成都理工大學
CHENGDU UNIVERSITY OF TECHNOLOGY

数据结构与程序设计
实验指导书

编写人：朱杰

二〇二二年

目　录

实验一 线性表的应用

一、 实验目的：

1. 掌握线性表的逻辑结构和存储结构特点；
2. 掌握线性表的基本操作，如建立、查找、插入和删除等。

二、 问题描述

智能家居系统创建一个家居环境参数表，包含“日期、时间、地点、温度、湿度”等信息。程序能够完成如下功能：

(1) 能够逐条输入信息，创建表；

(2) 能够显示表中的所有信息；

(3) 根据时间和地点进行查找，返回相关参数信息；

(4) 给定一条环境参数信息，按照日期和时间顺序插入到表中指定的位置；

(5) 删除指定日期的记录。

三、线性表存储结构表示示例

1. 顺序表结构

```
typedef struct {
    string id;//ISBN
    string name;//书名
    double price;//定价
} Book;
typedef struct {
    Book *elem; //存储空间的基地址
    int length; //当前长度
} SqList;
```

2. 单链表结构

```
typedef struct {
    string id;//ISBN
```

```
    string name;//书名
    double price;//定价
} Book;
typedef struct LNode {
    Book data; //结点的数据域
    struct LNode *next; //结点的指针域
} LNode, *LinkList; //LinkList 为指向结构体 LNode 的指针类型
```

四、参考算法

```
Status InitList_Sq(SqList &L) { //算法 2.1  顺序表的初始化
    //构造一个空的顺序表 L
    L.elem = new Book[MAXSIZE]; //为顺序表分配一个大小为 MAXSIZE 的数组空间
    if (!L.elem)
        exit(OVERFLOW); //存储分配失败退出
    L.length = 0; //空表长度为 0
    return OK;
}
```

```
Status GetElem(SqList L, int i, Book &e) {//算法 2.2    顺序表的取值
    if (i < 1 || i > L.length)
        return ERROR; //判断 i 值是否合理，若不合理，返回 ERROR
    e = L.elem[i - 1]; //elem[i-1]单元存储第 i 个数据元素
    return OK;
}
```

```
int LocateElem_Sq(SqList L, double e) { //算法 2.3  顺序表的查找
    //顺序表的查找
    for (int i = 0; i < L.length; i++)
        if (L.elem[i].price == e)
            return i + 1;//查找成功，返回序号 i+1
    return 0;//查找失败，返回 0
}
```

```
Status ListInsert_Sq(SqList &L, int i, Book e) { //算法 2.4  顺序表的插入
    //在顺序表 L 中第 i 个位置之前插入新的元素 e
    //i 值的合法范围是 1<=i<=L.length+1
    if ((i < 1) || (i > L.length + 1))
        return ERROR; //i 值不合法
    if (L.length == MAXSIZE)
```

```
        return ERROR; //当前存储空间已满
    for (int j = L.length - 1; j >= i - 1; j--)
        L.elem[j + 1] = L.elem[j]; //插入位置及之后的元素后移
    L.elem[i - 1] = e; //将新元素 e 放入第 i 个位置
    ++L.length; //表长增 1
    return OK;
}
```

```
Status ListDelete_Sq(SqList &L, int i) { //算法 2.5  顺序表的删除
    //在顺序表 L 中删除第 i 个元素，并用 e 返回其值
    //i 值的合法范围是 1<=i<=L.length
    if ((i < 1) || (i > L.length))
        return ERROR; //i 值不合法
    for (int j = i; j <= L.length; j++)
        L.elem[j - 1] = L.elem[j]; //被删除元素之后的元素前移
    --L.length; //表长减 1
    return OK;
}
```

```
Status GetElem_L(LinkList L, int i, Book &e) { //算法 2.7  单链表的取值
    //在带头结点的单链表 L 中查找第 i 个元素
    //用 e 返回 L 中第 i 个数据元素的值
    int j;
    LinkList p;
    p = L->next;
    j = 1; //初始化，p 指向第一个结点，j 为计数器
    while (j < i && p) { //顺链域向后扫描，直到 p 指向第 i 个元素或 p 为空
        p = p->next; //p 指向下一个结点
        ++j; //计数器 j 相应加 1
    }
    if (!p || j > i)
        return ERROR; //i 值不合法 i＞n 或 i<=0
    e = p->data; //取第 i 个结点的数据域
    return OK;
} //GetElem_L
```

```
LNode *LocateElem_L(LinkList L, int e) { //算法 2.8  按值查找
    //在带头结点的单链表 L 中查找值为 e 的元素
    LinkList p;
    p = L->next;
    while (p && p->data.price != e)//顺链域向后扫描，直到 p 为空或 p 所指结点的数据域等于 e
```

```
        p = p->next; //p 指向下一个结点
    return p; //查找成功返回值为 e 的结点地址 p，查找失败 p 为 NULL
} //LocateElem_L
```

```
Status ListInsert_L(LinkList &L, int i, Book &e) { //算法 2.9  单链表的插入
    //在带头结点的单链表 L 中第 i 个位置插入值为 e 的新结点
    int j;
    LinkList p, s;
    p = L;
    j = 0;
    while (p && j < i - 1) {
        p = p->next;
        ++j;
    }//查找第 i?1 个结点，p 指向该结点
    if (!p || j > i - 1)
        return ERROR; //i＞n+1 或者 i＜1
    s = new LNode; //生成新结点*s
    s->data = e; //将结点*s 的数据域置为 e
    s->next = p->next; //将结点*s 的指针域指向结点 ai
    p->next = s; //将结点*p 的指针域指向结点*s
    ++length;
    return OK;
} //ListInsert_L
```

```
void CreateList_H(LinkList &L, int n) { //算法 2.11  前插法创建单链表
    //逆位序输入 n 个元素的值，建立到头结点的单链表 L
    LinkList p;
    L = new LNode;
    L->next = NULL; //先建立一个带头结点的空链表
    length = 0;
    fstream file;
    file.open("book.txt");
    if (!file) {
        cout << "未找到相关文件，无法打开！" << endl;
        exit(ERROR);
    }
    file >> head_1 >> head_2 >> head_3;
    while (!file.eof()) {
        p = new LNode; //生成新结点*p
        file >> p->data.id >> p->data.name >> p->data.price; //输入元素值赋给新结点*p
的数据域
```

```
            p->next = L->next;
            L->next = p; //将新结点*p 插入到头结点之后
            length++;//同时对链表长度进行统计
        }
        file.close();
} //CreateList_F
```

实验二 栈和队列的应用

一、 实验目的：

1、握栈和队列的逻辑结构及存储结构；

2、运用栈和队列原理完成设计的内容

二、 问题描述

1、完成数字十进制到八进制的转换。

输入示例：

请输入需转换的数的个数：

3

请输入需转换的数：

28，58，190

输出示例：

转换结果为：

1、 34

2、 72

3、 276

2、银行排队系统实现

功能要求：

（1） 客户进入排队系统；

（2） 客户离开；

（3） 查询当前客户前面还有几人；

（4） 查询截至目前总共办理多少客户。

输出要求：每进行一次操作后，输出当前排队成员情况。

三、栈与队列存储结构表示示例

1. 顺序栈的表示

```
typedef struct {
    SElemType *base;//栈底指针
    SElemType *top;//栈顶指针
    int stacksize;//栈可用的最大容量
} SqStack;
```

2. 链栈的表示

```
typedef struct StackNode {
    SElemType data;
    struct StackNode *next;
} StackNode, *LinkStack;
```

3. 队列的顺序存储表示

```
typedef struct {
    char name[20]; //姓名
    char sex[8]; //性别，'F'表示女性，'M'表示男性
} Person;
//- - - - - 队列的顺序存储结构- - - - -
typedef struct {
    Person *base; //队列中数据元素类型为 Person
    int front; //头指针
    int rear; //尾指针
} SqQueue;
```

4. 队列的链式存储表示

```
//- - - - - 队列的链式存储结构- - - - -
typedef struct QNode {
    char data;
    struct QNode *next;
} QNode, *QueuePtr;
typedef struct {
    QueuePtr front; //队头指针
    QueuePtr rear; //队尾指针
} LinkQueue;
```

四、参考算法

参考并应用课程群提供的堆栈操作程序 stack.h 和队列操作程序 queue.h。部分参考示例代码如下：

```
算法 3.1　顺序栈的初始化
Status InitStack(SqStack &S) {
    //构造一个空栈 S
    S.base = new SElemType[MAXSIZE];//为顺序栈动态分配一个最大容量为 MAXSIZE 的数组空间
    if (!S.base)
        exit(OVERFLOW); //存储分配失败
    S.top = S.base; //top 初始为 base，空栈
    S.stacksize = MAXSIZE; //stacksize 置为栈的最大容量 MAXSIZE
    return OK;
}
```

```
//算法 3.2　顺序栈的入栈
Status Push(SqStack &S, SElemType e) {
    // 插入元素 e 为新的栈顶元素
    if (S.top - S.base == S.stacksize)
        return ERROR; //栈满
    *(S.top++) = e; //元素 e 压入栈顶，栈顶指针加 1
    return OK;
}
```

```
//算法 3.3　顺序栈的出栈
Status Pop(SqStack &S, SElemType &e) {
    //删除 S 的栈顶元素，用 e 返回其值
    if (S.base == S.top)
        return ERROR;//栈空
    e = *(--S.top); //栈顶指针减 1，将栈顶元素赋给 e
    return OK;
}
```

```
//算法 3.4　取顺序栈的栈顶元素
char GetTop(SqStack S) {//返回 S 的栈顶元素，不修改栈顶指针
    if (S.top != S.base) //栈非空
        return *(S.top - 1); //返回栈顶元素的值，栈顶指针不变
}
```

```
//算法 3.5　链栈的初始化
Status InitStack(LinkStack &S) { // 构造一个空栈 S，栈顶指针置空
```

```
    S = NULL;
    return OK;
}
```

```
//算法 3.6　链栈的入栈
Status Push(LinkStack &S, SElemType e) {//在栈顶插入元素 e
    LinkStack p;
    p = new StackNode; //生成新结点
    p->data = e; //将新结点数据域置为 e
    p->next = S; //将新结点插入栈顶
    S = p; //修改栈顶指针为 p
    return OK;
}
```

```
//算法 3.7　链栈的出栈
Status Pop(LinkStack &S, SElemType &e) {//删除 S 的栈顶元素，用 e 返回其值
    LinkStack p;
    if (S == NULL)
        return ERROR; //栈空
    e = S->data; //将栈顶元素赋给 e
    p = S; //用 p 临时保存栈顶元素空间，以备释放
    S = S->next; //修改栈顶指针
    delete p; //释放原栈顶元素的空间
    return OK;
}
```

```
//算法 3.8　取链栈的栈顶元素
SElemType GetTop(LinkStack S) {//返回 S 的栈顶元素，不修改栈顶指针
    if (S != NULL) //栈非空
        return S->data; //返回栈顶元素的值，栈顶指针不变
}
```

```
//后插法创建链表算法
void CreateList_L(LinkList &L, int n) {
    L = new LNode;
    L->next = NULL;
    LNode *p, *r;
    r = L;
    for (int i = 0; i < n; i++) {
        p = new LNode;
        cin >> p->data;
        p->next = NULL;
```

```
            r->next = p;
            r = p;
        }
}
```

实验三 二叉树的应用

一、 实验目的：

1、掌握二叉树的定义和存储表示，掌握二叉树建立的算法；

2、掌握二叉树的遍历（先序、中序、后序）算法

二、 问题描述

1、查找并绘制自己家族的族谱二叉树；

2、族谱二叉树的建立（树的深度要>=4）；

3、三种不同遍历算法遍历此二叉树；

4、统计二叉树的深度，输出叶子结点的信息。

三、树的存储结构表示示例

二叉链表表示

```
typedef struct BiNode{                    //二叉链表定义
    char data;
    struct BiNode *lchild,*rchild;
}BiTNode,*BiTree;
```

四、参考算法

```
//用算法 5.3  先序遍历的顺序建立二叉链表
void CreateBiTree(BiTree &T){
    //按先序次序输入二叉树中结点的值（一个字符），创建二叉链表表示的二叉树 T
    char ch;
    cin >> ch;
    if(ch=='#')   T=NULL;               //递归结束，建空树
    else{
        T=new BiTNode;
        T->data=ch;                     //生成根结点
        CreateBiTree(T->lchild); //递归创建左子树
        CreateBiTree(T->rchild); //递归创建右子树
    }                                   //else
```

```
}
```

```
void InOrderTraverse(BiTree T){
    //中序遍历二叉树 T 的递归算法
    if(T){
        InOrderTraverse(T->lchild);
        cout << T->data;
        InOrderTraverse(T->rchild);
    }
}
```

```
int Depth(BiTree T)
{
    int m,n;
    if(T == NULL ) return 0;            //如果是空树，深度为 0，递归结束
    else
    {
        m=Depth(T->lchild);             //递归计算左子树的深度记为 m
        n=Depth(T->rchild);             //递归计算右子树的深度记为 n
        if(m>n) return(m+1);        //二叉树的深度为 m 与 n 的较大者加 1
        else return (n+1);
    }
}
```

实验四 图的应用

一、 实验目的：

1、掌握图的基本概念；

2、掌握图的遍历算法，最短路径算法。

二、 问题描述

1、绘制基于理工的地图网（结点不少于 6），注：边的权值代表距离；实现网的创建；

2、按照深度遍历和广度遍历算法输出结点信息；

3、实现从西门到香樟的最短路径算法。

三、树的存储结构表示

1. 图的邻接矩阵表示

```
//- - - - -图的邻接矩阵存储表示- - - - -
typedef struct{
    VerTexType vexs[MVNum];                 //顶点表
    ArcType arcs[MVNum][MVNum];             //邻接矩阵
    int vexnum,arcnum;                      //图的当前点数和边数
}AMGraph;
```

2. 邻接表存储表示

```
//- - - - -图的邻接表存储表示- - - - -
typedef struct ArcNode{                     //边结点
    int adjvex;                             //该边所指向的顶点的位置
    struct ArcNode *nextarc;                //指向下一条边的指针
    OtherInfo info;                         //和边相关的信息
}ArcNode;

typedef struct VNode{
    VerTexType data;                        //顶点信息
```

```
    ArcNode *firstarc;                          //指向第一条依附该顶点的边的指针
}VNode, AdjList[MVNum];                         //AdjList 表示邻接表类型
typedef struct{
    AdjList vertices;                           //邻接表
    int vexnum, arcnum;                         //图的当前顶点数和边数
}ALGraph;
```

四、参考算法

```
int CreateUDN(AMGraph &G){
    //采用邻接矩阵表示法，创建无向网 G
    int i , j , k;
    cout <<"请输入总顶点数，总边数，以空格隔开：";
     cin >> G.vexnum >> G.arcnum;                                   //输入总顶点数，总边数
    cout << endl;

    cout << "输入点的名称，如 a" << endl;

     for(i = 0; i < G.vexnum; ++i){
        cout << "请输入第" << (i+1) << "个点的名称:";
        cin >> G.vexs[i];                                           //依次输入点的信息
    }
    cout << endl;
     for(i = 0; i < G.vexnum; ++i)                                  //初始化邻接矩阵，边的权
值均置为极大值 MaxInt
        for(j = 0; j < G.vexnum; ++j)
            G.arcs[i][j] = MaxInt;
    cout << "输入边依附的顶点及权值，如  a b 5" << endl;
    for(k = 0; k < G.arcnum;++k){                                   //构造邻接矩阵
        VerTexType v1 , v2;
        ArcType w;
        cout << "请输入第" << (k + 1) << "条边依附的顶点及权值:";
        cin >> v1 >> v2 >> w;                                       //输入一条边依附的顶点
及权值
        i = LocateVex(G, v1);    j = LocateVex(G, v2);        //确定 v1 和 v2 在 G 中的位置，
即顶点数组的下标
        G.arcs[i][j] = w;                                           //边<v1, v2>的权值置为 w
        G.arcs[j][i] = G.arcs[i][j];                          //置<v1, v2>的对称边<v2, v1>的权
```

```
值为 w
    }//for
    return OK;
}//CreateUDN
```

```
int CreateUDG(ALGraph &G){
    //采用邻接表表示法，创建无向图 G
    int i , k;
    cout <<"请输入总顶点数，总边数中间以空格隔开:";
    cin >> G.vexnum >> G.arcnum;                    //输入总顶点数，总边数
     cout << endl;

    cout << "输入点的名称，如 a " <<endl;
    for(i = 0; i < G.vexnum; ++i){                  //输入各点，构造表头结点表
        cout << "请输入第" << (i+1) << "个点的名称:";
        cin >> G.vertices[i].data;                  //输入顶点值
        G.vertices[i].firstarc=NULL;                //初始化表头结点的指针域为 NULL
     }//for
    cout << endl;

    cout << "请输入一条边依附的顶点,如 a b" << endl;
    for(k = 0; k < G.arcnum;++k){                   //输入各边，构造邻接表
        VerTexType v1 , v2;
        int i , j;
        cout << "请输入第" << (k + 1) << "条边依附的顶点:";
        cin >> v1 >> v2;                            //输入一条边依附的两个顶点
        i = LocateVex(G, v1);   j = LocateVex(G, v2);
        //确定 v1 和 v2 在 G 中位置，即顶点在 G.vertices 中的序号

        ArcNode *p1=new ArcNode;                        //生成一个新的边结点*p1
        p1->adjvex=j;                               //邻接点序号为 j
        p1->nextarc= G.vertices[i].firstarc;   G.vertices[i].firstarc=p1;
        //将新结点*p1 插入顶点 vi 的边表头部

        ArcNode *p2=new ArcNode;                        //生成另一个对称的新的边结点
*p2
```

```
        p2->adjvex=i;                                   //邻接点序号为 i
        p2->nextarc= G.vertices[j].firstarc;    G.vertices[j].firstarc=p2;
        //将新结点*p2 插入顶点 vj 的边表头部
    }//for
    return OK;
}//CreateUDG
```

```
void DFS(Graph G, int v){              //从第 v 个顶点出发递归地深度优先遍历图 G
    cout << G.vexs[v] << "      ";   visited[v] = true;        //访问第 v 个顶点，并置访问标志数组相应分量值为 true
    int w;
    for(w = FirstAdjVex(G, v);   w >= 0;   w = NextAdjVex(G, v, w))
        //依次检查 v 的所有邻接点 w ，FirstAdjVex(G, v)表示 v 的第一个邻接点
        //NextAdjVex(G, v, w)表示 v 相对于 w 的下一个邻接点，w≥0 表示存在邻接点
        if(!visited[w]) DFS(G, w);                      //对 v 的尚未访问的邻接顶点 w 递归调用 DFS
}//DFS
```

```
void ShortestPath_DIJ(AMGraph G, int v0){
    //用 Dijkstra 算法求有向网 G 的 v0 顶点到其余顶点的最短路径
    int v , i , w , min;
    int n = G.vexnum;                                   //n 为 G 中顶点的个数

    for(v = 0; v < n; ++v){                             //n 个顶点依次初始化
        S[v] = false;                                   //S 初始为空集
        D[v] = G.arcs[v0][v];                           //将 v0 到各个终点的最短路径长度初始化为弧上的权值
        if(D[v] < MaxInt)   Path [v] = v0;              //如果 v0 和 v 之间有弧，则将 v 的前驱置为 v0
        else Path [v] = -1;                             //如果 v0 和 v 之间无弧，则将 v 的前驱置为-1
    }//for

    S[v0]=true;                                         //将 v0 加入 S
    D[v0]=0;                                            //源点到源点的距离为 0

    /*－初始化结束，开始主循环，每次求得 v0 到某个顶点 v 的最短路径，将 v 加到 S 集－*/
    for(i = 1;i < n; ++i){                              //对其余 n-1 个顶点，依次进行
```

```
计算
          min= MaxInt;
          for(w = 0; w < n; ++w)
              if(!S[w] && D[w] < min){          //选择一条当前的最短路径，终点为 v
                  v = w;
                  min = D[w];
              }//if
          S[v]=true;                                          //将 v 加入 S
          for(w = 0;w < n; ++w)    //更新从 v0 出发到集合 V?S 上所有顶点的最短路径长度
              if(!S[w] && (D[v] + G.arcs[v][w] < D[w])){
                  D[w] = D[v] + G.arcs[v][w];                 //更新 D[w]
                  Path [w] = v;                                   //更改 w 的前驱为 v
              }//if
      }//for
}//ShortestPath_DIJ
```

实验五 查找的应用

一、 实验目的：

1、掌握各种查找方法及适用场合，并能在解决实际问题时灵活应用。

2、增强上机编程调试能力。

二、 问题描述

1、分别利用顺序查找和折半查找方法完成查找。

有序表（3,4,5,7,24,30,42,54,63,72,87,95）

输入示例：

请输入查找元素：52

输出示例：

顺序查找：

第一次比较元素 95

第二次比较元素 87 ……..

查找成功，i=**/查找失败

折半查找：

第一次比较元素 30

第二次比较元素 63 …..

2、利用序列（12,7,17,11,16,2,13,9,21,4）建立二叉排序树，并完成指定元素的查询。

输入输出示例同题 1 的要求。

三、存储结构表示示例

查找表的存储表示

```
typedef struct{
    int key;//关键字域
}ElemType;

typedef struct{
```

```
    ElemType *R;
    int length;
}SSTable;
```

树表的存储表示

```
typedef struct ElemType{
    char key;
}ElemType;

typedef struct BSTNode{
    ElemType data;  //结点数据域
    BSTNode *lchild,*rchild;//左右孩子指针
}BSTNode,*BSTree;
```

四、参考算法

```
int Search_Seq(SSTable ST, int key){
    //在顺序表 ST 中顺序查找其关键字等于 key 的数据元素。若找到，则函数值为
    //该元素在表中的位置，否则为 0
      for (int i=ST.length; i>=1; --i)
                if (ST.R[i].key==key) return i;        //从后往前找
      return 0;
   }// Search_Seq
```

```
int Search_Bin(SSTable ST,int key) {
   // 在有序表 ST 中折半查找其关键字等于 key 的数据元素。若找到，则函数值为
   // 该元素在表中的位置，否则为 0
   int low=1,high=ST.length;                              //置查找区间初值
   int   mid;
   while(low<=high) {
        mid=(low+high) / 2;
       if (key==ST.R[mid].key)   return mid;                  //找到待查元素
       else if (key<ST.R[mid].key)   high = mid -1;           //继续在前一子表进行查找
       else   low =mid +1;                                    //继续在后一子表进行查找
   }//while
   return 0;                                              //表中不存在待查元素
}// Search_Bin
```

```
//算法 7.6   二叉排序树的创建
void CreateBST(BSTree &T ) {
```

```
  //依次读入一个关键字为 key 的结点，将此结点插入二叉排序树 T 中
  T=NULL;
  ElemType e;
  cin>>e.key;              //???
  while(e.key!=ENDFLAG){      //ENDFLAG 为自定义常量，作为输入结束标志
    InsertBST(T, e);          //将此结点插入二叉排序树 T 中
    cin>>e.key;           //???
  }//while
}//CreatBST
```

```
//算法 7.4   二叉排序树的递归查找
BSTree SearchBST(BSTree T,char key) {
  //在根指针 T 所指二叉排序树中递归地查找某关键字等于 key 的数据元素
  //若查找成功，则返回指向该数据元素结点的指针，否则返回空指针
  if((!T)|| key==T->data.key) return T;                          //查找结束
  else if (key<T->data.key)    return SearchBST(T->lchild,key); //在左子树中继续查找
  else return SearchBST(T->rchild,key);                          //在右子树中继续查找
} // SearchBST
```

实验六 排序的应用

一、 实验目的：

1、掌握直接插入排序、折半插入排序、冒泡排序、快速排序和归并排序等排序算法的思想。

2、实现直接插入排序、折半插入排序、冒泡排序、快速排序和归并排序等排序算法的编程应用。

二、 问题描述

实现数据的折半插入排序、冒泡排序、快速排序和二路归并排序。

输入实例：

请输入待排序数据数目：

3

请输入待排序数据：23,6,45

输出示例：

折半插入排序：

比较次数

移动元素次数

排序结果 6,23,45。

三、参考算法

```
void BInsertSort(SqList &L){
    //对顺序表 L 做折半插入排序
    int i,j,low,high,m;
    for(i=2;i<=L.length;++i)
    {
        L.r[0]=L.r[i];                          //将待插入的记录暂存到监视哨中
        low=1; high=i-1;                            //置查找区间初值
        while(low<=high)
        {                                       //在 r[low..high]中折半查找插入的位置
            m=(low+high)/2;                         //折半
```

```
            if(L.r[0].key<L.r[m].key)    high=m-1; //插入点在前一子表
            else    low=m+1;                          //插入点在后一子表
        }//while
        for(j=i-1;j>=high+1;--j)    L.r[j+1]=L.r[j];   //记录后移
        L.r[high+1]=L.r[0];                           //将 r[0]即原 r[i]，插入到正确位置
    }                                                 //for
}                                                     //BInsertSort
```

```
void BubbleSort(SqList &L)
{
    //对顺序表 L 做冒泡排序
    int m,j,flag;
    ElemType t;
     m=L.length-1; flag=1;                //flag 用来标记某一趟排序是否发生交换
     while((m>0)&&(flag==1))
    {
        flag=0;                           //flag 置为 0，如果本趟排序没有发生交换，则
不会执行下一趟排序
          for(j=1;j<=m;j++)
            if(L.r[j].key>L.r[j+1].key)
            {
                flag=1;                   //flag 置为 1，表示本趟排序发生了交换
                t=L.r[j];L.r[j]=L.r[j+1];L.r[j+1]=t; //交换前后两个记录
            }                             //if
        --m;
    }                                     //while
}
```

```
int Partition(SqList &L,int low,int high)
{
    //对顺序表 L 中的子表 r[low..high]进行一趟排序，返回枢轴位置
    int pivotkey;
    L.r[0]=L.r[low];                          //用子表的第一个记录做枢轴记录
    pivotkey=L.r[low].key;                    //枢轴记录关键字保存在 pivotkey 中
    while(low<high)
    {                                         //从表的两端交替地向中间扫描
        while(low<high&&L.r[high].key>=pivotkey) --high;
        L.r[low]=L.r[high];                   //将比枢轴记录小的记录移到低端
        while(low<high&&L.r[low].key<=pivotkey) ++low;
        L.r[high]=L.r[low];                   //将比枢轴记录大的记录移到高端
    }//while
```

```
    L.r[low]=L.r[0];                          //枢轴记录到位
    return   low;                             //返回枢轴位置
}//Partition

void QSort(SqList &L,int low,int high)
{   //调用前置初值：low=1; high=L.length;
    //对顺序表 L 中的子序列 L.r[low..high]做快速排序
    int pivotloc;
     if(low<high)
    {                                         //长度大于 1
        pivotloc=Partition(L,low,high);       //将 L.r[low..high]一分为二，pivotloc 是枢
轴位置
        QSort(L,low,pivotloc-1);              //对左子表递归排序
        QSort(L,pivotloc+1,high);             //对右子表递归排序
    }
}                                             //QSort

void QuickSort(SqList &L)
{
   //对顺序表 L 做快速排序
   QSort(L,1,L.length);
}
```

```
void Merge(RedType R[],RedType T[],int low,int mid,int high)
{
   //将有序表 R[low..mid]和 R[mid+1..high]归并为有序表 T[low..high]
    int i,j,k;
    i=low; j=mid+1;k=low;
     while(i<=mid&&j<=high)
    {
        //将 R 中记录由小到大地并入 T 中
        if(R[i].key<=R[j].key) T[k++]=R[i++];
          else T[k++]=R[j++];
    }
    while(i<=mid)                                 //将剩余的 R[low..mid]复制到 T 中
        T[k++]=R[i++];
    while(j<=high)                                /将剩余的 R[j.high]复制到 T 中
        T[k++]=R[j++];
}//Merge
```