LZW String encoding (and decoding)

CMSC420

This material is not covered during the Fall of 2019, but we include it so that you can learn some things about LZW compression.

String compression

- So far we've dealt with efficient character **encoding**
  - An allocation of bit length to characters that, over the long run, will save us space (and processing time at the destination!)
- Technically, this is still string compression
  - But, it turns out we can do better!

Compression: Lossy vs Lossless

- A different kind of debate when compared to character encoding strategies.
- **Lossy**: describes a compression process during which **some data loss is incurred**.
- **Lossless**: not lossy!
- Different applications have different needs.
  - **Audio:** highly compressible. Lossy compression (mp3/mp4) with variable bitrate (128, 160, … , 320 kb/s)
  - **Images**: **JPEG** and **JPEG2000** algorithms both lossy, but can compress an image 10 to 1 with virtually **no perceived quality loss.**
  - **Text:** Here, we can't risk mistakes**. Lossless** compression is the only way
    - Patent documents
    - Courtroom proceedings
    - University admission decisions
    - ….

Concatenating Huffman encodings

- We *could* concatenate the Huffman encodings of the individual characters…
- But this doesn't take into consideration that, in language, groups of consecutive characters (n-grams), are very often used repeatedly!

Concatenating Huffman encodings

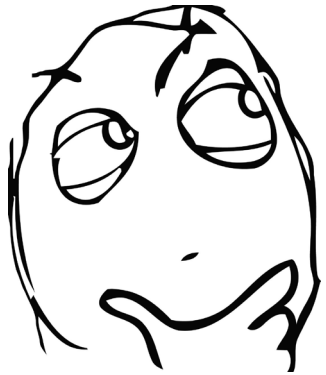
- We *could* concatenate the Huffman encodings of the individual characters…
- But this doesn't take into consideration that, in language, groups of consecutive characters (n-grams), are very often used repeatedly!
  - For example: "th" is used in "the, that, than, there, these, those, thor".

Concatenating Huffman encodings

- We *could* concatenate the Huffman encodings of the individual characters…
- But this doesn't take into consideration that, in language, groups of consecutive characters (n-grams), are very often used repeatedly!
  - For example: "th" is used in "the, that, than, there, these, those, thor".
    - But wait! "the" was also used more than once, in the words "the", "there" and "these"!
    - "tha" was also used twice, and "tho" was used twice as well!

Concatenating Huffman encodings

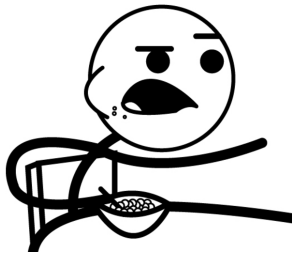
- We *could* concatenate the Huffman encodings of the individual characters…
- But this doesn't take into consideration that, in language, groups of consecutive characters (n-grams), are very often used repeatedly!
  - For example: "th" is used in "the, that, than, there, these, those, thor".
    - But wait! "the" was also used more than once, in the words "the", "there" and "these"!
    - "tha" was also used twice, and "tho" was used twice as well!
- If only we could somehow cheaply encode n-grams…



Recall the pipeline…

Recall the pipeline…

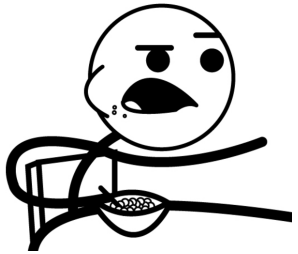
UNICODE



No storage standard

Recall the pipeline…

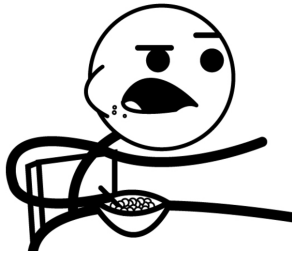
UNICODE



No storage standard

Recall the pipeline…

UNICODE → UCS-2

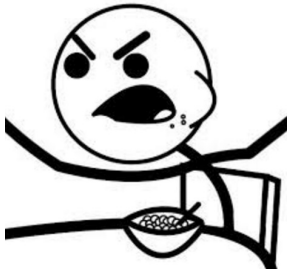


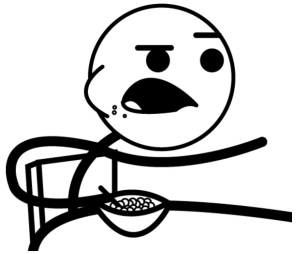
No storage standard

Recall the pipeline…

UNICODE → UCS-2

Rare symbols should pay more than frequent ones!





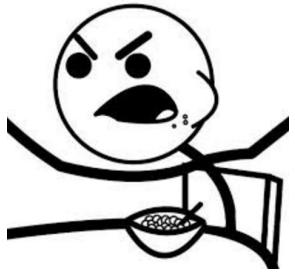
No storage standard

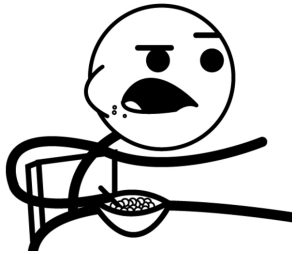
Recall the pipeline…

UNICODE → UCS-2 → UTF-8

UTF-16

Rare symbols should pay more than frequent ones!

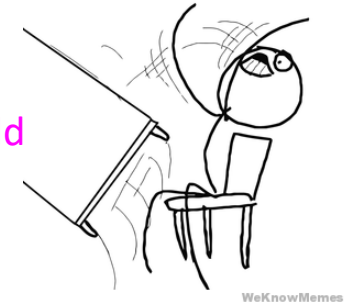




No storage standard

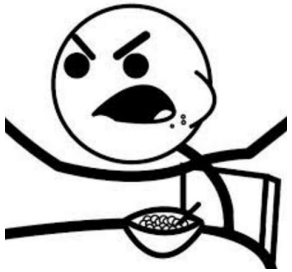
Recall the pipeline…

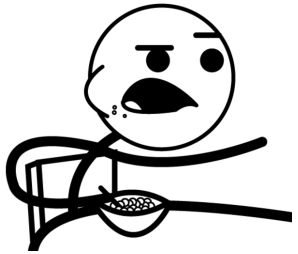
Unigram frequencies should be leveraged for better **data compression**



UNICODE → UCS-2 → UTF-8 / UTF-16

Rare symbols should pay more than frequent ones!

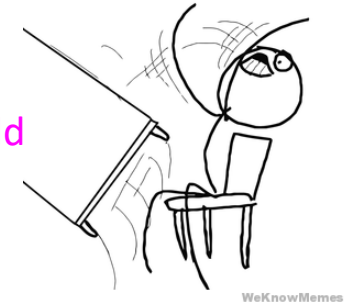




No storage standard

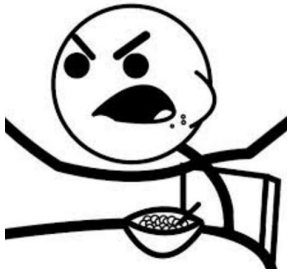
Recall the pipeline…

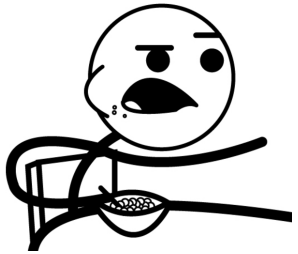
Unigram frequencies should be leveraged for better **data compression**



UNICODE → UCS-2 → UTF-8 / UTF-16 → Huffman

Rare symbols should pay more than frequent ones!



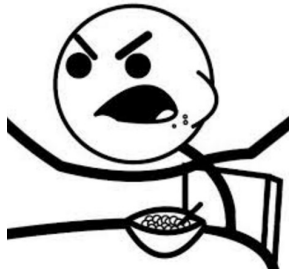


No storage standard

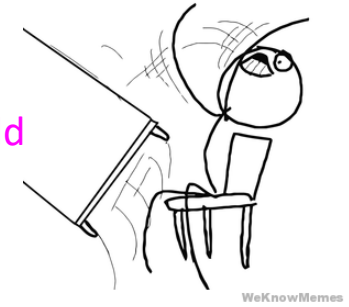
Recall the pipeline…

UNICODE → UCS-2 → UTF-8 / UTF-16 → Huffman

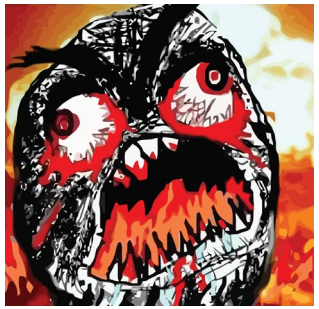
Rare symbols should pay more than frequent ones!

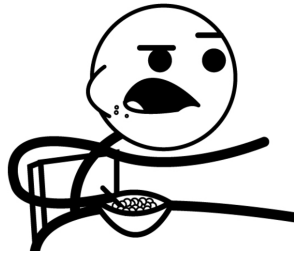


Unigram frequencies should be leveraged for better **data compression**



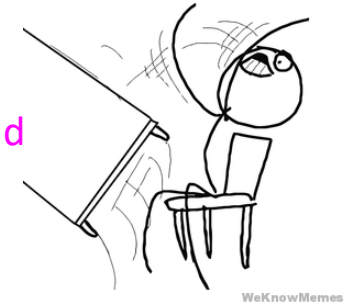
Bi-Tri/… N-gram frequencies should be **leveraged for better data compression!**





No storage standard

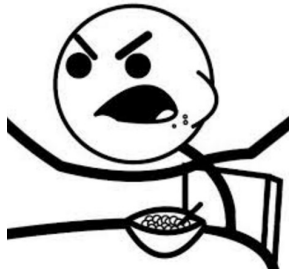
Recall the pipeline…

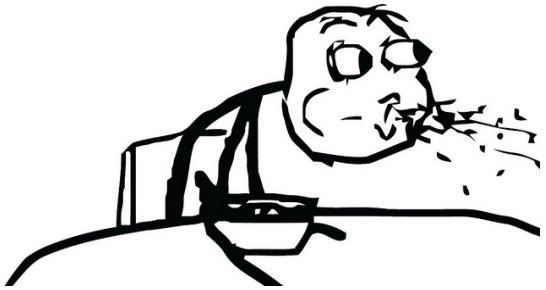


Unigram frequencies should be leveraged for better **data compression**

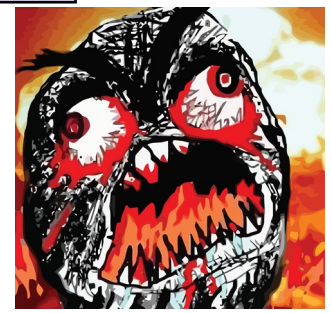
UNICODE → UCS-2 → UTF-8 / UTF-16 → Huffman → Lempel-Ziw-Welch (LZW) String Compression Algorithm

Rare symbols should pay more than frequent ones!





Bi-Tri/… N-gram frequencies should be **leveraged for better data compression!**



Naïve approach won't work

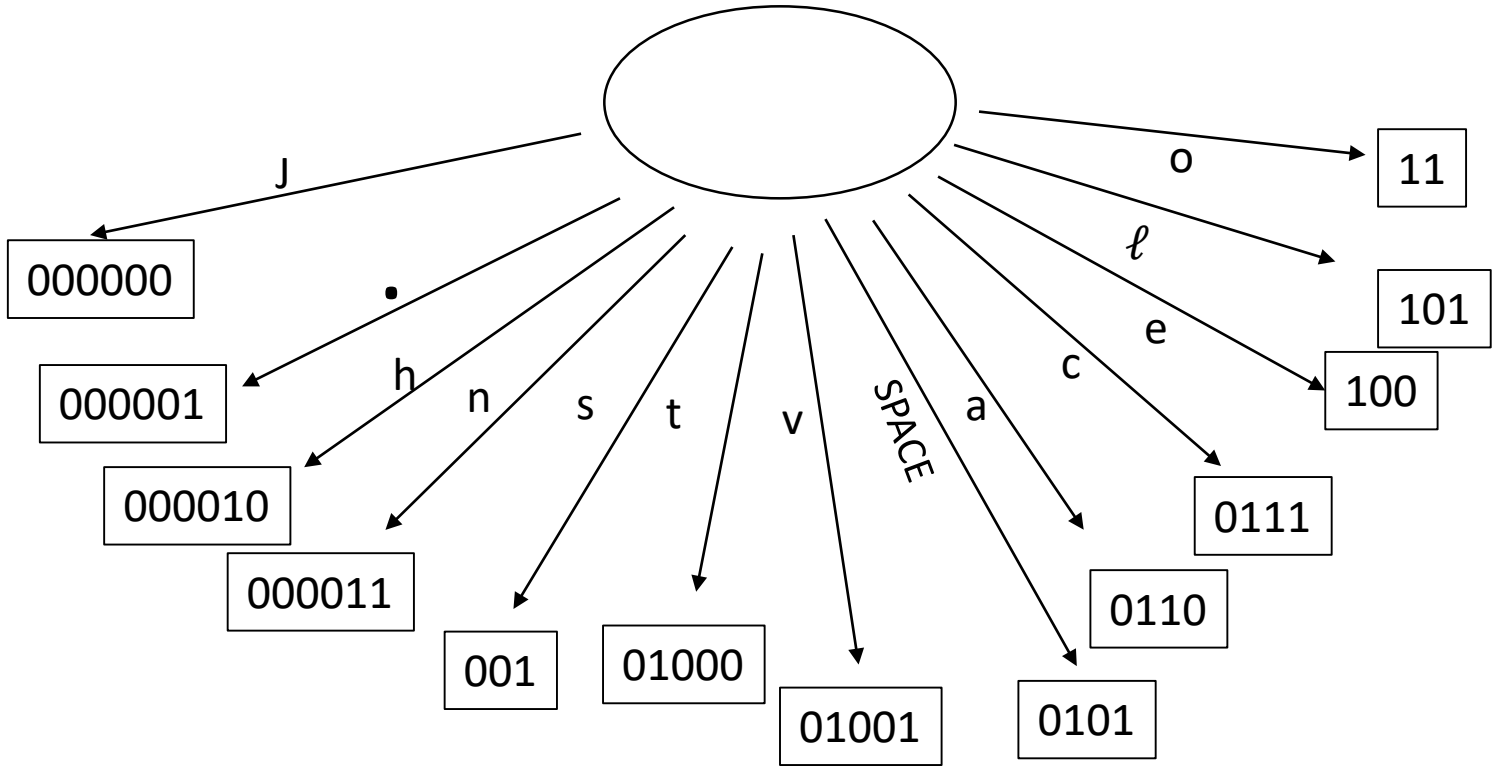
- We could start thinking about encoding bigrams, trigrams,… , n-grams the same way we did with Huffman coding
- Problem: This can become expensive real fast ☹
- In time
  - For every character, read $n-1$ positions up front to populate histogram.
  - This leads to $\mathcal{O}(|T| * (n-1))$ time for building the histogram.
- And in space!
    - The leaves are polynomially many more than the previous single-character leaves $(P(|\Sigma|, n)$,
    - The histogram of n-grams would be quite sparse, so many n –grams would share the same bit length…

Lookup tables and “source” tries

- Huffman ended with a 2-way lookup table like this:
- As mentioned, a two-way lookup table can be implemented via a pair of hash tables: One on characters, one on binary strings
- The encoder will take the one on binary strings and transform it into a trie-like structure known as a **source trie**

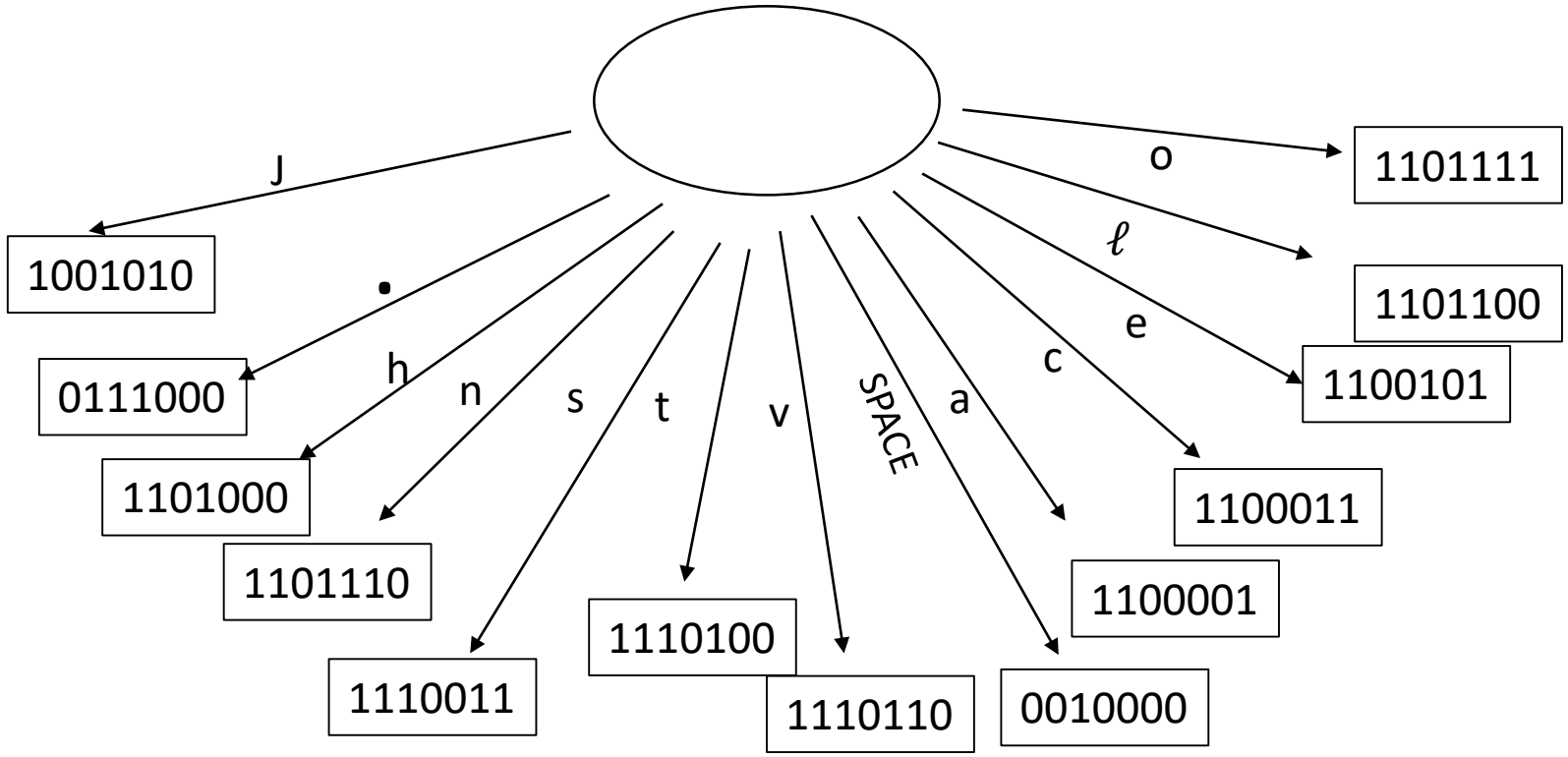
Character	Huffman Binary encoding
h	000000
.	000001
J	000010
n	000011
s	001
t	1000
v	1001
SPACE	0101
a	110
c	111
e	100
ℓ	101
o	11

Source trie corresponding to Huffman encoding



Character	Huffman Binary encoding	ASCII (7-bit) encoding
J	00000	1001010
.	00001	0111000
h	00010	1101000
n	00011	1101110
s	001	1110011
t	01000	1110100
v	01001	1110110
SPACE	0101	0010000
a	0110	1100001
c	0111	1100011
e	100	1100101
ℓ	101	1101100
o	11	1101111

Source trie corresponding to ASCII encoding!

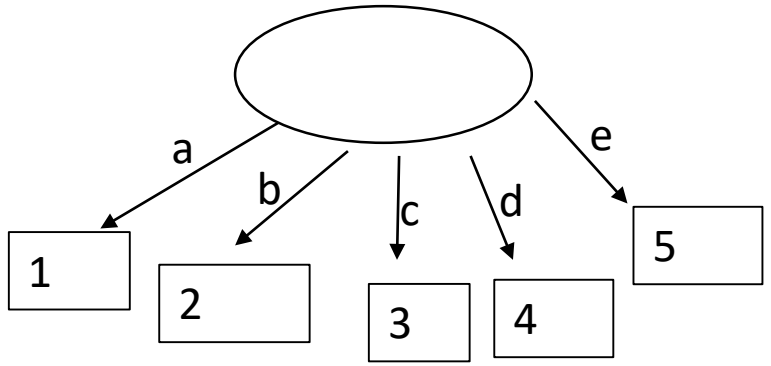


Character	Huffman Binary encoding	ASCII (7-bit) encoding
J	00000	1001010
.	00001	0111000
h	00010	1101000
n	00011	1101110
s	001	1110011
t	01000	1110100
v	01001	1110110
SPACE	0101	0010000
a	0110	1100001
c	0111	1100011
e	100	1100101
ℓ	101	1101100
o	11	1101111

- **Any** character encoding can be fed to LZW (we show ASCII here)!
- We will always be **explicit** about **which one we assume** in exams.

An easier example

- Vocabulary: lowercase english characters
- LZW needs a special character to play the role of the end-of-string character
  - NULL: '\0' sounds pretty appropriate ☺
- Since the encoding can be any one, we will assume integer codewords, which need $\lceil \log_2(n+1) \rceil$ bits to transmit



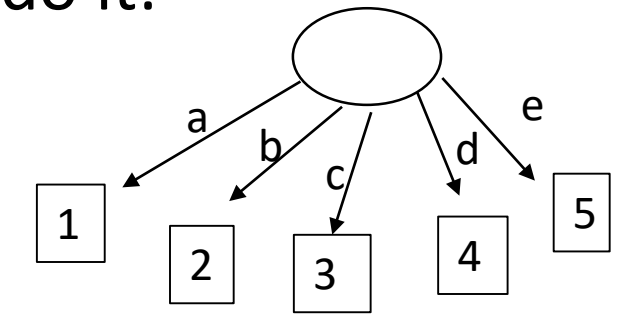
Encoding example

- Suppose that we want to encode the string

  ededadedeedeeeee\0.

- This is how LZW would do it:

MAX CODE = 5



Numbers transmitted:

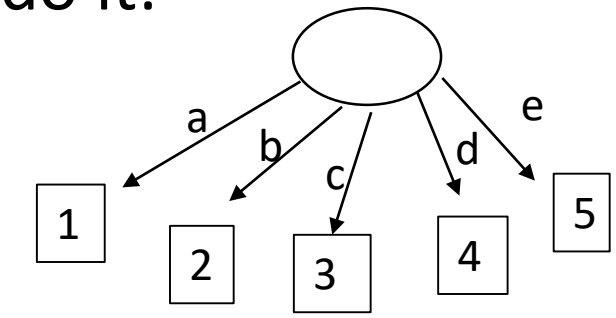
Encoding example

- Suppose that we want to encode the string

ededadedeedeeeee\0.

If I transmit character-by-character character codes, I get 5454145455455555

- This is how LZW would do it:



MAX CODE = 5

Numbers transmitted:

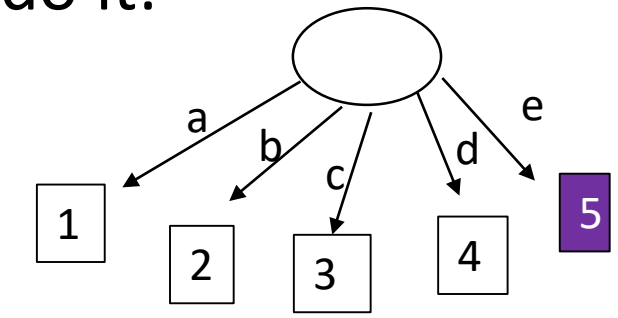
Encoding example

- Suppose that we want to encode the string

  ededadedeedeeeee\0.

- This is how LZW would do it:

MAX CODE = 5



Numbers transmitted: 5

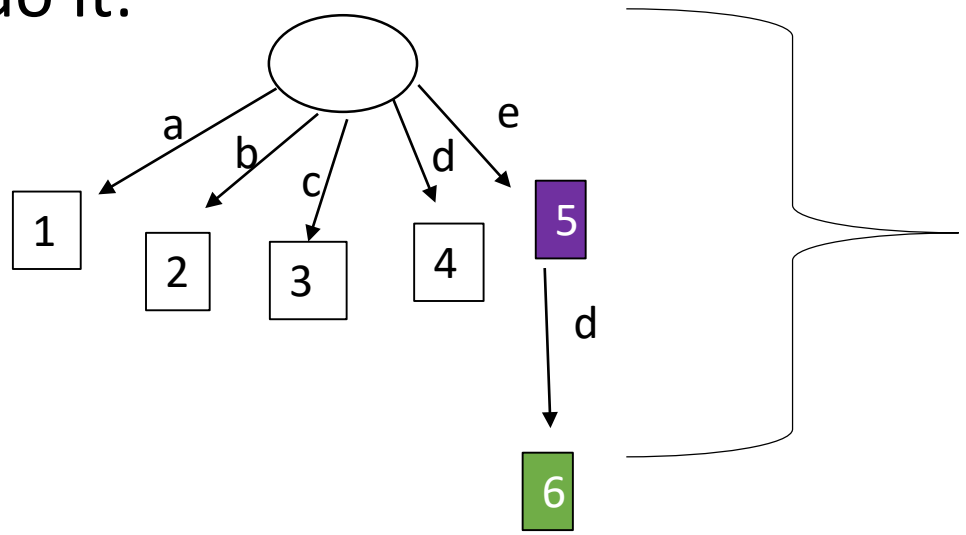
Encoding example

- Suppose that we want to encode the string

  ededadedeedeeeee\0.

- This is how LZW would do it:

MAX CODE = 5



Key step of LZW: we will peek ahead, look at the next character, make a new path that encodes this bigram and append MAX CODE + 1 to it.

Numbers transmitted: 5

Encoding example

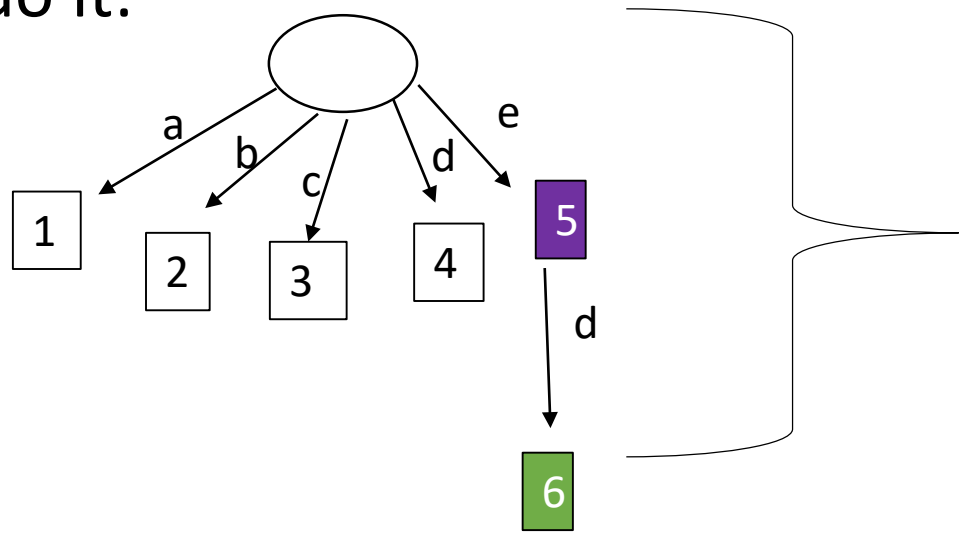
- Suppose that we want to encode the string

ededadedeedeeeee\0.

- This is how LZW would do it:

MAX CODE =6

Let's not forget to increase MAX CODE…



Key step of LZW: we will peek ahead, look at the next character, make a new path that encodes this bigram and append MAX CODE + 1 to it.

Numbers transmitted: 5

Encoding example

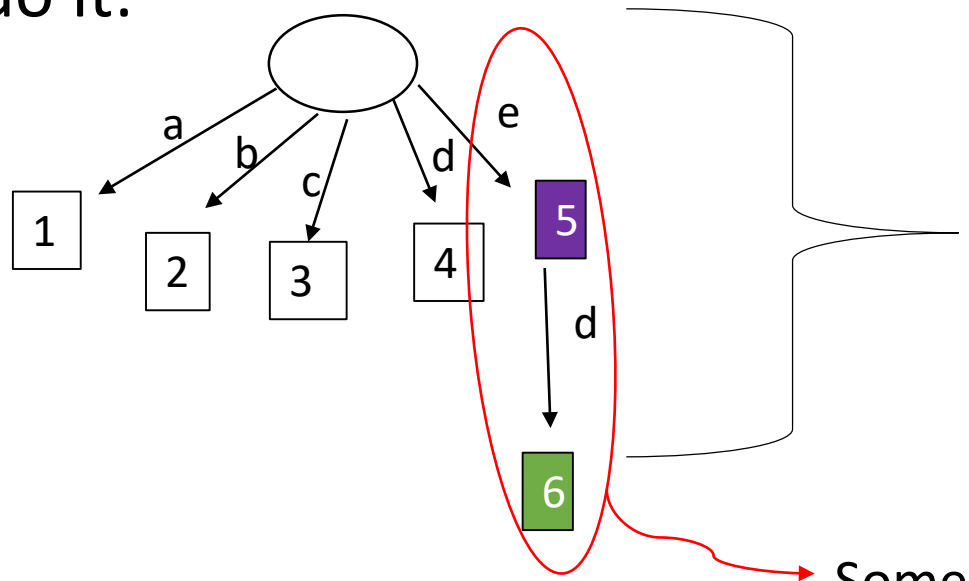
- Suppose that we want to encode the string

ededadedeedeeeee\0.

- This is how LZW would do it:

MAX CODE =6

Let's not forget to increase MAX CODE…



Key step of LZW: we will peek ahead, look at the next character, make a new path that encodes this bigram and append MAX CODE + 1 to it.

Some people call these longer sequences **tokens.**

Numbers transmitted: 5

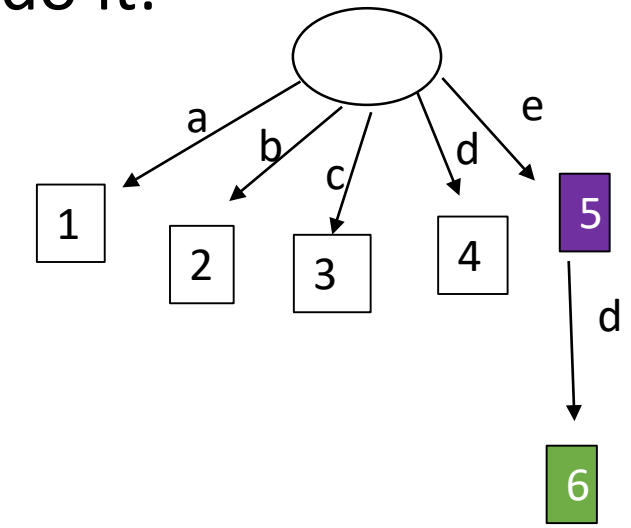
Encoding example

- Suppose that we want to encode the string

  ededadedeedeeeee\0.

- This is how LZW would do it:

MAX CODE =6



Numbers transmitted: 5

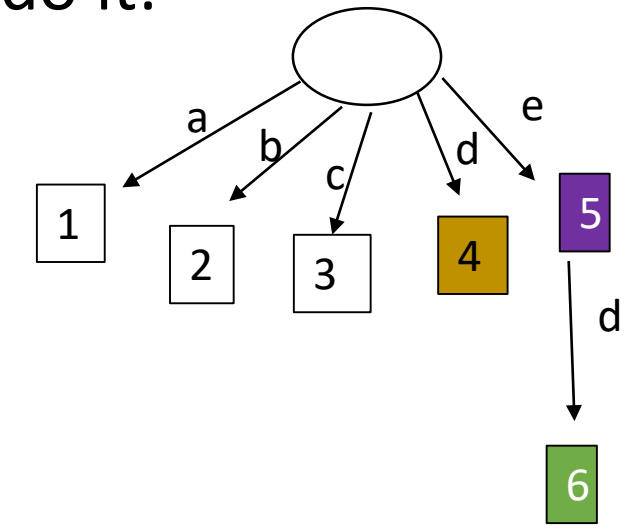
Encoding example

- Suppose that we want to encode the string

  ededadedeedeeeee\0.

- This is how LZW would do it:

MAX CODE =6



Numbers transmitted: 5 4

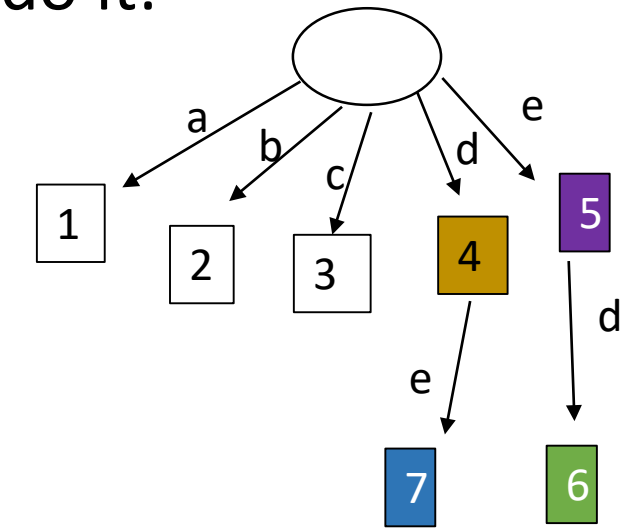
Encoding example

- Suppose that we want to encode the string

  ededadededeeeeee\0.

- This is how LZW would do it:

MAX CODE =7



Peek ahead and give me a cheap encoding for token "de"!

Numbers transmitted: 5 4

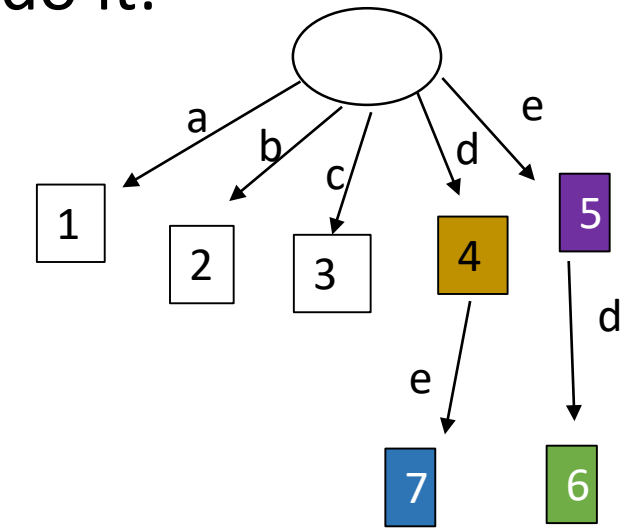
Encoding example

- Suppose that we want to encode the string

  ededadedeedeeeee\0.

- This is how LZW would do it:

MAX CODE =7



Numbers transmitted: 5 4

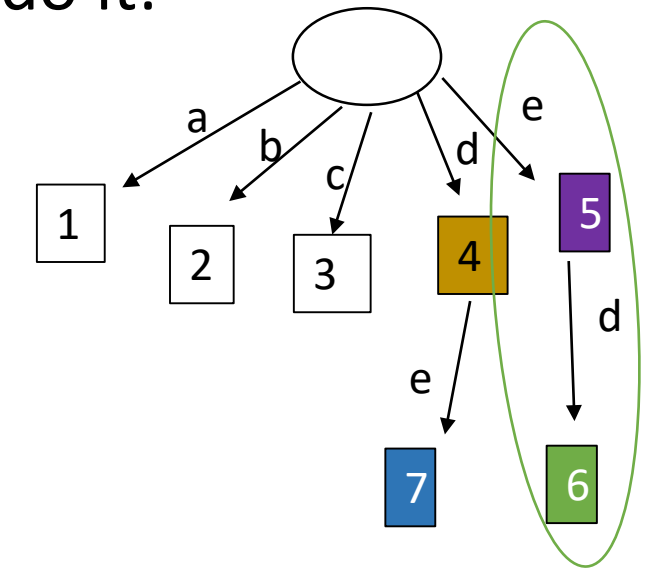
Encoding example

- Suppose that we want to encode the string

  ededadedeedeeeee\0.

- This is how LZW would do it:

MAX CODE =7



Now we can take advantage of the **cheap encoding** for token "ed" which has already been inserted into source trie ☺

Numbers transmitted: 5 4 6

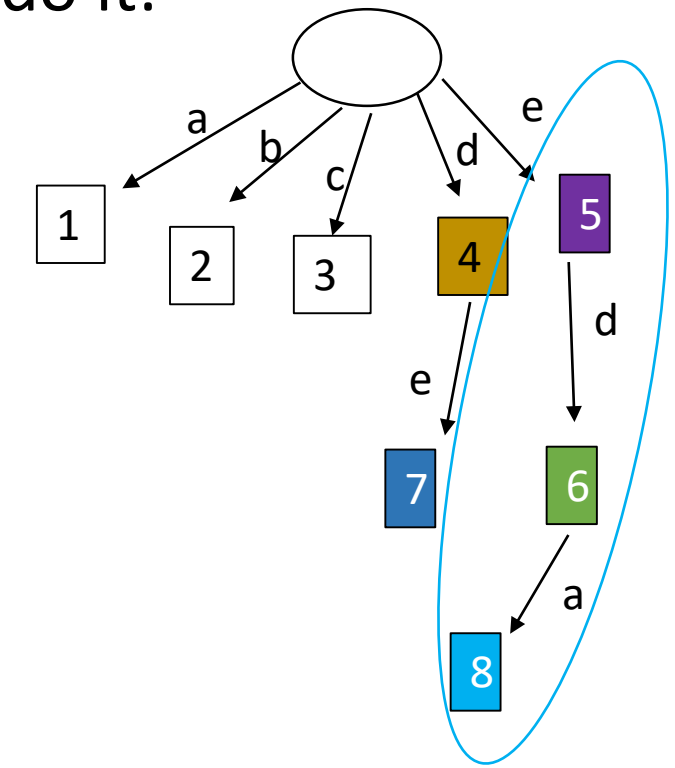
Encoding example

- Suppose that we want to encode the string

  ededadedeedeeeee\0.

- This is how LZW would do it:

MAX CODE =8



Now we can take advantage of the **cheap encoding** for token "ed" which has already been inserted into source trie ☺

But we should not forget to peek ahead, and add a larger token to our trie!

Numbers transmitted: 5 4 6

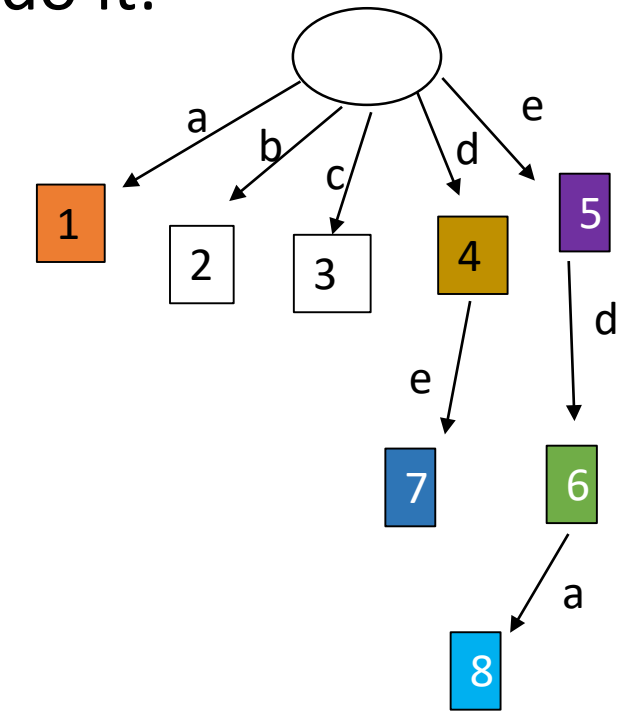
Encoding example

- Suppose that we want to encode the string

  ededadedeedeeeee\0.

- This is how LZW would do it:

MAX CODE =8



Numbers transmitted: 5 4 6 1

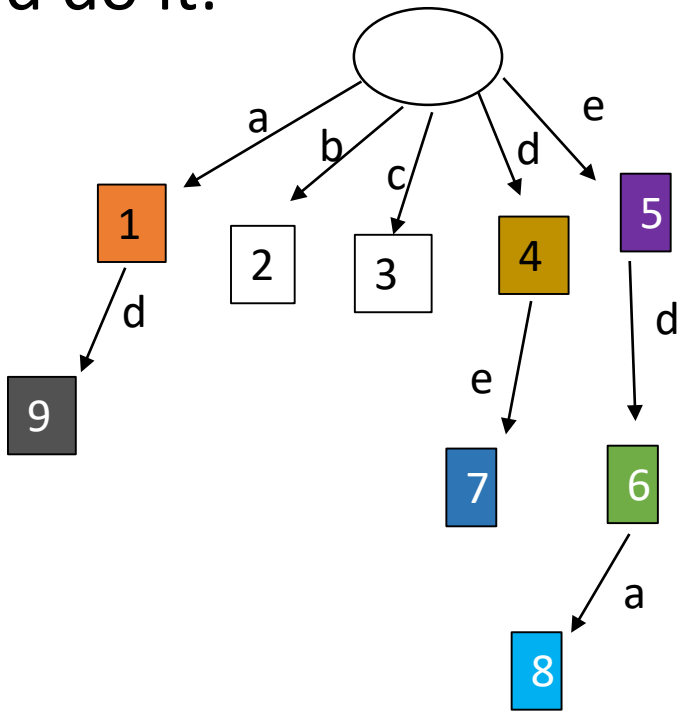
Encoding example

- Suppose that we want to encode the string

  ededadedeedeeeee\0.

- This is how LZW would do it:

MAX CODE =9



Numbers transmitted: 5 4 6 1

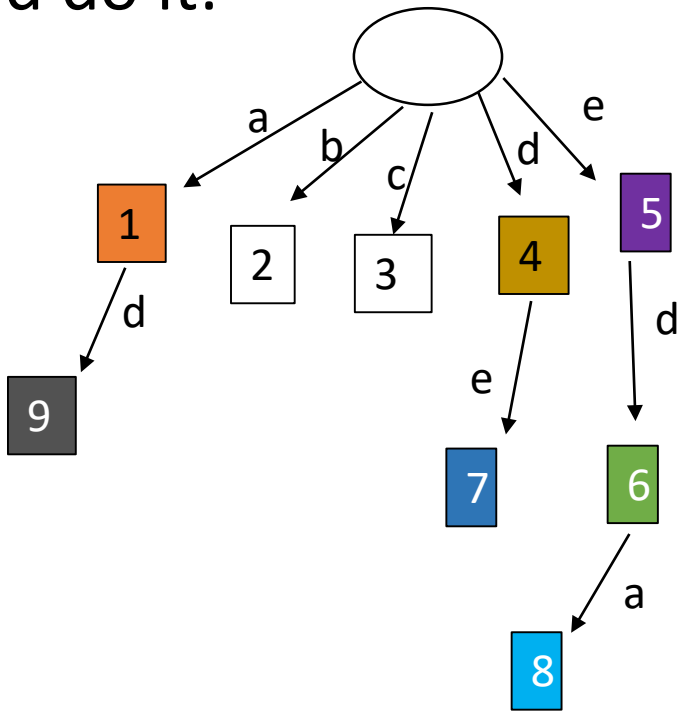
Encoding example

- Suppose that we want to encode the string

  ededadededeeeeee\0.

- This is how LZW would do it:

MAX CODE =9



- **de already in the trie ☺**
- **Transmit its codeword, 7**

Numbers transmitted: 5 4 6 1 7

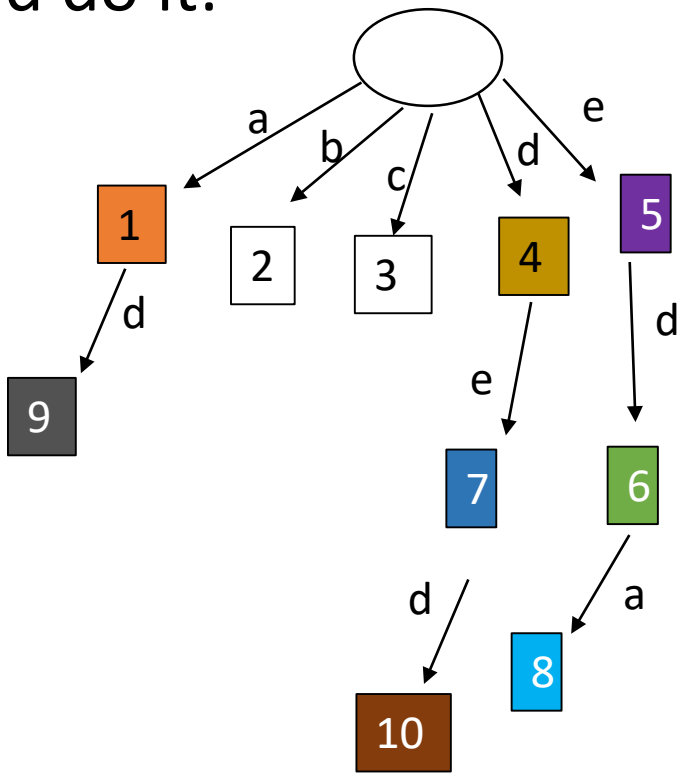
Encoding example

- Suppose that we want to encode the string

  ededadedeedeeeee\0.

- This is how LZW would do it:

MAX CODE =**10**



- **de already in the trie ☺**
- **Transmit its codeword, 7**
- **Peeking ahead** gives us a **new token** with its codeword=max code + 1

Numbers transmitted: 5 4 6 1 7

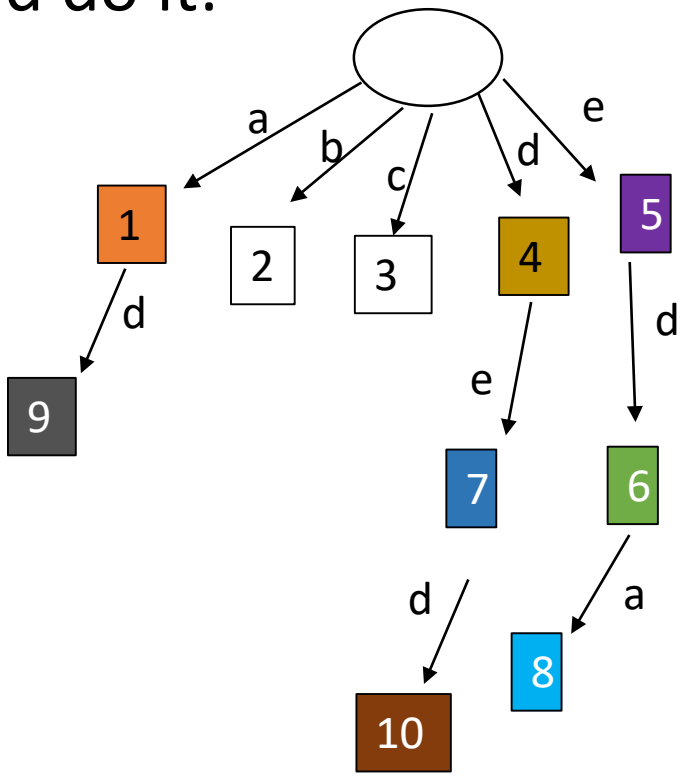
Encoding example

- Suppose that we want to encode the string

  ededadedeedeeeee\0.

- This is how LZW would do it:

MAX CODE =**10**



- **de already in the trie** ☺
- **Transmit its codeword, 7**
- **Peeking ahead** gives us a **new token** with its codeword=max code + 1
- We also update MAX_CODE ☺

Numbers transmitted: 5 4 6 1 7

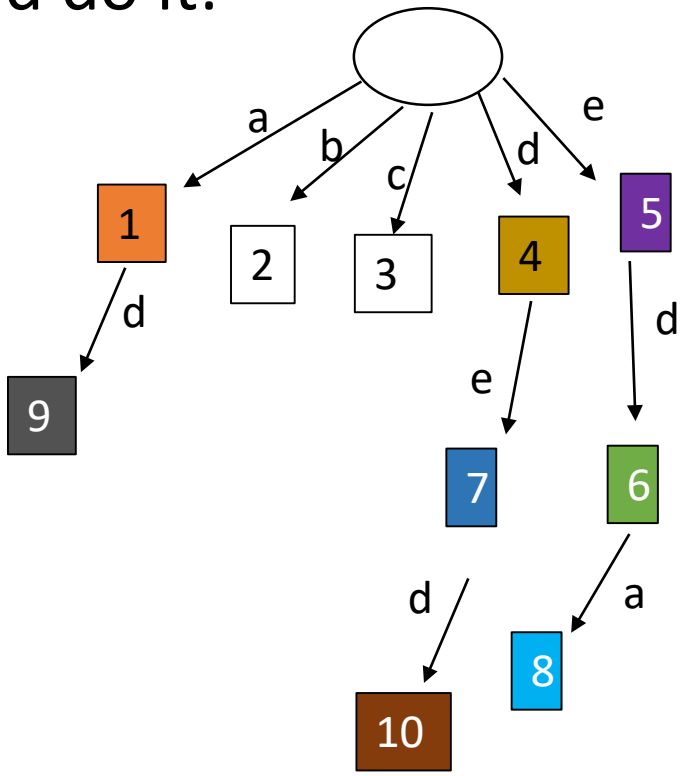
Encoding example

- Suppose that we want to encode the string

  ededadedeedeeeee\0.

- This is how LZW would do it:

MAX CODE =10



- de already in the trie (again!) ☺
- Transmit its codeword, 7

Numbers transmitted: 5 4 6 1 7 7

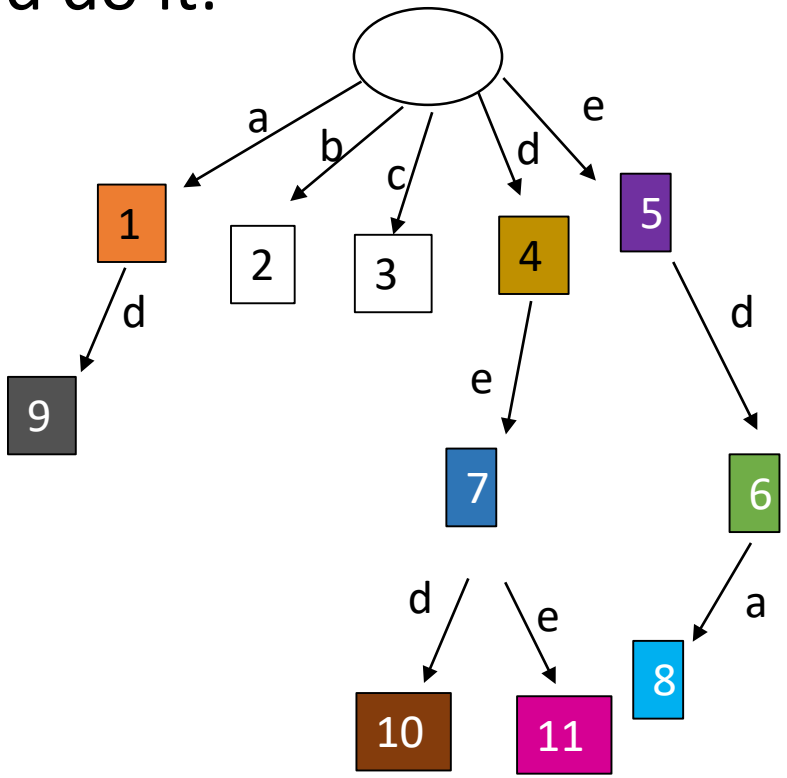
Encoding example

- Suppose that we want to encode the string

  ededadedeedeeeee\0.

- This is how LZW would do it:

MAX CODE =10



- de already in the trie (again!) ☺
- Transmit its codeword, 7
- Peek ahead, notice e and **insert token dee** in the trie, with code MAX_CODE + 1

Numbers transmitted: 5 4 6 1 7 7

Encoding example

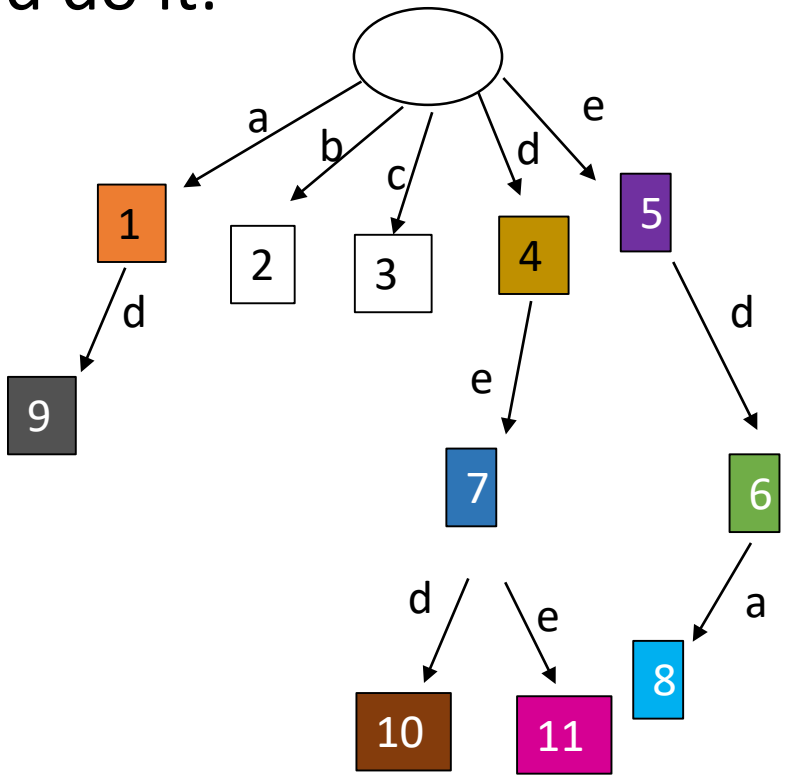
- Suppose that we want to encode the string

  ededadedeedeeeee\0.

- This is how LZW would do it:

MAX CODE =11

Update **MAX CODE** appropriately.



- de already in the trie (again!) ☺
- Transmit its codeword, 7
- Peek ahead, notice e and **insert token dee** in the trie, with code MAX_CODE + 1
- Update **MAX CODE** appropriately.

Numbers transmitted: 5 4 6 1 7 7

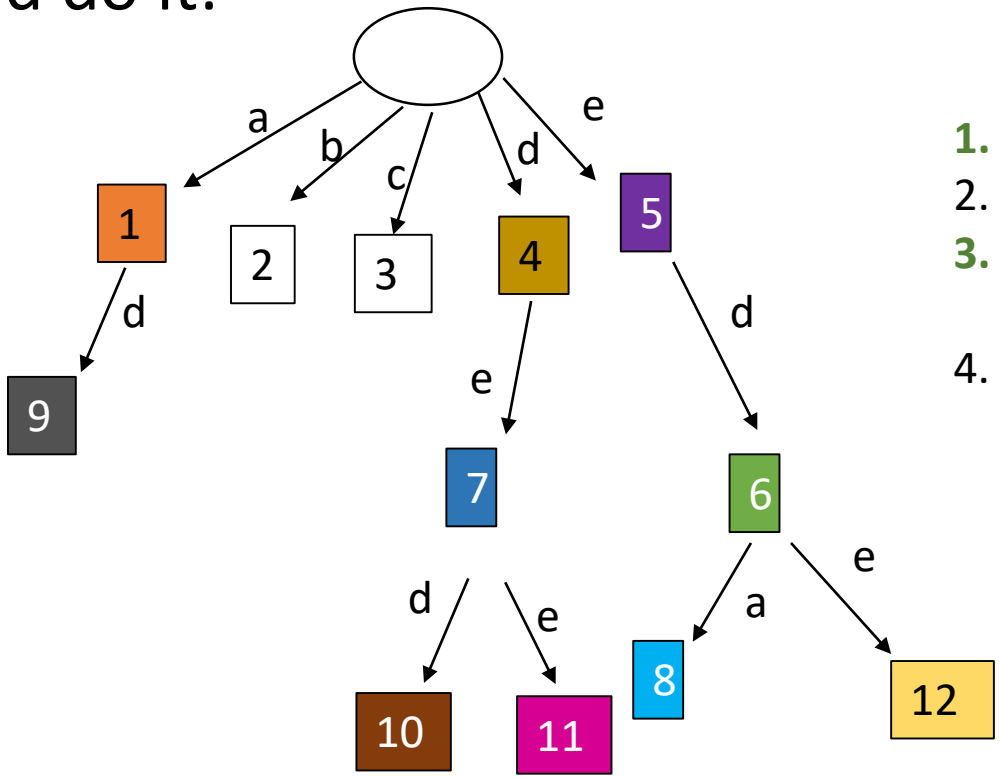
Encoding example

- Suppose that we want to encode the string

  ededadededeeeeee\0.

- This is how LZW would do it:

MAX CODE =12



1. **ed** in the trie
2. Transmit its codeword, **6**
3. **ede** is added as a trie token with codeword MAX CODE + 1
4. MAX CODE is updated

Numbers transmitted: 5 4 6 1 7 7 6

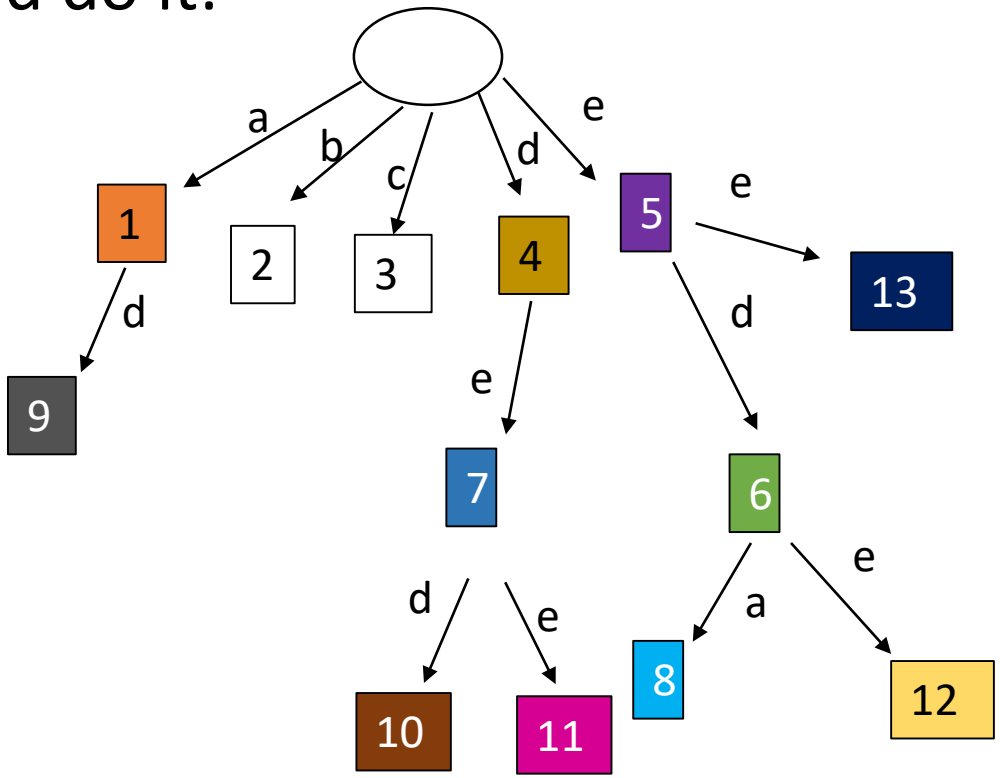
Encoding example

- Suppose that we want to encode the string

  ededadededeeeeee\0.

- This is how LZW would do it:

MAX CODE =13



1. **e** in the trie
2. Transmit its codeword, **5**
3. **ee** is added as a trie token with codeword MAX CODE + 1
4. MAX CODE is updated

Numbers transmitted: 5 4 6 1 7 7 6 5

Encoding example

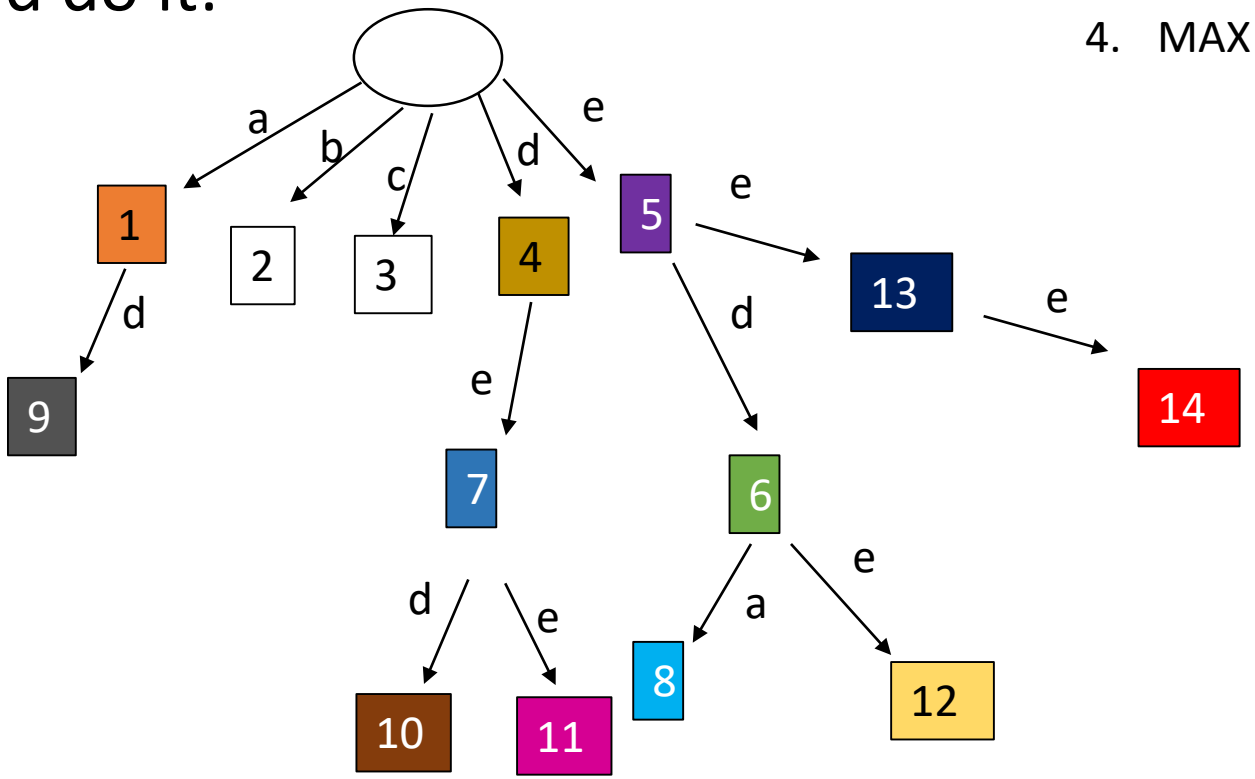
- Suppose that we want to encode the string

  ededadededeeeeee\0.

- This is how LZW would do it:

1. **ee** in the trie
2. Transmit its codeword, **13**
3. **eee** is added as a trie token with codeword MAX CODE + 1
4. MAX CODE is updated

MAX CODE =14



Numbers transmitted: 5 4 6 1 7 7 6 5 **13**

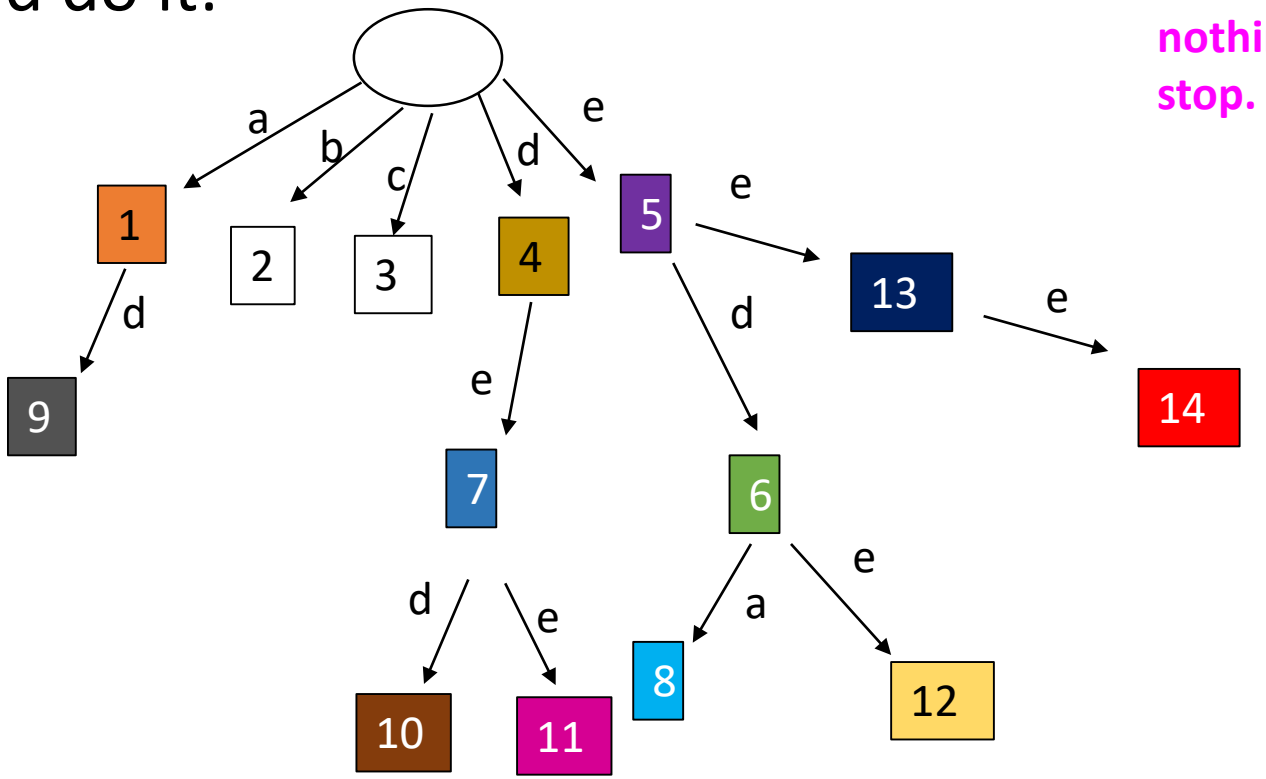
Encoding example

- Suppose that we want to encode the string

  ededadededeeeeee\0.

- This is how LZW would do it:

MAX CODE =14



1. **ee** in the trie
2. Transmit its codeword, **13**
3. **If we look ahead, we see the end-of-string-character, so nothing more to add; we can stop.**

Numbers transmitted: 5 4 6 1 7 7 6 5 **13 13**

Encoding example

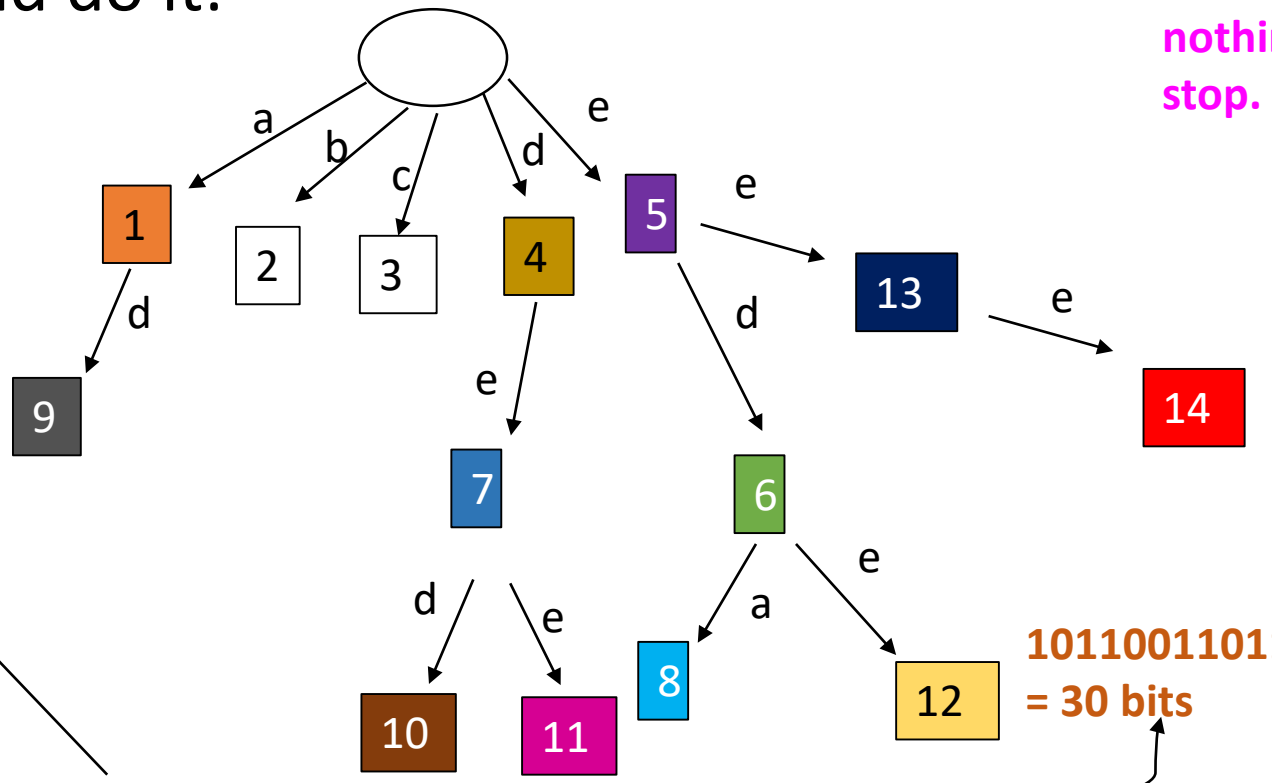
- Suppose that we want to encode the string

  ededadededeeeeee\0.

- This is how LZW would do it:

1. **ee** in the trie
2. Transmit its codeword, **13**
3. **If we look ahead, we see the end-of-string-character, so nothing more to add; we can stop.**

MAX CODE =14



Compare to the **character-by-character sending** from before:

5454145455455555

10110010110011001011001011011001010110101101 = **49 bits**

10110011011111111110101110111011101 = 30 bits

Numbers transmitted: 5 4 6 1 7 7 6 5 **13 13**

Characteristics of LZW encoding process

- During encoding, we **greedily** enlarged the trie after every transmission, since it looked like a good idea at the time!
- It turns out that it **is** a good idea, even if some tokens **are never actually used** (e.g the token “ad” was stored in the trie, but never used later on).

Characteristics of LZW encoding process

- During the encoding of the string you might have noticed that we partitioned the string into code-emitting tokens:

ededadededeeeeee\0

5 4 6 1 7 7 6 5 **13 13**

- This process is known as tokenization: Before transmitted, the string will have to be tokenized into tokens which have a 1-1 correspondence with a code.

Characteristics of LZW encoding process

- During the encoding of the string you might have noticed that we partitioned the string into code-emitting tokens:

e|d|ed|a|de|de|ed|e|ee|ee

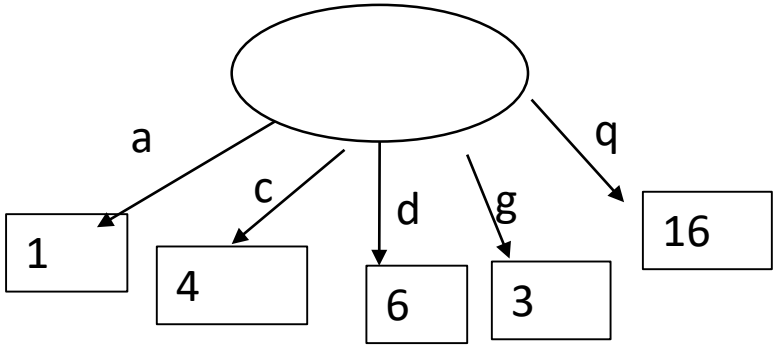
5| 4| 6| 1| 7| 7 |6 |5| 13| 13

Advice: Since you don't (necessarily) have access to multiple colored pens in exams, delineate the tokens with **vertical bars!**

- This process is known as tokenization: Before transmitted, the string will have to be tokenized into tokens which have a 1-1 correspondence with a code.

Let’s work on this!

- Run LZW encoding, using the following source trie to encode **acacdgqgqqgqqqq\0**



LZW decoding

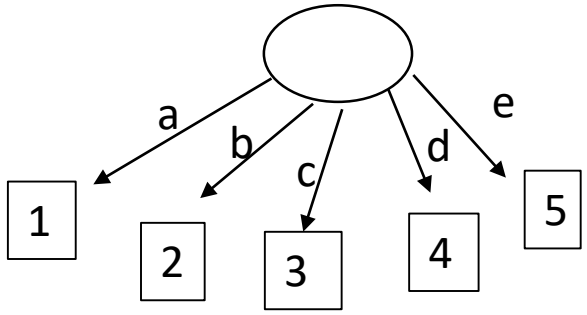
- The big question now is: Given a stream of integers, can we flawlessly *(losslessly)* recover the **original** character sequence?
  - This is particularly important given that integer codewords don't satisfy the prefix property like Huffman tries do!
    - E.g, $1_{(2)} = 1, 3_{(2)} = 11$, and the codeword for 1 is a prefix of that of 3 ☹

LZW decoding

- The big question now is: Given a stream of integers, can we flawlessly *(losslessly)* recover the **original** character sequence?
  - This is particularly important given that integer codewords don't satisfy the prefix property like Huffman tries do!
    - E.g, $1_{(2)} = 1, 3_{(2)} = 11$, and the codeword for 1 is a prefix of that of 3 ☹
- The answer is **yes, as long as the encoder and decoder share the same initial source trie**
  - That's not half bad! ☺

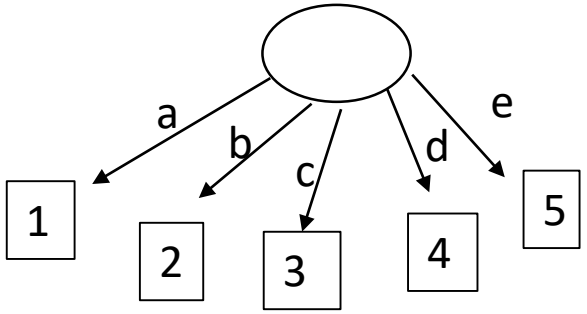
Decoding Example

- Assume I have the following source trie:



- And I receive 5| 4| 6| 1| 7| 7 |6 |5| 13| 13
- Our quest is to decode this tokenized sequence of integers into the precise original character sequence.
  - Remember: lossless compression!

Decoding Example

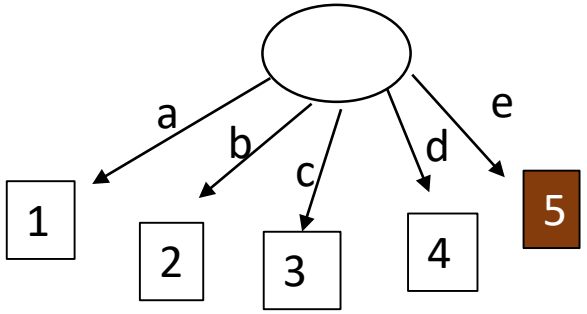


Decoder input: 5| 4| 6| 1| 7| 7 |6 |5| 13| 13

Decoder output:

Decoding Example

- Tokenized codeword ‘5’ corresponds to character ‘e’.
- ‘e’ is **in** the source trie!
- Clearly, this means that ‘e’ is exactly the character sent first.
- Put ‘e’ in output buffer!

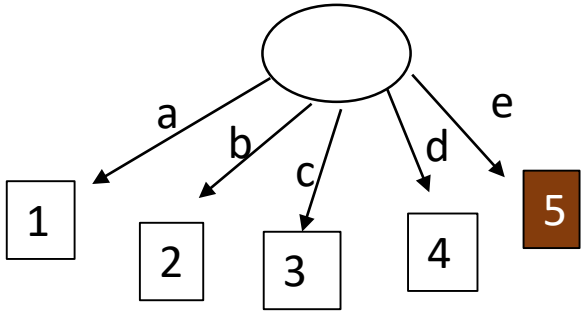


Decoder input: 5| 4| 6| 1| 7| 7 |6 |5| 13| 13

Decoder output: e |

Decoding Example

- Tokenized codeword ‘4’ corresponds to character ‘d’.
- So I can output ‘d’ immediately

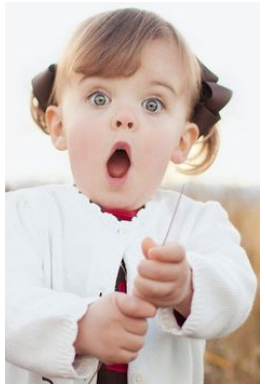


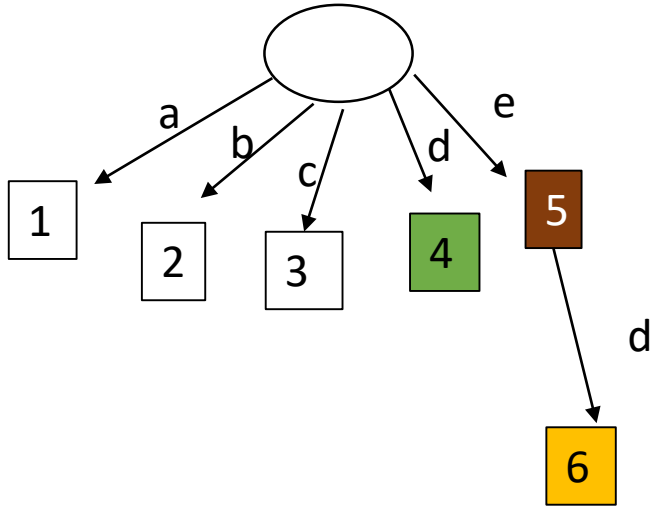
Decoder input: 5| 4| 6| 1| 7| 7 |6 |5| 13| 13

Decoder output: e | d

Decoding Example

- Tokenized codeword ‘4’ corresponds to character ‘d’.
- So I can output ‘d’ immediately
- **Key step:** After writing ‘d’ to the output buffer, **I will add ‘ed’ to my copy of the trie,** since I know that this is LZW, **and there’s no other way the encoder would operate!**



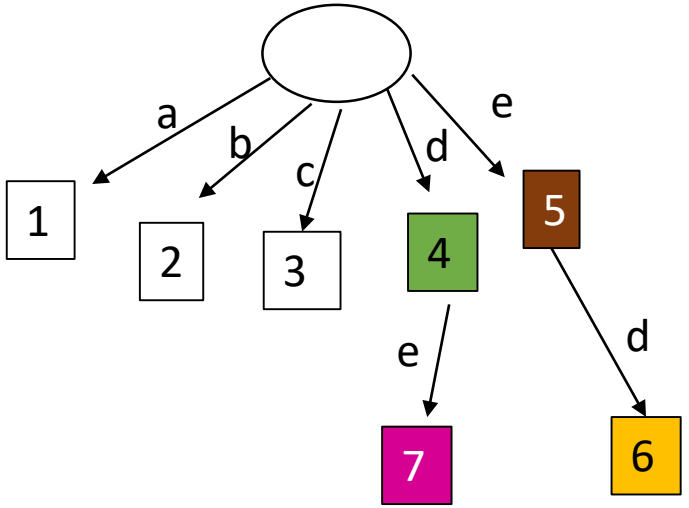


Decoder input: 5| 4| 6| 1| 7| 7 |6 |5| 13| 13

Decoder output: e | d

Decoding Example

- Codeword ‘6’ corresponds to token ‘ed’ which I just added to my trie!
- Output it.
- **Add “de” to the trie (last token transmitted + first character of current token)**

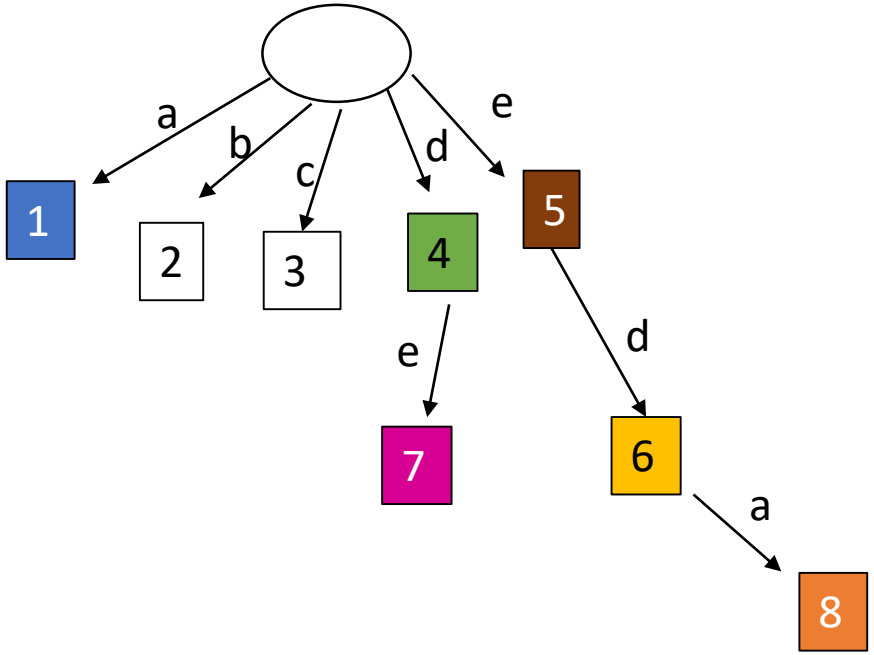


Decoder input: 5| 4| 6| 1| 7| 7 |6 |5| 13| 13

Decoder output: e | d | ed |

Decoding Example

- Codeword ‘1’ corresponds to stored token ‘a’
- Output it.
- **Add “eda” to the trie (last token transmitted + first character of current token)**

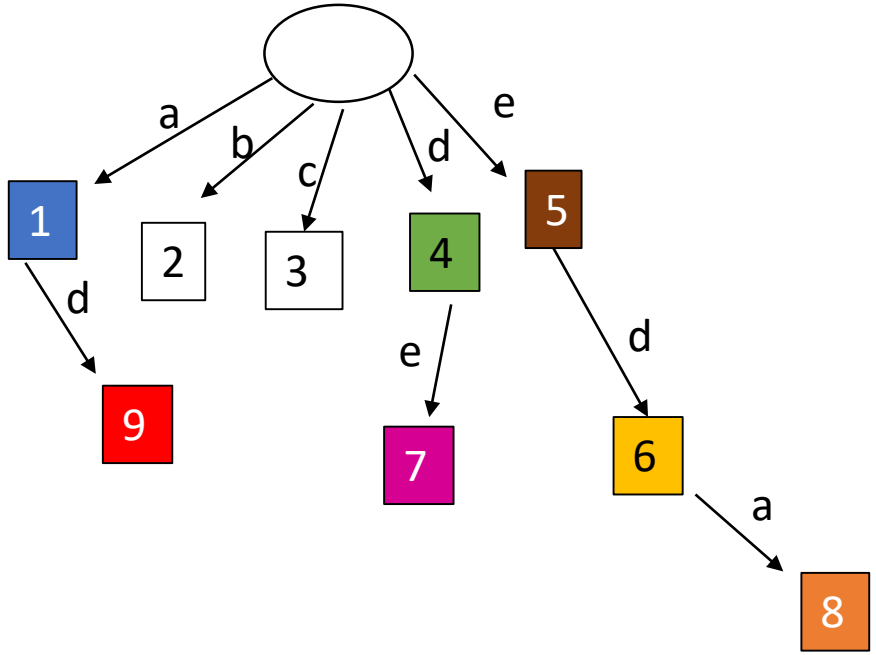


Decoder input: 5| 4| 6| 1| 7| 7 |6 |5| 13| 13

Decoder output: e | d | ed | a |

Decoding Example

- Codeword ‘7’ corresponds to (recently!) stored token “de”
- Output it.
- Add “ad” to the trie (last token transmitted + first character of current token)

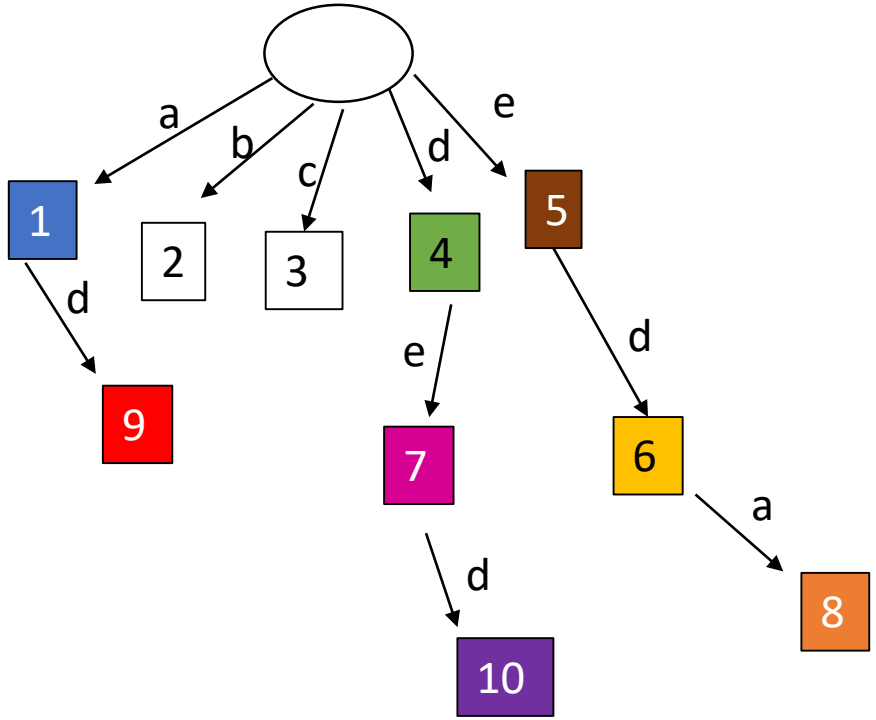


Decoder input: 5| 4| 6| 1| 7| 7 |6 |5| 13| 13

Decoder output: e | d | ed | a | de |

Decoding Example

- Codeword ‘7’ corresponds to (recently!) stored token “de” again!
- Output it.
- Add **“ded”** to the trie (last token transmitted + first character of current token)

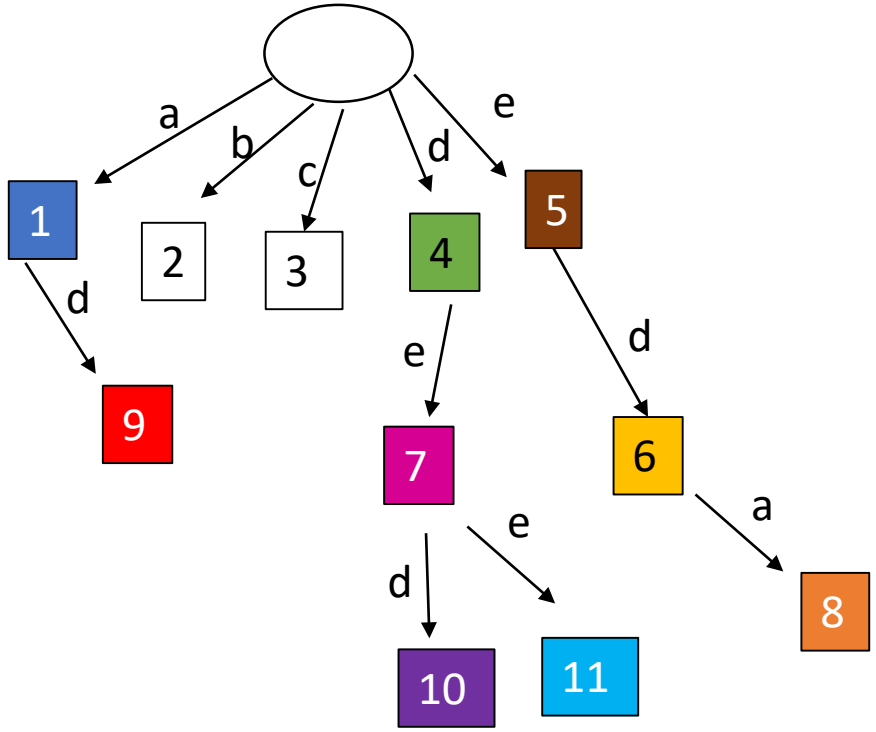


Decoder input: 5| 4| 6| 1| 7| 7 |6 |5| 13| 13

Decoder output: e | d | ed | a | de | de

Decoding Example

- Codeword ‘6’ corresponds to (recently!) stored token “ed”
- Output it.
- Add **“dee”** to the trie (last token transmitted + first character of current token)

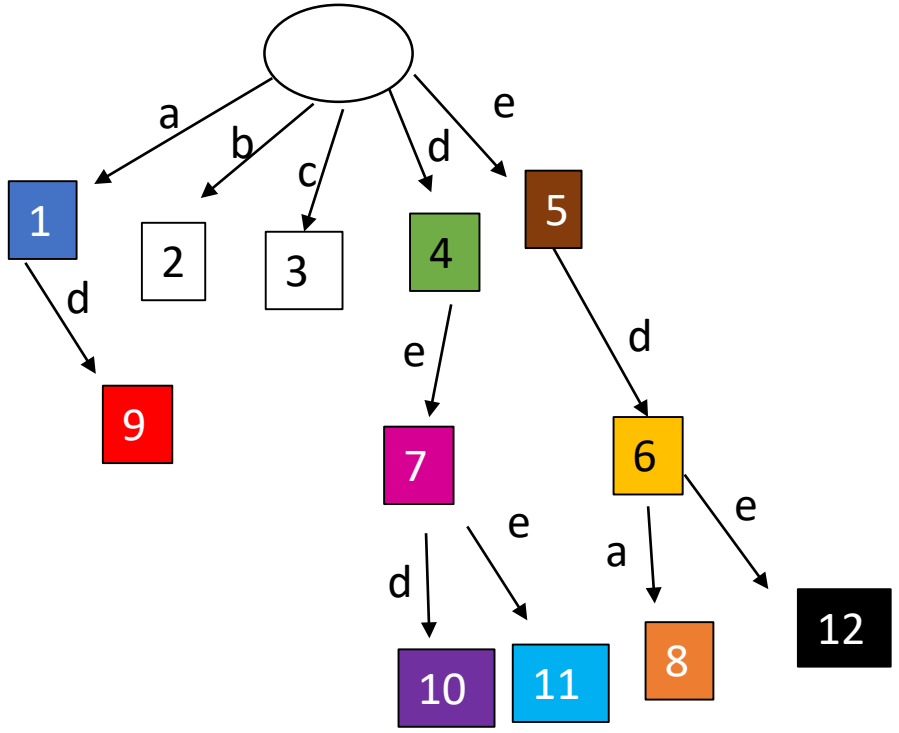


Decoder input: 5| 4| 6| 1| 7| 7 |6 |5| 13| 13

Decoder output: e | d | ed | a | de | de | ed

Decoding Example

- Codeword ‘5’ corresponds to stored token “e”
- Output it.
- Add **“ede”** to the trie (last token transmitted + first character of current token)

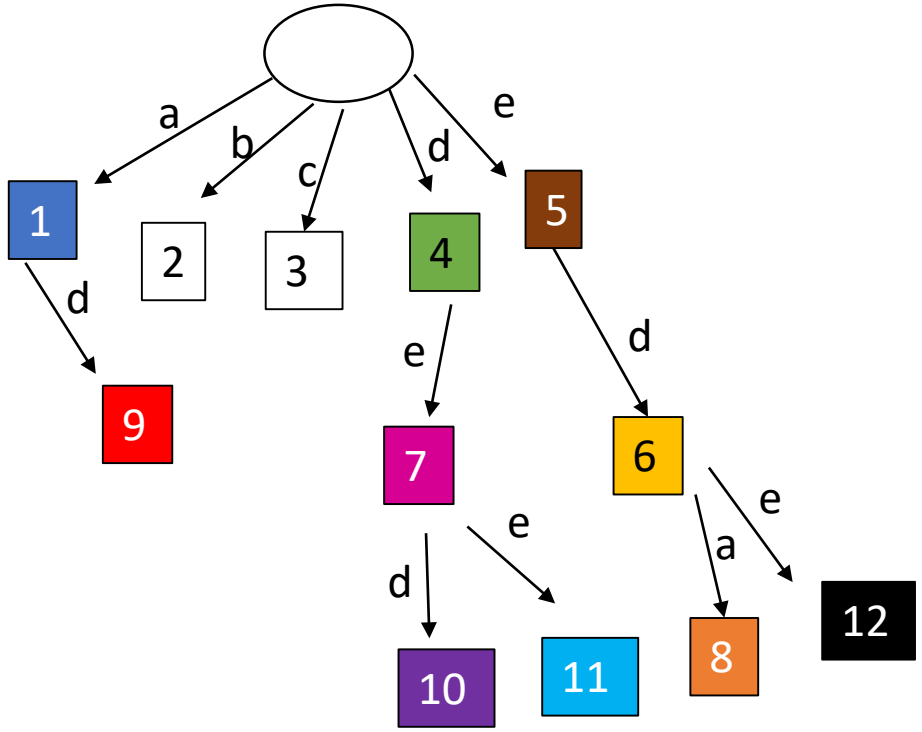


Decoder input: 5| 4| 6| 1| 7| 7 |6 | 5 | 13| 13

Decoder output: e | d | ed | a | de | de | ed | e

Decoding Example

- Codeword ‘13’ corresponds to…
  wait, what?

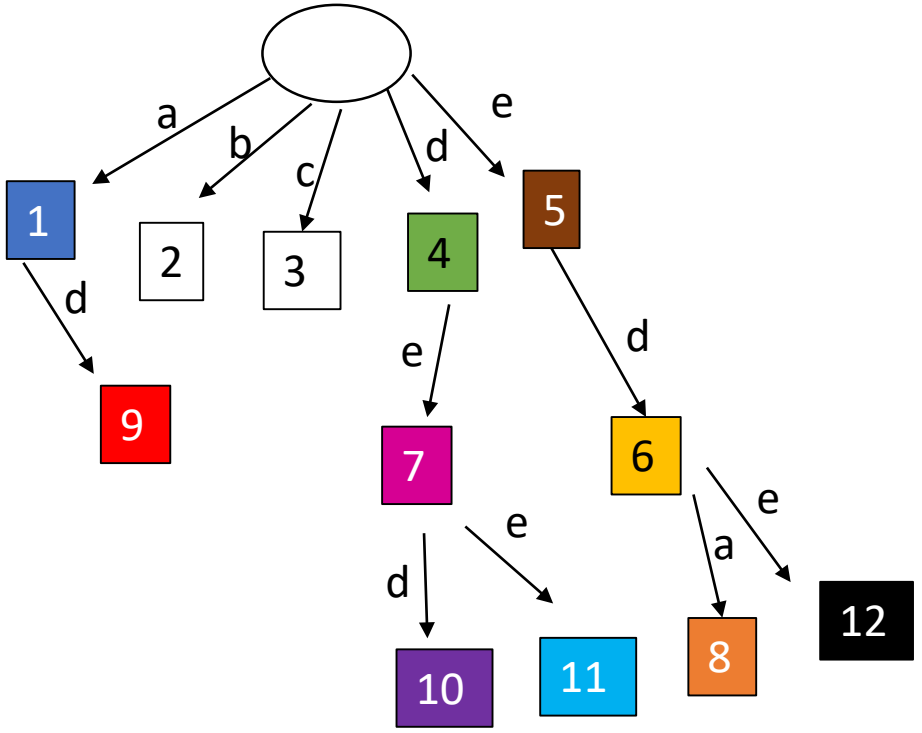


Decoder input: 5| 4| 6| 1| 7| 7 |6 | 5 | 13| 13

Decoder output: e | d | ed | a | de | de | ed | e

Decoding Example

- Codeword ‘13’ corresponds to… **wait, what**?
- **‘13’ is not in the trie!**



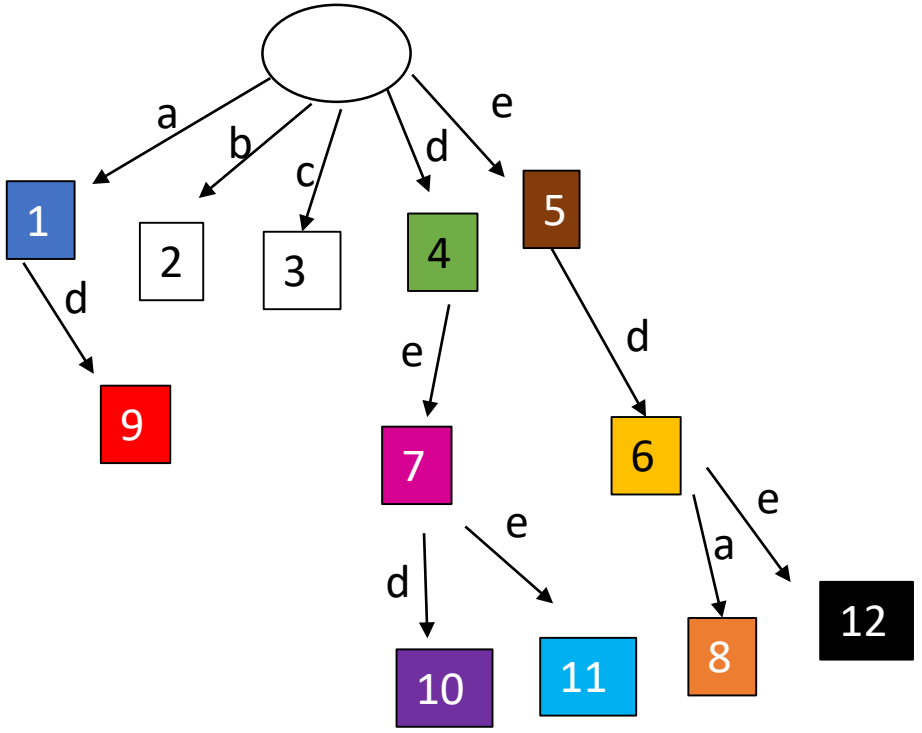
Decoder input: 5| 4| 6| 1| 7| 7 |6 | 5 | 13| 13

Decoder output: e | d | ed | a | de | de | ed | e

Decoding Example

- Codeword ‘13’ corresponds to… **wait, what**?
- **‘13’ is not in the trie!**

Which token do you guys think should 13 be decoded to?



ede

ee

deee

Something Else (what?)

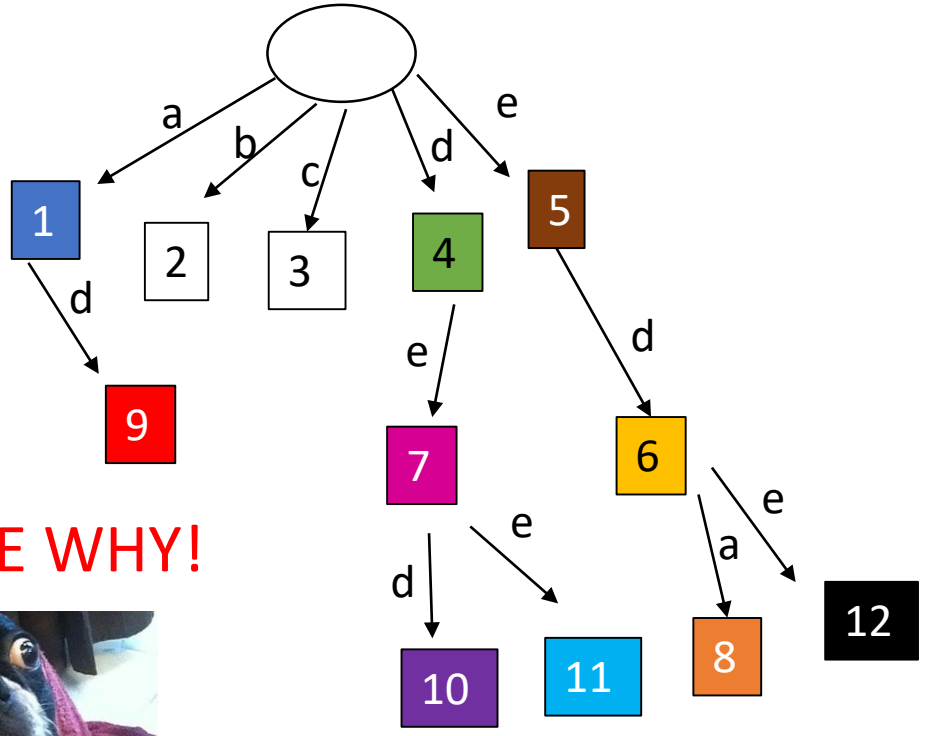
Decoder input: 5| 4| 6| 1| 7| 7 |6 | 5 | 13| 13

Decoder output: e | d | ed | a | de | de | ed | e

Decoding Example

- Codeword ‘13’ corresponds to… **wait, what**?
- **‘13’ is not in the trie!**

Which token do you guys think should 13 be decoded to?



ede

ee

deee

Something Else (what?)

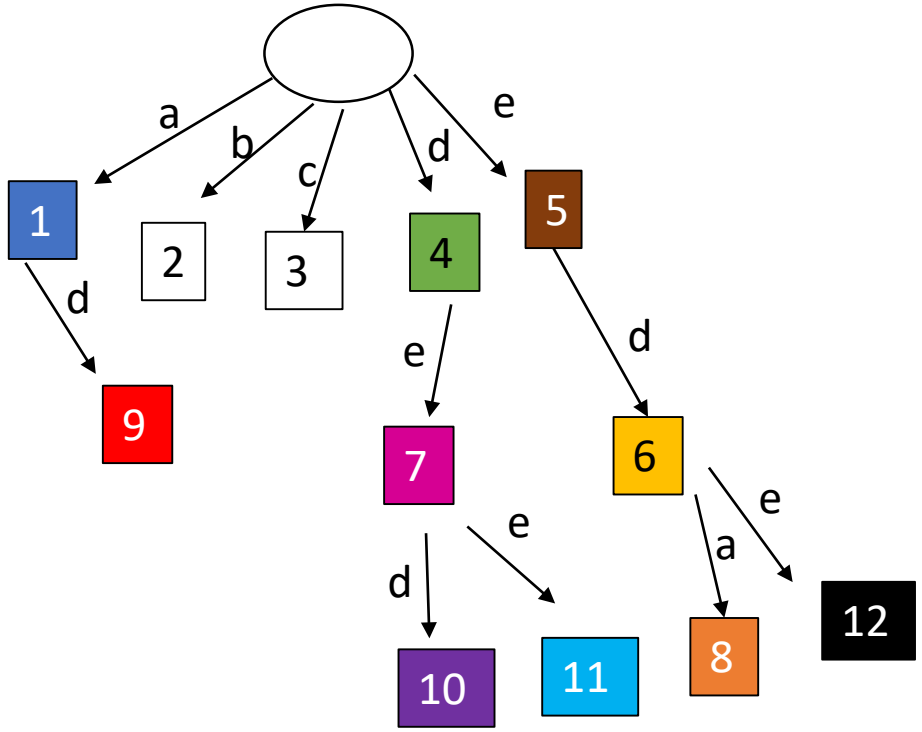
LET’S SEE WHY!

Decoder input: 5| 4| 6| 1| 7| 7 |6 | 5 | 13| 13

Decoder output: e | d | ed | a | de | de | ed | e

Decoding Example

- **CRUCIAL STEP**: Since codeword 13 is not in the trie, we know that the encoder, before transmitting it, **had just created it.**



Decoder input: 5| 4| 6| 1| 7| 7 |6 | 5 | 13| 13

Decoder output: e | d | ed | a | de | de | ed | e

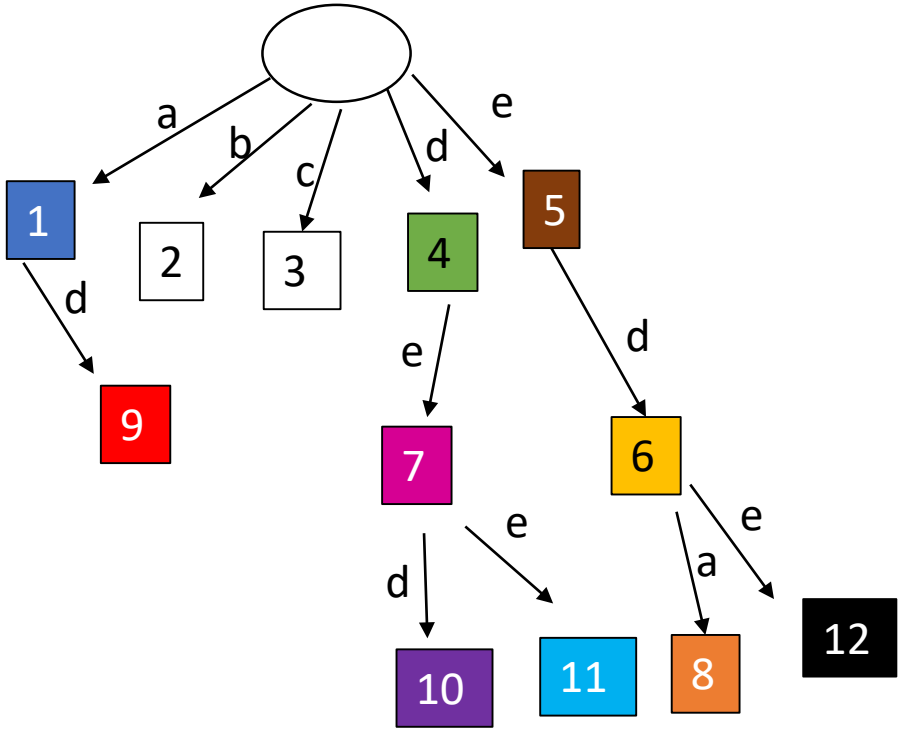
Decoding Example

- **CRUCIAL STEP**: Since codeword 13 is not in the trie, we know that the encoder, before transmitting it, **had just created it.**
- **What should the minimum length of the corresponding token be?**

1

2

Something Else (what?)

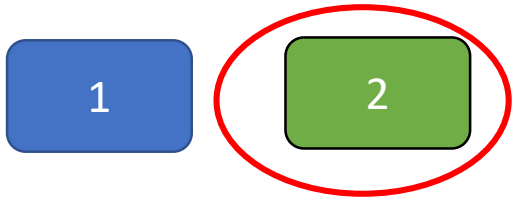


Decoder input: 5| 4| 6| 1| 7| 7 |6 | 5 | 13| 13

Decoder output: e | d | ed | a | de | de | ed | e

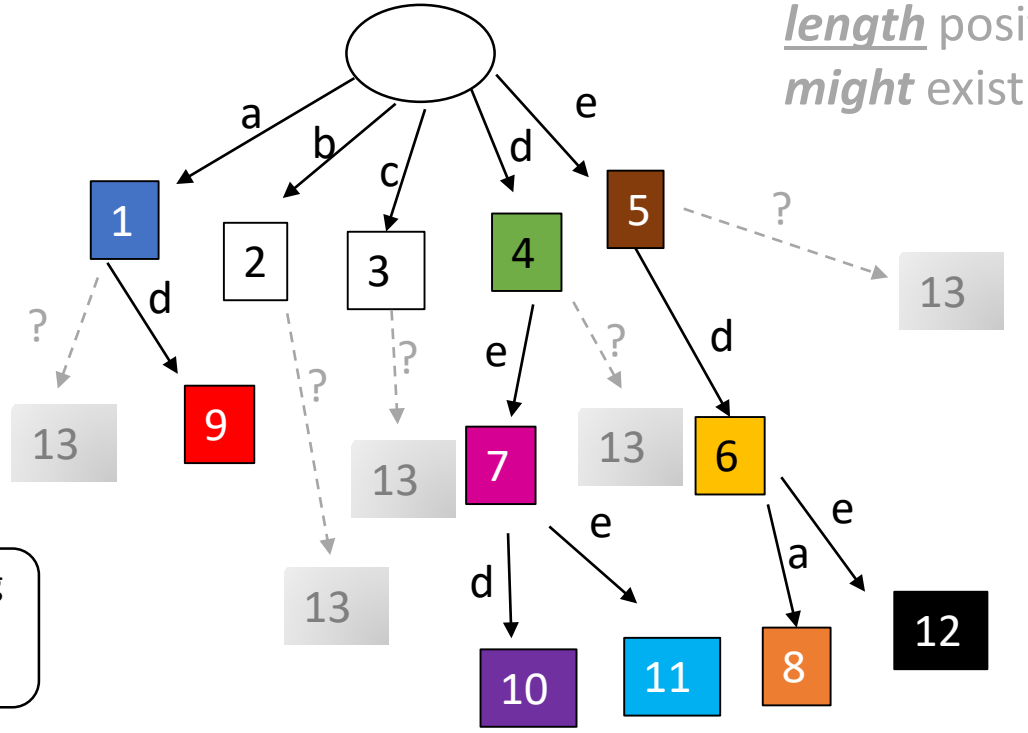
Decoding Example

- **CRUCIAL STEP**: Since codeword 13 is not in the trie, we know that the encoder, before transmitting it, **had just created it.**
- **What should the minimum length of the corresponding token be?**



Something Else (what?)

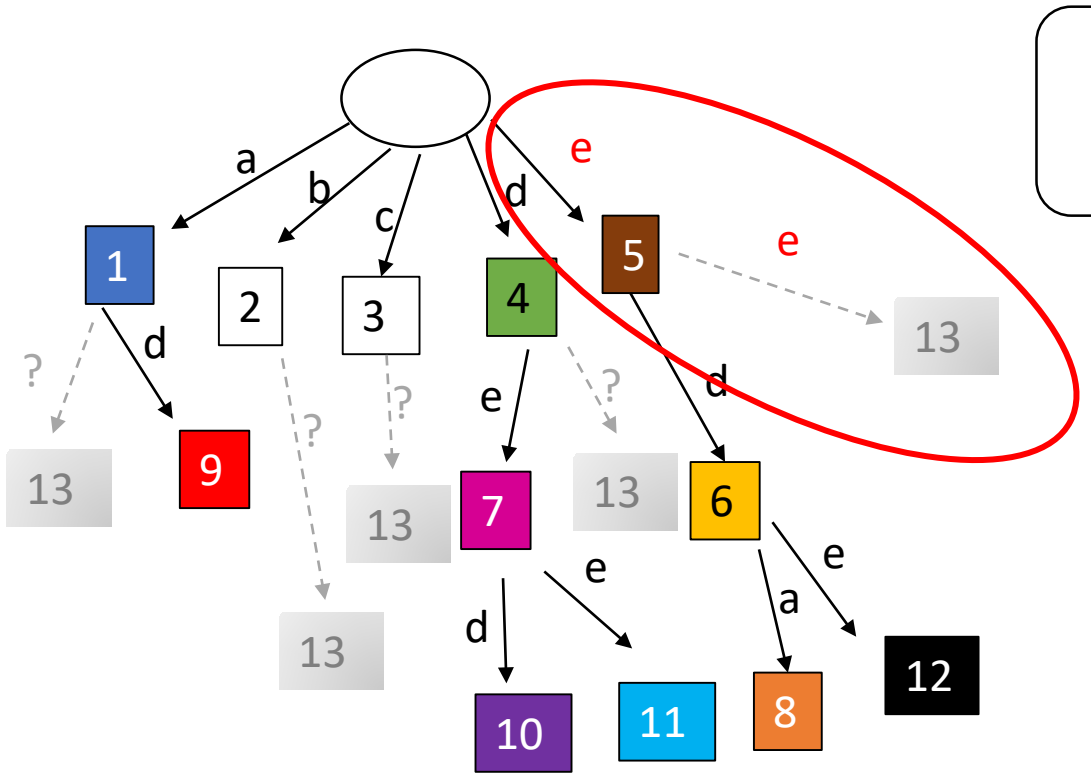
These are my ***minimal length*** positions where 13 ***might*** exist….

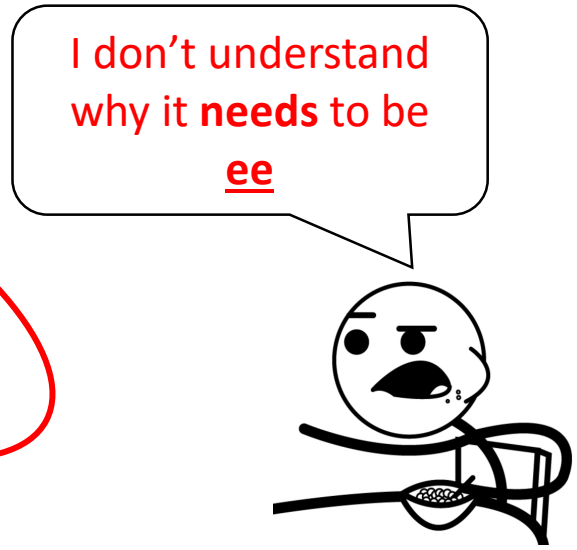


Decoder input: 5| 4| 6| 1| 7| 7 |6 | 5 | 13| 13

Decoder output: e | d | ed | a | de | de | ed | e

Decoding Example

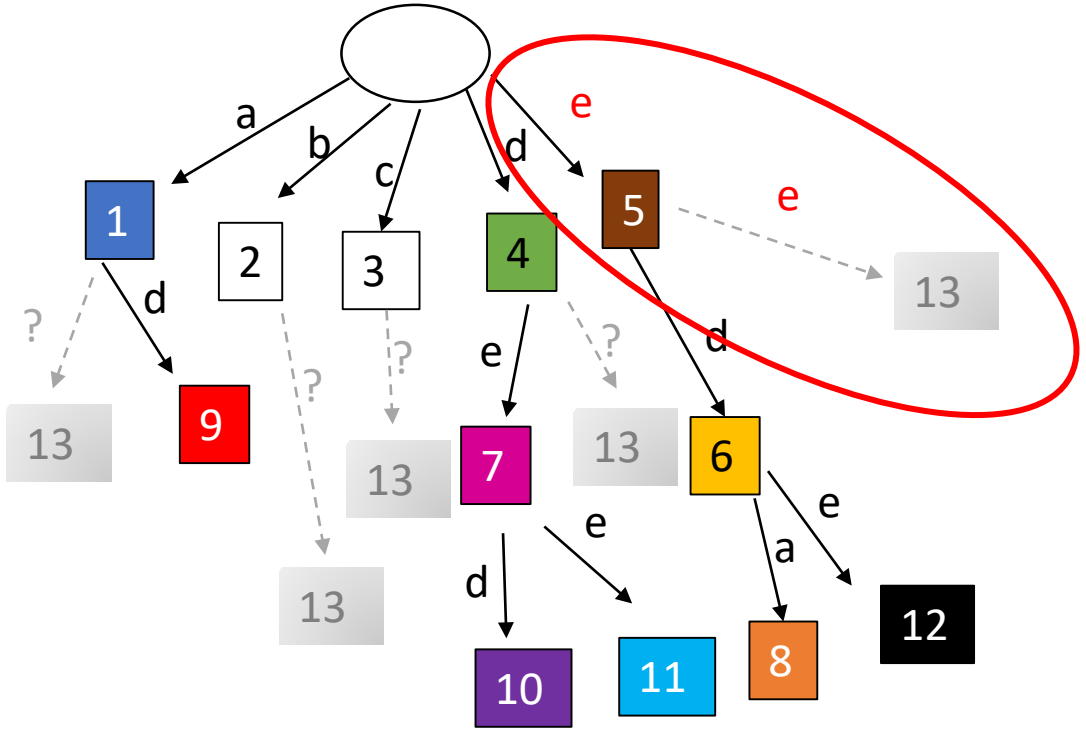




Decoder input: 5| 4| 6| 1| 7| 7 |6 | 5 | 13| 13

Decoder output: e | d | ed | a | de | de | ed | e

Decoding Example



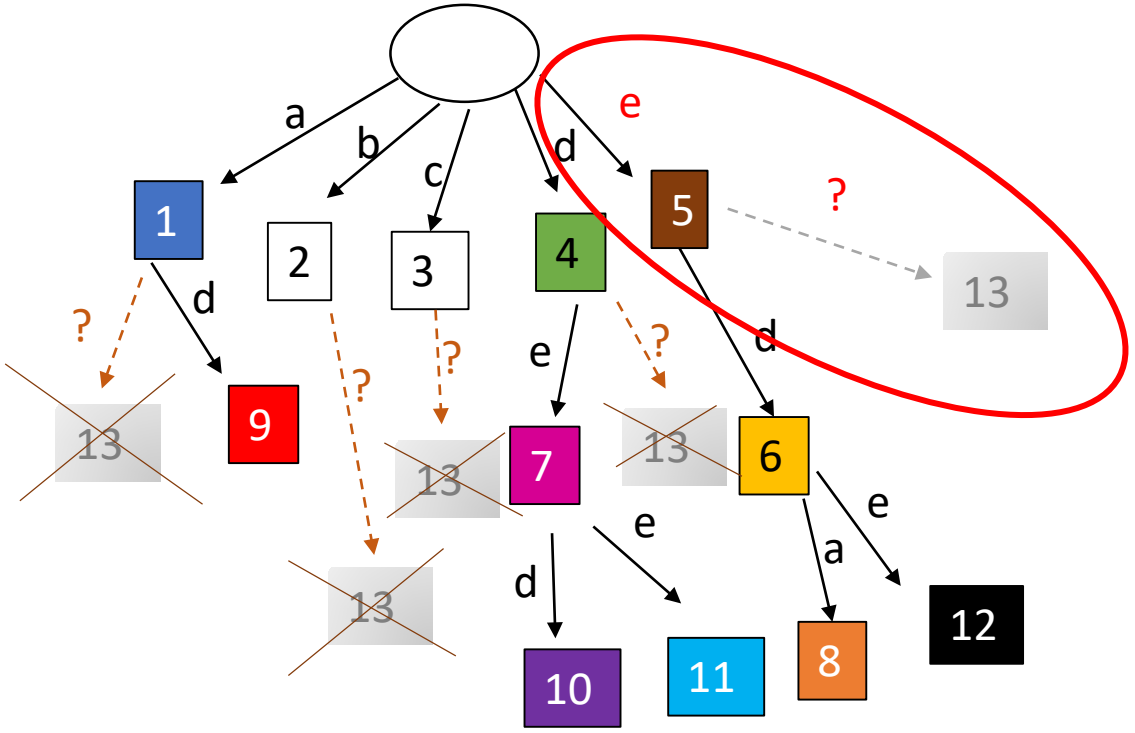
Decoder input: 5| 4| 6| 1| 7| 7 |6 | 5 | 13| 13

Decoder output: e | d | ed | a | de | de | ed | e | eσ

Well, it's clearly of the form eσ for some non-empty σ, since the encoder transmitted 'e' in the previous step, which means that it added eσ to its trie with MAX_CODE + 1 = 13…

Decoding Example

So all of the other hypotheses can now safely go

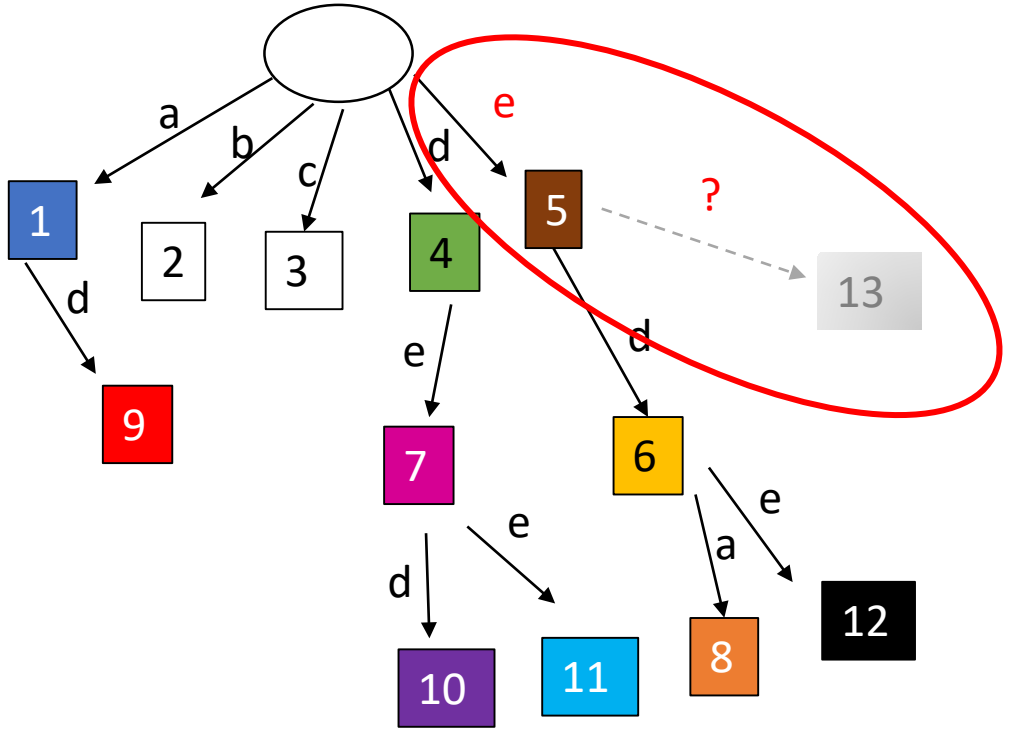


Decoder input: 5| 4| 6| 1| 7| 7 |6 | 5 | 13| 13

Decoder output: e | d | ed | a | de | de | ed | e | eσ

Well, it's clearly of the form eσ for some non-empty σ, since the encoder transmitted 'e' in the previous step, which means that it added eσ to its trie with MAX_CODE + 1 = 13…

Decoding Example

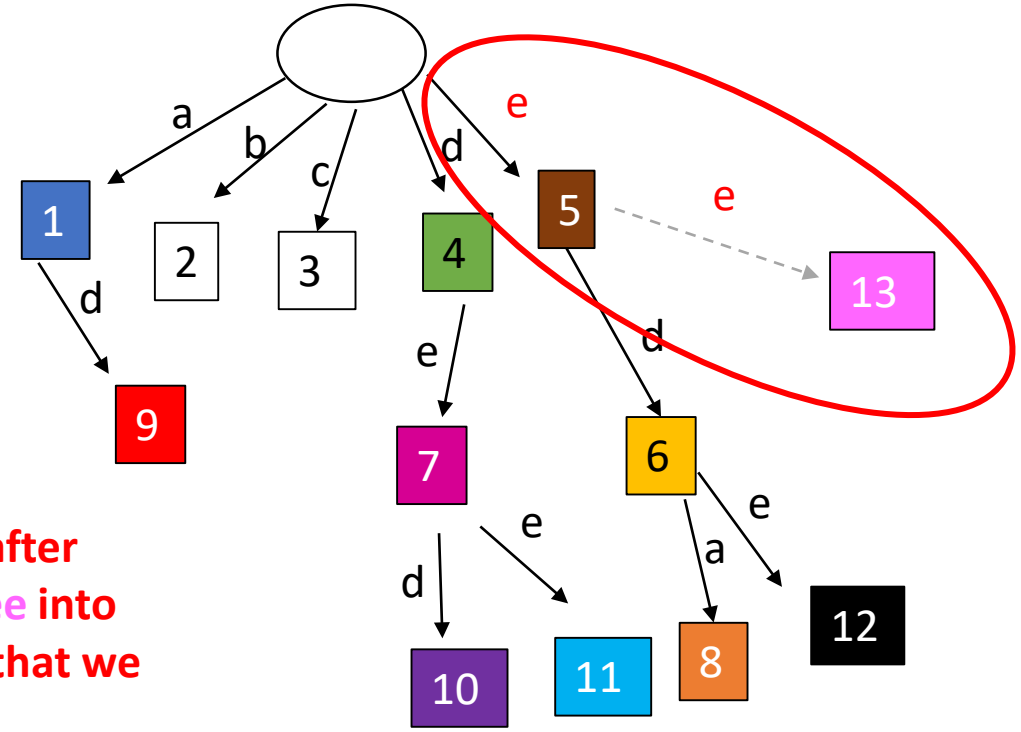


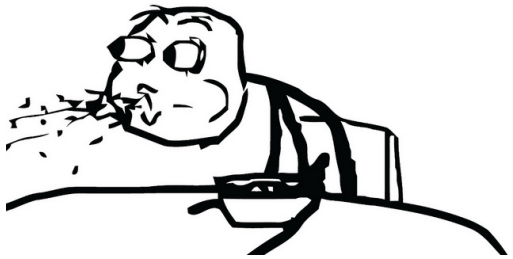
Decoder input: 5| 4| 6| 1| 7| 7 |6 | 5 | 13| 13

Decoder output: e | d | ed | a | de | de | ed | e | eσ

Well, it's clearly of the form eσ for some non-empty σ, since the encoder transmitted 'e' in the previous step, which means that it added eσ to its trie with MAX_CODE + 1 = 13…

Decoding Example





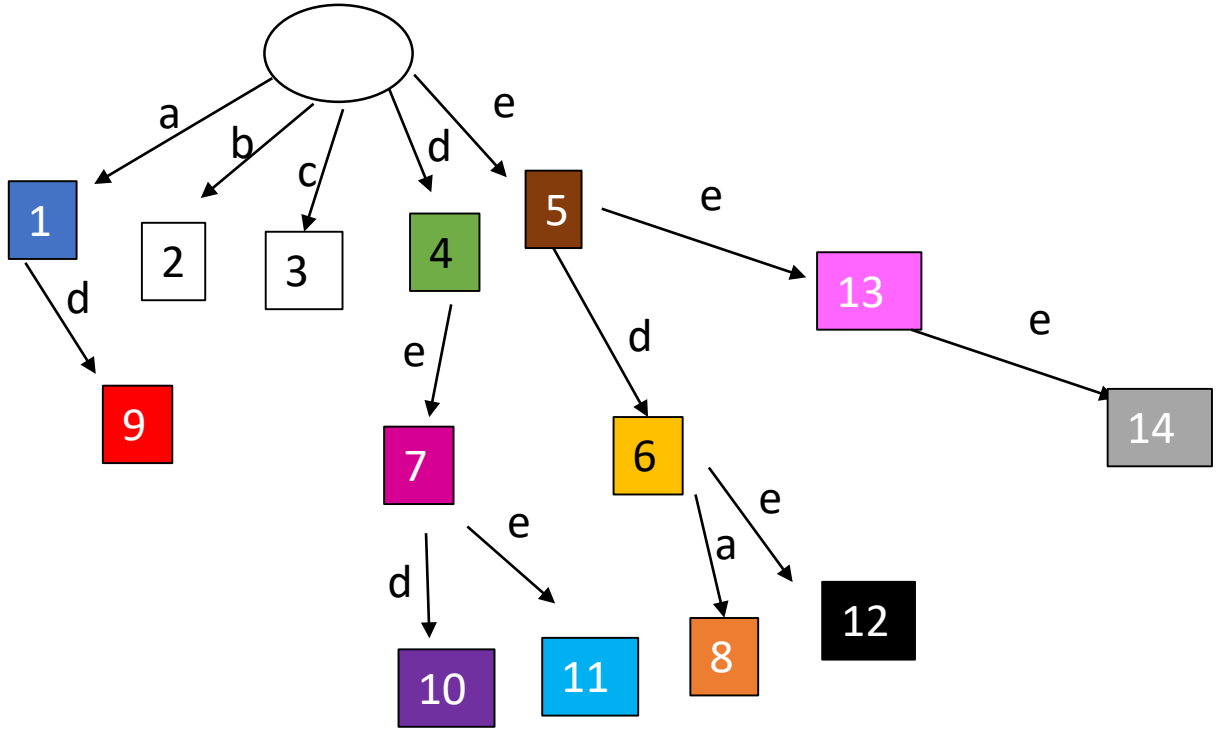
But it's also the case that σ=e, since after transmitting eσ, the encoder added ee into the trie, giving it the max code of 13 that we observed!

Well, it's clearly of the form eσ for some non-empty σ, since the encoder transmitted 'e' in the previous step, which means that it added eσ to its trie with MAX_CODE + 1 = 13…

Decoder input: 5| 4| 6| 1| 7| 7 |6 | 5 | 13| 13

Decoder output: e | d | ed | a | de | de | ed | e | ee

Decoding Example

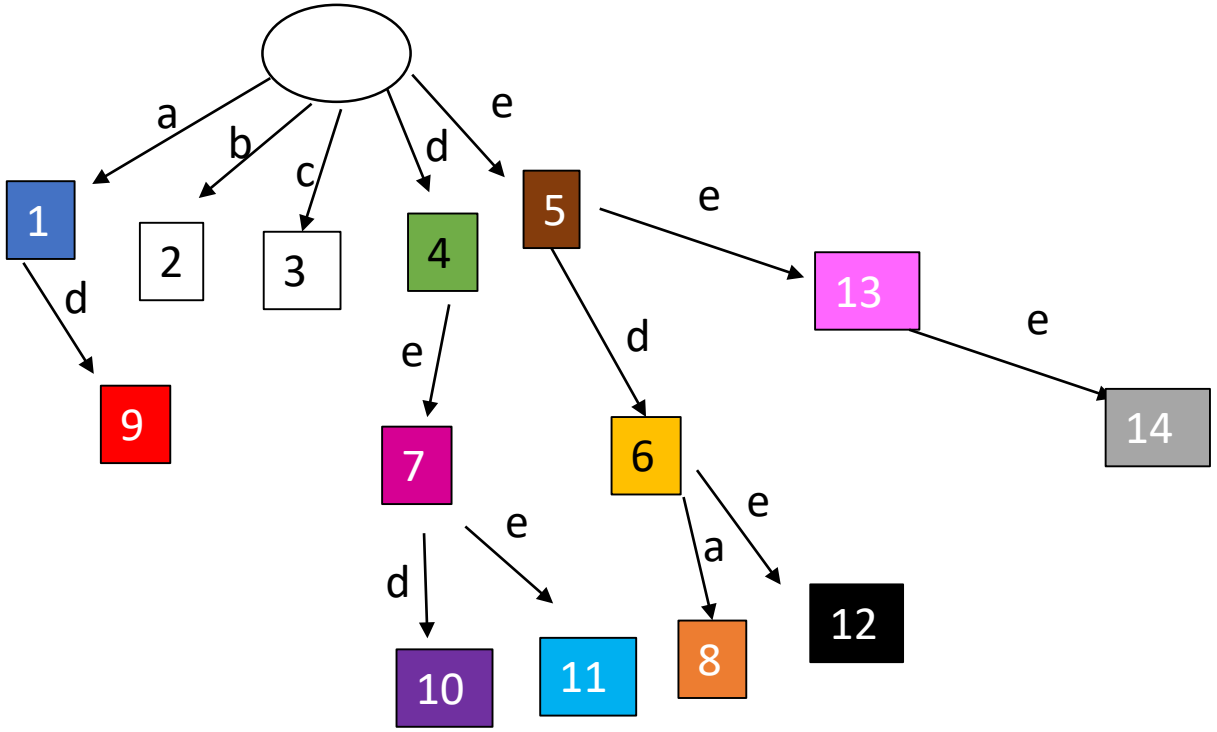


- Receive 13, **which is now in the trie**
- Output 13 again
- Expand trie with token "**eee**"

Decoder input: 5| 4| 6| 1| 7| 7 |6 | 5 | 13| 13

Decoder output: e | d | ed | a | de | de | ed | e | ee | ee

Decoding Example



- Receive 13, **which is now in the trie**
- Output 13 again
- Expand trie with token "**eee**"

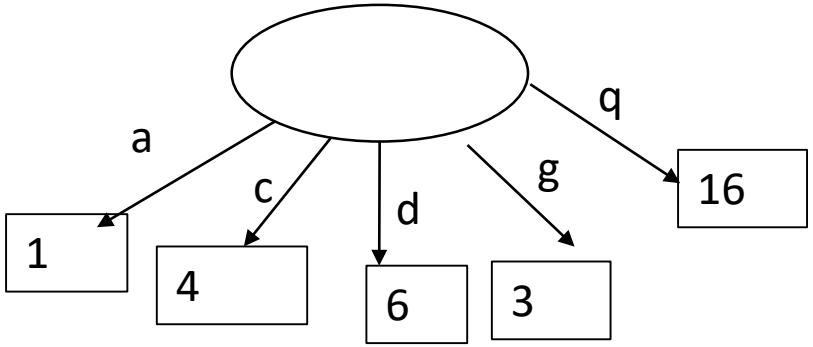
Done!

Decoder input: 5| 4| 6| 1| 7| 7 |6 | 5 | 13| 13

Decoder output: e | d | ed | a | de | de | ed | e | ee | ee

Your turn!

- Assuming this source trie…



- Decode 1 | 4 | 17 |6 |3 |16 |21| 22 | 16 | 25 | 16.

Criticisms of LZW

- Have to transmit initial source trie.
  - This can be slow for large alphabets with variable length encodings (not easy to take chunks of data without "breaking into" the invidual encodings)
- Encoding and decoding is slow, since the "string encoding" produced by LZW is variable-length: two strings of the same length are **not guaranteed** an encoding of the same length.
- …? ideas ?