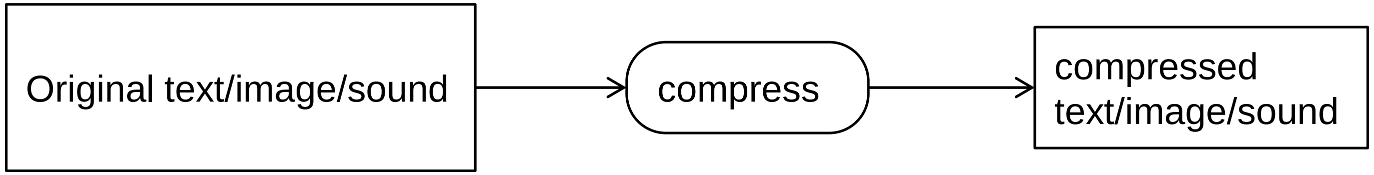
# Data Compression 1:

# Huffman Coding

*Fang-Lue Zhang*

# Data/Text Compression

- Files containing text, sound, video etc. can easily become huge.
  E.g. a blu ray movie is about 25Gb.
- Can we reduce the amount of time/space required to transmit/store them?
- E.g. text files are hugely redundant – we use 8 bits (or more) to store each character, but there is far less information than that.
- Compression is about reducing the memory required to store some information.


Original text/image/sound

compress

compressed text/image/sound

# Lossless v. Lossy Data Compression

Data compression may be:

- Lossless:
  - No information is lost – just gets stored more compactly
  - Can retrieve the original data exactly (decompress)
  - Important for text and some numerical data
    - compress to store/transmit, decompress to use
- Lossy:
  - Information may be lost
  - Can't retrieve the original data exactly
  - Acceptable in some contexts
    - data is stored and used in compressed form
  - E.g. JPEG compresses image files

# Lossless v. Lossy Data Compression

- Lossless compression only possible if there is *redundancy* in the original.

- Compression identifies and removes some of the redundant elements.

- Eg:
  - Identify repeated patterns
  - If lots of repeated characters, replace by count and character
  - Construct a dictionary and replace words by indices to it

# Encoding: compression, one symbol at a time

- Problem:
  - Given a set of symbols (characters, numbers, …)
  - Encode them as bit strings
    - Use a separate code for each symbol
  - Try to minimise the total number of bits.

- Today: Huffman coding
  - Very clever solution, very widely used（JPEG/MP3 as a back-end）
  - Combining several great ideas!

- Note: When coding data to store/transmit, we often add extra bits (i.e. redundancy) so we can detect errors:
  - See parity bits, error-correcting codes.
  - This can still be done with compressed data.

# Equal Length Codes

- Obvious approach:
  Use the same number of bits for every symbol to be encoded.

- E.g. digits:

| symbol: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| code: | 0000 | 0001 | 0010 | 0011 | 0100 | 0101 | 0110 | 0111 | 1000 | 1001 |

- E.g. letters:

| symbol: | a | b | c | d | e | f | g | … | z |
|---|---|---|---|---|---|---|---|---|---|
| code: | 00001 | 00010 | 00011 | 00100 | 00101 | 00110 | 00111 | … | 11010 |

Ex: How many bits for upper and lower case letters, and 0-9?

cf: ASCII

**How many bits are needed? 26 symbols -> 5 bits, much better than 8!**

# Equal Length Codes

- With N bits, we can have up to $2^N$ different codes.

- For N different symbols, need $\log_2 N$ bits per symbol

    10 numbers, message length = 4

    26 letters, message length = 5

- If there are many repeated symbols, can we do better?

String: a a b a j a b a a b

# Variable Length Codes

- **Great idea #1**:
  - – Use fewer bits for more common symbols
- Eg for letters, suppose:

| | |
|---|---|
| a | occurs 50% of the time, |
| b-c | occurs 15% of time, |
| d-e | each occur 5% of time, |
| f-j | each occur 2% of time. |

Encode:
a by ‘0’
b by ‘1’
c by ‘10’
d by ‘100’
e by ‘101’
f by ‘110’
…
i by ‘1001’

# Variable Length Codes

| sym: | a | b | c | d | e | f | g | h | i | j |
|---|---|---|---|---|---|---|---|---|---|---|
| code: | 0 | 1 | 10 | 100 | 101 | 110 | 111 | 1000 | 1001 | 1010 |

String: a a b a j a b a a b

Fixed: 0000 0000 0001 0000 1001 0000 0010 0000 0000 0001
(using 4 bits each as only 10 letters used)

Variable: 0 0 1 0 1001 0 10 0 0 1

Takes **14 bits, rather than 40.**

**Compression Rate Comparing to ASCII: 35%（Avg Bit Number / 8）**

# Variable length encoding

- Problem: where are the boundaries?

| sym: | a | b | c | d | e | f | g | h | i | j |
|---|---|---|---|---|---|---|---|---|---|---|
| code: | 0 | 1 | 10 | 100 | 101 | 110 | 111 | 1000 | 1001 | 1010 |

- How can we tell if 1001 is code for i, db or baab?

- A possible approach:
  - – Use 0 as a “sentinel bit” to mark the end of a code
  - – But then can only use 1’s for the code itself

-

| Sym: | a | b | c | d | e | f | … | j |
|---|---|---|---|---|---|---|---|---|
| Code: | 10 | 110 | 1110 | 11110 | 111110 | 1111110… | | 11111111110 |

- That’s not so good – can we do better?

# Prefix-free codes

- **Great idea #2**:

- Design codes so that no code is the prefix of another code!

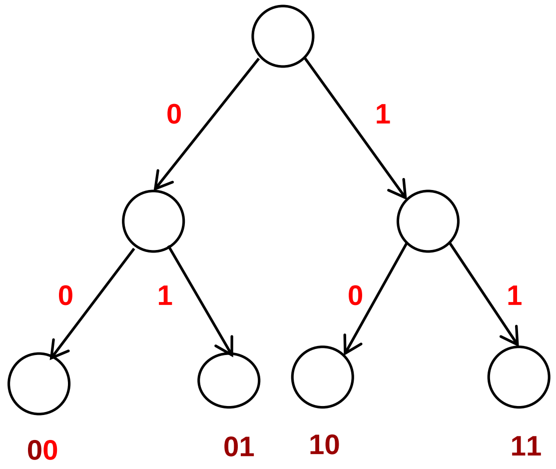
- Eg:

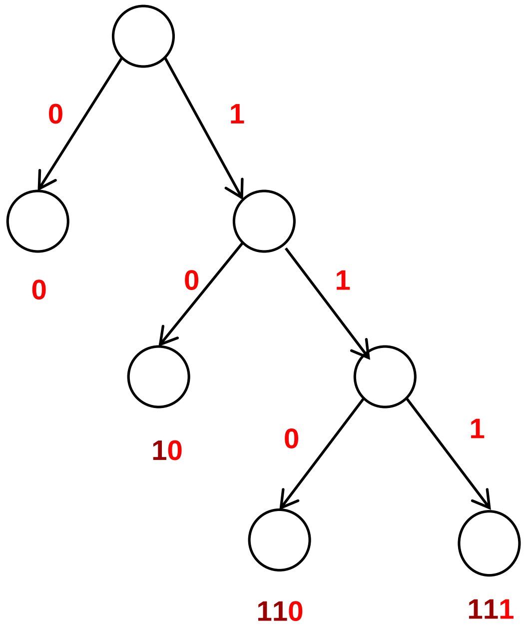
| sym: | a | b | c | d | e | f | g | h | |
|---|---|---|---|---|---|---|---|---|---|
| code: | 0 | 10 | 1100 | 11101 | 11100 | 11111 | 11010 | 110110 | .... |

- How do we design codes that are *prefix-free*?

# Prefix-free codes

- We can think of prefix-free codes as path labels to leaves in a binary tree





- Balanced tree gives equal length codes
- Linear tree is like using a sentinel bit
- What tree shape will give best codes?
- Want more frequent symbols at the top, less frequent at the bottom – but not too far away!

# Designing a good prefix-free code

- **Great idea #3**:

- Build the tree from the bottom-up, combining nodes with lowest frequencies.

    - Start with a leaf for each symbol, labelled with its frequency.
    - At each step, combine two nodes with smallest frequencies, add a new node as their parent, labelled with the sum of their frequencies.
    - Stop when all nodes are combined into a single tree.

# Example: Building the tree

~~🚲 50%~~

~~🛡 20%~~

~~🐞 5%~~

~~✦ 5%~~

~~⟲ 5%~~

~~👁 5%~~

~~👂 2%~~

~~📣 2%~~

~~♥ 2%~~

~~💐 2%~~

~~⛳ 2%~~

~~J 50%~~

~~H 30%~~

~~G 19%~~

~~F 11%~~

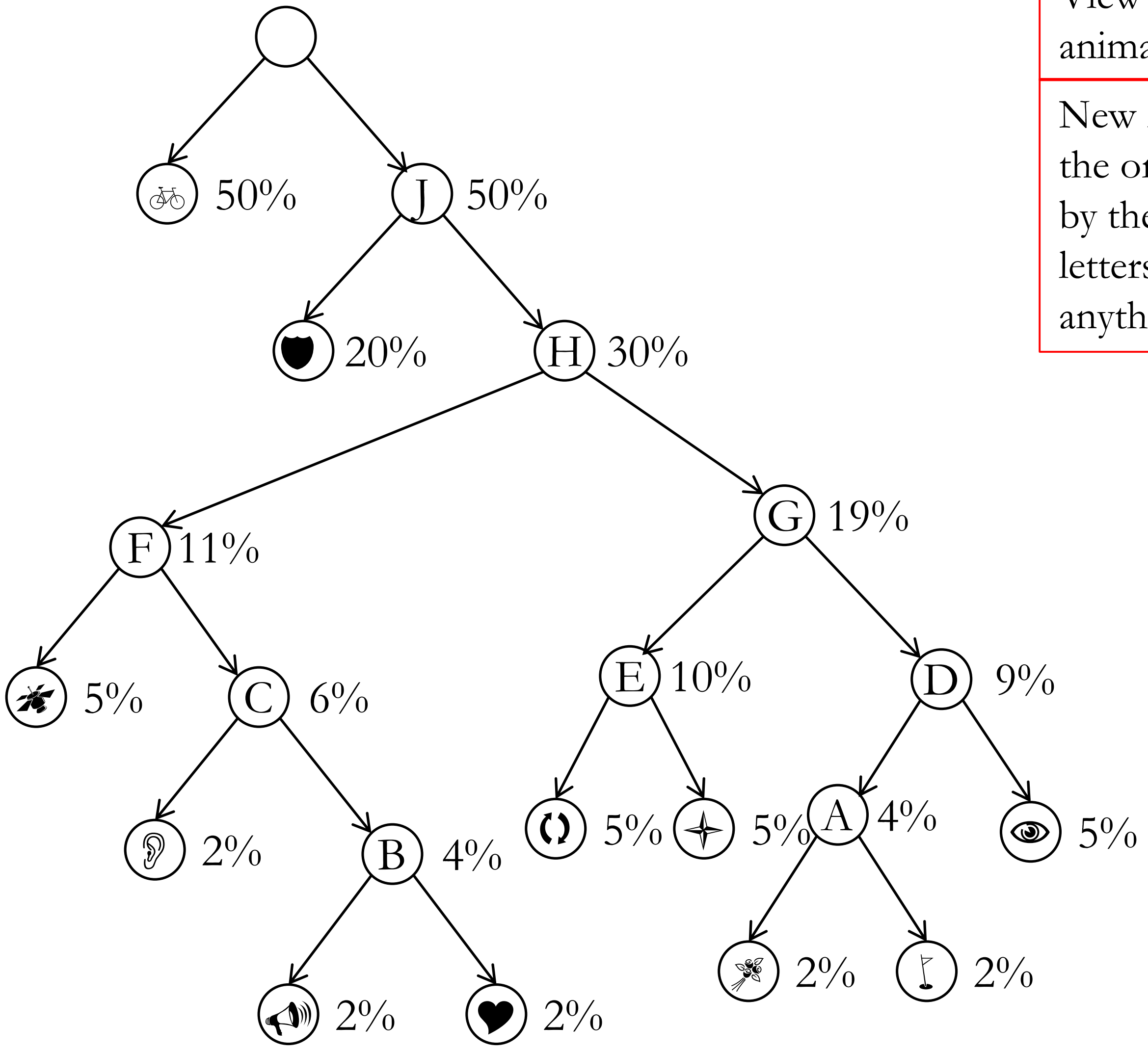
~~E 10%~~

~~D 9%~~

~~C 6%~~

~~A 4%~~

~~B 4%~~

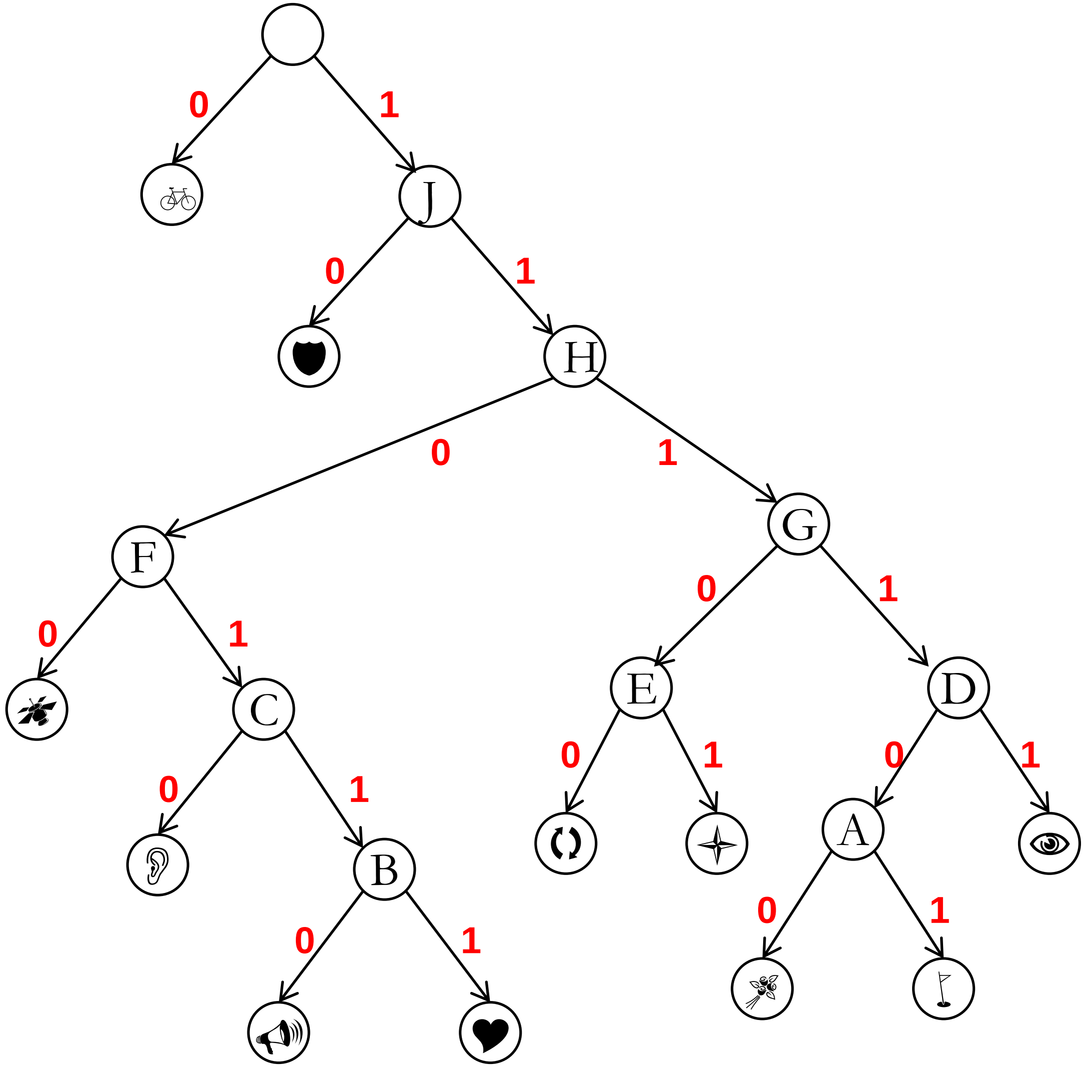


View the powerpoint animation!

New nodes added in the order indicated by their letters… ie. letters don't mean anything

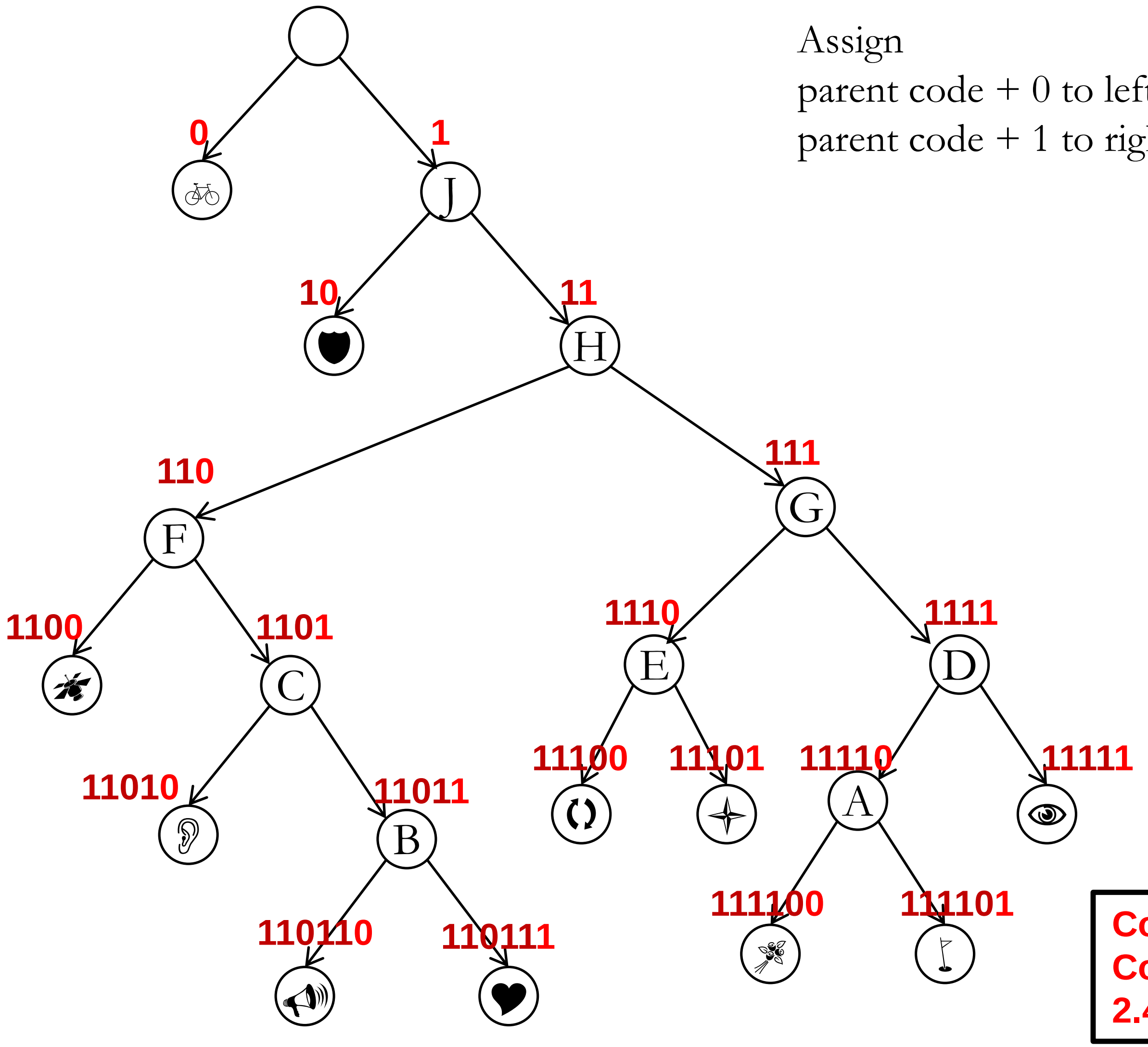
# Example: assigning the codes

| | |
|---|---|
| 🚲 | 50% |
| 🛡 | 20% |
| 🐝 | 5% |
| ✦ | 5% |
| 🔄 | 5% |
| 👁 | 5% |
| 👂 | 2% |
| 📢 | 2% |
| ♥ | 2% |
| 💐 | 2% |
| ⛳ | 2% |



# Example: assigning the codes

| | | |
|---|---|---|
| 🚲 | 50% | 0 |
| ⛊ | 20% | 10 |
| 🐞 | 5% | 1100 |
| ✦ | 5% | 11101 |
| ↻ | 5% | 11100 |
| 👁 | 5% | 11111 |
| 👂 | 2% | 11010 |
| 📢 | 2% | 110110 |
| ♥ | 2% | 110111 |
| ✿ | 2% | 111100 |
| ⚑ | 2% | 111101 |



Assign
parent code + 0 to left child
parent code + 1 to right child

**Compression Rate Comparing to ASCII: 2.43 / 8 = 30.3%**

average code length = (1*.5)+(2*.2)+(4*.05)+(5*.17)+(6*.08) = 2.43 bits

# Huffman Coding

- Generates the *best* set of codes,  given frequencies/probabilities on all the symbols.
- Creates a binary tree, which is used to construct the codes.

```
Construct a leaf node (singleton tree) for each symbol.
Put these nodes into a priority queue, with frequency as priority.
    (lowest frequency = highest priority)

while there is more than one node in the queue:  (i.e. > 1 tree)
    remove the top two nodes
    create a new tree node with these two nodes as children.
        node frequency = sum of frequencies of the two nodes
    add new node to the queue

Final node is root of tree.
Traverse this tree to assign codes:
if node has code c, assign  c0 to left child, c1 to right child
```

# Huffman Coding

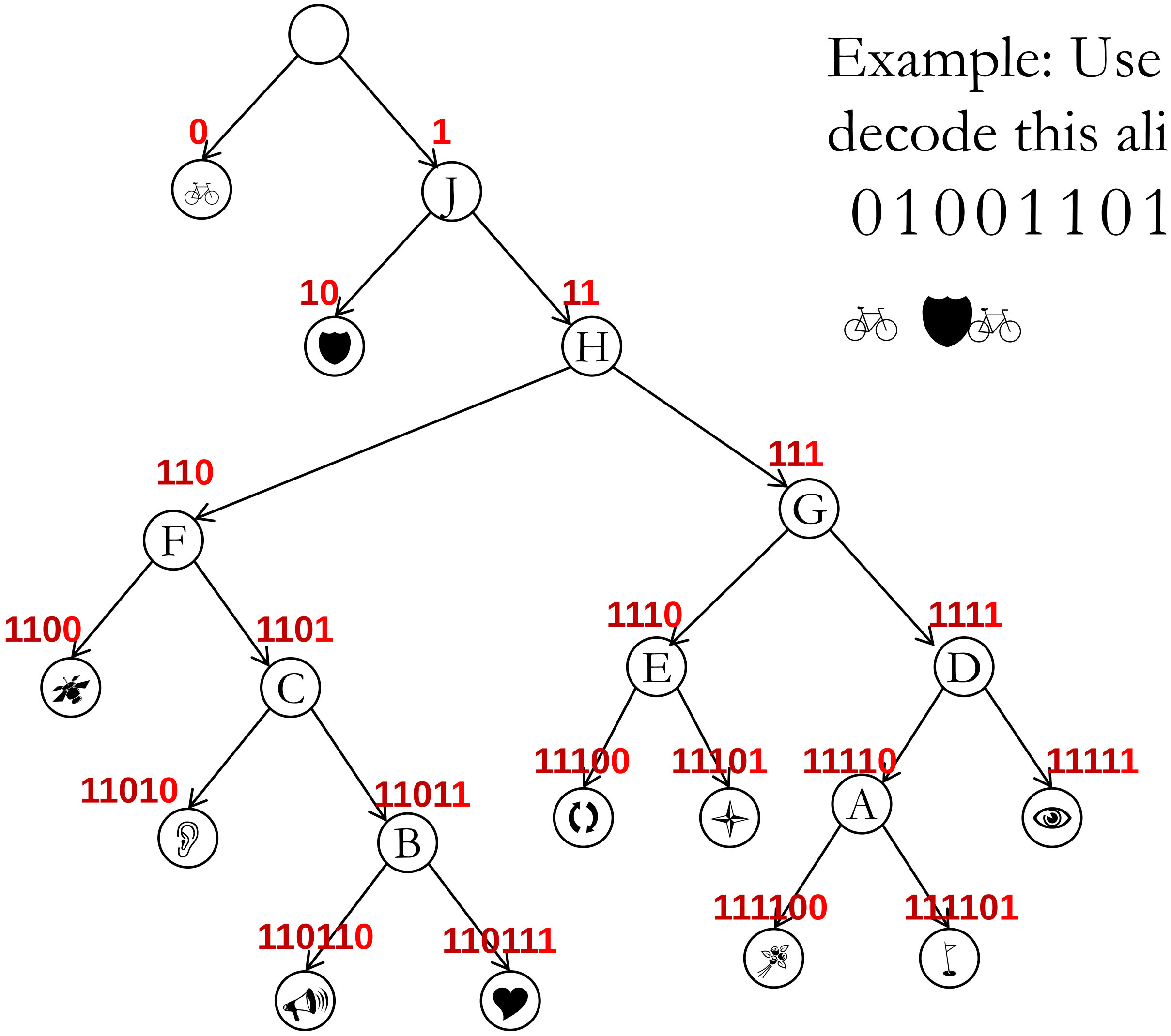
- To decode, we need a table of the codes used.

- If we label the edges of the tree with 0’s and 1’s, as added at each level, we get a *trie* which can be used like a scanner to split the coded string/file into separate codes to be decoded.

- See examples in the following slides

# Example: assigning the codes

| | | |
|---|---|---|
| 🚲 | 50% | 0 |
| 🛡 | 20% | 10 |
| 🪰 | 5% | 1100 |
| ✦ | 5% | 11101 |
| 🔄 | 5% | 11100 |
| 👁 | 5% | 11111 |
| 👂 | 2% | 11010 |
| 📢 | 2% | 110110 |
| ♥ | 2% | 110111 |
| 💐 | 2% | 111100 |
| ⛳ | 2% | 111101 |



Example: Use tree to decode this alien message

0 1 0 0 1 1 0 1 0 1 1 1 1 1

🚲 🛡 🚲 👂 👁

# Huffman Coding

- When storing/transmitting a compressed file, we need to include the tree for decompressing.
  - Can reduce efficiency of coding.

- Or, use a standard frequency table, not based on the particular file, for code.
  - E.g. use known frequencies of letters in English text.