**Exercício-Programa 2 de**

**Algoritmos e Estrutura de Dados I**

**Fernanda Moraes Bernardo**

**Nº USP: 7971991**

**Turma: 94**

**Professor: Clodoaldo**

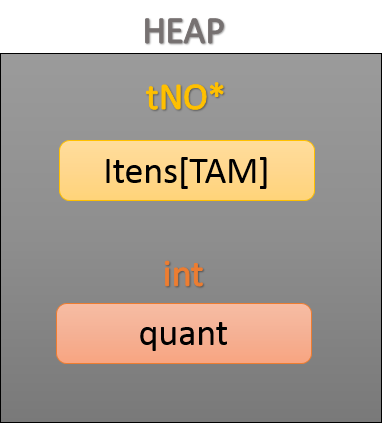
**Introdução**

Para resolver o problema de arquivos de texto que ocupam muito espaço na memória, é necessário a compactação desse arquivo. Com isso, esse EP exerce essa função, além da de descompactação. De maneira geral, esse programa é capaz de compactar e descompactar um arquivo (da extensão .txt para .bin e ao contrário), utilizando um algoritmo de Huffman. Esse algoritmo codifica um alfabeto de uma mensagem em função da frequência de cada caractere aparecido na mensagem.

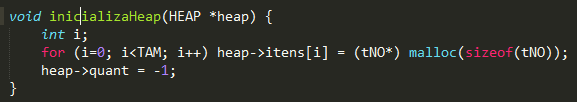
**Resumo**

O código de codificação/decodificação funciona da seguinte forma: para a codificação, um arquivo de texto é lido, dele cada caractere é adicionado em uma estrutura de heap (arranjo de tNO). Em cada posição do arranjo tem um caractere e sua frequência no texto. Após essa etapa, o heap (arranjo) é ordenado. A partir disso, dois nós são retirados desse arranjo e unidos em apenas um tNO, suas frequências são somadas e os filhos desse novo tNO são os que foram retirados do heap. Esse novo tNO é novamente adicionado no heap de forma ordenada. Esse processo ocorre até que sobre apenas uma posição do arranjo. Essa posição contém uma árvore, e esse tNO será adicionado na raiz de uma árvore. O próximo passo é a codificação, para cada filho à esquerda um ‘0’ é adicionado e para cada filho à direita um ‘1’ é adicionado. Após feita a codificação, é o momento de voltar à mensagem lida do arquivo e trocar cada caractere por seu respectivo código. O último passo é gravar a árvore utilizada e o código gerado em um arquivo binário. O processo de decodificação é inverso à esse.

**Estruturas**

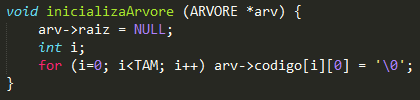


**Métodos**

* void inicializaHeap (HEAP \*heap)

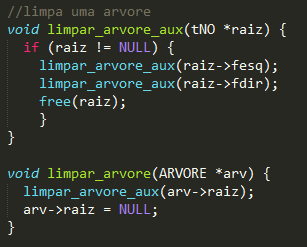
Esse método inicializa todos os componentes do HEAP. Aloca espaço para cada tNO do arranjo (itens) e coloca quant como -1, ou seja, não tem nenhum elemento no HEAP.

* void inicializaArvore (ARVORE \*arv)



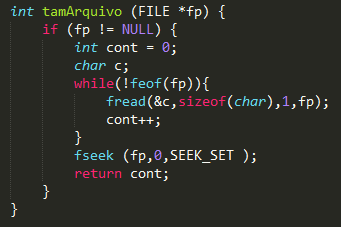
Esse método inicializa todos os componentes da ARVORE. Coloca a raiz apontando para NULL (não tem nada na arvore) e coloca um caracter final em cada linha do arranjo código.

* void limpar\_arvore(ARVORE \*arv)



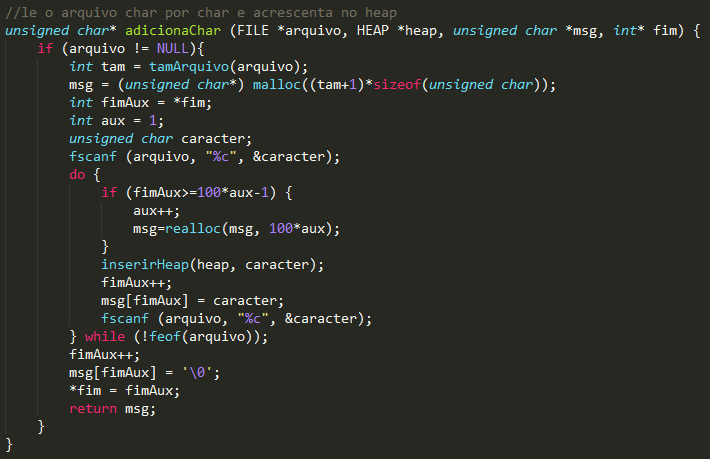
Esse método usa recursão para limpar (free) todos os nós da árvore e apontar a raiz para NULL

* int tamArquivo (FILE \*fp)

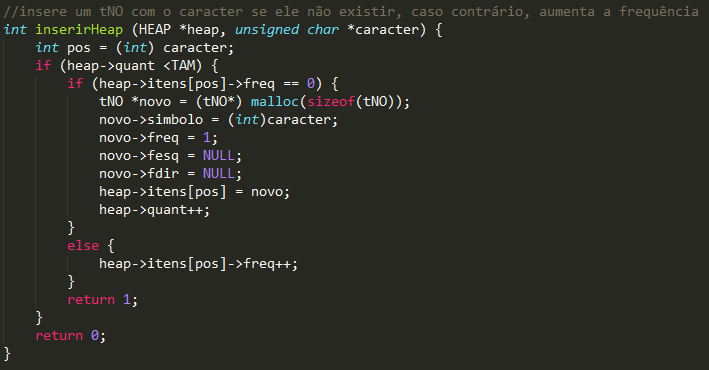


A partir de um arquivo, esse método é capaz de contar quantos caracteres tem nele. Logo após a contagem, coloca o ponteiro novamente na posição 0 do arquivo.

* unsigned char\* adicionaChar (FILE \*arquivo, HEAP \*heap, unsigned char \*msg, int\* fim)

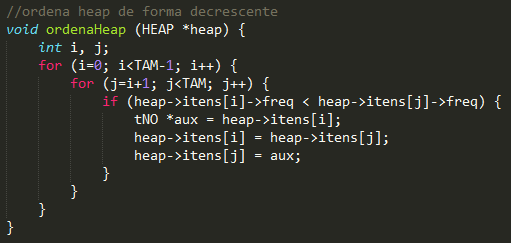


A partir de um arquivo, é calculado o número de caracteres dele (tamArquivo), que é usado para alocar espaço na memória para armazenar o texto desse arquivo. Após isso, lê caractere por caractere do arquivo e armazena em msg.

* int inserirHeap (HEAP \*heap, unsigned char \*caracter)

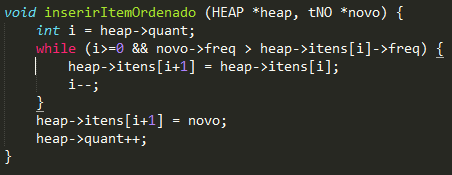
Esse método recebe um caractere para inserir no heap. Dessa forma verifica na posição do caractere (código binário do char em int) se ele já existe no heap. Se ele existir, adiciona um na frequência; caso contrário, cria um novo tNO e coloca ele no heap.

* void ordenaHeap (HEAP \*heap)



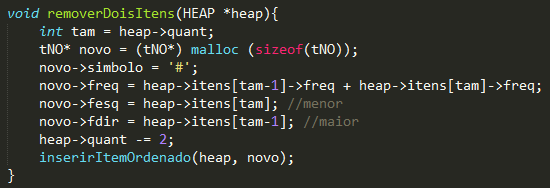
A partir de um heap, ordena ele de forma decrescente pelas frequências dos caracteres.

* void inserirItemOrdenado (HEAP \*heap, tNO \*novo)



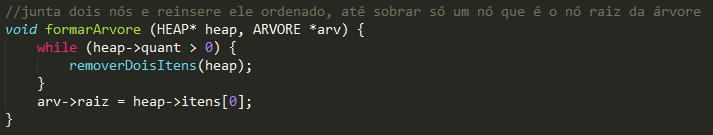
Insere um novo tNO de forma ordenada no heap (decrescente).

* void removerDoisItens(HEAP \*heap)



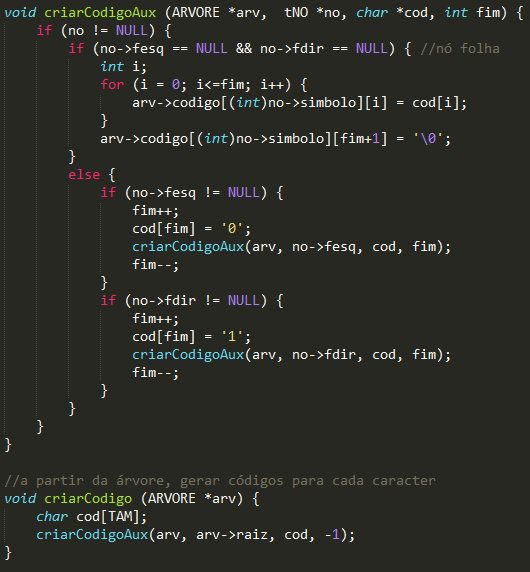
Aloca espaço para um novo tNO, com símbolo ‘#’. A frequência será a soma dos dois últimos tNO’s. O filho da direita (fdir) será o maior e o filho da esquerda (fesq) será o menor. Esses dois nós são removidos do heap e o novo nó será inserido de forma ordenada no heap

* void formarArvore (HEAP\* heap, ARVORE \*arv)



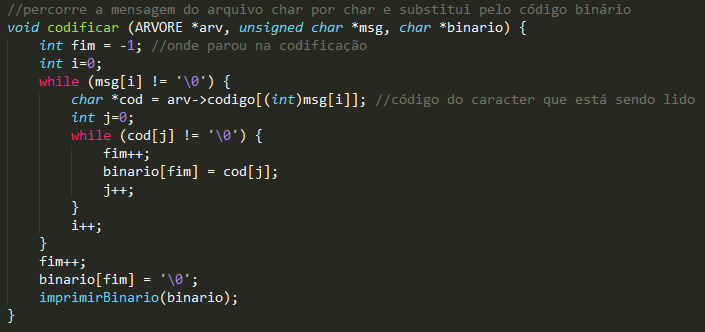
Faz um laço para remover os nós do heap e juntá-los em um, dois a dois, até sobrar apenas a primeira posição, que constitui a árvore. Portanto a raiz será a primeira posição do arranjo do heap.

* void criarCodigo (ARVORE \*arv)



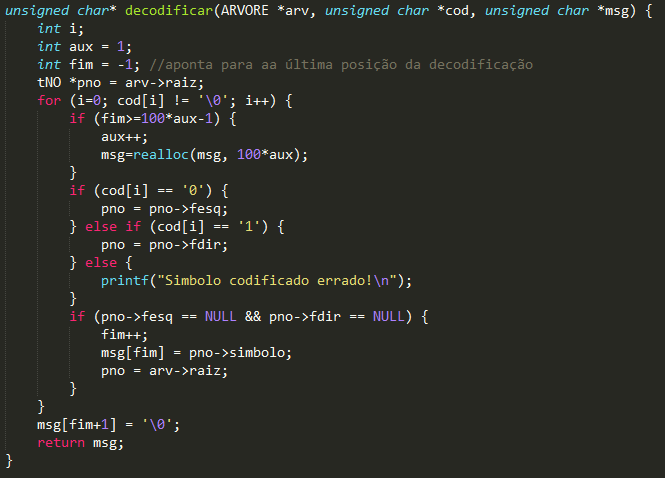
A partir de uma recursão, percorre toda a árvore, e cada vez que, na árvore, vai para a esquerda coloca ‘0’ no código e para a direita ‘1’. Dessa forma quando chega na folha, tem todo o caminho que fez para chegar até ela. Esse código é salvo na matriz da estrutura ARVORE.

* void codificar (ARVORE \*arv, unsigned char \*msg, char \*binario)

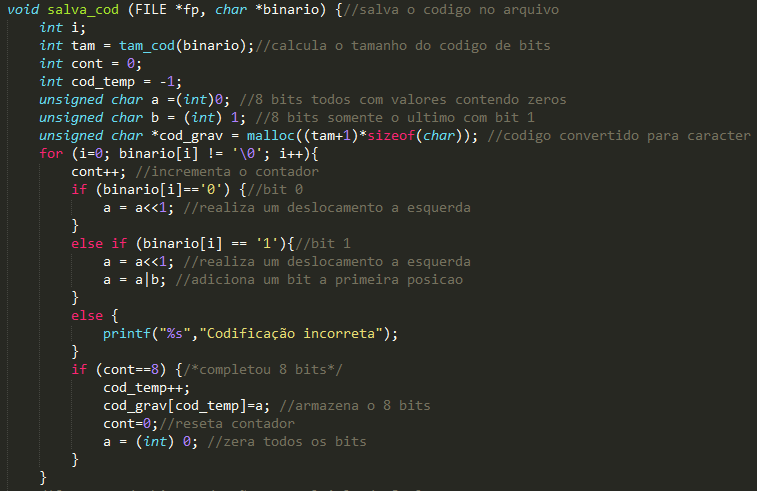


Esse método recebe a mensagem lida do arquivo, e percorre char por char e vai colocando em uma nova “string” o código em 0’s e 1’s correspondente a cada letra.

* unsigned char\* decodificar(ARVORE \*arv, unsigned char \*cod, unsigned char \*msg)

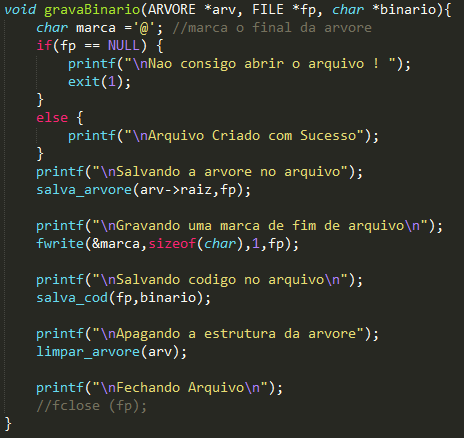
Esse código, a partir da árvore e da mensagem codificada (0’s e 1’s) substitui cada código binário na letra correspondente, formando a mensagem novamente.

* void salva\_cod (FILE \*fp, char \*binario)



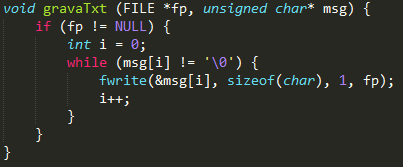
Esse método salva o código gerado em 0’s e 1’s em um arquivo binário (.bin)

* void gravaBinario(ARVORE \*arv, FILE \*fp, char \*binario)

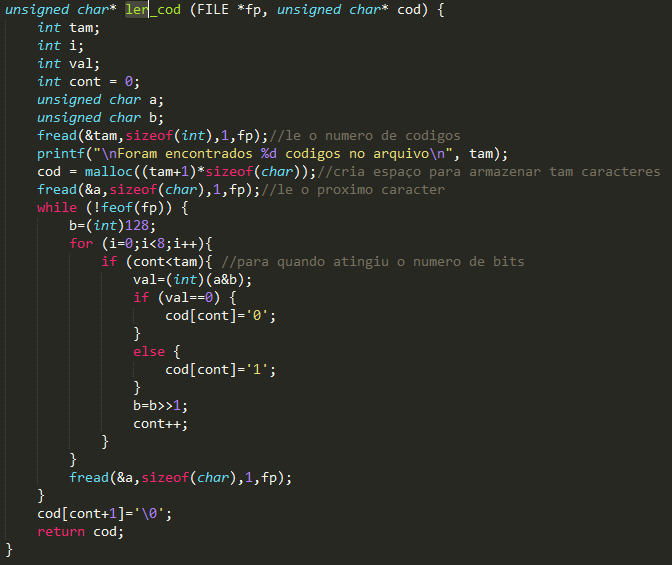


Esse método chama os métodos para gravar a árvore e o código em um arquivo binário.

* void gravaTxt (FILE \*fp, unsigned char\* msg)

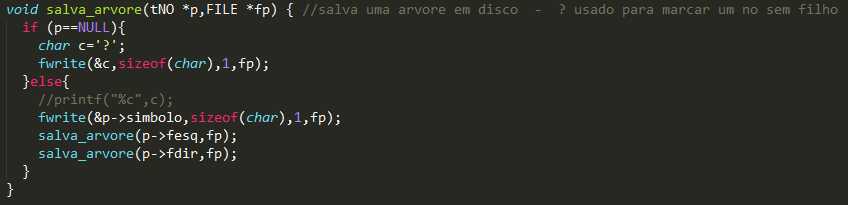


Esse método grava uma mensagem em um arquivo texto (.txt)

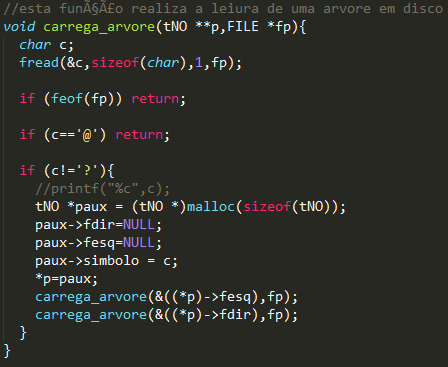
* unsigned char\* ler\_cod (FILE \*fp, unsigned char\* cod)

Esse método a partir de um arquivo, lê todos os caracteres binários de um arquivo e salva-os em uma “string”

* void salva\_arvore(tNO \*p,FILE \*fp)

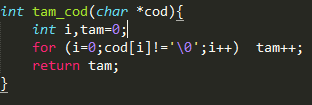


Esse método salva a estrutura da árvore em um arquivo binário.

* void carrega\_arvore(tNO \*\*p,FILE \*fp)

Esse método le uma árvore de um arquivo binário e armazena na estrutura correspondente

* int tam\_cod(char \*cod)



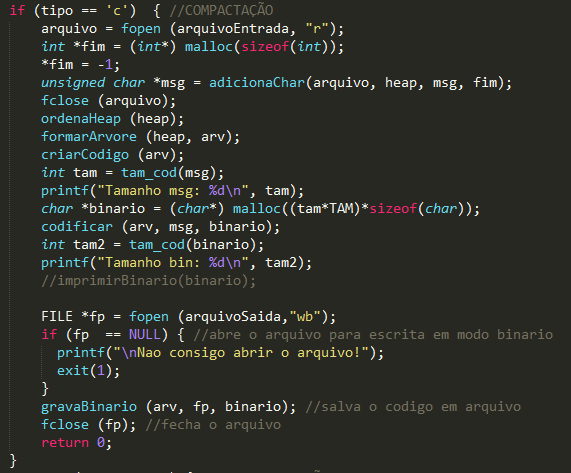
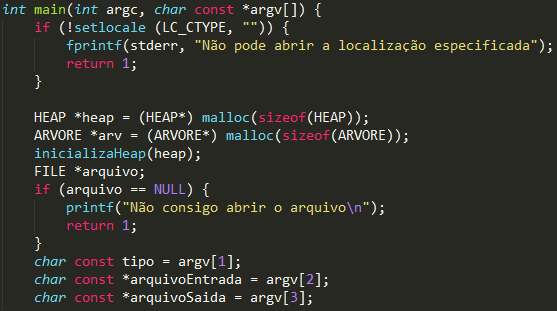
Esse método a partir de uma “string”, calcula o tamanho dela.

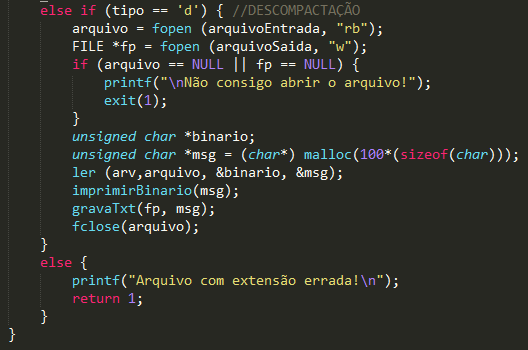
* void ler(ARVORE \*arv, FILE \*fp, unsigned char \*\*binario, unsigned char \*\*msg)



Esse método lê um arquivo binário, tanto a parte da árvore (localizada na primeira parte do arquivo) quanto a parte do código (localizada logo depois da árvore, separada por um caractere ‘@’). Armazenando cada parte em sua respectiva estrutura.

* int main(int argc, char const \*argv[])





No main, todas as estruturas são alocadas e inicializadas (através dos métodos de inicialização já mostrados). O que será executado (codificação ou decodificação) e os arquivos de entrada e saída são passados pelo usuário na hora da chamada do programa.

Se for uma compactação, o arquivo será aperto para leitura (‘r’) e serão chamados os métodos para leitura dos caracteres e para adicioná-los no heap. Logo depois, será chamado o método para ordenar o heap e formar a árvore. Após isso, ele irá codificar a mensagem e gravar no arquivo binário.

Se for uma descompactação, será o processo contrário. Primeiro, abrirá o arquivo binário para leitura, lerá o texto, decodificará e gravará em um arquivo de texto (.txt.)