Huffman Coding Visualizer

Divyanshi Maurya
12219959

CERTIFICATE

of Training Completion

LINUX SOCIALS

Divyanshi Maurya

Congratulations of successful completion of you DSA-Training with LinuxSocials. We wish you all the very best in your endeavours ahead.

28 - August - 2024

DATE

RAHUL MAHESHWARI

Founder @ LINUX SOCIALS

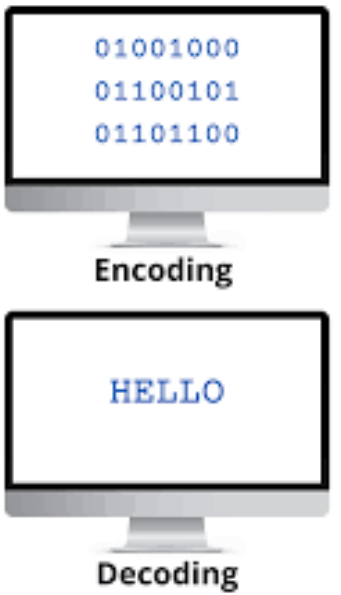
WHAT IS DATA ENCODING?

Data encoding is the process of transforming data into a specific format using a defined scheme, allowing for efficient storage, transmission, and interpretation of the information.

Encoding focuses on representing data in a way that is suitable for different systems and protocols.

It helps on maintaining integrity during transmission and storage.

It helps in compatability.



DATA ENCODING V/S DATA ENCRYPTION

Data encoding and data encryption serve different purposes.

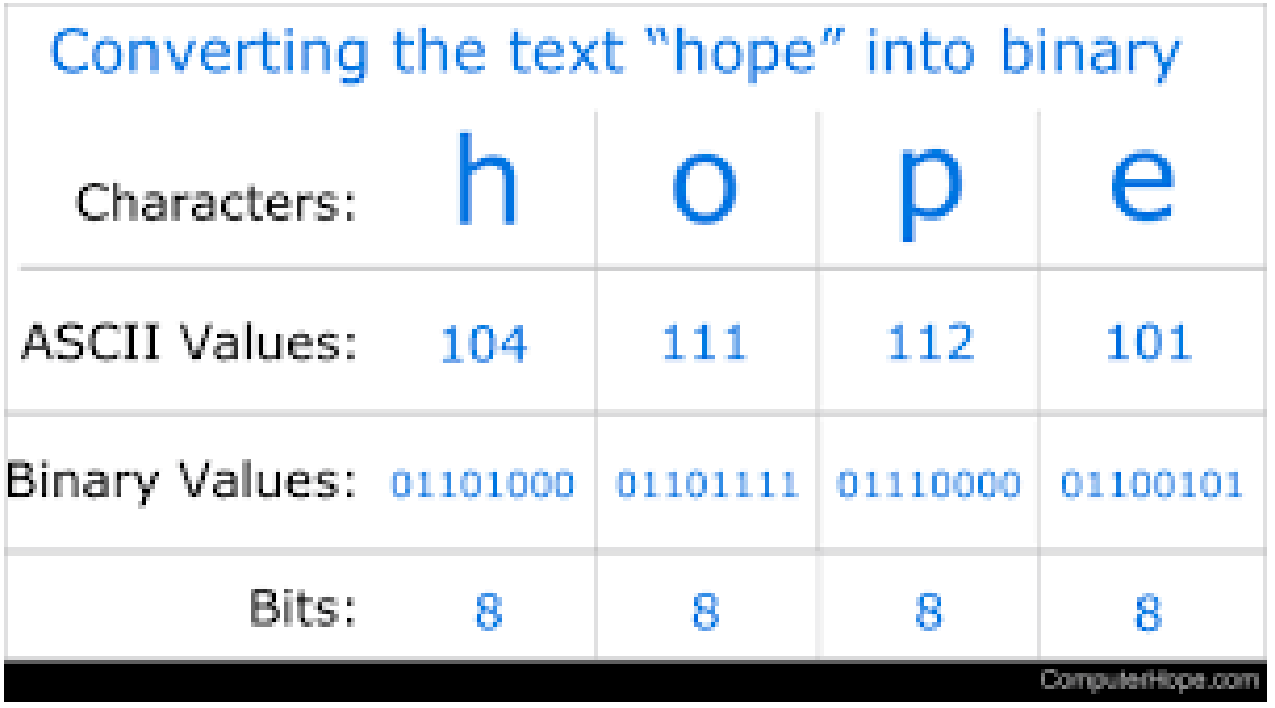
Encoding means to transform data into a specific format for efficient storage and transmission, ensuring compatibility across systems. Hence, it is not meant for security. It is straightforward and reversible.

Encryption converts data into a coded format to protect it from unauthorized access, requiring a decryption key to return to its original form. It is focused on confidentiality and data security.

WHAT ARE THE TYPES OF DATA ENCODING?

1. Character Encoding
2. Base Encoding
3. URL Encoding
4. Hexadecimal Encoding
5. Binary Encoding
6. Morse Code
7. Audio and Video Encoding

HUFFMAN CODING COMES UNDER BINARY ENCODING

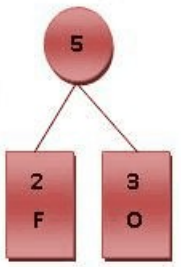


WHAT IS HUFFMAN CODING?

It a lossless data compression algorithm that ensures no information is lost during the compression process.

It assigns variable-length binary codes to characters based on their frequency of occurrence.

The Huffman coding technique guarantees that no code assigned to one character is a prefix of another character's code. This prefix property allows for unambiguous



WHAT IS HUFFMAN TREE?

It is a binary tree used in the Huffman coding algorithm.

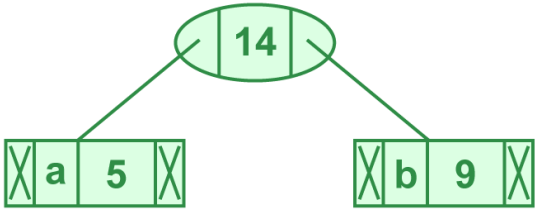
It represents characters and their frequencies as nodes.

It is created using a priority queue implemented as a min-heap.

Each leaf node corresponds to a character, while internal nodes represent the sum of the frequencies of their child nodes.

The path from the root to a leaf node determines the binary code for that character.

The structure of the Huffman Tree reflects the frequency of characters, with more frequent characters closer to the root and assigned shorter codes.



WHAT IS A PRIORITY QUEUE?

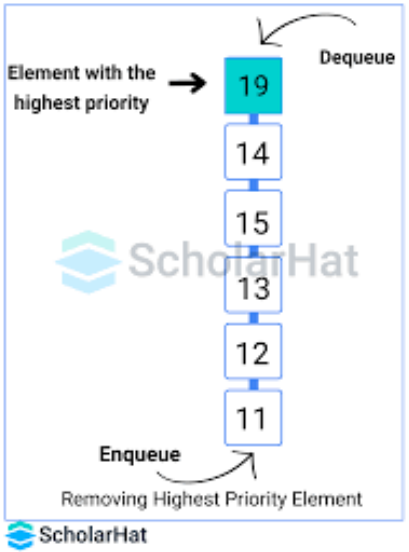
It is an abstract data structure

It extends the basic queue functionality by assigning a priority to each element.

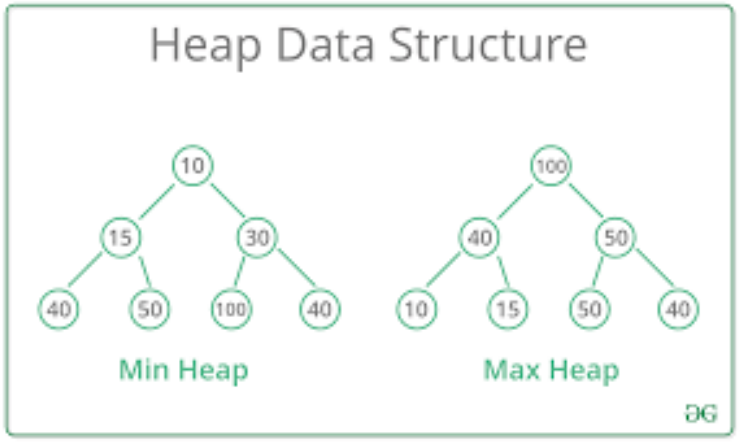
Elements are processed based on their priority rather than the order of their insertion. This means that an element with a higher priority is served before elements with lower priorities.

It is implemented using various underlying data structures, such as arrays linked lists, or heaps.

Commonly used in algorithms like Dijkstra’s shortest path and Huffman coding, where tasks must be executed in order of importance or urgency.



WHAT IS MIN HEAP?



A min-heap is a binary tree-based data structure where the value of each parent node is smaller than or equal to the values of its child nodes. The smallest value is always found at the root of the tree. A min-heap ensures that the tree is complete, meaning all levels are fully filled except possibly for the last, which is filled from left to right. Min-heaps are often implemented using arrays and are used in algorithms like

The heap property allows efficient retrieval and removal of the smallest element, typically in logarithmic time.

MY PROJECT

FILE STRUCTURE

```
huffman_visualizer
 |
 |_Static
 |   |_style.css
 |_Templates
 |   |_index.html
 |_app.py
```

I have used static and template directory for the better organization and separation of the files.

Flask uses Jinja2 templating engines, so it expects the HTML files to be placed in template directory for dynamic rendering of HTML content in Python

Static directory is used to contain all static contents : css and images, they do not change dynamically. It also ensures that these sites are easily accesible and manageable.

OBJECTIVE AND AIM

The objective of this project is to implement Huffman coding as a data compression algorithm using a Flask web application. The project aims to provide an interactive platform where users can input text data and visualize the resulting Huffman tree and corresponding binary codes. By achieving this, the project seeks to enhance understanding of lossless data compression techniques, demonstrate the practical application of Huffman coding, and enable users to explore the efficiency of data encoding based on character frequency.

The aim of this project is to provide an educational and practical implementation of Huffman coding that helps users understand the significance of data compression techniques in computer science. By developing a web-based application, the project aims to make complex concepts more accessible and engaging, enabling users to explore the mechanics of Huffman coding and appreciate its relevance in real-world applications such as file compression, data transmission, and storage efficiency.

Huffman Coding Visualizer

Enter Text:

hello

Generate Huffman Tree

Huffman Codes:

o: 00 (Freq: 1)
e: 01 (Freq: 1)
h: 10 (Freq: 1)
l: 11 (Freq: 2)

ASCII Codes:

```
h: 104
e: 101
l: 108
l: 108
o: 111
```

Huffman Tree Structure:

```
|-- 5
|   |-- 2
|   |   |-- 1
|   |   |   o(1)
|   |   |-- 1
|   |       e(1)
|   |-- 3
|       |-- 1
|       |   h(1)
|       |-- 2
|           l(2)
```

Ratio of Huffman Codes to ASCII Codes:

Huffman Code Length: 10, ASCII Code Length: 40, Ratio: 0.25

DATA STRUCTURES I HAVE USED

Binary Tree: In the code , The Node class represents a node in the Huffman tree, which is a binary tree. Each node contains information about the frequency (freq), the character (symbol), and pointers to its left and right children. The binary tree is constructed during the Huffman coding process to represent the characters and their frequencies hierarchically.

Min-Heap (Priority Queue): I have used the heap is implemented using a min-heap, created with the **heapq** module. This data structure is used to efficiently retrieve and remove the nodes with the smallest frequency when building the Huffman tree. The heapq library provides functions to maintain the heap property, ensuring that the node with the lowest frequency is always at the root.

Dictionary: The **codes** dictionary stores the Huffman codes for each character, where the keys are the characters and the values are their corresponding binary codes.
Another dictionary-like structure, created using **Counter**, is used to count the frequency of each character in the input text.

TECHNOLOGIES I HAVE USED

PSEUDO-CODE

```
FUNCTION Node(frequency, symbol):
    SET self.freq = frequency
    SET self.symbol = symbol
    SET self.left = None
    SET self.right = None


    FUNCTION __lt__(other):
        RETURN self.freq < other.freq

FUNCTION huffman_coding(text):
    IF text is empty THEN
        RETURN empty dictionary
```

```
frequency = COUNTED the character frequencies in text
heap = CREATED a list of Node objects for each character with its frequency
HEAPIFY(heap)
WHILE length of heap > 1 DO
    left = HEAPPOP(heap)
    right = HEAPPOP(heap)
    merged = Node(left.freq + right.freq)
    SET merged.left = left
    SET merged.right = right
    HEAPPUSH(heap, merged)
root = heap[0]
codes = {}
CALL _generate_codes(root, "", codes)
RETURN codes, frequency, root
```

```
FUNCTION _generate_codes(node, current_code, codes):
    IF node is not None THEN
        IF node.symbol is not None THEN
            SET codes[node.symbol] = current_code
        CALL _generate_codes(node.left, current_code + "0", codes)
        CALL _generate_codes(node.right, current_code + "1", codes)
```

```
FUNCTION build_tree_structure(node, prefix, is_left):
    IF node is None THEN
        RETURN empty string

    tree_str = prefix + ("|-- " if is_left ELSE "|-- ") + node.freq + newline

    IF node.symbol is not None THEN
        tree_str += prefix + ("|   " if is_left ELSE "    ") + node.symbol + "(" + node.freq + ")" + newline

    new_prefix = prefix + ("|   " if is_left ELSE "    ")
    tree_str += build_tree_structure(node.left, new_prefix, True)
    tree_str += build_tree_structure(node.right, new_prefix, False)

    RETURN tree_str
```

```
FUNCTION index():
    RETURN render_template('index.html')


FUNCTION build_huffman_tree():
    data = request.json
    text = data['text']
    codes, frequency, root = huffman_coding(text)


    tree_structure = build_tree_structure(root)
    huffman_codes = []
    FOR each character, code in codes DO
        APPEND {"symbol": char, "code": code, "freq": frequency[char]} TO huffman_codes
    RETURN jsonify(huffman_codes=huffman_codes, tree_structure=tree_structure)


IF __name__ == '__main__':
    RUN app with debug mode enabled
```

FUTURE GOAL

For future goals, the project can be expanded by incorporating additional features such as:

In future we can allow users to upload text files for compression instead of manual text input.

We can also implement a functionality to decompress the Huffman-encoded data and retrieve the original text.

We can also include performance analysis tools to compare compression ratios and speeds for different datasets.

We can also improve the graphical representation of the Huffman tree using libraries like D3.js for better user engagement and understanding.

We can also extend the application to support multiple languages and character sets, making it more versatile.

CONCLUSION

In conclusion, the Huffman coding project successfully demonstrates the principles of lossless data compression through a user-friendly Flask web application. The implementation allows users to visualize the Huffman tree and the assigned binary codes, highlighting the efficiency of variable-length encoding based on character frequencies. This project not only showcases the practical application of Huffman coding but also serves as an educational tool for understanding fundamental concepts in data compression and encoding techniques.