Resumo IV

Bernardo Cavanellas Biondini - 20.1.4112

Nicolle Canudo Nunes - 20.1.4022

Compressão de Textos

consiste em representar o texto original de documentos em menos espaço (aplicar uma técnica para reduzir seu tamanho em bytes) → substituir os símbolos do texto por outros que ocupam um numero menor de bits ou bytes

Codificação: pega cada símbolo do texto original e transforma em outro, em um código, de forma que esse novo símbolo ocupe um espaço menor que o antigo, criando um novo arquivo com esses novos símbolos.

Ganho: o texto comprimido ocupa menos espaço de armazenamento, sendo mais eficiente em pesquisas e leituras por exemplo

Preço a se pagar: custo computacional para codificar e decodificar o texto

Decodificar: quando se quer descomprimir o texto, restaurar o texto para sua forma normal. Voltar os símbolos gerados para os símbolos originais.

em diversas ocasiões é mais interessante que o processo de decodificação seja mais rápido pois, ao pesquisar, por exemplo, por algum texto que foi codificado e armazenado, precisamos ter acesso a ele de forma que consigamos entender o que está escrito, sendo necessário que a decodificação seja rápida. Além disso, é interessante que a técnica de compressão permita realizar o casamento de cadeias no texto comprimido, diminuindo o custo da busca sequencial, uma vez que o tamanho do arquivo é menor. Outro fator importante é a possibilidade de acesso direto a qualquer parte do texto comprimido de forma em que possa haver a descompressão a partir da parte acessada, sem a necessidade de descomprimir um texto todo para ter acesso a uma parte dele.

Existe uma métrica capaz de determinar qual método é mais eficiente, é a Razão de Compressão → corresponde a porcentagem que o arquivo comprimido representa em relação ao tamanho do arquivo não comprimido
Exemplo: arquivo não comprimido de 100 bytes e arquivo comprimido de 30 bytes, a razão é 30%

a ideia é que a razão seja a menor possível, de forma que a compressão seja maior

Método de Huffman

método de compressão bastante eficaz, mais utilizado

Um código único, de tamanho variável, é atribuído a cada símbolo diferente do texto, alguns códigos, portanto, serão maiores que outros.

Códigos mais curtos são atribuídos a símbolos com frequências altas → se por exemplo uma palavra aparece diversas vezes no texto, sua frequência é alta e seu código será mais curto

Antes as implementações dos métodos consideravam caracteres como símbolos, mas atualmente utilizamos palavras como símbolos. Portanto, a implementação do método precisa do reconhecimento de um vocabulário e para cada palavra do texto iremos determinar um código, que será menor caso a frequência seja alta e maior se a frequência for baixa. Logo, além de estabelecer o vocabulário, precisamos de encontrar as ocorrências de cada uma delas.

Considera cada palavra do texto como símbolo → conta a frequência das palavras → gera um código de Huffman para elas → comprime o texto substituindo cada palavra pelo seu código correspondernte

A compressão é feita com 2 passadas pelo texto, a primeira para obter o vocabulário e a frequência das palavras e a segunda para realizar a compressão

Um texto em linguagem natural possui caracteres separadores (espaços,vírgulas, pontos, etc) e o tratamento desses separadores é feita da seguinte forma:
se uma palavra é seguida de um espaço somente a palavra é codificada, caso contrário, a palavra e o separador são codificados separadamente
no momento da decodificação, supõe-se que um espaço simples segue cada palavra a não ser que o próximo símbolo seja um separador.

Outra forma de considerar separadores seria codificá-las junto as palavras. Ex: "casa" ≠ "casa,", dessa forma teríamos um código para casa e outro para casa com a vírgula.

Entretanto, essa forma, pode não ser tão eficiente, uma vez que teríamos diversos códigos para uma mesma palavra, se diferenciando devido aos separadores

O algoritmo de codificação de Huffman se baseia em uma árvore, chamada de árvore de codificação.

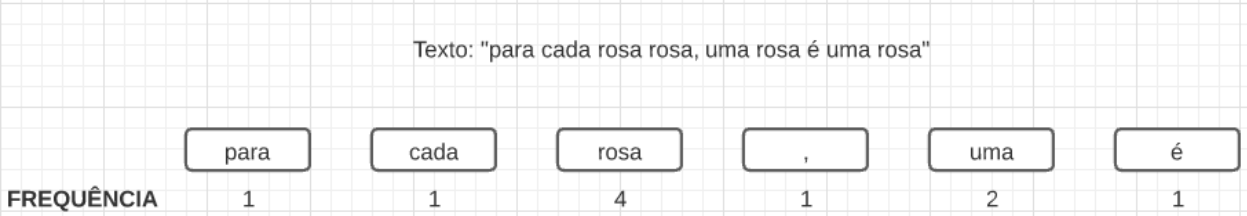
- O algoritmo de Huffman constrói uma árvore de codificação, partindo-se de baixo para cima.
  - Inicialmente, há um conjunto de *n* folhas representando as palavras do vocabulário e suas respectivas frequências.
  - A cada interação, as duas árvores com as menores frequências são combinadas em uma única árvore e a soma de suas frequências é associada ao nó raiz da árvore gerada.
  - Ao final de (*n*-1) iterações, obtém-se a árvore de codificação, na qual o código associado a uma palavra é representado pela sequência dos rótulos das arestas da raiz à folha que a representa.

A árvore de codificação é canônica, isso significa que um dos lados da árvore é maior ou igual ao outro.

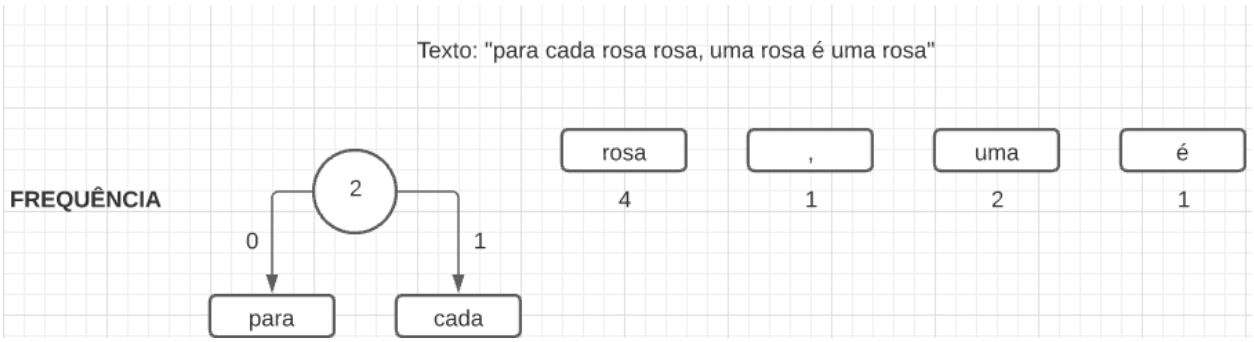
Exemplo:

considerando o texto: "para cada rosa rosa, uma rosa é uma rosa"

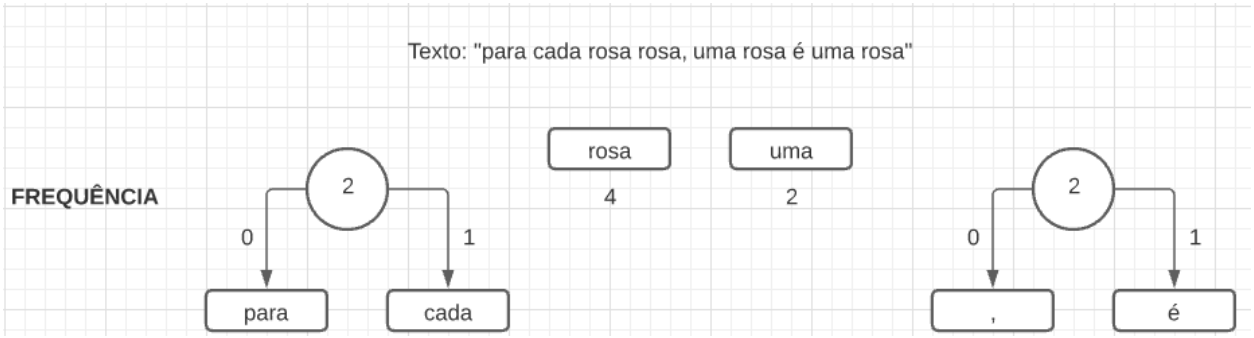
temos 5 palavras e um separador



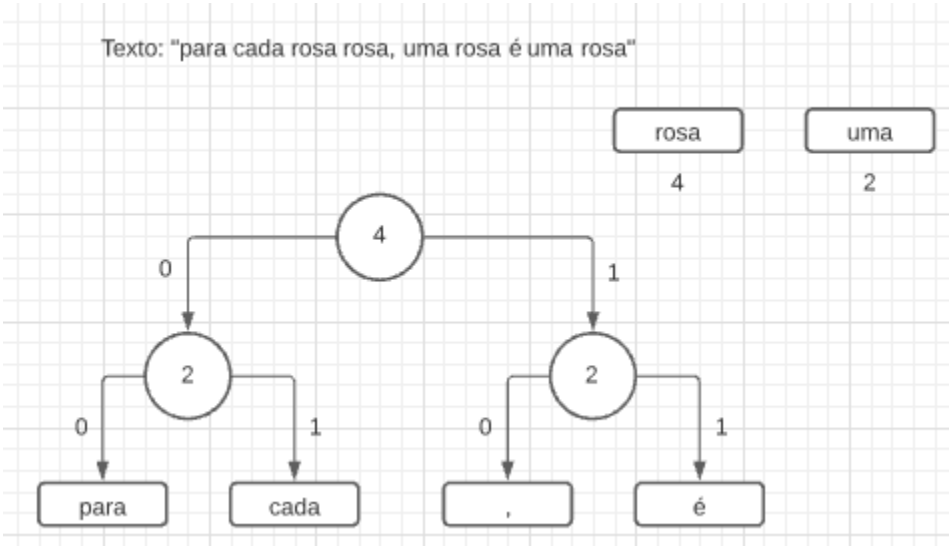
Para começar a montar a árvore, pegamos as palavras com menores ocorrências e o nó raiz da árvore gerada representa a soma de suas frequências



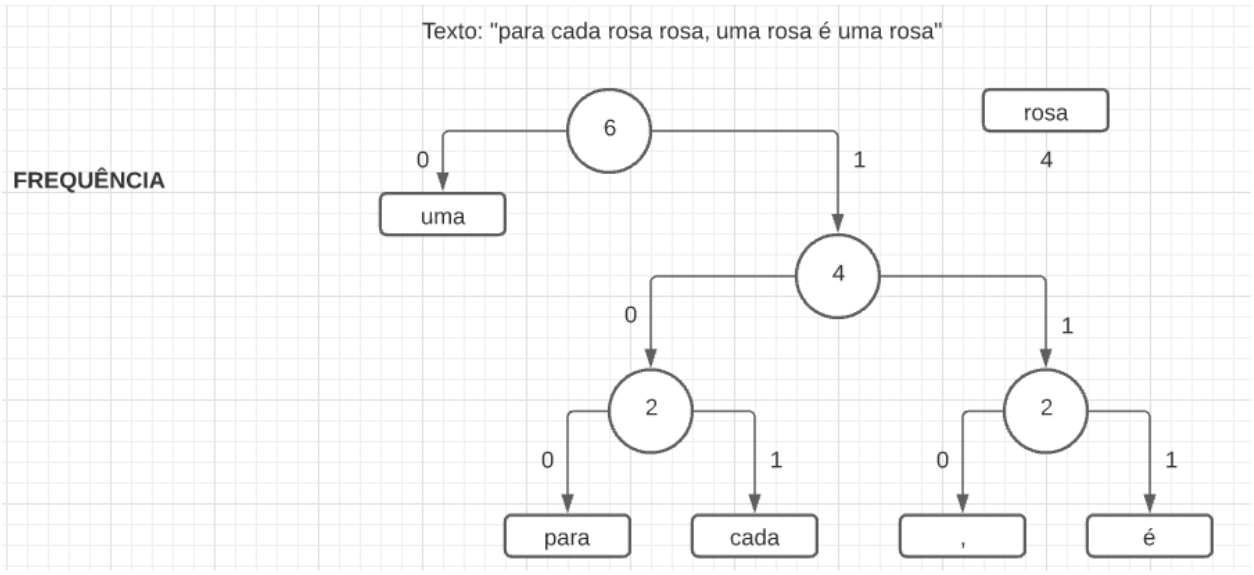
fazemos a mesma coisa para as próximas palavras



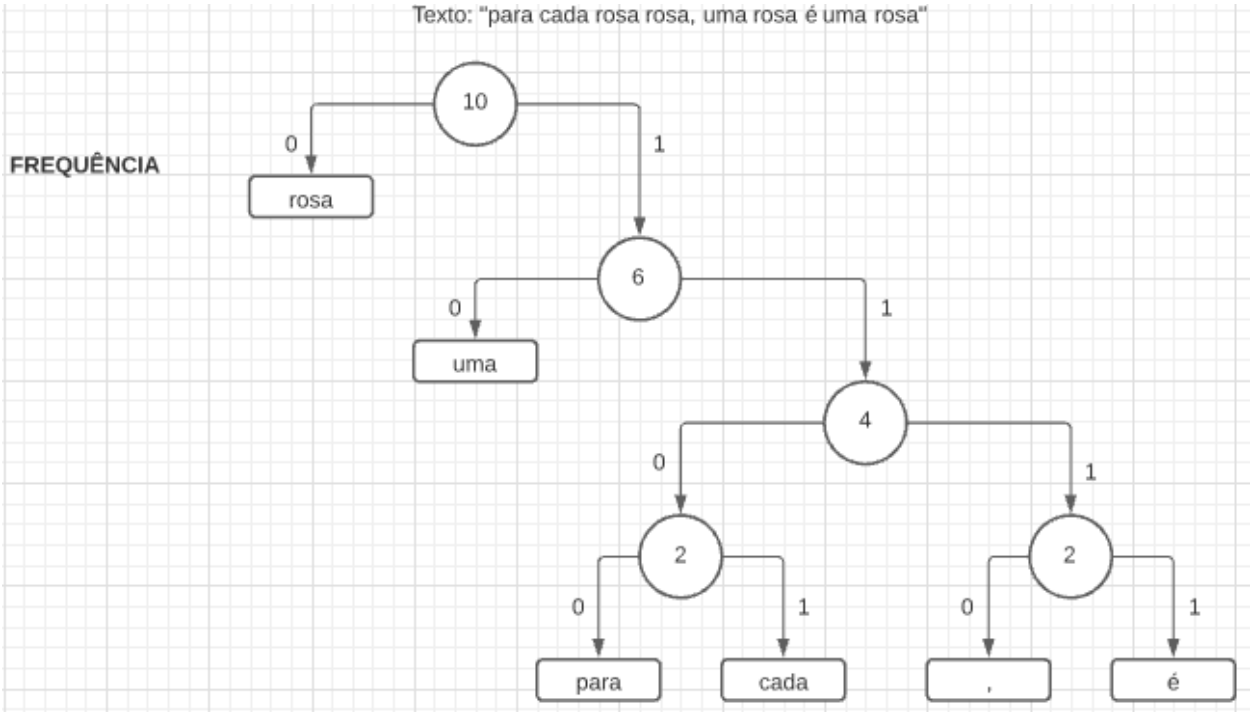
agora que já temos 2 subárvore, unimos elas para gerar uma subárvore canônica mais efetiva



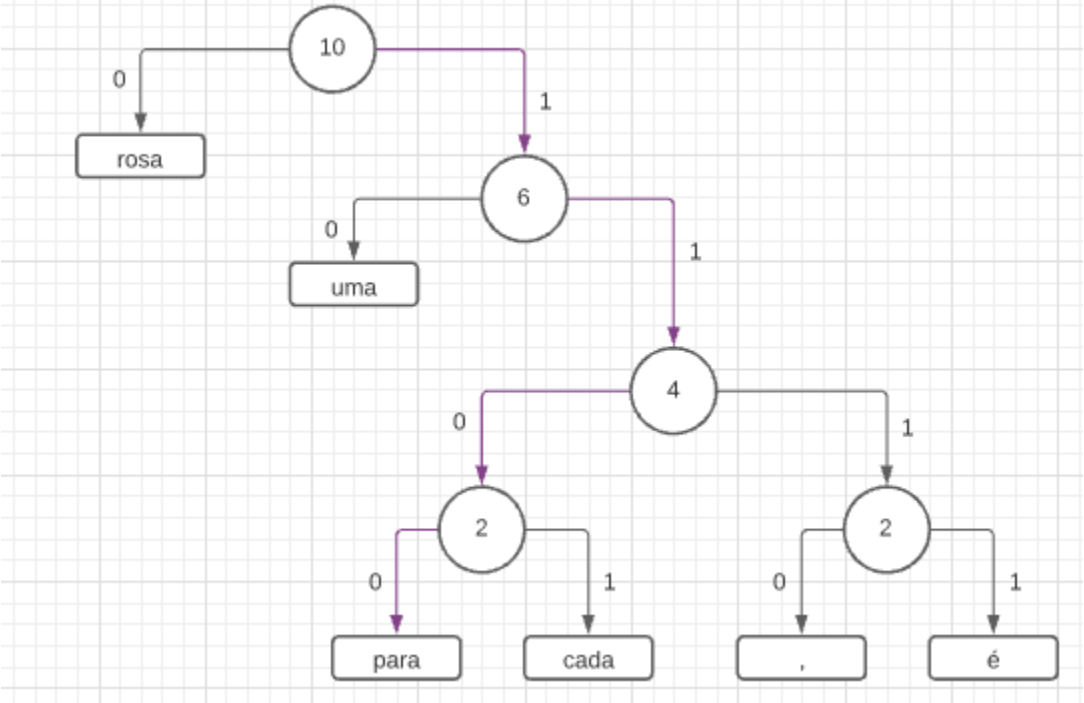
dessa forma, agora podemos "pegar" a palavra de frequência 2 e unir com a subárvore formada



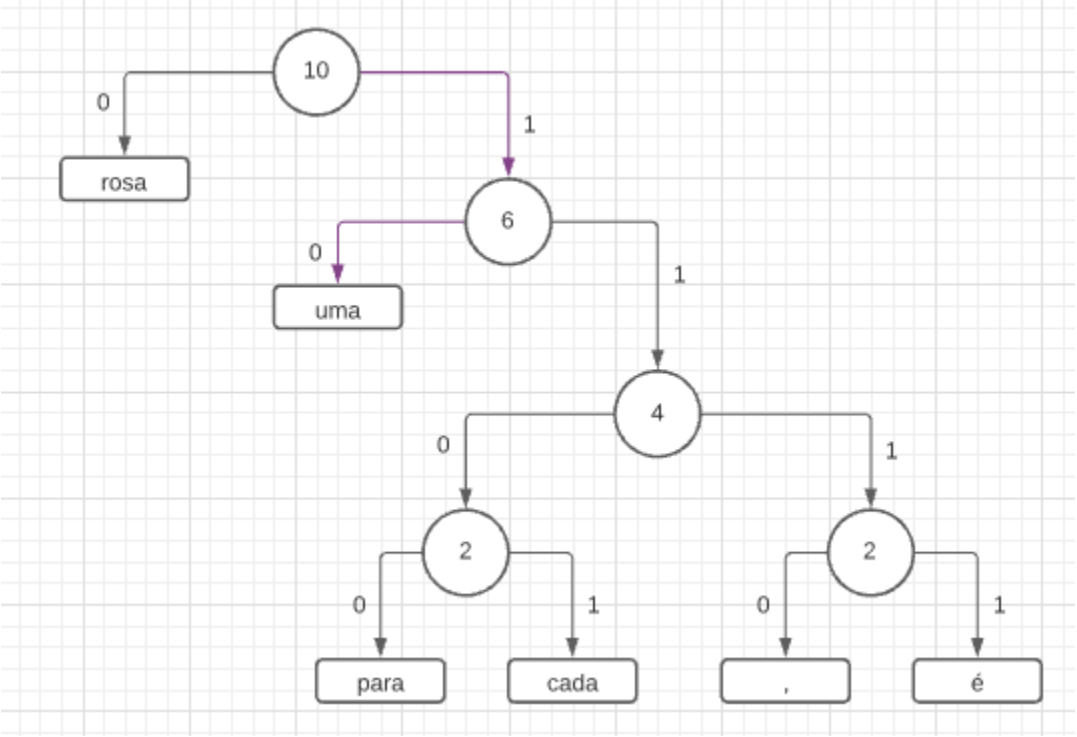
para o próximo e último passo pegamos a última palavra e unimos com a árvore já formada



com a árvore pronta temos o código referente a cada palavra do texto, bastando fazer o caminhamento para ter acesso ao código de cada palavra.



nesse exemplo a palavra "para" tem código 1100, enquanto "uma" tem código 10



dessa forma, mantemos a "regra" de que palavras com maiores ocorrências possuem código menor

Agora, para fazer o processo de compressão, é preciso substituir as palavras pelos seus respectivos códigos. A sequência, o código, é feito a partir do caminhamento da árvore. Entretanto, para vocabulários extensos, realizar o caminhamento da árvore para cada palavra se torna muito custoso. Dessa forma, antes de codificar, fazemos o caminhamento inteiro da árvore e guardamos o código referente a cada uma das palavras, construindo uma estrutura eficiente de pesquisa (árvore B, árvore TRIE, etc) para o vocabulário.

Já no processo de descompressão, esse processo é mais rápido, uma vez que nos deparamos com o código da palavra codificada, sendo mais direto acessar a palavra referente, basta seguir a sequência binária.

Para vocabulários maiores, a construção dessa estrutura pode ser muito custosa, levando em consideração o espaço utilizado e o tempo. Dessa forma, é possível simular essa estrutura de forma que o custo seja menor, a partir do algoritmo de Moffat e Katajainen.

Algoritmo de Moffat e Katajainen

baseado na codificação canônica (a partir de uma árvore de codificação canônica)

> Permite, a partir de um texto original, descobrir quais os códigos referentes a cada palavra, sem construir a árvore, apenas a simulando

apresenta comportamento linear em tempo e espaço

o algoritmo não encontra os códigos, mas calcula os comprimentos dos códigos em lugar dos códigos propriamente ditos, após esse cálculo podemos identificar qual o código propriamente dito, em uma etapa posterior.

O algoritmo se divide em 3 etapas:

- Combinação de nós
- Determinação das profundidades de nós internos
- Determinação das profundidades dos nós folhas (comprimentos dos códigos)

O algoritmo consiste na construção de um vetor contendo a frequência das palavras do vocabulário em ordem decrescente, cada posição representa uma palavra. Portanto

para o texto "para cada rosa rosa, uma rosa é uma rosa", teremos um vetor A:

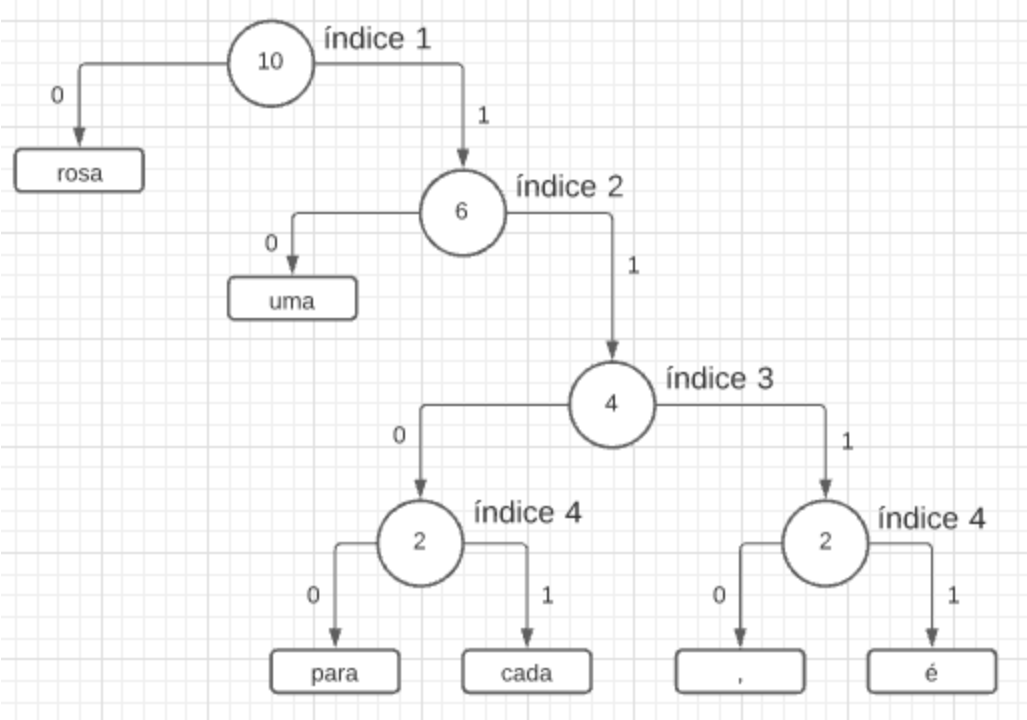
4	2	1	1	1	1

durante a execução serão usados sub-vetores temporários que coexistem dentro do próprio vetor, realizando algumas operações de forma a simular a árvore canônica sem a utilização de nenhuma outra estrutura de complexidade alta.

Dessa forma, é possível realizar as três fases com apenas um vetor.

1a fase: Combinação de nós

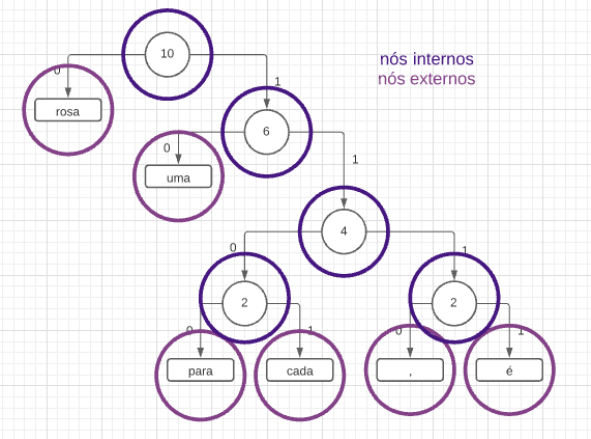
tem como objetivo calcular os índices → corresponde ao nível da árvore que o nó se encontra



vetor como dado de entrada

cada posição representa uma palavra e seus valores são as frequências delas.

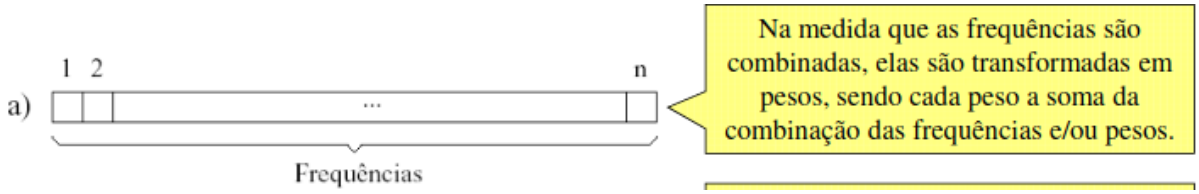
um fator importante sobre a árvore canônica é que a quantidade de nós



internos é sempre menor em 1 do que os nós externos

O vetor inicial guarda as frequências das palavras do vocabulário

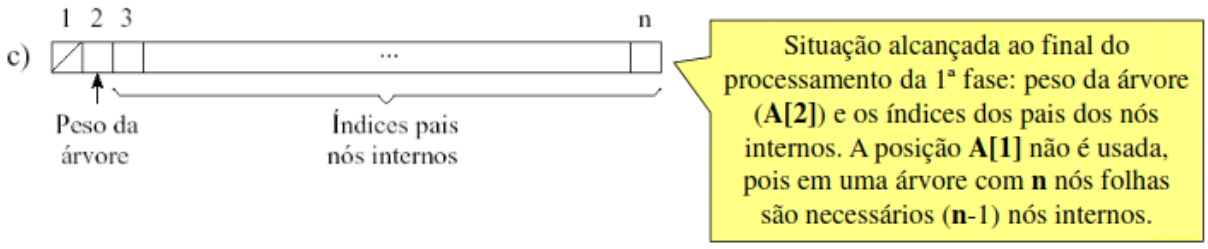
na medida que as frequências são combinadas, elas se transformam em pesos, sendo esses a soma da combinação das frequências e/ou pesos



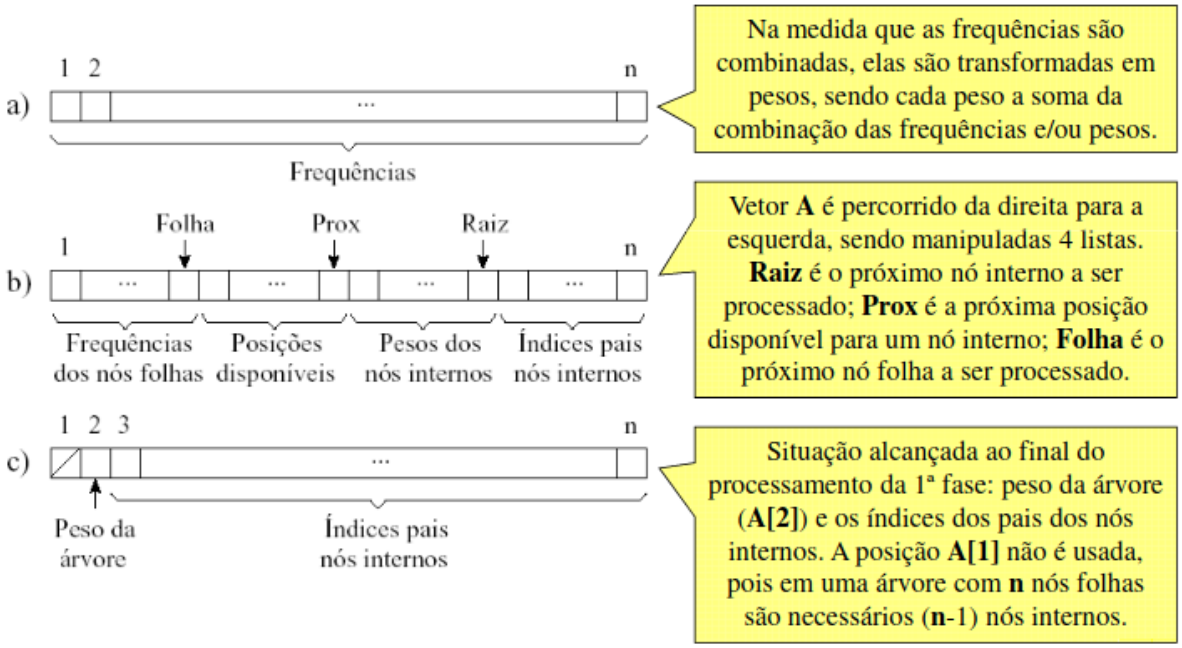
O vetor final deve guardar os índices dos pais dos nós internos

como a quantidade de nós internos é menor que a quantidade de nós externos, a primeira posição do vetor fica vazia

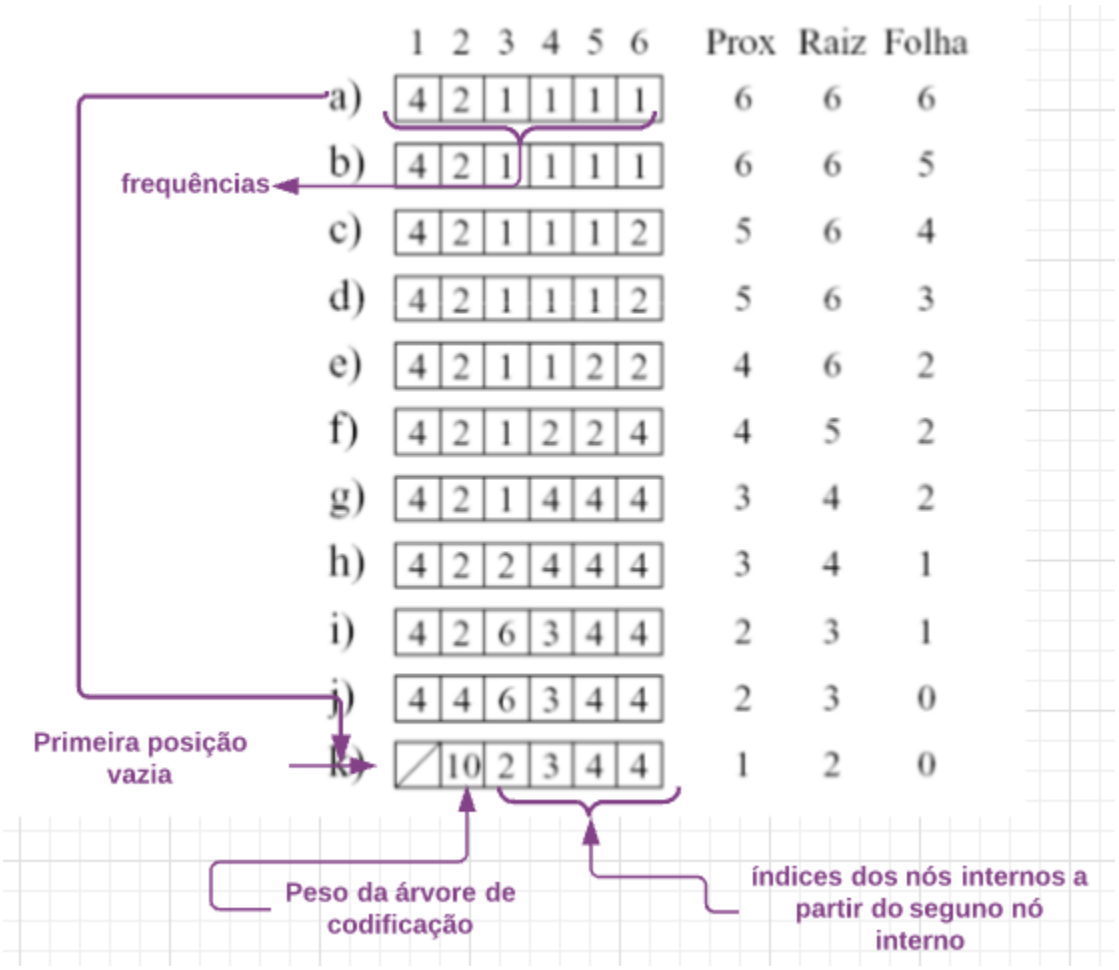
a segunda posição guarda o peso da raiz da árvore e as demais os índices dos pais dos nós internos, alcançando a seguinte situação:



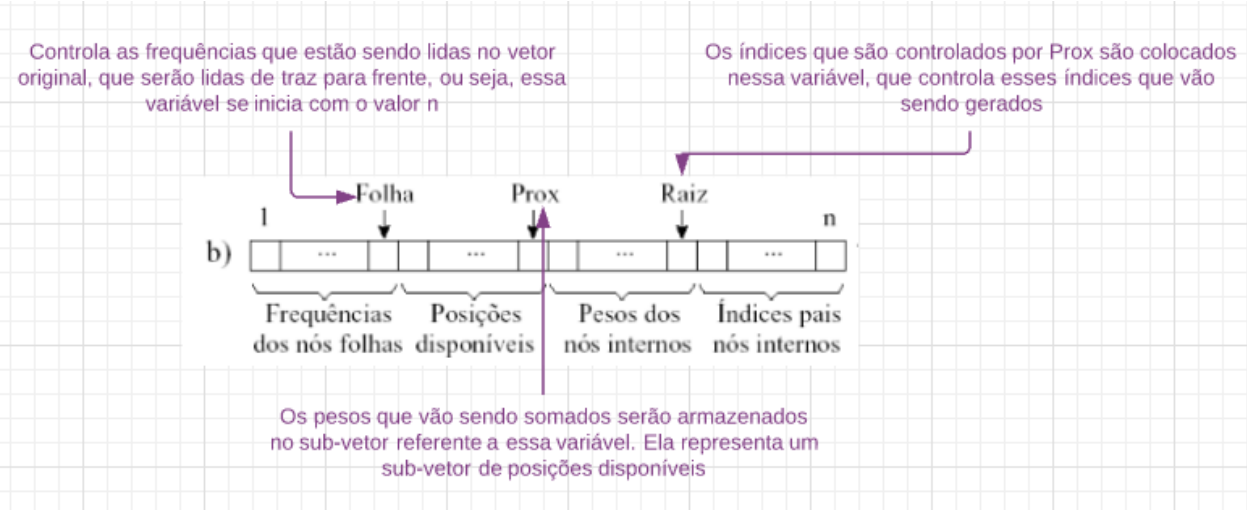
Portanto, o algoritmo possui os seguintes passos nessa fase:



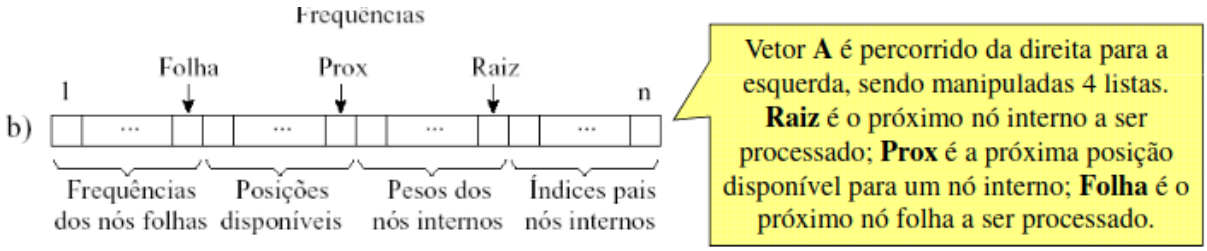
Seguindo o exemplo do Texto: "para cada rosa rosa, uma rosa é uma rosa", o vetor deve ter, durante os passos, o seguinte formato:



o algoritmo cria sub-vetores temporários para realizar a transformação do vetor, esses sub-vetores são controlados pelas variáveis Folha, Prox e Raiz



Portanto, pegamos os pesos controlados pela variável Folha e "jogamos" seus somatórios nas posições da variável Prox, que representa um sub-vetor de posições disponíveis, e, a partir disso, conseguimos calcular os índices pais dos nós internos, pela variável Raiz que controla o sub-vetor com esses índices. A raiz, vai sendo decrementada e, assim, o sub-vetor de índices pais dos nós internos aumenta.



O algoritmo da primeira fase:

```
PrimeiraFase (A, n)
{ Raiz = n; Folha = n;
  for (Prox = n; n >= 2; Prox—)
  { /* Procura Posicao */
    if ((nao existe Folha) || ((Raiz > Prox) && (A[Raiz] <= A[Folha])))
    { A[Prox] = A[Raiz]; A[Raiz] = Prox; Raiz = Raiz – 1; /* No interno */
    else { A[Prox] = A[Folha]; Folha = Folha – 1; /* No folha */ }
    /* Atualiza Frequencias */
    if ((nao existe Folha) || ((Raiz > Prox) && (A[Raiz] <= A[Folha])))
    { /* No interno */
      A[Prox] = A[Prox] + A[Raiz]; A[Raiz] = Prox; Raiz = Raiz – 1;
    }
    else { A[Prox] = A[Prox] + A[Folha]; Folha = Folha – 1; /* No folha */ }
  }
}
```

```
PrimeiraFase (A, n)
{ Raiz = n; Folha = n;
  for (Prox = n; Prox >= 2; Prox--)
  { /* Procura Posicao */
    if ((nao existe Folha) || ((Raiz > Prox) && (A[Raiz] <= A[Folha])))
    { A[Prox] = A[Raiz]; A[Raiz] = Prox; Raiz = Raiz - 1; /* No interno */ }
    else { A[Prox] = A[Folha]; Folha = Folha - 1; /* No folha */ }
    /* Atualiza Frequencias */
    if ((nao existe Folha) || ((Raiz > Prox) && (A[Raiz] <= A[Folha])))
    { /* No interno */
      A[Prox] = A[Prox] + A[Raiz]; A[Raiz] = Prox; Raiz = Raiz - 1;
    }
    else { A[Prox] = A[Prox] + A[Folha]; Folha = Folha - 1; /* No folha */ }
  }
}
```

Raiz, Folha e Prox começam com o valor n

Trata dos nós internos

Faz a combinação dos pesos

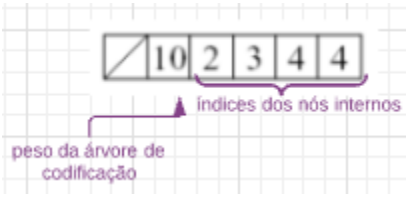
trabalha com as frequências

Trabalha com os índices dos nós internos

Atualização ds frequências, gerando os índices dos nós internos

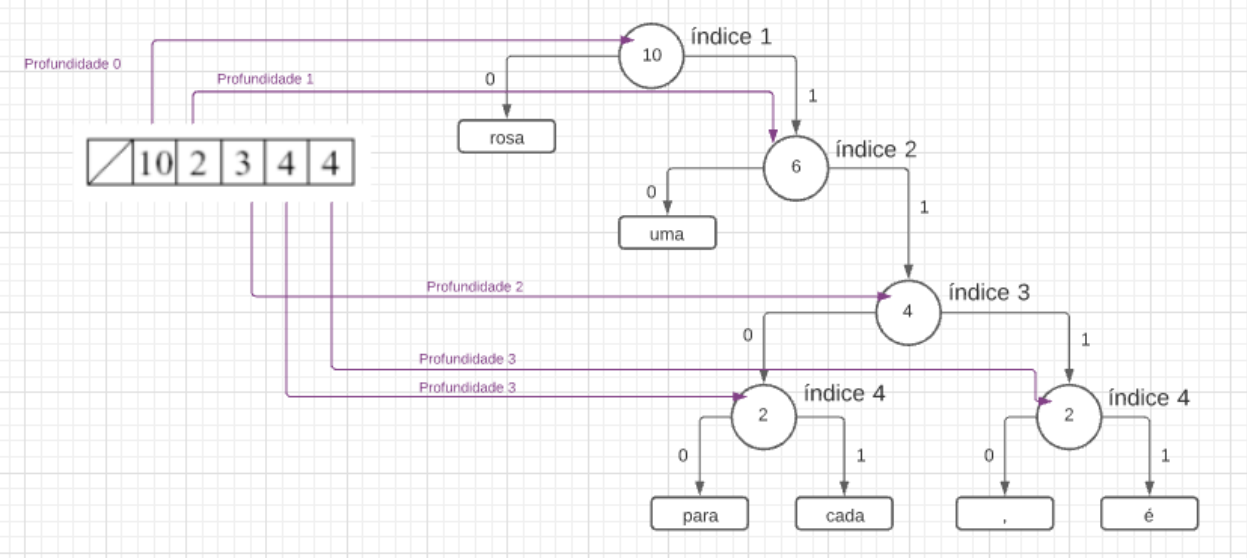
combinação dos pesos das folhas

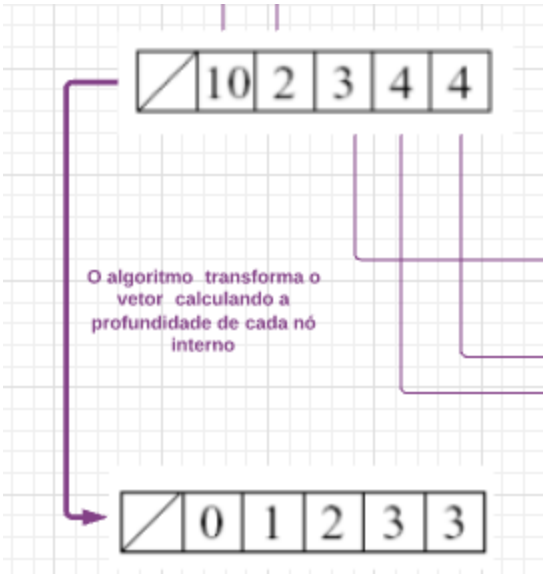
após esse processo, o vetor referente ao texto "para cada rosa rosa, uma rosa é uma rosa", na primeira fase, é:



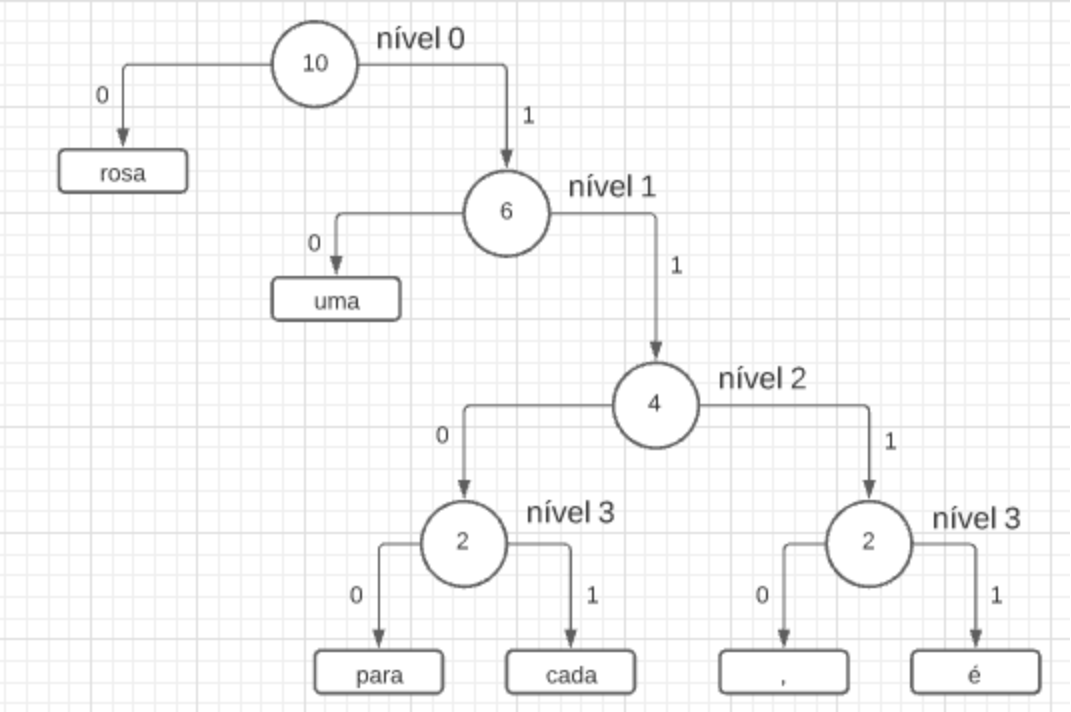
2a fase: Profundidade dos nós internos

dado o vetor gerado na primeira fase, vamos calcular a profundidade dos nós internos, qual a profundidade de cada nó interno cujos níveis já se sabem





é possível calcular a profundidade dos níveis a partir da raiz. Uma vez que a raiz tem nível 0, o próximo sera 0+1 e o próximo 0+1+1 e assim por diante



isso ocorre por meio do seguinte algoritmo:

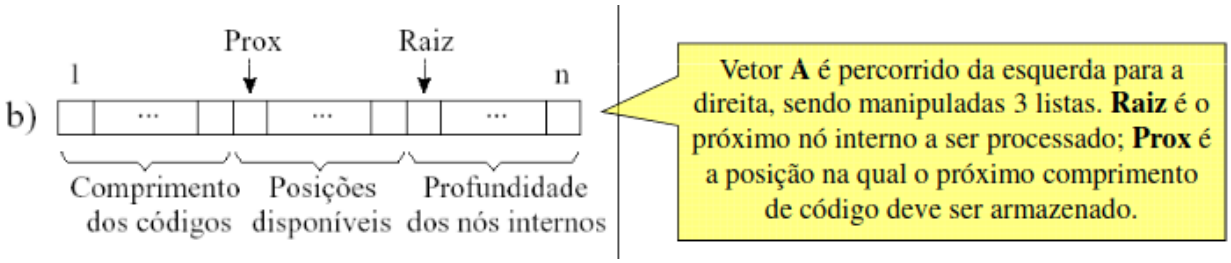
```
SegundaFase (A, n)
{ A[2] = 0;
  for (Prox = 3; Prox <= n; Prox++) A[Prox] = A[A[Prox]] + 1;
}
```

3a fase: Profundidade dos nós folhas

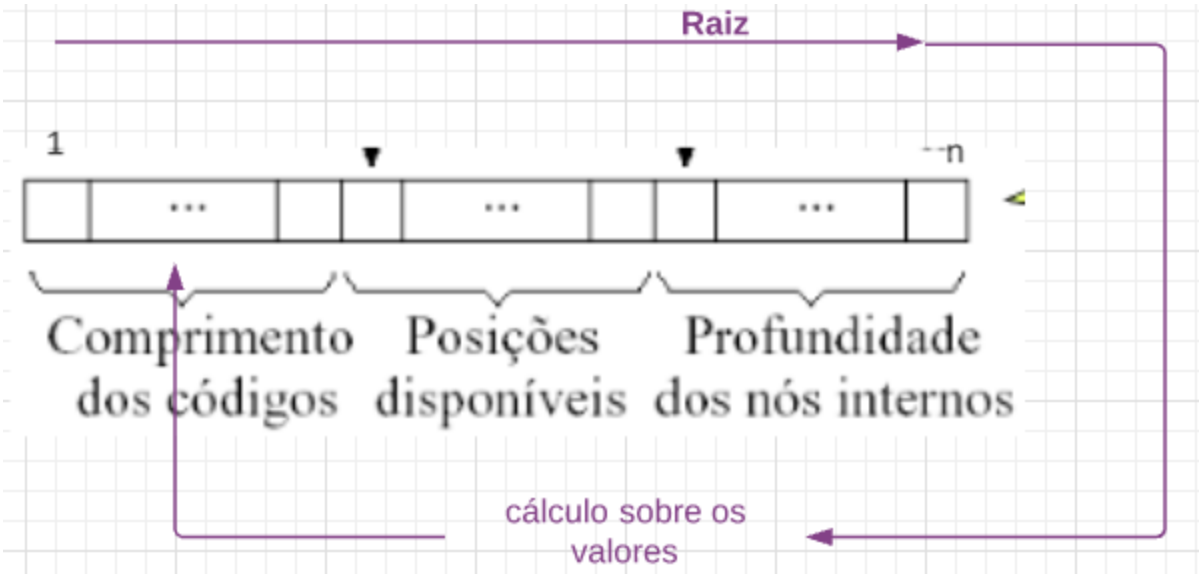
dado o vetor resultante da fase 2, vamos calcular a profundidade dos nós internos que é igual ao comprimento dos códigos

para isso utilizaremos 2 variáveis para representar sub-vetores do vetor A, Prox que vai calcular as posições disponíveis e Raiz que vai controlar a profundidade dos nós internos

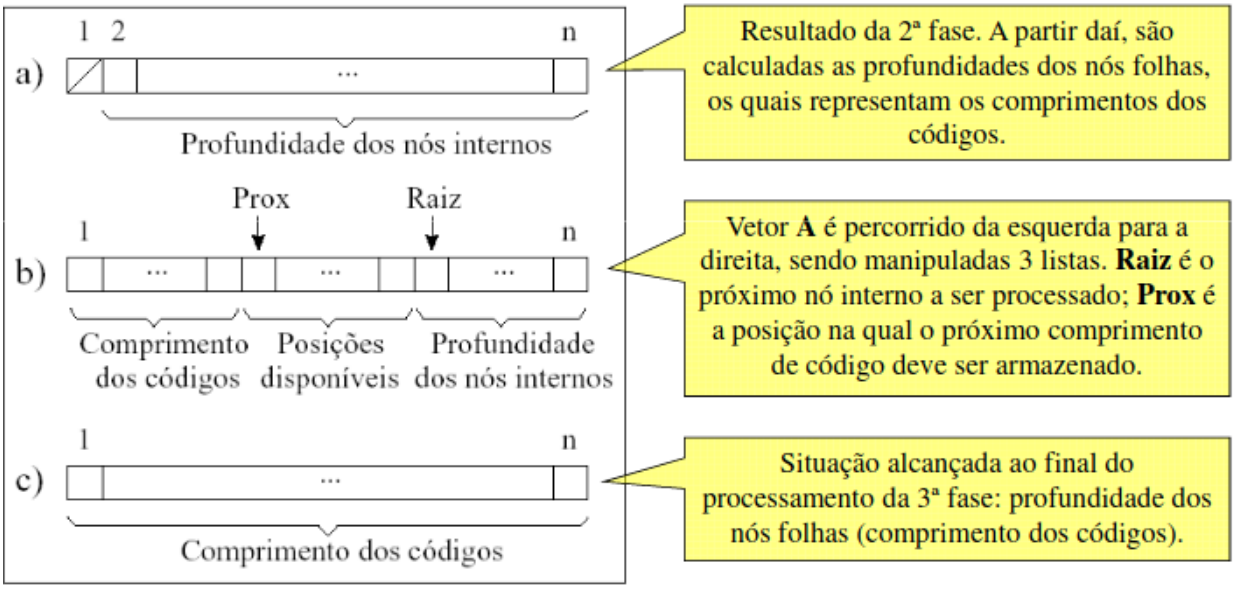
dessa vez, vamos lendo o vetor de forma sequencial



a partir da posição 2, vamos fazer o caminhamento da variável Raiz para a direita, realizando cálculos sobre os valores para "jogar" os comprimentos dos códigos na parte anterior, controlada por Prox



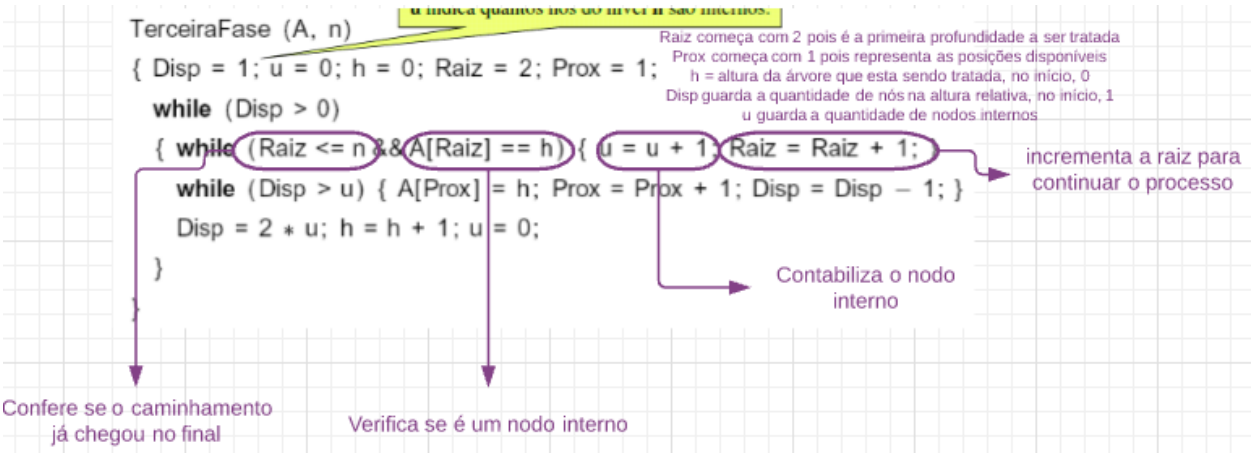
dessa forma passamos a cada vez mais aumentar o espaço referente ao comprimento dos códigos

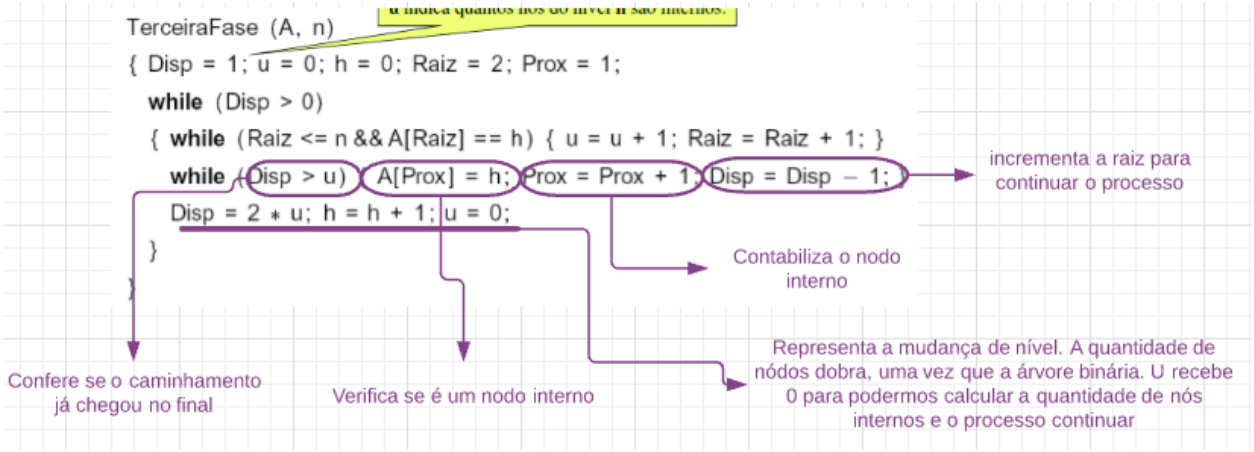


o algoritmo referente a essa fase:

Disp armazena quantos nós estão disponíveis no nível **h** da árvore.
u indica quantos nós do nível **h** são internos.

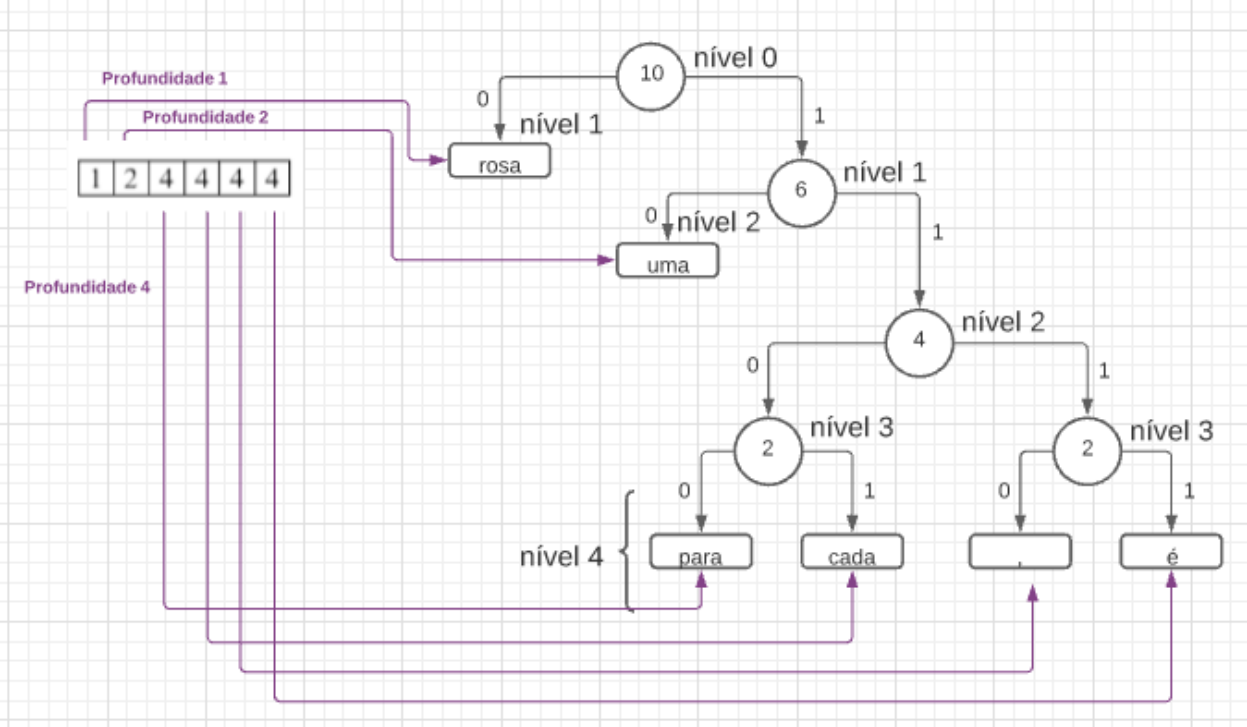
```
TerceiraFase (A, n)
{ Disp = 1; u = 0; h = 0; Raiz = 2; Prox = 1;
  while (Disp > 0)
  { while (Raiz <= n && A[Raiz] == h) { u = u + 1; Raiz = Raiz + 1; }
    while (Disp > u) { A[Prox] = h; Prox = Prox + 1; Disp = Disp – 1; }
    Disp = 2 * u; h = h + 1; u = 0;
  }
}
```





após esse processo o vetor A terá a seguinte cara:

1	2	4	4	4	4



cada posição guarda o tamanho do código de cada uma das palavras, ou seja, a palavra "rosa" tem código de tamanho 1 (0) e a palavra "para" tem código de tamanho 4 (1100)

Para execução do algoritmo completo

```
CalculaCompCodigo (A, n)
{ A = PrimeiraFase (A, n);
  A = SegundaFase (A, n);
  A = TerceiraFase (A, n);
}
```

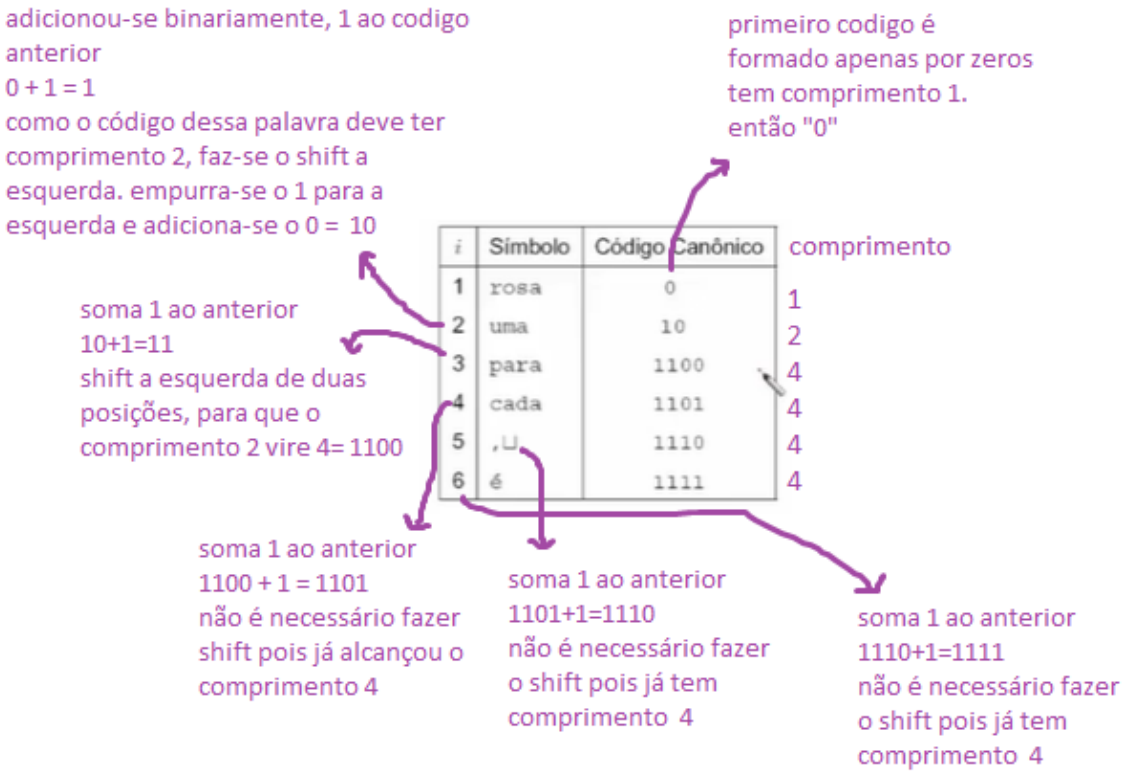
A partir do tamanho do código de cada palavra, obtido por esse algoritmo, podemos calcular os código propriamente ditos

Compressão de textos: Obtenção de códigos canônicos

- os comprimentos dos códigos canônicos seguem o algoritmo de Huffman
- códigos de mesmo comprimento são inteiros consecutivos

cálculo dos códigos

- o primeiro código é composto apenas por zeros
- para os demais, adiciona-se 1 ao código anterior e, se necessário, faz-se um deslocamento à esquerda para obter-se o comprimento necessário

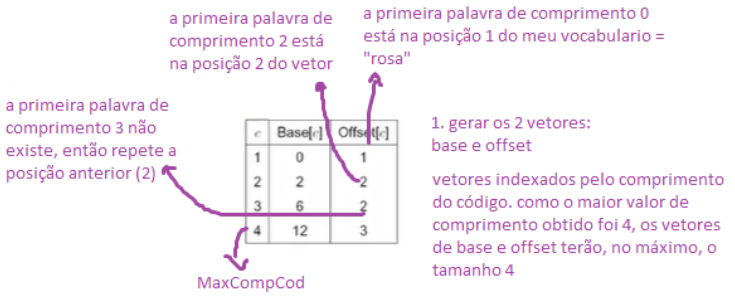


chega-se nos mesmos códigos fornecidos no caminhamento da árvore de codificação

o arquivo comprimido é o arquivo onde os símbolos são substituídos por seus respectivos códigos canônicos. mas, no momento da descompressão, é necessário

saber essa relação entre código x palavra. por isso, no cabeçalho do arquivo comprimido, deve-se armazenar esses dados, por mais que ocupe muito espaço.

uma forma de poupar o armazenamento ao guardar essa relação, seria gerar o código canônico apenas no momento em que ele seja necessário, da seguinte forma:



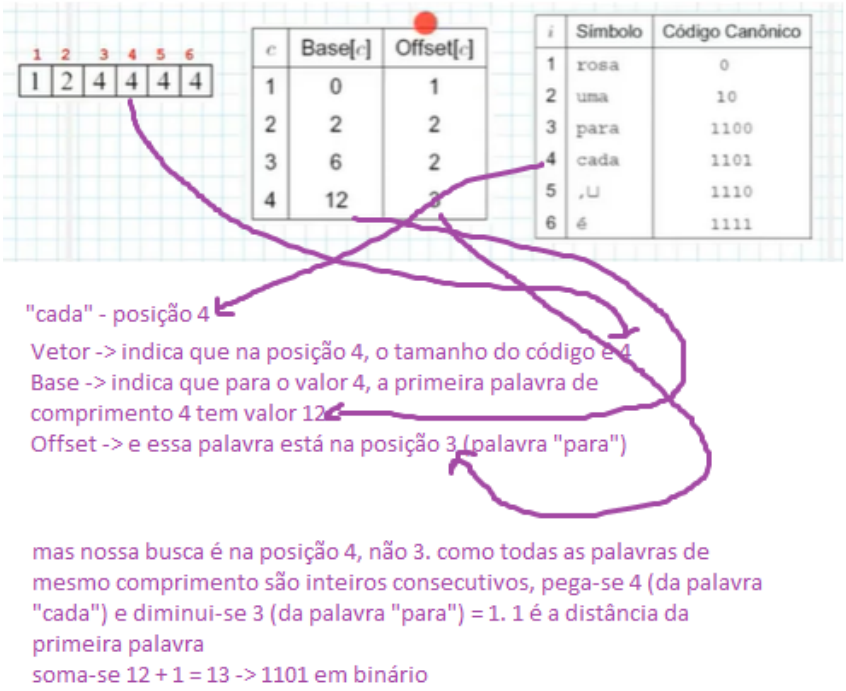
Base: indica, para um comprimento c, o valor inteiro do primeiro código com esse comprimento

Nº de códigos com comprimento (c-1).

$$\text{Base}[c] = \begin{cases} 0 & \text{se } c = 1, \\ 2 \times (\text{Base}[c-1] + w_{c-1}) & \text{caso contrário.} \end{cases}$$

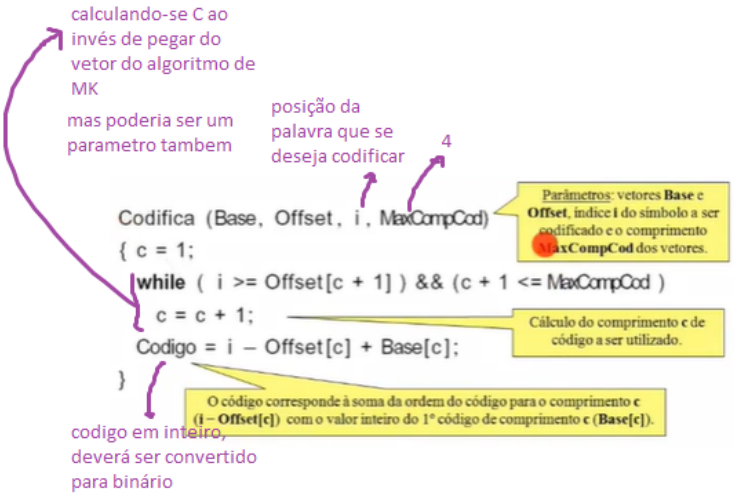
Offset: indica o índice do vocabulário que diz respeito a primeira palavra com esse comprimento

exemplo: palavra CADA

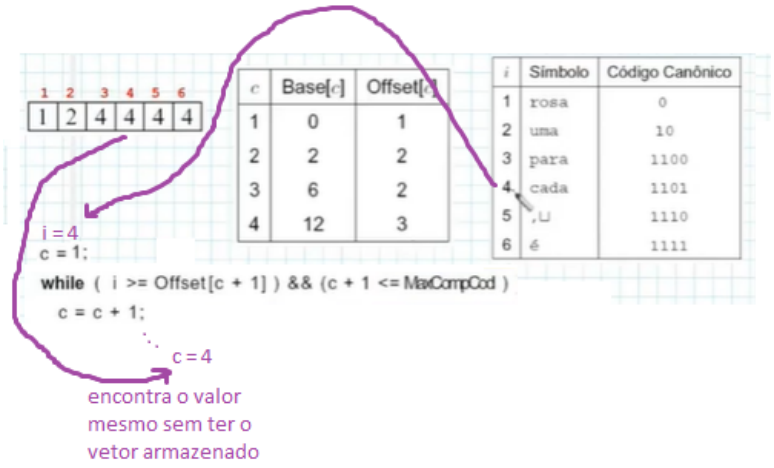


dessa forma, não será necessário armazenar os códigos canônicos no arquivo comprimido. somente as palavras e valores; vetores base e offset

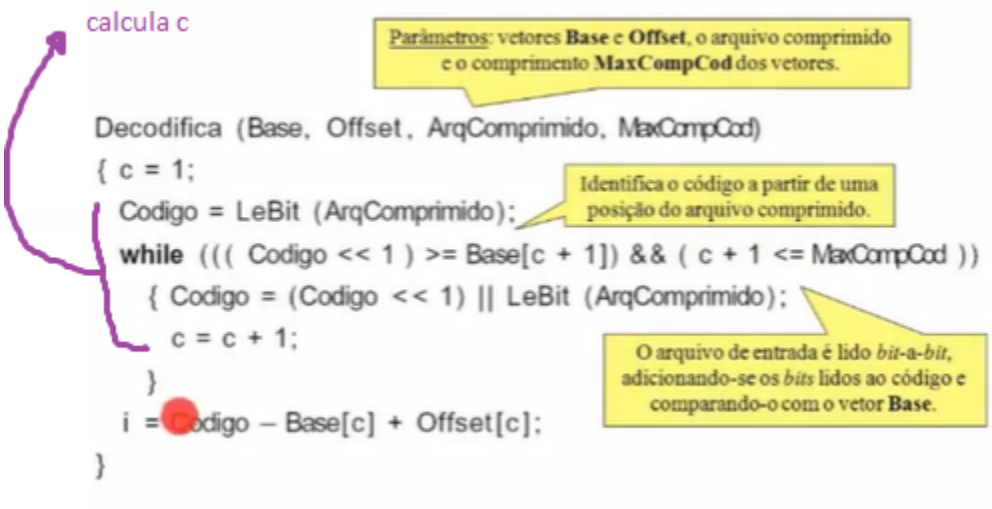
função de codificar

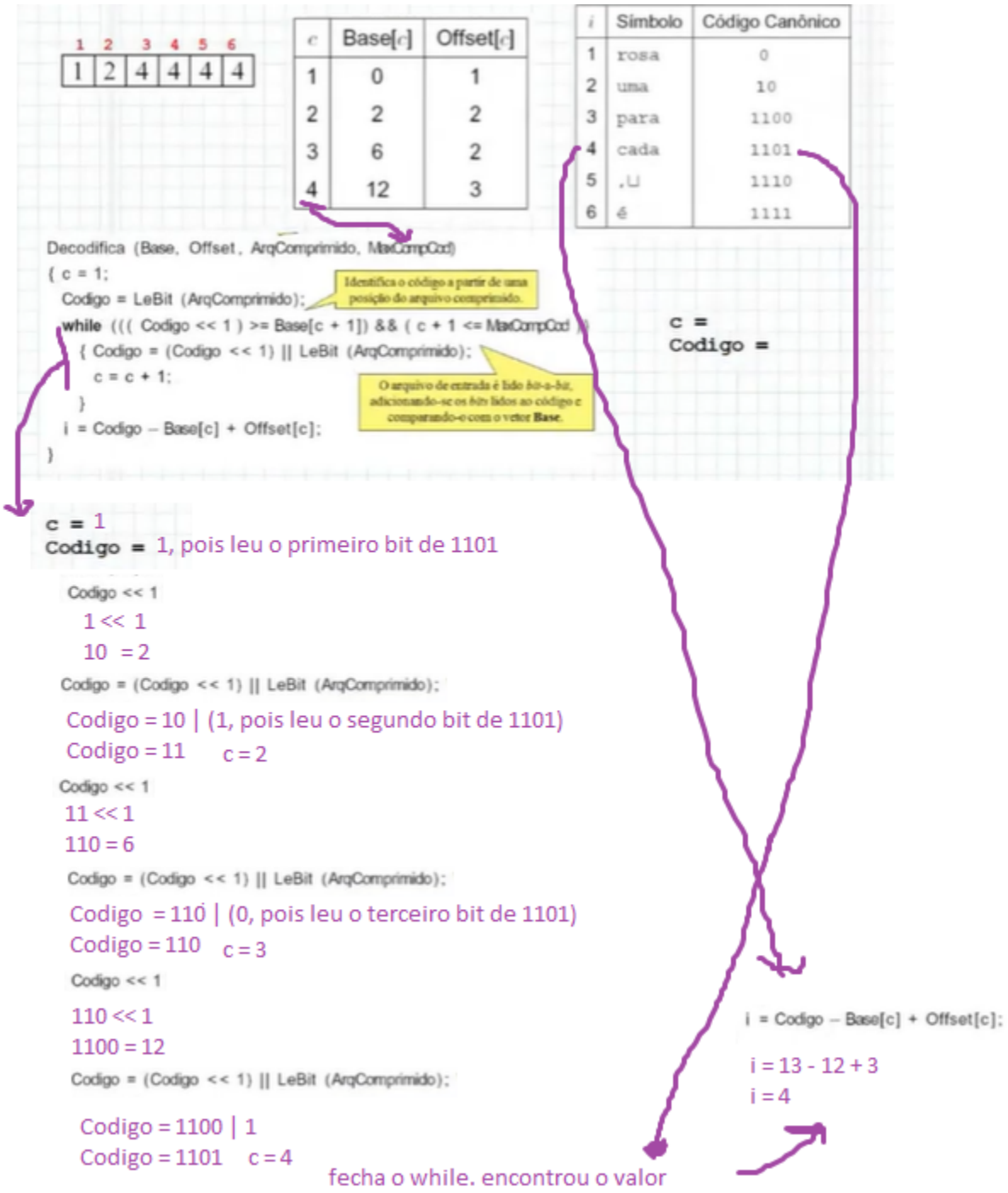


cálculo do comprimento usando offset e maxcomp



função de decodificar





Recapitulando...

- Compressão

processo mais demorado

3 etapas

1: calcular vocabulário e frequência

2: ordenar por frequência, aplicar o algoritmo MK, calcular base e offset e gravar no arquivo os vetores, assim como o vocabulário

3: compressão propriamente dita, percorre o arquivo, extrai e codifica as palavras

- Descompressão

mais simples e rápida, apenas 1 etapa

leitura dos vetores base, offset e vocabulário

leitura dos códigos, decodificando-os e gravando as palavras no .txt