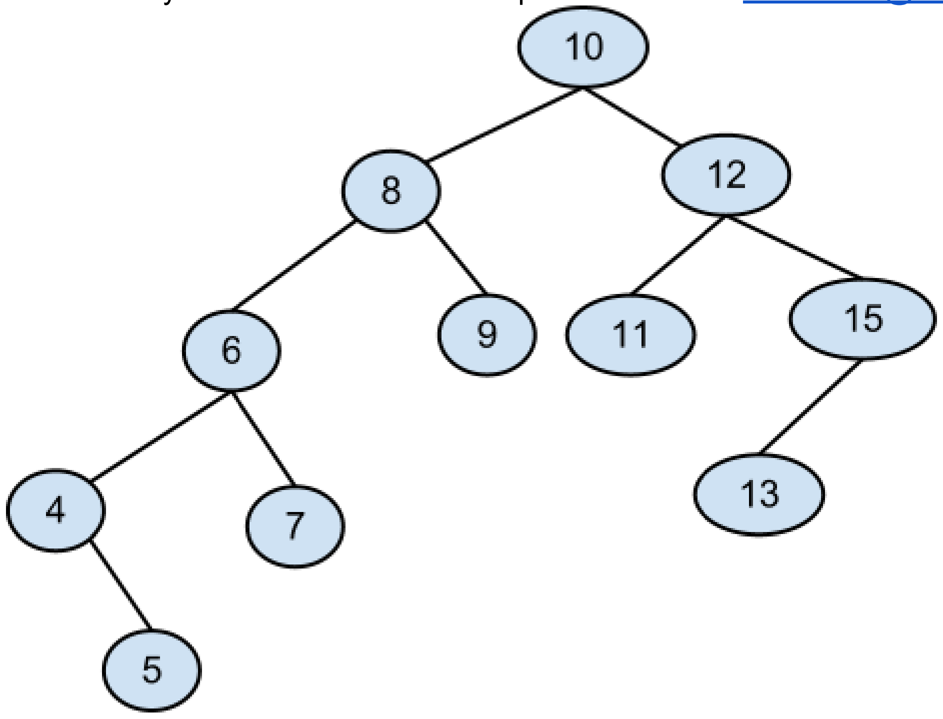
O(ME)

SENTHU SIVASAMBU

Should you need to contact me drop me an email at sivasambu@msn.com



The problem we are tackling here is how to do an in-order traversal - remember the in-order traversal is a scheme of traversing the key where a left node is traversed first, followed by parent node and then the right node.

First let's take a look at a recursive solution.

```
IN-ORDER(T)
1. p <-- T
2. IF p == NIL
3.        RETURN
2. IN-ORDER(p.left)
3. print p.key
5. IN-ORDER(p.right)
```

The nice thing about a recursive solution is that it implicitly uses a (function / method) stack to enforce the in-order traversal.

So, what do you think could be a potential problem with this solution - well in the worst case scenario of a binary tree its depth could be bound from above by O(n). So if n is large then our solution above will soon eat up all the spaces in the stack (memory).

Therefore an iterative solution is preferable. There is an algorithm commonly known as Morris Tree Traversal algorithm - the issue with this algorithm is that it modifies the tree temporarily - though it is reverted back to original state at the end.

Following is my algorithm for traversing a tree without modifying it - even temporarily and using constant additional space.

If someone were to ask you to write this in an interview - the problem could be coming up with a solution addressing all the boundary conditions within certain time - this is only achievable if you have the solution in the memory. You see the problem with me is that memory is not my strongest natural skill. Anyway, I am sure, given enough time anyone could come up with a solution.

Let's take a look at how I have approached this problem

- If left child is the one supposed to be printed first, then left most child of the left sub-tree rooted at T.root is what supposed to be printed first.
- It also makes sense to think that we will have to find the left most nodes in the right sub-tree as well.
- Therefore let's make finding the left most node a subroutine.
- now, once I am at the left most child node, I need to repetitively traverse the tree in order. What that means is I am going to need a loop, a termination condition, and possibly one or more chase pointers.
- I often find, it is easy to work on an example tree as given above and then enhance it to cover all the scenarios.

```
IN-ORDER(T)
1. p <-- T.root
2. rootCount = 0
3. p <-- FIND-LEFT-MOST(p)
4. WHILE p != NIL
5.      print p.key
6.      IF  p.right != NIL
7.           p <-- FIND-LEFT-MOST(p.right)
8.      ELSE
9.           IF p.π.right == p    // π denotes the parent, a convention inherited from CLRS
10.               p <-- RIGHT-GO-UP(p)
```

```
11.             IF p == T.root
12.                 rootCount = rootCount + 1
13.                 IF rootCount == 2  //is there a better way to detect this?
14.                     RETURN
15.         ELSE  // current is the left child
16.             p <-- p.π
```

And that’s it - just 16 lines :).

We use two sub routines here. FIND-LEFT-MOST(..) - this subroutine finds the left most child of a tree node. if there is no left child it returns current, so it looks like following

```
FIND-LEFT-MOST(T)
1. p <-- T
2. WHILE p != NIL AND p.left != NIL
3.      p <-- p.left
4. return p
```

The other subroutine is RIGHT-GO-UP(..) This subroutine takes a tree node as a parameter and it goes up until it finds a direct ancestor who is a left node of its parent - once it found this ancestor, it returns the parent node of this ancestor. This is how it works,

```
RIGHT-GO-UP(T)
1. p <-- T
2. WHILE p != NIL AND p.π != NIL  AND p.π.right == p
3.      p <-- p.π    // this is the first encounter of an ancestor who is a left node to its parent
4. IF p == T.root
5.      RETURN p
6. RETURN p.π
```

Alright, why do we return the parent of the ancestor and not the ancestor itself? Well because the check on line 2 in RIGHT-GO-UP subroutine recognises that it is in a right-sub-tree of a parent node. If the execution is in the right sub-tree then parent of which has already been traversed and its key has been printed. Therefore we go one step further to do it.

The quick and sketchy way to see if the completeness and correctness of a draft solution is to do a dry run - Use the tree given about to find out if the above explanation makes any sense.

Oh, don’t forget to give me (you can find my email address at the top) a shout if you find any bugs! I hate bugs :)