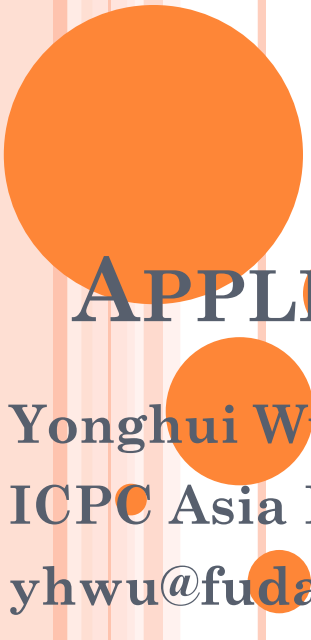


Application of Binary Trees

Yonghui Wu

ICPC Asia Programming Contest 1st Training Committee – Chair

yhwu@fudan.edu.cn

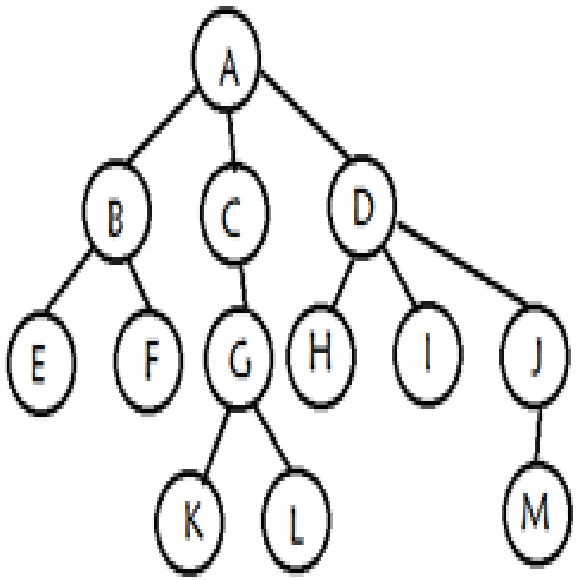
- Binary Tree
  - A binary tree is a tree in which no node can have more than two children (referred to as the left child and the right child).

- Converting Ordered Trees to Binary Trees
- Paths of Binary Trees
- Traversal of Binary Trees

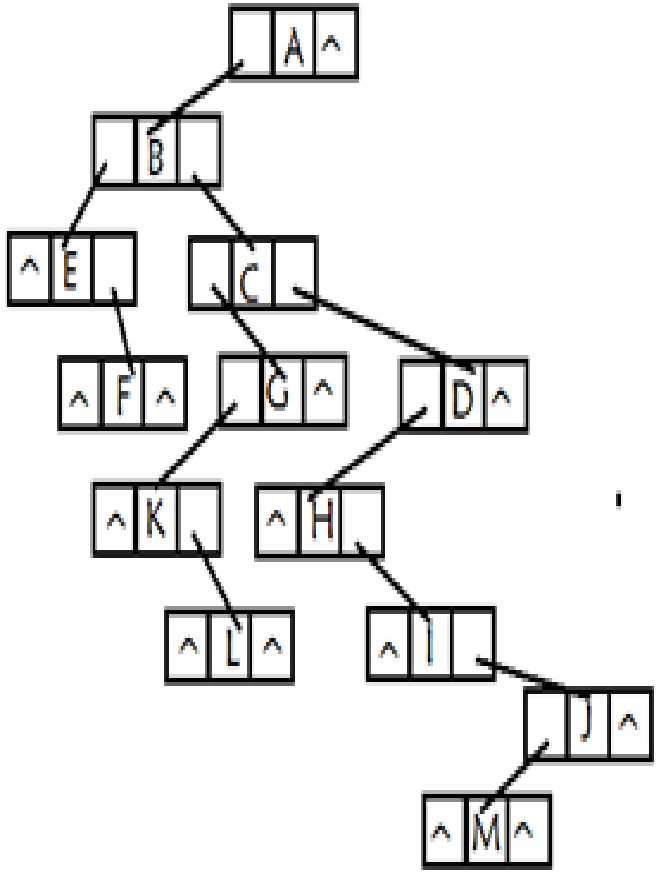
Converting Ordered Trees to Binary Trees

- In the real world some problems can be modeled as ordered trees. In order to save memory and process conveniently, ordered trees should be converted to corresponding binary trees.
- "Corresponding binary trees" refers to preorder traversal and postorder traversal for an ordered tree are same as preorder traversal and postorder traversal for a converted binary tree.

- The process of converting an ordered tree T to a binary tree T 1 is as follows:
  - The root of T is as the root of T1;
  - The first child of T is as the root of the left subtree of T1 x;
  - If x has children, its first child is as the root of its left subtree; and if x has siblings, its first sibling is as the root of its right subtree.

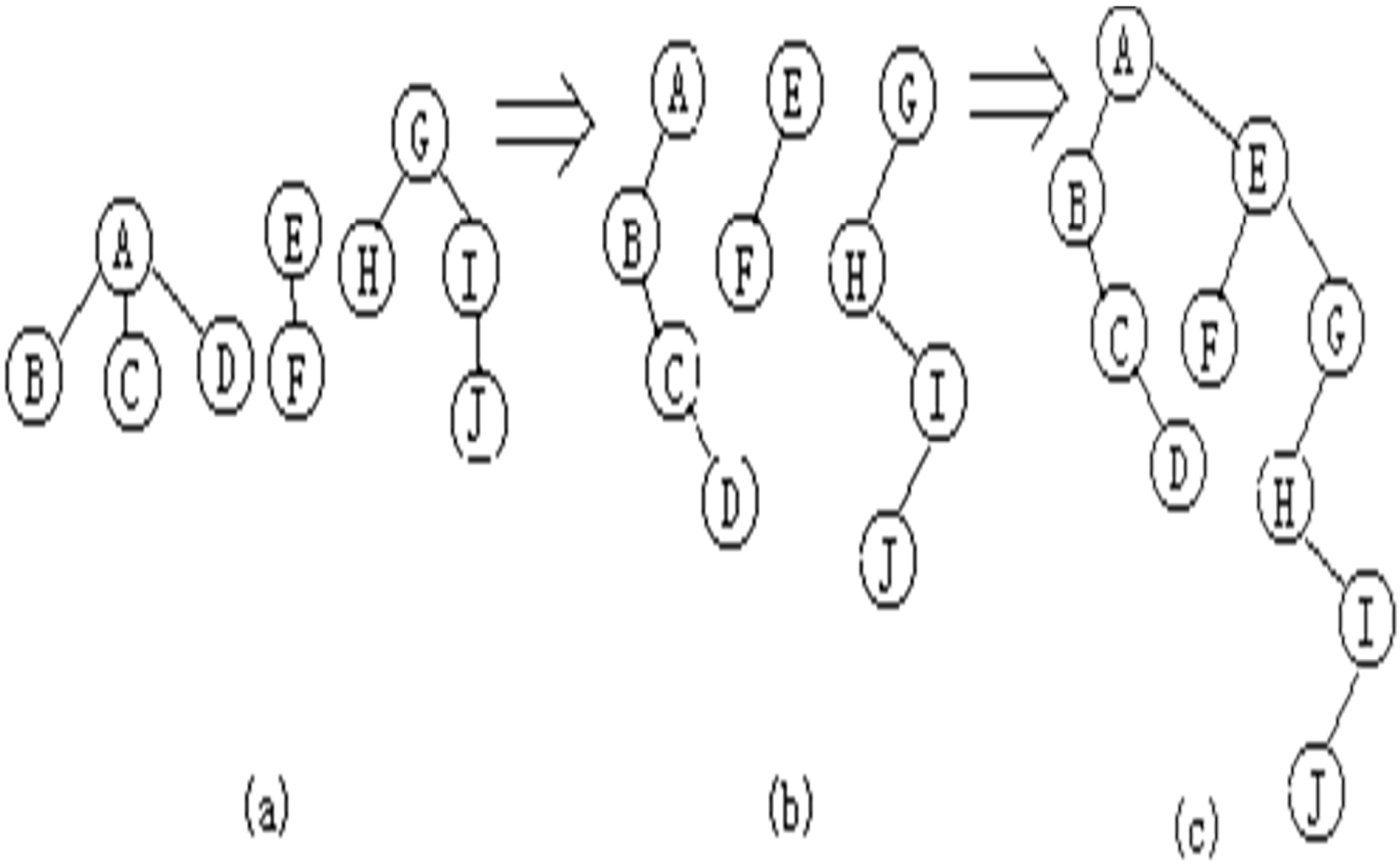


(a) tree



(b) The binary tree converted by(a)

- A forest can also be transferred into a binary tree.
  - Firstly all trees are transferred into binary trees.
  - Then all binary trees are transferred into a corresponding binary tree: from the first binary tree, the next tree's root is as the right child of the previous tree's root.


A
B
C
D
E
F
G
H
I
J
(a)
(b)
(c)

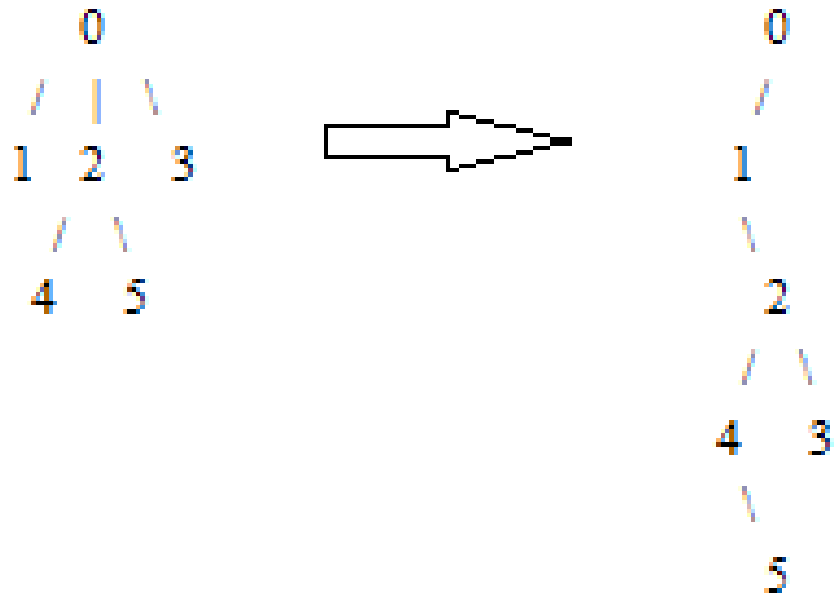
Tree Grafting

- **Source: ACM Rocky Mountain 2007**
- **IDs for Online Judge: POJ 3437, UVA 3821**

- Trees have many applications in computer science. Perhaps the most commonly used trees are rooted binary trees, but there are other types of rooted trees that may be useful as well. One example is ordered trees, in which the subtrees for any given node are ordered. The number of children of each node is variable, and there is no limit on the number. Formally, an ordered tree consists of a finite set of nodes T such that
  - there is one node designated as the root, denoted root(T);
  - the remaining nodes are partitioned into subsets T_1, T_2, ..., T_m, each of which is also a tree (subtrees).

- Also, define root(T_1), ..., root(T_m) to be the children of root(T)的, with root(T_i) being the i-th child. The nodes root(T_1), ..., root(T_m) are siblings.
- It is often more convenient to represent an ordered tree as a rooted binary tree, so that each node can be stored in the same amount of memory. The conversion is performed by the following steps:
  - 1. remove all edges from each node to its children;
  - 2. for each node, add an edge to its first child in T (if any) as the left child;
  - 3. for each node, add an edge to its next sibling in T (if any) as the right child.

- This is illustrated by the following:

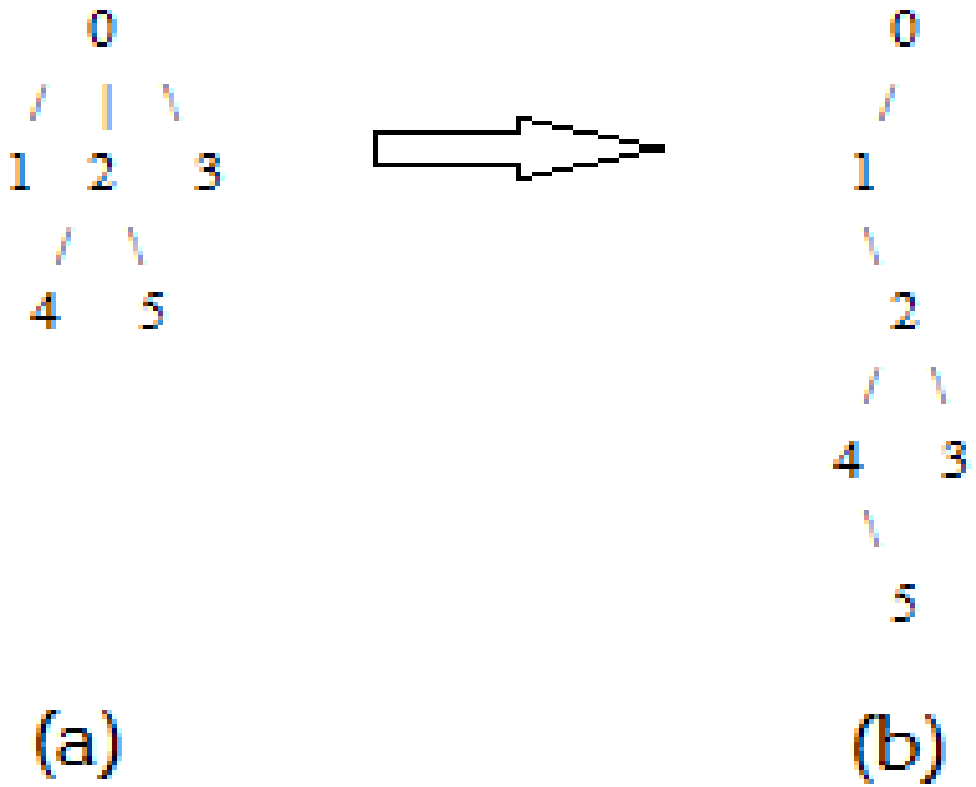


- In most cases, the height of the tree (the number of edges in the longest root-to-leaf path) increases after the conversion. This is undesirable because the complexity of many algorithms on trees depends on its height.
- You are asked to write a program that computes the height of the tree before and after the conversion.

- **Input**
- The input is given by a number of lines giving the directions taken in a depth-first traversal of the trees. There is one line for each tree. For example, the tree above would give dudduduudu, meaning 0 down to 1, 1 up to 0, 0 down to 2, etc. The input is terminated by a line whose first character is #. You may assume that each tree has at least 2 and no more than 10000 nodes.

- **Output**
- For each tree, print the heights of the tree before and after the conversion specified above. Use the format:
- Tree *t*: *h*1 => *h*2
- where t is the case number (starting from 1), *h*1 is the height of the tree before the conversion, and *h*2 is the height of the tree after the conversion.

- Analysis
  - The problem is about the transformation from a tree into a binary tree.
  - For the figure showed in the problem, the depth-first traversal for the tree (a) is ‘dudduduudu’, and the transferred binary tree is showed in (b).


0
1 2 3
4 5
(a)
0
1
2
4 3
5
(b)

- Suppose the root is at level 0, and so on; the heights of the tree before and after the conversion are *height*1 and *height*2, respectively; the current level number is *level*.
- For each ‘d’ in a direction taken in a depth-first traversal of a tree, ‘d’ means down (*level*++). In Figure , the first ‘d’ visits vertex 1, and vertex 1 is at level 1; the second ‘d’ visits vertex 2 at level 2; the third ‘d’ visits vertex 4 at level 3, the fourth ‘d’ visits vertex 5 at level 4, and the fifth ‘d’ visits vertex 3 at level 3, from vertex 2.

- Node X's level number in the binary tree=
  - The level number for X's parent in the binary tree
  - +
  - the sequence number for X as a child in original tree

- Function *calc_height*(*level*, **height*1) is used to return the height of the transformed tree, where for the tree before it is transformed , the current level number is *level*, and the height of the tree is *height*1:
  - Initially the height of the transformed tree *height*=0;
  - If the current level number is larger than *height*1(*level*>**height*1), the height of the tree is adjusted (**height*1= *level*);
  - For each ‘d’ in a direction taken in a depth-first traversal of a tree (the number of current ‘d’ is *n*+1) (for (*n*=0; (*c*= fgetc(in))== 'd'; *n*++)): The current level number is *level*+1; *calc_height*(*level*+1, *height*1) runs recursively; get the height after transformation (*t*=*calc_height*(*level*+1, *height*1) + *n*+1). If (*t* > *height*), then *height* = *t*.
  - Return the height of the transformed tree *height*.

Paths of Binary Trees

- Paths in a tree are paths from the root to other nodes. In a tree there is no cycle, therefore there is unique path from the root to any node.

- In a complete binary tree with n nodes, nodes' numbers are from 0 to n-1, and nodes are numbered top-down, and from left to right.
  - For a node i, if it has parent, then its parent is node $\left\lfloor \frac{i-1}{2} \right\rfloor$ ($1 \le i \le n-1$); if it has left child ($2*i+1 \le n-1$), then its left child is node $2*i+1$; if it has right child ($2*i+2 \le n-1$), then its left child is node $2*i+2$.
  - If i is even and $i \neq 0$, the left sibling for node i is node i-1; and if i is odd and $i \neq n-1$, the right sibling for node i is node $i+1$.
  - The level where node i is at is $\lfloor \log_2(i+1) \rfloor$, that is, the length of the path from node i to the root is $\lfloor \log_2(i+1) \rfloor$.
- Based on it, a complete binary tree can be stored in an array.

Binary Tree

- **Source: TUD Programming Contest 2005 (Training Session), Darmstadt, Germany**
- **IDs for Online Judge: POJ 2499**

- Binary trees are a common data structure in computer science. In this problem we will look at an infinite binary tree where the nodes contain a pair of integers. The tree is constructed like this:
- The root contains the pair (1, 1).
- If a node contains (a, b) then its left child contains $(a + b, b)$ and its right child $(a, a + b)$.
- Given the contents (a, b) of some node of the binary tree described above, suppose you are walking from the root of the tree to the given node along the shortest possible path. Can you find out how often you have to go to a left child and how often to a right child?

- **Input**
- The first line contains the number of scenarios. Every scenario consists of a single line containing two integers i and j ($1 \leq i, j \leq 2*10^9$) that represent a node (i, j). You can assume that this is a valid node in the binary tree described above.

- **Output**
- The output for every scenario begins with a line containing "Scenario #*i*:", where *i* is the number of the scenario starting at 1. Then print a single line containing two numbers *l* and *r* separated by a single space, where *l* is how often you have to go left and *r* is how often you have to go right when traversing the tree from the root to the node given in the input. Print an empty line after every scenario.

- Analysis
- Becuase the root contains the pair (1, 1), and if a node contains (a, b) then its left child contains (a + b, b) and its right child (a, a + b); numbers in each pair are positive numbers, and for each pair, we can determine it is left child or right child by comparing the two numbers. For example, if a node contains (a + b, b), its parent must be (a, b) gotten by (a+b)- b. Therefore the path from the root to a node root can be gotten and how often you have to go to a left child and how often to a right child can be found out. The path is unique.

- For a pair (a, b), greedy algorithm is used to calculate how often you have to go to a left child and how often to a right child.
- When $a>b$, then from (a, b) it takes $\left\lfloor \frac{a-1}{b} \right\rfloor$ steps to the left, in each step the left parameter – b; otherwise it takes $\left\lfloor \frac{b-1}{a} \right\rfloor$ steps to the right, and in each step the right parameter – a; finally it reaches (1, 1).

Traversal of Binary Trees

- Preorder Traversal
- Inorder Traversal
- Postorder Traversal

- Preorder Traversal:
  - Visit the tree root;
  - Traverse the left subtree by recursively calling the preorder function;
  - Traverse the right subtree by recursively calling the preorder function;

- Inorder Traversal:
  - Traverse the left subtree by recursively calling the inorder function;
  - Visit the tree root;
  - Traverse the right subtree by recursively calling the inorder function;

- Postorder Traversal:
  - Traverse the left subtree by recursively calling the postorder function;
  - Traverse the right subtree by recursively calling the postorder function;
  - Visit the tree root;

- In preorder traversal, the tree root is visited firstly.
- In postorder traversal, the tree root is visited finally.
- In inorder traversal, the substring before the tree root is the result of inorder traversal for the left subtree, and the substring after the tree root is the result of inorder traversal for the right subtree.
  - Therefore the results of preorder traversal and inorder traversal, and the results of postorder traversal and inorder traversal, can determine the structure of a binary tree.
  - But the results of preorder traversal and postorder traversal can't determine the structure of a binary tree.

- **From the results of postorder traversal and inorder traversal of a binary tree, the result of preorder traversal of a binary tree can be gotten.**

- Suppose the result of inorder traversal for a binary tree s'=s_1'…s_k' …s_n', and the result of postorder traversal for a binary tree s"=s_1"……s_n". Obviously, from s_1"……s_n" produced by postorder traversal, s_n" is the tree root. In s_1'…s_k' …s_n' produced by inorder traversal there is a character s_k' which is equal to s_n", and if k>1, the left subtree exists and the substring s_1'…s_{k-1}' before s_k' is the result of inorder traversal for the left subtree, the prefix of the result of postorder traversal s_1"…s_{k-1}" is result of postorder traversal for the left subtree; and if k<n, the right subtree exists and the substring s_{k+1}' …s_n' after s_k' is the result of inorder traversal for the right subtree, and s_k"…s_{n-1}" is is the result of postorder traversal for the right subtree.
- If there exists the left subtree or the right subtree, the above process is called recursively.

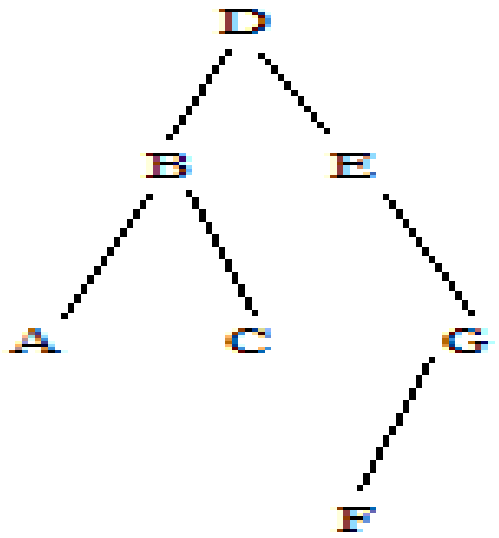
- **From the results of preorder traversal and inorder traversal of a binary tree, the result of postorder traversal of a binary tree can be gotten.**

- In like manner, the first character of the result of preorder traversal is the tree root. Suppose s'=s_1'…s_k' …s_n' is the result of the result of inorder traversal of a binary tree, and s"=s_1"……s_n" is the result of preorder traversal of a binary tree. Obviously s_1", the first character of the result of preorder traversal, is the tree root of the binary tree. Suppose s_k' equals to s_1" in s_1'…s_k' …s_n'.
- If k>1, the left subtree exists; s_1'…s_{k-1}' is the result of inorder traversal of the left subtree; and s_2"…s_k" is the result of preorder traversal of the left subtree.
- If k<n, the right subtree exists; s_{k+1}' …s_n' is the result of inorder traversal of the right subtree; and $s_{k+1"…sn}$" is the result of preorder traversal of the right subtree.
- If there exists the left subtree or the right subtree, the above process is called recursively. Finally the root s_1" (or s_k') is outputted.

Tree Recovery

- **Source: Ulm Local 1997**
- **IDs for Online Judge: POJ 2255, ZOJ 1944, UVA 536**

- Little Valentine liked playing with binary trees very much. Her favorite game was constructing randomly looking binary trees with capital letters in the nodes. This is an example of one of her creations:



- To record her trees for future generations, she wrote down two strings for each tree: a preorder traversal (root, left subtree, right subtree) and an inorder traversal (left subtree, root, right subtree). For the tree drawn above the preorder traversal is DBACEGF and the inorder traversal is ABCDEFG.
- She thought that such a pair of strings would give enough information to reconstruct the tree later (but she never tried it).

- Now, years later, looking again at the strings, she realized that reconstructing the trees was indeed possible, but only because she never had used the same letter twice in the same tree.
- However, doing the reconstruction by hand, soon turned out to be tedious. So now she asks you to write a program that does the job for her!

- **Input**
- The input will contain one or more test cases.
- Each test case consists of one line containing two strings preord and inord, representing the preorder traversal and inorder traversal of a binary tree. Both strings consist of unique capital letters. (Thus they are not longer than 26 characters.)
- Input is terminated by end of file.

- **Output**
- For each test case, recover Valentine's binary tree and print one line containing the tree's postorder traversal (left subtree, right subtree, root).

- Analysis
- Based on definitions of preorder traversal and inorder traversal, for a tree, the first character in preorder traversal is the root of the tree; and in inorder traversal, the string before the character is inorder traversal of its left subtree, the string after the character is inorder traversal of its right subtree.

- A recursive function *recover*($preord_l$, $preord_r$, $inord_l$, $inord_r$) is used to produce the tree's postorder traversal based on preorder traversal and inorder traversal of the tree, where preorder traversal of the tree is *preord*, $preord_l$ and $preord_r$ are pointers for the front pointer and the rear pointer respectively; and inorder traversal of the tree is *inord*, $inord_l$ and $inord_r$ are pointers for the front pointer and the rear pointer respectively.

- Calculation the root's position in the inorder traversal of the tree ($inord[root]==preord[preord_l]$);
- Calculate the size of the left subtree l_l ($root\text{-}inord_l$) and the size of the right subtree l_r ($inord_r\text{-}\ root$);
- If the left subtree isn't empty ($l_l>0$), then $recover(preord_l, preord_l+l_l, inord_l, root\text{-}1)$, where $preord_l$ and $preord_l+l_l$ are front pointer and rear pointer for preorder traversal of the left subtree; and $inord_l$ and $root$-1 are front pointer and rear pointer for inorder traversal of the left subtree;
- If the right subtree isn't empty ($l_r>0$), then $recover(preord_l+l_l+1, preord_r, root+1, inord_r)$, where $preord_l+l_l+1$ and $preord_r$ are front pointer and rear pointer for preorder traversal of the right subtree; and $root$+1 and $inord_r$ are front pointer and rear pointer for inorder traversal of the right subtree;
- Output the root $inord[root]$).