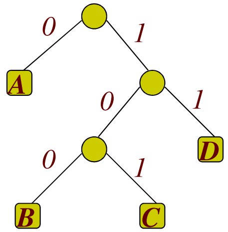
Huffman Coding

- ***David A. Huffman** (1951)*
- *Huffman coding* uses frequencies of symbols in a string to build a variable rate prefix code
  - Each symbol is mapped to a binary string
  - More frequent symbols have shorter codes
  - No code is a prefix of another
- Example:

A	*0*
B	*100*
C	*101*
D	*11*



Variable Rate Codes

- ***Example:***

  1) $A \rightarrow 00$; $B \rightarrow 01$; $C \rightarrow 10$; $D \rightarrow 11$;

  2) $A \rightarrow 0$; $B \rightarrow 100$; $C \rightarrow 101$; $D \rightarrow 11$;
- Two different encodings of *AABDDCAA*
  - *0000011111100000* (16 bits)
  - *00100111110100* (14 bits)

Cost of Huffman Trees

- Let $A=\{a_1, a_2, .., a_m\}$ be the alphabet in which each symbol a_i has probability p_i
- We can define the cost of the Huffman tree *HT* as

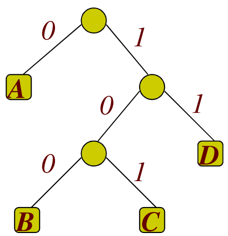
$$C(HT)=\sum_{i=1}^{m} p_i \cdot r_i,$$

  where r_i is the length of the path from the root to a_i
- The cost *C(HT)* is the *expected length* (in bits) of a *code word* represented by the tree *HT*. The value of *C(HT)* is called the *bit rate* of the code.

Cost of Huffman Trees - example

- ***Example:***
  - Let $a_1=A$, $p_1=1/2$; $a_2=B$, $p_2=1/8$; $a_3=C$, $p_3=1/8$; $a_4=D$, $p_4=1/4$ where $r_1=1$, $r_2=3$, $r_3=3$, and $r_4=2$

HT



$$C(HT) = 1 \cdot 1/2 + 3 \cdot 1/8 + 3 \cdot 1/8 + 2 \cdot 1/4 = 1.75$$

Huffman Tree Property

- ***Input:*** Given probabilities $p_1, p_2, .., p_m$ for symbols $a_1, a_2, .., a_m$ from alphabet A
- ***Output:*** A tree that *minimizes the average number of bits* (bit rate) to code a symbol from A
- I.e., the goal is to minimize function:

$$C(HT)=\Sigma p_i \cdot r_i,$$

where r_i is the length of the path from the root to leaf a_i.

This is called a *Huffman tree* or *Huffman code* for alphabet A

Huffman Tree Property

- ***Input:*** Given probabilities $p_1, p_2, .., p_m$ for symbols $a_1, a_2, .., a_m$ from alphabet A
- ***Output:*** A tree that *minimizes the average number of bits* (bit rate) to code a symbol from A
- I.e., the goal is to minimize function:

$$C(HT)=\Sigma p_i \cdot r_i,$$

where r_i is the length of the path from the root to leaf a_i.

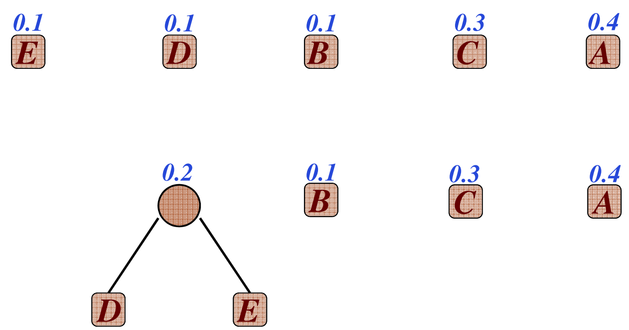
This is called a *Huffman tree* or *Huffman code* for alphabet A

Construction of Huffman Trees

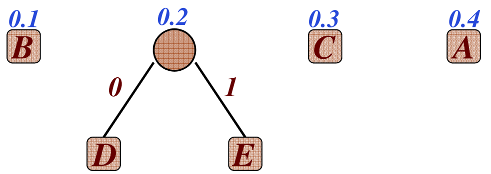
- Form a (tree) node for each symbol a_i with weight p_i
- Insert all nodes to a priority queue PQ (e.g., a heap) ordered by nodes probabilities
- **while** (the priority queue has more than two nodes)
  - $min_1 \leftarrow remove\text{-}min(PQ)$; $min_2 \leftarrow remove\text{-}min(PQ)$;
  - create a new (tree) node T;
  - $T.weight \leftarrow min_1.weight + min_2.weight$;
  - $T.left \leftarrow min_1$; $T.right \leftarrow min_2$;
  - $insert(PQ, T)$
- **return** (last node in PQ)

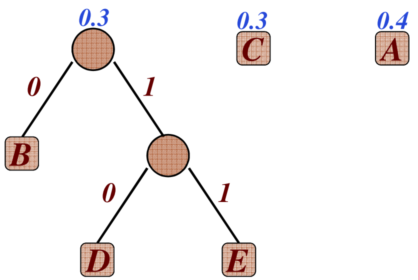
Construction of Huffman Trees

P(*A*)= *0.4*, P(*B*)= *0.1*, P(*C*)= *0.3*, P(*D*)= *0.1*, P(*E*)= *0.1*

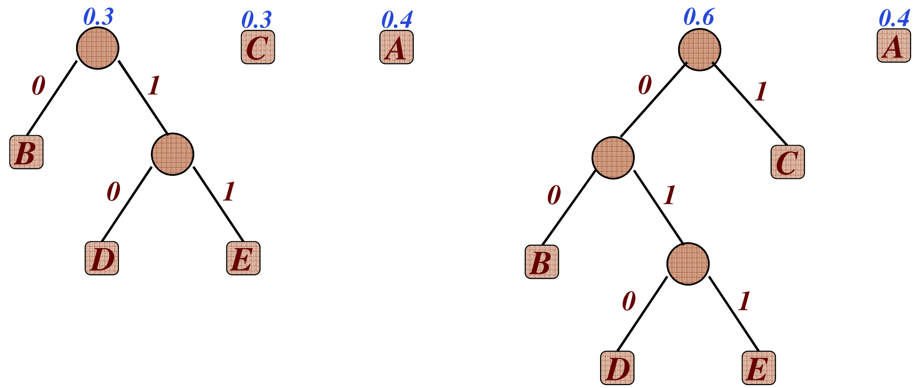


Construction of Huffman Trees

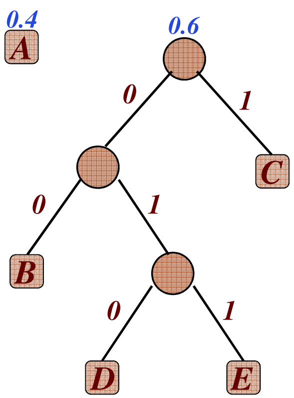


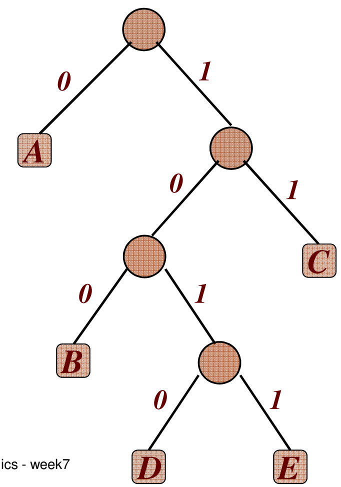


Construction of Huffman Trees

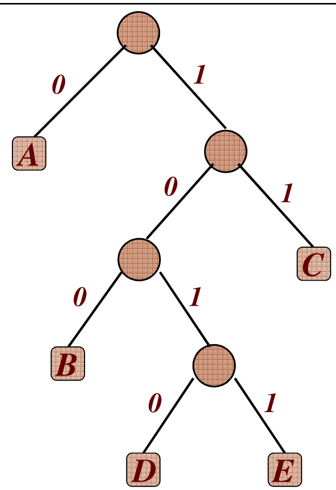


Construction of Huffman Trees





Construction of Huffman Trees



A = 0

B = 100

C = 11

D = 1010

E = 1011

Huffman Codes

- ***Theorem:*** For any source S the Huffman code can be computed efficiently in time $O(n \cdot log\ n)$, where n is the size of the source S.

  Proof: The time complexity of Huffman coding algorithm is dominated by the use of priority queues
- One can also prove that Huffman coding creates the most efficient set of prefix codes for a given text
- It is also one of the most efficient entropy coder

Basics of Information Theory

- The entropy of an information source (string) S built over alphabet $A=\{a_1, a_2, .., a_m\}$is defined as:

  $$H(S) = \sum_i p_i \cdot log_2(1/p_i)$$

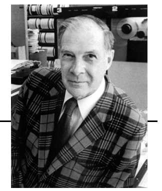
  where p_i is the probability that symbol a_i in S will occur
- $log_2(1/p_i)$ indicates the amount of information contained in a_i, i.e., the number of bits needed to code a_i.
- For example, in an image with uniform distribution of gray-level intensity, i.e. all $p_i = 1/256$, then the number of bits needed to encode each gray level is 8 bits. The entropy of this image is 8.

Huffman Code vs. Entropy

P(A)= 0.4, P(B)= 0.1, P(C)= 0.3, P(D)= 0.1, P(E)= 0.1

- Entropy:
  - $0.4 \cdot log_2(10/4) + 0.1 \cdot log_2(10) + 0.3 \cdot log_2(10/3) + 0.1 \cdot log_2(10) + 0.1 \cdot log_2(10) = 2.05$ bits per symbol
- Huffman Code:
  - $0.4 \cdot 1 + 0.1 \cdot 3 + 0.3 \cdot 2 + 0.1 \cdot 4 + 0.1 \cdot 4 = 2.10$
  - Not bad, not bad at all.

Error detection and correction

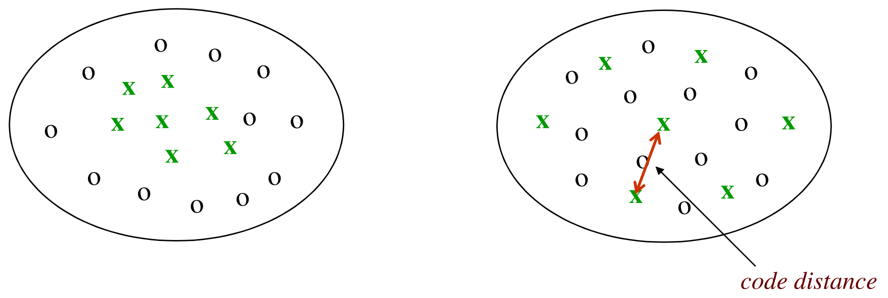


- Hamming codes:
  - *codewords* in Hamming (error detecting and error correcting) codes consist of m data bits and r redundant bits.
  - *Hamming distance* between two strings represents the number of bit positions on which two bit patterns differ (similar to pattern matching k mismatches).
  - *Hamming distance* of the code is determined by the two codewords whose Hamming distance is the *smallest*.
  - *error detection* involves determining if codewords in the received message match closely enough legal codewords.

Error detection and correction

(a) A code with poor distance properties

(b) A code with good distance properties

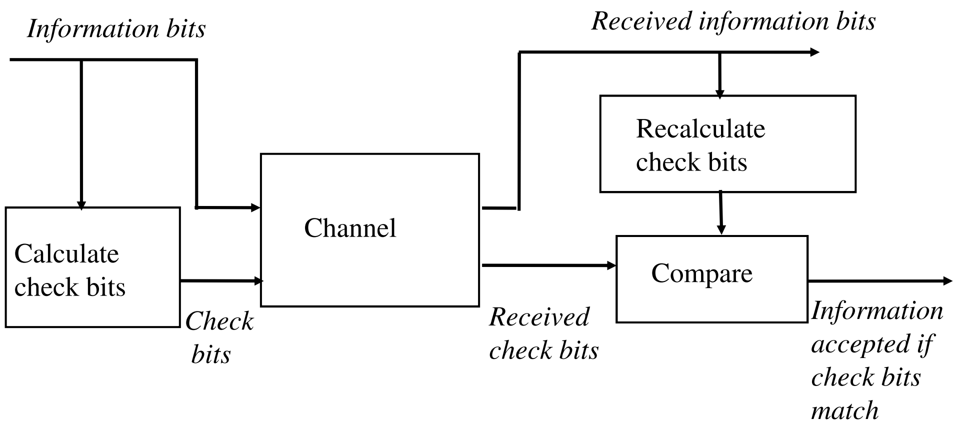


x = codewords o = non-codewords

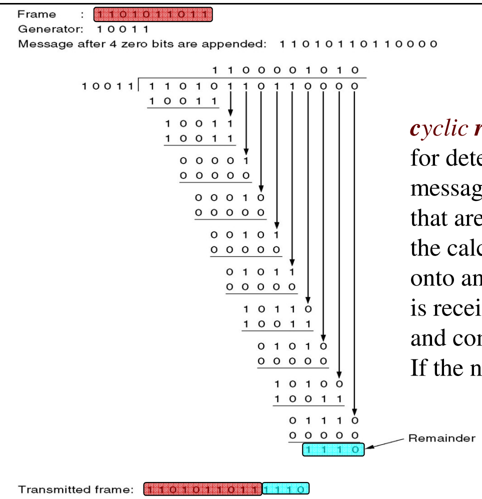
Error detection and correction

- To *detect* properly ***d*** single bit errors, one needs to apply a ***d+1*** code distance.
- To *correct* properly ***d*** single bit errors, one needs to apply a ***2d+1*** code distance.
- In general, the price for redundant bits is *too expensive (!!)* to do *error correction* for all network messages
- Thus safety and integrity of network communication is based on error detecting codes and extra transmissions in case any errors were detected

Error-Detection System using Check Bits



Cyclic Redundancy Checking (CRC)



***c**yclic **r**edundancy **c**heck (CRC)* is a popular technique for detecting data transmission errors. Transmitted messages are divided into predetermined lengths that are divided by a fixed divisor. According to the calculation, the remainder number is appended onto and sent with the message. When the message is received, the computer recalculates the remainder and compares it to the transmitted remainder.
If the numbers do not match, an error is detected.

Error detection -- via parity of subsets of bits

Char.	ASCII	Check bits
H	1001000	00110010000
a	1100001	10111001001
m	1101101	11101010101
m	1101101	11101010101
i	1101001	01101011001
n	1101110	01101010110
g	1100111	01111001111
	0100000	10011000000
c	1100011	11111000011
o	1101111	10101011111
d	1100100	11111001100
e	1100101	00111000101
		12345678 ....

Order of bit transmission

Note
Check bits occupy power of 2 slots

Detection via parity of subsets of bits

Consider 4 bit words. Add 3 parity bits.

$D_3D_2D_1D_0$ $P_2P_1P_0$
0 1 1 0   ? ? ?

Each parity bit computed on a subset of bits

$P_2 = D_3 \text{ xor } D_2 \text{ xor } D_1 = 0 \text{ xor } 1 \text{ xor } 1 = 0$
$P_1 = D_3 \text{ xor } D_2 \text{ xor } D_0 = 0 \text{ xor } 1 \text{ xor } 0 = 1$
$P_0 = D_3 \text{ xor } D_1 \text{ xor } D_0 = 0 \text{ xor } 1 \text{ xor } 0 = 1$

Use this word bit arrangement

$D_3D_2D_1P_2D_0P_1P_0$
0 1 1 0 0 1 1

Check bits occupy power of 2 slots!

Detection via parity of subsets of bits - no error occurred

First, we send: $D_3D_2D_1P_2D_0P_1P_0$
0 1 1 0 0 1 1

Later, someone gets: $D_3D_2D_1P_2D_0P_1P_0$
0 1 1 0 0 1 1

And computes:

$B_2 = P_2 \text{ xor } D_3 \text{ xor } D_2 \text{ xor } D_1 = 0 \text{ xor } 0 \text{ xor } 1 \text{ xor } 1 = 0$
$B_1 = P_1 \text{ xor } D_3 \text{ xor } D_2 \text{ xor } D_0 = 1 \text{ xor } 0 \text{ xor } 1 \text{ xor } 0 = 0$
$B_0 = P_0 \text{ xor } D_3 \text{ xor } D_1 \text{ xor } D_0 = 1 \text{ xor } 0 \text{ xor } 1 \text{ xor } 0 = 0$

No error occurred. But how do we know that?

If all $B_2, B_1, B_0 = 0$ there are no errors!

These equations come from how we computed:

$P_2 = D_3 \text{ xor } D_2 \text{ xor } D_1 = 0 \text{ xor } 1 \text{ xor } 1 = 0$
$P_1 = D_3 \text{ xor } D_2 \text{ xor } D_0 = 0 \text{ xor } 1 \text{ xor } 0 = 1$
$P_0 = D_3 \text{ xor } D_1 \text{ xor } D_0 = 0 \text{ xor } 1 \text{ xor } 0 = 1$

Detection via parity of subsets of bits - single bit is twisted

First, we send: $D_3D_2D_1P_2D_0P_1P_0$
0 1 1 0 0 1 1

What if a cosmic ray hit D_1?
How would we know that?

Later, someone gets: $D_3D_2D_1P_2D_0P_1P_0$
0 1 0 0 0 1 1

And computes:

$B_2 = P_2 \text{ xor } D_3 \text{ xor } D_2 \text{ xor } D_1 = 0 \text{ xor } 0 \text{ xor } 1 \text{ xor } 0 = 1$
$B_1 = P_1 \text{ xor } D_3 \text{ xor } D_2 \text{ xor } D_0 = 1 \text{ xor } 0 \text{ xor } 1 \text{ xor } 0 = 0$
$B_0 = P_0 \text{ xor } D_3 \text{ xor } D_1 \text{ xor } D_0 = 1 \text{ xor } 0 \text{ xor } 0 \text{ xor } 0 = 1$

$B_2B_1B_0 = 101 = 5$
And what does 101= 5 mean?

We number the least significant bit with 1, not 0!
0 is reserved for “no errors”.

7 6 5 4 3 2 1
$D_3D_2D_1P_2D_0P_1P_0$
0 1 0 0 0 1 1

The position of the flipped bit!
To repair, just flip it back

Detection via parity of subsets of bits - magic trick revealed

For any 4 bit word   we add 3 parity bits

$D_3D_2D_1D_0$ $P_2P_1P_0$

0 1 1 0   ? ? ?

Observation: The parity bits need to encode "no error" scenario, plus a number for each bit (both data and parity bits)

For p parity bits and d data bits: $d + p + 1 \leq 2^p$

Detection via parity of subsets of bits - magic trick revealed

For any 4 bit word $D_3D_2D_1D_0$ $P_2P_1P_0$ we add 3 parity bits

Question: Why do we arrange bits?

Start by numbering, 1 to 7.

7 6 5 4 3 2 1

D_3 D_1 D_0 P_0

D_3D_2 D_0P_1

$D_3D_2D_1P_2$

$D_3D_2D_1P_2D_0P_1P_0$

With this order, an odd parity means an error in 1,3,5, or 7. So, P_0 is the right parity bit to use:

An odd parity means a mistake must be in 2, 3, 6, or 7 -- the four numbers possible if $P_1 = 1$!

etc ... each bit narrows down the suspect bits, until it is certain.

Detection via parity of subsets of bits - magic trick revealed

For any 4 bit word $D_3D_2D_1D_0$ $P_2P_1P_0$ we add 3 parity bits

$P_2 = D_3 \text{ xor } D_2 \text{ xor } D_1$ $P_1 = D_3 \text{ xor } D_2 \text{ xor } D_0$ $P_0 = D_3 \text{ xor } D_1 \text{ xor } D_0$

	D_3	D_2	D_1	P_2	D_0	P_1	P_0
0	0	0	0	0	0	0	0
1	0	0	0	0	1	1	1
2	0	0	1	1	0	0	1
3	0	0	1	1	1	1	0
4	0	1	0	1	0	1	0
5	0	1	0	1	1	0	1
6	0	1	1	0	0	1	1
7	0	1	1	0	1	0	0
8	1	0	0	1	0	1	1
9	1	0	0	1	1	0	0
10	1	0	1	0	0	1	0
11	1	0	1	0	1	0	1
12	1	1	0	0	0	0	1
13	1	1	0	0	1	1	0
14	1	1	1	1	0	0	0
15	1	1	1	1	1	1	1

7 bits can code 128 numbers, but only 16 of these numbers are legal.

It takes 3 bit flips to move from one legal number to another (for all 16 numbers)

If only one bit flips, we can always figure out the "closest" legal number, and correct the number.