

大家好，我是蓝蓝。

2023年尝试做了5期面向考研的初试算法训练营，一共45天。到今天为止，一二期全部结束。采取的形式是早上发题，下午发题解+答疑的形式。不夸张地说，作为一个小小的博主，每次收到大家的反馈都好好高兴的，真真的觉得值得。这五期中，每一期45天，题+题解+答疑收费15.5，全网也不可能有这个性价比，加上现在我还把汇总的版本给大家，大家一定一定要加油哦。

#算法专题9分计划总结#

总共完成了41天的代码题，对栈的入栈出栈、队列的入队出队，和他们的结构体以及增删改查，在顺序存储和链式存储的情况下熟练掌握（队列的链式存储结构体比较复杂，需要考前记一记，其实就是分成了2个来写）。对于邻接矩阵，邻接链表的遍历以及查找操作都能比较熟练的在考场上写出来，对于他们的复杂度还需要特别记一记。在这40多天里，我基本都是闭卷自己独立写题，虽然很多题没能想到最优解，但是使用基本的操作也都能写出来，也算是完成了蓝蓝发起这个计划的初衷，在考场上稳定的得分。还有就是对二叉树的几个遍历也都能独立写出来（虽然层序遍历非递归的写起来比较麻烦，但是我也能用辅助队列写个大概，还有就是深度优先和广度优先的代码我还是不太熟悉，需要再看看），KMP算法只要自己熟悉流程即可，代码不用多加了解，还有就是要背一个快速排序的代码（考场上一时忘记用冒泡代替很稳）。感谢蓝蓝能够提供这个机会，让我把之前学的东西都融汇贯通一遍，写代码的思路比之前清晰很多，之前链表线性表的逆置一直都没什么思路，现在看到题差不多就知道要用什么方法来写了🤔

收起

为了方便大家后续的复习，我这段时间对所有内容进行了整合，并提供了默写本的版本。并不是掌握这些就够了，我也不是神仙，筛选出来都是非常基础的题目+历年的真题了。

今天陆陆续续会开展蛮多的训练营，数据结构的应用题训练营也到了第五期，收官了，看看我们第一期的评价。嘿嘿，看到这些我可得瑟了~

#应用题专项训练营总结# #应用题第二期#

在第二期开营之前，我结束数据结构很久了。有些知识遗忘很多，我抓住这次机会又复习了一遍ds的知识点，每次做题都带着痛点在回顾，看到这个知识点想到了什么题。

今天再次回顾，有些知识点竟然可以得到再一次的补充，是我没想到的。我原本只是按照书本学习不管知识正误的，在做了两天真题之后，发现有很多知识点可以去改正看法，加深了解和记忆。

学习过程：

1-4天学习了顺序表的相关-矩阵（考点：可结合图的邻接矩阵）、队列（循环队列的拓展、删除插入、判空判满）、栈（删除插入操作的关键步骤）；

5-9天学习了树-基本性质+结点与树高关系、区分k叉树与普通树之间的特殊性质、一些不同二叉树的性质（平衡、线索、特殊、红黑树）、树的存储结构、树的应用（哈夫曼树、编码、并查集）、树、森林与二叉树的转换；

10-11天学习了查找—折半查找!!（算法题中更为重要）、动态查找中—二叉排序树、平衡二叉树、红黑树的性质+插入删除过程!!；

12-13天学习了图—图的存储结构（邻接矩阵，邻接表）!!、遍历（BFS广度优先-层序-求最短路径问题、DFS深度-先序）!!、相关应用（求最小生成树的两种算法、Dijkstra算法求最短路径、带权有向图求关键路径）!!

最后感谢蓝总给予这个平台，我真的有学到东西。同时也感谢解答同学的耐心指导口
继续加油！

收起

写到这里，我不得不介绍下我维护的3000+计算机考研的圈子了，一句话：在这里你会觉得好心安。[欢迎加入到组织](#)，我的微信(lanlankaoyanshan02)



当然对于这份PDF，我也有做一些视频，不过太懒了吧，没有更新多少~~~

[蓝蓝的视频在这里](#)

一、C语言基础

- 1、数组
- 2、for循环
- 3、结构体
- 4、指针

二、结构体合集

- 1 线性表的结构体定义
 - 1.1 顺序表结构体定义
 - 1.2 链表结构体定义
- 2 栈和队列的结构体定义
 - 2.1 栈结构体定义
 - 2.2 队列结构体定义
- 3 串结构体定义
 - 3.1 定长顺序存储结构体定义
 - 3.2 堆分配结构体定义
 - 3.3 块存储结构体定义
- 4 树和二叉树
 - 4.1 链式存储结构体定义
 - 4.2 线索二叉树结构体定义
 - 4.3 哈夫曼树结构体定义
 - 4.4 树之双亲表示法结构体定义
 - 4.5 孩子兄弟表示法结构体定义
- 5 图
 - 5.1 邻接矩阵法结构体定义
 - 5.2 邻接表法结构体定义

三、基础算法题

- 1、顺序表删除最小值
- 2、顺序表的逆置
- 3、删除顺序表中所有值为x的数据
- 4、删除下标 $i \sim j$ ($i \leq j$) 的所有元素
- 5、小于表头放前面，大于放后面
- 6、有序表中删除值在 $s \sim t$ 中元素
- 7、有序表删除重复元素
- 8、将两个顺序表合并成新有序表
- 9、两个顺序表交换
- 10、二维数组的查找

- 11、判断括号是否匹配
- 12、折半查找
- 13、找出第k小的元素
- 14、冒泡排序
- 15、选择排序
- 16、快速排序
- 17、直接插入排序
- 18、10年真题数组循环左移
- 19、11年中位数
- 20、13年求主元素
- 21、18年数组最小正整数
- 22、20年三元组最小距离
- 22、尾插法建立链表
- 23、头插法建立链表
- 24、从尾到头输出结点
- 25、删除单链表L所有值为x的结点
- 26、带头结点单链表就地逆置
- 27、删除表中介于给定两值之间元素
- 28、09年链表倒数第k个结点
- 29、判断带头结点循环双链表是否对称
- 30、找出两个链表的公共结点
- 31、两个链表A和B，奇数存放在A，偶数存放在B
- 32、递增链表去掉数值相同的元素
- 33、删除一个最小值的高效算法
- 34、按递增输出单链表结点的数据元素
- 35、求二叉树的最大深度
- 36、递归先序遍历
- 37、递归中序
- 38、递归后序遍历
- 39、非递归层次遍历
- 40、二叉树所有叶子结点个数
- 41、判断二叉树是否为二叉排序树
- 42、判断二叉树是否平衡二叉树
- 43、判断二叉树是否为完全二叉树
- 44、输出先序遍历的第k个结点的值
- 45、找出二叉树中最大值的点
- 46、并查集基本操作
- 47、14真题求T的WPL算法
- 48、17年真题二叉树转中缀表达式
- 49、图的初始化
- 50、图的广度优先遍历
- 51、图的深度遍历
- 52、计算机顶点i的出度
- 53、计算机顶点i的入读
- 54、图用邻接表表示，设计一个算法，输出从顶点 V_i 到顶点 V_j 的所有简单路径
- 55、23真题图算法

一、C语言基础

1、数组

数组中的每一个数据叫做数组元素 `Element`，数组中的每个元素都有一个序号，这个序号从 `0` 开始，称为下标 `Index`，例如，`a[0]` 表示第 `0` 个元素，`a[3]` 表示第 `3` 个元素。数组是一个整体，它的内存是连续的，内存示意图

a[0]	a[1]	a[2]	a[3]
------	------	------	------

公众号：蓝蓝考研

定义数组

```
//整形数组
int a[4] = {1,2,3,4};    //数组长度为4，只能存放4个元素
//对元素赋值
a[0] = 10;
a[1] = 5; //后面咱们基本上都是循环访问哈
//当然，你们也可以这样子初始化
int a[10]={12, 19, 33 , 56, 66};
```

访问数组

大家要直到，定义数组，数组中的每一个元素都有一个序号，需要从 0 开始，a[0] 表示第 0 个元素

```
//公众号：蓝蓝考研
//算法训练第一期
#include<stdlib.h>
#include<stdio.h>
void main()
{
    int a[10] = { 10, 12, 12 , 13, 14 }; //定义数组
    printf("index:%d  value:%d\n", 0, a[0]); //输出第一个元素
    printf("index:%d  value:%d\n", 2, a[2]);
    printf("index:%d  value:%d\n", 4, a[4]);
    printf("index:%d  value:%d\n", 8, a[8]);
    system("pause");
}
/*
输出：
index:0  value:10
index:2  value:12
index:4  value:13
index:8  value:0
*/
```

2、for循环

C 语言中的 for 循环由关键字 for 和三个表达式组成，表达式与表达式之间用英文状态的分号 ; 隔开；

```
for(表达式1; 表达式2; 表达式3){
    语句块
}
```

- a.先执行 表达式 1 (用于对变量初始化操作，仅仅只会执行一次);
- b.再执行 表达式 2，如果 表达式 2 为真，则执行循环体，否则结束循环;
- c.执行完循环体后再执行 表达式 3 做自增 ++ / 自减 -- 操作;
- d.重复执行步骤 c 和 d，直到 表达式 2 的值为假，就结束循环;

3、结构体

当一组数据有多个不同类型的时候，用结构体的方式处理数据。

定义个学生的结构

```
struct Student{           //声明结构体
    char name[20];        //姓名
    int num;              //学号
    float score;          //成绩
};
```

访问结构体

```
struct Student stu1;      //定义结构体变量
stu1.num = 10;
```

使用typedef

```
typedef struct student
{ ..
....
}stu,*student
```

stu是声明变量的别名，比如stu z; 与 struct stu z; 等价。

student是声明指针的别名，比如student y; 与struct student *y;等价。

换句话说，stu 是 struct student 的别名，

student 是 struct student * 的别名。

4、指针

指针指向存放变量的值的地址。因此我们可以通过指针来修改变量的值。

如下图：p 里面存放 a 的地址

*p 代表取这个地址上的值

此时 *p的值 = a 的值，把 p 的值修改了说明把这个地址的值也修改了所以 a 的值也会修改

```
int main() {
    int a = 10;
    int* p = &a;
    *p += 5;
    cout << "*p = " << *p << endl;
    cout << " a= " << a << endl;
    return 0;
}
```

```

选择 Microsoft Visual Studio 调试控制台
*p = 15
a=15

F:\visual\Project2\x64\Debug\Project2.exe (进程 66712)已退出, 代码为 0。
按任意键关闭此窗口. . .

```

数组名是一种特殊的指针指向数组第一个元素的地址。指针可以做运算：

```

#include <iostream>

using namespace std;

int main()
{
    int a[5] = {1, 2, 3, 4, 5};

    for (int i = 0; i < 5; i ++ ) cout << *(a + i) << endl;

    return 0;
}

```

引用和指针类似，相当于给变量起了个别名。

什么意思呢？下面代码的

a相当于周树人

我用&符合给a取了个别名p

这个 p 就相当于鲁迅。

PS：C语言中用printf输出，C++用cout

```

int main() {
    /*
    我们把a(周树人)的值设为10;
    然后我们给a(周树人)取个别名p(就叫鲁迅好了)
    我们把p(鲁迅)的值+5，也相当于把a(周树人)的值+5
    鲁迅 = 15; 周树人也 = 15;
    */
    int a = 10;
    int& p = a;
    p += 5;
    cout << " p = " << p << endl;
    cout << " a = " << a << endl;
    return 0;
}

```

```

Microsoft Visual Studio 调试控制台
p = 15
a = 15

F:\visual\Project2\x64\Debug\Project2.exe (进程 8512)已退出, 代码为 0。
按任意键关闭此窗口. . .

```

二、结构体合集

1 线性表的结构体定义

1.1 顺序表结构体定义

这里包含静态和动态两种方式，大家要注意哈，通常的情况下，静态的时候用数组的方式，动态都会使用到开辟释放空间。

对于内存空间的申请，一定要写会，尽量去理解，实在理解不了，给我**每天写一遍，写个一周**。

静态顺序表

```
#define Max 50 //定义线性表最大长度
//顺序表定义
typedef struct
{
    int data[maxSize];
    int length; //实际数据的个数 是用来判断是否达到最大空间
}Sqlist;
//考试的时候可以用下面的这种定义 直接一个数组
int A[maxSize];
int n;
```

动态顺序表

```
typedef struct {
    ElmeType* data;
    int size; //用来记录表中数据的实际个数方便比较
    int capacity; //设置用来记录L的最大空间 若是达到最大空间则需要扩容
}SeqList;
```

关于申请空间，一定一定要会哈。

```
L.data=(ElemType*)malloc(sizeof(Elemtype)*InitSize)
```

1.2 链表结构体定义

每一个结点不仅要放数据域，还要有一个指针域。

单链表

```
typedef struct LNode{
    ElemType data;
    struct LNode *next;
}LNode,*LinkList; //这两个是等价的
```

双链表

```
typedef struct DNode{
    ElemType data;
    struct LNode *prior, *next;
}DNode,*DLinkList; //这两个是等价的
```

2 栈和队列的结构体定义

2.1 栈结构体定义

顺序栈

```
//顺序栈定义
typedef struct
{
    int data[maxSize];
    int top;//栈顶指针
}SqStack;
//假设元素为int型，可以直接使用下面方法进行定义并初始化
int stack[maxStack];
int top=-1;
```

动态栈

```
typedef struct{
    ElemType* data;
    int top;
    //int size;可以不需要size 因为top 就是指向最后的位置
    int capacity;//这个时候是需要容量的 方便扩容
}SqStack;
```

链栈

```
// 链栈的存储结构
typedef struct StackNode
{
    int data;
    struct StackNode *next;
}StackNode,*LinkStack;
```

2.2 队列结构体定义

顺序队列[公众号：蓝蓝考研]

```
//顺序队列定义
typedef struct
{
    int data[maxSize];
    int front;
    int rear;
}SqQueue;
```

链队


```
//链队定义
//1.队节点定义
typedef struct QNode
{
    int data;//数据域
    struct QNode *next;//指针域
}QNode;
//2.类型定义
typedef struct
{
    QNode *front;//队头指针
    QNode *rear;//队尾指针
}LQueue;
```

3 串结构体定义

3.1 定长顺序存储结构体定义

串的定长顺序存储好像没有什么结构上的要求要求 直接使用静态线性表的结构体定义即可

```
#define MAXLEN 255 //预定义最大串长为255

typedef struct{
    char ch[MAXLEN]; //每个分量存储一个字符
    int length; //串的实际长度
}SString;
```

3.2 堆分配结构体定义

```
typedef struct{
    char *ch; //按照串长分配存储区 ch指向串的基地址
    int length; //串的长度
}HString;
```

3.3 块存储结构体定义

```
//块链存储
typedef struct Chunk{
    char ch[3]; //这里我定义的是放三个值
    struct Chunk *next;
}Chunk;
```

//定义头尾指针的作用是 方便进行连接操作

//连接的时候别忘了处理第一个串尾的无效字符

```
typedef struct{
    Chunk *head,*tail; //串的头尾指针
    int curlen; //串的当前长度 (链表的节点数)
}LString;
```

4 树和二叉树

正如我们所熟知的 存储方式一般有两种，顺序存储或者链式存储，若是使用顺序存储，则一般二叉树为了反映二叉树中结点之间的逻辑关系，只能添加并不存在的空结点构造树像完全二叉树一样，每一个结点与完全二叉树上的结点对应，再存储到一维数组的相应分量中。这样有可能造成空间的极大浪费，所以这里我们使用链式存储，为了方便各种操作，链式存储中也分为了几种方式，这里就不多赘述了，用到一个写一个

4.1 链式存储结构体定义

```
//二叉树的存储结构，一个数据域，2个指针域
typedef struct BiTNode
{
    char data;
    struct BiTNode *lchild,*rchild;
}BiTNode,*BiTree;
```

4.2 线索二叉树结构体定义

```
typedef struct BiThrNode{
    struct BiThrNode* Lchild;
    int Ltag;
    Elemtype data;
    int Rtag;
    struct BiThrNode* Rchild;
}BiThrNode;
```

4.3 哈夫曼树结构体定义

```
typedef struct HNode
{
    char data; //数据,非叶节点为NULL
    double weight; //权重
    int parent; //双亲, -1表示没有双亲, 即根节点
    int lchild; //左孩子, 数组下标, -1表示无左孩子, 即叶节点
    int rchild; //右孩子
}Hnode;
```

4.4 树之双亲表示法结构体定义

```
typedef struct Snode{
    char data;
    int parent;
} PTNode;

typedef struct{
    PTNode tnode[MAX_SIZE]; // 存放树中所有结点
    int n; // 结点数
}
```

4.5 孩子兄弟表示法结构体定义

```
#define tree_size 100//宏定义树中结点的最大数量
#define TElemType int//宏定义树结构中数据类型
typedef struct PTNode{
    TElemType data;//树中结点的数据类型
    int parent;//结点的父结点在数组中的位置下标
}PTNode;
typedef struct {
    PTNode nodes[tree_size];//存放树中所有结点
    int r,n;//根的位置下标和结点数
}PTree;
```

5 图

5.1 邻接矩阵法结构体定义

```
//图的邻接矩阵定义
typedef struct
{
    int no;//顶点编号
    char info;
}VertexType;
typedef struct
{
    int edges[maxSize][maxSize];
    int n,e;//顶点个数和边个数
    VertexType vex[maxSize];//存放节点信息
}MGraph;
//上面的定义如果没记住，也要记住里面的元素，如n，e等含义
```

5.2 邻接表法结构体定义

```
//结点的定义
typedef char VertexType;
typedef int EdgeType;
#define MaxVex 100
typedef struct EdgeNode //边表结点
{
    int adjvex; //邻接点域，存储邻接顶点对应的下标
    EdgeType weight; //用于存储权值，对于非网图可以不需要
    struct EdgeNode *next; //链域，指向下一个邻接点
}EdgeNode;
typedef struct VertexNode //顶点表结点
{
    VertexType data; //顶点域，存储顶点信息
    EdgeNode *firstedge; //边表头指针
}VertexNode, AdjList[MaxVex];

typedef struct
{
    AdjList adjList;
    int numVertexes, numEdges;
```

```
//图中当前顶点数和边数
```

三、基础算法题

1、顺序表删除最小值



题目描述：

从顺序表中删除具有最小值的元素(假设唯一)并由函数返回被删元素的值。空出的位置由最后一个元素填补

165

天

16

时

56

分

13

秒

by 公众号蓝蓝考研

思路

- 遍历找到最小值
- 从第二个元素逐个和当前最小值元素比较，并更新最小值且记录下标
- 保存最小值，因为最后要返回，同时实现空出的位置由最后一个元素填补。
- 最后返回删除的值即可

实现

```
/*
搜索整个顺序表，查找最小值元素并记住其位置，搜索结束后用最后一个元素填补空出的原最小值元素的位置。
*/
bool Del_Min(sqList &L, int &value)
{
    if(L.length == 0)
        return false;
    value = L.data[0];
    int pos = 0;
    for(int i=1; i<L.length; i++)
    {
        if(L.data[i] < value)
        {
            value = L.data[i];
            pos = i; //循环找最小值的元素
        }
        //空出的位置由最后一个填补
        L.data[pos] = L.data[L.length - 1];
        L.length--;
        return true;
    }
}
```

2、顺序表的逆置



题目描述：

设计一个高效算法，将顺序表 L 的所有元素逆置，要求算法的空间复杂度为O(1)

```
//方式1: 直接下标处理完成交换
void reverse(Sqlist &L){
    int tmp;//交换变量
    for(int i = 0;i<L.length/2;i++){
        tmp = L[i];
        L[i] = L[L.length-1-i];
        L[L.length-1-i] = tmp;
    }
}
```

3、删除顺序表中所有值为x的数据

💡 题目描述:

对长度为 n 的顺序表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 的数据元素。

方法一

```
void Del_x(Sqlist &L, int x){
    int k = 0;//用来记录需要移动的个数
    for(int i=0;i<L.length;i++){
        if(L.data[i] == x){
            k++;
        }else{
            L.data[i-k]=L.data[i];
        }
    }
    L.length = L.length - k;
}
```

方法二

- 两个指针，一个指针'p'用于遍历顺序表中的元素，另一个指针'q'用于记录当前有效的元素位置。
- 初始化'q'为0。
- 遍历顺序表，将不等于'x'的元素复制到'q'所指向的位置，并将'q'向后移动一位。
- 遍历完成后，将顺序表的长度更新即可。

```
int removeElementOfX(int *nums, int length, int x) {
    int q = 0; //记录有效元素位置
    for (int p = 0; p < length; p++) {
        if(nums[p] != x) {
            nums[q] = nums[p];
            q++;
        }
    }
    return q; //返回新长度
}
```

4、删除下标 $i \sim j$ ($i \leq j$) 的所有元素

💡 题目描述:

从给定顺序表L中删除下标 $i \sim j$ ($i \leq j$) 的所有元素，假定 i, j 都是合法的

思路

给定顺序表{0, 1, 2, 3, 4}

给定范围：1 - 2

(把3和4挪到前面给定的范围，再把顺序表长度缩小)

输出的顺序表{0, 3, 4}

这段代码是用来操作一个顺序表的数据结构。首先，定义了一个结构体 `SqList` 用来表示顺序表，它包含一个整型数组 `arr` 用来存储数据，数组的大小由 `Maxsize` 定义，`length` 表示当前顺序表中的元素个数。

实现

```
bool deleteElem(SqList &L, int i, int j){
    if(i>j || i<0 || j<0 || j>L.length){
        return false;
    }
    int distance = j-i+1;
    for(j;j<L.length;j++){
        L.data[i] = L.data[j+1];
        i++;
    }
    L.length = L.length - distance;
    return true;
}
```

5、小于表头放前面，大于放后面

💡 题目描述：

有个存放整数类型的顺序表L，设计算法将L中所有小于等于表头元素的整数放在前半部分，大于表头元素的整数放在后半部分

思路

遍历顺序表L中的所有元素，将其与表头元素比较，如果小于表头元素，则从该元素位置开始，将其之前的所有元素向后移动一个位置，并将该元素放入移动后的空位（也可以说是表头位置）



公众号 蓝蓝考研

实现

```

/* 移动元素将L中所有小于表头元素的整数放在前半部分，大于表头元素的放在后半部分（注意：没有考虑与
表头元素相等的元素） */
/* L[]指的是要移动的顺序表；m指的是顺序表中元素的个数 */
void moveEle(SqList &L, int m){
    int k=0; // 记录表头元素的下标
    int tmp=L.data[k]; // 记录表头元素的值
    int i=0; // 记录循环
    while(i<m){
        if(L.data[i]<tmp){ // 判断是否小于表头元素
            int save=L.data[i]; // 保存小于表头的元素
            for(int q=i; q>0; q--){ // 将L[i]之前的所有元素向后移动一个位置
                L.data[q]=L.data[q-1];
            }
            L.data[k]=save; // 再将剩下的空即L[k]填入save
        }
        i++; // 计数器加1
    }
}

```

6、有序表中删除值在s~t中元素

💡 题目描述：

从有序顺序表中删除其值在给定值 s 与 t 之间(要求 $s < t$)的所有元素，若 s 或 t 不合理或顺序表为空，则显示出错信息并退出运行

公众号 蓝蓝考研

思路

分析：因为是有序表，所以删除的元素必然是相连的整数。于是，我们可以先寻找值大于等于 s 的第一个元素（即第一个删除的元素），然后再寻找大于 t 的第一个元素（即最后一个删除的元素的下一个元素），要讲这段元素删除，则只需直接将后面大于 t 的元素前移就可以了。

实现

```
void del_1(Sqlist &L, int s, int t)
{
    int i = 0;
    while(i < L.length && L.a[i] < s) ++i; //寻找值 ≥ s 的第一个元素
    int j = i;
    while(j < L.length && L.a[j] <= t) ++j; //寻找值 > t 的第一个元素

    for(; j < L.length; ++j, ++i)
        L.a[i] = L.a[j]; //前移，填补被删元素位置

    n = i;
}
```

7、有序表删除重复元素

24计算机考研成员一战成硕！

💡 题目描述：

从有序(升序)顺序表中删除所有其值重复的元素，使表中所有元素的值均不同

公众号蓝蓝考研

思路

用类似于直接插入排序的思想，初始时将第一个元素视为非重复的有序表，之后依次判断后面的元素是否与前面的非重复元素有序表的最后一个元素相同，若相同则继续向后判断，若不同则插入前面的非重复有序表的，最后直至判断到表尾为止

实现

```
bool del(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])//查找下一个与上个元素不同的元素
            i++;
        L.data[i]=L.data[j];//找到后，将元素前移
    L.length=i+1;
    return true;
}
```

8、将两个顺序表合并成新有序表

24计算机考研成员一战成硕！

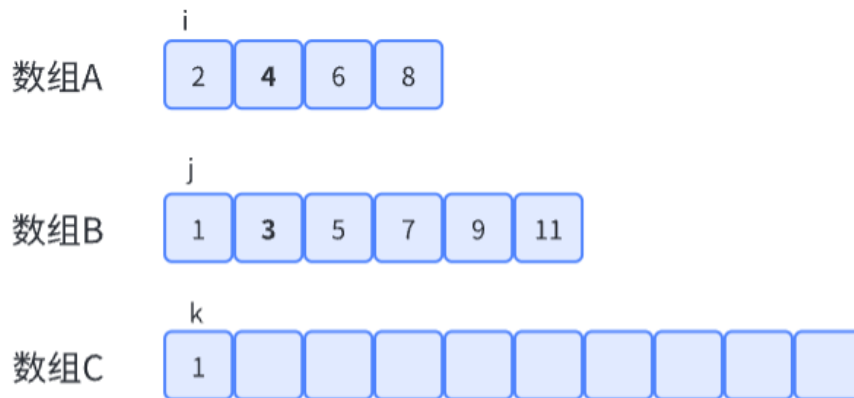
💡 题目描述：

将两个有序(升序)顺序表合并为一个新的有序顺序表，并由函数返回结果顺序表

公众号蓝蓝考研

思路

首先有两个顺序表，现在要合并成一个新的顺序表C,就是将依次比较大小然后放入C中，如果还有剩下没有比较完的，直接加在C顺序表后面



举例：

A[0]和B[0]比较，B[0]更小，则将B[0]放入到C数组中
A[0]和B[1]比较，A[0]更小，则将A[0]放入到C数组中

滴滴滴滴以此类推：
将更长的B数组剩余的元素放入到C数组中

公众号蓝蓝考研

实现

```
typedef int ElemType;
typedef struct {
    ElemType *data; // 指示动态分配数组的指针
    int Maxsize=100, length; // 数组的最大容量和当前个数
} SeqList;

bool Merge(SeqList A, SeqList B, SeqList &C) {
    // 将有顺序的顺序表A, B合并到新的有序顺序表C中

    if(A.length+B.length>C.Maxsize) // 判断非空
        return false;

    int i=0, j=0, k=0;
    // i, j, k分别记录ABC元素的下标
    while(i<A.length&& j<B.length){ // 循环，两两比较，小者存入结果表
        if(A.data[i]<=B.data[j]) // A表头比B表头小
            C.data[k++]=A.data[i++]; // 存A
        else
            C.data[k++]=B.data[j++]; // B表头比A小，存B
    }
    while(i<A.length) // 还剩一个没有比较完的顺序表，多余的部分，直接加在后面
        C.data[k++]=A.data[i++];
    while(j<B.length)
        C.data[k++]=B.data[j++];
    C.length = k;
    return true;
}
```

9、两个顺序表交换

💡 题目描述:

已知在一维数组A[m+n]中依次存放两个线性表(a1, a2, a3, ..., am)和(b1, b2, b3, ..., bn)。编写一个函数，将数组中两个顺序表的位置互换，即将(b1, b2, b3, ..., bn)放在(a1, a2, a3, ..., an)的前面

公众号：蓝蓝考研

思路1

方法一应该比较直观的方式，也可能是初试的时候容易想到的方式。题目没有做其他额外的要求，我们新建一个临时数组tmp，将数组前半部分放入tmp，后半部分移动到前面位置，然后tmp放后面，其实就是一个数的交换。

实现

```
void DealArrayC(SqList &L, int m, int n){
    int tmp[256];
    for(int i=0; i<m; i++){
        tmp[i] = L.data[i];
    }
    for(int i=m; i<m+n; i++){
        L.data[i-m] = L.data[i];
    }
    for(int i=0; i<m; i++){
        L.data[m+i] = tmp[i];
    }
}
```

思路2

- 先把(a1, a2, a3, ..., am)逆序为(am, ..., a3, a2, a1);
- 再把(b1, b2, b3, ..., bn)逆序为(bn, ..., b3, b2, b1);
- 此时数组变为了(am, ..., a3, a2, a1, bn, ..., b3, b2, b1)，观察发现此时将整个数组再逆序一次即可。

实现

```
void Exchange(SeqList &L, int m, int n){
    if(m + n != L.length || m < 0 || n < 0){
        return; //传入的参数 m, n 不合法
    }
    ElemType temp; //用来交换顺序表的两个元素的临时变量
    for(int i = 0; i < m / 2; i++) {
        temp = L.data[i];
        L.data[i] = L.data[m - i - 1]; //交换顺序表前面部分 L[0, ..., m] 对称位置元素值
        L.data[m - i - 1] = temp;
    }
    for(int j = m, i = 0; j < m + n / 2; j++, i++){
        temp = L.data[j];
        L.data[j] = L.data[L.length - i - 1]; //交换顺序表后面部分 L[m, ..., m+n] 对称位置元素值
        L.data[L.length - i - 1] = temp;
    }
    for(int k = 0; k < L.length / 2; k++){
        temp = L.data[k];
        L.data[k] = L.data[L.length - k - 1]; //交换整个顺序表 L[0, ..., m+n] 对称位置元素值
    }
}
```

```

        L.data[L.length - k - 1] = temp;
    }
}

```

10、二维数组的查找

24计算机考研成员一战成硕!

💡 题目描述:

在一个二维数组array中（每个一维数组的长度相同），每一行都按照从左到右递增的顺序排序，每一列都按照从上到下递增的顺序排序。请完成一个函数，输入这样的一个二维数组和一个整数，判断数组中是否含有该整数。

```

[
  [1,2,8,9],
  [2,4,9,12],
  [4,7,10,13],
  [6,8,11,15]
]

```

给定 target = 7，返回 true。

给定 target = 3，返回 false。

数据范围：矩阵的长宽满足 $0 \leq n, m \leq 500$ ，矩阵中的值满足 $0 \leq \text{val} \leq 9$

进阶：空间复杂度 $O(1)$ ，时间复杂度 $O(n+m)$

公众号 蓝蓝考研

思路1-暴力

双循环遍历所有的元素。

```

bool isExist(int **nums, int target){
    boolean flag = false;
    for(int i=0; i<nums.length; i++){
        for(int j=0; j<nums[0].length; j++){
            if(nums[i][j] == target)
            {
                flag = true;
            }
        }
    }
    return flag;
}

```

思路二双指针

	j	1	4	7	11	15
i	1	1	4	7	11	15
2	2	5	8	12	19	
3	3	6	9	16	22	
10	10	13	14	17	24	

Target = 8
当前遍历的值: 10
10 > 8, 往上走
i--, 相当于去除一行。j=0

	j	1	4	7	11	15
i	2	5	8	12	19	
3	3	6	9	16	22	
10	10	13	14	17	24	

Target = 8
当前遍历的值: 3
3 < 8, 没必要再往上选值比较,
j++, 相当于去除一列。

	j	1	4	7	11	15
i	2	5	8	12	19	
3	3	6	9	16	22	
10	10	13	14	17	24	

Target = 8
当前遍历的值: 6
6 < 8, 没必要再往上选值比较,
j++, 相当于去除一列, 4和5不用比较

	j	1	4	7	11	15
i	2	5	8	12	19	
3	3	6	9	16	22	
10	10	13	14	17	24	

Target = 8
当前遍历的值: 9
9 > 8, 没必要再往右选值比较,
i--, 相当于去除一列, 16和22不用比较

	j	1	4	7	11	15
i	2	5	8	12	19	
3	3	6	9	16	22	
10	10	13	14	17	24	

Target = 8
当前遍历的值: 8
相等, 结束

公众号 蓝蓝考研

```
bool isExist(int** nums, int row, int col, int target){
    int i = row - 1;
    int j = 0;
    if (target > nums[row-1][col-1] || target < nums[0][0]){
        return 0;
    }
    while (i >= 0 && j < col-1){
        if(target == nums[i][j]) {
            return true;
        }else if (nums[i][j] > target){
            i--;
        }else if (nums[i][j] < target){
            j++;
        }else{
            return true;
        }
    }
    return false;
}
```

11、判断括号是否匹配

思路

从左到右扫描字符数组, 遇到左括号入栈, 反之输出。栈顶元素与当前字符比较, 如果相同继续执行, 反之结束。

不相等的三种情况

- 弹出的栈顶元素与当前字符不匹配
- 栈空，字符未扫描完
- 栈不空，字符扫描完成

实现

```
//括号匹配
bool match(SqList s, char a[]) {
    int i;
    char t;
    for (i = 0; a[i] != '\0'; i++) {
        if (a[i] == '(' || a[i] == '[' || a[i] == '{') {
            push(s, a[i]);
        }
        else {
            if (isEmpty(s)) return false; //有多余的右括号
            pop(s, t);
            //判断栈顶元素是否与当前右括号匹配
            if (t == '(' && a[i] != ')') return false;
            else if (t == '[' && a[i] != ']') return false;
            else if (t == '{' && a[i] != '}') return false;
        }
    }
    return isEmpty(s); // 是否有多余的左括号
}
```

12、折半查找

思路

1. 初始化左右指针left和right,分别指向数组的起始位置和结束位置。
2. 重复以下步骤，直到left指针大于right指针的时候停止。
 1. 将中间元素与目标值进行比较。
 2. 如果中间元素等于目标值，表示找到了目标值，返回中间元素的索引。
 3. 如果中间元素大于目标值，则标指目标值可能在该中间元素的左侧，更新右指针为mid-1。
 4. 如果中间元素小于目标值，则标指目标值可能在该中间元素的右侧，更新左指针为mid+1。
3. 循环结束仍然没有找到目标值，则表示目标值不存在于数组中。

实现

```
int binarySearch(int arr[], int length, int target) {
    // 初始化左右指针
    int left = 0;
    int right = n - 1;

    while (left <= right) {
        int mid = (left + right) / 2;

        if(arr[mid] == target) {
            return mid; // 返回目标元素的下标
        }else if (arr[mid] < target) {
            left = mid + 1;
        }else {
            right = mid - 1
        }
    }
}
```

```

}

return -1; // 未找到，返回一个错误表示
}

```

13、找出第k小的元素

24计算机考研成员一战成硕！



题目描述：

在数组 $L[1...n]$ 中找出第K小的元素(从小到大处于第k个位置的元素)

公众号 蓝蓝考研

思路

1.先排序

2.直接输出值（题解没给最优解）

也有基于快排或者堆排的更低复杂度的算法，但是不建议。

思路一：不知道你们看到找到前k个元素的时候，你会想到什么，不知道会不会想到王道课后习题里面考察的一类题：经过第一轮排序，某个元素已经再正确的位置的排序有什么...例如冒泡排序，可以每一次都把最小的那个元素冒泡到第一个，题目要求找到第k小的元素，我们是不是只需要冒k个泡就行，所以时间复杂度可以在 $O(kn)$ ，选择排序同理

思路二：还有一种方法是建堆，建立大根堆或者小根堆，大根堆是最大的元素在最上面，小根堆是最小的元素在最上面，我们可以把整个数组建一个小根堆，然后每次弹出最上面那个元素，然后根据堆的特性，最小的元素会上浮，一次上浮的时间复杂度是 $\log n$ ，建堆的时间复杂度是 $o n$ ，所以弹出前k个元素的时间复杂度是不是 $O(n+k*\log n)$

实现

```

void Quicksort(int q[] , int l , int r) {
    //如果只有一个元素直接返回
    if( l >= r ) return;
    //第一部分：选取最左边的元素为x，因为边界不能动，并且使用do...while语句，所以把i和j
    分别设成边界元素的后一个位置
    int x = q[l], i = l - 1, j = r + 1;
    //第二部分：
    while( i < j ) {
        //从左边寻找一个不满足小于x的元素
        do i++; while( q[i] < x );
        //从右边寻找一个不满足大于x的元素
        do j--; while( q[j] > x );
        //并且在i < j情况在再交换
        //如果当不满足i<j说明此时以j为下标的分界线把数组分为了两个部分
        if( i < j ) swap(q[i] , q[j]);
    }
    //第三部分：
    //通过上面的循环把利用元素x把数组分成了逻辑上的两个部分，物理上是用下标j分为了两个部
    分，然后再分别对这两部分进行递归就行
    // 例如：原数组：5, 2, 7, 9, 24, -1, 7, 354
    //一次quicksort之后 [-1, 2 (j), 7, 9, 24, 5 (x), 7, 354], j前面的都是小于x的，
    j后面都是大于x的，但是前后不保证有序，需要递归
    quicksort(q , l , j);
    quicksort(q , j + 1 , r);
}

```

```

}
void main() {
    int k;
    cin >> k;
    int arr[] = { 5,7,5,3,1 };
    Quicksort(arr , 0 , 4);
    cout << arr[k - 1];
}

```

14、冒泡排序

思路

1. 从头遍历数组，依次比较相邻元素的大小，如果前一个元素大于后一个元素，则交换它们的位置。
2. 重复以上步骤，直到遍历完成。

实现

```

void bubbleSort(int arr[], int length) {
    for (int i = 0; i < n - 1; i++) {
        for(int j = 0; j < n - 1 - i; j++) {
            // 比较相邻元素，如果前一个元素大于后一个元素，则交换它们的位置
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

```

15、选择排序

在算法实现时，每一趟确定最小元素的时候会通过不断地比较交换来使得首位置为当前最小，交换是个比较耗时的操作。

其实我们很容易发现，在还未完全确定当前最小元素之前，这些交换都是无意义的。我们可以通过设置一个变量min，每一次比较仅存储较小元素的数组下标，当轮循环结束之后，那这个变量存储的就是当前最小元素的下标，此时再执行交换操作即可

实现

```

// 选择排序函数
void selectionSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int minIndex = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        // 交换当前位置与最小值的位置
        int temp = arr[i];
        arr[i] = arr[minIndex];
        arr[minIndex] = temp;
    }
}

```

16、快速排序

1. 快速排序（Quick Sort）是一种常用的排序算法，其基本思想是通过递归地将数组划分为较小和较大的两个子数组，然后对子数组进行排序，最终完成整个数组的排序。
2. 分区操作是快速排序的核心，通过选择基准元素和两个指针的移动，将数组分为两部分。
3. 可以通过递归方式实现快速排序，将左右两个子数组分别进行排序。

时间复杂度： $O(n\log n)$, 其中 n 是数组的长度。

空间复杂度： $O(\log n)$, 每次递归调用都需要额外的栈空间。

思路

1. 选择一个基准元素（通常选择数组的第一个元素）。
2. 将数组分为两部分，比基准元素小的或等于的元素放在左边，比基准元素大的元素放在右边。这一步称为分区操作。
3. 对左右两个子数组递归的进行快速排序。

实现

```
#include <stdio.h>

// 快速排序函数
void quickSort(int* arr, int left, int right) {
    if(left >= right) return; // 基线条件，只有一个元素时无需操作

    int pivot = arr[left]; // 选择第一个元素作为基准元素
    int low = left;
    int high = right;

    while (low < high) {
        while (low < high && arr[high] >= pivot) high--; // 从右往左找到第一个小于基准元素的值
        arr[low] = arr[high];

        while(low < high && arr[low] <= pivot) low++; // 从左往右找到第一个大于基准元素的值
        arr[high] = arr[low];
    }

    arr[low] = pivot; // 将基准元素放到正确的位置

    // 递归的对左右两部分进行快速的排序
    quickSort(arr, left, low - 1); // 左子数组
    quickSort(arr, low + 1, right); // 右子数组
}
```

17、直接插入排序

把 n 个待排序的元素看成为一个有序表和一个无序表。开始时有序表中只包含1个元素，无序表中包含有 $n-1$ 个元素，排序过程中每次从无序表中取出第一个元素，将它插入到有序表中的适当位置，使之成为新的有序表，重复 $n-1$ 次可完成排序过程

```
// 直接插入排序函数
```



```

void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;

        // 将比key大的元素向右移动
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }

        // 找到key合适的位置，插入
        arr[j + 1] = key;
    }
}

```

18、10年真题数组循环左移

💡 题目描述：

【2010统考真题】设将 $n(n>1)$ 个整数存放到一维数组 R 中。设计一个在时间和空间两方面都尽可能高效的算法。将 R 中保存的序列循环左移 $p(0<p<n)$ 个位置，即将 R 中的数据由 $(x(0), x(1), x(2), \dots, x(n-1))$ 变换为 $(x(p), x(p+1), \dots, x(0), x(1), \dots, x(p-1))$ 要求：

- 1) 给出算法的基本设计思想。
- 2) 根据设计思想，采用C或C++或Java语言描述算法，关键之处给出注释。
- 3) 说明你所设计算法的时间复杂度和空间复杂度。

暴力解

```

void Reverse(int []R, int p){
    int n=R.length;
    int A[p]; //开辟一个数组，用于存放R的前p个元素
    for(int i=0; i<p; i++){
        A[i]=R[i];
    } //将R中前p个元素暂存到A中

    for(int i=0; i<n-p; i++){
        R[i]=R[i+p];
    } //将R中后n-p个元素移到R中最前面，经过第一步后，前半部分最后一个元素下标是p-1，所以后半部分第一个元素从p开始

    for(int i=0; i<p; i++) //将A中的元素放回R中的后面，在第二步中，R数组前面部分有n-p个元素，那么最后元素的下标为n-p-1，所以我们移动的这部分元素第一个元素下标为n-p，第二个元素是n-p+1，一共移动p个元素
        R[i+n-p]=A[i];
}

```

快慢指针

```

/*
算法思路：
1. 先将n个数据原地逆置 9,8,7,6,5,4,3,2,1,0;
2. 将n个数据拆解成[9,8,7,6,5,4,3] [2,1,0]

```

2. 将前 $n-p$ 个数据和后 p 个数据分别原地逆置；[3,4,5,6,7,8,9] [0,1,2]

复杂度分析：

时间复杂度： $O(n)$ ；空间复杂度： $O(1)$ ；

*/

```
void Reverse(Sqlist &L,int left ,int right){
```

```
    //将数组R中的数据原地逆置
```

```
    //i等于左边界left,j等于右边界right;
```

```
    int i = left,j = right;
```

```
    int temp;
```

```
    //交换pre[i] 和 pre[j] 的值
```

```
    while (i < j) {
```

```
        //交换
```

```
        temp = L.data[i];
```

```
        L.data[i] = L.data[j];
```

```
        L.data[j] = temp;
```

```
        //i右移,j左移
```

```
        i++;
```

```
        j--;
```

```
    }
```

```
}
```

```
void Leftshift(int *pre,int n,int p){
```

```
    //将长度为n的数组pre 中的数据循环左移p个位置
```

```
    if (p>0 && p<n) {
```

```
        //1. 将数组中所有元素全部逆置
```

```
        Reverse(L, 0, n-1);
```

```
        //2. 将前n-p个数据逆置
```

```
        Reverse(L, 0, n-p-1);
```


```
        //3. 将后p个数据逆置
```

```
        Reverse(L, n-p, n-1);
```

```
    }
```

```
}
```

19、11年中位数

 题目描述：

【2011统考真题】一个长度为 $L(L \geq 1)$ 的升序序列 S ，处在第 $\lceil L/2 \rceil$ 个位置的数称为 S 的中位数。例如，若序列 $S(1)=(11, 13, 15, 17, 19)$ ，则 $S(1)$ 的中位数是15，两个序列的中位数是含它们所有元素的升序序列的中位数。例如，若 $S(2)=(2, 4, 6, 8, 20)$ ，则 $S(1)$ 和 $S(2)$ 的中位数是11。现在有两个等长升序序列 A 和 B ，试设计一个在时间和空间两方面都尽可能高效的算法，找出两个序列 A 和 B 的中位数。要求：

- 1) 给出算法的基本设计思想。
- 2) 根据设计思想，采用C或C++或Java语言描述算法，关键之处给出注释。
- 3) 说明你所设计算法的时间复杂度和空间复杂度。

i	2	4	6	8	20
---	---	---	---	---	----

j	11	13	15	17	19
---	----	----	----	----	----

2	4	6	8	11	13	15	17	19	20
---	---	---	---	----	----	----	----	----	----

暴力解

- 新建一个数组C
- 合并到数组C
- 排序

```
#include <stdio.h>
#include <stdlib.h>

void deal(int A[], int B[], int n){
    int* C = (int*) malloc(sizeof(int) * n * 2);
    int i = 0, j = 0, index = 0;
    while(i < n && j < n){ //归并合并
        if(A[i] < B[j]) C[index++] = A[i++];
        else C[index++] = B[j++];
    }
    while(i < n) C[index++] = A[i++]; //处理剩余的元素
    while(j < n) C[index++] = B[j++];
    printf("%d", C[n - 1]);
}

int main(){
    int A[] = {11, 13, 15, 17, 19};
    int B[] = {2, 4, 6, 8, 20};
    ans(A, B, sizeof(A)/sizeof(A[0]));

    return 0;
}
```

优解

- 做本题最简单直接的一种想法大概就是将两个升序序列合并成一个长度为 $2L$ 的升序序列，中位数就是第 L 个元素。但是这样的暴力解法时间和空间复杂度较高，所以本题我在暴力解法的基础上做一些**优化**降低时间和空间复杂度。
- 其实我们没有必要把两个长度为 L 的升序序列合并，因为我们只需要确定前 L 个元素的位置即可，后 L 个元素则不需要考虑，基于此我们可以使用双指针的方式，两个指针分别指向两个数组当前最小值，每次比较指针所指元素的值，更小的一方指针向前移动一步，并且计数器加1，当计数器的值等于 L 的时候说明我们已经找到了中位数。 **具体步骤：**

1. 循环确定前L个元素的位置，其中每次指针所指元素更小的一方，让其指针向前移动一步，同时计数器加1。
2. 当计数器的值等于L的时候就可以返回中位数。
3. 如果循环结束仍然没有返回中位数，则返回0，属于未知错误

```
int Search(int arr1[], int arr2[], int size) {
    int median = 0, count = 0; // 中位数, 计数器
    int idx1 = 0, idx2 = 0;
    // 确定升序序列中第n位元素
    while (idx1 < size && idx2 < size) {
        if (arr1[idx1] < arr2[idx2]) {
            if (++count == size) {
                median = arr1[idx1];
            }
            idx1++; // 指针加1
        } else {
            if (++count == size) {
                median = arr2[idx2];
            }
            idx2++; // 指针加1
        }
        if (count >= size) {
            return median; // 优化循环次数，一旦确定了第n位元素，立即返回
        }
    }
    return 0; // 未知错误，确保健壮性
}
```

20、13年求主元素

💡 题目描述：

【2013统考真题】已知一个整数序列 $A=(a_1, a_2, a_3, \dots, a_n)$ ，其中 $0 \leq a(i) < n$ ($0 \leq i < n$)。若存在 $a(p_1)=a(p_2)=\dots=a(p_m)=x$ 且 $m > n/2$ ($0 \leq p(k) < n$, $1 \leq k \leq m$)，则称 x 为 A 的主元素。例如 $A=(0, 5, 5, 3, 5, 7, 5, 5)$ ，则5为主元素；又如 $Z=(0, 5, 5, 3, 5, 1, 5, 7)$ ，则 A 中没有主元素。假设 A 中的 n 个元素保存在一个一维数组中，请设计一个尽可能高效的算法，找出 A 的主元素。若存在主元素，则输出该元素；否则输出-1。要求：

- 1) 给出算法的基本设计思想。
- 2) 根据设计思想，采用C或C++或Java语言描述算法，关键之处给出注释。
- 3) 说明你所设计算法的时间复杂度和空间复杂度。

公众号蓝蓝考研

思路1

题目要求我们查找是否存在主元素，那可以直接定义找到为1，没找到为0，并写好注释，这样改卷人看的也非常的清晰。既然要找某个数是否满足主元素的性质，那我就每个数去检查是否为主元素，要处理每个元素，则少不了遍历。

- 选择数组的每一个元素 i
- 统计 i 在整个数组出现的次数
- 如果大于 $n/2$ 则返回

```
int majority(int A[], n){
    int m;
    //遍历每一个元素
```

```

for (int i=0; i<n; i++){
    //由于每次遍历的元素 都是从0开始统计出现的次数
    m=0;
    for (int j=0; j<n; j++)
        if (A[i]==A[j])
            m++;
    if (m>n/2){
        //找到了主元素
        return A[m];
    }
}
//未找到主元素
return -1;
}

```

思路2

```

void Func(int A[], int n)
{
    //动态申请一个数组，用于记录A中各个不同元素的出现次数
    int* arr = (int*)malloc(n * sizeof(int));
    //初始化为0
    for (int i = 0; i < n; i++)
        arr[i] = 0;
    //以A中元素作为下标，记录A中各个不同元素的出现次数
    for (int i = 0; i < n; i++)
        arr[A[i]]++;

    //用k记录是否找到一个元素出现次数大于n / 2
    int k = -1;
    for (int i = 0; i < n; i++)
    {
        if (arr[A[i]] > n / 2)
        {
            k = A[i];
            break;
        }
    }
    //输出k，并释放申请的动态数组
    printf("%d", k);
    free(arr);
}

```

21、18年数组最小正整数

💡 题目描述：

【2018统考真题】给定一个含 $n(n \geq 1)$ 个整数的数组，请设计一个在**时间**上尽可能高效的算法，找出数组中未出现的最小正整数。例如，数组 $\{-5, 3, 2, 3\}$ 中未出现的最小正整数是1；数组 $\{1, 2, 3\}$ 中未出现的最小正整数是4。要求：

- 1) 给出算法的基本设计思想。
- 2) 根据设计思想，采用C或C++语言描述算法，关键之处给出注释。
- 3) 说明你所设计算法的时间复杂度和空间复杂度。

1	2	3	4
---	---	---	---

1	2	5	9
---	---	---	---

0	1	1	1	1
0	1	2	3	

0	1	1	0	0	1			0	1
0	1	2	3	4	5	6	7	8	9



公众号 蓝蓝考研

暴力解

```
int ans(int A[], int n){
    bool flag;
    for(int i = 1; i < n + 1; i++){
        flag = false;
        for (int j = 0; j < n; ++j) {
            if(A[j] == i){
                flag = true;
                break;
            }
        }
        if(flag == false){
            return i;
        }
    }
    return n + 1;
}
```

优解

1. 新建一个数组，所有值初始化为0
2. 以A数组的元素值为基础，根据值找新建数组的下标，存在则赋值为1
3. 遍历新建数组，如果为0则直接返回下标，没有为0的值说明所有当前数组元素都存在，则最小值为 $n+1$

```

int findMissMin(int A[],int n){
    int i,*B;
    B=(int *)malloc(sizeof(int)*(n+1)); //分配空间
    memset(B,0,sizeof(int)*n);
    for(i=0;i<n;i++){
        if(A[i]>0&&A[i]<=n)
            B[A[i]]=1;
    }
    for(i=1;i<=n,i++){
        if(B[i]==0)
            return i;
    }
    return n+1;
}

```

22、20年三元组最小距离

🔦 题目描述：

【2020统考真题】定义三元组(a, b, c)(a、b、c均为正数)的距离 $D=|a-b|+|b-c|+|c-a|$ 给定3个非空整数集合S1、S2和S3，按升序分别存储在3个数组中。请设计一个尽可能高效的算法，计算并输出所有可能的三元组(a, b, c)($a \in S1, b \in S2, c \in S3$)中的最小距离。例如S1 ={-1, 0, 9}, S2={-25, -10, 10, 11}, S3={2, 9, 17, 30, 41}，则最小距离为2，相应的三元组为(9, 10, 9)。要求：

要求：

- 1) 给出算法的基本设计思想。
- 2) 根据设计思想，采用C或C++语言描述算法，关键之处给出注释。
- 3) 说明你所设计的算法的时间复杂度和空间复杂度。

公众号蓝蓝考研

暴力解

```

/**
 * 求一个数的绝对值
 *
 * @param num 指定整数，可以是正数，也可以是负数
 * @return 绝对值
 */
int abs(int num) {
    if (num < 0) {
        return -num;
    } else {
        return num;
    }
}

/**
 * 求所有可能的三元组中的最小距离
 *
 * @param S1 第一个非空整数集合
 * @param n1 第一个数组的长度
 * @param S2 第二个非空整数集合
 * @param n2 第二个数组的长度
 * @param S3 第三个非空整数集合
 * @param n3 第三个数组的长度
 * @return 三元组中的最小距离
 */
int minDistance(int S1[], int n1, int S2[], int n2, int S3[], int n3) {
    // 变量，记录 S1 中的元素

```

```

int a = s1[0];
// 变量，记录 s2 中的元素
int b = s2[0];
// 变量，记录 s3 中的元素
int c = s3[0];
// 变量，记录三元组中的最小距离
int minD = abs(a - b) + abs(b - c) + abs(c - a);

// 三重 for 循环遍历三个非空整数集合
for (int x = 0; x < n1; x++) {
    for (int y = 0; y < n2; y++) {
        for (int z = 0; z < n3; z++) {
            // 计算由 s1[x]、s2[y] 和 s3[z] 组成的三元组的最小距离
            int min = abs(s1[x] - s2[y]) + abs(s2[y] - s3[z]) + abs(s3[z] -
s1[x]);

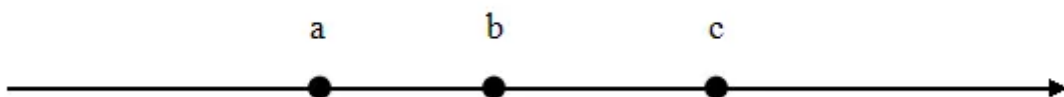
            // 如果比 minD 更小，那么更新
            if (min < minD) {
                minD = min;
                a = s1[x];
                b = s2[y];
                c = s3[z];
            }
        }
    }
}
printf("最小距离为 %d，相应的三元组为 (%d, %d, %d)。", minD, a, b, c);
return minD;
}

```

优解

本题的算法思想比较考验水平，所以这里尽可能的详细一些，通过图文的方式，给大家好好解释一下。

题目让我们求三元组中的最小距离，当然可以通过三个for循环实现暴力解法，既然可以通过嵌套循环实现，那一般也可以通过指针实现并降低时间复杂度，这里我们采用三指针的方式实现。假设*i j k*分别指向三个数组*a b c*，呢我们可以通过一个while循环来计算 $|a[i]-b[j]| + |b[j]-c[k]| + |c[k]-a[i]|$ ，关键点是在于每次循环后，该如何更新指针变量。这里详细解释一下：



如上图：*a b c*就相当于每次遍历的三个数组元素，三元组的距离自然是 $ab+bc+ac = 2ac$ （可以发现距离与点*b*没有关系）。我们要求的是最小距离，所以我们要考虑的就是如何移动这*a c*这两个点，使得 $ab+bc+ac$ 的值最小。

1. 假设将*a*向右移动，可以发现*ac*变小，没问题，但向左移动，则*ac*变大，不可以
2. 假设将*c*向右移动，可以发现*ac*变大，不可以，但向左移动，则*ac*变小，可以。综上所述我们可以发现，有两种移动指针的方式，可以使得每次遍历得到的三元组距离变小，其中我们采用第一种方式，将*a*向右移动。（☺够详细了吧！！！）

```

#include <stdio.h>

// 求三数中的最小值
int MinNum(int a, int b, int c) {

```



```

int min = a < b ? a : b;
min = min < c ? min : c;
return min;
}

// 求三元组的最小距离
int MinDistance(int a[], int b[], int c[], int aLen, int bLen, int cLen) {
    int minDis = 100000000; // 初始值为无穷大，表示三元组中的最小距离
    int curDis = 0;          // 表示当前三元组距离
    int min = 0;             // 当前遍历的三个数中的最小值
    int i = 0, j = 0, k = 0; // 三个整型数组的指针
    while (i < aLen && j < bLen && k < cLen) {
        // 计算当前三元组距离
        curDis = abs(a[i] - b[j]) + abs(b[j] - a[i]) + abs(c[k] - a[i]);
        if (curDis < minDis) minDis = curDis;
        min = MinNum(a[i], b[j], c[k]);
        // 让当前最小的数哪一个数的数组指针向右(向前)移动一位
        if (min == a[i]) i++;
        else if (min == b[j]) j++;
        else if (min == c[k]) k++;
    }
    return minDis; // 返回结果
}

```

22、尾插法建立链表

```

LinkList List_TailInsert(LinkList &L)
{
    int x;
    L = (LinkList)malloc(sizeof(LNode));
    LNode *s, *r = L; //r为表尾指针
    scanf("%d", &x);
    while(x != -1)
    {
        s = (LNode*)malloc(sizeof(LNode));
        s->data = x;
        r->next = s;
        r = s; //r指向新的表尾结点
        scanf("%d", &x);
    }
}

```

23、头插法建立链表

```

/*
采用头插法建立单链表时，读入数据的顺序与生成的链表中的元素是相反的。每个结点插入的时间为 $O(1)$ ，设单链表长为 $n$ ，则总时间复杂度为 $O(n)$ 。
*/
LinkList List_HeadInsert(LinkList &L)
{
    LNode *s;
    int x;
    L = (LinkList)malloc(sizeof(LNode));
    L->next = NULL;

```

```

scanf("%d",&x);
while(x!=999)
{
    s = (LNode*)malloc(sizeof(LNode)); //创建新节点
    s->data = x;
    s->next = L->next;
    L->next = s; //将新结点插入表中，L为头指针
    scanf("%d",&x);
}
}

```

24、从尾到头输出结点

24计算机考研成员一战成硕!



题目描述:

设 L 为带头结点的单链表，编写算法实现从尾到头反向输出每个结点的值

```

void R_Print(LinkList &L)
{
    if(L->next != NULL)
    {
        R_Print(L->next);
    }
    print(L->data);
}

```

25、删除单链表L所有值为x的结点

```

/*
用p从头至尾扫描单链表，pre指向*p结点的前驱。若p所指结点的值为x，则删除，并让p移向下一个结点，否则让pre、p指针同步后移一个结点。
该段代码可以一直使用，if条件可以更改。时间复杂度为O(n)，空间复杂度为O(1)。
*/
void Del_X(LinkList &L,int x)
{
    LNode *p = L->next,*pre = L,*q;
    while(p!=NULL)
    {
        if(p->data ==x)
        {
            q = p; //q指向该结点
            p = p->next;
            pre->next = p; //删除*q结点
            free(q);
        }
        else //否则,pre和p同步后移
        {
            pre= p;
            p = p->next;
        }
    }
} //while
}

```

26、带头结点单链表就地逆置

24计算机考研成员一战成硕！



题目描述：

将带头结点的单链表就地逆置

```
/*
将头结点摘下，然后从第一结点开始，依次插入到头结点的后面，直到最后一个结点为止，这样就实现了链表的逆置。
*/
LinkList Reverse(LinkList L)
{
    LNode *p,*r; //p为工作指针，r为p的后继，以防断链
    p = L->next; //从第一个元素结点开始
    L->next = NULL; //先将头结点L的next域置为NULL
    while(p!=NULL) //依次将元素结点摘下
    {
        r = p->next; //暂存p的后继
        p->next = L->next; //将p结点插入到头结点之后
        L->next = p;
        p = r;
    }
    return L;
}
```

27、删除表中介于给定两值之间元素

依次检查每个节点，当节点符合题目要求时进行删除

```
void RangeDelete(LinkList &L,int min,int max){
    LNode *pr=L,*p=L->next; //p是检测指针，pr是其前驱
    while(p!=NULL)
    if(p->data>min&&pr->data<max){ //寻找到被删结点，删除该结点
        pr->next=p->next;
        free(p);
        p=pr->next;
    }
    else{ //否则继续寻找被删结点
        pr=p;
        p=p->next;
    }
}
```

28、09年链表倒数第k个结点

42. (15分) 已知一个带有表头结点的单链表，结点结构为



假设该链表只给出了头指针list。在不改变链表的前提下，请设计一个尽可能高效的算法，查找链表中倒数第k个位置上的结点（k为正整数）。若查找成功，算法输出该结点的data值，并返回1；否则，只返回0。要求：

- (1) 描述算法的基本设计思想
- (2) 描述算法的详细实现步骤
- (3) 根据设计思想和实现步骤，采用程序设计语言描述算法（使用C或C++或JAVA语言实现），关键之处请给出简要注释。

公众号 蓝蓝考研

暴力解

```
/**
 * 暴力求解
 * 公众号：蓝蓝考研
 * 先求出链表结点长度，然后再遍历扫描len-k个结点即可
 */
int search_force(Linklist list, int k) {
    int len = 0;
    Linklist *p = list->link;
    //求出链表的长度
    while (p != NULL) {
        p = p->link;
        len++;
    }
    //边界
    if (len < k) {
        return 0;
    }
    p=list;
    //求出是正数的第几个结点
    int cnt = len - k + 1;
    for(int i=0; i<cnt; i++){
        p = p->link;
    }
    printf("%d\n", p->data);
    return 1;
}
```

优解

```

int search_k(Linklist head, int k) {
    ListNode* slow=list->link;
    ListNode* fast=list->link;
    while(--k) fast=fast->link; //倒数第k个位置 k从1开始 如果循环条件是k-- left和right之
    间的距离会少1 即会输出倒数第k+1个元素
    if(fast==NULL) return 0; //fast有可能为空，最好判断一下
    while(fast->link!=NULL){
        fast=fast->link;
        slow=slow->link;
    }
    printf("%d\n", slow->data);
    return 1;
}

```

29、判断带头结点循环双链表是否对称

思路

函数中的变量iRet初始化为1，表示默认是对称的。接下来定义了两个指针p和q，分别指向链表的第一个节点和最后一个节点。

while循环的条件是p不等于q或者p的前驱节点等于q，同时iRet等于1。循环中，如果p节点的数据等于q节点的数据，则将p指向下一个节点的指针p->rLink，q指向前一个节点的指针q->lLink。否则，将iRet设为0，表示链表不对称。

最后返回iRet的值，即判断结果。如果iRet为1，表示链表对称；如果iRet为0，表示链表不对称。

实现

```

int symme (Dbllist DL)
{
    int iRet =1;
    DbllNode*p=DL->rlink, *q-->lLink;
    while ((p!=q||p->lLink==q)&& (iRet==1))
        if (p->data ==q->data)
        {
            p=p->rLink;;
            q=q->lLink;
        }
        else iRet =0;
    return iRet;
}

```

30、找出两个链表的公共结点

思路

- 1.先让快指针走k个单位，然后我们再让快慢指针一起走
- 2.是不是当快指针指向最后一个节点的时候慢指针就刚好指到倒数第k个节点了？

```

ListNode* FindNode(ListNode* head,int k) {
    if( !head )return head;
    ListNode* fast = head;
    ListNode* slow = head;
    ListNode* preslow = nullptr;
    //先让快指针走k个单位，然后我们再让快慢指针一起走，

```

```

//是不是当快指针指向最后一个节点的时候慢指针就刚好指到倒数第k个节点了?
while( k-- )fast = fast->next;
while( fast ) {
    //preslow用来指向慢指针前一个元素方便删除,
    //因为链表删除节点只能用前一个节点删除后面的哪个节点
    preslow = slow;
    slow = slow->next;
    fast = fast->next;
}
//如果要删除的结点就是头结点,
//那此时preslow=null, 只需让头节点向后移动, 再删除原来的头节点
if(!preslow){
    ListNode* pDel = head;
    head = head->next;
    delete pDel;
}
//如果要删除的不是头节点, 那么只需要让被删除结点的前一个结点指向被删除结点的后一个结点,
//即让preslow的下一个结点为slow的下一个结点, 删除slow

else {
    preslow->next = slow->next;
    delete slow;
}
return head;
}

```

31、两个链表A和B，奇数存放在A，偶数存放在B



题目描述：

将带头结点的单链表A分解为两个带头结点的单链表A和B，使得A表中含有原表中值为奇数的元素，而B表中含有原表中值为偶数的元素

思路

- 1.申请一个变量,确定奇偶
- 2.将偶数序号的摘下来尾插到链表B,奇数序号的拆下来尾插到链表A

实现

```

ListNode* splitAtoAB(ListNode*A)
{
    ListNode*B=new ListNode(-1);
    B->next=NULL;

    ListNode*tailA=A;    //A的尾指针
    ListNode*cur=A->next;//工作循环的遍历指针

    ListNode*tailB=B;    //B的尾指针

    int os=1;
    while(cur)
    {
        if(os%2==1)
        {
            tailA->next=cur;

```

```

        tailA=tailA->next;
    }
    else
    {
        tailB->next=cur;
        tailB=tailB->next;
    }
    cur=cur->next;
    os++;
}
tailA->next=NULL;
tailB->next=NULL;
return B;
}

```

32、递增链表去掉数值相同的元素



题目描述：

递增有序的线性表，存储方式为单链表，设计算法去掉数值相同的元素

思路

首先，定义了三个指针变量p、q，其中p指向链表的第一个节点，q用于辅助删除操作。

接下来，通过判断p是否为空来确定链表是否为空。如果链表为空，则直接返回。

然后，进入一个while循环，循环条件是p的下一个节点不为空。这是因为循环的目的是遍历整个链表，所以需要保证p不是最后一个节点。

在循环内部，首先将q指向p的下一个节点。然后判断p节点的值和q节点的值是否相等。如果相等，则说明p和q指向的节点是重复的节点，需要删除q节点。

删除操作是通过将p的next指针指向q的next指针实现的，即将p的next指针跳过q节点，直接指向q的下一个节点。

如果p和q指向的节点不相等，则说明p和q指向的节点不重复，需要将p指针移动到下一个节点。

循环结束后，整个链表中的重复节点已经被删除。

实现

```

void delNode(ListNode* pHead) {
    ListNode* p = pHead->next;
    ListNode* q;
    if(!p) return;
    while(p->next){
        q = p->next;
        if(p->data == q->data){
            p->next = q->next;
        }
        else
            p = p->next;
    }
}

```

33、删除一个最小值的高效算法



题目描述:

试编写在带头结点的单链表 L 中删除一个最小值结点的高效算法

实现

```
LinkList Delete_Min(LinkList &L)
{
    LNode *pre = L, *p = pre->next; //p为工作指针,pre指向其前驱
    LNode *minpre = pre, *minp = p; //保存最小值结点及其前驱
    while(p!=NULL)
    {
        if(p->data < minp->data)
        {
            minp = p; //找到比之前找到的最小值结点更小的结点
            minpre = pre;
        }
        pre = p; //继续扫描下一个结点
        p = p->next;
    }
    minpre->next = minp->next; //删除最小值结点
    free(minp);
    return L;
}
```

34、按递增输出单链表结点的数据元素

实现

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

void Min_print(LinkList L){
    LNode *p, *q, *pre;
    while(L->next){
        pre=L;
        p=L->next;
        while(p->next){ //这里用p->next其实和冒泡排序内层循环,j和j+1比较,j不用走到最后
            就已经完成了全部比较。
            if(p->next->data<p->data)
                pre=p; //链表的删除操作,需要用到前一个结点,避免断链
            p=p->next;
        }
        printf("%d",pre->next->data);
        q=pre->next;
        pre->next=q->next;
        free(q);
    }
    free(L); //最后不要忘记free掉头结点
}
```


35、求二叉树的最大深度

思路

1. 使用递归的方式来求解二叉树的最大深度，将问题分解为子问题。
2. 在递归过程中要注意递归结束的条件，即空树的深度为0。

时间复杂度： $O(n)$,其中 n 为二叉树的节点个数。

空间复杂度： $O(n)$,最坏情况下二叉树退化成了链表，则递归栈的深度为 n 。

实现

1. 如果树为空，则深度为0。
2. 否则，通过递归求解左子树和右子树的深度，取较大值。

```
#include <stdio.h>
#include <stdlib.h>

// 定义二叉树节点
typedef struct BiTNode {
    Element data; // 数据域
    BiTNode *left, *right; // 左右节点
}BiTNode, *BiTree;

// 计算二叉树深度
int TreeDepth(BiTree T) {
    if(!T) return 0; // 空树深度为0
    // 递归计算左右子树的深度
    int LDepth = TreeDepth(T->left);
    int RDepth = TreeDepth(T->right);
    return LDepth > RDepth ? LDepth + 1 : RDepth + 1;
}
```

36、递归先序遍历

当解决树结构的问题时，递归先序遍历是一个常见的算法。先序遍历的意思是，首先访问根节点，然后按照先左后右的顺序遍历其左右子树。递归先序遍历的基本思路是从根节点开始，先访问当前节点，然后递归地遍历左子树，最后递归地遍历右子树。

思路

1. 如果当前节点为空，则返回。
2. 访问当前节点，并执行相应操作。
3. 递归调用函数，遍历左子树。
4. 递归调用函数，遍历右子树。

实现

```
#include <stdio.h>

// 树节点的定义
struct TreeNode {
    int val;
    struct TreeNode *left;
```

```

    struct TreeNode *right;
};

// 递归先序遍历函数
void PreOrder(struct TreeNode* root) {
    if (root == NULL) {
        return;
    }
    // 访问当前节点并执行操作
    printf("%d ", root->val);
    // 递归遍历左子树
    PreOrder(root->left);
    // 递归遍历右子树
    PreOrder(root->right);
}

```

37、递归中序

思路

1. 如果当前节点为空，则返回。
2. 递归调用函数，遍历左子树。
3. 访问当前节点，并执行相应操作。
4. 递归调用函数，遍历右子树。

实现

```

#include <stdio.h>

// 树节点的定义
typedef struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
};

// 递归中序遍历函数
void InOrder(struct TreeNode* root) {
    if (root == NULL) {
        return;
    }
    // 递归遍历左子树
    InOrder (root->left);

    // 访问当前节点
    printf("%d ", root->val);

    // 递归遍历右子树
    InOrder (root->right);
}

```

38、递归后序遍历

思路

1. 如果当前节点为空，则返回。
2. 递归调用函数，遍历左子树。
3. 递归调用函数，遍历右子树。
4. 访问当前节点，并执行相应操作。

实现

```
#include <stdio.h>

// 树节点的定义
typedef struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
};

// 递归后序遍历函数
void PostOrder(struct TreeNode* root) {
    if (root == NULL) {
        return;
    }
    // 递归遍历左子树
    PostOrder (root->left);
    // 递归遍历右子树
    PostOrder (root->right);
    // 访问当前节点
    printf("%d ", root->val);
}
```

39、非递归层次遍历

思路

1. 初始化一个辅助队列。
2. 根节点入队。
3. 若队列非空，则队头节点出队，访问该节点，并将其左、右孩子插入队尾。
4. 重复步骤3，直至队列为空。

实现

```
#include<stdio.h>

// 定义二叉树节点
typedef struct BiTNode {
    Element data;
    BiTNode *lchild, * rchild;
}BiTNode, *BiTree;

// 定义链表节点
typedef struct LinkNode {
    BiTNode* data;
    LinkNode* next;
}LinkNode;

// 定义链式队列
```

```

typedef struct LinkQueue {
    linkNode* front, *rear;
}LinkQueue;

// 初始化队列（带头结点）
void InitQueue(LinkQueue& Q) {
    Q.front = Q.rear = new LinkNode(); // 初始时头尾指针都指向头节点
    Q.front->next = nullptr;
}

// 队列判空
bool QueueEmpty(linkQueue Q) {
    if(Q.front == Q.rear) return true;
    return false;
}

// 入队操作（按照王道的方式，这里采用混用的方式，c中是没有引用类型）
bool EnQueue(LinkQueue& Q, BitNode* p) { // p按值传递
    LinkNode* s = new LinkNode(); // 申请待入队节点
    s->data = p; // 将二叉树节点赋值到链表节点的数据域中
    s->next = nullptr;
    Q.rear->next = s;
    Q.rear = s;
    return true;
}

// 出队操作（按照王道的方式，这里采用混用的方式，c中是没有引用类型）
bool DeQueue(LinkQueue& Q, BitNode& p) { // p按地址传递（引用类型）
    if(QueueEmpty(Q)) return false;
    LinkNode* q = Q.front->next; // 待出队节点
    p = q->data; // 获取待出队节点的数据域，用于访问
    Q.front->next = q->next;
    if(Q.rear == q) Q.rear = Q.front; // 当最后一个节点出队时，特殊处理一下
    delete(q); // 释放空间
    return true;
}

void visit(BitNode p) {
    printf("%d", p->data) ;
}

// 层序遍历二叉树
void LevelOrder(BiTree T) {
    // 初始化一个链式队列
    LinkQueue Q;
    InitQueue(Q);
    BitNode* p; // 用来临时记录出队节点的数据域
    EnQueue(Q,T); //根节点入队
    while (!QueueEmpty(Q)) {
        DeQueue(Q, p); // 队头节点出队
        visit(p); // 访问该节点数据
        if (p->lchild) {
            EnQueue(Q, p->lchild); // 左孩子入队
        }
        if(p->rchild) {
            EnQueue(Q, p->rchild); // 右孩子入队
        }
    }
}

```

}

40、二叉树所有叶子结点个数

1. 计算二叉树中所有叶子节点的个数是一个常见的问题，可以通过递归遍历二叉树来解决。递归是一种自底向上的思想，对于每个节点，如果它是叶子节点，则返回 1，否则返回左子树和右子树的叶子节点个数之和。

- 时间复杂度： $O(n)$
- 空间复杂度： $O(n)$

实现

```
// 递归计算叶子节点个数
int countLeafNode(BiTree T) {
    if(!T) return 0; // 空树
    if(!T->left && !T->right) return 1;
    return countLeafNode(T->left) + countLeafNode(T->right);
}
```

41、判断二叉树是否为二叉排序树

1. 二叉排序树的中序遍历结果应为升序排列，利用这个性质进行判断。
2. 在遍历过程中记录前一个节点的值，与当前节点的值进行比较，判断是否满足二叉排序树的条件。

- 时间复杂度： $O(n)$
- 空间复杂度： $O(n)$

思路

1. 对二叉树进行中序遍历，同时记录遍历的前一个节点的值。
2. 检查当前遍历的节点值是否大于前一个节点的值，如果大于，则继续遍历；如果小于或等于，则说明不满足二叉排序树的条件，返回 false。
3. 遍历完成后，如果没有发现不满足条件的节点，说明二叉树是二叉排序树，返回 true。

实现

```
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

// 辅助函数，中序遍历二叉树并检查是否属于升序序列（参考王道风格，这里&属于混用）
bool InOrder(TreeNode* root, int& prev) {
    if(!root) return true;

    // 遍历左子树
    if(!InOrder(root->left, prev)) return false;

    // 检查当前节点值是否大于前一个节点值
    if (root->val <= prev) return false;
    prev = root->val;

    // 遍历右子树
    return InOrder(root->right, prev);
}
```

```
// 判断是否为二叉排序树
bool isBST(TreeNode* root) {
    if (root) {
        int prev = INT_MIN; // 初始化为最小值
        Inorder(root, prev);
    } else return false; // 空树
}
```

42、判断二叉树是否平衡二叉树

思路

1. 递归检查每个节点的左右子树的高度差，如果有任何一个节点的左右子树高度差大于1，则该树不是平衡二叉树。
2. 对每个节点，需要计算其左子树和右子树的高度，并比较高度差。
3. 使用递归进行深度优先遍历，同时计算每个节点的高度。
4. 如果遍历过程中发现任何一个节点的左右子树高度差大于1，则返回 false，表示该树不是平衡二叉树。
5. 如果遍历结束时没有发现不平衡的节点，返回 true，表示该树是平衡二叉树。

实现

```
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>

// 定义二叉树节点
typedef struct BiTNode {
    Element data; // 数据域
    BiTNode *left, *right; // 左右节点
}BiTNode, *BiTree;

// 计算二叉树高度
int TreeHight(BiTree T) {
    if(!T) return 0; // 空树高度为0
    // 递归计算左右子树的高度
    int LHight = TreeHight(T->left);
    int RHight = TreeHight(T->right);
    return LDepth > RDepth ? LDepth + 1 : RDepth + 1;
}

// 判断是否为平衡二叉树
bool IsBalanced(BiTree T) {
    if(root = Null) return true;

    int LHight = TreeHight(T->left); // 计算左子树高度
    int RHight = TreeHight(T->right); // 计算右子树高度

    if(abs(LHight - RHight) > 1) return false;
    return IsBalanced(T->left) && IsBalanced(T->right);
}
```

43、判断二叉树是否为完全二叉树

思路

1. 针对给定的二叉树，我们可以使用层序遍历（BFS）来逐层遍历树的节点。
2. 在层序遍历过程中，遇到以下情况时，二叉树不是完全二叉树：
 1. 如果一个节点有右子树，但没有左子树，那么它不是完全二叉树。
 2. 如果一个节点只有左子树或没有子树，那么它之后的所有节点都应该是叶子节点，否则不是完全二叉树。
3. 如果遍历完成后没有发现上述情况，那么二叉树是完全二叉树。

实现

```
#include <stdbool.h>
#include <stdlib.h>
#include <stdio.h>

// 二叉树节点的定义
typedef struct TreeNode {
    int val;
    struct TreeNode* left;
    struct TreeNode* right;
}TreeNode;

// 判断二叉树是否为完全二叉树
bool isCompleteTree(struct TreeNode* root) {
    if (root == NULL) {
        return true;
    }

    bool leafFound = false;
    struct TreeNode* queue[1000];
    int front = 0, rear = 0;
    queue[rear++] = root; // 入队

    while (front < rear) {
        TreeNode* node = queue[front++]; // 出队加赋值

        // 如果之前已经遇到过叶子节点，但当前节点不是叶子节点，说明不是完全二叉树
        if (leafFound && (node->left != NULL || node->right != NULL)) {
            return false;
        }

        // 如果只有右子树没有左子树，说明不是完全二叉树
        if (node->left == NULL && node->right != NULL) {
            return false;
        }

        // 将当前节点的子节点加入队列
        if (node->left != NULL) {
            queue[rear++] = node->left;
        }

        if (node->right != NULL) {
            queue[rear++] = node->right;
        }

        // 如果当前节点只有左子树无子树，后面的节点都应该是叶子节点
        if (node->left == NULL || node->right == NULL) {
            leafFound = true;
        }
    }
}
```

```

    }
}

return true;
}

```

44、输出先序遍历的第k个结点的值

思路

1. 定义一个计数器count,用于记录当前遍历到第几个节点。
2. 进行先序遍历：顺序为：根节点->左子树->右子树。
3. 遍历每个节点的时候count++, 当count等于k的时候，输出该节点的值。

算法

```

#include <stdio.h>
#include <stdlib.h>

// 二叉树节点的定义
typedef struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
}TreeNode;

// 定义全局计数器
int count = 0;

// 先序遍历函数
void PreOrder(TreeNode* root, int k) {
    if (!root) return;

    // 访问当前节点
    count++;
    if(count == k) printf("第%d个节点的值为: %d\n",k, root->val);

    // 递归左子树
    PreOrder(root->left, k);

    // 递归右子树
    PreOrder(root->right, k);
}

```

45、找出二叉树中最大值的点

思路

1. 初始化最大值变量为负无穷大，最大值节点指针为空。
2. 从根节点开始进行深度优先搜索：
 1. 如果当前节点的值大于最大值，则更新最大值和最大值节点指针。
 2. 递归调用遍历左子树。
 3. 递归调用遍历右子树。
3. 返回最大值节点指针。

实现


```

#include <stdio.h>
#include <limits.h>

// 二叉树节点结构
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
};

// 深度有限搜索函数
void DFS(TreeNode* node, TreeNode** maxNode, int* maxVal) {
    if (!node) return;

    // 寻找最大值节点
    if (node->val > *maxVal){
        *maxVal = node->val;
        *maxNode = node;
    }

    // 递归左右子树
    DFS(node->left, maxNode, maxVal);
    DFS(node->right, maxNode, maxVal);
}

// 找出二叉树中的最大值的节点
TreeNode* findMaxNode(struct TreeNode* root) {
    if(!root) return NULL;
    struct TreeNode* maxNode = NULL; // 最大值节点
    int maxVal = INT_MIN; // 初始化为最小值

    DFS(root, &maxNode, &maxVal);
    return maxNode;
}

```

46、并查集基本操作

思路

- 初始化：创建一个并查集数据结构，每个元素初始时都是独立的集合。
- 查找操作：给定一个元素，查找其所属的集合代表元素（根节点）。
- 合并操作：给定两个元素，将它们所属的集合合并为一个集合。

实现

```

#include <stdlib.h>
#include <stdio.h>
// 并查集数据结构
#define MAX_SIZE 100
int parent[MAX_SIZE]; // 全局变量

// 初始化并查集
void initialize(int n) {
    for (int i = 0; i < n; i++) {
        parent[i] = -1; // 初始化都为独立元素
    }
}

```

```
// 查找元素所属集合的代表元素（根节点）
int find(int x) {
    while(parent[x] != -1) x = parent[x];
    return x; // 根节点
}

// 合并，将两个元素所对应的集合合并
void union(int x, int y) {
    int r1 = find(x);
    int r2 = find(y);
    if(r1 != r2) parent[r2] = r1; // 将集合1合并到集合2中
}
```

47、14真题求T的WPL算法

(13分) 二叉树的带权路径长度 (WPL) 是二叉树中所有叶结点的带权路径长度之和。给定一棵二叉树T，采用二叉链表存储，结点结构如下：

left	weight	right

其中叶结点的weight域保存该结点的非负权值。设root为指向T的根结点的指针，请设计求T的WPL的算法，要求：

- 1) 给出算法的基本设计思想。
- 2) 使用C或C++语言，给出二叉树结点的数据类型定义。
- 3) 根据设计思想，采用C或C++语言描述算法，关键之处给出注释。

思路

1. 遍历树的所有叶子节点，计算每个叶子节点的路径长度（从根节点到该叶子节点的路径上的边数）。
2. 将每个叶子节点的权值乘以其路径长度，累加得到带权路径长度。

实现

```
#include <iostream>

typedef struct TreeNode {
    int weight;
    TreeNode* left;
    TreeNode* right;
}TreeNode;

// 计算带权路径长度
int calculateWPL(TreeNode* root, int depth) {
    if(root == NULL) {
        return 0;
    }
```

```

// 叶子节点，返回权值乘以路径长度
if(root->left == NULL && root->right == NULL) {
    return root->weight * depth;
}
// 非叶子节点递归计算wpl
return calculateWPL(root->left, depth + 1) + calculateWPL(root->right, depth
+ 1);
}

```

48、17年真题二叉树转中缀表达式

💡 题目描述:

请设计一个算法，将给定的表达式树（二叉树）转换为等价的中缀表达式（通过括号反映操作符的计算次序）并输出。例如，当下列两颗表达式树作为算法的输入时：

输出的等价中缀表达式分别为 $(a+b)(c-d)$ 和 $(a*b)+(-(c-d))$ 。

要求：

- (1) 给出算法的基本设计思想。
- (2) 根据设计思想，采用C或者C++语言描述算法，关键之处给出注释

思路

1. 从根节点开始递归遍历树。
2. 如果当前节点是叶子节点，输出该节点的值。
3. 如果当前节点是操作符节点（加法、减法、乘法、除法等），则输出左括号，然后递归处理左子树和右子树。
4. 最后，输出操作符节点的值，再输出右括号。

实现

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// 定义二叉树节点结构
struct TreeNode {
    char val[100];
    struct TreeNode* left;
    struct TreeNode* right;
};

// 递归将表达式树转换为中缀表达式
void expressionToInfix(struct TreeNode* root) {
    if (root == NULL) {
        return;
    }
    // 如果是叶子节点，直接输出值
    if (root->left == NULL && root->right == NULL) {
        printf("%s", root->val);
    } else {
        // 否则，输出左括号，递归处理左子树和右子树，输出操作符，输出右括号
        printf("(");
        expressionToInfix(root->left);
        printf("%s", root->val);
    }
}

```

```

        expressionToInfix(root->right);
        printf(")");
    }
}

```

49、图的初始化

```

1. void CreateGraph(MGraph &G)
2. {
3.     int i,j,k,weight;
4.     int gn,ge;
5.     printf("请输入顶点数和边数:");
6.     scanf("%d %d",&Gn,&Ge);
7.     printf("请输入顶点:");
8.     for(i = 0; i < Gn; i++)
9.         scanf("%d",&vexs[i]);
10.    for(i = 0; i < Ge; i++)
11.    {
12.        for(j = 0; j < Gn; j++)
13.        {
14.            G.AdjMatrix[i][j] = infinity;
15.        }
16.    }
17.    printf("请输入边的顶点下标i,j以及权重w :");
18.    for(k = 0; k < Ge; k++)
19.    {
20.        scanf("%d %d %d",&i,&j,&weight);
21.        G.AdjMatrix[i][j] = weight;
22.        G.AdjMatrix[j][i] = G.AdjMatrix[i][j];
23.    }
24. }

```

50、图的广度优先遍历

```

1. bool visit[MAX_VERTEX_NUM];
2. void BFST(Graph G)          //访问标记数组
3. {
4.     for(i = 0; i < G.vexnum; i++)
5.         visit[i] = false;    //访问标记数组初始化
6.     InitQueue(Q);            //初始化辅助队列
7.     for(i = 0; i < G.vexnum; i++) //从0号结点开始遍历
8.     {
9.         if(!visit[i])
10.            BFS(G,i);
11.    }
12. }
13. void BFS(Graph G,int v)
14. {
15.     visit(v);                //访问初始结点v
16.     visited(v) = true;       //置访问标记
17.     EnQueue(Q,v);
18.     while(!IsEmpty(Q))
19.     {
20.         DeQueue(Q,v);
21.         //检测v所有邻接结点

```

```

22.         for(w = FirstNeighbor(G,v), w >= 0; w = NextNeighbor(G,v,w))
23.             if(!visited[w]) //w为v当前未访问的邻接结点
24.             {
25.                 visit(w);
26.                 visited[w] = true;
27.                 EnQueue(Q,w);
28.             }
29.     }
30. }

```

51、图的深度遍历

```

1.  bool visit[MAX_VERTEX_NUM];
2.  void DFST(Graph G) //访问标记数组
3.  {
4.      for(i = 0; i < G.vexnum; i++)
5.          visit[i] = false; //访问标记数组初始化
6.      for(i = 0; i < G.vexnum; i++) //从0号结点开始遍历
7.      {
8.          if(!visit[i])
9.              DFS(G,i);
10.     }
11. }
12. void DFS(Graph G,int v)
13. {
14.     visit(v); //访问初始结点v
15.     visited(v) = true; //置访问标记
16.     for(w = FirstNeighbor(G,v), w >= 0; w = NextNeighbor(G,v,w))
17.         if(!visited[w]) DFS(G,w); //w为v当前未访问的邻接结点
18. }

```

52、计算机顶点i的出度

```

1.  int getOutDegree(MGraph G,VertexType i)
2.  {
3.      int j,sum = 0;
4.      for(j = 0; j < G.vernum; j++)
5.      {
6.          if(AdjMatrix[i][j] == 1)
7.          {
8.              sum++;
9.          }
10.     }
11.     return sum;
12. }

```

53、计算机顶点i的入读

```

1.  int getInDegree(MGraph G, vertexType i)
2.  {
3.      int j, sum = 0;
4.      for(j = 0; j < G.vernum; j++)
5.      {
6.          if(AdjMatrix[j][i] == 1)
7.          {
8.              sum++;
9.          }
10.     }
11.     return sum;
12. }

```

54、图用邻接表表示，设计一个算法，输出从顶点 V_i 到顶点 V_j 的所有简单路径

```

/*
本题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)。
*/

void FindPath(AGraph *G, int u, int v, int path[], int d){
    int w, i;
    ArcNode *p;
    d++;                      //路径长度增1
    path[d]=u;                //将当前顶点添加到路径中
    visited[u]=1;              //置已访问标记
    if(u==v)                   //找到一条路径则输出
        print(path[]);        //输出路径上的结点
    p=G->adjlist[u].firstarc; //p指向u的第一个相邻点
    while(p!=NULL){
        w=p->adjvex;           //若顶点w未访问，递归访问它
        if(visited[w]==0)
            FindPath(G, w, v, path, d);
        p = p->nextarc;        //p指向u的下一个相邻点
    }
}

```

55、23真题图算法

对于一个有向图，如果一个顶点的出度大于入度，则这个顶点称为KJ顶点。有向图用邻接矩阵存储，数据结构定义如下：

```

typedef struct {
    int numVertices, numEdges; //顶点数、边数
    char verticesList[MAXV]; //顶点表
    int Edge[MAXV][MAXV]; //邻接矩阵
}MGraph;

```

要求实现函数 `int printVertices(MGraph G)`，输出有向图中所有KJ顶点，并返回KJ顶点的总数。

要求：

- (1) 说明算法思想；
- (2) 用C/C++实现算法

思路

遍历有向图的所有顶点，并统计各顶点的入度（矩阵第*i*行元素个数）和出度（矩阵第*i*列的元素个数），输出出度大于入度的K顶点，使用count变量统计K顶点的总数。

实现

```
int printVertices(MGraph G){
    int count = 0; //K顶点总数
    for (int i = 0; i < G.numVertices; ++i) {
        int in_degree = 0, out_degree = 0;
        for (int j = 0; j < G.numVertices; ++j) {
            if(G.Edge[i][j] > 0) out_degree++;
        }
        for (int j = 0; j < G.numVertices; ++j) {
            if(G.Edge[j][i] > 0) in_degree++;
        }
        if(out_degree > in_degree){
            count++;
            std::cout<<"K顶点为: "<<G.VerticesList[i]<<"\n";
        }
    }
    return count;
}
```