

# Huffman Coding

Section: A1

Group: 1

2005001-Anik Saha

2005006-Kowshik Saha Kabya

2005023-Jaber Ahmed Deedar

Department of CSE

Bangladesh University of Engineering and Technology

February 13, 2024

# What is Huffman Coding?

## Huffman Coding

Huffman coding is an efficient method of compressing data without losing information. It provides an efficient, unambiguous code by analyzing the frequencies that certain symbols appear in a message. The algorithm is named after David A. Huffman, who developed it while he was a Sc.D student at MIT.

## A little bit of History

In 1951, Professor Robert M. Fano assigned a term paper on the problem of finding the most efficient binary code. In his paper, David A. Huffman developed an algorithm where he assigned shorter codes to the most frequently occurring characters and longer codes to the less frequently occurring characters, thus employing a variable-length encoding system.



David A. Huffman



Robert M. Fano



Claude Shannon

In doing so, Huffman outdid Fano, who alongside Claude Shannon developed a similar method. Huffman's bottom-up approach turned out to be more optimal than the top-down approach of Shannon and Fano.

## Where is Huffman Coding used?

Huffman coding is used in various applications where data compression is necessary. Some of those applications are:

- **File Compression:** ZIP, gzip, and zlib

# Where is Huffman Coding used?

Huffman coding is used in various applications where data compression is necessary. Some of those applications are:

- **File Compression:** ZIP, gzip, and zlib
- **Text Compression:** ASCII compression, HTML compression

# Where is Huffman Coding used?

Huffman coding is used in various applications where data compression is necessary. Some of those applications are:

- **File Compression:** ZIP, gzip, and zlib
- **Text Compression:** ASCII compression, HTML compression
- **Image Compression:** Entropy encoding in JPEG, image data compression in PNG

# Where is Huffman Coding used?

Huffman coding is used in various applications where data compression is necessary. Some of those applications are:

- **File Compression:** ZIP, gzip, and zlib
- **Text Compression:** ASCII compression, HTML compression
- **Image Compression:** Entropy encoding in JPEG, image data compression in PNG
- **Audio Compression:** MP3, AAC

# Where is Huffman Coding used?

Huffman coding is used in various applications where data compression is necessary. Some of those applications are:

- **File Compression:** ZIP, gzip, and zlib
- **Text Compression:** ASCII compression, HTML compression
- **Image Compression:** Entropy encoding in JPEG, image data compression in PNG
- **Audio Compression:** MP3, AAC
- **Network Communication:** Reduction of the size of data packets transmitted over networks



# What's our objective?

## The Problem

Efficient digital communication requires converting any type of information into binary strings. Since resource is limited, we must find out a way of conversion such that the message requires least possible bandwidth to transmit.

# What's our objective?

## The Problem

Efficient digital communication requires converting any type of information into binary strings. Since resource is limited, we must find out a way of conversion such that the message requires least possible bandwidth to transmit.

## Our Goal

Now we look forward to finding an encoding scheme that requires minimum binary characters on average to encode a message.

# A Naive Solution

Let's assume, the alphabet consists of the letters A, B, C, D, and E.

## A Naive Solution

Let's assume, the alphabet consists of the letters A, B, C, D, and E.

Since there are 5 characters, we need at least 3 bits to uniquely encode each of them. So, we may randomly assign 3-bit binary strings from **000** to **100**

## A Naive Solution

Let's assume, the alphabet consists of the letters A, B, C, D, and E.

Since there are 5 characters, we need at least 3 bits to uniquely encode each of them. So, we may randomly assign 3-bit binary strings from **000** to **100**

Doing so, we obtain the following table.

Letter	Binary Code
A	000
B	001
C	010
D	011
E	100

## A Naive Solution

Letter	Binary Code
A	000
B	001
C	010
D	011
E	100

- Each of the letters needs 3 bits to encode.
- Therefore, if the length of a message =  $n$ , the expected length of the binary code =  $3n$

# We Can Do Better...

Here, as a saviour, arrives the ingenious idea of Huffman

# Huffman's Idea

## The Observation

- Assigning fewer bits for characters that are more frequent decreases the average length of the encoded message
- The paths from the root to the leaves of a tree always correspond to a unique prefix-free-encoding



# Huffman's Idea

## A Greedy Algorithm

- 1 Maintain a min-heap of nodes with their relative frequencies as weights. For instance, if the relative appearance of a character is 17%, we assign the weight of that node to be 0.17
- 2 Pop two nodes with minimum weights from the min-heap.
- 3 Merge the nodes into a single node with the weight as the summation of the previous nodes.
- 4 Push the newly created node back into the priority queue
- 5 Repeat from Step 2 until the priority queue contains a single node, the root of the tree.
- 6 Use the Huffman tree to encode any message

# Why Greedy Technique Works in Huffman Coding

## The Intuition Behind

- The greedy approach consistently selects the least frequent symbols and places them at the deepest position of the tree, ensuring that longer codes are assigned to less probable symbols and vice versa.
- The construction of Huffman codes follows a bottom-up greedy approach, where the optimal solution is built iteratively.

# The Huffman Tree

## Huffman Tree

In Huffman encoding, symbols are represented by nodes in a binary tree structure. It is built using bottom-up greedy approach.

## Properties of Huffman Tree

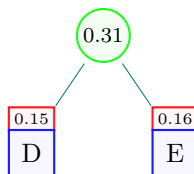
- Symbols are represented by **leaves** in a binary tree structure.
- Each symbol in the tree has a **unique path** from the root to its corresponding leaf node.
- The path consists of left (0) and right (1) branches, **uniquely** identifying each symbol.
- Ensures **prefix-free** encoding.

# Construction of Huffman Tree



Minimum-weighted two nodes. Now these will be combined...

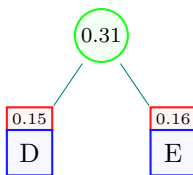
# Construction of Huffman Tree



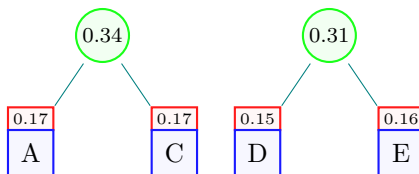
# Construction of Huffman Tree



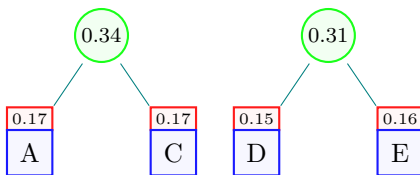
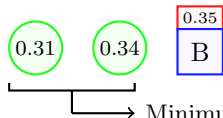
Minimum-weighted two nodes. Now these will be combined...



# Construction of Huffman Tree

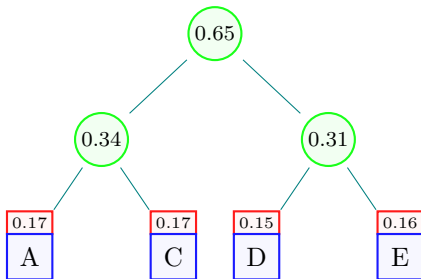


# Construction of Huffman Tree

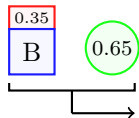




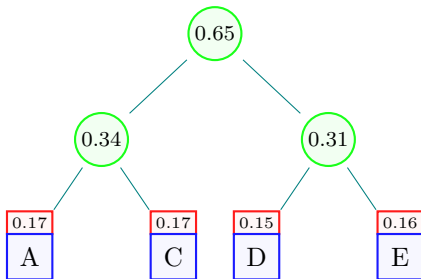
# Construction of Huffman Tree



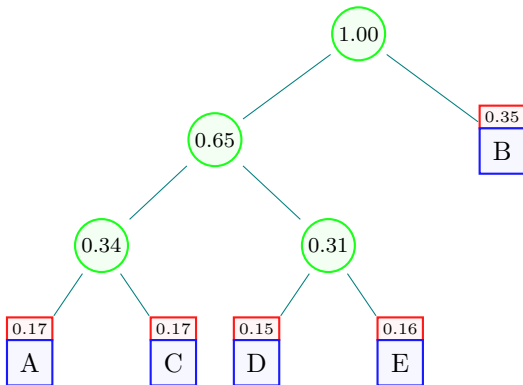
# Construction of Huffman Tree



Minimum-weighted two nodes. Now these will be combined...



# Construction of Huffman Tree



# Interpretation of Huffman Tree

## Example

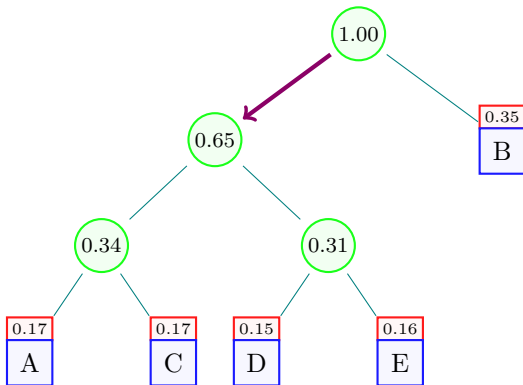
Let's say we want to find the binary encoding of the letter 'D'

## Solution

We find the path from root to the leaf containing 'D'. For every left branch, we append a **0** and for every right branch, we append a **1** to the binary encoding.

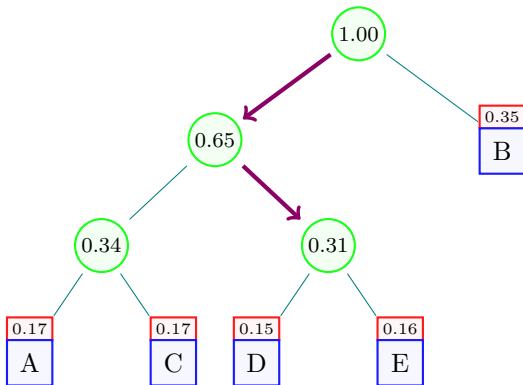
# Interpretation of Huffman Tree

0



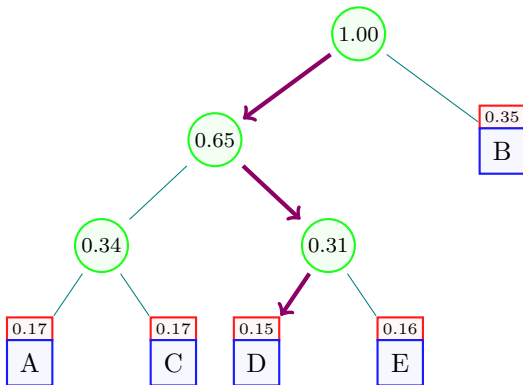
# Interpretation of Huffman Tree

0 1



# Interpretation of Huffman Tree

0 1 0



# Interpretation of Huffman Tree

So, the binary encoding of 'D' = **010**



# Interpretation of Huffman Tree

Similarly, we determine all the codes...

Letter	Binary Code
A	000
B	1
C	001
D	010
E	011

## We have indeed done better ...

- Now let's calculate the average length of the binary encoding for a message.
- Average Length =  $0.17 \times 3 + 0.35 \times 1 + 0.17 \times 3 + 0.15 \times 3 + 0.16 \times 3$   
 $= 2.3$
- Compression ratio =  $\frac{3-2.3}{3} \times 100\% = \mathbf{23.33\%}$

# How to Encode?

Encoding is easy, isn't it?

**Idea:** We may just look up the binary code for each of the letters and append them one by one to the resulting encoding.

# Example

## Example

Let's find the encoding for the word 'DECADE'.

# Example

Letter	Binary Code
A	000
B	1
C	001
D	010
E	011

D E C A D E

010 011 001 000 010 011

# Example

Letter	Binary Code
A	000
B	1
C	001
D	010
E	011

D E C A D E

010 011 001 000 010 011

# Example

Letter	Binary Code
A	000
B	1
C	001
D	010
E	011

D E C A D E

010 011 001 000 010 011

# Example

Letter	Binary Code
A	000
B	1
C	001
D	010
E	011

D E C A D E

010 011 001 000 010 011



# Example

Letter	Binary Code
A	000
B	1
C	001
D	010
E	011

D E C A D E

010 011 001 000 010 011

# Example

Letter	Binary Code
A	000
B	1
C	001
D	010
E	011

D E C A D E

010 011 001 000 010 011

# Efficiency of Encoding

$$\text{Entropy, } H = - \sum_{i=1}^n p_i \cdot \log_2(p_i)$$

Here  $p_i$  denotes the probability of  $i$ th symbol. For our word "DECODE", we calculate entropy as follows-

$$\begin{aligned} H = & - (0.17 \cdot \log_2(0.17) \\ & + 0.35 \cdot \log_2(0.35) \\ & + 0.17 \cdot \log_2(0.17) \\ & + 0.15 \cdot \log_2(0.15) \\ & + 0.16 \cdot \log_2(0.16)) = 2.23 \end{aligned}$$

# Efficiency of Encoding

Encoding of our word "DECADE" is- 010 011 001 000 010 011

# How to Decode?

Decoding is a bit more interesting...

**Idea:** Keep scanning the binary string from *left to right* and upon decoding a letter, we append it to the result.

## How to Decode?

Decoding is a bit more interesting...

**Idea:** Keep scanning the binary string from *left to right* and upon decoding a letter, we append it to the result.

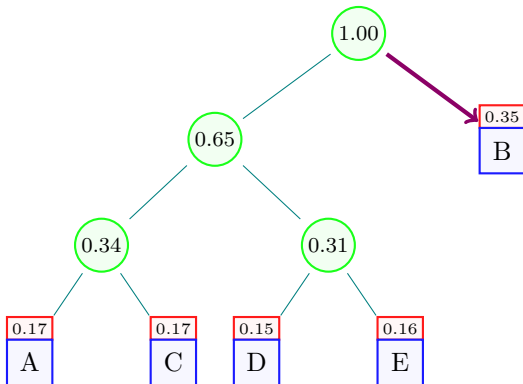
We may think of scanning the input as traversing the Huffman-Tree. When we read a **0**, we turn **left** and when **1**, we turn **right**. Upon reaching a leaf, we add the corresponding letter to our result.

# Example

Let's now decode '1011011'.

# Example

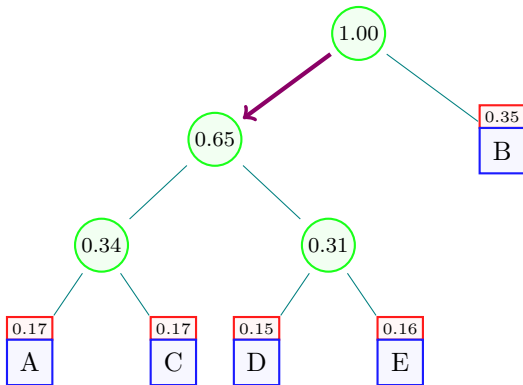
1 011011 B





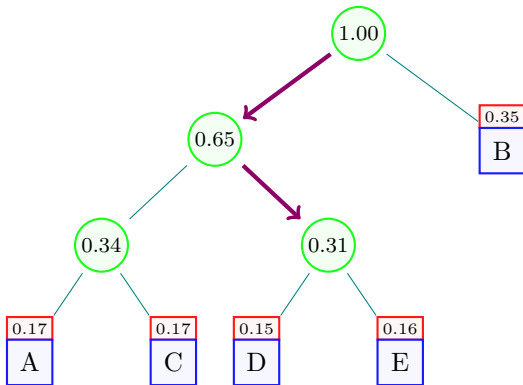
# Example

1 0 11011 B



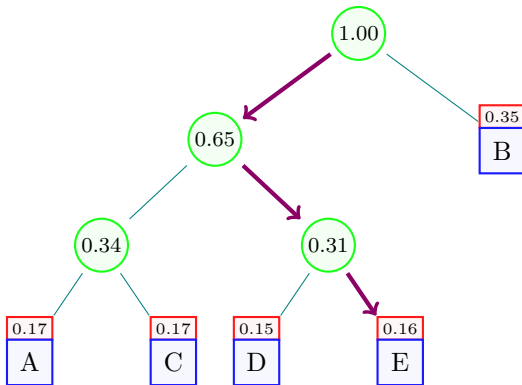
# Example

1 01 1011 B



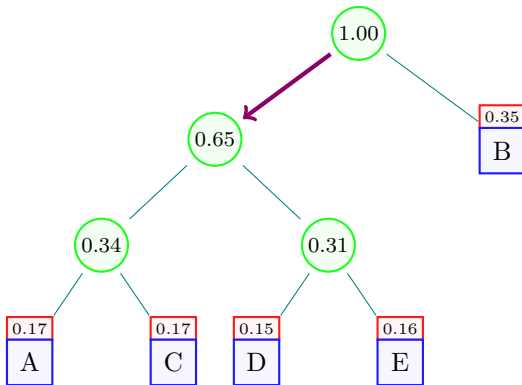
# Example

1 011 011      B E



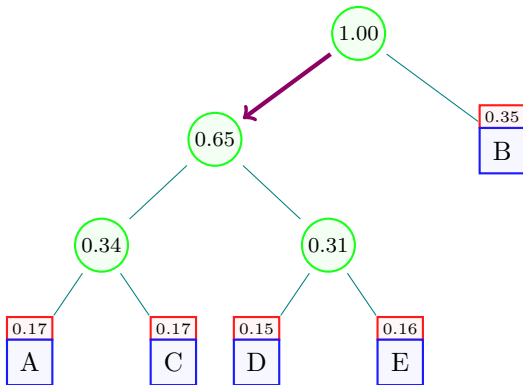
# Example

1011 0 11 BE



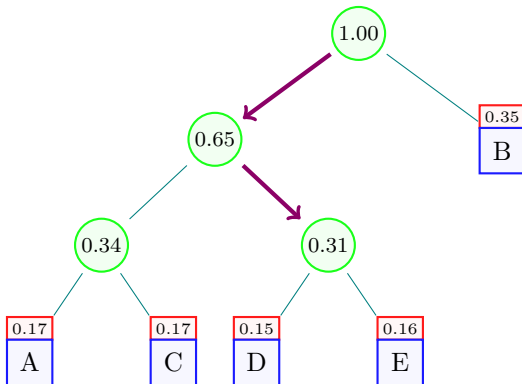
# Example

1011 0 11 BE



# Example

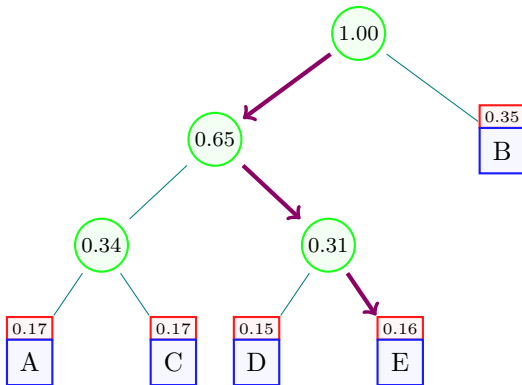
1011 01 1 BE



# Example

1011 011

BE E



# Example

So, decoded message = '**BEE**'