# Teoria da Informação

# Trabalho Prático nº 2

# Descompactação de Ficheiros 'gzip'

# Introdução

Período de execução: 6 aulas práticas

**Ritmo de execução esperado para avaliação:**

- Semana 1: 1 a 2
- Semana 2: 3
- Semana 3: 4
- Semana 4: 5
- Semana 5: 6,7
- Semana 6: 8

Prazo de Entrega: aula prática a seguir às 6 aulas

Esforço extra aulas previsto: 30 h / aluno

Linguagem de Programação: C, C++ ou Java

Objectivo: Pretende-se que o aluno adquira sensibilidade para as questões fundamentais relacionadas com codificação usando árvores de Huffman e dicionários LZ77.

# Trabalho Prático

Neste trabalho, pretende-se implementar o descodificador do algoritmo *deflate* (usado em ficheiros gzip). Em particular, será objectivo levar a cabo a descompactação de blocos comprimidos com **códigos de Huffman dinâmicos**. Todas as restantes situações de descodificação deverão ser ignoradas no âmbito do presente trabalho.

## A. Preparação

1. Leitura dos documentos de apoio ao trabalho prático:
   - **(Doc1)**: Slides fornecidos nas aulas teórico-práticas.
   - **(Doc2)**: Request for Comment (RFC) do deflate.
   - **(Doc3)**: RFC do cabeçalho do gzip.
   - **(Doc4)**: ficheiro *byteStream.txt* → ficheiro com a sequência de bytes após o cabeçalho do gzip (para o exemplo fornecido, FAQ.txt.gz).
   - **(Doc5)**: ficheiro *Códigos.xls* → resultados esperados para os códigos de Huffman a obter nas várias etapas do algoritmo, para o exemplo FAQ.txt.gz
2. O seguinte código fonte é-lhe fornecido como base de trabalho. Poderá utilizá-lo, caso considere pertinente. Nesse caso, deverá estudar as funcionalidades implementadas. Nota: código fonte em C/C++. Recomenda-se a utilização da plataforma de desenvolvimento **Bloodshed Dev-C++**.

   a) Ficheiro **gzip.cpp**: classe principal para descompactação de um ficheiro no formato gzip:
      - Linha de comando: gzip <nome.gz>
      - Estruturas principais (definidas no ficheiro **gzip.h**)
        - **gzipHeader**: estrutura relativa ao cabeçalho do ficheiro .gz

```
typedef struct header
{
        //elementos fixos
        unsigned char ID1, ID2, CM, XFL, OS;
        unsigned long MTIME;
        unsigned char FLG_FTEXT, FLG_FHCRC, FLG_FEXTRA,
                FLG_FNAME, FLG_FCOMMENT;   //bits 0, 1, 2, 3 e 4,
                respectivamente (restantes 3: reservados)

        // FLG_FTEXT --> ignorado deliberadamente (tipicamente igual a
0)
        //se FLG_FEXTRA == 1
        unsigned char xlen;
        unsigned char *extraField;

        //se FLG_FNAME == 1
```

```
        char *fName;  //terminada por um byte a 0

        //se FLG_FCOMMENT == 1
        char *fComment; //terminada por um byte a 0

        //se FLG_HCRC == 1
        unsigned char *HCRC;
} gzipHeader;
```

Observação: no presente trabalho não necessita dos dados do cabeçalho, à excepção do nome do ficheiro original (campo fName).

- Campos principais da estrutura (ver explicações no **Doc3**):
  - *unsigned char ID1, ID2, CM, FLG, XFL, OS*;
  - *unsigned long MTIME*;
  - *unsigned char FLG_FTEXT, FLG_FHCRC, FLG_FEXTRA, FLG_FNAME, FLG_FCOMMENT;*
  - *char * fName*: nome do ficheiro original (antes da compactação);

- Variáveis principais definidas na função main:
  - *gzipHeader gzh*: estrutura que armazena a informação disponibilizada no cabeçalho do ficheiro gz (ver ficheiro gzip.h, abaixo);
  - *char *filename*: nome do ficheiro a descompactar;
  - *FILE *gzFile*: ponteiro para o ficheiro a descompactar;
  - *long fileSize*: número de bytes do ficheiro .gz;
  - *int numBlocks*: número de blocos contidos no ficheiro (a determinar).
- Funções principais:
  - *int getHeader(FILE *gzFile, gzipHeader *gzh)*: lê o cabeçalho do ficheiro gzFile para o campo gzh ; devolve -1 em caso de erro (formato inválido) ou 0 se tudo normal; (Nota: a organização do cabeçalho de ficheiros gzip pode ser consultada no **Doc3**);

- *int isDynamicHuffman(unsigned char rb)*: analisa os 2 bits menos significativos do byte *rb* e verifica se o bloco foi comprimido com códigos de Huffman dinâmicos; devolve 1 em caso positivo e 0 em caso negativo;
- *long getOrigFileSize(FILE * gzFile)*: lê os 4 bytes finais do ficheiro .gz, correspondentes ao tamanho do ficheiro original não compactado, e devolve este valor;
- *void bits2String(char *strBits, unsigned char byte)*: método auxiliar de debugging o qual converte o byte *byte* para a string *strBits* correspondente à sua representação binária;
- *int main(int argc, char** argv)*: função principal
  - argumentos passados na linha de comando
  - Contém a programação necessária à leitura do cabeçalho do ficheiro .gz, bem como a análise do block header; de notar que os valores de BFINAL e BTYPE estão já determinados.

b) Ficheiro **Huffman.cpp**: conjunto de funções para criação, acesso e gestão de árvores de Huffman:

Observação: o ficheiro auxiliar **TesteHuffman.cpp** (ver abaixo) contém algumas exemplos de utilização das funções para manipulação de árvores de Huffman (inserção de um dado código na árvore, pesquisa de um código na árvore, ...); tal como se referiu, o ficheiro TesteHuffman.cpp é um ficheiro auxiliar, de modo que não deverá ser incluído no projecto;

- Estruturas principais (definidas no ficheiro **Huffman.h**):
  - **HFNode**: contém informação relativa a um nó da árvore de Huffman

```
typedef struct hfnode
```

```
{
    short index;  //se folha, guarda posição no alfabeto; senão, -1;
    short level; // nível do nó na árvore
    struct hfnode *left, *right;  //referências para os filhos direito e esquerdo:
        é folha se ambos forem NULL
} HFNode;
```

- Campos da estrutura
  - *short index*: guarda posição do nó no alfabeto, caso seja folha; -1 caso contrário;
  - *short level*: nível na árvore em que se encontra o nó actual;
  - struct hfnode **left, *right*:, referências para os filhos esquerdo e direito do nó actual;

- **HuffmanTree**: define uma árvore de Huffman

```
typedef struct huffmantree
{
    HFNode *root, *curNode;  //raíz da árvore e nó actual
} HuffmanTree;
```

- Campos da estrutura
  - *HFNode *root*: raiz da árvore;
  - *HFNode *curNode*: nó actual da árvore;

- Funções principais:
  - *HuffmanTree* createHFTree()*: cria uma árvore de Huffman vazia; devolve um ponteiro para uma *struct* HuffmanTree;
  - *int addNode(HuffmanTree *hft, char *s, int ind, short verbose)*: adiciona nó à árvore:
    - recebe um ponteiro para a árvore em causa (*HuffmanTree *hft*), uma string (*char *s*) com o código (sequência de 0s e 1s), o índice no alfabeto (*int ind*)

e um campo 'verbose' para visualização ou não de resultados;

- devolve -1 se o nó já existe; -2 se a inserção implicasse que o código deixasse de ser de prefixo; 0 se adicionou bem;

- *int nextNode(HuffmanTree *hft, char c)*: *:* actualiza o nó corrente na árvore com base no nó actual (curNode da árvore hft) e no próximo bit (*char c*):
  - recebe um caracter *c* com valor '0' ou '1';
  - devolve -1 se não encontrou o nó; -2 se encontrou mas não é folha; o índice no alfabeto se é folha.
  - **É esta a função a utilizar na pesquisa bit a bit, tal como decorre da leitura de bits referentes a códigos de Huffman**
- *int findNode(HuffmanTree *hft, char *s, HFNode *cur, short verbose)*: procura código na árvore, a partir de um nó especificado:
  - recebe um ponteiro para a árvore de Huffman, uma string com o código (sequência de 0s e 1s), o nó a partir do qual a pesquisa deve ser efectuada e um campo 'verbose'
  - devolve -1 se não encontrou; -2 se é prefixo de código existente; índice no alfabeto se encontrou;
- *int findNode(HuffmanTree *hft, char* s, short verbose)*: procura código na árvore, a partir da raiz:
  - chama o método anterior com cur = root;
  - Nota: esta função e a anterior procuram um código completo (e não de forma iterativa), pelo que, na prática, não deverão ser utilizadas;
- *short isLeaf(HFNode *n)*: verifica se o nó *n* é folha;
- *void resetCurNode (HuffmanTree *hft)*: reposiciona curNode na raiz da árvore.

c) Ficheiro **TesteHuffman.cpp**: contém exemplos de utilização de árvores de Huffman:
   - Apenas contém a função main com alguns exemplos.

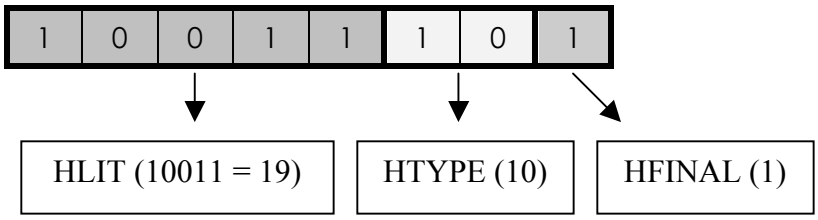
## B. Implementação do descompactador:

Notas:

- As alíneas seguintes são apenas *sugestões* de implementação. Poderá seguir outra estratégia que considere mais adequada.
- Todas as funções desenvolvidas, assim como partes do código particularmente complexas, deverão estar **comentadas** de forma compreensível.
- É apresentada uma proposta de planeamento temporal, ao longo das 4 semanas do projecto.

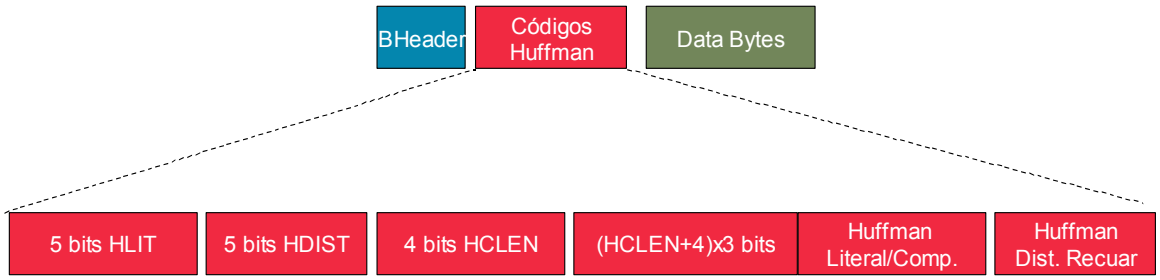
- **Importante! Ordem dos bits nos bytes**:
  - No *deflate*, a sequência de bits correspondente a códigos de Huffman é ordenada no byte começando com o bit *mais* significativo.
    - Exemplo: byte '**01101**011', em que os bits em negrito correspondem a um código de Huffman:
      - Atendendo à regra acima, e lendo bit a bit, o código será 10110 (i.e., pela ordem inversa).
  - Em elementos que não sejam códigos de Huffman, os bits são ordenados no byte começando com o bit *menos* significativo.
    - Exemplo: determinação dos comprimentos dos códigos do alfabeto de comprimentos de códigos, em que um dado byte tem a informação '00**100**010' (os bits em negrito denotam os 3 bits do comprimento a ler);
    - Com base nesta regra, o comprimento será 100 = 4 bits.

- Num byte que contenha vários elementos, os mesmos são armazenados da "direita para esquerda no byte"
  - Exemplo: block header




**1ª Semana**

1. Crie um método que leia o formato do bloco (i.e., devolva o valor correspondente a HLIT, HDIST e HCLEN), de acordo com a estrutura de cada bloco, apresentada na figura seguinte:




2. Crie um método que armazene num array os comprimentos dos códigos do "alfabeto de comprimentos de códigos", com base em HCLEN:
   - Tenha em atenção que as sequências de 3 bits a ler correspondem à ordem 16, 17, 18, 0, 8, 7, 9, 6, 10, 5, 11, 4, 12, 3, 13, 2, 14, 1, 15 no array de códigos (ver Doc1, Doc2; resultados a obter: Doc5)

**2ª Semana**

3. Crie um método que converta os comprimentos dos códigos da alínea anterior em códigos de Huffman do "alfabeto de comprimentos de códigos" (ver Doc5);

**3ª Semana**

4. Crie um método que leia e armazene num array os HLIT + 257 comprimentos dos códigos referentes ao alfabeto de literais/comprimentos, codificados segundo o código de Huffman de comprimentos de códigos (ver Doc5):
   - Recorra às funções do ficheiro Huffman.cpp, nomeadamente às funções de pesquisa de códigos de forma sequencial (i.e., bit a bit);
   - Tenha em atenção que alguns códigos requerem a leitura de alguns bits extra (nomeadamente os índices 16, 17 e 18 do alfabeto);

**4ª Semana**

5. Crie um método que leia e armazene num array os HDIST + 1 comprimentos de código referentes ao alfabeto de literais/comprimentos, codificados segundo o código de Huffman de comprimentos de códigos (ver Doc5):
   - Recorra às funções do ficheiro Huffman.cpp, nomeadamente às funções de pesquisa de códigos de forma sequencial (i.e., bit a bit);
   - Tenha em atenção que alguns códigos requerem a leitura de alguns bits extra (nomeadamente os índices 16, 17 e 18 do alfabeto);

**5ª Semana**

6. Usando o método do ponto 3), determine os códigos de Huffman referentes aos dois alfabetos (literais / comprimentos e distâncias) e armazene-os num array (ver Doc5).

7. Crie as funções necessárias à descompactação dos dados comprimidos, com base nos códigos de Huffman e no algoritmo LZ77 (ver Doc1 e Doc2).
   - Recorra funções do ficheiro Huffman.cpp, nomeadamente às funções de pesquisa de códigos de forma sequencial (i.e., bit a bit);
   - Tenha em atenção que alguns códigos requerem a leitura de alguns bits extra (por exemplo os índices 265 a 285 no alfabeto de literais/comprimentos ou os índices 4 a 29 no alfabeto de distâncias);

**6ª Semana**

8. Grave os dados descompactados num ficheiro com o nome original (consulte a estrutura *gzipHeader*, nomeadamente o campo *fName* e analize a função *getHeader* do ficheiro gzip.cpp).