

CSCI203

Algorithms and Data Structures



Huffman Trees

Lecturer: Dr. Fenghui Ren

Room 3.203

Email: fren@uow.edu.au

Encoding messages



- ▶ Encode a message composed of a string of characters
- ▶ Codes used by computer systems
 - ASCII
 - uses 8 bits per character
 - can encode 256 characters
 - Unicode
 - 16 bits per character
 - can encode 65536 characters
 - includes all characters encoded by ASCII
- ▶ ASCII and Unicode are *fixed-length codes*
 - all characters represented by same number of bits

Problems



- ▶ Suppose that we want to encode a message constructed from the symbols **A**, **B**, **C**, **D**, and **E** using a fixed-length code
 - How many bits are required to encode each symbol?
 - ◆ at least **3 bits** are required
 - ◆ 2 bits are not enough (can only encode four symbols)
 - ◆ How many bits are required to encode the message **DEAACAAAAABA**?
 - ◆ there are twelve symbols, each requires 3 bits
 - ◆ $12 \times 3 =$ **36 bits** are required

Drawbacks of fixed-length codes

- ▶ Wasted space
 - Unicode uses twice as much space as ASCII
 - inefficient for plain-text messages containing only ASCII characters
- ▶ Same number of bits used to represent all characters
 - 'a' and 'e' occur more frequently than 'q' and 'z'
- ▶ Potential solution: use **variable-length codes**
 - variable number of bits to represent characters when frequency of occurrence is known
 - short codes for characters that occur frequently

Advantages of variable-length codes

- ▶ The advantage of variable-length codes over fixed-length is short codes can be given to characters that occur frequently
 - on average, the length of the encoded message is less than fixed-length encoding
- ▶ **Potential problem:** how do we know where one character ends and another begins?
 - not a problem if number of bits is fixed!

A = 00

B = 01

C = 10

D = 11

0010110111001111111111

A C D B A D D D D D

Prefix property

- ▶ A code has the **prefix property** if no character code is the prefix (start of the code) for another character

- ▶ Example:

Symbol	Code
P	000
Q	11
R	01
S	001
T	10

01001101100010

R S T Q P T

- ▶ 000 is not a prefix of 11, 01, 001, or 10
- ▶ 11 is not a prefix of 000, 01, 001, or 10 ...

Code without prefix property

- ▶ The following code does not have prefix property

Symbol	Code
P	0
Q	1
R	01
S	10
T	11

Problem



- ▶ Design a variable-length prefix-free code such that the message **DEAACAAAAABA** can be encoded using 22 bits
- ▶ Possible solution:
 - **A** occurs eight times while **B**, **C**, **D**, and **E** each occur once
 - represent **A** with a one bit code, say 0
 - remaining codes cannot start with 0
 - represent **B** with the two bit code 10
 - remaining codes cannot start with 0 or 10
 - represent **C** with 110
 - represent **D** with 1110
 - represent **E** with 11110

Encoded message

DEAACAAAAABA

Symbol	Code
A	0
B	10
C	110
D	1110
E	11110

1110111100011000000100

22 bits

Another possible code

DEAACAAAAABA

Symbol	Code
A	0
B	100
C	101
D	1101
E	1111

1101111100101000001000

22 bits

Better code

DEAACAAAAABA

Symbol	Code
A	0
B	100
C	101
D	110
E	111

11011100101000001000

20 bits

What code to use?



▶ Question:

- Is there a variable-length code that makes the most efficient use of space?

Answer: Yes!

Huffman coding tree



- ▶ Binary tree
 - each leaf contains symbol (character)
 - label edge from a node to its left child with 0
 - label edge from a node to its right child with 1
- ▶ Code for any symbol obtained by following path from root to the leaf containing symbol
- ▶ Code has prefix property
 - leaf node cannot appear on path to another leaf
 - *note:* fixed-length codes are represented by a complete Huffman tree and clearly have the prefix property

Building a Huffman tree



- ▶ Find frequencies of each symbol occurring in message
- ▶ Begin with a forest of single node trees
 - each contain symbol and its frequency
- ▶ Do recursively
 - select two trees with smallest frequency at the root
 - produce a new binary tree with the selected trees as children and store the sum of their frequencies in the root
- ▶ Recursion ends when there is one tree
 - this is the Huffman coding tree

Example

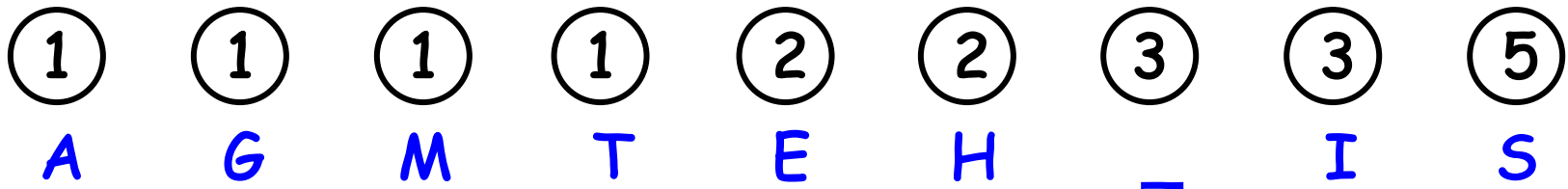
- ▶ Build the Huffman coding tree for the message

This is his message

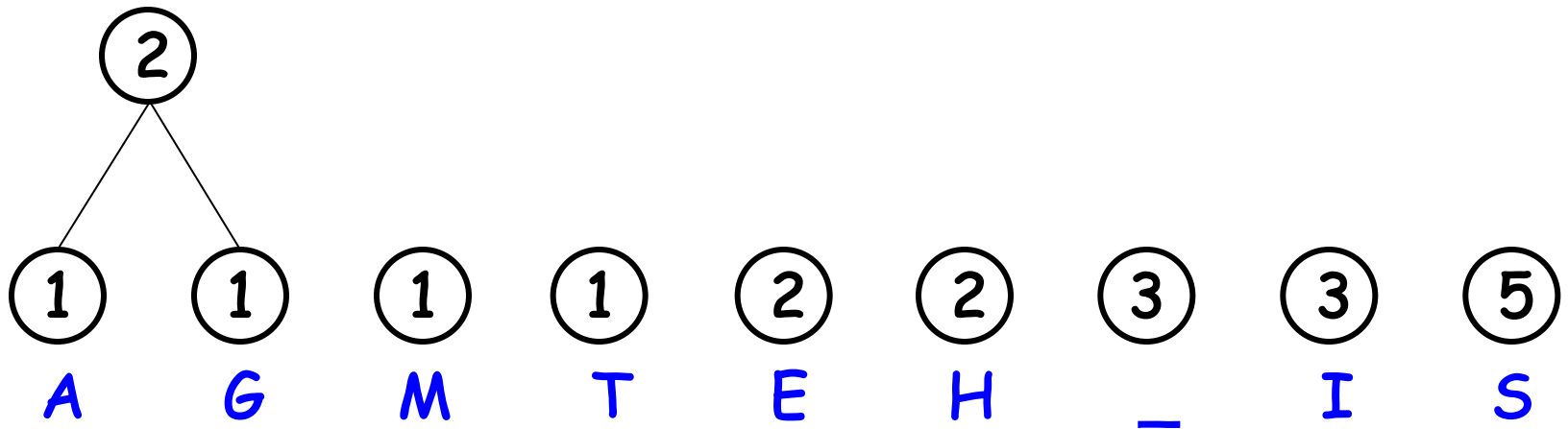
- ▶ Character frequencies

A	G	M	T	E	H	_	I	S
1	1	1	1	2	2	3	3	5

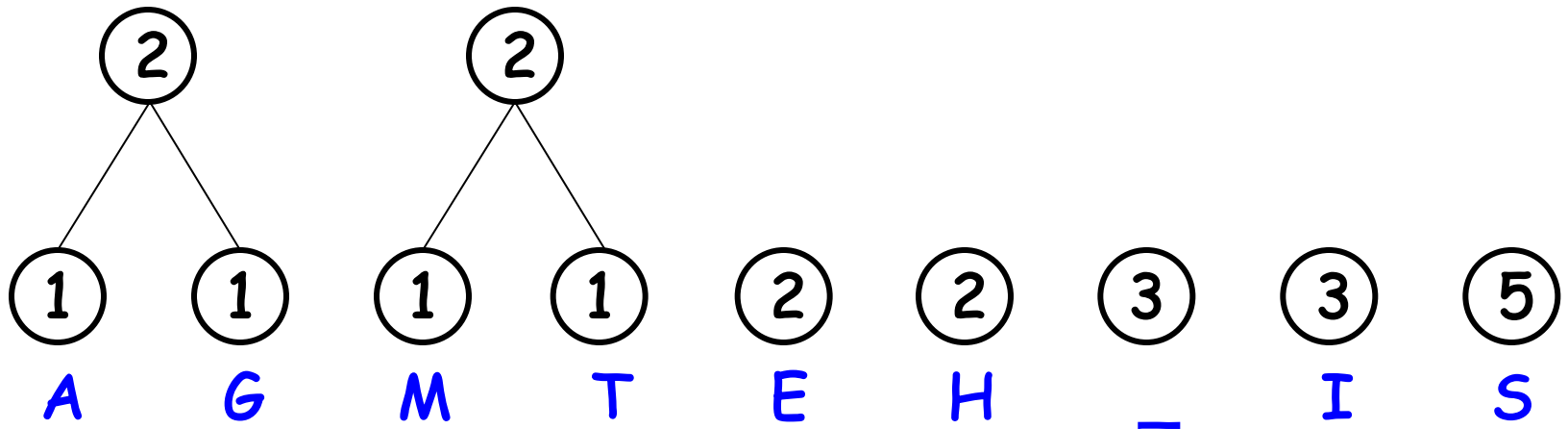
- ▶ Begin with forest of single trees



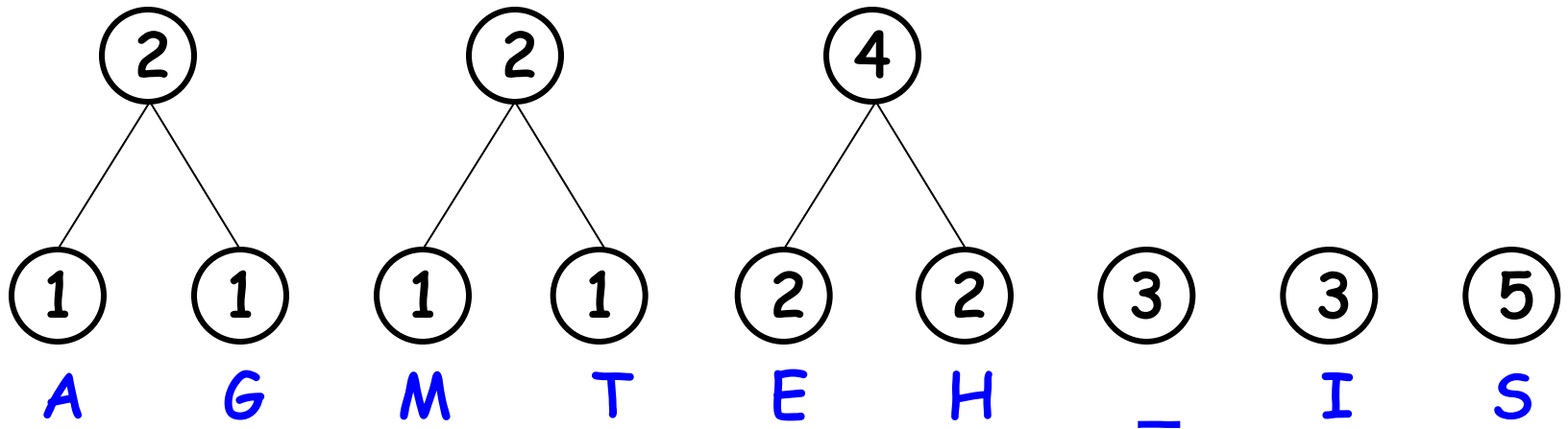
Step 1



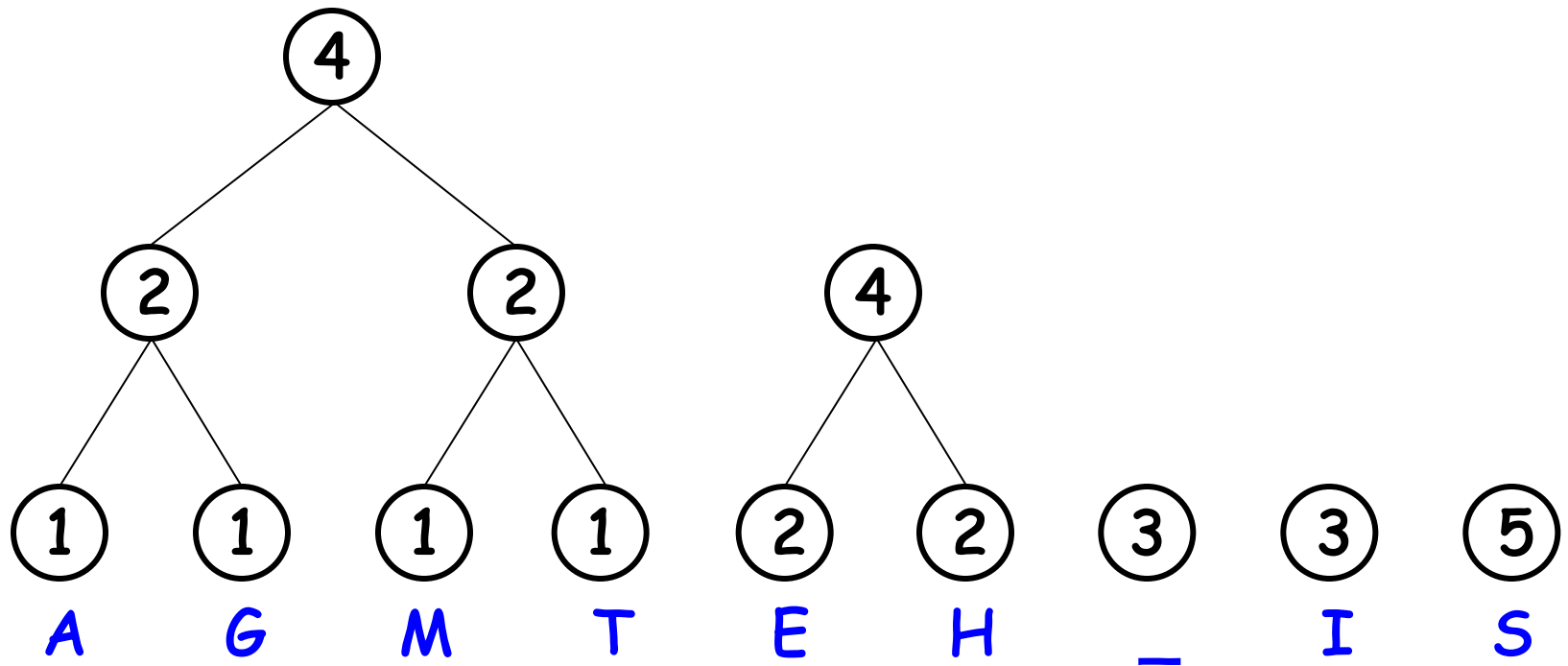
Step 2



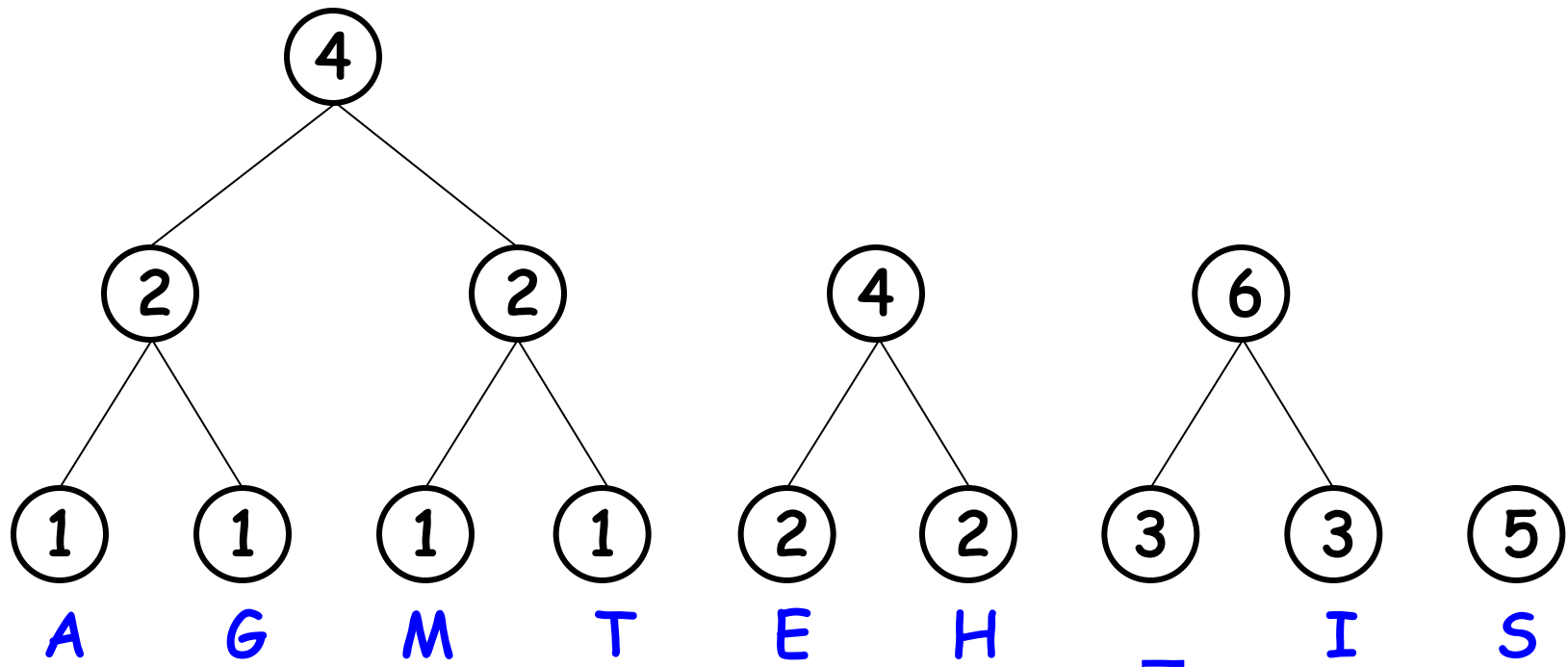
Step 3



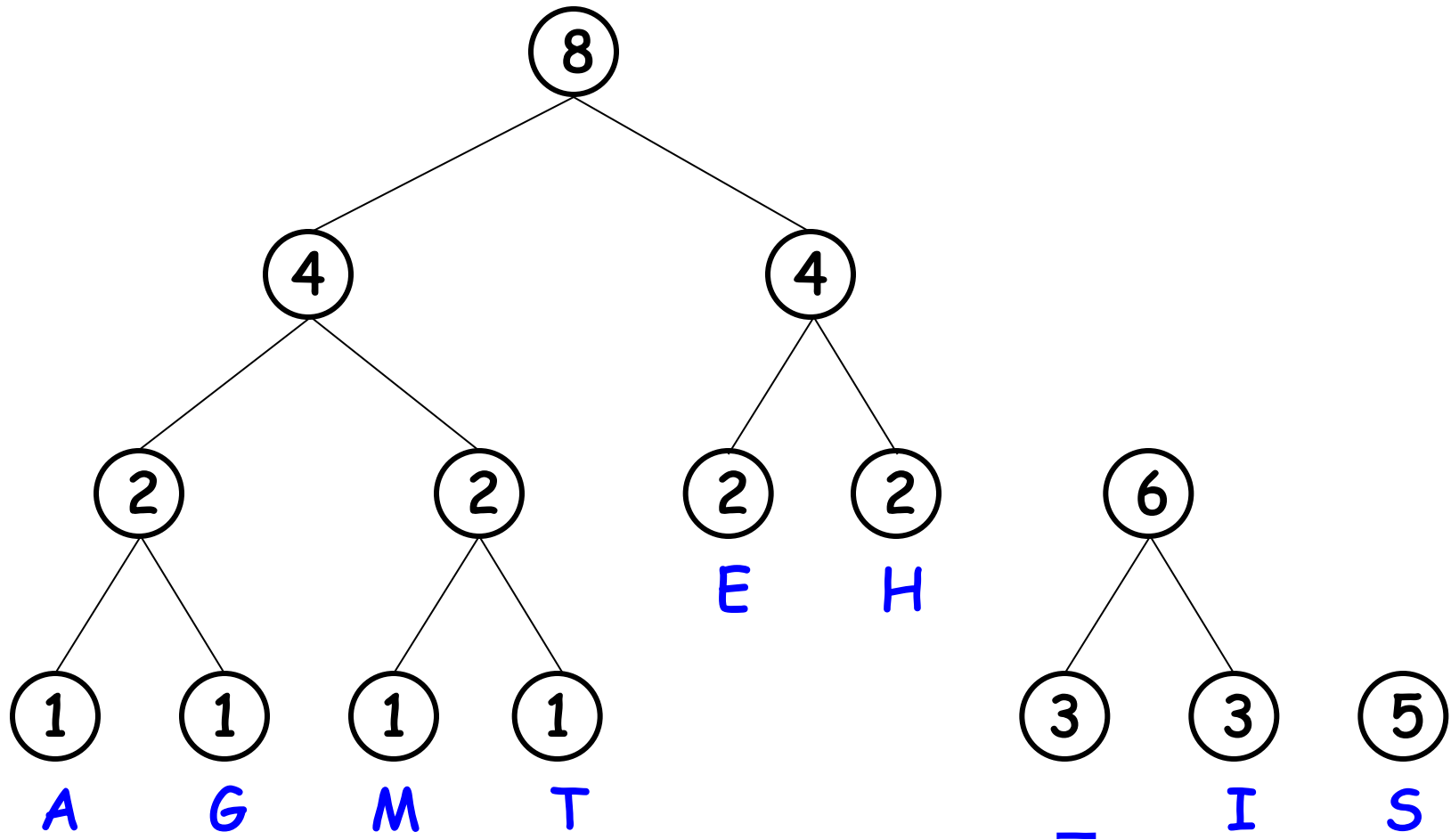
Step 4



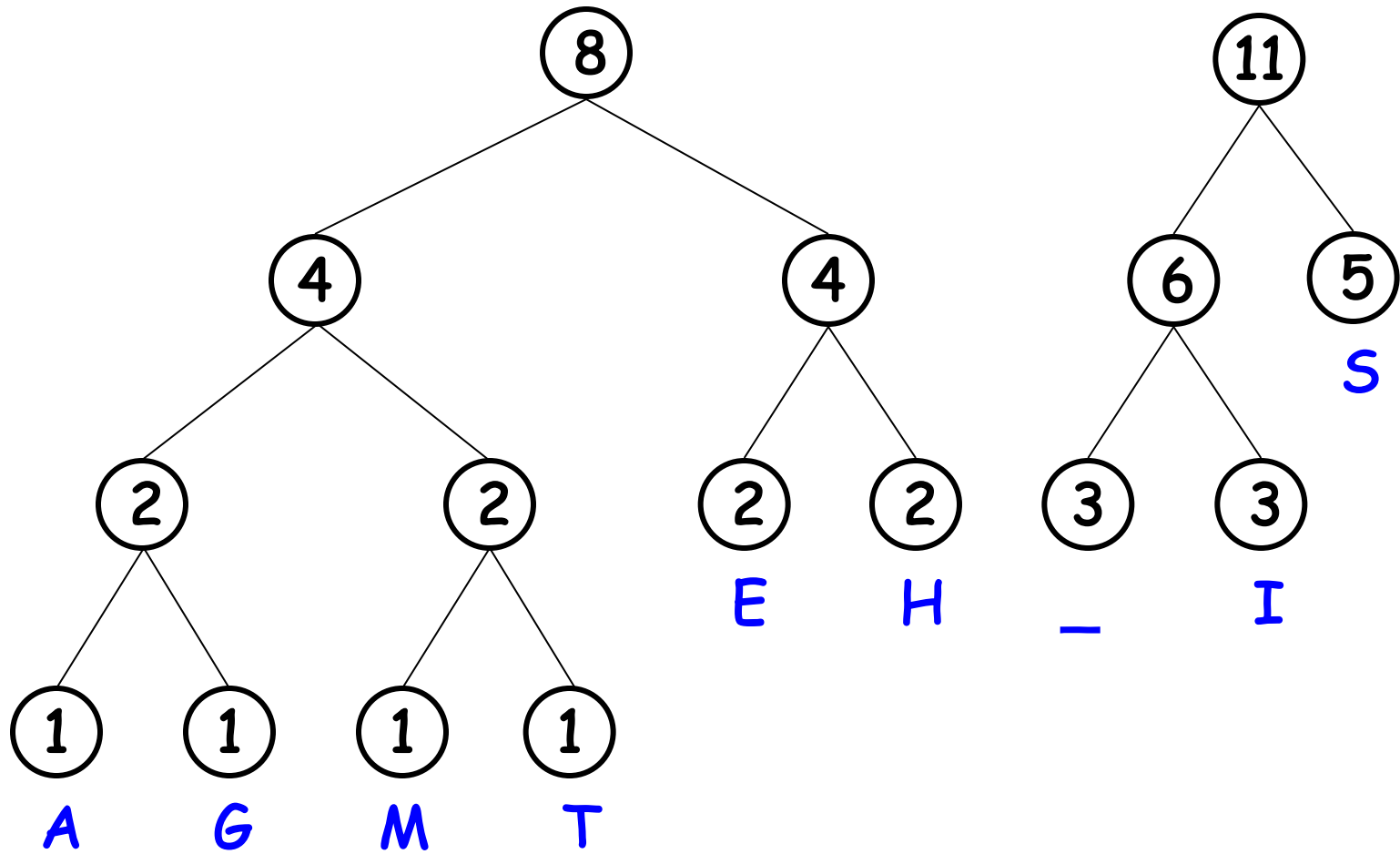
Step 5



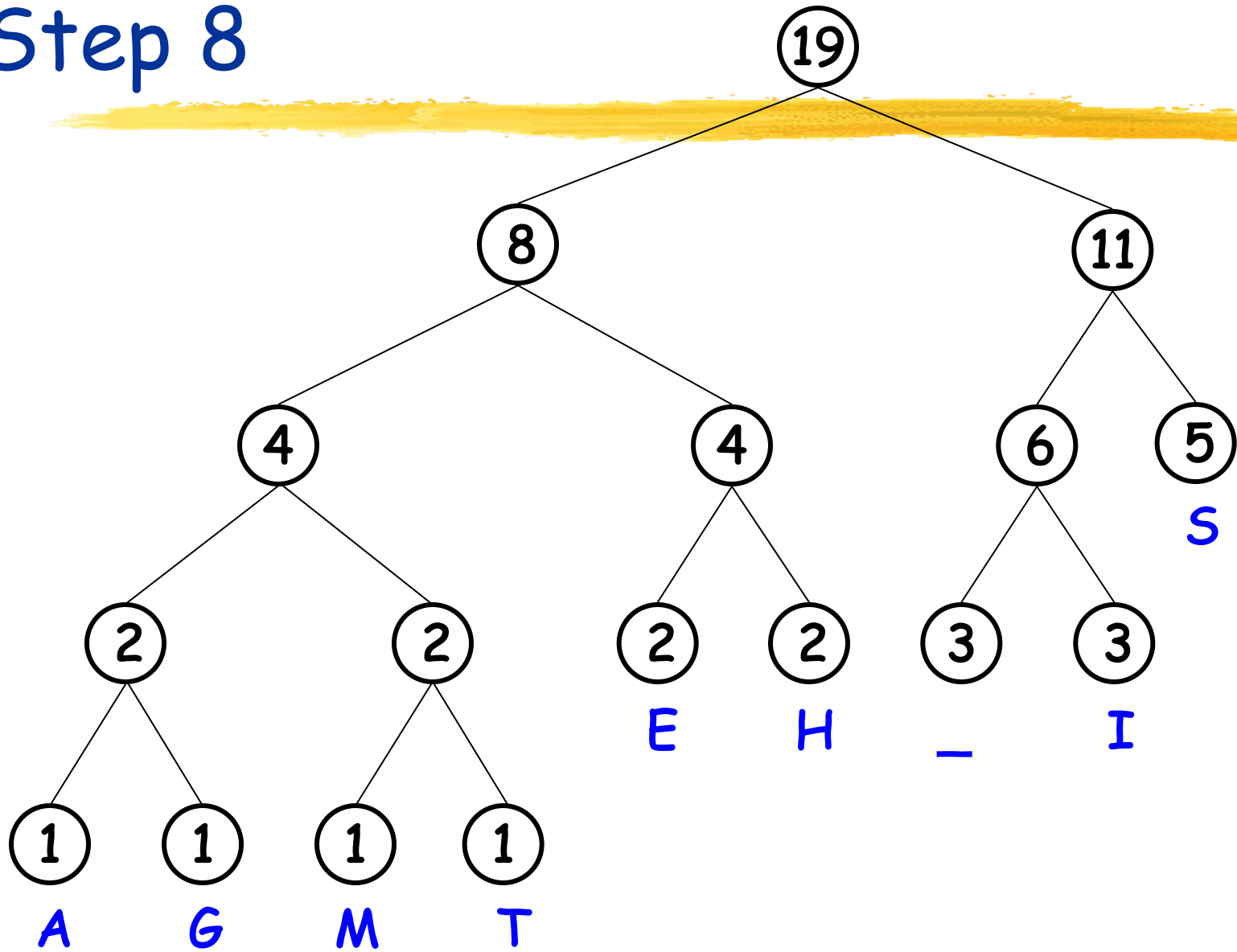
Step 6



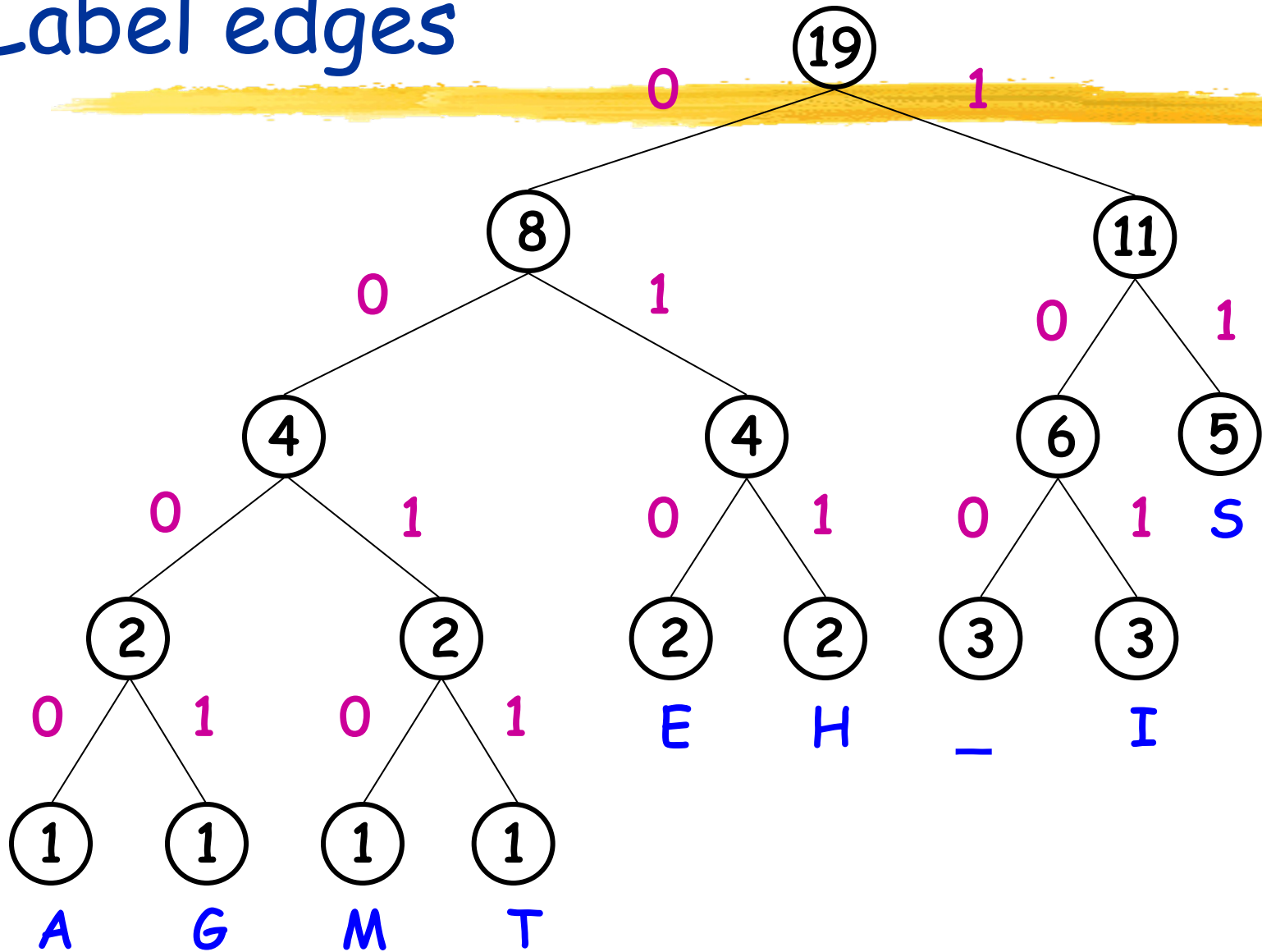
Step 7



Step 8



Label edges



Huffman code & encoded message

This is his message

S	11
E	010
H	011
_	100
I	101
A	0000
G	0001
M	0010
T	0011

00110111011110010111100011101111000010010111100000001010

Huffman code

```
Procedure Huffman(C):
```

```
    // C is the set of n characters and their frequencies
```

```
    n = C.size
```

```
    Q = priority_queue()
```

```
    for i = 1 to n
```

```
        n = node(C[i])
```

```
        Q.push(n)
```

```
    end for
```

```
    while Q.size() is not equal to 1
```

```
        Z = new node()
```

```
        Z.left = x = Q.pop
```

```
        Z.right = y = Q.pop
```

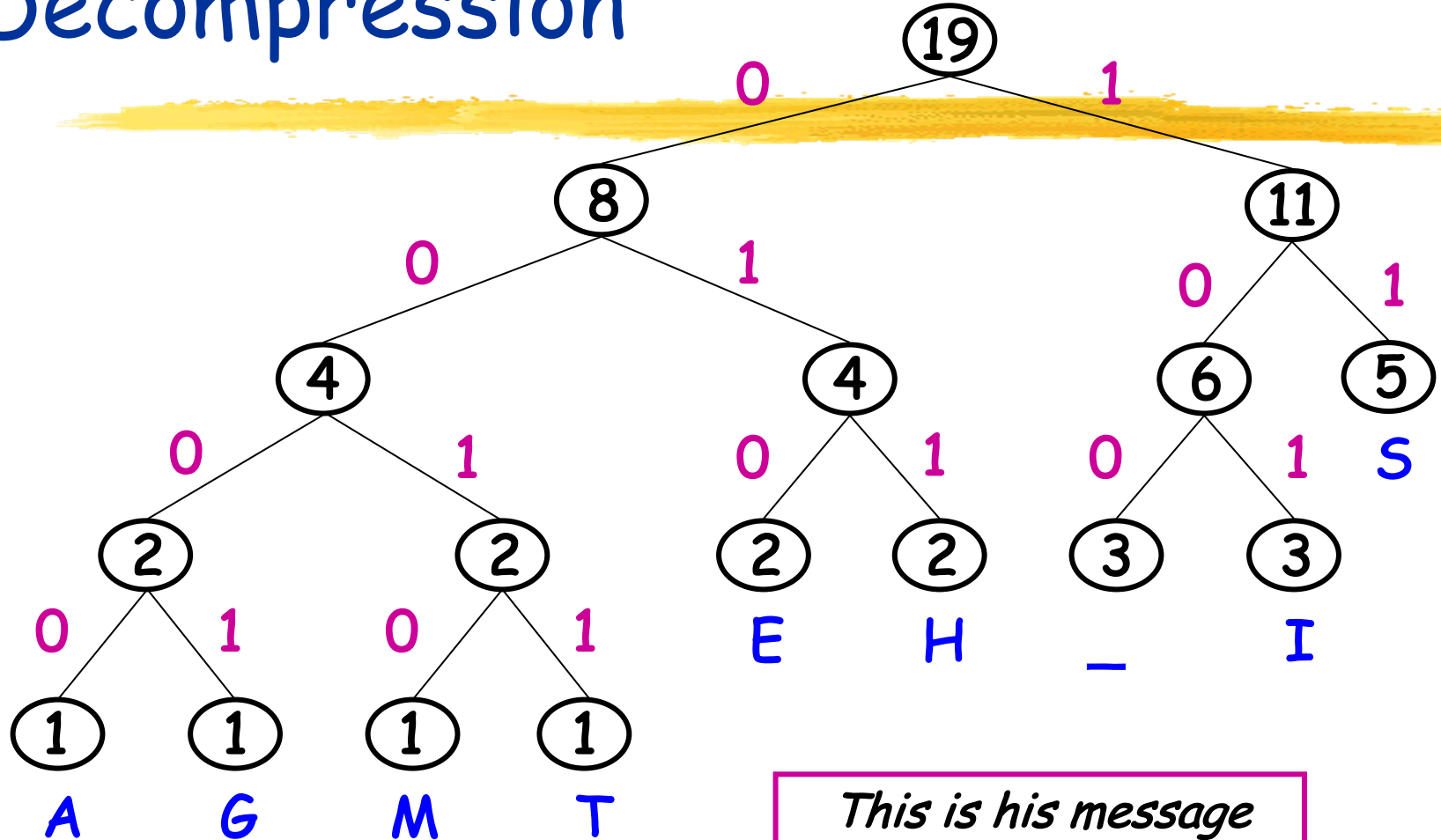
```
        Z.frequency = x.frequency + y.frequency
```

```
        Q.push(Z)
```

```
    end while
```

```
    Return Q
```

Decompression



001101101110010111100011101111000010010111100000001010

Average Bits Per Symbol

Symbol	Frequency	Probability (p_i)	Code	Code (l_i bits)
A	1	1/19=0.0526	0000	4
G	1	0.0526	0001	4
M	1	0.0526	0010	4
T	1	0.0526	0011	4
E	2	0.105	010	3
H	2	0.105	011	3
—	3	0.158	100	3
I	3	0.158	101	3
S	5	0.263	11	2
	19	1.0		

$$\text{Average bits per symbol} = \sum_1^9 p_i l_i$$

Decompression

```
Procedure HuffmanDecompression(root, S):  
    // root represents the root of Huffman Tree  
    // S refers to bit-stream to be decompressed  
    n := S.length  
    for i := 1 to n  
        current = root  
        while current.left != NULL and current.right != NULL  
            if S[i] is equal to '0'  
                current := current.left  
            else  
                current := current.right  
            endif  
            i := i+1  
        endwhile  
        print current.symbol  
    endfor
```

Related References



- ▶ Introduction to the Design and Analysis of Algorithms, A. Levitin, 3rd Ed., Pearson 2011.
 - Chapters 9.4
- ▶ Some slides are based on the ones prepared by Dr Steve Goddard@cse.unl.edu