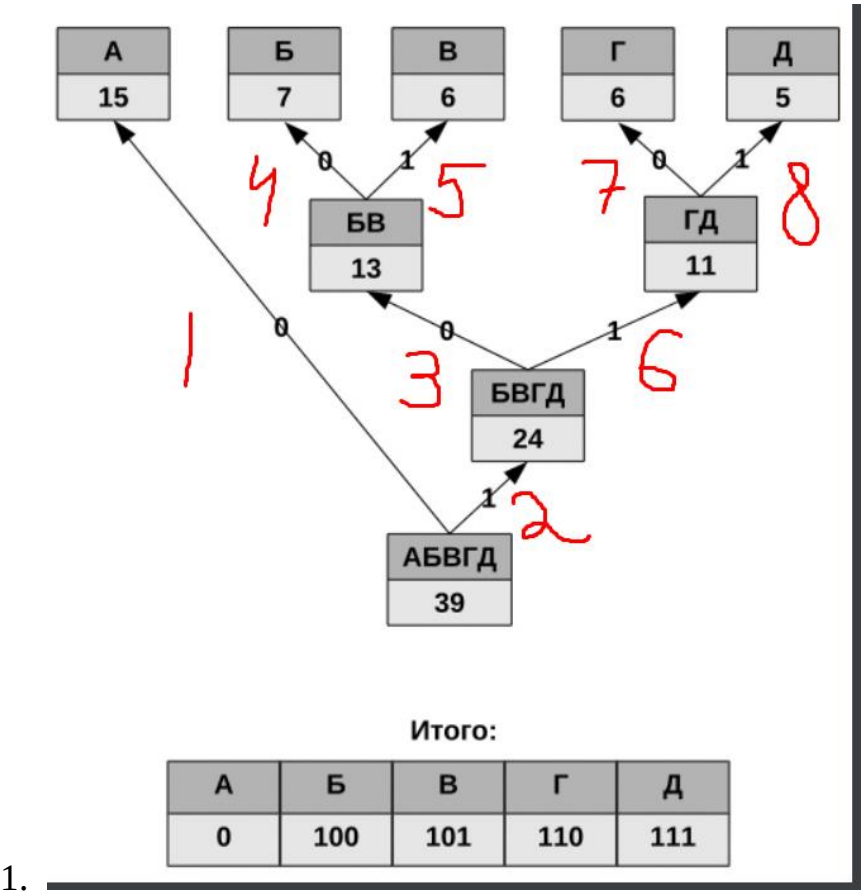
Алгоритм Хаффмана.

1. Тонкости реализации
    a. Основные тонкости реализаци заключаются в создание дерева и прохода по нему, для того, чтобы расставить закодированные значения символов текста.
    b. Создание дерева.:
        i. У нас есть список с частотой каждого символа, его мы сортируем в обратном порядке, после каждой итерации.
        ii. Итерация – “склеивание” двух правых, то есть самых редких элементов, далее это Нода. Процесс идет до момента, пока мы не пройдем все символы.
        iii. У Ноды есть ссылка на два нижних элемента “правый” и “левый”
    c. Расстановка для каждого символа свои значения.
        i. После создания дерева скрипт рекурсивно проходит по каждой Нод-е
        ii. Пример по шагам:

1. 

2. Структуры данных были выбраны и с какой целью
    a. Собстенный класс Node. В нем хранятся: символ, код символа и его соседи, если таковые имеются.
    b. Список состоящий из Node, для прохождения по всем нодам и выдачи каждому символу своего значения, а также списки и словари.

3. Кратко указать описание структуры приложений.

a.

```python
def main():
    encode_or_decode, file_path_with_text, file_path_with_encode_text, file_path_with_codecs, file_path_with_decode_text
    _____= get_parameters()

    algorithm = AlgorithmHuffman(file_path_with_text=file_path_with_text,
                                 file_path_with_encode_text=file_path_with_encode_text,
                                 file_path_with_codecs=file_path_with_codecs,
                                 file_path_with_decode_text=file_path_with_decode_text)

    if encode_or_decode == '1':
        algorithm.encoding_huffman()

    elif encode_or_decode == '2':
        algorithm.decoding_huffman()


if __name__ == '__main__':
    main()
```

i. Сбор седений о путях до файлов.
ii. Создание инстанса класса алгоритма.
iii. При разных параметрах запуск кодирощика или декодера.

b.

```python
def encoding_huffman(self):
    """
    Алгоритм Хаффмана

    :return:
    """
    # Стираем данные о кодировке из файла
    with open(self.__file_path_with_codecs, 'w') as file_writer:
        file_writer.write('')

    # Стираем закодированный текст из файла
    with open(self.__file_path_with_encode_text, 'w') as file_writer:
        file_writer.write('')

    # Получаем строки из файла
    lines = self.get_lines_from_file(self.__file_path_with_text)

    # Получаем лист частоты появлений символов в тексте
    frequency_list = self.get_the_char_frequency(lines=lines)

    # Создание дерева
    root = self.create_the_tree_huffman(frequency_list=frequency_list)

    # Расстановка значений по дереву
    self.set_code_for_char(node=root, code=0)

    # Кодирование текста
    codecs = self.get_codecs(self.get_lines_from_file(self.__file_path_with_cod
    self.encoding_text(codecs, lines)
```

c.

```python
@staticmethod
def get_the_char_frequency(lines: list) -> list:
    """
    Подсчет частоты символов встречающихся в тексте

    :return: Возвращает лист состоящий из некоторого количества Node (буква и количество ее встреч)
    """
    frequency_dict = dict()
    for line in lines:
        for char in line:
            if char.isalpha():
                if char in frequency_dict:
                    frequency_dict[char] += 1
                else:
                    frequency_dict.update({char: 1})

    return [Node(k, v) for k, v in frequency_dict.items()]
```

d.

```python
@staticmethod
def create_the_tree_huffman(frequency_list: list):
    """
    Создание дерева элементов

    :param frequency_list:
    :return:
    """
    tree = copy.deepcopy(frequency_list)

    while len(tree) != 1:
        tree.sort(key=lambda node: node.value, reverse=True)

        first_element = tree.pop(-1)
        second_element = tree.pop(-1)
        name_first_element = first_element.name if first_element.name else first_element.sub_name
        name_second_element = second_element.name if second_element.name else second_element.sub_name

        n = Node(sub_name=name_first_element + ' ' + name_second_element,
                 value=(first_element.value + second_element.value))

        n.left_child = first_element
        n.right_child = second_element

        tree.append(n)

    return tree[0]
```

i. Здесь остановимся. Посняю, что происходит во время одной итерации.
ii. Сортируем список с частотами символов.
iii. Берем два наименьших значения с конца списка и удаляем их из списка
iv. Беру у них имя символа, например буква “a” и “d”, и создается Node с “не основным именем “a d”. и суммой их частот.
v. В Node также записываются ссылки на Node со значениями “a” и “d”.
vi. Новая Node “a d” добавляется в список и начинается новая итерация.

e.

```
def set_code_for_char(self, node, code):
    """
    Проходим по дереву, чтобы подсчитать кодировку, каждого значения
    :param code:
    :param node:
    :return:
    """
    # print(node.name if node.name else node.sub_name, node.value, node.left_child, node.right_
    # print(self.buffer)
    if node.name:
        symbol_value = ''
        for i in range(code):
            symbol_value += str(self.buffer[i])

        with open(self.__file_path_with_codecs, 'a', newline='') as file_writer:
            file_writer.write(f'{node.name} is: {symbol_value}\n')

    elif node.sub_name:
        # Происходит разбиение на то в какую сторону идет дерево, если 0, то влево, если 1, то
        self.buffer[code] = 0
        self.set_code_for_char(node.left_child, code + 1)
        self.buffer[code] = 1
        self.set_code_for_char(node.right_child, code + 1)

    else:

        return
```

i. В данном методе проходим по каждой Node начиная с корневой, то есть со старшей в которой находятся все символы.
ii. Рекурсивно проходим по каждой Node низ по дереву, доходя до значений
iii. Особенности реализации. Если у node существует name – это означет, что это односимвольная node, например с символов “a”