CSE100 Algorithm Design and Analysis — Lab #9
Spring semester 2013 — Miguel Á. Carreira-Perpiñán

Huffman Codes

Description Suppose that we have to store a sequence of symbols (a file) efficiently, namely we want to minimize the amount of memory needed. For the sake of simplicity let us assume that the symbols are restricted to the first 6 letters of the alphabet. For example, let us assume that the frequency of different symbols that you have to store are the following:

symbol	frequency
A	1000
B	150
C	200
D	800
E	300
F	50
Total	2500

As we have to store 6 different symbols, the obvious way is to encode each of them in 3 bits, as with 3 bits it is possible to encode 2^3 different symbols. With this encoding, we need $2500 \times 3 = 7500$ bits to store the above symbols. A different way to address the problem is the following. Instead of assigning to each symbol a code with the same length (i.e., number of bits), we assign shorter codes to symbols that are more frequent, and longer codes to symbols that are less frequent. One possible encoding according to this sequence is the following

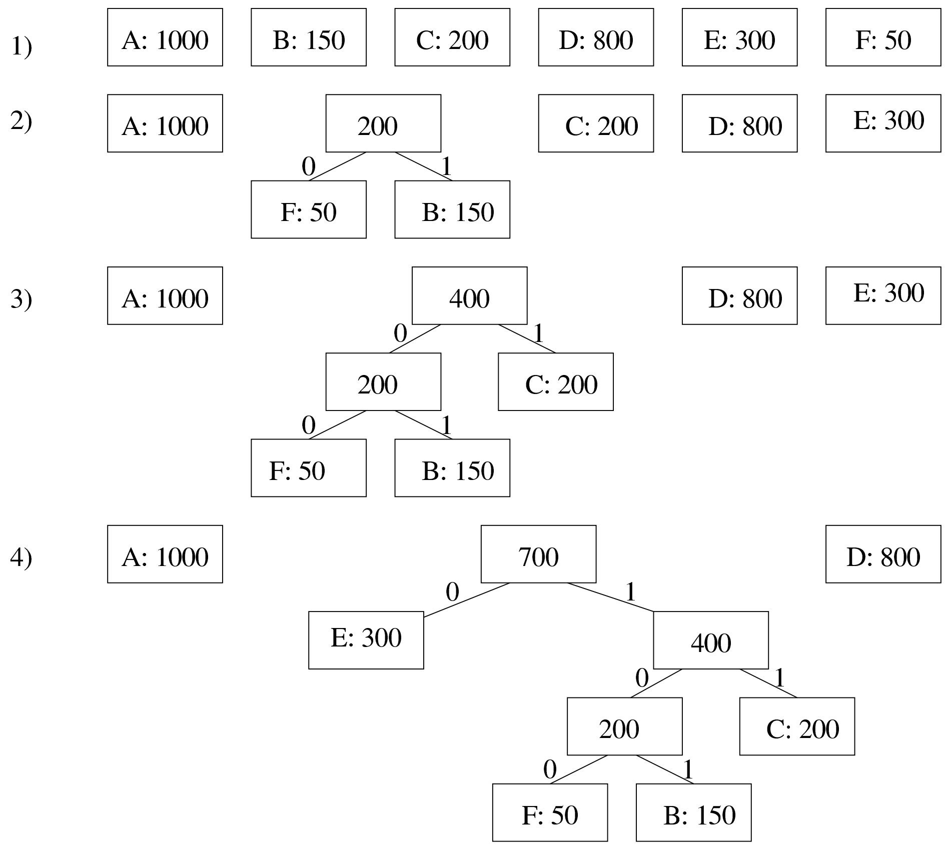
symbol	encoding
A	0
B	10101
C	1011
D	11
E	100
F	10100

According to this encoding the number of required bits is:

$$1000 \times 1 + 150 \times 5 + 200 \times 4 + 800 \times 2 + 300 \times 3 + 50 \times 5 = 5300.$$

This idea is at the basis of the programs used to compress files. First they analyze the input, then they choose the codes, and they recode the input accordingly to the determined codes.

While this idea brings benefits in terms of the space requirements, using variable length codes presents some problems. Once we have coded a file according to a variable length code, we must also be able to decode it in the original format (i.e., once we have compressed the file, we want to able to decompress it). The encoding works only if the codes assigned to different characters are such that no code is a prefix of any other code. If this property does not hold, there is a problem of ambiguity when trying to decompress the sequence. You can prove that in the depicted example no code is a prefix of any other code. For example: no code starts with 0 except from the code of A. So while decompressing the file, if we find a symbol whose code starts with 0, we know it's A. If we find character whose code starts with 11, we know it's D. It can't be any other symbol, as no code starts with 11 other than D's code. And so on. How do we assign codes? This is done through a greedy algorithm. We assign the shortest code to the most frequent character, the second longest one to the second most frequent character, and so on. The figure illustrates the first few stages of the algorithm.



Given N characters with their respective frequencies, the algorithm initially builds N trees, each one consisting just of a single node (step 1, in the figure). Then, iteratively, it joins together the trees whose roots have the lowest frequencies (steps 2, 3, etc. in the figure). The tree with the lowest root frequency becomes the left child and the tree with the second-lowest root frequency becomes the right child. Left children are associated with the bit 0, right children with the bit 1. Internal nodes (i.e., root nodes created) can be thought of as dummy nodes storing a fictitious character (which does not appear in our sequence). This procedure is iterated until there is just one tree. At this point, in order to know the code associated with one symbol you simply need to concatenate the 0s and 1s you encounter while moving from the root down to the symbol.

Note that the greedy strategy is applied in the reverse way. Symbols with low frequencies end up down in the tree (i.e., they are associated with long codes), while nodes with high frequencies are near the root (i.e., they are assigned short codes).

Questions and input structure

1. You are given the class BST, defined in the provided files `BST.cpp` and `BST.h`, along with a `Makefile`. Read the sequence of symbols from the keyboard (you may assume that all the symbols are in the range $A, \ldots, F$). The input sequence terminates when you read the symbol Z (which should not be processed). At that point build the corresponding tree and print it out using the provided function `PrintLetterFreqs`.
   Testcase number to use with `grade_me`: 14.

2. **Bonus Question**. After you have built the tree, print out the binary code assigned to each character. Print the characters in order, one per line. See the example for more information.
   Testcase number to use with `grade_me`: 15.

Using the BST class provided will save you time, but you don't have to use it if you prefer not to. The test cases have been built so that while building trees it never happens that the same frequency appears twice. Then, the decision about which tree goes to the left and which one goes to the right is always straightforward.

Examples of input and output

Input for Test Case 14

```
AAAAAAADCCBBBEEEFFFFFFFFFFFAAAAAAAAEBEZ
```

Output for Test Case 14

```
A 15
F 11
E 5
D 1
C 2
B 4
```

Input for Test Case 15

```
AAAAAAADCCBBBEEEFFFFFFFFFFFAAAAAAAAEBEZ
```

Output for Test Case 15

```
A 0
B 1111
C 11101
D 11100
E 110
F 10
```

Your solutions Before leaving the lab, submit a zipped tar archive of your program through the assignments page of UCMCROPS. Please use your UCMNetID as the filename for the zipped tar archive. Be careful since UCMCROPS strictly enforces the assignment deadlines (deadlines will be every lab date at either 4:20pm or 7:20pm depending on your lab session.).