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;
```

```
15          while(pb->next) pb=pb->next;
16          hc=hb;
17          pb->next=ha;
18      } //if
19  }//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。

```
1  //递增序列 删除表中所有值大于mink且小于mark的元素
2  void DelBet(LinkList &L, int mink ,int maxk ){
3          LNode *p,*q;
4          p=L->next; q=L;
5      if( p->data>=maxk)
6          cout<<"不存在大于"<<mink<<"并且小于"<<maxk<<"的元素"<<endl;
7      else {
8                  while(p && p->data<=mink){
9                          q=p;
10                         p=p->next;
11                 }
12         while(p && p->data<maxk){
13                         q->next=p->next;
14             delete p;
15             p=q->next;
16         }
17     }//else
18 }//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。

```
1 typedef int QElemType;
2 typedef struct QNode {
```

```
3           QElemType data;
4       struct QNode *next;
5   } LNode, *QueuePtr;    // 结点类型
6   typedef struct{
7       QueuePtr  rear;    // 队尾指针
8   }CLinkQueue;
9
10  void InitCQueue(CLinkQueue &Q){
11          //初始化循环链表表示的队列Q
12          Q.rear = new LNode;
13      Q.rear->next=Q.rear;
14   } //InitCQueue
15  //入队列
16  void EnCQueue(CLinkQueue &Q, QElemType e){
17          QueuePtr p;
18          p=new LNode;
19      p->data = e;
20      p->next=Q.rear->next;
21      Q.rear->next=p;
22  }//EnCQueue
23
24  //出队列
25  bool  DeCQueue(CLinkQueue &Q , QElemType &e){
26      QueuePtr p,q;
27      if (Q.rear->next == Q.rear)
28          return false;
29      while(p->next!=Q.rear){
30          q=p;
31          p=p->next;
32          }
33      e = p->data;
34      q->next=Q.rear;
35      delete p;
36
37  }//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++;
        }
```

```
37          else{
38                  //若同行，则开始依列号访问稀疏矩阵
39                  if(col_a>col_b){
40                          //B的列号小，复制B到C
41                          C.data[index_c].i = B.data[index_b].i;
42                          C.data[index_c].j = B.data[index_b].j;
43                          C.data[index_c].e = B.data[index_b].e;
44                          //向后步进
45                          index_b++;
46                          index_c++;
47                  }
48                  else if(col_a<col_b){
49                          //A的列号小，复制A到C
50                          C.data[index_c].i = A.data[index_a].i;
51                          C.data[index_c].j = A.data[index_a].j;
52                          C.data[index_c].e = A.data[index_a].e;
53                          //向后步进
54                          index_a++;
55                          index_c++;
56                  }
57                  else{
58                          //行列号相同 ,需判断元素相加是否为零
59                  t=A.data[index_a].e+B.data[index_b].e;
60                          if(t){
61                                  C.data[index_c].i = A.data[index_a].i;
62                                  C.data[index_c].j = A.data[index_a].j;
63                                  C.data[index_c].e = t;
64                                  index_c++;
65                          }
66                          //向后步进
67                          index_a++;
68                          index_b++;
69              }
70          }
71      }
72      //B取完A未取完
73      while (index_a <= A.tu){
74          C.data[index_c].i = A.data[index_a].i;
75          C.data[index_c].j = A.data[index_a].j;
76          C.data[index_c].e = A.data[index_a].e;
77          index_a++;
```

```
78              index_c++;
79          }
80          //A取完B未取完
81          while (index_b <= B.tu){
82              C.data[index_c].i = B.data[index_b].i;
83              C.data[index_c].j = B.data[index_b].j;
84              C.data[index_c].e = B.data[index_b].e;
85              index_b++;
86              index_c++;
87          }
88          C.tu = index_c - 1;
89      }
```

```
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。

```
1  //求二叉树中叶子结点的数目
2  int LeafCount(BiTree T){
3          if(!T)  return 0;
4          else if(!T->lchild && !T->rchild)
5          return 1;
6          else return Leaf_Count(T->lchild)+Leaf_Count(T->rchild);
7  } //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

```
1  //求二叉树的深度
2  void BiTreeDepth(BiTree T,int h,int &depth){
3          //h的初值为 1, depth的初值为 0
4          //h 指向结点所在的层 ,depth是深度
5          if(T){
6                  if(h>depth) depth=h;
7                  BiTreeDepth(T->lchild ,h+1,depth);
8                  BiTreeDepth(T->rchild ,h+1,depth);
9          }
10 }//BiTreeDepth
11
12 //求子树深度的递归算法
13 int Get_Depth(BiTree T) {
```

```
14          int m,n;
15          if(!T)
16                  return 0;
17          else {
18                  m=Get_Depth(T->lchild);
19                  n=Get_Depth(T->rchild);
20                  return (m>n?m:n)+1;
21          }
22  } //Get_Depth
23
24  int Get_Sub_Depth(BiTree T,TElemType x, int &depth){
25          if(T->data==x){
26                  depth=Get_Depth(T);
27                  return 0;
28          }
29          else{
30                  if(T->lchild)
31                          Get_Sub_Depth(T->lchild,x,depth);
32                  if(T->rchild)
33                          Get_Sub_Depth(T->rchild,x,depth);
34      }
35  }
```

```
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

```
1  //陈文宇
2  //10200115
3
4  //#include"stdafx .h"
5  #include<iostream>
6  using namespace std;
7
8
9  const int LISTINIT_SIZE=100;
10 const int LISTINCREMENT=10;
11 const bool TRUE=1;
12 const bool FALSE=0;
13 typedef int ElemType;
14
15 //单链表定义
16 typedef struct LNode{
17         ElemType data;
18         struct LNode *next;
19 }LNode,*LinkList;
20
21
22 //单链表基本操作
23
24 int ListLength_L(LinkList L);
25 LNode* LocateElem_L(LinkList L,ElemType e);
26 void LinkInsert_L(LinkList &L,LNode *p,LNode *s);
27 void ListDelete_L(LinkList L,LNode *p);
28 void CreateList_L(LinkList &L,ElemType *A,int n);
29 void ListConcat(LinkList &ha,LinkList &hb,LinkList &hc);
30 void ListTraverse_L(LinkList L);
31 void DelBet(LinkList &L, int mink ,int maxk );
32
33 int main(){
34         ElemType A[3]={1,2,3},B[7]={1,2,3,4,5,6,7};
35         LinkList L,p,s,q,V,W;
36         L=NULL;
37         V=NULL;
```

```
38          W=new LNode;
39          //printf("1");
40
41          CreateList_L(L,A,3);
42          CreateList_L(V,B,7);
43          //printf("2");
44          cout<<"单链表输出:";
45          ListTraverse_L(L);
46
47          cout<<"长度: "<<ListLength_L(L)<<endl;
48
49          p=LocateElem_L(L,1);
50          cout<<"获取数据为1的结点, 它的数据为:"<<p->data<<endl;
51          //printf("1");
52
53          s=new LNode;
54          (*s).data=0;//等价于s->data=0;
55          LinkInsert_L(L,p,s);
56          cout<<"插入结点后, 单链表输出:";
57          ListTraverse_L(L);
58          cout<<"长度: "<<ListLength_L(L)<<endl;
59
60          q=LocateElem_L(L,2);
61          cout<<"获取数据为2的结点, 它的数据为:"<<q->data<<endl;
62          ListDelete_L(L,q);
63          cout<<"删除结点后, 单链表输出:";
64          ListTraverse_L(L);
65          cout<<"长度: "<<ListLength_L(L)<<endl;
66
67          cout<<"单链表输出:"<<endl;
68          ListTraverse_L(L);
69          ListTraverse_L(V);
70          ListConcat(L,V,W);
71          cout<<"单链表拼接结果: ";
72          ListTraverse_L(W);
73
74          cout<<"单链表输出:";
75          ListTraverse_L(V);
76          DelBet(V,2,6);
77          cout<<"删除条件元素后的结果";
78          ListTraverse_L(V);
```

```
79          delete s;
80  }
81  //线性表基本操作
82  //求线性表的长度
83  int ListLength_L(LinkList L){
84          //L为链表的头指针，本函数返回L 所指链表的长度
85          LinkList p;
86          int i=0;
87          p=L;
88          while(p){
89                  i++;
90                  p=p->next;
91          }
92          return i;
93  }
94  //查找元素
95  LinkList LocateElem_L(LinkList L,ElemType e){
96          LinkList p;
97          p=L;
98          while(p && p->data!=e) p=p->next;
99
100         return p;
101 }
102 //插入结点操作
103 void LinkInsert_L(LinkList &L,LNode *p,LNode *s){
104         //将 s 插入到 p前
105         LNode *q;
106
107         if(p==L){
108                 s->next=p;
109                 L=s;
110         }
111         else {
112                 q=L;
113                 while(q->next != p) q=q->next;
114                 s->next=p;
115                 q->next=s;
116         }
117 }
118 //删除结点操作
119 void ListDelete_L(LinkList L,LNode *p){
```

```
120         LNode *q;
121         if(p==L){
122                 L=p->next;
123         }
124         else{
125                 q=L;
126                 while(q->next!=p) q=q->next;
127                 q->next=p->next;
128         }
129         delete p;
130 }
131 //创建单链表
132 void CreateList_L(LinkList &L,ElemType *A,int n){
133         int i;
134         LNode *s;
135         L = NULL;
136         for(i=n-1;i>=0;--i){
137                 s=new LNode;
138                 s->data=A[i];
139                 s->next=L;
140                 L=s;
141         }
142 }
143 //单链表的拼接
144 void ListConcat(LinkList &ha,LinkList &hb,LinkList &hc){
145
146     LNode *pa,*pb;
147     int m,n;
148     m=ListLength_L(ha);
149         n=ListLength_L(hb);
150     if(m<=n){
151                 pa=ha->next;
152         while(pa->next) pa=pa->next;
153         hc=ha;
154                 pa->next=hb;
155     }
156     else {
157         pb=hb->next;
158         while(pb->next) pb=pb->next;
159         hc=hb;
160                 pb->next=ha;
```

```
161     } //if
162
163 }//ListConcat
164 //递增序列 删除表中所有值大于mink且小于mark的元素
165 void DelBet(LinkList &L, int mink ,int maxk ){
166         LNode  *p,*q;
167         p=L->next; q=L;
168     if( p->data>=maxk)
169         cout<<"不存在大于"<<mink<<"并且小于"<<maxk<<"的元素"<<endl;
170     else  {
171                 while(p && p->data<=mink){
172                         q=p;
173                         p=p->next;
174                 }
175         while(p && p->data<maxk){
176                         q->next=p->next;
177             delete p;
178             p=q->next;
179         }
180     }//else
181 }//Delete_Between
182 //遍历输出
183 void ListTraverse_L(LinkList L){
184         LNode *p;
185         p=L;
186         while(p){
187                 cout<<p->data<<"␣";
188                 p=p->next;
189         }
190         cout<<endl;
191 }
```

A.2 第三章习题的完整代码

```
1 //陈文宇
2 //10200115
3 #include<iostream>
4 using namespace std;
5
6 const int MAXSIZE=20;
7 const bool TRUE=1;
```

```
8  const bool FALSE=0;
9
10 //定义变量类型
11 typedef int KeyType;
12 typedef char InfoType;
13
14 typedef struct{
15         KeyType key;
16         InfoType val;
17 }RcdType;
18
19 typedef struct {
20         RcdType r[MAXSIZE+1];
21         int length;
22 }SqList;
23
24
25 //函数声明
26 void SelectPass(SqList &L,int i);
27 void SelectSort(SqList &L);
28
29 void InsertPass(SqList &L,int i);
30 void InsertSort(SqList &L);
31
32 void BubbleSort(SqList &L);
33 void BubbleSort2(SqList &l);
34
35 int Partition(RcdType R[], int low, int high);
36 void Qsort(RcdType R[], int s,int t);
37 void QuickSort(SqList &L);
38
39 void Merge(RcdType SR[], RcdType TR[], int i, int m, int n);//归并排序
40 void Msort(RcdType SR[], RcdType TR1[], int s,int t,int n);
41 void MergeSort(SqList &L);
42
43 int main(){
44     SqList L,M,N,O,P,Q;
45
46 //L.r=new RcdType[MAXSIZE+1];
47 L.length=8; M.length=8; N.length=8; O.length=8; P.length=8; Q.length=8;
48
```

```
49          L.r[1].key=49;
50          L.r[2].key=38;
51          L.r[3].key=65;
52          L.r[4].key=49;
53          L.r[5].key=76;
54          L.r[6].key=13;
55          L.r[7].key=27;
56          L.r[8].key=52;
57
58      for(int i=1; i<=8; i++){
59          M.r[i].key=L.r[i].key;
60          N.r[i].key=L.r[i].key;
61          O.r[i].key=L.r[i].key;
62          P.r[i].key=L.r[i].key;
63          Q.r[i].key=L.r[i].key;
64          }
65
66
67          //-------------------------------------------------------------
68          printf("选择排序\n");
69          cout<<"排序前： "<<endl;
70          for(int i=1;i<=L.length; i++){
71                  cout<<L.r[i].key<<"  ";
72          }
73          cout<<endl;
74          SelectSort(L);
75          cout<<"排序后： ";
76          for(int i=1;i<=L.length; i++){
77                  cout<<L.r[i].key<<"  ";
78          }
79
80
81          //-------------------------------------------------------------
82          printf("\n\n\n插入排序\n");
83          cout<<"排序前： ";
84          for(int i=1;i<=M.length; i++){
85                  cout<<M.r[i].key<<"  ";
86          }
87          cout<<endl;
88          InsertSort(M);
89          cout<<"排序后： ";
```

```
90          for(int i=1;i<=M.length; i++){
91                  cout<<M.r[i].key<<"  ";
92          }
93
94
95          //-------------------------------------------------------------
96          printf("\n\n\n冒泡排序\n");
97          cout<<"排序前： ";
98          for(int i=1;i<=N.length; i++){
99                  cout<<N.r[i].key<<"  ";
100         }
101         cout<<endl;
102         BubbleSort(N);
103         cout<<"排序后： ";
104         for(int i=1;i<=N.length; i++){
105                 cout<<N.r[i].key<<"  ";
106         }
107
108         //-------------------------------------------------------------
109         printf("\n\n\n双向冒泡排序\n");
110         cout<<"排序前： ";
111         for(int i=1;i<=Q.length; i++){
112                 cout<<Q.r[i].key<<"  ";
113         }
114         cout<<endl;
115         BubbleSort2(Q);
116         cout<<"排序后： ";
117         for(int i=1;i<=Q.length; i++){
118                 cout<<Q.r[i].key<<"  ";
119         }
120
121
122         //-------------------------------------------------------------
123         printf("\n\n\n快速排序\n");
124         cout<<"排序前： ";
125         for(int i=1;i<=O.length; i++){
126                 cout<<O.r[i].key<<"  ";
127         }
128         cout<<endl;
129         QuickSort(O);
130         cout<<"排序后： ";
```

```
131         for(int i=1;i<=O.length; i++){
132                 cout<<O.r[i].key<<"  ";
133         }
134
135
136         //-----------------------------------------------
137         printf("\n\n\n归并排序\n");
138         cout<<"排序前： ";
139         for(int i=1;i<=P.length; i++){
140                 cout<<P.r[i].key<<"  ";
141         }
142         cout<<endl;
143         MergeSort(P);
144         cout<<"排序后： ";
145         for(int i=1;i<=P.length; i++){
146                 cout<<P.r[i].key<<"  ";
147         }
148
149
150 }
151
152
153 //选择排序
154 void SelectPass(SqList &L,int i){
155     //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
156     //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
157     int j=i;
158     RcdType W;
159     for(int k=i+1; k<=L.length; k++)
160         if(L.r[k].key<L.r[j].key) j=k;
161     if(i!=j){
162         W=L.r[j];
163         L.r[j]=L.r[i];
164         L.r[i]=W;
165     }
166 }// SelectPass
167
168 //顺序表的选择排序
169 void SelectSort(SqList &L){
170         RcdType W;
171         int j;
```

```
172         int k;
173         for(int i=1; i<L.length; i++){
174                 j=i;
175                 for(k=i+1; k<=L.length; k++){
176                         if(L.r[k].key<L.r[j].key) j=k;
177                 }
178                 if(i!=j){
179                         W=L.r[j]; L.r[j]=L.r[i]; L.r[i]=W;
180                 }
181         }
182 }//SelectSort
183
184
185 //插入排序
186 void InsertPass(SqList &L,int i){
187         int j=i-1;
188         L.r[0]=L.r[i];
189
190         for(; L.r[0].key < L.r[j].key; j--)
191                 L.r[j+1]=L.r[j];
192
193         L.r[j+1]=L.r[j];
194 }//InsertPass
195
196
197 //顺序表的插入排序
198 void InsertSort(SqList &L){
199         int j;
200         for(int i=2; i<=L.length; i++){
201                 if(L.r[i].key < L.r[i-1].key){
202                         L.r[0]=L.r[i];
203                         for(j=i-1; L.r[0].key<L.r[j].key; j--)
204                                 L.r[j+1]=L.r[j];
205                         L.r[j+1]=L.r[0];
206                 }//if
207         }//for
208 }//InsertSort
209
210
211 // 顺序表的起泡排序
212 void BubbleSort(SqList &L){
```

```
213     int i=L.length,LastExchangeIndex;
214     RcdType W;
215     int j;
216     while(i>1){
217         LastExchangeIndex=1;
218         for(j=1; j<i; j++){
219             if(L.r[j+1].key<L.r[j].key){
220                     W=L.r[j]; L.r[j]=L.r[j+1]; L.r[j+1]=W;
221                     LastExchangeIndex=j;
222                 }//if
223         }//for
224         i=LastExchangeIndex;
225     }//while
226 } //BubbleSort
227
228 //双向冒泡排序
229 void BubbleSort2(SqList &l){
230     int change=1,low,high,i;
231     low=1;
232     high=l.length;
233     while(low<high && change){
234                 change=0;
235                 for(i=low;i<high;i++)
236             if(l.r[i].key>l.r[i+1].key){
237                         l.r[0]=l.r[i];
238                             l.r[i]=l.r[i+1];
239                             l.r[i+1]=l.r[0];
240                             change=1;
241                  }
242                 high--;
243                 for(i=high;i>low;i--)
244             if(l.r[i].key<l.r[i-1].key){
245                             l.r[0]=l.r[i];
246                         l.r[i]=l.r[i-1];
247                         l.r[i-1]=l.r[0];
248                         change=1;
249                 }
250         low++;
251      }
252 }
253
```

```
254 //快速排序算法
255 int Partition(RcdType R[], int low, int high){
256         R[0]=R[low];
257         KeyType pivotkey=R[low].key;
258         while(low<high){
259                 while(low<high && R[high].key>=pivotkey) --high;
260                 if(low<high) R[low++]=R[high];
261                 while(low<high && R[low].key<=pivotkey) ++low;
262                 if(low<high) R[high--]=R[low];
263         }//while
264         R[low]=R[0];
265         //printf("陈文宇");
266         return low;
267
268 } //Partition
269
270 void Qsort(RcdType R[], int s,int t){
271         int pivotloc;
272         if(s<t){
273                 pivotloc=Partition(R,s,t);
274                 Qsort(R,s,pivotloc-1);
275                 Qsort(R,pivotloc+1,t);
276         }//if
277
278 }//Qsort
279
280 void QuickSort(SqList &L){
281         Qsort(L.r,1,L.length);
282 }//QuickSort
283
284
285 //归并排序
286 void Merge(RcdType SR[], RcdType TR[], int i, int m, int n){
287         int j=m+1;
288         int k;
289         for(k=i; i<=m && j<=n; k++){
290                 if(SR[i].key<=SR[j].key) TR[k]=SR[i++];
291                 else TR[k]=SR[j++];
292         }//for
293         while(i<=m) TR[k++]=SR[i++];
294         while(j<=n) TR[k++]=SR[j++];
```

```
295
296 } //Merge
297
298 void Msort(RcdType SR[], RcdType TR1[], int s,int t,int n){
299         RcdType TR2[n];
300         int m;
301         if(s==t) TR1[s]=SR[s];
302         else{
303                 m=(s+t)/2;
304                 Msort(SR,TR2,s,m,n);
305                 Msort(SR,TR2,m+1,t,n);
306                 Merge(TR2,TR1,s,m,t);
307         }//else
308
309 }//Msort
310
311 void MergeSort(SqList &L){
312         Msort(L.r, L.r, 1, L.length,L.length+1);
313 }//MergeSort
```

A.3 第四章习题的完整代码

```
1  //陈文宇
2  //10200115
3  #include<iostream>
4  using namespace std;
5
6  typedef int QElemType;
7  typedef struct QNode {
8          QElemType data;
9      struct QNode *next;
10 } LNode, *QueuePtr;    // 结点类型
11 typedef struct{
12     QueuePtr  rear;    // 队尾指针
13 }CLinkQueue;
14
15
16 void InitCQueue(CLinkQueue &Q);
17 void EnCQueue(CLinkQueue &Q, QElemType  e);
18 bool DeCQueue(CLinkQueue &Q , QElemType &e);
19 void ListTraverse_L(CLinkQueue L);
```

```
20
21  int main(){
22          CLinkQueue Q;
23          QElemType e;
24
25          InitCQueue(Q);
26
27          EnCQueue(Q,1);
28          EnCQueue(Q,2);
29          EnCQueue(Q,3);
30          EnCQueue(Q,4);
31          cout<<"出队列前为：";
32          ListTraverse_L(Q);
33          DeCQueue(Q,e);
34          cout<<"出队列后为：";
35          ListTraverse_L(Q);
36  }
37
38
39  void InitCQueue(CLinkQueue &Q){
40          //初始化循环链表表示的队列Q
41          Q.rear = new LNode;
42      Q.rear->next=Q.rear;
43   } //InitCQueue
44
45  //入队列
46  void EnCQueue(CLinkQueue &Q, QElemType e){
47          QueuePtr p;
48          p=new LNode;
49      p->data = e;
50      p->next=Q.rear->next;
51      Q.rear->next=p;
52  }//EnCQueue
53
54  //出队列
55  bool  DeCQueue(CLinkQueue &Q , QElemType &e){
56      QueuePtr p,q;
57      if (Q.rear->next == Q.rear)
58          return false;
59      while(p->next!=Q.rear){
60          q=p;
```

```
61          p=p->next;
62          }
63      e = p->data;
64      q->next=Q.rear;
65      delete p;
66
67  }//DeCQueue
68
69  //遍历输出
70  void ListTraverse_L(CLinkQueue L){
71          QueuePtr p;
72          p=L.rear->next;
73          while(p != L.rear){
74                  cout<<p->data<<"␣";
75                  p=p->next;
76          }
77          cout<<endl;
78  }//ListTraverse_L
```

A.4 第五章习题的完整代码

```
1   //陈文宇
2   //10200115
3   #include<iostream>
4   #include<stdlib.h>
5   #include<malloc.h>
6
7   using namespace std;
8   const int MAXSIZE=100;
9   const bool TRUE=1;
10  const bool FALSE=0;
11
12  typedef int ElemType;
13
14  typedef struct{
15      int i,j;            //非零元的行下标和列下标
16      ElemType e;         //该非零元的元素值
17  }Triple;
18  typedef struct{
19      Triple data[MAXSIZE];//非零元三元组表, data[0]未用
20      int mu,nu,tu;                   //稀疏矩阵的行数, 列数和非零元个数
```

```
21  }TSMatrix;
22
23  void TSMattrans(int** M,TSMatrix &MS,int m,int n);
24  void coutMat(int** M,int m, int n);
25  void coutTSMat(TSMatrix MS);
26  bool Matrix_Addition(TSMatrix A, TSMatrix B, TSMatrix &C);
27
28
29  int main(){
30          int m=3,n=3;
31          int** M=new int*[m];
32          int** T=new int*[m];
33          TSMatrix MS,TS,QS;
34
35          //初始化
36          for(int i=0;i<m;i++){
37                  M[i]=new int[n];
38                  T[i]=new int[n];
39          }
40          MS.mu=m;            MS.nu=n;
41          TS.mu=m;            TS.nu=n;
42          for(int p=0;p<m;p++){
43                  for(int q=0;q<n;q++){
44                          M[p][q]=0;
45                          T[p][q]=0;
46                  }
47                  M[p][p]=p+1;
48                  T[p][p]=p+2;
49          }
50          M[0][n-1]=1;
51          T[m-1][0]=1;
52
53          //以三元组 形式存储
54          TSMattrans(M,MS,m,n);
55          coutMat(M,m,n);
56          cout<<"MSMatrix = "<<endl;
57          coutTSMat(MS);
58
59          TSMattrans(T,TS,m,n);
60          coutMat(T,m,n);
61          cout<<"TSMatrix = "<<endl;
```

```
62          coutTSMat(TS);
63
64          //稀疏矩阵加法
65          Matrix_Addition(MS,TS,QS);
66          cout<<"QSMatrix = "<<endl;
67          coutTSMat(QS);
68
69          //销毁矩阵
70          for(int p=0;p<m;p++){
71                  delete[] M[p];
72                  delete[] T[p];
73          }
74          delete[] M;
75          delete[] T;
76  }
77  //用于录入稀疏矩阵 并以三元组 形式存储
78  void TSMattrans(int** M,TSMatrix &MS,int m,int n){
79          int k=1;
80          for(int p=0; p<m; p++){
81                  for(int q=0; q<n; q++){
82                          if(M[p][q] != 0 ){
83                                  MS.data[k].i=p+1;
84                                  MS.data[k].j=q+1;
85                                  MS.data[k].e=M[p][q];
86                                  k++;
87                          }
88                  }
89          }
90          MS.tu=k-1;
91  }
92
93  // 三元组存储的稀疏矩阵求和算法：C=A+B
94  bool Matrix_Addition(TSMatrix A, TSMatrix B, TSMatrix &C){
95      int row_a, row_b,col_a, col_b, index_a, index_b, index_c;
96      ElemType t;
97      //行号，列号和各三元组的序号
98
99      //同类型矩阵才能相加
100     if(A.mu!=B.mu || A.nu!=B.nu) return FALSE;
101     C.mu = A.mu;          C.nu = A.nu;
102
```

```
103     //同时遍历两个三元组
104     index_a=1; index_b=1; index_c=1;
105     for( ; index_a<=A.tu&&index_b<=B.tu; ){
106     //获取行列号
107         row_a = A.data[index_a].i;      col_a = A.data[index_a].j;
108         row_b = B.data[index_b].i;      col_b = B.data[index_b].j;
109
110         //依行号访问稀疏矩阵
111         if(row_a>row_b){
112                 //B的行号小 则复制B到C
113                 C.data[index_c].i = B.data[index_b].i;
114                 C.data[index_c].j = B.data[index_b].j;
115                 C.data[index_c].e = B.data[index_b].e;
116                 //向后步进
117                 index_b++;
118                 index_c++;
119         }
120         else if(row_a<row_b){
121                 //A的行号小 则复制A到C
122                 C.data[index_c].i = A.data[index_a].i;
123                 C.data[index_c].j = A.data[index_a].j;
124                 C.data[index_c].e = A.data[index_a].e;
125                 //向后步进
126                 index_a++;
127                 index_c++;
128         }
129         else{
130                 //若同行, 则开始依列号访问稀疏矩阵
131                 if(col_a>col_b){
132                         //B的列号小, 复制B到C
133                         C.data[index_c].i = B.data[index_b].i;
134                         C.data[index_c].j = B.data[index_b].j;
135                         C.data[index_c].e = B.data[index_b].e;
136                         //向后步进
137                         index_b++;
138                         index_c++;
139                 }
140                 else if(col_a<col_b){
141                         //A的列号小, 复制A到C
142                         C.data[index_c].i = A.data[index_a].i;
143                         C.data[index_c].j = A.data[index_a].j;
```

```
144                         C.data[index_c].e = A.data[index_a].e;
145                         //向后步进
146                         index_a++;
147                         index_c++;
148                 }
149                 else{
150                         //行列号相同 ,需判断元素相加是否为零
151                 t=A.data[index_a].e+B.data[index_b].e;
152                         if(t){
153                                 C.data[index_c].i = A.data[index_a].i;
154                                 C.data[index_c].j = A.data[index_a].j;
155                                 C.data[index_c].e = t;
156                                 index_c++;
157                         }
158                         //向后步进
159                         index_a++;
160                         index_b++;
161             }
162         }
163     }
164     //B取完A未取完
165     while (index_a <= A.tu){
166         C.data[index_c].i = A.data[index_a].i;
167         C.data[index_c].j = A.data[index_a].j;
168         C.data[index_c].e = A.data[index_a].e;
169         index_a++;
170         index_c++;
171     }
172  //A取完B未取完
173     while (index_b <= B.tu){
174         C.data[index_c].i = B.data[index_b].i;
175         C.data[index_c].j = B.data[index_b].j;
176         C.data[index_c].e = B.data[index_b].e;
177         index_b++;
178         index_c++;
179     }
180     C.tu = index_c - 1;
181 }
182 //矩阵输出
183 void coutMat(int** M,int m, int n){
184         cout<<"Matrix = "<<endl;
```

```
185         for(int i=0;i<m;i++){
186                 for(int j=0;j<n;j++){
187                         cout<<M[i][j]<<"  ";
188                 }
189                 cout<<endl;
190         }
191         cout<<endl;
192 }
193 //三元组输出
194 void coutTSMat(TSMatrix MS){
195
196     for(int p=1;p<=MS.tu;p++){
197         cout<<"("<<MS.data[p].i<<",";
198         cout<<MS.data[p].j<<",";
199         cout<<MS.data[p].e<<")"<<endl;
200     }
201     cout<<endl;
202 }
```

A.5 第六章习题的完整代码

```
1  //陈文宇
2  //10200115
3  #include<iostream>
4  using namespace std;
5
6  //二叉树定义
7  typedef  char TElemType;
8  typedef struct BiTNode{
9          TElemType data;
10         struct BiTNode *lchild,*rchild;
11 }BiTNode,*BiTree;
12
13 void CreatebiTree(BiTree &T);
14 int Get_Depth(BiTree T);
15 int Get_Sub_Depth(BiTree T,TElemType x, int &depth);
16 int LeafCount(BiTree T);
17 void Preorder(BiTree T, void (*visit)(BiTree T));
18
19 int main(){
20
```

```
21          BiTree T;
22
23          //-------------------------------------------------------------
24          //树的深度
25          cout<<"先序遍历的结果为:";
26          Preorder(T, visit);
27          cout<<endl;
28          int h=1,depth=0;
29          BiTreeDepth(T,h,depth);
30          cout<<"树的深度为: "<<depth<<endl;
31          cout<<endl;
32
33          //条件子树的深度
34          cout<<"以B为根的树的深度: ";
35          Get_Sub_Depth(T,'B',depth);
36          cout<<depth<<endl;
37          cout<<endl;
38
39          cout<<"先序遍历的结果为:";
40          Preorder(T, visit);
41          cout<<endl;
42          //二叉树的叶子结点个数
43          cout<<"叶子结点个数:";
44          cout<<LeafCount(T)<<endl;
45          cout<<endl;
46  }
47  //二叉链表创建二叉树
48  void CreatebiTree(BiTree &T){
49          TElemType ch;
50          cin>>ch;
51          if(ch=='#')T=NULL;//使用整型时请用 0 代替
52          else{
53                  T=new BiTNode;
54                  T->data=ch;
55                  CreatebiTree(T->lchild);
56                  CreatebiTree(T->rchild);
57          }
58  }//CreatebiTree
59
60  //求子树深度的递归算法
61  int Get_Depth(BiTree T) {
```

```
62          int m,n;
63          if(!T)
64                  return 0;
65          else {
66                  m=Get_Depth(T->lchild);
67                  n=Get_Depth(T->rchild);
68                  return (m>n?m:n)+1;
69          }
70  } //Get_Depth
71
72  int Get_Sub_Depth(BiTree T,TElemType x, int &depth){
73          if(T->data==x){
74                  depth=Get_Depth(T);
75                  return 0;
76          }
77          else{
78                  if(T->lchild)
79                          Get_Sub_Depth(T->lchild,x,depth);
80                  if(T->rchild)
81                          Get_Sub_Depth(T->rchild,x,depth);
82     }
83  }
84  //求二叉树中叶子结点的数目
85  int LeafCount(BiTree T){
86          if(!T)  return 0;
87          else if(!T->lchild && !T->rchild)
88          return 1;
89          else return LeafCount(T->lchild)+LeafCount(T->rchild);
90
91  } //LeafCount
92
93  //先序遍历  (递归)
94  void Preorder(BiTree T, void (*visit)(BiTree)){
95          if(T){
96                  visit(T);
97                  Preorder(T->lchild,visit);
98                  Preorder(T->rchild,visit);
99          }
100 }
```