This short article provides a quick summary of how to traverse a tree data-structure by means of a space-minimal stack.

Let us first define the set $\mathbb{S}$:

$$\mathbb{S} := \bigcup_{n \in \mathbb{N}} \mathbb{N}^n$$

to be the set of all finite length sequences of natural numbers, to be interpreted as all possible stacks.

Let *iterator* be a tree-node iterator of a given tree:

$$iterator := \left\{ \begin{array}{cccl} \text{stack} & \in \mathbb{S} & : & \text{is a stack holding a path of the current iterator's node from its root.} \\ \text{children} & \in \mathbb{N} & : & \text{is the number of children of th current iterator's node.} \\ \text{push}(c) & \mathbb{Z} \to \emptyset & : & \text{if } c < \text{children, then this method moves the iterator forward} \\ & & & \text{by updating children to the number of its } c\text{th child,} \\ & & & \text{as well as pushes } c \text{ onto stack; otherwise it does nothing.} \\ \text{pop}(h) & \mathbb{S} \to \mathbb{N} & : & \text{if } h\text{.stack is a proper substack, this method moves backward} \\ & & & \text{by updating children to the number of its parent, as well as pops} \\ & & & \text{the top } c \text{ of stack and returns it; otherwise it only returns} \\ & & & \text{the value } -1. \end{array} \right\}$$

Notice it is implicitly assumed that the methods have a direct access means of determining the number of children of its parent or given child (assuming they exist) with no additional information than what is on its stack. Furthermore, notice *pop* takes what seems to be an unnecessary argument, but is in fact defined this way for *genericicity*. This way, you can specify any node to be the "root" for this class of iterators. In practice I'm sure it could be coded as a compile-time parameter rather than a run-time parameter.

If *head* is a node of a given tree, with $next = 0$ and $move \in \mathbb{N}$ (arbitrary), then the given algorithm:

———

```
current := head

while (current ≠ head ∧ move ≠ −1∧ next ≠ head.children)
        if (next < current.children)
                current.push(move := next)
                next := 0
        else
                move := −1
                next := current.pop(head) +1
```

———

1. Halts for all finite trees, with final state next = head.children; move = -1; current = head.

2. Traverses all nodes possessing *head* as a substack, furthermore, with a shortest number of moves.

Proof The proof is by mathematical induction on the depth of the assumed tree (relative to *head*).

depth = 0: We enter the while loop (move $\neq -1$). Given depth = 0 (head only) we know that head.children = 0 as well as next = 0. Thus by the initial assumptions, we move to the *else* clause and set move = -1, but since current = head, by the definition of *pop* we only return -1 and thus next = 0.

The conditions for the while loop are no longer satisified and thus the algorithm halts. By default, it is noted that we have iterated the one-and-only node possessing head as a substack, and that we didnt actually move. This is to say the number of movements made in this iteration is zero, which is a minimum.

And so concludes this portion of the proof.

depth $\leq d \Longrightarrow$ **depth** $d+1$: We enter the while loop (move $\neq -1$). Given that depth $= d+1 > 0$ we know that

$$0 = \text{next} < \text{head.children}$$

and so we enter the *if* clause. We set move = next, and push move onto the stack. Finally, we set next = 0.

This is to say, we visit the 0th child of head in a single move. From there—from an inductive step point of view—we have a new head, call it *head*0 with the conditions that next = 0 and move $\in \mathbb{N}$. This subtree of head is of depth at most d and so by the inductive step we can reapply the algorithm to this subtree with head0 as its root and minimalistically traverse and halt in final state next = head0.children; move = -1; current = head0.

We return our original loop conditions and find that we now enter it again: (current $\neq$ head). We notice

$$\text{next} = \text{head0.children} \not< \text{current.children}$$

and so we enter the *else* clause. We set move = -1, pop 0 off the stack, set next = 1, and current = head.

If there are no more children, we are done. If not, we repeat this process until next = head.children, in which case the algorithm halts. As such, we have visited each child and traversed it; moreover, for each such traversal, we made exactly one move from our head to the given child, and exactly one move back: thus we have made a bare minimum journey. In any case, the conditions are now met, completing our proof. ∎

As a corollary, to traverse an entire tree, set head = root and for simplicity set move = 0. Any implementation of this algorithm requires minimal translation, for which a new proof only need consider the translation details.