Inorder Search

Burton Rosenberg
Dept. of Math and Comp. Sci.
University of Miami

Definition 1 (Left to Right Order) *Let (N, λ, ρ) be a tree. For $n', n'' \in N$, n' is said to be to the left of n'', denoted $n' <_L n''$, if there exists an $n \in N$ such that $n' \in L(n)$ and $n'' \in R(n)$.*

Lemma 1 *The relation $<_L$ is transitive.*

PROOF: Suppose we have n', n'' and $n''' \in N$ with $n' <_L n''$ and $n'' <_L n'''$. Then there exists $n_1, n_2 \in N$ with $n' \in L(n_1)$, $n'' \in R(n_1)$, $n'' \in L(n_2)$ and $n''' \in R(n_2)$. Note that $n_1 \neq n_2$ and either $n_1 \rightarrow^* n_2$ or $n_2 \rightarrow^* n_1$. Else there are distinct paths from the root to n_1 and n_2, but there are paths from each n_1 and n_2 to n'', which would give two different paths from the root to n''.

This implies that either $n_1 \in L(n_2)$ or $n_2 \in R(n_1)$. In the first case, $n' \in L(n_2)$ and $n''' \in R(n_2)$, hence $n' <_L n'''$. The second case is similar.

Definition 2 (Preorder) *There is a total order on the tree $\leq_{pre}$, called the preorder, defined by,*

1. *Reflexivity: $\forall n \in N,\ n \leq_{pre} n$.*
2. *Parents first: $\forall n \in N$ and all $n' \in L(n) \cup R(n)$, $n \leq_{pre} n'$.*
3. *Leftmost first: $\forall n', n'' \in N$ with $n' <_L n''$, then $n' \leq_{pre} n''$.*

Suppose we wish to "paint green" the nodes of a tree such that the nodes are painted in preorder. That is, if $n \leq_{\text{pre}} n'$, for $n, n' \in N$, n gets painted green before n'. The following recursive algorithm will do the trick:

```
preorder( n: n is a node or null ) {
   if n is null, return
   else n is a node {
      paint n green
      preorder( the left child of n )
      preorder( the right child of n )
   }
   return
}
```

It is clear that the nodes are painted in preorder: the parent is painted before any children, and all children to the left of the parent are painted before any child to the right of the parent is painted. However, it is important to understand how the algorithm can jump inside itself, re-running the preorder code, but not get confused concerning the local variable n.

Recursive algorithms reuse code, but instantiate fresh local variables for each reuse.

Using a stack, we can remove the recursion.

```
set n to the root
while ( true ) {
   if n is null {
      if stack is empty, then break
      else pop the stack into n
   }
   else {
      paint n green
      push the right child of n on the stack
      set n to the left child of n
   }
} end of while loop
```

Having derived this algorithm from the recursive algorithm its correctness follows from,

1. That the recursive version is correct.
2. That the transformation to a non-recursive version did not introduce incorrectness.

Another manner to show correctness of algorithms with loops is to establish a *Loop Invariant.* A Loop Invariant is a true statement about the world, it is a logical equation of balance for your data structures. It is disturbed during the body of the loop, but before the loop body is done, the invariant will be reestablished. You know that the Invariant is true going into the loop, and again when coming out. Supposedly, the temporary inbalance of the data structure is the result of pushing the algorithm on ahead towards its goal.

Here is an Invariant for the above algorithm:

Definition 3 (Example Loop Invariant) *In the previous algorithm, nodes are unpainted, green, red or blue. At first all nodes are unpainted. The algorithm specifies when a node is painted green. When a node becomes n it is painted red. When a node is pushed on the stack it is painted blue. For the sake of the proof, each null child is considered an actual node.*

- *The path from the root to the red node is all green.*
- *Any and all right children of green nodes on this path are blue, and no other node is blue.*
- *Descendants from red and blue nodes are not painted, all other nodes are painted.*

- *The blue nodes are on the stack in Left to Right order.*

From this invariant we argue that the algorithm is correct. First we ascertain that the invariant is indeed invariant. Then we show how the invariant implies correctness. This is the value of an invariant, it breaks the correctness proof into two parts, offering us us a resting place at the invariant while woring towards the proof.

We leave it to the reader to argue invariance. From the invariant we need to argue that the green nodes are painted in preorder and that all nodes are green at the termination of the algorithm. Supposing that so far through the loop the nodes have been painted in the correct order, and red node is not null, the red node is the next in preorder to paint because all other non-green nodes are either a descendant of the red node, blue, or a descendant of a blue node, which are all to the right of the red node. If the red node is null, the stack is popped, making a blue node red, without harming the invariant, that is, the remaining blue nodes are to the right of this red node.

When the stack is empty, the collection of unpainted nodes includes only the descendants of the red node. Therefore, when the red node is null and the stack is empty, all nodes have been painted, none are red or blue, hence all are green.