Analysis and Design of Algorithms

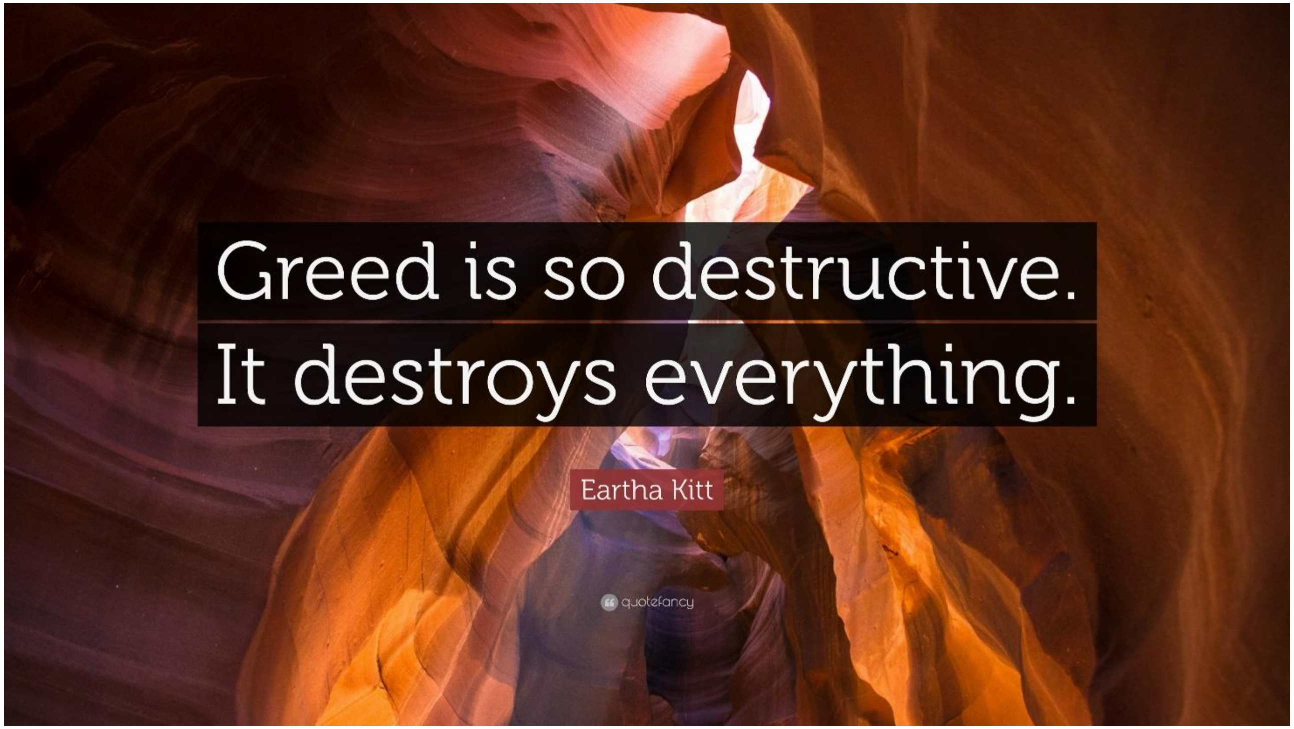
Greedy Algorithms (Part 2):

Counting Money and Huffman Compression

Instructor: **Morteza Zakeri**

Slide by: Hossein Rahmani

Modified by: Morteza Zakeri

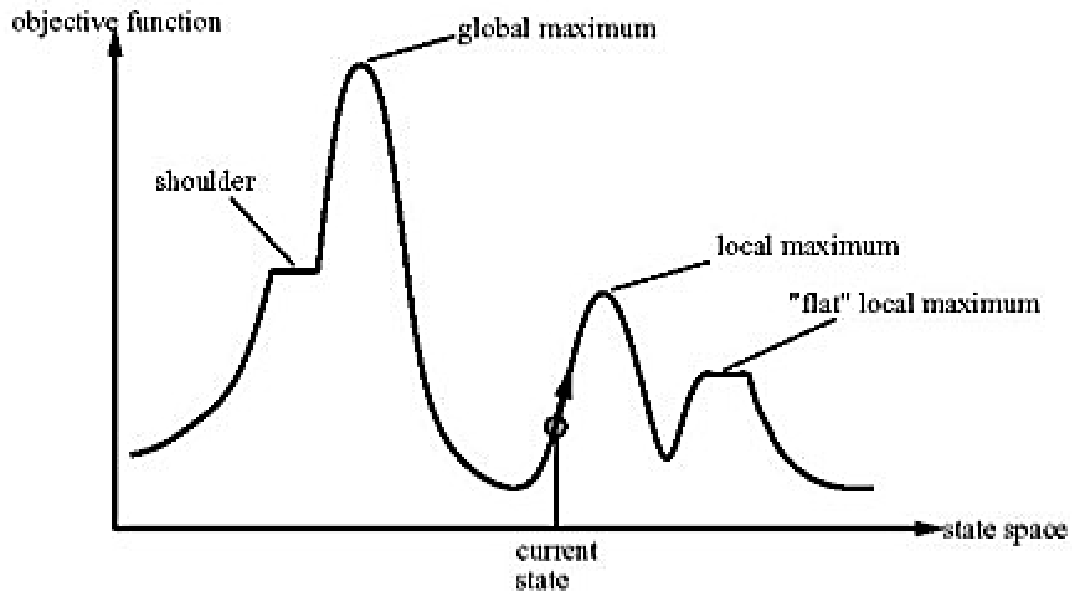

Greed is so destructive.
It destroys everything.
Eartha Kitt
quotefancy


I THINK GREED
SOMETIMES GETS
THE BEST OF
EVERYBODY.
Alan Haft
QUOTEHD.COM

Optimization Problems

- For most optimization problems you want to find, not just *a* solution, but the *best* solution.
- A *greedy algorithm* sometimes works well for optimization problems. It works in phases. At each phase:
  - You take the best you can get right now, without regard for future consequences.
  - You hope that by choosing a *local* optimum at each step, you will end up at a *global* optimum.

Hill Climbing – Some Problems



Example: Counting Money

- Suppose you want to count out a certain amount of money, using the fewest possible bills and coins
- A greedy algorithm to do this would be:

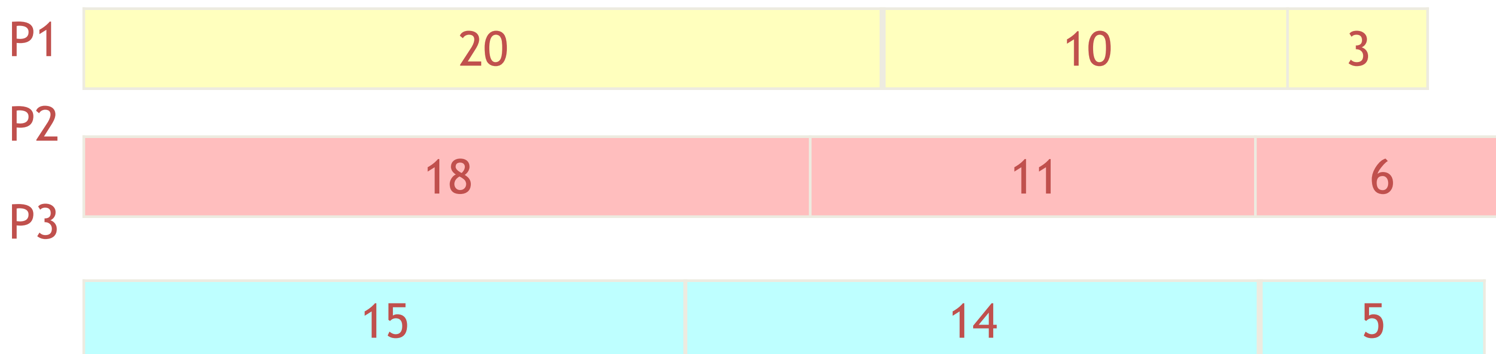
  At each step, take the largest possible bill or coin that does not overshoot
  - Example: To make $6.39, you can choose:
    - a $5 bill
    - a $1 bill, to make $6
    - a 25¢ coin, to make $6.25
    - A 10¢ coin, to make $6.35
    - four 1¢ coins, to make $6.39
- For US money, the greedy algorithm always gives the optimum solution

Greedy Algorithm Failure

- In some (fictional) monetary system, "krons"
- come in 1 kron, 7 kron, and 10 kron coins
- Using a greedy algorithm to count out 15 krons, you would get
  - A 10 kron piece
  - Five 1 kron pieces, for a total of 15 krons
  - This requires six coins
- A better solution would be to use two 7 kron pieces and one 1 kron piece
  - This only requires three coins
- **The greedy algorithm results in a solution, but NOT in an optimal solution**

A Scheduling Problem

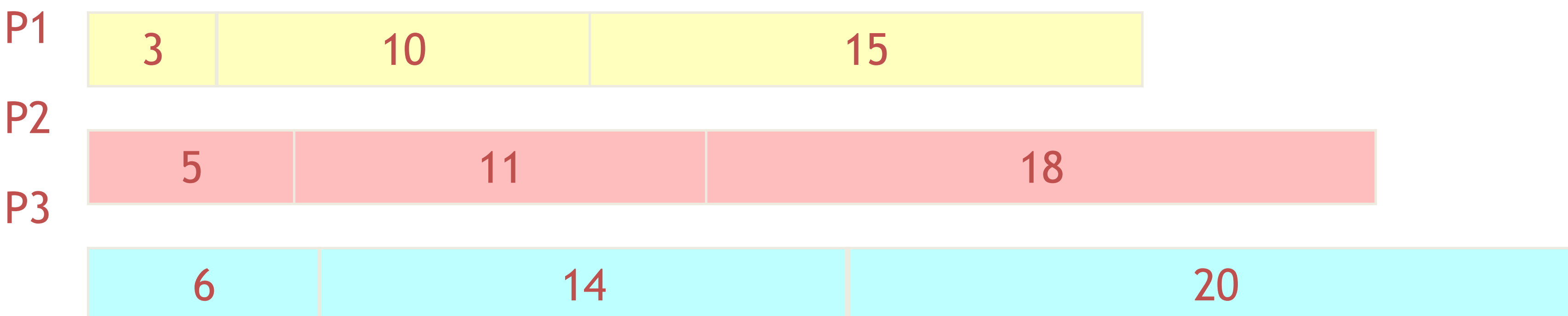
- You have to run nine jobs, with running times of 3, 5, 6, 10, 11, 14, 15, 18, and 20 minutes.
- You have three processors on which you can run these jobs.
- You decide to do the longest-running jobs first, on whatever processor is available.



- Time to completion: 18 + 11 + 6 = 35 minutes
- This solution isn't bad, but we might be able to do better

Another Approach

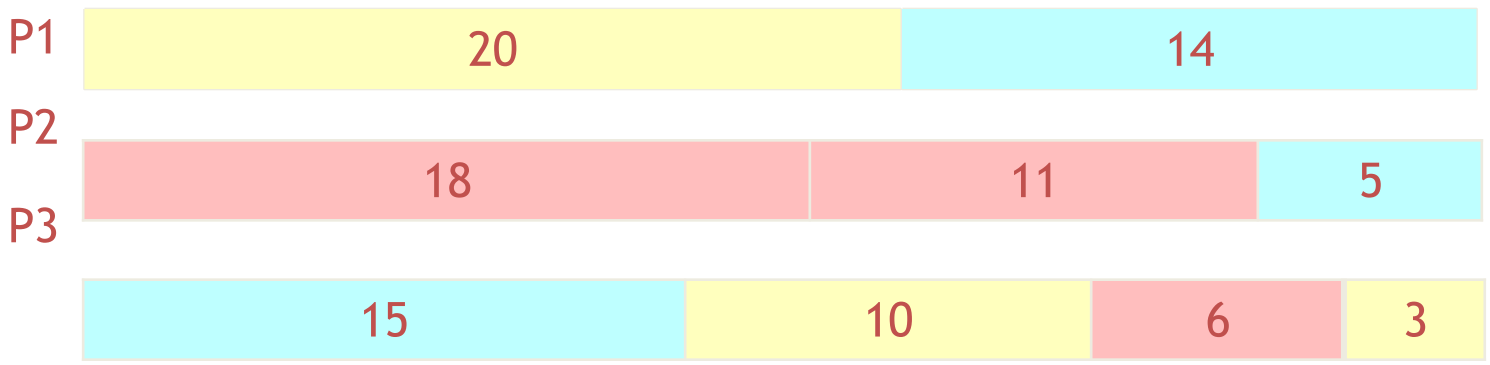
- What would be the result if you ran the *shortest* job first?
- Again, the running times are 3, 5, 6, 10, 11, 14, 15, 18, and 20 minutes



- That wasn't such a good idea; time to completion is now
  6 + 14 + 20 = 40 minutes
- Note, however, that the greedy algorithm itself is fast
  - All we had to do at each stage was pick the minimum or maximum

An Optimum Solution

- Better solutions do exist:



- How do we find such a solution?
  - One way: Try all possible assignments of jobs to processors
  - Unfortunately, this approach can take exponential time

Compression

- Definition
  - Reduce size of data
    (number of bits needed to represent data)
- Benefits
  - Reduce storage needed
  - Reduce transmission cost / bandwidth

Sources of Compressibility

- Redundancy
  - Recognize repeating patterns
  - Exploit using
    - Dictionary
    - Variable length encoding
- Human perception
  - Less sensitive to some information
  - Can discard less important data

Types of Compression

- Lossless
  - Preserves all information
  - Exploits redundancy in data
  - Applied to general data
- Lossy
  - May lose some information
  - Exploits redundancy & human perception
  - Applied to audio, image, video

Effectiveness of Compression

- Metrics
  - Bits per byte (8 bits)
    - 2 bits / byte $\Rightarrow$ ¼ original size
    - 8 bits / byte $\Rightarrow$ no compression
  - Percentage
    - 75% compression $\Rightarrow$ ¼ original size

Effectiveness of Compression

- Depends on data
  - Random data $\Rightarrow$ hard
    - Example: 1001110100 $\Rightarrow$?
  - Organized data $\Rightarrow$ easy
    - Example: 1111111111 $\Rightarrow$ 1×10
- Corollary
  - No universally best compression algorithm

Effectiveness of Compression

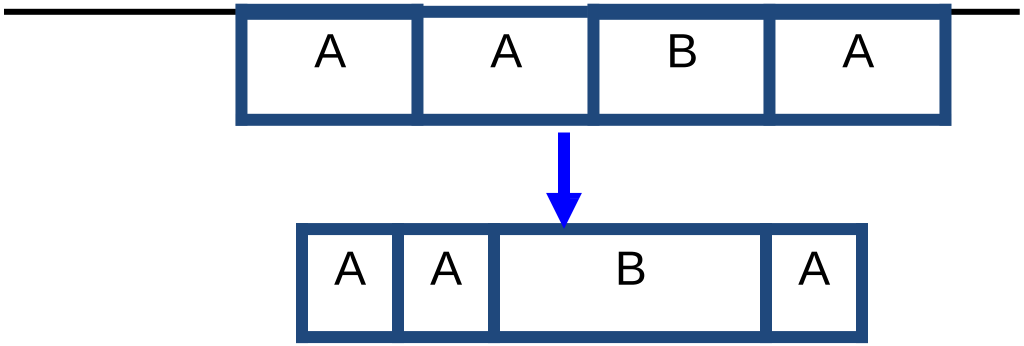
- Lossless Compression is not always possible
  - If compression is always possible (alternative view)
    - Compress file (reduce size by 1 bit)
    - Recompress output
    - Repeat (until we can store data with 0 bits)

Lossless Compression Techniques

- LZW (Lempel-Ziv-Welch) compression
  - Build pattern dictionary
  - Replace patterns with index into dictionary
- Run length encoding
  - Find & compress repetitive sequences
- Huffman codes
  - Use variable length codes based on frequency

Huffman Code

- Approach
  - Variable length encoding of symbols
  - Exploit statistical frequency of symbols
  - Efficient when symbol probabilities vary widely
- Principle
  - Use fewer bits to represent frequent symbols
  - Use more bits to represent infrequent symbols



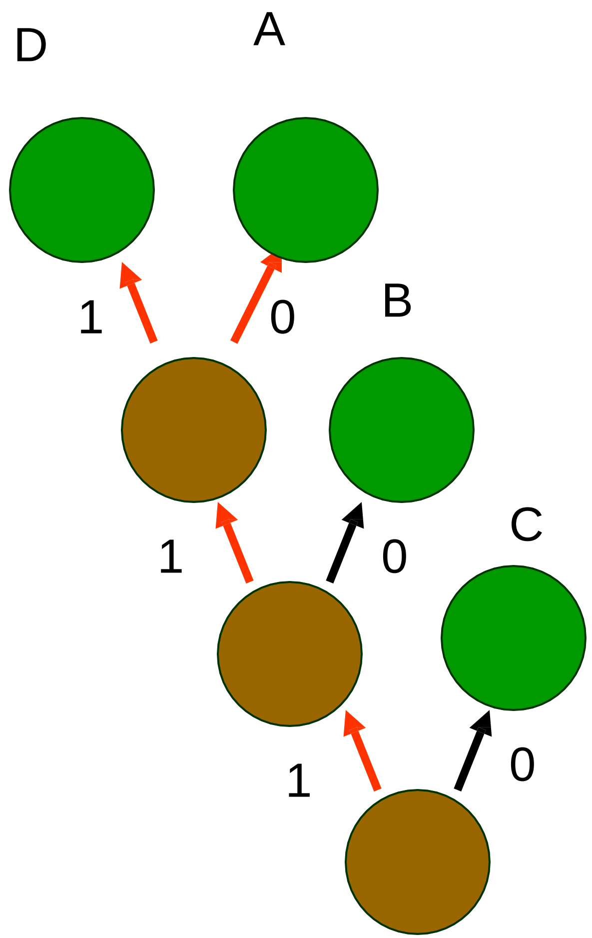
Huffman Code Example

Symbol	A	B	C	D
Frequency	13%	25%	50%	12%
Original Encoding	00	01	10	11
	2 bits	2 bits	2 bits	2 bits
Huffman Encoding	110	10	0	111
	3 bits	2 bits	1 bit	3 bits

- Expected size
  - Original $\Rightarrow$ $1/8 \times 2 + 1/4 \times 2 + 1/2 \times 2 + 1/8 \times 2 = 2$ bits / symbol
  - Huffman $\Rightarrow$ $1/8 \times 3 + 1/4 \times 2 + 1/2 \times 1 + 1/8 \times 3 = 1.75$ bits / symbol

Huffman Code Data Structures

- Binary (Huffman) tree
  - Represents Huffman code
  - Edge ⇒ code (0 or 1)
  - Leaf ⇒ symbol
  - Path to leaf ⇒ encoding
  - Example
    - A = “110”, B = “10”, C = “0”



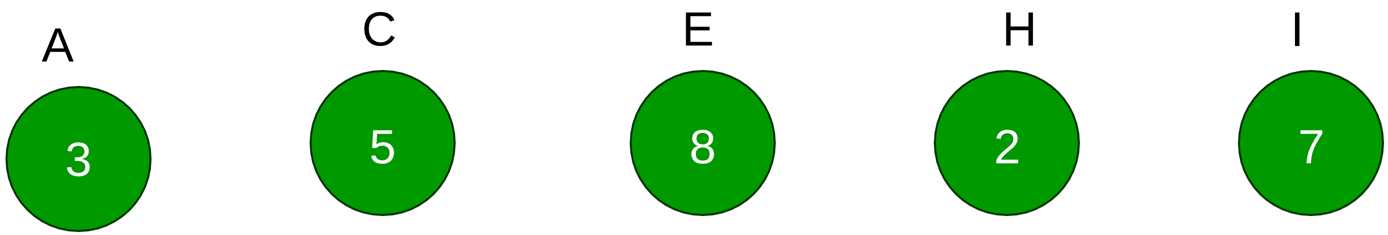
Huffman Code Algorithm Overview

- Encoding
  - Calculate frequency of symbols in file
  - Create binary tree representing “best” encoding
  - Use binary tree to encode compressed file
    - For each symbol, output path from root to leaf
    - Size of encoding = length of path
  - Save binary tree

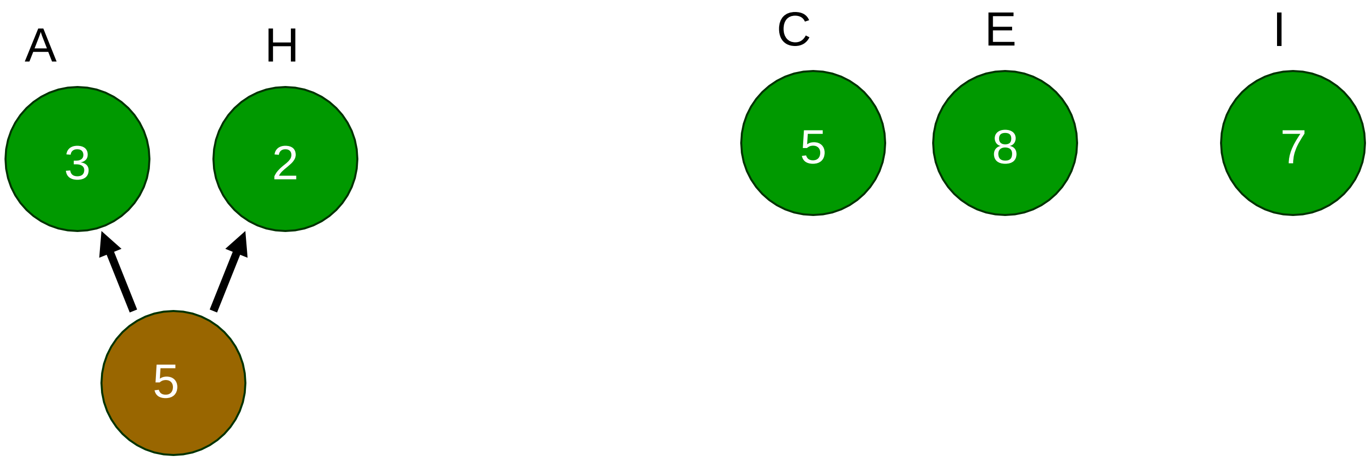
Huffman Code – Creating Tree

- Algorithm
  - Place each <u>symbol</u> in <u>leaf</u>
    - <u>Weight</u> of leaf = symbol <u>frequency</u>
  - Select two trees L and R (initially leafs)
    - Such that L, R have <u>lowest frequencies</u> in tree
  - Create new (internal) node
    - Left child $\Rightarrow$ L
    - Right child $\Rightarrow$ R
    - New <u>frequency $\Rightarrow$ frequency(L) + frequency(R)</u>
  - <u>Repeat</u> until all nodes <u>merged</u> into one tree

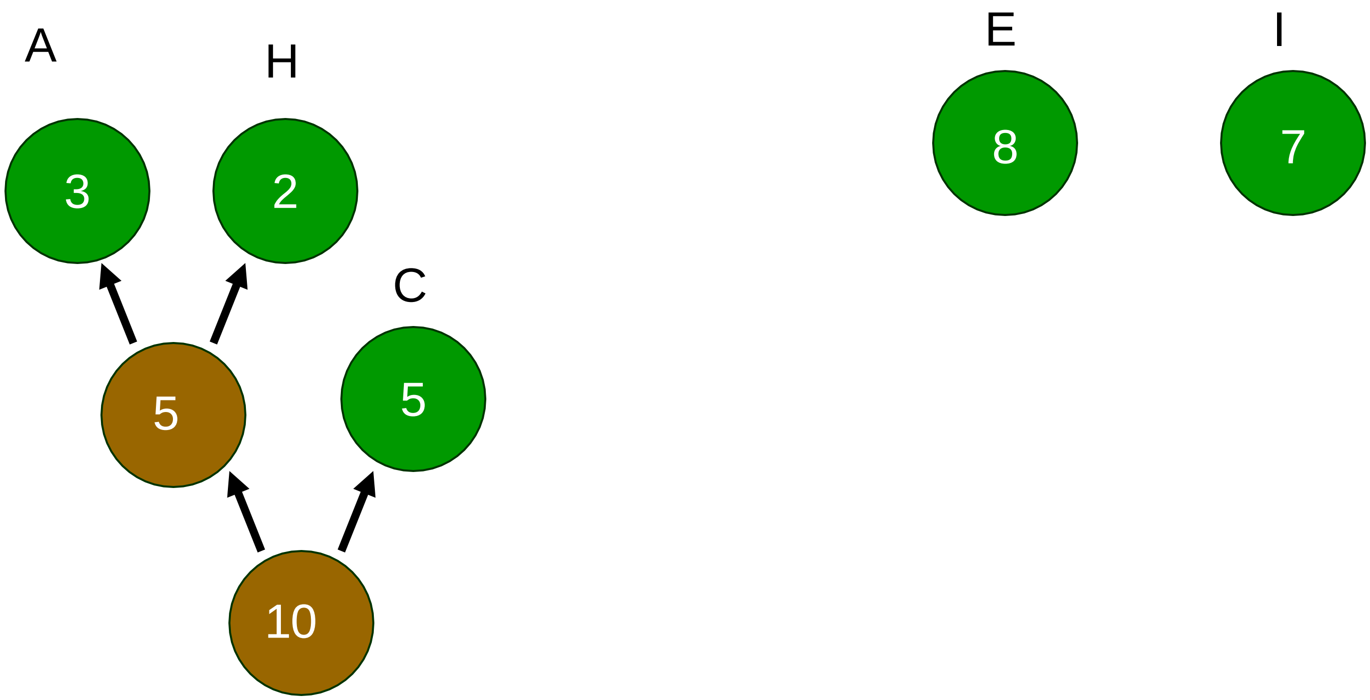
Huffman Tree Construction 1



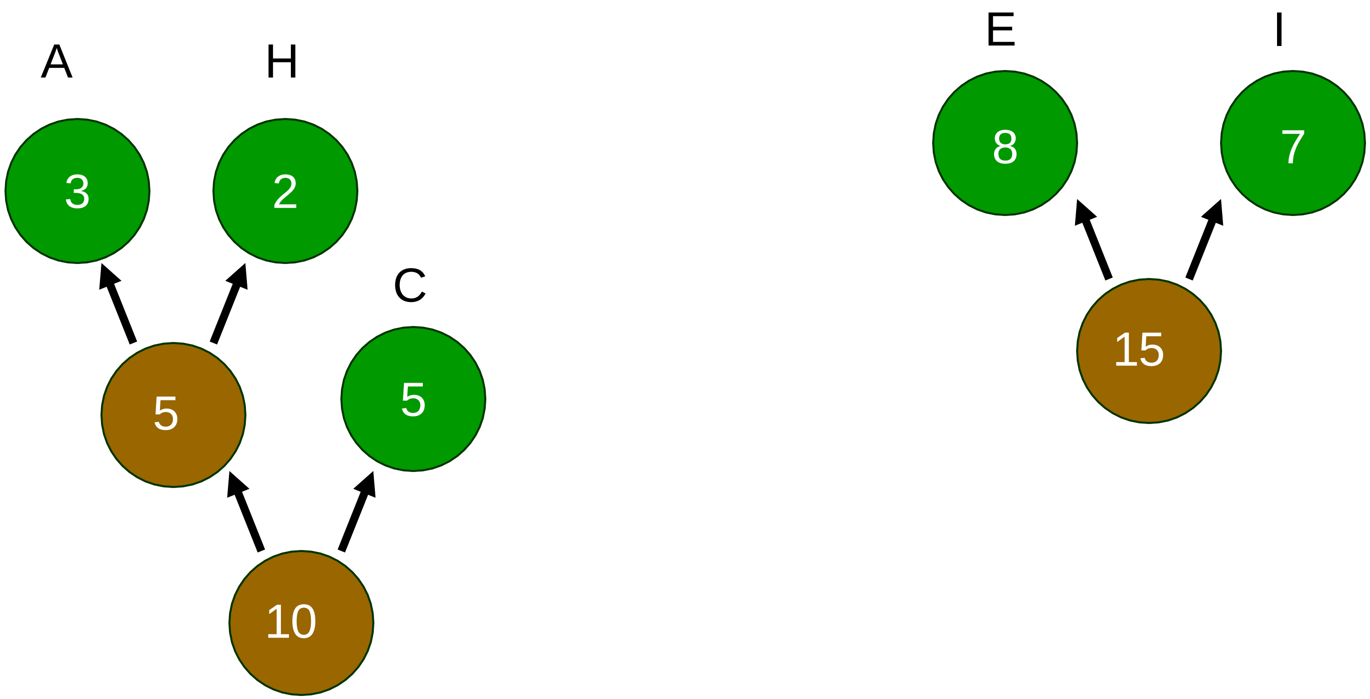
Huffman Tree Construction 2



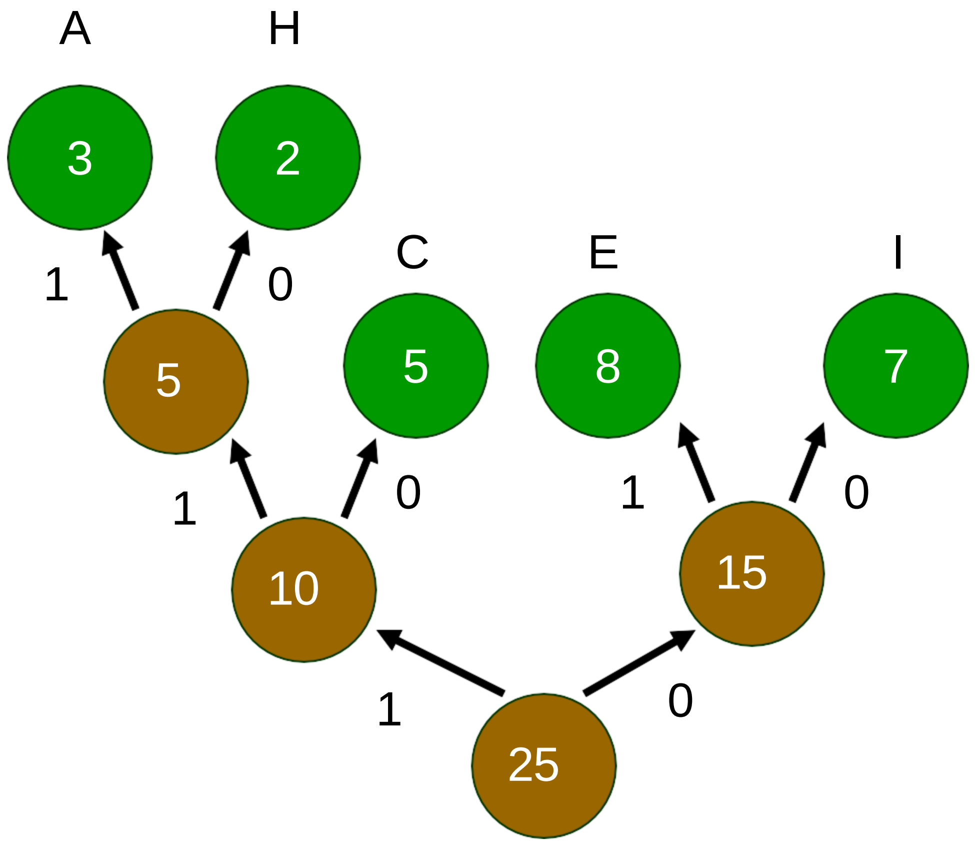
Huffman Tree Construction 3



Huffman Tree Construction 4



Huffman Tree Construction 5



E	=	01
I	=	00
C	=	10
A	=	111
H	=	110

Huffman Coding Example

- Huffman code

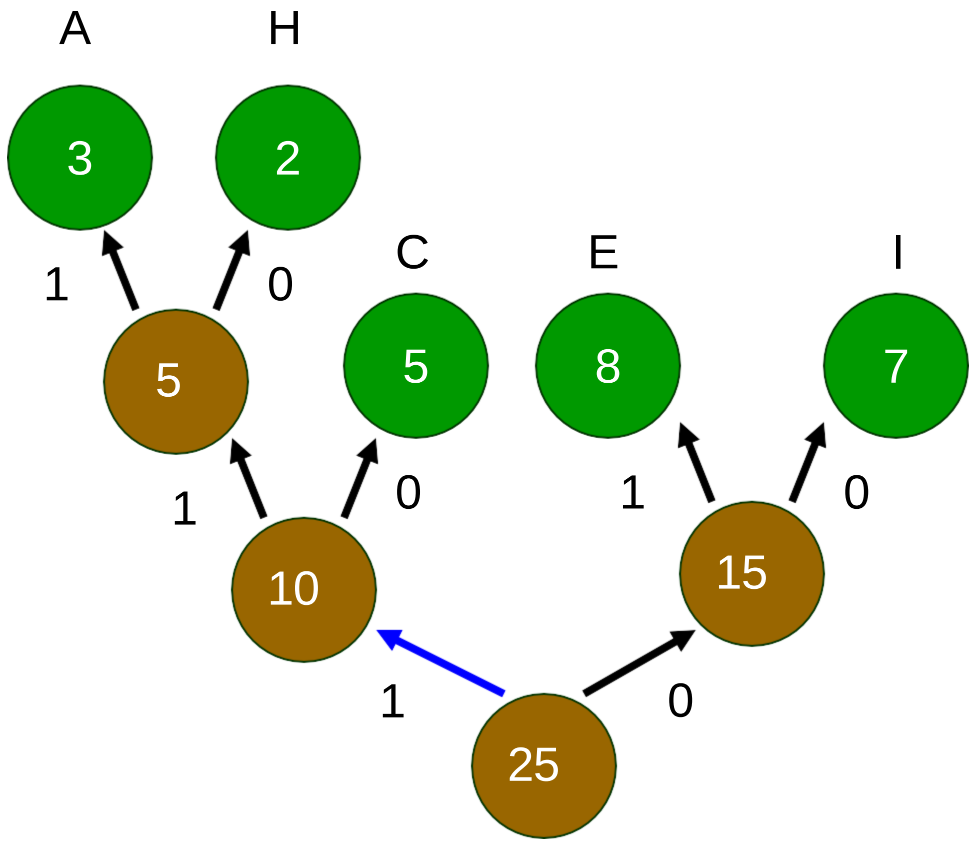
E	=	01
I	=	00
C	=	10
A	=	111
H	=	110

- Input
  - ACE
- Output
  - (111)(10)(01) = 1111001

Huffman Code Algorithm Overview

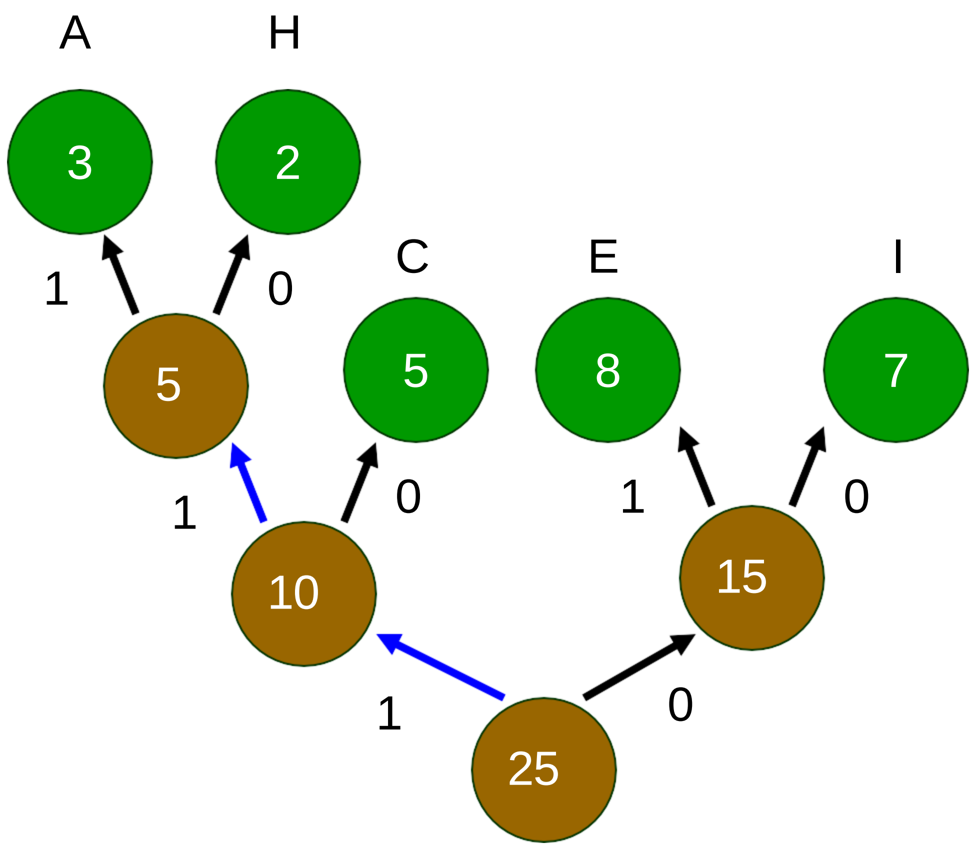
- Decoding
  - Read compressed file & binary tree
  - Use binary tree to decode file
    - Follow path from root to leaf

Huffman Decoding 1



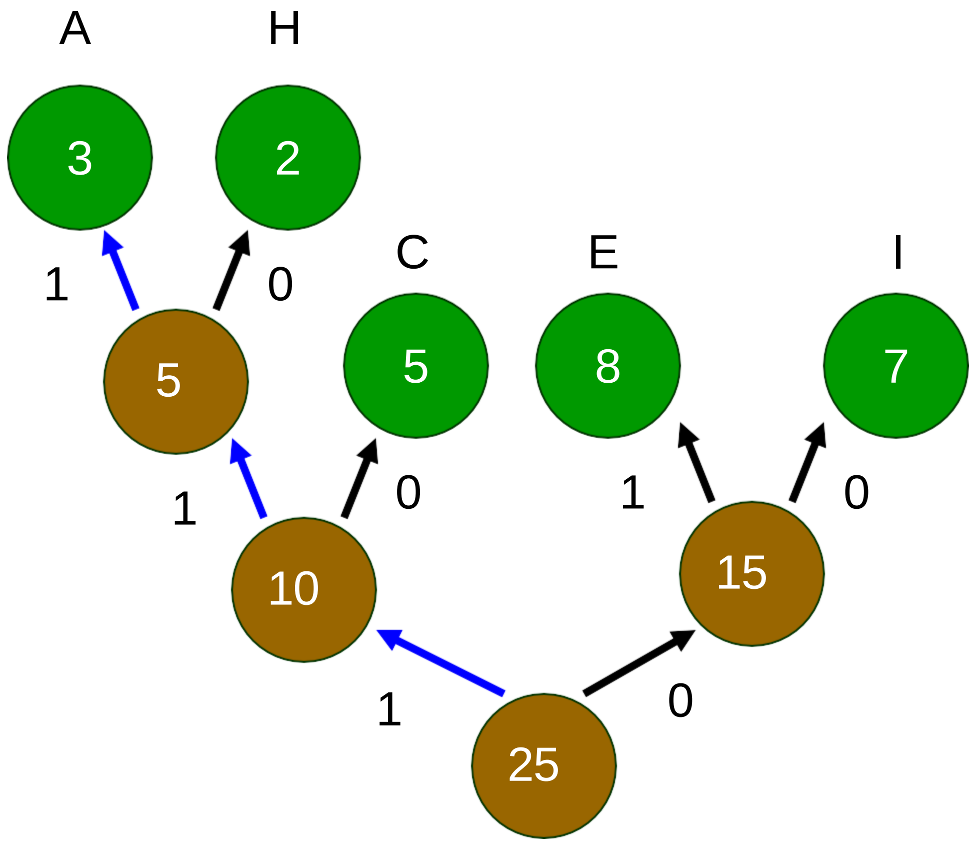
1111001

Huffman Decoding 2



1111001

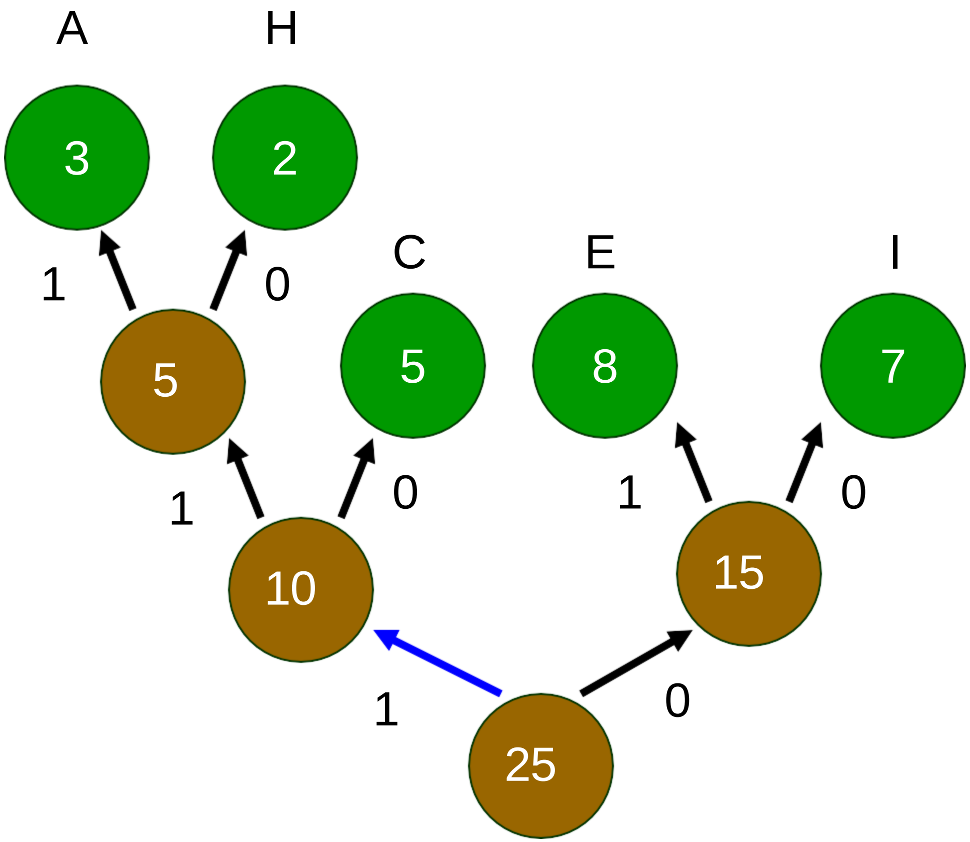
Huffman Decoding 3



1111001

A

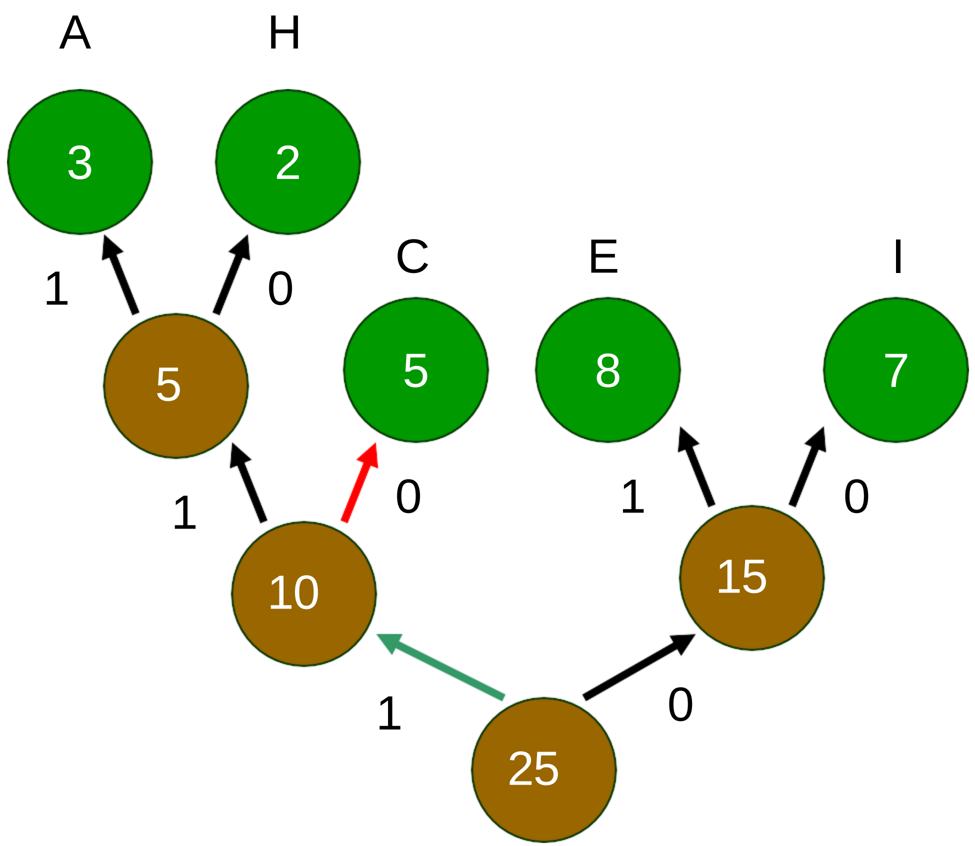
Huffman Decoding 4



1111001

A

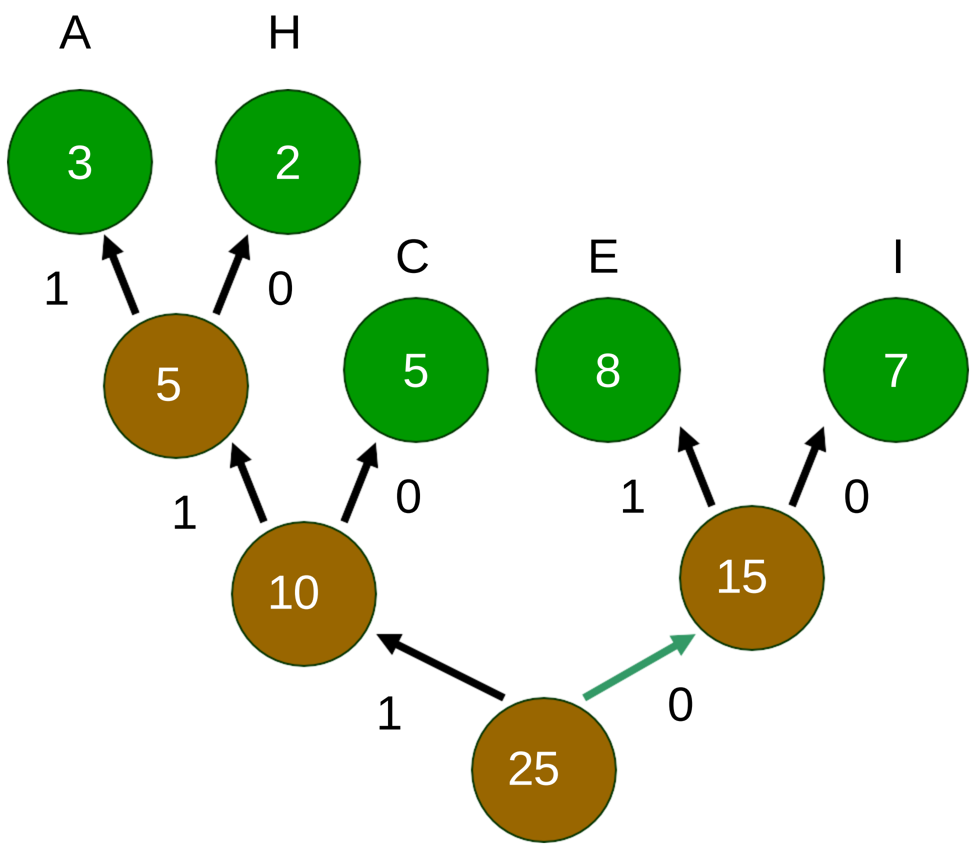
Huffman Decoding 5



1111001

AC

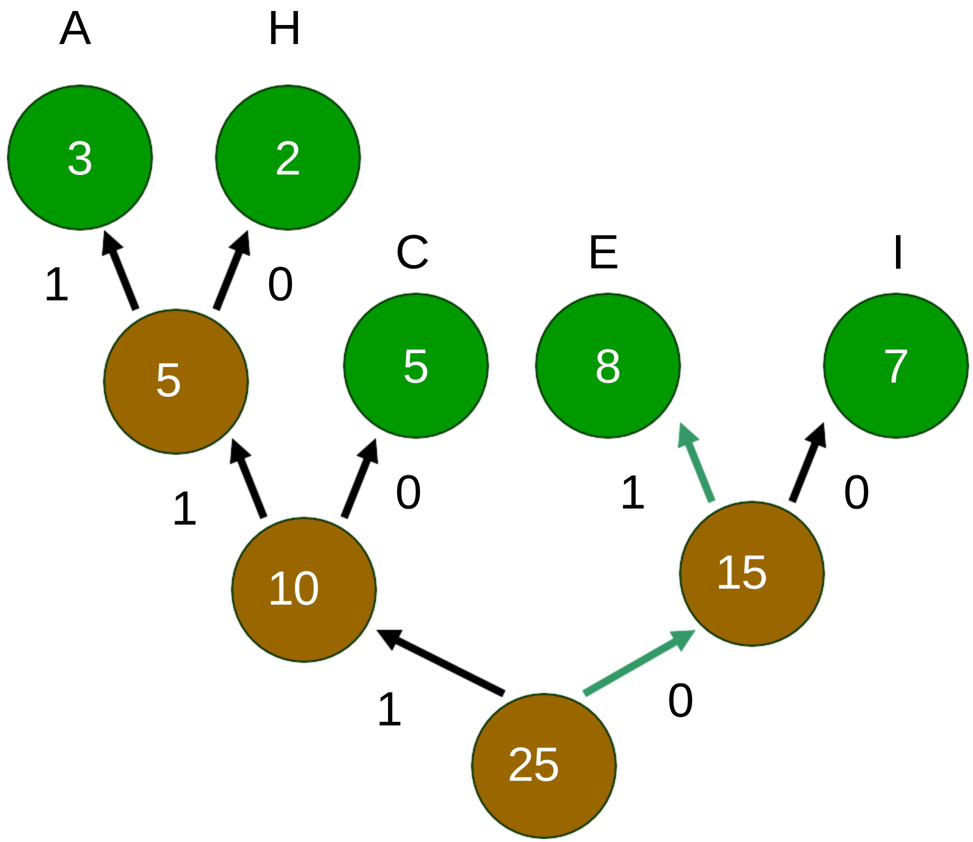
Huffman Decoding 6



1111001

AC

Huffman Decoding 7



1111001

ACE

Huffman Code Properties

- Prefix code
  - No code is a prefix of another code
  - Example
    - Huffman("I") $\Rightarrow$ 00
    - Huffman("X") $\Rightarrow$ 001 // not legal prefix code
  - Can stop as soon as complete code found
  - No need for end-of-code marker
- Nondeterministic
  - Multiple Huffman coding possible for same input
  - If more than two trees with same minimal weight

Huffman Code Properties

- Greedy algorithm
  - Chooses best local solution at each step
  - Combines 2 trees with lowest frequency
- Still yields overall best solution
  - Optimal prefix code
  - Based on statistical frequency
- Better compression possible (depends on data)
  - Using other approaches (e.g., pattern dictionary)

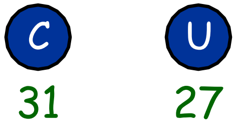
Huffman Code Construction

- Character count in text.
- Character Encoding?

Char	Freq
E	125
T	93
A	80
O	76
I	73
N	71
S	65
R	61
H	55
L	41
D	40
C	31
U	27

Huffman Code Construction

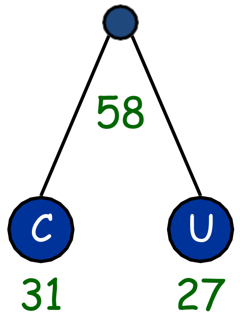
Char	Freq
E	125
T	93
A	80
O	76
I	73
N	71
S	65
R	61
H	55
L	41
D	40
C	31
U	27



Huffman Code Construction

Char	Freq
E	125
T	93
A	80
O	76
I	73
N	71
S	65
R	61
	58
H	55
L	41
D	40

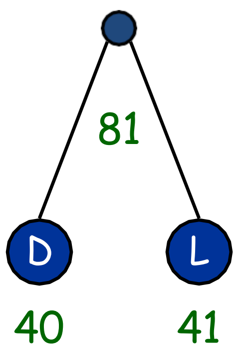
C	31
U	27

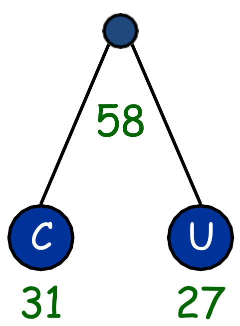


Huffman Code Construction

Char	Freq
E	125
T	93
	81
A	80
O	76
I	73
N	71
S	65
R	61
	58
H	55

L	41
D	40

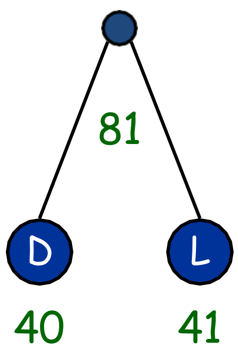


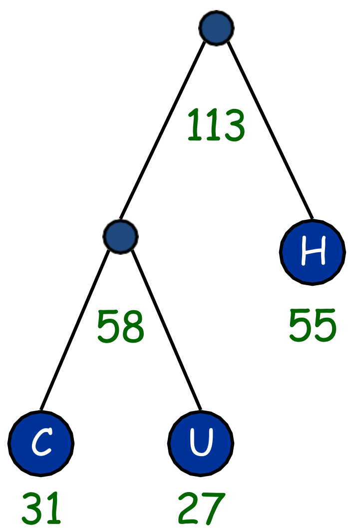


Huffman Code Construction

Char	Freq
E	125
	113
T	93
	81
A	80
O	76
I	73
N	71
S	65
R	61

	58
H	55

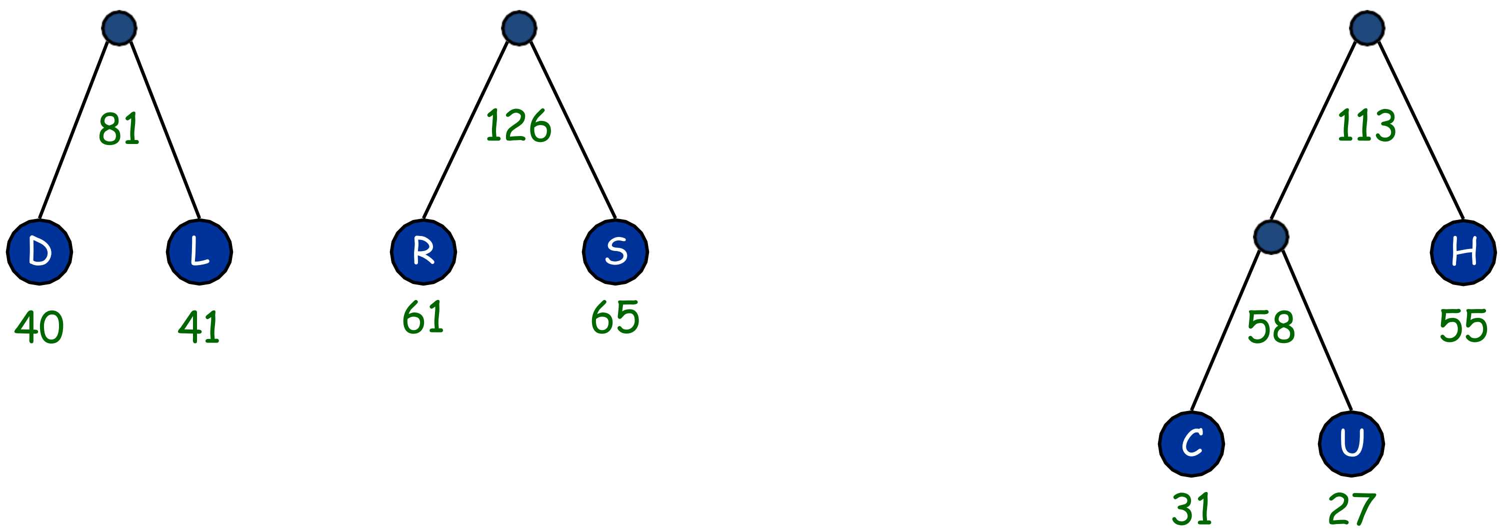




Huffman Code Construction

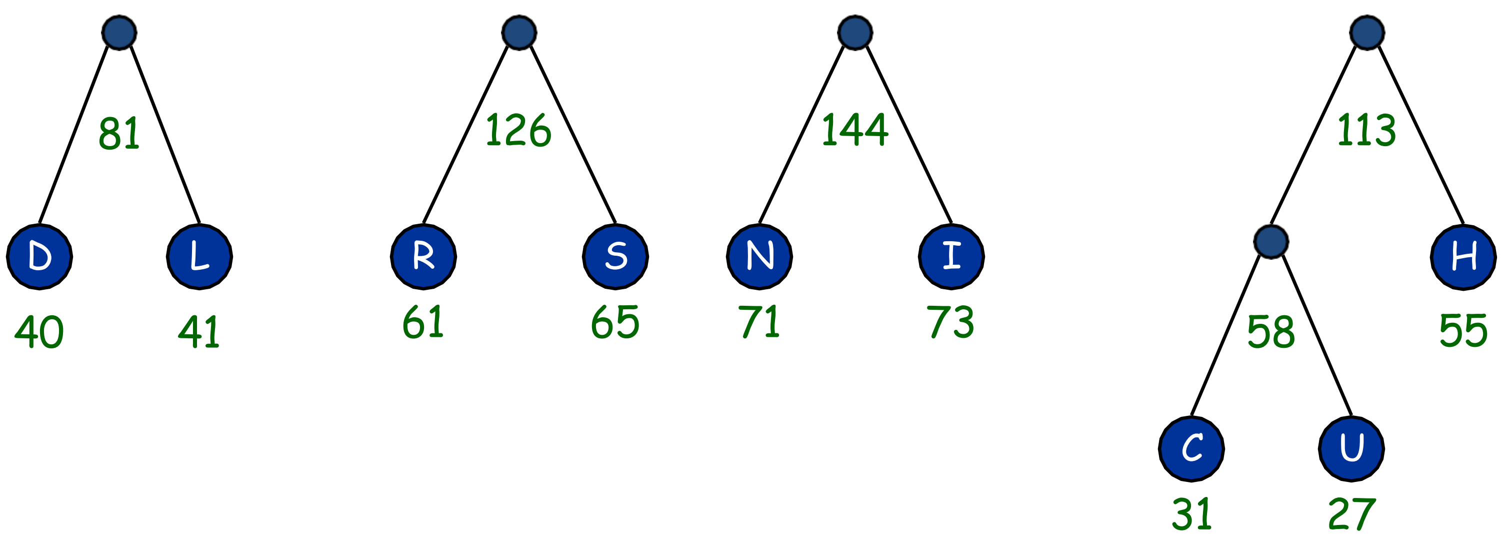
Char	Freq
	126
E	125
	113
T	93
	81
A	80
O	76
I	73
N	71

S	65
R	61



Huffman Code Construction

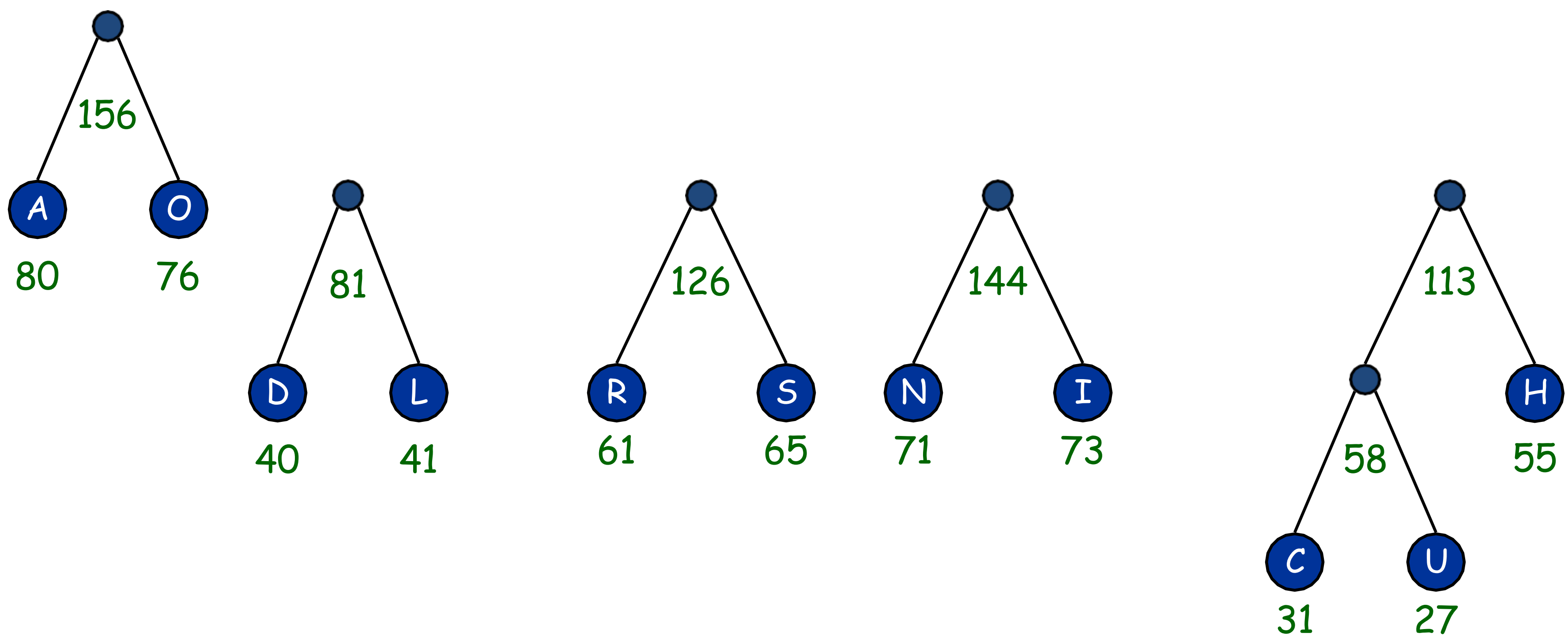
Char	Freq
	144
	126
E	125
	113
T	93
	81
A	80
O	76
I	73
N	71



Huffman Code Construction

Char	Freq
	156
	144
	126
E	125
	113
T	93
	81

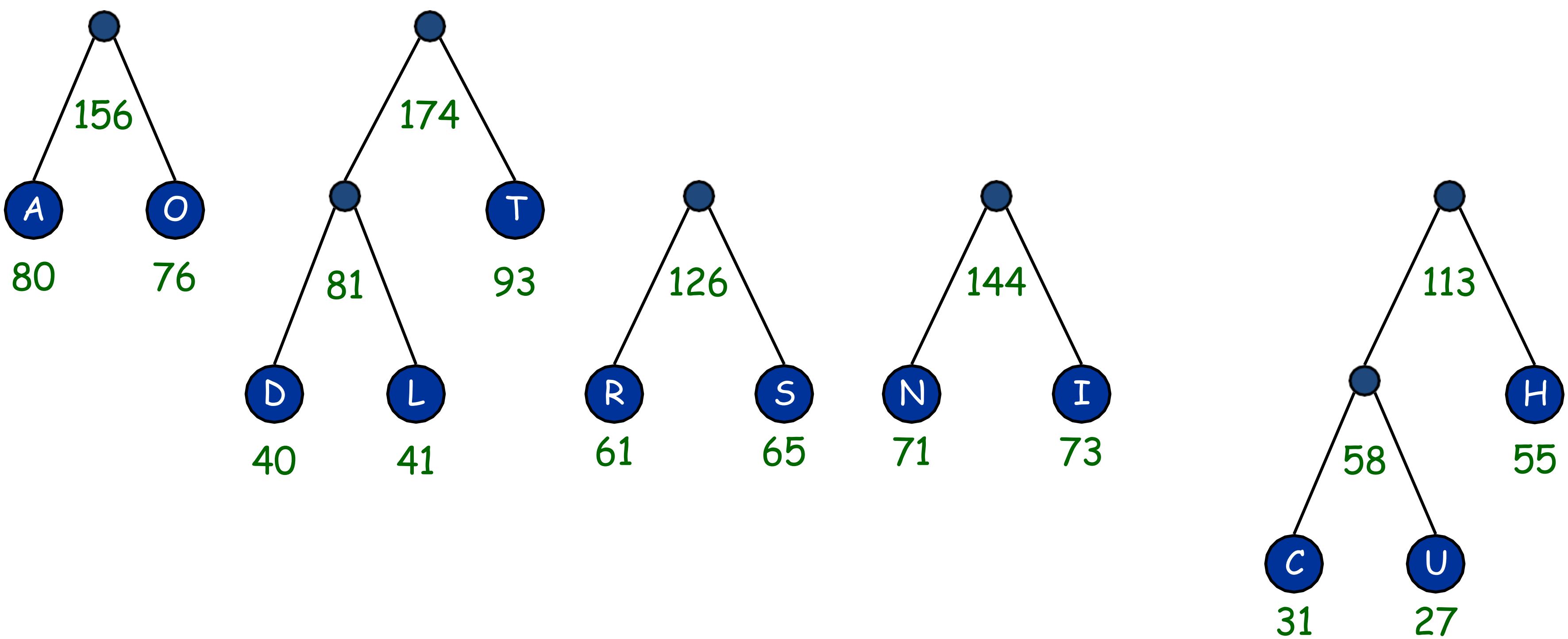
A	80
O	76



Huffman Code Construction

Char	Freq
	174
	156
	144
	126
E	125
	113

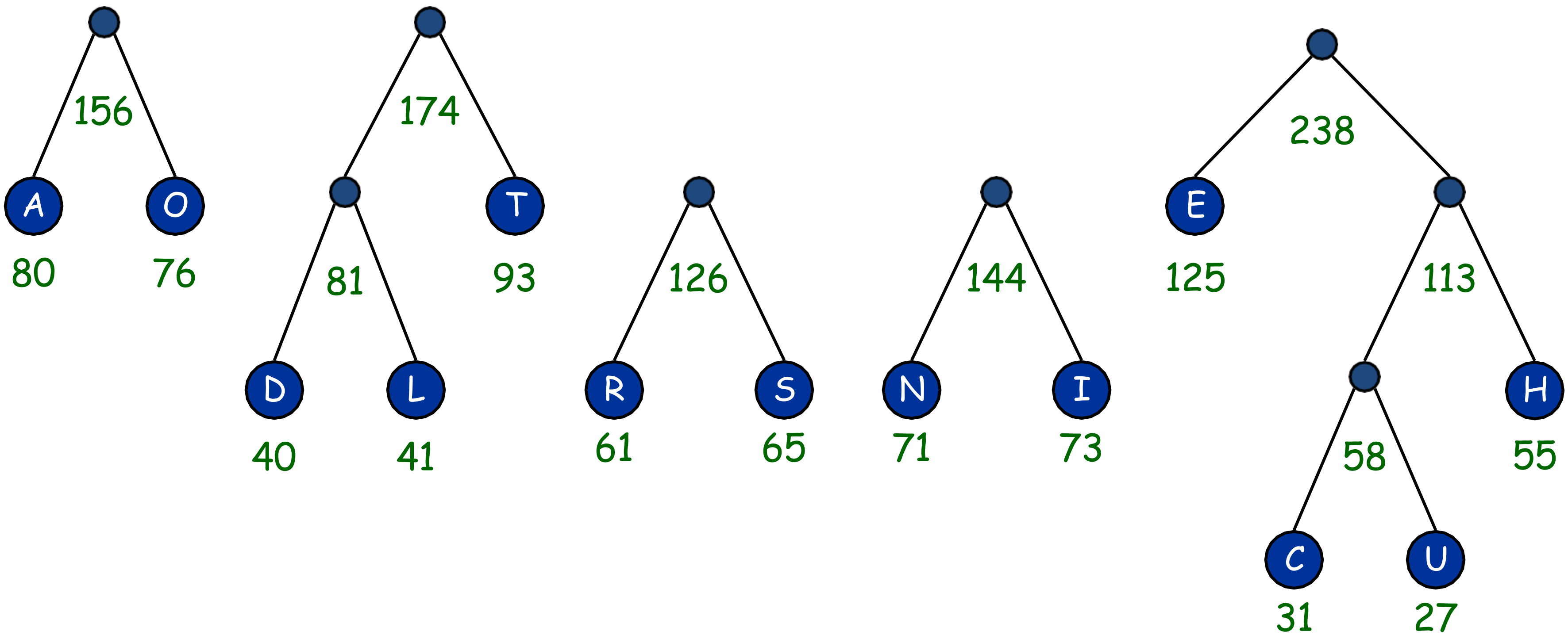
T	93
	81



Huffman Code Construction

Char	Freq
	238
	174
	156
	144
	126

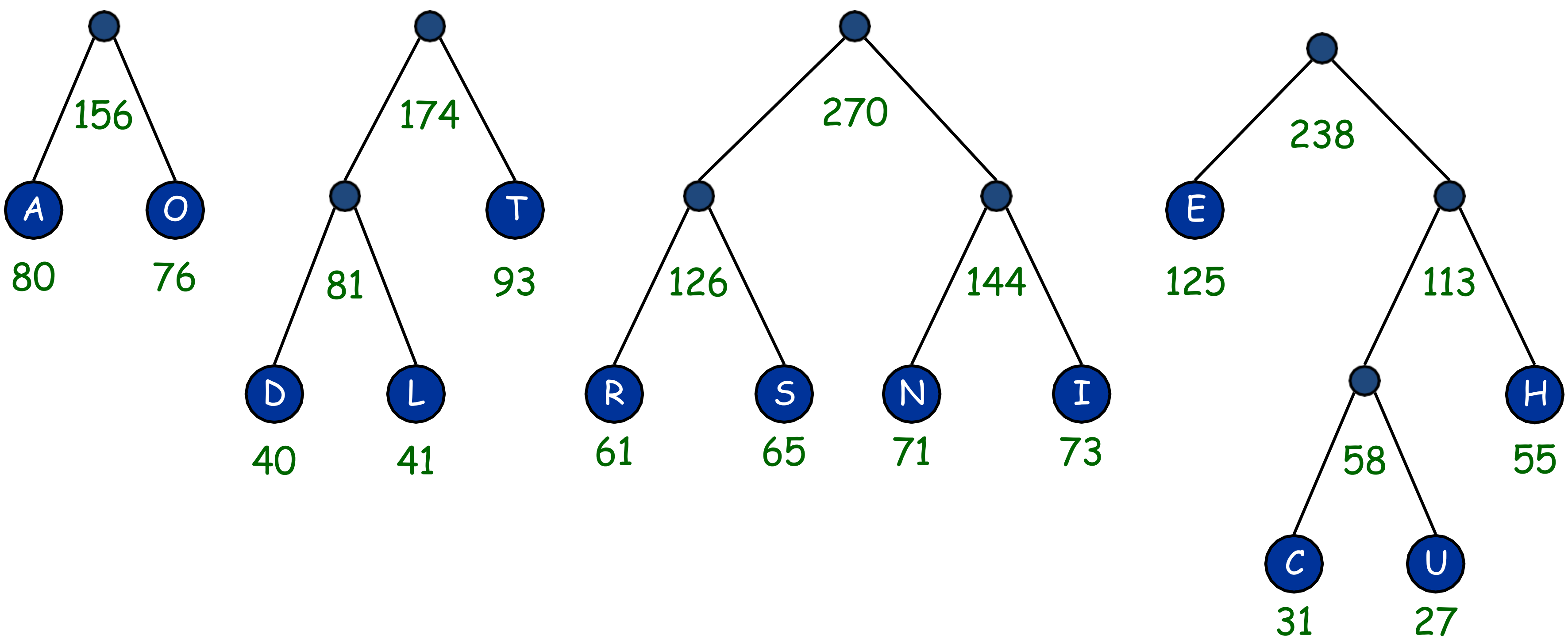
E	125
	113



Huffman Code Construction

Char	Freq
	270
	238
	174
	156

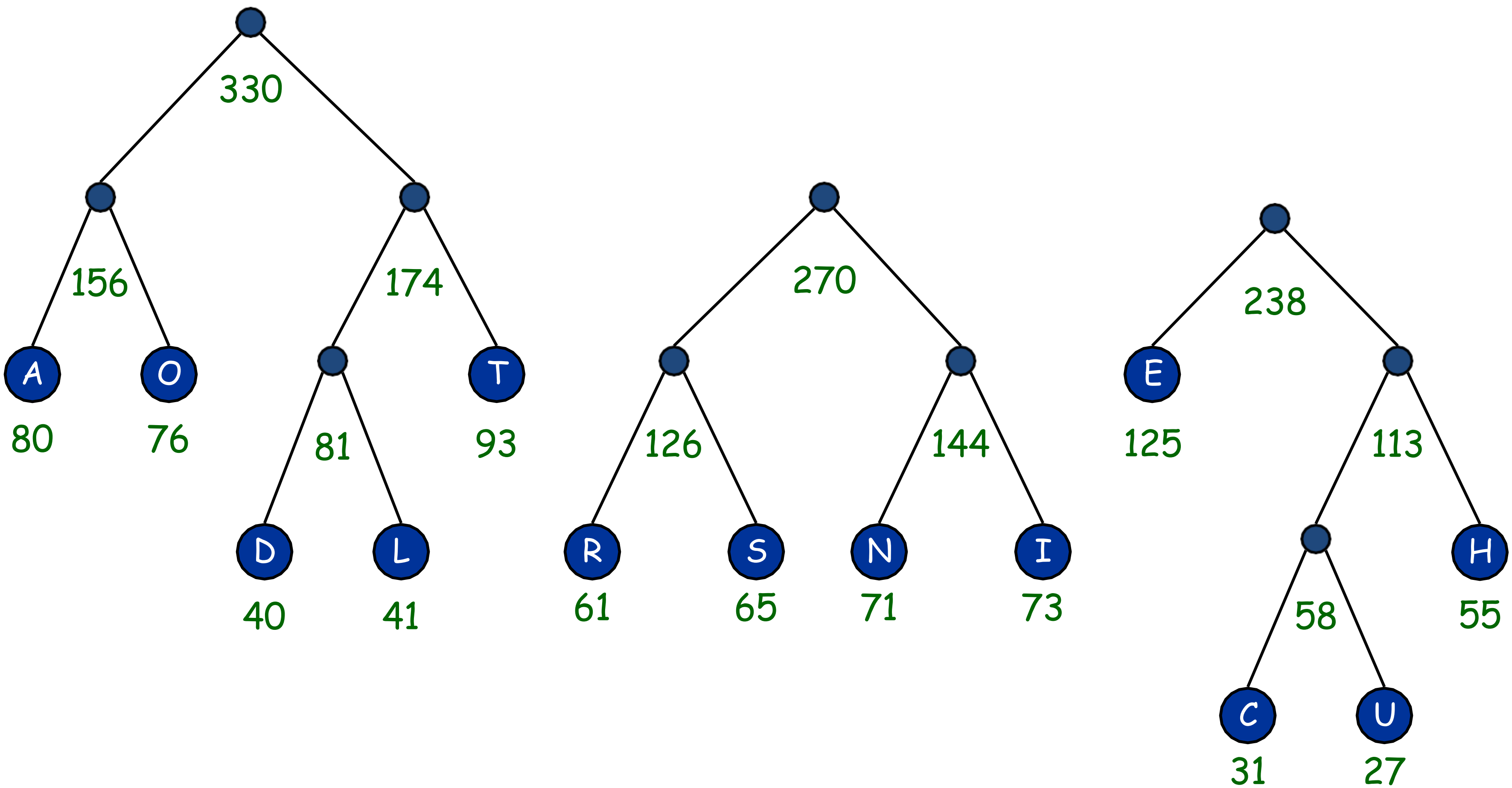
	144
	126



Huffman Code Construction

Char	Freq
	330
	270
	238

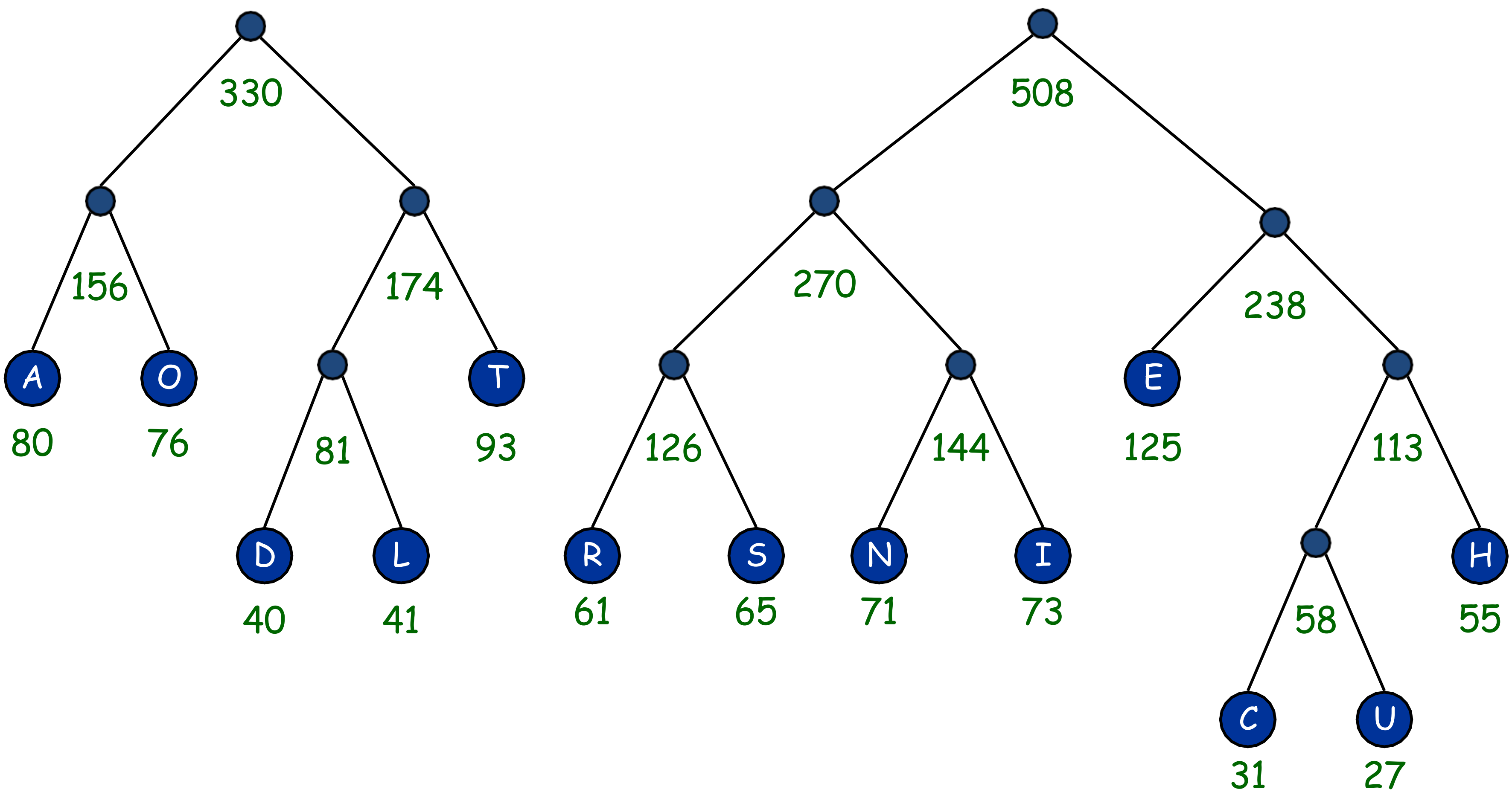
	174
	156



Huffman Code Construction

Char	Freq
	508
	330

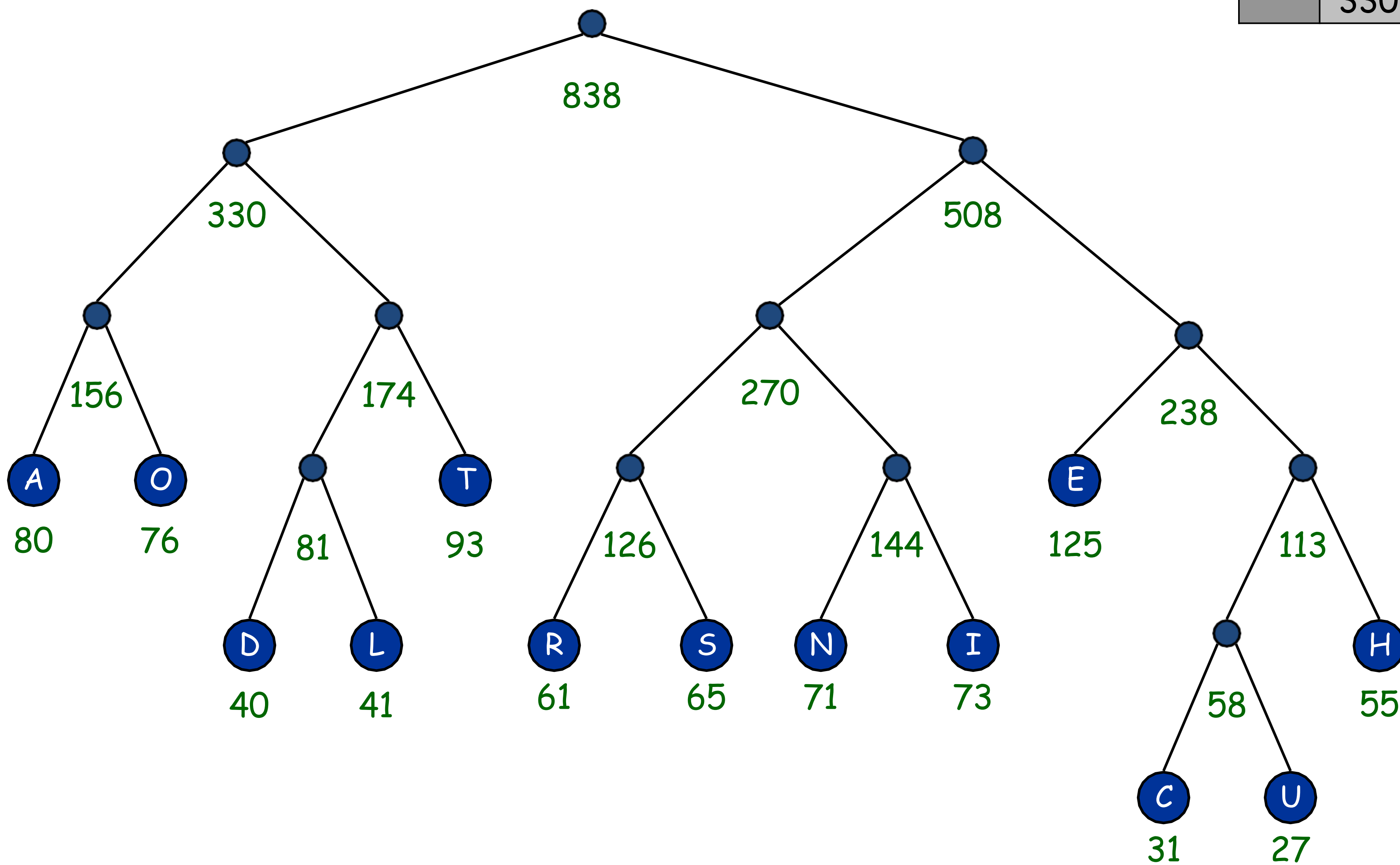
	270
	238



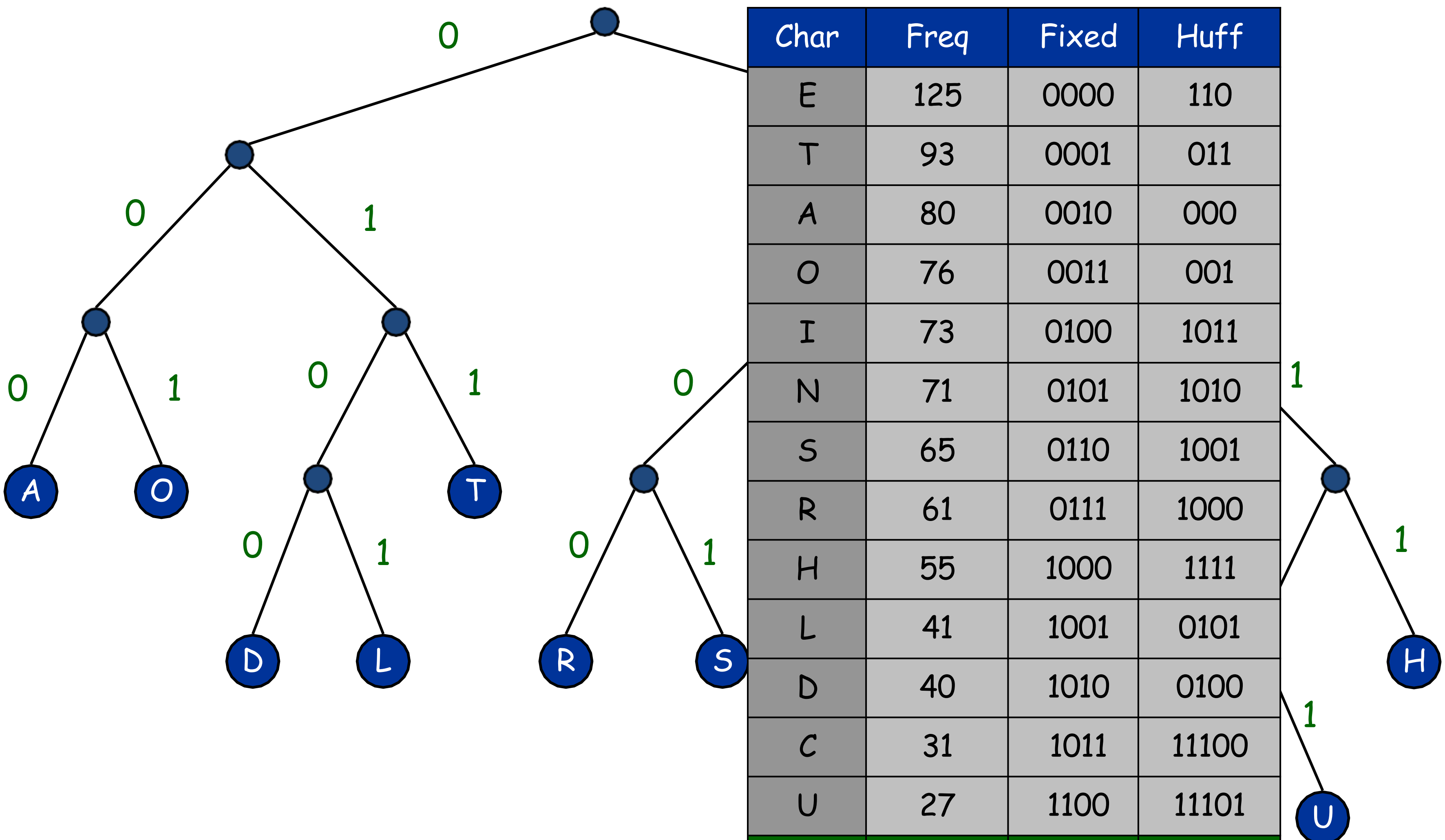
Huffman Code Construction

Char	Freq
	838

	508
	330



Huffman Code Construction



Char	Freq	Fixed	Huff
E	125	0000	110
T	93	0001	011
A	80	0010	000
O	76	0011	001
I	73	0100	1011
N	71	0101	1010
S	65	0110	1001
R	61	0111	1000
H	55	1000	1111
L	41	1001	0101
D	40	1010	0100
C	31	1011	11100
U	27	1100	11101
Total	838	4.00	3.62