BINARY CHARACTER CODES

for

LOSSLESS COMPRESSION

- Introduction to data compression - Khalid Sayood

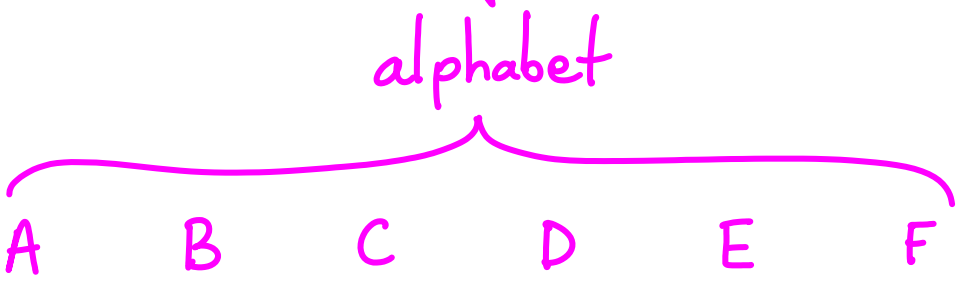
& CLRS

BINARY CHARACTER CODES

How to encode ...ABACBBFAEFDACB... in binary, knowing full string

BINARY CHARACTER CODES

How to encode ...ABACBBFAEFDACB... in binary, knowing full string

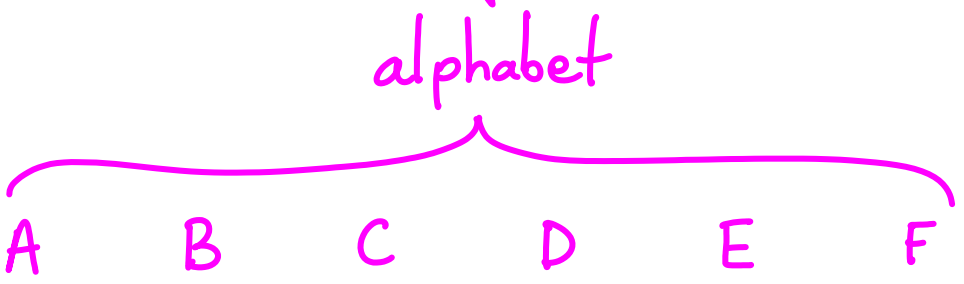


000 001 010 011 100 101 → 3 bits/char.

impossible w/ 2 bits

BINARY CHARACTER CODES

How to encode ...ABACBBFAEFDACB... in binary, knowing full string



	000	001	010	011	100	101	→
Why not	0	1	01	011	100	101	?

3 bits/char.

impossible w/ 2 bits

A	B	C	D	
0	0	1	10	terrible

A	B	C	D	
0	0	1	10	terrible
0	1	00	11	ambiguous

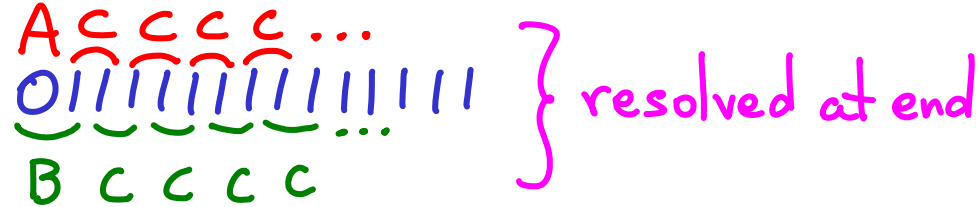
A	B	C	D	
0	0	1	10	terrible
0	1	00	11	ambiguous
0	10	110	111	uniquely decodable

A	B	C	D	
0	0	1	10	terrible
0	1	00	11	ambiguous
0	10	110	111	uniquely decodable
0	01	011	0111	?

A	B	C	D	
0	0	1	10	terrible
0	1	00	11	ambiguous
0	10	110	111	uniquely decodable
0	01	011	0111	uniquely decodable but not instantaneous

01 = "B" or continue?

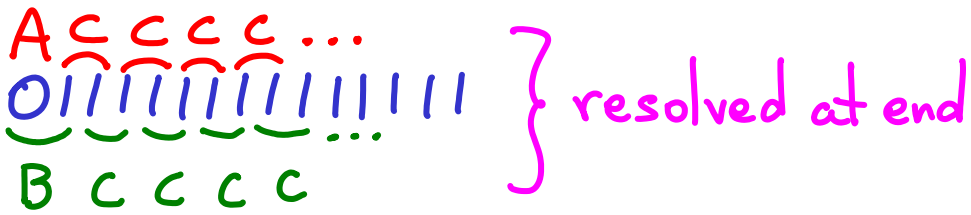
A	B	C	D	
0	0	1	10	terrible
0	1	00	11	ambiguous
0	10	110	111	uniquely decodable
0	01	011	0111	uniquely decodable but not instantaneous

0	01	11	—	

How do we know if a code is uniquely decodable?

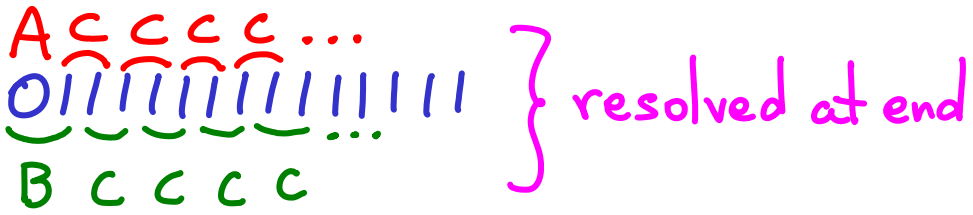
0 01 11

A B C



How do we know if a code is uniquely decodable?

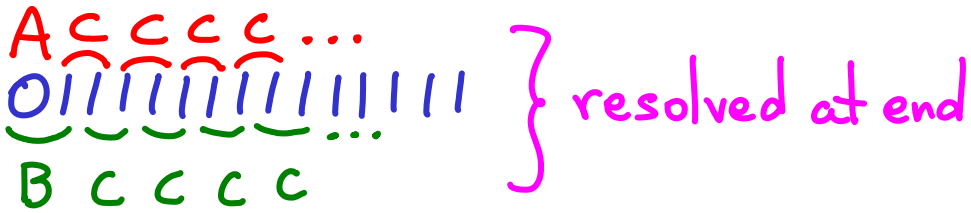
0	01	11
A	B	C
0	01	10

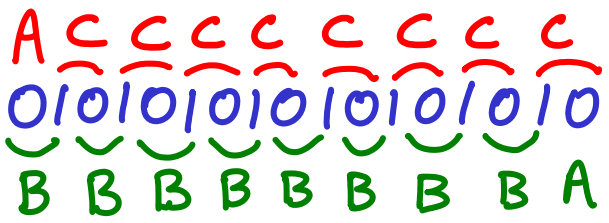


?

How do we know if a code is uniquely decodable?

0	01	11
A	B	C
0	01	10

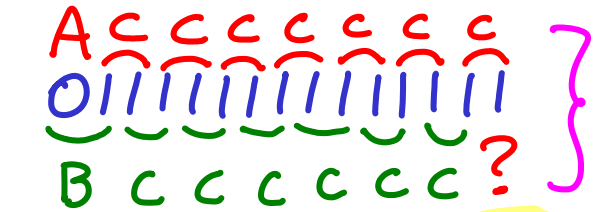




Problem

How do we know if a code is uniquely decodable?

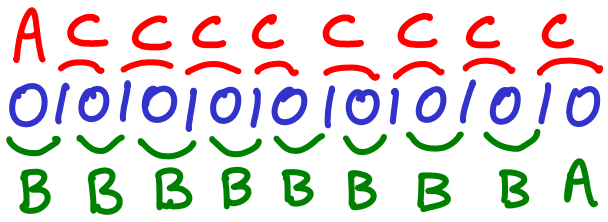
0	01	11
A	B	C



If we encoded ACC··· then decoding BCC··· fails ☺

(& vice versa)

0	01	10



Problem

Whatever we encoded, both decodings work ☹

How do we know if a code is uniquely decodable?

$$A = c_1 c_2 c_3 \cdots c_k$$

$$B = c_1 c_2 c_3 \cdots c_k d_1 d_2 \cdots d_j$$

} A is a prefix of B

How do we know if a code is uniquely decodable?

$A = c_1 c_2 c_3 \cdots c_k$

$B = c_1 c_2 c_3 \cdots c_k \underbrace{d_1 d_2 \cdots d_j}_{\text{"dangling suffix"}}$

} A is a prefix of B

How do we know if a code is uniquely decodable?

Not good:

$A = c_1 c_2 c_3 \cdots c_k$

$B = c_1 c_2 c_3 \cdots c_k \underbrace{d_1 d_2 \cdots d_j}$ "dangling suffix"

A is a prefix of B... and ... is another codeword

A	B	C
0	01	1

How do we know if a code is uniquely decodable?

Not good:

(but also not a sufficient test)

$A = c_1 c_2 c_3 \cdots c_k$

$B = c_1 c_2 c_3 \cdots c_k \underbrace{d_1 d_2 \cdots d_j}_{\text{"dangling suffix"}}$

} A is a prefix of B... and "dangling suffix"... is another codeword

0	01	11	☺
A	B	C	
0	01	10	☹

ok

also ok?!

How do we know if a code is uniquely decodable?

Not good:

(but also not a sufficient test)

$A = c_1 c_2 c_3 \cdots c_k$

$B = c_1 c_2 c_3 \cdots c_k \underbrace{d_1 d_2 \cdots d_j}_{\text{"dangling suffix"}}$

} A is a prefix of B... and ... is another codeword

0	01	11	☺
A	B	C	
0	01	10	☹

Extended test: if ∃ dangling suffix

---

0 01 11

0 01 10

How do we know if a code is uniquely decodable?

Not good:

(but also not a sufficient test)

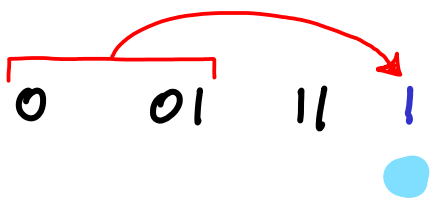
$A = c_1 c_2 c_3 \cdots c_k$

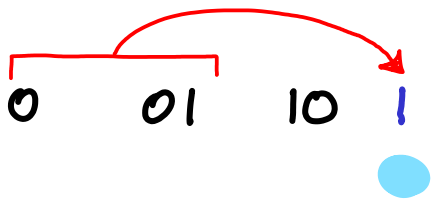
$B = c_1 c_2 c_3 \cdots c_k \underbrace{d_1 d_2 \cdots d_j}$ "dangling suffix"

} A is a prefix of B... and ... is another codeword

0	01	11	☺
A	B	C	
0	01	10	☹

Extended test: if ∃ dangling suffix add it to a list (set)





How do we know if a code is uniquely decodable?

Not good:

(but also not a sufficient test)

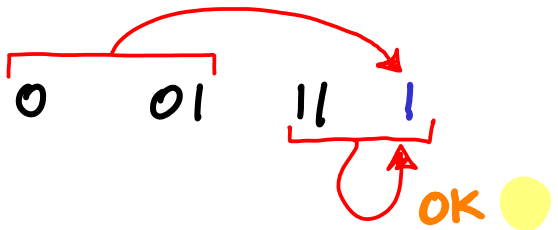
$A = c_1 c_2 c_3 \cdots c_k$

$B = c_1 c_2 c_3 \cdots c_k \underbrace{d_1 d_2 \cdots d_j}$ "dangling suffix"

} A is a prefix of B... and ... is another codeword

0	01	11	☺
A	B	C	
0	01	10	☹

Extended test: if ∃ dangling suffix add it to a list (set) & repeat original test on combined set until no new listwords get generated.



0 01 10 1

How do we know if a code is uniquely decodable?

Not good:

(but also not a sufficient test)

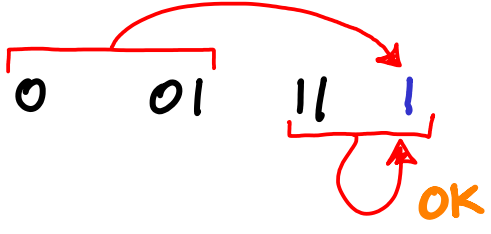
$A = c_1 c_2 c_3 \cdots c_K$

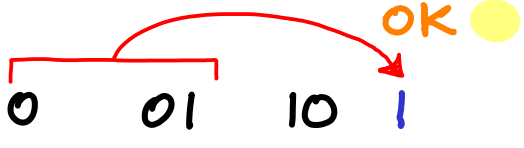
$B = c_1 c_2 c_3 \cdots c_K \underbrace{d_1 d_2 \cdots d_j}$

"dangling suffix"

A is a prefix of B... and ... is another codeword

0	01	11	☺
A	B	C	
0	01	10	☹





Extended test: if ∃ dangling suffix add it to a list (set) & repeat original test on combined set until no new listwords get generated.

How do we know if a code is uniquely decodable?

Not good:
(but also not a sufficient test)

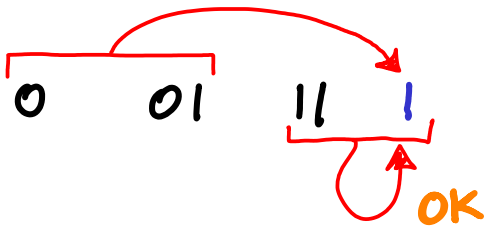
$A = c_1 c_2 c_3 \cdots c_k$

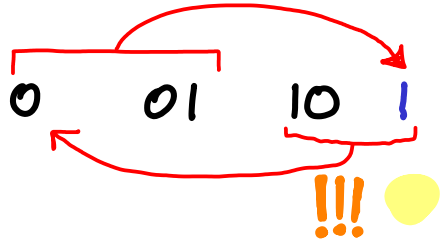
$B = c_1 c_2 c_3 \cdots c_k \underbrace{d_1 d_2 \cdots d_j}$ "dangling suffix"

} A is a prefix of B... and ... is another codeword

0	01	11	☺
A	B	C	
0	01	10	☹

Extended test: if ∃ dangling suffix add it to a list (set) & repeat original test on combined set until no new listwords get generated.





How do we know if a code is uniquely decodable?

Not good:
(but also not a sufficient test)

$A = c_1 c_2 c_3 \cdots c_k$

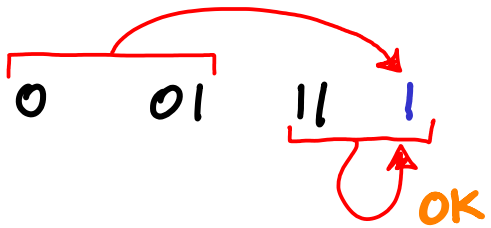
$B = c_1 c_2 c_3 \cdots c_k \underbrace{d_1 d_2 \cdots d_j}$ "dangling suffix"

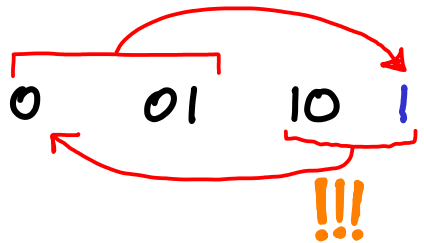
} A is a prefix of B... and ... is another codeword

0	01	11	STILL OK ☺
A	B	C	
0	01	10	NOT OK ☹

Extended test: PROOF OMITTED

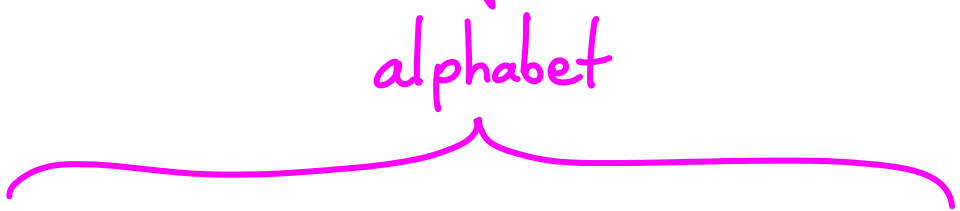
if ∃ dangling suffix
add it to a list (set)
& repeat original test
on combined set
until no new listwords
get generated.





BINARY CHARACTER CODES

How to encode ...ABACBBFAEFDACB... in binary, knowing full string



A	B	C	D	E	F	: % frequencies
45	13	12	16	9	5	
000	001	010	011	100	101	→ 3 bits/char.

impossible w/ 2 bits

BINARY CHARACTER CODES

How to encode ...ABACBBFAEFDACB... in binary, knowing full string

alphabet

A	B	C	D	E	F		
45	13	12	16	9	5	:	% frequencies
000	001	010	011	100	101	→	3 bits/char.

impossible w/ 2 bits

0	101	100	111	1101	1100	:	Variable length code

BINARY CHARACTER CODES

How to encode ...ABACBBFAEFDACB... in binary, knowing full string

alphabet

A	B	C	D	E	F	
45	13	12	16	9	5	: % frequencies
000	001	010	011	100	101	→ average 3 bits/char.
						impossible w/ 2 bits
0	101	100	111	1101	1100	: Variable length code

Σ(freq.bits) .45·1 + (.13+.12+.16)·3 + (.09+.05)·4

BINARY CHARACTER CODES

How to encode ...ABACBBFAEFDACB... in binary, knowing full string

alphabet

A	B	C	D	E	F	
45	13	12	16	9	5	: % frequencies
000	001	010	011	100	101	→ average 3 bits/char.

impossible w/ 2 bits

0	101	100	111	1101	1100	: Variable length code

Σ(freq.bits) $.45 \cdot 1 + (.13+.12+.16) \cdot 3 + (.09+.05) \cdot 4$ → average 2.24 bits/char.

BINARY CHARACTER CODES

How to encode ...ABACBBFAEFDACB... in binary, knowing full string

alphabet

	A	B	C	D	E	F	
New example →	20	13	17	16	19	15	: % frequencies
	000	001	010	011	100	101	→ average 3 bits/char.
							impossible w/ 2 bits
	10	000	010	011	11	001	: Variable length code
	.2·2	+(.13+.17+.16)·3			+.19·2	+.15·3	→ average 2.46 bits/char.

BINARY CHARACTER CODES

How to encode ...ABACBBFAEFDACB... in binary, knowing full string

alphabet

3rd example →

A	B	C	D	E	F	
95	1	1	1	1	1	: % frequencies
000	001	010	011	100	101	→ average 3 bits/char.

for fixed length: impossible w/ 2 bits

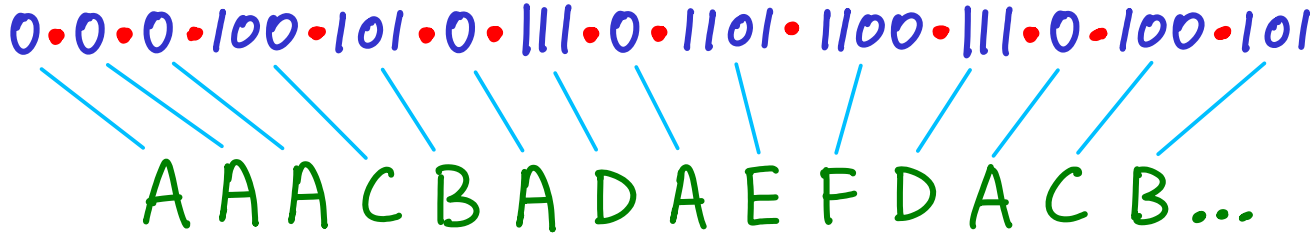
1	000	001	0100	0101	011	: Variable length code

$.95 \cdot 1 + (.01+.01) \cdot 3 + (.01+.01) \cdot 4 + .01 \cdot 3$ → average 1.12 bits/char.

How to encode A A A C B A D A E F D A C B ...

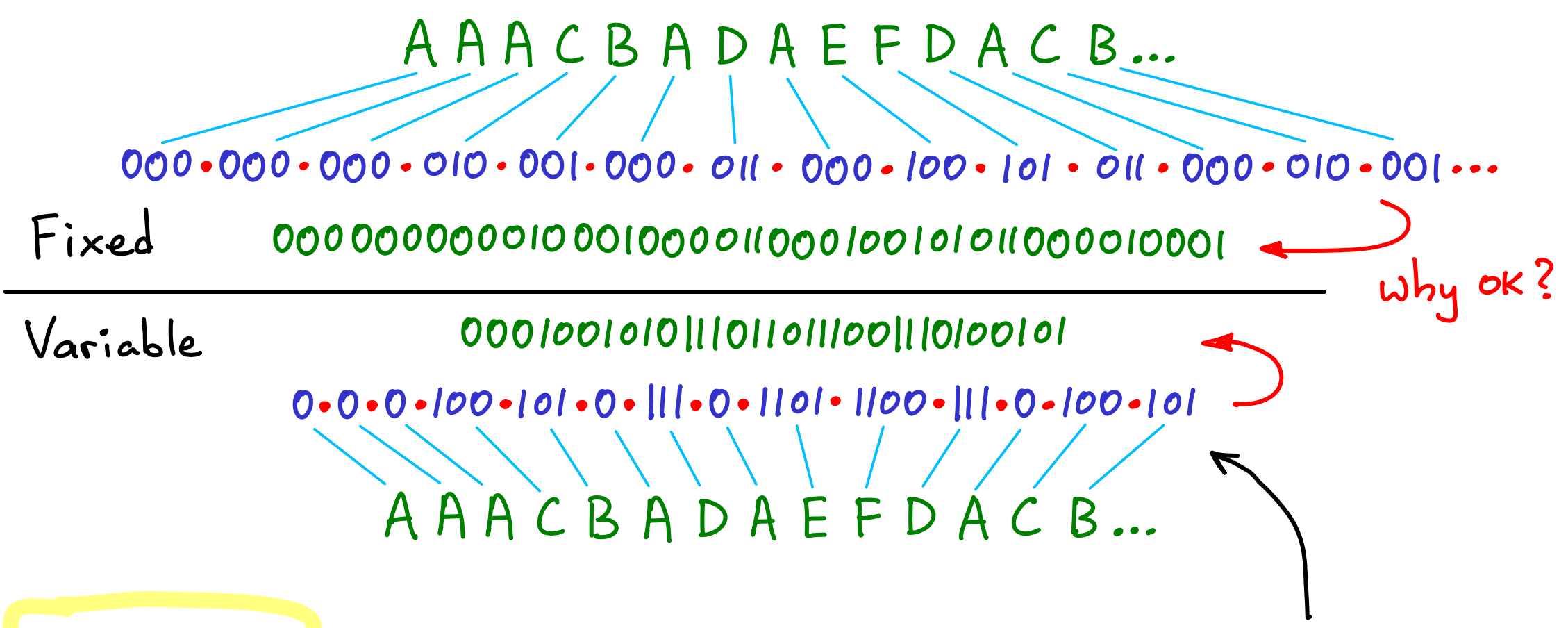
A	B	C	D	E	F	
45	13	12	16	9	5	: % frequencies

How to encode A A A C B A D A E F D A C B...



A	B	C	D	E	F	
45	13	12	16	9	5	: % frequencies
000	001	010	011	100	101	: fixed length code
0	101	100	111	1101	1100	: Variable length code


How to encode A A A C B A D A E F D A C B...
000 • 000 • 000 • 010 • 001 • 000 • 011 • 000 • 100 • 101 • 011 • 000 • 010 • 001...
Fixed
000000000010010000110001001010110000010001
why ok?
Variable
0001001010111011011100111010010 1
0 • 0 • 0 • 100 • 101 • 0 • 111 • 0 • 1101 • 1100 • 111 • 0 • 100 • 101
A A A C B A D A E F D A C B...
A B C D E F
45 13 12 16 9 5 : % frequencies
000 001 010 011 100 101 : fixed length code
0 101 100 111 1101 1100 : Variable length code


A A A C B A D A E F D A C B...
000·000·000·010·001·000·011·000·100·101·011·000·010·001...
Fixed
000000000010001000011000100101011000010001
why ok?
Variable
000100101011101101110011101000101
0·0·0·100·101·0·111·0·1101·1100·111·0·100·101
A A A C B A D A E F D A C B...
Prefix codes: no code is a prefix of another e.g., only A starts with 0
unlike, say, A=01, B=011, C=1 → 011 = B or AC ?

We still need to know when each code ends

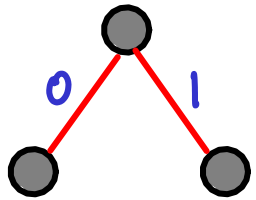
^word

How do we know there is no 0001 ?

0001001010111011011100111010010 1

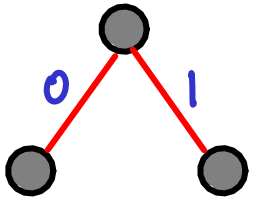
A	B	C	D	E	F	
45	13	12	16	9	5	: % frequencies
0	101	100	111	1101	1100	: Variable length (prefix) code

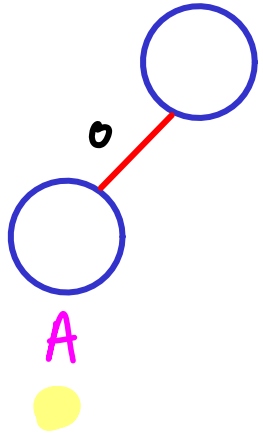
We still need to know when each code ends → use binary tree



A	B	C	D	E	F	
45	13	12	16	9	5	: % frequencies
0	101	100	111	1101	1100	: Variable length (prefix) code

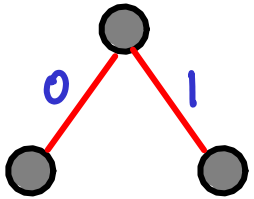
We still need to know when each code ends → use binary tree

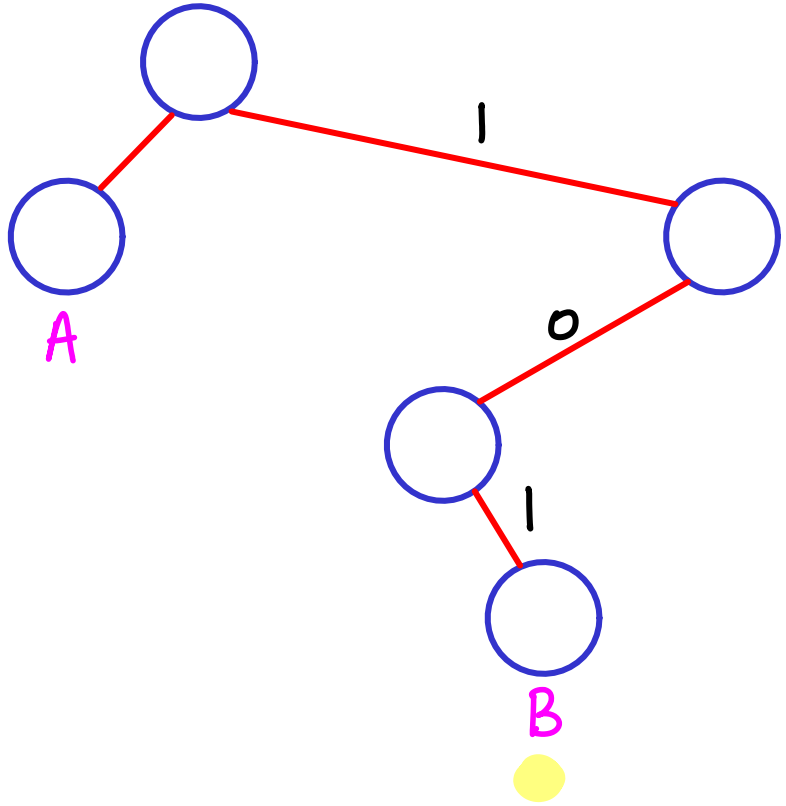




A	B	C	D	E	F	
45	13	12	16	9	5	: % frequencies
0	101	100	111	1101	1100	: Variable length (prefix) code

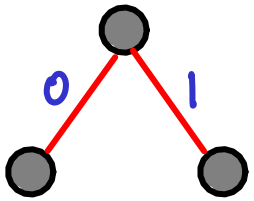
We still need to know when each code ends → use binary tree

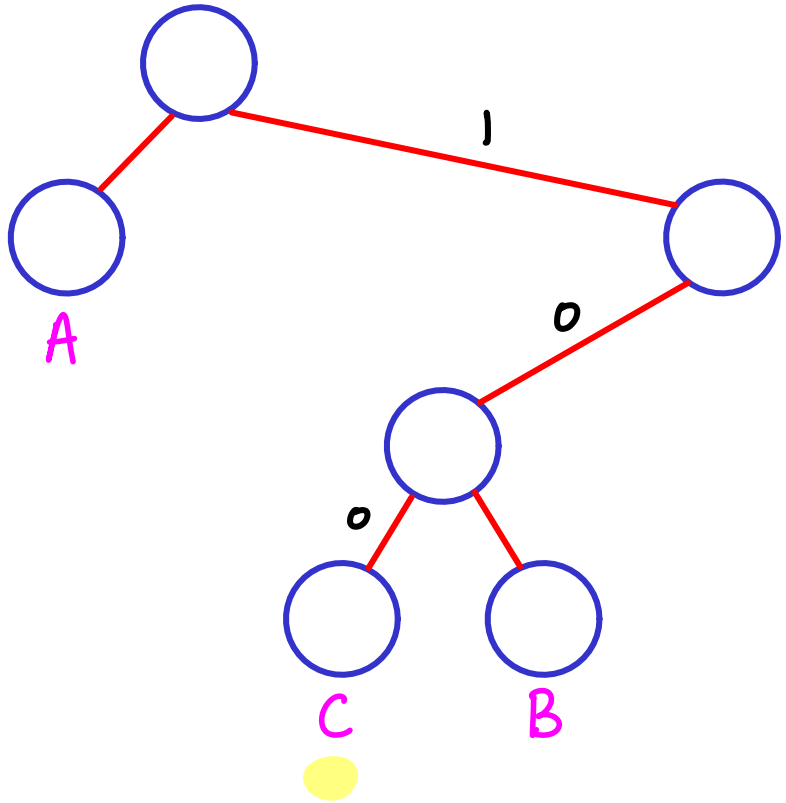




A	B	C	D	E	F	
45	13	12	16	9	5	: % frequencies
0	101	100	111	1101	1100	: Variable length (prefix) code

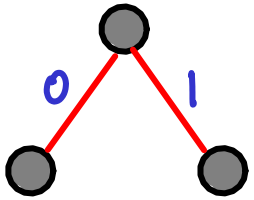
We still need to know when each code ends → use binary tree

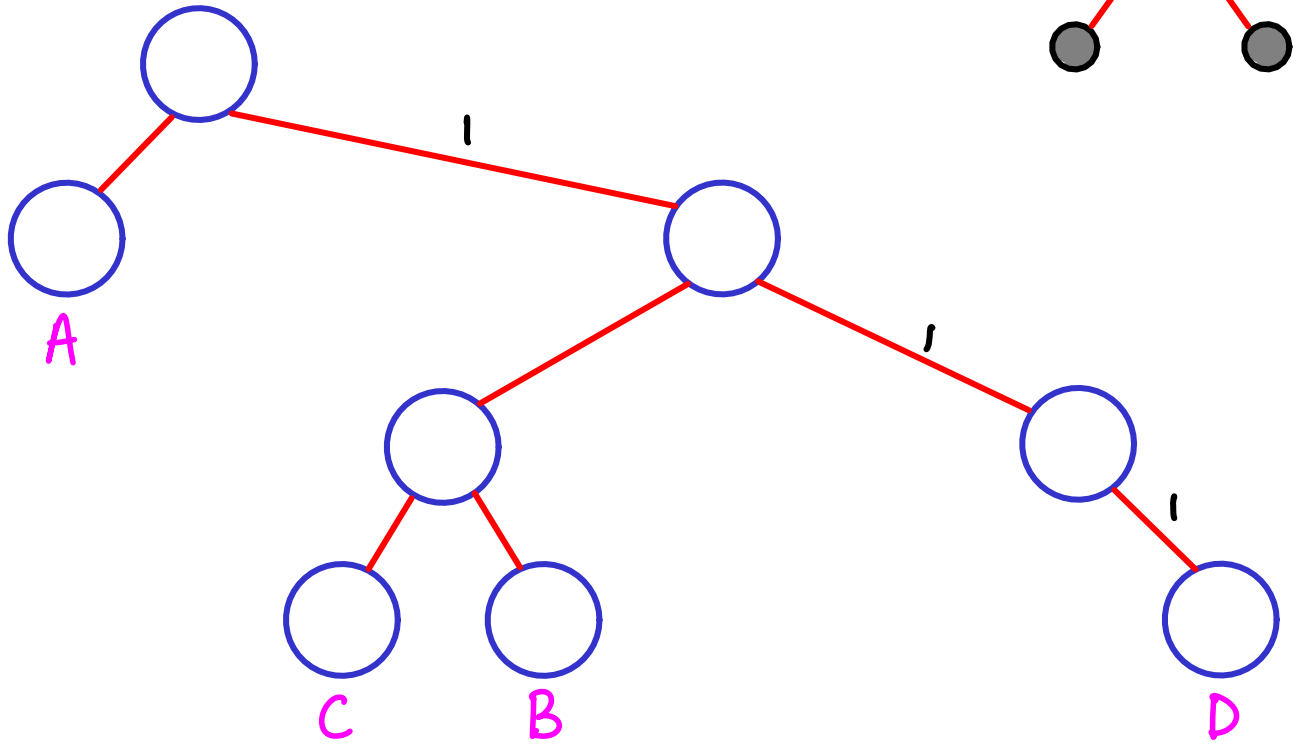




A	B	C	D	E	F	
45	13	12	16	9	5	: % frequencies
0	101	100	111	1101	1100	: Variable length (prefix) code

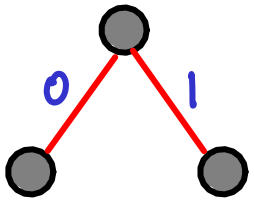
We still need to know when each code ends → use binary tree

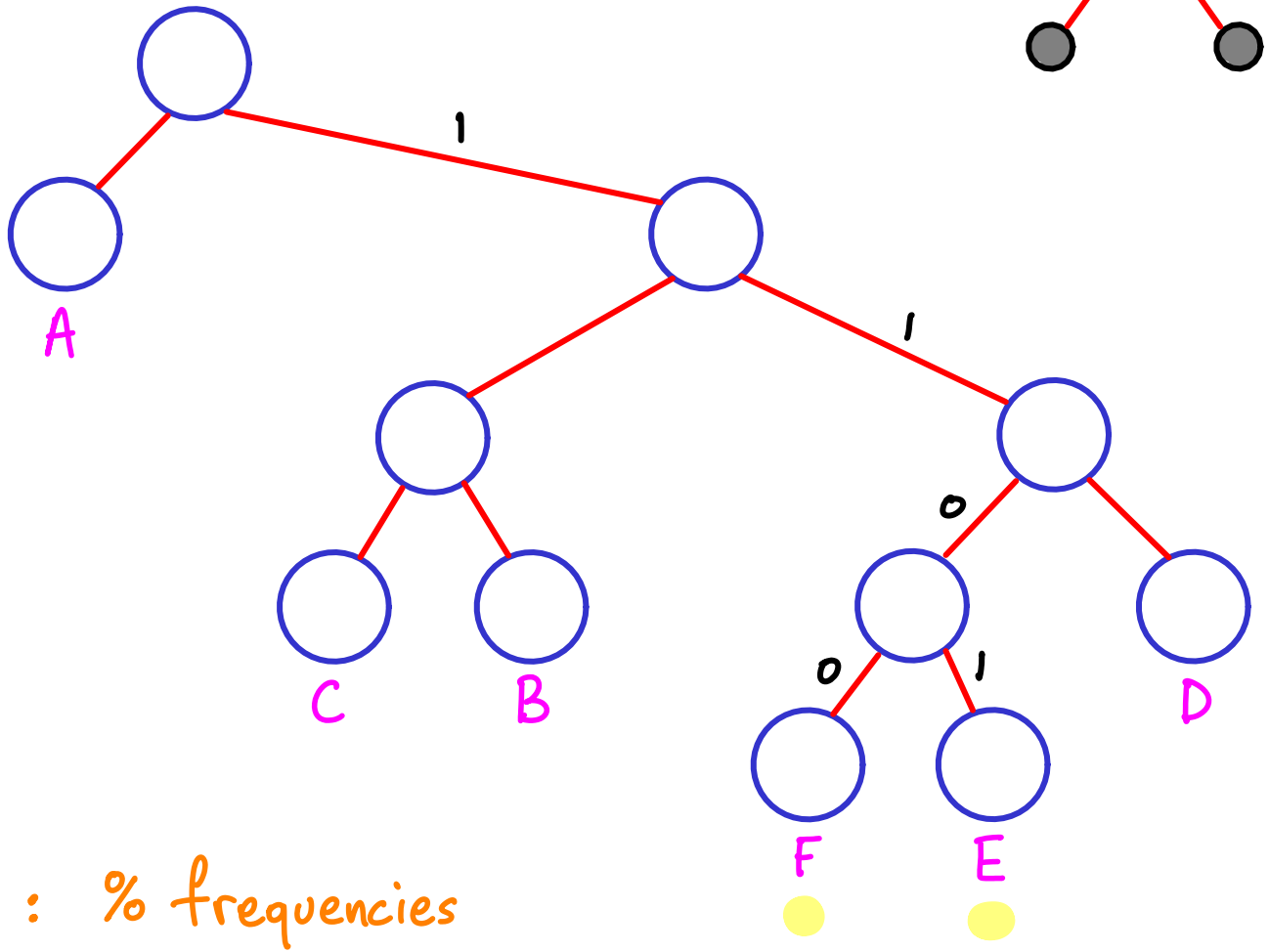




A	B	C	D	E	F	
45	13	12	16	9	5	: % frequencies
0	101	100	111	1101	1100	: Variable length (prefix) code

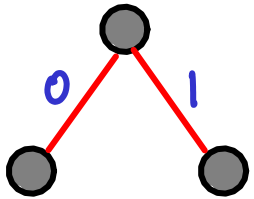
We still need to know when each code ends → use binary tree

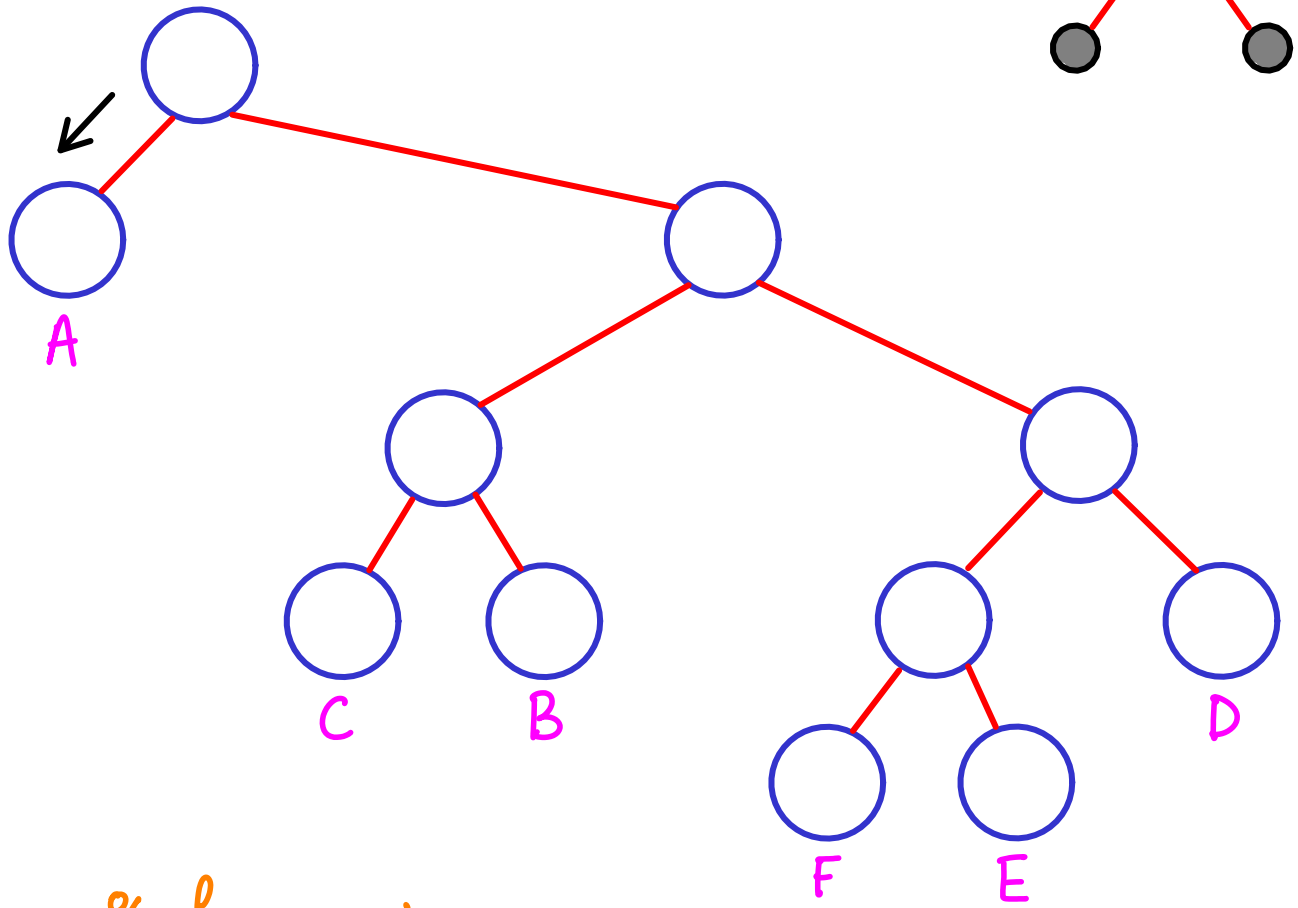




A	B	C	D	E	F	
45	13	12	16	9	5	: % frequencies
0	101	100	111	1101	1100	: Variable length (prefix) code

We still need to know when each code ends → use binary tree



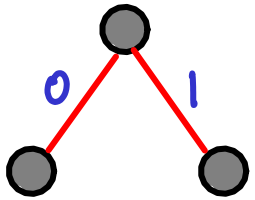


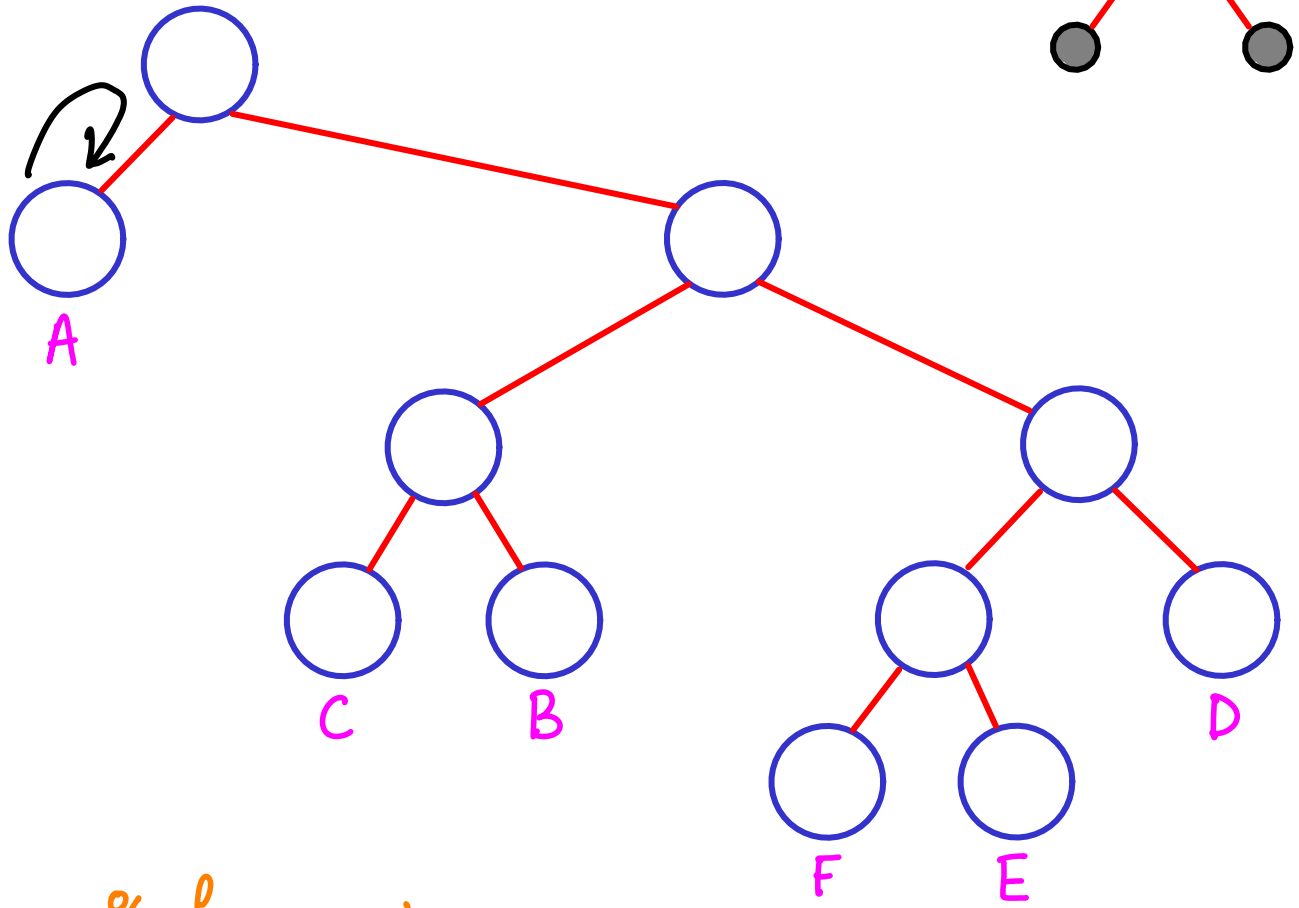
000100101011101101110011010010 1

A

A	B	C	D	E	F	
45	13	12	16	9	5	: % frequencies
0	101	100	111	1101	1100	: Variable length (prefix) code

We still need to know when each code ends → use binary tree



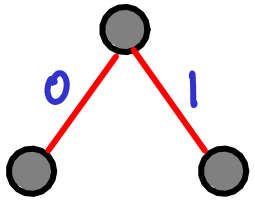


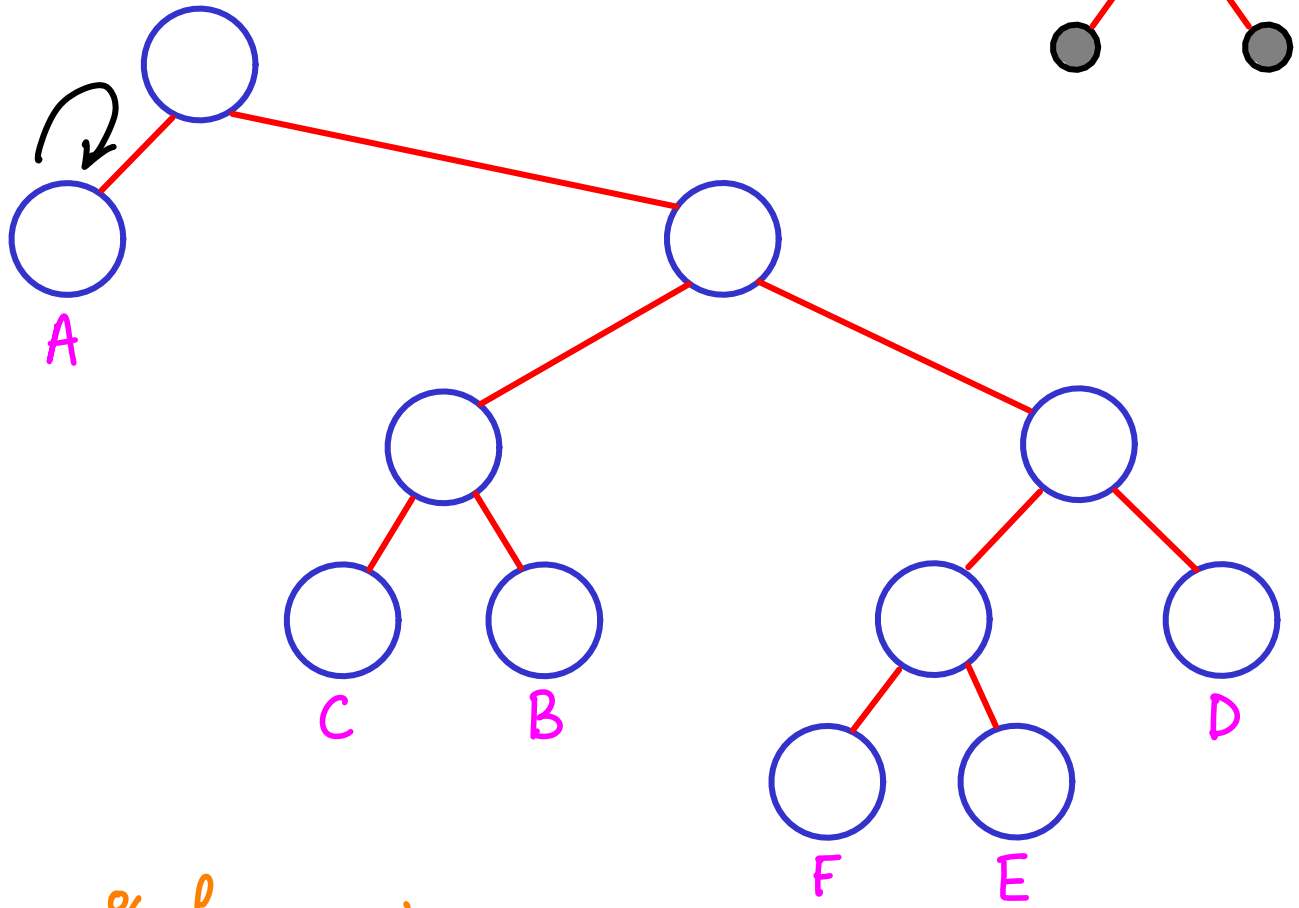
000100101011101101110011010010 1

AA

A	B	C	D	E	F	
45	13	12	16	9	5	: % frequencies
0	101	100	111	1101	1100	: Variable length (prefix) code

We still need to know when each code ends → use binary tree



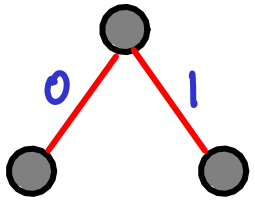


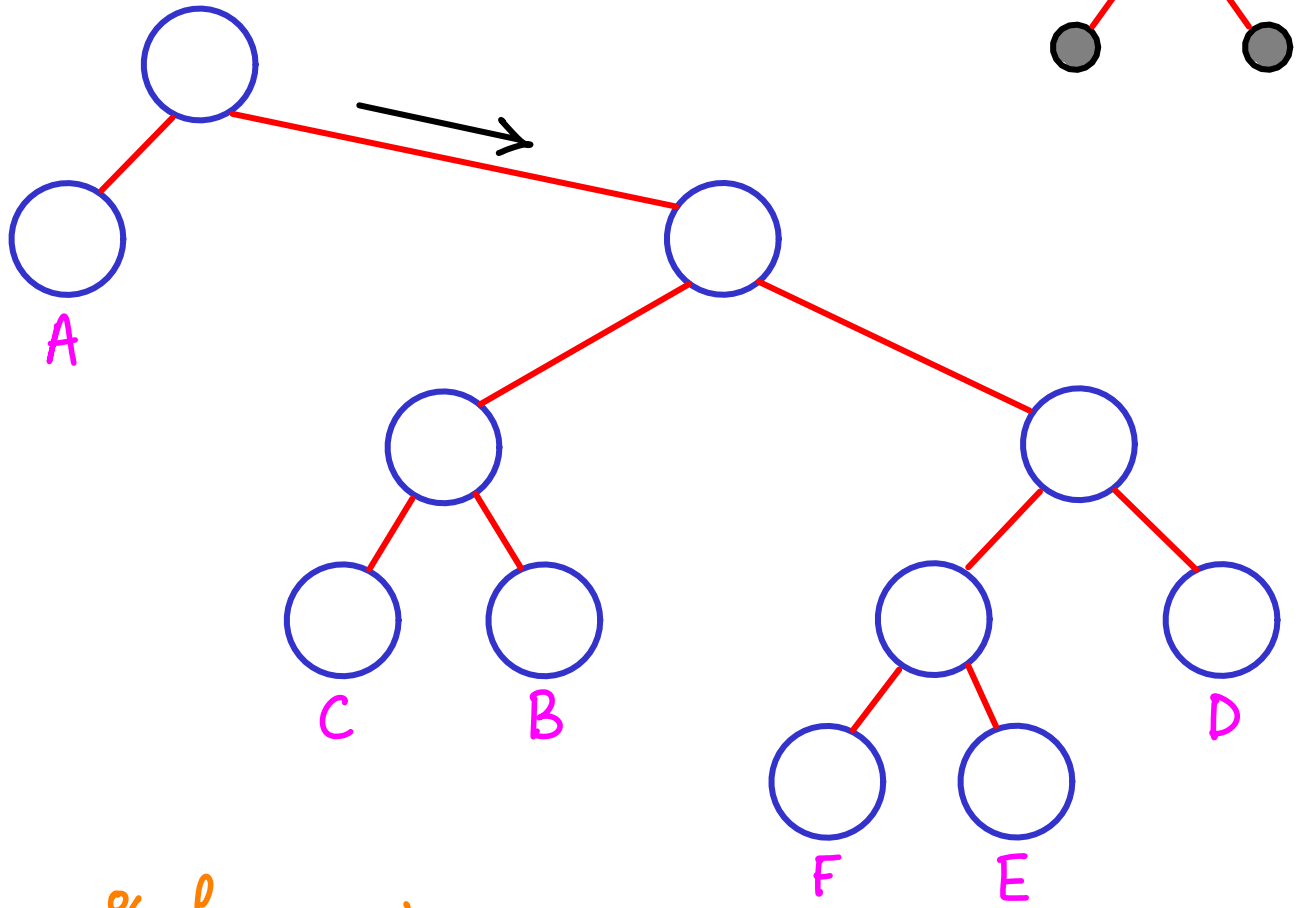
000100101011101101110011010010 1

AAA

A	B	C	D	E	F	
45	13	12	16	9	5	: % frequencies
0	101	100	111	1101	1100	: Variable length (prefix) code

We still need to know when each code ends → use binary tree



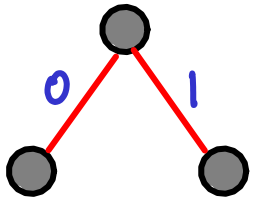


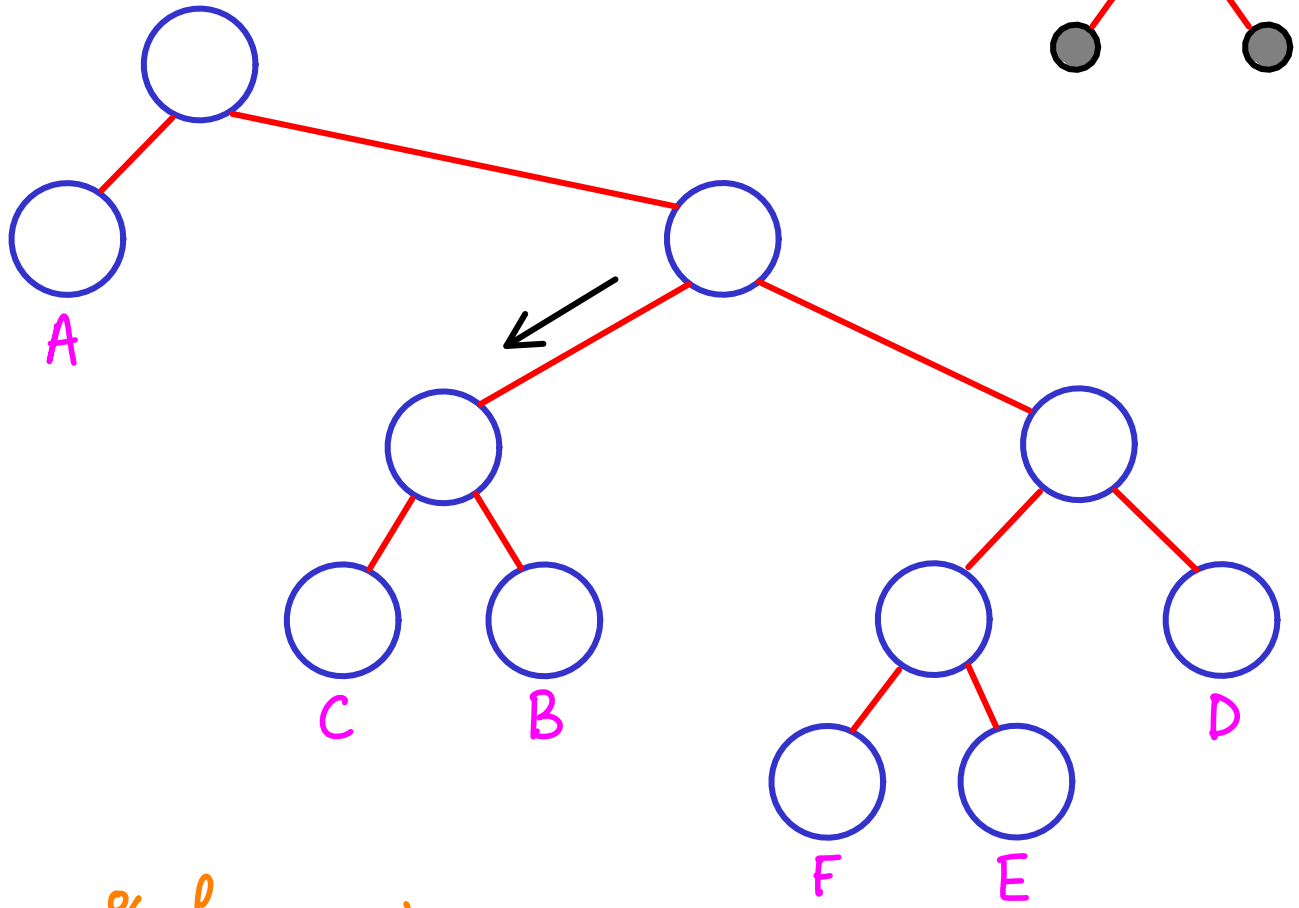
000100101011101101110011010010 1

AAA

A	B	C	D	E	F	
45	13	12	16	9	5	: % frequencies
0	101	100	111	1101	1100	: Variable length (prefix) code

We still need to know when each code ends → use binary tree



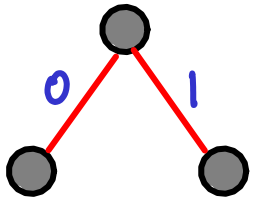


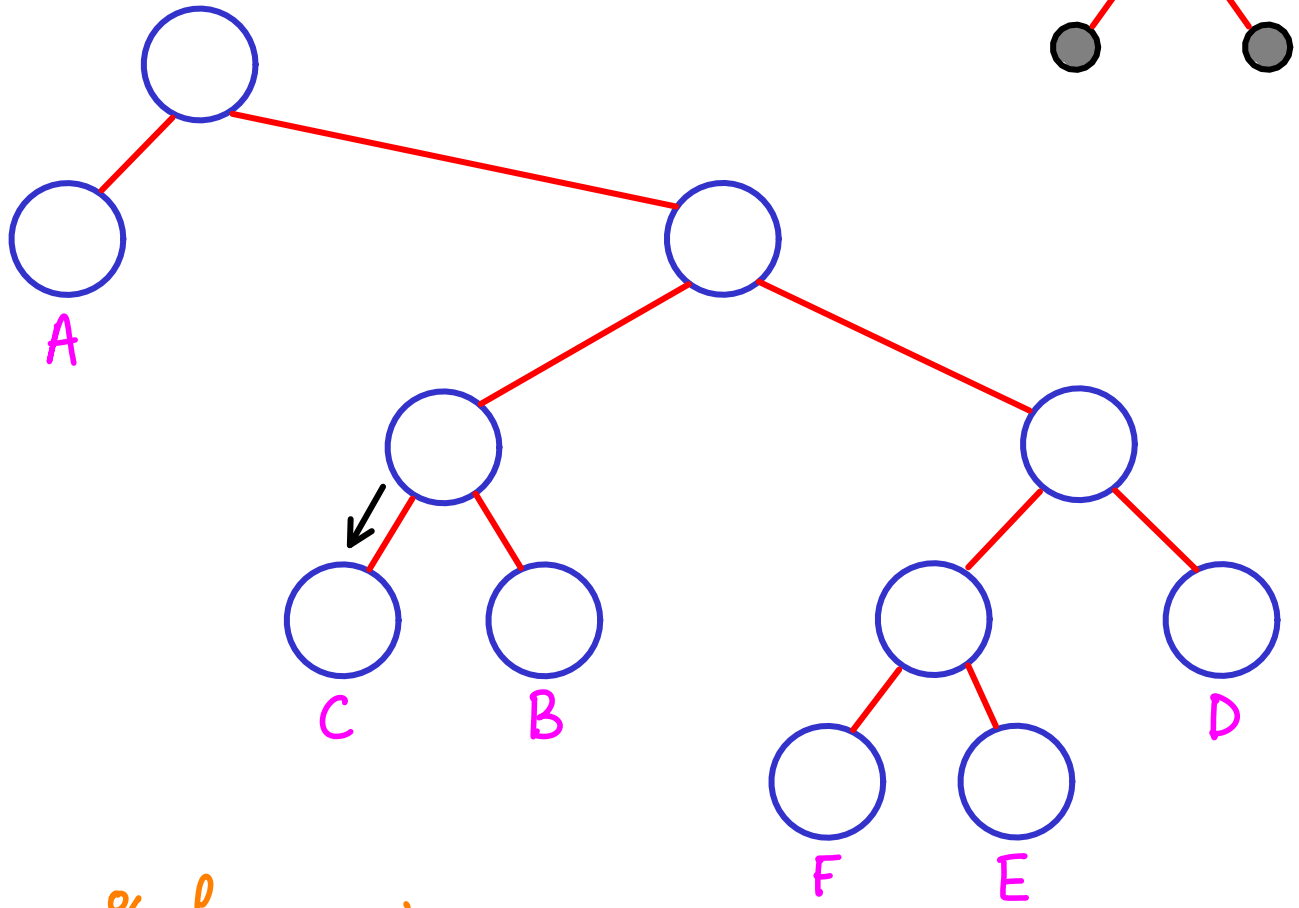
000100101011101101110011010010 1

AAA

A	B	C	D	E	F	
45	13	12	16	9	5	: % frequencies
0	101	100	111	1101	1100	: Variable length (prefix) code

We still need to know when each code ends → use binary tree



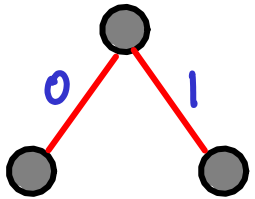


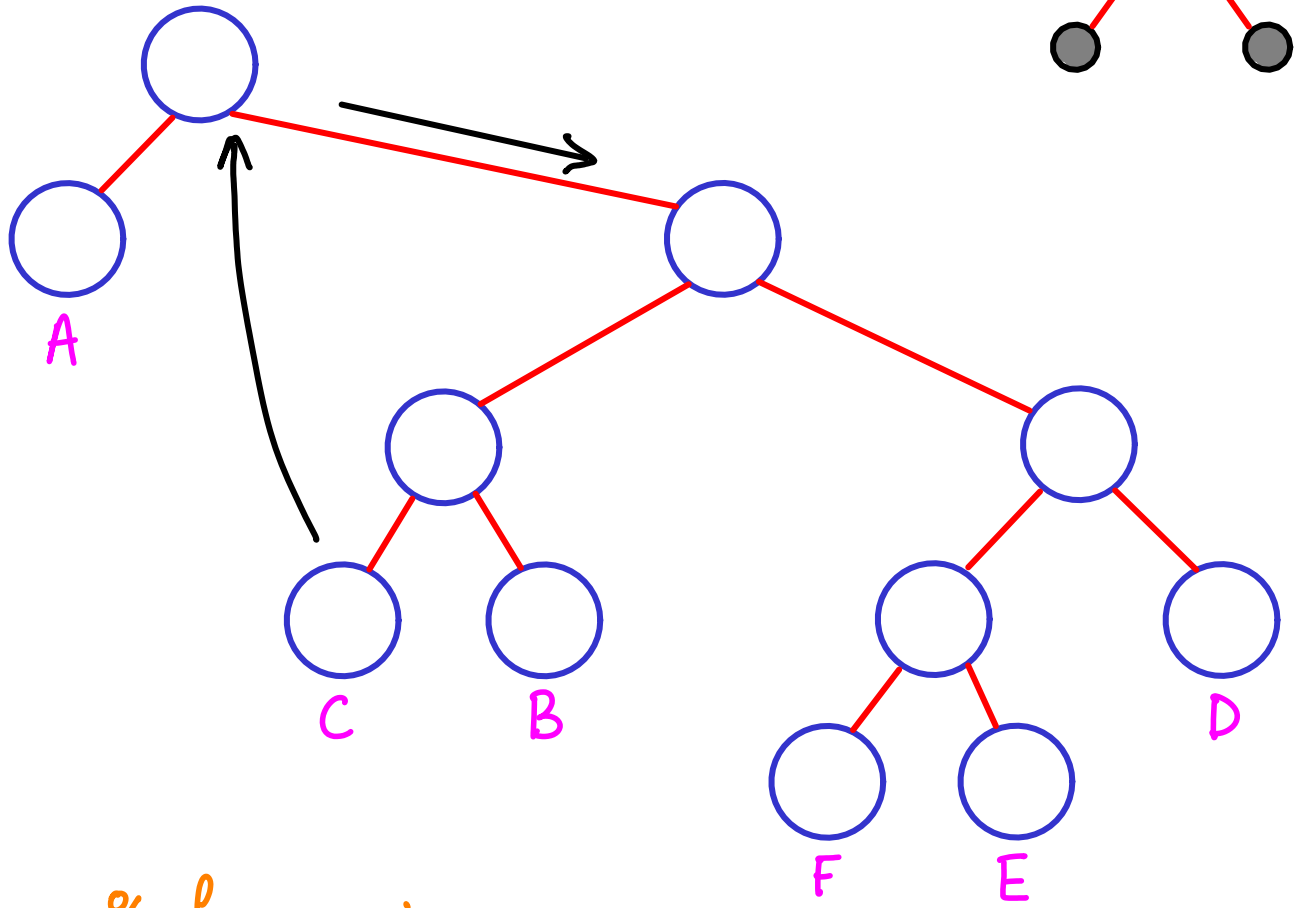
000100101011101101100111010010 1

AAA C

A	B	C	D	E	F	
45	13	12	16	9	5	: % frequencies
0	101	100	111	1101	1100	: Variable length (prefix) code

We still need to know when each code ends → use binary tree



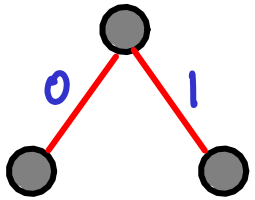


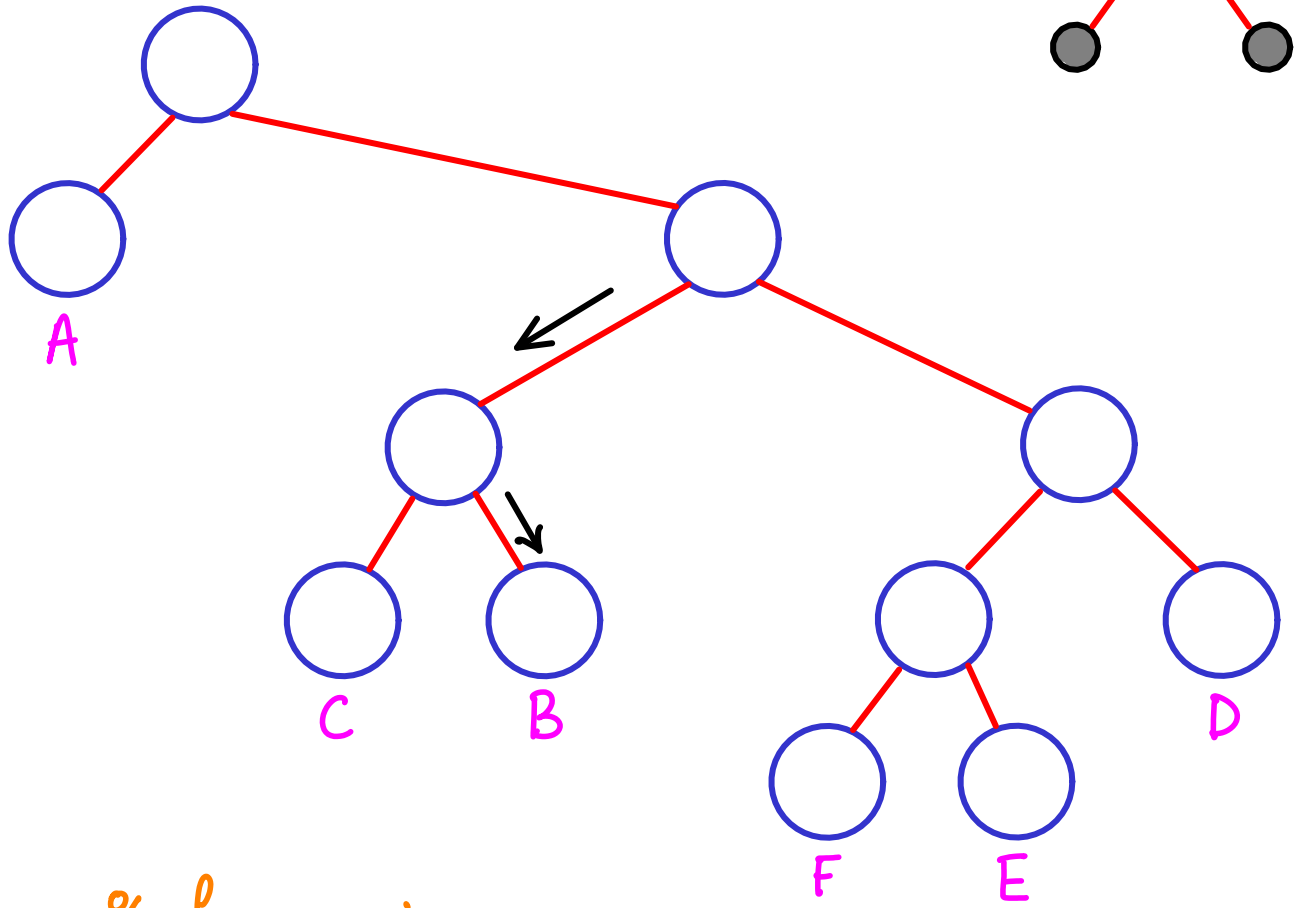
000100101011101101100111010010 1

AAA C

A	B	C	D	E	F	
45	13	12	16	9	5	: % frequencies
0	101	100	111	1101	1100	: Variable length (prefix) code

We still need to know when each code ends → use binary tree



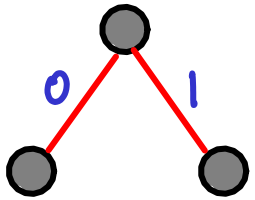


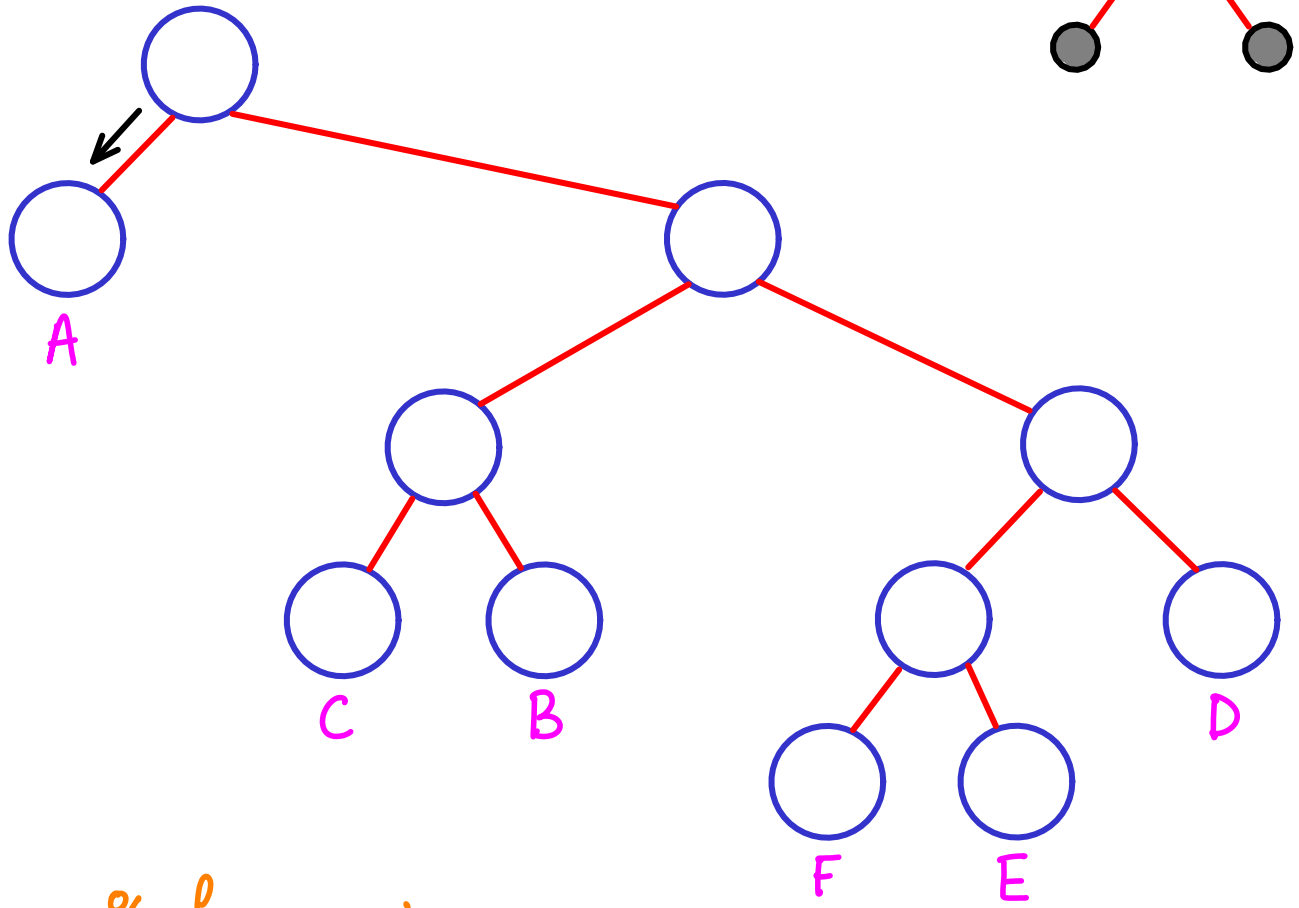
000100101011101101110011010010101

AAA C B

A	B	C	D	E	F	
45	13	12	16	9	5	: % frequencies
0	101	100	111	1101	1100	: Variable length (prefix) code

We still need to know when each code ends → use binary tree



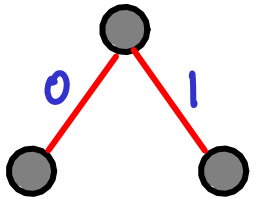


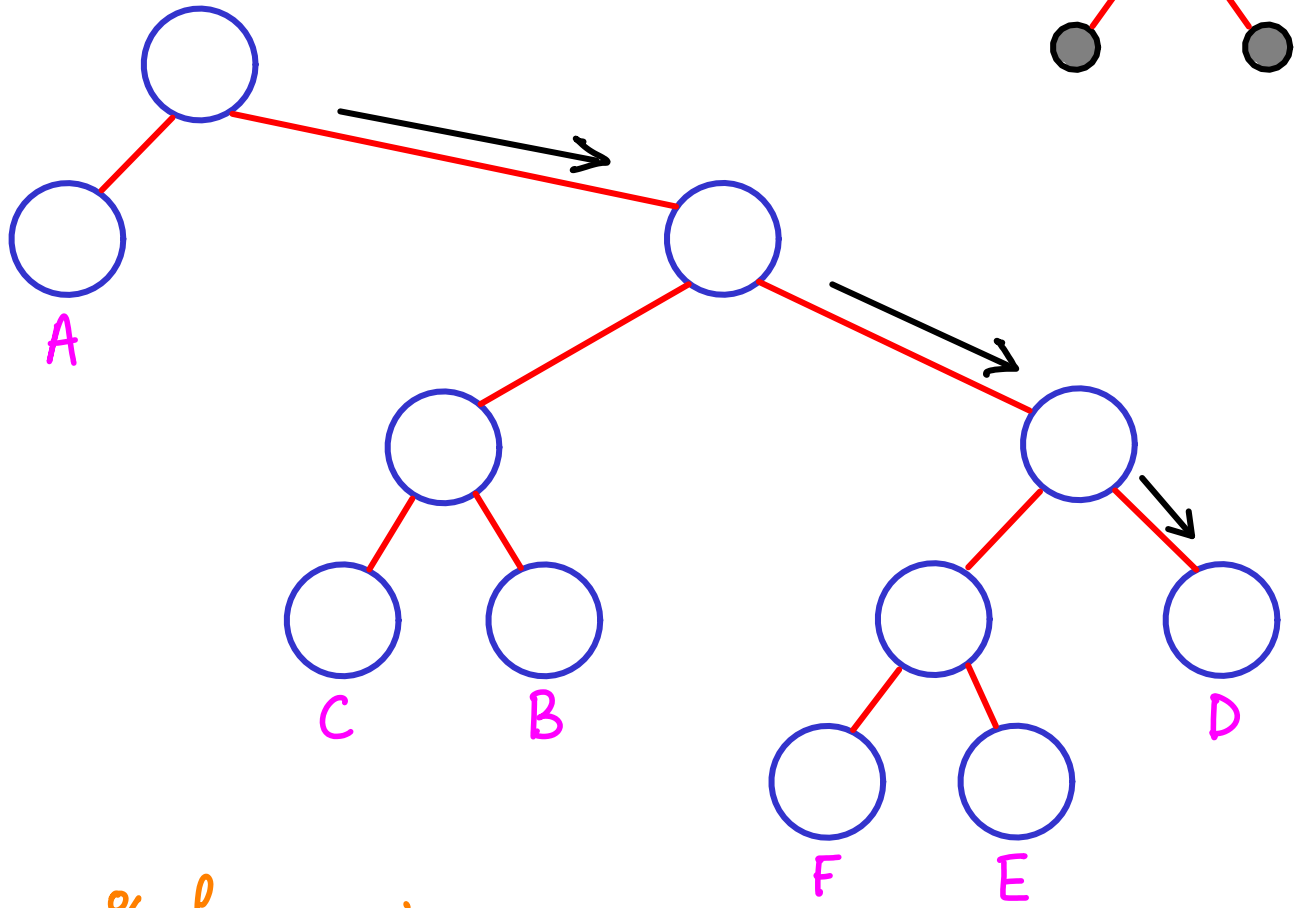
000100101011101101110011010100101

AAA C BA

A	B	C	D	E	F	
45	13	12	16	9	5	: % frequencies
0	101	100	111	1101	1100	: Variable length (prefix) code

We still need to know when each code ends → use binary tree





000|00|10|10|111|0|11|0|111|00|111|0|00|10|1

AAA C BA D etc

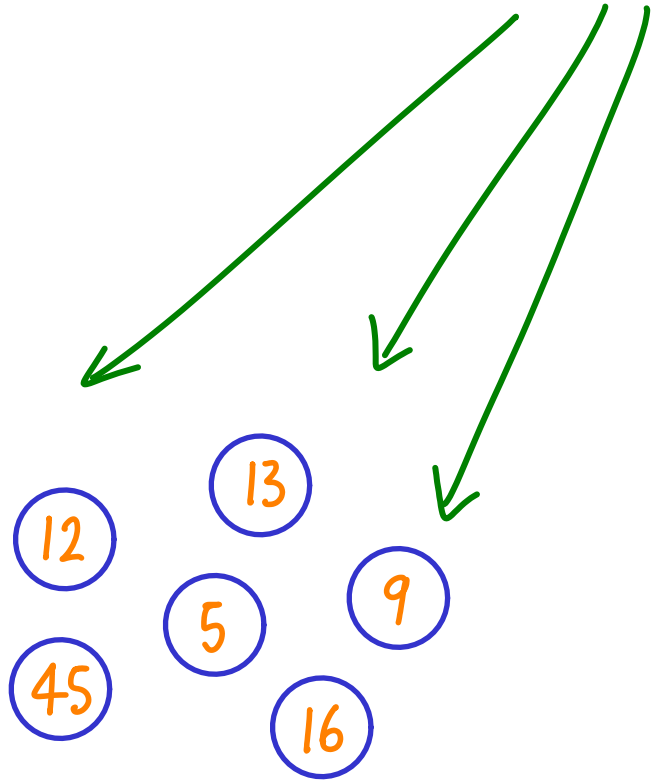
A	B	C	D	E	F	
45	13	12	16	9	5	: % frequencies
0	101	100	111	1101	1100	: Variable length (prefix) code

Huffman code

developed in 1951 as a solution to a class assignment

typical compression 20-90%

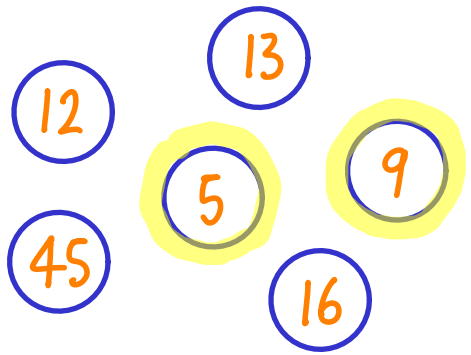
Huffman code: Make n trees of size 1: one tree per character.



A	B	C	D	E	F		
45	13	12	16	9	5	:	%

Huffman code: Make n trees of size 1: one tree per character.

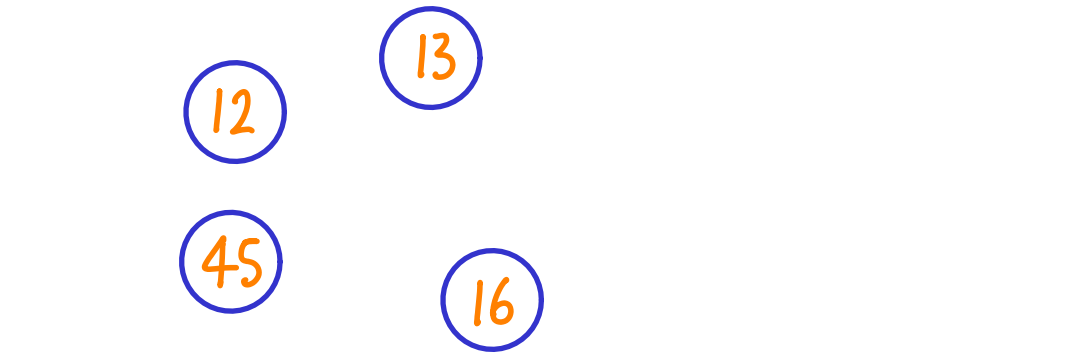
- The 2 roots with lowest frequencies become siblings.



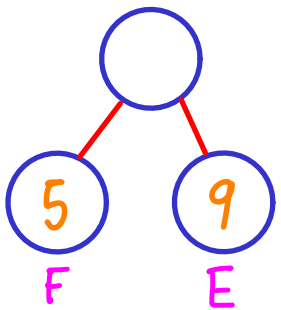
A	B	C	D	E	F		
45	13	12	16	9	5	:	%

Huffman code: Make n trees of size 1: one tree per character.

- The 2 roots with lowest frequencies become siblings.



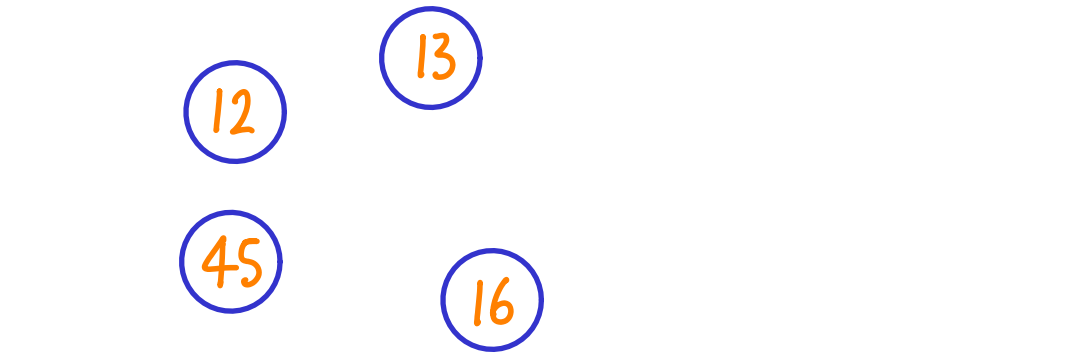
A	B	C	D	E	F		
45	13	12	16	9	5	:	%



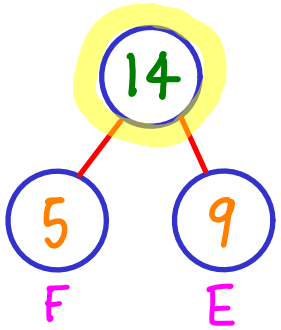
Huffman code: Make n trees of size 1: one tree per character.

The 2 roots with lowest frequencies become siblings.

- Their new parent gets the sum of their frequencies.



A	B	C	D	E	F		
45	13	12	16	9	5	:	%

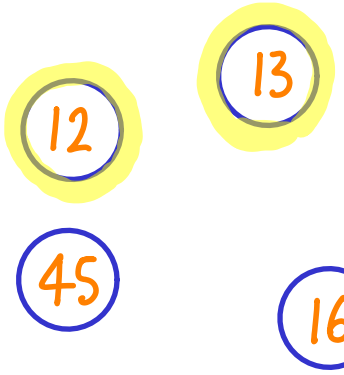


Huffman code: Make n trees of size 1: one tree per character.

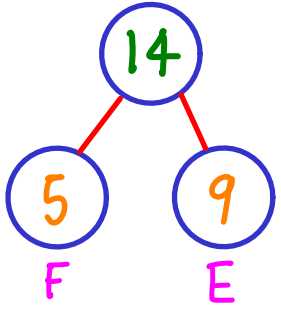
The 2 roots with lowest frequencies become siblings.

Their new parent gets the sum of their frequencies.

Repeat



A	B	C	D	E	F		
45	13	12	16	9	5	:	%

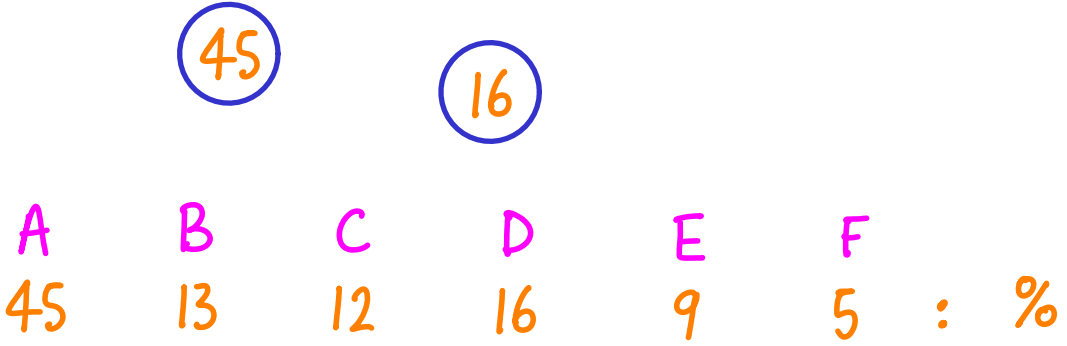


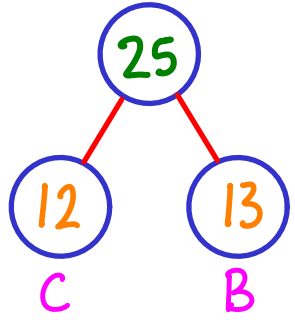
Huffman code: Make n trees of size 1: one tree per character.

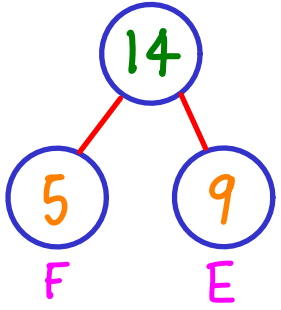
The 2 roots with lowest frequencies become siblings.

Their new parent gets the sum of their frequencies.

Repeat







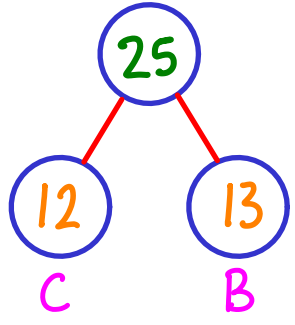
Huffman code: Make n trees of size 1: one tree per character.

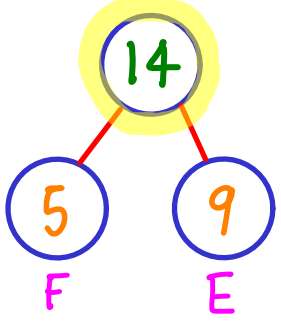
The 2 roots with lowest frequencies become siblings.

Their new parent gets the sum of their frequencies.

Repeat

A	B	C	D	E	F		
45	13	12	16	9	5	:	%





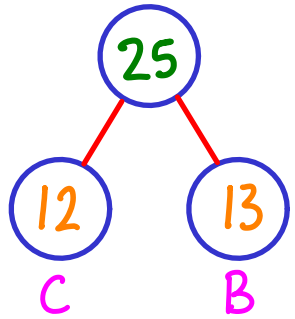
Huffman code: Make n trees of size 1: one tree per character.

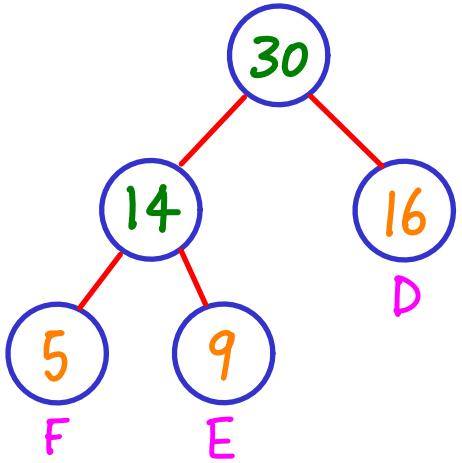
The 2 roots with lowest frequencies become siblings.

Their new parent gets the sum of their frequencies.

Repeat

A	B	C	D	E	F		
45	13	12	16	9	5	:	%





Huffman code: Make n trees of size 1: one tree per character.

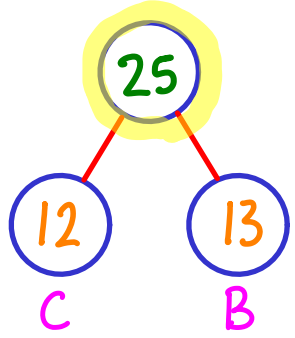
The 2 roots with lowest frequencies become siblings.

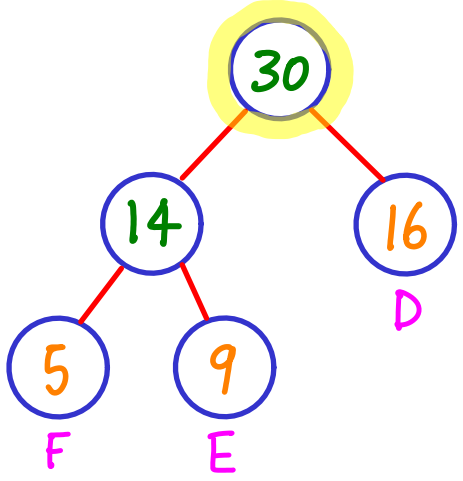
Their new parent gets the sum of their frequencies.

Repeat

A	B	C	D	E	F
45	13	12	16	9	5

: %





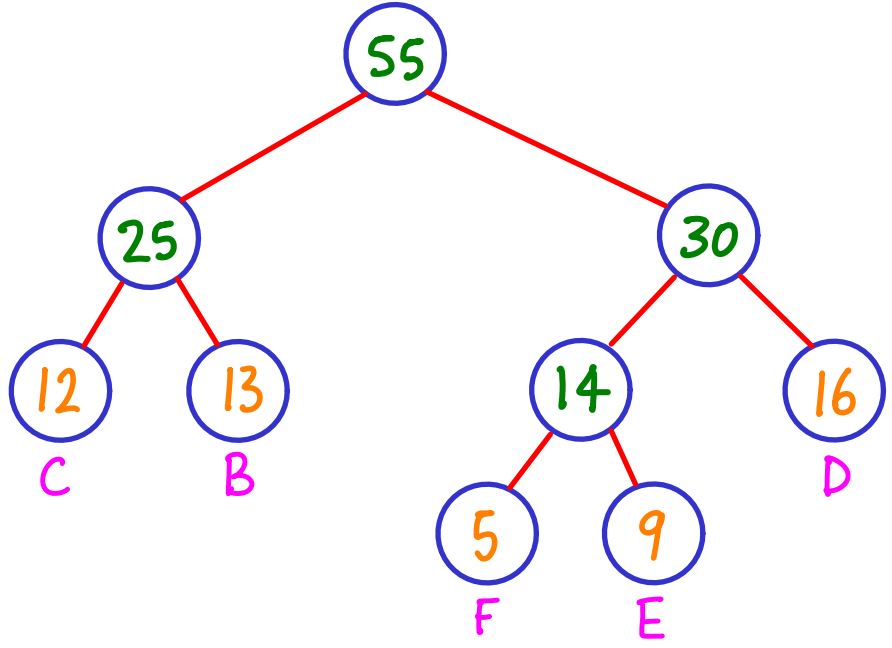
Huffman code: Make n trees of size 1: one tree per character.

The 2 roots with lowest frequencies become siblings.

Their new parent gets the sum of their frequencies.

Repeat

A	B	C	D	E	F		
45	13	12	16	9	5	:	%



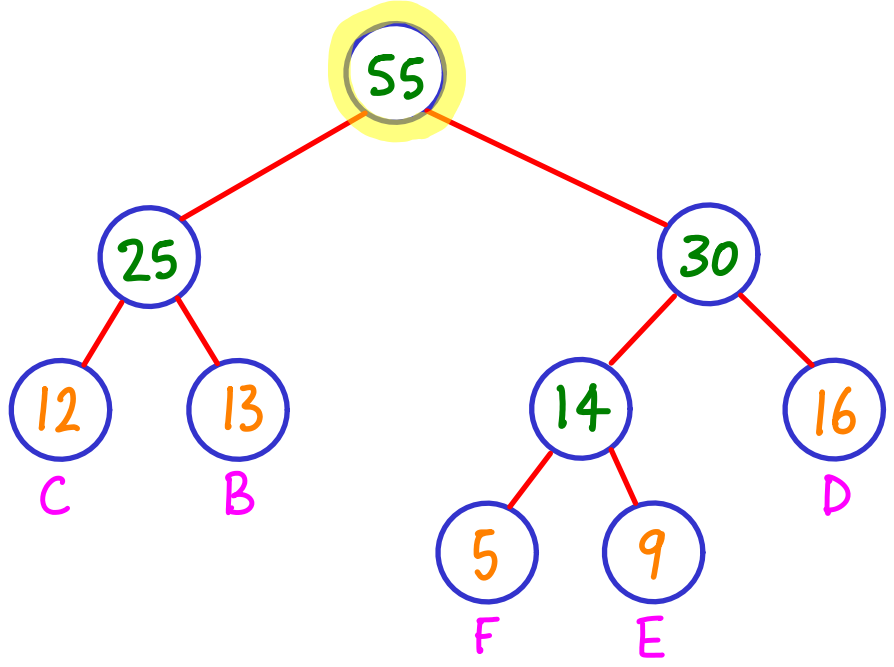
Huffman code: Make n trees of size 1: one tree per character.

The 2 roots with lowest frequencies become siblings.

Their new parent gets the sum of their frequencies.

Repeat

A	B	C	D	E	F		
45	13	12	16	9	5	:	%

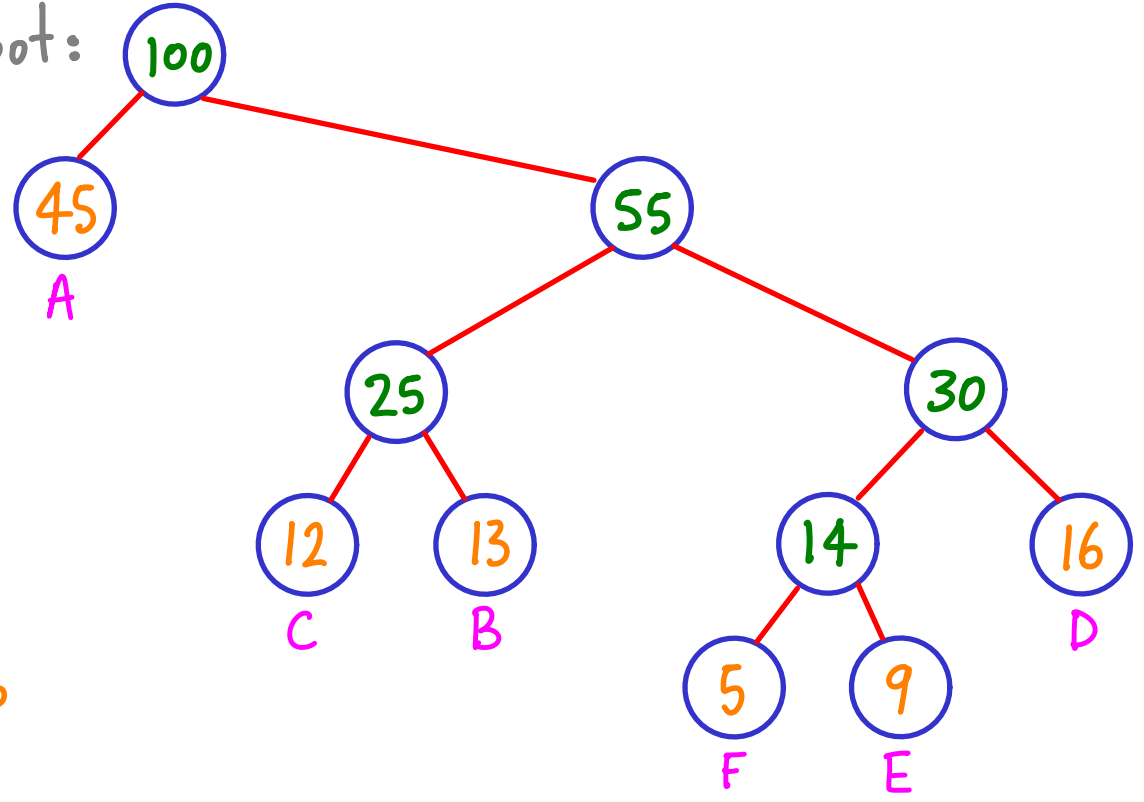


Huffman code: Make n trees of size 1: one tree per character.

The 2 roots with lowest frequencies become siblings.

Their new parent gets the sum of their frequencies.

Repeat until 1 root:



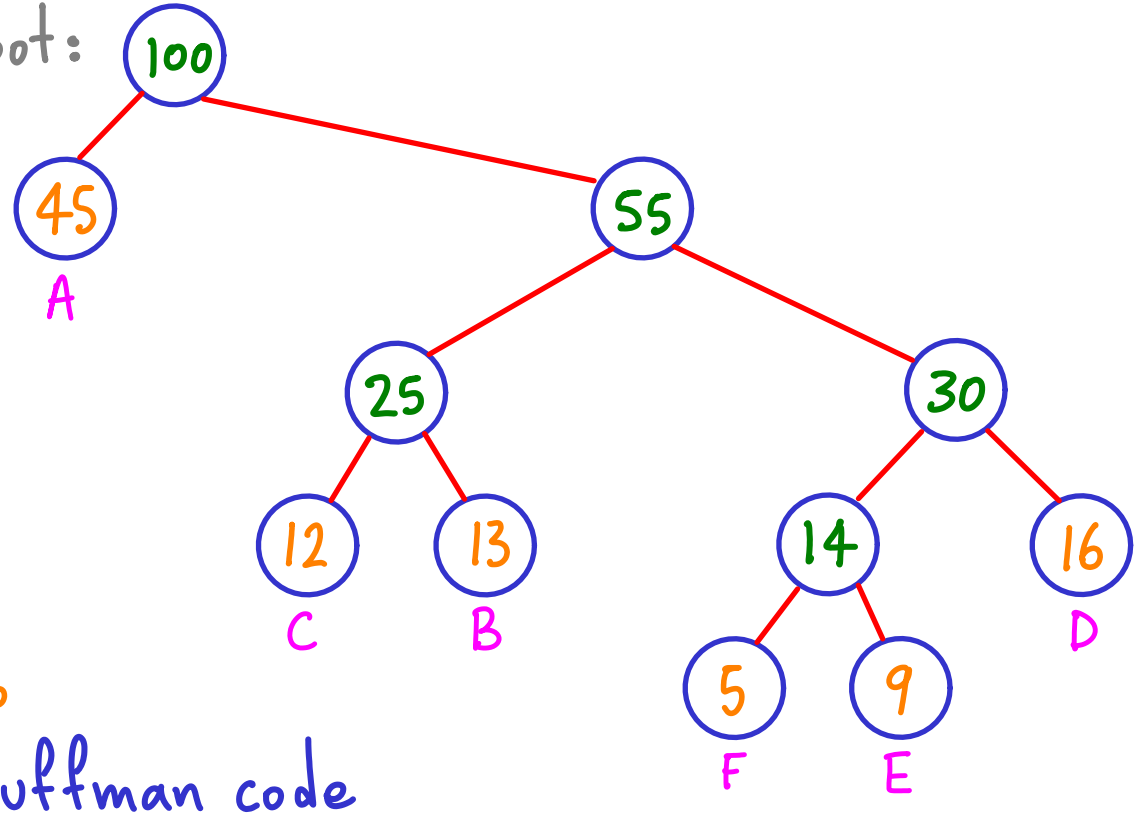
A	B	C	D	E	F		
45	13	12	16	9	5	:	%

Huffman code: Make n trees of size 1: one tree per character.

(greedy algo) → The 2 roots with lowest frequencies become siblings.

Their new parent gets the sum of their frequencies.

Repeat until 1 root:



A	B	C	D	E	F		
45	13	12	16	9	5	:	%
0	101	100	111	1101	1100	:	Huffman code

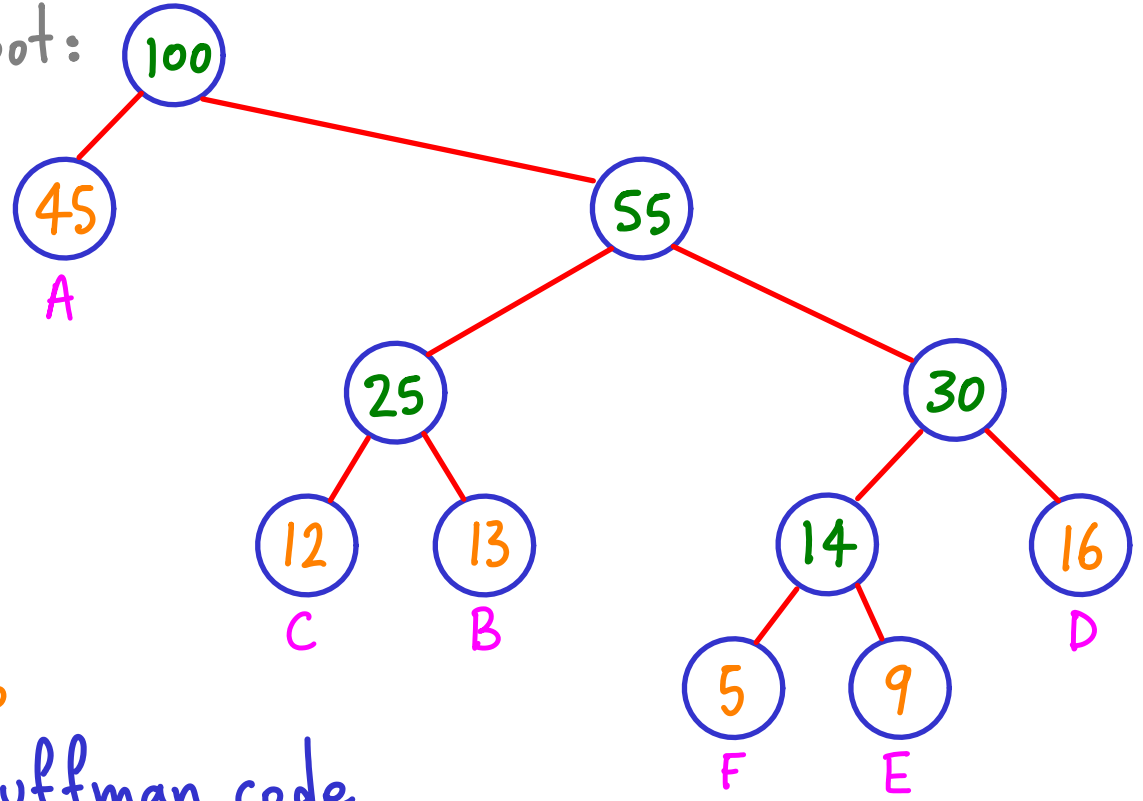
Huffman code: Make n trees of size 1: one tree per character.

The 2 roots with lowest frequencies become siblings.

Their new parent gets the sum of their frequencies.

Repeat until 1 root:

- Optimal! to show
- Not unique
  - arbitrary sibling order
  - weights could be equal



A	B	C	D	E	F	
45	13	12	16	9	5	: %
0	101	100	111	1101	1100	: Huffman code

Huffman code: time to build?

Huffman code: easy to implement in $O(n \log n)$ time & $\Theta(n)$ space

- priority queue to identify 2 lowest frequencies
  - initialize with n chars : $\Theta(n)$
  - extract twice, insert new parent node : $O(\log n)$
  - n-1 rounds

Huffman code: easy to implement in $O(n \log n)$ time & $\Theta(n)$ space

→ priority queue to identify 2 lowest frequencies

- initialize with n chars : $\Theta(n)$
- extract twice, insert new parent node : $O(\log n)$
- n-1 rounds

$O(n \log\log n)$ possible, using a van Emde Boas tree

CLRS ch. 20

Huffman code: Not unique

Must store representation, like BST, otherwise decoder can't work.

↳ costs extra space :(

OPTIMALITY

minimize $\sum_{\text{all chars}} (\text{freq.} \cdot \text{bit length}) = \sum (\text{freq.} \cdot \text{node depth})$

- Not optimal number of bits in general (more later)
- We get OPT assuming one codeword per character

OPTIMALITY

minimize $\sum_{\text{all chars}} (\text{freq.} \cdot \text{bit length}) = \sum (\text{freq.} \cdot \text{node depth})$

Claim: there is always a prefix code that achieves OPT

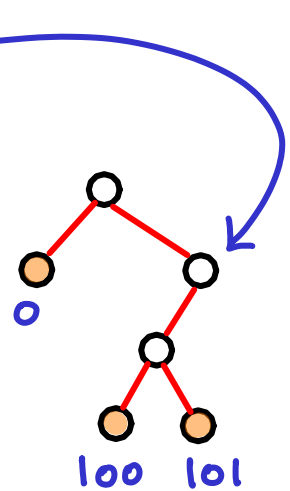
- CLRS skips the proof
- See Sayood 2.4.3 : 3-page proof

OPTIMALITY

minimize $\sum_{\text{all chars}} (\text{freq.} \cdot \text{bit length}) = \sum (\text{freq.} \cdot \text{node depth})$

Claim 2:

OPT tree can't have an internal node w/ 1 child



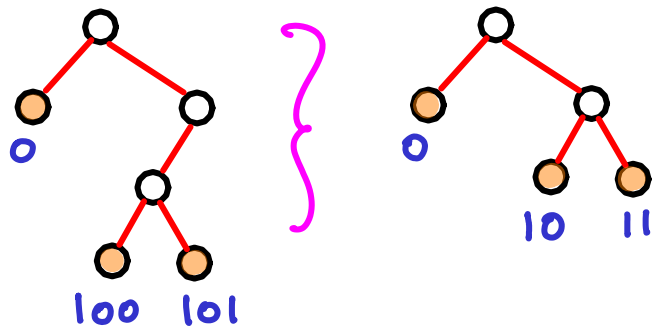
OPTIMALITY

minimize $\sum_{\text{all chars}} (\text{freq.} \cdot \text{bit length}) = \sum (\text{freq.} \cdot \text{node depth})$

Claim 2:

OPT tree can't have an internal node w/ 1 child

↳ contract, improve Σ, still a prefix code : contradiction



OPTIMALITY

minimize $\sum$ (freq. · node depth)

✓ OPT tree can't have an internal node w/ 1 child

Claim 3:

∃ OPT tree w/ 2 lowest frequencies as siblings at lowest level

OPTIMALITY

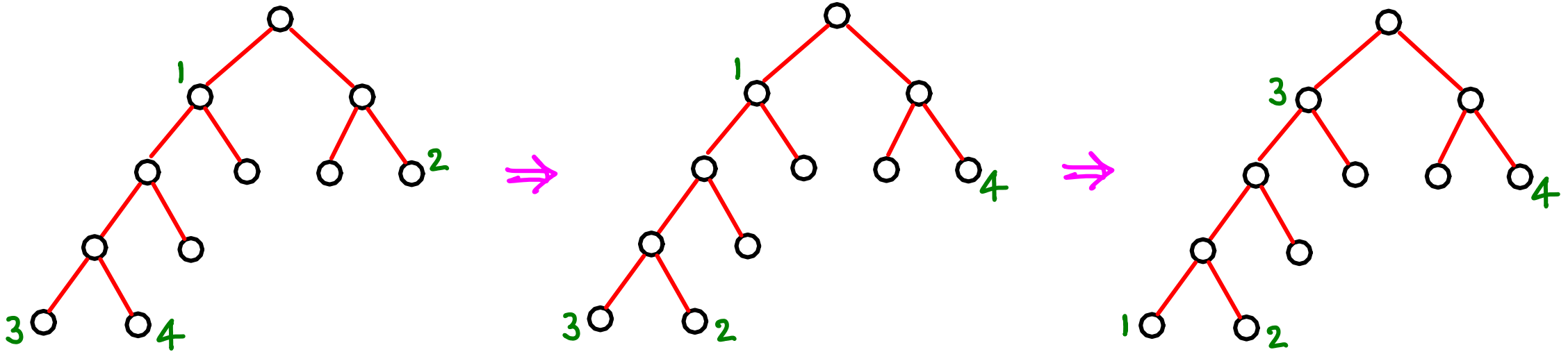
minimize $\sum$ (freq. · node depth)

✓ OPT tree can't have an internal node w/ 1 child

Claim 3:

∃ OPT tree w/ 2 lowest frequencies as siblings at lowest level

↳ by contradiction: take "OPT", swap nodes, improve $\sum$



OPTIMALITY

minimize $\sum$ (freq. · node depth)

- ✓ OPT tree can't have an internal node w/ 1 child
- ✓ $\exists$ OPT tree w/ 2 lowest frequencies as siblings at lowest level

OPTIMALITY

minimize $\sum$ (freq. · node depth)

✓ OPT tree can't have an internal node w/ 1 child

✓ ∃ OPT tree w/ 2 lowest frequencies as siblings at lowest level

Last claim:

if we merge 2 lowest frequencies x, y in tree T
and new tree T' is OPT'
then T was OPT.

OPTIMALITY

minimize $\sum$ (freq. · node depth)

✓ OPT tree can't have an internal node w/ 1 child

✓ ∃ OPT tree w/ 2 lowest frequencies as siblings at lowest level

Last claim:

if we merge 2 lowest frequencies x,y in tree T
and new tree T′ is OPT′ (node z : x.freq + y.freq)
then T was OPT.

Weight diff for T, T' = $1 \cdot (x.freq + y.freq)$ // all other nodes unchanged

OPTIMALITY

minimize $\sum$ (freq. · node depth)

✓ OPT tree can't have an internal node w/ 1 child

✓ ∃ OPT tree w/ 2 lowest frequencies as siblings at lowest level *

Last claim:

if we merge 2 lowest frequencies x,y in tree T
and new tree T′ is OPT′ (node z : x.freq + y.freq)
then T was OPT.

Weight diff for T, T′ = 1 · (x.freq + y.freq) // all other nodes unchanged

↳ Same for diff of any OPT tree that beats T: x,y must be siblings* so we can contract them & get a tree that beats T′ : contradiction

OPTIMALITY

minimize $\sum$ (freq. · node depth)

- ✓ OPT tree can't have an internal node w/ 1 child
- ✓ ∃ OPT tree w/ 2 lowest frequencies as siblings at lowest level
- ✓ if we merge 2 lowest frequencies x,y in tree T and new tree T′ is optimal then T was OPT.
  - ↳ merge 2 smallest, apply induction, uncontract → get OPT
  - ↳ incremental construction is equivalent

□

A few words about extensions & other ideas

Huffman code with presorted frequencies

- use 2 queues instead of 1 priority queue

Huffman code with presorted frequencies

- use 2 queues instead of 1 priority queue
    - place frequencies (single nodes) in queue A
    - queue B holds merged nodes

Huffman code with presorted frequencies

- use 2 queues instead of 1 priority queue
  - place frequencies (single nodes) in queue A
  - queue B holds merged nodes
  - merge two smallest frequencies & insert in B, repeat

    (min two elements are always at top 2+2 positions)

Huffman code with presorted frequencies

O(n) time

- use 2 queues instead of 1 priority queue
  - place frequencies (single nodes) in queue A
  - queue B holds merged nodes
  - merge two smallest frequencies & insert in B, repeat

(min two elements are always at top 2+2 positions)

- proof of correctness is straightforward

Huffman code with minimum codeword length (variance)

Huffman code with minimum codeword length (variance)

- prioritize merging shorter trees
    - sort all frequencies, place in queue A
    - use queue B as for presorted data

Huffman code with minimum codeword length (variance)

- prioritize merging shorter trees
  - sort all frequencies, place in queue A
  - use queue B as for presorted data
  - when dequeing min two elements, prioritize first queue if tied

Decompressing options (affects storage / transmission)

- store frequencies (could be really bad)
- store tree
- use "canonical tree"

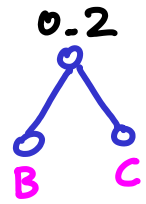
⋮

adaptive Huffman coding

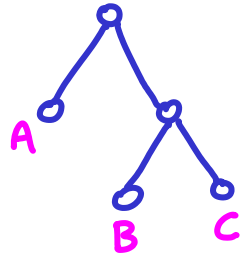
A	B	C
0.8	0.02	0.18

A B C

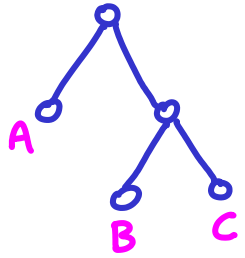
0.8 0.02 0.18



A	B	C
0	11	10
0.8	0.02	0.18



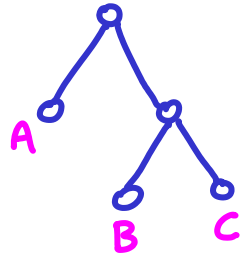
A	B	C
0	11	10
0.8	0.02	0.18



$0.8 + 0.04 + 0.36 = 1.2$

Improve this by encoding blocks

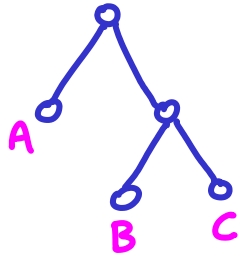
A	B	C
0	11	10
0.8	0.02	0.18



$0.8 + 0.04 + 0.36 = 1.2$

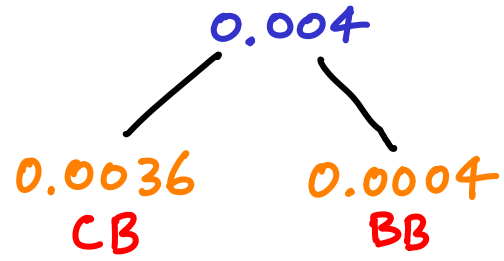
AA	0.64
AB	0.016
AC	0.144
BA	0.016
BB	0.0004
BC	0.0036
CA	0.144
CB	0.0036
CC	0.0324

A	B	C
0	11	10
0.8	0.02	0.18

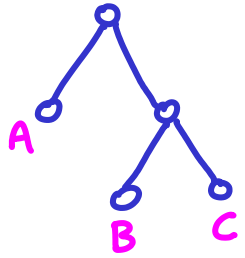


$0.8 + 0.04 + 0.36 = 1.2$

AA	0.64
AB	0.016
AC	0.144
BA	0.016
BB	0.0004
BC	0.0036
CA	0.144
CB	0.0036
CC	0.0324

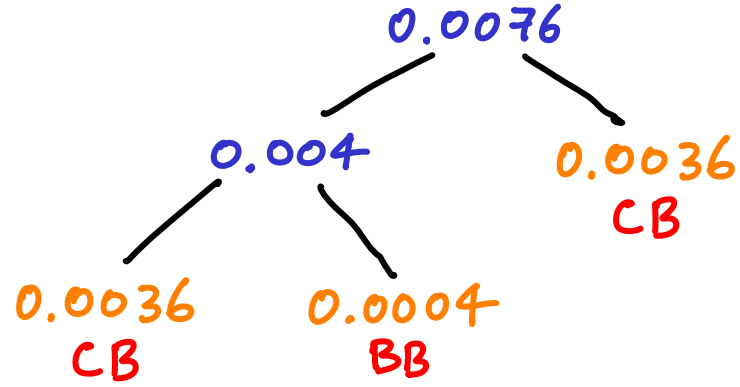


A	B	C
0	11	10
0.8	0.02	0.18

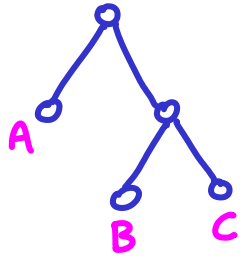


$0.8 + 0.04 + 0.36 = 1.2$

AA	0.64
AB	0.016
AC	0.144
BA	0.016
BB	0.0004 •
BC	0.0036 •
CA	0.144
CB	0.0036 ⊙
CC	0.0324

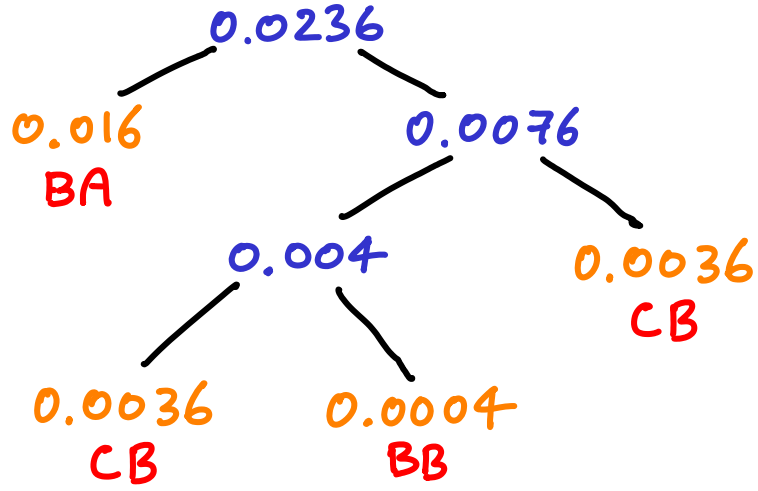


A	B	C
0	11	10
0.8	0.02	0.18

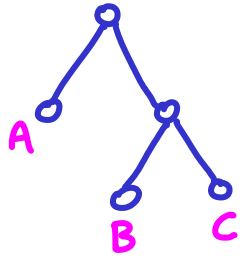


$0.8 + 0.04 + 0.36 = 1.2$

AA	0.64	
AB	0.016	
AC	0.144	
BA	0.016	⊙
BB	0.0004	•
BC	0.0036	•
CA	0.144	
CB	0.0036	•
CC	0.0324	

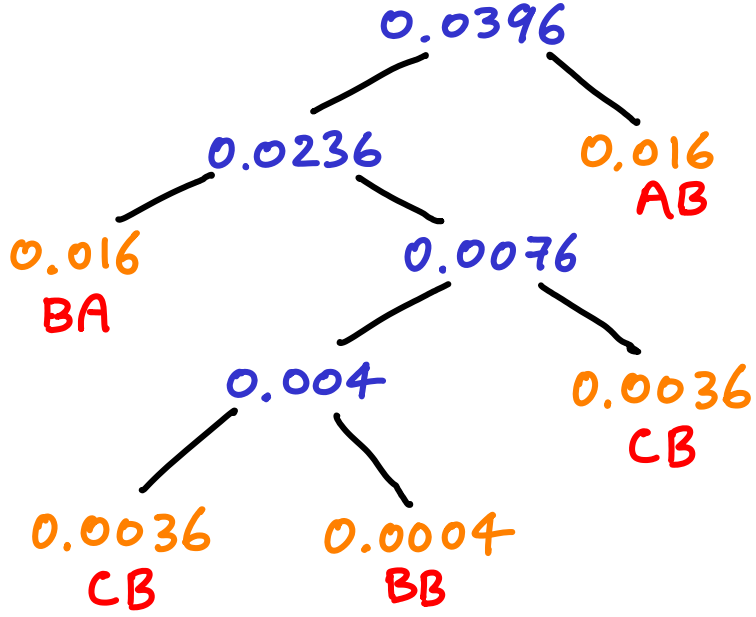


A	B	C
0	11	10
0.8	0.02	0.18

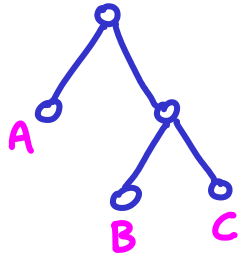


$0.8 + 0.04 + 0.36 = 1.2$

AA	0.64
AB	0.016 ⊙
AC	0.144
BA	0.016 •
BB	0.0004 •
BC	0.0036 •
CA	0.144
CB	0.0036 •
CC	0.0324

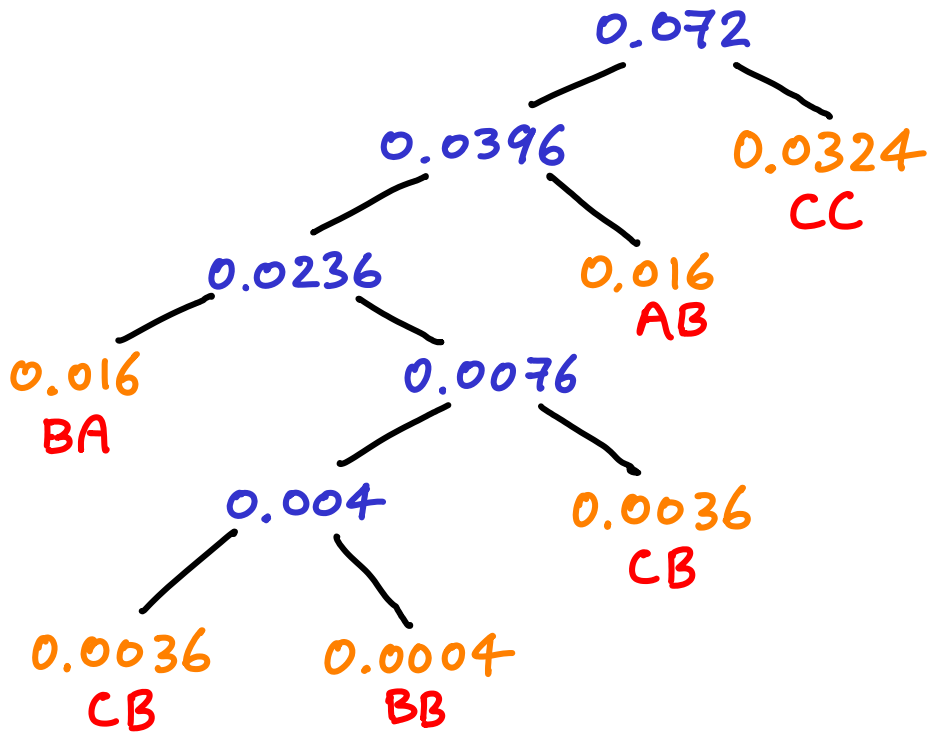


A	B	C
0	11	10
0.8	0.02	0.18

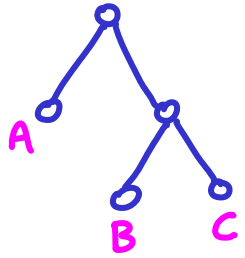


$0.8 + 0.04 + 0.36 = 1.2$

AA	0.64	
AB	0.016	•
AC	0.144	
BA	0.016	•
BB	0.0004	•
BC	0.0036	•
CA	0.144	
CB	0.0036	•
CC	0.0324	⊙

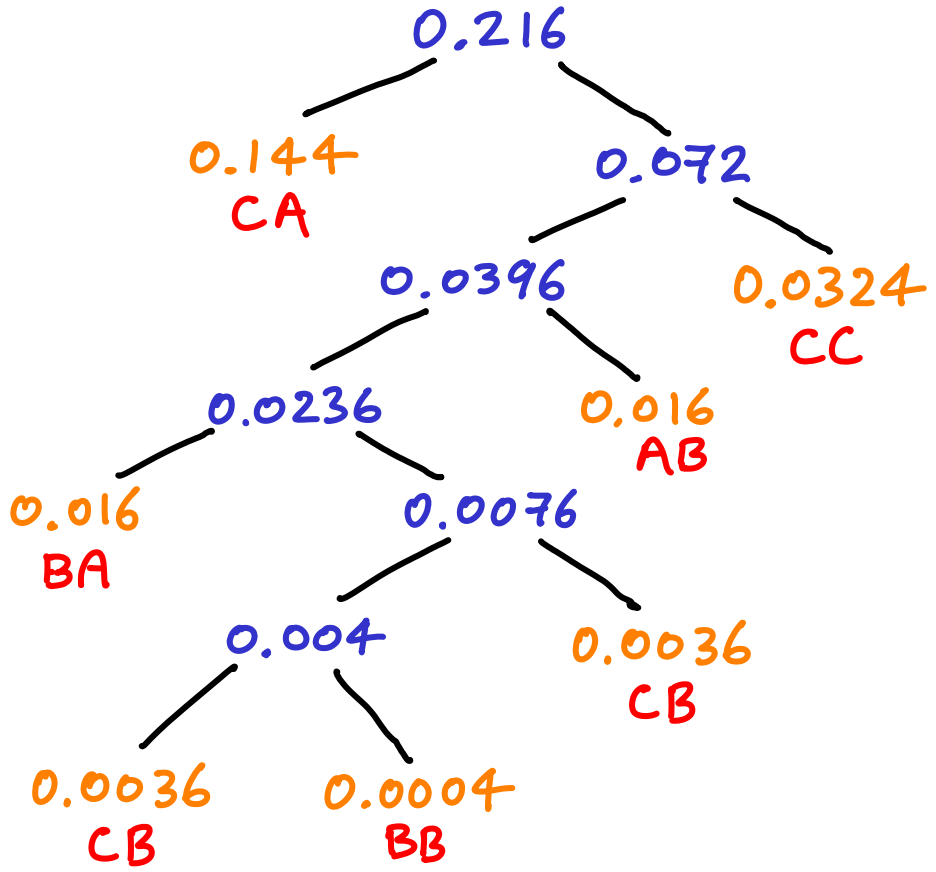


A	B	C
0	11	10
0.8	0.02	0.18

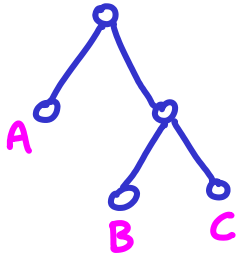


$0.8 + 0.04 + 0.36 = 1.2$

AA	0.64	
AB	0.016	•
AC	0.144	
BA	0.016	•
BB	0.0004	•
BC	0.0036	•
CA	0.144	⊙
CB	0.0036	•
CC	0.0324	•

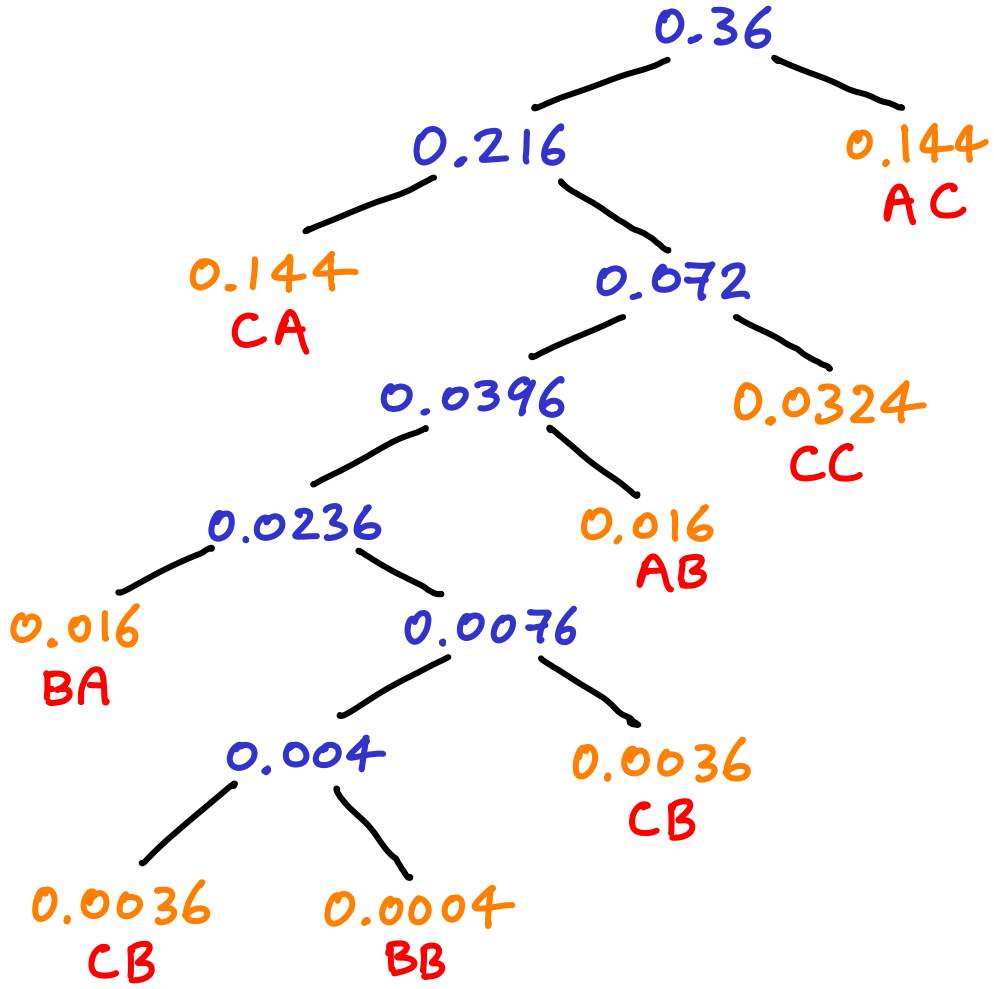


A	B	C
0	11	10
0.8	0.02	0.18

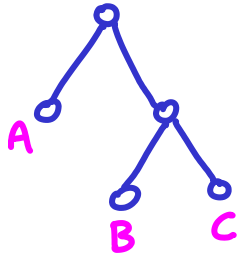


$0.8 + 0.04 + 0.36 = 1.2$

AA	0.64	
AB	0.016	•
AC	0.144	⊙
BA	0.016	•
BB	0.0004	•
BC	0.0036	•
CA	0.144	•
CB	0.0036	•
CC	0.0324	•

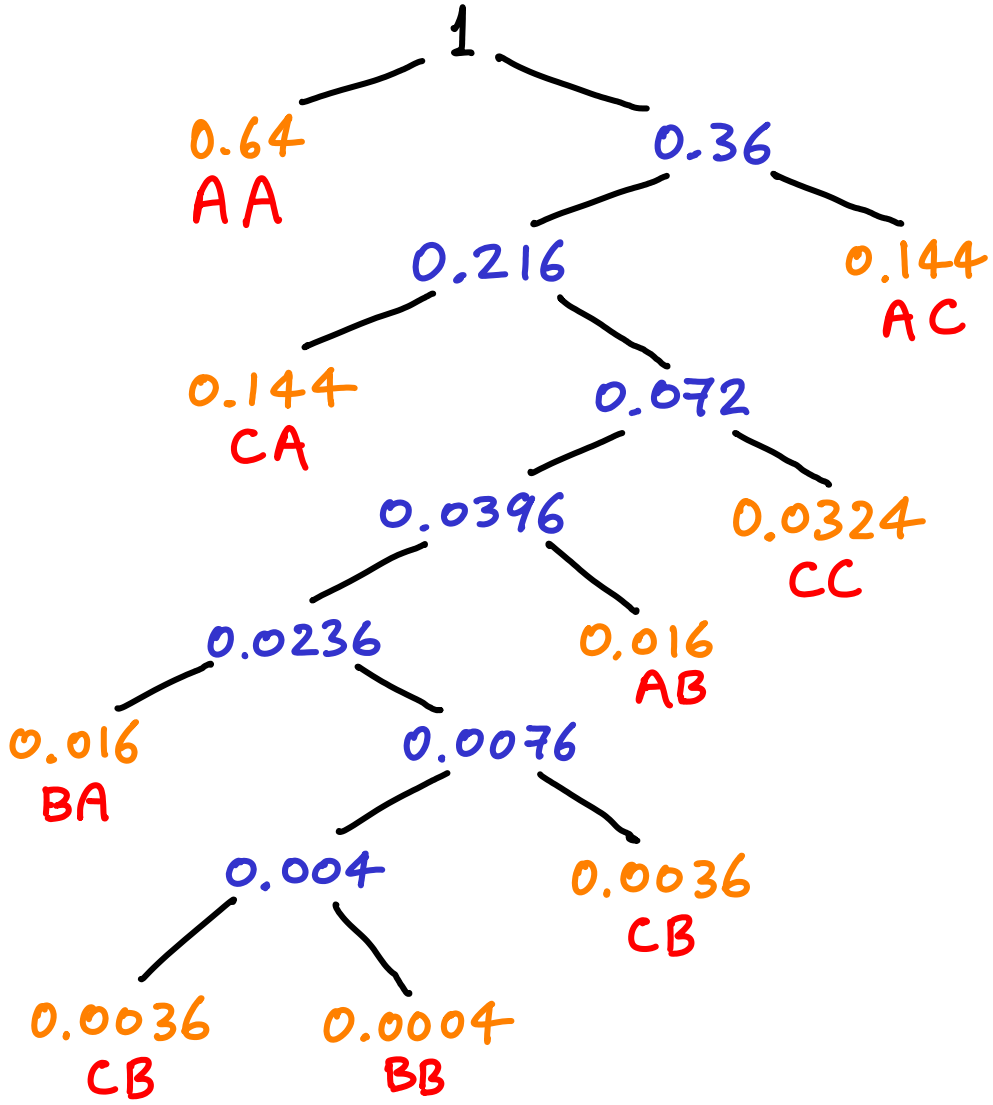


A	B	C
0	11	10
0.8	0.02	0.18

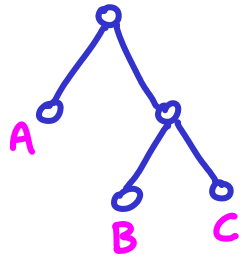


$0.8 + 0.04 + 0.36 = 1.2$

AA	0.64	⊙
AB	0.016	•
AC	0.144	•
BA	0.016	•
BB	0.0004	•
BC	0.0036	•
CA	0.144	•
CB	0.0036	•
CC	0.0324	•



A	B	C
0	11	10
0.8	0.02	0.18



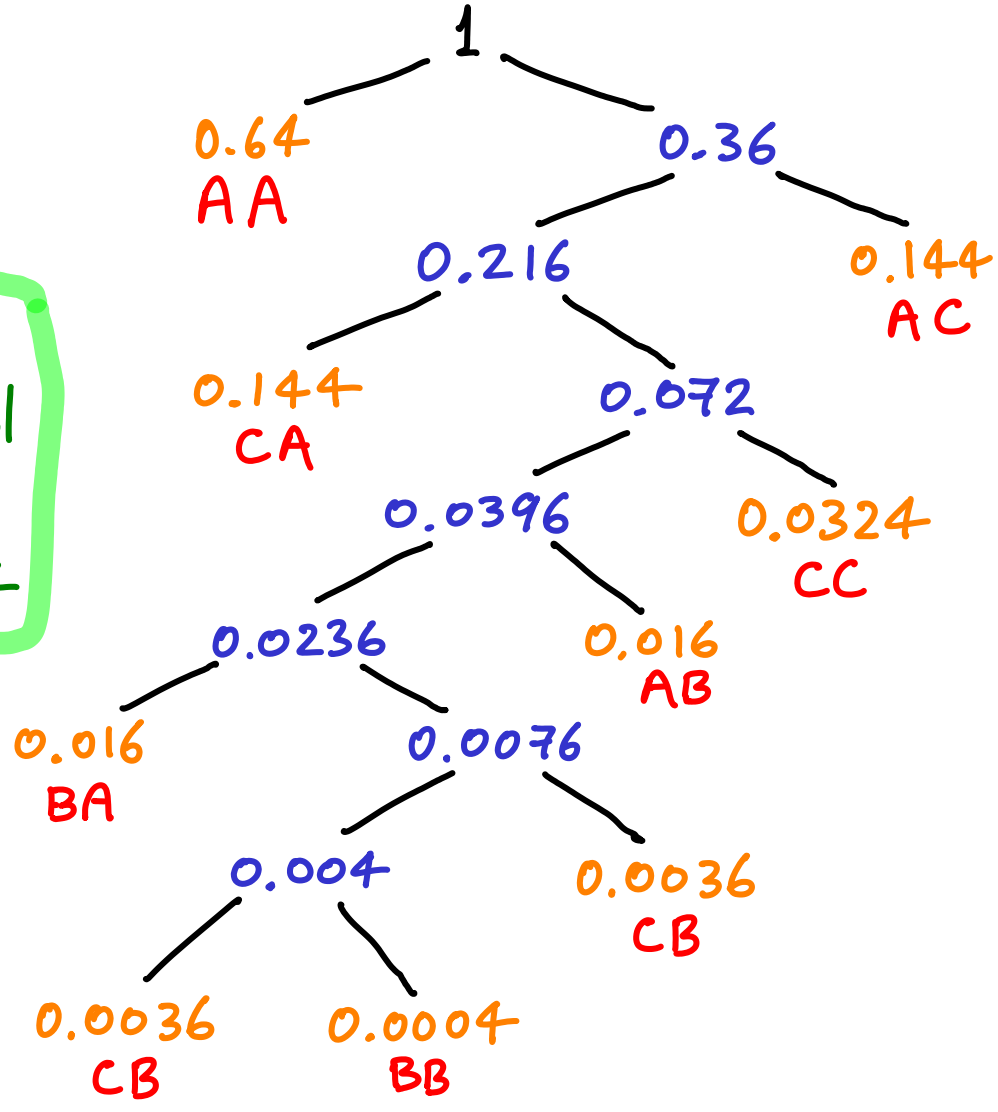
$0.8 + 0.04 + 0.36 = 1.2$

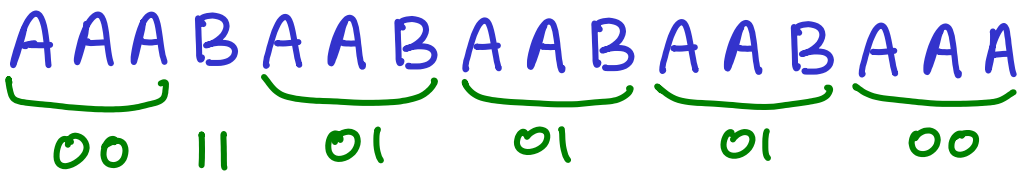
AA	0.64	0
AB	0.016	1 0 1 0 1
AC	0.144	1 1
BA	0.016	1 0 1 0 0 0
BB	0.0004	1 0 1 0 0 1 0 1
BC	0.0036	1 0 1 0 0 1 1
CA	0.144	1 0 0
CB	0.0036	1 0 1 0 0 1 0 0
CC	0.0324	1 0 1 1

1.7228
bits / symbol

↓

/2 = 0.8614





AAA	00
AAB	01
AB	10
B	11

2-bit Tunstall code (fixed length)

errors do not propagate (Huffman: one incorrect bit can ruin everything)