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.

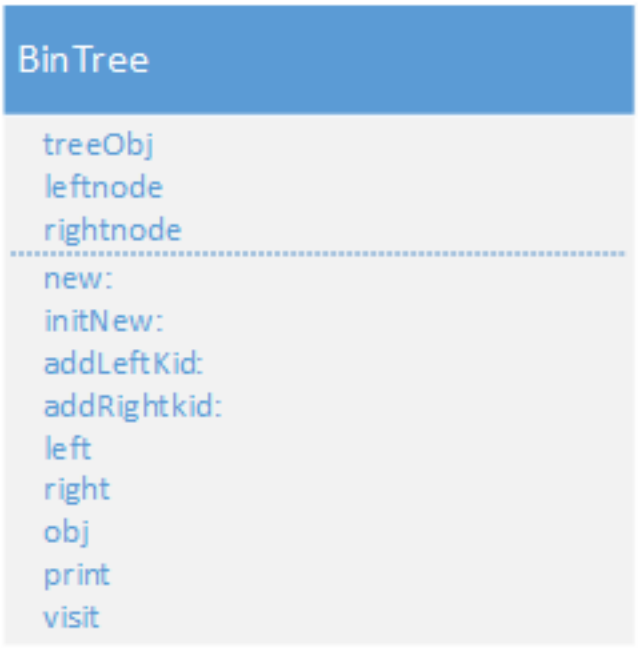
Usage of the traversal routines are broken into two specific cases, recursive and iterator. For recursive, there is no need to instantiate the traversal class. The method takes a root node as a parameter and calls the node's "visit" method.

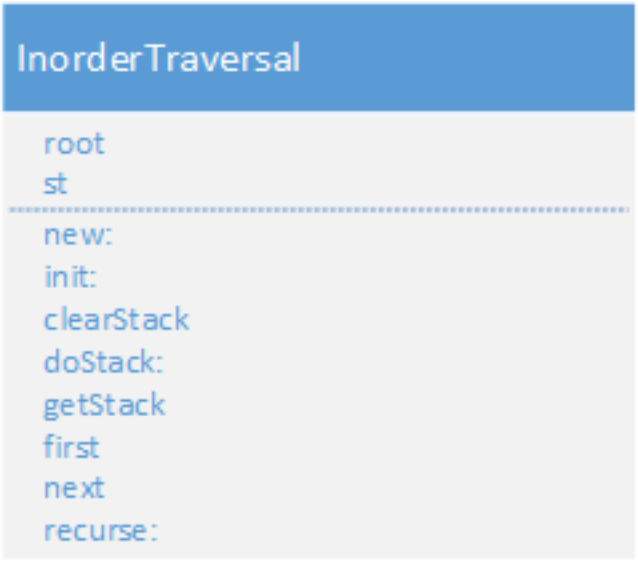
Example: PreorderTraversal recurse: rootNode

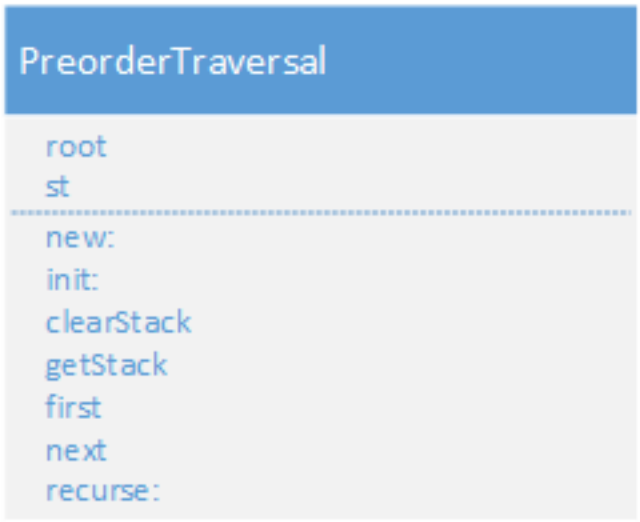
For the iterator usage, the class needs to be instantiated with the root node, then a call to "first" will start the traversal and return the first node. Any call to "first" will reset the traversal. Repetitive calls to "next" will return subsequent nodes.

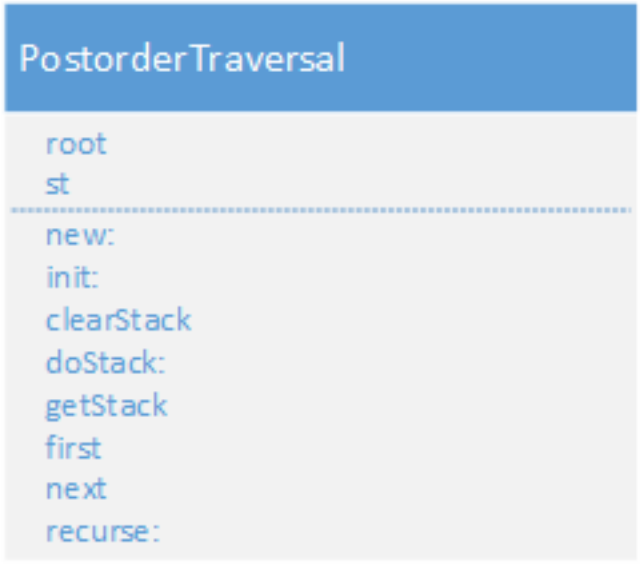
```
Example:    traversal := InorderTraversal new: node1
            nextLabel := traversal first
            (nextLabel notNil) whileTrue:
                    [action on node. nextLabel := traversal next]
```

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

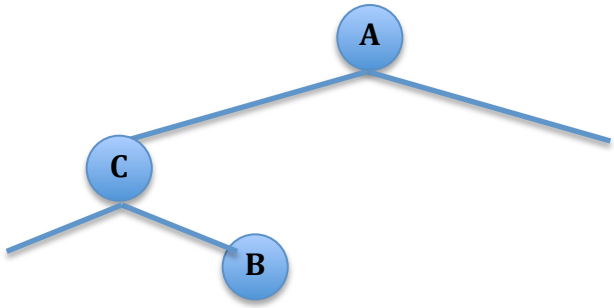
The TraversalTest class contains a small and large test method. The small method is the three nodes as specified in the project description. The large test is a 17-node test.

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:

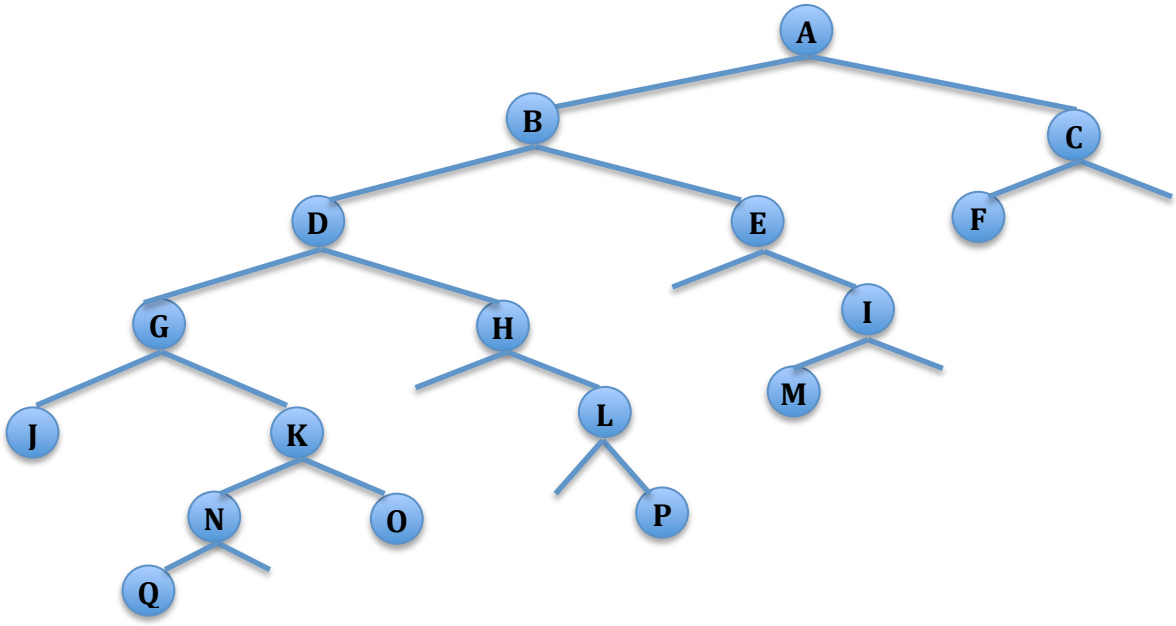
Post-order.
Current node: B
Current stack: 'A''C'
Current node: C
Current stack: 'A'
Current node: A
Current stack:

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.