CS 6351 DATA COMPRESSION

THIS LECTURE:
LOSSLESS COMPRESSION PART I

Instructor: Abdou Youssef

OBJECTIVES OF THIS LECTURE

By the end of this lecture, you will be able to:

- Apply Huffman coding for lossless compression, and explain where Huffman coding is applicable
- Implement Huffman coding/decoding using Huffman trees and Huffman tables
- Evaluate the bitrate of Huffman codes
- Explain the prefix property and prove that Huffman coding satisfies that property
- Adapt Huffman coding to block-Huffman coding
- Apply Run-length Encoding (RLE), address its implementation issues, and explain where RLE applies
- Describe and apply Golomb and differential Golomb coding, and where to apply each
- Explicate the connection between Golomb, differential Golomb, and RLE

OUTLINE

- Huffman coding and Huffman trees
- Huffman decoding and the prefix property
- Huffman code bitrate
- Run-length encoding (RLE), binary RLE, and implementation issues
- Golomb coding and decoding of binary data
- Computation of the optimal Golomb parameter
- Differential Golomb and where it applies

HUFFMAN CODING
-- PRELIMINARIES --

- Assume we have a memoryless source where the alphabet is $\{a_1, a_2, \dots, a_n\}$
- Assume that we know the probability of the occurrence of each alphabet symbol a_i: $p_i = \Pr[a_i]$
- Huffman coding finds a distinct (binary) codeword for each alphabet symbol (the algorithm will be explained shortly)
- Once we have the codewords, coding an input sequence of symbols is simply the following process:
  1. Replace each symbol in the sequence by its codeword
  2. Concatenate those codewords, getting the coded bitstream of the input

> - Input can be a text file, a text message, etc.
> - The symbols are from the above alphabet

HUFFMAN CODING
-- THE CODING ALGORITHM--

Input: alphabet $\{a_1, a_2, \dots, a_n\}$ and symbol probabilities $\{p_1, p_2, \dots, p_n\}$

Output: the codewords of the alphabet symbols

Method: (a Greedy method for creating a Huffman tree as follows)

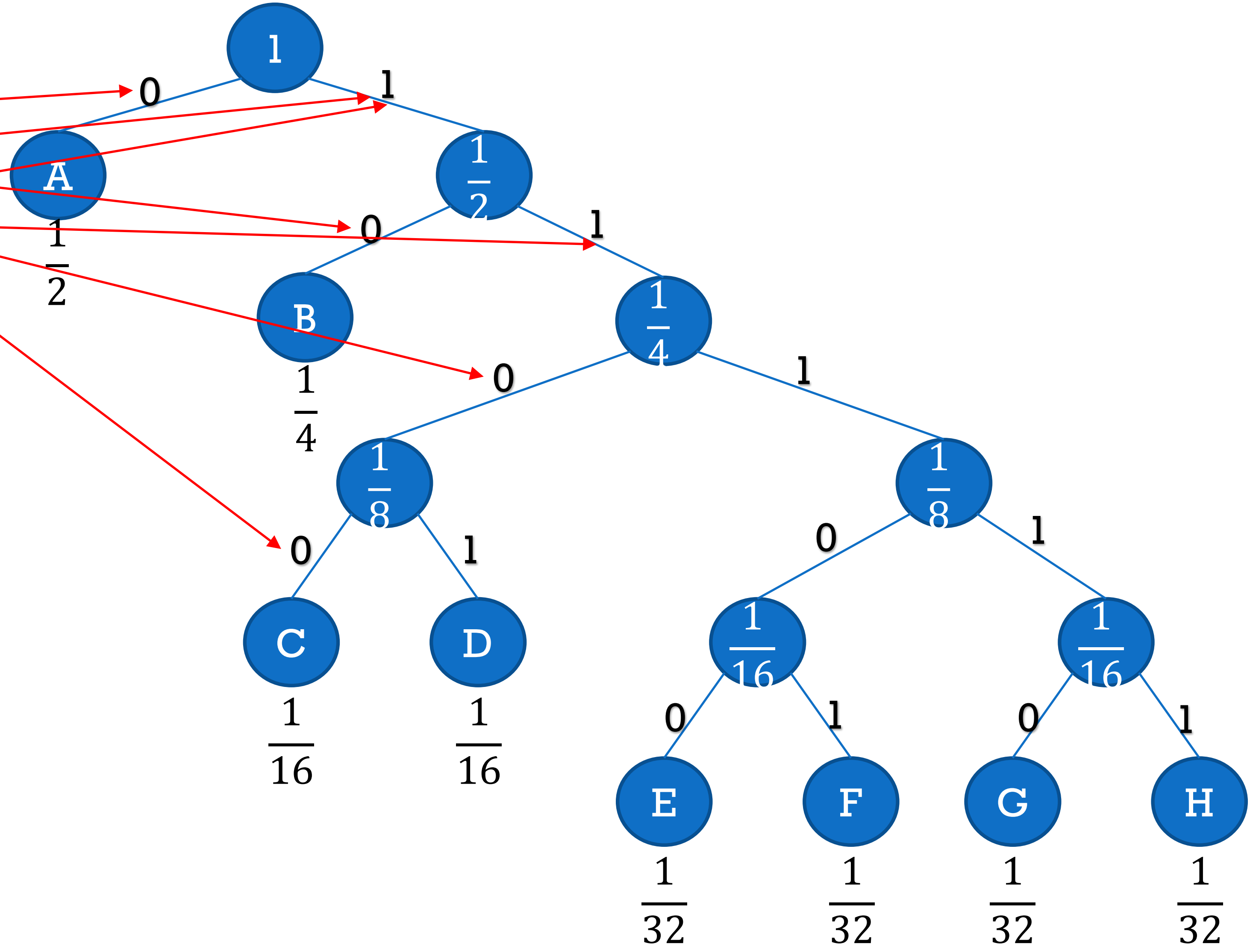
1. Create a node for each symbol a_i // these nodes will be the leaves
2. **While** (there are two or more uncombined nodes) **do**
   - Select 2 uncombined nodes a and b of <u>minimum</u> probabilities
   - Create a new node c of prob $P_a + P_b$, and make a and b children of c
3. Label the tree edges: left edges with 0, right edges with 1
4. The codeword of each alphabet symbol a_i (a leaf) is the binary string that labels the path from the root down to leaf a_i

HUFFMAN CODING

-- ILLUSTRATION OF THE CODING ALGORITHM --

Alphabet={A,B,C,D,E,F,G,H}, $P_A = \frac{1}{2}$, $P_B = \frac{1}{4}$, $P_C = \frac{1}{16}$, $P_D = \frac{1}{16}$, $P_E = P_F = P_G = P_H = \frac{1}{32}$

Codewords
- A: 0
- B: 10
- C: 1100
- D: 1101
- E: 11100
- F: 11101
- G: 11110
- H: 111111



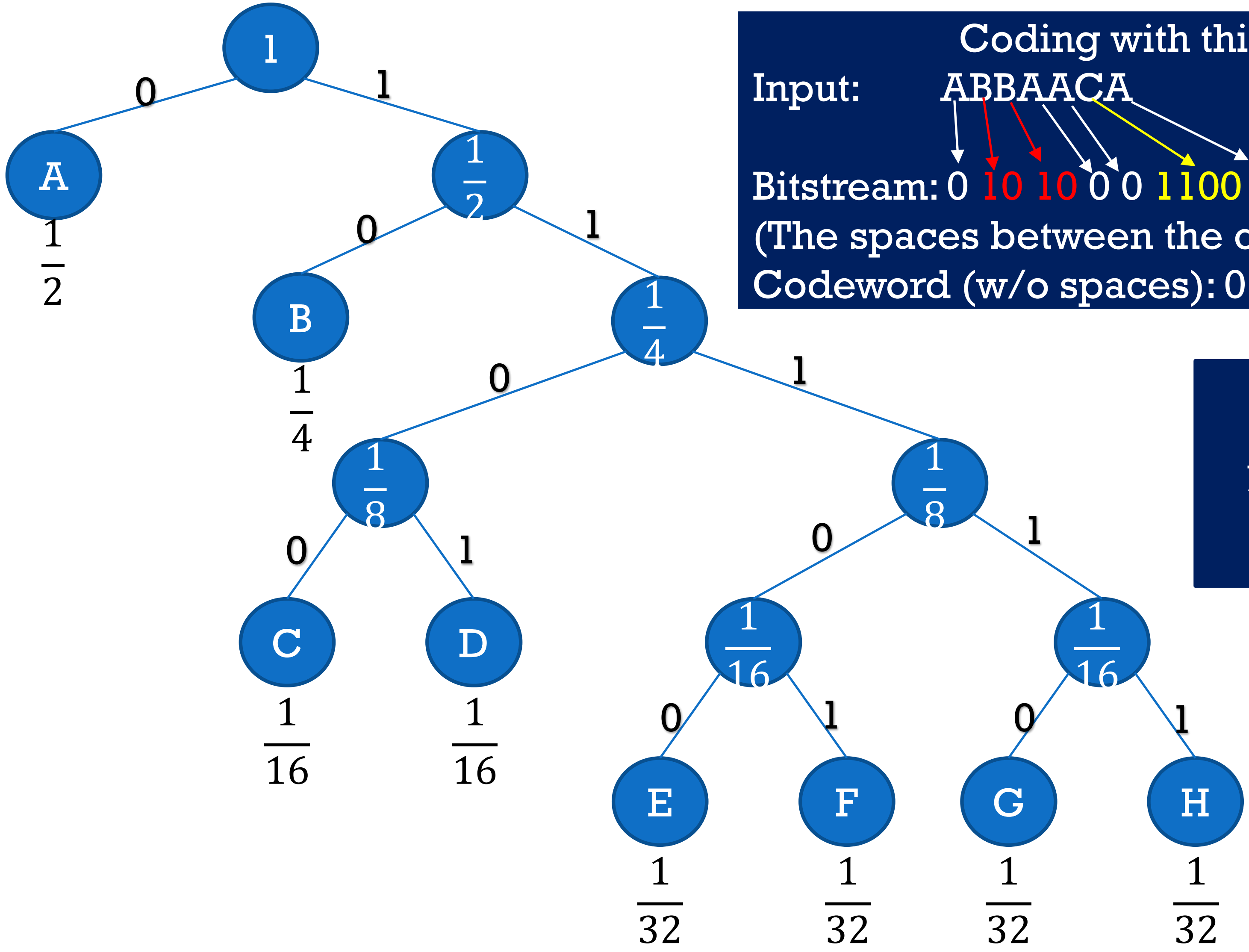
HUFFMAN CODING

-- ILLUSTRATION OF CODING SOME INPUT--

Alphabet={A,B,C,D,E,F,G,H}, $P_A = \frac{1}{2}$, $P_B = \frac{1}{4}$, $P_C = \frac{1}{16}$, $P_D = \frac{1}{16}$, $P_E = P_F = P_G = P_H = \frac{1}{32}$

Codewords

A: 0
B: 10
C: 1100
D: 1101
E: 11100
F: 11101
G: 11110
H: 111111



Coding with this Huffman code

Input: ABBAACA

Bitstream: 0 10 10 0 0 1100 0

(The spaces between the codewords are for clarity)

Codeword (w/o spaces): 010100011000

Bitrate of this example:

$$\frac{length(bitstream)}{num.of\ symbols\ in\ input} = \frac{12}{7} = 1.714 \text{ bits/symbol}$$

HUFFMAN CODING
-- CODING PERFORMANCE --

- In lossless compression of memoryless sources
  - If the coder works by computing a codeword for each alphabet symbol
  - Then, we can compute a **coder bitrate**, independent of any actual input data
- Notation:
  - For any binary string s, denote by **|s|** the number of bits in s
  - Let $\mathbf{codeword(a_i)}$ denote the codeword for symbol a_i
- Coder bitrate: $\mathrm{BR} = \sum_{i=1}^{n} p_i \,|\mathrm{codeword}(a_i)|$
- Source Entropy: $H = -\sum_{i=1}^{n} p_i \log p_i$

Example: the Huffman coder just presented

- The codewords and the probabilities are

Codewords	Length	Probabilities
A: 0	1	1/2
B: 10	2	1/4
C: 1100	4	1/16
D: 1101	4	1/16
E: 11100	5	1/32
F: 11101	5	1/32
G: 11110	5	1/32
H: 11111	5	1/32

- BR=$1 \times \frac{1}{2} + 2 \times \frac{1}{4} + 4 \times \frac{1}{16} + 4 \times \frac{1}{16} + 5 \times \frac{1}{32} + 5 \times \frac{1}{32} + 5 \times \frac{1}{32} + 5 \times \frac{1}{32} = \frac{69}{32} = 2.125\ bits/symbol$
- Entropy: H=$-\left(\frac{1}{2}\log\frac{1}{2} + \frac{1}{4}\log\frac{1}{4} + \frac{1}{16}\log\frac{1}{16} + \frac{1}{16}\log\frac{1}{16} + \right.$

HUFFMAN CODING
-- OBSERVATIONS (1/2) --

- Observation 1 (about the example):
  - **BR=H**, i.e., the coder bitrate achieved the entropy, the best possible
  - Does that mean Huffman coding always achieves the entropy?
  - No. See the theorem next
- **Theorem**: If all the probabilities (in a memoryless source) are powers of ½ , then Huffman achieves BR=H. The further away the probabilities are from powers of ½ , the further away BR is from entropy H (i.e., BR>H).
- We will not prove that theorem, but it is important that you keep it in mind

HUFFMAN CODING
-- OBSERVATIONS (2/2) --

- Observation 2 (about the data example):
  - When applying the example Huffman coder on input ABBAACA, the data bitrate was 1.714 bits/symbol, which is less the entropy H=2.125
  - This seems to violate that no lossless code can produce a bitrate smaller than the entropy, and lead us to the following important questions
- Questions:
  1. Do we have a contradiction?
  2. If not, how do you reconcile the two?
  3. What kind of input data yields a data bitrate smaller than the entropy of the source?

HUFFMAN DECODING

Input: A coded bitstream $b_1 b_2 \dots b_N$ (and we have the Huffman tree)

Output: The reconstructed data (will be identical to the original data)

Method:

1. Initialize: i=1, and let node pointer **ptr** point at the tree root;
2. While (i<N) do
   - If $b_i == 0$, let ptr go to left child, else go to the right child
   - If ptr is pointing to a leaf node,
     - Append to the output the symbol corresponding to that leaf;
     - Reset ptr back to the root
     - i=i+1;
   - Else: i=i+1;

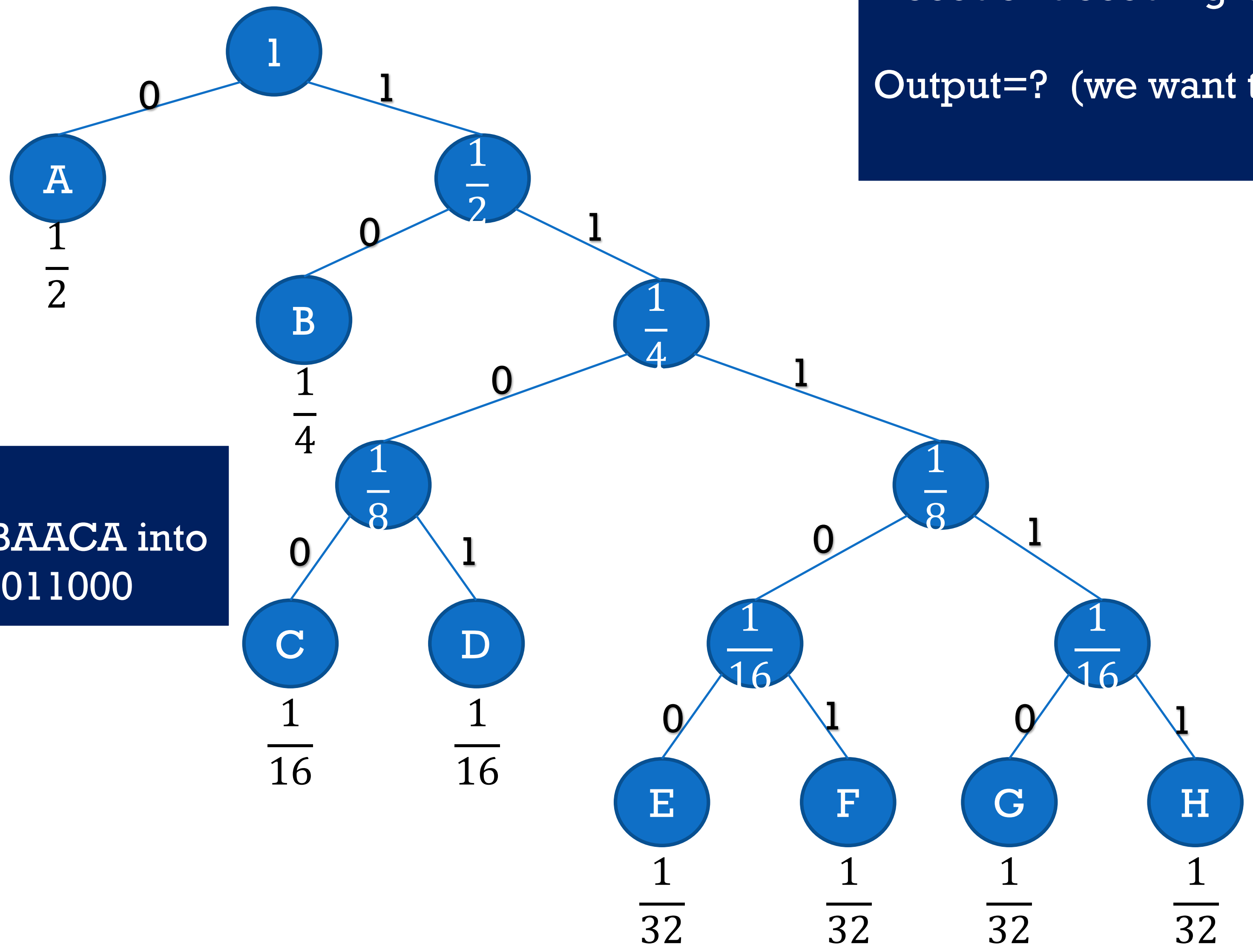
HUFFMAN DECODING
-- ILLUSTRATION: DECODING 010100011000 (1/21) --

Codewords
A: 0
B: 10
C: 1100
D: 1101
E: 11100
F: 11101
G: 11110
H: 111111

Decoder decoding bitstream 010100011000

Output=? (we want to get back ABBAACA)

Recall:
Coder coded ABBAACA into
Bitstream: 010100011000

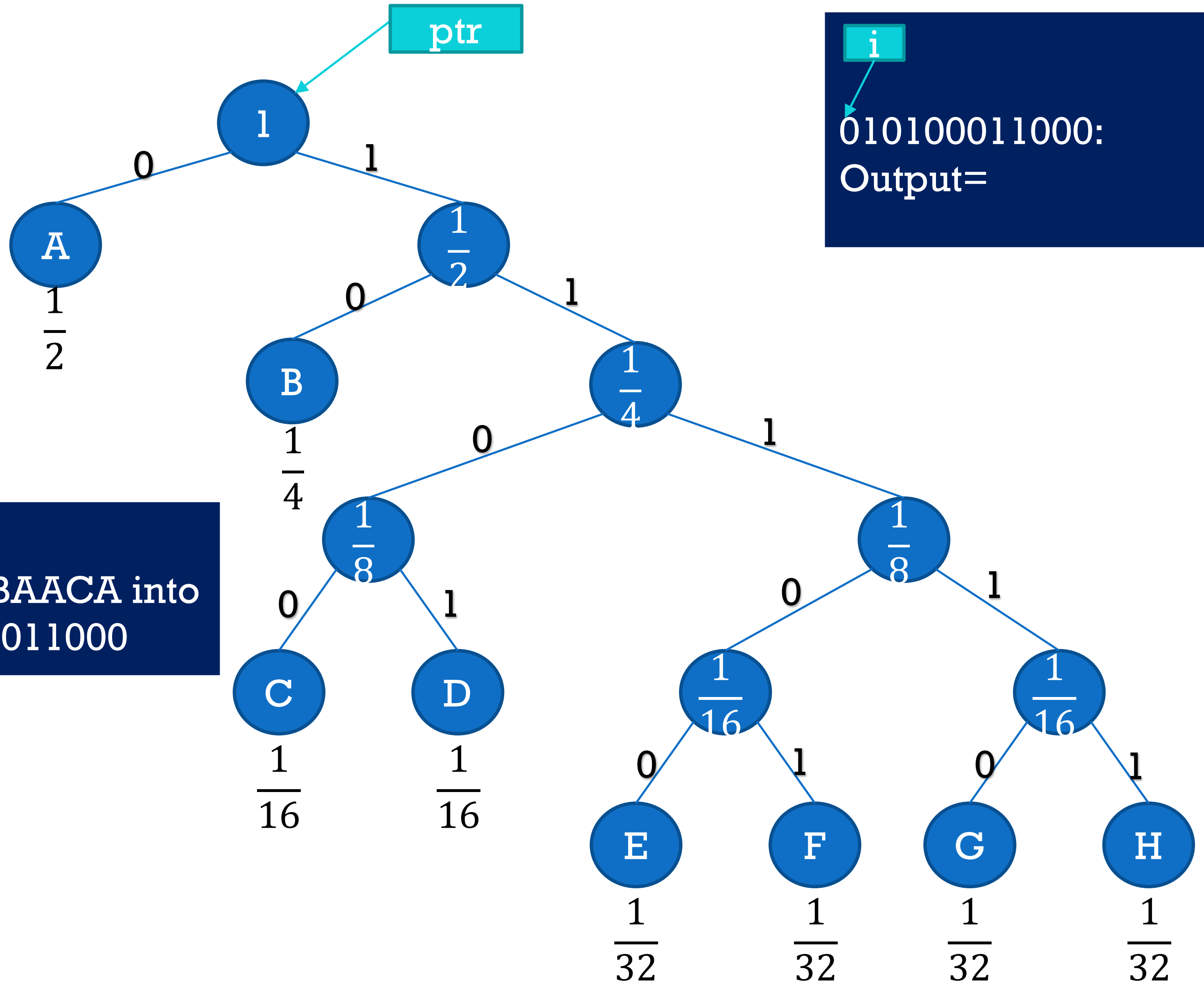


HUFFMAN DECODING

-- ILLUSTRATION: DECODING 010100011000 (2/21) --

Codewords
A: 0
B: 10
C: 1100
D: 1101
E: 11100
F: 11101
G: 11110
H: 111111

Recall:
Coder coded ABBAACA into
Bitstream: 010100011000



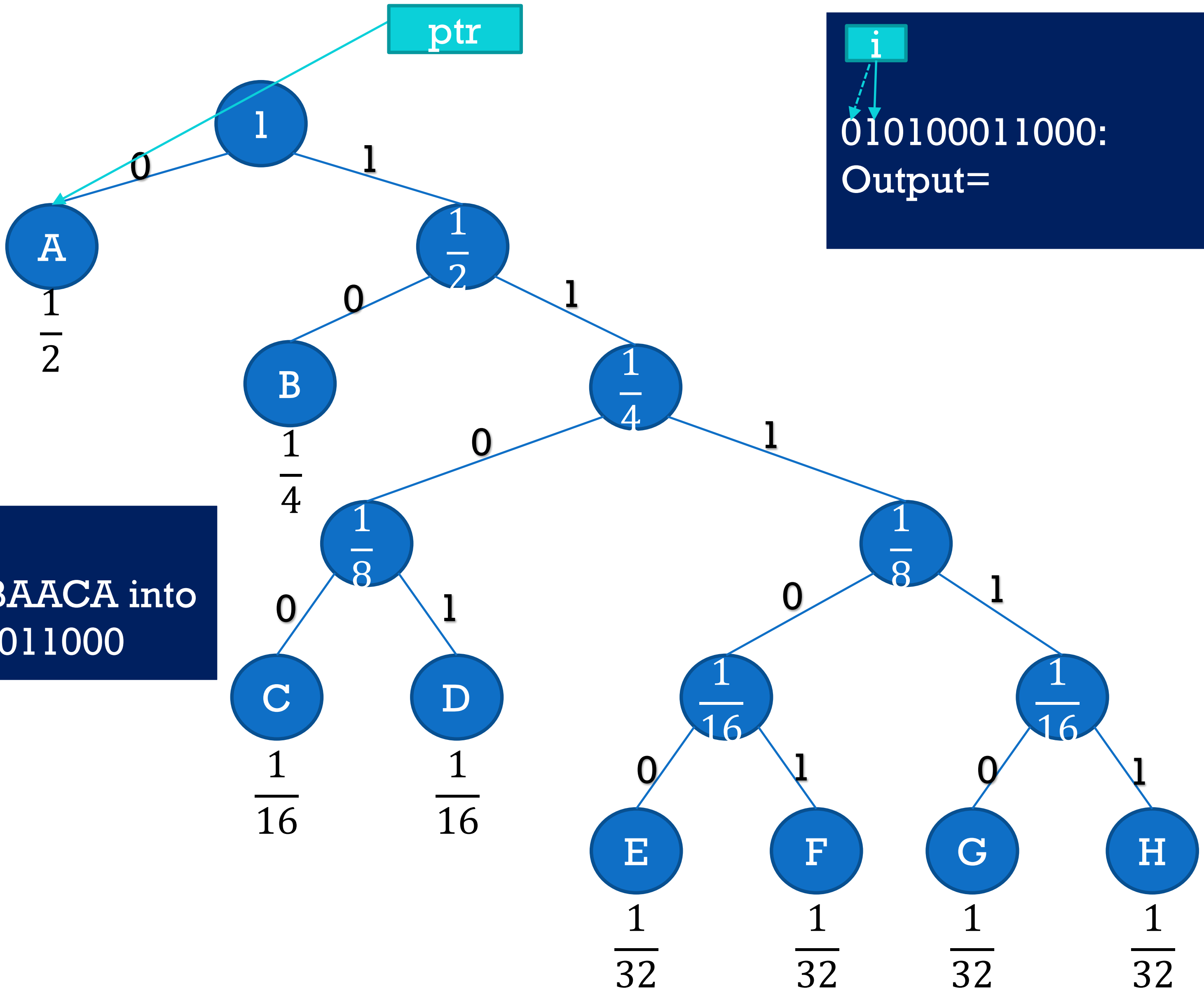
i
010100011000:
Output=

Initialize:
- i points to 1st bit
- ptr points to the root

HUFFMAN DECODING

-- ILLUSTRATION: DECODING 010100011000 (3/21) --

Codewords
A: 0
B: 10
C: 1100
D: 1101
E: 11100
F: 11101
G: 11110
H: 111111



Recall:
Coder coded ABBAACA into
Bitstream: 010100011000

i

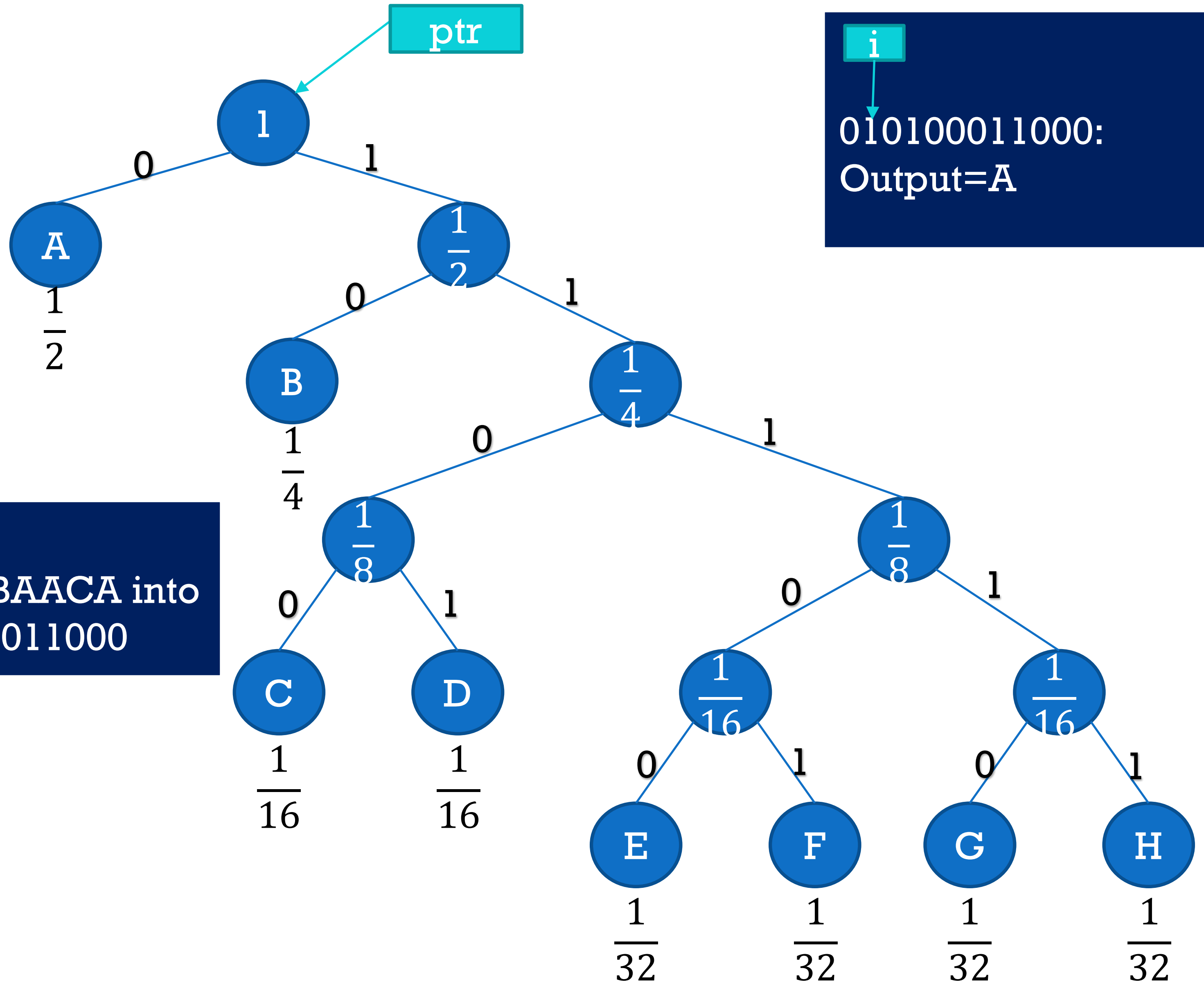
010100011000:
Output=

Bit i is 0, ptr goes left, and i advances

HUFFMAN DECODING
-- ILLUSTRATION: DECODING 010100011000 (4/21) --

Codewords
A: 0
B: 10
C: 1100
D: 1101
E: 11100
F: 11101
G: 11110
H: 111111

Recall:
Coder coded ABBAACA into
Bitstream: 010100011000



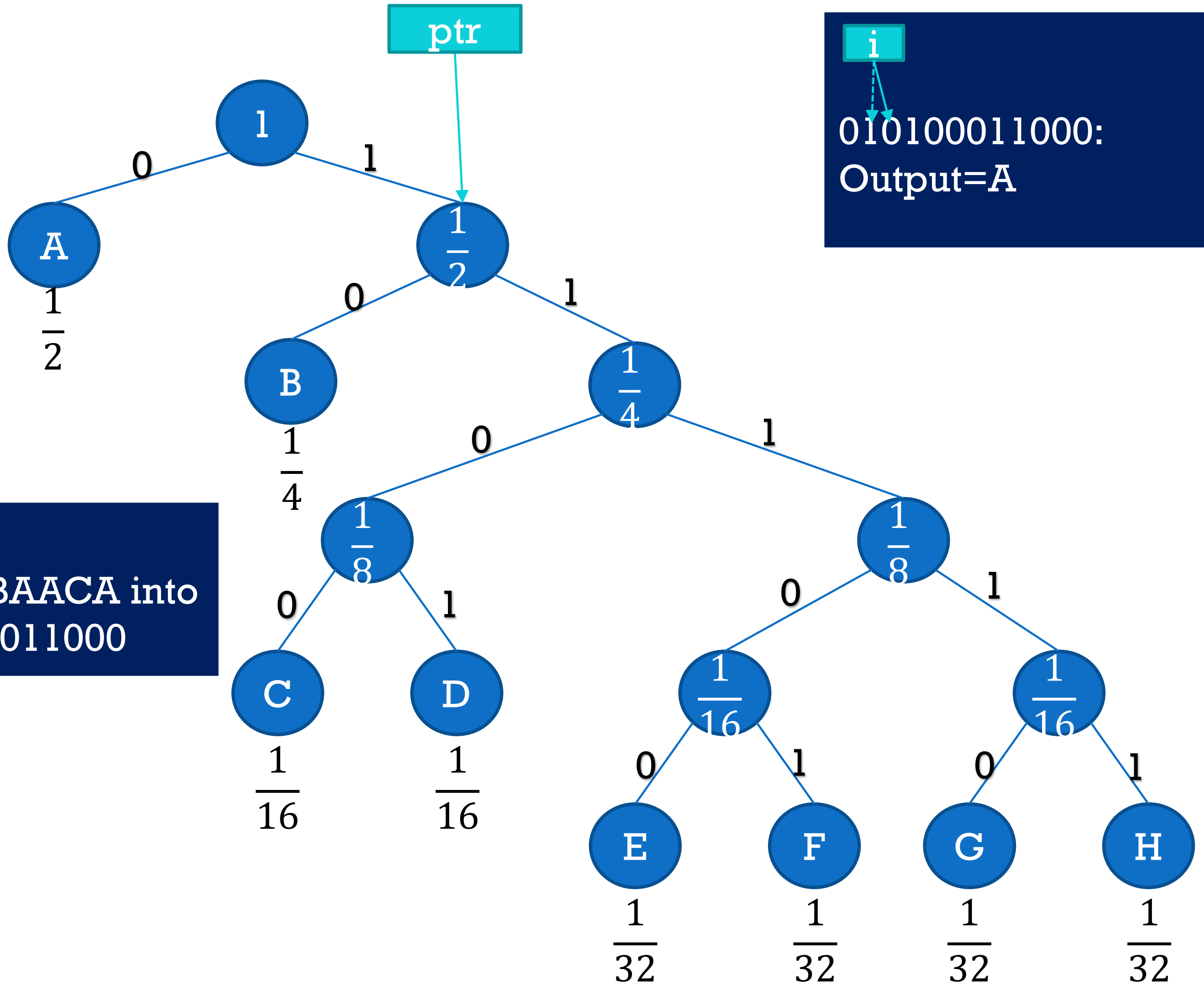
i
010100011000:
Output=A

Found a leaf (A), append A to output, and reset ptr to the root ;
Next: decode bit i

HUFFMAN DECODING

-- ILLUSTRATION: DECODING 010100011000 (5/21) --

Codewords
A: 0
B: 10
C: 1100
D: 1101
E: 11100
F: 11101
G: 11110
H: 111111



i

010100011000:
Output=A

Bit i is 1, ptr goes right, and i advances

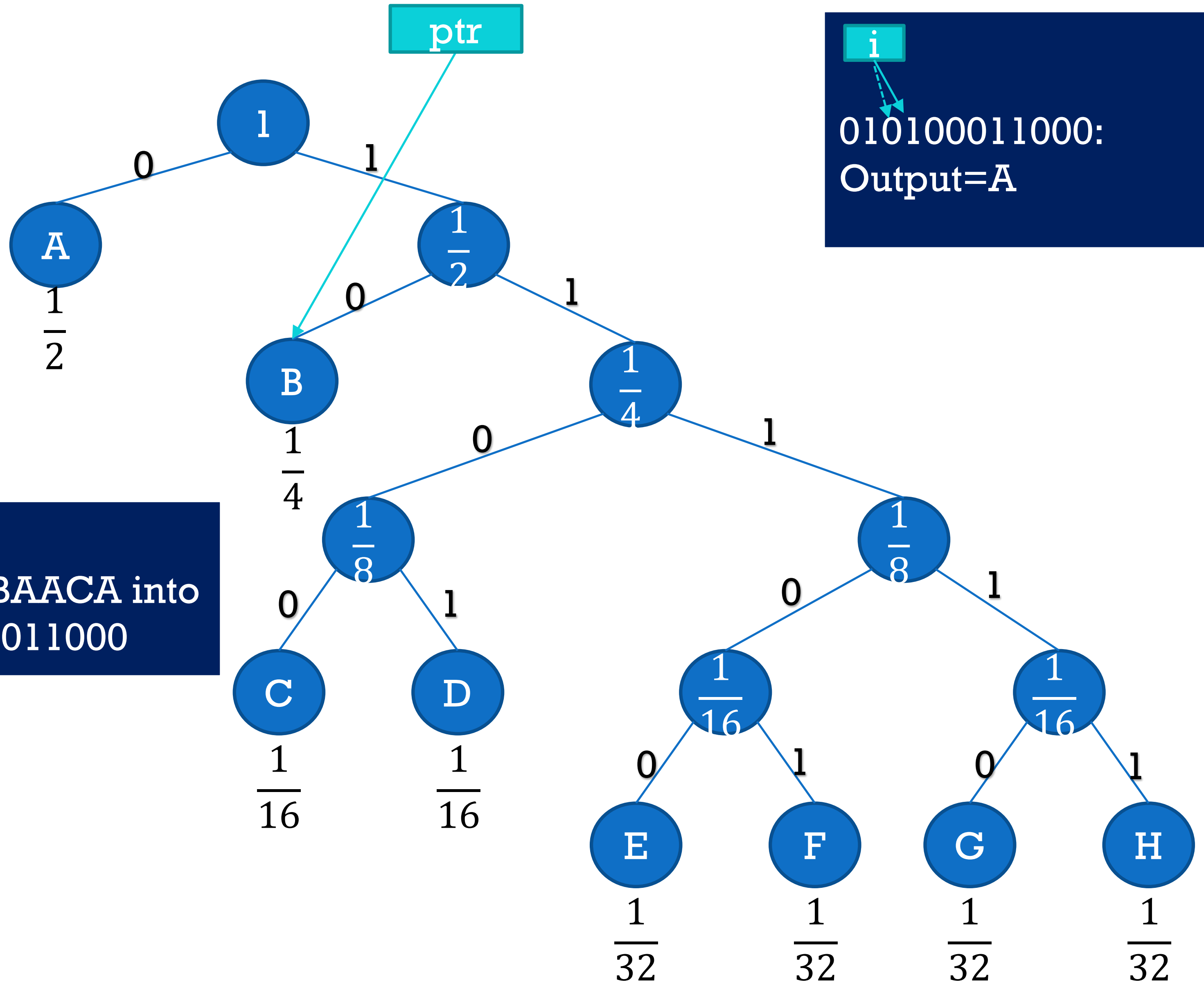
Recall:
Coder coded ABBAACA into
Bitstream: 010100011000

HUFFMAN DECODING

-- ILLUSTRATION: DECODING 010100011000 (6/21) --

Codewords
A: 0
B: 10
C: 1100
D: 1101
E: 11100
F: 11101
G: 11110
H: 111111

Recall:
Coder coded ABBAACA into
Bitstream: 010100011000



i

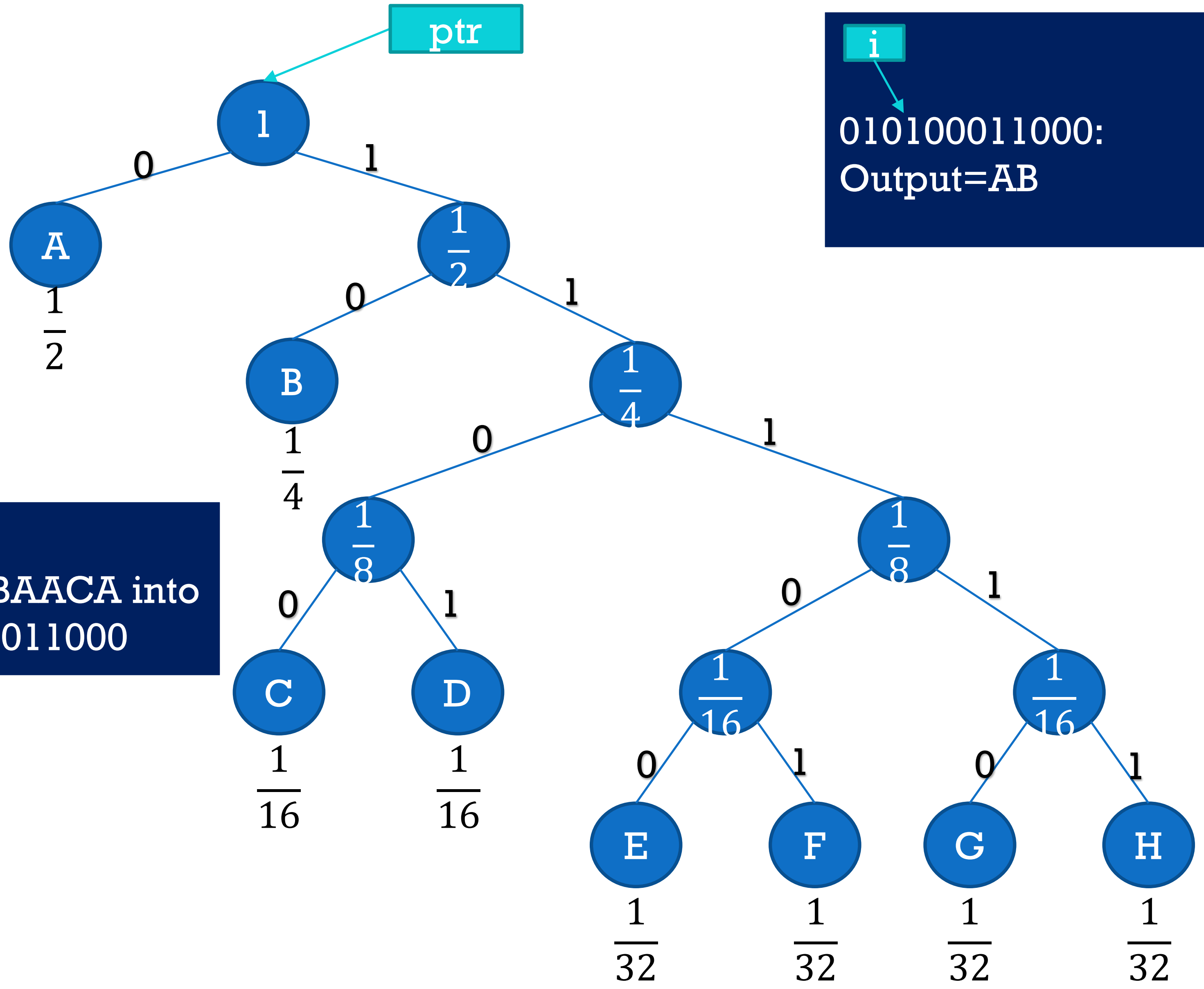
010100011000:
Output=A

Bit i is 0, ptr goes left, and i advances

HUFFMAN DECODING
-- ILLUSTRATION: DECODING 010100011000 (7/21) --

21

Codewords
A: 0
B: 10
C: 1100
D: 1101
E: 11100
F: 11101
G: 11110
H: 111111



i
010100011000:
Output=AB

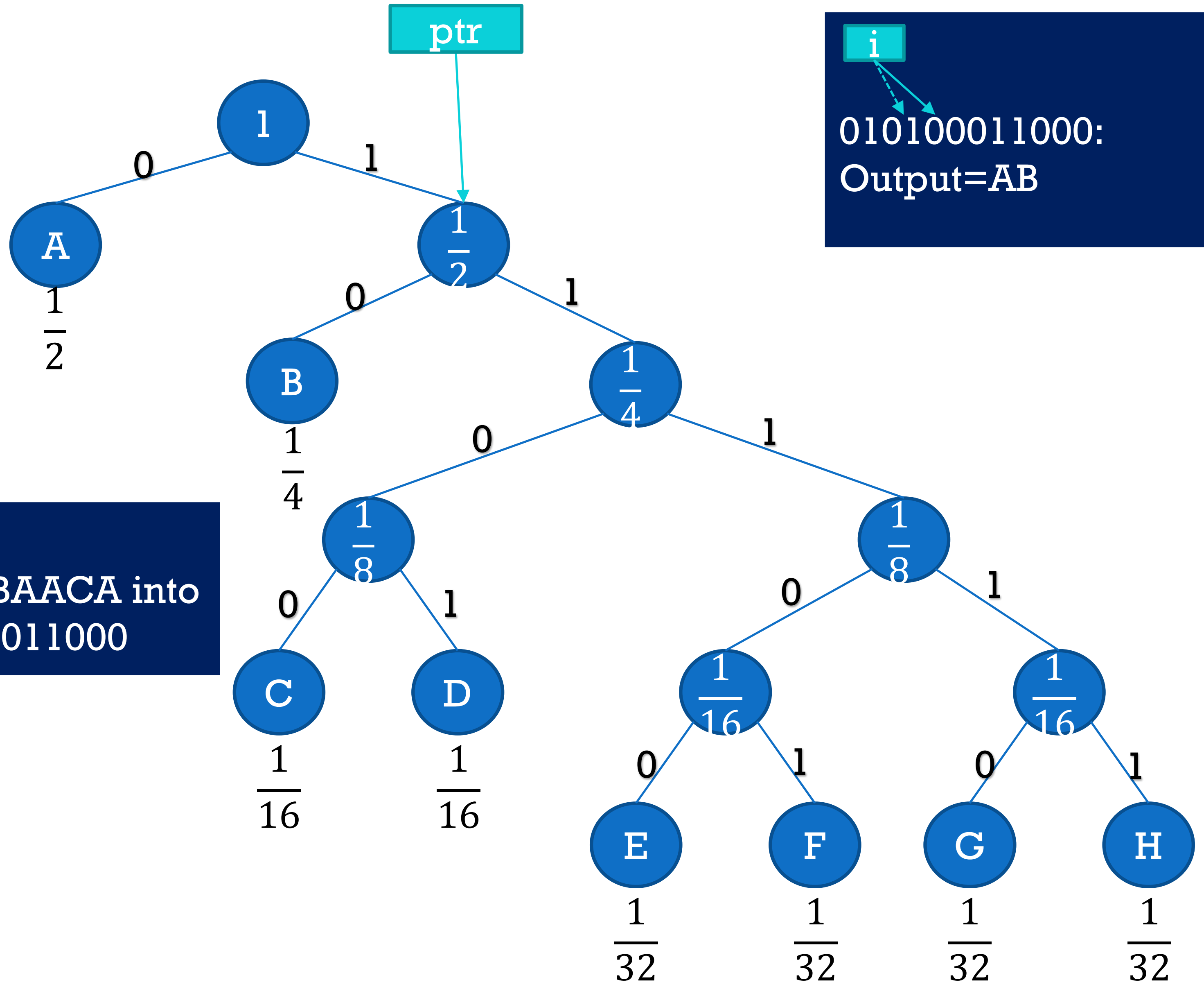
Found a leaf (B), append B to output, and reset ptr to the root ;
Next: decode bit i

Recall:
Coder coded ABBAACA into
Bitstream: 010100011000

HUFFMAN DECODING
-- ILLUSTRATION: DECODING 010100011000 (8/21) --

Codewords
A: 0
B: 10
C: 1100
D: 1101
E: 11100
F: 11101
G: 11110
H: 111111

Recall:
Coder coded ABBAACA into
Bitstream: 010100011000



i
010100011000:
Output=AB

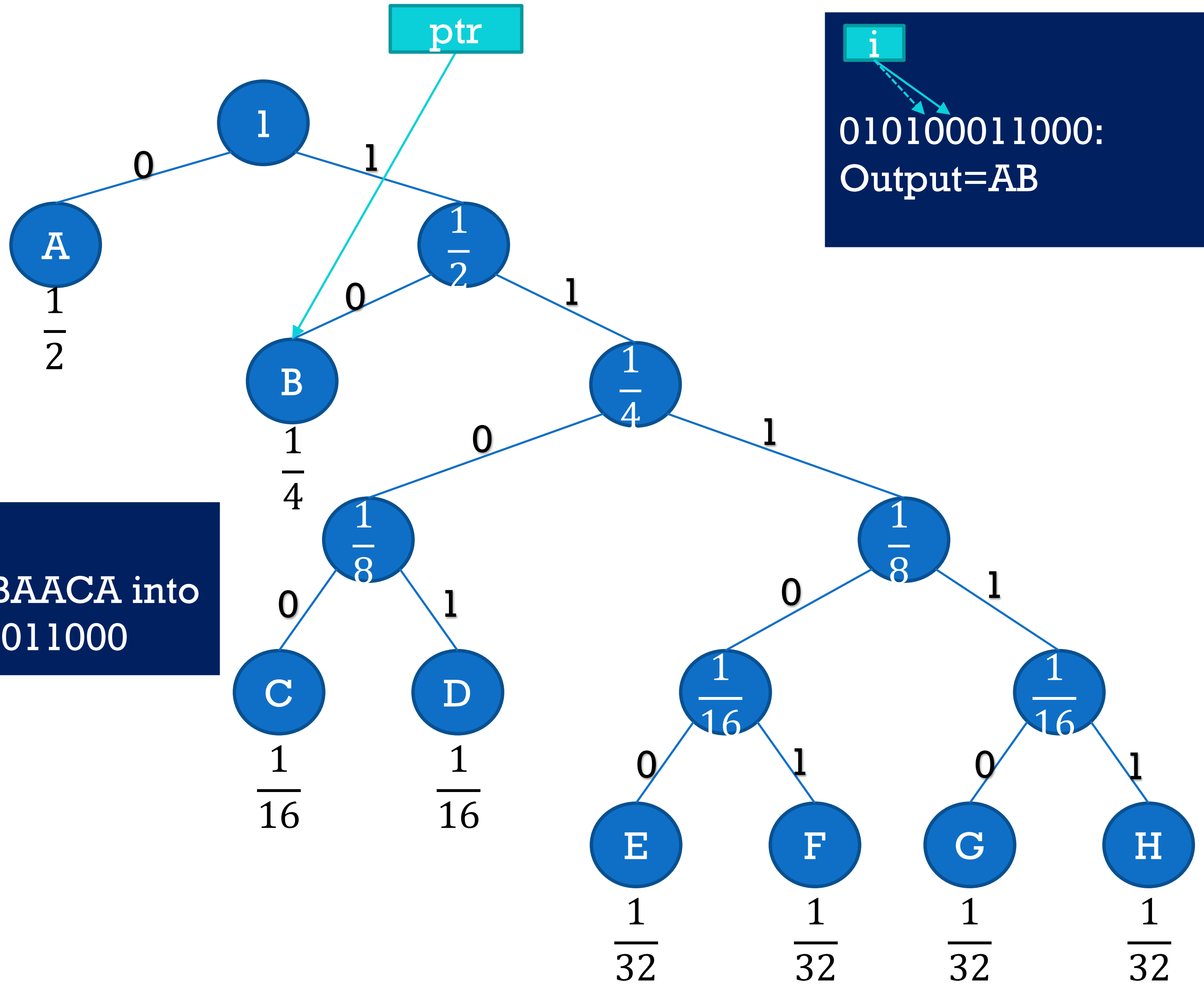
Bit i is 1, ptr goes right, and i advances

HUFFMAN DECODING

-- ILLUSTRATION: DECODING 010100011000 (9/21) --

Codewords
A: 0
B: 10
C: 1100
D: 1101
E: 11100
F: 11101
G: 11110
H: 111111

Recall:
Coder coded ABBAACA into
Bitstream: 010100011000



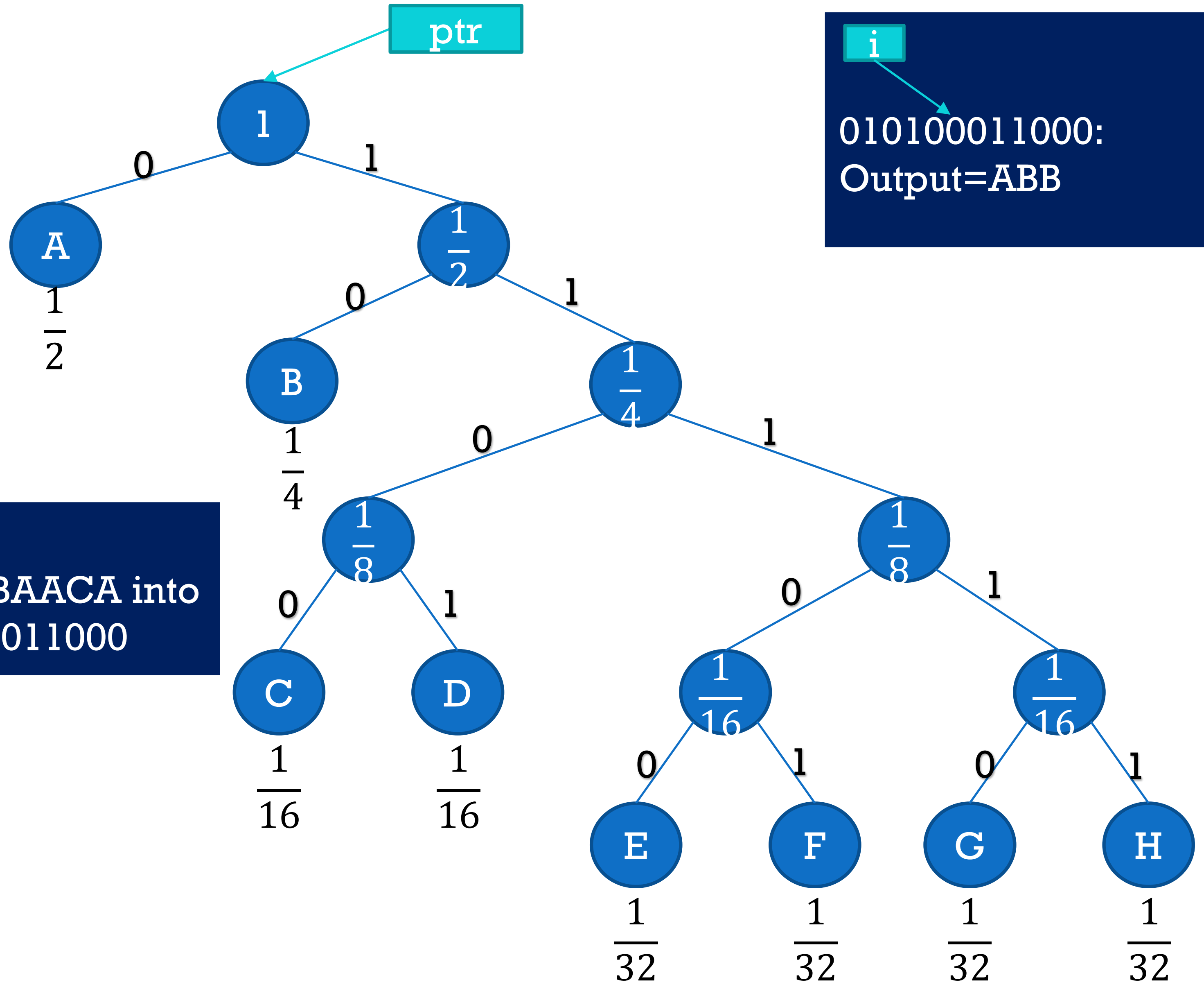
i
010100011000:
Output=AB

Bit i is 0, ptr goes left, and i advances

HUFFMAN DECODING

--ILLUSTRATION: DECODING 010100011000 (10/21) --

Codewords
A: 0
B: 10
C: 1100
D: 1101
E: 11100
F: 11101
G: 11110
H: 111111



i
010100011000:
Output=ABB

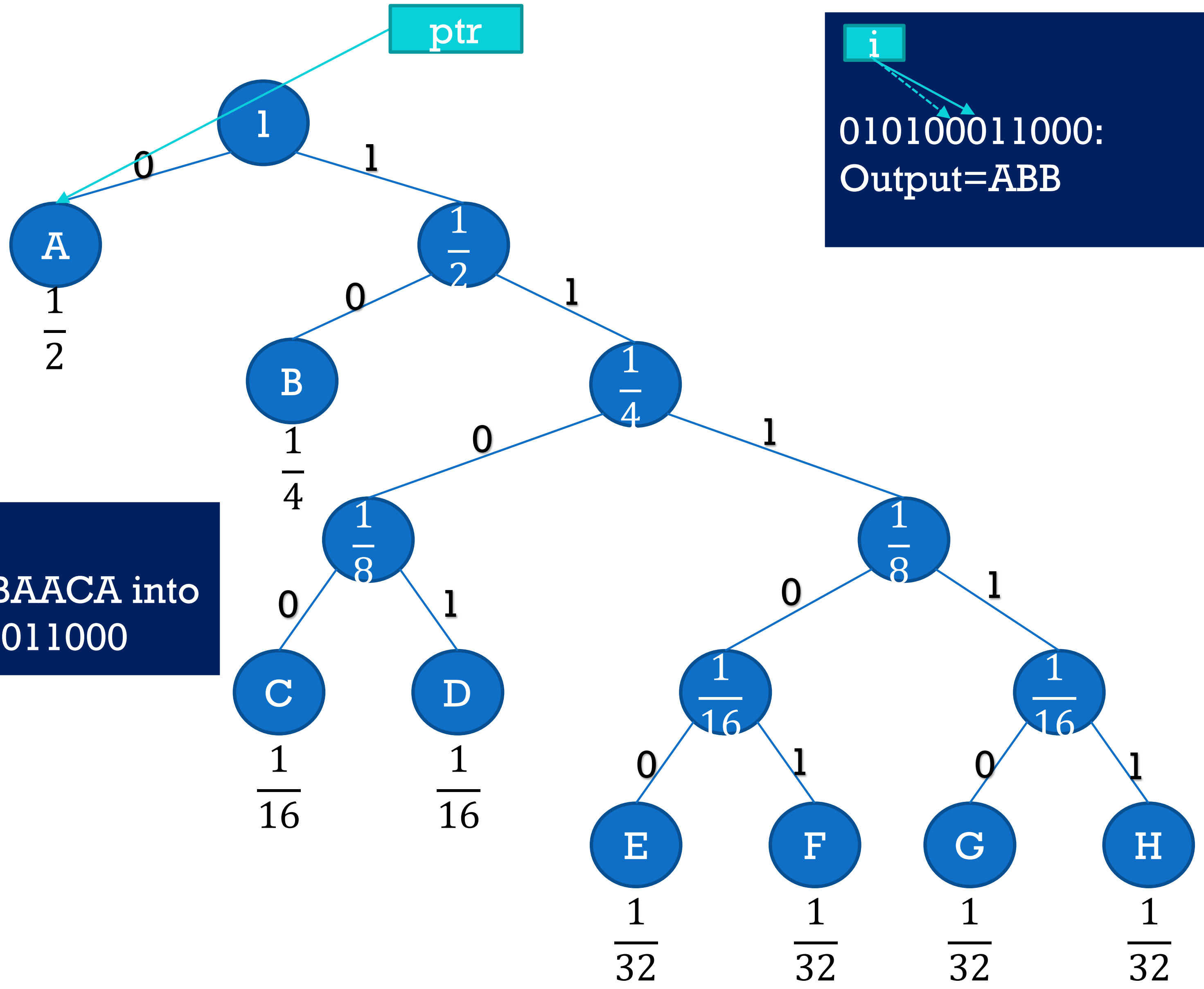
Found a leaf (B), append B to output, and reset ptr to the root ;
Next: decode bit i

Recall:
Coder coded ABBAACA into
Bitstream: 010100011000

HUFFMAN DECODING

--ILLUSTRATION: DECODING 010100011000 (11/21) --

Codewords
A: 0
B: 10
C: 1100
D: 1101
E: 11100
F: 11101
G: 11110
H: 111111



i
010100011000:
Output=ABB

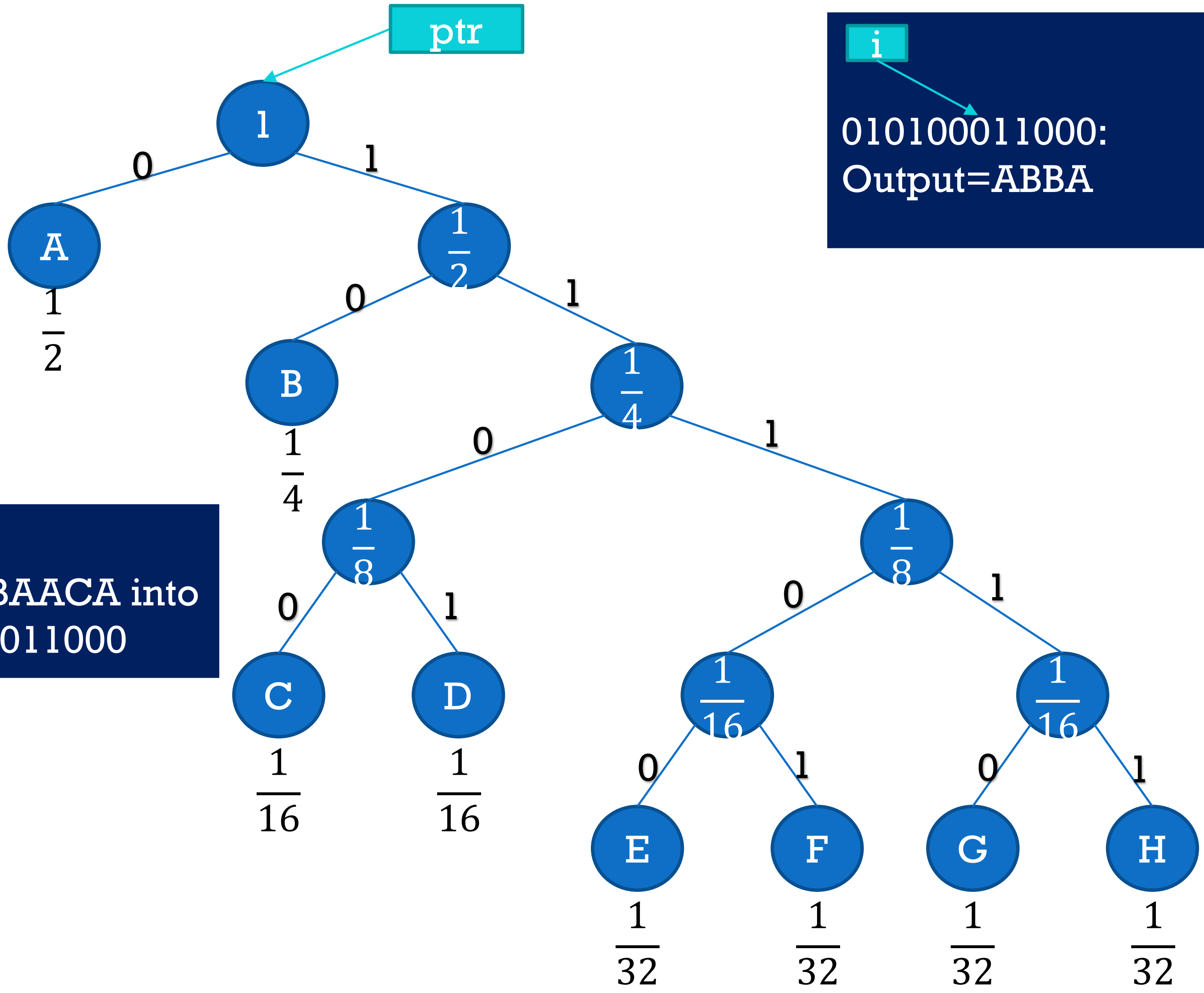
Bit i is 0, ptr goes left, and i advances

Recall:
Coder coded ABBAACA into
Bitstream: 010100011000

HUFFMAN DECODING

--ILLUSTRATION: DECODING 010100011000 (12/21) --

Codewords
A: 0
B: 10
C: 1100
D: 1101
E: 11100
F: 11101
G: 11110
H: 111111



i
010100011000:
Output=ABBA

Found a leaf (A), append A to output, and reset ptr to the root ;
Next: decode bit i

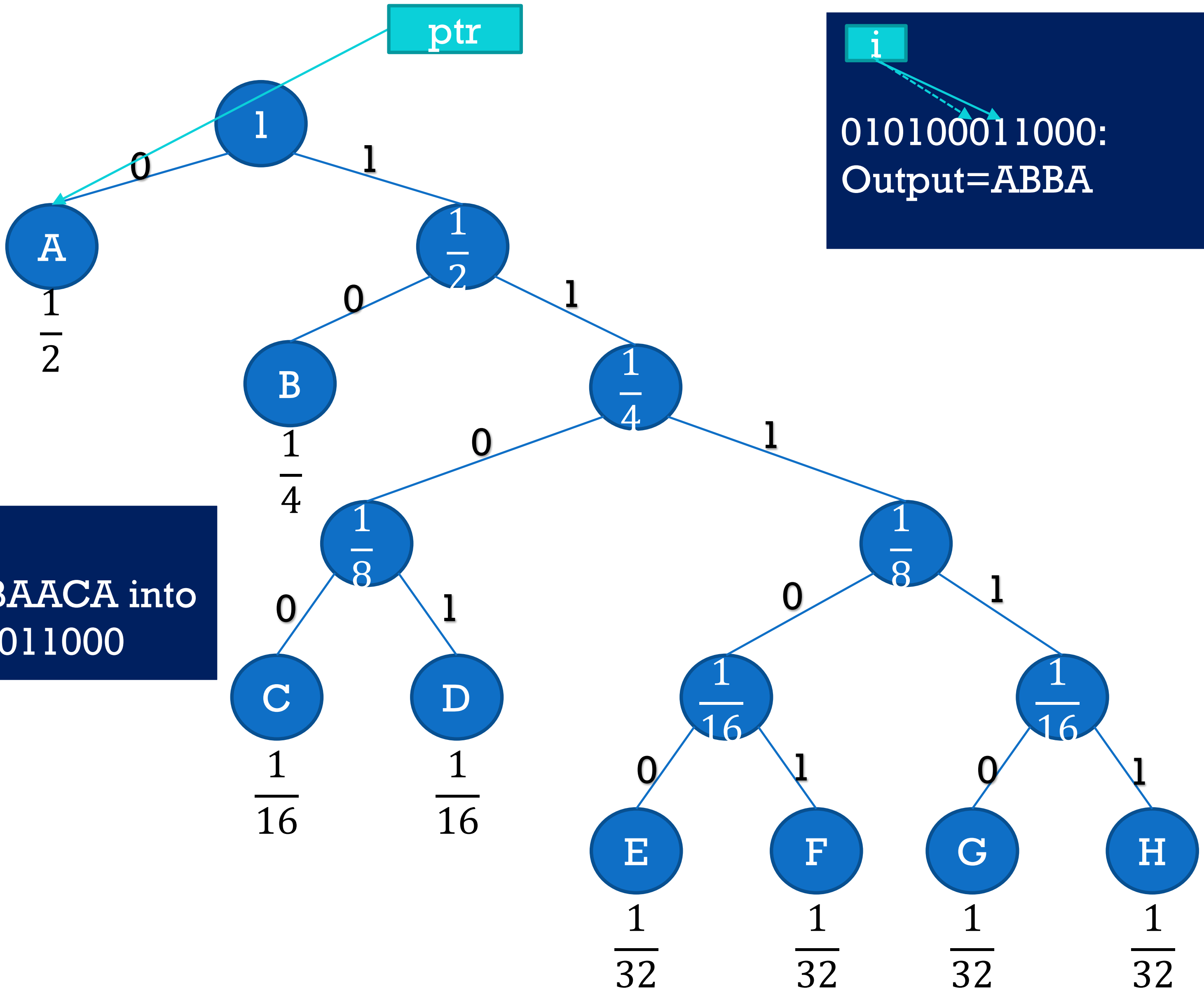
Recall:
Coder coded ABBAACA into
Bitstream: 010100011000

HUFFMAN DECODING

--ILLUSTRATION: DECODING 010100011000 (13/21) --

Codewords
A: 0
B: 10
C: 1100
D: 1101
E: 11100
F: 11101
G: 11110
H: 111111

Recall:
Coder coded ABBAACA into
Bitstream: 010100011000



i
010100011000:
Output=ABBA

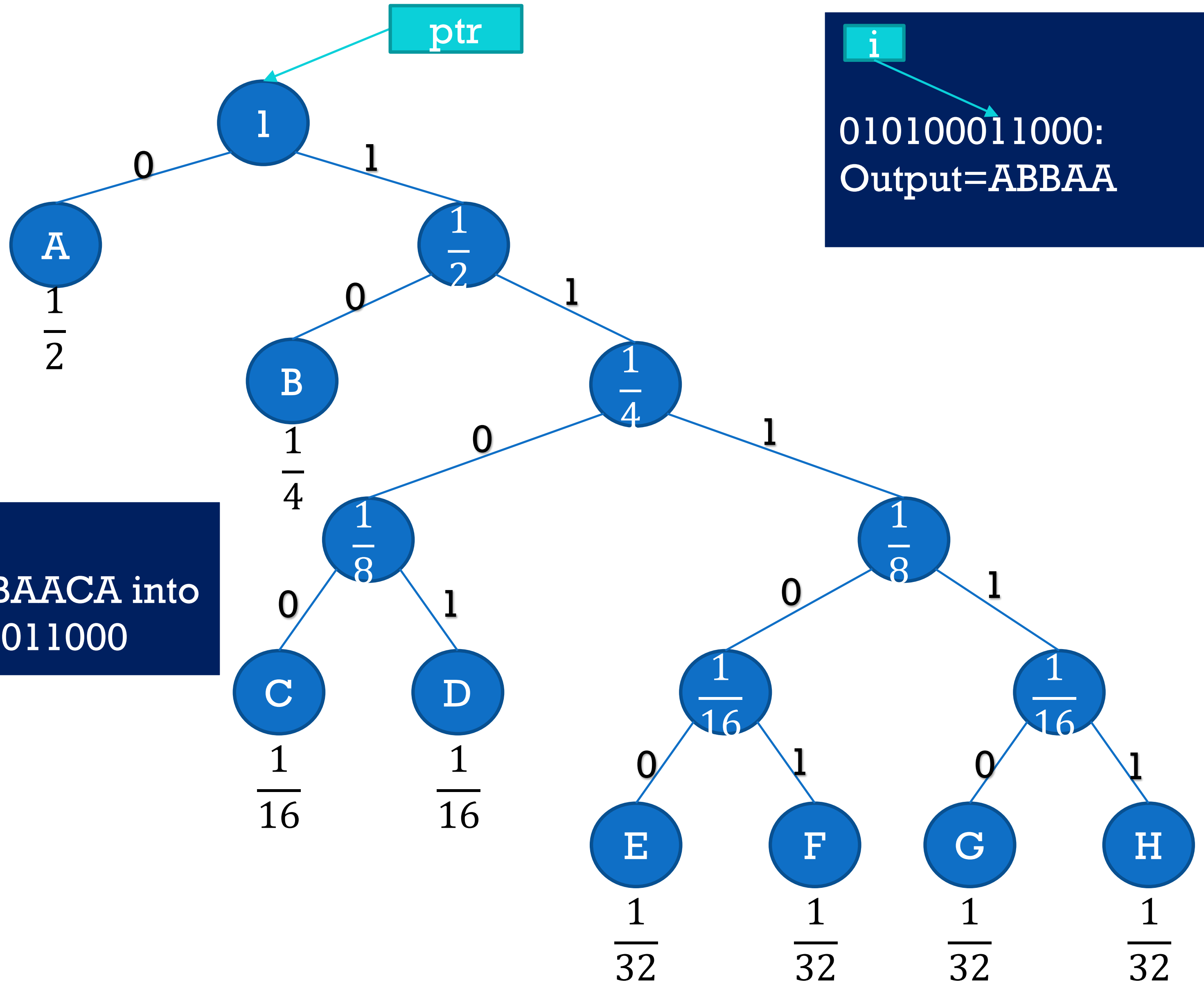
Bit i is 0, ptr goes left, and i advances

HUFFMAN DECODING

--ILLUSTRATION: DECODING 010100011000 (14/21) --

Codewords
A: 0
B: 10
C: 1100
D: 1101
E: 11100
F: 11101
G: 11110
H: 111111

Recall:
Coder coded ABBAACA into
Bitstream: 010100011000



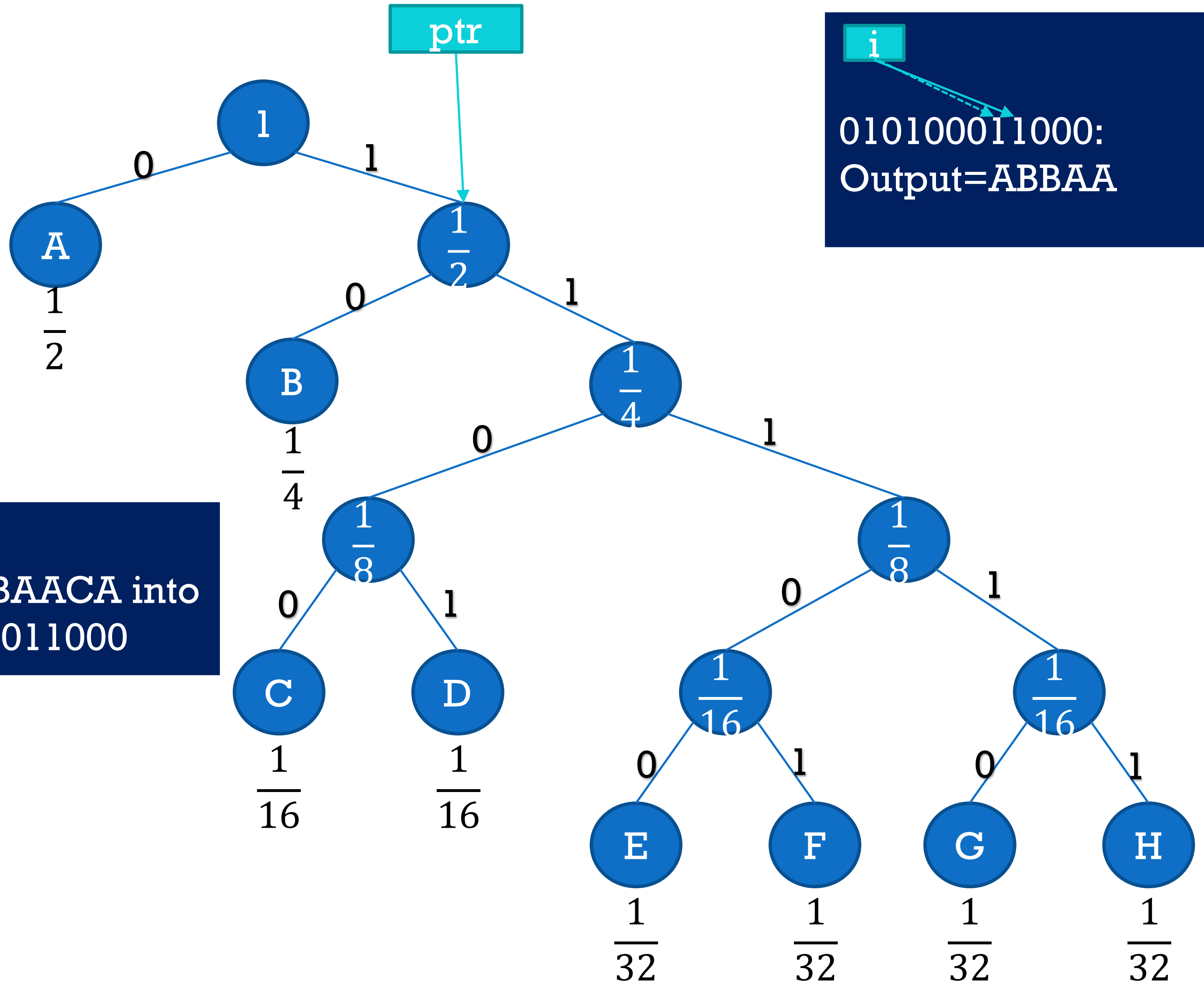
i
010100011000:
Output=ABBAA

Found a leaf (A), append A to output, and reset ptr to the root ;
Next: decode bit i

HUFFMAN DECODING
--ILLUSTRATION: DECODING 010100011000 (15/21) --

Codewords
A: 0
B: 10
C: 1100
D: 1101
E: 11100
F: 11101
G: 11110
H: 111111

Recall:
Coder coded ABBAACA into
Bitstream: 010100011000



i
010100011000:
Output=ABBAA

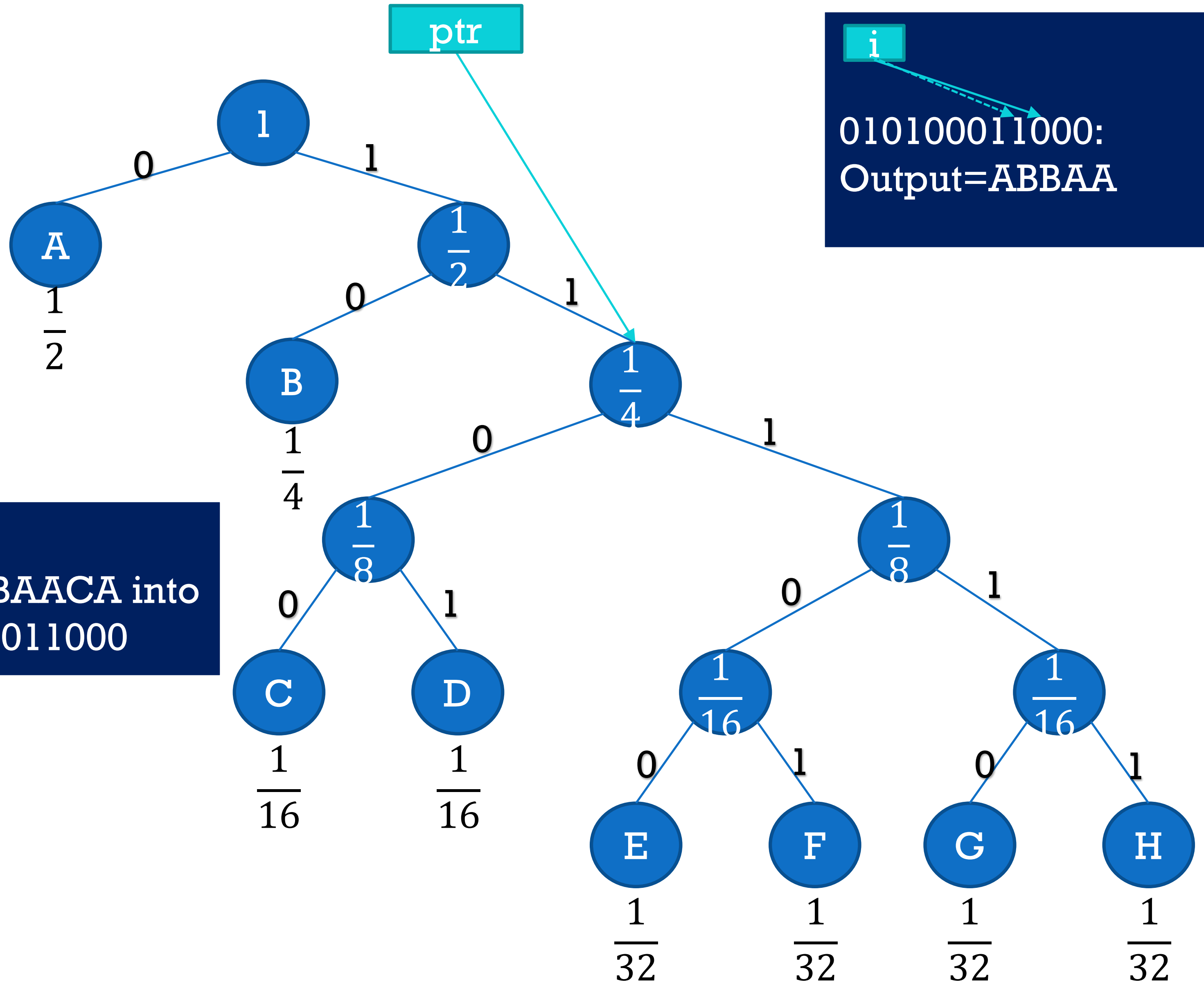
Bit i is 1, ptr goes right, and i advances

HUFFMAN DECODING

--ILLUSTRATION: DECODING 010100011000 (16/21) --

Codewords
A: 0
B: 10
C: 1100
D: 1101
E: 11100
F: 11101
G: 11110
H: 111111

Recall:
Coder coded ABBAACA into
Bitstream: 010100011000



i
010100011000:
Output=ABBAA

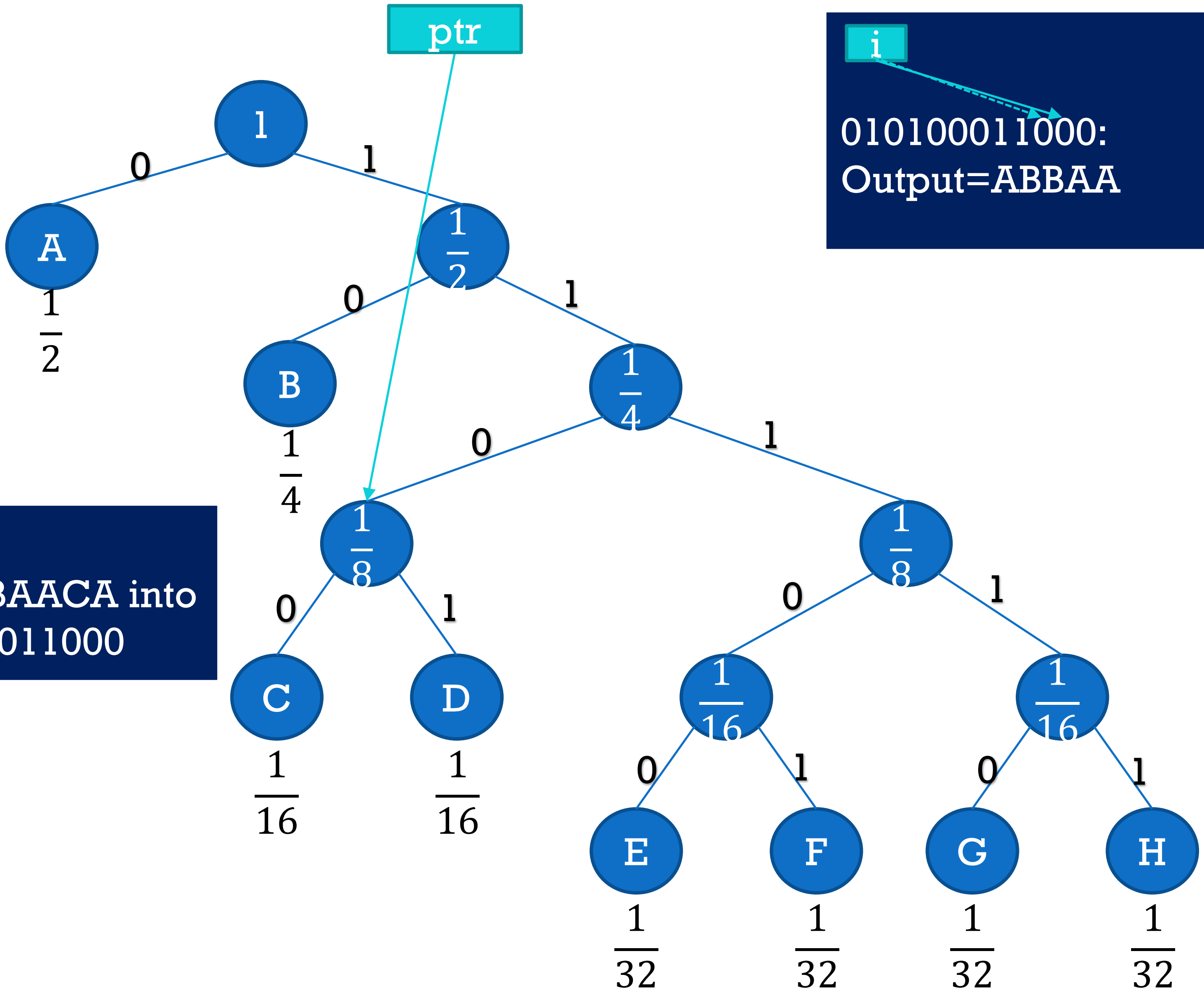
Bit i is 1, ptr goes right, and i advances

HUFFMAN DECODING

--ILLUSTRATION: DECODING 010100011000 (17/21) --

Codewords
A: 0
B: 10
C: 1100
D: 1101
E: 11100
F: 11101
G: 11110
H: 111111

Recall:
Coder coded ABBAACA into
Bitstream: 010100011000



i
010100011000:
Output=ABBAA

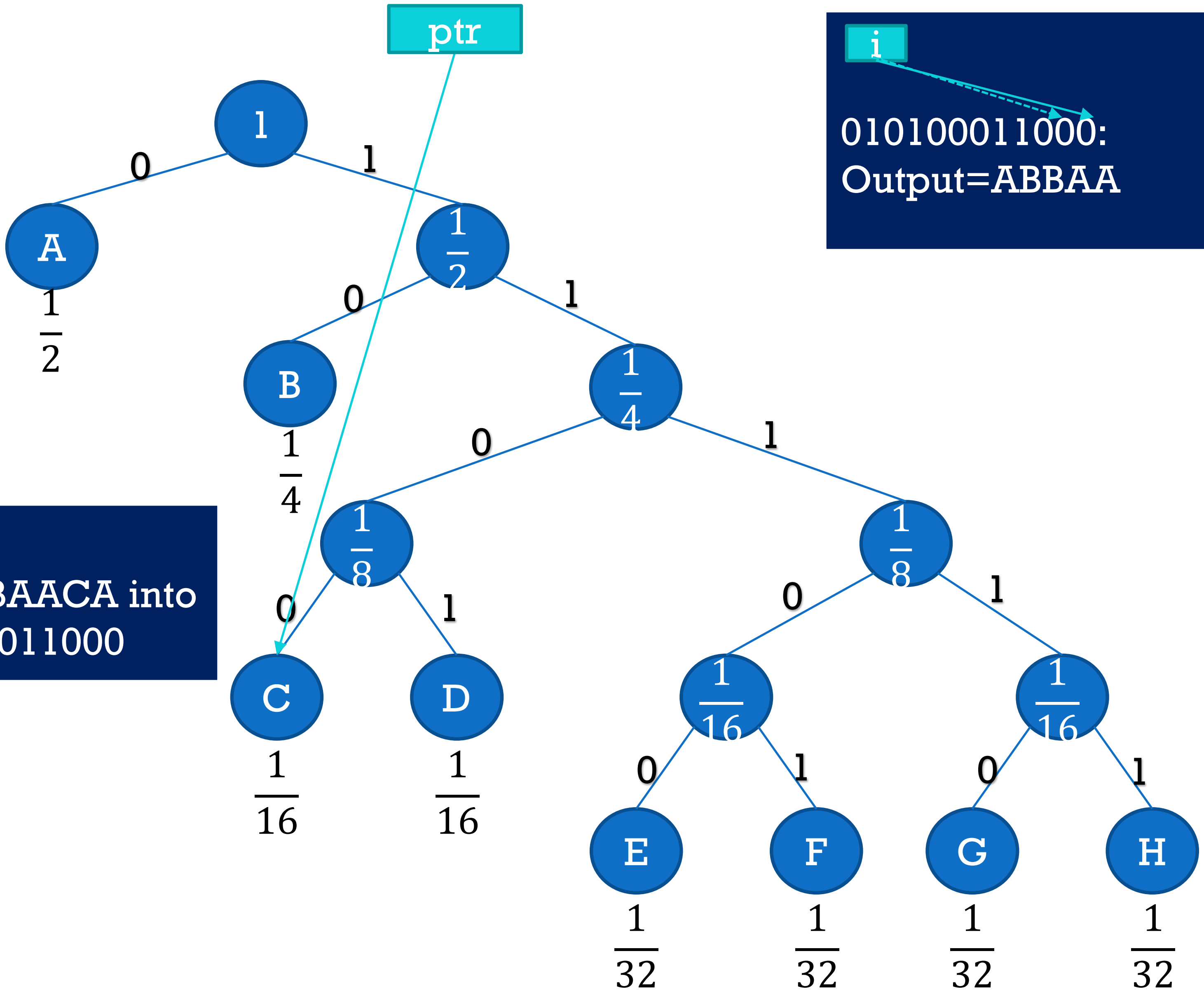
Bit i is 0, ptr goes left, and i advances

HUFFMAN DECODING

--ILLUSTRATION: DECODING 010100011000 (18/21) --

Codewords
A: 0
B: 10
C: 1100
D: 1101
E: 11100
F: 11101
G: 11110
H: 111111

Recall:
Coder coded ABBAACA into
Bitstream: 010100011000



i

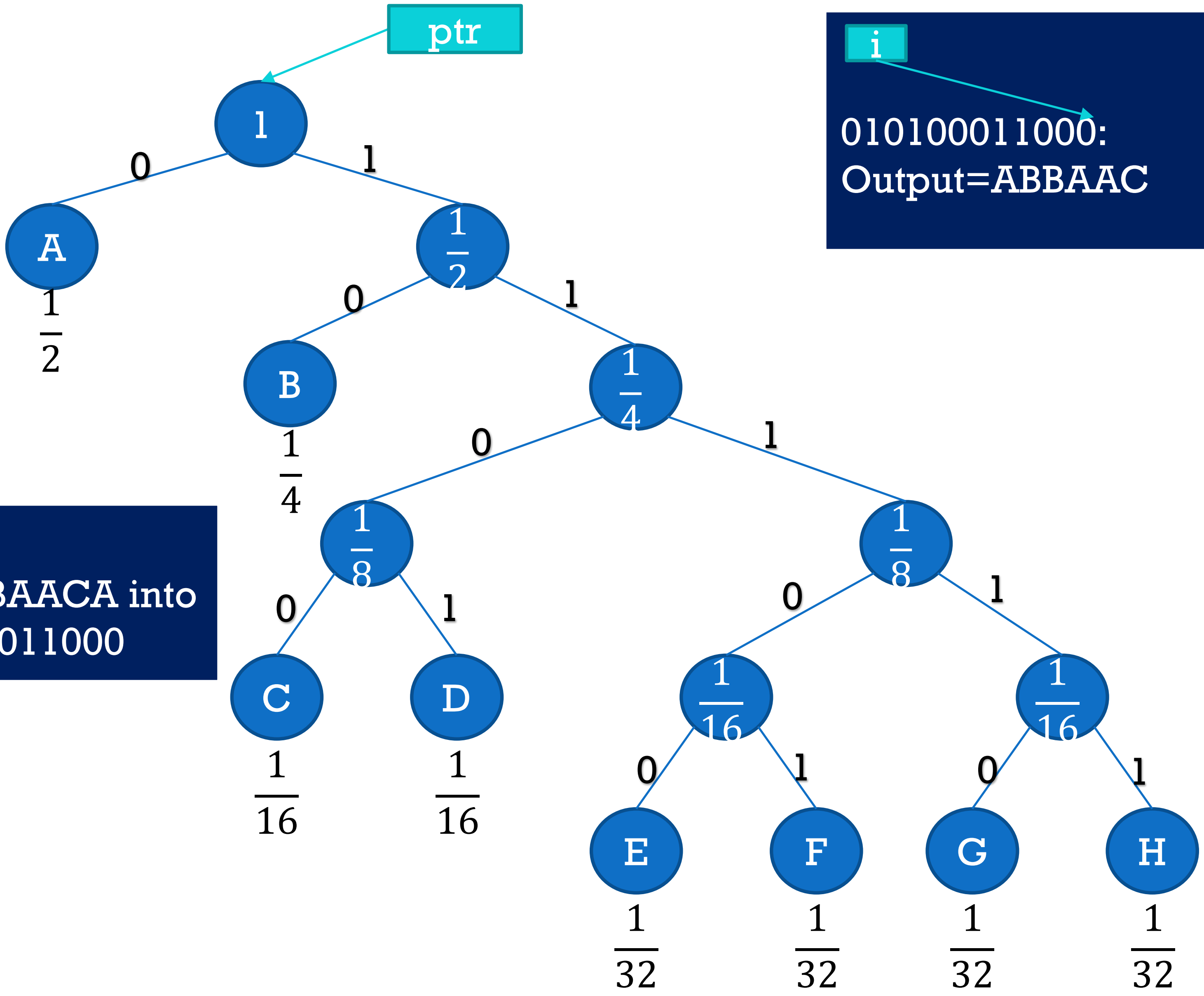
010100011000:
Output=ABBAA

Bit i is 0, ptr goes left, and i advances

HUFFMAN DECODING
--ILLUSTRATION: DECODING 010100011000 (19/21) --

Codewords
A: 0
B: 10
C: 1100
D: 1101
E: 11100
F: 11101
G: 11110
H: 111111

Recall:
Coder coded ABBAACA into
Bitstream: 010100011000



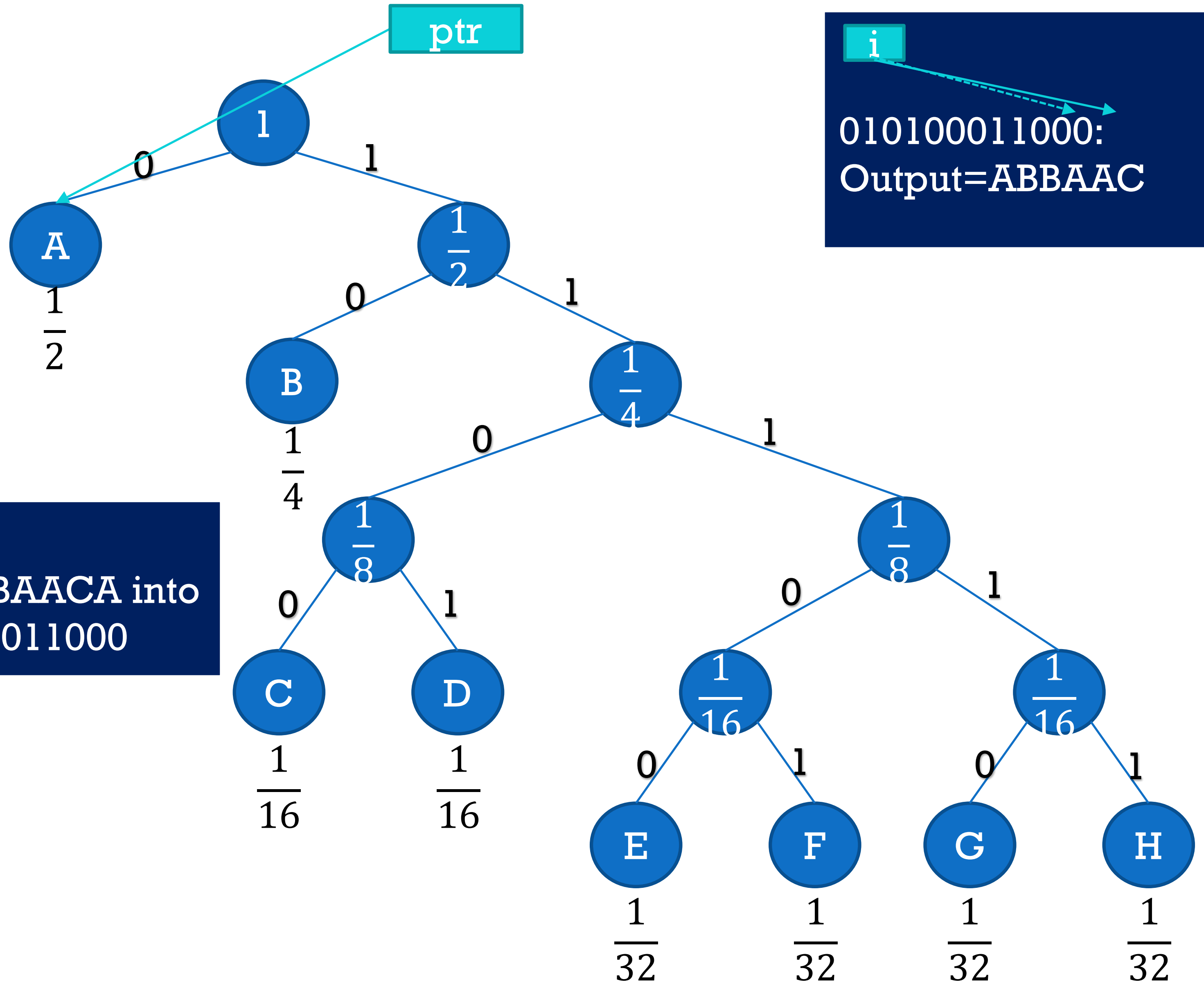
i
010100011000:
Output=ABBAAC

Found a leaf (C), append C to output, and reset ptr to the root ; Next: decode bit i

HUFFMAN DECODING

--ILLUSTRATION: DECODING 010100011000 (20/21) --

Codewords
A: 0
B: 10
C: 1100
D: 1101
E: 11100
F: 11101
G: 11110
H: 111111



i

010100011000:
Output=ABBAAC

Bit i is 0, ptr goes left, and i advances

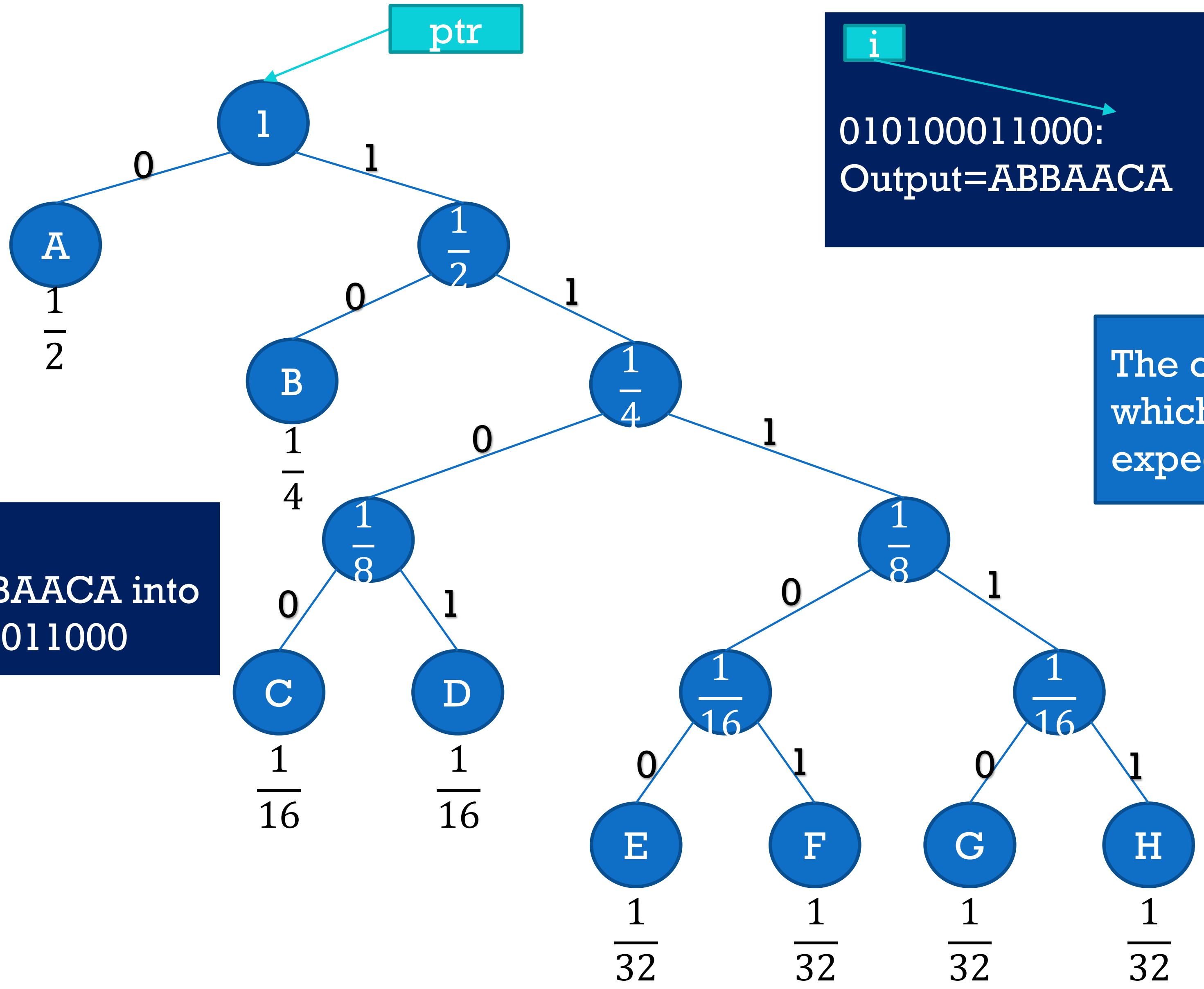
Recall:
Coder coded ABBAACA into
Bitstream: 010100011000

HUFFMAN DECODING

--ILLUSTRATION: DECODING 010100011000 (21/21) --

Codewords
A: 0
B: 10
C: 1100
D: 1101
E: 11100
F: 11101
G: 11110
H: 111111

Recall:
Coder coded ABBAACA into
Bitstream: 010100011000



i

010100011000:
Output=ABBAACA

Found a leaf (A), append A to output, and reset ptr to the root ;
Next: i>N. Stop

The output is ABBAACA, which is what we are expecting!

HUFFMAN CODING/DECODING
-- IMPLEMENTATION ISSUES (1/2) --

- The coder is straightforward and presents no implementation issues
- The decoder needs the tree, which is OK, but can we find a more convenient alternative?
- Yes, we can use the table of codewords: called the ***Huffman Table*** **(HT)**
  - Easier to store
  - But where do we store it?
- This leads to a more general question:
  - The decoder typically needs information from the (en)coder, e.g., tree/HT
  - How/where do we provide that info to the decoder?
  - Remember, the coder may code some data now, and the decoder gets to decode it 10 years later

HUFFMAN CODING/DECODING
-- IMPLEMENTATION ISSUES (2/2) --

- The general question:
  - How/where do we provide that info to the decoder?
- Two alternatives:
  A. We can store that information with <u>every</u> coded bitstream, as a *header*
     - That info, e.g., the HT, is turned into a string and prefixed to the bitstream
  B. Or store it in the decoder software once and for all
- Pros and cons of each alternative:

	Alternative A: in bitstream	Alternative B: in the decoder
Pros	The decoder is not limited to just one particular source type or one HT/Tree	No overhead, i.e., no increase of the bitstream size, thus getting better BR and CR
Cons	Takes more bits per bitstream, worsening the BR and CR	Decoder limited to one source type, one HT/tree. OK for a single application

HUFFMAN CODING
-- THE PREFIX PROPERTY --

- **The prefix property**: a coding scheme where every alphabet symbol is coded with a codeword is said to have the prefix property if no codeword is a prefix of another codeword

A string x is a *prefix* of a string y if y starts with x, i.e., y=xz.
Example: 010 is a prefix of 01011 (because 01011= 01011)

- Huffman coding has the prefix property because
  - Every codeword is a path from the root to a leaf in the Huffman tree, and no leaf is on the path from the root to another leaf
- Why is the prefix property essential for coders that rely on symbol codewords?
- **Exercise**: Take an alphabet {A,B,C} and codewords: A:0, B:1, C:01
  - Code AAB and AC. What are the resulting bitstreams? What do you observe?
  - What is wrong if 2 input data sequences are coded to the same bitstream?

BLOCK HUFFMAN

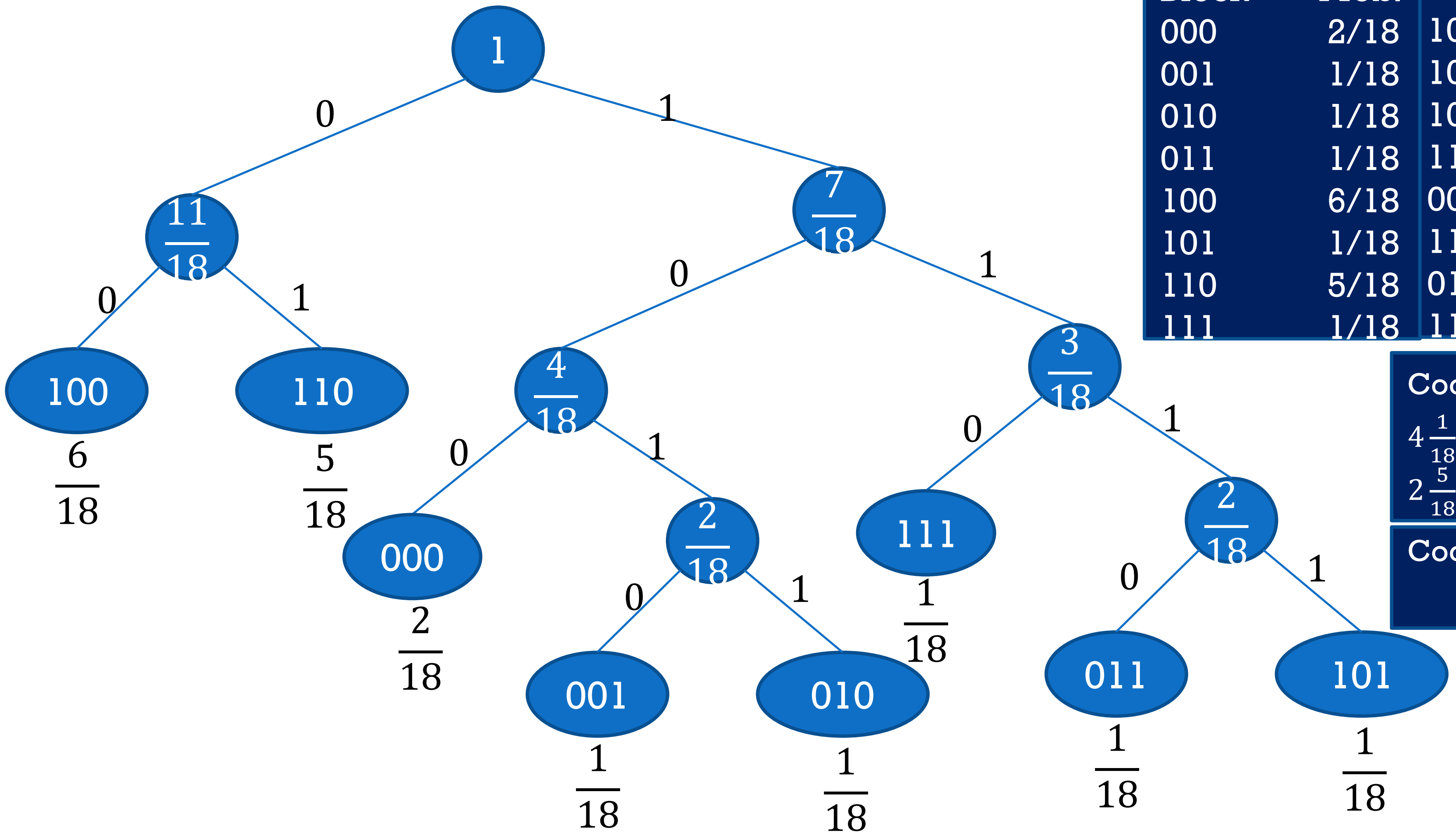
- We saw last lecture that certain (binary) inputs seem to be random but, when looked at as blocks (of a carefully chosen size), some blocks occur more often than others
- In such cases:
  - Treat every block as a new (macro)symbol
  - Take all possible macro-symbols (as a new macro-alphabet)
  - Compute the probabilities of the individual macro-symbols
  - Apply Huffman coding on the macro-symbols, getting a tree and codewords for the macro-symbols
  - Code the original input by replacing each block by its corresponding codeword
  - Decode similarly, getting back the blocks, which are appended to the output

BLOCK HUFFMAN
-- EXAMPLE (1/3) --

- Input I: 100100110100110110100110100000100010000110111011101001
- Break into 3-bit blocks:

  I: 100 100 110 100 110 110 100 110 100 000 100 010 000 110 111 011 101 001
- Number of block occurrences in I: 18 blocks
- The number of distinct blocks is 2^3=8: 000, 001, 010, 011, 100, 101, 110, 111
- If you wish, give them letter names: A B C D E F G H
- Probabilities: Pr[block]=$\frac{number\ of\ occurrences\ of\ the\ block}{18}$
- $\Pr[001] = \Pr[010] = \Pr[011] = \Pr[101] = \Pr[111] = \frac{1}{18}, \Pr[100] = \frac{6}{18},$
- $\Pr[110] = \frac{5}{18}, \Pr[000] = \frac{2}{18}$

BLOCK HUFFMAN
-- EXAMPLE (2/3) --



Block	Prob.	Codewords for the blocks:
000	2/18	100
001	1/18	1010
010	1/18	1011
011	1/18	1110
100	6/18	00
101	1/18	1111
110	5/18	01
111	1/18	110

Code BR: $3\frac{2}{18}+4\frac{1}{18}+4\frac{1}{18}+4\frac{1}{18}+2\frac{6}{18}+4\frac{1}{18}+2\frac{5}{18}+3\frac{1}{18}=\frac{47}{18}$ bits/block

Code BR: $\frac{47}{18}$ bits/block $=\frac{47/18}{3}=\frac{47}{54}$ bits/bit

BLOCK HUFFMAN
-- EXAMPLE (3/3) --

- I: 100 100 110 100 110 110 100 110 100 000 100 010 000 110 111 011 101 001
- Bitstream: 00 00 01 00 01 01 00 01 00 100 00 1011 100 01 110 1110 1111 1010
- Code BR: $\frac{47}{18}$ bits/block = $\frac{47}{54}$ **bits/bit** (from last slide)
- Data BR: $\frac{|\text{Bitstream}|}{|I|} = \frac{47}{54}$
- It is not surprising that Code BR=Data BR because the probabilities were derived from the input data I
- Compression ratio CR=$\frac{|I|}{|\text{Bitstream}|} = \frac{54}{47}$

Block	Prob.	Codewords for the blocks:
000	2/18	100
001	1/18	1010
010	1/18	1011
011	1/18	1110
100	6/18	00
101	1/18	1111
110	5/18	01
111	1/18	110

WRAP-UP OF HUFFMAN
-- WHERE DO WE GET THE PROBABILITIES? --

- In a real-world setting, all you have is data that you want to compress
- It could be a single text file, or a collection of text files in an organization
- How do you get the probabilities?

No, not from the professor, and not from your boss!

- You compute them from your data files
  - For example, to figure out $\Pr[\text{‘}a\text{’}]$:
    - compute the number of times ‘a’ occurs in your data, and
    - divide that by the length (i.e., number of all symbols) in your data
  - You do similar things if you’re doing block coding

RUN-LENGTH ENCODING (RLE)
-- CONTEXT AND MOTIVATION --

- Suppose your data shows long bursts of identical symbols
- Example: aaaaaaaaaa bbbbbbbbbb dddddddd bbbbbb cccccc aaaaaa
- Where could that happen?
  - When you scan a document into a binary B/W file
  - The scanner shines a very thin line of light horizontally across the page
  - Every point of intersection between the light line and a text line (e.g., ~~foot~~), is turned into a bit=1, and where the line doesn't intersect any writing, the points are turned into 0-bits
  - Thus, each light line on a page becomes like: 0000000100001100000001…
- What is the best way to code such data?
- Block coding can work, but better alternatives exist

RUN-LENGTH ENCODING (RLE)
-- HOW IT WORKS--

- Suppose your data shows long bursts of identical symbols
- Example: aaaaaaaaaaa bbbbbbbbbb dddddddd bbbbbb cccccc aaaaa
- Definition: each maximal stretch of identical symbols is called a ***run***
- For example, each same-color in the following example is a run, for a total of 6 runs:

aaaaaaaaaaa bbbbbbbbbb dddddddd bbbbbb cccccc aaaaa

- RLE coding method: replace each run by (a,L) where 'a' is the symbol repeating in the run, and L is the length of the run, i.e., number of times 'a' occurs in the run
- EX: the example above is coded as (a,11) (b,10) (d,8) (b,6) (c,6) (a,5)

RUN-LENGTH ENCODING (RLE)
-- IMPLEMENTATION ISSUES (1/4) --

- The list of pairs is actually an **intermediate** step in RLE
  - We still need to determine how to code (in binary) each symbol 'a' and each length L in each pair (a,L)
- To start, the lengths L can be converted to binary using decimal-to-binary conversion
- But, not all L's take the same number of bits
  - How will the decoder know where the code of every length L begins and ends in the coded bitstream?
- That is an issue unless we agree on a protocol
- One approach is to fix the number of bits (say M bits) that will be allocated to each length

RUN-LENGTH ENCODING (RLE)
-- IMPLEMENTATION ISSUES (2/4) --

- One approach (for coding the L's) is to fix the number of bits (say M bits) that will be allocated to each length
- With M bits, the maximum length L that can be represented is $2^{M} - 1$
- But what if a run length L > $2^{M} - 1$?
- First approach: **avoidance of the problem by setting M to max ever**
  - If we can determine ahead of time the maximum L, then set M=log(max L)
  - Otherwise, per input, find the max L, set M=log(max L), allocate a fixed number of bytes in the header to store the value of M so the decoder knows it

RUN-LENGTH ENCODING (RLE)
-- IMPLEMENTATION ISSUES (3/4) --

- First approach: avoidance of the problem **by setting M to max ever**
  - If we can determine ahead of time the maximum L, then set M=log(max L)
  - Otherwise, per input, find the max L, set M=log(max L), allocate a fixed number of bytes in header to store the value of M so the decoder knows it
- If most runs are of lengths << M, then allocating M bits per L is wasteful
- Second approach: **run-splitting**
  - Set M to a reasonable, non-wasteful value even if M<log(max L)
  - If a run has length $L>2^M-1$:
    - $L=q(2^M-1)+r$ where $r<2^M-1$
    - split the run into q+1 runs: the first q runs are of length 2^M-1 (each needs M bits)
    - the last run of length r (needs $\leq$ M bits)

RUN-LENGTH ENCODING (RLE)
-- RUN-SPLITTING EXAMPLE --

- I: aaaaaaaaaaa bbbbbbbbbb dddddddd bbbbbb cccccc aaaaa
- Intermediate code: (a,11) (b,10) (d,8) (b,6) (c,6) (a,5)
- Assume M=3 (so, max run-length representable is $7=2^3-1$)
- (a,11) is split into (a,7) (a,4), which in binary is (a,111) (a, 100)
- (b,10) is splits into (b,7) (b,3), which in binary is (b,111) (b, 011)
- (d,8) is splits into (d,7) (d,1), which in binary is (d,111) (d, 001)
- (b,6) (c,6) (a,5) need not be split: in binary (b,110) (c,110) (a,101)
- So, the next intermediate code becomes:

(a,111) (a, 100) (b,111) (b, 011) (d,111) (d, 001) (b,110) (c,110) (a,101)

RUN-LENGTH ENCODING (RLE)
-- IMPLEMENTATION ISSUES (4/4) --

- Now we need to code the symbols 'a' in each pair
- Well, we can use fixed-length codes, like ascii or Unicode
- But what if we have a much smaller alphabet, and/or some symbols occur (in the intermediate representation) more often then others?
- Variable-length coding is a better choice
- Which coding?
  - Huffman coding
- In the last example
  - (a,111) (a, 100) (b,111) (b, 011) (d,111) (d, 001) (b,110) (c,110) (a,101)
  - $\Pr[a] = \frac{3}{9}, \Pr[b] = \frac{3}{9}, \Pr[c] = \frac{1}{9}, \Pr[d] = \frac{2}{9}$. Run Huffman coding

RUN-LENGTH ENCODING (RLE)
-- FINISHING THE EXAMPLE--

- In the last example
  - (a,111) (a, 100) (b,111) (b, 011) (d,111) (d, 001) (b,110) (c,110) (a,101)
  - $\Pr[a] = \frac{3}{9}, \Pr[b] = \frac{3}{9}, \Pr[c] = \frac{1}{9}, \Pr[d] = \frac{2}{9}$. Run Huffman coding
- After we get the tree, we find the codewords: a:0, b:10, c:110, d:111
- The next intermediate code

  (0,111) (0, 100) (10,111) (10, 011) (111,111) (111, 001) (10,110) (110,110) (0,101)
- Final bitstream: remove the commas and parentheses:
  - 0 111 0 100 10 111 10 011 111 111 111 001 10 110 110 110 0 101
  - 0111010010111100111111111100110110110100101
- BR: $\frac{44}{46}$ bits/symbol

RLE DECODING

- It is an alternating sequence of decodings:
  - Huffman decoding, bin-2-dec of next M bits, Huffman decoding, bin-2-dec of the next M bits, …
- This decodes to an intermediate representation of (a,L) pairs
- Finally, replace each (a,L) by aaaa…a, where the run is of length L
- Example:

Codewords: a:0, b:10, c:110, d:111

M=3

  - Bitstream= 011101001011110011111111111001101101101100101
  - Alternating decoding: (a,7) (a, 4) (b,7) (b, 3) (d,7) (d, 1) (b,6) (c,6) (a,5)
  - Final decoding: aaaaaaa aaaa bbbbbbb bbb ddddddd d bbbbbb cccccc aaaaa
  - Without spaces: aaaaaaaaaaabbbbbbbbbbdddddddbbbbbbccccccaaaaa
  - You can verify that the decoded data is identical to the original data

RUN-LENGTH ENCODING (RLE)

-- SPECIAL CASE: BINARY DATA --

- If the data to be coded is all binary, like

  000000111111100000011111100000000011111110000 0 …

- The intermediate code is like $(0,L_1)$ $(1,L_2)$ $(0,L_3)$ $(1,L_4)$ $(0,L_5)$ $(1,L_6)$ …
- Notice how the symbol parts alternate between 0 and 1 predictably
- So we can remove them for a cleaner intermediary: $L_1\ L_2\ L_3\ L_4\ L_5\ L_6$…
- This saves bits, and reduces the problem to coding the lengths only
- But now, the run-splitting approach doesn't work. Why?
- Instead, we have to resort to a max-M approach, which can be wasteful
- We will see next a better, more practical approach for encoding binary runs: Golomb and differential Golomb coding

GOLOMB CODING
-- PRELIMINARIES (1/2) --

- Golomb coding (GC) is a practical and powerful implementation of Run-Length Encoding of binary streams
- It works by dividing the input stream into ***Golomb runs*** of the form 0^i1,
- Notation: 0^i stands for a run of i 0's. Ex: $0^4 \equiv 0000$, 0^1=0, 0^0=empty string
- For example, if the input stream is 0000001000100000111000001
  - Break the input as follows: 0000001 0001 000001 1 1 00001
  - This is represented for now as: $0^6 1$ $0^3 1$ $0^5 1$ $0^0 1$ $0^0 1$ $0^4 1$
- Then, GC will code each substring 0^i1 in a special way explained later
- The last bit (i.e., bit 1) of a Golomb run 0^i1 is called the ***tail bit*** of the run
- The last Golomb run of an input stream may or may not have a tail bit

GOLOMB CODING
-- PRELIMINARIES (2/2) --

- Golomb coding has a parameter m
  - m is a positive integer set by the algorithm implementer or by the user
  - the optimal value of m = the nearest power of 2 to $p \times \frac{\text{Ln } 2}{1-p}$ (proof is later)
  - where p is the probability of the more probable bit in the input stream
  - p is easily computable: count the number of 0's (say N_0) and the number of 1's (say N_1) in the input binary stream, then $p = \max(\frac{N_0}{N_0+N_1}, \frac{N_1}{N_0+N_1})$
  - let's assume that the more probable bit (MPB) is 0
  - If the MPB is 1, then each run is of the form $1^i 0$
- Note: $\text{Ln } 2 = 0.6931$

GOLOMB CODING
-- METHOD: ASSUME 0 IS THE MPB --

1. Break the input into runs of the form 0^i1
2. Code each Golomb run 0^i1 as 1^q0y where
   - Divide i by m, integer division, we get quotient q and a remainder r, i.e., $\boldsymbol{i = qm + r}$
   - y is the binary representation of r, using $\log m$ bits: $\boldsymbol{y = (r)_2}$.

Code of 0^i1: $\longrightarrow 1^q \underbrace{0}_{\text{Separator bit}} y$

3. The final coded bitstream is: **MPB code$_1$ code$_2$ … code$_n$ tail?** where
   - MPB: the (1-bit) value of the more probable bit
   - code$_j$: the code of the j-th Golomb run
   - tail?: a single bit that is 1 if the last run has a tail; it is 0 if the last run has no tail

GOLOMB CODING
-- EXAMPLE--

- Input stream: $x = 0^9 1 0^{15} 1 1$
- MPB=0 since 0 occurs 24 times, while 1 occurs 3 times;
- $p = \frac{24}{27}, p \times \frac{\text{Ln } 2}{1-p} = 8 \times 0.6931 = 5.54$, the closest 2-power to 5.54 is 4, so $\boldsymbol{m} = \mathbf{4}$, **log** $\boldsymbol{m}$**=2**;
- The Golomb runs of $x = 0^9 1 0^{15} 1 1$ are $0^9 1, 0^{15} 1$, and $1 = 0^0 1$

Code $0^i 1$ as $1^q 0 y$

- Code $0^9 1: 9 = 2 \times 4 + \mathbf{1}$, so $q = 2$ and $\boldsymbol{r} = \mathbf{1} = (\mathbf{01})_{\mathbf{2}}$, thus code($0^{10}1$)=$1^2\ 0\ 01$
- Code $0^{15} 1: 15 = 3 \times 4 + \mathbf{3}$, so $q = 3$ and $\boldsymbol{r} = \mathbf{3} = (\mathbf{11})_{\mathbf{2}}$, thus code($0^{15}1$)=$1^3\ 0\ \mathbf{11}$
- Code $0^0 1: 0 = 0 \times 4 + \mathbf{0}$, so $q = 0$ and $\boldsymbol{r} = \mathbf{0} = (\mathbf{00})_{\mathbf{2}}$, thus code($0^0 1$)=$1^0\ 0\ \mathbf{00} = 000$
- tail? = 1 because the last Golomb run has a tail
- The code of x is: **0** $1^2 001\ 1^3 011\ 000$ **1**, i.e., 0110011110110001
- CR=$\frac{27}{16} = 1.69$, $\quad \text{BR} = \frac{16}{27} = 0.59$ bits/bit

GOLOMB DECODING
-- PRELIMINARIES --

- Given a coded bitstream BS (like **0**11001111011000**1**), the decoder must
  1. Break BS[2:N-1] down into substrings of the form 1^q0y (y has $\log m$ bits)
  2. Convert 1^q0y into 0^i1 where $i = q \times m + r$ and r=decimal(y)
- The 2nd step is easy
- The 1st step requires finding where every substring 1^q0y begins and ends in the coded bitstream:
  - it is a bit challenging because q varies from substring to substring
- Well, 1^q ends at the "next" 0, y is the next $\log m$ bits after that 0, and the next 1^q0y starts after that
  - The very first 1^q0y starts at the 2nd bit of the coded bitstream

GOLOMB DECODING

-- METHOD --

Input: a coded bistream (& parameter m); **Output**: the original data

Method:

1. Grab first bit as the MPB, and the last bit of the coded bitstream as the tail
2. Set k=2 // index of the bits in the coded bitstream
3. Scan the bitstream rightward from position k looking for successive 1's, keeping a count q, and incrementing k along the way
4. When a 0 is met, read the next $\log m$ bits as y, set k=k+ $\log m$
5. Set r=b2d(y) // binary to decimal conversion
6. Compute $i = q \times m + r$
7. Append $0^i 1$ to the output (if MPB==0), else append $1^i 0$ to the output
8. Repeat from 3 until the coded bitstream is exhausted
9. If the last bit (tail?) is 0, strip the final bit from the output

GOLOMB DECODING
-- EXAMPLE --

- Example: coded bitstream 0110011110110001, and $m = 4$ ($\log m = 2$)
- MPB=0, tail=1;
- k=2, scan the first two 1's till 0, then read 2 bits: 0110011110110001
- So q=2, y=$(01)_2$, r=b2d(01)=1, $i = q \times m + r = 2 \times 4 + 1 = 9$, k=7
- Append $0^9 1$ to output: output=$0^9 1$
- From k=7 scan for 1's till 0: three 1's, 0 and the next two bits 11: 0110011110110001
- So q=3, y=$(11)_2$, r=b2d(11)=3, $i = q \times m + r = 3 \times 4 + 3 = 15$, k=13
- Append $0^{15} 1$ to output: output=$0^9 1 0^{15} 1$
- Look for 1's from position k=13; none found, so q=0; skip 0 and read the next 2 bits, y=00, so r=0; 0110011110110001; so i=0x4+0=0; append $0^0 1$: output=$0^9 1 0^{15} 1 1$
- Now we reached the last bit, tail=1, so we keep the tail. **Final output: $0^9 1 0^{15} 1 1$**

GOLOMB CODING

-- FINDING THE OPTIMAL VALUE OF THE PARAMETER (1/4) --

Theorem: The optimal value of m is the nearest 2-power of $p \times \frac{\text{Ln } 2}{1-p}$, where p is the probability of the most probable bit.

Proof:

- Assume 0 is the more probable bit, its probability is p, and bits are independent
- Let L be the number of 0s in any arbitrary run $0^L 1$
- L is a **random variable** that takes on values in $\{0,1,2,\ldots\}$
- Need the ***average value*** of L, denoted $\bar{L}$, also called ***mean*** or ***expected value*** of L
- From $\bar{L}$, we seek to find the optimal value of m that minimizes the length of the code $1^q 0y$ of the "average" Golomb run $0^{\bar{L}}1$
- Let $p_n = \Pr[L = n]$ for all n. Once we compute the p_n's, we can derive the mean $\bar{L}$

GOLOMB CODING

-- FINDING THE OPTIMAL VALUE OF THE PARAMETER (2/4) --

Proof (Continued):

- $p_n = \Pr[L = n] = \Pr[0^n 1] = \Pr[1^{st}\ bit = 0] \times \Pr[2^{nd} bit = 0] \times \ \ldots \times \Pr[n^{th} bit = 0] \times \Pr[(n+1)^{st} bit = 1]$
- $p_n = p.p. \ldots .p.(1-p) = p^n(1-p)$
- Since L can be 0, 1, 2, 3, …, the mean of L is the average of those values, weighted by their probabilities
- $\bar{L} = 0 \times p_0 + 1 \times p_1 + 2 \times p_2 + 3 \times p_3 + \cdots = \sum_{n=0}^{\infty} n \times p_n = \sum_{n=0}^{\infty} n \times p^n(1-p)$
- $\bar{L} = (1-p)\sum_{n=0}^{\infty} n \times p^n = p(1-p)\sum_{n=0}^{\infty} n \times p^{n-1} = p(1-p)\frac{1}{(1-p)^2} = \frac{p}{1-p}$
- $\bar{L} = \frac{p}{1-p}$
- Now the task is to find m that minimizes the length of the code $1^q 0y$ of $0^{\bar{L}} 1$

GOLOMB CODING

-- FINDING THE OPTIMAL VALUE OF THE PARAMETER (3/4) --

Proof (Continued):

- $\bar{L} = \frac{p}{1-p}$, and the task is to find m that minimizes the length of the code $1^q 0y$ of $0^{\bar{L}}1$
- The length of the code $1^q 0y$ is $\boldsymbol{f = q + 1 + \log m}$**, and we want to minimize it**
- Recall how we get q: $\bar{L} = qm + r$, where $q = \lfloor \frac{\bar{L}}{\mathrm{m}} \rfloor$ and $0 \leq r \leq m - 1$, y=d2b(r) of $\log m$ bits
- Ignore the floor $\lfloor . \rfloor$ in q, and so view $q = \frac{\bar{L}}{\mathrm{m}}$, without significant loss of precision
- Thus, $q = \frac{\bar{L}}{\mathrm{m}} = \frac{\frac{p}{1-p}}{m} = \frac{p}{(1-p)m}$, and so $f = q + 1 + \log m = \frac{p}{(1-p)m} + 1 + \log m = \frac{p}{(1-p)m} + 1 + \frac{Ln\, m}{Ln\, 2}$
- So f, the length of the code, is a function of m, $f(m) = \frac{p}{(1-p)m} + 1 + \frac{Ln\, m}{Ln\, 2}$
- To minimize $f(m)$, we compute its derivative $f'(m)$ and set it to 0 according to Calculus

GOLOMB CODING

-- FINDING THE OPTIMAL VALUE OF THE PARAMETER (4/4) --

Proof (Continued):

- The length of the code, is a function of m, $f(m) = \frac{p}{(1-p)m} + 1 + \frac{Ln\ m}{Ln\ 2}$
- To minimize $f(m)$, we compute its derivative $f'(m)$ and set it to 0 according to Calculus
- To make that possible, we view m as a real variable (not just an integer)
- Computing $f'(m)$ using Calculus: $f'(m) = -\frac{p}{(1-p)m^2} + \left(\frac{1}{m}\right) Ln\ 2$
- Setting $f'(m)$=0 and solving the equation, we get: $\boldsymbol{m = p \times \frac{Ln\ 2}{1-p}}$
- You can verify that this corresponds to a minimum rather than a maximum by proving (easily) that $f'(m) < 0$ for $m < p \times \frac{\text{Ln } 2}{1-p}$, and $f'(m) > 0$ for $m > p \times \frac{\text{Ln } 2}{1-p}$
- To force log m to be integer, we need to take m to be **the closet 2-power to $\boldsymbol{p \times \frac{Ln\ 2}{1-p}}$**. Q.E.D.

DIFFERENTIAL GOLOMB
-- MOTIVATION --

- Consider input streams like 00000011111110000001111111….., i.e., long runs of 0's and long runs of 1's, where 0 is only slightly more probable than 1
- Then Golomb coding does not compress the data at all
- Indeed, when 0 and 1 occur nearly equally, this is what happens
  - the optimal value of m will be equal to 1
  - each 1-run 1^n translates to $n-1$ Golomb runs,of the form $0^0 1$
  - each Golomb run $0^0 1$ codes to $1^0 0$=0
  - therefore, each 1-run 1^n codes to 0^n
  - That is, no compression is achieved for 1-runs
- Differential Golomb improves the situation considerably

DIFFERENTIAL GOLOMB
-- PRELIMINARIES--

- Consider input streams $x_1 x_2 \ldots x_n \ldots$ like 00000011111111000000111111100001
- Compute the successive differences, i.e., replace each bit x_i by $x_i - x_{i-1}$ for all $i \geq 2$
- We get: 0000001000000(-1)00000100000(-1)0001…
- What are we seeing?
  - A nice sequence of Golomb runs
  - But "corrupted" a bit with the negative (-1) of certain tails
- Do those negatives occur with predictable regularity?
  - Yes: every other tail is a -1, and the first tail is always 1
- So, we can remove the negatives, and remember them in the decoding
  - So, intermediate data: 00000010000000100000100000010001…
- Now we can apply regular Golomb coding

DIFFERENTIAL GOLOMB
-- CODING METHOD --

Input: $x = x_1 x_2 \dots x_n$ (& the parameter m)

Output: coded bitstream

Method:

1. Transform x to $z = z_1 z_2 \dots z_n$ where $z_i = x_i - x_{i-1} \forall i > 1$, and $z_1 = x_1$
2. Delete the alternating negatives of the tails, we get $z' = z'_1 z'_2 \dots z'_n$
3. Code z' using Golomb coding

DIFFERENTIAL GOLOMB
-- DECODER METHOD --

Input: Coded bitstream (& the parameter m)

Output: Original data

Method:

1. Golomb-decode the coded bitstream into $z' = z_1' z_2' \dots z_n'$
2. Keep the first Golomb run's tail at 1
3. Alternate the signs of the remaining tails in z', getting $z = z_1 z_2 \dots z_n$
4. Set $x_1 = z_1$, and for $i = 2$ to n do: $x_i = x_{i-1} + z_i$
5. Set output=$x_1 x_2 \dots x_n$

NEXT LECTURE

- Arithmetic coding
- Lempel-Ziv compression