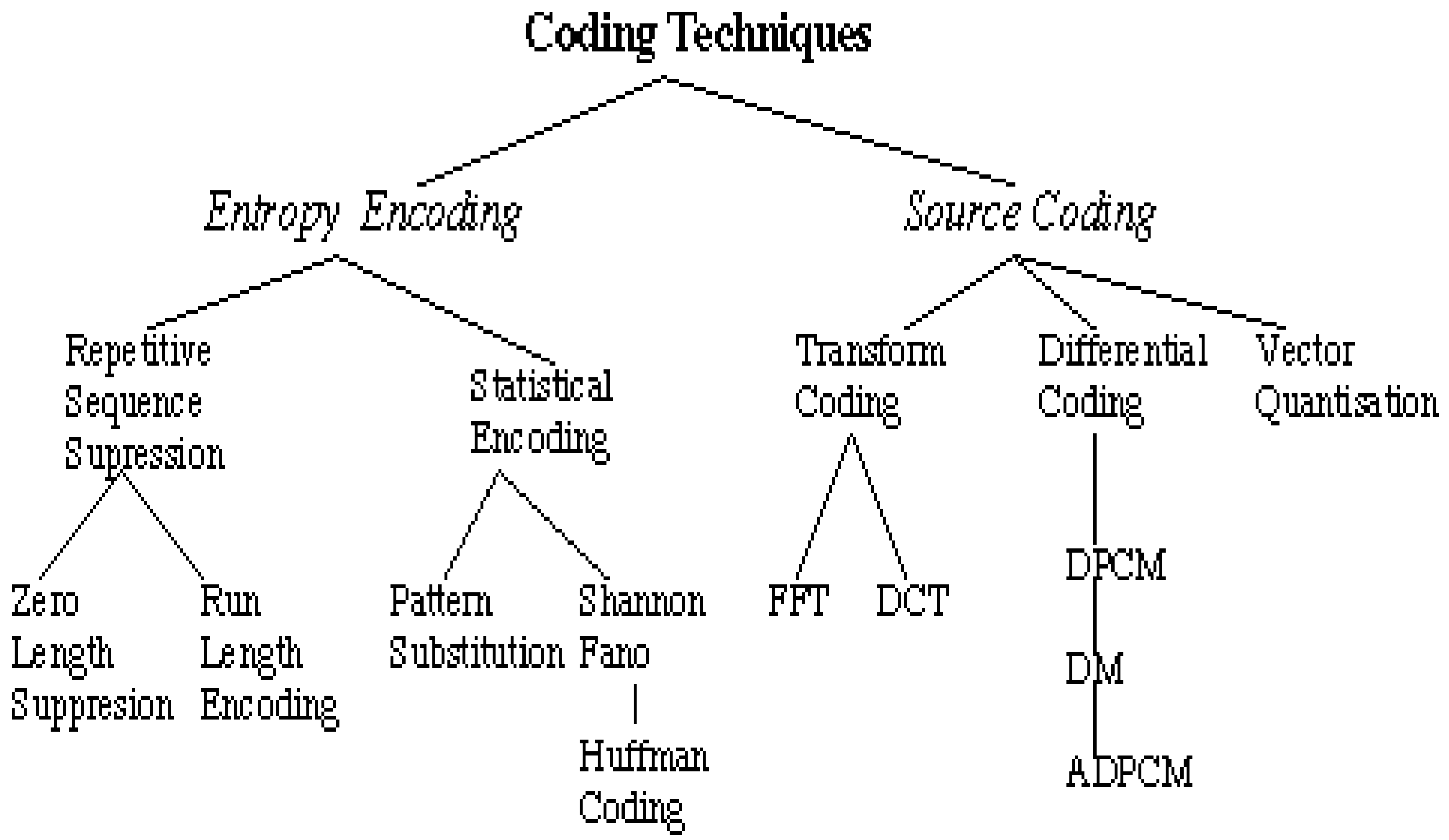

Coding Techniques
Entropy Encoding
Source Coding
Repetitive Sequence Supression
Statistical Encoding
Transform Coding
Differential Coding
Vector Quantisation
Zero Length Suppresion
Run Length Encoding
Pattern Substitution
Shannon Fano
Huffman Coding
FFT
DCT
DPCM
DM
ADPCM

Run-length encoding

- **Run-length encoding** (**RLE**) is a very simple form of data compression in which *runs* of data are stored as a single data value and count, rather than as the original run.

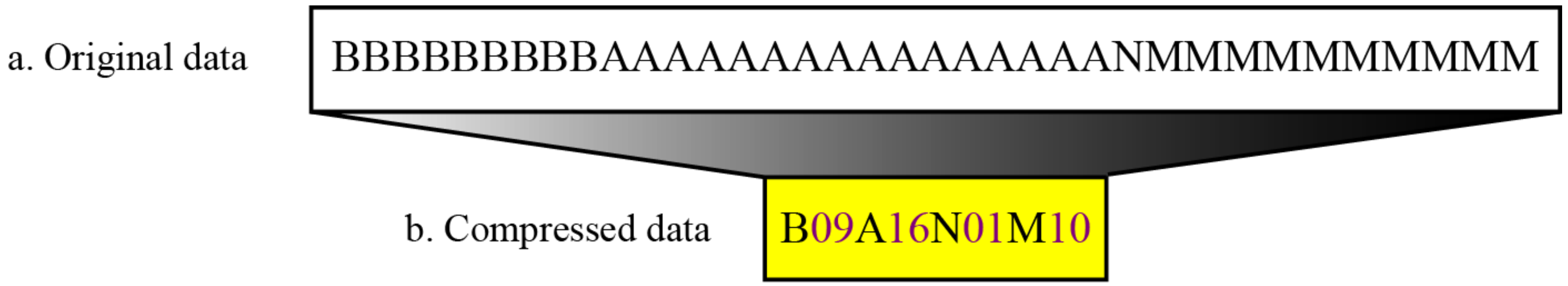
- This is most useful on data that contains many such runs.
  - for example, simple graphic images such as icons, line drawings, and animations. It is not useful with files that don't have many runs as it could greatly increase the file size.

Run-length encoding

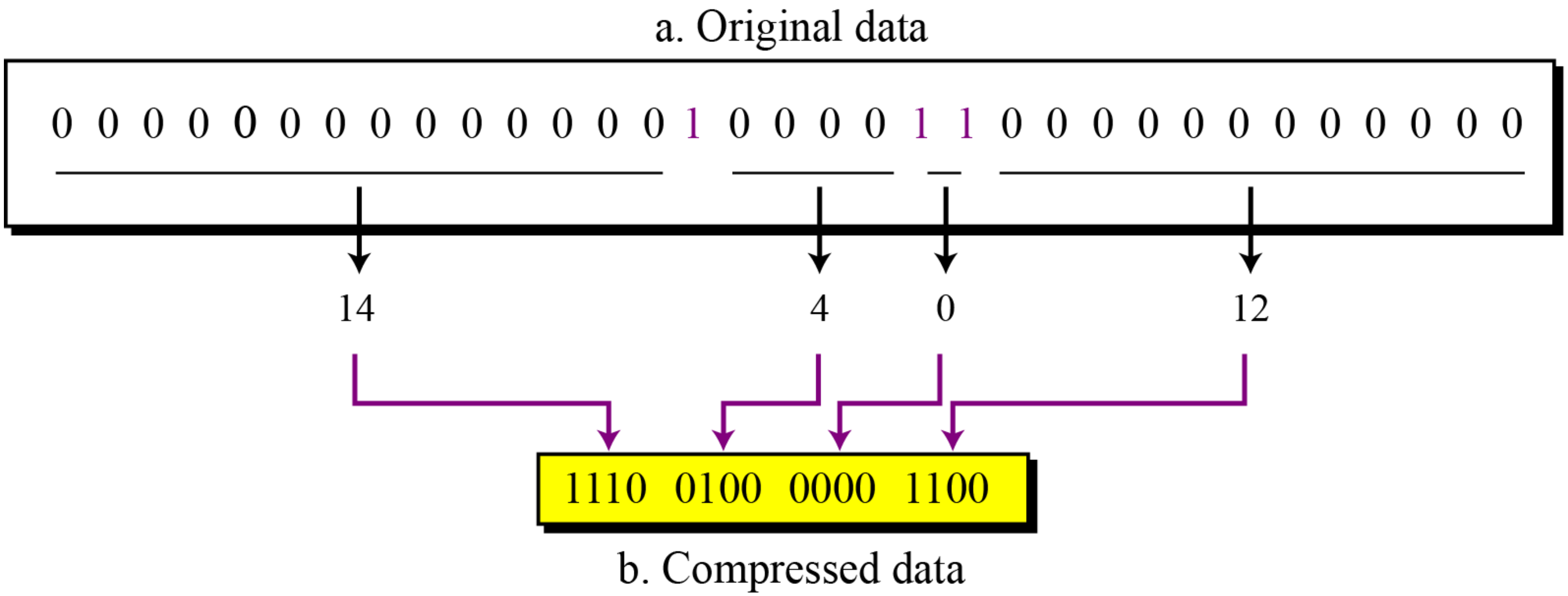
Run-length encoding can be used to compress data made of any combination of symbols. It does not need to know the frequency of occurrence of symbols and can be very efficient if data is represented as 0s and 1s.

The general idea behind this method is to replace consecutive repeating occurrences of a symbol by one occurrence of the symbol followed by the number of occurrences.

The method can be even more efficient if the data uses only two symbols (for example 0 and 1) in its bit pattern and one symbol is more frequent than the other.



Run-length encoding example



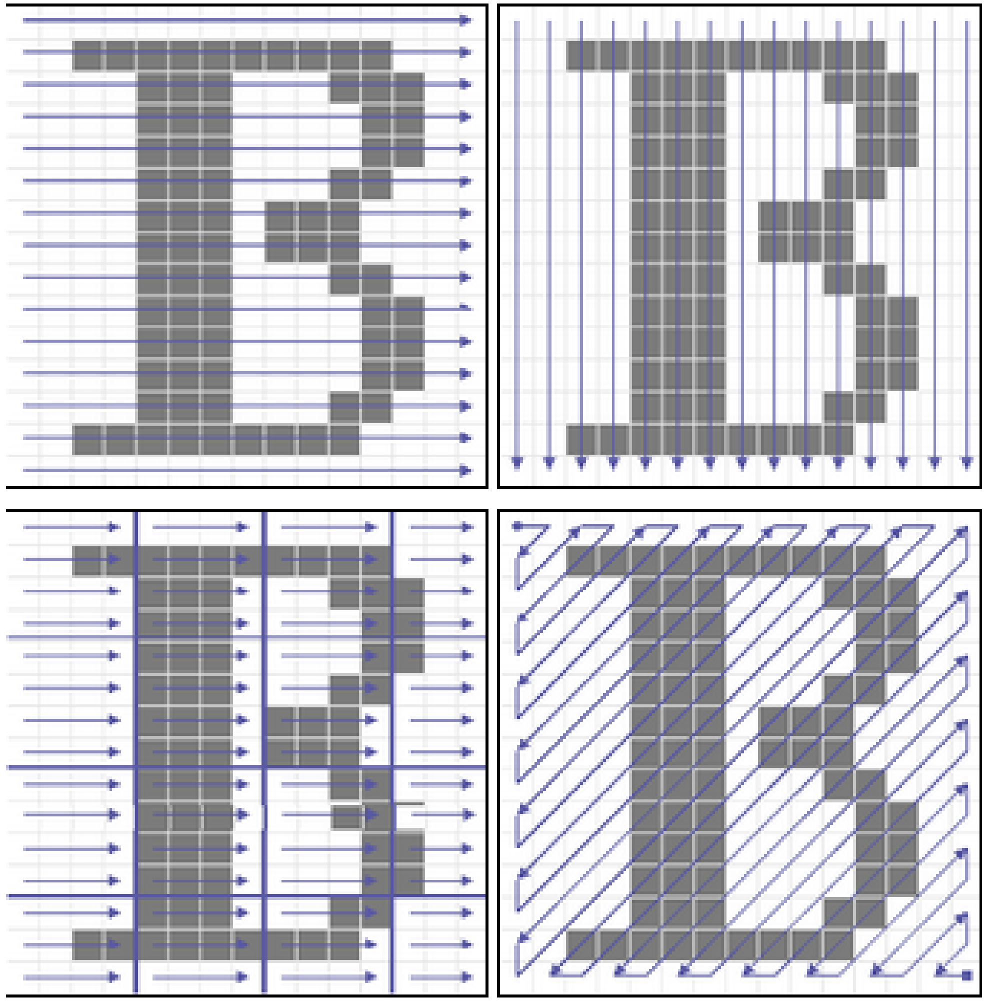
<u>Note:</u>

1110 = 14 zeroes
0100 = 4 zeroes
1100= 12 zeroes

Run-length encoding for two symbols

Sequential Processing

A general purpose encoder processes data sequentially in a one-dimensional structure. Procedures especially designed for image data could additionally benefit from their two-dimensional characteristic. The diagrams below shall demonstrate some possible strategies:



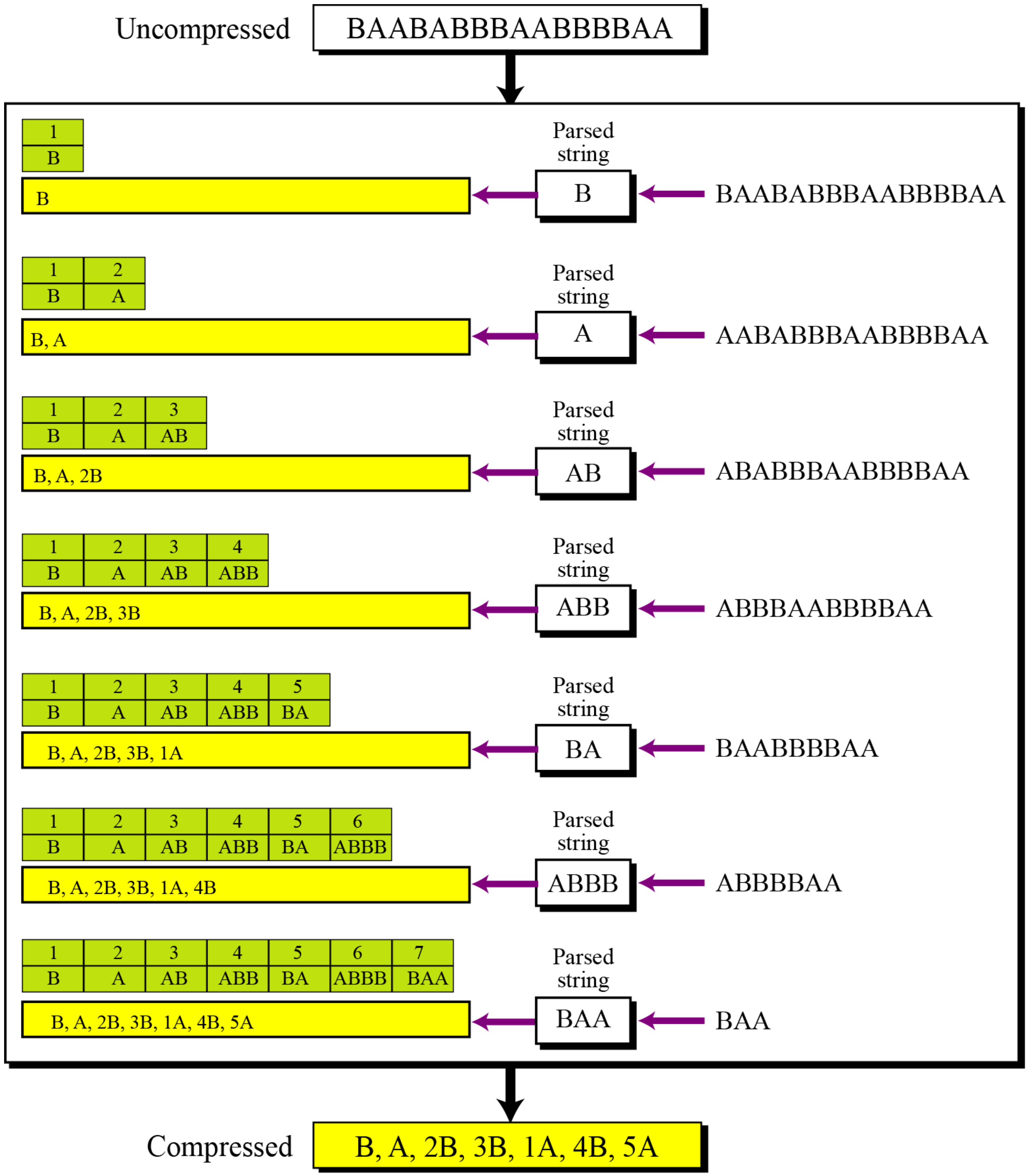
Lempel Ziv encoding

- Lempel Ziv (LZ) encoding is an example of a category of algorithms called *dictionary-based* encoding.

- The idea is to create a dictionary of strings used during the communication session.

- If both the sender and the receiver have a copy of the dictionary, then previously-encountered strings can be substituted by their index in the dictionary to reduce the amount of information transmitted.

An example of Lempel Ziv encoding



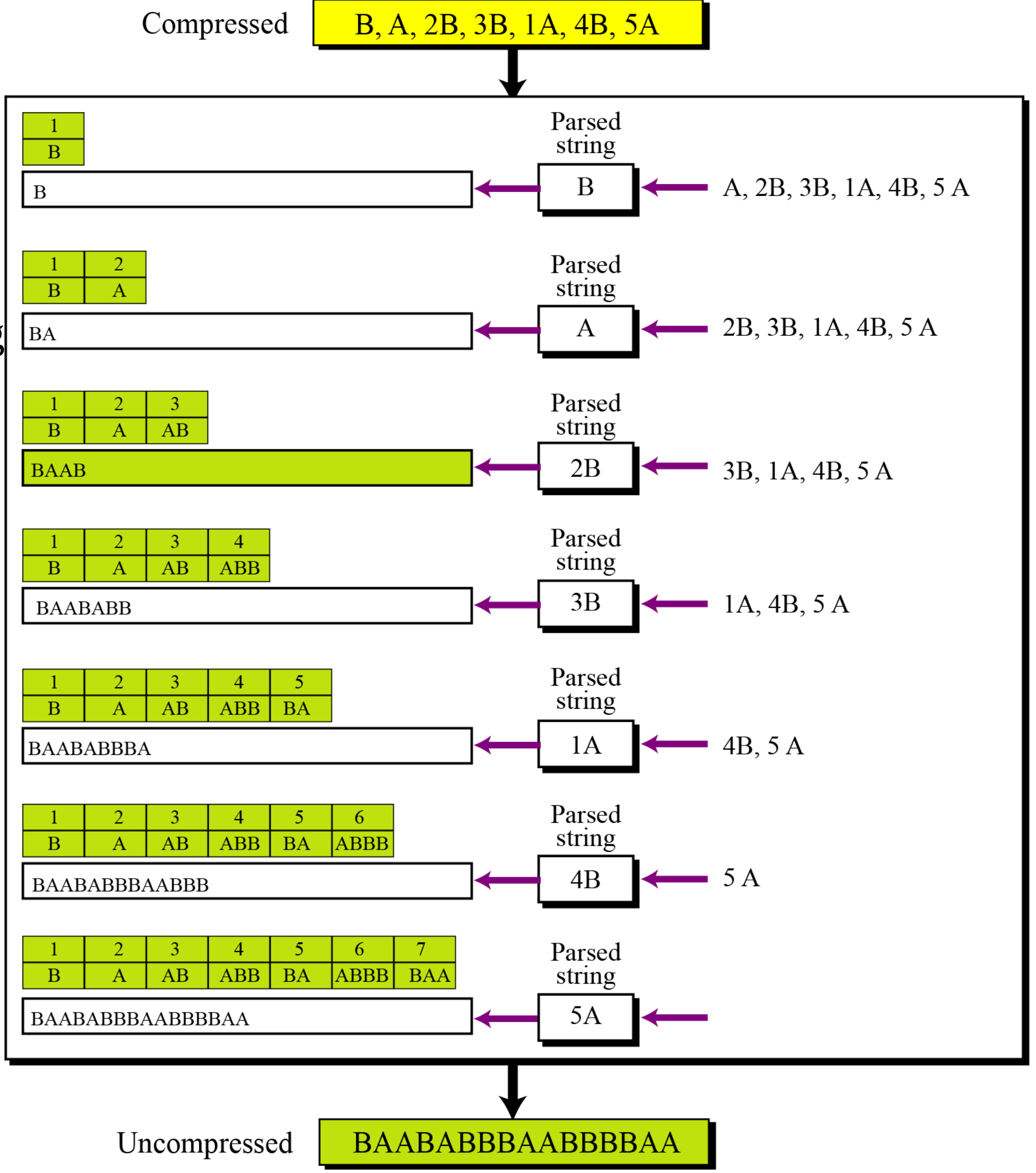
In this phase there are two concurrent events: building an indexed dictionary and compressing a string of symbols.

The algorithm extracts the smallest substring that cannot be found in the dictionary.

It then stores a copy of this substring in the dictionary as a new entry and assigns it an index value.

Compression occurs when the substring, except for the last character, is replaced with the index found in the dictionary. The process then inserts the index and the last character of the substring into the compressed string.

An example of Lempel Ziv decoding



Decompression

Decompression is the inverse of the compression process.

The process extracts the substrings from the compressed string and tries to replace the indexes with the corresponding entry in the dictionary, which is empty at first and built up gradually.

The idea is that when an index is received, there is already an entry in the dictionary corresponding to that index.

Lempel-Ziv-Welch (LZW) Algorithm

The LZW algorithm is a very common compression technique.

Suppose we want to encode the Oxford Concise English dictionary which contains about 159,000 entries. Why not just transmit each word as an 18 bit number?

Problems:

- Too many bits,
- everyone needs a dictionary,
- only works for English text.

- EXTRA

The LZW Compression Algorithm can summarised as follows:

```
w = NIL;
while ( read a character k )
   {
         if wk exists in the dictionary
                  w = wk;
         else
                  add wk to the dictionary;
                  output the code for w;
                  w = k;
   }
```

The LZW Decompression Algorithm is as follows:

```
read a character k;
    output k;
    w = k;
    while ( read a character k )
    /* k could be a character or a code. */
        {
                entry = dictionary entry for k;
                output entry;
                add w + entry[0] to dictionary;
                w = entry;
        }
```