Indexing a Binary Search Tree

Given that an in-order traversal of a binary search tree $T$ produces the elements in sorted order, it is natural to ask for an element's index in this sorted list. Define:

**INDEX($d,v,T$):** Return the index of $d$ in the sorted list of data items stored the subtree of $T$ rooted at $v$ (where 1 is the index of the first element), or 0 if $d$ is not in this subtree.

**Example:**

```
            horse
           /   \
        cat    zebra
       /   \
    ant    dog
       /   \
      goat
```

*sorted order:* ant, cat, dog, goat, horse, zebra
*index:* 1, 2, 3, 4, 5, 6

Idea:

- \( \text{COUNT}(w) \) stores the number of vertices in the subtree rooted at \( w \) (including \( w \)).

- If \( d < \text{DATA}(v) \) then the index of \( d \) in the subtree rooted at \( v \) is the same as the index of \( d \) in the subtree rooted at \( \text{LCHILD}(v) \).

- If \( d = \text{DATA}(v) \) then it is \( \text{COUNT}(\text{LCHILD}(v)) + 1 \), otherwise compute the index of \( d \) in the right subtree of \( v \) and add that to \( \text{COUNT}(\text{LCHILD}(v)) + 1 \).
Recursive Algorithm

Note: Define $COUNT(nil)=0$.

function INDEX($d,v$)
    if $v=nil$ then return 0
    else if $d<DATA(v)$ then return INDEX($d,LCHILD(v)$)
    else if $d=DATA(v)$ then return COUNT(LCHILD($v$))+1
    else begin
        $i:=\text{INDEX}(d,RCHILD(v))$
        if $i=0$ then return 0 else return COUNT(LCHILD($v$))+1+$i$
    end
end
(indexing a BST continued)

Non-recursive algorithm: Move down from $v$, keeping a total of counts added thus far in the variable $i$; return the index of $d$ when it is found, or return 0 if the while loop falls out the bottom of the tree.

function INDEX($d,v$)  
  $i := 0$

  while $v \neq \text{nil}$ do begin
    if $d < \text{DATA}(v)$ then $v := \text{LCHILD}(v)$
    else begin
      $i := i + \text{COUNT(LCHILD}(v)) + 1$
      if $d = \text{DATA}(v)$ then return $i$ else $v := \text{RCHILD}(v)$
    end
  end

  return 0
end

Maintaining the COUNT fields: INSERT and DELETE can be augmented to increment or decrement the counts along the corresponding root-to-leaf path; operations like JOIN and SPLIT can also be adapted (see the exercises).