David González Fabián

Huffman coding

Brief explanation about it

&

Code Explanation

***Index***

# 1. Huffman algorithm explanation (about the working of it)

# 2.Code

## Standard Template Libraries Used

## Main and How to input any text

## BuildTree () function created

## Encoding

## Decoding

## Structures

# Huffman code explanation

The Huffman algorithm is based in the Shannon-Fano algorithm, that creates a binary tree. For a list of symbols, it creates a list with the frequency of each symbol, and then sort the the symbols from left to right with the most common on the left.

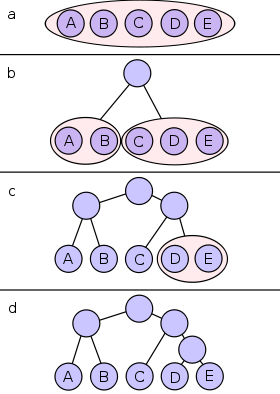
Then, divide the list in two parts were symbols in one part have the same (or close to the same) frecuency as the symbols on the other part. Then, a 0 bit is given to the simbols on the left and a 1 to the ones on the right. Then you divide each part again and give another bit to each symbol until every symbol is sorted . The most common symbol will then have a 00… code and the less common a 11…

Figure 1 : Shanno Fano Tree

This algorithm, doesn’t always generate the most optimal code, while the Huffman algorithm, working in a similar way, always does. Instead of creating the branches from root to leaves, the Huffman does it in the opposite way, from leaves to the root.

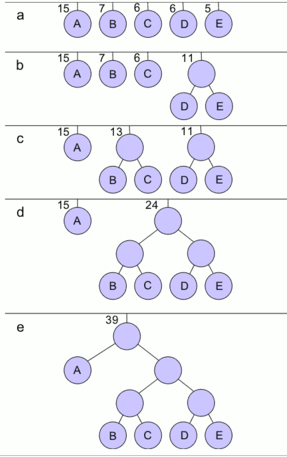
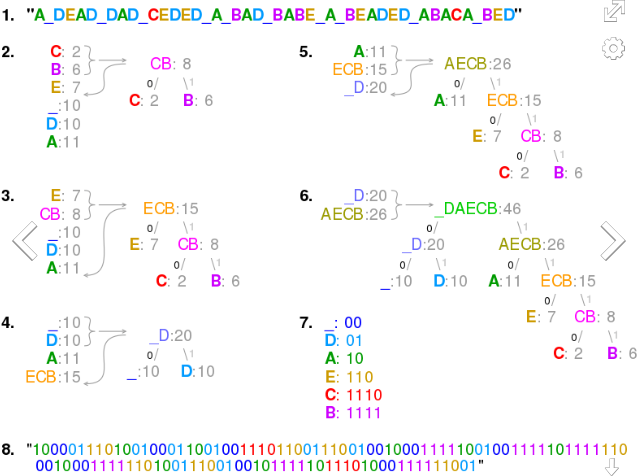
 The first step is to create a node (leaf) for each symbol and put the most common symbols in the first positions of a queue. While there is more than one node in the queue, you remove 2 nodes with the less probability and assing them a 0 and a 1 to each node removed. Create a new branch node with those 2 nodes in it, and put other node to the queue, repeat until the tree is completed

Figure 2 Huffman Tree

The Huffman code is a prefix code used in lossless data compression. This algorithm creates a table that encodes the source symbols using the binary tree explained before.

The most common caracters use fewer bits than the less common ones.

Here is a colorful example using an small alphabet of only 5 letters.



The step 7 is the code that is used to transcript, the space is the most common symbol given only a 00. Using ASCII code, for example, each letter will take 8 bits instead of the 2-4 it uses here. It could even be greedier if space was only one bit instead of 2.

This code is optimal for symbol by symbol coding if we know the input probability, but not optimal if the probability is unknown, were other coding methods are way optimal.

The worst case for this coding happens when the probability of the most likely symbol far exceeds 0.5 (half the probability) , making the upper limit inneficient

Code Explanation

# Standard Template Libraries

We will use some STL ( Standard Libraries Templates ) for our code.

The libraries we will need to use are **iostream,** for cin<< and cout>> in the input output terminal. It’s the one we will use to work around in sending messages in the terminal of the executable.

Other one is **queue**, (std::queue) that allows us to work in a First In is the First Out ( FIFO ) that will also allow us to measure the length of our queue or string , and use push and front that will move the elements around our string in the desired order to match the Huffman algorithm.

Double ended queues (deque) might be used if we want to work inserting and deleting a lot of items in the end ( if we want to use other sorting algorithm different from the bubble sorting for example), I couldn’t work with this library

The **string** library will allow us to use string class that will internally use an array of char to store the character. Also, allows to measure the length of the strings as using *###.length()* or *###.size()* as an int. In some other cases we can access a character of the string using *###.at(5)* were 5 is the 6th position of the character.

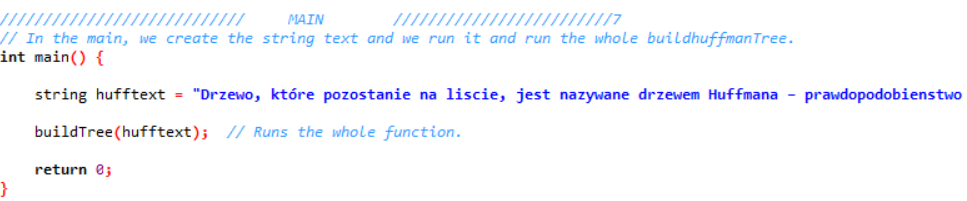
**Unordered\_map** is the last one, at first we could have using the library map, because they get stored in a hash table, so using the unordered\_map, we get the elements not stored in order, which is better and keeps us from using pointers or other ways of saving the data before calling the map.

Std unorderedmap is better when

You need to keep count of some data (Example – strings) and no ordering is required.

You need single element access i.e. no traversal. We use it in *pair.first, pair.second* to show the coding letters

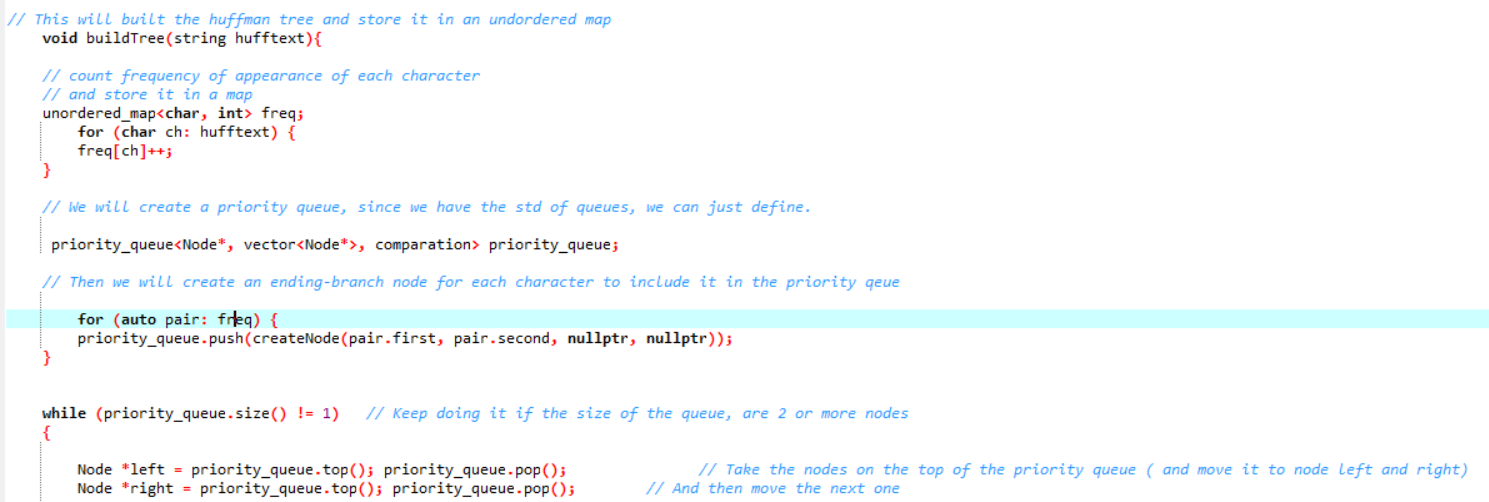
# Main () & How to input text

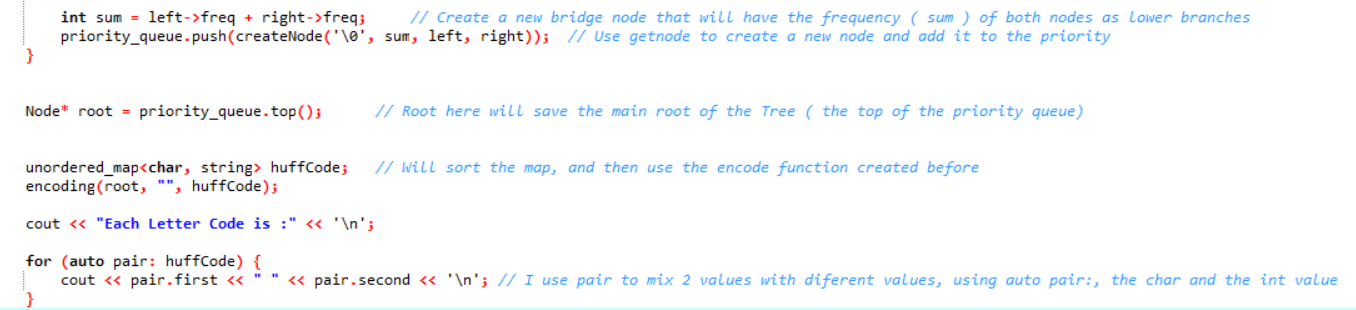


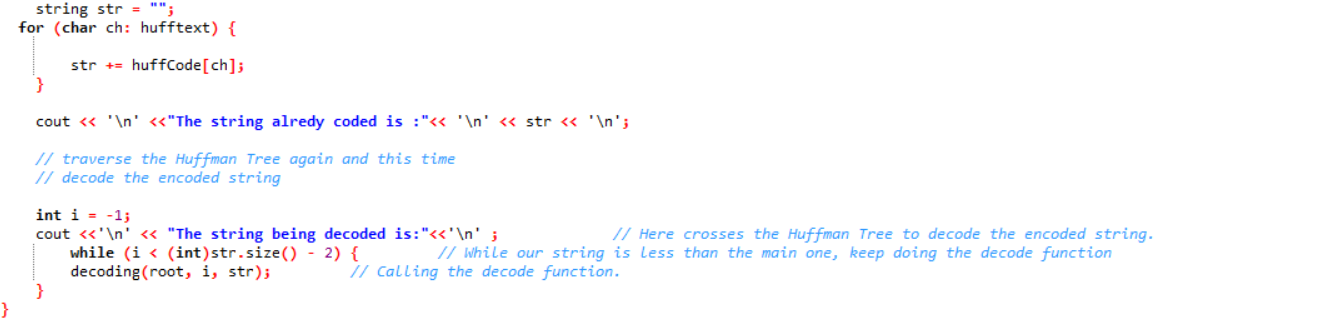
We just define the text we want to as a string, (string is defined below). If we want to open it from a .txt external file we should’ve created a script that runs this code in a terminal, opening this main and some .txt, however, I don’t know if this will be runned in a .bat of windows, or a script bash in linux, so I just made everything in one file making it easier to work in any terminal/computer.

Afterthat we run the buildTree function with the text string

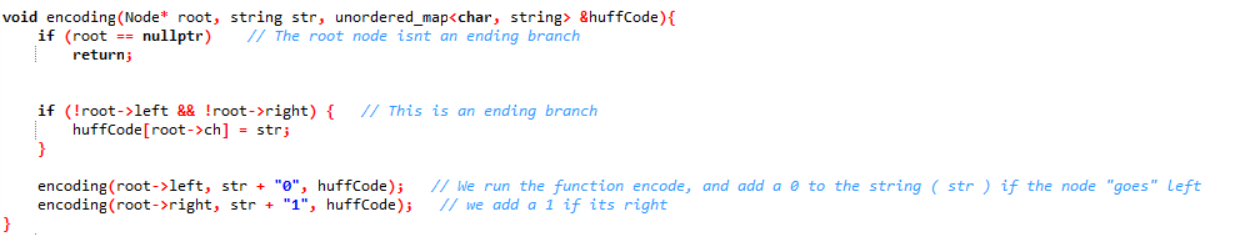
# Build Tree



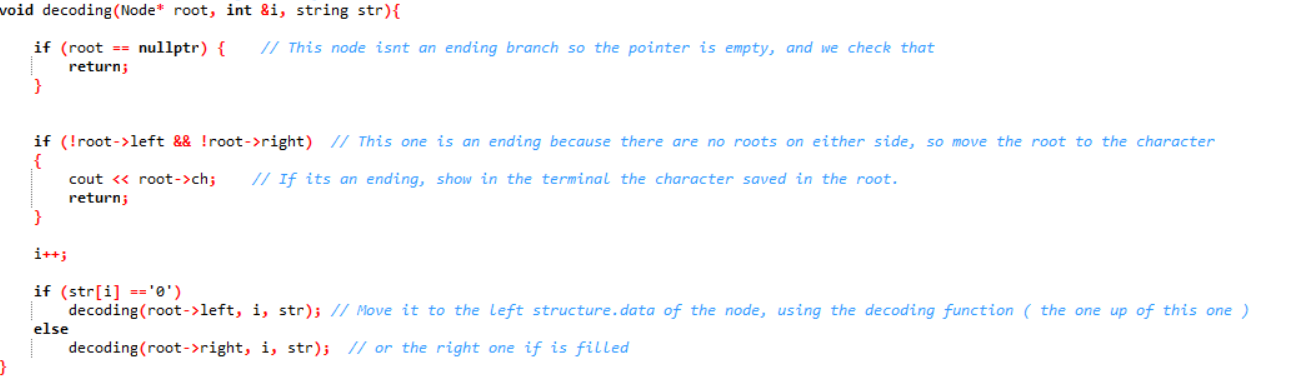




# Encoding Function



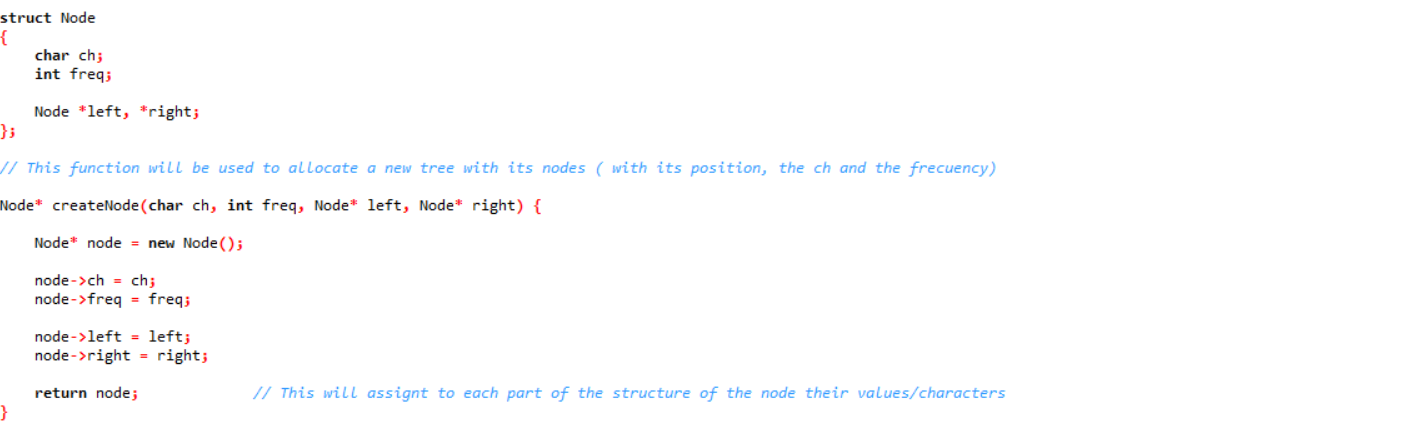
# Decoding Function



This both functions (voids) runs in the buildtree function. They both check the pointer in the root of each node, and if it isn’t empty they move the character to the string, and the coding one adds a 1 or a 0 to the huffCode function, while the decoding moves across and outs in the terminal de character stored in the root.

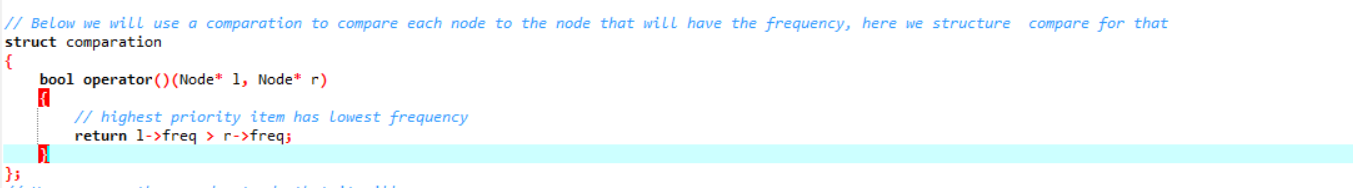
The order is a 0 to the most frecuent and a 1 to the less, but it could be otherwise and don’t affect the code size at all.

# Structures



We use the structure of the node, that will contain both the ch and the frecuency.

Also they will get sorted using left or right, like a tree wise.



The comparation is other structure ( couldn’t find any library std that could do something like this) will compare the nodes and return in despise the frecuency, it will be used in HuffCode