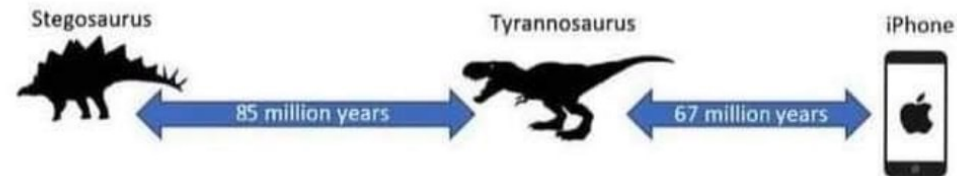




Fishbourne Roman Palace

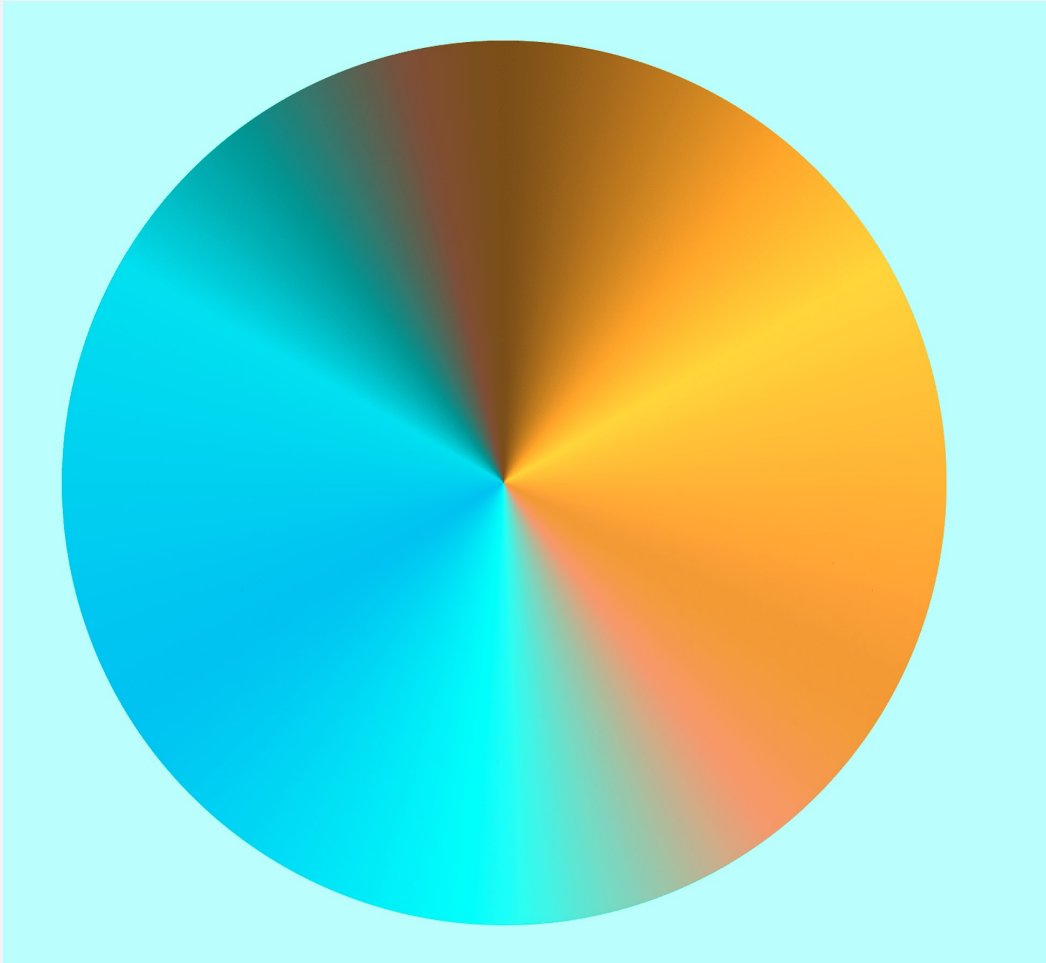
@romanpalace

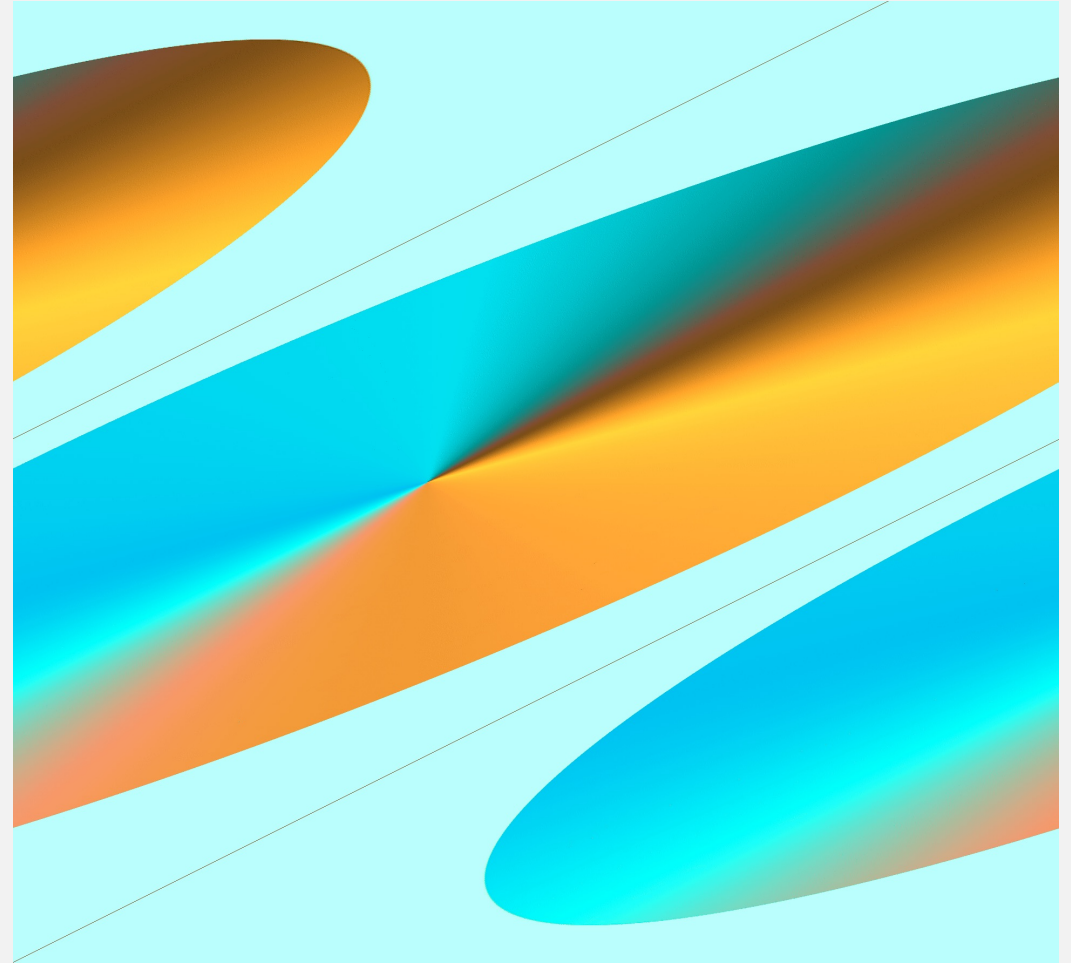
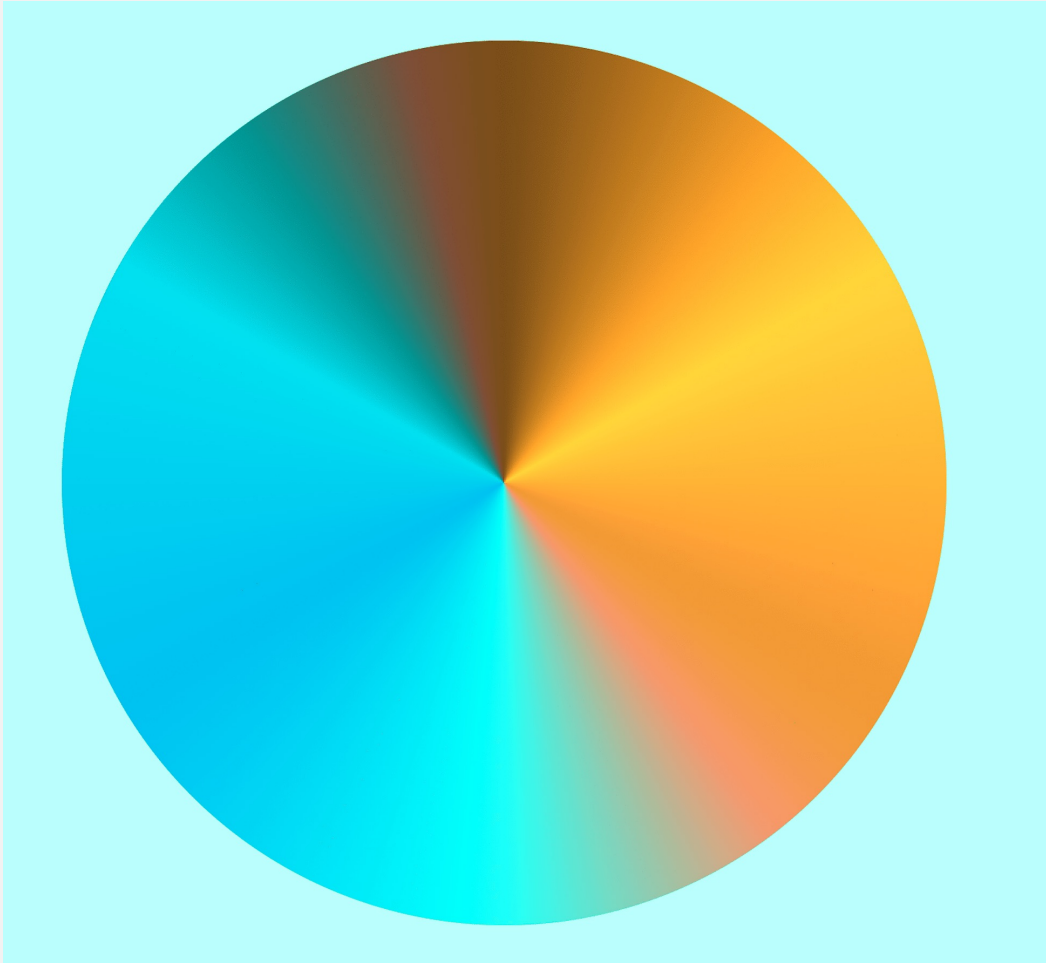
Today we will mainly be dealing with the realization that a Tyrannosaurus is chronologically closer to an iPhone than it is to a Stegosaurus.



8:17 · 07 Apr 21 · [Twitter Web App](#)

**1,573** Retweets **200** Quote Tweets **4,768** Likes





# DATA COMPRESSION

Prof. Darrell Long  
CSE 13S

3/9/20

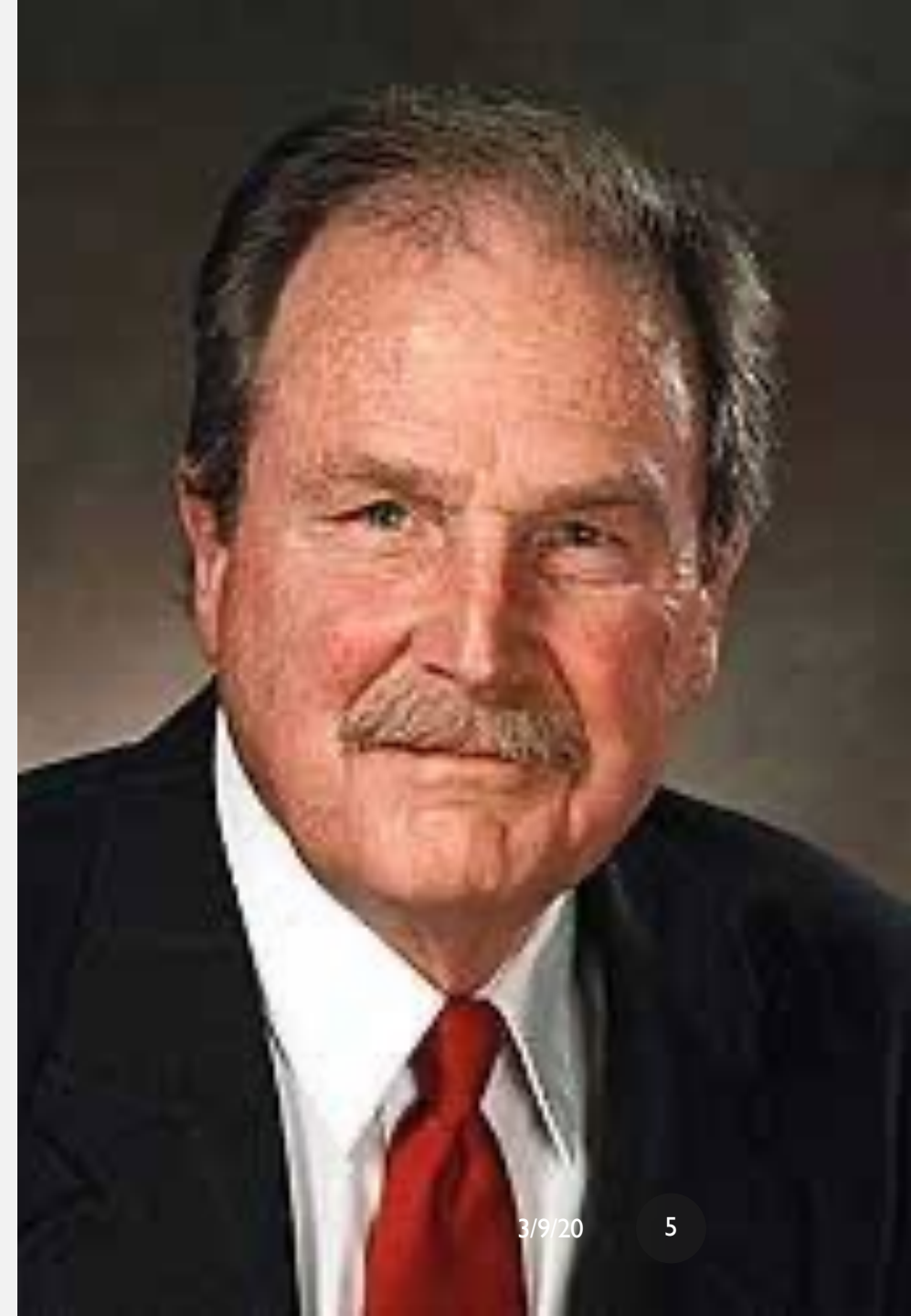
© 2023 Darrell Long

4



# 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?



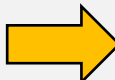
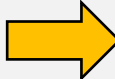
# CONSTRUCTING A HISTOGRAM

- 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

# CONSTRUCTING A HISTOGRAM

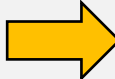
- We'll compress the message "bananas".
- First, we create a histogram of the message's unique symbol and their frequencies.
- We assume that each symbol is an ASCII character.
- We can represent this histogram as an array with 256 indices.



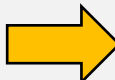
Symbol	Frequency
0	0
...	
a = 97	3
b = 98	1
...	0
n = 110	2
...	
s = 115	1
255	0

# 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.



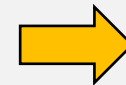
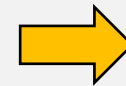
Symbol	Frequency
0	0
...	
a = 97	3
b = 98	1
...	0
n = 110	2
...	
s = 115	1
255	0





# CONSTRUCTING A HISTOGRAM

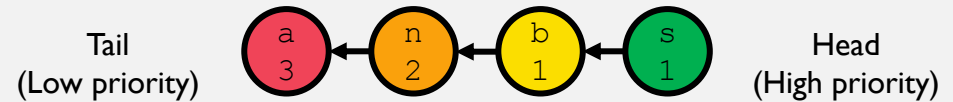
- We'll compress the message "bananas".
- First, we create a histogram of the message's unique symbol and their frequencies.
- We assume that each symbol is an ASCII character.
- We can represent this histogram as an array with 256 indices.



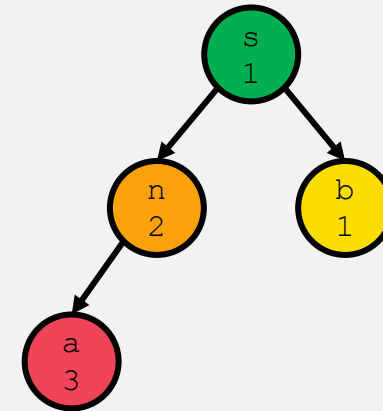
Symbol	Frequency
0	0
...	
a = 97	3
b = 98	1
...	0
n = 110	2
...	
s = 115	1
255	0

# 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.

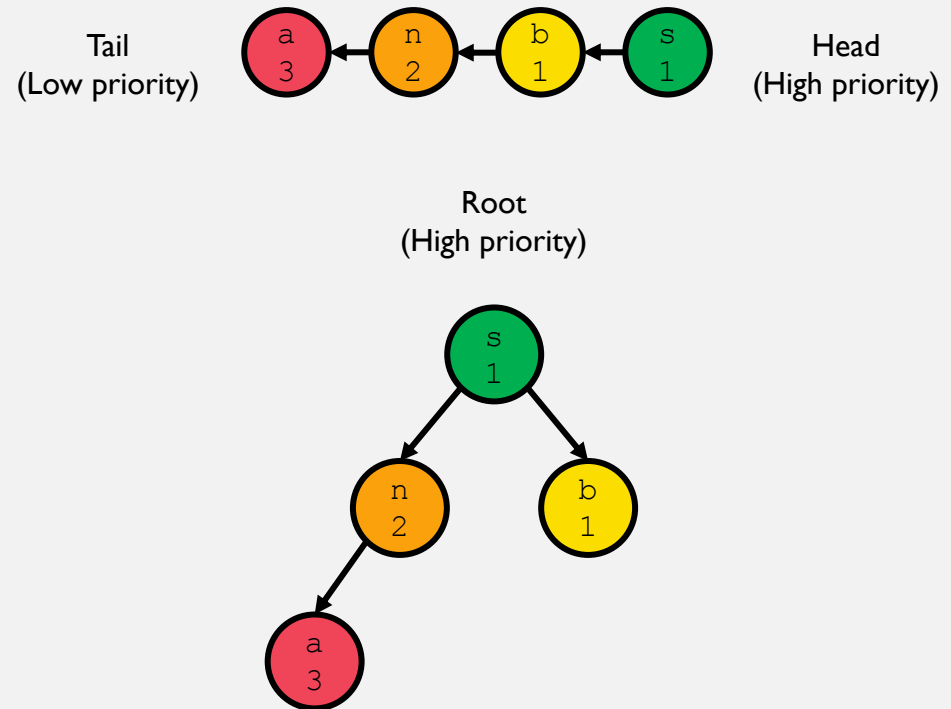


Root  
(High priority)



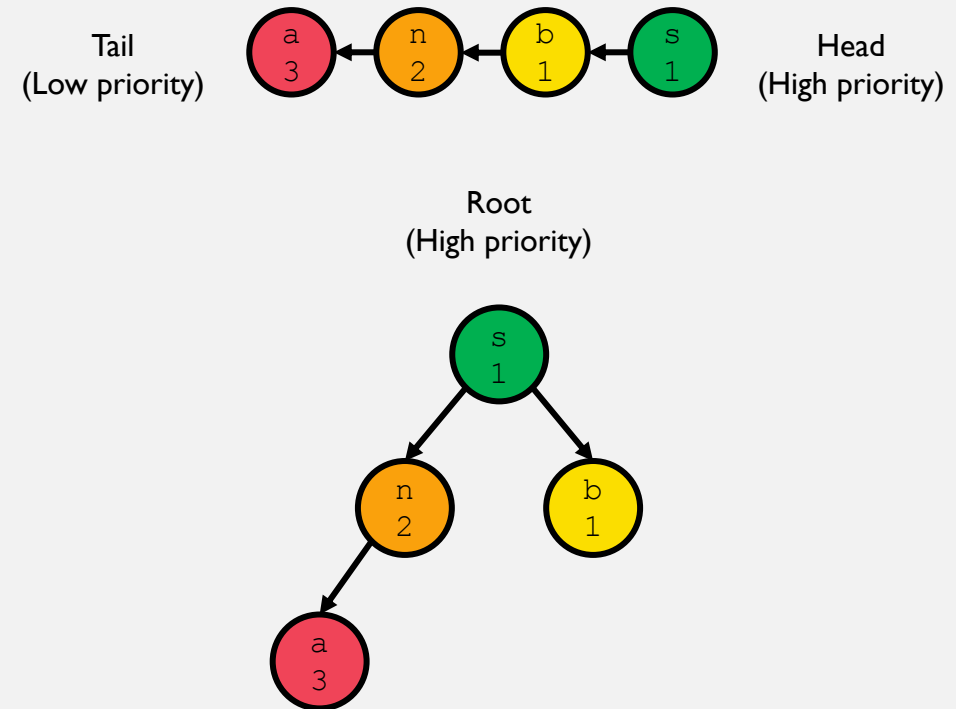
# PRIORITY QUEUING

- 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.



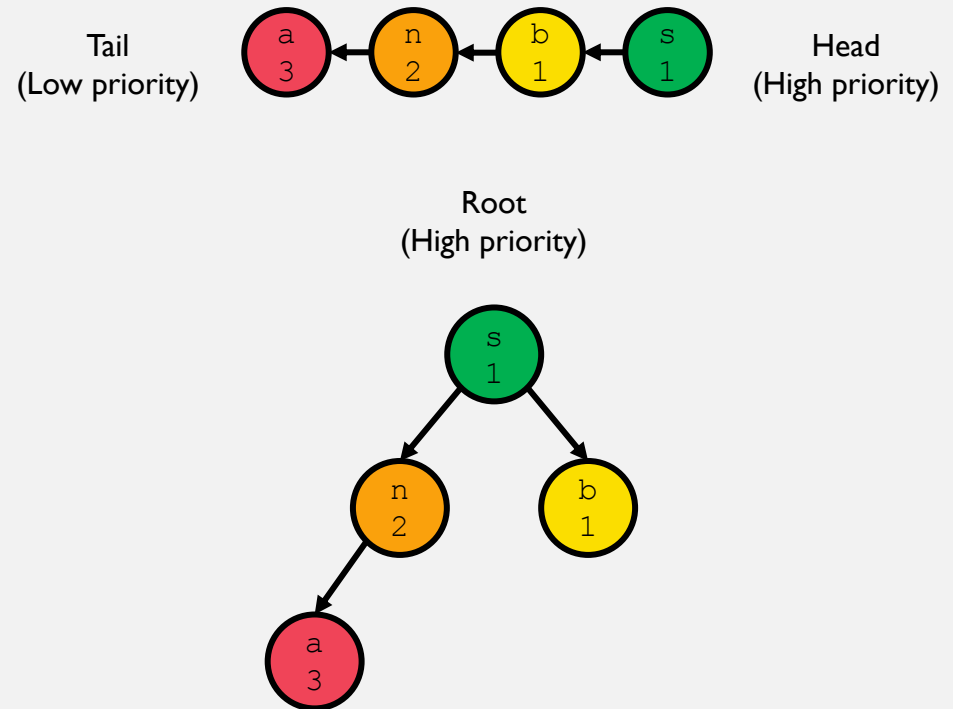
# PRIORITY QUEUING

- 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.



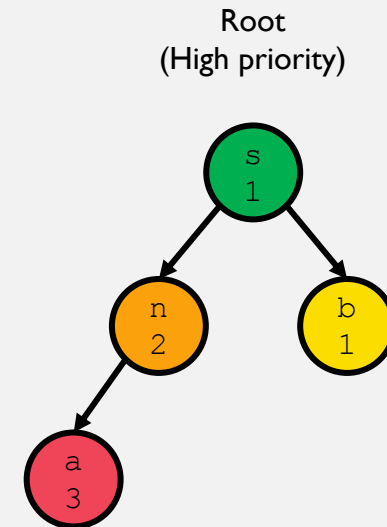
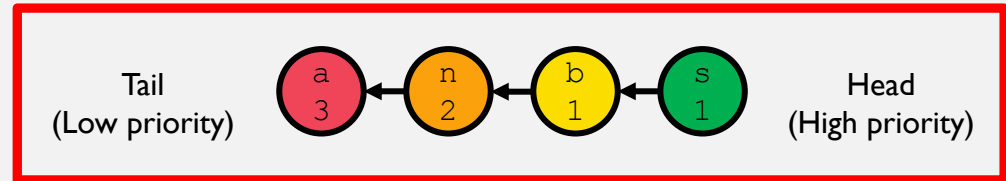
# PRIORITY QUEUING

- 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.



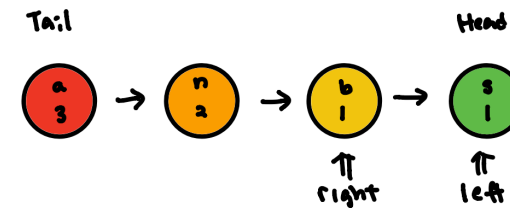
# PRIORITY QUEUING

- 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.



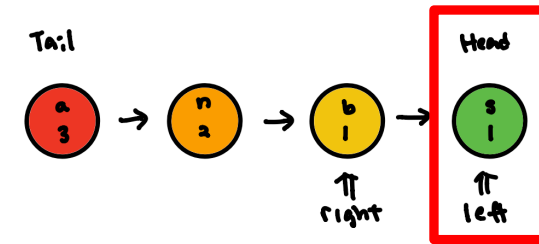
# BUILDING A HUFFMAN TREE

- 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

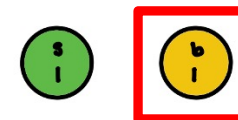
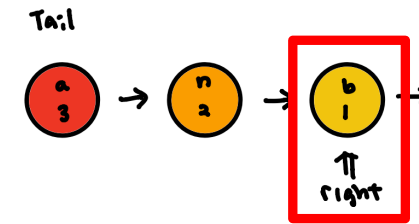
- We will create a Huffman tree using the nodes in the priority queue.
- While the queue has more than one node,
  - ➔ 1. Dequeue a node. This will be the left child node.
  - 2. Dequeue another node. This will be the right child node.
  - 3. Create a parent node for the left and right child nodes. The 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.





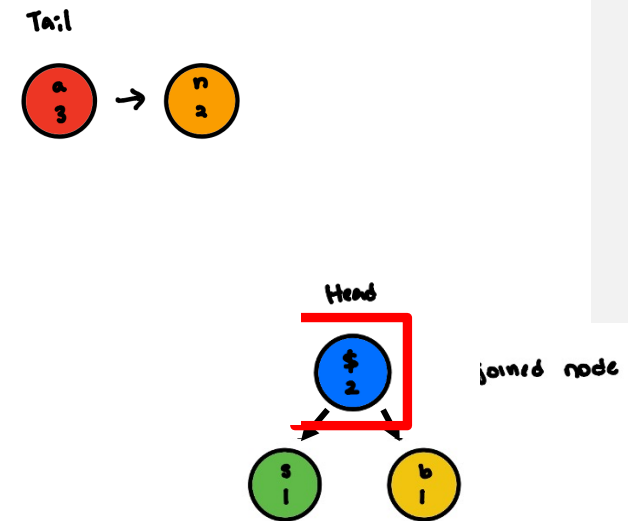
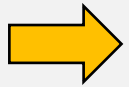
# BUILDING A HUFFMAN TREE

- 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.



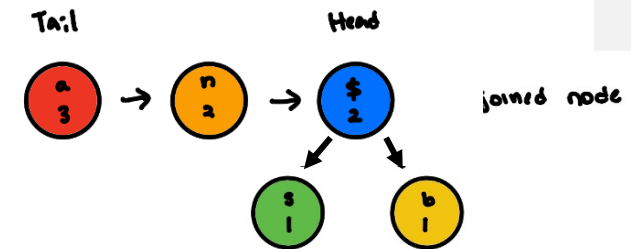
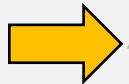
# BUILDING A HUFFMAN TREE

- 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.



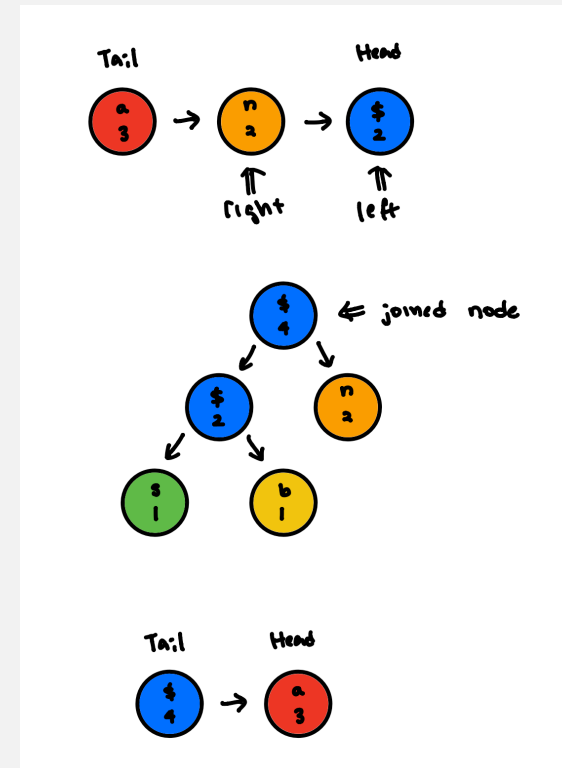
# BUILDING A HUFFMAN TREE

- 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.



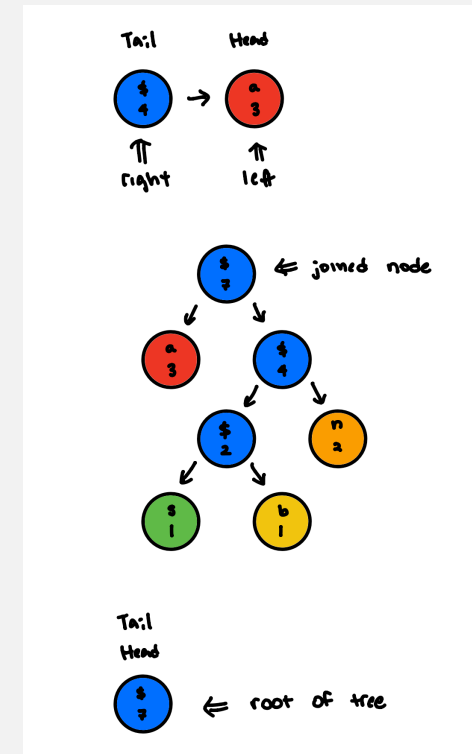
# BUILDING A HUFFMAN TREE

- 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.



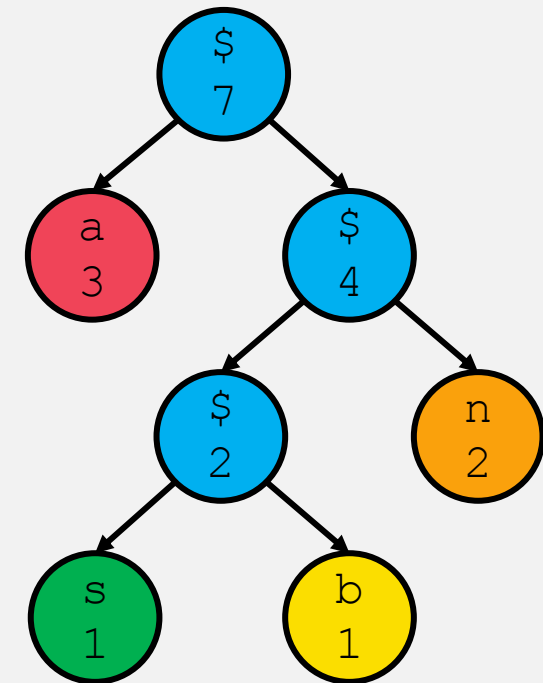
# BUILDING A HUFFMAN TREE

- 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.



# 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.

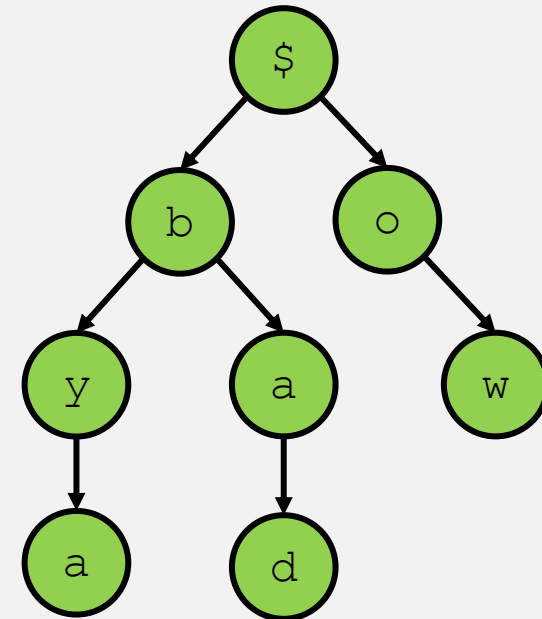
- Code – an unsigned 16-bit integer that denotes a word.

- 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: ■.



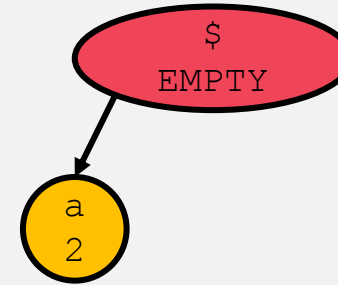
# LZ78 ENCODING

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



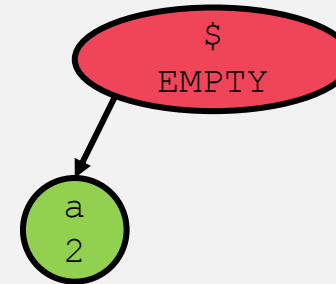
# LZ78 ENCODING

- 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.



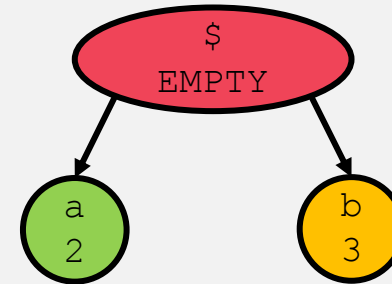
# LZ78 ENCODING

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



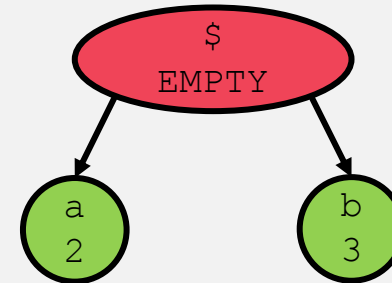
# LZ78 ENCODING

- 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.



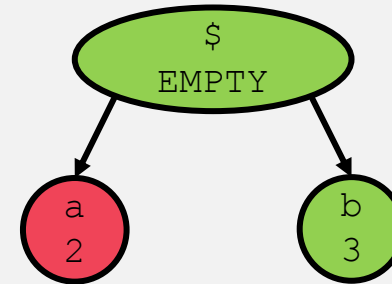
# LZ78 ENCODING

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



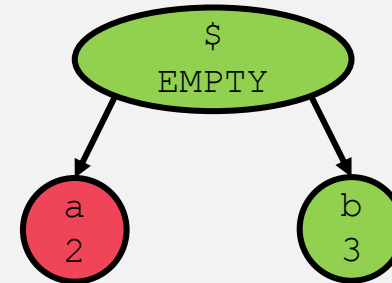
# LZ78 ENCODING

- 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

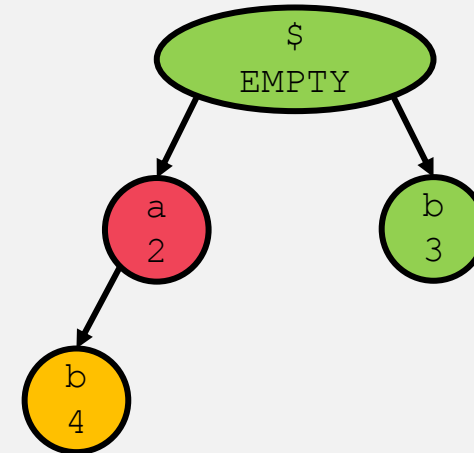
- Message: "abababa".
- The current symbol to consider is 'b'.
- Check if the current node has the child 'b'.





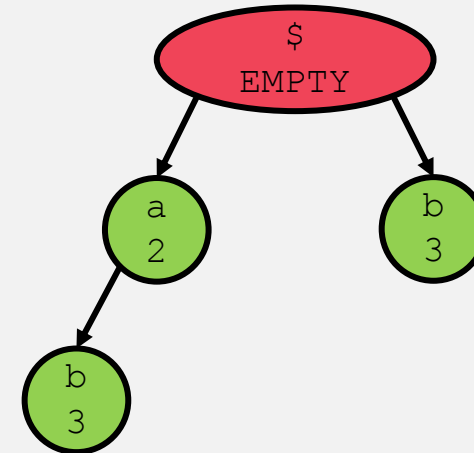
# LZ78 ENCODING

- 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.



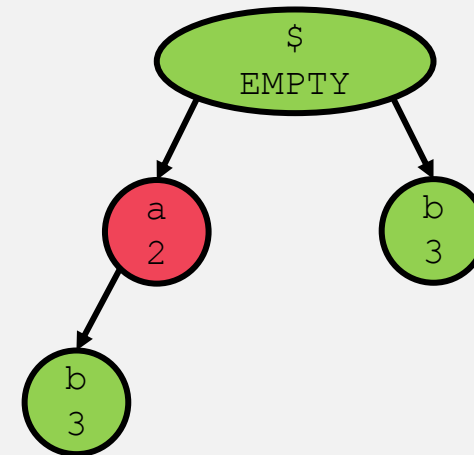
# LZ78 ENCODING

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



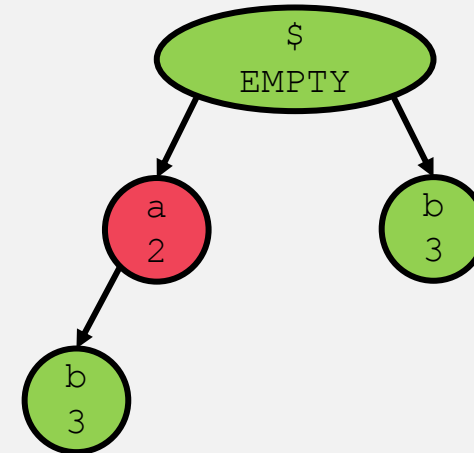
# LZ78 ENCODING

- 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.



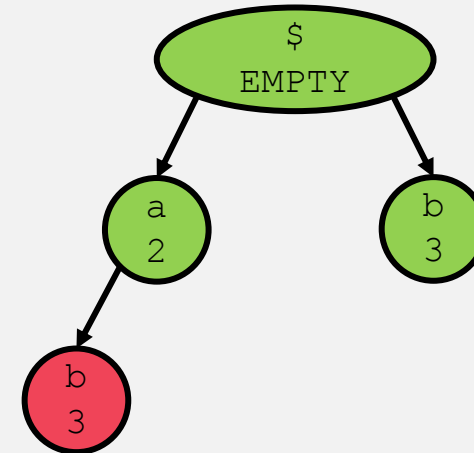
# LZ78 ENCODING

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



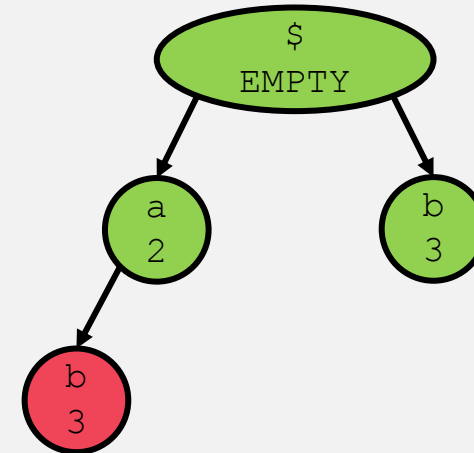
# LZ78 ENCODING

- 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.



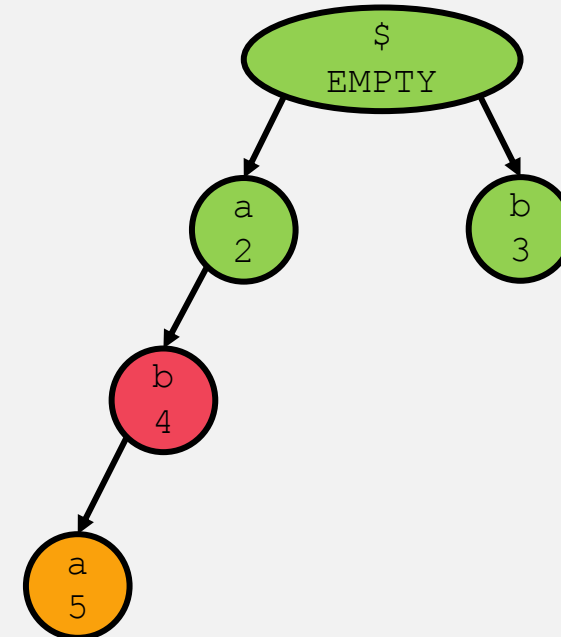
# LZ78 ENCODING

- 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.

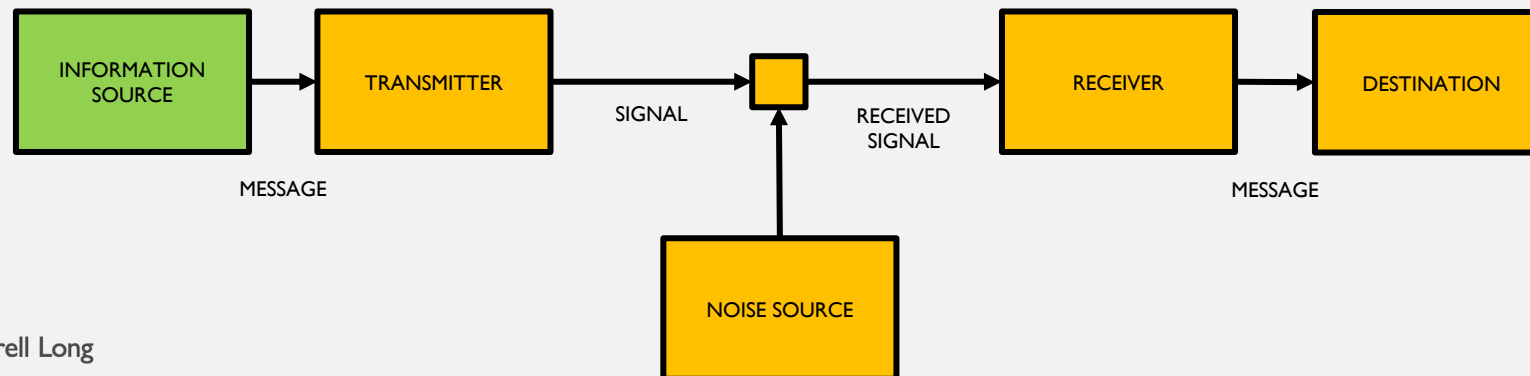


# COMMUNICATION

- 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

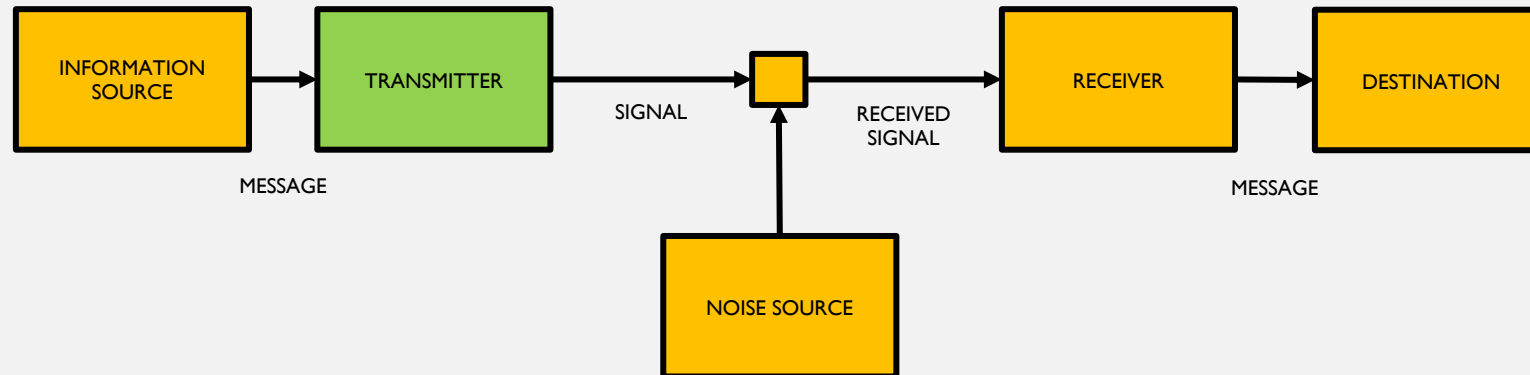
# INFORMATION SOURCE

- 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

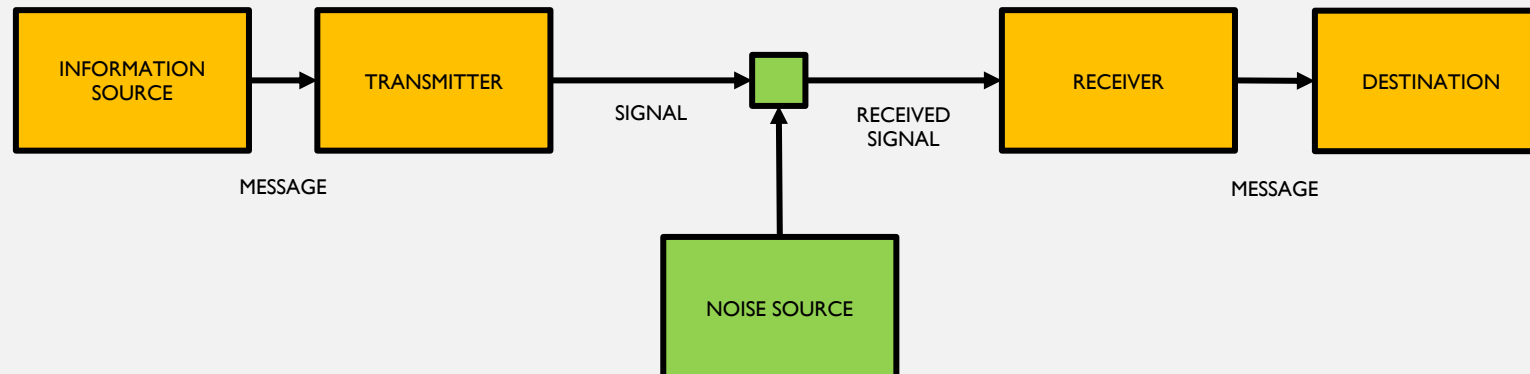
- Operates on the message to produce a signal suitable for transmission over the channel.
- This is where compression and/or encryption occurs.





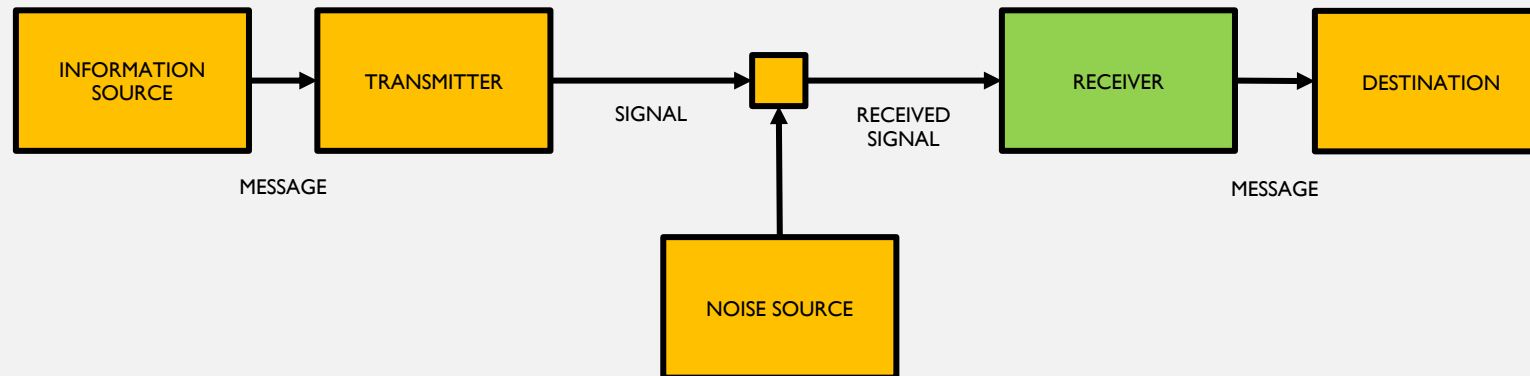
# CHANNEL

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



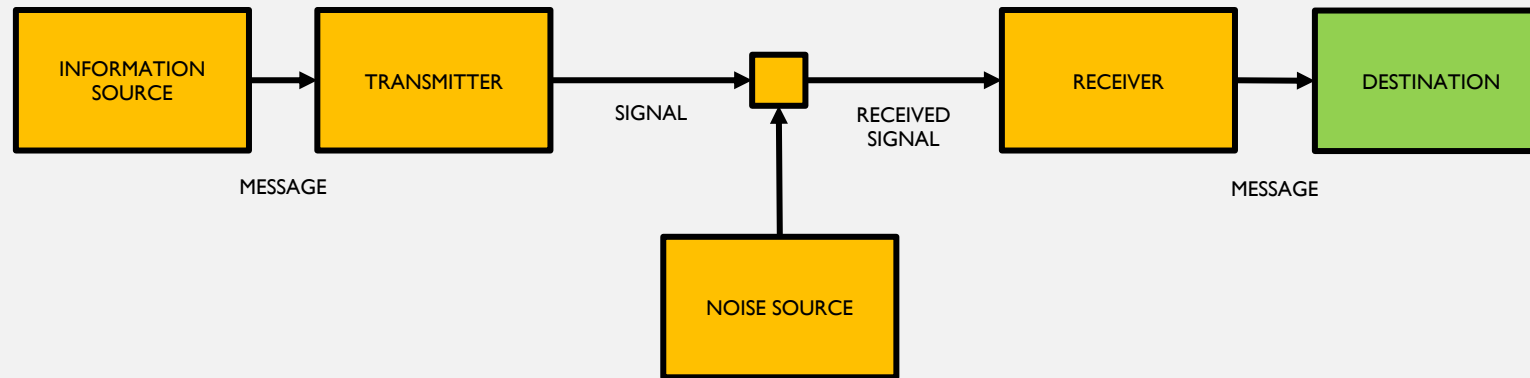
# RECEIVER

- 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!

