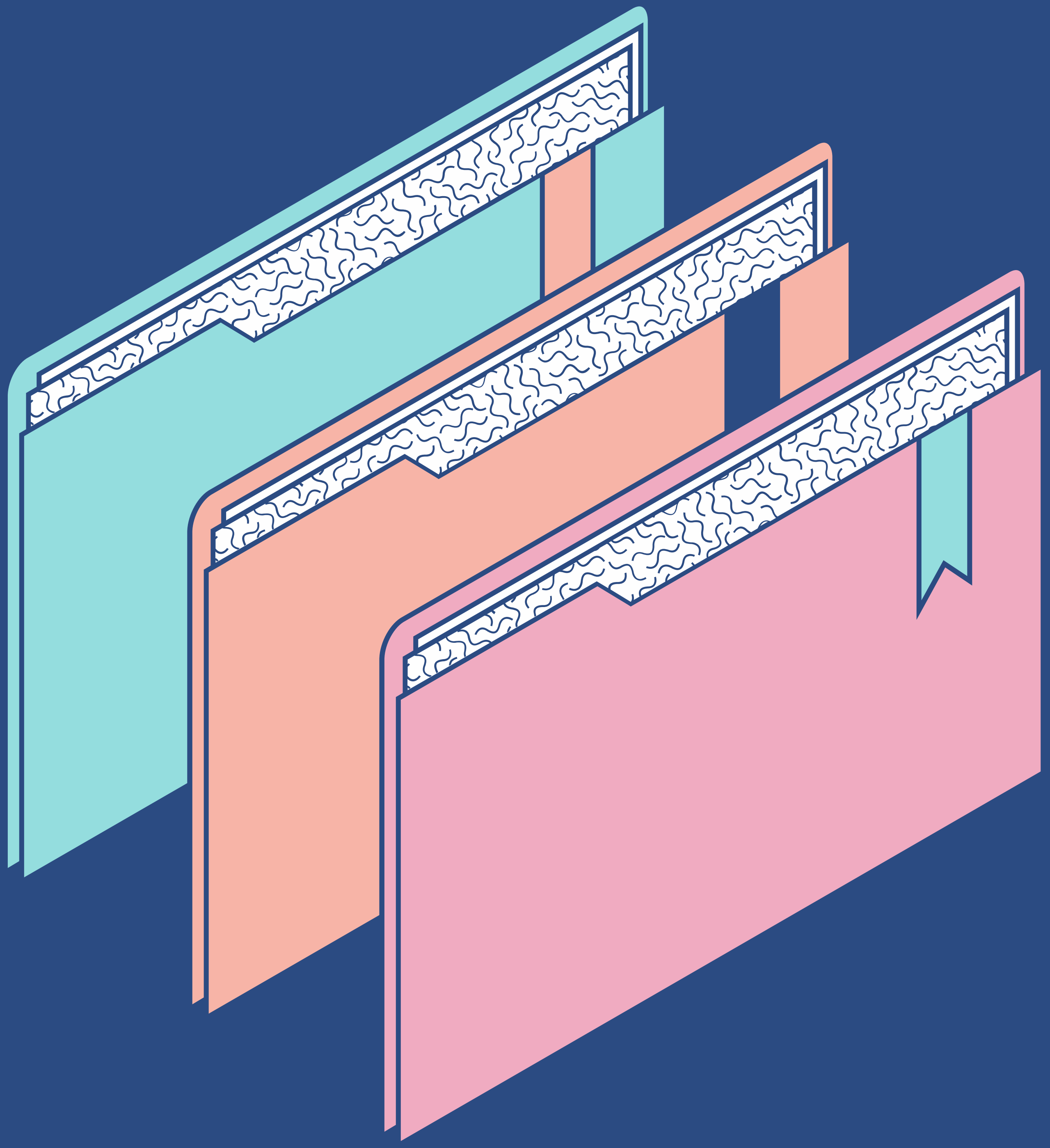
TEXT FILE COMPRESSION

-Using Huffman coding algorithm

BY-
AYESHA MOHANTY(2105784)
NEHA BAJPAYEE(2105807)



Objective

THE MAIN OBJECTIVE OF THE PROJECT

- To compress the text file size for effectiveness
- Provides an efficient, unambiguous code by analyzing the frequencies that certain symbols appear in a message.
-

INTRODUCTION

WHAT IS TEXT FILE COMPRESSION?

Text file compression is a method of reducing the size of a storage file to save storage space and reduce transmission time when transmitting a file over the web. Text files can be of various file types, such as HTML, JavaScript, CSS, .txt, and so on.

Text file compression can be broadly categorized into two types: lossless compression and lossy compression.

Text compression is required because uncompressed data can take up a lot of space, which is inconvenient for device storage and file sharing.

Compression has led for easy
File Storage,Data Transfer,Archiving

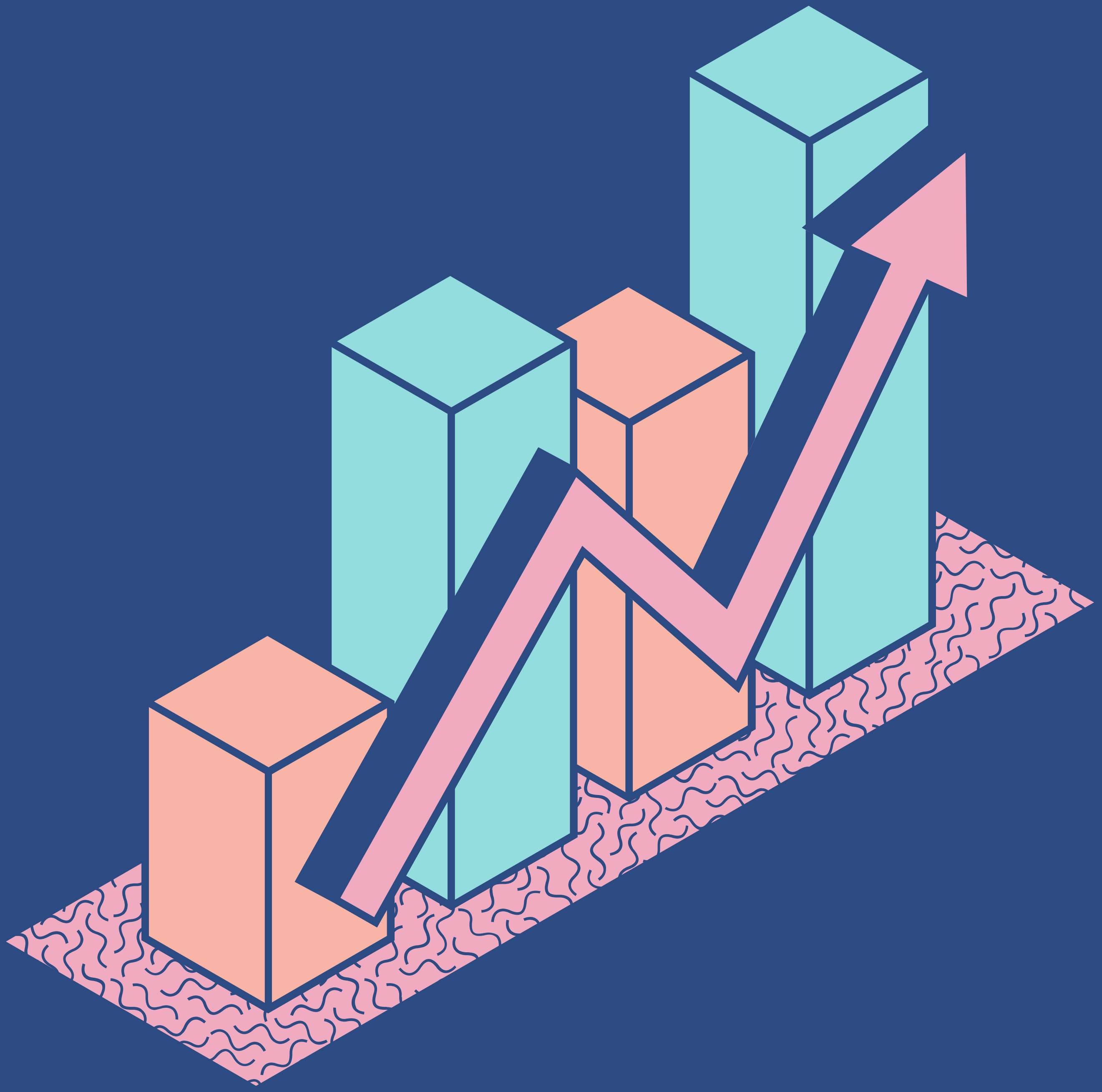
Lossless compression algorithms reduce the file size without losing any data. This means that the original text file can be perfectly reconstructed from the compressed version.

Huffman Coding: Assigns variable-length codes to characters based on their frequencies in the text.

Run-Length Encoding: Replaces sequences of repeated characters with a single character and a count of the repetition. 00003.

Lempel-Ziv-Welch (LZW): Dictionary-based algorithm that replaces repeated sequences of characters with shorter codes.

Lossy compression, on the other hand, sacrifices some amount of data to achieve higher compression ratios. While lossy compression is not suitable for text files where preserving every detail is crucial, it is commonly used for multimedia data like images, audio, and video.
JPEG (Joint Photographic Experts Group):
00002. MP3 (MPEG Audio Layer III):
00003. AAC (Advanced Audio Coding):



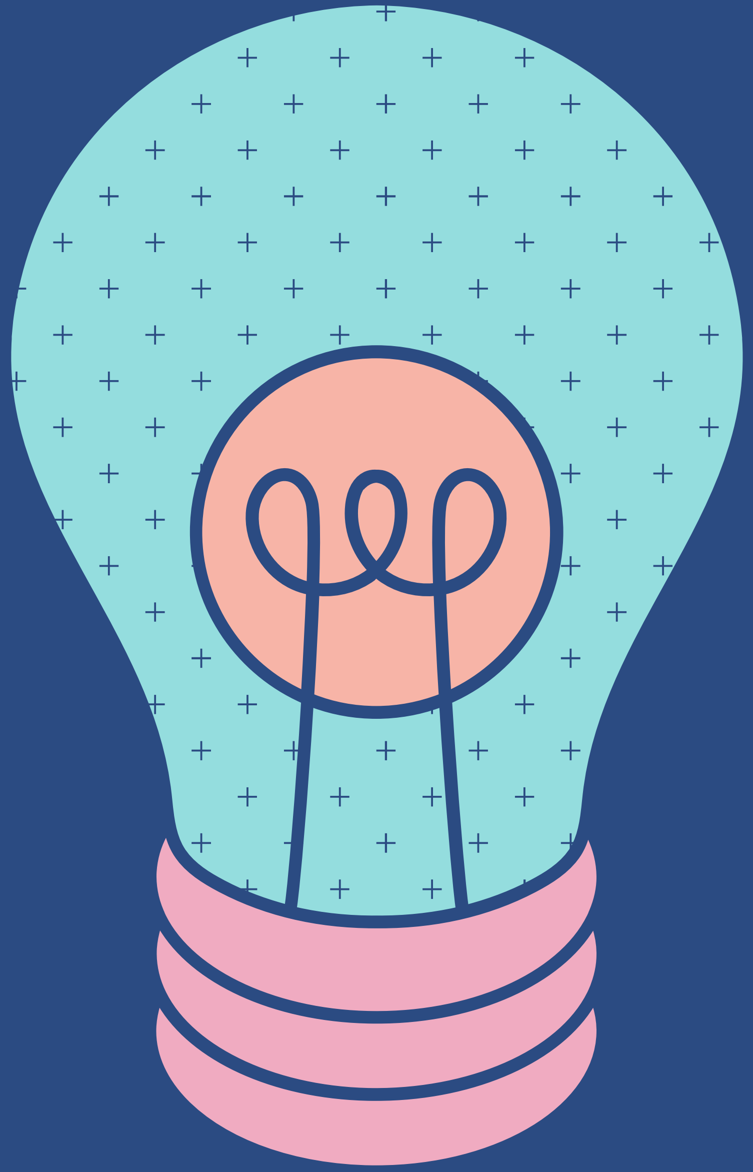
METHODOLOGY

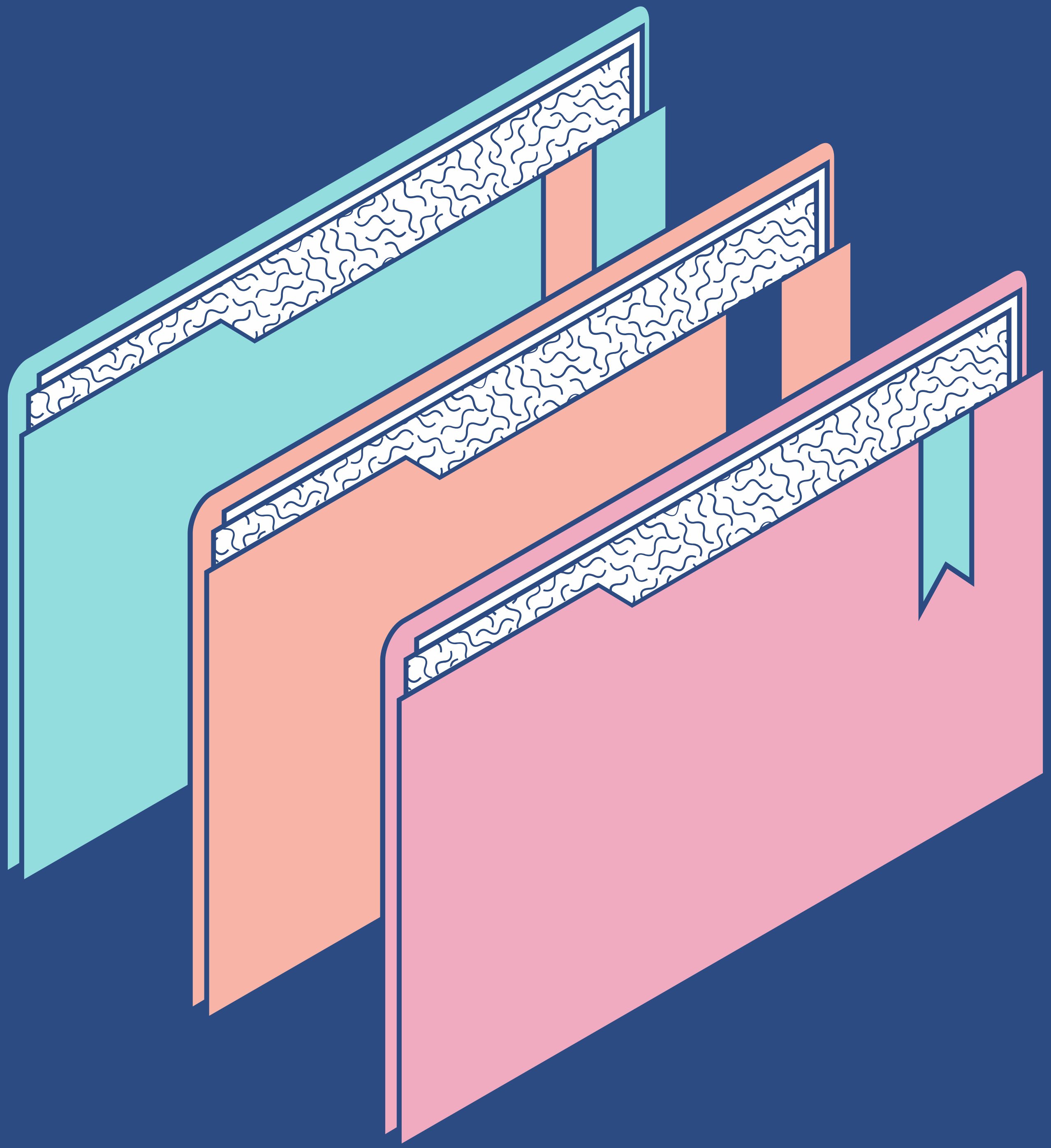
HOW IS A TEXT FILE COMPRESSED?

Instead of using a fixed number of bits for each character (as in ASCII or Unicode where each character is represented using a fixed number of bits), Huffman coding assigns variable-length codes to characters based on their frequency.

By using shorter codes for common characters and longer codes for rare characters, Huffman coding ensures that the most frequently occurring characters are represented using fewer bits. This efficient representation reduces the overall number of bits required to encode the entire data.

When the input data is encoded using these variable-length Huffman codes, the resulting compressed data occupies fewer bits than the original data. This reduction in the number of bits directly translates to a smaller file size.





THE VARIOUS STRUCTURES USED IN THE IMPLEMENTATION OF COMPRESSION:

1. NODE STRUCTURE:

NODE STRUCTURE REPRESENTS A NODE IN THE HUFFMAN TREE.IT CONTAINS A SYMBOL (CHARACTER), FREQUENCY (NUMBER OF OCCURRENCES), AND POINTERS TO THE LEFT AND RIGHT CHILD NODES.

2. MINHEAP STRUCTURE:

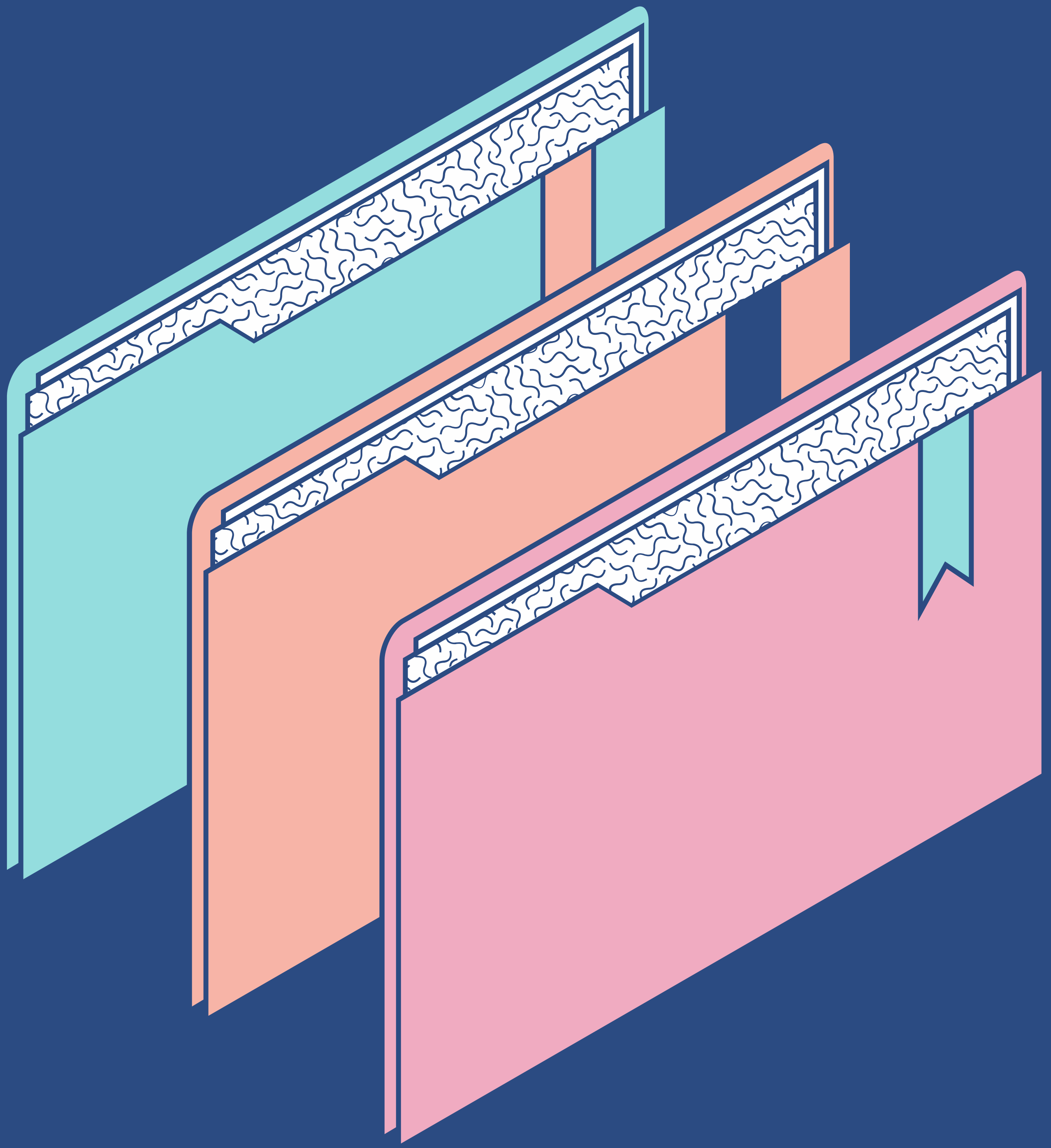
MINHEAP STRUCTURE REPRESENTS A MIN-HEAP DATA STRUCTURE TO EFFICIENTLY EXTRACT NODES WITH MINIMUM FREQUENCIES DURING HUFFMAN TREE CONSTRUCTION.

IT CONTAINS A SIZE (NUMBER OF ELEMENTS) AND AN ARRAY OF POINTERS TO NODE STRUCTURES.

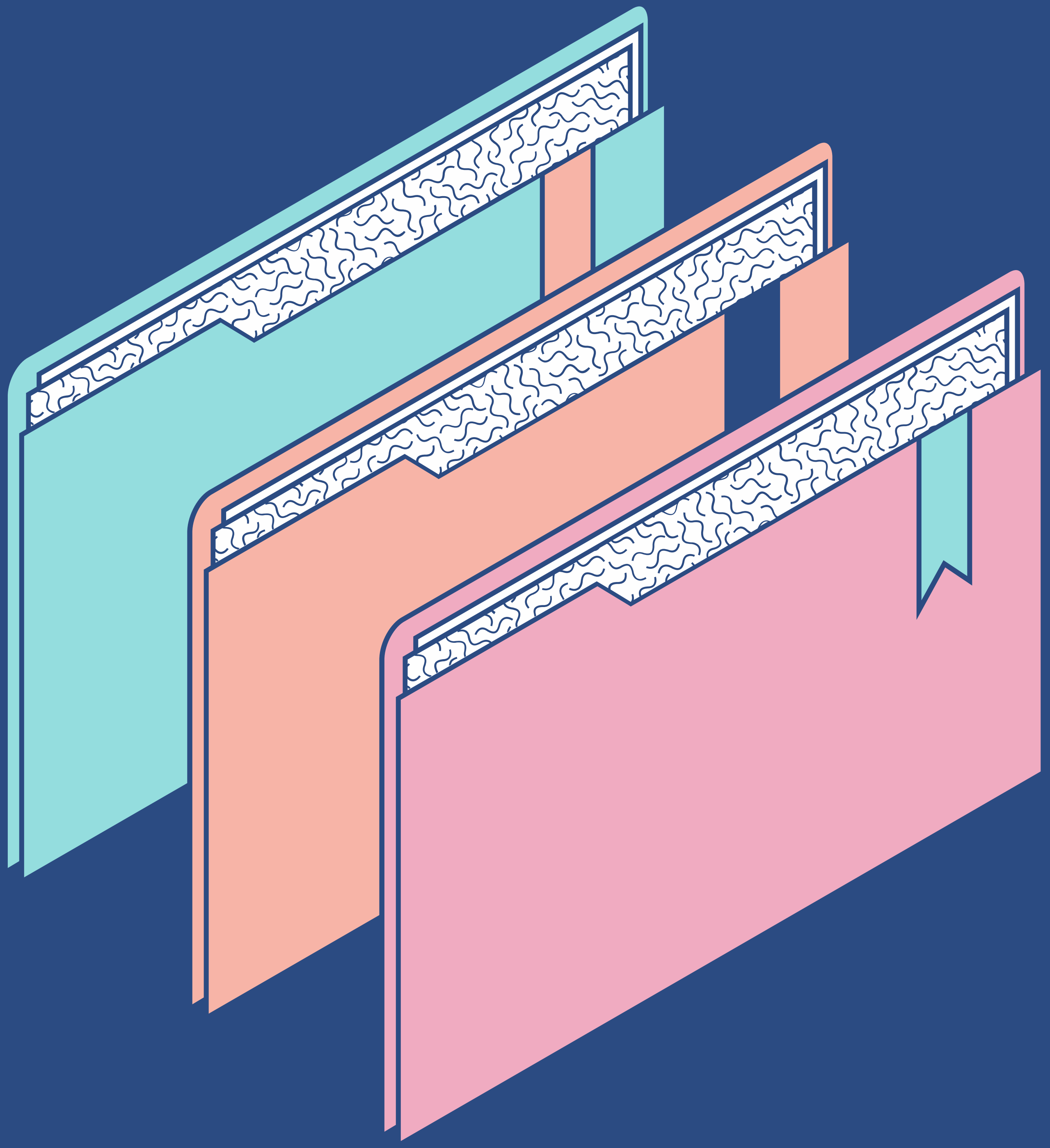
3. CODE STRUCTURE:

CODE STRUCTURE IS USED TO STORE HUFFMAN CODES FOR EACH SYMBOL.

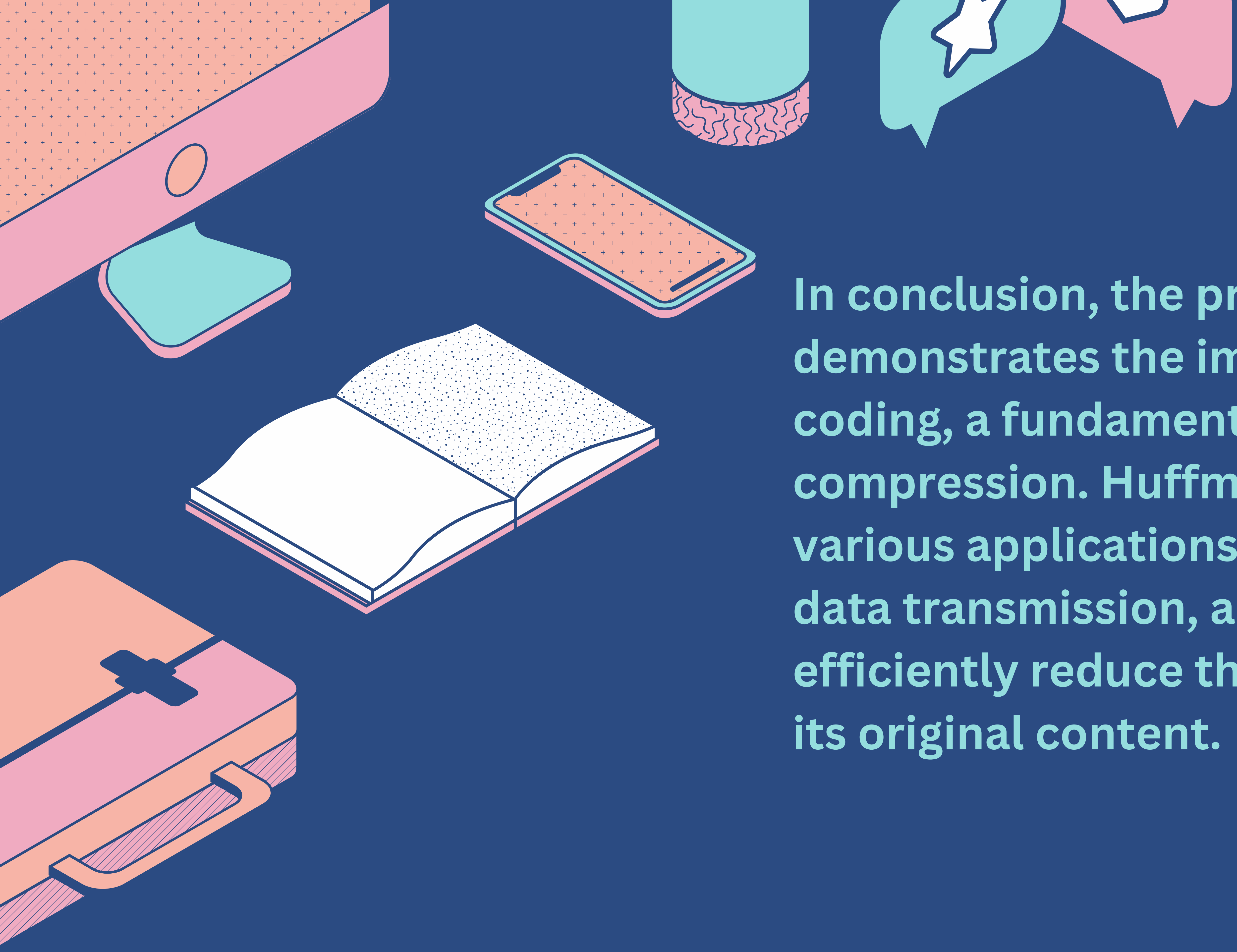
IT CONTAINS A SYMBOL (CHARACTER) AND A CODE (STRING OF '0'S AND '1'S REPRESENTING THE HUFFMAN CODE).



1. createNode(char symbol, int frequency):
Allocates memory for a new Node structure, initializes its fields, and returns the
created node.
2. createMinHeap():
Allocates memory for a new MinHeap structure, initializes its size to 0, and returns
the created min-heap.
3. swapNodes(Node** a, Node** b):
Swaps two Node pointers.
4. minHeapify(MinHeap* minHeap, int index):
Maintains the min-heap property starting from a given index in the minHeap array.
5. buildMinHeap(MinHeap* minHeap):
Builds a min-heap from the given minHeap array.
6. createAndBuildMinHeap(char symbols[], int frequencies[], int size):
Creates a min-heap from symbols and their frequencies, and returns the min-heap.

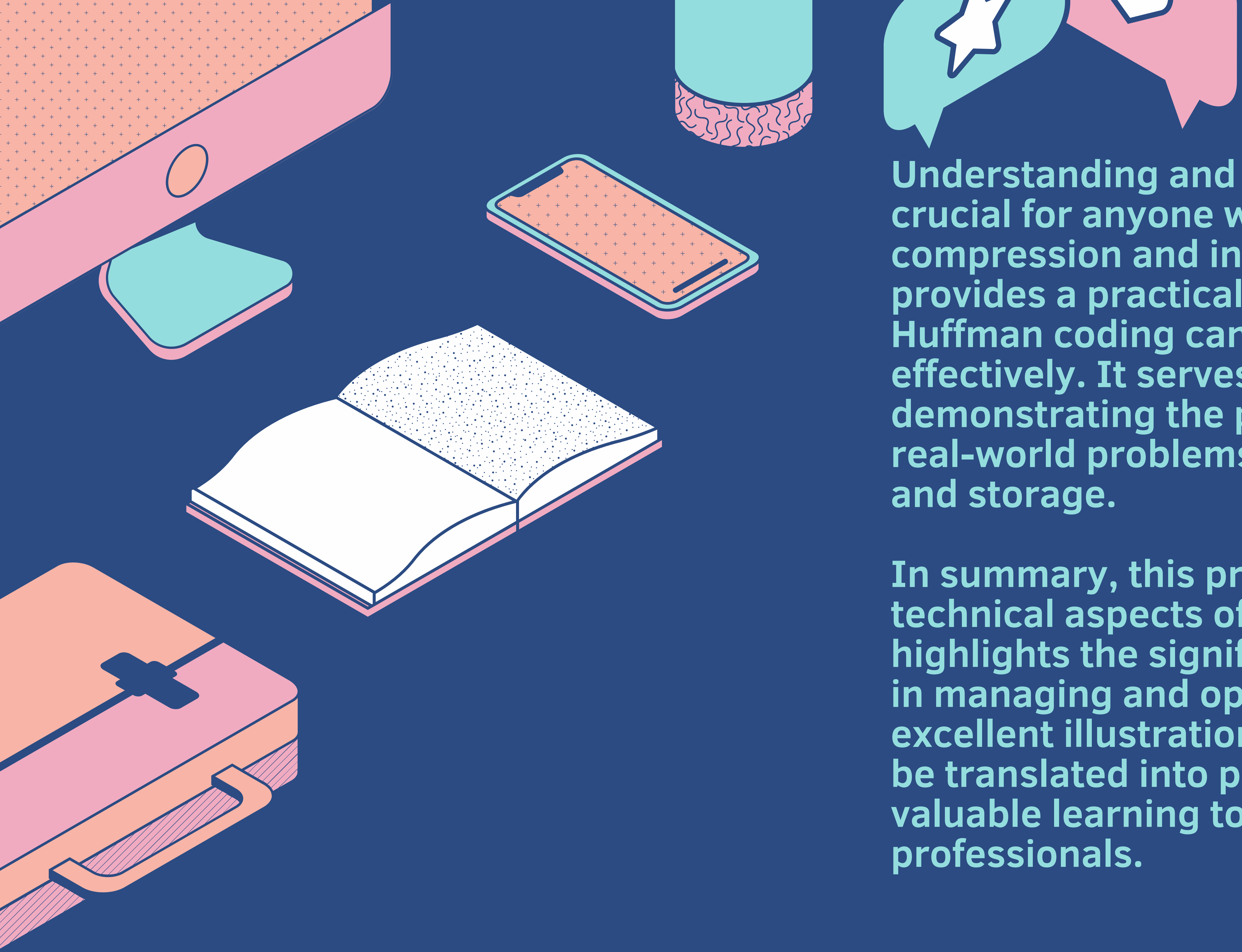


7. extractMin(MinHeap* minHeap):
Extracts the node with the minimum frequency from the min-heap.
8. insertMinHeap(MinHeap* minHeap, Node* node):
Inserts a new node into the min-heap while maintaining the min-heap property.
9. buildHuffmanTree(char symbols[], int frequencies[], int size): Builds the Huffman tree using the given symbols and their frequencies and returns the
root node of the tree.
10. generateCodes(Node* root, char code[], int depth, Code codes[], int*
codesIndex): Recursively generates Huffman codes for each symbol in the Huffman tree.
11.writeCompressedData(FILE* outFile, Code codes[], char* inputFileName): Reads the input file character by character and writes corresponding Huffman codes
to the output file.



CONCLUSION

In conclusion, the presented program demonstrates the implementation of Huffman coding, a fundamental technique in data compression. Huffman coding is widely used in various applications, such as file compression, data transmission, and multimedia systems, to efficiently reduce the size of data while preserving its original content.



Understanding and implementing Huffman coding is crucial for anyone working in the field of data compression and information theory. This program provides a practical and clear example of how Huffman coding can be applied to compress data effectively. It serves as an educational resource, demonstrating the power of algorithms in solving real-world problems related to data manipulation and storage.

In summary, this program not only showcases the technical aspects of Huffman coding but also highlights the significance of algorithmic efficiency in managing and optimizing data. It serves as an excellent illustration of how theoretical concepts can be translated into practical solutions, making it a valuable learning tool for students and professionals.

THANK YOU