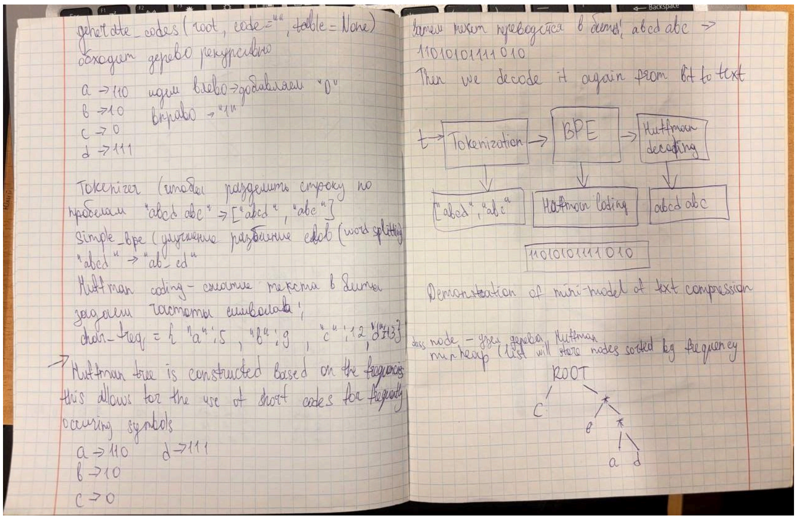
Project overview:

In this mini project, we combined three main ideas:

1. Huffman Coding -a classic lossless compression algorithm that builds a tree based on character frequencies.
2. Tokenizer -simply splits text into pieces.
3. Mini-BPE - a very simple version of Byte-Pair Encoding (used here just for demonstration).

The goal of the project is to show how text moves from characters → tokens → Huffman codes → compressed bits and back.



generate_codes(root, code="", table=None)
обходит дерево рекурсивно

a → 110 идём влево → добавляем "0"
b → 10 вправо → "1"
c → 0
d → 111

Tokenizer (чтобы разделить строку по пробелам "abcd abc" → ["abcd", "abc"]
simple_bpe (улучшение разбиение слов (word splitting)
"abcd" → "ab_cd"
Huffman coding - сжатие текста в биты
задаем частоты символов;
char_freq = {"a":5, "b":9, "c":12, "d":13}
→ Huffman tree is constructed based on the frequencies this allows for the use of short codes for frequently occuring symbols

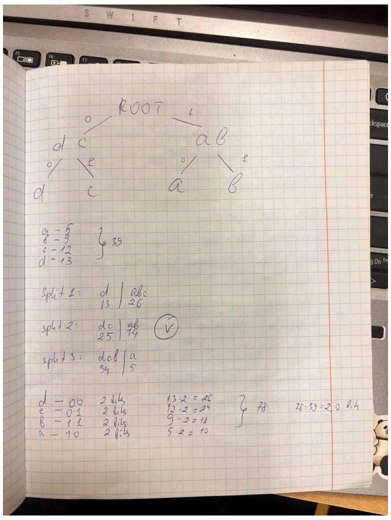
a → 110 d → 111
b → 10
c → 0

затем текст переводится в биты: abcd abc →
11010101111010
Then we decode it again from bit to text

Demonstration of mini-model of text compression

class node - узел дерева Huffman
min_heap (list will store nodes sorted by frequency

ROOT
c
*
b
*
a
d



ROOT
0
1
dc
ab
0
1
0
1
d
c
a
b

a - 5
b - 9
c - 12
d - 13 } 39

split 1: d 13 | abc 26
split 2: dc 25 | ab 14 ✓
split 3: dcb 34 | a 5

d - 00 2 bits 13·2 = 26
c - 01 2 bits 12·2 = 24
b - 11 2 bits 9·2 = 18
a - 10 2 bits 5·2 = 10 } 78 78:39=2,0 bits

2. Node- Huffman Tree Node

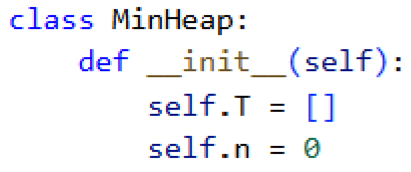
```
class Node:
    def __init__(self):
        self.info = '\0'
        self.freq = 0
        self.code = ''
        self.Llink = None
        self.Rlink = None
```

char- the character stored in the node ('a').

freq-how many times the character appears. The more frequent, the shorter its code.

left and right — pointers to children. In Huffman tree, left = 0, right = 1.

3. MinHeap- Priority Queue



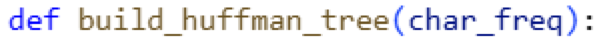
```
class MinHeap:
    def __init__(self):
        self.T = []
        self.n = 0
```

Why a heap?

Huffman tree needs to repeatedly take the two least frequent nodes and combine them.
A min-heap lets us find the smallest frequency efficiently.

- insert() — adds a node and moves it up if necessary.
- remove_min() — removes the smallest node.
- _up() and _down() — maintain heap property.
- _swap() — swaps two nodes.

4. Building the Huffman Tree



```
def build_huffman_tree(char_freq):
```

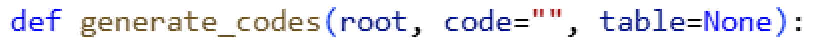
Process

1. Turn each character into a node.
2. Put all nodes into a heap.
3. While more than one node exists:

    - Remove two nodes with smallest frequencies.
    - Combine them into a new node "*" (internal node).
    - Insert it back into the heap.

4. The last remaining node is the root.

5. Generating Huffman Codes (Recursive)



```
def generate_codes(root, code="", table=None):
```

How it works:

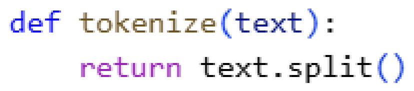
- If the node is a leaf → save its path (0s and 1s) as its code.
- If the node is internal → go left/right recursively.

Example result
{'a':'110', 'b':'10', 'c':'0', 'd':'111'}

Why it works?

More frequent symbols end up closer to the root → shorter codes.

6. Tokenizer

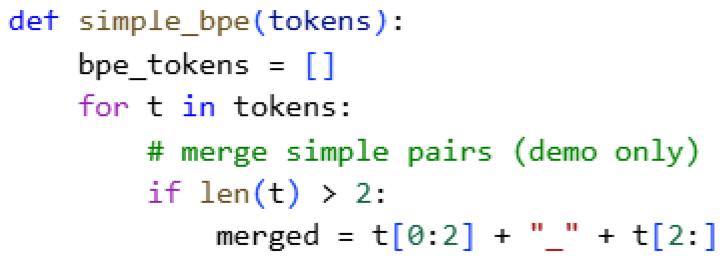


```
def tokenize(text):
    return text.split()
```

How it works?

Simply splits text by spaces:
"abcd abc" → ["abcd", "abc"].

7. Simple BPE (Very Simplified)



```python
def simple_bpe(tokens):
    bpe_tokens = []
    for t in tokens:
        # merge simple pairs (demo only)
        if len(t) > 2:
            merged = t[0:2] + "_" + t[2:]
```

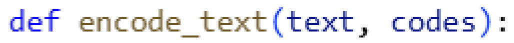
Real BPE

- Finds most frequent symbol pairs and merges them.

Here

We just add _ between 2nd and 3rd character as a demonstration.

8. Encoding Text



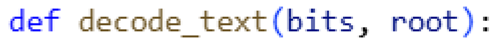
```python
def encode_text(text, codes):
```

What it does? Each character is replaced by its Huffman code.

Example:
 Text: "abcd"
 Codes: a=110, b=10, c=0, d=111

→ "110100111"

9. Decoding



```python
def decode_text(bits, root):
```

How it works:

1. Read bits one by one.

2. Traverse the tree:

   a. 0 → left

   b. 1 → right

3. When you reach a leaf → found a character → return to root.

10. Example Input
char_freq = {"a":5, "b":9, "c":12, "d":13}

- Rare characters → longer code

- Frequent characters → shorter code

This project implements a mini pipeline:
text → tokens → BPE → Huffman → bit string → text.

We manually built the Huffman tree, generated codes for each character, and demonstrated text compression and decompression.
We also added a simple version of BPE to illustrate how token merging works, showing the idea behind modern text compression and tokenization in NLP models.