# IP Lecture 18: Binary Trees

Joe Pitt-Francis

—with thanks to Mike Spivey & Gavin Lowe—

# Bag of words: specifying the operations

```
/** Add an occurrence of word
  * post: count = count_0 ⊕ {word → count_0(word) + 1} */
def add(word: String)

/** Find the count stored for word
  * post: count = count_0 ∧ returns count(word) */
def count(word: String) : Int

/** Print the bag
  * post: count = count_0 ∧
  *     prints word → count(word), in alphabetical order, */
  *     for each word s.t. count(word) > 0, */
def printBag

/** Delete one occurrence of word
  * post: if count_0(word) = 0 then count = count_0
  *       else count = count_0 ⊕ {word → count_0(word) − 1} */
def delete(word: String)
```

# Binary search tree: abstraction function

Define $T(t)$ to be the set of `Tree` nodes reachable from $t$.

$$T(\texttt{null}) = \{\}$$

$$T(t) = \{t\} \cup T(t.\texttt{left}) \cup T(t.\texttt{right})$$

Each object represents

**Abs:** $count =$

$$\{st \to 0 \mid st \in \texttt{String}\} \oplus \{t.\texttt{word} \to t.\texttt{count} \mid t \in T(\texttt{root})\}$$

(The first term is necessary because we've specified *count* to be a *total* function, whereas the second term produces only a *partial* function.)

# The datatype invariant

The datatype invariant captures that: the tree satisfies the normal ordering property for a binary search tree; we only store information about words that have at least one occurrence in the bag; and the tree is acyclic and finite.

**DTI:** $\forall t \in T(\texttt{root}) \bullet$

$(\forall t' \in T(t.\texttt{left}) \bullet t'.\texttt{word} < t.\texttt{word}) \wedge$

$(\forall t' \in T(t.\texttt{right}) \bullet t'.\texttt{word} > t.\texttt{word}) \wedge$

$t.\texttt{count} > 0 \wedge$

$t \notin T(t.\texttt{left}) \cup T(t.\texttt{right}) \wedge$

T(root) is finite

# Searching in the tree

We can write a function `count` that returns the number of occurrences of a given word in the bag.

Here's a recursive version.

```
/** Find the count stored for a particular word */
def count(word: String) : Int = countInTree(word, root)

/** Find the count stored for a particular word, within t */
private def countInTree(word: String, t: Tree) : Int =
  if(t == null) 0
  else if(word == t.word) t.count
  else if(word < t.word) countInTree(word, t.left)
  else countInTree(word, t.right)
```

# Searching in the tree

And here's an iterative version.

```
/** Find count stored for particular word, iterative version */
def count(word: String) : Int = {
  var t = root
  // Invariant: if word is within the main tree, then it is
  // within the tree rooted at t:
  // for all t1: if t1 in T(root) and t1.word = word,
  // then t1 in T(t)
  while(t != null && t.word != word)
    if(word < t.word) t = t.left else t = t.right
  if(t == null) 0 else t.count
}
```

In fact, the Scala compiler will convert the recursive version into something equivalent to the iterative version, by eliminating the tail recursions.

# Adding a word

Here's a definition to add a word, similar to how you'd do it in Haskell.

```
def add(word: String) = root = addToTree(word, root)

/** Add an occurrence of word to the given tree, returning the
  * new tree. */
private def addToTree(word: String, t: Tree) : Tree =
  if(t == null) Tree(word, 1, null, null)
  else if(word == t.word) Tree(word, t.count+1, t.left, t.right)
  else if(word < t.word)
    Tree(t.word, t.count, addToTree(word, t.left), t.right)
  else Tree(t.word, t.count, t.left, addToTree(word, t.right))
```

However, that involves creating new tree nodes, whereas we could just update the existing nodes.

# Adding a word

The following version updates nodes in situ.

```
private def addToTree(word: String, t: Tree) : Tree =
  if(t == null) Tree(word, 1, null, null)
  else if(word == t.word){ t.count += 1; t }
  else if(word < t.word){ t.left = addToTree(word, t.left); t }
  else{ t.right = addToTree(word, t.right); t }
```

# An iterative version

We can produce an iterative version of `add`. However, this is easy to get wrong: if we iterate down the tree until we reach `null` then we have gone too far.

We need to iterate until we find a node `t` such that:

- The word `word` we're inserting matches `t.word`;
- We need to insert into `t`'s left subtree (i.e. `word < t.word`), but that subtree is currently null; or
- We need to insert into `t`'s right subtree (i.e. `word > t.word`), but that subtree is currently null.

Also, we need to treat `root == null` as a special case.

# An iterative version

```
def add(word: String) =
  if(root==null) root = Tree(word, 1, null, null)
  else{
    var t = root
    // Invariant: word needs to be inserted within the tree
    // rooted at t; t != null.
    while(word < t.word && t.left != null ||
          word > t.word && t.right != null)
    {
      if(word < t.word) t = t.left else t = t.right
    }
    if(word == t.word) t.count += 1
    // In the cases below, t.left, resp. t.right, is null
    else if(word < t.word) t.left = Tree(word, 1, null, null)
    else t.right = Tree(word, 1, null, null)
  }
```

# Printing the bag

Here's a simple recursive function to print the contents of the bag, in alphabetical order.

```
/** Print the contents of the entire bag tree */
def printBag = printTree(root)

/** Print the contents of tree t */
private def printTree(t: Tree) : Unit =
  if(t!=null){
    printTree(t.left)
    println(t.word+" -> "+t.count)
    printTree(t.right)
  }
```

# Printing the bag, iteratively

Printing a bag iteratively is slightly trickier. We will use a stack to record where we have to back-track to.

A stack is a data structure that stores a sequence of data. A new value `x` may be added to the stack using the operation `push(x)`. The operation `pop` removes the most-recently added value and returns it. The operation `isEmpty` tests whether the stack is empty. Exercise: specify this formally.

# Printing the bag, iteratively

When we encounter a non-null node, we will push it onto the stack. This will act to record that we still need to print the word on this node, and all the nodes in the right subtree. We then continue working with the left subtree.

More precisely, the algorithm will keep track of the current tree `t` and the stack `stack`. Collectively these represent a requirement to print all of `t`, and for each tree `t1` in `stack` to print the data in the top node, and the data in the nodes of the right subtree (in the order of the stack).

# Printing the bag, iteratively, using a stack

```
/** Print the contents of the tree */
def printBag = {
  var t = root
  val stack = new scala.collection.mutable.Stack[Tree]

  // Invariant: We still need to print t; and for each tree t1
  // in the stack, we still need to print the data in the top
  // node, and the data in the nodes of the right subtree
  // (in the order of the stack).
  while(t != null || !stack.isEmpty){
    if(t != null){ stack.push(t); t = t.left }
    else{
      val t1 = stack.pop
      println(t1.word+" -> "+t1.count)
      t = t1.right
    }
  }
}
```

# Deleting

To delete a word, the best approach is to write a subsidiary recursive function:

```
/** Delete one occurrence of word from the tree. */
def delete(word: String) : Unit = root = deleteFromTree(word, root)

/** Delete one occurrence of word from t, returning the
  * resulting tree. */
private def deleteFromTree(word: String, t: Tree) : Tree =
  if(t == null) null
  else if(word < t.word){ t.left = deleteFromTree(word, t.left); t }
  else if(word > t.word){ t.right = deleteFromTree(word, t.right); t }
  else if(t.count > 1){ t.count -= 1; t }
  // delete the contents of this node
  else if(t.left == null) t.right
  else if(t.right == null) t.left
  else{ ... }
```

# Deleting

We've reached a point where we want to delete the data on the node `t`, and merge the two sub-trees, both of which are non-`null`. What we will do is find the node with the smallest word in the right sub-tree: we can find this node by following `left` pointers within `t.right` until we reach a node whose left sub-tree is `null`. We then delete this node from `t.right`, and move the data from this node up to the node `t`.

# Deleting

```
    ...
    else{
      val (w, c, newR) = delMin(t.right)
      t.word = w; t.count = c; t.right = newR
      t
    }

  /** Delete the minimum node of t, returning the word and count
    * from that node, and the resulting tree.
    * Precondition: t!=null */
  private def delMin(t: Tree) : (String, Int, Tree) =
    if(t.left==null) (t.word, t.count, t.right)
    else{
      val (w, c, newL) = delMin(t.left)
      t.left = newL; (w, c, t)
    }
```

# Testing

```
class BinaryTreeTest extends FunSuite{
  val bag = new BinaryTreeBag

  test("one"){
    for(w <- List("c", "a", "f", "k", "d", "e", "1", "f", "g",
                  "l", "z", "y", "s", "r", "q"))
      bag.add(w)
    bag.print
    assert(bag.count("c")===1); assert(bag.count("a")===1)
    assert(bag.count("f")===2); assert(bag.count("p")===0)
  }
  test("delete repeated entry"){
    bag.delete("f"); assert(bag.count("f")===1);
  }
  ...
}
```

# Tree balance

Most operations have a complexity $O(d)$, where $d$ is the maximum depth of any leaf in the tree. The tree constructed in the `Scalatest` excerpt on the previous slide was unbalanced. (The leaf containing "1" was at depth 2 and the leaf containing "q" was at depth 9.)

What can we do?

- Monitor the amount of "unbalance" (exercise);
- Store unbalance of subtrees in nodes and rotate when needed;
- Use red-black tree (which continuously rotates subtrees);
- Use splay tree (which moves most used nodes towards root).

# Aside: higher-level property testing

`ScalaCheck` is a unit testing library that uses property specifications with random automatic test data generation.

- High level descriptions. For example, $\forall$ sets $A, B$: $\#(A \cup B) \leq \#A + \#B$
- Test case minimisation: what's the smallest test input that fails?
- Custom random input generators.
- Property-based tests give more testing for less code.
- Automatic data-generation allows us to focus on the purpose of test. case, rather enumerating corner cases.
- We can combine property-based tests (`ScalaCheck`) with assertion-based tests (`ScalaTest`).

# Setting up a property test with scalacheck

```
import org.scalacheck.Properties
import org.scalacheck.Prop.forAll
import org.scalacheck.Gen

object BagSpecification extends Properties("Bag") {
  val smallWord = for {word <- Gen.alphaStr}
                  yield word.take(2).toUpperCase
  println("Word sample: "+smallWord.sample)

  property("Words are small") =
    forAll(smallWord){ n =>  n.length<=2}
  property("Failing small word") =  // This one should fail
    forAll(smallWord){ n =>  n.length==2}

  val listWord = Gen.listOf(smallWord)
  println("List sample "+listWord.sample.get.sorted)

  ...
```

# Higher-level property test for a bag

```
  ...
  property("List count vs bag") = forAll(listWord){ alist =>
    val bag = new BinaryTreeBag // Make bag
    for (w <- alist) bag.add(w)
    var c_match = true
    for (w <- alist){
      c_match = c_match && (alist.count(_==w) == bag.count(w))
    }
    c_match
  }
}
```

```
Word sample: Some(TB)
List sample List(, AE, AG, AM, AO, AZ, AZ, BE, BK, BO, BR, BY, CY
+ Bag.Words are small: OK, passed 100 tests.
! Bag.Failing small word: Falsified after 0 passed tests.
> ARG_0: ""
+ Bag.Sorted list maps to bag: OK, passed 100 tests.
```

# White-box property testing

`ScalaCheck` also allows us to check that the internals of the modules we are testing behave as expected.

This may require some refactoring of our code. Essentially every operation can be split into separate component functions: `run`, `preCondition`, `postCondition`.... The testing framework can then automatically check pre- and post-conditions before and after each actual operation.

We can also add operations to classes which automatically check data type invariants.

"Don't write tests—generate them!"
John Hughes (creator of Haskell's QuickCheck)

# Summary

- Representing binary trees using reference-linked nodes;
- Adding, searching, printing, deleting;
- Property checking.
- Next time: Priority queue.