Implementation.

The class implementing in-order traversal maintains two fields: root, which is the root node of the binary tree to be traversed; and stack, which is used to manage the correct ordering of node traversal. The root is provided to the traverser at the time the object is instantiated;

InorderTraversal responds to two messages, following the Generator interface. These are *first*, which returns the lowest item in the sequence, and *next*, which returns the lowest item not yet returned by either *first* or *next* since the most recent *first* call. Additionally, an internal message, *doStack*, is used to manage the ordering of elements.

In-order traversal can be understood as requiring both that any given node's left descendants be returned before the node itself, and that its right descendants be returned after the node itself.

first:

Clears the stack. *This ensures that the sequence gets a "fresh start"*

Calls *doStack* on the root. *The doStack call will ensure that the left-most child is at the top of the stack.*

Calls and returns the value from *next. At this point, the stack is in a good initial state, so first can simply function like next.*

next:

If the stack is empty, returns nil. *This terminates the sequence after all nodes are exhausted. Note that the stack will have been populated by first and/or other next calls before this happens.*

Otherwise, pops a node from the stack. *Both first & next make sure that, at any given point, an appropriate next return value is on top of the stack.*

Then, calls *doStack* on that node's right child. *This ensures that the node's right descendants get into the stack before any parent of the node below it in the stack gets reached. Note that doStack functions as a no-op if given a nil argument, so it is no problem if the node has no right child.*

Finally, returns the node.

doStack:

Does nothing if the argument is nil. *This serves both to end doStack's recursive step, and to ignore any attempts to queue non-existent left/right children.*

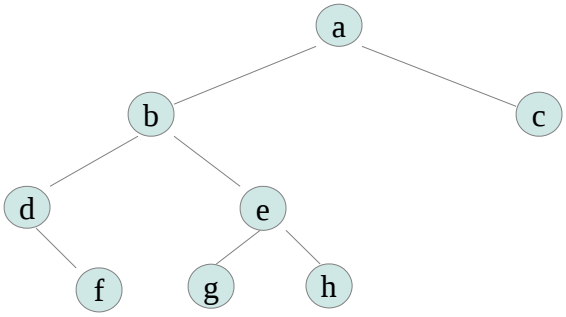
Otherwise, pushes the node on to the stack. *This ensures that the node will get popped & returned at some later point.*

Then, issues another *doStack* on the node's left child. *This will place any left children higher on the stack, ensuring that they get popped first.*

Pictorial Explanation.

In practical terms, the success of this implementation is dependent on the ability to view any given node as a binary tree in and of itself. The *doStack* method can then be defined with the goal of leaving the generator in a state where subsequent *next* calls will traverse a specific sub-tree in a specific order. This permits the simple base case – where a node has no left child and is the next item to visit via in-order traversal – to be implemented, and for *doStack* to then be used recursively to handle the other cases (preparing left children for visitation beforehand; preparing right children for visitation afterward.)

Consider the following tree:

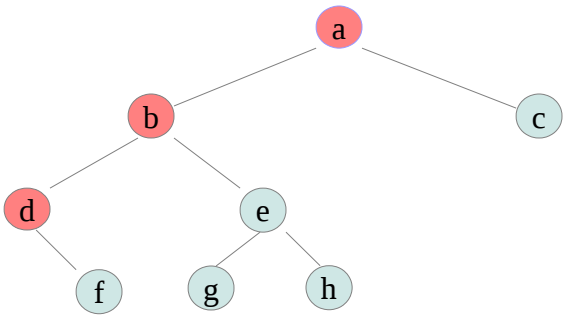


An initial *first* call will empty the stack, and then invoke an initial *doStack* on the root, node a. This *doStack* will push nodes on the stack and then issue recursive calls upon left-hand children as long as they are available – so, until *d* has been encountered and pushed. Walking through a *first* call, this appears as:

***first* invocation**	**Stack** *(bottom … top)*
Clear the stack	
… *doStack* pushes root...	a
...which calls *doStack* on the root's left child, pushing it...	a b
…which calls *doStack* on that node's left child, pushing it...	a b d
...which calls *doStack* on that node's left child, which is nil, causing the chain of calls to return to *first*.	a b d
Then, *next* is called, which pops the top item from the stack	a b

...and calls *doStack* on its right child...	a b f
...which calls *doStack* on that node's left child, which is nil, causing the chain of calls to return to *next*. At this point *next* (as well as *first* from which it was called) can return the initial node, *d*, that was popped previously. Additionally, the stack is left with the appropriate node for the subsequent *next* call at the top.	a b f

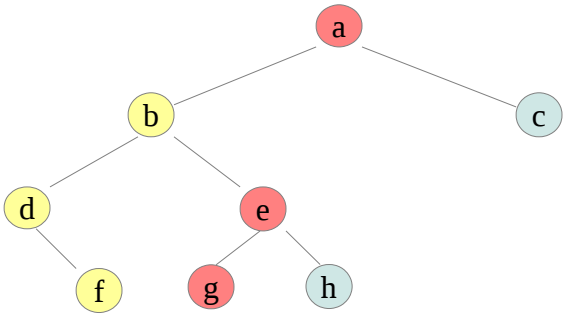
One important note is that following the call to *doStack* on the root, all nodes going leftward from the root are on the stack in an appropriate order (the nodes which should be returned sooner are closer to the top). Since items returned by the generator must come off the top of the stack, this indicates that left-hand nodes are handled earliest, as expected. Below, items after *doStack a* are colored red:



a b d

At the conclusion of the initial *first* call, it can be seen that the appropriate next node for in-order traversal, *f*, is then placed at the top (and *first* returns *d*). A subsequent *next* call will pop *f*, call *doStack* on *f*'s right-hand child (which is nil, resulted in no change), and pop *f*.

The behavior becomes clearer during the following *next* call. This pops *b*, calls *doStack* upon *e*, and returns *b*. As explained above, the *doStack* call upon *e* will add left children to the stack recursively. This leaves the tree as follows (returned nodes in yellow):



a e g

This case demonstrates that right-hand nodes are handled correctly (after their immediate parents, but before their grandparents). Since *e* is treated no differently than the root (or any other node) by the *doStack* method, it is fair to assume that in-order traversal will apply at the sub-tree level as well as at the full-tree level.

A more rigorous argument for the correctness of this assertion follows.

Correctness Criteria.

In-order traversal shall be defined as an iteration over nodes which satisfies the following criteria:

1) **Completeness:** Each node is returned exactly once by the generator.
2) **Order:** The sequence of return values from the generator shall obey the total ordering:

$$a = b_{leftKid} \Rightarrow a < b$$
$$b = a_{rightKid} \Rightarrow a < b$$
$$a_{parent} < b \Rightarrow a < b$$
$$b_{parent} > a \Rightarrow a < b$$

That is, a node's left-hand child and all of its descendants shall be considered less than that node, and a node's right-hand child and all of its descendants shall be considered greater than that node.

Observations.

By convention, the generator has no more elements when *first* or *next* returns nil. This occurs when and only when the underlying stack is observed to be empty during a *next* call. As such, the end of generation occurs only when the stack has become empty.

The returned value of *next*, unless nil, is the value at the top of the stack at the time *next* was invoked.

All nodes enter the stack during the *doStack* method. This method will invoke itself recursively upon every node's left child after pushing that node, unless it is invoked upon nil.

Nodes leave the stack only when either *first* is called (and the sequence is intentionally restarted; this case will be disregarded in subsequent arguments), and when they are popped during the *next* call.

Any node popped during a *next* call is the return value of the same *next* call.

Any node given as an argument to *doStack* is placed on the stack.

Demonstration of Completeness.

Completeness may be shown by examining possible cases. If a node *a* exists within the binary tree, it is either the root, or the left child of another node in the tree, or the right child of another node in the tree. For completeness to be true, each *a* must be returned by *next* (including the *next* called by *first)* exactly once. Equivalently, for completeness to fail, some *a* must exist that is returned either zero times, or more than one time.

The root must be returned before the end of the sequence, as defined by a *next* call which returns nil. This only occurs when the stack is empty. The root, if non-null, is placed on the stack during the initial *first* call (by way of *doStack root*); if the stack is empty, this implies that the root has been removed from the stack, which in turn only occurs if it is popped during a *next* call, in which case it must have been the return value for the same call. This shows that, when the stack is empty at the start of a *next* call, the root has been returned at least once.

If the root is returned more than once, this implies that it has occupied the top of the stack for more than one *next* call. However, the first time it is returned, it has already been removed. It is also only given as an argument to *doStack* in the original *first* call: All other *doStack* calls use either the left or the right child of a node in their argument, and the root is by definition no node's child. As *doStack* is the only place in which nodes are added to the stack, the root can not have been placed on the stack more than once.

If *a* is the left child of some node *b* in the binary tree, then it can be shown that if *b* is returned exactly once, then *a* is also returned exactly once.

To show that *a* is returned at least once, consider that *b*, by virtue of being returned in the sequence, has at some point occupied the stack. The only way *b* may have entered the stack is by way of the *doStack* call; this call also recursively invokes *doStack* on *b's* left child *a*, which therefore must also have entered the stack. Therefore, by the end of the sequence, *a* must have been returned at least once as well (otherwise the stack could not have become empty.)

Likewise, *a* cannot have been returned more than once. The only place any left children are considered is during the recursive step of the *doStack* method, in which they are considered exactly once. To have been added more than once, *doStack* must have been called more than once on *a*'s parent; however, *a*'s parent is *b*, which is returned only once, so this would imply a contradiction.

Similarly, if *a* is the right child of some node *b* in the binary tree, then it can be shown that if *b* is returned exactly once, then *a* is also returned exactly once.

To show that *a* is returned at least once, consider that *b*, by virtue of having been returned, must have at some point occupied the top of the stack during a *next* call. Prior to being returned by *next*, any such node is examined and its right child, which would be *a*, is given as the argument to *doStack*, which in turn would place *a* on the stack. As such, if *b* is returned then at some point *a* occupies some position on the stack, and so for the stack to become empty *a* must have at some point been returned.

Finally, *a* cannot have been returned more than once. The right child of a node is only considered in one method, *next*, and is only considered once per invocation. For *a* to have been submitted as the argument to *doStack* more than once, then more than one *next* invocation must have found (and also returned) *a*'s parent at the top of the stack. However, *a's* parent is *b*, which is returned only once, so this would imply a contradiction.

In summary: Completeness applies for the root; for any left child of a node for which completeness applies; and for any right child of a node for which completeness applies. So, completeness applies to the root's children, their children, and so on, including by induction the entire tree.

Demonstration of Order.

The returned node of *first*, if non-nil, is less than the returned nodes of all subsequent *next* calls.

Consider *a* to be the *first* return value and *b* to be the return value of any subsequent *next* call.

Note that if $b < a$, this implies either:

1) *b* is the left child of *a*, or a descendant of the left child of *a*.
2) *a* is the right child of *b*, or a descendant of the right child of *b*.

However, (1) cannot be true; it is known that the top-most item of the stack had no left child. Per the definition of a stack, the top is the most recent item pushed which has not yet been returned. As no items have been returned since the stack was cleared at the start of *first*, this is simply the last node pushed. If *a* had any left child, this would have been pushed after *a* during the *doStack* invocation, and *a* could not have been the return value of *first* (a contradiction).

Likewise, (2) cannot be true; no right children can have been added by the time *a* was popped. The stack is cleared at the start of *first*, and the return value of *first* is given as the result of one *next* call, which (as above) is the value at the top of the stack at the time *next* is invoked. However, the only reference to a node's right-hand child is during the *next* call: Since this has not occurred at the start of that *next* call, there is no way for a right-hand child (or its descendants) to have been considered at the time the return value is determined.

Additionally, the possibility that $a = b$ can be ruled out, as this would violate completeness, which has been demonstrated.

This leaves only that $a < b$ for all *b* returned by next subsequent to *a* returned by first. This is consistent with the ordering requirement.

Next, it can be shown that the full sequence of *next* calls satisfies the ordering criteria. Because the total order is transitive, this can be demonstrated by showing that any two consecutive return values of *next* shall satisfy the ordering. That is, $a < b$ where *a* is one return value of *next* and *b* is the return value of *next* immediately following. (Note that *a* and *b* are presumed to be non-nil; if either is nil, the generator is complete and ordering is no longer an issue.)

As before, $b < a$ implies either:

1) *b* is the left child of *a*, or a descendant of the left child of *a*.
2) *a* is the right child of *b*, or a descendant of the right child of *b*.

The only child of *a* that is placed on the stack after *a* is popped is its right child. Therefore, any nodes that are popped from the stack after *a* are either its right descendants, or nodes below it on the stack. None of the possible *b* nodes in (1) are right descendants, so they can only occur subsequent to *a* if they are below it on the stack. However, *a*'s left children or only considered for placement on the stack immediately after *a* is pushed, and therefore must be placed above *a* on the stack and be popped first. Therefore, (1) cannot occur.

In the case of (2), *b*'s right children (and therefore any of its ancestors) are only considered for placement on the stack after *b* has been popped. The node *b* is only popped when it occupies the top of the stack at the start of a *next* call, for which it will ultimately be returned. As such, its right ancestors only occupy the stack (and therefore, the top of the stack, from which they might be returned) at the start of *next* calls which occur subsequent to *b*'s return. As such, no *a* may exist which satisfies (2) above.

This rules out $b < a$; the case where $b = a$ is also ruled out by the uniqueness demonstrated by the completeness property. As such, $a < b$ for any consecutive nodes *a* and *b* returned by next, which in turn implies that ordering is satisfied by the sequence of *next* calls.

In summary, the return value of *first* is correctly ordered (no lesser node exists in the binary tree), and all *next* calls also exhibit correct ordering, meaning that all methods of the generator are ordered correctly.