# Relatório 2⁰ Trabalho de Estrutura de Dados

*Professora: Patrícia Dockhorn Costa*

*Período: 2022/1*

*Decodificando e codificando com Huffman*

*Alunos: Rhuan Garcia de Assis Teixeira e Gabriel Braga Ladislau*

# **Introdução:**

Para esse trabalho temos que implementar um compactador e um descompactador de arquivos.

Para isso teremos que mexer no baixo nível de bits, usando o TAD *bitmap*, com ele conseguimos fazer tudo necessário para trabalhar com bytes. Usamos o *doxygen* para documentar o código e conseguir criar um padrão de comentários. Fizemos comentários de todas as funções e estruturas presentes no nosso código. Usamos, também o serviço do *GitHub* para versionar nosso código e trabalhar com mais eficiência.

# **Implementação:**

Para começar, fizemos uma implementação sem seguir nenhum padrão, reutilizamos as árvores que implementamos nas aulas para fazer a árvore de Huffman.

Usamos a biblioteca *assert.h* para verificar se os ponteiros usados nas funções do programa estão devidamente alocados e diferentes de *NULL*. Caso esteja *NULL* a função *assert()* para a execução do programa com erro (0 = erro).

Começamos primeiro pelo compactador e seus TADs para depois seguir para o Descompactador.

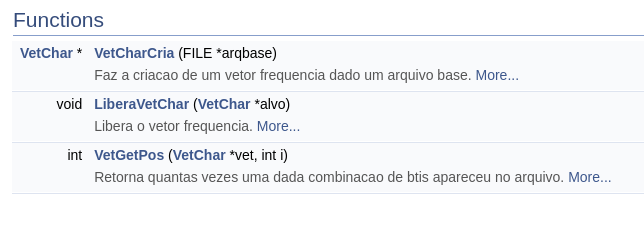
***TAD VetChar:***



Nesse TAD foi implementado o vetor de frequência de caracteres. Sua estrutura é basicamente composta por um vetor de 256 inteiros (uma posição para cada combinação possível em 8bits) em que armazenamos quantas vezes cada combinação apareceu durante o arquivo.

Ele possui as seguintes funções:

Dentre essas, vemos uma que vale ser melhor explicada neste relatório dada a maior complexidade :



VetCharCria

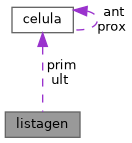
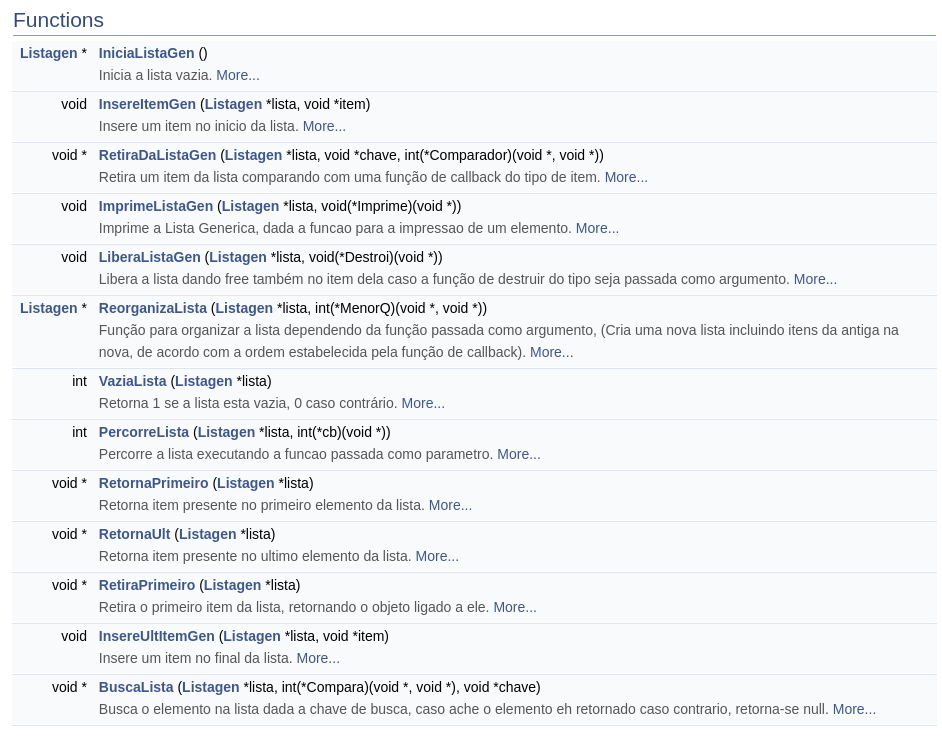


Nessa função alocamos a memória necessária para o vetor de frequência, inicializamos o mesmo com 0’s para todos valores e em seguida chamamos uma função (presente abaixo) para que o vetor seja preenchido com as devidas informações



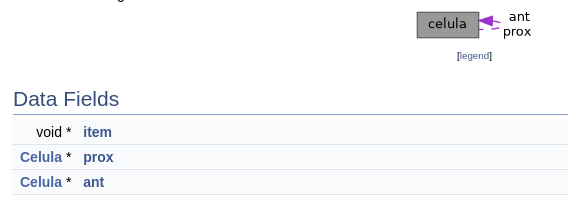
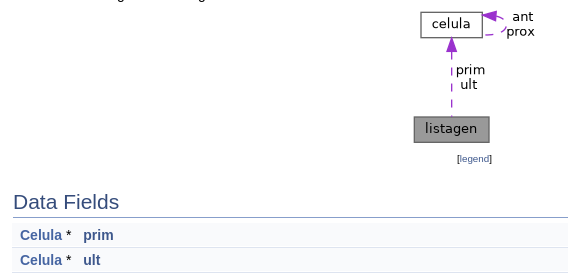
Nessa função, o arquivo é lido byte a byte, a cada leitura esse byte é convertido em um número usado como índice para nosso vetor de frequência. E então o contador dessa combinação de bits é incrementada.

***TAD ListaGen***



É uma simples implementação de lista genérica, podemos ver a estrutura de sua célula e sentinela abaixo

Célula: Sentinela:



Organização:

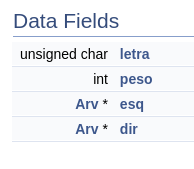
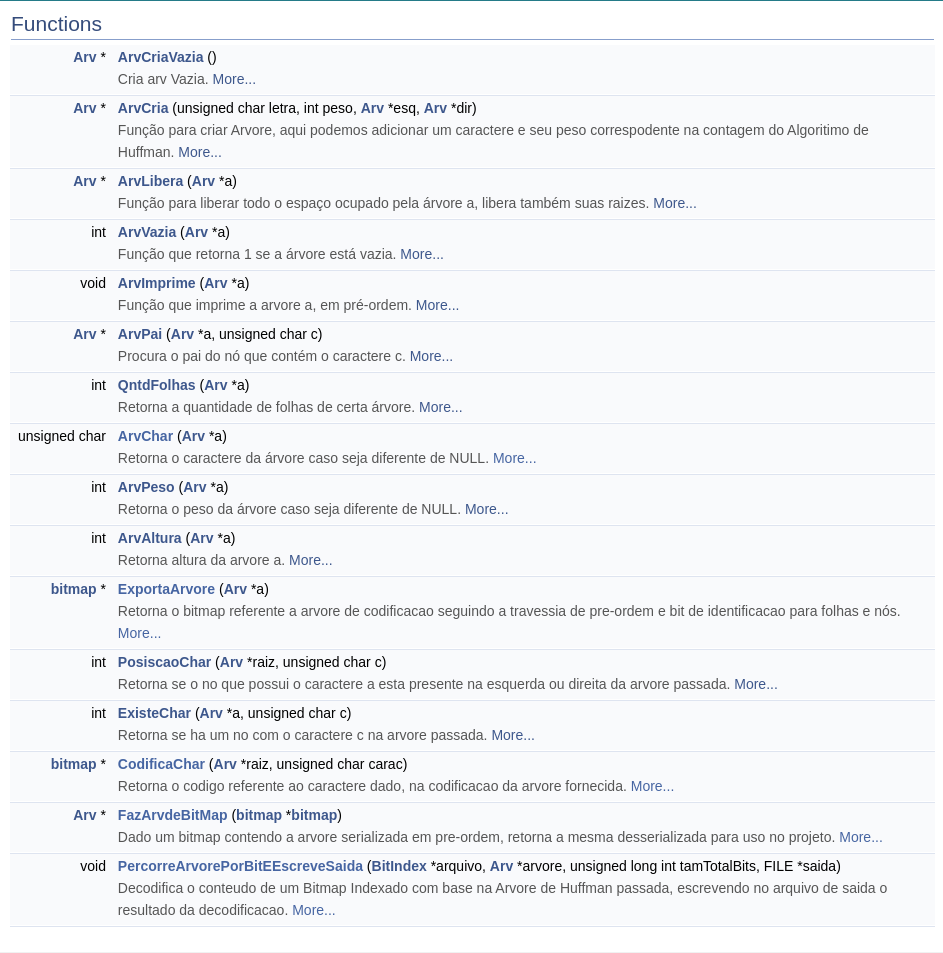
Ele possui as seguintes funções:

Dentre essas, deve ser melhor explicada a função ReorganizaLista, presente abaixo.



Nela é reorganizada a lista dada a função de comparação entre elementos (itens), um exemplo de uso é a reorganização de uma lista de arvores com base no peso das mesmas

***TAD Arvore:***



Para que seja possível implementar a árvore de codificação de Huffman foi necessário que a Arvore em si carregasse um inteiro representando o peso daquele caractere e um *char* que seria o caractere em questão, vemos a estrutura de arv abaixo:

Ele possui as funções abaixo:

Dentre elas, serão uteis maiores explicações sobre as seguintes:

ExportaArvore

Dada uma árvore binária, a função a serializa em pré-ordem colocando a saída em um bitmap

CodificaChar

Dada uma árvore de Huffman e um char presente na mesma,

FazArvdeBitMap

PercorreArvorePorBitEEscreveSaida

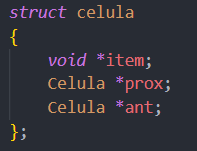
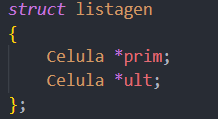
Para essa estrutura fizemos 3 funções privadas, *max* usada para calcular a altura da árvore, VarreduraArv que varre a arvore de Huffman para exporta-la para o cabeçalho do arquivo de saída e Recursiva que Percorre a arvore até chegar no caractere requerido adicionando 0's e 1's no bitmap para descobrir como tal caractere está codificado na arvore de Huffman. Segue uma foto abaixo das declarações, iremos mostrar suas definições depois:



Pulando as funções normais de arvore, que não fogem do padrão, seguimos para as mais complexas. Começando pela

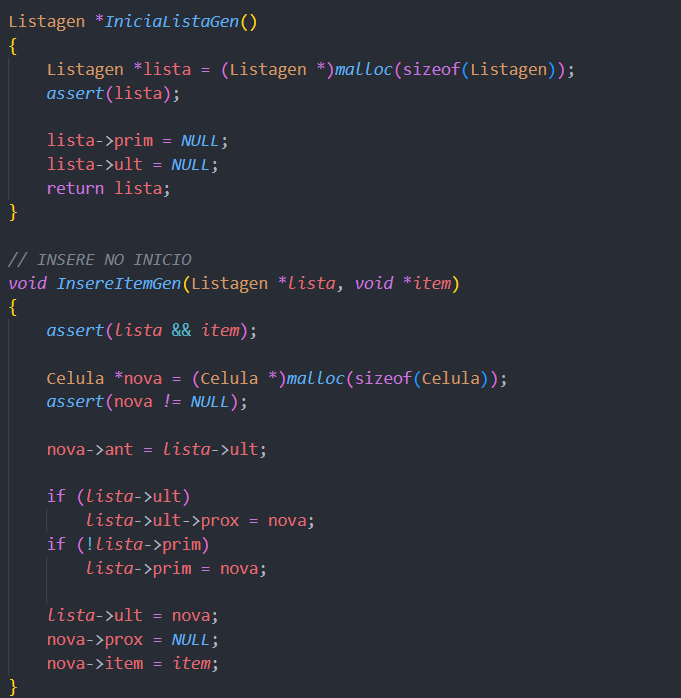
***TAD ListaGen e ListaArv:***

Esse TAD define uma lista genérica duplamente encadeada com sentinela, que no caso é usada para carregar as arvores que serão usadas para depois montar a arvore de *Huffman.* Segue a estrutura da sentinela e da célula da lista respectivamente:

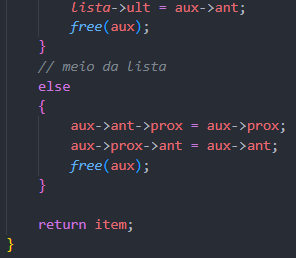
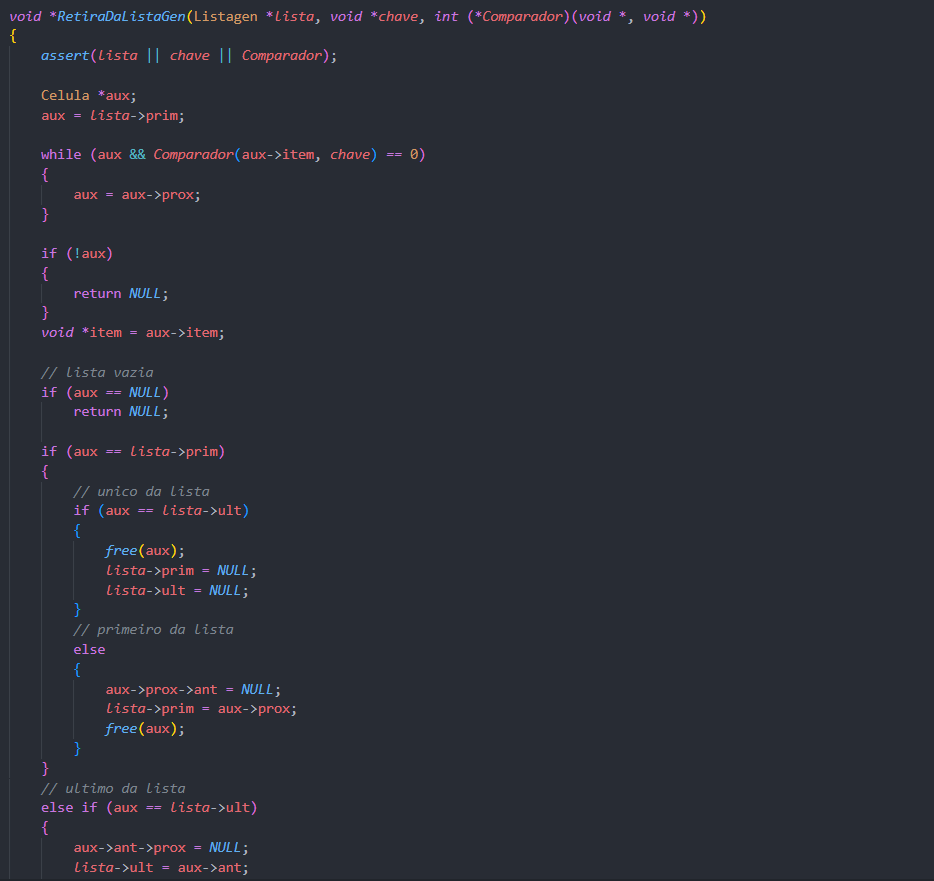


Irei simplesmente mostrar as funções da lista que são básicas, elas todas estão comentadas no .h, explicarei a fundo as mais especificas e complexas.

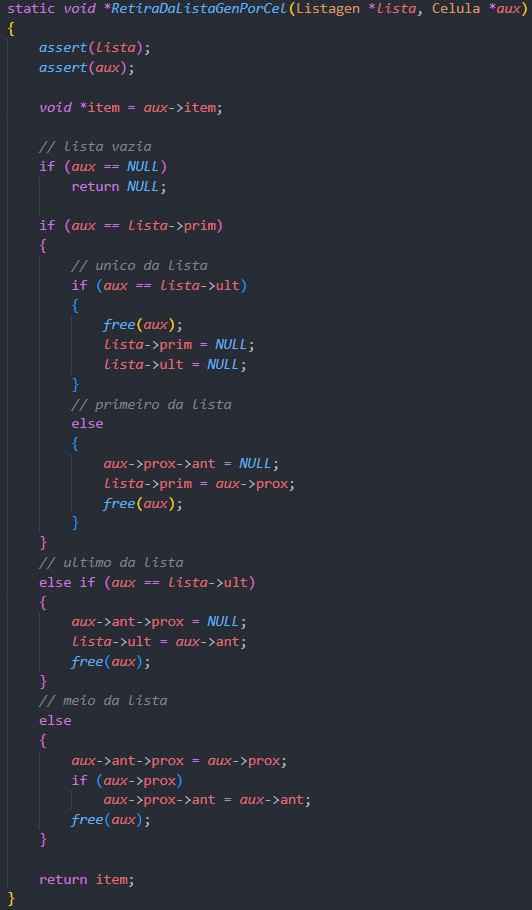
Inicia (cria) e Insere no Inicio:



Retira (Normal) comparador definido pelo tipo ListaArv:

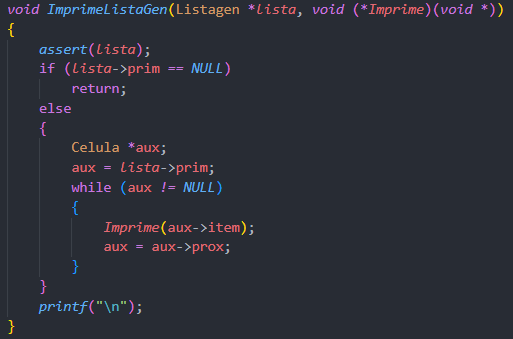


Retira por Célula ( função privada de uso especifico para reorganizar a lista):

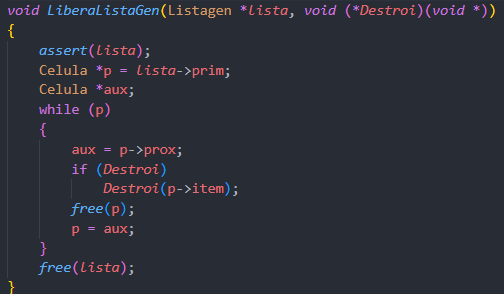


A função acima é usada para organizar a lista de forma que ela sempre recebera uma célula contida na lista assim evitando a busca que gasta tempo.

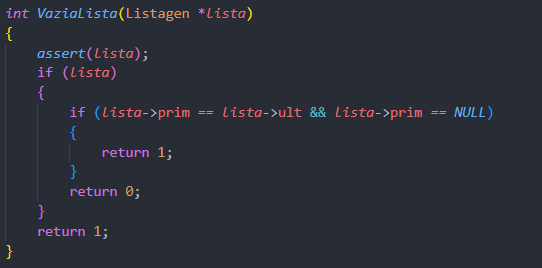
Imprime :



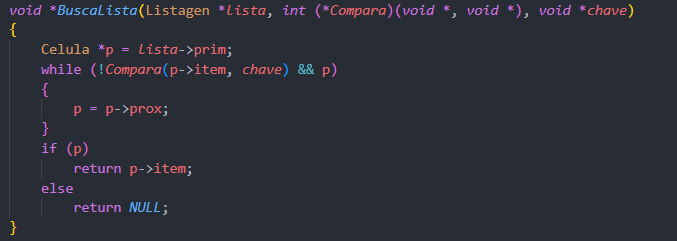
Libera:



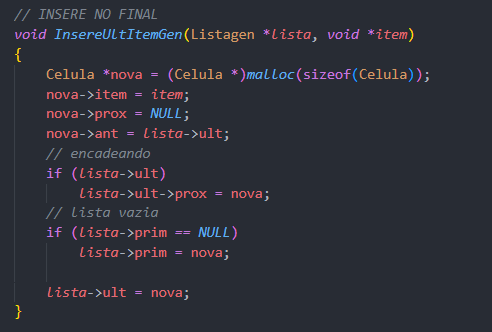
ListaVazia (1 se lista estiver vazia):



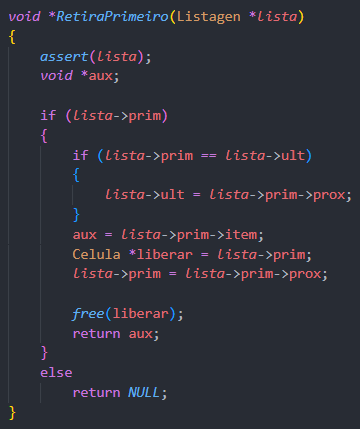
Busca Lista (retorna item):



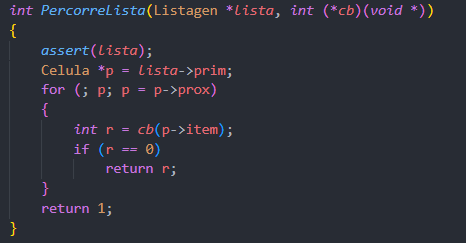
Outro insere porém nesse caso é no final:



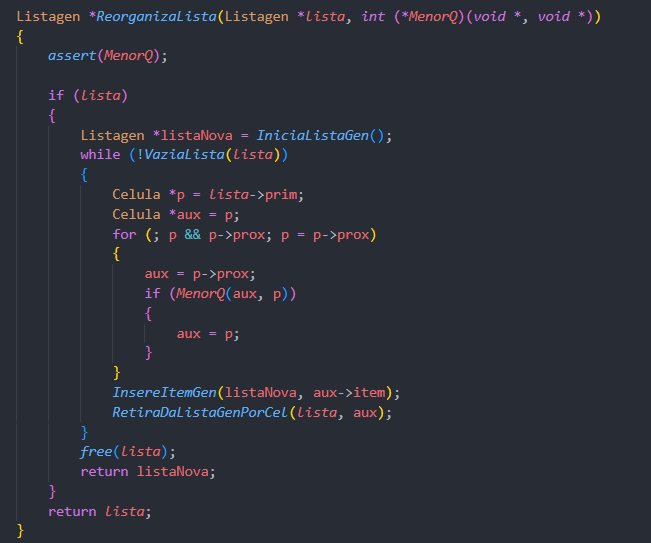
Outro Retira porém agora da primeira posição direto:



Uma percorre Lista:

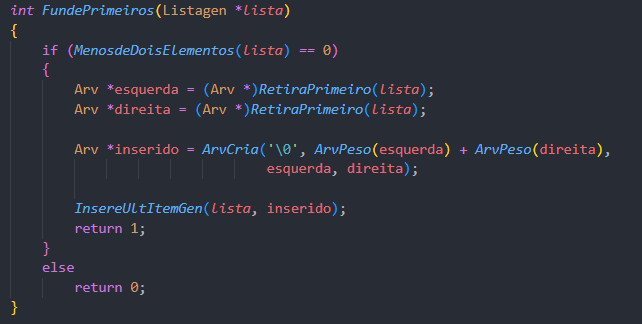


Agora chegamos na função Reorganiza Lista, aqui estamos implementando uma função para ajudar na hora de fazer a arvore de Huffman deixando os caracteres de menos peso mais ao final da lista e os de maior peso mais no inicio, usando uma função de callback, que no caso pode organizar a lista de qualquer maneira que está função definir:



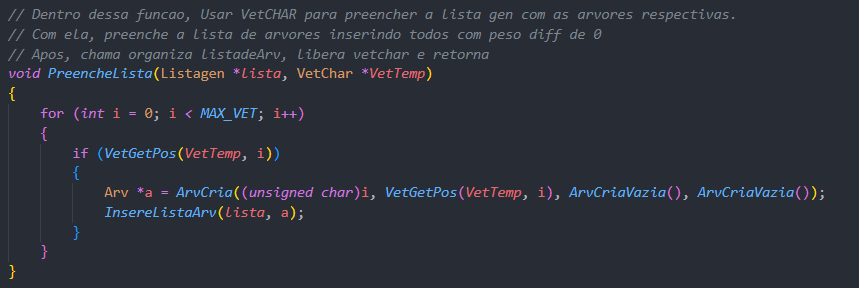
Vemos que uma nova lista é criada e para cada elemento da lista nós fazemos uma exaustiva comparação entre eles para inserir na nova lista da forma estabelecida pela função MenorQ passada como parâmetro. Depois a lista antiga é liberada e retornamos a nova.

Em ListaArv temos somente um tipo intermediário que faz com que a lista genérica carregue arvores em seus itens. Como muitas das funções são somente 1 linha, explicarei somente as mais complexas. Começamos pela FundePrimeiros que pega as 2 primeiras arvores da lista e as junta somando seus pesos e colocando um nó pai entre elas, colocando esse nó na ultima posição da lista logo depois , função que é usada no algorítimo de *Huffman*. Segue ela abaixo:

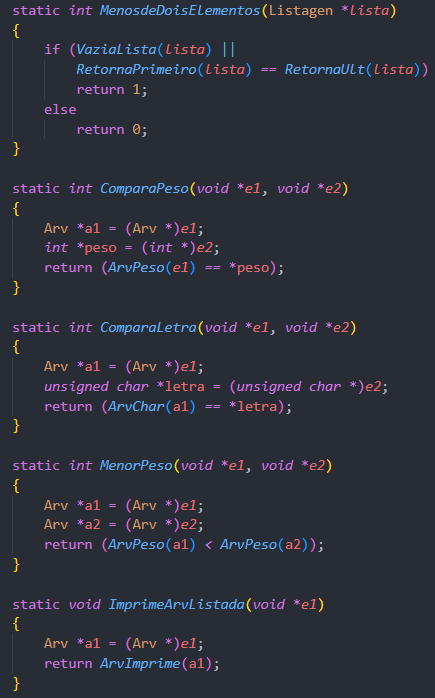


Se a lista tiver menos de 2 elementos a função acima para e retorna 2.

A função abaixo preenche a lista com arvores de acordo com o vetor de frequência, criando arvores com os respectivos caracteres caso seus pesos sejam diferentes de 0:

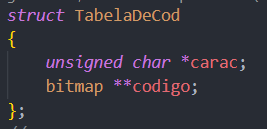


Abaixo seguem funções utilizadas como callback nas funções da lista genérica:



***TAD Codificador:***

O nome é codificador porém este TAD é responsável pela compactação do arquivo. Nele encontramos a estrutura tabela de codificação que carrega para cada *char* em um arquivo, seu correspondente em *bitmap* tendo já a arvore de huffman. Segue abaixo:



Para essa tabela fizemos funções de montar de acordo com a arvore de *Huffman* e o vetor de freq. e Libera para liberar memória.

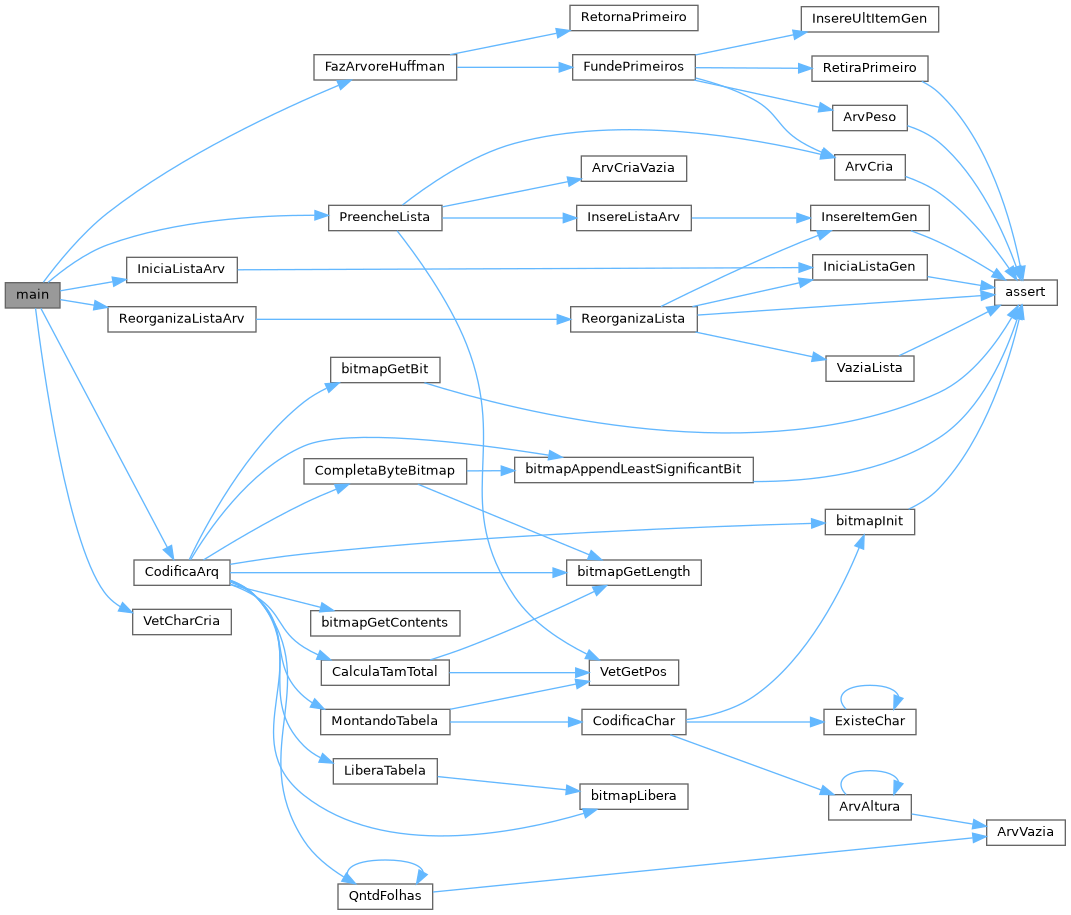


Como podemos ver para cada um dos caracteres a função CodificaChar pega e gera um código em bitmap (de acordo com a arvore de *Huffman*) para o caractere i caso seu peso seja diferente de 0, temos assim uma tabela que para cada caractere no vetor de *char* seu respectivo código corresponde *(mesmo index)* no vetor de ponteiros para bitmap.

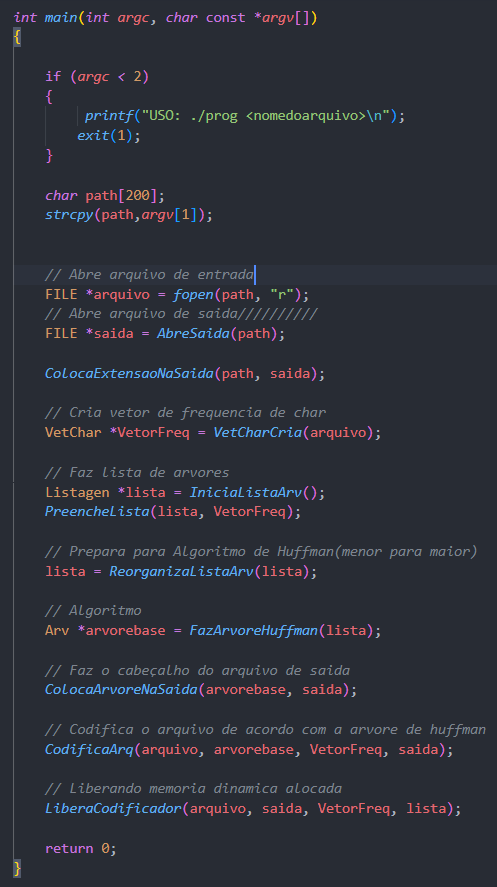
Voltando ao compactador em si temos essas funções privadas:



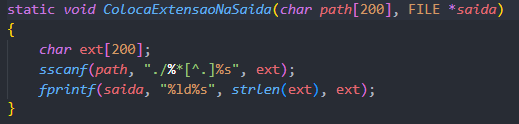
A *main* segue esse fluxograma:



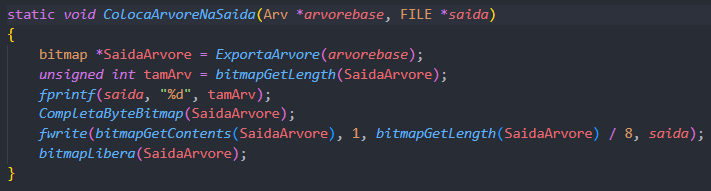
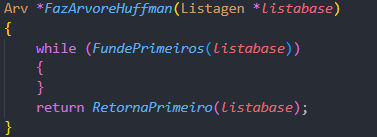
Olhando o código:



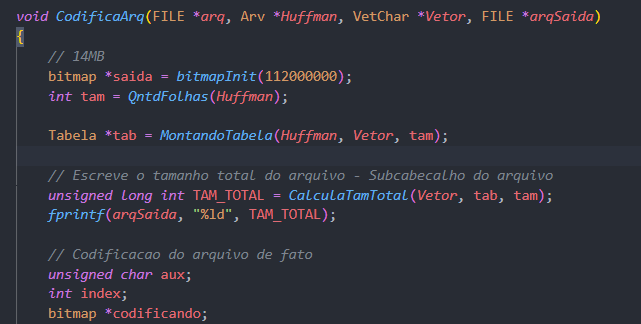
Vemos que a *main* começa verificando caso o programa foi chamado corretamente, depois passamos o caminho para o arquivo salvando no path, abrimos o arquivo de entrada e a saida, depois usamos uma função para colocar a extensão do arquivo e seu tamanho em bytes (4(tamanho) .txt por exemplo) na saída, segue a função:



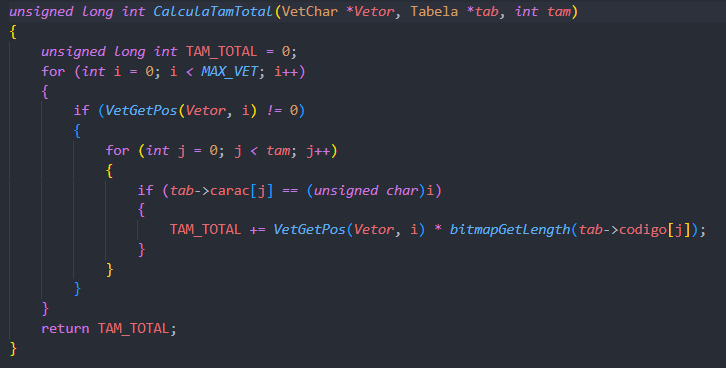
Após isso criamos o vetor de frequência de caracteres para formar a arvore de codificação, mas antes colocamos os caracteres presentes em arvores e inserimos na lista, que logo após é organizada para usar o algorítimo de *Huffman.* Com isso pronto montamos a arvore e colocamos ela na saída, colocamos seu tamanho em bits antes para saber quando parar de ler a arvore, completamos o bitmap com 0 e exportamos:



Depois começamos a realmente compactar o arquivo com a função *CodificaArq*.



Acima vemos que ela começa inicializando um *bitmap* de 14mb e pegamos o tamanho da tabela de codificação vendo quantas folhas a arvore de *Huffman* tem, montamos a tabela e calculamos quanto o arquivo vai ter de tamanho e escrevemos na saída. CalculaTamTotal:



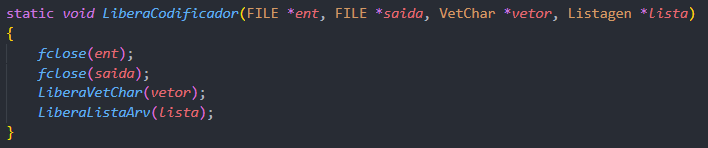
A função acima simplesmente calcula o tamanho do código gasto por cada caractere da tabela de codificação vezes seu peso para saber quantos bits no total serão usados.



O resto da função faz o seguinte, lê um byte, procura seu código na tabela e escreve no bitmap de saída, caso este mapa tenha ultrapassado 7mb nós escrevemos na saída e liberamos ele, assim evitamos de gastar muita memória usando o bitmap.

Depois que o arquivo todo foi compactado nós verificamos se ainda tem algo na saída completamos com 0 e escrevemos na saída. Liberamos o mapa a tabela e voltamos o ponteiro do arquivo para o inicio com *rewind*.

Voltando a *main* o próximo passo é liberar tudo:



Obs.: Como a lista ainda tem a raiz da arvore de *Huffman* liberando a lista nós liberamos a Arvore e assim termina o Compactador.