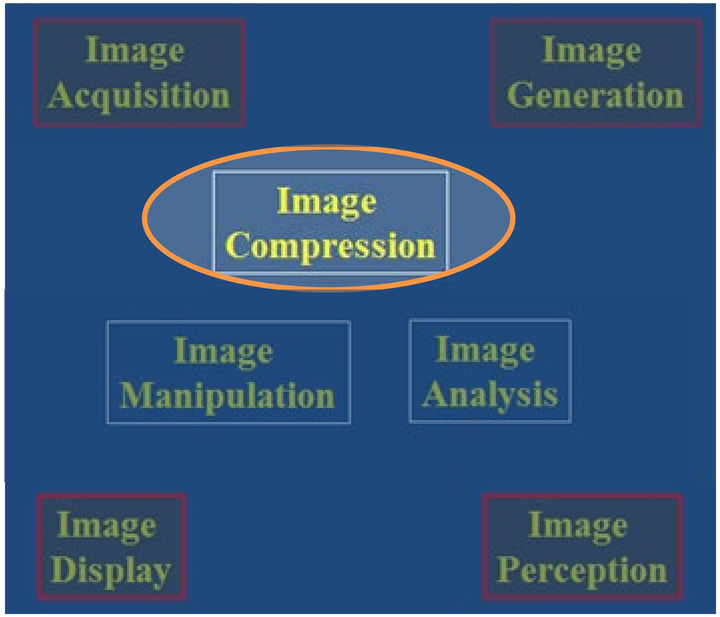
Digital Image Processing

Image Compression



Caution:
The PDF version of this presentation will appear to have errors due to heavy use of animations

Brent M. Dingle, Ph.D. 2015
Game Design and Development Program
Mathematics, Statistics and Computer Science
University of Wisconsin - Stout

Material in this presentation is largely based on/derived from presentation(s) and book: The Digital Image by Dr. Donald House at Texas A&M University

Lecture Objectives

- Previously
  - Filtering
  - Interpolation
  - Warping
  - Morphing

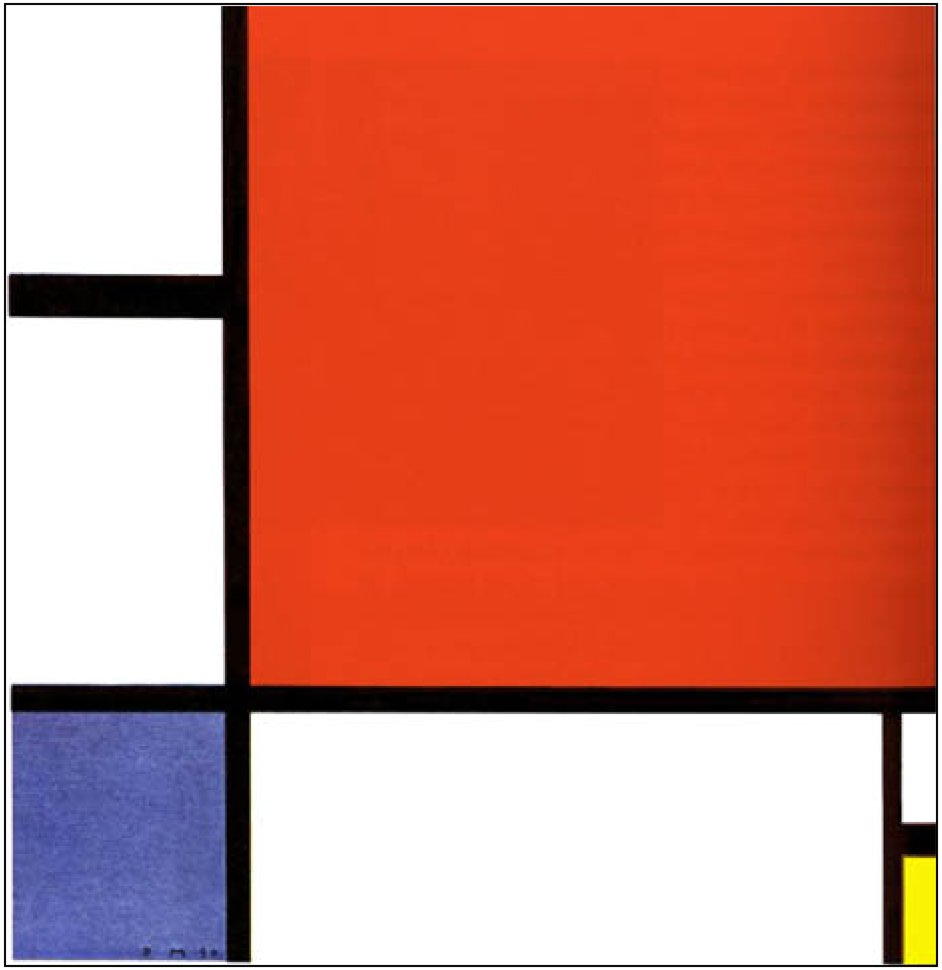
Image Manipulation and Enhancement

- Today
  - Image Compression

Definition: File Compression

- ***Compression***: the process of encoding information in fewer bits
  - Wasting space is bad, so compression is good
  - Image Compression
    - Redundant information in images
      - Identical colors
      - Smooth variation in light intensity
      - Repeating texture

Identical Colors

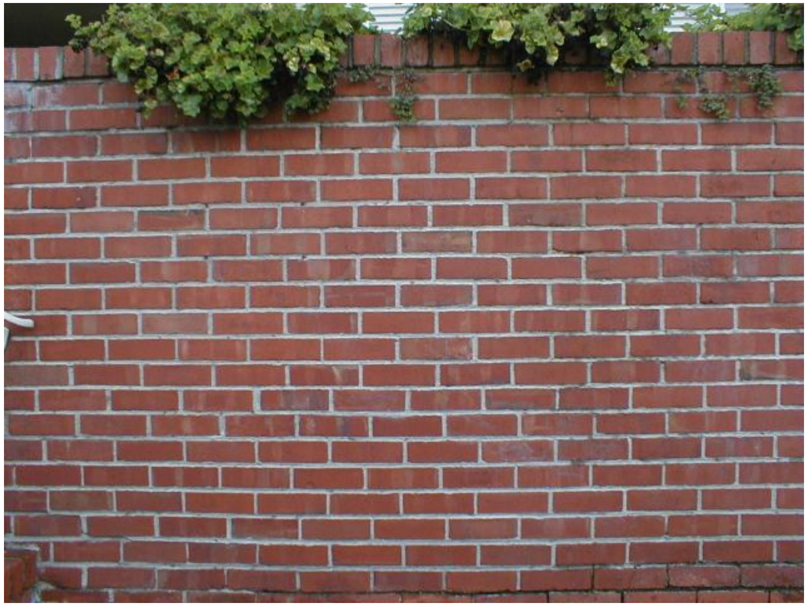


Mondrian's Composition 1930

Smooth Variation in Light Intensity

Digital rendering using Autodesk VIZ.
(Image Credit: Alejandro Vazquez.)

Repeating Texture



Alvar Aalto Summer House 1953

What is Compression Really?

- Works because of data redundancy
  - Temporal
    - In 1D data, 1D signals, Audio…
  - Spatial
    - correlation between neighboring pixels or data items
  - Spectral
    - correlation between color or luminescence components
    - uses the frequency domain to exploit relationships between frequency of change in data
  - Psycho-visual
    - exploits perceptual properties of the human (visual) system

Two General Types

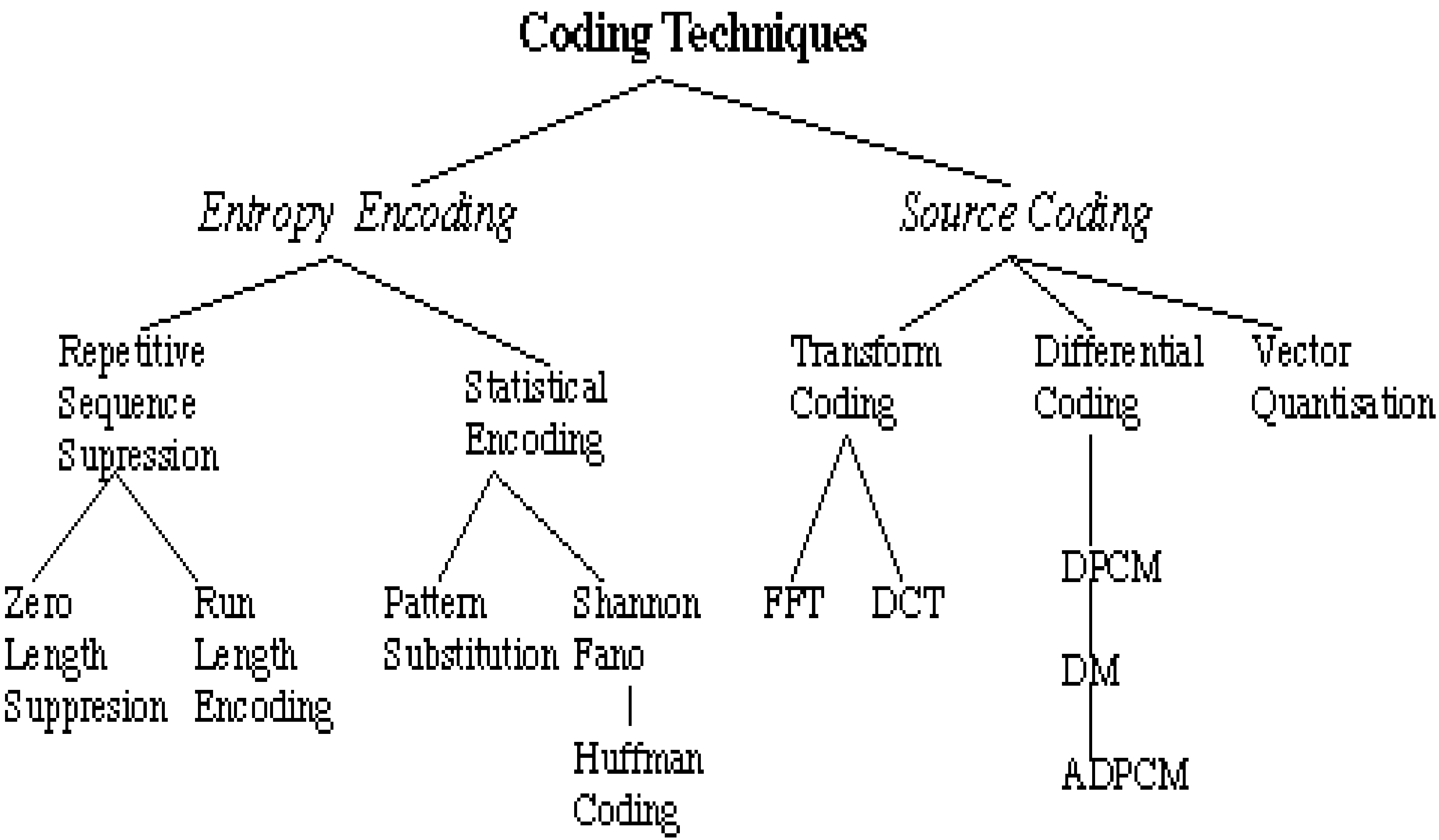
- Lossless Compression
  - data is compressed and can be uncompressed without loss of detail or information
    - bit-preserving
    - reversible
- Lossy Compression
  - purpose is to obtain the best possible fidelity for a given bit-rate
    - or minimizing the bit-rate to achieve a given fidelity measure
  - Video and audio commonly use lossy compression
    - because humans have limited perception of finer details

Two Types

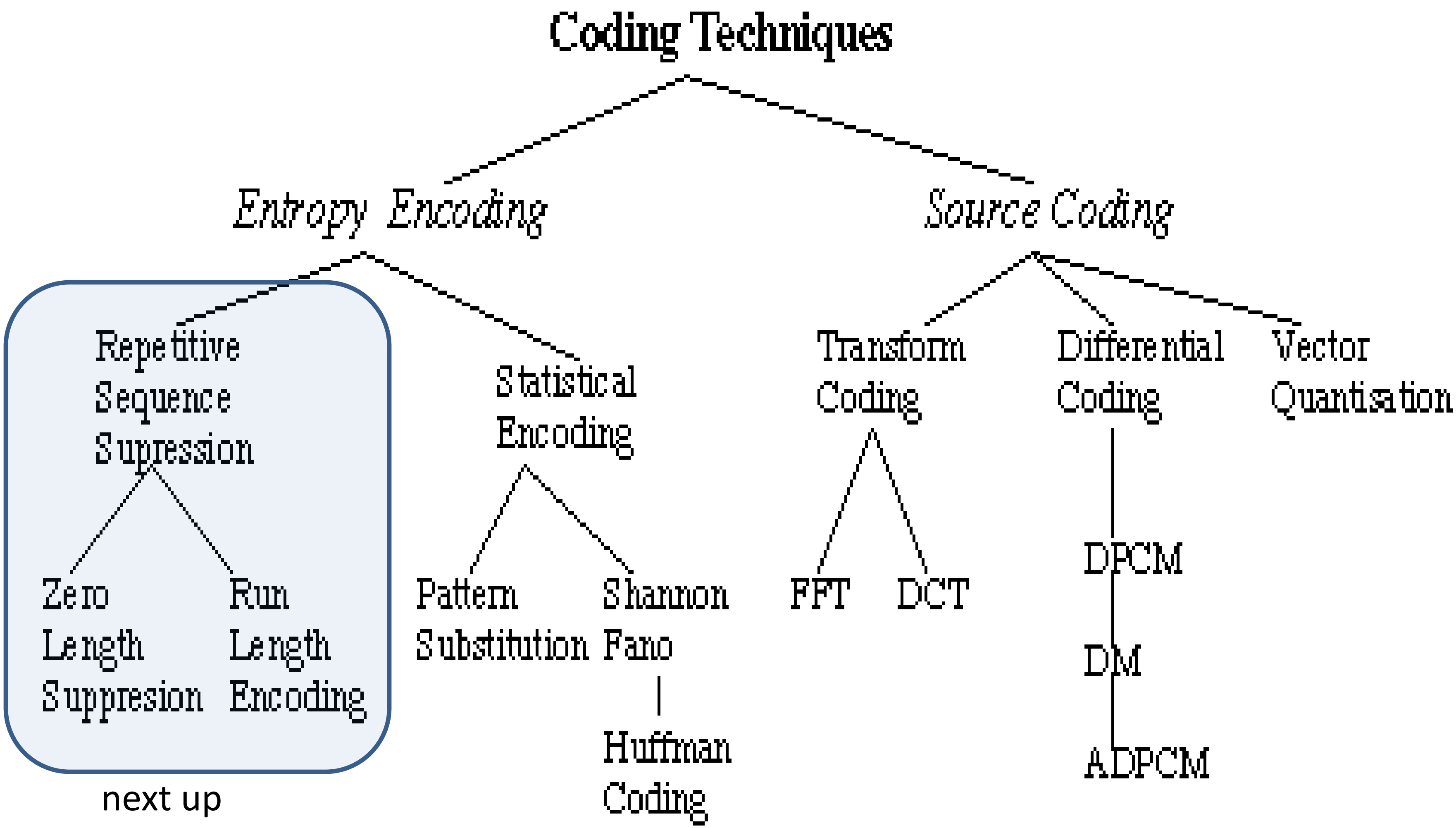
- Lossless compression often involves some form of entropy encoding
  - based in information theoretic techniques
    - see next slide for visual

- Lossy compression uses source encoding techniques that may involve transform encoding, differential encoding or vector quanatization
    - see next slide for visual

Compression Methods



Compression Methods



Simple Lossless Compression

- Simple Repetition Suppression
  - If a sequence contains a series of N successive tokens
  - Then they can be replaced with a single token and a count of the number of times the token repeats
    - This does require a flag to denote when the repeated token appears
  - Example
    - 1234444444444
    - can be denoted
    - 123f9
    - where f is the flag for four

Run-length Encoding (RLE)

- RLE is often applied to images
  - It is a small component used in JPEG compression

- Conceptually
  - sequences of image elements $X_1, X_2, \ldots, X_n$ are mapped to pairs $(c_1, L_1), (c_2, L_2), \ldots, (c_n, L_n)$
    - where c_i represent the image intensity or color
    - and L_i the length of the i^{th} run of pixels

Run-Length Encoding (RLE): lossless

- Scanline: 2 2 2 2 2 2 2 3 4 1 1 1

12 values

- Run-length encoding

(7 2) (1 3) (1 4) (3 1)

8 values

Run-Length Encoding (RLE): lossless

- Scanline: 2 2 2 2 2 2 2 3 4 1 1 1

12 values

run of
repeating values

- Run-length encoding

  (7 2) (1 3) (1 4) (3 1)

8 values

repeat count

pixel value

25% reduction of memory use

RLE: worst case

- Scanline: 1 2 3 4 5 6 7 8

8 values

- Run-length encoding:

(1 1)(1 2)(1 3)(1 4)(1 5)(1 6)(1 7)

16 values

doubles space

RLE: Improving

- Scanline: 5 5 5 5 5 5 5 3 4 1 1 1

- Run-length encoding

need to flag this as meaning "not repeating"

(7 5) (2 3 4)(3 1)

(7 2) → 5 5 5 5 5 5 5

(2 3 4) → 3 4

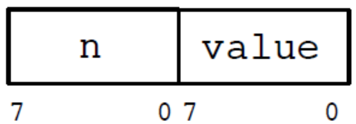
(3 1) → 1 1 1

the flag indicates that 2 explicitly given values follow

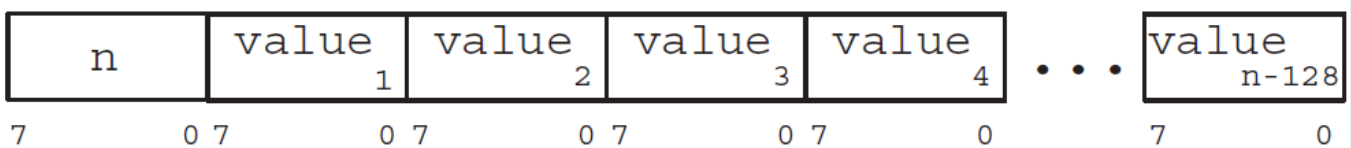
Using this improvement
The worst case then only adds 1 extra value

How to flag the repeat/no repeat?

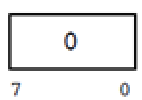
- SGI Iris RGB Run-length Encoding Scheme



$1 \leq n \leq 127$, Repeat count for run of length $= n$

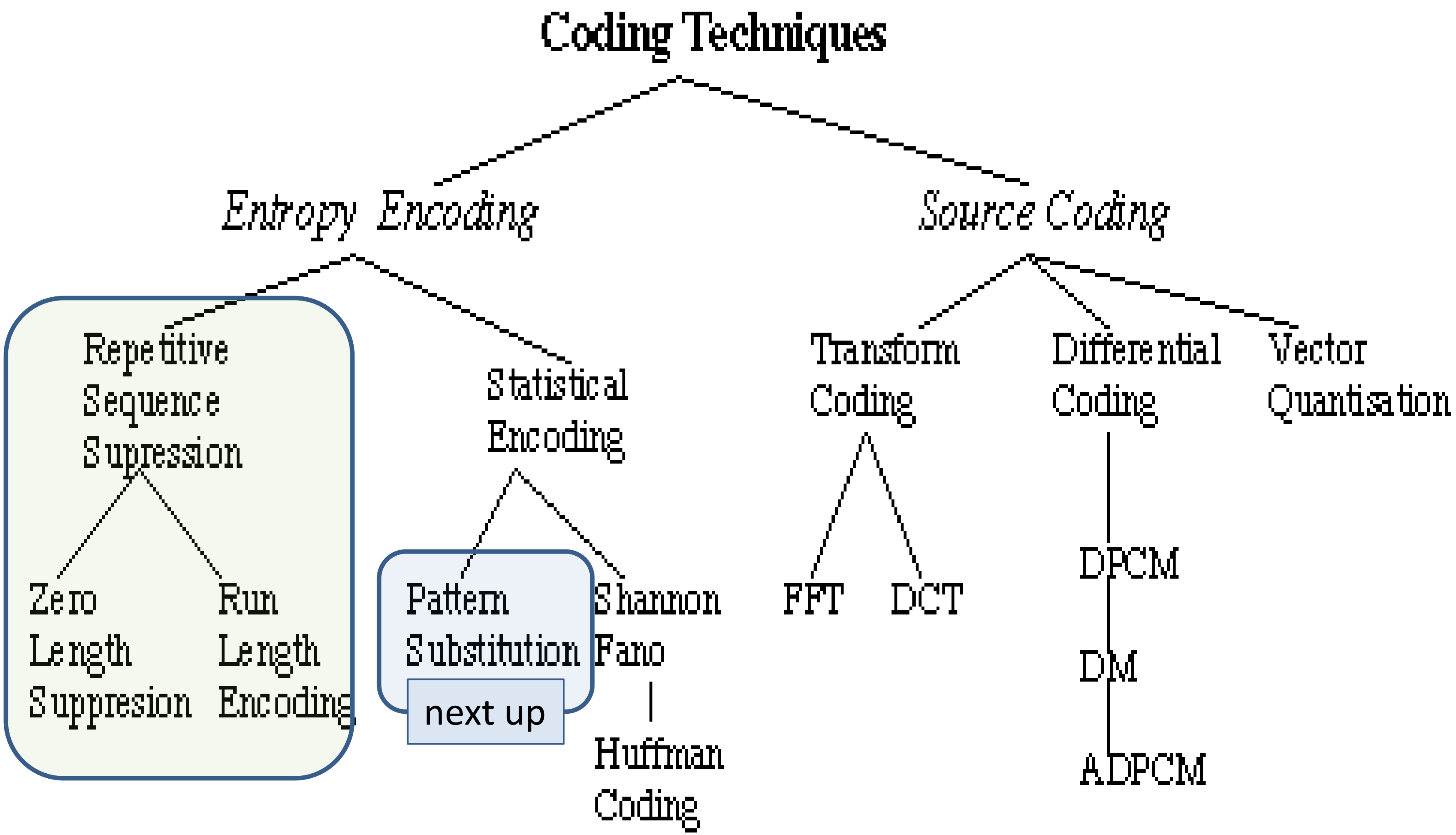


$128 \leq n \leq 255$, $n - 128$ gives number of nonrepeating values that follow



end of data

Compression Methods



Compression: Pattern Substitution

- Pattern Substitution, lossless
- Simple form of statistical encoding
- Concept
  - Substitute a frequently repeating pattern with a shorter code
    - the shorter code(s) may be predefined by the algorithm being used or dynamically created

Table Lookup

- Table Lookup can be viewed as a Pattern Substitution Method
- Example
  - Allow full range of colors (24 bit, RGB)
  - Image only uses 256 (or less) unique colors (8 bits)
  - Create a table of which colors are used
    - Use 8 bits for each color instead of 24 bits
      - Conceptually how older BMPs worked
        - » color depth <= 8 bits

Table Lookup: GIF

- Graphics Interchange File Format (GIF)
  - uses table lookups → Color LookUp Table (CLUT)

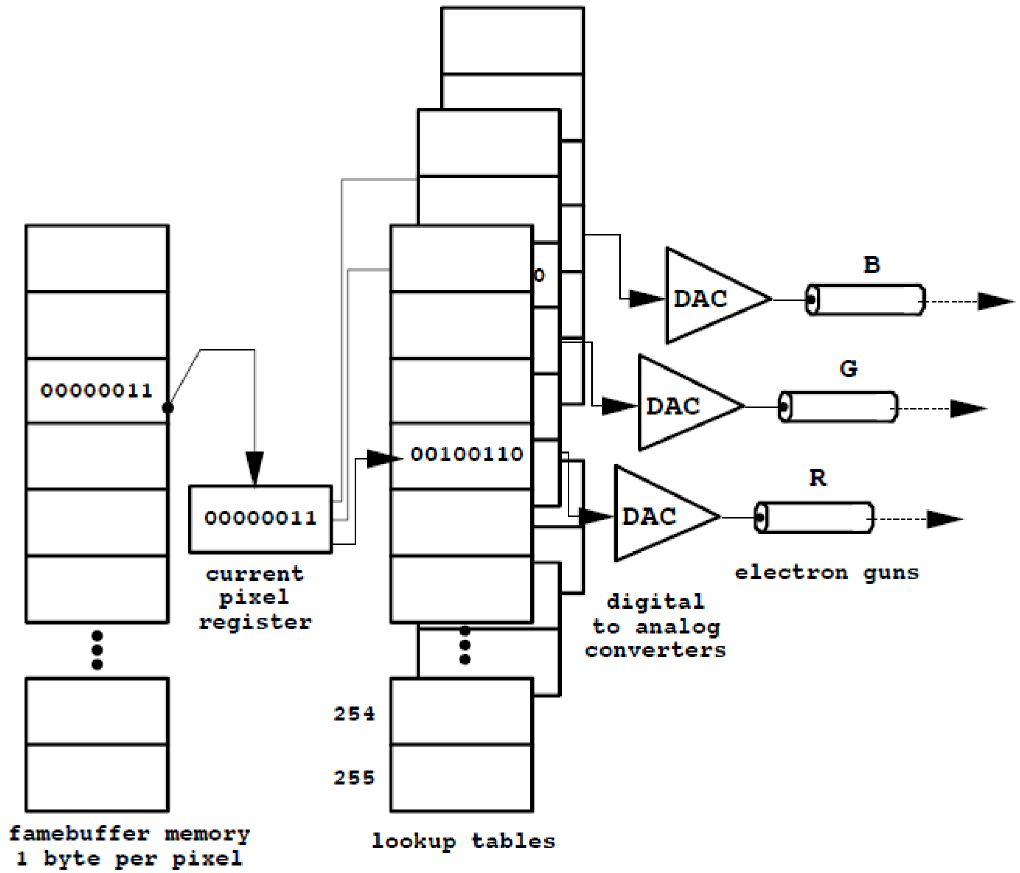


Figure 3.8: 8-Bit Color Framebuffer with 3 Lookup Tables

GIF Compression with Color LookUp Table (CLUT)

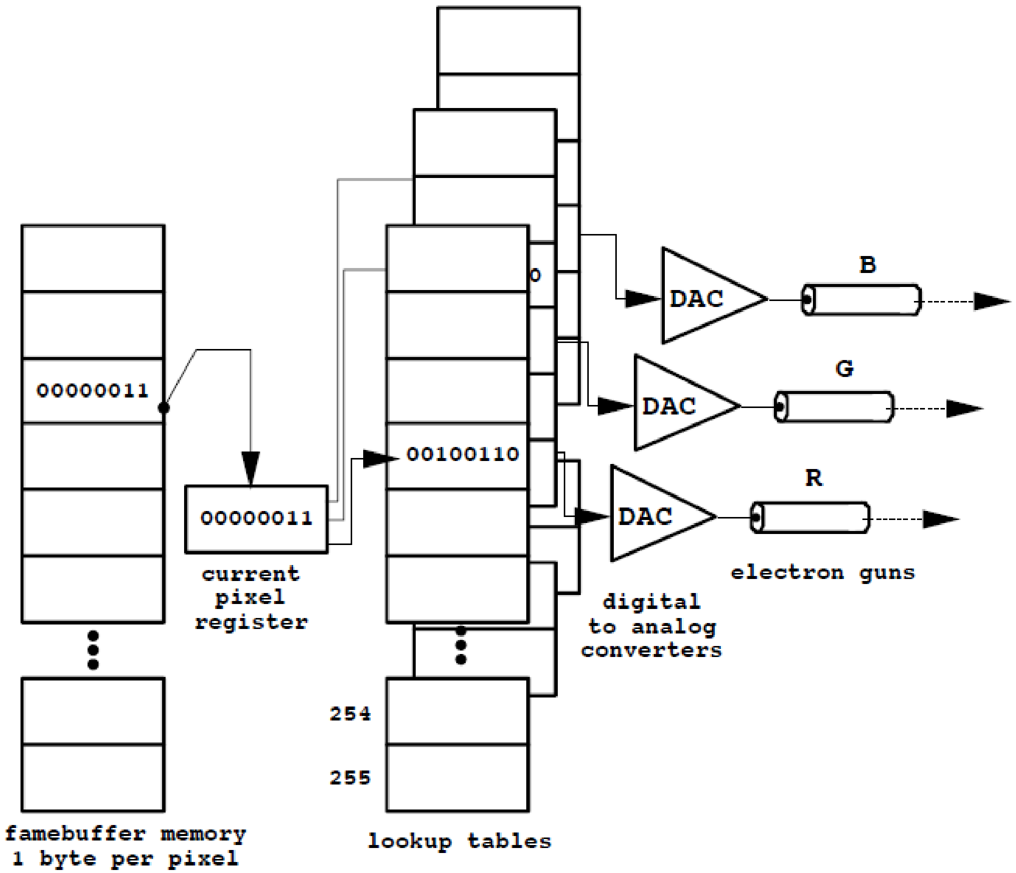


Figure 3.8: 8-Bit Color Framebuffer with 3 Lookup Tables

Example

Image Size = 1000x1000
256 colors
Each color 24 bit (RGB)

without CLUT
1000*1000*24 bits

with CLUT
1000*1000*8 bit (index data)
+ 3*256*8bit (table data)

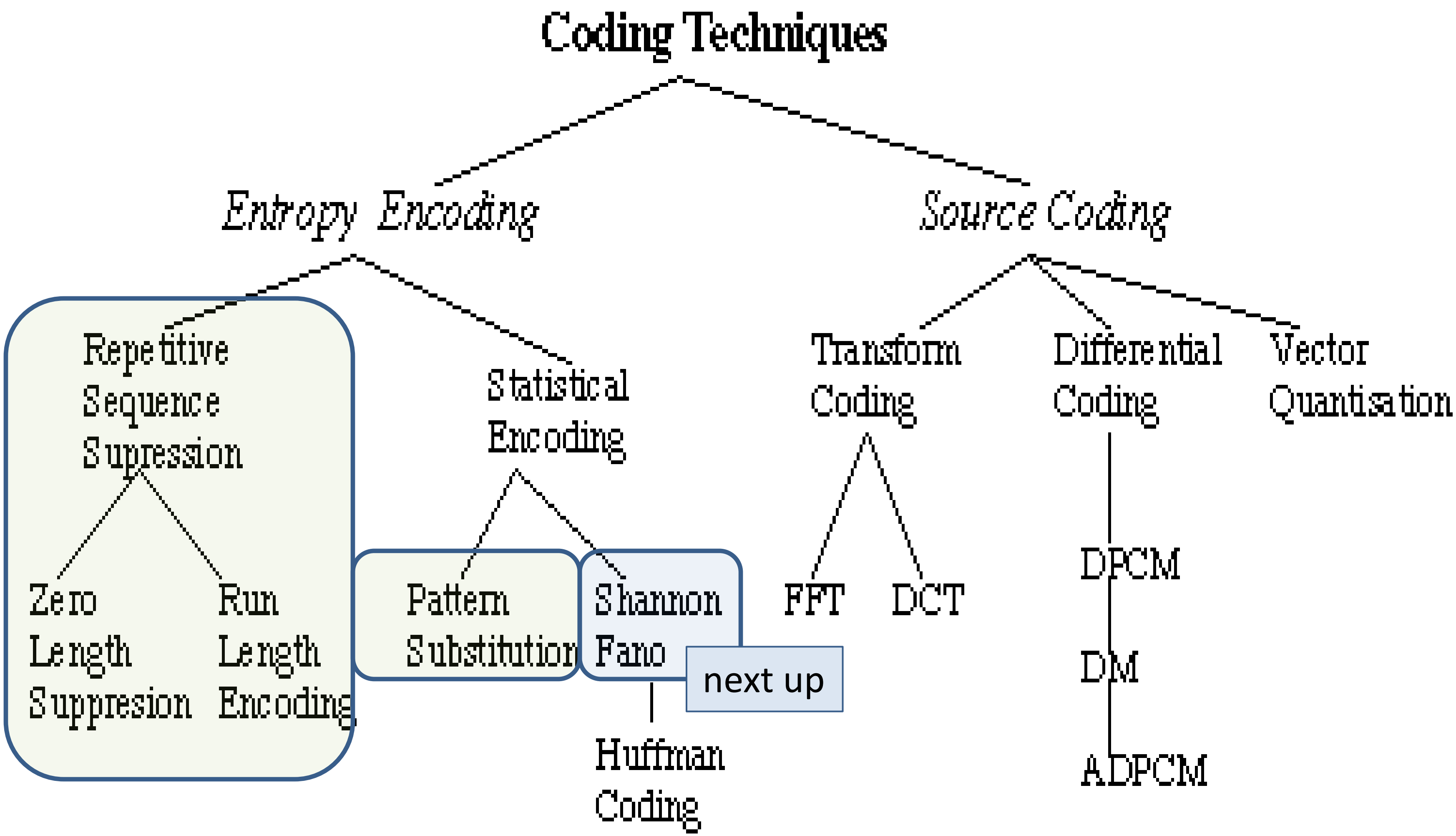
Use about 2/3 the space
(when image size is "big")

Compression: Pattern Substitution

- Table lookups work

- But Pattern Substitution typically is more dynamic
  - Counts occurrence of tokens
  - Sorts (say descending order)
  - Assign highest counts shortest codes

Compression Methods



Lossless Compression: Entropy Encoding

- Lossless compression often involves some form of entropy encoding and are based in information theoretic techniques
  - *Aside:*
    - *Claude Shannon is considered the father of information theory*

Shannon-Fano Algorithm

- Technique proposed in Shannon’s 1948 article. introducing the field of Information Theory:
  - A Mathematical Theory of Communication
    - Shannon, C.E. (July 1948). "A Mathematical Theory of Communication". Bell System Technical Journal 27: 379–423.

- Method Attributed to Robert Fano, as published in a technical report
  - The transmission of information
    - Fano, R.M. (1949). "The transmission of information". Technical Report No. 65 (Cambridge (Mass.), USA: Research Laboratory of Electronics at MIT).

Example: Shannon-Fano Algorithm

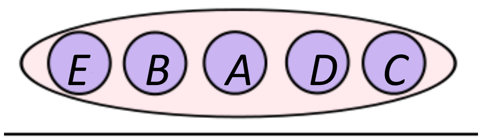
Symbol	A	B	C	D	E
Count	15	7	6	6	5

Example: Shannon-Fano Algorithm

Symbol	E	B	A	D	C
Count	15	7	6	6	5

Step 1: Sort the symbols by frequency/probability

As shown:

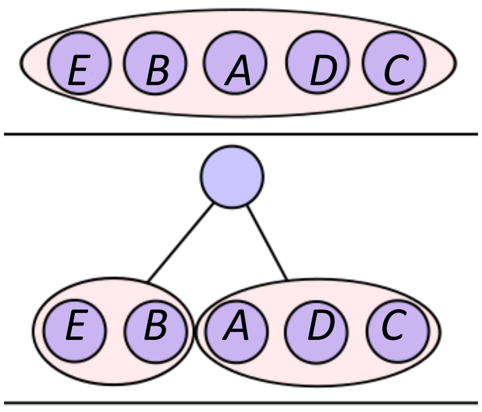


Example: Shannon-Fano Algorithm

Symbol	E	B	A	D	C
Count	15	7	6	6	5

Step 1: Sort the symbols by frequency/probability

As shown:



Step 2: Recursively divide into 2 parts
Each with about same number of counts

Dividing between B and A
results in 22 on the left and 17 on the right
-- minimizing difference totals between groups

This division means E and B codes start with 0
and A D and C codes start with 1

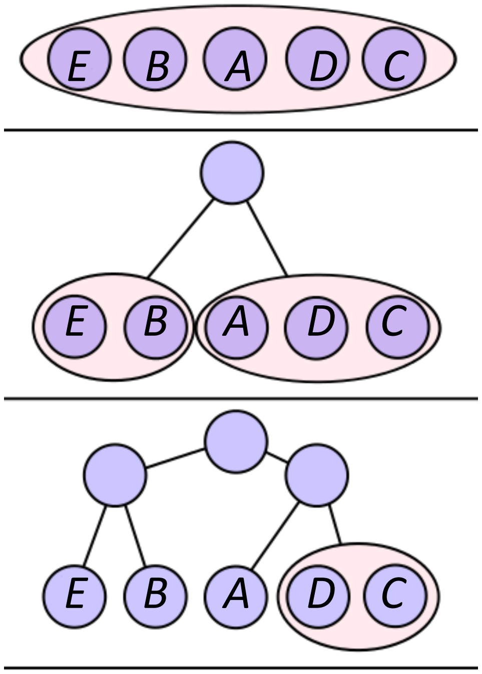
Example: Shannon-Fano Algorithm

Symbol	E	B	A	D	C
Count	15	7	6	6	5

Step 1: Sort the symbols by frequency/probability

Step 2: Recursively divide into 2 parts
Each with about same number of counts

As shown:



Dividing between B and A
results in 22 on the left and 17 on the right
-- minimizing difference totals between groups

This division means E and B codes start with 0
and A D and C codes start with 1

E and B are then divided (15:7)
A is divided from D and C (6:11)

So E is leaf with code 00,
B is a leaf with code 01
A is a leaf with code 10
D and C need divided again

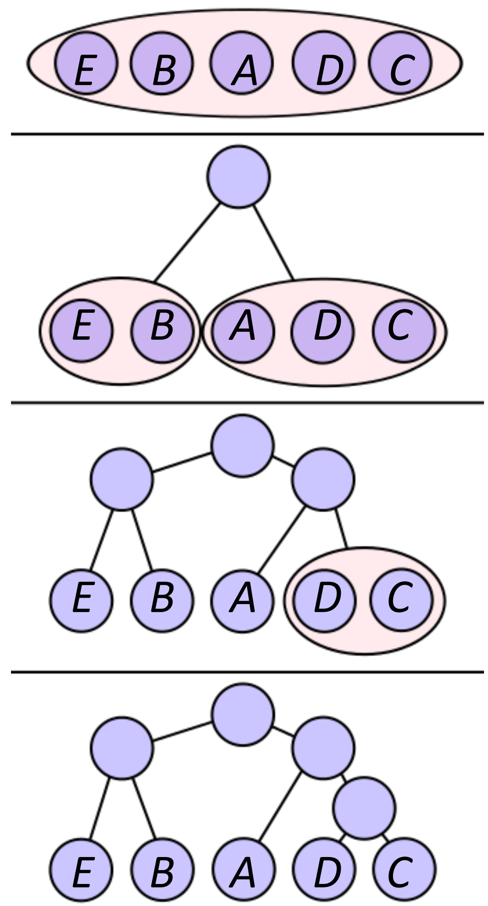
Example: Shannon-Fano Algorithm

Symbol	E	B	A	D	C
Count	15	7	6	6	5

Step 1: Sort the symbols by frequency/probability

Step 2: Recursively divide into 2 parts
Each with about same number of counts

As shown:



So E is leaf with code 00,
B is a leaf with code 01
A is a leaf with code 10
D and C need divided again

Divide D and C (6:5)

D becomes a leaf with code 110
C becomes a leaf with code 111

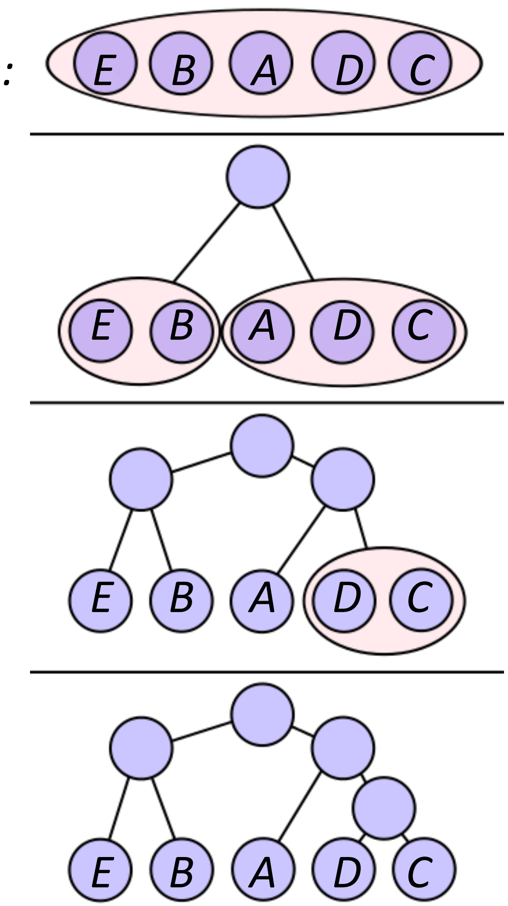
Example: Shannon-Fano Algorithm

Symbol	E	B	A	D	C
Count	15	7	6	6	5

Step 1: Sort the symbols by frequency/probability

Step 2: Recursively divide into 2 parts
Each with about same number of counts

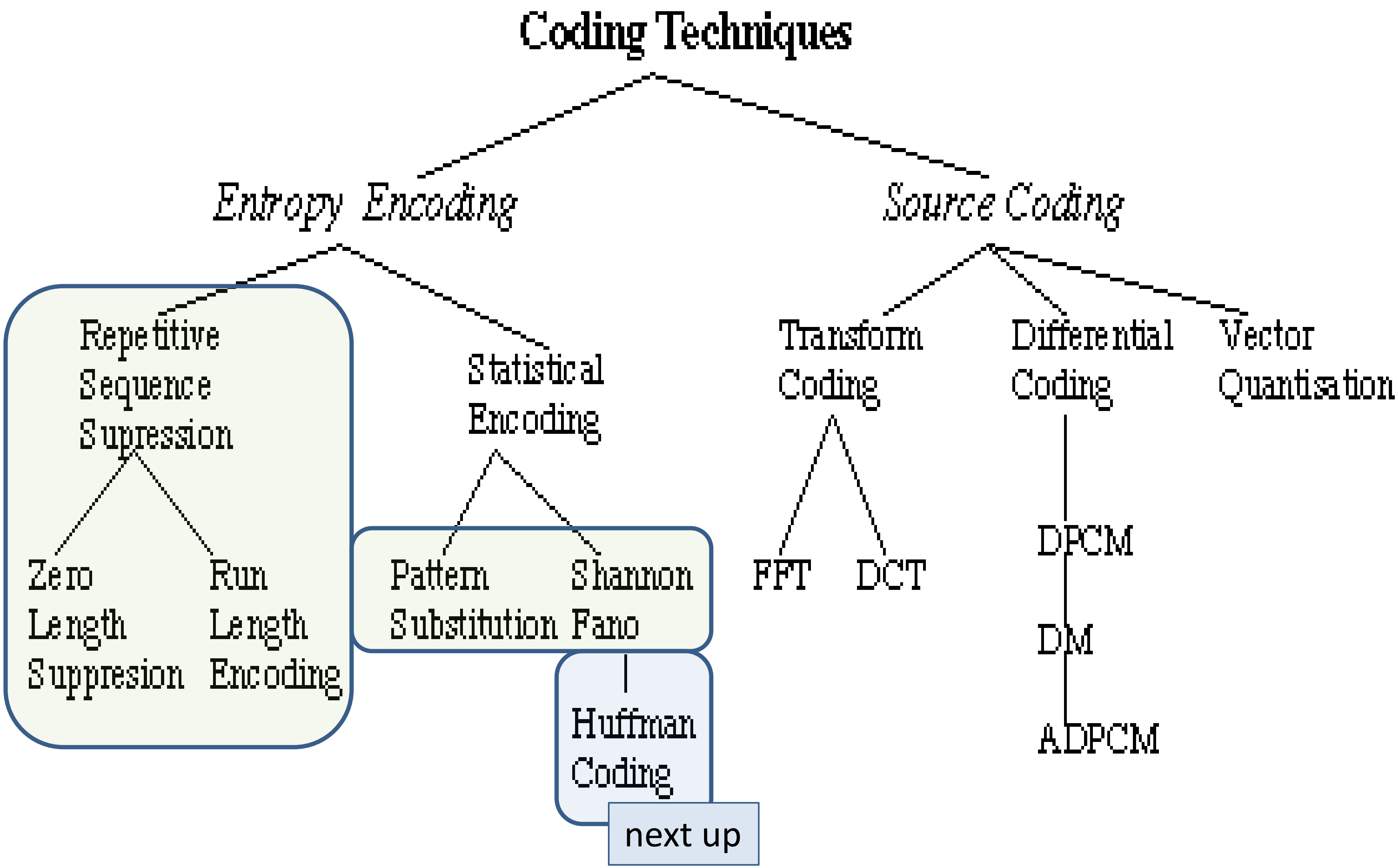
As shown:



Final Encoding:

Symbol	E	B	A	D	C
Count	00	01	10	110	111

Compression Methods



Quick Summary: Huffman Algorithm

- Encoding Summary

Step 1: Initialization
    Put all nodes in an OPEN list (keep it sorted at all times)

Step 2: While OPEN list has more than 1 node
    Step 2a: From OPEN pick 2 nodes having the lowest frequency/probability
    Create a parent node for them
    Step 2b: Assign the sum of the frequencies of the selected node
    to their newly created parent
    Step 2c: Assign code 0 to the left branch
    Assign code 1 to the right branch
    Remove the selected children from OPEN
    (note the newly created parent node remains in OPEN)

Observation

- Some characters in the English alphabet occur more frequently than others
  - The table below is based on Robert Lewand's *Cryptological Mathematics*

Letter	Relative frequency in the English language
a	8.167%
b	1.492%
c	2.782%
d	4.253%
e	12.702%
f	2.228%
g	2.015%
h	6.094%
i	6.966%
j	0.153%
k	0.772%
l	4.025%
m	2.406%

Letter	Relative frequency in the English language
n	6.749%
o	7.507%
p	1.929%
q	0.095%
r	5.987%
s	6.327%
t	9.056%
u	2.758%
v	0.978%
w	2.360%
x	0.150%
y	1.974%
z	0.074%

Huffman Encoding (English Letters)

- **Huffman encoding**: Uses variable lengths for different characters to take advantage of their relative frequencies
  - Some characters occur more often than others
    - If those characters use < 8 bits each, the file will be smaller
  - Other characters may need > 8 bits
    - but that's ok → they don't show up often

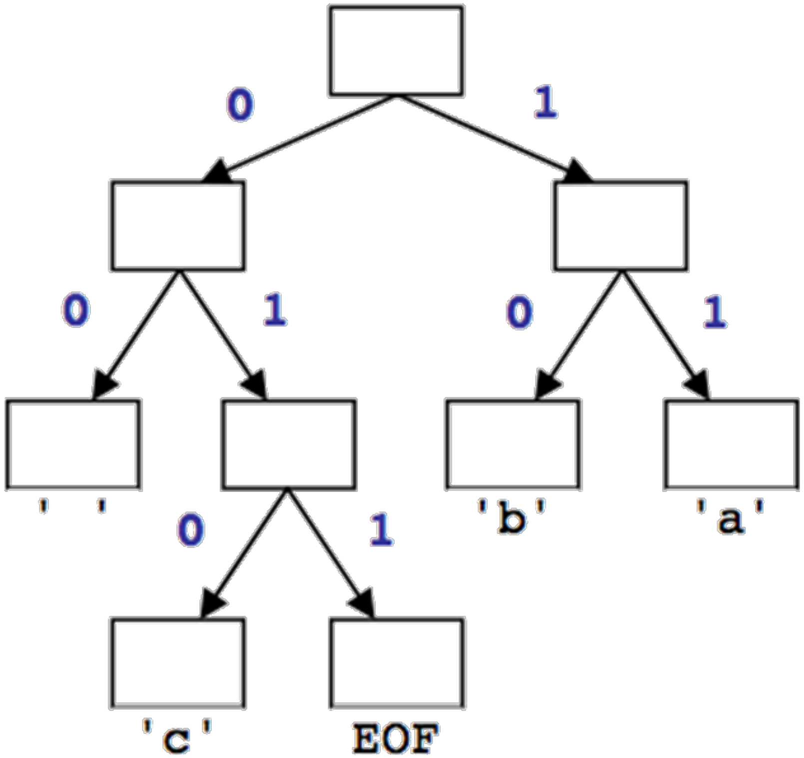
Char	ASCII value	ASCII (binary)	Hypothetical Huffman
' '	32	00100000	10
'a'	97	01100001	0001
'b'	98	01100010	01110100
'c'	99	01100011	001100
'e'	101	01100101	1100
'z'	122	01111010	00100011110

Huffman's Algorithm

- **The idea**: Create a "Huffman Tree" that will tell us a good binary representation for each character
  - Left means 0
  - Right means 1
    - Example 'b' is 10

- More frequent characters will be higher in the tree (have a shorter binary value).

- To build this tree, we must do **a few steps first**
  - Count occurrences of each unique character in the file to compress
  - Use a priority queue to order them from least to most frequent
  - Make a tree and use it



Huffman Compression – Overview

- Step 1
  - Count characters (frequency of characters in the message)
- Step 2
  - Create a Priority Queue
- Step 3
  - Build a Huffman Tree
- Step 4
  - Traverse the Tree to find the Character to Binary Mapping
- Step 5
  - Use the mapping to encode the message

Step 1: Count Characters

- Example message (input file) contents:
  ab ab cab

  file ends with an invisible EOF character

  – counts: { ' ' = 2, 'b'=3, 'a' =3, 'c' =1, EOF=1 }

byte	1	2	3	4	5	6	7	8	9	10
char	'a'	'b'	' '	'a'	'b'	' '	'c'	'a'	'b'	EOF
ASCII	97	98	32	97	98	32	99	97	98	256
binary	01100001	01100010	00100000	01100001	01100010	00100000	01100011	01100001	01100010	N/A

  – File size currently = 10 bytes = 80 bits

Step 2: Create a Priority Queue

- Each node of the PQ is a tree
  - The root of the tree is the 'key'
  - The other internal nodes hold 'subkeys'
  - The leaves hold the character values

- Insert each into the PQ using the PQ's function
  - insertItem(count, character)

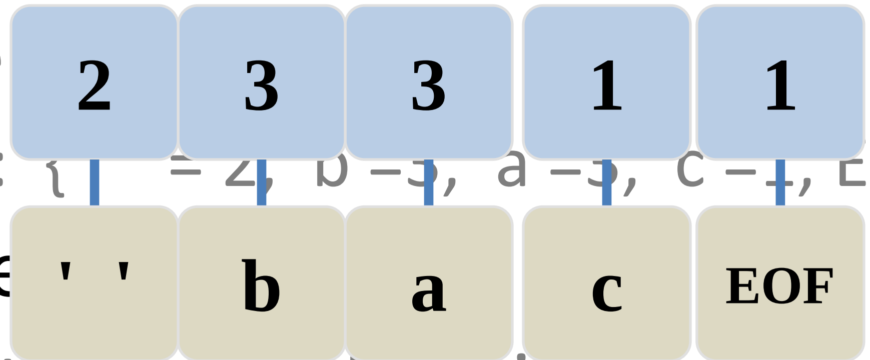
- The PQ should organize them into ascending order
  - So the smallest value is highest priority
    - *We will use an example with the PQ implemented as an ordered list*
      - *But the PQ could be implemented in whatever way works best*
        - » *could be a minheap, unordered list, or 'other'*

Step 2: PQ Creation, An Illustration

- From step 1 we have
  - counts: { ' ' = 2, 'b'=3, 'a'=3, 'c'=1, EOF=1 }
- Make these into trees
- Add the trees to a Priority Queue
  - Assume PQ is implemented as a sorted list

Step 2: PQ Creation, An Illustration

- From ste
  - counts: { = 2, b =3, a =3, c =1, EOF=1 }
- Make the

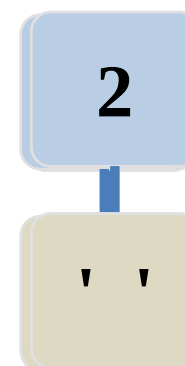


- Add the trees to a Priority Queue
  - Assume PQ is implemented as a sorted list

Step 2: PQ Creation, An Illustration

- From step 1 we have
  - counts: { ' ' = 2, 'b'=3, 'a'=3, 'c'=1, EOF=1 }
- Make these into trees
- Add the trees to a Priority Queue
  - *Assume PQ is implemented as a sorted list*

Recall: Each Insert is an *O(n)* operation
This is done *n* times. So this is $O(n^2)$



Step 3: Build the Huffman Tree

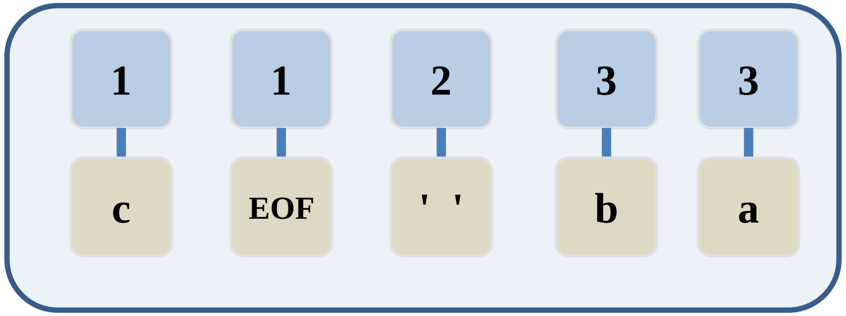
- *Aside: All nodes should be in the PQ*
- While PQ.size() > 1
  - Remove the two highest priority (rarest) nodes
    - Removal done using PQ's removeMin() function
  - Combine the two nodes into a single node
    - So the new node is a tree with
      - root has key value = sum of keys of nodes being combined
      - left subtree is the first removed node
      - right subtree is the second removed node
  - Insert the combined node back into the PQ
- end While

- Remove the one node from the PQ
  - This is the Huffman Tree

Example next slide

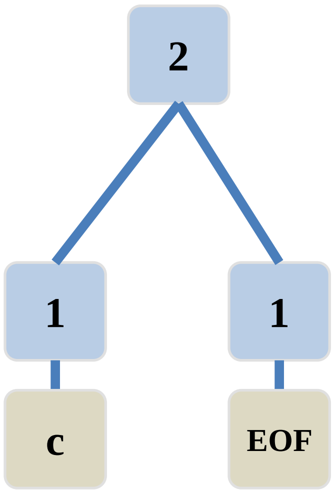
Step 3a: Building Huffman Tree, Illus.

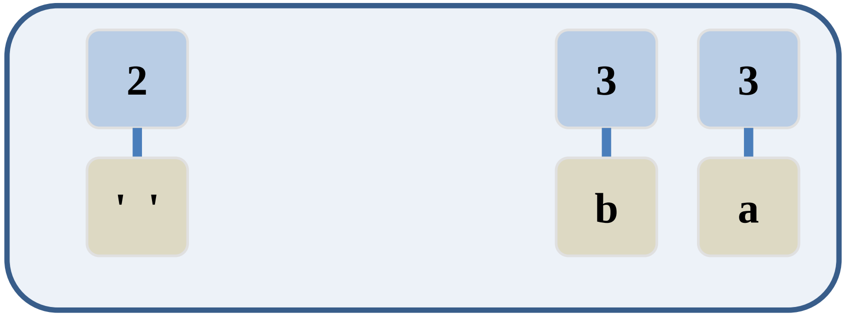
- Remove the two highest priority (rarest) nodes



Step 3b: Building Huffman Tree, Illus.

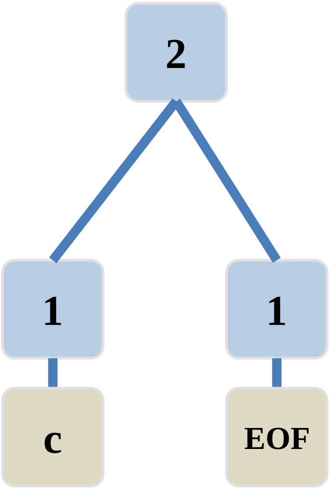
- Combine the two nodes into a single node

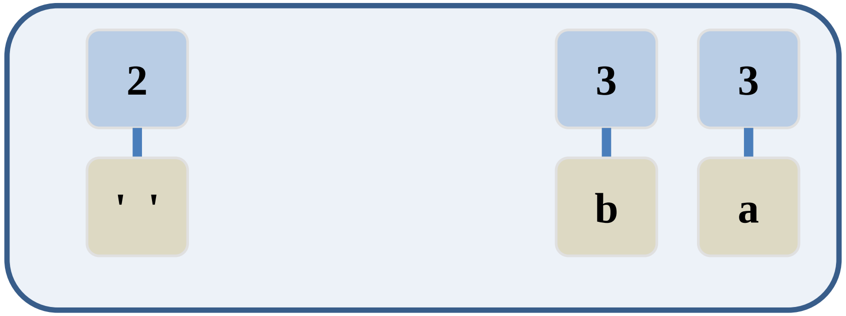




Step 3c: Building Huffman Tree, Illus.

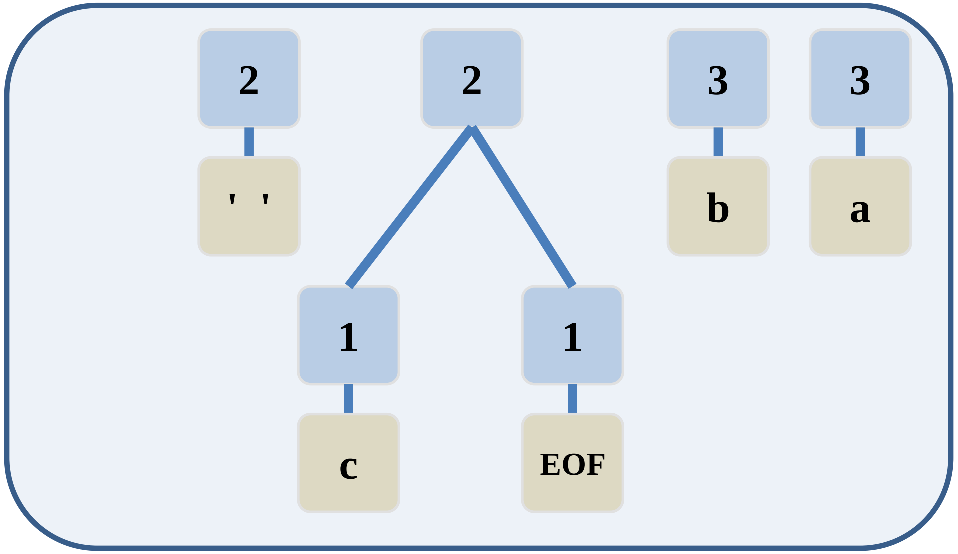
- Insert the combined node back into the PQ





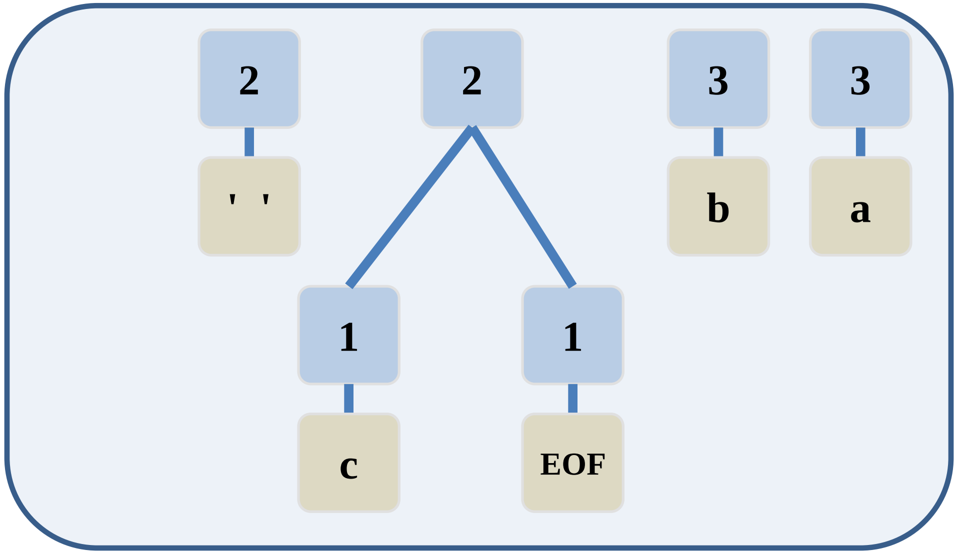
Step 3d: Building Huffman Tree, Illus.

- PQ has 4 nodes still, so repeat



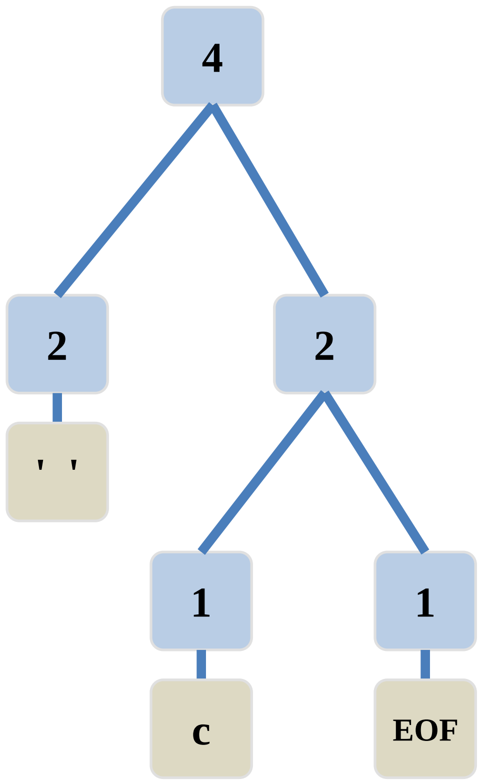
Step 3a: Building Huffman Tree, Illus.

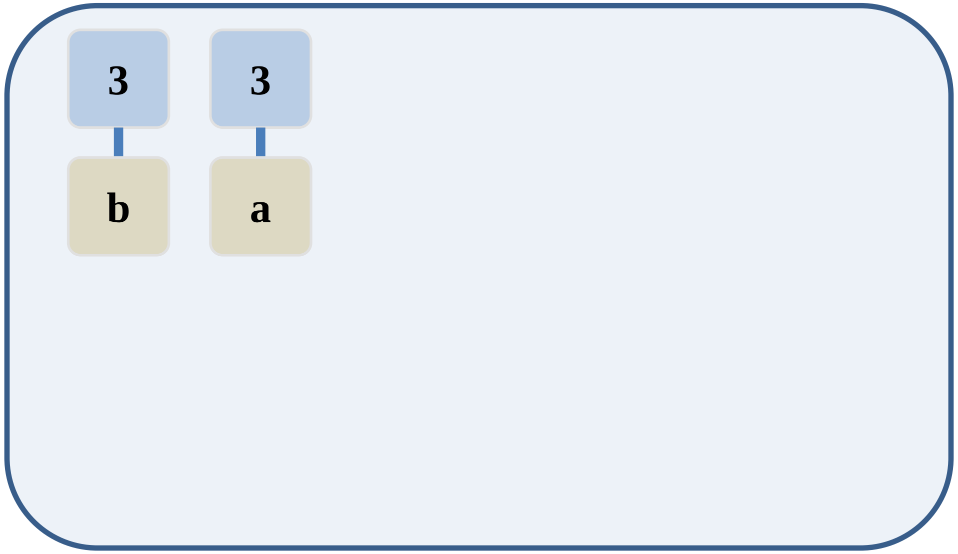
- Remove the two highest priority (rarest) nodes



Step 3b: Building Huffman Tree, Illus.

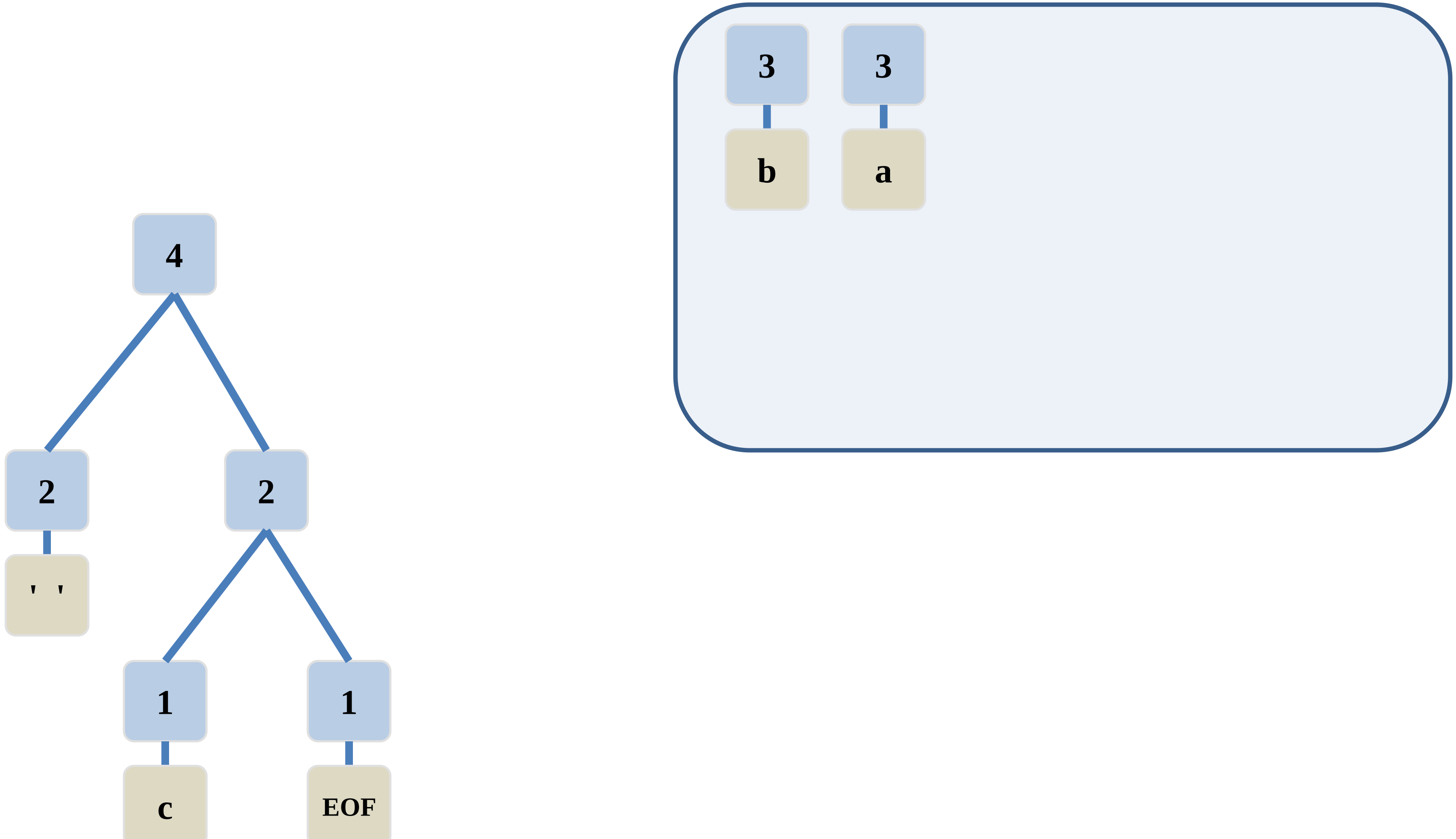
- Combine the two nodes into a single node





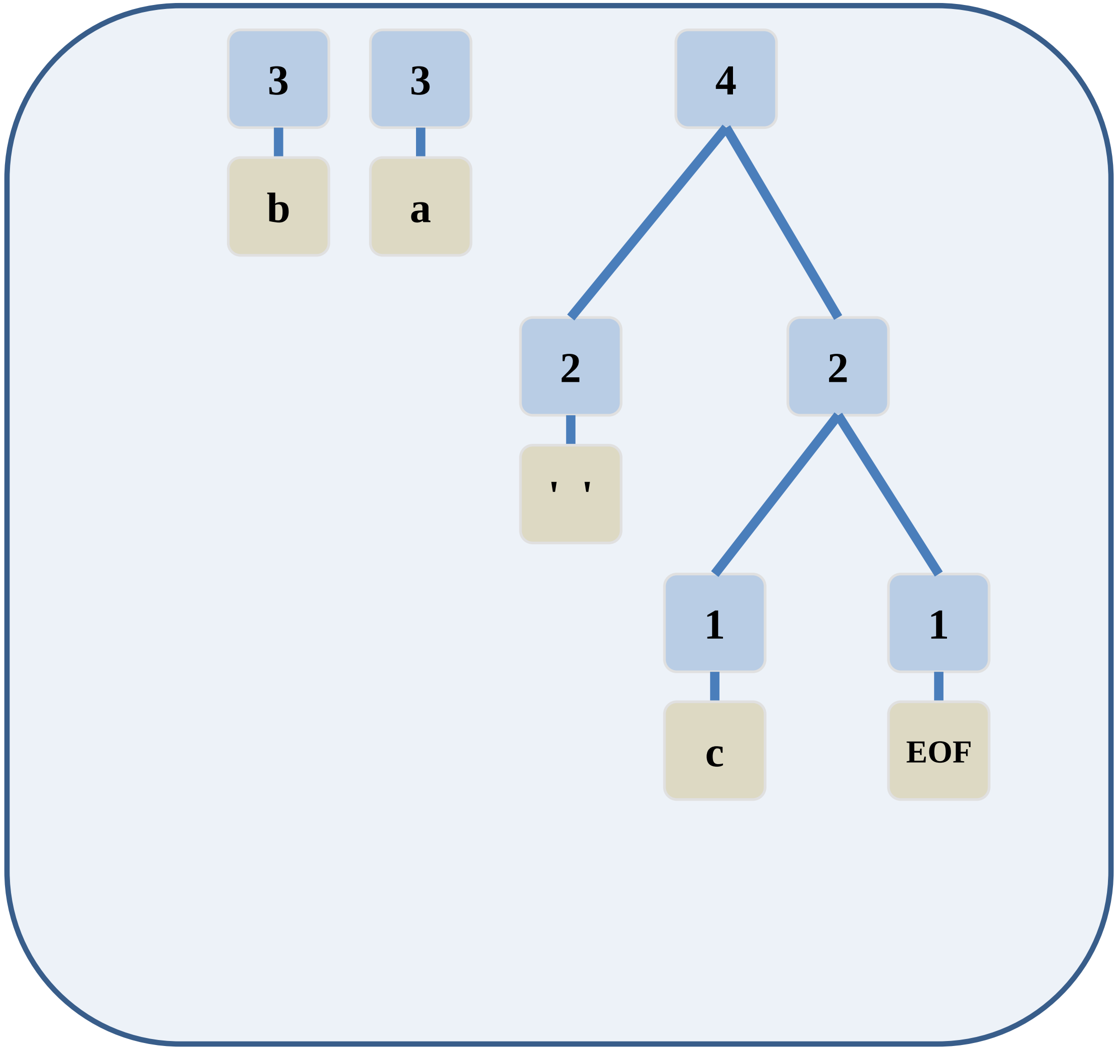
Step 3c: Building Huffman Tree, Illus.

- Insert the combined node back into the PQ



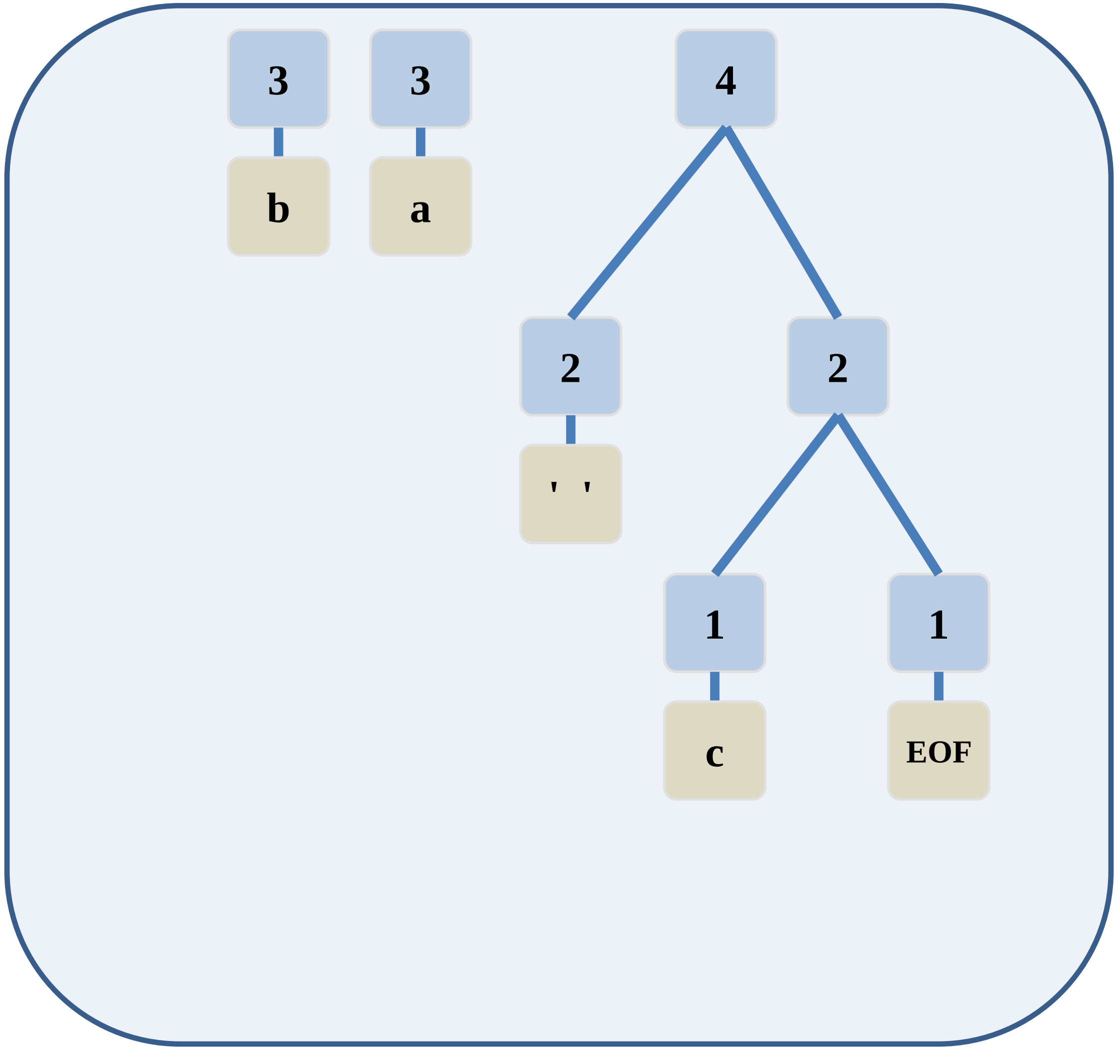
Step 3d: Building Huffman Tree, Illus.

- 3 nodes remain in PQ, repeat again



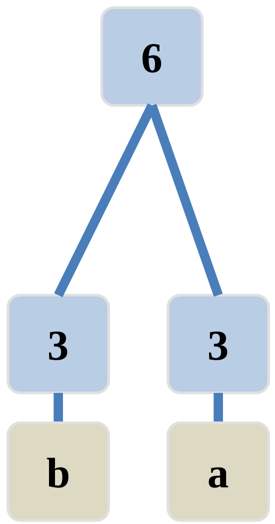
Step 3a: Building Huffman Tree, Illus.

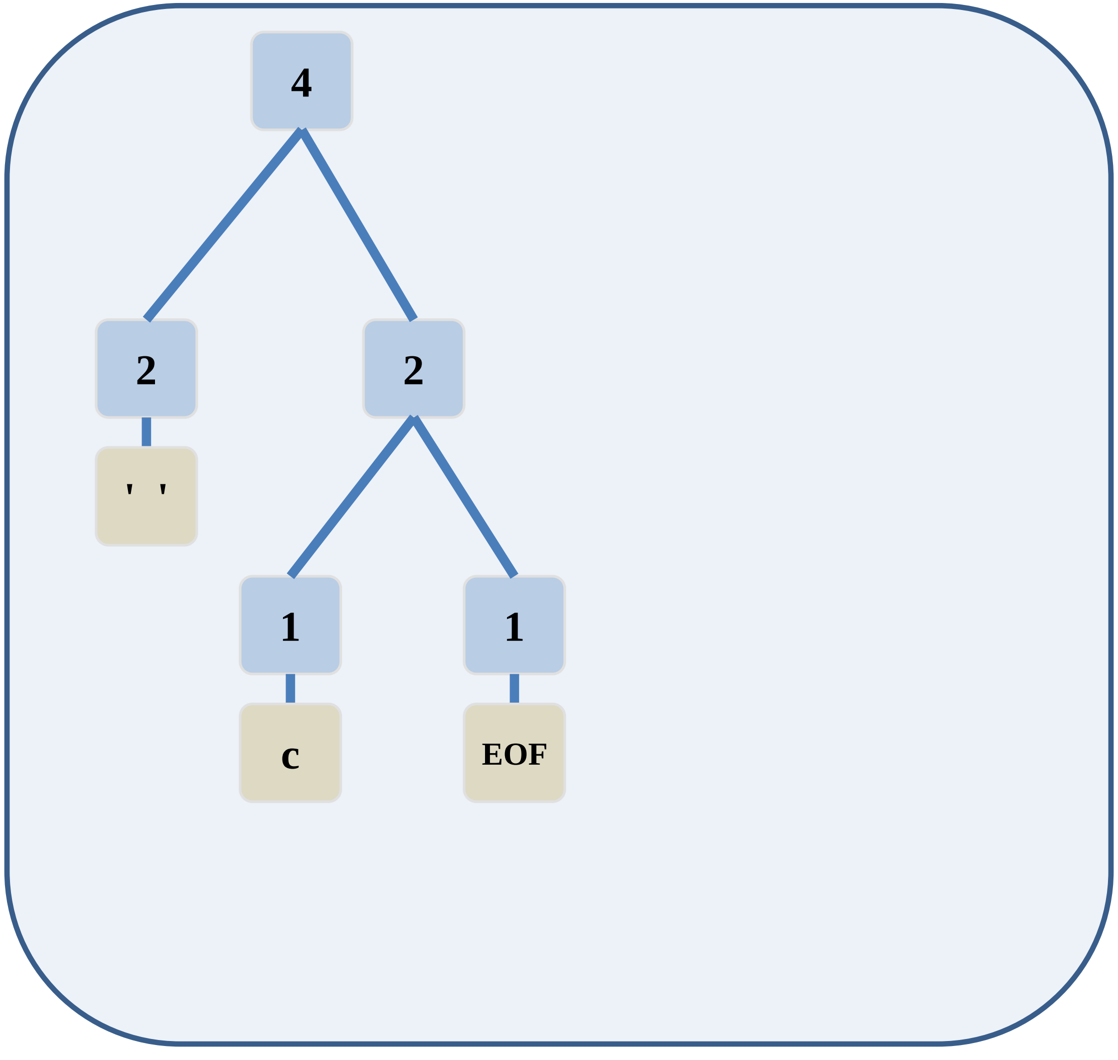
- Remove the two highest priority (rarest) nodes



Step 3b: Building Huffman Tree, Illus.

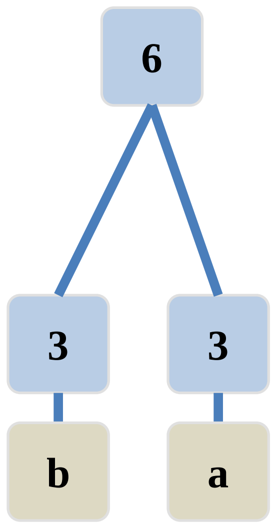
- Combine the two nodes into a single node

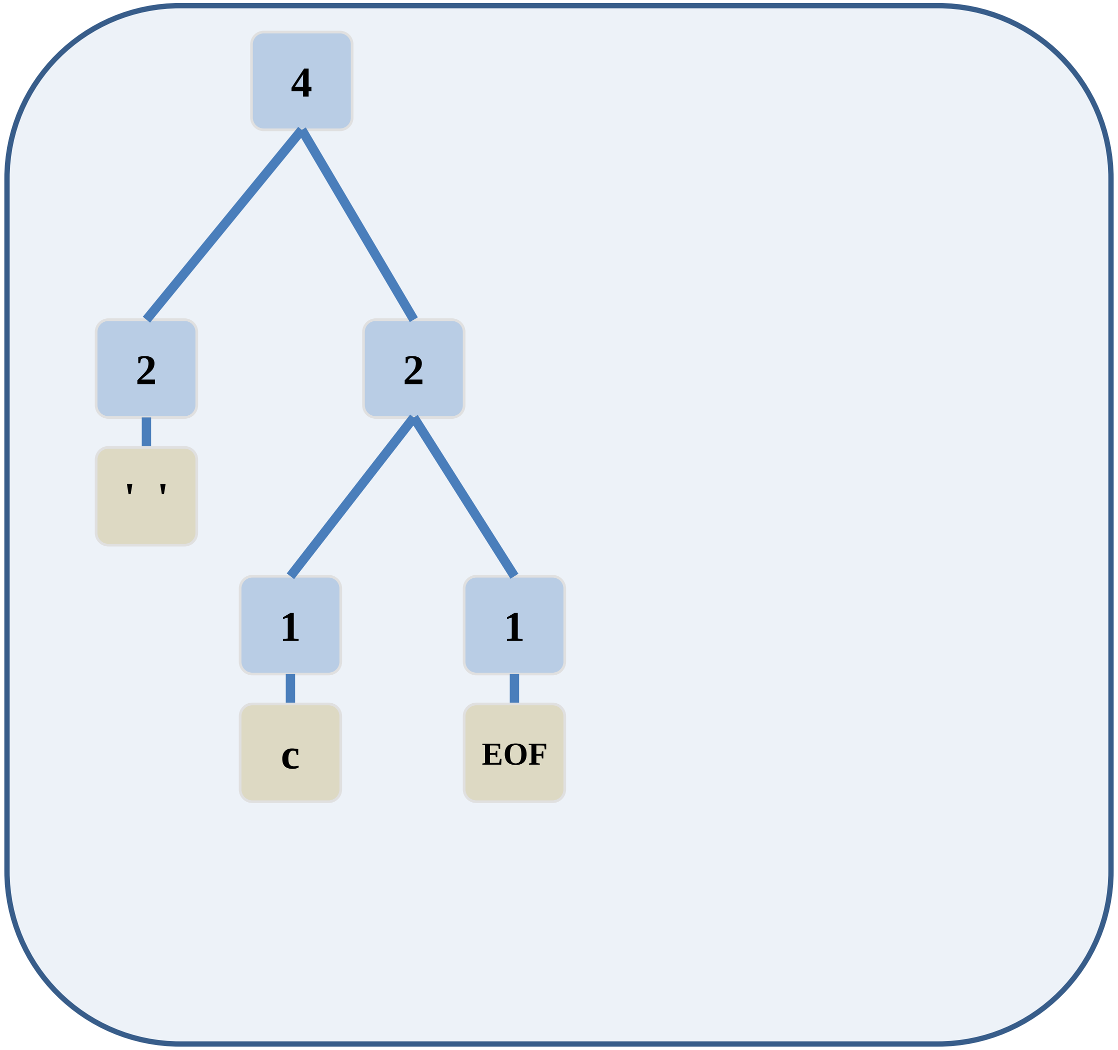




Step 3c: Building Huffman Tree, Illus.

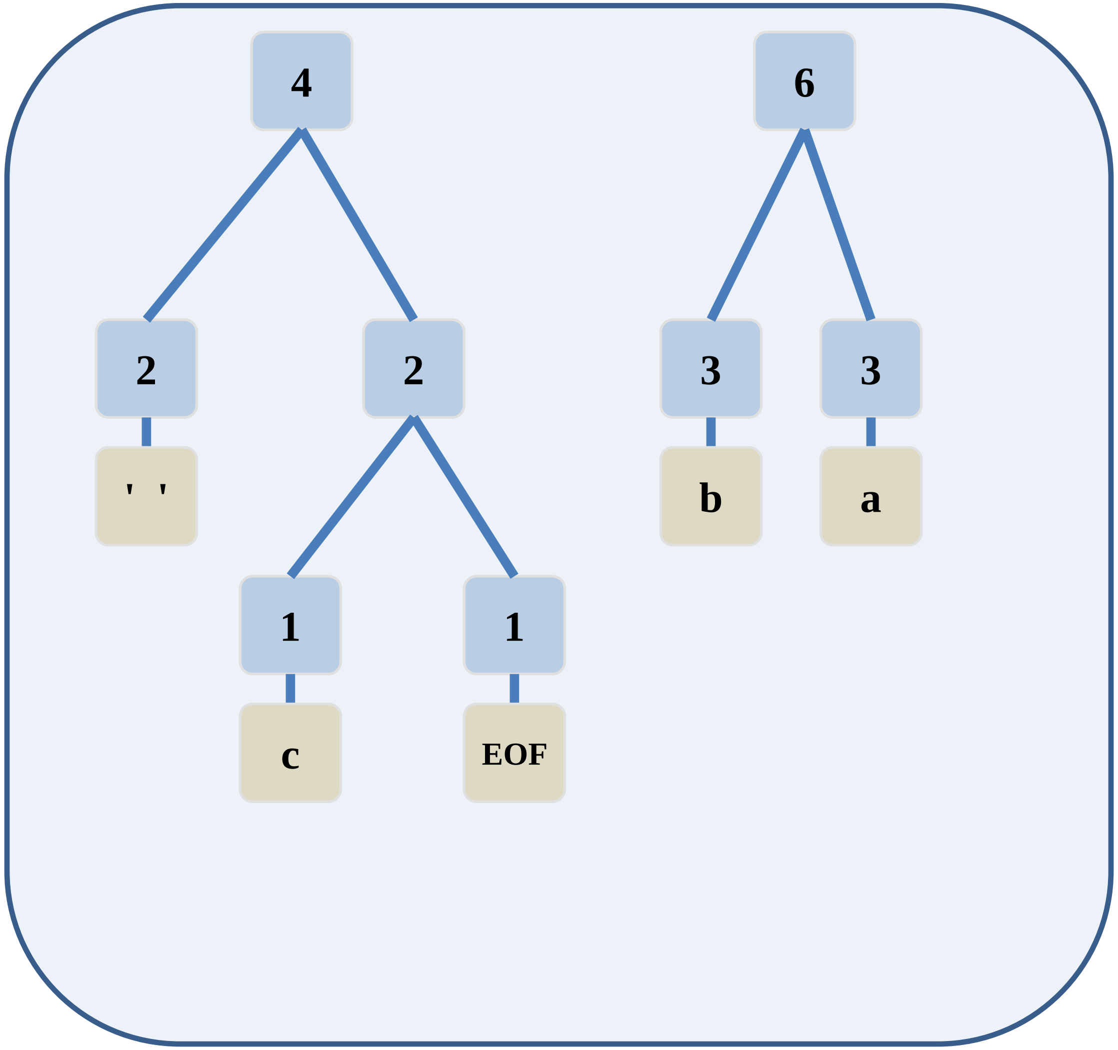
- Insert the combined node back into the PQ





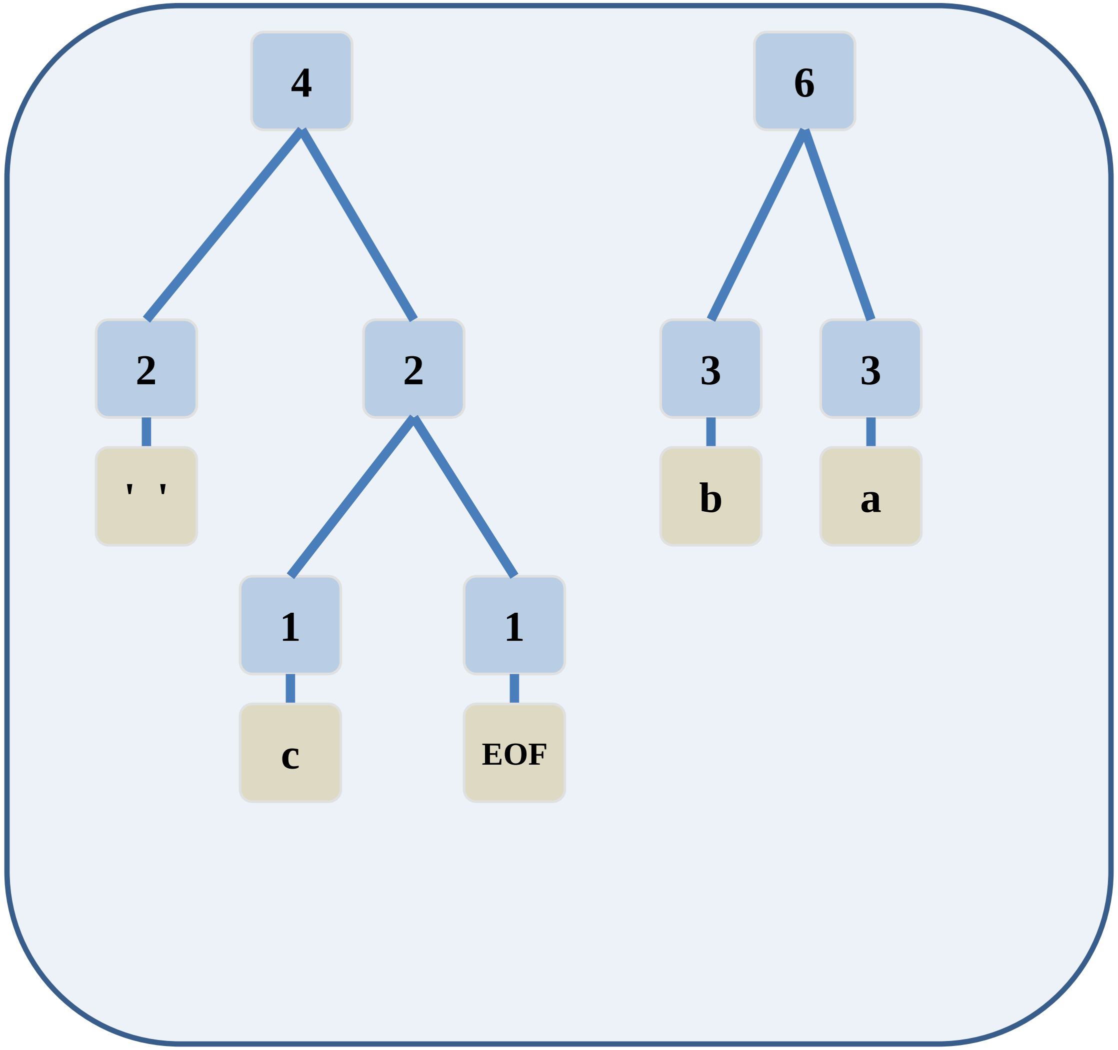
Step 3d: Building Huffman Tree, Illus.

- 2 nodes still in PQ, repeat one more time



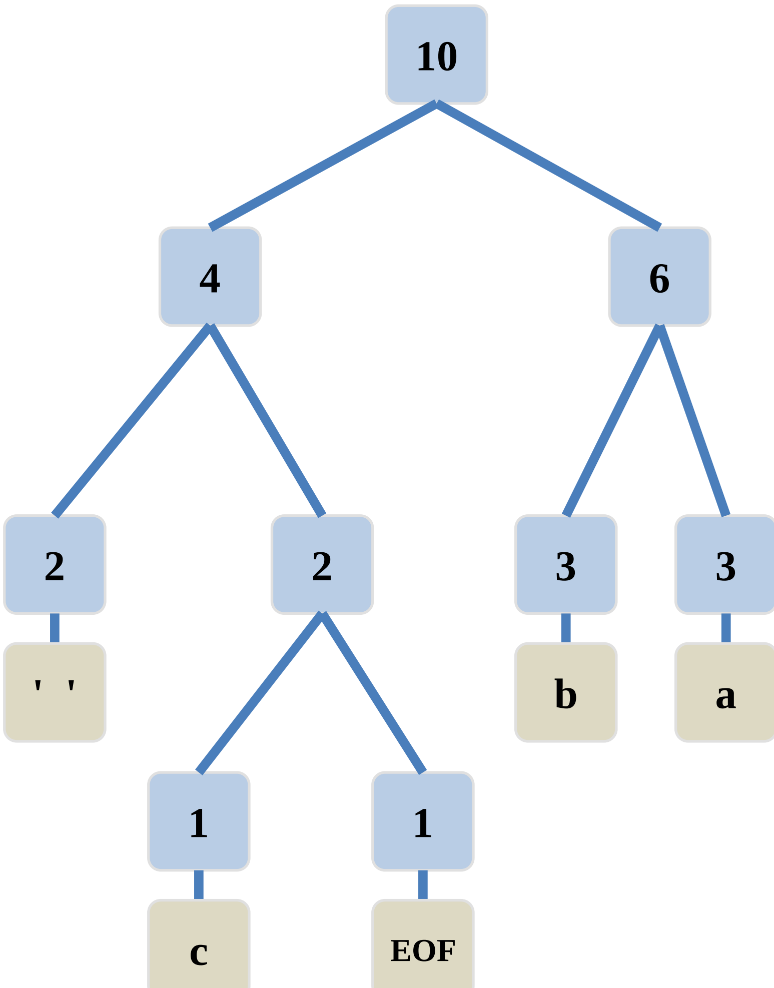
Step 3a: Building Huffman Tree, Illus.

- Remove the two highest priority (rarest) nodes



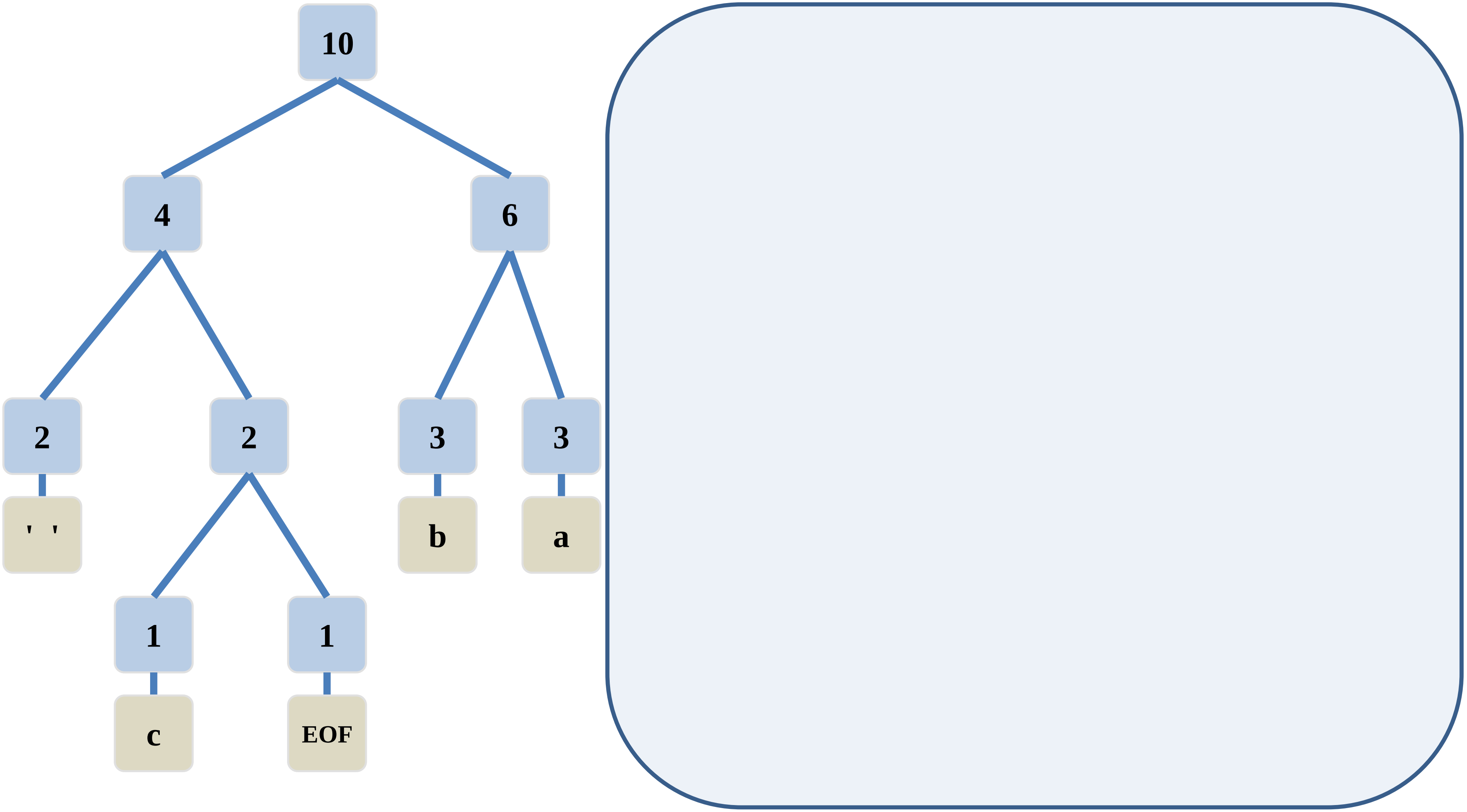
Step 3b: Building Huffman Tree, Illus.

- Combine the two nodes into a single node



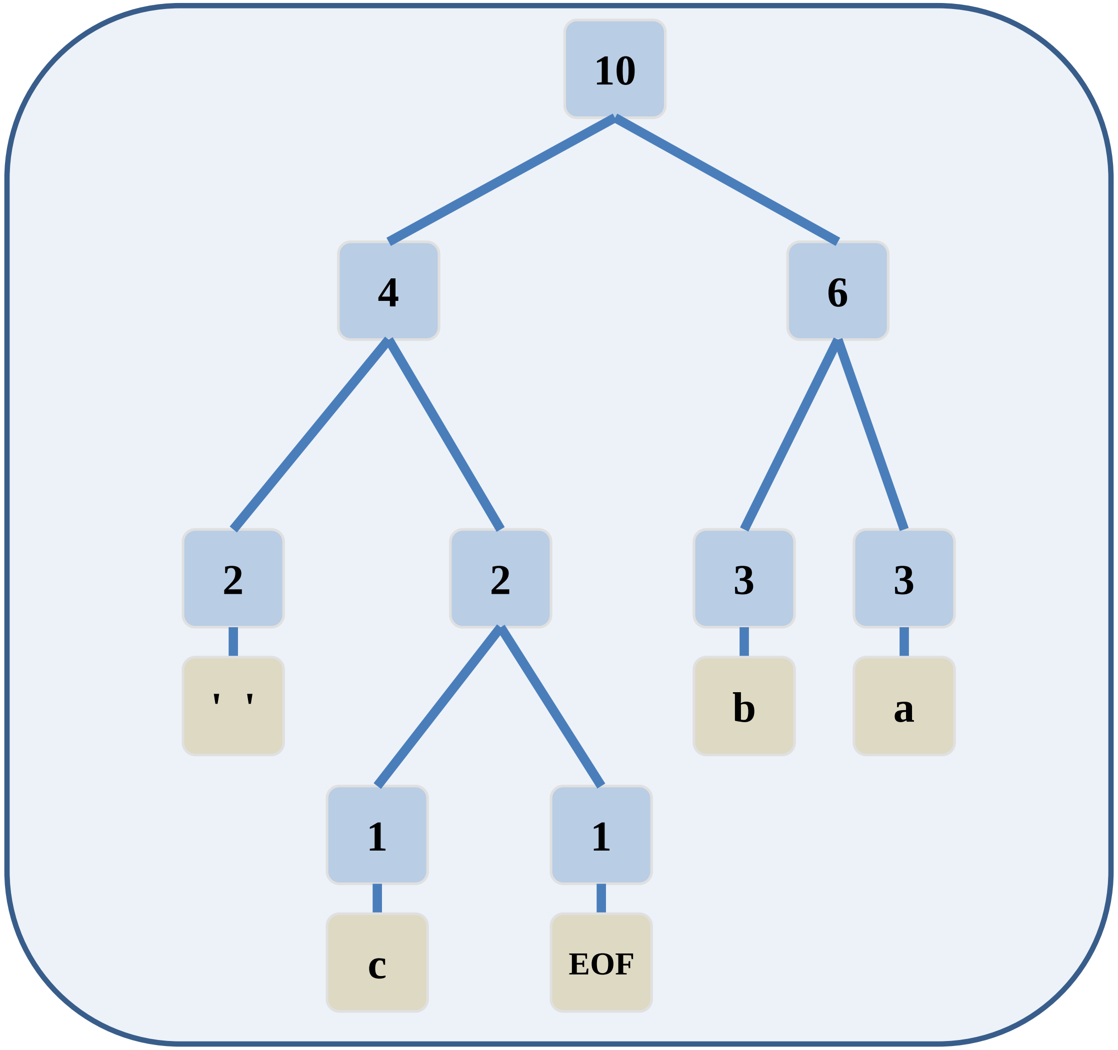
Step 3c: Building Huffman Tree, Illus.

- Insert the combined node back into the PQ



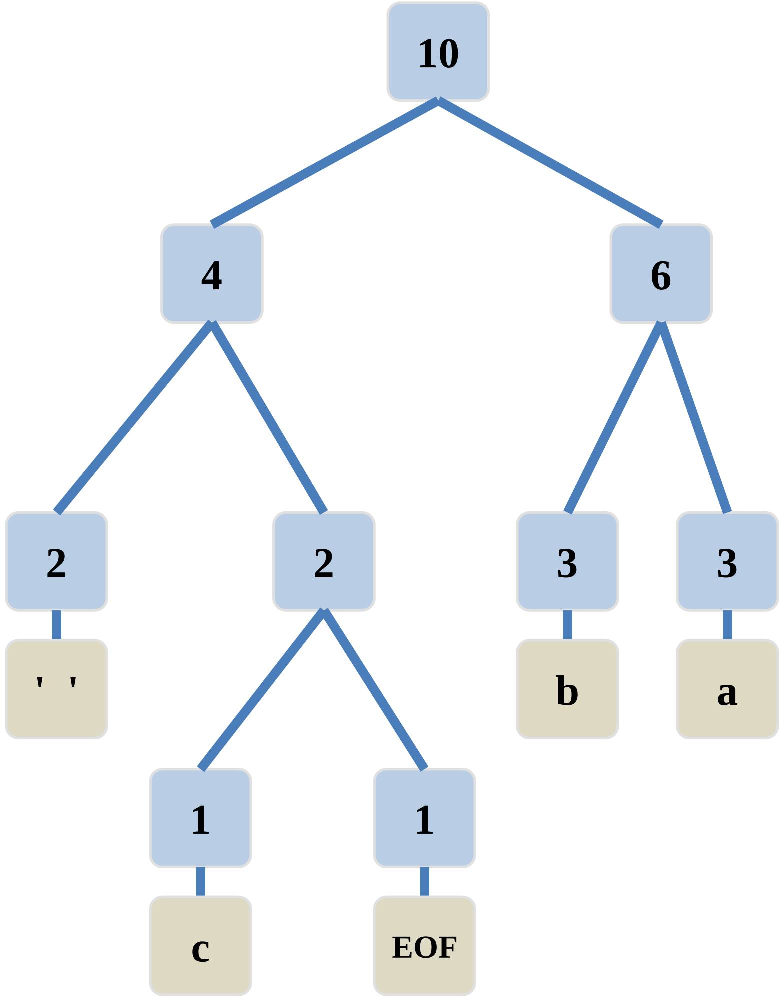
Step 3d: Building Huffman Tree, Illus.

- Only 1 node remains in the PQ, so while loop ends



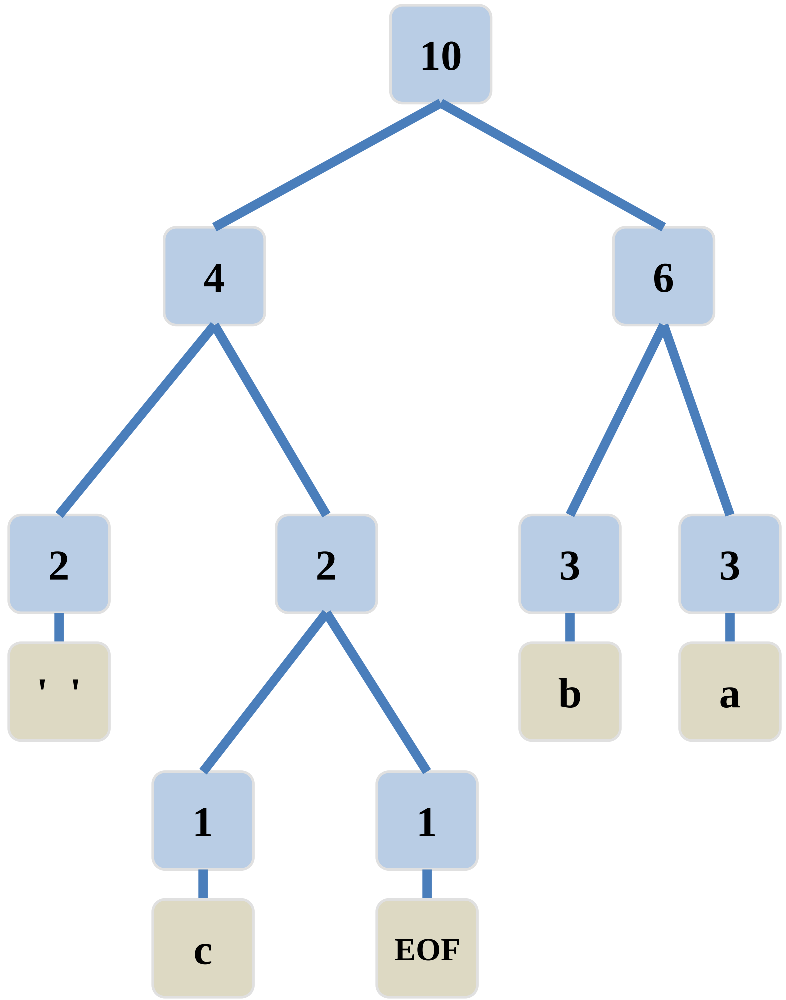
Step 3: Building Huffman Tree, Illus.

- Huffman tree is complete



Step 4: Traverse Tree to Find the Character to Binary Mapping

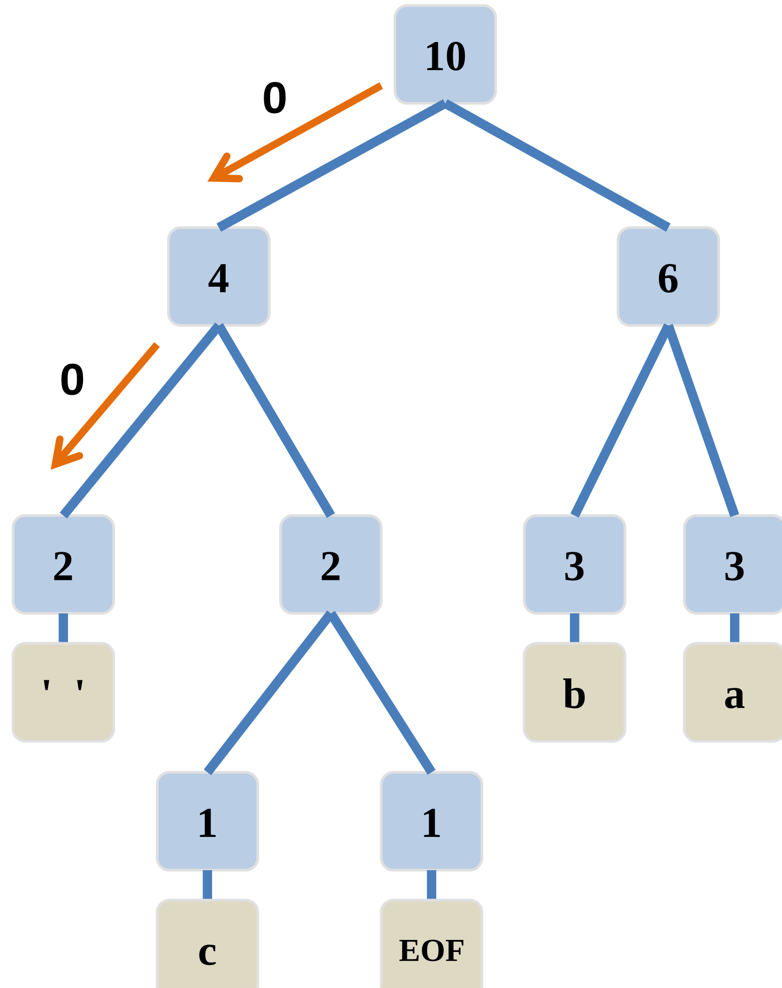
- ' ' =
- 'c' =
- EOF =
- 'b' =
- 'a' =



Recall
Left is 0
Right is 1

Step 4: Traverse Tree to Find the Character to Binary Mapping

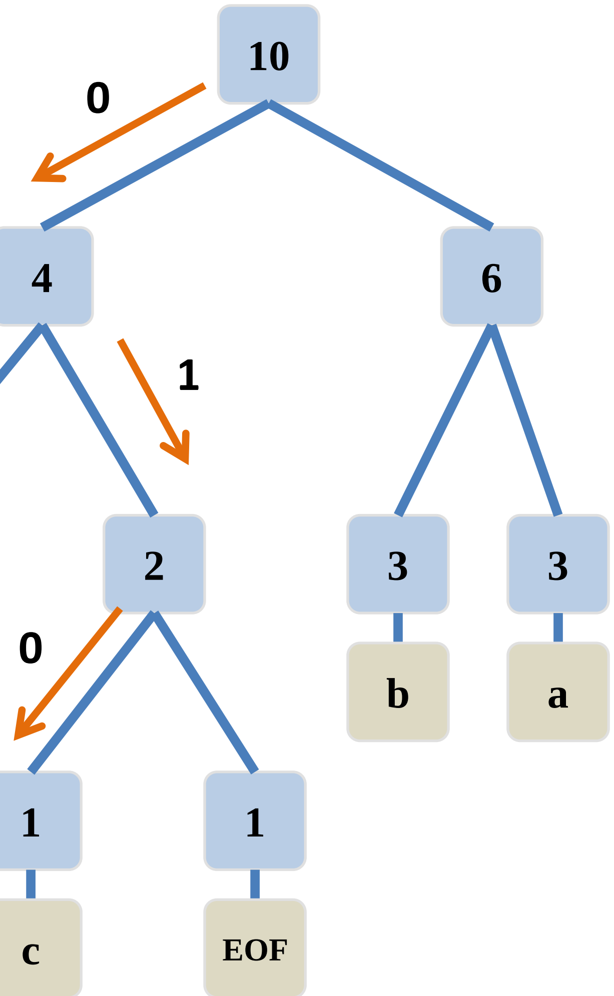
- **' ' = 00**
- 'c' =
- EOF =
- 'b' =
- 'a' =



Recall
Left is 0
Right is 1

Step 4: Traverse Tree to Find the Character to Binary Mapping

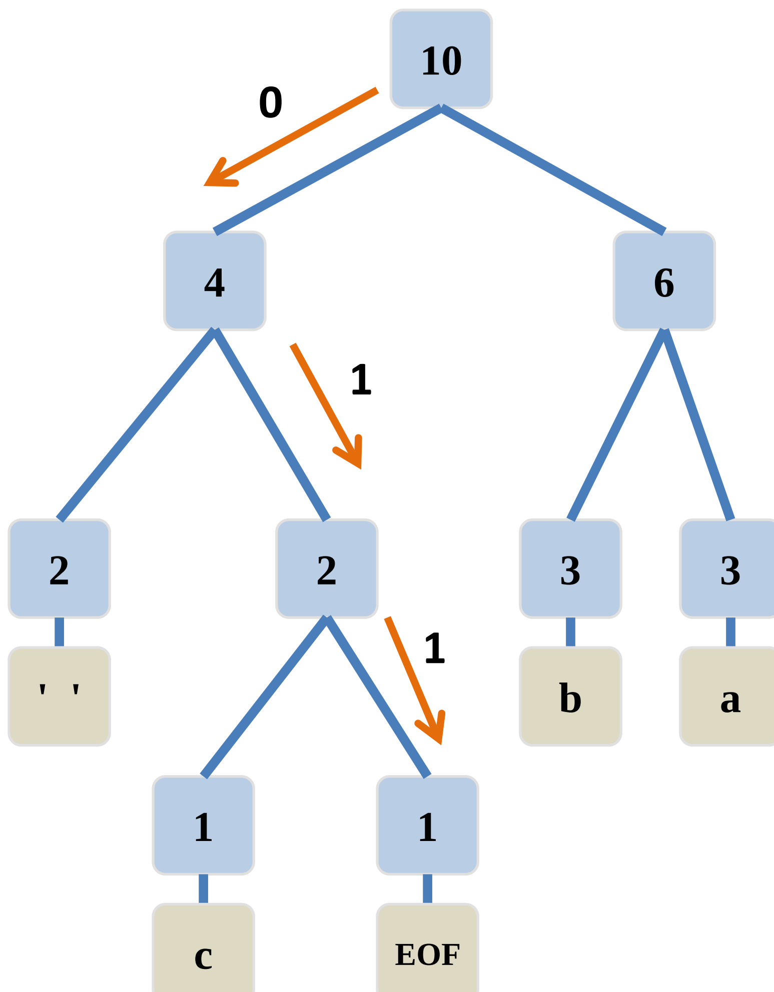
- ' ' = 00
- 'c' = **010**
- EOF =
- 'b' =
- 'a' =



Recall
Left is 0
Right is 1

Step 4: Traverse Tree to Find the Character to Binary Mapping

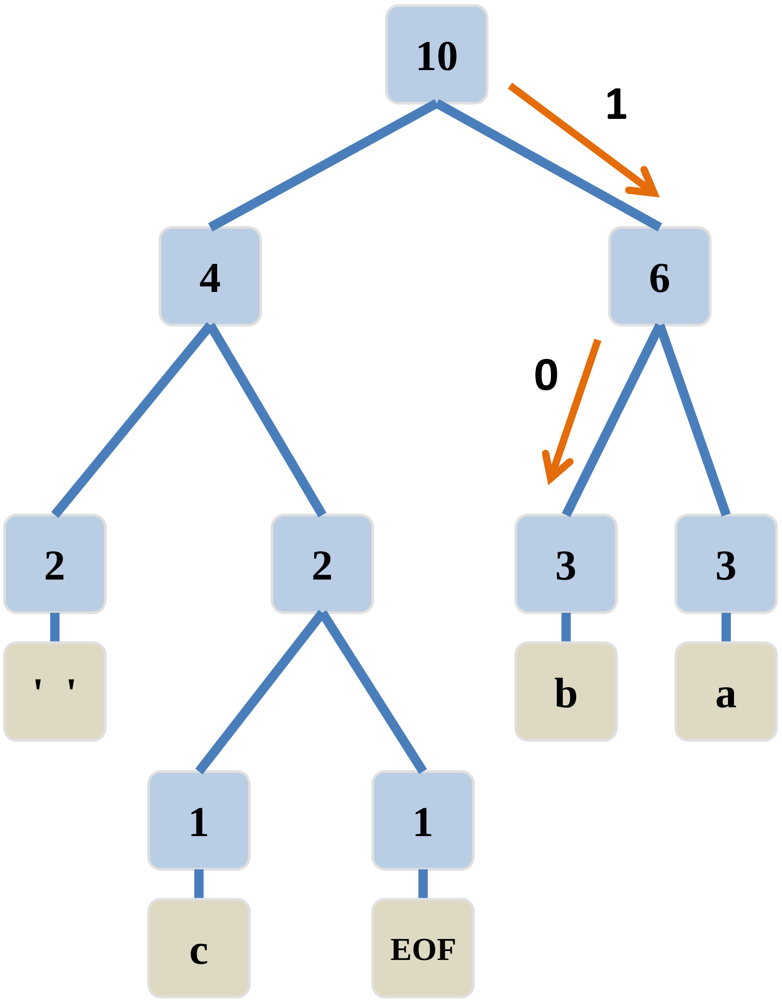
- ' ' = 00
- 'c' = 010
- EOF = **011**
- 'b' =
- 'a' =



Recall
Left is 0
Right is 1

Step 4: Traverse Tree to Find the Character to Binary Mapping

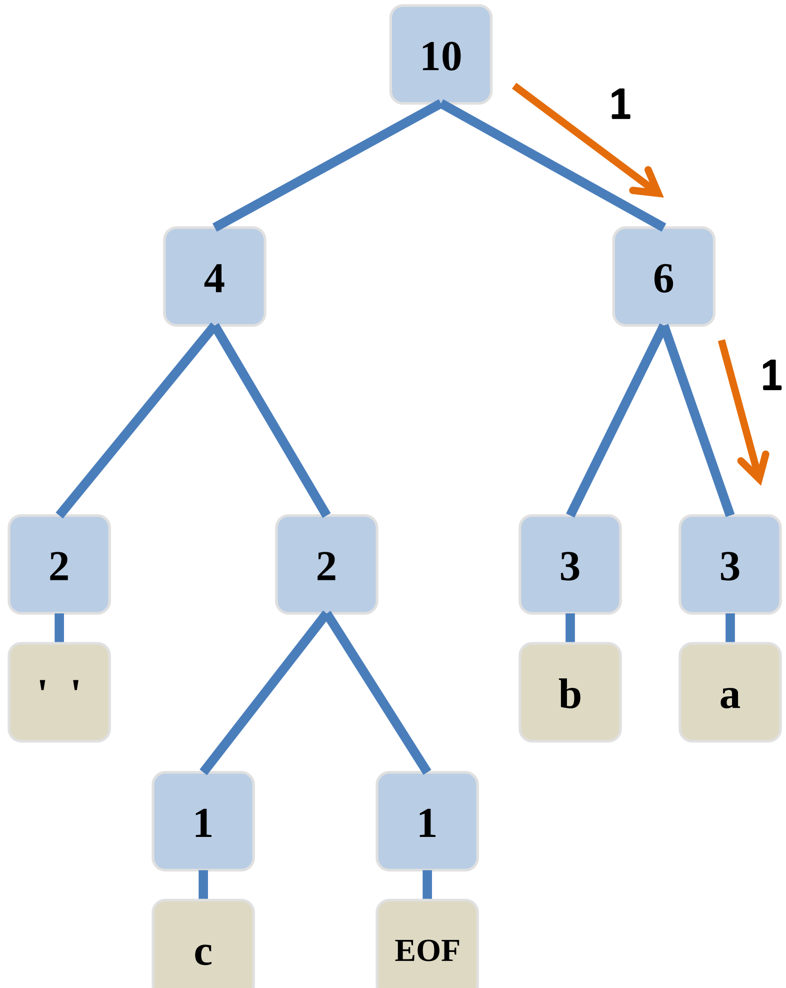
- ' ' = **00**
- 'c' = **010**
- EOF = **011**
- 'b' = **10**
- 'a' =



Recall
Left is 0
Right is 1

Step 4: Traverse Tree to Find the Character to Binary Mapping

- ' ' = 00
- 'c' = 010
- EOF = 011
- 'b' = 10
- 'a' = 11



Recall
Left is 0
Right is 1

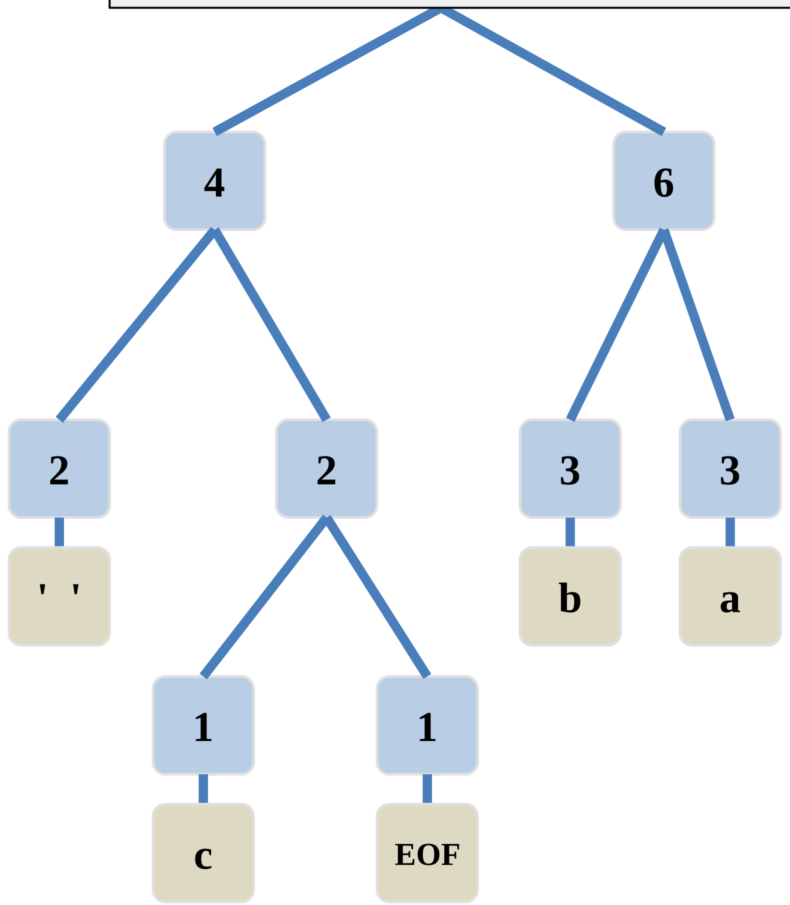
Step 5: Encode the Message

file ends with an invisible EOF character

- ' ' = 00
- 'c' = 010
- EOF = 011
- 'b' = 10
- 'a' = 11

- Example message (input file) contents:

  ab ab cab



Challenge: Encode the Message

- ' ' = 00
- 'c' = 010
- EOF = 011
- 'b' = 10
- 'a' = 11

- Example message (input file) contents:

  ab ab cab

file ends with an invisible EOF character

Step 5: Encode the Message

file ends with an invisible EOF character

- ' ' = 00
- 'c' = 010
- EOF = 011
- 'b' = 10
- 'a' = 11

- Example message (input file) contents:

ab ab cab

- 11

Step 5: Encode the Message

file ends with an invisible EOF character

- ' ' = 00
- 'c' = 010
- EOF = 011
- 'b' = 10
- 'a' = 11

- Example message (input file) contents:

  ab ab cab

- 11<u>10</u>

Step 5: Encode the Message

- ' ' = 00
- 'c' = 010
- EOF = 011
- 'b' = 10
- 'a' = 11

- Example message (input file) contents:

ab ab cab

file ends with an invisible EOF character

- 111000

Step 5: Encode the Message

file ends with an invisible EOF character

- ' ' = 00
- 'c' = 010
- EOF = 011
- 'b' = 10
- 'a' = 11

- Example message (input file) contents:

ab ab cab

- 11100011

Step 5: Encode the Message

file ends with an invisible EOF character

- ' ' = 00
- 'c' = 010
- EOF = 011
- 'b' = 10
- 'a' = 11

- Example message (input file) contents:

  ab ab cab

- 1110001110

Step 5: Encode the Message

file ends with an invisible EOF character

- ' ' = 00
- 'c' = 010
- EOF = 011
- 'b' = 10
- 'a' = 11

- Example message (input file) contents:

  ab ab cab

- 111000111000

Step 5: Encode the Message

file ends with an invisible EOF character

- ' ' = 00
- 'c' = 010
- EOF = 011
- 'b' = 10
- 'a' = 11

- Example message (input file) contents:

ab ab cab

- 11100011100010

Step 5: Encode the Message

file ends with an invisible EOF character

- ' ' = 00
- 'c' = 010
- EOF = 011
- 'b' = 10
- 'a' = 11

- Example message (input file) contents:

  ab ab cab

- 11100011100001011

Step 5: Encode the Message

file ends with an invisible EOF character

- ' ' = 00
- 'c' = 010
- EOF = 011
- 'b' = 10
- 'a' = 11

- Example message (input file) contents:

  ab ab cab

- 1110001110000101110

Step 5: Encode the Message

file ends with an invisible EOF character

- ' ' = 00
- 'c' = 010
- EOF = 011
- 'b' = 10
- 'a' = 11

- Example message (input file) contents:

  ab ab cab

- 1110001110000101110011

Step 5: Encode the Message

file ends with an invisible EOF character

- ' ' = 00
- 'c' = 010
- EOF = 011
- 'b' = 10
- 'a' = 11

- Example message (input file) contents:

  ab ab cab

- 11<u>10</u>00<u>11</u>10<u>00</u>010<u>11</u>10<u>011</u>

- Count the bits used = 22 bits
- versus the 80
  - previously needed
- File is almost ¼ the size
  - lots of savings

Decompression

- From the previous tree shown we now have the message characters encoded as:

char	'a'	'b'	' '	'a'	'b'	' '	'c'	'a'	'b'	EOF
binary	11	10	00	11	10	00	010	11	10	011

- Which compresses to bytes 3 like so:

byte	1	2	3
char	a b a	b c a	b EOF
binary	11 10 00 11	10 00 010 1	1 10 011

- How to decompress?
  - *Hint: Lookup table is not the best answer,*
    what is the first symbol?... 1=? or is it 11? or 111? or 1110? or…

Decompression via Tree

- The tree is known to the recipient of the message
  - So use it

- To identify symbols we will
  Apply the Prefix Property
  - No encoding *A* is the prefix of another encoding *B*
  - Never will have x→**011** and y→**011**100110

Decompression via Tree

- Apply the Prefix Property
  - No encoding *A* is the prefix of another encoding *B*
  - Never will have x→**011** and y→**011**100110

- the Algorithm
  - Read each bit one at a time from the input
  - If the bit is 0 go left in the tree
  - Else if the bit is 1 go right
  - If you reach a leaf node
    - output the character at that leaf
    - and go back to the tree root

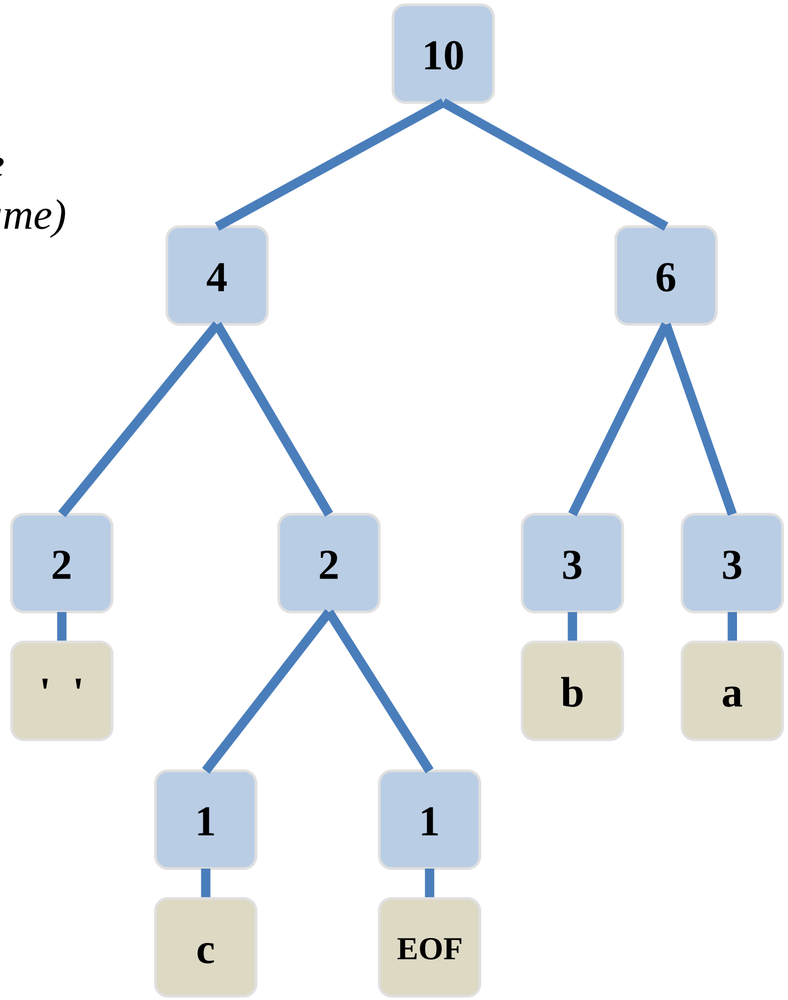
Decompressing Example

- Say the encrypted message was:
- `1011101000110101011011`
  - *note: this is NOT the same message as the encryption just done (but the tree the is same)*

- Read each bit one at a time

- If it is 0 go left
- If it is 1 go right

- If you reach a leaf, output the character there and go back to the tree root



Class Activity: Decompressing Example

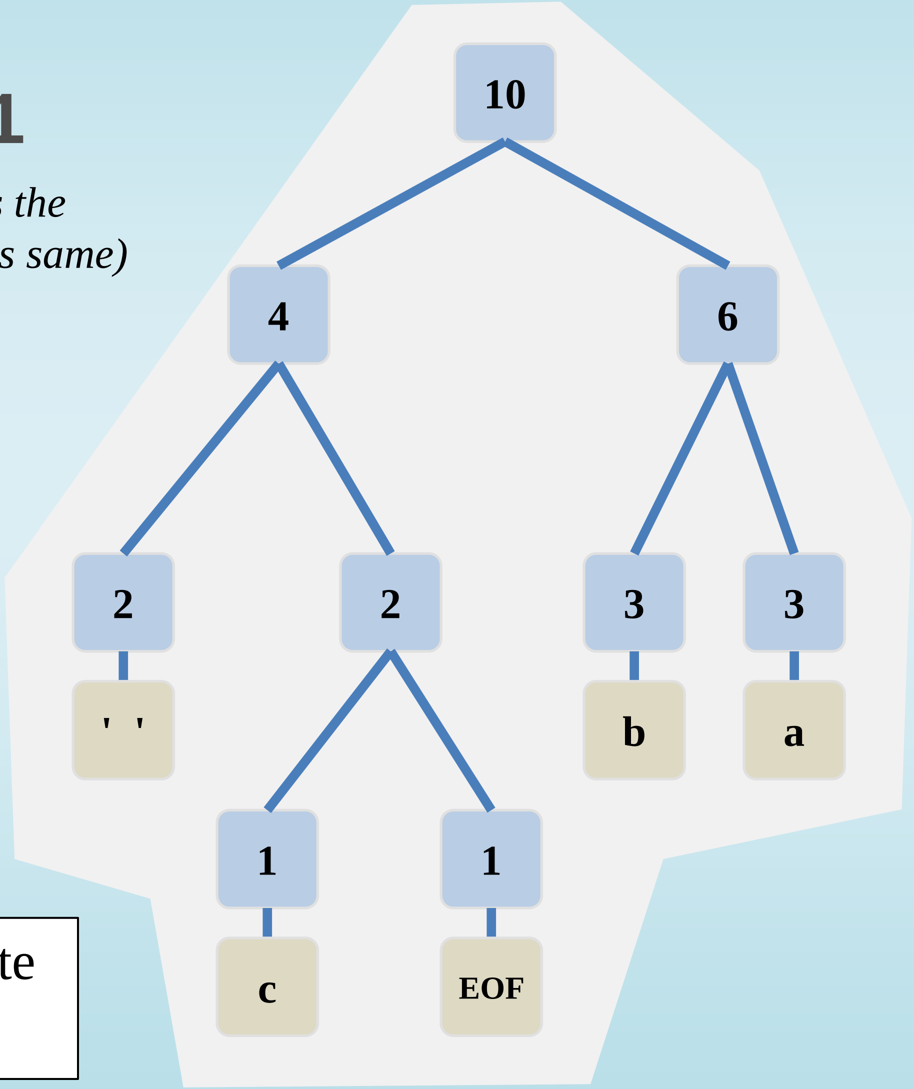
- Say the encrypted message was:
- **1011101000110101011011**
  - *note: this is NOT the same message as the encryption just done (but the tree the is same)*

- Read each bit one at a time

- If it is 0 go left
- If it is 1 go right

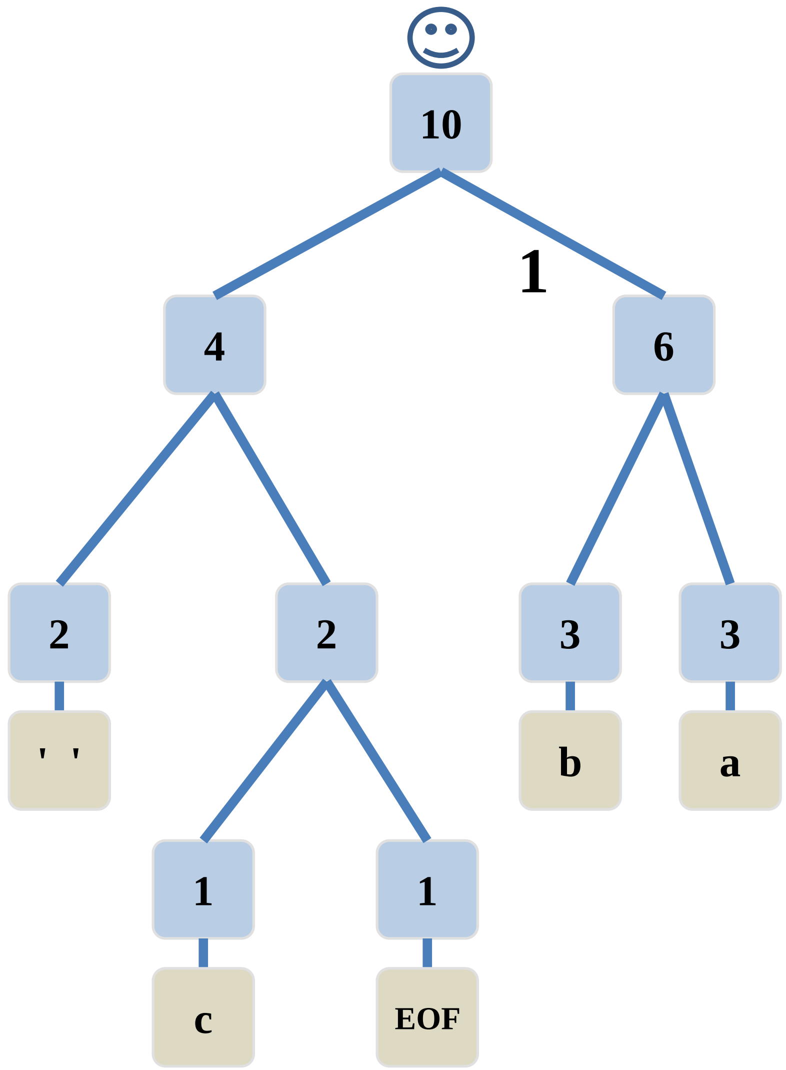
- If you reach a leaf, output the character there and go back to the tree root

- Pause for students to complete



Decompressing Example

- Say the encrypted message was:
- **1011101000110101101011011**



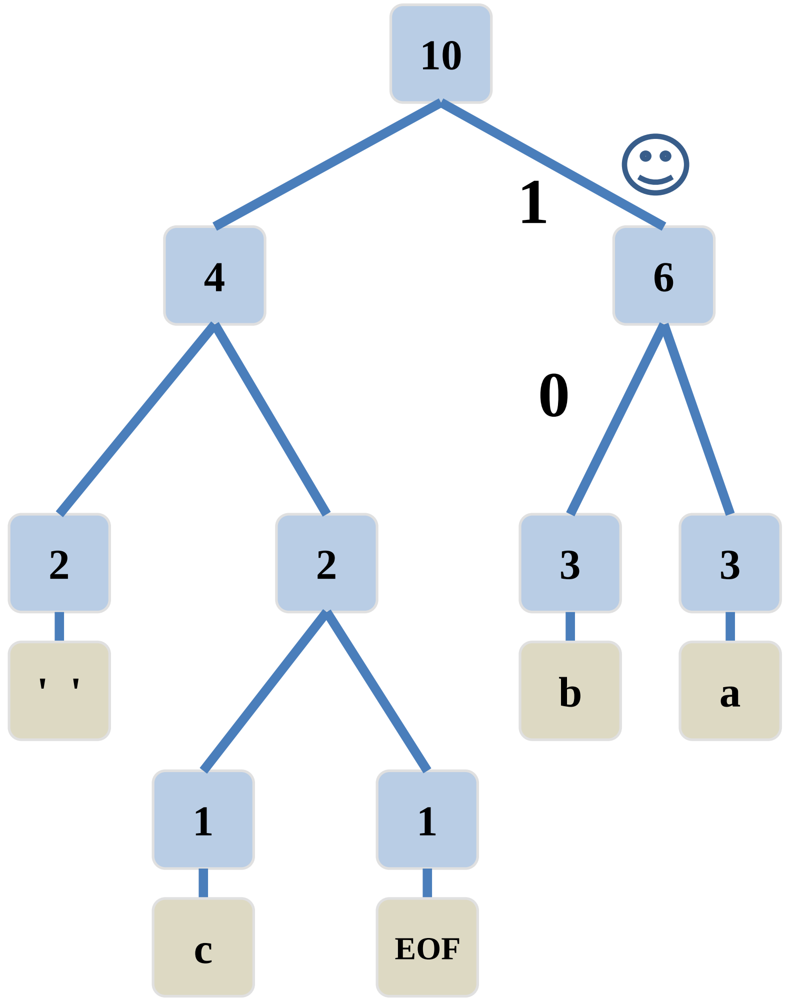
- Read each bit one at a time
- If it is 0 go left
- If it is 1 go right
- If you reach a leaf, output the character there and go back to the tree root

Decompressing Example

- Say the encrypted message was:
- **1011101000110101011011**

b

- Read each bit one at a time
- If it is 0 go left
- If it is 1 go right
- If you reach a leaf, output the character there and go back to the tree root

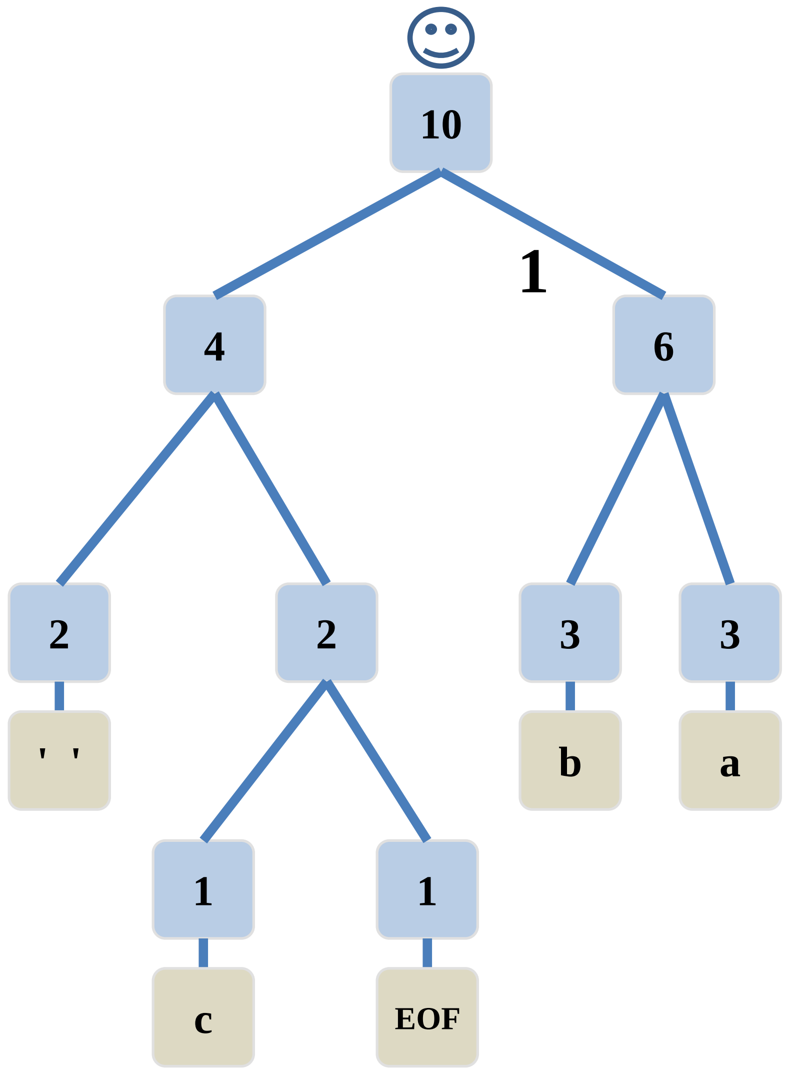


Decompressing Example

- Say the encrypted message was:
- **1011010001101011011**

b

- Read each bit one at a time
- If it is 0 go left
- If it is 1 go right
- If you reach a leaf, output the character there and go back to the tree root

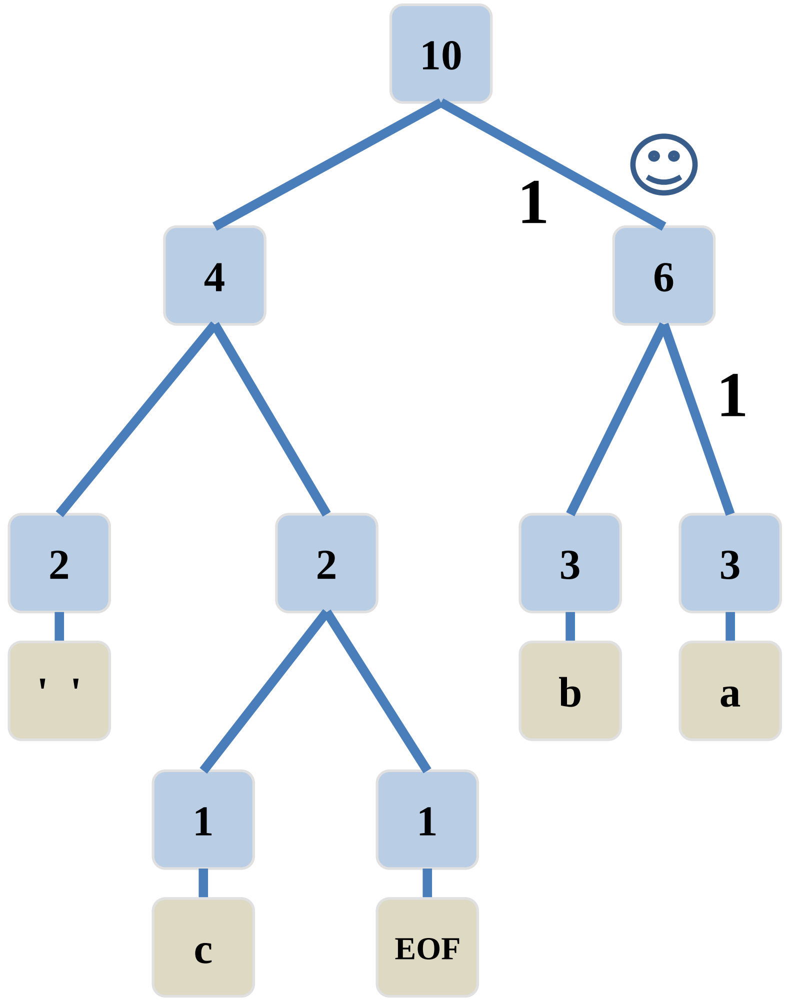


Decompressing Example

- Say the encrypted message was:
- **1011101000110101011011**

b	a

- Read each bit one at a time
- If it is 0 go left
- If it is 1 go right
- If you reach a leaf, output the character there and go back to the tree root

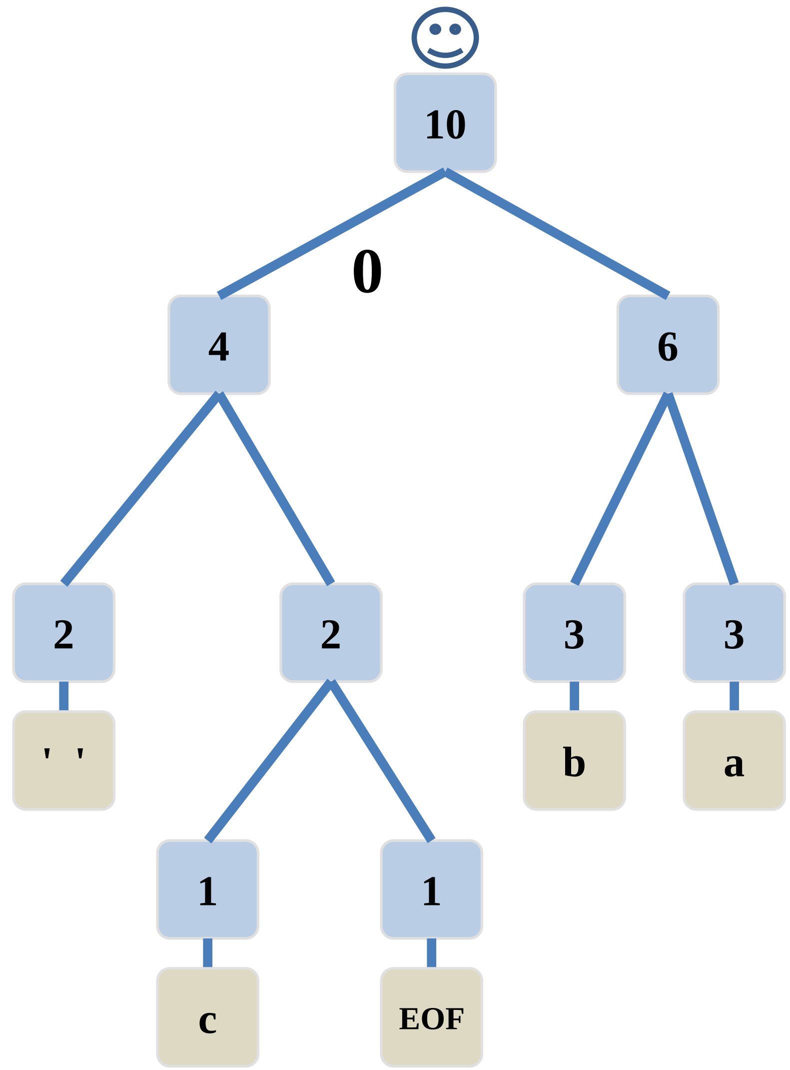


Decompressing Example

- Say the encrypted message was:
- **1011010001101011011**

b	**a**

- Read each bit one at a time
- If it is 0 go left
- If it is 1 go right
- If you reach a leaf, output the character there and go back to the tree root



Decompressing Example

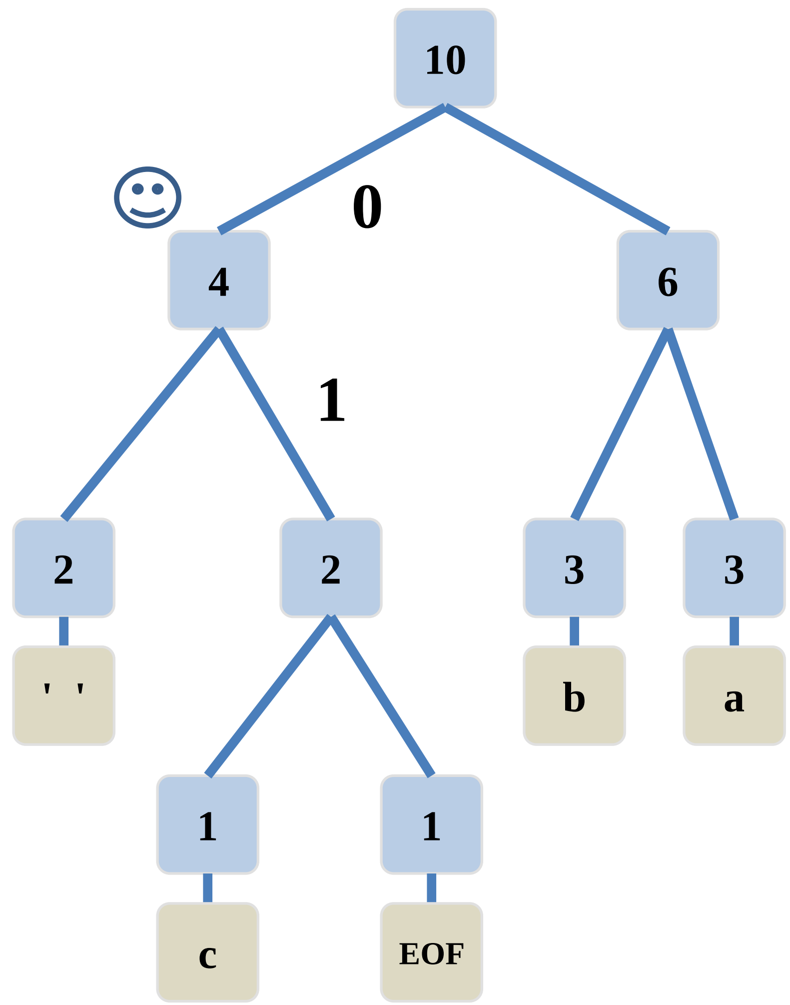
- Say the encrypted message was:
- **1011010001101011011**

b	**a**

- Read each bit one at a time

- If it is 0 go left
- If it is 1 go right

- If you reach a leaf, output the character there and go back to the tree root

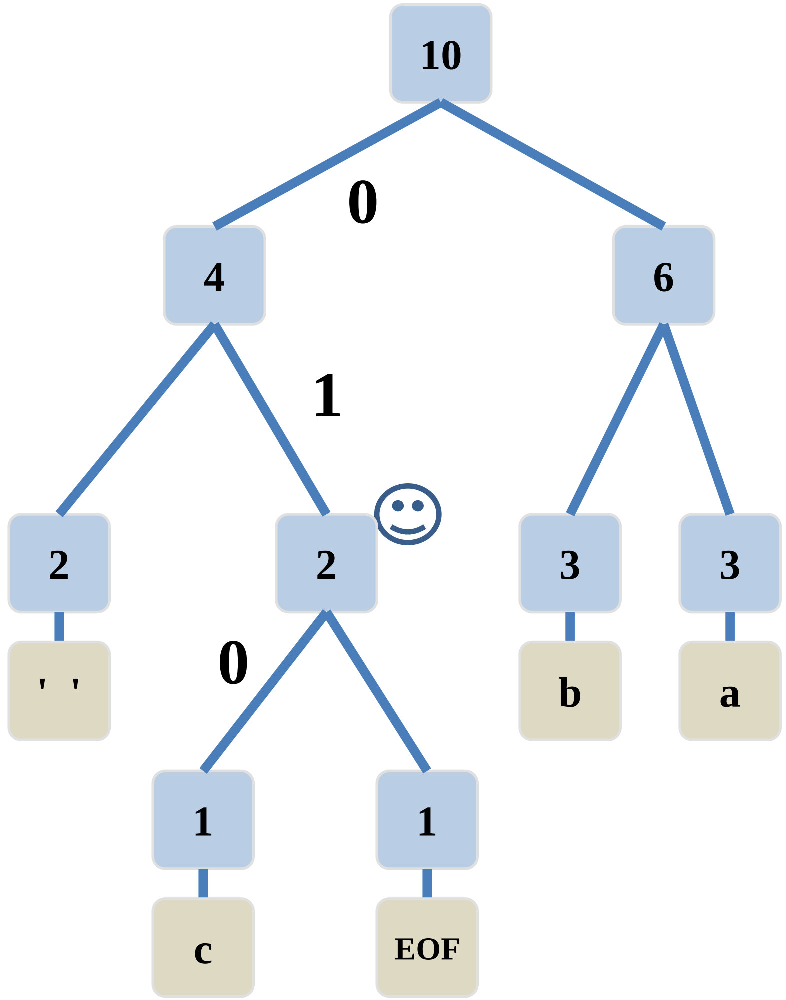


Decompressing Example

- Say the encrypted message was:
- **10110100011010011011**

b	a	c

- Read each bit one at a time
- If it is 0 go left
- If it is 1 go right
- If you reach a leaf, output the character there and go back to the tree root

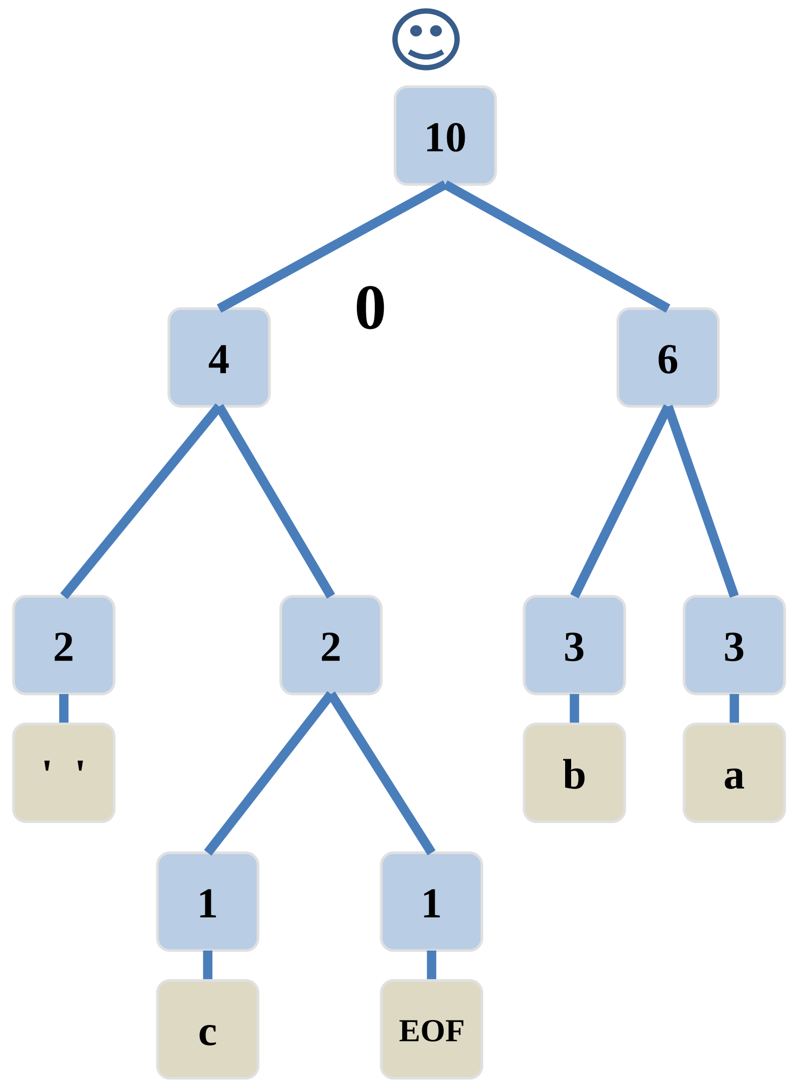


Decompressing Example

- Say the encrypted message was:
- **10110100001101011011**

b	a	c

- Read each bit one at a time
- If it is 0 go left
- If it is 1 go right
- If you reach a leaf, output the character there and go back to the tree root



Decompressing Example

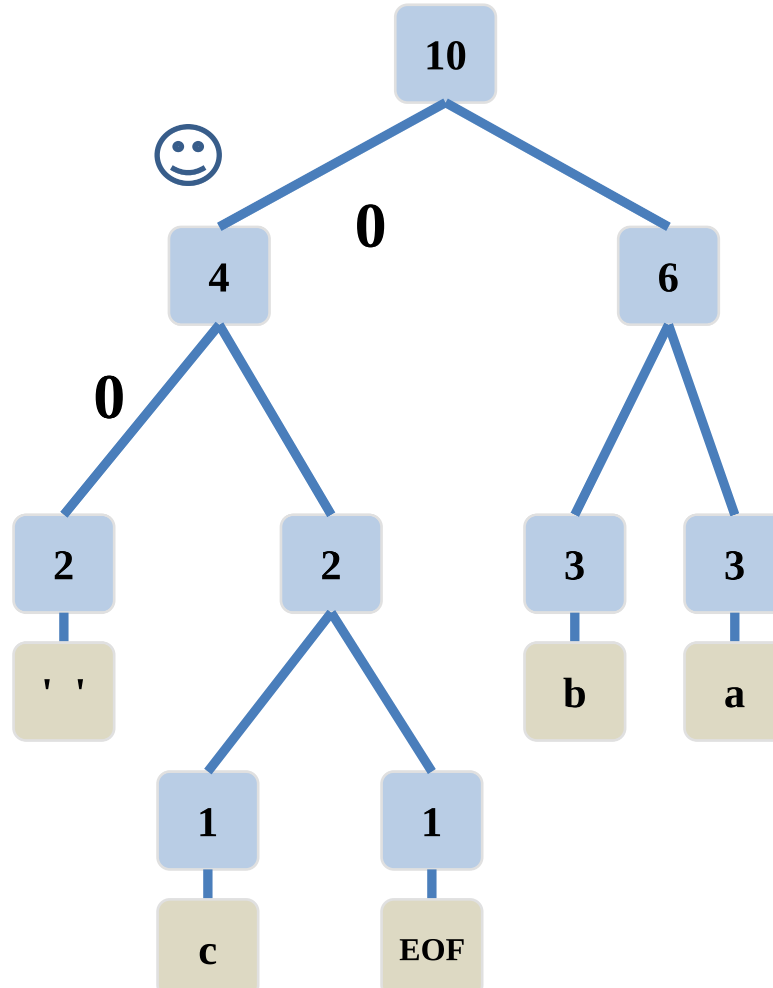
- Say the encrypted message was:
- **1011010001101011011**

b	a	c	

- Read each bit one at a time

- If it is 0 go left
- If it is 1 go right

- If you reach a leaf, output the character there and go back to the tree root

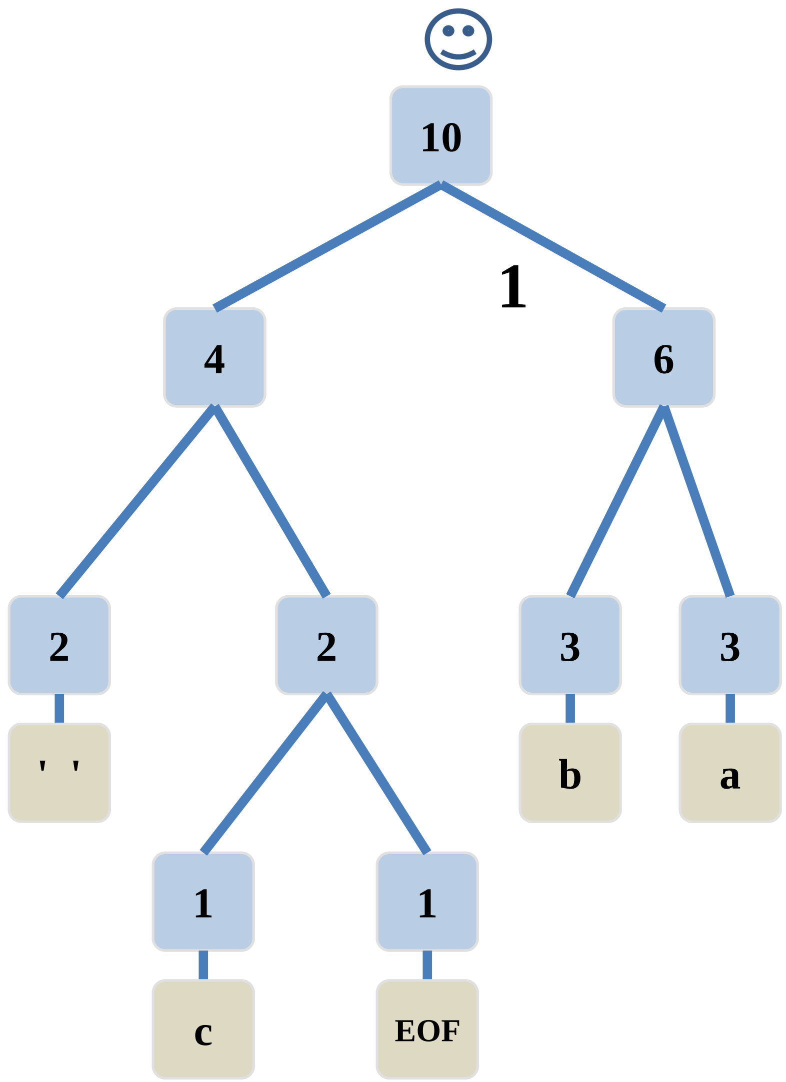


Decompressing Example

- Say the encrypted message was:
- **10111010001101011011**

b	a	c	

- Read each bit one at a time
- If it is 0 go left
- If it is 1 go right
- If you reach a leaf, output the character there and go back to the tree root

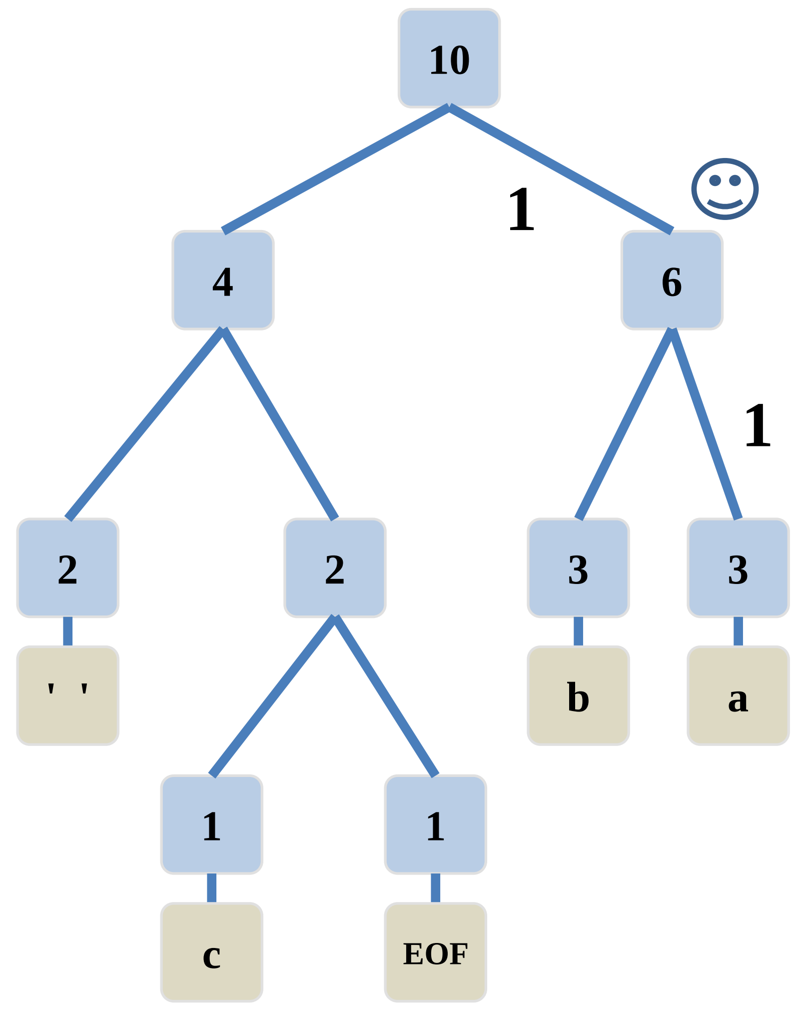


Decompressing Example

- Say the encrypted message was:
- **1011010000110101011011**

b	**a**	**c**		**a**

- Read each bit one at a time
- If it is 0 go left
- If it is 1 go right
- If you reach a leaf, output the character there and go back to the tree root

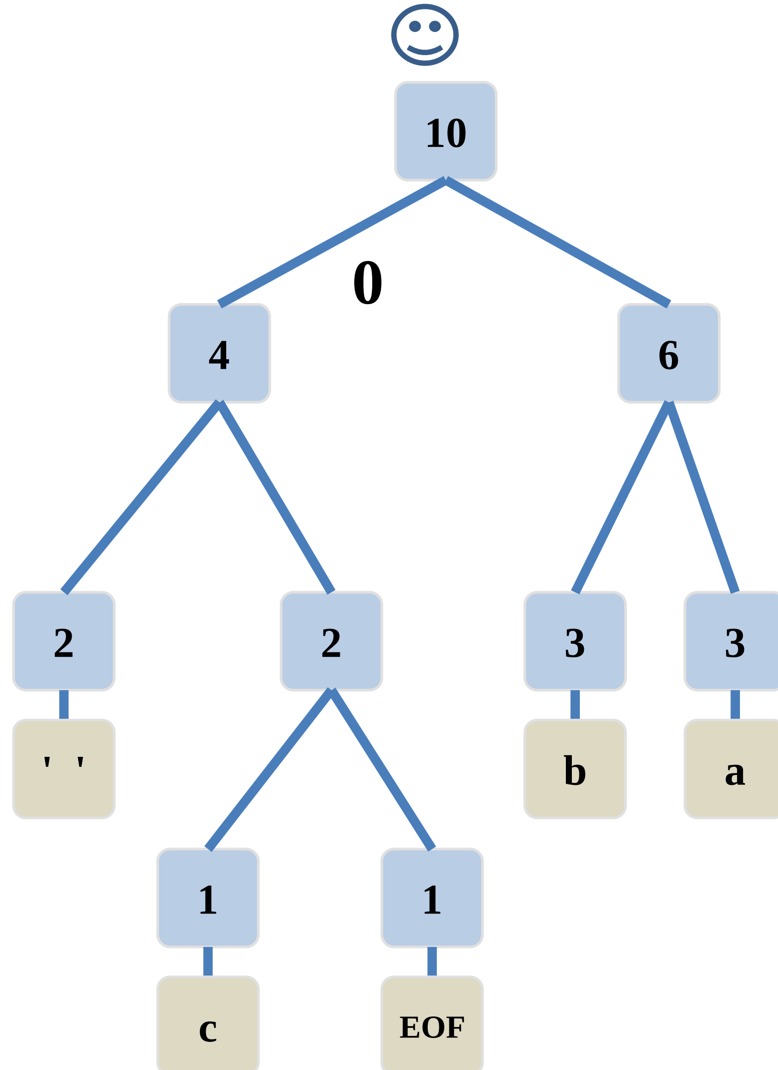


Decompressing Example

- Say the encrypted message was:
- **1011010001101011011**

b	**a**	**c**		**a**

- Read each bit one at a time
- If it is 0 go left
- If it is 1 go right
- If you reach a leaf, output the character there and go back to the tree root

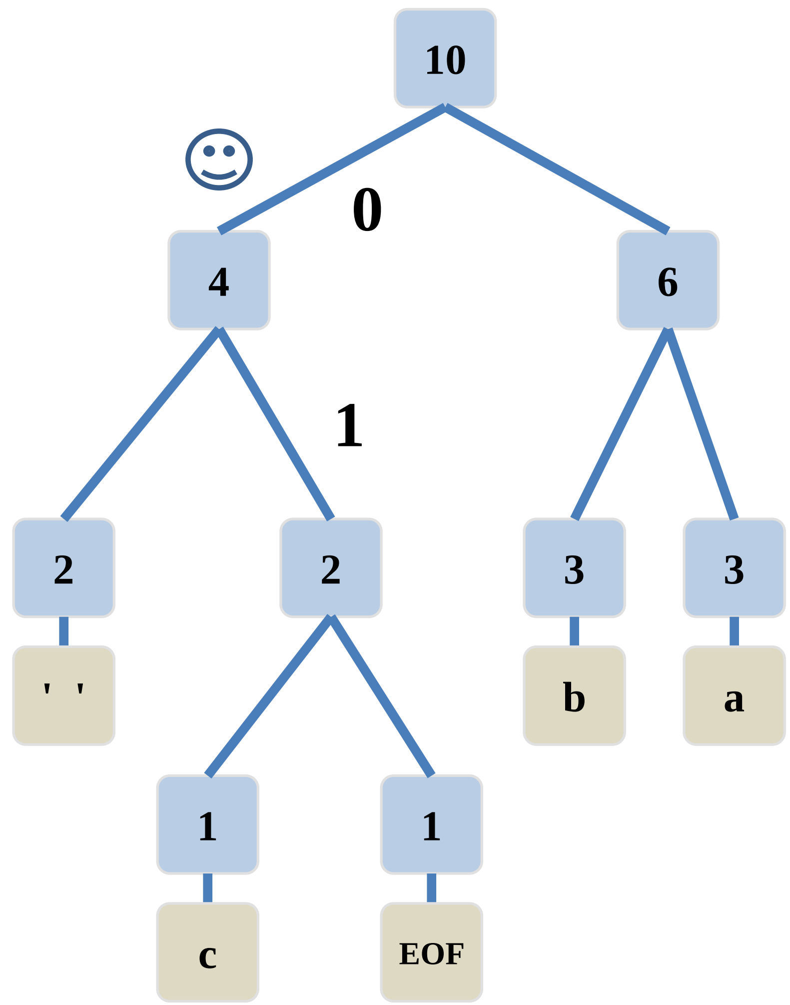


Decompressing Example

- Say the encrypted message was:
- **1011010001101011011**

b	a	c		a

- Read each bit one at a time
- If it is 0 go left
- If it is 1 go right
- If you reach a leaf, output the character there and go back to the tree root

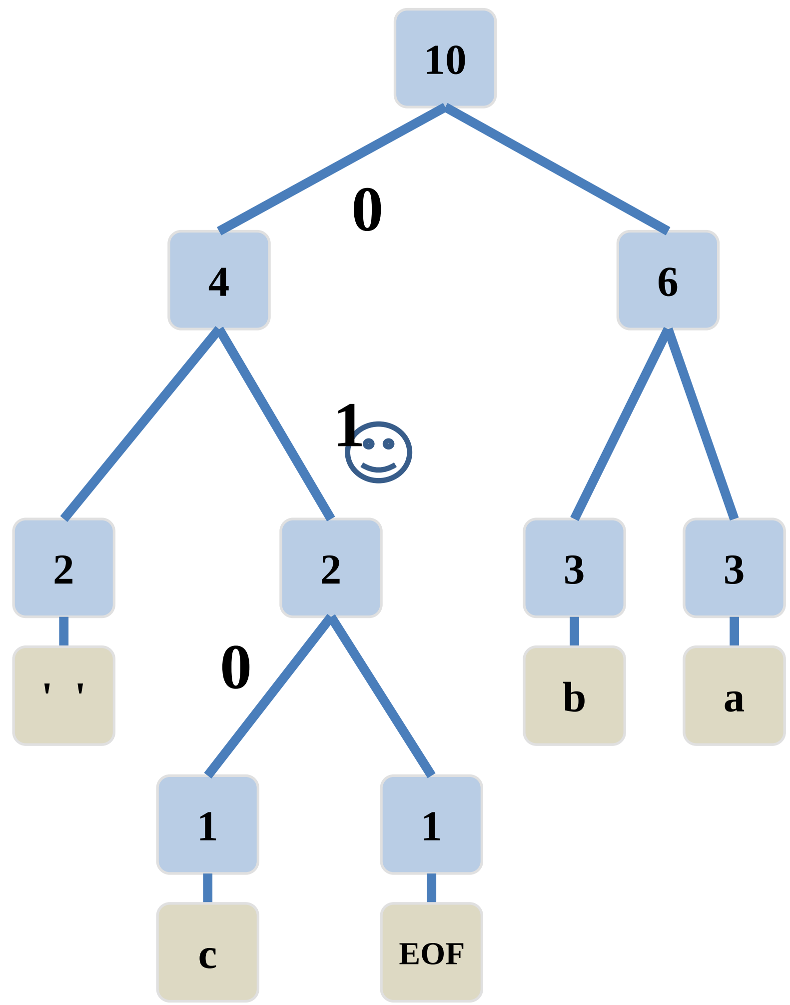


Decompressing Example

- Say the encrypted message was:
- **1011010001101011011**

b	**a**	**c**		**a**	**c**

- Read each bit one at a time
- If it is 0 go left
- If it is 1 go right
- If you reach a leaf, output the character there and go back to the tree root

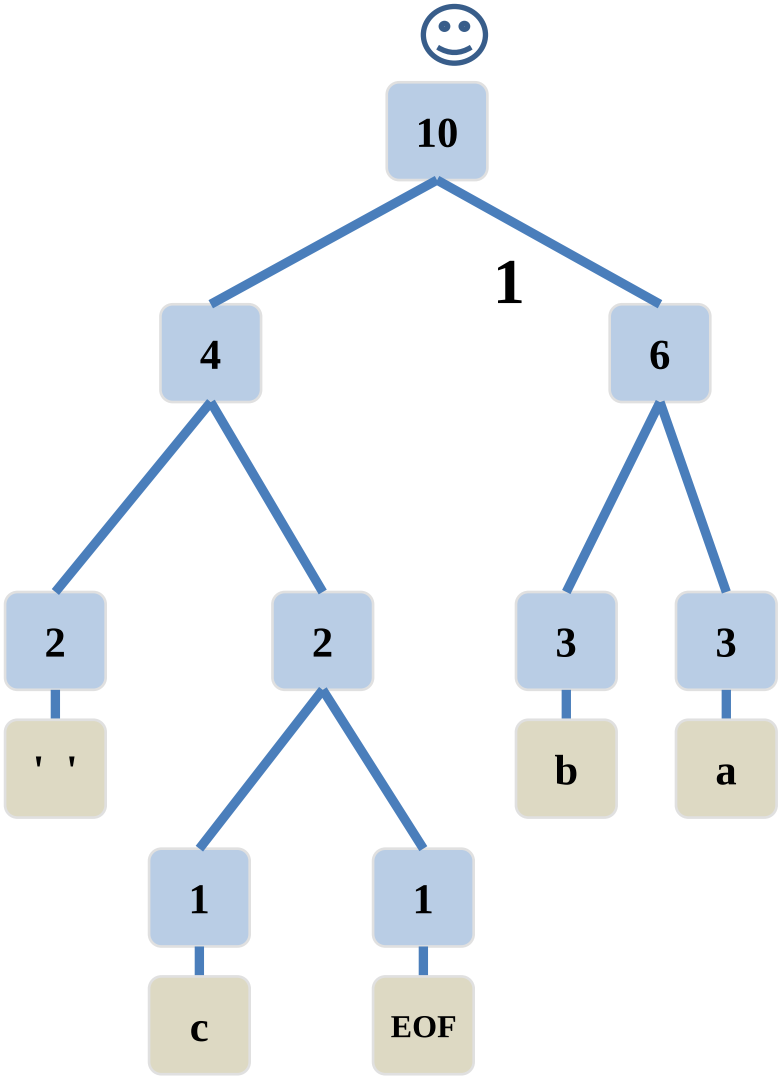


Decompressing Example

- Say the encrypted message was:
- **1011010001101011011**

b	a	c		a	c

- Read each bit one at a time
- If it is 0 go left
- If it is 1 go right
- If you reach a leaf, output the character there and go back to the tree root



Decompressing Example

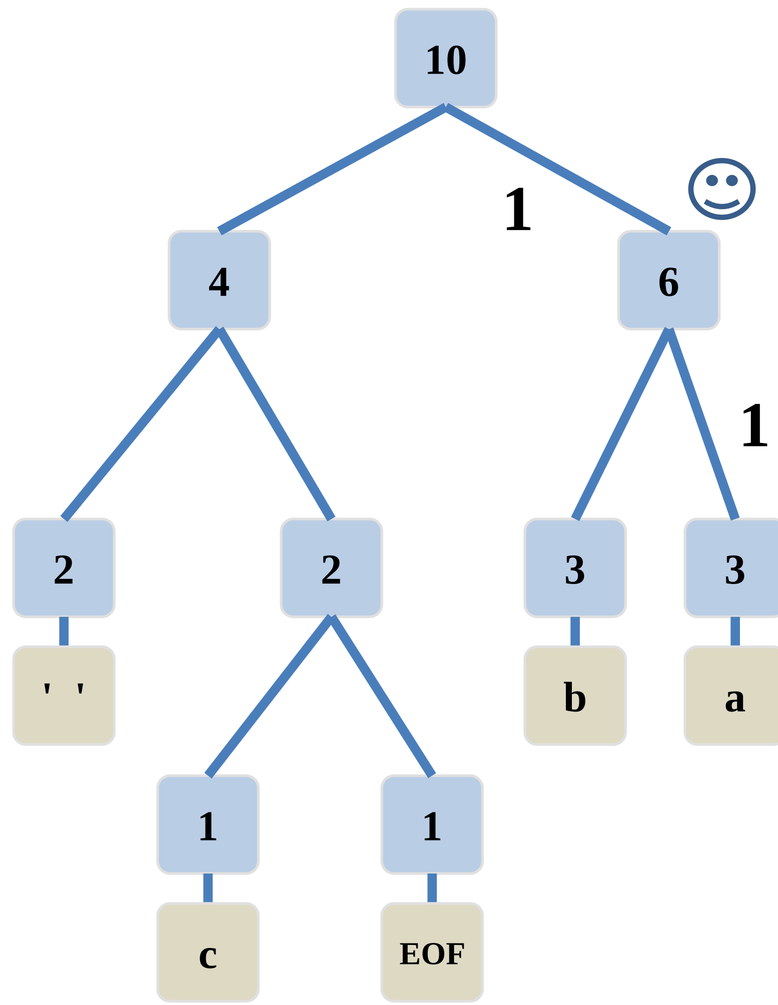
- Say the encrypted message was:
- **1011010001101011011**

b	**a**	**c**		**a**	**c**	**a**

- Read each bit one at a time

- If it is 0 go left
- If it is 1 go right

- If you reach a leaf, output the character there and go back to the tree root

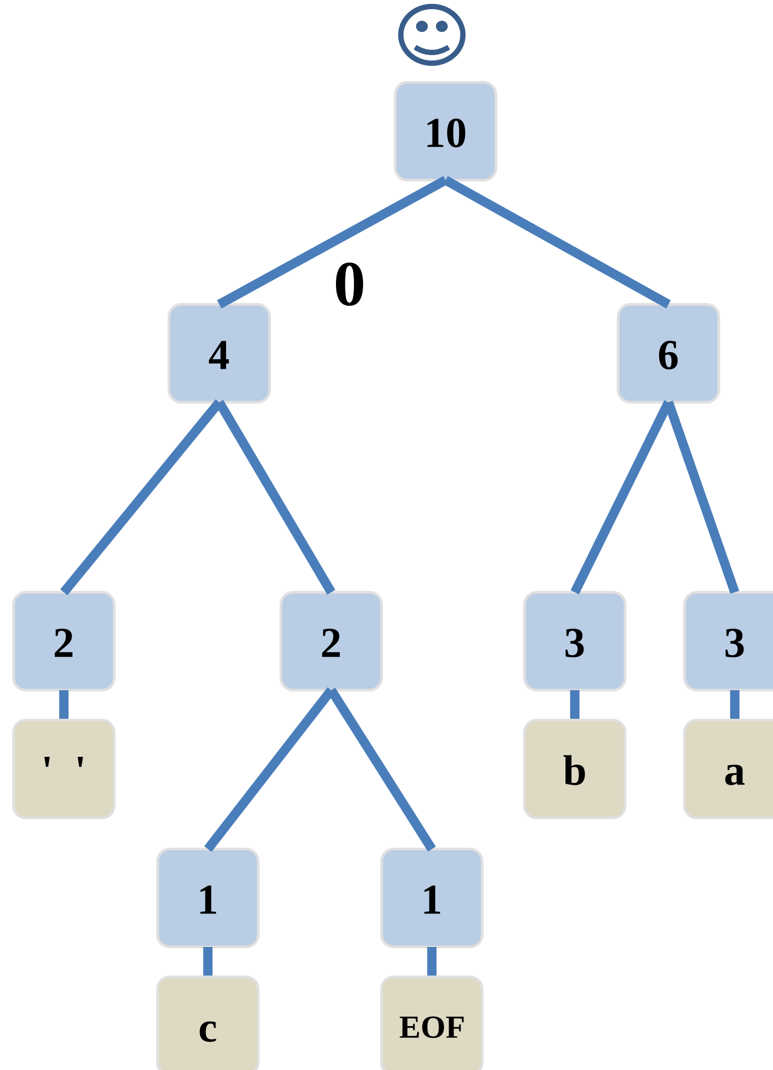


Decompressing Example

- Say the encrypted message was:
- **1011010001101011011**

b	**a**	**c**		**a**	**c**	**a**

- Read each bit one at a time
- If it is 0 go left
- If it is 1 go right
- If you reach a leaf, output the character there and go back to the tree root

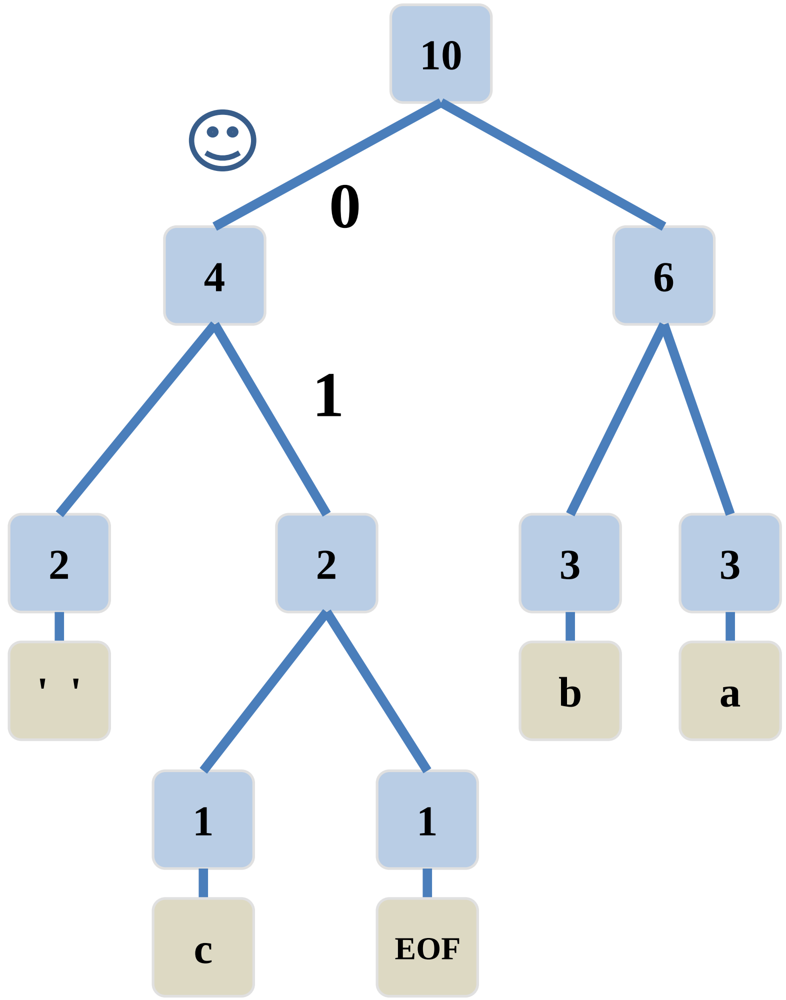


Decompressing Example

- Say the encrypted message was:
- 101101000110101011011

b	a	c		a	c	a

- Read each bit one at a time
- If it is 0 go left
- If it is 1 go right
- If you reach a leaf, output the character there and go back to the tree root



Decompressing Example

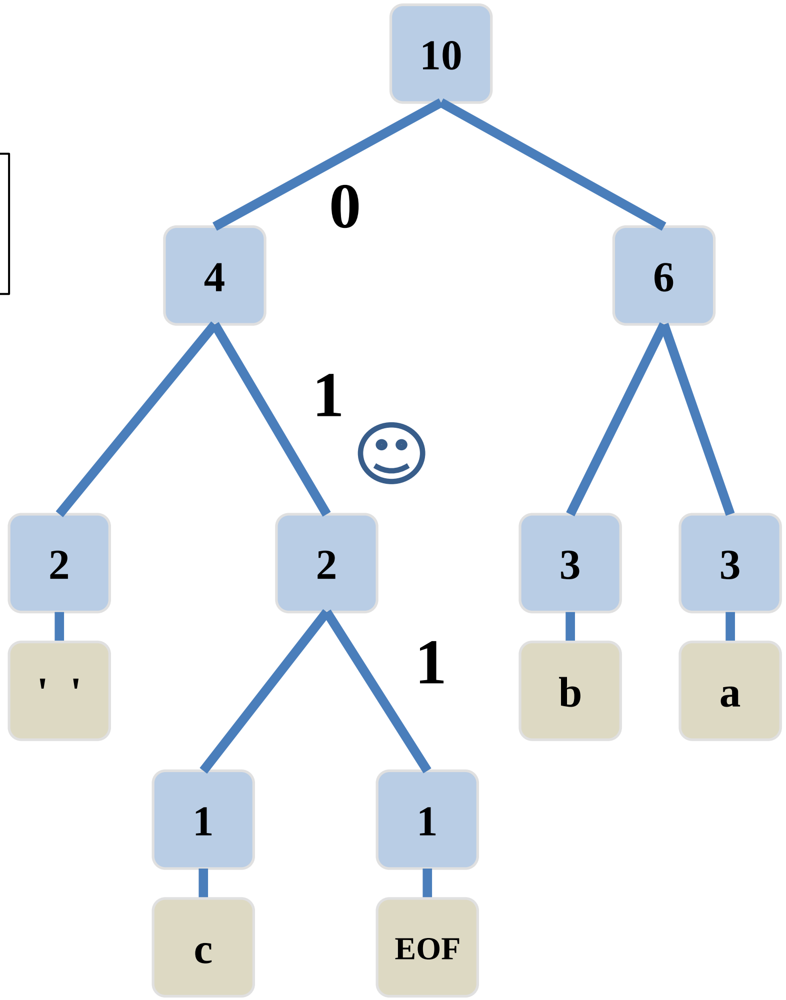
- Say the encrypted message was:
- **1011010001101011011**

b	a	c		a	c	a	EOF

- Read each bit one at a time

- If it is 0 go left
- If it is 1 go right

- If you reach a leaf, output the character there and go back to the tree root



Decompressing Example

- Say the encrypted message was:
- **1011010001101011011**

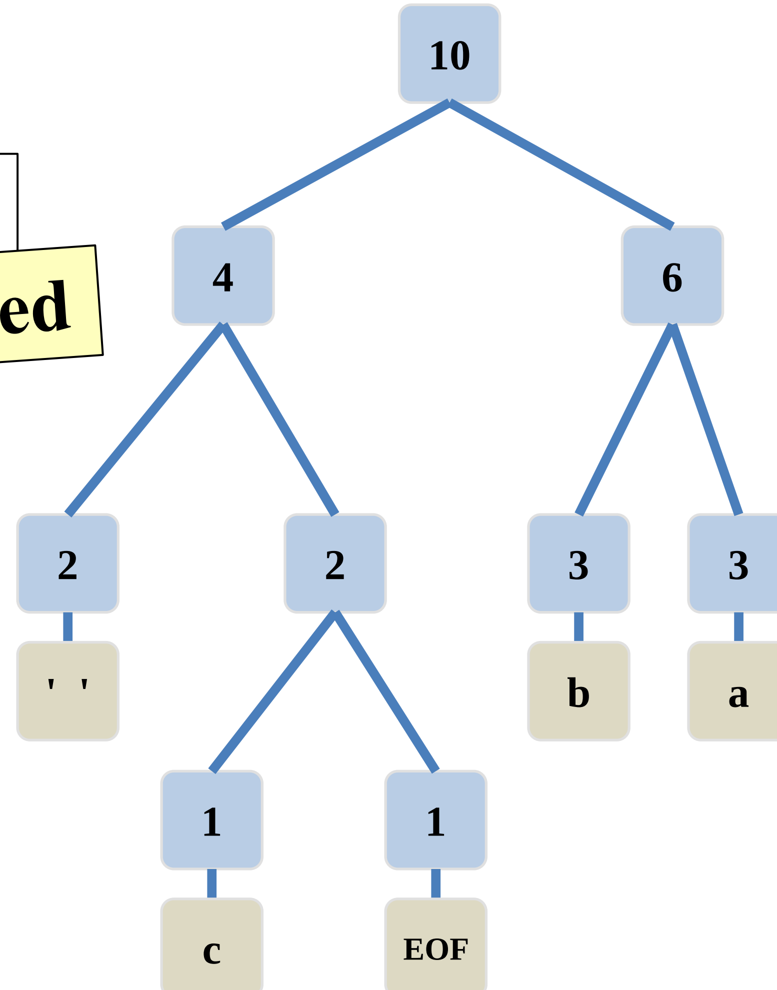
b	a	c		a	c	a	EOF

And the file is now decoded

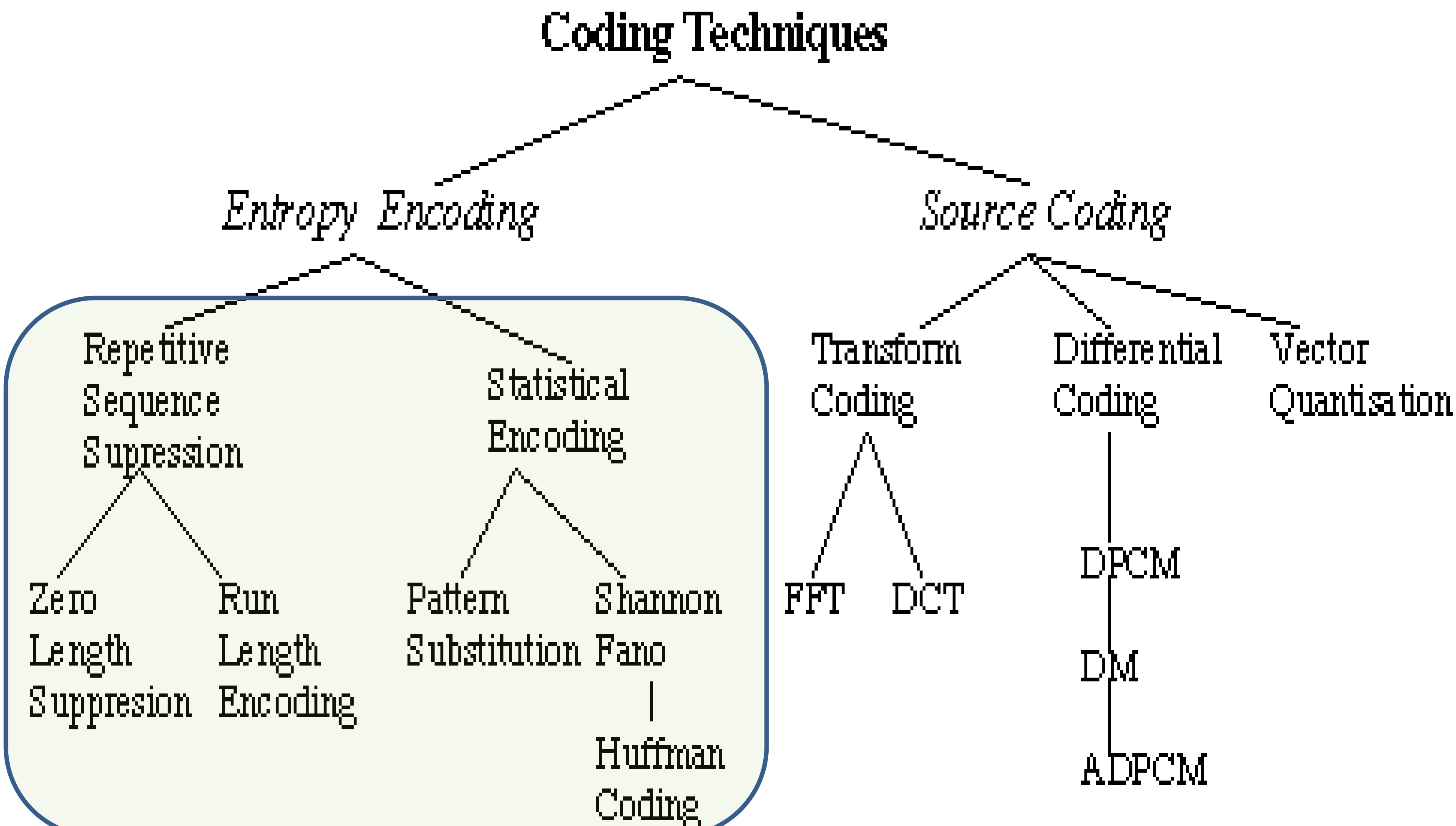
- Read each bit one at a time

- If it is 0 go left
- If it is 1 go right

- If you reach a leaf, output the character there and go back to the tree root



Compression Methods



up next: ONE MORE EXAMPLE

Lempel-Ziv-Welch (LZW) Compression

- Lossless
- Has a table
- Does not store the table

LZW Compression

- Discovers and remembers **patterns** of colors
- Stores the patterns in a table
  - BUT **only table indices are stored** in the file

- LZW table entries can grow arbitrarily long,
  - So one table index can stand for a long string of data in the file
  - BUT again the table itself never needs to be stored in the file

LZW Encoder: Pseudocode

```
initialize TABLE[0 to 255] = code for individual bytes
STRING = get input symbol
while there are still input symbols:
   SYMBOL = get input symbol
   if STRING + SYMBOL is in TABLE:
       STRING = STRING + SYMBOL
   else:
       output the code for STRING
       add STRING + SYMBOL to TABLE
       STRING = SYMBOL
output the code for STRING
```

RGB = 3 bytes
but idea stays same

From Chapter 3 of
MIT 6.02 DRAFT Lecture Notes, Feb 13, 2012

LZW Decoder: Pseudocode

```
initialize TABLE[0 to 255] = code for individual bytes
CODE = read next code from encoder
STRING = TABLE[CODE]
output STRING

while there are still codes to receive:
    CODE = read next code from encoder
    if TABLE[CODE] is not defined: // needed because sometimes the
        ENTRY = STRING + STRING[0] // decoder may not yet have entry
    else:
        ENTRY = TABLE[CODE]
    output ENTRY
    add STRING+ENTRY[0] to TABLE
    STRING = ENTRY
```

RGB = 3 bytes
but idea stays same

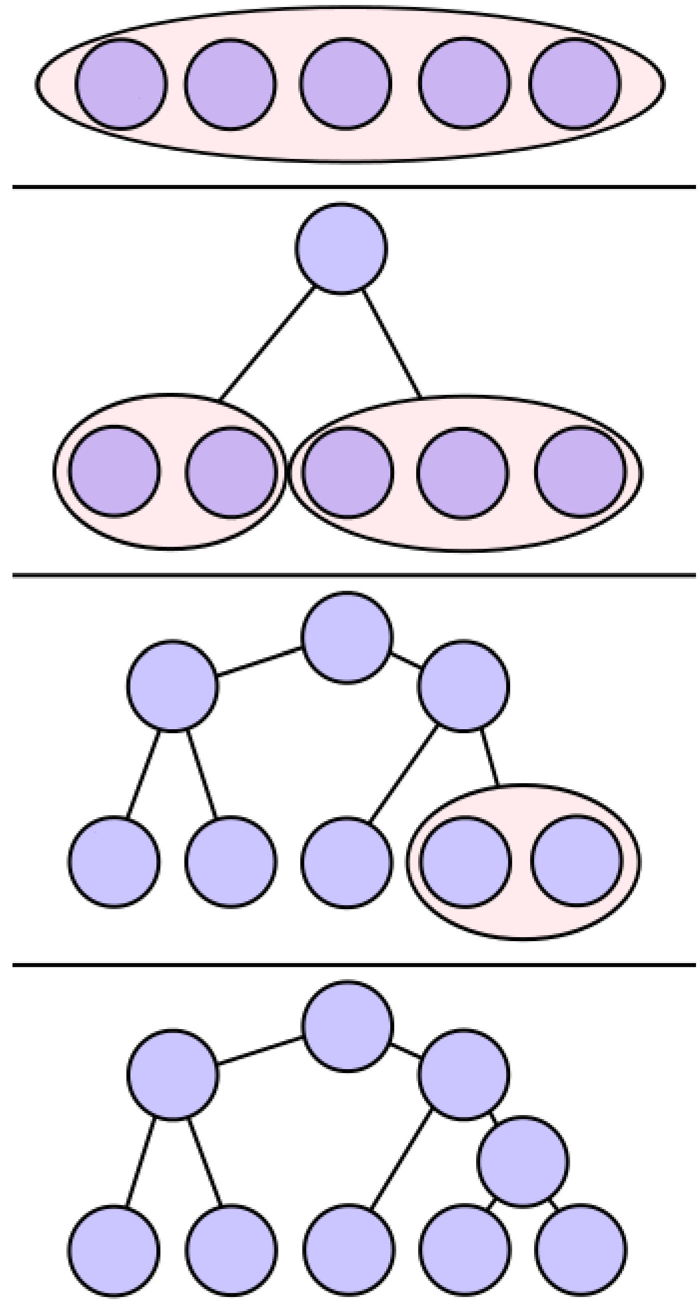
From Chapter 3 of
MIT 6.02 DRAFT Lecture Notes, Feb 13, 2012

Questions?

- Beyond D2L
  - Examples and information can be found online at:
    - *http://docdingle.com/teaching/cs.html*

    - *Continue to more stuff as needed*

Extra Reference Stuff Follows



Credits

- Much of the content derived/based on slides for use with the book:
  - *Digital Image Processing,* Gonzalez and Woods

- Some layout and presentation style derived/based on presentations by
  - Donald House, Texas A&M University, 1999
  - Bernd Girod, Stanford University, 2007
  - Shreekanth Mandayam, Rowan University, 2009
  - Igor Aizenberg, TAMUT, 2013
  - Xin Li, WVU, 2014
  - George Wolberg, City College of New York, 2015
  - Yao Wang and Zhu Liu, NYU-Poly, 2015
  - Sinisa Todorovic, Oregon State, 2015