Iterating over a Tree Structure

- Iterating over linear structures (lists, vectors) is simple – just start from one end and stop at the other
- How should we implement an iterator over a non-linear structure such as a tree?
- We have already discussed various tree traversals: pre-order, in-order, post-order.
- So, it is logical to implement iterators that move over the elements in the tree in these same orders.
- ***Warning***: in providing an iterator to client code the BST may not always maintain the BST ordering property if the client uses the reference to mutate the key/primary ordering field.

Iterator Dilemma

- Tree traversals were implemented recursively
  - When a traversal method is called by client code, it doesn't return until every node in the tree has been processed (likely with operations carried out on each element; for instance, visitors).
- On the other hand, iterators are non-recursive
  - The client code calls a function to advance the iterator by one element, and then it returns immediately.
- These paradigms are vastly different – thus, implementing an iterator is much more involved than just making a simple modification to the recursive traversal code, because we cannot "pause and resume" the recursion.

Recursion: inner functionality

- Consider how recursion is implemented in Java (and most other runtime environments)
- Function calls push a return address, parameter values, and space for local variables onto the runtime stack.
- In a BST traversal a reference to the current node is passed as an argument to the recursive function and these build up on the stack as the calls go deeper down the tree.
- So, the **top of the runtime stack contains a reference to the current node**, and the elements under it are the history required to backtrack up the tree to find the next node in a particular ordering.

Mimicking Recursion

- What if we keep track of the node stack ourselves?
  - Add a ***Stack*** object as a field member of the iterator class.

- By severing the node history from the call stack, we are no longer “trapped” inside a sequence of recursive calls, so we can pause the traversal at will.

- For a basic forward iterator, we must consider three cases:
  - What should be the state of the iterator initially?
  - What should be the state of the iterator after all nodes have been visited?
  - How does **next()** method modify the iterator object state to advance to the next element in the traversal order?

- These notes will not discuss how to implement reverse tree iterators.

Adding an Iterator

A BST iterator can be added in a very similar way as an iterator for a linked list.

To implement:

- Declare that your BST class implements the `Iterable` interface.
- add an inner class declaration for the `iterator` within the BST declaration.
- an iterator will store a pointer to a tree node; the `next()` method dereferences it to return the data element within that node.
- implement the `hasNext()` and `next()` methods of the inner iterator class.
- add the iterator-supplying method to the BST class: `iterator()`
  - One for each type of traversal.
- there's no case for an iterator-based insertion function in a BST, but there may be a case for an iterator-based deletion function

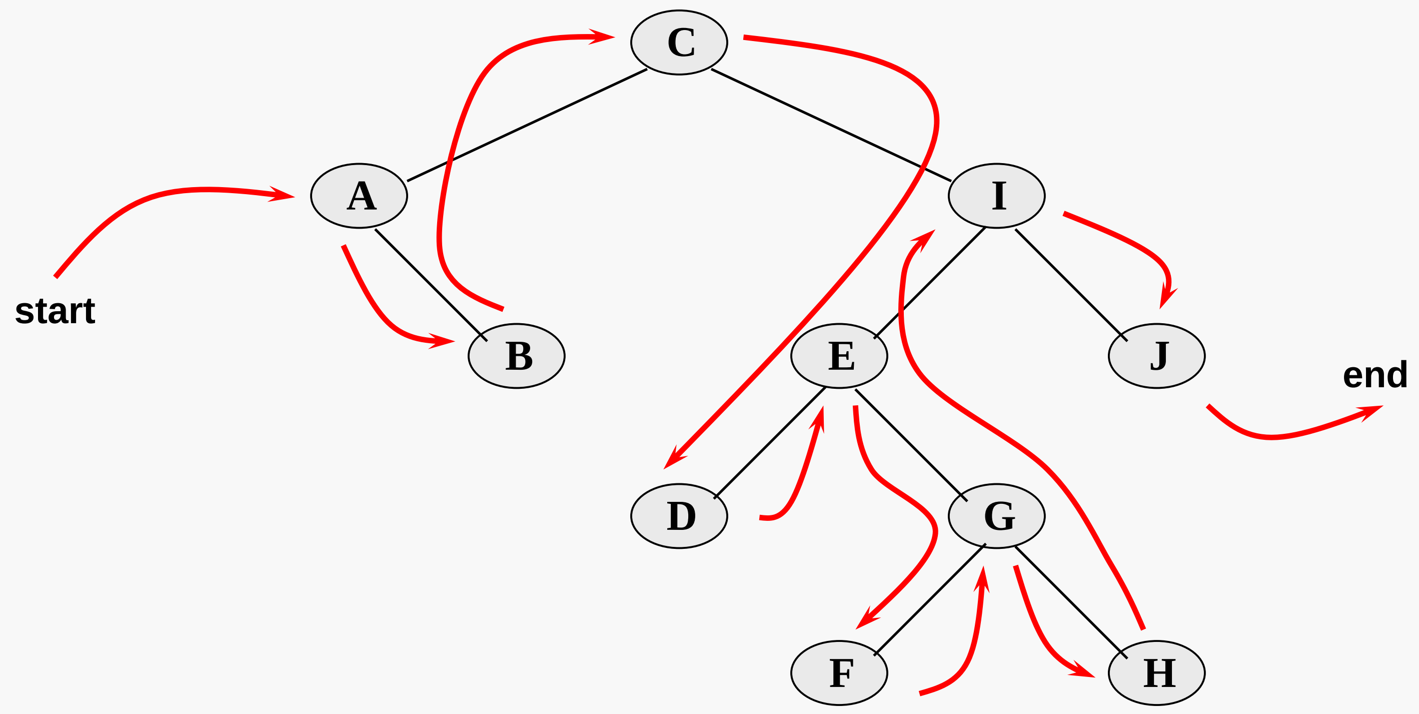
Consider adding a search function that returns an iterator:

- what traversal pattern should the iterator provide?
- is it necessary to modify the BST implementation aside from adding support functions?

Iterator Increment Logic

The only issue that is handled differently from the the linked list iterator is the pattern by which an iterator steps forward or backwards within the BST.

Consider stepping forward as in an inorder traversal:



The pattern is reasonably straightforward, but how can we move up from a node to its parent within the BST?

Pre-Order Iteration

- We will consider the pre-order traversal, because it is the simplest to implement in a non-recursive iterator
  - This is due to the fact that the operation on the element comes before either of the recursive calls

- Consider code for a pre-order traversal:

```
public void preOrder () {
    preOrderHelper(root);
}

private void preOrderHelper(BinaryNode<T> t) {
    if (current == null) return;

    System.out.println( t.element );
    preOrderHelper(t.left);
    preOrderHelper(t.right);
}
```

Implementing Pre-Order: initial state

- What operations occur in the recursive pre-order traversal before the `System.out.println( t.element );` line is reached?

- In this case, nothing – we call the traversal function for the root node, and the `System.out.println( t.element );` immediately executes.

- So, we can mimic this by implementing the iterator constructor to push the root node onto the iterator's stack to start the traversal.

Implementing Pre-Order: end state

- What happens when the traversal is complete?
  - The initial recursive call returns to the client code, so the runtime stack is returned to the same state it was in prior to the traversal.
- So, we can represent the end of the iteration by an empty node stack.
- In fact, this is exactly the case for all of the standard tree traversals – pre-order, in-order, and post-order.

Implementing Pre-Order: next()

- If we are sitting at a particular node in the tree, what is the next node in a pre-order traversal?
  - If the previous node was the parent, then the next node is the immediate left child
  - If we have already visited the left child, then the next node is the immediate right child
  - This assumes, of course, that those children exist – if not, we have to backtrack up the tree

- So, we pop the current node from the stack, then push its right and left children, if they exist.
  - Note that we must push right first, then left, so that the left child is the first to be popped back off later.

Pre-Order: next()

- Non-Tested Code

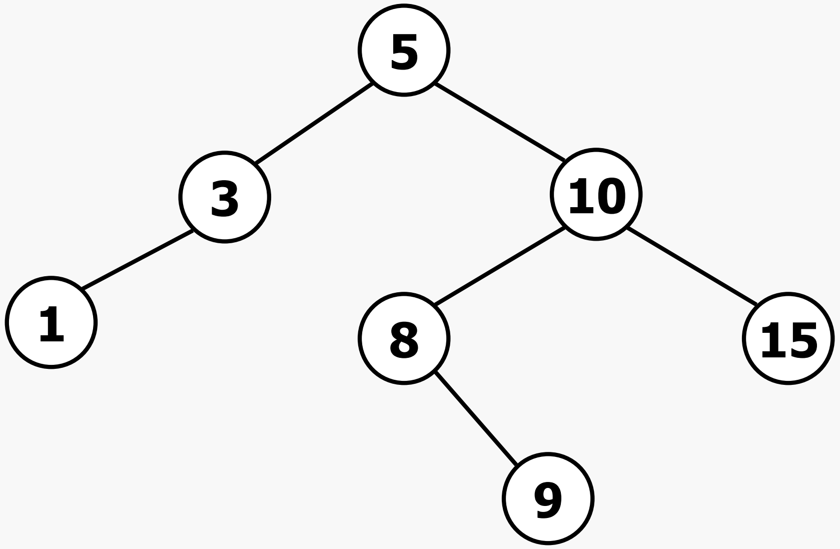
```
public T next() {
    if (!nodeStack.empty()) {
         BinaryNode<T> current = nodeStack.peek();
        nodeStack.pop();

        if (current.right != null) {
            nodeStack.push(current.right);
        }

        if (current.left != null) {
            nodeStack.push(current.left);
        }
    }

    return (current.element);
}
```

"*Beware of bugs in the above code; I have only proved it correct, not tried it.*"
– D.E. Knuth
(It's worse than that, the code above hasn't even been proved correct.)

Pre-Order Iterator Example

- This example traces the contents of the node stack over a pre-order traversal using the iterator. On each call to **next(),** the top element is popped off and its right and left children, if any, are pushed onto the stack, in that order. (The top of the stack is the **leftmost** element in the diagrams below.)

construction:	<**5**>
next():	<**3**, 10>
next():	<**1**, 10>
next():	<**10**>
next():	<**8**, 15>
next():	<**9**, 15>
next():	<**15**>
next():	<> (end)



In-Order Iteration

- Consider code for an in-order traversal:

```
public void inOrder () {
    inOrderHelper(root);
}

private void inOrderHelper(BinaryNode<T> t) {
    if (current == null) return;

    inOrderHelper(t.left);
    System.out.println( t.element );
    inOrderHelper(t.right);
}
```

- Logically, what does this traversal look like?
  - First, we go as far left as possible from the root, keeping track of the nodes we pass
  - At a leaf, we process the node, return to its parent, process it, and then move to the right child
  - But before processing the right child, we have to again go all the way left, and repeat

Implementing In-Order: initial state

- For the in-order traversal iterator to be properly positioned at the first node in the traversal, it is not enough to just push the root.

- The in-order constructor has to push all the nodes along the leftmost branch, starting from the root until we reach a node with no left child.

```
public inorder_iterator() {
    if (root != null){
         nodeStack = new Stack<BinaryNode<T>>();
        goLeftFrom(root);
    }
}

private void goLeftFrom(BinaryNode<T> t)
{
    while (t != null) {
        nodeStack.push(t);
        t = t.left;
    }
}
```

Implementing In-Order: next()

- If we are sitting at a particular node in the tree, what is the next node in a in-order traversal?
  - It will **never** be the left child – we have already visited those
  - If the node has a right child, then the next node is somewhere in the right subtree
  - More specifically, it is the leftmost node in the right subtree
- So, we pop the current node from the stack, then if it has a right child, we push it **and** all of the nodes down its leftmost branch.

In-Order: next()

- Pseudo-Code

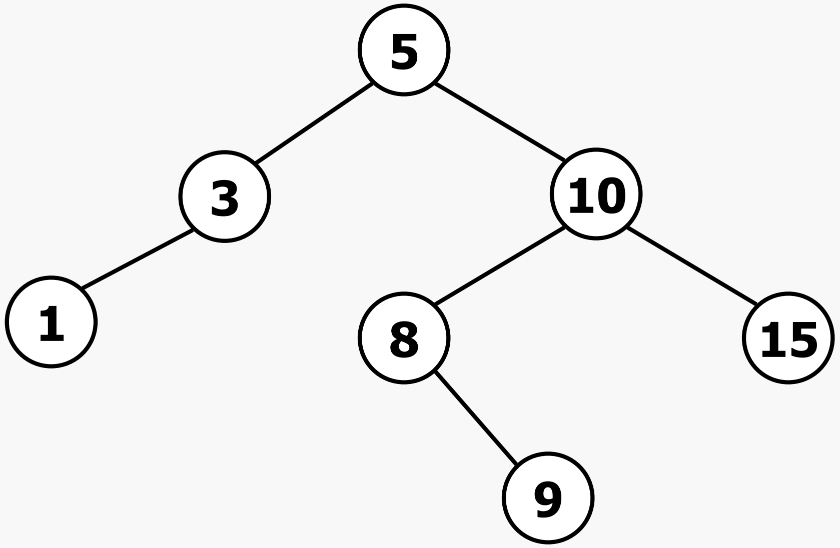
```
public T next(){
    if (!nodeStack.empty()) {
        BinaryNode<T> current = nodeStack.peek();
        nodeStack.pop();

        if (current.right != null) {
            goLeftFrom(current.right);
        }
    }

    return (current.element);
}
```

"*Beware of bugs in the above code; I have only proved it correct, not tried it.*"
– D.E. Knuth
(It's worse than that, the code above hasn't even been proved correct.)

In-Order Iterator Example

- This example traces the contents of the node stack over an in-order traversal using the iterator. Since multiple elements may be pushed at once, these are highlighted in red.

construction :	<**1**, 3, 5>
next():	<**3**, 5>
next():	<**5**>
next():=	<**8**, 10>
next():	<**9**, 10>
next():	<**10**>
next():	<**15**>
next():	<> (end)



Post-Order Iteration

- Consider code for a post-order traversal:

```
public void postOrder () {
    postOrderHelper(root);
}

private void postOrderHelper(BinaryNode<T> t) {
    if (current == null) return;

    inOrderHelper(t.left);
    inOrderHelper(t.right);
    System.out.println( t.element );
}
```

- Logically, what does this traversal look like?
  - First, we go as far left as possible from the root, until we reach a node with no left child.
  - If the node we stop at has a right child, we move to it, then again descend to the left as far as possible, and repeat.

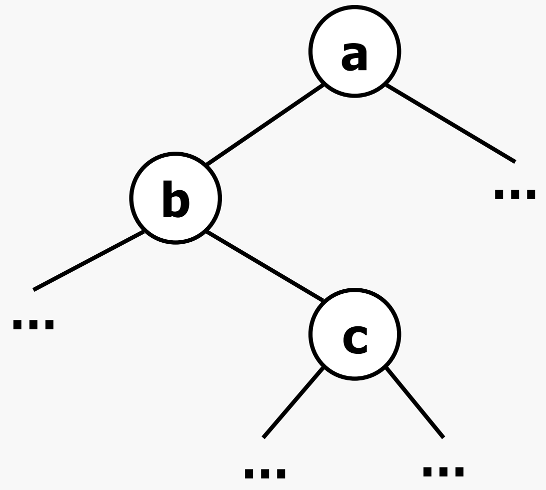
Post-Order Complications

- In the pre-order and in-order traversals, once we passed a node, we pushed it on the stack so we could process it on the way back up.
- After we processed that node on the way back up, we could pop it off and forget about it, because we never needed to consider it again later.
- In a post-order traversal, however, we pass over a node twice before we are ready to process it – first, to traverse its left children, then to traverse its right children.

Post-Order Complications

Consider this example:

- Starting from the root (a), we move left to its child (b)
- At this point, we have pushed (a) and (b) onto the stack, and continue to (b)'s left children, if any
- Once we have processed the left subtree under (b), we return to a point where (b) is the top of the stack
- But, we can't pop (b) off yet – we have to process its right subtree **and** keep (b) on the stack so that we know to stop there later



Post-Order Complications

- So, the stack in a post-order traversal does not only need to contain nodes, but also for each node a Boolean flag that indicates whether we have visited its right child or not.
  - When we reach a node during the backtracking process, we pop it off as usual
  - If we haven't visited its right child, then we set this flag, **push the node back onto the stack,** and then walk down the right subtree
  - If we already have visited its right child, then we stop here so that the client code can process the element

Post-Order Descending Logic

- We write a helper function to descend down the tree as described previously
- We keep **(node, wentRight)** pairs on the stack; **wentRight** is a Boolean flag that indicates if we have visited the node's right subtree
- We also keep track of the **next** node to visit
- Consider an arbitrary **(node, wentRight)** pair
  - If **node** has a left child, then push **(node, false)** onto the stack; the **next** node to visit is the left child
  - If **node** does not have a left child, then push **(node, true)** onto the stack; the **next** node to visit is the right child
- We stop when the **next** node becomes null

Post-Order Helper Function

- Pseudo-Code

```
private void visitSubtree(BinaryNode<T> t) {
    while (t != null) {
        BinaryNode<T> next;
        boolean wentRight;
        if (t.left == null) {
            next = t.right;
            wentRight = true;
        }
        else
        {
            next = t.left;
            wentRight = false;
        }

        nodeStack.push(new NodeInfo(t, wentRight));
        t = next;
    }
}
```

"*Beware of bugs in the above code; I have only proved it correct, not tried it*."
– D.E. Knuth
(It's worse than that, the code above hasn't even been proved correct.)

Implementing Post-Order: initial state

- The action taken by the post-order constructor is simply to use the traversal defined by the helper function above, starting from the root

```
public postorder_iterator() {
    if (root != null) {
        nodeStack = new Stack<NodeInfo>();
        visitSubTree(root);
    }
}
```

Implementing Post-Order: next()

- To advance the iterator to the next node, we follow the logic described above
  - Descend as far as possible until we hit a leaf
  - Process the leaf
  - Backtrack up the tree
  - If we hit a node during backtracking whose right subtree hasn't yet been visited, descend into it
  - Once the right subtree has been completely processed, only then do we process the internal node, and then backtrack again

In-Order: next()

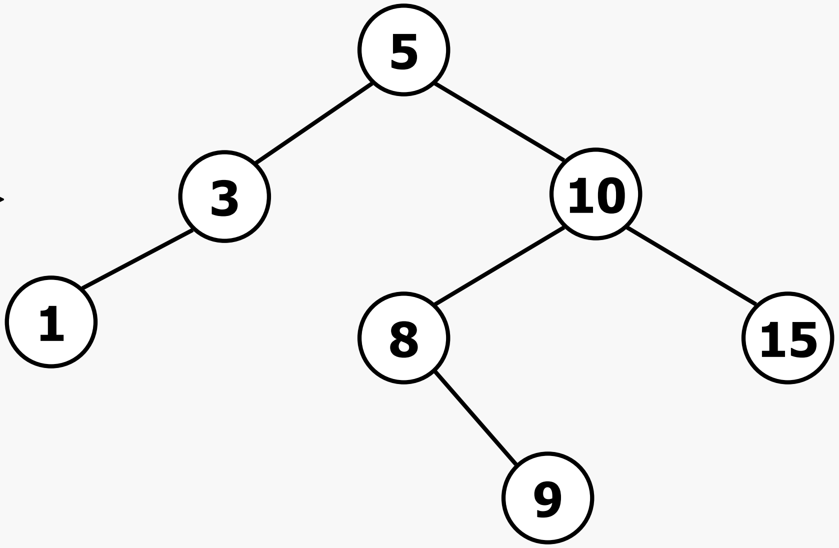
- Pseudo-Code

```
public T next() {
    if (!nodeStack.empty()) {
        NodeInfo current = nodeStack.peek();
        BinaryNode<T> next = current.getNode();
        nodeStack.pop();

        if (!nodeStack.empty()) {
            current = nodeStack.peek();

            if(!current.getWentRight()) {
                nodeStack.pop();

                current.setWentRight(true);
                nodeStack.push(current);

                visitSubtree(current.getNode().right);
            }
        }
    }

    return (next.element);
}
```

"*Beware of bugs in the above code; I have only proved it correct, not tried it.*"
– D.E. Knuth
(It's worse than that, the code above hasn't even been proved correct.)

Post-Order Iterator Example

- This example traces the contents of the node stack over a post-order traversal using the iterator. Since multiple elements may be pushed at once, these are highlighted in red. (Superscript **T/F** indicate whether **wentRight** is true or false, respectively.)
- constructor: <**1^T**, 3^F, 5^F>
- next(): <**3^T**, 5^F>
- next(): <**9^T**, 8^T, 10^F, 5^T>
- next(): <**8^T**, 10^F, 5^T>
- next(): <**15^T**, 10^T, 5^T>
- next(): <**10^T**, 5^T>
- next(): <**5^T**>
- next(): <> (end)



Post-Order Iterator - alternative

- When the node at the top of the Stack is removed the new top node is always the parent of the removed node, (except for the root).

- Use this fact to eliminate the wentRight flag.
- When incrementing, pop the top node & access the new top node.
- Compare the popped node to determine if it is the left or right child of the top node.
- If it is the left child then use visitSubtree to push the post-order right sub-tree node.
- If it is the right child, do nothing

When Are Two Iterators Equal?

- First, we need to decide whether iterators of *different* traversal types can be compared
- This begs the question: Is the iterator merely a **position** in the tree, or does it represent an entire **traversal** whose state changes as we move through the elements?
  - If we choose **position,** we should be able to compare different kinds of iterators (e.g., pre-order and in-order) to see if they point to the same node
  - If we choose **traversal,** this should **not** be allowed
- Allowing iterators of different types to be compared can raise implementation issues, because we must be able to compare all possible pairs of types of iterators
  - In this case, perhaps move some of the comparison logic into a common base class?

Implementing equals

- Since the top of the stack represents the current node in the traversal, it is sufficient to compare this element regardless of the rest of the stack contents

```
 public boolean equals(Object other) {//pre-order iterator

      if ( other == null ) return false;

      if ( !this.getClass().equals(other.getClass()) )
         return false;

      preorder_iterator rhs = (preorder_iterator) other;

      if (nodeStack.isEmpty() && rhs.nodeStack.isEmpty())
        return true;
      else if (nodeStack.isEmpty() != rhs.nodeStack.isEmpty())
        return false;
      else       //identity reference comparison
        return (nodeStack.peek() == rhs.nodeStack.peek());
}
```

"*Beware of bugs in the above code; I have only proved it correct, not tried it.*"
– D.E. Knuth
(It's worse than that, the code above hasn't even been proved correct.)

Can’t We Simplify All of This?

- The complexity of these iteration algorithms provides one argument for using parent pointers in a tree implementation.
- For the iterations described above, we can uniquely determine the next node based on only two properties: the node we are currently at, and the node from which we just came.
- With parent pointers, we can easily backtrack up the tree as needed, instead of requiring a stack to keep track of the history.