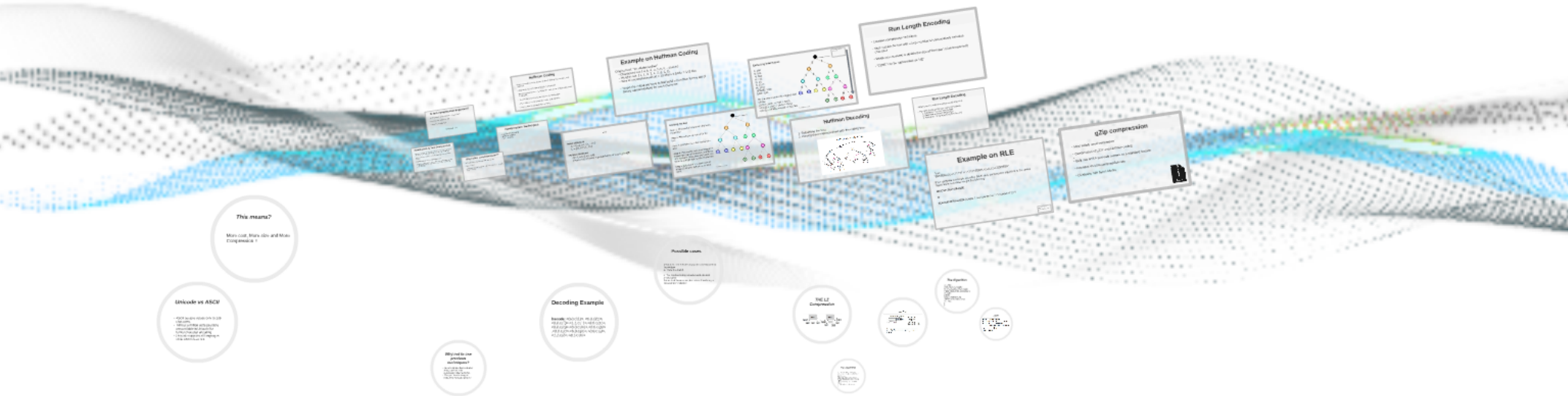
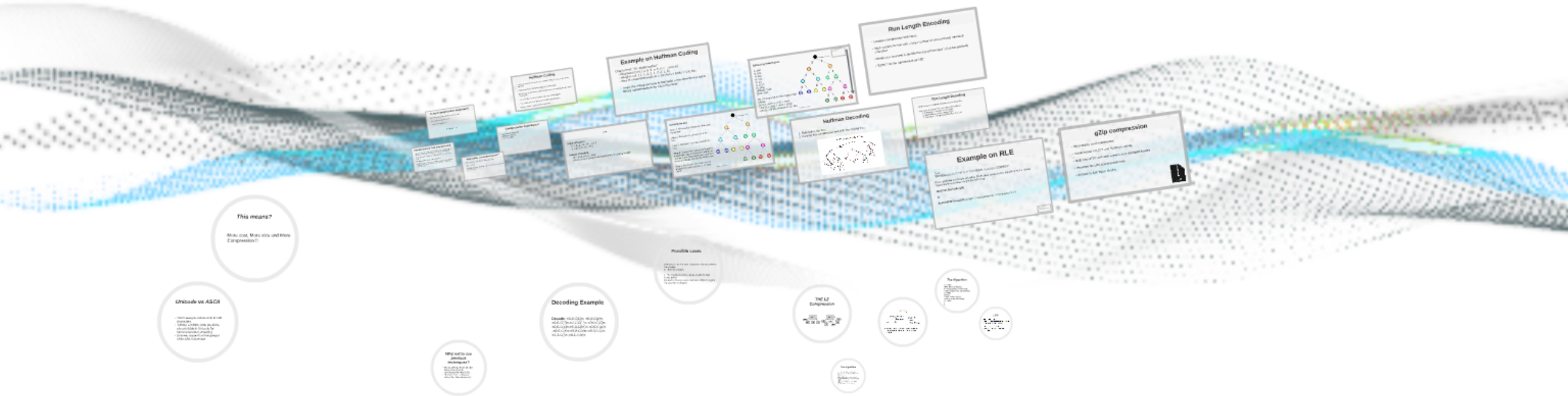
Compression



Compression



Introduction to Text Compression

- **Compression** is the conversion of data in such a format that requires few bits usually formed to store and transmit the data easily and efficiently
- Reducing resources usage, such as data storage space or transmission capacity.
- **Lossless compression** and **lossy compression**
- Examples: Shannon-Fano Coding, Huffman coding, Run length encoding. Lempel Ziv scheme (LZ77 & LZ78), Burrows–Wheeler transform and Arithmetic coding

Is text compression important?

- Each character is represented by 1 byte (8 bits)
- Fast data connections speeds
- Large data storage space

So why bother ??

Why bother compressing text?

- WhatsApp handles 50 billion text messages per day
- Mobile carriers
- Websites like Google, Facebook, Twitter etc... (Space for terabytes of text is needed)

Compression Techniques

- Run length encoding
- Huffman coding
- LZ77 & LZ78

Run Length Encoding

- Lossless compression technique
- Most suitable for text with a large number of consecutively repeated character
- Worst-case scenario is double the size of the input ! (Can be avoided)
- "EEEE" can be represented as "4E"

Example on RLE

Text :
"EEEEEECCCCTTTTTTTTTTTTTTTTEEETLLLLLLLLLLZZEEEEE"

If we apply the run-length encoding (RLE) data compression algorithm to the above hypothetical scan line, we get the following:

6E4C14T3E1T10L2Z5E

or

6E4C14T3ET10L2Z5E *(Single T represented as "T" instead of "1T")*

RLE programming

```
function RLE_COMPRESS(INPUT_STR)
    COMPRESSED_STR = "";
    count = 0
    loop do
        If NEXT_SYMBOL == CURRENT_SYMBOL then count++
        else COMPRESSED_STR += count + NEXT_SYMBOL, count = 1
    return COMPRESSED_STR;
end
```

Run Length Decoding

- Used to get the original message encoded by RLE

- Algorithm should go through a list of procedures
    1. Determine encoded string length
    2. Determine decoded string length
    3. Decode the string by starting from the end

Huffman Coding

- Lossless compression technique created in 1951 by 26-year-old, David Huffman
- Uses frequency-sorted binary trees for compression
- Similar to Shannon-Fano coding but the trees are built bottom up instead of top down
- The input is a set of characters and a set of their weights
- The output is a set of binary codes with varying lengths
- Analyze input stream before compresing

...

Input structure

C = **{** c*1*, c*2*, c*3*.... c*n* **}**

L = **{** l*1*, l*2*, l*3*... l*n* **}**

Output structure

O = **{** o*1*,o*2*,o*3*...o*n***}**

where o*i* is a binary representation of varying length

Example on Huffman Coding

Original text: "Dr. Abdennadher"

- Characters set: { a, b, d, e, h, n, r, ., space }
- Weights set: { 2, 1, 3, 2, 1, 2, 2, 1, 1}
- Size of uncompressed text = 19 chars x 8 bits = 152 bits

- To get the output we have to first build a tree then form a set of binary representations for each character

So the n
will be
(2x3) + (
+ (3x2) +
+(4x1) =

Building the tree

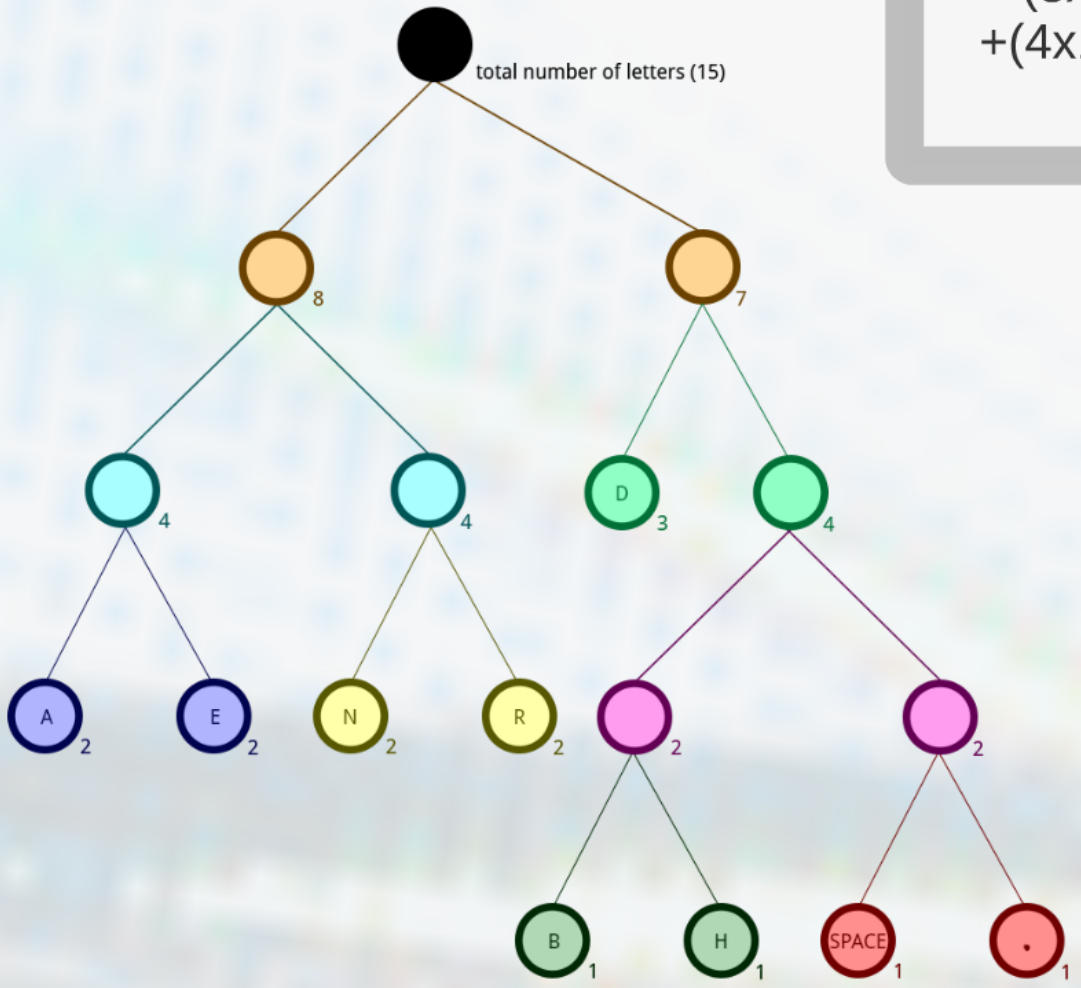
Step 1: Choose the letters with the least frequency

Step 2: Place them as leaves for the tree

Step 3: Combine each two nodes into one

Step 4: Choose the next most frequent nodes then repeat steps two and three until no more characters are found and there is a single root for the whole tree

Step 5: Add 0 and 1 to each pair of edges (0 on your left and 1 on your right)

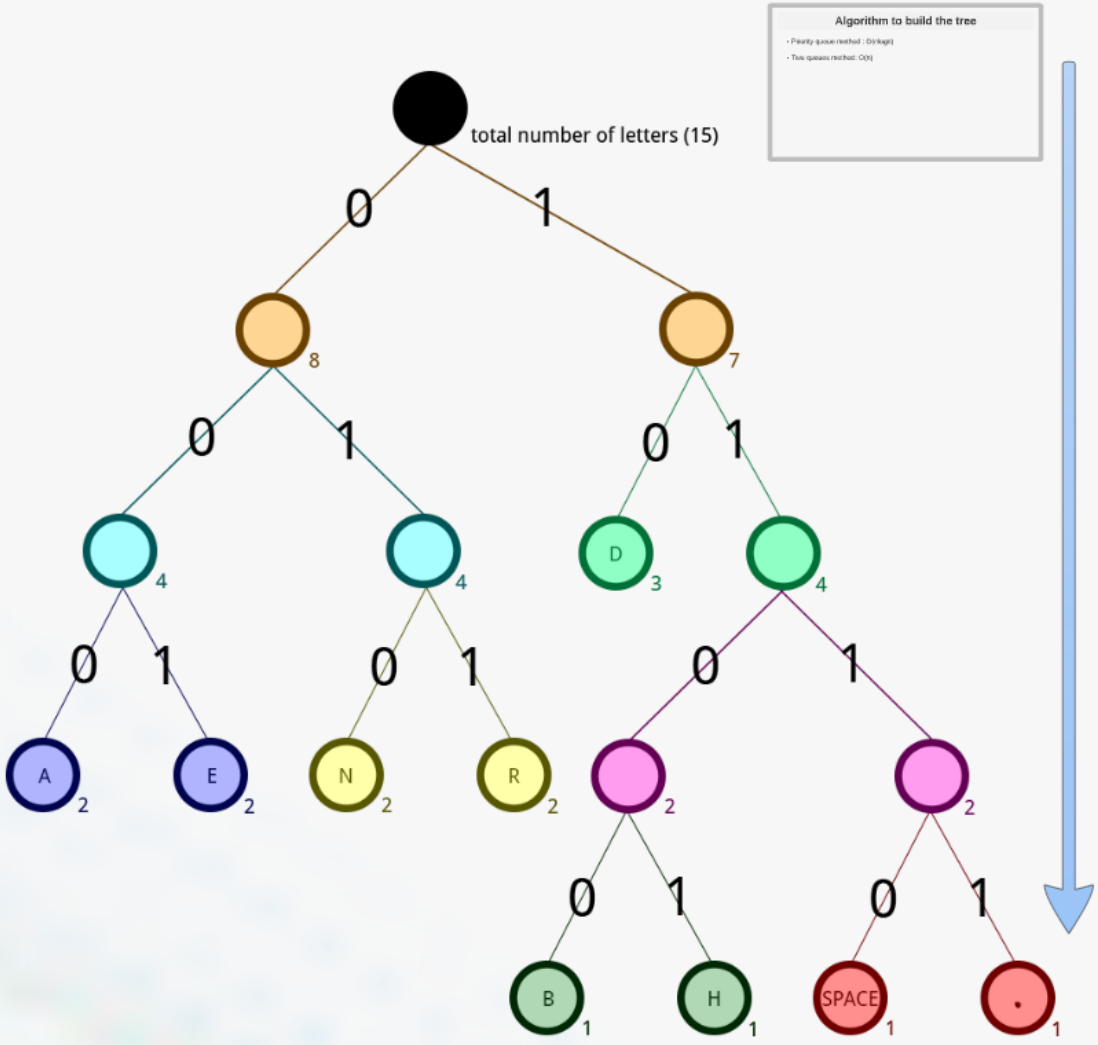


Dr.Abdennadher" huffman coding tree

Extracting information:

A: 000
E: 001
N: 010
R: 011
D: 10
B: 1100
H: 1101
SPACE: 1110
DOT: 1111

So the new size for the original text will be
(2x3) + (2x3) + (2x3) + (2x3)
+ (3x2) + (4x1) + (4x1) + (4x1)
+(4x1) = 46 bits instead of 152 bits



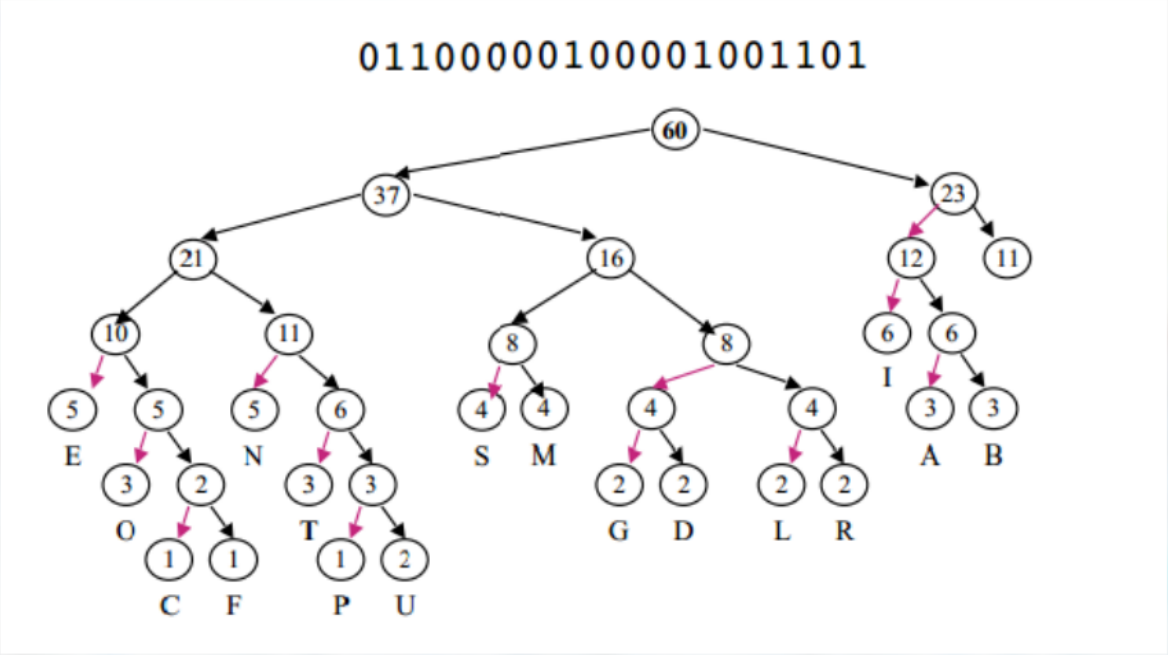
Dr.Abdennadher" huffman coding tree

Algorithm to build the tree

- Priority queue method : O(nlogn)
- Two queues method: O(n)

Huffman Decoding

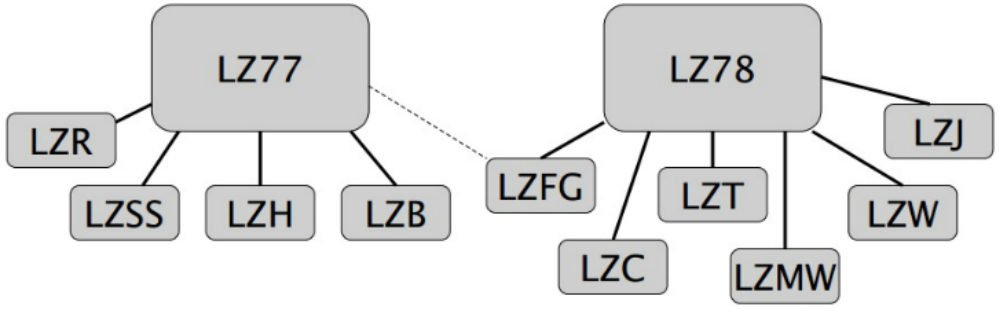
1. Rebuilding the tree.
2. Parsing the compressed text with the coding tree.



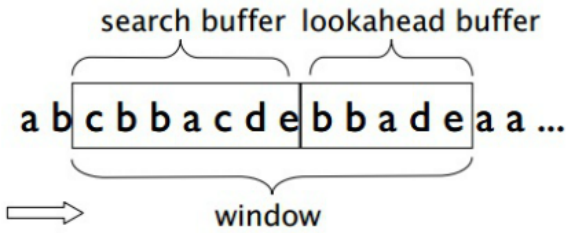
gZip compression

- Most widely used compressor
- Combination of LZ77 and Huffman coding
-
- Built into HTTP and web servers as a standard feature
- Provides 40-50% compression rate
- Divided to 32K bytes blocks

THE LZ Compression



LZ77 (1)



Match: “bba”
Position: 3
Length: 3
Next symbol: ‘d’
Output: (3, 3, ‘d’)

- Memory / speed constraints require restrictions
  ⇨ use a fixed-size window (“sliding window” principle)

Possible cases

1- No match for the next character to be encoded in the window.
2- There is a match.

3- The matched string extends inside the look-ahead buffer
For each of these cases, we have a triple to signal the case to the decode

Decoding Example

Decode: *<0,0,C(L)>, <0,0,C(z)>, <0,0,C(7)>,<1,1,C(_)>,<0,0,C(C)>, <0,0,C(o)>,<0,0,C(m)>,<0,0,C(p)> ,<0,0,C(r)>,<0,0,C(e)>,<0,0,C(s)>, <1,1,C(i)>,<8,1,C(n)>*

The Algorithm

```
while (lookAheadBuffer not empty) {
get a reference (position ,length) to longest
match;
if (length > 0) {
output (position, length, next symbol);
shift the window length+1 positions along;
} else {
output (0, 0, first symbol in lookahead
buffer);
shift the window 1 position along;
}
}
```

LZ78

- Maintain explicit dictionary
- Gradually build dictionary during encoding
- Codeword consists of 2 elements:
  - index (reference to longest match in dictionary)
  - first non-matching symbol
- Every codeword also becomes new dictionary entry

The Algorithm

```
w := NIL;
while (there is input) {
K := next symbol from input;
if (wK exists in the dictionary) {
w := wK;
} else {
output (index(w), K);
add wK to the dictionary;
w := NIL;
}
}
```

Unicode vs ASCII

- ASCII assigns values only to 128 characters
- Almost a million code positions are available in Unicode for formal character encoding
- Unicode supports all languages while ASCII does not.

This means?

More cost, More size and More Compression !!

Why not to use previous techniques?

- We actually use them but after doing some unicode compression like the SCSU.
- The goal here is simply to reduce the “Unicode penalty”

Compression