【第一章：绪论】

1. 通常，用 O(1)表示常数计算时间。常见的渐进时间复杂度按数量级递增排列为：

$O(1) < O(\log_2 n) < O(n) < O(n\log_2 n) < O(n^2) < O(n^3) < O(2^n)$

2. 程序一定是算法：算法和程序的一个重要区别在于，程序不一定是有穷的，而算法必须是有穷的。也就是说，一个算法必须具有有穷性，而程序则不一定。

【第二章：线性表】

3. 对于如下语句的理解：

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

这是定义一个结构体，这个结构体有两个属性，一个是 int 类型的 data; 另一个是这个结构体本身类型的指针 next;

给这个结构定义了一个别名:LNode，一个**指针别名**:LinkList;

LNode a; 等价于 struct node a; 声明一个 struct node 结构体类型的结构体变量 a;

LinkList b; 等价于 struct node *b; 等价于 LNode *b; 声明一个 struct node 结构体类型的指针变量 b;

4. 通常，“较稳定”的线性表选择顺序存储，而频繁做插入删除等“动态性”较强的线性表宜选择链式存储。(**线性表**的存储分为**顺序存储**以及**链式存储**)

5. 【单链表逆置】算法思想：分为带头结点和不带头结点两种情况；

（1）带头结点的单链表逆置：保存第一个数据结点（即头指针指向的头结点的下一个结点）至 p,此时，可以通过 p 访问到链表中的每一个结点。这时将原链表置空，再将 p 指向的结点保存在 q 中，并让 p 指向 p 的下一个结点，将 q 插入头结点后即可。（利用了前插法）

代码如下：

```
void reverseWithHeadNode(LinkList H)
{
    LNode *p,*q;
    p = H->next;//让 p 指向第一个数据结点；
    H->next = NULL;//原链表置空；
    while(p)
    {
        q = p;//将当前数据结点保存在 q 中；
        p = p->next;//当前结点指向下一个；
        //将 q 插到头结点后；
        q->next = H->next;
        H->next = q;
    }
}
```

（2）不带头结点的单链表逆置：新建一个空表 p，让 q 指向待操作的表。保存当前处理结

点的后继并取出待逆置结点。将该结点利用前插法插入新表中。
代码如下：

```
//注意带有返回值是因为创建了新表，而需要将原表逆置为新表
LinkList reverse(LinkList H)
{
    LNode *p,*q;
    p = NULL;//新建一个空表；
    q = H;//让 q 指向待操作的表；
    while(q != NULL)
    {
        q = (LNode *)malloc(sizeof(LNode));
        q->data = H->data;
        H = H->next;//保存当前结点的后继至 H；
        //将待逆置结点 q 利用前插法插入新表 p；
        q->next = p;
        p = q;
        q = H;//q 指向下一个待逆置结点
    }
    return p;
}
```

6. 队列采用链式存储结构，入队时只需申请新的结点，不存在队满的情况。

7. VC 规定结构体的各变量存放的起始地址相对于结构体的起始地址的偏移量必须是该变量的类型所占字节数的倍数，并且整个结构体的字节数必须是该结构体中占用空间最大的类型的字节数的整数倍。示例程序如下：

```
#include<iostream.h>
struct {char a; int b; double c;}a;//内存对齐方式：1 000 1111 11111111
struct {int b; char a;  double c;}b;//内存对齐方式：1111 1 000 11111111
struct {char a; double c;  int b;}c;//内存对齐方式：1 0000000 11111111 1000 0000
int main()
{
    cout<<sizeof(a)<<endl;//16
    cout<<sizeof(b)<<endl;//16
    cout<<sizeof(c)<<endl;//24
    return 0;
}
```

8. 试描述头指针和头结点的区别，并说明头指针和头结点的作用。

答：头指针是一个指针变量,用于存放链表中首结点的地址，并以此来标识一个链表。例如，链表 H、链表 L 等，表示链表中第一个结点的地址存放在 H、L 中。

头结点则是附加在第一个元素结点之前的一个结点，头指针指向头结点。当该链表表示一个非空的线性表时，头结点的指针域指向第一个元素结点；当为空表时，该指针域为空。

头指针的作用是用来唯一地标识一个单链表。

头结点的作用有两个：一是使得对空表和非空表的处理得以统一；而是在链表的第一个位置上的操作和在其他位置上的操作一致，无需特殊处理。

9. 【删除重复元素】算法思想：用指针 p 指向第一个数据结点，从它的后继结点开始到表的结束，找与其值相同的结点并删除之；p 指向下一个；以此类推，p 指向最后结点时算法结束。代码如下：

```
//带有头结点的单链表中删除重复结点
void deleteRepeatElement(LinkList H)
{
    LNode *p,*q,*r;
    p = H->next;
    if(p == NULL) return;
    while(p->next)
    {
        q = p;
        while(q->next)
        {
            if(q->next->data == p->data)
            {
                r = q->next;//删除结点时一定要先把结点保存起来
                q->next = r->next;
                free(r);//后来再释放掉空间
            }
            else
                q = q->next;

        }
        p = p->next;
    }

}
```

10. 排列组合公式：

$$P_n^r = n(n-1)\ldots(n-r+1) = \frac{n!}{(n-r)!}$$

$$C_n^r = \frac{P_n^r}{r!} = \frac{n!}{r!(n-r)!}$$

$$C_n^r = C_n^{n-r}$$

11. 假设长度大于 1 的循环单链表中，既无头结点也无头指针，s 为指向该链表中某一结点的指针，编写算法删除该结点的前驱结点。

算法思想：利用循环单链表的特点，可以通过给定的结点找到其前驱结点 p 及 p 的前驱结点 q，删除 p 即可。代码如下：

创建循环单链表的代码：

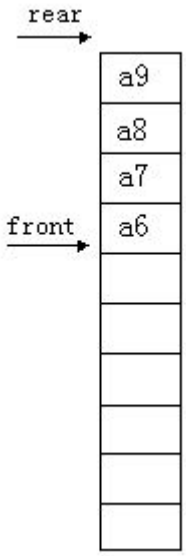
```
LinkList init(LinkList H)
{
```

```
    int data;
    LNode *s;
    scanf("%d",&data);
    while(data > 0)
    {
        s = (LNode *)malloc(sizeof(LNode));
        s->data = data;
        if(H == NULL)
        {
            H = s;
            H->next = H;
        }
        else
        {
            LNode *q;
            q = H;
            while(q->next != H)
                q = q->next;
            q->next = s;
            q->next->next = H;
        }
        scanf("%d",&data);
    }
    return H;
}
//删除给定结点的前驱
void deletePre(LNode *s)
{
    LNode *p,*q;//p 指向 s 前驱，q 指向 p 前驱即 s 前驱的前驱
    p = s;
    while(p->next != s)
    {
        q = p;
        p = p->next;
    }
    q->next = s;
    free(p);
}
```

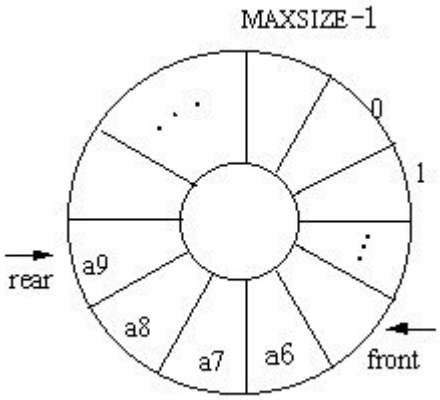
【第三章：栈和队列】

12. 什么是假溢出？如何解决？

答：随着入队出队的进行，会使整个队列整体向后移动（队头队尾指针不断后移），这样就导致：队尾指针已经移到了最后，再有元素入队就会出现溢出，而事实上，此时队中并未真的“满员”，这种现象为“假溢出”。解决方法之一是将队列的数据区看成头尾相接的循环结构，头尾指针的关系不变，将其称为“循环队”。解释如下图：



“假溢出”现象



解决“假溢出”：循环队列

13. 队列中常见操作：

（1）空队时设置为：q->front = q->rear = -1;

（2）在**不溢出的情况下**(条件很重要，作为编程时的判断依据)，入队操作：队尾指针加 1，指向新位置后，元素入队：q->rear++; q->data[q->rear] = x;

（3）在**队不空的情况下**，出队的操作：队头指针加 1，表明原队头元素出队，指针后面的是新队头元素：q->front++; x = q->data[q->front];

（4）循环队列通常少用一个元素空间，约定以“队列头指针在队尾指针的下一位置上”作为队列呈“满”状态的标志。

（5）设有循环队列 sq，队满的判别条件为 (sq->rear+1)%maxsize == sq->front;

或 sq->num == maxsize; 队空的判别条件为 sq->rear == sq->front

14. 假设以数组 A[m]存放循环队列的元素，其头尾指针分别为 front 和 rear，则当前队列中的元素个数为____B____。

A. rear-front+1　　B.(rear-front+m)%m　　C.(front-rear+m)%m　　D.(rear-front)%m

分析：由于循环队列是用数组模拟实现的，当队列满后，再进行一些出队、入队操作之后，可能**存在尾指针小于头指针**的情况，因此，数组中空闲的空间为| rear-front |个。

15. 与栈有关的结论：

（1）栈的应用：数制转换、递归算法、求回文等；

（2）一般情况下，将递归算法转换成等价的非递归算法应设置**堆栈；**

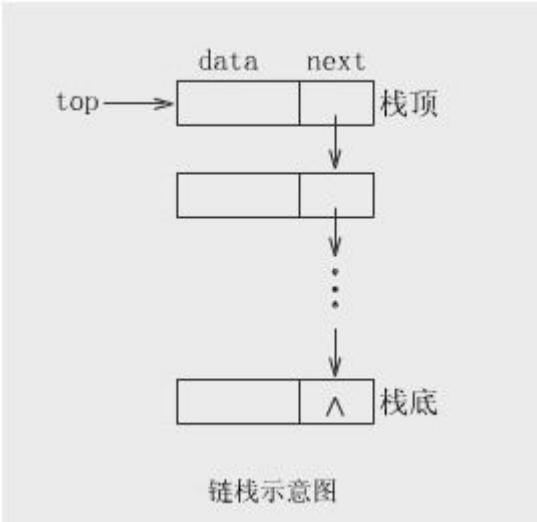
（3）消除递归不一定需要使用栈，因为对于**尾递归**（递归过程中的递归调用语句出现在过程结束之前）可以不设置栈，而直接通过改变过程中的参数值，利用循环结构代替递归调用。如下：

```
void process(int n)
{if(n>1)
   {printf(n);
    process(n-1);
   }
}
```

```
void process(int n)
 { int i;
   i = n;
   while(i>1)
         printf(i--);
 }
```

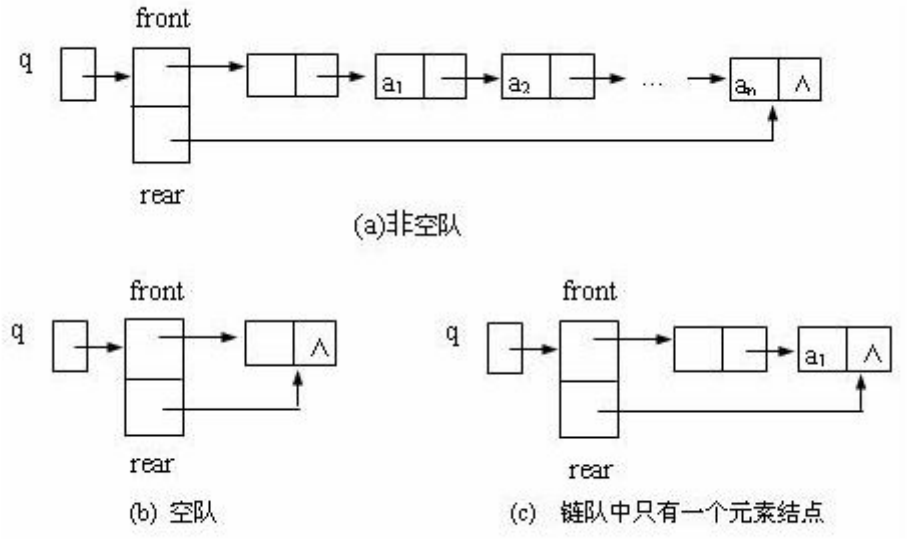
（4）递归程序的优点是程序结构简单、清晰、易证明其正确性。递归程序的缺点是执行中占内存空间较多，运行效率低。
（5）堆和栈的区别：堆和栈的区别可以用如下的比喻来看出：使用栈就象我们去饭馆里吃饭，只管点菜（发出申请）、付钱、和吃（使用），吃饱了就走，不必理会切菜、洗菜等准备工作和洗碗、刷锅等扫尾工作，他的好处是快捷，但是自由度小。使用堆就象是自己动手做喜欢吃的菜肴，比较麻烦，但是比较符合自己的口味，而且自由度大。
（6）任何一个递归过程都可以转换成非递归过程；（结论性的语句，机器不能识别递归）

16. 向一个栈顶指针为 Top 的链栈中插入一个 p 所指结点时，其操作步骤为：p->next = Top;Top = p;(链栈头指针为 Top，相当于无头结点的前插法)



链栈示意图

17. 在一个链队列中，若 f，r 分别为队首、队尾指针，则插入 s 所指结点的操作为：r->next = s;r = s;（头尾指针封装在一起的链队）



(a)非空队

(b) 空队　　(c) 链队中只有一个元素结点

18. 用不带头结点的单链表存储队列时，在进行删除运算时（D）
A. 仅修改头指针　B.仅修改尾指针　C. 头尾指针都要修改　D.头尾指针可能都要修改

分析：只有一个元素时，出队后队空，此时还要修改队尾指针。即当队列中最后一个元素被删后，队列尾指针也丢失了，因此需对队尾指针重新赋值（指向头结点）。

19. “循环队列通常用指针来实现队列的头尾相接。”表述错误，循环队列不存在也不需要指针，实现循环是通过条件转变实现。“循环队列也存在空间溢出问题。”表述正确，因为循环队列只是一个大小为 maxsize 的数组，空间有限，自然存在溢出问题。

20. 逆波兰式

（1）前缀表达式：前缀表达式就是不含括号的算术表达式，而且它是将运算符写在前面，操作数写在后面的表达式，也称为“波兰式”。例如，- 1 + 2 3，它等价于 1-(2+3)。

（2）中缀表达式：我们把平时所用的标准四则运算表达式，即“9+(3-1)*3+10/2"叫做中缀表达式。

（3）后缀表达式：不包含括号，运算符放在两个运算对象的后面，所有的计算按运算符出现的顺序，严格从左向右进行（不再考虑运算符的优先规则），如：(2 + 1) * 3 ， 即 2 1 + 3 *。也称逆波兰式（Reverse Polish notation，RPN，或逆波兰记法）。

将中缀表达式转换为逆波兰式的原则：遇到运算符，则准备入栈，如果准备入栈的运算符优先级比栈顶运算符优先级高，直接入栈。否则，取栈顶运算符出栈，直至将准备入栈的运算符压入栈中。另外，左括号‘(’在栈外时它的级别最高，而进栈后它的级别则最低了；右括号‘)’优先级最低，且不入栈，当遇到右括号‘)’时，一直需要对运算符出栈，并且做相应运算，直到遇到栈顶为左括号‘(’时，将其出栈。

例：中缀表达式为“3*2^(4+2*2-1*3)-5”的后缀表达式为“32422*+13*-^*5-”。

【第四章：字符串及线性结构的扩展】

21. 广义表的表头指第一个元素，而表尾指除表头外其余元素组成的子表。对于任意一个非空的列表，其表头可能是单元素也可能是列表，而表尾必为列表。

22. KMP 算法分析（见后页手写笔记）

23. KMP 算法较朴素的模式匹配算法有哪些改进？

答：KMP 算法的主要优点是主串指针不回溯。当主串很大不能一次读入内存且经常发生部分匹配时，KMP 算法的优点更为突出，

整个匹配过程中，对主串仅需从头至尾扫描一遍，这对处理从外设输入的庞大文件很有效，可以边读入边匹配，而无需回头重读。

【第五章：树结构】

24. 二叉树具有五种基本形态：a）空二叉树；b）只有根节点；c）根节点只有非空左子树；d）根节点只有非空右子树；e）根节点有非空的左右子树。

25. 结点的度：结点所拥有的子树的个数称为该结点的度。

26. 二叉树的性质：

（1）对于一棵非空的二叉树，若叶子结点数为 n_0，度数为 2 的结点数为 n_2，则有 $n_0 = n_2 + 1$

（2）具有 n 个结点的完全二叉树的深度 k 为取不大于 $\log_2 n$ 的最大整数并加 1

26. 二叉树的遍历：先序遍历为“根左右”，中序遍历为“左根右”，后序遍历为“左右根”；即先中后是由根节点被遍历的次序决定的，这是一个递归的过程。

27. 以非递归方法实现二叉树的遍历思想：在沿左子树深入时，深入一个结点入栈一个结点，若为先序遍历，则在入栈之前访问之；当沿左分支深入不下去时，则返回，即从栈中弹出前面压入的结点并到它的右子树去；若为中序遍历，则此时访问该结点，然后从该结点的右子树继续深入；若为后序遍历，则需要后序遍历它的右子树，之后再访问该结点，因此需要将此结点再次入栈，然后从该结点的右子树继续深入，与前面类同，仍为深入一个结点入栈一个结点，深入不下去再返回，直到第二次从栈里弹出该结点，即从右子树返回时，才访问它。

28. 理解递归以及非递归实现二叉树遍历！！！

29. 已知结点的先序序列和中序序列，能唯一确定一棵二叉树；由二叉树的后序序列和中序序列也可唯一确定一棵二叉树；如果只知道二叉树的先序序列和后序序列，则不能唯一确定一棵二叉树。

30. 在某种遍历序列中，指向直接前驱结点和指向直接后继结点的指针被称为线索，加了线索的二叉树称为线索二叉树。一个具有 n 个结点的二叉树若采用二叉链表存储结构，共有 2 n 个指针域，在 2n 个指针域中只有 n-1 个指针域是用来存储结点孩子的地址，而另外 n+1 个指针域存放的都是空指针，可以利用这些空指针域存放线索。

31. 树具有下面两个特点：

（1）树的根结点没有前驱结点，**除根结点之外的所有结点有且只有一个前驱结点**；

（2）树中所有结点可以有零个或多个后继结点。

31. 树的先序遍历与其转换的相应二叉树的先序遍历的结果序列相同；树的后序遍历与其转换的相应二叉树的中序遍历的结果序列相同。森林的先序遍历和后序遍历与所转换的二叉树的先序遍历和中序遍历的结果序列相同。

32. 要实现任意二叉树的后序遍历的非递归算法而不使用栈结构，最佳的方案是二叉树采用（B）存储结构。

A. 二叉链表　　B. 三叉链表　　C.广义表　　D.顺序表

分析：由于三叉链表有一个指针指向父结点，所以可以不适用栈也能很容易实现回溯。

33. 一棵有 124 个叶子结点的完全二叉树，最多有__248__个结点。

分析：利用二叉树性质求出度为 2 的结点数，另：**完全二叉树度为 1 的结点数要么为 0 要么为 1。**

34. 若一个森林具有 N 个结点，K 条边，且 N>K，则该森林中必有（C）课树。

A. K　　B. N　　C. N-K　　D. 1

分析：设此森林中有 m 棵树，每棵树具有的顶点数为v_i（1<=i<=m）,则：

$$v_1 + v_2 + ... + v_m = N \qquad ①$$

$$(v_1 - 1) + (v_2 - 1) + ... + (v_m - 1) = K \qquad ②$$

①-②得：m = N - K

35. 在中序线索二叉树中，每一个非空的线索均指向其祖先结点。因为在中序线索二叉树中，每一个非空的左线索指针指向该结点所在子树的根结点的父亲结点；每一个非空的右线索指针指向该结点的父亲结点。

36. 已知一棵度为 3 的树有 2 个度为 1 的结点，3 个度为 2 的结点，4 个度为 3 的结点，则该树叶子结点的个数为__12__。

分析：树中所有结点的度数之和等于分支数，且树的边数等于结点数减 1。所以，设叶子结点数目为 x，则有 4*3+3*2+2*1+x*0 = 4+3+2+x-1，解得 x=12。

37. 已知树的先序遍历和中序遍历序列，可以通过恢复其对应的二叉树，再变换成树。

38. 任何一棵二叉树的叶结点在先序、中序、后序遍历序列中的相对次序不发生改变。

39. 在二叉树中某结点所对应的森林中结点为叶子结点的条件是该结点在森林中既没有孩子也没有右兄弟结点。

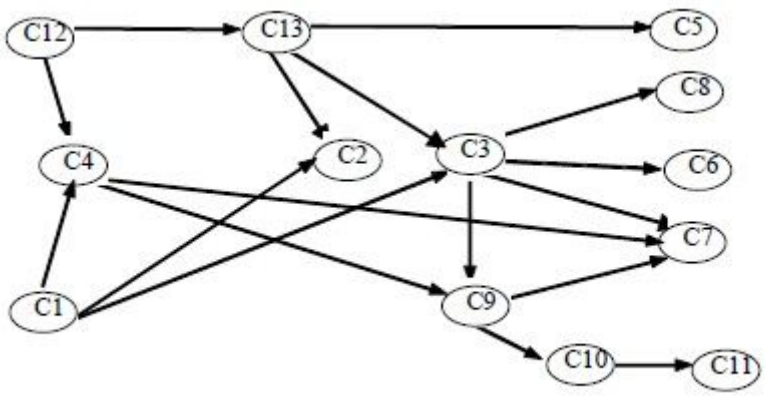
40. 在结点数多于 1 的哈弗曼树中不存在度为 1 的结点。

41. 若哈弗曼树中有 n 个叶结点，则树中共有 2n-1 个结点。

【第六章：图结构】

42. 十字链表是有向图的一种存储方法；邻接多重表主要用于存储无向图。

43. 拓扑有序序列：对一个有向无环图(Directed Acyclic Graph 简称 DAG)G 进行拓扑排序，是将 G 中所有顶点排成一个线性序列，使得图中任意一对顶点 u 和 v，若边(u,v)∈E(G)，则 u 在线性序列中出现在 v 之前。通常，这样的线性序列称为满足拓扑次序(Topological Order)的序列，简称拓扑序列。简单的说，由某个集合上的一个偏序得到该集合上的一个全序，这个操作称之为拓扑排序。如下图：一个 AOV（顶点活动网）网实例，C1、C12、C4、C13、C5、C2、C3、C9、C7、C10、C11、C6、C8 就是它的一个拓扑序列。



【第七章：查找】

44. 折半查找的时间复杂度为 O($\log_2 n$)。

45. 分块查找时，平均查找长度不仅和表的总长度 n 有关，而且和所分的子表个数 m 有关。可以证明，对于表长 n 确定的情况下，m 取$\sqrt{n}$时，ASL=$\sqrt{n}+1$达到最小值。

46. 二叉排序树或者是一棵空树，或者是具有下列性质的二叉树：

1）若左子树不空，则左子树上所有结点的值均小于根结点的值；若右子树不空，则右子树上所有结点的值均大于根结点的值。

2）左右子树也分别是二叉排序树。

对二叉排序树进行中序遍历，得到一个按关键码有序的序列。

最坏情况下，二叉排序树是一棵单枝树，这时树的高度最大，平均查找长度与顺序查找相同，为(n+1)/2。在最好情况下，二叉排序树的形态比较均匀，与折半查找的判定树类似，其平均查找长度大约为 O($\log_2 n$)。

47. 平衡二叉树，又称 AVL 树。它或者是一棵空树，或者是具有下列性质的二叉排序树：根结点的平衡因子（该结点的左子树高度与右子树高度之差）绝对值不超过 1；其左子树和右子树都是平衡二叉树。在平衡树上查找的时间复杂度为 O($\log_2 n$)。

48. 顺序查找算法的性能与查找表是否有序无关。

49. 在含有 n 个关键字的 m 阶 B 树中进行查找时，从根结点到关键字所在结点的路径上包含的结点数不超过 $1+\log_{\text{取不大于}m/2\text{的最大整数}}(n+1)/2$，所以，在一棵含有 n 个关键字的 m 阶 B 树中进行查找，至多读盘次数也是该数字。

50. 在非空 m 阶 B 树上，除根结点以外的所有其他非终端结点至少有（取不小于 m/2 的最小整数）棵子树。

51. 链表表示的有序表不能用折半查找法查找。

52. 最优二叉树不是 AVL 树，最优二叉树是静态树表，平衡二叉树是动态树表。

53. “在任意一棵非空二叉排序树中，删除某结点后又将其插入，则所得二叉排序树与删除前原二叉排序树相同。”该说法错误，因为除非被删除的结点是叶子结点，否则删除后再插入同一结点得到的二叉排序树与原来的二叉排序树不同。

54. 高度为 4 的 3 阶 B 树中，最多有__26__个关键字。

分析：第四层是叶子结点，每个结点两个关键字（关键字最多为阶数减一），2*1+2*3+2*3*3 = 26

55. m 路 B+树是一棵 m 路平衡索引树，其结点中关键字最多为 m 个，最少为（取不小于 m/2 的最小整数）个。

56. 将二叉排序树 T1 的先序遍历序列依次插入初始为空的树中，所得到的二叉排序树 T2 和 T1 的形态完全相同。

【第八章：排序】

57. 在数据结构中关键码指的是数据元素中能起标识作用的数据项，例如，书目信息中的登陆号和书名等。其中能起唯一标识作用的关键码称为“主关键码”，如登陆号；反之称为“次关键码”，如书名，作者名等。通常一个数据元素只有一个主码，但可以有多个次码。

58. 排序算法的稳定性通俗地讲就是能保证排序前 2 个相等的数其在序列的前后位置顺序和排序后它们两个的前后位置顺序相同。即如果 Ai = Aj，Ai 原来在位置前，排序后 Ai 还是要在 Aj 位置前。 选择排序、快速排序、希尔排序、堆排序是不稳定的排序算法。（速记：“快选堆希”）

59.

排序类别	排序名称
交换排序	冒泡排序、快速排序
插入排序	直接插入排序、希尔排序
选择排序	直接选择排序、堆排序
合并排序	归并排序
其他排序	基数排序、计数排序、桶排序

排序方法	时间复杂度			空间复杂度	稳定性	复杂性
	平均情况	最坏情况	最好情况			
直接插入排序	O（n^2）	O（n^2）	O（n）	O(1)	稳定	简单
折半插入排序	O（n^2）	O（n^2）	O（n）	O(1)	稳定	一般
希尔排序	O($n^{1.3\sim1.5}$)			O(1)	不稳定	较复杂
简单选择排序	O（n^2）	O（n^2）	O（n^2）	O(1)	不稳定	简单
堆排序	O($n\log_2 n$)	O($n\log_2 n$)	O($n\log_2 n$)	O(1)	不稳定	较复杂
冒泡排序	O（n^2）	O（n^2）	O（n）	O(1)	稳定	简单
快速排序	O($n\log_2 n$)	O（n^2）	O($n\log_2 n$)	O($\log_2 n$)	不稳定	较复杂
归并排序	O($n\log_2 n$)	O($n\log_2 n$)	O($n\log_2 n$)	O(n)	稳定	较复杂
基数排序	O(d(n+rd))	O(d(n+rd))	O(d(n+rd))	O(2rd+n)	稳定	较复杂

60. 基数排序不能对实数排序；归并排序稳定，且可以对实数排序。

61. 已知待排序的 n 个元素可分成 n/k 个组，每个组包含 k 个元素，且任一组内的各元素均分别大于前一组内所有元素和小于后一组内的所有元素，若采用基于比较的排序，其时间下届应为__O($n\log_2 k$)__。

分析：总共分为 n/k 组，每组有 k 个元素。由题意，只需对每一组排序即可。对于基于比较的排序，长度为 k 的数组，其时间下界为 O($k\log_2 k$)，总的时间下界为 n/k* $k\log_2 k$ = $n\log_2 k$ 。

62. 在用堆排序算法排序时，如果要进行升序排序，则需要采用“大顶堆”。因为堆排序算法是用数组来实现的，由于大顶堆中每次选出的最大元素将与最后一个尚未调整过的元素交换位置，故最终数组中的元素是按升序排列进行的。

63. 堆一定是一棵完全二叉树。因为堆在初始建立时是将给定序列按照层次遍历序列形式形成一棵完全二叉树后再调整成堆的，故必是二叉树。