顺序表

```
//折半查找
void Search(int A[],int x){
    int low=0,high=n-1,mid;
    while(low<=high){
        mid=(low+high)/2;
        if(A[mid]==x)
            break;
        else if(A[mid]<x)
            low=mid+1;
        else
            high=mid-1;
    }
    if(A[mid]=x&&mid!=n-1){
        t=A[mid];
        A[mid]=A[mid+1];
        A[mid+1]=t;
    }
    if(low>high){
        for(i=n-1;i>high;i--)
            A[i+1]=A[i];c
        A[i+1]=x;
    }
}
```

```
//快速排序
int huafen(int A[],int low,int high){
    int pivot=A[low];       //第一个元素作为枢轴
    while(low<high){        //搜索枢轴位置
        while(low<high&&A[high]>=pivot) //循环找到比枢轴元素更小的值
            high--;
        A[low]=A[high];
        while(low<high& &A[low]<=pivot) //循环找到比枢轴元素更大的值
            low++;
        A[high]=A[low];
    }
    A[low]=pivot;           //放置枢轴
    return low;             //返回枢轴位置
}

void QuickSort(int A[],int low,int high){c
    if(low<high){
        int mid=huafen(A,low,high);
        QuickSort(A,low,mid-1);     //划分左表
        QuickSort(A,mid+1,high);    //划分右表
    }
}
```

```
//归并排序
void Merge(int A[],int n,int B[],int m,int C[]){
    int i=0,j=0,k=0;
    while(i<n&&j<m){
        if(A[i]<=B[j])
            C[k++]=A[i++];
        else
            C[k++]=B[j++];
    }
    while(i<n)
        C[k++]=A[i++];
    while(j<m)
        C[k++]=B[j++];
}
//事件复杂度O(N+M)
```

链表

```
//头插法
LinkList Creat(LinkList &L){
    LNode *s;
    int x;
    L=(LNode *)malloc(sizeof(LNode));
    L->next=NULL;
    scanf("%d",&x);
    while(x!=9999){
        s=(LNode *)mallocf(sizeof(LNode));
        s->data=x;
        s->next=L->next;
        L->next=s;
        scanf("%d",&x);
    }
    return L;
}
```

```
//尾插法
LinkList Creat(LinkList &L){
    L=(LNode *)malloc(sizeof(LNode));
    LNode *s;
    LNode *tail=L;
    int x;
    scanf("%d",&x);
    while(x!=9999){
        s=(LNode *)malloc(sizeof(LNode));
        s->data=x;
        tail->next=s;
        tail=s;
        scanf("%d",&x);
    }
}
```

```
//单链表遍历
LNode *p=L->next;
while(p!=null){
    ……;
    p=p->next;
}
```

树

```
//求二叉树高度
typedef struct BiTNode{
    int data;
    struct BiTNode *lchild,*rchild;
}BiTNode,*BiTree;

int height=0;

void ProOrder(BiTree T,int n){
    if(T==null)
        return;
    if(n>height)
        height=n;
    ProOrder(T->lchild,n+1);
    ProOrder(T->rchild,n+1);
}

int height(BiTree T){
    if(T==null)
        return 0;
    else
        PreOrder(T,1);
    return height;
}
```

```
//求树宽度
typedef struct BiTNode{
    int data;
    struct BiTNode *lchild,*rchild;
}BiTNode,*BiTree;
int width[MAX];
//先序遍历，同时统计各层结点数
void PreOrder(BiTiree T,int level){
    if(T==null) return;
    width[level]++;
    PreOrder(T->lchild,level+1);
    PreOrder(T->rchild,level+1);
}
//求树宽度
int treeWidth(BiTree T){
    for(int i=0;i<Max;i++)      //初始化数组
        width[i]=0;
    PreOrder(T,0);
    int maxWidth=0;
    for(int i=0;i<Max;i++){
        if(width[i]>maxWidth)
            maxWidth=width[i];
    }
    return maxWidth;
}
```

```
//求树WPL
typedef struct BiTNode{
    int weight;
    struct BiTNode *lchild,*rchild;
}BiTNode,*BiTree;
int WPL=0;

void PreOrder(BiTree T,int n){
    if(T==null) return;
    if(T->lchild==null&&T->rchild==null)
        WPL=WPL+T->weight*n;
    PreOrder(lchild,n+1);
    PreOrder(rchild,n+1);
}
int TreeWeight(BiTree T){
    PreOrder(T,0);
    return WPL;
}
```

```
//判断是否为二叉排序树

//利用中序遍历，检查中序遍历得到的遍历序列是否递增
typedef struct BiTNode{
    int weight;
    struct BiTNode *lchild,*rchild;
}BiTNode,*BiTree;

int temp=Min_int; //记录以访问过的最大值
bool flag=true;

void InOrder(BiTree T){
    if(T==null) return;
    InOrder(T->lchild);
    if(T->data>=temp)
        temp=T->data;
    else
        flag=false;

    InOrder(T->rchild);
}
```

```
//判断二叉树是否平衡

//利用后序遍历
typedef struct BiTNode{
    int weight;
    struct BiTNode *lchild,*rchild;
}BiTNode,*BiTree;

int PostOrder(BiTree T){
    if(T==null) return 0;
    int l=PostOrder(T->lchild);
    int r=PostOrder(T->rchild);

    if(left-right>1)  flag=false;
    if(left-right<-1) flag=false;

    if(left>right)
        return left+1;
    else
      return right+1;
}
```

图

```
#define Max 100 //顶点数目的最大值
typedef struct {
    char Vex[Max];      //顶点表
    int Edge[Max][Max]; //邻接矩阵，可记录权值
    int numV,numE;      //图的当前顶点数和边数
}MGraph;
```

```
#define Max 100;
//边表节点
typedef struct EdgeNode{
    int adjvex;             //这条边所指向的顶点的位置
    struct EdgeNode *next;//指向下一条边的指针
    int wight;
}EdgeNode;

//顶点表结点
typedef struct VNode{
    char data;              //顶点信息
    struct EdgeNode *first;//指向第一条依附该顶点的边的指针
}VNode,VList[Max];

//临界表
typedef struct{
    VList List;         //邻接表
    int numV,numE;          //图的当前顶点数和边数
}Graph;
```