PMR3201 - Computação para Automação
Exercício Programa 1 - 2022

Compactação e descompactação de dados utilizando a codificação de Huffman

Prof. Thiago de Castro Martins
Prof. Marcos de Sales Guerra Tsuzuki
Prof. Newton Maruyama
Profa. Larissa Driemeier

Deadline: 27/06/2022 - 23h59min

1 Introdução

Os algoritmos de compactação de dados permitem a redução do número de *bytes* que representam uma determinada informação. Para um texto, por exemplo, ao invés do uso de caracteres representados através do código ASCII de 7 *bits* (na prática a menor informação corresponde a um *byte*) podemos utilizar um número menor de *bits*.

Suponha que desejamos codificar a *string* `ABRACADABRA!`, utilizando o código ASCII essa *string* pode ser interpretada através da seguinte cadeia de *bits*:

1000001100001010100101000001100001110000011000100100000110000101010010100000010100001

Para decodificar essa cadeia de *bits* deve ser realizado a leitura de 7 bits a cada vez e consultar a tabela *ASCII* para encontrar o caracter correspondente.

Para a *string* em questão temos:

Dec	Hex	Bin	Símbolo
65	41	1000001	A
66	42	1000010	B
67	43	1000011	C
68	44	1000100	D
82	52	1010010	R
33	21	0100001	!

A idéia fundamental em compactação de dados é a utilização de um número menor de *bits* para codificar a mesma informação. Uma possível solução seria a utilização de um número de *bits* variável para codificar os caracteres. Caracteres com maior frequência devem possuir uma representação com um número menor de *bits*.

Por exemplo, poderíamos codificar A com 0, B com 1, R com 00, C com 01, D com 10 e ! com 11, dessa forma `ABRACADABRA!` seria representado como `0 1 00 0 01 0 10 0 1 00 0 11`. Essa representação utiliza 17 *bits* ao invés dos 77 *bits* utilizando código *ASCII*. Entretanto essa representação necessita da inserção de espaços para delimitação. Sem os espaços a cadeia de *bits* seria: 01000010100100011, não sendo possível sua decodificação.

Na codificação proposta acima o código para o caracter `A`, `0`, é também o prefixo do código `00` para o caracter `R`, por isso a necessidade de espaços para delimitação. Necessitamos portanto de uma codificação livre de prefixo *(prefix free code)*, isto é, que não haja coincidência de caracteres no início do código da representação de caracteres diferentes. Dessa forma, não seria necessário a utilização de delimitadores.

Uma possível representação de códigos livre de prefixo é por meio do uso de árvores de busca binária.

Nesse caso, as folhas da árvore (nós terminais) representam cada um dos caracteres a serem codificados. O código de cada caracter é interpretado através do caminho percorrido até a folha. Ao se movimentar em profundidade, para cada mudança para o lado esquerdo utiliza-se o código `0` e para o lado direito utiliza-se o código `1`.

A figura 1 ilustra a representação em árvore de duas possíveis codificações para `ABRACADABRA!` uma com 29 *bits* e outra com 30 *bits*.

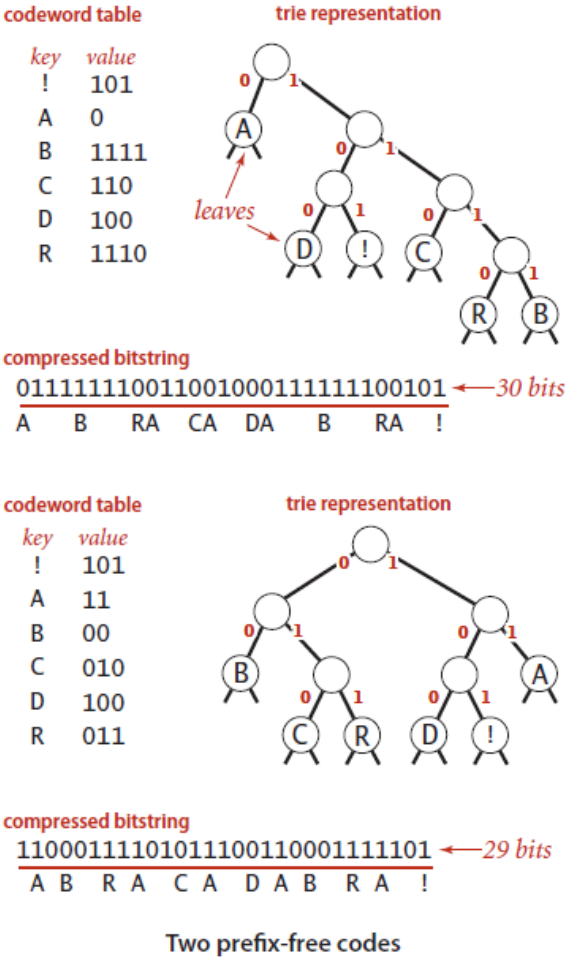

Figura 1: Duas possíveis codificações livre de prefixo para ABRACADABRA! (Extraído de Sedgewick and Wayne pg. 827).

Se o objetivo é a compactação de dados deveríamos estar interessados numa codificação ótima, i.e., uma codificação que leve sempre ao menor número de *bits*.

A seguir apresenta-se a codificação de Huffman que sempre leva a uma codificação ótima.

2 Codificação de Huffman

O algoritmo de codificação de Huffman utiliza uma árvore de busca binária cuja construção depende do número de ocorrências α de cada caracter presente em um determinado texto. Dessa forma, a codificação é sempre diferente para cada texto.

Por exemplo, vamos considerar a seguinte frase:

```
it was the best of times it was the worst of times
```

Na prática deveríamos enxergar os espaços e caracteres de controle como LF (Line Feed):

```
itSPwasSPtheSPbestSPofSPtimesSPitSPwasSPtheSPworstSPofSPtimesLF
```

Inicialmente é necessário ler o texto por completo e calcular o número de ocorrências α de cada caracter incluindo espaços e caracteres de controle.

Cada caracter presente no texto será representado através de um nó terminal. O nó terminal conterá a identificação do caracter e o número de ocorrências α associado.

Inicialmente os nós são criados e colocados em ordem crescente de acordo com o número de ocorrências α como ilustrado na figura 2. Dentro do contexto da implementação isso pode ser realizado através de uma estrutura do tipo fila de prioridades.

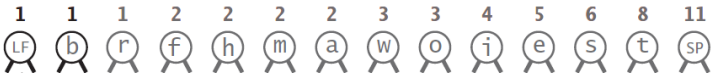

Figura 2: Candidatos a nós terminais ordenados de acordo com o número de ocorrências α de cada caracter (Extraído de Sedgewick and Wayne pg. 831).

A construção da árvore é realizada de maneira *Bottom-Up*. Cria-se um nó onde os nós filhos esquerdo e direito são os dois nós terminais com menor número de ocorrências. No caso os nós relativos aos caracteres `LF` e `b` que possuem número de ocorrências $\alpha(\texttt{LF}) = \alpha(\texttt{b}) = 1$. Esse novo nó não terminal passa a ter o número de ocorrências igual a $\alpha(\texttt{LF} + \texttt{b}) = 2$ o que equivale a soma do número de ocorrências dos filhos e é colocado na fila de prioridades como ilustrado na figura 3.

Figura 3: (Extraído de Sedgewick and Wayne pg. 831.)

Novamente, um novo nó é criado onde os dois nós filhos esquerdo e direito devem ser os nós com o menor número de ocorrências que estão na fila de prioridades, ou seja, o nó relativo ao caracter `r` com $\alpha(r) = 1$ e o nó composto anteriormente com $\alpha(\texttt{LF} + \texttt{b}) = 2$. O número de occorrências para esse nó é calculado como $\alpha(\texttt{r} + \texttt{LF} + \texttt{b}) = 3$. O novo nó também é colocado na fila de prioridades. Os dois primeiros passos indicando a fila de prioridades com a árvore parcial está ilustrado na figura 4.

Figura 4: (Extraído de Sedgewick and Wayne pg. 831).

O processo chega ao final quando não restam mais nós na fila de prioridades. As etapas completas da construção da árvore de busca binária estão ilustradas na figura 5.

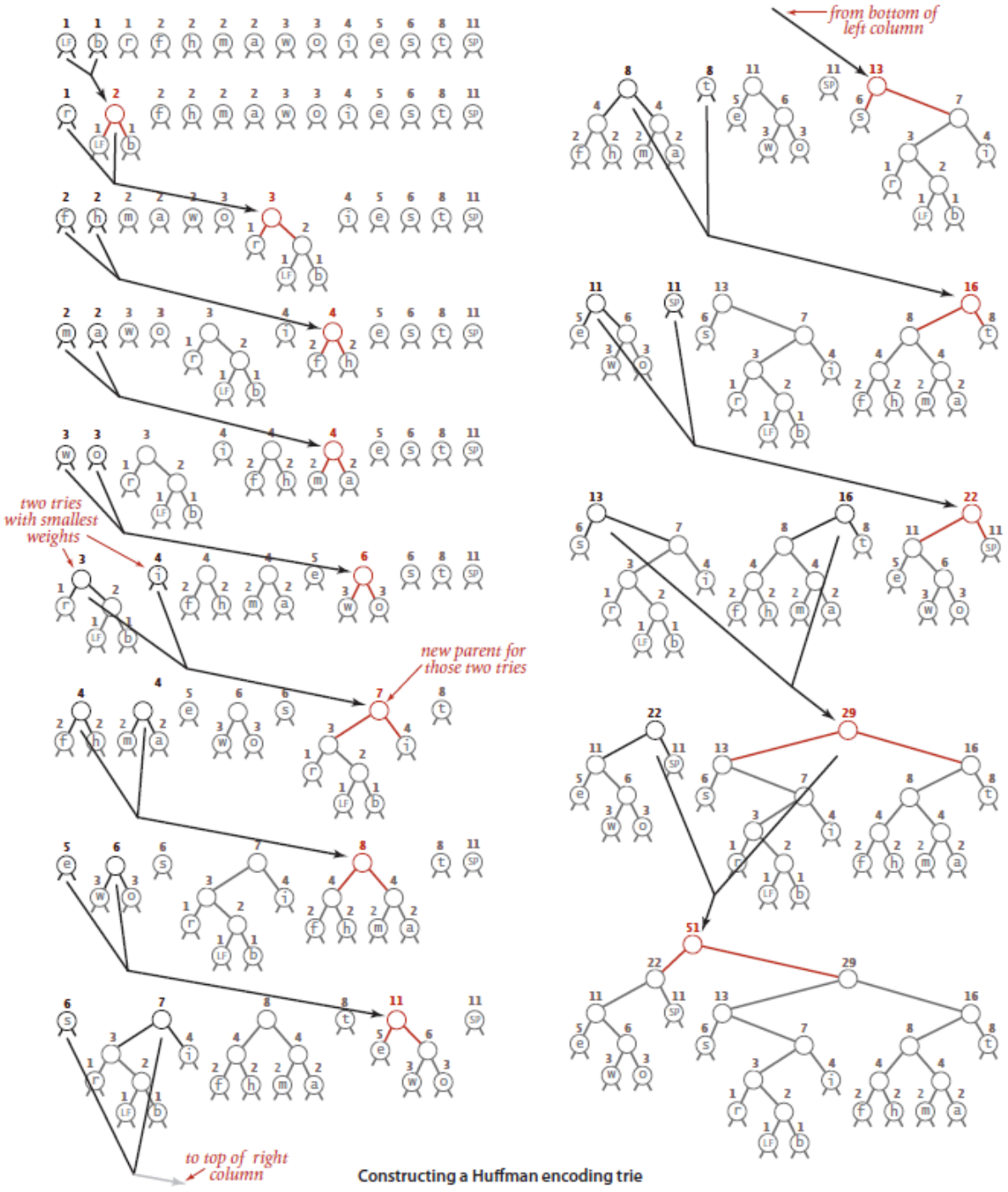

Figura 5: Etapas completas da construção da árvore de codificação de Huffman (Extraído de Sedgewick and Wayne pag. 831).

Com a árvore binária construída os códigos relativos a cada caracter são definidos pelo caminho do nó raiz até o nó terminal correspondente. Uma possível convenção é a geração de um bit 0 ao código ao descer pelo nó filho esquerdo e a geração de um bit 1 caso contrário. Pode-se gerar dessa forma uma tabela de códigos contendo cada caracter e o código associado.

A árvore de busca binária completa e a tabela de códigos é ilustrada na figura 6.

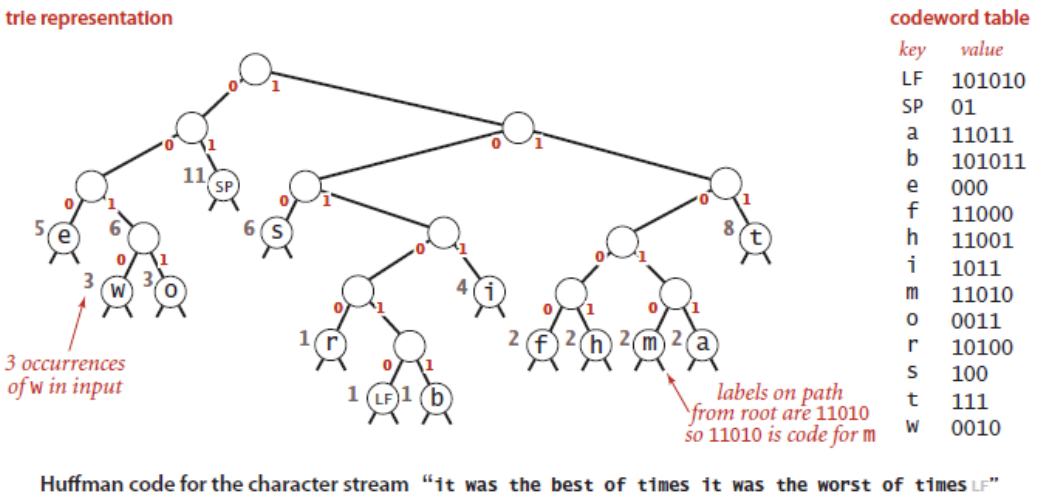

Figura 6: Árvore de codificação e tabela de códigos associada (Extraído de Sedgewick and Wayne pg. 832).

Através da tabela de códigos é possível converter a frase descrita pela cadeia de caracteres:

itSPwasSPtheSPbestSPofSPtimesSPitSPwasSPtheSPworstSPofSPtimesLF

101111101001011011100011111100100000110101100- 010011101001111000011111011110100001000110 11-
111010010110111000111111001000010010001110 10- 010011101001111000011111011110100001001010 10

A representação compactada utiliza 176 bits ao invés dos 408 bits utilizando o código ASCII de 8 bits.

Podemos agora escrever um arquivo com essa informação compactada e enviar para uma outra pessoa via e-mail ?

Quem recebe o arquivo deve descompactar o arquivo o que equivale a decodificar a informação transformando-a no texto original.

De alguma forma, a tabela de códigos deve ser inserida no próprio arquivo junto com a informação compactada.

A tabela de códigos usualmente é inserida no início do arquivo utilizando as informações que descrevem a árvore de busca binária que contém o código de Huffman.

Utiliza-se um atravessamento em pré-ordem (preorder traversal) escrevendo bit 0 quando um nó intermediário é visitado e escrevendo o bit 1 quando um nó terminal é atingido. Após o bit 1 o código ASCII do caracter associado deve ser escrito. A figura 7 ilustra esse processo.

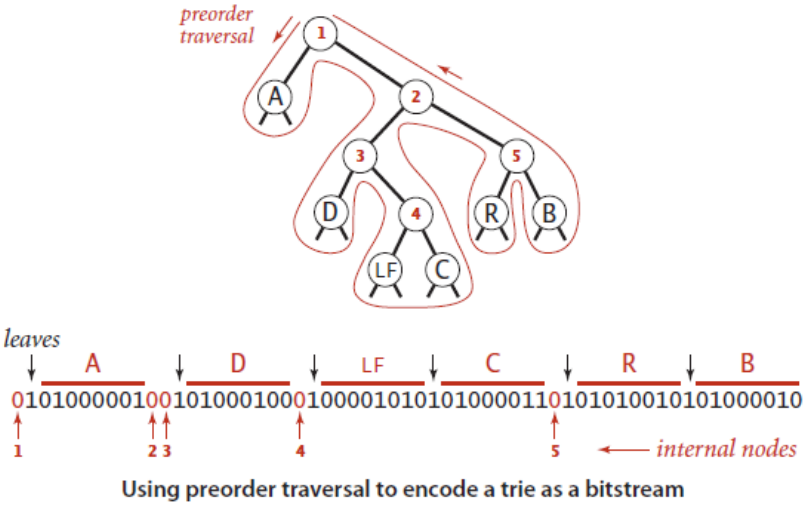

Figura 7: Transformação da árvore de codificação para a forma linear realizando atravessamento em pré-ordem (Extraído de Sedgewick and Wayne pag. 834).

A processo de decodificação começa pela leitura, no arquivo, da informação que codifica a árvore de busca binária. Posteriormente, a árvore de busca binária deve ser construída. Em seguida faz-se uma leitura do código *bit* a *bit* utilizando os *bits* 0s e 1s para atravessar a árvore até encontrar um nó terminal que está associado a um caracter específico. Quando encontra-se um nó terminal o próximo *bit* se refere a um outro caracter, ou seja, deve ser iniciado um novo atravessamento a partir da raiz da árvore.

3 Diretivas para implementação

3.1 Resumo do algoritmo de codificação de Huffman

Podemos resumir o algoritmo de codificação de Huffman através dos seguintes passos:

1. Leitura do arquivo texto `original.txt`.
2. Geração de estatísticas. Criar tabela onde cada linha está associado a um caracter e o número de ocorrências α associado.
3. Criação de nós para cada um dos caracteres.
4. Inserção dos nós numa fila de prioridades.
5. Criação da árvore de binária que contém as informações de codificação.
6. Construção da tabela de codificação onde cada linha se refere a uma caracter e o seu código de compactação.
7. Geração dos bytes que representam a informação compactada.
8. Geração da árvore escrita sob a forma linear.
9. Geração do arquivo compactado `orignal.huf`.

3.2 Implementação da árvore de codificação

A implementação da árvore de codificação deve ser feita através de duas classes básicas aqui denominadas `Node` e `BST`.

A seguir apresenta-se um exemplo **REDUZIDO** (arquivo `teste3.py`) para implementação dessas duas classes. O aluno deve completar a implementação com outras funções que seja necessárias.

O programa `main` ilustra a construção de uma árvore com 5 nós. Note que o método `preordertraverse()` é uma função recursiva que atravessa a árvore em pré-ordem imprimindo '0' para os nós não terminais e '1' para os nós terminais.

Note que na classe `Node` foi definido a função `__lt__` que estabelece a definição da hierarquia em ordem crescente. Essa função significa *less than* o que é equivalente ao operador de comparação '<'. A redefinição da função `__lt__` utilizando o operador '<=' é necessária para que a fila de prioridades (que será explicada em seguida) admita nós com o mesmo número de ocorrências.

```
class Node: # Classe que define um no da arvore binaria
def __init__(self, car=None,freq=None,left=None,right=None):
self.caracter = car # caracter caso seja um no terminal
self.frequencia = freq # numero de ocorrencias do caracter
self.left = left    # ponteiro filho da esquerda
self.right = right  # ponteiro filho da direita

def __str__(self):        # Utilizado para impressao de um
if (self.caracter is None): # objeto tipo Node
return('0')  # No nao terminal
else:
return('1')  # No terminal (leaf node)

def __lt__(self,other):
if (self.frequencia <= other.frequencia):
return(True)
else:
return(False)

class BST:       # Classe que define uma arvore de busca binaria
def __init__(self,root=None): # BST (Binary Search Tree)
self.root = root

def devolveroot(self):
return self.root

def preordertraverse(self,p):
if(p is not None):           # p deve existir
# ou seja o no deve existir
if (p.caracter is not None): # No terminal
print(p,end='')
else:                        # No terminal
print(p,end='')
self.preordertraverse(p.left) # recursao a esquerda
self.preordertraverse(p.right) # recursao a direita
# O comando print(objeto) utiliza a funcao
# __str__() correspondente. No caso de node o resultado
# e' a impressao de '0' ou '1'
# O comando print(objeto) usualmente termina com o caracter
# que indica mudanca de linha '\n' quando se utiliza
# end='' nao ha mudanca de linha

def main():
a = Node('a',1) # cria um no terminal com o caracter 'a', frequencia=1
b = Node('b',3) # cria um no terminal com o caracter 'b', frequencia=3
c = Node(None,4,a,b) # cria um no nao terminal com frequencia=4
# filho da esquerda = a
# filho da direita = b
d = Node('d',3) # cria um no terminal com o caracter 'd', frequencia=3
e = Node(None,7,d,c) # cria um no nao terminal com frequencia=7
# filho da esquerda = d
# filho da direita = c
x = BST(e)       # cria um objeto BST com raiz dado por e
p = x.devolveroot() # p = raiz da arvore
x.preordertraverse(p) # atravessa a arvore em pre-ordem

#               e (7)
#              / \
#          (3) d c (4)
#               / \
#            (1) a b (3)
```

```
if __name__ == "__main__": main()
```

3.3 Implementação da fila de prioridades

A fila de prioridades onde os elementos da fila são objetos da classe `Node` pode ser implementada através da biblioteca `heapq.py` que implementa heaps numa lista simples. *Heaps* são árvores binárias onde o nó raiz contém o menor elemento. A *heap* é sempre mantida ordenada com a inserção ou retirada de elementos.

A listagem a seguir (arquivo `teste4.py`) ilustra a implementação de uma *heap*. Note que os elementos inseridos são objetos do tipo tuplas `(key,object)`. O primeiro elemento da tupla deve ser uma chave numérica para que possa ser utilizado para ordenação. No caso o primeiro elemento da tupla é o número de ocorrências do caracter.

A inserção de um elemento é feito através do método `heappush()`. O método `heappop()` retira o menor elemento.

A fila de prioridades admite inserção de elemento com o mesmo número de ocorrências, como pode ser observado no exemplo abaixo, onde os caracteres `'c'` e
'd' possuem o valor `frequencia=4`.

```
from heapq import heappush, heappop
from teste3 import Node

def main():
# cria objetos do tipo Node. O numero inteiro se refere a frequencia
a = Node('a',9)
b = Node('b',2)
c = Node('c',4)
d = Node('d',4)
e = Node('e',3)

h=[] # cria uma lista vazia aonde sera armazenada
# um objeto do tipo heap
# coloca cada objeto tipo Node na lista h[]
# objetos sao inseridos como uma tupla (key,object)
# A ordenacao e'feita com o valor da chave (key)
heappush(h,(d.frequencia,d))
heappush(h,(a.frequencia,a))
heappush(h,(b.frequencia,b))
heappush(h,(e.frequencia,e))
heappush(h,(c.frequencia,c))

# A heap e'organizada como uma arvore de busca binaria
# onde o no raiz se refere ao menor elemento da heap
# O comportamento da heap e' equivalente ao de uma fila de prioridades
# A cada insercao ou retirada de elemento a heap e'reorganizada para
# manter a estrutura de arvore

while len(h) is not 0: # Percorre ate a lista ficar vazia
x = heappop(h) # retira um elemento da heap
# o elemento com menor key
# e' retirado primeiro
# x e' uma tupla (key, object)
print('frequencia = ',x[0],' -> caracter = ',x[1].caracter)
# lembre-se que x[1] e' um objeto do tipo Node

if __name__ == "__main__": main()
```

3.4 Manipulação de bits

A menor unidade de memória que pode ser armazenada é equivalente a um *byte*. Para uma implementação real do algoritmo de codificação de Huffman é necessário a manipulação dos *bits* nas variáveis que armazenarão o código de Huffman.

O código de Huffman inicialmente é gerado através de um *array* de caracteres ou *string*. Por exemplo, seja a string `x` representando uma cadeia de 18 bits:

```
x='10011101 10100001 10'
```

Inicialmente, devemos separar a *string* em grupos de 8 bits para a formação de um *byte*:

```
byte1=x[0:8] # the first 8 bits
byte2=x[8:16] # the second group of 8 bits
```

Agora os conteúdos de `byte1` e `byte2` se tornam: `byte1 = '10011101'` e `byte2 = '10100001'`.

Agora devemos converter cada uma dessas *strings* no número correspondente:

```
val1=int(byte1,2)
val2=int(byte2,2)
```

Agora temos: `val1 = 157` e `val2 = 161`.

Aos 2 *bits* remanescentes devem ser acrescidos uma cadeia de seis zeros '000000' para completar a dimensão de um *byte*:

```
byte3=x[16:18]+'000000'
```

Agora a variável `byte3` se torna: `10000000`.

Como escrever esses números em um arquivo ?

```
arquivo=open('codigo.huf','wb');
arquivo.write(bytes(val1),bytes(val2),bytes(val3));
arquivo.close();
```

O uso da função `bytes()` garante que o que será escrito no arquivo tem o tamanho de um *byte*.

O arquivo `codigo.huf` contem apenas 3 *bytes*. Uma possível abstração para a organização interna desse arquivo pode ser como a ilustração a seguir:

$9D	$A1	$80

Tabela 1: Organização interna do arquivo codigo.huf representada por números hexadecimais.

Ou ainda poderíamos representar a organização interna através de números binários como ilustrado a seguir:

10011101	10100001	10000000

Tabela 2: Organização interna do arquivo codigo.huf representada por números binários.

3.5 Organização dos arquivos compactados *.huf

Os arquivos compactados devem ser organizados com a seguinte sequencia de informações:

1. Os primeiros 3 *bytes* devem ser interpretados como uma *string* de caracteres do tipo algarismo (`'0'-'9'`). Os algarismos compõem um número que se refere ao número de caracteres n_c presentes no texto. Essa informação é fundamental para reconstruir a árvore de codificação.

2. Os próximos 4 *bytes* representam outra *string* de caracteres que se referem ao número de *bits* n_b que compõem toda a informação codificada.

3. Os próximos *bytes* se referem a uma sequência de caracteres '0's e '1's contendo a informação de atravessamento em pré-ordem da árvore de codificação. Como é possível saber o tamanho dessa cadeia de caracteres ? Não é necessário fornecer essa informação ? O número de caracteres n_c presentes corresponde ao número de nós terminais que são representados pelo caracter '1'.

4. Todos os *bytes* em seguida se referem à informação codificada. Para esses *bytes* estamos interessados em ter acesso a cada um dos *bits*. O número de *bits* n_b é necessário para no último *byte* sabermos qual o número de *bits* que realmente pertencem a informação codificada.

4 Referências

1. Algorithms, Robert Sedgewick and Kevin Wayne, Addison-Wesley Professional, 4th Edition, 2011.

5 Para você fazer

1. Utilizando a IDE Jupyter Notebook, projete e implemente um algoritmo de codificação e de decodificação de Huffman segundo as diretivas apresentadas acima.

2. Implemente um programa principal onde o usuário pode escolher entre a operação de codificação ou de decodificação além do nome dos arquivos.

3. Utilizar os seguintes testes:

   (a) `ABRACADABRA!`

   (b) `it was the best of times it was the worst of times`

   (c) arquivo `Alice1.txt` contendo uma extração do primeiro capítulo de 'Alice in Wonderland'.

4. Em uma célula de texto (Markdown) do seu arquivo Jupyter Notebook escreva um relatório equivalente a no máximo uma página A4 contendo as seguintes informações:

   (a) Uma descrição da sua solução de maneira resumida.

   (b) Uma análise dos resultados obtidos com os arquivos teste.

5. O arquivo Jupyter Notebook com a sua solução deve ser submetido no sistema Moodle.

6. **DEADLINE: 27/06/2022 - 23h59min.**