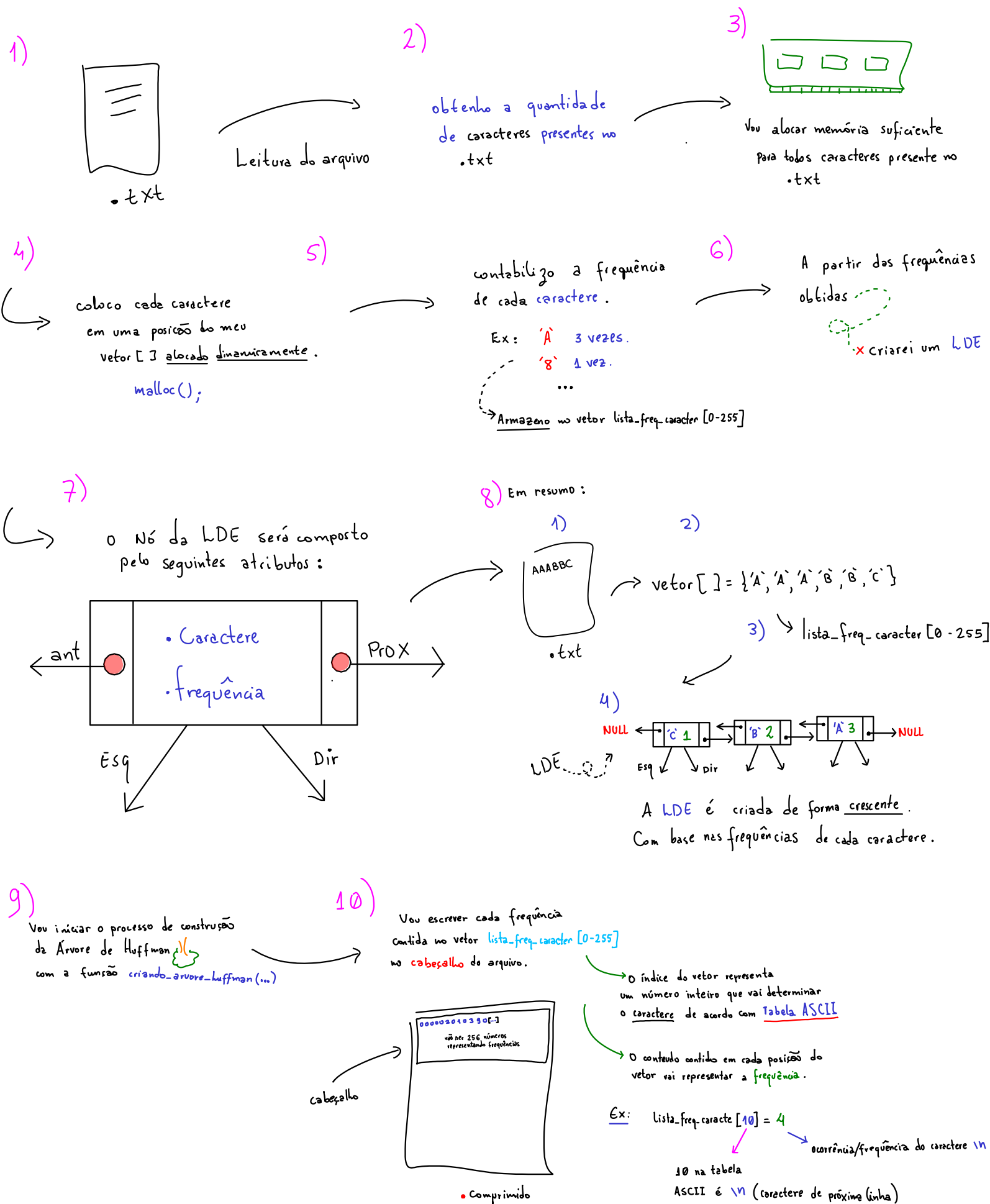


# Visão Geral sobre o funcionamento do programa:

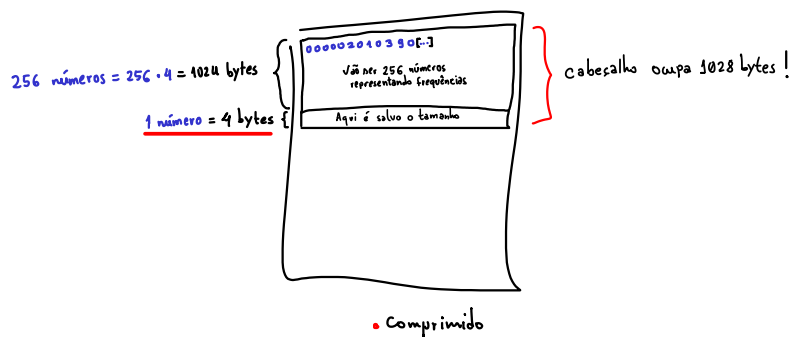


11)

Vou obter o tamanho da cadeia de bits gerada pela função de Huffman, que corresponde ao tamanho contido do .txt original.

12)

Vou escrever no cabecalho do arquivo um número decimal que corresponde o tamanho da cadeia de bits gerada pela função.



13)

Em seguida pego as cadeias de bits (gerado pela função gera\_codigo(...)). Essa cadeia de bits vai possuir TAMANHO VARIÁVEL, Ex: 0011 ou 00011101 ou 00011100011. Então a partir dessa cadeia recebida, eu vou converter ela para um número decimal. Porém, esse número decimal deve SEMPRE ser formado por 8 bits.

----- → 8 bits

Logo, se eu receber uma cadeia de bits composta por 5 bits, Ex: 01101, eu preciso ESPERAR outra cadeia chegar para pegar os outros 3 bits que estão faltando para assim completar os 8 bits.

Suponha que a 1ª cadeia que chegou foi: 01101 → faltam 3 bits → 2ª cadeia que chegou foi: 110101 → concateno 01101 + 110101  
vou precisar desses 3 primeiros bits

→ Agora completei 8 bits que precisava [01101110]  
Resultado final

• Esse processo foi feito utilizando: FILAS - utilizadas para armazenar os 8 bits, cada bit em um NÓ. OBS: cada fila é desalocada da memória quando atinge tamanho = 8.



VETORES - utilizados para guardar alguns bits, que acabam sobrando no momento em que é feita a transferência dos bits, para as FILAS. Os vetores vão ser importantes, pois às vezes vão chegar cadeias de bits da seguinte forma: 1011, então preciso de +4 bits, para assim completar 8 bits.

No entanto, a próxima cadeia que chegou possui 1011101. Preciso somente dos 4 primeiros bits, mas não posso perder os 3 bits restantes. Portanto, preciso armazenar esses 3 últimos bits no vetor [...] para não perder esses bits.

14)

[01101110]

Resultado final

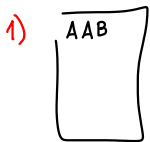
Utilizando a função `converte_binario_para_decimal(...)`  
converto esses 8 bits para um número decimal, em seguida para um caractere ASCII equivalente ao número decimal.

Exemplo:

- Entrada em binário: 01101110 → Aqui eu tinha 8 bytes, pois  $8 * [\text{char}] = 8 \text{ bytes}$ .
  - Processamento.....: 110 em decimal → Aqui eu tenho 4 bytes, pois é um  $n = [\text{inteiro}]$ .
  - Saída - ASCII.....: n → Aqui eu tenho 1 byte, pois é um  $[\text{char}]$ .
- Escrevo o caractere n no arquivo.

Esse processo é feito com todas as cadeias de bits que a árvore de Huffman gerou.

\* Mas por que realizar esse processo ?



.txt

Arquivo que quero comprimir



A árvore gerou a seguinte cadeia de bits → 01101110  
A A B

3)

Se eu fosse escrever esses 8 bits [01101110] no arquivo comprimido eu já ocuparia 8 - 1 byte = 8 bytes, fora os bytes ocupados pelo cabeçalho.

4)

Por outro lado, se eu converter essa cadeia [01101110] para um número decimal, vamos obter o número 110, isto é, um inteiro de 4 bytes. Perceba que aconteceu um Redução de bytes, mas mesmo assim continuamos com nosso dado coerente.

5) Podemos melhorar ainda mais a nossa compressão.

- 110 corresponde ao caractere n na Tabela ASCII.
- Então, se antes precisávamos de 4 bytes para armazenar o 110, agora vamos precisar de somente 1 byte para escrever o 'n' no arquivo comprimido.

∴ ocorreu uma redução dos números de bytes para representar AAB  
1 byte 1 byte 1 byte  
3 bytes

E agora, o caractere 'n' ocupa somente 1 byte.

6) Para descomprimir, basta fazer o processo contrário.



n → é o 110 em ASCII → 110 em binário é 1101110  
7 bits

completo os bits restantes 01101110 → 01101110  
3 bits A A B



.txt  
descomprimido

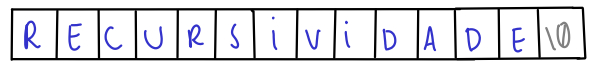
AAB ocupa 3 bytes

Esse é um processo resumido do funcionamento geral do programa!

\* Criação da  de Huffman:

1) Ex: possui a palavra RECURSIVIDADE

2) Palavra é colocada em um VETOR []:



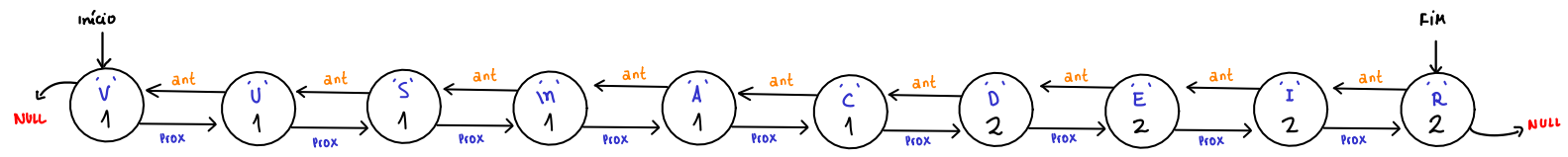
3) A função `ocorrencia_caractere(...)` conta a ocorrência de cada caractere no vetor e adiciona no vetor:


`lista_freq_caracter [0-255]`

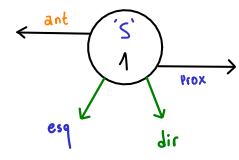
Em seguida, cria um Nó que possui o caractere e sua frequência e add na LDE.

OBS: a função `insere_no_na_lista(...)` sempre add na LDE deixando ela ordenada.

tamanho = 10

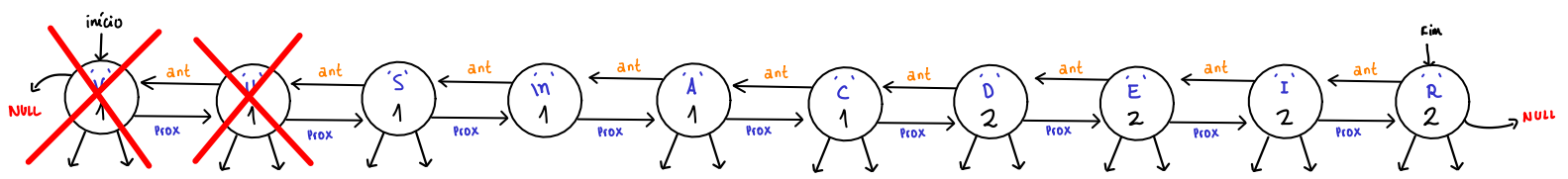


porém, minha LDE possui a seguinte particularidade: Além dos mecanismos de ligação `prox` e `ant`, minha LDE possui o mecanismo de ligação `esq` e `dir` (são importantes para a Montagem da ). Então imagine que cada Nó da lista duplamente encadeada é da seguinte forma:



## Montagem: Algoritmo de Huffman

1) Removemos 2 nós que possuem menores frequência:



```
printf("..... PROCESSO DE CONSTRUÇÃO DA ÁRVORE ..... \n");
imprimir_lista();
// Algoritmo de construção da Árvore de Huffman:
while (tam > 1) {
    // 2°
    filho0 = remover_menor();
    filho1 = remover_menor();

    // 3° e 4°
    pai = criando_no_arvore('+', filho0 -> freq, filho1 -> freq);
    pai -> esq = filho0;
    pai -> dir = filho1;

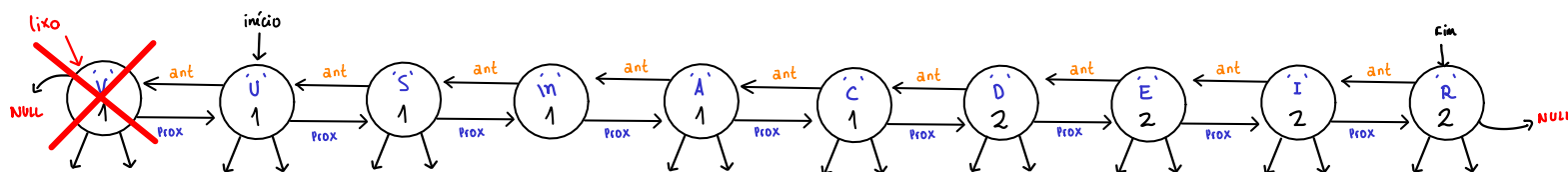
    // 5°
    insere_no_na_lista('+', 0, pai);

    printf("\n");
    imprimir_lista();
    printf("\n");
}

pai = remover_menor();
return pai; // Raiz da Árvore de Huffman.
```

Removendo os 2 nós de menores frequências.

Remoção - demonstração:



```

No* remover_menor () {
    No* lixo;

    lixo = inicio;
    inicio = inicio -> prox;

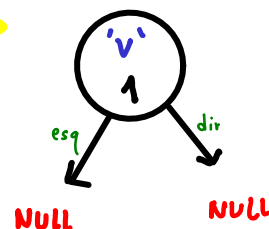
    // Realizando cópia do nó que será removido:
    No* menor = (No*) malloc (sizeof(No));
    menor -> freq = lixo -> freq;
    menor -> c = lixo -> c;

    // Se o Nó que desejo remover, tem filhos:
    if (lixo -> esq != NULL || lixo -> dir != NULL) {
        menor -> esq = lixo -> esq;
        menor -> dir = lixo -> dir;
    }
    // Se o Nó que desejo remover, não possui filhos.
    else {
        menor -> esq = NULL;
        menor -> dir = NULL;
    }

    tam--;
    free(lixo);

    return menor;
}

```



X desalocando nó da

2) Após Remover os 2 nós, que possuíam as menores frequências precisamos criar e copular o Nó pai e em seguida ligar os filhos ao pai:

```

printf("..... PROCESSO DE CONSTRUÇÃO DA ÁRVORE ..... \n");
imprimir_lista();
// Algoritmo de construção da Árvore de Huffman:
while (tam > 1) {
    // 2°
    filho0 = remover_menor();
    filho1 = remover_menor();

    // 3°
    pai = criando_no_arvore('+', filho0 -> freq, filho1 -> freq);
    pai -> esq = filho0;
    pai -> dir = filho1;

    // 5°
    insere_no_na_lista ('+', 0, pai);

    printf("\n");
    imprimir_lista();
    printf("\n");

    pai = remover_menor();

    return pai; // Raiz da Árvore de Huffman.
}

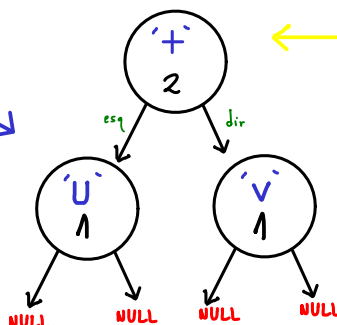
```

```

No* criando_no_arvore (char ch, int freq0, int freq1) {
    No* pai = (No*) malloc (sizeof(No));
    pai -> esq = NULL;
    pai -> dir = NULL;
    pai -> prox = NULL;
    pai -> ant = NULL;
    pai -> freq = freq0 + freq1;
    pai -> c = ch;

    return pai;
}

```



• OBS: frequência do pai é igual a soma das frequências dos 2 nós removidos.

3) Agora preciso ir conectando o Nó pai aos outros nó, para assim ir criando a  de Huffman:

```
printf("..... PROCESSO DE CONSTRUÇÃO DA ÁRVORE ..... \n");
imprimir_lista();
// Algoritmo de construção da Árvore de Huffman:
while (tam > 1) {
    // 2°
    filho0 = remover_menor();
    filho1 = remover_menor();

    // 3° e 4°
    pai = criando_no_arvore('+', filho0 -> freq, filho1 -> freq);
    pai -> esq = filho0;
    pai -> dir = filho1;

    // 5°
    insere_no_na_lista ('+', 0, pai);

    printf("\n");
    imprimir_lista();
    printf("\n");
}

pai = remover_menor();

return pai; // Raiz da Árvore de Huffman.
```

```
// Procedimento: responsável por inserir um Nó em uma Lista Duplamente Encadeada, de Maneira 'Ordenada' e crescente.
void insere_no_na_lista (char c, int freq, No* x) {
    No* novo;

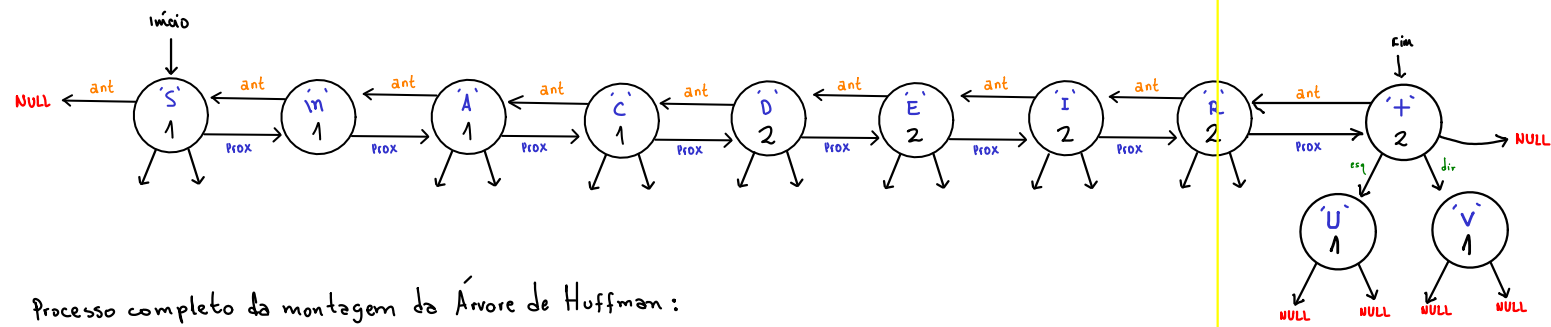
    if (x == NULL) {
        novo = (No*) malloc (sizeof(No));
        novo -> freq = freq;
        novo -> c = c;
        novo -> prox = NULL;
        novo -> ant = NULL;
        novo -> esq = NULL;
        novo -> dir = NULL;
    }
    else {
        novo = x;
    }

    // 1° CASO: LDE está vazia.
    if (inicio == NULL) {
        inicio = novo;
        fim = novo;
    }

    // 2° CASO: Nó que desejo adicionar é maior que o Último Nó da LDE.
    else if (novo -> freq >= fim -> freq) {
        fim -> prox = novo;
        novo -> ant = fim;
        fim = novo;
    }

    // 3° CASO: Nó que desejo adicionar é menor que o Primeiro Nó.
    else if (novo -> freq <= inicio -> freq && tam >= 1) {
        novo -> prox = inicio;
        inicio -> ant = novo;
        inicio = novo;
    }

    // 4° CASO: Nó que desejo adicionar deve ficar entre DOIS Nós.
    else {
        No* aux = inicio;
        while (aux != NULL) {
            if (novo -> freq >= aux -> freq && novo -> freq <= aux -> prox -> freq) {
                aux -> prox = novo;
                novo -> ant = aux;
                novo -> prox = aux -> prox;
                aux -> prox = novo;
                break;
            }
            aux = aux -> prox;
        }
    }
    tam++;
}
```



Processo completo da montagem da Árvore de Huffman:

```
paulo@maquinaPh:~/Área de Trabalho/huffman-versão-Final$ ./huffman c arquivo2.txt comprimido.code
..... PROCESSO DE CONSTRUÇÃO DA ÁRVORE .....
V --> 1 vez.
U --> 1 vez.
S --> 1 vez.
--> 1 vez.
A --> 1 vez.
C --> 1 vez.
D --> 2 vezes.
E --> 2 vezes.
I --> 2 vezes.
R --> 2 vezes.
S --> 1 vez.
--> 1 vez.
A --> 1 vez.
C --> 1 vez.
D --> 2 vezes.
E --> 2 vezes.
I --> 2 vezes.
R --> 2 vezes.
+ --> 2 vezes.
A --> 1 vez.
C --> 1 vez.
D --> 2 vezes.
E --> 2 vezes.
I --> 2 vezes.
R --> 2 vezes.
+ --> 2 vezes.
D --> 2 vezes.
E --> 2 vezes.
I --> 2 vezes.
R --> 2 vezes.
+ --> 2 vezes.
+ --> 2 vezes.
```

```
I --> 2 vezes.
R --> 2 vezes.
+ --> 2 vezes.
+ --> 2 vezes.
+ --> 4 vezes.
+ --> 4 vezes.
+ --> 2 vezes.
+ --> 2 vezes.
+ --> 4 vezes.
+ --> 4 vezes.
+ --> 4 vezes.
+ --> 6 vezes.
+ --> 6 vezes.
+ --> 8 vezes.
+ --> 14 vezes.
Conteúdo presente no arquivo: RECURSIVIDADE
..... CADEIA DE BITS GERADO PELA ÁRVORE DE HUFFMAN .....
01001100010010000101100110110110100000
R E C U R S I V
..... TABELA DE CÓDIGO DE CARACTERE .....
----> 0000
A ----> 111
C ----> 110
D ----> 101
E ----> 100
I ----> 011
R ----> 010
S ----> 001
U ----> 0010
V ----> 0011
```

cadeia de bits gerada pela



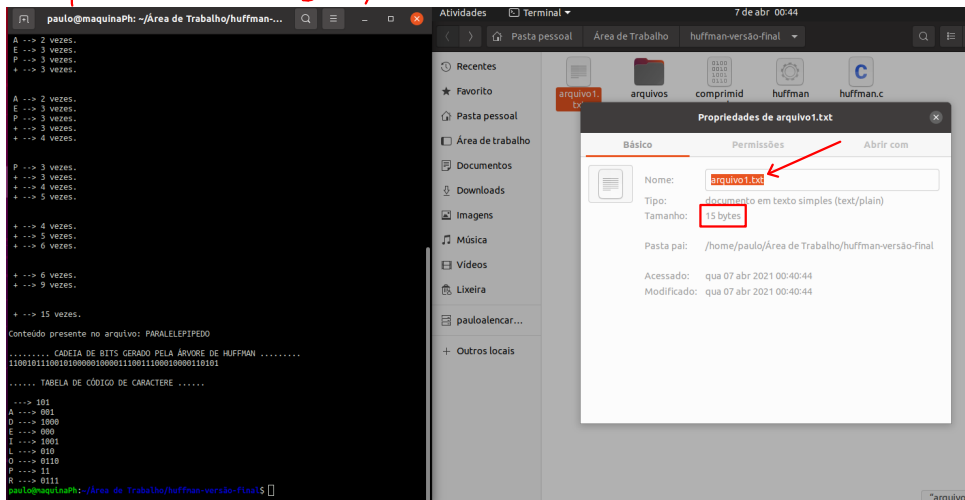
## \* Análise dos testes :

1º arquivo -

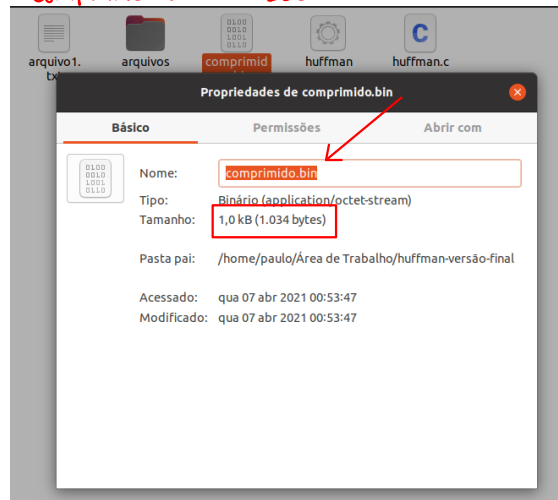
OBS: a partir do arquivo2.txt eu coloquei textos aleatórios gerado pelo Gerador de textos Lorem Ipsum.

não ocorreu compressão porque o cabeçalho ocupa 1028 bytes  
 $\therefore 1034 - 1028 = 6 \text{ bytes}$   
 bytes ocupados pelo texto.

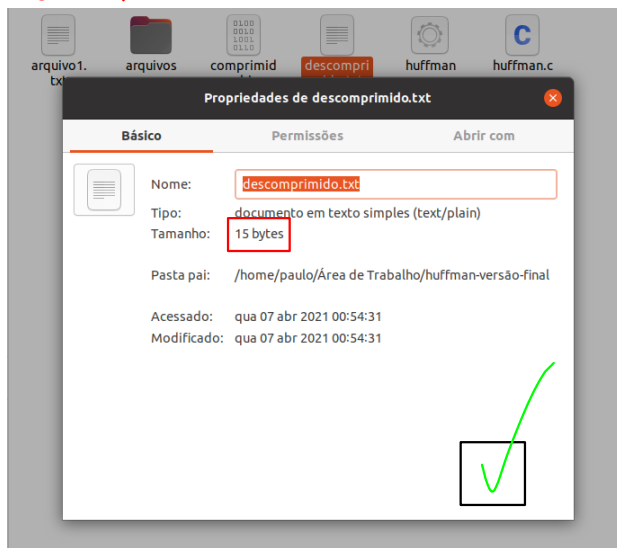
arquivo1.txt → 15 bytes



Comprimido.bin 1034 bytes



descomprimido.txt → 15 bytes

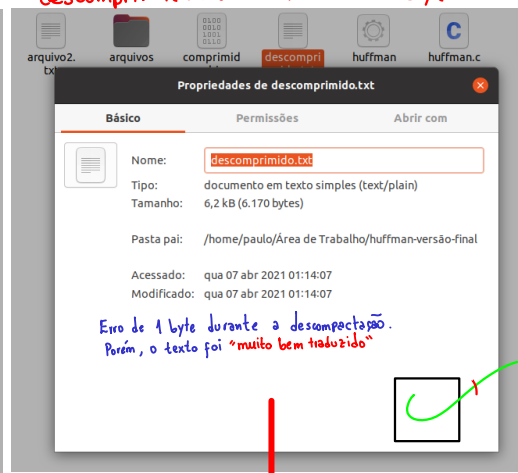
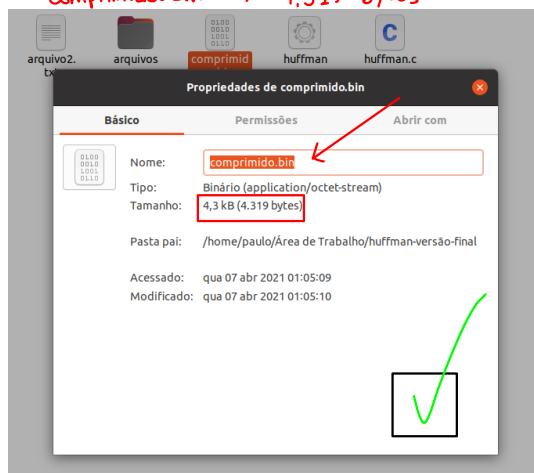
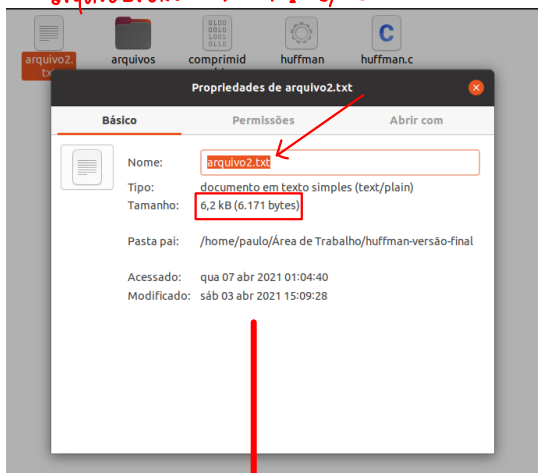


2º arquivo -

arquivo2.txt → 6.171 bytes

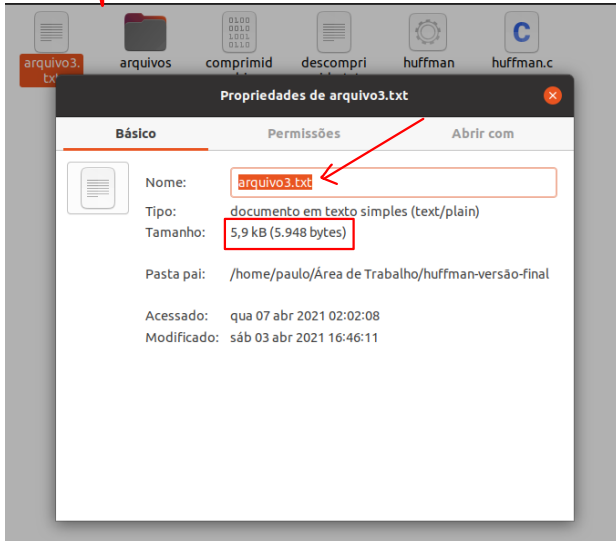
comprimido.bin → 4.319 bytes

descomprimido.txt → 6.170 bytes

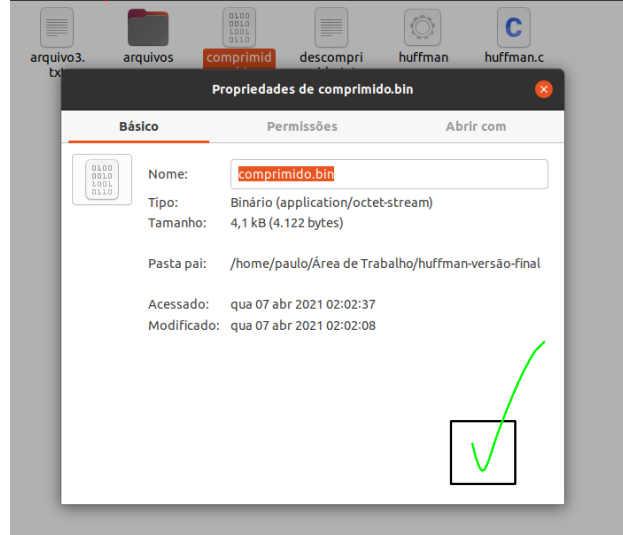


3º arquivo -

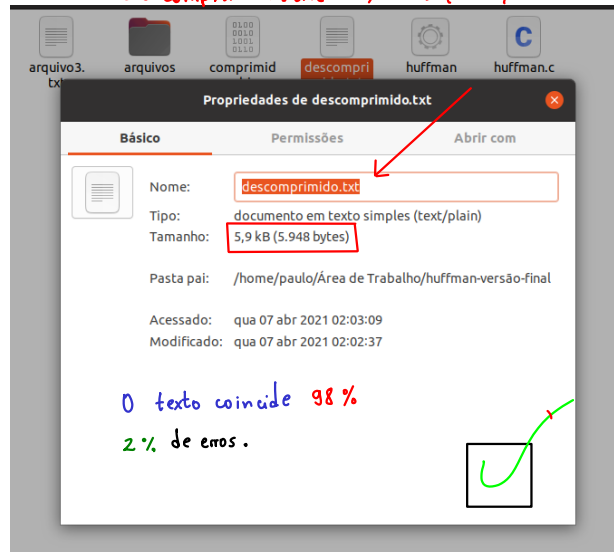
arquivo3.txt → 5.948 bytes



comprimido.bin → 4.122 byte



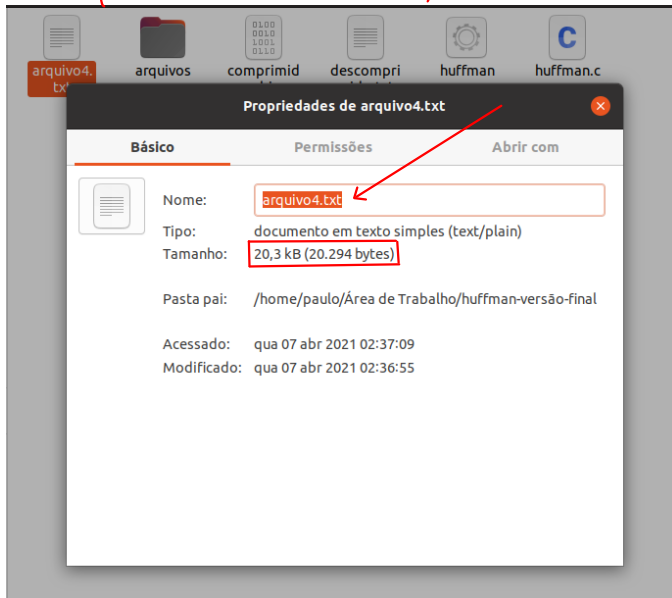
descomprimido.txt → 5.948 bytes



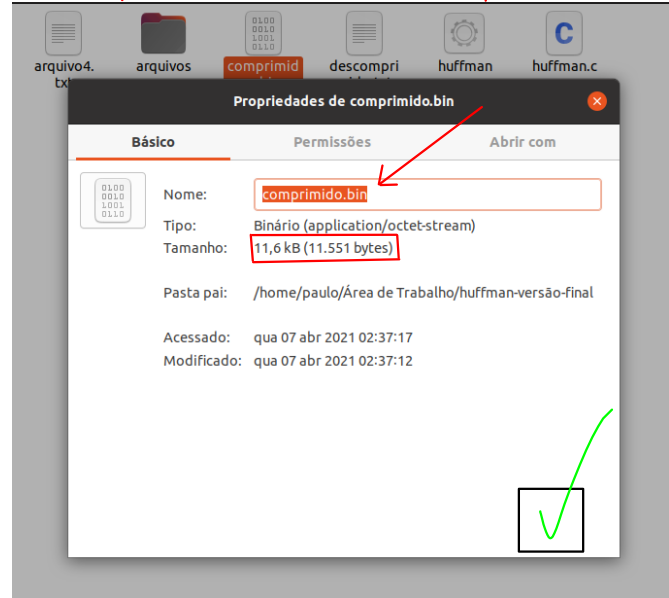


4 = arquivo -

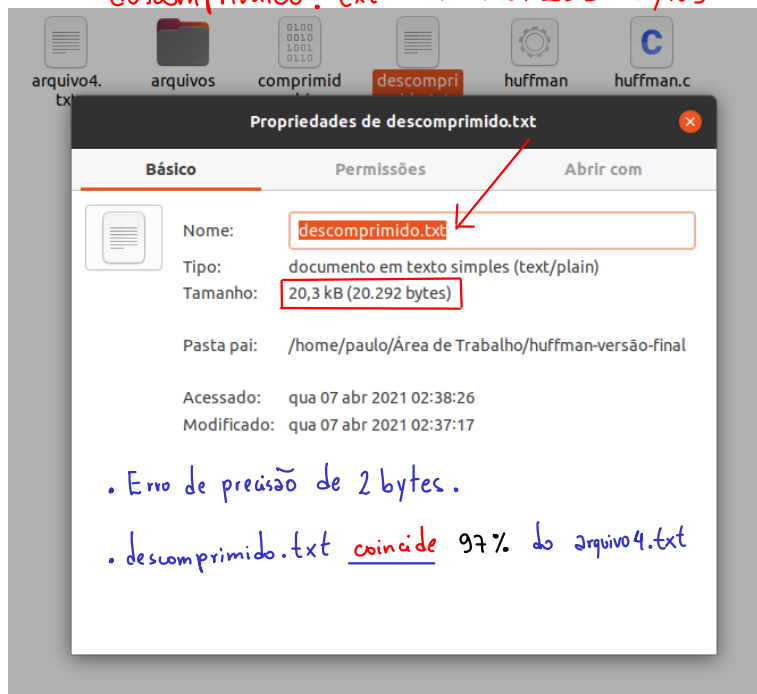
arquivo4.txt → 20.294 bytes



comprimido.bin → 11.551 bytes



descomprimido.txt → 20.292 bytes



5 = arquivo -

