University of Pittsburgh

Algorithms and Data Structures 2
CS 1501

Fall 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming Deadlines
  - Homework 6: this Friday @ 11:59 pm
  - Lab 5: next Monday @ 11:59 pm
  - Assignment 1: Monday Oct 10th @ 11:59 pm
    - Autograder is up on GradeScope
- If you think you lost points in a lab assignment because of the autograder or because of a simple mistake
  - please reach out to Grader TA over Piazza
- **Live support session** for Assignment 1
  - Recording and slides on Canvas
- **Student Support Hours** of the teaching team are posted on the Syllabus page

Previous lecture

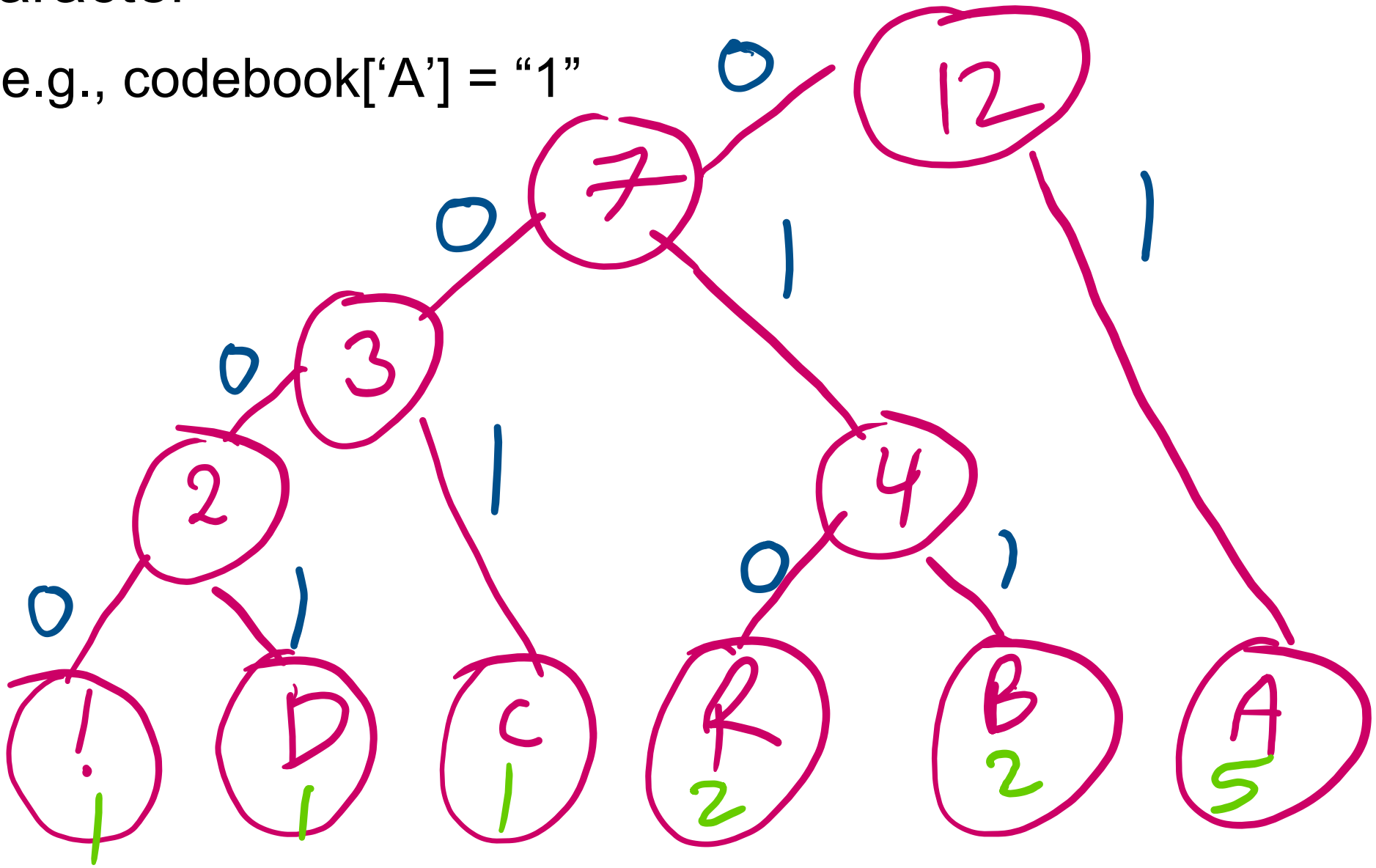
- Huffman Compression
  - Huffman Trie construction algorithm
  - Trie writing and reading
  - Binary I/O

This Lecture

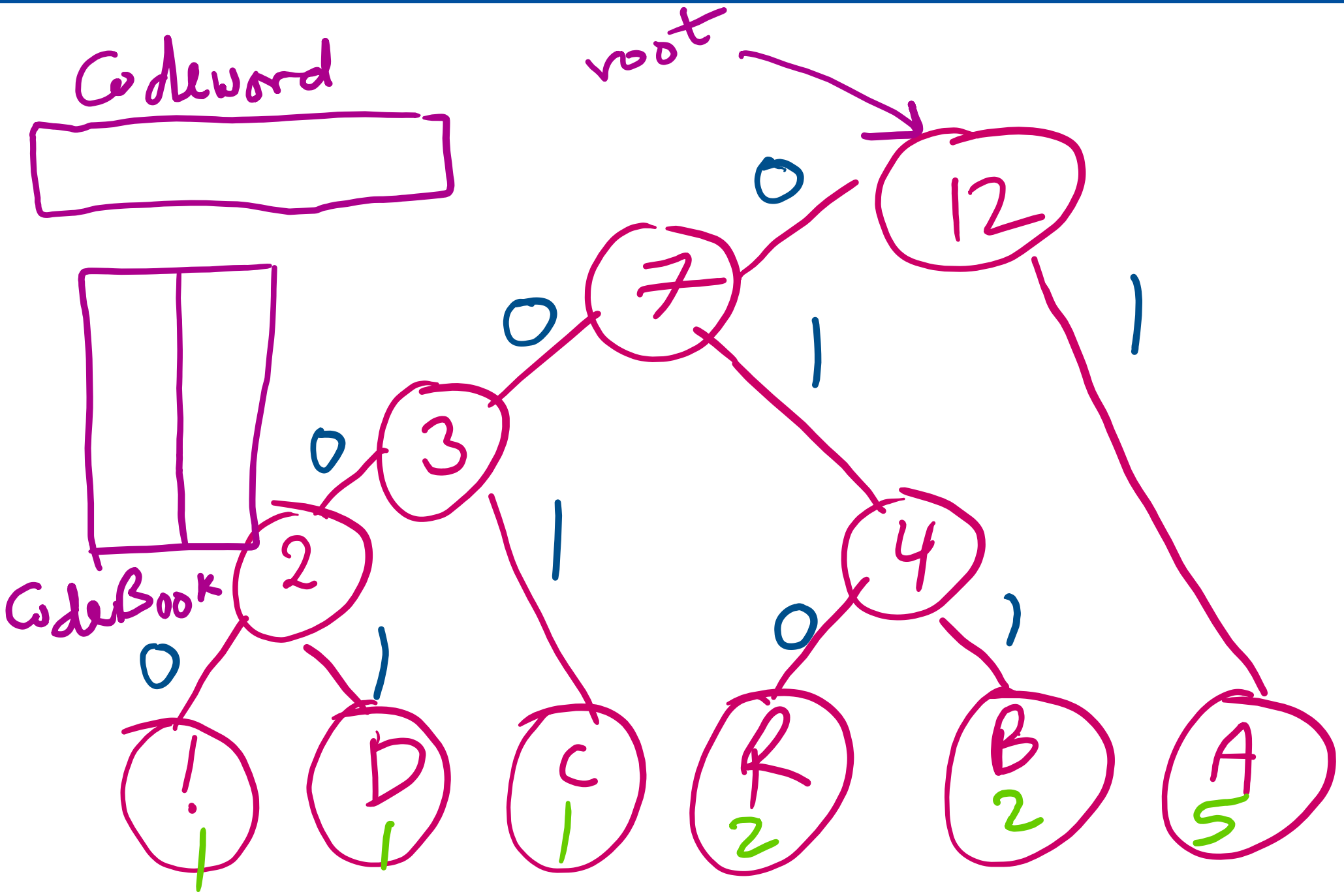
- Huffman Compression
  - How to compute character frequencies
- Run-length Encoding
- LZW

Huffman Compression: Generating the *Codebook*

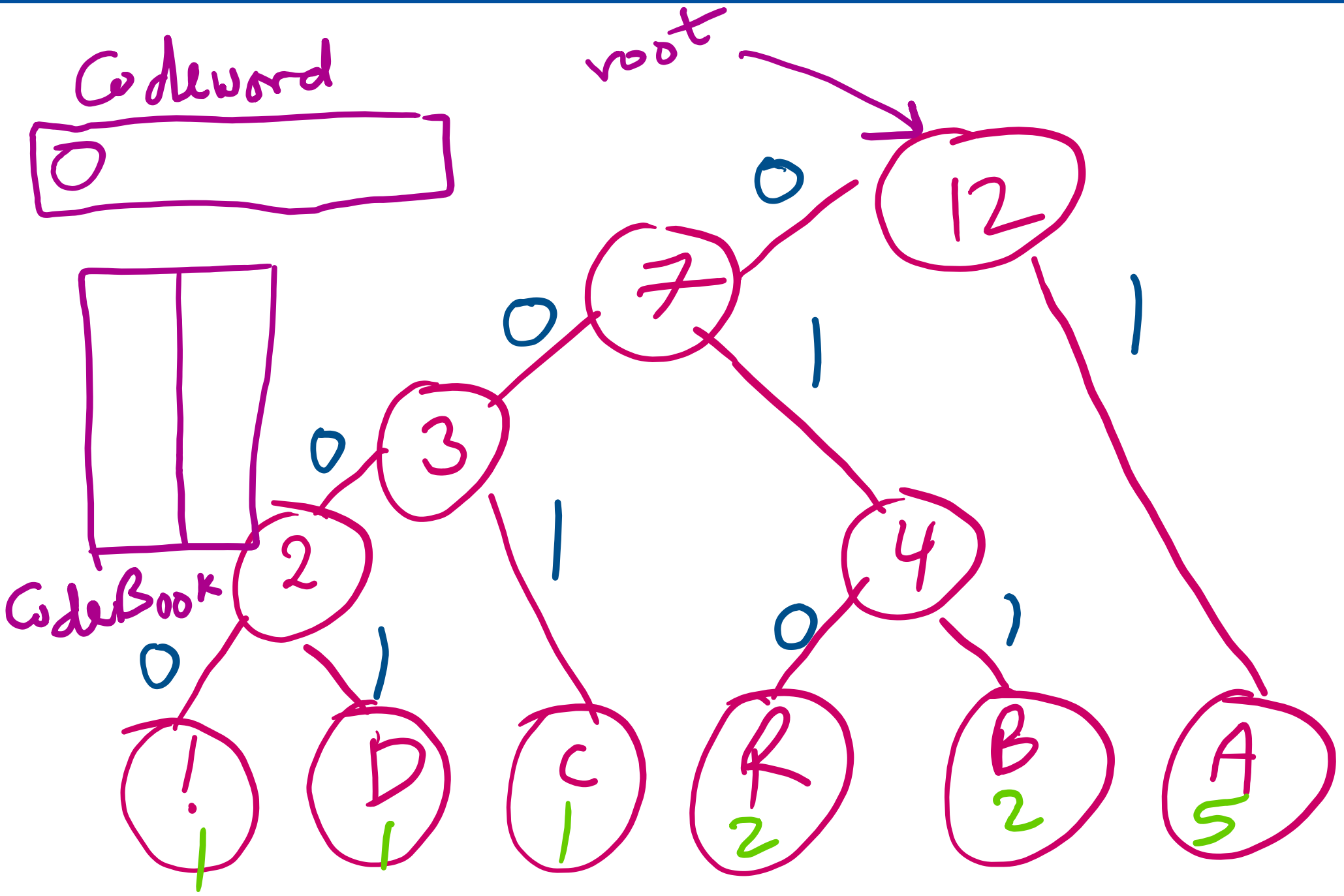
- The codebook is a table of codewords indexed by character
  - e.g., codebook['A'] = "1"



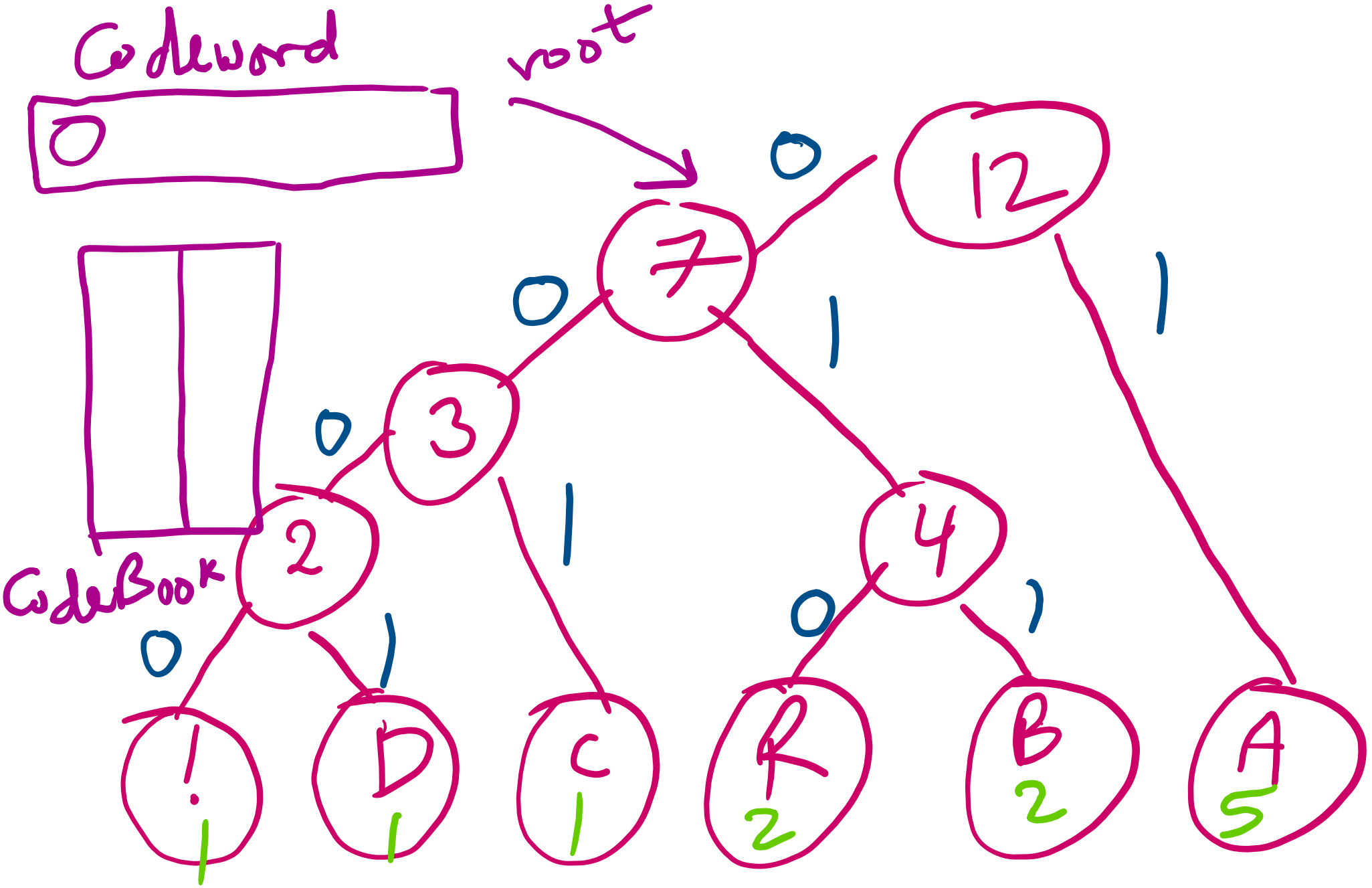
Huffman Compression: Generating the *Codebook*



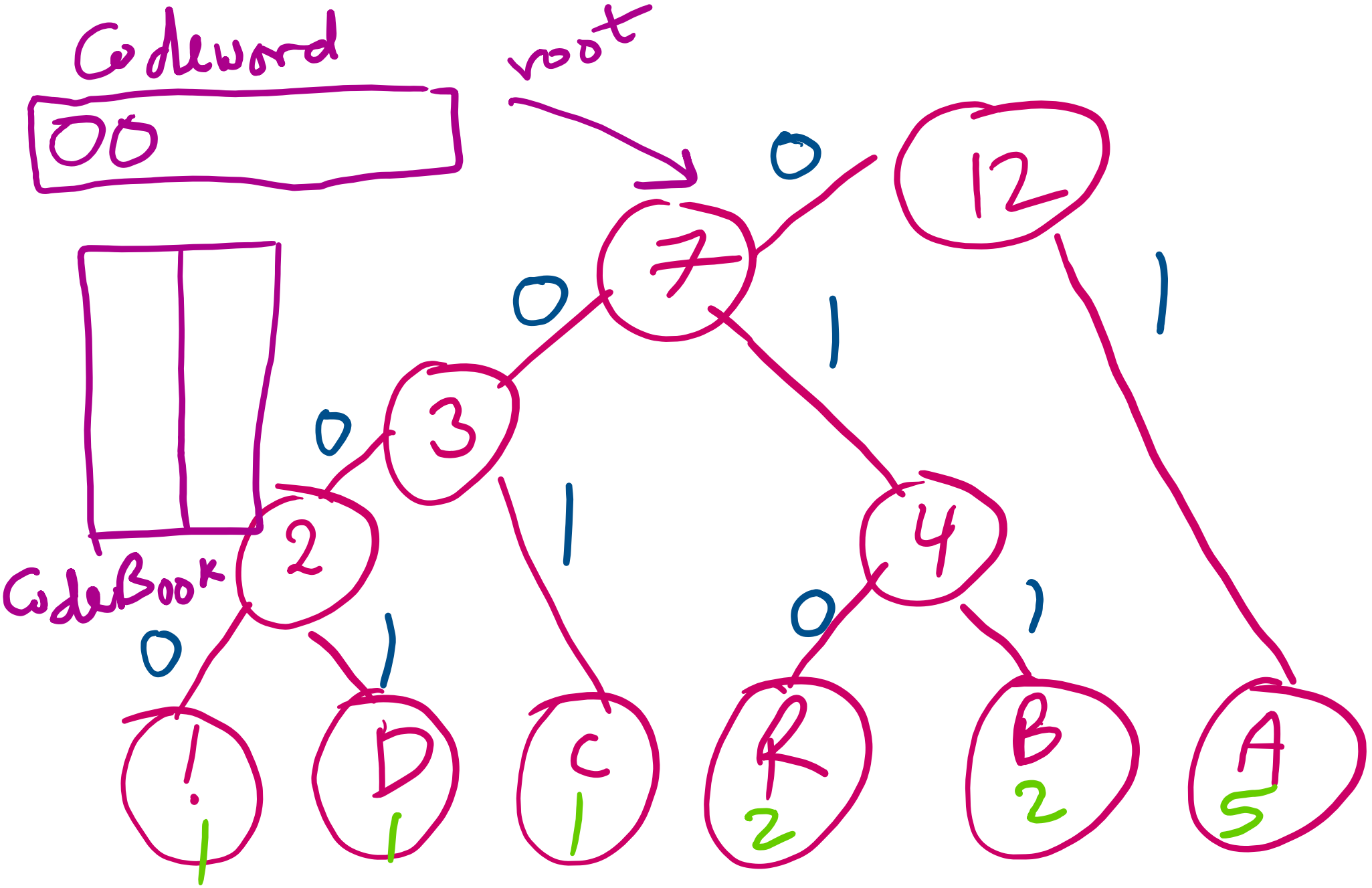
Huffman Compression: Generating the *Codebook*



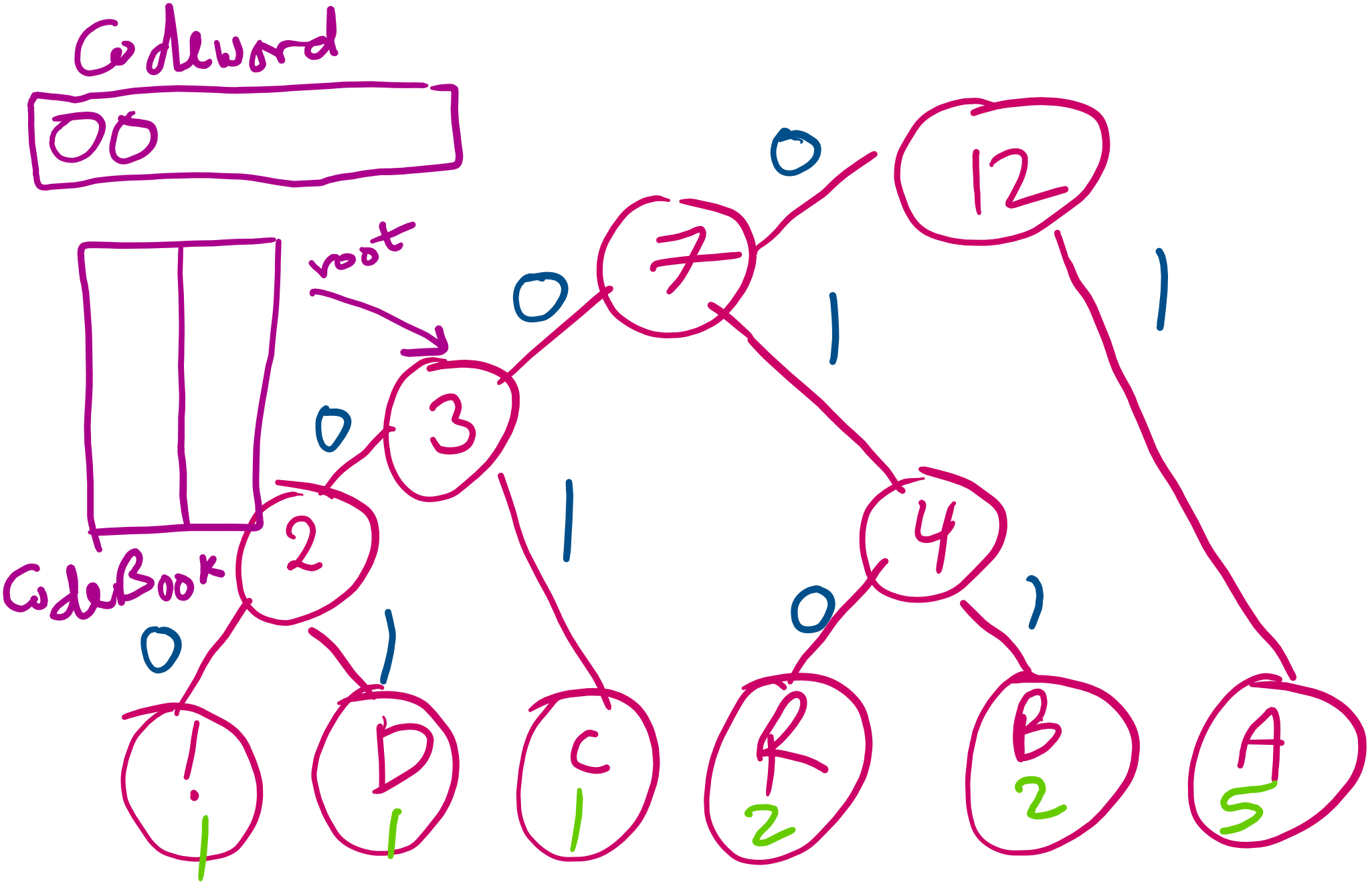
Huffman Compression: Generating the *Codebook*



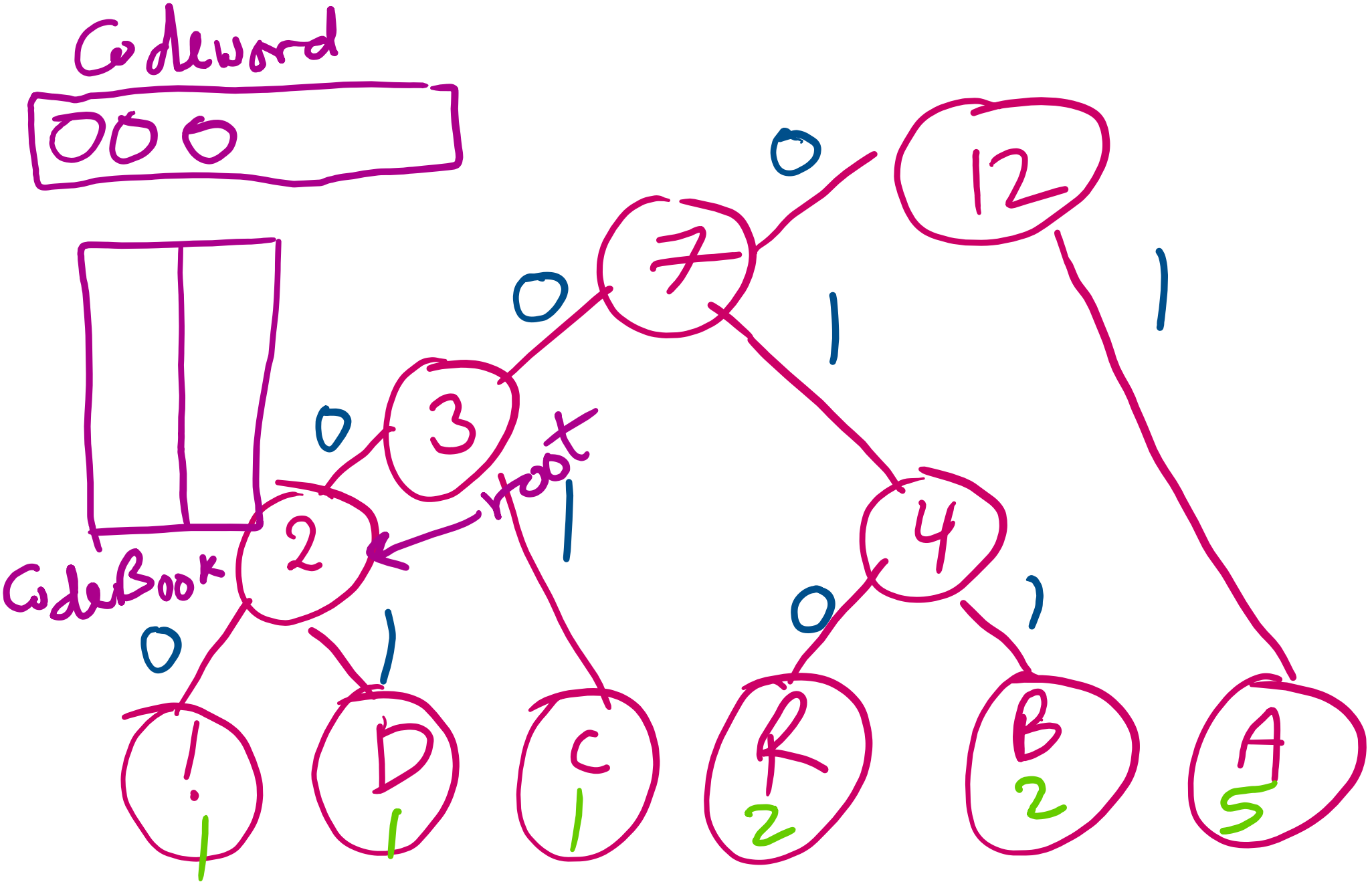
Huffman Compression: Generating the *Codebook*



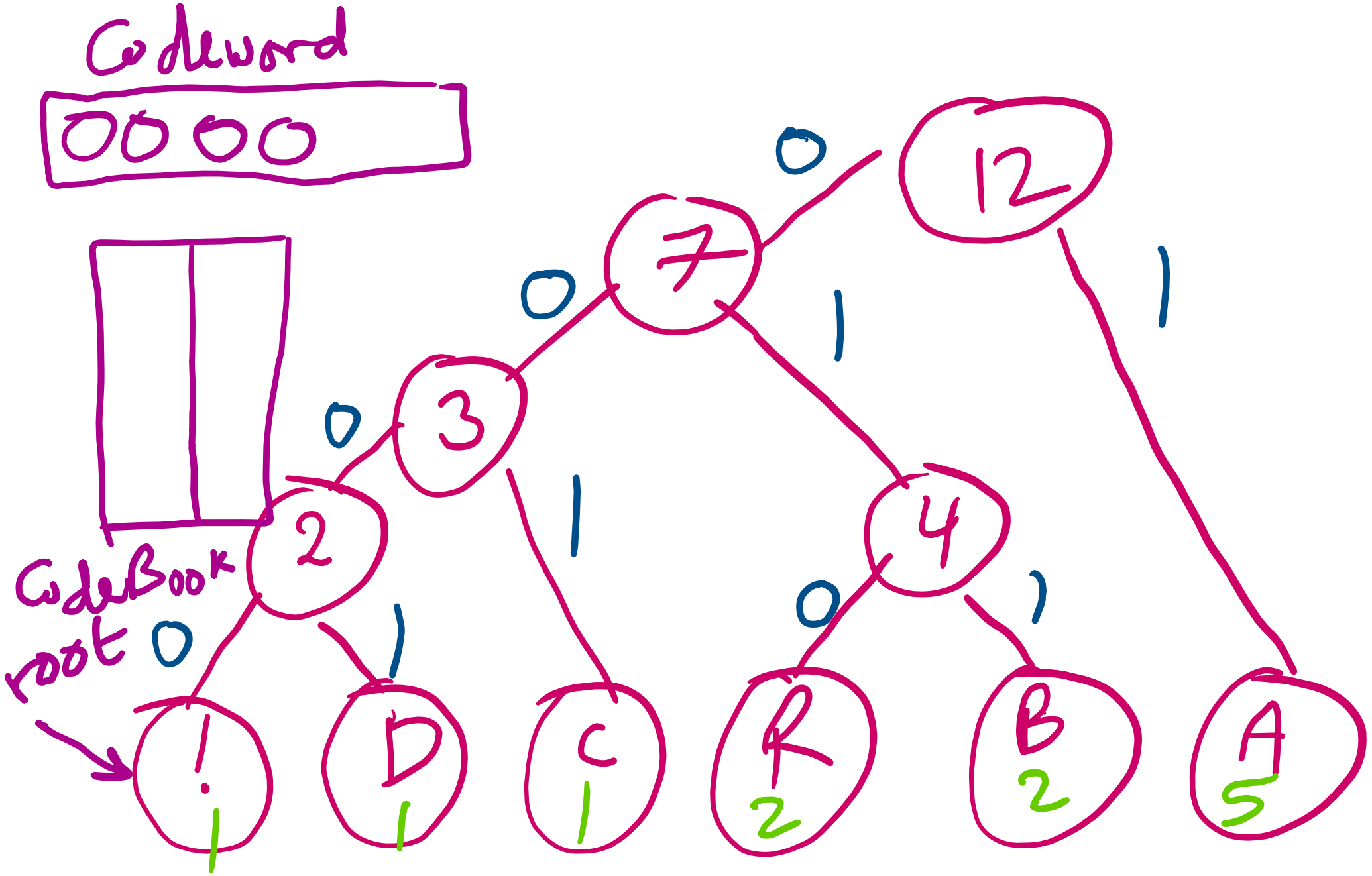
Huffman Compression: Generating the *Codebook*



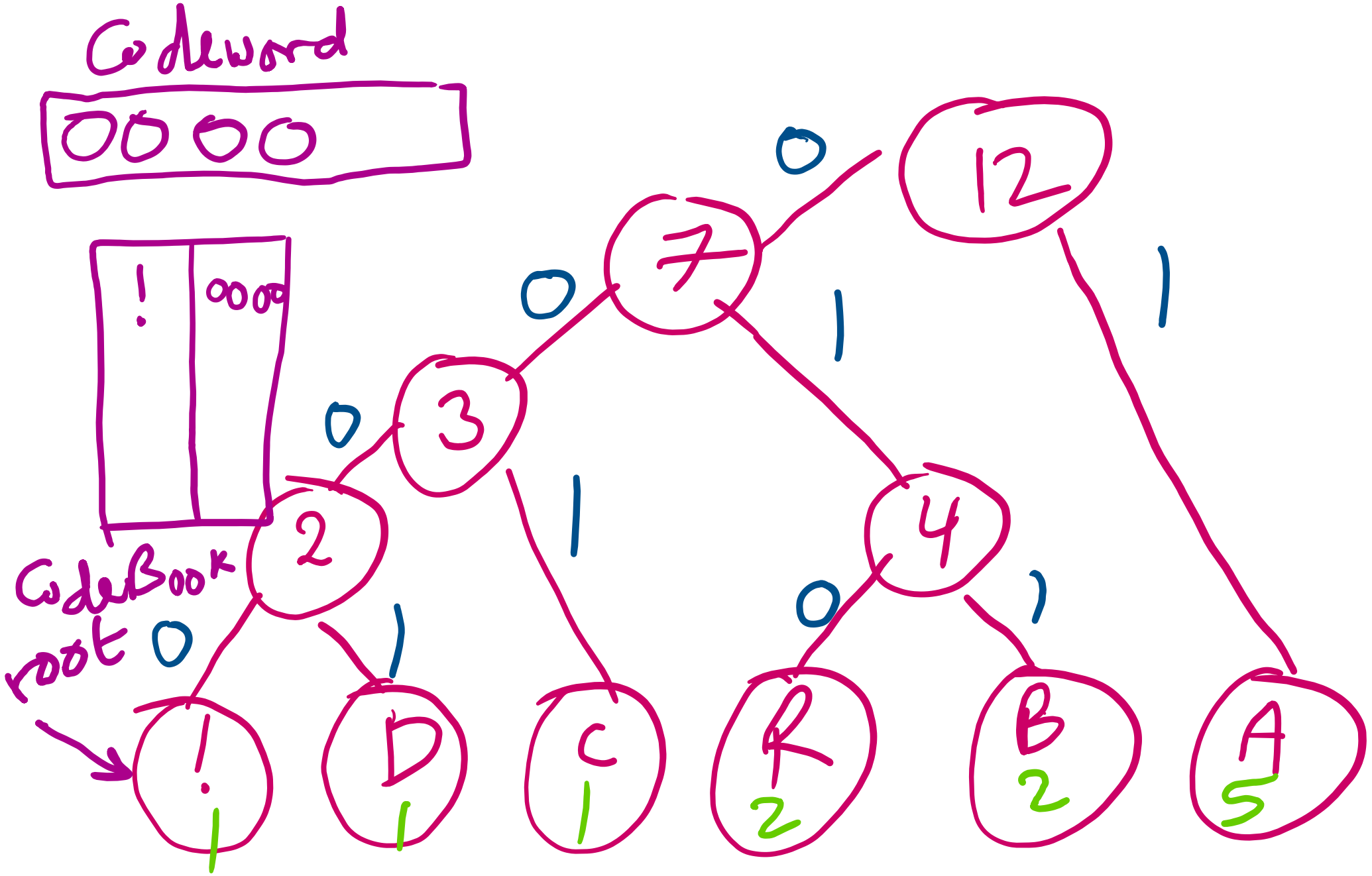
Huffman Compression: Generating the *Codebook*



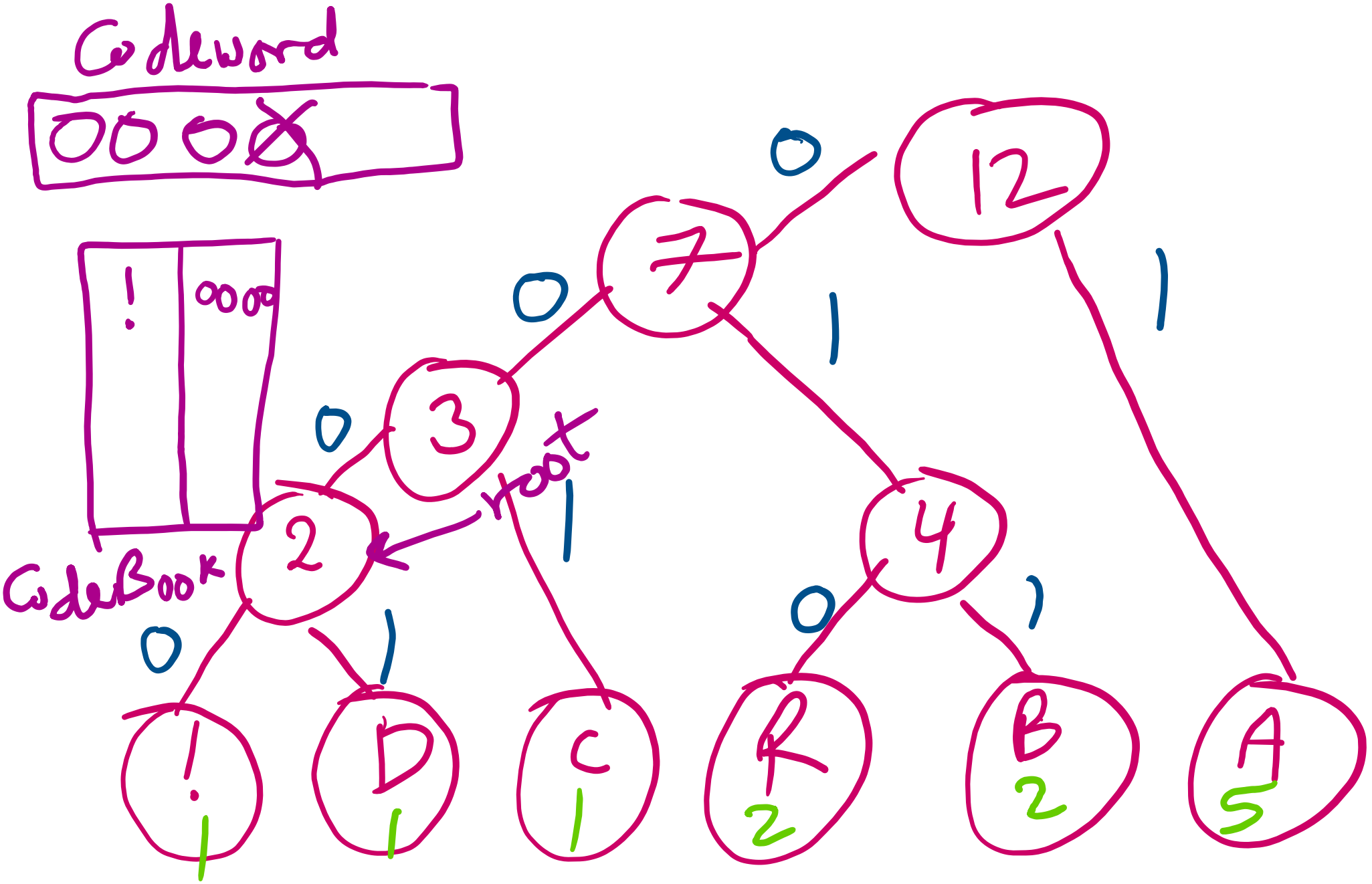
Huffman Compression: Generating the *Codebook*



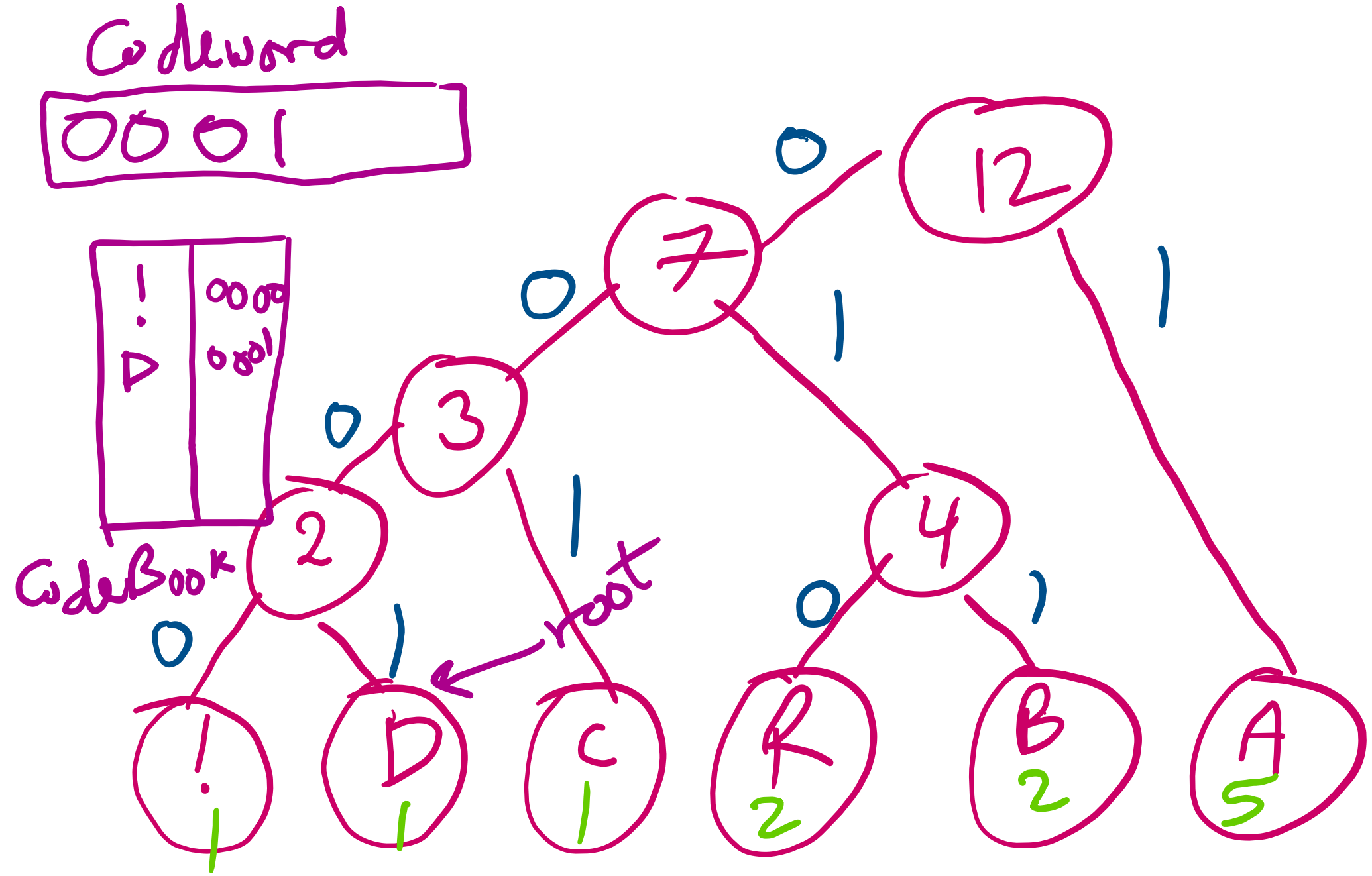
Huffman Compression: Generating the *Codebook*



Huffman Compression: Generating the *Codebook*



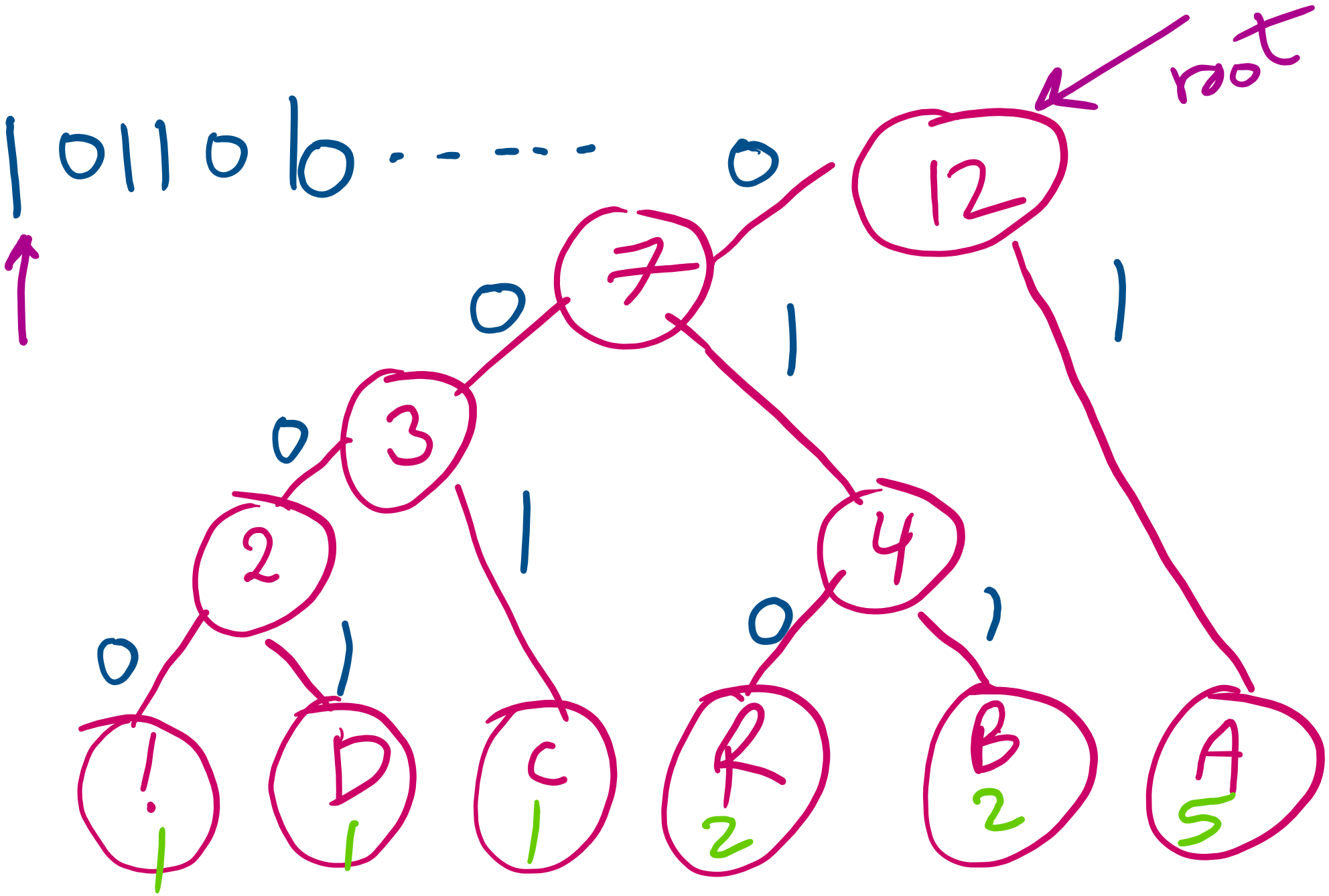
Huffman Compression: Generating the *Codebook*



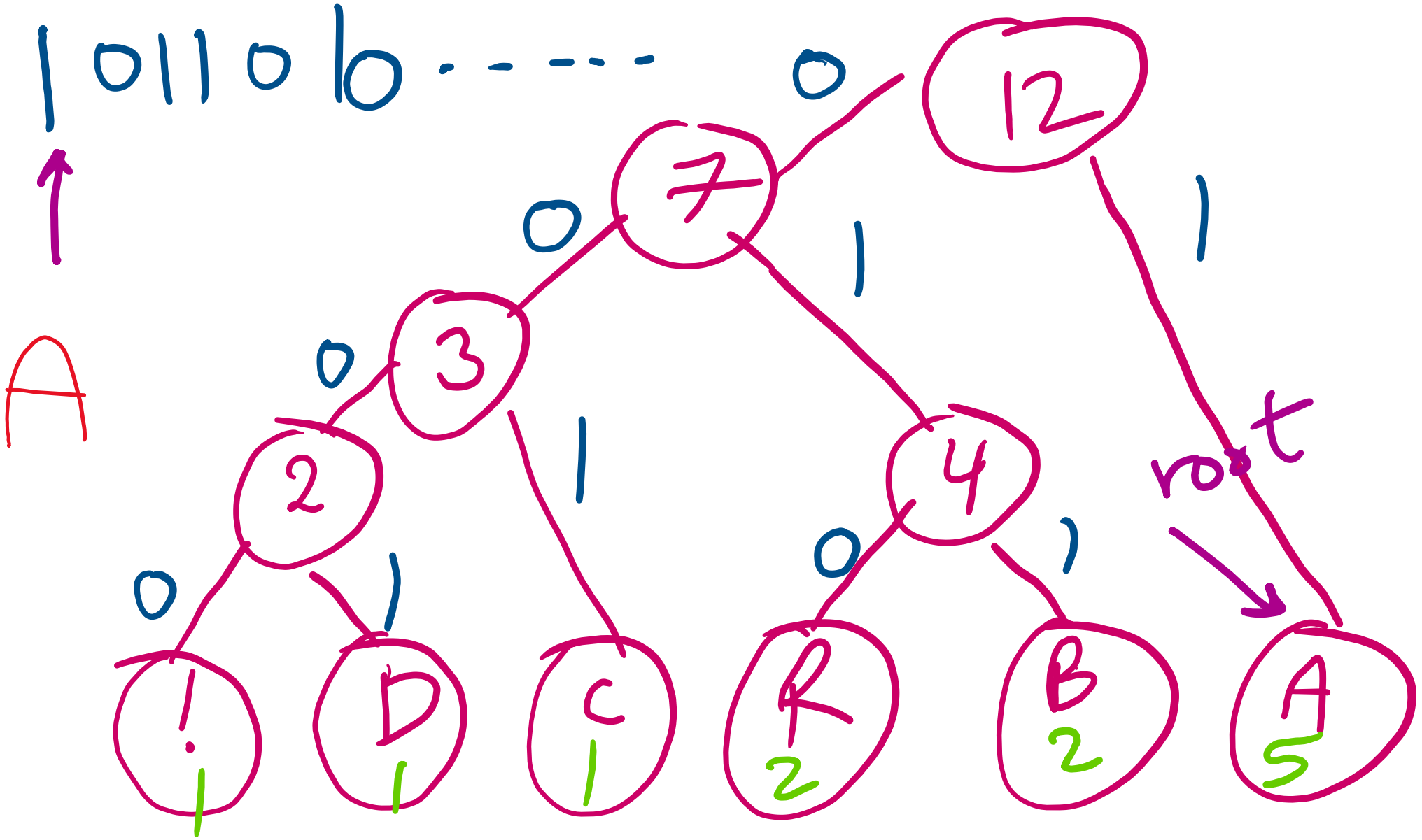
Huffman Compression: Generating the *Codebook*

```
void generateCodeBook(Node root, StringBuilder codeword) {
    if(root.isLeaf()){
        codebook[root.data] = codeword.toString();
    }
    if(root.left != null){
        //append 0 to codeword, move left, pop last character in codeword
    }
    if(root.right != null){
        //append 1 to codeword, move right, pop last character in codeword
    }
}
```

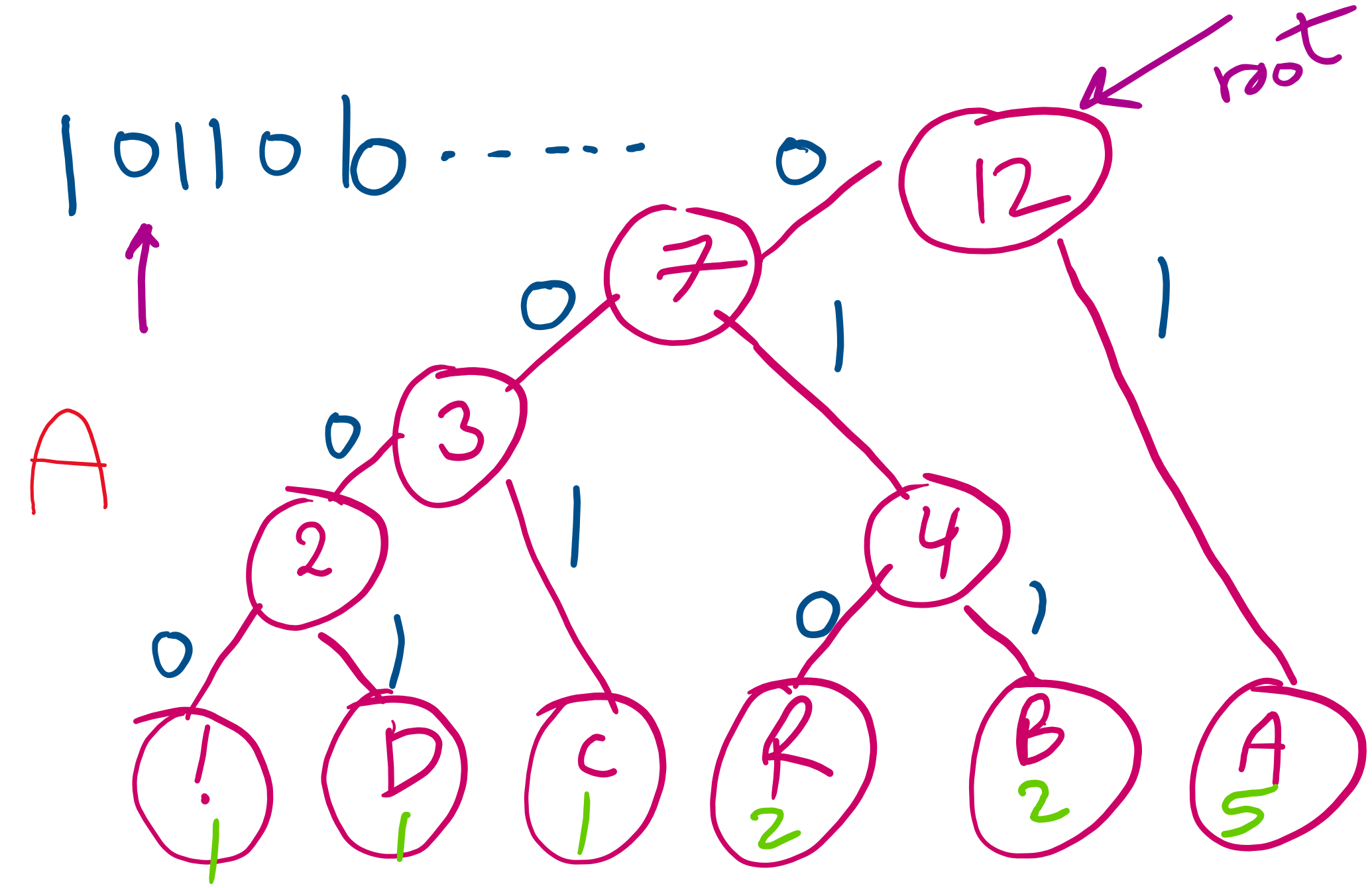
Huffman Compression: Decoding



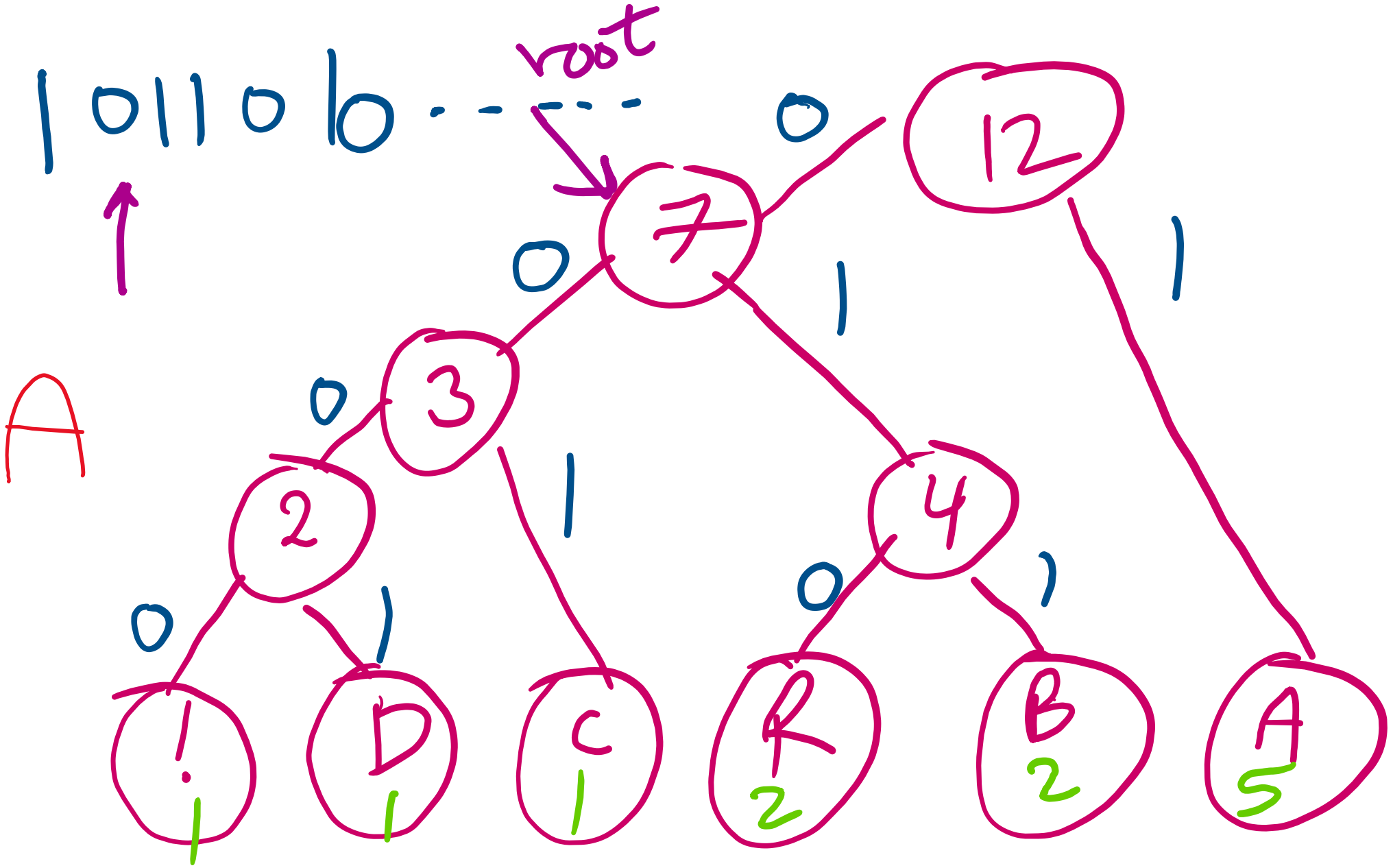
Huffman Compression: Decoding



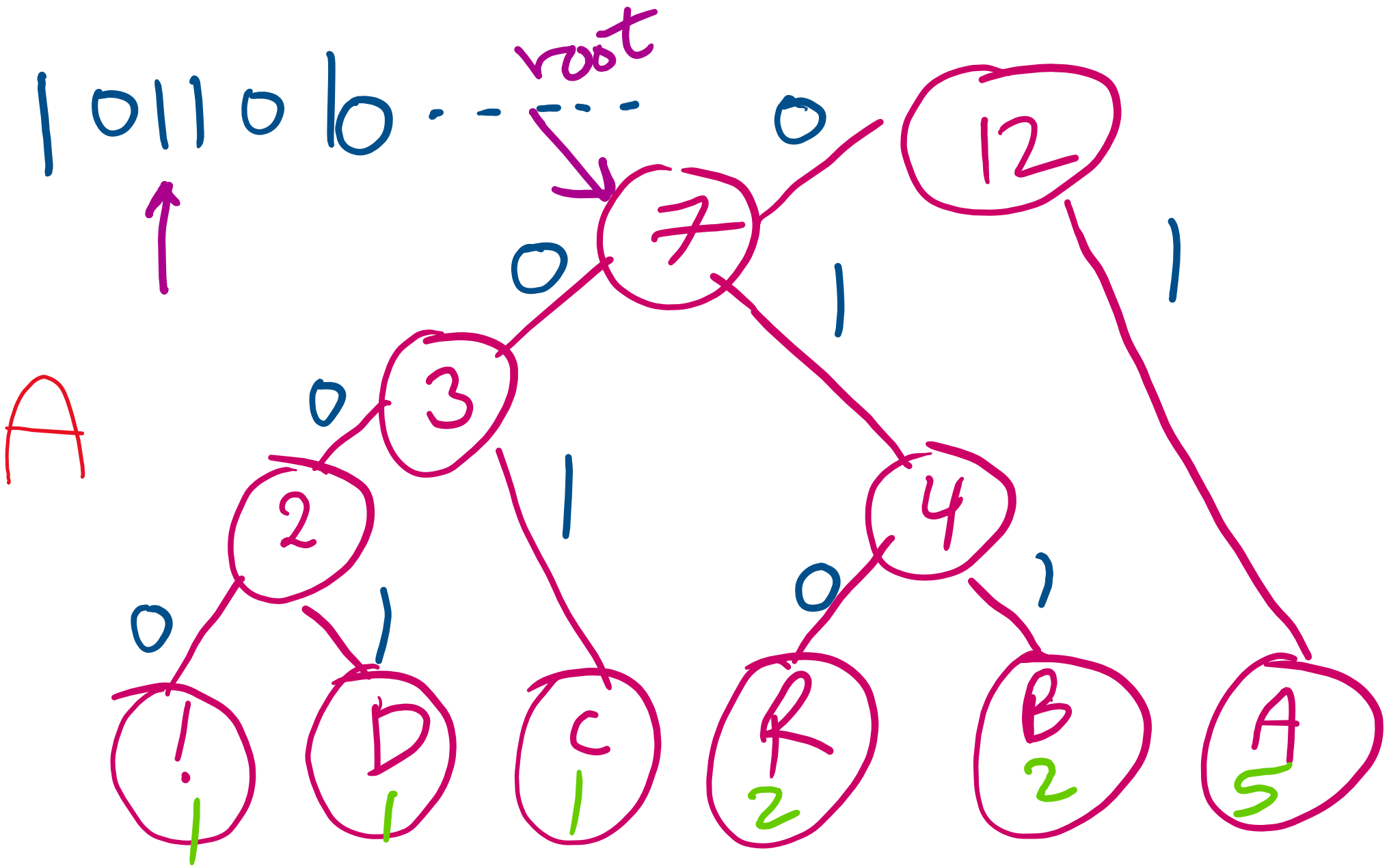
Huffman Compression: Decoding



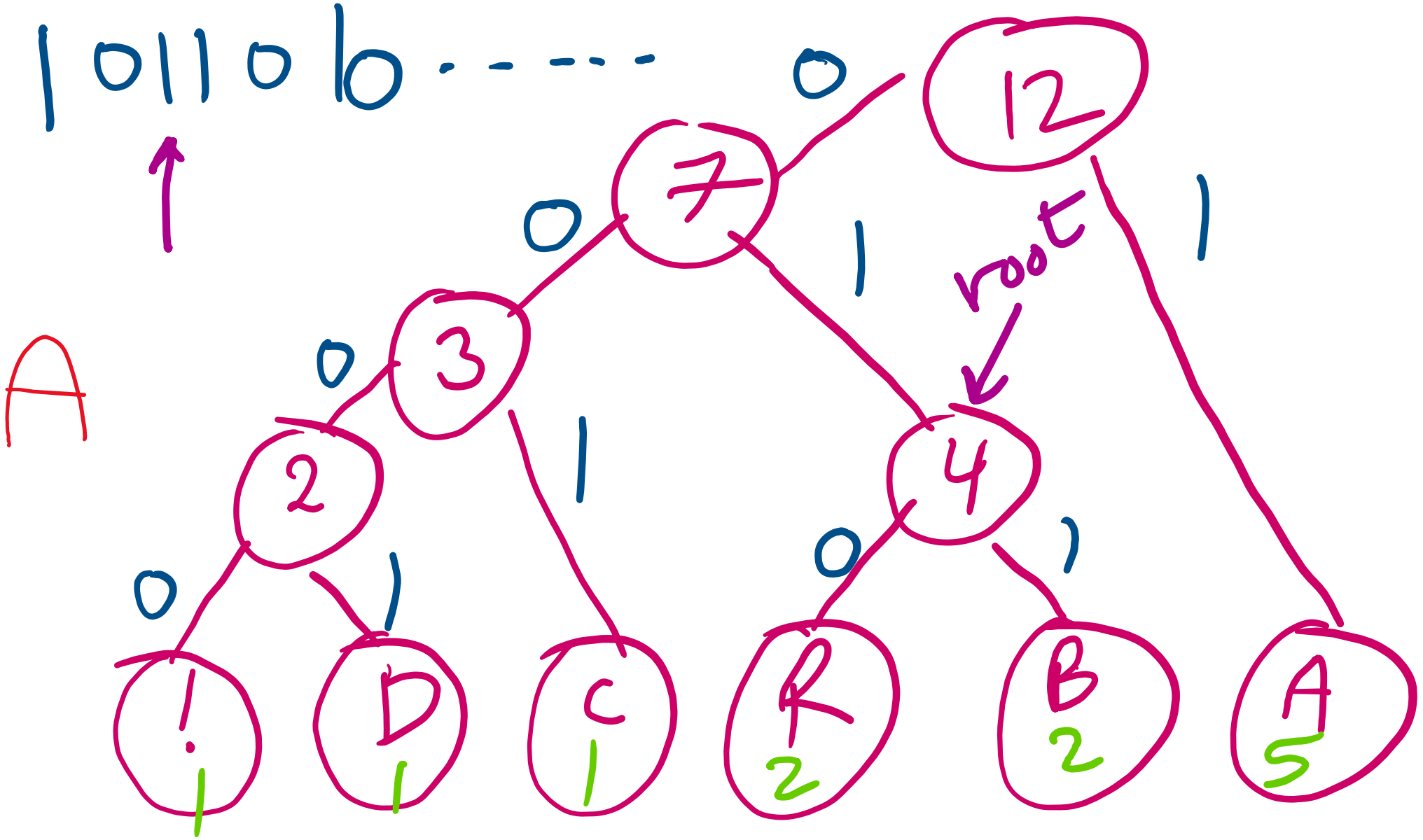
Huffman Compression: Decoding



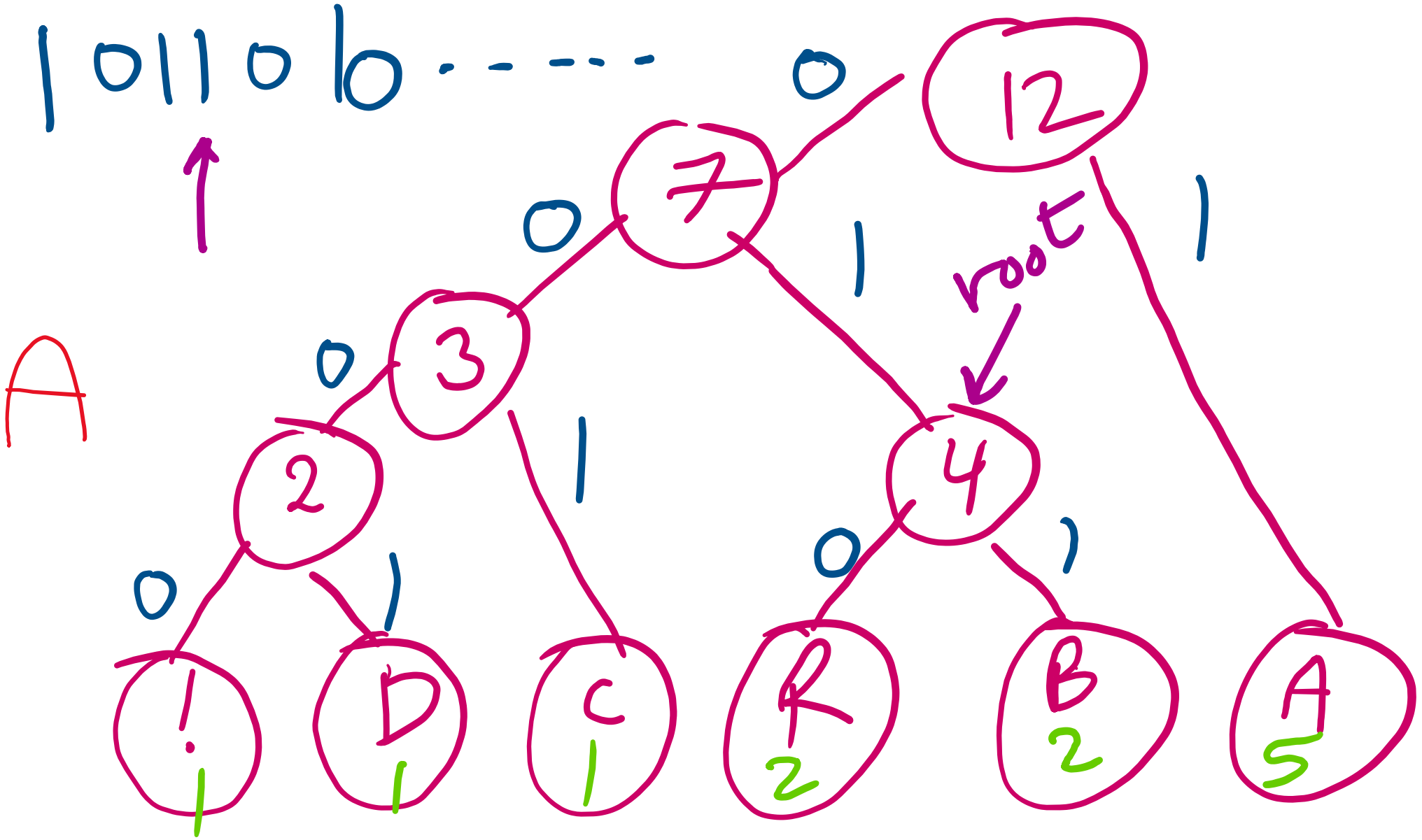
Huffman Compression: Decoding



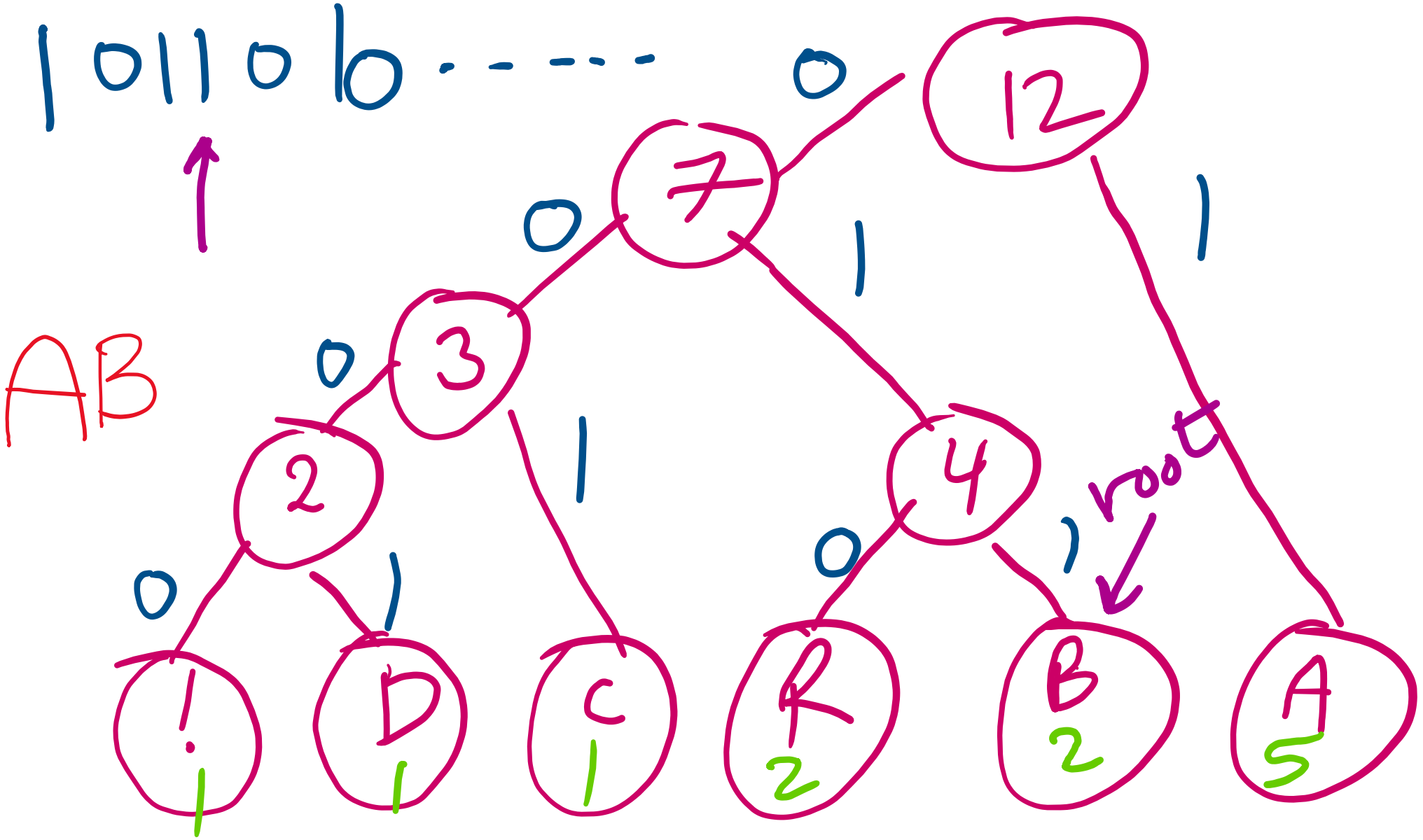
Huffman Compression: Decoding



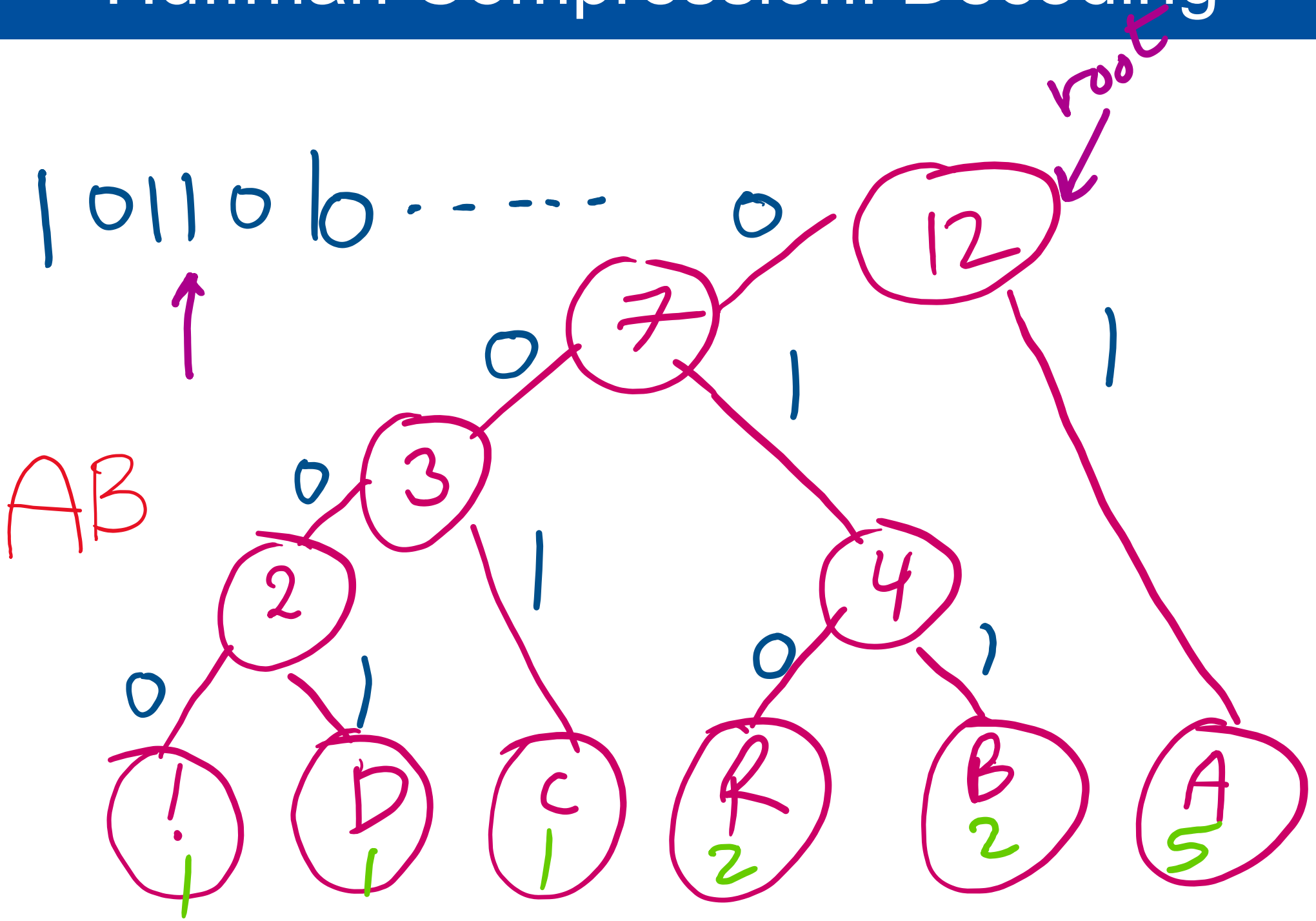
Huffman Compression: Decoding



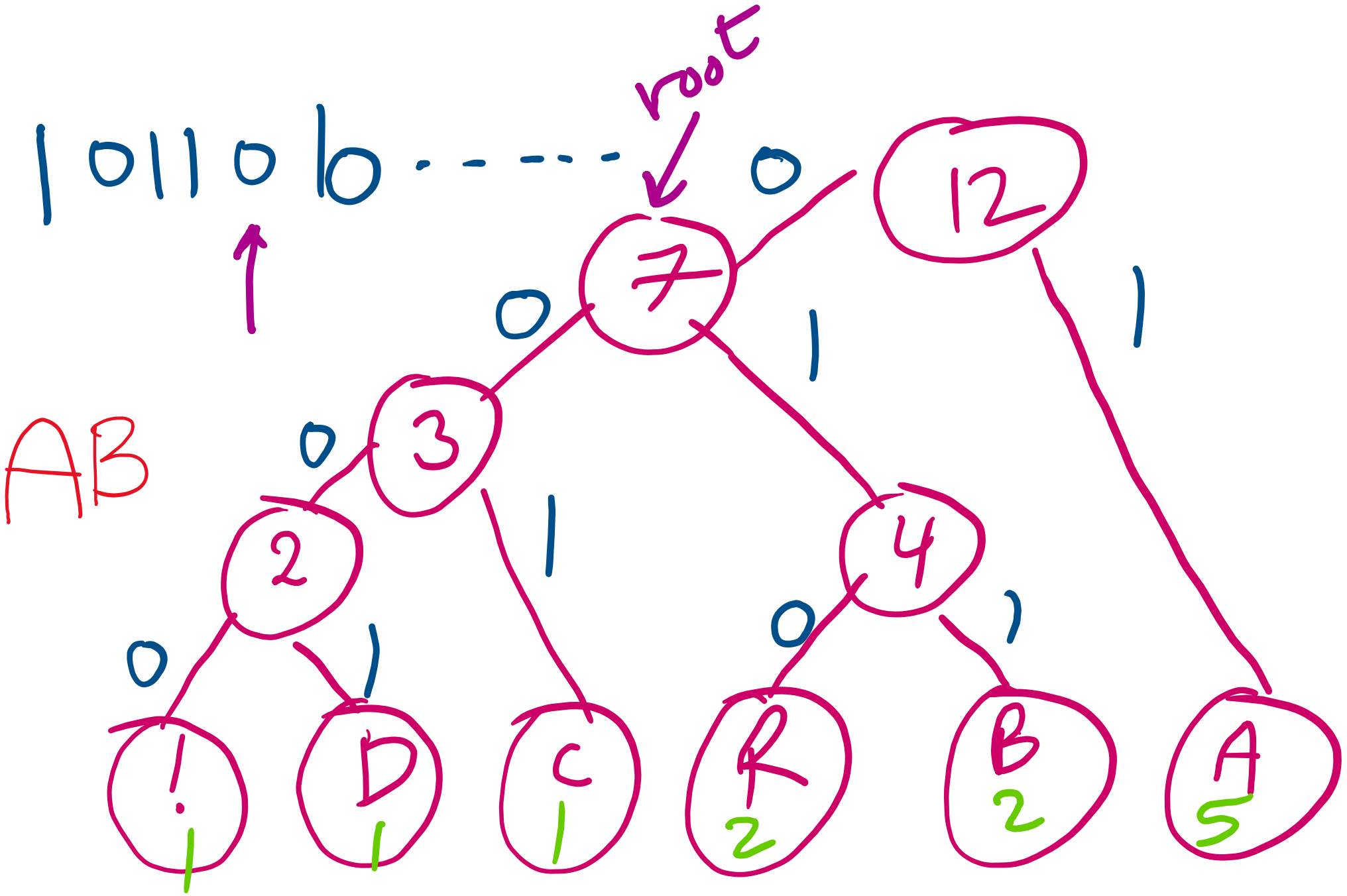
Huffman Compression: Decoding



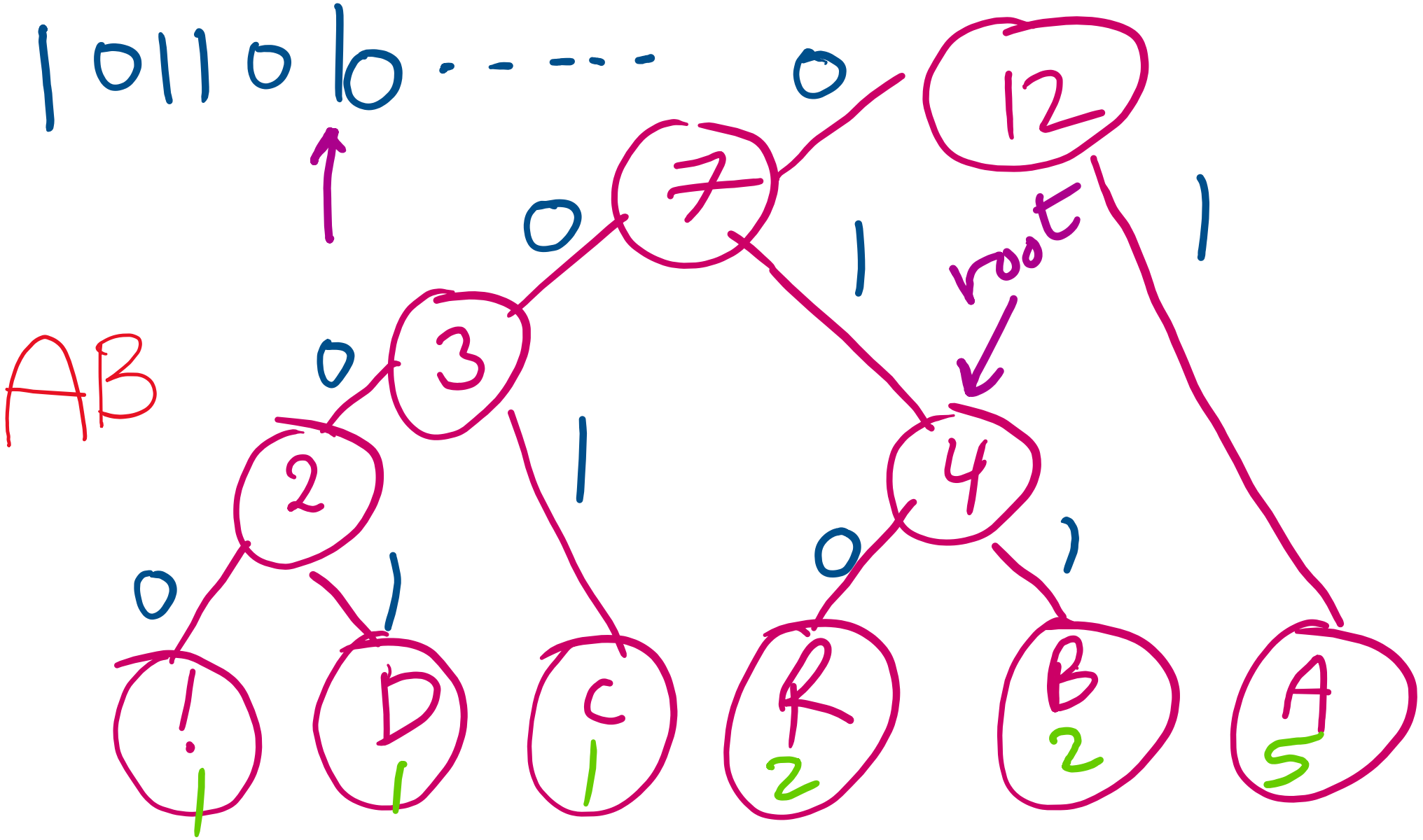
Huffman Compression: Decoding



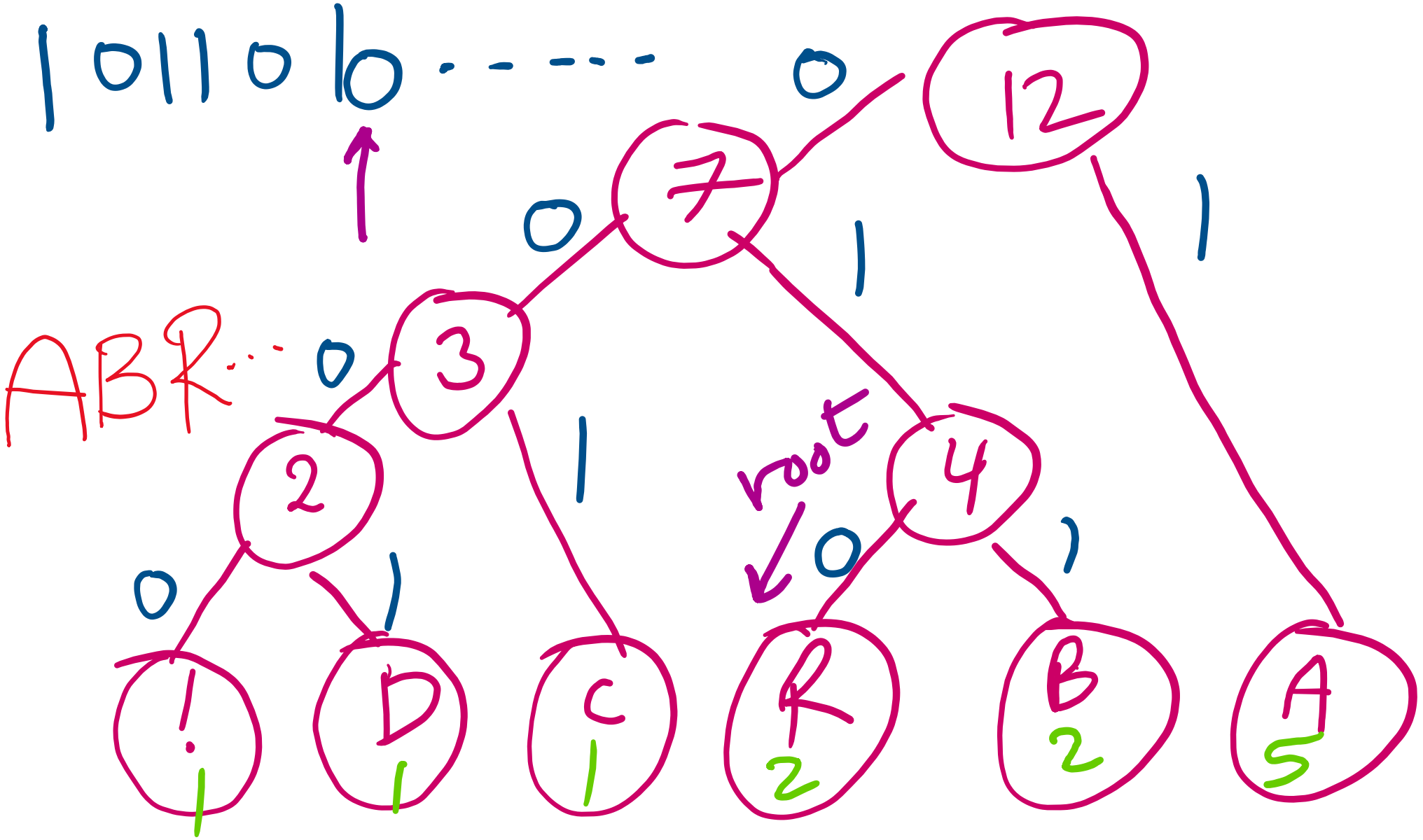
Huffman Compression: Decoding



Huffman Compression: Decoding



Huffman Compression: Decoding



How do we determine character frequencies?

- Option 1: Preprocess the file to be compressed
  - Upside: Ensure that Huffman's algorithm will produce the best output for the given file
  - Downsides:
    - Requires two passes over the input, one to analyze frequencies/build the trie/build the code lookup table, and another to compress the file
    - Trie must be stored with the compressed file, reducing the quality of the compression
      - This especially hurts small files
      - Generally, large files are more amenable to Huffman compression
        - Just because a file is large, however, does not mean that it will compress well!

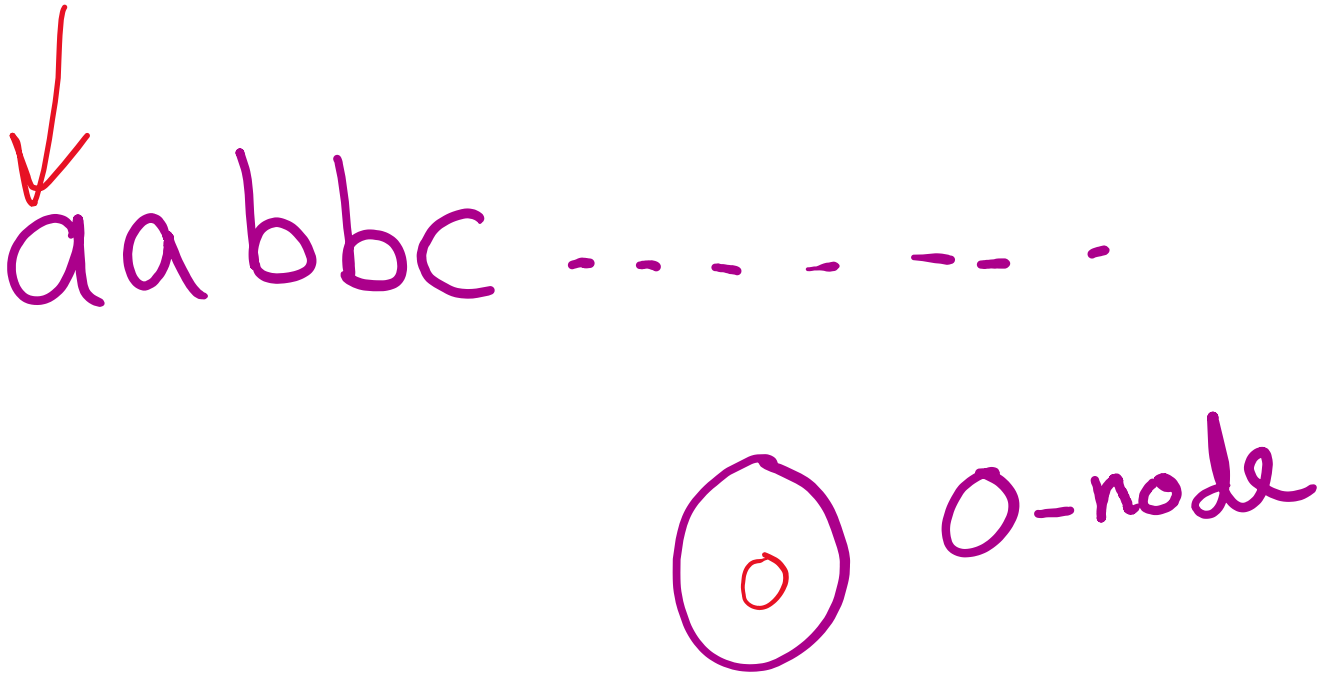
How do we determine character frequencies?

- Option 2: Use a static trie
  - Analyze multiple sample files, build a single tree that will be used for all compressions/expansions
  - Saves on trie storage overhead…
  - But in general not a very good approach
    - Different character frequency characteristics of different files means that a code set/trie that works well for one file could work very poorly for another
      - Could even cause an increase in file size after "compression"!

How do we determine character frequencies?

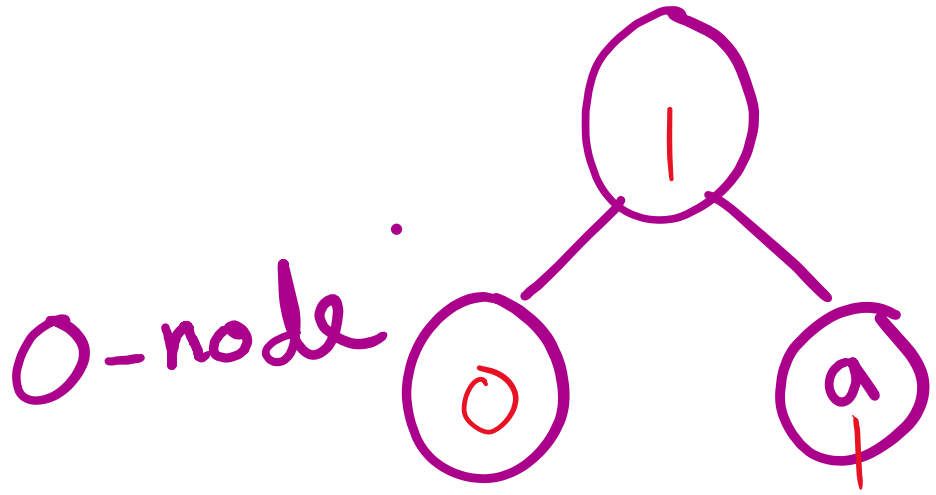
- Option 3: Adaptive Huffman coding
  - Single pass over the data to construct the codes and compress a file with no background knowledge of the source distribution
  - Not going to really focus on adaptive Huffman in the class, just pointing out that it exists...

Adaptive Huffman



Adaptive Huffman

aabbc.....

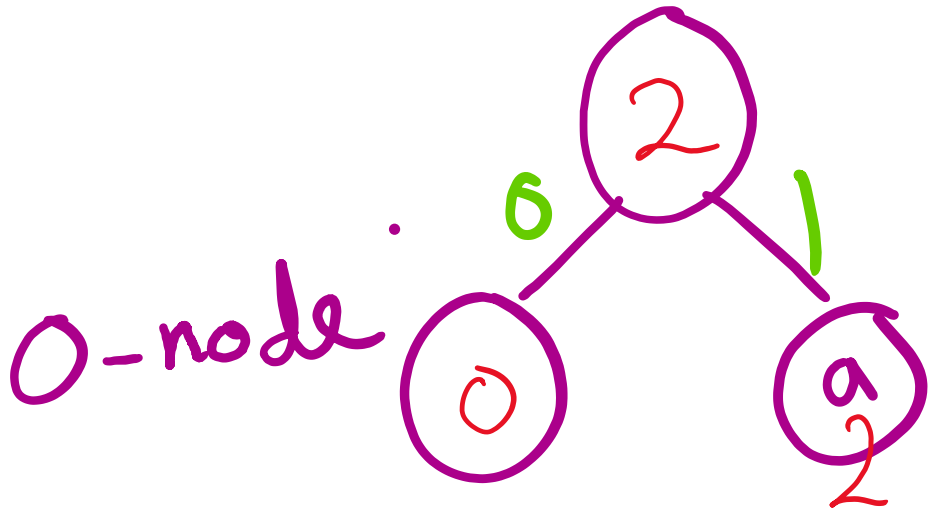


0-node

Ascii of a

Adaptive Huffman

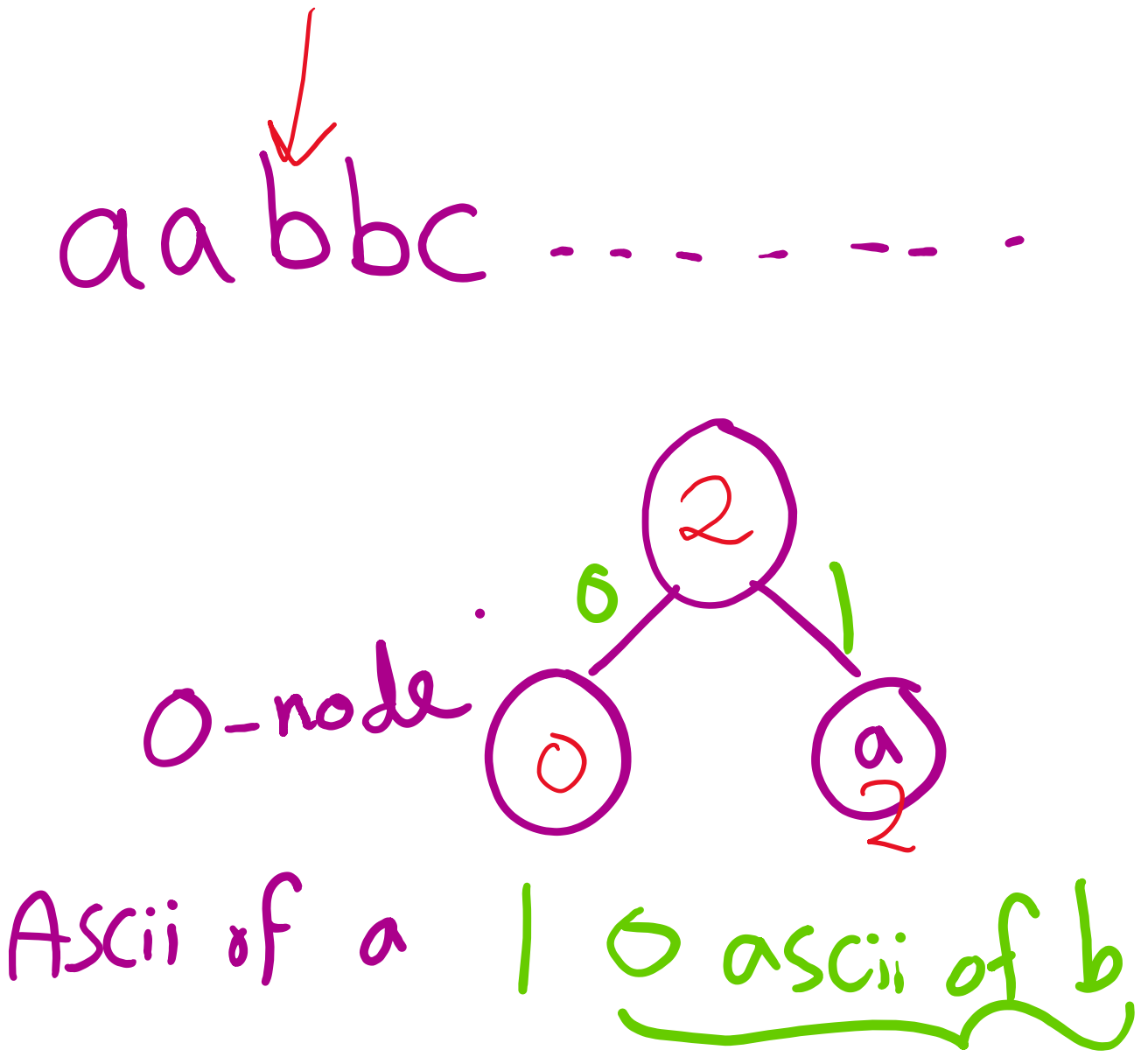
aabbc.....



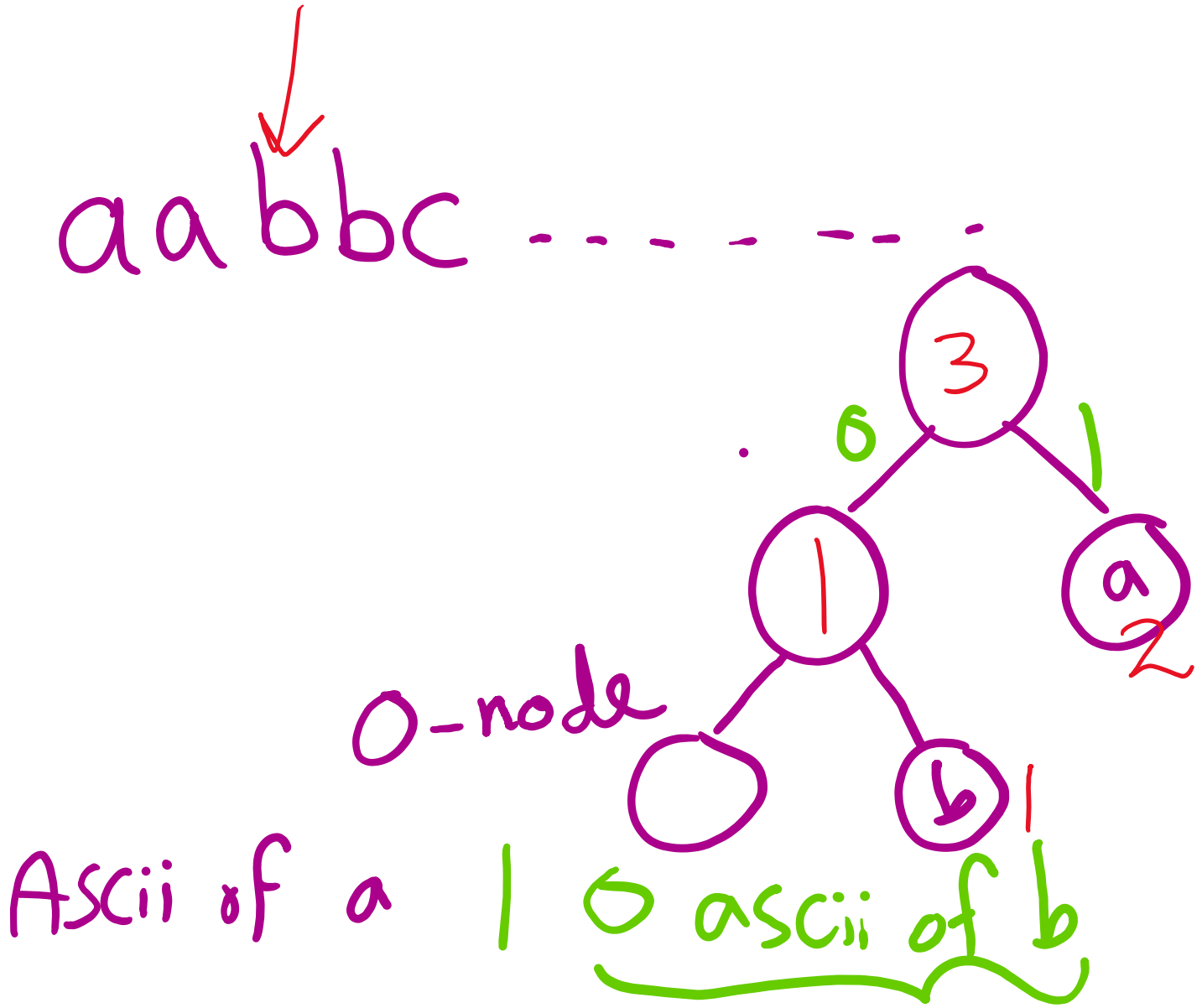
0-node

Ascii of a 1

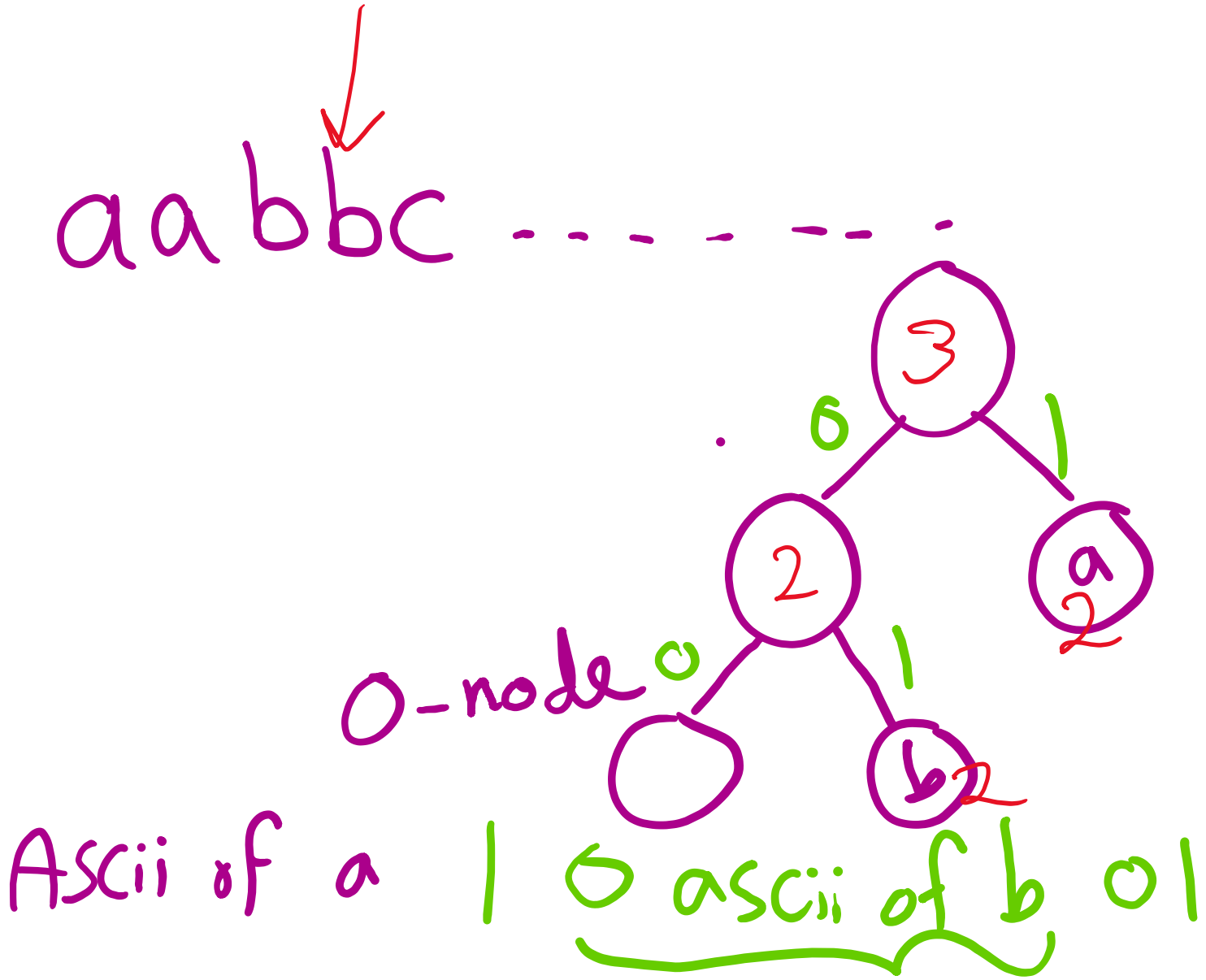
Adaptive Huffman



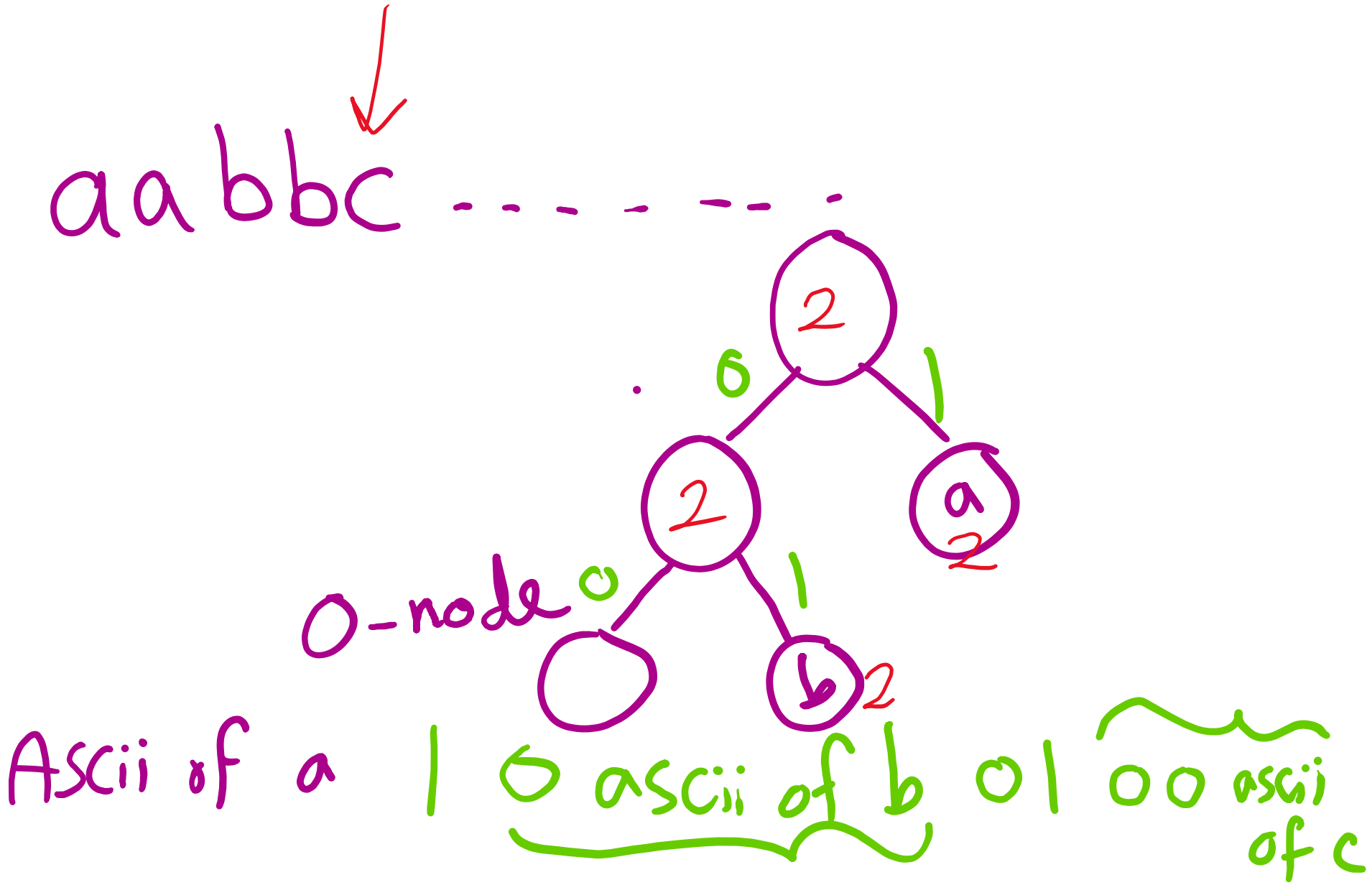
Adaptive Huffman



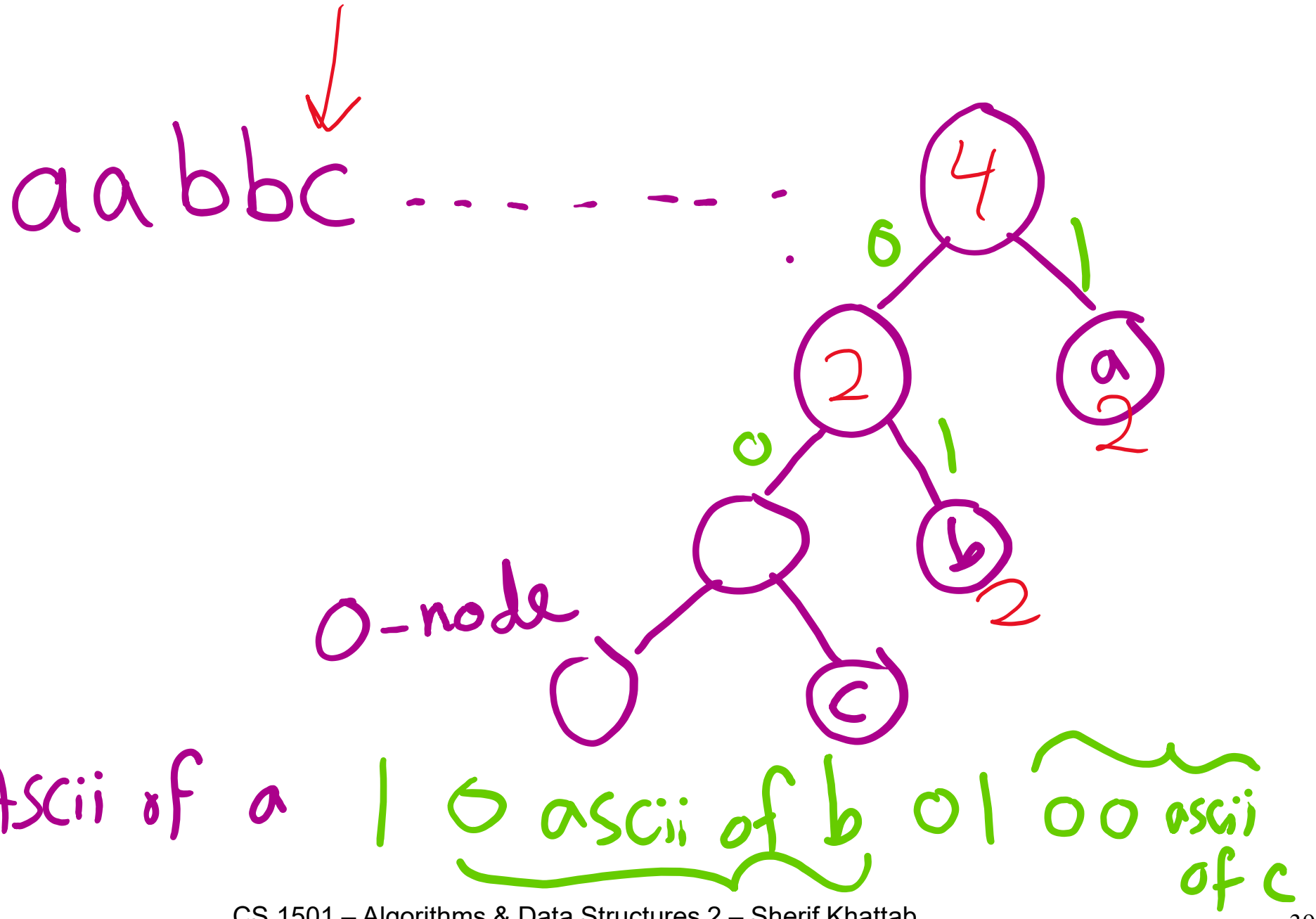
Adaptive Huffman



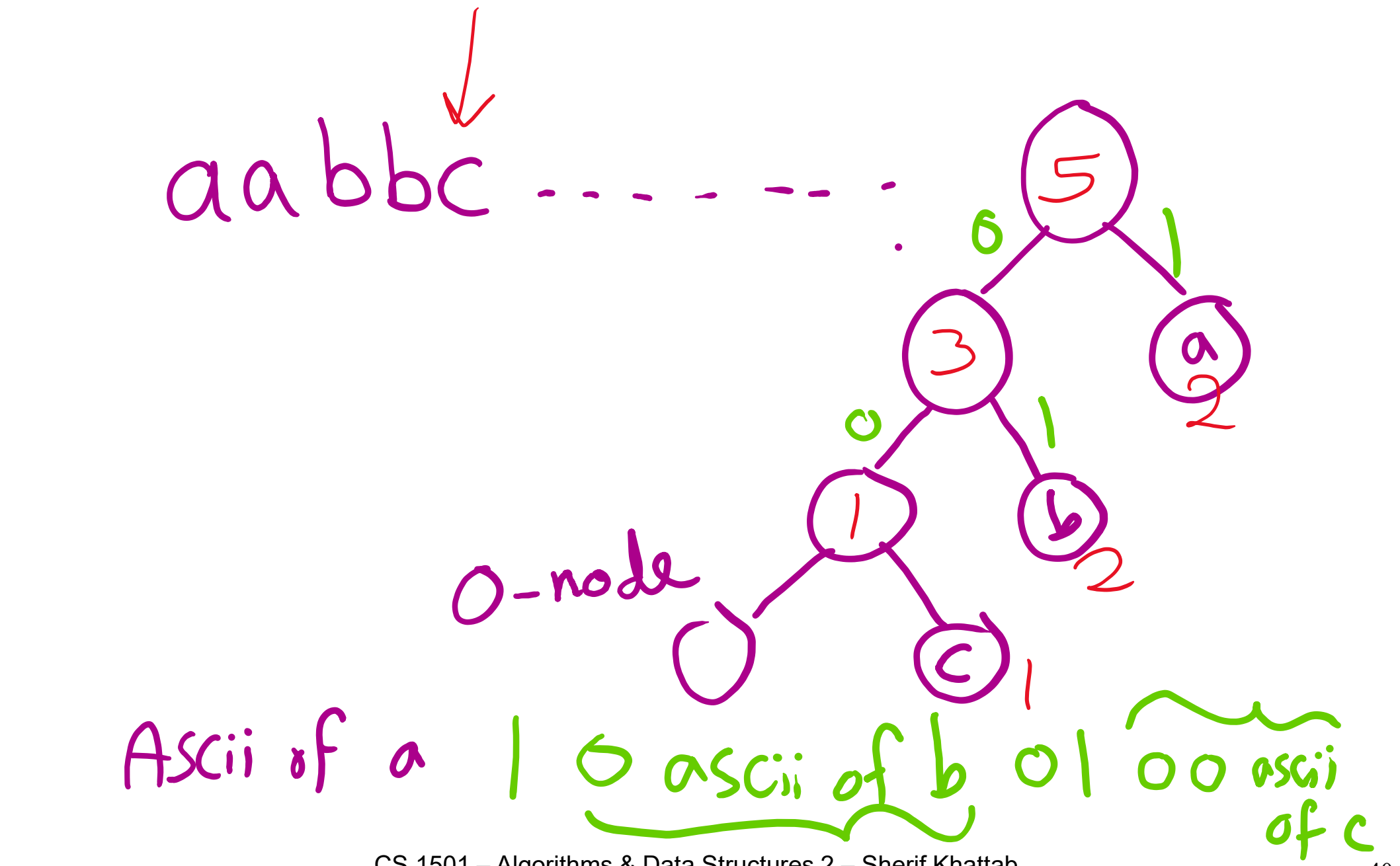
Adaptive Huffman



Adaptive Huffman

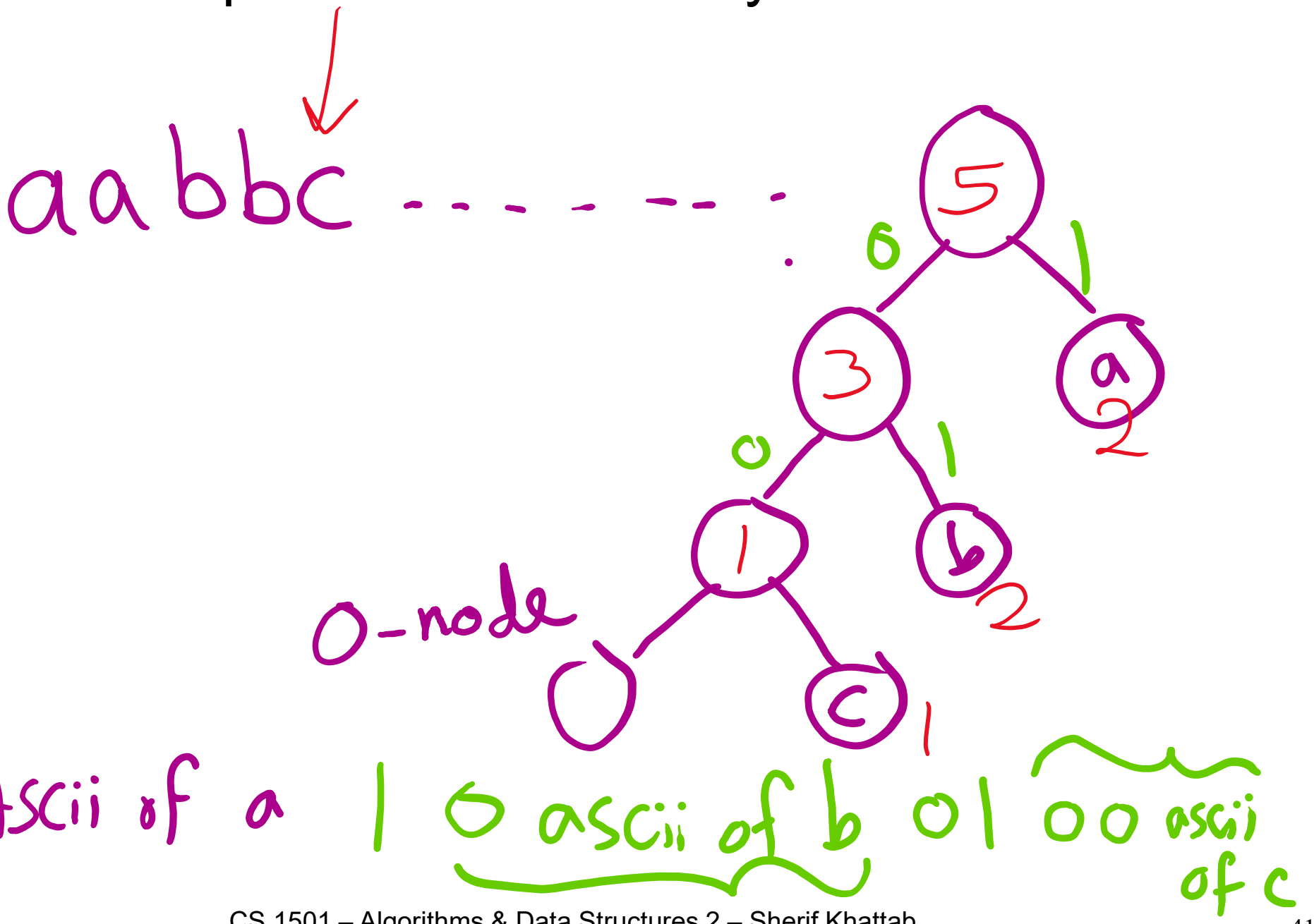


Adaptive Huffman



Adaptive Huffman

- need to swap nodes if necessary



Ok, so how good is Huffman compression

- ASCII requires 8n bits to store n characters
- For a file containing c different characters
  - Given Huffman codes $\{h_0, h_1, h_2, \ldots, h_{(c-1)}\}$
  - And frequencies $\{f_0, f_1, f_2, \ldots, f_{(c-1)}\}$
  - Sum from 0 to c-1: $|h_i| * f_i$
- Total storage depends on the differences in frequencies
  - The bigger the differences, the better the potential for compression
- Huffman is optimal for character-by-character prefix-free encodings
  - Proof in Propositions T and U of Section 5.5 of the text
- If all characters in the alphabet have the same usage frequency, we can't beat block code (fixed-size codewords)
  - Huffman reduces to fixed-size codewords
  - On a character by character basis…

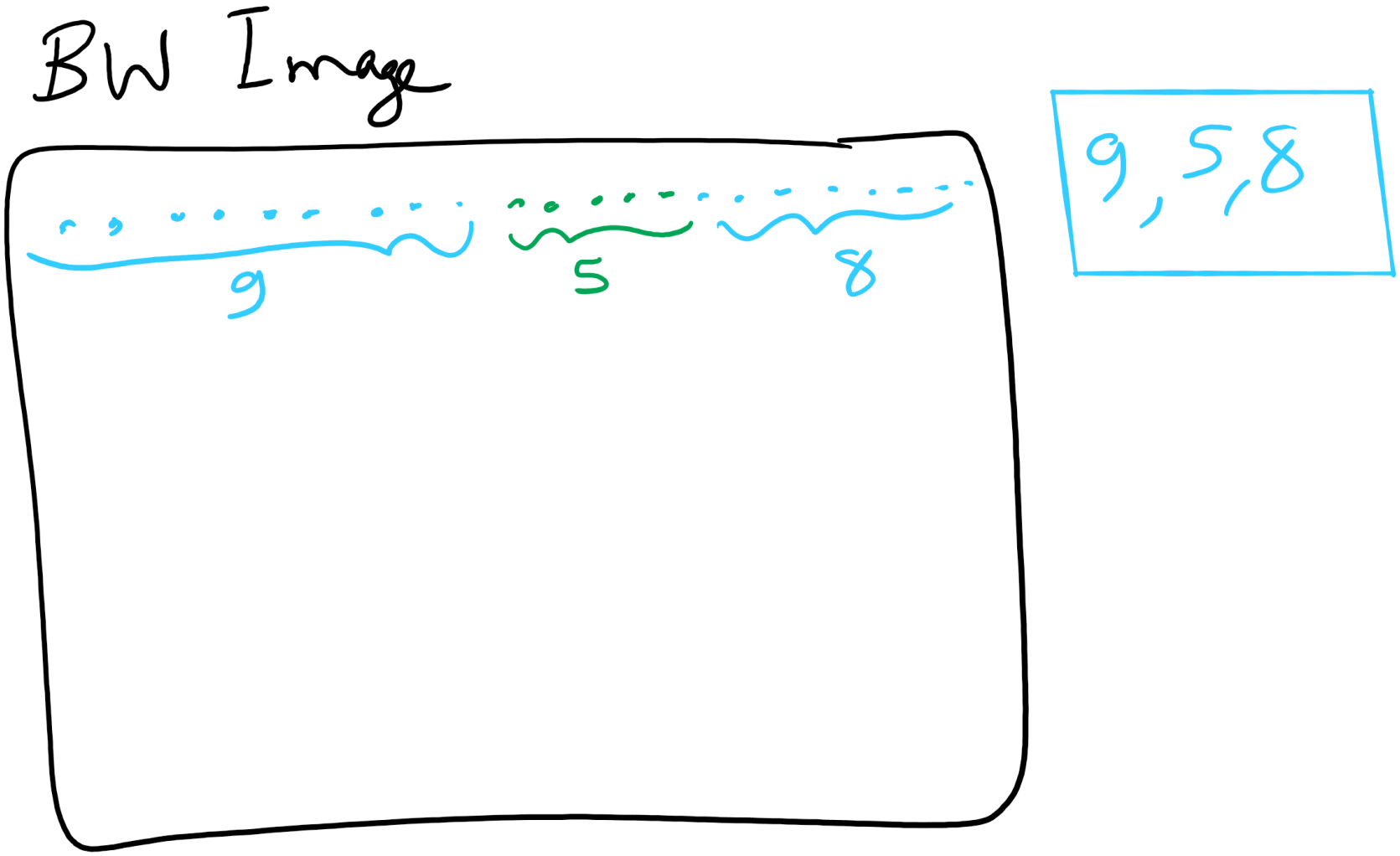
That seems like a bit of a caveat...

- Where does Huffman fall short?
  - What about repeated patterns of multiple characters?
    - Consider a file containing:
      - 1000 A's
      - 1000 B's
      - ...
      - 1000 of every ASCII character
    - Will this compress at all with Huffman encoding?
      - Nope!
    - But it seems like it should be compressible...

Run length encoding

- Could represent the previously mentioned string as:
  - 1000A1000B1000C, etc.
    - Assuming we use 10 bits to represent the number of repeats, and 8 bits to represent the character…
      - 4608 bits needed to store run length encoded file
      - vs. 2048000 bits for input file
      - Huge savings!
- Note that this incredible compression performance is based on a very specific scenario…
  - Run length encoding is not generally effective for most files, as they often lack long runs of repeated characters

Run-length Encoding



What else can we do to compress files?

Patterns are compressible, need a general approach

- Huffman used variable-length codewords to represent fixed-length portions of the input…
  - Let's try another approach that uses fixed-length codewords to represent variable-length portions of the input
- Idea: the more characters can be represented by a single codeword, the better the compression
  - Consider "the": 24 bits in ASCII
  - Representing "the" with a single 12 bit codeword cuts the used space in half
    - Similarly, representing longer strings with a 12 bit codeword would mean even better savings!

How do we know that “the” will be in our file?

- Need to avoid the same problems as the use of a static trie for Huffman encoding…
- So use an adaptive algorithm and build up our patterns and codewords **as we go** through the file

LZW compression

- Initialize codebook to all single characters
  - e.g., character maps to its ASCII value
  - codewords 0 to 255 are filled now
- While !EOF: (EOF is End Of File)
  - Find the longest match in codebook
  - Output codeword of that match
  - Take this longest match + the next character in the file
    - add that to the codebook with the next available codeword value
  - Start from the character right after the match

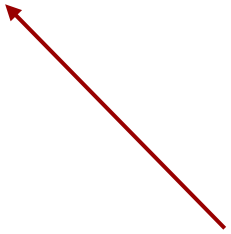
LZW compression example

- Compress, using 12 bit codewords:
  - TOBEORNOTTOBEORTOBEORNOT

Cur	Output	Add
T	84	TO:256
O	79	OB:257
B	66	BE:258
E	69	EO:259
O	79	OR:260
R	82	RN:261
N	78	NO:262
O	79	OT:263
T	84	TT:264
TO	256	TOB:265
BE	258	BEO:266
OR	260	ORT:267
TOB	265	TOBE:268
EO	259	EOR:269
RN	261	RNO:270
OT	263	--

LZW expansion

- Initialize codebook to all single characters
  - e.g., ASCII value maps to its character
- While !EOF:
  - Read next codeword from file
  - Lookup corresponding pattern in the codebook
  - Output that pattern
  - Add the previous pattern + the first character of the current pattern to the codebook



Note this means no codebook addition after first pattern output!

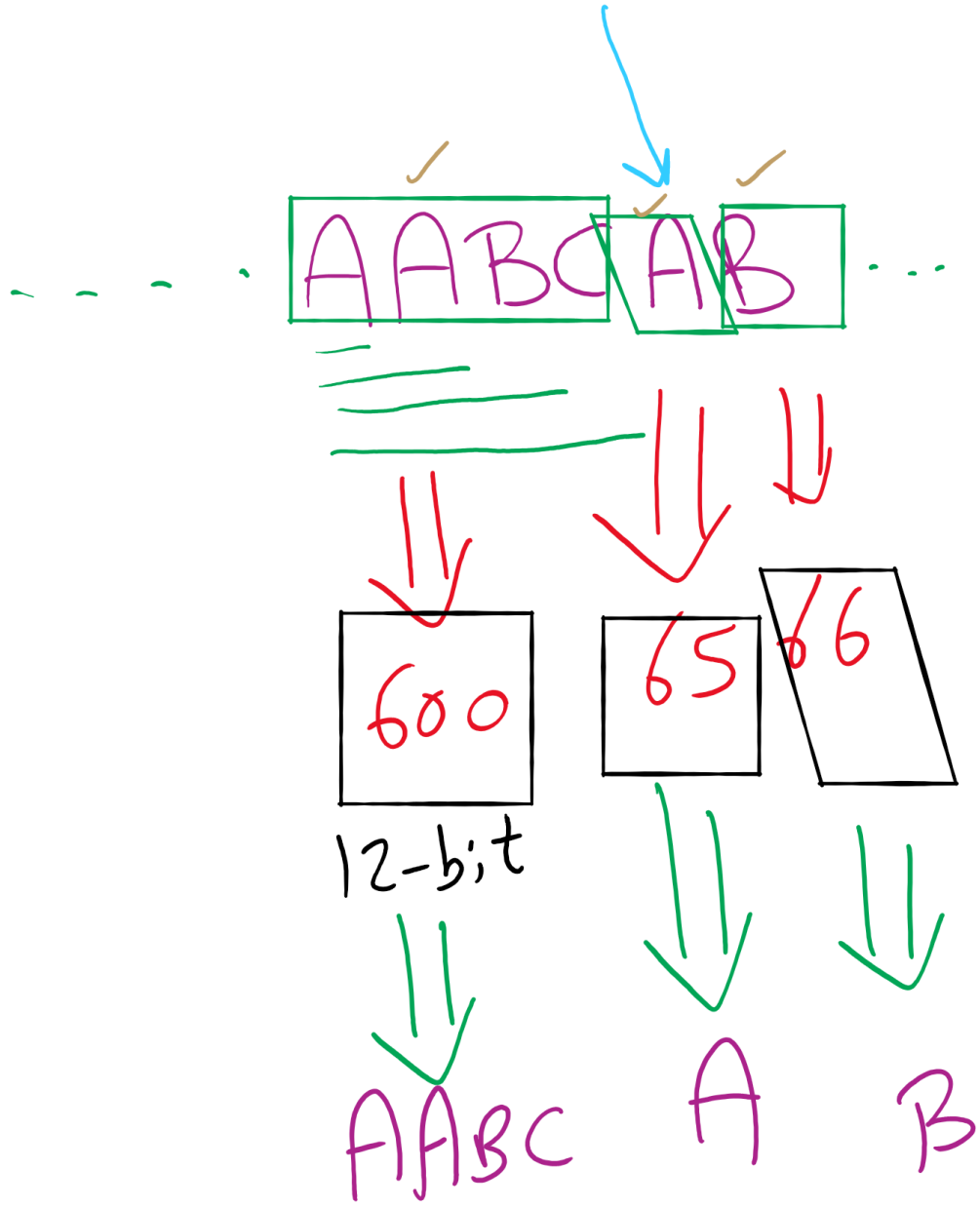
LZW expansion example

Cur	Output	Add
84	T	--
79	O	256:TO
66	B	257:OB
69	E	258:BE
79	O	259:EO
82	R	260:OR
78	N	261:RN
79	O	262:NO
84	T	263:OT
256	TO	264:TT
258	BE	265:TOB
260	OR	266:BEO
265	TOB	267:ORT
259	EO	268:TOBE
261	RN	269:EOR
263	OT	270:RNO

How does this work out?

- Both compression and expansion construct the same codebook!
  - Compression stores character string → codeword
  - Expansion stores codeword → character string
  - They contain the same pairs in the same order
    - Hence, the codebook doesn't need to be stored with the compressed file, saving space

LZW Example



A	65
B	66
AA	300
AAB	400
AABB	500
AABC	600
AABCA	601
AB	602

Just one tiny little issue to sort out...

- Expansion can sometimes be a step ahead of compression…
  - If, during compression, the (pattern, codeword) that was just added to the dictionary is immediately used in the next step, the decompression algorithm will not yet know the codeword.
  - This is easily detected and dealt with, however

LZW corner case example

- Compress, using 12 bit codewords: AAAAAA

Cur	Output	Add
A	65	AA:256
AA	256	AAA:257
AAA	257	--

- Expansion:

Cur	Output	Add
65	A	--
256	AA	256:AA
257	AAA	257:AAA

LZW Corner Case