Help me understand Inorder Traversal without using recursion

I am able to understand preorder traversal without using recursion, but I'm having a hard time with inorder traversal. I just don't seem to get it, perhaps, because I haven't understood the inner working of recursion.

This is what I've tried so far:

```
def traverseInorder(node):
    lifo = Lifo()
    lifo.push(node)
    while True:
        if node is None:
            break
        if node.left is not None:
            lifo.push(node.left)
            node = node.left
            continue
        prev = node
        while True:
            if node is None:
                break
            print node.value
            prev = node
            node = lifo.pop()
        node = prev
        if node.right is not None:
            lifo.push(node.right)
            node = node.right
        else:
            break
```

The inner while-loop just doesn't feel right. Also, some of the elements are getting printed twice; may be I can solve this by checking if that node has been printed before, but that requires another variable, which, again, doesn't feel right. Where am I going wrong?

I haven't tried postorder traversal, but I guess it's similar and I will face the same conceptual blockage there, too.

Thanks for your time!

P.S.: Definitions of `Lifo` and `Node` :

```
class Node:
    def __init__(self, value, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right

class Lifo:
    def __init__(self):
        self.lifo = ()
    def push(self, data):
        self.lifo = (data, self.lifo)
    def pop(self):
        if len(self.lifo) == 0:
            return None
        ret, self.lifo = self.lifo
        return ret
```

python | algorithm | tree | tree-traversal | non-recursive

asked **Jan 22 '10 at 10:44**

artknish
2,548 3 21 50
100% accept rate

feedback

7 Answers

Start with the recursive algorithm (pseudocode) :

```
traverse(node):
  if node != None do:
    traverse(node.left)
    print node.value
    traverse(node.right)
  endif
```

This is a clear case of tail recursion, so you can easily turn it into a while-loop.

```
traverse(node):
  while node != None do:
    traverse(node.left)
    print node.value
    node = node.right
  endwhile
```

You're left with a recursive call. What the recursive call does is push a new context on the stack, run the code from the beginning, then retrieve the context and keep doing what it was doing. So, you create a stack for storage, and a loop that determines, on every iteration, whether we're in a "first run" situation (non-null node) or a "returning" situation (null node, non-empty stack) and runs the appropriate code:

```
traverse(node):
  stack = []
  while !empty(stack) || node != None do:
    if node != None do: // this is a normal call, recurse
      push(stack,node)
      node = node.left
    else // we're returning: pop and print the current node
      node = pop(stack)
      print node.value
      node = node.right
    endif
  endwhile
```

The hard thing to grasp is the "return" part: you have to determine, in your loop, whether the code you're running is in the "entering the function" situation or in the "returning from a call" situation, and you will have an `if/else` chain with as many cases as you have non-terminal recursions in your code.

In this specific situation, we're using the node to keep information about the situation. Another way would be to store that in the stack itself (just like a computer does for recursion). With that technique, the code is less optimal, but easier to follow

```
traverse(node):
  // entry:
  if node == NULL do return
  traverse(node.left)
  // after-left-traversal:
  print node.value
  traverse(node.right)

traverse(node):
   stack = [node,'entry']
   while !empty(stack) do:
     [node,state] = pop(stack)
     switch state:
       case 'entry':
         if node == None do: break; // return
         push(stack,[node,'after-left-traversal']) // store return address
         push(stack,[node.left,'entry']) // recursive call
         break;
       case 'after-left-traversal':
         print node.value;
         // tail call : no return address
```

```
        push(stack,[node.right,'entry']) // recursive call
      end
    endwhile
```

answered **Jan 22 '10 at 11:05**

Victor Nicollet
12.4k 8 32

@Victor: Thank you! Your hint to think about the parts of code that have to be run in "entering-the-function" situation and "returning-from-a-call" situation helped me understand intuitively. Also, thanks for the intermediate step where you unwound the tail-recursion; I've heard about it, but seeing it in action helped a lot! – artknish Jan 22 '10 at 14:05

That's a nice explanation... I figured the same in an difficult way.. But the above way of step-by-step breakdown has made it understand very simpler – Harish Aug 16 '10 at 4:44

feedback

Here is a simple in-order non-recursive c++ code ..

```
void inorder (node *n)
{
    stack s;

    while(n){
        s.push(n);
        n=n->left;
    }

    while(!s.empty()){
        node *t=s.pop();
        cout<<t->data;
        t=t->right;

        while(t){
            s.push(t);
            t = t->left;
        }
    }
}
```

edited **Feb 17 at 6:00**

answered **Feb 6 '11 at 19:55**

EmAdpres
55 6

feedback

```
def traverseInorder(node):
   lifo = Lifo()

  while node is not None:
    if node.left is not None:
       lifo.push(node)
       node = node.left
       continue

   print node.value

   if node.right is not None:
      node = node.right
      continue

   node = lifo.Pop()
   if node is not None :
      print node.value
      node = node.right
```

PS: I don't know Python so there may be a few syntax issues.

answered **Jan 22 '10 at 11:06**

Henk Holterman
90.2k 5 48 117

Yeah, your continues are pointless. :) – Lennart Regebro Jan 22 '10 at 11:10

feedback

State can be remembered implicitly,

```
traverse(node) {
   if(!node) return;
   push(stack, node);
   while (!empty(stack)) {
     /*Remember the left nodes in stack*/
     while (node->left) {
        push(stack, node->left);
        node = node->left;
      }

      /*Process the node*/
      printf("%d", node->data);

      /*Do the tail recursion*/
      if(node->right) {
         node = node->right
      } else {
         node = pop(stack); /*New Node will be from previous*/
      }
    }
 }
```

answered **Mar 1 '10 at 3:40**
user283368

feedback

I think part of the problem is the use of the "prev" variable. You shouldn't have to store the previous node you should be able to maintain the state on the stack (Lifo) itself.

From Wikipedia, the algorithm you are aiming for is:

1. Visit the root.
2. Traverse the left subtree
3. Traverse the right subtree

In pseudo code (disclaimer, I don't know Python so apologies for the Python/C++ style code below!) your algorithm would be something like:

```
lifo = Lifo();
lifo.push(rootNode);

while(!lifo.empty())
{
    node = lifo.pop();
    if(node is not None)
    {
        print node.value;
        if(node.right is not None)
        {
            lifo.push(node.right);
        }
        if(node.left is not None)
        {
```

```
            lifo.push(node.left);
        }
    }
}
```

For postorder traversal you simply swap the order you push the left and right subtrees onto the stack.

answered **Jan 22 '10 at 11:07**

Paolo
8,585 13 20

@Paolo: This is pre-order traversal, not in-order. Thanks for your reply, anyway :) – artknish Jan 22 '10 at 18:53

D'oh! Mis-read the question... – Paolo Jan 22 '10 at 20:11

feedback

@Victor, I have some suggestion on your implementation trying to push the state into the stack. I don't see it is necessary. Because every element you take from the stack is already left traversed. so instead of store the information into the stack, all we need is a flag to indicate if the next node to be processed is from that stack or not. Following is my implementation which works fine:

```
def intraverse(node):
    stack = []
    leftChecked = False
    while node != None:
        if not leftChecked and node.left != None:
            stack.append(node)
            node = node.left
        else:
            print node.data
            if node.right != None:
                node = node.right
                leftChecked = False
            elif len(stack)>0:
                node = stack.pop()
                leftChecked = True
            else:
                node = None
```

answered **Jul 31 '11 at 2:25**

Leonmax
1

feedback

```
def print_tree_in(root):
    stack = []
    current = root
    while True:
        while current is not None:
            stack.append(current)
            current = current.getLeft();
        if not stack:
            return
        current = stack.pop()
        print current.getValue()
        while current.getRight is None and stack:
            current = stack.pop()
            print current.getValue()
        current = current.getRight();
```

answered **Jan 2 at 18:44**

Ashish

1

feedback

Not the answer you're looking for? Browse other questions tagged python algorithm tree tree-traversal non-recursive **or ask your own question.**

question feed