《算法导论》学习笔记

Version 1.0

项目托管：http://code.google.com/p/introduction-to-algorithms-notes/
Blog：https://sites.google.com/site/chuanqitan/
E-mail：chuanqi.tan(at)gmail.com
欢迎交流与反馈错误等，转载及引用请注明
By Hannosogno @ BIT 2011
QQ 121131818

第一部分：基础知识

第 1 章：算法在计算中的作用

1. 算法即是一系列的计算步骤，用来将一个有效的输入转换成一个有效的输出。
2. 计算机的有限的资源必须被有效的利用，算法就是来解决这些问题的方法。

第 2 章：算法入门

1. 循环不变式的三个性质：（循环不变式通常用来证明递归的正确性）
    1. 初始化：它在循环的第一轮迭代开始之前，应该是正确的。
    2. 保持：如果在循环的某一次迭代开始之前它是正确的，那么，在下一次迭代开始之前，它也应该保持正确。
    3. 终止：当循环结束时，不变式给了我们一个有用的性质，它有助于表明算法是正确的。
2. 伪代码中的约定：
    1. 书写上的“缩进”表示程序中的分程序（程序块）结构。
    2. while,for,repeat 等循环结构和 if,then,else 条件结构与 Pascal 中相同。
    3. 符号 "▷”表示后面部分是个注释。
    4. 多重赋值 i←j←e 是将表达式 e 的值赋给变量 i 和 j；等价于 j←e，再进行赋值 i←j。
    5. 变量（如 i,j 和 key 等）是局部给定过程的。
    6. 数组元素是通过“数组名[下标]”这样的形式来访问的。
    7. 复合数据一般组织成对象，它们是由属性(attribute)和域(field)所组成的。
    8. 参数采用按值传递方式：被调用的过程会收到参数的一份副本。
    9. 布尔运算符"and”和"or”都是具有短路能力。
3. 算法分析即指对一个算法所需要的资源进行预测。
4. 对于一个算法，一般只考察其最坏情况的运行时间，理由有三：
    1. 一个算法的最坏情况运行时间是在任何输入下运行时间的一个上界。
    2. 对于某些算法来说，最坏情况出现得还是相当频繁的。
    3. 大致上看来，“平均情况”通常和最坏情况一样差。
5. 分治策略：将原问题划分成 n 个规模较小而结构与原问题相似的子问题；递归地解决这些小问题，然后再合并其结果，就得到原问题的解。
6. 分治模式在每一层递归上都有三个步骤：
    1. 分解(Divde)：将原问题分解成一系列子问题；
    2. 解决(Conquer)：递归地解答各子问题。若子问题足够小，则直接求解；
    3. 合并(Combine)：将子问题的结果合并成原问题的解。

第 3 章：函数的增长

1. 对几个记号的大意：o（非渐近紧确上界） ≈ <； O（渐近上界） ≈ ≤；
   Θ（渐近紧界）≈ =； Ω（渐近下界）≈ ≥； ω（非渐近紧确下界）≈ >;
   这里的<,≤,=,≥,>指的是规模上的比较，即 o(g(n))的规模比 g(n)小。

- o(g(n))={ f(n):
  对任意正常数 c，存在常数 $n_0>0$，使对所有的 $n \geqq n_0$，有 $0 \leqq f(n)<cg(n)$ }

- O(g(n))={ f(n):
  存在正常数 c 和 n_0，使对所有 n≧n_0，有 0≦f(n)≦cg(n) }
- Θ(g(n))={ f(n):
  存在正常数 c_1、c_2和 n_0，使对所有的 n≧n_0，有 0≦c_1g(n)≦f(n)≦c_2g(n) }
- Ω(g(n))={ f(n):
  存在正常数 c 和 n_0，使对所有 n≧n_0，有 0≦cg(n)≦f(n) }
- ω(g(n))={ f(n):
  对任意正常数 c，存在常数 n_0>0，使对所有的 n≧n_0，有 0≦cg(n)<f(n) }

第 4 章：递归式

8. 递归式是一组等式或不等式，它所描述的函数是用在更小的输入下该函数的值来定义的。例如 Merge-Sort 的最坏情况运行时间 T(n)可以用以下递归式来表示：

$$T(n) = \begin{cases} 2T(n/2) + n, & \text{if } n > 1 \\ 1, & \text{if } n = 1 \end{cases}$$

9. 解递归式的方法主要有三种：代换法、递归树方法、主方法。
10. 代换法(Substitution method)(P38~P40)
    定义：先猜测某个界的存在，再用数学归纳法去证明该猜测的正确性。
    缺点：只能用于解的形式很容易猜的情形。
    总结：这种方法需要经验的积累，可以通过转换为先前见过的类似递归式来求解。
11. 递归树方法(Recursion-tree method)
    起因：代换法有时很难得到一个正确的好的猜测值。
    用途：画出一个递归树是一种得到好猜测的直接方法。
    分析(重点)：在递归树中，每一个结点都代表递归函数调用集合中一个子问题的代价。将递归树中每一层内的代价相加得到一个每层代价的集合，再将每层的代价相加得到递归式所有层次的总代价。
    总结：递归树最适合用来产生好的猜测，然后用代换法加以验证。
    递归树的方法非常直观，总的代价就是把所有层次的代价相加起来得到。但是分析这个总代价的规模却不是件很容易的事情，有时需要用到很多数学的知识。
12. 主方法(Master method)
    主方法是最好用的 Cookbook 方法，太神奇了，可以瞬间估计出递归算法的时间复杂度，主方法总结了常见的情况并给出了一个公式。
    优点：针对形如 T(n)=aT(n/b)+f(n)的递归式
    缺点：并不能解所有形如上式的递归式的解。因为主方法在第 1 种情况与第 2 种情况之间、第 2 种情况与第 3 种情况之间都存在着一条沟，所以会存在着不能适用的情况。
    直觉上：实际上主方法一直在比较 f(n)与 $N^{Log}{}_b{}^a$ 的规模，然后选取规模大的作为最后的递归式的规模。
    主方法：设 a≥1 和 b≥1 是常数 f (n)是定义在非负整数上的一个确定的非负函数。又设 T(n)也是定义在非负整数上的一个非负函数，且满足递归方程 T(n)=aT(n/b)+f(n)。方程 T(n)=aT(n/b)+f(n)中的 n/b 可以是[n/b]，也可以是 n/b。那么，在 f(n)的三类情况下，我们有 T(n)的渐近估计式：

    1. 若对于某常数 ε>0，有 $f(n) = O(n^{\log_b a - \varepsilon})$，则 $T(n) = \theta(n^{\log_b a})$；
    2. 若 $f(n) = \theta(n^{\log_b a})$，则 $T(n) = \theta(n^{\log_b a} \cdot \log n)$；
    3. 若对其常数 ε>0，有 $f(n) = \Omega(n^{\log_b a + \varepsilon})$ 且对于某常数 c>1 和所有充分大的正整数 n 有 af(n/b)≤cf(n)，则 T(n)=θ(f(n))。

第 5 章：概率分析与随机算法

1：随机算法：如果一个算法的行为不只是由输入决定，同时也由随机数生成器所产生的数值决定，则称这个算法是随机的。

2：指示器随机变量 I(A)的定义很简单：

$$I(A) = \begin{cases} 0 & \text{如果 A 不发生的话} \\ 1 & \text{如果 A 发生的话} \end{cases}$$

事件 A 对应的指示器随机变量的期望期等于事件 A 发生的概率。

3：介绍了两种随机排列数组的生成方法：

1. 随机优先级法：为数组的每个元素赋一个随机的优先级，再根据这个优先级对数组中的元素进行排序。可证这样得到的数字满足随机的性质。
2. 原地交换法：依次把 A[i]与 A[Random(i+1, Length(A))]进行 swap，得到的新数组也满足随机性。

```
for i ← 1 to n
    do swap A[i] ↔ A[Random(i, Length(A))]
```

4：有时间，在真正的环境中的输入可能并不是随机的，所以我们可以采用先将输入进行随机打乱的方法来保证输入数据的随机性，这点在很多算法中得以体现，比如快排有其随机选取种子数来向输入中加入随机化的成分。

5：===5.4 节带*未看===

不过值得提一下的是“在线雇佣问题”与“苏格拉底的择偶观”很相似。

先用三分之一的时间，即分出大、中、小三类，再用三分之一的时间验证自己的观点是否正确，等到最后三分之一时，选择了属于大类中的一支美丽的麦穗。

第二部分：排序和顺序统计学

1：这一部分将要给出几个解决以下排序问题的算法：

- 输入：n 个数的序列<a_1,a_2, … a_n>
- 输出：输入序列的一个重排<a_1',a_2',...,a_n'>，使 $a_1' \leqq a_2' \leqq ... \leqq a_n'$

2：原地排序算法：只有线性个数的元素会被移动到集合之外的排序算法。

3：第 6 章介绍堆排序

4：第 7 章介绍快速排序

5：第 8 章介绍了基于“比较”排序的算法的下界为 Ω(nlgn)。并介绍了几种不基于比较的排序方法，它们能突破 Ω(nlgn)的下界。计数排序、基数排序、桶排序。

6：第 9 章介绍了顺序统计的概念：第 i 个顺序统计是集合中第 i 小的数。并介绍了两个算法：

- 最坏情况为 O(n^2)，但平均情况下为线性 O(n)的算法
- 最坏情况下为线性 O(n)的算法

第 6 章：堆排序

1：堆排序是一个时间复杂度为 O(nlgn)、原地排序算法。

2：“堆”数据结构不只在推排序时有用，还可以构成一个有效的优先队列（堆的主要应用之一）。

3：堆的定义是这样的：

- 一个堆是一颗完全二叉树对于大（小）根堆
- 每个节点的值都比它的子节点要大（小）

4：虽然堆排的理论效率好，但是往往一个好的快排的实现要优于堆排。

5：所以堆更常见于作为高效的优先级队列：一个堆可以在 O(lgn)的时间内，支持大小为 n 的集合上的任意优先队列的操作。

Output（代码：http://code.google.com/p/introduction-to-algorithms-notes/）：

```
原始数组，准备进行堆排序：8  5  78  45  64  987  45  34  23  4  23
堆排序结束：4  5  8  23  23  34  45  45  64  78  987

初始化一个优先队列：41  169  334  358  464  724  478  500  962  467
开始不断的取最高优先级的任务出列：
41:     169  358  334  467  464  724  478  500  962
169:    334  358  478  467  464  724  962  500
334:    358  464  478  467  500  724  962
358:    464  467  478  962  500  724
464:    467  500  478  962  724
467:    478  500  724  962
478:    500  962  724
```

```
500:    724  962
724:    962
962:
开始添加任务入列：
4
4  5
4  5  8
4  5  8  23
4  5  8  23  23
4  5  8  23  23  34
4  5  8  23  23  34  45
4  5  8  23  23  34  45  45
4  5  8  23  23  34  45  45  64
4  5  8  23  23  34  45  45  64  78
4  5  8  23  23  34  45  45  64  78  987
4  5  8  23  23  34  45  45  64  78  987
请按任意键继续. . .
```

第 7 章：快速排序

1：快速排序的最坏运行时间为 $O(n^2)$，期望运行时间为 O(nlgn)，且由于 O(nlgn)中隐含的常数因子很小，所以快排通常是用于排序的最佳的实用选择（因为其平均性能非常好）。
快排真的太棒了：平均性能非常好、原地排序不需要额外的空间、算法简单只需要寥寥几行就搞定（比冒泡还少）。
对 10W 个随机数进行排序比较，快排平均在 600MS，而堆排平均在 900MS，性能差距可见一斑啊。

2：快排的平均情况运行时间与其最佳情况运行时间很接近，而不是非常接近于其最坏情况运行时间，所以一般来说快排效率是最高的，这是快排在现代得以大规模的使用的根本原因。

3：快排很需要随机化技术：因为在真正的应用时很容易出现待排序的数组其实已经是有序的情况，而这种已经有序的情况却正好又是快排算法的软肋，它在待排数组有序时的效率是最差的 O(n^2)，所以很需要随机化技术！

4：快速排序的随机化版本：正如第 5 章所说的，由于工程中的输入可能不随机的，所以我们要将其随机化。有两种可选方案
（1）直接对输入数据进行随机化排列
（2）采用随机取样的随机化技术。
随机取样的效率更高一些，所以在快速排序的随机化版本中采用随机取样的技术。
方法很简单，就是在每趟 sort 之前随机选取一个数与最末尾的元素进行交换操作，这样简单高效的实现了随机化。

```
//加入随机取样的随机化技术
intrandom_swap = (rand() % (EndIndex - BeginIndex + 1)) + BeginIndex;
std::swap(ToSort[random_swap], ToSort[EndIndex]);
```

这个技术太有用啊，因为快速排序在输入数据已经有序时的性能是最差的，但是输入数据已经有序的情况又会经常发生，所以这个随机取样就显得异常的重要。如果没有这个随机取样，快排绝得不到这样的应用。
在我做的实验中，对 2000 个有序的数据进行排序，在未没采用随机化的情况下，平均耗时 860MS，而使用了随机取样之后平均耗时 8MS，效率提高了 100 倍。由于输入有序的情况是非常常见的，所以这个随机化才更显示重要了！

Output（代码：http://code.google.com/p/introduction-to-algorithms-notes/）：

```
==========================快速排序============================
随机填充 100 个数：
41   18467   6334   26500   19169   15724   11478   29358   26962   24464   5705   28145   2
```

```
3281   16827   9961   491   2995   11942   4827   5436   32391   14604   3902   153   292   1
2382   17421   18716   19718   19895   5447   21726   14771   11538   1869   19912   25667
 26299   17035   9894   28703   23811   31322   30333   17673   4664   15141   7711   28253
  6868   25547   27644   32662   32757   20037   12859   8723   9741   27529   778   12316
 3035   22190   1842   288   30106   9040   8942   19264   22648   27446   23805   15890   6
729   24370   15350   15006   31101   24393   3548   19629   12623   24084   19954   18756
 11840   4966   7376   13931   26308   16944   32439   24626   11323   5537   21538   16118
  2082   22929   16541
快速排序的结果如下：
41   153   288   292   491   778   1842   1869   2082   2995   3035   3548   3902   4664   482
7   4966   5436   5447   5537   5705   6334   6729   6868   7376   7711   8723   8942   9040
 9741   9894   9961   11323   11478   11538   11840   11942   12316   12382   12623   12859
  13931   14604   14771   15006   15141   15350   15724   15890   16118   16541   16827   1
6944   17035   17421   17673   18467   18716   18756   19169   19264   19629   19718   1989
5   19912   19954   20037   21538   21726   22190   22648   22929   23281   23805   23811
24084   24370   24393   24464   24626   25547   25667   26299   26308   26500   26962   274
46   27529   27644   28145   28253   28703   29358   30106   30333   31101   31322   32391
 32439   32662   32757
======================模糊区间的快速排序========================
41        -->     43
21        -->     69
34        -->     87
36        -->     110
20        -->     116
83        -->     178
99        -->     167
84        -->     165
50        -->     141
99        -->     117
请按任意键继续. . .
```

第 8 章：线性时间排序

1. 任何比较的排序在最坏的情况下都要用 Ω(nlgn)次比较来进行排序，所以合并排序和堆排序是渐近最优的。
   注意的是：快排不是渐近最优的，因为它在最坏的情况下是 $O(n^2)$。
2. 三种以线性时间运行的排序算法：计数排序、基数排序和桶排序。它们都是非比较的。
3. 计数排序
   a) 计数排序的一个重要性就是它是稳定的排序算法，这个稳定性是基数排序的基石。
   b) 计数排序的想法真的很简单、高效、可靠
   c) 缺点在于：
      i. 需要很多额外的空间（当前类型的值的范围）
      ii. 只能对离散的类型有效比如 int（double 就不行了）
      iii. 基于假设：输入是小范围内的整数构成的。
4. 基数排序
   a) 基数排序时对每一维进行调用子排序算法时要求这个子排序算法必须是稳定的。
   b) 基数排序与直觉相反：它是按照从底位到高位的顺序排序的。
      我觉得原因在于：高有效位对底有效位有着决定性的作用。
5. 桶排序
   a) 桶排序也只是期望运行时间能达到线性，对于最坏的情况，它的运行时间取决于它内部使用的子排序算法的运行时间，一般为 O(nlgn)。

b) 桶排序基于假设：输入的的元素均匀的分布在区间[0, 1]上。

c) 感觉桶排没有什么大的实现价值，因为它限定了输入的区间，还要求最好是均匀分布，它的最坏情况并不好。

6. 所有的线性时间内的排序算法，都作出了一定的假设，是建立在一定的假设基础上的。

Output（代码：http://code.google.com/p/introduction-to-algorithms-notes/）：

```
==========开始计数排序==========
0  1  2  3  4  5  5  6  6  8  11  11  12  16  16  18  18  21  22  23  23  23  24
  26  26  27  27  29  29  29  29  31  33  34  35  35  36  37  37  38  38  39  40
  40  41  41  41  41  42  42  42  44  44  45  46  47  47  48  48  50  53  53  54
  56  57  58  59  61  62  62  64  64  64  66  67  67  68  69  69  70  71  73  76
  78  78  81  82  82  84  88  90  90  91  91  92  93  94  95  95  99
==========开始基数排序==========
833  115  639  658  704  930  977  306  673  386
115  306  386  639  658  673  704  833  930  977
==========开始桶排序==========
0.21  0.45  0.24  0.72  0.7  0.29  0.77  0.73  0.97  0.12
0.12  0.21  0.24  0.29  0.45  0.7  0.72  0.73  0.77  0.97
请按任意键继续. . .
```

第 9 章：中位数和顺序统计学

1. 第 i 个顺序统计量是该集合中第 i 小的元素。
   最小值是第 1 个顺序统计量(i=1)最大值是第 n 个顺序统计量(i=n)
2. 中位数是它所在集合的“中点元素”
3. 找最大最小值的算法，一般人可能以为需要 2n 次比较，实际上只需要最多 $3\lfloor \frac{n}{2} \rfloor$次比较，使用的技巧是：

   将一对元素比较，然后把较大者于 max 比较，较小者与 min 比较，这样就只需要 $3\lfloor \frac{n}{2} \rfloor$次比较就能得到最后的结果。
4. 以期望线性时间选择顺序统计量的方法是以快速排序为模型。如同在快速排序中一样，此算法的思想也是对输入数组进行递归划分。但和快速排序不同的是，快速排序会递归处理划分的两边，而 randomized-select 只处理划分的一边。并由此将期望的运行时间由 O(nlgn)下降到了 O(n)。
   这就是顺序统计量算法能够如此高效的核心原因所在！
   我觉得 C++ STL 中的 nth_element 用的可能就是这个算法，所以它的效率应该很高。
5. 最坏线性时间选择顺序统计量的方法的核心在于：要保证对数组的划分是一个好的划分。
   于是方法使用了一个很奇怪的取主元的方法，虽然看起来很奇怪，但是该方法被这样的提出就肯定有它的理论基础的。
   不过这种取巧的方法不太值得去写一遍，而且明显写出来也很容易的，没有什么新技术和新想法。

Output（代码：http://code.google.com/p/introduction-to-algorithms-notes/）：

```
41  467  334  500  169  724  478  358  962  464
0th element is:41
1th element is:169
2th element is:334
3th element is:358
4th element is:464
5th element is:467
6th element is:478
7th element is:500
8th element is:724
```

```
9th element is:962
请按任意键继续. . .
```

第三部分：数据结构

第 10 章：基本数据结构

没发现有什么值得看的，貌似就是下面这些基本的知识，这些知识都不知道就没法混啦。

1. 栈
2. 队列
3. 链表
4. 树的“左孩子、右兄弟”表示法

第 11 章：散列表

1. 在散列表中查找一个元素的时间与在链表中查找一个元素的时候相同，在最坏情况为 O(n)，但期望时间为 O(1)。
   在实践中，散列表的效率是很高的，一般可认为是 O(1)。
2. 散列是一种极其有效和实用的技术，基本的字典操作只需要 O(1)的平均时间。
3. 当待排序的关键字集合是静态的（即当关键字集合一旦存入后不再改变），“完全散列能够在 O(1)的最坏情况时间内支持关键字查找。
4. 散列是一种极其有效和实用的技术：基本的字典操作只需要 O(1)的平均时间。而且当待排序的关键字的集合是静态的（即当关键字集合一旦存入后不需要再改变），“完全散列”能够在 O(1)的最坏时间内支持查找操作。
5. 在众多的简单的解决碰撞的方法中，我觉得比较好的是通过链表法解决碰撞，虽然这个方法的理论最坏效率为 O(n)，但是在平均情况下，它的性能也是非常好的，实现简单又高效。
6. 装载因子：给定一个能存放 n 个元素的、具有 m 个槽位的散列表 T，定义 T 的装载因子 $\alpha = n/m$，即一个链中平均存储的元素数。
7. 多数的散列函数都假定关键字域为自然数集 N，如果所给关键字不是自然数，则必须有一种方法来将它们解释为自然数。
   a) 除法散列法：$h(k) = k \bmod m$
      一般选取 m 的值为与 2 的整数幂不大接近的质数
   b) 乘法散列法：$h(k) = \lfloor m(kA \bmod 1) \rfloor$
      构造散列函数的乘法方法包含两个步骤：首先用关键字剩上常数 A(0<A<1)，并抽取 kA 的小数部分；然后用 m 剩以这个值，再取结果的底。

      Knuth 认为$A \approx \frac{(\sqrt{5}-1)}{2}$是一个比较理想的值。
   c) 全域散列：全域散列的基本思想是在执行开始时，就从一族仔细设计的函数中，随机地选择一个作为散列函数。
      i. 首先：全域散列表是一种使用“键接法”来解决碰撞问题的散列表方法。
      ii. 随机化保证了对于任何输入，算法都具有较好的平均性能。
      iii. 全域的散列函数组：设 H 为一组散列函数，它将给定的关键字域 U 映射到{0,1,…,m-1}中，这样的一个函数组称为是全域的。如果从 H 中随机地选择一个散列函数，当关键字 K≠J 时，两者发生碰撞的概率不大于 1/m。
      iv. 常用的一个全域散列函数类：
      首先选择一个足够大的质数 p ，使得每一个可能的关键字 k 都落到 0 到 p-1 的范围内，包括首尾的 0 和 p-1。这里我们假设全域是 0 – 15，p 为 17。设集合 Zp 为{0, 1, 2, …, p-1}，集合 Zp* 为{1, 2, 3, …, p-1}。由于 p 是质数，我们可以定义散列函数 h(a, b, k) = ((a*k + b) mod p) mod m。其中 a 属于 Zp，b 属于 Zp*。由所有这样的 a 和 b 构成的散列函数，组成了函数簇。即全域散列。
      v. 明白这个散列函数的选取是在“执行开始”随机的选取一个是很重要的，要不然就会不明白到时候怎么进行查找。这里所谓的随机性应该这样理解：对于某一个散列表来说，它在初始化时已经把 a,b 固定了，但是对于一个还未初

始化的全域散列表来说，a,b 是随机选取的。

8. 开放寻址法：所有的元素都放在散列表里
   a) 开放寻址法的好处就在于它根本不用指针，而是计算出要存取的各个槽。这样一来，由于不用存储指针就节省了空间，从而可以用同样的空间来提供更多的槽，其潜在的效果就是可以减少碰撞，提高查找速度。
   b) 感觉上开放寻址法很像一个启发式的搜索，它的最坏性能也是 O(n)，只不过散列函数为它提供了启发信息从而使得一般的平均性能会很好。
   c) 在开放寻址法中，对散列元素的删除操作执行起来比较困难，因为删除操作会影响查找操作。解决办法是在槽里的值被删除后置一个特定的值 DELETED，而不是删除后不管，查找的时候处理一下就可以了
   d) 线性探查、二次探查和双重散列都是对最基本的数组法的改进，虽然它们很漂亮，但是思想上并没有太大的革新，看起来很容易懂的。
   e) 双重散钱是用于开放寻址的最好方法之一，因为它所产生的排列具有随机选择的排列的许多特性。
9. 完全散列：如果某一种散列技术在进行查找时，其最坏情况内存访问次数为 O(1)的话，则称其为完全散列。
   书上使用了一种两级的散列方案，每一级都采用全域散列。
   通常利用一种两级的散列方案，每一级上都采用全域散列。为了确保在第二级上不出现碰撞，需要让第二级散列表 Sj 的大小 mj 为散列到槽 j 中的关键字数 nj 的平方。如果利用从某一全域散列函数类中随机选出的散列函数 h，来将 n 个关键字存储到一个大小为 m=n 的散列表中，并将每个二次散列表的大小置为 mj=nj2 (j=0, 1, …, m-1)，则在一个完全散列方案中，存储所有二次散列表所需的存储总量的期望值小于 2n。
   完全散列的关键在于：二次散列表中要求没有碰撞。这是通过确保槽的个数是关键字的个数的平方来实现的。

Output（代码：http://code.google.com/p/introduction-to-algorithms-notes/）：

```
开始往 UniversalHashTable 里添加内容[0,100)：
槽[  0  ] -> 98 -> 92 -> 79 -> 73 -> 60 -> 54 -> 41 -> 35 -> 22 -> 16 -> 10
槽[  1  ] -> 99 -> 86 -> 80 -> 67 -> 61 -> 48 -> 42 -> 29 -> 23 -> 4
槽[  2  ] -> 93 -> 87 -> 74 -> 68 -> 55 -> 49 -> 36 -> 30 -> 17 -> 11
槽[  3  ] -> 81 -> 75 -> 62 -> 56 -> 43 -> 37 -> 24 -> 18 -> 5
槽[  4  ] -> 94 -> 88 -> 69 -> 63 -> 50 -> 44 -> 31 -> 25 -> 12 -> 6
槽[  5  ] -> 95 -> 82 -> 76 -> 57 -> 51 -> 38 -> 32 -> 19 -> 13 -> 0
槽[  6  ] -> 89 -> 83 -> 70 -> 64 -> 45 -> 39 -> 26 -> 20 -> 7 -> 1
槽[  7  ] -> 96 -> 90 -> 77 -> 71 -> 58 -> 52 -> 33 -> 27 -> 14 -> 8
槽[  8  ] -> 97 -> 84 -> 78 -> 65 -> 59 -> 46 -> 40 -> 21 -> 15 -> 2
槽[  9  ] -> 91 -> 85 -> 72 -> 66 -> 53 -> 47 -> 34 -> 28 -> 9 -> 3
开始删除内容[0,5)：
槽[  0  ] -> 98 -> 92 -> 79 -> 73 -> 60 -> 54 -> 41 -> 35 -> 22 -> 16 -> 10
槽[  1  ] -> 99 -> 86 -> 80 -> 67 -> 61 -> 48 -> 42 -> 29 -> 23
槽[  2  ] -> 93 -> 87 -> 74 -> 68 -> 55 -> 49 -> 36 -> 30 -> 17 -> 11
槽[  3  ] -> 81 -> 75 -> 62 -> 56 -> 43 -> 37 -> 24 -> 18 -> 5
槽[  4  ] -> 94 -> 88 -> 69 -> 63 -> 50 -> 44 -> 31 -> 25 -> 12 -> 6
槽[  5  ] -> 95 -> 82 -> 76 -> 57 -> 51 -> 38 -> 32 -> 19 -> 13
槽[  6  ] -> 89 -> 83 -> 70 -> 64 -> 45 -> 39 -> 26 -> 20 -> 7
槽[  7  ] -> 96 -> 90 -> 77 -> 71 -> 58 -> 52 -> 33 -> 27 -> 14 -> 8
槽[  8  ] -> 97 -> 84 -> 78 -> 65 -> 59 -> 46 -> 40 -> 21 -> 15
槽[  9  ] -> 91 -> 85 -> 72 -> 66 -> 53 -> 47 -> 34 -> 28 -> 9
开始检索结点[0]：未找到
开始检索结点[1]：未找到
开始检索结点[2]：未找到
开始检索结点[3]：未找到
开始检索结点[4]：未找到
开始检索结点[5]：5
开始检索结点[6]：6
开始检索结点[7]：7
开始检索结点[8]：8
```

```
开始检索结点[9]：9
请按任意键继续. . .
```

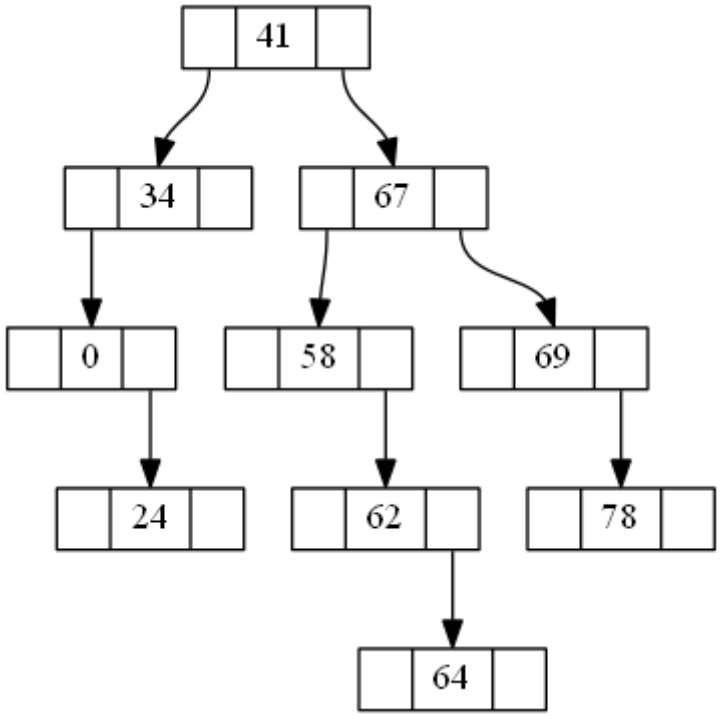
第 12 章：二叉查找树

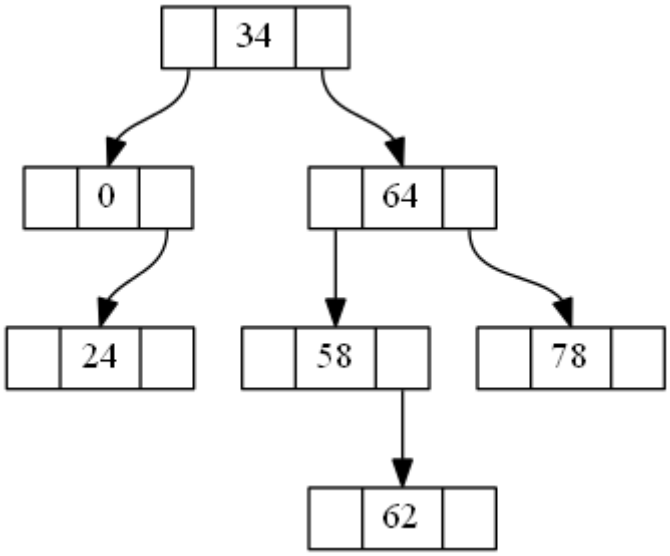
1. 二叉查找树的定义：对任何结点 X，其左子树中的关键字最大不超过 key[X]；其右子树中的关键字最小不小于 key[x]。
2. 首先先明显：二叉查找树上的基本操作的时间都与树的高度成正比的，所以高度越小的树性能越高。
3. 查询二叉查找树可以考虑使用非递归的版本，它运行要快得多而且也很容易理解

```
SEARCH(x, k):
while (x != NULL && x.Key != k){
    if (k < x.Key) x = x.Left;
    else x = x.Right;
}
return x;
```

4. 前趋和后继：
   前趋：左一次，然后右到头；
   后继：右一次，然后左到头。
   错！不止这么简单，以后继为例：当结点的右子树不存在时，应该一路向上传递，直到找到根结点（没有后继）或者是找到一次非右子树传递（后继找到）为止。我的代码就在这里犯一次错误了，本以为很简单的！
5. 对二叉查找树的插入和删除操作也不复杂，唯一有点难度的地方就是在删除同时存在左右子树的结点时需要进行一些处理。书上叙述的有点过度的复杂，其实可以很简单地说明白：对于这样的结点 x，找到 x 结点的前趋（或后继）y，将 x 的值替换为 y 的值，然后递归删除 y 结点就可以了。因为 y 一定没有右子树（后继对应没有左子树），所以递归删除的时候就是很简单的情况了。
6. 可以证明：随机构造的二叉村在平均情况下的行为更接近于最佳情况下的行为，而不是接近最坏情况下的行为。所以一棵在 n 个关键字上随机构造的二叉查找树的期望高度为 O(lgn)。

Output（代码：http://code.google.com/p/introduction-to-algorithms-notes/）：





Deleted [41]

Deleted [67]

Deleted [69]

搜索[0]元素：　成功

搜索[24]元素：　成功

搜索[34]元素：　成功

搜索[58]元素：　成功

搜索[62]元素：　成功

搜索[64]元素：　成功

搜索[78]元素：　成功

请按任意键继续. . .

第 13 章：红黑树

满足下面几个条件(红黑性质)的二叉搜索树，称为**红黑树**：

1. 每个结点或是红色，或是是黑色。
2. 根结点是黑的。
3. 所有的叶结点(NULL)是黑色的。（NULL 被视为一个哨兵结点，所有应该指向 NULL 的指针，都看成指向了 NULL 结点。）
4. 如果一个结点是红色的，则它的两个儿子节点都是黑色的。
5. 对每个结点，从该结点到其子孙结点的所有路径上包含相同数目的黑结点。

1. 黑高度的定义： 从某个结点出发(不包括该结点)到达一个叶结点的任意一条路径上，黑色结点的个数成为该结点 x 的黑高度。红黑树的黑高度定义为其根结点的黑高度。
2. 红黑树是真正的在实际中得到大量应用的复杂数据结构：C++STL 中的关联容器 map,set 都是红黑树的应用（所以标准库容器的效率太好了，能用标准库容器尽量使用标准库容器）；Linux 内核中的用户态地址空间管理也使用了红黑树。
3. 红黑权势是许多“平衡的”查找树中的一种（首先：红黑树是一种近似平衡的二叉树），它能保证在最坏的情况下，基本的动态集合操作的时间为 O(lgn)。
4. 通过对任何一条从根到叶子的路径上各个结点着色方式的限制，红黑树确保没有一条路径会比其它路径长出两倍，因而是接近平衡的。
5. 一要全是黑结点的满二叉树也满足红黑树的定义。满二叉树的效率本身就非常高啊，它是效率最好的二叉树了，所以说它是红黑树的一个特例；普通的红黑树要求并没有满二叉树这么严格。
6. 旋转操作（左旋和右旋）：旋转操作是一种能保持二叉查找树性质的查找树局部操作。
   所有对红黑树结构的修改都只能通过左右旋来完成，这样才能保证修改后的红黑树首先是一棵二叉查找树。
7. 红黑树的插入操作：将结点 Z 插入树 T 中，就好像 T 是一棵普通的二叉查找树一样，然后将 Z 着为红色。为保证红黑性质

能继续保持，我们调用一个辅助程序来对结点重新着色并旋转。

这么做是有它的智慧的：首先，插入结点 Z 的位置的确应该和普通二叉查找树一样，因为红黑树本身就首先是一棵二叉查找树；然后将 Z 着为红色，是为了保证性质 5 的正确性，因为性质 5 如果被破坏了是最难以恢复的；到这里，有可能被破坏的性质就只剩下性质 2 和性质 4 了，这都可以通过后来的辅助程序进行修复的。

插入操作可能破坏的性质：

a) 性质 2：当被一棵空树进行插入操作时发生；

b) 性质 4：当新结点被插入到红色结点之后时发生；

8. 红黑树的删除操作：和插入操作一样，先用 BST 的删除结点操作，然后调用相应的辅助函数做相应的调整。

首先只有被删除的结点为黑结点时才需要进行修补，理由如下：

a) 树中各结点的黑高度都没有变化

b) 不存在两个相邻的红色结点

c) 因为如果被删除的点是红色，就不可能是根，所以根仍然是黑色的

9. 当被删除了黑结点之后，红黑树的性质 5 被破坏，上面说过了性质 5 被破坏后的修复难度是最大的。所以这里的修复过程使用了一个很新的思想，即视为被删除的结点的子结点有额外的一种黑色，当这一重额外的黑色存在之后，性质 5 就得到了继续。然后再通过转移的方法逐步把这一重额外的黑色逐渐向上转移直到根或者红色的结点，最后消除这一重额外的黑色。

删除操作中可能被破坏的性质：

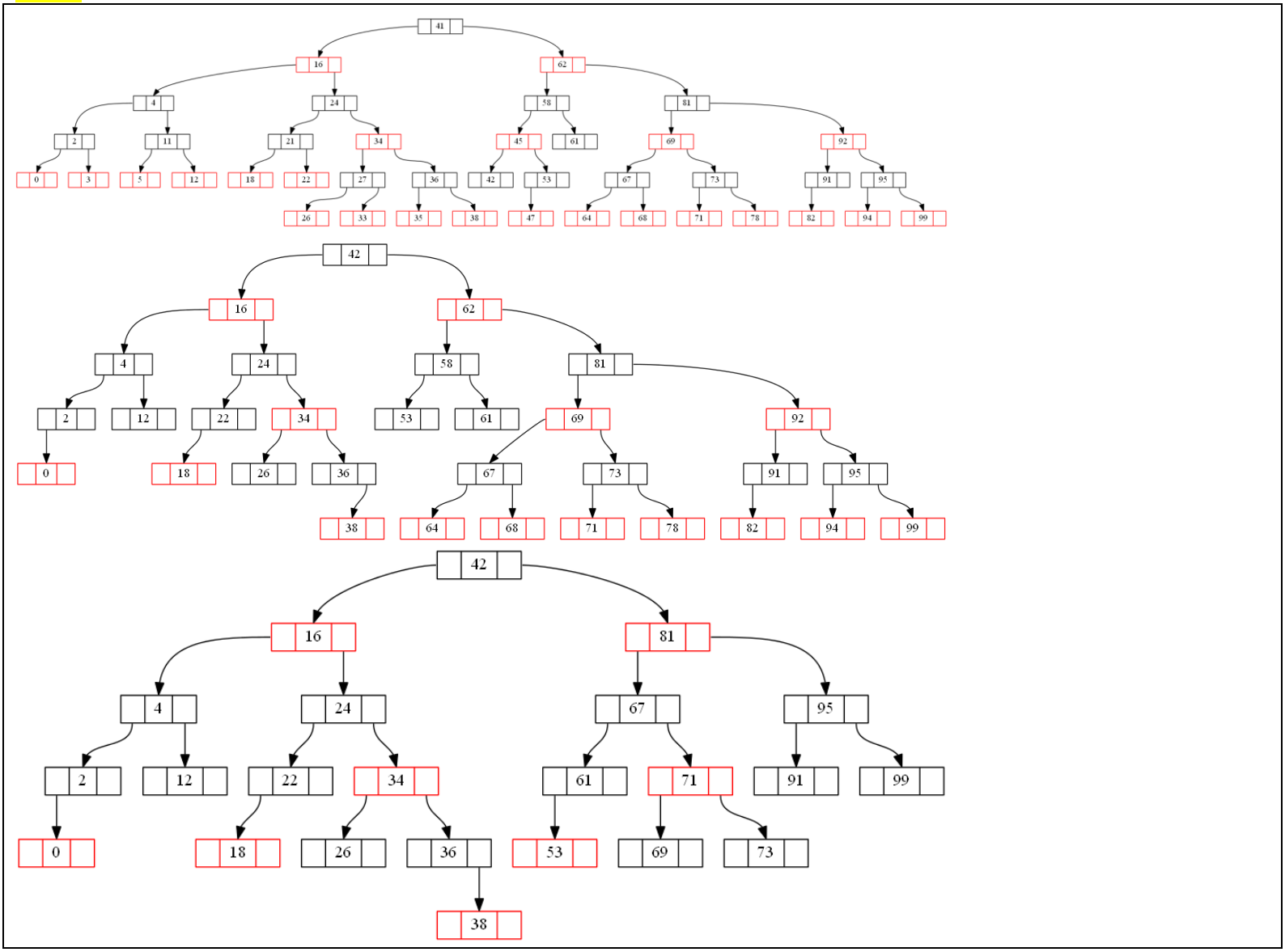
a) 性质 2：当 y 是根时，且 y 的一个孩子是红色，若此时这个孩子成为根结点；

b) 性质 4：当 x 和 p[y]都是红色时；

c) 性质 5：包含 y 的路径中，黑高度都减少了；

10. 红黑树是平衡查找树，还有 B 树也是另一类平衡查找树。

Output（代码：http://code.google.com/p/introduction-to-algorithms-notes/）：



Deleted [3]

Deleted [5]

Deleted [11]

Deleted [21]

Deleted [27]

Deleted [33]

Deleted [35]

Deleted [41]

Deleted [45]

Deleted [47]

Deleted [58]

Deleted [62]

Deleted [64]

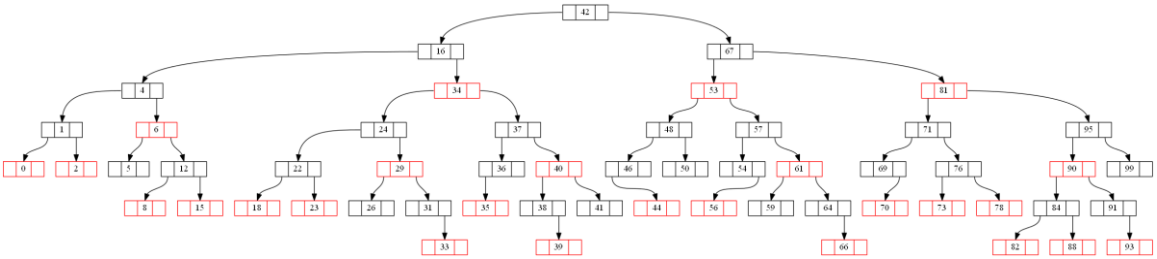
Deleted [68]

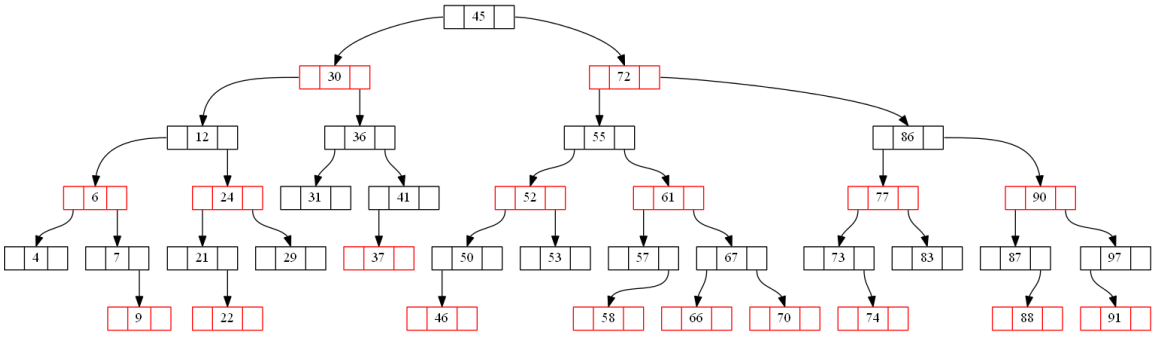
Deleted [78]

Deleted [82]

Deleted [92]

Deleted [94]





Deleted [0]

Deleted [1]

Deleted [2]

Deleted [4]

Deleted [5]

Deleted [6]

Deleted [8]

Deleted [12]

Deleted [15]

Deleted [16]

Deleted [18]

Deleted [22]

Deleted [23]

Deleted [24]

Deleted [26]

Deleted [29]

Deleted [31]

Deleted [33]

Deleted [34]

```
Deleted [35]
Deleted [36]
Deleted [37]
Deleted [38]
Deleted [39]
Deleted [40]
Deleted [41]
Deleted [42]
Deleted [44]
Deleted [46]
Deleted [48]
Deleted [50]
Deleted [53]
Deleted [54]
Deleted [56]
Deleted [57]
Deleted [59]
Deleted [61]
Deleted [64]
Deleted [66]
Deleted [67]
Deleted [69]
Deleted [70]
Deleted [71]
Deleted [73]
Deleted [76]
Deleted [78]
Deleted [81]
Deleted [82]
Deleted [84]
Deleted [88]
Deleted [90]
Deleted [91]
Deleted [93]
Deleted [95]
Deleted [99]
请按任意键继续. . .
```

第 14 章：数据结构的扩张

1. 实际的工程中，极少会去创造新的数据结构，通常是对标准的数据结构附加一些信息，并添加一些新的操作以支持应用的要求。
2. 数据结构的扩张：指在实际应用数据结构时对标准的数据结构中增加一些信息、编入一些新的操作等等。附加的信息必须能够为该数据结构上的常规操作所更新和维护。
3. 对一种数据结构的扩张过程可以分为四个步骤：
   a) 选择基础的数据结构
   b) 确定要在基础数据结构中添加哪些信息
   c) 验证可用基础数据结构上的基本修改操作来维护这些新添加的信息
   d) 设计新的操作

4. 红黑树的扩张定理：当结点中新添加的信息可以由该结点和它的左右子树来决定，那么就可以在不影响时间复杂度的前提下在插入和删除等操作中对红黑树的这些附加信息进行维护。

第四部分：高级设计和分析技术

设计和分析高效算法的三种重要技术：动态规划、贪心算法和平摊分析。
动态规划通常应用于最优化问题，即要做出一组选择以达到一个最优解时。在做选择的同时，经常出现同样形式的子问题。关键技术是存储这些子问题每一个的解，以备它重复出现。
贪心算法通常也是应用于最优化问题，该算法的思想是以局部最优的方式来做每一个选择。采用贪心算法可以比动态规划更快的得出一个最优解，但是关键是不容易判断贪心算法所得到的是否真的是最优解。
贪心算法可以在一定的理论之下，通过只考虑局部最优解就可以保证得到的一定是全局最优解。如赫夫曼编码。
平摊分析是一种用来分析执行一系列类似操作的算法的工具。在一个操作序列中，不可能每一个都以其已知的最坏情况运行，某些操作的代价高些，而其它的低一些。

所有的最优化问题都可以通过穷举法来解决，但是这在时间上是不可接受的。
所有的高效算法都是为了加快速度：
1. 动态规划：保存子问题的解，以备重复使用
2. 贪心算法：用局部最优解来求全局最优解

平摊分析是用来分析算法的工具，它本身并不是一种算法。

第 15 章：动态规划

1. 动态规划与分治法之间的区别：
   a) 分治法是指将问题分成一些独立的子问题，递归的求解各子问题
   b) 动态规划适用于这些子问题不是独立的情况，也就是各子问题包含公共子问题
2. 动态规划算法的设计可以分为 4 个步骤：
   a) 描述最优解的子结构
   b) 递归定义最优解的值
   c) 按自底向上的方法计算最优解的值
   d) 由计算出的结果反向构造出一个最优解
3. 动态规划最最最重要的就是要找出最优解的子结构！
4. 最优子结构在问题域中以两种方式变化（在找出这两个问题的解之后，构造出原问题的最优子结构往往就不是难事了）：
   a) 有多少个子问题被用在原问题的一个最优解中
   b) 在决定一个最优解中使用哪些子问题有多少个选择
5. 动态规划说白了就是一个递归的反向展开的过程：在满足①最优子结构②重叠子问题这 2 个条件下，通过把递归从下至上的进行展开以避免重复计算子问题从而加速了最终问题的求解的过程。
6. 再次强调“动态规划最关键的一步就是：寻找最优子结构”
7. 动态规划能够消除重复计算子问题是因为它与普通递归相反，它是通过自下而上的方式来进行求解的。
8. 正确使用动态规划方法的 2 个关键要素：最优子结构 和 重叠子问题。
   如果问题可以由递归来解决，并且在递归的过程中会不断的出现重复的子问题需要解决，那毫不犹豫的采用动态规划吧！
9. 如果问题的一个最优解中包含了子问题的最优解，则该问题具有最优子结构；而当一个问题具有最优子结构时，提示我们动态规划可能会适用。
10. 剪贴技术：用来证明在问题的一个最优解中，使用的子问题的解本身也必须是最优的
11. 为了描述子问题空间，可以遵循这样一条有效的经验规则，就是尽量保持这个空间简单，然后在需要时再扩充它
12. 非正式地：一个动态规划算法的运行时间依赖于两个因素的乘积：子问题的总个数和每个子问题中有多少种选择。
    问题解的代价通常是子问题的代价加上选择本身带来的开销。
13. 贪心算法与动态规划有一个显著的区别：就是在贪心算法中，是以自顶向下的方式使用最优子结构的。贪心算法会先做选择，在当时看起来是最优的选择，然后再求解一个结果子问题，而不是先寻找子问题的最优解，然后再选择。

14. 要注意：在不能应用最优子结构的时间，就一定不能假设它能够应用。
    警惕使用动态规划去解决缺乏最优子结构的问题！
15. 使用动态规划时：子问题必须是相互独立的！可以这样理解，N 个子问题域互不相干，属于完全不同的空间。
    子问题必须独立！
16. 重叠子问题：不同的子问题的数目是输入规模的一个多项式。
    这样，动态规划算法才能充分利用重叠的子问题，减少计算量。即通过每个子问题只解一次，把解保存在一个需要时就可以查看的表中，而每次查表只需要常数时间。
    从这段描述可以看出：动态规划与递归时做备忘录的本质是完全相同的，所以说备忘录方法与普通的动态递归本质完全相同，没有孰优孰劣之分，哪个方便用哪个。
17. 由计算出的结果反向构造一个最优解：把动态规划或者是递归过程中作出的每一次选择（记住：保存的是每次作出的选择）都保存下来，在最后就一定可以通过这些保存的选择来反向构造出最优解。
18. 做**备忘录**的递归方法：这种方法是动态规划的一个变形，它本质上与动态规划是一样的，但是比动态规划更好理解！
    a) 使用普通的递归结构，自上而下的解决问题。
    b) 当在递归算法的执行中每一次遇到一个子问题时，就计算它的解并填入一个表中。以后每次遇到该子问题时，只要查看并返回表中先前填入的值即可。
19. 备忘录方法与动态递归方法的比较：如果所有的子问题都至少要被计算一次，则一个自底向上的动态规划算法通常比一个自顶向下的做备忘录算法好出一个常数因子。因为动态规划没有使用递归的代价，只用到了循环，所以常数因子肯定比递归要好一些。
    此外，在有些问题中，还可以用动态规划算法中的表存取模式来进一步的减少时间和空间上的需求；或者，如果子问题中的某些子问题根本没有必须求解，做备忘录的方法有着只解那些肯定要求解的子问题的优点。（而且这点是自动获得的，那些不必要计算的子问题在备忘录方法中会被自动的抛弃）
20. 备忘录方法总结：由“是否所有的子问题都至少需要被计算一次”来决定使用动态规划还是备忘录。
    再次下定义：这两种方法没孰优孰劣之分，因为它们的本质思想是完全一样的；消除重复子问题。
21. 动态规划：最重要最重要的就是找到最优子结构。在找到最优子结构之后的消除重复子问题，这点我太容易处理了，无论是动态规划的自底向上的递推，还是备忘录，或者是备忘录的变型，都可以轻松的应付。关键就是最优子结构。

Output（代码：http://code.google.com/p/introduction-to-algorithms-notes/）：

```
最短的装配路线需要时间：38
0 --> 1 --> 0 --> 1 --> 1 --> 0 -->
15125
((A0(A1A2))((A3A4)A5))
12
bdaadcddbaba
12
12
2.75

hq h meayl lfdxfi cvscxggb kfnqdux fnf zvsrt       |
jprep gxrpn vy tmwcysy cqpev k ffmznim kasvwsr    |
nzkycxf tlsg p fadpooe xzbcoejuvp aboygp eylfp    |
npljvrvi ya yehwqnq qpmxuj loovao u wh sncbxcoks|
zkvatxdk lyjyhfix swnkkufnux zrzbmn gqooketly      |
nkoaugzqrc di teiojwa yzp scmpsajlfv ubfaaov       |
zylntrkdcp srtesjwhd zcobzcn wl i tvdw xhrcbldv |
ylwgbus mb rxtlhcs pxo gm nke fdx t gbgxpey n      |
etcukepzsh lju ggekj qzje pevqgxie jsrd jazujl    |
chhb qm imw obiwybxd u fsksrsrt kmqdcyz ee hmsrq|
ozijipf oneeddpsz navymmta bdzqsoem vnpppsu         |
cbazu mhecthlegr unkd bppweqtgj p rmowz qyoxyt    |
bbhawdyd prjbxph ohp wqy hrq hnbn uvqnqqlr          |
jpxiog liexdz zosrk usvojb zmwzpo kjilefraa
```

```
贪心算法得出的最小代价（除了最后一行）为：34
动态规划得出的最小代价（除了最后一行）为：34
网上流传的错误的动态规划算法得出的最小代价（除了最后一行）为：42
525.584
请按任意键继续. . .
```

第 16 章：贪心算法

1. 贪心算法使所作的选择看起来都是当前最佳的，期望通所做的局部最优选择来产生一个全局最优解。
   对算法中的每一个决策点，做一个当时看起来是最佳的选择，这种启发式策略并不是总能产生最优解的。
2. 贪心算法对大多数优化问题能产生最优解，但很多问题也不能通过这样的局部最优解得到全局最优解（0-1 背包）。实际上，使用贪心算法最关键的就是在于如何证明贪心选择性质以证明当前的问题可以由贪心算法得到最优解
3. 在活动选择问题中，非常关键的一点先决条件就是：活动要先按照结束时间的单调递增顺序排序，这样才能贪心的选择第一个。
   从直觉上来看，这种活动选择方法是一种“贪婪的”选择方法，它给后面剩下的待调度任务留下了尽可能多的机会。也就是说，此处的贪心选择使得剩下的、未调度的时间最大化
4. 可以说：动态规划其实是“安全的贪心算法”的基础。无论如何，在每一个贪心算法的下面，几乎总是会有一个更加复杂的动态规划解。贪心算法实现简单速度快，但是证明贪心的正确性往往是很困难的，所以说安全的（能够取得最优解的）贪心算法下面总有一个动态规划算法来证明其正确性。
   所谓安全的贪心算法就是指一定能产生出全局最优解的贪心算法
5. 贪心算法的一般步骤（定义）
   a) 将优化问题转化成这样的一个问题，即先做出选择（对应于动态规划的先解决子问题再选择），再解决剩下的一个子问题。
   b) 证明原问题总是有一个最优解是做贪心选择得到的，从而说明贪心选择的安全。
   c) 说明在做出贪心选择后，剩余的子问题具有这样的一个性质。即如果将子问题的最优解和我们所做的贪心选择联合起来，就可以得出原问题的一个最优解。
6. 正确使用贪心算法的 2 个关键要素：贪心选择性质 和 最优子结构。
   a) 贪心选择性质：一个全局最优解可以通过局部最优（贪心）选择来达到；即当考虑做何选择时，我们只考虑对当前问题最佳的选择而不考虑子问题的结果。
   b) 最优子结构：一个问题的最优解包含了其子问题的最优解。
7. 要使用贪心算法就必须先证明以下两个性质：
   a) 每一步所做的贪心选择最终能产生一个全局最优解。
      在证明中先考察一个全局最优解，然后证明对该解加以修改，使其采用贪心选择，这个选择将原问题变为一个相似的、但更小的问题。
   b) 子问题的最优解与所做的贪心选择合并后，的确可以得到原问题的一个最优解。
8. 贪心算法所做的当前选择可能要依赖于已经做出的所有选择，但不依赖于有待于做出的选择或子问题的解。
9. 前缀编码：在所有的编码方案中，没有一个编码是另一个编码的前缀。
10. 一般地，就算证明不出来贪心算法能给出最优解，但是它一般都至少能给出次优解。所以贪心算法在实际的应用中是非常的普及的。

Output（代码：http://code.google.com/p/introduction-to-algorithms-notes/）：

```
a:  0.357143
b:  0.214286
c:  0.0571429
d:  0.171429
e:  0.0428571
f:  0.0571429
s:  0.0714286
```

```
w:   0.0285714
d ===> 00
b ===> 01
c ===> 1000
f ===> 1001
s ===> 1010
w ===> 10110
e ===> 10111
a ===> 11

请按任意键继续. . .
```

第 17 章：平摊分析

1. 平摊代价：N 个操作的总平摊代价即为这 N 个操作的总实际代价的上界。
2. 在平摊分析中，执行一系列数据结构操作所需要的时间是通过对执行所有操作所花费的时间求平均而得到的。
   平摊分析与平均情况分析的不同之处在于它不牵涉到概率；平摊分析保证在最坏情况下，每个操作具有平均性能。
3. 平摊分析表明：并不能总以最坏情况来衡量算法，因为最坏的情况并不总会发生，而且在绝大多数应用中最坏情况出现的次数一定是很少很少的。
   最明显的例子就是 C++ STL 中 vector.push_back 操作。
4. 通过平摊分析，可以获得对某种特定数据结构的认识，这种认识有助于优化设计。
5. 三种方法进行平摊分析：
   a) 聚集分析：就是指分析一系列操作的总时间
   b) 记账法：对每次操作的对象进行预先记账
   c) 势能方法：与记账法类似，但是将每次预留的势能视为是整个数据结构共享的
6. 聚集分析：由 N 个操作所构成的序列的总时间在最坏的情况下为 T(n)，则每个操作的平均代价（平摊代价）为 T(n)/n。
   以前的时间复杂度分析都是以单次操作为对象的分析它的最坏时间复杂度，而聚集分析所分析的是 N 次操作的总时间的最坏情况。应注意其与平均情况分析的不同之处！
   聚集分析：由序列的总最坏时间➔单次的平摊时间
7. 记账法：对序列操作中的每一个操作收取一定的费用，当所收取的费用比它实际应支付的费用多时就把多余的部分当作存款存起来，一个操作的平摊代价可以看作两部分：实际代价和存款（或被储蓄或被用完）。
   a) 存款可以用来在以后补偿那些其平摊代价低于其实际代价的操作。
   b) 如果希望通过对平摊代价的分析来说明每次操作的最坏情况平均代价较小，则操作序列的总平摊代价就必须是该序列的总的实际代价的一个上界。因为平摊代价是最坏时的平均代价，这意味着：只能有存款而不能有欠账！
   c) 明白这个存款不能为负很重要，因为这种分析方法是不允许欠账的，否则就不能满足平摊代价是操作总时间的最坏情况的平均这个定义。
   d) Eg: 对于栈的操作，当每次入栈时记账支付 2 元，1 元支付该 PUSH 操作的实际代价，还有 1 元用于支付该元素被 POP 出来时的代价。则可以保证在任何时间内都不会有欠账，POP 操作可以不收取任何费用。
8. 势能方法：将已预付的工作作为一种“势能”保存，它在需要时可以释放出来，以支付后面的操作。势能是与整个数据结构而不是其中的个别对象发生联系的。
   （记账法中的账与个别对象发生联系，比如栈操作时支付的 2 元就记在入栈的那个元素上）
9. 动态表：比如 C++ STL 中的 vector
   a) 可以通过平摊分析证明：插入和删除操作的平摊代价都仅为 O(1)
   b) 表扩张时：常用的启发式（启发信息就是容器原来的大小）方法是分配一个原表两倍大小的新表。
   c) 可以通过聚集分析来证明插入操作的平摊代价：

$$\sum_{i=1}^{N} C_i \le N + \sum_{j=0}^{\lfloor \lg N \rfloor} 2^j < n + 2n = 3n$$

   所以每次操作的平摊代价为 O(1)=3

d) 通过记账法来证明插入、删除操作的平摊代价：

i. 插入操作：每次插入操作需要支付 3 元的代价，分别是：将其自身插入到当前表中、当表扩张时其自身的移动、以及对另一个在扩张表时已经移动过的另一项的移动。于 1+1+1=3 元的代价

原有的元素……	NewItem
↑在原有的元素里还要再支付一个可能移动的代价	↑支付自身以后可能的移动操作的代价

ii. 并且可以证明：无论扩张时分配多少倍的新空间，每次插入操作的平摊代价都大于 2。所以标准库中取每次扩张时进行翻倍，这是一个均衡的很好的选择。

iii. 删除操作：

1. C++STL 的 vector 不会在删除元素时进行自动收缩，因为如果在元素不足 1/2 时进行收缩，最坏时会产生在这的情况（来回收缩和扩张）：在一次扩张之后，还没有做足够的删除来支付一次收缩的代价。类似地，在一次收缩之后，也有可能没有做足够的插入来支付一次扩张的代价。
   所以 vector 选择了在删除元素时不自动收缩来回避这个问题。如果要进行收缩，应该是元素减少为 1/4 时进行自动收缩，这样才能保证在表收缩之后有足够的存款进行支付一次收缩的代价。
2. 当删除至 1/4 个元素时进行自动收缩的平摊代价分析：
   每次删除操作需要支付的代价为：其自身的删除操作代价 和 剩下的元素可能进行的移动操作的代价。
   即最坏为：1 + 1 = 2 元的代价，最好为 1 + 0.33333 = 1.3333 元的代价，取最坏情况为 2 元。

剩下的元素……			
↑还要支付一个可能的移动		↑最早可能从这开始删除	最晚可能从这开始删除↑

iv. 因此：对于 动态表 的插入和删除操作而言，每个操作的平摊代价都有一个常数的上界。

第五部分：高级数据结构

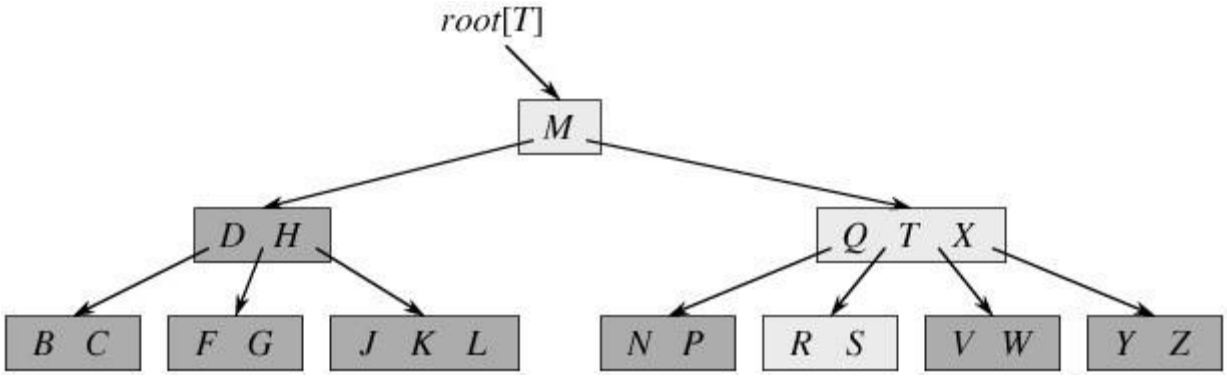
- B 树是一种被设计成专门存储在磁盘上的平衡查找树。因为磁盘的速度远远慢于内存，所以 B 树被设计成尽量减少磁盘访问的次数，知道了这一点之后就会很清楚明白 B 树的变形 B+树了，B＋树通过将数据存储在叶子结点从而增大了一个结点所包含的信息进而更加的减少了磁盘的访问次数。
- 可合并堆：这种堆支持 Insert,Mininum,Extract-Min,union,delete,decrease-key 操作。
- 二项堆能够在 O(lgn)的最坏情况时间内支持以上的各种操作。当必须支持 union 操作时，二项堆优越于二叉堆，因为后者在最坏情况下，合并两个二叉堆要花 O(n)的时间。
- 斐波那契堆对于以上各种除了 extract-min,delete 的操作外都只需要 O(1)的实际时间，而 extract-min,delete 也只需要 O(lgn)的平摊时间。它的重要优点在于 decrease-key 也只需要 O(1)的平摊时间。
  注意斐波那契堆的一些操作都只是平摊时间，并非最坏情况时间。现代的快速图算法中，很多是使用斐波那契堆作为其核心数据结构的。
- 不相交集合（查并集）：通过用一棵简单的有根树来表示每个集合，就可以得到惊人的快速操作：一个由 m 个操作构成的序列的运行时间为 O(n α(n))，而对于宇宙中的原子数总和 n，α(n)也<=4，所以可以认为实际时间是 O(n)。

第 18 章：B 树

1. B 树与红黑树类似，但是在降低磁盘的 I/O 次数方面要更好一些。所以 B 树一般都是应用于磁盘操作的系统中。
2. B 树思想产生在背景：就是大规模数据存储中，实现索引查询这样一个实际背景下，树节点存储的元素数量是有限的（如果元素数量非常多的话，查找就退化成节点内部的线性查找了），这样导致二叉查找树结构由于树的深度过大而造成磁盘 I/O 读写过于频繁，进而导致查询效率低下（因为读取一次磁盘相当于访问了无数次内存！），那么如何减少树的深度（当然是不能减少查询的数据量），一个基本的、很自然的想法就是：采用多叉树结构（由于树节点元素数量是有限的，自然该节点的子树数量也就是有限的）[5]
3. 在大多数系统中，B 树算法的运行时间主要由它所执行的 disk-read 和 disk-write 操作的次数所决定，因而应该有效地使用这两种操作，即让它们读取更多的信息更少的次数。由于这个原因，在 B 树中，一个结点的大小通常相当于一个完整的磁盘

页。因此，一个 B 树结点可以拥有的子女数就由磁盘页的大小所决定。

4. 很明显：B 树的分支因子越大越好，因为这样运行时间的绝大部分都是由磁盘存取次数决定的。分支因子越大，需要进行的磁盘存取次数就越少。
   但是这个分支因子是有限制的，一个结点的总大小不能大于磁盘中一个页的大小，否则在一个结点内操作时还要来回访问内存，反而会拖慢效率。
5. 一个常见的 B 树的变形，称作为 B＋树，所有的附属数据都保存在叶结点中，只将关键字和子女指针保存于内结点里，因此最大化了内结点的分支因子。
6. B 树又叫平衡多路查找树。一棵 t（t>=2，t 是 B 树的最小度数）阶的 B 树的特性如下：



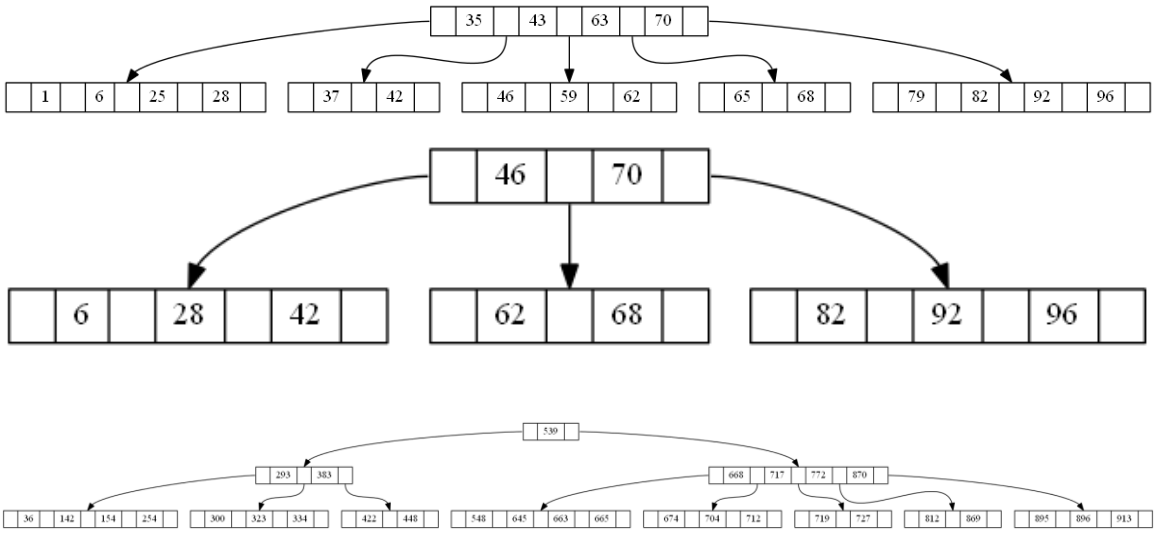
   a) 每个叶子结点具有相同的深度，即树的高度 h。
   b) 每一个结点能包含的关键字数有一个上界和下界，这些界可用 B 树的最小度数 t 来表示（t-1 <= n <= 2t-1）：
      i. 每个非根的结点必须至少含有 t-1 个关键字。每个非根的内结点至少有 t 个子女。如果树是非空的，则根结点至少包含一个关键字（同时意味着 2 个子女，根结点的下限永远是 1 个关键字，2 个女子）；
      ii. 每个结点可包含至多 2t-1 个关键字。所以一个内结点至多可有 2t 个子女。如果一个结点恰好有 2t-1 个关键字，我们就说这个结点是满的；
   c) 每个非终端结点中包含有 n 个关键字信息： (n，P0，K1，P1，K2，P2，……，Kn，Pn)。其中：
      i. Ki (i=1...n)为关键字，且关键字按顺序升序排序 K(i-1)< Ki。
      ii. Pi 为指向子树根的接点，且指针 P(i-1)指向子树种所有结点的关键字均小于 Ki，但都大于 K(i-1)。
      iii. 关键字的个数 n 必须满足：t-1 <= n <= 2t-1。
7. 当 t=2 时为最简单的 B 树，又称为 2-3-4 树，即每个结点的孩子数可能为 2,3,4 个；包含的关键字个数为 1,2,3 个。
8. 与红黑树相比，这里我们看到了 B 树的能力，虽然两者的高度都以 O(lgn)的速度增长，但对于 B 树来说底要大很多倍。对大多数的树的操作来说，要查找的结点数在 B 树中要比红黑树中少大约 lgt 的因子。因为在树中查找任意一个结点通常需要一次磁盘存取，所以磁盘存取的次数大大的减少了。
   如果让我来实现一个数据库系统，我肯定也会选取类似 B 树的数据结构作为基本。
9. B 树首先一是棵查找树，所以它的查找操作将会非常简单和高效。
10. B 树的插入操作： 插入操作一定发生在叶子结点，因为 B 树首先必须是一棵查找树。
    这是我的代码所使用的思路：因为插入操作肯定是在叶子结点上进行的,首先顺着书向下走直到要进行插入操作的叶子结点将新值插入到该叶子结点中去. 如果因为这个插入操作而使用该结点的值的个数>2*t-1 的上界，就需要递归向上进行分裂操作。如果分裂到了根结点，还要处理树长高的情况。这种思路简单，但不是很高效，因为经过了两趟，一趟向下；一趟向上。
    《算法导论》上介绍的方法则比较好：边下行边分裂，当沿着树往下查找新关键字时所属位置时，就分裂沿途遇到的每个满结点（包含叶结点本身）。因此每当要分裂一个满结点时，就能确保它的双亲不是满的。这种方法只需要一趟就可以完成，极大的减少磁盘的读取次数，而这正是 B 树所设计追求的，因而这种一趟的方法是更优的！
    但是我的代码只实现了我自己的简单方法，书上的方法实现起来应该也不难。
11. 插入操作时 B 树的分裂是 B 树升高的唯一途径！
12. B 树的删除操作：删除操作稍微复杂一些，因为删除操作不仅仅会发生在叶子结点，还可能会发生在内结点，这与插入操作不同。但是可以通过一个技巧消除这一点，找到要删除结点的前驱，然后与要删除的结点的关键值进行对调，再删除这个前驱结点就可以保证每次要删除的都是叶子结点。
    同样有可以进行边下行边合并的快速方法，而同样，我这里的代码也没有实现这种快速的方法，而是选用了传统的两趟的删除方法，具体如下：
    任一关键字 K 的中序前趋(后继)必是 K 的左子树(右子树)中最右(左)下的结点中最后(最前)一个关键字。根据 B 树的性质：B 树上每一个结点的 Key 的个数必须为[t-1, 2*t-1]之间，所以这里的 Min = t - 1。若被删关键字 K 所在的结点非树叶，则用 K 的中序前趋(或后继)K'取代 K，然后从叶子中删去 K'。从叶子*x 开始删去某关键字 K 的三种情形为：
    a) 情形一：若 x->keynum>Min，则只需删去 K 及其右指针(*x 是叶子，K 的右指针为空)即可使删除操作结束。

b) 情形二：若 x->keynum=Min，该叶子中的关键字个数已是最小值，删 K 及其右指针后会破坏 B-树的性质(3)。若*x 的左(或右)邻兄弟结点*y 中的关键字数目大于 Min，则将*y 中的最大(或最小)关键字上移至双亲结点*parent 中，而将*parent 中相应的关键字下移至 x 中。显然这种移动使得双亲中关键字数目不变；*y 被移出一个关键字，故其 keynum 减 1，因它原大于 Min，故减少 1 个关键字后 keynum 仍大于等于 Min；而*x 中已移入一个关键字，故删 K 后*x 中仍有 Min 个关键字。涉及移动关键字的三个结点均满足 B-树的性质(3)。 请读者验证，上述操作后仍满足 B-树的性质(1)。移动完成后，删除过程亦结束。

c) 情形三：若*x 及其相邻的左右兄弟(也可能只有一个兄弟)中的关键字数目均为最小值 Min，则上述的移动操作就不奏效，此时须*x 和左或右兄弟合并。不妨设*x 有右邻兄弟*y(对左邻兄弟的讨论与此类似)，在*x 中删去 K 及其右子树后，将双亲结点*parent 中介于*x 和*y 之间的关键字 K，作为中间关键字，与并 x 和*y 中的关键字一起"合并"为一个新的结点取代*x 和*y。因为*x 和*y 原各有 Min 个关键字，从双亲中移人的 K'抵消了从*x 中删除的 K，故新结点中恰有 2Min(即 2「m/2」-2≤m-1)个关键字，没有破坏 B-树的性质(3)。但由于 K'从双亲中移到新结点后，相当于从*parent 中删去了 K'，若 parent->keynum 原大于 Min，则删除操作到此结束；否则，同样要通过移动*parent 的左右兄弟中的关键字或将*parent 与其 左右兄弟合并的方法来维护 B-树性质。最坏情况下，合并操作会向上传播至根，当根中只有一个关键字时，合并操作将会使根结点及其两个孩子合并成一个新的根，从而使整棵树的高度减少一层。

13. 删除操作看似复杂，但是对一棵高度为 h 的 B 树，它只需要 O(h)次磁盘操作，因为在递归调用的过程之间，仅需要 O(1)次对 disk-read,disk-write 的调用，时间复杂度为 O(th)=O(tlog$_t$n).

Output（代码：http://code.google.com/p/introduction-to-algorithms-notes/）：

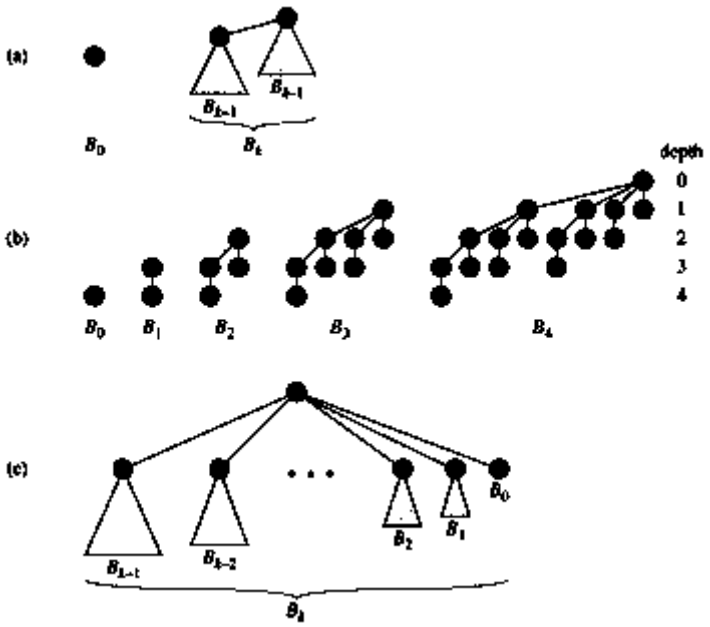


请按任意键继续. . .

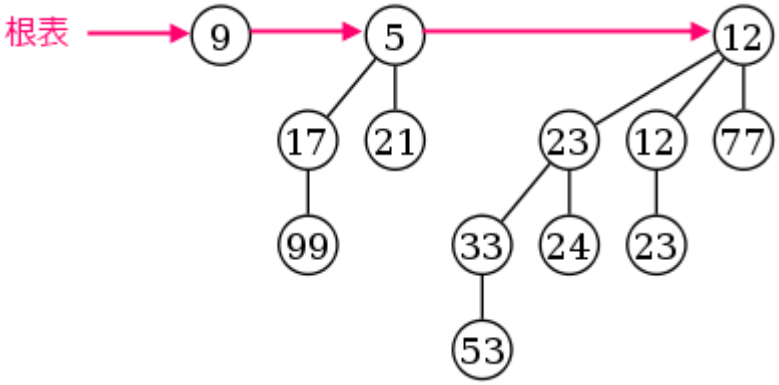
第 19 章：二项堆

1. 可合并堆就是普通的堆+支持 decrease-key,union 操作，但是应该高效的实现 union 操作是可合并堆最关键的部分。
2. 如果不需要高效的支持 UNION 操作，则普通的堆结构就很好了
3. 对于所有的堆结构：二叉堆、二项堆、斐波那契堆。它们的 search 操作都是很慢的，不能高效的支持 search 操作！因而在 decrease-key 和 delete 等涉及结点的操作时都需要一个指向结点的指针。
4. 一个二项堆是由一组二项树所构成的。
5. 二项树是一种递归的定义:
   1. 二项树 B[0]仅仅包含一个节点
   2. B[k]是由两棵 B[k-1]二项树组成，其中一颗树是另外一颗树的子树。

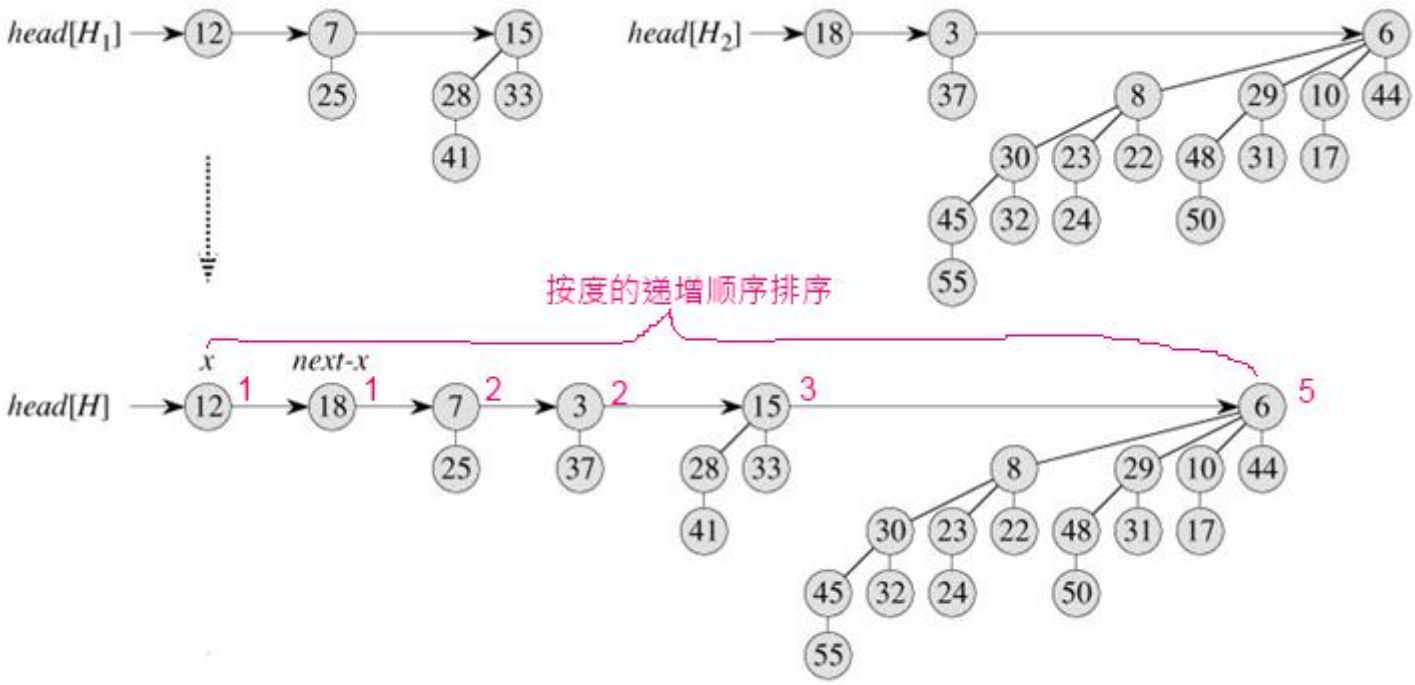
   下面是 B0 - B4 二项树：



6. 显然二项树具有如下的性质：
   a) 对于树 B[k]该树含有 2^k 个节点；
   b) 树的高度是 k；
   c) 在深度为 i 中含有 C_i^k 节点，其中 i = 0， 1，2 … , k;
7. 二项堆是由一组满足下面的二项树组成：
   第一个性质保证了二项树的根结点包含了最小的关键字。第二个性质则说明结点数为 n 的二项堆最多只有 logn + 1 棵二项树。
   a) H 中的每个二项树遵循最小堆性质：结点的关键字大于或等于其父结点的关键字。我们说这种树是最小堆有序的。
   b) 对于任意的整数 k 的话，在 H 不存在另外一个度数也是 k 的二项树；即至多（有 0 或 1 棵）有一棵二项树的根具有度数 K。
8. 由于二进制的 0、1 可以表示出任意的数值，所以任意个结点数的堆也可以由二项堆来表示。
   例如：13<1101>个结点的堆可由二项树 B3,B2,B0 来表示出来。
9. 根表：一个二项堆中的各二项树的根被组织成一个键表，我们称之为根表！

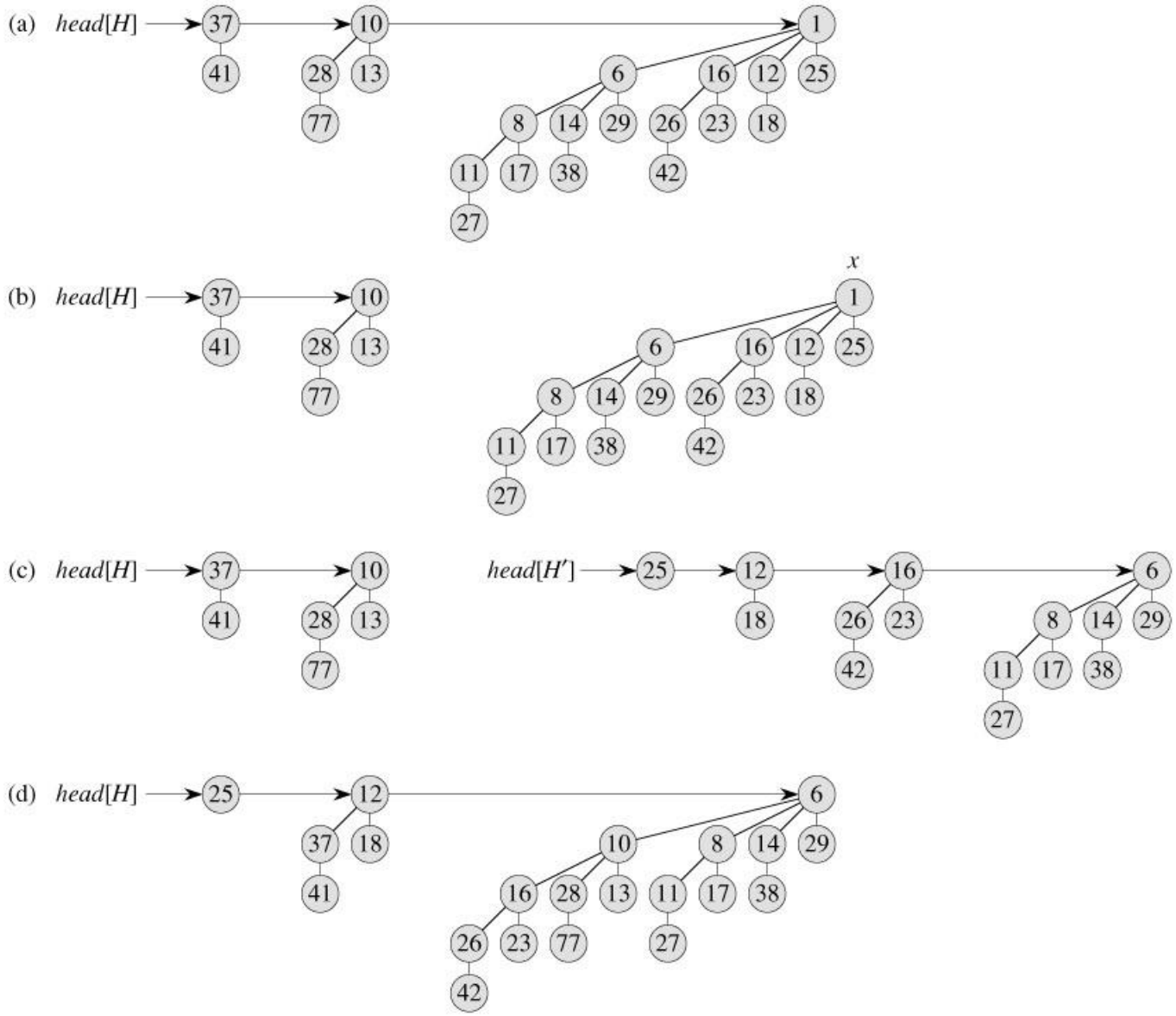


10. 二项堆的最重要的一个操作就是 UNION 操作，其它的操作都可以在 UNION 操作的基础上轻松的实现。
    大概思路为：将两个二项堆的根表连接起来组成一个大的二项树的连接，按“度”的单调递增顺序进行排序之后，从左至右来消除具有重复度的二项树。因为原本的每个二项堆中任意度 K 至多只有一个相应的二项树，所以这个消除重复的操作会非常容易。从小到大找到两个相同度 K 的二项树，然后连接成一个 K+1 度的二项树，直到链尾时合并完毕。
    此时的二项堆就满足对任意度 K 至多只有一棵二项树。



此操作的时间复杂度为 O(logn)

11. 其它的操作都可以通过将 UNION 操作作为基础操作来实现：
    a) 插入：NEW 一个一个结点的二项堆，然后调用 UNION 与原二项堆来进行合并
    b) 抽取最小关键字的结点：
    从原二项堆的根键上取下最小关键字结点的二项树，独立出来；
    将独立出来的二项树的根结点去掉（这就是整个二项堆的最小关键字），然后将它的子结点逆转并组合成为一个不含最小值的二项堆；
    最后将这个新的二项堆与原来剩下的二项堆进行合并操作。



c) 减少关键字：将要减小的关键字与它的父结点进行比较，如果需要就进行交换操作直到根链。使得该关键字在堆中“冒泡上升”直到正确的位置。

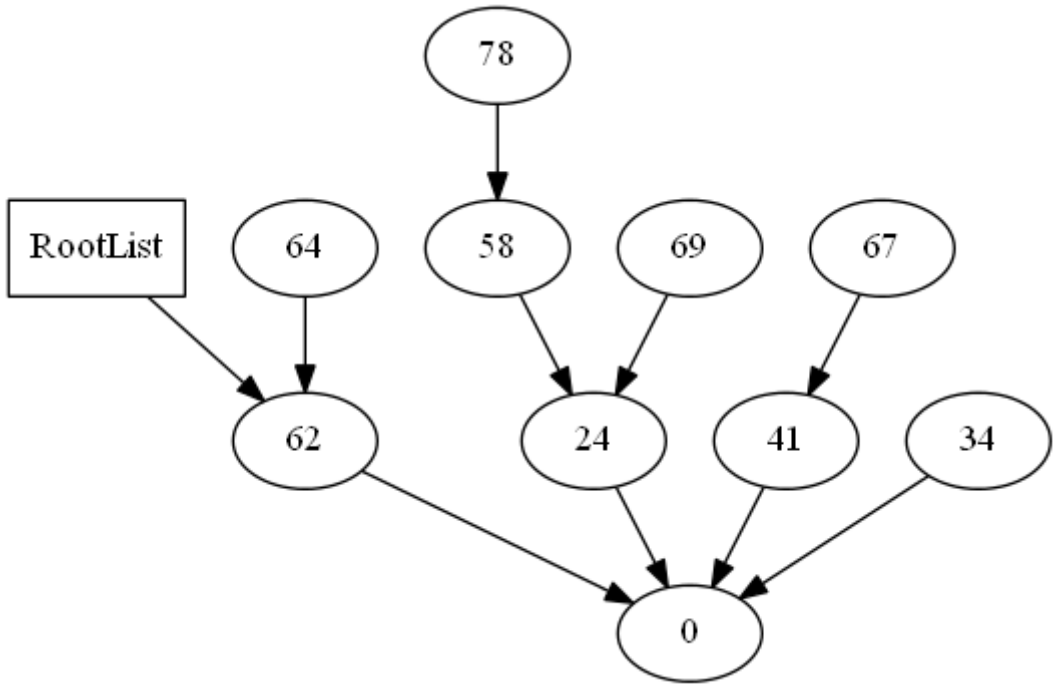
d) 删除一个关键字（调用其它操作实现）：

decrease-key(x, -max)

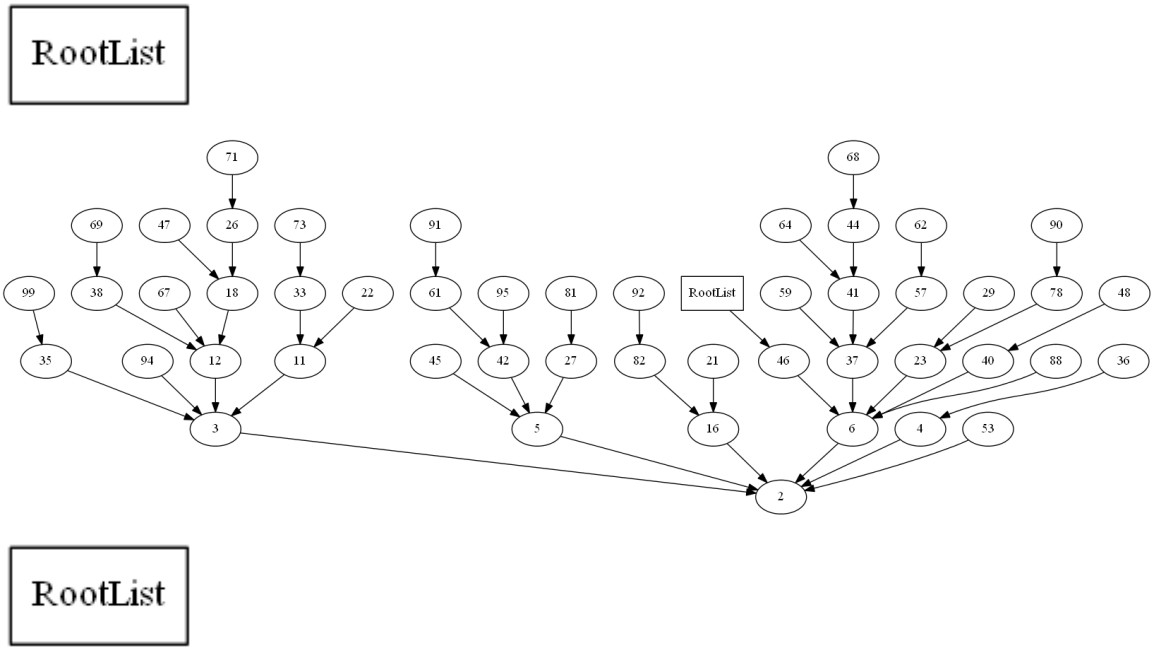
extract-min()

Output（代码：http://code.google.com/p/introduction-to-algorithms-notes/）：

二项堆



0 24 34 41 58 62 64 67 69 78

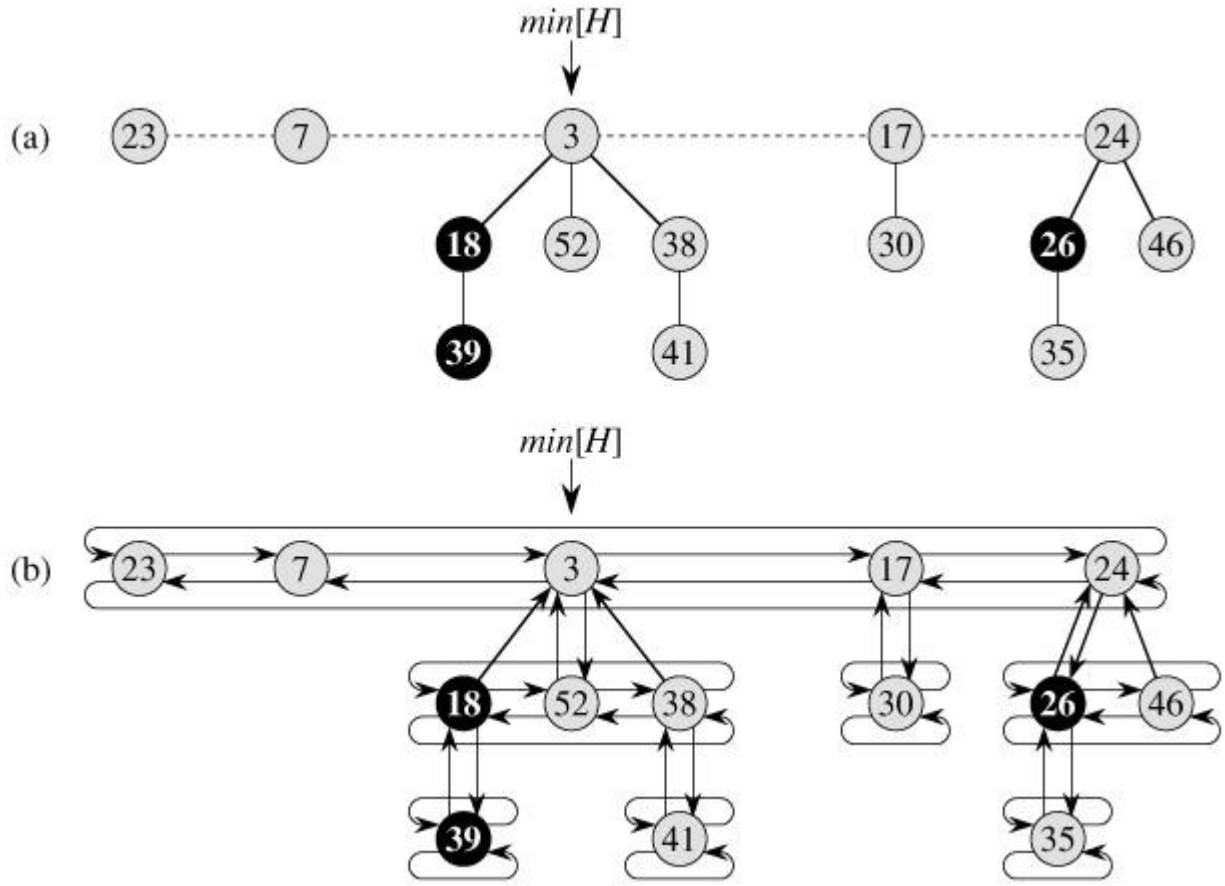


请按任意键继续. . .

第 20 章：斐波那契堆

1. 斐波那契堆与二项堆相比，取消掉了 2 个限制，因而更加的松散了：
   a) 斐波那契堆中的树不必要求是二项树
   b) 根链中的树必没有“每个度数至多只能有一棵树”的要求了
2. 斐波那契堆首先也是一种堆，所以它也满足堆的性质：子结点比父结点大！
3. 斐波那契堆的特点：不涉及删除元素的操作有 O(1)的平摊时间。Extract-Min 和 Delete 的数目和其它相比较小时效率更佳。
4. 斐波那契堆在优化加速图算法中有很大的用途。比如用于解决诸如最小生成树、寻找单源最短路径等问题的快速算法都要用到斐波那契堆。
5. 斐波那契堆由一组树（并不要求二项树）组成。但实际上，这种堆松散地基于二项堆。斐波那契堆的结构比二项堆更松散一些，从而可以改进渐近时间界。对结构的维护工作尽可能的拖延，直到方便时再做。
6. 同任何堆一样：斐波那契堆也不能很好地支持 search 操作。
7. 与二项堆中的树都是有序的不同，斐波那契堆中的树都是有根而无序的。每个节点包含一个指向其父节点的指针 p[x]，以

及一个指向其任一子女的指针 child[x]（指向子女双链表），因为这些子结点本身就没有顺序关系（兄弟之间是无序的，所以更加松散），所以可以随意的指向。



其中：mark[x]用于指示自从 x 上一次成为另一个结点子女以来，它是否失掉了一个孩子，在图中用黑色的结点来表示。

8. 如果仅支持“make-heap, insert, minimum, extract-min, union 操作，那么每个斐波那契堆就只是一组“无序的”二项树。
9. 对斐波那契堆上的各种可合并堆操作来说，其关键思想就是尽可能久地将工作推后。
   这种 lazy-compute 很可能会大大的改善性质，但是也会造成后面真正进行维护时的算法执行时间的常数因子过大。
10. 斐波那契堆是通过指向根链上的最小结点的指针 min[H]来进行访问的。
11. 斐波那契堆的日常操作都非常简单，除了 extract-min 以外：
    注：这里的时间复杂度均指的是平摊代价，而非一般使用的最坏情况时间。
    a) 插入操作 O(1)：直接插入到根链上去，再与最小结点的 min[H]进行一下比如，如果比 min[H]还小，就将 min[H]指针指向新插入的结点。
    b) 寻找最小节点 O(1)： min[x]指向的节点即为最小节点。
    c) 合并两个斐波那契堆 O(1)：分为 3 步： 1：合并根表；2：设置新的 min[h]；3。重置 n[x]。
       斐波那契堆过于松散的性质使得连合并操作都只需要 O(1)的时间，只需要合并两个根链就可以了。
    d) 抽取最小节点 O(lgn)：相对来说这是最复杂的工作。被延迟的对根表的调整工作最终由这个操作进行。
       1：去掉根表上的最小值，将这个最小值的每个孩子都加入根表；2：将根表上相同度数树的合并。
       （一点都不复杂，代码与二项堆一样容易实现）
       这里的在根链上合并操作与二项树基本相同，就是按“度的递增”顺序排列所有的子树之后，合并具有相同度的子树，使得最后的根链上每一个度 K 都只有至多一棵子树。（这里又与二项堆神似了，所以说斐波那契堆是松散地基于二项堆。形式化地描述为：
       “调整根表的步骤 1：在根表中找出两个具有相同度数的根 x 和 y，且 key[x]<key[y] 2：将 y 与 x 连接。将 y 从根表里去掉，成为 x 一个孩子，并增加 degree[x]。同时，如果 y 上有标记的话也被清除掉。”
    e) 减小一个节点的权值 O(1)：
       i. 若此减小不影响堆序，不作调整；
       ii. 若影响堆序，则从堆中删除该节点，将其加入根表，并检查其父亲的 mark 位；若为 false，则停止，并将其置为 true；若为 true，则删除其父亲，继续递归向上执行；直到一个节点 mark 域为 false 或该节点为根节点为止。
    f) 删除一个节点 O(lgn)：1：将该节点权值调整至最小；2：抽取最小值。
12. 总结：斐波那契堆之所以高效，就是因为它松散了，将很多二项堆要维护的工作都推迟到了 extract-min 和 delete 操作之中去了。而它之所以可以这样进行偷懒，就是因为它的限制非常的少，并没有像二项堆一样的要求子树全部必须是二项树，而且根表上的任意度数的子树至多只能有一个。没有了这些限制，斐波那契堆就可以实现的非常的松散以至少将日常工作

的维护时间都平摊给 extract-min 和 delete 操作。

但是，正如斐波那契堆现在更多的应用于理论中，实践中使用的并不多的现象所映射的，在大部分的应用中斐波那契堆并不能带来大幅度的效率的提升。因为它并不是减少了要做的工作，只是将很多要做的工作都进行了推迟，可能出现这样的情况：

a) 合并操作 O(lgn)=>O(1)
b) 插入操作 O(lgn)=>O(1)
c) 减少权值操作 O(lgn)=>O(1)
d) 但是 extract-min 操作 O(lgn)=>O(3*lgn)≈O(lgn)

这使得在理论分析时，它的所有操作的渐近时间都至少不变坏，但是在实际使用中却没有任何的改善。

当然这是最坏的情况了，一般对于规模足够大的输入，斐波那契堆是能够很大改善算法的性能的，尤其是对于一些 extract-min 和 delete 操作少的情况。比如斐波那契堆的使用能加快 Prime 和 Djikstra 算法的执行速度。

Output（代码：http://code.google.com/p/introduction-to-algorithms-notes/）：

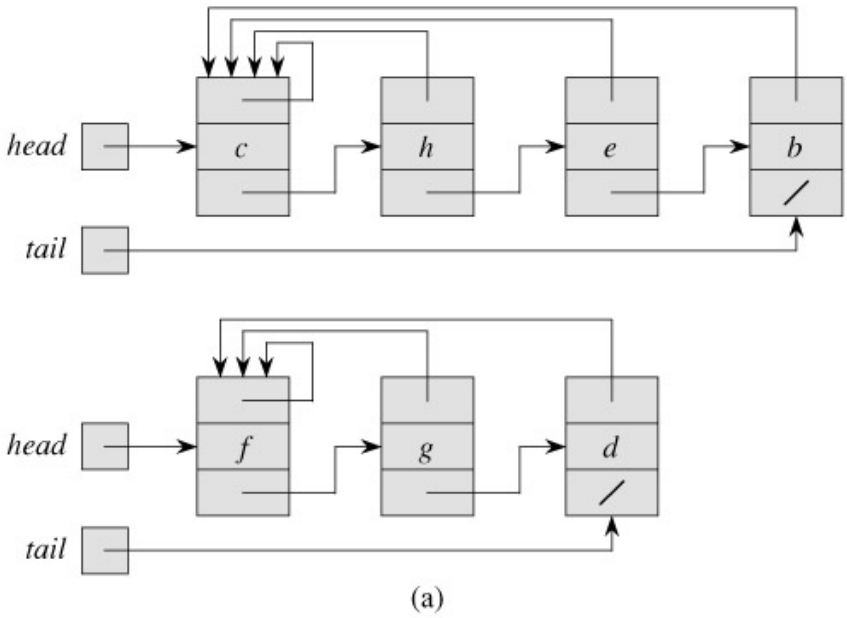
```
insert a:4
insert b:2
insert c:7
insert d:5
insert e:1
insert f:8
maxDegree=0    count=6    roots=e:1:0:0 f:8:0:0 d:5:0:0 c:7:0:0 a:4:0:0 b:2:0:0
min=e:1
removeMinimum
maxDegree=2    count=5    roots=b:2:0:0 a:4:2:0(c:7:0:0 d:5:1:0(f:8:0:0 ) )
min=b:2
removeMinimum
maxDegree=2    count=4    roots=a:4:2:0(c:7:0:0 d:5:1:0(f:8:0:0 ) )
min=a:4
removeMinimum
maxDegree=1    count=3    roots=d:5:1:0(f:8:0:0 ) c:7:0:0
min=d:5
removeMinimum
maxDegree=1    count=2    roots=c:7:1:0(f:8:0:0 )
min=c:7
removeMinimum
maxDegree=0    count=1    roots=f:8:0:0
min=f:8
removeMinimum
maxDegree=0    count=0    roots=


insert a:400
insert b:200
insert c:70
insert d:50
insert e:10
insert f:80
maxDegree=0    count=6    roots=e:10:0:0 f:80:0:0 a:400:0:0 b:200:0:0 c:70:0:0 d:50:
0:0
min=e:10
removeMinimum
min=d:50
```

```
maxDegree=2    count=5    roots=d:50:0:0 c:70:2:0(b:200:0:0 f:80:1:0(a:400:0:0 ) )
decrease key of a:400 to 40
min=a:40
maxDegree=2    count=5    roots=a:40:0:0 c:70:2:0(b:200:0:0 f:80:0:1 ) d:50:0:0
decrease key of b:200 to 20
min=b:20
maxDegree=2    count=5    roots=b:20:0:0 c:70:1:0(f:80:0:1 ) d:50:0:0 a:40:0:0
decrease key of c:70 to 7
min=c:7
maxDegree=2    count=5    roots=c:7:1:0(f:80:0:1 ) d:50:0:0 a:40:0:0 b:20:0:0
decrease key of d:50 to 5
min=d:5
maxDegree=2    count=5    roots=d:5:0:0 a:40:0:0 b:20:0:0 c:7:1:0(f:80:0:1 )
decrease key of f:80 to 8
min=d:5
maxDegree=2    count=5    roots=d:5:0:0 a:40:0:0 b:20:0:0 c:7:1:0(f:8:0:1 )



insert AA:4
insert BB:2
insert CC:7
insert DD:5
insert EE:1
insert FF:8
maxDegree=2    count=11    roots=EE:1:0:0 FF:8:0:0 DD:5:0:0 CC:7:0:0 a:40:0:0 b:20:0
:0 c:7:1:0(f:8:0:1 ) d:5:0:0 AA:4:0:0 BB:2:0:0
min=EE:1
removeMinimum
maxDegree=3    count=10    roots=BB:2:1:0(AA:4:0:0 ) d:5:3:0(b:20:0:0 DD:5:2:0(FF:8:
0:0 CC:7:1:0(a:40:0:0 ) ) c:7:1:0(f:8:0:1 ) )
min=BB:2
removeMinimum
maxDegree=3    count=9    roots=AA:4:0:0 d:5:3:0(b:20:0:0 DD:5:2:0(FF:8:0:0 CC:7:1:0
(a:40:0:0 ) ) c:7:1:0(f:8:0:1 ) )
min=AA:4
removeMinimum
maxDegree=3    count=8    roots=d:5:3:0(b:20:0:0 DD:5:2:0(FF:8:0:0 CC:7:1:0(a:40:0:0
 ) ) c:7:1:0(f:8:0:1 ) )
min=d:5
removeMinimum
maxDegree=2    count=7    roots=DD:5:2:0(FF:8:0:0 CC:7:1:0(a:40:0:0 ) ) b:20:0:0 c:7
:1:0(f:8:0:1 )
min=DD:5
removeMinimum
maxDegree=2    count=6    roots=c:7:1:0(f:8:0:1 ) CC:7:2:0(a:40:0:0 FF:8:1:0(b:20:0:
0 ) )
min=c:7
removeMinimum
maxDegree=2    count=5    roots=CC:7:2:0(a:40:0:0 FF:8:1:0(b:20:0:0 ) ) f:8:0:1
min=CC:7
removeMinimum
maxDegree=2    count=4    roots=f:8:2:1(a:40:0:0 FF:8:1:0(b:20:0:0 ) )
```

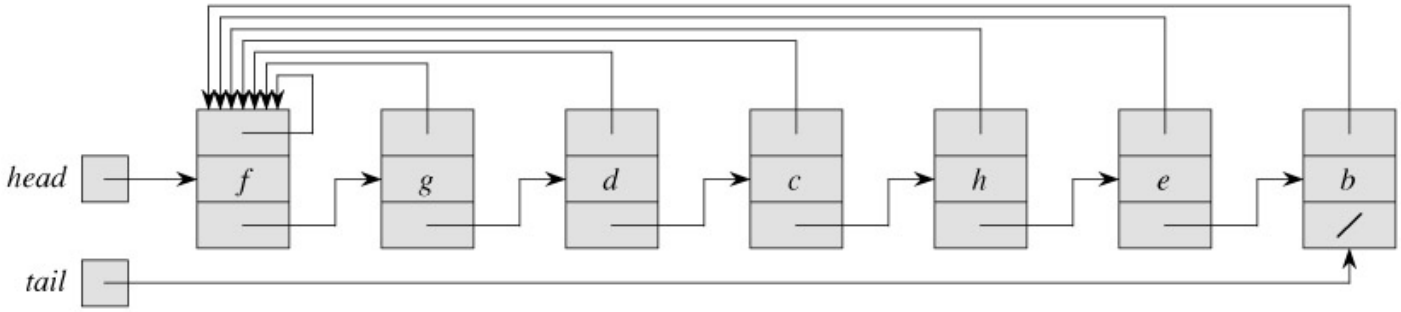
```
min=f:8
removeMinimum
maxDegree=1    count=3    roots=FF:8:1:0(b:20:0:0 ) a:40:0:0
min=FF:8
removeMinimum
maxDegree=1    count=2    roots=b:20:1:0(a:40:0:0 )
min=b:20
removeMinimum
maxDegree=0    count=1    roots=a:40:0:0
min=a:40
removeMinimum
maxDegree=0    count=0    roots=


请按任意键继续. . .
```

第 21 章：用于不相交集合的数据结构

1. 用于不相交集合的数据结构又称为查并集：在很多的应用中（比如图的算法中经常使用，还有哈夫曼编码等），将 N 个不同的元素分成一组不相交的集合，不相交集合上有两个重要的操作：找出给定元素所属的集合 和 合并两个集合。
2. 常用的两种表示方法：一种为链表的实现；另一种是更有效的树的表示法。
3. 采用树的表示法的运行时间在实践上是线性的，但从理论上来说是超线性的。
   这句话的意思是：理论上分析的运行时间是大于（超过）线性的，但是在实践中证明它的运行时间却是线性的。
4. 每个集合通过一个代表来识别，代表即集合中的某个成员。在某些应用中，哪一个成员被选作代表是无所谓的，而关键的是在集合未被修改的前提下，两次寻找得到的答案应该是相同的。
5. 不相交集合的链表表示：



(a)



(b)

a) 每个链表的第一个对象作为它所在集合的代表

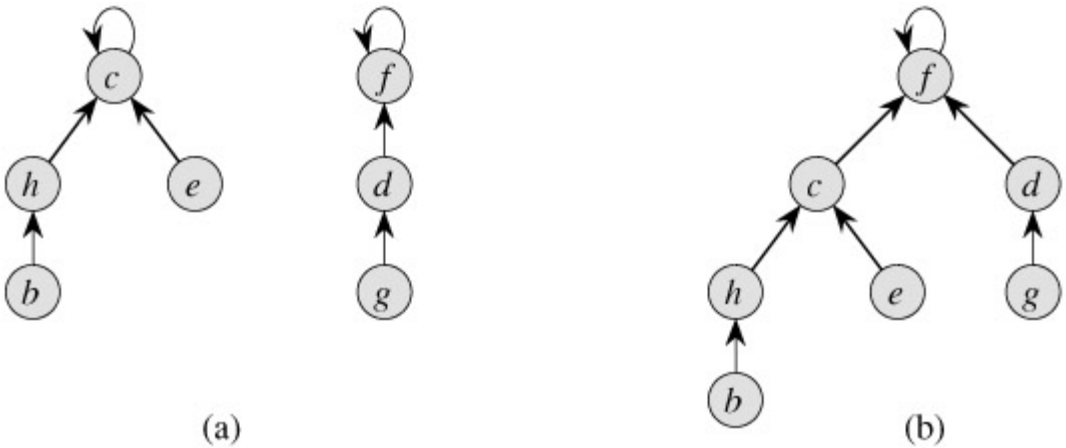
b) 链表上的第个结点都有指向它所在集合的代表的指针，从而使得找出所属集合的操作时间降为 O(1)

c) 但是因为每个结点都有指向代表的指针，使得在进行合并操作时，需要更新所有结点的指向代表的指针，从而时间复杂度为 O(n)。这算是有利也有弊吧。

d) 一种加权合并启发式策略：在进行合并操作时，总是把较短的表拼到较长的表上去。

这种简单的小技巧的正式的名字都可以叫做：启发式信息，启发式策略！以后涉及到这种情况的时候也可以这么叫。

6. 不相交集合森林表示法：用有根树来表示集合，树中的每个结点都包含集合的一个成员，每棵树表示一个集合。每个成员仅指向其父结点，每棵树的根包含了代表，并且是它自己的父结点。



(a)

(b)

7. 在引入了两种启发式策略（按秩合并、路径压缩）之后，不相交集合森林表示法就是目前已经的、渐近意义最快的不相交集合数据结构了。

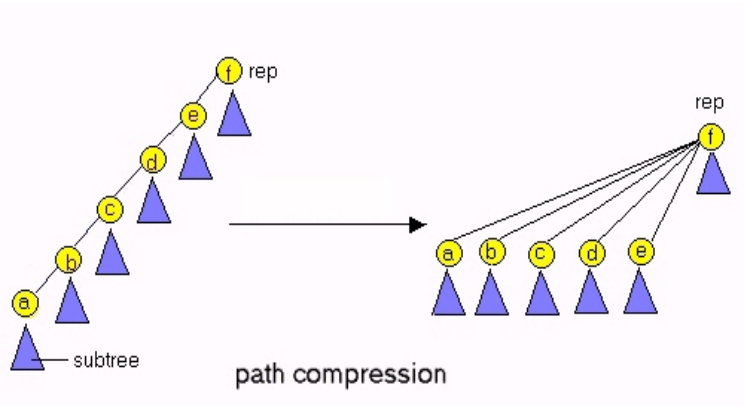
a) 接秩合并：便包含较少结点的树的根指向包含较多结点的树的根。秩表示的是结点高度的一个上界。

每个结点的秩是结点高度的一个上界，只有在进行 UNION 操作时，而且进行合并操作的两个集合的秩相同时，才会给最后的根结点的秩+1。

```
template<typename T>
static void _Link(DisjointSet<T> &x, DisjointSet<T> &y)
{
```

```
    if (x.Rank > y.Rank)
    {
        y.Parent = &x;
    }
    else
    {
        x.Parent = &y;
    }
    if (x.Rank == y.Rank)
    {//只有在秩相同时才会将最后的根结点的秩+1
        ++y.Rank;
    }
}
```

b) 路径压缩：（查找路径：在查找一个结点所属的集合时的查找路径上访问过的所有的结点）在每次进行查找元素所属集合的操作时，使得查找路径上的每个结点都直接指向根结点。路径压缩并不改变结点的秩。
这种路径压缩使用了两趟的方法：一趟是沿查找路径向上升，直至找到根；第二趟时沿查找路径下降，以便更新查找路径上的每个结点，使之指向根。这种路径压缩的思想简直是太棒了！神奇啊！应该学习这种思想，碰到类似的问题也要想到类似的解决办法。




```
template<typename T>
static DisjointSet<T> & FindSet(DisjointSet<T>
&a_set)
{
    //路径压缩
    if (&a_set != a_set.Parent)    //判断本身不是
代表
    {
        a_set.Parent = &FindSet(*a_set.Parent);
    }
    return *a_set.Parent;
}
```

c) 使用了这两种启发式策略之后，算法的最坏运行时间为 O(m*α(n))，其中 m 是操作序列中所有的操作的个数总和。对于宇宙中的所有原子数之和 x, α(x)<=4，所以说算法的实际时间是线性的。

Output（代码：http://code.google.com/p/introduction-to-algorithms-notes/）：

```
Union : 1---7
Union : 4---0
Union : 9---4
Union : 8---8
Union : 2---4
0------------->0
1------------->7
2------------->0
3------------->3
4------------->0
5------------->5
6------------->6
```

```
7---------->7
8---------->8
9---------->0
请按任意键继续. . .
```

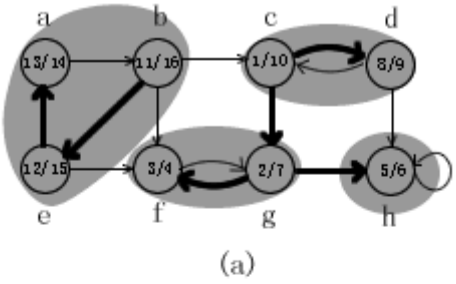
第六部分：图算法

图是计算机科学中常用的一类数据结构。
两种图的遍历方法：广度优先遍历（求每条边都是单位权值的图的最短路径）和深度优先遍历（拓扑排序、将有向图分解为强连通子图）。
最小权生成树：由连接了图中所有顶点的、且权值最小的路径所构成。
有向图中的最大流问题：这是个一般性的问题，会以多种形式出现；一个好的计算最大流量的算法可以用来有效地解决多种相关的问题。

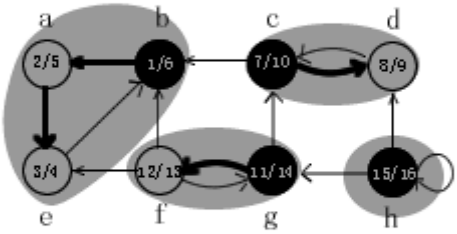
第 22 章：图的基本算法

1. 图的搜索技术是图算法领域的核心
2. 两种最普通的图的表示方法：邻接表法（节省空间）和邻接矩阵法（查询高效）
   稀疏图多用邻接表法来表示；而稠密图则用邻接矩阵法来表示比较好。
3. 不论是有向图还是无向图，邻接表法都有一个很好的特性，即它所需要的存储空间为 O(V+E)；邻接表表示法稍作修改就能支持其它多种图的变体，因而有着很强的适应性。
4. 无向图的邻接矩阵 A 就是它的转置矩阵：$A=A^T$。在某些应用中，可以只存储邻接矩阵的对角线及对角线以上的部分，这样一来，图所占用的存储空间几乎可以减少一半
5. 邻接表表示和邻接矩阵表示在渐近意义上至少是一样有效的，但由于邻接矩阵简单明了，因而当图较小时，更多多地采用邻接矩阵来表示。另外，如果一个图不是加权的，采用邻接矩阵的存储形式还有一个优越性：在存储邻接矩阵的每个元素时，可以只用一个二进位，而不必有一个字的空间。
   这样，当采用了二进位以及表示无向图的技巧时，邻接矩阵法的占用空间大的缺点就可以得到一定程度上的改善！
6. 广度优先搜索能够得到这种意义上的最短路径：每条边的权值都为 1，即所有的边都具有单位权值。
   广度优先所产生的广度优先树是每个顶点到 s 的最短距离。
7. 深度优先搜索除了创建一个深度优先森林外，深度优先搜索同时为每个顶点加盖时间戳。每个顶点 v 有两个时间戳：当顶点 v 第一次被发现时，记录下第一个时间戳 d[v]，当结束检查 v 的邻接表时，记录下第二个时间戳 f[v]。
   许多基于深度优先搜索的图算法都用到了时间戳，它们对推算深度优先搜索的时行情况有很大的帮助。
8. 这次重新复习深度优先算法，得到的最大的启示就是使用了这 2 个时间戳，真的是很有用很好的创新啊。当记录下这 2 个时间戳之后，很多东西都可以由这对时间戳来推导出来了（比如拓扑排序、深度遍历的次序等）。
9. 广度搜索通常用于从某个源顶点开始，寻找最短路径距离（以及相关的先辈子图）。深度优先搜索通常作为另一个算法中的一个子程序。
10. 后裔区间的嵌套：在一个（有向或无向）图 G 中的深度优先森林中，顶点 v 是顶点 u 的后裔，当且仅当 d[u]<d[v]<f[v]<d[u]，由这关系可以推导出大部分的与时间戳相关的性质。
11. 边的分类：树边、反向边、正向边、交叉边。
12. 拓扑排序：在很多应用中，有向无回路图说明事件发生的先后次序。
    在深度优先遍历的基础上，对有向无回路图进行拓扑排序简直是小菜一碟。根据遍历所得到的时间戳 f[i]逆向排序就好了。
13. 拓扑排序的顶点以与其完成时间时间相反的顺序出现。这种新方法真是长见识啊，比我以前使用的方法好多了，这种方法在遍历完只需要一个简单的 sort 就完成了拓扑排序，时间复杂度也降低为 O(V+E)。
14. 这种新的拓扑排序的方法的理论基础是：对于任一对不同的顶点 u,v，如果存在一条从 u -> v 的边，那么 u 在拓扑排序的结果中一定在 v 的前面。而又根据后裔区间嵌套的定理：如果存在 u -> v，那么 f[v]<f[u]，所以得证根据 f 逆向排序得到的顺序一定为拓扑排序。

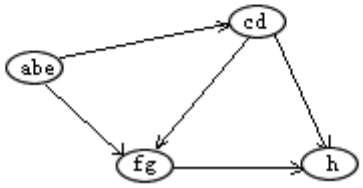
15. 强连通分支：有向图 G=(V, E)的一个强连通分支就是一个最大的顶点集合 C，对于 C 中的每一对顶点 u,v，都有 u -> v 及 v -> u；亦即顶点 u 和 v 是互相可达的。
16. 寻找强连通分支的简明算法：



(a)



(b)



(c)

a) 对 G 进行深度优先遍历得到每个顶点的时间戳 f[x]；
b) 求得 G 的返回图 G^T；
c) 按照 f[x]的逆向顺序为顶点顺序对 G^T 进行深度优先遍历，即按照 G 的拓扑排序的顺序对 G^T 再进行深度优先遍历；
d) 步骤 c 得到的各棵子树就是原图 G 的各强连通分支。

17. 寻找强连通分支的算法的时间复杂度为 O(V+E)。

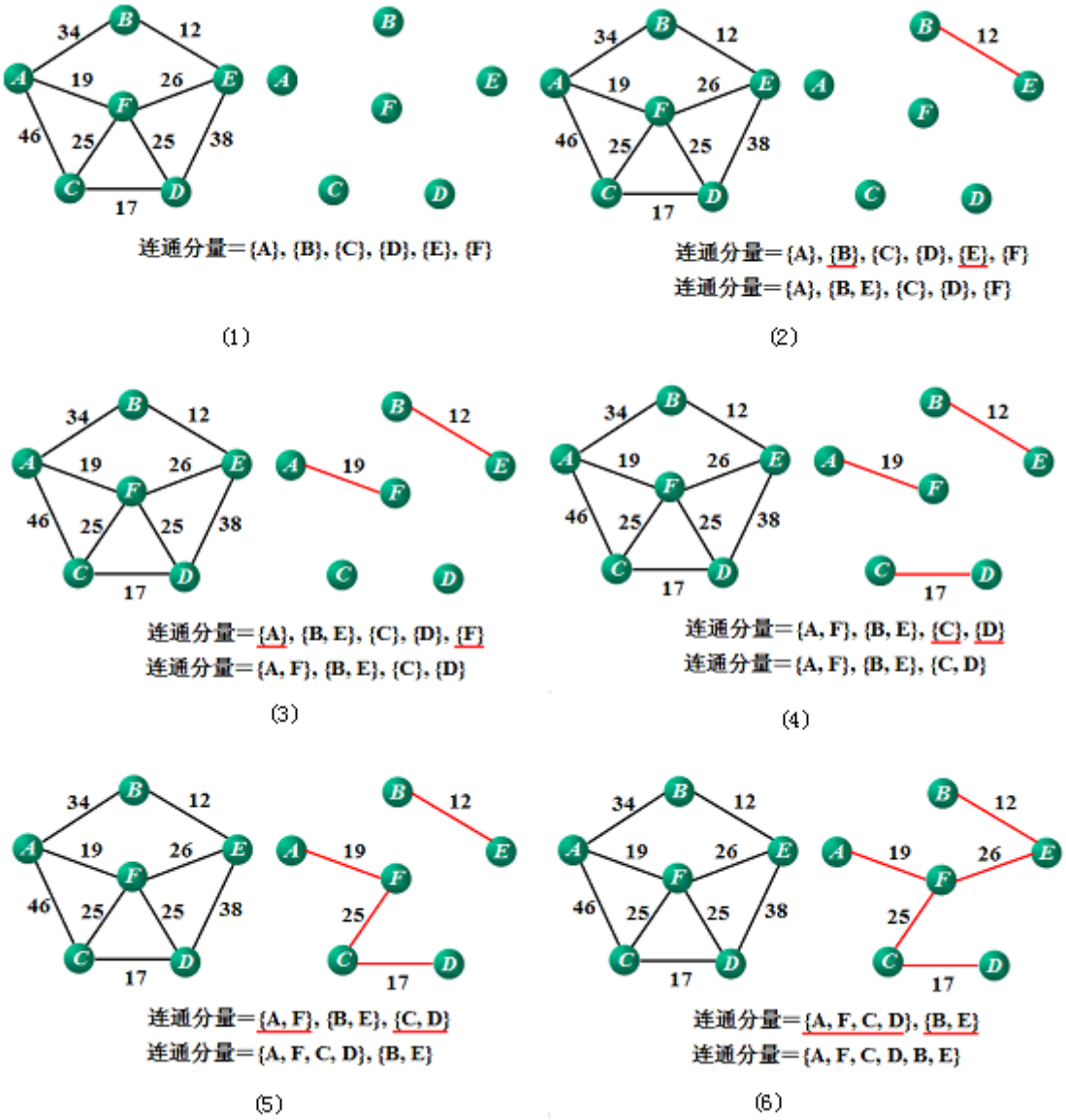
Output（代码：http://code.google.com/p/introduction-to-algorithms-notes/）：

```
广度优先遍历：s  r  w  v  t  x  u  y

深度优先遍历：r[1,16] s[2,13] w[3,12] t[4,11] u[5,10] x[6,9] y[7,8] v[14,15]
深度优先遍历：r[1,16] s[2,13] w[3,12] t[4,11] u[5,10] x[6,9] y[7,8] v[14,15]
拓扑排序：
shirt[15,18]
tie[16,17]
watch[13,14]
socks[11,12]
undershorts[1,10]
pants[2,9]
belt[5,8]
jacket[6,7]
shoes[3,4]
强连接分支
a[1,16] b[2,15] e[13,14] c[3,12] g[8,11] f[9,10] d[4,7] h[5,6]
aeb
cd
gf
```

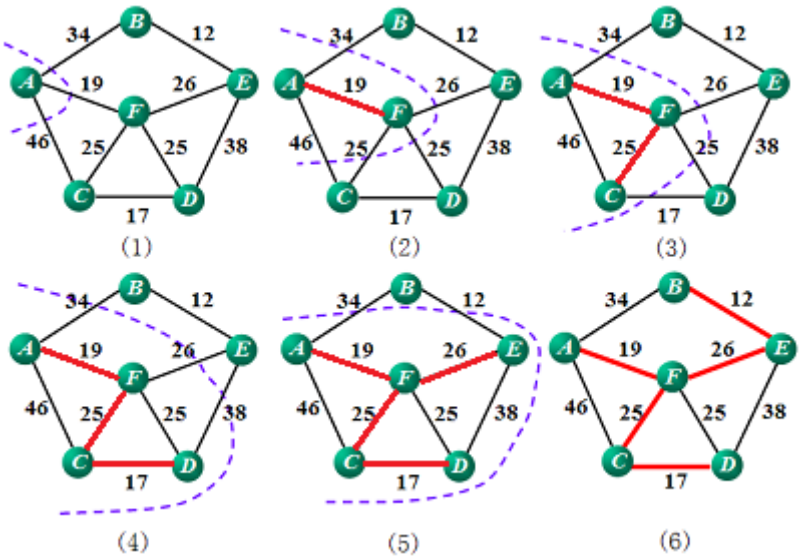
```
h
请按任意键继续. . .
```

第 23 章：最小生成树

1. 最小（权值）生成树：由 n-1 条边，连接了所有的 n 个顶点，并且所有边上的权值和最小。
2. Kruskal 算法和 Prim 算法：这两种算法中都使用普通的二叉堆就可以很容易地达到 O(ElgV)的运行时间。通过采用斐波那契堆，Prim 算法的运行时间可以减少到 O(E +VlgV)，这对于稠密图来说是个很大的改进。
3. 贪心策略可以在最小生成树问题中得到最优解，事实上这里的 Kruskal、Prim 方法都是贪心算法。它们也都是可以被证明的一定能够得到最优解！
4. 在 Kruskal 算法中，集合 A 是一个森林，加入集合 A 中的安全边总是图中连接两个不同连通分支的最小权边；



在 Prim 算法中，集合 A 仅形成单棵树，添入集合 A 中的安全边总是连接树与一个不在树中的顶点的最小权边。



5. 为什么说 Prim 算法有着更好的实际效率：
   a) Prim 算法在执行的过程中，将不在树中的所有顶点都放在一个基于 key 域的最小优先队列 Q 中；
   b) 每次在选取安全边时，只需要从 Q 中弹出最小 key 值的顶点即可，而不需要像 Kruskal 一样为对所有的边的权值进行一次排序；
   c) Prim 算法使用了用于快速或者最小权值边的技巧（二叉堆、二项堆、斐波那契堆）来加速算法的运行。在最朴素的选取安全边的算法中，需要遍历剩下的所有的边，所需要的时间复杂度为 O(EV)，而采用堆来优化后只需要 O(ElgV)，大大地改善了算法的执行时间（堆的好处）。
   d) 不过 Kruskal 算法实现简单，所以对于一般的应用 Kruskal 算法很常见。
6. Prim 算法的性能取决于优先队列 Q 是如何实现的，因此如果使用斐波那契堆来实现最小优先队列，就可以将 Prim 算法的运行时间改进为 O(E+VlgV)。

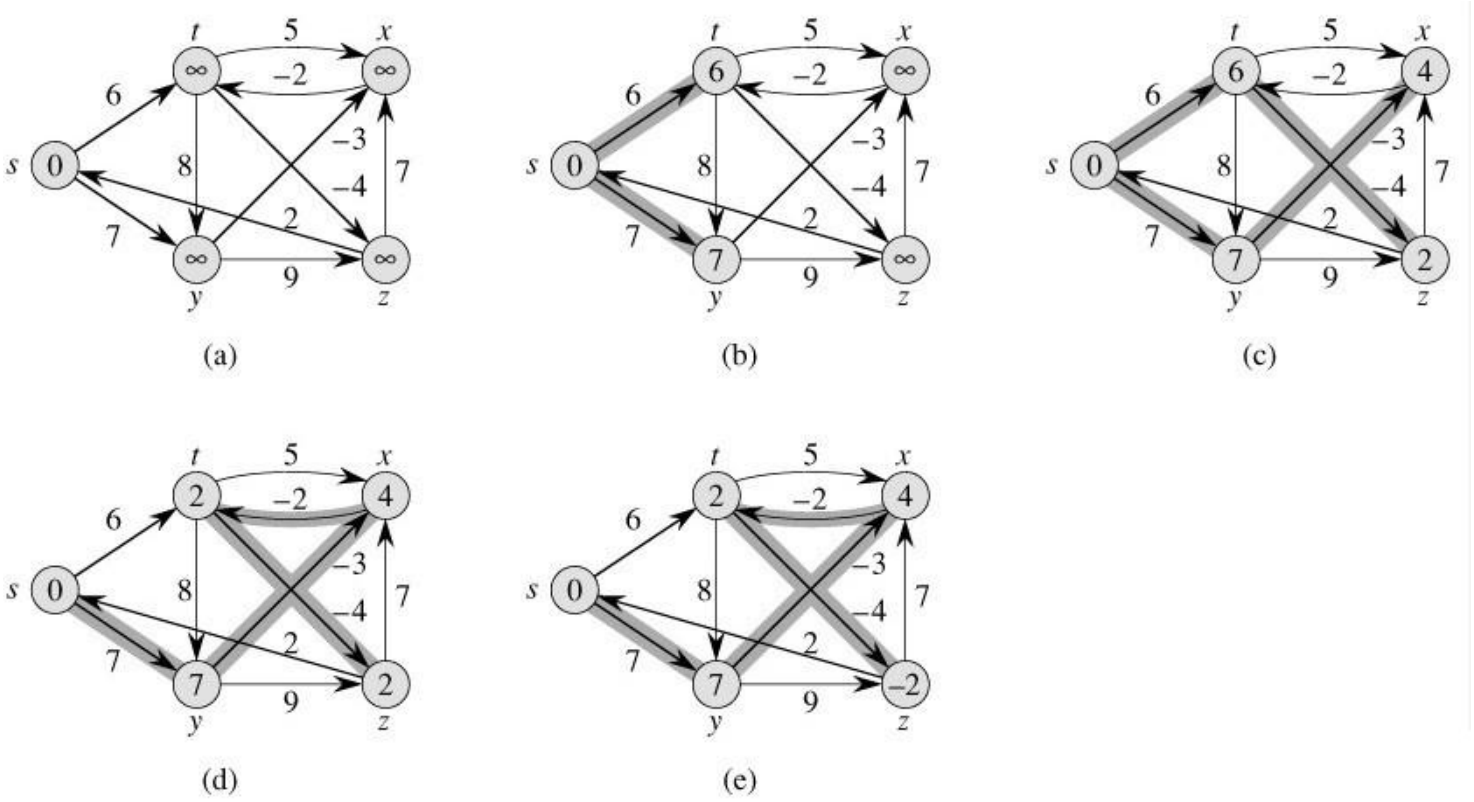
Output（代码：http://code.google.com/p/introduction-to-algorithms-notes/）：

```
6   -->   7
2   -->   8
5   -->   6
0   -->   1
2   -->   5
2   -->   3
0   -->   7
3   -->   4
Prim 最小生成树
& -- a
a -- b
b -- c
c -- i
c -- f
f -- g
g -- h
c -- d
d -- e
请按任意键继续. . .
```

第 24 章：单源最短路径

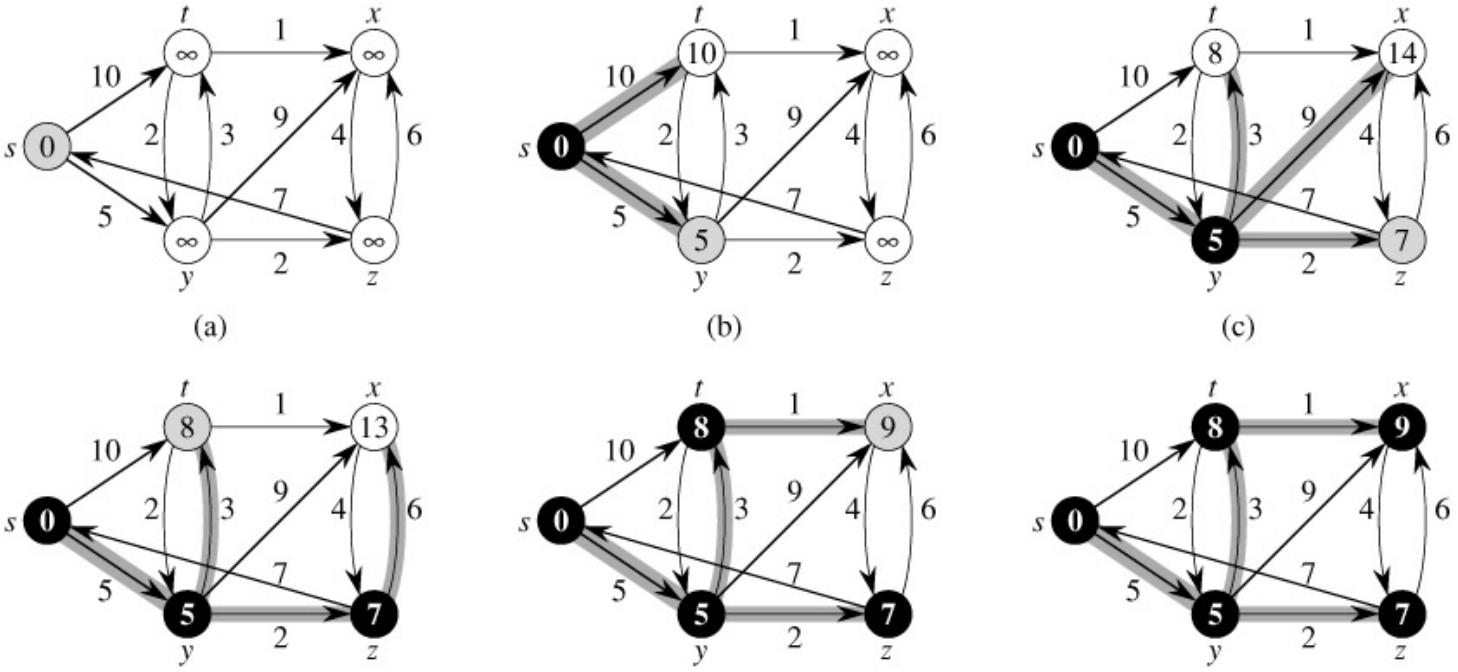
1. 最短路径：一个顶点到另一个顶点的最短权值路径。广度优先搜索算法就是一种在无权（单位权值）图上执行的最短路径算法。
2. 从渐近意义上来说，解决一对顶点的最短路径问题的复杂度与单源最短路径的复杂度相同。
3. 最短路径算法通常依赖于一种性质，也就是一条两顶点间的最短路径包含路径上其它的最短路径。这里动态规划和贪心算法的特征之一：最优子结构。

   Dijkstra 算法是一个贪心算法。DIJKSTRA 算法假定输入图中的所有边的权值都是非负的，而 floyd 算法允许输入边存在负权边，只要不存在从源点可达的负权回路。而且如果存在着负权回路，它还能检测出来。
4. Bellman-Ford 算法非常简单：对所有的边进行|v|-1 遍循环，在每次循环中对每一条边进行松弛的操作。



5. 按顶点的拓扑顺序对某加权有向无回路图的边进行松弛后，就可以在 O(V+E)时间内计算出单源最短路径。在一个有向无回路图中最短路径总是存在的，因为即使图中有权值为负的边，也不可能存在负权回路（因为它根本没有任何回路）。
6. Dijkstra 算法是一种贪心策略的算法，它的运行时间一般比 Bellman-Ford 算法要好。

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S ← ∅
3  Q ← V[G]
4  while Q ≠ ∅
5        do u ← EXTRACT-MIN(Q)
6           S ← S ∪{u}
7           for each vertex v ∈ Adj[u]
8                 do RELAX(u, v, w)
```



Dijkstra 算法在每次循环中，每次仅仅提取 d 值最小的顶点 u 是保证这个贪心策略正确性的关键核心所在。因为通过集合 S 中所有的元素都已经去试着松弛过 u 了，而非 S 中的点由于本身它的 d 值都比 u 要大，所以即使用它们中的任何一个去松弛 u，也不可能达到比现在更小的 d 值了。因此，在每次循环中选取当前 d 值最小的顶点加入到 S 集合中去一定能保证最后得到全局最优解。

7. 很多问题都可以转换成图的问题，使用最短路径的算法来加以解决的。要善于把一些看似不相干的问题转化为图的问题。

Output（代码：http://code.google.com/p/introduction-to-algorithms-notes/）：

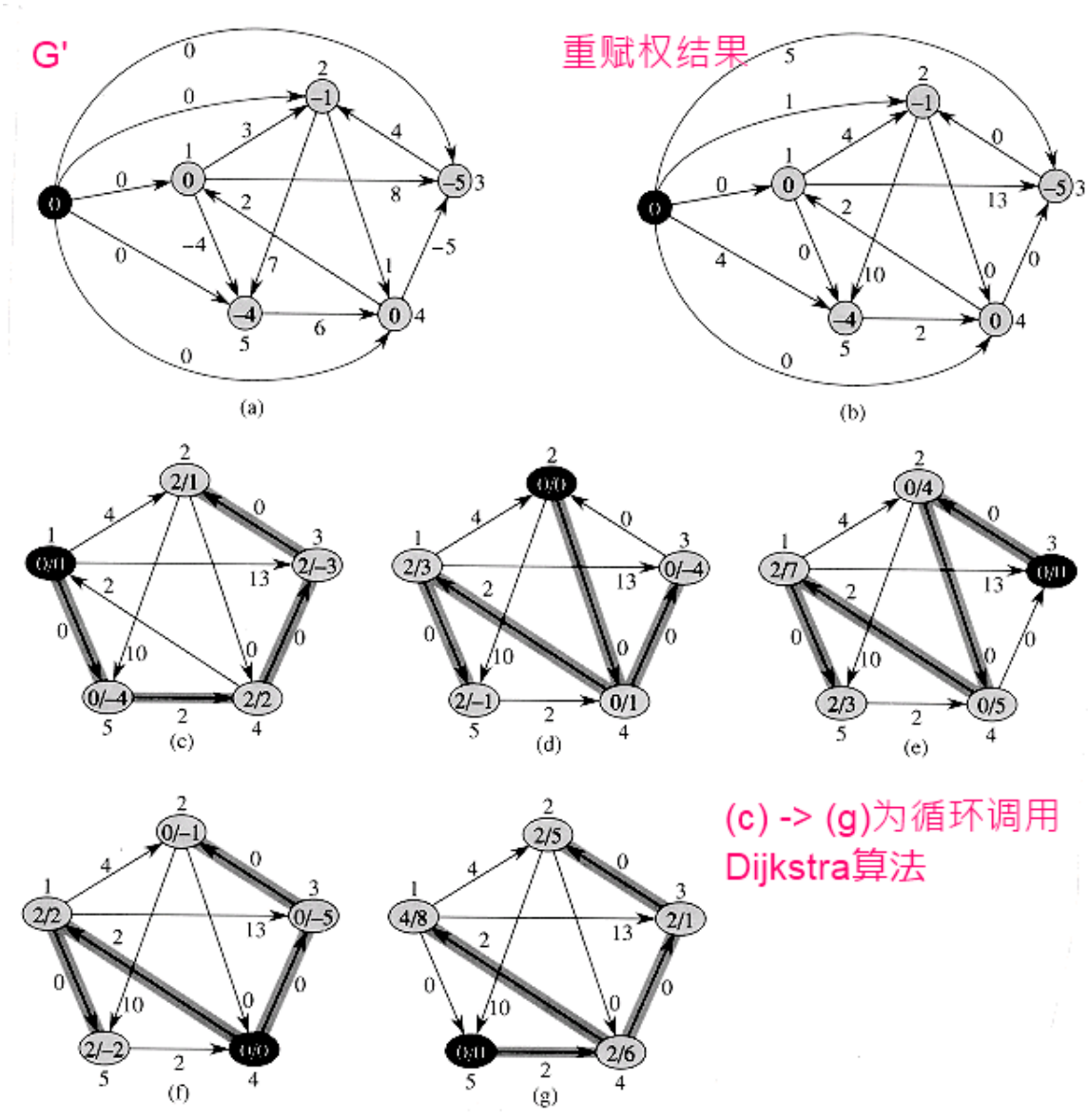
```
Bellman-Ford 最短路径
s | 0
t | 2
x | 4
z | -2
y | 7
Dijkstra 最短路径
s | 0
t | 8
x | 9
z | 7
y | 5
请按任意键继续. . .
```

第 25 章：每对顶点间的最短路径

1. Folyd-Warshall 是一个动态规划算法，运行时间为 $O(V^3)$；Johnson 算法使用了几种算法作为其子算法，运行时间为 $O(V^2 \lg V+VE)$，尤其适合对大型稀疏图。
2. 求每对顶点之间的最短路径问题很适合于动态规划算法来求解，因为它满足“最优子问题的结构”和“重叠子问题”两个特征。
3. 一个朴素的思路就是使用 $L_{ij}^{(m)}$来表示从顶点 i 到顶点 j 的，中间包含至多 m 条路径的最短路径。这种思路是朴素的动态规划思想，效率比较好，时间复杂度为 $O(V^3 \lg V)$，但是后面提及的两种算法都对这个“最优子问题的结构”进行了更多的优化。
4. Folyd-Warshall 算法的运行时间为 $O(V^3)$，它允许权值为负的边，但是假定了不存在权值为负的回路。而且它的代码是紧凑的，而且不包含复杂的数据结构，隐含的常数因子很小。因此，即便对于中等规模的输入图来说 Folyd-Warshall 算法仍然相当的

实用。

5. Folyd-Warshall 的核心在于：它改进了“最优子问题结构”，使用 dij(k)来表示从顶点 i 到顶点 j、且满足所有中间顶点皆属于集合{1,2,…,k}的一条最短路径的权值。这种限定了起始点的技巧大大的减少了实现的计算量。
6. 有向图的传递闭包：G 的传递闭包定义为图 G*=(V, E*)，其中 E*={(i,j)：图 G 中存在着一条从 i 到 j 的路径}。
   对于一个大图，即使是只需要确定是否存在路径可达都不是一件很容易的事件。解决此问题的整体思路与 Folyd-Warshall 算法一样，只是把 Folyd-Warshall 算法中的 min 和+操作替换为相应的 OR 和 AND 逻辑运算来加快速度，本质并没有区别。
7. Johnson 算法可在 $O(V^2lgV+VE)$时间内，求出每对顶点间的最短路径。Johnson 算法使用 Dijkstra 和 Bellman-Ford 算法作为其子程序。
   Johnson 算法是一个实际上非常好的算法，它使得所有的情况（可能存在负权值和负权回路）都可以使用最好的 Dijkstra 算法来达到最好的运行效率。
8. Johnson 算法在所有的边为非负时，把每对顶点依次作为源点来执行 Dijkstra 算法，就可以找到每对顶点间的最短路径；利用斐波那契最小优先队列，该算法的运行时间为 $O(V^2lgV+VE)$。
   因此也可以总结出：对于确定无负权值的图，直接循环调用 Dijkstra 算法就是求每对顶点间最短路径的最佳算法。
9. Johnson 算法使用了重赋权技术，对每一条边的权值 w 赋予一个新的权值 w’，使用新的边权值集合满足以下两个性质：
   a) 对所有的顶点 u,v，如果路径 p 是在权值函数 w 下从 u 到 v 的最短路径，当且仅当 p 也是在权值函数 w’下从 u 到 v 的最短路径；
   b) 对于所有的边 u,v，新的权值 w’(u,v)是非负的（于是满足 Dijkstra 算法的要求）。
10. Johnson 算法的简明步骤：



   a) 生成一个新图 G’，G’是在 G 上扩展一个起始点后的结果；
   b) 在 G’上调用 Bellman-Ford 算法。由于 Bellman-Ford 算法能够检测负权回路，如果存在负权回路则报告存在负权回路并结束整个算法；否则得到在 G’上调用 Bellman-Ford 算法得到的 h(x)函数；
   c) 根据 h(x)函数对 G 中的每一条边进行重赋权，使得 G 中的每一条边都是非负的；
   d) 对重赋权后的 G 进行循环调用 Dijkstra 算法，得到每对顶点间的最短路径；

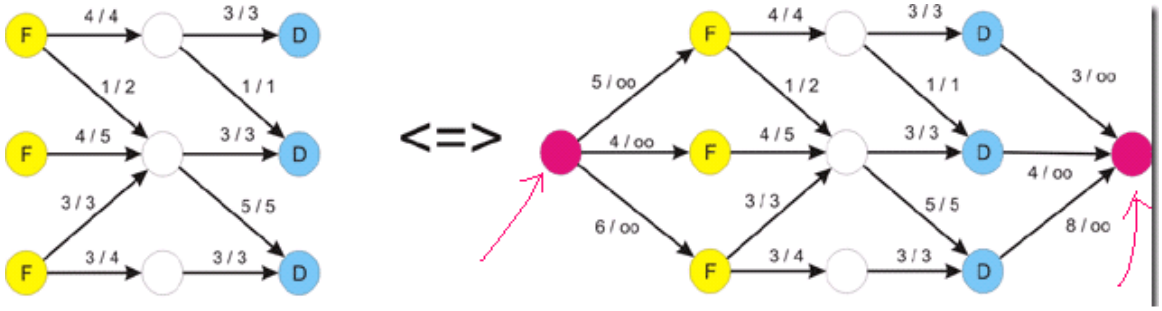
e) 对得到的每对顶点间的最短路径再根据 h(x)函数反向构造出原来权值下的最短路径值。

Output（代码：http://code.google.com/p/introduction-to-algorithms-notes/）：

```
FloydWarshall 最短路径
0   1   -3   2   -4
3   0   -4   1   -1
7   4   0   5   3
2   -1   -5   0   -2
8   5   1   6   0
Johnson 最短路径
0   1   -3   2   -4
3   0   -4   1   -1
7   4   0   5   3
2   -1   -5   0   -2
8   5   1   6   0
请按任意键继续. . .
```

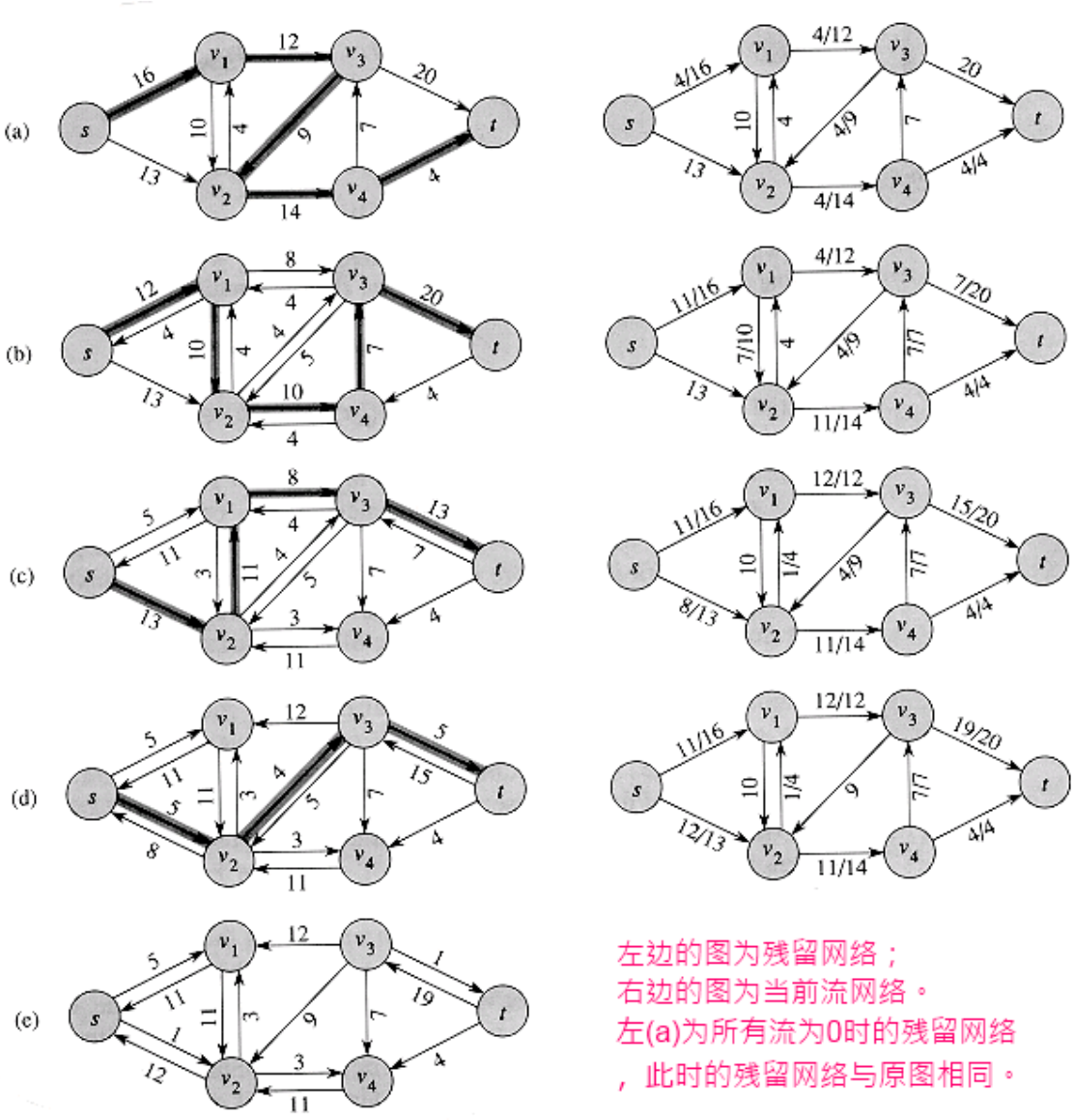
第 26 章：最大流

1. 流守恒：物质进入到某顶点的速度必须等于离开该顶点的速度。
2. 最大流问题：是关于流网络的最简单问题，它提出这样的问题：在不违背容量限制的条件下，把物质从源点传输到汇点的最大速率是多少？
3. 某个顶点处的总的净流量：为离开该顶点的总的正流量，减去进入该顶点的总的正流量。流守恒性的一种解释是这要瓣，即进行某个非源点非汇点顶点的正网络流，必须等于离开该顶点的正网络流。这个性质（即一个顶点处总的净流量必定为 0）通常被非形式化地称为“流进等于流出”。
4. 抵消：利用了抵消处理，可以将两城市间的运输用一个流来表示，该流在两个顶点之间的至多一条边上是正的。也就是说，任何在两城市间相互运输球的情况，都可以通过抵消将其转化成一个相等的情况，球只在一个方向上传输：沿正网络流的方向。
   这样：两个顶点之间至多有两条边，而这两条边至多会有一条有正的流值，另一条相应的边的流值为 0（明白这点在理解算法时是有用的）。
5. 具有多个源点和多个汇点的网络最大流问题可以转化成普通的单源点、单汇点的最大流问题（通过添加一个单一的源点和一个单一的汇点并置新添加的边的容量为无穷大来达到）。



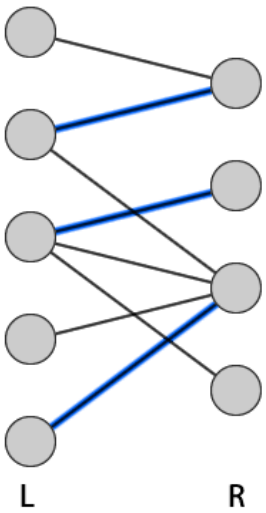
注意：新添加的所有边的容量都为“无穷大”！

6. Ford-Fulkerson 算法是求最大流的一般方法，它利用了三点：残留网络、增广路径、最大流最小割定理。
   a) 残留网络：G_f 由可以容纳更多网络流的边所组成；
   b) 增广路径：为残留网络 G_f 上从 s 到 t 的一条简单路径 p，其中 p 中所的边的最小权值为该增广路径的残留容量；
   c) 最大流最小割定理：证明了 Ford-Fulkerson 算法能够得到全局最优解“当一个流是最大流，当且仅当它的残留网络不包含增广路径”。
7. Ford-Fulkerson 算法的简明步骤：



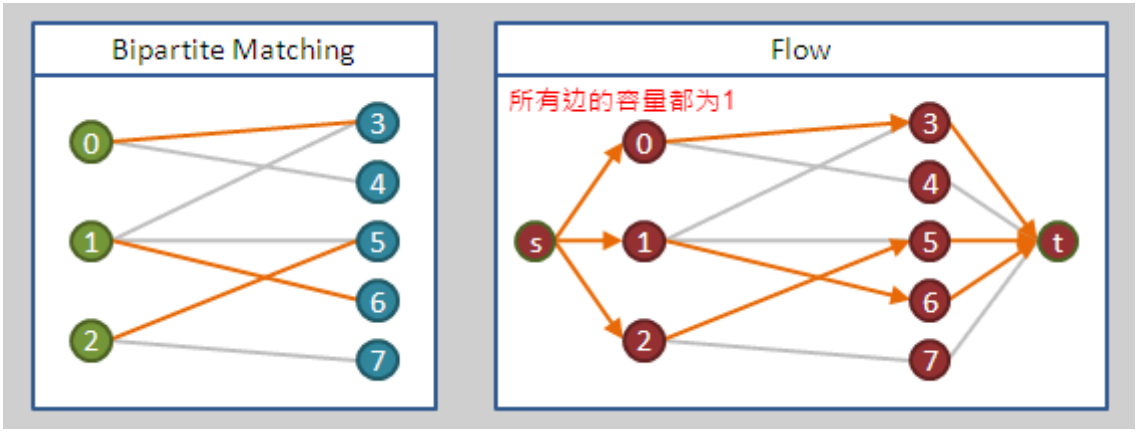
a) 初始化 G 中所有边的流为 0；
b) 计算当前图与当前流所映射的残留网络 G_f；
c) 从残留网络上选取一条增广路径并计算出残留容量，将残留容量添加到图的当前流上；
d) 循环步骤 b-c 直到残留网络 G_f 中不存在增广路径为止；
e) 此时的流即为 G 的最大流。

8. 使用“广度优先搜索”来求增广路径的 Ford-Fulkerson 算法即称之为 Edmonds-Karp 算法，这种使用广度优先搜索来求增广路径的算法能够改善 Ford-Fulkerson 算法的运行时间。
9. 最大二分匹配问题：这是最大流最重要的应用之一，并且有许多其它的问题可以归约成它，有着许多的实际应用。



例如：把一个机器集合 L 和要同时执行的任何集合 R 相匹配。E 中的边(u,v)，就说明一台特定机器 u 能够完成一项特定任务 v，最大匹配可以为尽可能多的机器提供任务）

10. 解决最大二分匹配问题：先生成一个扩展图 G’，再对 G’的每条边赋予单位流量 1，然后求出最大流，就是该最大二分匹配问题的解。



Output（代码：http://code.google.com/p/introduction-to-algorithms-notes/）：

```
FordFulkerson 最大流
23
请按任意键继续. . .
```

参考：

[1] 《算法导论》第 2 版，机械工业出版社
[2] http://staff.ustc.edu.cn/~lszhuang/alg/
[3] http://www.wutianqi.com/
[4] http://www.cnblogs.com/timebug/
[5] http://blog.csdn.net/v_JULY_v/
[6] 如有参考而未能引用的，深表感谢与歉意！