第 2 章 基本线性结构

1. 选择题

（1）A （2）A （3）D （4）C （5）C （6）B （7）C （8）B （9）A （10）D
（11）B （12）D （13）C （14）B （15）D （16）B （17）B （18）C （19）B （20）D
（21）C （22）C

2. 判断题

（1）X （2）√ （3）X （4）√ （5）X （6）√ （7）X （8）√ （9）√ （10）X
（11）X（12）X （13）√ （14）√ （15）√（16）X （17）X

3. 简答题

1．循环队列的优点是什么？如何判别它的空和满？

循环队列的优点是能够克服“假溢满”现象。

设有循环队列 sq，队满的判别条件为：

（sq->rear+1）%maxsize==sq->front;或 sq->num==maxsize。

队空的判别条件为：

sq->rear==sq->front。

2．栈和队列数据结构各有什么特点，什么情况下用到栈，什么情况下用到队列？

栈和队列都是操作受限的线性表，栈的运算规则是“后进先出”，队列的运算规则是“先进先出”。栈的应用如数制转换、递归算法的实现等，队列的应用如树的层次遍历等。

3．什么是递归？递归程序有什么优缺点？

一个函数在结束本函数之前，直接或间接调用函数自身，称为递归。例如，函数 f 在执行中，又调用函数 f 自身，这称为直接递归；若函数 f 在执行中，调用函数 g，而 g 在执行中，又调用函数 f，这称为间接递归。在实际应用中，多为直接递归，也常简称为递归。

递归程序的优点是程序结构简单、清晰，易证明其正确性。缺点是执行中占内存空间较多，运行效率低。

4．设有编号为 1,2,3,4 的四辆车，顺序进入一个栈式结构的站台，试写出这四辆车开出车站的所有可能的顺序（每辆车可能入站，可能不入站，时间也可能不等）。

1234，1243，1324，1342，1432，213，2143，2314，2341，2431，3214，3241，3421，4321

二. 算法设计题

1．设线性表存放在向量 A[arrsize]的前 elenum 个分量中，且递增有序。试写一算法，将 x 插入到线性表的适当位置上，以保持线性表的有序性，并且分析算法的时间复杂度。

【提示】直接用题目中所给定的数据结构（顺序存储的思想是用物理上的相邻表示逻辑上的相邻，不一定将向量和表示线性表长度的变量封装成一个结构体），因为是顺序存储，分配的存储空间是固定大小的，所以首先确定是否还有存储空间，若有，则根据原线性表中元素的有序性，来确定插入元素的插入位置，后面的元素为它让出位置，（也可以从高下标端开始一边比较，一边移位）然后插入 x ，最后修改表示表长的变量。

```
int insert (datatype A[],int *elenum,datatype x)          /*设 elenum 为表的最大下标*/
    {if (*elenum==arrsize-1)    return 0;                 /*表已满，无法插入*/
     else {i=*elenum;
           while (i>=0 && A[i]>x)                         /*边找位置边移动*/
              {A[i+1]=A[i];
               i--;
              }
           A[i+1]=x;                                     /*找到的位置是插入位的下一位*/
           (*elenum)++;
           return 1;                                     /*插入成功*/
          }
    }
```

时间复杂度为 O(n)。

2．已知一顺序表 A，其元素值非递减有序排列，编写一个算法删除顺序表中多余的值相同的元素。

【提示】对顺序表 A，从第一个元素开始，查找其后与之值相同的所有元素，将它们删除；再对第二个元素做同样处理，依此类推。

```
void delete(Seqlist *A)
    {i=0;
     while(i<A->last)                                  /*将第 i 个元素以后与其值相同的元素删除*/
         {k=i+1;
          while(k<=A->last&&A->data[i]==A->data[k])
                k++;                                   /*使 k 指向第一个与 A[i]不同的元素*/
          n=k-i-1;                                     /*n 表示要删除元素的个数*/
          for(j=k;j<=A->last;j++)
                A->data[j-n]=A->data[j];               /*删除多余元素*/
          A->last= A->last -n;
          i++;
         }
    }
```

3．写一个算法，从一个给定的顺序表 A 中删除值在 x~y(x<=y)之间的所有元素，要求以较高的效率来实现。

【提示】对顺序表 A，从前向后依次判断当前元素 A->data[i]是否介于 x 和 y 之间，若是，并不立即删除，而是用 n 记录删除时应前移元素的位移量；若不是，则将 A->data[i]向前移动 n 位。n 用来记录当前已删除元素的个数。

```
void delete(Seqlist *A,int x,int y)
    {i=0;
     n=0;
     while (i<A->last)
         {if (A->data[i]>=x && A->data[i]<=y)     n++;  /*若 A->data[i] 介于 x 和 y 之间，n 自增*/
          else    A->data[i-n]=A->data[i];              /*否则向前移动 A->data[i]*/
          i++;
         }
     A->last-=n;
    }
```

4．线性表中有 n 个元素，每个元素是一个字符，现存于向量 R[n]中，试写一算法，使 R 中的字符按字母字符、数字字符和其它字符的顺序排列。要求利用原来的存储空间，元素移动次数最小。

【提示】对线性表进行两次扫描，第一次将所有的字母放在前面，第二次将所有的数字放在字母之后，其它字符之前。

```
int fch(char c)                                          /*判断 c 是否字母*/
   {if(c>='a'&&c<='z'||c>='A'&&c<='Z')   return (1);
    else      return (0);
    }
int fnum(char c)                                         /*判断 c 是否数字*/
   {if(c>='0'&&c<='9')          return (1);
     else      return (0);
   }
void process(char R[n])
   {low=0;
    high=n-1;
    while(low<high)                                      /*将字母放在前面*/
        {while(low<high&&fch(R[low]))    low++;
         while(low<high&&!fch(R[high]))    high--;
         if(low<high)
            {k=R[low];
             R[low]=R[high];
             R[high]=k;
            }
        }
    low=low+1;
    high=n-1;
    while(low<high)                                      /*将数字放在字母后面，其它字符前面*/
        {while(low<high&&fnum(R[low])) low++;
         while(low<high&&!fnum(R[high])) high--;
         if(low<high)
             {k=R[low];
              R[low]=R[high];
              R[high]=k;
              }
        }
   }
```

5．线性表用顺序存储，设计一个算法，用尽可能少的辅助存储空间将顺序表中前 m 个元素和后 n 个元素进行整体互换。即将线性表：

（$a_1, a_2, \dots, a_m, b_1, b_2, \dots, b_n$）改变为：

（$b_1, b_2, \dots, b_n, a_1, a_2, \dots, a_m$）。

【提示】比较 m 和 n 的大小，若 m<n，则将表中元素依次前移 m 次；否则,将表中元素依次后移 n 次。

```
void process(Seqlist *L,int m,int n)
    {if(m<=n)
       for(i=1;i<=m;i++)
          {x=L->data[0];
           for(k=1;k<=L->last;k++)
               L->data[k-1]=L->data[k];
           L->data[L->last]=x;
```

```
        }
    else for(i=1;i<=n;i++)
          {x=L->data[L->last];
           for(k=L->last-1;k>=0;k- -)
               L->data[k+1]=L->data[k];
            L->data[0]=x;
          }
    }
```

6．已知带头结点的单链表 L 中的结点是按整数值递增排列的，试写一算法，将值为 x 的结点插入到表 L 中，使得 L 仍然递增有序，并且分析算法的时间复杂度。

```
LinkList insert(LinkList L, int x)
  {p=L;
    while(p->next && x>p->next->data)
       p=p->next;                                  /*寻找插入位置*/
    s=(LNode *)malloc(sizeof(LNode));              /*申请结点空间*/
    s->data=x;                                     /*填装结点*/
    s->next=p->next;
    p->next=s;                                     /*将结点插入到链表中*/
    return(L);
    }
```

7．假设有两个已排序（递增）的单链表 A 和 B，编写算法将它们合并成一个链表 C 而不改变其排序性。

```
LinkList Combine(LinkList A, LinkList B)
    {C=A;
     rc=C;
     pa=A->next;              /*pa 指向表 A 的第一个结点*/
     pb=B->next;              /*pb 指向表 B 的第一个结点*/
     free(B);                 /*释放 B 的头结点*/
     while (pa && pb)         /*将 pa、pb 所指向结点中，值较小的一个插入到链表 C 的表尾*/
          if(pa->data<pb->data)
              {rc->next=pa;
               rc=pa;
               pa=pa->next;
               }
          else
               {rc->next=pb;
                rc=pb;
                pb=pb->next;
               }
     if(pa)     rc->next=pa;
     else       rc->next=pb;      /*将链表 A 或 B 中剩余的部分链接到链表 C 的表尾*/
     return(C);
    }
```

8．假设长度大于 1 的循环单链表中，既无头结点也无头指针，p 为指向该链表中某一结点的指针，编写算法删除该结点的前驱结点。

【提示】利用循环单链表的特点，通过 s 指针可循环找到其前驱结点 p 及 p 的前驱结点 q，然后可删除结

点*p。

```
viod delepre(LNode *s)
    {LNode    *p, *q;
     p=s;
     while (p->next!=s)
          {q=p;
           p=p->next;
          }
     q->next=s;
     free(p);
    }
```

9．已知两个单链表 A 和 B 分别表示两个集合，其元素递增排列，编写算法求出 A 和 B 的交集 C，要求 C 同样以元素递增的单链表形式存储。

【提示】交集指的是两个单链表的元素值相同的结点的集合，为了操作方便，先让单链表 C 带有一个头结点，最后将其删除掉。算法中指针 p 用来指向 A 中的当前结点，指针 q 用来指向 B 中的当前结点，将其值进行比较，两者相等时，属于交集中的一个元素，两者不等时，将其较小者跳过，继续后面的比较。

```
LinkList Intersect(LinkList A, LinkList B)
     {LNode *q, *p, *r, *s;
      LinkList C;
      C= (LNode *)malloc(sizeof(LNode));
      C->next=NULL;
      r=C;
      p=A;
      q=B;
      while (p && q )
           if (p->data<q->data) p=p->next;
           else    if (p->data==q->data)
                      {s=(LNode *)malloc(sizeof(LNode));
                       s->data=p->data;
                       r->next=s;
                       r=s;
                       p=p->next;
                       q=q->next;
                      }
                   else    q=q->next;
      r->next=NULL;
      C=C->next;
      return C;
      }
```

10．设有一个双向链表，每个结点中除有 prior、data 和 next 域外，还有一个访问频度 freq 域，在链表被起用之前，该域的值初始化为零。每当在链表进行一次 Locata(L,x)运算后，令值为 x 的结点中的 freq 域增 1，并调整表中结点的次序，使其按访问频度的非递增序列排列，以便使频繁访问的结点总是靠近表头。试写一个满足上述要求的 Locata(L,x)算法。

【提示】在定位操作的同时，需要调整链表中结点的次序：每次进行定位操作后，要查看所查找结点的 freq 域，将其同前面结点的 freq 域进行比较，同时进行结点次序的调整。

```
typedef struct dnode
```

```
    {datatype data;
     int freq;
     struct DLnode *prior,*next;
    }DLnode,*DLinkList;
DlinkList Locate(DLinkList L, datatype x)
    {p=L->next;
     while(p&&p->data!=x) p=p->next;                    /*查找值为 x 的结点，使 p 指向它*/
     if(!p) return(NULL);                               /*若查找失败，返回空指针*/
          p->freq++;                                    /*修改 p 的 freq 域*/
     while(p->prior!=L&&p->prior->freq<p->freq)         /*调整结点的次序*/
        {k=p->prior->data;
         p->prior->data=p->data;
         p->data=k;
         k=p->prior->freq;
         p->prior->freq=p->freq;
         p->freq=k;
         p=p->prior;
        }
     return(p);                                         /*返回找到的结点的地址*/
    }
```

11．称正读和反读都相同的字符序列为“回文”，例如，“abcddcba”、“qwerewq”是回文，“ashgash”不是回文。试写一个算法判断读入的一个以‘@’为结束符的字符序列是否为回文。

【提示】使用栈。将字符串的前一半入栈，再依次出栈，与后一半进行比较，若有不等则不是回文，若依次相等，则是回文。

```
int Huiwen(char a[ ], int n)
    {SeqStack *S;
     char *str;
     int i=0,j=1;
     Init_Stack(S);                          /*初始化栈 s*/
     while (i<n/2)                           /*字符串的前一半入栈 s*/
          {Push_Stack(S,a[i]);
           i=i+1;
          }
      if(n %2!=0) i=i+1;                     /*若 n 为奇数 i 加 1，越过中间的一个数*/
      while(i<n && j==1)
          if (a[1]==Top_Stack(S))
              {Pop_Stack(S,str);
               i=i+1;
              }
          else{j=0;
               break;
              }
      return(j);                             /*返回值 j＝0，则不是回文；若 j＝1，则是回文*/
    }
```

12．设以数组 se[m]存放循环队列的元素，同时设变量 rear 和 front 分别作为队头队尾指针，且队头指针指向队头前一个位置，写出这样设计的循环队列入队和出队算法。

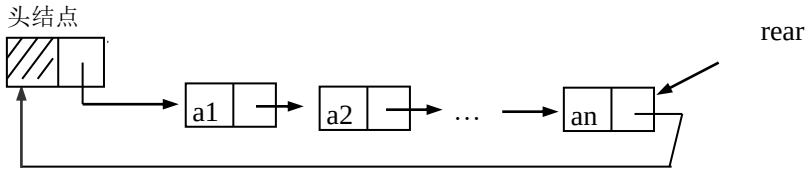
① int EnQueue(datatype se[m],int rear,int front,datatype e)
{if((rear+1)%m==front) return(-1) /*若队列满，返回-1*/
rear=(rear+1)%m;
se[rear]=e;
return(1);
}
② int DeQueue(datatype se[m],int rear,int front,datatype *e)
{if(rear==front) return(-1); /*若队空，返回-1*/
Front=(front+1)%m;
*e=se[front];
return(1);
}

13．假设以数组 se[m]存放循环队列的元素，同时设变量 rear 和 num 分别作为队尾指针和队中元素个数记录，试给出判别此循环队列的队满条件，并写出相应入队和出队算法。

① int EnQueue(datatype se[m],int rear,int num,datatype e)
{if(num==m) return(-1);
rear=(rear+1)%m;
se[rear]=e;
num++;
return(1);
}
② int DeQueue(datatype se[m],int rear,int num,datatype *e)
{if(num==0) return(-1);
*e=se[(rear-num+1)%m];
num--;
return(1);
}

14．假设以带头结点的循环链表表示一个队列，并且只设一个队尾指针指向尾元素结点（注意不设头指针），试写出相应的置空队、入队、出队的算法。

【提示】算法思想：带头结点的循环链表表示的队列如下图所示，仅有队尾指针 rear，但可通过 rear->next 找到头结点，再通过头结点找到队头，即 rear->next->next。



①置空队
LinkList SetNull ( )
{LinkList rear;
rear=(LNode *)malloc(sizeof(LNode));
rear->next=rear;
return(rear);
}
②入队
LinkList EnQueue(LinkList rear, datatype x)
{LinkList p;

```
    p=malloc(sizeof(LinkList));
    p->data=x;
    p->next=rear->next;                    /*将 p 插入到 rear->next 之后*/
    rear->next=p;
    rear=p;
    return   rear;                         /*返回新的队尾指针*/
   }
```

③出队

```
Linklist DeQueue(Linklist rear, datatype *x)
   {if(rear->next==rear)                   /*若队空，则输出队空信息*/
       printf("EMPTY");
    else {q=rear->next;
          p=q->next;
         }                                 /*否则 q 指向头结点，p 指向队头*/
    if(p==rear)   rear=q;                  /*若队中仅有一个元素，则将 rear 指向头结点*/
    q->next=p->next;                       /*将 p 所指结点出队*/
    *x=p->data;
    free(p);                               /*将对头结点的值赋给形参*x*/
    return(rear);                          /*返回队尾指针*/
   }
```

15．设计一个算法判别一个算术表达式的圆括号是否正确配对。

【提示】如遇左括号就压入栈中，如遇右括号则与栈顶元素比较，如是与其配对的括号，则弹出栈顶元素；否则，括号就不配对。

```
int process()
   {Init_Stack(S);
    scanf("%c", &c);                                  /*读入字符 c*/
    while(c!='@')                                     /*设表达式以@为结束标志*
        {if(c=='(') Push_Stack(S,c);                  /*当前字符是左括号，则入栈*/
         if(c==')')
            {if(!empty_stack(s))   Pop_Stack(s, a);
             else return(-1);
            }                  /*当前字符是右括号时，判断栈的状态：若栈空，则返回-1，否则出栈*/
         scanf("%c", &c);
        }
    if(Empty_Stack(S))   return(1);
    else   return(-1);                    /*若栈空，说明表达式中括号配对，返回 1；否则，返回-1*/
   }
```

16．编写算法，借助于栈将一个单链表置逆。

【提示】利用栈后进先出的特点，将单链表中的结点从链表头开始依次压栈，然后再依次出栈，采用尾插法重新生成单链表。

```
void reverse(LinkList A)
      {p=A->next;
       r=A;
       A->next=NULL;
       Iinit_Stack(S);                                    /*初始化栈*/
       while(p)                                           /*将链表中结点依次压栈*/
```

```
            {Push_Stack(S,p);
             p=p->next;
            }
        while(!Empty_Stack(S))                                  /*将栈中元素依次出栈*/
            {Pop_Stack(S, q);
             r->next=q;
             r=q;
            }
         r->next=NULL;
       }
```

17．两个栈共享向量空间 v[m]，它们的栈底分别设在向量的两端，每个元素占一个分量，试写出两个栈公用的栈操作算法：push(S,i,x)、pop(S,i)和 top(S,i)，其中 s 表示栈，i=0 和 1 ,用以指示栈号。

【提示】双向栈是两个栈共享一个向量空间，栈 0 的底下标为 0，栈 1 的底下标为为 m－1，初始时栈 0 的顶为－1，栈 1 的顶为 m；当栈 0 有元素进栈时，让栈 0 的栈顶指针加 1；退栈时减 1。当栈 1 有元素进栈时，让栈 1 的栈顶指针减 1，退栈时加 1。

```
  #define   m   100                          /*设定双向栈的向量空间大小*/
  typedef   int   datatype                   /*定义栈元素类型*/
  typedef   struct
  {datatype v[m];                            /*定义栈空间*/
    int top0,top1;                           /*top0 为 0 栈栈顶位置，top1 为 1 栈栈顶位置*/
  }dstack;
①void push(dstack *S, int i, datatype x)
     {if (i!=0 || i!=1) printf("error!");
      else if(i==1)
             {if(S->top1-1==S->top0)   printf("overflow!");          /*栈已满*/
              else {S->top1--;
                    S->v[S->top1]=x; }
             }
          else {if (S->top0＋1== S->top1)   printf("overflow!");     /*栈已满*/
                else{S->top0++;
                     S->v[S->top0]=x;}
                }
     }
②datatype pop(dstack *S, int i)
     {if(i!=0 ||i!=1) printf("error!");
      else if(i==1)
             {if(S->top1==m)   printf("underflow!");                 /*栈 1 空*/
              else { S->top1++;
                     return(S->v[S->top1-1]);}
             }
           else { if(S->top0==-1)   printf("underflow!");             /*栈 0 空*/
                  else {S->top0--;
                        return(S->v[S->top0+1]);}
                }
     }
③datatype   top(dstack *S, int i)
```

```
    {if(i!=0 ||i!=1) printf("error!");
     else if (i==1)
                if (S->top1==m)    printf("underflow!");                    /*栈 1 空*/
                 else    return(S->v[S->top1]);
          else if (i==0)
                if(S->top0==-1) printf("underflow!");                       /*栈 0 空*/
                else      return(S->v[S->top0]);
}
```