Laboratorium 3 - Michał Szczurek

In [1]:
```python
from bitarray import bitarray
from collections import deque
from timeit import default_timer as timer
import os
```

Statyczne kodowanie Huffmana

Plik wynikowy zawiera:

- informację o ilości symboli
- Kody symboli w postaci:
  [długość symblou] [symbol] [kod] długość symblou - to liczba bajtów symbolu (niektóre symbole są zapisywane na więcej niż 1 w utf-8)
- informajcę o tym ile ostatnich bitów zignorować (są dopełnieniem do bajtu)

In [2]:
```python
class StaticNode:
    def __init__(self, a, weight, left, right):
        self.a = a
        self.weight = weight
        self.left = left
        self.right = right

    def __str__(self):
        res = ""
        if (self.a is None):
            res += ("Letter: -\n")
        else:
             res += (f"Letter: {self.a}\n")
        res += (f"Weight: {self.weight}")
        res += ("\n=====\n")
        return res
```

In [3]:
```python
class StaticHuffman():

    @staticmethod
    def compress(source, dest):
        f = open(source, "r", encoding="utf8")
        text = f.read()
        letter_counts = {}

        for letter in text:
            if letter not in letter_counts:
                letter_counts[letter] = 1
            else:
                letter_counts[letter] += 1

        nodes = []
        for a, weight in letter_counts.items():
            nodes.append(StaticNode(a, weight, None, None)) # leaf
        internal_nodes = deque([])
        leafs = deque(sorted(nodes, key=lambda n: n.weight))
        while(len(leafs) + len(internal_nodes) > 1):
            element_1 = StaticHuffman._pop_rarest(internal_nodes, leafs)
            element_2 = StaticHuffman._pop_rarest(internal_nodes, leafs)
            internal_nodes.append(StaticNode(None, element_1.weight + element_2.weight, element_1, element_2))

        code_dict = {}
        StaticHuffman._generate_code_form_tree(internal_nodes[0], code_dict)
        res_bits = bitarray()
        for letter in text:
            res_bits += code_dict[letter]
        remainder = (8 - len(res_bits)%8)%8
        StaticHuffman._write_meta_info(dest, code_dict, remainder)
        f = open(dest, 'ab')
        res_bits.tofile(f)
        f.close()

    @staticmethod
    def decompress(source, dest, print_res=False):
        code_dict, remainder, array = StaticHuffman._preprocess_file(source)
        arr = array.to01()
        n = len(arr)
        n -= remainder
        res = ""
        curr = ""
        i = 0
        while i < n:
            curr += str(arr[i])
            if curr in code_dict:
                res += code_dict[curr]
                curr = ""
            i += 1
        if print_res:
            print(res)
        res_file = open(dest,'w', newline='', encoding="utf8")
        res_file.write(res)
        res_file.close()

    @staticmethod
    def _write_num(num, file):
        arr = bitarray()
        arr.frombytes(num.to_bytes(1, byteorder='big', signed=False))
        arr.tofile(file)

    @staticmethod
    def _write_char(char, file):
        arr = bitarray()
        # writing char length as well - some have more than 1 byte
        StaticHuffman._write_num(len(char.encode('utf-8')), file)
        arr.frombytes(char.encode('utf-8'))
        arr.tofile(file)

    @staticmethod
    def _write_meta_info(file_name, code_dict, remainder):
        file = open(file_name, 'wb')
        StaticHuffman._write_num(len(code_dict), file)
        for key, value in code_dict.items():
            StaticHuffman._write_char(key, file)
            StaticHuffman._write_num(len(value), file)
            value.tofile(file)
        StaticHuffman._write_num(remainder, file)
        file.close()

    @staticmethod
    def _generate_code_form_tree(root, code_dict, code=None):
        if root is None:
            return
        if code is None:
            code = ""
        if root.a is not None:
            code_dict[root.a] = bitarray(code)
        else:
            StaticHuffman._generate_code_form_tree(root.left, code_dict, code + "0")
            StaticHuffman._generate_code_form_tree(root.right, code_dict, code + "1")

    @staticmethod
    def _preprocess_file(file_name):
        file = open(file_name, 'rb')
        arr = bitarray()
        arr.fromfile(file)
        pointer = 0 # is a little less elegant but faster
        num = int.from_bytes(arr[:8], byteorder='big', signed=True)
        pointer +=8
        code_dict = {}
        for i in range(num):
            symbol_len = int.from_bytes(arr[pointer:pointer +8], byteorder='big', signed=True)
            pointer +=8
            symbol =  arr[pointer: pointer+ 8*symbol_len].tobytes().decode()
            pointer += 8*symbol_len
            code_len =  int.from_bytes(arr[pointer:pointer+8], byteorder='big', signed=True)
            pointer += 8
            code = bitarray(arr[pointer:code_len+pointer])
            if code_len % 8 == 0:
                offset = code_len
            else:
                offset = (code_len//8)*8+8
            arr = arr[offset:]
            code_dict[code.to01()] = symbol
        remainder = int.from_bytes(arr[pointer:pointer+8], byteorder='big', signed=True)
        pointer += 8
        file.close()
        return code_dict, int(remainder), arr[pointer:]

    @staticmethod
    def _pop_rarest(internal_nodes, leafs):
        if len(internal_nodes) == 0:
            e = leafs.popleft()
        elif len(leafs) == 0:
            e = internal_nodes.popleft()
        elif internal_nodes[0].weight < leafs[0].weight:
            e = internal_nodes.popleft()
        else:
            e = leafs.popleft()
        return e
```

Testy statycznego kodowania Huffmana

In [4]:
```python
def measure_static(file_name):
    print("Test dla:", file_name)
    start = timer()
    StaticHuffman.compress(file_name, "res")
    comp_time = timer() - start
    print("Czas kompresji:", comp_time)
    start = timer()
    StaticHuffman.decompress("res", file_name)
    decomp_time = timer() - start
    print("Czas dekompresji:", decomp_time)
    print("Współczynnik kompresji:", 1 - os.path.getsize("res") / os.path.getsize(file_name))
```

In [5]:
```python
for f_name in ["guthenberg_file_1kB.txt","guthenberg_file_10kB.txt","guthenberg_file_100kB.txt",
               "guthenberg_file_1MB.txt","github_file_1kB.txt","github_file_10kB.txt",
               "github_file_100kB.txt","github_file_1MB.txt"]:
    measure_static(f_name)
    print("==========\n")
```

```
Test dla: guthenberg_file_1kB.txt
Czas kompresji: 0.003699300000000072
Czas dekompresji: 0.007256300000000326
Współczynnik kompresji: 0.06659836065573765
==========

Test dla: guthenberg_file_10kB.txt
Czas kompresji: 0.00700719999999988
Czas dekompresji: 0.04951889999999981
Współczynnik kompresji: 0.3798542477787761
==========

Test dla: guthenberg_file_100kB.txt
Czas kompresji: 0.03230690000000003
Czas dekompresji: 0.4659162999999995
Współczynnik kompresji: 0.4223419697361357
==========

Test dla: guthenberg_file_1MB.txt
Czas kompresji: 0.2534502999999999
Czas dekompresji: 5.284068200000001
Współczynnik kompresji: 0.4241315827486687
==========

Test dla: github_file_1kB.txt
Czas kompresji: 0.0016417999999998045
Czas dekompresji: 0.0052032000000004075
Współczynnik kompresji: 0.20198019801980194
==========

Test dla: github_file_10kB.txt
Czas kompresji: 0.004234099999999685
Czas dekompresji: 0.05917069999999924
Współczynnik kompresji: 0.29294046661303297
==========

Test dla: github_file_100kB.txt
Czas kompresji: 0.029420999999999253
Czas dekompresji: 0.5782863000000003
Współczynnik kompresji: 0.32209186195486594
==========

Test dla: github_file_1MB.txt
Czas kompresji: 0.33359389999999856
Czas dekompresji: 6.4478608999999985
Współczynnik kompresji: 0.32221126556396484
==========
```

Współczynnik kompresji jest mały dla małych plików - metadane zajmują wtedy znaczącą część skompresowanego pliku

Dynamiczne kodowanie Huffmana

Pisząc kod wzorowałem się na poniższym opisie:
https://www2.cs.duke.edu/csed/curious/compression/adaptivehuff.html

Tutaj struktura pliu jest nieco inna- Pierwszą linię stanowi liczba bitów do zignorowania na końcu, następnie plik zawiera kody kolejnych symboli oraz nowe symbole pojawiające się w pliku, w momencie, w którym po raz pierwsze mają zostać użyte poprzedzone długością symbolu.

In [6]:
```python
class AdaptiveNode():
    def __init__(self, a, weight, num, parent):
        self.a = a
        self.weight = weight
        self.left = None
        self.right = None
        self.parent = parent
        self.ord = num
```

In [7]:
```python
class AdaptiveHuffman():

    def __init__(self):
        self.root = AdaptiveNode(None, 0, -1, None)
        self.last = self.root
        self.ord = 0
        self.nodes = []
        self.symbols = set()
        self.leafs = {}
        self.leafs["last"] = self.root

    def _get_ord(self):
        self.ord += 1
        return self.ord

    def _get_path(self, node, path):
        if node is self.root:
            return path[::-1]
        if node.parent.right is node:
            path += "1"
        else:
            path += "0"
        return AdaptiveHuffman._get_path(self, node.parent, path)

    @staticmethod
    def swap_nodes(a, b):
        if a.parent is b.parent:
            tmp = a.parent.left
            a.parent.left = a.parent.right
            a.parent.right = tmp
            return

        if a.parent.left is a:
            a.parent.left = b
        else:
            a.parent.right = b
        if b.parent.left is b:
            b.parent.left = a
        else:
            b.parent.right = a

        tmp = a.parent
        a.parent = b.parent
        b.parent = tmp


    def _increase_w(self, node):
        if node is self.root:
            return
        node.weight += 1
        left = node
        min_ord = 10000000
        for n in self.nodes:
            if n.weight + 1 == node.weight and n.ord < min_ord and node.parent is not n and n.parent is not node:
                left = n
                min_ord = n.ord
            if left is not node:
                AdaptiveHuffman.swap_nodes(node,left)
        self._increase_w(node.parent)

    def _add(self, symbol):
        if symbol not in self.symbols:
            self.symbols.add(symbol)
            self.last.ord =  self._get_ord()
            self.last.left = AdaptiveNode(None, 0,  self._get_ord(), self.last)
            self.last.right = AdaptiveNode(symbol, 1, self._get_ord(), self.last)
            self.leafs[symbol] = self.last.right
            self.nodes.append(self.last.left)
            self.nodes.append(self.last.right)
            self.last = self.last.left
            self.leafs["last"] = self.last
            self._increase_w(self.last.parent)
        else:
            self._increase_w(self.leafs[symbol])

    @staticmethod
    def compress(source, dest):
        f = open(source, "r", encoding="utf8")
        res = open(dest, 'wb')
        res_bits = bitarray()
        data = f.read()
        tree = AdaptiveHuffman()
        for symbol in data:
            if symbol not in tree.symbols:
                res_bits += bitarray(AdaptiveHuffman._get_path(tree, tree.leafs["last"], ""))
                AdaptiveHuffman._add_char(symbol,  res_bits)
            else:
                res_bits += bitarray(AdaptiveHuffman._get_path(tree, tree.leafs[symbol],""))

            tree._add(symbol)
        remainder = (8 - len(res_bits)%8)%8
        num_bits = bitarray()
        num_bits.frombytes(remainder.to_bytes(1, byteorder='big', signed=False))
        res_bits = num_bits + res_bits
        res_bits.tofile(res)
        res.close()
        f.close()

    @staticmethod
    def decompress(source, dest):
        f = open(source, 'rb')
        arr = bitarray()
        arr.fromfile(f)
        tree = AdaptiveHuffman()
        res = ""
        remainder = int.from_bytes(arr[0 : 8], byteorder='big', signed=True)
        pointer = 8
        while pointer < len(arr) - remainder:
            node = tree.root
            while node.left is not None and node.right is not None:
                if arr[pointer]:
                    node = node.right
                else:
                    node = node.left
                pointer += 1
            if node is tree.last:
                symbol_len = int.from_bytes(arr[pointer:pointer + 8], byteorder='big', signed=True)
                pointer += 8
                symbol =  arr[pointer: pointer+ 8*symbol_len].tobytes().decode()
                pointer +=  8*symbol_len
                tree._add(symbol)
                res += symbol
            else:
                tree._add(node.a)
                res += node.a
        f.close()
        res_file = open(dest, 'w', encoding='utf-8')
        res_file.write(res)
        res_file.close()

    @staticmethod
    def _add_num(num, bits):
        arr = bitarray()
        arr.frombytes(num.to_bytes(1, byteorder='big', signed=False))
        bits += arr

    @staticmethod
    def _add_char(char, bits):
        arr = bitarray()
        # writing char length as well - some have more than 1 byte
        AdaptiveHuffman._add_num(len(char.encode('utf-8')), bits)
        arr.frombytes(char.encode('utf-8'))
        bits += arr
```

Testy dynamicznego kodowania Huffmana

In [8]:
```python
def measure_adaptive(file_name):
    print("Test dla:", file_name)
    start = timer()
    AdaptiveHuffman.compress(file_name, "res")
    comp_time = timer() - start
    print("Czas kompresji:", comp_time)
    start = timer()
    AdaptiveHuffman.decompress("res", file_name)
    decomp_time = timer() - start
    print("Czas dekompresji:", decomp_time)
    print("Współczynnik kompresji:", 1 - os.path.getsize("res") / os.path.getsize(file_name))
```

In [9]:
```python
for f_name in ["guthenberg_file_1kB.txt","guthenberg_file_10kB.txt","guthenberg_file_100kB.txt",
               "guthenberg_file_1MB.txt","github_file_1kB.txt","github_file_10kB.txt",
               "github_file_100kB.txt","github_file_1MB.txt"]:
    measure_adaptive(f_name)
    print("==========\n")
```

```
Test dla: guthenberg_file_1kB.txt
Czas kompresji: 0.1783520999999979
Czas dekompresji: 0.16235320000000186
Współczynnik kompresji: 0.0751953125
==========

Test dla: guthenberg_file_10kB.txt
Czas kompresji: 1.3385818
Czas dekompresji: 1.3523543000000018
Współczynnik kompresji: 0.24929206132213655
==========

Test dla: guthenberg_file_100kB.txt
Czas kompresji: 14.5324883
Czas dekompresji: 13.913350000000001
Współczynnik kompresji: 0.27180664062500004
==========

Test dla: guthenberg_file_1MB.txt
Czas kompresji: 187.0569601
Czas dekompresji: 197.94360369999998
Współczynnik kompresji: 0.2687101364135742
==========

Test dla: github_file_1kB.txt
Czas kompresji: 0.17272719999999708
Czas dekompresji: 0.14032190000000355
Współczynnik kompresji: 0.13671875
==========

Test dla: github_file_10kB.txt
Czas kompresji: 2.162289899999962
Czas dekompresji: 1.8722402000000216
Współczynnik kompresji: 0.15664062499999998
==========

Test dla: github_file_100kB.txt
Czas kompresji: 20.94024429999996
Czas dekompresji: 20.416902900000025
Współczynnik kompresji: 0.16924804687499995
==========

Test dla: github_file_1MB.txt
Czas kompresji: 220.78444120000006
Czas dekompresji: 230.484019
Współczynnik kompresji: 0.17505888438339967
==========
```

Moje testy wykazują przewagę kodowania statycznego zarówno pod kątem czsu działania jak i współczynnika kompresji.