CSC 2001
Project 3: Huffman Coding Example

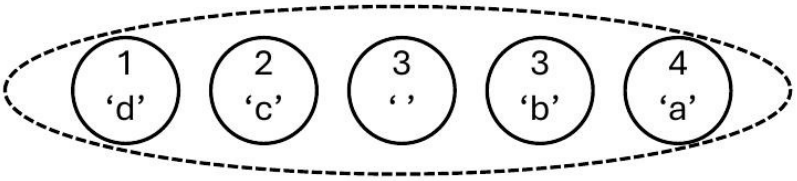
Suppose you have a text file with the contents:
abcd abc ab a

We want to turn each character into a code composed of ' 0 ' s and ' 1 ' s such that the more frequently a character occurs, the shorter its code. So, let's start by looking at how frequently each letter occurs!

Character	'a'	'b'	'c'	'd'	' '
Frequency	4	3	2	1	3

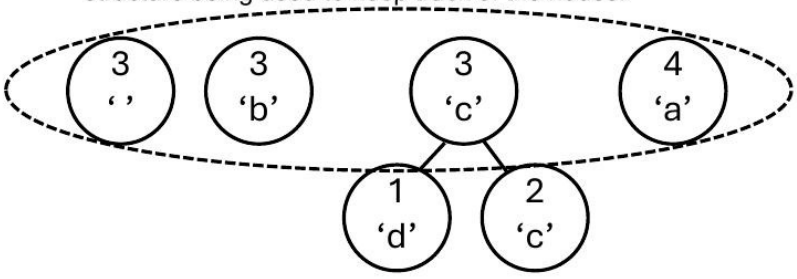
Now let's start building a tree! The tree building process will rely on knowing the characters with the lowest two frequencies. Keeping track of that can be done in multiple ways – for example, using a sorted list or a priority queue.
For our implementation, ties in frequency will be broken by looking at the ASCII value of the two characters involved, and the character with the smaller value makes for the smaller character/frequency combination

1. To get started, we'll create Huffman Nodes for each character. These nodes will all end up as leaf nodes in the completed tree (shown here in a sorted list).

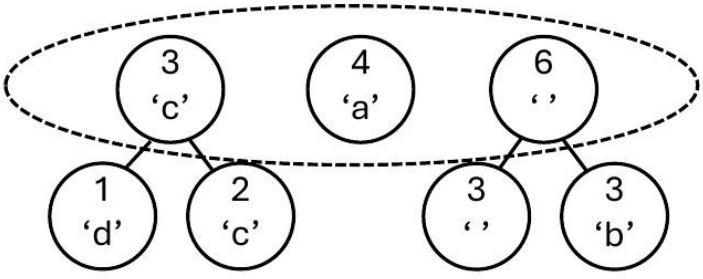


Note that the 3 / ' ' node is less than the 3 / 'b' node:
ASCII value of ' ' = 32
ASCII value of 'b' = 98

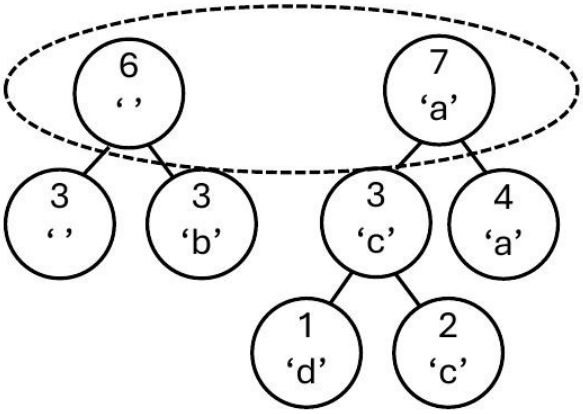
2. While there is more than one node in the set:
   a) Remove the two lowest nodes from the set
   b) Create a new internal node with the two removed nodes as children, with the lesser of the two nodes on the left. The new node's frequency will be the sum of the child node frequencies, and the character will be the smaller of the two children's characters, based again on ASCII value.
   c) Add the newly created node to the set, ensuring its proper location based on the type of data structure being used to keep track of the nodes.



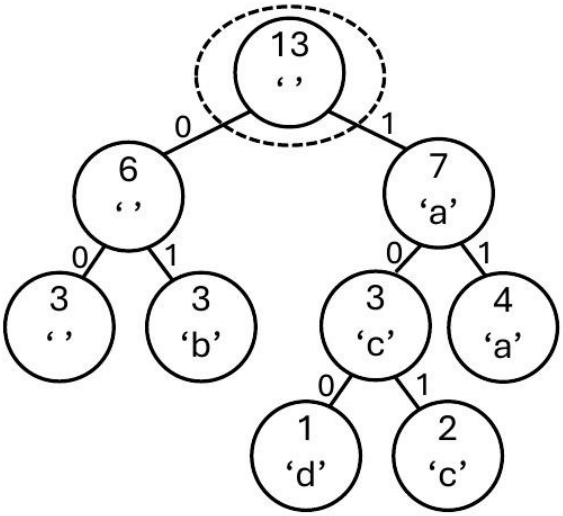
The 1 / 'd' and 2 / 'c' nodes were removed. The 3 / 'c' node was created and added to set. The 3 / 'c' node is positioned after the 3 / 'b' based on ASCII values.



The 3 / ' ' and 3 / 'b' nodes were removed. The 6 / ' ' node was created and added to set.



The 3 / ‘c’ and 4 / ‘a’ nodes were removed. The 7 / ‘a’ node was created and added to set.



The 6 / ‘ ’ and 7 / ‘a’ nodes were removed. The 13 / ‘ ’ node was created and added to set.

At this point we have only one node left in the set, so we are done building the Huffman Tree.

Note that a ‘0’ label has been added to each edge from a parent to the left child and a ‘1’ label has been added to each edge from a parent to the right child. These labels are added for illustrative purposes only – to help understand the next step.

3. Now that we have built the Huffman Tree, we can use it to determine the code for each character. Note again that our original Huffman Nodes are all leaf nodes in the completed tree. As we navigate from the root node to each of those leaf nodes, the path taken to reach each node will determine the associated code, represented by the sequence of 0’s and 1’s.

Character	‘ ’	‘b’	‘d’	‘c’	‘a’
Code	00	01	100	101	11

If we were to compress the entire file from earlier ("abcd abc ab a"), we would get the string of codes "1101101100001101101001101001 1" . One thing you may notice is that there are no spaces separating the codes. How will we find the breaks when decompressing? We'll know that we've found the character corresponding to a code when we hit a leaf in the tree!

So, to decompress the string "1101101100001101101001101001 1", let's traverse the tree. A '1' will mean to go right, a '0' will mean to go left. Every time we hit a leaf node, we've decompressed one character. We'll then move back to the root and keep going. This does mean we’ll need to have the Huffman Tree available when decoding.