第二章

2.16 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.
```

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
```

2.19 已知线性表中的元素以值递增有序排列，并以单链表作存储结构。试写一高效的算法，删除表中所有值大于 mink 且小于 maxk 的元素(若表中存在这样的元素)，同时释放被删结点空间，并分析你的算法的时间复杂度(注意：mink 和 maxk 是给定的两个参变量，它们的值可以和表中的元素相同，也可以不同)。

```
Status ListDelete_L(LinkList &L,ElemType mink,ElemType maxk)
{
    LinkList p,q,prev=NULL;
    if(mink>maxk)return ERROR;
    p=L;
    prev=p;
    p=p->next;
    while(p&&p->data<maxk){
        if(p->data<=mink){
            prev=p;
            p=p->next;
        }
        else{
            prev->next=p->next;
            q=p;
            p=p->next;
            free(q);
        }
    }
    return OK;
}
```

2.22 试写一算法，对单链表实现就地逆置。

1.头插法：将表尾元素取出，头插法插到表头

```
void converse(LinkList *head)
{
    LinkList *p,*q;
    p=head->next;
    head->next=NULL;
    while(p)
    {
        /*向后挪动一个位置*/
        q=p;
        p=p->next;
        /*头插*/
        q->next=head->next;
        head->next=q;
    }
  }
```

2.递归：将表头结点从链表中拆出来，然后对其余部分进行逆序，最后将当前的表头结点链接到逆序链表的尾部

```
ListNode *reverse(ListNode *head)
{
```

```
        if(head==NULL || head->next ==NULL)
            return head;
        /*递归*/
        ListNode* headOfReverse = reverse(head->next);
        // cout<<head->next<<" "<<headOfReverse<<endl;
        /*回溯：将当前表头结点链接到逆序链表的尾部*/
        head->next->next = head;
        head->next = NULL;
        return headOfReverse;
}
```

2.38 设有一个双向循环链表，每个结点中除有 prc,data 和 next 三个域外，还增设了一个访问频度域 freq。在链表被起作用之前，频度域 freq 的值均初始化为零，而每当对链表进行一次 LOCATE(L，x)的操作后，被访问的节点（即元素值等于 x 的结点）中的频度域 freq 的值便增 1，同时调整链表中结点之间的次序，使其按访问频度非递增的次序顺序排列，以便始终保持被频繁访问的节点总是靠近表头节点。试编写符合上述要求的 LOCATE 操作的算法。

```
DuLinkList ListLocate_DuL(DuLinkList &L,ElemType e)
{
    DuLinkList p,q;
    p=L->next;
    while(p!=L && p->data!=e) p=p->next;
    if(p==L) return NULL;
    else{
        p->freq++; //  删除结点
        p->pre->next=p->next;
        p->next->pre=p->pre; //  插入到合适的位置
        q=L->next;
        while(q!=L && q->freq>p->freq) q=q->next;
        if(q==L){
            p->next=q->next;
            q->next=p;
            p->pre=q->pre;
            q->pre=p;
        }
        else{ //  在  q  之前插入
            p->next=q->pre->next;
            q->pre->next=p;
            p->pre=q->pre;
            q->pre=p;
        }
        return p;
    }
}
```

第三章

3.6 试证明：若借助栈由输入序列 12…n 得到的输出序列为 $p_1p_2…p_n$(它是输入序列的个排列)，则在输出序列中不可能出现这样的情形：存在着 i<j<k，使 $p_j<p_k<p_i$.

证明：因为输入序列是从小到大排列的，所以若 $p_j<p_k<p_i$，则可以理解为通过输入序列 p_j, p_k, p_i 可以得到输出序列 p_i, p_j, p_k，显然通过序列 123 是无法得到 312 的,所以不可能存在着 i<j<k 使 $p_j<p_k<p_i$.

3.9 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--);
}
```

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

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。

3.22 如题 3.21 的假设条件，试写一个算法，对逆波兰式表示的表达式求值。

```
char CalVal_InverPoland(char Buffer[])
{
    Stack Opnd;
    InitStack(Opnd);
    int i=0;
    char c;
    ElemType e1, e2;
    while(Buffer[i]!='#')
    {
        if(!Is0perator(Buffer[i]))
        {
            Push(Opnd, Buffer[i]);
        )
        else
        {
            Pop(Opnd, e2);
            Pop(Opnd, e1);
```

```
                c=Cal(e1,Buffer[i], e2);
                Push(Opnd, c);
            }
            i++;
        }
        return c;
}

char Cal(char c1, char op, char c2)
{
        int x, x1, x2;
        char ch[10];
        ch[0]=c1;
        ch[1]='\0';
        xl=atoi(ch);
        ch[0]=c2;
        ch[1]='\0';
        x2=atoi(ch);
        switch(op)
        {
            case '+': x=x1+x2; break;
            case '-': x=x1-x2; break;
            case '*': x=x1*x2; break;
            case '/': x=x1/x2; break ;
            default: break;
        }
        itoa(x,ch,10);
        return ch[0];
}
```

3.24 试编写如下定义的递归函数的递归算法,并根据算法画出求 g(5,2)时找的变化过程。

$$g(m,n)=\begin{cases}0 & m=0,n\geqslant 0\\ g(m-1,2n)+n & m>0,n\geqslant 0\end{cases}$$

```
int g(m,n)
{
        if(m<0 || n<0)    exit(error)
        if(m==0) return 0;
        else return g(m-1,2*n)+n;
}
```

国际象棋之跳马

```
#include <iostream>
using namespace std;
```

```
const int m = 5, n = 5;
int countN = 0;
int trace[m][n] = {0};
int changedir[][2] = { {1,2},{2,1},{1,-2},{2,-1},{-1,2},{-2,1},{-1,-2},{-2,-1} };

void printT()
{
	for (int i = 0; i < m; i++)
	{
		for (int j = 0; j < n; j++)
		cout << trace[i][j] << '\t';
	cout << endl;
	}
	cout << "------------" << endl;
}

void visit(int x, int y, int step)
{
	trace[x][y] = step;
	if (m*n == step)
	{
		countN++;
		printT();
		return;
	}
	int nextx, nexty;
	for (int i = 0; i < 8; i++)
	{
		nextx = x + changedir[i][0];
		nexty = y + changedir[i][1];
		if (nextx < 0 || nextx>(m-1) || nexty < 0 || nexty>(n-1)|| trace[nextx][nexty]!=0)
		continue;
		visit(nextx, nexty, step + 1);
		trace[nextx][nexty] = 0;
	}
}

void main()
{
	int firstX , firstY ;
	cout << "输入起始位置（棋盘的左上角编号位置（0，0））：" << endl;
	cin >> firstX >> firstY;
	visit(firstX, firstY, 1);
```

```
    cout << "Count " << countN << endl;
}
```

3.27 Ackerman 函数

（1）

```
int ack(int m,int n)
{
    if(m == 0) return n+1;
    else if(n == 0) return ack(m-1,1);
        else return ack(m-1,ack(m,n-1));
}
```

（3）

```
    akm(2,1)
    akm(1,akm(2,0))
    akm(1,akm(1,1))
    akm(1,akm(0,akm(1,0)))
    akm(1,akm(0,akm(0,1)))
    akm(1,akm(0,2))
    akm(1,3)
    akm(0,akm(1,2))
    akm(0,akm(0,akm(1,1)))    PS: akm(1,1)前面已经推过了
    akm(0,akm(0,3))
    akm(0,4)
    5
```

3.29 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)。

```
#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。

3.31 假设称正读和反读都相同的字符序列为“回文”，例如，'abba'和'abcba'是回文，'abcde'和 'ababab'则不是回文。试写一个算法判别读入的一个以'@'为结束符的字符序列是否是“回文”。

```
Status SymmetryString(char* p)
{
     Queue q;
     if(!InitQueue(q)) return 0;
     Stack s;
     InitStack(s);
     ElemType e1,e2;
     while(*p){
          Push(s,*p);
          EnQueue(q,*p);
          p++;
     }
     while(!StackEmpty(s)){
          Pop(s,e1);
          DeQueue(q,e2);
          if(e1!=e2) return FALSE;
     }
     return OK;
}
```

3.33 在顺序存储结构上实现输出受限的双端循环队列的入列和出列(只允许队头出列)算法。

Status EnQueue(Queue Q,ElemType e)

```
{
    head=Q.base[Q.front];// 头元素值
    tail=Q.base[(Q.rear-1+MAXQSIZE)%MAXQSIZE];// 尾元素值
    if((Q.rear+1)%MAXQSIZE==Q.front) return ERROR; //队列满
    if(!QueueEmpty(Q)||e>=(head+tail)/2){ //插在队尾
        Q.base[Q.rear] = e;
        Q.rear = (Q.rear+1)%MAXQSIZE;
    }
    else{
        Q.front = (Q.front-1+MAXQSIZE)%MAXQSIZE;
        Q.base[Q.front] = e;
    }
}
```

注意更改 Q.front 或 Q.rear 的时候要对 MAXQSIZE 取模。

第四章

4.23 假设以块链结构作串，试编写判别给定串是否具有对称性的算法。

int LString_Palindrome(LString L)//判断以块链结构存储的串 L 是否为回文序列，是则返回 1，否则返回 0

```
{
    InitStack(S);
    p=S.head;i=0;k=1;
    for(k=1;k<=S.curlen+1;k++)
    {
        if(k<=S.curlen/2)Push(S,p->ch[i]);
        else if(k>(S.curlen+1)/2){
            Pop(S,c);
            if(p->ch[i]!=c)return 0;
        }
        if(++i==CHUNKSIZE){
            p=p->next;
            i=0;
        }
    }
    return 1;
}
```

第五章

5.7③ 设有三对角矩阵$(a_{ij})_{n\times n}$，将其三条对角线上的元素逐行地存于数组 B[3n－2]中，使得 B[k]=a_{ij}，求：

(1) 用 i,j 表示 k 的下标变换公式；

(2) 用 k 表示 i,j 的下标变换公式。

(1)k = (i-1)×3-1+j-i+1=2i+j-3

(2)i = ⌊(k+1)/3⌋ + 1， j = k - 2i + 3 = k+1-2×⌊(k+1)/3⌋

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。

(1) p；　(2) (k,p,h)；　(3) (a,b)；　(4) $((c,d))$；

(5) (c,d)；(6) (b)；　(7) b；　(8) (d)。

5.32 试编写判别两个广义表是否相等的递归算法。

```
Status Equal(GList A, GList B)
/* 判断广义表A和B是否相等,是则返回TRUE,否则返回FALSE */
{
  if(A -> tag == ATOM && B -> tag == ATOM){
    //当都为原子节点ATOM时
        if(A -> un.atom == B -> un.atom)
            return TRUE;
        else
            return FALSE;
  }else if(A -> tag == LIST && B -> tag == LIST){
    //当都为广义表节点LIST时
        if(Equal(A -> un.ptr.hp,B -> un.ptr.hp) && Equal(A -> un.ptr.tp,B -> un.ptr.tp))
            //递归判断表头节点是否相等，表尾节点是否相等
            return TRUE;
        else
            return FALSE;
  }
}
```

5.34 试编写递归算法，逆转广义表中的数据元素。

```
void reverse_str(char *s, int l, int r)
{
    int n = r-l;
    for (int i = 0; i < n/2; ++i)
```

```
    swap(s[l+i], s[l+(n-1-i)]);
}

void reverse_list_recur(char *s, int l, int r)
{
    int i = l, last_i = l;
    int c1 = 0, c2 = 0;
    do {
        if (s[i] == '(') ++c1;
        else if (s[i] == ')') ++c2;
        if ((c1 == c2+1 && s[i] == ',') || (c1 == c2 && s[i] == ')')) {
            reverse_list_recur(s, last_i+1, i);
            reverse_str(s, last_i+1, i);
            last_i = i;
        }
        ++i;
    } while (c1 != c2);
    reverse_str(s, l+1, r-1);
}
```

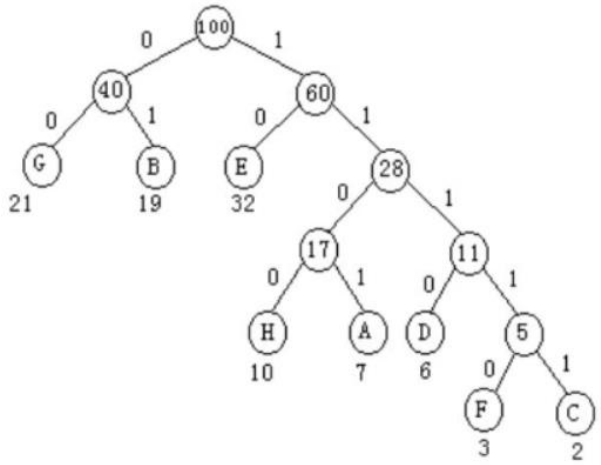
5.36 编写按上题描述的格式输出广义表的递归算法。

```
void GList_PrintList(GList A)
{
    if(!A) printf("()");
    else if(!A->tag) printf("%d",A->atom);
    else
    {
        printf("(");
        for(p=A;p;p=->ptr.tp)
        {
            GList_PrintList(p->ptr.hp);
            if(p->ptr.tp)printf(",");
        }
        printf(")");
    }
}
```

第六章

6.26 假设用于通信的电文仅由 8 个字母组成,字母在电文中出现的频率分别为 0.07, 0.19, 0.02, 0.06, 0.32, 0.03, 0.21, 0.10，试为这 8 个字母设计哈夫曼编码。使用 0~7 的二进制表示形式是另一种编码方案。对于上述实例，比较两种方案的优缺点。

6.26 解：不妨设这 8 个结点为 A、B、C、D、E、F、G、H，其相应的权为 7、19、2、6、32、3、21、10。



A:1101 B:01 C:11111 D:1110 E:10 F:11110 G:00 H:1100

采用这种方式编码，电文最短。

6.38 同 6.37 题条件，写出后序遍历的非递归算法(提示：为分辨后序遍历时两次进栈的不同返回点,需在指针进栈时同时将一个标志进栈)。

```
void PostOrder(BiTree T){
    InitStack(S);
    p=T;
    r=NULL;
    while(p!=NULL||!IsEmpty(s)){
        if(p! =NULL){               //走到最左边
            push(S,p);
            p=p->lchild;
        }
        else{               //向右
            GetTop(S,p);    //读栈顶节点（非出栈）
            if(p->rchild&&p->rchild!=r){    //若右子树存在，且未被访问过
                p=p->rchild;    //转向右
            }
            else{               //否则弹出结点并访问
                pop(S,p);
                visit(p->data); //访问该结点
                r=p;            //记录最近访问的结点
                p=NULL;         //结点访问完后，重置p指针
            }
        }
    }
```

6.41 编写递归算法，在二叉树中求位于先序序列中第 k 个位置的结点的值。

```
typedef struct BiTNode{
    char data;
```

```
    struct BiTNode *Lchild, *Rchild;
}BiTNode,*BiTree;

bool Find_K(BiTree T,int* i, int k){
    if(T == NULL)
        return false;
    (*i)++;
    if(k == *i){
        printf("%c\n", T->data);
        return true;
    }
    if(Find_K(T->Lchild,i,k) || Find_K(T->Rchild,i,k))
        return true;
    return false;
}
```

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。

```
void ReChange(BiTree root)
{
    if(root==NULL) return;
    else
    {
        BiTree temp=root->lchild;
        root->lchild=root->rchild;
        root->rchild=temp;
        ReChange(root->lchild);
        ReChange(root->rchild);
    }
}
```

6.48 已知在二叉树中，* root 为根结点，*p 和*q 为二叉树中两个结点，试编写求距离它们最近的共同祖先的算法。

1.当这两个节点在某个节点的左右子树中时，这个节点为这两个节点的最近共同祖先。
2.其中一个节点为另一个节点的祖先的情况，那么这个节点的双亲就是他俩的最近共同祖先。

```
void traverse(BSTNode* root, BSTNode* p, BSTNode* q, int &k) //当前树中有 p,q 中的几个结点（0/1/2）
{
    if(root == NULL)
        return;
    else
    {
        if(root == p || root == q)
```

```
            k++;
        traverse(root->left, p, q, k);
        traverse(root->right, p, q, k);
    }
}

BSTNode* Find(BSTNode* root, BSTNode* p, BSTNode* q)
{
    int left = 0, right = 0;
    traverse(root->left, p, q, left);//找到左子树中节点个数
    traverse(root->right, p, q, right);//找到右子树中节点个数
    if(root->ch == p || root->ch == q)//如果当前结点为其中一个
        return root;//返回当前结点
    else if(left == right)//如果当前结点左右子树各有一个节点，返回当前结点
        return root;
    else   //如果两个节点都在左子树中，那么在左子树中找，反之，在右子树中找
        if(left == 2)
            find(root->left, p, q);
        else
            find(root->right, p, q);
    }
}
```

6.53 试编写算法，求给定二叉树上从根节点到叶子节点的一条其路径长度等于树的深度减一的路径，若这样的路径存在多条，则输出路径终点（叶子结点）在“最左”的一条。

```
Status MaxPathBiTree(BiTree& T)
{
    if(T){
        if(BiTDepth(T)-BiTDepth(T->lchild)!=1)
            DelBiTree(T->lchild);
        else
            MaxPathBiTree(T->lchild);
        if(BiTDepth(T)-BiTDepth(T->rchild)!=1)
            DelBiTree(T->rchild);
        else
            MaxPathBiTree(T->rchild);
    }
    return OK;
}

//从根到叶子最长路径中最左方的路径树
Status LMaxPathBiTree(BiTree& T)
{
    if(T){
```

```
        if(BiTDepth(T)-BiTDepth(T->lchild)==1){
            DelBiTree(T->rchild);
            LMaxPathBiTree(T->lchild);
        }
        else{
            DelBiTree(T->lchild);
            if(BiTDepth(T)-BiTDepth(T->rchild)==1)
                LMaxPathBiTree(T->rchild);
            else
                DelBiTree(T->rchild);
        }
    }
    return OK;
}
```

6.56 试写一个算法，在先序后继线索二叉树中，查找给定结点*p 在先序序列中的后继(假设二叉树的根结点未知)。并讨论实现此算法对存储结构有何要求?

```
typedef struct BiTNode{
    char data;
    int ltag,rtag;
    struct BiTNode *Lchild, *Rchild;
}BiTNode,*BiTree;

BiTNode PreNext(BiTNode p){
    if(p->Ltag==0)
        p=p->Lchild;
    else
        p=p->Rchild;
    return p;
}
```

二叉链表

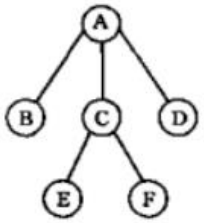
6.65 已知一棵二叉树的前序序列和中序序列分别存于两个一维数组中，试编写算法建立该二叉树的二叉链表。

```
BiTree Resume_BiTree(TElemType *pre,TElemType *mid,int prelen,int midlen)
//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;
}
```

6.67④ 假设以二元组(F,C)的形式输入一棵树的诸边(其中F表示双亲结点的标识,C表示孩子结点标识),且在输入的二元组序列中,C是按层次顺序出现的。F='^'时C为根结点标识,若C也为'^',则表示输入结束。例如,如下所示树的输入序列为:

```
^ A
AB
AC
AD
CE
CF
^ ^
```



试编写算法,由输入的二元组序列建立树的孩子-兄弟链表。

```
#define maxSize 50
Status CreateCSTreeByDuplet(CSTree *pT)
{
    char input[5];
    CSNode *queue[maxSize];int front,rear;
    CSNode *p, *q;
    front=rear=0; //对队列初始化
    for (scanf("%s", input); input[1]!='^'; scanf("%s", input)) {//创建结点
        p = (CSNode *)malloc(sizeof(CSNode)); if (!p) exit(OVERFLOW);
        p->data=input[1];p->firstchild=p->nextsibling=NULL; //入队列
        queue[rear]=p;rear=(rear+1)%maxSize; //找爸爸
        if (input[0]=='^')   //根结点-->不需要找爸爸
            *pT = p; //传出去
        else {
            for (q=queue[front]; q->data!=input[0]; front=(front+1)%maxSize,q=queue[front]) ;
//找爸爸
            //找哥哥
```

```
            if (!q->firstchild) q->firstchild=p; //它是最大的
            else { //它不是最大的
                for(q=q->firstchild; q->nextsibling; q=q->nextsibling) ; //找最近的哥哥
                q->nextsibling = p; //和哥哥牵手
            }
        }
    }
    return OK;
}
```

6.74 试写一递归算法，以 6.73 给定的树的广义表表示法的字符序列形式输出以孩子-兄弟链表表示的树。

```
void PrintGlist_CSTree(CSTree T)
{
    printf("%c",T->data);
    if(T->firstchild)
    {
        printf("(");
        for(p=T->firstchild;p;p=p->nextsib)
        {
            PrintGist_CSTree(p);
            if(p->nextsib)printf(",");
        }
        printf(")");
    }
}
```

6.75 试写一递归算法，由 6.73 题定义的广义表表示法的字符序列，构造树的孩子链表。

```
Status CreateCTreeByGList(CTree *pT, int parent)
{
    char c;
    CNode *p, *q;
    int newNode;
    newNode = pT->n;
    for (c=getchar(); c!='\n'; c=getchar() )
    {
        if (c>='A' && c<='Z')
        {
            pT->nodes[newNode].data = c;
            pT->nodes[newNode].firstchild = NULL;
            pT->n++;
            if (parent!=-1) {
                p = (CNode *)malloc(sizeof(CNode));
                p->index = newNode;
```

```
                p->next = NULL;
                if (pT->nodes[parent].firstchild==NULL) {
                    pT->nodes[parent].firstchild = p;
                }
                else {
                    for (q=pT->nodes[parent].firstchild; q->next; q=q->next) ;
                    q->next = p;
                }
            }
        }
        else if (c=='(')
            CreateCTreeByGList(pT, newNode);
        else if (c==',') {
            CreateCTreeByGList(pT, parent);
            return OK;
        }
        else if (c==')')
            return OK;
    }
    return OK;
}
```

第七章

7.15 试在邻接矩阵存储结构上实现图的基本操作：InsertVex(G,v), InsertAre(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
```

7.19 编写算法，由依次输入的顶点数目、边的数目、各顶点的信息和各条边的信息建立无向图的邻接多重表。

```
#include <stdio.h>
#include <stdlib.h>
#define maxsize 100

typedef int VexType;
typedef struct ArcNode
{
    struct ArcNode *nextarc;
    int adjvex;   //顶点编号
}ArcNode;

typedef struct
{
    ArcNode *firstarc;
    VexType data;     //该边指向的结点的位置
}VNode;

typedef struct
{
    VNode AdjList[maxsize];
    int vexnum, arcnum;
}AGraph;

VexType locate(AGraph *G, VexType x)
{
    for (int i=0; i<G->vexnum; i++)
        if (G->AdjList[i].data == x)
            return i;

    return -1;
}
AGraph *creat()
{
    AGraph *G;
    printf("请输入顶点数目：");
    scanf("%d", &(G->vexnum));
    printf("请输入弧的数目：");
    scanf("%d", &(G->arcnum));

    int i,k;
    VexType vex;
    VexType v1, v2;
```

```
    for (i = 0; i < G->vexnum; i++)
    {
        printf("正在创建顶点表，请输入顶点信息：\n");
        scanf("%d", &vex);
        G->AdjList[i].data = vex;
        G->AdjList[i].firstarc = NULL;
    }

    for (k = 0; k < G->arcnum; k++)
    {
        printf("正在连接各个顶点，请输入弧的信息：\n");
        scanf("%d%d", &v1, &v2);   //v1 为弧尾，v2 为弧头；
        int a = locate(G, v1);          //求顶点 v1 在顶点表中的编号
        int b = locate(G, v2);          //求顶点 v2 在顶点表中的编号

        //采用头插法建表
        ArcNode *p = (ArcNode*)malloc(sizeof(ArcNode));
        p->adjvex = b;
        p->nextarc = G->AdjList[a].firstarc;
        G->AdjList[a].firstarc = p;
    }
    return G;
}

int visit[maxsize];
void dfs(AGraph *G, int v0)
{//采用深度优先遍历的方法对图进行打印，该图存储在邻接表中
    ArcNode *p;
    visit[v0] = 1;
    printf("检查待输入数组是否被标记为已访问：%d\n", visit[v0]);
    printf("%d\n", G->AdjList[v0].data);
    p = G->AdjList[v0].firstarc;
    while(p != NULL)
    {
        if(visit[p->adjvex] == 0)
            dfs(G, p->adjvex);
        p = p->nextarc;
    }
}

void print(AGraph *G)
{//为避免要打印的图为非连通图，将深度优先遍历嵌套在 for 循环中
    for (int i=0; i<G->vexnum; i++)
        visit[i] = 0;        //初始化 visit 数组
```

```
    printf("\n");
    for (int i=0; i<G->vexnum; i++)
        if(visit[i] == 0)
            dfs(G, i);
}

void main()
{
    AGraph *G = creat();
    print(G);
}
```

7.23 同 7.22 题要求，试基于图的广度优先搜索策略写一算法。

```
Status Path(AGraph* G,int vi, int vj){
    int queue[MaxSize];
    int front = 0,rear = 0;
    ArcNode* node;
    rear = (rear + 1)%MaxSize;
    queue[rear] = vi;
    while (front != rear) {
        front = (front + 1)%MaxSize;
        int num = queue[front];
        node = G->adjList[num].firstArc;
        while (node) {
            rear = (rear + 1)%MaxSize;
            queue[rear] = node->adjNum;
            if (visit[node->adjNum] == 0) {
                visit[node->adjNum] = 1;
            }
            node = node->next;
        }
    }
    if (visit[vj] == 1)
        return 1;
    else
        return 0;
}
```

7.24 试利用栈的基本操作编写，按深度优先搜索策略遍历一个强连通图的非递归形式的算法。算法中不规定的具体的存储结构，而将图 Graph 看成是一种抽象的数据类型，

```
void STraverse_Nonrecursive(Graph G)
{
    int visited[MAXSIZE];
    InitStack(S);
```

```
    Push(S,GetVex(S,1));
    visit(1);
    visited=1;
    while(!StackEmpty(S))
    {
        while(Gettop(S,i)&&i)
        {
            j=FirstADjVex(G,i);
            if(j&&!visited[j])
            {
                visit(j);
                visited[j]=1;
                Push(S,j);
            }
        }
        if(!StackEmpty(S))
        {
            Pop(S,j);
            Gettop(S,i);
            k=NextAdjVex(G,i,j);
            if(k&&!visited[k])
            {
                visit(k);
                visited[k]=1;
                Push(S,k);
            }
        }
    }
}
```

7.26 试证明，对有向图中顶点适当地编号，可使其邻接矩阵为下三角形且主对角线为全零的充要条件是：该有向图不含回路。然后写一算法对无环有向图的顶点重新编号，使其邻接矩阵变为下三角形，并输出新旧编号对照表。

证明：该有向图顶点编号的规律是让弧尾顶点的编号大于弧头顶点的编号。由于不允许从某顶点发出并回到自身顶点的弧，所以邻接矩阵主对角元素均为 0。先证明该命题的充分条件。由于弧尾顶点的编号均大于弧头顶点的编号，在邻接矩阵中，非零元素（A[i][j]=1）自然是落到下三角矩阵中；命题的必要条件是要使上三角为 0，则不允许出现弧头顶点编号大于弧尾顶点编号的弧，否则，就必然存在环路。（对该类有向无环图顶点编号，应按顶点出度顺序编号。）

```
Status Change_into_LTM(MGraph &G){
    //将采用邻接矩阵存储的有向无环图的邻接矩阵转换为下三角矩阵，并输出新旧编号对照表
    int topologic_order[G.vexnum];//拓扑序列，反过来就是逆拓扑序列
    //topologic_order[G.vexnum - 1 -  新序号] =  旧序号
```

```
    int num = 0;//拓扑序列指针
    FindInDegree(G,indegree);//求入度序列
    InitStack(S);
    for(i = 0;i < G.vexnum;i++)
        if(!indegree[i]) Push(S,i);
    while(!StackEmpty(S)){
        Pop(S,i); topologic_order[num++] = G.vertices[i].data;
        for(j = 0;j < G.vexnum;j++)
            if(G.arcs[i][j] && !(--indegree[j])) Push(S,j);
    }//while
    int new_old[G.vexnum];//新旧参照表
    for(i = 0;i < G.vexnum;i++) new_old[topologic_order[G.vexnum - 1 - i]] = i;
    //new_old[旧序号] = 新序号
    int keep[G.vexnum][G.vexnum];//缓存数组
    for(i = 0;i < G.vexnum;i++)
        for(j = 0;j < G.vexnum;j++)
            keep[i][j] = G.arcs[i][j];
    for(i = 0;i < G.vexnum;i++)
        G.vertices[i].data = new_old[G.vertices[i].data];
    for(i = 0;i < G.vexnum;i++){
        tmp = topologic_order[G.vexnum - 1 - i];
        for(j = 0;j < G.vexnum;j++){
            if(keep[tmp][j])
                G.arcs[i][new_old[j]] = 1;
            else
                G.arcs[i][new_old[j]] = 0;
        }//for
    }//for
    for(i = 0;i < G.vexnum;i++)
        cout << i << "--->" << new_old[i] << endl;
    return OK;
}//Change_into_LTM
```

7.27 采用邻接表存储结构，编写一个判别无向图中任意给定的两个顶点之间是否存在一条长度为 k 的简单路径的算法。

```
int visited[MAXSIZE];
int exist_path_k(ALGraph G,int i,int j,int k){
    if(i==j&&k==0)
        return 1;
    else if(k>0) {
        visited[i]=1;
        for(p=G.vertices[i].firstarc;p!=NULL;p=p->nextarc) {
            m=p->adjvex;
            if(!visited[m])
```

```
                if(exist_path_len(G,m,j,k-1))
                return 1;
            }
            visited[i]=0;
        }
        return 0; //没找到
}
```

7.29 试写一个算法，在以邻接矩阵方式存储的有向图 G 中求顶点 i 到顶点 j 的不含回路的、长度为 k 的路径数。

```
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
```

7.30 试写一个求有向图 G 中所有简单回路的算法。

思路：
1. DFS 搜索，直到搜索到已经遍历到的结点，说明找到了回路
2. 判断该回路是否已经搜索到了，若不重复，放入结果集

```
VertexType cycles[maxSize][MAX_VERTEX_NUM+1]; //存放所有回路
int path[MAX_VERTEX_NUM+1]; //路径，前面定义过
int visit[MAX_VERTEX_NUM]; //访问标记，前面定义过
int pathnum=0; //已发现的路径个数，前面已经定义过
Status ExistCycle(ALGraph G, int start, int end) { // [start,end)
    int i,j,k,e;
    int len;
    int flag=0;
    len = end-start;
    for (i=0; i<pathnum; i++) {
        if (strlen(cycles[i])==len) { //长度一样
            //[start,end) ?= cycles[i]-->判断两个回路是否相同
```

```
                flag=0; //找到了 0 个一样的
                for (j=start; j<end; j++) {
                    e = path[j];
                    //在 cycles[i]中找 e
                    for (k=0; cycles[i][k]!='\0'; k++) {
                        if (cycles[i][k]==G.vers[e].data) flag++; //找到了
                    }
                }
                if (flag==len) return TRUE; //找到了 len 一样的元素-->完全相同
            }
        }
        return FALSE; //不存在
}
void FindAllCycle(ALGraph G, int v, int k) {
        ArcNode *p;
        int i,j;
        int start,nextadj;
        visit[v]=1;
        path[k]=v;
        //  从 v 的邻边开始走
        for (p=G.vers[v].firstarc; p; p=p->next) {
            nextadj = p->adjV; //下一个结点
            if (visit[nextadj]) { //已经访问过了-->找到了回路
                //找到这条回路的起始点
                for (i=0; i<k; i++) {
                    if (path[i]==nextadj) {
                        start=i;
                    }
                }
                if (!ExistCycle(G, start, k+1)) { //这个回路没有重复
                    for (i=start, j=0; i<=k; i++,j++) {
                        cycles[pathnum][j] = G.vers[ path[i] ].data;
                    }
                    cycles[pathnum][j]='\0';
                    pathnum++;
                }
            } else { //没有访问过，继续访问
                FindAllCycle(G, nextadj, k+1);
            }
        }
        //  回溯
        visit[v]=0;
        path[k]=0;
}
```

```
void GetAllCycle(ALGraph G) {
    int i;
    for (i=0; i<G.vernum; i++) visit[i]=0; //访问标记初始化
    pathnum=0; //路径个数初始化
    for (i=0; i<G.vernum; i++) {
        if (visit[i]==0) FindAllCycle(G, i, 0);
    }
}
```

7.34 试编写一个算法，给有向无环图 G 中每个顶点赋以一个整数序号，并满足以下条件：若从顶点 i 至顶点 j 有一条弧，则应使 i<j.

```
Status TopoSeq(ALGraph G,int new[])
{
    int indegree[MAXSIZE];
    FindIndegree(G,indegree);
    Initstack(S);
    for(i=0;i<G.vexnum;i++)
        if(!indegree[i])
            Push(S,i);
    count=0;
    while(!stackempty(S))
    {
        Pop(S,i);
        new[i]=++count;
        for(p=G.vertices[i].firstarc;p;p=p->nextarc)
        {
            k=p->adjvex;
            if(!(--indegree[k]))
                Push(S,k);
        }
    }
    if(count<G.vexnum)
        return ERROR;
    return OK;
}
```

7.37 试设计一个求有向无环图中最长路径的算法，并估计其时间复杂度。

```
int *indegree;//保存顶点入度
int *topo;//保存拓扑排序
int *maxpath;//保存此点的最长路径
//获得各顶点入度
FindIndegree(G,indegree)   //拓扑排序
void topologicalSort(MGraph G){
    int i,j,k;
```

```
    for(i=0;i<G.vexnum;i++) {
        j=0;
        while(j<G.vexnum&&indegree[j]!=0)
            j++;
        topo[i]=j;
        indegree[j]=-1;
        for(k=0;k<G.vexnum;k++)
            if(G.arcs[j][k]!=0)
                indegree[k]--;
    }
}
//求最长路径
void getmaxpath(MGraph G){
int max,*maxpath,*prepath,maxpathend;
max=0;//最长路径长度
maxpath=new int[G.vexnum];//点的最长路径
prepath=new int[G.vexnum];//点的最长路径的前一个点
maxpathend=-1;//最长路径的终点
    for(int i=0;i<G.vexnum;i++){
        maxpath[i]=0;
        prepath[i]=-1;
    }
    for(int i=0;i<G.vexnum;i++){
        int v2=topo[i];
        for(int k=0;k<i;k++){
            int v1=topo[k];
            if(G.arcs[v1][v2]!=0){
                if(maxpath[v1]+G.arcs[v1][v2]>maxpath[v2]){
                    maxpath[v2]=maxpath[v1]+G.arcs[v1][v2];
                    prepath[v2]=v1
                    if(maxpath[v2]>max){
                        max=maxpath[v2];
                        maxpathend=v2;
                    }
                }
            }
        }
    }
    while(prepath[maxpathend]!=-1){
    printf("%d\n",maxpathend );
    maxpathend = prepath[maxpathend]
    }
}
```

时间复杂度为 $O(G.vexnum^2)$