# Practical 1

This practical is worth 50% of the coursework credits for this module. Its due date is Friday 7th of October 2016, at 21:00 (i.e. end of week 4). The usual penalties for lateness apply, namely Scheme B, 1 mark per 8 hour period or part thereof.

The purpose of this practical is to gain experience with a widely used compression algorithm.

## Adaptive Huffman Coding

In the lectures we discussed Huffman coding, under the assumption that the information source is fixed. By this assumption, a fixed code tree can be used, which we may assume sender and received have communicated preceding actual transfer of source messages.

In many applications, however, the probability distribution of source symbols is not known in advance, or may vary widely between one source message and the next. One way to deal with this is to estimate probabilities using relative frequencies in any given source message. The receiver would have to know these frequencies as well, in order to construct the same code tree; this can be achieved by sending a 'code book' along with the actual coded message. This is rather cumbersome however and would require two passes over the source message.

A better approach is known as *adaptive Huffman coding.* The intuition is that we continuously monitor the relative frequencies of source symbols seen so far, and restructure the code tree whenever it no longer reflects a Huffman code for the currently estimated probability distribution. Moreover, we omit potential source symbols from consideration until they are seen for the first time. Initially, there is only one (artificial) source symbol in the code tree, which we call NYT ('not yet transmitted'). Whenever we encounter a new source symbol, we transmit the code word for NYT followed by the uncompressed representation of that source symbol.

Decoding is not much different from encoding. The same behaviour needs to be mimicked, so that encoder and decoder have the same code tree at the same stage of processing of the message from beginning to end.

## Implementation

Implement adaptive Huffman encoding and decoding. There are some variants of adaptive Huffman coding in the literature, and you can choose any reasonable one. As a fall-back solution you could reconstruct the entire code tree after reading each token from the source message; that is far from elegant and is slow, but it doesn't harm the compression. With this fall-back solution, the marks you can gain are restricted (see below).

It is better however to make only incremental changes to the code tree when needed. You may want to read up on adaptive Huffman coding (developed by Faller, Gallagher, Knuth, Vitter and others). The intuition is that each node of the code tree is annotated with a 'weight', which is the sum of the frequencies of all leaves below it. Moreover, the nodes of the code tree are ordered in a list; let us say the first element in the list has index #0, the second index #1, etc. The ordering in the list satisfies three requirements:

1. a node with a higher weight has a higher index,
2. siblings have consecutive indices,
3. a non-leaf always has a higher index than its children.

The last two requirements are called the *sibling property.* The last requirement seems to follow from the first two, but is needed because of the artificial NYT symbol, which is assumed to have weight 0 (in contrast to all other nodes, which have non-zero weight).

If any of the above three properties risks being broken by incrementing the weights for the next token from the source message, we need to swap two subtrees of the code tree. Concretely, before we increment the weight of a node, we check whether another node exists (excluding the parent) with the same weight and a higher index. If so, there might be several such nodes, and among them we take the one with the highest index. This node is swapped with the one whose weight had to be incremented. The swap involves the labels of the nodes and the subtrees, except for the indices, which remain in the same place in the tree. After incrementing the weight by one, we continue the process from the parent, until we arrive in the root. Then we read the next token.

If a symbol is seen for the first time, we place a new internal node in place of the NYT leaf, whose first child is the NYT leaf, and the second child is the leaf for the new symbol. The indices of all nodes are updated, and we increment the weights of ancestors as before, possibly involving swaps. A very good description of adaptive Huffman coding is found in Sayood, Section 3.4 (and various sources freely available on the web). In the appendix, we also give an example.

The preferred implementation language is Java. If desired, C, C++ and Python are also acceptable, but Haskell is not. In any case, you should make sure the application runs on the lab machines.

Please make it easy for me to compile and run the code. I should not have to use an IDE such as Eclipse. For Java, use of `ant` is strongly encouraged; for C or C++, a Makefile would be appropriate. Packing the compiled code in a JAR file causes complications, as I would have to run it in a 'sandbox' mode for security reasons, which makes it impossible to use file I/O.

There is no need to spend too much time on the user interface, and plain text output may be as good as use of GUIs. The meaning of the output (reflecting how much compression is achieved) should be self-evident however.

There is also no great need to optimise the implementation for speed. It is more important to consider how much compression is achieved.

## Experimentation

Try out your code on some sample files. An obvious choices would be text files, but you could also consider audio or images; note that many image formats already do compression. For symbols you could take bytes (or characters), but it could also be interesting to look at sequences of 4 bits, or 8 bits, or 16 bits, ... You get the most benefit out of this practical if you experiment, observe and reflect on what you have observed.

## Report

Your report should have the following sections as minimum:

- A short description of how you realised adaptive Huffman coding. Acknowledge sources you consulted as usual.

- Description of the experiments done with the implementation.
- Reflection on the outcome.

There is no minimum or maximum to the length of the report. It is quite conceivable that a highly informative report, with interesting observations, could be as short as 2 pages. If you want to present your results using graphs, the report would typically be longer.

## Submission

- Submit source code, e.g. `.java` files.
- Add instructions for use, i.e. for compiling **and** for running, preferably in a README file.
- Submit the report in PDF.

## Extras

For the highest marks, some additional effort is required, which could include:

- A detailed description of the algorithm, and reflection on the various subtle differences between variants of adaptive Huffman coding.
- Further analysis of different kinds of source data.
- Implementation of other compression algorithms, and their adaptive forms, and experimentation with these and analysis.

## Assessment

The marks allocated for this practical will be in line with the scheme given in the school's student handbook.

For marks above 17, required are both a correct, well-designed implementation (in good programming style, with tidy layout, code commenting, etc.) and a well-written report with meaningful experiments and analysis. For higher marks, at least one of the extras is required.

With the fall-back implementation of adaptive Huffman coding whereby a fresh code tree is computed for each token from the source message, no more than 14 marks can be gained.

Decoding is less interesting than encoding, and is strictly speaking not needed in order to measure how much compression is achieved. If you run out of time, you may omit implementation of decoding, but this restricts the marks to 16 (or 13 in combination with the fall-back implementation of encoding).

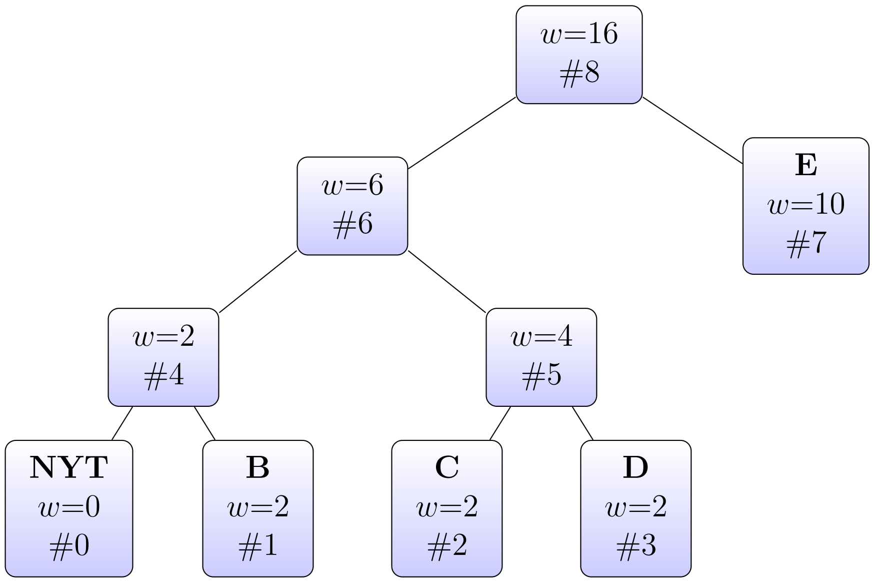
## Pointers

- Marking
  `http://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/feedback.html#Mark_Descriptors`
- Lateness
  `http://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/assessment.html#lateness-penalties`
- Good Academic Practice
  `https://www.st-andrews.ac.uk/students/rules/academicpractice/`

## Example

In the below diagrams, the bold face letters in the leaf nodes are the source symbols. The weight of a node is indicated by $w$; the weight of NYT is always 0. The number in the bottom line in each node, following #, indicates the index in the list.

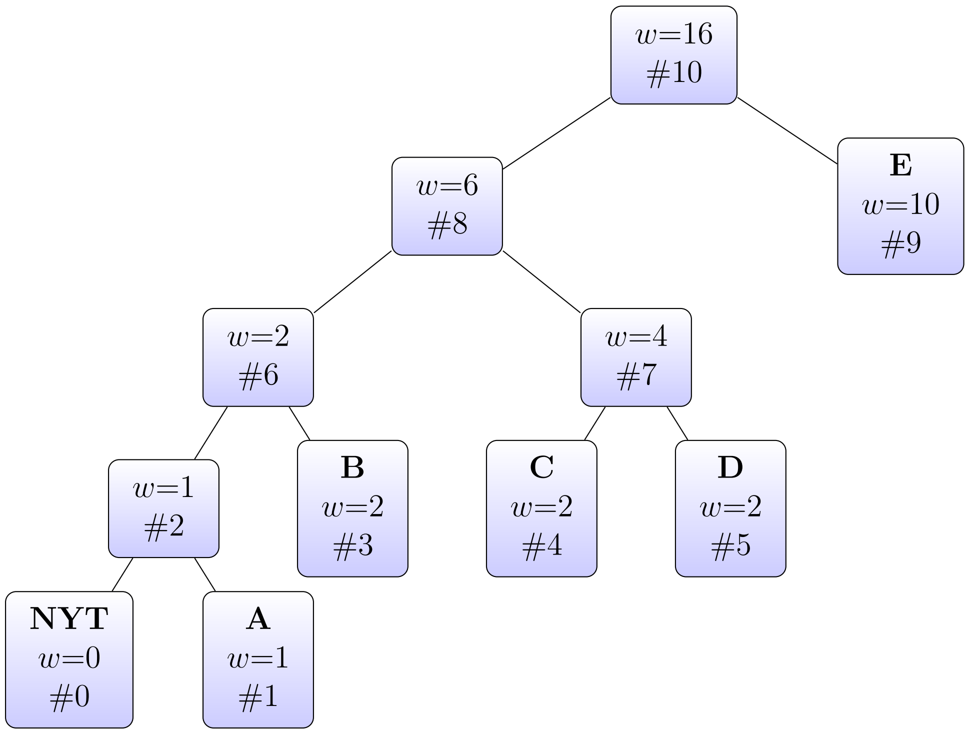
Suppose after a number of steps we have this.



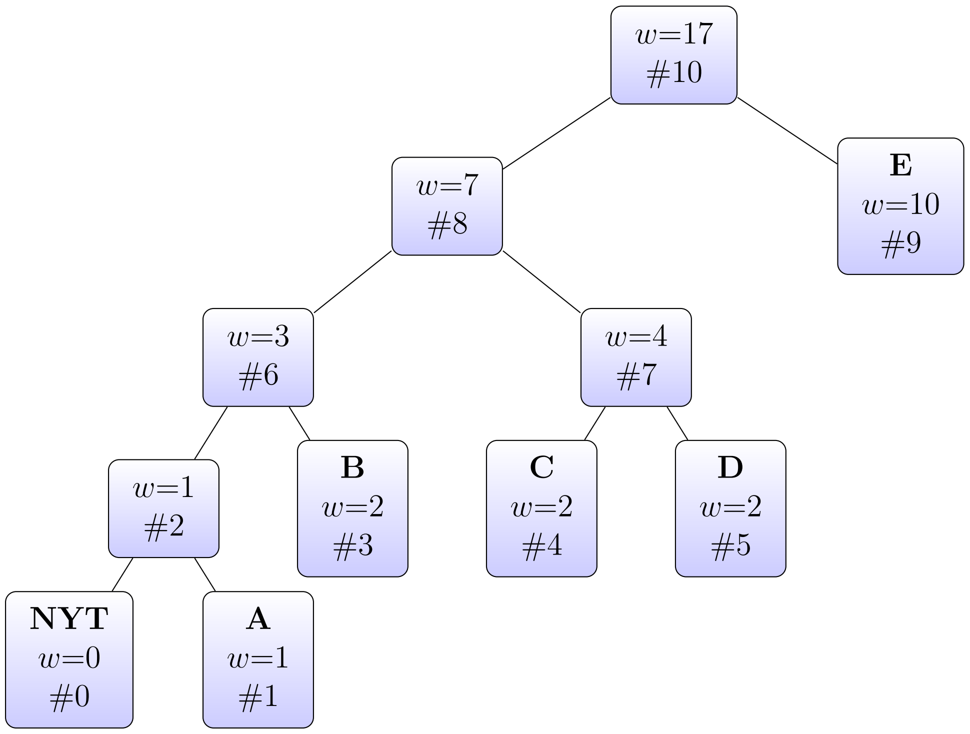
Verify:

- Each non-leaf has exactly two children.
- The weight of each non-leaf is the sum of the weights of the two children.
- Higher weighted nodes have higher indices.
- A pair of siblings have consecutive indices $2j$ and $2j + 1$, in this order, some $j$.
- The parent has a higher index than either of its children.
- The NYT node has index #0.

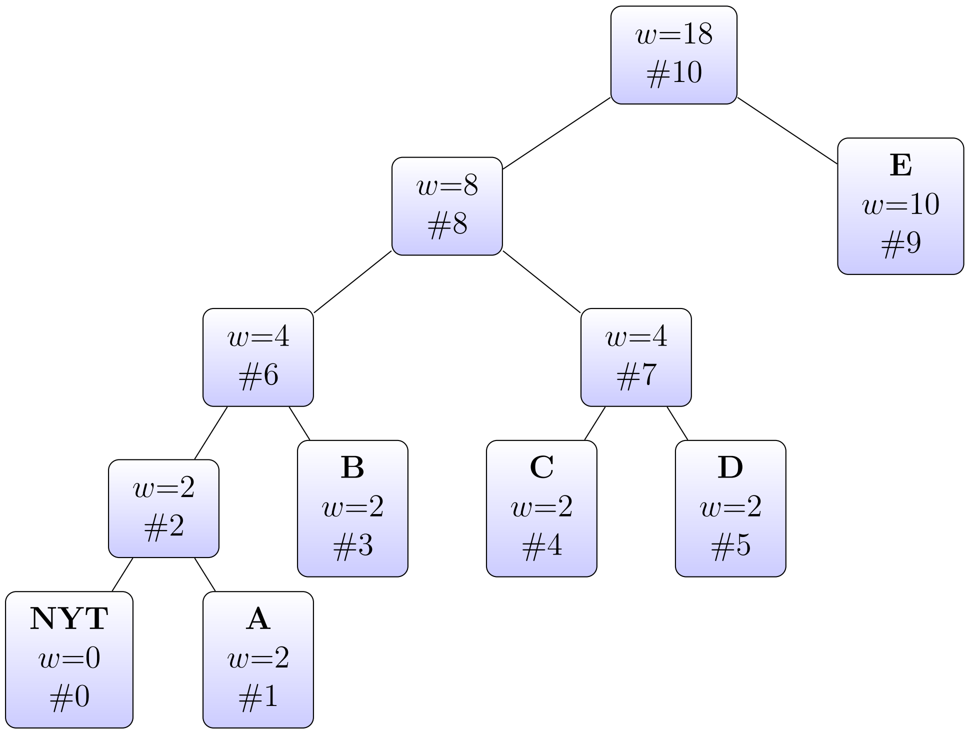
Suppose we see **A** for the first time. We replace the NYT node by a non-leaf node with weight 1, whose first child is the NYT node and the second is the new leaf for **A**, again with weight 1. The list has two more members, so we need to update the indices. To be continued below.



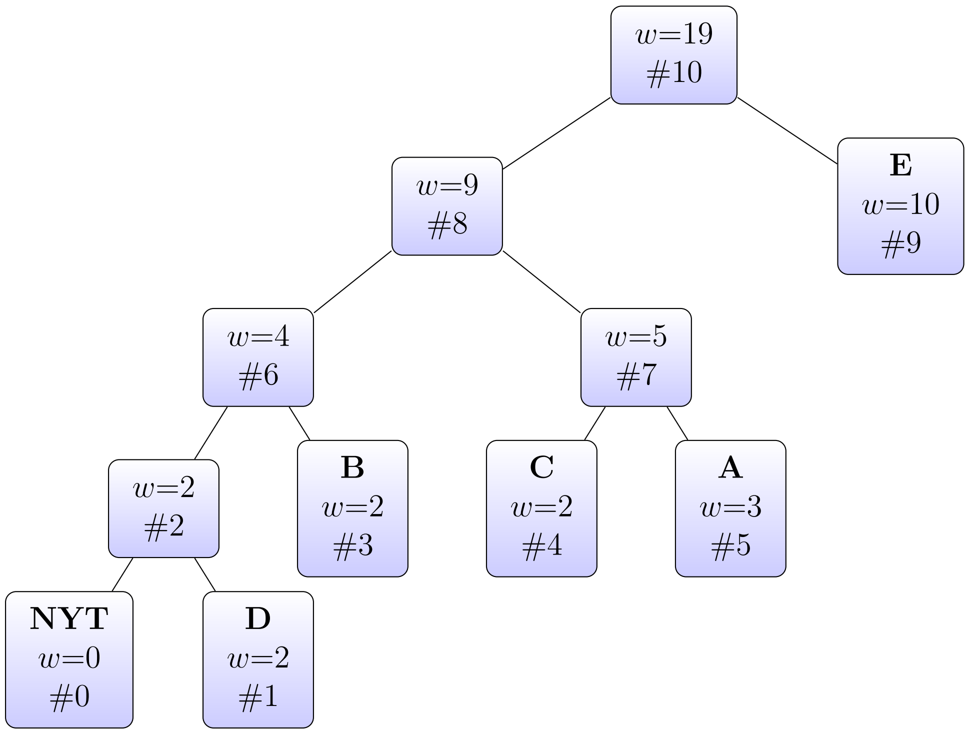
We have some unfinished business, as the extra symbol has not yet been counted in the parent of the new node #2, which is #6. First, let us check whether there is another node with $w = 2$ that has a higher index than #6. This is not the case, so in #6 we can update $w = 2$ to $w = 3$ and turn to the parent, node #8. Is there another node with $w = 6$ that has a higher index than #8. No, so we update $w = 6$ to $w = 7$ and turn to the parent, etc., and then we can stop because #10 is the root. Note all the invariants (the bullet points above) are again satisfied.



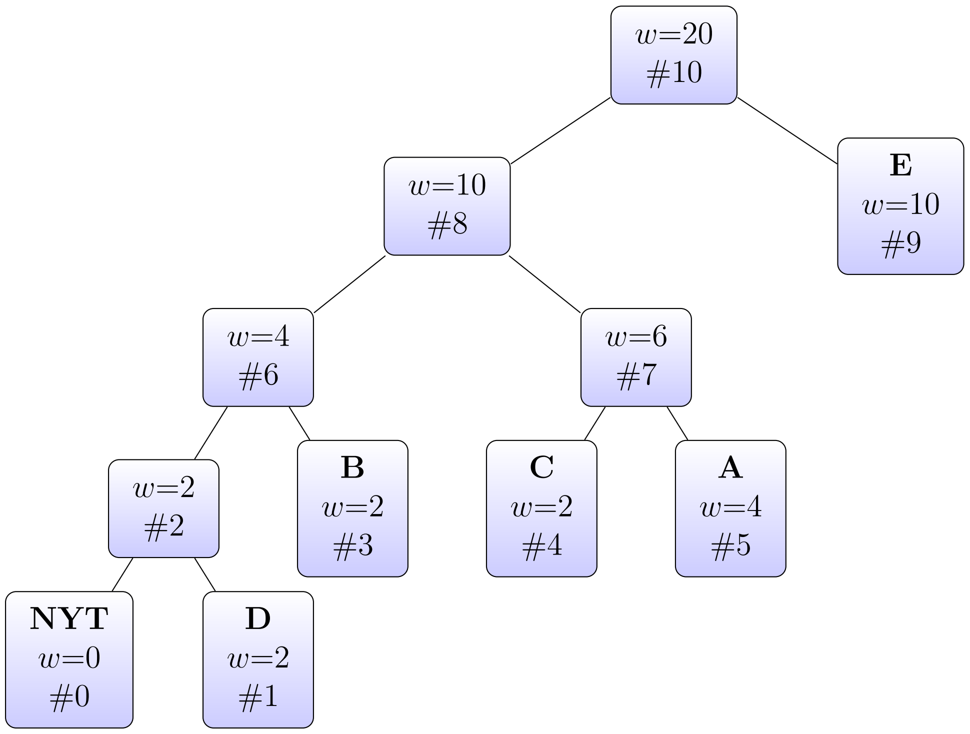
Suppose another **A** is read. We again propagate upwards, starting from #1. We verify there is no other node with $w = 1$ that has higher index than #1 (apart from #2, which is the parent of #1, and is therefore left out of consideration). We verify there is no other node with $w = 1$ that has higher index than #2. We verify there is no other node with $w = 3$ that has higher index than #6, etc., until we get to the root. Again, all the invariants are satisfied.



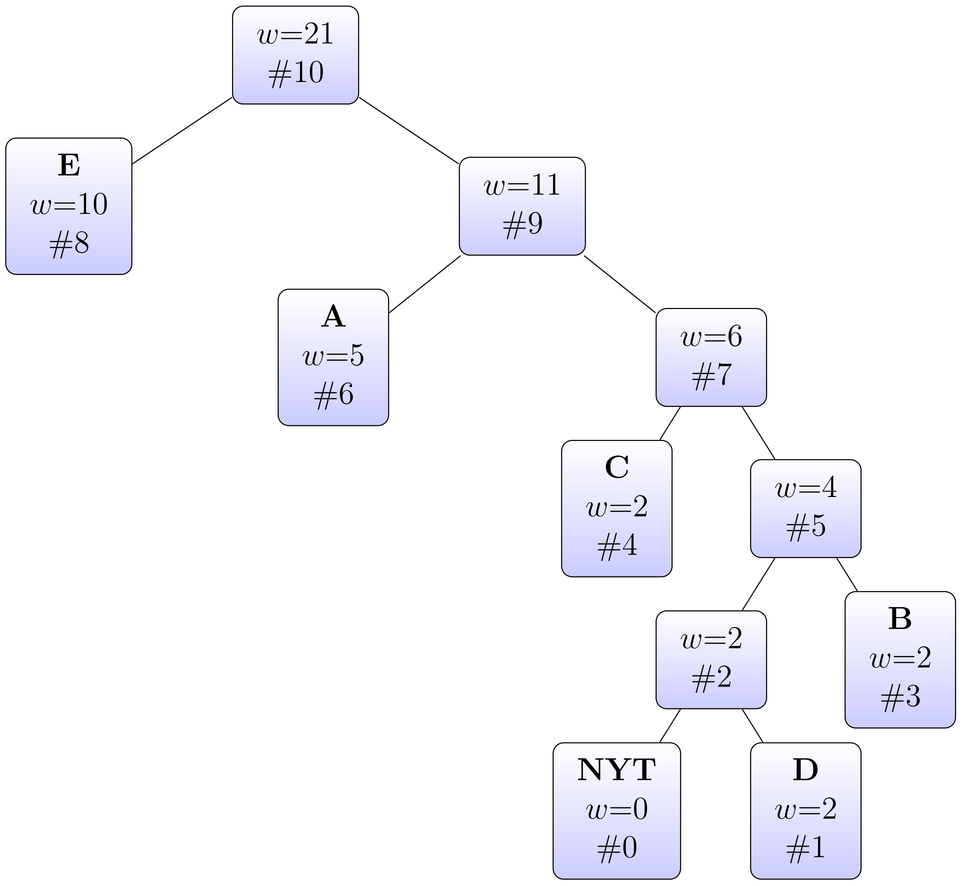
Suppose another **A** is read. This time, we find there is another node with $w = 2$ that has higher index than #1, namely #5. (If there were several, we would take the one with the highest index.) This means we need to swap the contents of #1 and #5 (but we keep the indices of the nodes). We then propagate upwards as before, verifying that there is no other node with $w = 4$ that has higher index than #7, etc.



Suppose another **A** is read. No swaps are needed this time.



If the next symbol is again **A**, we find there is another node with $w = 4$ that has a higher index than #5, namely #6, so we need to swap. Note that subtrees are involved as well in swaps. Continuing from the parent of #6, we find there is another node with $w = 10$ that has a higher index than #8, namely #9, so we need to swap again.



Note once more the invariants are satisfied.