                cout<<"    【出栈】"<<endl;
                S.DisplayStack();               //这个输出可以给出整个出栈过程
                currentColor = x.ColorIndex + 1;                    //调整染色编号,再尝试
                currentArea = x.AreaIndex;
            }
            flag = true;
        }
    }
//return S;                     //S 堆栈在此算法内部定义时,可以使用这个方式返回 S 中的结果
}
```

堆栈的运算在计算机科学中应用相当广泛,无论是在系统软件还是在应用设计软件中,堆栈随处可见,在第 4 章中树的多种非递归运算都将使用堆栈。

3.6 队列的定义

3.6.1 队列的逻辑结构

逻辑上,队列也本应属于线性表的范畴,只是与线性表相比,队列的运算受到了严格的限制(故也称为限定性线性表)。

队列是一个线性表,其数据元素只能从这个线性表的一端插入,插入端称为队列的队尾(rear),而从另一端删除,删除端称为队列的队头(front)。

也可以说,队列是限定只能在表头(或表尾)进行插入和在表尾(或表头)删除运算的线

性表。表头和表尾为开放运算的两端。

现在用实例来说明这个定义的含义。图 3.16 是一个队列，依次往队列中插入 4 个元素 X、A、B、C、D。则 X 在队列的最前面，B 在 A 的后面，C 在 B 的后面，D 又在 C 的后面。在这个队列中只能先取出 X，才能取出 A，图中给出了取出 X 后的示意图。

取数据(删除)又称出队，总是从队列中 front 所指的位置取出数据。插入数据又称进队，总是在队列中 rear 所指的位置后面添加。取出数据与插入时的顺序恰好相同，故队列也称为先进先出表，简称 FIFO(First In First Out)表。

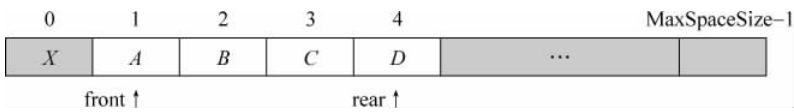

图 3.16 队列

3.6.2 队列的抽象数据类型

队列的抽象数据类型如下：

```
ADT Queue
{
    Data: 一个从两端分别进行插入和删除的限定性线性表
    Relation: 队列的一端称为队头(front),而另一端称为队尾(rear)
    Operation:
        CreatQueue(MaxQueueSize)     //构造大小为 MaxQueueSize 的空队列
        IsEmpty()                    //判断队列是否为空,如果为空返回 true,否则返回 false
        IsFull()                     //判断队列是否为满,如果为满返回 true,否则返回 false
        GetFront(result)             //返回队列队头元素到 result 中
        GetRear(result)              //返回队列队尾元素到 result 中
        EnQueue(newvalue)            //向队列添加元素 newvalue(进队)
        DeQueue(result)              //从队列取出元素到 result 中(出队)
}
```

进队运算是将数据元素值 newvalue 添加到队列的队尾指针所指的元素后面的空间中。它相当于线性表的插入运算，即在线性表的末端插入一个元素。

出队操作是将队列的队头元素取出，相当于线性表的删除运算，即删除线性表的第一个元素。

判队列空运算的作用是判断队列是否为空。为空时，若再执行出队操作就发生错误，这种情况称为下溢(underflow)。

判队列满运算的作用是判队列是否已满(即预留的空间已被元素充满)，若满，再执行入队运算就发生错误，称为上溢(overflow)。

返回队头数据元素即取出队列的队头上的元素值，但不删除队头上的元素，相当于查找第一个数据元素。

如同线性表一样，队列也有顺序存储和链式存储两种存储方式。在不同的存储方式下，上述运算的实现是不相同的，下面就两种不同的存储方式讨论队列运算的实现。

3.7 队列的顺序存储及操作

3.7.1 队列的顺序存储

1. 队列顺序存储概念

队列顺序存储方式是将队列中的数据元素依次存放于存储器相邻的单元,来保证队列数据元素逻辑上的有序性。

但为队列分配的连续存储空间的第一个存储单元不一定存储的是队列队头元素。这是因为,如果第一个存储单元总是存储队列中队头元素(front 指针不动),就会出现每次出队一个元素,就要将队列中后面的元素全部逐个地向队头方向移动一个位置,这会产生大量数据的移动,影响算法效率,如图 3.17 和图 3.18 所示。

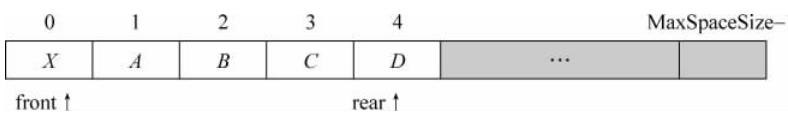

图 3.17 队列出队前

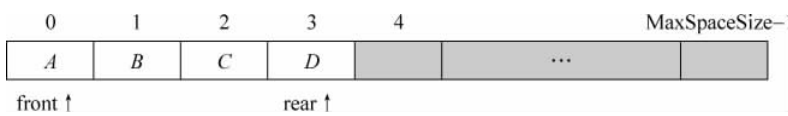

图 3.18 队列出队后(元素前移)

如果定义队列在每次进队或出队时,队头指针 front 或队尾指针 rear 就后移一个位置,是否就解决了队列的所有问题呢? 回答是否定的。因为,如果出队或进队时,队列指针单纯地向同一个方向移动,就会造成队列像一条蠕虫慢慢地“爬过”队列定义的全部空间,而这条蠕虫只要“爬过”的空间就无法再次得到利用,即造成 front 指针前面的存储单元不能再存储数据元素,如图 3.19 所示。

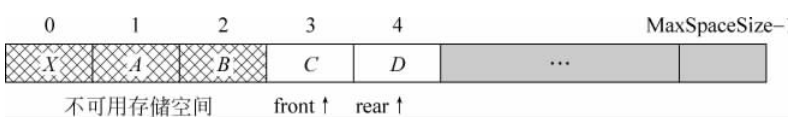

图 3.19 队列“爬过”存储空间

可见,无论是用静态队头方式还是用简单的动态队头方式构造队列运算,都会出现问题,要想合理地解决这些问题,就要重新定义队列中指针的变化。

解决的办法是,从逻辑上将队列空间的头尾看成是相连的,即 0 下标的存储单元与 MaxSpaceSize−1 下标的存储单元是相邻的。无论是 front 指针还是 rear 指针移到最后一个存储单元(MaxSpaceSize−1 位置)时,如果继续后移,就会移到队列存储的开始位置(0 下标位置)。这样,队列就可以循环地利用所定义的存储空间,由于队列的指针动起来了,出队运算时,不必移动大量数据,空间又不会被慢慢地耗尽,只要还有空间未存放队列元素,队列

就不会满。图 3.20 就是队列指针“转头”的示例。

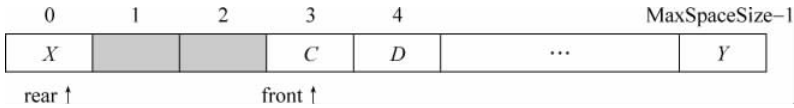

图 3.20　队列指针转头

事实上，至此为止问题还没有完全解决。下面首先分析队列中如果没有一个队列元素（即队列为空时）队列的指针状态。以图 3.20 为例，将队列中的所有元素全部出队。队列出队时，总是将 front 所指的元素值取走，然后将 front 指针后移一个位置。队列中的全部元素出队后队列的指针状态如图 3.21 所示。可见，队空时 front 指针指向 rear 指针的“下一个”位置。

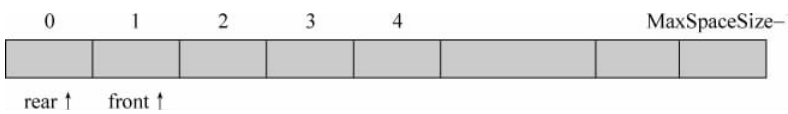

图 3.21　队列为空

如果在图 3.20 所示中再插入若干个元素，直到队满（所有存储空间都存放数据元素），则出现如图 3.22 所示的状态。

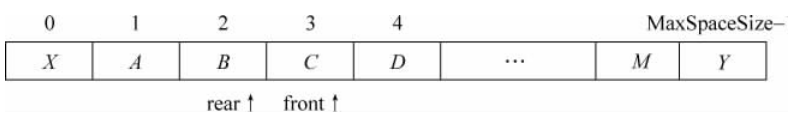

图 3.22　队列为满

从图 3.22 可见，front 指针指向 rear 指针的“下一个”位置。就是说，无论队空或队满，指针的相对位置是一样的。这就使判断队空和队满出现问题。

解决这个问题的方法是：在定义的队列存储空间中留出一个数据元素的空间不用（为了保持队列空间有 MaxSpaceSize，在定义数组时，设数组的大小是 MaxSpaceSize+1），可以理解这个不用的空间是一个环形管中的“活塞”，它总是紧邻队列中的第一个数据元素，是可以移动的，即这个空间随着队头元素的移动而移动；另外，将队头指针做一点调整，使 front 指针始终指在这个不用的空间位置上，队尾指针仍然指在队列中的最后一个数据元素，如图 3.23 所示。

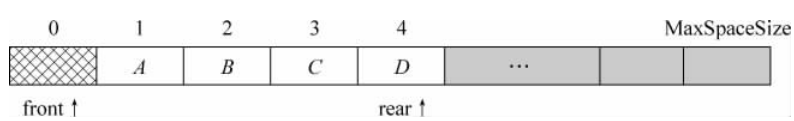

图 3.23　循环队列

如此定义后，再来讨论出队及队空、进队及队满的情况。出队时，首先将 front 指针移动到“下一个”位置，再将 front 所指的数据元素取出（如图 3.24 所示，数据元素 A 出队）。

当队列中的所有数据元素全部出队后，队列为空时，front 和 rear 正好指向同一个位置，即 front＝rear，如图 3.25 所示。

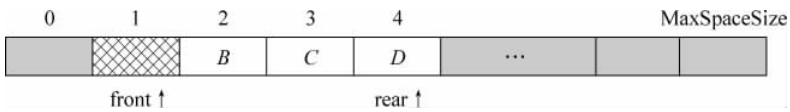

图 3.24 循环队列队头出队

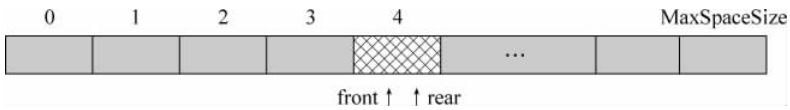

图 3.25 循环队列为空状态

进队时,首先将 rear 指针移动到“下一个”位置,再将新数据元素存到 rear 所指的位置,如果队列存满了就会出现图 3.26 所示的情况。

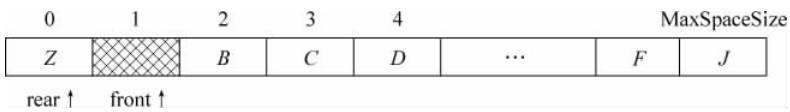

图 3.26 循环队列存满

此时,队尾指针 rear 的“下一个”位置就是 front 指针所指的位置,即队满时的指针状态。

至此,通过付出一个存储空间的代价,最终解决了对动态指针的循环队列区分队空和队满的问题。

图 3.27 就是一个循环队列的完整结构。

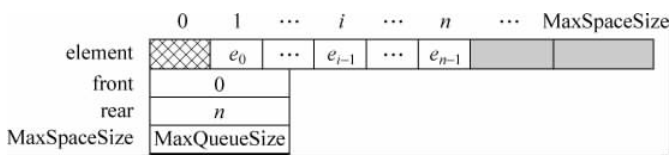

图 3.27 循环队列的顺序存储结构

2. 顺序存储结构队列的模板类定义

```
template<class QueueType>
class Queue
{   //队列模板类定义.队头指针指向队头元素前一个位置,队尾指针指向队尾元素位置
    //数据元素存放于 element[0..MaxSpaceSize]范围
public:
    Queue(int MaxQueueSize = 20);              //构造函数(构造循环队列)
    ~Queue() {delete [] element;};             //析构函数
    bool IsEmpty(){return front == rear;};     //判断队列空
    bool IsFull(){return (front == (rear+1) % (MaxSpaceSize+1) ? 1 : 0);};
                                               //判断队列满
    bool GetFront(QueueType& result);          //获取队头元素值
    bool GetRear(QueueType& result);           //获取队尾元素值
    bool EnQueue(QueueType& newvalue);         //循环队列进队
    bool DeQueue(QueueType& result);           //循环队列出队
```

```
private:                                          //队列数据结构定义
    QueueType * element;                          //队列元素空间
    int       front;                              //队头指针
    int       rear;                               //队尾指针
    int       MaxSpaceSize;                       //队列空间大小
};
```

Queue 是一个顺序存储的队列，其中 element 是一个一维数组首地址，每个数组元素空间用于存放队列元素数据值，front 指向队列的队头元素的“前一个”位置，rear 指向队列中的队尾元素，MaxSpaceSize 记载队列可存储的最多数据元素（实际的数组空间为 MaxSpaceSize+1）。

3.7.2 顺序存储结构下队列的运算实现

下面讨论顺序存储结构下队列的几种主要运算。

1. 构造空队列

所谓空队列，是指队列中没有一个数据元素，但已有数据元素的空间和队列结构，如图 3.28 所示。

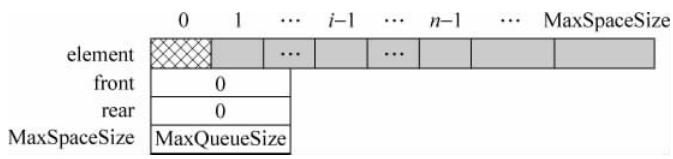

图 3.28 循环队列的顺序存储空队列

空队列产生后，就存在一个 ElementType 类型的数组，大小是 MaxSpaceSize+1，队列设为 0（用户约定），即队列为空时，队列的队头和队尾指针“指向”数据存储区的首地址空间。不难分析，算法的时间复杂性是 $O(1)$。

构造空队列算法（Queue 类的构造算法）如下：

```
template<class QueueType>
Queue<QueueType>::
Queue(int MaxQueueSize)
{   //队列构造算法
    MaxSpaceSize = MaxQueueSize;
    element = new QueueType[MaxSpaceSize + 1];
    front = 0;
    rear = 0;
}
```

2. 返回队列队头元素的值

返回队列队头元素的值，是将 front 后面一个位置的队列元素的值取出，但是 front 指针不移动。

返回队列中队头元素的值的算法 GetFront 如下：

```
template<class QueueType>
bool Queue<QueueType>::
GetFront(QueueType& result)
{   //获取队头元素值
    if (IsEmpty()) return false;
    result = element[(front + 1) % (MaxSpaceSize + 1)];
    return true;
}
```

3. 返回队列队尾元素的值

返回队列队尾元素的值,是将 rear 位置的队列元素的值取出,但是 rear 指针不移动。返回队列中队尾元素的值的算法 GetRear 如下:

```
template<class QueueType>
bool Queue<QueueType>::
GetRear(QueueType& result)
{   //获取队尾元素值
    if (IsEmpty()) return false;
    result = element[rear];
    return true;
}
```

4. 进队运算

进队列运算是将一个新元素 newvalue 存储到当前 rear 所指空间的"下一个"位置。进队时,首先要判断队列中是否存在元素存放的空间,即先判断队列是否满,队列不满时,newvalue 可以进队列,否则出错。

进队算法 EnQueue 如下:

```
template<class QueueType>
bool Queue<QueueType>::
EnQueue(QueueType& newvalue)
{   //循环队列进队
    if (IsFull()) return false;
    rear = (rear + 1) % (MaxSpaceSize + 1);  //循环队列指针移到下一个位置
    element[rear] = newvalue;
    return true;
}
```

5. 出队运算

出队运算是将队列中队头指针所指的"下一个"位置的元素取出。做法是:首先将队列队头指针 front 先向"下一个"位置移动,然后,取出移动后 front 所指的数据元素。出队时,首先要判断队列中是否有元素可取,即先判断队列是否空,不空时,可以出队,否则出错。注意,这个算法与取队列队头元素值的算法 GetFront 有所不同,两个算法都可以取得队列中队头的元素值,但 GetFront 算法取值后不会移动 front 指针,取值后队列中元素的个数也不

发生改变。

出队算法 DeQueue 如下：

```
template < class QueueType >
bool Queue < QueueType > : :
DeQueue(QueueType& result)
{   //循环队列出队
    if (IsEmpty()) return false;
    front = (front + 1) % (MaxSpaceSize + 1);//循环队列指针移到下一个位置
    result = element[front];
    return true;
}
```

上面讨论的是队列的顺序存储及相关操作，可以看到，队列的顺序存储运算中主要是指针的变化较复杂。

3.8 队列的链式存储及操作

3.8.1 队列的链式存储

队列的链式存储结构的特点是用物理上不一定相邻的存储单元来存储队列的元素，为了保证队列元素之间逻辑上的连续性，存储元素时，除了存储它本身的内容以外，还附加一个指针域（也叫链域）来指出逻辑上相邻元素的存储地址。

在 C++语言中，首先定义动态存储空间分配方式下队列的数据元素的类：

```
template < class ElementType >
class ChainNode
{
public:
    ElementType              data;
    ChainNode < ElementType > * link;
};
```

在动态链式结构中，队列结点由两部分构成，一是数据元素的数据域，二是数据元素的链接域。链接域指向后一个相邻的队列元素存储空间的起始地址，链式队列中的第一个结点就是队头元素，最后一个结点就是队尾元素，最后一个结点的链接域的值为空。

图 3.29 给出了一个链式队列的结构。front 始终指向队列队头结点，即链表的第一个结点。rear 始终指向队列队尾结点，即链表的最后一个结点。链式队列一般不存在队列满问题，除非存储空间全部被耗尽；链式队列空表现为链式队列的 front 和 rear 的值同时为空，即链表为空。

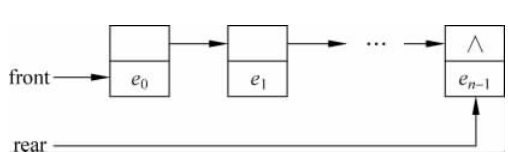

图 3.29 链式队列

3.8.2 链式队列模板类的定义

链式队列模板类的定义如下：

```
template<class ElementType>
class ChainQueue {
public:
    ChainQueue() {front = NULL;rear = NULL;};  //构造函数
    ~ChainQueue();                             //析构函数
    bool IsEmpty() {return front == NULL;};    //判断队空
    bool GetFront(ElementType& result);        //获取队头结点元素值
    bool GetRear(ElementType& result);         //获取队尾结点元素值
    bool EnQueue(ElementType& newvalue);       //进队
    bool DeQueue(ElementType& x);              //出队
private:
    ChainNode<ElementType> *front;             //front 指向队头结点
    ChainNode<ElementType> *rear;              //rear 指向队尾结点
};
```

3.8.3 链式队列的操作

1. 释放链式队列空间

释放链式队列空间就是链式队列的析构运算。链表结构的所有结点空间都是动态申请，只能使用 delete 运算释放空间。

链式队列析构算法~ChainQueue 如下：

```
template<class ElementType>
ChainQueue<ElementType>::
~ChainQueue()
{   //析构函数
    ChainNode<ElementType> *nextPtr;
    while (front)
    {
        nextPtr = front->link;
        delete front;
        front = nextPtr;
    }
}
```

2. 获取队列队头元素值

返回链式队列队头元素的值，是指将 front 所指的队列元素的值取出，但是 front 指针不改变。

获取队列队头元素值算法 GetFront 如下：

```
template<class ElementType>
bool ChainQueue<ElementType>::
```

```
GetFront(ElementType& result)
{   //获取队列队头元素值
    if (IsEmpty()) return false;
    result = front -> data;                    //引用变量 result 返回队头元素值
    return true;
}
```

3. 获取队列队尾元素值

返回链式队列队尾元素的值,是指将 rear 所指的队列元素的值取出。

获取队列队尾元素值算法 GetRear 如下:

```
template < class ElementType >
bool ChainQueue < ElementType >::
GetRear(ElementType& result)
{   //获取队列队尾元素值
    if (IsEmpty()) return false;
    result = rear -> data;
    return true;
}
```

4. 链式队列进队运算

进队运算是将一个新元素值 newvalue 的结点链入到链式队列中,作为链表的最后一个结点。

链式队列进队算法 EnQueue 如下:

```
template < class ElementType >
bool ChainQueue < ElementType >::
EnQueue(ElementType& newvalue)
{   //newvalue 元素进栈
    ChainNode < ElementType > * p = new ChainNode < ElementType >;
    p -> data = newvalue;
    p -> link = NULL;
    if (front)
        rear -> link = p;                      //队列非空时,链在队尾结点后面
    else
        front = p;                             //队列空时,队头指向进队结点
    rear = p;                                  //队尾指向进队元素结点
    return true;
}
```

5. 链式队列出队运算

链式队列的出队运算是将 front 指针所指的结点值取出,且将队列指针 front 指向下一个结点,并将原来的第一个结点释放。出队列时,首先要判断队列中是否存在元素结点可取,即先判断队列是否空,不空时,可以出队,否则出错。注意,这个算法与取队列队头元素值的算法 GetFront 有所不同, GetFront 算法取值后不会改变 front 指针。

链式队列出队算法 DeQueue 如下：

```
template<class ElementType>
bool ChainQueue<ElementType>::
DeQueue(ElementType& result)
{   //出栈元素到 result
    if (IsEmpty()) return false;
    result = front->data;
    ChainNode<ElementType> *p = front;
    front = front->link;
    delete p;
    return true;
}
```

3.9 队列的应用

3.9.1 列车重排

货运列车共有 n 节车厢，各节车厢以不同的车站为目的地。假定 m 个车站的编号分别为 1,2,…,m，货运列车按照第 m 站至第 1 站的次序经过这些车站。车厢到达某个目的地时，为了便于从列车上卸掉尾部的车厢，就必须重新排列车厢，使各车厢排列为：靠近车头的车厢是到达第 1 站(最后一站)，而靠近车尾的车厢是到达 m 站(第一站)。在出发站时，准备发出的车厢编号存储在 r[]数组中，$r[i]$的值是车厢的编号，编号越小的，就是发往越远的车站。

如果开始时 r[]中的次序是 4,7,1,5,8,6,9,2,3，如图 3.30 所示，那么，在发车前，就要将车厢重新编排，按 1,2,3,4,5,6,7,8,9 的次序与车头相连接，如图 3.31 所示。当所有的车厢按照这种次序排列好后，在到达每个车站时，只需卸掉尾部的车厢即可。

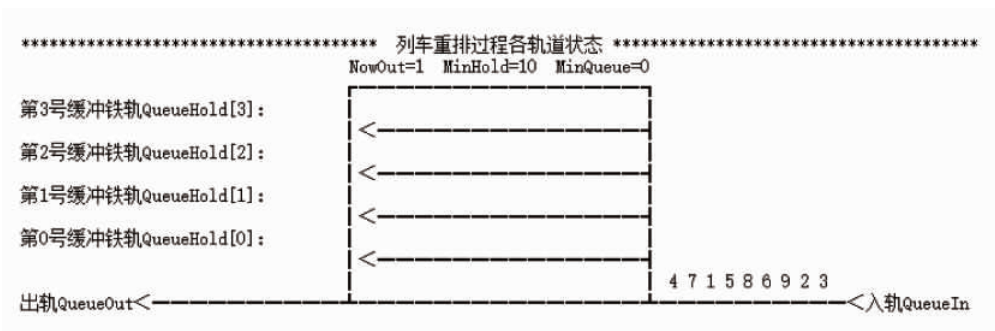

图 3.30 车厢调度转轨(初态)

为实现重排，在发车前，在本站的转轨站中进行调度，完成车厢的重排工作。在转轨站中有一个入轨、一个出轨和 k 个缓冲铁轨(位于入轨和出轨之间)。

开始时，车厢从入轨进入缓冲铁轨，再从缓冲铁轨按从小至大的次序离开缓冲铁轨，进入出轨。

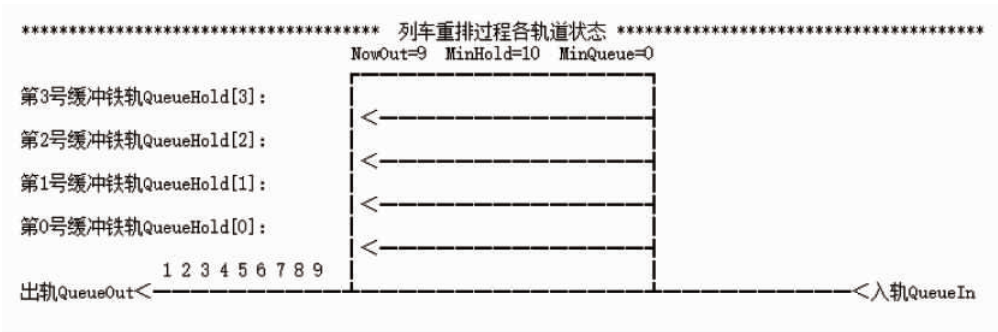

图 3.31　车厢调度转轨结果

为了重排车厢，需按车厢到达入轨的先后，依次检查入轨上的所有车厢。如果正在检查的车厢就是下一个满足排列要求的车厢，可以直接把它放到出轨上去。

如果不能直接移到出轨上，则把它移动到缓冲铁轨上，再按输出次序要求，直到轮到它时才将它放到出轨上。

车厢只能单向移动，即只能从入轨向出轨或缓冲铁轨方向移动，不能向回移动。

入轨上的数据形成入轨上的队列，入轨队列将来只有出队运算；出轨上的数据将来只有进队运算，初始时，出轨队列为空；缓冲铁轨是 k 个队列，k 值可以事先给定，初始时，缓冲铁轨队列为空。

调度过程中，有的车厢直接从入轨进入出轨，有的不能直接进入出轨，就移动到缓冲队列暂时存放，直到缓冲队列的车厢可以移动到出轨时，就将其从缓冲队列移动到出轨。

图 3.32 是在图 3.30 的入轨车厢排列的基础上移动 6 节车厢的状态图。其中，4、7 号车厢先移动到缓冲队列(进缓冲队列)，1 号车厢直接移动到出轨队列(进出轨队列)，5、8、6 只能先移动到缓冲队列(进缓冲队列)。缓冲队列中的数据还要满足队列中的数据由小到大，即，队头数据最小，队尾数据最大，这样才能保证不反方向移动车厢，也能在合适的时候将缓冲队列中的数据移动到出轨队列。

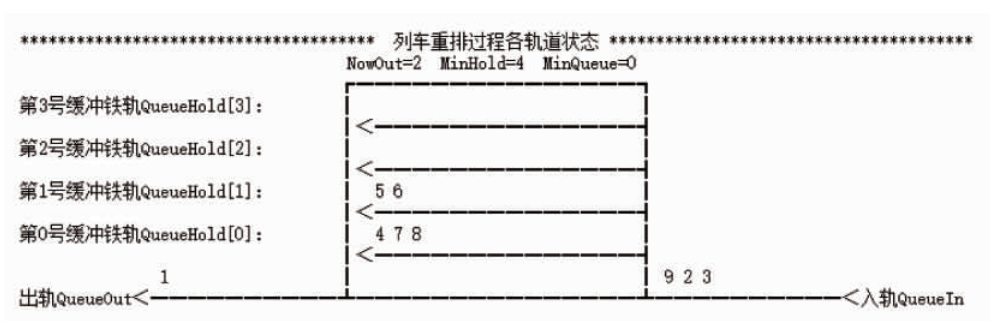

图 3.32　车厢调度转轨

图 3.33 所示，如果最初入轨队列的车厢排列是 9,8,7,6,5,4,3,2,1，缓冲队列有 4 条，调度过程中，9、8、7、6 这 4 节车厢各占一个缓冲队列，下一个要调度的 5 号车厢不能到出轨队列，又没有多余的缓冲队列可以进入，这种情况无法完成车厢的调度。这是缓冲队列不足

造成的结果,解决的方法是增加缓冲队列的个数。

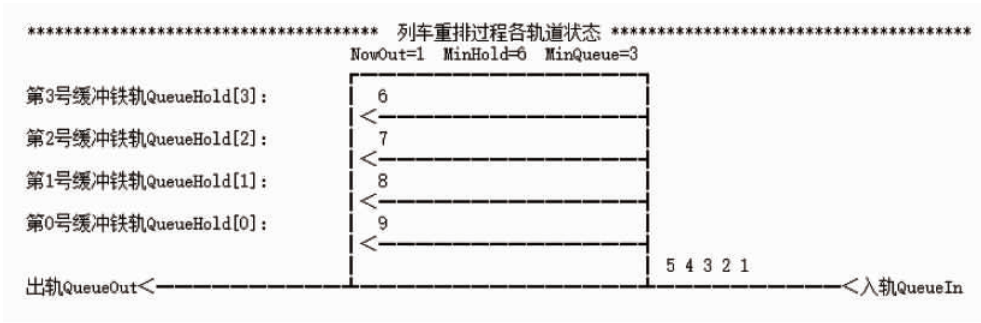

图 3.33 车厢调度转轨

重排车厢的算法由 RearrangementTrack 完成,它最多可使用 k 个缓冲铁轨,如果缓冲铁轨不足,就不能成功地重排,则返回 false,否则返回 true。

首先创建入轨上的队列 QueueIn,用于存放初始的车厢序列,用初始数据对 QueueIn 队列的数据初始化。

再创建出轨上的队列 QueueOut,用于形成有序车厢序列,即完成调度的车厢,此队列初始状态为空。

另外,创建一个指向 k 个队列首地址的指针 QueueHold,即,定义 k 个缓冲队列,也就是定义队列数组 QueueHold[i]($i=1,2,\cdots,k$),用于存放暂时无法直接调度到出轨的车厢。为了保证不会调度失败,缓冲队列数 k 可以取值为车厢数减 1。

NowOut 是下一个要输出至出轨的车厢号。

MinHold 是已进入各缓冲铁轨中编号最小的车厢,MinQueue 是 MinHold 编号车厢所在的缓冲队列编号。

RearrangementTrack 调度过程中调用 Output 和 Hold 两个算法。

RearrangementTrack 算法可以将入轨中能够直接移动到出轨的车厢移动至出轨,即数据从入轨队列出队,进入到出轨队列。

算法 Output 用于将当前在缓冲队列中可以送到出轨的车厢送至出轨队列,并修改 MinQueue 和 MinHold,它同时再判断缓冲铁轨中编号最小的车厢能否再送至出轨,如可以(MinHold==NowOut),则再将其送至出轨队列。

算法 Hold 根据车厢调度规则,把某个暂不能送至出轨的车厢 current 送入一个缓冲队列,如果 current 可以成为缓冲队列中新的编号最小车厢,就修改 MinQueue 和 MinHold。

将暂不能送至出轨的车厢移动到缓冲铁轨中时,采用如下的原则来确定应该移动到哪一个缓冲队列,这个原则可以减少缓冲队列的使用数。

(1) 该缓冲队列中进队车厢的编号均小于当前进队的车厢编号 current。

(2) 如果有多个缓冲铁轨满足条件(1),则选择一个缓冲队列队尾车厢编号最大的缓冲队列进入。

(3) 如果已有车厢的缓冲队列中队尾车厢编号都大于 current,则 current 选择一个空的缓冲队列(如果存在)进入。

(4) 如果无空缓冲队列可选择,则无法调度(缓冲铁轨数不足)。

列车重排算法 RearrangementTrack 如下：

```
class ElementType
{
public:
      int CarriageNumber;
};
bool Hold(Queue<ElementType> QueueHold[ ] ,
          int CarriageQuantity, int HoldQueueQuantity,
          int &MinHold, int &MinQueue, int &NowOut,int &current)
{   //为车厢 current 寻找最优缓冲铁轨,如果没有,则返回 false,否则返回 true
    int BestCushion = -1;              //最优缓冲队列编号,为-1 表示还未找到最优缓冲铁轨
    int BestLast = -1;                 //BestLast 保存 BestCushion 中最后一节车厢的编号
    ElementType temp;                  //进队出队车厢的编号变量
    for (int i = 0; i < HoldQueueQuantity; i++)
    //扫描所有缓冲铁轨,寻找最佳缓冲铁轨存放编号 current 的车厢
    if (!QueueHold[i].IsEmpty())
    {   //缓冲铁轨 i 不空,寻找最佳缓冲铁轨存放编号为 current 的车厢
        QueueHold[i].GetRear(temp);    //取得当前 i 号缓冲铁轨中最后一节车厢编号
        if (current > temp.CarriageNumber && temp.CarriageNumber > BestLast)
        {   //比较队尾车厢编号较大且小于 current 的车厢
            BestLast = temp.CarriageNumber;
            BestCushion = i;
        }
    }
    else                           //current 无法进入已使用的缓冲队列,进入未使用的缓冲铁轨 i
        if (BestCushion == -1) BestCushion = i;
    if (BestCushion == -1)
    {   //扫描所有缓冲铁轨,无可用的缓冲铁轨(BestCushion = -1),无法调度,失败
        cout <<"wrong!!!缓冲铁轨不足,无法调度,失败!"<< endl;
        return false;
    }
    temp.CarriageNumber = current;
    QueueHold[BestCushion].EnQueue(temp);//current 进入 BestCushion 队列中
    cout << "    【入轨到缓冲】从入轨将" << current << "号车箱移到最优缓冲铁轨"
         << BestCushion << endl;
    if (current < MinHold)
    {   //检查 current 可否成为新的 MinHold 和 MinQueue,如果是就修改
        MinHold = current;
        MinQueue = BestCushion;
    }
    return true;
}
void Output(Queue<ElementType> * QueueHold,
            Queue<ElementType> &QueueIn,
            Queue<ElementType> &QueueOut,
            int CarriageQuantity,int HoldQueueQuantity,
            int &MinHold,int &MinQueue,int &NowOut)
{   //从 MinQueue 中输出最小车厢 MinHold,并寻找新的最小的 MinHold 和 MinQueue
    int         current;               //当前车厢编号
    ElementType temp;
```

```
    QueueHold[MinQueue].DeQueue(temp); //编号最小的车厢 MinHold 从 MinQueue 出队
    cout << "    【缓冲到出轨】从"<< MinQueue <<"号缓冲铁轨输出"
        << MinHold << "号车箱到出轨" << endl;
    MinHold = CarriageQuantity + 1;   //假设一个最小车厢编号,它比实际车厢号大
    for (int i = 0; i < HoldQueueQuantity; i++)
    {   //比较所有缓冲队列中队头元素,寻找新的 MinHold 和 MinQueue
        QueueHold[i].GetFront(temp);   //获取 i 号缓冲队列的队头元素
        current = temp.CarriageNumber;
        if (!QueueHold[i].IsEmpty() && current < MinHold)
        {   //当前编号车厢比缓冲队列中最小车厢编号还小,替换 MinHold 和对应的 MinQueue
            MinHold = current;
            MinQueue = i;
        }
    }
}
bool RearrangementTrack (int CarriageNumber[],
                         int CarriageQuantity,
                         int HoldQueueQuantity)
{   //车厢初始排列为 CarriageNumber[1:n],如果重排成功返回 true,否则返回 false
    int         MaxQueueSize = 20;
    ElementType result;
    Queue<ElementType> QueueIn(MaxQueueSize);                    //创建【入轨】队列 QueueIn
    {   //对入轨队列 QueueIn 进行初始化,数据来自 CarriageNumber[]
        result.CarriageNumber = CarriageNumber[i];
    }
    Queue<ElementType> QueueOut(MaxQueueSize);                   //创建【出轨】队列 QueueOut
////////////////////////////////////////////////////////////////////////////
//创建[HoldQueueQuantity]条【缓冲轨道】上的数据队列,从 0 下标队列开始
//n 个车厢,有 n-1 个队列一定可调度成功.当所有车厢倒序排列时,需 n-1 个队列
////////////////////////////////////////////////////////////////////////////
Queue<ElementType> * QueueHold = new Queue<ElementType>[HoldQueueQuantity];
int NowOut = 1;                          //当前应该输出的车厢编号
int MinHold = CarriageQuantity+1;        //缓冲队列中编号最小的车厢编号,初值最大化
int MinQueue = 0;                        //MinHold 车厢对应的缓冲铁轨编号
for (i = 1; i <= CarriageQuantity; i++)
{   //重排车厢
    QueueIn.DeQueue(result);             //从入轨上的队列中出队一节车厢到 result 中
    if (result.CarriageNumber == NowOut)
    {   //当前入轨出队的车厢可直接到出轨,直接输出
        cout << "    【入轨到出轨】从入轨输出" << result.CarriageNumber
            <<"号车厢到出轨" << endl;
        QueueOut.EnQueue(result);        //输出到出轨的车厢进入出轨队列 QueueOut
        if(NowOut!= CarriageQuantity)
            NowOut++;
        while (MinHold == NowOut)
        {   //从缓冲铁轨中输出 MinHold
            result.CarriageNumber = MinHold;
            QueueOut.EnQueue(result); //输出到出轨的车厢进入出轨队列 QueueOut
            Output(QueueHold, QueueIn,QueueOut,
                   CarriageQuantity, HoldQueueQuantity,
                   MinHold, MinQueue, NowOut);
```

```
                if(NowOut!= CarriageQuantity)
                    NowOut++;
                }
            }
            else                                    //将 result.CarriageNumber 送入某个缓冲铁轨
            {   //Hold 返回 true 表示送入成功,否则因缓冲铁轨不足,调度算法失败
                if (!Hold(QueueHold,CarriageQuantity,HoldQueueQuantity,
                    MinHold,MinQueue,NowOut,result.CarriageNumber))
                    return false;
            }
        }
        return true;
    }
```

3.9.2 投资组合问题

企业已拥有一定资金,准备用这部分资金进行投资。经过投资分析,有多个可供投资的项目,并且可预知可供投资的项目的资金需求及投资回报率。面临的问题是:由于资金原因,不可能同时向需要资金量大的几个较高回报率的项目投资,只能从较高回报率项目和较低回报率项目中选择项目进行组合投资,以使投资回报最大化,同时不再追加资金。这样就可能产生多种组合方案,进而从多种组合方案中选择一种组合的处理。

为解决此类问题,首先决定哪些项目可以组合。不能同时选择的投资项目称为冲突项目。然后再核算各种项目组合的资金需求总量。如某种项目组合的资金需求总量超过已拥有的定量资金,则认为该种组合不可行;如存在多个组合方案的资金需求总量都不超过已拥有的资金总量,则企业经营者再从中选择投资回报率最大化的投资组合。

下面仅讨论求取各种不冲突组合的算法,这类问题又称为划分子集问题。企业决定将 $a_1,a_2,\cdots,a_n$ 项目作为投资候选项目,这里 a_i 表示候选项目的项目编号,抽象为项目集合 $A=\{a_1,a_2,\cdots,a_n\}$。

项目集合 A 中不能归入同一个投资组合方案的项目元素表现为项目集合 A 中的这些项目元素之间会发生冲突。为了说明项目集合 A 中各项目之间是否冲突,建立集合 A 中的项目关系集合 $R=\{(a_i,a_j)\}$。如 (a_i,a_j) 为 1(或 true),则表示 a_i 与 a_j 之间存在冲突;如 (a_i,a_j) 为 0(或 false),则表示 a_i 与 a_j 之间不冲突。

根据 R 所确定的关系将 A 集合划分成不冲突的若干个组合,即划分为不相交的子集 A_1, $A_2,\cdots,A_k(k\leqslant n)$,使任何子集上的元素均无冲突。

如果有 9 个投资项目,项目编号以整数表示,则集合 $A=\{1,2,3,4,5,6,7,8,9\}$,另根据调研,存在以下项目冲突关系:$R=\{(2,8),(4,9),(2,9),(2,1),(2,5),(2,6),(5,9),(5,6),(4,5),(5,7),(6,7),(3,7),(3,6)\}$。根据冲突关系集合导出一个冲突关系矩阵 r,如图 3.34 所示。关系矩阵中 a_i 与 a_j 对应位置值为 1,则表示冲突;a_i 与 a_j 对应位置值为 0,表示不冲突。

该冲突关系矩阵将用于划分子集处理算法,判断哪些项目可组合在同一个子集中。

定义一个状态数组,用于记录各项目经过划分后所属的子集编号。仍以上面的例子说明,最终可得出的可行子集划分为

$$r=\begin{array}{c} \\ 1\\2\\3\\4\\5\\6\\7\\8\\9 \end{array}\begin{array}{c} \begin{array}{ccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{array} \\ \begin{bmatrix} 0&1&0&0&0&0&0&0&0\\ 1&0&0&0&1&1&0&1&1\\ 0&0&0&0&0&1&1&0&0\\ 0&0&0&0&1&0&0&0&1\\ 0&1&0&1&0&1&1&0&1\\ 0&1&1&0&1&0&1&0&0\\ 0&0&1&0&1&1&0&0&0\\ 0&1&0&0&0&0&0&0&0\\ 0&1&0&1&1&0&0&0&0 \end{bmatrix} \end{array}$$

图 3.34 冲突关系矩阵

$$A_1=\{1,3,4,8\}\quad A_2=\{2,7\}\quad A_3=\{5\}\quad A_4=\{6,9\}$$

那么集合状态数组的最后结果如图 3.35 所示。

	1	2	3	4	5	6	7	8	9
set	1	2	1	1	3	4	2	1	4

图 3.35 可行子集划分结果

其中：

set[1]=set[3]=set[4]=set[8]=1，项目 a_1、a_3、a_4、a_8 属于同一子集，其子集编号是 1。

set[2]=set[7]=2，项目 a_2、a_7 属于同一子集，其子集编号是 2。

set[5]=3，项目 a_5 是一个子集，其子集编号是 3。

set[6]=set[9]=4，项目 a_6、a_9 属于同一子集，其子集编号是 4。

形成上述集合状态数组的过程就是划分子集的算法处理过程。为实现划分过程，定义一个循环队列 Q，初始化时，队列的每个元素存放项目编号，如图 3.36 所示。

图 3.36 可行子集划分结果(队列初态)

所有项目的信息存放在线性表中，如图 3.37 所示。每个项目包含以下信息：项目编号(ProjectNumber)、项目名称(name)、项目地址(place)、项目所属子集编号(set)。其中，项目所属子集编号(set)是划分子集的结果，初始时，所有项目假设均属于 0 号集合。

子集的变化由变量 setindex 表示，初值 setindex=0。每个项目划分后“项目子集”的值就是由 setindex 的值给定的。每划分一个子集后，执行 setindex++运算，为下一个子集划分做准备。

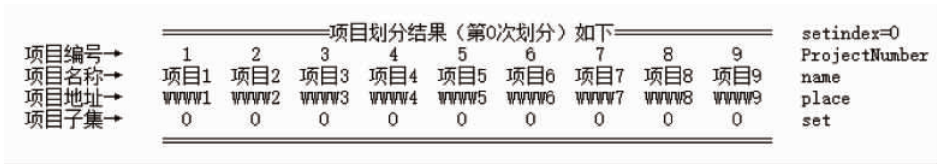

图 3.37　项目信息初态

划分子集时，队列的变化过程和项目信息空间的变化如图 3.38 到图 3.45 所示。

图 3.38　第一次划分后队列状态

图 3.39　第一次划分后项目信息状态

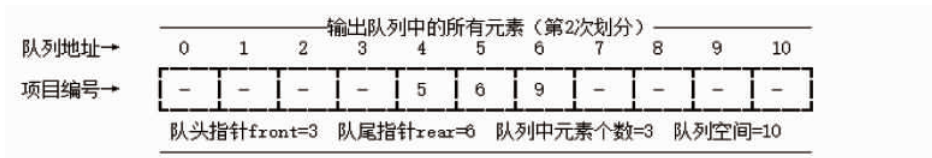

图 3.40　第二次划分后队列状态

图 3.41　第二次划分后项目信息状态

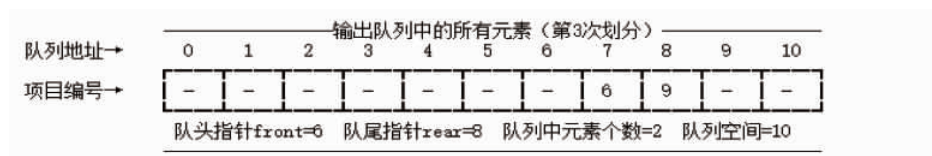

图 3.42　第三次划分后队列状态

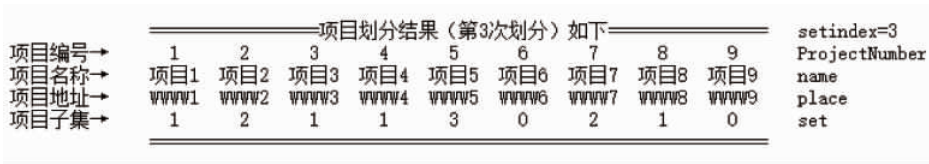

图 3.43 第三次划分后项目信息状态

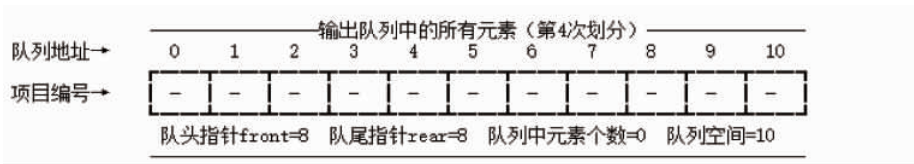

图 3.44 第四次划分后队列状态

图 3.45 第四次划分后项目信息状态

划分过程是将队列中的所有元素逐个出队一次。

每次第一个出队的项目编号作为进入新子集的第一个项目。

以后出队的元素与已进入当前子集的项目进行比较，有两种比较结果：

- 出队项目不与进入当前子集的任何项目发生冲突，则作为进入当前子集的一个项目，此项目不再进队。
- 出队项目与进入当前子集的某个项目发生冲突，则该项目不能进入当前子集，此项目重新进队，构成再次筛选的初始队列元素。

队列 Q 中的元素全部出队一次，就筛选出一个子集。由于形成某一子集的元素不再进队，队列元素在不断减少，直至队列中所有元素全部出队，子集划分过程就完成了。

子集划分结果保存在项目信息 Projects 的 set 成员域。

每次开始划分一个新子集时，第一个出队的元素不需要判断冲突关系，因为这时是一个新子集形成的开始，新子集中无任何项目，也就不存在进入项目与其中已进入项目的冲突，即直接作为所形成新子集的第一元素，直接形成 set 空间对应位置的状态值。而以后出队的元素是否属于该子集，就要根据冲突关系矩阵来比较决定。

划分子集算法 DivisionRun 如下：

```
struct PROJECT                          //项目数据元素类型的定义
{
    int  ProjectNumber;                 //项目编号
    char name[8];                       //项目名称
    char place[20];                     //项目地址
```

```
    int  set;                                   //划分后所属子集编号
};
class QueueType                                 //队列数据元素类型的定义
{
  public:
    int ProjectNumber;                          //项目编号
};
template<class PROJECT>
void Division<PROJECT>::
DivisionRun (int r[10][10], PROJECT * Projects, int ProjectCount)
{   //ProjectCount 个项目划分不冲突的子集,冲突关系在 r 数组中表示
    //PROJECT * Projects 项目信息的实例数据,
    int rearkeep;          //队尾保持指针变量,用于判断划分一个子集时队列元素是否全部出队
    int current;                                //项目编号变量
    int setindex;                               //子集编号变量
    int MaxQueueSize = ProjectCount + 1;        //定义队列大小值
    int i;
    QueueType x;
    Queue<QueueType> Q(MaxQueueSize);           //创建一个队列实例 Q
    for (i = 1; i<= ProjectCount;i++)
    {   //项目编号进队,初始化队列:项目一的项目编号为 1,项目二的项目编号为 2……
        x.ProjectNumber = Projects[i].ProjectNumber;
        Q.EnQueue(x);
    }
    setindex = 0;                               //子集编号初值 0,以后 + 1
    setindex ++;                                //子集编号从 1 开始编号
    while (! Q.IsEmpty())
    {
        rearkeep = Q.GetRearAddress();          //保留当前队尾指针 rear
        Q.DeQueue(x);                           //出队进入当前子集的第一个项目编号
        current = x.ProjectNumber;
        Projects[current].set = setindex; //第一个进入新子集的项目改变子集状态值
        while (!(Q.GetFrontAddress() == rearkeep))
        {   //当前队列中的所有项目尝试进入当前子集
            Q.DeQueue(x);                       //出队下一个项目编号
            current = x.ProjectNumber;
            for (i = 1;i<= ProjectCount;i++)
                //current 项目与已进入当前子集的所有项目比较是否发生冲突
                if (setindex == Projects[i].set && r[current][i])
                {   //current 项目与已进入当前子集的 i 号项目发生冲突
                    x.ProjectNumber = current;
                    Q.EnQueue(x);                   //current 项目重新进队
                    break;                          //终止比较冲突
                }
            if (i > ProjectCount)
            //for 循环正常退出(没有冲突)后,循环变量 i 大于循环终值 ProjectCount
                Projects[current].set = setindex;   //current 进入当前子集
        }
        setindex ++;                                //子集编号加 1,准备划分下一个子集
    }
}
```

3.10 堆栈和队列基本算法的程序实现

3.10.1 堆栈顺序存储结构程序实现

堆栈顺序存储结构程序由 3 个部分组成：

- 实例数据元素类型定义的头文件 AppData_LinearStack.h。
- 堆栈顺序存储结构模板类 Stack 定义的头文件 LinearStack_Class.h。
- 堆栈顺序存储结构主程序 LinearStack.cpp。

实例数据元素类型定义的头文件 AppData_LinearStack.h 如下：

```
#define STUDENT ElementType                          //实例数据元素句柄化

struct STUDENT                                       //实例数据元素类型的定义
{
    char number[10];
    char name[8];
    char sex[3];
    int  age;
    char place[20];
};
```

堆栈模板类 Stack 定义的头文件 LinearStack_Class.h 如下：

```
template<class StackType>
class Stack
{   //顺序存储堆栈结构模板类 Stack 的定义
public:
    Stack(int MaxStackSize = 20);                    //构造函数
    ~Stack() {delete [] element;};                   //析构函数(释放空间)
    bool IsEmpty() {return top == -1;};              //判断堆栈空
    bool IsFull() {return top >= MaxSpaceSize-1;};                              //判断堆栈满
    bool GetTop(StackType& result);                  //获取栈顶元素值,存放到 result 中
    bool Push(StackType& newvalue);                  //newvalue 值进栈
    bool Pop(StackType& result);                     //出栈值存放到 result 中
    int  GetTopAddress(){return top;};               //获取堆栈栈顶指针
    void DisplayStack();                             //显示输出堆栈中所有数据元素值(依赖应用)
    void DisplayElementStack(int i);                 //显示输出堆栈中 i 地址的数据元素值
private:                                             //堆栈数据结构定义
    int top;                                         //堆栈栈顶指针
    int MaxSpaceSize;                                //堆栈空间大小
    StackType * element;                             //堆栈数据元素存放空间
};
template<class StackType>
Stack<StackType>::
Stack(int MaxStackSize)
{   ///构造函数,堆栈数据元素存放于 element[0..MaxSpaceSize-1]
    MaxSpaceSize = MaxStackSize ;
```

```
    element = new StackType[MaxSpaceSize];
    top = -1;
}
template<class StackType>
bool Stack<StackType>::
GetTop(StackType& result)
{   //获取栈顶元素值
    if (IsEmpty()) return false;
    result = element[top];
    return true;
}
template<class StackType>
bool Stack<StackType>::
Push(StackType& newvalue)
{   //进栈
    if (IsFull()) return false;
        element[++top] = newvalue;
    return true;
}
template<class StackType>
bool Stack<StackType>::
Pop(StackType& result)
{   //出栈
    if (IsEmpty()) return false;
        result = element[top--];
    return true;
}
template<class StackType>
void Stack<StackType>::
DisplayStack()
{   //逐个输出堆栈中的数据元素(依赖应用)
    cout<<" ********** 输出堆栈中的所有元素 ********** "<<endl<<endl;
    cout<<"相对地址　学号　　姓名　　性别　年龄　住址"<<endl;
    cout<<" top→ ";
    for (int i = top; i>-1 ; i--)
    {
        cout<<i<<"    "
            <<element[i].number<<"    "
            <<element[i].name<<"    "
            <<element[i].sex<<"    "
            <<element[i].age<<" "
            <<element[i].place<<endl;
        cout<<"       ";
    }
    cout<<endl
        <<"堆栈元素个数 = "<<top+1<<" "
        <<"堆栈指针 top = "<<top<<" "
        <<"堆栈空间大小 = "<<MaxSpaceSize+1<<endl;
    cout<<endl<<endl;
}
template<class StackType>
```

```
void Stack < StackType >::
DisplayElementStack(int i)
{   //逐个输出堆栈中的数据元素
    cout <<" ********** 输出堆栈中的一个元素 ********** "<< endl << endl;
    cout <<"相对地址   学号      姓名      性别   年龄   住址"<< endl;
    if (i == top)
        cout <<" top→ ";
    else
        cout <<"       ";
    cout << i <<"     "
        << element[i].number <<"      "
        << element[i].name <<"      "
        << element[i].sex <<"      "
        << element[i].age <<"     "
        << element[i].place << endl;
        cout <<"        ";
    cout << endl << endl;
}
```

堆栈顺序存储结构主程序 LinearStack.cpp 如下：

```
#include < iostream.h >
#include < cstring >
#include < stdlib.h >
#include < conio.h >
#include "AppData_LinearStack.h"
#include "LinearStack.cpp"
int main()
{
    int  choice;
    char ok[2] = {"y"};
    int  MaxStackSize = 8;
    ElementType newvalue,result;
    //构造空堆栈
    cout <<"构造包含 MaxStackSize 个空间的空堆栈实例(对象)AppStack:"<< endl;
    Stack < ElementType > AppStack(MaxStackSize);
    //下面是堆栈的初值创建
    char number[][8] = {"1001","1002","1003","1004","1005","1006"};
    char name[][8] = {"第一","第二","第三","第四","第五","第六"};
    char sex[][8] = { "男", "男","女","男","男","女"};
    char place[][8] = {"www1","www2","www3","www4","www5","www6"};
    int  age[] = {101,102,103,104,105,106};
    for(int i = 0;i < 6;i++)
    {
        strcpy(newvalue.number, number[i]);
        strcpy(newvalue.name, name[i]);;
        strcpy(newvalue.sex, sex[i]);
        strcpy(newvalue.place, place[i]);
        newvalue.age = age[i];
        AppStack.Push(newvalue);
    }
```

```
cout <<"输入进栈元素的值:"<< endl;
strcpy(newvalue.number,"9999");
strcpy(newvalue.name,"元素");
strcpy(newvalue.sex,"中");
strcpy(newvalue.place,"武汉");
newvalue.age = 999;
while (true)
{
    cout <<" ********* 堆栈顺序存储的运算 ********** "<< endl;
    cout <<"    1 -------- 输出(不出栈)堆栈中的所有元素(栈顶在上)"<< endl;
    cout <<"    2 -------- 返回堆栈的栈顶元素(不出栈)"<< endl;
    cout <<"    3 -------- 进栈"<< endl;
    cout <<"    4 -------- 出栈"<< endl;
    cout <<"    0 -------- 退出"<< endl;
    cout <<" ********************************************* "endl;
    cout <<"请选择处理功能: "; cin >> choice;
    cout << endl;
    system("cls");
    switch(choice)
    {
    case 1:
    {   //1 -------- 输出线性表中的所有元素
        AppStack.DisplayStack();
        break;
    }
    case 2:
    {   //2 -------- 取栈顶元素
        cout <<"操作前堆栈状态: "<< endl;
        AppStack.DisplayStack();
        if (!AppStack.IsEmpty())
        {
            i = AppStack.GetTopAddress();
            AppStack.DisplayElementStack(i);
        }
        else
        {
            cout <<"ERROR 栈空,栈空,栈空,出栈失败 ERROR"<< endl;
            system("pause");
        }
        break;
    }
    case 3:
    {   //3 -------- 进栈
        cout <<"操作前堆栈状态: "<< endl;
        AppStack.DisplayStack();
        while(true)
        {
            if (AppStack.IsFull())
            {
                cout <<"ERROR 栈满,栈满,栈满,进栈失败 ERROR"<< endl;
                system("pause");
```

```
                    break;
                }
                cout<<"    学号    姓名    性别    住址    年龄"<<endl;
                cout<<"    "<<newvalue.number;
                cout<<"    "<<newvalue.name;
                cout<<"    "<<newvalue.sex;
                cout<<"    "<<newvalue.place;
                cout<<"    ";cin>>newvalue.age;
                cout<<endl;
//              cout<<"学号:";cin>>newvalue.number;
//              cout<<"姓名:";cin>>newvalue.name;
//              cout<<"性别:";cin>>newvalue.sex;
//              cout<<"年龄:";cin>>newvalue.age;
//              cout<<"位置:";cin>>newvalue.place;
                AppStack.Push(newvalue);
                cout<<"继续进栈吗:(y/n)?";cin>>ok;
                if(strcmp(ok,"y"))
                break;
            }
            cout<<"操作后堆栈状态:"<<endl;
            AppStack.DisplayStack();
            break;
        }
        case 4:
        {   //4 -------- 出栈
            cout<<"操作前堆栈状态:"<<endl;
            AppStack.DisplayStack();
            cout<<" *** 输出出栈元素 *** "<<endl<<endl;
            while(true)
            {
                if(AppStack.IsEmpty())
                {
                    cout<<"ERROR 栈空,栈空,栈空,出栈失败 ERROR"<<endl;
                    system("pause");
                    break;
                }
                AppStack.Pop(result);
                cout<<"    "
                    //<<AppStack.top+1<<"    "
                    <<result.number<<"    "
                    <<result.name<<"    "
                    <<result.sex<<"    "
                    <<result.age<<"    "
                    <<result.place<<endl;
                cout<<endl;
                cout<<"继续出栈吗:(y/n)?";cin>>ok;
                if(strcmp(ok,"y"))
                    break;
                }
                cout<<"操作后堆栈状态:"<<endl;
                AppStack.DisplayStack();
```

```
                break;
            }
            case 0:
            {
                return 0;
                break;
            }
        }
        system("pause");
        system("cls");
    }
}
```

3.10.2 队列顺序存储结构程序实现

队列顺序存储结构程序由 3 个部分组成：

- 实例数据元素类型定义的头文件 AppData_LinearQueue.h。
- 队列顺序存储结构模板类 Queue 定义的头文件 LinearQueue_Class.h。
- 队列顺序存储结构主程序 LinearQueue.cpp。

实例数据元素类型定义的头文件 AppData_LinearQueue.h 如下：

```
#define STUDENT ElementType                    //实例数据元素句柄化
struct STUDENT                                 //实例数据元素类型的定义
{
    char number[10];
    char name[8];
    char sex[3];
    int  age;
    char place[20];
};
```

队列模板类 Queue 定义的头文件 LinearQueue_Class.h 如下：

```
template<class QueueType>
class Queue
{   //队列模板类定义.队头指针指向队头元素前一个位置,队尾指针指向队尾元素位置
    //数据元素存放于 element[0..MaxSpaceSize]范围
public:
    Queue(int MaxQueueSize = 20);              //构造函数(构造循环队列)
    ~Queue() {delete [] element;};             //析构函数
    bool IsEmpty(){return front == rear;};     //判断队列空
    bool IsFull(){return (front == (rear+1) % (MaxSpaceSize+1) ? 1 : 0);};
        //判断队列满
    bool GetFront(QueueType& result);          //获取队头元素值
    bool GetRear(QueueType& result);           //获取队尾元素值
    bool EnQueue(QueueType& newvalue);         //循环队列进队
    bool DeQueue(QueueType& result);           //循环队列出队
    void DisplayQueue();                       //显示输出所有队列元素值(依赖应用)
    void DisplayElementQueue(int i);           //显示输出队列 i 地址元素的值(依赖应用)
```

```
private:
    QueueType * element;                    //队列元素空间
    int       front;                        //队头指针
    int       rear;                         //队尾指针
    int       MaxSpaceSize;                 //队列空间大小
};
template < class QueueType >
Queue < QueueType >::
Queue( int MaxQueueSize)
{   //队列构造算法
    MaxSpaceSize = MaxQueueSize;
    element = new QueueType[ MaxSpaceSize + 1];
    front = 0;
    rear = 0;
}
template < class QueueType >
bool Queue < QueueType >::
GetFront(QueueType& result)
{   //获取队头元素值
    if (IsEmpty()) return false;
    result = element[(front + 1) % (MaxSpaceSize + 1)];
    return true;
}
template < class QueueType >
bool Queue < QueueType >::
GetRear(QueueType& result)
{   //获取队尾元素值
    if (IsEmpty()) return false;
    result = element[rear];
    return true;
}
template < class QueueType >
bool Queue < QueueType >::
EnQueue(QueueType& newvalue)
{   //循环队列进队
    if (IsFull()) return false;
    rear = (rear + 1) % (MaxSpaceSize + 1);     //循环队列指针移到下一个位置
    element[rear] = newvalue;
    return true;
}
template < class QueueType >
bool Queue < QueueType >::
DeQueue(QueueType& result)
{   //循环队列出队
    if (IsEmpty()) return false;
    front = (front + 1) % (MaxSpaceSize + 1);  //循环队列指针移到下一个位置
    result = element[front];
    return true;
}
template < class QueueType >
void Queue < QueueType >::
```

```
DisplayQueue()
{   //逐个输出队列中的数据元素(学生信息),依赖于应用的算法
    cout<<" ********** 输出队列中的所有元素 ********** "<<endl;
    cout<<"相对地址   学号      姓名      性别   年龄   住址"<<endl;
    cout<<" ---------------------------------------------------- "<<endl;
    QueueType result;
    int frontkeep = front;
    int rearkeep = rear;
    int k = 0;
    while (!IsEmpty())
    {
        DeQueue(result);
        if(front<10)
            cout<<"     ";
        else
            cout<<"     ";
        cout<<front<<"     "
            <<result.number<<"      "
            <<result.name<<"      "
            <<result.sex<<"      "
            <<result.age<<"     "
            <<result.place<<endl;
    }
    front = frontkeep;
    rear = rearkeep;
    cout<<" ---------------------------------------------------- "<<endl;
    cout<<"             队头 front = "<<frontkeep<<" "
        <<"      队尾 rear = "<<rearkeep<<" "
        <<endl;
    cout<<endl<<endl;
}
```

队列顺序存储结构主程序 LinearQueue.cpp 如下：

```
#include<iostream.h>
#include<cstring>
#include<stdlib.h>
#include "AppData_LinearQueue.h"
#include "LinearQueue_Class.h"

int main()
{
    int  choice;
    char ok[2]={"y"};
    int  MaxQueueSize=15;
    ElementType newvalue,result;
    //构造包含 MaxQueueSize 个空间的空队列实例(对象)AppQueue
    Queue<ElementType> AppQueue(MaxQueueSize);
    //下面是队列的初值创建
    char number[][8]={"1001","1002","1003","1004","1005","1006"};
    char name[][8]={"第一","第二","第三","第四","第五","第六"};
```

```
char sex[][8] = { "男", "男","女","男","男","女"};
char place[][8] = {"www1","www2","www3","www4","www5","www6"};
int age[] = {101,102,103,104,105,106};
for(int i = 0;i < 6;i++)
{
    strcpy(newvalue.number, number[i]);
    strcpy(newvalue.name, name[i]);
    strcpy(newvalue.sex, sex[i]);
    strcpy(newvalue.place, place[i]);
    newvalue.age = age[i];
    AppQueue.EnQueue(newvalue);
}
cout <<"输入进队元素的值:"<< endl;
strcpy(newvalue.number,"9999");
strcpy(newvalue.name,"人类");
strcpy(newvalue.sex,"中");
strcpy(newvalue.place,"地球");
newvalue.age = 900;
while (true)
{
    cout <<" ********* 顺序存储队列的运算 ********** "<< endl;
    cout <<"    0 -------- 退出"<< endl;
    cout <<"    1 -------- 输出(不出队)队列中的所有元素"<< endl;
    cout <<"    2 -------- 返回队列的队头、队尾元素"<< endl;
    cout <<"    3 -------- 进队"<< endl;
    cout <<"    4 -------- 出队"<< endl;
    cout <<" ****************************************** "<< endl;
    cout <<"请选择处理功能: "; cin >> choice;
    cout << endl;
    system("cls");
    cout <<"操作前队列状态: "<< endl;
    AppQueue.DisplayQueue();
    switch(choice)
    {
    case 0:
    {
        return 0;
        break;
    }
    case 1:
    {   //1 -------- 输出线性表中的所有元素
        AppQueue.DisplayQueue();
        break;
    }
    case 2:
    {   //2 -------- 取队头、队尾元素
        if (!AppQueue.IsEmpty())
        {
            AppQueue.GetFront(result);
            cout <<"    学号    姓名    性别    年龄    住址"<< endl;
            cout <<"Front:"
```

```
                    //<< AppQueue.top + 1 <<"     "
                    << result.number <<"      "
                    << result.name <<"     "
                    << result.sex <<"     "
                    << result.age <<"     "
                    << result.place << endl;
                cout << endl;
                AppQueue.GetRear(result);
                cout <<" Rear:"
                    //<< AppQueue.top + 1 <<"     "
                    << result.number <<"      "
                    << result.name <<"     "
                    << result.sex <<"     "
                    << result.age <<"     "
                    << result.place << endl;
                cout << endl;
            }
            else
            {
                cout <<"ERROR 队空,队空,队空,失败 ERROR"<< endl;
                system("pause");
            }
            break;
        }
        case 3:
        {   //3 -------- 进队
            while(true)
            {
                if (AppQueue.IsFull())
                {
                    cout <<"ERROR 队满,队满,队满,进队失败 ERROR"<< endl;
                    system("pause");
                    break;
                }
                cout <<"    学号    姓名    性别    住址    年龄"<< endl;
                cout <<"    "<< newvalue.number;
                cout <<"    "<< newvalue.name;
                cout <<"    "<< newvalue.sex;
                cout <<"    "<< newvalue.place;
                cout <<"    "<<(newvalue.age++);
                cout << endl;
//              cout <<"学号:"; cin >> newvalue.number;
//              cout <<"姓名:"; cin >> newvalue.name;
//              cout <<"性别:"; cin >> newvalue.sex;
//              cout <<"年龄:"; cin >> newvalue.age;
//              cout <<"位置:"; cin >> newvalue.place;
                AppQueue.EnQueue(newvalue);
                cout <<"继续进队吗:(y/n)?";
                cin >> ok;
                if (strcmp(ok,"y") )
                break;
            }
```

```
            break;
        }
        case 4:
        {   //4 --------- 出队
            cout <<" ***** 输出出队元素 **** "<< endl << endl;
            while(true)
            {
                if (AppQueue.IsEmpty())
                {
                    cout <<"ERROR 队空,队空,队空,出队失败 ERROR"<< endl;
                    system("pause");
                    break;
                }
                AppQueue.DeQueue(result);
                cout <<"      "
                    << result.number <<"      "
                    << result.name <<"     "
                    << result.sex <<"     "
                    << result.age <<"      "
                    << result.place << endl;
                cout << endl;
                cout <<"继续出队吗: (y/n)?";
                cin >> ok;
                if (strcmp(ok,"y"))
                    break;
            }
            break;
        }
        }
        cout <<"操作后队列状态: "<< endl;
        AppQueue.DisplayQueue();
        system("pause");
        system("cls");
    }
}
```

习题 3

一、单选题

1. 设栈的输入序列是 1,2,3,4,则(　　)是不可能输出的序列。

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

2. 用一个大小为 6 的数组实现循环队列,rear 指向队尾元素,front 指向队头元素的前一个位置,且 rear 和 front 的值分别为 0 和 3。从队列中出队一个元素,再进队两个元素后,rear 和 front 的值分别为(　　)。

A. 1 和 5　　B. 2 和 4　　C. 4 和 2　　D. 5 和 1

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

该是(　　)。

A. 6　　B. 4　　C. 3　　D. 2

4. 一般情况下,将递归算法转换成等价的非递增归算法应该设置(　　)。

A. 堆栈　　B. 队列　　C. 堆栈或队列　　D. 数组

5. 假定一个顺序循环队列存储于数组 $a[n]$中,其队头和队尾指针分别用 front 和 rear 表示,则判断队满的条件是(　　)。

A. (rear－1)%n==front　　B. rear==(front－1)%n

C. (rear＋1)%n==front　　D. rear==(front＋1)%n

6. 链栈与顺序栈相比(　　)。

A. 插入操作更加不方便　　B. 通常不会出现栈满的情况

C. 不会出现栈空的情况　　D. 删除操作更加不方便

单选题答案:

1. D　2. B　3. C　4. A　5. C　6. B

二、填空题

1. 用数组 Q(其下标为 $O..n-1$,共有 n 个元素)表示一个循环队列,front 为当前队头元素的前一位置,rear 为队尾元素的位置。假定队列元素个数总小于 n,求队列中元素个数的公式是_____。

2. 一个循环队列存于 $a[n]$中,假定队头和队尾指针分别为 front 和 rear,front 为当前队头元素的前一位置,rear 为队尾元素的位置。则判断队空的条件为_____,判断队满的条件为_____。

3. 栈是特殊的线性表,其特殊性在于_____。

4. 栈又称为_____表,队列又称为_____表。

填空题答案:

1. (rear－front＋n)%n

2. front==rear,(rear＋1)%n==front

3. 只能在栈顶插入或删除元素

4. 先进后出,先进先出

三、简答题和算法题

1. 对于一个栈,给出输入项为 A,B,C,D。如果输入序列为 A,B,C,D,试给出全部可能的输出序列。试说明哪些输出序列是可能的,哪些输出序列是不可能的。

2. 试用栈实现链表倒排的算法。

3. 有一有序的链表(从小到大),试利用栈筛选出结点值大于给定值 V_0 的所有结点至栈中,最后输出栈中元素的名次(要求同值同名,名次不空缺)。

4. 利用两个栈 s1 和 s2 模拟一个队列,并写出队列空、入队和出队的算法。

5. 有两个栈共享空间 V(1:m),分别写出两个栈 s1 和 s2 的压入和弹出运算的算法。

6. 试说明栈、队列和线性表的异同点。