第1章　引论

1.3 由于浮点数的误差，通常我们将应输出中包含的小数部分位数特别指明，并以此进行向上舍入。否则将会输出的数会令人摸不到头脑。假设错误检查已经写好，分离整数与小数部分的例程留给读者完成。代码如图 1.1。

```
double RoundUp( double N, int DecPlaces )
{
        int i;
        double AmountToAdd = 0.5;

        for( i = 0; i < DecPlaces; i++ )
                AmountToAdd /= 10;
        return N + AmountToAdd;
}

void PrintFractionPart( double FractionPart, int DecPlaces )
{
        int i, Adigit;

        for( i = 0; i < DecPlaces; i++ )
        {
                FractionPart *= 10;
                ADigit = IntPart( FractionPart );
                PrintDigit( Adigit );
                FractionPart = DecPart( FractionPart );
        }
}

void PrintReal( double N, int DecPlaces )
{
        int IntegerPart;
        double FractionPart;

        if( N < 0 )
        {
                putchar('-');
                N = -N;
        }
        N = RoundUp( N, DecPlaces );
        IntegerPart = IntPart( N ); FractionPart = DecPart( N );
        PrintOut( IntegerPart );        /* Using routine in text */
        if( DecPlaces > 0 )
                putchar('.');
        PrintFractionPart( FractionPart, DecPlaces );
}
```

Fig. 1.1.

1.4 通常的做法是在程序头部声明如下函数

void ProcessFile(const char *Filename);

它的作用是打开文件，进行一切需要的操作，最后关闭文件。如果程序中有类似于

#include SomeFile

这样的代码，那么便递归的调用

ProcessFile(SomeFile);

通过维护一个由所有处于被 ProcessFile 函数打开状态的文件构成的列表，并在每一次调用 ProcessFile 时检查列表中是否有相同的文件，便可以检查出包含自己的链接。

1.5 (a)证明由数学归纳法给出。当 0<X<=1 时，logX<=0，命题显然成立。同样，容易发现 1<X<=2 时 logX<=1，命题也成立。假设对 p<X<=2p（p 为正整数）命题成立，那么，对任意 2p<Y<=4p（p>=1），有 logY = 1 + log(Y/2) < 1 + Y/2 < Y/2 + Y/2 = Y（其中第一个不等号是由归纳假设得出的），由数学归纳法得对任意 X>0 命题均成立。

(b)设 $A = 2^X$ ，则 $A^B = (2^X)^B = 2^{XB}$ ，即 $\log A^B = XB$ ，因为 $X = \log A$ ，所以命题成立

1.6 (a)直接由公式可得和为 4/3

(b) $S = \frac{1}{4} + \frac{2}{4^2} + \frac{3}{4^3} + \ldots$ ，$4S = 1 + \frac{2}{4} + \frac{3}{4^2} + \ldots$ ，用第一个式子去减第二个式子，得到

$3S = 1 + \frac{1}{4} + \frac{2}{4^2} + \ldots$ ，由(a)得 3S=4/3，即 S=4/9

(c) $S = \frac{1}{4} + \frac{4}{4^2} + \frac{9}{4^3} + \ldots$ ，$4S = 1 + \frac{4}{4} + \frac{9}{4^2} + \frac{16}{4^3} + \ldots$ 。用第一个式子去减第二个式子，得到 $3S = 1 + \frac{3}{4} + \frac{5}{4^2} + \frac{7}{4^3} + \ldots$ ，这个式子还可以写成： $3S = 2\sum_{i=0}^{\infty} \frac{i}{4^i} + \sum_{i=0}^{\infty} \frac{1}{4^i}$ ，因而 3S=2*(4/9)+4/3=20/9，即 S=20/27。

(d)令 $S_N = \sum_{i=0}^{\infty} \frac{i^N}{4^i}$ ，进行如同(a)-(c)的步骤，我们可以得到用 $S_{N-1}, S_{N-2}, \ldots, S_0$ 表示的 S_N 的公式。然而由这一递推式得出通项公式是非常困难的。

1.7 $\sum_{i=\lfloor N/2 \rfloor}^{N} \frac{1}{i} = \sum_{i=1}^{N} \frac{1}{i} - \sum_{i=1}^{\lfloor N/2-1 \rfloor} \frac{1}{i} \approx \ln N - \ln \frac{N}{2} = \ln 2$

1.8 $2^4 = 16 \equiv 1 \pmod 5$ ，$(2^4)^{25} \equiv 1^{25} \pmod 5$ ，所以 $2^{100} \equiv 1 \pmod 5$

1.9 (a)（译者注：书中此题要证明 $\sum_{i=1}^{N-2} F_i = F_N - 2$，而答案中证明的式子似乎是 $\sum_{i=1}^{N} F_i = F_{N+2} - 2$，不过只有形式上的区别）此题通过数学归纳法证明。N=1,2 时，命题显然成立，设命题对 N=1,2,3,4,…k 均成立（k 为正整数），因 $\sum_{i=1}^{k+1} F_i = \sum_{i=1}^{k} F_i + F_{k+1}$，由归纳假设得等号右边的值为 $F_{k+2} - 2 + F_{k+1} = F_{k+3} - 2$，所以对 N=k+1 命题也成立。因此对所有正整数 N 命题均成立。

(b)此题同样通过数学归纳法证明。注意到 $\phi + 1 = \phi^2$，即 $\phi^{-1} + \phi^{-2} = 1$。对 N=1 或 2，命题成立。假设 N=1,2,..,k 时命题均成立，由斐波那契数列的递推式

$$F_{k+1} = F_k + F_{k-1}$$

和归纳假设，可得

$$F_{k+1} < \phi^k + \phi^{k-1} = (\phi^{-1} + \phi^{-2})\phi^{k+1} = \phi^{k+1}$$

由此命题得证。

(c)可参阅章末的任意一本数学参考文献。答案涉及到对生成函数的应用。

1.10 (a) $\sum_{i=1}^{N} (2i-1) = 2\sum_{i=1}^{N} i - \sum_{i=1}^{N} 1 = N(N+1) - N = N^2$

(b)最简单的方法是通过数学归纳法证明。N=1 的情况显然成立。其他情况，

$$\begin{aligned}
\sum_{i=1}^{N+1} i^3 &= (N+1)^3 + \sum_{i=1}^{N} i^3 \\
&= (N+1)^3 + \frac{N^2(N+1)^2}{4} \\
&= (N+1)^2[\frac{N^2}{4} + (N+1)] \\
&= (N+1)^2[\frac{N^2+4N+4}{4}] \\
&= \frac{(N+1)^2(N+2)^2}{2^2} \\
&= [\frac{(N+1)(N+2)}{2}]^2 \\
&= (\sum_{i=1}^{N+1} i)^2
\end{aligned}$$

第2章　算法分析

2.1　2/N，37，$\sqrt{N}$ ，N，NloglogN，NlogN，$N\log(N^2)$ ，$N^{1.5}$ ，N^2 ，$N^2\log N$ ，N^3 ，$2^{N/2}$ ，2^N 。NlogN 与 $N\log(N^2)$ 增长率相同

2.2　(a)成立

(b)不成立，一个反例为 $T_1(N)=2N, T_2(N)=N$ ，而 $f(N)=N$

(c)不成立，一个反例为 $T_1(N)=N^2, T_2(N)=N$ ，$f(N)=N^2$

(d)不成立，反例如(c)

2.3　NlogN 增长更慢，为证明这一点，我们使用反证法：假设 $N^{\varepsilon/\sqrt{\log N}}$ 比 logN 增长更慢，两边同取对数，得到 $(\varepsilon/\sqrt{\log N})*\log N$ 比 loglogN 增长更慢，第一个表达式实际上是 $\varepsilon\sqrt{\log N}$ ，设 L=logN，则有 $\varepsilon\sqrt{L}$ 比 logL 增长得更慢，即 $\varepsilon^2 L$ 比 $\log^2 L$ 增长的更慢，但实际上我们知道 $\log^2 L=o(L)$ ，所以我们一开始的假设是错的，因此命题得证。

2.4　显然若 k1<k2，则 $\log^{k1} N=o(\log^{k2} N)$ ，所以我们只需考虑 k 为正数的情况。对 k=0 或 k=1，命题显然成立。假设对 k<1 命题均成立，由 L'Hospital 法则，

$$\lim_{N\to\infty}\frac{\log^i N}{N}=\lim_{N\to\infty} i\frac{\log^{i-1} N}{N}$$

由归纳假设，等号右边的极限为 0，所以命题得证。（译者注：本书中 log 默认为以 2 为底的对数，因此 $\log^i N$ 求导应为 $\frac{i\log^{i-1} N}{N\ln 2}$ 而不是 $\frac{i\log^{i-1} N}{N}$ ，应是答案的错误）

2.5　定义当 N 为偶数时 f(N)为 1，N 为奇数时 f(N)为 N，同样，定义当 N 为奇数时 g(N)为 1，N 为偶数时 g(N)为 N，f(N)与 g(N)的比值在 0 到∞间振荡。

2.6　对下面的每个程序，做出的时间分析都与模拟结果一致

(1)　运行时间为 O(N)

(2) 运行时间为$O(N^2)$

(3) 运行时间为$O(N^3)$

(4) 运行时间为$O(N^2)$

(5) j 最大可达到i^2，i^2 最大可达到N^2，k 最大可达到 j，即N^2，因此可能的最大的运行时间为$N \times N^2 \times N^2$，即$O(N^5)$

(6) 由前面的分析，if 语句被执行$O(N^3)$次，但实际上因为 i*i 个 j 的取值中只有 i 个使 if 语句为真，所以只有$O(N^2)$次进入 if 语句内层，而每次执行内层循环的用时为$O(j) = O(N^2)$，所以总共耗时为$O(N^4)$。这个例子说明简单的把各层循环的大小相乘有时会给出过高的估计结果。

2.7 (a)显然三种算法都生成合法的置换，前两种算法有确保不会发生数字重复的检测，第三种算法是打乱一个原本就没有重复的数组，所以三种都不会生成不合法的置换。前两种算法是完全随机的，每种置换都是等可能的。而这一点对第三种算法（它由 R.Floyd 提出）不是很显然，这一算法的正确性可以通过数学归纳法证明，读者可参阅 Jon Bentley 的《Programming Pearls》（《编程珠玑》）

需要指出的是若算法 3 的第二行改为

Swap(A[i], A[RandInt(0, N-1)]);

则各置换并不随机出现。为了说明这一点，注意到当 N=3 时，共有 27 种等可能的交换方式，而合法的置换有 6 种，无法平分 27，因此若算法是如此，每种置换不会等可能出现。

(b)第一种算法，判断在 A[i]位置随机生成的数是否与前面的重复需要耗费 O(i)时间，产生随机数个数的数学期望为 N/(N-i)，这个值由如下的步骤得出：N 个数中有 i 个是与前面重复的，所以产生一个不重复的数字的概率是(N-i)/N，且每次的结果互相独立，因此平均需要 N/(N-i)次尝试才能产生不重复的数字，所以时间界为

$$\sum_{i=0}^{N-1} \frac{Ni}{N-i} < \sum_{i=0}^{N-1} \frac{N^2}{N-i} = N^2 \sum_{i=0}^{N-1} \frac{1}{N-i} = N^2 \sum_{j=1}^{N-1} \frac{1}{j} = O(N^2 \log N)$$

第二种算法节省了可能出现的 i 次测试，所以时间界降为 O(NlogN)，第三种算法很明显是线性的。

(c,d)如果有足够的空间，运行结果会和上述的分析一致，如果没有足够的空间，第三个算法当 N 很大时会有激烈的增长，使它看起来不是线性的。

(e)无法为第一、二种算法的最坏运行时间找到一个界，因为对任意给定的时间 T，算法结束的概率都不为 0（虽然如此，算法是一定会结束的）。第三种算法的最坏运行时间仍是线性的，它的运行时间不依赖于产生的随机数序列。

2.8 算法 1：N=10,000 时耗时约 5 天，N=100,000 时耗时 14.2 年，N=1,000,000 时耗时 140

个世纪
算法 2：N=100,000 时耗时 3 小时，N=1,000,000 时耗时 2 星期
算法 3：N=1,000,000 时耗时 1.5 分钟
这些计算都建立在电脑具有足够空间储存数组的前提下，算法 4 在 N=1,000,000 时只耗时 3 秒

2.9 (a) $O(N^2)$

(b) $O(N \log N)$

2.10 (c)线性时间

2.11 使用二分查找的变种可以得到 O(logN)的运行时间（假设数组已经预先读入）

2.13 (a)首先判断 N 是否为奇数或 2，然后判断是否能被 3,5,7,…, $\sqrt{N}$ 整除

(b)假设每次除法均耗费一个单位时间，$O(\sqrt{N})$

(c)B=O(logN)

(d) $O(2^{B/2})$

(e)如果 20 位的数需要时间为 T，则 40 位的数需要时间为 T^2
(f)B 更合理，因为它更好的表达了输入的数据占据的空间大小

2.14 该算法的运行时间为 N 乘以所有小于 N 的素数的倒数之和，为 $O(N \log\log N)$ ，详见 Knuth《计算机程序设计艺术》第二卷

2.15 依次计算出 $X^2, X^4, X^8, X^{10}, X^{20}, X^{40}, X^{60}$ ，最终算出 X^{62}

2.16 维护一个数组 PowersOfX，它包含 $X, X^2, X^4, ..., X^{2^{\lfloor \log N \rfloor}}$ 。N 的二进制表示（可通过不断判断奇偶性并除以 2 得到）可用来选取数组中合适的项相乘得到结果。

2.17 对 N=0 或 N=1，乘法次数为 0。对 N>1，若 b(N)为 N 的二进制表示中数字 1 的个数，则乘法次数为

$$\lfloor \log N \rfloor + b(N) - 1$$

2.18 (a) A
(b)B
(c)题中给出的信息不足以得出答案，因为我们只有最坏时间界
(d)否（答案原为“Yes”，应该是答案有误）

2.19 (a)当数组 B 中只有两个或更少的元素时递归便不必要了

(b)一种方法是，注意到如果前 N-1 个元素中有主要元素，则最后一个元素对结果没有影响，否则最后一个元素可能是候选元，将它作为候选元。（译者疑问：可否比较 A_{N-2} 和 A_N 及 A_{N-1} 和 A_N 是否相等，若有一组相等则加入 B？）

(c)设运行时间为 T(N)，有递推式 T(N)=T(N/2)+O(N)，解得 T(N)=O(N)（参见第十章 10.2.1）

(d)保存一个数组 A 的拷贝后，可将要添加到数组 B 的元素直接覆盖在数组 A 上，每一级递归均可这样做。最初的递归策略需要 O(logN)个数组，而这样只需要两个。

2.20 如果不这样，我们可以巧妙地对数字编码，同时进行多个操作。例如，若 A=001，B=101，C=111，D=100，如果我们要计算 A+B 和 C+D，只需要计算 00A00C+00B00D 即可。我们可以把这个方法推广到任意 N 对数字，在一个单位时间内计算出每对的和。

2.22 不能，若 Low=1，High=2，则 Mid=1，递归永不会结束。

2.24 不能，和题 2.22 一样，递归可能永不会结束。

第3章　表、栈和队列

3.2 题 3.4 中关于抽象性程度的建议在本题中被应用。程序如图 3.1 所示，运行时间为 O(L+P)。

```
void
PrintLots( List L, List P )
{
        int Counter;
        Position Lpos, Ppos;

        Lpos = First( L );
        Ppos = First( P );
        Counter = 1;
        while( Lpos != NULL && Ppos != NULL )
        {
                if( Ppos->Element == Counter++ )
                {
                        printf( "%? ", Lpos->Element );
                        Ppos = Next( Ppos, P );
                }
                Lpos = Next( Lpos, L );
        }
}
```

Fig. 3.1.

3.3　(a)对单链表，代码如图 3.2 所示

```
/* BeforeP is the cell before the two adjacent cells that are to be swapped. */
/* Error checks are omitted for clarity. */

void
SwapWithNext( Position BeforeP, List L )
{
        Position P, AfterP;

        P = BeforeP->Next;
        AfterP = P->Next;     /* Both P and AfterP assumed not NULL. */

        P->Next = AfterP->Next;
        BeforeP->Next = AfterP;
        AfterP->Next = P;
}
```

Fig. 3.2.

（图中的注释为：BeforeP 为要交换的两个相邻元素之前的元素，为了代码的清晰错误检查已被省略）

(b)对双链表，代码如图 3.3 所示

```
/* P and AfterP are cells to be switched.  Error checks as before. */

void
SwapWithNext( Position P, List L )
{
        Position BeforeP, AfterP;

        BeforeP = P->Prev;
        AfterP  = P->Next;

        P->Next = AfterP->Next;
        BeforeP->Next = AfterP;
        AfterP->Next = P;
        P->Next->Prev = P;
        P->Prev = AfterP;
        AfterP->Prev = BeforeP;
}
```

Fig. 3.3.

（图中注释为：P 和 AfterP 是要被交换的两个元素，错误处理同样已省略）

3.4　求交集的函数 Intersect 如下所示：

```
/* This code can be made more abstract by using operations such as              */
/* Retrieve and IsPastEnd to replace L1Pos->Element and L1Pos != NULL.          */
/* We have avoided this because these operations were not rigorously defined.   */

List
Intersect( List L1, List L2 )
{
        List Result;
        Position L1Pos, L2Pos, ResultPos;

        L1Pos = First( L1 ); L2Pos = First( L2 );
        Result = MakeEmpty( NULL );
        ResultPos = First( Result );
        while( L1Pos != NULL && L2Pos != NULL )
        {
                if( L1Pos->Element < L2Pos->Element )
                        L1Pos = Next( L1Pos, L1 );
                else if( L1Pos->Element > L2Pos->Element )
                        L2Pos = Next( L2Pos, L2 );
                else
                {
                        Insert( L1Pos->Element, Result, ResultPos );
                        L1 = Next( L1Pos, L1 ); L2 = Next( L2Pos, L2 );
                        ResultPos = Next( ResultPos, Result );
                }
        }
        return Result;
}
```

（图中注释为：这个代码可以通过用 Retrieve 和 IsPastEnd 替代 L1Pos->Element 和 L1Pos!=NULL 的方法拥有更大的抽象性。我们不这样做是因为这些操作没有被严密定义）

3.5　求并集的函数 Union 如图 3.4 所示：

```
List
Union( List L1, List L2 )
{
        List Result;
        ElementType InsertElement;
        Position L1Pos, L2Pos, ResultPos;

        L1Pos = First( L1 ); L2Pos = First( L2 );
        Result = MakeEmpty( NULL );
        ResultPos = First( Result );
        while ( L1Pos != NULL && L2Pos != NULL ) {
                if( L1Pos->Element < L2Pos->Element ) {
                        InsertElement = L1Pos->Element;
                        L1Pos = Next( L1Pos, L1 );
                }
                else if( L1Pos->Element > L2Pos->Element ) {
                        InsertElement = L2Pos->Element;
                        L2Pos = Next( L2Pos, L2 );
                }
                else {
                        InsertElement = L1Pos->Element;
                        L1Pos = Next( L1Pos, L1 ); L2Pos = Next( L2Pos, L2 );
                }
                Insert( InsertElement, Result, ResultPos );
                ResultPos = Next( ResultPos, Result );
        }
        /* Flush out remaining list */
        while( L1Pos != NULL ) {
                Insert( L1Pos->Element, Result, ResultPos );
                L1Pos = Next( L1Pos, L1 ); ResultPos = Next( ResultPos, Result );
        }
        while( L2Pos != NULL ) {
                Insert( L2Pos->Element, Result, ResultPos );
                L2Pos = Next( L2Pos, L2 ); ResultPos = Next( ResultPos, Result );
        }
        return Result;
}
```

Fig. 3.4.

3.7 (a)一种算法是在计算的过程中始终保持结果按幂排列。MN 个乘积中每一个都需要对列表进行一次搜索以在有同类项时直接合并。因为链表的大小为 O(MN)，总运行时间为 $O(M^2N^2)$

(b)

(c)O(MNlogMN)的解法可以通过先计算出所有 MN 对的乘积，再用第七章的任意一个算法把它们按幂排列。之后可以很轻易的合并同类项。

(d)算法的选择取决于 M 和 N 的相对大小，如果它们很接近那么(c)的算法更好，如果有一个多项式十分短小，那么(b)的算法更好

3.8 可以将第二章中的 Pow 函数改编成适用于多项式乘法的版本。如果 P 较小，一个标准的方法是用 O(P)的乘法代替 O(logP)的乘法，因为此时乘法将一个大数和一个小数相乘， 这对上一题中(b)的乘法算法很有利。（这里我并没有理解）

3.10 设 M'=M mod N 可以使算法加速，因为此时热土豆不会传递多于一圈，如果当 M'>N/2 时，将热土豆反向传递相应的距离，也可以加快速度。这便需要一个双向链表。尽管这些技巧使当 M 和 N 很大时算法得到明显的加速，但最坏运行时间显然仍有 O(N min(M,N))。当 M=1 时，显然算法是线性的。VAX/VMS 系统下的 C 编译器的内存管理系统对这种情况下的 free 的处理很糟糕，以致于表现出 O(NlogN)的速度。

3.12 翻转单链表可以通过使用一个栈来非递归的实现，但这需要 O(N)的额外空间。图 3.5 中的方法和垃圾回收算法中使用的策略十分相似。在每次 while 循环中的第一条语句执行之前，从链表首端到 PreviousPos 已经被翻转了，剩下的部分，从 CurrentPos 到末端还是正常的顺序。这个算法只需要常数额外空间。

```
/* Assuming no header and L is not empty. */

List
ReverseList( List L )
{
        Position CurrentPos, NextPos, PreviousPos;

        PreviousPos = NULL;
        CurrentPos = L;
        NextPos = L->Next;
        while( NextPos != NULL )
        {
                CurrentPos->Next = PreviousPos;
                PreviousPos = CurrentPos;
                CurrentPos = NextPos;
                NextPos = NextPos->Next;
        }
        CurrentPos->Next = PreviousPos;
        return CurrentPos;
}
```

Fig. 3.5.

3.15 (a)代码如图 3.6

```
/* Array implementation, starting at slot 1 */

Position
Find( ElementType X, List L )
{
        int i, Where;

        Where = 0;
        for( i = 1; i < L.SizeOfList; i++ )
                if( X == L[i].Element )
                {
                        Where = i;
                        break;
                }

        if( Where )             /* Move to front. */
        {
                for( i = Where; i > 1; i-- )
                        L[i].Element = L[i-1].Element;
                L[1].Element = X;
                return 1;
        }
        else
                return 0;       /* Not found. */
}
```

Fig. 3.6.

(b)代码示于图 3.7

```
/* Assuming a header. */

Position
Find( ElementType X, List L )
{
        Position PrevPos, XPos;

        PrevPos = FindPrevious( X, L );
        if( PrevPos->Next != NULL )  /* Found. */
        {
                XPos = PrevPos ->Next;
                PrevPos->Next = XPos->Next;
                XPos->Next = L->Next;
                L->Next = XPos;
                return XPos;
        }
        else
                return NULL;
}
```

Fig. 3.7.

(c)这由一些著名的统计学结论得出，参见第 11 章参考文献中 Sleator 和 Tarjan 的论文。

3.16 (c) 删除操作耗费 O(N)时间，并且在两个大小为 N 的循环内部，因此明显存在一个 $O(N^3)$ 的时间界。更好的时间界为 $O(N^2)$ ，注意到一个长为 N 的表最多有 N 个元素被删除，所以所有的删除操作花费的时间为 $O(N^2)$ ，而其他操作花费的时间也为 $O(N^2)$ ，所以时间界为 $O(N^2)$ 。

(d) $O(N^2)$

(e)给表排序，接着将表遍历一次去除相同元素（相同元素排序后一定相邻）

3.17 (a)优点有：程序容易编写，如果被删除的元素可能在之后重新被插入到相同的位置，那么懒惰删除可能省去插入的开销。缺点有：使用了更多的空间，因为每个元素都有一个附加的域（通常占一个字节），并且被删除的元素的空间没有释放。

3.21 可以通过使其中一个栈从数组底端向上增长，另一个栈从数组顶端向下增长来在一个数组里实现两个栈

3.22 (a) 令 E 为我们的扩展栈。我们使用两个栈来实现 E。一个栈，我们叫做 S，用来执行 Push 和 Pop 操作，另一个栈，我们叫做 M，用来记录最小值。为了实现 Push(X,E)，我们实际执行的是 Push(X,S)。如果 X 比 M 的栈顶元素小，那么同时执行 Push(X,M)。为了实现 Pop(E)，同样地我们实际执行的是 Pop(S)。如果弹出的元素 X 等于 M 的栈顶元素，那么同时执行 Pop(M)。FindMin(E)通过返回 M 的栈顶元素实现。所有这些操作都显然是 O(1)的

(b) 这个证明需要第七章中排序一定会花费 $\Omega(N \log N)$ 时间的结论。进行 O(N)次的取出最小元素(FindMin)+压入另一个栈(Push)+删除最小元素(DeleteMin)，可以完成排序，所以一次 FindMin+Push+DeleteMin 操作的时间界为 $\Omega(\log N)$ ，所以它们三个不能同时需要 O(1)时间。

3.23 三个栈可以通过如下方式实现：一个从数组底端向上增长，一个从数组顶端向下增长，第三个可以从中间开始，向某个方向增长。如果第三个栈和其他两个栈中的一个冲突，就需要移动它。一个合理的策略是把它移动到使它的中心距其他两个栈栈顶元素相等的位置。

3.24 栈空间不会用完，因为只有 49 次调用储存在栈空间中。然而，如第二章所述，它的运行时间是指数型的，因此例程不会在合理的时间内结束。

3.25 队列需要包含指针 Q->Front 和 Q->Rear，分别指向链表的首端和末端。程序的细节留作练习，因为这题可能作为一个编程作业。

3.26 这是一个对队列的直接改造。它也有可能被留作编程作业，所以我们不给出答案。

第4章　树

4.1 (a) A
(b) G, H, I, L, M 和 K

4.2 以节点 B 为例：
(a) A
(b) D 和 E
(c) C
(d) 1
(e) 3

4.3 4

4.4 有 N 个节点，每个节点有 2 个指针，所以总共有 2N 个指针。除了根节点外的每个节点都有一个从它的父节点指向它的指针，共有 N-1 个。剩下的就是 NULL 节点。

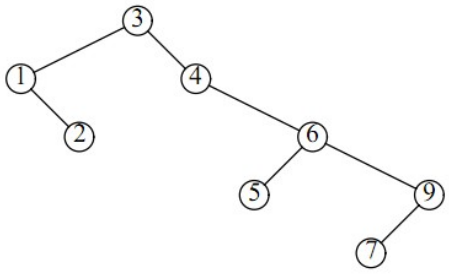
4.5 利用数学归纳法证明。H=0 时问题是平凡的。假设对 H=1,2,…,k 结论均正确，一个高度为 k+1 的二叉树有两个高度最多为 k 的子树，由归纳假设它们有最多 $2^{k+1}-1$ 个节点。这 $2*(2^{k+1}-1)=2^{k+2}-2$ 个节点加上根节点便证明了 H=k+1 的情况，因此对任意高度结论均正确。

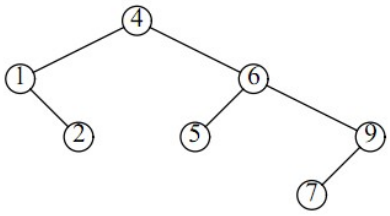
4.6 可以用数学归纳法证明。另外，也可以用下面的方法：设 N 为节点数，F 为满节点数，L 为树叶数，H 为半节点（只有一个儿子的节点）数。很明显，N=F+L+H，进一步，有 2F+H=N-1（见习题 4.4），消元后得 L-F=1。

4.7 可以通过数学归纳法证明。对没有节点的树，所求的和为 0，对只有一个节点的树，它的根是一个深度为 0 的树叶，所以结论是正确的。设结论对有 k 或 k 以下个节点的树均正确。考虑任意一个有 k+1 个节点的树，它有一个有 i 个节点的左子树和一个有 k-i 个节点的右子树。由归纳假设，相对于左子树的根来说，左子树树叶对应的和最大为 1,。因为相对于原二叉树的根来说，每个树叶的深度都增加了 1，所以左子树树叶的和最大为 1/2,，同样，右子树树叶的和最大也为 1/2，因此结论得到证明。等号成立当且仅当没有只有一个儿子的节点。如果有只有一个儿子的节点，那么等号不会成立，因为如果成立那么给这个节点增加一个儿子就会让和超过 1。如果没有只有一个儿子的节点，等号一定成立，因为把一对兄弟树叶去掉之后所求的和不变。重复这个步骤，最终会得到一个只有一个节点的树，它的和为 1。所以原树的和为 1。

4.8 (a) - * * a b + c d e
(b)((a * b) * (c + d)) – e
(c)a b * c d + * e –

4.9





4.11

这个问题和链表的游标法实现差异不大。我们维护一个数组，每个数组元素由一个储存该节点值的域和两个分别为 left 和 right 的整数组成。The free list can be maintained by linking through the left field（此处不知何意）。很容易写出 CursorNew 和 CursorDispose 例程，代替 malloc 和 free。

4.12

(a) 维护一个位数组 B。如果 i 在树上，令 B[i]为 true，反之为 false。不断产生随机数知道找到一个不在树上的数。如果树上有 N 个元素，那么有 M-N 个数不在树上，所以找到一个不在树上的数的概率是(M-N)/M，因此尝试次数的数学期望为 M/(M-N)=α/(α-1)

(b) 为了找到一个已在树上的数，不断产生随机数直到产生一个已经在树上的数。每次产生出满足条件的数概率是 N/M，所以次数的数学期望是 M/N=α。

(c) 一次 insert 和 delete 操作的总花费是 α/(α-1)+ α=1+α+1/(α-1)。α=2 时取得最小值。

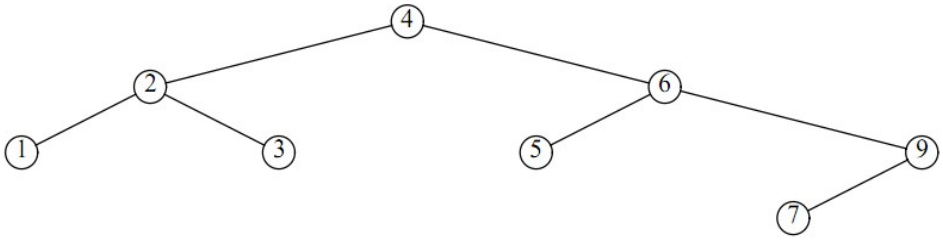
4.15

(a) N(0) = 1, N(1) = 2, N(H) = N(H-1) + N(H-2) + 1

(b) N(H) = F(H+2) - 1

（译者注：原文为“The heights are one less than the Fibonacci numbers”，似有错误，F(N)为第 N 个斐波那契数，此书定义斐波那契数为 F(0) = 1，F(1) = 1，F(N) = F(N - 1) + F(N - 2)）

4.16



4.17

很容易在纸上验证对 1<=k<=3，命题是成立的。设命题对 k=1, 2, 3, …, H 均成立。当 k=H+1 时，分析如下：

在经过前 2^H-1 次插入操作后，2^{H-1} 位于根，且右子树是一个含有 $2^{H-1}+1$ 到 2^H-1 的平衡树。接下来 2^{H-1} 个，也就是第 2^H 到第 $2^H+2^{H-1}-1$ 个插入操作中的每一个都给树插入了一个新的最大值并且插入到右子树中，最终形成一个高度

平衡的右子树。这一点可以从归纳假设得出，因为右子树可以看作由 $2^{H-1}+1$ 到 $2^{H}+2^{H-1}-1$ 这一段连续的整数插入形成的，正好有 $2^{H}-1$ 个。下一次插入在根产生了不平衡，引起一次单旋转。容易验证这次单旋转使 2^{H} 成为新的根，并形成了一个高为 H-1 的完全平衡的左子树。刚刚插入的新元素依附在高度为 H-2 的完全平衡的右子树上。因此右子树正好可以看成是 $2^{H}+1$ 到 $2^{H}+2^{H-1}$ 被依次插入形成的。由归纳假设，接下来的 $2^{H}+2^{H-1}+1$ 到 $2^{H+1}-1$ 这些数的插入会形成一个高为 H-1 的完全平衡的右子树。因此在最后一次插入之后，左子树和右子树都完全平衡，且高度相同，因此整个 $2^{H+1}-1$ 个节点构成的数也是完全平衡的（且高度为 H）

其余的两个函数是书中两个的镜像，只要交换每一处的 Right 和 Left 就可以。

4.18 在执行标准的二叉查找树删除算法后，位于根到被删除节点的路径上的节点可能需要旋转来重新平衡。不同于插入操作，删除操作可能会使多个节点需要旋转。

4.20 (a) O(loglogN)
(b) 是高度为 255 的最小 AVL 树（这个树十分巨大）

4.21 如图

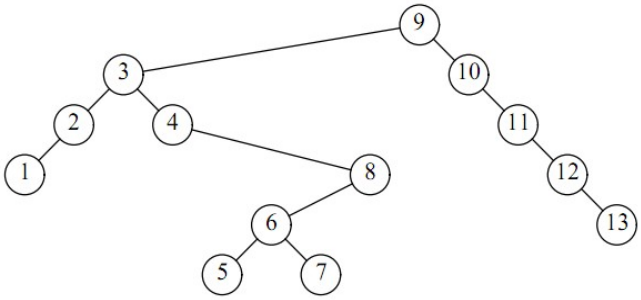
4.22

```
Position
DoubleRotateWithLeft( Position K3 )
{
    Position K1, K2;

    K1 = K3->Left;
    K2 = K1->Right;

    K1->Right = K2->Left;
    K3->Left  = K2->Right;
    K2->Left  = K1;
    K2->Right = K3;
    K1->Height = Max( Height(K1->Left), Height(K1->Right) ) + 1;
    K3->Height = Max( Height(K3->Left), Height(K3->Right) ) + 1;
    K2->Height = Max( K1->Height, K3->Height ) + 1;

    return K3;
}
```

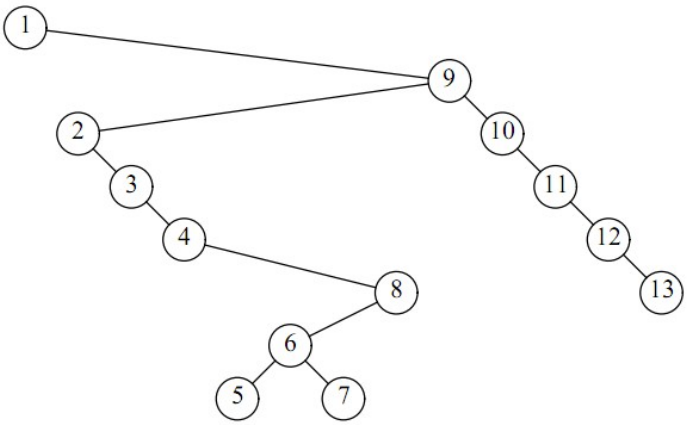
访问 3 后，

4.23

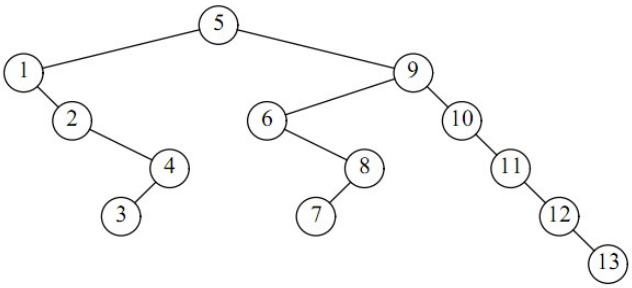
为 H-1
的 完 全



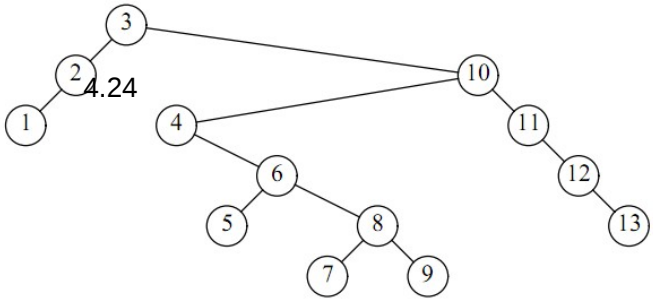
访问 1 后，



访问 5 后，



4.24



访 问 9 后，

4.25

523776

(b) 262166, 133114, 68216, 36836, 21181, 13873

(c) 执行 Find(9)之后

4.26

(a) 数学归纳法可以轻易的证明。

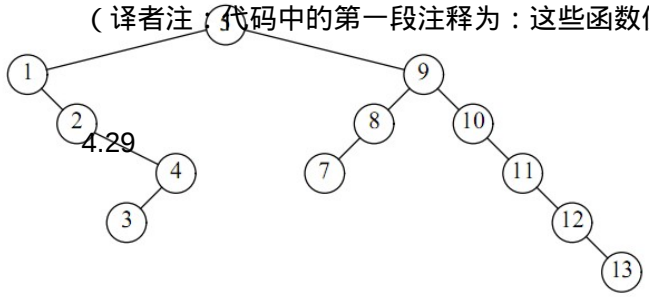
4.28

(a-c) 所有这些例程都耗费线性时间

```
/* These functions use the type BinaryTree, which is the same */
/* as TreeNode *, in Fig 4.16. */

int
CountNodes( BinaryTree T )
{
        if( T == NULL )
                return 0;
        return 1 + CountNodes(T->Left) + CountNodes(T->Right);
}

int
CountLeaves( BinaryTree T )
{
        if( T == NULL )
                return 0;
        else if( T->Left == NULL && T->Right == NULL )
                return 1;
        return CountLeaves(T->Left) + CountLeaves(T->Right);
}

 /* An alternative method is to use the results of Exercise 4.6. */

 int
 CountFull( BinaryTree T )
 {
         if( T == NULL )
                 return 0;
         return ( T->Left != NULL && T->Right != NULL ) +
                         CountFull(T->Left) + CountFull(T->Right);
 }
```

（译者注：代码中的第一段注释为：这些函数使用了 BinaryTree 类型，它和

4.29



(a)

4.6 的结论)

假设存在函数 RandInt(Lower, Upper)，在一个适当的间隔里产生均匀分布的随机数。当 N 不是正数，或过大使得空间不足时，MakeRandomTree 返回 NULL。

4.30

```
SearchTree
MakeRandomTree1( int Lower, int Upper )
{
        SearchTree T;
        int RandomValue;

        T = NULL;
        if( Lower <= Upper )
        {
                T = malloc( sizeof( struct TreeNode ) );
                if( T != NULL )
                {
                        T->Element = RandomValue = RandInt( Lower, Upper );
                        T->Left  = MakeRandomTree1( Lower, RandomValue - 1 );
                        T->Right = MakeRandomTree1( RandomValue + 1, Upper );
                }
                else
                        FatalError( "Out of space!" );
        }
        return T;
}

SearchTree
MakeRandomTree( int N )
{
        return MakeRandomTree1( 1, N );
}
```

如图

TreeNode* 是同一个类型，第二段注释为另一个方法是用习题

```
/* LastNode is the address containing last value that was assigned to a node */

SearchTree
GenTree( int Height, int *LastNode )
{
        SearchTree T;

        if( Height >= 0 )
        {
                T = malloc( sizeof( *T ) ); /* Error checks omitted; see Exercise 4.29. */
                T->Left = GenTree( Height - 1, LastNode );
                T->Element = ++*LastNode;
                T->Right = GenTree( Height - 2, LastNode );
                return T;
        }
        else
                return NULL;
}

SearchTree
MinAvlTree( int H )
{
        int LastNodeAssigned = 0;
        return GenTree( H, &LastNodeAssigned );
}
```

（译者注：图中第一处注释为：LastNode 是上一次赋给节点的值的地址，第二处注释为：错误处理已被忽略；见习题 4.29）

4.31 有两种显而易见的方法。一种是模仿习题 4.29 中的思路，将 RandInt(Lower, Upper)替换为(Lower + Upper) / 2。这种方法需要计算出 $2^{H+1}-1$ ，不过也不是那么困难。另一种方法模仿前一道习题的思路，不同之处是两个子树的高度都是 H-1。这种方法的代码如下：

```
/* LastNode is the address containing last value that was assigned to a node. */

SearchTree
GenTree( int Height, int *LastNode )
{
        SearchTree T = NULL;

        if( Height >= 0 )
        {
                T = malloc( sizeof( *T ) );  /* Error checks omitted; see Exercise 4.29. */
                T->Left = GenTree( Height - 1, LastNode );
                T->Element = ++*LastNode;
                T->Right = GenTree( Height - 1, LastNode );
        }
        return T;
}

SearchTree
PerfectTree( int H )
{
        int LastNodeAssigned = 0;
        return GenTree( H, &LastNodeAssigned );
}
```

（译者注：图中注释与上一题相同）

4.32

这被称作一维搜索问题。如果有非常多的节点被输出，那么执行遍历的时间将是O(K)，另外，时间也和树的深度成正比，因为我们可能一直向下搜索到树叶（比如不存在符合条件的节点）。因树的深度平均为O(logN)，所以有O(K + logN)的时间界。

```
void
PrintRange( ElementType Lower, ElementType Upper, SearchTree T )
{
        if( T != NULL )
        {
                if( Lower <= T->Element )
                        PrintRange( Lower, Upper, T->Left );
                if( Lower <= T->Element && T->Element <= Upper )
                        PrintLine( T->Element );
                if( T->Element <= Upper )
                        PrintRange( Lower, Upper, T->Right );
        }
}
```

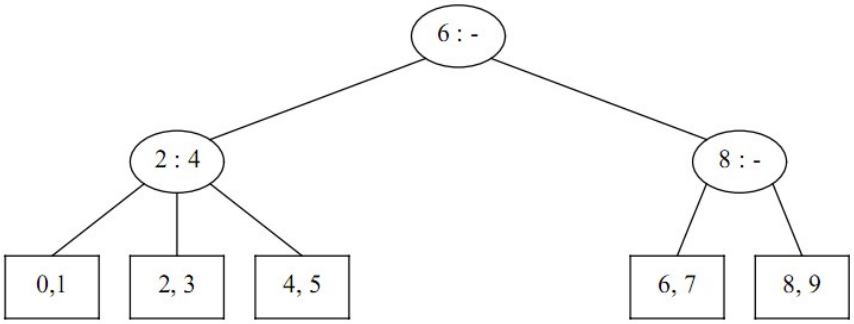
4.33

此题和习题4.34是可能被留作编程作业，因此我们不给出代码。

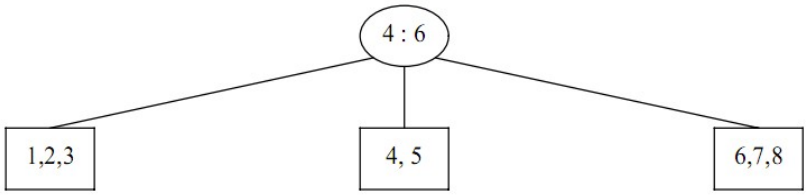
4.35

将根放在一个空队列，然后不断从队列中弹出节点并把这个节点的左右儿子（如果有的话）压入队列，直到队列为空。这耗费O(N)时间，因为每个队列操作耗费常数时间，共有N个出队列和N个入队列操作。

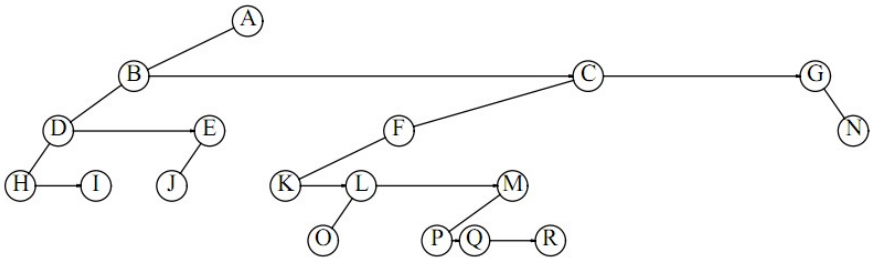
4.36 (a)



(b)



4.39



4.41 如下所示的函数显然有线性的时间界，因为最坏情况下它会遍历 T1 和 T2。

```
int
Similar( BinaryTree T1, BinaryTree T2 )
{
        if( T1 == NULL || T2 == NULL )
                return T1 == NULL && T2 == NULL;
        return Similar( T1->Left, T2->Left ) && Similar( T1->Right, T2->Right );
}
```

4.43 最简单的方法是用线性时间计算两棵树节点的中序遍历序号。如果 T2 的根节点的序号为 x，就在 T1 中找到同为 x 的节点并且旋转至根部。对 T1 的左子树和右子树递归的运用这个策略（通过查找 T2 左子树和右子树的根节点的值）。如果 d_N 是 x 的深度，那么运行时间满足 $T(N)=T(i)+T(N-i-1)+d_N$ ，其中 i 是左子树的大小。

在最坏情况下d_N总是O(N)，i总是0，所以最坏运行时间是二次的。假设i的各取值是等可能的（这看起来是有道理的），即使d_N仍是O(N)，T(N)的平均值也会下降到O(NlogN)。这个常见的递推式在这章经常出现，并且在第七章中得到了这个递推式的解。若更合理地假设d_N是O(logN)，那么运行时间将是O(N)。

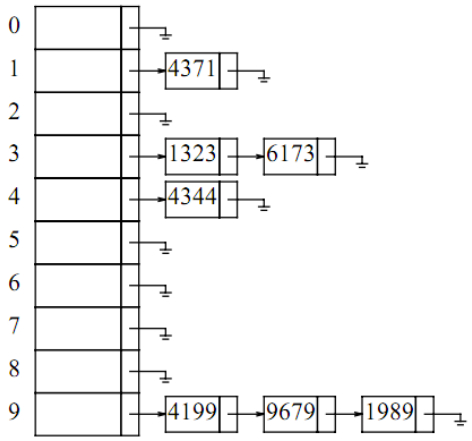
4.44 给每个节点增加一个域储存以它为根的子树的大小。可以通过计算中序遍历序号实现这一点。

4.45 (a) 需要增加一个域来标记线索
(c) 从某种程度上可以更简单的对树进行遍历且不需要递归。

第5章　散列

5.1

(a) 假设我们把元素加入至表的末端（这样在用手画时方便一些），分离链接散列表的结果如下：



(b)

0	9679
1	4371
2	1989
3	1323
4	6173
5	4344
6	
7	
8	
9	4199

(c)

0	9679
1	4371
2	
3	1323
4	6173
5	4344
6	
7	
8	1989
9	4199

5.4

(d)
1989 无法插入到表中因为 $hash_2(1$，可选5.5择的位置5,1,7和 3 都已经被占用，这时的表如下5.6所示：

0	
1	4371
2	
3	1323
4	6173
5	9679
6	
7	4344
8	
9	4199

再散列时，我们将表的大小选为一个大致是原表两倍的素数。在这道题中，合适的大小为 19，相应的散列函数 h(x) = x (mod 19)。

5.2 (a) 4371 的新位置为 1，1323 为 12，6173 为 17，4344 为 12，4199 为 0，9679 为 8，1989 为 13

(b) 9679 的新位置为 8，4371 为 1，1989 为 13，1323 为 12，6173 为 17，4344 为 14 因为 12 和 13 都已经被占用，4199 为 0

(c)　　别谨慎。设 p 为即将再散列为一个更小的表的临界状态下的装填因子。如果新的表大
9679 的小为 N，那么它就有 2pN 个元素。新表在经过 2N-2pN 次插入，或 pN 次删除之后
新 位 置都会引起再一次再散列。为了平衡两种可能的花销，p=2/3 是正确的选择。比如假设
为　　我们有一个大小为 300 的表。如果我们在有 200 个元素时再散列，那么新的表的大
8，437小为 150，我们可以进行 100 次插入操作或 100 次删除操作而不引起再一次再散列。
1　　为如果我们知道插入操作将比删除操作更频繁，这个 p 可能是偏大的。如果 p 太接近
1，1981.0，在删除少数几个元素后再插入多个元素就可以引起频繁的再散列。最坏的情况，
9　　为若 p=1.0，则（在临界状态时进行）交替的删除和插入每一次都可以引起一次再散
13，13列。
23　为
12，61(a) 因为每一个表项都被探测到，所以如果表不满，冲突一定会被解决。
73　为(b) 消除了一次聚集，但没有消除二次聚集，因为所有元素均使用同一种解决冲突
17，43的序列。
44　为(c, d) 运行时间可能和平方探测相似。它的优点是除非表满了否则插入不会失败。
16 因为(e) 一种不能产生随机数（即使是伪随机数）的方法能在参考文献中找到。我们可以
12　和使用习题 2.7 中的方法。
13 都已
经 被 占分离链接散列表需要使用指针，这会占用空间，并且引起调用空间配置例程的开销
用 ，41这通常是昂贵的。线性探测易于实现，但在装填因子增大时因为一次聚集它的表现
99 为 0 严重下降。平方探测实现起来只稍稍比线性探测复杂，且在实践中有很好的表现。但
(d)　　在表半满的时候插入可能失败，不过这不经常出现。再散列消除了一次和二次聚
9679 的
新 位 置
为
8，4375.7
1　　为
1，198
9　　为
13，13
23　为
12，61
73　为
17，43
44　为
15 因为
12 已经
被 占 用
4199 为5.8
0
　　5.9
我 们 必
须 对 频
繁 的 再
散 列 特

用标准的排序算法给这 MN 项排序并合并同类项需要 O(MNlogMN) 时间。如果通过散列表来合并同类项，那么每一项只需要花常数时间的开销，共需 O(MN) 的时间。如果输出的多项式只有 O(M+N) 项，那么可以在 O((M+N)log(M+N)) 的时间内给它们排序，这时间小于 O(MN)。因此总时间是 O(MN)。之所以这个界能更好是因为这题给出的限制较少：它不要求一定要用两个元素的比较来合并同类项，而是使用散列表。用桶式排序代替标准的排序算法也可以得到相似的时间界。在实践中，类似于哈希的操作要比比较操作耗费的时间多得多，因此这个时间界不一定能真的带来改进。另一方面，如果预计输出的多项式只有 O(M+N) 项，使用散列表可以节省大量

5.10 的空间，因为在这个情况下散列表只需要 O(M+N) 的空间。

实现这些操作的另一种方法是使用二叉查找树而不是散列表；我们需要一个平衡树因为插入到树中的元素十分有序。使用伸展树可能很适合这类问题，它对连续的访问表现良好。比较解决这个问题的各种方法是一个很好的编程作业。

5.14 表大概含 60,000 项，每项需要 8 字节，总共需要 480,000 字节。

(a) 正确。

(b) 即使一个单词被散列至一个其值为 1 的位置，也不能保证这个单词一定在词典中。可能它仅仅是被散列到了一个正好和其它某个在词典中的词一样的值。在这道题的情况中，表大约被填满了 10%（表大小为 300,007，有 30,000 个单词），所以一个不在词典里的单词有 10%的可能被散列至一个其值为 1 的位置。

(c) 在大多数机器上 300,007 位是 37,501 字节。

集 ， 但(d) 如在(b)中所讨论的，该算法会有十分之一的拼写错误不能检查出来。

第 二 个(e) 一个 20 页的文档应有 60 个拼写错误。该算法平均可检测出里面的 54 个。一个三

散 列 函倍大的 100KB 的表可以将算法出现错误的次数的期望降为 2。这对许多应用来说已

数 的 计经足够，特别是拼写检查这项工作本身也是一个十分不精确的事情。许多单词（特

算 也 可别是短单词）拼写错误后仍然是一个单词。例如，如果把 then 错写 them，不会有

能 是 开任何算法可以检查出来。

销 较 大

的。Gon对每个散列表中的位置，我们可以添加一个额外的域叫做 WhereOnStack，并且维

net 和护一个额外的栈。每个位置有第一个插入时，就把该位置的地址（或序号）压入栈

Baeza- 中，并让 WhereOnStack 指向栈顶。当访问一个位置时，检查 WhereOnStack 是否

Yates[8 指向栈的有效部分，且被 WhereOnStack 所指的栈中的项是否有该位置的地址。

]比较了

几 种 散

列 策 略

他 们 的

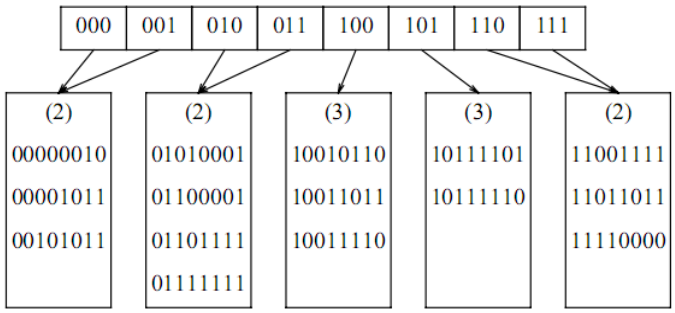
结 果 表

明 平 方

探 测 是

最 快 的

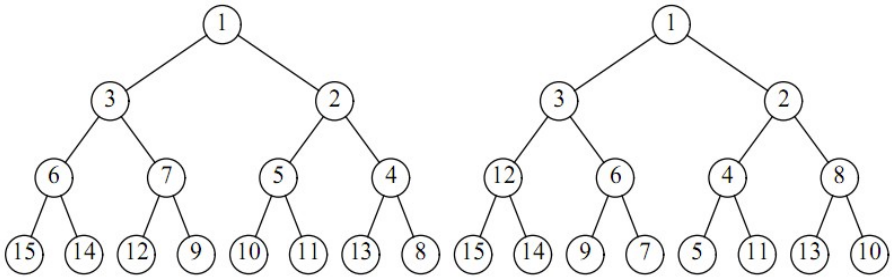
方法。



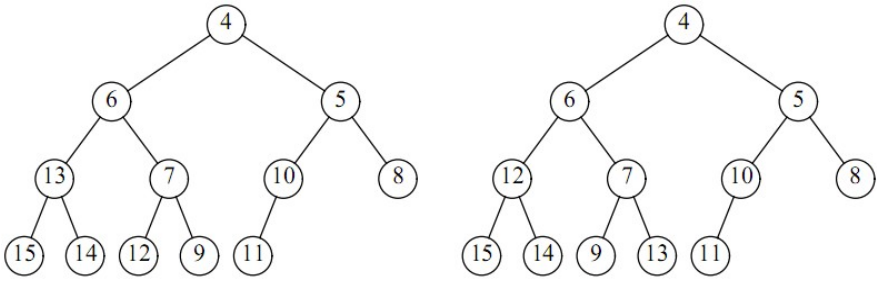
第6章 优先队列（堆）

6.1 当元素插入时，比较它和当前的最小值的大小，如果新元素较小则更改最小值 。DeleteMin 在这种体系中十分昂贵。

6.2



6.3 对习题 6.2 中的两个堆执行三次 DeleteMin 操作后的结果如下：



6.4 仅需要对教材中的代码进行简单的修改，留作编程作业。

6.5

6.6 225。为了算出这个值，设 i=1，从根开始，沿着指向最后一个节点的路径向下，如果遇到左儿子，将 i 乘以 2，如果遇到右儿子，将 i 乘以 2 再加 1。

6.7 (a) 设 H(N)为 N 个节点的完全二叉树各节点高度之和，我们证明 H(N) = N - b(N)，其中 b(N)是 N 的二进制表示中 1 的个数。对 N=0 和 N=1 的情况命题均成立。假设对 k 到 N-1（含 N-1）的每个值命题均成立。设左子树和右子树分别含 L 和 R 个节点。

因为根节点的高度为 $\lfloor \log N \rfloor$ ，我们有

$$\begin{aligned} H(N) &= \lfloor \log N \rfloor + H(L) + H(R) \\ &= \lfloor \log N \rfloor + L - b(L) + R - b(R) \\ &= N - 1 + (\lfloor \log N \rfloor - b(L) - b(R)) \end{aligned}$$

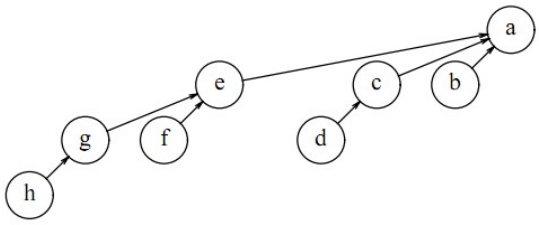
其中第二行由归纳假设得来，第三行由 L + R = N – 1 得来。最后一个节点要么在左子树，要么在右子树。如果在左子树，那么右子树就是一个完全二叉树，则 $b(R) = \lfloor \log N \rfloor - 1$。更进一步，N 和 L 的二进制表示是一样的，只是 N 中的前导 10 在 L 中变成了 1。（例如如果 N=37=100101，L=10101。）显然如果最后一个节点在左子树则 N 的第二位一定是 0。因此这种情况下，b(L) = b(N)，并且有

可以。

如果最后一个节点在右子树，则 $b(L)=\lfloor \log N \rfloor$ ，R 的二进制表示除了 N 的前导 1 在 R 中没有之外也是一样的。（例如如果 N=27=11011，R=1011），因此 b(R) = b(N) – 1，同样有

H(N) = N – b(N)

(b) 对这八个元素进行一场“单淘汰制比赛”。总共需要七次比较，产生的排序信息体现在这个二项树中。



第八次比较在 b 和 c 间进行。如果 c 小于 b，则令 b 为 c 的儿子，否则令 c 和 d 都为 b 的儿子。

(c) 使用递归的策略。假设 $N=2^k$ 。像(b)一样对 N 个元素建立一个二项树。接着根的最大子树递归地转化为 2^{k-1} 个元素的二叉堆。然后将堆的最后一个元素（它在堆中单独占一行）插入到剩下的二项树组成的二项队列里（译者注：指的应该是根的各个子树组成的二项队列），因而形成另一个 2^{k-1} 个元素的二项树。这时，根节点有一个可被看作是 $2^{k-1}-1$ 个元素形成的堆的子树，和另一个可被看作 2^{k-1} 个元素组成的二项树的子树。递归地把这个子树变成堆；这时整个结构都成为了二叉堆。对 $N=2^k$ 运行时间满足 $T(N)=2T(N/2)+\log N$ 。基准情况是 T(8) = 8。

6.8

令 $D_1, D_2, ..., D_k$ 分别为代表第一个、第二个、...、第 k 个最小元的深度的随机变量。

我们需要计算 $E(D_k)$ 。在下面我们假设堆的大小 N 是一个等于 2 的幂减 1 的数（这就是说堆的最后一层被填满），且充分大以使以 O(1 / N) 为界的项可以被忽略。我们可以不失一般性地假设第 k 个最小元在根的左子树中。令 $p_{j,k}$ 为这个元素在子堆中是第 j 个最小元的概率。

引理：对 k > 1，$E(D_k)=\sum_{j=1}^{k-1} p_{j,k}(E(D_j)+1)$

证明：在左子堆中深度为 d 的元素在整个堆中的深度为 d + 1。由 $E(D_j+1)=E(D_j)+1$ ，引理得到证明。

H(N) = N – b(N)

堆的最后一层是满的，第二、第三、...、第 k – 1 个最小元有 0.5 的概率在左子堆中（实际上，在右子树的概率应当为 $\frac{1}{2}-\frac{1}{N-1}$ ，在左子树的概率应当为 $\frac{1}{2}+\frac{1}{N-1}$ ，因为我们已经把第 k 个最小元放到了右边。回忆一下我们假设了大小为 O(1 / N) 的项可被忽略），因而

$$p_{j,k} = p_{k-j,k} = \frac{1}{2^{k-2}}\binom{k-2}{j-1}$$

定理： $E(D_k) \le \log k$ 。

证明：使用数学归纳法证明。显然 k = 1 或 k = 2 时定理成立。接着我们要证明对任意 k > 2，若命题对任意比 k 小的数成立，则命题对 k 成立。现在，由归纳假设，对任意的 $1 \le j \le k-1$ 来说，有

$$E(D_j) + E(D_{k-j}) \le \log j + \log(k-j)$$

因为对 x > 0，函数 $f(x) = \log x$ 是凸的，所以由 Jensen 不等式

$$\log j + \log(k-j) \le 2\log(k/2)$$

因此

$$E(D_j) + E(D_{k-j}) \le \log(k/2) + \log(k/2)$$

更进一步，因为 $p_{j,k} = p_{k-j,k}$ ，有

$$p_{j,k}E(D_j) + p_{k-j,k}E(D_{k-j}) \le p_{j,k}\log(k/2) + p_{k-j,k}\log(k/2)$$

由引理，有

$$\begin{aligned} E(D_k) &= \sum_{j=1}^{k-1} p_{j,k}(E(D_j)+1) \\ &= 1 + \sum_{j=1}^{k-1} p_{j,k}E(D_j) \end{aligned}$$

因而

$$\begin{aligned} E(D_k) &\le 1 + \sum_{j=1}^{k-1} p_{j,k}\log(k/2) \\ &= 1 + \log(k/2)\sum_{j=1}^{k-1} p_{j,k} \\ &= 1 + \log(k/2) \\ &= \log k \end{aligned}$$

由之前做过的假设，

证明完毕。

可以证明， $E(D_k)$ *渐近于* $\log(k-1)-0.273548$

6.9 (a) 对堆进行前序遍历。
(b) 可以扩展到左式堆和斜堆，对 d-堆来说运行时间是 O(Kd)

6.11 模拟显示线性时间算法对最坏的输入和随机输入都表现得更快。

6.12 (a) 如果堆是最小堆，从根处的空穴开始，沿最小的儿子向下形成一条路径直到树叶，这大概要花费 logN 次比较。为了寻找空穴最终要移动到的位置，对这 logN 个元素进行二分查找。这要花费 O(loglogN)次比较。
(b) 寻找最小儿子形成的路径，在经过 logN – loglogN 层后停止。这时很容易决定空穴应安放于终止点的上方或下方。如果在下方，继续寻找路径，但只对最后的 loglogN 个元素进行二分查找，总共需 logN + logloglogN 次比较。如果在上方，对前 logN – loglogN 个元素进行二分查找。二分查找最多需 loglogN 次比较，路径的寻找耗费了 logN – loglogN 次比较，总共需 logN 次。所以第一种情况是最坏的情况。
(c) 时间界可以被改进到 $\log N+\log^* N+O(1)$ ，其中 $\log^* N$ 是 Ackerman 函数的反函数（见第 8 章）。这个界可以在参考文献[16]中找到。

6.13 其父亲在 $\lfloor (i+d-2)/d \rfloor$ ，其儿子在 $(i-1)d+2,...,id+1$

6.14
(a) $O((M+dN)\log_d N)$

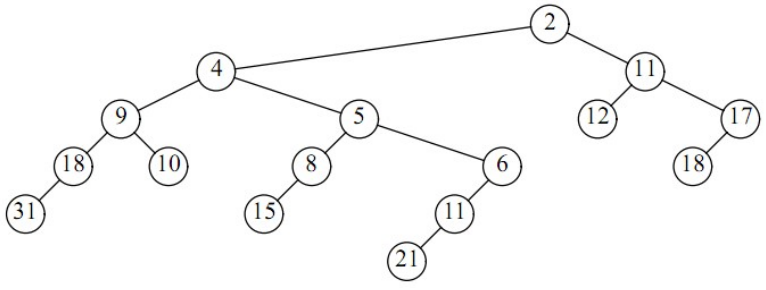
(b) $O((M+N)\log N)$

(c) $O(M+N^2)$

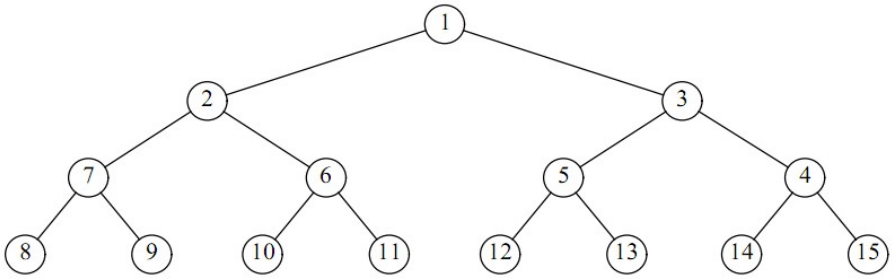
(d) $d=\max(2,M/N)$

（见章节 11.4 末尾的相关讨论）

6.16



6.17



6.18 结论是成立的。证明与习题 4.17 十分相似。

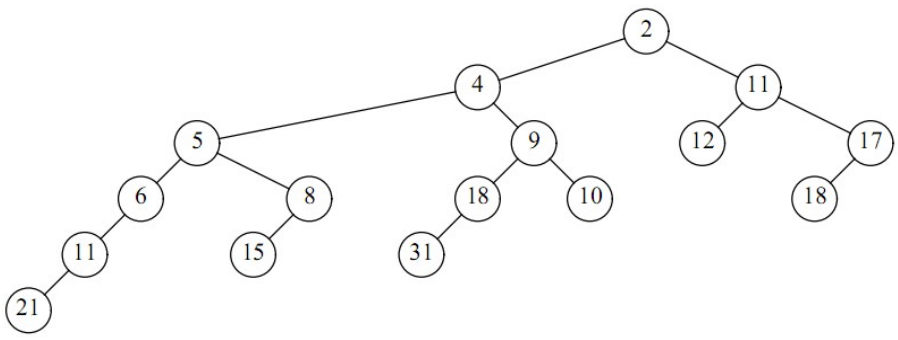
6.19 如果元素被反序输入，就会形成一个含有一连串左儿子的左式堆。这是最佳左式堆因为它的右路径最短。

6.20 (a) 如果将 DecreaseKey 应用于一个十分深（十分“左”）的节点，上滤的时间将是令人望而却步的。所以上滤的方法并不可行。然而我们仍然可以通过结合 Delete 和 Insert 来高效率的进行 DecreaseKey 操作。为了删除堆中任意一个节点 x，我们用 x 的左子堆和右子堆的合并结果来替代掉 x。这可能会在 x 的父节点到根的路径上造成不平衡，如果出现这种情况需要交换两个儿子来解决。然而，容易证明最多只有 logN 个节点被影响，保证了时间界。在第 11 章中也有相关讨论。

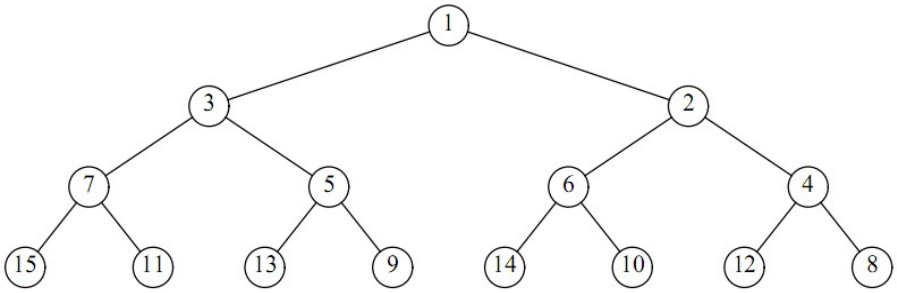
6.21 左式堆的懒惰删除在 Cheriton 和 Tarjan 的论文[9]中有所探讨。大致的思路是，如果根节点被标记为删除，则对堆进行前序遍历，将被标记节点中的 frontier（译者注：这里不能理解）移除，留下一系列堆。这些堆可以通过这样的方式两个两个地合并：将这些堆压入队列中，取出两个堆，合并它们，然后将结果再次压入队列，队列中只剩一个堆时终止。

6.22 (a) 标准的做法是把操作分解为几个部分。每当第一个元素再次在出队列的堆中出现时便是一个新部分的开始。第一部分大致耗费 2 * 1 * (N / 2)个时间单元，因为有 N / 2 个对右路径（译者注：含根节点）上只有一个节点的树进行的合并操作。下一个部分耗费 2 * 2 * (N / 4)个时间单元，因为大致有 N / 4 个对右路径上至多只有两个节点的树进行的合并操作。第三部分耗费 2 * 3 * (N / 8)个时间单元，以此类推。它们和收敛于 4N。
(b) 这样产生的左式堆更“左”。

6.23



6.24



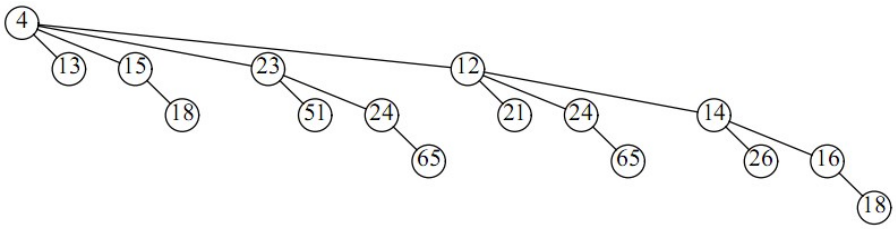
6.25 这个命题也是正确的，证明和习题 4.17 和习题 6.18 相似。

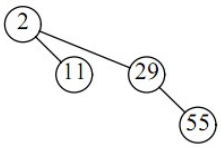
6.26 可以。每一步操作的时间从 6.22 中的最坏时间变成了摊还时间，根据摊还分析的定义，它们的和就是对一个序列来说的最坏时间界。

6.27 显然命题对 k = 1 成立。假设对任意的 i = 1, 2, …, k 命题均成立。二项树 B_{k+1} 是由将一个二项树 B_k 连接到另一个 B_k 的根上形成的。因此由数学归纳法，它包含 B_0 到 B_{k-1}，也包含 B_k，因此命题得证。

6.28 通过数学归纳法证明。显然命题对 k = 1 成立。假设对任意的 i = 1, 2, …, k 命题均成立。二项树 B_{k+1} 是由将一个二项树 B_k 附着到另一个 B_k 的根上形成的。原树在深度 d 上有 $\binom{k}{d}$ 个节点，附着上的另一棵树在深度 d – 1 上有 $\binom{k}{d-1}$ 个节点，这些节点在新树中也位于深度 d。把这两项相加，使用一个众所周知的公式就可以证明这个结论。（译者注：公式为 $\binom{m}{n}+\binom{m}{n+1}=\binom{m+1}{n+1}$）

6.29





6.30　在第 11 章中有证明。

6.31　见第 11 章。

6.35　在堆中不要存储它们的值，而只存储每个节点与它的父节点值的差。

6.36　O(N + klogN) 要好于 O(Nlogk)。如果 k = O(N / logN)，则第一个时间界为 O(N)。只要 k 在增长，第二个时间界就会比第一个时间界大。如果 $\Omega(N / \log N) = k = o(N)$ ，第一个界更好。如果 $k = \Theta(N)$ ，两个界相当。