Huffman coding

Useful library imports

In [1]:
```
from time import perf_counter
import os
from heapq import heappush, heappop
from bitarray import bitarray
```

Data imports

In [2]:
```
data = [open('1kb.txt', mode = 'r', encoding='utf-8').read(),
        open('10kb.txt', mode = 'r', encoding='utf-8').read(),
        open('100kb.txt', mode = 'r', encoding='utf-8').read(),
        open('1mb.txt', mode = 'r', encoding='utf-8').read()]

data2 = [open('pan-tadeusz.txt', mode = 'r', encoding='utf-8').read(),
        open('lalka-tom-pierwszy.txt', mode = 'r', encoding='utf-8').read()]
```

Static Huffman coding implementation

In [3]:
```
class Node:
    def __init__(self, char, left_child, right_child, weight):
        self.left = left_child
        self.right = right_child
        self.weight = weight
        self.char = char

    def __lt__(self, other):
        if not isinstance(other, type(self)):
            return NotImplemented
        return self.weight < other.weight

# słownik
def find_number_of_occurences(text):
    hashmap = {}

    for char in text:
        hashmap[char] = 1 if char not in hashmap.keys() else hashmap[char] + 1

#     print(hashmap)
    return sorted([elem for elem in hashmap.items()], key = lambda x : x[1])

def build_static_huffman_tree(char_cnt):
    nodes = [Node(char, None, None, weight) for char, weight in char_cnt]

    while len(nodes) > 1:
        node1 = heappop(nodes)
        node2 = heappop(nodes)

        heappush(nodes, Node(None, node1, node2, node1.weight + node2.weight))

    return nodes[0]

def static_find_encoding(root):
    encoding = {}
    def rek(node, code = ''):
        if node.left is None:
            encoding[node.char] = bitarray(code)
        else:
            rek(node.left, code + '1')
            rek(node.right, code + '0')

    rek(root)
    return encoding

def static_encode(text):
    char_cnt = find_number_of_occurences(text)
    root = build_static_huffman_tree(char_cnt)
    hashmap = static_find_encoding(root)

    alphabet_size  = len(hashmap.keys())
    alphabet = bitarray()
    alphabet.frombytes(alphabet_size.to_bytes(2, byteorder='big', signed=False))
    for char, weight in char_cnt:
        char_bits = bitarray()
        weight_bits = bitarray()
        char_bits.frombytes((ord(char)).to_bytes(2, byteorder='big', signed=False))
        weight_bits.frombytes(weight.to_bytes(3, byteorder='big', signed=False))
        alphabet += char_bits + weight_bits

    coded_text = bitarray()
    for char in text:
        coded_char = bitarray(hashmap[char])
        coded_text += coded_char

    return alphabet + coded_text

def static_decode(encoded_data):
    alphabet_size = int.from_bytes(encoded_data[:16], byteorder='big', signed=True)
    char_cnt = []
    encoded_data = encoded_data[16:]
    for _ in range(alphabet_size):
        char = chr(int.from_bytes(encoded_data[:16], byteorder='big', signed=False))
        encoded_data = encoded_data[16:]
        weight = int.from_bytes(encoded_data[:24], byteorder='big', signed=False)
        encoded_data = encoded_data[24:]
        char_cnt.append((char, weight))

    char_cnt = sorted(char_cnt, key = lambda x : x[1])
    root = build_static_huffman_tree(char_cnt)

    decoded_text = ''
    current_node = root
    i = 0
    n = len(encoded_data)

    while i < n:
        if current_node.left is None:
            decoded_text += current_node.char
            current_node = root
        else:
            if encoded_data[i] == True:
                current_node = current_node.left
            else:
                current_node = current_node.right
            i += 1

    return decoded_text
```

Opracowany format pliku:

- Plik rozpoczyna się od zapisanej na 2 bajtach ilości liter w alfabecie (zbiór znaków występujących w zakodowanym tekście) (oznaczmy jako **n**).
- Następnie zawiera zakodowane na dwóch bitach odpowiednio numer znaku w tablicy ASCII oraz ilość jego wystąpień w tekście. (Na tej podstawie jesteśmy w stanie utworzyć identyczne drzewo Huffmana, jak to, które zostało użyte do zakodowania tekstu).
- Reszta pliku to zakodowany tekst.

In [4]:
```
def test_static_huffman(filename):
    output_file = "encoded_" + filename
    with open(filename, 'r', encoding="UTF-8") as file:
        text = file.read()

    print("Testing static huffman tree operations.")
    print("File: " + filename)

    start_time = perf_counter()
    encoded_text = static_encode(text)
    end_time = perf_counter()

    print("Encoding time: ", end = '')
    print("{:.8f}".format(end_time - start_time) + " s")

    start_time = perf_counter()
    decoded_text = static_decode(encoded_text)
    end_time = perf_counter()

    print("Decoding time: ", end = '')
    print("{:.8f}".format(end_time - start_time) + " s")

    with open(output_file, "wb") as file:
        encoded_text.tofile(file)

    size_normal = os.path.getsize(filename)
    size_compressed = os.path.getsize(output_file)
    print('compression: ', end = "")
    print("{:.4f}".format(size_compressed * 100 / size_normal) + ' %')
```

In [5]:
```
test_static_huffman("1kb.txt")
```

```
Testing static huffman tree operations.
File: 1kb.txt
Encoding time: 0.00078820 s
Decoding time: 0.00114370 s
compression: 89.6818 %
```

In [6]:
```
test_static_huffman("10kb.txt")
```

```
Testing static huffman tree operations.
File: 10kb.txt
Encoding time: 0.00377620 s
Decoding time: 0.00829530 s
compression: 59.1386 %
```

In [7]:
```
test_static_huffman("100kb.txt")
```

```
Testing static huffman tree operations.
File: 100kb.txt
Encoding time: 0.02950290 s
Decoding time: 0.06557340 s
compression: 55.8587 %
```

In [8]:
```
test_static_huffman("1mb.txt")
```

```
Testing static huffman tree operations.
File: 1mb.txt
Encoding time: 0.28415600 s
Decoding time: 0.71688280 s
compression: 54.7553 %
```