Mechanically Deriving Binary Tree Iterators with Continuation Defunctionalization

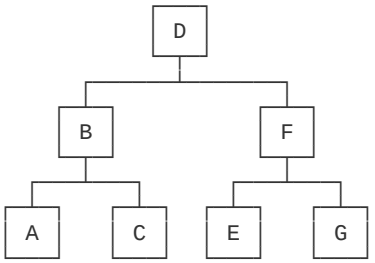
August 9, 2019
A twenty-two minute read
Tags: java, programming, algorithm

Binary tree is the simplest of tree data structures. It is a tree in which each node has at most two children. A tree traversal is a process of visiting each node in the tree, exactly once. There are multiple ways of traversing a binary tree in depth-first fashion with each traversal resulting in a different enumeration of the tree elements. These tree traversals are defined as simple recursive functions. But what if we want to write Java-style iterators for them? Is there a way to mechanically derive these iterators from the traversal functions? Let's find out.

Contents

Traversals and Iterations

This is a sample binary tree:



Different traversals of this tree will yield different sequences of elements:

Traversal	Output
in-order	ABCDEFG
pre-order	DBACFEG
post-order	ACBEGFD

The code for the recursive in-order traversal (traverse left child, then self, then right child) is very simple. Assuming a binary tree is represented with a `Tree` class like:

```
class Tree<T> {
  Tree<T> left;
  T content;
  Tree<T> right;
}
```

this code prints all elements of a tree in in-order:

```
static <T> void printRecursive(Tree<T> tree) {
  if (tree != null) {
    printRecursive(tree.left);
    System.out.print(tree.content);
    printRecursive(tree.right);
  }
}
```

But the code for an iterator which iterates over a tree in in-order is much more complicated:

```java
class InOrderIterator<T> implements Iterator<T> {
  private Tree<T> tree;
  private Stack<Tree<T>> stack = new Stack<>();

  InOrderIterator(Tree<T> tree) { this.tree = tree; }

  @Override
  public boolean hasNext() {
      return tree != null || !stack.isEmpty();
  }

  @Override
  public T next() {
    while (hasNext()) {
      if (tree != null) {
        stack.push(tree);
        tree = tree.left;
      } else {
        if (!stack.isEmpty()) {
          Tree<T> t = stack.pop();
          T content = t.content;
          tree = t.right;
          return content;
        }
      }
    }
    throw new NoSuchElementException();
  }
}

class Main {
  public static void main(String[] args) {
    Tree<String> tree = new Tree<>(...);
    InOrderIterator<String> inOrderIterator =
      new InOrderIterator<>(tree);
    while (inOrderIterator.hasNext()) {
      System.out.print(inOrderIterator.next());
    }
  }
}
```

The iterator code uses a `Stack` to simulate the program stack of the recursive traversal. It takes some thinking about the tree structure and the program flow to write this code and it is easy to get it wrong. Pre-order and Post-order iterators are even more complicated[1]. Is there a way to mechanically derive the iterators starting from the recursive traversals by following some rules? Indeed there is! Keep reading.

Binary Tree

Let’s start with some setup code:

```
import java.util.Iterator;
import java.util.NoSuchElementException;
import java.util.Stack;
import java.util.concurrent.ThreadLocalRandom;
import java.util.function.Consumer;
import java.util.function.Supplier;

class Utils {
  static <T> void printContent(T t) {
    System.out.printf(t + " ");
  }

  static String generateRandomAlphaString(int maxLength) {
    ThreadLocalRandom random = ThreadLocalRandom.current();
    int targetLength = random.nextInt(maxLength) + 1;
    StringBuilder sb = new StringBuilder(targetLength);
    for (int i = 0; i < targetLength; i++) {
      sb.append((char) random.nextInt('a', 'z' + 1));
    }
    return sb.toString();
  }

  static StringBuilder makeGuidelines(int times) {
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < times; i++) {
      sb.append("│ ");
    }
    return sb.append("├ ");
  }
}
```

With these imports and utility functions out of our way, let’s first define a Binary Tree:

```
class Tree<T> {
  Tree<T> left;
  T content;
  Tree<T> right;

  Tree(Tree<T> left, T content, Tree<T> right) {
    this.left = left;
    this.content = content;
    this.right = right;
  }

  public static <T> Tree<T> generate(
      int maxDepth, Supplier<T> gen, double nullProbability) {
    if (nullProbability < 0 || nullProbability > 1) {
      throw new IllegalArgumentException(
        "nullProbability must be between 0 and 1");
    }

    if (maxDepth == 0) {
      return null;
    }

    double rand = ThreadLocalRandom.current().nextDouble();
    if (rand < nullProbability) {
      return null;
    }

    Tree<T> left = generate(maxDepth - 1, gen, nullProbability);
    Tree<T> right = generate(maxDepth - 1, gen, nullProbability);
    return new Tree<>(left, gen.get(), right);
  }

  @Override
  public String toString() {
    return toStringLayout(0).toString();
  }

  private StringBuilder toStringLayout(int level) {
    StringBuilder sb = new StringBuilder()
      .append(Utils.makeGuidelines(level))
      .append(content)
      .append("\n");

    if (this.left == null && this.right == null) {
      return sb;
    }

    StringBuilder nullChildGuidelines =
      Utils.makeGuidelines(level + 1);
    if (this.left != null) {
```

```
51        sb.append(this.left.toStringLayout(level + 1));
52      } else {
53        sb.append(nullChildGuidelines).append("<NULL>\n");
54      }
55
56      if (this.right != null) {
57        sb.append(this.right.toStringLayout(level + 1));
58      } else {
59        sb.append(nullChildGuidelines).append("<NULL>\n");
60      }
61
62      return sb;
63    }
64  }
```

Each node in a binary tree has some content and two nullable child nodes. We have a constructor to create the tree. The `generate` function generates an arbitrary binary tree of a given maximum depth by using a random content generator. We use this function to generate trees to test our traversals and iterators.

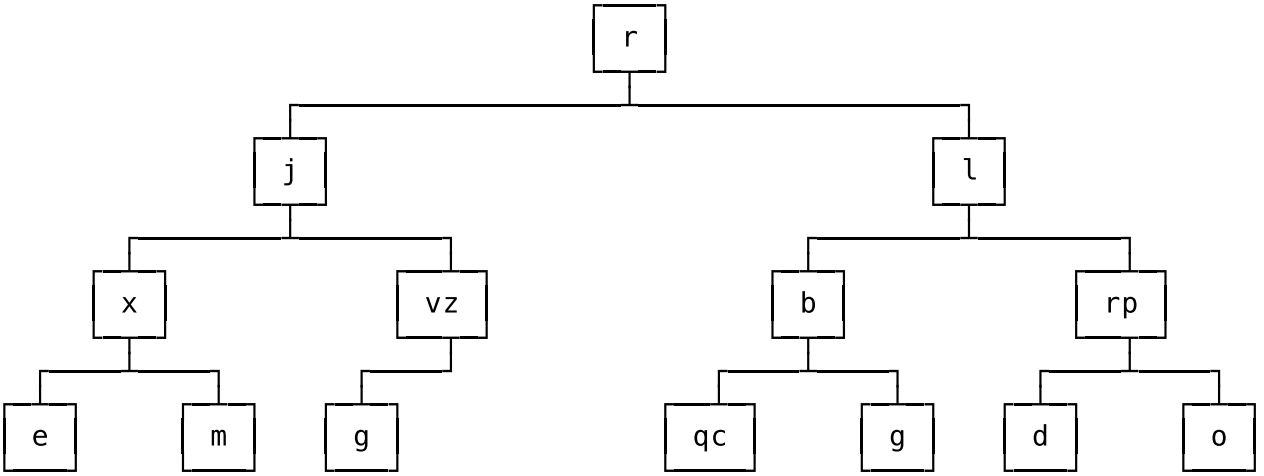
We also implement the `toString` method for the tree to create a string representation of the tree to help us in debugging. A sample run:

```
1  Tree<String> tree = Tree.generate(4, () ->
2    Utils.generateRandomAlphaString(2), 0.1);
3  System.out.println(tree);
```

Output:

```
├ r
│ ├ j
│ │ ├ x
│ │ │ ├ e
│ │ │ ├ m
│ │ ├ vz
│ │ │ ├ g
│ │ │ ├ <NULL>
│ ├ l
│ │ ├ b
│ │ │ ├ qc
│ │ │ ├ g
│ │ ├ rp
│ │ │ ├ d
│ │ │ ├ o
```

Pictorially:



We'll use this sample tree to test our code so refer back to this tree to check the correctness of the code.

Recursive Traversal

Credits first: this blog post and the code in it is inspired by The Best Refactoring You've Never Heard Of article (and talk) by James Koppel. In the talk, James shows how to transform a recursive in-order traversal into an iterative one. For this post, I've chosen to implement pre-order and post-order iterators.

Let's start with the recursive pre-order traversal which prints the content of every node in the tree in pre-order: each node's content is printed before its children's.

```
1  static <T> void printRecursive(Tree<T> tree) {
2    if (tree != null) {
3      Utils.printContent(tree.content);
4      printRecursive(tree.left);
5      printRecursive(tree.right);
6    }
7  }
8
9  printRecursive(tree);
10 // r j x e m vz g l b qc g rp d o
```

Short and sweet, simple and easy to understand. If the tree is not null, we print its content, then we recursively print the left and right child trees.

First thing to do is to extract out the print action into a function argument so that we can do different kinds of actions on the nodes instead of just printing them:

```
static <T> void iterateRecursive(
    Tree<T> tree, Consumer<T> action) {
  if (tree != null) {
    action.accept(tree.content);
    iterateRecursive(tree.left, action);
    iterateRecursive(tree.right, action);
  }
}
```

We use `Consumer` for the type of the action. Since `Consumer` is a functional interface, we can pass lambdas in its place. We can call it like this:

```
iterateRecursive(tree, Utils::printContent);
// r j x e m vz g l b qc g rp d o
```

This transformation was easy to grok. The next one requires a little head-tilting. We convert the simple recursion into a *Continuation-passing style* (CPS) recursion:

```
 static <T> void iterateCPS(
     Tree<T> tree, Consumer<T> action, Runnable cont) {
   if (tree != null) {
     action.accept(tree.content);
     iterateCPS(tree.left, action, () ->
       iterateCPS(tree.right, action, cont));
   } else {
     cont.run();
   }
}
```

So what are *Continuations*?

Continuations

In the usual direct imperative programming style, we write one statement after another, as a sequence of steps to execute. There is another way of thinking about it: after returning from executing one statement, the rest of the program — which can be thought of as a big statement itself — is run. In *Continuation-passing style*, this way is made explicit: each statement takes the rest of the program which comes after it as an argument, which it invokes explicitly. For example, if we have a program written in direct style like this:

```
static int start(String s) {
  int a = getSomething(s);
  doAnotherThing(a);
  return a;
}
```

It can be converted into an equivalent CPS style program like this:

```
1  static void startCPS(String s, Callable<Integer> cont) {
2    getSomethingCPS(s, (a) -> {
3      doAnotherThingCPS(a, () -> {
4        cont.call(a);
5      });
6    });
7  }
8
9  static void getSomethingCPS(String s, Callable<Integer> cont) {
10   cont.call(getSomething(s));
11 }
12
13 static void doAnotherThingCPS(int a, Runnable cont) {
14   doAnotherThing(a);
15   cont.run();
16 }
```

We see how each function call takes the rest of the program after it as a lambda and calls it explicitly to further the flow of the program. Instead of returning an `int` value, the `startCPS` function now takes a lambda as an additional `Callable` argument which it calls with the `int` value at the end of all the processing. These lambdas are known as *Continuations* because they **continue** the flow of the programs, and hence this style of writing programs is called the *Continuation-passing style*.

Comparing the direct and CPS recursive functions, we can now understand the transformation:

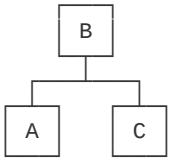
```
1  static <T> void iterateRecursive(
2      Tree<T> tree, Consumer<T> action) {
3    if (tree != null) {
4      action.accept(tree.content);
5      iterateRecursive(tree.left, action);
6      iterateRecursive(tree.right, action);
7    }
8  }
9
10 static <T> void iterateCPS(
11     Tree<T> tree, Consumer<T> action, Runnable cont) {
12   if (tree != null) {
13     action.accept(tree.content);
14     iterateCPS(tree.left, action, () ->
15       iterateCPS(tree.right, action, cont));
16   } else {
17     cont.run();
18   }
19 }
```

We first print the tree's content just like before. But instead of calling the function itself recursively twice for each child node, we call it only once for the left child node and pass a continuation lambda as the last parameter, which when called, calls the `iterateCPS` function for the right child node with the current continuation. This chain of continuations is called when the recursion bottoms out at the leftmost leaf node in line 17. Let's run it:

```
iterateCPS(tree, Utils::printContent, () -> { return; });
// r j x e m vz g l b qc g rp d o
```

We pass an empty lambda to start with which will be the last continuation to be called.

Readers are suggested to take a while to grok this transformation because this is a crucial step. Here's how the program call stack looks like for this simple tree:




```
iterateCPS("B", ac, return); // prints "B"
iterateCPS("A", ac, () ->
  iterateCPS("C", ac, return)); // prints "A"
iterateCPS(null, ac, () ->
  iterateCPS(null, ac, () -> iterateCPS("C", ac, return)));
iterateCPS(null, ac, () -> iterateCPS("C", ac, return));
iterateCPS("C", ac, return); // prints "C"
iterateCPS(null, ac, () -> iterateCPS(null, ac, return));
iterateCPS(null, ac, return);
return;
```

Readers should try to imagine (or work out) the program call stack when it runs on the sample tree and see how continuations are layered over continuations.

Defunctionalization

Defunctionalization is replacing functions with data. In this context, it means replacing the continuation lambdas with objects. The reason for doing Defunctionalization will become clear as we proceed.[2]

For defunctionalizing the continuations, we need to find out all possible cases of continuations we have:

```
static <T> void iterateCPS(
    Tree<T> tree, Consumer<T> action, Runnable cont) {
  if (tree != null) {
    action.accept(tree.content);
    iterateCPS(tree.left, action, () ->
      iterateCPS(tree.right, action, cont));
  } else {
    cont.run();
  }
}

iterateCPS(tree, Utils::printContent, () -> { return; });
```

Here, we have two cases of continuations which are emboldened above: first which recursively calls `iterateCPS` and second which is an empty lambda which terminates the recursion. To capture these two cases with an object, we use this class:

```
class Cont<T> {
  final Tree<T> tree;
  final Cont<T> next;

  Cont(Tree<T> tree, Cont<T> next) {
    this.tree = tree;
    this.next = next;
  }
}
```

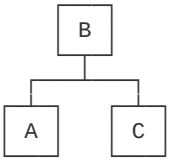
If the `tree` field is set, it's the first case, otherwise it's the second case. We also need the `cont` field to capture the current continuation which the first lambda captures in its third call argument. Now we replace all the lambdas with `Cont` objects and the lambda invocation with the two case of continuations:

```
 static <T> void iterateDefCPS(
     Tree<T> tree, Consumer<T> action, Cont<T> cont) {
   if (tree != null) {
     action.accept(tree.content);
     iterateDefCPS(tree.left, action, new Cont<>(tree.right, cont));
   } else {
     if (cont != null) {
       iterateDefCPS(cont.tree, action, cont.next);
     } else {
       return;
     }
   }
 }
```

Corresponding parts of the code have been emboldened to clearly see the transformation. We can run it and see if it works:

```
iterateDefCPS(tree, Utils::printContent, null)
// r j x e m vz g l b qc g rp d o
```

It works! To visualize it better, let's look at the program call stack again for the simple tree:



```
iterateDefCPS("B", ac, null); // prints "B"
iterateDefCPS("A", ac, Cont("C", null)); // prints "A"
iterateDefCPS(null, ac, Cont(null, Cont("C", null));
iterateDefCPS(null, ac, Cont("C", null));
iterateDefCPS("C", ac, null) // prints "C";
iterateDefCPS(null, ac, Cont(null, null));
iterateDefCPS(null, ac, null);
return
```

Again, readers are encouraged to spend some time thinking about and playing with this function to convince themselves that it works.

Path forward is pretty easy now.

Iteration

Notice how in the `iterateDefCPS` function, all the recursive calls are at tail call positions. That is, the last thing done in the `iterateDefCPS` function is to invoke itself recursively. That's what we achieved with Defunctionalization. Now we can do tail call optimization and replace the recursive calls with iteration:

```
 static <T> void iterate(
     Tree<T> tree, Consumer<T> action, Cont<T> cont) {
   while (true) {
     if (tree != null) {
       action.accept(tree.content);
       cont = new Cont<>(tree.right, cont);
       tree = tree.left;
     } else {
       if (cont != null) {
         tree = cont.tree;
         cont = cont.next;
       } else {
         return;
       }
     }
   }
 }
```

We just wrap the whole function body in a infinite `while` loop and instead of calling the function recursively, at the tail call points, we replace the function parameter variables with their new values. Rest of the code remains unchanged.

Upon running, it returns correct result:

```
iterate(tree, Utils::printContent, null);
// r j x e m vz g l b qc g rp d o
```

On to the final step, writing the *Iterator*.

Pre-order Iterator

To write the iterator, we need to realize that the `Cont` class is nothing but a Stack. Creating a new `Cont` object by passing it the current one is like pushing onto a stack. Similarly, writing `cont = cont.next` is like popping a stack.

With this realization, we simply hoist the parameters of the `iterateDefCPS` function into instance fields and replace `Cont` with a stack to transform it to an Iterator:

```
1  class PreOrderIterator<T> implements Iterator<T> {
2    private Tree<T> tree;
3    private Stack<Tree<T>> stack = new Stack<>();
4
5    PreOrderIterator(Tree<T> tree) { this.tree = tree; }
6
7    @Override
8    public boolean hasNext() {
9      return tree != null || !stack.isEmpty();
10   }
11
12   @Override
13   public T next() {
14     while (hasNext()) {
15       if (tree != null) {
16         T content = tree.content;
17         if (tree.right != null) {
18           stack.push(tree.right);
19         }
20         tree = tree.left;
21         return content;
22       } else {
23         if (!stack.isEmpty()) {
24           tree = stack.pop();
25         }
26       }
27     }
28     throw new NoSuchElementException();
29   }
30 }
```

`iterateDefCPS` returns when both `tree` and `cont` are `null`. So that is our iteration termination condition and it captured as such in the `hasNext` method of the iterator. `next` method is pretty much a copy of the `iterateDefCPS` function with `Cont` replaced by a `Stack` and an additional check to make sure to not push null elements onto the stack. We exercise the iterator like this:

```
PreOrderIterator<String> preOrderIterator =
  new PreOrderIterator<>(tree);
while (preOrderIterator.hasNext()) {
  Utils.printContent(preOrderIterator.next());
}
// r j x e m vz g l b qc g rp d o
```

It prints the elements in correct order. This completes our mechanical derivation of the pre-order iterator from recursive traversal code.

Post-order Iterator

Post-order iterator derivation turns out to be a bit more complicated that the pre-order one. Nevertheless, let's get started.

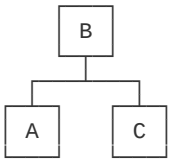
```
static <T> void printRecursive(Tree<T> tree) {
  if (tree != null) {
    printRecursive(tree.left);
    printRecursive(tree.right);
    Utils.printContent(tree.content);
  }
}

static <T> void iterateRecursive(
    Tree<T> tree, Consumer<T> action) {
  if (tree != null) {
    iterateRecursive(tree.left, action);
    iterateRecursive(tree.right, action);
    action.accept(tree.content);
  }
}

static <T> void iterateCPS(
    Tree<T> tree, Consumer<T> action, Runnable cont) {
  if (tree != null) {
    iterateCPS(tree.left, action, () -> {
      iterateCPS(tree.right, action, () -> {
        action.accept(tree.content);
        cont.run();
      }));
  } else {
    cont.run();
  }
}
```

We start with the recursive traversal function which prints the content of the node *after* recursively traversing the left and right child trees. We convert it to take an action instead of printing directly. Then we convert that form to a recursive CPS form.

The `iterateCPS` function this time has two lambda continuations. The first continuation calls `iterateCPS` recursively for the right child and the second one calls the action and invokes the current continuation. Let's call these functions for our sample tree:

```
printRecursive(tree);
// e m x g vz j qc g b d o rp l r
iterateRecursive(tree, Utils::printContent);
// e m x g vz j qc g b d o rp l r
iterateCPS(tree, Utils::printContent, () -> { return; });
// e m x g vz j qc g b d o rp l r
```

It works as expected but let's walk through the program call stack for our simple tree for `iterateCPS` to understand it better:




```
iterateCPS("B", ac, return);
iterateCPS("A", ac, () -> iterateCPS("C", ac, () -> {
  ac("B"); return
}));
iterateCPS(null, ac, () -> iterateCPS(null, ac, () -> {
  ac("A"); iterateCPS("C", ac, () -> { ac("B"); return })
}));
iterateCPS(null, ac, () -> {
  ac("A"); iterateCPS("C", ac, () -> {ac("B"); return })
});
ac("A"); // prints "A"
iterateCPS("C", ac, () -> { ac("B"); return });
iterateCPS(null, ac, () -> iterateCPS(null, ac, () -> {
  ac("C"); ac("B"); return
}));
iterateCPS(null, ac, () -> { ac("C"); ac("B"); return });
ac("C"); // prints "C"
ac("B"); // print "B"
return;
```

I hope the call stack makes it clear how `iterateCPS` works. Now we have to defunctionalize the continuations.

```
static <T> void iterateCPS(
    Tree<T> tree, Consumer<T> action, Runnable cont) {
  if (tree != null) {
    iterateCPS(tree.left, action, () -> {
      iterateCPS(tree.right, action, () -> {
        action.accept(tree.content);
        cont.run();
      }));
  } else {
    cont.run();
  }
}
```

```
static <T> void iterateCPS(
    Tree<T> tree, Consumer<T> action, Runnable cont) {
  if (tree != null) {
    iterateCPS(tree.left, action, () -> {
      iterateCPS(tree.right, action, () -> {
        action.accept(tree.content);
        cont.run();
      }));
  } else {
    cont.run();
  }
}

iterateCPS(tree, Utils::printContent, () -> { return; });
```

Unlike pre-order, in post-order `iterateCPS` function, we have three different cases of continuation as emboldened above. The first and second cases recursively call `iterateCPS` but the third one doesn't. The first one however works with the left child node and the second one with the right child node. That is the distinction between them.

To defunctiontionalize these three cases, we create a different `Cont` class:

```
class Cont<T> {
  final Tree<T> tree;
  final boolean isLeft;
  final Cont<T> next;

  Cont(Tree<T> tree, boolean isLeft, Cont<T> next) {
    this.tree = tree;
    this.next = next;
    this.isLeft = isLeft;
  }
}
```

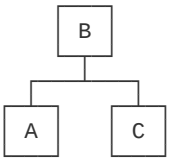
We have added a new boolean field `isLeft` to differentiate between the first and second cases. Now we replace the lambdas with `Cont` objects:

```
static <T> void iterateDefCPS(
    Tree<T> tree, Consumer<T> action, Cont<T> cont) {
  if (tree != null) {
    Cont<T> rCont = new Cont<>(tree, false, cont);
    Cont<T> lCont = new Cont<>(tree.right, true, rCont);
    iterateDefCPS(tree.left, action, lCont);
  } else {
    if (cont != null) {
      if (cont.isLeft) {
        iterateDefCPS(cont.tree, action, cont.next);
      } else {
        action.accept(cont.tree.content);
        iterateDefCPS(null, action, cont.next);
      }
    } else {
      return;
    }
  }
}
```

We create a right continuation object and then a left continuation object with the right one wrapped inside it. This nesting corresponds to the nested lambdas of the CPS form. In the invocation section, we can see the corresponding emboldened sections in the same order as before. Readers are suggested to mull over this transformation to convince themselves that it's correct. Let's run it now:

```
iterateDefCPS(tree, Utils::printContent, null);
// e m x g vz j qc g b d o rp l r
```

Let's walk through the call stack of `iterateDefCPS` for the simple tree:



```
iterateDefCPS("B", ac, null);
iterateDefCPS("A", ac, Cont("C", true, Cont("B", false, null)));
iterateDefCPS(null, ac,
  Cont(null, true,
    Cont("A", false, Cont("C", true, Cont("B", false, null)))));
iterateDefCPS(null, ac,
  Cont("A", false, Cont("C", true, Cont("B", false, null))));
ac("A"); // prints "A"
iterateDefCPS(null, ac, Cont("C", true, Cont("B", false, null)));
iterateDefCPS("C", ac, Cont("B", false, null));
iterateDefCPS(null, ac,
  Cont(null, true, Cont("C", false, Cont("B", false, null))));
iterateDefCPS(null, ac, Cont("C", false, Cont("B", false, null)));
ac("C"); // prints "C"
iterateDefCPS(null, ac, Cont("B", false, null));
ac("B"); // prints "B"
iterateDefCPS(null, ac, null);
return;
```

Turning `iterateDefCPS` into an iteration is straightforward now:

```
static <T> void iterate(
    Tree<T> tree, Consumer<T> action, Cont<T> cont) {
  while (true) {
    if (tree != null) {
      cont = new Cont<>(tree, false, cont);
      cont = new Cont<>(tree.right, true, cont);
      tree = tree.left;
    } else {
      if (cont != null) {
        if (cont.isLeft) {
          tree = cont.tree;
        } else {
          action.accept(cont.tree.content);
          tree = null;
        }
        cont = cont.next;
      } else {
        return;
      }
    }
  }
}
```

All the recursive calls to `iterateDefCPS` are at tail call position similar to pre-order case. As before, we wrap the function body in an infinite `while` loop and replace recursive calls with assignment to the function parameter variables.

This works too:

```
iterate(tree, Utils::printContent, null);
// e m x g vz j qc g b d o rp l r
```

Writing the post-order iterator is a bit trickier than the pre-order case. Since the `Cont` class now has a third field, we can't transform it directly into a `Stack` of `Trees`. We need to create another class – called `TreeTup` here – to capture the tree and the `isLeft` field value. Then we can create a stack of `TreeTups`. Rest of the transformation is pretty mechanical:

```java
class PostOrderIterator<T> implements Iterator<T> {
  private static class TreeTup<T> {
    final Tree<T> tree;
    final boolean isLeft;

    public TreeTup(Tree<T> tree, boolean isLeft) {
      this.tree = tree;
      this.isLeft = isLeft;
    }
  }

  private Tree<T> tree;
  private Stack<TreeTup<T>> stack = new Stack<>();

  PostOrderIterator(Tree<T> tree) { this.tree = tree; }

  @Override
  public boolean hasNext() {
    return tree != null || !stack.isEmpty();
  }

  @Override
  public T next() {
    while (hasNext()) {
      if (tree != null) {
        stack.push(new TreeTup<>(tree, false));
        if (tree.right != null) {
          stack.push(new TreeTup<>(tree.right, true));
        }
        tree = tree.left;
      } else {
        if (!stack.isEmpty()) {
          TreeTup<T> tup = stack.pop();
          if (tup.isLeft) {
            tree = tup.tree;
          } else {
            T content = tup.tree.content;
            tree = null;
            return content;
          }
        }
      }
    }
    throw new NoSuchElementException();
  }
}
```

We can compare the `iterate` function with the `next` method above and we see that it's pretty much the same code except for an additional check to not push null values onto the stack.

Now for the final run:

```
PostOrderIterator<String> postOrderIterator =
  new PostOrderIterator<>(tree);
while (postOrderIterator.hasNext()) {
  Utils.printContent(postOrderIterator.next());
}
// e m x g vz j qc g b d o rp l r
```

Conclusion

We have learned how to mechanically write pre-order and post-order iterators for binary trees. We started with simple recursive traversals and through a series of steps, we transformed them into Java-style iterators. *Continuation Defunctionalization* can be used to transform any recursion into an iteration. I hope it will come handy for you some day. The complete code with including the in-order iterator can be see here. Discuss this post on lobsters, r/programming or in the comments below.

Footnotes

1. These traversals can be generalized to take a function which they call with the content of each node. Such traversals are examples of Internal Iterators. Java-style iterators on the other hand are examples of External Iterators.↩
2. Learn more about *Defunctionalization* in the Defunctionalization at Work paper by Olivier Danvy and Lasse R. Nielsen.↩

