**软件设计与开发实践Ⅰ报告**

**王叶 1140310311**

**1403103班**

目录

[线性结构 2](#_Toc484178328)

[跳表的实现 2](#_Toc484178329)

[KMP算法 2](#_Toc484178330)

[优先队列 3](#_Toc484178331)

[树型结构 3](#_Toc484178332)

[线索二叉树 3](#_Toc484178333)

[二叉树与森林的转化 3](#_Toc484178334)

[K叉哈夫曼 4](#_Toc484178335)

[图型结构 5](#_Toc484178336)

[无向图的双连通 5](#_Toc484178337)

[有向图的强连通 6](#_Toc484178338)

[最小生成树 7](#_Toc484178339)

[最短路径的算法优化 7](#_Toc484178340)

[查找和散列 8](#_Toc484178341)

[红黑树 8](#_Toc484178342)

[散列查找 9](#_Toc484178343)

[内排序 10](#_Toc484178344)

[快速排序及优化 10](#_Toc484178345)

[线性排序算法 10](#_Toc484178346)

[大作业 AC 自动机 11](#_Toc484178347)

[原理及简单实现： 11](#_Toc484178348)

[优化及应用： 12](#_Toc484178349)

[测试及实现结果： 13](#_Toc484178350)

# 线性结构

## 跳表的实现

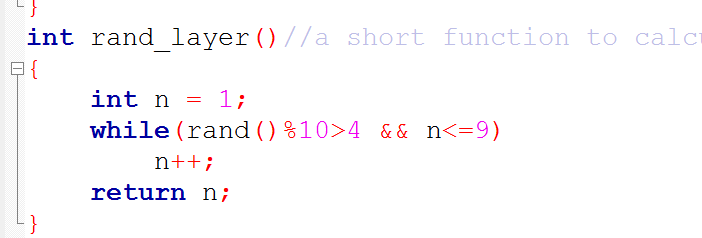
跳跃链表是按照层次建造的。最底层是一个普通的单向线性链表，而且其中的元素是有序的。除了最底层之外，之上的任何一层都是下一层链表的一个子序列。至于高层元素的选择，是从下一层链表中按照一个常数的概率选出一定数量的元素。所有链表的每个节点都含有指向本层内下一个元素，和指向下一层同一个元素的指针。

跳表的搜索通过高层的元素可以快速定位到下层的元素。从最高层的链表头开始（假设底层链表元素是升序的），沿着链表向后搜索，合适时向下层找。直到找到等于键值的元素，或者发现元素不存在于表中。跳表的插入和删除可以通过对各层的链表操作实现。

**问题：保证跳表的性能？**

跳表的建立基于随机过程，所以跳表并不能给出最坏情况下的性能保证，仍然存在产生非常不平衡的跳表的可能性。如果每层都以概率 p 抽取元素，则在平均情况下，跳表会消耗 O(n)的空间；搜索，插入，删除的时间为 O(logn)。在最差的情况下，跳表会 使用O(nlogn)的空间；搜索、插入、删除的时间为 O(n)。跳表的层数越高，跳表消耗的空间就会增加，但是减少层数或者减小p值，则会增加搜索的时间代价。

比较随机的决定插入数据所在的层：越高层的的概率越低，我们希望每一层都是下一层元素个数的1/2。



跳表具有和平衡树一样的渐进效率，因为使用链表，其维护更加轻松速度更快。数据规模小的情况下其优点比较突出。

## KMP算法

KMP的算法核心是针对模式字符串进行初始化，产生一个失配函数。从表中的信息可以知道在此处失配后可以后移几位，避免了重新检查已经匹配的字符，效率就可以得到提高。

其渐近效率为 O(n)。比较依赖与模式串和待匹配文本，适合重复比较多的字符串，在几乎没有重复的主串比较大的最差的情况下也有可能退化为暴力查找的方式，而且由于操作比较复杂，甚至有更低的效率。

对此还可以使用坏字符、好前缀等改进。BM算法就有更好的平均性能。

## 优先队列

使用最大/小堆来实现，构造的时间O(nlogn)，插入和删除为O(logn)。

除了二叉堆之外，使用二项堆可以构造一种可以快速合并两个队列的优先队列。 斐波那契堆的插入、提取、修改元素优先级等操作具有分摊常量时间复杂度。

有一些其他的优先队列实现可以在渐近的意义上具有更好的效率。比如Brodal Queue在最糟糕的情况下，仍然可以在O(1)内插入、找最大/小、合并两个队列，在 O(logn)内完成删除最大/小元素，和删除一般元素的操作。

# 树型结构

## 线索二叉树

在二叉树中可以非常方便地找到某个节点的父节点和子节点，但是经常需要找到元素的前驱或者后继元素的操作在二叉树中是很困难的，一般需要通过遍历算法才能动态获取。一种直观的解决方法是给每个节点都增加分别指向其前驱和后继的两个指针。虽然这种做法会额外消耗非常多空间，但是时间性能会得到提升。

在普通的二叉树中，所有的叶子节点的子节点都是空，其空间没有被有效利用。线索二叉树中如果左子为空，就可以将该指针指向该元素的后继，如果右子为空，则可以将该指针指向该元素的后继，节省了一些空间。只需要额外状态表示指针的具体含义，是哪种情况下的前驱后继，或者拥有多个指针，表示当前使用的有效的是哪两个。

二叉树的线索化过程可以通过对遍历算法稍加改造即可得到，在遍历过程中保存上一个遍历到的点，如果遍历到的下一个元素无左子，则将该指针指向其前驱。如果某个节点无右子，可以在遍历到下一个元素之后将其的右子指针指向其后继。将元素进行线索化之后，就可以很容易用非递归的形式对树进行遍历了。并且速度比递归地遍历，或者使用辅助数据结构的非递归遍历形式的速度都要快。

## 二叉树与森林的转化

**问题：如何实现来回的转换？**

任意一棵树都可以转换为二叉树。对于一个普通树的节点，转换为一个二叉树的节点，二叉树结点的左子指向原节点的长子节点，右子指针指向其兄弟节点，就可以用二叉树表示一棵二叉树。在这棵二叉树中向左走相当于在原来的树中向下走一层，向右走相当于在原树中走向其兄弟节点。

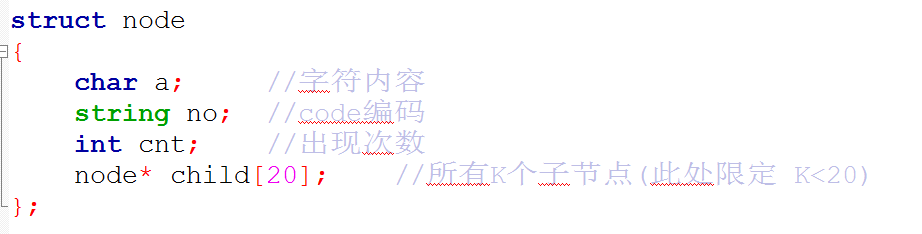
普通树向二叉树的转换可以这样直观地表示:在所有兄弟节点之间加一条线；对于树中的每一个节点，除了指向最左子的指针，删除所有指向子节点的指针。原树的树根也是新树的树根。根据生成的树调整树的层次，即成为一棵二叉树。由于根节点没有兄弟节点，所以这个过程生成的二叉树的树根一定没有右子节点。根据这个性质，实际上可以将普通的森林转换成一棵二叉树。只需要将森林中的每一棵树都转换成二叉树，按照一定顺序将后一棵二叉树的根作为前一棵二叉树的右子，最后形成一棵二叉树。

实现了以下功能，树、森林、二叉树之间随意的转换可以通过函数的组合使用完成。



## K叉哈夫曼

* 数据结构



因为K的值不固定，且子节点是父节点左起第几个节点决定了子节点的编码，所以使用数组这种方式来确认子节点的位置。数组的大小限制了本程序中K的最大值。

* 计数、排序及调整

此处我首先创建了一个足够大的数组，计数时不需要分辨节点是否出现过，而是用其ASCII确定节点的位置（类似HASH的思想）完成计数。

计数完成后排序，降序排列，未出现的字符的节点便自动后移。由于K叉哈夫曼的思想是每次取k个拥有最小cnt值的子树合成一棵新子树，直到只存在一棵子树，总叶节点数应该满足N=a\*（k-1）+ 1，其中a是整数。所以一开始可能需要补一些节点，此时只要根据(N-1)%(K-1)的余数对N进行调整即可，数组中一部分计数为0的点便自动被加入计算中。

**问题：具体实现中的步骤简化**

* 合并建树及实时排序

取有效子树的后K个进行合并并计算和，新节点从数组末尾开始向前寻找合适位置（时间复杂度低），然后重复取后K棵子树直到只有一棵子树。

* 编码

子节点的编码是父节点编码拼上子节点在父节点child数组中位置所对应的字符。

输出时避开一开始为了满足K叉条件增加的计数0的节点。

**提升部分：**

由于进制是任意的，编码使用的字符是可以修改的。哈夫曼可以生成最优前缀码。

哈夫曼树要求事先知道所有字符的频率，而且生成的编码长度不一定相同。在不能预知所有字符频率的情况下可以选择动态哈夫曼，但是维护过程繁琐，效率低下，实际很少用到。

实际解编码中常常用到动态版本的算术编码（不根据独立的字符进行编码，而是将整个文本作为整体进行编码，原始数据编码成一个数字，是一种entropy coding。压缩比自然比哈夫曼更优）。

哈夫曼编码的思想，参与编码的元素权重常常指出现的频率，但其实只需要元素中存在一个全序关系即可，由此可以得到哈夫曼模板算法。

变种：

1）增加编码最大长度的限制下的最优前缀编码问题，使用并包算法解决。

2）原始的哈夫曼编码问题中假定所有用于编码的字传输成本一样，差别在频率，如果使用不同传输成本的编码字，要求总成本最小，称为不同成本的哈夫曼编码，目前还没有该问题的很好的解决方法。

3）被编码字符中有某种字典顺序，要求最优前缀码也满足某种字典序，生成编码树为最优字典序二叉树。

把当前的编码存储起来，则在下次只需要读取编码，和使用该编码的文件就可以还原出encode前的文件内容。这个过程中不需要重新建立Huffman树，只需要维持一个数组或者map即可。

# 图型结构

## 无向图的双连通

连通的无向图中，如果存在一个节点，删除之后使原图分为两个连通分量，则这个节点称作割点。如果一个连通图中没有割点，则这个图称为双连通图。因此，判断一个连通无向图是否是双连通图的问题可以转化为求割点数量的问题。

一个图的连通性很容易用遍历的方式确定。对此一种直观的想法是，每次从图中删除一个点，再判断图的连通性，直到把所有的点都试一遍。判断一个图连通性的时间复杂度是 O(n)，因此总的时间复杂度为 O(n^2)。

**问题：有没有时间复杂度更低的算法？**

我们希望时间复杂度能够得到优化，一个解决此问题的算法是由 Robert Tarjan 提出的 Tarjan 算法。

从一个连通图的任意一个节点出发，进行深度优先搜索，到达未访问过的节点时经过的边叫做树边，到达已访问过的节点的边称作回边。则所有树边可以构成深度优先搜索树。树中有且只有两类的节点可能成为割点：

* 根节点如果只有两个子节点，则是一个割点。它是非常好判断的。
* 对于非根的内节点，如果其子孙节点没有通向该节点祖先的回边，则删除该点之后其所有子孙节点和祖先节点之间就不连通了，所以该节点是一个割点。对于非根内节点，使用 dfn[u]表示 u 在深度优先搜索中被遍历到的次序，用 low[u]表示从 u 或者 u 的子孙节点通过回边能够追溯到的最早的祖先节点。在遍历过程中计算：

Low[u] = min{ low[u] , low[v] }

min{ low[u] , dfn[v] }

Tarjan 算法需要将所有待处理的节点都压入栈中，当回溯的时候由于栈顶元素的子孙节点已经处理完毕，此时可以判断栈顶元素是否是 一个割点。 对于第二种情况，当(u,v)为树边且 low[v] >= dfn[u]时，节点 u 就是一个割点。

**这种算法的时间复杂度为O(n)，只需要对图深度优先搜索一遍。**

## 有向图的强连通

在有向图中，如果两个节点之间可以互相到达，那么这两个节点之间是强连通的。如果一个图中所有点都强连通，那么这个图就是一个强连通图。

一个直观思路就是分别测试每一个点到另外所有点的可达性，用深度优先搜索实现，其总的时间复杂度是 O(n^2)。

我们仍然希望找到一个更好的解决办法。

如果一个图只有一个强连通分量，那个这个强连通分量就是这个图本身，并且这个图就是强连通图。所以判断一个图的强连通性问题可以转化为求强连通分量的个数的问题。可以从该图的强连通分量的个数判断出这个图的强连通性。

**问题：有没有更好的算法？**

对于这个问题，Tarjan 提出了一个和判断无向图双连通性思路相通的有向图上的 **Tarjan 算法**，同样基于一遍的深度优先搜索。在搜索的同时维护一个储存节点标号的栈，其中存放已经访问过一次当时还没没有被分配于一个强连通分量的节点。和有向图上的 Tarjan 算法一致，在搜索的过程中计算每一个节点的low数值（表示该节点可达的最早的祖先节点）和dfn数值（在深度优先搜 索中访问到的顺序），利用这些数值可以用来决定何时将一些节点出栈并成为一 个新的强连通分量。

由于只需要判断是否是强连通图，如果找到多于一个强连通分量的时候就可以结束算法。Tarjan算法仍然基于一次深度优先搜索遍历，**时间复杂性O(N)**。

## 最小生成树

在一个无向的连通图的一个子图，这个子图是一棵树，使所有节点都在包含这个子图中，它们的边权和最小。

对树的构造，可以基于边的合理选择或者点的合理选择。

* **Prim算法**是对点的选择，每次选择一个到已加入集合的点举例最小的点，直到所有点都已经加入集合中，选择的边的权值和也是最小的。这个枚举过程中选择的是节点，时间复杂度主要和节点个数相关。如果使用邻接矩阵存储，时间复杂度会为O(N^2)。如果使用邻接表，就可以避免在边稀疏条件下遍历该节点所有边的繁琐步骤，时间复杂度可以为O(mlogn)。
* **Kruskal算法**是对边的选择，一开始将边按照权值大小排序，然后从最小的边开始检查其两端节点是否加入到集合中，注意的是如果两端节点都已经加入到集合，再加入本边会生成环。直到所有点都加入到点的集合中，可以得知已经生成了一棵最小生成树。边的排序的最优时间复杂度为O(mlogm)。后续步骤基本建立在选边的基础上，时间复杂度只为O(m)。所以算法总体的时间复杂度也是O(mlogm)。

这两种算法，毫无疑问，都属于贪心算法，在例如计算机网络等领域，一些问题都可以抽象为最小生成树问题。二者的时间复杂度，前者依赖于节点个数，后者主要和边数相关。稠密图中，使用Prim算法更合适，稀疏图中，Kruskal更优。在规模比较小的图中，二者的差别不大。

**问题：对于稀疏图的优化**

**对于稀疏图**，直接使用边集合存储，或者使用邻接表取代邻接矩阵，都可以提升算法的性能。

## 最短路径的算法优化

最短路径问题，可以分为单源最短、多源最短路径。解决本问题的经典算法有**Dijkstra，Floyd**以及**Bellman-Ford**算法。以上三种算法的基本原理非常简单此处避开不谈。

**问题：针对负环路**：图中如果出现负环路，那么对于节点到节点距离的更新是不会停止的，在合适的次数的更新之后，我们停止循环，并把当前的距离数组视为结果。如果新一轮的距离更新中，数组的值发生了改变，就可以说明负环路的存在。

**问题：针对单源最短路算法Bellman-Ford做优化**，我们有**SPFA**算法，Bellman-Ford的松弛过程，循环次数非常多，但是后期的一部分循环，可能是不需要的。因此我们使用一个队列，来省去不必要的计算。

初始状态下队列中只有源点，到除了本身之外所有点之间的最短路初始值为无限大。从队列中取出一个点，检查该点的所有出边，判断是否能够经由该点获得到达某点的更短路径，如果可以的话就更新最短路长度的数组，并将该目的顶点入队。重复上述操作，直到队列为空时算法结束。

很容易看出，如果一个顶点入队超过n次，那么它在路径中至少被经过了两次，从而更为简单的判断了图中负环路的存在。

# 查找和散列

## 红黑树

对于查找，我们知道平衡二叉树拥有非常好的性能，但是其维护过程过于复杂，在插入数据时维持平衡比较困难。

**问题：维持红黑树的黑平衡的意义？**

2-3树毫无疑问非常适合数据的维护，但是其在3-节点处，查找的性能低于二叉树。

我们希望有一种数据结构，能够结合二者的优点，于是我们使用红黑树，作为2-3树的二叉树实现方法。

我们假设，红黑树的颜色，是由父节点指向子节点的边的颜色，并且只能出现在父节点和左子节点之间。那么我们现在就有了3-节点的表示方式。

3节点

<A A<X<B >B

红黑树实现

>B

<A A<X<B

因此有在2-3树中的平衡，转化为红黑树中的黑平衡，而仅仅维护黑平衡，比维护整体平衡简单许多。现在整个对红黑树的操作，都可以模拟2-3树中的操作。

例如插入过程，对于2-3树，就是向上一直找到一个2节点，将其转化为3-节点，而对于红黑树，就是向上查找直到可以加入一条红边。

同理有删除等等。

## 散列查找

对于查找过程，即便是最平衡的二叉树也需要O(logn)。

如果在插入数据过程中，让数据和它的地址之中存在默认的规律。查找数据时，就可以直接推断其地址，获得O(1)的复杂度。

散列过程需要考虑两个问题，第一是如何散列，第二是如何解决冲突。

**问题：**

**如何散列：**

对于整数，我们很容易地使用求余的方法来计算其应该的地址。对于浮点数等，也可以采取类似的方法。假设数组的大小M，准备插入的数据对M求余即可。M应该是一个足够大，至少是满足数据规模需求的素数。选择素数的原因，是为了让数据更均匀的散列到数组当中。比如某些数据，可能固定几位，可能只出现1，0，如学生学号，倒数第二位基本只有0，1，2的情况，如果对100求余，其结果基本在0-30中。

**如何解决冲突：**

解决冲突的方法，同时取决与数据结构的设计。我们只讨论常用的两种:

**拉链法**，是在数组中直接存储一个指针，如果发生冲突，那么该指针后只需要新借一个存储新加数据的指针，这种做法允许数据个数超过数组的大小M。这种方法插入比较简单，只要数据分布较为均匀，数据大小与M的比值基本可以决定数组中一位的链表长度，查找只需要在链表上完成查找，删除时，将前一节点和后一节点连接，抛弃本节点即可。如果在插入过程中，使得散列值相同的数据在链表中按照顺序排列，就比较适合经常查找的情况，因为即便对于未出现的数据，也不需要查找整条链表。

**线性探测法**，是数组中仍然直接存储数据，但是冲突发生时，在数组中寻找相邻的空位插入数据。显然，数据的总数不能超过数组本身大小。如果装载因子本身已经很大，插入过程中探测的数组位置就很多。同时本方法的查找，最差需要遍历整个数组，因为倘若删除时，将同索引值的数据依次前移，并不能保证其间没有空位，所以删除不能到空就停止。如果删除后将后续数全部重新插入，虽然能保证到空即停止，但是可能需要对数组循环一次以上，才能将各数据送至最合适位置。两种删除代价都非常高，所以装载因子比较大的时候，线性探测可能比拉链法面对更多复杂的问题。当然，动态调整数组大小也是一种解决办法。

**散列在安全支付等的应用：**

单向散列算法：SHA,MD5等等。经过精心设计的散列函数，可以防止通过散列值逆向出原文的内容，适合用来保护密码，避免明文传输的安全风险。这些算法使得即便原文有些微的更改，散列值也会有巨大的变化，具有强混淆特性，可以应用于文件防止篡改和指出信息传输错误的作用。

除了强混淆特性的散列函数，有时候也需要混淆性弱、对微小变化健壮性非常好的散列方式。一个典型的应用就是：同样的一首乐曲，通过不同录音方式、不同码率、不同压制参数、不同编码方式都能得到截然不同的二进制文件，但是很多人都用过类似 Sound Hound 的app，其核心思想就是利用具有强健壮性的散列方式来处理音频信息，甚至对比较模糊的音频也能通过和数据库中存储的已知散列值比较，得到乐曲的名字和其他信息。因此散列方式意外地在语音识别领域得到了应用。

# 内排序

## 快速排序及优化

快速排序在平均情况下拥有非常好的性能，同时能实现归并排序所不能及的原地排序。

但是在一些情况下，其时间复杂性可能退化为O(n^2)。

本实验中主要针对随机数据、升序数据、降序数据考虑提升快速排序的性能。原始版本的快速排序的基础上：

* **改变轴心的选择方法**

一种比较简单的轴心选择方法，是直接选择需要排序内的某一位置元素，如最前、最后、中间。但是这种选择方法一旦选择到较小或者较大的轴心，分区将会变得不平衡，是效率退化。因此我们可以尝试从需要排序区域的前、中、后各选择一个数，对这三个数进行排序，然后选择值在中间的。算法陷入较差时间复杂性的概率就大大降低。

* 在数据规模较小的时候，直接使用**插入排序**

尽管快排在大部分情况下拥有非常好的性能，在实现小规模数据的排序时，看起来插入排序的性能更优。如果在递归过程中，能把递归的最后一层的速度提升，毫无疑问整个递归过程都会加速。因此在判断数组规模小于10时，我们放弃继续使用快速排序，直接使用插入排序。

## 线性排序算法

所有使用比较的排序，其性能的下界为O(nlogn)。我们有很多种算法能得到这个渐进性能。

但是如果不使用比较的方法排序，可以使算法的性能达到线性，以下讨论三种典型线性排序算法。

* **计数排序**

只需要记载每个数字出现过的个数在数组中，对数组的项求和就能计算出该数字在排序结果中的位置。

基本的计数排序需要在一开始获知输入数据的最大值从而合理的构造数组。当然我们也可以在最初建立一个足够大的数组，在获取输入过程中得到输入数据的最大值，从而有较好的灵活性。

* **桶排序**

桶排序可以看作计数排序的泛化，供给每个数字分配桶，并独立为各个桶中元素进行排序。同样我们希望数据的范围是已知的。

* **基数排序**

计数排序，由原理可见，适合的情况是数字范围较小的情况，其在数组中的位置恰好能表示它在所有待排序数中的大小。由此发展出了基数排序，已知最大位数的输入，从最低位开始对每一位使用计数排序，再次排序的过程中，当前位相同的情况下，维持低位排序的结果顺序的稳定，不需要排序的情况下就能完成排序。原则上说对某一位的排序可以使用任意稳定排序，但是可能会导致性能降低。

借用桶排序的思想，对当前位，我们将数据分别存入桶中，然后再按顺序拿出，就能维持稳定，同时很适合数字范围大的情况。

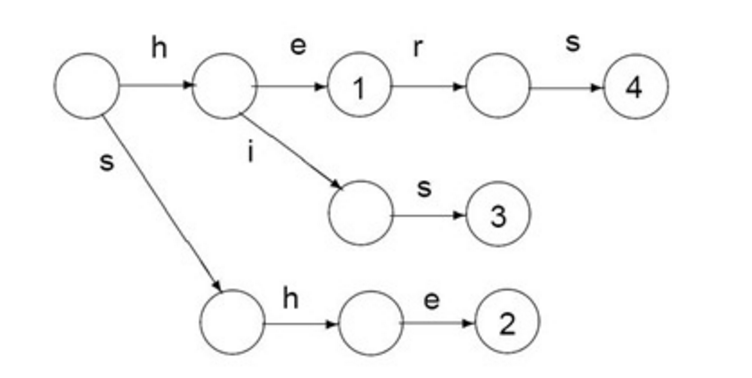
# 大作业 AC 自动机

## 原理及简单实现：

Aho-Corsick Algorithm，即AC自动机算法，是一种多模式匹配算法。

AC 算法的核心思想是构造词典的自动机(可以使用trie树来实现)，其算法复杂度是O(m+k+z), m是文本长度，k是所有pattern长度之和，z是字符串中出现pattern的个数。

对于Pattern P = {he; she; his; hers} ，建立如下图的字典树。



自动机的核心在于字母表、状态集以及状态转移函数。

AC自动机的思想在于，每个表示匹配成功的字符作为触发状态转移的输入，所以当当前处于某个状态的时候，说明已经匹配从初始状态到当前状态的所有字符。

根据以上思想可以建立状态转移函数，其好处在于，如果多个模式的前缀相同，那么只需要匹配一次。

失配函数，即如上图she匹配完成，此时再获得一个r，很明显，此时匹配到的是her三个字符，也就是说，she的后缀包含了hers 的前缀，失配函数可以简化这个匹配过程。

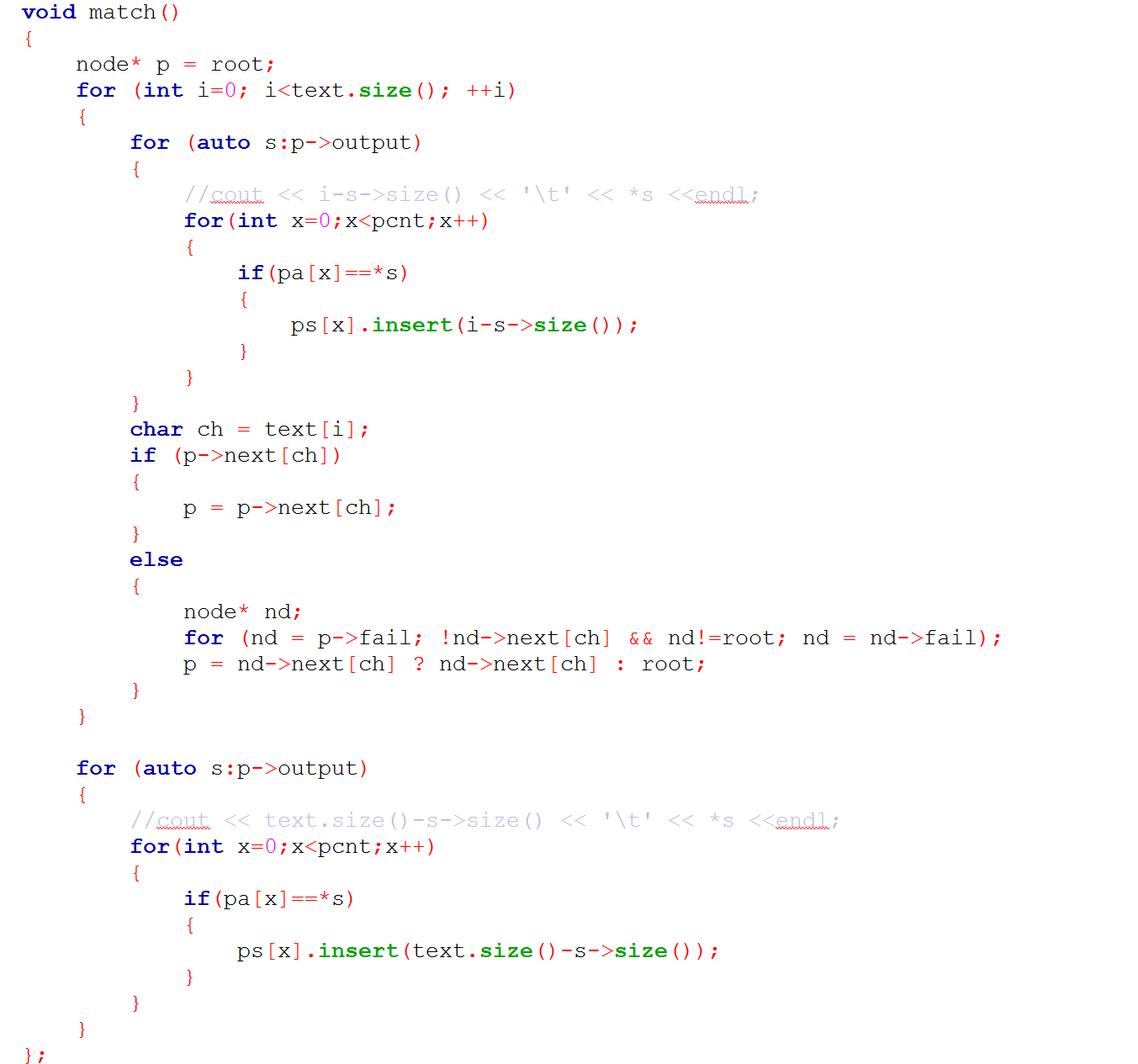
最后也是最容易确定的就是匹配成功的输出，一旦成功，判断当前状态是否对应着一个输出。

抽象的表达出来，有：

 1. goto函数 g（q,a）, 它表示，如果当前状态是q，输入输入字符是a，那么下一个状态是g(q,a)   
               a)如果边edge(q, r) 标记为a， 那么 g（q,a) = r   
               b) 如果a不输入0状态的出边 g（0,a)  = 0 , g(q,a) =  $(空)   
           2. failure函数 f（q）， 对q不为0的状态，failure函数表示在状态q失配   
               f（q）是状态机中的一个节点，标记为pattern中某个pattern恰当的最长后缀（如下图5的he）   
           3. out函数 out（q）， 状态q，识别出某个pattern

我们将整个过程主要拆分为建树、建立失配函数两个部分，输出和状态转移可以同时完成。

接下来读入试图匹配的内容，对当前状态，如果对于输入有对应的状态转换，则发生转换并检查是否有输出，如果没有，则按照失配函数处理。



## 优化及应用：

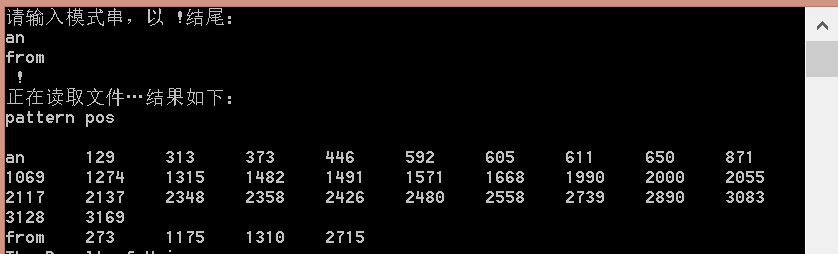
除了最简单的匹配之外。

为了能够实现多个模式串同时出现、或者多个只出现一个的查询，我们使用一个字符串一个集合的对应，完成匹配则将成功位置加入集合的方法。

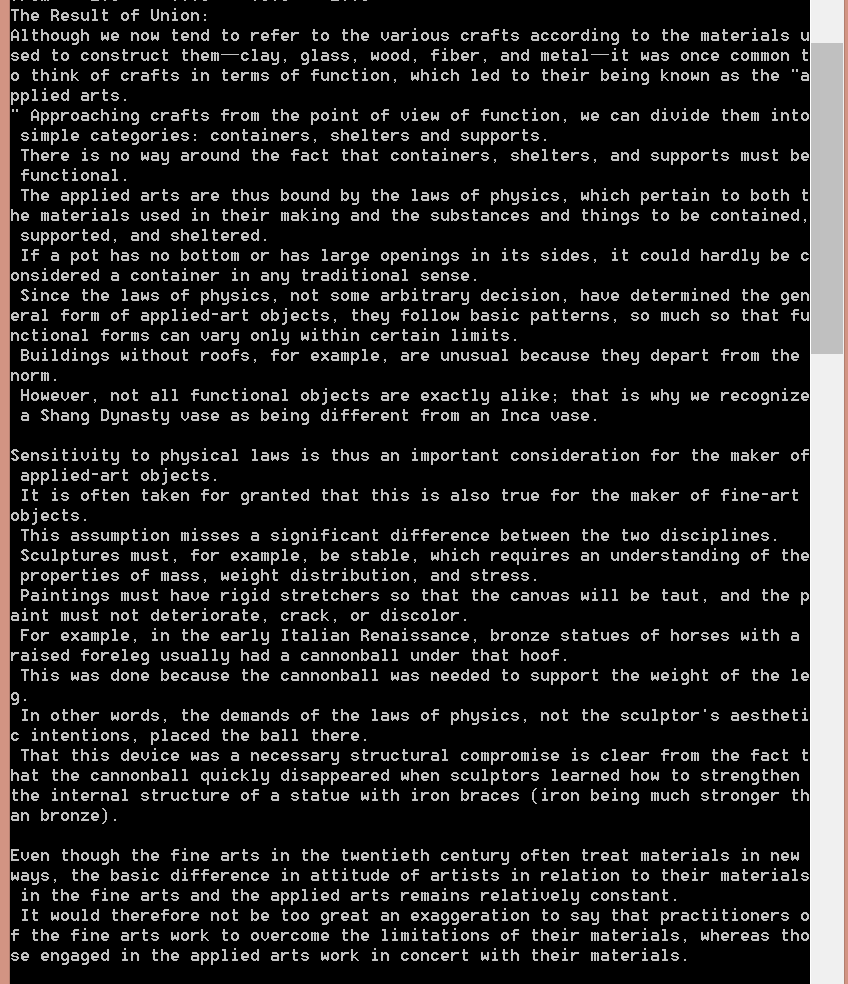
如果所有集合都不为空，则证明多个模式串都出现在了这个匹配文本中。

如果至少一个集合不空，则证明至少一个模式串出现在文本中。

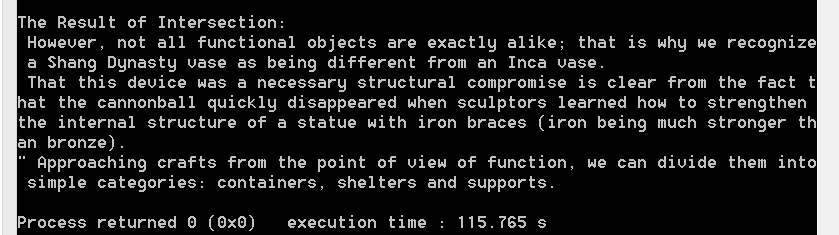
## 测试及实现结果：



针对一个文本（来自文件），直接输出位置（本代码可以实现多模式匹配，此处我们用两个模式串做测试）



输出两个单词，至少有一个出现的文本。



句子中同时出现an和from