

第 7 章 查 找

关注公众号【乘龙考研】
一手更新 稳定有保障

【考纲内容】

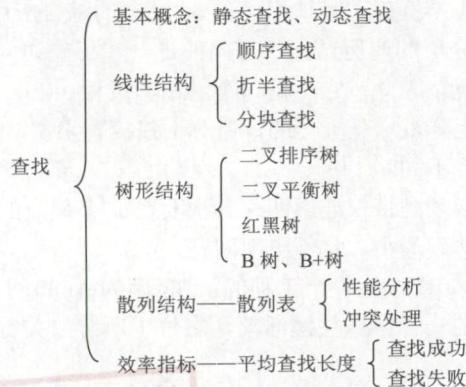
- (一) 查找的基本概念
- (二) 顺序查找法
- (三) 分块查找法
- (四) 折半查找法
- (五) 树型查找
 - 二叉搜索树；平衡二叉树；红黑树
- (六) B 树及其基本操作、B+树的基本概念
- (七) 散列(Hash)表
- (八) 查找算法的分析及应用

兑换后



看视频讲解

【知识框架】



【复习提示】

本章是考研命题的重点。对于折半查找，应掌握折半查找的过程、构造判定树、分析平均查找长度等。对于二叉排序树、二叉平衡树和红黑树，要了解它们的概念、性质和相关操作等。B 树和 B+树是本章的难点。对于 B 树，考研大纲要求掌握插入、删除和查找的操作过程；对于 B+树，仅要求了解其基本概念和性质。对于散列查找，应掌握散列表的构造、冲突处理方法（各种方法的处理过程）、查找成功和查找失败的平均查找长度、散列查找的特征和性能分析。

7.1 查找的基本概念

- 1) 查找。在数据集合中寻找满足某种条件的数据元素的过程称为查找。查找的结果一般分

为两种：一是查找成功，即在数据集合中找到了满足条件的数据元素；二是查找失败。

- 2) **查找表（查找结构）**。用于查找的数据集合称为查找表，它由同一类型的数据元素（或记录）组成，可以是一个数组或链表等数据类型。对查找表经常进行的操作一般有4种：①查询某个特定的数据元素是否在查找表中；②检索满足条件的某个特定的数据元素的各种属性；③在查找表中插入一个数据元素；④从查找表中删除某个数据元素。
- 3) **静态查找表**。若一个查找表的操作只涉及上述操作①和②，则无须动态地修改查找表，此类查找表称为静态查找表。与此对应，需要动态地插入或删除的查找表称为动态查找表。适合静态查找表的查找方法有顺序查找、折半查找、散列查找等；适合动态查找表的查找方法有二叉排序树的查找、散列查找等。
- 4) **关键字**。数据元素中唯一标识该元素的某个数据项的值，使用基于关键字的查找，查找结果应该是唯一的。例如，在由一个学生元素构成的数据集合中，学生元素中“学号”这一数据项的值唯一地标识一名学生。
- 5) **平均查找长度**。在查找过程中，一次查找的长度是指需要比较的关键字次数，而平均查找长度则是所有查找过程中进行关键字的比较次数的平均值，其数学定义为

$$ASL = \sum_{i=1}^n P_i C_i$$

式中， n 是查找表的长度； P_i 是查找第 i 个数据元素的概率，一般认为每个数据元素的查找概率相等，即 $P_i = 1/n$ ； C_i 是找到第 i 个数据元素所需进行的比较次数。平均查找长度是衡量查找算法效率的最主要的指标。

关注公众号【乘龙考研】
一手更新 稳定有保障

7.2 顺序查找和折半查找

7.2.1 顺序查找

顺序查找又称线性查找，它对顺序表和链表都是适用的。对于顺序表，可通过数组下标递增来顺序扫描每个元素；对于链表，可通过指针 `next` 来依次扫描每个元素。顺序查找通常分为对一般的无序线性表的顺序查找和对按关键字有序的线性表的顺序查找。下面分别进行讨论。

1. 一般线性表的顺序查找

作为一种最直观的查找方法，其基本思想是从线性表的一端开始，逐个检查关键字是否满足给定的条件。若查找到某个元素的关键字满足给定条件，则查找成功，返回该元素在线性表中的位置；若已经查找到表的另一端，但还没有查找到符合给定条件的元素，则返回查找失败的信息。下面给出其算法，主要是为了说明其中引入的“哨兵”的作用。

```

typedef struct{           //查找表的数据结构
    ELEMType *elem;      //元素存储空间基址，建表时按实际长度分配，0号单元留空
    int TableLen;         //表的长度
}SSTable;
int Search_Seq(SSTable ST,ELEMType key){
    ST.elem[0]=key;          //“哨兵”
    for(int i=ST.TableLen;ST.elem[i]!=key;--i);   //从后往前找
    return i; //若表中不存在关键字为key的元素，将查找到i为0时退出for循环
}

```

上述算法中，将 `ST.elem[0]` 称为哨兵，引入它的目的是使得 `Search_Seq` 内的循环不必判断数组是否会越界。算法从尾部开始查找，若找到 `ST.elem[i]==key` 则返回 `i` 值，查找成

功。否则一定在查找到 $ST.elem[0]==key$ 时跳出循环，此时返回的是 0，查找失败。在程序中引入“哨兵”并不是这个算法独有的，引入“哨兵”可以避免很多不必要的判断语句，从而提高程序效率。

对于有 n 个元素的表，给定值 key 与表中第 i 个元素相等，即定位第 i 个元素时，需进行 $n-i+1$ 次关键字的比较，即 $C_i = n-i+1$ 。查找成功时，顺序查找的平均长度为

$$ASL_{\text{成功}} = \sum_{i=1}^n P_i(n-i+1)$$

当每个元素的查找概率相等，即 $P_i = 1/n$ 时，有

$$ASL_{\text{成功}} = \sum_{i=1}^n P_i(n-i+1) = \frac{n+1}{2}$$

查找不到时，与表中各关键字的比较次数显然是 $n+1$ 次，即 $ASL_{\text{不成功}} = n+1$ 。

通常，查找表中记录的查找概率并不相等。若能预先得知每个记录的查找概率，则应先对记录的查找概率进行排序，使表中记录按查找概率由大至小重新排列。

综上所述，顺序查找的缺点是当 n 较大时，平均查找长度较大，效率低；优点是对数据元素的存储没有要求，顺序存储或链式存储皆可。对表中记录的有序性也没有要求，无论记录是否按关键字有序，均可应用。同时还需注意，对线性的链表只能进行顺序查找。

2. 有序表的顺序查找

若在查找之前就已经知道表是关键字有序的，则查找失败时可以不用再比较到表的另一端就能返回查找失败的信息，从而降低顺序查找失败的平均查找长度。

假设表 L 是按关键字从小到大排列的，查找的顺序是从前往后，待查找元素的关键字为 key ，当查找到第 i 个元素时，发现第 i 个元素对应的关键字小于 key ，但第 $i+1$ 个元素对应的关键字大于 key ，这时就可返回查找失败的信息，因为第 i 个元素之后的元素的关键字均大于 key ，所以表中不存在关键字为 key 的元素。

可以用如图 7.1 所示的判定树来描述有序线性表的查找过程。树中的圆形结点表示有序线性表中存在的元素；树中的矩形结点称为失败结点（若有 n 个结点，则相应地有 $n+1$ 个查找失败结点），它描述的是那些不在表中的数据值的集合。若查找到失败结点，则说明查找不到。

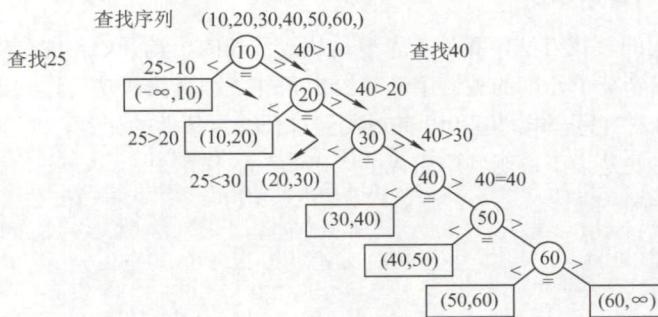


图 7.1 有序顺序表上的顺序查找判定树

在有序线性表的顺序查找中，查找成功的平均查找长度和一般线性表的顺序查找一样。查找失败时，查找指针一定走到了某个失败结点。这些失败结点是我们虚构的空结点，实际上是不存在的，所以到达失败结点时所查找的长度等于它上面的一个圆形结点的所在层数。查找不到的平均查找长度在相等查找概率的情形下为

$$\text{ASL}_{\text{不成功}} = \sum_{j=1}^n q_j(l_j - 1) = \frac{1+2+\dots+n+n}{n+1} = \frac{n}{2} + \frac{n}{n+1}$$

式中, q_j 是到达第 j 个失败结点的概率, 在相等查找概率的情形下, 它为 $1/(n+1)$; l_j 是第 j 个失败结点所在的层数。当 $n=6$ 时, $\text{ASL}_{\text{不成功}} = 6/2 + 6/7 = 3.86$, 比一般的顺序查找算法好一些。

注意, 有序线性表的顺序查找和后面的折半查找的思想是不一样的, 且有序线性表的顺序查找中的线性表可以是链式存储结构。

关注公众号【乘龙考研】
一手更新 稳定有保障

7.2.2 折半查找

折半查找又称二分查找, 它仅适用于有序的顺序表。

折半查找的基本思想: 首先将给定值 key 与表中中间位置的元素比较, 若相等, 则查找成功, 返回该元素的存储位置; 若不等, 则所需查找的元素只能在中间元素以外的前半部分或后半部分(例如, 在查找表升序排列时, 若给定值 key 大于中间元素, 则所查找的元素只可能在后半部分)。然后在缩小的范围内继续进行同样的查找, 如此重复, 直到找到为止, 或确定表中没有所需要查找的元素, 则查找不成功, 返回查找失败的信息。算法如下:

```
int Binary_Search(SSTable L, ElemenType key) {
    int low=0, high=L.TableLen-1, mid;
    while (low<=high) {
        mid=(low+high)/2;           //取中间位置
        if (L.elem[mid]==key)
            return mid;             //查找成功则返回所在位置
        else if (L.elem[mid]>key)
            high=mid-1;             //从前半部分继续查找
        else
            low=mid+1;              //从后半部分继续查找
    }
    return -1;                   //查找失败, 返回-1
}
```

例如, 已知 11 个元素的有序表 {7, 10, 13, 16, 19, 29, 32, 33, 37, 41, 43}, 要查找值为 11 和 32 的元素, 指针 low 和 $high$ 分别指向表的下界和上界, mid 则指向表的中间位置 $\lfloor (low+high)/2 \rfloor$ 。以下说明查找 11 的过程(查找 32 的过程请读者自行分析):

7	10	13	16	19	29	32	33	37	41	43
↑ low				↑ mid					↑ high	

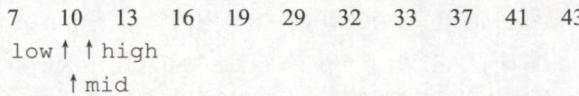
第一次查找, 将中间位置元素与 key 值比较。因为 $11 < 29$, 说明待查元素若存在, 则必在范围 $[low, mid-1]$ 内, 令指针 $high$ 指向位置 $mid-1$, $high=mid-1=5$, 重新求得 $mid=(1+5)/2=3$, 第二次的查找范围为 $[1, 5]$ 。

7	10	13	16	19	29	32	33	37	41	43
↑ low	↑ mid	↑ high								

第二次查找, 同样将中间位置元素与 key 值比较。因为 $11 < 13$, 说明待查元素若存在, 则必在范围 $[low, mid-1]$ 内, 令指针 $high$ 指向位置 $mid-1$, $high=mid-1=2$, 重新求得 $mid=(1+2)/2=1$, 第三次的查找范围为 $[1, 2]$ 。

7	10	13	16	19	29	32	33	37	41	43
low ↑	↑ high									
mid ↑										

第三次查找，将中间位置元素与 key 值比较。因为 $11 > 7$ ，说明待查元素若存在，则必在范围 $[mid+1, high]$ 内。令 $low=mid+1=2$, $mid=(2+2)/2=2$, 第四次的查找范围为 $[2, 2]$ 。



第四次查找，此时子表只含有一个元素，且 $10 \neq 11$ ，故表中不存在待查元素。

折半查找的过程可用图 7.2 所示的二叉树来描述，称为判定树。树中每个圆形结点表示一个记录，结点中的值为该记录的关键字值；树中最下面的叶结点都是方形的，它表示查找不成功的情况。从判定树可以看出，查找成功时的查找长度为从根结点到目的结点的路径上的结点数，而查找不成功时的查找长度为从根结点到对应失败结点的父结点的路径上的结点数；每个结点值均大于其左子结点值，且均小于其右子结点值。若有序序列有 n 个元素，则对应的判定树有 n 个圆形的非叶结点和 $n+1$ 个方形的叶结点。显然，判定树是一棵平衡二叉树。

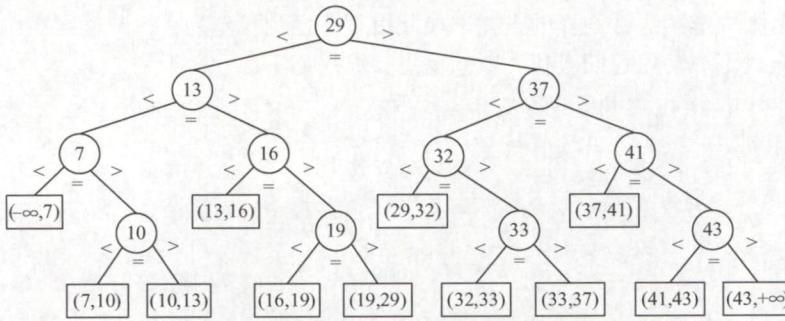


图 7.2 描述折半查找过程的判定树

由上述分析可知，用折半查找法查找到给定值的比较次数最多不会超过树的高度。在等概率查找时，查找成功的平均查找长度为

$$ASL = \frac{1}{n} \sum_{i=1}^n l_i = \frac{1}{n} (1 \times 1 + 2 \times 2 + \dots + h \times 2^{h-1}) = \frac{n+1}{n} \log_2(n+1) - 1 \approx \log_2(n+1) - 1$$

式中， h 是树的高度，并且元素个数为 n 时树高 $h = \lceil \log_2(n+1) \rceil$ 。所以，折半查找的时间复杂度为 $O(\log_2 n)$ ，平均情况下比顺序查找的效率高。

在图 7.2 所示的判定树中，在等概率情况下，查找成功（圆形结点）的 $ASL = (1 \times 1 + 2 \times 2 + 3 \times 4 + 4 \times 4)/11 = 3$ ，查找不到（方形结点）的 $ASL = (3 \times 4 + 4 \times 8)/12 = 11/3$ 。

因为折半查找需要方便地定位查找区域，所以它要求线性表必须具有随机存取的特性。因此，该查找法仅适合于顺序存储结构，不适合于链式存储结构，且要求元素按关键字有序排列。

7.2.3 分块查找

分块查找又称索引顺序查找，它吸取了顺序查找和折半查找各自的优点，既有动态结构，又适于快速查找。

分块查找的基本思想：将查找表分为若干子块。块内的元素可以无序，但块间的元素是有序的，即第一个块中的最大关键字小于第二个块中的所有记录的关键字，第二个块中的最大关键字小于第三个块中的所有记录的关键字，以此类推。再建立一个索引表，索引表中的每个元素含有各块的最大关键字和各块中的第一个元素的地址，索引表按关键字有序排列。

分块查找的过程分为两步：第一步是在索引表中确定待查记录所在的块，可以顺序查找或折

半查搜索索引表；第二步是在块内顺序查找。

例如，关键码集合为{88, 24, 72, 61, 21, 6, 32, 11, 8, 31, 22, 83, 78, 54}，按照关键码值 24, 54, 78, 88，分为 4 个块和索引表，如图 7.3 所示。

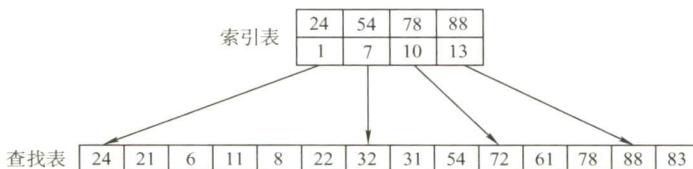


图 7.3 分块查找示意图

分块查找的平均查找长度为索引查找和块内查找的平均长度之和。设索引查找和块内查找的平均查找长度分别为 L_I , L_S ，则分块查找的平均查找长度为

$$ASL = L_I + L_S$$

将长度为 n 的查找表均匀地分为 b 块，每块有 s 个记录，在等概率情况下，若在块内和索引表中均采用顺序查找，则平均查找长度为

$$ASL = L_I + L_S = \frac{b+1}{2} + \frac{s+1}{2} = \frac{s^2 + 2s + n}{2s}$$

此时，若 $s = \sqrt{n}$ ，则平均查找长度取最小值 $\sqrt{n} + 1$ 。

7.2.4 本节试题精选

一、单项选择题

01. 顺序查找适合于存储结构为（ ）的线性表。
 - A. 顺序存储结构或链式存储结构
 - B. 散列存储结构
 - C. 索引存储结构
 - D. 压缩存储结构
02. 由 n 个数据元素组成的两个表：一个递增有序，一个无序。采用顺序查找算法，对有序表从头开始查找，发现当前元素已不小于待查元素时，停止查找，确定查找不成功，已知查找任意一个元素的概率是相同的，则在两种表中成功查找（ ）。
 - A. 平均时间后者小
 - B. 平均时间两者相同
 - C. 平均时间前者小
 - D. 无法确定
03. 对长度为 n 的有序单链表，若查找每个元素的概率相等，则顺序查找表中任意一个元素的查找成功的平均查找长度为（ ）。
 - A. $n/2$
 - B. $(n+1)/2$
 - C. $(n-1)/2$
 - D. $n/4$
04. 对长度为 3 的顺序表进行查找，若查找第一个元素的概率为 $1/2$ ，查找第二个元素的概率为 $1/3$ ，查找第三个元素的概率为 $1/6$ ，则查找任意一个元素的平均查找长度为（ ）。
 - A. $5/3$
 - B. 2
 - C. $7/3$
 - D. $4/3$
05. 下列关于二分查找的叙述中，正确的是（ ）。
 - A. 表必须有序，表可以顺序方式存储，也可以链表方式存储
 - B. 表必须有序且表中数据必须是整型、实型或字符型
 - C. 表必须有序，而且只能从小到大排列
 - D. 表必须有序，且表只能以顺序方式存储
06. 在一个顺序存储的有序线性表上查找一个数据时，既可以采用折半查找，也可以采用顺



- 序查找，但前者比后者的查找速度（ ）。
- 必然快
 - 取决于表是递增还是递减
 - 在大部分情况下要快
 - 必然不快
07. 折半查找过程所对应的判定树是一棵（ ）。
- 最小生成树
 - 平衡二叉树
 - 完全二叉树
 - 满二叉树
08. 折半查找和二叉排序树的时间性能（ ）。
- 相同
 - 有时不相同
 - 完全不同
 - 无法比较
09. 在有 11 个元素的有序表 $A[1, 2, \dots, 11]$ 中进行折半查找 ($\lfloor (low+high)/2 \rfloor$)，查找元素 $A[11]$ 时，被比较的元素下标依次是（ ）。
- 6, 8, 10, 11
 - 6, 9, 10, 11
 - 6, 7, 9, 11
 - 6, 8, 9, 11
10. 已知有序表(13, 18, 24, 35, 47, 50, 62, 83, 90, 115, 134)，当二分查找值为 90 的元素时，查找成功的元素比较次数为（ ）。
- 1
 - 2
 - 4
 - 6
11. 对表长为 n 的有序表进行折半查找，其判定树的高度为（ ）。
- $\lceil \log_2(n+1) \rceil$
 - $\lfloor \log_2(n+1) \rfloor - 1$
 - $\lceil \log_2 n \rceil$
 - $\lfloor \log_2 n \rfloor - 1$
12. 已知一个长度为 16 的顺序表，其元素按关键字有序排列，若采用折半查找算法查找一个不存在的元素，则比较的次数至少是（ ），至多是（ ）。
- 4
 - 5
 - 6
 - 7
13. 具有 12 个关键字的有序表中，对每个关键字的查找概率相同，折半查找算法查找成功的平均查找长度为（ ），折半查找查找失败的平均查找长度为（ ）。
- 37/12
 - 35/12
 - 39/13
 - 49/13
14. 采用分块查找时，数据的组织方式为（ ）。
- 数据分成若干块，每块内数据有序
 - 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
 - 数据分成若干块，每块内数据有序，每块内最大（或最小）的数据组成索引块
 - 数据分成若干块，每块（除最后一块外）中数据个数需相同
15. 对有 2500 个记录的索引顺序表（分块表）进行查找，最理想的块长为（ ）。
- 50
 - 125
 - 500
 - $\lceil \log_2 2500 \rceil$
16. 设顺序存储的某线性表共有 123 个元素，按分块查找的要求等分为 3 块。若对索引表采用顺序查找法来确定子块，且在确定的子块中也采用顺序查找法，则在等概率情况下，分块查找成功的平均查找长度为（ ）。
- 21
 - 23
 - 41
 - 62
17. 为提高查找效率，对有 65025 个元素的有序顺序表建立索引顺序结构，在最好情况下查找到表中已有元素最多需要执行（ ）次关键字比较。
- 10
 - 14
 - 16
 - 21
18. 【2010 统考真题】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 L 中不存在的元素，则关键字的比较次数最多是（ ）。
- 4
 - 5
 - 6
 - 7
19. 【2015 统考真题】下列选项中，不能构成折半查找中关键字比较序列的是（ ）。
- 500, 200, 450, 180
 - 500, 450, 200, 180

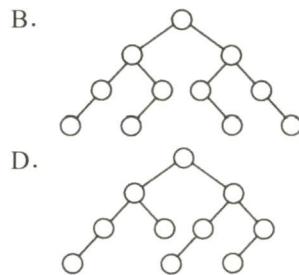
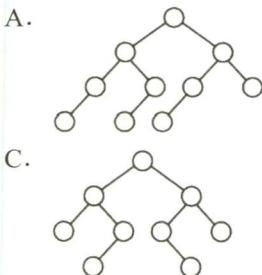
20. 【2016 统考真题】在有 n ($n > 1000$) 个元素的升序数组 A 中查找关键字 x。查找算法的伪代码如下所示。

```

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;

```

关注公众号【乘龙考研】
一手更新 稳定有保障



二、综合应用题

01. 若对有 n 个元素的有序顺序表和无序顺序表进行顺序查找，试就下列三种情况分别讨论两者在相等查找概率时的平均查找长度是否相同。

 - 1) 查找失败。
 - 2) 查找成功，且表中只有一个关键字等于给定值 k 的元素。
 - 3) 查找成功，且表中有若干关键字等于给定值 k 的元素，要求一次查找能找出所有元素。

02. 有序顺序表中的元素依次为 017, 094, 154, 170, 275, 503, 509, 512, 553, 612, 677, 765, 897, 908。
1) 试画出对其进行折半查找的判定树。
2) 若查找 275 或 684 的元素，将依次与表中的哪些元素比较？
3) 计算查找成功的平均查找长度和查找不成功的平均查找长度。

03. 类比二分查找算法，设计 k 分查找算法（ k 为大于 2 的整数）如下：首先检查 n/k 处（ n 为查找表的长度）的元素是否等于要搜索的值，然后检查 $2n/k$ 处的元素……这样，或者找到要查找的元素，或者把集合缩小到原来的 $1/k$ ，若未找到要查找的元素，则继续在得到的集合上进行 k 分查找；如此进行，直到找到要查找的元素或查找失败。试求查找成功和查找失败的时间复杂度。

04. 已知一个有序顺序表 $A[0..8n-1]$ 的表长为 $8n$ ，并且表中没有关键字相同的数据元素。假设按上述方法查找一个关键字值等于给定值 X 的数据元素：先在 $A[7], A[15], A[23], \dots, A[8k-1], \dots, A[8n-1]$ 中进行顺序查找，若查找成功，则算法报告成功位置并返回；若不成功，则当 $A[8K-1] < X < A[8 \times (k+1)-1]$ 时，可确定一个缩小的查找范围 $A[8k] \sim A[8 \times (k+1)-2]$ ，然后可在这个范围内执行折半查找。特殊情况：若 $X > A[8n-1]$ 的关键字，则查找失败。

- 1) 画出描述上述查找过程的判定树。
 - 2) 计算相等查找概率下查找成功的平均查找长度。
- 05.** 写出折半查找的递归算法。初始调用时, low 为 1, high 为 ST.length。
- 06.** 线性表中各结点的检索概率不等时, 可用如下策略提高顺序检索的效率: 若找到指定的结点, 则将该结点和其前驱结点(若存在)交换, 使得经常被检索的结点尽量位于表的前端。试设计在顺序结构和链式结构的线性表上实现上述策略的顺序检索算法。
- 07.** 【2013 统考真题】设包含 4 个数据元素的集合 $S = \{\text{'do'}, \text{'for'}, \text{'repeat'}, \text{'while'}\}$, 各元素的查找概率依次为 $p_1 = 0.35, p_2 = 0.15, p_3 = 0.15, p_4 = 0.35$ 。将 S 保存在一个长度为 4 的顺序表中, 采用折半查找法, 查找成功时的平均查找长度为 2.2。
 1) 若采用顺序存储结构保存 S , 且要求平均查找长度更短, 则元素应如何排列? 应使用何种查找方法? 查找成功时的平均查找长度是多少?
 2) 若采用链式存储结构保存 S , 且要求平均查找长度更短, 则元素应如何排列? 应使用何种查找方法? 查找成功时的平均查找长度是多少?

7.2.5 答案与解析

一、单项选择题

01. A

顺序查找是指从表的一端开始向另一端查找。它不要求查找表具有随机存取的特性, 可以是顺序存储结构或链式存储结构。

02. B

对于顺序查找, 不管线性表是有序的还是无序的, 成功查找第一个元素的比较次数为 1, 成功查找第二个元素的比较次数为 2, 以此类推, 即每个元素查找成功的比较次数只与其位置有关(与是否有序无关), 因此查找成功的平均时间两者相同。

03. B

在有序单链表上做顺序查找, 查找成功的平均查找长度与在无序顺序表或有序顺序表上做顺序查找的平均查找长度相同, 都是 $(n + 1)/2$ 。

04. A

在长度为 3 的顺序表中, 查找第一个元素的查找长度为 1, 查找第二个元素的查找长度为 2, 查找第三个元素的查找长度为 3, 故有

$$\text{ASL}_{\text{成功}} = \frac{1}{2} \times 1 + \frac{1}{3} \times 2 + \frac{1}{6} \times 3 = \frac{5}{3}$$

05. D

二分查找通过下标来定位中间位置元素, 故应采用顺序存储, 且二分查找能够进行的前提是查找表是有序的, 但具体是从大到小还是从小到大的顺序则不做要求。

06. C

折半查找的快体现在一般情况下, 在大部分情况下要快, 但是对于某些特殊情况, 顺序查找可能会快于折半查找。例如, 查找一个含 1000 个元素的有序表中的第一个元素时, 顺序查找的比较次数为 1 次, 而折半查找的比较次数却将近 10 次。

07. B

A 显然排除。对于选项 C, 考点精析示例中的判定树就不是完全二叉树。由选项 C 也可排除选项 D, 且满二叉树对结点数有要求。只可能选 B。事实上, 由折半查找的定义不难看出, 每次把一个数组从中间结点分割时, 总是把数组分为结点数相差最多不超过 1 的两个子数组, 从而使

得对应的判定树的两棵子树高度差的绝对值不超过 1，所以应是平衡二叉树。

08. B

折半查找的性能分析可以用二叉判定树来衡量，平均查找长度和最大查找长度都是 $O(\log_2 n)$ ；二叉排序树的查找性能与数据的输入顺序有关，最好情况下的平均查找长度与折半查找相同，但最坏情况即形成单支树时，其查找长度为 $O(n)$ 。

09. B

依据折半查找算法的思想，第一次 $\text{mid} = \lfloor (1+11)/2 \rfloor = 6$ ，第二次 $\text{mid} = \lfloor [(6+1)+11]/2 \rfloor = 9$ ，第三次 $\text{mid} = \lfloor [(9+1)+11]/2 \rfloor = 10$ ，第四次 $\text{mid} = 11$ 。

10. B

开始时 low 指向 13， high 指向 134， mid 指向 50，比较第一次 $90 > 50$ ，所以将 low 指向 62， high 指向 134， mid 指向 90，第二次比较找到 90。

11. A

对 n 个结点的判定树，设结点总数 $n = 2^h - 1$ ，则 $h = \lceil \log_2(n+1) \rceil$ 。

另解：特殊值代入法。直接将 $n=1$ 和 $n=2$ 的情况代入，仅有 A 满足要求。

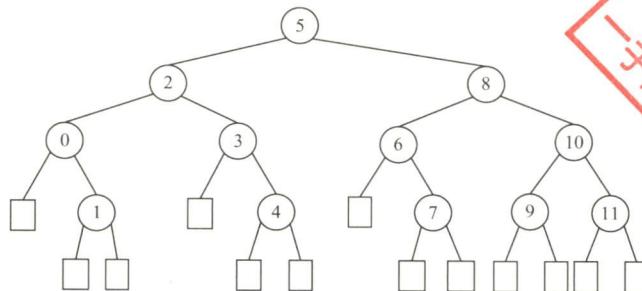
12. A、B

对于此类题，有两种做法：一种方法是，画出查找过程中构成的判定树，让最小的分支高度对应于最少的比较次数，让最大的分支高度对应于最多的比较次数，出现类似于长度为 15 的顺序表时，判定树刚好是一棵满树，此时最多比较次数与最少比较次数相等；另一种方法是，直接用公式求出最小的分支高度和最大分支高度，从前面的讲解不难看出最大分支高度为 $H = \lceil \log_2(n+1) \rceil = 5$ ，这对应的就是最多比较次数，然后由于判定树不是一棵满树，所以至少应该是 4（由判定树的各分支高度最多相差 1 得出）。

注意，若是求查找成功或查找失败的平均查找长度，则需要画出判定树进行求解。此外，对长度为 n 的有序表，采用折半查找时，查找成功和查找失败的最多比较次数相同，均为 $\lceil \log_2(n+1) \rceil$ 。

13. A、D

假设有序表中元素为 $A[0..11]$ ，不难画出对它进行折半查找的判定树如下图所示，圆圈是查找成功结点，方形是虚构的查找失败结点。从而可以求出查找成功的 $ASL = (1 + 2 \times 2 + 3 \times 4 + 4 \times 5)/12 = 37/12$ ，查找失败的 $ASL = (3 \times 3 + 4 \times 10)/13$ 。



注意：对于本类题目，应先根据所给 n 的值，画出如上图的折半查找判定树。另外，查找失败结点的 ASL 不是图中的方形结点，而是方形结点上一层的圆形结点。

14. B

通常情况下，在分块查找的结构中，不要求每个索引块中的元素个数都相等。

15. A

设块长为 b , 索引表包含 n/b 项, 索引表的 ASL = $(n/b + 1)/2$, 块内的 ASL = $(b + 1)/2$, 总 ASL = 索引表的 ASL + 块内的 ASL = $(b + n/b + 2)/2$, 其中对于 $b = n/b$, 由均值不等式知 $b = n/b$ 时有最小值, 此时 $b = \sqrt{n}$ 。则最理想块长为 $\sqrt{2500} = 50$ 。

16. B

根据公式 $ASL = L_1 + L_s = \frac{b+1}{2} + \frac{s+1}{2} = \frac{s^2 + 2s + n}{2s}$, 其中 $b = n/s, s = 123/3, n = 123$, 代入不难

得出 ASL 为 23。故选 B。另一方面, 可根据穷举法来一步步模拟。对于 A 块中的元素, 查找过程的第一步是先找到 A 块, 由于是顺序查找, 找到 A 块只需一步, 然后在 A 块中顺序查找。因此, A 块内各元素查找长度分别为 2, 3, 4, …, 42。对于 B 块, 采用类似的方法, 但查找到 B 块要比查找到 A 块多一步, 因此 B 块内各元素查找长度为 3, 4, 5, …, 43。同理, C 块中各个元素查找长度为 4, 5, 6, …, 44。所以平均查找长度为 $(2 + 3 + 4 + \dots + 42 + 3 + 4 + 5 + \dots + 43 + 4 + 5 + 6 + \dots + 44)/123 = 23$ 。

17. C

为使查找效率最高, 每个索引块的大小应是 $\sqrt{65025} = 255$, 为每个块建立索引, 则索引表中索引项的个数为 255。若对索引项和索引块内部都采用折半查找, 则查找效率最高, 为 $\lceil \log_2(255+1) \rceil + \lceil \log_2(255+1) \rceil = 16$ 。

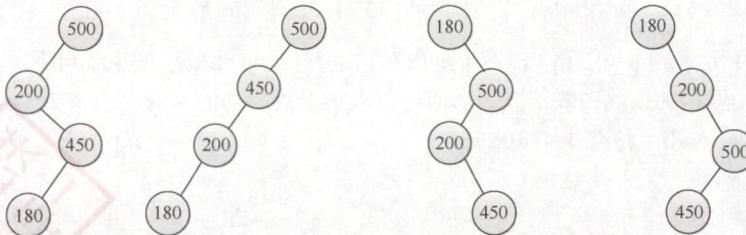
18. B

折半查找法在查找不成功时和给定值进行关键字的比较次数最多为树的高度, 即 $\lfloor \log_2 n \rfloor + 1$ 或 $\lceil \log_2(n+1) \rceil$ 。在本题中, $n = 16$, 故比较次数最多为 5。

注意: 在折半查找判定树中的方形结点是虚构的, 它并不计入比较的次数中。

19. A

如下图所示, 画出查找路径图, 因为折半查找的判定树是一棵二叉排序树, 因此看其是否满足二叉排序树的要求。



显然, 选项 A 的查找路径不满足。

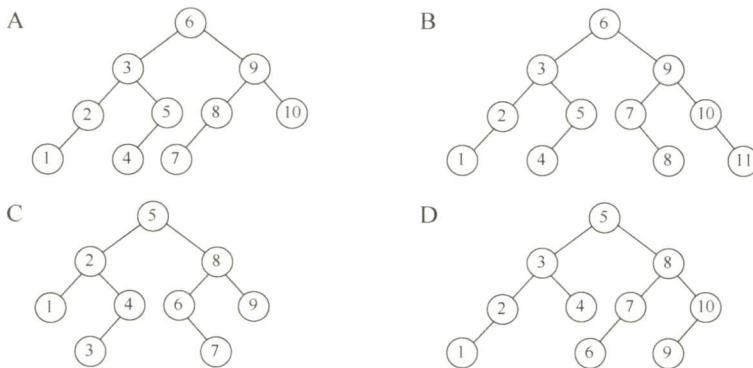
20. B

本题为送分题。

该程序采用跳跃式的顺序查找法查找升序数组中的 x。显然, x 越靠前, 比较次数越少。

21. A

折半查找判定树是一棵二叉排序树, 其中序序列是一个有序序列。可在树结点上依次填上相应的序号, 符合折半查找规则的树即为所求, 如下图所示。折半查找算法在选取中间结点时, 要么采用向上取整的方式, 要么采用向下取整的方式。B 选项, 4、5 相加除以 2 采用的是向上取整, 7、8 相加除以 2 采用的是向下取整, 矛盾。C 选项, 3、4 相加除以 2 采用的是向上取整, 6、7 相加除以 2 采用的是向下取整, 矛盾。D 选项, 1、10 相加除以 2 采用的是向下取整, 6、7 相加除以 2 采用的是向上取整, 矛盾。只有 A 符合折半查找规则。



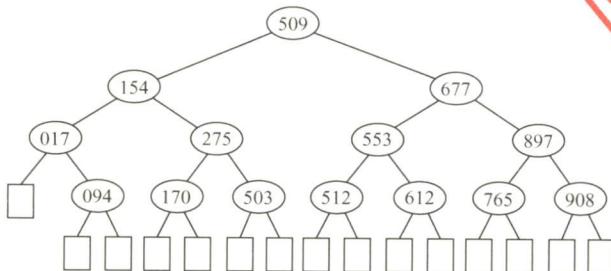
二、综合应用题

01. 【解答】

- 1) 平均查找长度不同。因为有序顺序表查找到其关键字值比要查找值大的元素时就停止查找，并报告失败信息，不必查找到表尾；而无序顺序表必须查找到表尾才能确定查找失败。
- 2) 平均查找长度相同。两者查找到表中元素的关键字值等于给定值时就停止查找。
- 3) 平均查找长度不同。有序顺序表中关键字相等的元素相继排列在一起，只要查找到第一个就可以连续查找到其他关键字相同的元素。而无序顺序表必须查找全部表中的元素才能找出相同关键字的元素，因此所需的时间不同。

02. 【解答】

- 1) 判定树如下图所示。



- 2) 若查找 275，依次与表中元素 509, 154, 275 进行比较，共比较 3 次。若查找 684，依次与表中元素 509, 677, 897, 765 进行比较，共比较 4 次。
- 3) 在查找成功时，会找到图中的某个圆形结点，其平均查找长度为

$$ASL_{\text{成功}} = \frac{1}{14} \sum_{i=1}^{14} C_i = \frac{1}{14} (1 + 2 \times 2 + 3 \times 4 + 4 \times 7) = \frac{45}{14}$$

在查找失败时，会找到图中的某个方形结点，但这个结点是虚构的，最后一次的比较元素为其父结点（圆形结点），故其平均查找长度为

$$ASL_{\text{不成功}} = \frac{1}{15} \sum_{i=0}^{14} C'_i = \frac{1}{15} (3 \times 1 + 4 \times 14) = \frac{59}{15}$$

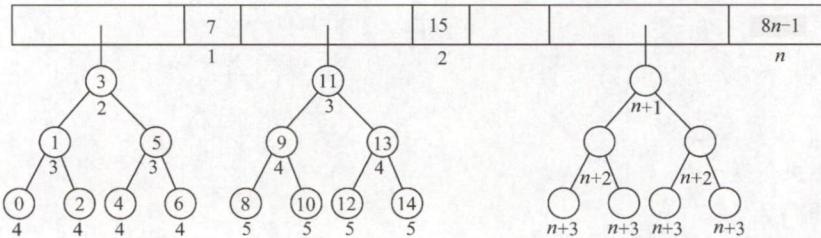
03. 【解答】

与二分查找类似， k 分查找法可用 k 叉树来描述。在最坏情况下，从第 2 层开始每层都比较 $k-1$ 次，具有 n 个结点的 k 叉树的深度为 $\lfloor \log_k n \rfloor + 1$ ，所以 k 分查找法在查找成功时和给定关键

字进行比较的次数至多为 $(k-1) \times \lfloor \log_k n \rfloor$, 即时间复杂度为 $O(\log_k n)$ 。同理, 查找不成功时, 和给定关键字进行比较的次数也至多为 $(k-1) \times \lfloor \log_k n \rfloor$, 故时间复杂度也为 $O(\log_k n)$ 。

04. 【解答】

- 1) 先在 $A[7], A[15], \dots, A[8n-1]$ 内顺序查找, 再在区间内折半查找。相应的判定树如下图所示。其中, 每个关键字下的数字为其查找成功时的关键字比较次数。



- 2) 等查找概率下, 平均每个关键字查找成功的概率为 $1/8n$; 0~7 之间的关键字, 顺序比较 1 次后, 进行折半查找, 查找成功的平均查找长度为 $2 + 3 \times 2 + 4 \times 4$; 8~15 之间的关键字, 先顺序比较 2 次后, 再进入折半查找; 以此类推, $8(n-1) \sim 8n-1$ 之间的关键字, 先顺序比较 n 次, 再进入折半查找, 如上图所示。故查找成功的平均查找长度为

$$\begin{aligned} ASL_{\text{成功}} &= \frac{1}{8n} \sum_{i=0}^{8n-1} C_i = \frac{1}{8n} \left(\sum_{i=1}^n i + \sum_{i=2}^{n+1} i + 2 \sum_{i=3}^{n+2} i + 4 \sum_{i=4}^{n+3} i \right) \\ &= \frac{1}{8n} \left(\sum_{i=1}^n (i + (i+1) + 2(i+2) + 4(i+3)) \right) \\ &= \frac{1}{8n} \sum_{i=1}^n (8i + 17) = \frac{1}{n} \sum_{i=1}^n i + \frac{17}{8} = \frac{n+1}{2} + \frac{17}{8} \end{aligned}$$

05. 【解答】

算法的基本思想: 根据查找的起始位置和终止位置, 将查找序列一分为二, 判断所查找的关键字在哪一部分, 然后用新的序列的起始位置和终止位置递归求解。

算法代码如下:

```

typedef struct{                               // 查找表的数据结构
    ElemType *elem;                         // 存储空间基址, 建表时按实际长度分配, 0 号留空
    int length;                            // 表的长度
} SSTable;

int BinSearchRec(SSTable ST, ElemType key, int low, int high){
    // 在有序表中递归折半查找其关键字为 key 的元素, 返回其在表中序号
    if (low > high)
        return 0;
    mid = (low + high) / 2;                  // 取中间位置
    if (key > ST.elem[mid])                 // 向后半部分查找
        BinSearchRec(ST, key, mid + 1, high);
    else if (key < ST.elem[mid])           // 向前半部分查找
        BinSearchRec(ST, key, low, mid - 1);
    else                                    // 查找成功
        return mid;
}

```

算法把规模为 n 的复杂问题经过多次递归调用转化为规模减半的子问题求解。时间复杂度为 $O(\log_2 n)$ ，算法中用到了一个递归工作栈，其规模与递归深度有关，也是 $O(\log_2 n)$ 。

06. 【解答】

算法的基本思想：检索时可先从表头开始向后顺序扫描，若找到指定的结点，则将该结点和其前趋结点（若存在）交换。采用顺序表存储结构的算法实现如下：

```
int SeqSrch(RcdType R[], ElemType k) {
    //顺序查找线性表，找到后和其前面的元素交换
    int i=0;
    while((R[i].key!=k) && (i<n)) {
        i++; //从前向后顺序查找指定结点
        if(i<n&&i>0){ //若找到，则交换
            temp=R[i];R[i]=R[i-1];R[i-1]=temp;
            return --i; //交换成功，返回交换后的位置
        }
        else return -1; //交换失败
    }
}
```

链表的实现方式请读者自行思考。注意，链表方式实现的基本思想与上述思想相似，但要注意用链表实现时，在交换两个结点之前需要保存指向一结点的指针。

07. 【解答】

1) 折半查找要求元素有序顺序存储，若各个元素的查找概率不同，折半查找的性能不一定优于顺序查找。采用顺序查找时，元素按其查找概率的降序排列时查找长度最小。

采用顺序存储结构，数据元素按其查找概率降序排列。采用顺序查找方法。

查找成功时的平均查找长度 = $0.35 \times 1 + 0.35 \times 2 + 0.15 \times 3 + 0.15 \times 4 = 2.1$ 。

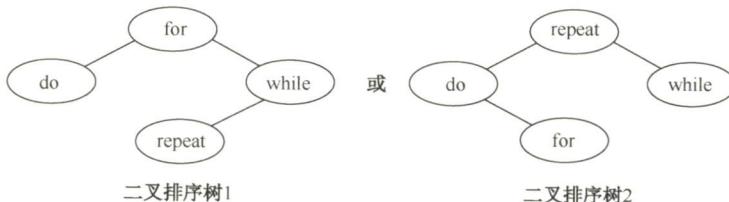
此时，显然查找长度比折半查找的更短。

2) 答案 1：采用链式存储结构时，只能采用顺序查找，其性能和顺序表一样，类似于上题。

数据元素按其查找概率降序排列，构成单链表。采用顺序查找方法。

查找成功时的平均查找长度 = $0.35 \times 1 + 0.35 \times 2 + 0.15 \times 3 + 0.15 \times 4 = 2.1$ 。

答案 2：还可以构造成二叉排序树的形式。采用二叉链表的存储结构，构造二叉排序树，元素的存储方式见下图。采用二叉排序树的查找方法。



查找成功时的平均查找长度 = $0.15 \times 1 + 0.35 \times 2 + 0.35 \times 2 + 0.15 \times 3 = 2.0$ 。

7.3 树型查找

7.3.1 二叉排序树 (BST)

构造一棵二叉排序树的目的并不是为了排序，而是为了提高查找、插入和删除关键字的速度，

二叉排序树这种非线性结构也有利于插入和删除的实现。

1. 二叉排序树的定义

二叉排序树（也称二叉查找树）或者是一棵空树，或者是具有下列特性的二叉树：

1) 若左子树非空，则左子树上所有结点的值均小于根结点的值。

2) 若右子树非空，则右子树上所有结点的值均大于根结点的值。

3) 左、右子树也分别是一棵二叉排序树。

根据二叉排序树的定义，左子树结点值 $<$ 根结点值 $<$ 右子树结点值，因此对二叉排序树进行中序遍历，可以得到一个递增的有序序列。例如，图 7.4 所示二叉排序树的中序遍历序列为 1 2 3 4 6 8。

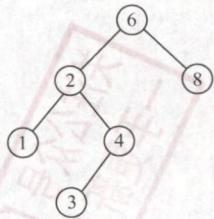


图 7.4 一棵二叉排序树

2. 二叉排序树的查找

二叉排序树的查找是从根结点开始，沿某个分支逐层向下比较的过程。若二叉排序树非空，先将给定值与根结点的关键字比较，若相等，则查找成功；若不等，如果小于根结点的关键字，则在根结点的左子树上查找，否则在根结点的右子树上查找。这显然是一个递归的过程。

二叉排序树的非递归查找算法：

```

BSTNode *BST_Search(BiTree T, ElemtType key) {
    while (T != NULL && key != T->data) { //若树空或等于根结点值，则结束循环
        if (key < T->data) T = T->lchild; //小于，则在左子树上查找
        else T = T->rchild; //大于，则在右子树上查找
    }
    return T;
}
  
```

例如，在图 7.4 中查找值为 4 的结点。首先 4 与根结点 6 比较。由于 4 小于 6，所以在根结点 6 的左子树中继续查找。由于 4 大于 2，所以在结点 2 的右子树中查找，查找成功。

同样，二叉排序树的查找也可用递归算法实现，递归算法比较简单，但执行效率较低。具体的代码实现，留给读者思考。

3. 二叉排序树的插入

二叉排序树作为一种动态树表，其特点是树的结构通常不是一次生成的，而是在查找过程中，当树中不存在关键字值等于给定值的结点时再进行插入的。

插入结点的过程如下：若原二叉排序树为空，则直接插入；否则，若关键字 k 小于根结点值，则插入到左子树，若关键字 k 大于根结点值，则插入到右子树。插入的结点一定是一个新添加的叶结点，且是查找失败时的查找路径上访问的最后一个结点的左孩子或右孩子。如图 7.5 所示在一棵二叉排序树中依次插入结点 28 和结点 58，虚线表示的边是其查找的路径。

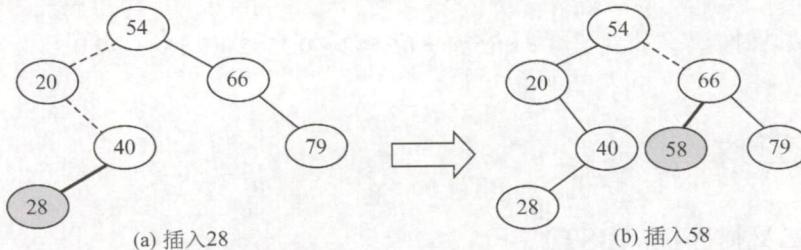


图 7.5 向二叉排序树中插入结点

二叉排序树插入操作的算法描述如下：

```
int BST_Insert(BiTree &T, KeyType k) {
    if (T==NULL) { //原树为空，新插入的记录为根结点
        T=(BiTree)malloc(sizeof(BSTNode));
        T->data=k;
        T->lchild=T->rchild=NULL;
        return 1; //返回 1，插入成功
    }
    else if (k==T->data) //树中存在相同关键字的结点，插入失败
        return 0;
    else if (k<T->data) //插入到 T 的左子树
        return BST_Insert(T->lchild, k);
    else //插入到 T 的右子树
        return BST_Insert(T->rchild, k);
}
```

4. 二叉排序树的构造

从一棵空树出发，依次输入元素，将它们插入二叉排序树中的合适位置。设查找的关键字序列为{45, 24, 53, 45, 12, 24}，则生成的二叉排序树如图 7.6 所示。

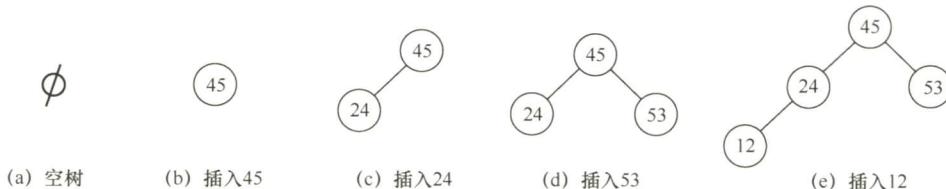


图 7.6 二叉排序树的构造过程

构造二叉排序树的算法描述如下：

```
void Creat_BST(BiTree &T, KeyType str[], int n) {
    T=NULL; //初始时 T 为空树
    int i=0;
    while(i<n) //依次将每个关键字插入到二叉排序树中
        BST_Insert(T, str[i]);
        i++;
}
```

5. 二叉排序树的删除

在二叉排序树中删除一个结点时，不能把以该结点为根的子树上的结点都删除，必须先把被删除结点从存储二叉排序树的链表上摘下，将因删除结点而断开的二叉链表重新链接起来，同时确保二叉排序树的性质不会丢失。删除操作的实现过程按 3 种情况来处理：

- ① 若被删除结点 z 是叶结点，则直接删除，不会破坏二叉排序树的性质。
- ② 若结点 z 只有一棵左子树或右子树，则让 z 的子树成为 z 父结点的子树，替代 z 的位置。
- ③ 若结点 z 有左、右两棵子树，则令 z 的直接后继（或直接前驱）替代 z，然后从二叉排序树中删去这个直接后继（或直接前驱），这样就转换成了第一或第二种情况。

图 7.7 显示了在 3 种情况下分别删除结点 45, 78, 78 的过程。

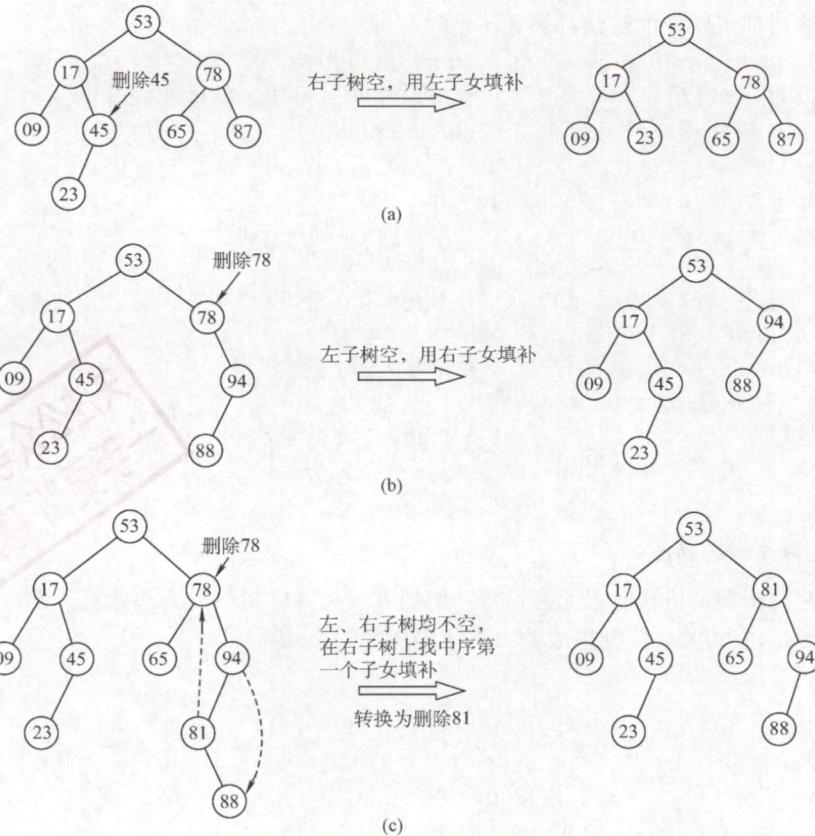


图 7.7 3 种情况下的删除过程

思考：若在二叉排序树中删除并插入某结点，得到的二叉排序树是否和原来的相同？

6. 二叉排序树的查找效率分析

二叉排序树的查找效率，主要取决于树的高度。若二叉排序树的左、右子树的高度之差的绝对值不超过 1（平衡二叉树，下一节），它的平均查找长度为 $O(\log_2 n)$ 。若二叉排序树是一个只有右（左）孩子的单支树（类似于有序的单链表），则其平均查找长度为 $O(n)$ 。

在最坏情况下，即构造二叉排序树的输入序列是有序的，则会形成一个倾斜的单支树，此时二叉排序树的性能显著变坏，树的高度也增加为元素个数 n ，如图 7.8(b)所示。

在等概率情况下，图 7.8(a)查找成功的平均查找长度为

$$ASL_a = (1 + 2 \times 2 + 3 \times 4 + 4 \times 3) / 10 = 2.9$$

而图 7.8(b)查找成功的平均查找长度为

$$ASL_b = (1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10) / 10 = 5.5$$

从查找过程看，二叉排序树与二分查找相似。就平均时间性能而言，二叉排序树上的查找和二分查找差不多。但二分查找的判定树唯一，而二叉排序树的查找不唯一，相同的关键字其插入顺序不同可能生成不同的二叉排序树，如图 7.8 所示。

就维护表的有序性而言，二叉排序树无须移动结点，只需修改指针即可完成插入和删除操作，平均执行时间为 $O(\log_2 n)$ 。二分查找的对象是有序顺序表，若有插入和删除结点的操作，所花的代价是 $O(n)$ 。当有序表是静态查找表时，宜用顺序表作为其存储结构，而采用二分查找实现其找操作；若有序表是动态查找表，则应选择二叉排序树作为其逻辑结构。

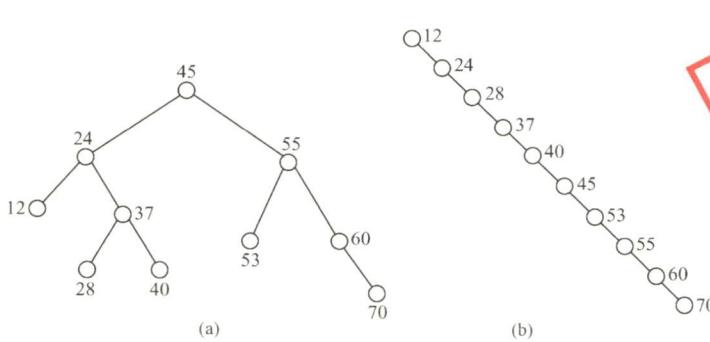


图 7.8 相同关键字组成的不同二叉排序树



7.3.2 平衡二叉树

1. 平衡二叉树的定义

为了避免树的高度增长过快，降低二叉排序树的性能，规定在插入和删除结点时，要保证任意结点的左、右子树高度差的绝对值不超过 1，将这样的二叉树称为平衡二叉树(Balanced Binary Tree)，或称 AVL 树。定义结点左子树与右子树的高度差为该结点的平衡因子，则平衡二叉树结点的平衡因子的值只可能是 -1、0 或 1。

因此，平衡二叉树可定义为或者是一棵空树，或者是具有下列性质的二叉树：它的左子树和右子树都是平衡二叉树，且左子树和右子树的高度差的绝对值不超过 1。图 7.9(a)所示是平衡二叉树，图 7.9(b)所示是不平衡的二叉树。结点中的值为该结点的平衡因子。

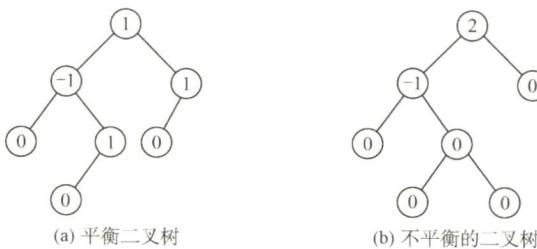


图 7.9 平衡二叉树和不平衡的二叉树

2. 平衡二叉树的插入

二叉排序树保证平衡的基本思想如下：每当在二叉排序树中插入（或删除）一个结点时，首先检查其插入路径上的结点是否因为此次操作而导致了不平衡。若导致了不平衡，则先找到插入路径上离插入结点最近的平衡因子的绝对值大于 1 的结点 A，再对以 A 为根的子树，在保持二叉排序树特性的前提下，调整各结点的位置关系，使之重新达到平衡。

注意：每次调整的对象都是最小不平衡子树，即以插入路径上离插入结点最近的平衡因子的绝对值大于 1 的结点作为根的子树。图 7.10 中的虚线框内为最小不平衡子树。

平衡二叉树的插入过程的前半部分与二叉排序树相同，但在新结点插入后，若造成查找路径上的某个结点不再平衡，则需要做出相应的调整。可将调整的规律归纳为下列 4 种情况：

- 1) LL 平衡旋转(右单旋转)。由于在结点 A 的左孩子(L)的左子树(L)上插入了新结点，A 的平衡因子由 1 增至 2，导致以 A 为根的子树失去平衡，需要一次向右的旋转操作。将 A 的左孩子 B 向右上旋转代替 A 成为根结点，将 A 结点向右下旋转成为 B 的右子树的根

结点，而 B 的原右子树则作为 A 结点的左子树。

如图 7.11 所示，结点旁的数值代表结点的平衡因子，而用方块表示相应结点的子树，下方数值代表该子树的高度。

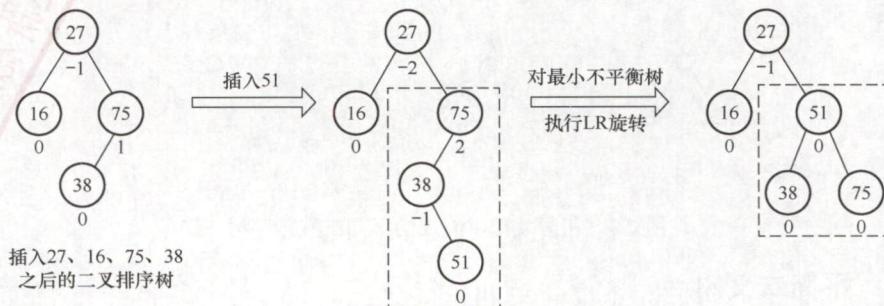


图 7.10 最小不平衡子树示意

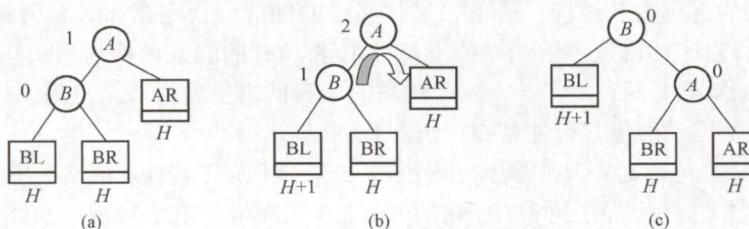


图 7.11 LL 平衡旋转

- 2) RR 平衡旋转 (左单旋转)。由于在结点 A 的右孩子 (R) 的右子树 (R) 上插入了新结点， A 的平衡因子由 -1 减至 -2 ，导致以 A 为根的子树失去平衡，需要一次向左的旋转操作。将 A 的右孩子 B 向左上旋转代替 A 成为根结点，将 A 结点向左下旋转成为 B 的左子树的根结点，而 B 的原左子树则作为 A 结点的右子树，如图 7.12 所示。

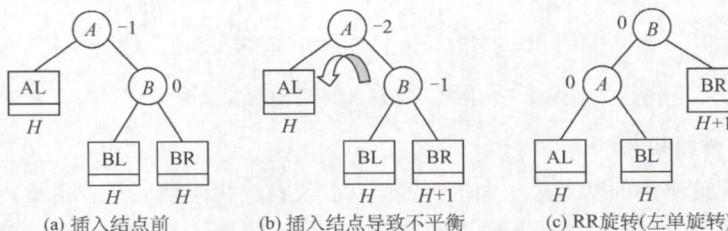


图 7.12 RR 平衡旋转

- 3) LR 平衡旋转 (先左后右双旋转)。由于在 A 的左孩子 (L) 的右子树 (R) 上插入新结点， A 的平衡因子由 1 增至 2 ，导致以 A 为根的子树失去平衡，需要进行两次旋转操作，先左旋转后右旋转。先将 A 结点的左孩子 B 的右子树的根结点 C 向左上旋转提升到 B 结点的位置，然后把该 C 结点向右上旋转提升到 A 结点的位置，如图 7.13 所示。
- 4) RL 平衡旋转 (先右后左双旋转)。由于在 A 的右孩子 (R) 的左子树 (L) 上插入新结点， A 的平衡因子由 -1 减至 -2 ，导致以 A 为根的子树失去平衡，需要进行两次旋转操作，先右旋转后左旋转。先将 A 结点的右孩子 B 的左子树的根结点 C 向右上旋转提升到 B 结点的位置，然后把该 C 结点向左上旋转提升到 A 结点的位置，如图 7.14 所示。

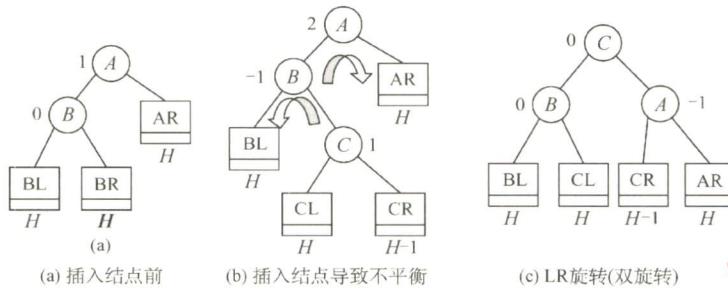


图 7.13 LR 平衡旋转

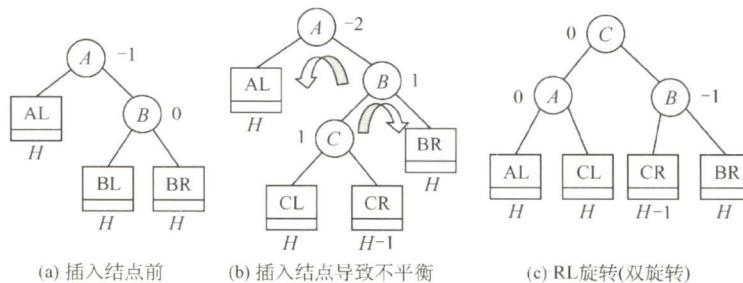


图 7.14 RL 平衡旋转

注意：LR 和 RL 旋转时，新结点究竟是插入 C 的左子树还是插入 C 的右子树不影响旋转过程，而图 7.13 和图 7.14 中以插入 C 的左子树中为例。

以关键字序列 {15, 3, 7, 10, 9, 8} 构造一棵平衡二叉树的过程为例，图 7.15(d)插入 7 后导致不平衡，最小不平衡子树的根为 15，插入位置为其左孩子的右子树，故执行 LR 旋转，先左后右双旋转，调整后的结果如图 7.15(e)所示。图 7.15(g)插入 9 后导致不平衡，最小不平衡子树的根为 15，插入位置为其左孩子的左子树，故执行 LL 旋转，右单旋转，调整后的结果如图 7.15(h)所示。图 7.15(i)插入 8 后导致不平衡，最小不平衡子树的根为 7，插入位置为其右孩子的左子树，故执行 RL 旋转，先右后左双旋转，调整后的结果如图 7.15(j)所示。

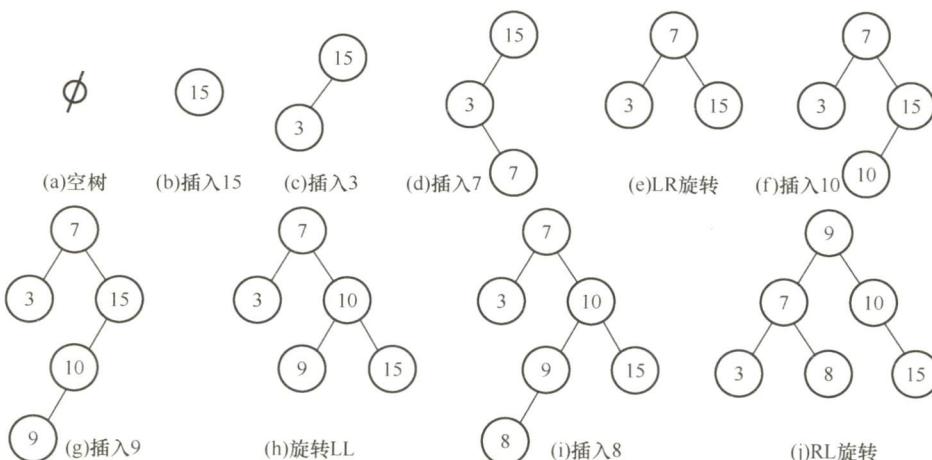


图 7.15 平衡二叉树的生成过程

3. 平衡二叉树的删除

与平衡二叉树的插入操作类似，以删除结点 w 为例来说明平衡二叉树删除操作的步骤：

- 1) 用二叉排序树的方法对结点 w 执行删除操作。
- 2) 若导致了不平衡，则从结点 w 开始向上回溯，找到第一个不平衡的结点 z（即最小不平衡子树）；y 为结点 z 的高度最高的孩子结点；x 是结点 y 的高度最高的孩子结点。
- 3) 然后对以 z 为根的子树进行平衡调整，其中 x、y 和 z 可能的位置有 4 种情况：
 - y 是 z 的左孩子，x 是 y 的左孩子（LL，右单旋转）；
 - y 是 z 的左孩子，x 是 y 的右孩子（LR，先左后右双旋转）；
 - y 是 z 的右孩子，x 是 y 的右孩子（RR，左单旋转）；
 - y 是 z 的右孩子，x 是 y 的左孩子（RL，先右后左双旋转）。

这四种情况与插入操作的调整方式一样。不同之处在于，插入操作仅需要对以 z 为根的子树进行平衡调整；而删除操作就不一样，先对以 z 为根的子树进行平衡调整，如果调整后子树的高度减 1，则可能需要对 z 的祖先结点进行平衡调整，甚至回溯到根结点（导致树高减 1）。

以删除图 7.16(a)的结点 32 为例，由于 32 为叶结点，直接删除即可，向上回溯找到第一个不平衡结点 44 (即 z)，z 的高度最高的孩子结点为 78 (y)，y 的高度最高的孩子结点为 50 (x)，满足 RL 情况，先右后左双旋转，调整后的结果如图 7.16(c)所示。

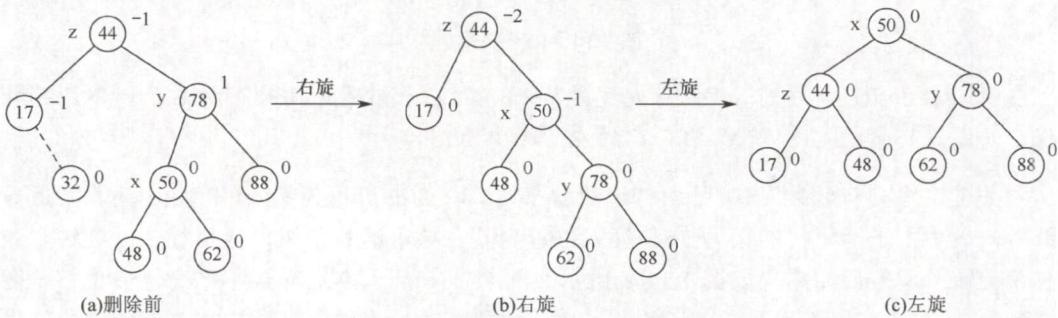


图 7.16 平衡二叉树的删除

4. 平衡二叉树的查找

在平衡二叉树上进行查找的过程与二叉排序树的相同。因此，在查找过程中，与给定值进行比较的关键字个数不超过树的深度。假设以 n_h 表示深度为 h 的平衡树中含有的最少结点数。显然，有 $n_0 = 0, n_1 = 1, n_2 = 2$ ，并且有 $n_h = n_{h-1} + n_{h-2} + 1$ 。可以证明，含有 n 个结点的平衡二叉树的最大深度为 $O(\log_2 n)$ ，因此平衡二叉树的平均查找长度为 $O(\log_2 n)$ ，如图 7.17 所示。

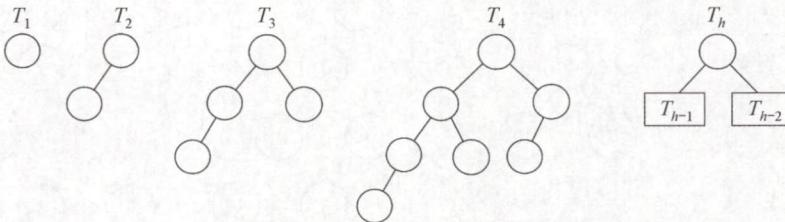


图 7.17 结点个数 n 最少的平衡二叉树

注意：该结论可用于求解给定结点数的平衡二叉树的查找所需的最多比较次数（或树的最大高度）。在含有 12 个结点的平衡二叉树中查找某个结点的最多比较次数是多少？

7.3.3 红黑树

1. 红黑树的定义

为了保持 AVL 树的平衡性，插入和删除操作后，非常频繁地调整全树整体拓扑结构，代价较大。为此在 AVL 树的平衡标准上进一步放宽条件，引入了红黑树的结构。

一棵红黑树是满足如下红黑性质的二叉排序树：

- ①每个结点或是红色，或是黑色的。
- ②根结点是黑色的。
- ③叶结点（虚构的外部结点、NULL 结点）都是黑色的。
- ④不存在两个相邻的红结点（即红结点的父结点和孩子结点均是黑色的）。
- ⑤对每个结点，从该结点到任意一个叶结点的简单路径上，所含黑结点的数量相同。

与折半查找树和 B 树类似，为了便于对红黑树的实现和理解，引入了 $n + 1$ 个外部叶结点，以保证红黑树中每个结点（内部结点）的左、右孩子均非空。图 7.18 所示是一棵红黑树。

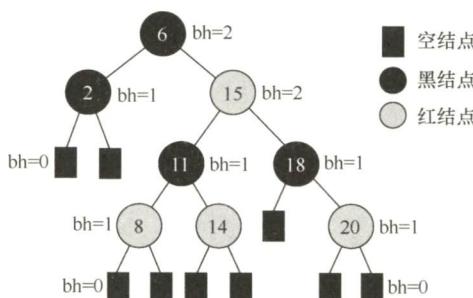


图 7.18 一棵红黑树

从某结点出发（不含该结点）到达一个叶结点的任意一个简单路径上的黑结点总数称为该结点的黑高（记为 bh ），黑高的概念是由性质⑤确定的。根结点的黑高称为红黑树的黑高。

结论 1：从根到叶结点的最长路径不大于最短路径的 2 倍。

由性质⑤，当从根到任意一个叶结点的简单路径最短时，这条路径必然全由黑结点构成。由性质④，当某条路径最长时，这条路径必然是由黑结点和红结点相间构成的，此时红结点和黑结点的数量相同。图 7.18 中的 6—2 和 6—15—18—20 就是这样的两条路径。

结论 2：有 n 个内部结点的红黑树的高度 $h \leq 2\log_2(n + 1)$ 。

证明：由结论 1 可知，从根到叶结点（不含叶结点）的任何一条简单路径上都至少有一半是黑结点，因此，根的黑高至少为 $h/2$ ，于是有 $n \geq 2^{h/2} - 1$ ，即可求得结论。

可见，红黑树的“适度平衡”，由 AVL 树的“高度平衡”，降低到“任意一个结点左右子树的高度，相差不超过 2 倍”，也降低了动态操作时调整的频率。对于一棵动态查找树，如果插入和删除操作比较少，查找操作比较多，采用 AVL 树比较合适，否则采用红黑树更合适。但由于维护这种高度平衡所付出的代价比获得的效益大得多，红黑树的实际应用更广泛，C++中的 map 和 set（Java 中的 TreeMap 和 TreeSet）就是用红黑树实现的。

2. 红黑树的插入

红黑树的插入过程和二叉查找树的插入过程基本类似，不同之处在于，在红黑树中插入新结点后需要进行调整（主要通过重新着色或旋转操作进行），以满足红黑树的性质。

结论 3：新插入红黑树中的结点初始着为红色。

假设新插入的结点初始着为黑色，那么这个结点所在的路径比其他路径多出一个黑结点（几乎每次插入都破坏性质⑤），调整起来也比较麻烦。如果插入的结点是红色的，此时所有路径上的黑结点数量不变，仅在出现连续两个红结点时才需要调整，而且这种调整也比较简单。

设结点 z 为新插入的结点。插入过程描述如下：

- 1) 用二叉查找树插入法插入，并将结点 z 着为红色。若结点 z 的父结点是黑色的，无须做任何调整，此时就是一棵标准的红黑树。
- 2) 如果结点 z 是根结点，将 z 着为黑色（树的黑高增 1），结束。
- 3) 如果结点 z 不是根结点，并且 z 的父结点 z.p 是红色的，则分为下面三种情况，区别在于 z 的叔结点 y 的颜色不同，因 z.p 是红色的，插入前的树是合法的，根据性质②和④，爷结点 z.p.p 必然存在且为黑色。性质④只在 z 和 z.p 之间被破坏了。

情况 1：z 的叔结点 y 是黑色的，且 z 是一个右孩子。

情况 2：z 的叔结点 y 是黑色的，且 z 是一个左孩子。

每棵子树 T_1 、 T_2 、 T_3 和 T_4 都有一个黑色根结点，且具有相同的黑高。

情况 1 (LR, 先左旋，再右旋)，即 z 是其爷结点的左孩子的右孩子。先做一次左旋将此情形转变为情况 2 (变为情况 2 后再做一次右旋)，左旋后 z 和父结点 z.p 交换位置。因为 z 和 z.p 都是红色的，所以左旋操作对结点的黑高和性质⑤都无影响。

情况 2 (LL, 右单旋)，即 z 是其爷结点的左孩子的左孩子。做一次右旋，并交换 z 的原父结点和原爷结点的颜色，就可以保持性质⑤，也不会改变树的黑高。这样，红黑树中也不再有连续两个红结点，结束。情况 1 和情况 2 的调整方式如图 7.19 所示。

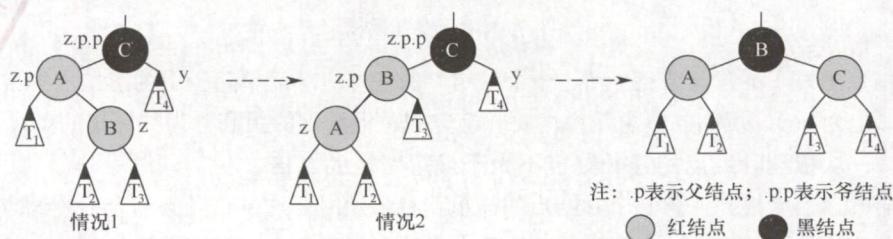


图 7.19 情况 1 和情况 2 的调整方式

若父结点 z.p 是爷结点 z.p.p 的右孩子，则还有两种对称的情况：RL（先右旋，再左旋）和 RR（左单旋），这里不再赘述。红黑树的调整方法和 AVL 树的调整方法有异曲同工之妙。

情况 3：如果 z 的叔结点 y 是红色。

情况 3 (z 是左孩子或右孩子无影响)，z 的父结点 z.p 和叔结点 y 都是红色的，因为爷结点 z.p.p 是黑色的，将 z.p 和 y 都着为黑色，将 z.p.p 着为红色，以在局部保持性质④和⑤。然后，把 z.p.p 作为新结点 z 来重复循环，指针 z 在树中上移两层。调整方式如图 7.20 所示。

若父结点 z.p 是爷结点 z.p.p 的右孩子，也还有两种对称的情况，不再赘述。

只要满足情况 3 的条件，就会不断循环，每次循环指针 z 都会上移两层，直到满足 2) (表示 z 上移到根结点) 或情况 1 或情况 2 的条件。

可能的疑问：虽然插入的初始位置一定是红黑树的某个叶结点，但因为在情况 3 中，结点 z 存在不断上升的可能，所以对于三种情况，结点 z 都有存在子树的可能。

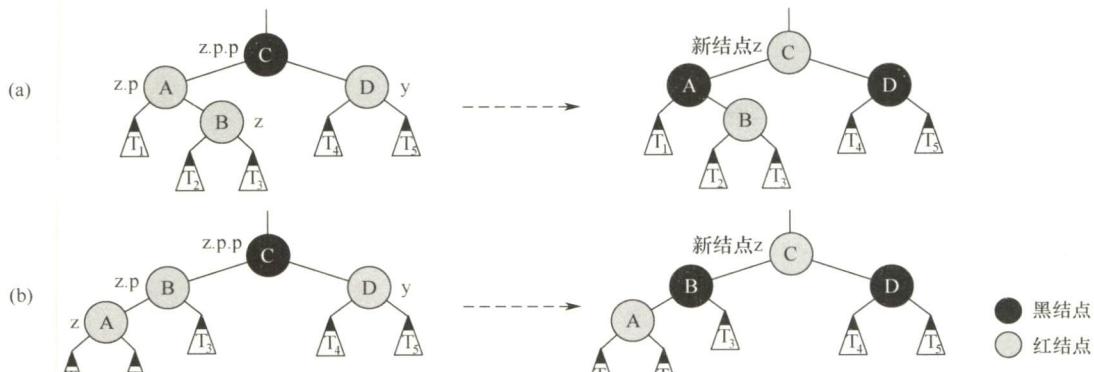


图 7.20 情况 3 的调整方式

以图 7.21(a)中的红黑树为例(虚线表示插入后的状态),先后插入 5、4 和 12 的过程如图 7.21 所示。插入 5, 为情况 3, 将 5 的父结点 3 和叔结点 10 着为黑色, 将 5 的爷结点变为红色, 此时因为 7 已是根, 故又重着为黑色, 树的黑高加 1, 结束。插入 4, 为情况 1 的对称情况 (RL), 此时特别注意虚构黑色空结点的存在, 先对 5 做右旋; 转变为情况 2 的对称情况 (RR), 交换 3 和 4 的颜色, 再对 3 做左旋, 结束。插入 12, 父结点是黑色的, 无须任何调整, 结束。

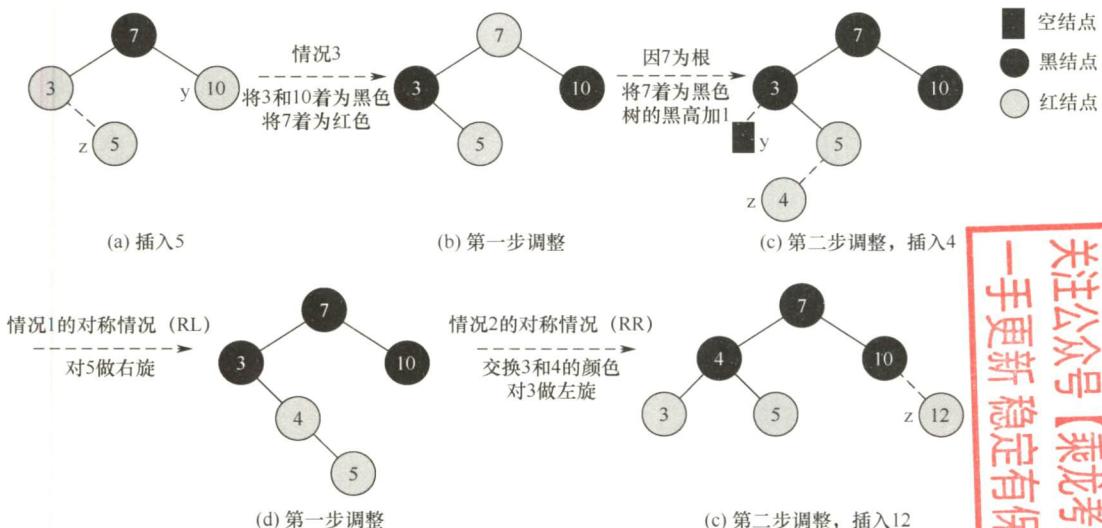


图 7.21 红黑树的插入过程

*3. 红黑树的删除^①

红黑树的插入操作容易导致连续的两个红结点, 破坏性质④。而删除操作容易造成子树黑高的变化 (删除黑结点会导致根结点到叶结点间的黑结点数量减少), 破坏性质⑤。

删除过程也是先执行二叉查找树的删除方法。若待删结点有两个孩子, 不能直接删除, 而要找到该结点的中序后继 (或前驱) 填补, 即右子树中最小的结点, 然后转换为删除该后继结点。由于后继结点至多只有一个孩子, 这样就转换为待删结点是终端结点或仅有一个孩子的情况。

最终, 删除一个结点有以下两种情况:

^① 本节难度较大, 考查的概率较低, 读者可根据自身情况决定学习的时机或是否学习。

- 待删结点只有右子树或左子树。
- 待删结点没有孩子。

1) 如果待删结点只有右子树或左子树, 则只有两种情况, 如图 7.22 所示。

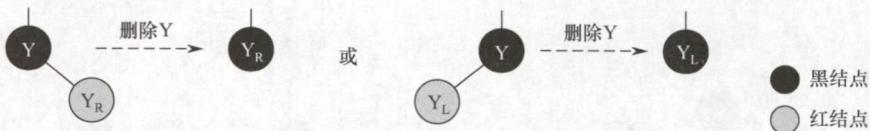


图 7.22 只有右子树或左子树的删除情况

只有这两种情况存在。子树只有一个结点, 且必然是红色, 否则会破坏性质⑤。

2) 如果待删结点没有孩子, 若该结点是红色的, 直接删除, 无须做任何调整。

3) 如果待删结点没有孩子, 并且该结点是黑色的。假设待删结点为 y , x 是用来替换 y 的结点 (注意, 当 y 是终端结点时, x 是黑色的 NULL 结点)。删除 y 后将导致先前包含 y 的任何路径上的黑结点数量减 1, 因此 y 的任何祖先都不再满足性质⑤, 简单的修正办法就是将替换 y 的结点 x 视为还有额外一重黑色, 定义为双黑结点。也就是说, 如果将任何包含结点 x 的路径上的黑结点数量加 1, 在此假设下, 性质⑤得到满足, 但破坏了性质①。于是, 删除操作的任务就转化为将双黑结点恢复为普通结点。

分为以下四种情况, 区别在于 x 的兄弟结点 w 及 w 的孩子结点的颜色不同。

情况 1: x 的兄弟结点 w 是红色的。

情况 1, w 必须有黑色左右孩子和父结点。交换 w 和父结点 $x.p$ 的颜色, 然后对 $x.p$ 做一次左旋, 而不会破坏红黑树的任何规则。现在, x 的新兄弟结点是旋转之前 w 的某个孩子结点, 其颜色为黑色, 这样, 就将情况 1 转换为情况 2、3 或 4 处理。调整方式如图 7.23 所示。

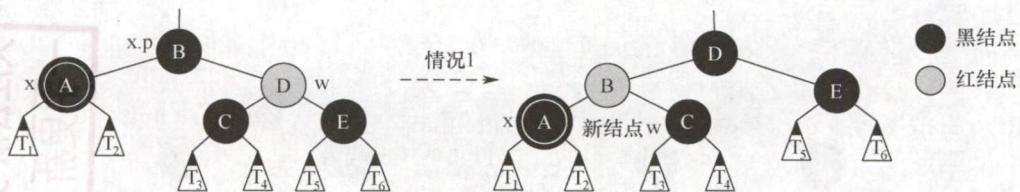


图 7.23 情况 1 的调整方式

情况 2: x 的兄弟结点 w 是黑色的, 且 w 的右孩子是红色的。

情况 3: x 的兄弟结点 w 是黑色的, w 的左孩子是红色的, w 的右孩子是黑色的。

情况 2 (RR, 左单旋), 即这个红结点是其爷结点的右孩子的右孩子。交换 w 和父结点 $x.p$ 的颜色, 把 w 的右孩子着为黑色, 并对 x 的父结点 $x.p$ 做一次左旋, 将 x 变为单重黑色, 此时不再破坏红黑树的任何性质, 结束。调整方式如图 7.24 所示。

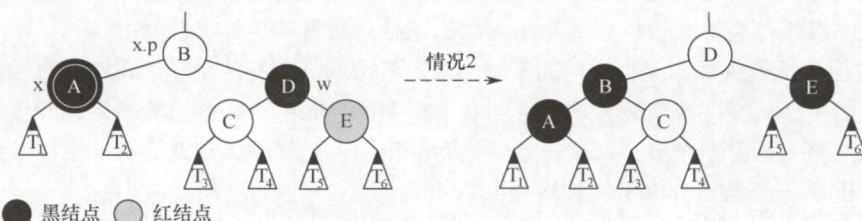


图 7.24 情况 2 的调整方式

情况3(RL, 先右旋, 再左旋), 即这个红结点是其父结点的右孩子的左孩子。交换w和其左孩子的颜色, 然后对w做一次右旋, 而不破坏红黑树的任何性质。现在, x的新兄弟结点w的右孩子是红色的, 这样就将情况3转换为了情况2。调整方式如图7.25所示。

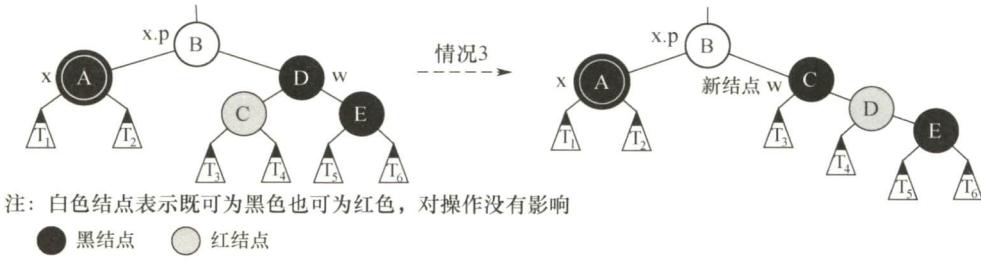


图 7.25 情况 3 的调整方式

情况4: x的兄弟结点w是黑色的, 且w的两个孩子结点都是黑色的。

情况4中, 因w也是黑色的, 故可从x和w上去掉一重黑色, 使得x只有一重黑色而w变为红色。为了补偿从x和w中去掉的一重黑色, 把x的父结点x.p额外着一层黑色, 以保持局部的黑高不变。通过将x.p作为新结点x来循环, x上升一层。如果是通过情况1进入情况4的, 因为原来的x.p是红色的, 将新结点x变为黑色, 终止循环, 结束。调整方式如图7.26所示。

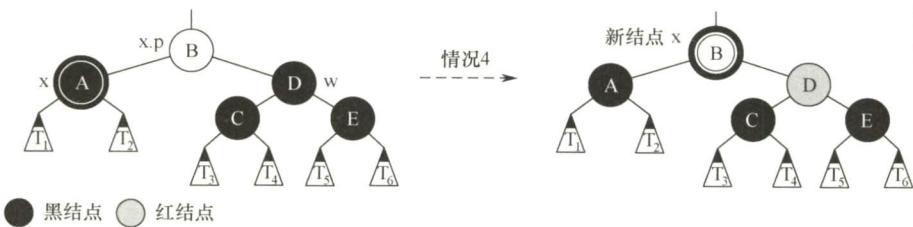


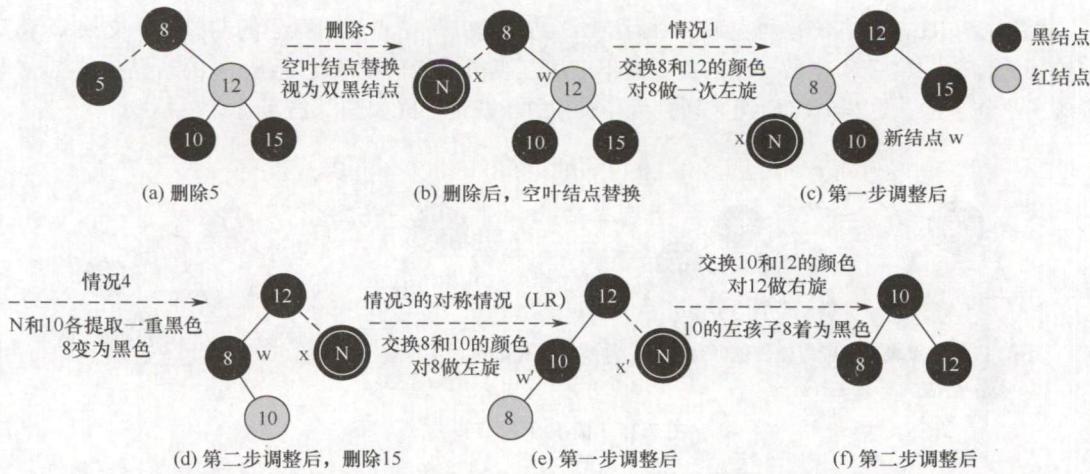
图 7.26 情况 4 的调整方式

若x是父结点x.p的右孩子, 则还有四种对称的情况, 处理方式类似, 不再赘述。

归纳总结: 在情况4中, 因x的兄弟结点w及左右孩子都是黑色, 可以从x和w中各提取一重黑色(以让x变为普通黑结点), 不会破坏性质④, 并把调整任务向上“推”给它们的父结点x.p。在情况1、2和3中, 因为x的兄弟结点w或w左右孩子中有红结点, 所以只能在x.p子树内用调整和重新着色的方式, 且不能改变x原根结点的颜色(否则向上可能破坏性质④)。情况1虽然可能会转换为情况4, 但因为新x的父结点x.p是红色的, 所以执行一次情况4就会结束。情况1、2和3在各执行常数次的颜色改变和至多3次旋转后便终止, 情况4是可能重复执行的唯一情况, 每执行一次指针x上升一层, 至多 $O(\log n)$ 次。

以图7.27(a)中的红黑树为例(虚线表示删除前的状态), 依次删除5和15的过程如图7.27所示。删除5, 用虚构的黑色NULL结点替换, 视为双黑NULL结点, 为情况1, 交换兄弟结点12和父结点8的颜色, 对8做一次左旋; 转变为情况4, 从双黑NULL结点和10中各提取一重黑色(提取后, 双黑NULL结点变为普通NULL结点, 图中省略, 10变为红色), 因原父结点8是红色, 故将8变为黑色, 结束。删除15, 为情况3的对称情况(LR), 交换8和10的颜色, 对8做左旋; 转变为情况2的对称情况(LL), 交换10和12的颜色(两者颜色一样, 无变化), 将10的左孩子8着为黑色, 对12做右旋, 结束。

关注公众号【乘龙考研】
一手更新 稳定有保障



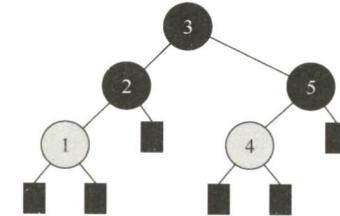
7.3.4 本节试题精选

一、单项选择题

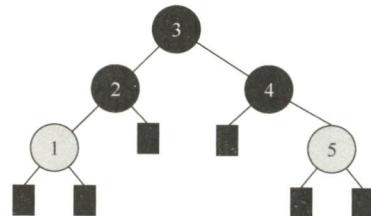
01. 对于二叉排序树，下面的说法中，() 是正确的。
 - A. 二叉排序树是动态树表，查找失败时插入新结点，会引起树的重新分裂和组合
 - B. 对二叉排序树进行层序遍历可得到有序序列
 - C. 用逐点插入法构造二叉排序树，若先后插入的关键字有序，二叉排序树的深度最大
 - D. 在二叉排序树中进行查找，关键字的比较次数不超过结点数的 $1/2$
02. 按() 遍历二叉排序树得到的序列是一个有序序列。
 - A. 先序
 - B. 中序
 - C. 后序
 - D. 层次
03. 在二叉排序树中进行查找的效率与() 有关。
 - A. 二叉排序树的深度
 - B. 二叉排序树的结点的个数
 - C. 被查找结点的度
 - D. 二叉排序树的存储结构
04. 在常用的描述二叉排序树的存储结构中，关键字值最大的结点()。
 - A. 左指针一定为空
 - B. 右指针一定为空
 - C. 左右指针均为空
 - D. 左右指针均不为空
05. 设二叉排序树中关键字由 1 到 1000 的整数构成，现要查找关键字为 363 的结点，下述关键字序列中，不可能是在二叉排序树上查找的序列是()。
 - A. 2, 252, 401, 398, 330, 344, 397, 363
 - B. 924, 220, 911, 244, 898, 258, 362, 363
 - C. 925, 202, 911, 240, 912, 245, 363
 - D. 2, 399, 387, 219, 266, 382, 381, 278, 363
06. 分别以下列序列构造二叉排序树，与用其他 3 个序列所构造的结果不同的是()。
 - A. (100, 80, 90, 60, 120, 110, 130)
 - B. (100, 120, 110, 130, 80, 60, 90)
 - C. (100, 60, 80, 90, 120, 110, 130)
 - D. (100, 80, 60, 90, 120, 130, 110)
07. 从空树开始，依次插入元素 52, 26, 14, 32, 71, 60, 93, 58, 24 和 41 后构成了一棵二叉排序树。在该树查找 60 要进行比较的次数为()。
 - A. 3
 - B. 4
 - C. 5
 - D. 6
08. 在含有 n 个结点的二叉排序树中查找某个关键字的结点时，最多进行() 次比较。
 - A. $n/2$
 - B. $\log_2 n$
 - C. $\log_2 n + 1$
 - D. n

09. 构造一棵具有 n 个结点的二叉排序树时，最理想情况下的深度为（ ）。
 A. $n/2$ B. n C. $\lfloor \log_2(n+1) \rfloor$ D. $\lceil \log_2(n+1) \rceil$
10. 含有 20 个结点的平衡二叉树的最大深度为（ ）。
 A. 4 B. 5 C. 6 D. 7
11. 具有 5 层结点的 AVL 至少有（ ）个结点。
 A. 10 B. 12 C. 15 D. 17
12. 下列关于红黑树的说法中，不正确的是（ ）。
 A. 一棵含有 n 个结点的红黑树的高度至多为 $2\log_2(n+1)$
 B. 如果一个结点是红色的，则它的父结点和孩子结点都是黑色的
 C. 从一个结点到其叶结点的所有简单路径上包含相同数量的黑结点
 D. 红黑树的查询效率一般要优于含有相同结点数的 AVL 树
13. 下列关于红黑树和 AVL 树的描述中，不正确的是（ ）。
 A. 两者都属于自平衡的二叉树
 B. 两者查找、插入、删除的时间复杂度都相同
 C. 红黑树插入和删除过程至多有 2 次旋转操作
 D. 红黑树的任意一个结点的左右子树高度（含叶结点）之比不超过 2
14. 下列关于红黑树的说法中，正确的是（ ）。
 A. 红黑树是一棵平衡二叉树
 B. 如果红黑树的所有结点都是黑色的，那么它一定是一棵满二叉树
 C. 红黑树的任何一个分支结点都有两个非空孩子结点
 D. 红黑树的子树也一定是红黑树
15. 将关键字 1, 2, 3, 4, 5, 6, 7 依次插入初始为空的红黑树 T ，则 T 中红结点的个数是（ ）。
 A. 1 B. 2 C. 3 D. 4
16. 将关键字 5, 4, 3, 2, 1 依次插入初始为空的红黑树 T ，则 T 的最终形态是（ ）。

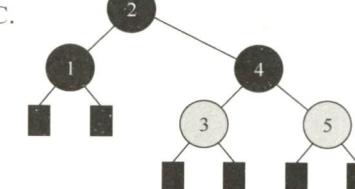
A.



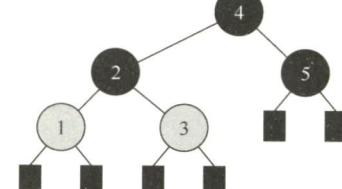
B.



C.



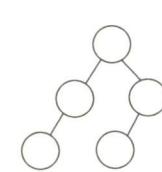
D.



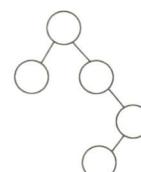
17. 【2009 统考真题】下列二叉排序树中，满足平衡二叉树定义的是（ ）。



A.



B.



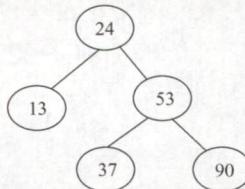
C.



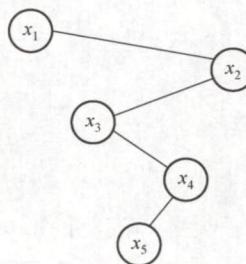
D.

关注公众号【乘龙考研】
一手更新 稳定有保障

18. 【2010 统考真题】在下图所示的平衡二叉树中插入关键字 48 后得到一棵新平衡二叉树，在新平衡二叉树中，关键字 37 所在结点的左、右子结点中保存的关键字分别是（ ）。



- A. 13, 48 B. 24, 48 C. 24, 53 D. 24, 90
19. 【2011 统考真题】对下列关键字序列，不可能构成某二叉排序树中一条查找路径的是（ ）。
- A. 95, 22, 91, 24, 94, 71 B. 92, 20, 91, 34, 88, 35
 C. 21, 89, 77, 29, 36, 38 D. 12, 25, 71, 68, 33, 34
20. 【2012 统考真题】若平衡二叉树的高度为 6，且所有非叶结点的平衡因子均为 1，则该平衡二叉树的结点总数为（ ）。
- A. 12 B. 20 C. 32 D. 33
21. 【2013 统考真题】在任意一棵非空二叉排序树 T_1 中，删除某结点 v 之后形成二叉排序树 T_2 ，再将 v 插入 T_2 形成二叉排序树 T_3 。下列关于 T_1 与 T_3 的叙述中，正确的是（ ）。
- I. 若 v 是 T_1 的叶结点，则 T_1 与 T_3 不同
 - II. 若 v 是 T_1 的叶结点，则 T_1 与 T_3 相同
 - III. 若 v 不是 T_1 的叶结点，则 T_1 与 T_3 不同
 - IV. 若 v 不是 T_1 的叶结点，则 T_1 与 T_3 相同
- A. 仅 I、III B. 仅 I、IV C. 仅 II、III D. 仅 II、IV
22. 【2013 统考真题】若将关键字 1, 2, 3, 4, 5, 6, 7 依次插入初始为空的平衡二叉树 T ，则 T 中平衡因子为 0 的分支结点的个数是（ ）。
- A. 0 B. 1 C. 2 D. 3
23. 【2015 统考真题】现有一棵无重复关键字的平衡二叉树（AVL），对其进行中序遍历可得到一个降序序列。下列关于该平衡二叉树的叙述中，正确的是（ ）。
- A. 根结点的度一定为 2 B. 树中最小元素一定是叶结点
 C. 最后插入的元素一定是叶结点 D. 树中最大元素一定是无左子树
24. 【2018 统考真题】已知二叉排序树如下图所示，元素之间应满足的大小关系是（ ）。



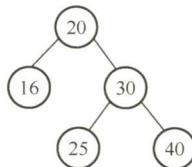
- A. $x_1 < x_2 < x_5$ B. $x_1 < x_4 < x_5$ C. $x_3 < x_5 < x_4$ D. $x_4 < x_3 < x_5$
25. 【2019 统考真题】在任意一棵非空平衡二叉树（AVL 树） T_1 中，删除某结点 v 之后形成平衡二叉树 T_2 ，再将 v 插入 T_2 形成平衡二叉树 T_3 。下列关于 T_1 与 T_3 的叙述中，正确的是（ ）。
- I. 若 v 是 T_1 的叶结点，则 T_1 与 T_3 可能不相同

- II. 若 v 不是 T_1 的叶结点，则 T_1 与 T_3 一定不相同
 III. 若 v 不是 T_1 的叶结点，则 T_1 与 T_3 一定相同
 A. 仅 I B. 仅 II C. 仅 I, II D. 仅 I, III

26. 【2020 统考真题】下列给定的关键字输入序列中，不能生成右边二叉排序树的是（ ）。

- A. 4, 5, 2, 1, 3 B. 4, 5, 1, 2, 3
 C. 4, 2, 5, 3, 1 D. 4, 2, 1, 3, 5

27. 【2021 统考真题】给定平衡二叉树如下图所示，插入关键字 23 后，根中的关键字是（ ）。



关注公众号【乘龙考研】
一手更新 稳定有保障

- A. 16 B. 20 C. 23 D. 25

二、综合应用题

01. 一棵二叉排序树按先序遍历得到的序列为(50, 38, 30, 45, 40, 48, 70, 60, 75, 80)，试画出该二叉排序树，并求出等概率下查找成功和查找失败的平均查找长度。
02. 按照序列(40, 72, 38, 35, 67, 51, 90, 8, 55, 21)建立一棵二叉排序树，画出该树，并求出在等概率的情况下，查找成功的平均查找长度。
03. 依次把结点(34, 23, 15, 98, 115, 28, 107)插入初始状态为空的平衡二叉排序树，使得在每次插入后保持该树仍然是平衡二叉树。请依次画出每次插入后所形成的平衡二叉排序树。
04. 给定一个关键字集合{25, 18, 34, 9, 14, 27, 42, 51, 38}，假定查找各关键字的概率相同，请画出其最佳二叉排序树。
05. 画出一棵二叉树，使得它既满足大根堆的要求又满足二叉排序树的要求。
06. 试编写一个算法，判断给定的二叉树是否是二叉排序树。
07. 设计一个算法，求出指定结点在给定二叉排序树中的层次。
08. 利用二叉树遍历的思想编写一个判断二叉树是否是平衡二叉树的算法。
09. 设计一个算法，求出给定二叉排序树中最小和最大的关键字。
10. 设计一个算法，从大到小输出二叉排序树中所有值不小于 k 的关键字。
11. 编写一个递归算法，在一棵有 n 个结点的、随机建立起来的二叉排序树上查找第 k ($1 \leq k \leq n$) 小的元素，并返回指向该结点的指针。要求算法的平均时间复杂度为 $O(\log_2 n)$ 。二叉排序树的每个结点中除 data, lchild, rchild 等数据成员外，增加一个 count 成员，保存以该结点为根的子树上的结点个数。

7.3.5 答案与解析

一、单项选择题

01. C

二叉排序树插入新结点时不会引起树的分裂组合。对二叉排序树进行中序遍历可得到有序序列。当插入的关键字有序时，二叉排序树会形成一个长链，此时深度最大。在此种情况下进行查找，有可能需要比较每个结点的关键字，超过总结点数的 $1/2$ 。

02. B

由二叉排序树的定义不难得出中序遍历二叉树得到的序列是一个有序序列。

03. A

二叉排序树的查找路径是自顶向下的，其平均查找长度主要取决于树的高度。

04. B

在二叉排序树的存储结构中，每个结点由三部分构成，其中左（或右）指针指向比该结点的关键字值小（或大）的结点。关键字值最大的结点位于二叉排序树的最右位置，因此它的右指针一定为空（有可能不是叶结点）。还可用反证法，若右指针不为空，则右指针上的关键字肯定比原关键字大，所以原关键字结点一定不是值最大的，与条件矛盾，所以右指针一定为空。

05. C

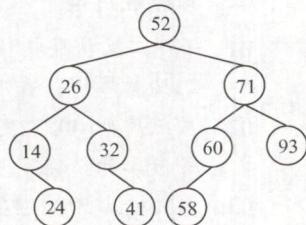
在二叉排序树上查找时，先与根结点值进行比较，若相同，则查找结束，否则根据比较结果，沿着左子树或右子树向下继续查找。根据二叉排序树的定义，有左子树结点值 \leq 根结点值 \leq 右子树结点值。C 序列中，比较 911 关键字后，应转向其左子树比较 240，左子树中不应出现比 911 更大的数值，但 240 竟有一个右孩子结点值为 912，所以不可能是正确的序列。

06. C

按照二叉排序树的构造方法，不难得到 A, B, D 序列的构造结果相同。

07. A

以第一个元素为根结点，依次将元素插入树，生成的二叉排序树如右图所示。进行查找时，先与根结点比较，然后根据比较结果，继续在左子树或右子树上进行查找。比较的结点依次为 52, 71, 60。

**08. D**

当输入序列是一个有序序列时，构造的二叉排序树是一个单支树，当查找一个不存在的关键字值或最后一个结点的关键字值时，需要 n 次比较。

09. D

当二叉排序树的叶结点全部都在相邻的两层内时，深度最小。理想情况是从第一层到倒数第二层为满二叉树。类比完全二叉树，可得深度为 $\lceil \log_2(n+1) \rceil$ 。

10. C

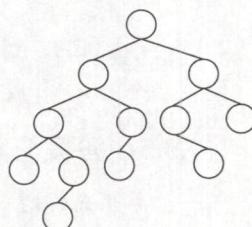
平衡二叉树结点数的递推公式为 $n_0 = 0$, $n_1 = 1$, $n_2 = 2$, $n_h = 1 + n_{h-1} + n_{h-2}$ (h 为平衡二叉树高度， n_h 为构造此高度的平衡二叉树所需的最少结点数)。通过递推公式可得，构造 5 层平衡二叉树至少需 12 个结点，构造 6 层至少需要 20 个结点。

11. B

设 n_h 表示高度为 h 的平衡二叉树中含有的最少结点数，则有 $n_1 = 1$, $n_2 = 2$, $n_h = n_{h-1} + n_{h-2} + 1$ ，由此求出 $n_5 = 12$ ，对应的 AVL 如右图所示。

12. D

选项 A、B 和 C 都是红黑树的性质。AVL 是高度平衡的二叉查找树，红黑树是适度平衡的二叉查找树，从这一点也可以看出 AVL 的查询效率往往更优。

**13. C**

自平衡的二叉排序树是指在插入和删除时能自动调整以保持其所定义的平衡性，红黑树和 AVL 都属于自平衡二叉树，选项 A 正确。在红黑树中删除结点时，情况 1 可能变为情况 2、3 或

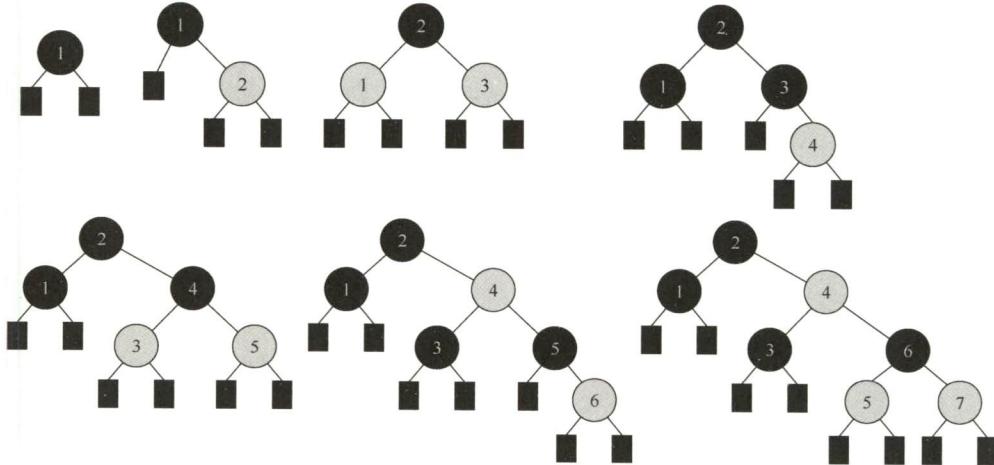
4, 情况2会变为情况3, 可能会出现旋转次数超过2次的情况, 选项C错误。

14. B

红黑树是一种特殊的二叉排序树, 平衡二叉树的左右子树的高度差小于或等于1, 红黑树显然不满足, 选项A错误。从根结点出发到所有叶结点的黑结点数是相同的, 若所有结点都是黑色, 则一定是满二叉树, 选项B正确。考虑某个黑结点, 它可以有一个空叶结点孩子和一个非空红结点孩子, 选项C错误。红黑树中可能存在红结点, 根结点为红结点的子树不是红黑树, 选项D错误。

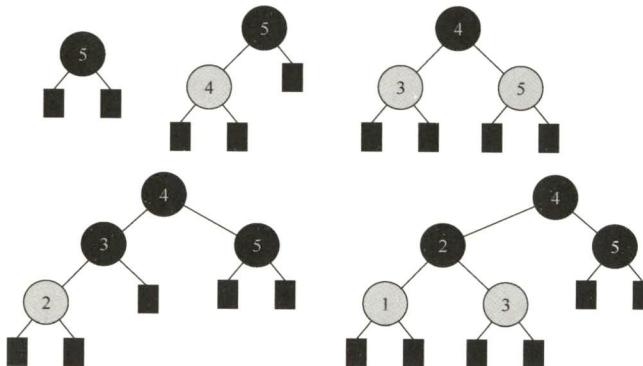
15. C

关键字1, 2, 3, 4, 5, 6, 7依次插入红黑树后的形态变化如下:



16. D

关键字5, 4, 3, 2, 1依次插入红黑树后的形态变化如下:



17. B

根据平衡二叉树的定义有, 任意结点的左、右子树高度差的绝对值不超过1。而其余3个答案均可以找到不满足条件的结点。

18. C

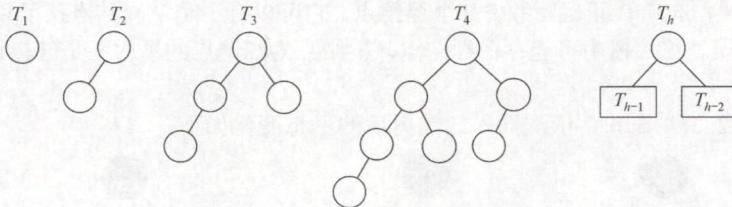
插入48后, 该二叉树根结点的平衡因子由-1变为-2, 失去平衡, 需进行两次旋转(先右旋后左旋)操作。

19. A

在二叉排序树中, 左子树结点值小于根结点, 右子树结点值大于根结点。在选项A中, 当查找到91后再向24查找, 说明这一条路径(左子树)之后查找的数都要比91小, 而后面却找到了94, 因此错误, 故选A。

20. B

所有非叶结点的平衡因子均为 1，即平衡二叉树满足平衡的最少结点情况，如下图所示。对于高度为 n 、左右子树的高度分别为 $n-1$ 和 $n-2$ 、所有非叶结点的平衡因子均为 1 的平衡二叉树，计算总结点数的公式为 $C_n = C_{n-1} + C_{n-2} + 1$, $C_1 = 1$, $C_2 = 2$, $C_3 = 2 + 1 + 1 = 4$, 可推出 $C_6 = 20$ 。



画图法：先画出 T_1 和 T_2 ；然后新建一个根结点，连接 T_2 、 T_1 构成 T_3 ；新建一个根结点，连接 T_3 、 T_2 构成 T_4 ……直到画出 T_6 ，可知 T_6 的结点数为 20。

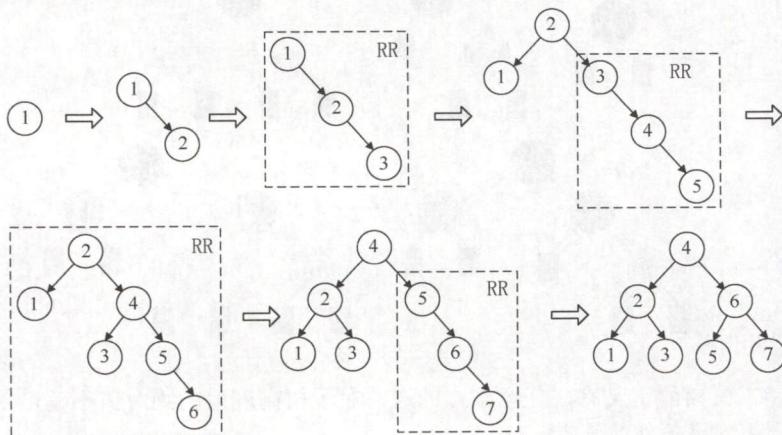
排除法：对于 A，高度为 6、结点数为 12 的树怎么也无法达到平衡。对于 C，结点较多时，考虑较极端的情形，即第 6 层只有最左叶子的完全二叉树刚好有 32 个结点，虽然满足平衡的条件，但显然再删去部分结点依然不影响平衡，不是最少结点的情况。同理选项 D 错误。只能选择选项 B。

21. C

在一棵二叉排序树中删除一个结点后，再将此结点插入二叉排序树，如果删除的是叶结点，则插入结点后的二叉排序树与删除结点之前的相同。如果删除的不是叶结点，则插入结点后的二叉排序树会发生变化，与删除结点之前的不相同。

22. D

利用 7 个关键字构建平衡二叉树 T ，平衡因子为 0 的分支结点个数为 3，构建的平衡二叉树及构造与调整过程如下图所示。

**23. D**

大多数教材将平衡二叉树定义为一种高度平衡的二叉排序树，二叉排序树的中序序列是一个升序序列，而题意正好相反。由此可知，命题老师认为平衡二叉树仅为一棵满足高度平衡的二叉树，不一定是二叉排序树。只有两个结点的平衡二叉树的根结点的度为 1，选项 A 错误。中序遍历后得到一个降序序列（与二叉排序树正好相反），树中最大元素一定无右子树（可能有左子树），这与二叉排序树也正好相反，也因此不一定是叶结点，选项 B 错误。最后插入的结点可能会导致平衡调整，而不一定是叶结点，选项 C 错误。

24. C

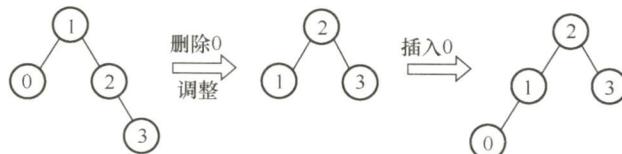
根据二叉排序树的特性：中序遍历（LNR）得到的是一个递增序列。图中二叉排序树的中序遍历序列为 x_1, x_3, x_5, x_4, x_2 ，可知 $x_3 < x_5 < x_4$ 。

25. A

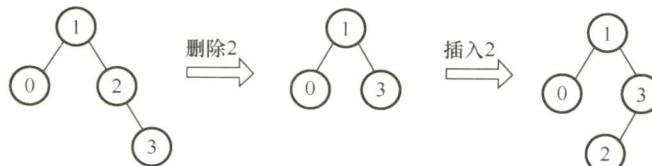
在非空平衡二叉树中插入结点，在失去平衡调整前，一定插入在叶结点的位置。

若删除的是 T_1 的叶结点，则删除后平衡二叉树可能不会失去平衡，即不会发生调整，再插入此结点得到的二叉平衡树 T_1 与 T_3 相同；若删除后平衡二叉树失去平衡而发生调整，再插入结点得到的二叉平衡树 T_3 与 T_1 可能不同。说法 I 正确。例如，如下图所示，删除结点 0，平衡二叉树失衡调整，再插入结点 0 后，平衡二叉树和以前不同。

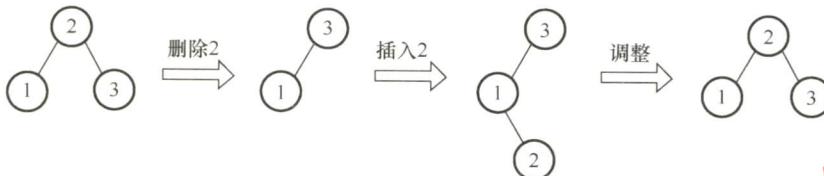
对于比较绝对的说法 II 和 III，通常只需举出反例即可。



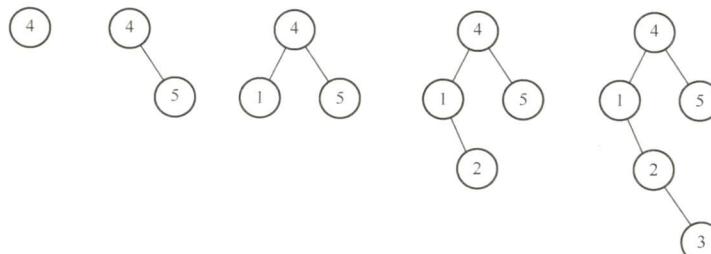
若删除的是 T_1 的非叶结点，且删除和插入操作均没有导致平衡二叉树的调整（这时可以首先想到删除的结点只有一个孩子的情况），则该结点从非叶结点变成了叶结点， T_1 与 T_3 显然不同。例如，如下图所示，删除结点 2，用右孩子结点 3 填补，再插入结点 2，平衡二叉树和以前不同。



若删除的是 T_1 的非叶结点，且删除和插入操作后导致了平衡二叉树的调整，则该结点有可能通过旋转后继续变成非叶结点， T_1 与 T_3 相同。例如，如下图所示，删除结点 2，用右孩子结点 3 填补，再插入结点 2，平衡二叉树失衡调整，调整后的平衡二叉树和以前相同。

**26. B**

每个选项都逐一验证，选项 B 生成二叉排序树的过程如下：



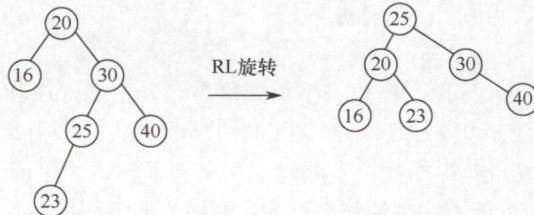
显然选项 B 错误。

27. D

关键字 23 的插入位置为 25 的左孩子，此时破坏了平衡的性质，需要对平衡二叉树进行调整。

关注公众号【乘龙考研】
一手更新 稳定有保障

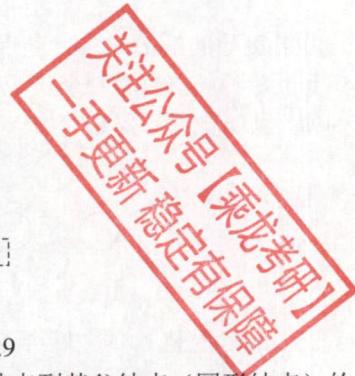
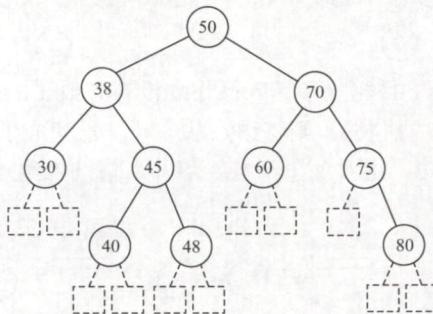
最小不平衡子树就是该树本身，插入位置在根结点的右子树的左子树上，因此需要进行 RL 旋转，RL 旋转过程如下图所示，旋转完成后根结点的关键字为 25，故选 D。



二、综合应用题

01. 【解答】

先序序列为(50, 38, 30, 45, 40, 48, 70, 60, 75, 80)，二叉树的中序序列是一个有序序列，故为(30, 38, 40, 45, 48, 50, 60, 70, 75, 80)，由先序序列和中序序列可以构造出对应的二叉树，如下图所示。



查找成功的平均查找长度为

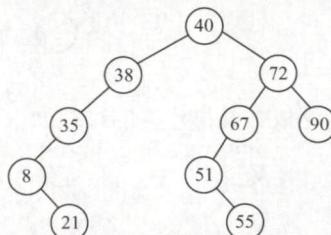
$$ASL = (1 \times 1 + 2 \times 2 + 3 \times 4 + 4 \times 3) / 10 = 2.9$$

图中的方块结点为虚构的查找失败结点，其查找路径为从根结点到其父结点（圆形结点）的结点序列，故对应的查找失败平均长度为

$$ASL = (3 \times 5 + 4 \times 6) / 11 = 39 / 11$$

02. 【解答】

根据二叉排序树的定义，该序列所对应的二叉排序树如下图所示。

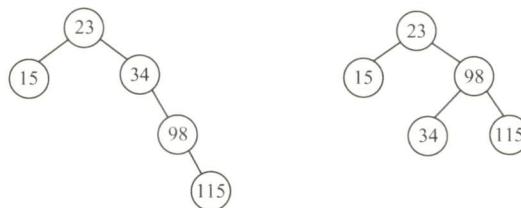


平均查找长度为 $ASL = (1 + 2 \times 2 + 3 \times 3 + 4 \times 2 + 5 \times 2) / 10 = 3.2$ 。

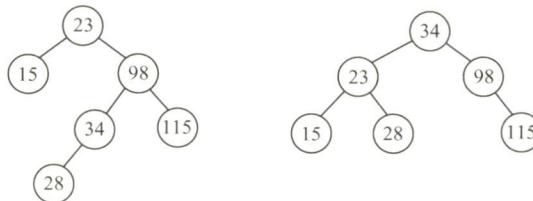
03. 【解答】



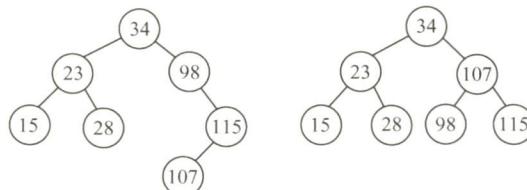
第一步：插入结点 34, 23, 15 后，需要根结点 34 的子树做 LL 调整。



第二步：插入结点 98, 115 后，需要根结点 34 的子树做 RR 调整。



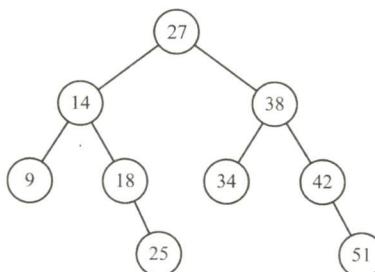
第三步：插入结点 28 后，需要根结点 23 的子树做 RL 调整。



第四步：插入结点 107 后，需要根结点 98 的子树做 RL 调整。

04. 【解答】

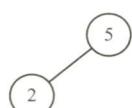
当各关键字的查找概率相等时，最佳二叉排序树应是高度最小的二叉排序树。构造过程分两步走：首先对各关键字按值从小到大排序，然后仿照折半查找的判定树的构造方法构造二叉排序树。这样得到的就是最佳二叉排序树，结果如下图所示。



05. 【解答】

大根堆要求根结点的关键字值既大于或等于左子女的关键字值，又大于或等于右子女的关键字值。二叉排序树要求根结点的关键字值大于左子女的关键字值，同时小于右子女的关键字值。两者的交集是：根结点的关键字值大于左子女的关键字值。这意味着它是一棵左斜单支树，但大根堆要求是完全二叉树，因此最后得到的只能是如右图所示的两个结点的二叉树。

读者也可能会注意到，当只有一个结点时，显然是满足题意的，但我们不举一个结点的例子是为了体现出排序树与大根堆的区别。



06. 【解答】

对二叉排序树来说，其中序遍历序列为一个递增有序序列。因此，对给定的二叉树进行中序遍历，若始终能保持前一个值比后一个值小，则说明该二叉树是一棵二叉排序树。算法实现如下：

```
KeyType predt=-32767; //predt 为全局变量，保存当前结点中序前驱的值，初值为-∞

int JudgeBST(BiTTree bt) {
    int b1,b2;
    if(bt==NULL) //空树
        return 1;
    else{
        b1=JudgeBST(bt->lchild); //判断左子树是否是二叉排序树
        if(b1==0||predt>=bt->data) //若左子树返回值为 0 或前驱大于或等于当前结点
            return 0; //则不是二叉排序树
        predt=bt->data; //保存当前结点的关键字
        b2=JudgeBST(bt->rchild); //判断右子树
        return b2; //返回右子树的结果
    }
}
```

07. 【解答】

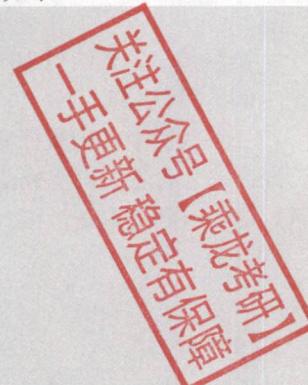
算法思想：设二叉树采用二叉链表存储结构。在二叉排序树中，查找一次就下降一层。因此，查找该结点所用的次数就是该结点在二叉排序树中的层次。采用二叉排序树非递归查找算法，用 n 保存查找层次，每查找一次， n 就加 1，直到找到相应的结点。算法如下：

```
int level(BiTTree bt,BSTNode *p) {
    int n=0; //统计查找次数
    BiTTree t=bt;
    if(bt!=NULL){
        n++;
        while(t->data!=p->data){
            if(p->data<t->data) //在左子树中查找
                t=t->lchild;
            else //在右子树中查找
                t=t->rchild;
            n++; //层次加 1
        }
    }
    return n;
}
```

08. 【解答】

设置二叉树的平衡标记 $balance$ ，以标记返回二叉树 bt 是否为平衡二叉树，若为平衡二叉树，则返回 1，否则返回 0； h 为二叉树 bt 的高度。采用后序遍历的递归算法：

- 1) 若 bt 为空，则高度为 0， $balance=1$ 。
- 2) 若 bt 仅有根结点，则高度为 1， $balance=1$ 。
- 3) 否则，对 bt 的左、右子树执行递归运算，返回左、右子树的高度和平衡标记， bt 的高度为最高子树的高度加 1。若左、右子树的高度差大于 1，则 $balance=0$ ；若左、右子树的高度差小于或等于 1，且左、右子树都平衡时， $balance=1$ ，否则 $balance=0$ 。



算法如下：

```
void Judge_AVL(BiTTree bt, int &balance, int &h) {
    int bl=0, br=0, hl=0, hr=0; //左、右子树的平衡标记和高度
    if(bt==NULL) { //空树，高度为0
        h=0;
        balance=1;
    }
    else if(bt->lchild==NULL&&bt->rchild==NULL) { //仅有根结点，则高度为1
        h=1;
        balance=1;
    }
    else{
        Judge_AVL(bt->lchild, bl, hl); //递归判断左子树
        Judge_AVL(bt->rchild, br, hr); //递归判断右子树
        h=(hl>hr?hl:hr)+1;
        if(abs(hl-hr)<2) //若子树高度差的绝对值<2，则看左、右子树是否都平衡
            balance=bl&br; //&&为逻辑与，即左、右子树都平衡时，二叉树平衡
        else
            balance=0;
    }
}
```

09. 【解答】

在一棵二叉排序树中，最左下结点即为关键字最小的结点，最右下结点即为关键字最大的结点，本算法只要找出这两个结点即可，而不需要比较关键字。算法如下：

```
KeyType MinKey(BSTNode *bt) {
    while(bt->lchild!=NULL)
        bt=bt->lchild;
    return bt->data;
}
KeyType MaxKey(BSTNode *bt) {
//求出二叉排序树中最大关键字结点
    while(bt->rchild!=NULL)
        bt=bt->rchild;
    return bt->data;
}
```



10. 【解答】

由二叉排序树的性质可知，右子树中所有的结点值均大于根结点值，左子树中所有的结点值均小于根结点值。为了从大到小输出，先遍历右子树，再访问根结点，后遍历左子树。算法如下：

```
void OutPut(BSTNode *bt, KeyType k)
{//本算法从大到小输出二叉排序树中所有值不小于 k 的关键字
    if(bt==NULL)
        return;
    if(bt->rchild!=NULL)
        OutPut(bt->rchild, k); //递归输出右子树结点
    if(bt->data>=k)
        printf("%d", bt->data); //只输出大于或等于 k 的结点值
    if(bt->lchild!=NULL)
        OutPut(bt->lchild, k); //递归输出左子树的结点
}
```

本题也可采用中序遍历加辅助栈的方法实现。

11. 【解答】

设二叉排序树的根结点为*t，根据结点存储的信息，有以下几种情况：

t->lchild 为空时，情况如下：

- 1) 若 t->rchild 非空且 k==1，则*t 即为第 k 小的元素，查找成功。
- 2) 若 t->rchild 非空且 k!=1，则第 k 小的元素必在*t 的右子树。

t->lchild 非空时，情况如下：

- 1) t->lchild->count==k-1，*t 即为第 k 小的元素，查找成功
- 2) t->lchild->count>k-1，第 k 小的元素必在*t 的左子树，继续到*t 的左子树中查找。
- 3) t->lchild->count<k-1，第 k 小的元素必在右子树，继续搜索右子树，寻找第 k-(t->lchild->count+1) 小的元素。

对左右子树的搜索采用相同的规则，递归实现的算法描述如下：

```
BSTNode *Search_Small(BSTNode*t, int k) {
    //在以 t 为根的子树上寻找第 k 小的元素，返回其所在结点的指针。k 从 11 开始计算
    //在树结点中增加一个 count 数据成员，存储以该结点为根的子树的结点个数
    if(k<1 || k>t->count) return NULL;
    if(t->lchild==NULL) {
        if(k==1) return t;
        else return Search_Small(t->rchild, k-1);
    }
    else{
        if(t->lchild->count==k-1) return t;
        if(t->lchild->count>k-1) return Search_Small(t->lchild, k);
        if(t->lchild->count<k-1)
            return Search_Small(t->rchild, k-(t->lchild->count+1));
    }
}
```

最大查找长度取决于树的高度。由于二叉排序树是随机生成的，其高度应是 $O(\log_2 n)$ ，算法的时间复杂度为 $O(\log_2 n)$ 。

7.4 B 树和 B+树

考研大纲对 B 树和 B+树的要求各不相同，重点在于考查 B 树，不仅要求理解 B 树的基本特点，还要求掌握 B 树的建立、插入和删除操作，而对 B+树则只考查基本概念。

7.4.1 B 树及其基本操作

所谓 m 阶 B 树是所有结点的平衡因子均等于 0 的 m 路平衡查找树。

一棵 m 阶 B 树或为空树，或为满足如下特性的 m 叉树：

- 1) 树中每个结点至多有 m 棵子树，即至多含有 m-1 个关键字。
- 2) 若根结点不是叶结点，则至少有两棵子树。
- 3) 除根结点外的所有非叶结点至少有 $\lceil m/2 \rceil$ 棵子树，即至少含有 $\lceil m/2 \rceil - 1$ 个关键字。
- 4) 所有非叶结点的结构如下：

n	P_0	K_1	P_1	K_2	P_2	\dots	K_n	P_n
-----	-------	-------	-------	-------	-------	---------	-------	-------

其中, K_i ($i = 1, 2, \dots, n$) 为结点的关键字, 且满足 $K_1 < K_2 < \dots < K_n$; P_i ($i = 0, 1, \dots, n$) 为指向子树根结点的指针, 且指针 P_{i-1} 所指子树中所有结点的关键字均小于 K_i , P_i 所指子树中所有结点的关键字均大于 K_i , n ($\lceil m/2 \rceil - 1 \leq n \leq m - 1$) 为结点中关键字的个数。

- 5) 所有的叶结点都出现在同一层次上, 并且不带信息 (可以视为外部结点或类似于折半查找判定树的查找失败结点, 实际上这些结点不存在, 指向这些结点的指针为空)。

图 7.28 所示为一棵 5 阶 B 树, 可以借助该实例来分析上述性质:

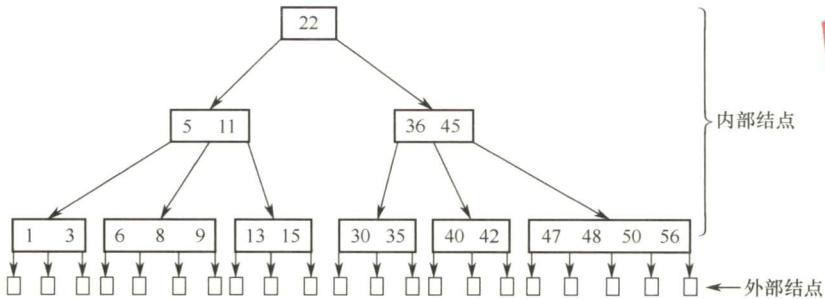


图 7.28 一棵 5 阶 B 树的实例



- 1) 结点的孩子个数等于该结点中关键字个数加 1。
- 2) 如果根结点没有关键字就没有子树, 此时 B 树为空; 如果根结点有关键字, 则其子树个数必然大于或等于 2, 因为子树个数等于关键字个数加 1。
- 3) 除根结点外的所有非叶结点至少有 $\lceil m/2 \rceil = \lceil 5/2 \rceil = 3$ 棵子树 (即至少有 $\lceil m/2 \rceil - 1 = \lceil 5/2 \rceil - 1 = 2$ 个关键字); 至多有 5 棵子树 (即至多有 4 个关键字)。
- 4) 结点中的关键字从左到右递增有序, 关键字两侧均有指向子树的指针, 左侧指针所指子树的所有关键字均小于该关键字, 右侧指针所指子树的所有关键字均大于该关键字。或者看成下层结点的关键字总是落在由上层结点的关键字所划分的区间内, 如第二层最左结点的关键字划分成了 3 个区间: $(-\infty, 5), (5, 11), (11, +\infty)$, 该结点中的 3 个指针所指子树的关键字均分别落在这 3 个区间内。
- 5) 所有叶结点均在第 4 层, 代表查找失败的位置。

1. B 树的高度 (磁盘存取次数)

由下一节将得知, B 树中的大部分操作所需的磁盘存取次数与 B 树的高度成正比。

下面来分析 B 树在不同情况下的高度。当然, 首先应该明确 B 树的高度不包括最后的不带任何信息的叶结点所处的那一层 (有些书对 B 树的高度的定义中, 包含最后的那一层)。

若 $n \geq 1$, 则对任意一棵包含 n 个关键字、高度为 h 、阶数为 m 的 B 树:

- 1) 因为 B 树中每个结点最多有 m 棵子树, $m-1$ 个关键字, 所以在一棵高度为 h 的 m 阶 B 树中关键字的个数应满足 $n \leq (m-1)(1+m+m^2+\dots+m^{h-1}) = m^h - 1$, 因此有

$$h \geq \log_m(n+1)$$

- 2) 若让每个结点中的关键字个数达到最少, 则容纳同样多关键字的 B 树的高度达到最大。

第一层至少有 1 个结点; 第二层至少有 2 个结点; 除根结点外的每个非叶结点至少有 $\lceil m/2 \rceil$ 棵子树, 则第三层至少有 $2\lceil m/2 \rceil$ 个结点……第 $h+1$ 层至少有 $2(\lceil m/2 \rceil)^{h-1}$ 个结点, 注意到第 $h+1$ 层是不包含任何信息的叶结点。对于关键字个数为 n 的 B 树, 叶结点即查找不到的结点为 $n+1$, 由此有 $n+1 \geq 2(\lceil m/2 \rceil)^{h-1}$, 即 $h \leq \log_{\lceil m/2 \rceil}((n+1)/2) + 1$ 。

例如，假设一棵 3 阶 B 树共有 8 个关键字，则其高度范围为 $2 \leq h \leq 3.17$ 。

2. B 树的查找

在 B 树上进行查找与二叉查找树很相似，只是每个结点都是多个关键字的有序表，在每个结点上所做的不是两路分支决定，而是根据该结点的子树所做的多路分支决定。

B 树的查找包含两个基本操作：① 在 B 树中找结点；② 在结点内找关键字。由于 B 树常存储在磁盘上，因此前一个查找操作是在磁盘上进行的，而后一个查找操作是在内存中进行的，即在找到目标结点后，先将结点信息读入内存，然后在结点内采用顺序查找法或折半查找法。

在 B 树上查找到某个结点后，先在有序表中进行查找，若找到则查找成功，否则按照对应的指针信息到所指的子树中去查找（例如，在图 7.28 中查找关键字 42，首先从根结点开始，根结点只有一个关键字，且 $42 > 22$ ，若存在，必在关键字 22 的右边子树上，右孩子结点有两个关键字，而 $36 < 42 < 45$ ，则若存在，必在 36 和 45 中间的子树上，在该子结点中查到关键字 42，查找成功）。查找到叶结点时（对应指针为空），则说明树中没有对应的关键字，查找失败。

3. B 树的插入

与二叉查找树的插入操作相比，B 树的插入操作要复杂得多。在 B 树中查找到插入的位置后，并不能简单地将其添加到终端结点（最底层的非叶结点）中，因为此时可能会导致整棵树不再满足 B 树定义中的要求。将关键字 key 插入 B 树的过程如下：

- 1) 定位。利用前述的 B 树查找算法，找出插入该关键字的最底层中的某个非叶结点（在 B 树中查找 key 时，会找到表示查找失败的叶结点，这样就确定了最底层非叶结点的插入位置。**注意：**插入位置一定是最底层中的某个非叶结点）。
- 2) 插入。在 B 树中，每个非失败结点的关键字个数都在区间 $\lceil m/2 \rceil - 1, m - 1$ 内。插入后的结点关键字个数小于 m ，可以直接插入；插入后检查被插入结点内关键字的个数，当插入后的结点关键字个数大于 $m - 1$ 时，必须对结点进行分裂。

分裂的方法是：取一个新结点，在插入 key 后的原结点，从中间位置 $(\lceil m/2 \rceil)$ 将其中的关键字分为两部分，左部分包含的关键字放在原结点中，右部分包含的关键字放到新结点中，中间位置 $(\lceil m/2 \rceil)$ 的结点插入原结点的父结点。若此时导致其父结点的关键字个数也超过了上限，则继续进行这种分裂操作，直至这个过程传到根结点为止，进而导致 B 树高度增 1。

对于 $m = 3$ 的 B 树，所有结点中最多有 $m - 1 = 2$ 个关键字，若某结点中已有两个关键字，则结点已满，如图 7.29(a) 所示。插入一个关键字 60 后，结点内的关键字个数超过了 $m - 1$ ，如图 7.29(b) 所示，此时必须进行结点分裂，分裂的结果如图 7.29(c) 所示。

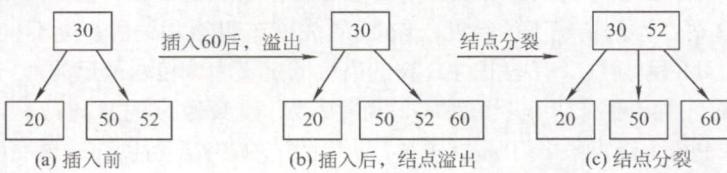


图 7.29 结点的“分裂”示意

4. B 树的删除

B 树中的删除操作与插入操作类似，但要稍微复杂一些，即要使得删除后的结点中的关键字个数 $\geq \lceil m/2 \rceil - 1$ ，因此将涉及结点的“合并”问题。

当被删关键字 k 不在终端结点（最底层的非叶结点）中时，可以用 k 的前驱（或后继） k' 来替代 k ，然后在相应的结点中删除 k' ，关键字 k' 必定落在某个终端结点中，则转换成了被删关键

字在终端结点中的情形。在图 7.30 的 4 阶 B 树中，删除关键字 80，用其前驱 78 替代，然后在终端结点中删除 78。因此只需讨论删除终端结点中的关键字的情形。

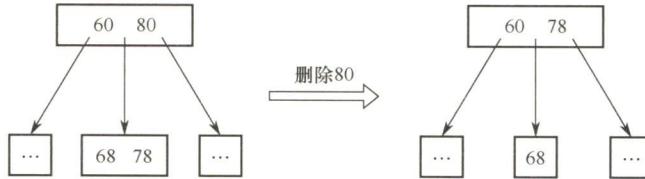


图 7.30 B 树中删除非终端结点关键字的取代

当被删关键字在终端结点中时，有下列三种情况：

- 1) 直接删除关键字。若被删关键字所在结点的关键字个数 $\geq \lceil m/2 \rceil$ ，表明删除该关键字后仍满足 B 树的定义，则直接删去该关键字。
- 2) 兄弟够借。若被删关键字所在结点删除前的关键字个数 $= \lceil m/2 \rceil - 1$ ，且与该结点相邻的右（或左）兄弟结点的关键字个数 $\geq \lceil m/2 \rceil$ ，则需要调整该结点、右（或左）兄弟结点及其双亲结点（父子换位法），以达到新的平衡。在图 7.31(a)中删除 4 阶 B 树的关键字 65，右兄弟关键字个数 $\geq \lceil m/2 \rceil = 2$ ，将 71 取代原 65 的位置，将 74 调整到 71 的位置。
- 3) 兄弟不够借。若被删关键字所在结点删除前的关键字个数 $= \lceil m/2 \rceil - 1$ ，且此时与该结点相邻的左、右兄弟结点的关键字个数均 $= \lceil m/2 \rceil - 1$ ，则将关键字删除后与左（或右）兄弟结点及双亲结点中的关键字进行合并。在图 7.31(b)中删除 4 阶 B 树的关键字 5，它及其右兄弟结点的关键字个数 $= \lceil m/2 \rceil - 1 = 1$ ，故在 5 删除后将 60 合并到 65 结点中。

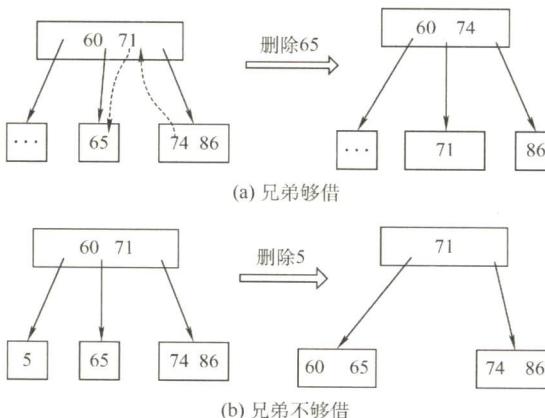


图 7.31 4 阶 B 树中删除终端结点关键字的示意图

- 3) 兄弟不够借。若被删关键字所在结点删除前的关键字个数 $= \lceil m/2 \rceil - 1$ ，且此时与该结点相邻的左、右兄弟结点的关键字个数均 $= \lceil m/2 \rceil - 1$ ，则将关键字删除后与左（或右）兄弟结点及双亲结点中的关键字进行合并。在图 7.31(b)中删除 4 阶 B 树的关键字 5，它及其右兄弟结点的关键字个数 $= \lceil m/2 \rceil - 1 = 1$ ，故在 5 删除后将 60 合并到 65 结点中。

在合并过程中，双亲结点中的关键字个数会减 1。若其双亲结点是根结点且关键字个数减少至 0（根结点关键字个数为 1 时，有 2 棵子树），则直接将根结点删除，合并后的新结点成为根；若双亲结点不是根结点，且关键字个数减少到 $\lceil m/2 \rceil - 2$ ，则又要与它自己的兄弟结点进行调整或合并操作，并重复上述步骤，直至符合 B 树的要求为止。

7.4.2 B+树的基本概念

B+树是应数据库所需而出现的一种 B 树的变形树。

一棵 m 阶的 B+树需满足下列条件：

- 1) 每个分支结点最多有 m 棵子树（孩子结点）。

关注公众号【乘龙考研】
一手更新 稳定有保障

- 2) 非叶根结点至少有两棵子树，其他每个分支结点至少有 $\lceil m/2 \rceil$ 棵子树。
- 3) 结点的子树个数与关键字个数相等。
- 4) 所有叶结点包含全部关键字及指向相应记录的指针，叶结点中将关键字按大小顺序排列，并且相邻叶结点按大小顺序相互链接起来。
- 5) 所有分支结点（可视为索引的索引）中仅包含它的各个子结点（即下一级的索引块）中关键字的最大值及指向其子结点的指针。

m 阶的 B+树与 m 阶的 B 树的主要差异如下：

- 1) 在 B+树中，具有 n 个关键字的结点只含有 n 棵子树，即每个关键字对应一棵子树；而在 B 树中，具有 n 个关键字的结点含有 $n+1$ 棵子树。
- 2) 在 B+树中，每个结点（非根内部结点）的关键字个数 n 的范围是 $\lceil m/2 \rceil \leq n \leq m$ （而根结点： $1 \leq n \leq m$ ）；而在 B 树中，每个结点（非根内部结点）的关键字个数 n 的范围是 $\lceil m/2 \rceil - 1 \leq n \leq m - 1$ （根结点： $1 \leq n \leq m - 1$ ）。
- 3) 在 B+树中，叶结点包含了全部关键字，非叶结点中出现的关键字也会出现在叶结点中；而在 B 树中，最外层的终端结点包含的关键字和其他结点包含的关键字是不重复的。
- 4) 在 B+树中，叶结点包含信息，所有非叶结点仅起索引作用，非叶结点中的每个索引项只含有对应子树的最大关键字和指向该子树的指针，不含有该关键字对应记录的存储地址。

图 7.32 所示为一棵 4 阶 B+树。可以看出，分支结点的某个关键字是其子树中最大关键字的副本。通常在 B+树中有两个头指针：一个指向根结点，另一个指向关键字最小的叶结点。因此，可以对 B+树进行两种查找运算：一种是从最小关键字开始的顺序查找，另一种是从根结点开始的多路查找。

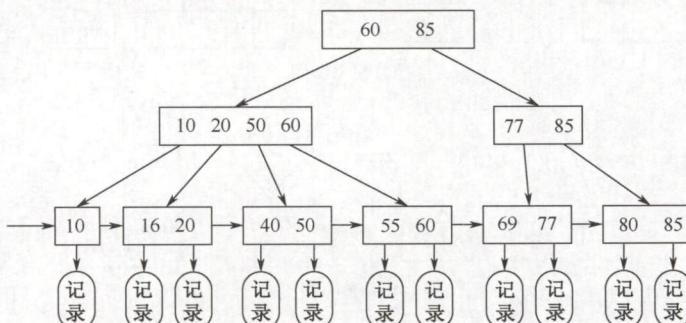


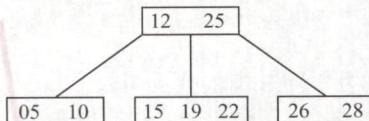
图 7.32 B+树结构示意图

B+树的查找、插入和删除操作和 B 树的基本类似。只是在查找过程中，非叶结点上的关键值等于给定值时并不终止，而是继续向下查找，直到叶结点上的该关键字为止。所以，在 B+树中查找时，无论查找成功与否，每次查找都是一条从根结点到叶结点的路径。

7.4.3 本节试题精选

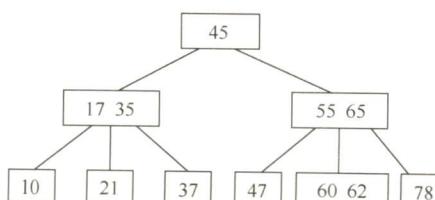
一、单项选择题

01. 下图所示是一棵 ()。



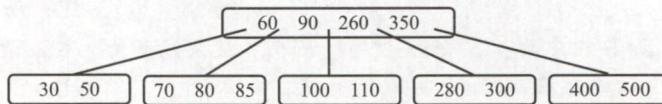
- A. 4 阶 B 树 B. 4 阶 B+树 C. 3 阶 B 树 D. 3 阶 B+树

02. 下列关于 m 阶 B 树的说法中，错误的是（ ）。
- 根结点至多有 m 棵子树
 - 所有叶结点都在同一层次上
 - 非叶结点至少有 $m/2$ (m 为偶数) 或 $(m+1)/2$ (m 为奇数) 棵子树
 - 根结点中的数据是有序的
03. 以下关于 m 阶 B 树的说法中，正确的是（ ）。
- 每个结点至少有两棵非空子树
 - 树中每个结点至多有 $m-1$ 个关键字
 - 所有叶结点在同一层
 - 插入一个元素引起 B 树结点分裂后，树长高一层
- I、II
 - II、III
 - III、IV
 - I、II、IV
04. 在一棵 m 阶 B 树中做插入操作前，若一个结点中的关键字个数等于（ ），则插入操作后必须分裂成两个结点；在一棵 m 阶 B 树中做删除操作前，若一个结点中的关键字个数等于（ ），则删除操作后可能需要同它的左兄弟或右兄弟结点合并成一个结点。
- $m, \lceil m/2 \rceil - 2$
 - $m-1, \lceil m/2 \rceil - 1$
 - $m+1, \lceil m/2 \rceil$
 - $m/2, \lceil m/2 \rceil + 1$
05. 具有 n 个关键字的 m 阶 B 树，应有（ ）个叶结点。
- $n+1$
 - $n-1$
 - mn
 - $nm/2$
06. 高度为 5 的 3 阶 B 树至少有（ ）个结点，至多有（ ）个结点。
- 32
 - 31
 - 120
 - 121
07. 含有 n 个非叶结点的 m 阶 B 树中至少包含（ ）个关键字。
- $n(m+1)$
 - n
 - $n(\lceil m/2 \rceil - 1)$
 - $(n-1)(\lceil m/2 \rceil - 1) + 1$
08. 已知一棵 5 阶 B 树中共有 53 个关键字，则树的最大高度为（ ），最小高度为（ ）。
- 2
 - 3
 - 4
 - 5
09. 已知一棵 3 阶 B 树中有 2047 个关键字，则此 B 树的最大高度为（ ），最小高度为（ ）。
- 11
 - 10
 - 8
 - 7
10. 下列关于 B 树和 B+树的叙述中，不正确的是（ ）。
- B 树和 B+树都能有效地支持顺序查找
 - B 树和 B+树都能有效地支持随机查找
 - B 树和 B+树都是平衡的多叉树
 - B 树和 B+树都可以用于文件索引结构
11. 【2009 统考真题】下列叙述中，不符合 m 阶 B 树定义要求的是（ ）。
- 根结点至多有 m 棵子树
 - 所有叶结点都在同一层上
 - 各结点内关键字均升序或降序排列
 - 叶结点之间通过指针链接
12. 【2012 统考真题】已知一棵 3 阶 B 树，如下图所示。删除关键字 78 得到一棵新 B 树，其最右叶结点中的关键字是（ ）。



关注公众号【乘龙考研】
一手更新 稳定有保障

- A. 60 B. 60, 62 C. 62, 65 D. 65
13. 【2013 统考真题】在一棵高度为 2 的 5 阶 B 树中，所含关键字的个数至少是（ ）。
 A. 5 B. 7 C. 8 D. 14
14. 【2014 统考真题】在一棵有 15 个关键字的 4 阶 B 树中，含关键字的结点个数最多是（ ）。
 A. 5 B. 6 C. 10 D. 15
15. 【2016 统考真题】B+树不同于 B 树的特点之一是（ ）。
 A. 能支持顺序查找 B. 结点中含有关键字
 C. 根结点至少有两个分支 D. 所有叶结点都在同一层上
16. 【2017 统考真题】下列应用中，适合使用 B+树的是（ ）。
 A. 编译器中的词法分析 B. 关系数据库系统中的索引
 C. 网络中的路由表快速查找 D. 操作系统的磁盘空闲块管理
17. 【2018 统考真题】高度为 5 的 3 阶 B 树含有的关键字个数至少是（ ）。
 A. 15 B. 31 C. 62 D. 242
18. 【2020 统考真题】依次将关键字 5, 6, 9, 13, 8, 2, 12, 15 插入初始为空的 4 阶 B 树后，根结点中包含的关键字是（ ）。
 A. 8 B. 6, 9 C. 8, 13 D. 9, 12
19. 【2021 统考真题】在一棵高度为 3 的 3 阶 B 树中，根为第 1 层，若第 2 层中有 4 个关键字，则该树的结点数最多是（ ）。
 A. 11 B. 10 C. 9 D. 8
20. 【2022 统考真题】在下图所示的 5 阶 B 树 T 中，删除关键字 260 之后需要进行必要的调整，得到新的 B 树 T_1 。下列选项中，不可能是 T_1 根结点中关键字序列的是（ ）。



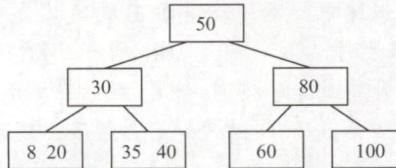
- A. 60, 90, 280 B. 60, 90, 350
 C. 60, 85, 110, 350 D. 60, 90, 110, 350

二、综合应用题

01. 给定一组关键字 {20, 30, 50, 52, 60, 68, 70}，给出创建一棵 3 阶 B 树的过程。

02. 对如下图所示的 3 阶 B 树，依次执行下列操作，画出各步操作的结果。

- 1) 插入 90 2) 插入 25 3) 插入 45 4) 删除 60 5) 删除 80



03. 利用 B 树做文件索引时，若假设磁盘页块的大小是 4000B（实际应是 2 的次幂，此处是为了计算方便），指示磁盘地址的指针需要 5B。现有 20000000 个记录构成的文件，每个记录为 200B，其中包括关键字 5B。

试问在这个采用 B 树作索引的文件中，B 树的阶数应为多少？假定文件数据部分未按关键字有序排列，则索引部分需要占用多少磁盘页块？

7.4.4 答案与解析

一、单项选择题

**关注公众号【乘龙考研】
一手更新 稳定有保障**

01. A

关键字数量比子树数量少 1，所以不是 B+树，而是 B 树。又因为 m 阶 B 树结点关键字数最多为 $m-1$ ，有一个结点关键字个数为 3，所以不可能为 3 阶。

02. C

除根结点外的所有非叶结点至少有 $\lceil m/2 \rceil$ 棵子树。对于根结点，最多有 m 棵子树，若其不是叶结点，则至少有 2 棵子树。

03. B

每个非根的内部结点必须至少有 $\lceil m/2 \rceil$ 棵子树，而根结点至少要有两棵子树，所以选项 I 不正确。选项 II、III 显然正确。对于 IV，插入一个元素引起 B 树结点分裂后，只要从根结点到该元素插入位置的路径上至少有一个结点未满，B 树就不会长高，如图 1 所示；只有当结点的分裂传到根结点，并使根结点也分裂时，才会导致树高增 1，如图 2 所示，因此选项 IV 错误。

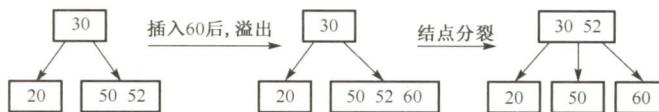


图 1 结点分裂不导致树高增 1 (3 阶 B 树)



图 2 结点分裂导致树高增 1 (3 阶 B 树)

04. B

由于 B 树每个结点内的关键字个数最多为 $m-1$ ，所以当关键字个数大于 $m-1$ 时，则应该分裂。每个结点内的关键字个数至少为 $\lceil m/2 \rceil - 1$ 个，所以当关键字个数少于 $\lceil m/2 \rceil - 1$ 时，则可能与其他结点合并（除非只有根结点）。若将本题题干改为 B+树，请读者思考上述问题的解答。

05. A

B 树的叶结点对应查找失败的情况，对有 n 个关键字的查找集合进行查找，失败可能性有 $n+1$ 种。

06. B、D

由 m 阶 B 树的性质可知，根结点至少有 2 棵子树；根结点外的所有非终端结点至少有 $\lceil m/2 \rceil$ 棵子树，结点数最少时，3 阶 B 树形状至少类似于一棵满二叉树，即高度为 5 的 B 树至少有 $2^5 - 1 = 31$ 个结点。由于每个结点最多有 m 棵子树，所以当结点数最多时，3 阶 B 树形状类似于满三叉树，结点数为 $(3^5 - 1)/2 = 121$ （注意，这里求的是结点数而非关键字数，若求的是关键字数，则还应把每个结点中关键字数的上下界确定出来）。

07. D

除根结点外， m 阶 B 树中的每个非叶结点至少有 $\lceil m/2 \rceil - 1$ 个关键字，根结点至少有一个关键字，所以总共包含的关键字最少个数 = $(n-1)(\lceil m/2 \rceil - 1) + 1$ 。

注意：由以上题目可知 B 树和 B+树的定义与性质尤为重要，需要熟练掌握。

08. C、B

5 阶 B 树中共有 53 个关键字，由最大高度公式 $H \leq \log_{m/2}((n+1)/2) + 1$ 得最大高度 $H \leq \log_3[(53+1)/2] + 1 = 4$ ，即最大高度为 4；由最小高度公式 $h \geq \log_m(n+1)$ 得最小高度 $h \geq \log_3 54 \approx 2.5$ ，从而最小高度为 3。

09. A、D

利用前面的公式即最小高度 $h \geq \log_m(n+1)$ 和最大高度 $H \leq \log_{\lceil m/2 \rceil}[(n+1)/2] + 1$ ，易算出最大高度 $H \leq \log_2[(2047+1)/2] + 1 = 11$ ，最小高度 $h \geq \log_3 2048 = 6.9$ ，从而最小高度取 7（注意，有些辅导书针对本题算出的高度要比这里给出的答案多 1，因为它们在对 B 树的高度定义中，把最底层不包含任何关键字的叶结点也算进去了）。

10. A

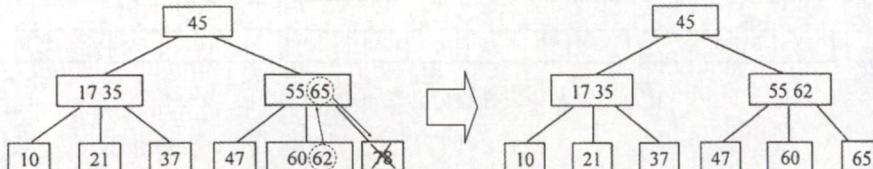
B 树和 B+树的差异主要体现在：①结点关键字和子树的个数；②B+树非叶结点仅起索引作用；③B 树叶结点关键字和其他结点包含的关键字是不重复的；④B+树支持顺序查找和随机查找，而 B 树仅支持随机查找。由于 B+树的所有叶结点中包含了全部的关键字信息，且叶结点本身依关键字从小到大顺序链接，因此可以进行顺序查找，而 B 树不支持顺序查找。

11. D

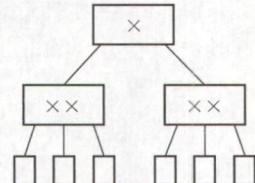
m 阶 B 树不要求将各叶结点之间用指针链接。选项 D 描述的实际上是 B+树。

12. D

对于图中所示的 3 阶 B 树，被删关键字 78 所在的结点在删除前的关键字个数 $= 1 = \lceil 3/2 \rceil - 1$ ，且其左兄弟结点的关键字个数 $= 2 \geq \lceil 3/2 \rceil$ ，属于“兄弟够借”的情况，因此要把该结点的左兄弟结点中的最大关键字上移到双亲结点中，同时把双亲结点中大于上移关键字的关键字下移到要删除关键字的结点中，这样就达到了新的平衡，如下图所示。

**13. A**

对于 5 阶 B 树，根结点的分支数最少为 2（关键字数最少为 1），其他非叶结点的分支数最少为 $\lceil n/2 \rceil = 3$ （关键字数最少为 2），因此关键字个数最少的情况如右图所示（叶结点不计入高度）。



注意：要与第 1 题相区别，有同学对此题的理解存在偏差。第 1 题要根据图示给出阶数并指出属于哪种树。对于本题所述的 5 阶 B 树，不要误认为：“存在至少有一个含关键字结点中的关键字达到 4”才符合 5 阶 B 树的要求，因为 5 阶 B 树中各个结点包含的关键字个数最少为 2 ($\lceil 5/2 \rceil - 1 = 2$)，最多为 4 ($5 - 1 = 4$)。当 5 阶 B 树中各个结点包含的关键字个数为 2 时，也满足 5 阶 B 树的要求 [此时若题目给定了关键字个数（如第 14 题），则可计算出该树中含关键字结点个数将达到最多；若各结点的关键字个数达到 4，则可计算出该树在总关键字个数一定的条件下含关键字结点最少的情况下。这与“当树高一定的情况下，求含关键字个数最多或最少（或相反“给定关键字个数求树高最大或最小”）”的思路（如 08 题）都是一样的。重中之重是对 B 树和 B+树的定义及特性要透彻理解。对此存在疑问的读者请回看 B 树的结构及特性，不要将二者混淆。]

14. D

关键字数量不变，要求结点数量最多，即要求每个结点中含关键字的数量最少。根据4阶B树的定义，根结点最少含1个关键字，非根结点中最少含 $\lceil 4/2 \rceil - 1 = 1$ 个关键字，所以每个结点中关键字数量最少都为1个，即每个结点都有2个分支，类似于排序二叉树，而15个结点正好可以构造一个4层的4阶B树，使得终端结点全在第四层，符合B树的定义，因此选D。

15. A

由于B+树的所有叶结点中包含了全部的关键字信息，且叶结点本身依关键字从小到大顺序链接，因此可以进行顺序查找，而B树不支持顺序查找（只支持多路查找）。

16. B

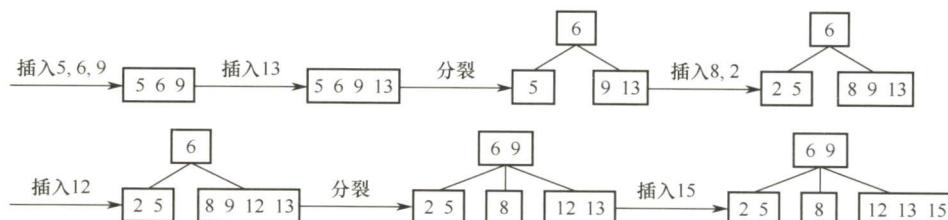
B+树是应文件系统所需而产生的B树的变形，前者比后者更加适用于实际应用中的操作系统的文件索引和数据库索引，因为前者的磁盘读写代价更低，查询效率更加稳定。编译器中的词法分析使用有穷自动机和语法树。网络中的路由表快速查找主要靠高速缓存、路由表压缩技术和快速查找算法。系统一般使用空闲空间链表管理磁盘空闲块。所以选项B正确。

17. B

m 阶B树的基本性质：根结点以外的非叶结点最少含有 $\lceil m/2 \rceil - 1$ 个关键字，代入 $m=3$ 得到每个非叶结点中最少包含1个关键字，而根结点含有1个关键字，因此所有非叶结点都有两个孩子。此时其树形与 $h=5$ 的满二叉树相同，可求得关键字最少为31个。

18. B

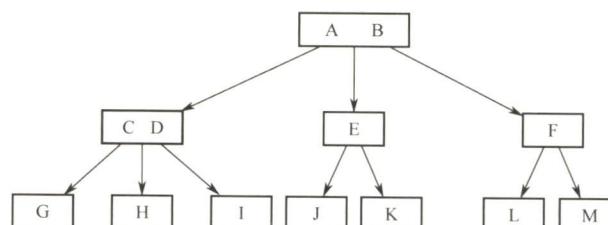
一个4阶B树的任意非叶结点至多含有 $m-1=3$ 个关键字，在关键字依次插入的过程中，会导致结点的不断分裂，插入过程如下所示。



得到根结点包含的关键字为6, 9。

19. A

在阶为3的B树中，每个结点至多含有2个关键字（至少1个），至多有3棵子树。本题规定第二层有4个关键字，欲使B树的结点数达到最多，则这4个关键字包含在3个结点中，B树树形如下图所示，其中A, B, C, …, M表示关键字，最多有11个结点，故选A。

**20. D**

在5阶B树中，除根结点外的非叶结点的关键字数 k 需要满足 $2 \leq k \leq 4$ 。当被删关键字 x 不在终端结点（最底层非叶结点）时，可以用 x 的前驱（或后继）关键字 y 来替代 x ，然后在相应结点中删除 y 。情况①：删除260，将其前驱110放入260处，删除110后的结点<100>不满足

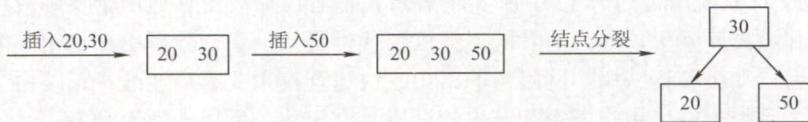
5 阶 B 树定义，从左兄弟中借 85，将 85 放入根中，将根中的 90 移入结点 $<100>$ 变为 $<90, 100>$ 。

情况②：删除 260，将其后继 280 放入 260 处，结点 $<300>$ 不满足 5 阶 B 树定义且左右兄弟都不够借，结点 $<300>$ 可以和左兄弟 $<100, 110>$ 以及关键字 280 合并成一个新的结点 $<100, 110, 280, 300>$ 。情况③：在情况②中，结点 $<300>$ 也可以和右兄弟 $<400, 500>$ 以及关键字 350 合并成一个新的结点 $<300, 350, 400, 500>$ 。综上， T_1 根结点中的关键字序列可能是 $<60, 85, 110, 350>$ 或 $<60, 90, 350>$ 或 $<60, 90, 280>$ ，仅 D 不可能。

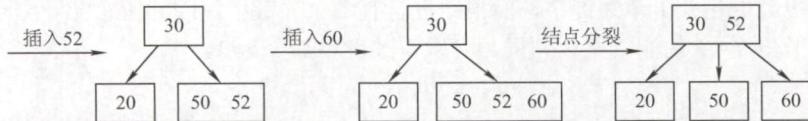
二、综合应用题

01. 【解答】

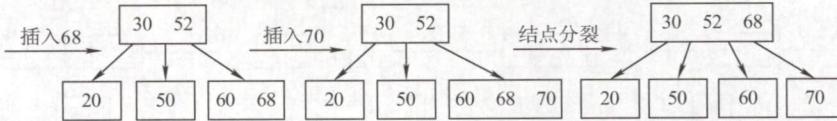
$m = 3$ ，因此除根结点外，非叶结点关键字个数为 1~2。



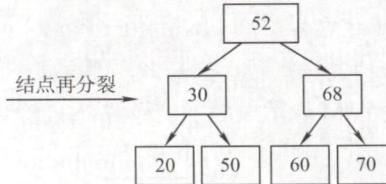
如上图所示，首先插入 20, 30，结点内关键字个数不超过 $m - 1 = 2$ ，不会引起分裂；插入 50，插入 20, 30 所在的结点，引起分裂，结点内第 $\lceil m/2 \rceil$ 个关键字 30 上升为父结点。



如上图所示，插入 52，插入 50 所在的结点，不会引起分裂；继续插入 60，插入 50, 52 所在的结点，引起分裂，52 上升到父结点中，不会引起父结点的分裂。

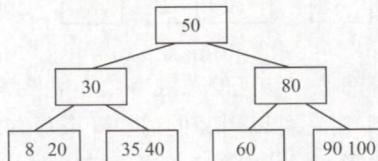


如上图所示，插入 68，插入 60 所在的结点，不会引起分裂；继续插入 70，插入 60, 68 所在的结点，引起分裂，68 上升为新的父结点，68 上升到 30, 52 所在的结点后，会继续引起该结点的分裂，故 52 上升为新的根结点。最后得到的 B 树如下图所示。

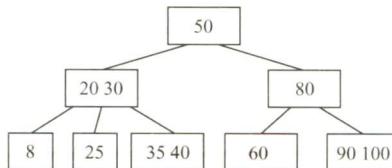


02. 【解答】

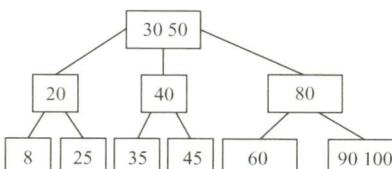
1) 插入 90：将 90 插入 100 所在的结点，插入 90 后该结点中的元素个数不超过 $\lceil 3/2 \rceil = 2$ ，不会引起结点的分裂，插入后的 B 树如下图所示。



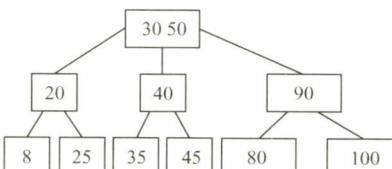
- 2) 插入 25: 将 25 插入 8, 20 所在的结点, 插入后结点内的元素个数为 3, 引起分裂。故将结点内的中间元素 20 上升到父结点中, 此时父结点中的元素个数为 2 (元素 20 和 30), 不会引起继续分裂, 插入 25 后的 B 树如下图所示。



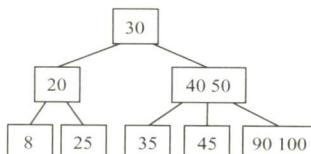
- 3) 插入 45: 将 45 插入 35, 40 所在的结点, 引起分裂, 中间元素 40 上升到父结点 (20, 30 所在的结点) 中, 引起父结点分裂, 中间元素 30 上升到父结点 (50 所在的结点) 中, 两次分裂后的 B 树如下图所示。



- 4) 删除 60: 删除 60 后, 其所在的结点元素为空, 从而导致借用右兄弟结点的元素, 调整后的 B 树如下图所示。



- 5) 删除 80: 删除 80 后, 导致 80 所在结点的父结点与其右兄弟结点合并, 这时父结点元素个数为 0, 再次对父结点进行调整。将 50 与 40 合并成一个新结点, 则 90, 100 所在结点为这个结点的子结点。从而构造的 B 树如下图所示。注意, 这次调整的过程实际上包含多次调整过程, 希望读者对照考点讲解中的删除过程仔细思考。



注意: B 树中结点的插入、删除操作 (特别是插入、删除后的结点分裂与合并) 是本节的重点, 也是难点, 请读者务必熟练掌握。

03. 【解答】

根据 B 树的概念, 一个索引结点应适应操作系统一次读写的物理记录大小, 其大小应取不超过但最接近一个磁盘页块的大小。假设 B 树为 m 阶, 一个 B 树结点最多存放 $m - 1$ 个关键字 (5B) 和对应的记录地址 (5B)、 m 个子树指针 (5B) 和 1 个指示结点中的实际关键字个数的整数 (2B), 则有

$$(2 \times (m - 1) + m) \times 5 + 2 \leq 4000$$

计算结果为 $m \leq 267$ 。

一个索引结点最多可以存放 $m - 1 = 266$ 个索引项，最少可以存放 $\lceil m/2 \rceil - 1 = 133$ 个索引项。全部有 $n = 20000000$ 个记录，每个记录占用空间 200B，每个页块可以存放 $4000/200 = 20$ 个记录，则全部记录分布在 $20000000/20 = 1000000$ 个页块中，最多需要占用 $1000000/133 = 7519$ 个磁盘页块作为 B 树索引，最少需要占用 $1000000/266 = 3760$ 个磁盘页块作为 B 树索引（注意 B 树与 B+ 树的不同，B 树所有对数据记录的索引项分布在各个层次的结点中，B+ 树所有对数据记录的索引项都在叶结点中）。

7.5 散列表

关注公众号【乘龙考研】
一手更新 稳定有保障

7.5.1 散列表的基本概念

在前面介绍的线性表和树表的查找中，记录在表中的位置与记录的关键字之间不存在确定关系，因此，在这些表中查找记录时需进行一系列的关键字比较。这类查找方法建立在“比较”的基础上，查找的效率取决于比较的次数。

散列函数：一个把查找表中的关键字映射成该关键字对应的地址的函数，记为 $H(key) = Addr$ （这里的地址可以是数组下标、索引或内存地址等）。

散列函数可能会把两个或两个以上的不同关键字映射到同一地址，称这种情况为冲突，这些发生碰撞的不同关键字称为同义词。一方面，设计得好的散列函数应尽量减少这样的冲突；另一方面，由于这样的冲突总是不可避免的，所以还要设计好处理冲突的方法。

散列表：根据关键字而直接进行访问的数据结构。也就是说，散列表建立了关键字和存储地址之间的一种直接映射关系。

理想情况下，对散列表进行查找的时间复杂度为 $O(1)$ ，即与表中元素的个数无关。下面分别介绍常用的散列函数和处理冲突的方法。

7.5.2 散列函数的构造方法

在构造散列函数时，必须注意以下几点：

- 1) 散列函数的定义域必须包含全部需要存储的关键字，而值域的范围则依赖于散列表的大小或地址范围。
- 2) 散列函数计算出来的地址应该能等概率、均匀地分布在整个地址空间中，从而减少冲突的发生。
- 3) 散列函数应尽量简单，能够在较短的时间内计算出任意一个关键字对应的散列地址。

下面介绍常用的散列函数。

1. 直接定址法

直接取关键字的某个线性函数值为散列地址，散列函数为

$$H(key) = key \text{ 或 } H(key) = a \times key + b$$

式中， a 和 b 是常数。这种方法计算最简单，且不会产生冲突。它适合关键字的分布基本连续的情况，若关键字分布不连续，空位较多，则会造成存储空间的浪费。

2. 除留余数法

这是一种最简单、最常用的方法，假定散列表表长为 m ，取一个不大于 m 但最接近或等于 m

的质数 p , 利用以下公式把关键字转换成散列地址。散列函数为

$$H(\text{key}) = \text{key \% } p$$

除留余数法的关键是选好 p , 使得每个关键字通过该函数转换后等概率地映射到散列空间上的任意一个地址, 从而尽可能减少冲突的可能性。

3. 数字分析法

设关键字是 r 进制数(如十进制数), 而 r 个数码在各位上出现的频率不一定相同, 可能在某些位上分布均匀一些, 每种数码出现的机会均等; 而在某些位上分布不均匀, 只有某几种数码经常出现, 此时应选取数码分布较为均匀的若干位作为散列地址。这种方法适合于已知的关键字集合, 若更换了关键字, 则需要重新构造新的散列函数。

4. 平方取中法

顾名思义, 这种方法取关键字的平方值的中间几位作为散列地址。具体取多少位要视实际情况而定。这种方法得到的散列地址与关键字的每位都有关系, 因此使得散列地址分布比较均匀, 适用于关键字的每位取值都不够均匀或均小于散列地址所需的位数。

在不同的情况下, 不同的散列函数具有不同的性能, 因此不能笼统地说哪种散列函数最好。在实际选择中, 采用何种构造散列函数的方法取决于关键字集合的情况, 但目标是尽量降低产生冲突的可能性。

7.5.3 处理冲突的方法

应该注意到, 任何设计出来的散列函数都不可能绝对地避免冲突。为此, 必须考虑在发生冲突时应该如何处理, 即为产生冲突的关键字寻找下一个“空”的 Hash 地址。用 H_i 表示处理冲突中第 i 次探测得到的散列地址, 假设得到的另一个散列地址 H_1 仍然发生冲突, 只得继续求下一个地址 H_2 , 以此类推, 直到 H_k 不发生冲突为止, 则 H_k 为关键字在表中的地址。

1. 开放定址法

所谓开放定址法, 是指可存放新表项的空闲地址既向它的同义词表项开放, 又向它的非同义词表项开放。其数学递推公式为

$$H_i = (H(\text{key}) + d_i) \% m$$

式中, $H(\text{key})$ 为散列函数; $i = 0, 1, 2, \dots, k$ ($k \leq m - 1$); m 表示散列表表长; d_i 为增量序列。

取定某一增量序列后, 对应的处理方法就是确定的。通常有以下 4 种取法:

- 1) 线性探测法。当 $d_i = 0, 1, 2, \dots, m - 1$ 时, 称为线性探测法。这种方法的特点是: 冲突发生时, 顺序查看表中下一个单元(探测到表尾地址 $m - 1$ 时, 下一个探测地址是表首地址 0), 直到找出一个空闲单元(当表未填满时一定能找到一个空闲单元)或查遍全表。线性探测法可能使第 i 个散列地址的同义词存入第 $i + 1$ 个散列地址, 这样本应存入第 $i + 1$ 个散列地址的元素就争夺第 $i + 2$ 个散列地址的元素的地址……从而造成大量元素在相邻的散列地址上“聚集”(或堆积)起来, 大大降低了查找效率。
- 2) 平方探测法。当 $d_i = 0^2, 1^2, -1^2, 2^2, -2^2, \dots, k^2, -k^2$ 时, 称为平方探测法, 其中 $k \leq m/2$, 散列表长度 m 必须是一个可以表示成 $4k + 3$ 的素数, 又称二次探测法。平方探测法是一种处理冲突的较好方法, 可以避免出现“堆积”问题, 它的缺点是不能探测到散列表上的所有单元, 但至少能探测到一半单元。
- 3) 双散列法。当 $d_i = \text{Hash}_2(\text{key})$ 时, 称为双散列法。需要使用两个散列函数, 当通过第一个

散列函数 $H(key)$ 得到的地址发生冲突时，则利用第二个散列函数 $\text{Hash}_2(key)$ 计算该关键字的地址增量。它的具体散列函数形式如下：

$$H_i = (H(key) + i \times \text{Hash}_2(key)) \% m$$

初始探测位置 $H_0 = H(key) \% m$ 。 i 是冲突的次数，初始为 0。在双散列法中，最多经过 $m - 1$ 次探测就会遍历表中所有位置，回到 H_0 位置。

4) 伪随机序列法。当 $d_i = \text{伪随机数序列}$ 时，称为伪随机序列法。

注意：在开放定址的情形下，不能随便物理删除表中的已有元素，因为若删除元素，则会截断其他具有相同散列地址的元素的查找地址。因此，要删除一个元素时，可给它做一个删除标记，进行逻辑删除。但这样做的副作用是：执行多次删除后，表面上看起来散列表很满，实际上有许多位置未利用，因此需要定期维护散列表，要把删除标记的元素物理删除。

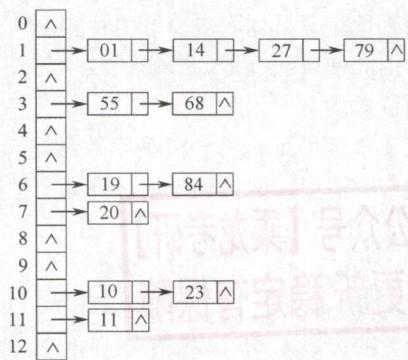


图 7.33 拉链法处理冲突的散列表

2. 拉链法（链接法，chaining）

显然，对于不同的关键字可能会通过散列函数映射到同一地址，为了避免非同义词发生冲突，可以把所有的同义词存储在一个线性链表中，这个线性链表由其散列地址唯一标识。假设散列地址为 i 的同义词链表的头指针存放在散列表的第 i 个单元中，因而查找、插入和删除操作主要在同义词链中进行。拉链法适用于经常进行插入和删除的情况。

例如，关键字序列为 {19, 14, 23, 01, 68, 20, 84, 27, 55, 11, 10, 79}，散列函数 $H(key) = key \% 13$ ，用拉链法处理冲突，建立的表如图 7.33 所示（学完下节内容后，可以尝试计算本例的平均查找长度 ASL）。

7.5.4 散列查找及性能分析

散列表的查找过程与构造散列表的过程基本一致。对于一个给定的关键字 key ，根据散列函数可以计算出其散列地址，执行步骤如下：

初始化： $Addr = Hash(key)$ ；

① 检测查找表中地址为 $Addr$ 的位置上是否有记录，若无记录，返回查找失败；若有记录，比较它与 key 的值，若相等，则返回查找成功标志，否则执行步骤②。

② 用给定的处理冲突方法计算“下一个散列地址”，并把 $Addr$ 置为此地址，转入步骤①。

例如，关键字序列 {19, 14, 23, 01, 68, 20, 84, 27, 55, 11, 10, 79} 按散列函数 $H(key) = key \% 13$ 和线性探测处理冲突构造所得的散列表 L 如图 7.34 所示。

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	14	01	68	27	55	19	20	84	79	23	11	10			

图 7.34 用线性探测法得到的散列表 L

给定值 84 的查找过程为：首先求得散列地址 $H(84) = 6$ ，因 $L[6]$ 不空且 $L[6] \neq 84$ ，则找第一次冲突处理后的地址 $H_1 = (6+1) \% 16 = 7$ ，而 $L[7]$ 不空且 $L[7] \neq 84$ ，则找第二次冲突处理后的地址 $H_2 = (6+2) \% 16 = 8$ ， $L[8]$ 不空且 $L[8] = 84$ ，查找成功，返回记录在表中的序号 8。

给定值 38 的查找过程为：先求散列地址 $H(38) = 12$ ， $L[12]$ 不空且 $L[12] \neq 38$ ，则找下一地址 $H_1 = (12+1) \% 16 = 13$ ，由于 $L[13]$ 是空记录，故表中不存在关键字为 38 的记录。

查找各关键字的比较次数如图 7.35 所示。

关键字	14	01	68	27	55	19	20	84	79	23	11	10
比较次数	1	2	1	4	3	1	1	3	9	1	1	3

图 7.35 查找各关键字的比较次数

平均查找长度 ASL 为

$$ASL = (1 \times 6 + 2 + 3 \times 3 + 4 + 9) / 12 = 2.5$$

对同一组关键字，设定相同的散列函数，则不同的处理冲突的方法得到的散列表不同，它们的平均查找长度也不同，本例与上节采用拉链法的平均查找长度不同。

从散列表的查找过程可见：

(1) 虽然散列表在关键字与记录的存储位置之间建立了直接映像，但由于“冲突”的产生，使得散列表的查找过程仍然是一个给定值和关键字进行比较的过程。因此，仍需要以平均查找长度作为衡量散列表的查找效率的度量。

(2) 散列表的查找效率取决于三个因素：散列函数、处理冲突的方法和装填因子。

装填因子。散列表的装填因子一般记为 α ，定义为一个表的装满程度，即

$$\alpha = \frac{\text{表中记录数 } n}{\text{散列表长度 } m}$$

散列表的平均查找长度依赖于散列表的装填因子 α ，而不直接依赖于 n 或 m 。直观地看， α 越大，表示装填的记录越“满”，发生冲突的可能性越大，反之发生冲突的可能性越小。

读者应能在给出散列表的长度、元素个数及散列函数和解决冲突的方法后，在求出散列表的基础上计算出查找成功时的平均查找长度和查找不成功的平均查找长度。

7.5.5 本节试题精选

一、单项选择题

01. 只能在顺序存储结构上进行的查找方法是()。
- 顺序查找法
 - 折半查找法
 - 树型查找法
 - 散列查找法
02. 散列查找一般适用于()的情况下查找。
- 查找表为链表
 - 查找表为有序表
 - 关键字集合比地址集合大得多
 - 关键字集合与地址集合之间存在对应关系
03. 下列关于散列表的说法中，正确的是()。
- 若散列表的填装因子 $\alpha < 1$ ，则可避免碰撞的产生
 - 散列查找中不需要任何关键字的比较
 - 散列表在查找成功时平均查找长度与表长有关
 - 若在散列表中删除一个元素，不能简单地将该元素删除
- I 和 IV
 - II 和 III
 - III
 - IV
04. 在开放定址法中散列到同一个地址而引起的“堆积”问题是由于()引起的。
- 同义词之间发生冲突
 - 非同义词之间发生冲突

关注公众号【乘龙考研】
一手更新 稳定有保障

- C. 同义词之间或非同义词之间发生冲突
D. 散列表“溢出”
05. 下列关于散列冲突处理方法的说法中，正确的有（ ）。
 I. 采用再散列法处理冲突时不易产生聚集
 II. 采用线性探测法处理冲突时，所有同义词在散列表中一定相邻
 III. 采用链地址法处理冲突时，若限定在链首插入，则插入任意一个元素的时间相同
 IV. 采用链地址法处理冲突易引起聚集现象
 A. I 和 III B. I、II 和 III C. III 和 IV D. I 和 IV
06. 设有一个含有 200 个表项的散列表，用线性探测法解决冲突，按关键字查询时找到一个表项的平均探测次数不超过 1.5，则散列表项应能够容纳（ ）个表项（设查找成功的平均查找长度为 $ASL = [1 + 1/(1 - \alpha)]/2$ ，其中 α 为装填因子）。
 A. 400 B. 526 C. 624 D. 676
07. 假定有 K 个关键字互为同义词，若用线性探测法把这 K 个关键字填入散列表，至少要进行（ ）次探测。
 A. $K - 1$ B. K C. $K + 1$ D. $K(K + 1)/2$
08. 对包含 n 个元素的散列表进行查找，平均查找长度（ ）。
 A. 为 $O(\log_2 n)$ B. 为 $O(1)$ C. 不直接依赖于 n D. 直接依赖于表长 m
09. 采用开放定址法解决冲突的散列查找中，发生聚集的原因主要是（ ）。
 A. 数据元素过多 B. 负载因子过大
 C. 散列函数选择不当 D. 解决冲突的方法选择不当
10. 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 $H(key) = key \bmod 13$ ，散列地址为 1 的链中有（ ）个记录。
 A. 1 B. 2 C. 3 D. 4
11. 在采用链地址法处理冲突所构成的散列表上查找某一关键字，则在查找成功的情况下，所探测的这些位置上的关键字值（ ）；若采用线性探测法，则（ ）。
 A. 一定都是同义词 B. 不一定都是同义词
 C. 都相同 D. 一定都不是同义词
12. 若采用链地址法构造散列表，散列函数为 $H(key) = key \bmod 17$ ，则需（①）个链表。这些链的链首指针构成一个指针数组，数组的下标范围为（②）。
 ①A. 17 B. 13 C. 16 D. 任意
 ②A. 0 ~ 17 B. 1 ~ 17 C. 0 ~ 16 D. 1 ~ 16
13. 设散列表长 $m = 14$ ，散列函数为 $H(key) = key \% 11$ ，表中仅有 4 个结点 $H(15) = 4$ ， $H(38) = 5$ ， $H(61) = 6$ ， $H(84) = 7$ ，若采用线性探测法处理冲突，则关键字为 49 的结点地址是（ ）。
 A. 8 B. 3 C. 5 D. 9
14. 将 10 个元素散列到 100000 个单元的散列表中，则（ ）产生冲突。
 A. 一定会 B. 一定不会 C. 仍可能会 D. 不确定
15. 【2011 统考真题】为提高散列表的查找效率，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
 A. 仅 I B. 仅 II C. 仅 I、II D. 仅 II、III
16. 【2014 统考真题】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（ ）。
 A. 存储效率 B. 散列函数
 C. 装填（装载）因子 D. 平均查找长度
17. 【2018 统考真题】现有长度为 7、初始为空的散列表 HT，散列函数 $H(k) = k \% 7$ ，用线性探测再散列法解决冲突。将关键字 22, 43, 15 依次插入 HT 后，查找成功的平均查找长度是（ ）。
 A. 1.5 B. 1.6 C. 2 D. 3
18. 【2019 统考真题】现有长度为 11 且初始为空的散列表 HT，散列函数是 $H(key) = key \% 7$ ，采用线性探查（线性探测再散列）法解决冲突。将关键字序列 87, 40, 30, 6, 11, 22, 98, 20 依次插入 HT 后，HT 查找失败的平均查找长度是（ ）。
 A. 4 B. 5.25 C. 6 D. 6.29
19. 【2022 统考真题】下列因素中，影响散列（哈希）方法平均查找长度的是（ ）。
 I. 装填因子 II. 散列函数 III. 冲突解决策略
 A. 仅 I、II B. 仅 I、III C. 仅 II、III D. I、II、III

二、综合应用题

01. 若要在散列表中删除一个记录，应如何操作？为什么？
02. 假定把关键字 key 散列到有 n 个表项（从 0 到 $n-1$ 编址）的散列表中。对于下面的每个函数 $H(key)$ (key 为整数)，这些函数能够当作散列函数吗？若能，它是一个好的散列函数吗？说明理由。设函数 $random(n)$ 返回一个 0 到 $n-1$ 之间的随机整数（包括 0 与 $n-1$ 在内）。
- 1) $H(key) = key/n$ 。
 - 2) $H(key) = 1$ 。
 - 3) $H(key) = (key + random(n)) \% n$ 。
 - 4) $H(key) = key \% p(n)$ ；其中 $p(n)$ 是不大于 n 的最大素数。
03. 使用散列函数 $H(key) = key \% 11$ ，把一个整数值转换成散列表下标，现在要把数据 {1, 13, 12, 34, 38, 33, 27, 22} 依次插入散列表。
- 1) 使用线性探测法来构造散列表。
 - 2) 使用链地址法构造散列表。
- 试针对这两种情况，分别确定查找成功所需的平均查找长度，及查找不成功所需的平均查找长度。
04. 已知一组关键字为 {26, 36, 41, 38, 44, 15, 68, 12, 6, 51, 25}，用链地址法解决冲突，假设装填因子 $\alpha = 0.75$ ，散列函数的形式为 $H(key) = key \% P$ ，回答以下问题：
- 1) 构造出散列函数。
 - 2) 分别计算出等概率情况下查找成功和查找失败的平均查找长度（查找失败的计算中只将与关键字的比较次数计算在内即可）。
05. 设散列表为 $HT[0..12]$ ，即表的大小为 $m = 13$ 。现采用双散列法解决冲突，散列函数和再散列函数分别为：

关注公众号【乘龙考研】
一手更新 稳定有保障

$H_0(key) = key \% 13$ 注: % 是求余数运算 ($=MOD$)

$H_i = (H_{i-1} + REV(key+1) \% 11 + 1) \% 13; \quad i = 1, 2, 3, \dots, m-1$

其中, 函数 $REV(x)$ 表示颠倒十进制数 x 的各位, 如 $REV(37) = 73$, $REV(7) = 7$ 等。若插入的关键码序列为 $(2, 8, 31, 20, 19, 18, 53, 27)$, 请回答:

1) 画出插入这 8 个关键码后的散列表。

2) 计算查找成功的平均查找长度 ASL。

06. 【2010 统考真题】将关键字序列 $(7, 8, 30, 11, 18, 9, 14)$ 散列存储到散列表中。散列表的存储空间是一个下标从 0 开始的一维数组, 散列函数为 $H(key) = (key \times 3) \bmod 7$, 处理冲突采用线性探测再散列法, 要求装填(载)因子为 0.7。

1) 请画出所构造的散列表。

2) 分别计算等概率情况下, 查找成功和查找不成功的平均查找长度。

7.5.6 答案与解析

一、单项选择题

01. B

顺序查找可以是顺序存储或链式存储; 折半查找只能是顺序存储且要求关键字有序; 树形查找要求采用树的存储结构, 既可以采用顺序存储也可以采用链式存储; 散列查找中的链地址法解决冲突时, 采用的是顺序存储与链式存储相结合的方式。

02. D

关键字集合与地址集合之间存在对应关系时, 通过散列函数表示这种关系。这样, 查找以计算散列函数而非比较的方式进行查找。

03. D

冲突(碰撞)是不可避免的, 与装填因子无关, 因此需要设计处理冲突的方法, 选项 I 错误。散列查找的思想是计算出散列地址来进行查找, 然后比较关键字以确定是否查找成功, 选项 II 错误。散列查找成功的平均查找长度与装填因子有关, 与表长无关, 选项 III 错误。在开放定址的情形下, 不能随便删除散列表中的某个元素, 否则可能会导致搜索路径被中断(因此通常的做法是在要删除的地方做删除标记, 而不是直接删除), 选项 IV 正确。

04. C

在开放定址法中散列到同一个地址而产生的“堆积”问题, 是同义词冲突的探查序列和非同义词之间不同的探查序列交织在一起, 导致关键字查询需要经过较长的探测距离, 降低了散列的效率。因此要选择好的处理冲突的方法来避免“堆积”。

05. A

利用再散列法处理冲突时, 按一定的距离, 跳跃式地寻找“下一个”空闲位置, 减少了发生聚集的可能, 选项 I 正确。散列地址 i 的关键字, 和为解决冲突形成的某次探测地址为 i 的关键字, 都争夺地址 $i, i+1, \dots$, 因此不一定相邻, 选项 II 错误。选项 III 正确。同义词冲突不等于聚集, 链地址法处理冲突时将同义词放在同一个链表中, 不会引起聚集现象, 选项 IV 错误。

06. A

若有 200 个表项要放入散列表, 采用线性探测法解决冲突, 限定查找成功的平均查找长度不超过 1.5, 则

$$ASL_{\text{成功}} = \frac{1}{2} \left(1 + \frac{1}{1-\alpha} \right) \leqslant 1.5 \Rightarrow \alpha = \frac{200}{m} \leqslant \frac{1}{2} \Rightarrow m \geqslant 400$$

07. D

K 个关键字在依次填入的过程中，只有第一个不会发生冲突，故探测次数为 $(1+2+3+\dots+K) = K(K+1)/2$ ，即选D。

08. C

在散列表中，平均查找长度与装填因子 α 直接相关，表的查找效率不直接依赖于表中已有表项个数 n 或表长 m 。若散列表中存放的记录全部是某个地址的同义词，则平均查找长度为 $O(n)$ 而非 $O(1)$ 。

09. D

聚集是因选取不当的处理冲突的方法，而导致不同关键字的元素对同一散列地址进行争夺的现象。用线性再探测法时，容易引发聚集现象。

10. D

由散列函数计算可知，14, 1, 27, 79散列后的地址都是1，所以有4个记录。

11. A, B

因为在链地址法中，映射到同一地址的关键字都会链到与此地址相对应的链表上，所以探测过程一定是在此链表上进行的，从而这些位置上的关键字均为同义词；但在线性探测法中出现两个同义关键字时，会把该关键字对应地址的下一个地址也占用掉，两个地址分别记为Addr、Addr+1，查找一个满足 $H(key)=Addr+1$ 的关键字key时，显然首次探测到的不是key的同义词。

12. A, C

H 的取值有17种可能，对应到不同的链表中，所以链表的个数应为17。由于 $H(key)$ 的取值范围是0~16，所以数组下标为0~16。

13. A

线性探测法的公式为 $H_i=(H(key)+d_i)\%m$ ，其中 $d_i=1, 2, 3, \dots, m-1$ 。 $H(49)=49\%11=5$ ，发生冲突； $H_1=(H(49)+1)\%14=6$ ，发生冲突； $H_2=(H(49)+2)\%14=7$ ，发生冲突； $H_3=(H(49)+3)\%14=8$ ，无冲突。选A。

14. C

由于散列函数的选取，仍然有可能产生地址冲突，冲突不能绝对地避免。

15. D

散列表的查找效率取决于：散列函数、处理冲突的方法和装填因子。显然，冲突的产生概率与装填因子（即表中记录数与表长之比）的大小成正比，选项I与题意相反。选项II显然正确。采用合适的冲突处理方法可避免聚集现象，也将提高查找效率，选项III正确。例如，用链地址法处理冲突时不存在聚集现象，用线性探测法处理冲突时易引起聚集现象。

16. D

产生堆积现象，即产生了冲突，它对存储效率、散列函数和装填因子均不会有影响，而平均查找长度会因为堆积现象而增大，选D。

17. C

根据题意，得到的HT如下：

0	1	2	3	4	5	6
	22	43	15			

$$ASL_{\text{成功}} = (1 + 2 + 3)/3 = 2。$$

18. C

采用线性探查法计算每个关键字的存放情况如下表所示。

散列地址	0	1	2	3	4	5	6	7	8	9	10
关键字	98	22	30	87	11	40	6	20			

由于 $H(key)=0 \sim 6$, 查找失败时可能对应的地址有 7 个, 对于计算出地址为 0 的关键字 key_0 , 只有比较完 0~8 号地址后才能确定该关键字不在表中, 比较次数为 9; 对于计算出地址为 1 的关键字 key_1 , 只有比较完 1~8 号地址后才能确定该关键字不在表中, 比较次数为 8; 以此类推。需要特别注意的是, 散列函数不可能计算出地址 7, 因此有

$$ASL_{\text{失败}} = (9 + 8 + 7 + 6 + 5 + 4 + 3) / 7 = 6$$

19. D

原题再现。填装因子越大, 说明哈希表中存储的元素越满, 发生冲突的可能性就越高, 导致平均查找长度越大。散列函数、冲突解决策略也会影响发生冲突的可能性。说法 I、II、III 都正确。

二、综合应用题

01. 【解答】

在散列表中删除一个记录, 在拉链法情况下可以物理地删除。但在开放定址法情况下, 不能物理地删除, 只能做删除标记。该地址可能是该记录的同义词查找路径上的地址, 物理地删除就中断了查找路径, 因为查找时碰到空地址就认为是查找失败。

02. 【解答】

- 1) 不能作为散列函数, 因为 key/n 可能大于 n , 这样就无法找到适合的位置。
- 2) 能够作为散列函数, 但不是一个好的散列函数, 因为所有关键字都映射到同一位置, 造成大量的冲突机会。
- 3) 不能当作散列函数, 因为该函数的返回值不确定, 这样无法进行正常的查找。
- 4) 能够作为散列函数, 是一个好的散列函数。

03. 【解答】

由散列函数可知散列地址的范围为 0~10。

采用线性探测法构造散列表时, 首先应计算出关键字对应的散列地址, 然后检查散列表中对应的地址是否已经有元素。若没有元素, 则直接将该关键字放入散列表对应的地址中; 若有元素, 则采用线性探测的方法查找下一个地址, 从而决定该关键字的存放位置。

采用链地址法构造散列表时, 在直接计算出关键字对应的散列地址后, 将关键字结点插入此散列地址所在的链表。

具体解答如下。

1) 线性探测法。

$H(1)=1$, 无冲突, 地址 1 存放关键字 1。 $H(13)=2$, 无冲突, 地址 2 存放关键字 13。 $H(12)=1$, 发生冲突, 根据线性探测法: $H_1=2$, 发生冲突, 继续探测 $H_2=3$, 无冲突, 于是 12 存放在地址为 3 的表项中。 $H(34)=1$, 发生冲突, 根据线性探测法: $H_1=2$, 发生冲突, $H_2=3$, 发生冲突, $H_3=4$, 没有冲突, 于是 34 存放在地址为 4 的表项中。

同理, 可以计算其他的数据存放情况, 最后结果如下表所示。

散列地址	0	1	2	3	4	5	6	7	8	9	10
关键字	33	1	13	12	34	38	27	22			
冲突次数	0	0	0	2	3	0	1	7			

下面计算平均查找长度：

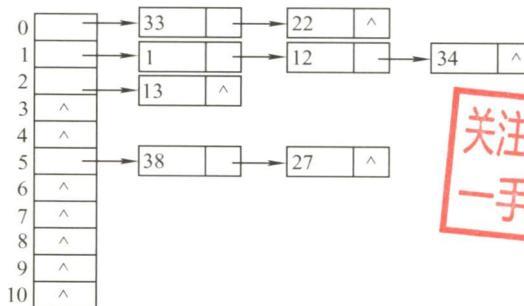
查找成功时，显然查找每个元素的概率都是 $1/8$ 。对于 33，由于冲突次数为 0，所以仅需 1 次比较便可查找成功；对于 22，由于计算出的地址为 0，但需要 8 次比较才能查找成功，所以 22 的查找长度为 8；其他元素的分析类似。因此有

$$ASL_{\text{成功}} = (1 + 1 + 1 + 3 + 4 + 1 + 2 + 8)/8 = 21/8$$

查找失败时，由于 $H(key) = 0 \sim 10$ ，因此对每个位置查找的概率都是 $1/11$ ，对于计算出的地址为 0 的关键字 key0，只有探测完 0~8 号地址后才能确定该元素不在表中，比较次数为 9；对于计算出的地址为 1 的关键字 key1，只有探测完 1~8 号地址后，才能确定该元素不在表中，比较次数为 8，以此类推。而对于计算出的地址为 8, 9, 10 的关键字，这些单元中没有存放元素，所以只需比较 1 次便可确定查找失败，因此有

$$ASL_{\text{失败}} = (9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + 1 + 1)/11 = 47/11$$

2) 链地址法构造的表如下：



在链地址表中查找成功时，查找关键字为 33 的记录需进行 1 次比较，查找关键字为 22 的记录需进行 2 次比较，以此类推。因此有

$$ASL_{\text{成功}} = (1 \times 4 + 2 \times 3 + 3)/8 = 13/8$$

查找失败时，对于地址 0，比较 3 次后确定元素不在表中（空指针算 1 次），所以其查找长度为 3；对于地址 1，其查找长度为 4；对于地址 2，查找长度为 2；以此类推。因此有

$$ASL_{\text{失败}} = (3 + 4 + 2 + 1 + 1 + 3 + 1 + 1 + 1 + 1 + 1)/11 = 19/11$$

值得注意的是，求查找失败的平均查找长度有两种观点：其一，认为比较到空结点才算失败，所以比较次数等于冲突次数加 1；其二，认为只有与关键字的比较才算比较次数。

04. 【解答】

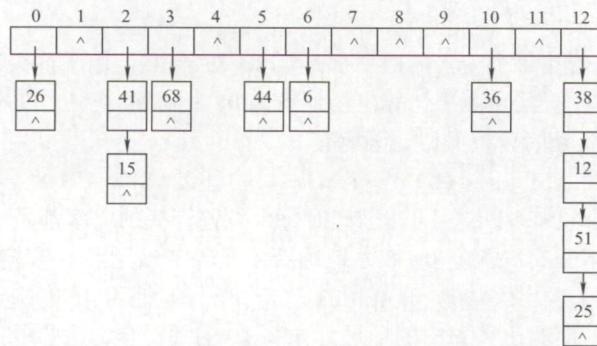
由装填因子的计算公式 $\alpha = n/N$ (n 为关键字个数， N 为表长)，不难得出表长，而根据散列函数的选择要求， P 应该取不大于表长的最大素数，从而可以确定 P 的大小，也就构造出了散列函数。这里采用链地址法解决冲突，两种情况下的平均查找长度的计算过程与上一题完全相似。

具体解答如下。

- 1) 由 $\alpha = n/N$ 得 $N = n/\alpha$ ，由于 N 为整数，故应该向上取整，即 $N = \lceil n/\alpha \rceil = 15$ ，从而 $P = 13$ 。
因此散列函数为 $H(key) = key \% 13$ 。
- 2) 由 1) 求出的散列函数，计算各关键字对应的散列地址如下表所示。

关键字	26	36	41	38	44	15	68	12	6	51	25
散列地址	0	10	2	12	5	2	3	12	6	12	12

由此构造的链地址法处理冲突的散列表为



由上图不难计算出

$$ASL_{\text{成功}} = (1 \times 7 + 2 \times 2 + 3 \times 1 + 4 \times 1) / 11 = 18/11$$

$$ASL_{\text{失败}} = (1 + 0 + 2 + 1 + 0 + 1 + 1 + 0 + 0 + 0 + 1 + 0 + 4) / 13 = 11/13$$

05. 【解答】

- 1) $H_0(2)=2, H_0(8)=8, H_0(31)=5, H_0(20)=7, H_0(19)=6$, 没有冲突。 $H_0(18)=5$, 发生冲突。
 $H_1(18)=(H_0(18)+REV(18+1)\%11+1)\%13=(5+3+1)\%13=9$, 没有冲突。 $H_0(53)=1$, 没有冲突。 $H_0(27)=1$, 发生冲突, $H_1(27)=(H_0(27)+REV(27+1)\%11+1)\%13=(1+5+1)\%13=7$, 发生冲突, $H_2(27)=(H_1(27)+REV(27+1)\%11+1)\%13=0$, 没有冲突。构造的散列表如下:

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键字	27	53	2			31	19	20	8	18			
比较次数	3	1	1			1	1	1	1	2			

- 2) 由 1) 中散列表的构造过程, 各个关键字查找成功的比较次数如上表所示, 故有

$$ASL_{\text{成功}} = (3 + 1 + 1 + 1 + 1 + 1 + 1 + 2) / 8 = 11/8$$

06. 【解答】

- 1) 由装填因子 0.7 和数据总数 7, 得一维数组大小为 $7/0.7 = 10$, 数组下标为 0~9。所构造的散列函数值如下所示:

key	7	8	30	11	18	9	14
H(key)	0	3	6	5	5	6	0

采用线性探测再散列法处理冲突, 所构造的散列表为

地址	0	1	2	3	4	5	6	7	8	9
关键字	7	14		8		11	30	18	9	

- 2) 查找成功时, 在等概率情况下, 查找每个表中元素的概率是相等的。因此, 根据表中元素的个数来计算平均查找长度, 各关键字的比较次数如下所示:

key	7	8	30	11	18	9	14
次数	1	1	1	1	3	3	2

故 $ASL_{\text{成功}} = \text{查找次数}/\text{元素个数} = (1 + 2 + 1 + 1 + 1 + 3 + 3) / 7 = 12/7$ 。

在计算查找失败时的平均查找长度时, 要特别注意防止思维定式, 在查找失败的情况下既不是根据表中的元素个数, 也不是根据表长来计算平均查找长度的。

查找失败时，在等概率情况下，经过散列函数计算后只可能映射到表中的0~6位置，且映射到0~6中任意一个位置的概率是相等的。因此，是根据散列函数(mod后面的数字)来计算平均查找长度的。在等概率情况下，查找失败的比较次数如下所示：

H(key)	0	1	2	3	4	5	6
次数	3	2	1	2	1	5	4

故 $ASL_{\text{不成功}} = \text{查找次数}/\text{散列后的地址个数} = (3 + 2 + 1 + 2 + 1 + 5 + 4)/7 = 18/7$ 。

归纳总结

本章的核心考查点是求平均查找长度(ASL)，以度量各种查找算法的性能。查找算法本身依托于查找结构，查找结构又是由相同数据类型的记录或结点构成的，故最终落脚于数据结构类型的区别。

不管是何种查找算法，其平均查找长度的计算公式都是一样的。

$$\text{查找成功的平均查找长度 } ASL_{\text{成功}} = \sum_{i=1}^n p_i c_i.$$

$$\text{查找失败的平均查找长度 } ASL_{\text{不成功}} = \sum_{j=0}^n q_j c_j.$$

关注公众号【乘龙考研】
一手更新 稳定有保障

设一个查找集合中已有 n 个数据元素，每个元素的查找概率为 p_i ，查找成功的数据比较次数为 c_i ($i = 1, 2, \dots, n$)；不在此集合中的数据元素分布在由这 n 个元素的间隔构成的 $n+1$ 个子集合内，每个子集合元素的查找概率为 q_j ，查找不成功的数据比较次数为 c_j ($j = 0, 1, \dots, n$)。因此，对某一特定查找算法的查找成功的 $ASL_{\text{成功}}$ 和查找失败的 $ASL_{\text{不成功}}$ ，是综合考虑还是分开考虑呢？

若综合考虑，即 $\sum_{i=1}^n p_i + \sum_{j=0}^n q_j = 1$ ，若所有元素查找概率相等，则有 $p_i = q_j = \frac{1}{2n+1}$ ；若分开

考虑，即 $\sum_{i=1}^n p_i = 1$ ， $\sum_{j=0}^n q_j = 1$ ，若所有元素查找概率相等，则有 $p_i = \frac{1}{n}$ ， $q_j = \frac{1}{n+1}$ 。

虽然综合考虑更为理想，但在实际应用中多数是分开考虑的，因为对于查找不到的情况，很多场合下没有明确给出，往往会被忽略掉。不过读者仍要注意的是，这两种考虑的计算结果是不同的，考试中一定要仔细阅读题目的要求，以免失误。

思维拓展

本章介绍了几种基本的查找算法，在实际中又会碰到怎样的查找问题呢？

题目：数组中有一个数字出现的次数超过了数组长度的一半，请找出这个数字。读者也许会想到先进行排序，位于位置 $(n+1)/2$ 的数即为要找的数，这样最小时间复杂度就为 $O(n \log_2 n)$ ；若进行散列查找，数字的范围又未知，那么应如何将时间复杂度控制在 $O(n)$ 内呢？

(提示：出现的次数超过数组长度的一半，表明这个数字出现的次数比其他数字出现的次数的总和还多。所以我们可以考虑每次删除两个不同的数，则在剩下的数中，待找数字出现的次数仍然超过总数的一半。通过不断重复这个过程，不断排除其他的数字，最终剩下的都为同一个数字，即为要找的数字。)

第 8 章 排 序

关注公众号【乘龙考研】
一手更新 稳定有保障

【考纲内容】

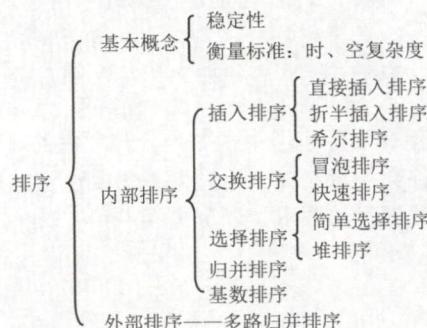
- (一) 排序的基本概念
- (二) 插入排序
直接插入排序；折半插入排序；希尔排序（shell sort）
- (三) 交换排序
冒泡排序（bubble sort）；快速排序
- (四) 选择排序
简单选择排序；堆排序
- (五) 二路归并排序（merge sort）
- (六) 基数排序
- (七) 外部排序
- (八) 排序算法的分析和应用

兑换后



看视频讲解

【知识框架】



【复习提示】

堆排序、快速排序和归并排序是本章的重难点。读者应深入掌握各种排序算法的思想、排序过程（能动手模拟）和特征（初态的影响、复杂度、稳定性、适用性等），通常以选择题的形式考查不同算法之间的对比。此外，对于一些常用排序算法的关键代码，要达到熟练编写的程度；看到某特定序列，读者应具有选择最优排序算法（根据排序算法特征）的能力。

8.1 排序的基本概念

8.1.1 排序的定义

排序，就是重新排列表中的元素，使表中的元素满足按关键字有序的过程。为了查找方便，通常希望计算机中的表是按关键字有序的。排序的确切定义如下：

输入： n 个记录 R_1, R_2, \dots, R_n ，对应的关键字为 k_1, k_2, \dots, k_n 。

输出：输入序列的一个重排 R'_1, R'_2, \dots, R'_n ，使得 $k'_1 \leq k'_2 \leq \dots \leq k'_n$ （其中“ \leq ”可以换成其他的比较大小的符号）。

算法的稳定性。若待排序表中有两个元素 R_i 和 R_j ，其对应的关键字相同即 $\text{key}_i = \text{key}_j$ ，且在排序前 R_i 在 R_j 的前面，若使用某一排序算法排序后， R_i 仍然在 R_j 的前面，则称这个排序算法是稳定的，否则称排序算法是不稳定的。需要注意的是，算法是否具有稳定性并不能衡量一个算法的优劣，它主要是对算法的性质进行描述。如果待排序表中的关键字不允许重复，则排序结果是唯一的，那么选择排序算法时的稳定与否就无关紧要。

注意：对于不稳定的排序算法，只需举出一组关键字的实例，说明它的不稳定性即可。

在排序过程中，根据数据元素是否完全在内存中，可将排序算法分为两类：①内部排序，是指在排序期间元素全部存放在内存中的排序；②外部排序，是指在排序期间元素无法全部同时存放在内存中，必须在排序的过程中根据要求不断地在内、外存之间移动的排序。

一般情况下，内部排序算法在执行过程中都要进行两种操作：比较和移动。通过比较两个关键字的大小，确定对应元素的前后关系，然后通过移动元素以达到有序。当然，并非所有的内部排序算法都要基于比较操作，事实上，基数排序就不基于比较。

每种排序算法都有各自的优缺点，适合在不同的环境下使用，就其全面性能而言，很难提出一种被认为是最好的算法。通常可以将排序算法分为插入排序、交换排序、选择排序、归并排序和基数排序五大类，后面几节会分别进行详细介绍。内部排序算法的性能取决于算法的时间复杂度和空间复杂度，而时间复杂度一般是由比较和移动的次数决定的。

注意：大多数的内部排序算法只适用于顺序存储的线性表。

8.1.2 本节试题精选

一、单项选择题

01. 下述排序方法中，不属于内部排序方法的是（ ）。
 - A. 插入排序
 - B. 选择排序
 - C. 拓扑排序
 - D. 冒泡排序
02. 排序算法的稳定性是指（ ）。
 - A. 经过排序后，能使关键字相同的元素保持原顺序中的相对位置不变
 - B. 经过排序后，能使关键字相同的元素保持原顺序中的绝对位置不变
 - C. 排序算法的性能与被排序元素个数关系不大
 - D. 排序算法的性能与被排序元素的个数关系密切
03. 下列关于排序的叙述中，正确的是（ ）。
 - A. 稳定的排序方法优于不稳定的排序方法

关注公众号【乘龙考研】
一手更新 稳定有保障

- B. 对同一线性表使用不同的排序方法进行排序，得到的排序结果可能不同
 - C. 排序方法都是在顺序表上实现的，在链表上无法实现排序方法
 - D. 在顺序表上实现的排序方法在链表上也可以实现
04. 对任意 7 个关键字进行基于比较的排序，至少要进行（ ）次关键字之间的两两比较。
- A. 13 B. 14 C. 15 D. 6

8.1.3 答案与解析

一、单项选择题

01. C

拓扑排序是将有向图中所有结点排成一个线性序列，虽然也是在内存中进行的，但它不属于我们这里所提到的内部排序范畴，也不满足前面排序的定义。

02. A

注意，这里的绝对位置是指若在排序前元素 R 在位置 i ，则绝对位置就是 i ，即排序后 R 的位置不发生变化，显然 B 是不对的。C、D 与题目要求无关。

03. B

算法的稳定性与算法优劣无关，A 排除。使用链表也可以进行排序，只是有些排序算法不再适用，因为这时定位元素只能顺序逐链查找，如折半插入排序。

04. A

对于任意序列进行基于比较的排序，求至少的比较次数应考虑最坏情况。对任意 n 个关键字排序的比较次数至少为 $\lceil \log_2(n!) \rceil$ 。将 $n=7$ 代入公式，答案为 13。

上述公式证明如下（仅供有兴趣的同学参考）：在基于比较的排序方法中，每次比较两个关键字后，仅出现两种可能的转移。假设整个排序过程至少需要做 t 次比较，则显然会有 2^t 种情况。由于 n 个记录共有 $n!$ 种不同的排列，因而必须有 $n!$ 种不同的比较路径，于是有 $2^t \geq n!$ ，即 $t \geq \log_2(n!)$ 。考虑到 t 为整数，故为 $\lceil \log_2(n!) \rceil$ 。

关注公众号【乘龙考研】
一手更新 稳定有保障

8.2 插入排序

插入排序是一种简单直观的排序方法，其基本思想是每次将一个待排序的记录按其关键字大小插入前面已排好序的子序列，直到全部记录插入完成。由插入排序的思想可以引申出三个重要的排序算法：直接插入排序、折半插入排序和希尔排序。

8.2.1 直接插入排序^①

根据上面的插入排序思想，不难得出一种最简单也最直观的直接插入排序算法。假设在排序过程中，待排序表 $L[1...n]$ 在某次排序过程中的某一时刻状态如下：

有序序列 $L[1...i-1]$	$L(i)$	无序序列 $L[i+1...n]$
-------------------	--------	-------------------

要将元素 $L(i)$ 插入已有序的子序列 $L[1...i-1]$ ，需要执行以下操作（为避免混淆，下面用 $L[]$ 表示一个表，而用 $L()$ 表示一个元素）：

① 凡在书中未加特殊说明的，通常默认排序结果为非递减有序序列。

- 1) 查找出 $L(i)$ 在 $L[1 \dots i-1]$ 中的插入位置 k 。
- 2) 将 $L[k \dots i-1]$ 中的所有元素依次后移一个位置。
- 3) 将 $L(i)$ 复制到 $L(k)$ 。

为了实现对 $L[1 \dots n]$ 的排序, 可以将 $L(2) \sim L(n)$ 依次插入前面已排好序的子序列, 初始 $L[1]$ 可以视为是一个已排好序的子序列。上述操作执行 $n-1$ 次就能得到一个有序的表。插入排序在实现上通常采用就地排序 (空间复杂度为 $O(1)$), 因而在从后向前的比较过程中, 需要反复把已排序元素逐步向后挪位, 为新元素提供插入空间。

下面是直接插入排序的代码, 其中再次用到了我们前面提到的“哨兵” (作用相同)。

```
void InsertSort(ElemType A[], int n) {
    int i, j;
    for(i=2; i<=n; i++)           //依次将 A[2]~A[n] 插入前面已排序序列
        if(A[i]<A[i-1]) {         //若 A[i] 关键码小于其前驱, 将 A[i] 插入有序表
            A[0]=A[i];             //复制为哨兵, A[0] 不存放元素
            for(j=i-1; A[0]<A[j]; --j) //从后往前查找待插入位置
                A[j+1]=A[j];         //向后挪位
            A[j+1]=A[0];           //复制到插入位置
        }
}
```

假定初始序列为 49, 38, 65, 97, 76, 13, 27, 49, 初始时 49 可以视为一个已排好序的子序列, 按照上述算法进行直接插入排序的过程如图 8.1 所示, 括号内是已排好序的子序列。

[初始关键字]:	(49)	38	65	97	76	13	27	<u>49</u>
$i=2:$	(38)	(38 49)	65	97	76	13	27	<u>49</u>
$i=3:$	(65)	(38 49	65)	97	76	13	27	<u>49</u>
$i=4:$	(97)	(38 49	65	97)	76	13	27	<u>49</u>
$i=5:$	(76)	(38 49	65	76	97)	13	27	<u>49</u>
$i=6:$	(13)	(13 38	49	65	76	97)	27	<u>49</u>
$i=7:$	(27)	(13 27	38	49	65	76	97)	<u>49</u>
$i=8:$	<u>(49)</u>	(13 27	38	49	<u>49</u>	65	76	97)

↑ 监视哨 L.R[0]

图 8.1 直接插入排序示例

关注公众号【乘龙考研】
一手更新 稳定有保障

直接插入排序算法的性能分析如下:

空间效率: 仅使用了常数个辅助单元, 因而空间复杂度为 $O(1)$ 。

时间效率: 在排序过程中, 向有序子表中逐个地插入元素的操作进行了 $n-1$ 趟, 每趟操作都分为比较关键字和移动元素, 而比较次数和移动次数取决于待排序表的初始状态。

在最好情况下, 表中元素已经有序, 此时每插入一个元素, 都只需比较一次而不用移动元素, 因而时间复杂度为 $O(n)$ 。

在最坏情况下, 表中元素顺序刚好与排序结果中的元素顺序相反 (逆序), 总的比较次数达到最大, 总的移动次数也达到最大, 总的时间复杂度为 $O(n^2)$ 。

平均情况下, 考虑待排序表中元素是随机的, 此时可以取上述最好与最坏情况的平均值作为平均情况下的时间复杂度, 总的比较次数与总的移动次数均约为 $n^2/4$ 。

因此，直接插入排序算法的时间复杂度为 $O(n^2)$ 。

稳定性：由于每次插入元素时总是从后向前先比较再移动，所以不会出现相同元素相对位置发生变化的情况，即直接插入排序是一个稳定的排序方法。

适用性：直接插入排序算法适用于顺序存储和链式存储的线性表。为链式存储时，可以从前往后查找指定元素的位置。

8.2.2 折半插入排序

从直接插入排序算法中，不难看出每趟插入的过程中都进行了两项工作：①从前面的有序子表中查找出待插入元素应该被插入的位置；②给插入位置腾出空间，将待插入元素复制到表中的插入位置。注意到在该算法中，总是边比较边移动元素。下面将比较和移动操作分离，即先折半查找出元素的待插入位置，然后统一地移动待插入位置之后的所有元素。当排序表为顺序表时，可以对直接插入排序算法做如下改进：由于是顺序存储的线性表，所以查找有序子表时可以用折半查找来实现。确定待插入位置后，就可统一地向后移动元素。算法代码如下：

```
void InsertSort(ElemType A[], int n) {
    int i, j, low, high, mid;
    for (i=2; i<=n; i++) {           //依次将 A[2]~A[n] 插入前面的已排序序列
        A[0]=A[i];                  //将 A[i] 暂存到 A[0]
        low=1; high=i-1;             //设置折半查找的范围
        while (low<=high) {         //折半查找(默认递增有序)
            mid=(low+high)/2;       //取中间点
            if (A[mid]>A[0]) high=mid-1; //查找左半子表
            else low=mid+1;         //查找右半子表
        }
        for (j=i-1; j>=high+1; --j)
            A[j+1]=A[j];           //统一后移元素，空出插入位置
        A[high+1]=A[0];            //插入操作
    }
}
```

从上述算法中，不难看出折半插入排序仅减少了比较元素的次数，约为 $O(n \log_2 n)$ ，该比较次数与待排序表的初始状态无关，仅取决于表中的元素个数 n ；而元素的移动次数并未改变，它依赖于待排序表的初始状态。因此，折半插入排序的时间复杂度仍为 $O(n^2)$ ，但对于数据量不很大的排序表，折半插入排序往往能表现出很好的性能。折半插入排序是一种稳定的排序方法。

8.2.3 希尔排序

从前面的分析可知，直接插入排序算法的时间复杂度为 $O(n^2)$ ，但若待排序列为“正序”时，其时间效率可提高至 $O(n)$ ，由此可见它更适用于基本有序的排序表和数据量不大的排序表。希尔排序正是基于这两点分析对直接插入排序进行改进而得来的，又称缩小增量排序。

希尔排序的基本思想是：先将待排序表分割成若干形如 $L[i, i+d, i+2d, \dots, i+kd]$ 的“特殊”子表，即把相隔某个“增量”的记录组成一个子表，对各个子表分别进行直接插入排序，当整个表中的元素已呈“基本有序”时，再对全体记录进行一次直接插入排序。

希尔排序的过程如下：先取一个小于 n 的步长 d_1 ，把表中的全部记录分成 d_1 组，所有距离为 d_1 的倍数的记录放在同一组，在各组内进行直接插入排序；然后取第二个步长 $d_2 < d_1$ ，重复上述过程，直到所取到的 $d_t = 1$ ，即所有记录已放在同一组中，再进行直接插入排序，由于此时已经

具有较好的局部有序性，故可以很快得到最终结果。到目前为止，尚未求得一个最好的增量序列。仍以 8.2.1 节的关键字为例，假定第一趟取增量 $d_1 = 5$ ，将该序列分成 5 个子序列，即图中第 2 行至第 6 行，分别对各子序列进行直接插入排序，结果如第 7 行所示；假定第二趟取增量 $d_2 = 3$ ，分别对 3 个子序列进行直接插入排序，结果如第 11 行所示；最后对整个序列进行一趟直接插入排序，整个排序过程如图 8.2 所示。

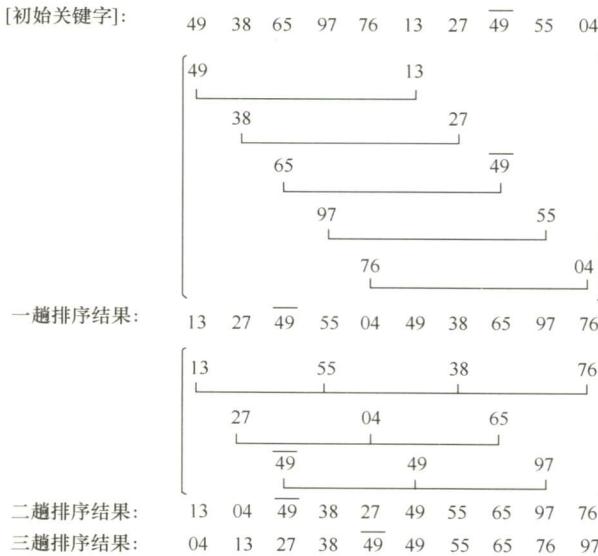


图 8.2 希尔排序示例

希尔排序算法的代码如下：

```
void ShellSort(ElemType A[], int n) {
    //A[0]只是暂存单元，不是哨兵，当 j<=0 时，插入位置已到
    int dk, i, j;
    for(dk=n/2; dk>=1; dk=dk/2)           //增量变化（无统一规定）
        for(i=dk+1; i<=n; ++i)
            if(A[i]<A[i-dk]) {             //需将 A[i] 插入有序增量子表
                A[0]=A[i];                  //暂存在 A[0]
                for(j=i-dk; j>0 && A[0]<A[j]; j-=dk)
                    A[j+dk]=A[j];          //记录后移，查找插入的位置
                A[j+dk]=A[0];              //插入
            } //if
    }
}
```

希尔排序算法的性能分析如下：

空间效率：仅使用了常数个辅助单元，因而空间复杂度为 $O(1)$ 。

时间效率：由于希尔排序的时间复杂度依赖于增量序列的函数，这涉及数学上尚未解决的难题，所以其时间复杂度分析比较困难。当 n 在某个特定范围时，希尔排序的时间复杂度约为 $O(n^{1.3})$ 。在最坏情况下希尔排序的时间复杂度为 $O(n^2)$ 。

稳定性：当相同关键字的记录被划分到不同的子表时，可能会改变它们之间的相对次序，因此希尔排序是一种不稳定的排序方法。例如，图 8.2 中 49 与 $\overline{49}$ 的相对次序已发生了变化。

适用性：希尔排序算法仅适用于线性表为顺序存储的情况。

8.2.4 本节试题精选

一、单项选择题

01. 对 5 个不同的数据元素进行直接插入排序，最多需要进行的比较次数是（ ）。
- A. 8 B. 10 C. 15 D. 25
02. 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（ ）。
- A. 直接插入排序 B. 简单选择排序 C. 快速排序 D. 归并排序
03. 对有 n 个元素的顺序表采用直接插入排序算法进行排序，在最坏情况下所需的比较次数是（ ）；在最好情况下所需的比较次数是（ ）。
- A. $n-1$ B. $n+1$ C. $n/2$ D. $n(n-1)/2$
04. 数据序列 {8, 10, 13, 4, 6, 7, 22, 2, 3} 只能是（ ）两趟排序后的结果。
- A. 简单选择排序 B. 起泡排序 C. 直接插入排序 D. 堆排序
05. 用直接插入排序算法对下列 4 个表进行（从小到大）排序，比较次数最少的是（ ）。
- A. 94, 32, 40, 90, 80, 46, 21, 69 B. 21, 32, 46, 40, 80, 69, 90, 94
C. 32, 40, 21, 46, 69, 94, 90, 80 D. 90, 69, 80, 46, 21, 32, 94, 40
06. 在下列算法中，（ ）算法可能出现下列情况：在最后一趟开始之前，所有元素都不在最终位置上。
- A. 堆排序 B. 冒泡排序 C. 直接插入排序 D. 快速排序
07. 希尔排序属于（ ）。
- A. 插入排序 B. 交换排序 C. 选择排序 D. 归并排序
08. 对序列 {15, 9, 7, 8, 20, -1, 4} 用希尔排序方法排序，经一趟后序列变为 {15, -1, 4, 8, 20, 9, 7}，则该次采用的增量是（ ）。
- A. 1 B. 4 C. 3 D. 2
09. 若对于第 8 题中的序列，经一趟排序后序列变成 {9, 15, 7, 8, 20, -1, 4}，则采用的是下列的（ ）。
- A. 选择排序 B. 快速排序 C. 直接插入排序 D. 冒泡排序
10. 对序列 {98, 36, -9, 0, 47, 23, 1, 8, 10, 7} 采用希尔排序，下列序列（ ）是增量为 4 的一趟排序结果。
- A. {10, 7, -9, 0, 47, 23, 1, 8, 98, 36} B. {-9, 0, 36, 98, 1, 8, 23, 47, 7, 10}
C. {36, 98, -9, 0, 23, 47, 1, 8, 7, 10} D. 以上都不对
11. 折半插入排序算法的时间复杂度为（ ）。
- A. $O(n)$ B. $O(n\log_2 n)$ C. $O(n^2)$ D. $O(n^3)$
12. 有些排序算法在每趟排序过程中，都会有一个元素被放置到其最终位置上，（ ）算法不会出现此种情况。
- A. 希尔排序 B. 堆排序 C. 冒泡排序 D. 快速排序
13. 以下排序算法中，不稳定的是（ ）。
- A. 冒泡排序 B. 直接插入排序 C. 希尔排序 D. 归并排序
14. 以下排序算法中，稳定的是（ ）。
- A. 快速排序 B. 堆排序 C. 直接插入排序 D. 简单选择排序
15. 【2009 统考真题】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（ ）。

- A. 冒泡排序 B. 插入排序 C. 选择排序 D. 2路归并排序
16. 【2012 统考真题】对同一待排序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（ ）。
- A. 排序的总趟数 B. 元素的移动次数
C. 使用辅助空间的数量 D. 元素之间的比较次数
17. 【2014 统考真题】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（ ）。
- A. 2 B. 3 C. 4 D. 5
18. 【2015 统考真题】希尔排序的组内排序采用的是（ ）。
- A. 直接插入排序 B. 折半插入排序 C. 快速排序 D. 归并排序
19. 【2018 统考真题】对初始数据序列(8, 3, 9, 11, 2, 1, 4, 7, 5, 10, 6)进行希尔排序。若第一趟排序结果为(1, 3, 7, 5, 2, 6, 4, 9, 11, 10, 8)，第二趟排序结果为(1, 2, 6, 4, 3, 7, 5, 8, 11, 10, 9)，则两趟排序采用的增量（间隔）依次是（ ）。
- A. 3, 1 B. 3, 2 C. 5, 2 D. 5, 3

二、综合应用题

01. 给出关键字序列{4, 5, 1, 2, 6, 3}的直接插入排序过程。
02. 给出关键字序列{50, 26, 38, 80, 70, 90, 8, 30, 40, 20}的希尔排序过程（取增量序列为 $d = \{5, 3, 1\}$ ，排序结果为从小到大排列）。

8.2.5 答案与解析

一、单项选择题

01. B

直接插入排序在最坏的情况下要做 $n(n-1)/2$ 次关键字的比较，当 $n=5$ 时，关键字的比较次数为 10。注意本题不考虑与哨兵的比较。

02. A

由于这里的序列基本有序，使用直接插入排序算法的时间复杂度接近 $O(n)$ ，而使用其他算法的时间复杂度均大于 $O(n)$ 。

03. D、A

待排序表为反序时，直接插入排序需要进行 $n(n-1)/2$ 次比较（从前往后依次需要比较 1, 2, …, $n-1$ 次）；待排序表为正序时，只需进行 $n-1$ 次比较。同样注意不考虑与哨兵的比较。

04. C

冒泡排序和选择排序经过两趟排序后，应该有两个最大（或最小）元素放在其最终位置；插入排序经过两趟排序后，前 3 个元素应该是局部有序的。只可能选择插入排序。

注意：在排序过程中，每趟都能确定一个元素在其最终位置的有冒泡排序、简单选择排序、堆排序、快速排序，其中前三者能形成全局有序的子序列，后者能确定枢轴元素的最终位置。

05. B

首先，越接近正序的序列，比较次数应是越少的；而越接近逆序，比较次数越多。不难得出选项 B 和 C 是比较接近正序的，然后分别判断两个序列的比较次数，以选项 B 为例：第一趟，插入 32，比较 1 次；第二趟，插入 46，比较 1 次；第三趟，插入 40，由于 40 比 46 小但比 32 大，所以比较 2 次；第四趟，插入 80，比较 1 次；第五趟，插入 69，比较 2 次……共比较 9 次。同

关注公众号【乘龙考研】
一手更新 稳定有保障

理求出选项 C 的比较次数为 11 次。故选择选项 B。

06. C

在直接插入排序中，若待排序列中的最后一个元素应插入表中的第一个位置，则前面的有序子序列中的所有元素都不在最终位置上。

07. A

希尔排序是对直接插入排序算法改进后提出来的，本质上仍属于插入排序的范畴。

08. B

希尔排序将序列分成若干组，记录只在组内进行交换。由观察可知，经过一趟后 9 和 -1 交换，7 和 4 交换，可知增量为 4。

09. C

前两个元素局部已经有序，很明显一趟直接插入排序算法有效。再排除其他算法即可。

10. A

增量为 4 意味着所有相距为 4 的记录构成一组，然后在组内进行直接插入排序，经观察，只有选项 A 满足要求。

11. C

虽然折半插入排序是对直接插入排序的改进，但它改进的只是比较的次数，而移动次数未发生变化，时间复杂度仍为 $O(n^2)$ 。

12. A

由于希尔排序是基于插入排序算法而提出的，它不一定在每趟排序过程后将某一元素放置到最终位置上。

13. C

希尔排序是一种复杂的插入排序方法，它是一种不稳定的排序方法。

14. C

基于插入、交换、选择的三类排序方法中，通常简单方法是稳定的（直接插入、折半插入、冒泡），但有一个例外就是简单选择，复杂方法都是不稳定的（希尔、快排、堆排）。

15. B

每趟冒泡和选择排序后，总会有一个元素被放置在最终位置上。显然，这里 {11, 12} 和 {4, 5} 所处的位置并不是最终位置，因此不可能是冒泡和选择排序。2 路归并算法经过第二趟后应该是每 4 个元素有序的，但 {11, 12, 13, 7} 并非有序，因此也不可能 2 路归并排序。

16. D

折半插入排序与直接插入排序都将待插入元素插入前面的有序子表，区别是：确定当前记录在前面有序子表中的位置时，直接插入排序采用顺序查找法，而折半插入排序采用折半查找法。排序的总趟数取决于元素个数 n ，两者都是 $n-1$ 趟。元素的移动次数都取决于初始序列，两者相同。使用辅助空间的数量也都是 $O(1)$ 。折半插入排序的比较次数与序列初态无关，为 $O(n \log_2 n)$ ；而直接插入排序的比较次数与序列初态有关，为 $O(n) \sim O(n^2)$ 。

17. B

首先，第二个元素为 1，是整个序列中的最小元素，因此可知该希尔排序为从小到大排序。然后考虑增量问题，若增量为 2，则第 1 + 2 个元素 4 明显比第 1 个元素 9 要小，选项 A 排除；若增量为 3，则第 $i, i+3, i+6$ ($i=1, 2, 3$) 个元素都为有序序列，符合希尔排序的定义；若增量为 4，则第 1 个元素 9 比第 1 + 4 个元素 7 要大，选项 C 排除；若增量为 5，则第 1 个元素 9 比第 1 + 5 个元素 8 要大，选项 D 排除，选择选项 B。

18. A

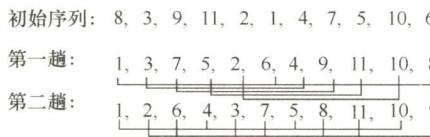
希尔排序的思想是：先将待排元素序列分割成若干子序列（由相隔某个“增量”的元素组成），分别进行直接插入排序，然后依次缩减增量再进行排序，待整个序列中的元素基本有序（增量足够小）时，再对全体元素进行一次直接插入排序。

19. D

如下图所示。

第一趟分组：8, 1, 6; 3, 4; 9, 7; 11, 5; 2, 10; 间隔为 5，排序后组内递增。

第二趟分组：1, 5, 4, 10; 3, 2, 9, 8; 7, 6, 11; 间隔为 3，排序后组内递增。



故答案选 D。

二、综合应用题

01. 【解答】

直接插入排序过程如下。

初始序列：	4, 5, 1, 2, 6, 3
第一趟：	4, 5, 1, 2, 6, 3 (将 5 插入 {4})
第二趟：	1, 4, 5, 2, 6, 3 (将 1 插入 {4, 5})
第三趟：	1, 2, 4, 5, 6, 3 (将 2 插入 {1, 4, 5})
第四趟：	1, 2, 4, 5, 6, 3 (将 6 插入 {1, 2, 4, 5})
第五趟：	1, 2, 3, 4, 5, 6 (将 3 插入 {1, 2, 4, 5, 6})

02. 【解答】

原始序列：	50, 26, 38, 80, 70, 90, 8, 30, 40, 20
第一趟 (增量 5)：	50, 8, 30, 40, 20, 90, 26, 38, 80, 70
第二趟 (增量 3)：	26, 8, 30, 40, 20, 80, 50, 38, 90, 70
第三趟 (增量 1)：	8, 20, 26, 30, 38, 40, 50, 70, 80, 90



8.3 交换排序

所谓交换，是指根据序列中两个元素关键字的比较结果来对换这两个记录在序列中的位置。基于交换的排序算法很多，本书主要介绍冒泡排序和快速排序，其中冒泡排序算法比较简单，一般不会单独考查，通常会重点考查快速排序算法的相关内容。

8.3.1 冒泡排序

冒泡排序的基本思想是：从后往前（或从前往后）两两比较相邻元素的值，若为逆序（即 $A[i-1] > A[i]$ ），则交换它们，直到序列比较完。我们称它为第一趟冒泡，结果是将最小的元素交换到待排序列的第一个位置（或将最大的元素交换到待排序列的最后一个位置），关键字最小的元素如气泡一般逐渐往上“漂浮”直至“水面”（或关键字最大的元素如石头一般下沉至水底）。下一趟冒泡时，前一趟确定的最小元素不再参与比较，每趟冒泡的结果是把序列中的最小元素（或

最大元素) 放到了序列的最终位置……这样最多做 $n-1$ 趟冒泡就能把所有元素排好序。

图 8.3 所示为冒泡排序的过程, 第一趟冒泡时: $27 < \overline{49}$, 不交换; $13 < 27$, 不交换; $76 > 13$, 交换; $97 > 13$, 交换; $65 > 13$, 交换; $38 > 13$, 交换; $49 > 13$, 交换。通过第一趟冒泡后, 最小元素已交换到第一个位置, 也是它的最终位置。第二趟冒泡时对剩余子序列采用同样方法进行排序, 以此类推, 到第五趟结束后没有发生交换, 说明表已有序, 冒泡排序结束。

	49	13	13	13	13	13	13
初始状态	38	49	27	27	27	27	27
	65	38	49	38	38	38	38
	97	65	38	49	49	49	49
	76	97	65	49	49	49	49
	13	76	97	65	65	65	65
	27	27	76	97	76	76	76
	49	49	49	76	97	97	97
第一趟后							
第二趟后							
第三趟后							
第四趟后							
第五趟后							
最终状态							

图 8.3 冒泡排序示例

冒泡排序算法的代码如下:

```
void BubbleSort(ElemType A[], int n) {
    for(int i=0;i<n-1;i++) {
        bool flag=false; //表示本趟冒泡是否发生交换的标志
        for(int j=n-1;j>i;j--) //一趟冒泡过程
            if(A[j-1]>A[j]) { //若为逆序
                swap(A[j-1],A[j]); //交换
                flag=true;
            }
        if(flag==false)
            return; //本趟遍历后没有发生交换, 说明表已经有序
    }
}
```

冒泡排序的性能分析如下:

空间效率: 仅使用了常数个辅助单元, 因而空间复杂度为 $O(1)$ 。

时间效率: 当初始序列有序时, 显然第一趟冒泡后 $flag$ 依然为 $false$ (本趟没有元素交换), 从而直接跳出循环, 比较次数为 $n-1$, 移动次数为 0, 从而最好情况下的时间复杂度为 $O(n)$; 当初始序列为逆序时, 需要进行 $n-1$ 趟排序, 第 i 趟排序要进行 $n-i$ 次关键字的比较, 而且每次比较后都必须移动元素 3 次来交换元素位置。这种情况下,

$$\text{比较次数} = \sum_{i=1}^{n-1} (n-i) = \frac{n(n-1)}{2}, \text{ 移动次数} = \sum_{i=1}^{n-1} 3(n-i) = \frac{3n(n-1)}{2}$$

从而, 最坏情况下的时间复杂度为 $O(n^2)$, 平均时间复杂度为 $O(n^2)$ 。

稳定性: 由于 $i > j$ 且 $A[i] = A[j]$ 时, 不会发生交换, 因此冒泡排序是一种稳定的排序方法。

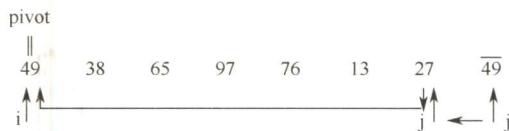
注意: 冒泡排序中所产生的有序子序列一定是全局有序的(不同于直接插入排序), 也就是说, 有序子序列中的所有元素的关键字一定小于(或大于)无序子序列中所有元素的关键字, 这样每趟排序都会将一个元素放置到其最终的位置上。

8.3.2 快速排序

快速排序的基本思想是基于分治法的：在待排序表 $L[1..n]$ 中任取一个元素 $pivot$ 作为枢轴（或称基准，通常取首元素），通过一趟排序将待排序表划分为独立的两部分 $L[1..k-1]$ 和 $L[k+1..n]$ ，使得 $L[1..k-1]$ 中的所有元素小于 $pivot$ ， $L[k+1..n]$ 中的所有元素大于或等于 $pivot$ ，则 $pivot$ 放在了其最终位置 $L(k)$ 上，这个过程称为一次划分。然后分别递归地对两个子表重复上述过程，直至每部分内只有一个元素或空为止，即所有元素放在了其最终位置上。

一趟快速排序的过程是一个交替搜索和交换的过程，下面通过实例来介绍，附设两个指针 i 和 j ，初值分别为 low 和 $high$ ，取第一个元素 49 为枢轴赋值到变量 $pivot$ 。

指针 j 从 $high$ 往前搜索找到第一个小于枢轴的元素 27，将 27 交换到 i 所指位置。



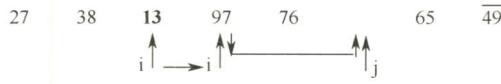
指针 i 从 low 往后搜索找到第一个大于枢轴的元素 65，将 65 交换到 j 所指位置。



指针 j 继续往前搜索找到小于枢轴的元素 13，将 13 交换到 i 所指位置。



指针 i 继续往后搜索找到大于枢轴的元素 97，将 97 交换到 j 所指位置。



指针 j 继续往前搜索小于枢轴的元素，直至 $i==j$ 。



关注公众号【乘龙考研】
一手更新 稳定有保障

此时，指针 i ($=j$) 之前的元素均小于 49，指针 i 之后的元素均大于或等于 49，将 49 放在 i 所指位置即其最终位置，经过一趟划分，将原序列分割成了前后两个子序列。

{27 38 13} {49} {76 97 65 49}

按照同样的方法对各子序列进行快速排序，若待排序列中只有一个元素，显然已有序。

{13}	27	{38}		{49 65}	76	{97}
结束		结束		49	{65}	结束
结束						

{13 27 38 49} {49} {65 76 97}

对算法的最好理解方式是手动地模拟一遍这些算法。

假设划分算法已知，记为 `Partition()`，返回的是上述的 k ，注意到 $L(k)$ 已放在其最终位置，因此可以先对表进行划分，而后对两个表调用同样的排序操作。因此可以递归地调用快速排序算法进行排序，具体的程序结构如下：

```
void QuickSort(ElemType A[], int low, int high){
    if (low < high) {
        // 递归跳出的条件
    }
}
```

```

    //Partition()就是划分操作，将表 A[low...high]划分为满足上述条件的两个子表
    int pivotpos=Partition(A,low,high);      //划分
    QuickSort(A,low,pivotpos-1);           //依次对两个子表进行递归排序
    QuickSort(A,pivotpos+1,high);
}
}

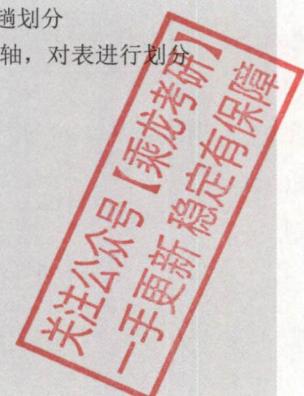
```

从上面的代码不难看出快速排序算法的关键在于划分操作，同时快速排序算法的性能也主要取决于划分操作的好坏。从快速排序算法提出至今，已有许多不同的划分操作版本，但考研所考查的快速排序的划分操作基本以严蔚敏的教材《数据结构》为主。假设每次总以当前表中第一个元素作为枢轴来对表进行划分，则将表中比枢轴大的元素向右移动，将比枢轴小的元素向左移动，使得一趟 Partition() 操作后，表中的元素被枢轴一分为二。代码如下：

```

int Partition(ElemType A[],int low,int high){ //一趟划分
    ELEMTYPE pivot=A[low]; //将当前表中第一个元素设为枢轴，对表进行划分
    while(low<high){ //循环跳出条件
        while(low<high&&A[high]>=pivot) --high;
        A[low]=A[high]; //将比枢轴小的元素移动到左端
        while(low<high&&A[low]<=pivot) ++low;
        A[high]=A[low]; //将比枢轴大的元素移动到右端
    }
    A[low]=pivot; //枢轴元素存放到最终位置
    return low; //返回存放枢轴的最终位置
}

```



快速排序算法的性能分析如下：

空间效率：由于快速排序是递归的，需要借助一个递归工作栈来保存每层递归调用的必要信息，其容量与递归调用的最大深度一致。最好情况下为 $O(\log_2 n)$ ；最坏情况下，因为要进行 $n-1$ 次递归调用，所以栈的深度为 $O(n)$ ；平均情况下，栈的深度为 $O(\log_2 n)$ 。

时间效率：快速排序的运行时间与划分是否对称有关，快速排序的最坏情况发生在两个区域分别包含 $n-1$ 个元素和 0 个元素时，这种最大限度的不对称性若发生在每层递归上，即对应于初始排序表基本有序或基本逆序时，就得到最坏情况下的时间复杂度为 $O(n^2)$ 。

有很多方法可以提高算法的效率：一种方法是尽量选取一个可以将数据中分的枢轴元素，如从序列的头尾及中间选取三个元素，再取这三个元素的中间值作为最终的枢轴元素；或者随机地从当前表中选取枢轴元素，这样做可使得最坏情况在实际排序中几乎不会发生。

在最理想的状态下，即 Partition() 可能做到最平衡的划分，得到的两个子问题的大小都不可能大于 $n/2$ ，在这种情况下，快速排序的运行速度将大大提升，此时，时间复杂度为 $O(n \log_2 n)$ 。好在快速排序平均情况下的运行时间与其最佳情况下的运行时间很接近，而不是接近其最坏情况下的运行时间。快速排序是所有内部排序算法中平均性能最优的排序算法。

稳定性：在划分算法中，若右端区间有两个关键字相同，且均小于基准值的记录，则在交换到左端区间后，它们的相对位置会发生变化，即快速排序是一种不稳定的排序方法。例如，表 $L = \{3, 2, 2\}$ ，经过一趟排序后 $L = \{2, 2, 3\}$ ，最终排序序列也是 $L = \{2, 2, 3\}$ ，显然，2 与 2 的相对次序已发生了变化。

注意：在快速排序算法中，并不产生有序子序列，但每趟排序后会将上一趟划分的各个无序子表的枢轴（基准）元素放到其最终的位置上。

8.3.3 本节试题精选

一、单项选择题

01. 对 n 个不同的元素利用冒泡法从小到大排序，在（ ）情况下元素交换的次数最多。
 A. 从大到小排列好的 B. 从小到大排列好的
 C. 元素无序 D. 元素基本有序
02. 若用冒泡排序算法对序列 {10, 14, 26, 29, 41, 52} 从大到小排序，则需进行（ ）次比较。
 A. 3 B. 10 C. 15 D. 25
03. 用某种排序方法对线性表 {25, 84, 21, 47, 15, 27, 68, 35, 20} 进行排序时，元素序列的变化情况如下：
 1) 25, 84, 21, 47, 15, 27, 68, 35, 20
 2) 20, 15, 21, 25, 47, 27, 68, 35, 84
 3) 15, 20, 21, 25, 35, 27, 47, 68, 84
 4) 15, 20, 21, 25, 27, 35, 47, 68, 84
 则所采用的排序方法是（ ）。
 A. 选择排序 B. 插入排序 C. 2 路归并排序 D. 快速排序
04. 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，从小到大得到的一次划分结果为（ ）。
 A. (38, 40, 46, 56, 79, 84) B. (40, 38, 46, 79, 56, 84)
 C. (40, 38, 46, 56, 79, 84) D. (40, 38, 46, 84, 56, 79)
05. 快速排序算法在（ ）情况下最不利于发挥其长处。
 A. 要排序的数据量太大 B. 要排序的数据中含有多个相同值
 C. 要排序的数据个数为奇数 D. 要排序的数据已基本有序
06. 就平均性能而言，目前最好的内部排序方法是（ ）。
 A. 冒泡排序 B. 直接插入排序 C. 希尔排序 D. 快速排序
07. 数据序列 $F = \{2, 1, 4, 9, 8, 10, 6, 20\}$ 只能是下列排序算法中的（ ）两趟排序后的结果。
 A. 快速排序 B. 冒泡排序 C. 选择排序 D. 插入排序
08. 对数据序列 {8, 9, 10, 4, 5, 6, 20, 1, 2} 采用冒泡排序（从后向前次序进行，要求升序），需要进行的趟数至少是（ ）。
 A. 3 B. 4 C. 5 D. 8
09. 对下列关键字序列用快排进行排序时，速度最快的情形是（ ），速度最慢的情形是（ ）。
 A. {21, 25, 5, 17, 9, 23, 30} B. {25, 23, 30, 17, 21, 5, 9}
 C. {21, 9, 17, 30, 25, 23, 5} D. {5, 9, 17, 21, 23, 25, 30}
10. 对下列 4 个序列，以第一个关键字为基准用快速排序算法进行排序，在第一趟过程中移动记录次数最多的是（ ）。
 A. 92, 96, 88, 42, 30, 35, 110, 100 B. 92, 96, 100, 110, 42, 35, 30, 88
 C. 100, 96, 92, 35, 30, 110, 88, 42 D. 42, 30, 35, 92, 100, 96, 88, 110
11. 下列序列中，（ ）可能是执行第一趟快速排序后所得到的序列。
 I. {68, 11, 18, 69, 23, 93, 73} II. {68, 11, 69, 23, 18, 93, 73}
 III. {93, 73, 68, 11, 69, 23, 18} IV. {68, 11, 69, 23, 18, 73, 93}
 A. I、IV B. II、III C. III、IV D. 只有 IV

关注公众号【乘龙考研】
一手更新 稳定有保障

12. 对 n 个关键字进行快速排序，最大递归深度为（ ），最小递归深度为（ ）。
 A. 1 B. n C. $\log_2 n$ D. $n \log_2 n$
13. 【2010 统考真题】对一组数据(2, 12, 16, 88, 5, 10)进行排序，若前 3 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（ ）。
 A. 冒泡排序 B. 希尔排序 C. 归并排序 D. 基数排序
14. 【2010 统考真题】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（ ）。
 A. 递归次数与初始数据的排列次序无关
 B. 每次划分后，先处理较长的分区可以减少递归次数
 C. 每次划分后，先处理较短的分区可以减少递归次数
 D. 递归次数与每次划分后得到的分区的处理顺序无关
15. 【2011 统考真题】为实现快速排序算法，待排序序列宜采用的存储方式是（ ）。
 A. 顺序存储 B. 散列存储 C. 链式存储 D. 索引存储
16. 【2014 统考真题】下列选项中，不可能是快速排序第 2 趟排序结果的是（ ）。
 A. 2, 3, 5, 4, 6, 7, 9 B. 2, 7, 5, 6, 4, 3, 9
 C. 3, 2, 5, 4, 7, 6, 9 D. 4, 2, 3, 5, 7, 6, 9
17. 【2019 统考真题】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”。下列序列中，不可能是快速排序第二趟结果的是（ ）。
 A. 5, 2, 16, 12, 28, 60, 32, 72 B. 2, 16, 5, 28, 12, 60, 32, 72
 C. 2, 12, 16, 5, 28, 32, 72, 60 D. 5, 2, 12, 28, 16, 32, 72, 60
- 二、综合应用题**
01. 在使用非递归方法实现快速排序时，通常要利用一个栈记忆待排序区间的两个端点。能否用队列来实现这个栈？为什么？
02. 编写双向冒泡排序算法，在正反两个方向交替进行扫描，即第一趟把关键字最大的元素放在序列的最后面，第二趟把关键字最小的元素放在序列的最前面，如此反复进行。
03. 已知线性表按顺序存储，且每个元素都是不相同的整型元素，设计把所有奇数移动到所有偶数前边的算法（要求时间最少，辅助空间最少）。
04. 试重新编写考点精析中的快速排序的划分算法，使之每次选取的枢轴值都是随机地从当前子表中选择的。
05. 试编写一个算法，使之能够在数组 $L[1 \dots n]$ 中找出第 k 小的元素（即从小到大排序后处于第 k 个位置的元素）。
06. 荷兰国旗问题：设有一个仅由红、白、蓝三种颜色的条块组成的条块序列，请编写一个时间复杂度为 $O(n)$ 的算法，使得这些条块按红、白、蓝的顺序排好，即排成荷兰国旗图案。
07. 【2016 统考真题】已知由 n ($n \geq 2$) 个正整数构成的集合 $A = \{a_k | 0 \leq k < n\}$ ，将其划分为两个不相交的子集 A_1 和 A_2 ，元素个数分别是 n_1 和 n_2 ， A_1 和 A_2 中的元素之和分别为 S_1 和 S_2 。设计一个尽可能高效的划分算法，满足 $|n_1 - n_2|$ 最小且 $|S_1 - S_2|$ 最大。要求：

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

8.3.4 答案与解析

一、单项选择题

关注公众号【乘龙考研】
一手更新 稳定有保障

01. A

通常情况下, 冒泡排序最少进行 1 次冒泡, 最多进行 $n-1$ 次冒泡。初始序列为逆序时, 需进行 $n-1$ 次冒泡, 并且需要交换的次数最多。初始序列为正序时, 进行 1 次冒泡(无交换)就可以终止算法。

02. C

冒泡排序始终在调整“逆序”, 因此交换次数为排列中逆序的个数。对逆序序列进行冒泡排序, 每个元素向后调整时都需要进行比较, 因此共需要比较 $5+4+3+2+1=15$ 次。

03. D

选择排序在每趟结束后可以确定一个元素的最终位置, 不对。插入排序, 第 i 趟后前 $i+1$ 个元素应该是有序的, 不对。第 2 趟 {20, 15} 和 {21, 25} 是反序的, 因此不是归并排序。快速排序每趟都将基准元素放在其最终位置, 然后以它为基准将序列划分为两个子序列。观察题中的排序过程, 可知是快速排序。

04. C

以 46 为基准元素, 首先从后向前扫描比 46 小的元素, 并与之进行交换, 而后从前向后扫描比 46 大的元素并将 46 与该元素交换, 得到 (40, 46, 56, 38, 79, 84)。此后, 继续重复从后向前扫描与从前往后扫描的操作, 直到 46 处于最终位置, 答案选择选项 C。

05. D

当待排序数据为基本有序时, 每次选取第 n 个元素为基准, 会导致划分区间分配不均匀, 不利于发挥快速排序算法的优势。相反, 当待排序数据分布较为随机时, 基准元素能将序列划分为两个长度大致相等的序列, 这时才能发挥快速排序的优势。

06. D

这里问的是平均性能, 选项 A、B 的平均性能都会达到 $O(n^2)$, 而希尔排序虽然大大降低了直接插入排序的时间复杂度, 但其平均性能不如快速排序。另外, 虽然众多排序算法的平均时间复杂度也是 $O(n \log_2 n)$, 但快速排序算法的常数因子是最小的。

07. A

若为插入排序, 则前三个元素应该是有序的, 显然不对。而冒泡排序和选择排序经过两趟排序后应该有两个元素处于最终位置(最左/右端), 无论是按从小到大还是从大到小排序, 数据序列中都没有两个满足这样的条件的元素, 因此只可能选择选项 A。

【另解】先写出排好序的序列, 并和题中的序列做对比。

题中序列: 2 1 4 9 8 10 6 20

已排好序序列: 1 2 4 6 8 9 10 20

在已排好序的序列中, 与题中序列相同元素的有 4、8 和 20, 最左和最右两个元素与题中的序列不同, 故不可能是冒泡排序、选择排序或插入排序。

08. C

从后向前“冒泡”的过程为, 第 1 趟 {1, 8, 9, 10, 4, 5, 6, 20, 2}, 第 2 趟 {1, 2, 8, 9, 10, 4, 5, 6, 20},

第 3 趟 {1, 2, 4, 8, 9, 10, 5, 6, 20}, 第 4 趟 {1, 2, 4, 5, 8, 9, 10, 6, 20}, 第 5 趟 {1, 2, 4, 5, 6, 8, 9, 10, 20}, 经过第 5 趟冒泡后，序列已经全局有序，故选择选项 C。实际每趟冒泡发生交换后可以判断是否会导致新的逆序对，如果不会产生，则本趟冒泡之后序列全局有序，所以最少 5 趟即可。

09. A、D

当每次的枢轴都把表等分为长度相近的两个子表时，速度是最快的；当表本身已经有序或逆序时，速度最慢。选项 D 中的序列已按关键字排好序，因此它是最慢的，而选项 A 中第一趟枢轴值 21 将表划分为两个子表 {2, 17, 5} 和 {25, 23, 30}，而后对两个子表划分时，枢轴值再次将它们等分，所以该序列是快速排序最优的情况，速度最快。其他选项可以类似分析。

10. B

对各序列分别执行一趟快速排序，可做如下分析（以选项 A 为例）：由于枢轴值为 92，因此 35 移动到第一个位置，96 移动到第六个位置，30 移动到第二个位置，再将枢轴值移动到 30 所在的单元，即第五个位置，所以选项 A 中序列移动的次数为 4。同样，可以分析出选项 B 中序列的移动次数为 8，选项 C 中序列的移动次数为 4，选项 D 中序列的移动次数为 2。

11. C

显然，若按从小到大排序，则最终有序的序列是 {11, 18, 23, 68, 69, 73, 93}；若按从大到小排序，则最终有序的序列是 {93, 73, 69, 68, 23, 18, 11}。对比可知选项 I、II 中没有处于最终位置的元素，故选项 I、II 都不可能。选项 III 中 73 和 93 处于从大到小排序后的最终位置，而且 73 将序列分割成大于 73 和小于 73 的两部分，故选项 III 是有可能的。选项 IV 中 73 和 93 处于从小到大排列后的最终位置，73 也将序列分割成大于 73 和小于 73 的两部分。

12. B、C

快速排序过程构成一个递归树，递归深度即递归树的高度。枢轴值每次都将子表等分时，递归树的高为 $\log_2 n$ ；枢轴值每次都是子表的最大值或最小值时，递归树退化为单链表，树高为 n 。

13. A

分别用其他 3 种排序算法执行数据，归并排序第一趟后的结果为 (2, 12, 16, 88, 5, 10)，基数排序第一趟后的结果为 (10, 2, 12, 5, 16, 88)，希尔排序显然是不符合的。只有冒泡排序符合条件。

【另解】由题干可以看出每趟都产生一个最大的数排在后面，可直接定位冒泡排序。

14. D

递归次数与各元素的初始排列有关。若每次划分后分区比较平衡，则递归次数少；若分区不平衡，递归次数多。递归次数与处理顺序是没有关系的。

15. A

绝大部分内部排序只适用于顺序存储结构。快速排序在排序的过程中，既要从后向前查找，又要从前向后查找，因此宜采用顺序存储。

16. C

快排的阶段性排序结果的特点是，第 i 趟完成时，会有 i 个以上的数出现在它最终将要出现的位置，即它左边的数都比它小，它右边的数都比它大。题目问第二趟排序的结果，即要找不存在两个这样的数的选项。选项 A 中 2, 3, 6, 7, 9 均符合，所以选项 A 排除；选项 B 中，2, 9 均符合，所以选项 B 排除；选项 D 中 5, 9 均符合，所以选项 D 排除；最后看选项 C，只有 9 一个数符合，所以选项 C 不可能是快速排序第二趟的结果。

17. D

要清晰掌握排序过程中“趟”的含义，题干也进行了解释——对尚未确定最终位置的所有元素都处理一遍才是一趟，所以此时要对前后两块子表各做一次快速排序才是一“趟”，如果只对

一块子表进行了排序，而未处理另一块子表，就不能算是完整的一趟。选项 A，第一趟匹配 72，只余一块无序序列，第二趟匹配 28，选项 A 可能。选项 B，第一趟匹配 2，第二趟匹配 72，B 可能。选项 C，第一趟匹配 2，第二趟匹配 28 或 32，选项 C 可能。选项 D，无论是先匹配 12 还是先匹配 32，都会将序列分成两块，那么第二趟必须有两个元素匹配，所以选项 D 不可能。

二、综合应用题

01. 【解答】

可以用队列来代替栈。在快速排序的过程中，通过一趟划分，可以把一个待排序区间分为两个子区间，然后分别对这两个子区间施行同样的划分。栈的作用是在处理一个子区间时，保存另一个子区间的上界和下界（排序过程中可能产生新的左、右子区间），待该区间处理完后再从栈中取出另一子区间的边界，对其进行处理。这个功能用队列也可以实现，只不过处理子区间的顺序有所变动而已。

02. 【解答】

这种排序方法又称双向起泡。奇数趟时，从前向后比较相邻元素的关键字，遇到逆序即交换，直到把序列中关键字最大的元素移动到序列尾部。偶数趟时，从后往前比较相邻元素的关键字，遇到逆序即交换，直到把序列中关键字最小的元素移动到序列前端。程序代码如下：

```
void BubbleSort(ElemType A[], int n){
    //双向起泡排序，交替进行正反两个方向的起泡过程
    int low=0,high=n-1;
    bool flag=true; //一趟冒泡后记录元素是否交换标志
    while(low<high&&flag){ //循环跳出条件，当 flag 为 false 说明已没有逆序
        flag=false; //每趟初始置 flag 为 false
        for(i=low;i<high;i++) //从前向后起泡
            if(A[i]>A[i+1]){ //发生逆序
                swap(A[i],A[i+1]); //交换
                flag=true; //置 flag
            }
        high--; //更新上界
        for(i=high;i>low;i--) //从后往前起泡
            if(A[i]<A[i-1]){ //发生逆序
                swap(A[i],A[i-1]); //交换
                flag=true; //置 flag
            }
        low++; //修改下界
    }
}
```

03. 【解答】

本题可采用基于快速排序的划分思想来设计算法，只需遍历一次即可，其时间复杂度为 $O(n)$ ，空间复杂度为 $O(1)$ 。假设表为 $L[1..n]$ ，基本思想是：先从前向后找到一个偶数元素 $L(i)$ ，再从后向前找到一个奇数元素 $L(j)$ ，将二者交换；重复上述过程直到 i 大于 j 。

算法的实现如下：

```
void move(ElemType A[], int len){
    //对表 A 按奇偶进行一趟划分
    int i=0,j=len-1; //i 表示左端偶数元素的下标；j 表示右端奇数元素的下标
    while(i<j){
        while(i<j&&A[i]%2!=0) i++; //从前向后找到一个偶数元素
```

```

        while(i<j&&A[j]%2!=1) j--; //从后向前找到一个奇数元素
        if (i<j){
            Swap(A[i],A[j]);           //交换这两个元素
            i++; j--;
        }
    }
}

```

04. 【解答】

这类题目比较简单，为方便起见，可直接先随机地求出枢轴的下标，然后将枢轴值与 $A[low]$ 交换，而后的思想就与前面的划分算法一样。算法的实现如下：

```

int Partition2(ElemType A[], int low, int high){
    int rand_Index=low+rand()%(high-low+1);
    Swap(A[rand_Index],A[low]);      //将枢轴值交换到第一个元素
    ElemenType pivot=A[low];         //置当前表中的第一个元素为枢轴值
    int i=low;                      //使得表  $A[low...i]$  中的所有元素小于 pivot，初始为空表
    for(int j=low+1;j<=high;j++)   //从第二个元素开始寻找小于基准的元素
        if(A[j]<pivot)             //找到后，交换到前面
            swap(A[++i],A[j]);
    swap(A[i],A[low]);              //将基准元素插入最终位置
    return i;                      //返回基准元素的位置
}

```

注意：本题代码中的比较方法和考点精析中的两种方法均不相同，请读者仔细模拟各种划分算法的执行步骤，做到真正掌握。对于下一题，读者可以尝试采用这三种比较方法来分别解决。

05. 【解答】

显然，本题最直接的做法是用排序算法对数组先进行从小到大的排序，然后直接提取 $L(k)$ 便得到了第 k 小元素，但其平均时间复杂度将达到 $O(n \log_2 n)$ 以上。此外，还可采用小顶堆的方法，每次堆顶元素都是最小值元素，时间复杂度为 $O(n + k \log_2 n)$ 。下面介绍一个更精彩的算法，它基于快速排序的划分操作。

这个算法的主要思想如下：从数组 $L[1..n]$ 中选择枢轴 $pivot$ （随机或直接取第一个）进行和快速排序一样的划分操作后，表 $L[1..n]$ 被划分为 $L[1..m-1]$ 和 $L[m+1..n]$ ，其中 $L(m)=pivot$ 。

讨论 m 与 k 的大小关系：

- 1) 当 $m=k$ 时，显然 $pivot$ 就是所要寻找的元素，直接返回 $pivot$ 即可。
- 2) 当 $m < k$ 时，所要寻找的元素一定落在 $L[m+1..n]$ 中，因此可对 $L[m+1..n]$ 递归地查找第 $k-m$ 小的元素。
- 3) 当 $m > k$ 时，所要寻找的元素一定落在 $L[1..m-1]$ 中，因此可对 $L[1..m-1]$ 递归地查找第 k 小的元素。

该算法的时间复杂度在平均情况下可以达到 $O(n)$ ，而所占空间的复杂度则取决于划分的方法。算法的实现如下：

```

int kth_elem(int a[], int low, int high, int k){
    int pivot=a[low];
    int low_temp=low;           //由于下面会修改 low 与 high，在递归时又要用到它们
    int high_temp=high;
    while(low<high){
        while(low<high&&a[high]>=pivot)
            --high;

```

```

    a[low]=a[high];
    while(low<high&&a[low]<=pivot)
        ++low;
    a[high]=a[low];
}
a[low]=pivot;
//上面即为快速排序中的划分算法
//以下就是本算法思想中所述的内容
if(low==k)           //由于与 k 相同，直接返回 pivot 元素
    return a[low];
else if(low>k)      //在前一部分表中递归寻找
    return kth_elem(a,low_temp,low-1,k);
else                 //在后一部分表中递归寻找
    return kth_elem(a,low+1,high_temp,k);
}

```

关注公众号【乘龙考研】
一手更新 稳定有保障

06. 【解答】

算法思想：顺序扫描线性表，将红色条块交换到线性表的最前面，蓝色条块交换到线性表的最后面。为此，设立三个指针，其中， j 为工作指针，表示当前扫描的元素， i 以前的元素全部为红色， k 以后的元素全部为蓝色。根据 j 所指示元素的颜色，决定将其交换到序列的前部或尾部。初始时 $i = 0$, $k = n - 1$ ，算法的实现如下：

```

typedef enum{RED,WHITE,BLUE} color; //设置枚举数组
void Flag_Arrange(color a[],int n){
    int i=0,j=0,k=n-1;
    while(j<=k)
        switch(a[j]){//判断条块的颜色
            case RED: Swap(a[i],a[j]);i++;j++;break;
            //红色，则和 i 交换
            case WHITE: j++;break;
            case BLUE: Swap(a[j],a[k]);k--;
            //蓝色，则和 k 交换
            //这里没有 j++语句以防止交换后 a[j] 仍为蓝色的情况
        }
}

```

例如，将元素值正数、负数和零排序为前面都是负数，接着是 0，最后是正数，也用同样的方法。思考：为什么 `case RED` 时不用考虑交换后 $a[j]$ 仍为红色，而 `case BLUE` 却需要考虑交换后 $a[j]$ 仍为蓝色？

07. 【解答】

1) 算法的基本设计思想

由题意知，将最小的 $\lfloor n/2 \rfloor$ 个元素放在 A_1 中，其余的元素放在 A_2 中，分组结果即可满足题目要求。仿照快速排序的思想，基于枢轴将 n 个整数划分为两个子集。根据划分后枢轴所处的位置 i 分别处理：

- ① 若 $i = \lfloor n/2 \rfloor$ ，则分组完成，算法结束。
- ② 若 $i < \lfloor n/2 \rfloor$ ，则枢轴及之前的所有元素均属于 A_1 ，继续对 i 之后的元素进行划分。
- ③ 若 $i > \lfloor n/2 \rfloor$ ，则枢轴及之后的所有元素均属于 A_2 ，继续对 i 之前的元素进行划分。

基于该设计思想实现的算法，无须对全部元素进行全排序，其平均时间复杂度是 $O(n)$ ，空间复杂度是 $O(1)$ 。

2) 算法实现

```

int setPartition(int a[], int n){
    int pivotkey, low=0,low0=0,high=n-1,high0=n-1,flag=1,k=n/2,i;
    int s1=0,s2=0;
    while(flag) {
        pivotkey=a[low];           //选择枢轴
        while(low<high) {         //基于枢轴对数据进行划分
            while(low<high && a[high]>=pivotkey) --high;
            if(low!=high) a[low]=a[high];
            while(low<high && a[low]<=pivotkey) ++low;
            if(low!=high) a[high]=a[low];
        }                           //end of while(low<high)
        a[low]=pivotkey;
        if(low==k-1)               //若枢轴是第 n/2 小元素，划分成功
            flag=0;
        else{                      //是否继续划分
            if(low<k-1){
                low0=++low;
                high=high0;
            }
            else{
                high0=--high;
                low=low0;
            }
        }
    }
    for(i=0;i<k;i++) s1+=a[i];
    for(i=k;i<n;i++) s2+=a[i];
    return s2-s1;
}

```

关注公众号【乘龙考研】
一手更新 稳定有保障

3) 本答案给出的算法平均时间复杂度是 $O(n)$, 空间复杂度是 $O(1)$ 。

8.4 选择排序

选择排序的基本思想是：每一趟（如第 i 趟）在后面 $n-i+1$ ($i=1, 2, \dots, n-1$) 个待排序元素中选取关键字最小的元素，作为有序子序列的第 i 个元素，直到第 $n-1$ 趟做完，待排序元素只剩下 1 个，就不用再选了。选择排序中的堆排序算法是历年考查的重点。

8.4.1 简单选择排序

根据上面选择排序的思想，可以很直观地得出简单选择排序算法的思想：假设排序表为 $L[1\dots n]$ ，第 i 趟排序即从 $L[i\dots n]$ 中选择关键字最小的元素与 $L(i)$ 交换，每一趟排序可以确定一个元素的最终位置，这样经过 $n-1$ 趟排序就可使得整个排序表有序。

简单选择排序算法的代码如下：

```

void SelectSort(ElemType A[], int n){
    for(int i=0;i<n-1;i++) {           //一共进行 n-1 趟
        int min=i;                     //记录最小元素位置

```

```

for(int j=i+1;j<n;j++)
    if(A[j]<A[min]) min=j;           //在 A[i...n-1] 中选择最小的元素
    if(min!=i) swap(A[i],A[min]);     //更新最小元素位置
}
}

```

简单选择排序算法的性能分析如下：

空间效率：仅使用常数个辅助单元，故空间效率为 $O(1)$ 。

时间效率：从上述伪码中不难看出，在简单选择排序过程中，元素移动的操作次数很少，不会超过 $3(n-1)$ 次，最好的情况是移动 0 次，此时对应的表已经有序；但元素间比较的次数与序列的初始状态无关，始终是 $n(n-1)/2$ 次，因此时间复杂度始终是 $O(n^2)$ 。

稳定性：在第 i 趟找到最小元素后，和第 i 个元素交换，可能会导致第 i 个元素与其含有相同关键字元素的相对位置发生改变。例如，表 $L = \{2, 2, 1\}$ ，经过一趟排序后 $L = \{1, 2, 2\}$ ，最终排序序列也是 $L = \{1, 2, 2\}$ ，显然，2 与 2 的相对次序已发生变化。因此，简单选择排序是一种不稳定的排序方法。

8.4.2 堆排序

堆的定义如下， n 个关键字序列 $L[1...n]$ 称为堆，当且仅当该序列满足：

- ① $L(i) >= L(2i)$ 且 $L(i) >= L(2i+1)$ 或
- ② $L(i) <= L(2i)$ 且 $L(i) <= L(2i+1)$ ($1 \leq i \leq \lfloor n/2 \rfloor$)

可以将堆视为一棵完全二叉树，满足条件①的堆称为大根堆（大顶堆），大根堆的最大元素存放在根结点，且其任意一个非根结点的值小于或等于其双亲结点值。满足条件②的堆称为小根堆（小顶堆），小根堆的定义刚好相反，根结点是最小元素。图 8.4 所示为一个大根堆。

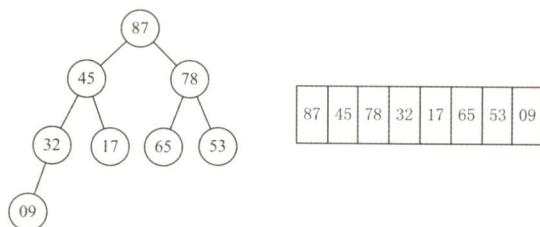


图 8.4 一个大根堆示意图

堆排序的思路很简单：首先将存放在 $L[1...n]$ 中的 n 个元素建成初始堆，由于堆本身的特点（以大根堆为例），堆顶元素就是最大值。输出堆顶元素后，通常将堆底元素送入堆顶，此时根结点已不满足大根堆的性质，堆被破坏，将堆顶元素向下调整使其继续保持大根堆的性质，再输出堆顶元素。如此重复，直到堆中仅剩一个元素为止。可见堆排序需要解决两个问题：①如何将无序序列构造成初始堆？②输出堆顶元素后，如何将剩余元素调整成新的堆？

堆排序的关键是构造初始堆。 n 个结点的完全二叉树，最后一个结点是第 $\lfloor n/2 \rfloor$ 个结点的孩子。对第 $\lfloor n/2 \rfloor$ 个结点为根的子树筛选（对于大根堆，若根结点的关键字小于左右孩子中关键字较大者，则交换），使该子树成为堆。之后向前依次对各结点 ($\lfloor n/2 \rfloor - 1 \sim 1$) 为根的子树进行筛选，看该结点值是否大于其左右子结点的值，若不大于，则将左右子结点中的较大值与之交换，交换后可能会破坏下一级的堆，于是继续采用上述方法构造下一级的堆，直到以该结点为根的子树构成堆为止。反复利用上述调整堆的方法建堆，直到根结点。

如图 8.5 所示，初始时调整 $L(4)$ 子树， $09 < 32$ ，交换，交换后满足堆的定义；向前继续调

整 L(3) 子树， $78 <$ 左右孩子的较大者 87，交换，交换后满足堆的定义；向前调整 L(2) 子树， $17 <$ 左右孩子的较大者 45，交换后满足堆的定义；向前调整至根结点 L(1)， $53 <$ 左右孩子的较大者 87，交换，交换后破坏了 L(3) 子树的堆，采用上述方法对 L(3) 进行调整， $53 <$ 左右孩子的较大者 78，交换，至此该完全二叉树满足堆的定义。

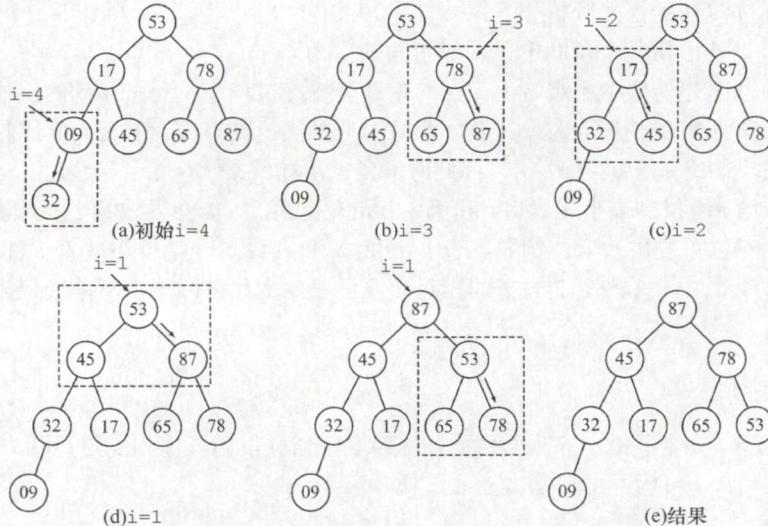


图 8.5 自下往上逐步调整为大根堆

输出堆顶元素后，将堆的最后一个元素与堆顶元素交换，此时堆的性质被破坏，需要向下进行筛选。将 09 和左右孩子的较大者 78 交换，交换后破坏了 L(3) 子树的堆，继续对 L(3) 子树向下筛选，将 09 和左右孩子的较大者 65 交换，交换后得到了新堆，调整过程如图 8.6 所示。

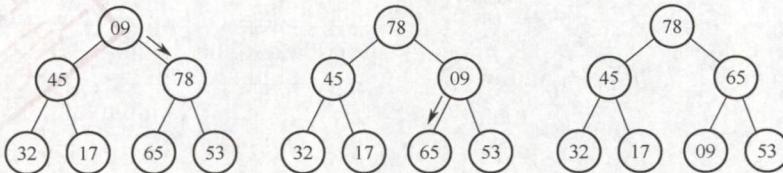


图 8.6 输出堆顶元素后再将剩余元素调整成新堆

下面是建立大根堆的算法：

```

void BuildMaxHeap(ElemType A[], int len) {
    for(int i=len/2; i>0; i--) //从 i=[n/2]~1, 反复调整堆
        HeadAdjust(A, i, len);
}

void HeadAdjust(ElemType A[], int k, int len) {
//函数 HeadAdjust 将元素 k 为根的子树进行调整
    A[0]=A[k]; //A[0]暂存子树的根结点
    for(int i=2*k; i<=len; i*=2) //沿 key 较大的子结点向下筛选
        if(i<len&&A[i]<A[i+1])
            i++; //取 key 较大的子结点的下标
        if(A[0]>=A[i]) break; //筛选结束
        else{
}
}

```

```

        A[k]=A[i];           //将 A[i] 调整到双亲结点上
        k=i;                 //修改 k 值, 以便继续向下筛选
    }
}
A[k]=A[0];           //被筛选结点的值放入最终位置
}

```

调整的时间与树高有关, 为 $O(h)$ 。在建含 n 个元素的堆时, 关键字的比较总次数不超过 $4n$, 时间复杂度为 $O(n)$, 这说明可以在线性时间内将一个无序数组建成一个堆。

下面是堆排序算法:

```

void HeapSort(ElemType A[], int len){
    BuildMaxHeap(A, len);           //初始建堆
    for(int i=len; i>1; i--) {
        Swap(A[i], A[1]);          //输出堆顶元素(和堆底元素交换)
        HeadAdjust(A, 1, i-1);      //调整, 把剩余的 i-1 个元素整理成堆
    }
}

```

同时, 堆也支持插入操作。对堆进行插入操作时, 先将新结点放在堆的末端, 再对这个新结点向上执行调整操作。大根堆的插入操作示例如图 8.7 所示。

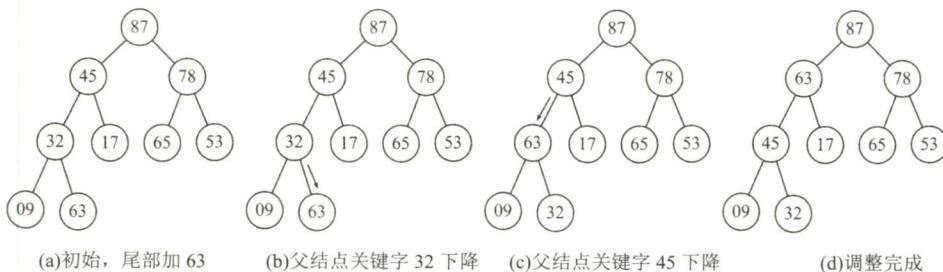


图 8.7 大根堆的插入操作示例

堆排序适合关键字较多的情况。例如, 在 1 亿个数中选出前 100 个最大值? 首先使用一个大小为 100 的数组, 读入前 100 个数, 建立小顶堆, 而后依次读入余下的数, 若小于堆顶则舍弃, 否则用该数取代堆顶并重新调整堆, 待数据读取完毕, 堆中 100 个数即为所求。

堆排序算法的性能分析如下:

空间效率: 仅使用了常数个辅助单元, 所以空间复杂度为 $O(1)$ 。

时间效率: 建堆时间为 $O(n)$, 之后有 $n-1$ 次向下调整操作, 每次调整的时间复杂度为 $O(h)$, 故在最好、最坏和平均情况下, 堆排序的时间复杂度为 $O(n \log_2 n)$ 。

稳定性: 进行筛选时, 有可能把后面相同关键字的元素调整到前面, 所以堆排序算法是一种不稳定的排序方法。例如, 表 $L = \{1, 2, 2\}$, 构造初始堆时可能将 2 交换到堆顶, 此时 $L = \{2, 1, 2\}$, 最终排序序列为 $L = \{1, 2, 2\}$, 显然, 2 与 2 的相对次序已发生变化。

8.4.3 本节试题精选

一、单项选择题

01. 在以下排序算法中, 每次从未排序的记录中选取最小关键字的记录, 加入已排序记录的末尾, 该排序方法是()。
- A. 简单选择排序 B. 冒泡排序 C. 堆排序 D. 直接插入排序

02. 简单选择排序算法的比较次数和移动次数分别为()。
- $O(n)$, $O(\log_2 n)$
 - $O(\log_2 n)$, $O(n^2)$
 - $O(n^2)$, $O(n)$
 - $O(n \log_2 n)$, $O(n)$
03. 设线性表中每个元素有两个数据项 k_1 和 k_2 , 现对线性表按以下规则进行排序: 先看数据项 k_1 , k_1 值小的元素在前, 大的元素在后; 在 k_1 值相同的情况下, 再看 k_2 , k_2 值小的在前, 大的元素在后。满足这种要求的排序方法是()。
- 先按 k_1 进行直接插入排序, 再按 k_2 进行简单选择排序
 - 先按 k_2 进行直接插入排序, 再按 k_1 进行简单选择排序
 - 先按 k_1 进行简单选择排序, 再按 k_2 进行直接插入排序
 - 先按 k_2 进行简单选择排序, 再按 k_1 进行直接插入排序
04. 若只想得到 1000 个元素组成的序列中第 10 个最小元素之前的部分排序的序列, 用()方法最快。
- 冒泡排序
 - 快速排序
 - 希尔排序
 - 堆排序
05. 下列()是一个堆。
- 19, 75, 34, 26, 97, 56
 - 97, 26, 34, 75, 19, 56
 - 19, 56, 26, 97, 34, 75
 - 19, 34, 26, 97, 56, 75
06. 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小根堆为()。
- 1, 4, 8, 9, 20, 7, 15, 7
 - 1, 7, 15, 7, 4, 8, 20, 9
 - 1, 4, 7, 8, 20, 15, 7, 9
 - A、B、C 均不对
07. 在含有 n 个关键字的小根堆中, 关键字最大的记录有可能存储在()位置。
- $n/2$
 - $n/2 + 2$
 - 1
 - $n/2 - 1$
08. 向具有 n 个结点的堆中插入一个新元素的时间复杂度为(), 删除一个元素的时间复杂度为()。
- $O(1)$
 - $O(n)$
 - $O(\log_2 n)$
 - $O(n \log_2 n)$
09. 构建 n 个记录的初始堆, 其时间复杂度为(); 对 n 个记录进行堆排序, 最坏情况下其时间复杂度为()。
- $O(n)$
 - $O(n^2)$
 - $O(\log_2 n)$
 - $O(n \log_2 n)$
10. 对关键码序列{23, 17, 72, 60, 25, 8, 68, 71, 52}进行堆排序, 输出两个最小关键码后的剩余堆是()。
- {23, 72, 60, 25, 68, 71, 52}
 - {23, 25, 52, 60, 71, 72, 68}
 - {71, 25, 23, 52, 60, 72, 68}
 - {23, 25, 68, 52, 60, 72, 71}
11. 【2009 统考真题】已知关键字序列 5, 8, 12, 19, 28, 20, 15, 22 是小根堆, 插入关键字 3, 调整好后得到的小根堆是()。
- 3, 5, 12, 8, 28, 20, 15, 22, 19
 - 3, 5, 12, 19, 20, 15, 22, 8, 28
 - 3, 8, 12, 5, 20, 15, 22, 28, 19
 - 3, 12, 5, 8, 28, 20, 15, 22, 19
12. 【2011 统考真题】已知序列 25, 13, 10, 12, 9 是大根堆, 在序列尾部插入新元素 18, 将其再调整为大根堆, 调整过程中元素之间进行的比较次数是()。
- 1
 - 2
 - 4
 - 5
13. 下列 4 种排序方法中, 排序过程中的比较次数与序列初始状态无关的是()。
- 选择排序法
 - 插入排序法
 - 快速排序法
 - 冒泡排序法
14. 【2015 统考真题】已知小根堆为 8, 15, 10, 21, 34, 16, 12, 删去关键字 8 之后需重建堆,

在此过程中，关键字之间的比较次数是（ ）。

- A. 1 B. 2 C. 3 D. 4

15. 【2018 统考真题】在将序列(6, 1, 5, 9, 8, 4, 7)建成大根堆时，正确的序列变化过程是（ ）。

- A. 6, 1, 7, 9, 8, 4, 5 → 6, 9, 7, 1, 8, 4, 5 → 9, 6, 7, 1, 8, 4, 5 → 9, 8, 7, 1, 6, 4, 5
 B. 6, 9, 5, 1, 8, 4, 7 → 6, 9, 7, 1, 8, 4, 5 → 9, 6, 7, 1, 8, 4, 5 → 9, 8, 7, 1, 6, 4, 5
 C. 6, 9, 5, 1, 8, 4, 7 → 9, 6, 5, 1, 8, 4, 7 → 9, 6, 7, 1, 8, 4, 5 → 9, 8, 7, 1, 6, 4, 5
 D. 6, 1, 7, 9, 8, 4, 5 → 7, 1, 6, 9, 8, 4, 5 → 7, 9, 6, 1, 8, 4, 5 → 9, 7, 6, 1, 8, 4, 5 → 9, 8, 6, 1, 7, 4, 5

16. 【2020 统考真题】下列关于大根堆（至少含 2 个元素）的叙述中，正确的是（ ）。

- I. 可以将堆视为一棵完全二叉树
 II. 可以采用顺序存储方式保存堆
 III. 可以将堆视为一棵二叉排序树
 IV. 堆中的次大值一定在根的下一层
- A. 仅 I、II B. 仅 II、III C. 仅 I、II 和 IV D. I、III 和 IV

17. 【2021 统考真题】将关键字 6, 9, 1, 5, 8, 4, 7 依次插入到初始为空的大根堆 H 中，得到的 H 是（ ）。

- A. 9, 8, 7, 6, 5, 4, 1 B. 9, 8, 7, 5, 6, 1, 4
 C. 9, 8, 7, 5, 6, 4, 1 D. 9, 6, 7, 5, 8, 4, 1

二、综合应用题

01. 指出堆和二叉排序树的区别？
 02. 若只想得到一个序列中第 k ($k \geq 5$) 个最小元素之前的部分排序序列，则最好采用什么排序方法？
 03. 有 n 个元素已构成一个小根堆，现在要增加一个元素 K_{n+1} ，请用文字简要说明如何在 $\log_2 n$ 的时间内将其重新调整为一个堆。
 04. 编写一个算法，在基于单链表表示的待排序关键字序列上进行简单选择排序。
 05. 试设计一个算法，判断一个数据序列是否构成一个小根堆。
 06. 【2022 统考真题】现有 n ($n > 100000$) 个数保存在一维数组 M 中，需要查找 M 中最小的 10 个数。请回答下列问题。
 1) 设计一个完成上述查找任务的算法，要求平均情况下的比较次数尽可能少，简述其算法思想（不需要程序实现）。
 2) 说明你所设计的算法平均情况下的时间复杂度和空间复杂度。

8.4.4 答案与解析

一、单项选择题

01. A

02. C

注意：读者应熟练掌握各种排序算法的思想、过程和特点。

03. D

本题思路来自基数排序的 LSD，首先应确定 k_1 、 k_2 的排序顺序，若先排 k_1 再排 k_2 ，则排序结果不符合题意，排除选项 A 和 C。再考虑算法的稳定性，当 k_2 排好序后，再对 k_1 排序，若对 k_1 排序采用的算法是不稳定的，则对于 k_1 相同而 k_2 不同的元素可能会改变相对次序，从



而不一定能满足题设要求。直接插入排序算法是稳定的，而简单选择排序算法是不稳定的，故只能选择选项 D。

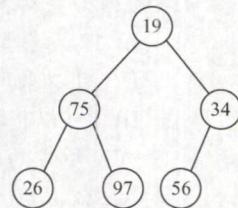
04. D

希尔排序和快速排序要等排序全部完成之后才能确定最小的 10 个元素。冒泡排序需要从后向前执行 10 趟冒泡才能得到 10 个最小的元素，而堆排序只需调整 10 次小根堆，调整时间与树高成正比。显然堆排序所需的时间更短。

通常，取一大堆数据中的 k 个最大（最小）的元素时，都优先采用堆排序。

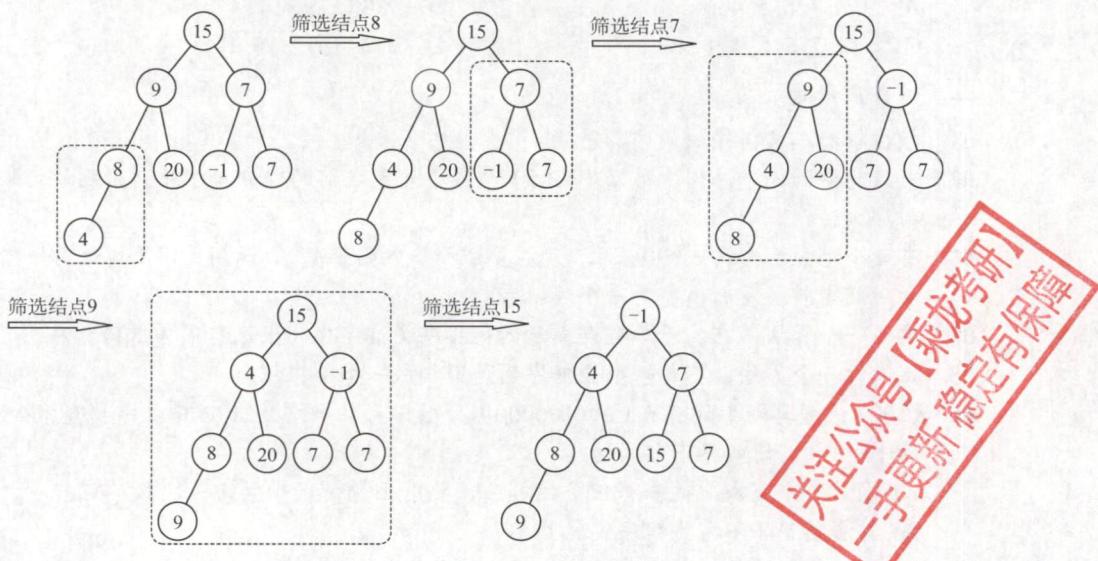
05. D

可将每个选项中的序列表示成完全二叉树，再看父结点与子结点的关系是否全部满足堆的定义。例如，选项 A 中序列对应的完全二叉树如右图所示。显然，最小元素 19 在根结点，因此可能是小根堆，但 75 与 26 的关系却不满足小根堆的定义，所以选项 A 中的序列不是一个堆。其他选项采用类似的过程分析。



06. C

从 $\lfloor n/2 \rfloor \sim 1$ 依次筛选堆的过程如下图所示，显然选择选项 C。



07. B

这是小根堆，关键字最大的记录一定存储在这个堆所对应的完全二叉树的叶结点中；又因为二叉树中的最后一个非叶结点存储在 $\lfloor n/2 \rfloor$ 中，所以关键字最大记录的存储范围为 $\lfloor n/2 \rfloor + 1 \sim n$ ，所以应该选择选项 B。

08. C、C

在向有 n 个元素的堆中插入一个新元素时，需要调用一个向上调整的算法，比较次数最多等于树的高度减 1，由于树的高度为 $\lfloor \log_2 n \rfloor + 1$ ，所以堆的向上调整算法的比较次数最多等于 $\lfloor \log_2 n \rfloor$ 。此处需要注意，调整堆和建初始堆的时间复杂度是不一样的，读者可以仔细分析两个算法的具体执行过程。

09. A、D

建堆过程中，向下调整的时间与树高 h 有关，为 $O(h)$ 。每次向下调整时，大部分结点的高度

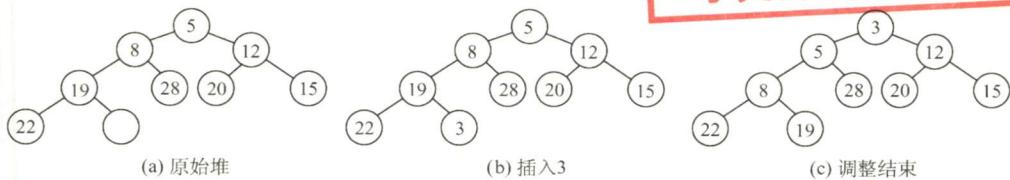
都较小。因此，可以证明在元素个数为 n 的序列上建堆，其时间复杂度为 $O(n)$ 。无论是在最好情况下还是在最坏情况下，堆排序的时间复杂度均为 $O(n \log_2 n)$ 。

10. D

筛选法初始建堆为 {8, 17, 23, 52, 25, 72, 68, 71, 60}，输出 8 后重建的堆为 {17, 25, 23, 52, 60, 72, 68, 71}，输出 17 后重建的堆为 {23, 25, 68, 52, 60, 72, 71}。

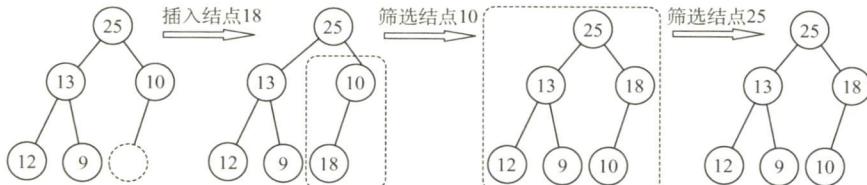
11. A

插入关键字 3 后，堆的变化过程如下图所示。



12. B

首先 18 与 10 比较，交换位置，再与 25 比较，不交换位置。共比较了 2 次，调整的过程如下图所示。



13. A

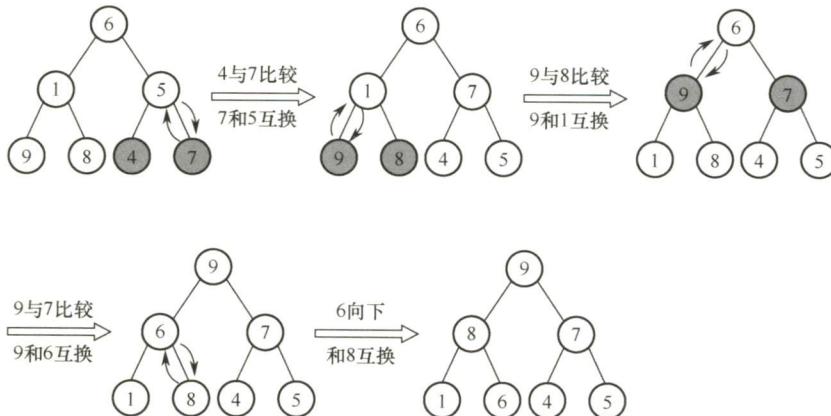
选择排序算法的比较次数始终为 $n(n - 1)/2$ ，与序列状态无关。

14. C

删除 8 后，将 12 移动到堆顶，第一次是 15 和 10 比较，第二次是 10 和 12 比较并交换，第三次还需比较 12 和 16，故比较次数为 3。

15. A

要熟练掌握建堆和调整堆的方法，从序列末尾开始向前遍历，变换过程如下图所示。



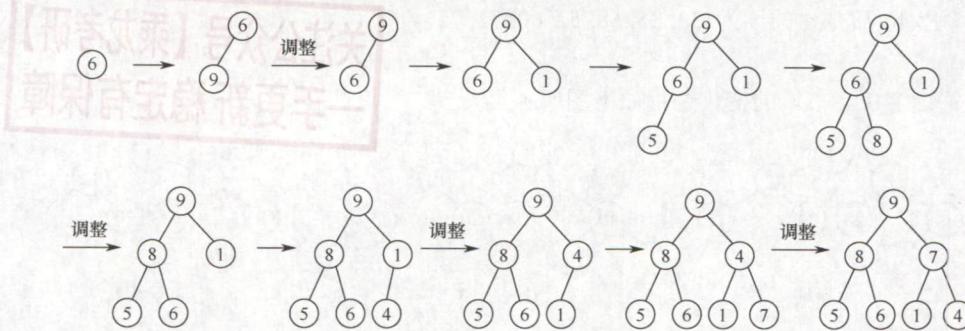
16. C

这是一道简单的概念题。堆是一棵完全树，采用一维数组存储，故选项 I 正确，选项 II 正确。

大根堆只要求根结点值大于左右孩子值，并不要求左右孩子值有序，所以堆的次大值一定是其左孩子或右孩子，选项 III 错误。堆的定义是递归的，所以其左右子树也是大根堆，所以堆的次大值一定是其左孩子或右孩子，选项 IV 正确。

17. B

要熟练掌握调整堆的方法，建堆的过程如下图所示，故答案选择选项 B。



二、综合应用题

01. 【解答】

以小根堆为例，堆的特点是双亲结点的关键字必然小于或等于该孩子结点的关键字，而两个孩子结点的关键字没有次序规定。在二叉排序树中，每个双亲结点的关键字均大于左子树结点的关键字，均小于右子树结点的关键字，也就是说，每个双亲结点的左、右孩子的关键字有次序关系。这样，当对两种树执行中序遍历后，二叉排序树会得到一个有序的序列，而堆则不一定能得到一个有序的序列。

02. 【解答】

在基于比较的排序方法中，插入排序、快速排序和归并排序只有在将元素全部排完序后，才能得到前 k 小的元素序列，算法的效率不高。

冒泡排序、堆排序和简单选择排序可以，因为它们在每一趟中都可以确定一个最小的元素。采用堆排序最合适，对于 n 个元素的序列，建立初始堆的时间不超过 $4n$ ，取得第 k 个最小元素之前的排序序列所花的时间为 $k \log_2 n$ ，总时间为 $4n + k \log_2 n$ ；冒泡和简单选择排序完成此功能所花时间为 kn ，当 $k \geq 5$ 时，通过比较可以得出堆排序最优。

注意：从本题可以得出结论，只需得到前 k 小元素的顺序排列可采用的排序算法有冒泡排序、堆排序和简单选择排序。

03. 【解答】

将 K_{n+1} 插入数组的第 $n+1$ 个位置（即作为一个树叶插入），然后将其与双亲比较，若它大于其双亲则停止调整，否则将 K_{n+1} 与其双亲交换，重复地将 K_{n+1} 与其新的双亲比较，算法终止于 K_{n+1} 大于或等于其双亲或 K_{n+1} 本身已上升为根。

04. 【解答】

算法的思想是：每趟在原始链表中摘下关键字最大的结点，把它插入结果链表的最前端。由于在原始链表中摘下的关键字越来越小，在结果链表前端插入的关键字也越来越小，因此最后形成的结果链表中的结点将按关键字非递减的顺序有序链接。

单链表的定义如第 2 章所述，假设它不带表头结点。

```
void selectSort(LinkedList& L) {
    //对不带表头结点的单链表 L 执行简单选择排序
```

```

LinkNode *h=L,*p,*q,*r,*s;
L=NULL;
while(h!=NULL) { //持续扫描原链表
    p=s=h;q=r=NULL;
    //指针 s 和 r 记忆最大结点和其前驱; p 为工作指针, q 为其前驱
    while(p!=NULL) { //扫描原链表寻找最大结点 s
        if(p->data>s->data){s=p;r=q;} //找到更大的, 记忆它和它的前驱
        q=p;p=p->link; //继续寻找
    }
    if(s==h)
        h=h->link; //最大结点在原链表前端
    else
        r->link=s->link; //最大结点在原链表表内
    s->link=L;L=s; //结点 s 插入结果链前端
}
}
}

```

05. 【解答】

将顺序表 $L[1..n]$ 视为一个完全二叉树，扫描所有分支结点，遇到孩子结点的关键字小于根结点的关键字时返回 `false`，扫描完后返回 `true`。算法的实现如下：

```

bool IsMinHeap(ElemType A[], int len){
    if(len%2==0){ //len 为偶数, 有一个单分支结点
        if(A[len/2]>A[len]) //判断单分支结点
            return false;
        for(i=len/2-1;i>=1;i--) //判断所有双分支结点
            if(A[i]>A[2*i]||A[i]>A[2*i+1])
                return false;
    }
    else{ //len 为奇数时, 没有单分支结点
        for(i=len/2;i>=1;i--) //判断所有双分支结点
            if(A[i]>A[2*i]||A[i]>A[2*i+1])
                return false;
    }
    return true;
}

```

关注公众号【乘龙考研】
一手更新 稳定有保障

06. 【解答】

1) 算法思想。

【方法1】 定义含 10 个元素的数组 A，初始时元素值均为该数组类型能表示的最大数 MAX。
`for M 中的每个元素 s`

`if (s<A[9]) 丢弃 A[9] 并将 s 按升序插入到 A 中;`

`当数据全部扫描完毕，数组 A[0]~A[9] 保存的即是最大的 10 个数。`

【方法2】 定义含 10 个元素的大根堆 H，元素值均为该堆元素类型能表示的最大数 MAX。
`for M 中的每个元素 s`

`if (s<H 的堆顶元素) 删除堆顶元素并将 s 插入到 H 中;`

`当数据全部扫描完毕，堆 H 中保存的即是最大的 10 个数。`

2) 算法平均情况下的时间复杂度是 $O(n)$ ，空间复杂度是 $O(1)$ 。

8.5 归并排序和基数排序

8.5.1 归并排序

归并排序与上述基于交换、选择等排序的思想不一样，“归并”的含义是将两个或两个以上的有序表合并成一个新的有序表。假定待排序表含有 n 个记录，则可将其视为 n 个有序的子表，每个子表的长度为 1，然后两两归并，得到 $\lceil n/2 \rceil$ 个长度为 2 或 1 的有序表；继续两两归并……如此重复，直到合并成一个长度为 n 的有序表为止，这种排序方法称为 2 路归并排序。

图 8.8 所示为 2 路归并排序的一个例子，经过三趟归并后合并成了有序序列。

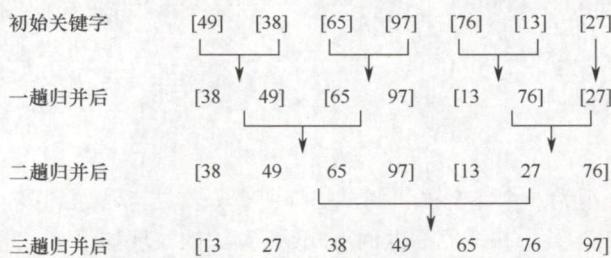
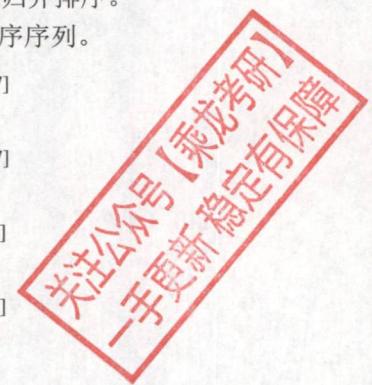


图 8.8 2 路归并排序示例



`Merge()` 的功能是将前后相邻的两个有序表归并为一个有序表。设两段有序表 $A[low...mid]$ 、 $A[mid+1...high]$ 存放在同一顺序表中的相邻位置，先将它们复制到辅助数组 B 中。每次从对应 B 中的两个段取出一个记录进行关键字的比较，将较小者放入 A 中，当数组 B 中有一段的下标超出其对应的表长（即该段的所有元素都已复制到 A 中）时，将另一段中的剩余部分直接复制到 A 中。算法如下：

```

ElemType *B=(ElemType *)malloc((n+1)*sizeof(ElemType)); //辅助数组 B
void Merge(ElemType A[], int low, int mid, int high) {
    //表 A 的两段 A[low...mid] 和 A[mid+1...high] 各自有序，将它们合并成一个有序表
    int i, j, k;
    for(k=low; k<=high; k++)
        B[k]=A[k]; //将 A 中所有元素复制到 B 中
    for(i=low, j=mid+1, k=i; i<=mid && j<=high; k++) {
        if(B[i]<=B[j]) //比较 B 的左右两段中的元素
            A[k]=B[i++]; //将较小值复制到 A 中
        else
            A[k]=B[j++];
    }
    while(i<=mid) A[k++]=B[i++]; //若第一个表未检测完，复制
    while(j<=high) A[k++]=B[j++]; //若第二个表未检测完，复制
}

```

注意：上面的代码中，最后两个 `while` 循环只有一个会执行。

一趟归并排序的操作是，调用 $\lceil n/2h \rceil$ 次算法 `merge()`，将 $L[1...n]$ 中前后相邻且长度为 h 的有序段进行两两归并，得到前后相邻、长度为 $2h$ 的有序段，整个归并排序需要进行 $\lceil \log_2 n \rceil$ 趟。

递归形式的 2 路归并排序算法是基于分治的，其过程如下。

分解：将含有 n 个元素的待排序表分成各含 $n/2$ 个元素的子表，采用 2 路归并排序算法对两个子表递归地进行排序。

合并：合并两个已排序的子表得到排序结果。

```
void MergeSort(ElemType A[], int low, int high) {
    if (low < high) {
        int mid = (low + high) / 2; // 从中间划分两个子序列
        MergeSort(A, low, mid); // 对左侧子序列进行递归排序
        MergeSort(A, mid + 1, high); // 对右侧子序列进行递归排序
        Merge(A, low, mid, high); // 归并
    } // if
}
```

2 路归并排序算法的性能分析如下：

关注公众号【乘龙考研】
一手更新 稳定有保障

空间效率：Merge() 操作中，辅助空间刚好为 n 个单元，所以算法的空间复杂度为 $O(n)$ 。

时间效率：每趟归并的时间复杂度为 $O(n)$ ，共需进行 $\lceil \log_2 n \rceil$ 趟归并，所以算法的时间复杂度为 $O(n \log_2 n)$ 。

稳定性：由于 Merge() 操作不会改变相同关键字记录的相对次序，所以 2 路归并排序算法是一种稳定的排序方法。

注意：一般而言，对于 N 个元素进行 k 路归并排序时，排序的趟数 m 满足 $k^m = N$ ，从而 $m = \log_k N$ ，又考虑到 m 为整数，所以 $m = \lceil \log_k N \rceil$ 。这和前面的 2 路归并是一致的。

8.5.2 基数排序

基数排序是一种很特别的排序方法，它不基于比较和移动进行排序，而基于关键字各位的大小进行排序。基数排序是一种借助多关键字排序的思想对单逻辑关键字进行排序的方法。

假设长度为 n 的线性表中每个结点 a_j 的关键字由 d 元组 $(k_j^{d-1}, k_j^{d-2}, \dots, k_j^1, k_j^0)$ 组成，满足 $0 \leq k_j^i \leq r - 1$ ($0 \leq j < n, 0 \leq i \leq d - 1$)。其中 k_j^{d-1} 为最主位关键字， k_j^0 为最次位关键字。

为实现多关键字排序，通常有两种方法：第一种是最高位优先（MSD）法，按关键字位权重递减依次逐层划分成若干更小的子序列，最后将所有子序列依次连接成一个有序序列。第二种是最低位优先（LSD）法，按关键字位权重递增依次进行排序，最后形成一个有序序列。

下面描述以 r 为基数的最低位优先基数排序的过程，在排序过程中，使用 r 个队列 Q_0, Q_1, \dots, Q_{r-1} 。基数排序的过程如下：

对 $i = 0, 1, \dots, d - 1$ ，依次做一次“分配”和“收集”（其实是一次稳定的排序过程）。

分配：开始时，把 Q_0, Q_1, \dots, Q_{r-1} 各个队列置成空队列，然后依次考察线性表中的每个结点 a_j ($j = 0, 1, \dots, n - 1$)，若 a_j 的关键字 $k_j^i = k$ ，就把 a_j 放进 Q_k 队列中。

收集：把 Q_0, Q_1, \dots, Q_{r-1} 各个队列中的结点依次首尾相接，得到新的结点序列，从而组成新的线性表。

通常采用链式基数排序，假设对如下 10 个记录进行排序：



每个关键字是 1000 以下的正整数，基数 $r = 10$ ，在排序过程中需要借助 10 个链队列，每个关键字由 3 位子关键字构成 $K^1 K^2 K^3$ ，分别代表百位、十位和个位，一共需要进行三趟“分配”和“收集”操作。第一趟分配用最低位子关键字 K^3 进行，将所有最低位子关键字（个位）相等的记

录分配到同一个队列，如图 8.9(a)所示，然后进行收集操作，第一趟收集后的结果如图 8.9(b)所示。

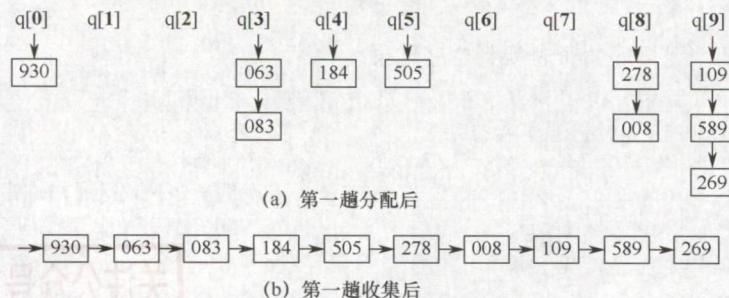


图 8.9 第一趟链式基数排序操作

第二趟分配用次低位子关键字 K^2 进行，将所有次低位子关键字（十位）相等的记录分配到同一个队列，如图 8.10(a)所示，第二趟收集后的结果如图 8.10(b)所示。

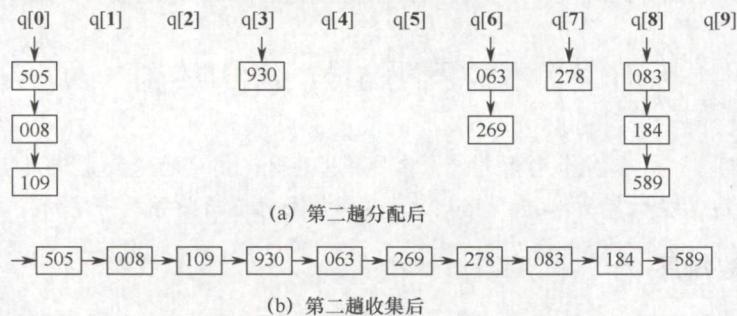


图 8.10 第二趟链式基数排序操作

第三趟分配用最高位子关键字 K^1 进行，将所有最高位子关键字（百位）相等的记录分配到同一个队列，如图 8.11(a)所示，第三趟收集后的结果如图 8.11(b)所示，至此整个排序结束。

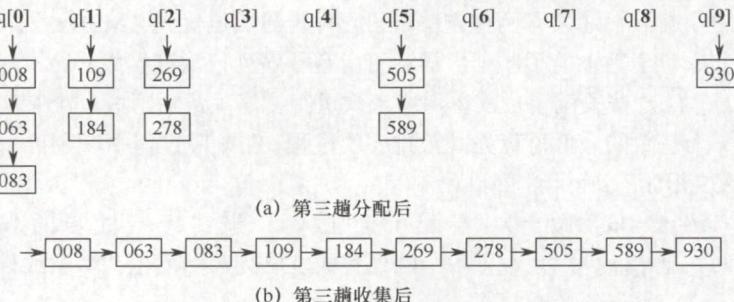


图 8.11 第三趟链式基数排序操作

基数排序算法的性能分析如下。

空间效率：一趟排序需要的辅助存储空间为 r (r 个队列: r 个队头指针和 r 个队尾指针)，但以后的排序中会重复使用这些队列，所以基数排序的空间复杂度为 $O(r)$ 。

时间效率：基数排序需要进行 d 趟分配和收集，一趟分配需要 $O(n)$ ，一趟收集需要 $O(r)$ ，所以基数排序的时间复杂度为 $O(d(n + r))$ ，它与序列的初始状态无关。

稳定性：对于基数排序算法而言，很重要一点就是按位排序时必须是稳定的。因此，这也保证了基数排序的稳定性。

关注公众号【乘龙考研】
一手更新 稳定有保障

8.5.3 本节试题精选

一、单项选择题

01. 以下排序方法中, () 在一趟结束后不一定能选出一个元素放在其最终位置上。
 A. 简单选择排序 B. 冒泡排序 C. 归并排序 D. 堆排序
02. 以下排序算法中, () 不需要进行关键字的比较。
 A. 快速排序 B. 归并排序 C. 基数排序 D. 堆排序
03. 在下列排序算法中, 平均情况下空间复杂度为 $O(n)$ 的是 (); 最坏情况下空间复杂度为 $O(n)$ 的是 ()。
 I. 希尔排序 II. 堆排序 III. 冒泡排序
 IV. 归并排序 V. 快速排序 VI. 基数排序
 A. I、IV、VI B. II、V C. IV、V D. IV
04. 下列排序方法中, 排序过程中比较次数的数量级与序列初始状态无关的是 ()。
 A. 归并排序 B. 插入排序 C. 快速排序 D. 冒泡排序
05. 2路归并排序中, 归并趟数的数量级是 ()。
 A. $O(n)$ B. $O(\log_2 n)$ C. $O(n \log_2 n)$ D. $O(n^2)$
06. 若对 27 个元素只进行三趟多路归并排序, 则选取的归并路数最少为 ()。
 A. 2 B. 3 C. 4 D. 5
07. 将两个各有 N 个元素的有序表合并成一个有序表, 最少的比较次数是 (), 最多的比较次数是 ()。
 A. N B. $2N - 1$ C. $2N$ D. $N - 1$
08. 一组经过第一趟 2 路归并排序后的记录的关键字为 {25, 50, 15, 35, 80, 85, 20, 40, 36, 70}, 其中包含 5 个长度为 2 的有序表, 用 2 路归并排序方法对该序列进行第二趟归并后的结果为 ()。
 A. 15, 25, 35, 50, 80, 20, 85, 40, 70, 36 B. 15, 25, 35, 50, 20, 40, 80, 85, 36, 70
 C. 15, 25, 50, 35, 80, 85, 20, 36, 40, 70 D. 15, 25, 35, 50, 80, 20, 36, 40, 70, 85
09. 若将中国人按照生日 (不考虑年份, 只考虑月、日) 来排序, 则使用下列排序算法时, 最快的是 ()。
 A. 归并排序 B. 希尔排序 C. 快速排序 D. 基数排序
10. 对 {05, 46, 13, 55, 94, 17, 42} 进行基数排序, 一趟排序的结果是 ()。
 A. 05, 46, 13, 55, 94, 17, 42 B. 05, 13, 17, 42, 46, 55, 94
 C. 42, 13, 94, 05, 55, 46, 17 D. 05, 13, 46, 55, 17, 42, 94
11. 【2013 统考真题】对给定的关键字序列 110, 119, 007, 911, 114, 120, 122 进行基数排序, 第 2 趟分配收集后得到的关键字序列是 ()。
 A. 007, 110, 119, 114, 911, 120, 122 B. 007, 110, 119, 114, 911, 122, 120
 C. 007, 110, 911, 114, 119, 120, 122 D. 110, 120, 911, 122, 114, 007, 119
12. 【2016 统考真题】对 10TB 的数据文件进行排序, 应使用的方法是 ()。
 A. 希尔排序 B. 堆排序 C. 快速排序 D. 归并排序
13. 【2017 统考真题】在内部排序时, 若选择了归并排序而未选择插入排序, 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
 A. 仅 II B. 仅 III C. 仅 I、II D. 仅 I、III
14. 【2021 统考真题】设数组 $S[] = \{93, 946, 372, 9, 146, 151, 301, 485, 236, 327, 43, 892\}$, 采用最低位优先 (LSD) 基数排序将 S 排列成升序序列。第一趟分配、收集后, 元素 372 之前、之后紧邻的元素分别是 ()。
 A. 43, 892 B. 236, 301 C. 301, 892 D. 485, 301
15. 【2022 统考真题】使用二路归并排序对含 n 个元素的数组 M 进行排序时, 二路归并操作的功能是 ()。
 A. 将两个有序表合并为一个新的有序表
 B. 将 M 划分为两部分, 两部分的元素个数大致相等
 C. 将 M 划分为 n 个部分, 每个部分中仅含有一个元素
 D. 将 M 划分为两部分, 一部分元素的值均小于另一部分元素的值

二、综合应用题

01. 已知序列 $\{503, 87, 512, 61, 908, 170, 897, 275, 653, 462\}$, 采用 2 路归并排序法对该序列做升序排序时需要几趟排序? 给出每一趟的结果。
02. 设待排序的排序码序列为 $\{12, 2, 16, 30, 28, 10, 16^*, 20, 6, 18\}$, 试写出使用基数排序方法每趟排序后的结果, 并说明做了多少次排序码比较。

8.5.4 答案与解析

一、单项选择题

01. C

前面我们知道插入排序不能保证在一趟结束后一定有元素放在最终位置上。事实上, 归并排序也不能保证。例如, 序列 $\{6, 5, 7, 8, 2, 1, 4, 3\}$ 进行一趟 2 路归并排序 (从小到大) 后为 $\{5, 6, 7, 8, 1, 2, 3, 4\}$, 显然它们都未被放在最终位置上。

02. C

基数排序是基于关键字各位的大小进行排序的, 而不是基于关键字的比较进行的。

03. D、C

归并排序算法在平均情况下和最坏情况下的空间复杂度都会达到 $O(n)$, 快速排序只在最坏情况下才会达到 $O(n)$, 平均情况下为 $O(\log_2 n)$ 。所以归并排序算法可视为本章所有算法中占用辅助空间最多的排序算法。

04. A

前面已讲过选择排序的比较次数与序列初始状态无关, 归并排序的比较次数的数量级也与序列的初始状态无关。读者应能从算法的原理方面来考虑为什么和初始状态无关。

05. B

对于 N 个元素进行 k 路归并排序时, 排序的趟数 m 满足 $k^m = N$, 所以 $m = \lceil \log_k N \rceil$, 本题中即为 $\lceil \log_2 n \rceil$ 。

06. B

利用上题中的公式, 这里要求的是 k , 代入可得 $k = 3$ 。

07. A、B

注意到当一个表中的最小元素比另一个表中的最大元素还大时, 比较的次数是最少的, 仅比较 N 次; 而当两个表中的元素依次间隔地比较时, 即 $a_1 < b_1 < a_2 < b_2 < \dots < a_n < b_n$ 时, 比较的次数

是最多的，为 $2N-1$ 次。

建议读者对此举一反三：若将本题的中的两个有序表的长度分别设为 M 和 N ，则最多（或最少）的比较次数是多少？时间复杂度又是多少？

08. B

由于这里采用2路归并排序算法，而且是第二趟排序，因此每4个元素放在一起归并，可将序列划分为{25, 50, 15, 35}, {80, 85, 20, 40}和{36, 70}，分别对它们进行排序后有{15, 25, 35, 50}, {20, 40, 80, 85}和{36, 70}，故选择选项B。

09. D

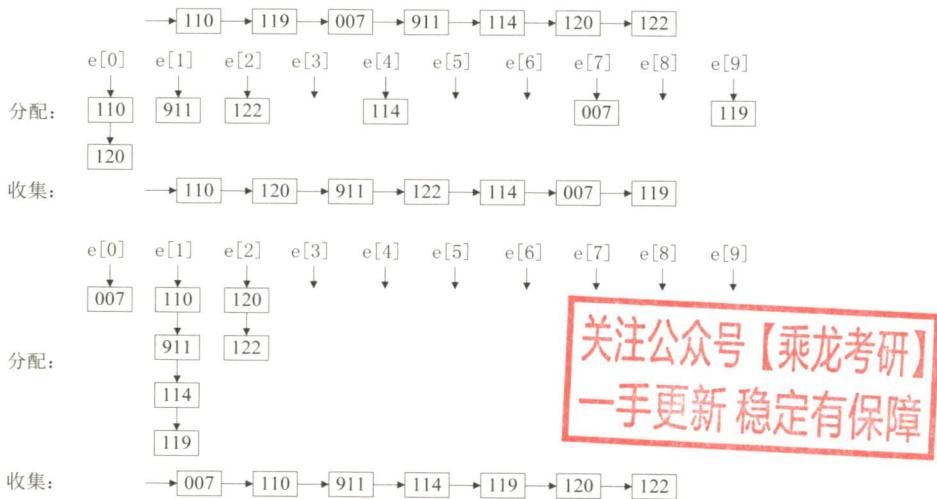
按照所有中国人的生日排序，一方面 N 是非常大的，另一方面关键字所含的排序码数为2，且一个排序码的基数为12，另一个排序码的基数为31，都是较小的常数值，因此采用基数排序可以在 $O(N)$ 内完成排序过程。

10. C

基数排序有MSD和LSD两种，且基数排序是稳定的。对于选项A，不符合LSD和MSD；对于选项B，符合MSD，但关键字42、46的相对位置发生了变化；对于选项D，不符合LSD和MSD。

11. C

基数排序的第一趟排序是按照个位数字的大小来进行的，第二趟排序是按照十位数字的大小来进行的，排序的过程如下图所示。



12. D

外部排序指待排序文件较大，内存一次性放不下，需存放在外部介质中。外部排序通常采用归并排序法。选项A、B、C都是内部排序的方法。

13. B

归并排序代码比选择插入排序更复杂，前者的空间复杂度是 $O(n)$ ，后者的是 $O(1)$ 。但前者的时间复杂度是 $O(n \log n)$ ，后者的是 $O(n^2)$ 。所以选项B正确。

14. C

基数排序是一种稳定的排序方法。由于采用最低位优先(LSD)的基数排序，即第一趟对个位进行分配和收集操作，因此第一趟分配和收集后的结果是{151, 301, 372, 892, 93, 43, 485, 946, 146, 236, 327, 9}，元素372之前、之后紧邻的元素分别是301和892。

15. A。

送分题。本书对归并的定义原话是“归并的含义是将两个或两个以上的有序表合并成一个新的有序表”，而二路归并是将两个有序表合并为一个新的有序表。

二、综合应用题**01. 【解答】**

$n = 10$, 需要排序的趟数 $= \lceil \log_2 10 \rceil = 4$, 各趟的排序结果如下:

初始序列: 503, 87, 512, 61, 908, 170, 897, 275, 653, 462

第一趟: 87, 503, 61, 512, 170, 908, 275, 897, 462, 653 (长度为 2)

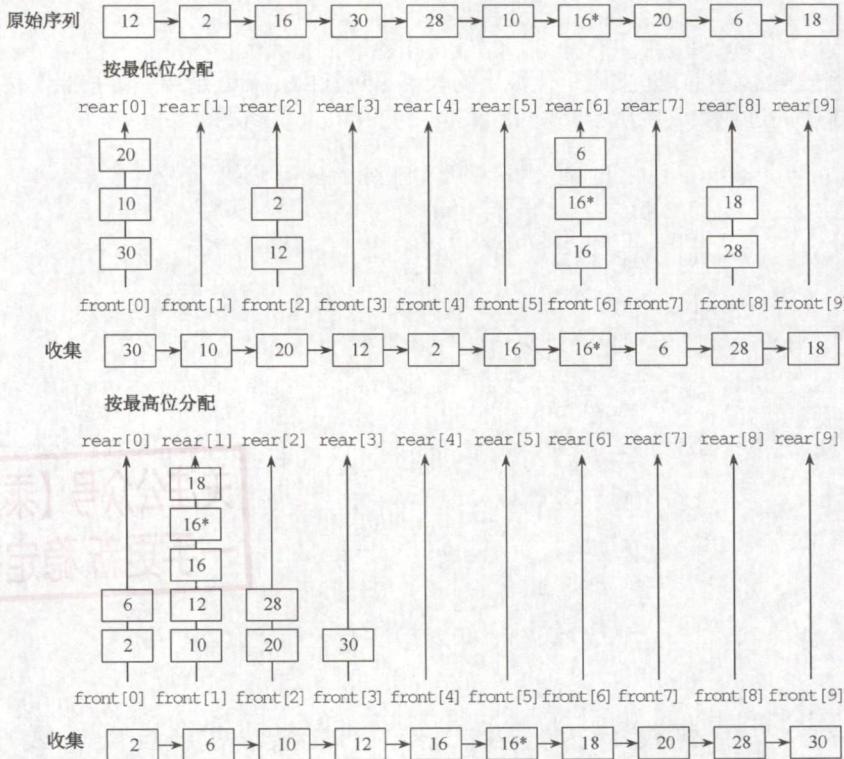
第二趟: 61, 87, 503, 512, 170, 275, 897, 908, 462, 653 (长度为 4)

第三趟: 61, 87, 170, 275, 503, 512, 897, 908, 462, 653 (长度为 8)

第四趟: 61, 87, 170, 275, 462, 503, 512, 653, 897, 908 (长度为 10)

02. 【解答】

使用链式队列的基数排序的排序过程如下图所示。



需要通过 2 次“分配”和“收集”完成排序。

8.6 各种内部排序算法的比较及应用

8.6.1 内部排序算法的比较

前面讨论的排序算法很多，对各种排序算法的比较是考研常考的内容。一般基于三个因素进

行对比：时空复杂度、算法的稳定性、算法的过程特征。

从时间复杂度看：简单选择排序、直接插入排序和冒泡排序平均情况下的时间复杂度都为 $O(n^2)$ ，且实现过程也较为简单，但直接插入排序和冒泡排序最好情况下的时间复杂度可以达到 $O(n)$ ，而简单选择排序则与序列的初始状态无关。希尔排序作为插入排序的拓展，对较大规模的数据都可以达到很高的效率，但目前未得出其精确的渐近时间。堆排序利用了一种称为堆的数据结构，可以在线性时间内完成建堆，且在 $O(n\log_2 n)$ 内完成排序过程。快速排序基于分治的思想，虽然最坏情况下的时间复杂度会达到 $O(n^2)$ ，但快速排序的平均性能可以达到 $O(n\log_2 n)$ ，在实际应用中常常优于其他排序算法。归并排序同样基于分治的思想，但由于其分割子序列与初始序列的排列无关，因此它的最好、最坏和平均时间复杂度均为 $O(n\log_2 n)$ 。

从空间复杂度看：简单选择排序、插入排序、冒泡排序、希尔排序和堆排序都仅需借助常数个辅助空间。快速排序需要借助一个递归工作栈，平均大小为 $O(\log_2 n)$ ，当然在最坏情况下可能会增长到 $O(n)$ 。2 路归并排序在合并操作中需要借助较多的辅助空间用于元素复制，大小为 $O(n)$ ，虽然有方法能克服这个缺点，但其代价是算法会很复杂而且时间复杂度会增加。

从稳定性看：插入排序、冒泡排序、归并排序和基数排序是稳定的排序方法，而简单选择排序、快速排序、希尔排序和堆排序都是不稳定的排序方法。平均时间复杂度为 $O(n\log_2 n)$ 的稳定排序算法只有归并排序，对于不稳定的排序方法，只需举出一个不稳定的实例即可。对于排序方法的稳定性，读者应能从算法本身的原理上去理解，而不应拘泥于死记硬背。

从过程特征看：采用不同的排序算法，在一次循环或几次循环后的排序结果可能是不同的，考研题中经常出现给出一个待排序的初始序列和已经部分排序的序列，问其采用何种排序算法。这就要对各类排序算法的过程特征十分熟悉，如冒泡排序和堆排序在每趟处理后都能产生当前的最大值或最小值，而快速排序一趟处理至少能确定一个元素的最终位置等。

表 8.1 列出了各种排序算法的时空复杂度和稳定性情况，其中空间复杂度仅列举了平均情况的复杂度，由于希尔排序的时间复杂度依赖于增量函数，所以无法准确给出其时间复杂度。

表 8.1 各种排序算法的性质

算法种类	时间复杂度			空间复杂度	是否稳定
	最好情况	平均情况	最坏情况		
直接插入排序	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	是
冒泡排序	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	是
简单选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	否
希尔排序				$O(1)$	否
快速排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n^2)$	$O(\log_2 n)$	否
堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	否
2 路归并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	是
基数排序	$O(d(n + r))$	$O(d(n + r))$	$O(d(n + r))$	$O(r)$	是

8.6.2 内部排序算法的应用

通常情况，对排序算法的比较和应用应考虑以下情况。

1) 选取排序方法需要考虑的因素

- ① 待排序的元素数目 n 。
- ② 元素本身信息量的大小。
- ③ 关键字的结构及其分布情况。

关注公众号【乘龙考研】
一手更新 稳定有保障

关注公众号【乘龙考研】
一手更新 稳定有保障

- ④ 稳定性的要求。
- ⑤ 语言工具的条件，存储结构及辅助空间的大小等。

2) 排序算法小结

- ① 若 n 较小，可采用直接插入排序或简单选择排序。由于直接插入排序所需的记录移动次数较简单选择排序的多，因而当记录本身信息量较大时，用简单选择排序较好。
- ② 若文件的初始状态已按关键字基本有序，则选用直接插入或冒泡排序为宜。
- ③ 若 n 较大，则应采用时间复杂度为 $O(n \log_2 n)$ 的排序方法：快速排序、堆排序或归并排序。快速排序被认为是目前基于比较的内部排序方法中最好的方法，当待排序的关键字随机分布时，快速排序的平均时间最短。堆排序所需的辅助空间少于快速排序，并且不会出现快速排序可能出现的最坏情况，这两种排序都是不稳定的。若要求排序稳定且时间复杂度为 $O(n \log_2 n)$ ，则可选用归并排序。但本章介绍的从单个记录起进行两两归并的排序算法并不值得提倡，通常可以将它和直接插入排序结合在一起使用。先利用直接插入排序求得较长的有序子文件，然后两两归并。直接插入排序是稳定的，因此改进后的归并排序仍是稳定的。
- ④ 在基于比较的排序方法中，每次比较两个关键字的大小之后，仅出现两种可能的转移，因此可以用一棵二叉树来描述比较判定过程，由此可以证明：当文件的 n 个关键字随机分布时，任何借助于“比较”的排序算法，至少需要 $O(n \log_2 n)$ 的时间。
- ⑤ 若 n 很大，记录的关键字位数较少且可以分解时，采用基数排序较好。
- ⑥ 当记录本身信息量较大时，为避免耗费大量时间移动记录，可用链表作为存储结构。

8.6.3 本节试题精选

一、单项选择题

01. 若要求排序是稳定的，且关键字为实数，则在下列排序方法中应选（ ）。
 - A. 直接插入排序
 - B. 选择排序
 - C. 基数排序
 - D. 快速排序
02. 以下排序方法中时间复杂度为 $O(n \log_2 n)$ 且稳定的是（ ）。
 - A. 堆排序
 - B. 快速排序
 - C. 归并排序
 - D. 直接插入排序
03. 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有序的情况下，用直接插入排序、归并排序和快速排序对其进行排序，这些算法的时间复杂度应为（ ）。
 - A. $O(N), O(N), O(N)$
 - B. $O(N), O(N \log_2 N), O(N \log_2 N)$
 - C. $O(N), O(N \log_2 N), O(N^2)$
 - D. $O(N^2), O(N \log_2 N), O(N^2)$
04. 下列排序算法中属于稳定排序的是（①），平均时间复杂度为 $O(n \log_2 n)$ 的是（②），在最好的情况下，时间复杂度可以达到线性时间的有（③）。(注：多选题)
 - I. 冒泡排序
 - II. 堆排序
 - III. 选择排序
 - IV. 直接插入排序
 - V. 希尔排序
 - VI. 归并排序
 - VII. 快速排序
05. 就排序算法所用的辅助空间而言，堆排序、快速排序和归并排序的关系是（ ）。
 - A. 堆排序<快速排序<归并排序
 - B. 堆排序<归并排序<快速排序
 - C. 堆排序>归并排序>快速排序
 - D. 堆排序>快速排序>归并排序
06. 排序趟数与序列的原始状态无关的排序方法是（ ）。
 - I. 直接插入排序
 - II. 简单选择排序
 - III. 冒泡排序
 - IV. 基数排序
07. A. I, III
- B. I, II, IV
- C. I, II, III
- D. I, IV
07. 若序列的原始状态为 {1, 2, 3, 4, 5, 10, 6, 7, 8, 9}，要想使得排序过程中的元素比较次数最

- 少，则应该采用（ ）方法。
- A. 插入排序 B. 选择排序 C. 希尔排序 D. 冒泡排序
08. 一般情况下，以下查找效率最低的数据结构是（ ）。
- A. 有序顺序表 B. 二叉排序树 C. 堆 D. 平衡二叉树
09. 排序趟数与序列的原始状态有关的排序方法是（ ）排序法。
- A. 插入 B. 选择 C. 冒泡 D. 基数
10. 【2012 统考真题】在内部排序过程中，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，每趟排序结束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. 2路归并排序
- A. 仅 I、III、IV B. 仅 I、III、V C. 仅 II、III、IV D. 仅 III、IV、V
11. 【2015 统考真题】下列排序算法中，元素的移动次数与关键字的初始排列次序无关的是（ ）。
- A. 直接插入排序 B. 起泡排序 C. 基数排序 D. 快速排序
12. 【2017 统考真题】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
- A. 仅 I、II B. 仅 II、III C. 仅 III、IV D. 仅 IV、V
13. 【2019 统考真题】选择一个排序算法时，除算法的时空效率外，下列因素中，还需要考虑的是（ ）。
- I. 数据的规模 II. 数据的存储方式 III. 算法的稳定性 IV. 数据的初始状态
- A. 仅 III B. 仅 I、II C. 仅 II、III、IV D. I、II、III、IV
14. 【2020 统考真题】对大部分元素已有序的数组排序时，直接插入排序比简单选择排序效率更高，其原因是（ ）。
- I. 直接插入排序过程中元素之间的比较次数更少
II. 直接插入排序过程中所需的辅助空间更少
III. 直接插入排序过程中元素的移动次数更少
- A. 仅 I B. 仅 III C. 仅 I、II D. I、II 和 III
15. 【2022 统考真题】对数据进行排序时，若采用直接插入排序而不采用快速排序，则可能的原因是（ ）。
- I. 大部分元素已有序 II. 待排序元素数量很少
III. 要求空间复杂度为 $O(1)$ IV. 要求排序算法是稳定的
- A. 仅 I、II B. 仅 III、IV C. 仅 I、II、IV D. I、II、III、IV

二、综合应用题

01. 设关键字序列为 {3, 7, 6, 9, 7, 1, 4, 5, 20}，对其进行排序的最小交换次数是多少？
02. 设顺序表用数组 A[] 表示，表中元素存储在数组下标 1 ~ m+n 的范围内，前 m 个元素递增有序，后 n 个元素递增有序，设计一个算法，使得整个顺序表有序。
- 1) 给出算法的基本设计思想。
 - 2) 根据设计思想，采用 C/C++ 描述算法，关键之处给出注释。
 - 3) 说明你所设计算法的时间复杂度与空间复杂度。

关注公众号【乘龙考研】
一手更新 稳定有保障

03. 有一种简单的排序算法，称为计数排序（count sorting）。这种排序算法对一个待排序的表（用数组表示）进行排序，并将排序结果存放到另一个新的表中。必须注意的是，表中所有待排序的关键码互不相同，计数排序算法针对表中的每个记录，扫描待排序的表一趟，统计表中有多少个记录的关键码比该记录的关键码小，假设针对某个记录统计出的计数值为 c ，则这个记录在新有序表中的合适存放位置即为 c 。

- 1) 设计实现计数排序的算法。
- 2) 对于有 n 个记录的表，关键码比较次数是多少？
- 3) 与简单选择排序相比较，这种方法是否更好？为什么？

04. 设有一个数组中存放了一个无序的关键序列 K_1, K_2, \dots, K_n 。现要求将 K_n 放在将元素排序后的正确位置上，试编写实现该功能的算法，要求比较关键字的次数不超过 n 。

05. 【2021 统考真题】 已知某排序算法如下：

```
void cmpCountSort(int a[], int b[], int n) {
    int i, j, *count;
    count = (int *)malloc(sizeof(int) * n);
    //C++语言: count = new int[n];
    for (i=0; i<n; i++) count[i] = 0;
    for (i=0; i<n-1; i++)
        for (j=i+1; j<n; j++)
            if (a[i] < a[j]) count[j]++;
            else count[i]++;
    for (i=0; i<n; i++) b[count[i]] = a[i];
    free(count); //C++语言: delete count;
}
```

请回答下列问题。

- 1) 若有 $\text{int } a[] = \{25, -10, 25, 10, 11, 19\}, b[6]$ ，则调用 $\text{cmpCountSort}(a, b, 6)$ 后数组 b 中的内容是什么？
- 2) 若 a 中含有 n 个元素，则算法执行过程中，元素之间的比较次数是多少？
- 3) 该算法是稳定的吗？若是，阐述理由；否则，修改为稳定排序算法。

8.6.4 答案与解析

一、单项选择题

01. A

采用排除法。由于题目要求是稳定排序，排除选项 B 和 D，又由于基数排序不能对 float 和 double 类型的实数进行排序，故排除选项 C。

02. C

堆排序和快速排序不是稳定排序方法，而直接插入排序算法的时间复杂度为 $O(n^2)$ 。

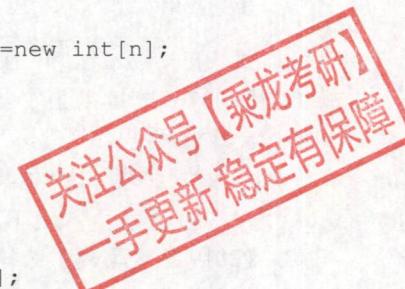
03. C

读者应熟练掌握各种排序算法的时间和空间复杂度、稳定性等，详见表 8.1。

04. ① I、IV、VI ② II、VI、VII ③ I、IV

读者应能从算法的原理上理解算法的稳定性情况。堆排序和归并排序在最坏情况下的时间复杂度与最好情况下的时间复杂度是同一数量级的，都是 $O(n \log_2 n)$ 。

05. A



由于堆排序的空间复杂度为 $O(1)$, 快速排序的空间复杂度在最坏情况下为 $O(n)$, 平均空间复杂度为 $O(\log_2 n)$, 归并排序的空间复杂度为 $O(n)$, 所以不难得出正确选项是 A。

06. B

交换类的排序, 其趟数和原始序列状态有关, 故冒泡排序与初始序列有关。直接插入排序: 每趟排序都插入一个元素, 所以排序趟数固定为 $n - 1$; 简单选择排序: 每趟排序都选出一个最小(或最大)的元素, 所以排序趟数固定为 $n - 1$; 基数排序: 每趟排序都要进行“分配”和“收集”, 排序趟数固定为 d 。

07. A

选择排序和序列初态无关, 直接排除。初始序列基本有序时, 插入排序比较次数较少。本题中, 插入排序仅需比较 $n - 1 + 4$ 次, 而希尔排序和冒泡排序的比较次数均远大于此。

08. C

堆是用于排序的, 在查找时它是无序的, 所以效率没有其他查找结构的高。

09. C

插入排序和选择排序的排序趟数始终为 $n - 1$, 与序列初态无关。对于冒泡排序, 如果初始基本有序, 某趟比较后没有发生元素交换, 则说明已排好序。对于快速排序, 若每趟的枢轴元素都能平均的划分两个子序列, 则需要的趟数最少; 若原始序列有序, 则需要的趟数最多。

10. A

对于 I, 简单选择排序每次选择未排序序列中的最小元素放入其最终位置。对于选项 II, 希尔排序每次对划分的子表进行排序, 得到局部有序的结果, 所以不能保证每趟排序结束都能确定一个元素的最终位置。对于选项 III, 快速排序每趟排序结束后都将枢轴元素放到最终位置。对于选项 IV, 堆排序属于选择排序, 每次都将大根堆的根结点与表尾结点交换, 确定其最终位置。对于选项 V, 2 路归并排序每趟对子表进行两两归并, 从而得到若干局部有序的结果, 但无法确定最终位置。

11. C

基数排序的元素移动次数与关键字的初始排列次序无关, 而其他三种排序都与关键字的初始排列明显相关。

12. D

插入排序、选择排序、起泡排序的原本时间复杂度是 $O(n^2)$, 更换为链式存储后的时间复杂度还是 $O(n^2)$ 。希尔排序和堆排序都利用了顺序存储的随机访问特性, 而链式存储不支持这种性质, 所以时间复杂度会增加, 因此选择选项 D。

注意: 时间效率降低就是指时间复杂度增加, 两者是相反的概念。

13. D

当数据规模较小时可选择复杂度为 $O(n^2)$ 的简单排序方法, 当数据规模较大时应选择复杂度为 $O(n \log_2 n)$ 的排序方法, 当数据规模大到内存无法放下时需选择外部排序方法, 说法 I 正确。数据的存储方式主要分为顺序存储和链式存储, 有些排序方法(如堆排序)只能用于顺序存储方式, 说法 II 正确。若对数据稳定性有要求, 则不能选择不稳定的排序方法, 说法 III 显然正确。当数据初始基本有序时, 直接插入排序的效率最高, 冒泡排序和直接插入排序的时间复杂度都是 $O(n)$, 而归并排序的时间复杂度依旧是 $O(n \log_2 n)$, 说法 IV 正确。所以选择选项 D。

14. A

考虑较极端的情况, 对于有序数组, 直接插入排序的比较次数为 $n - 1$, 简单选择排序的比较次数始终为 $1 + 2 + \dots + n - 1 = n(n - 1)/2$, 说法 I 正确。两种排序方法的辅助空间都是 $O(1)$, 无差

别，说法 II 错误。初始有序时，移动次数均为 0；对于通常情况，直接插入排序每趟插入都需要依次向后挪位，而简单选择排序只需与找到的最小元素交换位置，后者的移动次数少很多，说法 III 错误。

15. D

直接插入排序和快速排序的特点如下表所示。

	适合初始序列情况	适合元素数量	空间复杂度	稳定性
直接插入排序	大部分元素有序	较少	$O(1)$	稳定
快速排序	基本无序	较多	$O(\log_2 n)$	不稳定

可见，选项 I、II、III、IV 都是采用直接插入排序而不采用快速排序的可能原因。

二、综合应用题

01. 【解答】

由于关键字序列数较小，采用直接插入排序或简单选择排序，直接插入排序的交换次数更多，选择简单选择排序。

初始序列：3, 7, 6, 9, 7, 1, 4, 5, 20

第一次：1, 7, 6, 9, 7, 3, 4, 5, 20 交换 1, 3

第二次：1, 3, 6, 9, 7, 7, 4, 5, 20 交换 3, 7

第三次：1, 3, 4, 9, 7, 7, 6, 5, 20 交换 4, 6

第四次：1, 3, 4, 5, 7, 7, 6, 9, 20 交换 5, 9

第五次：1, 3, 4, 5, 6, 7, 7, 9, 20 交换 6, 7

所以最小交换次数为 5（注意这里是交换次数，不是移动次数或比较次数）。

02. 【解答】

1) 算法的基本设计思想如下：将数组 $A[1..m+n]$ 视为一个已经过 m 趟插入排序的表，则从 $m+1$ 趟开始，将后 n 个元素依次插入前面的有序表中。

2) 算法的实现如下：

```
void Insert_Sort(ElemType A[], int m, int n) {
    int i, j;
    for (i=m+1; i<=m+n; i++) {           //依次将 A[m+1..m+n] 插入有序表
        A[0]=A[i];                         //复制为哨兵
        for (j=i-1; A[j]>A[0]; j--)       //从后往前插入
            A[j+1]=A[j];                  //元素后移
        A[j+1]=A[0];                      //插入
    }
}
```

3) 时间复杂度由 m 和 n 共同决定，从上面的算法不难看出，在最坏情况下元素的比较次数为 $O(mn)$ ，而元素移动的次数为 $O(mn)$ ，所以时间复杂度为 $O(mn)$ 。

由于算法只用到了常数个辅助空间，所以空间复杂度为 $O(1)$ 。

此外，本题也可采用归并排序，将 $A[1..m]$ 和 $A[m+1..m+n]$ 视为两个待归并的有序子序列，算法的时间复杂度为 $O(m+n)$ ，空间复杂度为 $O(m+n)$ 。

03. 【解答】

1) 算法的思想：对每个元素，统计关键字比它小的元素个数，然后把它放入另一个数组对应的位置上。

算法的实现如下：

关注公众号【乘龙考研】
一手更新 稳定有保障

```

void CountSort(RecType A[], RecType B[], int n) {
    //计数排序算法，将 A 中记录排序放入 B 中
    int cnt;           //计数变量
    for(i=0; i<n; i++) { //对每个元素
        for(j=0, cnt=0; j<n; j++)
            if(A[j].key<A[i].key)
                cnt++; //统计关键字比它小的元素个数
        B[cnt]=A[i]; //放入对应的位置
    }
}

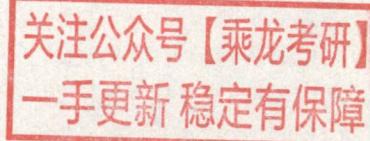
```

- 2) 对于有 n 个记录的表，每个关键码都要与 n 个记录（含自身）进行比较，因此关键码的比较次数为 n^2 。
- 3) 简单选择排序算法比本算法好。简单选择排序的比较次数是 $n(n-1)/2$ ，且只用一个交换记录的空间；而这种方法的比较次数是 n^2 ，且需要另一数组空间。另外，因题目要求“针对表中的每个记录，扫描待排序的表一趟”，所以比较次数是 n^2 。若限制“对任意两个记录之间只进行一次比较”，则可把以上算法中的比较语句改为

```

for(i=0; i<n; i++)
    a[i].count=0; //各元素再增加一个计数域，初始化为 0
for(i=0; i<n; i++)
    for(j=i+1; j<n; j++) {
        if(a[i].key<a[j].key)
            a[j].count++;
        else
            a[i].count++;
    }
}

```



04. 【解答】

基本思想：以 K_n 为枢轴进行一趟快速排序。将快速排序算法改为以最后一个为枢轴先从前向后再从后向前。算法的代码如下：

```

int Partition(ElemType K[], int n) {
    //交换序列 K[1..n] 中的记录，使枢轴到位，并返回其所在位置
    int i=1, j=n; //设置两个交替变量初值分别为 1 和 n
    ElemenType pivot=K[j]; //枢轴
    while(i<j) { //循环跳出条件
        while(i<j && K[i]<=pivot)
            i++; //从前往后找比枢轴大的元素
        if(i<j)
            K[j]=K[i]; //移动到右端
        while(i<j && K[j]>=pivot)
            j--; //从后往前找比枢轴小的元素
        if(i<j)
            K[i]=K[j]; //移动到左端
    } //while
    K[i]=pivot; //枢轴存放在最终位置
    return i; //返回存放枢轴的位置
}

```

05. 【解答】

cmpCountSort 算法基于计数排序的思想，对序列进行排序。cmpCountSort 算法遍历数组中的元素，count 数组记录比对应待排序数组元素下标大的元素个数，例如， $count[1]=3$ 的意思

是数组 a 中有 3 个元素比 $a[1]$ 小，即 $a[1]$ 是第 4 大元素， $a[1]$ 的正确位置应是 $b[3]$ 。

- 1) 排序结果为 $b[6]=\{-10, 10, 11, 19, 25, 25\}$ 。
- 2) 由代码 `for(i=0; i<n-1; i++)` 和 `for(j=i+1; j<n; j++)` 可知，在循环过程中，每个元素都与它后面的所有元素比较一次（即所有元素都两两比较一次），比较次数之和为 $(n-1)+(n-2)+\dots+1$ ，故总比较次数是 $n(n-1)/2$ 。
- 3) 不是。需要将程序中的 `if` 语句修改如下：

```
if(a[i]<=a[j]) count[j]++;
else count[i]++;
```

如果不加等号，两个相等的元素比较时，前面元素的 `count` 值会加 1，导致原序列中靠前的元素在排序后的序列中处于靠后的位置。

关注公众号【乘龙考研】
一手更新 稳定有保障

8.7 外部排序

外部排序可能会考查相关概念、方法和排序过程，外部排序的算法比较复杂，不会在算法设计上进行考查。本节的主要内容有：

- ① 外部排序指待排序文件较大，内存一次放不下，需存放在外存的文件的排序。
- ② 为减少平衡归并中外存读写次数所采取的方法：增大归并路数和减少归并段个数。
- ③ 利用败者树增大归并路数。
- ④ 利用置换-选择排序增大归并段长度来减少归并段个数。
- ⑤ 由长度不等的归并段，进行多路平衡归并，需要构造最佳归并树。

8.7.1 外部排序的基本概念

前面介绍过的排序方法都是在内存中进行的（称为内部排序）。而在许多应用中，经常需要对大文件进行排序，因为文件中的记录很多，无法将整个文件复制进内存中进行排序。因此，需要将待排序的记录存储在外存上，排序时再把数据一部分一部分地调入内存进行排序，在排序过程中需要多次进行内存和外存之间的交换。这种排序方法就称为外部排序。

8.7.2 外部排序的方法

文件通常是按块存储在磁盘上的，操作系统也是按块对磁盘上的信息进行读写的。因为磁盘读/写的机械动作所需的时间远远超过内存运算的时间（相比而言可以忽略不计），因此在外部排序过程中的时间代价主要考虑访问磁盘的次数，即 I/O 次数。

外部排序通常采用归并排序法。它包括两个阶段：①根据内存缓冲区大小，将外存上的文件分成若干长度为 ℓ 的子文件，依次读入内存并利用内部排序方法对它们进行排序，并将排序后得到的有序子文件重新写回外存，称这些有序子文件为归并段或顺串；②对这些归并段进行逐趟归并，使归并段（有序子文件）逐渐由小到大，直至得到整个有序文件为止。

例如，一个含有 2000 个记录的文件，每个磁盘块可容纳 125 个记录，首先通过 8 次内部排序得到 8 个初始归并段 R1~R8，每个段都含 250 个记录。然后对该文件做如图 8.13 所示的两两归并，直至得到一个有序文件。可以把内存工作区等分为 3 个缓冲区，如图 8.12 所示，其中的两个为输入缓冲区，一个为输出缓冲区。首先，从两个输入归并段 R1 和 R2 中分别读入一个块，放在输入缓冲区 1 和输入缓冲区 2 中。然后，在内存中进行 2 路归并，归并后的对象顺序存放在输出缓冲区中。若输出缓冲区中对象存满，则将其顺序写到输出归并段 (R1') 中，再清空输出缓冲

区，继续存放归并后的对象。若某个输入缓冲区中的对象取空，则从对应的输入归并段中再读取下一块，继续参加归并。如此继续，直到两个输入归并段中的对象全部读入内存并都归并完成为止。当 R1 和 R2 归并完后，再归并 R3 和 R4、R5 和 R6、最后归并 R7 和 R8，这是一趟归并。再把上趟的结果 R1' 和 R2'、R3' 和 R4' 两两归并，这又是一趟归并。最后把 R1'' 和 R2'' 两个归并段归并，结果得到最终的有序文件，一共进行了 3 趟归并。

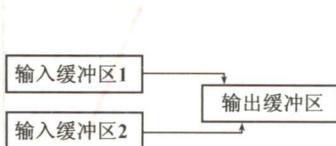


图 8.12 2 路归并

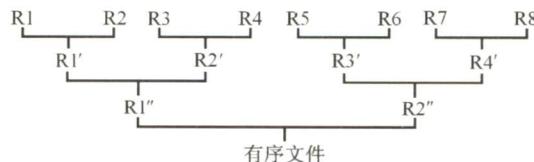


图 8.13 2 路平衡归并的排序过程

在外部排序中实现两两归并时，由于不可能将两个有序段及归并结果段同时存放在内存中，因此需要不停地将数据读出、写入磁盘，而这些会耗费大量的时间。一般情况下：

$$\text{外部排序的总时间} = \text{内部排序所需的时间} + \text{外存信息读写的时间} + \text{内部归并所需的时间}$$

显然，外存信息读写的时间远大于内部排序和内部归并的时间，因此应着力减少 I/O 次数。由于外存信息的读/写是以“磁盘块”为单位的，可知每一趟归并需进行 16 次“读”和 16 次“写”，3 趟归并加上内部排序时所需进行的读/写，使得总共需进行 $32 \times 3 + 32 = 128$ 次读写。

若改用 4 路归并排序，则只需 2 趟归并，外部排序时的总读/写次数便减至 $32 \times 2 + 32 = 96$ 。因此，增大归并路数，可减少归并趟数，进而减少总的磁盘 I/O 次数，如图 8.14 所示。

一般地，对 r 个初始归并段，做 k 路平衡归并，归并树可用严格 k 叉树（即只有度为 k 与度为 0 的结点的 k 叉树）来表示。第一趟可将 r 个初始归并段归并为 $\lceil r/k \rceil$ 个归并段，以后每趟归并将 m 个归并段归并成 $\lceil m/k \rceil$ 个归并段，直至最后形成一个大的归并段为止。树的高度 $-1 = \lceil \log_k r \rceil =$ 归并趟数 S 。可见，只要增大归并路数 k ，或减少初始归并段个数 r ，都能减少归并趟数 S ，进而减少读写磁盘的次数，达到提高外部排序速度的目的。

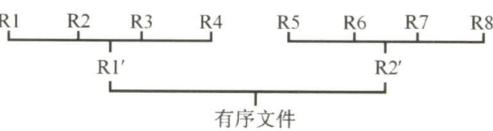


图 8.14 4 路平衡归并的排序过程

8.7.3 多路平衡归并与败者树

上节讨论过，增加归并路数 k 能减少归并趟数 S ，进而减少 I/O 次数。然而，增加归并路数 k 时，内部归并的时间将增加。做内部归并时，在 k 个元素中选择关键字最小的记录需要比较 $k-1$ 次。每趟归并 n 个元素需要做 $(n-1)(k-1)$ 次比较， S 趟归并总共需要的比较次数为

$$S(n-1)(k-1) = \lceil \log_k r \rceil (n-1)(k-1) = \lceil \log_2 r \rceil (n-1)(k-1) / \lceil \log_2 k \rceil$$

式中， $(k-1)\lceil \log_2 k \rceil$ 随 k 增长而增长，因此内部归并时间亦随 k 的增长而增长。这将抵消由于增大 k 而减少外存访问次数所得到的效益。因此，不能使用普通的内部归并排序算法。

为了使内部归并不受 k 的增大的影响，引入了败者树。败者树是树形选择排序的一种变体，可视为一棵完全二叉树。 k 个叶结点分别存放 k 个归并段在归并过程中当前参加比较的记录，内部结点用来记忆左右子树中的“失败者”，而让胜者往上继续进行比较，一直到根结点。若比较两个数，大的为失败者、小的为胜利者，则根结点指向的数为最小数。

如图 8.15(a)所示，b3 与 b4 比较，b4 是败者，将段号 4 写入父结点 ls[4]。b1 与 b2 比较，

b2 是败者，将段号 2 写入 ls[3]。b3 与 b4 的胜者 b3 与 b0 比较，b0 是败者，将段号 0 写入 ls[2]。最后两个胜者 b3 与 b1 比较，b1 是败者，将段号 1 写入 ls[1]。而将胜者 b3 的段号 3 写入 ls[0]。此时，根结点 ls[0] 所指的段的关键字最小。b3 中的 6 输出后，将下一关键字填入 b3，继续比较。

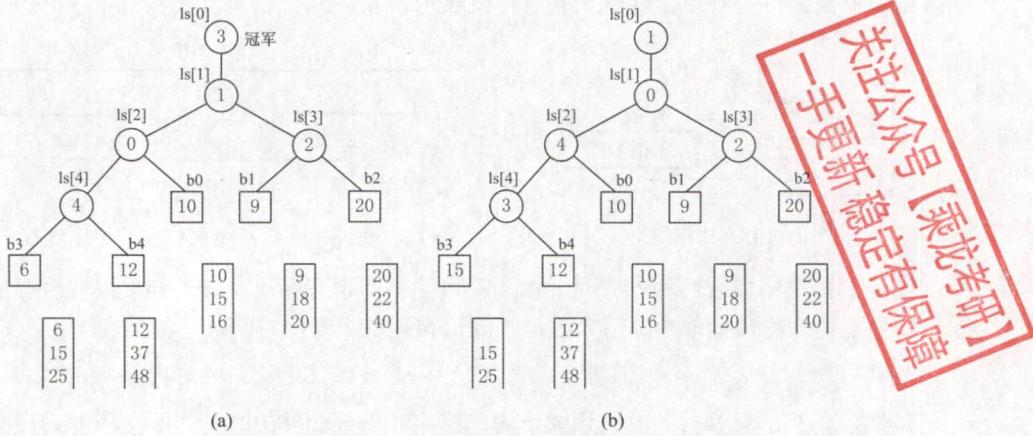


图 8.15 实现 5 路归并的败者树

因为 k 路归并的败者树深度为 $\lceil \log_2 k \rceil$ ，因此 k 个记录中选择最小关键字，最多需要 $\lceil \log_2 k \rceil$ 次比较。所以总的比较次数为

$$S(n-1)\lceil \log_2 k \rceil = \lceil \log_k r \rceil (n-1)\lceil \log_2 k \rceil = (n-1)\lceil \log_2 r \rceil$$

可见，使用败者树后，内部归并的比较次数与 k 无关了。因此，只要内存空间允许，增大归并路数 k 将有效地减少归并树的高度，从而减少 I/O 次数，提高外部排序的速度。

值得说明的是，归并路数 k 并不是越大越好。归并路数 k 增大时，相应地需要增加输入缓冲区的个数。若可供使用的内存空间不变，势必要减少每个输入缓冲区的容量，使得内存、外存交换数据的次数增大。当 k 值过大时，虽然归并趟数会减少，但读写外存的次数仍会增加。

8.7.4 置换-选择排序（生成初始归并段）

从 8.7.2 节的讨论可知，减少初始归并段个数 r 也可以减少归并趟数 S 。若总的记录个数为 n ，每个归并段的长度为 ℓ ，则归并段的个数 $r = \lceil n/\ell \rceil$ 。采用内部排序方法得到的各个初始归并段长度都相同（除最后一段外），它依赖于内部排序时可用内存工作区的大小。因此，必须探索新的方法，用来产生更长的初始归并段，这就是本节要介绍的置换-选择算法。

设初始待排文件为 FI，初始归并段输出文件为 FO，内存工作区为 WA，FO 和 WA 的初始状态为空，WA 可容纳 w 个记录。置换-选择算法的步骤如下：

- 1) 从 FI 输入 w 个记录到工作区 WA。
- 2) 从 WA 中选出其中关键字取最小值的记录，记为 MINIMAX 记录。
- 3) 将 MINIMAX 记录输出到 FO 中去。
- 4) 若 FI 不空，则从 FI 输入下一个记录到 WA 中。
- 5) 从 WA 中所有关键字比 MINIMAX 记录的关键字大的记录中选出最小关键字记录，作为新的 MINIMAX 记录。
- 6) 重复 3) ~5)，直至在 WA 中选不出新的 MINIMAX 记录为止，由此得到一个初始归并段。

段，输出一个归并段的结束标志到 FO 中去。

7) 重复 2) ~6)，直至 WA 为空。由此得到全部初始归并段。

设待排文件 $FI = \{17, 21, 05, 44, 10, 12, 56, 32, 29\}$ ，WA 容量为 3。排序过程如表 8.2 所示。

表 8.2 置换-选择排序过程示例

输出文件 FO	工作区 WA	输入文件 FI
—	—	17, 21, 05, 44, 10, 12, 56, 32, 29
—	17 21 05	44, 10, 12, 56, 32, 29
05	17 21 44	10, 12, 56, 32, 29
05 17	10 21 44	12, 56, 32, 29
05 17 21	10 12 44	56, 32, 29
05 17 21 44	10 12 56	32, 29
05 17 21 44 56	10 12 32	29
05 17 21 44 56 #	10 12 32	29
10	29 12 32	—
10 12	29 32	—
10 12 29	32	—
10 12 29 32	—	—
10 12 29 32 #	—	—

上述算法，在 WA 中选择 MINIMAX 记录的过程需利用败者树来实现。

8.7.5 最佳归并树

文件经过置换-选择排序后，得到的是长度不等的初始归并段。下面讨论如何组织长度不等的初始归并段的归并顺序，使得 I/O 次数最少？假设由置换-选择得到 9 个初始归并段，其长度（记录数）依次为 9, 30, 12, 18, 3, 17, 2, 6, 24。现做 3 路平衡归并，其归并树如图 8.16 所示。

在图 8.16 中，各叶结点表示一个初始归并段，上面的权值表示该归并段的长度，叶结点到根的路径长度表示其参加归并的趟数，各非叶结点代表归并成的新归并段，根结点表示最终生成的归并段。树的带权路径长度 WPL 为归并过程中的总读记录数，故 I/O 次数 = $2 \times WPL = 484$ 。

显然，归并方案不同，所得归并树亦不同，树的带权路径长度（ I/O 次数）亦不同。为了优化归并树的 WPL，可以将哈夫曼树的思想推广到 m 叉树的情形，在归并树中，让记录数少的初始归并段最先归并，记录数多的初始归并段最晚归并，就可以建立总的 I/O 次数最少的最佳归并树。上述 9 个初始归并段可构造成一棵如图 8.17 所示的归并树，按此树进行归并，仅需对外存进行 446 次读/写，这棵归并树便称为最佳归并树。

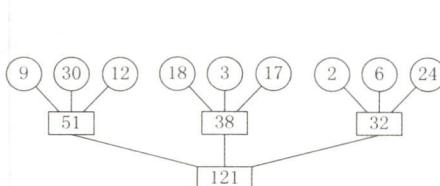


图 8.16 3 路平衡归并的归并树

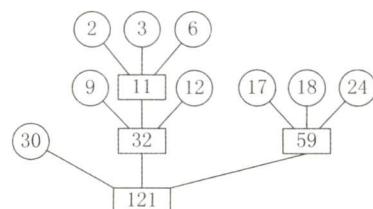


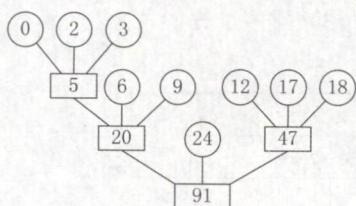
图 8.17 3 路平衡归并的最佳归并树

图 8.17 中的哈夫曼树是一棵严格 3 叉树，即树中只有度为 3 或 0 的结点。若只有 8 个初始归并段，如上例中少了一个长度为 30 的归并段。若在设计归并方案时，缺额的归并段留在最后，即除最后一次做 2 路归并外，其他各次归并仍是 3 路归并，此归并方案的外存读/写次数为 386。显然，这不是最佳方案。

正确的做法是：若初始归并段不足以构成一棵严格 k 叉树时，需添加长度为 0 的“虚段”，按照哈夫曼树的原则，权为 0 的叶子应离树根最远。因此，最佳归并树应如图 8.18 所示。

如何判定添加虚段的数目？

图 8.18 8 个归并段的最佳归并树



设度为 0 的结点有 $n_0 (=n)$ 个，度为 k 的结点有 n_k 个，则对严格 k 叉树有 $n_0 = (k-1)n_k + 1$ ，由此可得 $n_k = (n_0 - 1)/(k-1)$ 。

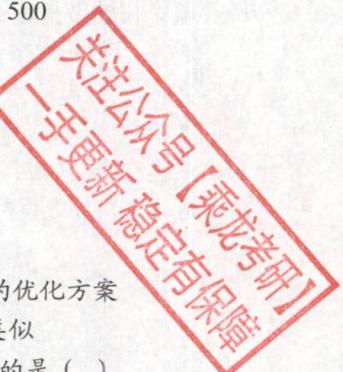
- 若 $(n_0 - 1)%(k - 1) = 0$ （% 为取余运算），则说明这 n_0 个叶结点（初始归并段）正好可以构造 k 叉归并树。此时，内结点有 n_k 个。
- 若 $(n_0 - 1)%(k - 1) = u \neq 0$ ，则说明对于这 n_0 个叶结点，其中有 u 个多余，不能包含在 k 叉归并树中。为构造包含所有 n_0 个初始归并段的 k 叉归并树，应在原有 n_k 个内结点的基础上再增加 1 个内结点。它在归并树中代替了一个叶结点的位置，被代替的叶结点加上刚才多出的 u 个叶结点，即再加上 $k-u-1$ 个空归并段，就可以建立归并树。

以图 8.18 为例，用 8 个归并段构成 3 叉树， $(n_0 - 1)%(k - 1) = (8 - 1)%(3 - 1) = 1$ ，说明 7 个归并段刚好可以构成一棵严格 3 叉树（假设把以 5 为根的树视为一个叶子）。为此，将叶子 5 变成一个内结点，再添加 $3-1-1=1$ 个空归并段，就可以构成一棵严格 3 叉树。

8.7.6 本节试题精选

一、单项选择题

01. 设在磁盘上存放有 375000 个记录，做 5 路平衡归并排序，内存工作区能容纳 600 个记录，为把所有记录排好序，需要做（ ）趟归并排序。
A. 3 B. 4 C. 5 D. 6
02. 设有 5 个初始归并段，每个归并段有 20 个记录，采用 5 路平衡归并排序，若不采用败者树，使用传统的顺序选出最小记录（简单选择排序）的方法，总的比较次数为（①）；若采用败者树最小的方法，总的比较次数约为（②）。
A. 20 B. 300 C. 396 D. 500
03. 置换-选择排序的作用是（ ）。
A. 用于生成外部排序的初始归并段
B. 完成将一个磁盘文件排序成有序文件的有效的外部排序算法
C. 生成的初始归并段的长度是内存工作区的 2 倍
D. 对外部排序中输入/归并/输出的并行处理
04. 最佳归并树在外部排序中的作用是（ ）。
A. 完成 m 路归并排序
B. 设计 m 路归并排序的优化方案
C. 产生初始归并段
D. 与锦标赛树的作用类似
05. 在下列关于外部排序过程输入/输出缓冲区作用的叙述中，不正确的是（ ）。
A. 暂存输入/输出记录
B. 内部归并的工作区
C. 产生初始归并段的工作区
D. 传送用户界面的消息
06. 在做 m 路平衡归并排序的过程中，为实现输入/内部归并/输出的并行处理，需要设置（①）



个输入缓冲区和(②)个输出缓冲区。

- | | | | |
|--------|--------|-------------|---------|
| ① A. 2 | B. m | C. $2m - 1$ | D. $2m$ |
| ② A. 2 | B. m | C. $2m - 1$ | D. $2m$ |

07. 【2013 统考真题】已知三叉树 T 中 6 个叶结点的权分别是 2, 3, 4, 5, 6, 7, T 的带权(外部)路径长度最小是()。

- | | | | |
|-------|-------|-------|-------|
| A. 27 | B. 46 | C. 54 | D. 56 |
|-------|-------|-------|-------|

08. 【2019 统考真题】设外存上有 120 个初始归并段, 进行 12 路归并时, 为实现最佳归并, 需要补充的虚段个数是()。

- | | | | |
|------|------|------|------|
| A. 1 | B. 2 | C. 3 | D. 4 |
|------|------|------|------|

二、综合应用题

01. 多路平衡归并排序是外部排序的主要方法, 试问多路平衡归并排序包括哪两个相对独立的阶段? 每个阶段完成何种工作?

02. 若某个文件经内部排序得到 80 个初始归并段, 试问:

- 1) 若使用多路平衡归并执行 3 趟完成排序, 则应取得的归并路数至少应为多少?
- 2) 若操作系统要求一个程序同时可用的输入/输出文件的总数不超过 15 个, 则按多路归并至少需要几趟可以完成排序? 若限定这个趟数, 可取的最低路数是多少?

03. 假设文件有 4500 个记录, 在磁盘上每个块可放 75 个记录。计算机中用于排序的内存区可容纳 450 个记录。试问:

- 1) 可以建立多少个初始归并段? 每个初始归并段有多少记录? 存放于多少个块中?
- 2) 应采用几路归并? 请写出归并过程及每趟需要读写磁盘的块数。

04. 设初始归并段为(10, 15, 31), (9, 20), (22, 34, 37), (6, 15, 42), (12, 37), (84, 95)。试利用败者树进行 m 路归并, 手工执行选择最小的 5 个关键字的过程。

05. 给出 12 个初始归并段, 其长度分别为 30, 44, 8, 6, 3, 20, 60, 18, 9, 62, 68, 85。现要做 4 路外归并排序, 试画出表示归并过程的最佳归并树, 并计算该归并树的带权路径长度 WPL。

8.7.7 答案与解析

一、单项选择题

01. B

关注公众号【乘龙考研】
一手更新 稳定有保障

初始归并段的个数 $r = 375000/600 = 625$, 因此, 归并趟数 $S = \lceil \log_m r \rceil = \lceil \log_5 625 \rceil = 4$ 。第一趟把 625 个归并段归并成 $625/5 = 125$ 个; 第二趟把 125 个归并段归并成 $125/5 = 25$ 个; 第三趟把 25 个归并段归并成 $25/5 = 5$ 个; 第四趟把 5 个归并段归并成 $5/5 = 1$ 个。

02. C、B

- ① 不采用败者树时, 在 5 个记录中选出最小的需要做 4 次比较, 共有 100 个记录, 需要做 99 次选择最小记录的操作, 所以需要的比较次数为 $4 \times 99 = 396$, 故选择选项 C。
- ② 采用败者树时, 5 路归并意味着败者树的外结点有 5 个, 败者树的高度 $h = \lceil \log_2 5 \rceil = 3$ 。每次在参加比较的记录中选择一个关键字最小的记录, 比较次数不超过 h , 共 100 个记录, 需要的比较次数不超过 $100 \times 3 = 300$, 故选择选项 B。

03. A

置换-选择排序是外部排序中生成初始归并段的方法, 用此方法得到的初始归并段的长度是不等长的, 其长度平均是传统等长初始归并段的 2 倍, 从而使得初始归并段数减少到原来的近二分之一。但是, 置换-选择排序不是一种完整的生成有序文件的外部排序算法。

04. B

最佳归并树在外部排序中的作用是设计 m 路归并排序的优化方案，仿照构造哈夫曼树的方法，以初始归并段的长度为权值，构造具有最小带权路径长度的 m 叉哈夫曼树，可以有效地减少归并过程中的读写记录数，加快外部排序的速度。

05. D

在外部排序过程中输入/输出缓冲区就是排序的内存工作区，例如做 m 路平衡归并需要 m 个输入缓冲区和 1 个输出缓冲区，用以存放参加归并的和归并完成的记录。在产生初始归并段时也可用作内部排序的工作区。它没有传送用户界面的消息的任务。

06. D、A

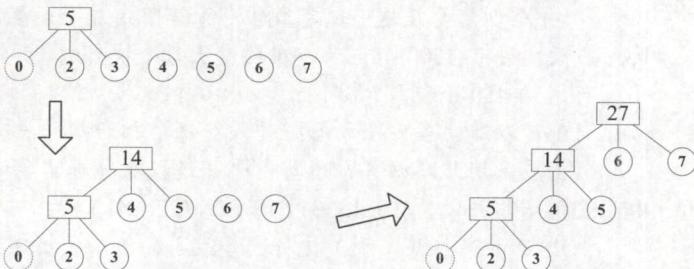
在做 m 路平衡归并排序的过程中，为实现输入/内部归并/输出的并行处理，需要设置 $2m$ 个输入缓冲区和 2 个输出缓冲区，以便在执行内部归并时，能同时进行输入/输出操作。若仅设置 m 个输入缓冲区，则仅能进行串行操作，无法并行处理。注重理解 2 路归并排序并在解题中留心题眼是正确解答的关键。

07. B

将哈夫曼树的思想推广到三叉树的情形。为了构成严格的三叉树，需添加权为 0 的虚叶结点，对于严格的三叉树， $(n_0 - 1) \% (3 - 1) =$

$u = 1 \neq 0$ ，需要添加 $m - u - 1 = 3 - 1 - 1 = 1$ 个叶结点，说明 7 个叶结点刚好可以构成一棵严格的三叉树。按照哈夫曼树的原则，权为 0 的叶结点应离树根最远，构造最小带权生成树的过程如右图所示。

最小的带权路径长度为 $(2 + 3) \times 3 + (4 + 5) \times 2 + (6 + 7) \times 1 = 46$ 。

**08. B**

在 12 路归并树中只存在度为 0 和度为 12 的结点，设度为 0 的结点数、度为 12 的结点数和要补充的结点数分别为 n_0 , n_{12} 和 $n_{\text{补}}$ ，则有 $n_0 = 120 + n_{\text{补}}$ ， $n_0 = (12 - 1)n_{12} + 1$ ，可得 $n_{12} = (120 - 1 + n_{\text{补}}) / (12 - 1)$ 。

由于结点数 n_{12} 为整数，所以 $n_{\text{补}}$ 是使上式整除的最小整数，求得 $n_{\text{补}} = 2$ ，故答案选择选项 B。

二、综合应用题

01. 【解答】

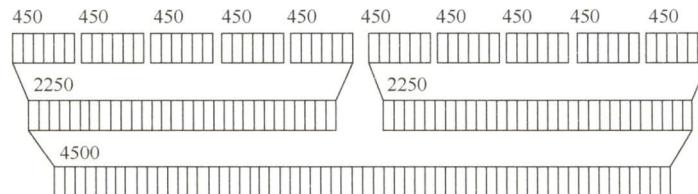
多路平衡归并排序由两个相对独立的阶段组成：生成初始归并段阶段和多趟归并排序阶段。生成初始归并段阶段根据内存工作区的大小，将有 n 个记录的磁盘文件分批输入内存，采用有效的内部排序方法分别进行排序，生成若干有序的子文件，即初始归并段。多趟归并排序阶段采用多路归并方法将这些归并段逐趟归并，最后归并成一个有序文件。

02. 【解答】

- 1) 设归并路数为 m ，初始归并段个数 $r = 80$ ，根据归并趟数计算公式 $S = \lceil \log_m r \rceil = \lceil \log_m 80 \rceil = 3$ ，得 $\log_m 80 \leq 3$ ， $m^3 \geq 80$ 。由此解得 $m \geq 5$ ，即应取的归并路数至少为 5。
- 2) 设多路归并的归并路数为 m ，需要 m 个输入缓冲区和 1 个输出缓冲区。一个缓冲区对应一个文件，有 $m + 1 = 15$ ，因此 $m = 14$ ，可做 14 路归并。由 $S = \lceil \log_m r \rceil = \lceil \log_{14} 80 \rceil = 2$ ，即至少需要 2 趟归并可完成排序。若限定趟数为 2，由 $S = \lceil \log_m 80 \rceil = 2$ ，有 $80 \leq m^2$ ，可取的最低路数为 9。即要在 2 趟内完成排序，进行 9 路归并排序即可。

03. 【解答】

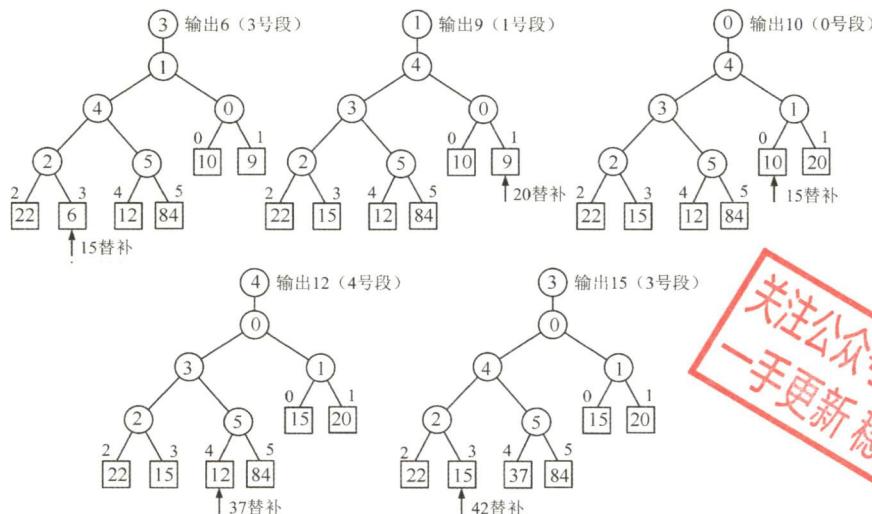
- 1) 文件有 4500 个记录, 用于排序的内存区可容纳 450 个记录, 可建立的初始归并段有 $4500/450 = 10$ 个。每个初始归并段中有 450 个记录, 存于 $450/75 = 6$ 个块中。
- 2) 内存区可容纳 6 个块, 可建立 6 个缓冲区, 其中 5 个缓冲区用于输入, 1 个缓冲区用于输出, 因此可采用 5 路归并, 归并过程如下图所示。



共做了 2 趟归并, 每趟需要读 60 块、写 60 块。

04. 【解答】

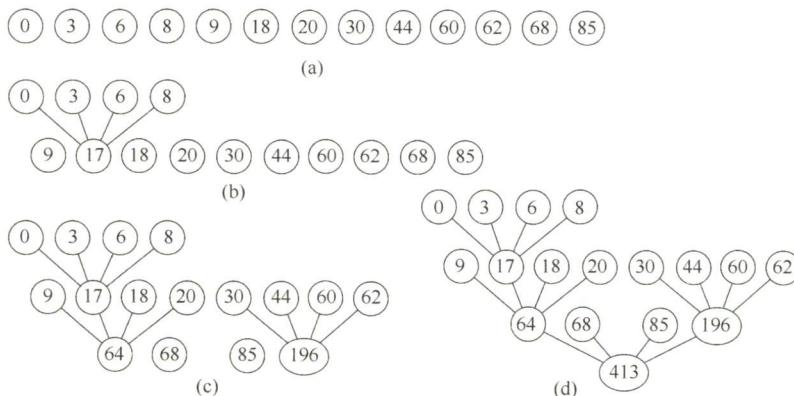
做 6 路归并排序, 选择最小的 5 个关键字的败者树如下图所示。



05. 【解答】

设初始归并段个数 $n = 12$, 外归并路数 $k = 4$, 计算 $(n-1)(k-1) = 11 \times 3 = 2 \neq 0$, 说明不能做完全的 4 路归并, 因为多出了 2 个初始归并段, 必须添加 $k-2-1=1$ 个长度为 0 的空归并段, 才能构成严格的 4 路归并树, 即每次归并都有 k 个归并段参加归并。

此时, 归并树的内结点应有 $(n-1+1)/(k-1) = 12/3 = 4$ 个, 如下图所示。



$$WPL = (3 + 6 + 8) \times 3 + (9 + 18 + 20 + 30 + 44 + 60 + 62) \times 2 + (68 + 85) \times 1 = 51 + 486 + 153 = 690.$$

归纳总结

下面对本章所介绍的排序算法进行一次系统的比较和复习。

1. 直接插入排序、冒泡排序和简单选择排序是基本的排序方法，它们主要用于元素个数 n 不是很大 ($n < 10000$) 的情形。

它们的平均时间复杂度均为 $O(n^2)$ ，实现也都非常简单。直接插入排序对于规模很小的元素序列 ($n \leq 25$) 非常有效。它的时间复杂度与待排序元素序列的初始排列有关。在最好情况下，直接插入排序只需要 $n - 1$ 次比较操作就可以完成，且不需要交换操作。在平均情况下和最差情况下，直接插入排序的比较和交换操作都是 $O(n^2)$ 。冒泡排序在最好情况下只需要一趟排序过程就可以完成，此时也需要 $n - 1$ 次比较操作，不需要交换操作。简单选择排序的关键字比较次数与待排序元素序列的初始排列无关，其比较次数总是 $O(n^2)$ ，但元素移动次数则与待排序元素序列的初始排列有关，最好情况下数据不需要移动，最坏情况下元素移动次数不超过 $3(n - 1)$ 。

从空间复杂度来看，这三种基本的排序方法除一个辅助元素外，都不需要其他额外空间。从稳定性来看，直接插入排序和冒泡排序都是稳定的，但简单选择排序不是。

2. 对于中等规模的元素序列 ($n \leq 1000$)，希尔排序是一种很好的选择。

在希尔排序中，开始时增量较大，分量较多，每个组内的记录数较少，因而记录的比较和移动次数较少，且移动距离较远；到后来步长越来越小（最后一步为 1），分组越少，每个组内的记录数越多，但同时记录次序也越来越接近有序，因而记录的比较和移动次数也都比较少。从理论上和实验上都已证明，在希尔排序中，记录的总比较次数和总移动次数比直接插入排序时少得多，特别是当 n 越大时效果越明显。而且，希尔排序代码简单，基本上不需要什么额外内存，但希尔排序是一种不稳定的排序算法。

3. 对于元素个数 n 很大的情况，可以采用快排、堆排序、归并排序或基数排序，其中快排和堆排序都是不稳定的，而归并排序和基数排序是稳定的排序算法。

快速排序是最通用的高效内部排序算法，特别是它的划分思想经常在很多算法设计题中出现。平均情况下它的时间复杂度为 $O(n \log_2 n)$ ，一般情况下所需要的额外空间也是 $O(\log_2 n)$ 。但是快速排序在有些情况下也可能退化（如元素序列已经有序时），时间复杂度会增加到 $O(n^2)$ ，空间复杂度也会增加到 $O(n)$ 。但我们可以“三者取中”法来避免最坏情况的发生。

堆排序也是一种高效的内部排序算法，它的时间复杂度是 $O(n \log_2 n)$ ，而且没有什么最坏情况会导致堆排序的运行明显变慢，并且堆排序基本上不需要额外的空间。但堆排序不大可能提供比快速排序更好的平均性能。

归并排序也是一个重要的高效排序算法，它的一个重要特性是性能与输入元素序列无关，时间复杂度总是 $O(n \log_2 n)$ 。归并排序的主要缺点是需要 $O(n)$ 的额外存储空间。

基数排序是一种相对特殊的排序算法，这类算法不仅是对元素序列的关键字进行比较，更重要的是它们对关键字的不同位部分进行处理和比较。虽然基数排序具有线性增长的时间复杂度，但由于在常规编程环境中，基数排序的线性时间开销实际上并不比快速排序的时间开销小很多，并且由于基数排序基于的关键字抽取算法受到操作系统和排序元素的影响，其适应性远不如普通的进行比较和交换操作的排序方法。因此，在实际工作中，常规的高效排序算法如快速排序的应

用要比基数排序广泛得多。基数排序需要的额外存储空间包括和待排序元素序列规模相同的存储空间及与基数数目相等的一系列桶（一般用队列实现）。

4. 混合使用。

我们可以混合使用不同的排序算法，这也是得到普遍应用的一种算法改进方法，例如，可以将直接插入排序集成到归并排序的算法中。这种混合算法能够充分发挥不同算法各自的优势，从而在整体上得到更好的性能。

思维拓展

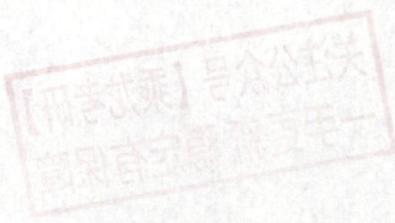
下面是一道看起来很吓人的题目：对 n 个整数进行排序，要求时间复杂度为 $O(n)$ ，空间复杂度为 $O(1)$ 。

（提示：假设待排序整数的范围为 0~65535，设定一个数组 int count[65535] 并初始化为 0，则所需空间与 n 无关，为 $O(1)$ 。扫描一遍待排序列 X[]，count[X[i]]++，时间复杂度为 $O(n)$ ；再扫描一次 count[]，当 count[i] > 0 时，输出 count[i] 个 i，排序完毕所需的时间复杂度也为 $O(n)$ ；故总的时间复杂度为 $O(n)$ ，空间复杂度为 $O(1)$ 。另外，读者可能会问假如有负整数怎么办，这种情况下可以给所有整数都加上一个偏移量，使之都变成正整数，再使用上述方法即可。）



参 考 文 献

- [1] 严蔚敏, 吴伟民. 数据结构 (C 语言版) [M]. 北京: 清华大学出版社, 2009.
- [2] Thomas H. Cormen, 等. 算法导论[M]. 北京: 机械工业出版社, 2013.
- [3] 李春葆, 尹为民, 等. 数据结构教程[M]. 北京: 清华大学出版社, 2009.
- [4] 陈守孔, 胡潇琨, 李玲. 算法与数据结构考研试题精析[M]. 北京: 机械工业出版社, 2007.
- [5] 夏清国. 数据结构考研教案[M]. 西安: 西北工业大学出版社, 2006.
- [6] 本书编写组. 计算机专业基础综合考试大纲解析[M]. 北京: 高等教育出版社, 2009.
- [7] 李春葆等. 数据结构联考辅导教程[M]. 北京: 清华大学出版社, 2010.



经久才是王道

十五年考研磨练

2024年王道计算机考研课程

- ◆ 阶段一：计算机考研零基础入门
- ◆ 阶段二：大纲考点、经典习题精讲
- ◆ 阶段三：暑期强化提升训练
- ◆ 阶段四：真题、模拟题、考前冲刺
- ◆ 阶段五：复试机试课程



加客服领券购课

抄/底/价/299元起

- 2024年数据结构考研复习指导
- 2024年计算机组成原理考研复习指导
- 2024年操作系统考研复习指导
- 2024年计算机网络考研复习指导
- 2024年计算机专业基础综合考试历年真题解析
- 2024年计算机专业基础综合考试冲刺模拟题
- 计算机考研——机试指南（第2版）



责任编辑：谭海平
封面设计：张 昱

ISBN 978-7-121-44471-5



9 787121 444715

定价：75.00 元