The Great Tree-List Recursion Problem

by Nick Parlante
nick.parlante@cs.stanford.edu

This article presents one of the neatest recursive pointer problems ever devised. This an advanced problem that uses pointers, binary trees, linked lists, and some significant recursion. This article includes the problem statement, a few explanatory diagrams, and sample solution code in Java and C. Thanks to Stuart Reges for originally showing me the problem.

Stanford CS Education Library Doc #109

This is article #109 in the Stanford CS Education Library -- http://cslibrary.stanford.edu/109/. This and other free educational materials are available at http://cslibrary.stanford.edu/. Permission is given for this article to be used, reproduced, or sold so long this paragraph and the copyright are clearly reproduced. Related articles in the library include Linked List Basics (#103), Linked List Problems (#105), and Binary Trees (#110).

Contents

1. Ordered binary tree
2. Circular doubly linked list
3. The Challenge
4. Problem Statement
5. Lessons and Solution Code

Introduction

The problem will use two data structures -- an ordered binary tree and a circular doubly linked list. Both data structures store sorted elements, but they look very different.

1. Ordered Binary Tree

In the ordered binary tree, each node contains a single data element and "small" and "large" pointers to sub-trees (sometimes the two pointers are just called "left" and "right"). Here's an ordered binary tree of the numbers 1 through 5...

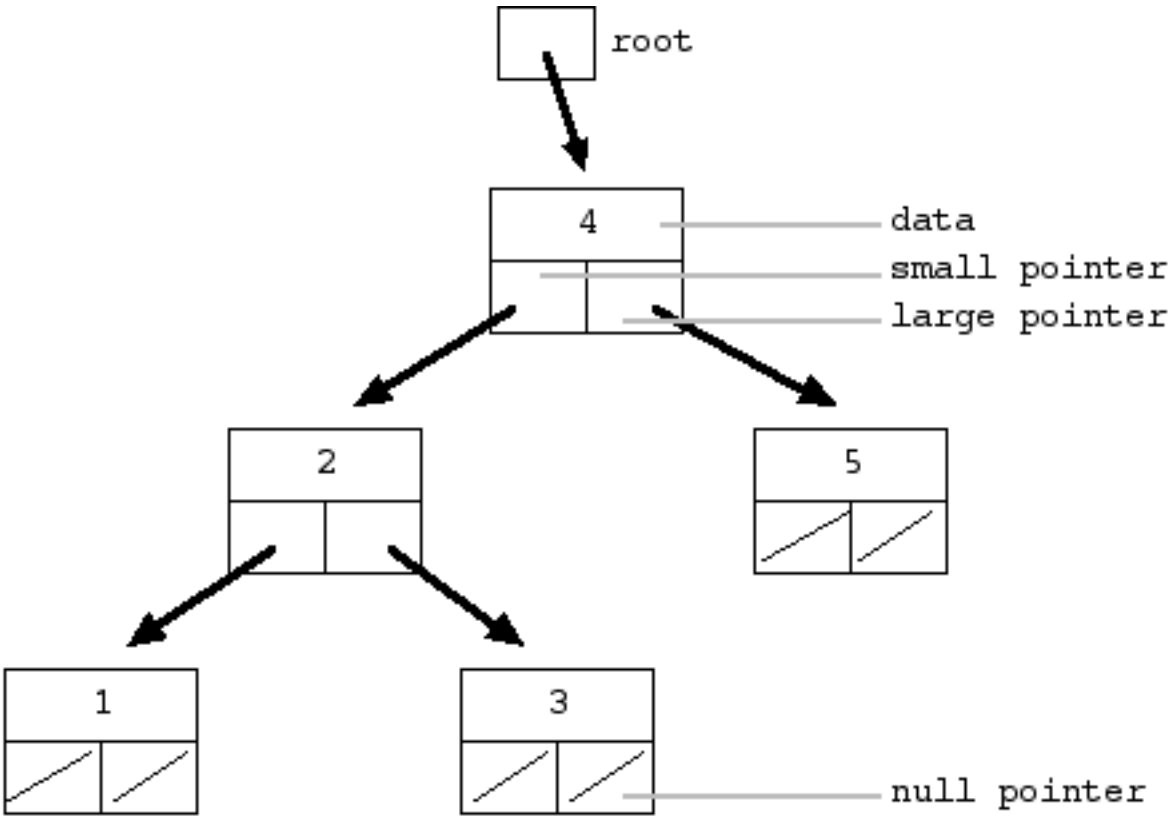

Figure-1 -- ordered binary tree

All the nodes in the "small" sub-tree are less than or equal to the data in the parent node. All the nodes in the "large" sub-tree are greater than the parent node. So in the example above, all the nodes in the "small" sub-tree off the 4 node are less than or equal to 4, and all the nodes in "large" sub-tree are greater than 4. That pattern applies for each node in the tree. A null pointer effectively marks the end of a branch in the tree. Formally, a null pointer represents a tree with zero elements. The pointer to the topmost node in a tree is called the "root".

2. Circular Doubly Linked List

Here's a circular doubly linked list of the numbers 1 through 5...

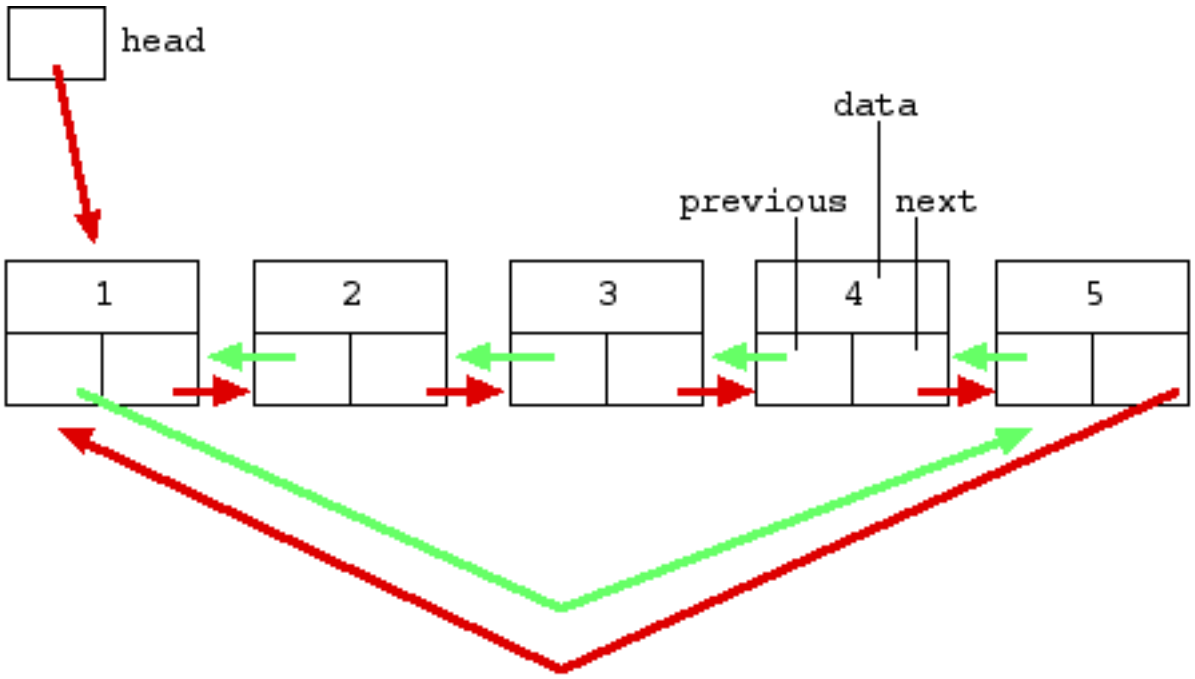

Figure-2 -- doubly linked circular list

The circular doubly linked list is a standard linked list with two additional features...

- "Doubly linked" means that each node has two pointers -- the usual "next" pointer that points to the next node in the list and a "previous" pointer to the previous node.
- "Circular" means that the list does not terminate at the first and last nodes. Instead, the "next" from the last node wraps around to the first node. Likewise, the "previous" from the first node wraps around to the last node.

We'll use the convention that a null pointer represents a list with zero elements. It turns out that a length-1 list looks a little silly...

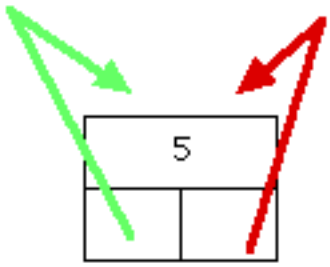

Figure-3 -- a length-1 circular doubly linked list

The single node in a length-1 list is both the first and last node, so its pointers point to itself. Fortunately, the length-1 case obeys the rules above so no special case is required.

The Trick -- Separated at Birth?

Here's the trick that underlies the Great Tree-List Problem: look at the nodes that make up the ordered binary tree. Now look at the nodes that make up the linked list. The nodes have the same type structure -- they each contain an element and two pointers. The only difference is that in the tree, the two pointers are labeled "small"

and "large" while in the list they are labeled "previous" and "next". Ignoring the labeling, the two node types are the same.

---

3. The Challenge

The challenge is to take an ordered binary tree and rearrange the internal pointers to make a circular doubly linked list out of it. The "small" pointer should play the role of "previous" and the "large" pointer should play the role of "next". The list should be arranged so that the nodes are in increasing order...

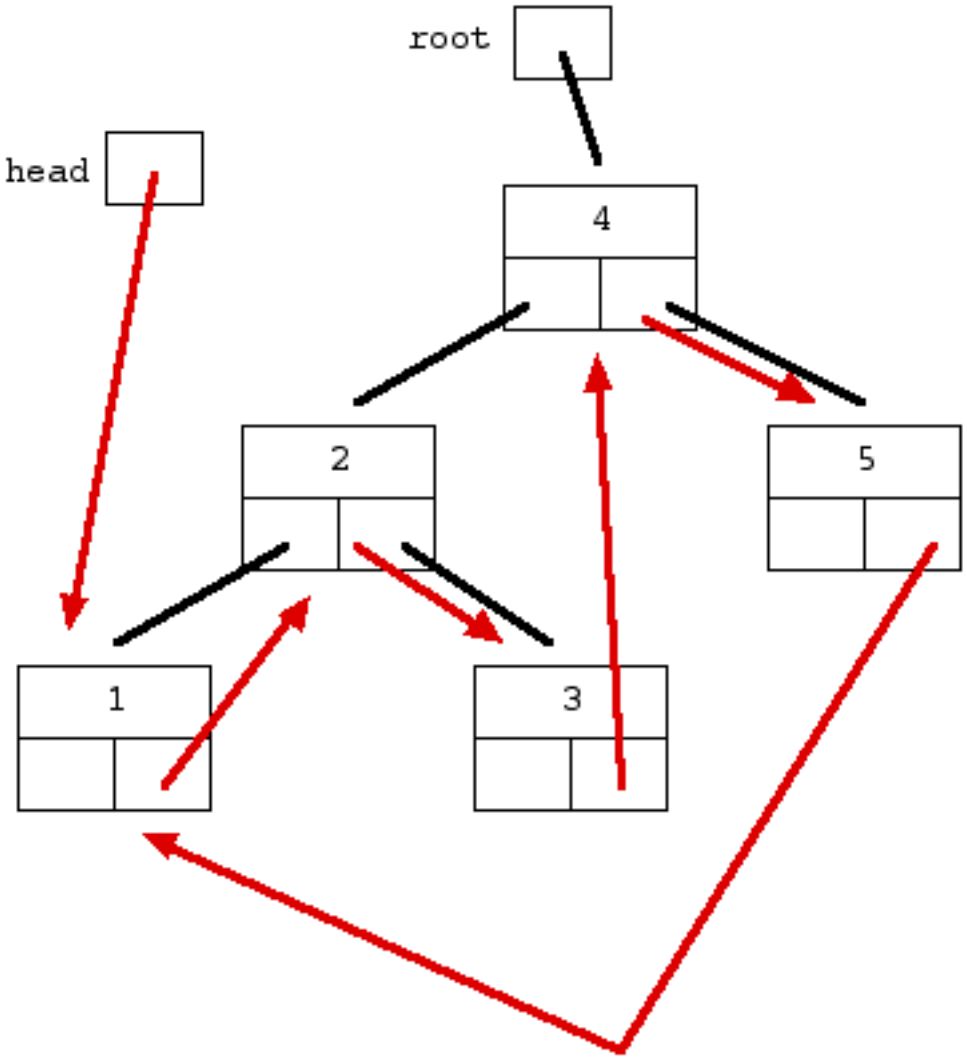

Figure-4 -- original tree with list "next" arrows added

This drawing shows the original tree drawn with plain black lines with the "next" pointers for the desired list structure drawn as arrows. The "previous" pointers are not shown.

Complete Drawing

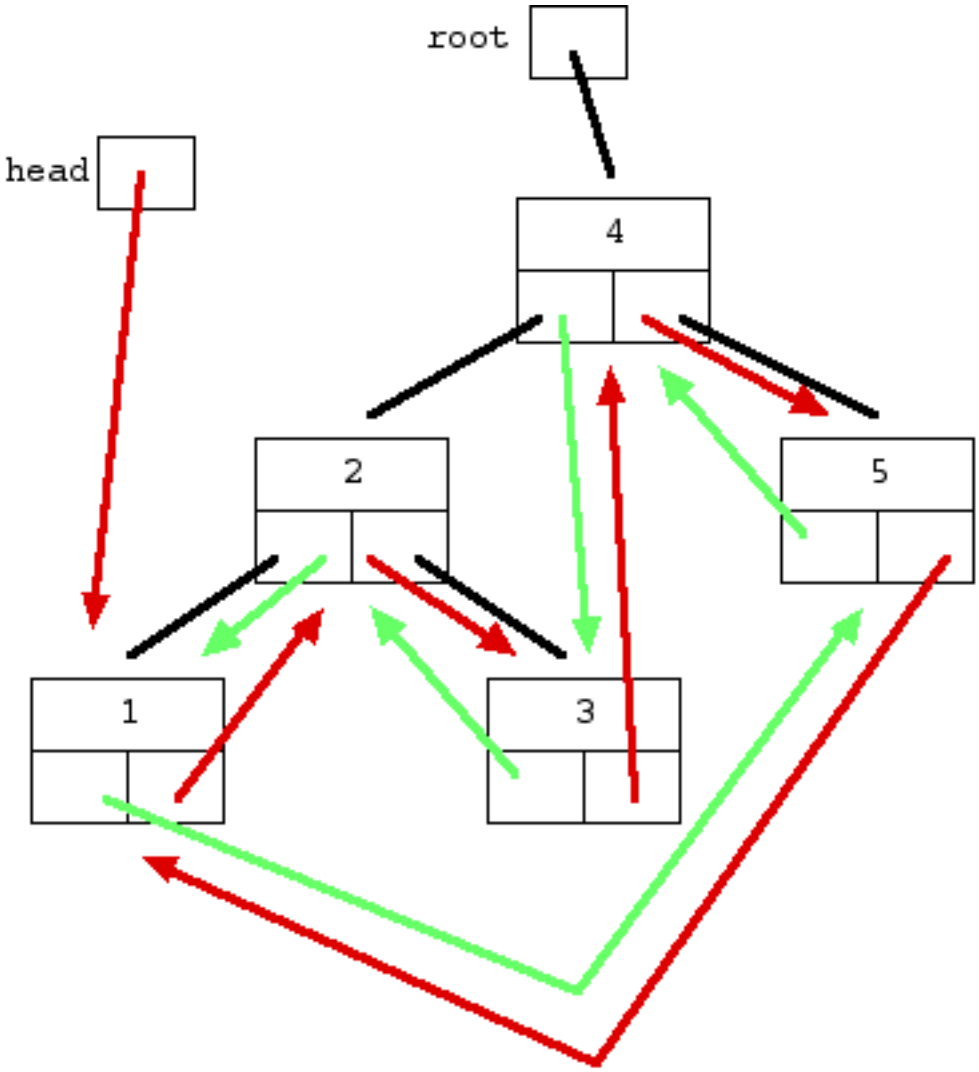

Figure-5 -- original tree with "next" and "previous" list arrows added

This drawing shows the all of the problem state -- the original tree is drawn with plain black lines and the desired next/previous pointers are added in as arrows. Notice that starting with the head pointer, the structure of next/previous pointers defines a list of the numbers 1 through 5 with exactly the same structure as the list in figure-2. Although the nodes appear to have different spatial arrangement between the two drawings, that's just an artifact of the drawing. The structure defined by the the pointers is what matters.

4. Problem Statement

Here's the formal problem statement: Write a recursive function `treeToList(Node root)` that takes an ordered binary tree and rearranges the internal pointers to make a circular doubly linked list out of the tree nodes. The "previous" pointers should be stored in the "small" field and the "next" pointers should be stored in the "large" field. The list should be arranged so that the nodes are in increasing order. Return the head pointer to the new list. The operation can be done in O(n) time -- essentially operating on each node once. Basically take figure-1 as input and rearrange the pointers to make figure-2.

Try the problem directly, or see the hints below.

Hints