2.16

2.16③ 已知指针 la 和 lb 分别指向两个无头结点单链表中的首元结点。下列算法是从表 la 中删除自第 *i* 个元素起共 *len* 个元素后，将它们插入到表 lb 中第 *i* 个元素之前。试问此算法是否正确？若有错，则请改正之。

```
Status DeleteAndInsertSub (LinkedList la, LinkedList lb , int i, int j, int len {
    if (i<0 || j<0 ||len<0) return INFEASIBLE;
        p =la;  k =1;
        while (k<i) { p =p->next;  k++; }
        q =p;
        while (k<=len) { q =q->next; k++; }
        s =lb; k =1;
        while (k<j) { s =s->next;  k++; }
        s->next =p;  q->next =s->next;
        return OK;
    }//DeleteAndInsertSub.
```

参考答案：

注意此题中的条件是，采用的存储结构(单链表)中无头结点，因此在写算法时，特别要注意空表和第一个结点的处理。算法中尚有其他类型的错误，如结点的计数，修改指针的次序等。此题的正确算法如下：

```
Status DeleteAndInsertSub (LinkedList &la, LinkedList &lb, int i, int j, int len )
{
  // la 和 lb 分别指向两个单链表中第一个结点，本算法是从 la 表中删去自第 i 个
  // 元素起共 len 个元素，并将它们插入到 lb 表中第 j 个元素之前，若 lb 表中只
  // 有 j-1 个元素，则插在表尾。
  // 入口断言：(i>0) ∧ (j>0) ∧ (len>0)
    if (i<0 || j<0 || len<0 ) return INFEASIBLE;
    p =la; k =1; prev=NULL;
    while (p && k<i )       // 在 la 表中查找第 i 个结点
      { prev=p; p =p->next; k++; }
    if (!p ) return INFEASIBLE;
    q =p; k=1;              // p 指向 la 表中第 i 个结点
    while (q && k<len )
      { q =q->next; k++; }  // 查找 la 表中第 i+len-1 个结点
    if (!q ) return INFEASIBLE;
    if (!prep )   la=q->next;       // i=1 的情况
    else  prep->next=q->next;       // 完成删除
    // 将从 la 中删除的结点插入到 lb 中
    if (j==1) {q->next=lb; lb=p; }
    else {   // j≥2
      s =lb; k =1;
      while (s && k<j-1 ) { s =s->next; k++; }
                // 查找 lb 表中第 j-1 个元素
      if (!s ) return INFEASIBLE;
      q->next =s->next; s->next =p; // 完成插入
      return OK;
    }
}//DeleteAndInsertSub
```

3.9

3.9③ 试将下列递推过程改写为递归过程。

```
void ditui(int n) {
   int i;
   i=n;
   while (i>1)
     printf(i--);
}
```

参考答案:

该递推过程可改写为下列递归过程:

```
void digui(int j)
{
  if (j>1) {
    printf(j);
    digui(j-1);
   }
} // digui
```

由于该递归过程中的递归调用语句出现在过程结束之前,俗称"尾递归",因此可以不设栈,而通过直接改变过程中的参数值,利用循环结构代替递归调用。

3.29

3.29③ 如果希望循环队列中的元素都能得到利用,则需设置一个标志域 tag,并以 tag 的值为 0 或 1 来区分,尾指针和头指针值相同时的队列状态是"空"还是"满"。试编写与此结构相应的入队列和出队列的算法,并从时间和空间角度讨论设标志和不设标志这两种方法的使用范围(如当循环队列容量较小而队列中每个元素占的空间较多时,哪一种方法较好)。

参考答案:

```
#define MaxQSize   4
typedef int ElemType;
typedef struct{
     ElemType *base;
     int front;
     int rear;
     Status tag;
}Queue;
Status InitQueue(Queue& q)
{
     q.base=new ElemType[MaxQSize];
     if(!q.base) return FALSE;
     q.front=0;
     q.rear=0;
     q.tag=0;
     return OK;
}
 Status EnQueue(Queue& q,ElemType e)
 {
      if(q.front==q.rear&&q.tag) return FALSE;
      else{
           q.base[q.rear]=e;
           q.rear=(q.rear+1)%MaxQSize;
           if(q.rear==q.front)q.tag=1;
```

```
        }
        return OK;
    }
    Status DeQueue(Queue& q,ElemType& e)
    {
        if(q.front==q.rear&&!q.tag)return FALSE;
        else{
            e=q.base[q.front];
            q.front=(q.front+1)%MaxQSize;
            q.tag=0;
        }
        return OK;
    }
```

设标志节省存储空间，但运行时间较长。不设标志则正好相反。

5.10

5.10②　求下列广义表操作的结果：

(1) GetHead【(*p*,*h*,*w*)】;

(2) GetTail【(*b*,*k*,*p*,*h*)】;

(3) GetHead【((*a*,*b*),(*c*,*d*))】;

(4) GetTail【((*a*,*b*),(*c*,*d*))】;

(5) GetHead【GetTail【((*a*,*b*),(*c*,*d*))】】;

(6) GetTail【GetHead【((*a*,*b*),(*c*,*d*))】】;

(7) GetHead【GetTail【GetHead【((*a*,*b*),(*c*,*d*))】】】;

(8) GetTail【GetHead【GetTail【((*a*,*b*),(*c*,*d*))】】】。

注意：【】是函数的符号。

参考答案

(1)(a, b)(2)(c,d)(3)b(4)(d)

6.43

编写递归算法，将二叉树中所有结点的左、右子树相互交换。

参考答案:

```
#include<stdio.h>
#include<malloc.h>
typedef struct binode{
int data;
struct binode *lchild,*rchild;
}binode,*bitree;
typedef struct{
bitree elem[100];
int top;
}stack;
bitree creat_bt(){ //按扩展前序建二叉树
bitree t;int x;
scanf("%d",&x);
if (x==0) t=NULL;
else { t=(bitree)malloc(sizeof(binode));
t->data=x;
t->lchild=creat_bt();
t->rchild=creat_bt();
}
```

```
}
return t;
}
void exchange(bitree t) //左、右子树交换
{bitree p;
if(t!=NULL)
{ p=t->lchild;t->lchild=t->rchild;
t->rchild=p;
exchange(t->lchild);
exchange(t->rchild);
}
}
void inorder(bitree bt) //递归的中序遍历
{ if (bt){
inorder(bt->lchild);
printf("% d",bt->data);
inorder(bt->rchild);
}
}
main()
{bitree root;
printf("\n");
printf("建二叉树，输入元素：");
root=creat_bt(); /*create tree of useing preorder*/
printf("交换前的中序序列是：");
inorder(root);
 exchange(root);
 printf("\n交换后的中序序列是：");
 inorder(root);
 printf("\n");
 }
```

6.65:

◆6.65④ 已知一棵二叉树的前序序列和中序序列分别存于两个一维数组中，试编写算法建立该二叉树的二叉链表。

参考答案:

```
BiTree Resume_BiTree(TElemType *pre,TElemType *mid,int prelen,int midlen)
//6-65 前序序列和中序序列求出二叉树
{
    BiTree want;
    if(! (want=(BiTree)malloc(sizeof(BiTNode)))) exit(OVERFLOW);
    if(prelen==0&&midlen==0)
    return NULL;
    want->data=pre[0];
    int rootposition=0;
    if(pre[0]==mid[0])
    want->lchild=NULL;
    else
    {
        rootposition=SearchNum(want->data,mid,midlen);
        want->lchild=Resume_BiTree(pre+1,mid,rootposition,rootposition);
```

```
    }
    if(pre[0]==mid[midlen-1])
    want->rchild=NULL;
    else
    {
        want->rchild=Resume_BiTree(pre+rootposition+1,mid+rootposition+1,prelen-
rootposition-1,midlen-rootposition-1);
    }
    return want;
}
```

7.15

◆7.15③ 试在邻接矩阵存储结构上实现图的基本操作：InsertVex(G,v)，InsertArc(G,v,w)，DeleteVex(G,v)和 DeleteArc(G,v,w)。

```
Status Insert_Vex(MGraph &G, char v)//在邻接矩阵表示的图 G 上插入顶点 v
{
if(G.vexnum+1)>MAX_VERTEX_NUM return INFEASIBLE;
G.vexs[++G.vexnum]=v;
return OK;
}//Insert_Vex

Status Insert_Arc(MGraph &G,char v,char w)//在邻接矩阵表示的图 G 上插入边(v,w)
{
if((i=LocateVex(G,v))<0) return ERROR;
if((j=LocateVex(G,w))<0) return ERROR;
if(i==j) return ERROR;
if(!G.arcs[j].adj)
{
arcs[j].adj=1;
G.arcnum++;
}
return OK;
}//Insert_Arc

Status Delete_Vex(MGraph &G,char v)//在邻接矩阵表示的图 G 上删除顶点 v
{
n=G.vexnum;
if((m=LocateVex(G,v))<0) return ERROR;
G.vexs[m]<->G.vexs[n]; //将待删除顶点交换到最后一个顶点


for(i=0;i<n;i++)
{
G.arcs[m]=G.arcs[n];
G.arcs[m]=G.arcs[n]; //将边的关系随之交换
}
G.arcs[m][m].adj=0;
G.vexnum--;
return OK;
}//Delete_Vex

Status Delete_Arc(MGraph &G,char v,char w)//在邻接矩阵表示的图 G 上删除边(v,w)
{
if((i=LocateVex(G,v))<0) return ERROR;
if((j=LocateVex(G,w))<0) return ERROR;
if(G.arcs[j].adj)
{
G.arcs[j].adj=0;
G.arcnum--;
}
return OK;
}//Delete_Arc
```

7.29

参考答案

```
int GetPathNum_Len(ALGraph G,int i,int j,int len)//求邻接表方式存储的有向图 G 的顶点 i 到
j 之间长度为 len 的简单路径条数
{
if(i==j&&len==0) return 1; //找到了一条路径,且长度符合要求
else if(len>0)
{
sum=0; //sum 表示通过本结点的路径数
visited[i]=1;
for(p=G.vertices[i].firstarc;p;p=p->nextarc)
{
l=p->adjvex;
if(!visited[l])
sum+=GetPathNum_Len(G,l,j,len-1)//剩余路径长度减一
}//for
visited[i]=0;
}//else
return sum;
}//GetPathNum_Len
```