1. 在计算机中，算法是指什么？
答案：解题方案的准确而完整的描述。
2. 在下列选项中，哪个不是一个算法一般应该具有的基本特征？
说明：算法的四个基本特征是：可行性、确定性、有穷性和拥有足够的情报。
答案：无穷性。
3. 算法一般都可以用哪几种控制结构组合而成？
答案：顺序、选择、循环。
4. 算法的时间复杂度是指？
答案：算法执行过程中所需要的基本运算次数。
5. 算法的空间复杂度是指？
答案：执行过程中所需要的存储空间。
6. 算法分析的目的是？
答案：分析算法的效率以求改进。
7. 下列叙述正确的是（C）
A. 算法的执行效率与数据的存储结构无关
B. 算法的空间复杂度是指算法程序中指令（或语句）的条数
C. 算法的有穷性是指算法必须能在执行有限个步骤之后终止
D. 算法的时间复杂度是指执行算法程序所需要的时间
8. 数据结构作为计算机的一门学科，主要研究什么？
答案：主要研究数据的逻辑结构、对各种数据结构进行的运算，以及数据的存储结构。
9. 数据结构中与所使用的计算机无关的是数据的（C）
A. 存储结构 B. 物理结构
C. 逻辑结构 D. 物理和存储结构
10. 下列叙述中，错误的是（B）
A. 数据的存储结构与数据处理的效率密切相关
B. 数据的存储结构与数据处理的效率无关
C. 数据的存储结构在计算机中所占的空间不一定是连续的
D. 一种数据的逻辑结构可以有多种存储结构
11. 数据的存储结构是指什么？
答案：数据的逻辑结构在计算机中的表示。
12. 数据的逻辑结构是指？
答案：反映数据元素之间逻辑关系的数据结构。
13. 根据数据结构中各数据元素之间前后件关系的复杂程度，一般将数据结构分为？
答案：线性结构和非线性结构。
14. 下列数据结构具有记忆功能的是（C）
A. 队列
B. 循环队列
C. 栈
D. 顺序表
15. 下列数据结构中，按先进后出原则组织数据的是（B）
A. 线性链表
B. 栈
C. 循环链表
D. 顺序表

16. 递归算法一般需要利用什么实现？
答案：队列
17. 下列关于栈的叙述中正确的是（D）
A．在栈中只能插入数据
B．在栈中只能删除数据
C．栈是先进先出的线性表
D．栈是先进后出的线性表
18. 由两个栈共享一个存储空间的好处是？
答案：节省存储空间，降低上溢发生的机率。
19. 下列关于队列的叙述中正确的是（C）
A．在队列中只能插入数据
B．在队列中只能删除数据
C．队列是先进先出的线性表
D．队列是先进后出的线性表
20. 下列叙述中，正确的是（D）
A．线性链表中的各元素在存储空间中的位置必须是连续的
B．线性链表中的表头元素一定存储在其他元素的前面
C．线性链表中的各元素在存储空间中的位置不一定是连续的，但表头元素一定存储在其他元素的前面
D．线性链表中的各元素在存储空间中的位置不一定是连续的，且各元素的存储顺序也是任意的
21. 下列叙述中正确的是（A）
A．线性表是线性结构
B．栈与队列是非线性结构
C．线性链表是非线性结构
D．二叉树是线性结构
22. 线性表 L＝（a1,a2,a3,……ai,……an），下列说法正确的是（D）
A．每个元素都有一个直接前件和直接后件
B．线性表中至少要有一个元素
C．表中诸元素的排列顺序必须是由小到大或由大到小
D．除第一个元素和最后一个元素外，其余每个元素都有一个且只有一个直接前件和直接后件
23. 线性表若采用链式存储结构时，要求内存中可用存储单元的地址怎么样？
答案：连续不连续都可以。
24. 链表不具有的特点是（B）
A．不必事先估计存储空间
B．可随机访问任一元素
C．插入删除不需要移动元素
D．所需空间与线性表长度成正比
25. 在（D）中，只要指出表中任何一个结点的位置，就可以从它出发依次访问到表中其他所有结点。
A．线性单链表
B．双向链表
C．线性链表

D．循环链表
26. 以下数据结构属于非线性数据结构的是（C）
A．队列
B．线性表
C．二叉树
D．栈
27. 树是结点的集合，它的根结点数目是多少？
答案：有且只有 1。
28. 在一棵二叉树上第 8 层的结点数最多是？
答案：128
29. 在深度为 5 的满二叉树中，叶子结点的个数为？
答案：16
30. 在深度为 5 的满二叉树中，共有多少个结点？
答案：31
31. 设一棵完全二叉树共有 699 个结点，则在该二叉树中的叶子结点数为？
答案：350
说明：完全二叉树总结点数为 N，若 N 为奇数，则叶子结点数为（N+1）/2；若 N 为偶数，则叶子结点数为 N/2。
32. 设有下列二叉树，对此二叉树中序遍历的结果是（B）
A．ABCDEF
B．DBEAFC
C．ABDECF
D．DEBFCA
33. 若某二叉树的前序遍历访问顺序是 abdgcefh，中序遍历访问顺序是 dgbaechf，则其后序遍历的结点访问顺序是？
答案：gdbehfca
34. 串的长度是？
答案：串中所含字符的个数。
35. 设有两个串 p 和 q，求 q 在 p 中首次出现位置的运算称做？
答案：模式匹配。
36. N 个顶点的连通图中边的条数至少为？
答案：N-1
37. N 个顶点的强连通图的边数至少有？
答案：N
38. 对长度为 n 的线性表进行顺序查找，在最坏情况下所需要的比较次数为？
答案：N
39. 最简单的交换排序方法是？
答案：冒泡排序
40. 假设线性表的长度为 n，则在最坏情况下，冒泡排序需要的比较次数为？
答案：n(n-1)/2
41. 在待排序的元素序列基本有序的前提下，效率最高的排序方法是？
答案：冒泡排序
42. 在最坏情况下，下列顺序方法中时间复杂度最小的是？
答案：堆排序

43. 希尔排序法属于？
答案：插入类排序
44. 堆排序法属于？
答案：选择类排序
45. 在下列几种排序方法中，要求内存量最大的是？
答案：归并排序
46. 已知数据表 A 中每个元素距其最终位置不远，为节省时间，应采用？
答案：直接插入排序

（1）交换类排序法交换类排序法是指借助数据元素之间的互相交换进行排序的一种方法。冒泡排序法与快速排序法都属于交换类排序方法。冒泡排序法是一种最简单的交换类排序方法，它是通过相邻数据元素的交换逐步将线性表变成有序。假设线性表的长度为 n，则在最坏情况下，冒泡排序需要经过 n/2 遍的从前往后的扫描和 n/2 遍的从后往前的扫描，需要的比较次数为 n(n–1)/2。但这个工作量不是必需的，一般情况下要小于这个工作量。快速排序法也是一种交换类的排序方法，但由于它比冒泡排序法的速度快，因此称之为快速排序法。其关键是对线性表进行分割，以及对各分割出的子表再进行分割。
（2）插入类排序法插入类排序法主要有简单插入排序法和希尔排序法。简单插入排序法，是指将无序序列中的各元素依次插入到已经有序的线性表中。在这种排序方法中，每一次比较后最多移掉一个逆序，因此，这种排序方法的效率与冒泡排序法相同。在最坏情况下，简单插入排序需要 n(n–1)/2 次比较。希尔排序法对简单插入排序做了较大的改进。它是将整个无序序列分割成若干小的子序列分别进行插入排序。希尔排序的效率与所选取的增量序列有关。在最坏情况下，希尔排序所需要的比较次数为 O(n1.5)。
（3）选择类排序选择类排序主要有简单选择类排序法和堆排序法。简单选择排序法的基本思想是：扫描整个线性表，从中选出最小的元素，将它交换到表的最前面（这是它应有的位置）；然后对剩下的子表采用同样的方法，直到子表空为止。对于长度为 n 的线性表，在最坏情况下需要比较 n(n–1)/2 次。堆排序法也属于选择类排序法。具有 n 个元素的序列（h1, h2, …, hn），当且仅当满足条件：　或 (i=1, 2, …, n/2）时称之为堆。可见，堆顶元素（即第一个元素）必为最大项。堆排序的方法对于规模较小的线性表并不适合，但对于较大规模的线性表来说是很有效的。在最坏情况下，堆排序需要比较的次数为 O(nlog2n)。

第一章 数据结构与算法

一.算法的基本概念

计算机解题的过程实际上是在实施某种算法，这种算法称为计算机算法。

1.算法的基本特征：可行性，确定性，有穷性，拥有足够的情报。

2.算法的基本要素：算法中对数据的运算和操作、算法的控制结构。

3.算法设计的基本方法：列举法、归纳法、递推、递归、减半递推技术、回溯法。

4.算法设计的要求：正确性、可读性、健壮性、效率与低存储量需求

二.算法的复杂度

1.算法的时间复杂度：指执行算法所需要的计算工作量

2.算法的空间复杂度：执行这个算法所需要的内存空间

三.数据结构的定义

1.数据的逻辑结构：反映数据元素之间的关系的数据元素集合的表示。数据的逻辑结构包括集合、线形结构、树形结构和图形结构四种。

2.数据的存储结构：数据的逻辑结构在计算机存储空间种的存放形式称为数据的存储结构。常用的存储结构有顺序、链接、索引等存储结构。

四.数据结构的图形表示：

在数据结构中，没有前件的结点称为根结点；没有后件的结点成为终端结点。插入和删除是对数据结构的两种基本运算。还有查找、分类、合并、分解、复制和修改等。

五.线性结构和非线性结构

根据数据结构中各数据元素之间前后件关系的复杂程度，一般将数据结构分为两大类型：线性结构和非线性结构。

线性结构：非空数据结构满足：有且只有一个根结点；每个结点最多有一个前件，最多只有一个后件。非线性结构：如果一个数据结构不是线性结构，称之为非线性结构。

常见的线性结构：线性表、栈、队列

六.线性表的定义

线性表是 n 个元素构成的有限序列（A1，A2，A3……）。表中的每一个数据元素，除了第一个以外，有且只有一个前件。除了最后一个以外有且只有一个后件。即线性表是一个空表，或可以表示为（a1,a2,……an），其中 ai(I=1,2,……n)是属于数据对象的元素，通常也称其为线性表中的一个结点。

非空线性表有如下一些特征：

（1）有且只有一个根结点 a1,它无前件；

（2）有且只有一个终端结点 an，它无后件；

（3）除根结点与终端结点外，其他所有结点有且只有一个前件，也有且只有一个后件。线性表中结点的个数 n 称为线性表的长度。当 n=0 时称为空表。

七.线性表的顺序存储结构

线性表的顺序表指的是用一组地址连续的存储单元依次存储线性表的数据元素。

线性表的顺序存储结构具备如下两个基本特征：

1.线性表中的所有元素所占的存储空间是连续的；

2.线性表中各数据元素在存储空间中是按逻辑顺序依次存放的。

即线性表逻辑上相邻、物理也相邻，则已知第一个元素首地址和每个元素所占字节数，则可求出任一个元素首地址。

假设线性表的每个元素需占用 K 个存储单元，并以所占的第一个单元的存储地址作为数据元素的存储位置。则线性表中第 i+1 个数据元素的存储位置 LOC(ai+1)和第 i 个数据元素的存储位置 LOC(ai)之间满足下列关系:

LOC(ai+1)=LOC(ai)+K

LOC(ai)=LOC(a1)+(i-1)*K　①

其中，LOC(a1)是线性表的第一个数据元素 a1 的存储位置，通常称做线性表的起始位置或基地址。

因为在顺序存储结构中，每个数据元素地址可以通过公式①计算得到，所以线性表的顺序存储结构是随机存取的存储结构。

在线性表的顺序存储结构下，可以对线性表做以下运算：

插入、删除、查找、排序、分解、合并、复制、逆转

八.顺序表的插入运算

线性表的插入运算是指在表的第 I 个位置上，插入一个新结点 x，使长度为 n 的线性表（a1,a2 …ai…an）变成长度为 n+1 的线性表(a1,a2…x,ai…an).

该算法的时间主要花费在循环的结点后移语句上，执行次数是 n-I+1。

当 I=n+1,最好情况，时间复杂度 o(1)　当 I=1, 最坏情况，时间复杂度 o(n)

算法的平均时间复杂度为 o(n)

九.顺序表的删除运算

线性表的删除运算是指在表的第 I 个位置上，删除一个新结点 x，使长度为 n 的线性表（a1,a2 …ai…an）变成长度为 n-1 的线性表(a1,a2…ai-1,ai+1…an).

当 I=n,时间复杂度 o(1),当 I=1,时间复杂度 o(n) ,平均时间复杂度为 o(n)

十.栈及其基本运算

1.什么是栈？　栈实际上也是一个线性表，只不过是一种特殊的线性表。栈是只能在表的一端进行插入和删除运算的线性表，通常称插入、删除这一端为栈顶（TOP），另一端为栈底（BOTTOM）。当表中没有元素时称为空栈。栈顶元素总是后被插入的元素，从而也是最先被删除的元素；栈底元素总是最先被插入的元素，从而也是最后才能被删除的元素。

假设栈 S=（a1,a2,a3,……an），则 a1　　称为栈底元素，an 称为栈顶元素。栈中元素按 a1,a2,a3……an 的次序进栈，退栈的第一个元素应该是栈顶元素。即后进先出。

2.栈的顺序存储及其运算

用 S（1：M）作为栈的顺序存储空间。M 为栈的最大容量。

栈的基本运算有三种：入栈、退栈与读栈顶元素。

入栈运算：在栈顶位置插入一个新元素。

首先将栈顶指针进一（TOP+1），然后将新元素插入到栈顶指针指向的位置。

退栈运算：指取出栈顶元素并赋给一个指定的变量。

首先将栈顶元素赋给一个指定的变量，然后将栈顶指针退一（TOP-1）

读栈顶元素：将栈顶元素赋给一个指定的变量。栈顶指针不会改变。

十一.队列及其基本运算

1.什么是队列

队列是只允许在一端删除，在另一端插入的顺序表，允许删除的一端叫做对头，允许插入的一端叫做对尾。

队列的修改是先进先出。往队尾插入一个元素成为入队运算。从对头删除一个元素称为退队运算。

2.循环队列及其运算

在实际应用中，队列的顺序存储结构一般采用循环队列的形式。所谓循环队列，就是将队列存储空间的最后一个位置绕到第一个位置，形成逻辑上的环状空间。在循环队列中，，用队尾指针 rear 指向队列中的队尾元素，用排头指针 front 指向排头元素的前一个位置，因此，从排头指针 front 指向的后一个位置直到队尾指针　rear 指向的位置之间所有的元素均

为队列中的元素。
在实际使用循环队列时，为了能区分队满还是队列空，通常需要增加一个标志 S：
队列空，则 S=0，rear=front=m　队列满，则 S=1，rear=front=m
循环队列主要有两种基本运算：入队运算和退队运算
n　入队运算
指在循环队列的队尾加入一个新元素，首先 rear=rear+1,当 rear=m+1 时，置 rear=1,然后将新元素插入到队尾指针指向的位置。当 S=1，rear=front,说明队列已满，不能进行入队运算，称为“上溢”。
n　退队运算
指在循环队列的排头位置退出一个元素并赋给指定的变量。首先 front=front+1,并当 front=m+1 时，置 front=1,然后将排头指针指向的元素赋给指定的变量。当循环队列为空 S=0，不能进行退队运算，这种情况成为“下溢”。
十二.线性单链表的结构及其基本运算
1.线性单链表的基本概念
一组任意的存储单元存储线性表的数据元素，因此，为了表示每个数据元素 ai 与其直接后继数据元素 ai+1 之间的逻辑关系，对数据元素 ai 来说，除了存储其本身的信息之外，还需存储一个指示其直接后继的信息（即直接后继的存储位置）。这两部分信息组成数据元素 ai 的存储映象，成为结点。它包括两个域：其中存储数据元素信息的域称为数据域，存储直接后继存储位置的域称为指针域。指针域中存储的信息称做指针或链。N 个结点链结成一个链表，即为线性表(a1, a2,……,an)的链式存储结构。又由于此链表的每个结点中只包含一个指针域，故又称线性链表或单链表。
有时，我们在单链表的第一个结点之前附设一个结点，称之为头结点，它指向表中第一个结点。头结点的数据域可以不存储任何信息，也可存储如线性表的长度等类的附加信息，头结点的指针域存储指向第一个结点的指针（即第一个元素结点的存储位置）。
在单链表中，取得第 I 个数据元素必须从头指针出发寻找，因此，单链表是非随机存取的存储结构　链表的形式：单向，双向
2.线性单链表的存储结构
3 带链
3.带列的栈与队列
栈也是线性表，也可以采用链式存储结构。
队列也是线性表，也可以采用链式存储结构。
十三.线性链表的基本运算 1.线性链表的插入 2.线性链表的删除
十四.双向链表的结构及其基本运算
在双向链表的结点中有两个指针域，其一指向直接后继，另一指向直接前驱。
十五.循环链表的结构及其基本运算
是另一种形式的链式存储结构，它的特点是表中最后一个结点的指针域指向头结点，整个链表形成一个环。因此，从表中任一结点出发均可找到表中其他结点。
十六.树的定义
树是一种简单的非线性结构。树型结构的特点：
1.每个结点只有一个前件，称为父结点，没有前件的结点只有一个，称为树的根结点。
2.每一个结点可以有多个后件结点，称为该结点的子结点。没有后件的结点称为叶子结点
3.一个结点所拥有的后件个数称为树的结点度
4.树的最大层次称为树的深度。
十七.二叉树的定义及其基本性质

1.二叉树是另一种树型结构，它的特点是每个结点至多只有二棵子树（即二叉树中不存在度大于 2 的结点），并且，二叉树的子树有左右之分，其次序不能任意颠倒。
2.二叉树的基本性质
①在二叉树的第 I 层上至多有 2i-1 个结点。
②深度为 k 的二叉树至多有 2k-1 个结点(k>=1)
③在任意一个二叉树中，度为 0 的结点总是比度为 2 的结点多一个；
④具有 n 个结点的二叉树，其深度至少为[log2n]+1。
一棵深度为 k 且有 2k-1 个结点的二叉树称为满二叉树。这种树的特点是每一层上的结点数都是最大结点数。
3.满二叉树与完全二叉树
满二叉树：除最后一层以外，每一层上的所有结点都有两个子结点。在满二叉树的第 K 层上有 2K-1 个结点，且深度为 M 的满二叉树右 2M-1 个结点
完全二叉树：除最后一层以外，每一层上的结点数均达到最大值；在最后一层上只缺少右边的若干结点。具有 N 个结点的完全二叉树的深度为[log2n]+1
完全二叉树总结点数为 N，
若 N 为奇数，则叶子结点数为（N+1）/2 若 N 为偶数，则叶子结点数为 N/2
4.二叉树的存储结构
二叉树通常采用链式存储结构
二叉树具有下列重要特性：
性质 1 在二叉树的第 i 层上至多有 2i-1 个结点(i≥1)。
利用归纳法容易证得此性质。
i=1 时，只有一个根结点。 显然，2i-1=20=1 是对的。
现在假定对所有的 j，1≤j<i，命题成立，即第 j 层上至多有 2j-1 个结点。那么，可以证明 j=i 时命题也成立。
由归纳假设：第 i-1 层上至多有 2i-2 个结点。由于二叉树的每个结点的度至多为 2，故在第 i 层上的最大结点数为第 i-1 层上的最大结点数的 2 倍，即 2*2i-2=2i-1。
性质 2 深度为 k 的二叉树至多有 2k-1 个结点，(k≥1)。
由性质 1 可见，深度为 k 的二叉树的最大结点数为
k k
∑(第 i 层上的最大结点数) =∑2i-1=2k -1
i=1 i=1
性质 3 对任何一棵二叉树 T，如果其终端结点数为 n0，度为 2 的结点数为 n2，则 n0=n2+1。
设 n1 为二叉树 T 中度为 1 的结点数。因为二叉树中所有结点的度均小于或等于 2 所以其结点总数为
n=n0+n1+n2 (6—1)
再看二叉树中的分支数。除了根结点外，其余结点都有一个分支进入，设 B 为分支总数，则 n=B+1。由于这些分支是由度为 1 或 2 的结点射出的，所以又有 B=n1+ 2n2。
n=n1+2*n2+1 (6—2)
于是得由式(6—1)和(6—2)得 n0=n2+1
十八.二叉树的遍历
就是遵从某种次序，访问二叉树中的所有结点，使得每个结点仅被访问一次。一般先左后右。
1.前序遍历 DLR 首先访问根结点，然后遍历左子树，最后遍历右子树。

2.中序遍历 LDR 首先遍历左子树，然后根结点，最后右子树
3.后序遍历 LRD 首先遍历左子树，然后遍历右子树，最后访问根结点。
十九.顺序查找与二分查找
1.顺序查找 在两种情况下只能用顺序查找：线性表为无序表、链式存储结构的有序表
2.二分查找 只适用于顺序存储的有序表（从小到大）。
对于长度为 N 的有序线性表，在最坏情况下，二分查找只需要比较 log2N 次，而顺序查找要比较 N 次。 排序：指将一个无序序列整理成按值非递减顺序排列的有序序列。
二十.交换类排序法
冒泡排序与快速排序法属于交换类的排序方法
1.冒泡排序法 假设线性表的长度为 N，则在最坏的情况下，冒跑排序需要经过 N/2 遍的从前往后的扫描和 N/2 遍的从后往前的扫描，需要的比较次数为 N（N-1）/2
2.快速排序法
二十一.选择类排序法 1.简单选择排序法 2.堆排序法
二十三.插入类排序法 1.简单插入排序法 2.希尔排序法

最坏情况下	最好情况下	说明
交换排序	冒泡排序	n(n-1)/2	最简单的交换排序。在待排序的元素序列基本有序的前提下，效率最高
快速排序	n(n-1)/2	O(Nlog2 N)	
插入排序	简单插入排序	n(n-1)/2	每个元素距其最终位置不远时适用
希尔排序	O(n1.5)		
选择排序	简单选择排序	n(n-1)/2	
堆排序	O(nlog2n)	适用于较大规模的线性表	

练习：
1.栈和队列的共同特点是（只允许在端点处插入和删除元素）
2.如果进栈序列为 e1,e2,e3,e4，则可能的出栈序列是（e2,e4,e3,e1）
3.栈底至栈顶依次存放元素 A、B、C、D，在第五个元素 E 入栈前，栈中元素可以出栈，则出栈序列可能是（DCBEA）
4.栈通常采用的两种存储结构是（线性存储结构和链表存储结构）
5.下列关于栈的叙述正确的是（D）
A.栈是非线性结构 B.栈是一种树状结构 C.栈具有先进先出的特征 D.栈有后进先出的特征
6.链表不具有的特点是（B）A.不必事先估计存储空间 B.可随机访问任一元素
C.插入删除不需要移动元素 D.所需空间与线性表长度成正比
7.用链表表示线性表的优点是（便于插入和删除操作）
8.在单链表中，增加头结点的目的是（方便运算的实现）
9.循环链表的主要优点是（从表中任一结点出发都能访问到整个链表）
10.线性表 L＝（a1,a2,a3,……ai,……an），下列说法正确的是（D）
A.每个元素都有一个直接前件和直接后件 B.线性表中至少要有一个元素
C.表中诸元素的排列顺序必须是由小到大或由大到小
D.除第一个和最后一个元素外，其余每个元素都有一个且只有一个直接前件和直接后件
11.线性表若采用链式存储结构时，要求内存中可用存储单元的地址（D）
A.必须是连续的 B.部分地址必须是连续的 C.一定是不连续的 D.连续不连续都可以
12.线性表的顺序存储结构和线性表的链式存储结构分别是（随机存取的存储结构、顺序存取的存储结构）
13.树是结点的集合，它的根结点数目是（有且只有 1）

14.在深度为 5 的满二叉树中，叶子结点的个数为（31）
15.具有 3 个结点的二叉树有（5 种形态）
16.设一棵二叉树中有 3 个叶子结点，有 8 个度为 1 的结点，则该二叉树中总的结点数为（13）
17.已知二叉树后序遍历序列是 dabec，中序遍历序列是 debac，它的前序遍历序列是（cedba）
18.已知一棵二叉树前序遍历和中序遍历分别为 ABDEGCFH 和 DBGEACHF，则该二叉树的后序遍历为（DGEBHFCA）
19.若某二叉树的前序遍历访问顺序是 abdgcefh，中序遍历访问顺序是 dgbaechf，则其后序遍历的结点访问顺序是（gdbehfca）
20.数据库保护分为：安全性控制、 完整性控制 、并发性控制和数据的恢复。
1. 在计算机中，算法是指（解题方案的准确而完整的描述）
2.在下列选项中，哪个不是一个算法一般应该具有的基本特征（无穷性）
说明：算法的四个基本特征是：可行性、确定性、有穷性和拥有足够的情报。
3. 算法一般都可以用哪几种控制结构组合而成（顺序、选择、循环）
4.算法的时间复杂度是指（算法执行过程中所需要的基本运算次数）
5. 算法的空间复杂度是指（执行过程中所需要的存储空间）
6. 算法分析的目的是（分析算法的效率以求改进）
7. 下列叙述正确的是（C）
A．算法的执行效率与数据的存储结构无关
B．算法的空间复杂度是指算法程序中指令（或语句）的条数
C．算法的有穷性是指算法必须能在执行有限个步骤之后终止
D．算法的时间复杂度是指执行算法程序所需要的时间
8.数据结构作为计算机的一门学科，主要研究数据的逻辑结构、对各种数据结构进行的运算，以及（数据的存储结构）
9. 数据结构中，与所使用的计算机无关的是数据的（C）
A．存储结构 B．物理结构 C．逻辑结构 D．物理和存储结构
10. 下列叙述中，错误的是（B）
A．数据的存储结构与数据处理的效率密切相关
B．数据的存储结构与数据处理的效率无关
C．数据的存储结构在计算机中所占的空间不一定是连续的
D．一种数据的逻辑结构可以有多种存储结构
11. 数据的存储结构是指（数据的逻辑结构在计算机中的表示）
12. 数据的逻辑结构是指（反映数据元素之间逻辑关系的数据结构）
13. 根据数据结构中各数据元素之间前后件关系的复杂程度，一般将数据结构分为（线性结构和非线性结构）
14. 下列数据结构具有记忆功能的是（C）A．队列 B．循环队列 C．栈 D．顺序表
15. 下列数据结构中，按先进后出原则组织数据的是（B）
A．线性链表 B．栈 C．循环链表 D．顺序表
16. 递归算法一般需要利用（队列）实现。
17. 下列关于栈的叙述中正确的是（D）A．在栈中只能插入数据 B．在栈中只能删除数据
C．栈是先进先出的线性表 D．栈是先进后出的线性表
18. 栈底至栈顶依次存放元素 A、B、C、D，在第五个元素 E 入栈前，栈中元素可以出栈，则出栈序列可能是（DCBEA）

19.如果进栈序列为 e1,e2,e3,e4，则可能的出栈序列是（e2,e4,e3,e1）
20. 由两个栈共享一个存储空间的好处是（节省存储空间，降低上溢发生的机率）
21. 应用程序在执行过程中，需要通过打印机输出数据时，一般先形成一个打印作业，将其存放在硬盘中的一个指定（队列）中，当打印机空闲时，就会按先来先服务的方式从中取出待打印的作业进行打印。
22.下列关于队列的叙述中正确的是（C）A．在队列中只能插入数据 B．在队列中只能删除数据 C．队列是先进先出的线性表 D．队列是先进后出的线性表
23.下列叙述中，正确的是（D）A．线性链表中的各元素在存储空间中的位置必须是连续的 B．线性链表中的表头元素一定存储在其他元素的前面 C．线性链表中的各元素在存储空间中的位置不一定是连续的，但表头元素一定存储在其他元素的前面 D．线性链表中的各元素在存储空间中的位置不一定是连续的，且各元素的存储顺序也是任意的
24.下列叙述中正确的是（A）A．线性表是线性结构 B．栈与队列是非线性结构
C．线性链表是非线性结构 D．二叉树是线性结构
25. 线性表 L＝（a1,a2,a3,……ai,……an），下列说法正确的是（D）
A．每个元素都有一个直接前件和直接后件 B．线性表中至少要有一个元素
C．表中诸元素的排列顺序必须是由小到大或由大到小 D．除第一个元素和最后一个元素外，其余每个元素都有一个且只有一个直接前件和直接后件
26.线性表若采用链式存储结构时，要求内存中可用存储单元的地址（连续不连续都可以）
27. 链表不具有的特点是（B）A．不必事先估计存储空间 B．可随机访问任一元素
C．插入删除不需要移动元素 D．所需空间与线性表长度成正比
28. 非空的循环单链表 head 的尾结点（由 p 所指向），满足（p->next=head）
29.与单向链表相比，双向链表的优点之一是（更容易访问相邻结点）
30. 在（D）中，只要指出表中任何一个结点的位置，就可以从它出发依次访问到表中其他所有结点。A．线性单链表 B．双向链表 C．线性链表 D．循环链表
31. 以下数据结构属于非线性数据结构的是（C）A．队列 B．线性表 C．二叉树 D．栈
32.树是结点的集合，它的根结点数目是（有且只有 1）
33.具有 3 个结点的二叉树有（5 种形态）
34. 在一棵二叉树上第 8 层的结点数最多是（128） 注：2K-1
35. 在深度为 5 的满二叉树中，叶子结点的个数为（16） 注：2n-1
36. 在深度为 5 的满二叉树中，共有（31）个结点。 注：2n－1
37.设一棵完全二叉树共有 699 个结点，则在该二叉树中的叶子结点数为（350）
说明：完全二叉树总结点数为 N，若 N 为奇数，则叶子结点数为（N+1）/2；若 N 为偶数，则叶子结点数为 N/2。
38. 设有下列二叉树，对此二叉树中序遍历的结果是（B）
A．ABCDEF
B．DBEAFC
C．ABDECF
D．DEBFCA
39.已知二叉树后序遍历序列是 dabec，中序遍历序列 debac，它的前序遍历序列是（cedba）
40. 已知一棵二叉树前序遍历和中序遍历分别为 ABDEGCFH 和 DBGEACHF，则该二叉树的后序遍历为（DGEBHFCA）
41.若某二叉树的前序遍历访问顺序是 abdgcefh，中序遍历访问顺序是 dgbaechf，则其后序遍历的结点访问顺序是（gdbehfca）

42. 串的长度是（串中所含字符的个数）
43.设有两个串 p 和 q，求 q 在 p 中首次出现位置的运算称做（模式匹配）
44. N 个顶点的连通图中边的条数至少为（N-1）
45.N 个顶点的强连通图的边数至少有（N）
46.对长度为 n 的线性表进行顺序查找，在最坏情况下所需要的比较次数为（N）
47. 最简单的交换排序方法是（冒泡排序）
48.假设线性表的长度为 n，则在最坏情况下，冒泡排序需要的比较次数为（n(n-1)/2）
49. 在待排序的元素序列基本有序的前提下，效率最高的排序方法是（冒泡排序）
50. 在最坏情况下，下列顺序方法中时间复杂度最小的是（堆排序）
51. 希尔排序法属于（插入类排序）
52. 堆排序法属于（选择类排序）
53. 在下列几种排序方法中，要求内存量最大的是（归并排序）
54. 已知数据表 A 中每个元素距其最终位置不远，为节省时间，应采用（直接插入排序）
55. 算法的基本特征是可行性、确定性、 有穷性 和拥有足够的情报。
1.一个算法通常由两种基本要素组成：一是对数据对象的运算和操作，二是算法的控制结构。
1. 算法的复杂度主要包括时间复杂度和 空间 复杂度。
2. 实现算法所需的存储单元多少和算法的工作量大小分别称为算法的空间复杂度和时间复杂度 。
3.所谓数据处理是指对数据集合中的各元素以各种方式进行运算，包括插入、删除、查找、更改等运算，也包括对数据元素进行分析。
4.数据结构是指相互有关联的 数据元素 的集合。
5.数据结构分为逻辑结构与存储结构，线性链表属于 存储结构 。
6.数据结构包括数据的 逻辑 结构和数据的存储结构。
7. 数据结构包括数据的逻辑结构、数据的 存储结构 以及对数据的操作运算。
8.数据元素之间的任何关系都可以用 前趋和后继 关系来描述。
9.数据的逻辑结构有线性结构和非线性结构两大类。
10.常用的存储结构有顺序、链接、 索引 等存储结构。
11. 顺序存储方法是把逻辑上相邻的结点存储在物理位置 相邻 的存储单元中。
12. 栈的基本运算有三种：入栈、退栈与读栈顶元素 。
13. 队列主要有两种基本运算：入队运算与 退队运算 。
14. 在实际应用中，带链的栈可以用来收集计算机存储空间中所有空闲的存储结点，这种带链的栈称为 可利用栈 。
15.栈和队列通常采用的存储结构是 链式存储和顺序存储 。
16.当线性表采用顺序存储结构实现存储时，其主要特点是 逻辑结构中相邻的结点在存储结构中仍相邻 。
17. 循环队列主要有两种基本运算：入队运算与退队运算。每进行一次入队运算，队尾指针就 进 1 。
18.当循环队列非空且队尾指针等于对头指针时，说明循环队列已满，不能进行入队运算。这种情况称为 上溢 。
19.当循环队列为空时，不能进行退队运算，这种情况称为 下溢 。
20. 在一个容量为 25 的循环队列中，若头指针 front=16，尾指针 rear=9，则该循环队列中共有 18 个元素。注：当 rear<front 时，元素个数＝总容量－（front－rear）；
当 rear>front 时，元素个数＝rear－front。

21. 在一个容量为 15 的循环队列中，若头指针 front=6，尾指针 rear=9，则该循环队列中共有 3 个元素。
22.顺序查找一般是指在 线性表 中查找指定的元素。
23.在计算机中存放线性表，一种最简单的方法是 顺序存储 。
24.在程序设计语言中，通常定义一个 一维数组 来表示线性表的顺序存储空间。
25.在链式存储方式中，要求每个结点由两部分组成：一部分用于存放数据元素值，称为数据域，另一部分用于存放指针，称为 指针域 。其中指针用于指向该结点的前一个或后一个结点（即前件或后件）。
26.在 线性单链表中 ，每一个结点只有一个指针域，由这个指针只能找到后继结点，但不能找到前驱结点。
27. 为了要在线性链表中插入一个新元素，首先要给该元素分配一个 新结点 ，以便用于存储该元素的值。
28. 在线性链表中删除一个元素后，只需要改变被删除元素所在结点的前一个结点的 指针域 即可。
29. 用链表表示线性表的突出优点是 便于插入和删除操作 。
30. 在树形结构中，树根结点没有 前件 。
31. 在树结构中，一个结点所拥有的后件个数称为该结点的度。叶子结点的度为 0 。
32. 设一棵二叉树中有 3 个叶子结点，8 个度为 1 的结点，则该二叉树中总的结点数为 13。
33. 设一棵完全二叉树共有 739 个结点，则在该二叉树中有 370 个叶子结点。
34. 设一棵完全二叉树共有 700 个结点，则在该二叉树中有 350 个叶子结点。
35. 在先左后右的原则下，根据访问根结点的次序，二叉树的遍历可以分为三种：前序遍历、 中序 遍历和后序遍历。
36. 若串 S=”Program”，则其子串的数目是 29 。 注：n(n+1)/2+1
37. 若串 S=”MathTypes”，则其子串的数目是 46 。
38. 对长度为 n 的线性表进行插入一个新元素或删除一个元素时，在最坏情况下所需要的比较次数为 n 。
39. 在长度为 n 的有序线性表中进行顺序查找。最坏的情况下，需要的比较次数为 n 。
40. 在长度为 n 的有序线性表中进行二分查找。最坏的情况下，需要的比较次数为 log2n 。
41. 长度为 n 的顺序存储线性表中，当在任何位置上插入一个元素概率都相等时，插入一个元素所需移动元素的平均个数为 n/2 。
42. 排序是计算机程序设计中的一种重要操作，常见的排序方法有插入排序、 交换排序和选择排序等。
43. 快速排序法可以实现通过一次交换而消除多个 逆序 。
44. 快速排序法的关键是对线性表进行 分割 。
45. 冒泡排序算法在最好的情况下的元素交换次数为 0 。
46. 在最坏情况下，冒泡排序的时间复杂度为 n(n-1) /2 。
47. 对于长度为 n 的线性表，在最坏情况下，快速排序所需要的比较次数为 n(n-1) /2 。
48.在最坏情况下，简单插入排序需要比较的次数为 n(n-1) /2 。
49.在最坏情况下，希尔排序需要比较的次数为 O(n1.5) 。注：括号里是 n 的 1.5 次方。
50. 在最坏情况下，简单选择排序需要比较的次数为 n(n-1) /2 。
51. 在最坏情况下，堆排序需要比较的次数为 o(nlog2n) 。
52.对于输入为 N 个数进行快速排序算法的平均时间复杂度是 O(Nlog2 N)。

数据结构与算法，这个部分的内容其实是十分的庞大，要想都覆盖到不太容易。在校学习阶段我们可能需要对每种结构，每种算法都学习，但是找工作笔试或者面试的时候，要在很短的时间内考察一个人这方面的能力，把每种结构和算法都问一遍不太现实。所以，实际的情况是，企业一般考察一些看起来很基本的概念和算法，或者是一些变形，然后让你去实现。也许看起来简单，但是如果真让你在纸上或者是计算机上快速地完成一个算法，并且设计测试案例，最后跑起来，你就会发现会很难了。这就要求我们要熟悉，并牢固掌握常用的算法，特别是那些看起来貌似简单的算法，正是这些用起来很普遍的算法，才要求我们能很扎实的掌握，在实际工作中提高工作效率。遇到复杂的算法，通过分析和扎实的基本功，应该可以很快地进行开发。

闲话少说，下面进入正题。

一.数据结构部分

1.数组和链表的区别。（很简单，但是很常考，记得要回答全面）

C++语言中可以用数组处理一组数据类型相同的数据，但不允许动态定义数组的大小，即在使用数组之前必须确定数组的大小。而在实际应用中，用户使用数组之前有时无法准确确定数组的大小，只能将数组定义成足够大小，这样数组中有些空间可能不被使用，从而造成内存空间的浪费。链表是一种常见的数据组织形式，它采用动态分配内存的形式实现。需要时可以用 new 分配内存空间，不需要时用 delete 将已分配的空间释放，不会造成内存空间的浪费。

从逻辑结构来看：数组必须事先定义固定的长度（元素个数），不能适应数据动态地增减的情况，即数组的大小一旦定义就不能改变。当数据增加时，可能超出原先定义的元素个数；当数据减少时，造成内存浪费；链表动态地进行存储分配，可以适应数据动态地增减的情况，且可以方便地插入、删除数据项。（数组中插入、删除数据项时，需要移动其它数据项）。

从内存存储来看：（静态)数组从栈中分配空间（用 NEW 创建的在堆中），　对于程序员方便快速,但是自由度小；链表从堆中分配空间, 自由度大但是申请管理比较麻烦.

从访问方式来看：数组在内存中是连续存储的，因此，可以利用下标索引进行随机访问；链表是链式存储结构，在访问元素的时候只能通过线性的方式由前到后顺序访问，所以访问效率比数组要低。

2.链表的一些操作，如链表的反转，链表存在环路的判断（快慢指针），双向链表，循环链表相关操作。

3.队列（特殊的如优先级队列），栈的应用。（比如队列用在消息队列，栈用在递归调用中）

4.二叉树的基本操作

二叉树的三种遍历方式（前序，中序，后序）及其递归和非递归实现，三种遍历方式的主要应用（如后缀表达式等）。相关操作的时间复杂度。

5.字符串相关

整数，浮点数和字符串之间的转换（atoi,atof,itoa）

字符串拷贝注意异常检查，比如空指针，字符串重叠，自赋值，字符串结束符'\0'等。

二.算法部分

1.排序算法：

排序可以算是最基本的，最常用的算法，也是笔试面试中最常被考察到的算法。最基本的冒泡排序，选择排序，插入排序要可以很快的用代码实现，这些主要考察你的实际编码能力。堆排序，归并排序，快排序，这些算法需要熟悉主要的思想，和需要注意的细节地方。需要熟悉常用排序算法的时间和空间复杂度。

各种排序算法的使用范围总结：（1）当数据规模较小的时候，可以用简单的排序算法如直接插入排序或直接选择排序。（2）当文件的初态已经基本有序时，可以用直接插入排序或冒泡排序。（3）当数据规模比较大时，应用速度快的排序算法。可以考虑用快速排序。当记录随机分布的时候，快排的平均时间最短，但可能出现最坏的情况，这时候的时间复杂度是 O(n^2)，且递归深度为 n,所需的栈空间问 O(n)。（4）堆排序不会出现快排那样的最坏情况，且堆排序所需的辅助空间比快排要少。但这两种算法都不是稳定的，若要求排序时稳定的，可以考虑用归并排序。（5）归并排序可以用于内排序，也可以用于外排序。在外排序时，通常采用多路归并，并且通过解决长顺串的合并，产生长的初始串，提高主机与外设并行能力等措施，以减少访问外存额次数，提高外排序的效率。

2,查找算法

能够熟练写出或者是上机编码出二分查找的程序。

3.hash 算法

4.一些算法设计思想。

贪心算法，分治算法，动态规划算法，随机化算法，回溯算法等。这些可以根据具体的例子程序来复习。

5.STL

STL(Standard Template Library)是一个 C++领域中，用模版技术实现的数据结构和算法库，已经包含在了 C++标准库中。其中的 vecor,list,stack,queue 等结构不仅拥有更强大的功能，还有了更高的安全性。除了数据结构外，STL 还包含泛化了的迭代器，和运行在迭代器上的

各种实用算法。这些对于对性能要求不是太高，但又不希望自己从底层实现算法的应用还是很具有诱惑力的。

1.栈和队列的共同特点是（只允许在端点处插入和删除元素）
4.栈通常采用的两种存储结构是（线性存储结构和链表存储结构）
5.下列关于栈的叙述正确的是（D）
A.栈是非线性结构 B.栈是一种树状结构 C.栈具有先进先出的特征 D.栈有后进先出的特征
6.链表不具有的特点是（B）A.不必事先估计存储空间　　B.可随机访问任一元素
C.插入删除不需要移动元素　　D.所需空间与线性表长度成正比
7.用链表表示线性表的优点是（便于插入和删除操作）
8.在单链表中，增加头结点的目的是（方便运算的实现）
9.循环链表的主要优点是（从表中任一结点出发都能访问到整个链表）
10.线性表 L＝（a1,a2,a3,……ai,……an），下列说法正确的是（D）
A.每个元素都有一个直接前件和直接后件　B.线性表中至少要有一个元素
C.表中诸元素的排列顺序必须是由小到大或由大到小
D.除第一个和最后一个元素外，其余每个元素都有一个且只有一个直接前件和直接后件
11.线性表若采用链式存储结构时，要求内存中可用存储单元的地址（D）
A.必须是连续的 B.部分地址必须是连续的 C.一定是不连续的 D.连续不连续都可以
12.线性表的顺序存储结构和线性表的链式存储结构分别是（随机存取的存储结构、顺序存取的存储结构）
13.树是结点的集合，它的根结点数目是（有且只有 1）
14.在深度为 5 的满二叉树中，叶子结点的个数为（31）
15.具有 3 个结点的二叉树有（5 种形态）
16.设一棵二叉树中有 3 个叶子结点，有 8 个度为 1 的结点，则该二叉树中总的结点数为（13）
17.已知二叉树后序遍历序列是 dabec，中序遍历序列是 debac，它的前序遍历序列是（cedba）
18.已知一棵二叉树前序遍历和中序遍历分别为 ABDEGCFH 和 DBGEACHF，则该二叉树的后序遍历为（DGEBHFCA）
19.若某二叉树的前序遍历访问顺序是 abdgcefh，中序遍历访问顺序是 dgbaechf，则其后序遍历的结点访问顺序是（gdbehfca）
20.数据库保护分为：安全性控制、 完整性控制 、并发性控制和数据的恢复。
1. 在计算机中，算法是指（解题方案的准确而完整的描述）
2.在下列选项中，哪个不是一个算法一般应该具有的基本特征（无穷性）
说明：算法的四个基本特征是：可行性、确定性、有穷性和拥有足够的情报。
3. 算法一般都可以用哪几种控制结构组合而成（顺序、选择、循环）
4.算法的时间复杂度是指（算法执行过程中所需要的基本运算次数）
5. 算法的空间复杂度是指（执行过程中所需要的存储空间）
6. 算法分析的目的是（分析算法的效率以求改进）
7. 下列叙述正确的是（C）
A．算法的执行效率与数据的存储结构无关
B．算法的空间复杂度是指算法程序中指令（或语句）的条数
C．算法的有穷性是指算法必须能在执行有限个步骤之后终止
D．算法的时间复杂度是指执行算法程序所需要的时间
8.数据结构作为计算机的一门学科，主要研究数据的逻辑结构、对各种数据结构进行的运算，以及（数据的存储结构）

9. 数据结构中，与所使用的计算机无关的是数据的（C）
A．存储结构 B．物理结构 C．逻辑结构 D．物理和存储结构
10. 下列叙述中，错误的是（B）
A．数据的存储结构与数据处理的效率密切相关
B．数据的存储结构与数据处理的效率无关
C．数据的存储结构在计算机中所占的空间不一定是连续的
D．一种数据的逻辑结构可以有多种存储结构
11. 数据的存储结构是指（数据的逻辑结构在计算机中的表示）
12. 数据的逻辑结构是指（反映数据元素之间逻辑关系的数据结构）
13. 根据数据结构中各数据元素之间前后件关系的复杂程度，一般将数据结构分为（线性结构和非线性结构）
14. 下列数据结构具有记忆功能的是（C）A．队列 B．循环队列 C．栈 D．顺序表
15. 下列数据结构中，按先进后出原则组织数据的是（B）
A．线性链表 B．栈 C．循环链表 D．顺序表
16. 递归算法一般需要利用（队列）实现。
17. 下列关于栈的叙述中正确的是（D）A．在栈中只能插入数据 B．在栈中只能删除数据
C．栈是先进先出的线性表 D．栈是先进后出的线性表
20. 由两个栈共享一个存储空间的好处是（节省存储空间，降低上溢发生的机率）
21. 应用程序在执行过程中，需要通过打印机输出数据时，一般先形成一个打印作业，将其存放在硬盘中的一个指定（队列）中，当打印机空闲时，就会按先来先服务的方式从中取出待打印的作业进行打印。
22.下列关于队列的叙述中正确的是（C）A．在队列中只能插入数据 B．在队列中只能删除数据 C．队列是先进先出的线性表 D．队列是先进后出的线性表
23.下列叙述中，正确的是（D）A．线性链表中的各元素在存储空间中的位置必须是连续的
B．线性链表中的表头元素一定存储在其他元素的前面 C．线性链表中的各元素在存储空间中的位置不一定是连续的，但表头元素一定存储在其他元素的前面 D．线性链表中的各元素在存储空间中的位置不一定是连续的，且各元素的存储顺序也是任意的
24.下列叙述中正确的是（A）A．线性表是线性结构 B．栈与队列是非线性结构
C．线性链表是非线性结构 D．二叉树是线性结构
25. 线性表 L＝（a1,a2,a3,……ai,……an），下列说法正确的是（D）
A．每个元素都有一个直接前件和直接后件 B．线性表中至少要有一个元素
C．表中诸元素的排列顺序必须是由小到大或由大到小 D．除第一个元素和最后一个元素外，其余每个元素都有一个且只有一个直接前件和直接后件
26.线性表若采用链式存储结构时，要求内存中可用存储单元的地址（连续不连续都可以）
27. 链表不具有的特点是（B）A．不必事先估计存储空间 B．可随机访问任一元素
C．插入删除不需要移动元素 D．所需空间与线性表长度成正比
28. 非空的循环单链表 head 的尾结点（由 p 所指向），满足（p->next=head）
29.与单向链表相比，双向链表的优点之一是（更容易访问相邻结点）
30. 在（D）中，只要指出表中任何一个结点的位置，就可以从它出发依次访问到表中其他所有结点。A．线性单链表 B．双向链表 C．线性链表 D．循环链表
31. 以下数据结构属于非线性数据结构的是（C）A．队列 B．线性表 C．二叉树 D．栈
32.树是结点的集合，它的根结点数目是（有且只有 1）

33.具有 3 个结点的二叉树有（5 种形态）
34. 在一棵二叉树上第 8 层的结点数最多是（128） 注：2K-1
35. 在深度为 5 的满二叉树中，叶子结点的个数为（16） 注：2n-1
36. 在深度为 5 的满二叉树中，共有（31）个结点。 注：2n－1
37.设一棵完全二叉树共有 699 个结点，则在该二叉树中的叶子结点数为（350）
说明：完全二叉树总结点数为 N，若 N 为奇数，则叶子结点数为（N+1）/2；若 N 为偶数，则叶子结点数为 N/2。
38. 设有下列二叉树，对此二叉树中序遍历的结果是（B）
A．ABCDEF
B．DBEAFC
C．ABDECF
D．DEBFCA
39.已知二叉树后序遍历序列是 dabec，中序遍历序列 debac，它的前序遍历序列是（cedba）
40. 已知一棵二叉树前序遍历和中序遍历分别为 ABDEGCFH 和 DBGEACHF，则该二叉树的后序遍历为（DGEBHFCA）
41.若某二叉树的前序遍历访问顺序是 abdgcefh，中序遍历访问顺序是 dgbaechf，则其后序遍历的结点访问顺序是（gdbehfca）
42. 串的长度是（串中所含字符的个数）
43.设有两个串 p 和 q，求 q 在 p 中首次出现位置的运算称做（模式匹配）
44. N 个顶点的连通图中边的条数至少为（N-1）
45.N 个顶点的强连通图的边数至少有（N）
46.对长度为 n 的线性表进行顺序查找，在最坏情况下所需要的比较次数为（N）
47. 最简单的交换排序方法是（冒泡排序）
48.假设线性表的长度为 n，则在最坏情况下，冒泡排序需要的比较次数为（n(n-1)/2）
49. 在待排序的元素序列基本有序的前提下，效率最高的排序方法是（冒泡排序）
50. 在最坏情况下，下列顺序方法中时间复杂度最小的是（堆排序）
51. 希尔排序法属于（插入类排序）
52. 堆排序法属于（选择类排序）
53. 在下列几种排序方法中，要求内存量最大的是（归并排序）
54. 已知数据表 A 中每个元素距其最终位置不远，为节省时间，应采用（直接插入排序）
55. 算法的基本特征是可行性、确定性、 有穷性 和拥有足够的情报。
1.一个算法通常由两种基本要素组成：一是对数据对象的运算和操作，二是算法的控制结构。
1. 算法的复杂度主要包括时间复杂度和 空间 复杂度。
2. 实现算法所需的存储单元多少和算法的工作量大小分别称为算法的空间复杂度和时间复杂度 。
3.所谓数据处理是指对数据集合中的各元素以各种方式进行运算，包括插入、删除、查找、更改等运算，也包括对数据元素进行分析。
4.数据结构是指相互有关联的 数据元素 的集合。
5.数据结构分为逻辑结构与存储结构，线性链表属于 存储结构 。
6.数据结构包括数据的 逻辑 结构和数据的存储结构。
7. 数据结构包括数据的逻辑结构、数据的 存储结构 以及对数据的操作运算。
8.数据元素之间的任何关系都可以用 前趋和后继 关系来描述。

9.数据的逻辑结构有线性结构和非线性结构两大类。
10.常用的存储结构有顺序、链接、 索引 等存储结构。
11. 顺序存储方法是把逻辑上相邻的结点存储在物理位置 相邻 的存储单元中。
12. 栈的基本运算有三种：入栈、退栈与读栈顶元素 。
13. 队列主要有两种基本运算：入队运算与 退队运算 。
14. 在实际应用中，带链的栈可以用来收集计算机存储空间中所有空闲的存储结点，这种带链的栈称为 可利用栈 。
15.栈和队列通常采用的存储结构是 链式存储和顺序存储 。
16.当线性表采用顺序存储结构实现存储时，其主要特点是 逻辑结构中相邻的结点在存储结构中仍相邻 。
17. 循环队列主要有两种基本运算：入队运算与退队运算。每进行一次入队运算，队尾指针就 进 1 。
18.当循环队列非空且队尾指针等于对头指针时，说明循环队列已满，不能进行入队运算。这种情况称为 上溢 。
19.当循环队列为空时，不能进行退队运算，这种情况称为 下溢 。
20. 在一个容量为 25 的循环队列中，若头指针 front=16，尾指针 rear=9，则该循环队列中共有 18 个元素。注：当 rear<front 时，元素个数＝总容量－（front－rear）；
当 rear>front 时，元素个数＝rear－front。
1.判断链表是否存在环型链表问题：判断一个链表是否存在环，例如下面这个链表就存在一个环：
例如 N1->N2->N3->N4->N5->N2 就是一个有环的链表，环的开始结点是 N5 这里有一个比较简单的解法。设置两个指针 p1，p2。每次循环 p1 向前走一步，p2 向前走两步。直到 p2 碰到 NULL 指针或者两个指针相等结束循环。如果两个指针相等则说明存在环。

```
struct link
{
        int data;
         link* next;
};

bool IsLoop(link* head)
{
         link* p1=head, *p2 = head;
          if (head ==NULL || head->next ==NULL)
          {
                    return false;
          }
          do{
             p1= p1->next;
             p2 = p2->next->next;
          } while(p2 && p2->next && p1!=p2);
          if(p1 == p2)
                    return true;
          else
                    return false;
```

}

2,链表反转　单向链表的反转是一个经常被问到的一个面试题，也是一个非常基础的问题。比如一个链表是这样的：　1->2->3->4->5　通过反转后成为 5->4->3->2->1。最容易想到的方法遍历一遍链表，利用一个辅助指针，存储遍历过程中当前指针指向的下一个元素，然后将当前节点元素的指针反转后，利用已经存储的指针往后面继续遍历。源代码如下：

```
struct linka {
        int data;
        linka* next;
};

void reverse(linka*& head)
{
        if(head ==NULL)
            return;
        linka*pre, *cur, *ne;
        pre=head;
        cur=head->next;
        while(cur)
        {
            ne = cur->next;
            cur->next = pre;
            pre = cur;
            cur = ne;
        }
        head->next = NULL;
        head = pre;
}
```

还有一种利用递归的方法。这种方法的基本思想是在反转当前节点之前先调用递归函数反转后续节点。源代码如下。不过这个方法有一个缺点，就是在反转后的最后一个结点会形成一个环，所以必须将函数的返回的节点的 next 域置为 NULL。因为要改变 head 指针，所以我用了引用。算法的源代码如下：

```
linka* reverse(linka* p,linka*& head)
{
        if(p == NULL || p->next == NULL)
        {
            head=p;
            return p;
        }
        else
        {
            linka* tmp = reverse(p->next,head);
            tmp->next = p;
            return p;
        }
```

}

3,判断两个数组中是否存在相同的数字　给定两个排好序的数组，怎样高效得判断这两个数组中存在相同的数字？

这个问题首先想到的是一个 O(nlogn)的算法。就是任意挑选一个数组，遍历这个数组的所有元素，遍历过程中，在另一个数组中对第一个数组中的每个元素进行 binary　search。用 C++实现代码如下：

```
bool findcommon(int a[],int size1,int b[],int size2)
{
        int i;
        for(i=0;i<size1;i++)
        {
            int start=0,end=size2-1,mid;
            while(start<=end)
            {
                mid=(start+end)/2;
                if(a[i]==b[mid])
                    return true;
                else if (a[i]<b[mid])
                    end=mid-1;
                else
                    start=mid+1;
            }
        }
        return false;
}
```

后来发现有一个　O(n)算法。因为两个数组都是排好序的。所以只要一次遍历就行了。首先设两个下标，分别初始化为两个数组的起始地址，依次向前推进。推进的规则是比较两个数组中的数字，小的那个数组的下标向前推进一步，直到任何一个数组的下标到达数组末尾时，如果这时还没碰到相同的数字，说明数组中没有相同的数字。

```
bool findcommon2(int a[], int size1, int b[], int size2)
{
        int i=0,j=0;
        while(i<size1 && j<size2)
        {
            if(a[i]==b[j])
                    return true;
            if(a[i]>b[j])
                j++;
            if(a[i]<b[j])
                i++;
        }
        return false;
}
```

4,最大子序列　问题：

给定一整数序列 A1， A2，... An （可能有负数），求 A1~An 的一个子序列 Ai~Aj，使得 Ai 到 Aj 的和最大

例如：

整数序列-2, 11, -4, 13, -5, 2, -5, -3, 12, -9 的最大子序列的和为 21。

对于这个问题，最简单也是最容易想到的那就是穷举所有子序列的方法。利用三重循环，依次求出所有子序列的和然后取最大的那个。当然算法复杂度会达到 O(n^3)。显然这种方法不是最优的，下面给出一个算法复杂度为 O(n)的线性算法实现，算法的来源于 Programming Pearls 一书。 在给出线性算法之前，先来看一个对穷举算法进行优化的算法，它的算法复杂度为 O(n^2)。其实这个算法只是对对穷举算法稍微做了一些修改：其实子序列的和我们并不需要每次都重新计算一遍。假设 Sum(i, j)是 A[i] ... A[j]的和，那么 Sum(i, j+1) = Sum(i, j) + A[j+1]。利用这一个递推，我们就可以得到下面这个算法：

```
int max_sub(int a[],int size)
{
        int i,j,v,max=a[0];
        for(i=0;i<size;i++)
        {
            v=0;
            for(j=i;j<size;j++)
            {
                v=v+a[j];//Sum(i, j+1) = Sum(i, j) + A[j+1]
                if(v>max)
                    max=v;
            }
        }
        return max;
}
```

那怎样才能达到线性复杂度呢？这里运用动态规划的思想。先看一下源代码实现：

```
int max_sub2(int a[], int size)
{
        int i,max=0,temp_sum=0;
        for(i=0;i<size;i++)
        {
            temp_sum+=a[i];
            if(temp_sum>max)
                max=temp_sum;
            else if(temp_sum<0)
                temp_sum=0;
        }
        return max;
}
```

6,按单词反转字符串 并不是简单的字符串反转，而是按给定字符串里的单词将字符串倒转过来，就是说字符串里面的单词还是保持原来的顺序，这里的每个单词用空格分开。例如：

Here is www.fishksy.com.cn

经过反转后变为：

www.fishksy.com.cn is Here

如果只是简单的将所有字符串翻转的话，可以遍历字符串，将第一个字符和最后一个交换，第二个和倒数第二个交换，依次循环。其实按照单词反转的话可以在第一遍遍历的基础上，再遍历一遍字符串，对每一个单词再反转一次。这样每个单词又恢复了原来的顺序。

```
char* reverse_word(const char* str)
{
        int len = strlen(str);
        char* restr = new char[len+1];
        strcpy(restr,str);
        int i,j;
        for(i=0,j=len-1;i<j;i++,j--)
        {
            char temp=restr[i];
            restr[i]=restr[j];
            restr[j]=temp;
        }
        int k=0;
        while(k<len)
        {
            i=j=k;
            while(restr[j]!=' ' && restr[j]!='' )
                j++;
            k=j+1;
            j--;
            for(;i<j;i++,j--)
            {
                char temp=restr[i];
                restr[i]=restr[j];
                restr[j]=temp;
            }
        }
        return restr;
}
```

如果考虑空间和时间的优化的话，当然可以将上面代码里两个字符串交换部分改为异或实现。

例如将

```
            char temp=restr[i];
```

```
        restr[i]=restr[j];
        restr[j]=temp;
```
改为
```
        restr[i]^=restr[j];
            restr[j]^=restr[i];
        restr[i]^=restr[j];
```

7,字符串反转　我没有记错的话是一道 MSN 的笔试题，网上无意中看到的，拿来做了一下。题目是这样的，给定一个字符串，一个这个字符串的子串，将第一个字符串反转，但保留子串的顺序不变。例如：

输入：第一个字符串: "This is fishsky 's Chinese site: http://www.fishsky.com.cn/cn"

子串: "fishsky"

输出：　"nc/nc.moc.fishsky.www//:ptth :etis esenihC s'fishsky si sihT"

一般的方法是先扫描一边第一个字符串，然后用 stack 把它反转，同时记录下子串出现的位置。然后再扫描一遍把记录下来的子串再用 stack 反转。我用的方法是用一遍扫描数组的方法。扫描中如果发现子串，就将子串倒过来压入堆栈。

最后再将堆栈里的字符弹出，这样子串又恢复了原来的顺序。源代码如下：
```
#include <iostream>
#include <cassert>
#include <stack>
using namespace std;
//reverse the string 's1' except the substring 'token'.
const char* reverse(const char* s1, const char* token)
{
        assert(s1 && token);
        stack<char> stack1;
        const char* ptoken = token, *head = s1, *rear = s1;
        while (*head != '')
        {
                while(*head!= '' && *ptoken == *head)
                {
                  ptoken++;
                  head++;
                }
                if(*ptoken == '')//contain the token
                {
                  const char* p;
                  for(p=head-1;p>=rear;p--)
                          stack1.push(*p);
```

```
                ptoken = token;
                rear = head;
            }
            else
            {
                stack1.push(*rear);
                    head=++rear;
                ptoken = token;
            }
        }
        char * return_v = new char[strlen(s1)+1];
        int i=0;
        while(!stack1.empty())
        {
            return_v[i++] = stack1.top();
            stack1.pop();
        }
        return_v[i]='';
        return return_v;
}
int main(int argc, char* argv[])
{cout<<"This is fishsky 's Chinese site: http://www.fishsky.com.cn/cn
";
        cout<<reverse("This is fishsky's Chinese site: http://www. fishsky.com.cn/cn","
fishsky ");
        return 0;
}
```

8, 删除数组中重复的数字　问题：一个动态长度可变的数字序列，以数字 0 为结束标志，要求将重复的数字用一个数字代替，例如：

将数组 1,1,1,2,2,2,2,2,7,7,1,5,5,5,0 转变成 1,2,7,1,5,0 问题比较简单，要注意的是这个数组是动态的。所以避免麻烦我还是用了 STL 的 vector。

```
#include <iostream>
#include <vector>
using namespace std;
//remove the duplicated numbers in an intger array, the array was end with 0;
//e.g. 1,1,1,2,2,5,4,4,4,4,1,0 --->1,2,5,4,1,0
void static remove_duplicated(int a[], vector<int>& _st)
{
        _st.push_back(a[0]);
        for(int i=1;_st[_st.size()-1]!=0;i++)
        {
            if(a[i-1]!=a[i])
```

```
                    _st.push_back(a[i]);
        }
}
```

当然如果可以改变原来的数组的话，可以不用 STL，仅需要指针操作就可以了。下面这个程序将修改原来数组的内容。

```
void static remove_duplicated2(int a[])
{
        if(a[0]==0 || a==NULL)
                return;
        int insert=1,current=1;
        while(a[current]!=0)
        {
                if(a[current]!=a[current-1])
                {
                  a[insert]=a[current];
                  insert++;
                  current++;
                }
                else
                  current++;
        }
        a[insert]=0;
}
```

9,如何判断一棵二叉树是否是平衡二叉树　　问题：判断一个二叉排序树是否是平衡二叉树

解决方案：

根据平衡二叉树的定义，如果任意节点的左右子树的深度相差不超过 1，那这棵树就是平衡二叉树。

首先编写一个计算二叉树深度的函数，利用递归实现。

```
template<typename T>
static int Depth(BSTreeNode<T>* pbs)
{
        if (pbs==NULL)
                return 0;
        else
        {
                int ld = Depth(pbs->left);
                int rd = Depth(pbs->right);
                return 1 + (ld >rd ? ld : rd);
        }
}
```

下面是利用递归判断左右子树的深度是否相差 1 来判断是否是平衡二叉树的函数：

```
template<typename T>
static bool isBalance(BSTreeNode<T>* pbs)
```

```
{
        if (pbs==NULL)
                return true;
        int dis = Depth(pbs->left) - Depth(pbs->right);
        if (dis>1 || dis<-1 )
                return false;
        else
                return isBalance(pbs->left) && isBalance(pbs->right);
```

```
4.abstract class Something {
  private abstract String doSomething ();
}
```

该段代码有错吗？

答案: 错。abstract 的 methods 不能以 private 修饰。abstract 的 methods 就是让子类 implement(实现)具体细节的，怎么可以用 private 把 abstract method 封锁起来呢? (同理，abstract method 前不能加 final)。

5.看看下面的代码段错在哪里？

```
public class Something {
  void doSomething () {
    private String s = "";
    int l = s.length();
  }
}
```

答案: 错。局部变量前不能放置任何访问修饰符 (private，public，和 protected)。final 可以用来修饰局部变量
(final 如同 abstract 和 strictfp，都是非访问修饰符，strictfp 只能修饰 class 和 method 而非 variable)。

6. 下面该段代码是否有错，若是有错错在哪里？

```
abstract class Name {
  private String name;
  public abstract boolean isStupidName(String name) {}
}
```

答案: 错。abstract method 必须以分号结尾，且不带花括号。

第一部分选择题

一、单项选择题（本大题共 14 小题，每小题 1 分，共 14 分）在每小题列出的四个选项中只有一个选项是符合题目要求的，请将正确选项前的字母填在题后的括号内。

1.算法分析的目的是(　　C　)
A.找出数据结构的合理性　　B.研究算法中的输入/输出关系
C.分析算法的效率以求改进　　D.分析算法的易读性

2.在需要经常查找结点的前驱与后继的场合中，使用(　B　)比较合适。
A.单链表　　B.双链表
C.顺序表　　D.循环链表

3.下面关于线性表的叙述中，错误的为(　D　　)
A.顺序表使用一维数组实现的线性表
B.顺序表必须占用一片连续的存储单元
C.顺序表的空间利用率高于链表
D.在链表中，每个结点只有一个链域

4.带头结点的单链表 head 为空的判断条件是(　　B　　)
A. head=NIL　　B. head->next=NIL
C. head->next=head　　D. head<>NIL

5.队列通常采用两种存储结构是(　　A　　)
A.顺序存储结构和链表存储结构　　B.散列方式和索引方式
C.链表存储结构和数组　　D.线性存储结构和非线性存储结构

6.按照二叉树的定义，具有 3 个结点的二叉树有(　C　)种。

A.3　　B.4　　C.5　　D.6

7.二叉树的结构如下图所示，其中序遍历的序列为(　　)

A.a,b,d,g,c,e,f,h
B.d,g,b,a,e,c,h,f
C.g,d,b,e,h,f,c,a
D.a,b,c,d,e,f,g,h

8.深度为 5 的二叉树至多有(　C　)个结点。 (2^M-1)
A.16　　B.32　　C.31　　D.10

9.对于一个具有 n 个顶点的无向图，若采用邻接表表示，则存放表头结点的数组的大小为(　A　)
A.n　　B.n+1　　C.n-1　　D.n+边数

10.在一个具有 n 个顶点的无向图中，要连通全部顶点至少需要(　C　)条边。
A.n　　B.n+1　　C.n-1　　D.n/2

11.静态查找表与动态查找表二者的根本差别在于(　B　)
A.它们的逻辑结构不一样
B.施加在其上的操作不同
C.所包含的数据元素的类型不一样
D.存储实现不一样

12.散列文件使用散列函数将记录的关键字值计算转化为记录的存放地址。因为散列函数不是一对一的关系，所以选择好的(D　)方法是散列文件的关键。
A.散列函数　　B.除余法中的质数
C.冲突处理　　D.散列函数和冲突处理

13.对于大文件的排序要研究在外设上的排序技术，即(　　C　)
A.快速排序法　　B.内排序法
C.外排序法　　D.交叉排序法

14.设有 5000 个无序的元素，希望用最快的速度挑选出其中前 50 个最大的元素，最好选用(　C　)法。
A.冒泡排序　　B.快速排序
C.堆排序　　D.基数排序

二、判断题(判断下列各题，正确的在题干后面括号内打“√”，错误的打“×”。每小题 2 分，共 20 分)

1.所谓数据的逻辑结构指的是数据元素之间的逻辑关系。(　　)

2.在线性结构中，每个结点都有一个直接前驱和一个直接后继。(　　)

3.插入和删除是数组的两种基本操作。(　　)

4.在链栈的头部必须要设置头结点。(　　)

5.在二叉树中插入结点则该二叉树便不再是二叉树。(　　)

6.查找表的逻辑结构是集合。(　　)

7.静态查找表的检索与修改被分成两个不交叉的阶段分别进行。(　　)

8.在索引顺序文件中插入新的记录时，必须复制整个文件。(　　)

9.如果某种排序算法是不稳定的，则该方法没有实际的应用价值。(　　)

10.对于 n 个记录的集合进行冒泡排序，在最坏情况下所需要的时间是 0(n2)(　　)

三、填空题(每小题 2 分，共 30 分)

1.程序设计的实质是_____和_____。

2.设由字符串 a=′ data′ 、b=′ structure′ 、c=′ -′ ,则 a 与 c 连接然后与 b 连接的结果为：_____。

3.通常单链表的头结点指的是_____；单链表的首结点指的是_____。

4.一个队列的入队序列是 a、b、c、d，则队列的输出序列为_____。

5.栈结构通常采用的两种存储结构是_____和_____。

6.具有 N 个结点的完全二叉树的深度为_____。

7.树的三种主要的遍历方法是：_____、_____和层次遍历。

8.在无向图的邻接矩阵 A 中，若 A〔i,j〕等于 1，则 A〔j,i〕等于_____。

9.采用散列技术实现散列表时，需要考虑的两个主要问题是：构造_____和解决_____。

10.索引顺序表上的查找分两个阶段：(1)_____；(2)_____。

11.散列文件中的记录通常是成组存放的。若干的记录组成一个存储单位，称作_____。

12.就文件而言，按用户的观点所确定的基本存储单元称为_____。按外设的观点所确定的基本存储单元称为_____。

13.文件的检索有三种方式：_____存取、_____存取和按关键字存取。

14.最简单的交换排序方法是_____排序。

15.外排序的基本方法是_____。

四、应用题(每小题 6 分，共 18 分)

1.假定在学生的档案中含有：姓名、学号、年龄、性别。如采用线性表作为数据结构来实现档案管理问题，分别给出线性表的在顺序实现下的类型定义和在链接实现下的类型定义。

2.有一份电文中共使用五个字符：a、b、c、d、e,它们的出现频率依次为
8、14、10、4、18，请构造相应的哈夫曼树(左子树根结点的权小于等于右子树根结点的权)，求出每个字符的哈夫曼编码。

3.有初始的无序序列为{98,65,38,40,12,51,100,77,26,88},给出对其进行归并排序(升序)的每一趟的结果。

五、设计题(每小题 6 分，共 18 分)

1.假设用一个循环单链表来表示队列(称为循环链队)，该队列中只设一个队尾指针 rear,不设队首指针。请编写向循环链队中插入一个元素 X 的过程。

2.以邻接表为存储结构，写出连通图的深度优先搜索算法。

3.设有一组关键字{19,01,23,14,55,20,84,27,68,11,10,77},采用散列函数：H(key)=key MOD 13,采用线性探测法解决冲突，试在 0～18 的散列地址空间中对该关键字序列构造散列表。

数据结构导论试题参考答案

一、单项选择题(每小题 1 分，共 14 分)　1.C　2.B　3.D　4.B　5.A

6.C　7.B　8.C　9.A　10.C

11.B　12.D　13.C　14.C

二、判断题(每小题 2 分，共 20 分)

1. ×　2. ×　3. ×□　4. ×　5. ×□

6. √　7. √　8. ×　9. ×　10. √□三、填空题(每小题 2 分，共 30 分)　1.(1)数据表示　(2)数据处理。　2.′data-structure′。　3.(1)在单链表第一个结点之前增设的一个类型相同的结点

(2)表结点中的第一个结点。　4. a、b、c、d。　5.(1)顺序存储结构　(2)链表存储结构。

6.〔log2N〕+1。

7.(1)先根遍历　(2)后根遍历。

8.1。

9.(1)散列函数　(2)冲突。

10.(1)确定待查元素所在的块　(2)在块内查找待查的元素。

11.桶。

12.(1)逻辑结构　(2)物理结构。

13.(1)顺序　(2)直接。

14.冒泡排序。　15.归并。四、应用题(每小题 6 分，共 18 分)　1.顺序实现：

```
constmaxsize：=100; {顺序表的容量}　type datatype=record {档案数据类型}
        name：string〔10〕;{姓名}
        number：integer;{学号}
        sex：boolean;{性别}
        age：integer;{年龄}
end;
typeslist =record
        data：array 〔1..maxsize〕 of datatype;
        last：integer;
end;
```

链接实现：

```
        type pointer= ↑ node;
node=record
              name：string 〔10〕;｛姓名｝
              number：interger；｛学号｝
              sex：boolean;｛性别｝
              age：integer;｛年龄｝
              next：pointer;｛结点的后继指针｝
end;
list=pointer;
```

2.哈夫曼树为：

相应的哈夫曼编码为：

a:001　　b:10　　c:01　　d:000　　e:11

画出正确的哈夫曼树给 4 分，写出相应哈夫曼编码给 2 分

3.

初始无序序列：　98　　65　　38　　40　　12　　51　　100　　77　　26　　88

｛98｝｛65｝｛38｝｛40｝｛12｝｛51｝｛100｝｛77｝｛26｝｛88｝

第一次归并：｛65　　98｝｛38　　40｝｛12　　51｝｛77　　100｝｛26　　88｝

第二次归并：｛38　　40　　65　　98｝｛12　　51　　77　　100｝｛26　　88｝

第三次归并：｛12　　38　　40　　51　　65　　77　　98　　100｝｛26　　88｝

第四次归并：｛12　　26　　38　　40　　51　　65　　77　　88　　98　　100｝

五、设计题(每小题 6 分，共 18 分)

1.

```
PROCEDURE insert (VAR rear：pointer; x：integer);
  VAR head, tmp：pointer;
  BEGIN
new(tmp);
tmp ↑ .data：=x;
    if (rear=NIL) then 　｛循环队列为空，新结点是队列的首结点｝
      BEGIN
      rear：=tmp;
      rear ↑ .next：=tmp;
    END
  else　　｛队列不空，将新结点插入在队列尾｝
    BEGIN
      head：=rear ↑ .next;
```

```
            rear↑.next：=tmp;
            rear：=tmp;
            rear↑.next：=head;
          END
       END;
```

2.procedure dfs(g:adj-list;v1：integer);

{从 v1 出发，深度优先遍历图 g}

```
begin
write(v1);
        visited(v1)：=true;        {标志 v1 已访问}
        p=g〔v1〕.link;        {找 v1 的第一个邻接点}
while p<>nil do
〔 if not (visited〔p↑.vertex〕)
                then dfs(g,p↑.vertex);
                p：=p↑.next〕{找 v1 的下一个邻接点}
end;
```

以邻接表为存储结构，连通图的深度优先搜索就是顺序查找链表。

3.构造过程如下：

```
H(19)=19 MOD 13=6
H(01)=01 MOD 13=1
H(23)=23 MOD 13=10
      H(14)=14 MOD 13=1(冲突)
H(14)=(1+1) MOD 19=2
H(55)=55 MOD 13=3
H(20)=20 MOD 13=7
      H(84)=84 MOD 13=6 (冲突)
      H(84)=(6+1) MOD 19=7 (仍冲突)
H(84)=(6+2) MOD 19=8
      H(27)=27 MOD 13=1 (冲突)
      H(27)=(1+1) MOD 19=2 (冲突)
      H(27)=(1+2) MOD 19=3 (仍冲突)
H(27)=(1+3) MOD 19=4
      H(68)=68 MOD 13=3 (冲突)
      H(68)=(3+1) MOD 19=4 (仍冲突)
H(68)=(3+2) MOD 19=5
H(11)=11 MOD 13=11
      H(10)=10 MOD 13=10 (冲突)
      H(10)=(10+1) MOD 19=11 (仍冲突)
H(10)=(10+2) MOD 19=12
      H(77)=77 MOD 13=12 (冲突)
H(77)=(12+1) MOD 19=13
```

因此，各关键字相应的地址分配如下：

```
address(01)=1
address(14)=2
```

address(55)=3
address(27)=4
address(68)=5
address(19)=6
address(20)=7
address(84)=8
address(23)=10
address(11)=11
address(10)=12
address(77)=13
其余的地址中为空。

第二部分　数据结构(共 100 分)

一、单项选择题　(本大题共 12 小题，每小题 2 分，共 24 分)

1、双向链表的一个结点有(　B　)个指针。

A、l　　**B**、2

C、0　　D、3

2、在 n 个结点的顺序表中，算法的时间复杂度都是 O(1)的操作是(　A　)。

A、访问第 i 个结点(l≤i≤n)和求第 i 个结点的直接前趋(l≤i≤n)

B、在第 i 个结点后插入一个新结点(l≤i≤n)

C、删除第 i 个结点(l≤i≤n)

D、将 n 个结点从小到大排序

3、在队列中存取数据的原则是(　A　)。

A、先进先出　　B、后进后出

C、先进后出　　D、不进不出

4、在栈中，出栈操作的时间复杂度为(　A　)。

A、O(1)　　B、$O(\log_2 n)$

C、O(n)　　D、$O(n^2)$

5、设长度为 n 的链队列用单循环链表表示，若只设头指针，则入队操作的时间复杂度为(　C　)。

A、O(1)　　B、$O(\log_2 n)$

C、O(n)　　D、$O(n^2)$

6、如果二叉树的叶结点数为 n0，则具有双分支的结点数也等于(　D　)。

A、n0+1　　B、n0

C、2*n0　　**D**、n0-1

7、一棵二叉树满足下列条件:对任一结点，若存在左、右子树，则其值都小于它的左子树上所有结点的值，而大于右子树上所有结点的值。现采用(　B　)遍历方式就可以得到这棵二叉树所有结点的递增序列。

A、先根　　**B**、中根

C、后根　　D、层次

8、用邻接表表示图进行深度优先遍历时，其非递归算法通常采用(　A　)来实现算法。

A、栈　　B、队列

C、树　　D、图

9、广度优先遍历类似于二叉树的(　D　)。

A、先序遍历　　B、中序遍历

C、后序遍历　　**D**、层次遍历

10、在一个有向图中，所有顶点的人度之和等于所有顶点的出度之和的(　B　)。

A、1/2 倍　　**B**、l 倍

C、2 倍　　D、4 倍

11、任何一个带权无向连通图的最小生成树(　B　)。

A、只有一棵　　**B**、一棵或多棵

C、一定有多棵　　D、可能不存在

12、设非空单链表的数据域都为 data，指针域都为 next，指针 p 指向单链表的

第 i 个结点，s 指向生成的新结点，现将 s 结点插入到单链表中，使其成为第 i 结点，下列算法段能正确完成上述要求的是(　C　)。

A、s->next=p->next; p->next=s;

B、p->next=s; s->next=p->next;

C、s->next=p->next; p->next=s; 交换 p->data 和 s->data

D、p=s; s->next=p;

二、填空题(本大题共 8 小题，共 10 个空格，每空 2 分，共 20 分)

1、数据的逻辑结构反映_数据元素之间的逻辑关系

2、对于队列，只能在___队尾___插入元素，在____队首__删除元素。

3、算法是一运算序列，它应有:有限性、确定性、可行性、可以无任何输入，但必须有输出。

4、由一棵二叉树的前序序列和中序序列可唯一确定这棵二叉树的结构。

5、如果图的存储结构用邻接矩阵表示，从某指定顶点作为初始点进行广度优先搜索，得到的广度优先搜索序列唯一。

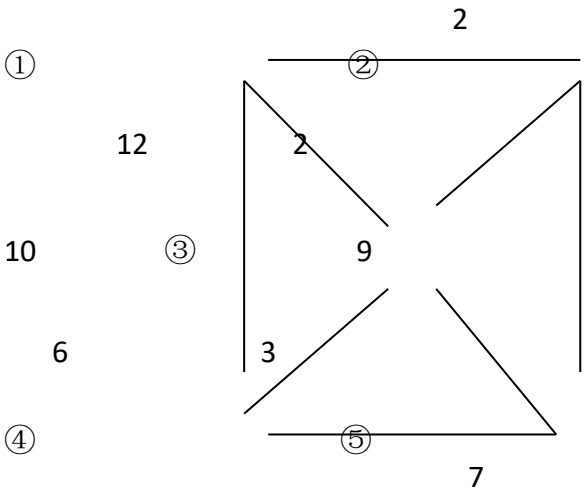
6、用 Dijkstra 算法求某一顶点到其余各顶点间的最短路径是按路径长度___递增的次序来得到最短路径的。

7、线性表(a_1，a_2，a_3……a_n。)(n>=1)中，每个元素占 c 个存储单元，m 为 a_1 首地址，则按顺序存储方式存储线性表，a_i 的存储地址是 m+c*(i-1)。

8、n 个结点的无向图，最多有 n*(n-1)/2 条边。

三、应用题(本大题共 4 小题，每小题 8 分，共 32 分)

I、用 Prim 算法求下图连通的带权图的最小代价生成树，在算法执行的某一刻，已选取的顶点集合 U=[1，2，3]，边的集合 TE=[(1，2)，(2，3)]，要选取下一条权值最小的边，应当从哪些边中选择?



[(2,5)，(3,5)，(3,4),(1,4)]中选（3，5）

2、若用插人排序方法对线性表(25，84，21，47，15，27，68，35，20)进行排序时，请给出前四趟排序结点序列的变化情况。

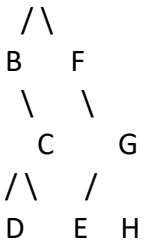
<1>(25，84)，21，47，15，27，68，35，20

<2>(21，25，84)，47，15，27，68，35，20

<3>(21，25，47，84)，15，27，68，35，20
<4>(15, 21，25，47，84)，27，68，35，20

3、已知一棵二叉树的中序序列和后序序列分别为 BDCEAFHG 和 DECBHGFA，请画出该二叉树。

A



4、设将整数 a，b，c，d 依次进栈，但只要出栈时栈非空，则可将出栈操作按任意次序夹入其中，请回答:若入、出栈次序为 Push(a)，Pop()，Push(b)，Push(c)，Pop()，Push(d)，Pop()，则出栈的字符序列是什么? acd

四、算法设计　(本大题共 3 小题，每小题 8 分，共 24 分)

1　二叉树以二叉链表为存储结构，类型声明如下，请写出一个求二叉树中结点个数的算法。

```
Typedef struct node
{datatype data;
struct node * Lchild;
struct node * Rchild;}BinaTree;
    int count ( BinaTree root)
{ if (root==0) return 0;
      else return   1+count(root->Lchild)+count ( root->Rchild);
}

void count (BinaTree   root ,int *n)
 { if ( root)
       { *n+=1;
count (root->Lchild,n);
count (root->Rchild);}
 }
```

2 设线性表用顺序结构实现，声明如下：

```
typedef struct sqlist
{char data[maxsize];
int n;}Sqlist;
```

请写一个算法，判断其是否回文？（顺读与倒读一样 如：“ababbaba”为回文）

```
  int dete (Sqlist s,int beg,int end)
    { if (beg==end) return  1;
```

```
else { while ( end-beg>=2 && s.data[beg]==s.data[end]) { beg++;end--;}
         if (s.data[beg]==s.data[end]) return 1; else return 0;
      }
    }
```

或 int dete1(Sqlist s,int beg,int end)

```
{if (beg==end) return 1;
   else    if(s[beg]!=s[end]) return 0;
        else if (end-beg>=2)
         return dete1 (s,beg+1,end-1);
         else    if(s[beg]==s[end]) return 1; else return 0;
 }
```

3　阅读下列程序，判断它是用什么方法实现排序（升序）的？并完善下列程序。

```
#include <stdio.h>
viod bubble(int[ ],int);
main( )
{        int array[ ] = {55,2,6,4,32,12,9,73,26,37};
int size = sizeof (array )/sizeof ( int );
bubble ( _array,   size_ );
}
void bubble ( int a[ ], int size )
{
int i ,temp;
int end _=0_ ;
int pass = 1;
while (! End && pass < size )
{    end = 1;
for (i = 0, i < size –pass; i ++ )
if ( _a[i]>a[i+1] )
{    temp = a [ i ]; a [ i ] = a [ i + 1]; a [ i +1] =temp; end =0;    }
_pass++;
}
for ( i =0; i <size ; i ++)
printf ( " % d ",a [ i ] );
}
```

一、单链表

目录

首先写一个单链表的 C#实现，这是我们的基石：

```
public class Link
{
    public Link Next;
    public string Data;
    public Link(Link next, string data)
    {
        this.Next = next;
        this.Data = data;
    }
}
```

其中，我们需要人为地在单链表前面加一个空节点，称其为 head。例如，一个单链表是 1->2->5，如图所示：

对一个单链表的遍历如下所示：

```
static void Main(string[] args)
{
    Link head = GenerateLink();
Link curr = head;
while (curr != null)
    {
        Console.WriteLine(curr.Data);
```

```
            curr = curr.Next;
        }
    }
```

1.单链表反转

这道题目有两种算法，既然是要反转，那么肯定是要破坏原有的数据结构的：

算法 1：我们需要额外的两个变量来存储当前节点 curr 的下一个节点 next、再下一个节点 nextnext：

```
public static Link ReverseLink1(Link head)
{
    Link curr = head.Next;
    Link next = null;
    Link nextnext = null;
    //if no elements or only one element exists
    if (curr == null || curr.Next == null)
    {
        return head;
    }
    //if more than one element
    while (curr.Next != null)
    {
        next = curr.Next;           //1
        nextnext = next.Next;       //2
        next.Next = head.Next;      //3
        head.Next = next;           //4
        curr.Next = nextnext;       //5
    }
    return head;
}
```

算法的核心是 while 循环中的 5 句话

我们发现，curr 始终指向第 1 个元素。

此外，出于编程的严谨性，还要考虑 2 种极特殊的情况：没有元素的单链表，以及只有一个元素的单链表，都是不需要反转的。

算法 2：自然是递归

如果题目简化为逆序输出这个单链表，那么递归是很简单的，在递归函数之后输出当前元素，这样能确保输出第 N 个元素语句永远在第 N+1 个递归函数之后执行，也就是说第 N 个元素永远在第 N+1 个元素之后输出，最终我们先输出最后一个元素，然后是倒数第 2 个、

倒数第 3 个，直到输出第 1 个：

```
public static void ReverseLink2(Link head)
{
    if (head.Next != null)
    {
        ReverseLink2(head.Next);
        Console.WriteLine(head.Next.Data);
    }
}
```

但是，现实应用中往往不是要求我们逆序输出（不损坏原有的单链表），而是把这个单链表逆序（破坏型）。这就要求我们在递归的时候，还要处理递归后的逻辑。
首先，要把判断单链表有 0 或 1 个元素这部分逻辑独立出来，而不需要在递归中每次都比较一次：

```
//利用递归来实现了将链表的逆序输出，使得链表在每个函数里面都有一个 head.next 调用
public static Link ReverseLink3(Link head)
{
    //if no elements or only one element exists
    if (head.Next == null || head.Next.Next == null)
        return head;
    head.Next = ReverseLink(head.Next);
    return head;
}
```

我们观测到：

```
head.Next = ReverseLink(head.Next);
```

这句话的意思是为 ReverseLink 方法生成的逆序链表添加一个空表头。
接下来就是递归的核心算法 ReverseLink 了：

```
static Link ReverseLink(Link head)
{
    if (head.Next == null)
        return head;
    Link rHead = ReverseLink(head.Next);
    head.Next.Next = head;
    head.Next = null;
    return rHead;
}
```

算法的关键就在于递归后的两条语句：

```
head.Next.Next = head;    //1
head.Next = null;         //2
```

啥意思呢？画个图表示就是：

这样，就得到了一个逆序的单链表，我们只用到了 1 个额外的变量 rHead。

2.找出单链表的倒数第 4 个元素
这道题目有两种算法，但无论哪种算法，都要考虑单链表少于 4 个元素的情况：
第 1 种算法，建立两个指针，第一个先走 4 步，然后第 2 个指针也开始走，两个指针步伐（前进速度）一致。

```
static Link GetLast4thOne(Link head)
{
    Link first = head;
    Link second = head;
    for (int i = 0; i < 4; i++)
    {
        if (first.Next == null)
            throw new Exception("Less than 4 elements");
        first = first.Next;
    }
    while (first != null)
    {
        first = first.Next;
        second = second.Next;
    }
    return second;
}
```

第 2 种算法，做一个数组 arr[4]，让我们遍历单链表，把第 0 个、第 4 个、第 8 个……第 4N 个扔到 arr[0]，把第 1 个、第 5 个、第 9 个……第 4N+1 个扔到 arr[1]，把第 2 个、第 6 个、第 10 个……第 4N+2 个扔到 arr[2]，把第 3 个、第 7 个、第 11 个……第 4N+3 个扔到 arr[3]，这样随着单链表的遍历结束，arr 中存储的就是单链表的最后 4 个元素，找到最后一个元素对应的 arr[i]，让 k=(i+1)%4，则 arr[k]就是倒数第 4 个元素。

```
static Link GetLast4thOneByArray(Link head)
{
    Link curr = head;
    int i = 0;
    Link[] arr = new Link[4];
    while (curr.Next != null)
    {
        arr[i] = curr.Next;
        curr = curr.Next;
        i = (i + 1) % 4;
```

```
        }
        if (arr[i] == null)
            throw new Exception("Less than 4 elements");
        return arr[i];
}
```

本题目源代码下载：
推而广之，对倒数第 K 个元素，都能用以上 2 种算法找出来。

3.找出单链表的中间元素
算法思想：类似于上题，还是使用两个指针 first 和 second，只是 first 每次走一步，second 每次走两步：

```
static Link GetMiddleOne(Link head)
{
        Link first = head;
        Link second = head;
        while (first != null && first.Next != null)
        {
            first = first.Next.Next;
            second = second.Next;
        }
        return second;
}
```

但是，这道题目有个地方需要注意，就是对于链表元素个数为奇数，以上算法成立。如果链表元素个数为偶数，那么在返回 second 的同时，还要返回 second.Next 也就是下一个元素，它俩都算是单链表的中间元素。
下面是加强版的算法，无论奇数偶数，一概通杀：

```
static void Main(string[] args)
{
        Link head = GenerateLink();
        bool isOdd = true;
        Link middle = GetMiddleOne(head, ref isOdd);
        if (isOdd)
        {
            Console.WriteLine(middle.Data);
        }
        else
        {
            Console.WriteLine(middle.Data);
            Console.WriteLine(middle.Next.Data);
        }
        Console.Read();
```

```
}
static Link GetMiddleOne(Link head, ref bool isOdd)
{
    Link first = head;
    Link second = head;
    while (first != null && first.Next != null)
    {
        first = first.Next.Next;
        second = second.Next;
    }
    if (first != null)
        isOdd = false;
    return second;
}
```

4.一个单链表，很长，遍历一遍很慢，我们仅知道一个指向某节点的指针 curr，而我们又想删除这个节点。
这道题目是典型的“狸猫换太子”，如下图所示：

如果不考虑任何特殊情况，代码就 2 行：

```
curr.Data = curr.Next.Data;

curr.Next = curr.Next.Next;
```

上述代码由一个地方需要注意，就是如果要删除的是最后一个元素呢？那就只能从头遍历一次找到倒数第二个节点了。

此外，这道题目的一个变身就是将一个环状单链表拆开（即删除其中一个元素），此时，只要使用上面那两行代码就可以了，不需要考虑表尾。
相关问题：只给定单链表中某个结点 p(非空结点)，在 p 前面插入一个结点 q。
话说，交换单链表任意两个节点，也可以用交换值的方法。但这样就没意思了，所以，才会有第 7 题霸王硬上工的做法。

5.两个不交叉的有序链表的合并
有两个有序链表，各自内部是有序的，但是两个链表之间是无序的。
算法思路：当然是循环逐项比较两个链表了，如果一个到了头，就不比较了，直接加上去。

注意，对于 2 个元素的 Data 相等（仅仅是 Data 相等哦，而不是相同的引用），我们可以把它视作前面的 Data 大于后面的 Data，从而节省了算法逻辑。

```
static Link MergeTwoLink(Link head1, Link head2)
{
```

```
        Link head = new Link(null, Int16.MinValue);
        Link pre = head;
        Link curr = head.Next;
        Link curr1 = head1;
        Link curr2 = head2;
        //compare until one link run to the end
        while (curr1.Next != null && curr2.Next != null)
        {
            if (curr1.Next.Data < curr2.Next.Data)
            {
                curr = new Link(null, curr1.Next.Data);
                curr1 = curr1.Next;
            }
            else
            {
                curr = new Link(null, curr2.Next.Data);
                curr2 = curr2.Next;
            }
            pre.Next = curr;
            pre = pre.Next;
        }
        //if head1 run to the end
        while (curr1.Next != null)
        {
            curr = new Link(null, curr1.Next.Data);
            curr1 = curr1.Next;
            pre.Next = curr;
            pre = pre.Next;
        }
        //if head2 run to the end
        while (curr2.Next != null)
        {
            curr = new Link(null, curr2.Next.Data);
            curr2 = curr2.Next;
            pre.Next = curr;
            pre = pre.Next;
        }
        return head;
    }
```

如果这两个有序链表交叉组成了 Y 型呢，比如说：

这时我们需要先找出这个交叉点（图中是 11），这个算法参见第 9 题，我们这里直接使用

第 10 道题目中的方法 GetIntersect。
然后局部修改上面的算法，只要其中一个链表到达了交叉点，就直接把另一个链表的剩余元素都加上去。如下所示：

```
static Link MergeTwoLink2(Link head1, Link head2)
{
    Link head = new Link(null, Int16.MinValue);
    Link pre = head;
    Link curr = head.Next;
    Link intersect = GetIntersect(head1, head2);
    Link curr1 = head1;
    Link curr2 = head2;
    //compare until one link run to the intersect
    while (curr1.Next != intersect && curr2.Next != intersect)
    {
        if (curr1.Next.Data < curr2.Next.Data)
        {
            curr = new Link(null, curr1.Next.Data);
            curr1 = curr1.Next;
        }
        else
        {
            curr = new Link(null, curr2.Next.Data);
            curr2 = curr2.Next;
        }
        pre.Next = curr;
        pre = pre.Next;
    }
    //if head1 run to the intersect
    if (curr1.Next == intersect)
    {
        while (curr2.Next != null)
        {
            curr = new Link(null, curr2.Next.Data);
            curr2 = curr2.Next;
            pre.Next = curr;
            pre = pre.Next;
        }
    }
    //if head2 run to the intersect
    else if (curr2.Next == intersect)
    {
        while (curr1.Next != null)
        {
            curr = new Link(null, curr1.Next.Data);
```

```
                curr1 = curr1.Next;
                pre.Next = curr;
                pre = pre.Next;
            }
        }
        return head;
}
```

6.有个二级单链表，其中每个元素都含有一个指向一个单链表的指针。写程序把这个二级链表展开称一级单链表。
这个简单，就是说，这个二级单链表只包括一些 head：

```
public class Link
{
        public Link Next;
        public int Data;
        public Link(Link next, int data)
        {
            this.Next = next;
            this.Data = data;
        }
}
public class CascadeLink
{
        public Link Next;
        public CascadeLink NextHead;
        public CascadeLink(CascadeLink nextHead, Link next)
        {
            this.Next = next;
            this.NextHead = nextHead;
        }
}
```

下面做一个二级单链表，GenerateLink1 和 GenerateLink2 方法在前面都已经介绍过了：

```
public static CascadeLink GenerateCascadeLink()
{
        Link head1 = GenerateLink1();
        Link head2 = GenerateLink2();
        Link head3 = GenerateLink1();
        CascadeLink element3 = new CascadeLink(null, head3);
        CascadeLink element2 = new CascadeLink(element3, head2);
        CascadeLink element1 = new CascadeLink(element2, head1);
        CascadeLink head = new CascadeLink(element1, null);
```

```
    return head;
}
```
就是说，这些单链表的表头 head1、head2、head3、head4……，它们组成了一个二级单链表 head：null －> head1 －> head2 －> head3 －> head4 －>

我们的算法思想是：进行两次遍历，在外层用 curr1 遍历二级单链表 head，在内层用 curr2 遍历每个单链表：

```
public static Link GenerateNewLink(CascadeLink head)
{
    CascadeLink curr1 = head.NextHead;
    Link newHead = curr1.Next;
    Link curr2 = newHead;
    while (curr1 != null)
    {
        curr2.Next = curr1.Next.Next;
        while (curr2.Next != null)
        {
            curr2 = curr2.Next;
        }
        curr1 = curr1.NextHead;
    }
    return newHead;
}
```

其中，curr2.Next = curr1.Next.Next; 这句话是关键，它负责把上一个单链表的表尾和下一个单链表的非空表头连接起来。

7.单链表交换任意两个元素（不包括表头）

先一次遍历找到这两个元素 curr1 和 curr2，同时存储这两个元素的前驱元素 pre1 和 pre2。

然后大换血

```
public static Link SwitchPoints(Link head, Link p, Link q)
{
    if (p == head || q == head)
        throw new Exception("No exchange with head");
    if (p == q)
        return head;
    //find p and q in the link
    Link curr = head;
    Link curr1 = p;
```

```
    Link curr2 = q;
    Link pre1 = null;
    Link pre2 = null;

    int count = 0;
while (curr != null)
    {
        if (curr.Next == p)
        {
            pre1 = curr;
            count++;
            if (count == 2)
                break;
        }
        else if (curr.Next == q)
        {
            pre2 = curr;
            count++;
            if (count == 2)
                break;
        }
        curr = curr.Next;
    }
    curr = curr1.Next;
    pre1.Next = curr2;
    curr1.Next = curr2.Next;
    pre2.Next = curr1;
    curr2.Next = curr;
    return head;
}
```

注意特例，如果相同元素，就没有必要交换；如果有一个是表头，就不交换。

8.判断单链表是否有环？如何找到环的“起始”点？如何知道环的长度？

算法思想：

先分析是否有环。为此我们建立两个指针，从 header 一起向前跑，一个步长为 1，一个步长为 2，用 while（直到步长 2 的跑到结尾）检查两个指针是否相等，直到找到为止。

```
static bool JudgeCircleExists(Link head)
{
    Link first = head;    //1 step each time
    Link second = head; //2 steps each time
    while (second.Next != null && second.Next.Next != null)
    {
        second = second.Next.Next;
```

```
            first = first.Next;
            if (second == first)
                return true;
        }
        return false;
}
```

那又如何知道环的长度呢？
根据上面的算法，在返回 true 的地方，也就是 2 个指针相遇处，这个位置的节点 P 肯定位于环上。我们从这个节点开始先前走，转了一圈肯定能回来：

```
static int GetCircleLength(Link point)
{
        int length = 1;
        Link curr = point;
        while (curr.Next != point)
        {
            length++;
            curr = curr.Next;
        }
         return length;
}
```

继续我们的讨论，如何找到环的“起始”点呢？
延续上面的思路，我们仍然在返回 true 的地方 P，计算一下从有环单链表的表头 head 到 P 点的距离

```
static int GetLengthFromHeadToPoint(Link head, Link point)
{
        int length = 1;
        Link curr = head;
        while (curr != point)
        {
            length++;
            curr = curr.Next;
        }
        return length;
}
```

如果我们把环从 P 点“切开”（当然并不是真的切，那就破坏原来的数据结构了），那么问题就转化为计算两个相交“单链表”的交点（第 10 题）：
一个单链表是从 P 点出发，到达 P（一个回圈），距离 M；另一个单链表从有环单链表的表头 head 出发，到达 P，距离 N。

我们可以参考第 10 题的 GetIntersect 方法并稍作修改。

```
private static Link FindIntersect(Link head)
{
    Link p = null;
    //get the point in the circle
    bool result = JudgeCircleExists(head, ref p);
    if (!result) return null;
    Link curr1 = head.Next;
    Link curr2 = p.Next;
    //length from head to p
    int M = 1;
    while (curr1 != p)
    {
        M++;
        curr1 = curr1.Next;
    }
    //circle length
    int N = 1;
    while (curr2 != p)
    {
        N++;
        curr2 = curr2.Next;
    }
    //recover curr1 & curr2
    curr1 = head.Next;
    curr2 = p.Next;
    //make 2 links have the same distance to the intersect
    if (M > N)
    {
        for (int i = 0; i < M - N; i++)
            curr1 = curr1.Next;
    }
    else if (M < N)
    {
        for (int i = 0; i < N - M; i++)
            curr2 = curr2.Next;
    }
    //goto the intersect
    while (curr1 != p)
    {
        if (curr1 == curr2)
        {
            return curr1;
        }
```

```
            curr1 = curr1.Next;
            curr2 = curr2.Next;
        }
        return null;
}
```

9.判断两个单链表是否相交
这道题有多种算法。
算法 1：把第一个链表逐项存在 hashtable 中，遍历第 2 个链表的每一项，如果能在第一个链表中找到，则必然相交。

```
static bool JudgeIntersectLink1(Link head1, Link head2)
{
        Hashtable ht = new Hashtable();
        Link curr1 = head1;
        Link curr2 = head2;
        //store all the elements of link1
        while (curr1.Next != null)
        {
            ht[curr1.Next] = string.Empty;
            curr1 = curr1.Next;
        }
        //check all the elements in link2 if exists in Hashtable or not
        while (curr2.Next != null)
        {
            //if exists
            if (ht[curr2.Next] != null)
            {
                return true;
            }
            curr2 = curr2.Next;
        }
        return false;
}
```

算法 2：把一个链表 A 接在另一个链表 B 的末尾，如果有环，则必然相交。如何判断有环呢？从 A 开始遍历，如果能回到 A 的表头，则肯定有环。
注意，在返回结果之前，要把刚才连接上的两个链表断开，恢复原状。

```
static bool JudgeIntersectLink2(Link head1, Link head2)
{
        bool exists = false;
        Link curr1 = head1;
        Link curr2 = head2;
```

```
    //goto the end of the link1
    while (curr1.Next != null)
    {
        curr1 = curr1.Next;
    }
    //join these two links
    curr1.Next = head2;
    //iterate link2
    while (curr2.Next != null)
    {
        if (curr2.Next == head2)
        {
            exists = true;
            break;
        }
        curr2 = curr2.Next;
    }
    //recover original status, whether exists or not
    curr1.Next = null;
    return exists;
}
```

算法 3：如果两个链表的末尾元素相同，则必相交。

```
static bool JudgeIntersectLink3(Link head1, Link head2)
{
    Link curr1 = head1;
    Link curr2 = head2;
    //goto the end of the link1
    while (curr1.Next != null)
    {
        curr1 = curr1.Next;
    }
    //goto the end of the link2
    while (curr2.Next != null)
    {
        curr2 = curr2.Next;
    }
    if (curr1 != curr2)
        return false;
    else
        return true;
}
```

10.两个单链表相交，计算相交点

分别遍历两个单链表，计算出它们的长度 M 和 N，假设 M 比 N 大，则长度 M 的链表先前进 M-N，然后两个链表同时以步长 1 前进，前进的同时比较当前的元素，如果相同，则必是交点。

```
public static Link GetIntersect(Link head1, Link head2)
{
    Link curr1 = head1;
    Link curr2 = head2;
    int M = 0, N = 0;
    //goto the end of the link1
    while (curr1.Next != null)
    {
        curr1 = curr1.Next;
        M++;
    }
    //goto the end of the link2
    while (curr2.Next != null)
    {
        curr2 = curr2.Next;
        N++;
    }
    //return to the begining of the link
    curr1 = head1;
    curr2 = head2;
    if (M > N)
    {
        for (int i = 0; i < M - N; i++)
            curr1 = curr1.Next;
    }
    else if (M < N)
    {
        for (int i = 0; i < N - M; i++)
            curr2 = curr2.Next;
    }
    while (curr1.Next != null)
    {
        if (curr1 == curr2)
        {
            return curr1;
        }
        curr1 = curr1.Next;
        curr2 = curr2.Next;
    }
```

```
    return null;
}
```

11.用链表模拟大整数加法运算

例如：9>9>9>NULL + 1>NULL =>

1>0>0>0>NULL

肯定是使用递归啦，不然没办法解决进位+1 问题，因为这时候要让前面的节点加 1，而我们的单链表是永远指向前的。

此外对于 999+1=1000，新得到的值的位数（4 位）比原来的两个值（1 个 1 位，1 个 3 位）都多，所以我们将表头的值设置为 0，如果多出一位来，就暂时存放到表头。递归结束后，如果表头为 1，就在新的链表外再加一个新的表头。

```
//head1 length > head2, so M > N
public static int Add(Link head1, Link head2, ref Link newHead, int M, int N)
{
    // goto the end
    if (head1 == null)
        return 0;
    int temp = 0;
    int result = 0;
    newHead = new Link(null, 0);
    if (M > N)
    {
        result = Add(head1.Next, head2, ref newHead.Next, M - 1, N);
        temp = head1.Data + result;
        newHead.Data = temp % 10;
        return temp >= 10
 1 : 0;
    }
    else // M == N
    {
        result = Add(head1.Next, head2.Next, ref newHead.Next, M - 1, N - 1);
        temp = head1.Data + head2.Data + +result;
        newHead.Data = temp % 10;
        return temp >= 10
 1 : 0;
    }
}
```

这里假设 head1 比 head2 长，而且 M、N 分别是 head1 和 head2 的长度。

12.单链表排序

无外乎是冒泡、选择、插入等排序方法。关键是交换算法，需要额外考虑。第 7 题我编写了一个交换算法，在本题的排序过程中，我们可以在外层和内层循环里面，捕捉到 pre1

和 pre2，然后进行交换，而无需每次交换又要遍历一次单链表。
在实践中，我发现冒泡排序和选择排序都要求内层循环从链表的末尾向前走，这明显是不合时宜的。
所以我最终选择了插入排序算法，如下所示：
先给出基于数组的算法：

代码

```
staticint[]
InsertSort(int[]arr)
{
for(inti=1;i<arr.Length;i++)
{
for(intj=i;(j>0)&&arr[j]<arr[j-1];j--)
{
arr[j]=arr[j]^arr[j-1];
arr[j-1]=arr[j]^arr[j-1];
arr[j]=arr[j]^arr[j-1];
}
}

returnarr;
}
```

仿照上面的思想，我们来编写基于 Link 的算法：

```
public static Link SortLink(Link head)
{
    Link pre1 = head;
    Link pre2 = head.Next;
    Link min = null;
    for (Link curr1 = head.Next; curr1 != null; curr1 = min.Next)
    {
        if (curr1.Next == null)
            break;
        min = curr1;
        for (Link curr2 = curr1.Next; curr2 != null; curr2 = curr2.Next)
        {
            //swap curr1 and curr2
            if (curr2.Data < curr1.Data)
            {
                min = curr2;
                curr2 = curr1;
                curr1 = min;
                pre1.Next = curr1;
                curr2.Next = curr1.Next;
```

```
                    curr1.Next = pre2;
                    //if exchange element n-1 and n, no need to add reference from pre2 to curr2, because they are the same one
                    if (pre2 != curr2)
                        pre2.Next = curr2;
                }
                pre2 = curr2;
            }
            pre1 = min;
            pre2 = min.Next;
        }
        return head;
}
```

值得注意的是，很多人的算法不能交换相邻两个元素，这是因为 pre2 和 curr2 是相等的，如果此时还执行 pre2.Next = curr2; 会造成一个自己引用自己的环。

交换指针很是麻烦，而且效率也不高，需要经常排序的东西最好不要用链表来实现，还是数组好一些。

13.删除单链表中重复的元素

用 Hashtable 辅助，遍历一遍单链表就能搞定。

实践中发现，curr 从表头开始，每次判断下一个元素 curr.Netx 是否重复，如果重复直接使用 curr.Next = curr.Next.Next; 就可以删除重复元素——这是最好的算法。唯一的例外就是表尾，所以到达表尾，就 break 跳出 while 循环。

```
public static Link DeleteDuplexElements(Link head)
{
    Hashtable ht = new Hashtable();
    Link curr = head;
    while (curr != null)
    {
        if (curr.Next == null)
        {
            break;
        }
        if (ht[curr.Next.Data] != null)
        {
            curr.Next = curr.Next.Next;
        }
        else
        {
```

```
                ht[curr.Next.Data] = "";
            }
            curr = curr.Next;
        }
        return head;
}
```

结语：
单链表只有一个向前指针 Next，所以要使用 1-2 个额外变量来存储当前元素的前一个或后一个指针。
尽量用 while 循环而不要用 for 循环，来进行遍历。
哇塞，我就是不用指针，照样能“修改地址”，达到和 C++同样的效果，虽然很烦~
遍历的时候，不要在 while 循环中 head=head.Next;这样会改变原先的数据结构。我们要这么写：Link curr=head;然后 curr=curr.Next;
有时我们需要临时把环切开，有时我们需要临时把单链表首尾相连成一个环。
究竟是玩 curr 还是 curr.Next，根据不同题目而各有用武之地，没有定论，不必强求。

二、栈和队列

目录：

1.设计含 min 函数的栈，要求 min、push 和 pop 的时间复杂度都是 o(1)。
算法思想：需要设计一个辅助栈，用来存储当前栈中元素的最小值。网上有人说存储当前栈中元素的最小值的所在位置，虽然能节省空间，这其实是不对的，因为我在调用 Min 函数的时候，只能得到位置，还要对存储元素的栈不断的 pop，才能得到最小值——时间复杂度 o(1)。
所以，还是在辅助栈中存储元素吧。
此外，还要额外注意 Push 操作，第一个元素不用比较，自动成为最小值入栈。其它元素每次都要和栈顶元素比较，小的那个放到栈顶。

```
public class NewStack
{
    private Stack dataStack;
    private Stack mindataStack;
public NewStack()
    {
        dataStack = new Stack();
        mindataStack = new Stack();
    }
public void Push(int element)
    {
        dataStack.Push(element);
if (mindataStack.Count == 0)
            mindataStack.Push(element);
        else if (element <= (int)mindataStack.Peek())
            mindataStack.Push(element);
        else //(element > mindataStack.Peek)
            mindataStack.Push(mindataStack.Peek());
    }

    public int Pop()
    {
        if (dataStack.Count == 0)
            throw new Exception("The stack is empty");

        mindataStack.Pop();
        return (int)dataStack.Pop();
    }
public int Min()
    {
        if (dataStack.Count == 0)
            throw new Exception("The stack is empty");

        return (int)mindataStack.Peek();
    }
}
```

2.设计含 min 函数的栈的另解

话说，和青菜脸呆久了，就沾染了上海小市民意识，再加上原本我就很抠门儿，于是对于上一题目，我把一个栈当成两个用，就是说，每次 push，先入站当前元素，然后入栈当前栈中最小元素；pop 则每次弹出 2 个元素。

算法代码如下所示（这里最小元素位于当前元素之上，为了下次比较方便）：

```
public class NewStack
{
    private Stack stack;
public NewStack()
    {
        stack = new Stack();
    }
public void Push(int element)
    {
        if (stack.Count == 0)
        {
            stack.Push(element);
            stack.Push(element);
        }
        else if (element <= (int)stack.Peek())
        {
            stack.Push(element);
            stack.Push(element);
        }
        else //(element > stack.Peek)
        {
            object min = stack.Peek();
            stack.Push(element);
            stack.Push(min);
        }
    }
public int Pop()
    {
        if (stack.Count == 0)
            throw new Exception("The stack is empty");
stack.Pop();
        return (int)stack.Pop();
    }
public int Min()
    {
        if (stack.Count == 0)
            throw new Exception("The stack is empty");
return (int)stack.Peek();
    }
}
```

之所以说我这个算法比较叩门，是因为我只使用了一个栈，空间复杂度 o(N)，节省了一半

的空间（算法 1 的空间复杂度 o(2N)）。

3.用两个栈实现队列

实现队列，就要实现它的 3 个方法：Enqueue（入队）、Dequeue（出队）和 Peek（队头）。

1）stack1 存的是每次进来的元素，所以 Enqueue 就是把进来的元素 push 到 stack1 中。

2）而对于 Dequeue，一开始 stack2 是空的，所以我们把 stack1 中的元素全都 pop 到 stack2 中，这样 stack2 的栈顶就是队头。只要 stack2 不为空，那么每次出队，就相当于 stack2 的 pop。

3）接下来，每入队一个元素，仍然 push 到 stack1 中。每出队一个元素，如果 stack2 不为空，就从 stack2 中 pop 一个元素；如果 stack2 为空，就重复上面的操作——把 stack1 中的元素全都 pop 到 stack2 中。

4）Peek 操作，类似于 Dequeue，只是不需要出队，所以我们调用 stack2 的 Peek 操作。当然，如果 stack2 为空，就把 stack1 中的元素全都 pop 到 stack2 中。

5）注意边界的处理，如果 stack2 和 stack1 都为空，才等于队列为空，此时不能进行 Peek 和 Dequeue 操作。

按照上述分析，算法实现如下：

```
public class NewQueue
{
    private Stack stack1;
    private Stack stack2;
public NewQueue()
    {
        stack1 = new Stack();
        stack2 = new Stack();
    }
public void Enqueue(int element)
    {
        stack1.Push(element);
    }
public int Dequeue()
    {
        if (stack2.Count == 0)
        {
            if (stack1.Count == 0)
                throw new Exception("The queue is empty");
            else
                while (stack1.Count > 0)
                    stack2.Push(stack1.Pop());
        }
return (int)stack2.Pop();
```

```
    }
public int Peek()
    {
        if (stack2.Count == 0)
        {
            if (stack1.Count == 0)
                throw new Exception("The queue is empty");
            else
                while (stack1.Count > 0)
                    stack2.Push(stack1.Pop());
        }
return (int)stack2.Peek();
    }
}
```

4.用两个队列实现栈

这个嘛，就要 queue1 和 queue2 轮流存储数据了。这个“轮流”发生在 Pop 和 Peek 的时候，假设此时我们把所有数据存在 queue1 中（此时 queue2 为空），我们把 queue1 的 n-1 个元素放到 queue2 中，queue 中最后一个元素就是我们想要 pop 的元素，此时 queue2 存有 n-1 个元素（queue1 为空）。

至于 Peek，则是每次转移 n 个数据，再转移最后一个元素的时候，将其计下并返回。

那么 Push 的操作，则需要判断当前 queue1 和 queue2 哪个为空，将新元素放到不为空的队列中。

```
public class NewStack
{
    private Queue queue1;
    private Queue queue2;
public NewStack()
    {
        queue1 = new Queue();
        queue2 = new Queue();
    }
public void Push(int element)
    {
        if (queue1.Count == 0)
            queue2.Enqueue(element);
        else
            queue1.Enqueue(element);
    }
public int Pop()
    {
        if (queue1.Count == 0 && queue2.Count == 0)
```

```
                throw new Exception("The stack is empty");
if (queue1.Count > 0)
            {
                while (queue1.Count > 1)
                {
                    queue2.Enqueue(queue1.Dequeue());
                }
//还剩一个
                return (int)queue1.Dequeue();
            }
            else    //queue2.Count > 0
            {
                while (queue2.Count > 1)
                {
                    queue1.Enqueue(queue2.Dequeue());
                }
//还剩一个
                return (int)queue2.Dequeue();
            }
        }
public int Peek()
        {
            if (queue1.Count == 0 && queue2.Count == 0)
                throw new Exception("The stack is empty");
int result = 0;
if (queue1.Count > 0)
            {
                while (queue1.Count > 1)
                {
                    queue2.Enqueue(queue1.Dequeue());
                }
//还剩一个
                result = (int)queue1.Dequeue();
                queue2.Enqueue(result);
}
            else    //queue2.Count > 0
            {
                while (queue2.Count > 1)
                {
                    queue1.Enqueue(queue2.Dequeue());
                }
//还剩一个
                result = (int)queue2.Dequeue();
                queue1.Enqueue(result);
```

```
            }
return result;
      }
}
```

5.栈的 push、pop 序列是否一致
输入两个整数序列。其中一个序列表示栈的 push 顺序，判断另一个序列有没有可能是对应的 pop 顺序。为了简单起见，我们假设 push 序列的任意两个整数都是不相等的。
比如输入的 push 序列是 1、2、3、4、5，那么 4、5、3、2、1 就有可能是一个 pop 系列。因为可以有如下的 push 和 pop 序列：push 1，push 2，push 3，push 4，pop，push 5，pop，pop，pop，pop，这样得到的 pop 序列就是 4、5、3、2、1。但序列
4、3、5、1、2 就不可能是 push 序列 1、2、3、4、5 的 pop 序列。

网上的若干算法都太复杂了，现提出包氏算法如下：
先 for 循环把 arr1 中的元素入栈，并在每次遍历时，检索 arr2 中可以 pop 的元素。如果循环结束，而 stack 中还有元素，就说明 arr2 序列不是 pop 序列。

```
static bool
JudgeSequenceIsPossible(int[] arr1,int[] arr2)
{
Stackstack =newStack();
for(inti = 0, j = 0; i < arr1.Length; i++)
{
stack.Push(arr1[i]);
while(stack.Count > 0 && (int)stack.Peek() == arr2[j])
{
stack.Pop();
j++;
}
}
returnstack.Count == 0;
}
```

6.递归反转一个栈，要求不得重新申请一个同样的栈，空间复杂度 o(1)
算法思想：汉诺塔的思想，非常复杂，玩过九连环的人都想得通的

```
static void ReverseStack(ref Stack stack)
{
      if (stack.Count == 0)
            return;
```

```
object top = stack.Pop();
ReverseStack(ref stack);
if (stack.Count == 0)
    {
        stack.Push(top);
        return;
    }
object top2 = stack.Pop();
    ReverseStack(ref stack);
stack.Push(top);
    ReverseStack(ref stack);
    stack.Push(top2);
}
```

7.给栈排个序

本题目是上一题目的延伸

```
static void Sort(ref Stack stack)
{
    if (stack.Count == 0)
        return;
object top = stack.Pop();
Sort(ref stack);
if (stack.Count == 0)
    {
        stack.Push(top);
        return;
    }
object top2 = stack.Pop();
    if ((int)top > (int)top2)
    {
        stack.Push(top);
        Sort(ref stack);
        stack.Push(top2);
    }
    else
    {
        stack.Push(top2);
        Sort(ref stack);
        stack.Push(top);
    }
}
```

8..如何用一个数组实现两个栈
继续我所提倡的抠门儿思想，也不枉我和青菜脸相交一场。
网上流传着两种方法：
方法 1
采用交叉索引的方法
一号栈所占数组索引为 0, 2, 4, 6, 8.....(K*2)

二号栈所占数组索引为 1，3，5，7，9 .....(K*2 + 1)

算法实现如下：

```
public class NewStack
{
    object[] arr;
    int top1;
    int top2;
public NewStack(int capticy)
    {
        arr = new object[capticy];
top1 = -1;
        top2 = -2;
    }
public void Push(int type, object element)
    {
        if (type == 1)
        {
            if (top1 + 2 >= arr.Length)
                throw new Exception("The stack is full");
            else
            {
                top1 += 2;
                arr[top1] = element;
            }
        }
else //type==2
        {
            if (top2 + 2 >= arr.Length)
                throw new Exception("The stack is full");
            else
            {
                top2 += 2;
```

```
                    arr[top2] = element;
                }
            }
        }
public object Pop(int type)
        {
            object obj = null;
if (type == 1)
            {
                if (top1 == -1)
                    throw new Exception("The stack is empty");
                else
                {
                    obj = arr[top1];
                    arr[top1] = null;
                    top1 -= 2;
                }
            }
else //type == 2
            {
                if (top2 == -2)
                    throw new Exception("The stack is empty");
                else
                {
                    obj = arr[top2];
                    arr[top2] = null;
                    top2 -= 2;
                }
            }
return obj;
        }
public object Peek(int type)
        {
            if (type == 1)
            {
                if (top1 == -1)
                    throw new Exception("The stack is empty");
return arr[top1];
            }
else //type == 2
            {
                if (top2 == -2)
                    throw new Exception("The stack is empty");
return arr[top2];
```

```
            }
        }
}
```

方法 2：

第一个栈 A：从最左向右增长

第二个栈 B：从最右向左增长

代码实现如下：

```
public class NewStack
{
        object[] arr;
        int top1;
        int top2;
public NewStack(int captcy)
        {
                arr = new object[captcy];
top1 = 0;
                top2 = captcy;
        }
public void Push(int type, object element)
        {
                if (top1 == top2)
                        throw new Exception("The stack is full");
if (type == 1)
                {
                        arr[top1] = element;
                        top1++;
                }
                else //type==2
                {
                        top2--;
                        arr[top2] = element;
                }
        }
public object Pop(int type)
        {
                object obj = null;
if (type == 1)
                {
                        if (top1 == 0)
```

```
                throw new Exception("The stack is empty");
            else
            {
                top1--;
                obj = arr[top1];
                arr[top1] = null;
            }
        }
else //type == 2
        {
            if (top2 == arr.Length)
                throw new Exception("The stack is empty");
            else
            {
                obj = arr[top2];
                arr[top2] = null;
                top2++;
            }
        }
return obj;
    }
public object Peek(int type)
    {
        if (type == 1)
        {
            if (top1 == 0)
                throw new Exception("The stack is empty");
return arr[top1 - 1];
        }
else //type == 2
        {
            if (top2 == arr.Length)
                throw new Exception("The stack is empty");
return arr[top2];
        }
    }
}
```

综合比较上述两种算法，我们发现，算法 1 实现的两个栈，每个都只有 n/2 个空间大小；而算法 2 实现的两个栈，如果其中一个很小，另一个则可以很大，它们的和为常数 n。

9..如何用一个数组实现三个栈

最后，让我们把抠门儿进行到底，相信看完本文，你已经从物质和精神上都升级为一个抠门儿主义者。

如果还使用交叉索引的办法，每个栈都只有 N/3 个空间。
让我们只好使用上个题目的第 2 个方法，不过这只能容纳 2 个栈，我们还需要一个位置存放第 3 个栈，不如考虑数组中间的位置——第 3 个栈的增长规律可以如下：

第 1 个入栈 C 的元素进 mid 处

第 2 个入栈 C 的元素进 mid+1 处

第 3 个入栈 C 的元素进 mid-1 处

第 4 个入栈 C 的元素进 mid+2 处

这个方法的好处是，每个栈都有接近 N/3 个空间。

```
public class NewStack
{
    object[] arr;
    int top1;
    int top2;
int top3_left;
    int top3_right;
    bool isLeft;
public NewStack(int captiсy)
    {
        arr = new object[captiсy];
top1 = 0;
        top2 = captiсy;
isLeft = true;
        top3_left = captiсy / 2;
        top3_right = top3_left + 1;
    }
public void Push(int type, object element)
    {
        if (type == 1)
        {
            if (top1 == top3_left + 1)
                throw new Exception("The stack is full");
arr[top1] = element;
            top1++;
        }
        else if (type == 2)
        {
            if (top2 == top3_right)
                throw new Exception("The stack is full");
top2--;
            arr[top2] = element;
        }
```

```
            else //type==3
            {
                if (isLeft)
                {
                    if (top1 == top3_left + 1)
                        throw new Exception("The stack is full");
arr[top3_left] = element;
                    top3_left--;
                }
                else
                {
                    if (top2 == top3_right)
                        throw new Exception("The stack is full");
arr[top3_right] = element;
                    top3_right++;
                }
isLeft = !isLeft;
            }
        }
public object Pop(int type)
        {
            object obj = null;
if (type == 1)
            {
                if (top1 == 0)
                    throw new Exception("The stack is empty");
                else
                {
                    top1--;
                    obj = arr[top1];
                    arr[top1] = null;
                }
            }
            else if (type == 2)
            {
                if (top2 == arr.Length)
                    throw new Exception("The stack is empty");
                else
                {
                    obj = arr[top2];
                    arr[top2] = null;
                    top2++;
                }
            }
```

```
            else //type==3
            {
                if (top3_right == top3_left + 1)
                    throw new Exception("The stack is empty");
if (isLeft)
                {
                    top3_left++;
                    obj = arr[top3_left];
                    arr[top3_left] = null;
                }
                else
                {
                    top3_right--;
                    obj = arr[top3_right];
                    arr[top3_right] = null;
                }
isLeft = !isLeft;
            }
return obj;
        }
public object Peek(int type)
        {
            if (type == 1)
            {
                if (top1 == 0)
                    throw new Exception("The stack is empty");
return arr[top1 - 1];
            }
else if (type == 2)
            {
                if (top2 == arr.Length)
                    throw new Exception("The stack is empty");
return arr[top2];
            }
            else //type==3
            {
                if (top3_right == top3_left + 1)
                    throw new Exception("The stack is empty");
if (isLeft)
                    return arr[top3_left + 1];
                else
                    return arr[top3_right - 1];
            }
        }
```

}

三、二叉树

目录：

首先写一个二叉树的 C#实现，这是我们的基石：

```
public class BinNode
{
    public int Element;
    public BinNode Left;
    public BinNode Right;
    public BinNode(int element, BinNode left, BinNode right)
    {
        this.Element = element;
        this.Left = left;
        this.Right = right;
    }

    public bool IsLeaf()
    {
        return this.Left == null && this.Right == null;
    }
}
```

1.二叉树三种周游（traversal）方式：

1）前序周游（preorder）：节点－>子节点 Left（包括其子树）－>子节点 Right（包括其子树）

```
static void PreOrder(BinNode root)
{
    if (root == null)
        return;
    //visit current node
    Console.WriteLine(root.Element);
    PreOrder(root.Left);
    PreOrder(root.Right);
}
```

2）后序周游（postorder）：子节点 Left（包括其子树）－>子节点 Right（包括其子树）－>节点

```
static void PostOrder(BinNode root)
{
    if (root == null)
        return;
    PostOrder(root.Left);
    PostOrder(root.Right);
    //visit current node
    Console.WriteLine(root.Element);
}
```

3）中序周游（inorder）：子节点 Left（包括其子树）－>节点－>子节点 Right（包括其子树）

```
static void InOrder(BinNode root)
{
    if (root == null)
        return;
    InOrder(root.Left);
    //visit current node
    Console.WriteLine(root.Element);
    InOrder(root.Right);
}
```

我们发现，三种周游的 code 实现，仅仅是访问当前节点的这条语句所在位置不同而已。

2.怎样从顶部开始逐层打印二叉树结点数据
有 2 种算法：
算法 1：基于 Queue 来实现，也就是广度优先搜索（BFS）的思想

```
static void PrintTree1(BinNode root)
{
    if (root == null) return;
    BinNode tmp = null;
    Queue queue = new Queue();
    queue.Enqueue(root);
    while (queue.Count > 0)
    {
        tmp = (BinNode)queue.Dequeue();
        Console.WriteLine(tmp.Element);
        if (tmp.Left != null)
            queue.Enqueue(tmp.Left);
        if (tmp.Right != null)
            queue.Enqueue(tmp.Right);
    }
}
```

话说，BFS 和 DFS 思想本来是用于图的，但我们不能被传统的思维方式所束缚。

算法 2：基于单链表实现
如果没有 Queue 给我们用，我们只好使用单链表，把每个节点存在单链表的 Data 中，实现如下：

```
public class Link
{
    public Link Next;
    public BinNode Data;
    public Link(Link next, BinNode data)
    {
        this.Next = next;
        this.Data = data;
    }
}
```

看过了 Queue 的实现，我们发现永远是先出队 1 个（队头），然后入队 2 个（把出队的 Left 和 Right 放到队尾）。
对于单链表而言，我们可以先模拟入队——把 first 的 Data 所对应的 Left 和 Right，先后插到 second 的后面，即 second.Next 和 second.Next.Next 位置，同时 second 向前走 0、1 或 2 次，再次到达链表末尾，这取决于 Left 和 Right 是否为空；然后我们模拟出队——first 前进 1 步。

当 first 指针走不下去了，那么任务也就结束了。

```
static void PrintTree2(BinNode root)
{
    if (root == null) return;
    Link head = new Link(null, root);
    Link first = head;
    Link second = head;
    while (first != null)
    {
        if (first.Data.Left != null)
        {
            second.Next = new Link(null, first.Data.Left);
            second = second.Next;
        }
        if (first.Data.Right != null)
        {
            second.Next = new Link(null, first.Data.Right);
            second = second.Next;
        }
        Console.WriteLine(first.Data.Element);
        first = first.Next;
    }
}
```

3.如何判断一棵二叉树是否是平衡二叉树

平衡二叉树的定义，如果任意节点的左右子树的深度相差不超过 1，那这棵树就是平衡二叉树。

算法思路：先编写一个计算二叉树深度的函数 GetDepth，利用递归实现；然后再递归判断每个节点的左右子树的深度是否相差 1

```
static int GetDepth(BinNode root)
{
    if (root == null)
        return 0;
    int leftLength = GetDepth(root.Left);
    int rightLength = GetDepth(root.Right);
    return (leftLength > rightLength
 leftLength : rightLength) + 1;
}
```

注意这里的+1，对应于 root 不为空（算作当前 1 个深度）

```
static bool IsBalanceTree(BinNode root)
{
```

```
    if (root == null)
        return true;
    int leftLength = GetDepth(root.Left);
    int rightLength = GetDepth(root.Right);
    int distance = leftLength > rightLength
 leftLength - rightLength : rightLength - leftLength;

    if (distance > 1)
        return false;
    else
        return IsBalanceTree(root.Left) && IsBalanceTree(root.Right);
}
```

上述程序的逻辑是，只要当前节点 root 的 Left 和 Right 深度差不超过 1，就递归判断 Left 和 Right 是否也符合条件，直到为 Left 或 Right 为 null，这意味着它们的深度为 0，能走到这一步，前面必然都符合条件，所以整个二叉树都符合条件。

4.设计一个算法，找出二叉树上任意两个节点的最近共同父结点，复杂度如果是 $O(n^2)$则不得分。
本题网上有很多算法，都不怎么样。这里提出包氏的两个算法：
算法 1：做一个容器，我们在遍历二叉树寻找节点的同时，把从根到节点的路径扔进去（两个节点就是两个容器）。由于根节点最后一个被扔进去，但我们接下来又需要第一个就能访问到它——后进先出，所以这个容器是一个栈。时间复杂度 O(N)，空间复杂度 O(N)。

```
static bool GetPositionByNode(BinNode root, BinNode node, ref Stack stack)
{
    if (root == null)
        return false;
    if (root == node)
    {
        stack.Push(root);
        return true;
    }
    if (GetPositionByNode(root.Left, node, ref stack) || GetPositionByNode(root.Right, node, ref
stack))
    {
        stack.Push(root);
        return true;
    }
    return false;
}
```

然后我们要同时弹出这两个容器的元素，直到它们不相等，那么之前那个相等的元素就是我们要求的父亲节点。

```
static BinNode FindParentNode(BinNode root, BinNode node1, BinNode node2)
{
    Stack stack1 = new Stack();
    GetPositionByNode(root, node1, ref stack1);
    Stack stack2 = new Stack();
    GetPositionByNode(root, node2, ref stack2);
    BinNode tempNode = null;
    while (stack1.Peek() == stack2.Peek())
    {
        tempNode = (BinNode)stack1.Pop();
        stack2.Pop();
    }
    return tempNode;
}
```

算法 2：如果要求 o(1)的空间复杂度，就是说，只能用一个变量来辅助我们。

我们选择一个 64 位的整数，然后从 1 开始，从左到右逐层为二叉树的每个元素赋值，root 对应 1，root.Left 对应 2，root.Right 对应 3，依次类推，而不管实际这个位置上是否有节点，我们发现两个规律：

```
////                    1
////          2                   3
////      4         5         6         7
//// 8         9         10
```

如果要找的是 5 和 9 位置上的节点。

我们发现，它们的二进制分别是 101 和 1001，右移 1001 使之与 101 位数相同，于是 1001 变成了 100（也就是 9 的父亲 4）。

这时 101 和 100（也就是 4 和 5 位于同样的深度），我们从左往右找，101 和 100 具有 2 位相同，即 10，这就是我们要找的 4 和 5 的父亲，也就是 9 和 5 的最近父亲。

由上面观察，得到算法：

1)将找到的两个节点对应的数字

```
static bool GetPositionByNode(BinNode root, BinNode node, ref int pos)
{
    if (root == null)
        return false;
    if (root == node)
        return true;
    int temp = pos;
    //这么写很别扭，但是能保证只要找到就不再进行下去
    pos = temp * 2;
    if (GetPositionByNode(root.Left, node, ref pos))
    {
```

```
            return true;
        }
        else
        {
            //找不到左边找右边
            pos = temp * 2 + 1;
            return GetPositionByNode(root.Right, node, ref pos);
        }
    }
```

2)它们的二进制表示，从左向右逐一比较，直到一个结束或不再相同，则最大的相同子串，就是我们需要得到的最近父亲所对应的位置 K。

```
static int FindParentPosition(int larger, int smaller)
{
    if (larger == smaller) return larger;
    int left = GetLen(larger) - GetLen(smaller);
    while (left > 0)
    {
        larger = larger >> 1;
        left--;
    }
    while (larger != smaller)
    {
        larger = larger >> 1;
        smaller = smaller >> 1;
    }
    return smaller;
}
static int GetLen(int num)
{
    int length = 0;
    while (num != 0)
    {
        num = num >> 1;
        length++;
    }
    return length;
}
```

3)第 3 次递归遍历，寻找 K 所对应的节点。

函数 GetNodeByPosition 的思想是，先算出 k 在第几层 power，观察 k 的二进制表示，比如说 12，即 1100，从左向右数第一个位 1 不算，还剩下 100，1 表示向右走，0 表示向左走，于是从 root 出发，1->3->6->12。

```
static BinNode GetNodeByPosition(BinNode root, int num)
{
    if (num == 1) return root;
```

```
    int pow = (int)Math.Floor(Math.Log(num, 2)); //1 return 0, 2-3 return 1, 4-7 return 2
    //第一个位不算
    num -= 1 << pow;
    while (pow > 0)
    {
        if ((num & 1 << (pow - 1)) == 0)
            root = root.Left;
        else
            root = root.Right;
        pow--;
    }
    return root;
}
```

总结上面的 3 个步骤：

```
static BinNode FindParentNode(BinNode root, BinNode node1, BinNode node2)
{
    int pos1 = 1;
    GetPositionByNode(root, node1, ref pos1);
    int pos2 = 1;
    GetPositionByNode(root, node2, ref pos2);
    int parentposition = 0;
    if (pos1 >= pos2)
    {
        parentposition = FindParentPosition(pos1, pos2);
    }
    else //pos1<pos2
    {
        parentposition = FindParentPosition(pos2, pos1);
    }
    return GetNodeByPosition(root, parentposition);
}
```

5.如何不用递归实现二叉树的前序/后序/中序遍历？

算法思想：三种算法的思想都是让 root 的 Left 的 Left 的 Left 全都入栈。所以第一个 while 循环的逻辑，都是相同的。

下面详细分析第 2 个 while 循环，这是一个出栈动作，只要栈不为空，就始终要弹出栈顶元素，由于我们之前入栈的都是 Left 节点，所以每次在出栈的时候，我们都要考虑 Right 节点是否存在。因为前序/后序/中序遍历顺序的不同，所以在具体的实现上有略为区别。

1)前序遍历

这个是最简单的。

前序遍历是 root->root.Left->root.Right 的顺序。

因为在第一个 while 循环中，每次进栈的都可以认为是一个 root，所以我们直接打印，然后 root.Right 和 root.Left 先后进栈，那么出栈的时候，就能确保先左后右的顺序。

```
static void PreOrder(BinNode root)
{
    Stack stack = new Stack();
    BinNode temp = root;
    //入栈
    while (temp != null)
    {
        Console.WriteLine(temp.Element);
        if (temp.Right != null)
            stack.Push(temp.Right);
        temp = temp.Left;
    }
    //出栈，当然也有入栈
    while (stack.Count > 0)
    {
        temp = (BinNode)stack.Pop();
        Console.WriteLine(temp.Element);
        while (temp != null)
        {
            if (temp.Right != null)
                stack.Push(temp.Right);
            temp = temp.Left;
        }
    }
}
```

//后序遍历比较麻烦，需要记录上一个访问的节点，然后在本次循环中判断当前节点的 Right 或 Left 是否为上个节点，当前节点的 Right 为 null 表示没有右节点。

```
static void PostOrder(BinNode root)
{
    Stack stack = new Stack();
    BinNode temp = root;
    //入栈
    while (temp != null)
    {
        if (temp != null)
            stack.Push(temp);
        temp = temp.Left;
    }
    //出栈，当然也有入栈
    while (stack.Count > 0)
    {
        BinNode lastvisit = temp;
```

```
            temp = (BinNode)stack.Pop();
            if (temp.Right == null || temp.Right == lastvisit)
            {
                Console.WriteLine(temp.Element);
            }
            else if (temp.Left == lastvisit)
            {
                stack.Push(temp);
                temp = temp.Right;
                stack.Push(temp);
                while (temp != null)
                {
                    if (temp.Left != null)
                        stack.Push(temp.Left);
                    temp = temp.Left;
                }
            }
        }
    }
    //中序遍历，类似于前序遍历
    static void InOrder(BinNode root)
    {
        Stack stack = new Stack();
        BinNode temp = root;
        //入栈
        while (temp != null)
        {
            if (temp != null)
                stack.Push(temp);
            temp = temp.Left;
        }
        //出栈，当然也有入栈
        while (stack.Count > 0)
        {
            temp = (BinNode)stack.Pop();
            Console.WriteLine(temp.Element);
            if (temp.Right != null)
            {
                temp = temp.Right;
                stack.Push(temp);
                while (temp != null)
                {
                    if (temp.Left != null)
                        stack.Push(temp.Left);
```

```
                temp = temp.Left;
            }
        }
    }
}
```

6.在二叉树中找出和为某一值的所有路径

算法思想：这道题目的苦恼在于，如果用递归，只能打出一条路径来，其它符合条件的路径打不出来。

为此，我们需要一个 Stack，来保存访问过的节点，即在对该节点的递归前让其进栈，对该节点的递归结束后，再让其出栈——深度优先原则（DFS）。

此外，在递归中，如果发现某节点（及其路径）符合条件，如何从头到尾打印是比较头疼的，因为 DFS 使用的是 stack 而不是 queue，为此我们需要一个临时栈，来辅助打印。

```
static void FindBinNode(BinNode root, int sum, Stack stack)
{
    if (root == null)
        return;
    stack.Push(root.Element);
    //Leaf
    if (root.IsLeaf())
    {
        if (root.Element == sum)
        {
            Stack tempStack = new Stack();
            while (stack.Count > 0)
            {
                tempStack.Push(stack.Pop());
            }
            while (tempStack.Count > 0)
            {
                Console.WriteLine(tempStack.Peek());
                stack.Push(tempStack.Pop());
            }
            Console.WriteLine();
        }
    }
    if (root.Left != null)
        FindBinNode(root.Left, sum - root.Element, stack);
    if (root.Right != null)
        FindBinNode(root.Right, sum - root.Element, stack);
    stack.Pop();
}
```

7.怎样编写一个程序，把一个有序整数数组放到二叉树中？
算法思想：我们该如何构造这棵二叉树呢？当然是越平衡越好，如下所示：

```
          ////                    arr[0]
          ////          arr[1]                arr[2]
          //// arr[3]     arr[4]        arr[5]
```

相应编码如下：

```
public static void InsertArrayIntoTree(int[] arr, int pos, ref BinNode root)
{
    root = new BinNode(arr[pos], null, null);
    root.Element = arr[pos];
    //if Left value less than arr length
    if (pos * 2 + 1 > arr.Length - 1)
    {
        return;
    }
    else
    {
        InsertArrayIntoTree(arr, pos * 2 + 1, ref root.Left);
    }
    //if Right value less than arr length
    if (pos * 2 + 2 > arr.Length - 1)
    {
        return;
    }
    else
    {
        root.Right = new BinNode(arr[pos * 2 + 2], null, null);
        InsertArrayIntoTree(arr, pos * 2 + 2, ref root.Right);
    }
}
```

8.判断整数序列是不是二叉搜索树的后序遍历结果
比如，给你一个数组： int a[] = [1, 6, 4, 3, 5] ，则 F(a) => false
算法思想：在后续遍历得到的序列中，最后一个元素为树的根结点。从头开始扫描这个序列，比根结点小的元素都应该位于序列的左半部分；从第一个大于跟结点开始到跟结点前面的一个元素为止，所有元素都应该大于跟结点，因为这部分元素对应的是树的右子树。根据这样的划分，把序列划分为左右两部分，我们递归地确认序列的左、右两部分是不是都是二元查找树。
由于不能使用动态数组，所以我们每次递归都使用同一个数组 arr，通过 start 和 length 来模拟“部分”数组。

```
public static bool VerifyArrayOfBST(int[] arr, int start, int length)
{
    if (arr == null || arr.Length == 0 || arr.Length == 1)
    {
        return false;
    }
    int root = arr[length + start - 1];
    int i = start;
    for (; i < length - 1; i++)
    {
        if (arr[i] >= root)
            break;
    }
    int j = i;
    for (; j < length - 1; j++)
    {
        if (arr[j] < root)
            return false;
    }
    bool left = true;
    if (i > start)
    {
        left = VerifyArrayOfBST(arr, start, i - start);
    }
    bool right = true;
    if (j > i)
    {
        right = VerifyArrayOfBST(arr, i, j - i + 1);
    }
    return left && right;
}
```

9.求二叉树的镜像

算法 1：利用上述遍历二叉树的方法（比如说前序遍历），把访问操作修改为交换左右节点的逻辑：

```
static void PreOrder(ref BinNode root)
{
    if (root == null)
        return;
    //visit current node
    BinNode temp = root.Left;
    root.Left = root.Right;
    root.Right = temp;
```

```
        PreOrder(ref root.Left);
        PreOrder(ref root.Right);
}
```

算法 2：使用循环也可以完成相同的功能。

```
static void PreOrder2(ref BinNode root)
{
        if (root == null)
                return;
        Stack stack = new Stack();
        stack.Push(root);
        while (stack.Count > 0)
        {
                //visit current node
                BinNode temp = root.Left;
                root.Left = root.Right;
                root.Right = temp;
if (root.Left != null)
                        stack.Push(root.Left);
                if (root.Right != null)
                        stack.Push(root.Right);
        }
}
```

10.一棵排序二叉树（即二叉搜索树 BST），令　f=(最大值+最小值)/2，设计一个算法，找出距离 f 值最近、大于 f 值的结点。复杂度如果是 $O(n^2)$则不得分。

算法思想：最小最大节点分别在最左下与最右下节点，O(N)

```
static BinNode Find(BinNode root)
{
        BinNode min = FindMinNode(root);
        BinNode max = FindMaxNode(root);
        double find = (double)(min.Element + max.Element) / 2;
        return FindNode(root, find);
}


static BinNode FindMinNode(BinNode root)
{
        BinNode min = root;
        while (min.Left != null)
        {
                min = min.Left;
```

```
        }
        return min;
}
static BinNode FindMaxNode(BinNode root)
{
        BinNode max = root;
        while (max.Right != null)
        {
                max = max.Right;
        }
        return max;
}
```

递归寻找 BST 的节点，O(logN)。

```
static BinNode FindNode(BinNode root, double mid)
{
        //如果小于相等，则从右边找一个最小值
        if (root.Element <= mid)
        {
                if (root.Right == null)
                        return root;
                BinNode find = FindNode(root.Right, mid);
                //不一定找得到
                return find.Element < mid
  root : find;
        }
        //如果大于，则找到 Left
        else    //temp.Element > find
        {
                if (root.Left == null)
                        return root;
                BinNode find = FindNode(root.Left, mid);
                //不一定找得到
                return find.Element < mid
  root : find;
        }
}
```

11.把二叉搜索树转变成排序的双向链表，如

```
////              13
////        10          15
//// 5           11          17
////                    16          22
```

转变为 Link：5=10=11=13=15=16=17=22
算法思想：这个就是中序遍历啦，因为 BST 的中序遍历就是一个从小到大的访问顺序。局部修改中序遍历算法，于是有如下代码：

```
static void ConvertNodeToLink(BinNode root, ref DoubleLink link)
{
    if (root == null)
        return;
    BinNode temp = root;
    if (temp.Left != null)
        ConvertNodeToLink(temp.Left, ref link);
    //visit current node
    link.Next = new DoubleLink(link, null, root);
    link = link.Next;
    if (temp.Right != null)
        ConvertNodeToLink(temp.Right, ref link);
}
```

但是我们发现，这样得到的 Link 是指向双链表最后一个元素 22，而我们想要得到的是表头 5，为此，我们不得不额外进行 while 循环，将指针向前移动到表头：

```
static DoubleLink ReverseDoubleLink(BinNode root, ref DoubleLink link)
{
    ConvertNodeToLink(root, ref link);
    DoubleLink temp = link;
    while (temp.Prev != null)
    {
        temp = temp.Prev;
    }
    return temp;
}
```

这么写有点蠢，为什么不直接在递归中就把顺序反转呢？于是有算法 2：

算法 2：观察算法 1 的递归方法，访问顺序是 Left -> Root －> Right，所以我们要把访问顺序修改为 Right -> Root －> Left。
此外，算法的节点访问逻辑，是连接当前节点和新节点，同时指针 link 向前走，即
5=10=11=13=15=16=17=22=link
代码如下所示：

```
    link.Next = new DoubleLink(link, null, root);
link = link.Next;
```

那么，即使我们颠倒了访问顺序，新的 Link 也只是变为：22=17=16=15=13=11=10=5=link。
为此，我们修改上面的节点访问逻辑——将 Next 和 Prev 属性交换：

```
    link.Prev = new DoubleLink(null, link, root);
link = link.Prev;
```

这样，新的 Link 就变成这样的顺序了：link=5=10=11=13=15=16=17=22
算法代码如下所示：

```
static void ConvertNodeToLink2(BinNode root, ref DoubleLink link)
{
    if (root == null)
        return;
    BinNode temp = root;
    if (temp.Right != null)
        ConvertNodeToLink2(temp.Right, ref link);
    //visit current node
    link.Prev = new DoubleLink(null, link, root);
    link = link.Prev;
    if (temp.Left != null)
        ConvertNodeToLink2(temp.Left, ref link);
}
```

以下算法属于二叉树的基本概念，未列出：
1.Huffman Tree 的生成、编码和反编码
2.BST 的实现
3.Heap 的实现，优先队列

4.非平衡二叉树如何变成平衡二叉树？
http://www.cppblog.com/bellgrade/archive/2009/10/12/98402.html

玩二叉树，基本都要用到递归算法。
唉，对于递归函数，我一直纠结，到底要不要返回值？到底先干正事还是先递归？到底要不要破坏原来的数据结构？到底要不要额外做个 stack/queue/link/array 来转存，还是说完全在递归里面实现？到底终结条件要写成什么样子？ ref 在递归里面貌似用的很多哦。