

DATA COMPRESSION

Prof. Darrell Long
CSE 13S

3/9/20

© 2023 Darrell Long

I



EXAMPLES OF DATA COMPRESSION

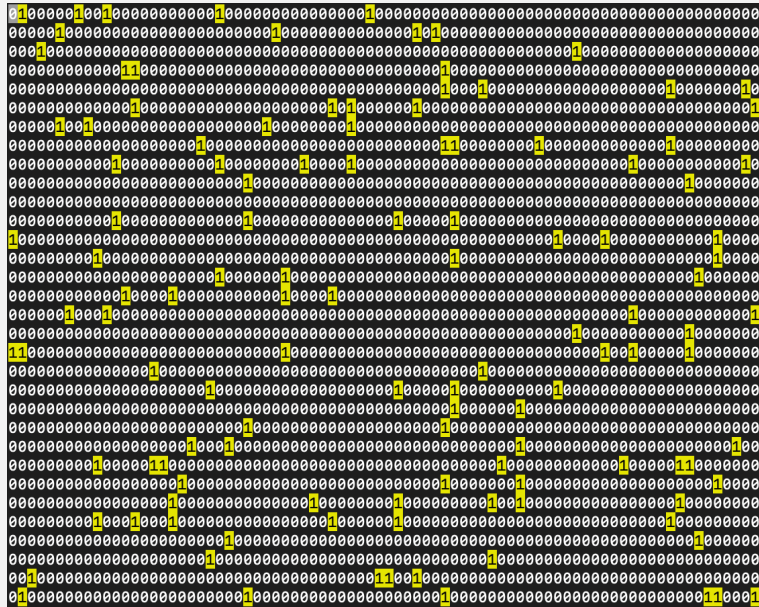


Audio Compression

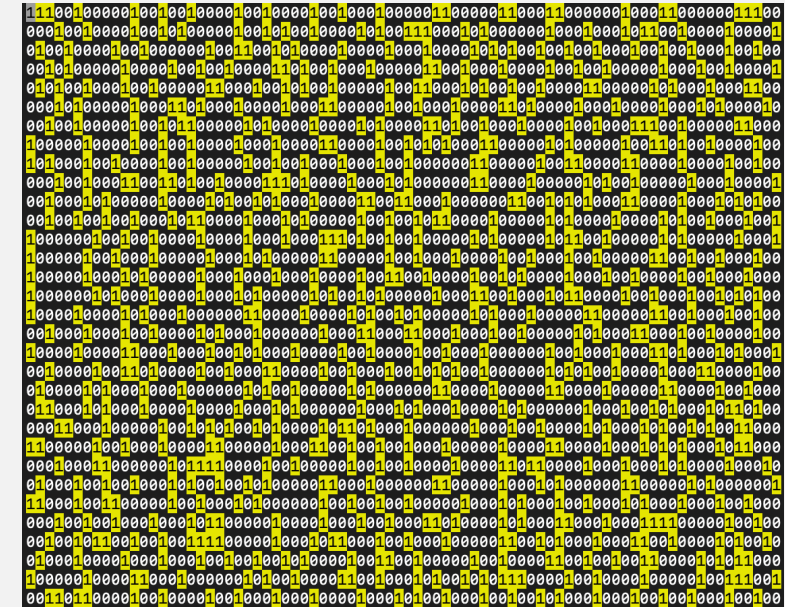


Image Compression

EXAMPLES OF DATA COMPRESSION

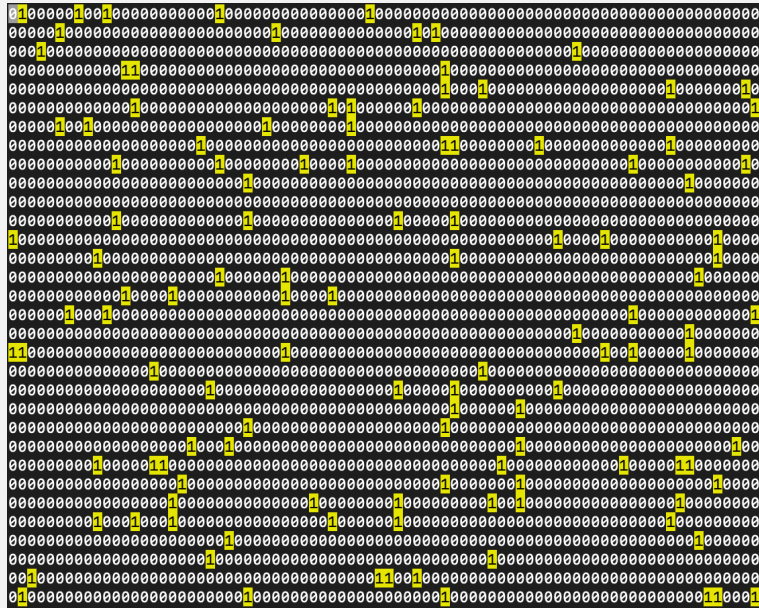
A large block of binary data with a 5% bit density, showing a single '1' bit for every 19 zero bits. The data is represented as a dense grid of '0's and '1's, where '1's are sparse and appear at regular intervals.

Binary Data with 5% "Bit Density"
(a single one bit for every 19 zero bits)

The same binary data after compression using a "Presence Bits" algorithm. The data is represented as a dense grid of '0's and '1's, where '1's are much more frequent than in the original data, indicating a significant reduction in the number of zero bits.

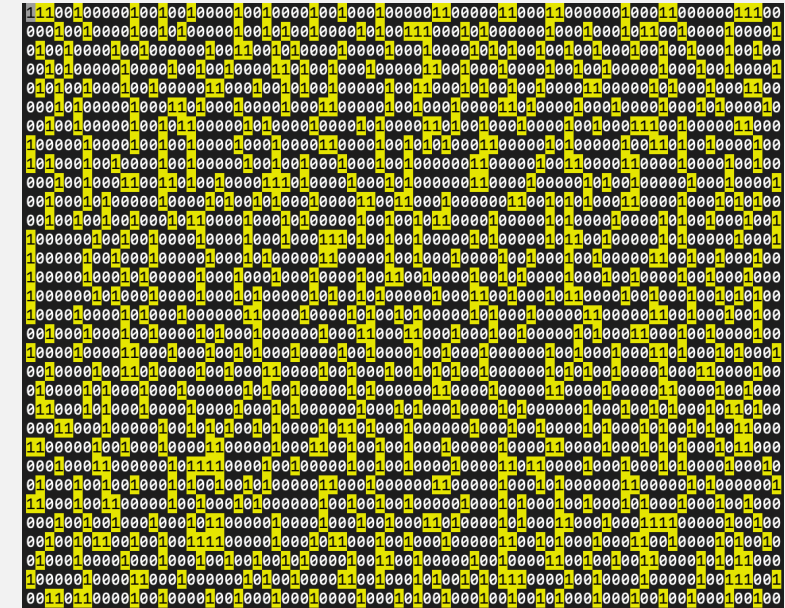
After Compression
Using a "Presence Bits" Algorithm

EXAMPLES OF DATA COMPRESSION



A large block of binary data with a 5% bit density, appearing as a sparse field of 1s and 0s. The data is organized into a grid-like structure with many rows and columns.

Binary Data with 5% "Bit Density"
800,000 bits



A large block of binary data after compression, showing a significantly reduced size compared to the original data. The data is organized into a grid-like structure with many rows and columns.

After Compression
232,000 bits

"PRESENCE BITS" DATA COMPRESSION

[illegible]

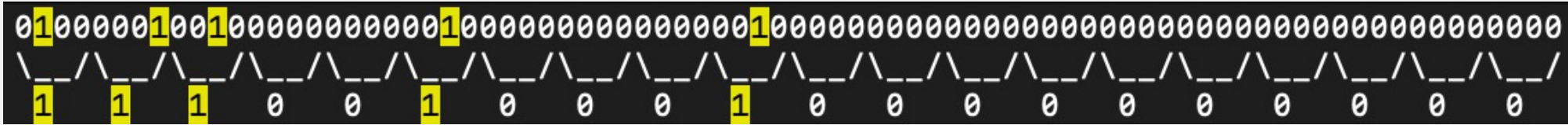
Binary Data with 5% "Bit Density"
(a single one bit for every 19 zero bits)

"PRESENCE BITS" DATA COMPRESSION

[illegible]

Divide into 4-bit "nibbles"

"PRESENCE BITS" DATA COMPRESSION



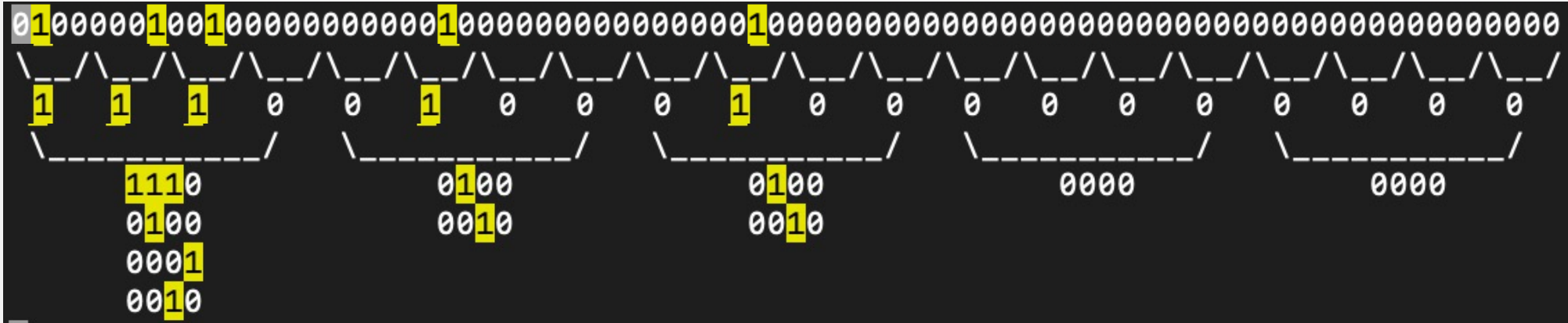
Identify nibbles that are not 0000

"PRESENCE BITS" DATA COMPRESSION



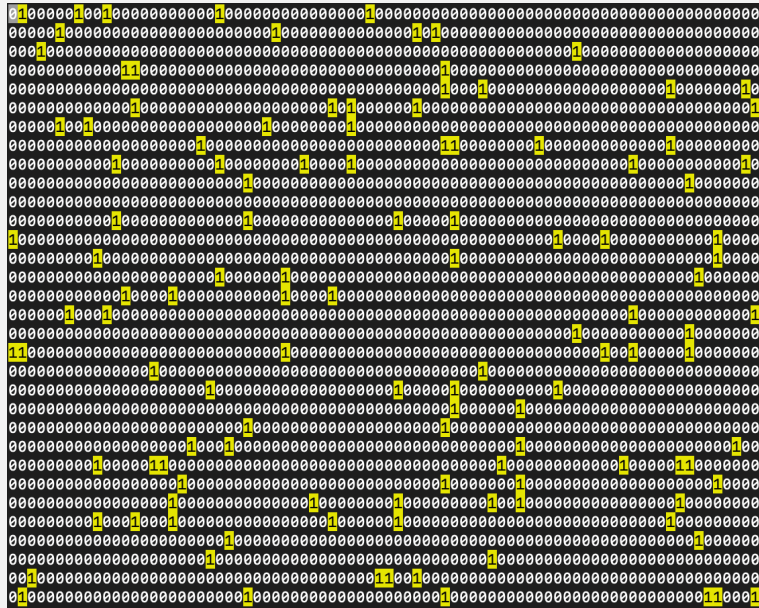
Group presence bits into "presence nibbles"

"PRESENCE BITS" DATA COMPRESSION



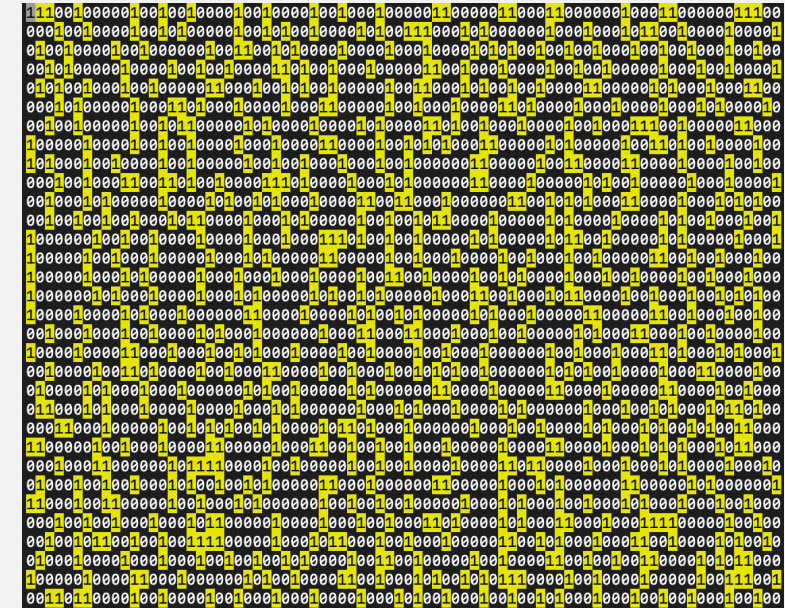
Write each "presence nibble" and just the non-0000 nibbles

"PRESENCE BITS" DATA COMPRESSION



A large block of binary data with a 5% bit density, showing a sparse distribution of 1s among many 0s. The data is presented as a single long line of characters.

Binary Data with 5% "Bit Density"
(a single one bit for every 19 zero bits)

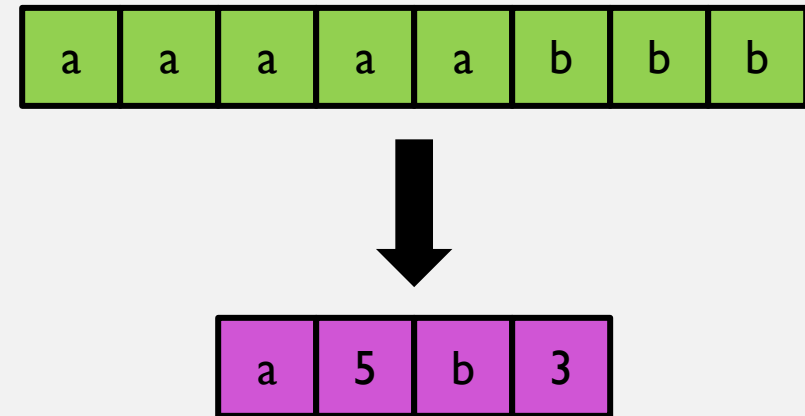


The same binary data after compression using the "Presence Bits" algorithm. The data is significantly more compact, with the 1s being more densely packed and the 0s being represented more efficiently.

After Compression using a
"Presence Bits" Algorithm

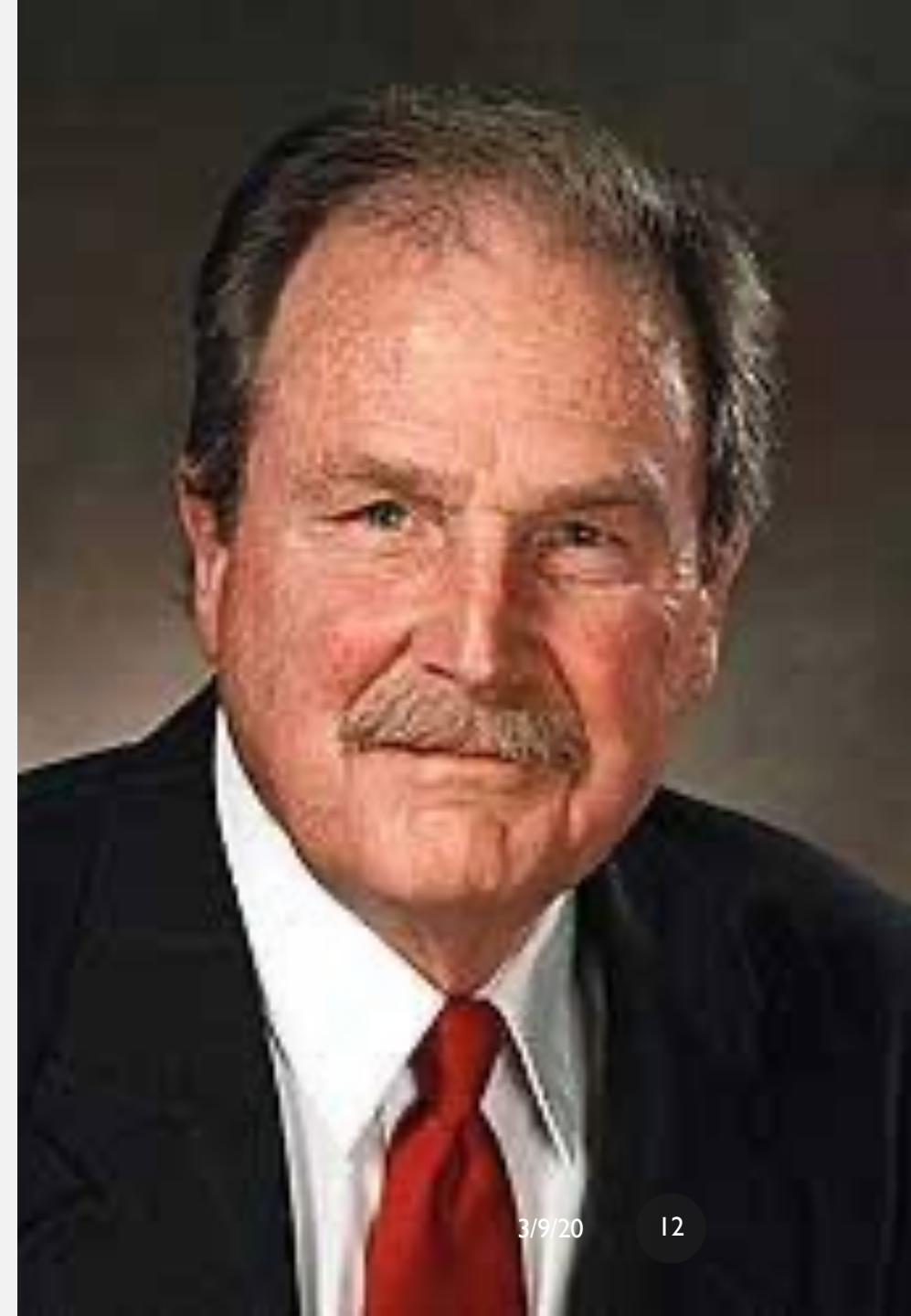
RUN-LENGTH CODING

- Idea: Split up a message into sequences of identical symbols.
 - These sequences are referred to as runs.
 - Runs are represented as a single instance of the repeated value followed by a repetition count.
- Great for files containing long runs of repeated data.



HUFFMAN CODING

- Developed by David A. Huffman.
 - Distinguished member of the UCSC faculty.
 - One of the founders of UCSC's Computer Science department.
- Idea: Assign each symbol a unique bit-string code.
 - Generate a histogram of unique symbols in data.
 - The more frequently a symbol appears, the shorter its code.
- So how does it work?



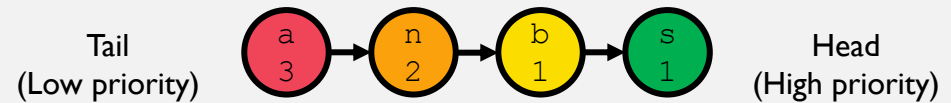
CONSTRUCTING A HISTOGRAM

- We'll compress the message "bananas".
- First, we create a histogram of the message's unique symbol and their frequencies.
- We assume that each symbol is an ASCII character.
- We can represent this histogram as an array with 256 indices.

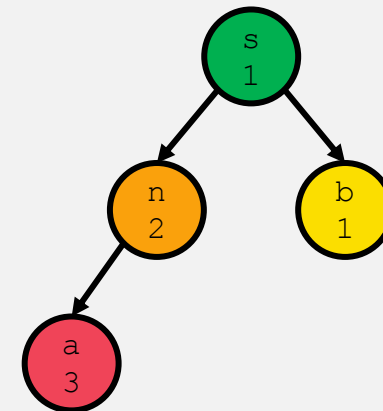
Symbol	Frequency
0	0
...	
a = 97	3
b = 98	1
...	0
n = 110	2
...	
s = 115	1
255	0

PRIORITY QUEUING

- Each symbol with a non-zero frequency is added to a priority queue as a node.
- The lower the frequency, the higher the priority.
- In a priority queue, the only guarantee is that the highest priority element is first.
 - The order of the rest is not well defined.
 - We can provide this guarantee by:
 1. Sorting the queue.
 2. Using a heap.

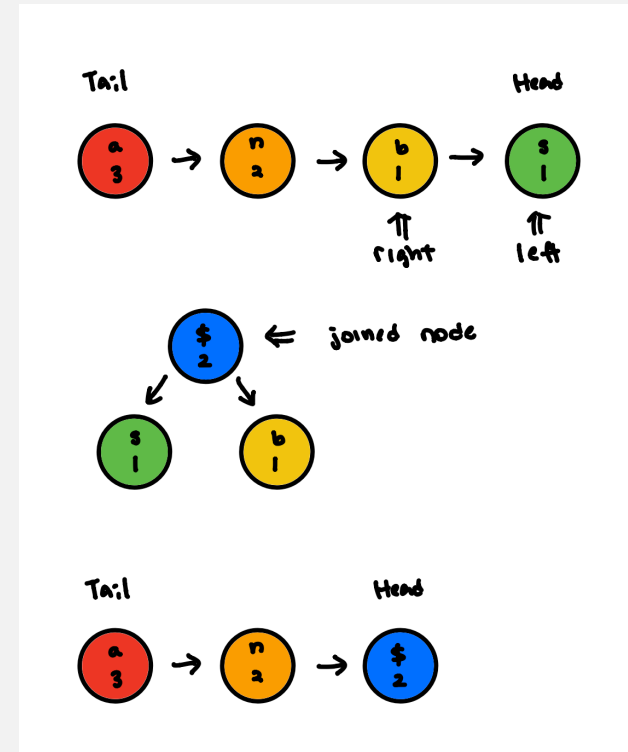


Root (High priority)



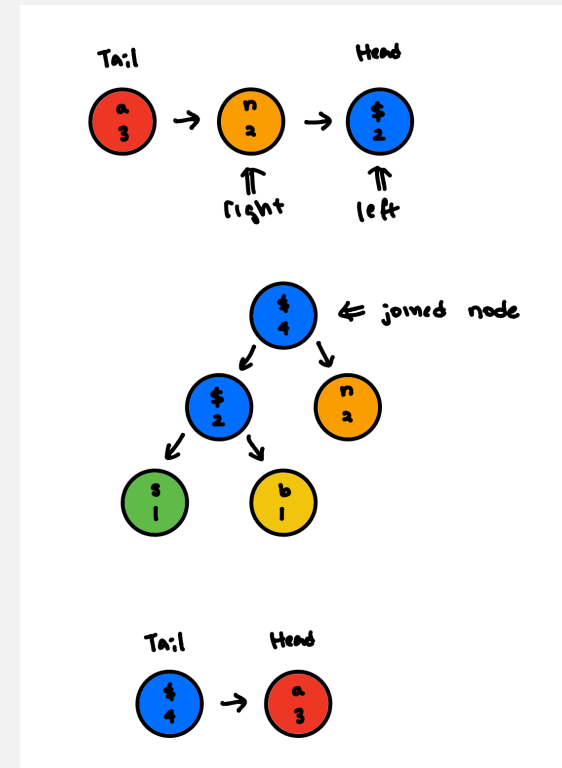
BUILDING A HUFFMAN TREE

- We will create a Huffman tree using the nodes in the priority queue.
- While the queue has more than one node,
 1. Dequeue a node. This will be the left child node.
 2. Dequeue another node. This will be the right child node.
 3. Create a parent node for the left and right child nodes. The frequency of the parent node is the sum of its children's frequencies.
 4. Enqueue the parent node.
- The last node in the queue is the root of the tree.



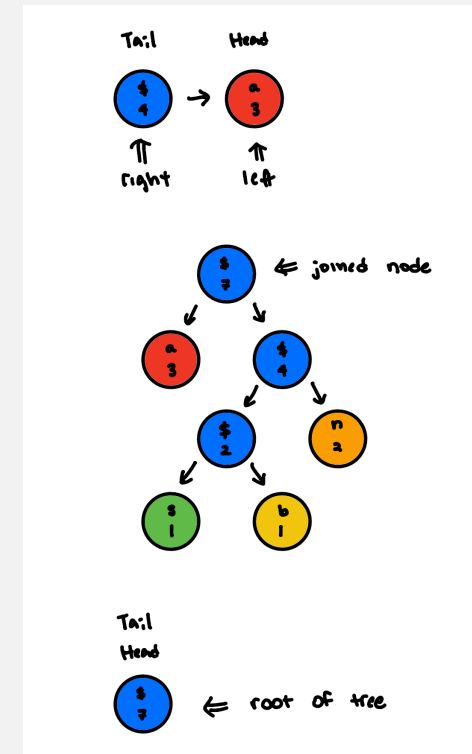
BUILDING A HUFFMAN TREE

- We will create a Huffman tree using the nodes in the priority queue.
- While the queue has more than one node,
 1. Dequeue a node. This will be the left child node.
 2. Dequeue another node. This will be the right child node.
 3. Create a parent node for the left and right child nodes. The frequency of the parent node is the sum of its children's frequencies.
 4. Enqueue the parent node.
- The last node in the queue is the root of the tree.



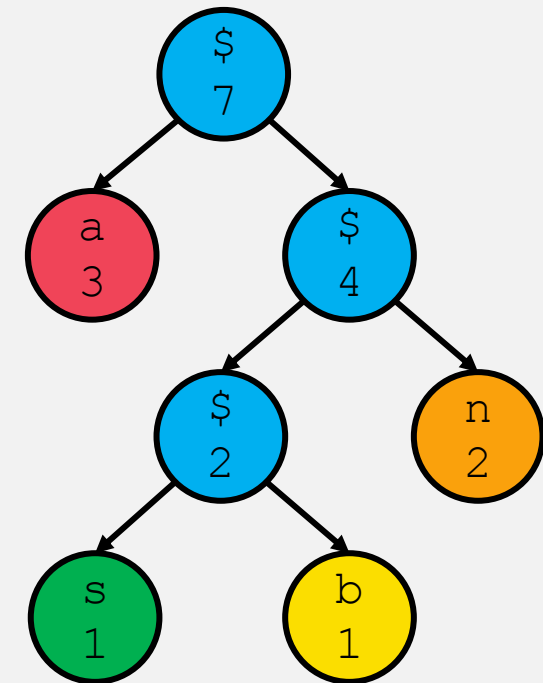
BUILDING A HUFFMAN TREE

- We will create a Huffman tree using the nodes in the priority queue.
- While the queue has more than one node,
 1. Dequeue a node. This will be the left child node.
 2. Dequeue another node. This will be the right child node.
 3. Create a parent node for the left and right child nodes. The frequency of the parent node is the sum of its children's frequencies.
 4. Enqueue the parent node.
- The last node in the queue is the root of the tree.



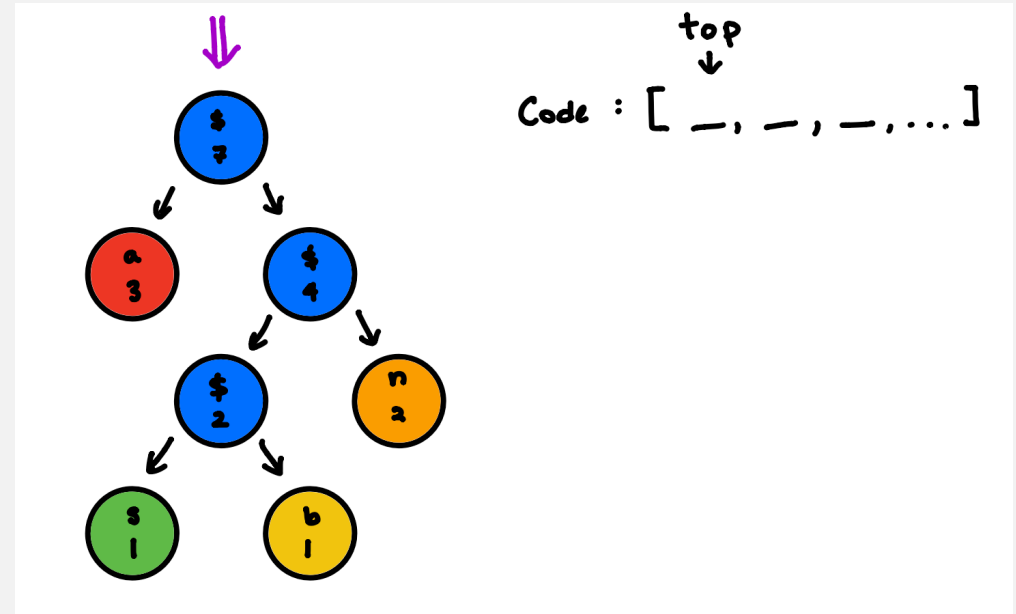
BUILDING A HUFFMAN TREE

- We will create a Huffman tree using the nodes in the priority queue.
- While the queue has more than one node,
 1. Dequeue a node. This will be the left child node.
 2. Dequeue another node. This will be the right child node.
 3. Create a parent node for the left and right child nodes. The frequency of the parent node is the sum of its children's frequencies.
 4. Enqueue the parent node.
- The last node in the queue is the root of the tree.



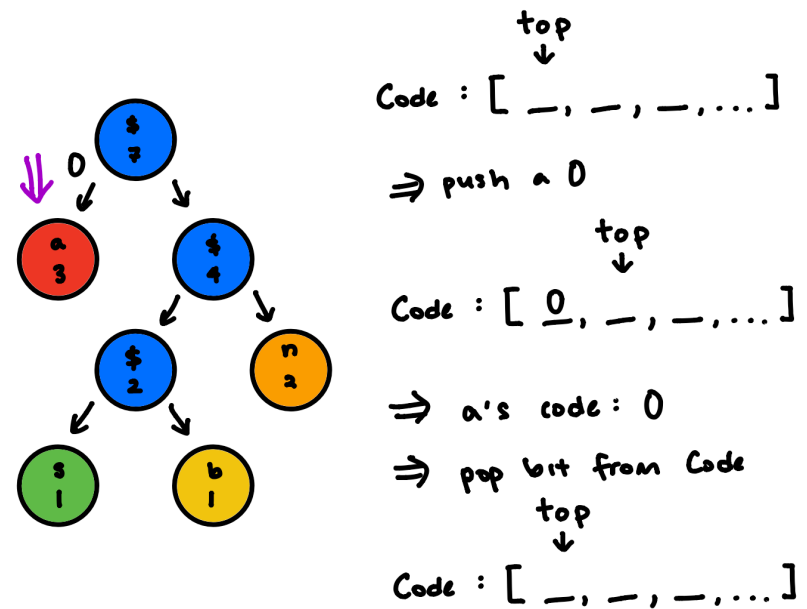
BUILDING THE CODE TABLE

- We now populate a table of codes.
 - Like the histogram there are 256 indices, and each index stores a binary code.
- We will utilize a stack of bits to keep track of the code.
- To build these codes, we perform a *post-order traversal*.
 - If we walk down to the left child, we push a 0 to the bit-stack.
 - If we walk down to the right child, we push a 1 to the bit-stack.
 - If we reach a leaf node, the code for the node's symbol is the code in the bit-stack.
- A code can be, at maximum, 256 bits long!



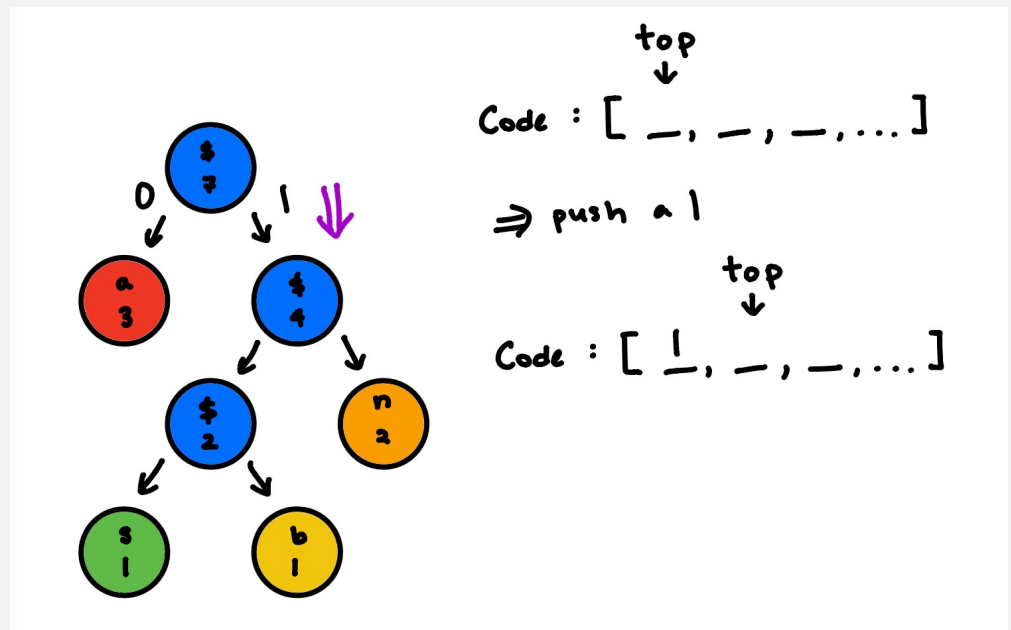
BUILDING THE CODE TABLE

- We now populate a table of codes.
 - Like the histogram there are 256 indices, and each index stores a binary code.
- We will utilize a stack of bits to keep track of the code.
- To build these codes, we perform a *post-order traversal*.
 - If we walk down to the left child, we push a 0 to the bit-stack.
 - If we walk down to the right child, we push a 1 to the bit-stack.
 - If we reach a leaf node, the code for the node's symbol is the code in the bit-stack.
- A code can be, at maximum, 256 bits long!



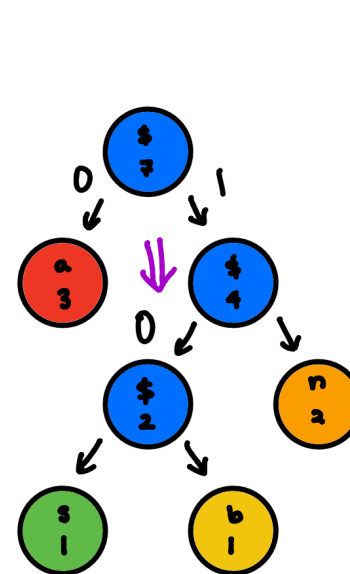
BUILDING THE CODE TABLE

- We now populate a table of codes.
 - Like the histogram there are 256 indices, and each index stores a binary code.
- We will utilize a stack of bits to keep track of the code.
- To build these codes, we perform a *post-order traversal*.
 - If we walk down to the left child, we push a 0 to the bit-stack.
 - If we walk down to the right child, we push a 1 to the bit-stack.
 - If we reach a leaf node, the code for the node's symbol is the code in the bit-stack.
- A code can be, at maximum, 256 bits long!



BUILDING THE CODE TABLE

- We now populate a table of codes.
 - Like the histogram there are 256 indices, and each index stores a binary code.
- We will utilize a stack of bits to keep track of the code.
- To build these codes, we perform a *post-order traversal*.
 - If we walk down to the left child, we push a 0 to the bit-stack.
 - If we walk down to the right child, we push a 1 to the bit-stack.
 - If we reach a leaf node, the code for the node's symbol is the code in the bit-stack.
- A code can be, at maximum, 256 bits long!



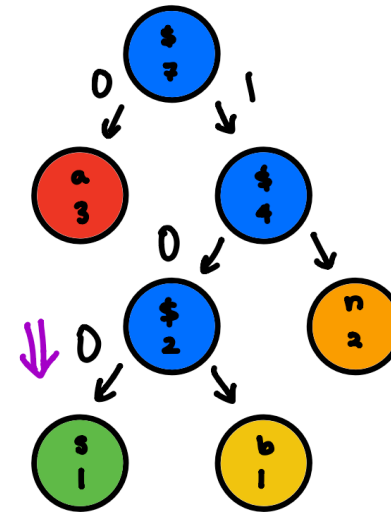
Code : [$\overset{\text{top}}{\downarrow} 1, -, -, \dots]$

\Rightarrow push a 0

Code : [$\overset{\text{top}}{\downarrow} 1, 0, -, \dots]$

BUILDING THE CODE TABLE

- We now populate a table of codes.
 - Like the histogram there are 256 indices, and each index stores a binary code.
- We will utilize a stack of bits to keep track of the code.
- To build these codes, we perform a *post-order traversal*.
 - If we walk down to the left child, we push a 0 to the bit-stack.
 - If we walk down to the right child, we push a 1 to the bit-stack.
 - If we reach a leaf node, the code for the node's symbol is the code in the bit-stack.
- A code can be, at maximum, 256 bits long!



Code : [1, 0, —, ...]

⇒ push a 0

Code : [1, 0, 0, ...]

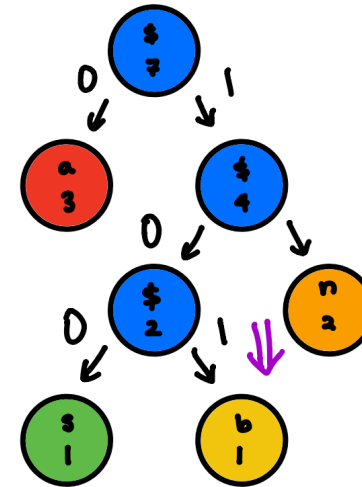
⇒ s's code: 100

⇒ pop bit from Code

Code : [1, 0, —, ...]

BUILDING THE CODE TABLE

- We now populate a table of codes.
 - Like the histogram there are 256 indices, and each index stores a binary code.
- We will utilize a stack of bits to keep track of the code.
- To build these codes, we perform a *post-order traversal*.
 - If we walk down to the left child, we push a 0 to the bit-stack.
 - If we walk down to the right child, we push a 1 to the bit-stack.
 - If we reach a leaf node, the code for the node's symbol is the code in the bit-stack.
- A code can be, at maximum, 256 bits long!



Code : [1, 0, —, ...]
top
↓

⇒ push a 1

Code : [1, 0, 1, ...]
top
↓

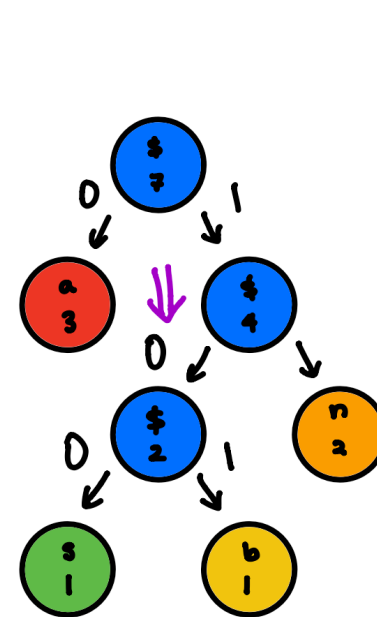
⇒ b's code: 101

⇒ pop bit from Code

Code : [1, 0, —, ...]
top
↓

BUILDING THE CODE TABLE

- We now populate a table of codes.
 - Like the histogram there are 256 indices, and each index stores a binary code.
- We will utilize a stack of bits to keep track of the code.
- To build these codes, we perform a *post-order traversal*.
 - If we walk down to the left child, we push a 0 to the bit-stack.
 - If we walk down to the right child, we push a 1 to the bit-stack.
 - If we reach a leaf node, the code for the node's symbol is the code in the bit-stack.
- A code can be, at maximum, 256 bits long!



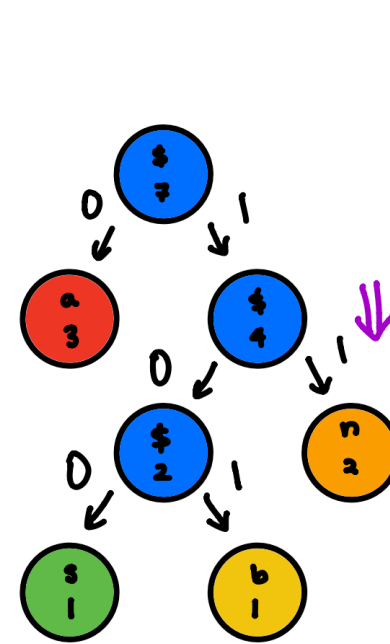
Code : [1, 0, top, ...]

⇒ pop bit from Code

Code : [1, top, top, ...]

BUILDING THE CODE TABLE

- We now populate a table of codes.
 - Like the histogram there are 256 indices, and each index stores a binary code.
- We will utilize a stack of bits to keep track of the code.
- To build these codes, we perform a *post-order traversal*.
 - If we walk down to the left child, we push a 0 to the bit-stack.
 - If we walk down to the right child, we push a 1 to the bit-stack.
 - If we reach a leaf node, the code for the node's symbol is the code in the bit-stack.
- A code can be, at maximum, 256 bits long!



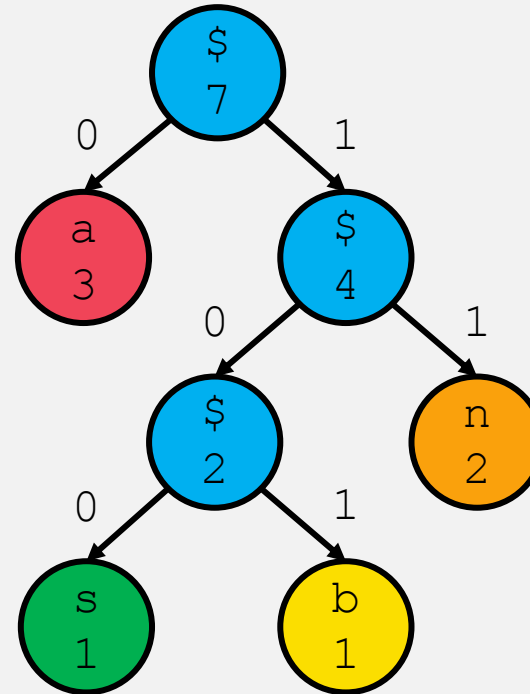
top
↓
Code : [1, -, -, ...]
⇒ push a 1

top
↓
Code : [1, 1, -, ...]
⇒ n's code : 11

⇒ process continues until
post-order traversal ends

BUILDING THE CODE TABLE

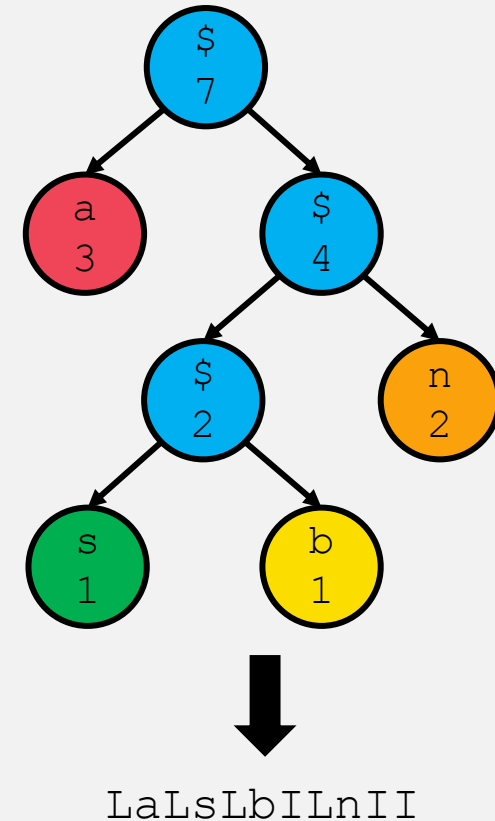
- We now populate a table of codes.
 - Like the histogram there are 256 indices, and each index stores a binary code.
- We will utilize a stack of bits to keep track of the code.
- To build these codes, we perform a *post-order traversal*.
 - If we walk down to the left child, we push a 0 to the bit-stack.
 - If we walk down to the right child, we push a 1 to the bit-stack.
 - If we reach a leaf node, the code for the node's symbol is the code in the bit-stack.
- A code can be, at maximum, 256 bits long!



Symbol	Code
0	
...	
a = 97	0
b = 98	101
...	
n = 110	11
...	
s = 115	100
255	

DUMPING THE TREE

- Some versions of Huffman coding emit the constructed tree.
 - The tree is dumped to the encoding through a post-order traversal.
- Performing a post-order traversal of the tree,
 - If a leaf node is reached, output an 'L' followed by the node's symbol.
 - If an interior node's children have been traversed, output an 'I' to indicate an interior node.
- The number of symbols representing the tree dump is $(3 \times \text{leaves}) - 1$.



EMITTING CODES

- For each symbol in the message, output its code.
- We use the constructed code table for fast symbol to code translation.
- It's a function – table: symbol \rightarrow code.

Symbol	Code
0	
...	
97	0
98	101
...	
110	11
...	
115	100
255	

bananas

↓

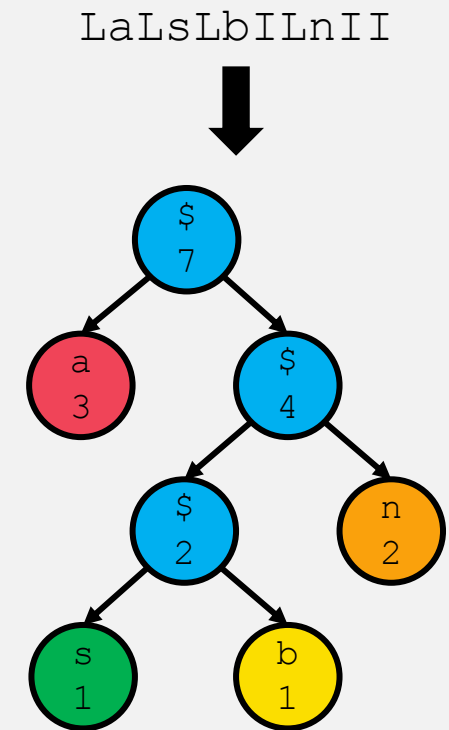
[101] [0] [11] [0] [11] [0] [100]

↓

1010110110100

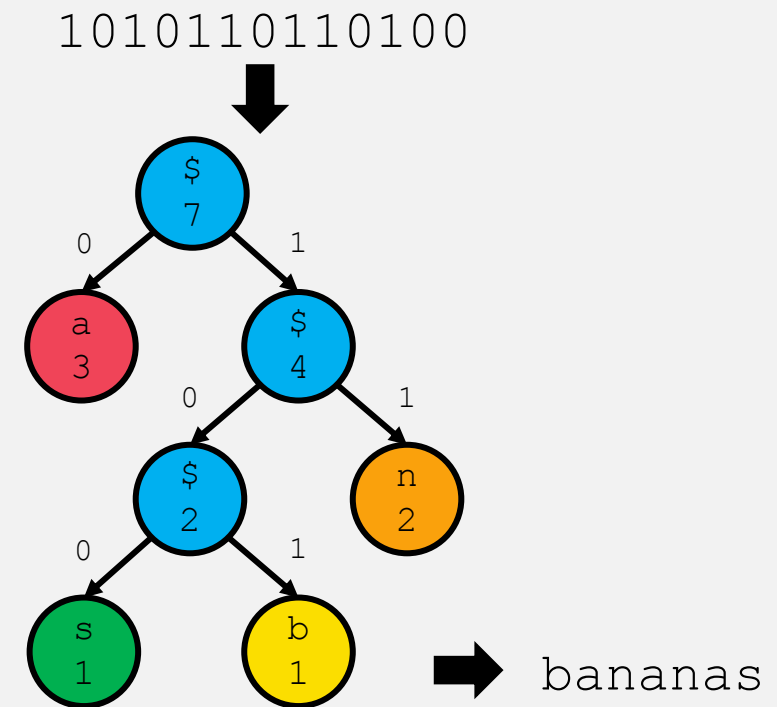
RECONSTRUCTING THE TREE

- To decompress the compressed message, we first reconstruct the tree from its dump.
 - Recall that the dump was performed through post-order traversal.
- Initialize a stack of nodes.
- While all the symbols of the dump haven't been processed,
 - If the current symbol is an 'L', then the next symbol in the dump is the leaf's symbol.
 - Put this symbol into a node and push it onto the node-stack.
 - If the current symbol is an 'I', then an interior node has been reached.
 - Pop the stack for the right child, then pop again for the left child.
 - Create the parent node for these children and push it onto the node-stack.



DECODING BIT-BY-BIT

- To decode the binary codes, we walk the tree.
- Read in each bit from the input.
 - If the bit is a 0, walk down to the left child.
 - If the bit is a 1, walk down to the right child.
 - If a leaf node is reached, output its symbol and restart from the root of the tree.



LEMPER-ZIV CODING (LZ78)

Developed by Abraham Lempel and Jacob Ziv.

An adaptive dictionary coder.

Dictionaries aren't output as part of encoding.

Dictionaries are built incrementally as encoded data is processed.

Idea: If a message isn't uniformly random, it is likely to contain recurring patterns.

Use a dictionary to store seen patterns and give them unique codes.

Terminology :

Word – a sequence of bytes.

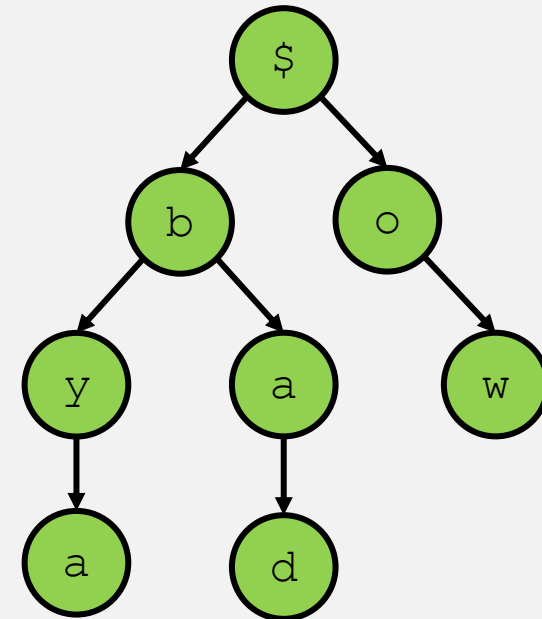
Code – an unsigned 16-bit integer that denotes a word.

Pair – a code followed by a symbol.



A DICTIONARY OF WORDS

- The best-case scenario for LZ78 is a message with words with long, common prefixes.
 - Which means, the larger the codes, the longer the words they denote.
- To store these words, we use a trie.
 - Named because it is an efficient information re-*trie*-val data structure.
 - Also called a prefix tree.
 - The trie node representing the end of a word stores the code for the word.
- Minimizes redundancies when storing many words with common prefixes.



LZ78 ENCODING

- The message to encode is “abababa”.
- Initialize the trie to just the root node.
 - It will be given the code `EMPTY = 1` since the root denotes the empty word.
- We will need to keep track of the current node, which is colored: ■.
- Nodes that are added to the trie at any step are colored: ■.
- All other uninteresting nodes are colored: ■.



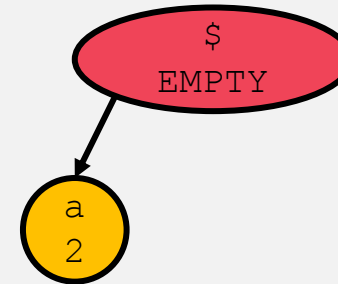
LZ78 ENCODING

- Message: "abababa".
- The first symbol, and thus the current symbol, to consider is 'a'.
- Check if the current node has the child 'a'.



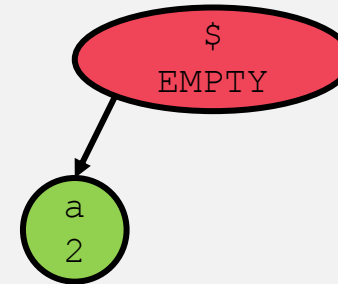
LZ78 ENCODING

- Since it doesn't, we add the child.
 - The code for the child is the next unused code.
 - In this case, the code is 2.
- Each time we come across a word that doesn't exist in the trie, we output a pair.
 - The pair is made from the current code and symbol.
 - So we output (EMPTY, 'a').
- We have just added a new word to the trie, so now we reset back to the root.



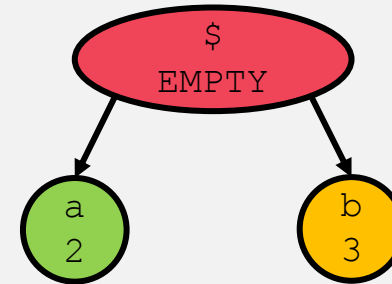
LZ78 ENCODING

- Message: "abababa".
- The current symbol to consider is 'b'.
- Check if the current node has the child 'b'.



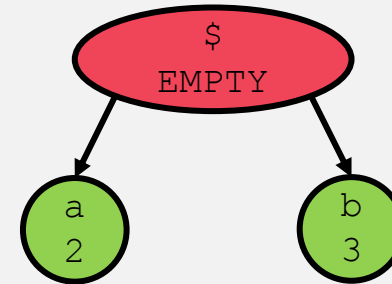
LZ78 ENCODING

- Since it doesn't, we add the child.
 - The code for the child is the next unused code.
 - In this case, the code is 3.
- We output the pair (EMPTY, 'b').
- We have just added a new word to the trie, so now we reset back to the root.



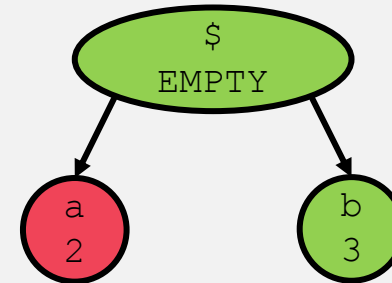
LZ78 ENCODING

- Message: "abababa".
- The current symbol to consider is 'a'.
- Check if the current node has the child 'a'.



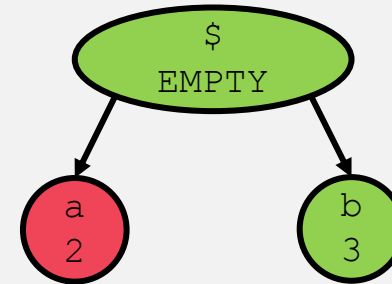
LZ78 ENCODING

- Since it does, we simply step down the trie.
 - The current node is now the child node containing 'a'.
- Since we haven't added anything, no pair is output.



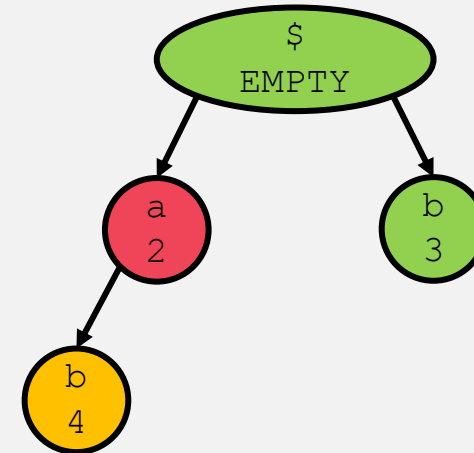
LZ78 ENCODING

- Message: "abababa".
- The current symbol to consider is 'b'.
- Check if the current node has the child 'b'.



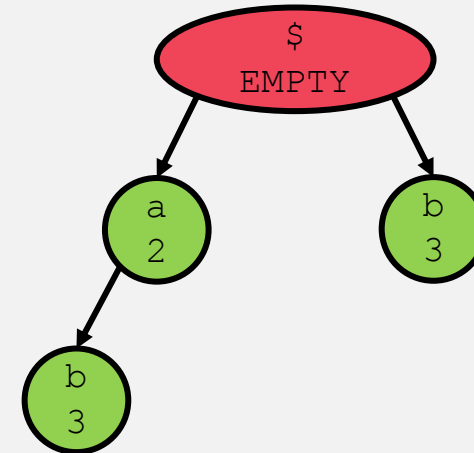
LZ78 ENCODING

- Since it doesn't, we add the child.
 - The code for the child is the next unused code.
 - In this case, the code is 4.
- We output the pair (2, 'b').
- We have just added a new word to the trie, so now we reset back to the root.



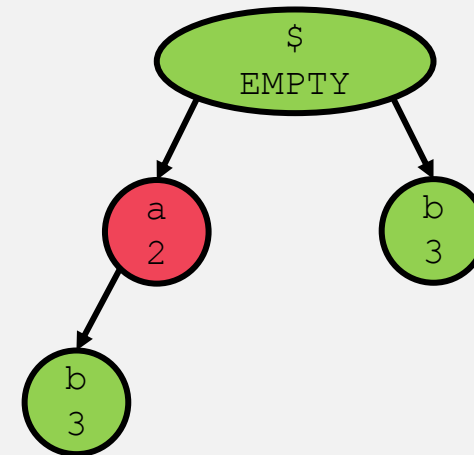
LZ78 ENCODING

- Message: "abababa".
- The current symbol to consider is 'a'.
- Check if the current node has the child 'a'.



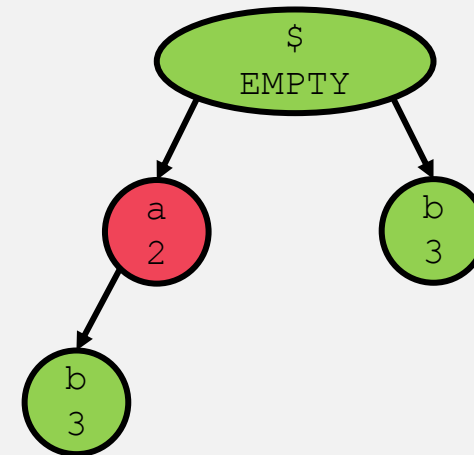
LZ78 ENCODING

- Since it does, we simply step down the trie.
 - The current node is now the child node containing 'a'.
- Since we haven't added anything, no pair is output.



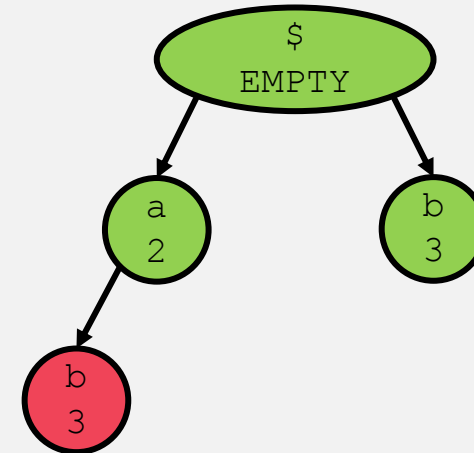
LZ78 ENCODING

- Message: "abababa".
- The current symbol to consider is 'b'.
- Check if the current node has the child 'b'.



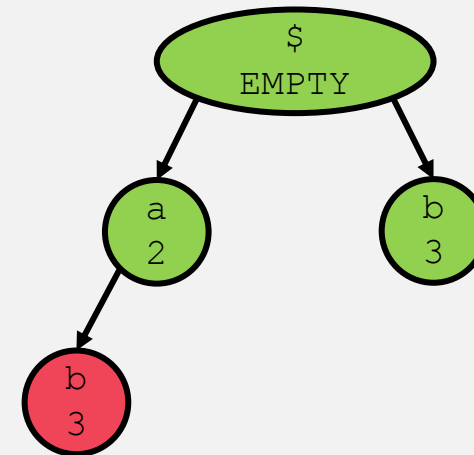
LZ78 ENCODING

- Since it does, we simply step down the trie.
 - The current node is now the child node containing 'b' .
- Since we haven't added anything, no pair is output.



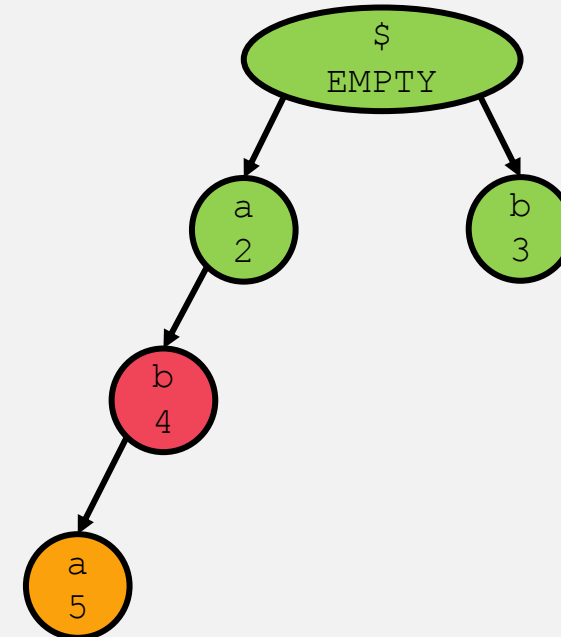
LZ78 ENCODING

- Message: "abababaa".
- The last symbol to consider is 'a'.
- Check if the current node has the child 'a'.



LZ78 ENCODING

- Since it doesn't, we add the child.
 - The code for the child is the next unused code.
 - In this case, the code is 5.
- We output the pair (4, 'a').
- The message has now been compressed to the following pairs, in order:
 1. (EMPTY, 'a')
 2. (EMPTY, 'b')
 3. (2, 'b')
 4. (4, 'a')



LZ78 DECODING

- To decompress, we decode the pairs, filling out a dictionary as we go.
 - The key is a code, and the value the word that the code denotes.
- Like with the trie used in compression, the empty word is added initially to the dictionary.
 - Compression and decompression must agree on the starting set of words and codes.

Code	Word
EMPTY	""

LZ78 DECODING

- **Pairs:**
 1. (EMPTY, 'a') ←
 2. (EMPTY, 'b')
 3. (2, 'b')
 4. (4, 'a')
- We append 'a' to the word denoted by EMPTY.
 - This yields the new word: "a".
 - We add this word to the dictionary, giving it the next unused code of 2.
- Output the word.
 - Current output: "a".

Code	Word
EMPTY	""
2	"a"

LZ78 DECODING

- **Pairs:**
 1. (EMPTY, 'a')
 2. (EMPTY, 'b') ←
 3. (2, 'b')
 4. (4, 'a')
- **We append 'b' to the word denoted by EMPTY.**
 - This yields the new word: "b".
 - We add this word to the dictionary, giving it the next unused code of 3.
- **Output the word.**
 - Current output: "ab".

Code	Word
EMPTY	""
2	"a"
3	"b"

LZ78 DECODING

- **Pairs:**
 1. (EMPTY, 'a')
 2. (EMPTY, 'b')
 3. (2, 'b') ←
 4. (4, 'a')
- **We append 'b' to the word denoted by 2.**
 - This yields the new word: "ab".
 - We add this word to the dictionary, giving it the next unused code of 4.
- **Output the word.**
 - Current output: "abab".

Code	Word
EMPTY	""
2	"a"
3	"b"
4	"ab"

LZ78 DECODING

- **Pairs:**
 1. (EMPTY, 'a')
 2. (EMPTY, 'b')
 3. (2, 'b')
 4. (4, 'a') ←
- We append 'a' to the word denoted by 4.
 - This yields the new word: "aba".
 - We add this word to the dictionary, giving it the next unused code of 5.
- Output the word.
 - Current output: "abababa".

Code	Word
EMPTY	""
2	"a"
3	"b"
4	"ab"
5	"aba"

SUMMARY

- Two lossless compression algorithms:
 1. Huffman Coding
 2. LZ78
- Knowing that messages exhibit varying levels of entropy, it is good to pick your tools wisely.
 - Would it be good to use LZ78 for messages with high entropy?
 - Would it be better to use Huffman over LZ78 for message with low entropy?
- Compression algorithms have their limits.
 - They can't hope to compress messages consisting of uniform randomness.

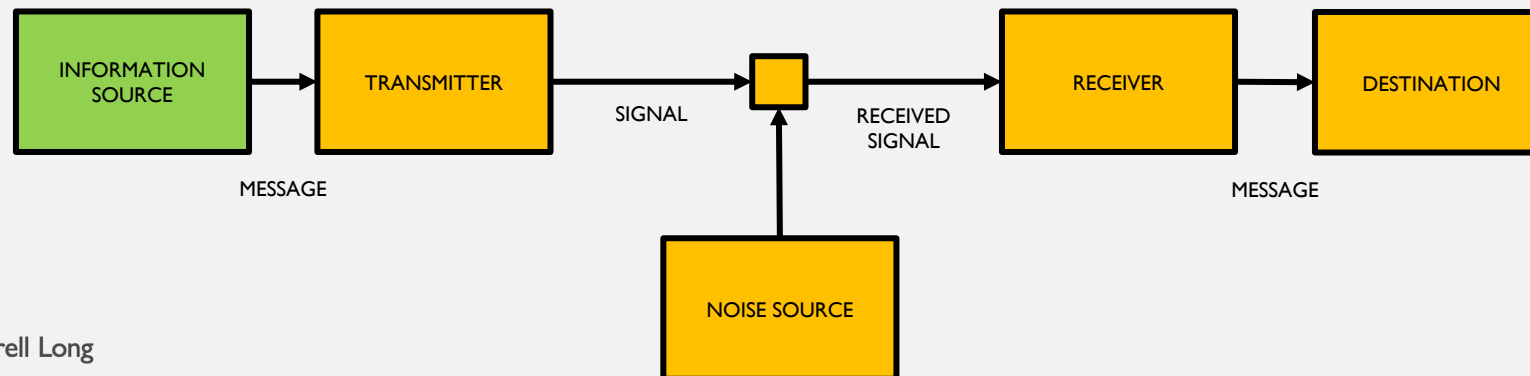


COMMUNICATION

- Claude Shannon, the “father of modern information theory,” breaks down communication into five parts:
 1. Information source
 2. Transmitter
 3. Channel
 4. Receiver
 5. Destination

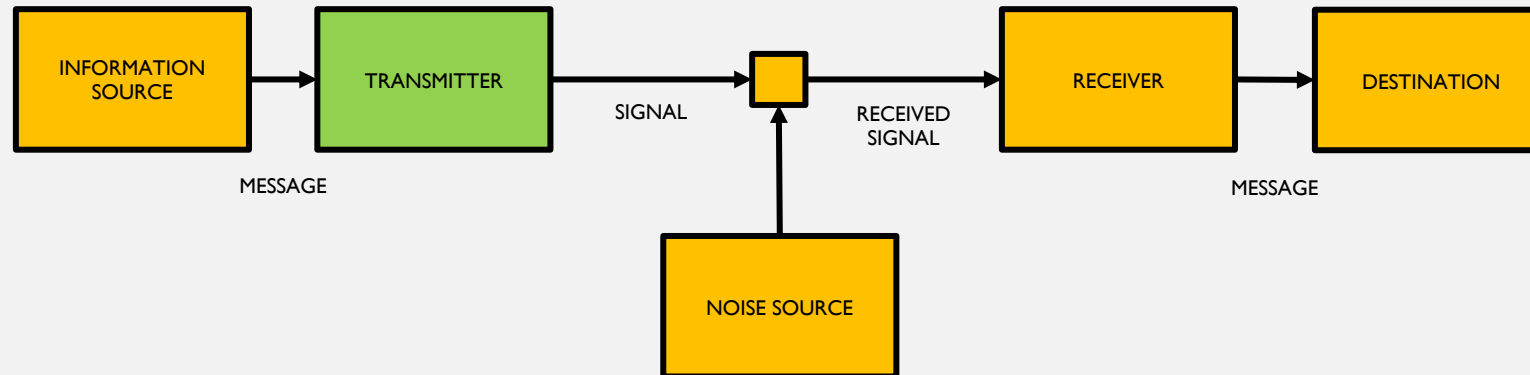
INFORMATION SOURCE

- Produces a message, or a sequence of messages, to be communicated to the receiving terminal.
- A message takes on various forms:
 - A sequence of symbols like in a telegraph or teletype system.
 - A single function of time like radio.
 - A function of time and other variables like black/white television.
 - Two or more functions of time like sound transmission (several channels).
 - Several functions of variables like color television.



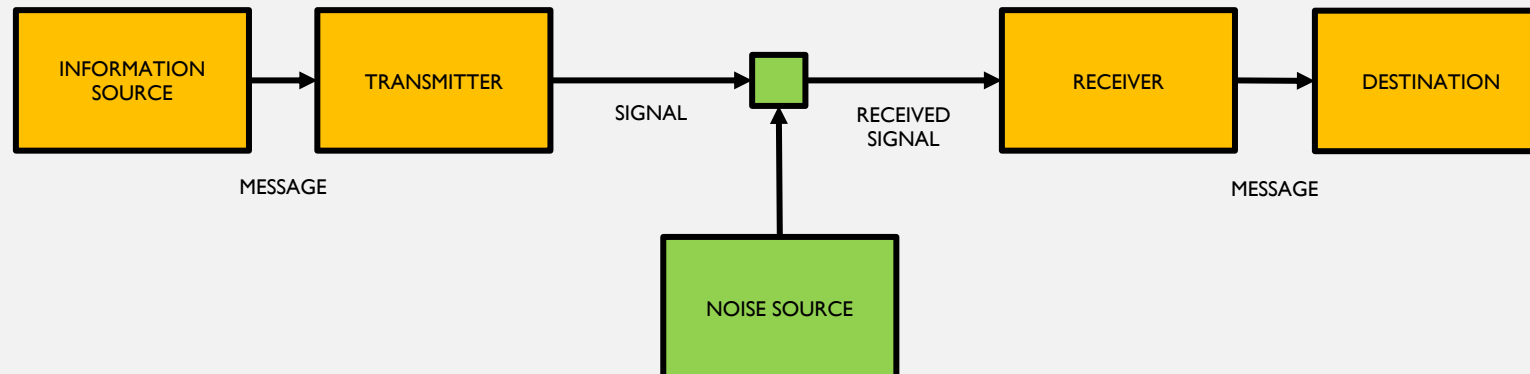
TRANSMITTER

- Operates on the message to produce a signal suitable for transmission over the channel.
- This is where compression and/or encryption occurs.



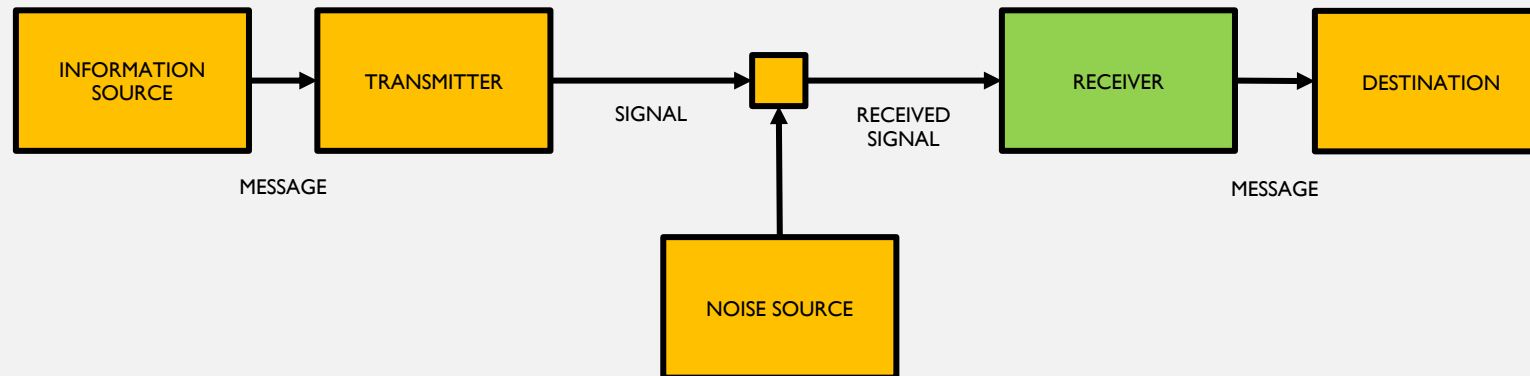
CHANNEL

- The medium through which a signal is transmitted.
- Could be, but not limited to:
 - Radio frequencies.
 - Beams of light.
 - Coaxial cables.
 - Wires.
 - Fiber optics.



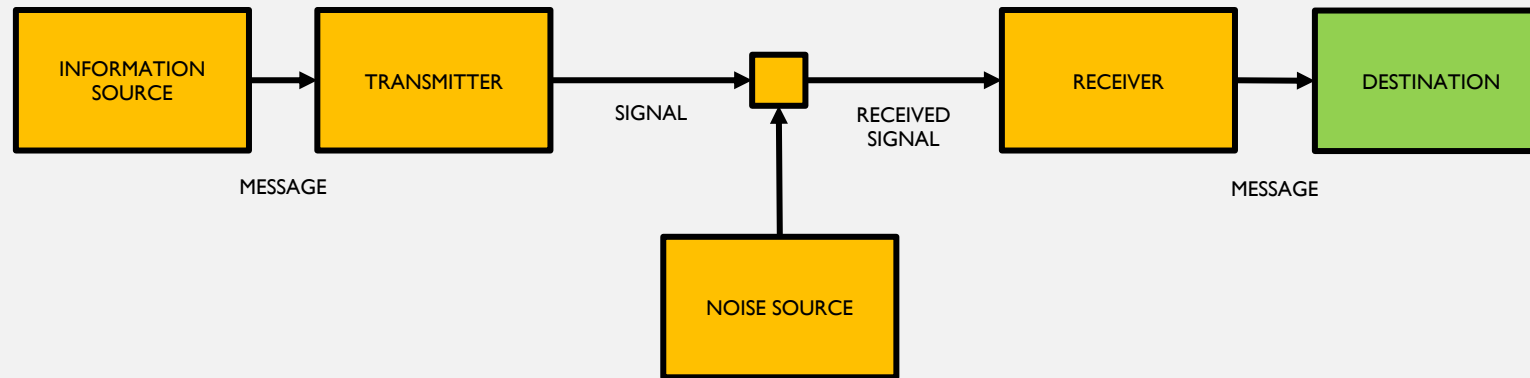
RECEIVER

- Performs the inverse operation of that done by the transmitter.
- Decompression if the transmitter used compression, decryption if the transmitter used encryption.



DESTINATION

- The intended target of the message.



ENTROPY

- Defined by Shannon as a measure of uncertainty of the occurrence of an event.
- Assume a set of possible events with probabilities $p = (p_1, p_2, \dots, p_n)$ such that $\sum_{i=1}^n p_i = 1$.
 - That is, the probability of event e_i occurring is p_i for all $1 \leq i \leq n$.
- If such a measure exists, call it $H(p_1, p_2, \dots, p_n)$, it can be required to have the following properties:
 1. H is continuous in p
 2. If all p_i equal, $p_i = \frac{1}{n}$, thus H is a monotonically increasing function of n .
 - There is some amount of uncertainty that occurs given equally likely events.
 - The more events, the more uncertainty.
 3. If a choice is broken down into two successive choices, the original H is the weighted sum of the individual values of H .
- Thus, $H = -\sum_{i=1}^n p_i \log_2(p_i)$
 - We call this entropy.

WHAT DOES IT ALL MEAN?!

- Assume the following three messages:
 1. *AAAA*
 2. *AABC*
 3. *ABCD*
- Which message, if we pick a random symbol and guess what it is, has:
 - The greatest probability of correctly guessing the symbol?
 - The least probability of correctly guessing the symbol?
- In other words, which message has:
 - The least entropy?
 - The most entropy?

BETTER UNDERSTANDING ENTROPY

- Consider message (I), which is *AAAA*.
 - $p(A) = 1$
- Using the formula for entropy we see that

$$\begin{aligned} H &= - \sum_{i=1}^n p_i \log_2(p_i) \\ &= - 1 \log_2(1) \\ &= 0 \end{aligned}$$

- Which means the entropy for this message is 0.

BETTER UNDERSTANDING ENTROPY

- Consider message (2), which is *AABC*.

- $p(A) = \frac{1}{2}, p(B) = \frac{1}{4}, p(C) = \frac{1}{4}$

- Using the formula for entropy we see that

$$\begin{aligned} H &= - \sum_{i=1}^n p_i \log_2(p_i) \\ &= -\frac{1}{2} \log_2\left(\frac{1}{2}\right) - \frac{1}{4} \log_2\left(\frac{1}{4}\right) - \frac{1}{4} \log_2\left(\frac{1}{4}\right) \\ &= \frac{3}{2} \end{aligned}$$

- Which means the entropy for this message is $\frac{3}{2}$.

BETTER UNDERSTANDING ENTROPY

- Consider message (3), which is $ABCD$.
 - $p(A) = \frac{1}{4}, p(B) = \frac{1}{4}, p(C) = \frac{1}{4}, p(D) = \frac{1}{4}$
- Using the formula for entropy we see that

$$\begin{aligned} H &= - \sum_{i=1}^n p_i \log_2(p_i) \\ &= -\frac{1}{4} \log_2 \left(\frac{1}{4} \right) - \frac{1}{4} \log_2 \left(\frac{1}{4} \right) - \frac{1}{4} \log_2 \left(\frac{1}{4} \right) - \frac{1}{4} \log_2 \left(\frac{1}{4} \right) \\ &= 2 \end{aligned}$$

- Which means the entropy for this message is 2.

GUESSING A SYMBOL

- We can think of entropy as the average number of questions needed to be asked to correctly guess what random symbol we picked from the message.
- Consider message (3), which is $ABCD$.
- The best way to guess a random symbol randomly selected out of this message is to simulate binary search.
 - Is the symbol A or B ?
 - If yes, then is it A ?
 - Else, it must be B .
 - Else, the symbol must be either C or D .
 - Is it C ?
 - Else, it must be D .
- Thus the average number of questions asked is 2, which is exactly the entropy we calculated for this message!

