**后缀树【Suffix Tree】 收藏**

在pongba的讨论组上看到一道Amazon的面试题：找出给定字符串里的最长回文。例子：输入XMADAMYX。则输出MADAM。这道题的流行解法是用后缀树（Suffix Tree)。这坨数据结构最酷的地方是用它能高效解决一大票复杂的字符串编程问题：

•在文本T里查询T是否包含子串P（复杂度同流行的KMP相当）。

•文本T里找出最长重复子串。比如abcdabcefda里abc同da都重复出现，而最长重复子串是abc。

•找出字符串S1同S2的最长公共子串。注意不是常用作动态规划例子的LCS哈。比如字符串acdfg同akdfc的最长公共子串为df，而他们的LCS是adf。

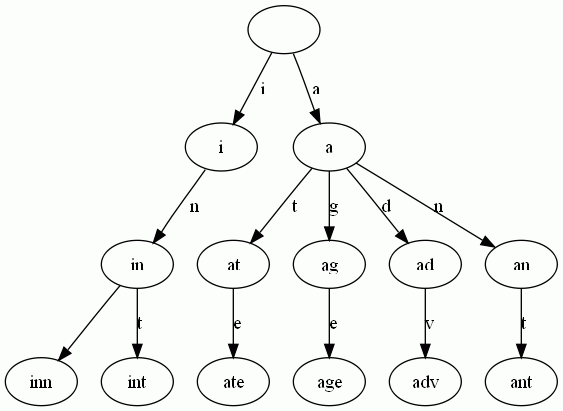
•Ziv-Lampel无损压缩算法。

•还有就是这道面试题问的最长回文了。

另外后缀树在生物信息学里应该应用广泛。碱基匹配和选取的计算本质上就是操作超长的{C, T, A, G, U}\*字符串嘛。

虽说后缀树的概念独立于Trie的概念，但我觉得从Trie推出后缀树自然简洁，所以先简单解释一下Trie。“Trie”这个单词来自于"retrieve"，可见它的用途主要是字符串查询。不过词汇变迁多半比较诡异，Trie不发tree的音，而发try的音。

Trie是坨简单但实用的数据结构，通常用于实现字典查询。我们做即时响应用户输入的AJAX搜索框时，就是Trie开始。谁说学点数据结构没用来 着？本质上，Trie是一颗存储多个字符串的树。相邻节点间的边代表一个字符，这样树的每条分支代表一则子串，而树的叶节点则代表完整的字符串。和普通树 不同的地方是，相同的字符串前缀共享同一条分支。还是例子最清楚。给出一组单词，inn, int, at, age, adv, ant, 我们可以得到下面的Trie：



可以看出：

•每条边对应一个字母。

•每个节点对应一项前缀。叶节点对应最长前缀，即单词本身。

•单词inn与单词int有共同的前缀“in”, 因此他们共享左边的一条分支，root->i->in。同理，ate, age, adv, 和ant共享前缀"a"，所以他们共享从根节点到节点"a"的边。

•查询非常简单。比如要查找int，顺着路径i -> in -> int就找到了。

•搭建Trie的基本算法也很简单，无非是逐一把每则单词的每个字母插入Trie。插入前先看前缀是否存在。如果存在，就共享，否则创建对应的节点和边。比如要插入单词add，就有下面几步：

1.考察前缀"a"，发现边a已经存在。于是顺着边a走到节点a。

2.考察剩下的字符串"dd"的前缀"d"，发现从节点a出发，已经有边d存在。于是顺着边d走到节点ad

3.考察最后一个字符"d"，这下从节点ad出发没有边d了，于是创建节点ad的子节点add，并把边ad->add标记为d。

继续插播广告。Graph作图软件Graphviz还不错，用的DSL相当简单。上面的图就是用它做的。三步就够了：

1.实现Trie数据结构。这步不用花哨。10行代码，一坨hash足矣。

2.把上面的结构翻译成Graphviz的DSL。简单的深度优先足矣。

3.调用Graphviz的命令。图就生成乐。

多花20分钟，避免了手工作图排版的自虐行为。而且可以自由试验各式例子而不用担心反复画图的琐碎，何乐而不为嗫？

有了Trie，后缀树就容易理解了。先说说后缀的定义。给定一长度为n的字符串S=S1S2..Si..Sn，和整数i，1 <= i <= n，子串SiSi+1...Sn都是字符串S的后缀。以字符串S=XMADAMYX为例，它的长度为8，所以S[1..8], S[2..8], ... , S[8..8]都算S的后缀，我们一般还把空字串也算成后缀。这样，我们一共有如下后缀。对于后缀S[i..n]，我们说这项后缀起始于i。

1.S[1..8], XMADAMYX， 也就是字符串本身，起始位置为1

2.S[2..8], MADAMYX，起始位置为2

3.S[3..8], ADAMYX，起始位置为3

4.S[4..8], DAMYX，起始位置为4

5.S[5..8], AMYX，起始位置为5

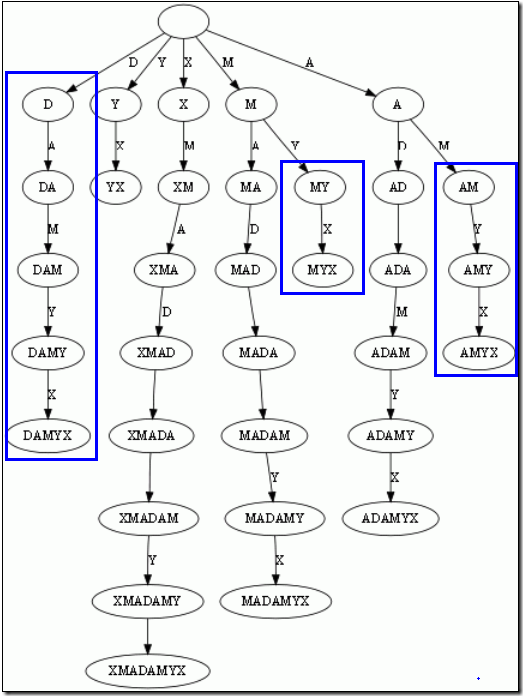
6.S[6..8], MYX，起始位置为6

7.S[7..8], YX，起始位置为7

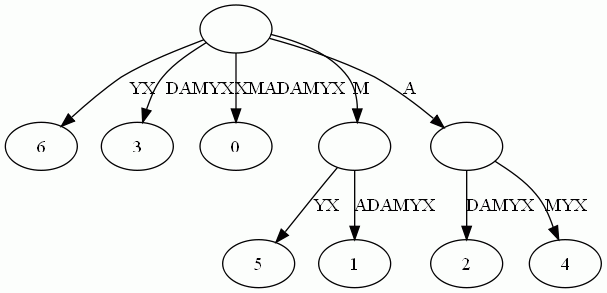
8.S[8..8], X，起始位置为8

9.空字串。记为$。

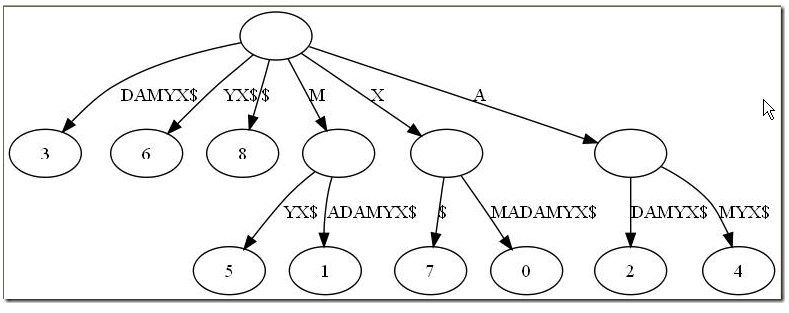
而后缀树，就是包含一则字符串所有后缀的压缩Trie。把上面的后缀加入Trie后，我们得到下面的结构：



仔细观察上图，我们可以看到不少值得压缩的地方。比如蓝框标注的分支都是独苗，没有必要用单独的节点同边表示。如果我们允许任意一条边里包含多个字 母，就可以把这种没有分叉的路径压缩到一条边。另外每条边已经包含了足够的后缀信息，我们就不用再给节点标注字符串信息了。我们只需要在叶节点上标注上每 项后缀的起始位置。于是我们得到下图：



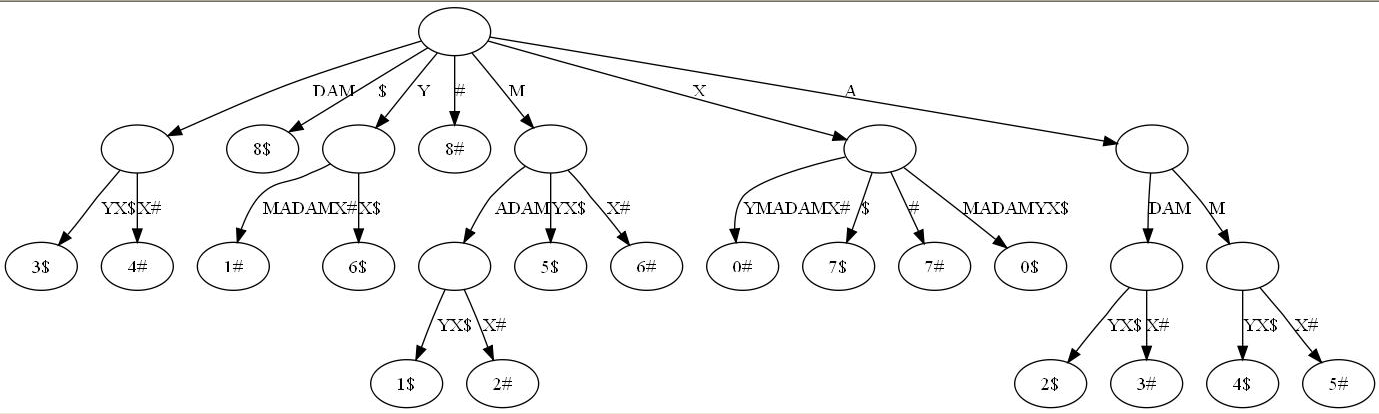
这样的结构丢失了某些后缀。比如后缀X在上图中消失了，因为它正好是字符串XMADAMYX的前缀。为了避免这种情况，我们也规定每项后缀不能是其 它后缀的前缀。要解决这个问题其实挺简单，在待处理的子串后加一坨空字串就行了。例如我们处理XMADAMYX前，先把XMADAMYX变为 XMADAMYX$，于是就得到suffix tree乐。



那后缀树同最长回文有什么关系呢？我们得先知道两坨坨简单概念：

•最低共有祖先，LCA（Lowest Common Ancestor)，也就是任意两节点（多个也行）最长的共有前缀。比如下图中，节点7同节点10的共同祖先是节点1与借点，但最低共同祖先是5。 查找LCA的算法是O(1)的复杂度，这年头少见。代价是需要对后缀树做复杂度为O(n)的预处理。

•广 义后缀树(Generalized Suffix Tree)。传统的后缀树处理一坨单词的所有后缀。广义后缀树存储任意多个单词的所有后缀。例如下图是单词XMADAMYX与XYMADAMX的广义后缀 树。注意我们需要区分不同单词的后缀，所以叶节点用不同的特殊符号与后缀位置配对。



有了上面的概念，查找最长回文相对简单了。思维的突破点在于考察回文的半径，而不是回文本身。所谓半径，就是回文对折后的字串。比如回文MADAM 的半径为MAD，半径长度为3，半径的中心是字母D。显然，最长回文必有最长半径，且两条半径相等。还是以MADAM为例，以D为中心往左，我们得到半径 DAM；以D为中心向右，我们得到半径DAM。二者肯定相等。因为MADAM已经是单词XMADAMYX里的最长回文，我们可以肯定从D往左数的字串 DAMX与从D往右数的子串DAMYX共享最长前缀DAM。而这，正是解决回文问题的关键。现在我们有后缀树，怎么把从D向左数的字串DAMX变成后缀 呢？到这个地步，答案应该明显：把单词XMADAMYX翻转就行了。于是我们把寻找回文的问题转换成了寻找两坨后缀的LCA的问题。当然，我们还需要知道 到底查询那些后缀间的LCA。这也简单，给定字符串S，如果最长回文的中心在i，那从位置i向右数的后缀刚好是S(i)，而向左数的字符串刚好是翻转S后 得到的字符串S‘的后缀S'(n-i+1)。这里的n是字符串S的长度。有了这套直观解释，算法自然呼之欲出：

1.预处理后缀树，使得查询LCA的复杂度为O(1)。这步的开销是O(N)，N是单词S的长度

2.对单词的每一位置i(也就是从0到N-1)，获取LCA(S(i), S(N-i+1)) 以及LCA(S(i+1), S(n-i+1))。查找两次的原因是我们需要考虑奇数回文和偶数回文的情况。这步要考察每坨i，所以复杂度是O(N)

3.找到最大的LCA，我们也就得到了回文的中心i以及回文的半径长度，自然也就得到了最长回文。总的复杂度O(n)。

用上图做例子，i为3时，LCA(3$, 4#)为DAM，正好是最长半径。当然，这只是直观的叙述。

这篇帖子只大致描述了后缀树的基本思路。要想写出实用代码，至少还得知道下面的知识：

•创建后缀树的O(n)算法。至于是Peter Weiner的73年年度最佳算法，还是Edward McCreight1976的改进算法，还是1995年E. Ukkonen大幅简化的算法，还是Juha Kärkkäinen 和 Peter Sanders2003年进一步简化的线性算法，各位老大随喜。

•实现后缀树用的数据结构。比如常用的子结点加兄弟节点列表，Directed

•优化后缀树空间的办法。比如不存储子串，而存储读取子串必需的位置。以及Directed Acyclic Word Graph，常缩写为黑哥哥们挂在嘴边的DAWG。