CSC 868.01 – Spring 2013
Advanced Object Oriented Software Design and Development
Prof. Levine

SmallTalk Lab Documentation

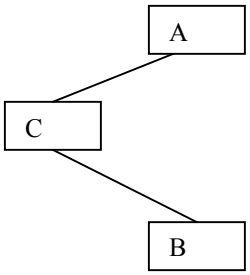
Group 1
March 5, 2013

Group members
Robert (Bob) Bierman
Victor Woeltjen
Jason Lum
Steven Gimeno
Ying Kit Ng (Kent)
Bianca Uy
Kay Choi

Project Description: Squeak Smalltalk Exercise

Define a binary tree class and construct a tree internally via Smalltalk statements, e.g.
node1 _ BinTree new: 'A' "build a new tree with node labeled A".
node2 _ BinTree new: 'C'.
node3 _ BinTree new: 'B'.
node1 addLeftKid: node2.
node2 addRightKid node3.

At this point, the tree looks like:



Perform traversals on the tree; print the node labels as they are encountered
Inorder traversal (left root right) C B A
Preorder traversal (root, left, right) A C B
Postorder traversal (left, right, root) B C A

Note: view a tree as a ***collection*** (ordered) of nodes; use *first*, *next* to yield nodes in indicated order; you should have a *BinTree* class and traversal classes with constructors - e.g. you would like to do something like:

suppose *theTree* is the variable referencing the tree just constructed;
traversal _ InorderTraversal new: theTree.
"the statement above performs initialization; it DOES NOT create a new collection"
nextLabel _ traversal first.
(nextLabel notNil) whileTrue: [nextLabel print. nextLabel _ traversal next]
...

DO NOT create a collection, besides the original BinTree; *simply keep track where you are traversing in the tree* (for each traversal type) so you can determine which node to yield for the next request of first/next. DO NOT include a link from each node to its parent. **For this part of the exercise you are only required to provide the code for the *inorder* case, not the *pre/post-order* cases.**

Notes:
Since you will be simulating recursion you will need to maintain an explicit stack to be used for continuing each time the user requests a *next* call. The stack will maintain a sequence of nodes leading back to the root from the current node being scrutinized. DO NOT traverse the tree more than once when implementing *first/next*.

In summary, you will need to

- build trees as described above – use my class names and other id's where provided; I might run your code so my test harness will expect to call classes/methods that I have named above (*InorderTraversal, first, next, BinTree, addLeftKid:, addRightKid:)* .
- provide in-/pre-/post-order traversal code without stopping at each step
- provide inorder traversal using first/next

Turn in:

1. Test cases
2. Class diagrams
3. Documentation on your traversal algorithms
4. Smalltalk code – use Squeak Smalltalk

A note on documentation: since you will develop your own inorder traversal algorithm to be used with *first/next*, it is essential to provide good algorithm documentation so I can easily understand the methodology you used. To this end, it's best to use lots of pictures, especially pictures depicting the various stages of the traversal process in a step-by-step fashion. It's ok to provide more detail than necessary but it's not ok to leave me guessing on the details of the algorithm if you do not use visual aids to assist in the description. For instance, it would be best to provide a step-by-step description of your algorithm using a sample tree to help me understand precisely how the traversal is performed and what information/data structures are used, as well as how the data structures are used.

Binary Tree Collection

This project created a new category called “Collections-BinTree”, the collection contains the node class called BinTree, three traversal classes, and a test class.

The **BinTree** class stores the minimal amount of information, which includes the nodes object along with a right and left node which are initialized to nil.

BinTree has 4 categories:

Adding

With the methods “addLeftKid:” and “addRightKid:” which both take a BinTree object and set the left or right child.

Accessing

Contains the three methods, “left”, “right”, and “obj” which just return the values of those instance variables.

Initialize

Contains two methods, one instance, “initNew:” which does the actual variable initialization and the class method “new:” which allows the creation of the node and assignment to a variable and set the nodes object.

Print

Contains “print”, and “visit”. The “visit” method is the method called by the recursive traversal methods and is a simple print of the nodes object. The class could be sub-classed and visit replaced for other purposes.

Usage of the BinTree class is very simple. A new node is created with a call to new: with the object desired to be stored in the node.

Example: myNode := BinTree new: “Text Object”

After you create a second and further nodes, you can link them into a tree by assigning one node to the left or right child links of another node.

Example: To connect Node2 to the left child of Node1 do the following

Node1 addLeftKid: Node2

To get a value out of the node, such as the left child, use the access method:

Example: leftnode := Node1 left

The three traversal classes are "**InorderTraversal**", "**PreorderTraversal**", and "**PostorderTraversal**". All three classes have a common structure, with only the variance in algorithm and traversal order. There are two instance variables, one for the node of the tree, the other for a stack for iterator traversal. All three traversals support both a recursive traversal, and a first/next iterator traversal.

The traversal classes are broken down into the following categories and associated methods:

Initialize-release and class initialization

These contain the method "new:" and "init:" to create the class and initialize the instance variables.

Private

These are internal methods, "clearStack" and "doStack:", used to maintain the internal stack for the iterator methods. PreorderTraversal does not use "doStack:" .

Traverse

This contains the recursive method "recurse:" which visits every node in the tree in the one call and calls the "visit" method for each node.

Iterators

The two methods "first" and "next" which return the first node of the tree traversal and each subsequent node until nil when complete.

Tracing

This just contains a method "getStack" to return the internal stack for tracing and display in the test cases to show what is on the stack each step through the iterators.

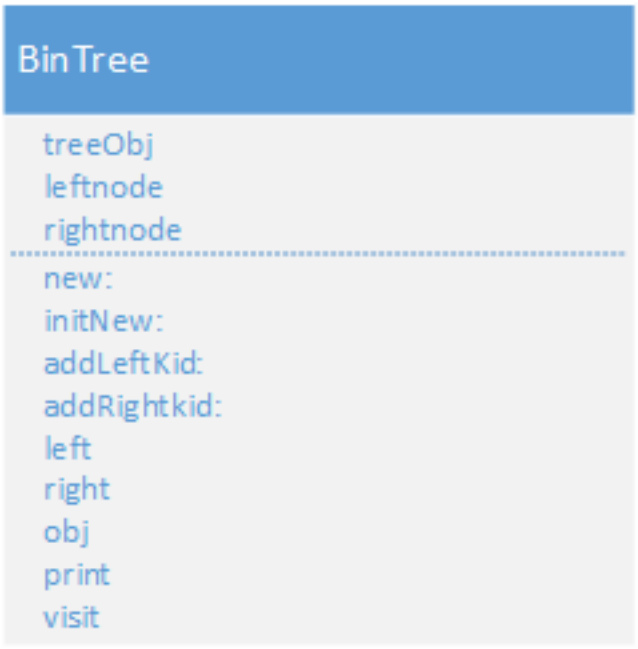
Usage of the traversal routines are broken into two specific cases, recursive and iterator. For recursive, there is no need to instantiate the traversal class. The method takes a root node as a parameter and calls the node's "visit" method.

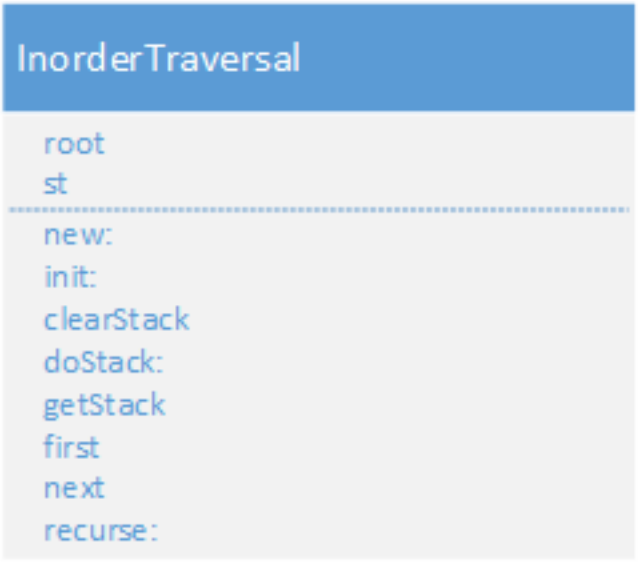
Example: PreorderTraversal recurse: rootNode

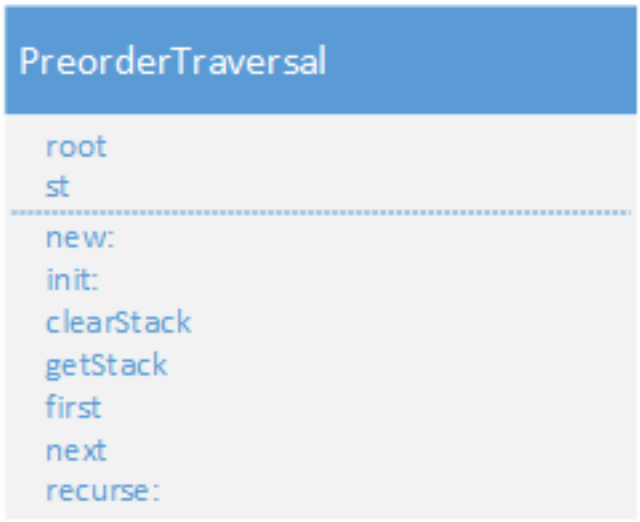
For the iterator usage, the class needs to be instantiated with the root node, then a call to "first" will start the traversal and return the first node. Any call to "first" will reset the traversal. Repetitive calls to "next" will return subsequent nodes.

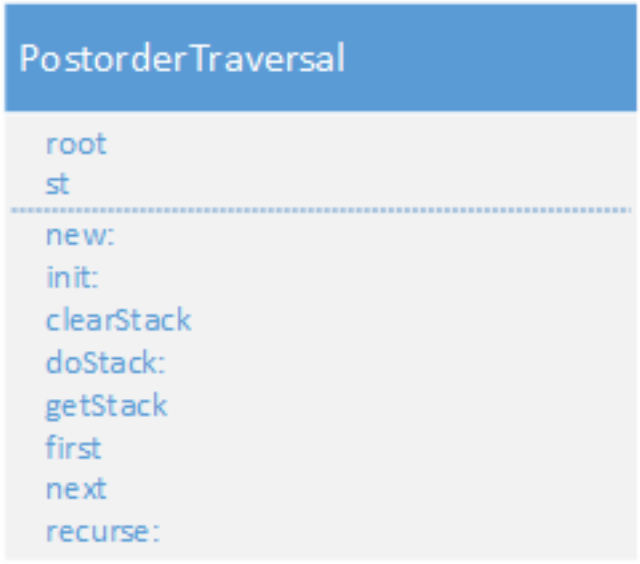
```
Example:    traversal := InorderTraversal new: node1
            nextLabel := traversal first
            (nextLabel notNil) whileTrue:
                    [action on node. nextLabel := traversal next]
```

Class Diagrams






PreorderTraversal
root
st
new:
init:
clearStack
getStack
first
next
recurse:


PostorderTraversal
root
st
new:
init:
clearStack
doStack:
getStack
first
next
recurse:


TraversalTest
largeTest
smallTest

Testing

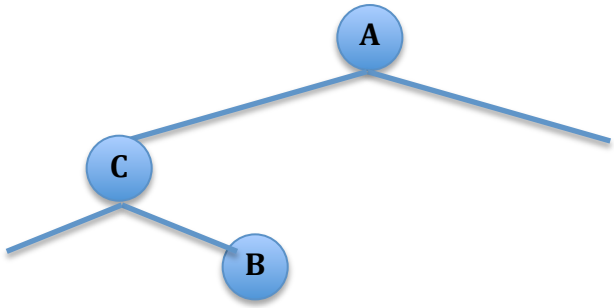
The TraversalTest class contains a small and large test method. The small method is the three nodes as specified in the project description. The large test is a 17-node test.

The tests create a tree and use the root node of the tree for all three forms of the traversals, and perform both an iterator and a recursive traversal for each form.

While traversing the tree using the iterator approach, the stack trace is also done showing the contents of the stack after each call. The "Current Node" shows the actual node being referenced for that iteration. The iterator and recursive methods show the same results.

Small Test

This is a three node test with the tree described below



Test case: a binary tree with three nodes.
A is the root node, C is left of A, B is right of C.

In-order:
Current node: C
Current stack: 'A''B'
Current node: B
Current stack: 'A'
Current node: A
Current stack:

Pre-order.
Current node: A
Current stack: 'C'
Current node: C

Current stack: 'B'
Current node: B
Current stack:

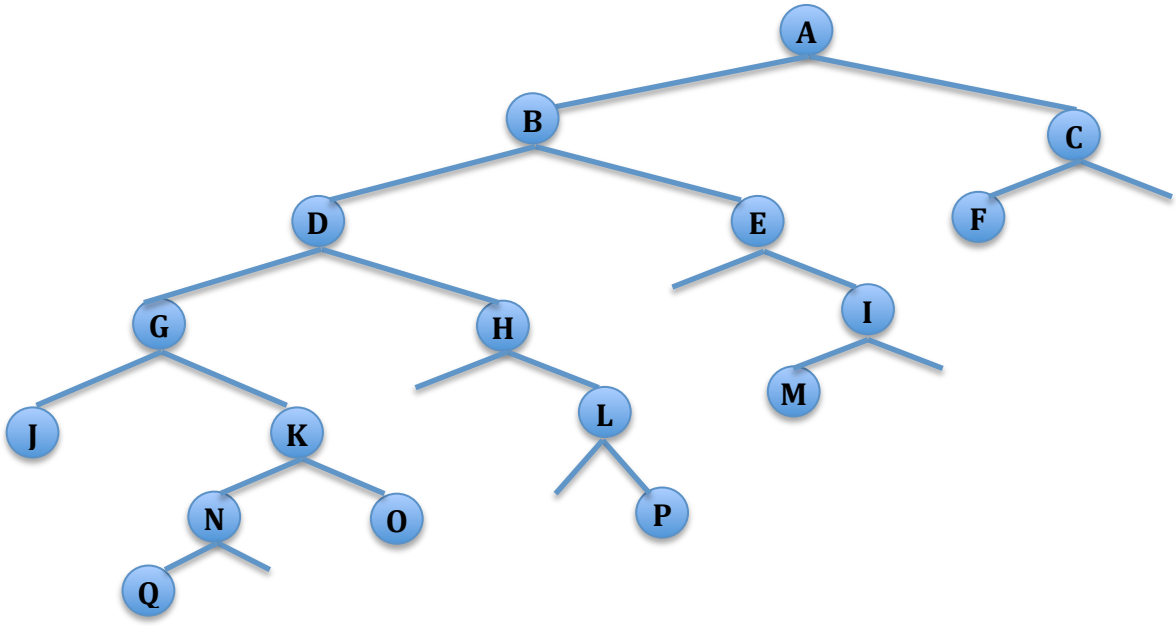
Post-order.
Current node: B
Current stack: 'A''C'
Current node: C
Current stack: 'A'
Current node: A
Current stack:

Recursive InOrder Traversal.
C - B - A -

Recursive PreOrder Traversal.
A - C - B -

Recursive PostOrder Traversal.
B - C - A -

Larger Test



Test case: a binary tree with seventeen nodes.

In-order:
Current node: J
Current stack: 'A''B''D''G'
Current node: G
Current stack: 'A''B''D''K''N''Q'
Current node: Q
Current stack: 'A''B''D''K''N'
Current node: N
Current stack: 'A''B''D''K'
Current node: K
Current stack: 'A''B''D''O'
Current node: O
Current stack: 'A''B''D'
Current node: D
Current stack: 'A''B''H'
Current node: H
Current stack: 'A''B''L'
Current node: L
Current stack: 'A''B''P'
Current node: P
Current stack: 'A''B'
Current node: B
Current stack: 'A''E'
Current node: E
Current stack: 'A''I''M'
Current node: M
Current stack: 'A''I'
Current node: I
Current stack: 'A'
Current node: A
Current stack: 'C''F'
Current node: F
Current stack: 'C'
Current node: C
Current stack:

Pre-order.
Current node: A
Current stack: 'C''B'
Current node: B
Current stack: 'C''E''D'
Current node: D

Current stack: 'C''E''H''G'
Current node: G
Current stack: 'C''E''H''K''J'
Current node: J
Current stack: 'C''E''H''K'
Current node: K
Current stack: 'C''E''H''O''N'
Current node: N
Current stack: 'C''E''H''O''Q'
Current node: Q
Current stack: 'C''E''H''O'
Current node: O
Current stack: 'C''E''H'
Current node: H
Current stack: 'C''E''L'
Current node: L
Current stack: 'C''E''P'
Current node: P
Current stack: 'C''E'
Current node: E
Current stack: 'C''I'
Current node: I
Current stack: 'C''M'
Current node: M
Current stack: 'C'
Current node: C
Current stack: 'F'
Current node: F
Current stack:

Post-order.
Current node: J
Current stack: 'A''B''D''G''K''N''Q'
Current node: Q
Current stack: 'A''B''D''G''K''N'
Current node: N
Current stack: 'A''B''D''G''K''O'
Current node: O
Current stack: 'A''B''D''G''K'
Current node: K
Current stack: 'A''B''D''G'
Current node: G
Current stack: 'A''B''D''H''L''P'
Current node: P
Current stack: 'A''B''D''H''L'

Current node: L
Current stack: 'A''B''D''H'
Current node: H
Current stack: 'A''B''D'
Current node: D
Current stack: 'A''B''E''I''M'
Current node: M
Current stack: 'A''B''E''I'
Current node: I
Current stack: 'A''B''E'
Current node: E
Current stack: 'A''B'
Current node: B
Current stack: 'A''C''F'
Current node: F
Current stack: 'A''C'
Current node: C
Current stack: 'A'
Current node: A
Current stack:

Recursive InOrder Traversal.
J - G - Q - N - K - O - D - H - L - P - B - E - M - I - A - F - C -

Recursive PreOrder Traversal.
A - B - D - G - J - K - N - Q - O - H - L - P - E - I - M - C - F -

Recursive PostOrder Traversal.
J - Q - N - O - K - G - P - L - H - D - M - I - E - B - F - C - A -

Algorithms

The algorithm documentation is divided into two parts. The first is a general description followed by a full in-depth analysis of the In-order iterator traversal. Starting with the recursive versions, all three follow the same pattern:

```
Preorder: Takes a tree T (root node)
        While (T != null)
                Visit T
                Preorder (T.left)
                Preorder (T.right)

Inorder: Takes a tree T (root node)
        While (T != null)
                Preorder (T.left)
                Visit T
                Preorder (T.right)

Postorder: Takes a tree T (root node)
        While (T != null)
                Preorder (T.left)
                Preorder (T.right)
                Visit T
```

Since T can be null, when it is, nothing is there to display and the algorithms take that into account therefore they handle the case of T equal to NULL.
Any Node within tree T is a sub tree, if the algorithms work for the sub-tree by induction it will work with the entire tree. Letting T not be empty and is the root node, preorder is correct because that node is visited; by induction when the left tree is visited it is also correct as a sub-tree and is done for all levels. Also by induction the same is true as it visits the right node, therefore this is correct for all trees in Preorder. By induction this is also correct for Inorder and Post order based on traversing down the left chain until a node with no left node is reached, then in Inorder the node is visited before looking right, and in Postorder right is visited until a leaf node where the node is visited.

For the Preorder iterative traversal the root is first pushed on the stack. For each iteration, the node is popped from the stack and the right node (if not null) is pushed on the stack and then the left node (if not null) is pushed on the stack. This meets the requirements of preorder as the root node of any sub-tree is returned as it's right sub-tree then left sub-tree are added to the stack by way of those sub-trees root nodes. Therefore by induction the algorithm is correct.

For Postorder, the algorithm is similar to Inorder except that doStack processes right prior to the visit. In-depth explanation of Inorder follows.

CSC 868.01 – Spring 2013
Advanced Object Oriented Software Design and Development
Prof. Levine

SmallTalk: In-order Traversal
Documentation

Victor Woeltjen, Group 1

February 28, 2013

Group members

Robert (Bob) Bierman
Victor Woeltjen
Jason Lum
Steven Gimeno
Ying Kit Ng (Kent)
Bianca Uy
Kay Choi

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. When *doStack* is called, it prepares the stack with an appropriate initial state for traversing the tree or sub-tree whose root is its argument.

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.*

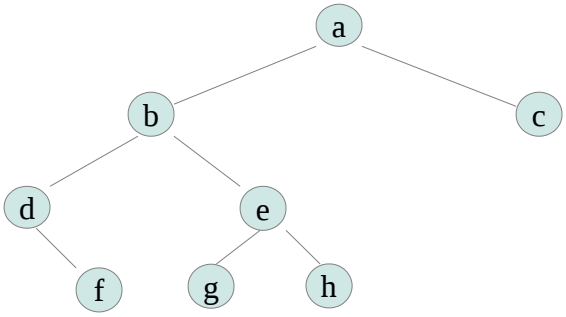
Otherwise, pushes the node on to the stack. *This ensures that the node will get popped & returned at some later point.*

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.*

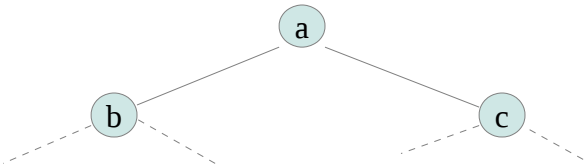
Pictorial Explanation.

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 the correct 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; and, when called by *next*, preparing right children for visitation afterward.)

Consider the following tree:



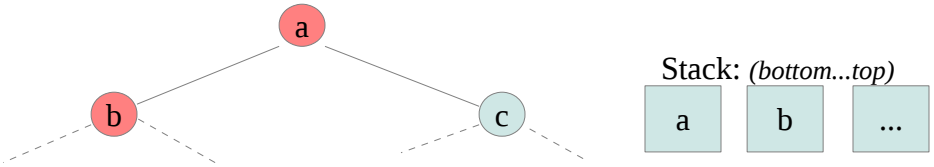
This may be considered as a tree *a* with two sub-trees *b* and *c*:



In this simpler form, in-order traversal may be understood more simply as handling tree *b* before node *a*, and then handling tree *c*.

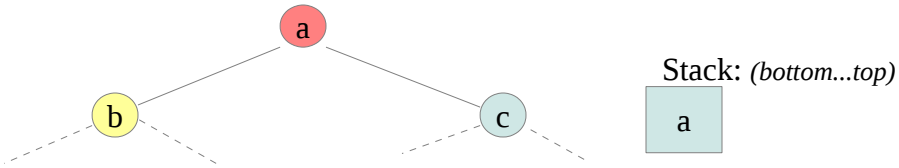
The problem of handling sub-tree *b* first shall be considered first. Since nodes are handled top down on the stack, this is achieved by pushing tree *b* onto the stack after node *a*. This appears in the *doStack* call for *a:* The node *a* is pushed explicitly to the stack, and then the subtree *b* is pushed by an explicit *doStack* call.

At this point, the stack appears as (note that b's children are assumed to be stacked appropriately; **nodes on the stack** after *doStack a* are colored red):

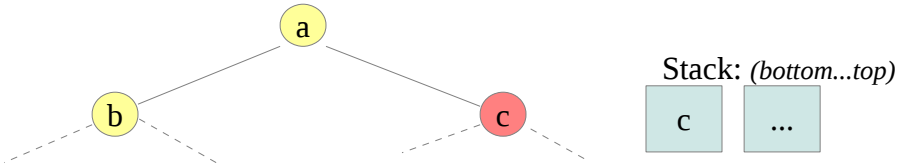


Subsequent *next* calls will pop and return nodes involved in sub-tree *b*, until that sub-tree is exhausted. (We assume these are in the correct order; this assumption is verified once *a* and its sub-trees are shown to be handled in the correct order, as each sub-tree is stacked and returned in the same manner as *a*.)

At this point, sub-tree *a* is on top of the stack (**nodes that have been returned** are in yellow):



In this state, a *next* call will encounter and return *a*. Before returning, it will also invoke the *doStack* method on sub-tree *c*. This will add any necessary nodes from sub-tree *c* to the stack, with *c* at the bottom, in the same manner as *a*.

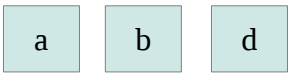
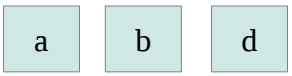
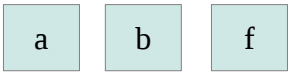
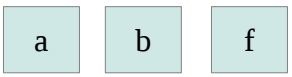


From here, the sub-tree *c* can be traversed by subsequent *next* calls. Note again that the same *doStack* invocation was used for sub-tree *c* that was used for tree *a*, so its own sub-trees will also be returned in an appropriate order. The stack is therefore only permitted to become empty – which is the state that *next* checks for when deciding to indicate that the generator has concluded – after its last node has been found to have no right-hand children.

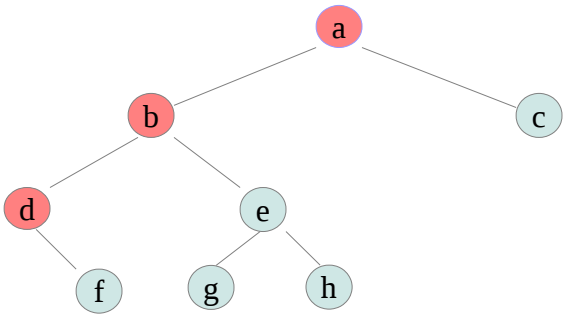
A more detailed view of the behavior of the stack follows.

The following table illustrates the behavior of the stack for one *first* and one subsequent *next* call.

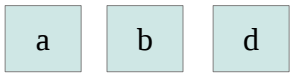
The initial *first* call will begin by emptying the stack, and then invoke an initial *doStack* on the root, node a. This *doStack* will push nodes on the stack and then issue recursive calls upon left-hand children as long as they are available – so, until *d* has been encountered and pushed. Walking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

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

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. 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, **nodes on the stack** after *doStack a* are colored red:

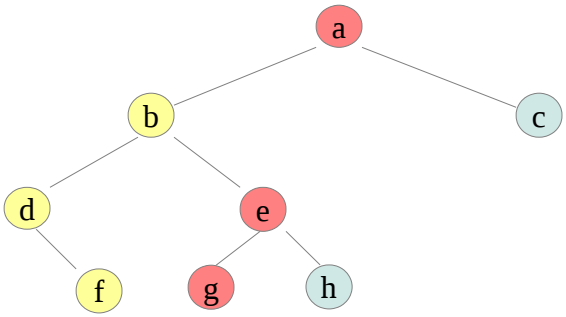


Stack: *(bottom...top)*

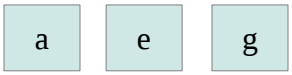


At the conclusion of the initial *first* call, it can be seen that the appropriate next node for in-order traversal, *f*, is then placed at the top (and *first* returns *d*). A subsequent *next* call will pop *f*, call *doStack* on *f*'s right-hand child (which is nil, resulting in no change), and pop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



Stack: *(bottom...top)*



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.

It is taken as an assumption that this ordering is transitive; that is, that $a < b$ and $b < c$ implies that $a < c$.

Observations.

The following observations may be made based on a superficial inspection of the data structure as implemented:

(1) 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.

(2) The returned value of *next* is the value at the top of the stack at the time *next* was invoked, except in the case where the stack is empty at that time.

(3) The only way a node enters the stack is through the *doStack* method.

(4) Any and every node given as an argument to *doStack* is pushed on the stack. (Note that nil is not a node.)

(5) The *doStack* method will invoke itself recursively upon every node's left child after pushing that node, unless (and until) it is invoked upon nil.

(6) Nodes only leave the stack when they are popped by *next*. (Except in the case where generation is intentionally restarted by a *first* call; this will clear the stack as its first action.)

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

Demonstration of Completeness.

Completeness may be shown by examining all ways in which a node may be included in a tree. 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.

Completeness shall therefore be shown that each of the three types of node (roots, left children, and right children) cannot be returned zero times, and cannot be returned 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 (1). 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 (6), in which case it must have been the return value for the same call (7). 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 (2). However, the first time it is returned, it has already been removed (7). 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 first is only called once, and *doStack* is the only place in which nodes are added to the stack (3), the root can not have been placed on the stack more than once.

In the case of children, completeness shall be considered as an inheritable property. 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 (2). The only way *b* may have entered the stack is by way of the *doStack* call (3); this call also recursively invokes *doStack* on *b's* left child *a*, (5) which therefore must also have entered the stack. So, by the end of the sequence, *a* must have been returned at least once as well – otherwise the stack could not have become empty. (1)

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;* if *doStack b* had been invoked more than once, *b* would have been pushed to the stack more than once, and to have left the stack more then once it would have been returned more than once (6,7). However, *b* is known to have been returned only once by way of assumption, so this would imply a contradiction. Therefore, *a* cannot have been returned more than once.

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 (2). 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. (4) 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 popped (6) and returned. (7)

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 (7) – *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 Ordering.

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, option #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, (5) and *a* could not have been the return value of *first* (a contradiction).

Likewise, option #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. (2) 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*.

In the case of option #1, The only immediate child of *a* that is placed on the stack after *a* is popped – which occurs during the same *next* call in which it is returned (7) – is its right child. Therefore, any nodes that are popped from the stack after *a* are either its right descendants, or nodes that were already below it on the stack. However, the tree to *a*'s left is only considered after *a* has been pushed, in the recursive step of *doStack*, (5) so no such descendants can exist below it. As such, *b* cannot be a left descendant of *a*.

In the case of option #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, (2) for which it will ultimately be returned. (7)

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 option #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.

SmallTalk Source Code

Collections-BinTree

```
Object subclass: #BinTree
	instanceVariableNames: 'treeObj leftnode rightnode'
	classVariableNames: ''
	poolDictionaries: ''
	category: 'Collections-BinTree'!
!BinTree commentStamp: 'RAB 2/21/2013 11:41' prior: 0!
Tree nodes just store an object and a left/right link to other tree
nodes!


!BinTree methodsFor: 'adding' stamp: 'RAB 2/23/2013 12:07'!
addLeftKid: node
	"Set the left child for this node, must be a BinTree object"

	node class == self class ifFalse: [ ^false ].
	leftnode_node! !

!BinTree methodsFor: 'adding' stamp: 'RAB 2/23/2013 12:07'!
addRightKid: node
	"Set the right child for this node, must be a BinTree object"

	node class == self class ifFalse: [ ^false ].
	rightnode_node! !


!BinTree methodsFor: 'accessing' stamp: 'RAB 2/23/2013 12:09'!
left
	"return node.left"

	^ leftnode! !

!BinTree methodsFor: 'accessing' stamp: 'RAB 2/23/2013 12:09'!
obj
	"return node.object"

	^ treeObj! !

!BinTree methodsFor: 'accessing' stamp: 'RAB 2/23/2013 12:09'!
right
	"return node.right"

	^ rightnode! !


!BinTree methodsFor: 'Initialize' stamp: 'RAB 2/23/2013 12:08'!
initNew: objectItem
	"This is called by the class new:, it sets the node's object
and initialized left and right to nil"

	treeObj_objectItem.
```

```
        leftnode_nil.
        rightnode_nil.
        ^self
        ! !


!BinTree methodsFor: 'print' stamp: 'RAB 2/23/2013 12:11'!
print
        "comment stating purpose of message"


        ^treeObj printString! !

!BinTree methodsFor: 'print' stamp: 'RAB 3/4/2013 18:23'!
visit
        "routine called from recursive traversals"


        Transcript show: (self obj); show: '  -  '. ! !

"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

BinTree class
        instanceVariableNames: ''!

!BinTree class methodsFor: 'class initialization' stamp: 'RAB
2/23/2013 12:13'!
new: objectItem
        "this creates an instance of the BinTree object (node) and
initializes it with the object passed in"

        ^(self new)
        initNew: objectItem
! !



Object subclass: #InorderTraversal
        instanceVariableNames: 'st root'
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Collections-BinTree'!

!InorderTraversal methodsFor: 'initialize-release' stamp: 'RAB
2/23/2013 13:05'!
init: theTree
        "called by new: to set the root"


        root := theTree.
```

```
        st := Stack new.
        ^self! !


!InorderTraversal methodsFor: 'Iterators' stamp: 'RAB 3/4/2013 21:20'!
first
        "Reset the traversal and return the first node"

        "Clear Stack"
        self clearStack.

        "Initialize stack from root"
        self doStack: root.

        "return the next node, in this case it is node number 1"
        ^self next

! !

!InorderTraversal methodsFor: 'Iterators' stamp: 'RAB 2/23/2013
14:10'!
next
        "return the next node in the traversal"

        | node |
        st isEmpty ifTrue: [node := nil.] ifFalse: [node := st pop.
self doStack: (node right).].
        ^ node

        ! !


!InorderTraversal methodsFor: 'tracing' stamp: 'RAB 2/27/2013 17:21'!
getStack
        "return the stack used internally for tracing purposes"

        ^st! !


!InorderTraversal methodsFor: 'private' stamp: 'RAB 2/23/2013 13:23'!
clearStack
        "comment stating purpose of message"

        st isEmpty ifFalse: [st pop. self clearStack]! !

!InorderTraversal methodsFor: 'private' stamp: 'RAB 2/23/2013 13:28'!
doStack: node
        "comment stating purpose of message"

        node == nil ifFalse: [st push: node. self doStack: (node
```

```
left)]! !

"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

InorderTraversal class
        instanceVariableNames: ''!

!InorderTraversal class methodsFor: 'class initialization' stamp: 'RAB
3/4/2013 21:19'!
new: theTree
        "instantiate a new InorderTraversal speficify the root node"

        ^(self new)
        init: theTree
        ! !


!InorderTraversal class methodsFor: 'traverse' stamp: 'RAB 3/4/2013
17:02'!
recurse: node
        "Show the entire tree using InorderTraversal"


        node == nil ifFalse: [
                self recurse: (node left).
                node visit.
                self recurse: (node right).].

        ! !



Object subclass: #PostorderTraversal
        instanceVariableNames: 'root st'
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Collections-BinTree'!

!PostorderTraversal methodsFor: 'private' stamp: 'RAB 2/23/2013
15:32'!
clearStack
        "comment stating purpose of message"

        st isEmpty ifFalse: [st pop. self clearStack]! !

!PostorderTraversal methodsFor: 'private' stamp: 'RAB 2/23/2013
16:45'!
doStack: node
        "comment stating purpose of message"
```

```
        node == nil ifFalse: [st push: node. self doStack: (node
left). (node left) == nil ifTrue: [self doStack: (node right)].]! !


!PostorderTraversal methodsFor: 'iterators' stamp: 'RAB 2/23/2013
16:10'!
first
        "Initialize and return first node of traversal"

        "Clear Stack"
        self clearStack.

        "Initialize stack from root"
        self doStack: root.

        "return the next node, in this case it is node number 1"
        ^self next

! !

!PostorderTraversal methodsFor: 'iterators' stamp: 'RAB 2/23/2013
17:01'!
next
        "return the next node in the traversal"

        | node |
        st isEmpty ifTrue: [node := nil.] ifFalse: [node := st pop. st
isEmpty ifFalse: [node == ((st top) right) ifFalse: [self doStack:
((st top) right)].].].
        ^node

        ! !


!PostorderTraversal methodsFor: 'initialize-release' stamp: 'RAB
2/23/2013 15:27'!
init: theTree
        "called by new: to set the root"

        root := theTree.
        st := Stack new.
        ^self! !


!PostorderTraversal methodsFor: 'tracing' stamp: 'RAB 2/27/2013
17:21'!
getStack
        "return the stack used internally for tracing purposes"

        ^st! !
```

```
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

PostorderTraversal class
	instanceVariableNames: ''!

!PostorderTraversal class methodsFor: 'class initialization' stamp:
'RAB 3/4/2013 21:20'!
new: theTree
	"instantiate a new PostorderTraversal speficify the root node"

	^(self new)
	init: theTree
	! !


!PostorderTraversal class methodsFor: 'traverse' stamp: 'RAB 3/4/2013
17:02'!
recurse: node
	"Show the entire tree using PostorderTraversal"


	node == nil ifFalse: [
		self recurse: (node left).
		self recurse: (node right).
		node visit
		].

! !



Object subclass: #PreorderTraversal
	instanceVariableNames: 'root st'
	classVariableNames: ''
	poolDictionaries: ''
	category: 'Collections-BinTree'!

!PreorderTraversal methodsFor: 'private' stamp: 'RAB 2/23/2013 14:26'!
clearStack
	"comment stating purpose of message"

	st isEmpty ifFalse: [st pop. self clearStack]! !


!PreorderTraversal methodsFor: 'iterators' stamp: 'RAB 2/23/2013
15:10'!
first
	"Initialize and return first node of traversal"
```

```
        "Clear Stack"
        self clearStack.

        "Initialize stack from root"
        st push: root.

        "return the next node, in this case it is node number 1, the
root"
        ^self next

! !

!PreorderTraversal methodsFor: 'iterators' stamp: 'RAB 2/23/2013
15:17'!
next
        "return the next node in the traversal"

        | node |
        st isEmpty ifTrue: [node := nil.] ifFalse: [node := st pop.
                (node right) == nil ifFalse: [st push: (node right)].
                (node left) == nil ifFalse: [st push: (node left)].].

        ^ node

        ! !


!PreorderTraversal methodsFor: 'tracing' stamp: 'RAB 2/27/2013 17:21'!
getStack
        "return the stack used internally for tracing purposes"

        ^st! !


!PreorderTraversal methodsFor: 'initialize-release' stamp: 'RAB
2/23/2013 14:58'!
init: theTree
        "called by new: to set the root"

        root := theTree.
        st := Stack new.
        ^self! !

"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

PreorderTraversal class
        instanceVariableNames: ''!

!PreorderTraversal class methodsFor: 'traverse' stamp: 'RAB 3/4/2013
17:03'!
```

```
recurse: node
        "Show the entire tree using PreorderTraversal"


        node == nil ifFalse: [
                node visit.
                self recurse: (node left).
                self recurse: (node right).
                ].

! !


!PreorderTraversal class methodsFor: 'class initialization' stamp:
'RAB 3/4/2013 21:20'!
new: theTree
        "instantiate a new PreorderTraversal speficify the root node"

        ^(self new)
        init: theTree
        ! !



Object subclass: #TraversalTest
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Collections-BinTree'!

"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

TraversalTest class
        instanceVariableNames: ''!

!TraversalTest class methodsFor: 'test method' stamp: 'RAB 3/4/2013
18:57'!
largeTest
        "Function for testing the BinTree and traversal classes.
        A severnteen node binary tree is created and traversed."

| nodeA nodeB nodeC nodeD nodeE nodeF nodeG nodeH nodeI nodeJ nodeK
nodeL nodeM nodeN nodeO nodeP nodeQ traverse nextNode travStack
tempStack tempEntry |

Transcript clear.
Transcript show: 'Test case: a binary tree with seventeen nodes.'; cr.
Transcript show: 'See documentation for tree layout.'; cr; cr.
nodeA := BinTree new: 'A'.
nodeB := BinTree new: 'B'.
```

```
nodeC := BinTree new: 'C'.
nodeD := BinTree new: 'D'.
nodeE := BinTree new: 'E'.
nodeF := BinTree new: 'F'.
nodeG := BinTree new: 'G'.
nodeH := BinTree new: 'H'.
nodeI := BinTree new: 'I'.
nodeJ := BinTree new: 'J'.
nodeK := BinTree new: 'K'.
nodeL := BinTree new: 'L'.
nodeM := BinTree new: 'M'.
nodeN := BinTree new: 'N'.
nodeO := BinTree new: 'O'.
nodeP := BinTree new: 'P'.
nodeQ := BinTree new: 'Q'.

nodeA addLeftKid: nodeB.
nodeA addRightKid: nodeC.
nodeB addLeftKid: nodeD.
nodeB addRightKid: nodeE.
nodeC addLeftKid: nodeF.
nodeD addLeftKid: nodeG.
nodeD addRightKid: nodeH.
nodeE addRightKid: nodeI.
nodeI addLeftKid: nodeM.
nodeG addLeftKid: nodeJ.
nodeG addRightKid: nodeK.
nodeH addRightKid: nodeL.
nodeL addRightKid: nodeP.
nodeK addLeftKid: nodeN.
nodeK addRightKid: nodeO.
nodeN addLeftKid: nodeQ.

tempStack := Stack new.

Transcript show: 'In-order:'; cr.
traverse := InorderTraversal new: nodeA.
nextNode := traverse first.
[nextNode notNil]
        whileTrue: [
                Transcript show: 'Current node: '.
                Transcript show: nextNode obj; cr.
                Transcript show: 'Current stack: '.
                travStack := traverse getStack.
                [travStack isEmpty]
                        whileFalse: [
                                tempStack push: travStack pop.
                        ].
                [tempStack isEmpty]
                        whileFalse: [
```

```
                                tempEntry := tempStack pop.
                                Transcript show: tempEntry print.
                                travStack push: tempEntry.
                        ].
                Transcript cr.
                nextNode := traverse next
        ].

Transcript cr; cr; show: 'Pre-order.'; cr.
traverse := PreorderTraversal new: nodeA.
nextNode := traverse first.
[nextNode notNil]
        whileTrue: [
                Transcript show: 'Current node: '.
                Transcript show: nextNode obj; cr.
                Transcript show: 'Current stack: '.
                travStack := traverse getStack.
                [travStack isEmpty]
                        whileFalse: [
                                tempStack push: travStack pop.
                        ].
                [tempStack isEmpty]
                        whileFalse: [
                                tempEntry := tempStack pop.
                                Transcript show: tempEntry print.
                                travStack push: tempEntry.
                        ].
                Transcript cr.
                nextNode := traverse next
        ].

Transcript cr; cr; show: 'Post-order.'; cr.
traverse := PostorderTraversal new: nodeA.
nextNode := traverse first.
[nextNode notNil]
        whileTrue: [
                Transcript show: 'Current node: '.
                Transcript show: nextNode obj; cr.
                Transcript show: 'Current stack: '.
                travStack := traverse getStack.
                [travStack isEmpty]
                        whileFalse: [
                                tempStack push: travStack pop.
                        ].
                [tempStack isEmpty]
                        whileFalse: [
                                tempEntry := tempStack pop.
                                Transcript show: tempEntry print.
                                travStack push: tempEntry.
                        ].
```

```
                Transcript cr.
                nextNode := traverse next
        ].

Transcript cr; cr; show: 'Recursive InOrder Traversal.'; cr.
InorderTraversal recurse: nodeA.
Transcript cr.

Transcript cr; cr; show: 'Recursive PreOrder Traversal.'; cr.
PreorderTraversal recurse: nodeA.
Transcript cr.

Transcript cr; cr; show: 'Recursive PostOrder Traversal.'; cr.
PostorderTraversal recurse: nodeA.
Transcript cr; cr.
! !

!TraversalTest class methodsFor: 'test method' stamp: 'RAB 3/4/2013
18:52'!
smallTest
        "Function for testing the BinTree and traversal classes.
        A three node binary tree is created and traversed."

| node1 node2 node3 traverse nextNode travStack tempStack tempEntry |

Transcript clear.
Transcript show: 'Test case: a binary tree with three nodes.'; cr.
Transcript show: 'A is the root node, C is left of A, B is right of
C.'; cr; cr.

node1 := BinTree new: 'A'.
node2 := BinTree new: 'C'.
node3 := BinTree new: 'B'.
node1 addLeftKid: node2.
node2 addRightKid: node3.

tempStack := Stack new.

Transcript show: 'In-order:'; cr.
traverse := InorderTraversal new: node1.
nextNode := traverse first.
[nextNode notNil]
        whileTrue: [
                Transcript show: 'Current node: '.
                Transcript show: nextNode obj; cr.
                Transcript show: 'Current stack: '.
                travStack := traverse getStack.
                [travStack isEmpty]
                        whileFalse: [
                                tempStack push: travStack pop.
```

```
                        ].
                [tempStack isEmpty]
                        whileFalse: [
                                tempEntry := tempStack pop.
                                Transcript show: tempEntry print.
                                travStack push: tempEntry.
                        ].
                Transcript cr.
                nextNode := traverse next
        ].

Transcript cr; cr; show: 'Pre-order.'; cr.
traverse := PreorderTraversal new: node1.
nextNode := traverse first.
[nextNode notNil]
        whileTrue: [
                Transcript show: 'Current node: '.
                Transcript show: nextNode obj; cr.
                Transcript show: 'Current stack: '.
                travStack := traverse getStack.
                [travStack isEmpty]
                        whileFalse: [
                                tempStack push: travStack pop.
                        ].
                [tempStack isEmpty]
                        whileFalse: [
                                tempEntry := tempStack pop.
                                Transcript show: tempEntry print.
                                travStack push: tempEntry.
                        ].
                Transcript cr.
                nextNode := traverse next
        ].

Transcript cr; cr; show: 'Post-order.'; cr.
traverse := PostorderTraversal new: node1.
nextNode := traverse first.
[nextNode notNil]
        whileTrue: [
                Transcript show: 'Current node: '.
                Transcript show: nextNode obj; cr.
                Transcript show: 'Current stack: '.
                travStack := traverse getStack.
                [travStack isEmpty]
                        whileFalse: [
                                tempStack push: travStack pop.
                        ].
                [tempStack isEmpty]
                        whileFalse: [
                                tempEntry := tempStack pop.
```

```
                                Transcript show: tempEntry print.
                                travStack push: tempEntry.
                        ].
                Transcript cr.
                nextNode := traverse next
        ].

Transcript cr; cr; show: 'Recursive InOrder Traversal.'; cr.
InorderTraversal recurse: node1.

Transcript cr; cr; show: 'Recursive PreOrder Traversal.'; cr.
PreorderTraversal recurse: node1.

Transcript cr; cr; show: 'Recursive PostOrder Traversal.'; cr.
PostorderTraversal recurse: node1.
! !
```