

CMSC 132: Object-Oriented Programming II



Compression & Huffman Codes

Department of Computer Science
University of Maryland, College Park

Overview

■ Compression

- Examples
- Sources
- Types
- Effectiveness

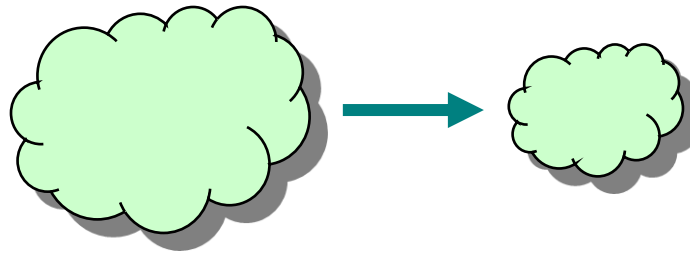
■ Huffman Code

- Properties
- Huffman tree (encoding)
- Decoding

Compression

■ Definition

- Reduce size of data
(number of bits needed to represent data)



■ Benefits

- Reduce storage needed
- Reduce transmission cost / latency / bandwidth

Compression Examples

■ Formats

■ General

■ .zip, .rar

■ Images

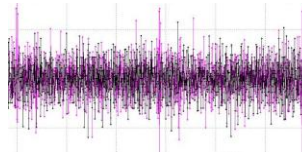
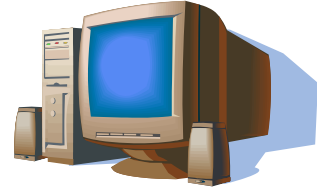
■ .jpg, .gif

■ Audio

■ .mp3, .wmv

■ Video

■ .mpg, .mov



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
 - Some lossless audio formats (e.g., FLAC)

■ Lossy

- May lose some information
- Exploits **redundancy & human perception**
- Applied to audio, image, video, multimedia

Effectiveness of Compression

■ Metrics

■ Bits per byte (8 bits)

- 2 bits / byte $\Rightarrow$ $\frac{1}{4}$ original size
- 8 bits / byte $\Rightarrow$ no compression

■ Percentage

- 75% compression $\Rightarrow$ $\frac{1}{4}$ original size

Effectiveness of Compression

■ Depends on data

■ Random data $\Rightarrow$ hard

■ Example: 1001110100 $\Rightarrow$?

■ Organized data $\Rightarrow$ easy

■ Example: 1111111111 $\Rightarrow 1 \times 10$

■ Corollary

■ No universally best compression algorithm

Effectiveness of Compression

■ Lossless Compression is not guaranteed

■ Pigeonhole principle

- Reduce size 1 bit $\Rightarrow$ can only store $\frac{1}{2}$ of data

- Example

 - 000, 001, 010, 011, 100, 101, 110, 111 $\Rightarrow$ 00, 01, 10, 11

■ 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 code**
 - Use variable length codes based on frequency

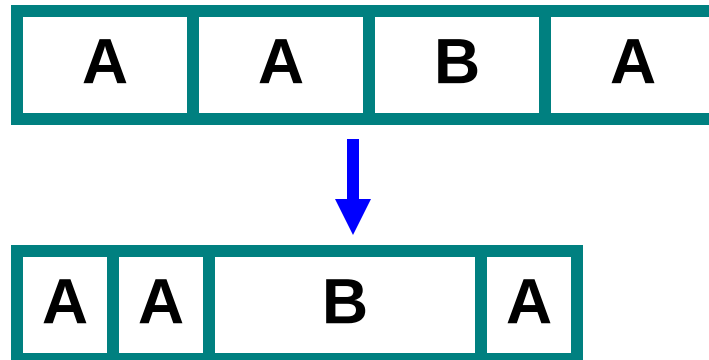
Huffman Code

■ 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	Dog	Cat	Bird	Fish
Frequency	1/8	1/4	1/2	1/8
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

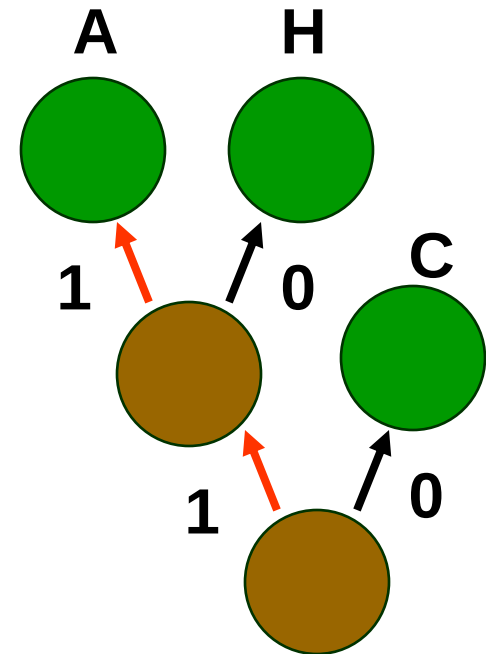
■ Binary (Huffman) tree

- Represents Huffman code
- Edge $\Rightarrow$ code (0 or 1)
- Leaf $\Rightarrow$ symbol
- Path to leaf $\Rightarrow$ encoding
- Example

■ $A = "11"$, $H = "10"$, $C = "0"$

■ Priority queue

- To efficiently build binary tree



Huffman Code Algorithm Overview

■ Encoding

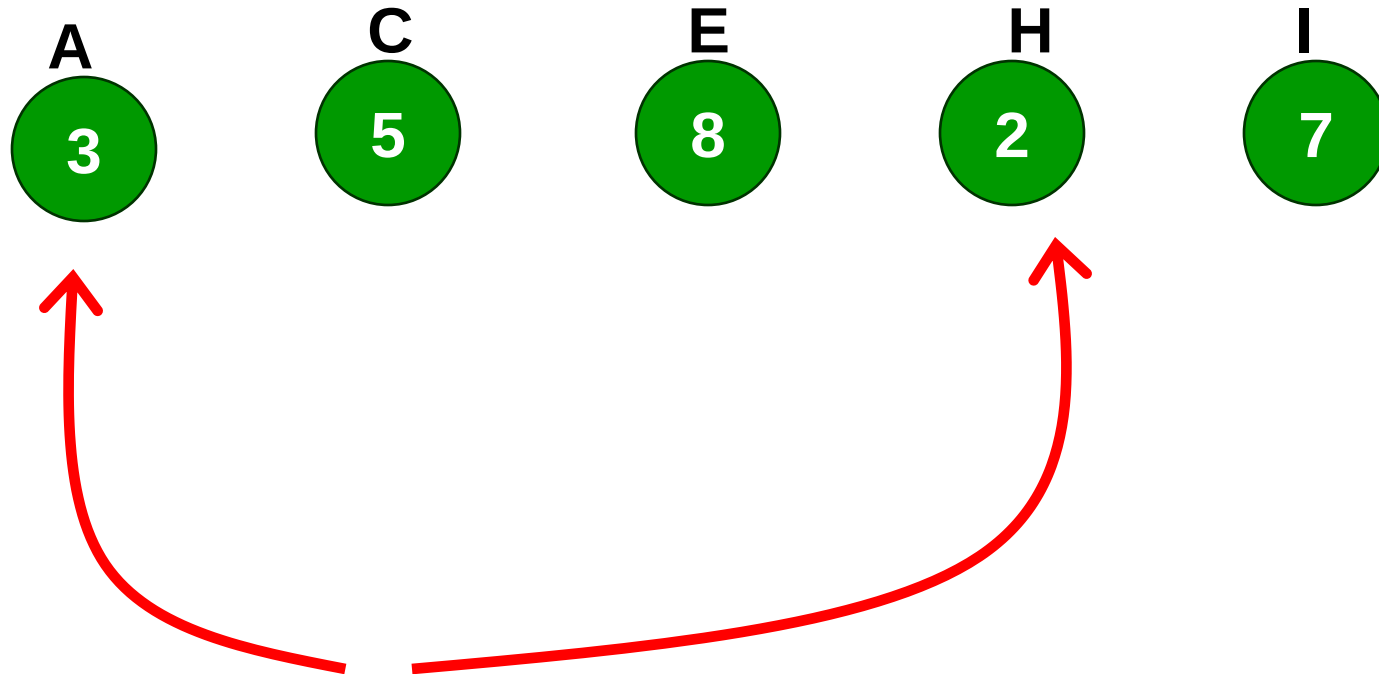
1. Calculate frequency of symbols in file
2. Create binary tree representing “best” encoding
3. Use binary tree to encode compressed file
 - For each symbol, output path from root to leaf
 - Size of encoding = length of path
4. Save binary tree

Huffman Code – Creating Tree

■ Algorithm

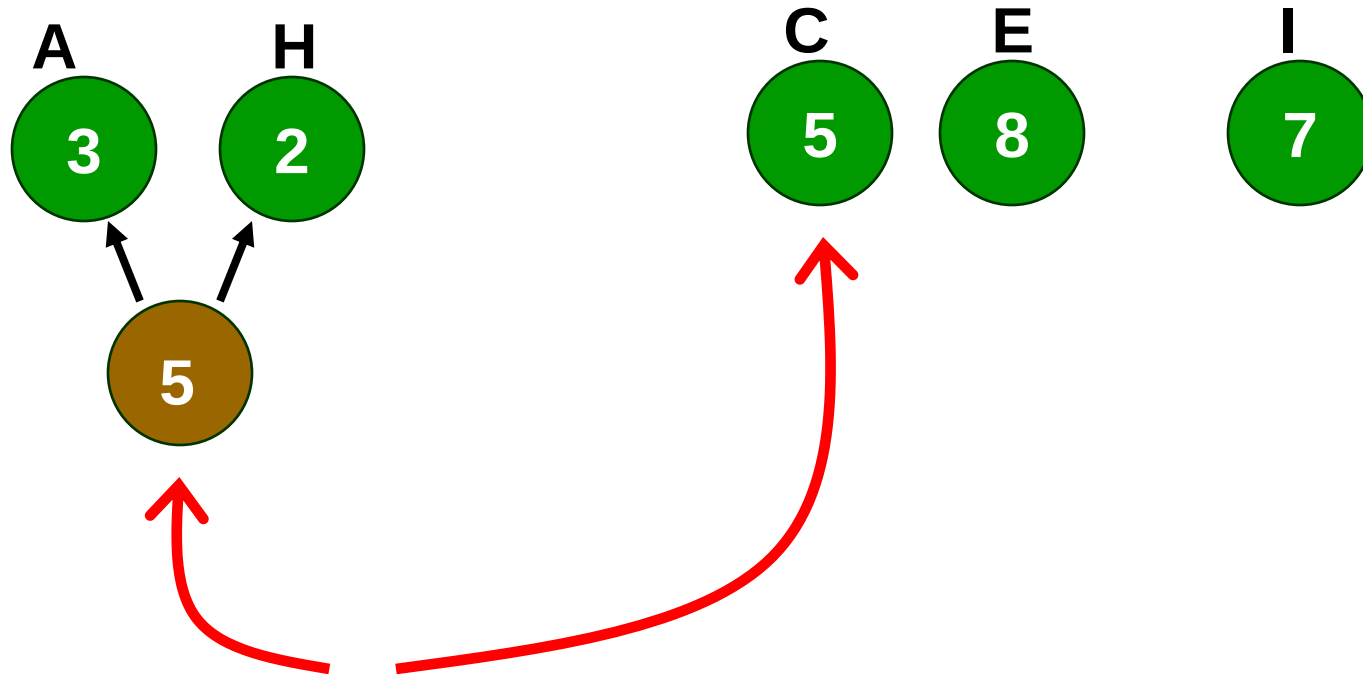
1. Place each symbol in leaf
 - Weight of leaf = symbol frequency
2. Select two trees L and R (initially leafs)
 - Such that L, R have lowest frequencies in tree
3. Create new (internal) node
 - Left child $\Rightarrow$ L
 - Right child $\Rightarrow$ R
 - New frequency $\Rightarrow$ frequency(L) + frequency(R)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

Huffman Tree Construction 1



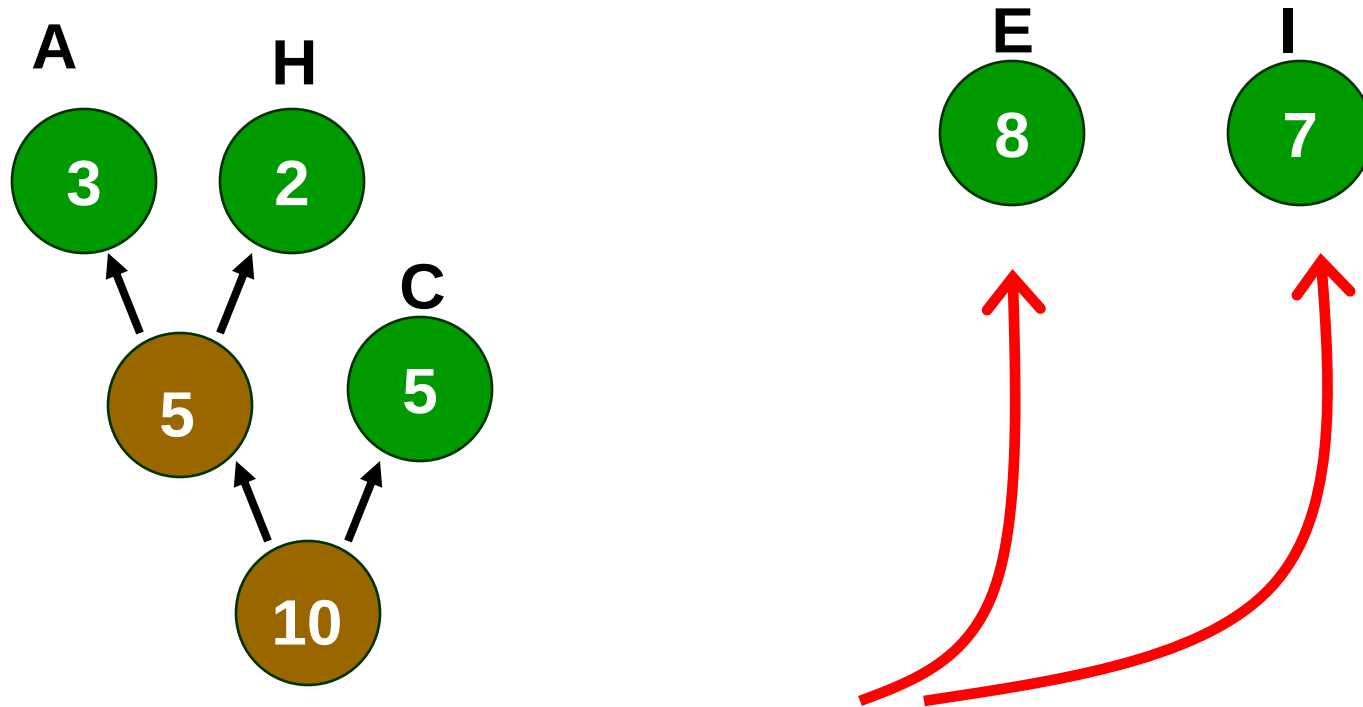
**2 trees with
lowest
frequency**

Huffman Tree Construction 2



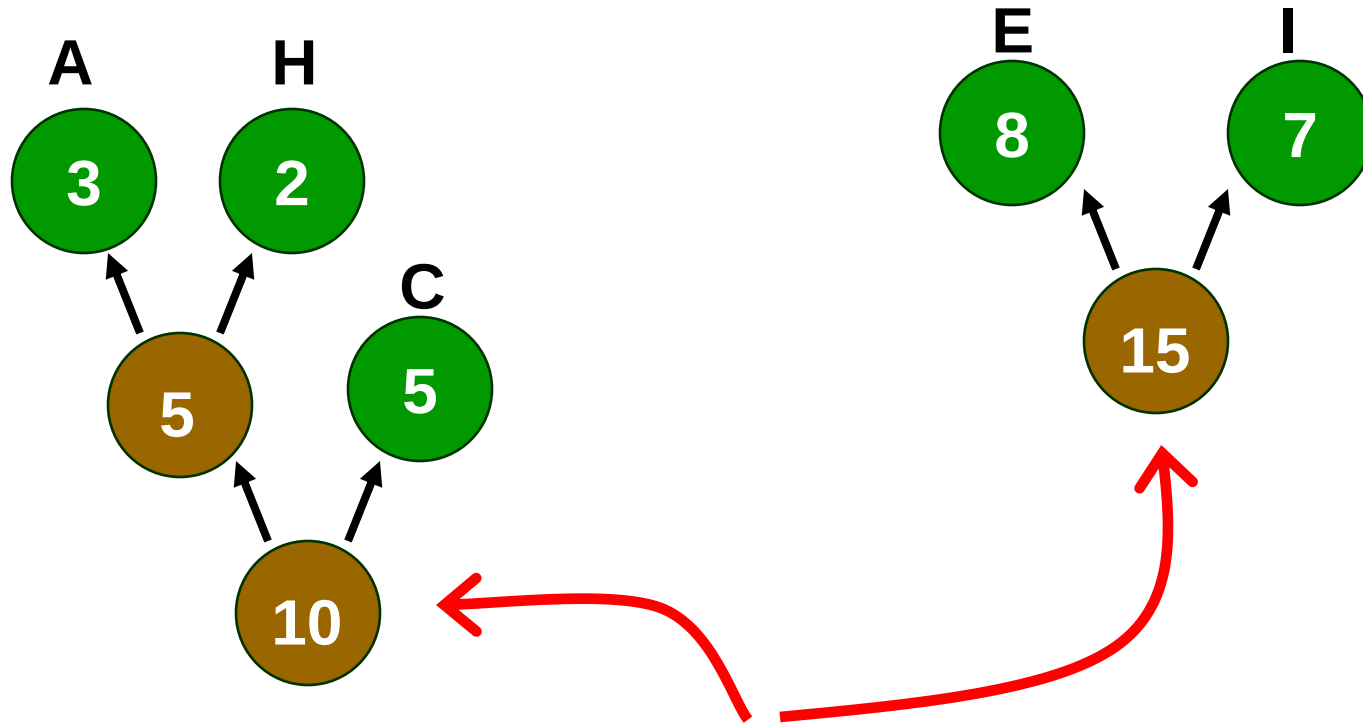
**2 trees with
lowest
frequency**

Huffman Tree Construction 3



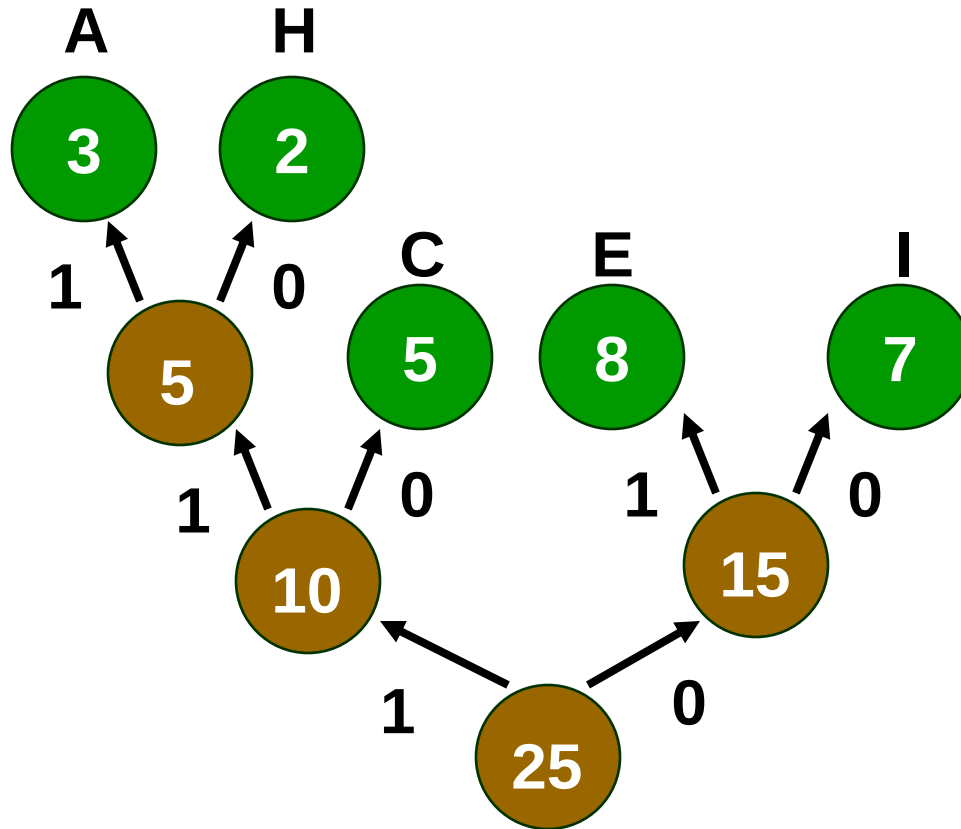
2 trees with
lowest
frequency

Huffman Tree Construction 4



**2 trees with
lowest
frequency**

Huffman Tree Construction 5



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

**Huffman
code for
each leaf**

Huffman Coding Example

■ Huffman code

E = 01

I = 00

C = 10

A = 111

H = 110

■ Input

■ ACE

■ Output

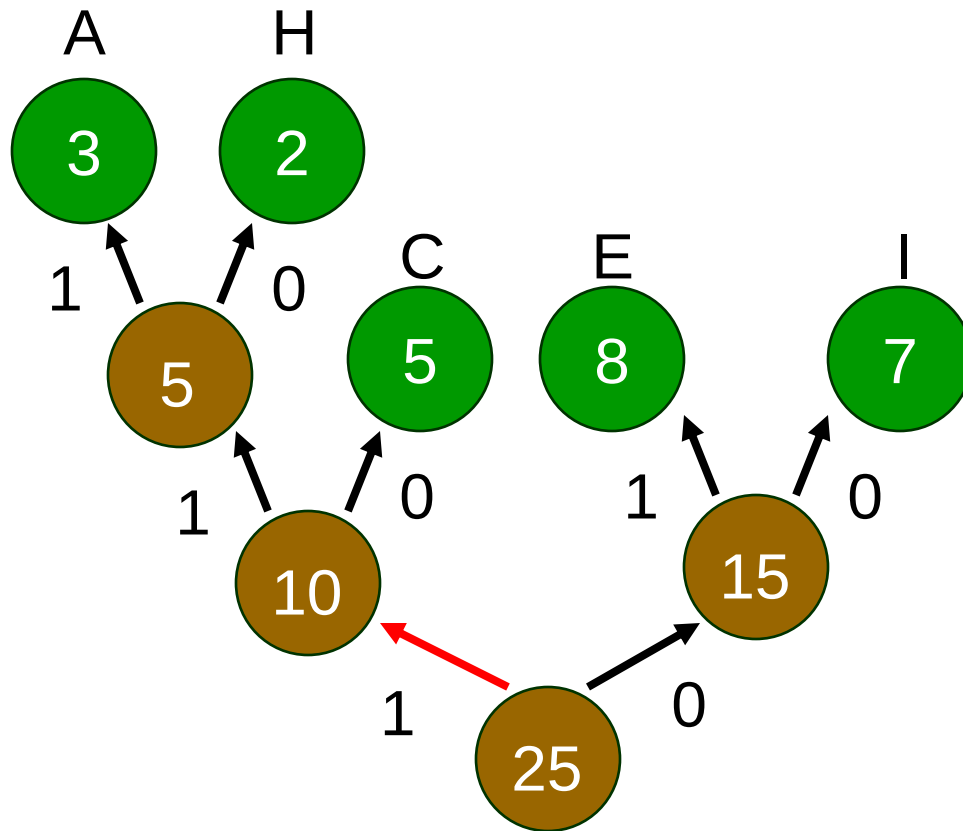
■ (111)(10)(01) = 1111001

Huffman Code Algorithm Overview

■ Decoding

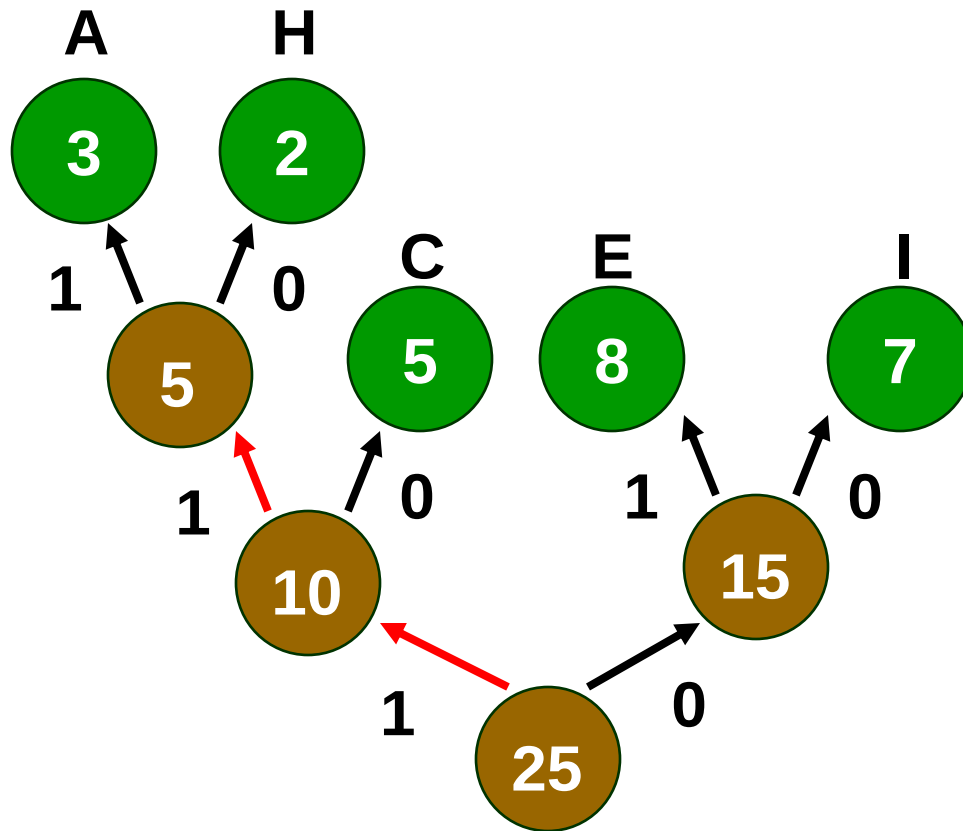
1. Read compressed file & binary tree
2. Use binary tree to decode file
 - Follow path from root to leaf

Huffman Decoding 1



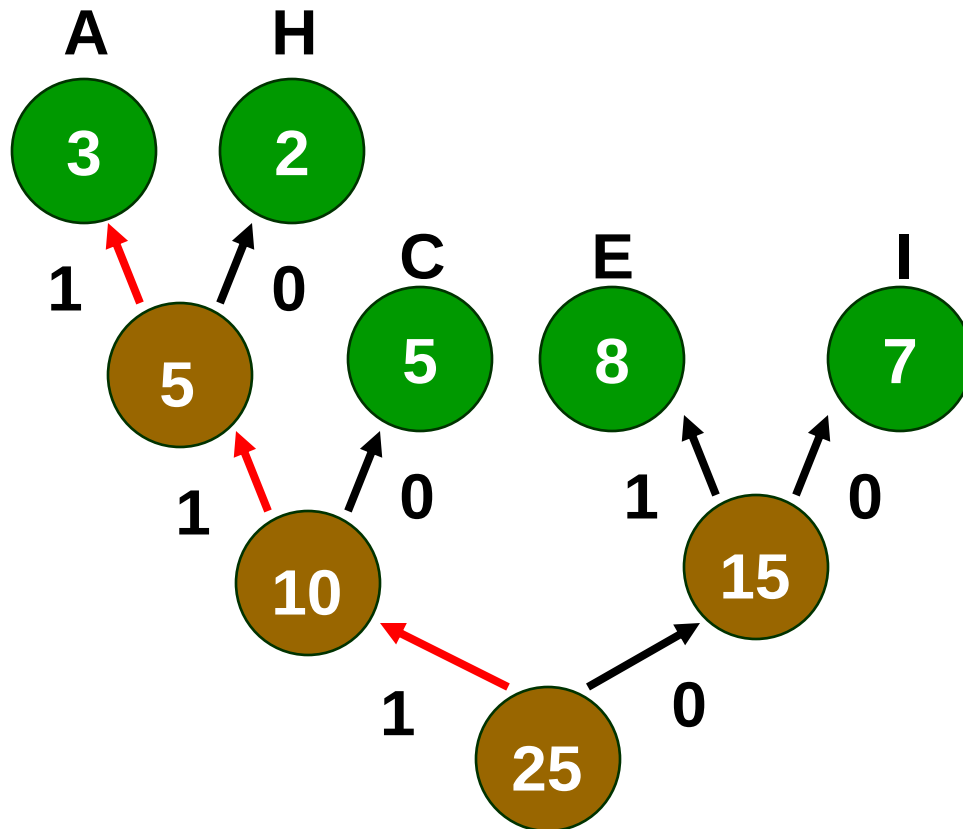
1111001

Huffman Decoding 2



1111001

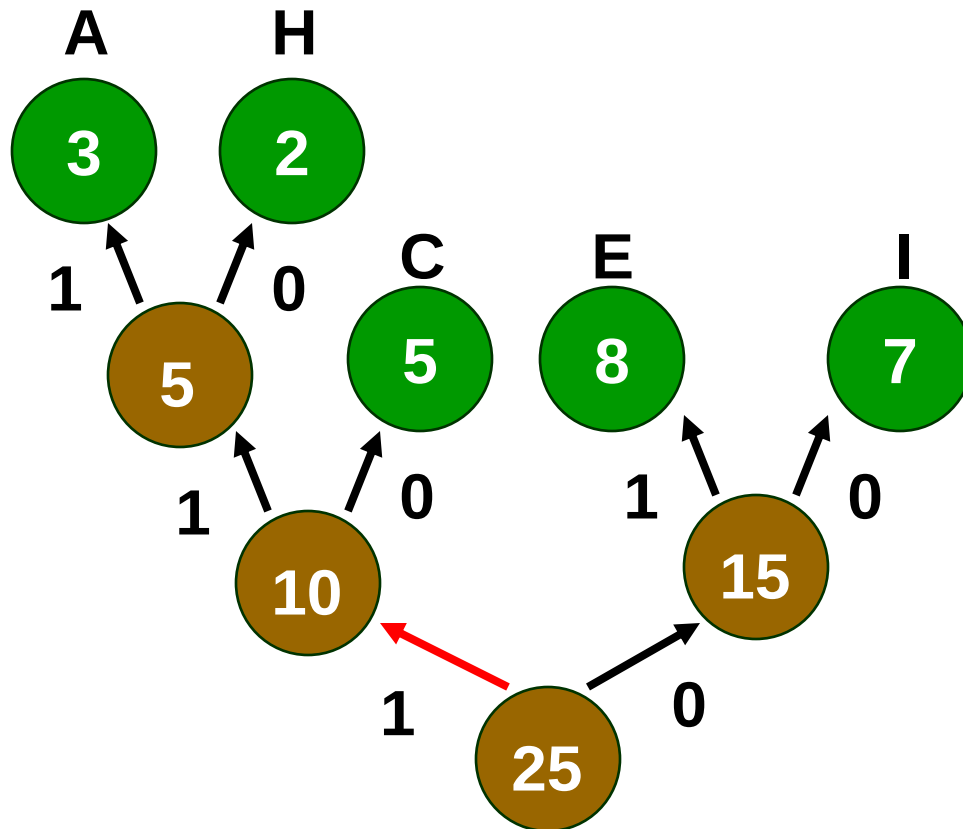
Huffman Decoding 3



1111001

A

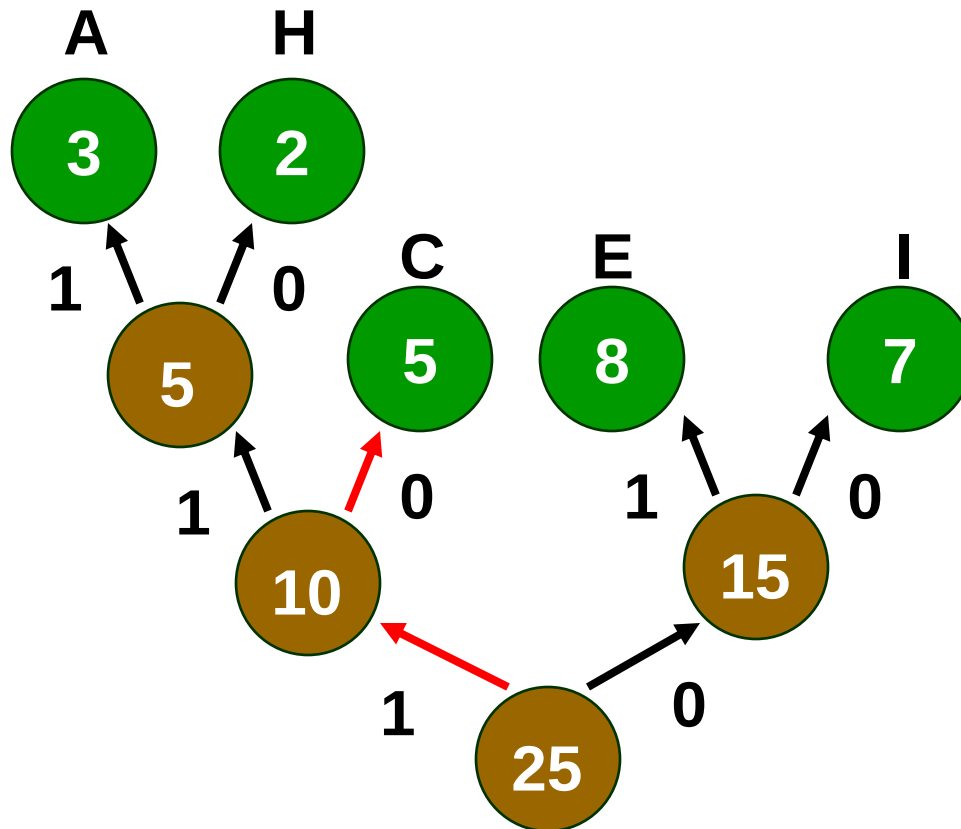
Huffman Decoding 4



1111001

A

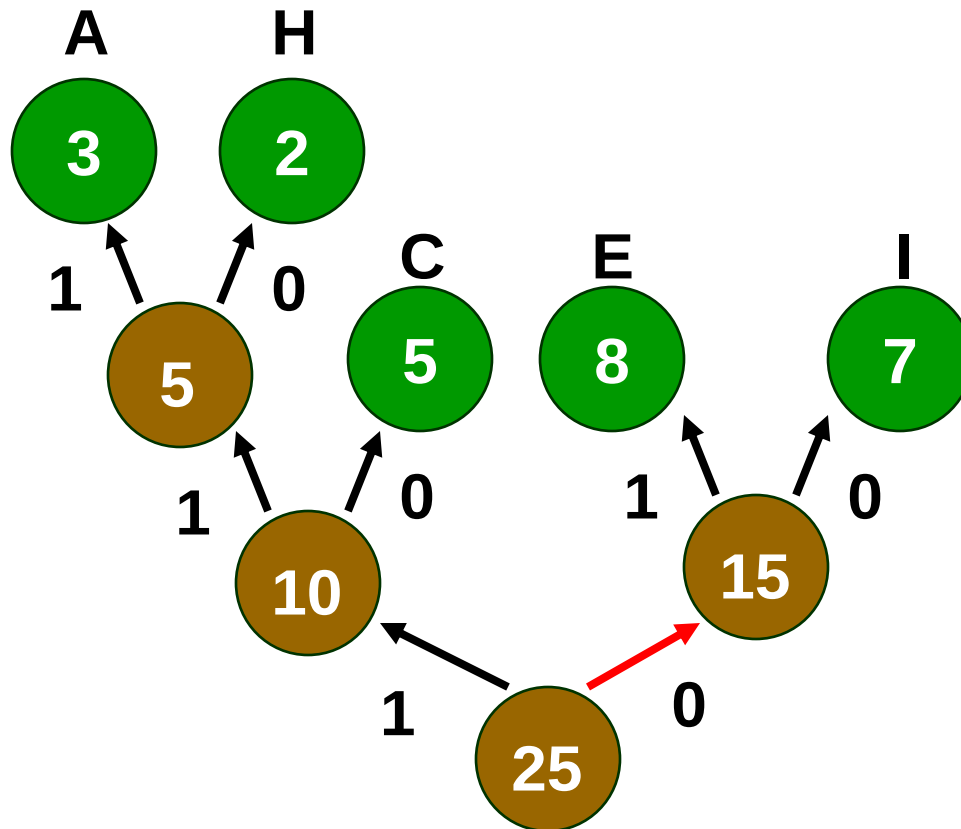
Huffman Decoding 5



1111001

AC

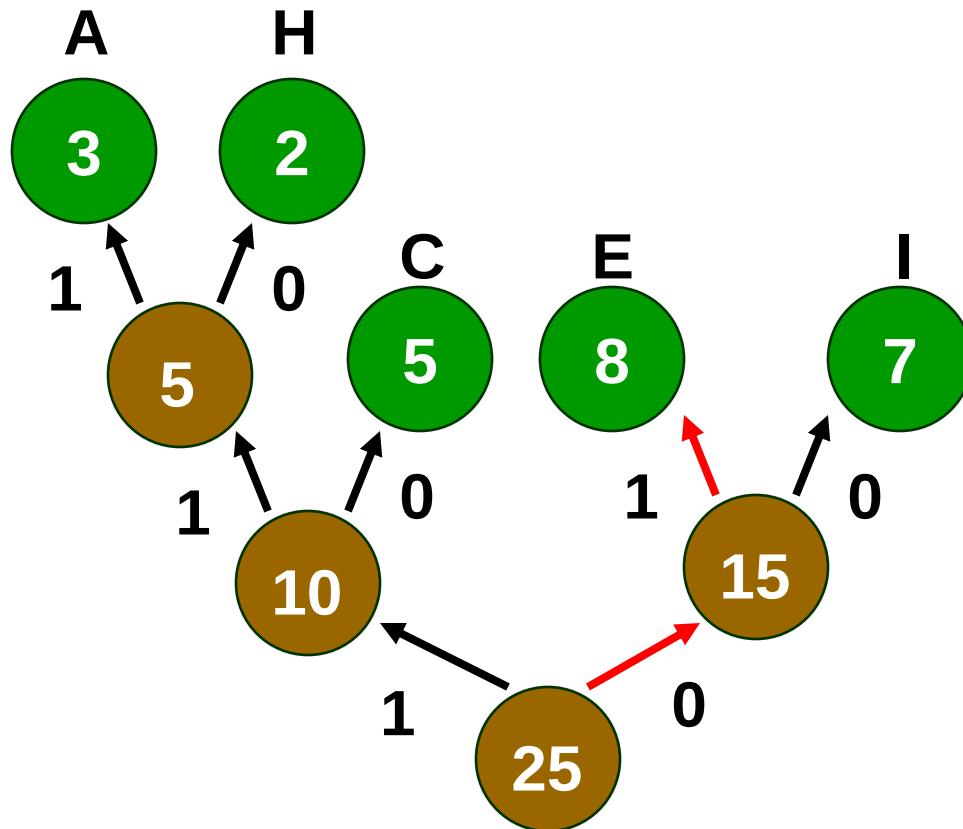
Huffman Decoding 6



1111001

AC

Huffman Decoding 7



1111001

ACE

Huffman Code Properties

■ Prefix code

- No code is a **prefix** of another code

- Example

 - $\text{Huffman}(\text{"dog"}) \Rightarrow 01$

 - $\text{Huffman}(\text{"cat"}) \Rightarrow 011$ // 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)