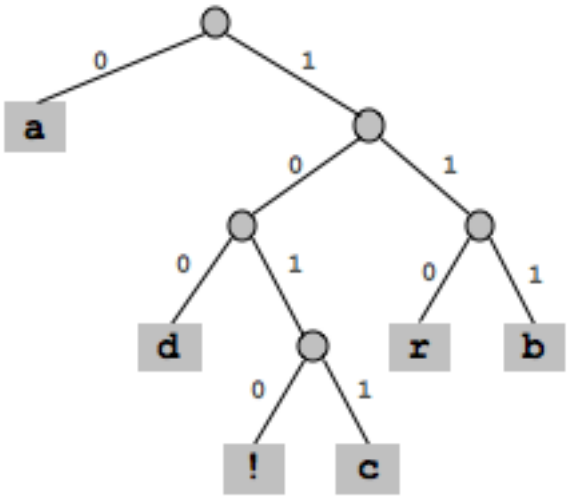
Data structures
Huffman coding

1 Introduction

Programming languages represent text as sequences of characters. Each character is encoded to a binary representation using some sort of coding scheme. The most common one is the ASCII encoding. In ASCII, every character is encoded using 8 bits. Since there are 256 different values that can be encoded with 8 bits, there are potentially 256 different characters in the ASCII alphabet.

ASCII is a general purpose encoding scheme. If we have knowledge about the relative frequency of each character in a text, we can come up with a more efficient encoding that uses less bits for common characters and more bits for relatively rare characters.

This is exactly what Huffman coding does. Huffman coding first counts how many times each character occurs in a text and uses this information to build a Huffman tree as shown in figure 1. In this tree, each character is stored in a leaf node. To encode a character we descend the tree until we end up at the correct leaf node. The path from root to leaf then determines the encoding of the character: each time we follow the left child, we add "0" to the representation, each time we follow the right child, we add "1" to the representation.



char	encoding
a	0
b	111
c	1011
d	100
r	110
!	1010

Figure 1: An example Huffman tree and the corresponding encoding table.

To obtain an efficient encoding we have to make sure that the path to a common character is short relative to the path of a rare character. In the example you can see that “a” is encoded using just a single bit while “!” requires four bits.

To build the tree, we use a recursive procedure as shown in figure 2.

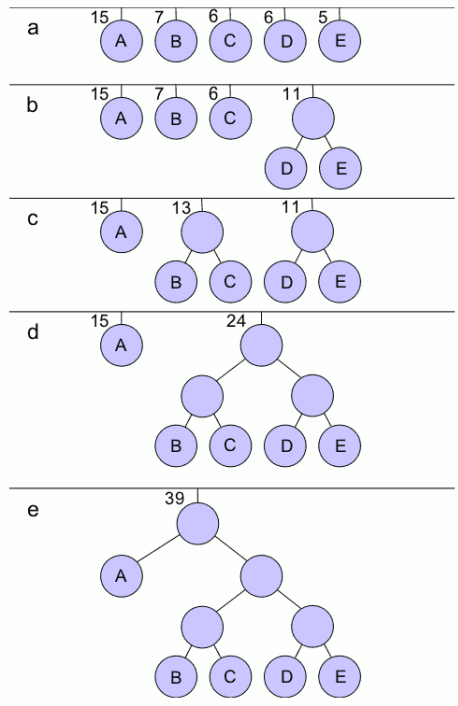


Figure 2: Building a Huffman tree.

First, we count how many times each character occurs in the text. We then make a new tree (just a single node) for each character. The node stores the character together with the times that character has occurred. In the example of figure 2, the character “a” has occurred 15 times in the text while “D” has occurred six times.

We then find the two nodes with the smallest counts (D and E in the example) and combine them into a larger binary tree, making them children of a new node with a weight corresponding to the sum of the weights of the children. For convenience, we can also store both characters into this new root node. In the figure above, this is visualized in step (b). The nodes corresponding to character “D” and “E” are now combined in a new tree. This new node has weight 11 and also stores both characters. This is needed later to descend the tree.

We keep repeating this procedure: find the two trees with the smallest total weight and merge them into a larger binary tree until we end up with a single tree. This is then the Huffman tree.

2 Assignment

- The startcode contains a basic implementation of a BinaryTree class that you can use to build the Huffman tree. You are free to add additional data attributes, constructors or methods to this class if needed.

- Implement the different methods in the HuffmanCoder class. This class uses the BinaryTree class to build a HuffmanTree. The encode and decode methods use this HuffmanTree to encode a text to a binary representation or to decode the binary representation back into the original text. Again, you are completely free to add additional methods, attributes or constructors to this class.
- Implement the toString() method of the BinaryTree to return a textual representation as shown below.
- An example output of the Main program for the sentence "Test Test Test test test test" is shown below. You can see that "e" is encoded using two bits while "T" uses three bits.

```
Test Test Test test test test
1010100 1100101 1110011 1110100 100000 1010100 1100101 1110011 1110100 100000 1010100
Ascii encoding requires 232 bits
seT t(29.0)
--0--> se(12.0)
    --0--> s(6.0)
    --1--> e(6.0)
--1--> T t(17.0)
    --0--> T (8.0)
        --0--> T(3.0)
        --1-->  (5.0)
    --1--> t(9.0)

100 01 00 11 101 100 01 00 11 101 100 01 00 11 101 11 01 00 11 101 11 01 00 11 101 11

Huffman encoding requires 66 bits
Decoded: Test Test Test test test test
```