```
        length--;                                    //长度减 1
        if (capacity>initcap && length<=1.0*capacity/4)
            recap(capacity/2);                       //满足缩容条件则容量减半
        return true;
    }
    void DispList() {                                //输出顺序表 L 中的所有元素
        for (int i=0;i<length;i++)                   //遍历顺序表中的各元素值
            cout << data[i] << " ";
        cout << endl;
    }
private:
    void recap(int newcap) {                         //改变顺序表的容量为 newcap
        if (newcap<=0) return;
        T *olddata=data;
        data=new T[newcap];                          //分配新空间
        capacity=newcap;                             //更新容量
        for(int i=0;i<length;i++)                    //复制
            data[i]=olddata[i];
        delete [] olddata;                           //释放旧空间
    }
};
int main() {
    SqList<int> L;                                   //建立元素类型为 int 的顺序表对象 L
    printf("\n");
    printf("  ==========顺序表====================\n");
    printf("  建立空表 L,其容量=%d\n",L.capacity);
    int a[]={1,2,3,4,5,6};
    int n=sizeof(a)/sizeof(a[0]);
    printf("  1-6 创建 L\n");
    L.CreateList(a,n);
    printf("  L[容量=%d,长度=%d]: ",L.capacity,L.length); L.DispList();
    printf("  插入 6-10\n");
    for (int i=6;i<=10;i++)
        L.Add(i);
    printf("  L[容量=%d,长度=%d]: ",L.capacity,L.length); L.DispList();
    int e;
    L.GetElem(2,e);
    printf("  序号为 2 的元素=%d\n",e);
    printf("  设置序号为 2 的元素为 20\n");
    L.SetElem(2,20);
    printf("  L[容量=%d,长度=%d]: ",L.capacity,L.length); L.DispList();
    int x=6;
    printf("  第一个值为%d 的元素序号=%d\n",x,L.GetNo(x));
    n=L.length;
    for (int i=0;i<n-2;i++) {
        printf("  删除首元素\n");
        L.Delete(0);
        printf("  L[容量=%d,长度=%d]: ",L.capacity,L.length); L.DispList();
    }
    return 0;
}
```

上述程序的执行结果如图 2.4 所示。

```
============顺序表====================
建立空表L, 其容量=5
1-6创建L
L[容量=10,长度=6]: 1 2 3 4 5 6
插入6-10
L[容量=20,长度=11]: 1 2 3 4 5 6 6 7 8 9 10
序号为2的元素=3
设置序号为2的元素为20
L[容量=20,长度=11]: 1 2 20 4 5 6 6 7 8 9 10
第一个值为6的元素序号=5
删除首元素
L[容量=20,长度=10]: 2 20 4 5 6 6 7 8 9 10
删除首元素
L[容量=20,长度=9]: 20 4 5 6 6 7 8 9 10
删除首元素
L[容量=20,长度=8]: 4 5 6 6 7 8 9 10
删除首元素
L[容量=20,长度=7]: 5 6 6 7 8 9 10
删除首元素
L[容量=20,长度=6]: 6 6 7 8 9 10
删除首元素
L[容量=10,长度=5]: 6 7 8 9 10
删除首元素
L[容量=10,长度=4]: 7 8 9 10
删除首元素
L[容量=10,长度=3]: 8 9 10
删除首元素
L[容量=5,长度=2]: 9 10
```

图 2.4　第 2 章基础实验题 1 的执行结果

2. 解：单链表的基本运算算法的设计原理参见《教程》中的 2.3.2 节。包含单链表基本运算算法类 LinkList 以及测试主程序的 Exp1-2.cpp 文件如下：

```
#include <iostream>
using namespace std;
template <typename T>
struct LinkNode {                              //单链表结点类型
    T data;                                    //存放数据元素
    LinkNode<T> *next;                         //指向下一个结点的域
    LinkNode():next(NULL) {}                   //构造函数
    LinkNode(T d):data(d),next(NULL) {}        //重载构造函数
};
template <typename T>
class LinkList {                               //单链表类
public:
    LinkNode<T> *head;                         //单链表的头结点
    LinkList() {                               //构造函数,创建一个空单链表
        head=new LinkNode<T>();
    }
    ~LinkList() {                              //析构函数,销毁单链表
        LinkNode<T> *pre, *p;
        pre=head;p=pre->next;
        while (p!=NULL) {                      //用 p 遍历结点并释放其前驱结点
            delete pre;                        //释放 pre 结点
            pre=p; p=p->next;                  //pre、p 同步后移一个结点
        }
        delete pre;                            //p 为空时 pre 指向尾结点,此时释放尾结点
    }
    void CreateListF(T a[],int n) {            //用头插法建立单链表
        for (int i=0;i<n;i++) {                //循环建立数据结点
            LinkNode<T> *s=new LinkNode<T>(a[i]);
            s->next=head->next;                //将结点 s 插入 head 结点之后
            head->next=s;
        }
```

```
}
void CreateListR(T a[],int n) {                    //用尾插法建立单链表
    LinkNode<T> *s,*r;
    r=head;                                        //r 始终指向尾结点,开始时指向头结点
    for (int i=0;i<n;i++) {                        //循环建立数据结点
        s=new LinkNode<T>(a[i]);                   //创建数据结点 s
        r->next=s;                                 //将结点 s 插入结点 r 之后
        r=s;
    }
    r->next=NULL;                                  //将尾结点的 next 域置为 NULL
}
void Add(T e) {                                    //在单链表的末尾添加一个值为 e 的结点
    LinkNode<T> *s=new LinkNode<T>(e);
    LinkNode<T> *p=head;
    while (p->next!=NULL)                          //查找尾结点 p
        p=p->next;
    p->next=s;                                     //在尾结点之后插入结点 s
}
int Getlength() {                                  //求单链表中数据结点的个数
    LinkNode<T> *p=head;
    int cnt=0;
    while (p->next!=NULL) {                        //找到尾结点为止
        cnt++;
        p=p->next;
    }
    return cnt;
}
bool GetElem(int i,T &e) {                         //求单链表中序号为 i 的结点的值
    if (i<0) return false;                         //参数 i 错误返回 false
    LinkNode<T> *p=geti(i);                        //查找序号为 i 的结点 p
    if (p!=NULL) {                                 //找到了序号为 i 的结点
        e=p->data;
        return true;                               //成功找到返回 true
    }
    else return false;                             //没有找到序号为 i 的结点返回 false
}
bool SetElem(int i,T e) {                          //设置序号为 i 的结点的值
    if (i<0) return false;                         //参数 i 错误返回 false
    LinkNode<T> *p=geti(i);                        //查找序号为 i 的结点 p
    if (p!=NULL) {                                 //找到了序号为 i 的结点
        p->data=e;
        return true;
    }
    else                                           //没有找到序号为 i 的结点
        return false;                              //参数 i 错误返回 false
}
int GetNo(T e) {                                   //查找第一个为 e 的元素位置的序号
    int j=0;
    LinkNode<T> *p=head->next;
    while (p!=NULL && p->data!=e) {
        j++;                                       //查找元素 e
        p=p->next;
    }
```

```
        if (p==NULL) return -1;                    //未找到时返回-1
        else return j;                             //找到后返回其序号
    }
    bool Insert(int i,T e) {                       //在单链表中序号为 i 的位置插入值为 e 的结点
        if (i<0) return false;                     //参数 i 错误返回 false
        LinkNode<T> *s=new LinkNode<T>(e);
        LinkNode<T> *p=geti(i-1);                  //查找序号为 i-1 的结点 p
        if (p!=NULL) {                             //找到了序号为 i-1 的结点
            s->next=p->next;                       //在 p 结点的后面插入 s 结点
            p->next=s;
            return true;                           //插入成功返回 true
        }
        else                                       //没有找到序号为 i-1 的结点
            return false;                          //参数 i 错误返回 false
    }
    bool Delete(int i) {                           //在单链表中删除序号为 i 的位置的结点
        if (i<0) return false;                     //参数 i 错误返回 false
        LinkNode<T> *p=geti(i-1);                  //查找序号为 i-1 的结点 p
        if (p!=NULL) {                             //找到了序号为 i-1 的结点
            LinkNode<T> *q=p->next;                //q 指向序号为 i 的结点
            if (q!=NULL) {                         //存在序号为 i 的结点时删除它
                p->next=q->next;                   //删除 p 结点的后继结点
                delete q;                          //释放空间
                return true;                       //删除成功返回 true
            }
            else                                   //没有找到序号为 i 的结点
                return false;                      //参数 i 错误返回 false
        }
        else                                       //没有找到序号为 i-1 的结点
            return false;                          //参数 i 错误返回 false
    }
    void DispList() {                              //输出单链表中的所有结点值
        LinkNode<T> *p;
        p=head->next;                              //p 指向开始结点
        while (p!=NULL) {                          //p 不为 NULL,输出 p 结点的 data 域
            cout << p->data << " ";
            p=p->next;                             //p 移向下一个结点
        }
        cout << endl;
    }
private:
    //*********************************************
    //序号 i 的正确范围为-1≤i<n,超出范围返回 NULL
    //i=-1 时返回头结点 head
    //i≥0 并且 i<n 时返回序号为 i 的结点
    //*********************************************
    LinkNode<T> *geti(int i) {                     //返回序号为 i 的结点
        if (i<-1) return NULL;                     //i<-1 时返回 NULL
        LinkNode<T> *p=head;                       //首先 p 指向头结点
        int j=-1;                                  //j 置为-1(可以认为头结点的序号为-1)
        while (j<i && p!=NULL) {                   //指针 p 移动 i+1 个结点
            j++;
            p=p->next;
```

```
        }
        return p;                                    //返回 p
    }
};
int main() {
    LinkList<int> L;                                 //建立元素类型为 int 的单链表对象 L
    printf("\n");
    printf("  ==========单链表====================\n");
    printf("  建立空表 L,长度=%d\n",L.Getlength());
    int a[]={1,2,3,4,5,6};
    int n=sizeof(a)/sizeof(a[0]);
    printf("  1-6 创建 L\n");
    L.CreateListR(a,n);
    printf("  L[长度=%d]: ",L.Getlength()); L.DispList();
    printf("  添加 6-7\n");
    L.Add(6); L.Add(7);
    printf("  在序号 1 处插入 10\n");
    L.Insert(1,10);
    printf("  L[长度=%d]: ",L.Getlength()); L.DispList();
    int e;
    L.GetElem(2,e);
    printf("  序号为 2 的元素=%d\n",e);
    printf("  设置序号为 2 的元素为 20\n");
    L.SetElem(2,20);
    printf("  L[长度=%d]: ",L.Getlength()); L.DispList();
    int x=6;
    printf("  第一个值为%d 的元素序号=%d\n",x,L.GetNo(x));
    printf("  删除首元素\n");
    L.Delete(0);
    printf("  L[长度=%d]: ",L.Getlength()); L.DispList();
    printf("  删除序号为 5 的元素\n");
    L.Delete(5);
    printf("  L[长度=%d]: ",L.Getlength()); L.DispList();
    return 0;
}
```

上述程序的执行结果如图 2.5 所示。

```
==========单链表====================
建立空表L, 长度=0
1-6创建L
L[长度=6]: 1 2 3 4 5 6
添加6-7
在序号1处插入10
L[长度=9]: 1 10 2 3 4 5 6 6 7
序号为2的元素=2
设置序号为2的元素为20
L[长度=9]: 1 10 20 3 4 5 6 6 7
第一个值为6的元素序号=6
删除首元素
L[长度=8]: 10 20 3 4 5 6 6 7
删除序号为5的元素
L[长度=7]: 10 20 3 4 5 6 7
```

图 2.5　第 2 章基础实验题 2 的执行结果

3. 解：双链表的基本运算算法的设计原理参见《教程》中的 2.3.4 节。包含双链表基本运算算法类 DLinkList 以及测试主程序的 Exp1-3.cpp 文件如下：

```
#include <iostream>
using namespace std;
template <typename T>
struct DLinkNode {                                  //双链表结点类型
    T data;                                         //存放数据元素
    DLinkNode<T> *next;                             //指向后继结点的指针
    DLinkNode<T> *prior;                            //指向前驱结点的指针
    DLinkNode():next(NULL),prior(NULL) {}           //构造函数
    DLinkNode(T d):data(d),next(NULL),prior(NULL) {} //重载构造函数
};
template <typename T>
class DLinkList {                                   //双链表类
public:
    DLinkNode<T> *dhead;                            //双链表的头结点
    DLinkList() {                                   //构造函数,创建一个空双链表
        dhead=new DLinkNode<T>();
    }
    ~DLinkList() {                                  //析构函数,销毁双链表
        DLinkNode<T> *pre, *p;
        pre=dhead;p=pre->next;
        while (p!=NULL) {                           //用 p 遍历结点并释放其前驱结点
            delete pre;                             //释放 pre 结点
            pre=p; p=p->next;                       //pre、p 同步后移一个结点
        }
        delete pre;                                 //p 为空时 pre 指向尾结点,此时释放尾结点
    }
    void CreateListF(T a[],int n) {                 //用头插法建立双链表
        DLinkNode<T> *s;
        for (int i=0;i<n;i++) {                     //循环建立数据结点
            s=new DLinkNode<T>(a[i]);               //创建数据结点 s
            s->next=dhead->next;                    //修改 s 结点的 next 成员
            if (dhead->next!=NULL)                  //修改头结点的非空后继结点的 prior
                dhead->next->prior=s;
            dhead->next=s;                          //修改头结点的 next 域
            s->prior=dhead;                         //修改 s 结点的 prior 域
        }
    }
    void CreateListR(T a[],int n) {                 //用尾插法建立双链表
        DLinkNode<T> *s, *r;
        r=dhead;                                    //r 始终指向尾结点,开始时指向头结点
        for (int i=0;i<n;i++) {                     //循环建立数据结点
            s=new DLinkNode<T>(a[i]);               //创建数据结点 s
            r->next=s;                              //将 s 结点插入 r 结点之后
            s->prior=r;
            r=s;
        }
        r->next=NULL;                               //将尾结点的 next 域置为 NULL
    }
    void Add(T e) {                                 //在双链表的末尾添加一个值为 e 的结点
        DLinkNode<T> *s=new DLinkNode<T>(e);        //新建结点 s
        DLinkNode<T> *p=dhead;
        while (p->next!=NULL)                       //查找尾结点 p
            p=p->next;
```

```
    p->next=s;                                  //在尾结点之后插入结点 s
    s->prior=p;
}
int Getlength() {                               //求双链表中数据结点的个数
    DLinkNode<T> *p=dhead;
    int cnt=0;
    while (p->next!=NULL) {                     //找到尾结点为止
        cnt++;
        p=p->next;
    }
    return cnt;
}
bool GetElem(int i,T &e) {                      //求双链表中序号为 i 的结点的值
    if (i<0) return false;                      //参数 i 错误返回 false
    DLinkNode<T> *p=geti(i);
    if (p!=NULL) {                              //找到序号为 i 的结点
        e=p->data;
        return true;                            //成功找到返回 true
    }
    else return false;                          //没有找到序号为 i 的结点返回 false
}
bool SetElem(int i,T e) {                       //设置序号为 i 的结点的值
    if (i<0) return false;                      //参数 i 错误返回 false
    DLinkNode<T> *p=geti(i);
    if (p!=NULL) {                              //找到序号为 i 的结点
        p->data=e;
        return true;
    }
    else                                        //没有找到序号为 i 的结点
        return false;                           //参数 i 错误返回 false
}
int GetNo(T e) {                                //查找第一个为 e 的元素的序号
    int j=0;                                    //j 置为 0,p 指向首结点
    DLinkNode<T> *p=dhead->next;
    while (p!=NULL && p->data!=e) {
        j++;                                    //查找元素 e
        p=p->next;
    }
    if (p==NULL) return -1;                     //未找到时返回-1
    else return j;                              //找到后返回其序号
}
bool Insert(int i,T e) {                        //在双链表中序号为 i 的位置插入值为 e 的结点
    if (i<0) return false;                      //参数 i 错误返回 false
    DLinkNode<T> *s=new DLinkNode<T>(e);    //建立新结点 s
    DLinkNode<T> *p=geti(i-1);                  //查找序号为 i-1 的结点 p
    if (p!=NULL) {                              //找到了序号为 i-1 的结点
        s->next=p->next;                        //修改 s 结点的 next 域
        if (p->next!=NULL)                      //修改 p 结点的非空后继结点的 prior 域
            p->next->prior=s;
        p->next=s;                              //修改 p 结点的 next 域
        s->prior=p;                             //修改 s 结点的 prior 域
        return true;                            //插入成功返回 true
    }
```

```
        else                                          //没有找到序号为 i-1 的结点
            return false;                             //参数 i 错误返回 false
    }
    bool Delete(int i) {                              //在双链表中删除序号为 i 的位置的结点
        if (i<0) return false;                        //参数 i 错误返回 false
        DLinkNode<T> *p=geti(i);                      //查找序号为 i 的结点
        if (p!=NULL) {                                //找到了序号为 i 的结点 p
            p->prior->next=p->next;                   //修改 p 结点的前驱结点的 next 域
            if (p->next!=NULL)                        //修改 p 结点的非空后继结点的 prior 域
                p->next->prior=p->prior;
            delete p;                                 //释放空间
            return true;                              //删除成功返回 true
        }
        else                                          //没有找到序号为 i-1 的结点
            return false;                             //参数 i 错误返回 false
    }
    void DispList() {                                 //输出双链表中的所有结点值
        DLinkNode<T> *p;
        p=dhead->next;                                //p 指向开始结点
        while (p!=NULL) {                             //p 不为 NULL,输出 p 结点的 data 域
            cout << p->data << " ";
            p=p->next;                                //p 移向下一个结点
        }
        cout << endl;
    }
private:
    //**********************************************
    //序号 i 的正确范围为-1≤i<n,超出范围返回 NULL
    //i=-1 时返回头结点 head
    //i≥0 并且 i<n 时返回序号为 i 的结点
    //**********************************************
    DLinkNode<T> *geti(int i) {                       //返回序号为 i 的结点
        if (i<-1) return NULL;                        //i<-1 返回 NULL
        DLinkNode<T> *p=dhead;                        //首先 p 指向头结点
        int j=-1;                                     //j 置为-1(可以认为头结点的序号为-1)
        while (j<i && p!=NULL) {                      //指针 p 移动 i+1 个结点
            j++;
            p=p->next;
        }
        return p;                                     //返回 p
    }
};
int main() {
    DLinkList<int> L;                                 //建立元素类型为 int 的双链表对象 L
    printf("\n");
    printf("    ============双链表====================\n");
    printf("    建立空表 L,长度=%d\n",L.Getlength());
    int a[]={1,2,3,4,5,6};
    int n=sizeof(a)/sizeof(a[0]);
    printf("    1-6 创建 L\n");
    L.CreateListR(a,n);
    printf("    L[长度=%d]: ",L.Getlength()); L.DispList();
    printf("    添加 6-7\n");
```

```
    L.Add(6); L.Add(7);
    printf("  在序号 1 处插入 10\n");
    L.Insert(1,10);
    printf("  L[长度=%d]: ",L.Getlength()); L.DispList();
    int e;
    L.GetElem(2,e);
    printf("  序号为 2 的元素=%d\n",e);
    printf("  设置序号为 2 的元素为 20\n");
    L.SetElem(2,20);
    printf("  L[长度=%d]: ",L.Getlength()); L.DispList();
    int x=6;
    printf("  第一个值为%d 的元素序号=%d\n",x,L.GetNo(x));
    printf("  删除首元素\n");
    L.Delete(0);
    printf("  L[长度=%d]: ",L.Getlength()); L.DispList();
    printf("  删除序号为 5 的元素\n");
    L.Delete(5);
    printf("  L[长度=%d]: ",L.Getlength()); L.DispList();
    return 0;
}
```

上述程序的执行结果如图 2.6 所示。

```
=============双链表====================
建立空表L, 长度=0
1-6创建L
L[长度=6]: 1 2 3 4 5 6
添加6-7
在序号1处插入10
L[长度=9]: 1 10 2 3 4 5 6 6 7
序号为2的元素=2
设置序号为2的元素为20
L[长度=9]: 1 10 20 3 4 5 6 6 7
第一个值为6的元素序号=6
删除首元素
L[长度=8]: 10 20 3 4 5 6 6 7
删除序号为5的元素
L[长度=7]: 10 20 3 4 5 6 7
```

图 2.6　第 2 章基础实验题 3 的执行结果

4. 解：循环单链表的基本运算算法的设计原理参见《教程》中的 2.3.6 节。包含循环单链表基本运算算法类 CLinkList 以及测试主程序的 Exp1-4.cpp 文件如下：

```
#include <iostream>
using namespace std;
template <typename T>
struct LinkNode {                          //循环单链表结点类型
    T data;                                //存放数据元素
    LinkNode<T> *next;                     //指向下一个结点的域
    LinkNode():next(NULL) {}               //构造函数
    LinkNode(T d):data(d),next(NULL) {}    //重载构造函数
};
template <typename T>
class CLinkList {                          //循环单链表类
public:
    LinkNode<T> *head;                     //循环单链表的头结点
    CLinkList() {                          //构造函数,创建一个空循环单链表
        head=new LinkNode<T>();
```

```
    head->next=head;                          //构成循环的空链表
}
~CLinkList() {                                //析构函数,销毁循环单链表
    LinkNode<T> *pre, *p;
    pre=head;p=pre->next;
    while (p!=head) {                         //用 p 遍历结点并释放其前驱结点
        delete pre;                           //释放 pre 结点
        pre=p; p=p->next;                     //pre、p 同步后移一个结点
    }
    delete pre;                               //p 等于 head 时 pre 指向尾结点,此时释放尾结点
}
void CreateListF(T a[],int n) {               //用头插法建立循环单链表
    for (int i=0;i<n;i++) {                   //循环建立数据结点
        LinkNode<T> * s=new LinkNode<T>(a[i]);
        s->next=head->next;                   //将结点 s 插入 head 结点之后
        head->next=s;
    }
}
void CreateListR(T a[],int n) {               //用尾插法建立循环单链表
    LinkNode<T> *s, *r;
    r=head;                                   //r 始终指向尾结点,开始时指向头结点
    for (int i=0;i<n;i++) {                   //循环建立数据结点
        s=new LinkNode<T>(a[i]);              //创建数据结点 s
        r->next=s;                            //将结点 s 插入结点 r 之后
        r=s;
    }
    r->next=head;                             //将尾结点的 next 域置为 head
}
void Add(T e) {                               //在循环单链表的末尾添加一个值为 e 的结点
    LinkNode<T> *s=new LinkNode<T>(e);        //新建结点 s
    LinkNode<T> *p=head;
    while (p->next!=head)                     //查找尾结点 p
        p=p->next;
    s->next=p->next;                          //在结点 p 之后插入结点 s
    p->next=s;
}
int Getlength() {                             //求循环单链表中数据结点的个数
    LinkNode<T> *p=head;
    int cnt=0;
    while (p->next!=head) {                   //查找到尾结点为止
        cnt++;
        p=p->next;
    }
    return cnt;
}
bool GetElem(int i,T &e) {                    //求循环单链表中序号为 i 的结点的值
    if (i<0) return false;                    //参数 i 错误返回 false
    LinkNode<T> *p=geti(i);                   //查找序号为 i 的结点
    if (p!=head) {                            //找到了序号为 i 的结点 p
        e=p->data;
        return true;                          //成功找到返回 true
    }
    else return false;                        //没有找到序号为 i 的结点返回 false
```

```
}
bool SetElem(int i,T e) {                    //设置序号为 i 的结点的值
    if (i<0) return false;                   //参数 i 错误返回 false
    LinkNode<T> *p=geti(i);                  //查找序号为 i 的结点
    if (p!=NULL) {                           //找到了序号为 i 的结点 p
        p->data=e;
        return true;
    }
    else                                     //没有找到序号为 i 的结点
        return false;                        //参数 i 错误返回 false
}
int GetNo(T e) {                             //查找第一个为 e 的元素的序号
    int j=0;
    LinkNode<T> *p=head->next;
    while (p!=head && p->data!=e) {
        j++;                                 //查找元素 e
        p=p->next;
    }
    if (p==head) return -1;                  //未找到时返回-1
    else return j;                           //找到后返回其序号
}
bool Insert(int i,T e) {                     //在循环单链表中序号为 i 的位置插入值为 e 的结点
    if (i<0) return false;                   //参数 i 错误返回 false
    LinkNode<T> *s=new LinkNode<T>(e);       //建立新结点 s
    LinkNode<T> *p=geti(i-1);                //查找序号为 i-1 的结点 p
    if (p!=NULL) {                           //找到了序号为 i-1 的结点
        s->next=p->next;                     //在 p 结点的后面插入 s 结点
        p->next=s;
        return true;                         //插入成功返回 true
    }
    else                                     //没有找到序号为 i-1 的结点
        return false;                        //参数 i 错误返回 false
}
bool Delete(int i) {                         //在循环单链表中删除序号为 i 的位置的结点
    if (i<0) return false;                   //参数 i 错误返回 false
    LinkNode<T> *p=geti(i-1);                //查找序号为 i-1 的结点 p
    if (p!=NULL) {                           //找到了序号为 i-1 的结点
        LinkNode<T> *q=p->next;              //q 指向序号为 i 的结点
        if (q!=NULL) {                       //存在序号为 i 的结点时删除它
            p->next=q->next;                 //删除 p 结点的后继结点
            delete q;                        //释放空间
            return true;                     //删除成功返回 true
        }
        else                                 //没有序号为 i 的结点
            return false;                    //参数 i 错误返回 false
    }
    else                                     //没有找到序号为 i-1 的结点
        return false;                        //参数 i 错误返回 false
}
void DispList() {                            //输出循环单链表中的所有结点值
    LinkNode<T> *p;
    p=head->next;                            //p 指向开始结点
    while (p!=head) {                        //p 不为 head,输出 p 结点的 data 域
```

```
            cout << p->data << " ";
            p=p->next;                          //p 移向下一个结点
        }
        cout << endl;
    }
private:
    // ***********************************************
    //序号 i 的正确范围为-1≤i<n,超出范围返回 NULL
    //i=-1 时返回头结点 head
    //i≥0 并且 i<n 时返回序号为 i 的结点
    // ***********************************************
    LinkNode<T> *geti(int i) {                  //返回序号为 i 的结点
        if (i<-1) return NULL;                  //i<-1 时返回 NULL
        if (i==-1) return head;                 //i=-1 时返回头结点
        LinkNode<T> *p=head->next;              //首先 p 指向首结点
        int j=0;                                //j 置为 0
        while (j<i && p!=head) {                //p 移动 i 个结点
            j++;
            p=p->next;
        }
        if (p==head) return NULL;               //没有找到序号为 i 的结点返回 NULL
        else return p;
    }
};
int main() {
    CLinkList<int> L;                           //建立元素类型为 int 的循环单链表对象 L
    printf("\n");
    printf("  ==========循环单链表====================\n");
    printf("  建立空表 L,长度=%d\n",L.Getlength());
    int a[]={1,2,3,4,5,6};
    int n=sizeof(a)/sizeof(a[0]);
    printf("  1-6 创建 L\n");
    L.CreateListR(a,n);
    printf("  L[长度=%d]: ",L.Getlength()); L.DispList();
    printf("  添加 6-7\n");
    L.Add(6); L.Add(7);
    printf("  在序号 1 处插入 10\n");
    L.Insert(1,10);
    printf("  L[长度=%d]: ",L.Getlength()); L.DispList();
    int e;
    L.GetElem(2,e);
    printf("  序号为 2 的元素=%d\n",e);
    printf("  设置序号为 2 的元素为 20\n");
    L.SetElem(2,20);
    printf("  L[长度=%d]: ",L.Getlength()); L.DispList();
    int x=6;
    printf("  第一个值为%d 的元素序号=%d\n",x,L.GetNo(x));
    printf("  删除首元素\n");
    L.Delete(0);
    printf("  L[长度=%d]: ",L.Getlength()); L.DispList();
    printf("  删除序号为 5 的元素\n");
    L.Delete(5);
    printf("  L[长度=%d]: ",L.Getlength()); L.DispList();
    return 0;
}
```

上述程序的执行结果如图 2.7 所示。

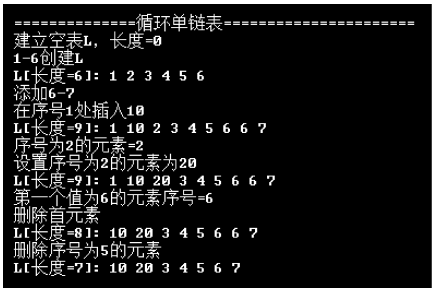


图 2.7　第 2 章基础实验题 4 的执行结果

5. 解：循环双链表的基本运算算法的设计原理参见《教程》中的 2.3.6 节。包含循环双链表基本运算算法类 CDLinkList 以及测试主程序的 Exp1-5.cpp 文件如下：

```
#include <iostream>
using namespace std;
template <typename T>
struct DLinkNode {                              //循环双链表结点类型
    T data;                                     //存放数据元素
    DLinkNode<T> *next;                         //指向后继结点的指针
    DLinkNode<T> *prior;                        //指向前驱结点的指针
    DLinkNode():next(NULL),prior(NULL) {}       //构造函数
    DLinkNode(T d):data(d),next(NULL),prior(NULL) {}        //重载构造函数
};
template <typename T>
class CDLinkList {                              //循环双链表类
public:
    DLinkNode<T> *dhead;                        //循环双链表的头结点
    CDLinkList() {                              //构造函数,创建一个空循环双链表
        dhead=new DLinkNode<T>();
        dhead->next=dhead;                      //构成循环的空链表
        dhead->prior=dhead;
    }
    ~CDLinkList() {                             //析构函数,销毁循环双链表
        DLinkNode<T> *pre, *p;
        pre=dhead;p=pre->next;
        while (p!=dhead) {                      //用 p 遍历结点并释放其前驱结点
            delete pre;                         //释放 pre 结点
            pre=p; p=p->next;                   //pre、p 同步后移一个结点
        }
        delete pre;                             //p 等于 dhead 时 pre 指向尾结点,释放尾结点
    }
    void CreateListF(T a[],int n) {             //用头插法建立循环双链表
        for (int i=0;i<n;i++) {                 //循环建立数据结点
            DLinkNode<T> *s=new DLinkNode<T>(a[i]);         //创建数据结点 s
            s->next=dhead->next;                //修改 s 结点的 next 域
            dhead->next->prior=s;
            dhead->next=s;                      //修改头结点的 next 域
            s->prior=dhead;                     //修改 s 结点的 prior 域
        }
    }
```

```
void CreateListR(T a[],int n) {                    //用尾插法建立循环双链表
    DLinkNode<T> *s, *r;
    r=dhead;                                       //r 始终指向尾结点,开始时指向头结点
    for (int i=0;i<n;i++) {                        //循环建立数据结点
        s=new DLinkNode<T>(a[i]);                  //创建数据结点 s
        r->next=s;                                 //将结点 s 插入结点 r 之后
        s->prior=r;
        r=s;
    }
    r->next=dhead;                                 //将尾结点的 next 域置为 dhead
    dhead->prior=r;                                //将头结点的 prior 域置为结点 r
}
void Add(T e) {                                    //在循环双链表的末尾添加一个值为 e 的结点
    DLinkNode<T> *s=new DLinkNode<T>(e);           //新建结点 s
    DLinkNode<T> *p=dhead;
    while (p->next!=dhead)                         //查找尾结点 p
        p=p->next;
    p->next->prior=s;                              //在结点 p 之后插入结点 s
    s->next=p->next;
    p->next=s;
    s->prior=p;
}
int Getlength() {                                  //求循环双链表中数据结点的个数
    DLinkNode<T> *p=dhead;
    int cnt=0;
    while (p->next!=dhead) {                       //查找到尾结点为止
        cnt++;
        p=p->next;
    }
    return cnt;
}
bool GetElem(int i,T &e) {                         //求循环双链表中序号为 i 的结点的值
    if (i<0) return false;                         //参数 i 错误返回 false
    DLinkNode<T> *p=geti(i);                       //查找序号为 i 的结点
    if (p!=dhead) {                                //找到了序号为 i 的结点 p
        e=p->data;
        return true;                               //成功找到返回 true
    }
    else return false;                             //没有找到序号为 i 的结点返回 false
}
bool SetElem(int i,T e) {                          //设置序号为 i 的结点的值
    if (i<0) return false;                         //参数 i 错误返回 false
    DLinkNode<T> *p=geti(i);                       //查找序号为 i 的结点
    if (p!=NULL) {                                 //找到了序号为 i 的结点 p
        p->data=e;
        return true;
    }
    else                                           //没有找到序号为 i 的结点
        return false;                              //参数 i 错误返回 false
}
int GetNo(T e) {                                   //查找第一个为 e 的元素的序号
    int j=0;
    DLinkNode<T> *p=dhead->next;
```

```
        while (p!=dhead && p->data!=e) {
            j++;                                    //查找元素 e
            p=p->next;
        }
        if (p==dhead) return -1;                    //未找到时返回-1
        else return j;                              //找到后返回其序号
    }
    bool Insert(int i,T e) {                        //在循环双链表中序号为 i 的位置插入值为 e 的结点
        if (i<0) return false;                      //参数 i 错误返回 false
        DLinkNode<T> *s=new DLinkNode<T>(e);        //建立新结点 s
        DLinkNode<T> *p=geti(i-1);                  //查找序号为 i-1 的结点 p
        if (p!=NULL) {                              //找到了序号为 i-1 的结点
            s->next=p->next;                        //修改 s 结点的 next 域
            if (p->next!=NULL)                      //修改 p 结点的非空后继结点的 prior 域
                p->next->prior=s;
            p->next=s;                              //修改 p 结点的 next 域
            s->prior=p;                             //修改 s 结点的 prior 域
            return true;                            //插入成功返回 true
        }
        else                                        //没有找到序号为 i-1 的结点
            return false;                           //参数 i 错误返回 false
    }
    bool Delete(int i) {                            //在循环双链表中删除序号为 i 的位置的结点
        if (i<0) return false;                      //参数 i 错误返回 false
        DLinkNode<T> *p=geti(i);                    //查找序号为 i 的结点
        if (p!=NULL) {                              //找到了序号为 i 的结点 p
            p->prior->next=p->next;                 //修改 p 结点的前驱结点的 next 域
            p->next->prior=p->prior;                //修改 p 结点的后继结点的 prior 域
            delete p;                               //释放空间
            return true;                            //删除成功返回 true
        }
        else                                        //没有找到序号为 i-1 的结点
            return false;                           //参数 i 错误返回 false
    }
    void DispList() {                               //输出循环双链表中的所有结点值
        DLinkNode<T> *p;
        p=dhead->next;                              //p 指向开始结点
        while (p!=dhead) {                          //p 不为 dhead,输出 p 结点的 data 域
            cout << p->data << " ";
            p=p->next;                              //p 移向下一个结点
        }
        cout << endl;
    }
private:
    //*********************************************
    //序号 i 的正确范围为-1≤i<n,超出范围返回 NULL
    //i=-1 时返回头结点 dhead
    //i≥0 并且 i<n 时返回序号为 i 的结点
    //*********************************************
    DLinkNode<T> *geti(int i) {                     //返回序号为 i 的结点
        if (i<-1) return NULL;                      //i<-1 时返回 NULL
        if (i==-1) return dhead;                    //i=-1 时返回头结点
        DLinkNode<T> *p=dhead->next;                //首先 p 指向首结点
```

```
        int j=0;                                    //j 置为 0
        while (j<i && p!=dhead) {                   //p 移动 i 个结点
            j++;
            p=p->next;
        }
        if (p==dhead) return NULL;                  //没有找到序号为 i 的结点返回 NULL
        else return p;
    }
};
int main() {
    CDLinkList<int> L;                              //建立元素类型为 int 的循环双链表对象 L
    printf("\n");
    printf("  ============循环双链表====================\n");
    printf("  建立空表 L,长度=%d\n",L.Getlength());
    int a[]={1,2,3,4,5,6};
    int n=sizeof(a)/sizeof(a[0]);
    printf("  1-6 创建 L\n");
    L.CreateListR(a,n);
    printf("  L[长度=%d]: ",L.Getlength()); L.DispList();
    printf("  添加 6-7\n");
    L.Add(6); L.Add(7);
    printf("  在序号 1 处插入 10\n");
    L.Insert(1,10);
    printf("  L[长度=%d]: ",L.Getlength()); L.DispList();
    int e;
    L.GetElem(2,e);
    printf("  序号为 2 的元素=%d\n",e);
    printf("  设置序号为 2 的元素为 20\n");
    L.SetElem(2,20);
    printf("  L[长度=%d]: ",L.Getlength()); L.DispList();
    int x=6;
    printf("  第一个值为%d 的元素序号=%d\n",x,L.GetNo(x));
    printf("  删除首元素\n");
    L.Delete(0);
    printf("  L[长度=%d]: ",L.Getlength()); L.DispList();
    printf("  删除序号为 5 的元素\n");
    L.Delete(5);
    printf("  L[长度=%d]: ",L.Getlength()); L.DispList();
    return 0;
}
```

上述程序的执行结果如图 2.8 所示。

图 2.8　第 2 章基础实验题 5 的执行结果

2.4 应用实验题及其参考答案

2.4.1 应用实验题

1. 编写一个实验程序实现以下功能。

(1) 从文本文件 xyz1.in 中读取两行整数(每行至少有一个整数,以换行符结束),每行的整数按递增排列,两个整数之间用一个空格分隔,全部整数的个数为 n。

(2) 求这 n 个整数中前 $k(1\leqslant k\leqslant n)$个较小的整数。

2. 编写一个实验程序实现以下功能。

(1) 从文本文件 xyz2.in 中读取 3 行整数,每行的整数按递增排列,两个整数之间用一个空格分隔,全部整数的个数为 n。

(2) 将这 n 个整数归并到递增有序表 L 中。

3. 编写一个实验程序实现以下功能。

(1) 从文本文件 xyz3.in 中读取 3 行整数(每行至少有一个整数,以换行符结束),每行的整数按递增排列,两个整数之间用一个空格分隔,全部整数的个数为 n,这 n 个整数均不相同。

(2) 求这 n 个整数中第 $k(1\leqslant k\leqslant n)$小的整数。

4. 有一个学生成绩文本文件 xyz4.in,第一行为整数 n,接下来为 n 行学生基本信息,包括学号、姓名和班号,然后为整数 m,然后为 m 行课程信息,包括课程编号和课程名,再接下来为整数 k,然后为 k 行学生成绩,包括学号、课程编号和分数。例如,$n=5$、$m=3$、$k=15$ 时的 exp1.txt 文件实例如下:

```
5
1 陈斌 101
3 王辉 102
5 李君 101
4 鲁明 101
2 张昂 102
3
2 数据结构
1 C 程序设计
3 计算机导论
15
1 1 82
4 1 78
5 1 85
2 1 90
3 1 62
1 2 77
4 2 86
5 2 84
2 2 88
3 2 80
1 3 60
```

```
4 3 79
5 3 88
2 3 86
3 3 90
```

编写一个程序按班号递增排序输出所有学生的成绩，相同班号按学号递增排序，同一个学生按课程编号递增排序，相邻的班号和学生信息不重复输出。例如，上述 exp1.txt 文件对应的输出如下：

```
  输出结果
=======班号:101=====================
  1  陈斌  C程序设计   82
           数据结构     77
           计算机导论   60
  4  鲁明  C程序设计   78
           数据结构     86
           计算机导论   79
  5  李君  C程序设计   85
           数据结构     84
           计算机导论   88

=======班号:102=====================
  2  张昂  C程序设计   90
           数据结构     88
           计算机导论   86
  3  王辉  C程序设计   62
           数据结构     80
           计算机导论   90
```

5. 编写一个实验程序实现以下功能。

(1) 输入一个正整数 $n(n>2)$，建立带头结点的整数双链表 L，$L=(1,2,\cdots,n)$，该双链表中的每个结点除了有 prior、data 和 next 域外，还有一个访问频度域 freq，初始时值均为 0。

(2) 可以多次按整数 x 查找，每次查找到 x 时，令元素值为 x 的结点的 freq 域值加 1，并调整表中结点的次序，使其按访问频度的递减顺序排列，以便使频繁访问的结点总是靠近表头。

2.4.2 应用实验题参考答案

1. 解：用 vector<int>向量数组 L 存放两个递增整数序列，两个递增整数序列的段号为 0～1。readdata()函数用于从文本文件 xyz1.in 读取数据建立 L，firstk()函数采用二路归并方法求前 $k(1\leqslant k\leqslant n)$个较小的整数。

firstk()函数的思路是用 vector<int>向量 ans 存放前 $k(1\leqslant k\leqslant n)$个较小的整数，用 i、j 变量分别遍历 $L[0]$和 $L[1]$(初始值均为 0)。用一维数组 x 存放各个有序段当前遍历的整数，min2()求 x 中最小整数的段号，当最小整数为 INF(∞)时段号取－1。首先在 x 中取各个段的第一个整数，然后如下循环：调用 min2()求最小元素的段号 mini，若为－1 则返回，否则将 x[mini]添加到 ans 中，累加归并次数 cnt。若 cnt=k 则返回，否则后移归并元素

所在段的遍历指针。若该指针没有越界，则将新元素存放在 x[mini]中，否则将 INF 存放在 x[mini]中。

对应的实验程序 Exp2-1.cpp 如下：

```
#include<iostream>
#include<fstream>
#include<sstream>
#include<vector>
using namespace std;
const int INF=0x3f3f3f3f;                    //∞表示最大的整数
vector<int> L[2];                            //存放两个有序整数序列
vector<int> ans;                             //存放结果
void readdata() {                            //从文件中读取两个有序整数序列
    ifstream fin("xyz1.in");
    string line;
    int i=0;
    while (getline(fin,line)) {              //从文件中读取各行数据
        stringstream ss;                     //利用字符串输入流 ss 提取各个整数
        ss << line;                          //向流中传值
        int tmp;
        while (ss >> tmp)                    //提取 int 数据
            L[i].push_back(tmp);             //保存到 3 个 vector 中
        i++;
    }
    fin.close();
}
int min2(int x[]) {                          //返回最小元素的段号
    int mini=0;
    if (x[1]<x[mini])
        mini=1;
    if (x[mini]==INF) return -1;
    else return mini;
}
void firstk(int k) {                         //求前 k 个较小的元素并存放在 ans 中
    int x[2];
    int i=0,j=0;
    x[0]=L[0][i];
    x[1]=L[1][j];
    int cnt=0;                               //累计归并次数
    while (true) {
        int mini=min2(x);
        if (mini==-1) return;                //归并结束
        ans.push_back(x[mini]);
        cnt++;                               //归并次数增 1
        if (cnt==k) return;                  //找到第 k 小的元素时返回
        if (mini==0) {
            i++;
            x[0]=(i<L[0].size()?L[0][i]:INF);
        }
        else {
            j++;
            x[1]=(j<L[1].size()?L[1][j]:INF);
```

```
        }
    }
}
int main() {
    readdata();
    cout << endl;
    for (int i=0;i<3;i++) {
        printf("  第%d个有序序列: ",i+1);
        for (int j=0;j<L[i].size();j++)
            cout << L[i][j] << " ";
        cout << endl;
    }
    cout << "  =======求解结果========" << endl;
    for (int k=1;k<=L[0].size()+L[1].size();k++) {
        cout << "  前" << k << "个小元素: ";
        ans.clear();
        firstk(k);
        for (int i=0;i<ans.size();i++)
            cout << " " << ans[i];
        cout << endl;
    }
    return 0;
}
```

假设 xyz1.in 文件如下：

```
1 2 2 3 5 8
2 3 4 5
```

```
第1个有序序列: 1 2 2 3 5 8
第2个有序序列: 2 3 4 5
第3个有序序列:
======求解结果=======
前1个小元素:  1
前2个小元素:  1 2
前3个小元素:  1 2 2
前4个小元素:  1 2 2 2
前5个小元素:  1 2 2 2 3
前6个小元素:  1 2 2 2 3 3
前7个小元素:  1 2 2 2 3 3 4
前8个小元素:  1 2 2 2 3 3 4 5
前9个小元素:  1 2 2 2 3 3 4 5 5
前10个小元素:  1 2 2 2 3 3 4 5 5 8
```

图 2.9 第 2 章应用实验题 1 的执行结果

执行上述程序的结果如图 2.9 所示。

2. 解：题目中所有元素均为整数，INF 表示最大整数(∞)，用 3 个 vector < int >向量 $L0$、$L1$、$L2$ 存放 3 个有序序列，它们的段号分别为 0、1、2，vector < int >向量 L 存放结果。用 i、j、k 作为遍历指针分别遍历 $L0$、$L1$、$L2$，将它们指向的元素值存放在数组 x 中，当段号为 i 的遍历完时 $x[i]$取值为 INF。

采用三路归并方法，求出 x 中的最小元素的段号 mini，当最小元素 x[mini]为 INF 时表示所有元素归并完毕，置 mini=－1，此时退出三路归并，否则将 x[mini]添加到 L 中，将 mini 段的遍历指针后移一个元素，x[mini]置为该指针指向的元素。

对应的实验程序 Exp2-2.cpp 如下：

```
#include < iostream >
#include < fstream >
#include < sstream >
#include < vector >
using namespace std;
const int INF=0x3f3f3f3f;                    //∞表示最大的整数
vector < int > L0,L1,L2;                     //存放 3 个有序整数序列
```

```
vector<int> L;                                  //存放三路归并结果
void trans(string line,vector<int> &L) {        //在 line 中提取整数放到 L 中
    stringstream ss;                            //利用字符串输入流 ss 提取各个整数
    ss << line;                                 //向流中传值
    int tmp;
    while (ss >> tmp)                           //提取 int 数据
        L.push_back(tmp);                       //保存到 L 中
}
void readdata() {                               //从文件中读取 3 个有序整数序列
    ifstream fin("xyz2.in");
    string line;
    getline(fin,line); trans(line,L0);
    getline(fin,line); trans(line,L1);
    getline(fin,line); trans(line,L2);
    fin.close();
}
int min3(int x[]) {                             //返回最小元素的段号
    int mini=0;
    for (int i=1;i<3;i++)
        if (x[i]<x[mini]) mini=i;
    if (x[mini]==INF) return -1;
    else return mini;
}
void Merge3() {                                 //三路归并
    int x[3];
    int i=0,j=0,k=0;
    x[0]=L0[i];                                 //初始化 x
    x[1]=L1[j];
    x[2]=L2[k];
    while (true) {
        int mini=min3(x);
        if (mini==-1) return;                   //归并结束
        L.push_back(x[mini]);                   //归并的元素添加到 L 中
        switch(mini) {
            case 0:
                i++;                            //归并元素所在段的指针后移
                x[0]=(i<L0.size()?L0[i]:INF);   //更新 x[mini]
                break;
            case 1:
                j++;
                x[1]=(j<L1.size()?L1[j]:INF);
                break;
            case 2:
                k++;
                x[2]=(k<L2.size()?L2[k]:INF);
                break;
        }
    }
}
void display(vector<int> L) {                   //输出 L
    for (int i=0;i<L.size();i++)
        cout << " " << L[i];
    cout << endl;
```

```
}
int main() {
    readdata();
    cout << endl;
    cout << "  第 1 个有序序列: "; display(L0);
    cout << "  第 2 个有序序列: "; display(L1);
    cout << "  第 3 个有序序列: "; display(L2);
    cout << "  ==========求解结果==========" << endl;
    Merge3();
    cout << "  L:";
    for (int i=0;i<L.size();i++)
        cout << " " << L[i];
    cout << endl;
    return 0;
}
```

假设 xyz2.in 文件如下：

```
1 5 8
2 4 7 10 12 13
3 6 9 11
```

```
第1个有序序列:  1 5 8
第2个有序序列:  2 4 7 10 12 13
第3个有序序列:  3 6 9 11
=========求解结果=========
L: 1 2 3 4 5 6 7 8 9 10 11 12 13
```

图 2.10　第 2 章应用实验题 2 的执行结果

上述程序的执行结果如图 2.10 所示。

3. 解：用 vector < int >向量数组 L 存放 3 个递增整数序列，3 个递增整数序列的段号为 0～2。readdata()函数用于从文本文件 xyz3.in 读取数据建立 L，topk()函数采用三路归并方法求第 k(1≤k≤n)小的整数。

topk()函数的思路是用 p[0]～p[2]整数变量分别遍历 L[0]～L[2]，初始值均为 0。一维数组 x 存放各个有序段当前遍历的整数，即 x[0]存放 L[0][p[0]]，x[1]存放 L[1][p[1]]，x[2]存放 L[2][p[2]]，min3()函数求 x 中最小整数的段号。首先在 x 中取各个段的第一个整数，然后如下循环：调用 min3()求最小元素的段号 mini，累加调用 min3()的次数 cnt，若 cnt=k，则 x[mini]便是第 k 小的整数，退出循环并返回该整数，否则后移一次 i[mini]，重置 x[mini]值(对应段没有遍历尾，取 x[mini]=L[mini][i[mini]]，否则取 x[mini]=INF)。

对应的实验程序 Exp2-3.cpp 如下：

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
using namespace std;
const int INF=0x3f3f3f3f;                    //设置∞表示最大的整数
vector<int> L[3];                            //存放 3 个有序整数序列
void readdata() {                            //从文件中读取 3 个有序整数序列
    ifstream fin("xyz3.in");
    string line;
    int i=0;
    while (getline(fin,line)) {              //从文件中读取各行数据
        stringstream ss;                     //利用字符串输入流 ss 提取各个整数
        ss << line;                          //向流中传值
```

```
        int tmp;
        while (ss >> tmp)                          //提取 int 数据
            L[i].push_back(tmp);                   //保存到 3 个 vector 中
        i++;
    }
    fin.close();
}
int min3(int x[]) {                                //返回最小元素的段号
    int mini=0;
    for (int i=1;i<3;i++)
        if (x[i]<x[mini]) mini=i;
    return mini;
}
int topk(int k) {                                  //返回第 k 小的元素
    int x[3];
    int p[3]={0,0,0};
    for (int i=0;i<3;i++)
        x[i]=L[i][p[i]];
    int cnt=0;                                     //累计归并次数
    while (true) {
        int mini=min3(x);
        cnt++;                                     //归并次数增 1
        if (cnt==k)                                //找到第 k 小的元素
            return x[mini];
        p[mini]++;                                 //归并元素所在段的指针后移
        if (p[mini]<L[mini].size())                //归并元素所在段没有改变完
            x[mini]=L[mini][p[mini]];              //取 p[mini]指向的元素
        else
            x[mini]=INF;
    }
}
int main() {
    readdata();
    cout << endl;
    for (int i=0;i<3;i++) {                        //输出 3 个有序序列
        printf("  第%d 个有序序列: ",i+1);
        for (int j=0;j<L[i].size();j++)
            cout << L[i][j] << " ";
        cout << endl;
    }
    cout << "  =======求解结果========" << endl;
    for (int k=1;k<=L[0].size()+L[1].size()+L[2].size();k++)
        cout << "  第" << k << "小的元素: " << topk(k) << endl;
    return 0;
}
```

假设 xyz3.in 文件如下：

```
1 5 8
2 4 7 10 12 15
3 6 9 11 13 14
```

图 2.11 第 2 章应用实验题 3 的执行结果

执行上述程序的结果如图 2.11 所示。

4. 解：设计 readdata()函数从 xyz4.in 文件中读取数据存放到 3 个 list 链表中，其中 list＜STUD＞链表容器 stud 存放学生基本信息(包含学号、姓名和班号成员)，list＜COURSE＞链表容器 cour 存放课程信息(包含课程号和课程名成员)，list＜FRACTION＞链表容器 frac 存放成绩信息(包含学号、姓名、班号、课程号、课程名和分数成员，初始时仅有学号、课程号和分数成员值)。

设计 solve()函数输出题目要求的结果，其过程是将 stud 和 frac 链表分别按学号递增排序，采用二路归并方法产生 frac 链表中每个结点的姓名和班号成员值，再将 cour 和 frac 分别按课程号递增排序，采用二路归并方法产生 frac 链表中每个结点的课程名成员值，然后将 frac 链表按班号＋学号递增排序，此时 frac 链表如下：

```
[101,1,陈斌,1,C 程序设计,82]
[101,1,陈斌,2,数据结构,77]
[101,1,陈斌,3,计算机导论,60]
[101,4,鲁明,1,C 程序设计,78]
[101,4,鲁明,2,数据结构,86]
[101,4,鲁明,3,计算机导论,79]
[101,5,李君,1,C 程序设计,85]
[101,5,李君,2,数据结构,84]
[101,5,李君,3,计算机导论,88]
[102,2,张昂,1,C 程序设计,90]
[102,2,张昂,2,数据结构,88]
[102,2,张昂,3,计算机导论,86]
[102,3,王辉,1,C 程序设计,62]
[102,3,王辉,2,数据结构,80]
[102,3,王辉,3,计算机导论,90]
```

最后输出结果，在输出中判断相邻的班号和学生信息以保证不重复输出。对应的实验程序 Exp2-4.cpp 如下：

```
#include＜iostream＞
#include＜fstream＞
#include＜sstream＞
#include＜list＞
using namespace std;
struct STUD {                                    //学生基本信息
    int xh;                                      //学号
    string xm;                                   //姓名
    string bh;                                   //班号
    STUD() {}                                    //构造函数
    STUD(int xh1,string xm1,string bh1) {        //重载构造函数
        xh=xh1;
        xm=xm1;
        bh=bh1;
    }
    bool operator＜(const STUD &s) {             //用于按 xh 递增排序
```

```
        return xh < s.xh;
    }
};
struct COURSE {                                    //课程信息
    int kch;                                       //课程号
    string kcm;                                    //课程名
    COURSE() {}                                    //构造函数
    COURSE(int kch1,string kcm1) {                 //重载构造函数
        kch=kch1;
        kcm=kcm1;
    }
    bool operator<(const COURSE &c) {              //用于按 kch 递增排序
        return kch < c.kch;
    }
};
struct FRACTION {                                  //成绩信息
    int xh;                                        //学号
    int kch;                                       //课程号
    int fs;                                        //分数
    string xm;                                     //姓名
    string bh;                                     //班号
    string kcm;                                    //课程名
    FRACTION() {}                                  //构造函数
    FRACTION(int xh1,int kch1,int fs1) {           //重载构造函数
        xh=xh1;
        kch=kch1;
        fs=fs1;
    }
    bool operator<(const FRACTION &f) {            //用于按 xh 递增排序
        return xh < f.xh;
    }
};
struct Cmp1 {                                      //重载()运算符 1
    bool operator()(const FRACTION &s,const FRACTION &t) const {
        return s.kch < t.kch;                      //用于按 kch 递增排序
    }
};
struct Cmp2 {                                      //重载()运算符 2
    bool operator()(const FRACTION &s,const FRACTION &t) const {
        if (s.bh==t.bh)                            //用于按 bh+xh 递增排序
            return s.xh < t.xh;
        else
            return s.bh < t.bh;
    }
};
// ********************************************
list<STUD> stud;                                   //学生基本信息链表 stud
list<COURSE> cour;                                 //课程信息链表 cour
list<FRACTION> frac;                               //成绩信息链表 frac
// ********************************************
void CreateStud(fstream &fin,int n) {              //创建学生基本信息链表 stud
    int xh;
    string xm,bh;
```

```
    for (int i=0;i<n;i++) {
        fin >> xh >> xm >> bh;
        STUD s(xh,xm,bh);
        stud.push_back(s);
    }
}
void CreateCour(fstream &fin,int n) {                    //创建课程信息链表 cour
    int kch;
    string kcm;
    for (int i=0;i<n;i++) {
        fin >> kch >> kcm;
        COURSE s(kch,kcm);
        cour.push_back(s);
    }
}
void CreateFrac(fstream &fin,int n) {                    //创建成绩信息链表 frac
    int xh,kch,fs;
    for (int i=0;i<n;i++) {
        fin >> xh >> kch >> fs;
        FRACTION s(xh,kch,fs);
        frac.push_back(s);
    }
}
void UpdateFrac() {                                      //修改 frac 获取姓名、班号和课程名成员值
    stud.sort();
    frac.sort();
    list<STUD>::iterator sit=stud.begin();
    list<FRACTION>::iterator fit=frac.begin();
    while (sit!=stud.end() && fit!=frac.end()) {   //学生基本信息链表和成绩信息链表按学号归并
        if (sit->xh<fit->xh)
            sit++;
        else if (sit->xh>fit->xh)
            fit++;
        else {
            fit->xm=sit->xm;
            fit->bh=sit->bh;
            fit++;
        }
    }
    cour.sort();                                         //cour 按课程号排序
    frac.sort(Cmp1());                                   //frac 按课程号排序
    list<COURSE>::iterator cit=cour.begin();
    fit=frac.begin();
    while (cit!=cour.end() && fit!=frac.end()) {    //课程信息链表和成绩信息链表按课程号归并
        if (cit->kch<fit->kch)
            cit++;
        else if (cit->kch>fit->kch)
            fit++;
        else {
            fit->kcm=cit->kcm;
            fit++;
        }
    }
```

```
}
void solve() {                                          //求解算法
    UpdateFrac();
    frac.sort(Cmp2());                                  //frac 按班号+学号排序
    printf("\n    输出结果\n\n");
    list<FRACTION>::iterator fit=frac.begin();
    string bh=fit->bh;
    int xh=fit->xh;
    cout << "    ====班号:" << bh << "==================" << endl;
    for (;fit!=frac.end();fit++) {
        if (fit->bh!=bh) {
            bh=fit->bh;
            printf("\n");
            cout << "    ====班号:" << bh << "==================" << endl;
        }
        if (fit==frac.begin() || fit->xh!=xh) {
            xh=fit->xh;
            cout << "    " << fit->xh << "\t" << fit->xm << "\t"
                 << fit->kcm << "\t" << fit->fs << endl;
        }
        else cout << "                " << fit->kcm << "\t" << fit->fs << endl;
    }
}
void readdata() {                                       //从文件中读取 3 个有序整数序列
    fstream fin("xyz4.in", ios::in);
    int n;
    fin >> n;
    CreateStud(fin,n);
    fin >> n;
    CreateCour(fin,n);
    fin >> n;
    CreateFrac(fin,n);
    fin.close();
}
int main() {
    readdata();
    solve();
    return 0;
}
```

上述程序的执行结果如图 2.12 所示。

```
输出结果

======班号:101==================
 1   陈斌    C程序设计      82
             数据结构       77
             计算机导论     60
 4   鲁明    C程序设计      78
             数据结构       86
             计算机导论     79
 5   李君    C程序设计      85
             数据结构       84
             计算机导论     88

======班号:102==================
 2   张昂    C程序设计      90
             数据结构       88
             计算机导论     86
 3   王辉    C程序设计      62
             数据结构       80
             计算机导论     90
```

图 2.12　第 2 章应用实验题 4 的执行结果

5. 解：设计双链表结点类 DLinkNode，它含数据成员 data、prior、next 和 freq(访问频度)。设计双链表类 DLinkList，它含双链表头结点的 dhead 成员，以及建表方法 CreateListR()和输出方法 DispList()，CreateListR()采用尾插法建立形如{1，2，…，n}的带头结点的整数双链表，每个结点的 freq 成员均置为 0。

设计 Locate(L，x)函数，先查找值为 x 的结点 p，找到后将 p 结点的 freq 成员增 1，再向前找到结点 pre，若满足 pre->freq≥p->freq，则将 p 结点移到 pre 结点之后，如图 2.13 所示。其操作是先删除 p 结点，再将其插入 pre 结点之后。

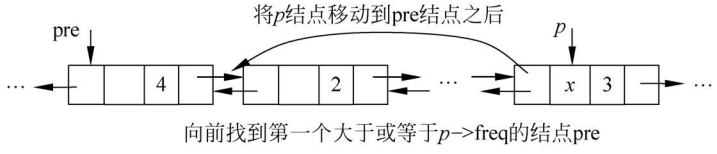


图 2.13 将 p 结点移动到 pre 结点之后

对应的实验程序 Exp2-5.cpp 如下：

```
#include<iostream>
using namespace std;
struct DLinkNode {                          //双链表结点类型
    int data;                               //存放数据元素
    int freq;                               //结点访问频度
    DLinkNode *next;                        //指向后继结点的指针
    DLinkNode *prior;                       //指向前驱结点的指针
    DLinkNode() {                           //构造函数
        next=prior=NULL;
        freq=0;
    }
    DLinkNode(int d) {                      //重载构造函数
        data=d;
        next=prior=NULL;
        freq=0;
    }
};
class DLinkList {                           //双链表类
public:
    DLinkNode *dhead;                       //双链表的头结点
    DLinkList() {                           //构造函数,创建一个空双链表
        dhead=new DLinkNode();
    }
    ~DLinkList() {                          //析构函数,销毁双链表
        DLinkNode *pre, *p;
        pre=dhead;p=pre->next;
        while (p!=NULL) {                   //用 p 遍历结点并释放其前驱结点
            delete pre;                     //释放 pre 结点
            pre=p; p=p->next;               //pre、p 同步后移一个结点
        }
        delete pre;                         //p 为空时 pre 指向尾结点,此时释放尾结点
    }
    void CreateListR(int n) {               //用尾插法建立双链表
        DLinkNode *s, *r;
```

```
        r=dhead;                                    //r 始终指向尾结点,开始时指向头结点
        for (int i=1;i<=n;i++) {                    //循环建立数据结点
            s=new DLinkNode(i);                     //创建数据结点 s
            r->next=s;                              //将 s 结点插入 r 结点之后
            s->prior=r;
            r=s;
        }
        r->next=NULL;                               //将尾结点的 next 域置为 NULL
    }
    void DispList() {                               //输出双链表的所有结点值
        DLinkNode *p=dhead->next;                   //p 指向开始结点
        while (p!=NULL) {                           //p 不为 NULL,输出 p 结点的 data 域
            cout << "  " << p->data << "[" << p->freq << "]";
            p=p->next;                              //p 移向下一个结点
        }
        cout << endl;
    }
};
//实现实验题功能的函数
bool Locate(DLinkList &L,int x) {                   //查找值为 x 的结点
    DLinkNode *p=L.dhead->next, *pre;
    while (p!=NULL && p->data!=x)
        p=p->next;                                  //查找 data 值为 x 的结点 p
    if (p==NULL)                                    //未找到这样的结点
        return false;
    else {                                          //找到这样的结点 p
        p->freq++;                                  //频度增 1
        pre=p->prior;                               //pre 为 p 的前驱结点
        if (pre!=L.dhead) {                         //若 pre 不为头结点
            while (pre!=L.dhead && pre->freq<p->freq)
                pre=pre->prior;                     //则向前查找 pre 结点
            p->prior->next=p->next;                 //先删除 p 结点
            if (p->next!=NULL)
                p->next->prior=p->prior;
            p->next=pre->next;                      //将 p 结点插入 pre 结点之后
            if (pre->next!=NULL)
                pre->next->prior=p;
            pre->next=p;
            p->prior=pre;
        }
        return true;
    }
}
void Find(DLinkList &L,int x) {                     //输出查找结果
    if (Locate(L,x)) {
        printf("  查找%d后的结果:",x);
        L.DispList();
    }
    else printf("  查找%d: 没有找到\n",x);
}
int main() {
    DLinkList L;
    L.CreateListR(5);
```

```
    printf("\n");
    printf("  初始 L:"); L.DispList();
    Find(L,10); Find(L,5); Find(L,1); Find(L,4);
    Find(L,5); Find(L,2); Find(L,4); Find(L,5);
    return 0;
}
```

上述程序的执行结果如图 2.14 所示。

```
初始L:  1[0]  2[0]  3[0]  4[0]  5[0]
查找10: 没有找到
查找5后的结果:  5[1]  1[0]  2[0]  3[0]  4[0]
查找1后的结果:  5[1]  1[1]  2[0]  3[0]  4[0]
查找4后的结果:  5[1]  1[1]  4[1]  2[0]  3[0]
查找5后的结果:  5[2]  1[1]  4[1]  2[0]  3[0]
查找2后的结果:  5[2]  1[1]  4[1]  2[1]  3[0]
查找4后的结果:  5[2]  4[2]  1[1]  2[1]  3[0]
查找5后的结果:  5[3]  4[2]  1[1]  2[1]  3[0]
```

图 2.14　第 2 章应用实验题 5 的执行结果

第3章 栈和队列

3.1 问答题及其参考答案

3.1.1 问答题

1. 简述线性表、栈和队列的异同。

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

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

4. 有n个不同元素的序列经过一个栈产生的出栈序列个数是多少？

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

6. 若采用数组data[0..$m-1$]存放栈元素，回答以下问题：

（1）只能以data[0]端作为栈底吗？

（2）为什么不能以data数组的中间位置作为栈底？

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

8. 什么叫队列的“假溢出”？如何解决假溢出？

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

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

3.1.2 问答题参考答案

1. 答：线性表、栈和队列的相同点是它们的元素的逻辑关系都是线性关系；不同点是运算不同，线性表可以在两端和中间任何位置插入和删除元素，而栈只能在一端插入和删除元素，队列只能在一端插入元素，在另外一端删除元素。

2. 答：高级语言变量名的定义规则是以字母开头的字母数字串。进栈次序为 123PA，以 A 最先出栈的序列为 AP321，以 P 最先出栈的序列为 P321A、P32A1、P3A21、PA321。可以作为高级语言的变量名的序列为 AP321、P321A、P32A1、P3A21 和 PA321。

3. 答：合法的栈操作序列必须满足以下两个条件。

① 在操作序列的任何前缀（从开始到任何一个操作时刻）中，I 的个数不得少于 O 的个数。

② 整个操作序列中 I 和 O 的个数相等。

4. 答：设 n 个不同元素的序列经过一个栈产生的出栈序列（顺序）的个数是 $f(n)$，设该输入序列为 $a,b,c,d,\cdots$，出栈序列有 n 个位置，元素 a 的各种可能性如下。

① 若元素 a 在出栈序列的第 1 个位置，则其操作是 a 进栈，a 出栈，还剩下 $n-1$ 个元素，出栈序列个数是 $f(n-1)$，这种情况下的出栈序列个数等于 $f(n-1)$。

② 若元素 a 在出栈序列的第 2 个位置，则一定有一个元素比 a 先出栈，即有 $f(1)$ 种可能的顺序（只能是 b），还剩 $c,d,\cdots$，其顺序个数是 $f(n-2)$。根据乘法原理，顺序个数为 $f(1)\times f(n-2)$。

③ 如果元素 a 在出栈序列的第 3 个位置，那么一定有两个元素比 a 先出栈，即有 $f(2)$ 种可能顺序（只能是 b、c），还剩 $d,\cdots$，其顺序个数是 $f(n-3)$。根据乘法原理，顺序个数为 $f(2)\times f(n-3)$。

以此类推，按照加法原理，假设 $f(0)=1$，有

$$f(n)=\sum_{i=0}^{n-1} f(i)\times f(n-1-i)$$

可以求出

$$f(n)=\frac{1}{n+1}C_{2n}^{n}=\frac{(2n)!}{(n+1)\times(n!)^2}$$

例如，$n=3$ 时，

$$f(3)=\frac{1}{3+1}C_{6}^{3}=\frac{6\times5\times4}{4\times3\times2\times1}=5$$

$n=4$ 时，

$$f(4)=\frac{1}{4+1}C_{8}^{4}=\frac{8\times7\times6\times5}{5\times4\times3\times2\times1}=14$$

5. 答：错误。从理论上讲，对该栈的进栈和出栈操作次数没有限制，但连续的进栈操作最多只能执行 n 次。

6. 答：（1）也可以将 data[$m-1$]端作为栈底。

（2）栈中元素是从栈底向栈顶方向生长的，如果以 data 数组的中间位置作为栈底，那么栈顶方向的另外一端空间就不能使用，造成空间浪费，所以不能以 data 数组的中间位置

作为栈底。

7. 答：栈具有顺序存取特性，假设从栈底到栈顶的元素是 $a_0,a_1,\cdots,a_{n-2},a_{n-1}$，出栈栈顶元素 a_{n-1} 后，下一次可以出栈新栈顶元素 a_{n-2}，以此类推，这称为顺序存取特性。链栈和顺序栈都是栈的存储结构，体现栈的特性，都只能顺序存取，而不能随机存取。

8. 答：在非循环顺序队中，当队尾指针已经到了数组的上界，不能再做进队操作，但其实数组中还有空位置，这就叫“假溢出”。解决假溢出的方式之一是采用循环队列。

9. 答：一般教科书中设计循环队列时，让队头指针 f 指向队头元素的前一个位置，队尾指针 r 指向队尾元素。这里是队头指针 f 指向队头元素，队尾指针 r 指向队尾元素的下一个位置。这两种方法本质上没有差别，实际上最重要的是能够方便设置队空、队满的条件。

对于题目中指定的循环队列，f、r 的初始值为 0，仍然以 $f==r$ 作为队空的条件，$(r+1)\%m==f$ 作为队满的条件。

元素 x 进队操作：data[r]$=x$；$r=(r+1)\%m$。队尾指针 r 指向队尾元素的下一个位置。

元素 x 出队操作：$x=$data[f]；$f=(f+1)\%m$。队头元素出队后，下一个元素成为队头元素。

10. 答：如果先保存的后处理，则应该采用栈数据结构存放这些元素。如果先保存的先处理，则应该采用队列数据结构存放这些元素。

3.2 算法设计题及其参考答案

3.2.1 算法设计题

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

2. 假设有一个链栈 st，设计一个算法，出栈从栈顶开始的第 k 个结点。

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

4. 用于列车编组的铁路转轨网络是一种栈结构，如图 3.1 所示。其中，右边轨道是输入端，左边轨道是输出端。当右边轨道上的车皮编号顺序为 1、2、3、4 时，如果执行操作进栈、进栈、出栈、进栈、进栈、出栈、出栈、出栈，则在左边轨道上的车皮编号顺序为 2、4、3、1。

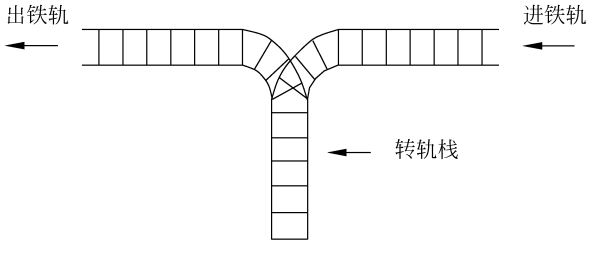
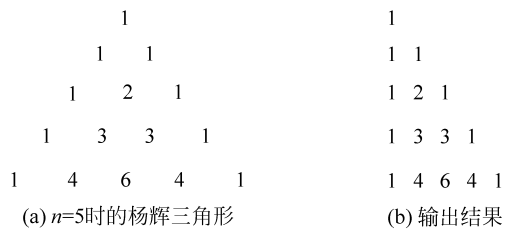


图 3.1 铁路转轨网络

设计一个算法，给定 n 个整数序列 a 表示右边轨道上的车皮编号顺序，用上述转轨栈对这些车皮重新编号，使得编号为奇数的车皮都排在编号为偶数的车皮的前面，要求产生所有操作的字符串 op 和最终结果字符串 ans。

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

6. 对于给定的正整数 $n(n>2)$，利用一个队列输出 n 阶杨辉三角形。5 阶杨辉三角形如图 3.2(a)所示，其输出结果如图 3.2(b)所示。



(a) n=5时的杨辉三角形　　(b) 输出结果

图 3.2　5 阶杨辉三角形及其生成过程

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

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

3.2.2　算法设计题参考答案

1. 解：设计一个栈 st。遍历 str，将其中'@'字符前面的所有字符进栈，再扫描 str 中'@'字符后面的所有字符，对于每个字符 ch，退栈一个字符，如果两者不相同则返回 false。当循环结束时，若 str 扫描完毕并且栈空则返回 true，否则返回 false。对应的算法如下：

```
bool match(string str) {
    SqStack<char> st;                          //定义一个顺序栈
    char e;
    int i=0;
    while (i<str.length() && str[i]!='@') {
        st.push(str[i]);
        i++;
    }
    if (i==str.length())                       //没有找到@,返回 false
        return false;
    i++;                                       //跳过@
    while (i<str.length() && !st.empty()) {    //str 没有扫描完毕并且栈不空时循环
        st.pop(e);
        if (str[i]!=e) return false;           //两者不等,返回 false
        i++;
    }
```

```
    if (i==str.length() && st.empty())          //str 扫描完毕并且栈空时返回 true
        return true;
    else                                        //其他返回 false
        return false;
}
```

2. 解：从链栈头结点 head 开始查找第 $k-1$ 个结点 pre，p 指向其后继结点。本算法是通过结点 pre 删除结点 p 并且取该结点的值，若删除成功则返回 true，若参数错误则返回 false。对应的算法如下：

```
bool popk(LinkStack<int> &st,int k,int &e) {
    if (k<=0) return false;
    LinkNode<int> *pre=st.head, *p;
    int j=0;
    while (pre!=NULL && j<k-1) {                //查找第 k-1 个结点 pre
        pre=pre->next;
        j++;
    }
    if (pre==NULL) return false;                //参数 k 错误
    p=pre->next;                                //p 指向第 k 个结点
    if (p==NULL) return false;                  //参数 k 错误
    e=p->data;                                  //取结点 p 的值
    pre->next=p->next;                          //删除结点 p
    delete p;
    return true;
}
```

3. 解：设置一个顺序栈 st，采用辗转相除法将十进制数 d 转换成 r 进制数，从低到高产生各个位并进栈，再通过栈从高到低将各个位连接起来生成字符串 s，最后返回 s。对应的算法如下：

```
string trans(int d,int r) {
    int x;
    SqStack<int> st;                            //定义一个顺序栈
    while (d>0) {                               //产生转换后的各个位并进栈
        st.push(d%r);
        d/=r;
    }
    string chars="0123456789ABCDEF";
    string s="";
    while (!st.empty()) {                       //将各个位从高到低连接起来
        st.pop(x);
        s+=chars[x];
    }
    return s;
}
```

4. 解：将铁路转轨网络看成一个栈，a 数组表示进栈序列，要求编号为奇数的车皮都排在编号为偶数的车皮的前面，所以遇到奇数的车皮将其进栈保存，遇到偶数的车皮将其进栈后立即出栈，最后将栈中的所有车皮出栈。op 表示操作字符串，ans 表示重编后的车皮序列（初始时均为空串）。对应的算法如下：

```
void solve(int a[],int n,string &op,string &ans) {
    SqStack<int> st;                                    //定义一个顺序栈
    for (int i=0;i<n;i++) {
        if (a[i]%2==1) {                                //若车皮编号为奇数,则进栈
            st.push(a[i]);
            op+="\t"+to_string(a[i])+"进队\n";
        }
        else {                                          //若车皮编号为偶数,则进栈后立即出栈
            op+="\t"+to_string(a[i])+"进栈\n";
            op+="\t"+to_string(a[i])+"出栈\n";
            ans+=to_string(a[i])+" ";
        }
    }
    int x;
    while (!st.empty()) {                               //出栈所有的车皮
        st.pop(x);
        op+="\t"+to_string(x)+"出栈\n";
        ans+=to_string(x)+" ";
    }
}
```

5. 解：设置一个顺序栈 st，先将循环队列 qu 中的所有元素出队并进到 st 栈中，再将栈 st 中的所有元素出栈并进到 qu 队列中。对应的算法如下：

```
void Reverse(CSqQueue<int> &qu) {
    int x;
    SqStack<int> st;                                    //定义一个顺序栈
    while (!qu.empty()) {                               //出队所有元素并进栈
        qu.pop(x);
        st.push(x);
    }
    while (!st.empty()) {                               //出栈所有元素并进队
        st.pop(x);
        qu.push(x);
    }
}
```

6. 解：由 n 阶杨辉三角形的特点可知，其高度为 n，第 $r(1\leqslant r\leqslant n)$ 行恰好包含 r 个数字。在每行前后添加一个 0（第 r 行包含 $r+2$ 个数字），采用迭代方式，定义一个队列 qu，由第 r 行生成第 $r+1$ 行。

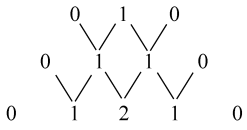


图 3.3 生成前 3 行的过程

① 先输出第 1 行，仅输出 1，将 0、1、0 进队。

② 当队列 qu 中包含第 r 行的全部数字时（队列中共 $r+2$ 个元素），生成并输出第 $r+1$ 行的过程是进队 0，出队元素 s（第一个元素为 0），再依次出队元素 t（共执行 $r+1$ 次），$e=s+t$，输出 e 并进队，重置 $t=s$，最后进队 0，这样输出了第 $r+1$ 行的 $r+1$ 个元素，队列中含 $r+3$ 个元素。图 3.3 所示为生成前 3 行的过程。

对应的算法如下：

```
void YHTriangle(int n) {
    int s,t,e;
```

```
    CSqQueue<int> qu;                          //定义一个循环队列
    printf("%4d\n",1);                         //输出第 1 行
    qu.push(0);                                //第 1 行进队
    qu.push(1);
    qu.push(0);
    for (int r=2;r<=n;r++) {                   //输出第 2 行到第 n 行
        qu.push(0);
        qu.pop(s);
        for (int c=0;c<r;c++) {                //输出第 r 行的 r 个数字
            qu.pop(t);
            e=s+t;
            printf("%4d",e);
            qu.push(e);
            s=t;
        }
        qu.push(0);
        printf("\n");
    }
}
```

7. 解：采用两个队列来实现，先将 a 中的所有奇数位元素进队 qu1 中，所有偶数位元素进队 qu2 中，再将 qu2 中的元素依次出队并放到 a 中，qu1 中的元素依次出队并放到 a 中。对应的算法如下：

```
void Move(int a[],int n) {
    CSqQueue<int> qu1;                         //存放奇数位元素
    CSqQueue<int> qu2;                         //存放偶数位元素
    int i=0,x;
    while (i<n) {
        qu1.push(a[i]);                        //奇数位元素进 qu1 队
        i++;
        if (i<n) {
            qu2.push(a[i]);                    //偶数位元素进 qu2 队
            i++;
        }
    }
    i=0;
    while (!qu2.empty()) {                     //先取 qu2 队列的元素
        qu2.pop(x);
        a[i]=x;
        i++;
    }
    while (!qu1.empty()) {                     //再取 qu1 队列的元素
        qu1.pop(x);
        a[i]=x;
        i++;
    }
}
```

8. 解：初始时 tag=false，front=rear=0，成功的进队操作后 tag=true（任何进队操作后队列都不可能空，但可能满），成功的出队操作后 tag=false（任何出队操作后队列都不可能满，但可能空），因此这样的队列的四要素如下。

① 队空条件：front==rear and tag=false；

② 队满条件：front==rear and tag=true；

③ 元素 x 进队：rear=(rear+1)%MaxSize；data[rear]=x；tag=true；

④ 元素 x 出队：front=(front+1)%MaxSize；x=data[front]；tag=false；

设计对应的循环队列类 QUEUE<T>如下：

```
#define MaxSize 100                          //队列的容量
template <typename T>
class QUEUE {                                //循环队列类模板
public:
    T *data;                                 //存放队中元素
    int front,rear;                          //队头和队尾指针
    bool tag;                                //为 false 表示可能队空,为 true 表示可能队满
    QUEUE() {                                //构造函数
        data=new T[MaxSize];                 //为 data 分配最大容量为 MaxSize 的空间
        front=rear=0;                        //队头、队尾指针置初值
        tag=false;                           //初始时队空,tag 置为 false
    }
    ~QUEUE() {                               //析构函数
        delete [] data;
    }
    //-----------循环队列基本运算算法--------------------
    bool empty() {                           //判队空运算
        return front==rear && tag==false;
    }
    bool full() {                            //判队满运算
        return front==rear && tag==true;
    }
    bool push(T e) {                         //进队列运算
        if (full()) return false;            //队满上溢出
        rear=(rear+1)%MaxSize;
        data[rear]=e;
        tag=true;                            //进队操作,队可能满
        return true;
    }
    bool pop(T &e) {                         //出队列运算
        if (empty()) return false;           //队空下溢出
        front=(front+1)%MaxSize;
        e=data[front];
        tag=true;                            //出队操作,队可能空
        return true;
    }
    bool gethead(T &e) {                     //取队头运算
        if (empty()) return false;           //队空下溢出
        int head=(front+1)%MaxSize;
        e=data[head];
        return true;
    }
};
```

3.3 基础实验题及其参考答案

3.3.1 基础实验题

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

3.3.2 基础实验题参考答案

1. 解：顺序栈的基本运算算法的设计原理参见《教程》中的 3.1.2 节。包含顺序栈基本运算算法类 SqStack 以及测试主程序的 Exp1-1.cpp 文件如下：

```
#include <iostream>
using namespace std;
const int MaxSize=100;                          //栈的容量
template <typename T>
class SqStack {                                 //顺序栈类
    T *data;                                    //存放栈中元素
    int top;                                    //栈顶指针
public:
    SqStack() {                                 //构造函数
        data=new T[MaxSize];                    //为 data 分配最大容量为 MaxSize 的空间
        top=-1;                                 //栈顶指针初始化
    }
    ~SqStack() {                                //析构函数
        delete [] data;
    }
    //-----------栈基本运算算法--------------------
    bool empty() {                              //判断栈是否为空
        return(top==-1);
    }
    bool push(T e) {                            //进栈算法
        if (top==MaxSize-1) return false;       //栈满时返回 false
        top++;                                  //栈顶指针增 1
        data[top]=e;                            //将 e 进栈
        return true;
    }
    bool pop(T &e) {                            //出栈算法
        if (empty()) return false;              //栈为空的情况，即栈下溢出
        e=data[top];                            //取栈顶指针位置的元素
        top--;                                  //栈顶指针减 1
        return true;
    }
    bool gettop(T &e) {                         //取栈顶元素算法
        if (empty()) return false;              //栈为空的情况，即栈下溢出
        e=data[top];                            //取栈顶指针位置的元素
```

```
        return true;
    }
};
int main() {
    SqStack<char> st;                                    //定义一个字符顺序栈 st
    char e;
    cout << "\n 建立空顺序栈 st\n";
    cout << " 栈 st" << (st.empty()?"空":"不空") << endl;
    cout << " 字符 a 进栈\n"; st.push('a');
    cout << " 字符 b 进栈\n"; st.push('b');
    cout << " 字符 c 进栈\n"; st.push('c');
    cout << " 字符 d 进栈\n"; st.push('d');
    cout << " 字符 e 进栈\n"; st.push('e');
    cout << " 栈 st" << (st.empty()?"空":"不空") << endl;
    st.gettop(e);
    cout << " 栈顶元素:" << e << endl;
    cout << " 所有元素出栈次序: ";
    while (!st.empty()) {                                //栈不空时循环
        st.pop(e);                                       //出栈元素 e 并输出
        cout << e << " ";
    }
    cout << endl;
    cout << " 销毁栈 st" << endl;
    return 0;
}
```

上述程序的执行结果如图 3.4 所示。

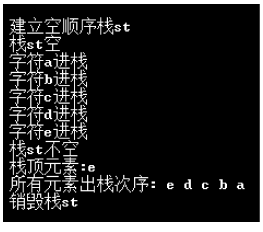


图 3.4　第 3 章基础实验题 1 的执行结果

2. 解：链栈的基本运算算法的设计原理参见《教程》中的 3.1.4 节。包含链栈基本运算算法类 LinkStack 以及测试主程序的 Exp1-2.cpp 文件如下：

```
#include <iostream>
using namespace std;
template <typename T>
struct LinkNode {                                        //链栈结点类型
    T data;                                              //数据域
    LinkNode *next;                                      //指针域
    LinkNode():next(NULL) {}                             //构造函数
    LinkNode(T d):data(d),next(NULL) {}                  //重载构造函数
};
template <typename T>
class LinkStack {                                        //链栈类模板
public:
```

```
    LinkNode<T> *head;                                  //链栈的头结点
    LinkStack() {                                       //构造函数
        head=new LinkNode<T>();
    }
    ~LinkStack() {                                      //析构函数
        LinkNode<T> *pre=head, *p=pre->next;
        while (p!=NULL) {
            delete pre;
            pre=p; p=p->next;                           //pre、p同步后移
        }
        delete pre;
    }
    bool empty() {                                      //判栈空算法
        return head->next==NULL;
    }
    bool push(T e) {                                    //进栈算法
        LinkNode<T> *p=new LinkNode<T>(e);              //新建结点p
        p->next=head->next;                             //插入结点p作为首结点
        head->next=p;
        return true;
    }
    bool pop(T &e) {                                    //出栈算法
        LinkNode<T> *p;
        if (head->next==NULL) return false;             //栈空的情况
        p=head->next;                                   //p指向开始结点
        e=p->data;
        head->next=p->next;                             //删除结点p
        delete p;                                       //释放结点p
        return true;
    }
    bool gettop(T &e) {                                 //取栈顶元素
        LinkNode<T> *p;
        if (head->next==NULL) return false;             //栈空的情况
        p=head->next;                                   //p指向开始结点
        e=p->data;
        return true;
    }
};
int main() {
    LinkStack<char> st;                                 //定义一个字符链栈st
    char e;
    cout << "\n 建立空链栈 st\n";
    cout << " 栈 st" << (st.empty()?"空":"不空") << endl;
    cout << " 字符 a 进栈\n"; st.push('a');
    cout << " 字符 b 进栈\n"; st.push('b');
    cout << " 字符 c 进栈\n"; st.push('c');
    cout << " 字符 d 进栈\n"; st.push('d');
    cout << " 字符 e 进栈\n"; st.push('e');
    cout << " 栈 st" << (st.empty()?"空":"不空") << endl;
    st.gettop(e);
    cout << " 栈顶元素:" << e << endl;
    cout << " 所有元素出栈次序: ";
    while (!st.empty()) {                               //栈不空时循环
```

```
        st.pop(e);                                          //出栈元素 e 并输出
        cout << e << " ";
    }
    cout << endl;
    cout << " 销毁栈 st" << endl;
    return 0;
}
```

上述程序的执行结果如图 3.5 所示。

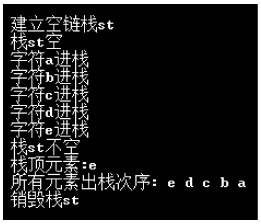


图 3.5　第 3 章基础实验题 2 的执行结果

3. 解：循环队列的基本运算算法的设计原理参见《教程》中的 3.2.2 节。包含循环队列基本运算算法类 CSqQueue 以及测试主程序的 Exp1-3.cpp 文件如下：

```
#include <iostream>
using namespace std;
#define MaxSize 100                                         //队列的容量
template <typename T>
class CSqQueue {                                            //循环队列类模板
public:
    T *data;                                                //存放队中元素
    int front, rear;                                        //队头和队尾指针
    CSqQueue() {                                            //构造函数
        data=new T[MaxSize];                                //为 data 分配最大容量为 MaxSize 的空间
        front=rear=0;                                       //队头、队尾指针置初值
    }
    ~CSqQueue() {                                           //析构函数
        delete [] data;
    }
    //————————————循环队列基本运算算法————————————————————
    bool empty() {                                          //判队空算法
        return (front==rear);
    }
    bool push(T e) {                                        //进队列算法
        if ((rear+1)%MaxSize==front) return false;          //队满上溢出
        rear=(rear+1)%MaxSize;
        data[rear]=e;
        return true;
    }
    bool pop(T &e) {                                        //出队列运算
        if (front==rear) return false;                      //队空下溢出
        front=(front+1)%MaxSize;
        e=data[front];
        return true;
```

```
    }
    bool gethead(T &e) {                                    //取队头运算
        if (front==rear) return false;                      //队空下溢出
        int head=(front+1)%MaxSize;
        e=data[head];
        return true;
    }
};
int main() {
    CSqQueue<char> qu;                                      //定义一个字符顺序队 sq
    char e;
    cout << "\n  建立空顺序队 sq\n";
    cout << "  队列 sq" << (qu.empty()?"空":"不空") << endl;
    cout << "  元素 a 进队\n"; qu.push('a');
    cout << "  元素 b 进队\n"; qu.push('b');
    cout << "  元素 c 进队\n"; qu.push('c');
    cout << "  元素 d 进队\n"; qu.push('d');
    cout << "  元素 e 进队\n"; qu.push('e');
    cout << "  队列 sq" << (qu.empty()?"空":"不空") << endl;
    cout << "  所有元素出队次序: ";
    while (!qu.empty()) {                                   //队不空时循环
        qu.pop(e);                                          //出队元素 e
        cout << e << " ";                                   //输出元素 e
    }
    cout << endl;
    cout << "  销毁队 sq" << endl;
    return 0;
}
```

上述程序的执行结果如图 3.6 所示。

图 3.6　第 3 章基础实验题 3 的执行结果

4. 解：链队的基本运算算法的设计原理参见《教程》中的 3.2.4 节。包含链队基本运算算法类 LinkQueue 以及测试主程序的 Exp1-4.cpp 文件如下：

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

```
class LinkQueue {                                   //链队类模板
public:
    LinkNode<T> *front;                             //队头指针
    LinkNode<T> *rear;                              //队尾指针
    LinkQueue():front(NULL),rear(NULL) {}           //构造函数
    ~LinkQueue() {                                  //析构函数
        LinkNode<T> *pre=front, *p;
        if (pre!=NULL) {                            //非空队的情况
            if (pre==rear)                          //只有一个数据结点的情况
                delete pre;                         //释放 pre 结点
            else {                                  //有两个或多个数据结点的情况
                p=pre->next;
                while (p!=NULL) {
                    delete pre;                     //释放 pre 结点
                    pre=p; p=p->next;               //pre、p 同步后移
                }
                delete pre;                         //释放尾结点
            }
        }
    }
    bool empty() {                                  //判队空运算
        return rear==NULL;
    }
    bool push(T e) {                                //进队运算
        LinkNode<T> *p=new LinkNode<T>(e);
        if (rear==NULL)                             //链队为空的情况
            front=rear=p;                           //新结点既是队首结点又是队尾结点
        else {                                      //链队不空的情况
            rear->next=p;                           //将 p 结点链到队尾,并将 rear 指向它
            rear=p;
        }
        return true;
    }
    bool pop(T &e) {                                //出队运算
        if (rear==NULL) return false;               //队列为空
        LinkNode<T> *p=front;                       //p 指向首结点
        if (front==rear)                            //队列中只有一个结点时
            front=rear=NULL;
        else                                        //队列中有多个结点时
            front=front->next;
        e=p->data;
        delete p;                                   //释放出队结点
        return true;
    }
    bool gethead(T &e) {                            //取队头运算
        if (rear==NULL)                             //队列为空
            return false;
        e=front->data;                              //取首结点的值
        return true;
    }
};
int main() {
    LinkQueue<char> qu;                             //定义一个字符链队 qu
    char e;
```

```
    cout << "\n   建立空链队 qu\n";
    cout << "   队列 qu" << (qu.empty()?"空":"不空") << endl;
    cout << "   元素 a 进队\n"; qu.push('a');
    cout << "   元素 b 进队\n"; qu.push('b');
    cout << "   元素 c 进队\n"; qu.push('c');
    cout << "   元素 d 进队\n"; qu.push('d');
    cout << "   元素 e 进队\n"; qu.push('e');
    cout << "   队列 qu" << (qu.empty()?"空":"不空") << endl;
    cout << "   所有元素出队次序: ";
    while (!qu.empty()) {                                    //队不空时循环
        qu.pop(e);                                           //出队元素 e
        cout << e << " ";                                    //输出元素 e
    }
    cout << endl;
    cout << "   销毁队 qu" << endl;
    return 0;
}
```

上述程序的执行结果如图 3.7 所示。

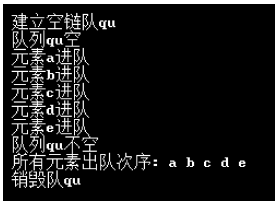


图 3.7　第 3 章基础实验题 4 的执行结果

3.4　应用实验题及其参考答案

3.4.1　应用实验题

1. 改进用栈求解迷宫问题的算法，累计如图 3.8 所示的迷宫的路径条数，并输出所有的迷宫路径。

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

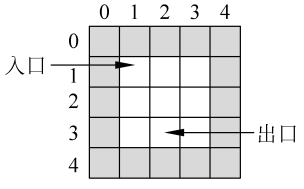


图 3.8　迷宫示意图

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

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

4. 采用一个不带头结点、只有一个尾结点指针 rear 的循环单链表存储队列，设计出这种链队的进队、出队、判队空和求队中元素个数的算法。

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

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

3.4.2 应用实验题参考答案

1. 解：修改《教程》中的 3.1.7 节用栈求解迷宫问题的 mgpath()算法，用 cnt 累计找到的迷宫路径条数(初始为 0)。当找到一条路径后并不返回，而是将 cnt 增加 1，输出该迷宫路径，然后出栈栈顶方块 *b* 并将该方块的 mg 值恢复为 0，继续前面的过程，直到栈空为止，最后返回 cnt。对应的实验程序 Exp2-1.cpp 如下：

```
#include<iostream>
#include<stack>
using namespace std;
const int MAX=10;
int cnt=0;                                  //累计迷宫路径条数
int mg[MAX][MAX]={{1,1,1,1,1},{1,0,0,0,1},{1,0,0,0,1},{1,0,0,0,1},{1,1,1,1,1}};
int m=5,n=5;                                //一个 5 行 5 列的迷宫图
int dx[]={-1,0,1,0};                        //x 方向的偏移量
int dy[]={0,1,0,-1};                        //y 方向的偏移量
struct Box {                                //方块结构体
    int i;                                  //方块的行号
    int j;                                  //方块的列号
    int di;                                 //di 是下一可走相邻方块的方位号
    Box() {}                                //构造函数
    Box(int i1,int j1,int di1) {            //重载构造函数
        i=i1; j=j1; di=di1;
    }
};
int mgpath(int xi,int yi,int xe,int ye) {   //求一条从(xi,yi)到(xe,ye)的迷宫路径
    int i,j,di,i1,j1;
    bool find;
    Box b,b1;
    stack<Box> st,st1;                      //定义一个顺序栈
    b=Box(xi,yi,-1);                        //建立入口方块对象
    st.push(b);                             //入口方块进栈
    mg[xi][yi]=-1;                          //为避免来回找相邻方块,置 mg 值为-1
    while (!st.empty()) {                   //栈不空时循环
        b=st.top();                         //取栈顶方块,称为当前方块
        if (b.i==xe && b.j==ye) {           //找到了出口,输出栈中的所有方块构成一条路径
            cnt++;
            printf("    迷宫路径%d: ",cnt);
```

```
            while(!st.empty()) {                    //将 st 的所有方块出栈并进栈 st1
                st1.push(st.top());
                st.pop();
            }
            while (!st1.empty()) {                  //输出一条迷宫路径
                b1=st1.top(); st1.pop();
                st.push(b1);                        //恢复 st 栈
                printf("[%d,%d] ",b1.i,b1.j);
            }
            printf("\n");
            mg[b.i][b.j]=0;                         //让该位置变为其他路径可走方块
            st.pop();                               //退栈
        }
        else {
            find=false;                             //继续找路径
            di=b.di;
            while (di<3 && find==false) {           //找 b 的一个相邻可走方块
                di++;                               //找下一个方位的相邻方块
                i=b.i+dx[di]; j=b.j+dy[di];         //找 b 的 di 方位的相邻方块(i,j)
                if (i>=0 && i<m && j>=0 && j<n && mg[i][j]==0)
                    find=true;                      //(i,j)方块有效且可走
            }
            if (find) {                             //找到了一个相邻可走方块(i,j)
                st.top().di=di;                     //修改栈顶方块的 di 为新值
                b1=Box(i,j,-1);                     //建立相邻可走方块(i,j)的对象 b1
                st.push(b1);                        //b1 进栈
                mg[i][j]=-1;                        //为避免来回找相邻方块,置 mg 值为-1
            }
            else {                                  //没有路径可走,则退栈
                mg[b.i][b.j]=0;                     //恢复当前方块的迷宫值
                st.pop();                           //将栈顶方块退栈
            }
        }
    }
    return cnt;                                     //返回找到的迷宫路径数
}
int main() {
    int xi=1,yi=1,xe=3,ye=2;
    printf("\n  求(%d,%d)到(%d,%d)的迷宫路径\n",xi,yi,xe,ye);
    int cnt=mgpath(xi,yi,xe,ye);
    printf("  共有%d 条迷宫路径\n",cnt);
    return 0;
}
```

上述程序的执行结果如图 3.9 所示。

```
求<1,1>到<3,2>的迷宫路径
  迷宫路径 1: [1,1] [1,2] [1,3] [2,3] [3,3] [3,2]
  迷宫路径 2: [1,1] [1,2] [1,3] [2,3] [2,2] [3,2]
  迷宫路径 3: [1,1] [1,2] [1,3] [2,3] [2,2] [2,1] [3,1] [3,2]
  迷宫路径 4: [1,1] [1,2] [2,2] [2,3] [3,3] [3,2]
  迷宫路径 5: [1,1] [1,2] [2,2] [3,2]
  迷宫路径 6: [1,1] [1,2] [2,2] [2,1] [3,1] [3,2]
  迷宫路径 7: [1,1] [2,1] [2,2] [1,2] [1,3] [2,3] [3,3] [3,2]
  迷宫路径 8: [1,1] [2,1] [2,2] [2,3] [3,3] [3,2]
  迷宫路径 9: [1,1] [2,1] [2,2] [3,2]
  迷宫路径10: [1,1] [2,1] [3,1] [3,2]
共有10条迷宫路径
```

图 3.9　第 3 章应用实验题 1 的执行结果

2. 解：对于字符串 s，设对应的输出字符串为 mark，采用栈 st 来产生 mark。遍历字符 $s[i]$，遇到'('时将其下标 i 进栈，遇到')'时，若栈中存在匹配的'('，置 mark[i]=' '，否则置 mark[i]='?'。当 s 遍历完毕时，若 st 栈不空，则 st 栈中的所有左括号都是没有右括号匹配的，将相应位置 j 的 mark 值置为'$'。对应的实验程序 Exp2-2.cpp 如下：

```
#include<iostream>
#include<stack>
using namespace std;
string solve(string s) {                    //求解算法
    stack<int> st;                          //定义一个栈
    string mark(s.length(),'*');            //定义输出字符串
    for (int i=0;i<s.length();i++) {
        if (s[i]=='(') {                    //遇到'('则入栈
            st.push(i);                     //将'('的下标暂存在栈中
            mark[i]=' ';                    //对应输出字符串暂且为' '
        }
        else if (s[i]==')') {               //遇到')'
            if (st.empty())                 //栈空,即没有'('相匹配
                mark[i]='?';                //对应输出字符串改为'?'
            else {                          //有'('相匹配
                mark[i]=' ';                //对应输出字符串改为' '
                st.pop();                   //栈顶位置的左括号与其匹配,弹出已经匹配的左括号
            }
        }
        else                                //其他字符与括号无关
            mark[i]=' ';                    //对应输出字符串改为' '
    }
    while (!st.empty()) {                   //若栈不空,则都没有匹配的左括号
        mark[st.top()]='$';                 //对应输出字符串改为'$'
        st.pop();
    }
    return mark;
}
int main() {
    printf("\n");
    printf("   测试 1\n");
    string s="((ABCD(x)";
    cout << "    表达式: " << s << endl;
    string ans=solve(s);
    cout << "    结   果: " << ans << endl;
    printf("   测试 2\n");
    s=")(rttyy())sss)(";
    cout << "    表达式: " << s << endl;
    ans=solve(s);
    cout << "    结   果: " << ans << endl;
    return 0;
}
```

上述程序的执行结果如图 3.10 所示。

3. 解：用全局变量 Initcap 存放初始容量，队列中增加 capacity 属性表示队列的当前容量，增加 recap(newcap)方法用于将当前容量改变为 newcap。其过程如下：

① 当参数 newcap 正确时(newcap>n)，建立长度为 newcap 的列表 tmp。

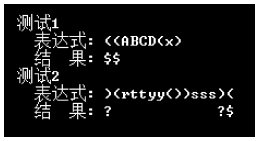


图 3.10 第 3 章应用实验题 2 的执行结果

② 出队 data 中的所有元素并依次存放到 tmp 中(从 tmp[1]开始)。

③ 置 data 为 tmp,队头指针 front 为 0,队尾指针 rear 为 n,新容量为 newcap。

在进队中队满和出队中满足指定的条件时调用 recap(newcap)方法。对应的实验程序 Exp2-3.cpp 如下:

```
#include<iostream>
using namespace std;
const int Initcap=3;                          //全局变量,初始容量为 3
template<typename T>
class CSqQueue {                              //非循环队列类
public:                                       //为了方便测试,将所有成员设置为公有的
    T *data;                                  //存放队中元素
    int capacity;                             //data 容量
    int length;                               //队中实际元素个数,即长度
    int front;                                //队头指针
    int rear;                                 //队尾指针
    CSqQueue() {                              //构造函数
        data=new T[Initcap];                  //为 data 分配容量为 Initcap 的空间
        capacity=Initcap;                     //设置容量
        front=rear=0;                         //初始化队头和队尾指针
        length=0;                             //初始化长度
    }
    ~CSqQueue() {                             //析构函数
        delete [] data;
    }
    void recap(int newcap) {                  //改变队列容量为 newcap
        if (newcap<length)
            throw("新容量大小错误");          //检测 newcap 参数的错误
        printf("  原容量=%d,原长度=%d,修改容量=%d",capacity,length,newcap);
        T *tmp=new T[newcap];                 //新建存放队列元素的空间
        int head=(front+1)%capacity;
        for (int i=0;i<length;i++) {          //出队所有元素存放到 tmp[1..length]中
            tmp[i+1]=data[head];              //从 tmp[1]开始,tmp[0]暂时不用
            head=(head+1)%capacity;
        }
        delete [] data;                       //释放原 data 空间
        data=tmp;                             //data 指向新空间
        front=0;                              //重置 front
        rear=length;                          //重置 rear
        capacity=newcap;                      //重置 capacity
    }
    bool empty() {                            //判队空运算
        return length==0;
    }
    bool full() {                             //判队满运算
```

```
        return length==capacity;
    }
    bool push(T e) {                                    //进队运算
        cout << "    进队" << e;
        if (full())                                     //队满上溢出
            recap(2 * capacity);                        //队满时倍增容量
        printf("\n");
        rear=(rear+1)%capacity;
        data[rear]=e;
        length++;                                       //增加一个元素
        return true;
    }
    bool pop(T &e) {                                    //出队运算
        if (empty()) return false;                      //队空下溢出
        front=(front+1)%capacity;
        e=data[front];
        length--;                                       //减少一个元素
        cout << "    出队" << e;
        if (capacity>Initcap && length==capacity/4)
            recap(capacity/2);                          //满足要求则容量减半
        printf("\n");
        return true;
    }
    bool gethead(T &e) {                                //取队头运算
        if (front==rear) return false;                  //队空下溢出
        int head=(front+1)%capacity;
        e=data[head];
        return true;
    }
};
int main() {
    int x;
    printf("\n");
    CSqQueue<int> qu;
    printf("  (1)进队 1,2\n");
    qu.push(1);
    qu.push(2);
    printf("  元素个数=%d,容量=%d\n",qu.length,qu.capacity);
    printf("  (2)进队 3~13\n");
    for (int i=3;i<=13;i++) qu.push(i);
    printf("  元素个数=%d,容量=%d\n",qu.length,qu.capacity);
    printf("  (3)出队所有元素\n");
    while (!qu.empty()) qu.pop(x);
    printf("  元素个数=%d,容量=%d\n",qu.length,qu.capacity);
    return 0;
}
```

上述程序的执行结果如图 3.11 所示。

4. 解：用只有尾结点指针 rear 的循环单链表作为队列存储结构，如图 3.12 所示，其中每个结点的类型为 LinkNode(同第 3 章基础实验题 4 中链队的结点类)。

在这样的链队中，队列为空时 rear=NULL，进队在链表的表尾进行，出队在链表的表头进行。例如，在空链队中进队 a、b、c 元素的结果如图 3.13(a)所示，出队两个元素后的结果如图 3.13(b)所示。

```
(1)进队1,2
  进队1
  进队2
元素个数=2,容量=3
(2)进队3～13
  进队3
  进队4  原容量=3,原长度=3,修改容量=6
  进队5
  进队6
  进队7  原容量=6,原长度=6,修改容量=12
  进队8
  进队9
  进队10
  进队11
  进队12
  进队13  原容量=12,原长度=12,修改容量=24
元素个数=13,容量=24
(3)出队所有元素
  出队1
  出队2
  出队3
  出队4
  出队5
  出队6
  出队7  原容量=24,原长度=6,修改容量=12
  出队8
  出队9
  出队10  原容量=12,原长度=3,修改容量=6
  出队11
  出队12  原容量=6,原长度=1,修改容量=3
  出队13
元素个数=0,容量=3
```

图 3.11　第 3 章应用实验题 3 的执行结果

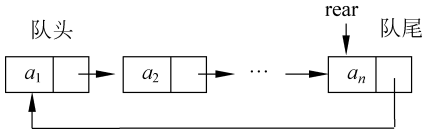


图 3.12　用只有尾结点指针的循环单链表作为队列存储结构

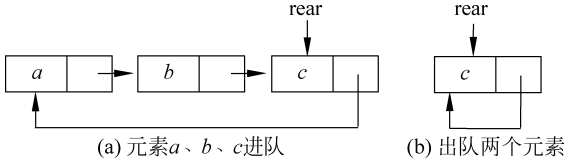


图 3.13　链队的进队和出队操作

对应的实验程序 Exp2-4.cpp 如下：

```
#include<iostream>
using namespace std;
template<typename T>
struct LinkNode{                                    //链栈结点类型
    T data;                                         //数据域
    LinkNode *next;                                 //指针域
    LinkNode():next(NULL){}                         //构造函数
    LinkNode(T d):data(d),next(NULL){}              //重载构造函数
};
template<typename T>
class LinkQueue{                                    //链队类模板
public:
    LinkNode<T> *rear;                              //链队的尾结点指针
    LinkQueue():rear(NULL){}                        //构造函数
    ~LinkQueue(){                                   //析构函数
        if(rear==NULL) return;
        LinkNode<T> *pre,*p;
```

```
        pre=rear; p=pre->next;
        while (p!=rear) {                          //用 p 遍历结点并释放其前驱结点
            delete pre;                            //释放 pre 结点
            pre=p; p=p->next;                      //pre、p 同步后移一个结点
        }
        delete pre;                                //p 等于 rear 时 pre 指向尾结点,此时释放尾结点
    }
    bool empty() {                                 //判队空运算
        return rear==NULL;
    }
    bool push(T e) {                               //进队运算
        LinkNode<T> *p=new LinkNode<T>(e);
        if (rear==NULL) {                          //链队为空的情况
            rear=p;                                //新结点既是队首结点又是队尾结点
            rear->next=rear;                       //构成循环单链表
        }
        else {                                     //链队不空的情况
            p->next=rear->next;                    //将 p 结点插入 rear 结点之后
            rear->next=p;
            rear=p;                                //让 rear 指向 p 结点
        }
        return true;
    }
    bool pop(T &e) {                               //出队运算
        if (empty()) return false;
        if (rear->next==rear) {                    //原链队只有一个结点
            e=rear->data;                          //取该结点值
            rear=NULL;                             //置为空队
        }
        else {                                     //原链队有多个结点
            e=rear->next->data;                    //取队头结点值
            rear->next=rear->next->next;           //删除队头结点
        }
        return true;
    }
    bool gethead(T &e) {                           //取队头运算
        if (empty()) return false;
        e=rear->next->data;
        return true;
    }
};
int main() {
    LinkQueue<char> qu;                            //定义一个字符队 qu
    char e;
    cout << "\n  建立一个空队 qu\n";
    cout << "  队 qu" << (qu.empty()?"空":"不空") << endl;
    cout << "  元素 a 进队\n"; qu.push('a');
    cout << "  元素 b 进队\n"; qu.push('b');
    qu.gethead(e); cout << "  队头元素: " << e << endl;
    cout << "  元素 c 进队\n"; qu.push('c');
    cout << "  元素 d 进队\n"; qu.push('d');
    cout << "  元素 e 进队\n"; qu.push('e');
    cout << "  队 qu" << (qu.empty()?"空":"不空") << endl;
```

```
    qu.gethead(e); cout << "  队头元素: " << e << endl;
    cout << "  所有元素出队次序: ";
    while (!qu.empty()) {                                    //队不空时循环
        qu.pop(e);                                           //出队元素 e
        cout << e << " ";                                    //输出元素 e
    }
    cout << endl;
    cout << "  销毁队 qu" << endl;
    return 0;
}
```

上述程序的执行结果如图 3.14 所示。

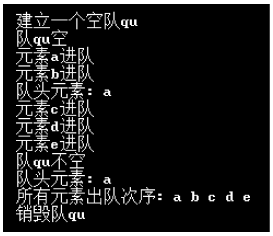


图 3.14　第 3 章应用实验题 4 的执行结果

5. 解：由于栈的特点是先进后出，而队列的特点是先进先出，在用两个栈 st1、st2 模拟队列时，st1 栈负责“进队”，st2 栈负责“出队”(反向)，在 st1 和 st2 都非空时保证 st2 中的元素都是先于 st1 中的元素进队。

队空的条件：栈 st1 和 st2 均为空。

元素 e 进队的操作：此时只需要直接将 e 进到 st1 栈，如图 3.15(a)所示(这里没有考虑栈满，若考虑栈满的情况，当 st1 栈满时先将 st1 的所有元素出栈并进栈 st2，再将 e 进到 st1 栈)。

出队元素 e 的操作：若 st1 和 st2 均为空，则返回 false；若 st2 不空，则 st2 出栈元素 e；若 st2 空但栈 st1 不空，则将栈 st1 中的所有元素出栈并进到 st2 栈中，再从 st2 出栈元素 e，如图 3.15(b)所示。

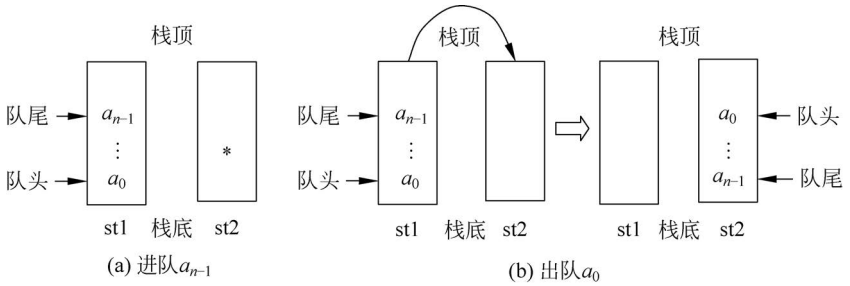


图 3.15　两个栈模拟队列

对应的实验程序 Exp2-5.cpp 如下：

```
#include <iostream>
#include <stack>
using namespace std;
```

```
template < typename T >
class QUEUE {                                    //两个栈模拟的队列
    stack < T > st1,st2;
public:
    bool empty() {                               //判队空运算
        return st1.empty() && st2.empty();
    }
    void push(T e) {                             //进队运算
        st1.push(e);
    }
    bool pop(T &e) {                             //出队运算
        if (empty()) return false;
        if (!st2.empty()) {                      //st2 不空时从 st2 出栈元素 e
            e=st2.top();                         //st2 出栈元素 e
            st2.pop();
        }
        else {                                   //st2 空时
            while (!st1.empty()) {               //将栈 st1 中的所有元素出栈并进到 st2 栈中
                st2.push(st1.top());
                st1.pop();
            }
            e=st2.top();                         //st2 出栈元素 e
            st2.pop();
        }
        return true;
    }
};
int main() {
    QUEUE < int > qu;                            //定义一个整数队 qu
    int e;
    cout << "\n   建立一个空队 qu\n";
    cout << "   队 qu" << (qu.empty()?"空":"不空") << endl;
    cout << "   元素 1 进队\n"; qu.push(1);
    cout << "   元素 2 进队\n"; qu.push(2);
    qu.pop(e); cout << "   出队元素：" << e << endl;
    cout << "   元素 3 进队\n"; qu.push(3);
    cout << "   元素 4 进队\n"; qu.push(4);
    qu.pop(e); cout << "   出队元素：" << e << endl;
    cout << "   元素 5 进队\n"; qu.push(5);
    cout << "   队 qu" << (qu.empty()?"空":"不空") << endl;
    cout << "   其他元素出队次序：";
    while (!qu.empty()) {                        //队不空时循环
        qu.pop(e);                               //出队元素 e
        cout << e << " ";                        //输出元素 e
    }
    cout << endl;
    cout << "   销毁队 qu" << endl;
    return 0;
}
```

上述程序的执行结果如图 3.16 所示。

6. 解：由于队列不会改变顺序，在用两个队列 qu1 和 qu2 模拟栈时采用来回倒的方法，保证一个队列是空的，用空队列临时存储队尾外的所有元素。

```
建立一个空队qu
队qu空
元素1进队
元素2进队
出队元素: 1
元素3进队
元素4进队
出队元素: 2
元素5进队
队qu不空
其他元素出队次序: 3 4 5
销毁队qu
```

图 3.16　第 3 章应用实验题 5 的执行结果

栈空的条件：两个队列均为空。

元素 e 进栈的操作：总有一个队列是空的，将 e 进到非空队中。假设 qu1 非空，进栈 a_{n-1} 的操作如图 3.17(a)所示。

出队元素 e 的操作：总有一个队列是空的，假设 qu1 非空，先将 qu1 中的前 $n-1$ 个元素 $a_0 \sim a_{n-2}$ 出队并进到 qu2，如图 3.17(b)所示，再从 qu1 出队元素 a_{n-1}。qu2 非空的操作与之类似。

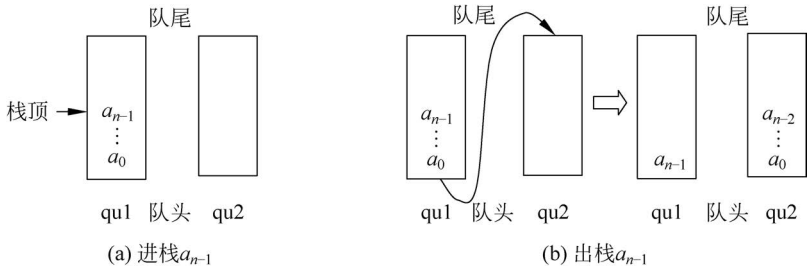


图 3.17　两个队列模拟栈

对应的实验程序 Exp2-6.cpp 如下：

```
#include<iostream>
#include<queue>
using namespace std;
template<typename T>
class STACK{                                    //两个队列模拟的栈
    queue<T> qu1,qu2;
public:
    bool empty(){                               //判栈空运算
        return qu1.empty() && qu2.empty();
    }
    void push(T e){                             //进栈运算
        if (!qu1.empty()) qu1.push(e);
        else qu2.push(e);
    }
    bool pop(T &e){                             //出栈运算
        if (empty()) return false;
        if (!qu1.empty()){                      //qu1 不空时
            while (qu1.size()>1){               //qu1 中的前 n-1 个元素出队并进到 qu2
                qu2.push(qu1.front());
                qu1.pop();
```

```
            }
            e=qu1.front(); qu1.pop();                    //qu1 出队最后一个元素 e
        }
        else {                                           //qu1 空时
            while (qu2.size()>1) {                       //qu2 中的前 n-1 个元素出队并进到 qu1
                qu1.push(qu2.front());
                qu2.pop();
            }
            e=qu2.front(); qu2.pop();
        }
        return true;
    }
};
int main() {
    STACK<int> st;                                       //定义一个整数栈 st
    int e;
    cout << "\n  建立一个空栈 st\n";
    cout << "  栈 st" << (st.empty()?"空":"不空") << endl;
    cout << "  元素 1 进栈\n"; st.push(1);
    cout << "  元素 2 进栈\n"; st.push(2);
    st.pop(e); cout << "  出栈元素: " << e << endl;
    cout << "  元素 3 进栈\n"; st.push(3);
    cout << "  元素 4 进栈\n"; st.push(4);
    st.pop(e); cout << "  出栈元素: " << e << endl;
    cout << "  元素 5 进栈\n"; st.push(5);
    cout << "  栈 st" << (st.empty()?"空":"不空") << endl;
    cout << "  其他元素出栈次序: ";
    while (!st.empty()) {                                //队不空时循环
        st.pop(e);                                       //出队元素 e
        cout << e << " ";                                //输出元素 e
    }
    cout << endl;
    cout << "  销毁栈 st" << endl;
    return 0;
}
```

上述程序的执行结果如图 3.18 所示。

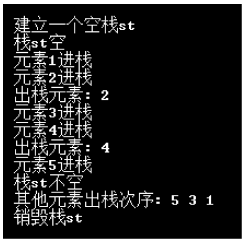


图 3.18　第 3 章应用实验题 6 的执行结果