第二章：

算法的复杂度分析，也就是执行时间的分析；当然这里的执行时间是仅对代码而言，做出了简化，对于某个算法的输入数据的不同，以及从磁盘读取数据所需的时间，以及缺页和不同操作之间所花时间的差别，以及编译器的优化等因素被忽略了；

要注意时间复杂度为对数的几种常见情况；

设计算法前一定要进行足够的分析，然后再编码验证分析的结果是否与实际运行的结果一致；

如果一个算法用常数时间将问题的大小削减为其一部分（通常是1/2），那么该算法就是O（logN)；另一方面，如果使用常数时间只是把问题减少一个常数（如将问题减少1），那么这种算法就是O（N）的；

对分查找以及欧几里得算法（长除法）等，都是O（logN）的算法；

第三章 表，栈和队列

ADT的概念，抽象数据类型，跟C++中的类的概念比较相似，都强调复用和封装性；将具体实现和功能分开；

表的链表和数组实现，链表中单链表，双链表和循环链表，可构造复杂的数据结构；

栈的应用，符号匹配（括号等成对的符号），后缀表达式，以及中缀表达式向后缀表达式的转换；

实际上队列，栈都属于表的范畴，只是栈的操作只在栈顶，而队列的插入和删除分别在表的两端；

这三种数据结构非常基本，栈的使用使过程语言成为可能；同时也解释了递归的内部实现；当然递归也可以展开，通过引入一个栈的数据结构；

尾递归可以通过改变递归时要用的参数以及goto语句来进行替代，加速的作用；

第四章 树

树的概念，深度，高度，路径，边，长度等的意思；

树的实现：每个节点的结构体如下：

typedef struct note \*ptrtonode;

struct node{

elementtype element;

ptrtonode firstson;

ptrtonode nextsibring;

}

一个指针指向第一个儿子，另一个指针指向下一个兄弟，对每个节点，这两个指针分别指向另外两个节点；向下建立，每个节点与其他节点的关系要么是父子关系，要么就是兄弟关系；

树的遍历：前序和后序；二叉树还有中序遍历；左，节点，右；前序是先处理跟节点；后序是先处理子节点；分别对应中缀表达式，前缀表达式和后缀表达式；

二叉树：主要应用是在编译器的设计领域，搜索当然也是重要的应用；

对二叉查找树（有序）的操作：

重复元的插入可以通过在节点记录中保留一个附加域以指示发生的频率来处理。这使整个的树增加了某些附加空间，但是，比将重复信息放到树中要好（使深度变大）；

任意节点的的深度的期望值为O(logN)；在删除有两个儿子的树节点时，选择用哪个节点来替换被删除的节点，是左子树的最大值，还是右子树的最小值？如果只选择一边，那么在进行多次插入和删除后，树就可能不平衡，也就是某一侧子树的深度要远远大于另一侧；要保持平衡，则需要随机的在两种可取情况中选择；这涉及到二叉查找树是否等可能出现的问题；

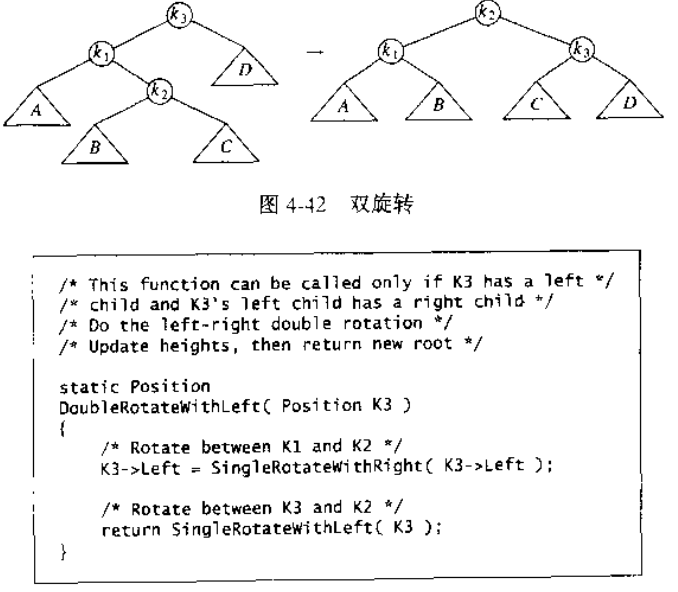
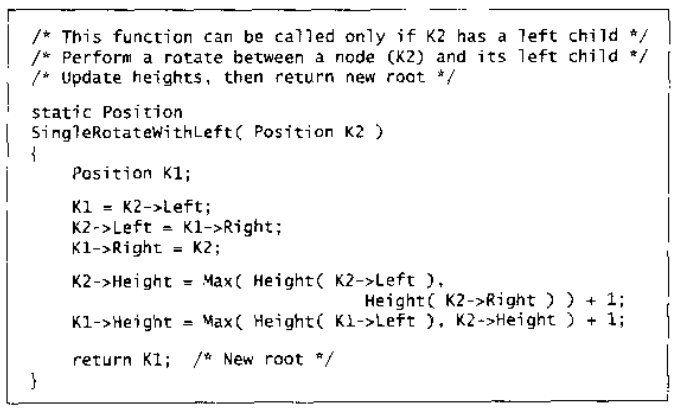
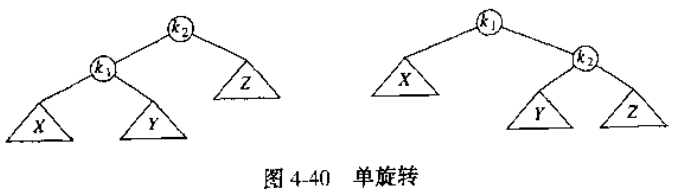
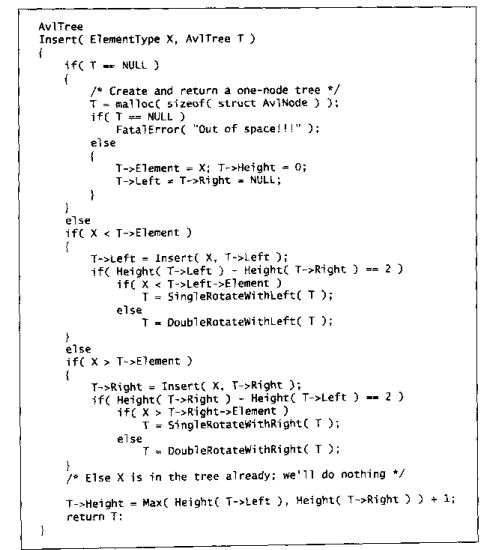
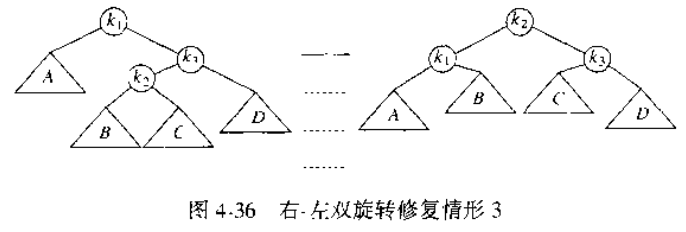
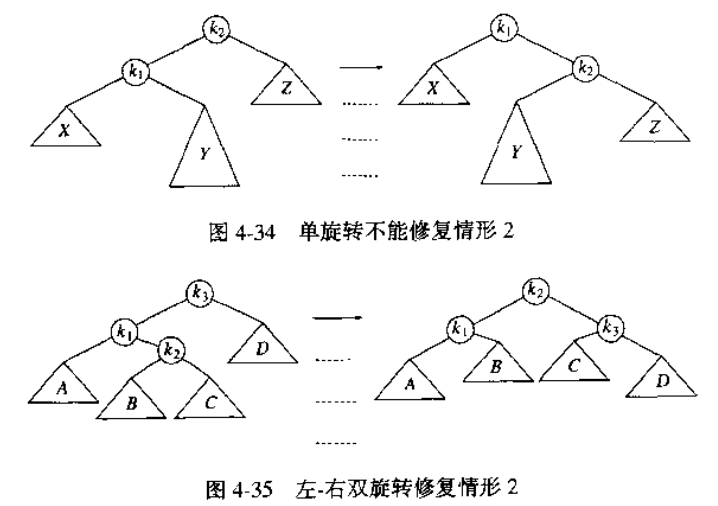
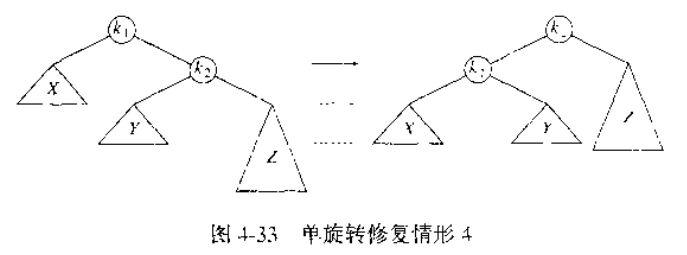
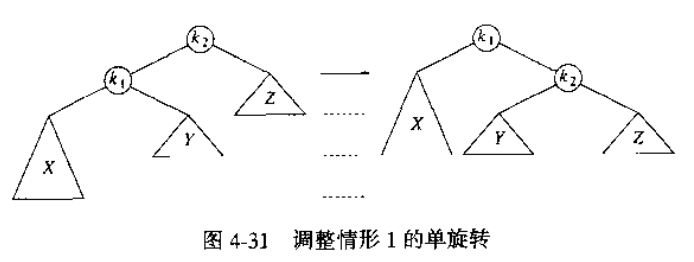
树还面临着另一类的平衡问题：

如果根节点的值范围很极端，那么树就会严重偏向一侧，深度就会加深，很多操作就不再是O(logN）的复杂度了。那么怎么来调整呢？一种是添加平衡条件，就是树节点的深度有个阈值；另一种就是放弃平衡条件，允许树有任意深度，但每次操作之后就会对树进行调整，这样就会使后面的操作更加高效（伸展树）；

AVL树：带有平衡条件的二叉查找树，其平衡条件是：左子树和右子树的高度最多差1的二叉查找树；

AVL树的旋转操作：

单旋转和双旋转两种操作，主要用于解决在插入操作时，AVL树的平衡性被破坏的情况的处理；



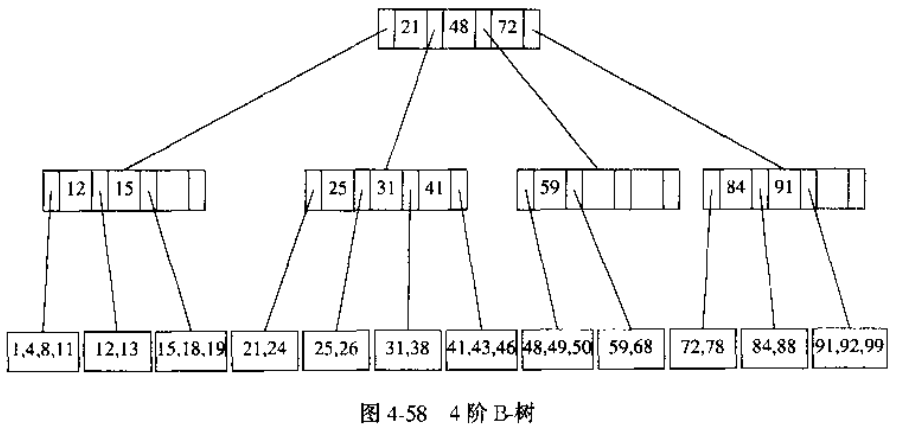
原理说的很复杂，没想到程序实现起来这么简单。主要是那个插入的递归程序的编写，要深入的理解；

伸展树：

它是在二叉树的基础上，每一次对树节点的访问，都将要调整树的结构，使被访问的树节点被换到根节点，也就是利用访问的局部性原理，同时，采取合适的方式，使调整后的树近似于平衡树的结构（展开的方式），这样，在连续M次访问，其复杂度为O(MlogN)；调整树的结构，实际上就是改变一些指针的指向，而且是针对当前节点的分步进行；相比于平衡树，它的操作的编程更简单，因为没有上述的那四种情况，而且节点也不需要维护高度信息；

B树的概念：

M阶B树：所有的数据都在页节点上；



树高度的增加是有插入操作引起的，但插入操作首先会查询页节点，看是否有空间能够容纳插入的数据；如果没有，那么就会将其父节点进行分裂（按阶的限制），添加页节点或者是中间节点，此过程递归进行，直到分裂根节点；同时，在删除一个节点的时候，可能涉及到合并两个页节点或中间结点；这都是为了使树紧凑，访问时间得到节约；

这个数据结构可以用来建模磁盘的组织，按这种节点组织方式，将根节点或者是一级节点放在内存中，而其他节点放在磁盘中，然后进行访问；具体磁盘中数据的组织方式，需要参考其他内容；

树的优点在于，各种操作的访问时间大致在O(logN)的级别；而且对于查找树，其建立的时候就是自然排序的，这一点也很有用；

第五章 散列（hash table）

以常数平均时间执行查找，插入和删除操作；对需要排序的操作效率很低；主要就是一个数据表，表中存储的值与索引之间建立了一个对应关系，这是与数组的最大区别；这个对应关系就叫散列函数；

主要涉及这么几个问题：一是散列函数的选择，原则是尽量覆盖整个表，同时避免冲突；二是表的大小，最好为素数，同时在冲突解决中，要留足空间；三就是冲突解决；

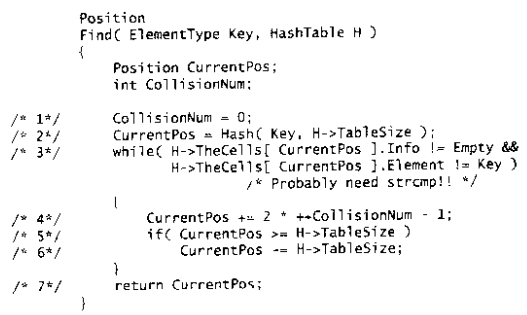
冲突解决有以下几种方法：

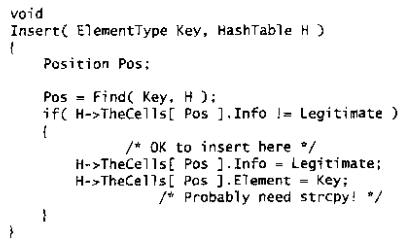
分离链表法，也就是表中的每个元素实际上是链表头，相同值的单元添加到对应的链表中去；这种方式的缺点在于每次添加元素需要分配空间，比较慢，相当于同时实现了两种数据结构；尽量使表的大小与需要存储的个数差不多，也就是链表中的元素大致为1个；当然，如果不涉及到删除操作，表头也是可以不要的；

开放定址法：表给足空间，一旦冲突，就在表中寻找空闲单元，然后存放元素；那么如何探测到空闲单元呢?线性探测法，也就是在散列函数中添加一个线性函数；那么需要注意的是，表中填充的元素越多，那么各种操作的性能就会越变越坏，它存在一个一次聚集的问题，就是在探查的时候，相隔不远的单元都是线性增加，那么它们靠近的可能性就更高，分散度就拉不开，那么在下一次探测的时候，就会走更多的路；所以表的大小最好要超过两倍大小；二次探测法：是前一种方法在分散度上的改进，很好理解；问题在于平方的变化是否能够完全覆盖表中的空闲单元呢？理论上应该是可以的；也要注意表的大小；

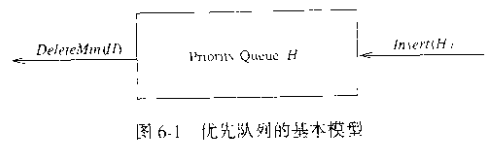
还有一个再散列，就是说当元素在表中的填充超过一半的时候，就建立一个更大的表，将旧值放到新表中，然后在新表中进行探测；这样可以根据存储元素的多少动态的扩大表的规模，节省空间；就是在再散列的时候会浪费时间；

可扩散列，感觉有点译码器的选择信号位数可变的那么一个概念，和磁盘目录的组织似乎联系比较紧密；



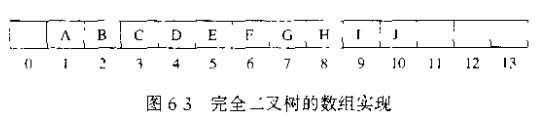
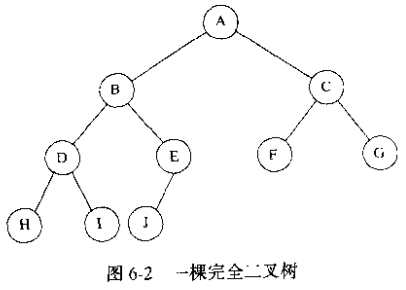


第六章 优先队列



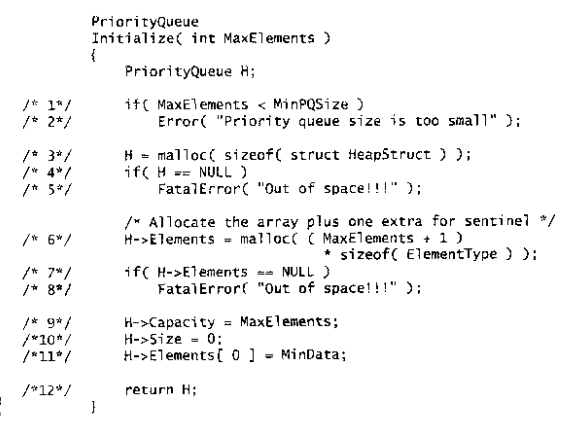
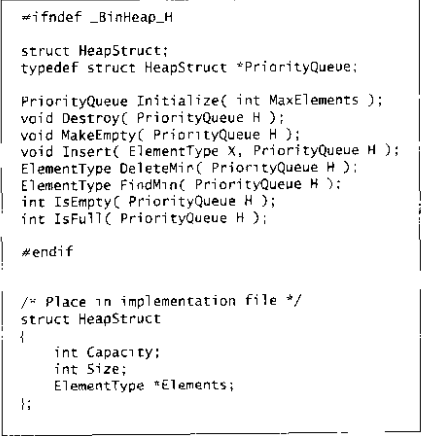
优先队列的实现由很多种方式，链表实现，插入简单，花费常数时间，删除最小值时为O（N）；也可以用二叉树实现，都花费O（logN）的时间；更常用的是二叉堆；

堆是一棵被完全填满的二叉树，有可能的例外是在底层，底层上的元素从左到右填入；这是结构性质；这种树结构可以通过一个数组来实现：



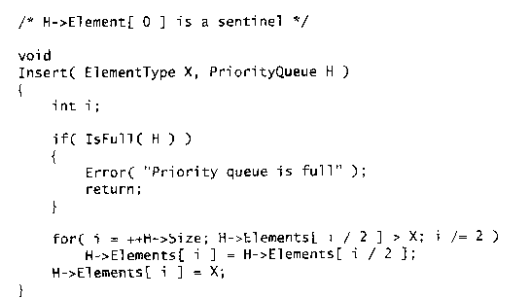
对于数组中任意位置i上的元素，其左儿子在2i处，右儿子在2i+1处，其父节点在[i/2]处；这种方式的问题就是要提前知道堆的大小；

第二个性质就是堆序性质，也就是在堆中的元素是带有顺序信息的，尽管不多；根据需要，这种顺序可以是树（子树）根节点的值小于或者是大于儿子节点的值；



基本操作：

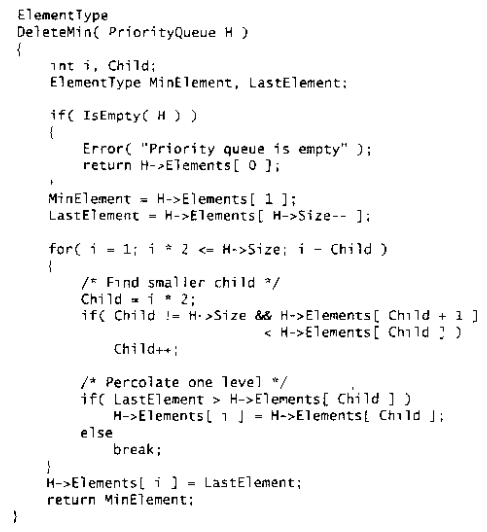
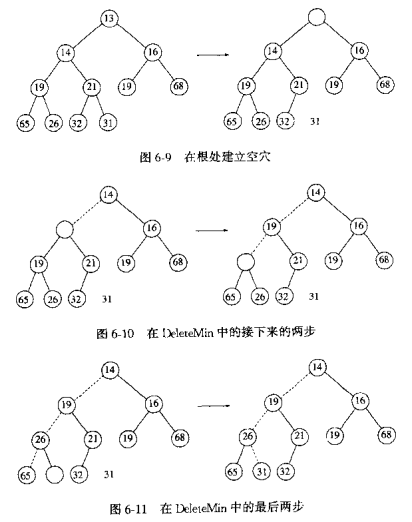
插入：在下一个空闲位置（儿子节点从左到右填充）创建一个空穴，如果X放在该空穴中而并不破坏堆的序，那么插入完成；否则就会上滤，将父节点层层往下移动；



最坏情况的复杂度为O(logN)，但实际上终止的要早很多；

删除最小元：

根节点就是最小元，但删除之后还需要用最后一个节点来填充这个空穴，同时满足堆序性；这就涉及到一个下滤的过程；



最坏情况为O(logN)，下滤一般会到底，因为填充的元素就来自树叶；

还有一些操作，比如降低/增加关键字的值，比如进程调度中改变优先级，需要知道某个进程在堆中的位置；同时在改变了关键字的值后，需要上滤或下滤，以维持堆序；

构建堆：这里有两种方式，一是通过插入来建立，一是先填入数组，然后下滤；复杂度都为O（N）；

应用：在操作系统中调度器的设计，事件模拟，选择问题，排序等；

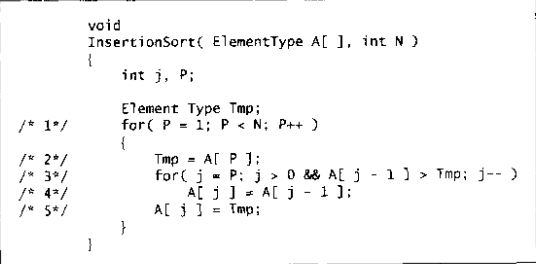
左式堆，斜堆的概念；

还有一个二项队列的概念，比较复杂；

第7章 排序

内部和基于比较的排序：

1. 插入排序：

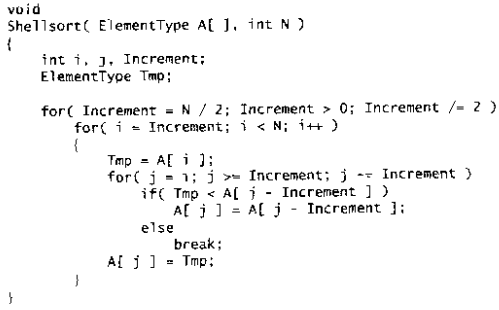


插入排序最坏情况下的复杂度为O（N2），相比于冒泡和选择排序，它是一种隐式的相邻元素的交换；通过逆序对数量的分析可以看出，N个互异数的数组的平均逆序数是N(N-1)/4，这也是一个下界；

通过交换相邻元素进行排序的任何算法平均需要O（N2）的时间；

1. 希尔排序

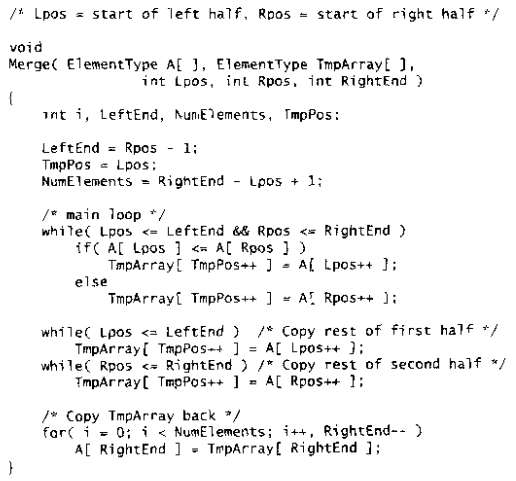
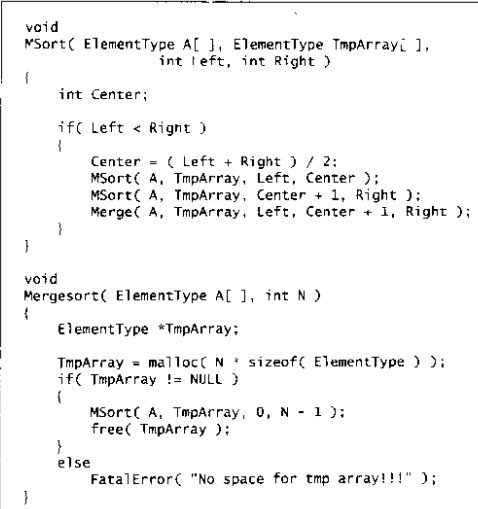
它实际上是一种细分的插入排序，细分的标准是一个增量序列；



不同的增量序列，得到的性能不同；最坏的情况是平方关系；有一个到普遍认同的增量序列，其性能比较好；

1. 堆排序
2. 归并排序

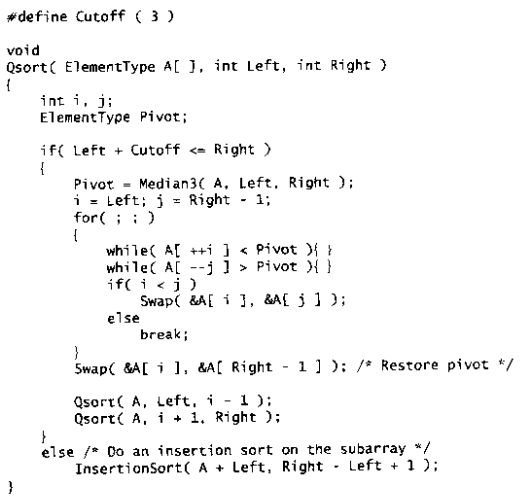
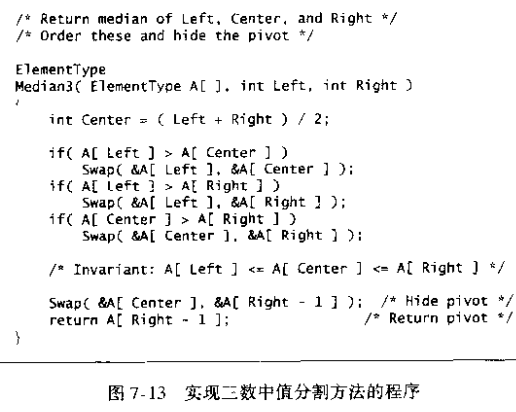
分治的策略，代码如下，其复杂度为O(NlogN)；



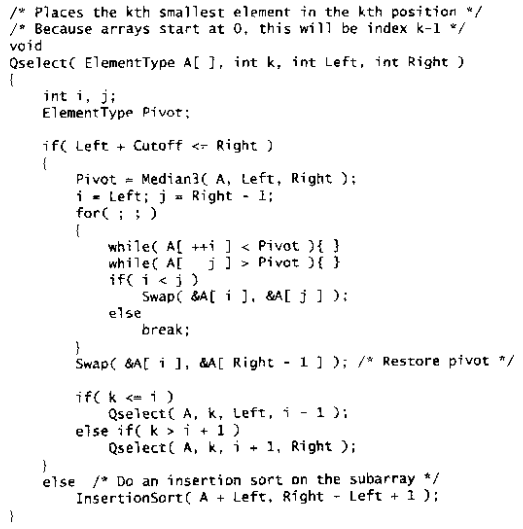
1. 快速排序

也是一种递归的实现方式。选择一个元素（枢纽元），然后根据选择元素的值大小将数组中的元素分成两部分，排序后就是左边子数组，枢纽元，右边子数组这样的结构。那么问题在于，枢纽元如何选取，以及怎么根据枢纽元的大小高效率地得到这两个子数组；枢纽元的选取一种方法是在第一个，中间元素和最后一个元素中选取中间值，其程序如下所示：

排序程序也给出来了，要注意的是对边界的处理；



由上面的理论，这里给出了一个快速选择的程序：



桶式排序：

使用了比简单比较更为强大的操作，就是直接索引，这一点非常重要，所以它可以达到线性的复杂度；

外部排序：

外部排序的中心思想是合并，什么意思呢？也就是主存中一次性容纳不下这么多的数 ，以磁带为例，那么就需要就排序的对象分割成小块，然后进行排序，完了之后写回磁带，然后将剩下的部分排序，然后将排序的结果进行合并。那么这里与程序实现的算法之间的关系又有什么样的联系呢？？

联系就在于，这个时候数据是不能直接内存寻址的，那么在设计算法的时候，就要考虑到这种访问外部存储的次数，以及外部存储本身的特点，比如磁带的顺序访问，这样就可以避免一个算法中效率很低的对存储单元的访问顺序，比如对磁带的访问，就要尽量操作邻近的元素；更好理解的就是，程序是面向内存编写的，那么把内存和外部存储进行比较，就可以得出存储本身的特点对程序的影响；

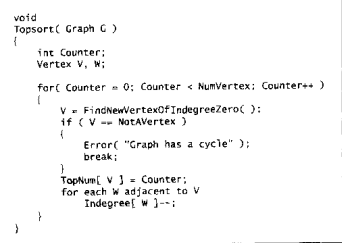
第九章 图论算法

图的定义：G=(V,E)，v为顶点的集合，e为边的集合；边实际上就是顶点对，根据顶点对是否有序，可分为有向图和无向图；对于有向图，强连通和弱连通的概念；

图的表示：

可用一个二维数组来表示，也叫邻接矩阵，但问题在于若邻接矩阵比较稀疏，就会浪费大量的空间；另一种方式叫做领接表，也就是对每个顶点建立一个链表，将与自己邻近的节点填入表中；

拓扑排序：是对有向无圈图的顶点的一种排序，它使得如果存在一条从Vi到Vj的路径，那么在排序中Vj出现在Vi的后面；



最短路径算法：

分为几种情况，不带权值，那么最短路径实际上就是边的数量；带权值，以及权值中有负数的情况；

网络流问题，就是图中的边上的流量已经给定，现在要计算在两个节点之间传输的最大流量的问题；流量可在不同的路径分支上分摊；

计算最小生成树的问题：

对于无向图来说，最小生成树是指，由该图的那些链接所有顶点的边构成的树，且其总价值最低；最小生成树的存在当且仅当G是连通的；

最后就是欧拉回路问题的求解；

关键是要知道一种数据结构中涉及到的问题，至于具体的实现细节，则在用到的时候再来仔细研究；

NP完全问题：

P对应于确定性多项式算法，而NP对应于非确定性多项式算法；P很好理解，就是对一个问题，我们能够找到一个算法，它能在多项式时间内求解；而NP问题不同，对每一个NP问题可能的解，我们可以用一个花费多项式时间的算法来验证该解是否正确，但要直接解出或搜索出该问题的解，在多项式的时间复杂度内，还不确定存在这样的算法；NP完全问题是指，可以用多项式复杂度的算法验证所有可能解的NP问题；同时，任何一个NP问题都能通过一个多项式时间算法转换为某个NP完全问题，也就是说NP完全问题是一类可以相互转换的问题；

目前对于NP问题的求解是通过搜索验证来实现，但其复杂度呈指数关系，在输入规模很大时，几乎不可用；

常用搜索算法：

近邻法，先甜后苦；

插入法，先苦后甜；

模拟退火算法:实际上退火过程只是用来产生那么接受概率，取的是退火过程渐趋稳定的这样一种想法，实际上就是引入了一个概率产生函数而已，不要被名字吓到；具体参见本文件夹中的pdf文档；

遗传算法，神经网络算法：也参见pdf文档；

后三种方法属于优化算法的范畴，实际上也就是通过不断的迭代搜索，然后去逼近答案；这个线性代数和矩阵分析中学到的牛顿法，还有其他几种逼近方法是类似的；现在明白这些逼近思想还是相当有用的啊…

第十章 算法设计

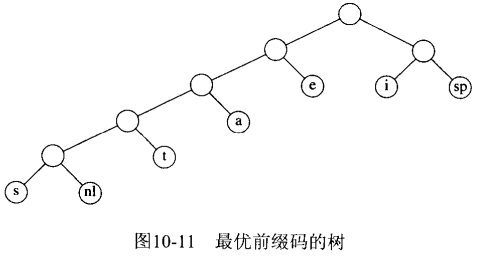
1、贪心算法

贪心的意思就是每一步都要求最好的结果，忽略对全局的考虑；得到的是局部最优结果，当然也可能是全局最优；一些例子如下：

1）、调度问题：将执行时间互异的一堆任务分派给处理器执行，调度有两个问题需要考虑，响应时间和吞吐量；响应时间就是让执行时间短的任务尽早执行，吞吐量就是这批任务尽可能早地执行完；对单处理器，要满足响应时间，调度就需要每步贪心，选择执行时间最小的任务，而吞吐量是固定的；对多处理器，响应时间的处理与单处理器类似，只是任务指派到不同的处理器中，而吞吐量的优化就是个NP完全问题，怎么在不同处理器中分派任务呢？只能对给出的结果进行比较，验证，直接产生似乎太难；

2）、郝夫曼编码：它是一个算法，用于文件压缩。

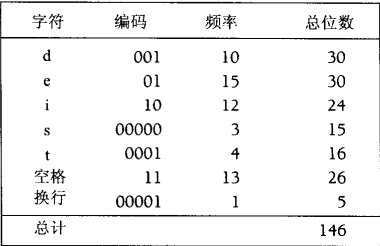
中心思想就是，对于一个文件，根据其中字符的出现频率来对其进行编码，当然字符的编码就不是固定位宽的了。那么这个编码的产生可以利用二叉树，左右儿子分别表示0或1，这里给一个图如下：



各个字符出现的频率如下表所示。可以看出，出现频率越低，在树中就越深，字符只在树叶上，如果在中间节点上，就会编码覆盖；

那么郝夫曼做了什么呢？他就提供了这么一个算法，来建立上面的树。算法很简单，每个字符出现的频率已知，把它看成一个单节点的树，然后找出森林中叶节点频率出现最低的两棵树，然后作为左右子树建立一个新树，迭此过程即可；

这种压缩是两趟算法，首先得扫描文件，得出各字符的出现频率，然后再扫描一次完成字符编码的替换；在压缩文件中需要给出字符编码以用于解码；



3）、近似装箱问题

装箱问题就是指，给定很多箱子，大小不一，往仓库里放置；仓库的大小是固定的，确定算法，目标是怎么放置使用的仓库最少；

这里有个概念，就是联机算法和脱机算法；联机算法就是给一个输入，处理一个输入，是动态的，好比两个计算机之间合作通信来实现这个算法；而脱机算法就是，在处理之前，所有的输入都已经得到了，这样就可以对整体的输入做出一定的处理，再进行操作；

那么很容易知道，联机算法很难有最优的解，因为对输入的不同序列，本应该做出不同的处理，但联机算法不可能兼顾到这么多不同的处理序列；对装箱问题，联机算法有几种实现方式：下项适配，就是来一个装一个，如果仓库空间不够，就换下一个仓库；首次适配：放不下的时候，它会依次扫描，直到找到一个能放下它的仓库；最佳适配：放不下的时候也会依次扫描，但只会放到那些刚好使仓库填满的仓库中；

脱机算法，就可以进一步优化。比如首次适配递减和最佳适配递减，就是先将箱子的大小排序，先装大的，再适配小的；

2、分治算法

在递归调用中至少有两个分支。把原问题分解为几个子问题来解决，最后合并结果；关键在于如何分割，同时在合并的时候如何不增加开销；

3、动态规划

避免在递归调用过程中出现的冗余计算，把中间结果用一个表（或者变量）存储起来；

4、随机化算法，算法中的决策至少一次是有随机过程决定的；

5、回溯算法：告诉公路收费点重建问题；

后面几种讲得比较泛泛，用到了再仔细查阅；教材中有很多参考文献值得一看；

摊还分析实际上就是分析算法在连续多次操作时的时间界；

本书中最后还介绍了一些高级的数据结构，大都是之前讲述的那些基本数据结构的变体，用到时再来仔细分析；