Objective:

- Consolidating the concept and coding of Binary Trees by applying some very easy to hard operations on them.
- As a result, it will also make your grip on recursive thinking.

Note: All the following functions are supposed to be done on link based Binary Tree.

Problem-23 to onward are based on BST.

Problem-1:

int BinaryTree<T>::countNodes( )

Purpose: This function returns the number of nodes in the *this object.

Problem-2:

T BinaryTree<T>::minValue( )

Purpose: This function returns the minimum value from the tree.

Problem-3:

int BinaryTree<T>::countLeafNodes( )

Purpose: This function returns the number of leaf nodes in the *this object.

Problem-4:

void BinaryTree<T>::nonRecPreOrder( )

Purpose: This function does the non-recursive PreOrder traversal of the tree.

Problem-5:

void BinaryTree<T>::nonRecPostOrder( )

Purpose: This function does the non-recursive PostOrder traversal of the tree.

Problem-6:

void BinaryTree<T>::nonRecInOrder( )

Purpose: This function does the non-recursive InOrder traversal of the tree.

Problem-7:

int BinaryTree<T>::isComplete( )

Purpose: This function determines whether the tree is complete or not.

Problem-8:

int BinaryTree<T>::isFull( )

Purpose: This function determines whether the tree is full or not.

Problem-9:

int BinaryTree<T>::findBalanceFactor(T)

Purpose: This function finds the balance factor of a given node 'T'.

Balance Factor of a particular node is defined as the difference between the height of its left and right sub-tree.

Problem-10:

int BinaryTree<T>::isIsomorphic(BinaryTree<T>&)

Purpose: This function determines the equality of two rooted trees.

Problem-11:

int BinaryTree<T>::isBTIsomorphic(BinaryTree<T>&)

Purpose: This function determines the equality of two binary trees.

Problem-12:

BinaryTree<T> BinaryTree<T>::createClone( )

Purpose: This function return the deep clone of *this.

Problem-13:

BinaryTree<T> BinaryTree<T>::createMirrorClone( )

Purpose: This function return the mirror image of *this.
Note: *this object must not be changed

Problem-14:

int BinaryTree<T>::isMirror(BinaryTree<T>&)

Purpose: This function returns true if the *this and received object are mirror image of each other other false.

Problem-15:

int BinaryTree<T>::findNodeLevel(T)

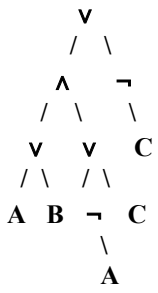
Purpose: This function finds that on which level number the given node 'T' exists?

Problem-16:

void BinaryTree<T>::displayLongestPath( )

Purpose: This function displays the longest path from root to leaf.

For Example: In the following Expression Tree representing the Boolean expression



The longest path is

∨ ---- ∧ ---- ∨ ----- ¬ ---- A

Problem-17:

void BinaryTree<T>::displayAllPaths( )

Purpose: This function displays the all possible paths from root to leaves. Obviously the number of paths will be equal to number of leaves (external node) in the tree.

For Example: In the above Expression Tree representing the Boolean expression; possible paths are as follows:

Path: ∨ ---- ∧ ---- ∨ ---- A
Path: ∨ ---- ∧ ---- ∨ ---- B
Path: ∨ ---- ∧ ---- ∨ ----- ¬ ---- A
Path: ∨ ---- ∧ ---- ∨ ----- C
Path: ∨ ---- ¬ ---- C

Problem-18:

void BinaryTree<T>::displayAllPathsLength( )

Purpose: This function displays the all possible paths from root to leaves. Obviously the number of paths will be equal to number of leaves (external node) in the tree.

The only difference in the previous and this function is the output which also shows length of each path.

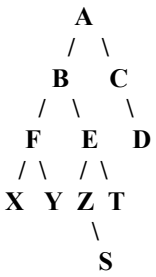
Path: ∨ ---- ∧ ---- ∨ ---- A = 4
Path: ∨ ---- ∧ ---- ∨ ---- B = 4
Path: ∨ ---- ∧ ---- ∨ ----- ¬ ---- A = 5
Path: ∨ ---- ∧ ---- ∨ ----- C = 4
Path: ∨ ---- ¬ ---- C = 3

Problem-19:

T BinaryTree<T>::lowestCommonAncestor(T a,T b)

Purpose: This function returns the lowest common ancestor of nodes 'a' and 'b'.

For Example: The lowest common ancestor of nodes 'F' and 'S' is 'B'
The lowest common ancestor of nodes 'Z' and 'T' is 'E'
The lowest common ancestor of nodes 'D' and 'S' is 'A'
The lowest common ancestor of nodes 'A' and 'D' is none



Problem-20:

int BinaryTree<T>::findDistance(T a,T b)

Purpose: This function returns the distance between the nodes 'a' and 'b'.
Let d_a denote the depth of 'a' in tree. The distance between two nodes a and b in tree is $d_a + d_b - 2d_x$, where 'x' is the LCA (lowest common ancestor) 'x' of 'a' and 'b'.

Problem-21:

int BinaryTree<T>::findDiameter(T a,T b)

Purpose: This function returns the maximum distance between two nodes in 'a' and 'b'.

Problem-22:

The following are some of the problems that came in Programming Competition related to Trees.
http://uva.onlinejudge.org/external/1/112.html
http://www.topcoder.com/stat?c=problem_statement&pm=3093&rd=5864
http://www.topcoder.com/stat?c=problem_statement&pm=3025&rd=5860
http://uva.onlinejudge.org/external/4/484.html

Explore the following tasks when you will be done with Binary Search Tree

Problem-23:

int BST<T>::countTrees(int numKeys)

This is not a binary tree-programming problem in the ordinary sense -- it's more of a math/combinatorics recursion problem that happens to use binary trees. (Thanks to Jerry Cain for suggesting this problem.)

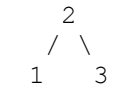
Suppose you are building an N node binary search tree with the values 1..N. How many structurally different binary search trees are there that store those values? Write a recursive function that, given the number of distinct values, computes the number of structurally unique binary search trees that store those values. For example, countTrees(4) should return 14, since there are 14 structurally unique binary search trees that store 1, 2, 3, and 4. The base case is easy, and the recursion is short but dense. Your code should not construct any actual trees; it's just a counting problem.

Problem-24:

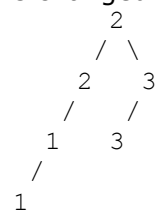
doubleTree()

For each node in a binary search tree, create a new duplicate node, and insert the duplicate as the left child of the original node. The resulting tree should still be a binary search tree.

So the tree...



is changed to...



Problem-25:

Binary Search Tree Checking (for *problem 3*)

This background is used by the next two problems: Given a plain binary tree, examine the tree to determine if it meets the requirement to be a binary search tree. To be a binary search tree, for every node, all of the nodes in its left tree must be <= the node, and all of the nodes in its right subtree must be > the node. Consider the following four examples...

```
a.  5    -> TRUE
   / \
  2   7


b.  5    -> FALSE, because the 6 is not ok to the left of the 5
   / \
  6   7


c.   5   -> TRUE
    / \
   2   7
  /
 1

d.   5   -> FALSE, the 6 is ok with the 2, but the 6 is not ok with the 5
    / \
   2   7
  / \
 1   6
```

For the first two cases, the right answer can be seen just by comparing each node to the two nodes immediately below it. However, the fourth case shows how checking the BST quality may depend on nodes which are several layers apart -- the 5 and the 6 in that case.

Problem-3

isBST() -- version 1

Suppose you have helper functions minValue() and maxValue() that return the min or max int value from a non-empty tree (see problem 3 above). Write an isBST() function that returns true if a tree is a binary search tree and false otherwise. Use the helper functions, and don't forget to check every node in the tree. It's ok if your solution is not very efficient. (Thanks to Owen Astrachan for the idea of having this problem, and comparing it to problem 14)

Returns true if a binary tree is a binary search tree.

```
int isBST(struct node* node) {
```

Problem-3

isBST() -- version 2

Version 1 above runs slowly since it traverses over some parts of the tree many times. A better solution looks at each node only once. The trick is to write a utility helper function isBSTRecur(struct node* node, int min, int max) that traverses down the tree keeping track of the narrowing min and max allowed values as it goes, looking at each node only once. The initial values for min and max should be INT_MIN and INT_MAX -- they narrow from there.

```
/*
 Returns true if the given tree is a binary search tree
```

```
 (efficient version).
*/
int isBST2(struct node* node) {
  return(isBSTRecur(node, INT_MIN, INT_MAX));
}

/*
 Returns true if the given tree is a BST and its
 values are >= min and <= max.
*/
int isBSTRecur(struct node* node, int min, int max);
```

Problem-26:

[http://cslibrary.stanford.edu/109/TreeListRecursion.html]

DNode BinaryTree<T>::treeToList(BinaryTree<T> &)

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 advanced problem that uses pointers, binary trees, linked lists, and some significant recursion. This article includes the problem statement, a few explanatory diagrams. Thanks to Stuart Reges for originally showing me the problem.

Contents

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...

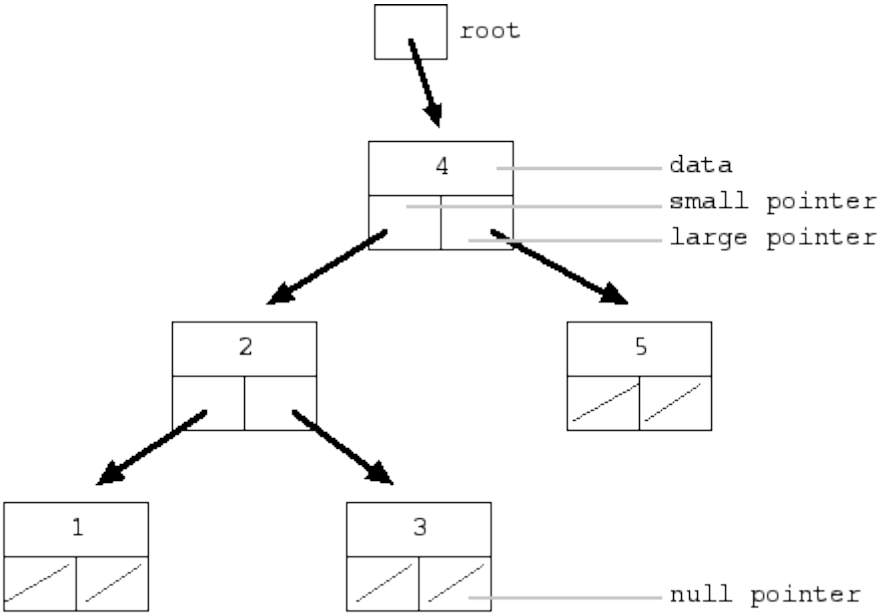


Figure-1 -- ordered binary tree

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...

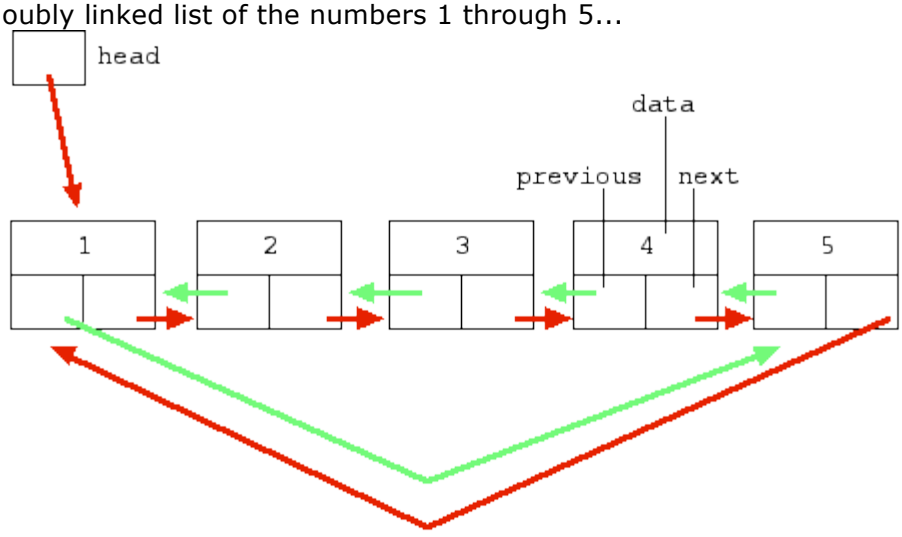


Figure-2 -- doubly linked circular list

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...

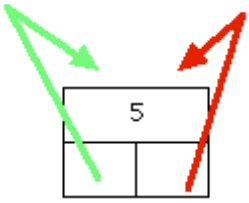


Figure-3 -- a length-1 circular doubly linked list

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...

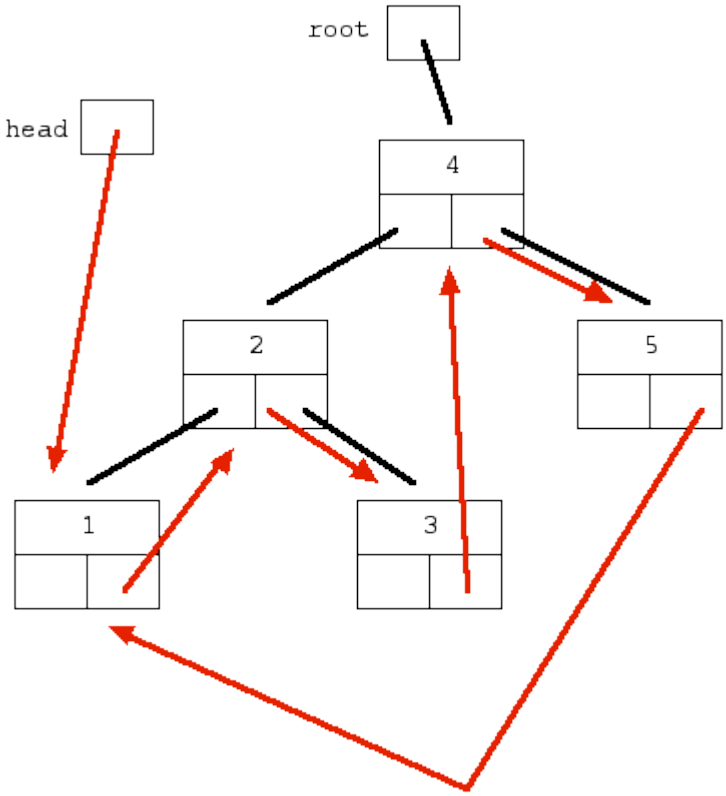


Figure-4 -- original tree with list "next" arrows added

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

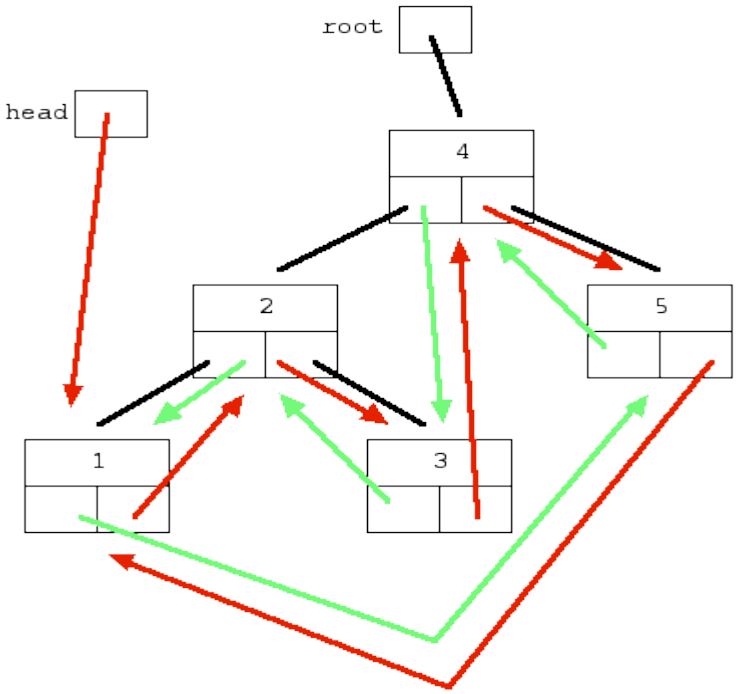


Figure-5 -- original tree with "next" and "previous" list arrows added

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

Hint #1

The recursion is key. Trust that the recursive call on each sub-tree works and concentrate on assembling the outputs of the recursive calls to build the result. It's too complex to delve into how each recursive call is going to work -- trust that it did work and assemble the answer from there.

Hint #2

The recursion will go down the tree, recursively changing the small and large sub-trees into lists, and then append those lists together with the parent node to make larger lists. Separate out a utility function append(Node a, Node b) that takes two circular doubly linked lists and appends them together to make one list which is returned. Writing a separate utility function helps move some of the complexity out of the recursive function.
