Teoria da Informação

Trabalho Prático Número 2

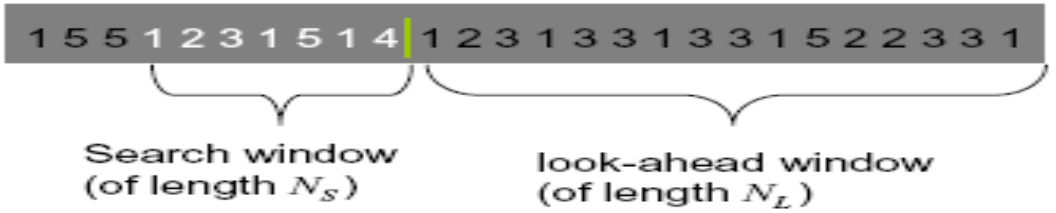
Deflate

Paulo de Carvalho

Departamento de Engenharia Informática
Faculdade de Ciências e Tecnologia da Universidade de Coimbra
Edição 2015-2016

Codificação Lempel-Ziv

- LZ77:
  - Existem dois buffers
    - *Search buffer* – buffer com os últimos N_s símbolos codificados
    - *Look-ahead buffer* – buffer com os próximos N_L símbolos a serem codificados
  - Ideia: procurar no *search buffer* o maior padrão que ocorre no *look-ahead buffer* (inicio no primeiro símbolo).
    - Codificar a posição relativa (permite codificar sequências de comprimento variável)
      - {*Offset, Length, código próximo símbolo*}
      - Próximo símbolo é enviado para o caso de não existir padrão ({*0,0,símbolo*})
      - Comprimento do padrão {0,…,N_S+N_L}



      - A codificação requer

$$\lceil \log_2 N_S \rceil + \lceil \log_2 N_S + N_L \rceil + \lceil \log_2 A \rceil \quad bits$$

Codificação Lempel-Ziv

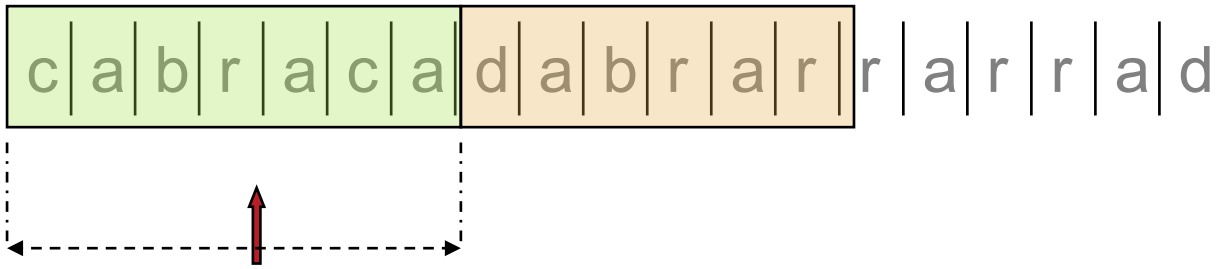
- LZ77:
  - Situações que podem ocorrer:
    - Não há correspondência no search buffer
    - Há correspondência no search buffer
    - Correspondência no search buffer estende-se ao look-ahead buffer
  - Exemplo:

c	a	b	r	a	c	a	d	a	b	r	a	r	r	a	r	r	a	d

    - Search buffer = 7 caracteres
    - Look-ahead buffer = 6 caracteres

Codificação Lempel-Ziv

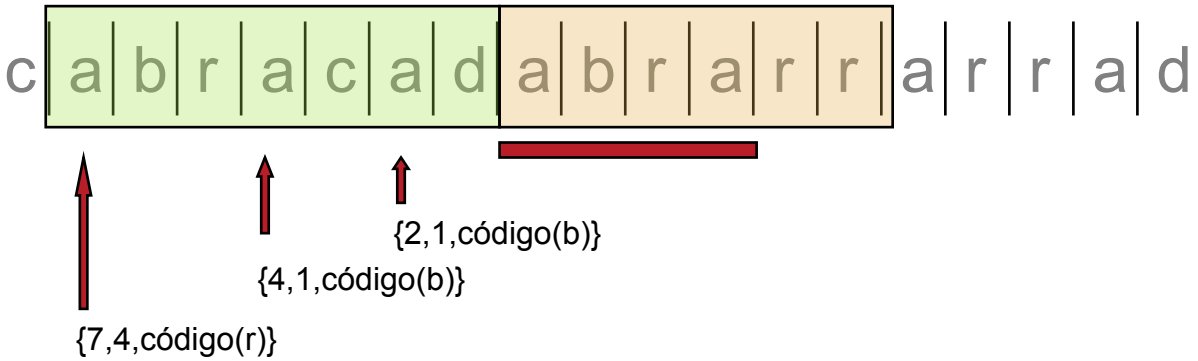
- LZ77:
  - Iteração 1:
  - Procura possíveis padrões (iniciados com o caracter “d”)



  - Código:
    - {*Offset, comprimento do padrão, próximo símbolo*} = {0,0,Código(d)}

Codificação Lempel-Ziv

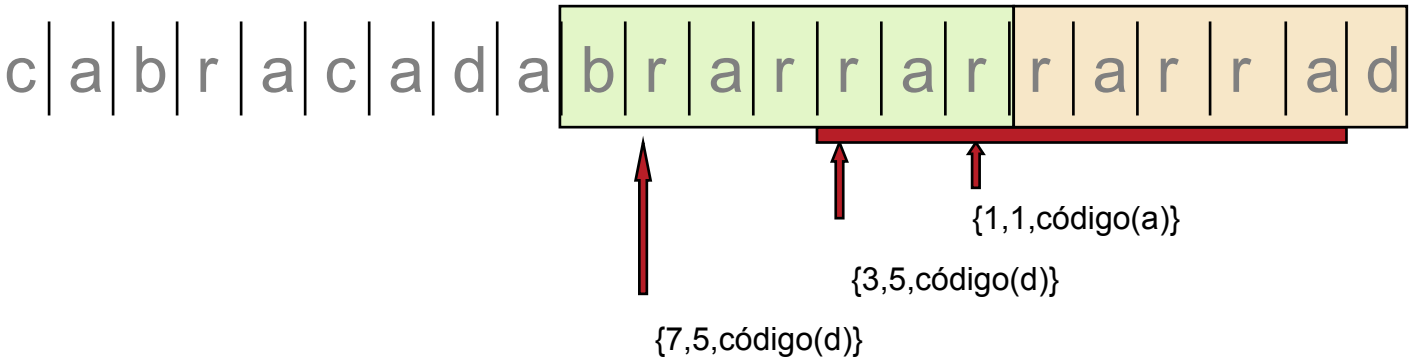
- LZ77:
  - Iteração 2:
  - Procura possíveis padrões (iniciados com o caracter “a”)



  - Código:
    - {*Offset, comprimento do padrão, próximo símbolo*} = {7,4,Código(r)}

Codificação Lempel-Ziv

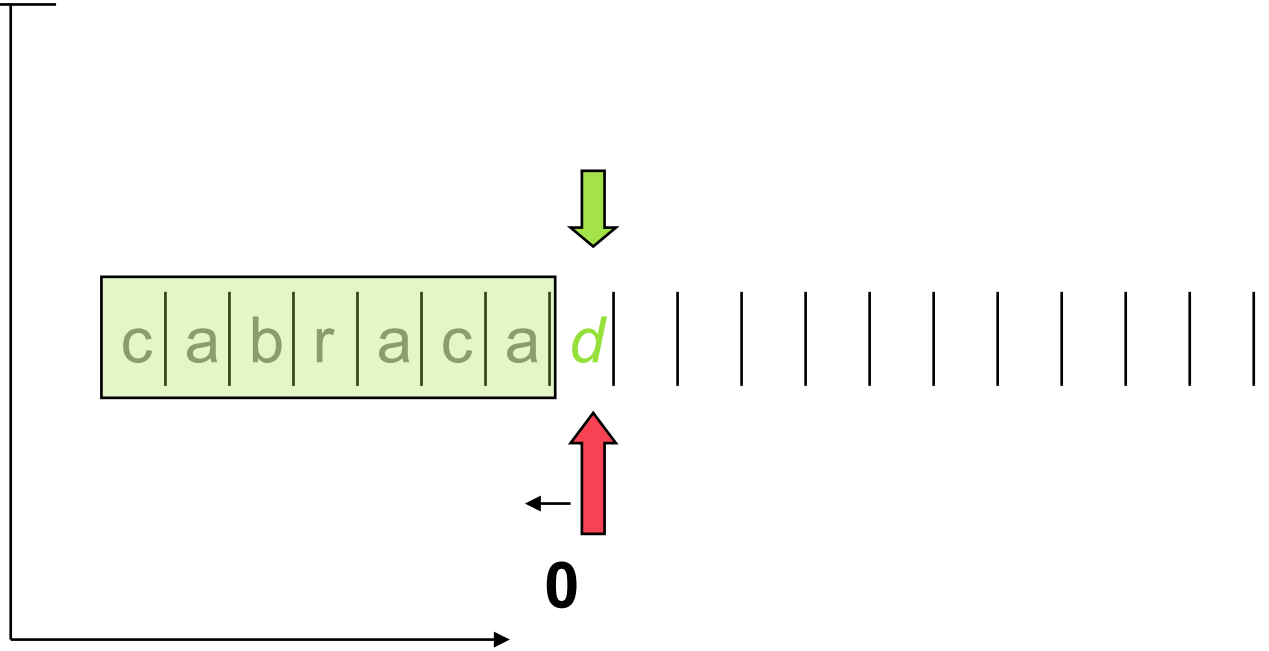
- LZ77:
  - Iteração 3:
  - Procura possíveis padrões (iniciados com o caracter “r”)



  - Código:
    - {*Offset, comprimento do padrão, próximo símbolo*} = {3,5,Código(d)}

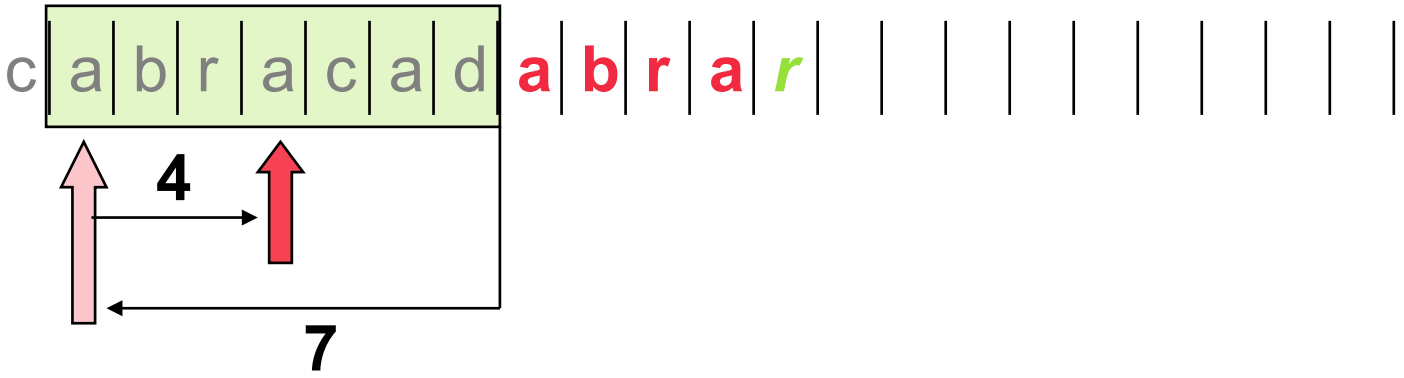
Codificação Lempel-Ziv

- LZ77:
  - Descodificador:
  - **< 0,0,Código(d)>,** <7,4,Código(r)>, <3,5,Código(d)>



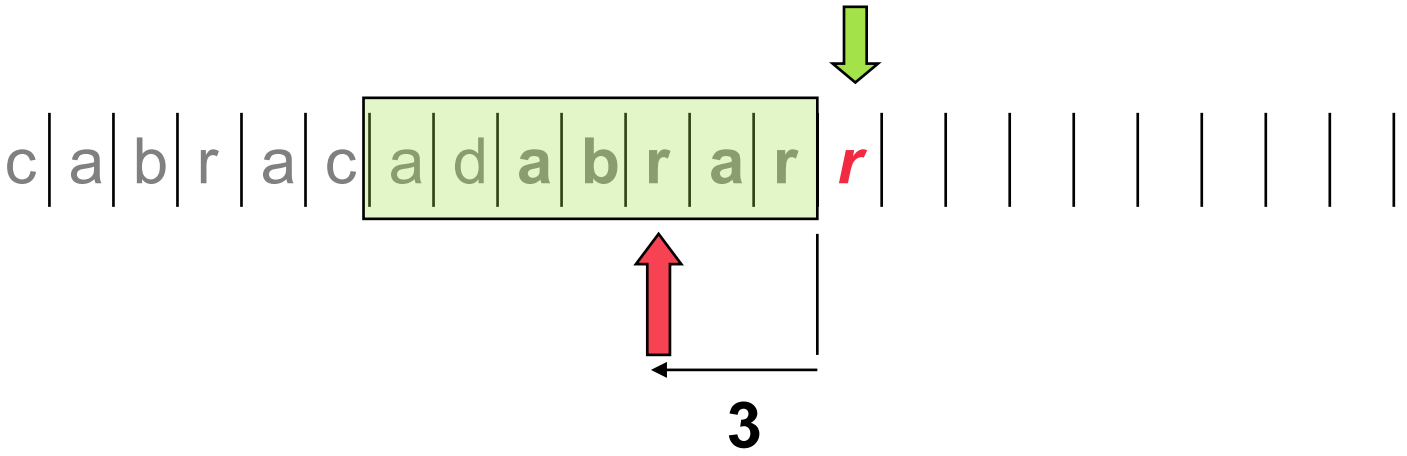
Codificação Lempel-Ziv

- LZ77:
  - Descodificador:
  - < 0,0,Código(d)>, **<7,4,Código(r)>,** <3,5,Código(d)>



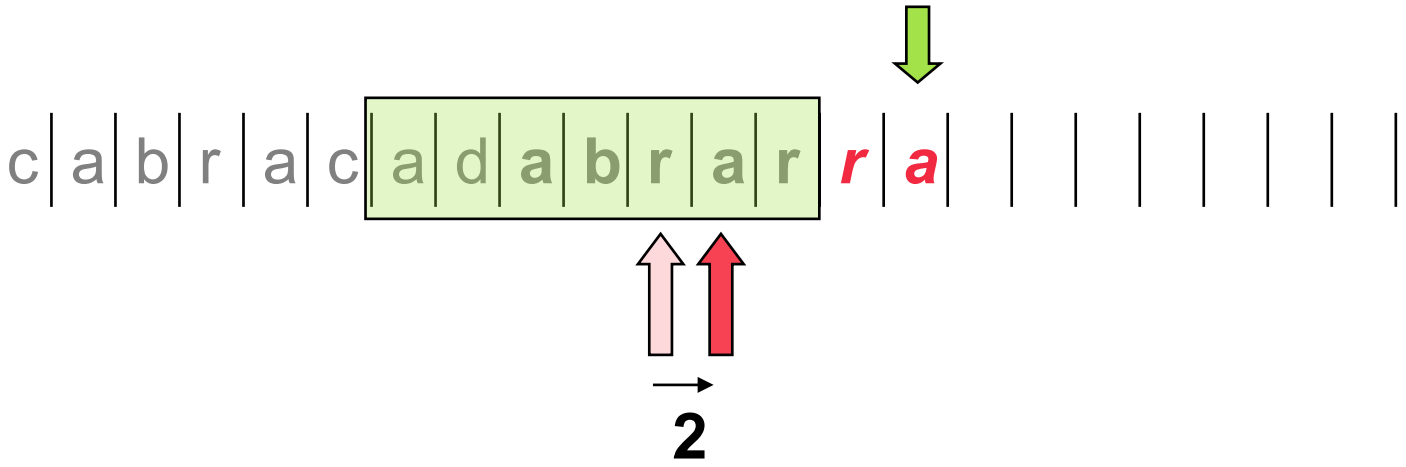
Codificação Lempel-Ziv

- LZ77:
  - Descodificador:
  - < 0,0,Código(d)>, <7,4,Código(r)>, **<3,5,Código(d)>**



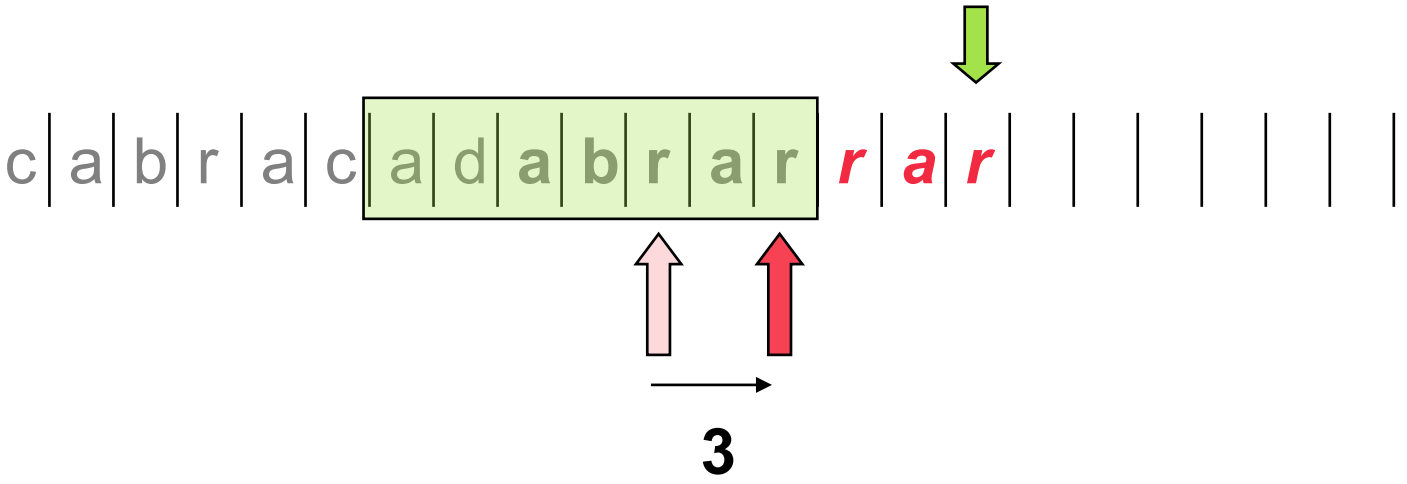
Codificação Lempel-Ziv

- LZ77:
  - Descodificador:
  - < 0,0,Código(d)>, <7,4,Código(r)>, **<3,5,Código(d)>**



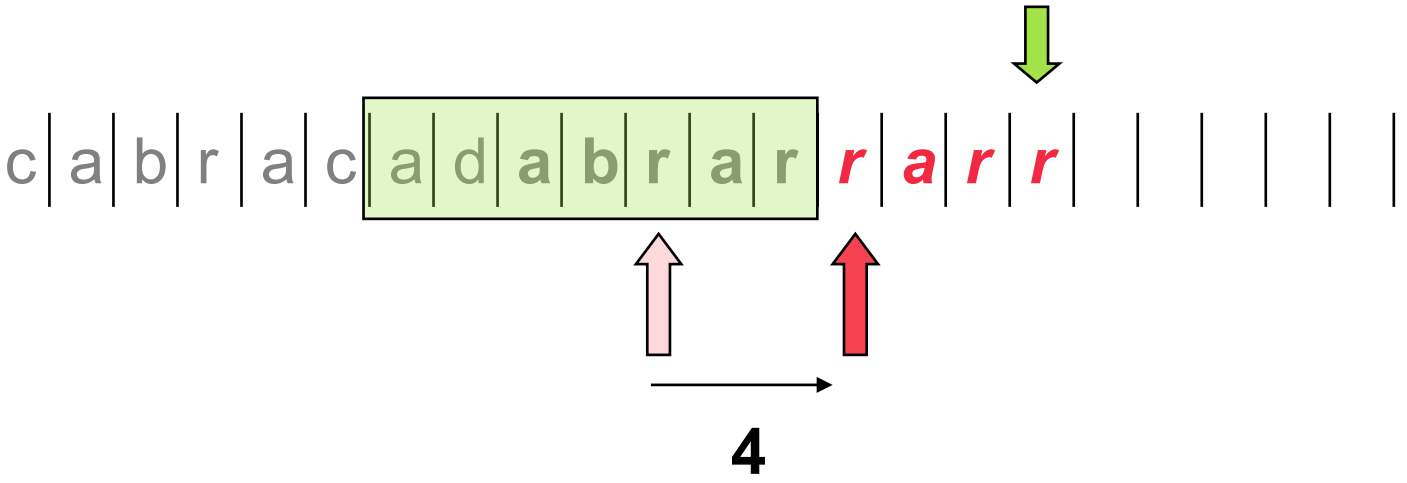
Codificação Lempel-Ziv

- LZ77:
  - Descodificador:
  - < 0,0,Código(d)>, <7,4,Código(r)>, **<3,5,Código(d)>**



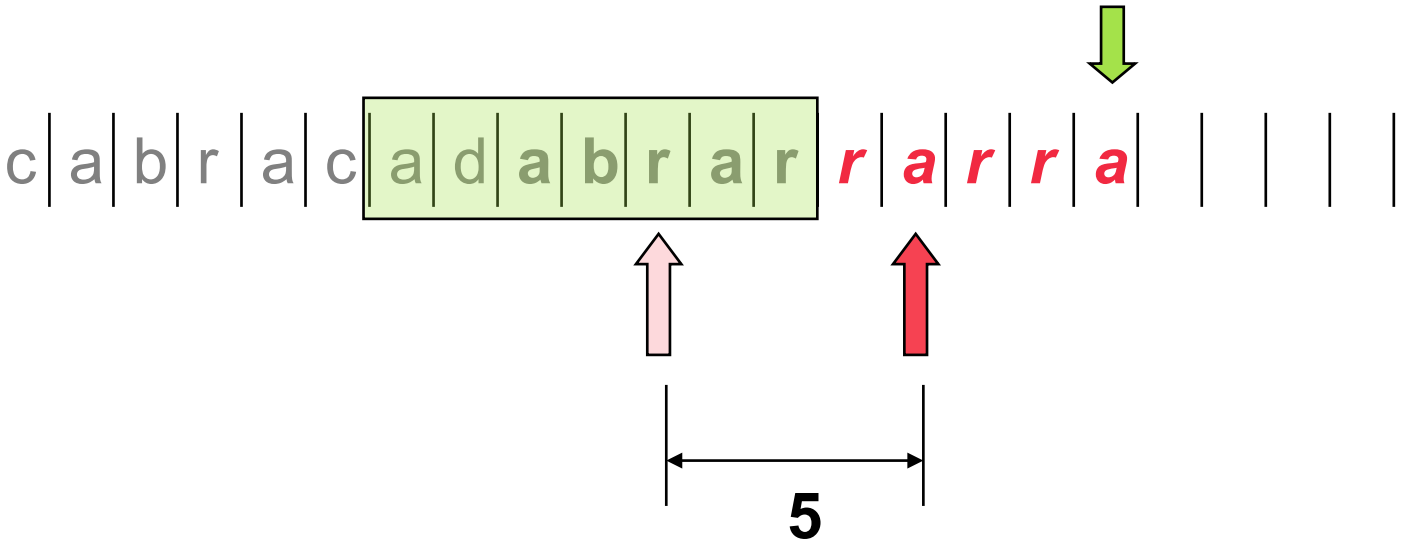
Codificação Lempel-Ziv

- LZ77:
  - Descodificador:
  - < 0,0,Código(d)>, <7,4,Código(r)>, **<3,5,Código(d)>**



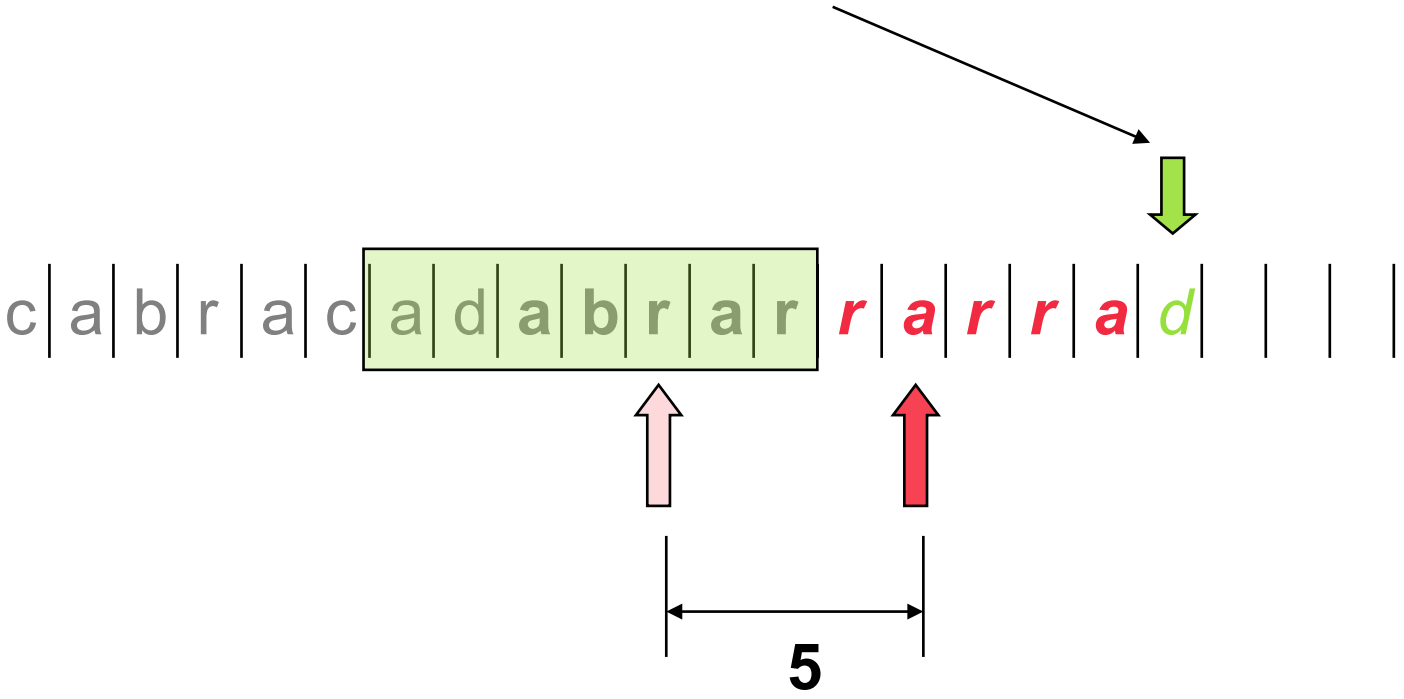
Codificação Lempel-Ziv

- LZ77:
  - Descodificador:
  - < 0,0,Código(d)>, <7,4,Código(r)>, **<3,5,Código(d)>**



Codificação Lempel-Ziv

- LZ77:
  - Descodificador:
  - < 0,0,Código(d)>, <7,4,Código(r)>, **<3,5,Código(d)>**



LZ77 – variante Deflate

- Símbolo
  - Se entre 0-255 ➔ literal
  - Se 256 ➔ fim de bloco
  - Se entre 257-285 ➔ <comprimento, distância a recuar>

Code	Extra Bits	Length(s)	Code	Extra Bits	Lengths	Code	Extra Bits	Length(s)
257	0	3	267	1	15,16	277	4	67-82
258	0	4	268	1	17,18	278	4	83-98
259	0	5	269	2	19-22	279	4	99-114
260	0	6	270	2	23-26	280	4	115-130
261	0	7	271	2	27-30	281	5	131-162
262	0	8	272	2	31-34	282	5	163-194
263	0	9	273	3	35-42	283	5	195-226
264	0	10	274	3	43-50	284	5	227-257
265	1	11,12	275	3	51-58	285	0	258
266	1	13,14	276	3	59-66			

Comprimento

LZ77 – variante Deflate

- Símbolo Distância
  - Se entre 0-255 ➔ literal
  - Se 256 ➔ fim de bloco
  - Se entre 257-285 ➔ <comprimento, distância a recuar>

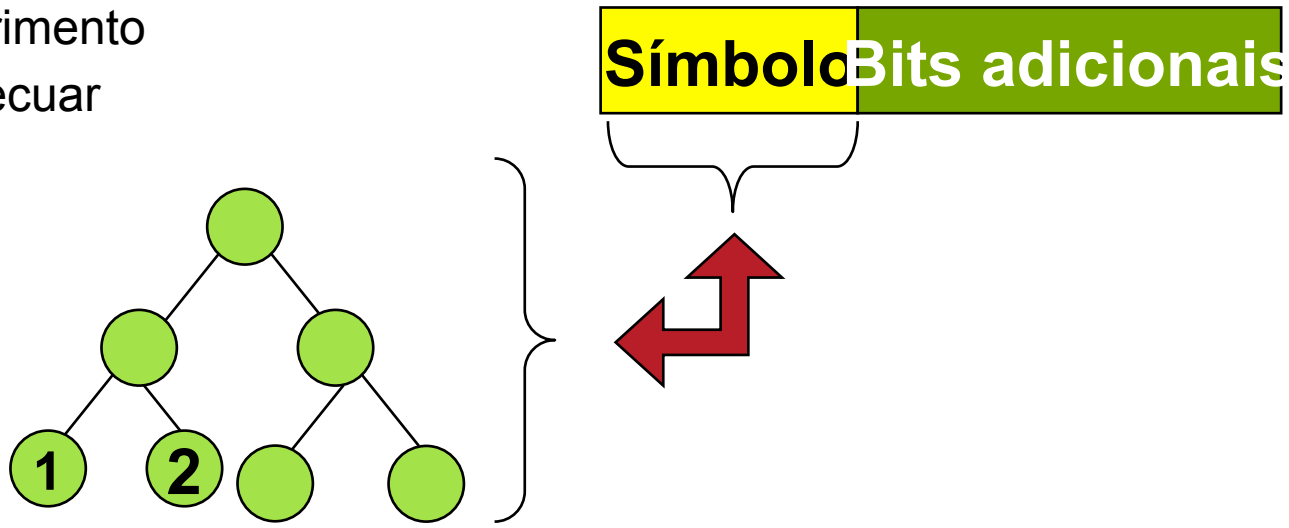
	Extra			Extra			Extra	
Code	Bits	Dist	Code	Bits	Dist	Code	Bits	Distance
0	0	1	10	4	33-48	20	9	1025-1536
1	0	2	11	4	49-64	21	9	1537-2048
2	0	3	12	5	65-96	22	10	2049-3072
3	0	4	13	5	97-128	23	10	3073-4096
4	1	5,6	14	6	129-192	24	11	4097-6144
5	1	7,8	15	6	193-256	25	11	6145-8192
6	2	9-12	16	7	257-384	26	12	8193-12288
7	2	13-16	17	7	385-512	27	12	12289-16384
8	3	17-24	18	8	513-768	28	13	16385-24576
9	3	25-32	19	8	769-1024	29	13	24577-32768

Distância

LZ77 – variante Deflate

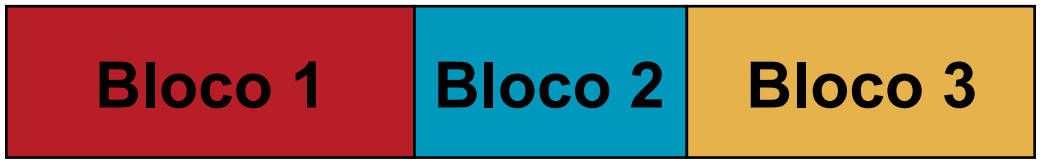
- Quantos bits?

- Existem códigos de Huffman distintos:
  - Literal/Comprimento
  - Distância a recuar

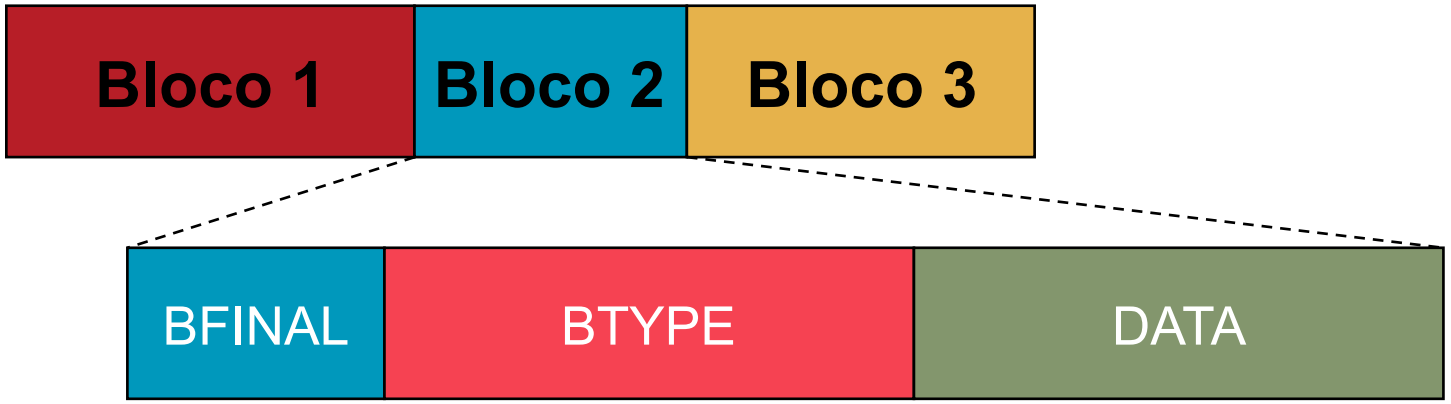


O Deflate - Bitstream

- Organização por blocos
- Cada bloco termina quando o código de Huffman tiver que ser alterado!
  - 256 ➔ fim de bloco



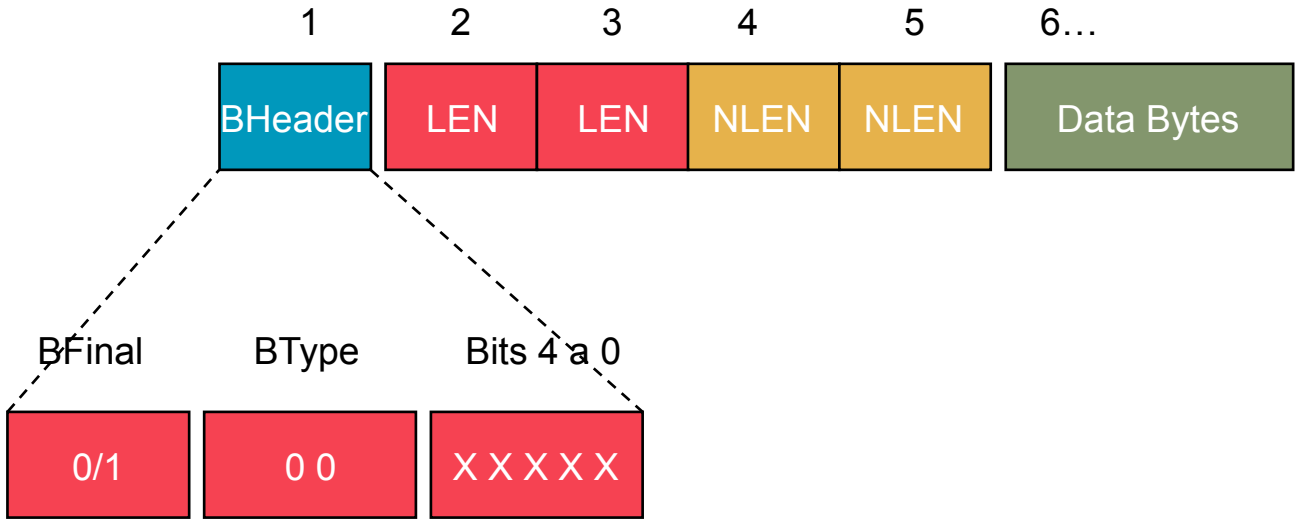
O Deflate



- BFINAL: 1 bit
  - =1 se este for o último bloco
  - =0 caso contrário
- BTYPE: 2 bits
  - 00 – sem compressão
  - 01 – comprimido com Huffman fixo
  - 10 – comprimido com Huffman dinâmico
  - 11 - reservado

O Deflate - Algoritmo

- **do**
- read block header from input stream.
- **if stored with no compression**
- skip any remaining bits in current partially processed byte
- read LEN and NLEN
- copy LEN bytes of data to output
- otherwise



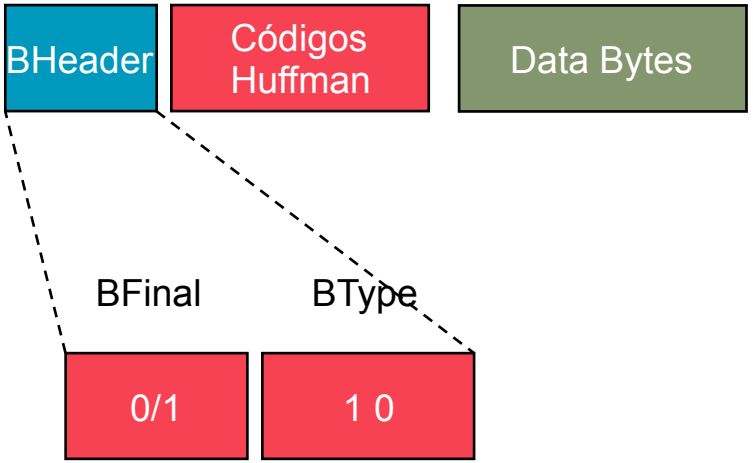
- while not last block

O Deflate - Algoritmo

- **do**
- read block header from input stream.
- **…**
- otherwise
- if compressed with dynamic Huffman codes
- read representation of code trees

- ….

- while not last block

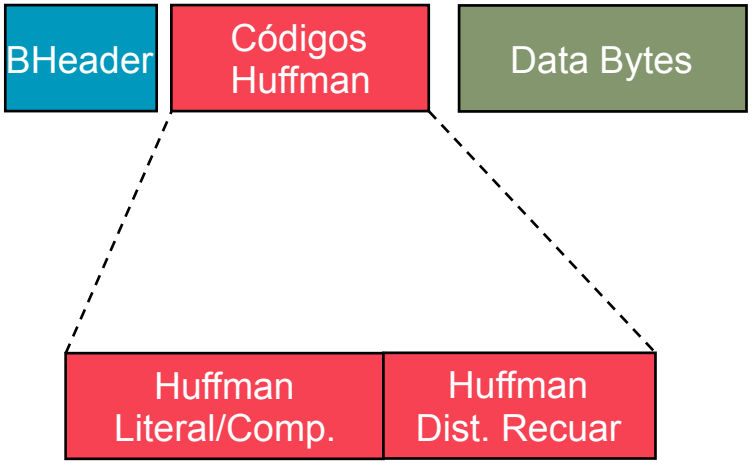


O Deflate - Algoritmo

- **do**
- read block header from input stream.
- **…**
- otherwise
- if compressed with dynamic Huffman codes
- read representation of code trees

- ….

- while not last block



O Deflate – Códigos de Huffman

- Pressupostos:
  - **Regra 1:** Todos os códigos com um dada número de bits são ordenados pela ordem lexicográfica
    - Para códigos com o mesmo comprimento usamos a mesma ordem que no alfabeto
  - **Regra 2:** Códigos mais curtos precedem os códigos mais longos em termos lexicográfica
    - Os códigos mais curtos têm valor numérico inferior
- Construção implícita com base nos comprimentos em bits
  - Exemplo
    - {2,1,3,3}
    - A=10
    - B=0 (Regra 2)
    - C=110 (Regra 1)
    - D=111 (Regra 1)

O Deflate – Códigos de Huffman

- Construção dos códigos

1. Construção dos códigos usando Huffman normal
   - Extrair os comprimentos em bits de cada símbolo

2. Aplicação da **regra 2**
   – O primeiro código é 0

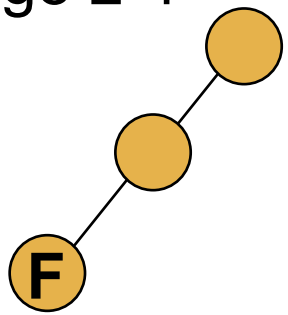
3. Aplicação da **regra 1**
   - Dado o código presente *z* com *k* bits, atribuir pela ordem lexicográfica o código *z+i*

O Deflate – Códigos de Huffman

- Construção dos códigos

  Aplicação da **regra 1**

  - Dado o código presente *z* com *k* bits, atribuir pela ordem lexicográfica o código z+i
    - Exemplo:
      - » {A,B,C,D,E,F,G,H,…}
      - » {3,3,3,3,3,2,4,4,…}

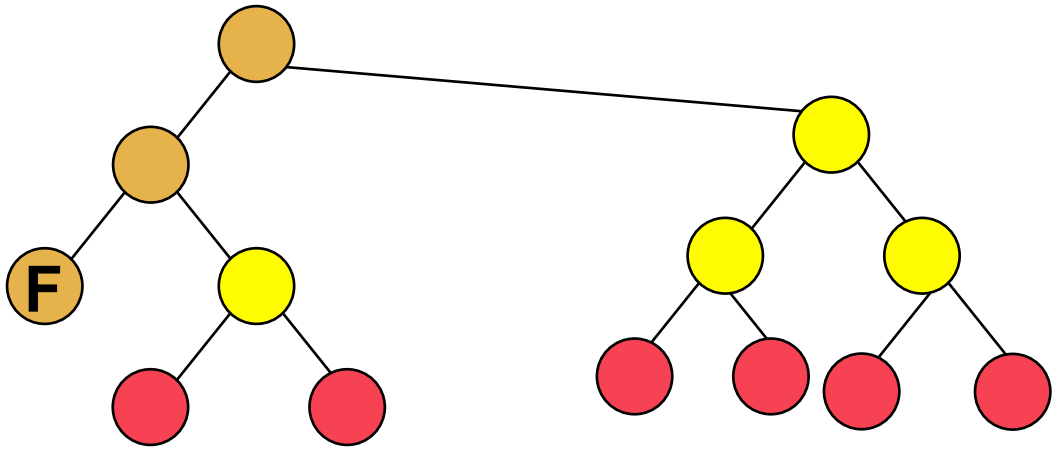


- Observação: F não pode ser prefixo do que já existe.

O Deflate – Códigos de Huffman

- Construção dos códigos
  - Exemplo:
    - {A,B,C,D,E,F,G,H,…}
    - {3,3,3,3,3,2,4,4,…}

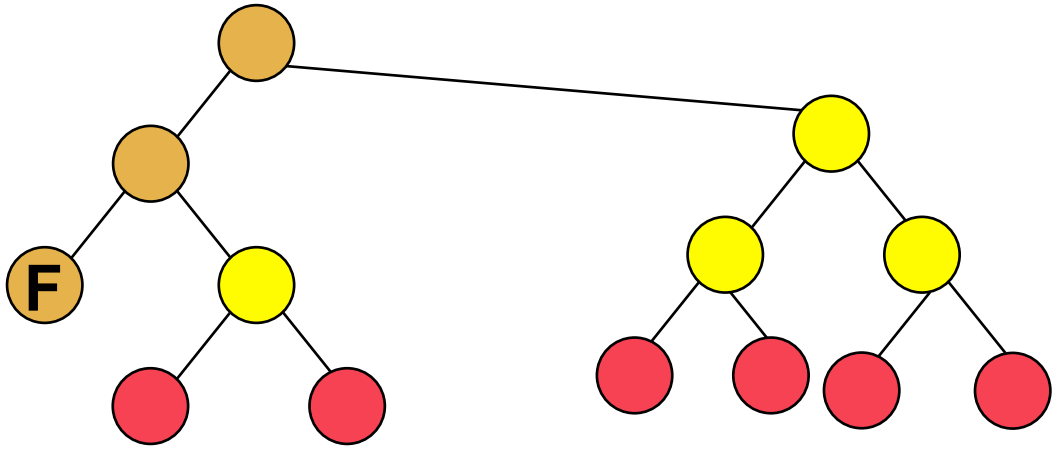
Como atribuir os códigos?



O Deflate – Códigos de Huffman

- Construção dos códigos
  - Exemplo:
    - {A,B,C,D,E,F,G,H,…}
    - {3,3,3,3,3,2,4,4,…}

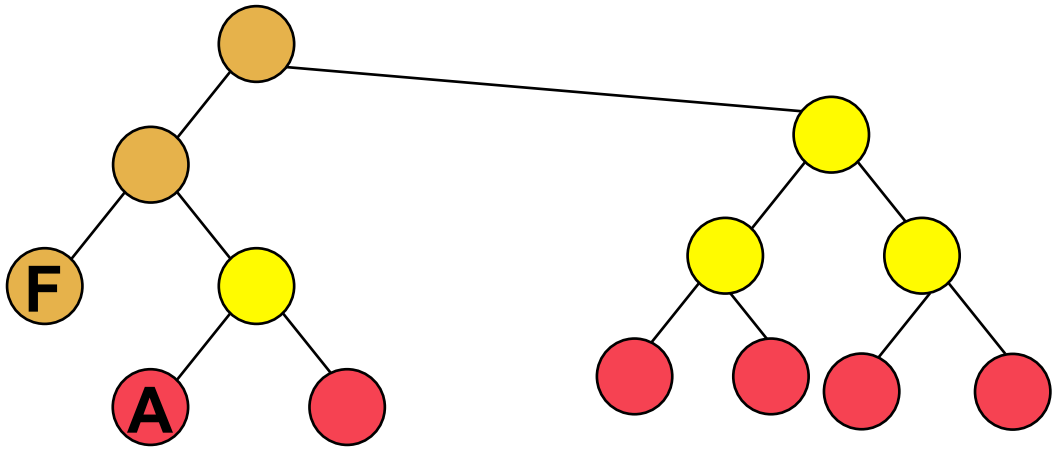
Vamos escolher por ordem Numérica!



O Deflate – Códigos de Huffman

- Construção dos códigos
  - Exemplo:
    - {A,B,C,D,E,F,G,H,…}
    - {3,3,3,3,3,2,4,4,…}

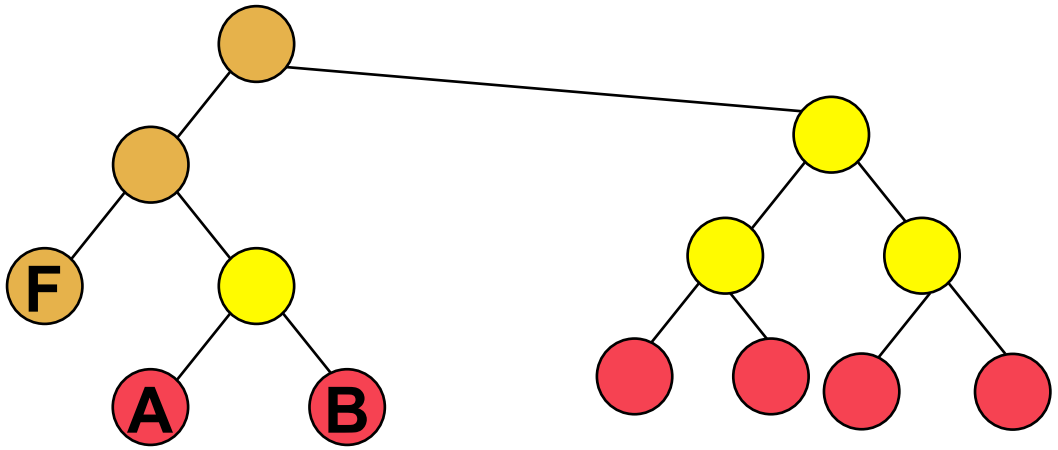
A=010



O Deflate – Códigos de Huffman

- Construção dos códigos
  - Exemplo:
    - {A,B,C,D,E,F,G,H,…}
    - {3,3,3,3,3,2,4,4,…}

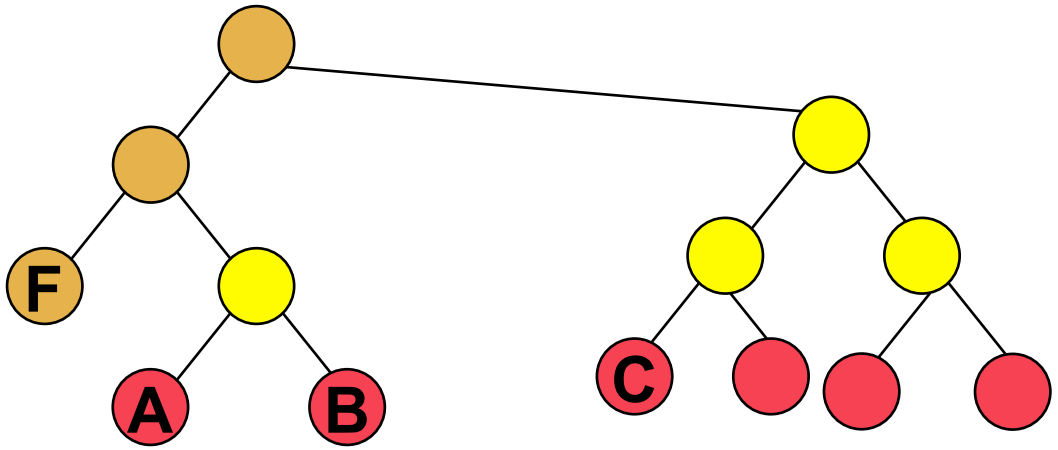
A=010
B=011



O Deflate – Códigos de Huffman

- Construção dos códigos
  - Exemplo:
    - {A,B,C,D,E,F,G,H,…}
    - {3,3,3,3,3,2,4,4,…}

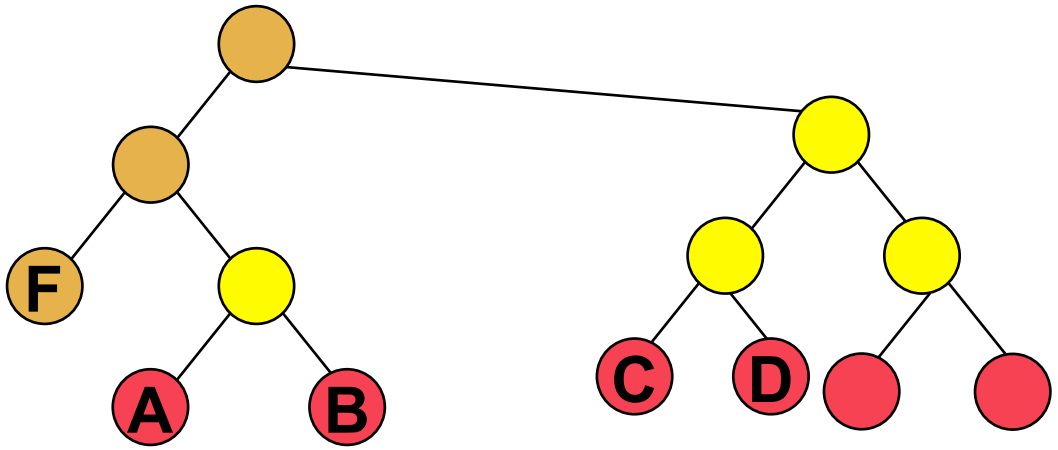
A=010
B=011
C=110



O Deflate – Códigos de Huffman

- Construção dos códigos
  - Exemplo:
    - {A,B,C,D,E,F,G,H,…}
    - {3,3,3,3,3,2,4,4,…}

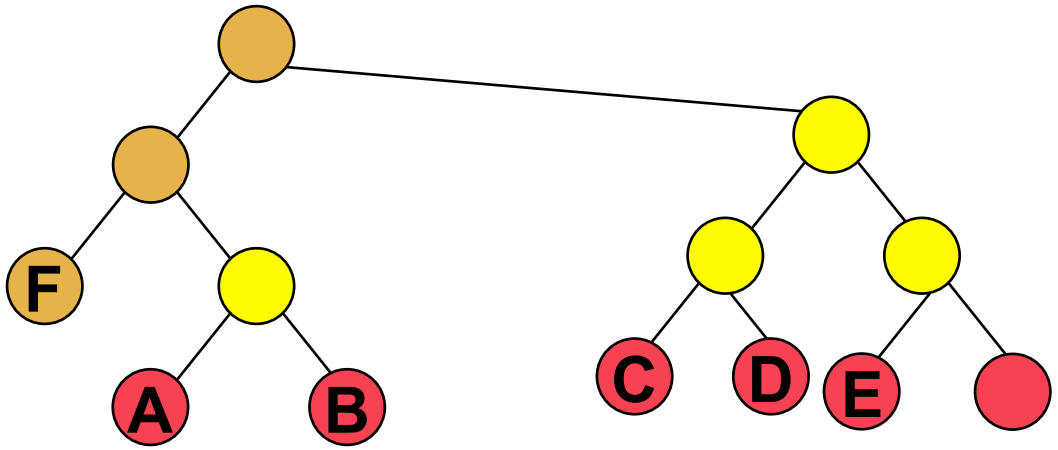
A=010
B=011
C=100
D=101



O Deflate – Códigos de Huffman

- Construção dos códigos
  - Exemplo:
    - {A,B,C,D,E,F,G,H,…}
    - {3,3,3,3,3,2,4,4,…}

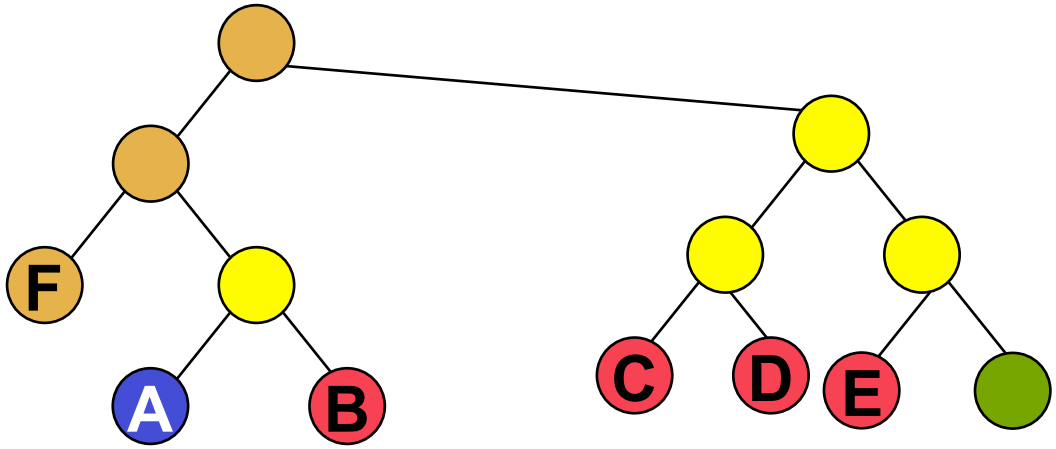
A=010
B=011
C=100
D=101
E=110



O Deflate – Códigos de Huffman

Primeiro nó com k bits Livre
Serve de prefixo para o próximo
Conjunto de códigos com k+1 bits

Esse código é base + Δ |0



O Deflate – Códigos de Huffman

- Code – código base
- bl_count [i] – número de símbolos a serem codificados com i bits

```
code = 0;
bl_count[0] = 0;
for (bits = 1; bits <= MAX_BITS; bits++) {
    code = (code + bl_count[bits-1]) << 1;
    next_code[bits] = code;
}
```

```
for (n = 0;  n <= max_code; n++) {
        len = tree[n].Len;
        if (len != 0) {
            tree[n].Code = next_code[len];
            next_code[len]++;
        }
    }
```

O Deflate – Códigos de Huffman

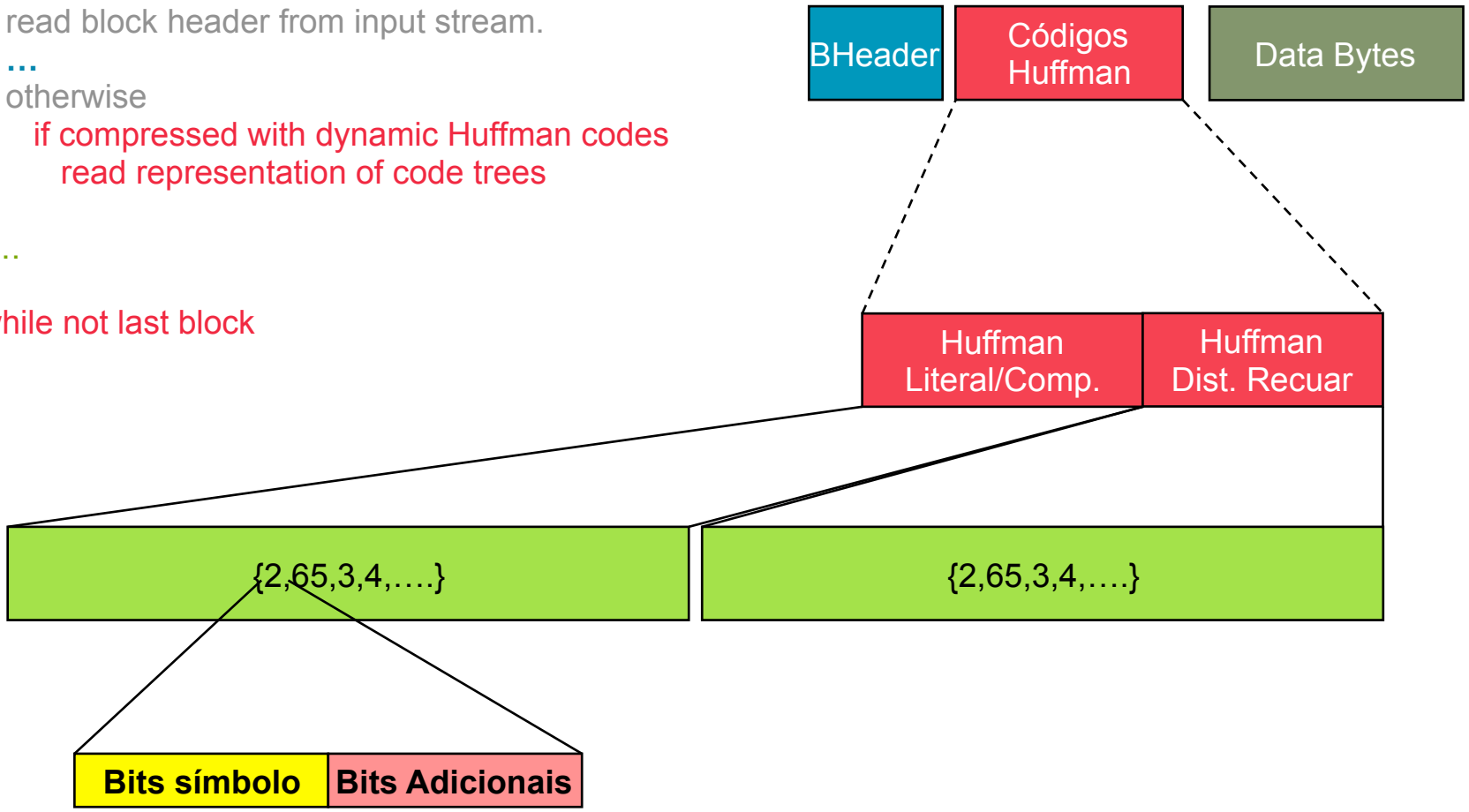
- And now Ladies and Gentleman the tricky part!

O Deflate - Algoritmo

- **do**
- read block header from input stream.
- **…**
- otherwise
- if compressed with dynamic Huffman codes
- read representation of code trees

- ….

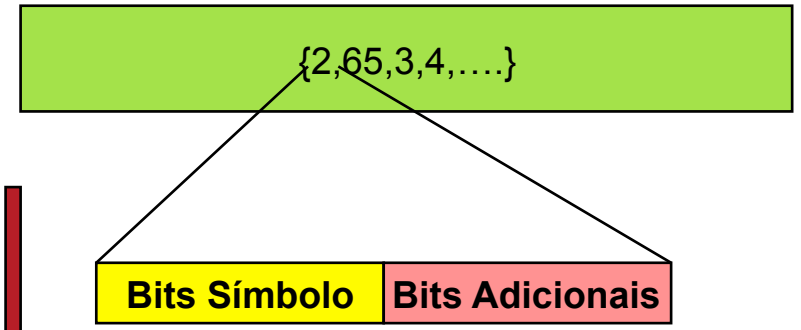
- while not last block



O Deflate - Algoritmo

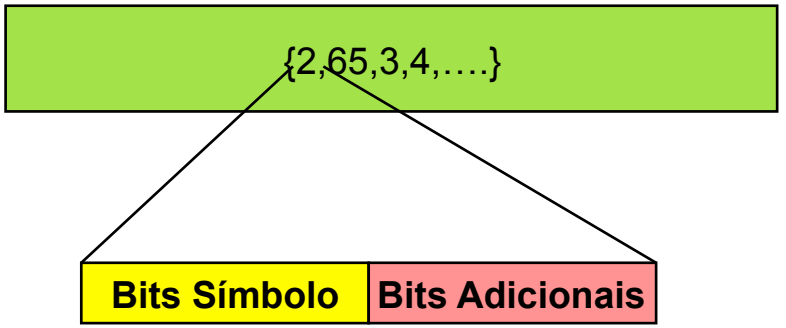
- Uma espécie de RLE

```
0 - 15: Represent code lengths of 0 - 15
    16: Copy the previous code length 3 - 6 times.
        The next 2 bits indicate repeat length
              (0 = 3, ... , 3 = 6)
           Example:  Codes 8, 16 (+2 bits 11),
                     16 (+2 bits 10) will expand to
                     12 code lengths of 8 (1 + 6 + 5)
    17: Repeat a code length of 0 for 3 - 10 times.
        (3 bits of length)
    18: Repeat a code length of 0 for 11 - 138 times
        (7 bits of length)
```



Estes códigos também estão Em Huffman

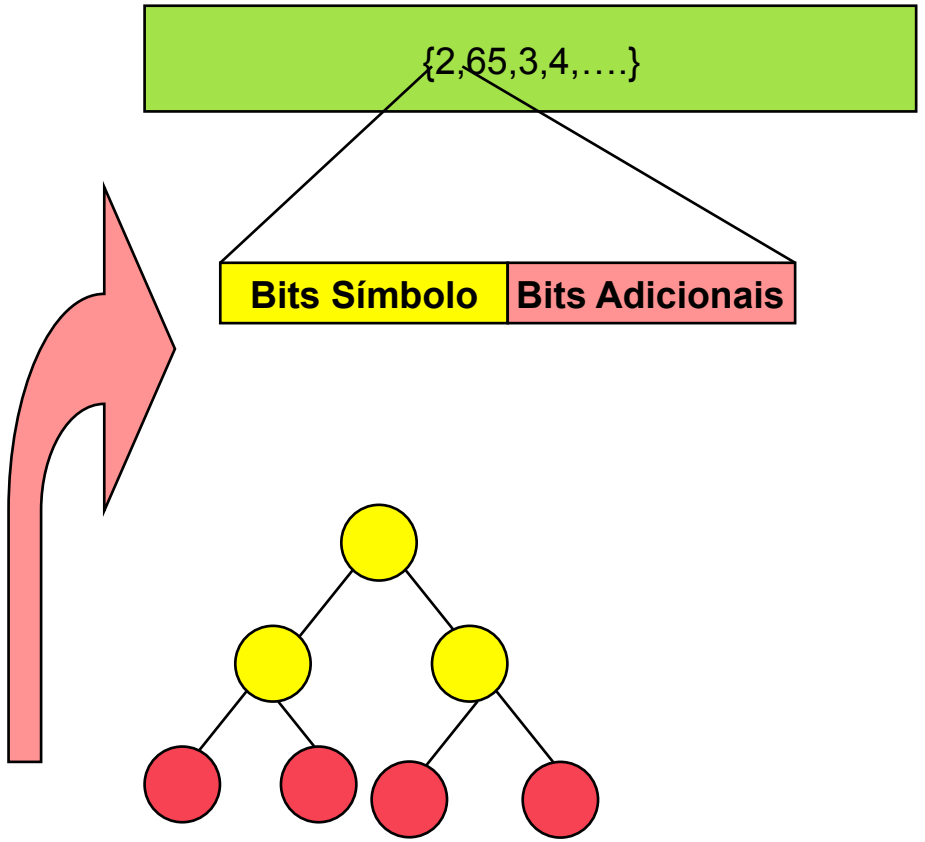
O Deflate - Algoritmo



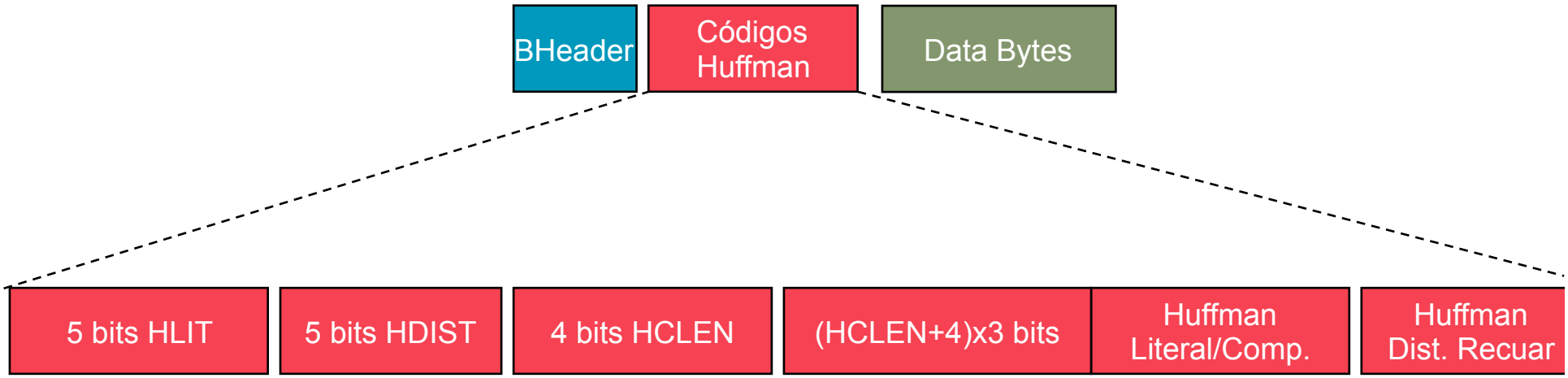
Still with me?

O Deflate - Algoritmo

Como construir estes códigos?



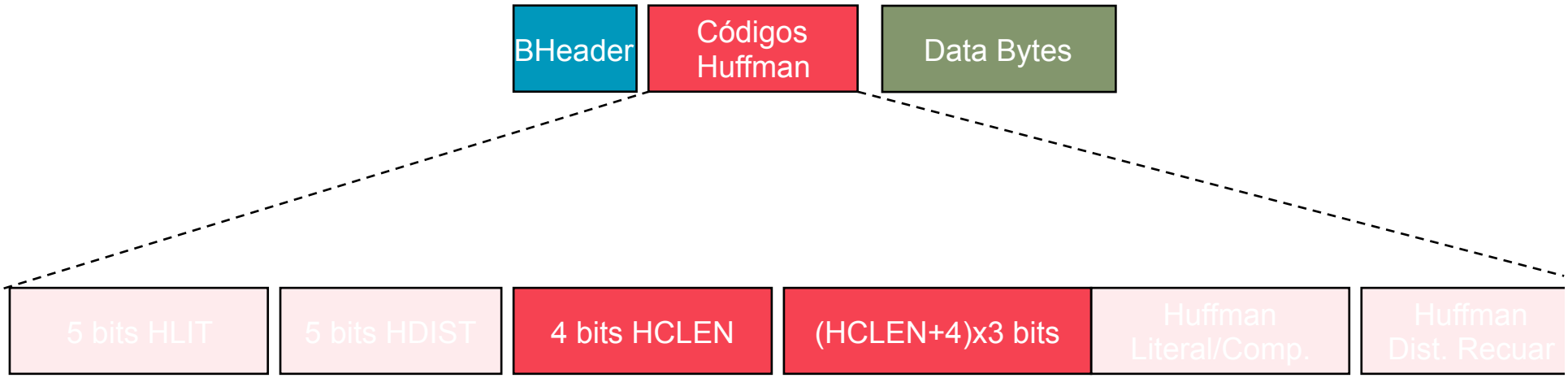
O Deflate - Algoritmo



- HLIT: # de códigos literais/comprimento – 257 (257 – 286)
- HDIST: # de códigos de distância -1 (1-32)
- HCLEN: # de códigos de “comprimento de código” – 4 (4 - 19)
- (HCLEN+4)x3: sequências de 3 bits com os comprimentos dos códigos do alfabeto do “comprimento de códigos” pela seguinte ordem:
  - 16, 17, 18, 0,8,7,9,6,10,5,11,4,12,3,13,2,14,1,15

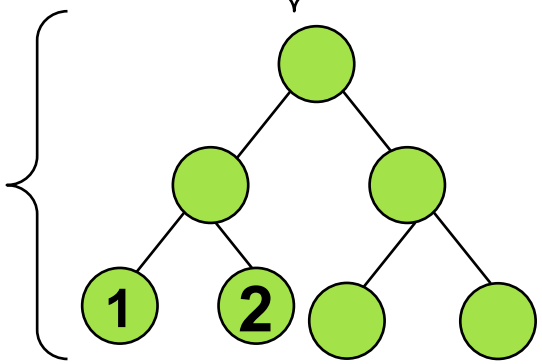
O Deflate – Construct Huffman

Primeiro



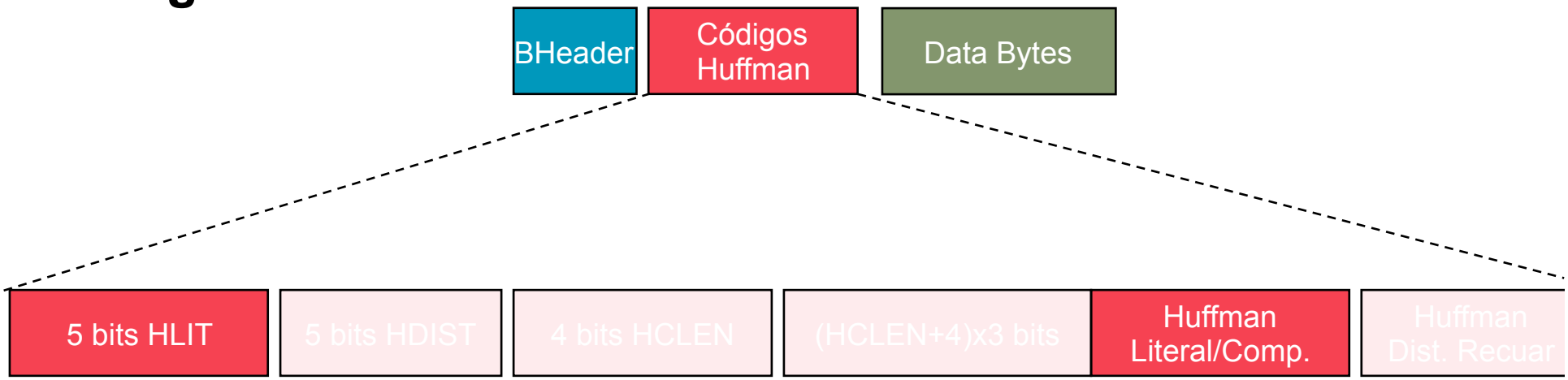
{2,2,4,4,7,…}

Códigos de comprimento de código

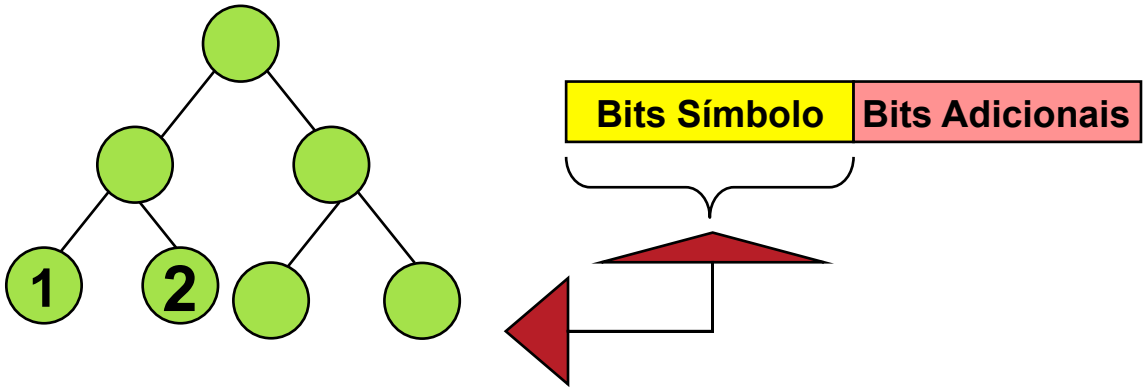


O Deflate – Construct Huffman

Segundo

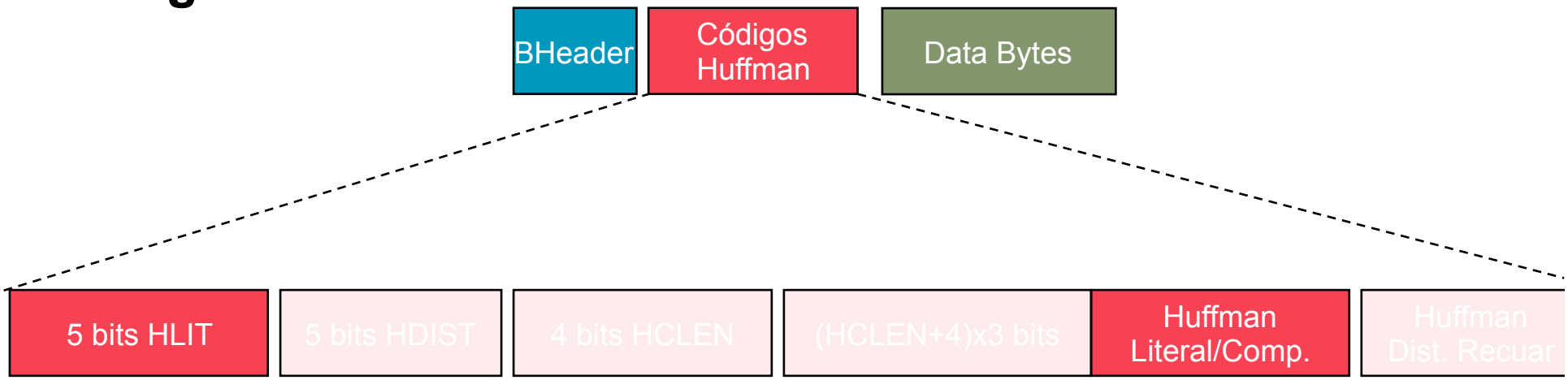


Códigos de "comprimento de código"

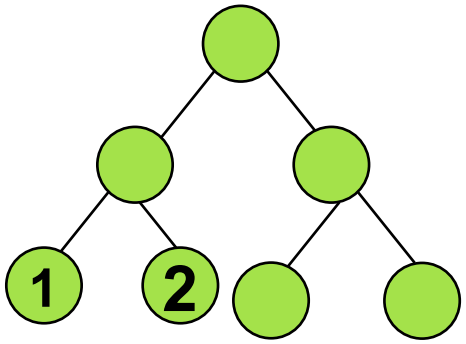


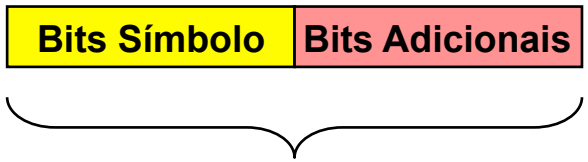
O Deflate – Construct Huffman

Segundo

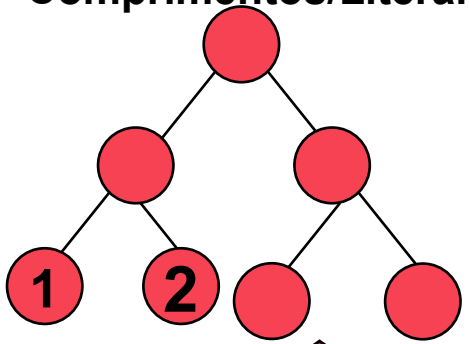


Códigos de "comprimento de código"





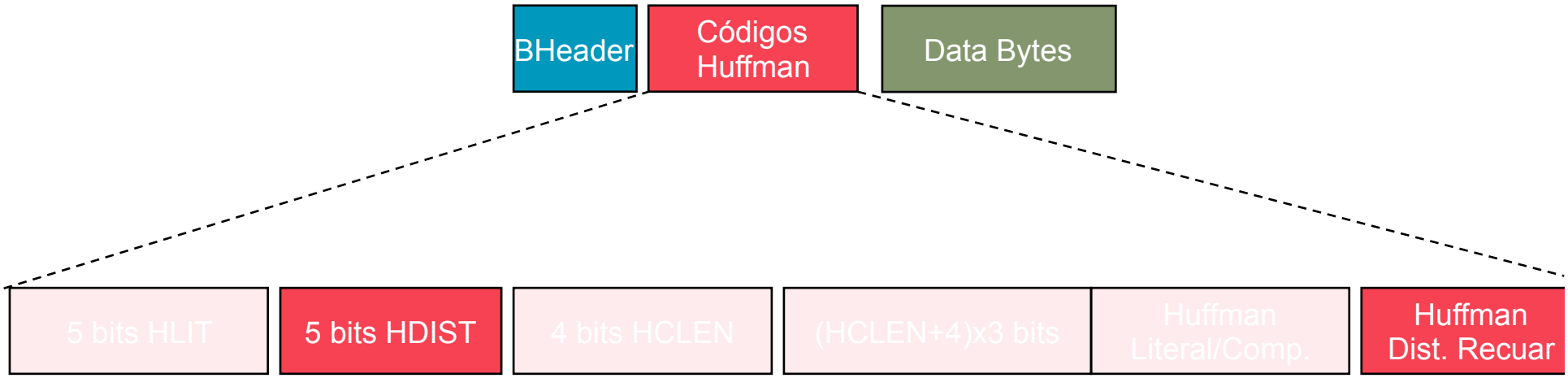
Códigos de Comprimentos/Literais



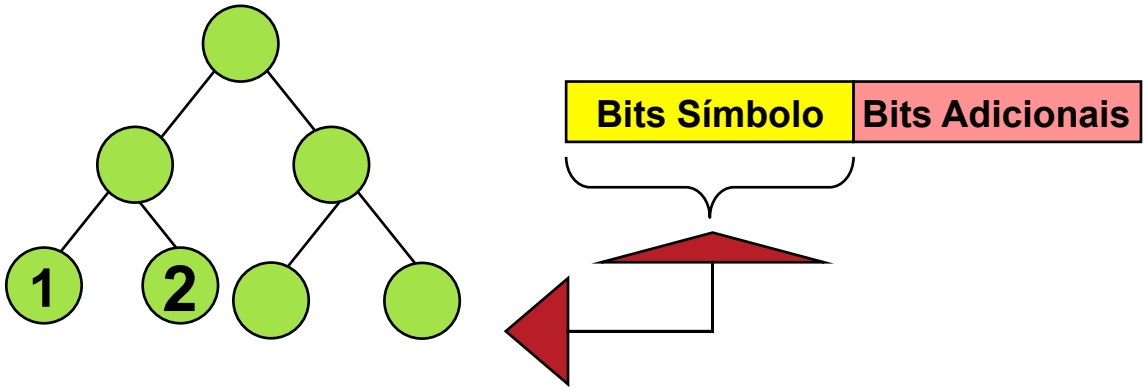
{1,0,0,21,3,4,4,4,5,2,…}

O Deflate – Construct Huffman

Terceiro

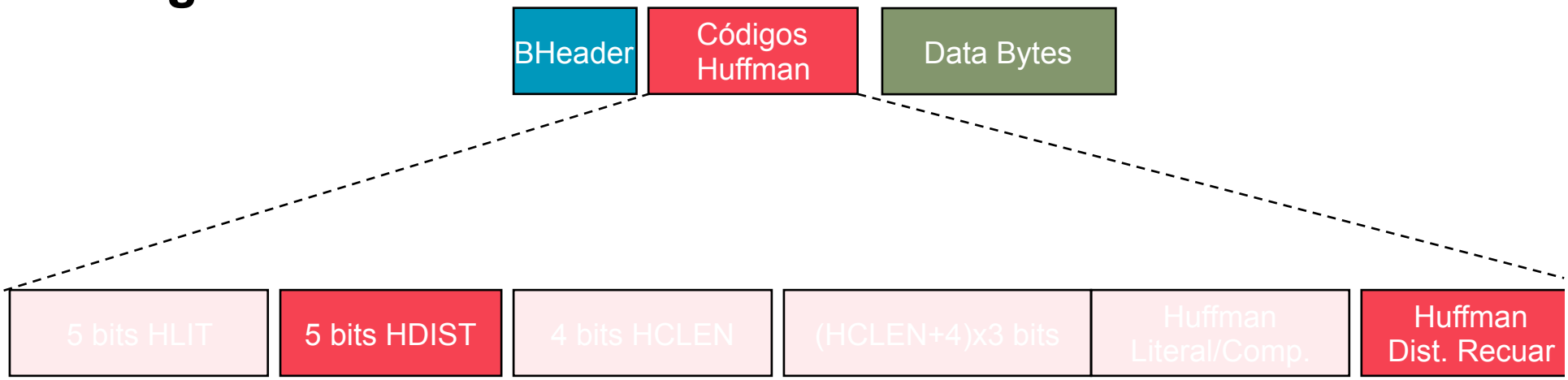


Códigos de "comprimento de código"

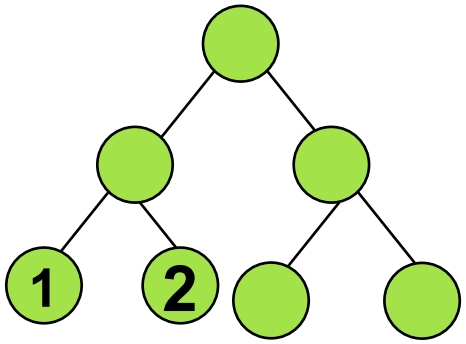


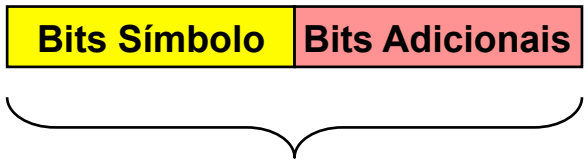
O Deflate – Construct Huffman

Segundo

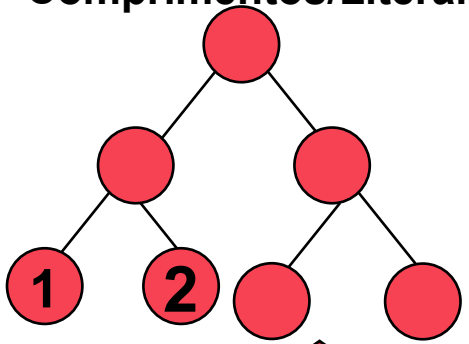


Códigos de "comprimento de código"





Códigos de Comprimentos/Literais

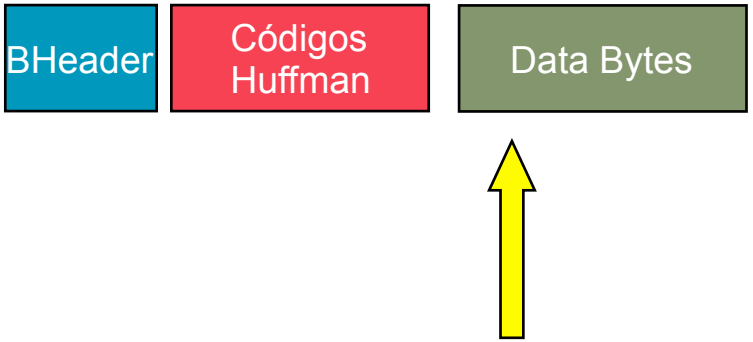


{1,0,0,21,3,4,4,4,5,2,…}

O Deflate - Algoritmo

```
do
   read block header from input stream.
   ...
   otherwise
      …
loop (until end of block code recognized)
        decode literal/length value from input stream
        if value < 256
           copy value (literal byte) to output stream
        otherwise
           if value = end of block (256)
              break from loop
           otherwise (value = 257..285)
              decode distance from input stream
              move backwards distance bytes in the output
              stream, and copy length bytes from this
              position to the output stream.
      end loop

while not last block
```



O Deflate - Algoritmo

- Mais informação
  - http://www.gzip.org/zlib/rfc-deflate.html

Let's do it in JAVA

- Programação ao Nível do Bit (em Java)
  - Operadores binários
    - Lógicos: and &, or |, not ~, xor ^
      - short a = 6, b = 4; //110, 100
        short c = a & b; // c = 4 → 100
    - Shift
      - Shift left: <<;
      - shift right: >>

        short d = c << 1; //→ d = c*2 = 8

Let's do it in JAVA

- Programação ao Nível do Bit (em Java)
  - É preferível utilizar-se notação hexadecimal em lugar da decimal.
    - Exemplo: representar o byte 1100 0100 → short a = 0xC4; //alternativamente: a = 196;
  - Os tipos de dados básicos em Java são ***signed***; deste modo, é mais seguro representarem-se bytes com recurso ao tipo ***short*** (16 bits), excepto em situações em que se tenha absoluto controlo sobre o o conteúdo do byte
    - Exemplo: byte a = 0x80; //se *unsigned* seria 128, mas como o tipo é *signed*, System.out.print(a) apresenta o valor -128