

DM509 Project 2 - Scala

Søren Elmely Pettersson

October 17, 2013

1 Specification

Using the given template for our implementation, namely the three files **HuffTree.scala**, **Huffman.scala**, and **HuffTests.scala**, we are to implement and test Huffman encoding and decoding in a purely function style. Changing in the template is allowed, but the 3 interface methods **createCodeTree**, **encode** and **decode** should keep their function signatures.

To actually produce something that adheres to that of functional programming, it is strictly forbidden to use the keyword **var** or anything from the **package scala.collection.mutable** collection.

The methods that are to be implemented will be discussed in the next section.

2 Design

2.1 `def weight(tree: HuffTree): Int`

Calculates the sum of the weights of all leaves in **tree**. Due to the structure of our HuffTree's the weight can actually just be read directly from the case classes, Node and Leaf, of the HuffTree class.

2.2 `def combine(left: HuffTree, right: HuffTree): HuffTree`

Combines two HuffTree's by creating a new internal node, and making the HuffTrees right and left subtree respectively. The weight of the new node will be the sum of the two subtrees - here we can utilize our weight function.

2.3 `def frequencies(chars: List[Char]): List[(Char, Int)]`

Converts a list of characters to a frequency table with tuples of (c,f(c)). This should be implemented in a very compact manner, utilizing some of the functionality of the inbuild methods of Scala - map and groupBy could be usefull here.

2.4 `def combineTrees(trees: SortedSet[HuffTree]): HuffTree`

Creates the Huffman Tree by repeatedly combining the 2 leftmost elements from **trees**, which is a **SortedSet**, so these will be the smallest elements. After combining these elements they will have to be reinserted into a version of **trees** without the first two elements - essentially combining the two first elements, as to shorten the list by one. Recursion will be usefull here.

2.5 def createCodeTree(chars: List[Char]): HuffTree

We combine the functionality of **frequencies**, **orderedLeaves** and **combineTrees** to create a HuffTree. Combining these in "the right order" is sufficient to complete this implementation. Given a frequency table, we use the supplied **orderedLeaves** function from the template, and lasty we will combine these using **combineTrees**.

2.6 def convert(tree: HuffTree): CodeTable

Converts a Huffman Tree into a **CodeTable**, which is an immutable **Map[Char,List[Bit]]**. Pattern matching on **tree**, utilizing recursion and an accumulator to "remember" the path to each character should suffice.

2.7 def codeBits(table: CodeTable)(char: Char): List[Bit]

Looks up in **table** to find the list of bits accosiated with the given **char**. Since table is a **Map** we can utilize some of the build in methods for these, to simply get the requested list of bits, given the **char**.

2.8 encode(tree: HuffTree)(text: List[Char]): List[Bit]

Encodes **text** using a code table. This is where we will create the codetable, given **tree** annd utilizing the **convert** method. Afterwards, clever use of some sort of **Map** should get us the encoding using the code table, and the method **codeBits** on all elements of the **text**.

2.9 getLetter(tree: HuffTree, bits: List[Bit]): (Char, List[Bit])

Given an encoded string, returns a tuple, with the first character obtained from the list of bits, given a **tree**, as well as the remaining **bits**. Pattern matching on **tree** and utilizing recursion, we can navigate around the tree given **bits** until we reach a basecase - meaning a leaf, which contains a character.

2.10 def decode(tree: HuffTree)(bits: List[Bit]): List[Char]

Decodes a string of bits with a given **tree** using **getLetter**. Again we will use pattern matching, this time on bits - since we want to utilize **getLetter** until the list is empty, pattern matching on bits allows us to monitor the cases of bits being either empty, or nonempty. Intermediate values will be neccesary, to gain acces to the **char** returned from **getLetter** as well as the remaining list of **bits**.

3 Implementation

3.1 def weight(tree: HuffTree): Int

We pattern match on the given **tree**. There are two cases. We either have a **Leaf**, in which case its second value is returned, this being the weight of the leaf (essentially the frequency of the character). The other case is if we have a **Node**, with a left and a right tree. The structure of our HuffTree's dictate that the weight of a node is the sum of the weight of the subtrees. So no calculation is needed, and again we simply return the weight given by the Node.

```
def weight(tree: HuffTree): Int = tree match {  
    case Leaf(char, vaegt) => vaegt  
    case Node(vaegt, left, right) => vaegt  
}
```

3.2 def combine(left: HuffTree, right: HuffTree): HuffTree

Given two HuffTree's, namely **left** and **right**, we are to combine them. We simply return a new node whose left and right subtrees, are, **left** and **right**. The weight of the new node is the sum of the weight of the subtrees. So, we utilize our **weight** function to obtain these and sum them.

```
def combine(left: HuffTree, right: HuffTree): HuffTree =  
    (Node((weight(left)+weight(right)), left, right))
```

3.3 def frequencies(chars: List[Char]): List[(Char, Int)]

Given a list of characters **chars**, we wish to group them, and we do so by the identity function ($w \rightarrow w$). So, the characters that are the same are grouped. We now wish to map these, according to the size of these respective lists, meaning their frequencies. The amount of elements in a list of the same characters are exactly the amount of times that specific character is found in **chars**. So, when we map to the size of this list, we get the frequency. We obtain mappings for each character, which is seen as sets of tuples. Therefor, all we need to do is to get a list by **.toList**.

```
def frequencies(chars: List[Char]): List[(Char, Int)] = (  
    chars.groupBy(w=>w).mapValues(_.size).toList    )
```

3.4 def combineTrees(trees: SortedSet[HuffTree])

We pattern match on **treess**. We have two cases. Either the **SortedSet** has only a single element, in which case we simply return that. The other case, is that we have more than one element. We combine the first two trees, which are the smallest (think of **treess** as a sorted list given the weight of the HuffTree's). So, we call our **combine** method on the first element of **treess**, using firstKey, as

well as the second smallest element, being the smallest element if we drop the first element in **trees**, and then take the firstKey afterwards. As said, these are combined using **combine**. They need to be added to a version of **trees** where the now 2 smallest elements are not present (since we replace them by a new, combined tree consisting of them). So, we drop 2 from trees, and add to it our newly combined element. SortedSet assures that it finds its rightful place. We call **combineTrees** again recursively on the resulting **SortedSet**, which now essentially has had its smallest element removed (combined with the second smallest), shortening it by 1, meaning that we are bound to hit our basecase at some point.

```
def combineTrees(trees: SortedSet[HuffTree]): HuffTree = trees match {
  case x if x.size < 2 => trees firstKey
  case x if x.size > 1 =>
    combineTrees((trees drop 2)+
      (combine(trees firstKey, (trees drop 1) firstKey)))
}
```

3.5 def createCodeTree(chars: List[Char]): HuffTree

Given a list of characters **chars**, we first obtain a list of Tuples giving the frequency of each character, using our method **frequencies**. Afterwards, we utilize the supplied **orderedLeaves** to obtain a **SortedSet** containing HuffTrees (Leaves at this point). We then use our **combineTrees** on the resulting list, to obtain a code tree, combining the list of Leaves into a single Huffman Tree.

```
def orderedLeaves(freqs: List[(Char, Int)]): SortedSet[HuffTree] =
  SortedSet[HuffTree](freqs.map(Function.tupled(Leaf)): _*)

def createCodeTree(chars: List[Char]): HuffTree = (
  combineTrees(orderedLeaves(frequencies(chars)))
)
```

3.6 def convert(tree: HuffTree, acc: List[Bit] = Nil): CodeTable

We pattern match on **tree**. If we have a leaf, we map the character into the list **acc**. In the other case, if we have a Node, we call **convert** recursively on both the right and the left tree. When we go left, we prepend a 0 to our **acc**, to keep track of the path we've taken when we eventually end up in a leaf, and thus a character. Similarly, we prepend the **acc** with a 1 if we go right. Since the notation of $x :: xs$ is easier to handle than that of adding the bits to the end of the list, we've used it here. This simply means, that when we do reach a character, and wish to map it to the string of bits that lead to it, we just need to reverse this list (since we prepend, the list is reversed, if you consider a path to start from the top, which we do!).

```
def convert(tree: HuffTree, acc: List[Bit] = Nil): CodeTable = tree match {
  case Leaf(char1, int) =>
    Map(char1 -> (acc reverse) )
  case Node(w, left, right) =>
    convert(left, 0 :: acc) ++ convert(right, 1 :: acc)
}
```

3.7 def codeBits(table: CodeTable)(char: Char): List[Bit]

We are given a code table **table** which is a **Map[Char, List[Bit]]**. Given a character **char** we simply use the `.getOrElse` method of a Map, to return the List of bits a given character maps to, if possible, or else we return Nil.

```
def codeBits(table: CodeTable)(char: Char): List[Bit] = (
  table.getOrElse(char, Nil)
)
```

3.8 encode(tree: HuffTree)(text: List[Char]): List[Bit]

First, given a HuffTree **tree**, we convert it to a CodeTable using **convert**. With our newly obtained codeTable, we utilize **codeBits** on each element to obtain the list of bits corresponding to its encoding given the codeTable (constant time). We use **flatMap** to apply our function **codeBits** to each element in the list of chars, as to in the end, end up with a list of bits.

```
def encode(tree: HuffTree)(text: List[Char]): List[Bit] = {
  val blazeit = convert(tree)
  text.flatMap(x => codeBits(blazeit)(x))
}
```

3.9 getLetter(tree: HuffTree, bits: List[Bit]): (Char, List[Bit])

We pattern match on **trees**. In our basecase, we have a Leaf, and if so, we return a tuple consisting of **char** as well as the list of bits **bits**

In our other case, we have a Node. If the the element in our string of bits (`bits.head`) is 0, we call **getLetter** recursively on the left subtree of Node, now with a version of the list of bits where the first element, which we just read, is removed. Conversely, if we don't read a 0, it must be a 1, so the else clause calls **getLetter** recursively on the right subtree, again with a version of the list of bits without the element we just read. So, we simply traverse the list of bits, make moves in the tree according to the bits read, until we end up in a leaf, in which case we return this character, as well as the remaining bits, together in a tuple (Char, List[Bit]).

```
def getLetter(tree: HuffTree, bits: List[Bit]): (Char, List[Bit]) = tree match {
  case Leaf(char, int) =>
    (char, bits)
  case Node(w, l, r) =>
    if (bits.head == 0)
      getLetter(l, bits.drop(1))
    else
      getLetter(r, bits.drop(1))
}
```

3.10 def decode(tree: HuffTree)(bits: List[Bit]): List[Char]

We pattern match on **bits**. Our base case is an empty list, in which case we return Nil. Our other case is bits being nonempty, as denoted by the case `_`. In this case, we need an intermediate val, being the tuple returned by **getLetter** called with the HuffTree **tree** and our list of bits **bits**. The return value is the last line evaluated, so here we ensure that we return a list of characters. The character just found using **getLetter** is prepended to the list, with the remainder of this list of characters, being a recursive call to **decode** with the same tree, and now with the list of bits with the remaining bits, given by the intermediate tuple from the **getLetter** call, namely the list of bits **remain**.

```
def decode(tree: HuffTree)(bits: List[Bit]): List[Char] = bits match {
  case Nil => Nil
  case _ =>
    val (char, remain) = getLetter(tree, bits)
    char :: decode(tree)(remain)
}
```

4 Testing

We test each method in turn, to ensure they work as intended, before we eventually use them to implement other methods. I will show the lines from **HuffTests.scala** for a given test, as well as the output in form of the terminal.

4.1 def weight(tree: HuffTree): Int

We don't have the functionality to create HuffTree's yet, so instead we test with some "hardcoded" trees, meaning we just spell out the nodes and leaves ourselves.

```
val x = Huffman.weight(Node(9, Node(7, Leaf('c', 3), Leaf('b', 4)), Leaf('c', 2)))
println(x)
```

```
Søren@Sorene /cygdrive/c/Datalogi/Programmeringssprog/Projekt2/project_handout_a
ss2
$ scala HuffTests.scala
9
```

As can be seen, the weight of the input tree is specified to 9, which is correctly returned.

4.2 def combine(left: HuffTree, right: HuffTree): HuffTree

We once again "hardcode" our trees since we do not have the functionality to construct them otherwise until later.

```
val y = Node(9, Node(7, Leaf('c', 3), Leaf('b', 4)), Leaf('c', 2))
val z = Node(9, Node(7, Leaf('c', 3), Leaf('b', 4)), Leaf('c', 2))
val a = Node(4, Node(2, Leaf('c', 1), Leaf('b', 1)), Leaf('y', 2))
val tree1 = Huffman.combine(y, z)
val tree2 = Huffman.combine(a, a)
println(tree1)
println("")
println(tree2)
println("")
```

```
Søren@Sorene /cygdrive/c/Datalogi/Programmeringssprog/Projekt2/project_handout_a
ss2
$ scala HuffTests.scala
Node(18, Node(9, Node(7, Leaf(c, 3), Leaf(b, 4)), Leaf(c, 2)), Node(9, Node(7, Leaf(c, 3), Leaf(b, 4)), Leaf(c, 2)))
Node(8, Node(4, Node(2, Leaf(c, 1), Leaf(b, 1)), Leaf(y, 2)), Node(4, Node(2, Leaf(c, 1), Leaf(b, 1)), Leaf(y, 2)))
```

The trees are simply added as subtrees to a new node, as can be seen. The new weight is calculated correctly.

4.3 def frequencies(chars: List[Char]): List[(Char, Int)]

We input a list of characters as follows:

```
val buu = "abracadabra".toList
println(frequencies(buu))
```

```
Søren@Sorene /cygdrive/c/Datalogi/Programmeringssprog/Projekt2/proj
ss2
$ scala HuffTests.scala
List((a,5), (b,2), (c,1), (r,2), (d,1))
```

We get a list of tuples of char and int as desired, and we see that the int corresponds to the number of occurrences in our string of chars.


4.4 def combineTrees(trees: SortedSet[HuffTree])

We still don't have a way to create CodeTrees (we're almos there!) so we use tree1 and tree2 from our previous test, for this test, as follows:


```

val y = Node(9,Node(7,Leaf('c',3),Leaf('b',4)),Leaf('c',2))
val z = Node(9,Node(7,Leaf('c',3),Leaf('b',4)),Leaf('c',2))
val a = Node(4,Node(2,Leaf('c',1),Leaf('b',1)),Leaf('y',2))
val tree1 = Huffman.combine(y,z)
val tree2 = Huffman.combine(a,a)
println(tree1)
println("")
println(tree2)
println("")
val t = SortedSet[HuffTree]() + tree1 + tree2
println("combineTrees:")
println(combineTrees(t))
println("")

```



```

Søren@Søren /cygdrive/c/Datalogi/Programmeringssprog/Projekt2/project_handout_a
ss2
$ scala HuffTests.scala
Node(18,Node(9,Node(7,Leaf(c,3),Leaf(b,4)),Leaf(c,2)),Node(9,Node(7,Leaf(c,3),L
eaf(b,4)),Leaf(c,2)))

Node(8,Node(4,Node(2,Leaf(c,1),Leaf(b,1)),Leaf(y,2)),Node(4,Node(2,Leaf(c,1),L
eaf(b,1)),Leaf(y,2)))

combineTrees:
Node(26,Node(8,Node(4,Node(2,Leaf(c,1),Leaf(b,1)),Leaf(y,2)),Node(4,Node(2,Lea
f(c,1),Leaf(b,1)),Leaf(y,2))),Node(18,Node(9,Node(7,Leaf(c,3),Leaf(b,4)),Leaf(c,2)
),Node(9,Node(7,Leaf(c,3),Leaf(b,4)),Leaf(c,2))))

```

We get the correct result. The input list only has two elements, which is not that much, but it is still enough to see that we utilize both cases. We will see in later tests by extension that this method works for SortedSet with more than 2 elements.

4.5 def createCodeTree(chars: List[Char]): HuffTree

We create 3 different lists. The last one has only a single element, to test this case as well.

```

val buu = "abracadabra".toList
println(createCodeTree(buu))
println("")
val str = "ratata".toList
println(createCodeTree(str))
println("")
val str1 = "r".toList
println(createCodeTree(str1))

```

```
Søren@Sorene /cygdrive/c/Datalogi/Programmeringssprog/Projekt2/project_handout_a
ss2
$ scala HuffTests.scala
Node(11,Leaf(a,5),Node(6,Leaf(b,2),Node(4,Node(2,Leaf(d,1),Leaf(c,1)),Leaf(r,2))
))
Node(6,Node(3,Leaf(r,1),Leaf(t,2)),Leaf(a,3))
Leaf(r,1)
```

We see that the Huffman Trees correspond to the input strings. The Most often used letters gets the shortest encodings as wished, and the rarely used letters get the longer encodings. In the case of just a single element, we get just a leaf, with that element. It actually ought to be `Node(1,Leaf(r,1),Nil)`. So our implementaiton actually does not work for just the case of just one single character. Arguably this is okay, since such cases are not interesting from a Huffman encodings perspective. But, optimally our implementation ought to produce the correct output in this case, which it doesn't. By extension we see that **combineTrees** works on `SortedSet` with more than 2 elements, due to our implementation.

4.6 def convert(tree: HuffTree): CodeTable

We create a code tree, and feed it to convert. The result can be seen below.

```
val buu = "abracadabra".toList
    val tree1 = createCodeTree(buu)
    val test1 = convert(tree1)
    println(tree1)
    println("")
    println(test1)
```

```
Søren@Sorene /cygdrive/c/Datalogi/Programmeringssprog/Projekt2/project_handout_a
ss2
$ scala HuffTests.scala
Node(11,Leaf(a,5),Node(6,Leaf(b,2),Node(4,Node(2,Leaf(d,1),Leaf(c,1)),Leaf(r,2))
))
Map(a -> List(0), b -> List(1, 0), c -> List(1, 1, 0, 1), r -> List(1, 1, 1), d
-> List(1, 1, 0, 0))
```

The notation can be hard to read, but the reader can verify that the mappings obtained are correct. Initial tests revealed the need for a **reverse** in our implementation, and as can be seen it works.

4.7 def codeBits(table: CodeTable)(char: Char): List[Bit]

We try obtaining the bits for the encoding of 'r' and 'd' using the codetable from the previos test.

```
val buu = "abracadabra".toList
val tree1 = createCodeTree(buu)
```

```

val test1 = convert(tree1)
val code1 = codeBits(test1)('r')
val code2 = codeBits(test1)('d')
println("")
println(code1)
println("")
println(code2)

```

```

Søren@Sorene /cygdrive/c/Datalogi/Programmeringssprog/Projekt2/project_handout_a
ss2
$ scala HuffTests.scala
List(1, 1, 1)
List(1, 1, 0, 0)

```

If we compare this to the result from the test of **convert** we see that we obtain the correct results.

4.8 **encode(table:CodeTable)(text: List[Char]): List[Bit]**

We once again use the string "abracadabra", create a codetree using it, and then test **encode**, on the same string. "abracadabra".

```

val buu = "abracadabra".toList
val tree1 = createCodeTree(buu)
val enc1 = encode(tree1)(buu)
println(enc1)

```

```

Søren@Sorene /cygdrive/c/Datalogi/Programmeringssprog/Projekt2/project_handout_a
ss2
$ scala HuffTests.scala
List(0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0)

```

Comparing the result to the Mapping obtained in the test of **convert**, we see that the result here is correct as well.

4.9 **getLetter(tree: HuffTree, bits: List[Bit]): (Char, List[Bit])**

Here we compare the result of **encode** with the result of **getLetter**, on the same string as before.

```

val buu = "abracadabra".toList
val tree1 = createCodeTree(buu)
val enc1 = encode(tree1)(buu)
println(enc1)
println(getLetter(tree1, enc1))

```

```
Søren@Sorene /cygdrive/c/Datalogi/Programmeringssprog/Projekt2/project_handout_a
ss2
$ scala HuffTests.scala
List(0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0)
(a,List(1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0))
```

As can be seen, the result is a tuple, with the first character we obtain, namely the 'a', as well as the remaining list, which can be seen to correspond to the previous result, with the leading 0 now decoded into the letter a. By extension, this function will be tested further in our last test of..

4.10 def decode(tree: HuffTree)(bits: List[Bit]): List[Char]

We create a codetree, encode the string from which it was made, print the codetable for readability, and then decode the list of bits that we encoded, to see whether the result is the same, using the supplied **edTest**

```
val str = "cocoon".toList
val tree = createCodeTree(str)
val encoded = encode(tree)(str)
val decoded = decode(tree)(encoded)
val codeta = convert(tree)
println(str)
println(tree)
println(codeta)
println(encoded)
println(edTest(str))

def edTest(str: List[Char]) = {
  val tree = createCodeTree(str)
  val encoded = encode(tree)(str)
  str == decode(tree)(encoded)
}
```

```
Søren@Sorene /cygdrive/c/Datalogi/Programmeringssprog/Projekt2/project_handout_a
ss2
$ scala HuffTests.scala
List(c, o, c, o, o, n)
Node(6,Node(3,Leaf(n,1),Leaf(c,2)),Leaf(o,3))
Map(n -> List(0, 0), c -> List(0, 1), o -> List(1))
List(0, 1, 1, 0, 1, 1, 1, 0, 0)
true
```

We obtain a tree that corresponds to a HuffTree of the supplied list. We get the correct mapping of characters to their respective bit encodings. We get the correct encoding of the string that was used to create the codeTable. The test whether encoding and decoding of the string is the same, using **edTest** yields true. It seems as though everything has come together, and works.

As we saw in the test of createCodeTree, our implementation does not consider

(and work for) the case where a list is made up of the same character only. Consider the following test:

```
val str = "llllllllll".toList
val tree = createCodeTree(str)
val encoded = encode(tree)(str)
val decoded = decode(tree)(encoded)
val codeta = convert(tree)
println(str)
println(tree)
println(codeta)
println(encoded)
println(edTest(str))
```

```
Søren@Sorene /cygdrive/c/Datalogi/Programmeringssprog/Projekt2/project_handout_a
ss2
$ scala HuffTests.scala
List(1, 1, 1, 1, 1, 1, 1, 1, 1)
Leaf(1,9)
Map(1 -> List())
List()
false
```

Due to the implementation of **combineTrees**, and by extension **createCodeTree** and **convert**, a single element in the inputstring results in, as can be seen Leaf(1,9). The correct answer would be Node(9,Leaf(1,9),Nil). Our implementation does not consider this - it is an easy fix however. One could insert standard characters to ensure that it would always work, or test for this specific case. This was not done, however, due to the fact that a Huffman Encoding of just a single element is wildly uninteresting. It is not hard to make the most efficient encoding of just a single element.

We have a similar result if our string is just the empty string. Testing with this, results in a NoSuchElementException, since we eventually end up with an empty map.

```
Søren@Sorene /cygdrive/c/Datalogi/Programmeringssprog/Projekt2/project_handout_a
ss2
$ scala HuffTests.scala
java.util.NoSuchElementException: empty map
    at scala.collection.immutable.RedBlackTree$.smallest(RedBlackTree.scala:
65)
```

4.11 Test for clarity

Here we've chosen the input of "abbcccddeeeeee" to have an increasing frequency, so the resulting tree and encoding/decoding is easier to follow for the reader.

```

Søren@Sorene /cygdrive/c/Datalogi/Programmeringssprog/Projekt2/project_handout_a
ss2
$ scala HuffTests.scala
List(a, b, b, c, c, c, d, d, d, d, e, e, e, e, e)

Node(15,Node(6,Node(3,Leaf(a,1),Leaf(b,2)),Leaf(c,3)),Node(9,Leaf(d,4),Leaf(e,5)
))

Map(e -> List(1, 1), a -> List(0, 0, 0), b -> List(0, 0, 1), c -> List(0, 1), d
-> List(1, 0))

List(0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1)

true

```

Here one can see that the more frequent characters get the shorter encodings, and the less frequent get the longer encodings. No encoding is the prefix of another.

4.12 Test on alphabet

We've seen tests on simple words having chars with somewhat high frequencies. We should test the opposite! What better input than a string consisting of the entire alphabet "abcdefghijklmnopqrstuvwxyz" (disclaimer: this might not be the correct listing of the alphabet). The HuffTests.scala is the same as in the test of decode, just with our new string. The result is as follows:

```

Søren@Sorene /cygdrive/c/Datalogi/Programmeringssprog/Projekt2/project_handout_a
ss2
$ scala HuffTests.scala
List(a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, x, z, y)

Node(25,Node(9,Node(4,Node(2,Leaf(j,1),Leaf(n,1)),Node(2,Leaf(t,1),Leaf(y,1))),N
ode(5,Node(2,Leaf(d,1),Leaf(z,1)),Node(3,Leaf(e,1),Node(2,Leaf(x,1),Leaf(s,1))))
),Node(16,Node(8,Node(4,Node(2,Leaf(v,1),Leaf(i,1)),Node(2,Leaf(b,1),Leaf(q,1))
),Node(4,Node(2,Leaf(f,1),Leaf(u,1)),Node(2,Leaf(m,1),Leaf(a,1))))),Node(8,Node(4,
Node(2,Leaf(r,1),Leaf(h,1)),Node(2,Leaf(o,1),Leaf(k,1))),Node(4,Node(2,Leaf(l,1)
,Leaf(g,1)),Node(2,Leaf(c,1),Leaf(p,1))))))

Map(e -> List(0, 1, 1, 0), s -> List(0, 1, 1, 1, 1), x -> List(0, 1, 1, 1, 0), n
-> List(0, 0, 0, 1), j -> List(0, 0, 0, 0), y -> List(0, 0, 1, 1), t -> List(0,
0, 1, 0), u -> List(1, 0, 1, 0, 1), f -> List(1, 0, 1, 0, 0), a -> List(1, 0, 1
, 1, 1), m -> List(1, 0, 1, 1, 0), i -> List(1, 0, 0, 0, 1), v -> List(1, 0, 0,
0, 0), q -> List(1, 0, 0, 1, 1), b -> List(1, 0, 0, 1, 0), g -> List(1, 1, 1, 0,
1), l -> List(1, 1, 1, 0, 0), p -> List(1, 1, 1, 1, 1), c -> List(1, 1, 1, 1, 0
), h -> List(1, 1, 0, 0, 1), r -> List(1, 1, 0, 0, 0), k -> List(1, 1, 0, 1, 1),
o -> List(1, 1, 0, 1, 0), z -> List(0, 1, 0, 1), d -> List(0, 1, 0, 0))

List(1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0,
1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1,
1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0
, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0,
0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1)

true

```

All characters have a unique encoding, where none is a prefix of another, as de-

sired.

5 Conclusion

Using pattern matching, recursion, as well as mapping we have succesfully implemented Huffman encoding with a functional approach. We have seen through our tests that the implementation is succesfull (on meaningfull input), and in the implementation section it can be seen that the approach is a functional one. The test cases chosen where meant to portray cases of interest i.e. having them differ in some way related to Huffman Encoding.

One of the last test concluded that on input of the empty string, or a string with only the same elements repeated a number of times, our implementation does not work. Good code terminates on every input and gives the correct answer, so this is unfortunate.

We argue that since the result of these case are of no use in the first place, no reason was seen to account for these cases. A fix would be to explicitly test for these cases, and print some sort of error message to warn the user. But if you want to use the functionality of this implementation, or Huffman encoding just generally, the cases of just a single element or no elements at all in the inputstring are irrelevant. When implementing methods, you should account for all possible input, but the user also holds the responsibility of not inputting nonsense.

6 Appendix- Huffman.scala

```
import scala.collection.immutable.SortedSet

object Huffman {

  def ??? = throw new RuntimeException("Method not implemented...")

  /**
   * *****
   * Part 1: Basics
   */
  def weight(tree: HuffTree): Int = tree match {
    case Leaf(char, vaegt) => vaegt
    case Node(vaegt, left, right) => vaegt
  }

  //Combine two HuffTrees into a single HuffTree
  def combine(left: HuffTree, right: HuffTree): HuffTree =
    (Node((weight(left)+weight(right)), left, right))
}
```

```

/**
 * *****
 * Part 2: Generating the HuffTree
 */

//Generate a list of (char,frequency) pairs.
def frequencies(chars: List[Char]): List[(Char, Int)] = (
    chars.groupBy(w=>w).mapValues(_.size).toList    )

//We use 'SortedSet' in place of a PriorityQueue because Scala currently
//does not support an immutable implementation of PQ's.
//This is okay since no two trees are equivalent.
def orderedLeaves(freqs: List[(Char, Int)]): SortedSet[HuffTree] =
    SortedSet[HuffTree](freqs.map(Function.tupled(Leaf)): _*)

//Repeatedly combine the 2 smallest elements and reinsert in the SortedSet
//to produce the final HuffTree (du kan reduce/fold SortedSet)
//overvej en case til empty, selvom det ser ud til at virke uden
def combineTrees(trees: SortedSet[HuffTree]): HuffTree = trees match {
    case x if x.size < 2 => trees firstKey
    case x if x.size > 1 =>
        combineTrees((trees drop 2)+(combine(trees firstKey, (trees drop
        }

//Using 'frequencies', 'orderedLeaves', and 'combineTrees', create the HuffTree
def createCodeTree(chars: List[Char]): HuffTree = (
    combineTrees(orderedLeaves(frequencies(chars)))
)

/**
 * *****
 * Part 3: Encoding and Decoding
 */
type Bit = Int
val One = 1
val Zero = 0

//To speed up encoding, let's use a hash-table
type CodeTable = Map[Char, List[Bit]]

//Convert HuffTree recursively to a codetable
def convert(tree: HuffTree, acc: List[Bit] = Nil): CodeTable = tree match {
    case Leaf(char1,int) =>
        Map(char1 -> (acc reverse) )

```



```

        case Node(w, left, right) =>
            convert(left, 0 :: acc) ++ convert(right, 1 :: acc)
    }

//Given 'table', return the bit-representation of 'char'
def codeBits(table: CodeTable)(char: Char): List[Bit] = (
    table.getOrElse(char, Nil)
)

//Given a tree and an encoded bit-string, return the first character along with
//the remaining part of the bit-string
//this also works for a tree of just a single leaf, since then theres no encoding
def getLetter(tree: HuffTree, bits: List[Bit]): (Char, List[Bit]) = tree match {
    case Leaf(char, int) =>
        (char, bits)
    case Node(w, l, r) =>
        if (bits.head == 0)
            getLetter(l, bits.drop(1)) //read 0, go left, sub
        else
            getLetter(r, bits.drop(1))
}

//Use a CodeTable to encode a text
def encode(tree: HuffTree)(text: List[Char]): List[Bit] = {
    val blazeit = convert(tree)
    text.flatMap(x => codeBits(blazeit)(x))
}

//Using 'getLetter', decode 'bits' using a HuffTree
//brug getLetter indtil listen er tom. Pattern matching paa bits?
def decode(tree: HuffTree)(bits: List[Bit]): List[Char] = bits match {
    case Nil => Nil
    case _ =>
        val (char, remain) = getLetter(tree, bits) //getletter return
        char :: decode(tree)(remain) //sidste
}

}

```