19.1

```
//顺序表的定义
typedef struct SqList
{
    ElemType data[MaxSize];
    int Length;
}


//从顺序表中删除最小的元素(假设唯一)，并返回被删除元素的值。空出的位置由最后一个元素来填补
bool func(SqList &list,ElemeType &value)
{
    if(n<=0)
    {
        printf("顺序表为空\n");
        return false;
    }
    int minIndex=0;
    for(int i=1;i<;i++)
    {
        if(list.data[i]<list.data[minIndex])
        {
            minIndex=i;
        }
    }
    value=list.data[minIndex];
    list.data[minIndex]=list.data[n-1];
    list.Length=n-1;
    return true;
}

//设计一个算法，将顺序表L中的所有的元素逆置
void ReverseLinkList(SqList &list)
{
    ElemType temp;
    for(int i=0;i<list.Length/2;i++)
    {
        temp=list.data[i];
        list.data[i]=list.data[list.Length-i-1];
        list.data[list.Length-i]=temp;
    }
}

/*
对长度为n的顺序表L，编写一个时间复杂度为O(n)、空间复杂度为O(1)的算法，该算法删除线性表中所有值为x的元素
*/
//用k记录顺序表L中不等于x的元素个数，遍扫描L遍统计k，将不等于x的元素向前移动k个位置
void delete_same_x(SqList &list,ElemType x)
{
    int k=0;//记录值不等于x的元素个数
    for(int i=0;i<list.Length;i++)
```

```
    {
        if(list.data[i]!=x)
        {
            list.data[k]=l.data[i];
            k++;//不等于x的个数增加1
        }
    }
}

//解决方法二：用k记录顺序表中等于x的元素个数，并将不等于x的元素前移k个位置
void delete_same_x_1(SqList &list,ElemType x)
{
    int k=0;//用于记录值等于x的元素个数
    while(i<list.Length)
    {
        if(list.data[i]==x)
        {
            k++;
        }else
        {
            list.data[i-k]=list.data[i];//当前元素迁移k个位置
        }
        i++;

    }
    list.Length=list.Length-k;//修改顺序表的长度
}


//从有序表删除值在给定值s与t之间的所有的元素(包含s和t)，如果s和t的值不和或顺序表为空，则显示信息并退出
bool func(SqlList &l,ElemType s,ElemType t)
{
    int i=0;//指向第一个大于s的元素下表
    int j=0;//指向第一个大于t的元素下表
    while(i<l.Length && l.data[i]<s)
    {
        i++;
    }
    if(i>=l.Length)
    {
        return false;
    }
    while(j<l.Length && l.data[j]<=t)
    {
        j++;
    }
    //把l和l后面的元素向前移动
    for(l;l<l.Length;l++,j++)
    {
        l.data[j]=l.data[l];
    }
```

```
    l.Length=j;
    return false;
}

//从有序顺序表中删除所有值重复的元素，使表中的元素值均不相同
//因为是有序表，所以表中相同值的元素都是连续出现的，可以利用这个特性来写

void func(SqList &l)
{
    if(l.LEngth==0)
    {
        return false;
    }
    int i,j;//i存储第一个不相同的元素，j为工作指针
    for(i=0,j=1;j<l.Length;j++)
    {
        if(l.data[i]!=l.data[j])//查找下一个与上一个不同的元素
        {
            l.data[++i]=l.data[j];//找到后将元素向前移

        }
    }
    l.Length=i+1;
    return true;
}

//将两个有序表合并为一个新的有序表，并由函数返回顺序表结果
//首先，按顺序不断取下两个顺序表表头比较小的结点存入新的顺序表中，然后看哪个表还有剩余，将剩余的部分加到新
bool Merge(SqList a,SqList b,SqList &c)
{
    if(a.Length+b.Length>c.Length)
    {
        return false;
    }
    int i=0,j=0,k=0;
    while(i<a.Length && j<b.Length)
    {
        if(a.data[i]<=b.data[j])
        {
            c.data[k++]=a.data[i++];
        }else{
            c.data[k++]=b.data[j++];
        }
        while(i<a.Length)
        {
            c.data[k++]=a.data[i];
        }
        while(j<b.Length;j++)
        {
            c.data[k++]=b.data[j++];
        }
```

```
    }
    c.Length=k;
}


//元素有序的线性表中，用最少的时间查找数值为x的元素，若找到则将其与后继元素位置交换，若找不到则将其插入线
bool func(SqlList &l,ElemType x)
{
    int i;
    for(i=0;i<l.Length;i++)
    {
        if(l.data[i]>=x)
        {
            break;
        }
    }
    //等于x且不是最后一个元素
    if(l.data[i]==x &&i<l.Length-1)
    {
        ElemType temp=l.data[i];
        l.data[i]=l.data[i+1];
        return true;
    }
    for(int j=l.Length;j>i;j--)
    {
        l.data[j]=l.data[j-1];
    }
    return false;
}

//题目中说用最少的时间，应该用折半查找
void func(ElemType data[],ElemType x)
{
    int low=0,high=n-1,mid;
    while(low<high)
    {
        mid=(low+high)/2;
        if(data[mid]<x)
        {
            low=mind+1;
        }else{
            high=mid-1;
        }
    }
    if(data[mid]==x && mid!=n-1)
    {
        ElemType temp=data[mid];
        data[mid]=data[mid+1];
        data[mid+1]=temp;
    }
    if(low>high)//查找失败,插入元素X
```

```
    {
        for(i=n-1;i>high;i--)
        {
            data[i+1]=data[i];//后移元素
        }
        data[i+1]=x;
    }
}


//数组的循环左移p位
//算法思想：把问题视为数组ab转换成ba(a代表数组的前p项，b代表数组的后n-p项)，先甲肮a逆置，再把逆置，最后i
void Reverse(int data[],int from,int to)
{
    int temp;
    for(int i=0;i<(to-from+1)/2;i++)
    {
        temp=data[from+i];
        data[from+i]=data[to-i];
        data[to-i]=temp;
    }
}

void Converse(int data[],int n,int p)
{
    Reverse(data,0,p-1);
    Reverse(data,p,n-1);
    Reverse(data,0,n-1);
}


//2011年真题
//一个长度为L的升序序列S,处在(L/2，向上取整)的位置的元素叫做序列的中位数。现在有两个升序序列A和B,找出这两
//利用二路归并的思路，从每个序列中取出最小的元素
void func(int a[],int b[],int n,int &value)
{
    int i=0;
    int a_index=0;
    int b_index=0;
    while(true)
    {
        if(a[a_index]<b[b_index])
        {
            a_index++;
            if(i==n-1)
            {
                value=a[a_index];
            }
        }else{
            b_index++;
            value=b[b_index];
```

```
        }
        i++
    }
}

//2013年真题
//在一个整数序列中一个元素出现的次数超过数列个数的一半，那么称这个元素为序列的主元素
//设计一个算法，找出序列的主元素

//思路一：可以先进行排序，然后很快可以判断出是否有主元素了
//这里先用冒泡排序进行排序
void Sort(int data[],int n)
{
    int temp;
    bool flag=true;
    while(flag)
    {
        flag=false;
        for(int i=0;i<n-1;i++)
        {
            if(data[i]>data[i+1])
            {
                temp=data[i];
                data[i]=data[i+1];
                data[i+1]=temp;
                flag=true;
            }
        }
    }
}

int func(int data[] int n)
{
    //先进行排序
    Sort(data,n);
    //如果存在主元素，肯定会连续出现
    int num=1;//记录一个元素连续出现的次数
    int i=1;
    int elem;
    while(num<=n/2 && i<n)
    {
        if(data[i]==data[i-1])
        {
            num++;
            elem=data[i];
        }else{
            num=1;
        }
    }
    if(num>n/2)
    {
```

```
        return return elem;
    }else{
        return -1;
    }
}
//2018年真题
//设计一个算法来找出一维数组中未出现的最小正整数
//用暴力破解，先进行排序，再找出没出现的元素
//这里用快速排序
int func(int data[],int n)
{
    QuickSort(data,n);
    for(int i=0;i<n-1;i++)
    {
        if(data[i]<=0 && data[i+1])
        {
            return data[i+1]
        }
    }
}




void QuickSort(int data[],int n)
{
    QSort(data,0,n-1);
}


void QSort(int data,int low,int high)
{
    int pivotPos=Partition[data,low,high];
    QSort(data,pivotPos+1,high);
    Qsort(data,low,pivotPos-1);
}

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

```
        data[high]=data[low];
    }
    data[low]=pivot;

}

//其实题目中说了设计一个时间上尽可能高效的算法，找出最小正整数，所以可以考虑用空间换时间的策略
//考试的时候也要注意题目的意思是要空间复杂度要高还是时间复杂度还是说了尽可能高效(这个是既要考虑时间复杂度
//分配一个用于标记数组B[n]用来记录A中是否出现了1-n中的正整数
//算法思路：从data[0]开始遍历，若0<data[i]<=n,则令b[data[i]-1]=1,否则不做操作

int findMissMin(int data[],int n)
{
    int i,*B;
    B=(int*)malloc(sizeof(in)*n);
    memset(B,0,sieof(int)*n);
    for(int i=0;i<n;i++)
    {
        if(data[i]>0 && data[i]<n)
        {
            B[data[i]-1]=1;
        }
    }
    for(i=0;i<n;i++)
    {
        if(B[i]==0)
        {
            break;
        }
    }
    return i+1;
}
//时间复杂度是O(n),空间复杂度是O(n)
```

//链表有关的算法

```
//单链表的结点类型定义
typedef struct LNode
{
    ElemType data;//数据域
    struct LNode *next;//指针域
}LNode,*LinkList

//头插法
void headInsert(LinkList &l,int data[],int n)
{
    LNode *p;//用于指向要插入的结点
    for(int i=0;i<n;i++)
    {
        p=(LNode*)malloc(sizeof(LNode));
        p->data=data[x];
        p->next=l.next;//p的指针指向L结点的下一个指针
        l.next=p;//头节点指向p
    }
}

//尾插法
void tailInsert(LinkList &l,int data[],int n)
{
    LNode *p;
    for(int i=0;i<n;i++)
    {
        p=(LNode*)malloc(sizeof(LNode));
        p->data=data[i];
        p->next=l->next;

    }
}

//设计一个算法，删除不带头节点的单链表L中所有值为x的结点
//思想：设f(L,x)的功能是删除以L为首结点指针的单链表中所有值等于x的结点，显然有f(L->next,x)的功能是删除以
//终止条件：f(L,x)=不做任何事情 若L为空表
void func(LinkList &l,ElemType x)
{
    LNode *p;
    if(l==NULL)
    {
        return;
    }
    if(L->next==x)
    {
        p=L;l=l->next;
        free(p);
        func(l,x);
    }else
    {
```

```
        func(l->next,x);
    }
}

//删除带头节点的单链表L中，删除所有值为x的结点，并释放空间，值为x的结点不唯一
void func(LinkList &l,int x)
{
    LNode *p=l->next, *pre=L,*q;
    while(p!=NULL)
    {
        q=p;//q指向该结点
        p=p->next;
        pre->next=p;
    }
}
```

//排序算法 2021-11-27
//快速排序

```
void QuickSort(int data[],int n)
{
    QSort(data,0,n-1);
}


void QSort(int data[],int low,int high)
{
    if(low<high)
    {
        pivot=Partition(data,low,high);
        QSort(data,low,pivot-1);
        QSort(data,pivot+1,high);
    }
}

//划分
int Parition(int data[],int low,int high)
{
    int pivot;
    pivot=data[low];
    //从线性表的两端交替地向中间扫描
    while(low<high)
    {
        while(low<high && data[high]>=pivot)
        {
            high--;
        }
        data[low]=data[high];
        while(low<high && data[low]>=pivot)
        {
            low++;
        }
        data[high]=pivot;
    }
    data[low]=pivot;
    return low;
}
```

//插入排序

```
void InsertSort(int data[],int n)
{
    int i,j,temp;
    for(i=1;i<n-1;i++)
    {
        if(data[i]<data[i-1])//比前面的元素要小，就需要插入到前面的有序队列中
        {
            j=i-1;
            tem=data[i];//要插入的元素先保存起来
            while(j>=0 && temp<data[j])//从前面有序队列中最后一个元素开始往后挪元素，知道第一个元素或
            {
                data[j+1]=data[j];
                j--
            }
            data[j+1]=temp;
        }
    }
}
```

//折半插入算法

```
//算法思想：算法同直接插入排序，只不过使用折半查找的方法来寻找插入位置
void BinInsertSort(int data,int n)
{
    int i,j,low,high,mid,temp;
    for(i=1;i<n-1;i++)
    {
        temp=data[i];
        low=0;
        high=i-1;
        while(low<=high)
        {
            mid=(low+high)/2;
            if(temp>data[mid])
            {
                low=mid+1;
            }else
            {
                high=mid-1;
            }
        }
        for(j=i;j>low;j--)
        {
            data[j]=dataa[j-1];
        }
        data[low]=temp;
    }
}
```

//选择排序

```
void SelectInsert(int data[],int n)
{
    int i,j,min,temp;
    for(i=0;i<nli++)
    {
        min=i;
        for(j=i+1;j<n;j++)
        {
            if(data[min]>data[j])
            {
                min=j;
            }
        }
        if(min!=i)
        {
            temp=data[min];
            data[min]=data[i];
            data[i]=temp;
        }
    }
}
```

//并查集

```
#define Size 13
int UFSets[Size];//集合元素数组

void Initial(int S[])
{
    for(int i=0;i<Size;i++)
    {
        S[i]=-1;//刚开始不知道结点之间的关系，可以看成独立的集合
    }
}

//Find 查操作 找到x所在的集合
int Find(int S[],int x)
{
    while(S[x]>=0)//因为根结点是用负数表示，所以小于零说明找到根结点
    {
        x=s[x];
    }
    return x;
}

//Union "并"操作 将两个集合合并成一个集合
void Union(int S[],int root1,int root2)
{
    //root1和root2是不同的集合
    if(root1!=root2)
    {
        //将root2连接到root1的下面
        s[root2]=root1;
    }
}


//这种无脑的合并方式可能会使得集合树的高度变高,会提高Find操作的复杂度，如果把小的树挂在大树的下面，则可以
void Union(int S[],int root1,int root2)
{
    if(root1==root2)
    {
        return;
    }
    if(S[root2]>S[root1])//root2的结点少
    {
        S[root1]+=s[root2];//累加结点总数
        S[root2]=root1;//小树合并到大树
    }else{
        S[root2]+=root1;
        S[root1]=root2;
    }
}
//Union操作优化以后最坏的Find复杂度是O(log_2n)
```

```
//可以通过压缩路径的优化方法实现更低的Find复杂度
int Find(int S[],int x)
{
    int root=x;
    while(S[x]>=0)//循环找到根
    {
        root=S[root];
    }
    while(x!=root)//压缩路径
    {
        int t=S[x];//x指向x的父亲结点
        S[x]=root;
        x=t;

    }
    return root;
}


//用并查集判断图的连通分量数
int ComponentCount(int g[5][5])
{
    //g[5][5]是二维数组表示的邻接矩阵
    int S[5];//定义，初始化并查集
    for(int i=0;i<5;i++)
    {
        S[i]=-1;
    }
    //遍历各条遍(无向图，遍历上三角部分即可)
    for(int i=0;i<5;i++)
    {
        for(int j=i+1;j<5;j++)
        {
            if(g[i][j]>0)//终点i,j之间有遍
            {
                int iRoot=Find(S,i);//通过并查集找到i所属的集合
                int jRoot=Find(S,j);//通过并查集找到j所在的集合
                if(iRoot!=jRoot)
                {
                    Union(S,iRoot,JRoot);
                }
            }
        }
    }
    //统计有几个连通分量
    int count=0;
    for(int i=0;i<5;i++)
    {
        if(S[i]<0)
        {
            count++;
```

```
        }
    }
    return count;
}

//判断图是否有环

int hasAcyclic(int g[5][5])
{
    //g[5][5]是二维数组表示的邻接矩阵
    int S[5];//定义，初始化并查集
    for(int i=0;i<5;i++)
    {
        S[i]=-1;
    }
    //遍历各条遍(无向图，遍历上三角部分即可)
    for(int i=0;i<5;i++)
    {
        for(int j=i+1;j<5;j++)
        {
            if(g[i][j]>0)//终点i,j之间有遍
            {
                int iRoot=Find(S,i);//通过并查集找到i所属的集合
                int jRoot=Find(S,j);//通过并查集找到j所在的集合
                if(iRoot!=jRoot)
                {
                    Union(S,iRoot,JRoot);
                }else{
                    return 1;//在一个连通图中，但凡再多一条遍，必定有环
                }
            }
        }
    }
    return 0;//图中没有环
}
```

//树相关的代码

```
//二叉链表存储结构
typedef struct BTNode
{
    char data;
    struct BTNode *lchild;
    struct BTNode *rchild;
}BTNode,*BiTree;


//树的前序遍历,递归遍历
void PreOrder(BTNode *p)
{
    if(p!=NULL)
    {
        Visit(p);
        PreOrder(p->lchild);
        PreOrder(p-rchild);
    }
}
//非递归遍历
//非递归遍历需要借助一个栈来实现
void PreOrder2(BiTree T)
{
    InitStack(S);//初始化栈
    BiTree p=T;
    while(p!=NULL || !IsEmpty(S))//栈不空或者p不空时循环
    {
        if(p)//一路向左
        {
            visit(p);Push(S,p);//访问当前的结点，并入栈
            p=p->lchild;//左还是不空，一直向左走
        }
        else{
            Pop(S,p);//出栈，并转向出栈结点的右孩子
            p=p->rchild;
        }
    }
}
//中序遍历
void InOrder(BTNode *p)
{
    if(p!=NULL)
    {
        INOrder(p->lchild);
        Visit(p);
        InOrder(p->rchild);
    }
}

//中序遍历的 非递归实现
```

```
void InOrder(BiTree T)
{
    InitStack(S);//初始化栈
    BiTree p=T;
    while(p!=NULL || !IsEmpty(S))
    {
        if(p)
        {//一路向左
            Push(S,p);
            p=p->lchild;
        }else{
            Pop(S,p);visit(p);
            p=p->rchild;
        }
    }


}
```

```
//后序遍历
void PostOrder(BTNode *p)
{
    if(p!=NULL)
    {
        PostOrder(p->lchild);
        PosrOrder(p->rchild);
        Visit(p);
    }
}
//后序遍历的非递归实现
//算法思想：后序遍历的非递归遍历二叉树是先访问左子树，再访问右子树，最后访问根结点。
//设置一个辅助指针指向最近访问过的结点
void PostOrder(BiTree T)
{
    InitStack(S);//初始化辅助栈
    BiTree p=T;
    BiTree 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);//读取栈顶指针
```

```
            if(p->rchild !=NULL && p->rchild!=r)
            {
                p=p->rchild;
                push(S,p);
                p=p->lchild;
            }
            else
            {
                pop(S,p);
                visit(p->data);
                r=p;
                p=NULL;
            }
        }

    }
}


//层序遍历
void LevelOrder(BiTree T)
{
    InitQueue(Q);//初始化辅助队列
    BiTree p;
    EnQueue(Q,T);//根结点入队
    while(!IsEmpty(Q))//队列不空循环
    {
        DeQueue(Q,p);//队头结点出队
        visit(p);
        if(p->lchild!=NULL)
        {
            EnQueue(Q,p->lchild);//左子树不空，则左子树根结点入队
        }
        if(p->rchild!=NULL)
        {
            EnQueue(Q,p->rchild);
        }
    }
}

//给出二叉树自下而上，从右到左的层序遍历
//算法思想：把二叉树的层序遍历的序列用栈转置一下就可以
void func(BiTree T)
{
    InitStack(S);//初始化辅助栈
    InitQueue(Q);//初始化辅助队列
    BiTree p;
    EnQueue(Q,T);//根结点入队
    while(!IsEmpty(Q))
    {
        DeQueue(Q,p);//队头元素出队
```

```
        Push(S,p);//入栈
        if(p->lchild!=NULL)
        {
            EnQueue(Q,p->lchild);
        }
        if(p->rchild!=NULL)
        {
            EnQueue(Q,p->rchild);
        }
    }
    while(!IsEmpty(S))
    {
        Pop(S,p);
        visit(p);
    }
}

//假设二叉树采用二叉链表存储结构，设计一个非递归算法求出二叉树的高度
//递归版
int treeDepth(BiTree T)
{
    if(T==NULL)
    {
        return 0;
    }
    int l=treeDepth(T->lchild);
    int r=treeDepth(T->rchild);
    return l>r?l+1:r+1;
}
//非递归版本
//采用层序遍历，设置变量level记录当前结点所在的层数，设置变量last指向当前层的最右边的结点，每次层次序列出
int treeDepth(BiTree T)
{
    if(T==NULL)
    {
        return 0;
    }
    int front=-1,rear=-1;
    int last=0,level=0;
    Bitree Q[MaxSize];//设置队列Q,元素是二叉树的结点指针且容量足够
    Q[++rear]=T;//根结点入队
    BiTree p;
    while(front<rear)//队列不空时循环
    {
        p=Q[++front];//队列元素出队，即正在访问的结点
        if(p->lchild!=NULL)
        {
            Q[++rear]=p->lchild;
        }
        if(p->rchild!=NULL)
        {
```

```
            Q[++rear]=p->rchild;
        }
        if(front==last)
        {
            level++;
            last=rear;
        }

    }
    return level;
}

//二叉树按二叉链表的形式存储，判断一个二叉树是否是完全二叉树
//算法思想：采用层序遍历，将所有的结点加入队列。遇到空结点时，查看其后面是否有非空结点。若有，则二叉树不是
bool IsCompelete(BiTree T)
{
    //本算法判定给定的二叉树是否是完全二叉树
    InitQueue(Q);//初始化队列
    if(T==NULL)
    {
        return true;
    }
    EnQueue(Q,T);//根节点入队
    while(!IsEmpty(Q))
    {
        DeQueue(Q,p);
        if(p!=NULL)//结点不空，左右结点入队
        {
            EnQueue(Q,p->lchild);
            EnQueue(Q,p->rchild);
        }
        else
        {
            while(!IsEmpty(Q))
            {
                DeQueue(Q,p);
                if(p!=NULL)
                {
                    return false;
                }
            }
        }
    }
    return true;
}
//交换二叉树的左右孩子结点
void swap(BiTree T)
{
    if(T)
    {
        swap(T->lchild);
```

```
        swap(T->rchild);
        temp=T->lchild;
        T->lchild=T->rchild;
        T->rchild=temp;
    }
}

//设计一个算法，求出先序遍历中第k个元素的结点值
//算法思想：设置一个全局变量记录已经访问过的结点的序号
//当二叉树为空时，返回特殊字符'#',当i==k时，表示找到了满足条件的结点,返回该结点
int i=1;//设置全局变量
ElemType PreNode(BiTree T,int k)
{
    if(T==NULL)
    {
        return '#';
    }
    if(i==k)
    {
        return T->data;
    }
    i++;
    ch=PreNode(T->lchild,k);
    if(ch!='#')
    {
        return ch;
    }
    ch=PreNode(T->rchild,k);
    if(ch!='#')
    {
        return ch;
    }
}









}
```