Lab 5 – Instructions

This lab is about the use of tree structures in the context of coding and data compression. Specifically, we will look at a simple form of entropy coding called *Huffman coding*, which uses trees to represent the code. In this lab, we will construct these trees and use them to encode and decode data.

Logistics

The logistics of this lab are as before:

1. Download the skeleton project and unpack it.

2. cd into its top-level directory, `edaa40lab5`. Start the Leiningen REPL there.

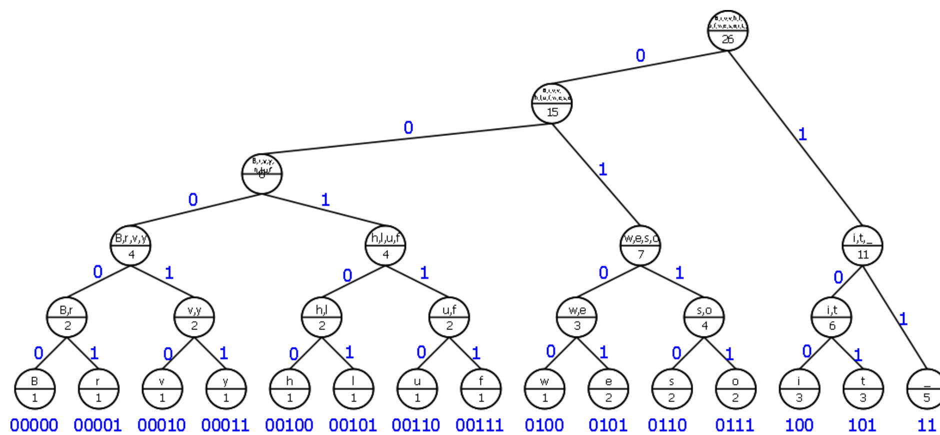
3. Have a look at the file

```
edaa40lab5/src/edaa40/lab5.clj
```

This is the file you are supposed to edit. As in previous labs, it contains commented-out incomplete code skeletons for you to provide implementations in. Also as before, it is a good idea to do this one step at a time, top to bottom, while making sure one set of tests pass before you proceed to the next part.

The package can be loaded, as you would expect, using:

```
(use 'edaa40.lab5 :reload)
```



Background

Suppose you wanted to encode a sequence of “symbols” – if that sequence is a text, the symbols might be the characters, if it is a binary file, they might be bytes and so on. The goal is to reduce the overall length of the encoded sequence, i.e. to compress it. The fundamental idea behind entropy codes such as Huffman codes is to represent different symbols by a varying number of bits depending on how frequently they occur in the sequence: very frequently occurring symbols are represented by shorter bit sequences, rare symbols are represented by longer sequences.

The discrete structure we care about in this lab is a *Huffman tree*, and it is used to represent the coding of symbols. This lab is about building a Huffman tree for a given sequence of symbols, and then using it to encode it into a sequence of bits, and to decode the original sequence of symbols from a tree and a sequence of bits.

This video gives some background, and also explains the basic technique for building the tree:

https://www.youtube.com/watch?v=0VEFJdjyXD0

Here are some additional resources:

https://www.siggraph.org/education/materials/HyperGraph/video/mpeg/mpegfaq/huffman_tutorial.html

https://en.wikipedia.org/wiki/Huffman_coding

Data structures

This is a quick description of the data structures used in this lab. You will find examples at the beginning of the program code, used for testing.

Huffman tree.

A Huffman tree is the central data structure of this lab. It includes, in its leaves, the symbols to be encoded, and also the frequencies, i.e. how often they occur in the original sequence. We only need the frequencies for building the tree – they are not used in encoding or decoding, so if you use the code for storing or loading files (not really part of this lab, but provided for you if you want to play with the code), you will find that the frequencies are not stored, and decoding still works perfectly fine.

The simplest tree is a *leaf*, which is represented by a Clojure map of the form

`{:kind :leaf, :frequency` *frequency*`, :value` *symbol*`}`

Here, *symbol* is the symbol represented by that leaf, and *frequency* is a number representing how often it occurs in the original sequence.

Any tree that is not a leaf is a *branch*, and it is also represented by a map, which in this case looks like this:

`{:kind :branch, :frequency` *frequency*`, :left` *child0*`, :right` *child1*`}`

In this case, *frequency* is the frequency of all the symbols in the left and right subtrees (the children) combined, in other words the sum of the frequency fields in the two children, which in turn are represented by *child0* and *child1*, respectively.

Huffman code.

A complete Huffman code represents an entire sequence of symbols, it consists of three parts, arranged in a map as follows:

`{:tree` *tree*`, :length` *length*`, :bits` *bits*`}`

In this case, *tree* is a Huffman tree as described above, *length* is an integer denoting the length of the original sequence (i.e. the number of symbols in it), and *bits* is a sequence of 0s and 1s that represent that sequence encoded according to the tree.

1. Building the tree

The first three functions you are supposed to implement deal with the construction of the Huffman tree from a sequence of symbols.

`(insert-into-queue` ***T Q*** `)`

This is a simple recursive insertion of a tree *T* into a sorted sequence (or queue) of trees *Q*. The elements of *Q* are sorted by their `:frequency` field, and *T* is supposed to be inserted into *Q* so that the result is again sorted by `:frequency`.

`(create-tree` ***Q*** `)`

Q is a non-empty, `:frequency`-sorted sequence of trees. If it contains only one element, that's the tree it is supposed to create. If it contains more than one element, `create-tree` takes the first two, forms a new branch with the first as the left and the second as the right child, and inserts the result into the remaining queue. Then it creates a tree from the resulting (now shorter) queue.

`(huffman-tree` ***S*** `)`

This creates a Huffman tree from a sequence of symbols *S*. It computes the frequencies of the symbols in the sequence, creates an initial queue that contains a leaf node for each symbol, sorts it, and then uses `create-tree` to produce the Huffman tree.

2. Encoding

The next couple of functions deal with encoding, i.e. turning a sequence of symbols into a string of bits according to a Huffman tree.

`(huffman-codes` ***T*** `)`

This functio returns a map from symbols to bit sequences, i.e. sequences of 0s and 1s. Each symbol in the tree *T* is mapped to the sequence of bits that encode it. Whichever way one chooses to do this, you might want to define one or two helper functions in order to implement this one.

`(huffman-encode` ***S*** `)`

This function computes the complete Huffman code (as described above under *Data structures*) from a sequence of symbols *S*.

3. Decoding

This function handles the decoding of a single symbol. It is called in a loop until all the symbols in the original sequence have been restored.

`(decode-symbol` ***T bits*****`)`**

T is a Huffman tree and bits a sequence of bits. This function returns a map with the following fields:

`{:value` *symbol*`, :remaining-bits` *rbits*`}`

Here, *symbol* is the next symbol encoded in the bit sequence, and *rbits* are the remaining bits not used to decode it.

Working with files

If you want to try your compression algorithm on different kinds of files, you can use the functions `huffman-compress` and `huffman-decompress` at the bottom of the lab5.clj file.