设一组初始记录关键字序列为(20，18，22，16，30，19)，则以20为中轴的一趟快速排序结果为\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_。（答案中用顿号、分隔各数字）

19、18、16、20、30、22

在一个具有n个顶点的无向完全图中，包含有\_\_\_\_\_\_\_\_条边，在一个具有n个顶点的有向完全图中，包含有\_\_\_\_\_\_\_\_条边。

n(n-1)/2  
n(n-1)

设哈夫曼树中共有n个结点，则该哈夫曼树中有\_\_\_\_\_\_\_\_个度数为1的结点。

0

设有向图G用邻接矩阵A[n][n]作为存储结构，则该邻接矩阵中第i行上所有元素之和等于顶点i的\_\_\_\_\_\_\_\_，第i列上所有元素之和等于顶点i的\_\_\_\_\_\_\_\_。

出度  
入度

在快速排序、堆排序、归并排序中，\_\_\_\_\_\_\_\_\_排序是稳定的。

归并

设有一个顺序循环队列中有M个存储单元，采用空一个位的方式区分队列空和满，则该循环队列中最多能够存储\_\_\_\_\_\_\_\_个队列元素；若队首指针为F,队尾指针为R，则当前实际存储\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_个队列元素（设指针F指向当前队头元素的前一个位置，尾指针指向当前队尾元素的位置）。

m-1  
(R-F+M)%M

通常从四个方面评价算法的质量：\_\_\_\_\_\_\_\_\_、易读性、健壮性和高效率。

正确性

设某棵二叉树中度数为0的结点数为N0，度数为1的结点数为N1，则该二叉树中度数为2的结点数为\_\_\_\_\_\_\_\_\_；若采用二叉链表作为该二叉树的存储结构，则该二叉树中共有\_\_\_\_\_\_\_个空指针域。

N0-1  
2N0+N1

中序遍历二叉排序树所得到的序列是\_\_\_\_\_\_\_\_\_\_\_序列（填有序或无序）。

有序

设输入序列为1、2、3，则经过栈的作用后可以得到\_\_\_\_\_\_\_\_\_\_\_种不同的输出序列。

5

设有向图G的存储结构用邻接矩阵A来表示，则A中第i行中所有非零元素个数之和等于顶点i的\_\_\_\_\_\_\_\_，第i列中所有非零元素个数之和等于顶点i的\_\_\_\_\_\_\_\_\_\_。

出度  
入度

设一组初始记录关键字序列为(20，18，22，16，30，19)，则根据这些初始关键字序列建成的初始堆为\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_。（答案中用顿号、分隔各数字）

16、18、19、20、32、22

设无向图对应的邻接矩阵为A，则A中第i上非0元素的个数\_\_\_\_\_\_\_\_\_第i列上非0元素的个数（填等于，大于或小于）。

等于

设哈夫曼树中共有99个结点，则该树中有\_\_\_\_\_\_\_\_\_个叶子结点；若采用二叉链表作为存储结构，则该树中有\_\_\_\_\_个空指针域。

50  
51

设一棵完全二叉树中有500个结点，则该二叉树的深度为\_\_\_\_\_\_\_\_\_\_；若用二叉链表作为该完全二叉树的存储结构，则共有\_\_\_\_\_\_\_\_\_\_\_个空指针域。

9  
501

设有n个结点的完全二叉树，如果按照从自上到下、从左到右从1开始顺序编号，则第i个结点的双亲结点编号为\_\_\_\_\_\_\_\_\_\_\_\_，右孩子结点的编号为\_\_\_\_\_\_\_\_\_\_\_。

i/2  
2i+1

若用链表存储一棵二叉树时，每个结点除数据域外，还有指向左孩子和右孩子的两个指针。在这种存储结构中，n个结点的二叉树共有\_\_\_\_\_\_\_\_个指针域，其中有\_\_\_\_\_\_\_\_个指针域是存放了地址，有\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_个指针是空指针。

2n  
n-1  
n+1

设顺序线性表中有n个数据元素，则第i个位置上插入一个数据元素需要移动表中\_\_\_\_\_\_\_个数据元素；删除第i个位置上的数据元素需要移动表中\_\_\_\_\_\_\_个元素。

n+1-I  
n-i

对于一个具有n个顶点和e条边的有向图和无向图，在其对应的邻接表中，所含边结点分别有\_\_\_\_\_\_\_个和\_\_\_\_\_\_\_\_个。

e  
2e

设有向图G中有n个顶点e条有向边，所有的顶点入度数之和为d，则e和d的关系为\_\_\_\_\_\_\_\_\_。（填e=d、e>d、e<d,e>=d,或e<=d）

e=d

一个算法的时间复杂度为(n3+n2log2n+14n)/n2，其数量级表示为\_\_\_\_\_\_\_\_。（填类似O(log2n)、O(n)、O(nlog2n)、O(n\*n)）

O(n)

快速排序的最坏时间复杂度为\_\_\_\_\_\_\_\_\_\_\_，平均时间复杂度为\_\_\_\_\_\_\_\_\_\_。（填类似O(log2n)、O(n)、O(nlog2n)、O(n\*n)）

O(n)  
O(nlog2n)

设查找表中有100个元素，如果用二分法查找方法查找数据元素X，则最多需要比较\_\_\_\_\_\_\_\_次就可以断定数据元素X是否在查找表中。

7

后缀算式9 2 3 +- 10 2 / -的值为\_\_\_\_\_\_\_\_\_\_。中缀算式（3+4X）-2Y/3对应的后缀算式为\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_。

-1  
3 4 X \* + 2 Y \* 3 / -

设有向图G中有向边的集合E={<1，2>，<2，3>，<1，4>，<4，2>，<4，3>}，则该图的一种拓扑序列为\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_。（答案中用顿号“、”分隔各数字）

1、4、3、2

设一组初始记录关键字序列为(55，63，44，38，75，80，31，56)，则利用筛选法建立的初始堆为\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_。(结果写在小括号中，用“，”分隔)

(31，38，54，56，75，80，55，63)

设一组初始记录关键字为(72，73，71，23，94，16，5)，则以记录关键字72为基准的一趟快速排序结果为\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_。(结果写在小括号中，用“，”分隔)

(5，16，71，23，72，94，73)

AOV网是一种\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_的图。

有向无环图

设一棵完全二叉树的顺序存储结构中存储数据元素为ABCDEF，则该二叉树的前序遍历序列为\_\_\_\_\_\_\_\_\_\_\_，中序遍历序列为\_\_\_\_\_\_\_\_\_\_\_，后序遍历序列为\_\_\_\_\_\_\_\_\_\_\_。

ABDECF  
DBEAFC  
DEBFCA

\_\_\_\_\_\_\_\_\_\_遍历二叉排序树中的结点可以得到一个递增的关键字序列（填先序、中序或后序）。

中序

设有n个无序的记录关键字，则直接插入排序的时间复杂度为\_\_\_\_\_\_\_\_，快速排序的平均时间复杂度为\_\_\_\_\_\_\_\_\_。（填类似O(log2n)、O(n)、O(nlogn)、O(n\*n)）

O(n)  
O(nlogn)

设某无向图中顶点数和边数分别为n和e，所有顶点的度数之和为d，则e=\_\_\_\_\_\_\_。

d/2

不论是顺序存储结构的栈还是链式存储结构的栈，其入栈和出栈操作的时间复杂度均为\_\_\_\_\_\_\_\_\_\_\_\_。

O(1)

设一棵完全二叉树有128个结点，则该完全二叉树的深度为\_\_\_\_\_\_\_\_，有\_\_\_\_\_\_\_\_\_\_个叶子结点。

8  
64

在一个无向图中，所有顶点的度数之和等于所有边数（ ）倍，在一个有向图中，所有顶点的入度之和等于所有顶点出度之和的（ ）倍。

2

1

下列说法正确的有：（ ）

根据任意一种遍历序列即可唯一确定对应的二叉树。

满二叉树也是完全二叉树。

树的后序遍历与其对应的二叉树的后序遍历序列相同

依据所有数据成员之间逻辑关系的不同，数据结构分为（ ）

线性结构

非线性结构

在所有的排序方法中，（ ）排序方法是不稳定的。

堆

直接选择

快速

下列说法正确的有：（ ）

数据的逻辑结构是指各数据元素之间的逻辑关系，是用户按使用需要建立的。

线性结构只能用顺序结构来存放，非线性结构只能用非顺序结构来存放。

数据结构是数据对象与对象中数据元素之间关系的集合。

算法就是程序。

下列说法正确的有：（ ）

在散列法中，采取闭散列法来解决冲突时，一般不要立刻做物理删除，否则在搜索时会发生错误。

理想情况下，在散列表中搜索一个元素的时间复杂度为Ｏ（１）。

集合是（ ）。

非线性的

无序的

适合元素个数n很大的排序方法是（ ）。

归并

堆

快速

线性表的顺序存储结构是一种( )的存储结构

顺序存取

随机存取

在时间复杂度为O(nlogn/log2)的排序方法中，（ ）排序方法是稳定的。

折半插入

归并

下列说法正确的有：（ ）

对于两棵具有相同关键码集合而形状不同的二叉搜索树，按中序遍历它们得到的序列的各元素的顺序是一样的

在二叉搜索树上插入新结点时，不必移动其它结点，仅需要改动某个结点的指针，使它由空变为非空即可。

算法的五大特性是（ ）、输入和输出

有限性

确定性

下列说法正确的有：（ ）

数据的逻辑结构与数据元素本身的内容和形式无关。

从逻辑关系上讲，数据结构分为两大类：线性结构和非线性结构。

所谓数据的逻辑结构是指数据元素之间的逻辑关系。

算法的性能标准是（ ）、高效率和低存储量。

正确性健壮性可读性

在所有的排序方法中，（ ）排序方法是稳定的。

起泡

直接插入

折半插入

归并

设一维数组中有n个数组元素，则读取第i个数组元素的平均时间复杂度为（ ）。

O(1)

用链接方式存储的队列，在进行插入运算时（ ）。

头、尾指针可能都要修改

栈和队列的共同特点是（ ）。

只允许在端点处插入和删除元素

设顺序循环队列Q[0：M-1]的头指针和尾指针分别为F和R，头指针F总是指向队头元素的前一位置，尾指针R总是指向队尾元素的当前位置，则该循环队列中的元素个数为（ ）。

(R-F+M)％M

若有18个元素的有序表存放在一维数组A[19]中，第一个元素放A[1]中，现进行二分查找，则查找A［3］的比较序列的下标依次为（ ）。

9，4，2，3

设某有向图中有n个顶点，则该有向图对应的邻接表中有（ ）个表头结点。

N

设一组初始关键字记录关键字为(20，15，14，18，21，36，40，10)，则以20为基准记录的一趟快速排序结束后的结果为（ ）。

10，15，14，18，20，36，40，21

下面关于线性表的叙述错误的是（ ）。

线性表采用顺序存储便于插入和删除操作的实现

设有5000个待排序的记录关键字，如果需要用最快的方法选出其中最小的10个记录关键字，则用下列（ ）方法可以达到此目的。

堆排序

设无向图G中有n个顶点e条边，则其对应的邻接表中的表头结点和表结点的个数分别为（ ）。

n，2e

设一组初始记录关键字序列(5，2，6，3，8)，以第一个记录关键字5为基准进行一趟快速排序的结果为（ ）

3，2，5，6，8

设某数据结构的二元组形式表示为A=(D，R)，D={01，02，03，04，05，06，07，08， 09}，R={r}，r={<01，02>，<01，03>，<01，04>，<02，05>，<02，06>，<03，07>，<03，08>，<03，09>}，则数据结构A是（ ）。

树型结构

设哈夫曼树中的叶子结点总数为m，若用二叉链表作为存储结构，则该哈夫曼树中总共有（ ）个空指针域。

2m

设有序表中有1000个元素，则用二分查找查找元素X最多需要比较（ ）次。

10

设某棵二叉树的中序遍历序列为ABCD，前序遍历序列为CABD，则后序遍历该二叉树得到序列为（ ）。

BADC

设一棵二叉树的深度为k，则该二叉树中最多有（ ）个结点。

(2^k)-1

设一组初始记录关键字序列为(345，253，674，924，627)，则用基数排序需要进行（ ）趟的分配和回收才能使得初始关键字序列变成有序序列。

3

设有n个待排序的记录关键字，则在堆排序中需要（ ）个辅助记录单元。

1

设某二叉树中度数为0的结点数为N0，度数为1的结点数为Nl，度数为2的结点数为N2，

则下列等式成立的是（ ）。

N0=N2+1

数据的最小单位是（ ）。

数据项

设用链表作为栈的存储结构则退栈操作（ ）。

必须判别栈是否为空

树最适合用来表示（ ）。

元素之间具有分支层次关系的数据

设某无向图中有n个顶点e条边，则该无向图中所有顶点的入度之和为（ ）。

2e

设输入序列是1、2、3、??、n，经过栈的作用后输出序列的第一个元素是n，则输出序列中第i个输出元素是（ ）。

n+1-i

设连通图G中的边集E={(a，b)，(a，e)，(a，c)，(b，e)，(e，d)，(d，f)，(f，c)}，则从顶点a出发可以得到一种深度优先遍历的顶点序列为（ ）。

Acfebd

二叉树的第k层的结点数最多为（ ）。

2^（k-1）

设某完全无向图中有n个顶点，则该完全无向图中有（ ）条边。

n(n-1)/2

设某棵二叉树中有2000个结点，则该二叉树的最小高度为（ ）。

11

设一组初始记录关键字序列为(50，40，95，20，15，70，60，45)，则以增量d=4的一趟希尔排序结束后前4条记录关键字为（ ）。

15，40，60，20

对于线性表（7，34，55，25，64，46，20，10）进行散列存储时，若选用H（K）=K %9作为散列函数，则散列地址为1的元素有（ ）个。

4

设一组初始记录关键字序列为(25，50，15，35，80，85，20，40，36，70)，其中含有5个长度为2的有序子表，则用归并排序的方法对该记录关键字序列进行一趟归并后的结果为（ ）。

15，25，35，50，20，40，80，85，36，70

设有一个二维数组A[m][n]，假设A[0][0]存放位置在644(10)，A[2][2]存放位置在676(10)，每个元素占一个空间，问A[3][3](10)存放在什么位置？脚注(10)表示用10进制表示。

692

下列四种排序中（ ）的空间复杂度最大。

快速排序

以下数据结构中哪一个是非线性结构？（ ）

二叉树

设有6个结点的无向图，该图至少应有（ ）条边才能确保是一个连通图。

5

下列四种排序中（ ）的空间复杂度最大。

归并排序

设某强连通图中有n个顶点，则该强连通图中至少有（ ）条边。

N

对n个记录的文件进行快速排序，所需要的辅助存储空间大致为（ ）。

O（1og2n）

设指针变量p指向单链表中结点A，若删除单链表中结点A，则需要修改指针的操作序列为（ ）。

q=p->next；p->data=q->data；p->next=q->next；free(q)；

请实现归并排序的一次归并。将相邻的两个有序子序列R[low..mid-1]和R[mid..high]，归并后形成更长的有序序列R[low..high]。  
快速排序的一次划分的具体做法是：设置两个指针low、high分别用来指示将要与枢轴进行比较的左侧记录和右侧记录，首先从high所指位置开始向前查找关键字小于枢轴关键字的记录，将其与枢轴交换，再从low所指位置开始向后查找关键字大于枢轴关键字的记录，将其与枢轴交换，反复执行以上两步，直到low与high相等。在这个过程中，记录交换都是与枢轴之间发生，每次交换要移动3次记录，我们可以先用临时变量暂存枢轴，只移动要与枢轴交换的记录，直到最后再将临时变量保存的枢轴放入最终位置，这种做法可减少排序中的记录移动次数。  
  
注意:请按照如下示例格式书写答案  
template <class Type>  
void merge(Type R[],Type tmp[],int low, int mid, int high){   
  
//函数体  
  
}

template <class Type>  
void merge(Type R[],Type tmp[],int low, int mid, int high){   
int i=low, j=mid, k=0;  
while (i<mid && j <= high) // R中记录由小到大地并入tmp  
if (R[i] < R[j]) tmp[k++] = R[i++]; // 将R[i]和R[j]的小者拷贝到tmp[k]  
else tmp[k++] = R[j++];   
while ( i<mid ) tmp[k++] = R[i++]; // 将剩余的R[i.. mid-1]复制到tmp  
while ( j<=high ) tmp[k++] = R[j++]; // 将剩余的R[j..high]复制到tmp  
for (i=0, k = low; k<=high; )   
R[k++] = tmp[i++]; // 排好序的记录由tmp拷回R  
}

请实现快速排序的一次划分,假设取最左侧元素做枢轴。  
快速排序的一次划分的具体做法是：设置两个指针low、high分别用来指示将要与枢轴进行比较的左侧记录和右侧记录，首先从high所指位置开始向前查找关键字小于枢轴关键字的记录，将其与枢轴交换，再从low所指位置开始向后查找关键字大于枢轴关键字的记录，将其与枢轴交换，反复执行以上两步，直到low与high相等。在这个过程中，记录交换都是与枢轴之间发生，每次交换要移动3次记录，我们可以先用临时变量暂存枢轴，只移动要与枢轴交换的记录，直到最后再将临时变量保存的枢轴放入最终位置，这种做法可减少排序中的记录移动次数。  
  
注意:请按照如下示例格式书写答案  
template <class Type>  
int partition( Type S[], int low, int high){   
//参数为待排序序列S，以及排序区间的下界low和上界high。  
  
//函数体  
  
}

一趟快速排序（或一次划分）。参数为待排序序列S，以及排序区间的下界low和上界high。  
template <class Type>  
int partition( Type S[], int low, int high){   
Type tmp = S[low]; // 暂存轴值  
while(low != high){ // 开始进行分割  
while (low<high && S[high]>=tmp)high--; // 大下标端找 < 轴值的记录  
if (low < high) { S[low] = S[high]; low++;} // 该记录移动到小下标端  
while (low < high && S[low] <=tmp) low++; // 小下标端找 > 轴值的记录  
if (low < high) { S[high] = S[low]; high--;} // 该记录移动到大下标端  
}   
S[low] = tmp; // 把轴值回填到分界位置上  
return low; // 返回枢轴位置  
}

请实现直接选择排序算法。  
直接选择排序的基本思想是：  
对于由size个记录组成的记录序列，第一趟，从size个记录中选取关键字值最小的记录与第一个记录互换；第二趟，从剩余的size -1个记录中选取关键字值最小的记录与第二个记录互换；一般地，第i趟，从剩余的size -i+1个记录中选取关键字值最小的记录与第i个记录互换。重复以上过程，直到剩余记录仅有一个为止。  
注意:请按照如下示例格式书写答案  
template <class Type>  
void straightSelectSort(Type R[], int size){   
  
//函数体  
  
}

template <class Type>  
void straightSelectSort(Type R[], int size){   
int pos, min ,j; // min为一趟排序中最小记录下标  
for (pos = 0; pos < size -1; pos++) { // pos为待存放当前最小记录的位置  
min = pos;  
for (j = pos+1; j < size; ++j)  
if (R[j] < R[min]) min = j; // 查找最小记录   
if(pos != min) swap(R[pos],R[min]); // 调用STL中的swap，头文件algorithm  
}  
}

请实现冒泡排序算法。  
冒泡排序算法基本思想是对所有相邻记录的关键字值进行比较，如果不满足排序要求（即逆序），则将其交换，最终直到所有记录排好序为止。若在某一趟排序中没有发生交换操作，说明待排序记录已全部有序，排序提前结束。  
  
注意:请按照如下示例格式书写答案  
template <class Type>  
void bubbleSort(Type R[], int size) {  
  
//函数体  
  
}

template <class Type>  
void bubbleSort(Type R[], int size) {  
int i, j;  
bool flag = true; // 记录一趟排序后是否发生过交换  
for (i = 1; i < size && flag; ++i) {  
flag = false; // 假定本趟排序没有交换  
for (j = 0; j < size-i; ++j)  
if (R[j+1] < R[j]){ // 逆序   
swap(R[j],R[j+1]); // 调用STL中的swap进行交换  
flag = true;  
}  
}  
}

请实现希尔排序算法。  
对于有size个记录的初始序列，希尔排序的具体步骤如下：  
(1) 首先取一个整数gap< size作为增量；  
(2) 将全部记录分为gap个子序列，所有间距为gap的记录分在同一个子序列中，对每个子序列分别实施直接插入排序；  
(3) 然后缩小增量gap，重复步骤(2)的子序列划分和排序工作，直到最后gap等于1，将所有记录放在一组，进行最后一次直接插入排序。  
Shell提出从gap= size /2开始划分子序列，每次缩小增量gap=gap/2，直到gap=1为止。  
  
注意:请按照如下示例格式书写答案  
template <class Type>  
void shellSort(Type R[], int size) {  
  
//函数体  
  
}

emplate <class Type>  
void shellSort(Type R[], int size) {  
int gap, pos, j;   
Type tmp;  
for (gap = size/2; gap > 0; gap /= 2){ // gap为希尔增量，即步长  
for (pos = gap; pos < size; pos++) { // pos为待插入记录位置  
tmp = R[pos];  
for (j = pos - gap; j >= 0 && R[j] > tmp; j -= gap)   
R[j+gap] = R[j]; // 记录后移   
R[j+gap] = tmp; // 将待插入记录放到合适位置  
}  
}

请实现直接插入排序算法。  
其算法思想为：记录序列为R[0..size-1]，首先将第一个记录R[0]看做一个有序子序列，然后依次将记录R[i]（1≤i≤size-1）插入到有序子序列R[0..i-1]中，使记录的有序子序列从R[0..i-1]变为R[0..i]。  
  
注意:请按照如下示例格式书写答案  
template <class Type>  
void straightInsertSort(Type R[], int size){  
  
//函数体  
  
}

template <class Type>  
void straightInsertSort(Type R[], int size){  
int pos,j; // pos为待插入记录位置   
Type tmp;  
for (pos = 1; pos < size; pos++) {  
tmp = R[pos]; // 将待插入记录放进临时变量  
for (j = pos-1; tmp < R[j] && j >= 0; j--) // 从后向前查找插入位置  
R[j+1] = R[j]; // 将大于待插入记录的记录向后移动  
R[j+1] = tmp; // 将待插入记录放到合适位置  
}  
}

请实现非递归的折半查找算法，在有序表set[1..n]中查找关键字为k的元素，若查找成功返回关键字为k的元素的下标，查找失败返回0.  
  
注意:请按照如下示例格式书写答案  
template <class RecType>  
int binarySearch(const vector<RecType> &set, const RecType &k) {  
  
//函数体  
  
}

template <class RecType>  
int binarySearch(const vector<RecType> &set, const RecType &k) {  
int low = 1, high = set.size() - 1, mid;  
while (low <= high ) { // 查找范围不为空  
mid = (low + high) / 2; // 计算中间位置  
if ( k == set[mid] ) return mid; // 查找成功  
if ( k < set[mid]) high = mid - 1; // 继续在前半区查找，修改high  
else low = mid + 1; // 继续在后半区查找，修改low  
}  
return 0; // 查找失败  
}

已知图采用邻接表存储表示，  
请实现图的广度优先遍历算法void bfsTraverse() const;，从任意顶点出发广度优先遍历图并输出广度优先遍历序列。  
  
图的邻接表表示法的定义如下:  
template <class VertexType, class EdgeType>  
class adjList {  
private:  
struct edgeNode { // 边表结点类型  
int to; // 边的终点编号（在顶边表中的下标）  
EdgeType weight; // 边上的权值  
edgeNode \*next; // 指向下一个边表结点  
edgeNode(){ } // 无参构造函数  
edgeNode(int t, EdgeType w, edgeNode \*n = NULL){  
to = t; weight = w; next = n;  
}  
};  
struct verNode{ // 顶点结点类型  
VertexType vertex; // 顶点信息  
edgeNode \*firstEdge; // 指向第一个邻接点的指针  
verNode(edgeNode \*h = NULL) { firstEdge = h; }  
};   
verNode \*verList; // 顶点表  
int verNum, edgeNum; // 图的顶点数和边数  
bool \*visited; // 访问标志数组  
void dfs(int start) const; // 从start号顶点出发深度优先遍历图  
……  
public:  
adjList(int size);  
~adjList();  
void dfsTraverse() const; // 调用私有dfs深度优先遍历图  
void bfsTraverse() const; // 广度优先遍历图  
bool topSort() const; // 拓扑排序  
……  
};

template <class VertexType, class EdgeType>  
void adjList<VertexType, EdgeType>::bfsTraverse()const{  
int v,i,count=0; // count可统计连通分量个数  
queue<int> q;  
edgeNode \*p;  
for (i = 0; i < verNum; i++) visited[i] = false; // 置访问标志为false  
for (i = 0; i < verNum; i++) {  
if (visited[i] == true) continue;  
cout << verList[i].vertex << ' '; // 访问顶点i  
visited[i] = true; // 置访问标志为true  
q.push(i); count++; // 顶点i入队  
while (!q.empty()) {  
v = q.front(); q.pop(); // 顶点v出队  
p = verList[v].firstEdge;  
while (p != NULL){ // 查找顶点v未被访问的临接点  
if (visited[p->to] == false){   
cout << verList[p->to].vertex << ' ';// 访问顶点p->to  
visited[p->to] = true; // 置访问标志为true  
q.push(p->to); // 顶点p->to入队  
}  
p = p->next;  
}  
}  
}  
cout<< endl;  
}

已知图采用邻接矩阵存储表示，  
请实现图的广度优先遍历算法void bfsTraverse() const;，从任意顶点出发广度优先遍历图并输出广度优先遍历序列。  
  
图的邻接矩阵表示法的定义如下:  
template <class VertexType, class EdgeType> // 顶点的数据类型和边（或弧）的类型  
class adjMatrix {  
private:  
VertexType \*vertexs; // 存放顶点信息  
EdgeType \*\*edges; // 邻接矩阵  
EdgeType noEdge; // 无边标志  
int verNum, edgeNum; // 图的顶点数和边数  
bool \*visited; // 访问标志数组  
void dfs(int start) const; // 从start顶点出发深度优先遍历图  
……  
public:  
adjMatrix(int size,EdgeType noEdgeFlag);  
~adjMatrix();  
void dfsTraverse() const; // 调用私有dfs深度优先遍历图  
void bfsTraverse() const; // 广度优先遍历图  
bool topSort() const; // 拓扑排序  
……  
};

template <class VertexType, class EdgeType>  
void adjMatrix<VertexType, EdgeType>::bfsTraverse()const{  
int v,i,j,count=0; // count可统计连通分量个数  
queue<int> Q;  
for (i = 0; i < verNum; i++) visited[i] = false; // 置访问标志为false  
for (i = 0; i < verNum; i++) {  
if (visited[i] == true) continue;  
cout << vertexs[i]<< ' '; // 访问顶点i  
visited[i] = true; // 置访问标志为true  
Q.push(i); count++;  
while (!Q.empty()) {  
v = Q.front(); Q.pop(); // 顶点v出队  
if (visited[v] == false) {  
for (j = 0; j < verNum; ++j) { // 查找顶点v未被访问的临接点  
if (edges[v][j] != noEdge && visited[j] == false){  
cout << vertexs[j]<< ' '; // 访问顶点j  
visited[j] = true; // 置访问标志为true  
Q.push(j);  
}  
}  
}  
}  
}  
cout <<endl;  
}

已知图采用邻接表存储表示，  
请实现图的深度优先遍历算法void dfs(int start) const;，输出从顶点start出发能够深度优先遍历到的所有顶点，输出深度优先遍历序列。  
  
图的邻接表表示法的定义如下:  
template <class VertexType, class EdgeType>  
class adjList {  
private:  
struct edgeNode { // 边表结点类型  
int to; // 边的终点编号（在顶边表中的下标）  
EdgeType weight; // 边上的权值  
edgeNode \*next; // 指向下一个边表结点  
edgeNode(){ } // 无参构造函数  
edgeNode(int t, EdgeType w, edgeNode \*n = NULL){  
to = t; weight = w; next = n;  
}  
};  
struct verNode{ // 顶点结点类型  
VertexType vertex; // 顶点信息  
edgeNode \*firstEdge; // 指向第一个邻接点的指针  
verNode(edgeNode \*h = NULL) { firstEdge = h; }  
};   
verNode \*verList; // 顶点表  
int verNum, edgeNum; // 图的顶点数和边数  
bool \*visited; // 访问标志数组  
  
void dfs(int start) const; // 从start号顶点出发深度优先遍历图  
……  
public:  
adjList(int size);  
~adjList();  
void dfsTraverse() const; // 调用私有dfs深度优先遍历图  
void bfsTraverse() const; // 广度优先遍历图  
bool topSort() const; // 拓扑排序  
……  
};

template <class VertexType, class EdgeType>  
void adjList<VertexType, EdgeType>::dfs(int start) const{  
edgeNode \*p = verList[start].firstEdge;  
cout << verList[start].vertex << ' '; // 访问顶点v  
visited[start] = true; // 置访问标志为true  
while (p != NULL){  
if (visited[p->to] == false) // 选取顶点start的未被访问的邻接点  
dfs(p->to); // 从p->to出发继续深度优先遍历图  
p = p->next;  
}  
}

已知图采用邻接矩阵存储表示，  
请实现图的深度优先遍历算法void dfs(int start) const; ，访问从顶点start出发能够深度优先遍历到的所有顶点，输出深度优先遍历序列。  
  
图的邻接矩阵表示法的定义如下:  
template <class VertexType, class EdgeType> // 顶点的数据类型和边（或弧）的类型  
class adjMatrix {  
private:  
VertexType \*vertexs; // 存放顶点信息  
EdgeType \*\*edges; // 邻接矩阵  
EdgeType noEdge; // 无边标志  
int verNum, edgeNum; // 图的顶点数和边数  
bool \*visited; // 访问标志数组  
void dfs(int start) const; // 从start顶点出发深度优先遍历图  
……  
public:  
adjMatrix(int size,EdgeType noEdgeFlag);  
~adjMatrix();  
void dfsTraverse() const; // 调用私有dfs深度优先遍历图  
void bfsTraverse() const; // 广度优先遍历图  
bool topSort() const; // 拓扑排序  
……  
};

template <class VertexType, class EdgeType>  
void adjMatrix<VertexType, EdgeType>::dfs(int start) const{  
int i;  
cout << vertexs[start] << ' '; // 访问顶点v  
visited[start] = true; // 置访问标志为true  
for (i = 0; i < verNum; i++){ // 选取顶点start的未被访问的邻接点  
if (edges[start][i] != noEdge && visited[i] == false)  
dfs(i); // 从i出发继续深度优先遍历图  
}  
}

已知二叉树采用二叉链表存储表示，  
请实现递归后序遍历二叉树的算法void postOrder( Node \*t ) const;。  
提示，求解思想为：若二叉树为空，则算法结束，否则执行：  
 后序遍历左子树  
 后序遍历右子树  
 访问根结点  
  
二叉树的二叉链表的表示如下:  
template <class elemType>   
class BinaryLinkList{  
private:  
struct Node { // 二叉链表结点  
Node \*left , \*right ; // 指向左、右孩子的指针  
elemType data; // 结点的数据域  
Node() : left(NULL), right(NULL) { } // 无参构造函数  
Node(elemType value, Node \*l = NULL, Node \* r =NULL ){  
data=value; left=l; right=r;   
}  
~Node() {}   
};  
Node \* root; // 私有，指向二叉树的根结点  
int size( Node \*t ) const; // 私有，二叉树的结点总数  
int height( Node \*t ) const; // 私有，二叉树的高度  
int leafNum(Node \*t )const; // 私有，二叉树的叶子数  
void preOrder( Node \*t ) const; // 私有，递归前序遍历  
void inOrder( Node \*t ) const; // 私有，递归中序遍历  
void postOrder( Node \*t ) const; // 私有，递归后序遍历  
……  
public:   
BinaryLinkList() : root( NULL) {} // 构造空二叉树  
~BinaryLinkList(){ clear(); }   
bool empty () const{ return root == NULL; } // 公有，判空  
void clear() {if (root) clear(root); root = NULL;} // 公有，清空  
int size() const { return size(root);} // 公有，求结点总数  
int height() const { return height(root); } // 公有，二叉树的高度  
int leafNum()const{ return leafNum(root); } // 公有，二叉树的叶子数  
void preOrderTraverse() const{ if(root) preOrder(root); } // 公有，前序遍历   
void inOrderTraverse() const { if(root) inOrder(root); } // 公有，中序遍历  
void postOrderTraverse() const{ if(root) postOrder(root);}// 公有，后序遍历  
……  
}   
};

template <class elemType>   
void BinaryLinkList<elemType>:: postOrder(Node \*t) const{   
if (t){   
postOrder(t->left);   
postOrder(t->right);   
cout <<t->data << ' ';   
}  
}

已知二叉树采用二叉链表存储表示，  
请实现递归中序遍历二叉树的算法void inOrder( Node \*t ) const;。  
提示，求解思想为：若二叉树为空，则算法结束，否则执行：  
 中序遍历左子树  
 访问根结点  
 中序遍历右子树  
  
二叉树的二叉链表的表示如下:  
template <class elemType>   
class BinaryLinkList{  
private:  
struct Node { // 二叉链表结点  
Node \*left , \*right ; // 指向左、右孩子的指针  
elemType data; // 结点的数据域  
Node() : left(NULL), right(NULL) { } // 无参构造函数  
Node(elemType value, Node \*l = NULL, Node \* r =NULL ){  
data=value; left=l; right=r;   
}  
~Node() {}   
};  
Node \* root; // 私有，指向二叉树的根结点  
int size( Node \*t ) const; // 私有，二叉树的结点总数  
int height( Node \*t ) const; // 私有，二叉树的高度  
int leafNum(Node \*t )const; // 私有，二叉树的叶子数  
void preOrder( Node \*t ) const; // 私有，递归前序遍历  
void inOrder( Node \*t ) const; // 私有，递归中序遍历  
void postOrder( Node \*t ) const; // 私有，递归后序遍历  
……  
public:   
BinaryLinkList() : root( NULL) {} // 构造空二叉树  
~BinaryLinkList(){ clear(); }   
bool empty () const{ return root == NULL; } // 公有，判空  
void clear() {if (root) clear(root); root = NULL;} // 公有，清空  
int size() const { return size(root);} // 公有，求结点总数  
int height() const { return height(root); } // 公有，二叉树的高度  
int leafNum()const{ return leafNum(root); } // 公有，二叉树的叶子数  
void preOrderTraverse() const{ if(root) preOrder(root); } // 公有，前序遍历   
void inOrderTraverse() const { if(root) inOrder(root); } // 公有，中序遍历  
void postOrderTraverse() const{ if(root) postOrder(root);}// 公有，后序遍历  
……  
}   
};

template <class elemType>   
void BinaryLinkList<elemType>:: inOrder(Node \*t) const{   
if (t){   
inOrder(t->left); // 中序遍历左子树  
cout <<t->data << ' '; // 访问当前结点  
inOrder(t->right); // 中序遍历右子树  
}  
}

已知二叉树采用二叉链表存储表示，  
请实现递归前序遍历二叉树的算法void preOrder( Node \*t ) const;。  
提示，求解思想为：若二叉树为空，则算法结束，否则执行：  
 访问根结点  
 递归前序遍历左子树  
 递归前序遍历右子树  
  
二叉树的二叉链表的表示如下:  
template <class elemType>   
class BinaryLinkList{  
private:  
struct Node { // 二叉链表结点  
Node \*left , \*right ; // 指向左、右孩子的指针  
elemType data; // 结点的数据域  
Node() : left(NULL), right(NULL) { } // 无参构造函数  
Node(elemType value, Node \*l = NULL, Node \* r =NULL ){  
data=value; left=l; right=r;   
}  
~Node() {}   
};  
Node \* root; // 私有，指向二叉树的根结点  
int size( Node \*t ) const; // 私有，二叉树的结点总数  
int height( Node \*t ) const; // 私有，二叉树的高度  
int leafNum(Node \*t )const; // 私有，二叉树的叶子数  
void preOrder( Node \*t ) const; // 私有，递归前序遍历  
void inOrder( Node \*t ) const; // 私有，递归中序遍历  
void postOrder( Node \*t ) const; // 私有，递归后序遍历  
……  
public:   
BinaryLinkList() : root( NULL) {} // 构造空二叉树  
~BinaryLinkList(){ clear(); }   
bool empty () const{ return root == NULL; } // 公有，判空  
void clear() {if (root) clear(root); root = NULL;} // 公有，清空  
int size() const { return size(root);} // 公有，求结点总数  
int height() const { return height(root); } // 公有，二叉树的高度  
int leafNum()const{ return leafNum(root); } // 公有，二叉树的叶子数  
void preOrderTraverse() const{ if(root) preOrder(root); } // 公有，前序遍历   
void inOrderTraverse() const { if(root) inOrder(root); } // 公有，中序遍历  
void postOrderTraverse() const{ if(root) postOrder(root);}// 公有，后序遍历  
……  
}   
};

template <class elemType>   
void BinaryLinkList<elemType>:: preOrder(Node \*t) const{   
if (t){   
cout <<t->data << ' '; // 访问当前结点  
preOrder(t->left); // 前序遍历左子树  
preOrder(t->right); // 前序遍历右子树  
}  
}

已知二叉树采用二叉链表存储表示，  
请实现求叶子数的算法int height( Node \*t ) const;，计算叶子结点总数。  
提示，该问题的一种求解思想为：空树叶结点数为0，若二叉树非空，对于每个结点来说：  
(1)如果它没有孩子结点，那么它就是叶结点。  
(2)如果它有孩子结点，那么它的叶结点数量 = 左子树叶结点数 + 右子树叶结点数  
  
二叉树的二叉链表的表示如下:  
template <class elemType>   
class BinaryLinkList{  
private:  
struct Node { // 二叉链表结点  
Node \*left , \*right ; // 指向左、右孩子的指针  
elemType data; // 结点的数据域  
Node() : left(NULL), right(NULL) { } // 无参构造函数  
Node(elemType value, Node \*l = NULL, Node \* r =NULL ){  
data=value; left=l; right=r;   
}  
~Node() {}   
};  
Node \* root; // 私有，指向二叉树的根结点  
int size( Node \*t ) const; // 私有，二叉树的结点总数  
int height( Node \*t ) const; // 私有，二叉树的高度  
int leafNum(Node \*t )const; // 私有，二叉树的叶子数  
void preOrder( Node \*t ) const; // 私有，递归前序遍历  
void inOrder( Node \*t ) const; // 私有，递归中序遍历  
void postOrder( Node \*t ) const; // 私有，递归后序遍历  
……  
public:   
BinaryLinkList() : root( NULL) {} // 构造空二叉树  
~BinaryLinkList(){ clear(); }   
bool empty () const{ return root == NULL; } // 公有，判空  
void clear() {if (root) clear(root); root = NULL;} // 公有，清空  
int size() const { return size(root);} // 公有，求结点总数  
int height() const { return height(root); } // 公有，二叉树的高度  
int leafNum()const{ return leafNum(root); } // 公有，二叉树的叶子数  
void preOrderTraverse() const{ if(root) preOrder(root); } // 公有，前序遍历   
void inOrderTraverse() const { if(root) inOrder(root); } // 公有，中序遍历  
void postOrderTraverse() const{ if(root) postOrder(root);}// 公有，后序遍历  
……  
}   
};

template <class elemType>   
int BinaryLinkList<elemType>::leafNum(Node\* t)const{  
if(t==NULL)return 0; // 递归出口：空树叶子数为0  
else if((t->left==NULL)&&(t->right==NULL))return 1; // 递归出口：是叶子结点  
else return leafNum(t->left)+leafNum(t->right); // 统计左、右子树叶子数总和  
}  
也有其他答案，不过这几个关键字不变

已知二叉树采用二叉链表存储表示，  
请实现求高度的算法int height( Node \*t ) const;，计算二叉树的高度。  
提示，该问题的一种求解思想为：空树高度为0；若二叉树非空，它的高度应该为，左右子树高度大者+1  
  
二叉树的二叉链表的表示如下:  
template <class elemType>   
class BinaryLinkList{  
private:  
struct Node { // 二叉链表结点  
Node \*left , \*right ; // 指向左、右孩子的指针  
elemType data; // 结点的数据域  
Node() : left(NULL), right(NULL) { } // 无参构造函数  
Node(elemType value, Node \*l = NULL, Node \* r =NULL ){  
data=value; left=l; right=r;   
}  
~Node() {}   
};  
Node \* root; // 私有，指向二叉树的根结点  
int size( Node \*t ) const; // 私有，二叉树的结点总数  
int height( Node \*t ) const; // 私有，二叉树的高度  
int leafNum(Node \*t )const; // 私有，二叉树的叶子数  
void preOrder( Node \*t ) const; // 私有，递归前序遍历  
void inOrder( Node \*t ) const; // 私有，递归中序遍历  
void postOrder( Node \*t ) const; // 私有，递归后序遍历  
……  
public:   
BinaryLinkList() : root( NULL) {} // 构造空二叉树  
~BinaryLinkList(){ clear(); }   
bool empty () const{ return root == NULL; } // 公有，判空  
void clear() {if (root) clear(root); root = NULL;} // 公有，清空  
int size() const { return size(root);} // 公有，求结点总数  
int height() const { return height(root); } // 公有，二叉树的高度  
int leafNum()const{ return leafNum(root); } // 公有，二叉树的叶子数  
void preOrderTraverse() const{ if(root) preOrder(root); } // 公有，前序遍历   
void inOrderTraverse() const { if(root) inOrder(root); } // 公有，中序遍历  
void postOrderTraverse() const{ if(root) postOrder(root);}// 公有，后序遍历  
……  
}   
};

emplate <class elemType>   
int BinaryLinkList<elemType>::height(Node \*t) const{   
if (t == NULL) return 0; // 递归出口：空树高度为0   
else {  
int lh = height(t->left), rh = height(t->right);  
return 1 + ( (lh > rh) ? lh : rh); // 树的高度为左右子树高度大者+1  
}  
}

已知二叉树采用二叉链表存储表示，  
请实现求结点数算法int size( Node \*t ) const，计算二叉树中结点总数。  
提示，该问题的一种求解思想为：空树结点数为0；若二叉树非空，它的结点总数应为：左子树的结点数 + 右子树的结点数 + 根结点（1个）  
  
二叉树的二叉链表的表示如下:  
template <class elemType>   
class BinaryLinkList{  
private:  
struct Node { // 二叉链表结点  
Node \*left , \*right ; // 指向左、右孩子的指针  
elemType data; // 结点的数据域  
Node() : left(NULL), right(NULL) { } // 无参构造函数  
Node(elemType value, Node \*l = NULL, Node \* r =NULL ){  
data=value; left=l; right=r;   
}  
~Node() {}   
};  
Node \* root; // 私有，指向二叉树的根结点  
int size( Node \*t ) const; // 私有，二叉树的结点总数  
int height( Node \*t ) const; // 私有，二叉树的高度  
int leafNum(Node \*t )const; // 私有，二叉树的叶子数  
void preOrder( Node \*t ) const; // 私有，递归前序遍历  
void inOrder( Node \*t ) const; // 私有，递归中序遍历  
void postOrder( Node \*t ) const; // 私有，递归后序遍历  
……  
public:   
BinaryLinkList() : root( NULL) {} // 构造空二叉树  
~BinaryLinkList(){ clear(); }   
bool empty () const{ return root == NULL; } // 公有，判空  
void clear() {if (root) clear(root); root = NULL;} // 公有，清空  
int size() const { return size(root);} // 公有，求结点总数  
int height() const { return height(root); } // 公有，二叉树的高度  
int leafNum()const{ return leafNum(root); } // 公有，二叉树的叶子数  
void preOrderTraverse() const{ if(root) preOrder(root); } // 公有，前序遍历   
void inOrderTraverse() const { if(root) inOrder(root); } // 公有，中序遍历  
void postOrderTraverse() const{ if(root) postOrder(root);}// 公有，后序遍历  
……  
}   
};

template <class elemType>   
int BinaryLinkList<elemType>::size(Node \*t) const{   
if (t == NULL) return 0; // 递归出口：空树结点数为0  
return 1 + size(t->left) + size(t->right); // 树的结点数为左右子树结点数+根结点  
}  
也有其他答案，不过这几个关键字不变

已知链栈的类型定义如下，  
请实现进栈算法void push(const T &value);，将值为value的元素压入栈顶。  
  
template <class T>  
class seqStack {   
private:  
T \* data; // 存放栈中元素的数组  
int top; // 栈顶指针，指向栈顶元素  
int maxSize; // 栈的大小  
void resize(); // 扩大栈空间  
public:  
seqStack(int initSize = 100);   
~seqStack(){ delete [] data;}  
void clear() { top = -1; } // 清空栈   
bool empty() const{ return top == -1;}// 判空  
int size() const{ return top+1; } // 求长度  
void push(const T &value); // 进栈  
T pop(); // 出栈  
T getTop() const; // 取栈顶元素  
};

template <class T>  
void linkStack<T>::push(const T &value) {  
Node \*p = new Node(value, top); // 在栈顶插入元素  
top = p; // p成为新的栈顶元素  
}

已知单链表的类型定义如下，  
请实现算法int size()const;求单链表的长度。注意：请将工作指针命名为p，计数器命名为count。  
  
单链表的类型定义：  
template <class elemType> // elemType为单链表存储的元素类型  
class linkList {   
private:   
struct Node { // 结点类型  
public:   
elemType data; // 结点的数据域   
Node \* next; // 结点的指针域  
Node(const elemType value, Node\* p = NULL){ // 两个参数的构造函数   
data = value; next = p;   
}   
Node(Node\* p = NULL) { // 一个参数的构造函数   
next = p;   
}   
};  
Node\* head; // 单链表的头指针   
public:   
linkList(); // 构造函数   
~linkList(); // 析构函数   
int size()const; // 返回单链表的当前实际长度   
……  
};

template <class elemType>   
int linkList<elemType>::size()const{ // 若类中没有数据成员curLength  
Node \*p =head->next; // 需要从头到尾遍历链表   
int count=0;  
while(p){ count++; p=p->next;}  
return count;  
}

已知单链表的类型定义如下，  
请实现查找算法int search(const elemType&value)const; 查找值为value的元素第一次出现时的位序（若表长为n，则位序范围为0~n-1）。注意：请将工作指针命名为p。  
  
单链表的类型定义：  
template <class elemType> // elemType为单链表存储的元素类型  
class linkList {   
private:   
struct Node { // 结点类型  
public:   
elemType data; // 结点的数据域   
Node \* next; // 结点的指针域  
Node(const elemType value, Node\* p = NULL){ // 两个参数的构造函数   
data = value; next = p;   
}   
Node(Node\* p = NULL) { // 一个参数的构造函数   
next = p;   
}   
};  
Node\* head; // 单链表的头指针   
public:   
linkList(); // 构造函数   
~linkList(); // 析构函数   
int search(const elemType&value)const; // 查找值为value的元素第一次出现的位序  
……  
};

template <class elemType>   
int linkList<elemType> ::search(const elemType&value)const{   
Node \*p = head->next; // 工作指针p指向首元结点  
int count=0; // 首元结点的位序为0  
while (p != NULL && p->data!=value) {   
p = p->next;   
count++;  
}   
if(p == NULL)return -1; // 查找失败返回-1，这里-1并非头结点  
else return count; // 查找成功，count为元素的位序  
}

已知顺序表的类型定义如下，  
请实现逆置算法void inverse();，调整线性表中元素顺序，使L由 (data0，data1，......， datan-1)变为(datan-1，... ，data1，data0)。注意：请将用于遍历顺序表的循环控制变量命名为i。  
  
顺序表的类型定义：  
template <class elemType> // elemType为顺序表存储的元素类型  
class seqList{   
private:  
elemType \*data; // 利用动态数组存储数据元素   
int curLength; // 当前顺序表中存储的元素个数  
int maxSize; // 顺序表的最大长度  
void resize(); // 表满时扩大表空间  
public:  
seqList(int initSize = 10); // 构造函数   
seqList(seqList & sl) ; // 拷贝构造函数  
~seqList(){ delete [] data; } // 析构函数   
void clear(){ curLength = 0; } // 清空表，只需置curLength为0  
bool empty()const{return curLength==0;} // 判空  
int size()const{ return curLength; } // 返回顺序表中当前存储元素的个数   
void inverse(); // 逆置顺序表  
……  
};

template<class elemType>  
void seqList<elemType>::inverse()   
{  
elemType tmp;  
for (int i = 0; i < curLength/2; i++){ // 控制交换的次数  
tmp = data[i];  
data[i] = data[curLength-i-1];  
data[curLength-i-1] = tmp;  
}  
}

已知顺序表的类型定义如下，  
请实现查找算法int search(const elemType &value) const ;，查找值为value的元素第一次出现时的下标（位序）。注意：请将用于遍历顺序表的循环控制变量命名为i。  
  
顺序表的类型定义：  
template <class elemType> // elemType为顺序表存储的元素类型  
class seqList{   
private:  
elemType \*data; // 利用动态数组存储数据元素   
int curLength; // 当前顺序表中存储的元素个数  
int maxSize; // 顺序表的最大长度  
void resize(); // 表满时扩大表空间  
public:  
seqList(int initSize = 10); // 构造函数   
seqList(seqList & sl) ; // 拷贝构造函数  
~seqList(){ delete [] data; } // 析构函数   
void clear(){ curLength = 0; } // 清空表，只需置curLength为0  
bool empty()const{return curLength==0;} // 判空  
int size()const{ return curLength; } // 返回顺序表中当前存储元素的个数   
int search(const elemType &value) const ; // 查找值为value的元素第一次出现的位序  
……  
};

template<class elemType>  
int seqList<elemType>::search(const elemType & value) const  
{  
for (int i = 0; i < curLength; i++)  
if (value == data[i]) return i;  
return -1; // 查找失败返回-1  
}

已知顺序表的类型定义如下，  
请实现删除算法void remove(int i)，删除下标（位序）为i的元素。  
  
顺序表的类型定义：  
template <class elemType> // elemType为顺序表存储的元素类型  
class seqList{   
private:  
elemType \*data; // 利用动态数组存储数据元素   
int curLength; // 当前顺序表中存储的元素个数  
int maxSize; // 顺序表的最大长度  
void resize(); // 表满时扩大表空间  
public:  
seqList(int initSize = 10); // 构造函数   
seqList(seqList & sl) ; // 拷贝构造函数  
~seqList(){ delete [] data; } // 析构函数   
void clear(){ curLength = 0; } // 清空表，只需置curLength为0  
bool empty()const{return curLength==0;} // 判空  
int size()const{ return curLength; } // 返回顺序表中当前存储元素的个数   
void remove(int i); // 删除位序i上的元素，表的长度减1   
……  
};

template <class elemType>  
void seqList<elemType>::remove(int i)   
{   
if (i < 0 || i > curLength-1) throw outOfRange(); // 合法的删除位置为[0..n-1]   
for (int j = i; j < curLength - 1; j++) // 下标j+1在[i+1..n-1]范围  
data[j] = data[j+1] ; // 的元素往前移动一步  
--curLength; // 表的实际长度减1  
}

已知顺序表的类型定义如下，  
请实现插入算法void insert(int i,const elemType &value)，在下标（位序）为i的位置插入值为value的元素。注意若线性表已满，需要扩大表空间。  
  
顺序表的类型定义：  
template <class elemType> // elemType为顺序表存储的元素类型  
class seqList{   
private:  
elemType \*data; // 利用动态数组存储数据元素   
int curLength; // 当前顺序表中存储的元素个数  
int maxSize; // 顺序表的最大长度  
void resize(); // 表满时扩大表空间  
public:  
seqList(int initSize = 10); // 构造函数   
seqList(seqList & sl) ; // 拷贝构造函数  
~seqList(){ delete [] data; } // 析构函数   
void clear(){ curLength = 0; } // 清空表，只需置curLength为0  
bool empty()const{return curLength==0;} // 判空  
int size()const{ return curLength; } // 返回顺序表中当前存储元素的个数   
void insert(int i,const elemType &value); // 在位序i上插入值为value的元素，表的长度增1  
……  
};

template <class elemType>  
void seqList<elemType>::insert(int i, const elemType &value)  
{   
if (i < 0 || i > curLength) throw outOfRange(); // 合法的插入位置为[0..n]  
if (curLength == maxSize) resize(); // 表满，扩大数组容量  
for (int j = curLength; j > i; j--) // 下标j-1在[curLength-1..i]范围  
data[j] = data[j-1]; // 的元素往后移动一步  
data[i] = value; // 将 value 置入位序为i的位置  
++curLength; // 表的实际长度增1  
}

已知顺序实现的字符串的类型定义如下，  
请实现朴素的模式匹配算法int bfFind(const String &s, int pos = 0) const。当匹配成功时，返回子串在主串中第一次出现的位置；当匹配失败时，返回-1；当子串是空串时返回0。注意：请使用整型变量i访问主串，整型变量j访问子串。  
  
class String{  
public:  
String(const char \*str = NULL); // 构造函数  
String(const String &str); // 拷贝构造函数  
~String(){ delete []data;} // 析构函数  
int capacity()const{return maxSize;} // 最大存储容量  
int size()const{return curLength;} // 求字符串长度  
bool empty()const{return curLength==0;}// 判空  
int bfFind(const String &s, int pos = 0) const;// 朴素模式匹配(BF)  
private:  
char \*data; // 存储字符串  
int maxSize; // 最大存储容量  
int curLength; // 串的长度  
void resize(int len); // 扩大数组空间  
};

int String::bfFind(const String &s, int pos) const{  
int i = 0, j = 0; // 主串和子串的指针  
if (curLength < s.curLength) return -1; // 主串比子串短，匹配失败   
while (i < curLength && j < s.curLength){   
if (data[i] == s.data[j]) // 对应字符相等指针后移  
i++, j++;   
else { // 对应字符不相等时  
i = i - j + 1; // 主串指针回溯  
j = 0; } // 子串从头开始  
}   
if (j >= s.curLength) return (i - s.curLength); // 返回子串在主串中的位置  
else return -1;   
}

已知循环队列的类型定义如下，  
请实现出队算法T deQueue();注意处理队列为空的情况。  
  
template <class T>  
class seqQueue{  
private:  
T \*data; // 指向存放元素的数组  
int maxSize; // 队列的大小  
int front, rear; // 队头和队尾指针  
void resize(); // 扩大队列空间  
public:  
seqQueue(int initsize = 100);  
~seqQueue(){ delete [] data; }  
void clear(){ front = rear = -1; } // 清空队列  
bool empty()const { return front == rear; } // 判空  
bool full()const { return (rear + 1) % MaxSize == front; }// 判满  
int size()const{ return (rear-front+maxSize)%maxSize; } // 队列长度  
void enQueue(const T &x); // 入队  
T deQueue(); // 出队  
T getHead()const; // 取队首元素  
};

template <class T>  
T seqQueue<T>::deQueue(){  
// 若队列空，抛出异常  
if (empty())throw outOfRange();   
front = (front + 1) % maxSize; // 移动队首指针  
return data[front]; // 返回队首元素  
}

已知循环队列的类型定义如下，  
请实现入队算法void enQueue(const T &x)，注意队满时扩大队列空间。  
template <class T>  
class seqQueue{  
private:  
T \*data; // 指向存放元素的数组  
int maxSize; // 队列的大小  
int front, rear; // 队头和队尾指针  
void resize(); // 扩大队列空间  
public:  
seqQueue(int initsize = 100);  
~seqQueue(){ delete [] data; }  
void clear(){ front = rear = -1; } // 清空队列  
bool empty()const { return front == rear; } // 判空  
bool full()const { return (rear + 1) % MaxSize == front; }// 判满  
int size()const{ return (rear-front+maxSize)%maxSize; } // 队列长度  
void enQueue(const T &x); // 入队  
T deQueue(); // 出队  
T getHead()const; // 取队首元素  
};

template <class T>  
void seqQueue<T>::enQueue(const T &x){  
// 若队列满，扩大队列  
if ((rear + 1) % maxSize == front) resize();  
rear = (rear + 1) % maxSize; // 移动队尾指针  
data[rear] = x; // x入队  
}

已知顺序栈的类型定义如下，  
请实现出栈算法T pop()，注意处理栈为空的情况。工作指针请命名为p。  
  
template <class T>  
class linkStack {  
private:  
struct Node{  
T data;  
Node\* next;  
Node(){ next = NULL; }  
Node(const T &value, Node \*p = NULL){ data = value; next = p;}  
};   
Node\* top; // 栈顶指针  
public:  
linkStack(){ top = NULL; } // 构造空栈  
~linkStack(){ clear(); }  
void clear(); // 清空  
bool empty()const{ return top == NULL; } // 判空  
int size()const; // 求长度  
void push(const T &value); // 压栈  
T pop(); // 出栈  
T getTop()const; // 取栈顶元素  
};

template <class T>  
T linkStack<T>::pop() {  
if (empty())throw outOfRange();  
Node \*p = top;  
T value = p->data; // value保存栈顶元素的值  
top = top->next; // top指向向后移动  
delete p; // 删除栈顶元素  
return value;  
}

已知链栈的类型定义如下，  
请实现出栈算法T pop()，注意处理栈为空的情况。  
  
template <class T>  
class seqStack {   
private:  
T \* data; // 存放栈中元素的数组  
int top; // 栈顶指针，指向栈顶元素  
int maxSize; // 栈的大小  
void resize(); // 扩大栈空间  
public:  
seqStack(int initSize = 100);   
~seqStack(){ delete [] data;}  
void clear() { top = -1; } // 清空栈   
bool empty() const{ return top == -1;}// 判空  
int size() const{ return top+1; } // 求长度  
void push(const T &value); // 进栈  
T pop(); // 出栈  
T getTop() const; // 取栈顶元素  
};

template <class T>  
T seqStack<T>::pop(){   
if(empty())throw outOfRange(); // 空栈无法弹栈  
return data[top--]; // 修改栈顶指针，返回栈顶元素  
}

已知顺序栈的类型定义如下，  
请实现进栈算法void push(const T &value) ，若栈满了需要扩大栈空间。  
  
template <class T>  
class seqStack {   
private:  
T \* data; // 存放栈中元素的数组  
int top; // 栈顶指针，指向栈顶元素  
int maxSize; // 栈的大小  
void resize(); // 扩大栈空间  
public:  
seqStack(int initSize = 100);   
~seqStack(){ delete [] data;}  
void clear() { top = -1; } // 清空栈   
bool empty() const{ return top == -1;}// 判空  
int size() const{ return top+1; } // 求长度  
void push(const T &value); // 进栈  
T pop(); // 出栈  
T getTop() const; // 取栈顶元素  
};

template <class T>  
void seqStack<T>::push(const T &value) {  
// 检查顺序栈是否已满   
if (top == maxSize - 1) resize();   
data[++top] = value;// 修改栈顶指针，新元素入栈  
}

设将n（n>1）个整数存放到一维数组R中。试设计一个在时间和空间两方面都尽可能高效的算法，将R中保存的序列循环左移P（0<P<n）个位置， 即将R中的数据由（X0,X1,…,Xn-1）变换为（Xp,Xp+1,…,Xn-1,X0,X1,…,Xp-1）。  
  
注意:请按照如下格式书写答案，用于交换的临时变量名为t  
void leftshift(int R[], int p, int n)  
//将有n个元素的一维数组R的序列循环左移p(0<p<n)个位置  
{   
  
//填代码  
  
}

void leftshift(int R[], int p, int n)  
//将有n个元素的一维数组R的序列循环左移p(0<p<n)个位置  
{ int t; //t和数组R中的元素具有相同类型  
for(i=0;i<p/2;i++) //逆置0..p-1段  
{ t=R[i]; R[i]=R[p-1-i]; R[p-1-i]=t;}  
for(i=p;i<(n+p)/2;i++) //逆置p..n-1段  
{ t=R[i]; R[i]=R[n-1-i+p];R[n-1-i+p]=t;}  
for(i=0;i<n/2;i++) //逆置0..n-1段，即整个数组逆置  
{ t=R[i]; R[i]=R[n-1-i]; R[n-1-i]=t;}  
}

已知单链表的类型定义如下，  
请实现查找算法int search(const elemType&value)const; 查找值为value的元素第一次出现时的位序（若表长为n，则位序范围为0~n-1）。注意：请将工作指针命名为p。  
  
单链表的类型定义：  
template <class elemType> // elemType为单链表存储的元素类型  
class linkList{   
private:   
struct Node { // 结点类型  
public:   
elemType data; // 结点的数据域   
Node \* next; // 结点的指针域  
Node(const elemType value, Node\* p = NULL){ // 两个参数的构造函数   
data = value; next = p;   
}   
Node(Node\* p = NULL) { // 一个参数的构造函数   
next = p;   
}   
};  
Node\* head; // 单链表的头指针   
public:   
linkList(); // 构造函数   
~linkList(); // 析构函数   
int search(const elemType&value)const; // 查找值为value的元素第一次出现的位序  
……  
};

void linkList:: inverse(){  
Node \*p,\*tmp;  
p=head->next;   
head->next=NULL;   
if(p)tail=p;   
while(p){  
tmp=p->next;   
p->next=head->next;   
head->next=p;   
p=tmp;   
}  
}

已知一个带有表头结点的单链表，结点结构为(data,link)，假设该链表只给出了头指针list。在不改变链表的前提下，请设计一个尽可能高效的算法，查找链表中倒数第k个位置上的结点（k为正整数），若查找成功，算法输出该结点的data域的值，并返回1；否则，只返回0.  
注意:请按照如下格式书写答案,工作指针命名为p、q  
int SearchInvK(const int k)  
{ //在单链表list上查找倒数第k个结点  
  
//填代码  
  
}

int SearchInvK(const int k)  
{   
p=list->link;   
q=list;   
i=1;  
while(p && i<k) {i++; p=p->link; }   
if(p==null) {cout<<”不存在\n”; return 0; }  
while(p) { q=q->link; p=p->link; }   
cout<<”倒数第k个元素的data域：”<<q->data<<endl;  
return 1;  
}

假设一个单循环链表，其结点含有三个域pre、data、next。其中data为数据域；pre为指针域，它的值为空指针（null）；next为指针域，它指向后继结点。请设计算法，将此表改成双向循环链表。  
  
注意:请按照如下格式书写答案  
void SToDouble（LinkedList la）  
{  
  
//填代码  
  
}

void SToDouble(LinkedList la)  
{ while(la->next->pre==null)  
{ la->next->pre=la;   
la=la->next;   
}  
}

当数据量特别大需借助外部存储器对数据进行排序，这种排序称为\_\_\_\_\_\_\_\_。

外排序

分别采用堆排序，快速排序，冒泡排序和归并排序，对初态为有序的表，则最省时间的是\_\_\_\_\_\_\_\_排序算法，最费时间的是\_\_\_\_\_\_\_\_排序算法。

冒泡  
快速

希尔排序是把记录按下标的一定增量分组，对每组记录进行直接插入排序，随着增量\_\_\_\_\_\_\_\_每组包含的记录越来越多，当增量的值为\_\_\_\_\_\_\_\_时所有记录合为一组。（第1个空填“增大”或“减小”）

减小  
1

若不考虑基数排序，则在排序过程中，主要进行的两种基本操作是关键字的\_\_\_\_\_\_\_\_和记录的\_\_\_\_\_\_\_\_。

比较  
移动

设有一组记录的关键字为 {１９，１４，２３，１，６８，２０，８４，２７，５５，１１，１０，７９}，用链地址法构造散列表，散列函数为H(key) ＝ key MOD 13，则需 \_\_\_\_\_\_\_\_个链表，这些链表的头指针构成一个指针数组，数组的最大下标为\_\_\_\_\_\_\_\_，散列地址为1的链中有\_\_\_\_\_\_\_\_个记录。

13  
12  
4

在线性表的哈希存储中，装填因子α又称为装填系数，若用ｍ表示哈希表的长度，ｎ表示表中的元素的个数，则α等于\_\_\_\_\_\_\_\_。

n/m

如果按关键码值递增的顺序依次将关键码值插入到二叉查找树中，则对这样的二叉查找树检索时，平均比较次数为\_\_\_\_\_\_\_\_。（用“/”表示除法）

(n+1)/2  
(1+n)/2

己知有序表为(12，18，24，35，47，50，62，83，90，115，134)当用折半查找法查找90时，需\_\_\_\_\_\_\_\_次查找成功，47时\_\_\_\_\_\_\_\_成功，查100时，需\_\_\_\_\_\_\_\_次才能确定不成功。

2  
4  
3

在有序表A[1…20]中，按折半查找方法进行查找，查找长度为4的元素的下标从小到大依次是\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_。(用英文状态的逗号“,”做下标序列的分隔符，最后一个数字后不需要分隔符)

1,3,6,8,11,13,16,19

顺序查找n个元素的顺序表，若查找成功，则比较关键字的次数最多为\_\_\_\_\_\_\_\_次；当使用监视哨时，若查找失败，则比较关键字的次数为\_\_\_\_\_\_\_\_。

n  
n+1

归并排序既适合内排序也适合外排序。

正确

在执行某个排序算法过程中，出现了排序码朝着最终排序序列位置相反方向移动，则该算法是不稳定的。

错误

由于希尔排序的最后一趟与直接插入排序过程相同，因此前者一定比后者花费的时间更多。

错误

排序算法中的比较次数与初始元素序列的排列无关。

错误

内排序要求数据一定要以顺序方式存储。

错误

随着装填因子α的增大，用闭散列法解决冲突，其平均搜索长度比用开散列法解决冲突时的平均搜索长度增长得慢。

错误

采用线性探测法处理散列时的冲突，当从哈希表删除一个记录时，不应将这个记录的所在位置置空，因为这会影响以后的查找。

正确

Hash表的平均查找长度与处理冲突的方法无关。

错误

散列函数越复杂越好，因为这样随机性好，冲突概率小。

错误

在散列检索中，“比较”操作一般是不可避免的。

正确

平衡二叉树中，若某个结点的左、右孩子的平衡因子为零，则该结点的平衡因子一定是零。

错误

在二叉排序树中插入一个新结点，总是插入到叶结点下面。

错误

折半查找法的查找速度一定比顺序查找法快 。

错误

用顺序表和单链表表示的有序表均可使用折半查找方法来提高查找速度。

错误

下列排序方法中，若将顺序存储更换为链式存储，则算法的时间效率会降低的是（ ）。  
Ⅰ．插入排序 Ⅱ．选择排序   
Ⅲ．起泡排序 Ⅳ．希尔排序   
Ⅴ．堆排序

仅Ⅳ、Ⅴ

在内部排序时，若选择了归并排序而没有选择插入排序，则可能的理由是（ ）。  
Ⅰ．归并排序的程序代码更短   
Ⅱ．归并排序的占用空间更少   
Ⅲ．归并排序的运行效率更高

仅Ⅲ

排序过程中，对尚未确定最终位置的所有元素进行一遍处理称为一趟排序。下列排序方法中，每一趟排序结束时都至少能够确定一个元素最终位置的方法是（ ）。  
I．简单选择排序 Ⅱ．希尔排序   
Ⅲ．快速排序 Ⅳ．堆排序   
V．二路归并排序

仅I、Ⅲ、Ⅳ

对一组数据（2，12，16，88，5，10）进行排序，若前三趟排序结果如下：  
第一趟排序结果：2，12，16，5，10，88  
第二趟排序结果：2，12，5，10，16，88  
第三趟排序结果：2，5，10，12，16，88  
则采用的排序方法可能是（ ）。

起泡排序

若数据元素序列11,12,13,7,8,9,23,4,5是采用下列排序方法之一得到的第二趟排序后的结果，则该排序算法只能是（ ）。

插入排序

下列选项中，不可能是快速排序第2趟排序结果的是（ ）。

3,2,5,4,7,6,9

为了实现快速排序算法，待排序序列宜采用的存储方式是（ ）。

顺序存储

采用递归方式对顺序表进行快速排序，下列关于递归次数的叙述中，正确的是（ ）。

递归次数与每次划分后得到的分区处理顺序无关

希尔排序的组内排序采用的是（ ）。

直接插入排序

用希尔排序方法对一个数据序列进行排序时，若第1趟排序结果为9，1，4，13，7，8，20，23，15，则该趟排序采用的增量（间隔）可能是（ ）。

3

对同一待排序列分别进行折半插入排序和直接插入排序，两者之间可能的不同之处是（ ）。

元素之间的比较次数

用哈希（散列）方法处理冲突（碰撞）时可能出现堆积（聚集）现象。下列选项中，会受堆积现象直接影响的是（ ）

平均查找长度

为提高散列（Hash）表的查找效率，可以采取的正确措施是（ ）  
I、增大装填（载）因子  
II、设计冲突（碰撞）少的散列函数  
III、处理冲突（碰撞）时避免产生聚集（堆积）现象

仅II

在任意一棵非空二叉排序树 T1 中，删除某结点 v 之后形成二叉排序树 T2，再将 v 插入 T2 形成二叉排序树 T3。下列关于 T1 与 T3 的叙述中，正确的是（ ）。  
I.若 v 是 T1 的叶结点，则 T1 与 T3 不同  
II.若 v 是 T1 的叶结点，则 T1 与 T3 相同  
III.若 v 不是 T1 的叶结点，则 T1 与 T3 不同  
IV.若 v 不是 T1 的叶结点，则 T1 与 T3 相同

仅 II、III

对于下列关键字序列，不可能构成某二叉排序树中一条查找路径的序列是（ ）。

95,22,91,24,94,71

在有n（n>1000）个元素的升序数组A中查找关键字x。查找算法的伪代码如下所示：  
k = 0;  
while ( k<n 且 A[k]<x ) k = k+3;  
if ( k<n 且 A[k]==x ) 查找成功;  
else if ( k-1<n 且 A[k-1]==x ) 查找成功;  
else if ( k-2<n 且 A[k-2]==x ) 查找成功;  
else 查找失败;  
本算法与二分查找（折半查找）算法相比，有可能具有更少比较次数的情形是（ ）。

当x接近数组开头处

下列选项中，不能构成折半查找中关键字比较序列的是（ ）。

500，450，200，180

已知一个长度为16的顺序表L，其元素按关键字有序排列，若采用折半查找法查找一个不存在的元素，则比较次数最多的是（ ）。

5

对{05,46,13,55,94,17,42}进行基数排序，一趟排序的结果是（ ）。

42,13,94,05,55,46,17

有一组数据(15，9，7，8，20，-1，7，4)，用堆排序的筛选方法建立的初始堆为（ ）。

-1，4，7，8，20，15，7，9

设被排序的结点序列共有N个结点，在该序列中的结点已十分接近排序的情况下，用直接插入法、归并法和一般的快速排序法对其排序，这些算法的时间复杂性应为（ ）。

O(N)，O(N\*logN),O(N\*N)

一组记录的关键码为(46，79，56，38，40，84)，则利用快速排序的方法，以第一个记录为基准得到的一次划分结果为（ ）。

(40,38,46,56,79,84)

如果只想得到1000个元素组成的序列中第5个最小元素之前的部分排序的序列，用（ ）方法最快。

堆排序

下列排序算法中（ ）排序在一趟结束后不一定能选出一个元素放在其最终位置上。

归并

就平均性能而言，目前最好的内排序方法是（ ）排序法。

快速

若需在O(nlog2n)的时间内完成对数组的排序，且要求排序是稳定的，则可选择的排序方法是（ ）。

归并排序

一个排序算法的时间复杂度与以下哪项有关（ ）。

所需比较关键字的次数

下面给出的四种排序方法中，排序过程中的比较次数与排序方法无关的是（ ）。

选择排序法

散列表的地址区间为0-17,散列函数为H(K)=K mod 17。采用线性探测法处理冲突，并将关键字序列26，25，72，38，8，18，59依次存储到散列表中。存放元素59需要搜索的次数是（ ）。

4

散列表的地址区间为0-17,散列函数为H(K)=K mod 17。采用线性探测法处理冲突，并将关键字序列26，25，72，38，8，18，59依次存储到散列表中。元素59存放在散列表中的地址是（ ）。

11

假定有k个关键字互为同义词，若用线性探测法把这k个关键字存入散列表中，至少要进行多少次探测？( )

k（k+1）/2次

设哈希表长M=14，哈希函数H(KEY)=KEY MOD 11。表中已有4个结点：ADDR(15)=4， ADDR(38)=5，ADDR(61)=6，ADDR(84)=7，其余地址为空，如用二次探测再散列处理冲突，关键字为49的结点的地址是( )。

9

下面关于哈希(Hash，散列)查找的说法正确的是( )

不存在特别好与坏的哈希函数，要视情况而定

设有一组记录的关键字为{19，14，23，1，68，20，84，27，55，11，10，79}，用链地址法构造散列表，散列函数为H（key）=key MOD 13,散列地址为1的链中有（ ）个记录。

4

在平衡二叉树中插入一个结点后造成了不平衡，设最低的不平衡结点为A,并已知A的左孩子的平衡因子为0，右孩子的平衡因子为1,则应作（ ）型调整以使其平衡。

RL

分别以下列序列构造二叉排序树，与用其它三个序列所构造的结果不同的是( )。

(100,60,80,90, 20,110,130)

当采用分块查找时，数据的组织方式为（ ）。

数据分成若干块，每块内数据不必有序，但块间必须有序，每块内最大（或最小）的数据组成索引块

具有12个关键字的有序表，折半查找的平均查找长度（ ）。

3.1

折半查找的时间复杂性为（ ）。

O(logN)

在一个有N个元素的有序单链表中查找具有给定关键字的结点，平均情况下的时间复杂性为（ ）。

O(N)

下列二叉排序树中查找效率最高的是（ ）。

平衡二叉树

对顺序表进行折半查找时，要求顺序表必须（ ）。

以顺序方式存储，且数据元素有序

Dijkstra最短路径算法从源点到其余各顶点的最短路径的路径长度按\_\_\_\_\_\_\_次序依次产生，当弧上的权值为\_\_\_\_\_\_\_数时，该算法不能正确产生最短路径。

递增  
负

对于一个具有n个顶点和e条边的连通图，其生成树中的顶点数和边数分别为\_\_\_\_\_\_\_和\_\_\_\_\_\_\_。

n  
n-1

一无向图G（V，E），其中V（G）={1，2，3，4，5，6，7}，E（G）={（1,2），(1,3），（2,4），（2,5），（3,6），（3,7），（6,7）（5,1）},对该图从顶点3开始进行遍历，去掉遍历中未走过的边，得一生成树G’(V，E’）,V（G’）=V（G），E（G’）={（1,3），（3,6），（7,3），（1,2），（1,5），（2,4）}，则采用的遍历方法是\_\_\_\_\_\_\_。

广度优先

已知一无向图G=（V，E），其中V={a，b，c，d，e }，E={(a,b)，(a,d)，(a,c)，(d,c)，(b,e)}现用某一种图遍历方法从顶点a开始遍历图，得到的序列为abecd，则采用的是\_\_\_\_\_\_\_遍历方法。

深度优先

在有n个顶点的有向图中，若要使任意两点间可以互相到达，则至少需要\_\_\_\_\_\_\_条弧。

N

在有n个顶点的有向图中，每个顶点的度最大可达\_\_\_\_\_\_\_。（用“\*”表示乘号，“/”表示除号）

2\*(n-1)

G是一个非连通无向图，共有28条边，则该图至少有\_\_\_\_\_\_个顶点。

9

若用n表示图中顶点数目，则有\_\_\_\_\_\_\_条边的无向图称为完全图。（用“\*”表示乘号，“/”表示除号）

n\*(n-1)/2

Dijkstra算法可解决存在负权值但无回路的有向图的单源点最短路径问题。（ ）

错误

有环图也能进行拓扑排序。（ ）

错误

连通图上各边权值均不相同，则该图的最小生成树是唯一的。（ ）

正确

对一个无向图进行深度优先搜索时，得到的深度优先搜索序列是唯一的。

错误

用邻接矩阵法存储一个图所需的存储单元数目与图的边数有关。

错误

无向图的邻接矩阵一定是对称矩阵，有向图的邻接矩阵一定是非对称矩阵。

错误

有n-1条边的图肯定都是生成树。

错误

若有向图不存在回路，即使不用访问标志位同一结点也不会被访问两次。

正确

n个结点的无向图，若不允许结点到自身的边，也不允许结点到结点的多重边,且边的总数为n(n-1)/2，则该无向图一定是连通图。

正确

若将n个顶点e条弧的有向图采用邻接表存储，则拓扑排序算法的时间复杂度是（ ）

O(n+e)

若用邻接矩阵存储有向图，矩阵中主对角线以下的元素均为零，则关于该图拓扑序列的结论是（ ）

存在，可能不唯一

下列关于图的叙述中，正确的是（ ）  
Ⅰ. 回路是简单路径   
Ⅱ. 存储稀疏图，用邻接矩阵比邻接表更省空间   
Ⅲ. 若有向图中存在拓扑序列，则该图不存在回路

仅Ⅲ

下列关于最小生成树的叙述中，正确的是（ ）  
Ⅰ. 最小生成树的代价唯一  
Ⅱ. 所有权值最小的边一定会出现在所有的最小生成树中   
Ⅲ. 使用普里姆（Prim）算法从不同顶点开始得到的最小生成树一定相同  
IV. 使用普里姆（Prim）算法和克鲁斯卡尔（Kruskal）算法得到的最小生成树总不相同

仅Ⅰ

对有n个顶点、e条边且使用邻接表存储的有向图进行广度优先遍历，其算法时间复杂度是（ ）

O(n+e)

设有向图G=(V,E)，顶点集V={V0,V1,V2,V3}，边集E={<V0,V1>,<V0,V2>,<V0,V3>，<V1,V3>},若从顶点V0开始对图进行深度优先遍历，则可能得到的不同遍历序列个数是（ ）

5

在有向图的邻接表存储结构中，顶点v在链表中出现的次数是( )

顶点v的入度

用邻接表存储图所用的空间大小（ ）

与图的顶点数和边数都有关

已知无向图G含有16条边，其中度为4的顶点个数为3，度为3的顶点个数为4，其他顶点的度均小于3。图G所含的顶点个数至少是（ ）

11

若无向图G=（V.E）中含7个顶点，则保证图G在任何情况下都是连通的，则需要的边数最少是（ ）

16

下列关于无向连通图特性的叙述中，正确的是（ ）  
Ⅰ. 所有顶点的度之和为偶数   
Ⅱ. 边数大于顶点个数减1   
Ⅲ. 至少有一个顶点的度为1

只有Ⅰ

在有向图G的拓扑序列中，若顶点Vi在顶点Vj之前，则下列情形不可能出现的是（ ）。

G中有一条从Vj到Vi的路径

已知有向图G=(V,E)，其中V={V1,V2,V3,V4,V5,V6,V7}，E={<V1,V2>,<V1,V3>,<V1,V4>,<V2,V5>,<V3,V5>,<V3,V6>,<V4,V6>,<V5,V7>,<V6,V7>},G的拓扑序列是（ ）。

V1,V3,V4,V6,V2,V5,V7

当各边上的权值（ ）时，BFS算法可用来解决单源最短路径问题。

均相等

(1) 求从指定源点到其余各顶点的迪杰斯特拉（Dijkstra）最短路径算法中弧上权不能为负的原因是在实际应用中无意义；  
(2) 利用Dijkstra求每一对不同顶点之间的最短路径的算法时间是O(n3 ) ；（图用邻接矩阵表示）  
(3) Floyed求每对不同顶点对的算法中允许弧上的权为负，但不能有权和为负的回路。  
上面不正确的是（ ）。

(1)

在图采用邻接表存储时，求最小生成树的 Prim 算法的时间复杂度为( )。

O(n+e)

判断一个有向图是否有环（回路）除了拓扑排序方法，还可以用( )。

深度优先遍历

若一个有向图具有拓扑排序序列，那么它的邻接矩阵必定为( )。

一般矩阵

下列哪一种图的邻接矩阵是对称矩阵？（ ）

无向图

n个结点的完全有向图含有边的数目（ ）。

n（n＋１）

要连通具有n个顶点的有向图，至少需要（ ）条边。

N

一个n个顶点的连通无向图，其边的个数至少为（ ）。

n-1

一个有n个结点的图，最多有（ ）个连通分量。

N

一个有n个结点的图，最少有（ ）个连通分量。

1

在有向图的邻接表存储结构中，顶点v在链表中出现的次数是( )。

顶点v的入度

无向图G=(V，E)，其中：V={a，b，c，d，e，f}，E={(a，b)，(a，e)，(a，c)，(b，e)，(c，f)，(f，d)，(e，d)}，对该图进行深度优先遍历，得到的顶点序列正确的是（ ）。

a,e,d,f,c,b

下列说法不正确的是（ ）。

图的深度遍历不适用于有向图

图的BFS生成树的树高比DFS生成树的树高（ ）

小或相等

堆一般采用\_\_\_\_\_\_\_\_结构存储表示。具有n个元素的最小堆，其最大值元素的可能位置介于\_\_\_\_\_\_\_\_和\_\_\_\_\_\_\_\_之间（根结点位于1号单元）。（遵循C语言规则，“/”用于整数除法求得结果为商）

顺序  
n/2+1  
n

给定一组数据{6，2，7，10，3，12}以它构造一棵哈夫曼树，则树高为\_\_\_\_\_\_\_\_，带权路径长度WPL的值为\_\_\_\_\_\_\_\_。

5  
96

设n为哈夫曼树的叶子结点数目，则该哈夫曼树共有\_\_\_\_\_\_\_\_个结点。(乘号用\*表示)

2\*n-1

有一份电文中共使用 6个字符:a,b,c,d,e,f,它们的出现频率依次为2,3,4,7,8,9，试构造一棵哈夫曼树，则其加权路径长度WPL为\_\_\_\_\_\_\_\_。

80

某二叉树的后序遍历序列是dabec，中序遍历序列是debac，前序遍历序列是\_\_\_\_\_\_\_\_\_。

Cedba

已知一棵二叉树的前序序列为abdecfhg,中序序列为dbeahfcg，则该二叉树的根为\_\_\_\_\_\_\_\_\_，左子树中有\_\_\_\_\_\_\_\_\_，右子树中有\_\_\_\_\_\_\_\_\_。

a  
dbe  
hfcg

完全二叉树中，结点个数为n，结点的编号从1开始，则编号最大的分支结点的编号为\_\_\_\_\_\_\_\_\_。（遵循C语言规则，“/”用于整数除法求得结果为商）

n/2

设F是由T1,T2,T3三棵树组成的森林,与F对应的二叉树为B,已知T1,T2,T3的结点数分别为n1,n2和n3则二叉树B的左子树中有\_\_\_\_\_\_\_\_\_个结点，右子树中有\_\_\_\_\_\_\_\_\_个结点。

n1-1  
n2+n3

一棵有n个结点的满二叉树有\_\_\_\_\_\_\_\_\_个度为1的结点，有\_\_\_\_\_\_\_\_\_个分枝（非终端）结点和\_\_\_\_\_\_\_\_\_个叶子。（遵循C语言规则，“/”用于整数除法求得结果为商）

0  
(n-1)/2

堆是满二叉树。

错误

哈夫曼树是带权路径长度最短的树，路径上权值较大的结点离根较近。

正确

哈夫曼树无左右子树之分。

错误

一棵哈夫曼树的带权路径长度等于其中所有分支结点的权值之和。

错误

若从二叉树的任一结点出发，到根的路径上所经过的结点序列按其关键字有序，则该二叉树一定是哈夫曼树。

错误

不用递归就不能实现二叉树的前序遍历。

错误

完全二叉树的存储结构通常采用顺序存储结构。

正确

给定一棵树，可以找到唯一的一棵二叉树与之对应。

正确

完全二叉树中，若一个结点没有左孩子，则它必是树叶。

正确

采用孩子兄弟链表作存储结构，树的前序遍历和其相应的二叉树的前序遍历的结果是一样的。

正确

在含有n个结点的树中，边数只能是n-1条。

正确

二叉树是度为2的有序树。

错误

已知一棵二叉树的树形如下图所示，其后序序列为e,a,c,b,d,g,f，树中与结点a同层的结点是( )

D

给定二叉树如图所示。设N代表二叉树的根，L代表根结点的左子树，R代表根结点的右子树。若遍历后的结点序列为3,1,7,5,6,2,4，则其遍历方式是( )

RNL

已知小根堆为8，15，10，21，34，16，12，删除关键字8之后需重建堆，在此过程中，关键字之间的比较数是（ ）。

3

已知关键字序列5,8,12,19,28,20,15,22是小根堆（最小堆）,插入关键字3,调整后得到的小根堆是（ ）。

3,5,12,8,28,20,15,22,19

已知字符集{a,b,c,d,e,f,g,h},若各字符的哈夫曼编码依次是0100,10,,0000，0101,001,011,11,0001，则编码序列0100011001001011110101 的译码结果是（ ）

a f e e f g d

下列选项给出的是从根分别到达两个叶节点路径上的权值序列，能属于同一棵哈夫曼树的是( )

24，10，5和 24，14，6

5个字符有如下4种编码方案，不是前缀编码的是( )

0,100,110,1110,1100

对n（n≥2）个权值均不相同的字符构造哈夫曼树。下列关于该哈夫曼树的叙述中，错误的是（ ）

该树一定是一棵完全二叉树

若森林F有15条边、25个结点，则F包含的树的个数是（ ）。

10

将森林F转换为对应的二叉树T，F中叶结点的个数等于（ ）。

T中左孩子指针为空的结点个数

已知一棵有2011个结点的树，其叶子结点个数为116，该树对应的二叉树中无右孩子的结点个数是（ ）。

1896

在一棵度为4的树T中，若有20个度为4的结点，10个度为3的结点，1个度为2的结点，10个度为1的结点，则树T的叶结点个数是（ ）。

82

将森林转换为对应的二叉树，若在二叉树中，结点u是结点v的父结点的父结点，则在原来的森林中，u和v可能具有的关系是（ ）。  
Ⅰ. 父子关系   
Ⅱ. 兄弟关系   
Ⅲ. u的父结点与v的父结点是兄弟关系

Ⅰ和Ⅱ

要使一棵非空二叉树的先序序列与中序序列相同，其所有非叶结点须满足的条件是 ( )

只有右子树

先序序列为a,b,c,d的不同二叉树的个数是( )

14

若一棵二叉树的前序遍历序列为a，e，b，d，c，后序遍历序列为b，c，d，e，a，则根结点的孩子结点（ ）

只有e

棵非空的二叉树的先序序列和后序序列正好相反，则该二叉树一定满足( )

其中只有一个叶子结点

若一棵二叉树的前序遍历序列和后序遍历序列分别是：1、2、3、4和4、3、2、1，则该二叉树的中序序列不会是（ ）

3,2,4,1

有n个结点，并且高度为n的二叉树的数目为（ ）。

2n-1

一棵深度为4的完全二叉树，最少有（ ）个结点。

8

若一棵完全二叉树有768个结点，则该二叉树中叶结点的个数是（ ）

384

已知一棵完全二叉树的第6层（设根是第1层）有8个叶结点，则该完全二叉树的结点个数最多是（ ）

111

有一组数据(15，9，7，8，20，-1，7，4)，用堆的向下筛选方法建立的初始小根堆为( )。

-1，4，7，8，20，15，7，9

从堆中删除一个元素的时间复杂度为( ) 。

0(log2n)

以下序列不是堆的是( )。

100,85,40,77,80,60,66,98,82,10,20

在含有n个关键字的小根堆(堆顶元素最小)中，关键字最大的记录有可能存储在( )位置上。

n/2 +2

根据以权值为{2，5，7，9，12}构造的哈夫曼树所构造的哈夫曼编码中最大的长度为（ ）

3

在下列情况中，可称为二叉树的是（ ）。

哈夫曼树

在哈夫曼树中，任何一个结点的度都是（ ）。

0或2

已知一算术表达式的中缀形式为A+B\*C-D/E，后缀形式为ABC\*+DE/-，其前缀形式为（ ）。

-+A\*BC/DE

设F是一个森林，B是由F变换得的二叉树。若F中有n个非终端结点，则B中右指针域为空的结点有（ ）个。

n+1

由3 个结点可以构造出多少种不同的二叉树？（ ）

5

对任意一棵树，设它有n个结点，这n个结点的度数之和为( )。

n-1

二叉树的先序遍历和中序遍历如下： 先序遍历：EFHIGJK；中序遍历: HFIEJKG 。该二叉树根的右子树的根是：

G

高度为h(h>0)的满二叉树对应的森林由( )棵树构成。

H

一棵124个叶结点的完全二叉树，最多有( )个结点。

248

有关二叉树下列说法正确的是（ ）

一棵二叉树的度可以小于2

树的后根遍历序列等同于该树对应的二叉树的( )

中序序列

若用一维数组表示一个深度为5、结点个数10的二叉树，数组的长度至少为（ ）。

16

已知一棵完全二叉树中共有626个结点，叶子结点的个数应为( )。

313

一个具有1025个结点的二叉树的高h为（ ）

11至1025之间

一棵树高为K的完全二叉树至少有（ ）个结点

2k-1

某二叉树的先序序列和后序序列正好相反，则该二叉树一定是( )的二叉树

高度等于其结点数

字符串’ababaaab’的nextVal数组值序列为\_\_\_\_\_\_\_\_\_\_\_\_\_\_。(用空格分隔各数值)

-1 0 -1 1 -1 3 1 0

模式串P=‘abaabcac’的next数组值序列为\_\_\_\_\_\_\_\_\_\_\_\_\_\_。(用空格分隔各数值)

-1 0 0 1 1 2 0 1

设主串长度为n，模式串长度为m，则KMP算法的时间复杂度为\_\_\_\_\_\_\_\_\_\_\_\_\_\_。

O(m+n)

组成串的数据元素只能是\_\_\_\_\_\_\_\_\_\_\_\_\_\_。

字符

用循环链表表示的队列长度为n，若只设头指针，则出队和入队的时间复杂度分别是\_\_\_\_\_\_和\_\_\_\_\_\_；若只设尾指针，则出队和入队的时间复杂度分别是\_\_\_\_\_\_和\_\_\_\_\_\_。（填O(1) 、O(log2n) 、O(n) 、O(nlog2n)或O(n\*n)）

O(1)  
O(n)  
O(1)  
O(1)

在循环队列中，队列长度为n ,存储位置从0到n-1编号，以rear指示实际的队尾元素，现要在此队列中插入一个新元素，新元素的位置是\_\_\_\_\_\_\_\_\_\_\_\_\_\_。（填rear的计算公式）

rear=(rear+1)%n

循环队列的引入，目的是为了克服\_\_\_\_\_\_\_\_\_\_\_\_\_\_的现象。

假溢出

表达式23+(12\*3-2)的后缀表达式是\_\_\_\_\_\_\_\_\_\_\_\_\_\_。

23 12 3\*2- +

栈和队列都是操作受限的线性表，栈的运算遵循\_\_\_\_\_\_\_\_\_\_\_\_\_\_的原则，队列的运算遵循\_\_\_\_\_\_\_\_\_\_\_\_\_\_的原则。

先进后出  
先进先出

对于一个具有n个结点的单链表，在已知的结点p后插入一个新结点的时间复杂度为\_\_\_\_\_\_\_\_\_,在值为x的结点后插入一个新结点的时间复杂度为\_\_\_\_\_\_\_\_\_。（填O(1) 、O(log2n) 、O(n) 、O(nlog2n)或O(n\*n)）

O(1)   
O(n)

在单链表L中，指针p所指结点有后继结点的条件是\_\_\_\_\_\_\_\_\_。(写出关系表达式)

p->next!=NULL

顺序存储结构是通过\_\_\_\_\_\_\_\_\_表示元素之间的关系的;链式存储结构是通过\_\_\_\_\_\_\_\_\_表示元素之间的关系的。

元素的存储位置  
指针

串长度是指串中不同字符的个数。

错误

两个长度不相同的串有可能相等。

错误

设模式串的长度为m,目标串的长度为n，当n≈m且处理只匹配一次的模式时，朴素的匹配（即子串定位函数）算法所花的时间代价可能会更为节省。

正确

KMP算法的特点是在模式匹配时指示主串的指针不会变小。

正确

栈和队列的存储方式，既可以是顺序方式，又可以是链式方式。

正确

在链队列中，即使不设置尾指针也能进行入队操作。

正确

栈和队列都是线性表，只是在插入和删除时受到了一些限制。

正确

通常使用队列来处理函数或过程的调用。

错误

消除递归不一定需要使用栈。

正确

线性表中每个元素都有一个直接前驱和一个直接后继。

错误

顺序存储方式的优点是存储密度大，且插入、删除运算效率高。

错误

顺序存储方式只能用于存储线性结构。

错误

对任何数据结构链式存储结构一定优于顺序存储结构。

错误

顺序存储方式插入和删除时效率太低，因此它不如链式存储方式好。

错误

线性表采用链表存储时，结点和结点之间的存储空间可以是不连续的。

正确

已知字符串S为“abaabaabacacaabaabcc”. 模式串t为“abaabc”, 采用KMP算法进行匹配，第一次出现“失配”(s[i] != t[i]) 时，i=j=5,则下次开始匹配时，i和j的值分别是

i=5，j=2

循环队列存放在一维数组A[0..M-1]中，end1指向队头元素，end2指向队尾元素的后一个位置。假设队列两端均可进行入队和出队操作，队列中最多能容纳M－1个元素，初始时为空。下列判断队空和队满的条件中，正确的是（ ）

队空：end1==end2; 队满：end1==(end2+1) mod M

某队列允许在其两端进行入队操作，但仅允许在一端进行出队操作。若元素a，b，c，d，e依次入此队列后再进行出队操作，则不可能得到的出队序列是（ ）

d,b,c,a,e

为解决计算机主机与打印机之间速度不匹配问题，通常设置一个打印数据缓冲区，主机将要输出的数据依次写入该缓冲区，而打印机则依次从该缓冲区中取出数据。该缓冲区的逻辑结构应该是（ ）

队列

下列关于栈的叙述中，错误的是（ ）  
I、采用非递归方式重写递归程序时必须使用栈   
II、函数调用时，系统要用栈保存必要的信息  
III、只要确定了入栈次序，即可确定出栈次序

仅I、III、IV

已知程序如下：  
int s(int n)  
{ return (n<=0) ? 0 : s(n-1) +n; }  
void main()  
{ cout<< s(1); }  
程序运行时使用栈来保存调用过程的信息，自栈底到栈顶保存的信息一次对应的是

main()->S(1)->S(0)

假设栈初始为空，将中缀表达式a/b+(c\*d-e\*f)/g转换为等价的后缀表达式的过程中，当扫描到f时，栈中的元素依次是（ ）

+(-\*

中缀表达式（A+B）\*（C-D）/(E-F\*G)的后缀表达式是（ ）

AB+CD-\*EFG\*-/

已知操作符包括‘+’，‘-’，‘\*’，‘/’，‘(’和‘)’。将中缀表达式a+b-a\*((c+d)/e-f)+g转换为等价的后缀表达式ab+acd+e/f-\*-g+时，用栈来存放暂时还不能确定运算次序的操作符。若栈初始时为空，则转换过程中同时保存在栈中的操作符的最大个数是（ ）

5

一个栈的入栈序列为 1, 2, 3,…,n，其出栈序列是 p1, p2 , p3…pn。若 p2为 3 ，则 p3 可能取值的个数是( )

n-1

元素a，b，c，d，e依次进入初始为空的栈中，若元素进栈后可停留、可出栈，直到所有元素都出栈，则在所有可能的出栈序列中，以元素d开头的序列个数是

4

若元素a，b，c，d，e，f依次进栈，允许进栈、退栈操作交替进行，但不允许连续三次进行退栈操作，则不可能得到的出栈序列是（ ）

a,f,e,d,c,b

己知一个带有表头结点的双向循环链表L，结点结构为（prev，data，next），其中prev和next分别是指向其直接前驱和直接后继结点的指针。现要删除指针p所指的结点，正确的语句序列是（ ）

p->next->prev = p->prev; p->prev->next = p->next; free (p);

若串S=“software”,其子串的数目是( )。

37

设有两个串S1和S2，求S2在S1中首次出现的位置的运算称作（ ）。

模型匹配

设S为一个长度为n的字符串，其中的字符各不相同，则S中的互异的非平凡子串（非空且不同于S本身）的个数为（ ）。

(n2/2)+(n/2)-1

串的长度是指（ ）。

串中所含字符的个数

串是一种特殊的线性表，下面哪个叙述体现了这种特殊性？

数据元素是一个字符

下面关于串的的叙述中，哪一个是不正确的？

空串是由空格构成的串

若以1234作为双端队列的输入序列，则既不能由输入受限的双端队列(一端输入，两端输出)得到，也不能由输出受限的双端队列(两端输入，一端输出)得到的输出序列是（ ）

4231

循环队列存储在数组A[0..m]中，则入队时的操作为（ ）。

rear=(rear+1)mod(m+1)

循环队列A[0..m-1]存放其元素值，用front和rear分别表示队头和队尾，则当前队列中的元素数是( )。

(rear-front+m)%m

中缀表达式（A+B）\*（C-D）/(E-F\*G)的后缀表达式是（ ）。

AB+CD-\*EFG\*-/

一个递归算法必须包括（ ）。

终止条件和递归部分

若栈采用顺序存储方式存储，现两栈共享空间V[1..m]，top[i]代表第i个栈( i =1,2)栈顶，栈1的底在v[1]，栈2的底在V[m]，则栈满的条件是（ ）。

top[1]+1=top[2]

设栈的输入序列是1，2，3，4,则（ ）不可能是其出栈序列。

4，3，1，2

一个栈的输入序列为123…n，若输出序列的第一个元素是n，输出第i（1<=i<=n）个元素是（ ）。

n-i+1

对于栈操作数据的原则是（ ）。

后进先出

在循环双链表的p结点之后插入s结点的操作是（ ）

s->prior=p; s->next=p->next; p>next->prior=s; p->next=s;

双向链表中有两个指针域，prior和next分别指向前趋及后继，设p指向链表中的一个结点，现要求删去p所指结点，则正确的删除是（ ）（链中结点数大于2，p不是第一个结点）

p->prior->next = p->next; p->next->prior = p->prior; delete p;

在单链表的指针为p的结点之后插入指针为s的结点，正确的操作是（ ）

s->next = p->next; p->next = s;

对于一个头指针为head的带头结点的单链表，判定该表为空表的条件是（ ）

head->next == NULL

在一个以h为头指针（指向头结点）的单循环链中，p指针指向链尾的条件是（ ）

p->next == h

单链表中，增加一个头结点的目的是为了( ）

方便运算的实现

在n个结点的线性表的顺序实现中，算法的时间复杂性是O(1)的操作是( )

访问第i个结点和求第i个结点的直接前驱

链表不具有的特点是（ ）

可随机访问任一元素

某线性表中最常用的操作是在最后一个元素之后插入一个元素和删除第一个元素，则采用（ ）存储方式最节省运算时间。

仅有尾指针的单循环链表

若某线性表最常用的操作是存取任一指定序号的元素和在最后进行插入和删除运算，则利用（ ）存储方式最节省时间。

顺序表

线性结构中元素之间存在1:1的关系，树形结构中元素之间存在\_\_\_\_\_\_\_\_\_\_的关系，图形结构中元素之间存在\_\_\_\_\_\_\_\_\_\_的关系。（按线性结构样式填写）

1:多  
多:多

评价算法的优劣通常主要考虑算法的\_\_\_\_\_\_\_\_\_\_和\_\_\_\_\_\_\_\_\_\_这两方面。

时间复杂度  
空间复杂度

算法的５个重要特性是有穷性、\_\_\_\_\_\_\_\_\_\_ 、 可行性、输入、\_\_\_\_\_\_\_\_\_\_。

确定性  
输出

常见的数据结构有集合、线性结构、\_\_\_\_\_\_\_\_\_\_结构、\_\_\_\_\_\_\_\_\_结构。

树形  
图形

抽象数据类型与计算机内部表示和实现无关；

正确

数据的逻辑结构与数据元素本身的内容和形式无关。

正确

算法可以没有输出。

错误

程序和算法没有区别。

错误

在数据结构中，从逻辑上可以将之分为( )。

线性结构和非线性结构

下列函数的时间复杂度( )。  
int func(int n)  
{  
int i=0,sum=0;  
while( sum<n ) sum += ++i;  
return i;  
}

O(n1/2)即n的1/2次方

下列程序段的时间复杂度( )。  
count=0;  
for(k=1; k<=n; k\*=2)  
for(j=1; j<=n; j+=1)  
count++;

O(nlogn)

求整数n(n>=0)阶乘的算法如下，其时间复杂度是( )。  
int fact(int n){  
if (n<=1) return 1；  
return n\*fact(n-1)；  
}

O(n)

设n是描述问题规模的非负整数，下面程序片段的时间复杂度是  
x=2;  
while(x<n/2)  
x=2\*x;

O(logn)

算法分析的目的是（ ）

分析算法的效率以求改进

算法的计算量大小称为算法的( )

时间复杂度

链式存储结构所占存储空间 ( )

分两部分一部分存放结点的值，另一部分存放表示结点间关系的指针

非线性结构中的每个结点( )

可能有多个直接前趋结点和多个直接后继结点

数据元素之间没有任何逻辑关系的是( )

集合

数据结构通常是研究数据的( )及它们之间的相互联系

存储结构和逻辑结构