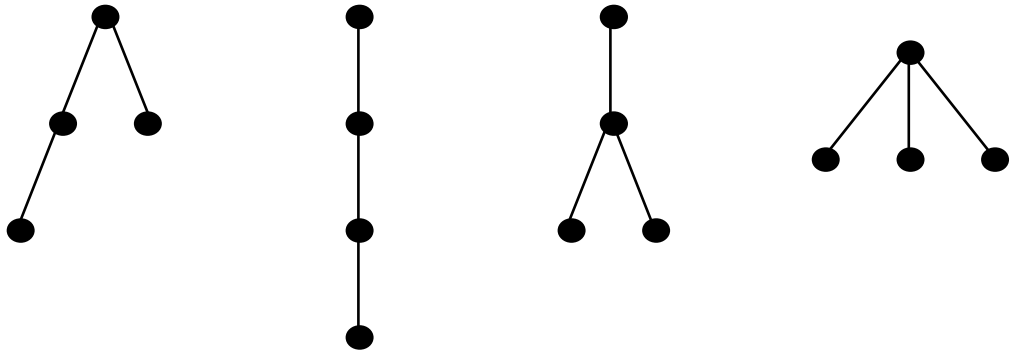
树的枚举

江苏省常州高级中学　李源

引子

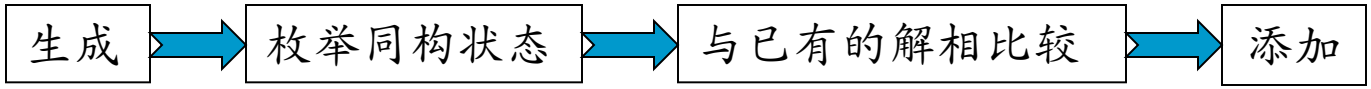
- 树，在计算机算法中是非常重要的非线形结构。即使撇开树的其他广泛应用不说，单单对树本身的形态进行思考与研究，也是一个十分有趣，且具有挑战性的过程。



4 个结点的树（有向树）

枚举算法

- 常规的搜索加判重的做法：



- 下面我们就来看一种**不重复地**生成**所有**确定结点数和深度的有向树的构造性算法。
  - 不重复性：树的大小定义
  - 不遗漏性：树的变换算法

树的大小定义

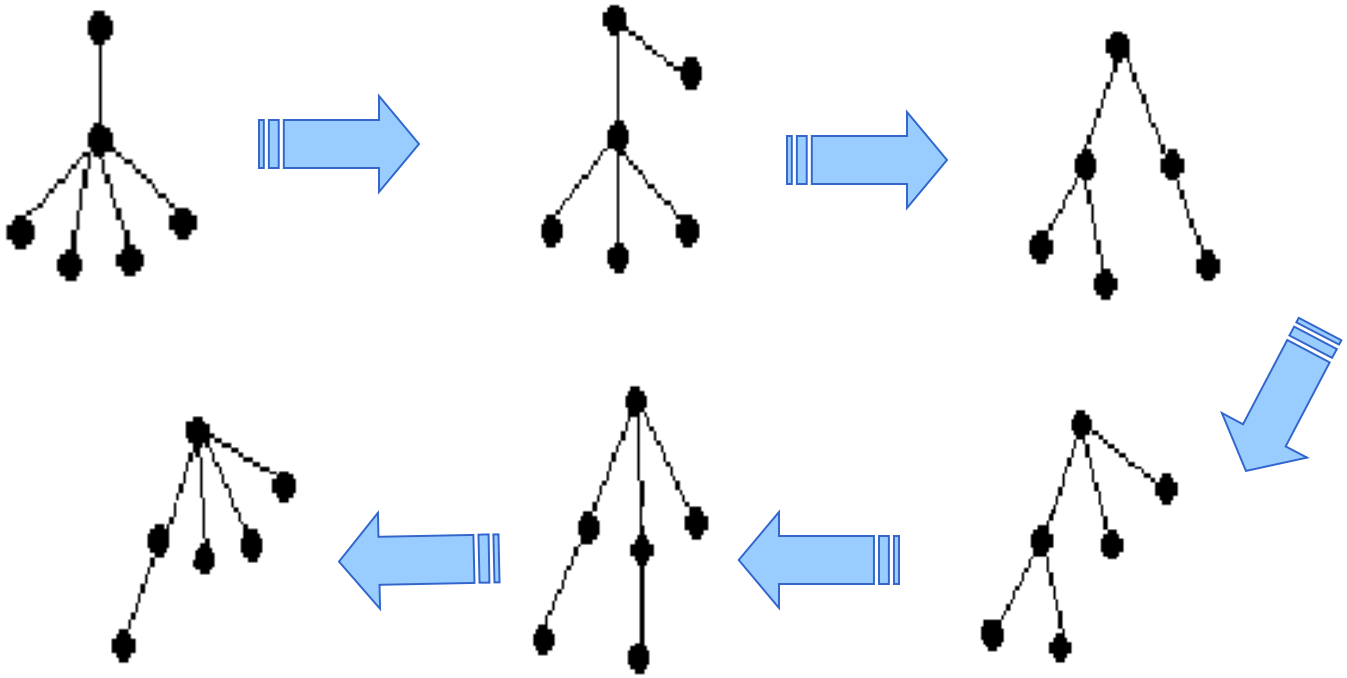
- 我们对树的“大小”作一个规定，使不同树结构之间的关系有序化。

- 假设当前要比较树 A 与树 B 的大小（树 A 与树 B 的子树也都要按照下述的大小关系递归地从大到小排序）。

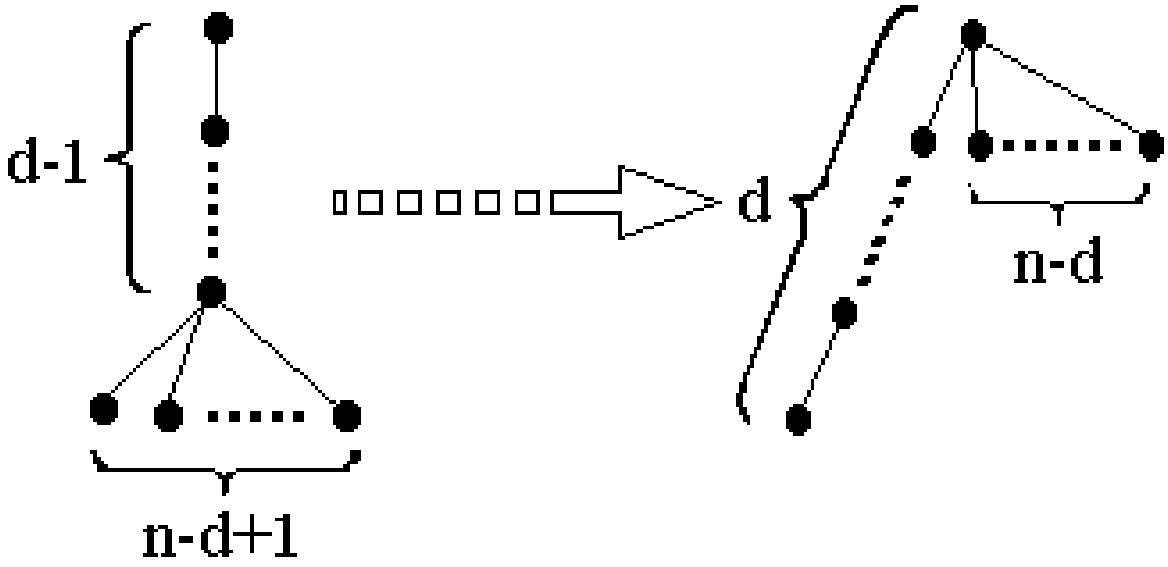
  比较过程为：

  若 A 的深度大于 B，则 A>B；若 A 的深度小于 B，则 A<B；
  若 A 与 B 深度相等，视 A 与 B 的结点数：
  若 A 的结点数大于 B，则 A>B；若 A 的结点数小于 B，则 A<B；
  若 A 与 B 的结点数相等，依次讨论 A 与 B 的子树：
  拥有较大子树的树较大。若当前讨论的子树相等，则讨论 A 与 B 的下一棵子树。

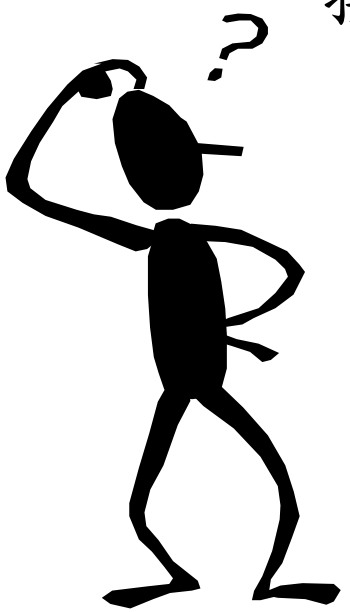
- 现在回到枚举有向树的问题上来：对于深度为 d，结点数为 n 的所有形态的有向树，根据上面的比较规则，可以把它们从大到小排成一个序列。举 d=3， n=6 为例，这个序列是：



- 进一步地，对于任意的 n 和 d ，在相应序列中的第一棵树和最后一棵树的形态是容易确定的，如下：

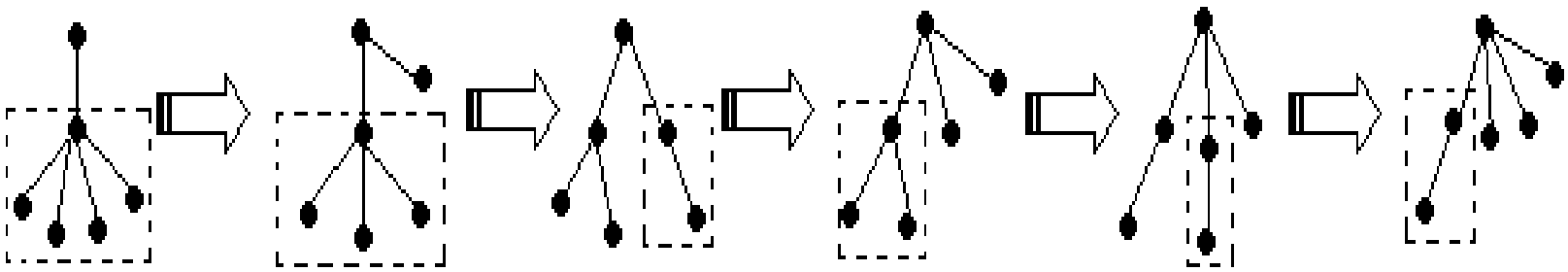


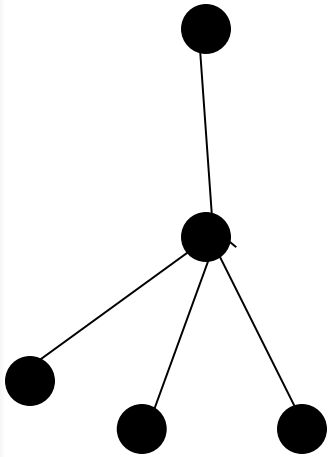
- 现在问题就转化为，根据上述的大小定义，能否找到一种变换的规则，使根据这种规则，可以从最小的一棵树开始，连续地、不遗漏不重复地生成接下来的所有不同形态的树？



变换过程

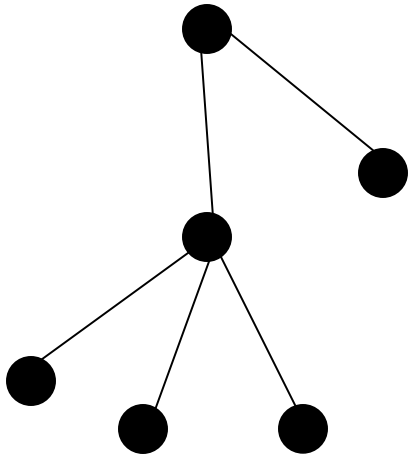
- 首先，从树的序列中的一个形态变换到另一个形态，是一个变小的过程。
- 在这个变换过程中，至少有一棵子树被变小了，变小的过程是通过夺走它的一个子结点实现的（我们用一个虚线框来表示被变小的子树）。





- 结点被夺去；

- 排在它后面的某些子树将会得到这个被夺走的结点，或被重新组合。



- 它们会适当地变大，然而由于它们在有向树大小比较的过程中的地位比先前被变小的那棵子树要低，于是，总的说来，整个有向树就被变小了。

- 过程 I　寻找被删去结点
- 过程 II　将被删的结点重组到后面的子树中

过程Ⅰ 寻找被删去结点

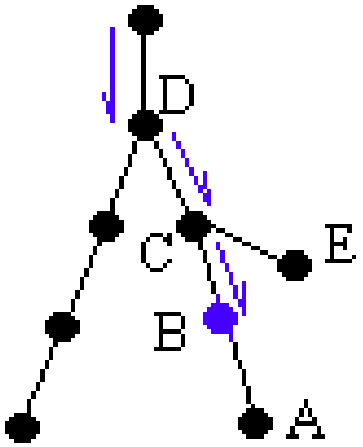
- 在选择被删去的结点时，其所在子树的变小对于整棵树的影响必须尽量小。使它经过变换后恰好可以成为**序列中紧邻它的一棵树**而不是跳跃性的变换。所以：

- 首先，这个被夺取的结点必然为叶结点。

- 第二，对于并列的若干子树，应当从后向前查找。

过程 I 算法

- 从根出发，在子结点中从后向前找到第一个非叶结点的结点，继续搜索直到到达一个子结点均为叶子的结点为止。

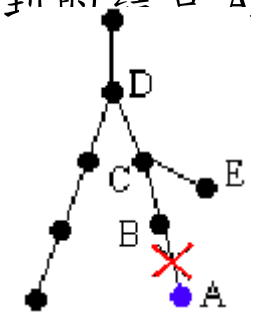


是否可认为该结点的最后一个子结点为待删除结点呢？事实上仍需进一步处理：

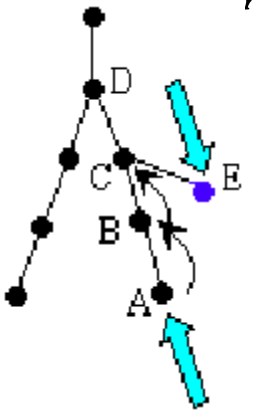
假如，找到的待删除结点 A 为其父亲 B 唯一的子结点，也就意味着如果将 A 从它的父结点 B 删除，那么以 B 为根的那棵子树的深度将会减少 1……

过程 I 算法

- 在对树的大小定义中，深度比结点数更优先。所以，在选择待删除结点时，应该**尽量选择删除后不影响子树深度的结点**优先处理。
- 因此，在找到 A 结点后，必须沿其父亲结点进行回溯，直到当前结点以下的子树的深度不因删除 A 而改变为止（在上图中直到 D 结点），在回溯的过程中，如果经过某一结点，它还有另一个子结点（比如上图中的 C，它还有另一个子结点 E），那么就舍弃原来找到的结点 A，把 E 定为待删除的结点。



如果将 A 从 B 断开，则以 B、C 为根的子树的深度都会受到影响。



C 有子结点 E，E→Target

回溯

找到 A，A→Target

过程 II　将被删的结点重组到后面的子树中

- 在找到该结点并删除之后，又需要进行一系列的处理以形成一棵新的树。在建立新树的时候要强调的一点就是，“要使整棵树变小，但变小的幅度必须达到最小”。

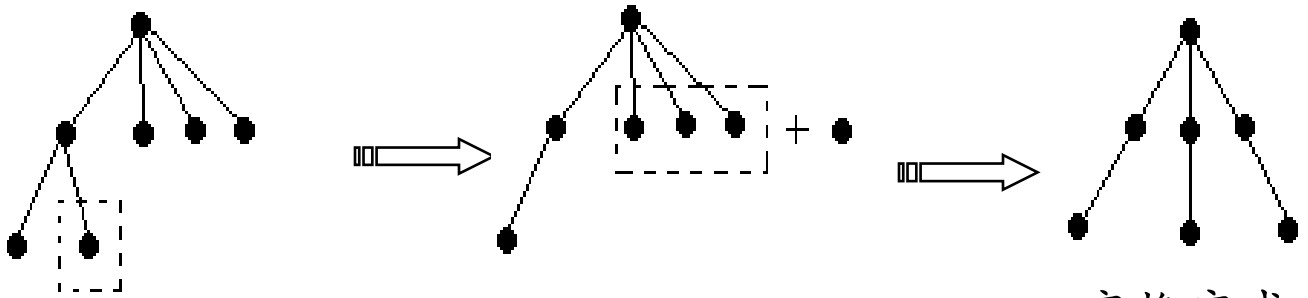
删除一个结点之后，会出现两种情况：

- **当前子树深度不变，结点数变小；**
- **或者子树深度变小。**

下面就对这两种情况分别进行讨论。

过程 II 算法　删除结点后子树深度不变

- 接下来，对排列在被改动的这棵子树以后的其它子树进行重新组合，使它们在满足不大于当前这棵子树的情况下变得最大（同时将被删除的那个结点加入）。



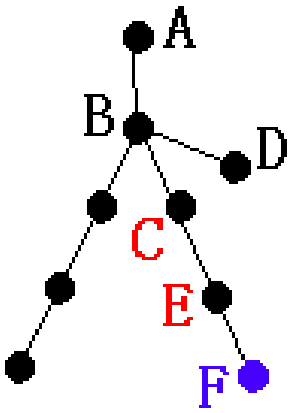
删除结点，使子树变小

将后面的子树重组（最大化）

变换完成

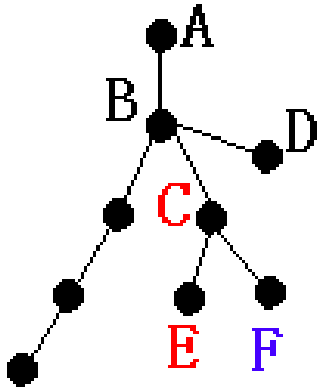
过程 II 算法　　删除结点后子树深度变小

- 对于删除结点后子树深度改变的，则要进行如下的处理（举例来说）：

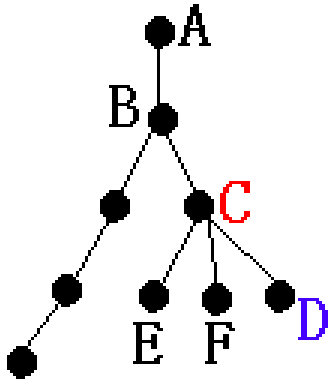


F：要从父亲处删除的结点。

CE：深度将会变小。要进一步处理。



将 F 删去，处理 E 极其兄弟（此处为空），连同删去的 F，以 C 为根进行重组。



处理 C 极其兄弟（D），连同 C 以下的所有子树，以 C 的父亲 B 为根进行重组。得到新的树。

- 完整的有根树枚举算法大致可以如下构成：

1 读入 d,n

2 找第一棵树（最大的）

3 while 未全部生成 do

{这步判断可以用计数算法得到的总数来判断，也可以先求得最小的一棵树用来判断}

4 找到待删除的结点 target;

5 删除 target;

6 对树进行变换；{包括上面的两种情况： 子树深度改变， 以及深度未改变}

7 end of while

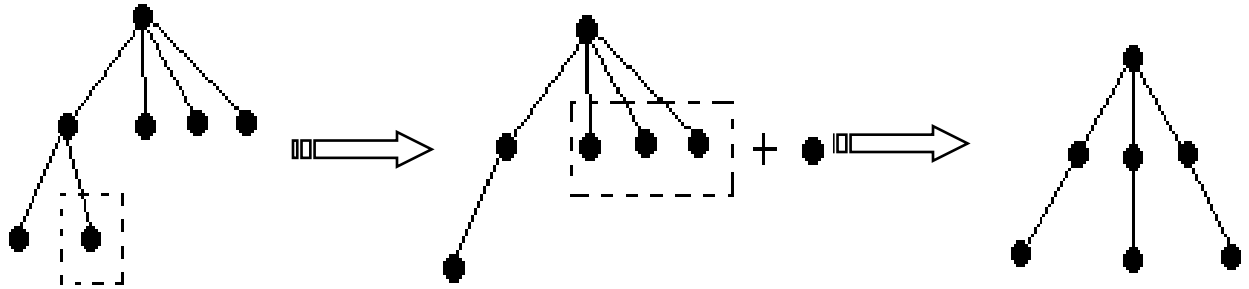
小结

- 虽然上面变换过程看似十分复杂，实质上它是以一种简洁和严谨的规律为基础的。

- 在仔细的研究中，大家可以体会到变换过程的和谐：它和自然数的递减与退位还颇有相似之处。

  比如说：为了使 2000 成为比它小，但又与它相差最少的自然数：

- 类似地，再看一个有向树变换的例子：



删除结点，使子树变小　　将后面的子树重组（最大化）　　变换完成

- 我们先从后向前找到一个待删除结点，删除它之后，然后对其它子树进行了一定的变换（类似于上面把0变成为9的过程），确保了整棵树变小的程度最小，从而得到了序列中的下一棵有向树。

- 以上介绍的算法可以实现给定深度d，结点数n，按照从大到小的顺序变换生成所有形态的有向树。算法中涉及的树的复制，以及遍历等，复杂度均为O（n），并且在树的生成过程中完全没有重复生成，所以整个算法的耗时主要与不同形态树的总数有关。

- 该算法最直接的应用是“无根树”问题。下表可以作为算法时间复杂度的直观参考。

我们可以用按照枚举算法编写的生成程序与搜索算法作一个比较（测试环境： PIII 500MHz， 192Mb RAM， Borland Pascal 7.0，用时单位： s）：

N	11	12	13	14	15	16	17	18	19	20
构造用时	0.05	0.05	0.11	0.16	0.49	1.21	3.19	8.52	23.08	62.91
搜索用时	0.27	0.71	2.09	6.04	17.69	51.92	152.20	-	-	-