数据结构作业

陈文宇

2023 年 6 月 11 日

目录

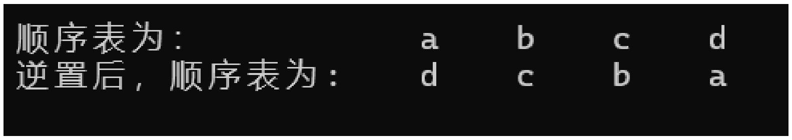
1 第一章作业

2 第二章作业

2.1 习题 2.6-线性表逆置

核心代码和运行结果如下, 完整代码详见附录。

```
//逆置
void invert(ElemType *R,int s,int t){
        //本算法将数组 R 中下标 自 t 到 s 的元素逆置
        int k;
        ElemType w;
        for(k= s; k<= (s+ t)/2.0; k++ ){
                w=R[k];
                R[k]= R[t+s-k];
                R[t+s-k]=w;
        }
}
```

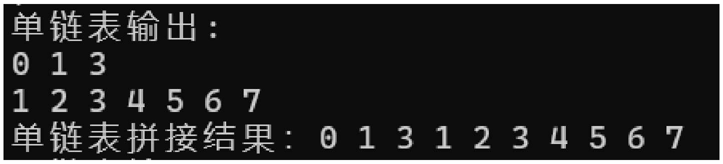


```
顺序表为:          a     b     c     d
逆置后，顺序表为:  d     c     b     a
```

2.2 习题 2.7-单链表的拼接

时间复杂度为 $O(min(m,n))$, 核心代码和运行结果如下，完整代码详见附录。

```
//单链表的拼接
void ListConcat(LinkList &ha,LinkList &hb,LinkList &hc){
    LNode *pa,*pb;
    int m,n;
    m=ListLength_L(ha);
    n=ListLength_L(hb);
    if(m<=n){
        pa=ha->next;
        while(pa->next) pa=pa->next;
        hc=ha;
        pa->next=hb;
    }
    else {
        pb=hb->next;
        while(pb->next) pb=pb->next;
        hc=hb;
        pb->next=ha;
    } //if
}//ListConcat
```

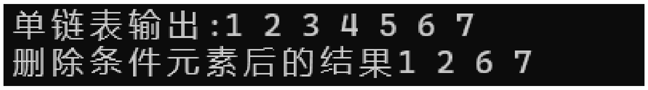


```
单链表输出:
0 1 3
1 2 3 4 5 6 7
单链表拼接结果: 0 1 3 1 2 3 4 5 6 7
```

2.3 习题 2.11-删除单增序列的某些元素

时间复杂度为 $O(n)$, 核心代码和运行结果如下，完整代码详见附录。

```
//递增序列  删除表中所有值大于mink且小于mark的元素
void DelBet(LinkList &L, int mink ,int maxk ){
        LNode  *p,*q;
        p=L->next; q=L;
    if( p->data>=maxk)
        cout<<"不存在大于"<<mink<<"并且小于"<<maxk<<"的元素"<<endl;
    else  {
                while(p && p->data<=mink){
                        q=p;
                        p=p->next;
                }
        while(p && p->data<maxk){
                        q->next=p->next;
            delete p;
            p=q->next;
        }
    }//else
}//Delete_Between
```



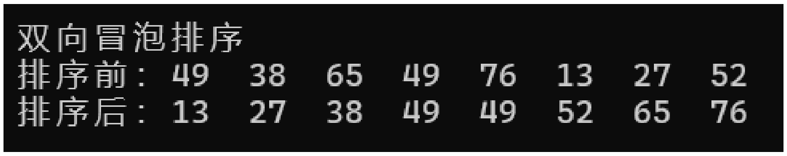
```
单链表输出:1 2 3 4 5 6 7
删除条件元素后的结果1 2 6 7
```

3 第三章作业

3.1 习题 3.8-双向起泡排序算法

核心代码和运行结果如下，完整代码详见附录。

```
//双向冒泡排序
void BubbleSort2(SqList &l){
    int change=1,low,high,i;
    low=1;
    high=l.length;
    while(low<high && change){
                change=0;
                for(i=low;i<high;i++)
             if(l.r[i].key>l.r[i+1].key){
                            l.r[0]=l.r[i];
                                l.r[i]=l.r[i+1];
                                l.r[i+1]=l.r[0];
                                change=1;
                 }
                high--;
                for(i=high;i>low;i--)
            if(l.r[i].key<l.r[i-1].key){
                                l.r[0]=l.r[i];
                            l.r[i]=l.r[i-1];
                            l.r[i-1]=l.r[0];
                            change=1;
                }
        low++;
     }
}
```



```
双向冒泡排序
排序前：49   38   65   49   76   13   27   52
排序后：13   27   38   49   49   52   65   76
```

4　第四章作业

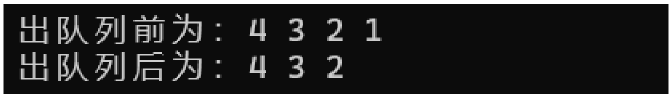
4.1　习题 4.7-表达式转换为后缀式图像

序号	当前字符																								运算符栈	后缀式
	(	a	+	b	)	*	(	c	/	(	d	-	e	)	+	f	)	+	a	*	b	*	c	#		
1	√																								#(	
2		√																							#(	a
3			√																						#(+	ab
4				√																					#(+	ab
5					√																				#	ab+
6					√																				#	ab+
7						√																			#*	ab+
8							√																		#*(	ab+
9								√																	#*(	ab+c
10									√																#*(/	ab+c
11										√															#*(/(	ab+c
12											√														#*(/(	ab+cd
13												√													#*(/(-	ab+cd
14													√												#*(/(-	ab+cde
15														√											#*(/	ab+cde-
16														√											#*(/	ab+cde-
17															√										#*(	ab+cde-/
18															√										#*(+	ab+cde-/
19																√									#*(+	ab+cde-/f
20																	√								#*	ab+cde-/f+
21																	√								#*	ab+cde-/f+
22																		√							#	ab+cde-/f+*
23																		√							#+	ab+cde-/f+*
24																			√						#+	ab+cde-/f+*a
25																				√					#+*	ab+cde-/f+*a
26																					√				#+*	ab+cde-/f+*ab
27																						√			#+	ab+cde-/f+*ab*
28																						√			#+*	ab+cde-/f+*ab*
29																							√		#+*	ab+cde-/f+*ab*c
30																								√	#+	ab+cde-/f+*ab*c*
31																								√	#	ab+cde-/f+*ab*c*+
32																								√		ab+cde-/f+*ab*c*+#

4.2　习题 4.9-队列的循环链表存储及基本操作

核心代码和运行结果如下，完整代码详见附录。

```
typedef int QElemType;
typedef struct QNode {
        QElemType data;
    struct QNode *next;
} LNode, *QueuePtr;    // 结点类型
typedef struct{
    QueuePtr  rear;    // 队尾指针
}CLinkQueue;

void InitCQueue(CLinkQueue &Q){
        //初始化循环链表表示的队列Q
        Q.rear = new LNode;
    Q.rear->next=Q.rear;
 } //InitCQueue
//入队列
void EnCQueue(CLinkQueue &Q, QElemType e){
        QueuePtr p;
        p=new LNode;
    p->data = e;
    p->next=Q.rear->next;
    Q.rear->next=p;
}//EnCQueue

//出队列
bool  DeCQueue(CLinkQueue &Q , QElemType &e){
    QueuePtr p,q;
    if (Q.rear->next == Q.rear)
        return false;
    while(p->next!=Q.rear){
        q=p;
        p=p->next;
        }
    e = p->data;
    q->next=Q.rear;
    delete p;

}//DeCQueue
```



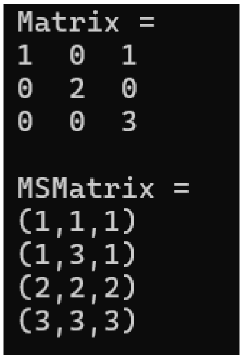
```
出队列前为: 4 3 2 1
出队列后为: 4 3 2
```

5 第五章作业

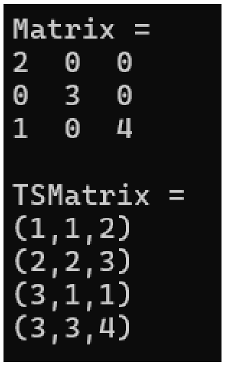
5.1 习题 5.11-三元组存储的稀疏矩阵求和算法

核心代码和运行结果如下，完整代码详见附录。

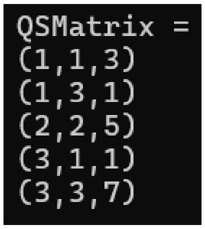
```
// 三元组存储的稀疏矩阵求和算法：C=A+B
bool Matrix_Addition(TSMatrix A, TSMatrix B, TSMatrix &C){
    int row_a, row_b,col_a, col_b, index_a, index_b, index_c;
    ElemType t;
    //行号，列号和各三元组的序号

    //同类型矩阵才能相加
    if(A.mu!=B.mu || A.nu!=B.nu) return FALSE;
    C.mu = A.mu;          C.nu = A.nu;

    //同时遍历两个三元组
    index_a=1; index_b=1; index_c=1;
    for( ; index_a<=A.tu&&index_b<=B.tu; ){
    //获取行列号
        row_a = A.data[index_a].i;        col_a = A.data[index_a].j;
        row_b = B.data[index_b].i;        col_b = B.data[index_b].j;

        //依行号访问稀疏矩阵
        if(row_a>row_b){
                //B的行号小 则复制B到C
                C.data[index_c].i = B.data[index_b].i;
                C.data[index_c].j = B.data[index_b].j;
                C.data[index_c].e = B.data[index_b].e;
                //向后步进
                index_b++;
                index_c++;
        }
        else if(row_a<row_b){
                //A的行号小 则复制A到C
                C.data[index_c].i = A.data[index_a].i;
                C.data[index_c].j = A.data[index_a].j;
                C.data[index_c].e = A.data[index_a].e;
                //向后步进
                index_a++;
                index_c++;
        }
        else{
                //若同行，则开始依列号访问稀疏矩阵
                if(col_a>col_b){
                        //B的列号小，复制B到C
                        C.data[index_c].i = B.data[index_b].i;
                        C.data[index_c].j = B.data[index_b].j;
                        C.data[index_c].e = B.data[index_b].e;
                        //向后步进
                        index_b++;
                        index_c++;
                }
                else  if(col_a<col_b){
                        //A的列号小，复制A到C
                        C.data[index_c].i = A.data[index_a].i;
                        C.data[index_c].j = A.data[index_a].j;
                        C.data[index_c].e = A.data[index_a].e;
                        //向后步进
                        index_a++;
                        index_c++;
                }
                else{
                        //行列号相同 ,需判断元素相加是否为零
                t=A.data[index_a].e+B.data[index_b].e;
                        if(t){
                                C.data[index_c].i = A.data[index_a].i;
                                C.data[index_c].j = A.data[index_a].j;
                                C.data[index_c].e = t;
                                index_c++;
                        }
                        //向后步进
                        index_a++;
                        index_b++;
            }
        }
    }
    //B取完A未取完
    while (index_a <= A.tu){
        C.data[index_c].i = A.data[index_a].i;
        C.data[index_c].j = A.data[index_a].j;
        C.data[index_c].e = A.data[index_a].e;
        index_a++;
        index_c++;
    }
    //A取完B未取完
    while (index_b <= B.tu){
        C.data[index_c].i = B.data[index_b].i;
        C.data[index_c].j = B.data[index_b].j;
        C.data[index_c].e = B.data[index_b].e;
        index_b++;
        index_c++;
    }
    C.tu = index_c - 1;
}
```



```
Matrix =
1  0  1
0  2  0
0  0  3

MSMatrix =
(1,1,1)
(1,3,1)
(2,2,2)
(3,3,3)
```



```
Matrix =
2  0  0
0  3  0
1  0  4

TSMatrix =
(1,1,2)
(2,2,3)
(3,1,1)
(3,3,4)
```



```
QSMatrix =
(1,1,3)
(1,3,1)
(2,2,5)
(3,1,1)
(3,3,7)
```

6 第六章作业

6.1 习题 6.7-寻找条件二叉树

1. 先序遍历和中序遍历时，得到的结点访问序列相同的二叉树

答：即所有结点的左子树为空的二叉树

2. 后序遍历和中序遍历时，得到的结点访问序列相同的二叉树

答：即所有结点的右子树为空的二叉树

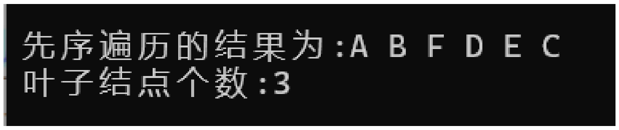
3. 先序遍历和后序遍历时，得到的结点访问序列相同的二叉树

答：即左右子树均为空的二叉树

6.2 习题 6.9-计算叶子结点的递归算法

核心代码和运行结果如下，完整代码详见附录。

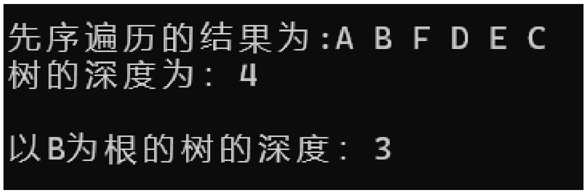
```
//求二叉树中叶子结点的数目
int LeafCount(BiTree T){
        if(!T)  return 0;
        else if(!T->lchild && !T->rchild)
        return 1;
        else return Leaf_Count(T->lchild)+Leaf_Count(T->rchild);
} //LeafCount
```



```
先序遍历的结果为:A B F D E C
叶子结点个数:3
```

6.3 习题 6.11-编写条件子树的深度

```
//求二叉树的深度
void BiTreeDepth(BiTree T,int h,int &depth){
        //h的初值为 1, depth的初值为 0
        //h 指向结点所在的层 ,depth是深度
        if(T){
                if(h>depth) depth=h;
                BiTreeDepth(T->lchild ,h+1,depth);
                BiTreeDepth(T->rchild ,h+1,depth);
        }
}//BiTreeDepth

//求子树深度的递归算法
int Get_Depth(BiTree T) {
    int m,n;
    if(!T)
        return 0;
    else {
        m=Get_Depth(T->lchild);
        n=Get_Depth(T->rchild);
        return (m>n?m:n)+1;
    }
} //Get_Depth

int Get_Sub_Depth(BiTree T,TElemType x, int &depth){
    if(T->data==x){
        depth=Get_Depth(T);
        return 0;
    }
    else{
        if(T->lchild)
            Get_Sub_Depth(T->lchild,x,depth);
        if(T->rchild)
            Get_Sub_Depth(T->rchild,x,depth);
  }
}
```



```
先序遍历的结果为:A B F D E C
树的深度为： 4

以B为根的树的深度： 3
```

A 附录

A.1 第二章习题的完整代码

```
//陈文宇
//10200115

//#include”stdafx.h”
#include<iostream>
using namespace std;


const int LISTINIT_SIZE=100;
const int LISTINCREMENT=10;
const bool TRUE=1;
const bool FALSE=0;
typedef int ElemType;

//单链表定义
typedef struct LNode{
        ElemType data;
        struct LNode *next;
}LNode,*LinkList;


//单链表基本操作

int ListLength_L(LinkList L);
LNode* LocateElem_L(LinkList L,ElemType e);
void LinkInsert_L(LinkList &L,LNode *p,LNode *s);
void ListDelete_L(LinkList L,LNode *p);
void CreateList_L(LinkList &L,ElemType *A,int n);
void ListConcat(LinkList &ha,LinkList &hb,LinkList &hc);
void ListTraverse_L(LinkList L);
void DelBet(LinkList &L, int mink ,int maxk );

int main(){
        ElemType A[3]={1,2,3},B[7]={1,2,3,4,5,6,7};
        LinkList L,p,s,q,V,W;
        L=NULL;
        V=NULL;
```

```
    W=new LNode;
    //printf("1");

    CreateList_L(L,A,3);
    CreateList_L(V,B,7);
    //printf("2");
    cout<<"单链表输出:";
    ListTraverse_L(L);

    cout<<"长度： "<<ListLength_L(L)<<endl;

    p=LocateElem_L(L,1);
    cout<<"获取数据为1的结点，它的数据为:"<<p->data<<endl;
    //printf("1");

    s=new LNode;
    (*s).data=0;//等价于s->data=0;
    LinkInsert_L(L,p,s);
    cout<<"插入结点后，单链表输出:";
    ListTraverse_L(L);
    cout<<"长度： "<<ListLength_L(L)<<endl;

    q=LocateElem_L(L,2);
    cout<<"获取数据为2的结点，它的数据为:"<<q->data<<endl;
    ListDelete_L(L,q);
    cout<<"删除结点后，单链表输出:";
    ListTraverse_L(L);
    cout<<"长度： "<<ListLength_L(L)<<endl;

    cout<<"单链表输出:"<<endl;
    ListTraverse_L(L);
    ListTraverse_L(V);
    ListConcat(L,V,W);
    cout<<"单链表拼接结果： ";
    ListTraverse_L(W);

    cout<<"单链表输出:";
    ListTraverse_L(V);
    DelBet(V,2,6);
    cout<<"删除条件元素后的结果";
    ListTraverse_L(V);
```

```
        delete s;
}
//线性表基本操作
//求线性表的长度
int ListLength_L(LinkList L){
        //L为链表的头指针，本函数返回L 所指链表的长度
        LinkList p;
        int i=0;
        p=L;
        while(p){
                i++;
                p=p->next;
        }
        return i;
}
//查找元素
LinkList LocateElem_L(LinkList L,ElemType e){
        LinkList p;
        p=L;
        while(p && p->data!=e) p=p->next;

        return p;
}
//插入结点操作
void LinkInsert_L(LinkList &L,LNode *p,LNode *s){
        //将 s 插入到 p前
        LNode *q;

        if(p==L){
                s->next=p;
                L=s;
        }
        else {
                q=L;
                while(q->next != p) q=q->next;
                s->next=p;
                q->next=s;
        }
}
//删除结点操作
void ListDelete_L(LinkList L,LNode *p){
        LNode *q;
        if(p==L){
                L=p->next;
        }
        else{
                q=L;
                while(q->next!=p) q=q->next;
                q->next=p->next;
        }
        delete p;
}
//创建单链表
void CreateList_L(LinkList &L,ElemType *A,int n){
        int i;
        LNode *s;
        L = NULL;
        for(i=n-1;i>=0;--i){
                s=new LNode;
                s->data=A[i];
                s->next=L;
                L=s;
        }
}
//单链表的拼接
void ListConcat(LinkList &ha,LinkList &hb,LinkList &hc){

    LNode *pa,*pb;
    int m,n;
    m=ListLength_L(ha);
        n=ListLength_L(hb);
    if(m<=n){
                pa=ha->next;
        while(pa->next) pa=pa->next;
        hc=ha;
                pa->next=hb;
    }
    else {
        pb=hb->next;
        while(pb->next) pb=pb->next;
        hc=hb;
                pb->next=ha;
    } //if

}//ListConcat
//递增序列 删除表中所有值大于mink且小于mark的元素
void DelBet(LinkList &L, int mink ,int maxk ){
        LNode  *p,*q;
        p=L->next; q=L;
    if( p->data>=maxk)
        cout<<”不存在大于”<<mink<<”并且小于”<<maxk<<”的元素”<<endl;
    else  {
                while(p && p->data<=mink){
                        q=p;
                        p=p->next;
                }
        while(p && p->data<maxk){
                        q->next=p->next;
            delete p;
            p=q->next;
        }
    }//else
}//Delete_Between
//遍历输出
void ListTraverse_L(LinkList L){
        LNode *p;
        p=L;
        while(p){
                cout<<p->data<<”␣”;
                p=p->next;
        }
        cout<<endl;
}
```

A.2 第三章习题的完整代码

```
//陈文宇
//10200115
#include<iostream>
using namespace std;

const int MAXSIZE=20;
const bool TRUE=1;
const bool FALSE=0;

//定义变量类型
typedef int KeyType;
typedef char InfoType;

typedef struct{
        KeyType key;
        InfoType val;
}RcdType;

typedef struct {
        RcdType r[MAXSIZE+1];
        int length;
}SqList;


//函数声明
void SelectPass(SqList &L,int i);
void SelectSort(SqList &L);

void InsertPass(SqList &L,int i);
void InsertSort(SqList &L);

void BubbleSort(SqList &L);
void BubbleSort2(SqList &l);

int Partition(RcdType R[], int low, int high);
void Qsort(RcdType R[], int s,int t);
void QuickSort(SqList &L);

void Merge(RcdType SR[], RcdType TR[], int i, int m, int n);//归并排序
void Msort(RcdType SR[], RcdType TR1[], int s,int t,int n);
void MergeSort(SqList &L);

int main(){
    SqList L,M,N,O,P,Q;

//L.r=new RcdType[MAXSIZE+1];
L.length=8; M.length=8; N.length=8; O.length=8; P.length=8; Q.length=8;

```

```
        L.r[1].key=49;
        L.r[2].key=38;
        L.r[3].key=65;
        L.r[4].key=49;
        L.r[5].key=76;
        L.r[6].key=13;
        L.r[7].key=27;
        L.r[8].key=52;

    for(int i=1; i<=8; i++){
        M.r[i].key=L.r[i].key;
        N.r[i].key=L.r[i].key;
        O.r[i].key=L.r[i].key;
        P.r[i].key=L.r[i].key;
        Q.r[i].key=L.r[i].key;
        }


        //------------------------------------------------------------
        printf("选择排序\n");
        cout<<"排序前："<<endl;
        for(int i=1;i<=L.length; i++){
                cout<<L.r[i].key<<"␣␣";
        }
        cout<<endl;
        SelectSort(L);
        cout<<"排序后：";
        for(int i=1;i<=L.length; i++){
                cout<<L.r[i].key<<"␣␣";
        }


        //-------------------------------------------------------------
        printf("\n\n\n插入排序\n");
        cout<<"排序前：";
        for(int i=1;i<=M.length; i++){
                cout<<M.r[i].key<<"␣␣";
        }
        cout<<endl;
        InsertSort(M);
        cout<<"排序后：";
        for(int i=1;i<=M.length; i++){
                cout<<M.r[i].key<<"␣␣";
        }


        //--------------------------------------------------------------
        printf("\n\n\n冒泡排序\n");
        cout<<"排序前：";
        for(int i=1;i<=N.length; i++){
                cout<<N.r[i].key<<"␣␣";
        }
        cout<<endl;
        BubbleSort(N);
        cout<<"排序后：";
        for(int i=1;i<=N.length; i++){
                cout<<N.r[i].key<<"␣␣";
        }

        //--------------------------------------------------------------
        printf("\n\n\n双向冒泡排序\n");
        cout<<"排序前：";
        for(int i=1;i<=Q.length; i++){
                cout<<Q.r[i].key<<"␣␣";
        }
        cout<<endl;
        BubbleSort2(Q);
        cout<<"排序后：";
        for(int i=1;i<=Q.length; i++){
                cout<<Q.r[i].key<<"␣␣";
        }


        //--------------------------------------------------------------
        printf("\n\n\n快速排序\n");
        cout<<"排序前：";
        for(int i=1;i<=O.length; i++){
                cout<<O.r[i].key<<"␣␣";
        }
        cout<<endl;
        QuickSort(O);
        cout<<"排序后：";
        for(int i=1;i<=O.length; i++){
                cout<<O.r[i].key<<"␣␣";
        }


        //------------------------------------------
        printf("\n\n\n归并排序\n");
        cout<<"排序前：";
        for(int i=1;i<=P.length; i++){
                cout<<P.r[i].key<<"␣␣";
        }
        cout<<endl;
        MergeSort(P);
        cout<<"排序后：";
        for(int i=1;i<=P.length; i++){
                cout<<P.r[i].key<<"␣␣";
        }


}


//选择排序
void SelectPass(SqList &L,int i){
    //已知L.r[1:1:i-1]中关键字非递减排序，本算法实现第i躺选择排序
    //即在L.r[i:1:n]的记录中选出关键字 最小的记录L.r[j]和r[i]进行交换
    int j=i;
    RcdType W;
    for(int k=i+1; k<=L.length; k++)
        if(L.r[k].key<L.r[j].key) j=k;
    if(i!=j){
        W=L.r[j];
        L.r[j]=L.r[i];
        L.r[i]=W;
    }
}// SelectPass

//顺序表的选择排序
void SelectSort(SqList &L){
        RcdType W;
        int j;
        int k;
        for(int i=1; i<L.length; i++){
                j=i;
                for(k=i+1; k<=L.length; k++){
                        if(L.r[k].key<L.r[j].key) j=k;
                }
                if(i!=j){
                        W=L.r[j]; L.r[j]=L.r[i]; L.r[i]=W;
                }
        }
}//SelectSort


//插入排序
void InsertPass(SqList &L,int i){
        int j=i-1;
        L.r[0]=L.r[i];

        for(; L.r[0].key < L.r[j].key; j--)
                L.r[j+1]=L.r[j];

        L.r[j+1]=L.r[j];
}//InsertPass


//顺序表的插入排序
void InsertSort(SqList &L){
        int j;
        for(int i=2; i<=L.length; i++){
                if(L.r[i].key < L.r[i-1].key){
                        L.r[0]=L.r[i];
                        for(j=i-1; L.r[0].key<L.r[j].key; j--)
                                L.r[j+1]=L.r[j];
                        L.r[j+1]=L.r[0];
                }//if
        }//for
}//InsertSort


// 顺序表的起泡排序
void BubbleSort(SqList &L){
    int i=L.length,LastExchangeIndex;
    RcdType W;
    int j;
    while(i>1){
        LastExchangeIndex=1;
        for(j=1; j<i; j++){
            if(L.r[j+1].key<L.r[j].key){
                       W=L.r[j]; L.r[j]=L.r[j+1]; L.r[j+1]=W;
                        LastExchangeIndex=j;
                }//if
        }//for
        i=LastExchangeIndex;
    }//while
} //BubbleSort

//双向冒泡排序
void BubbleSort2(SqList &l){
    int change=1,low,high,i;
    low=1;
    high=l.length;
    while(low<high && change){
                change=0;
                for(i=low;i<high;i++)
             if(l.r[i].key>l.r[i+1].key){
                            l.r[0]=l.r[i];
                                l.r[i]=l.r[i+1];
                                l.r[i+1]=l.r[0];
                                change=1;
                 }
                high--;
                for(i=high;i>low;i--)
            if(l.r[i].key<l.r[i-1].key){
                                l.r[0]=l.r[i];
                            l.r[i]=l.r[i-1];
                            l.r[i-1]=l.r[0];
                            change=1;
                }
        low++;
     }
}

```

```
//快速排序算法
int Partition(RcdType R[], int low, int high){
        R[0]=R[low];
        KeyType pivotkey=R[low].key;
        while(low<high){
                while(low<high && R[high].key>=pivotkey) --high;
                if(low<high) R[low++]=R[high];
                while(low<high && R[low].key<=pivotkey) ++low;
                if(low<high) R[high--]=R[low];
        }//while
        R[low]=R[0];
        //printf("陈文宇");
        return low;

} //Partition

void Qsort(RcdType R[], int s,int t){
        int pivotloc;
        if(s<t){
                pivotloc=Partition(R,s,t);
                Qsort(R,s,pivotloc -1);
                Qsort(R,pivotloc+1,t);
        }//if

}//Qsort

void QuickSort(SqList &L){
        Qsort(L.r,1,L.length);
}//QuickSort


//归并排序
void Merge(RcdType SR[], RcdType TR[], int i, int m, int n){
        int j=m+1;
        int k;
        for(k=i; i<=m && j<=n; k++){
                if(SR[i].key<=SR[j].key) TR[k]=SR[i++];
                else TR[k]=SR[j++];
        }//for
        while(i<=m) TR[k++]=SR[i++];
        while(j<=n) TR[k++]=SR[j++];

} //Merge

void Msort(RcdType SR[], RcdType TR1[], int s,int t,int n){
        RcdType TR2[n];
        int m;
        if(s==t) TR1[s]=SR[s];
        else{
                m=(s+t)/2;
                Msort(SR,TR2,s,m,n);
                Msort(SR,TR2,m+1,t,n);
                Merge(TR2,TR1,s,m,t);
        }//else

}//Msort

void MergeSort(SqList &L){
        Msort(L.r, L.r, 1, L.length,L.length+1);
}//MergeSort
```

A.3 第四章习题的完整代码

```
//陈文宇
//10200115
#include<iostream>
using namespace std;

typedef int QElemType;
typedef struct QNode {
        QElemType data;
    struct QNode *next;
} LNode, *QueuePtr;    // 结点类型
typedef struct{
    QueuePtr  rear;    // 队尾指针
}CLinkQueue;


void InitCQueue(CLinkQueue &Q);
void EnCQueue(CLinkQueue &Q, QElemType  e);
bool DeCQueue(CLinkQueue &Q , QElemType &e);
void ListTraverse_L(CLinkQueue L);
```

```

int main(){
            CLinkQueue Q;
            QElemType e;

            InitCQueue(Q);

            EnCQueue(Q,1);
            EnCQueue(Q,2);
            EnCQueue(Q,3);
            EnCQueue(Q,4);
            cout<<”出队列前为：”;
            ListTraverse_L(Q);
            DeCQueue(Q,e);
            cout<<”出队列后为：”;
            ListTraverse_L(Q);
}


void InitCQueue(CLinkQueue &Q){
            //初始化循环链表表示的队列Q
            Q.rear = new LNode;
      Q.rear->next=Q.rear;
 } //InitCQueue

//入队列
void EnCQueue(CLinkQueue &Q, QElemType e){
            QueuePtr p;
            p=new LNode;
      p->data = e;
      p->next=Q.rear->next;
      Q.rear->next=p;
}//EnCQueue

//出队列
bool  DeCQueue(CLinkQueue &Q , QElemType &e){
      QueuePtr p,q;
      if (Q.rear->next == Q.rear)
            return false;
      while(p->next!=Q.rear){
            q=p;
```

```
                p=p->next;
        }
    e = p->data;
    q->next=Q.rear;
    delete p;

}//DeCQueue

//遍历输出
void ListTraverse_L(CLinkQueue L){
        QueuePtr p;
        p=L.rear->next;
        while(p != L.rear){
                cout<<p->data<<”␣”;
                p=p->next;
        }
        cout<<endl;
}//ListTraverse_L
```

A.4 第五章习题的完整代码

```
//陈文宇
//10200115
#include<iostream>
#include<stdlib.h>
#include<malloc.h>

using namespace std;
const int MAXSIZE=100;
const bool TRUE=1;
const bool FALSE=0;

typedef int ElemType;

typedef struct{
    int i,j;                //非零元的行下标和列下标
    ElemType e;             //该非零元的元素值
}Triple;
typedef struct{
    Triple data[MAXSIZE];//非零元三元组表，data[0]未用
    int mu,nu,tu;                   //稀疏矩阵的行数，列数和非零元个数
```

```
}TSMatrix;

void TSMattrans(int** M,TSMatrix &MS,int m,int n);
void coutMat(int** M,int m, int n);
void coutTSMat(TSMatrix MS);
bool Matrix_Addition(TSMatrix A, TSMatrix B, TSMatrix &C);


int main(){
        int m=3,n=3;
        int** M=new int*[m];
        int** T=new int*[m];
        TSMatrix MS,TS,QS;

        //初始化
        for(int i=0;i<m;i++){
                M[i]=new int[n];
                T[i]=new int[n];
        }
        MS.mu=m;        MS.nu=n;
        TS.mu=m;        TS.nu=n;
        for(int p=0;p<m;p++){
                for(int q=0;q<n;q++){
                        M[p][q]=0;
                        T[p][q]=0;
                }
                M[p][p]=p+1;
                T[p][p]=p+2;
        }
        M[0][n-1]=1;
        T[m-1][0]=1;

        //以三元组 形式存储
        TSMattrans(M,MS,m,n);
        coutMat(M,m,n);
        cout<<"MSMatrix␣=␣"<<endl;
        coutTSMat(MS);

        TSMattrans(T,TS,m,n);
        coutMat(T,m,n);
        cout<<"TSMatrix␣=␣"<<endl;
```

```
                coutTSMat(TS);

                //稀疏矩阵加法
                Matrix_Addition(MS,TS,QS);
                cout<<”QSMatrix␣=␣”<<endl;
                coutTSMat(QS);

                //销毁矩阵
                for(int p=0;p<m;p++){
                                delete[] M[p];
                                delete[] T[p];
                }
                delete[] M;
                delete[] T;
}
//用于录入稀疏矩阵 并以三元组 形式存储
void TSMattrans(int** M,TSMatrix &MS,int m,int n){
                int k=1;
                for(int p=0; p<m; p++){
                                for(int q=0; q<n; q++){
                                                if(M[p][q] != 0 ){
                                                                MS.data[k].i=p+1;
                                                                MS.data[k].j=q+1;
                                                                MS.data[k].e=M[p][q];
                                                                k++;
                                                }
                                }
                }
                MS.tu=k-1;
}

// 三元组存储的稀疏矩阵求和算法: C=A+B
bool Matrix_Addition(TSMatrix A, TSMatrix B, TSMatrix &C){
        int row_a, row_b,col_a, col_b, index_a, index_b, index_c;
        ElemType t;
        //行号，列号和各三元组的序号

        //同类型矩阵才能相加
        if(A.mu!=B.mu || A.nu!=B.nu) return FALSE;
        C.mu = A.mu;            C.nu = A.nu;

```

```
    //同时遍历两个三元组
    index_a=1; index_b=1; index_c=1;
    for( ; index_a<=A.tu&&index_b<=B.tu; ){
    //获取行列号
        row_a = A.data[index_a].i;          col_a = A.data[index_a].j;
        row_b = B.data[index_b].i;          col_b = B.data[index_b].j;

        //依行号访问稀疏矩阵
        if(row_a>row_b){
                //B的行号小 则复制B到C
                C.data[index_c].i = B.data[index_b].i;
                C.data[index_c].j = B.data[index_b].j;
                C.data[index_c].e = B.data[index_b].e;
                //向后步进
                index_b++;
                index_c++;
        }
        else if(row_a<row_b){
                //A的行号小 则复制A到C
                C.data[index_c].i = A.data[index_a].i;
                C.data[index_c].j = A.data[index_a].j;
                C.data[index_c].e = A.data[index_a].e;
                //向后步进
                index_a++;
                index_c++;
        }
        else{
                //若同行，则开始依列号访问稀疏矩阵
                if(col_a>col_b){
                        //B的列号小，复制B到C
                        C.data[index_c].i = B.data[index_b].i;
                        C.data[index_c].j = B.data[index_b].j;
                        C.data[index_c].e = B.data[index_b].e;
                        //向后步进
                        index_b++;
                        index_c++;
                }
                else if(col_a<col_b){
                        //A的列号小，复制A到C
                        C.data[index_c].i = A.data[index_a].i;
                        C.data[index_c].j = A.data[index_a].j;
                        C.data[index_c].e = A.data[index_a].e;
                        //向后步进
                        index_a++;
                        index_c++;
                }
                else{
                        //行列号相同 ,需判断元素相加是否为零
                t=A.data[index_a].e+B.data[index_b].e;
                        if(t){
                                C.data[index_c].i = A.data[index_a].i;
                                C.data[index_c].j = A.data[index_a].j;
                                C.data[index_c].e = t;
                                index_c++;
                        }
                        //向后步进
                        index_a++;
                        index_b++;
            }
        }
    }
    //B取完A未取完
    while (index_a <= A.tu){
        C.data[index_c].i = A.data[index_a].i;
        C.data[index_c].j = A.data[index_a].j;
        C.data[index_c].e = A.data[index_a].e;
        index_a++;
        index_c++;
    }
 //A取完B未取完
    while (index_b <= B.tu){
        C.data[index_c].i = B.data[index_b].i;
        C.data[index_c].j = B.data[index_b].j;
        C.data[index_c].e = B.data[index_b].e;
        index_b++;
        index_c++;
    }
    C.tu = index_c - 1;
}
//矩阵输出
void coutMat(int** M,int m, int n){
        cout<<"Matrix␣=␣"<<endl;
        for(int i=0;i<m;i++){
                for(int j=0;j<n;j++){
                        cout<<M[i][j]<<"␣␣";
                }
                cout<<endl;
        }
        cout<<endl;
}
//三元组输出
void coutTSMat(TSMatrix MS){

    for(int p=1;p<=MS.tu;p++){
        cout<<"("<<MS.data[p].i<<",";
        cout<<MS.data[p].j<<",";
        cout<<MS.data[p].e<<")"<<endl;
    }
    cout<<endl;
}
```

A.5 第六章习题的完整代码

```
//陈文宇
//10200115
#include<iostream>
using namespace std;

//二叉树定义
typedef  char TElemType;
typedef struct BiTNode{
        TElemType data;
        struct BiTNode *lchild,*rchild;
}BiTNode,*BiTree;

void CreatebiTree(BiTree &T);
int Get_Depth(BiTree T);
int Get_Sub_Depth(BiTree T,TElemType x, int &depth);
int LeafCount(BiTree T);
void Preorder(BiTree T, void (*visit)(BiTree T));

int main(){

        BiTree T;

        //--------------------------------------------------------------------
        //树的深度
        cout<<"先序遍历的结果为:";
        Preorder(T, visit);
        cout<<endl;
        int h=1,depth=0;
        BiTreeDepth(T,h,depth);
        cout<<"树的深度为： "<<depth<<endl;
        cout<<endl;

        //条件子树的深度
        cout<<"以B为根的树的深度： ";
        Get_Sub_Depth(T,'B',depth);
        cout<<depth<<endl;
        cout<<endl;

        cout<<"先序遍历的结果为:";
        Preorder(T, visit);
        cout<<endl;
        //二叉树的叶子结点个数
        cout<<"叶子结点个数:";
        cout<<LeafCount(T)<<endl;
        cout<<endl;
}
//二叉链表创建二叉树
void CreatebiTree(BiTree &T){
        TElemType ch;
        cin>>ch;
        if(ch=='#')T=NULL;//使用整型时请用 0 代替
        else{
                T=new BiTNode;
                T->data=ch;
                CreatebiTree(T->lchild);
                CreatebiTree(T->rchild);
        }
}//CreatebiTree

//求子树深度的递归算法
int Get_Depth(BiTree T) {
        int m,n;
        if(!T)
                return 0;
        else {
                m=Get_Depth(T->lchild);
                n=Get_Depth(T->rchild);
                return (m>n?m:n)+1;
        }
} //Get_Depth

int Get_Sub_Depth(BiTree T,TElemType x, int &depth){
        if(T->data==x){
                depth=Get_Depth(T);
                return 0;
        }
        else{
                if(T->lchild)
                        Get_Sub_Depth(T->lchild,x,depth);
                if(T->rchild)
                        Get_Sub_Depth(T->rchild,x,depth);
   }
}
//求二叉树中叶子结点的数目
int LeafCount(BiTree T){
        if(!T) return 0;
        else if(!T->lchild && !T->rchild)
        return 1;
        else return LeafCount(T->lchild)+LeafCount(T->rchild);

} //LeafCount

//先序遍历 (递归)
void Preorder(BiTree T, void (*visit)(BiTree)){
        if(T){
                visit(T);
                Preorder(T->lchild,visit);
                Preorder(T->rchild,visit);
        }
}
```