线性表

1. 顺序表
    1. 定义

```
#define MAXSIZE 20
typedef struct{
    ElemType data[MAXSIZE];
    int length;
}SqList;

typedef struct{
    ElemType *data;
    int length;
}
```

    1. 查找

```
int LocateElem(SqList L,ElemType key){
    int i;
    for(i = 0;i < n;i ++){
        if(L.data[i] == key)
            return i+1;
    }
    return 0;
}
```

    2. 插入

```
bool InsertElem(SqList L,int i,ElemType e){
    if(i < 0 || i>L.length)
        return false;
    if(L.length == MAXSIZE)
        return false;
    else{
        for(j = L.length-1;j >= i;j --)
            L.data[j+1] = L.data[j];
        L.data[i-1] = e;
        L.length++;
        return ture;
    }
}
```

    3. 删除

```
bool DeleteElem(SqList L,int i,ElemType &e){
    if(i < 0 || i>L.length)
        return false;
    else{
        e = L.data[i-1];
        for(j = i;j < n;j ++){
            L.data[j-1] = L.data[j];
        }
        L.length--;
        return ture;
    }
}
```

4. 将顺序表中所有的元素逆置

```
void InverseList(SqList &L){
    int low,high;
    low = 0;
    high = L.length-1;
    while(low < high){
        swap(L.data[low],L.data[high]);
        low++;
        high--;
    }
}
```

5. 删除所有值为x的数

```
void DeleteXElem(SqList &L, ElemType x) {
    int low, high;
    int n = L.length;
    low = 0;
    high = n-1;
    while (low < high) {
        while (L.data[high] == x && high >= 0) {
                high--;
                L.length--;
        }
        while (L.data[low] != x && low < n) low++;
        if (low < high) {
        swap(L.data[low], L.data[high]);
        }
    }
}
```

6. 有序顺序表中删除给定区间的所有元素

```
bool DeleteNum(SqList &L, ElemType s, ElemType t) {
   if (s > t)
       return false;
   if (isEmpty(L) || s>L.data[L.length -1] || t < L.data[0])
       return false;
   else {
       int i = 0, j = L.length - 1;
       while (L.data[i] < s)
           i++;
       while (L.data[j] > t)
           j--;
       int k;
       for (k = j + 1; k < L.length; k++) {
           L.data[k - (j - i + 1)] = L.data[k];
       }
       L.length = L.length - (j - i + 1);
       return true;
   }
```

7. 从有序顺序表中删除所有重复的元素

```
void Unique(SqList &L) {
    int i, j;
    ElemType s,e;
    for (i = 0; i < L.length; i++) {
        s = L.data[i];
        for (j = i + 1; j < L.length; j++) {
            if (L.data[j] == s)
                DeleteSqList(L, j, e);
        }
    }
}
```

8. 由两个有序顺序表合成一个新的有序顺序表

```
SqList MergeList(SqList &L1, SqList &L2) {
    int i = 0, j = 0,k=0;
    SqList L;
    L.length = L1.length + L2.length;
    while (i<L1.length && j<L2.length)
    {
        if (L1.data[i] < L2.data[j]) {
            L.data[k] = L1.data[i];
            i++;
        }
        else {
            L.data[k] = L2.data[j];
            j++;
        }
```

```
        k++;
    }
    if (i < L1.length)
        while (k < L.length) {
            L.data[k++] = L1.data[i++];
        }
    else
    {
        while (k < L.length)
        {
            L.data[k++] = L2.data[j++];
        }
    }
    return L;
}
```

9. 交换一个顺序表中两个线性表的位置

```
bool ExchangeList(SqList &L,int m,int n) {
    if (m > L.length || n > L.length)
        return false;
    if (L.length == m || L.length == n)
        return true;
    else
    {
        int i, j;
        int low = 0, high = m - 1;
        while (low<high)                                    //前m个元素
逆置
        {
            swap(L.data[low], L.data[high]);
            low++;
            high--;
        }
        low = m;
        high = L.length - 1;
        while (low < high)                                  //后n个元素
逆置
        {
            swap(L.data[low], L.data[high]);
            low++;
            high--;
        }
        i = 0;
        j = L.length - 1;
        while (i<j)
        {
            swap(L.data[i], L.data[j]);                     //所有元素
逆置
            i++;
            j--;
        }
```

```
            return true;
        }
    }
```

10. 将一维数组中所有元素左移P个位置

```
void ROLList(SqList &L, int p) {
    if (p<0 || p>L.length)
        return;
    int m = p;
    int n = L.length - p;
    ExchangeList(L, m, n);
}
```

11. 找到两个有序表中的中位数

```
ElemType FindMid(SqList &L1, SqList &L2) {
    int len = L1.length + L2.length;
    int i = 0, j = 0, k = 0;
    while (k<len/2)
    {
        if (L1.data[i] < L2.data[j]) {
            k++;
            if (k == len / 2)
                return L1.data[i];
            i++;
        }
        else {
            k++;
            if (k == len / 2)
                return L2.data[j];
            j++;
        }
    }
}
```

12. 找到主元素

```
int FindMainElem(SqList L) {
    int i, j;
    int e;
    for (i = 0; i < L.length; i++) {
        int count = 1;
        e = L.data[i];
        for (j = i + 1; j < L.length; j++) {
            if (L.data[j] == e)
                count++;
```

```
        }
        if (count > L.length / 2)
            return e;
    }
    return -1;
}
```

13. 找到未出现的最小正整数

```
int FindMinNum(SqList L) {
    int i;
    bool *Flag = new bool[1001];
    for (i = 0; i < 1001; i++) {
        Flag[i] = 0;
    }
    for (i = 0; i < L.length; i++) {
        if (L.data[i] > 0)
            Flag[L.data[i]] = 1;
    }
    i = 1;
    while (Flag[i]!=0)
    {
        i++;
    }
    return i;
    delete[]Flag;
}
```

14. 找到所有三元组的最小距离

```
int* FindMinDst(SqList L1, SqList L2, SqList L3) {
    int i, j, k;
    int w=L1.data[0], y=L2.data[0], z=L3.data[0];
    int dst = 10000;
    int *s = new int[4];
    s[0] = dst; s[1] = w; s[2] = y; s[3] = z;
    for (i = 0; i < L1.length; i++) {
        for (j = 0; j < L2.length; j++) {
            for (k = 0; k < L3.length; k++) {
                int ldst = abs(L1.data[i] - L2.data[j]) + abs(L1.data[i] -
L3.data[k]) + abs(L2.data[j] - L3.data[k]);
                if (ldst < dst) {
                    dst = ldst;
                    w = L1.data[i];
                    y = L2.data[j];
                    z = L3.data[k];
                    s[0] = dst; s[1] = w; s[2] = y; s[3] = z;
                }
            }
```

```
        }
    }
    return s;
}
```

2. 链表

    1. 存储结构

```
typedef struct LNode{
    ElemType data;
    struct LNode *next;
}LNode,*LinkList;
```

    2. 原地反转链表

```
bool ReverseList(LinkList &L) {
    if (isEmpty(L)) {
        return FALSE;
    }
    int n = LinkLength(L);
    LNode *pre = L->next;
    int i;
    for (i = 0;i < n-1;i++)
    {
        p = pre->next;
        pre->next = p->next;
        p->next = L->next;
        L->next = p;
    }
}
```

    3. 使单链表的元素递增有序

```
void AscendingList(LinkList &L) {
    LNode *p, *q = L,*pre;
    int i, j;
    int len = LinkLength(L);
    for (i = 1; i < len; i++) {
        p = L;
        q = L->next;
        pre = L;
        for (j = 0; j < i; j++) {
            pre = pre->next;
            q = q->next;
        }
        while (q->data > p->next->data)
        {
```

```
            p = p->next;
        }
        pre->next = q->next;
        q->next = p->next;
        p->next = q;
    }
}
```

4. 删除有序单链表中在给定区间内的元素

```
void DeleteNode(LinkList &L,int s,int t) {
    LNode *p = L, *q = L;
    int n = LinkLength(L);
    while (p->next->data<s)
    {
        p = p->next;
    }
    while (q->next->data<=t)
    {
        q = q->next;
    }
    while (p->next != q) {
        LNode *r = p->next;
        p->next = r->next;
        free(r);
    }
    p->next = q->next;
    free(q);
}
```

5. 找到两个单链表的公共结点

```
void FindcComNode(LinkList &L1, LinkList &L2, LinkList &L) {
    LNode *p=L1, *q;
    L = new LNode;
    L->next = NULL;
    while(p->next) {
        p = p->next;
        q = L2->next;
        while (q && q->data!=p->data)
        {
            q = q->next;
        }
        if (q) {
            LNode *r = new LNode;
            r->data = p->data;
            r->next = L->next;
            L->next = r;
        }
```

```
    }
}
```

6. 输出单链表的增序，并释放结点

```
void AsdOutPut(LinkList &L) {
    LNode *p = L,*q;
    int min;
    cout << "该单链表的有序输出为: ";
    while (p->next)
    {
        min = p->next->data;
        q = p->next;
        LNode *r = p;
        while (q->next)
        {
            if (q->next->data < min) {
                min = q->next->data;
                r = q;
            }
            q = q->next;
        }
        cout << min<<"  ";
        LNode *w = r->next;
        r->next= w->next;
        free(w);
    }
}
```

7. 分解单链表为两个带头结点的单链表

```
void DivideTwoList(LinkList &L, LinkList &L1, LinkList&L2) {
    LNode*p=L, *q=L->next;
    L1 = new LNode;
    L1->next = NULL;
    L2 = new LNode;
    L2->next = NULL;
    int i;
    int n = LinkLength(L);
    for (i = 1; i <= n; i++) {
        if (i % 2 == 0 && p->next->next) {
            p = p->next->next;
            LinkInsert(L1, i / 2 , p->data);
        }
        if (i % 2 != 0 && q) {
            LinkInsert(L2, i / 2 + 1, q->data);
            if(q->next)
                q = q->next->next;
        }
```

```
    }
}
```

8. 删除单链表中重复的元素

```
void Unique(LinkList &L) {
    LNode *p = L, *q;
    int i,j=0;
    while (p->next)
    {
        j++;
        p = p->next;
        int temp = p->data;
        q = p->next;
        i = j;
        while (q)
        {
            i++;
            if (q->data == temp) {
                q = q->next;
                DeleteList(L, i);
                i--;
            }
            else
                q = q->next;
        }
    }
}
```

9. 将两个增序单链表合并成一个降序单链表

```
void AsdToDsd(LinkList &L1, LinkList &L2, LinkList &L) {
    L = new LNode;
    L->next = NULL;
    LNode *p = L1->next, *q = L2->next;
    while (p && q)
    {
        if (p->data < q->data) {
            LNode *r = p;
            p = p->next;
            r->next = L->next;
            L->next = r;
        }
        else
        {
            LNode *r = q;
            q = q->next;
            r->next = L->next;
            L->next = r;
```

```
        }
    }
        while (p)
        {
            LNode *r = p;
            p = p->next;
            r->next = L->next;
            L->next = r;
        }
        while (q)
        {
            LNode *r = q;
            q = q->next;
            r->next = L->next;
            L->next = r;
        }
}
```

树的有关算法

1. 顺序存储结构
2. 链式存储结构

```
typedef struct BiTnode{
    ElemType data;
    struct BiTnode *lchid,*rchild;
}BiTnode,*BiTree;
```

n个结点的二叉链表中，空链域的个数是n+1

3. 二叉树的遍历
    1. 先序遍历
        1. 递归

```
void PreOrder(BiTree T){
   if(T!=NULL){
       visit(T);
       PreOrder(T->lchild);
       PreOrder(T->rchild);
   }
}
```

        2. 非递归

```
    void PreOrder2(BiTree T){
        Stack S;
        InitStack(S);
        BiTnode p = T;
        while(p||!IsEmpty(S)){
            if(p){
                visit(T);
                Push(S,p);
                p->lchild;
            }
            else{
                Pop(S,p);
                p->rchild;
            }
        }

    }
```

2. 中序遍历

    1. 递归

```
void InOrder(BiTree T){
    if(T!=NULL){
        InOrder(BiTree T->lchild);
        visit(T);
        InOrder(BiTree T->rchlid);
    }
}
```

    2. 非递归

```
    void InOrder2(BiTree T){
        Stack S;
        InitStack(S);
        BiTnode p = T;
        while(p||!IsEmpty(S)){
            if(p){
                Push(S,p);
                p->lchild;
            }
            else{
                Pop(S,p);
                visit(p);
                p->rchild;
            }
        }
    }
```

3. 后续遍历

1. 递归

```
void PostOrder(BiTree T){
    PostOrder(BiTree T->lchid);
    PostOrder(BiTree T->rchild);
    visit(T);
}
```

2. 非递归

```
void PostOrder2(BiTree T){
    Stack S;
    InitStack(S);
    BiTnode *p = T;
    BiTnode *r = NULL;
    while(p||!IsEmpty(S)){
        if(p){
            Push(S,p);
            p = p->lchild;
        }
        else{
            GetTop(S,p);
            if(p->rchild&&p->child!=r){
                p=p->rchild;
            }
            else{
                Pop(S,p);
                visit(p);
                r = p;//recently visited
                p = NULL;
            }
        }
    }
}
```

4. 层次遍历

```
void LevelOrder(BiTree T){
    Queue Q;
    InitQueue(Q);
    BiTnode *p = T;
    EnQueue(Q,p);
    while(!IsEmpty(Q)){
        DeleteQueue(Q,p);
        visit(Q);
```

```
        if(p->lchild)
            EnQueue(Q,p->lchild);
        if(p->rchild)
            EnQueue(Q,p->rchild);
    }
}
```

5. 统计二叉树中度为0的结点个数

```
int NumsDegree_0(BiTree T){
    if(T){
        if(T->lchild == NULL && T->rchild == NULL){
            return 1;
        }
        else{
             return NumsDegree_0(T->lchild)+NumDegree_0(T->rchild);
         }
    }
    else
         return 0;
}
```

6. 统计二叉树中度为1的结点个数

```
int NumDegree_1(BiTree T){
    if(T){
        if((T->lchild && T->rchild == NULL)||(T->lchild==NULL && T-
>rchild))
         return NumDegree_1(T->lchild)+NumDegree_1(T->rchild)+1;
         else
         return NumDegree_1(T->lchild)+NumDegree_1(T->rchild);
    }
    else
         return 0;
}
```

7. 统计二叉树中度为2的结点个数

```
int NumDegree_2(BiTree T){
    if(T){
        if(T->lchild && T->rchild)
            return NumDegree_2(T->lchild)+NumDegree_2(T->rchild)+1;
        else
             return NumDegree_2(T->lchild)+NumDegree_2(T->rchild);
    }
    else
```

```
        return 0;
}
```

8. 统计二叉树的高度

```
int GetHeight(BiTree T){
    if(T=NULL)
        return 0;
     else
        int LHeight = GetHeight(T->lchild);
        int RHeight = GetHeight(T->rchild);
        return LHeight>RHeight ? (LHeight+1) : (RHeight+1);
}
```

9. 统计二叉树的宽度

```
int LevelWidth(BiTree T,int level){
    if(T == NULL);
        return 0;
     else{
        if(level==1)
            return 1;
        else
            level = LevelWidth(T->lchild,level-1)+LevelWidth(T-
>rchild,level-1);
     }
     return level;
}

int GetWidth(BiTree T){
    int width,i;
    int w[MAXLEVEL];
    for(i = 0;i < 20;i ++){
        w[i] = 0;
    }
    if(T!=NULL)
        width = 0;
    else{
        for(i = 0;i < GetHeight(T);i ++){
            w[i] = LevelWidth(T,i+1);
        }
    }
    i = 0;
    while(w[i]){
        if(w[i]>width)
            width = w[i];
            i++;
    }
```

```
    return width;
}
```

10. 计算给定结点*p所在的层次

```
int GetLevel(BiTree T,BiTnode *p){
    if(T == NULL){
        return 0;
    }
    if(T == p){
        return 1;
    }
    int dep1 = GetLevel(T->lchild,p);
    int dep2 = GetLevel(T->rchild,p);
    if(dep1 || dep2){
        if(dep1>dep2)
            return dep1+1;
        else
            return dep2+1
    }
}
```

11. 从二叉树中删除所有的叶节点

```
void DeleteLeaf(BiTree T){
    if(T!=NULL){
        if(T->lchild = NULL && T->rchild == NULL){
            free(T);
        }
        else{
            DeleteLeaf(T->lchild);
            DeleteLeaf(T->rchild);
        }
    }
}
```

12. 计算二叉树各结点中最大元素的值

```
int GetMax(BiTree T){
    if(T == NULL)
        return -10000;
    else{
        if(T->lchild == NULL && T->rchild == NULL)
            return T->data;
        else
            return MAX(GetMax(T->lchild),T->data,GetMAX(T->rchild));
```

```
    }
}
```

13. 交换二叉树中每个结点的两个子女

```
void ExchangeChild(BiTree T){
    if(T->child)
        return ExchangeChild(T->child);
    else if(T->rchild)
        return ExchangeChild(T->rchlid);
    else
        return;
    BiTnode *p;
    p = T->lchild;
    T->lchild = T->rchild;
    T->rchild = p;
}
```

14. 以先序次序输出一颗二叉树中所有结点的数据值以及结点所在的层次

```
void PAL(BiTree T,int i){
    if(T){
        cout<<'data:'<<T->data<<'layer:'<<i;
        PAL(T->lchild,i+1);
        PAL(T->rchild,i+1);
    }
}

void PALT(BiTree T){
    for(i = 1;i <= GetHeight(T);i++){
        PAL(T,i);
    }
}
```

图的有关算法

1. 图的存储
    1. 邻接矩阵法

```
#define MaxVerTexNum 100 //定义最大的定点数为100
typedef char VertexType;
typedef int EdgeType;
typedef struct{
    VertexType Vex[MaxVerTexNum];
    EdgeType Edge[MaxVerTexNum][MaxVerTexNum];
```

```
    int vexnum,arcnum;
}MGraph;
```

2. 邻接表法

```
#define MaxVerTexNum 100 //定义最大的定点数为100
typedef struct ArcNode{             //边表结点
    int adjvex;
    struct ArcNode *next;
}ArcNode;
typedef struct VNode{               //顶表结点
    VertexType data;
    ArcNode *first;
}VNode,*AdjList;
typedef struct{
    AdjList vertices[MaxVerTexNum];
    int vexnum,arcnum;
}ALGraph;
```

2. 图的建立

1. 邻接矩阵

```
void CreateGraph(MGraph G){
    int m,n,i,j,k;
    VertexType v;
    cout<<"请输入图的结点数和边数：";
    cin>>m>>n;
    G.vexnum = m;
    G.arcnum = n;
    cout<<"请输入各个顶点的值：";
    for(k = 0;k < G.vexnum;k ++){
        cin>>v;
        G.Vex[k] = v;
    }
    for(i = 0;i < G.vernum;i ++){
        for(j = 0;j < G.vexnum;j ++){
            G.Edge[i][j] = 0;
        }
    }
    cout<<"请输入边依附的顶点";
    for(k = 0;k < G.arcnum;k++){
         cin>>i>>j;
         G.Edge[i][j]=1;
         G.Edge[j][i]=1;
    }
}
```

2. 邻接表

```
void CreateALGraph(ALGraph G){
    int i,j,m,n,k,v;
    cout<<"请输入图的结点数和边数: "
    cin>>m>>n;
    G.vexnum = m;
    G.arcnum = n;
    cout<<"请输入各个顶点的值: ";
    for(k = 0;k < G.vexnum;k ++){
        cin>>v;
        VNode *p = new VNode;
        p.data = v;
        p.first = NULL;
        G.vertices[k] = p;
    }
    cout<<"请输入各条边依附的顶点对: ";
    for(k = 0;k < G.arcnum;k ++){
        cin>>i>>j;
        ArcNode *p = new ArcNode;
        p.adjvex = j;
        p.next = G.vertices[i].first;
        G.vertices[i].first = p.next;
        ArcNode *q = new ArcNode;
        q.adjvex = i;
        q.next = G.vertices[j].first;
        G.vetices[j].first = q;
}
```

3. 图的遍历
    1. 广度优先遍历

```
bool visited[MaxVertexNum];
void BFSTraverse(Graph G){
    for(i = 0;i < G.vexnum;i ++){
        visited[i] = false;
    }
    InitQueue(Q);
    for(i = 0;i < G.vexnum;i ++){
        if(!visited[i])
            BFS(G,i);
    }
}
void BFS(Graph G,int v){
    visit(v);
    visited[v] = ture;
    EnQueue(Q,v);
    while(!EmptyQueue(Q)){
        DeQueue(Q,v);
        for(w = FirstNeighbor(v);w > =0;w = NextNeighbor(G,v,w)){
            if(!visited[w]){
                visit(w);
```

```
                visited[w] = ture;
                EnQueue(Q,w);
            }
        }
    }
}
```

1. 深度优先遍历

```
bool visited[MaxVertexNum];
void DFSTraverse(Graph G){
    for(i = 0;i < G.vexnum;i ++){
        visited[i] = false;
    }
    InitQueue(Q);
    for(i = 0;i < G.vexnum;i ++){
        if(!visited[i])
            DFS(G,i);
    }
}
void DFS(Graph G,int v){
    visit(v);
    visited[v] = ture;
    for(w = FirstNeighbor(v);w > =0;w = NextNeighbor(G,v,w)){
        if(!visited[w]){
            DFS(G,w);
        }
    }
}
```

查找算法

1. 顺序查找

```
typedef struct{
    ElemType *elem;
    int TableLen;
}SSTable;

int Search_Seq(SSTable T,ElemType e){
    elem[0] = e;
    for(i = T.TableLen-1;T.elem[i]!=e;i --);
    return i;
}
```

2. 折半查找

```
int Binary_Search(SeqList L,ElemType key){
    int low = 0;
    int high = L.TableLen;
    while(high>=low){
        int mid = (low+high)/2;
        if(L.elem[mid]==key)
             return mid;
         else if(L.emel[mid]>key){
             high = mid-1;
         }
         else
             low = mid+1;
    }
    return -1;
}
```

排序算法

1. 直接插入排序

```
void InsertSort(ElemType A[],int n){
    int i,j;
    for(i = 2;i <= n;i ++){
        A[0] = A[i];
        for(j = i-1;A[0]<A[j];j --){
            A[j+1] = A[j];
        }
        A[j+1] = A[0];
    }
}
```

2. 折半插入排序

```
void InsertSort(ElemType A[],int n){
    int low,high,mid;
    int i,j;
    for(i=2;i<=n;i++){
        A[0] = A[i];
        low = 1;
        high = i-1;
        while(low<=high){
            mid = (low+high)/2;
            if(A[mid]>A[0])
                 high = mid-1;
             else
                 low = mid+1;
        }
```

```
        for(j = i-1;j >= high+1;j --){
            A[j+1] = A[j];
         }
         A[high+1] = A[0];
    }
}
```

3. 希尔排序

```
void ShellSort(ElemType A[],int n){
    for(dk=n/2;dk>=1;dk/=2){
        for(i=dk+1;i<=n;i++){
            if(A[i]<A[i-dk]){
                A[0] = A[i];
                for(j=i-dk;j>0 && A[0]<A[j];j--){
                    A[j+dk] = A[j];
                }
                A[j+dk] = A[j];
            }
        }
    }
}
```

4. 冒泡排序

```
void BubbleSort(ElemType A[],int n){
    for(i = 0;i < n;i ++){
        flag = false;
        for(j = n-1;j > i;j --){
            if(A[j]>A[j-1])
                 swap(A[j],A[j-1]);
                 flag = ture;
        }
        if(flag = ture)
             return;
    }
}
```

5. 快速排序

```
void QuickSort(ElemType A[],int low,int high){
    if(low<high){
        int pivotpos=Partition(A,low,high);
        QuickSort(A,low,pivotpos);
        QUickSort(A,pivotpos,high);
    }
}
```

```
Partition(ElemType A[],int low,int high){
    ElemType pivot = A[low];
    while(low<high){
        while(A[high]>pivot) high--;
        A[low] = A[high];
        while(A[low]<pivot) low++;
        A[high] = A[low];
    }
    A[low] = pivot;
    return low;
}
```

6. 简单选择排序

```
void SelectSort(ElemType A[],int n){
    for(i = 0;i < n-1;i ++){
        min = i;
        for(j = i+1;j < n;j ++){
            if(A[j]<A[min])
                min = j;
        }
        if(min != i)
            swap(A[i],A[k]);
    }
}
```

7. 堆排序
8. 二路归并排序

```
ElemType *B = new ElemType[n+1];
void Merge(ElemType A[],int low,int mid,int high){
    for(int k = low;k <= high;k ++){
        B[k] = A[k];
    }
    for(i = low,j = mid+1,k=i;i<=mid&&j<=high;k++){
        if(B[j]>=B[i])
            A[k] = A[++i];
        else
            A[k] = A{++j};
    }
    if(i>mid)
        A[k++] = A[j++];
    else
         A[k++] = A[i++];
}

void MergeSort(ElemType A[],int low,int high){
    if(low < high){
```

```
        int mid = (low+high)/2;
        MergeSort(A,low,mid);
        MergeSort(A,mid,high);
        Merge(A,low,mid,high);
    }
}
```