王道 P19-3

删除顺序表中值为 x 的元素，时间复杂度为 O(n)，空间复杂度 O(1)

方法一：采用交换的策略

采用国旗的方法，将值为 x 的元素交换至顺序表的尾部，

（链表形式，更简单，遍历过程保存前驱节点用于删除，即可）

考虑，考虑极端情况，顺序表中无 x，不执行交换，优于不交换算法，总之二者各有优劣。全是 x，交换 3n 次

```
void deleteX(SqList &L,int x){
    int i=0,j=L.length;
    ElemType temp;
    while(i<j){
        if(L.data[i]==x){
            temp = L.data[i];
            L.data[i] = L.data[j-1];
            L.data[j-1] = temp;
            j--;
        }
        else i++;
    }
    L.length = i;
}
```

方法二：

不执行交换。极端情况，顺序表中无 x，则赋值操作执行 n 次

全是 x，不执行赋值操作，所以可知，x 越多，该算法越优于方法一。

```
void deleteX_2(SqList &L,int x){
      int j=-1;    //j 保存遍历过程中，非 x 的序列尾端元素下标
      for(int i=0;i<L.length;i++){
       if(L.data[i]!=x){
           L.data[j+1] = L.data[i];
           j++;
       }
      }
      L.length = j+1;
}
```

P20-11

等长的有序序列，查找中位数，L/2 向上取整数。

分析：利用有序序列，进行二分查找的思想，将等长序列的进行拆分，判断出属于整个序列左侧、右侧的数，保留下序列中间的数。

再进行递归至只有两个数的时候，选择小的返回就可以了。

注意：两个偶数长度，两个奇数长度的序列，截取的下标不一致，需分开讨论

```
void getMidNum(SqList S1,SqList S2,int &mid){
    mid = supportMid(S1,S2,0,S1.length-1,0,S2.length-1);
}
int supportMid(SqList S1,SqList S2,int front1,int rear1,int front2,int rear2){
    int mid1= (front1+rear1)/2;
    int mid2= (front2+rear2)/2;
    int lenth = rear1-front1+1;
    if(lenth==1)
return S1.data[mid1]>S2.data[mid2]?S2.data[mid2]:S1.data[mid1];
    if(S1.data[mid1]==S2.data[mid2]) return mid1;
    else if(S1.data[mid1]>S2.data[mid2]){
        if(lenth%2==0)
        return supportMid(S1,S2,front1,mid1,mid2+1,rear2);
        else return supportMid(S1,S2,front1,mid1,mid2,rear2);
    }
    else{
        if(lenth%2==0)
            return supportMid(S1,S2,mid1+1,rear1,front2,mid2);
        else return supportMid(S1,S2,mid1,rear1,front2,mid2);
    }
}
```

Tips:关于数列中，重复数值的问题，最普适的方法，利用**散列**的思想，将重复的数据进行**统计**。作用：去重，统计数据频度（找主元素）。
关于散列的基本思路：当未知数组值得范围时，散列的辅助数组大小未知，故不好进行散列，但是，注意如果题目中将**数组值大小**给出，则**一定是进行散列**，用时间换空间。
*进行整体去重的算法，同时保留数据原始的顺序,另一种算法，采用**国旗**的思路：
利用 count 记录重复的数据个数，i 作为遍历过程中的数组下标进行数组遍历。

```
void delete_same(SqList &L){
    int temp,count = 0,j,i;
    for(i = 0;i<L.length;++i){
        temp = L.data[i];
        for(j = i-count-1;j>=0;--j){
            //此处可用散列进行优化，但是要耗费存储空间
            if(L.data[j] == temp){
```

```
                ++count;
                break;
            }
        }
        if(j<0) L.data[i-count]=L.data[i];
    }
    L.length = L.length -count;
}
```

删除值为 X，利用 j 保存不含有 X 的数组，也是**国旗**思路的延伸。

```
void deleteX_2(SqList &L,int x){
    int j=-1;
    for(int i=0;i<L.length;i++){
        if(L.data[i]!=x){
            L.data[j+1] = L.data[i];
            j++;
        }
    }
    L.length = j+1;
}
```

//转置算法，可用于序列的循环移动

```
void reverse(SqList &L,int start,int end){
    int i=start,j=end,temp;
    while(i<j){
        temp = L.data[i];
        L.data[i] = L.data[j];
        L.data[j] = temp;
        i++;j--;
    }
}
```

查找主元素的另一种思路，主元素：在序列中的频度大于等于表长的一半。

```
int seekMain(SqList &L){
    int count=1,temp=L.data[0];
    for(int i=1;i<L.length;++i){//查找可能的主元素，因为个数大于一半，所以哪怕是极端情况下，左侧全是相同的数，右侧全是相同的数，主元素也能剩下一个，被发现。
        if(L.data[i]==temp) ++count;
        else --count;
```

```
        if(count == 0){
            temp = L.data[i];
            count = 1;
        }
    }
    count = 0;
    for(int i=0;i<L.length;++i){
        if(L.data[i]==temp) ++count;
        if(count>L.length/2) return temp;
    }
    cout<<"没有主元素"<<endl;
    return -1;
}
```

Tips:关于链表删除的重要技巧，利用 C++中的引用符，**直接引用前驱节点的 next 指针域**，用于删除当前节点，此处在**二叉树删除**中也有应用。
递归删除链表中值为 X 的节点，不带头节点
如需删除**带头节点**的，则需要注明，将**头节点的 data 域设置成 INF**
或者不设置也可，**系统会自动将头节点的 data 域设置为系统最大值。**

```
void recur_Del_Linked(List &head,int x){
    if(head==NULL) return;
    if(head->data == x) {
        head = head->next;
        recur_Del_Linked(head,x);
    }
    else recur_Del_Linked(head->next,x);
}
```

常规的删除带头节点，需要进行走链。特别的，对于**不带头节点**的链表需要对删除的节点是否是当前 **head** 指向节点进行**单独讨论**。

王道中出现了链表的模式匹配问题：
防一手，记录 KMP 算法的 nextval[]数组的求法，以及主函数调用

```
void getNextval(String T,int nextval[]){
    int i=1,j=0;
    nextval[1] = 0;
    while(i<T.length){
        if(j==0||T.str[i]==T.str[j]){
            ++i;++j;
            if(T.str[i]!=T.str[j]) nextval[i]=j;
```

```
            else nextval[i] = nextval[j];
        }
        else j=nextval[j];
    }
}

int Index_KMP(String M,String T,int nextval[]){
    int i=1,j=1;
    while(i<=M.length&&j<=T.length){
        if(j==0||M.str[i]==T.str[j]){
            ++i;++j;
        }
        else j=nextval[j];
    }
    if(j>T.length) return i-T.length;
    else return 0;
}
```

注意：对于循环队列，规定一律是，front 有值，rear 的指向无值。
常规队满：（rear+1）%maxSize==front
队空：rear==front
队中元素：（rear-front+maxSize）%maxSize
以牺牲一个存储空间为代价。
循环对列，当采用 tag 作为队满的标志时，

```
bool Dequeue(TagQueue &q,Data &e){
    if(q.tag==0) return false;
    e = q.data[q.front];
    q.front=(q.front+1)%maxSize;
    if(q.front==q.rear) q.tag = 0;
    return true;
}
bool Enqueue(TagQueue &q,Data e){
    if(q.tag==1) return false;
    q.data[q.rear] = e;
    q.rear = (q.rear+1)%maxSize;
    if(q.front == q.rear) q.tag = 1;
    return true;
}
```

双栈模拟队列
当输入栈满，而输出栈非空时，不能执行“进队操作”
当输入栈满，而输出栈为空时，将输入栈的数据全部转移至输出栈

执行“入队操作”

```
bool analogyEnQueue(Stack &in,Stack &out,int x){
    if(in.top==maxSize-1&&out.top!=-1) return false;
    if(in.top==maxSize-1){
        Data temp;
        while(in.top!=-1){
            pop(in,temp);
            push(out,temp);
        }
    }
    push(in,x);
    return true;
}
```

当双栈都为空时，不能执行“出队”操作

当双栈不都为空时，如果输入为空，则将输入栈中的所有数据转移至输入栈中，

执行“出队操作”

```
bool intimateDeQueue(Stack &in,Stack &out,Data &e){
    if(in.top==-1&&out.top==-1) return false;
    if(out.top==-1){
        Data temp;
        while(in.top!=-1){
            pop(in,temp);
            push(out,temp);
        }
    }
    pop(out,e);
    return true;
}
```

栈的应用

1、括号匹配

```
bracketCheck(char str[]){
      int i=0;
      Stack s;
      Init_stack(s);
      char c;
      while(str[i]!='\0'){
       switch (str[i])
```

```
        {
        case '(':push(s,'(');break;
        case '[':push(s,'[');break;
        case '{':push(s,'{');break;
        case '}':pop(s,c);
            if(c!='{') return false;break;
        case ')':pop(s,c);
            if(c!='(') return false;break;
        case ']':pop(s,c);
            if(c!='[') return false;break;
        default:break;
        }
        ++i;
       }
       if(s.top!= -1) return false;
       return true;
}
```

2、中缀表达式转后缀

```
void infixTomidfix(ElemType in[],ElemType out[]){
    int i=0,j=0;
    Stack s;
    Init_stack(s);
    while(in[i]!='\0'){
        if(in[i]>='0'&&in[i]<='9'){
            out[j++]=in[i++];
        }
        else if(in[i]=='('){
            push(s,'(');++i;
        }
        else if(in[i]=='+'||in[i]=='-'||
                in[i]=='*'||in[i]=='/'){

            if( s.top==-1 || getPriority(getTop(s))<getPriority(in[i]) || getTop(s)=='('){
                    push(s,in[i]);++i;
                }
                else{
                    pop(s,out[j++]);
                }
        }
```

```
        else if(in[i]==')'){
            while(getTop(s)!='('){
                pop(s,out[j++]);
            }
            --s.top;
            ++i;//经常忘了下移
        }
    }//while
    while(s.top!=-1){
        pop(s ,out[j++]);
    }
}
```

3、后缀表达式的计算

关于中缀表达式的直接计算，只需要，在中缀表达式转后缀表达式的时候，在符号出栈的位置，将数字栈中的数据计算即可。

```
bool support(float left,char op,float right,float &out){
    switch(op){
    case '+': out = left+right;break;
    case '-': out = left-right;break;
    case '/':if(right==0) return false;
             else out = left/right;
             break;
    case '*': out = left*right;break;
    }
    return true;
}

bool cal(char in[],float &out){
    Stack s; InitStack(s);
    int i=0;
    while(in[i]!='\0'){
        if (in[i]>='0'&&in[i]<='9') Push(s,in[i++]-'0');
        else{
            float left,right,temp;
            Pop(s,right);
            Pop(s,left);
            if(support(left,in[i++],right,temp)) Push(s,temp);
            else return false;
        }
```

```
    }
    out = getTop(s);
    return true;
}
```

4、栈中最小值问题，O(1)时间复杂度，获取到当前栈中的最小值

原理：利用辅助栈，将保存当前栈中的最小值。

注意：最大值的思想也类似

```
Stack support;//辅助站中保存，当前栈中的最小值的下标
ElemType tag=INF;//tag 保存的是当前栈中的最小值

void Push_min(Stack &s,ElemType e){
    if (s.top <maxSize -1)
    {
        s.data[++(s.top)] = e;
        if(e<tag) {
            support.data[++(support.top)] = s.top;//入栈最小值的下标
            tag = e;//保存最小值
        }
    }
    else{
        cout<<"栈已满，禁止入栈"<<endl;
    }
}
void Pop_min(Stack &s,ElemType &e){

    if (!IsEmptyStack(s))
    {
        e = s.data[s.top];
        s.top--;
        if(s.top<support.data[support.top]) {
        //如果当前栈中的元素的下标<辅助栈顶值，说明，出栈的是当前的最小值
          那么辅助栈也应相应的出栈栈顶元素，同时修改 tag 值为辅助栈顶元素
          所对应的下标值
            support.top--;
            tag = s.data[support.data[support.top]];
        }
    }
    else{
        cout<<"栈空，禁止出栈"<<endl;
        tag = INF;
```

```
    }
}
bool getMin(Stack s,ElemType &e){
    if(!IsEmptyStack(s)) {e = tag; return true;}
    return false;
}
```

5、获取树的高度

方法一：

采用递归的特点、函数名的传参值，给结点添加了一个临时的层高 level 域

在递归时，利用函数名，将每个结点的层高（从根节点开始数至当前层的高度）进行标记 level。

最后，需要将所标记的 level 值中最大的，返回出来。

此方法的好处在于：没有采用队列，不会出现队列溢出的情况，在无法估计树的最大宽度时，可以用来获取任意规模的树高。其作用，类似于给每一个结点添加了一个临时的层高 level 域。

改进：递归转非递归栈。只需将栈中元素改成：结点指针+level 即可

```
//此处封装 getHighth()函数，是为了避免输入不规范。
int getHigh(BTree root){
    return getHighth(root,1);
}
int getHighth(BTree root,int level){
    if(root == NULL) return level-1;
    int left=level,right=level;

    //获取左子树中，层高 level 的最大值
    left = getHighth(root->lchild,level+1);

    //获取右子树中，层高 level 的最大值
    right = getHighth(root->rchild,level+1);
    //比较后，返回二者最大值
    return left>right?left:right;
}
```

方法二：递归转成栈

采用先序非递归的思想，添加 level 数组，用以记录栈中元素的层高 1

```
int getHigh3(BTree root){
    if(root == NULL) return 0;
    BTNode *s[maxSize] ,*p = root;
    int top = -1, level[maxSize] ,1 = 0, max = 0;//max 记录最大层高
```

```
    while(p||top!=-1){
        if(p){
            //p 指针入栈，同时 p 对应的层高 l 记录到 level 数组中
            s[++top] = p;
            level[top] = ++l;//此时 l 记录的是其父节点的层高，所以进行+1

            if(max<l) max= l;//用 max 记录最大层高值

            p = p->lchild;
        }
        else{
            p = s[top--];//出栈栈顶元素
            l = level[top+1];//同时修改 l 值为 p 的层高
            p = p->rchild;//p 指向其右孩子
        }
    }
    return max;
}
```

方法三：常规做法，利用层序遍历，进行计数

注意：这是层序的统一写法，基本上层序的题都能用这类代码修改即可
且，此代码中 last 的设置，两种不同的循环队列中都可使用

```
int getHighth2(BTree root){
    if(root==NULL) return 0;
    Queue Q;InitQueue(Q);//创建队列，并初始化
    EnQueue(Q,root);//根节点入队
    int level=0,last = Q.rear;//level 记录当前层的最右边节点下标
    BTNode *p=NULL;
    while(!IsEmpty(Q)){
        //根据当前层，将下一层的结点入队
        while (!IsEmpty(Q)&&Q.front!=last)
        {
            DeQueue(Q,p);
            if(p->lchild) EnQueue(Q,p->lchild);
            if(p->rchild) EnQueue(Q,p->rchild);
        }
        level++;//记录出队元素的所在，层数
        last = Q.rear;//更新，当前层的最右边节点下标 level
    }
    return level;//获得层数，放回结果
}
```

6、根据先序和中序还原二叉树

前提：没有重复节点，如有重复节点，则可以进行数据的 Id 设置，以 Id 左右每一个数据的唯一标识

```
BTNode* recoverTree(int pre[],int preStart,int preEnd,
                    int in[] ,int inStart,int inEnd){

    if(preStart>preEnd||inStart>inEnd) return NULL;

    //查找根节点在中序序列中的位置，以便进行左右子树的划分
    int index = inStart;
    while(index<=inEnd&&in[index]!=pre[preStart]) ++index;
    //中序中，
    //从 inStart,到 index-1 都是左子树，左子树有，index-inStart 个节点，记为 L
    // 从 index+1,到 inEnd 都是右子树，右子树有，inEnd-index 个节点，记为 R
    //先序中，从 preStart+1 到 L+preStart 属于左子树
    //       从 preEnd-R+1 到 preEnd 属于右子树

    int L=index-inStart,R = inEnd-index;

    //创建根节点，进行左右子树根节点的连接
    BTNode* root = (BTNode *)malloc(sizeof(BTNode));
    root->data = pre[preStart];
    root->lchild =recoverTree(pre,preStart+1,L+preStart,in,inStart,index-1);
    root->rchild = recoverTree(pre,preEnd-R+1,preEnd,in,index+1,inEnd);
    //返回 root 根节点
    return root;
}
```

7、非递归遍历二叉树专题：

/**先序非递归遍历*/

天勤版本：

```
void preorderNorecursion(BTNode *bt){
    BTNode *stack[maxSize],*p;
    int top = -1;
```

```
    stack[++top] = bt;
    while(top != -1){
       p = stack[top--];
       printData(p->data);
       s1[++n1]=p->data;
       if(p->rchild != NULL) stack[++top] = p->rchild;
       if(p->lchild != NULL) stack[++top] = p->lchild;
    }
}
```

王道版本：

```
void preorderNorecursion(BTNode *bt){
    BTNode* s[maxSize];
    int top = -1;
    BTNode *p=bt;
    while(p||top!=-1){
       if(p){
           visit(p);
           s[++top] = p;
           p = p->lchild;
       }
       else{
           p = s[top--];
           p = p->rchild;
       }
    }
}
```

/**中序非递归遍历*/

天勤版本：

```
void inorderNorecursion(BTNode *bt){
    BTNode *stack[maxSize],*p;
    int top = -1;
    p = bt;
    while(p!=NULL || top != -1){
       while(p!=NULL){
           stack[++top] = p;
           p = p->lchild;
       }
       if(top != -1){
```

```
            p = stack[top--];
            printData(p->data);
            s2[++n2] = p->data;
            p = p->rchild;
        }
    }
}
```

王道版本：

```
void ineorderNorecursion(BTNode *bt){
    BTNode* s[maxSize];
    int top = -1;
    BTNode *p=bt;
    while(p||top!=-1){
        if(p){

            s[++top] = p;
            p = p->lchild;
        }
        else{
            p = s[top--];
            visit(p);
            p = p->rchild;
        }
    }
}
```

/**后序非递归遍历*/

王道版本：（天勤版本作用不大，因为无法获取当前遍历节点 p 的所有父节点）

```
void postorderNorecursion(BTree root){
    Stack s;
    InitStack(s);
    BTNode *r = NULL,*p = root;
    while(p||!IsEmpty(s)){
        if(p){
            Push(s,p);
            p = p->lchild;
        }
        else{
            p = getTop(s);
```

```
            if(p->rchild&&p->rchild!=r){
                p = p->rchild;
                Push(s,p);
                p = p->lchild;
            }
            else{
                Pop(s,p);
                visit(p);//此处可用于进行非递归后序遍历的特性应用
                r = p;
                p = NULL;//此处是易错点
            }
        }
    }
}
```

总结：非递归全部采用王道，便于记忆。
总体流程：有左孩子的，则左孩子进栈，没左孩子那就向右走一步，然后循环即可。
其中，后续非递归则只能在右孩子被访问或者右孩子为空过后，才能将栈顶元素弹出，进行访问
且，访问过后将更新（最新访问指针）r,和将（访问指针）p 置空

后续递归查找最近公共祖先

```
//如果 root 包含 p,返回 p
//如果 root 包含 q，返回 q
//如果 root 都不包含，返回 NULL
//如果 root 都包含 p 和 q，返回 root（最近公共祖先）
BTNode *getMiniAncestor(BTNode *root,BTNode *p,BTNode *q){

    if(!root||root==p||root==q) return root;

    BTNode *left = getMiniAncestor(root->lchild,p,q);
    BTNode *right = getMiniAncestor(root->rchild,p,q);

    if(!left) return right;
    if(!right) return left;
    return root;
}
```

8，编写删除二叉树中，以 x 为根的子树

此处，主要强调的是引用符&的用法，引用父节点的指针域，将指针域置空，来达到将节点删除的作用。
采用后序遍历的原因：先序、中序在进行删除时，会出现空指针。
解决方案：添加 return 即可

```
void deleteTreeX(BTNode* &p,int x){
    if(p){
       deleteTreeX(p->lchild,x);
       deleteTreeX(p->rchild,x);
       if(p->data==x) {
           free(p);
           p = NULL;
       }
    }
}
```

常规做法：利用层序遍历，一层一层的检测队列结点，考察其子结点的值是否为 x，若是，则将其所指向的指针域置空，达到删除的目的。
代码：只需简单修改，计算树高的代码层序，即可。

9、表达式树的特点和搭建

特点：表达式树的先、中、后续遍历就是表达式的前缀、中缀、后缀表达式。
搭建算法：和中缀计算类似（中缀计算与中转后一致），只是把压入的计算结果，改成压入以操作符为根节点的二叉树，即是，只压入根节点即可。

其中，王道中的 408 真题（P151-20），不适用于与该方法。原因：出现了（-b）这样的表达式，不是二叉树，所以，在进行子二叉树的构造时会出现问题。
解决方案：中转后缀，再利用，中序、后序可以唯一确定一棵二叉树，构造二叉树。同时，由于结点中存在大量的+，-，*，/的重复结点，所以需要给每个结点设置唯一标识的 Id 值，用于查找。
其中，后缀表达式计算：从前往后，遇数入栈，遇算数符号弹出两个，先谈为右，后弹为左，计算值入栈。
前缀表达式计算：与后缀方向相反（扫描方向和算式位置）

表达式树，转中缀表达式。其中，deep 用于去除最外层多余的括号

```
void treeToinfix(BTNode *p,int deep){
    if(p){
        if(p->lchild==NULL&&p->rchild==NULL){//如果是叶子结点，直接输出
            printData(p->data);//打印操作数
        }
        else{//非叶子结点，采用中序遍历，并将“()”添加进去
```

```
            if(deep>1) cout<<"(";
            treeToinfix(p->lchild,deep+1);
            printData(p->data);//打印操作符
            treeToinfix(p->rchild,deep+1);
            if(deep>1) cout<<")";
        }
    }
}
```

10、删除叶子结点

思想：采用层序遍历，逐层检查是否为叶子结点，如果是则进行删除
但是，注意：进行删除只能在其父节点处进行，且为了代码简洁，对删除和判定进行封装
设计思路：对二叉树进行层序遍历，逐层检查每一个结点(记为 p)是否为叶子结点。如果 p 是叶子结点，则删除。否则，继续检查。

```
void destroyNode(BTNode *&p){
    free(p);
    p = NULL;
}
bool isLeaf(BTNode *p){
  if(p&&!p->lchild&&!p->rchild)
    return true;
  return false;
}
void deleteLeaf(BTree &root){
//对 root 的可能情况进行判定
  if(root==NULL) return;
  if(isLeaf(root)) { destroyNode(root); return;}

//开始层序遍历
  BTNode *p = root;
  Queue Q; InitQueue(Q);
  EnQueue(Q,p);
  int last = Q.rear;//last 当前层中最右侧元素下标标记
  while( !IsEmpty(Q) ){
    while(!IsEmpty(Q)&&Q.front!=last){
      DeQueue(Q,p);    //p 结点出队，对其孩子结点进行判定
      if(isLeaf(p->lchild)) destroyNode(p->lchild);  //左孩子为叶子结点，删除左孩子
      if(isLeaf(p->rchild)) destroyNode(p->rchild);
```

```
        //继续进行层序遍历
        if(p->lchild!=NULL) EnQueue(Q,p->lchild);
        if(p->rchild!=NULL) EnQueue(Q,p->rchild);
      }
      last = Q.rear;      //调整 last
   }//while
}
```

方法二：使用递归先序遍历，删除叶子结点

```
void deleteLeaf2(BTree &root){
     if(root){
      if(!root->lchild&&!root->rchild){
            free(root);
            root = NULL;
            return ;
      }
      deleteLeaf2(root->lchild);
      deleteLeaf2(root->rchild);

     }
}
```

平衡二叉树和排序二叉树专题

平衡二叉树（手工）：平衡二叉树是任意子树和其自身的平衡因子的绝对值<1 的二叉排序树；
平衡因子=左子树高度 - 右子树高度
构造过程：就是四种调整过程
4 种调整过程：
LL：调整最小不平衡树的根结点，将其右转。
RR：调整最小不平衡树的根结点，将其左转。
LR：调整最小不平衡树的从根节点开始的 L 边和 R 边，先向左，后向右调整，与 LR 相反的顺序。
RL：与 LR 类似。

平衡二叉树判定：

思路分析，判断二叉树是否平衡，需要条件：1、左、右子树平衡；2、左右高度差的绝对值小于 2。所以对于一棵树的判定，需要获取到左、右子树的 balance 和高度之后，才能进行判定。符合后序遍历的特点（先判定子树，再判定根节点的平衡因子）。由于，要在函数内部进行左右子树的 balance、level 的赋值，所以采用引用号&。

先讨论简单情况：**bt** 为空、仅有一个结点

1）**root** 为空，则 **balance = 1,h = 0;**

2）仅有 **root**，则 **balance = 1,h = 1;**

3）否则，进行后序递归，分别获取到左、右子树的 **balance** 和 **h**。**root** 的高度 **h = max{ leftHigh , rightHigh } + 1**。如果左、右子树高度差的绝对值**<2**，且左右子树平衡则，**root** 的 **balance = 1**；否则，**balance = 0**。

```
void judgeBalance(BTNode *p,bool &balance,int &maxLevel){
                bool left=0,right=0;
                int leftHigh=0,rightHigh=0;
                if(!p){
                  maxLevel = 0;
                  balance = 1;
                }
                else{
                  if(!p->lchild&&!p->rchild){
                        maxLevel = 1;
                        balance = 1;
                  }
                  else{
                        judgeBalance(p->lchild,left,leftHigh);
                        judgeBalance(p->rchild,right,rightHigh);
                        maxLevel = (leftHigh>rightHigh?leftHigh:rightHigh)+1;
                        if(abs(leftHigh - rightHigh)<2)
                              balance = left&&right;
                        else balance = 0;
                  }
                }
}
```

最小平衡二叉树公式及常规计算：

公式：$A_0 = 0 , A_1 = 1 , A_{n+1} = A_n+A_{n-1}+1$

排序二叉树（代码实现）

查找：

```
BTNode* findOrderTree(BTree root,int k){
        BTNode *p = root;
        while(p&&p->data!=k){
```

```
            if(p->data<k) p = p->rchild;
            else p = p->lchild;
          }
          return p;
}
```

排序二叉树插入(也是构造算法)：

```
bool insertOrder(BTree &root,int k){
   if(root == NULL){
                    root = (BTree)malloc(sizeof(BTNode));
                    root->data = k;
                    root->lchild = NULL;
                    root->rchild = NULL;
                    return true;
    }
    else if(root->data == k)
                      return false;
    else if(root->data < k)
                      return insertOrder(root->rchild,k);
    else
                      return insertOrder(root->lchild,k);
}
```

排序二叉树删除：

根据查找的递归算法，若未找到，则删除失败，返回 false；
若查找到，则获取到待删结点 del 的父结点指针域（这样，del=NULL 就可以从树中删除 del），然后将 del 所指向节点删除。
以 del 为根节点的二叉排序树，删除 del 结点，q 赋值为 del，共有以下几种情况：
1）del 是叶子结点，释放 del 指向的数据，del 置空（del = NULL）
2）del 仅有右孩子结点，将 del 赋值右孩子，后释放 del（q=del）
3）del 的左孩子 p 没有右孩子，将 p->rchild 赋值为 del->rchild，用 p 替换掉 del 在树中的位置。
4）其余情况（p 指向 del 左子树的最大元素，即最右侧结点），先将 p 从树中取出，再用 p 替换掉 del 在树中的位置

```
void deleteNode(BTree &del){
      if(!del->lchild&&!del->rchild){//删除叶子结点
            free(del);          //free 和置空不能调换位置
            del = NULL;
            return ;
```

```
    }
    BTNode *p = del->lchild,*pre = del,*q=del;

    while(p&&p->rchild) {pre = p;p = p->rchild;}

    if(!p) {      //del 没有左子树
        free(q);
        del = del->rchild;
    }
    else if(p==del->lchild){    //del 的左结点 没有 右孩子
        p->rchild = del->rchild;
        del = p;
        free(q);
    }
    else{
//在 del 的左子树中，p 指向左子树最大值（没有右孩子），pre 指向 p 的父节点

        pre->rchild = p->lchild;//接上 p 的左孩子
        p->lchild = del->lchild;//链接 del 的左右孩子，用 p 代替 del 节点的位置
        p->rchild = del->rchild;
        del = p; //替换掉 del（del 是 del 父结点的指针域）
        free(q);
    }
}

bool deleteOrderTree(BTree &root,int k){
    if(!root) return false;
    else if(root->data == k) {        //找到值为 k 的结点
        deleteNode(root);         //删除结点
        return true;
    }
    else if(root->data < k)  return deleteOrderTree(root->rchild,k);
    else return deleteOrderTree(root->lchild,k);
}
```

判定：采用先序遍历的思想，访问二叉排序树中的所有结点，判断当前结点 p 与其孩子结点的大小关系，如果不符合排序树规定，返回 false。如果符合，进行递归判定左右子树，返回左右子树的逻辑“与”的结果。

打印二叉树中，从叶子到根的，权值和为 K 的路径：先序遍历

```
void getPath(BTNode *p,int k,Stack &s,int sum){
   if(p){
      Push(s,p);
      sum += p->data;
      if(!p->lchild&&!p->rchild&&k==sum){
        printStack(s);//打印栈中元素，从栈底打印到栈顶
      }

      getPath(p->lchild,k,s,sum);
      getPath(p->rchild,k,s,sum);

      Pop(s,p);
      sum -= p->data;
   }
}
```

哈夫曼树的构造

思路：构造一个指针数组，指向数据域为权值的结点元素。数组中元素，根据权值降序排序。

1、每次选取权值最小的 2 个元素，组合成一个棵根节点权值为 a+b 的哈夫曼树

2、将根节点插入数组，并保证数组根据权值降序排列。

3、重复 1~2 过程，直至数组中只有一个元素

时间复杂度分析：降序排序采用快速排序(采用直接插入也可)，平均为 O(nlogn)

插入算法比较次数，平均为 $O(n^2)$

所以，总的时间复杂度为 $O(n^2)$

空间复杂度：使用了指针数组 O(n)

大顶堆和小顶堆的建立

核心算法--调整堆顶元素

```
void adjust(int A[],int r,int n){
        A[0] = A[r];
        int i=2*r;
        while(i<=n){
            if(i<n&&A[i]<A[i+1]) i++;

            if(A[0]>A[i]) break;
            else{
                A[r] = A[i];
                r = i;
            }
            i = 2*i;
        }//while

        A[i/2] = A[0];
}
void createMaxStack(int A[],int n){
        for(int i=n/2 ;i > 0 ; i--){
            adjust(A,i,n);
        }
}
```

树与森林专题

树和森林的遍历：分为先根遍历、后根遍历

首先，树的存储结构：孩子-兄弟的二叉链表形式，并查集形式，邻接链表形式

图专题：

1、广度优先

思想：广度优先搜素，和二叉树的层次遍历类似，采用的是队列，从 v 出发，依次访问，未被访问过的 v 的邻接顶点 w…，然后依次访问 w…的未被访问过的邻接顶点。
使用到图中常用函数：FirstNeighbor(G,v)，NextNeighbor(G,v,w)

邻接表的 FirstNeighbor、NextNeighbor

```
int FirstNeighbor(AGraph &G,int i){
    if(i<0||!G.adjList[i].firstarc) return -1;
    return G.adjList[i].firstarc->adjvex;
}

int NextNeighbor(AGraph &G,int x,int y){
    if(x<0||x>=G.n) return -1;
    ArcNode *p = G.adjList[x].firstarc;
    while(p&&p->adjvex!=y){
        p = p->nextarc;
    }
    if(!p||!p->nextarc) return -1;
    return p->nextarc->adjvex;
}
```

通用算法，不论是采用邻接表，或是，邻接矩阵

```
int visit[maxSize];
void BFS(AGraph G,int v){
    printVNode(G.adjList[v]); //访问顶点 v
    visit[v] = true;
    Queue Q;InitQueue(Q);   //初始化队列
    EnQueue(Q,v);           //v 入队
    while(!IsEmpty(Q)){
        DeQueue(Q,v);
        for(int w = FirstNeighbor(G,v);w>=0;w = NextNeighbor(G,v,w)){
            if (visit[w]!=1)
            {
                printVNode(G.adjList[w]);
                visit[w] = true;
                EnQueue(Q,w);
```

```
                }
            }
        }
    }

    void BFStravel(AGraph G){
        for(int i=0;i<G.n;++i){
            visit[i] = false;
        }
        for(int i=0;i<G.n;++i){
            if(visit[i]!=1){
                BFS(G,i);
            }
        }
    }
```

广度优先搜索的应用：获取顶点 **v** 到其余顶点的最短路径
核心思想，使用一个数组 **distance[]**，用于 **v** 到其余顶点的存储路径长度
初始化：**distance[v] = 0**； **visit[v] = true**； **distance[其余] = INF**；
路径长度更新：**distance[w] = distance[v]+1**；

注意：由于邻接表的存储形式不唯一，所以邻接表的广度优先生成树，不唯一。
而邻接矩阵的存储形式唯一，所以邻接矩阵的广度优先生成树唯一

2、深度优先

图的深度优先算法主要在于：下一个访问对象的选取。
对比二叉树的深度优先遍历（先、中、后序遍历），图没有孩子的概念，所以需要对其所有相邻的顶点，进行深度优先遍历。

```
void DFS(AGraph G,int v){
    printVNode(G.adjList[v]);
    visit[v] = true;
    for(int w = FirstNeighbor(G,v);w>=0;w = NextNeighbor(G,v,w))
        if(!visit[w]) DFS(G,w);
}

void DFStravel(AGraph G){
    for(int i=0;i<G.n;++i) visit[i] = false;
    for(int i=0;i<G.n;++i){
        if(!visit[i])
            DFS(G,i);
```

```
    }
}
```

关于广优和深优的算法分析：

广优先算法 **BFS** 分析：计算顶点入队次数，和搜索次数

对于邻接表，时间复杂度 **O(|V|+|E|)**。空间复杂度，每个顶点入队一次，最坏的情况 **O(|V|)**

对于邻接矩阵，时间复杂度 **$O(|V|^2)$**。空间复杂度，每个顶点入队一次，最坏的情况 **O(|V|)**

深度优先算法 **DFS** 分析：

借助系统栈，空间复杂度 **O(|V|)**。

时间复杂度，遍历的过程实质是对每一个顶点查找其邻接顶点的过程。

对于邻接表，查找时间复杂度为 **O(|E|)**，访问顶点时间复杂度为 **O(|V|)**，所以总的是时间复杂度 **O(|V|+|E|)**。

对于邻接矩阵，查找所有顶点的邻接顶点所需时间为 **O(|V|)**，总的时间复杂度 **$O(|V|^2)$**。

1、图的判连通分量：只需要进行深度或广度递归遍历，记录进行几次深度和广度递归遍历。

2、图的判断强连通分量：

3、判环问题，对于无向图来说，无环连通图就是树。

4、输出从 Vi 到 Vj 的所有简单路径

采用基于递归的深度优先遍历算法，从结点 **u** 出发，递归深度优先遍历图中结点，若访问到结点 **v**，则输出该搜索路径上的结点。为此，设置一个 **path** 数组来存放路径上的结点（初始为空），**d** 表示路径长度（初始为**-1**）。查找从顶点 **u** 到顶点 **v** 的简单路径过程如下（假设查找函数名为 **FindPath()**）

1）FindPath(G,u,v,path,d): d++ ;path[d] = u; 若找到 **u** 的未访问过的相邻结点 **u1**，则继续下去，否则置 **visit[u]=0** 并返回。

2）FindPath(G,u1,v,path,d): d++ ;path[d] = u1; 若找到 **u1** 的未访问过的相邻结点 **u2**，则继续下去，否则置 **visit[u1]=0** 并返回。

3）如此类推，继续上述过程，知道 **ui=v**，输出 **path**

```
void FindPath(AGraph G,int u,int v,int path[],int d){
    int w;
    ArcNode *p;
    d++;                //路径长度+1
    path[d] = u;        //将当前顶点添加到路径中
    visit[u] = true;    //置为已经访问
```

```
    if(u==v) print(path);  //找到一条路径则输出
    p = G.adjList[u].firstarc;   //p 指向 u 的第一个相邻结点
    while(p){
        w = p->adjvex;
        if(!visit[w]) FindPath(G,w,v,path,d);     //若顶点 w 未访问，递归访问它
        p = p->nextarc;  //p 指向 u 的下一个相邻顶点
    }
    visit[u] = false;   //回复环境，使该顶点可重新使用
}
```

5、最小生成树的 Prim、Kruskal 算法，以及最小生成树唯一的条件

生成树唯一条件：所有权值均不等，或者有相等的边，但是在构造最小生成树的过程中权值相等的边都被并入生成树中，其最小生成树唯一。

Prim 算法：

设有两个集合 S 和 T，集合 S 存放未被并入最小生成树的顶点，集合 T 存放已经被并入最小生成树的顶点。

引入数组 lowcost、tree 和 visit。

lowcost[ui]用于保存构造最小生成树 T 过程中，还未并入树 T 中的顶点 ui，与树 T 的最短距离。初态：初始只有根结点 v0，若 v0 到 ui 有边，则 lowcost[ui] = v0->ui 的权值，否则 lowcost[ui]=无穷

tree[ui]采用双亲表示法，保存最小生成树。tree[ui]表示顶点 ui 的双亲结点 p 的顶点。初态：仅有根结点 v0，所以 tree[ui]全部置为-1；

visit[]为标记数组，visit[vi] = 0 表示 vi 在集合 S 中，还未并入最小生成树中。visit[vi]=1 表示 vi 已经在 T 中，即已经并入最小生成树中。初态，T 中仅有 v0，所以 visit[v0] = 1，visit[vi]=0;

构造最小生成树过程如下，

1）。。。

```
void Prim(MGraph G,int v0){
    int lowcost[maxSize],sum =0,tree[maxSize];
    for(int i=0;i<G.n;++i){
        visit[i] = false;
        lowcost[i] = G.edge[v0][i];
        if(g.edge[v][i]<INF){
            tree[i] = v;
        }
        else{
            tree[i] = -1;
        }
    }
    visit[v0] = true;
    tree[v0] = -1;
```

```
    int min,k,v=v0;
    for(int i=0;i<G.n-1;++i){
        min = INF;
        for(int j=0;j<G.n;++j){      //查找当前生成树到剩余顶点中，最短的一条边
            if(!visit[j]&&min>lowcost[j]){
                min = lowcost[j];
                k = j;
            }
        }
        visit[k] = true;//此处注意不要添加 tree[v] = k;因为初始条件已经满足了
        sum += min;
        for(int l=0;l<G.n;++l){ //以新添加的顶点 k 为媒介，更新候选边
            if(!visit[l]&&lowcost[l]>G.edge[k][l]){
                lowcost[l] = G.edge[k][l];
                tree[l] = k;
            }
        }
    }//for
    //输出结果
    cout<<sum<<endl;
    for(int i=0;i<G.n;++i) cout<<tree[i]<<" ";
}
```

算法二：并查集，排序，Road 数组

```
void Krustral(MGraph G,int path[]){
    Road road[maxSize];
    int size = G.n*(G.n-1),sum = 0;
    getRoad(G,road,size);
    sort(road,size);
    for(int i=0;i<G.n;++i) path[i] = -1;

    for(int i=0;i<size;++i){
        Road temp = road[i];
        int a = getRoot(path,temp.a);
        int b = getRoot(path,temp.b);
        if(a!=b){
            path[temp.b] = temp.a;
            sum += temp.info;
        }
    }
    cout<<sum<<endl;
```

```
    for(int i=0;i<G.n;++i){
        cout<<path[i]<<" ";
    }
}
```

6、单源最短路径

Dijkstra（迪杰斯特拉算法） 和 **Prim** 算法类似，只不过修改了

将 if(!visit[l]&&lowcost[l]>G.edge[v][l]){进行修改为 **lowcost[v]+G.edge[v][l]<lowcost[l]**

```
        lowcost[l] = G.edge[v][l];  //lowcost[l] = G.edge[v][l] + lowcost[v];
        tree[l] = v;
    }
```

打印 **tree** 中从结点 **a** 到 **v0** 的算法

```
void printPathForDj(int path[],int a){
    if(path[a]==-1) cout<<a;
    else{
        printPathForDj(path,path[a]);
        cout<<"->"<<a;
    }
}
```

弗洛伊德算法：求任意一对顶点间的最短路径

```
void Floyd(MGraph &g,int path[][maxSize]){
    int i,j,k;
    int A[maxSize][maxSize];
    for(i = 0;i<g.n;++i){
        for(j=0;j<g.n;++j){
            A[i][j] = g.edge[i][j];
            path[i][j] = -1;
        }
    }
    for (k = 0;k<g.n;k++){
        for(i=0;i<g.n;i++){
            for(j=0;j<g.n;j++){
                if(A[i][j]>A[i][k]+A[k][j]){
                    A[i][j] = A[i][k]+A[k][j];
                    path[i][j] = k;
                }

            }
        }
```

```
    }
}
```

Floyd 路径打印算法：

```
int startFlag = 0;
void printPath(int a,int b,int path[][maxSize]){
    if(path[a][b] ==-1){
        if(startFlag == 0){
            printData(a);
            startFlag = 1;
        }
        printf("->");
        printData(b);
    }
    else{
        int mid = path[a][b];
        printPath(a,mid,path);
        printPath(mid,b,path);

    }
}
```

无向图成环判定

```
bool judgeCircuit(MGraph G,int pre,int p){
    visit[p] = true;
    for(int i=0;i<G.n;++i){
        if(G.edge[p][i]){
            if(!visit[i]) return judgeCircuit(G,p,i);
            else if( i !=pre) return false;
        }
    }
    return true;
}
```

AOV 网--拓扑排序，用于判断有向图成环问题。

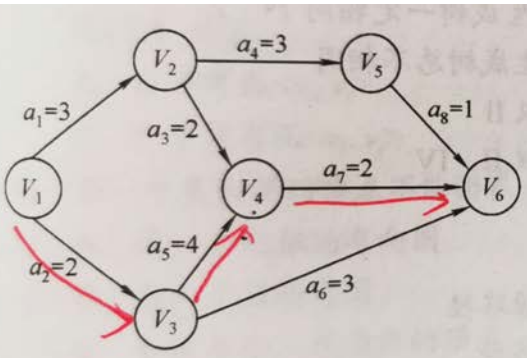
```
void TopSort(AGraph &G){
    Stack s;InitStack(s);
    for(int i=0;i<G.n;++i){
        visit[i] = false;
        if(G.adjList[i].in_degree==0) Push(s,i);
    }
```

```
    int k,count=0;
    while(!IsEmpty(s)){
        Pop(s,k);
        cout<<G.adjList[k].data<<" ";//visit(k)
        ++count;     //输出计数，用于判断是否成环

        visit[k] = true;

        ArcNode *p =G.adjList[k].firstarc;
        while(p){
            if(!visit[p->adjvex]){
                int &w = G.adjList[p->adjvex].in_degree;
                --w;
                if(w==0) Push(s,p->adjvex);
            }
            p = p->nextarc;
        }
    }
    if(count!=G.n) cout<<endl<<"成环图";
    else cout<<endl<<"非成环图";
}
```

AOE 网的计算

第一步：计算顶点 v（事件 v）的最早发生时间 ve，最晚发生时间 vl
计算 ve 是从前往后推，所有的时间取靠后的时间
计算 vl 是从后往前推，所有的时间取靠前的时间
注意：若出现 ve>vl，则肯定是计算有问题

第二步：计算，活动 a 的最早 e 开始、最晚 l 开始时间。
e 的计算：活动 a 弧的起点顶点 v 的最早发生时间 ve = e
l 的计算：活动 a 弧的终点顶点 v 的最晚发生时间 l = vl - 活动时长 a
第三步：观察，如果活动 a 的最早开始 e=最晚开始 l，说明活动是关键活动



	v_1	v_2	v_3	v_4	v_5	v_6
$ve(i)$	0	3	2	6	6	8
$vl(i)$	0	4	2	6	7	8

	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
$e(i)$	0	0	3	3	2	2	6	6
$l(i)$	1	0	4	4	2	5	6	7
$l(i)-e(i)$	1	0	1	1	0	3	0	1

图 6.23　求解关键路径的过程

注意：

关键路径不唯一，只提高一条关键路径上的活动速度，并不会缩短整个工期。

缩短关键活动，可能会使得关键活动变成非关键活动

查找和排序专题：

分类：顺序查找、分块查找、折半查找

顺序查找的重点在于，引入了“哨兵”的概念

关键字：数据元素中唯一标识该元素的某个数据项的值

平均查找长度：查找成功和查找失败。查找成功：一般认为权重为 1/n，其中 n 为数据元素个数。查找失败，1/m 为失败，查找失败点的个数

折半查找：就是二分查找。判定树，是一棵二叉树平衡树

值得注意的是：折半查找的 mid 可以向下取整，也可以向上取整。但二者判定树不同。

平均查找长度≈log(n+1)-1，判定树的树高 h=log(n+1)向上取整

分块查找：将查找表分为若干子块，块内元素可以有序，也可以无序，但是块间是有序的。

B 树的定义，平衡多路，平衡因子恒为 0（虽然考纲中未提到有关 B 树的考查）：
m 阶 B 树：
1、树中每个结点至多有 m 棵子树，且至多有 m-1 个关键字
2、根结点至少有 2 棵子树
3、非根、非叶子结点至少有 m/2 向上棵子树，m/2 向上-1<=关键字数<=m-1
4、每个结点关键字保持递增有序
5、所有叶子结点在同一层次且不带信息（即为 NULL）
注意：此处叶子结点是 NULL 而不是树最底层带关键字的结点

散列存储：建立关键字和存储地址之间的直接映射关系
注意：处理好同义词之间的冲突
常用的散列函数：
1、直接定址 H(key) = a*key+b 适用于关键字连续
2、除留余数法：表长为 m，（而不是数据长度），取 p=max{质数≤m}

$$H(key) = key\%p$$

3、数字分析法：对关键字进行分析，一般选取 r 进制数上各位数上不同的作为区分，尽量使得分布均匀。前提：已知关键字
4、平方取中间几位

冲突解决方法：
1、开放定址法：$H_1(key) = (H(key)+d)\%m$，m 为表长。可以探测到 p 之外的区域
d 的选取：
1）线性探测法 ：d= 1~m-1 的常数。优点：可以查遍全表。问题：会出现堆积现象。
2）平方探测法：d= 1^2，-1^2，2^2，-2^2 ....≤m/2，又称为二次探测法，可以避免堆积现象，但是不能探测到所有单元，但是至少可以探测到一半单元
3）再散列法：d= $i \times H_2(key)$，其中 i 是冲突次数，i 初始为 0。最多经过 m-1 次探测。

开放定址法：在进行删除时，若直接删除会截断其他具有相同散列地址的元素的查找地址。所以，应当进行逻辑删除。逻辑删除的坏处：执行多次删除后，散列表表面看起来很满，但是实际上有许多位置未被利用，所以要进行定期维护，把逻辑删除的元素进行物理删除。

2、链接法：把所有的同义词，存储在相应的线性链表。

平均查找长度：衡量散列表的查找效率
散列表的查找效率取决于三个因素：散列函数、处理冲突的方法、装填因子
装填因子：α=表中记录数没 n/散列表长度 m。装填因子越大，发上冲突的可能性越大。所以，取 m=n/α 向上取整

排序专题：
分类：插入排序、交换排序、选择排序、归并排序、基排
插入排序：直接插入、折半插入、希尔插入
交换排序：冒泡排序、快速排序
选择排序：简单选择排序、堆排序

直接插入排序算法：

```
insertSort(int a[],int n){
    int i,j;
    int temp;
    for(i=1;i<n;++i){
        temp = a[i];
        for(j=i;j>0&&a[j-1]>temp;--j){
            a[j]=a[j-1];
        }
        a[j]=temp;
    }
}
```

折半插入算法：
注意：折半插入只是减少了比较次数，且该比较次数和初始状态无关，但是，元素移动次数并未改变。
同时，查找过程中和二分查找不同的是，不需要将 **a[mid]==temp** 进行返回，总之，只要找到 **low>high** 的位置即可，且，**low** 的位置即是 **temp** 的插入位置，从 **low** 指向的位置开始到 **i** 的位置，全部后移动一个位置，并将 **temp** 赋值给 **a[low]** 即可。

```
void midSort(int a[],int n){
    for(int i=1;i<n;++i){
        int low = 0,high = i-1,temp =a[i];
        while(low<=high){
            int mid = (low+high)/2;
            if(a[mid]>temp) high = mid-1;
            else low = mid+1;
        }
        for(int k=i;k>low;--k){
            a[k] = a[k-1];
        }
    //注意，将 a[i]的值进行保存，否则上一个 for 循环，会将 a[i]覆盖
        a[low] = temp;
    }
```

```
}
```

希尔排序：

希尔排序的思想，是将数组分割成若干个形如 L[i,i+d,i+2d,...]的特殊子表，然后对个子表进行直接插入排序，当整个表中元素呈现“基本有序”，再对全体记录进行一次直接插入排序。

其中，步长的选取为 d1>d2>d3>...>dk=1。一般选取为 $d_{k-1}= d_k/2$，d1=n/2

```
void shell(int a[],int n){
    int d = n/2;
    for(;d>0;d/=2){
        for(int i=d;i<n;++i){
            int temp = a[i],k=i;
            for(;k>=d&&a[k-d]>temp;k-=d){
                a[k] = a[k-d];
            }
            a[k] = temp;
        }
    }
}
```

注意：根据希尔排序的思想，若 最后一次排序 d=2，则会出现 dk=0，而不进行 dk=1 的直接插入排序，这就可能使得最后的结果只能是接近于有序。但是，经过近 1000 次的排序机试，得出的结果是符合有序的要求。所以，不进行单独的 d=1 的直接插入排序。原因：等进一步深入学习再分析

数据结构版本(考试采用此版本)：

```
void Shell_Insert(int a[],int n,int d){
    for(int i=d;i<n;++i){
        int temp = a[i],k=i;
        for(;k>=d&&a[k-d]>temp;k-=d){
            a[k] = a[k-d];
        }
        a[k] = temp;
    }
}
void shellSort(int a[],int n,int d[],int m){
    for(int i=0;i<m;++i)
        Shell_Insert(a,n,d[i]);
}
```

交换排序：冒泡、快排

快速排序链表形式，代码测试过，带头节点

```
LinkedNode *partion(LinkedNode* start, LinkedNode *end){
    LinkedNode *pivotPost=start->next;
    Data data = pivotPost->data;
    LinkedNode *p = start->next,*pre = start,*temp;
    while (p!=end){
        if(p->data < data){
            //如果当前节点(p)的值小于中轴值，进行节点移动操作
            //temp 保存 p 的后继节点，方便进行下一次比较

            temp = p->next;

            //采用头插法，将小于 pivot 的节点移动至链表前端
            //开始进行节点的移动

            pre->next = p->next;
            p->next = start->next;
            start->next = p;
            p = temp;
            //开始下一次比较，pre 指向的仍然是 p 的前驱节点，     //pre
        不需要移动
        }
        else{  //否则，继续遍历
            pre = p;
            p = p->next;
        }
    }//while
    return pivotPost;
}

//关于调用传参数，head：链表的头节点，rear：NULL 即可
void quickSortLinked(LinkedNode *head,LinkedNode *rear){
    if(head->next!=rear){
        LinkedNode *pivot = partion(head,rear);
        quickSortLinked(head,pivot);
        quickSortLinked(pivot,rear);
    }
}
```

冒泡的链表形式，代码测试过

```
void bubbleSortList(List head){
    LinkedNode *front,*p,*last,*q,*rear;

    //front:指向遍历指针 p 的前驱节点；
    //p:遍历指针；q:指向 p 的后继节点
    //last:冒泡的尾端节点，冒泡排序过程中，有序的第一个节点；
    //rear：q 的后继节点，主要用于方便指针运算

    last = NULL; //初始有序序列为空
    while(head->next!=last){
        //链表的首节点不是（冒泡的）有序序列的第一个节点
        front = head;p = head->next;//从首节点开始冒泡排序
        while(p->next!=last){
            q = p->next;
            if(p->data>q->data){
                //如果当前遍历(p)的节点比后继节点(q)大
                //开始进行位置交换前，先用 rear 保存 q 的后继

                rear = q->next;
                p->next = rear;//开始进行 p,q 位置的互换
                front->next = q;
                q->next = p;
                front = q;//互换结束后，进行 front 的前移，
                    //此时 p 指向的是交换后，后一个位置
                        //即为下一个要遍历的位置，所以,p 不进行移动
            }
            else{               //当前遍历(p)的节点比后续节点(q)小
                front = p; //front 和 p 进行前移
                p = q;
            }

        } //while
        last = p;

        //使用 last，保存一趟冒泡排序后，有序序列的第一个节点

    } //while
}
```

快排和冒泡的顺序存储形式：

冒泡：

```
void bubbleSort(int a[],int n){
    int i,j,temp;
    for(i=0;i<n-1;++i){
        bool flag = true;
        for(j=0;j<n-i-1;++j){
            if (a[j]>a[j+1])
            {
                temp = a[j];
                a[j] = a[j+1];
                a[j+1] = temp;
                flag = false;
            }
        }
        if(flag) return;
    }
}
```

快排的基本思路和链式的一致。

简单选择也比较简单。

归并排序(伪代码)：

```
ElemType *B = (ElemType*)malloc((n+1)*sizeof(ElemType));
void Merge(ElemType A[],int low ,int mid ,int high){
    //将 A 中数据复制到 B 中
    for(int k=low;k<=high;)  B[k++] = A[k++];
    //比较 B 的左右两段中的元素，将较小值复制到 A 中
    for(int i=low,j=mid+1,k=i;i<=mid&&j<=high;++k){
        if(B[i]<=B[j]) A[k]=B[i++];
        else A[k] = B[j++];
    }
    //若一个表中未检测完，复制
    while(i<=mid) A[k++] = B[i++];
    while(j<=high) A[k++] = B[j++];
}

void MergeSort(ElemType A[],int low,int high){
    if(low<high){
        int mid = (low+high)/2;
        MergeSort(A,low,mid);
```

```
        MergeSort(A,mid+1,high);
        Merge(A,low,mid,high);
}
```

基排序：
有两种方法：高位优先和低位优先。
从小到大，则是低位优先，先排序低位的关键字，存储空间为 r 个队列（先进先出原则）。

定义：

- 线性表的定义：线性表是具有相同数据类型的 **n(n≥0)**个数据元素的有序序列，其中 **n** 为表长，当 **n=0** 时，线性表是空表。

特点：

1、除第一个（表头元素）和最后一个元素（表尾元素）外，线性表内元素都有且仅有一个直接前驱和直接后继

2、第一个元素只有直接后继，没有前驱。最后一个元素只有直接前驱，没有后继

3、表中元素个数有限，逻辑上具有顺序性

4、表中元素都是数据元素，且数据类型相同

- 二叉树的定义：

二叉树是 **n(n≥0)**个结点的有限集合：**1**，或者为空二叉树，即 **n=0;2**，或者由一个根节点和两个互不相交的被称为根的左子树和右子树组成。左子树和右子树又分别是一个二叉树。

注意，二叉树和度为 **2** 的有序树的区别：

1、度为 **2** 的树，至少有 **3** 个节点，而二叉树可以为空

2、度为 **2** 的有序树的孩子左右次序是相对于另一个孩子而言，若只有一个孩子，则无所谓左右。而二叉树无论孩子数是否为 **2**，须确定其左右次序。

- 树的定义与术语：

定义：树是 **n** 个结点的有限集。当 **n=0** 时，是空树。在任意一棵非空树中应满足，**1**，有且仅有一个特定的称为根的结点；**2**，当 **n>1** 时，其余结点可分为 **m(m>0)**个互不相交的有限集合 T_1、T_2、…、T_m，其中每个集合本身又是一棵树，并且称为根的子树。

术语：

结点的度：结点的孩子个数

结点的层次：从树的根结点开始，根节点为第 **1** 层，它的子结点为第二层，以此类推。

结点深度：从根节点开始自顶向下的逐层累加

结点高度：从叶结点开始自底下向上逐层累加

有序树和无序树：树中结点的各子树从左到右是有次序的，不能互换，称为有序树，否则称为无序树。

路径：结点之间的路径是由两个结点之间所经过的结点序列构成的。

路径长度：路径上的所经过的边的个数。

结点带权路径长度=路径长度*结点权值，树的带权路径=所有结点的带权路径

概念：

- 图的基本概念

图：

图由顶点集合和边集合组成，记为 G=(V,E)，其中 V(G)表示图 G 中顶点的有限非空集合；E(G)表示 V(G)中顶点之间的关系集合。注意：顶点集合不能是空集。

有向图：有向边构成的边集。无向图：无向边构成边集

简单图：不存在重复边，不存在顶点到自身的边

多重图：和简单图概念相反

完全图：任意两个顶点之间都存在边。

子图：若边集 E 和顶点集 V 都属于 G，同时 E 和 V 构成图 G'，则称 G'为 G 的子图

生成子图：若 G' 包含了所有的顶点，则称 G'为 G 的生成子图

无向图的概念：连通图，连通分量

连通图：无向图中，任意两个顶点都是连通的

连通分量：无向图中，极大连通子图就是连通分量。极大连通子图，要求包含所有的边，而不是只要连通就可以的最少的边。

有向图概念：强连通图，强连通分量

强连通：有向图中，从顶点 u 到顶点 v 有路径，从顶点 v 到顶点 u 也有路径，则称 u、v 强连通。

强连通图：有向图中，任意一对顶点都是强连通的

强连通分量：极大强连通子图

生成树：连通图的生成树是包含图中全部顶点的一个极小连通子图。

生成森林：连通分量的生成树构成了生成森林。

顶点的度：无向图，所有顶点度之和为边 e 的 2 倍；

有向图，单个顶点的度 = 入度+出度，整体树的入度= 出度 = e 边

重要结论：无向图，只要边数大于 n-1 就一定有环，而有向图不一定。

- 查找的基本概念
- 排序的基本概念
- 数据结构的基本概念

ADT 的写法

栈和队列的基本概念和基本操作的设计

结构：

- 线性表的顺序存储结构和链式存储结构实现
- 栈和队列的顺序存储结构和链式存储结构实现

- 二叉树的顺序存储结构和链式存储结构实现
- 树的顺序存储结构和链式存储结构实现

- 图的存储及基本操作

  邻接矩阵法

  结构体：

```
typedef struct {
    Data data;//数据
    int no;//顶点在数组中的下标
}VertexType;

typedef struct {
    int edge[maxSize][maxSize];
    int n,e;
    VertexType vex[maxSize];
}MGraph;
```

  邻接表法

  结构体：

```
typedef struct ArcNode{
    int adjvex;
    struct ArcNode *nextarc;
    int info;//边的权值
}ArcNode;

typedef struct {
    Data data;
    ArcNode *firstarc;
}VNode;

typedef struct {
    VNode adjList[maxSize];
    int n,e;
}AGraph;
```

应用：

- 线性表的应用
- 栈和队列的应用
- 二叉树的遍历及应用
- 图的应用

1、拓扑排序
2、关键路径
3、 最短路径
4、最小（代价）生成树

- 查找算法的分析及应用
- 内排序算法的应用