Relatório 2o 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*

**Sumário**

[Relatório 2o Trabalho de Estrutura de Dados 1](#__RefHeading___Toc6145_1733058768)

[Introdução: 3](#__RefHeading___Toc6147_1733058768)

[Implementação: 3](#__RefHeading___Toc6149_1733058768)

[TAD VetChar: 4](#__RefHeading___Toc6151_1733058768)

[VetCharCria 5](#__RefHeading___Toc6371_1733058768)

[TAD ListaGen 6](#__RefHeading___Toc6153_1733058768)

[TAD Arvore: 8](#__RefHeading___Toc6155_1733058768)

[ExportaArvore 9](#__RefHeading___Toc6373_1733058768)

[CodificaChar 10](#__RefHeading___Toc6375_1733058768)

[FazArvdeBitMap 10](#__RefHeading___Toc6377_1733058768)

[PercorreArvorePorBitEEscreveSaida 11](#__RefHeading___Toc6379_1733058768)

[TAD ListaArv 12](#__RefHeading___Toc6157_1733058768)

[FundePrimeiros 12](#__RefHeading___Toc6381_1733058768)

[PreencheLista 13](#__RefHeading___Toc6383_1733058768)

[TAD BitIndex 14](#__RefHeading___Toc6159_1733058768)

# 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.

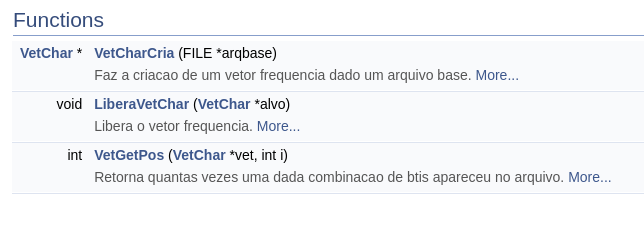
## TAD VetChar:

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, sendo ela:



### 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

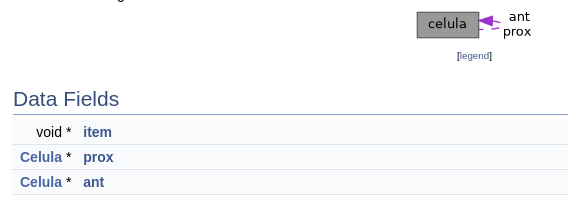
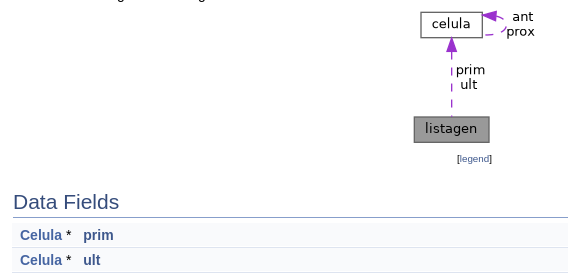


Nessa função, o arquivo é lido byte a byte, a cada leitura esse byte é convertido em um número usado como índice para nosso vetor de frequência. E então o contador dessa combinação de bits é incrementada.

## TAD ListaGen

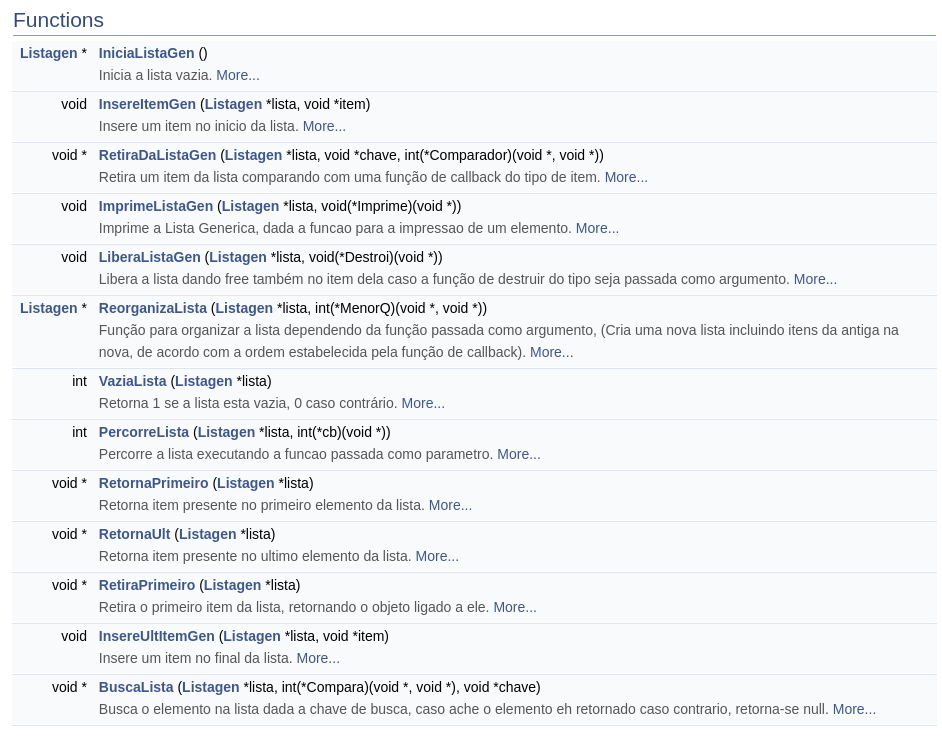
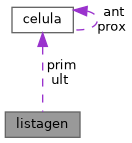
É uma simples implementação de lista genérica, podemos ver a estrutura de sua célula e sentinela abaixo

Célula: Sentinela:



Organização:

**Ele possui as seguintes funções:**



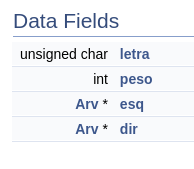
Dentre essas, deve ser melhor explicada a função **ReorganizaLista**, presente abaixo.

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



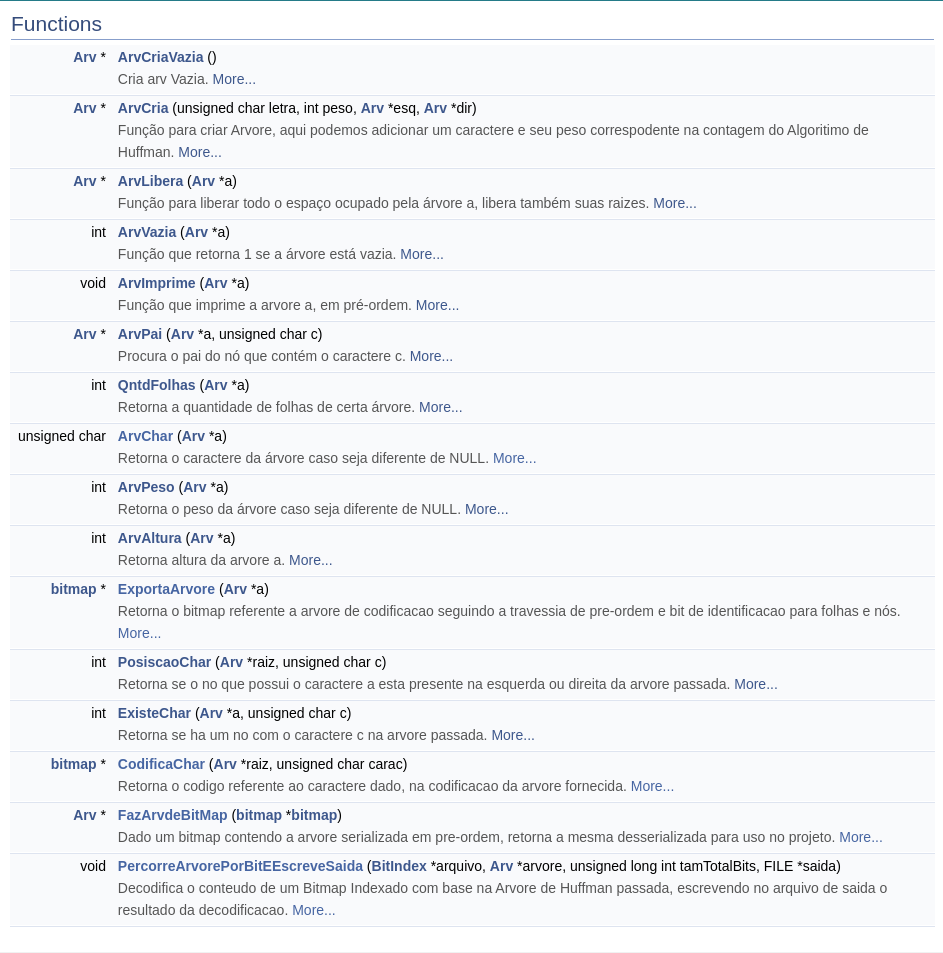
## TAD Arvore:

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:



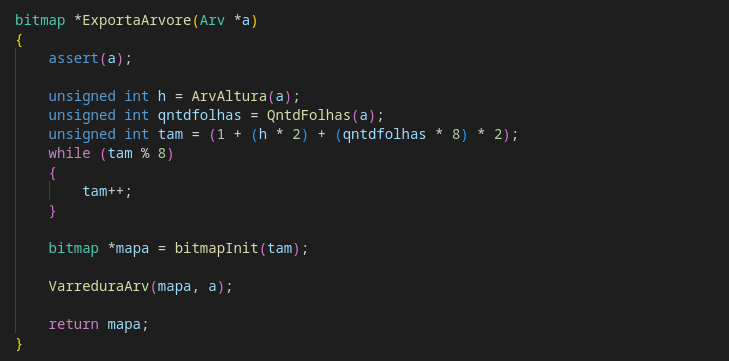
**Ele possui as funções abaixo:**

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. Nós são codificados como bit 0, folhas como bit 1, seguido do caractere presente na mesma.



### CodificaChar

Dada uma árvore de Huffman e um char presente na mesma, essa função verifica de que lado o caractere informado se encontra na arvore, escreve o dado obtido no bitmap e em seguida faz uma recursão até que chegue a folha onde o caractere está presente.



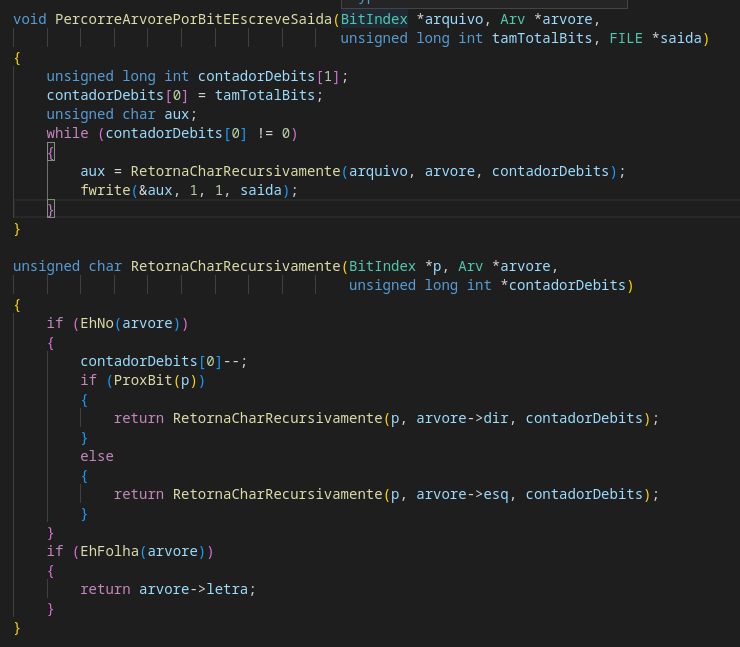
### FazArvdeBitMap

Faz a de-serialização de um arvore binária serializada em pré-ordem. Para isso, faz a indexação do bitmap de entrada, e faz a leitura do mesmo bit a bit, tratando-o como uma pilha. Ao encontrar uma folha (1) no bitmap de entrada, faz a leitura de 1 byte do bitmap, o colocando em uma folha devidamente localizada na arvore de saída.



### PercorreArvorePorBitEEscreveSaida

Dado um bitmap contendo um arquivo (sem cabeçalho e/ou arvore serializada) codificado usando uma arvore de Huffman e sua arvore de Huffman, essa função chama **RetornaCharRecursivamente** (que faz a leitura bit a bit do arquivo, até que se complete um caractere, retornando o mesmo) e escreve em disco o caractere recebido, até que o bitmap acabe.



## TAD ListaArv

Utilizando a lista genérica anteriormente explicada, foi construída a lista de árvores, que utilizamos para a montagem da árvore de Huffman.

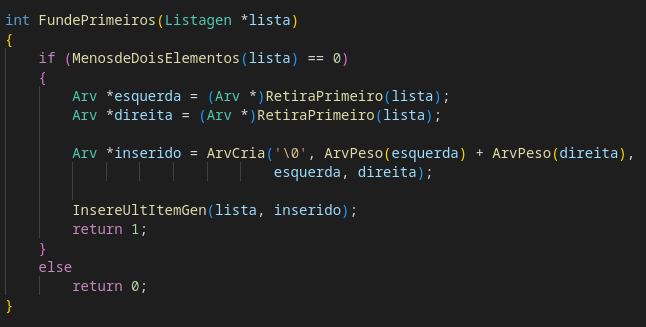
**Ele possui as seguintes funções:**



Entre elas, são válidas de trazer a esse relatório as seguintes funções:

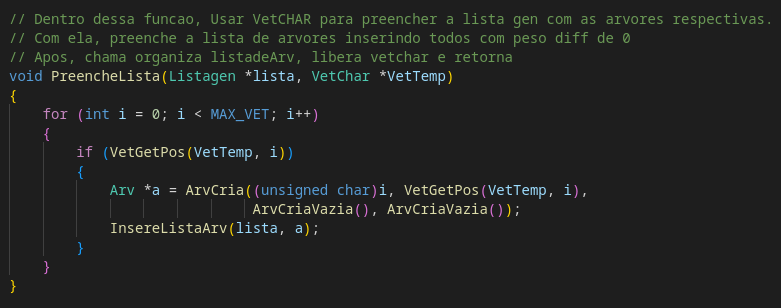
### FundePrimeiros

Nessa função, retiramos da lista os dois primeiros elementos e criamos um nó, cujo peso é igual a soma dos pesos e seus filhos são as árvores retiradas da lista. Após esse processo, inserimos esse nó criado no fim da lista.



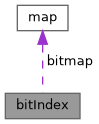
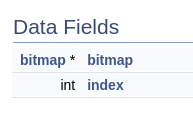
### PreencheLista

Essa função possui a funcionalidade de, dado um vetor de frequência, preencher uma lista de árvores com a informação presente. Para isso, varremos o vetor e caso obtenha uma frequência maior que 0, cria uma folha cujo caractere é a combinação de bits (byte//caractere) relacionada a frequência.



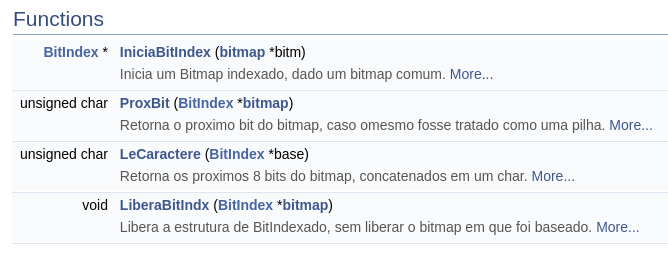
## TAD BitIndex

É um simples tipo para adicionarmos algumas funcionalidades ao bitmap, tendo um index, podemos saber onde está a leitura do arquivo e, por exemplo, o tratar como uma pilha de bits.



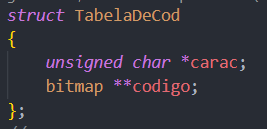
Nele Temos as seguintes funções:

Dada a simplicidade de todas, não se faz necessária maiores explicações sobre o funcionamento de nenhuma.



***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.

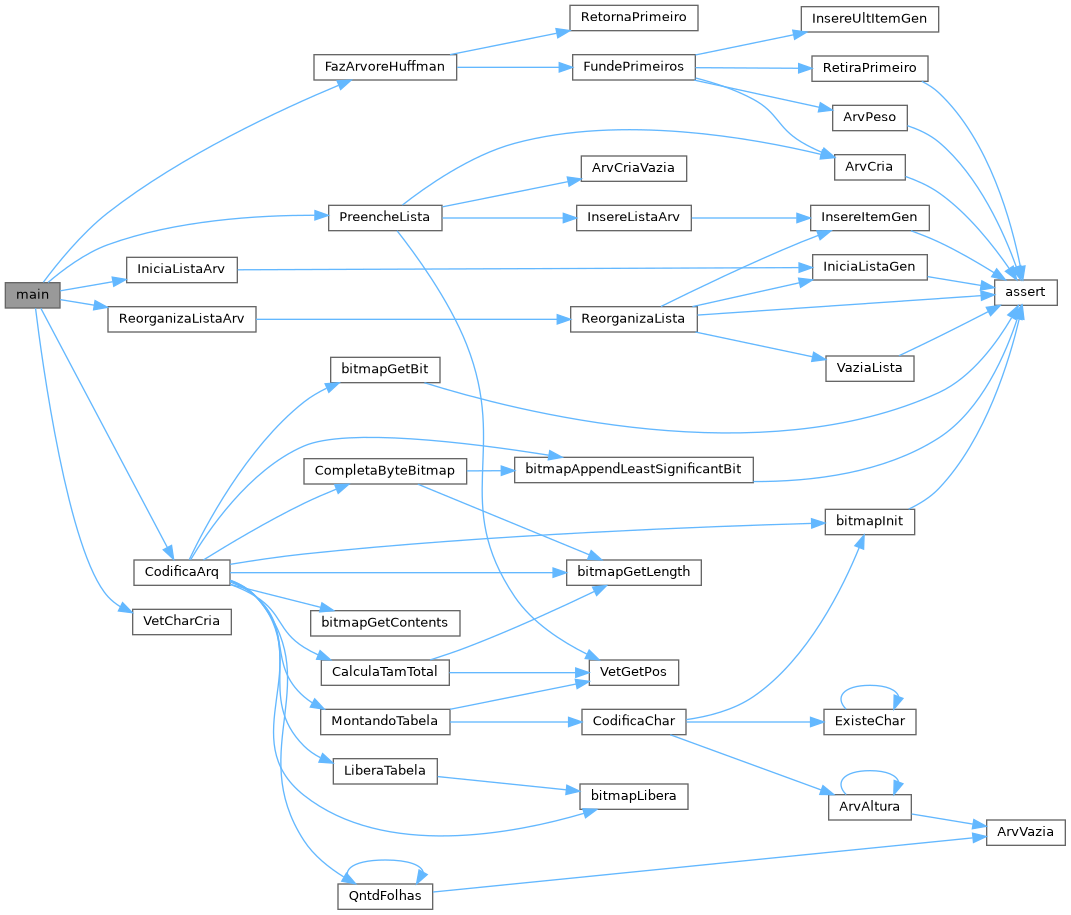


Como podemos ver para cada um dos caracteres a função CodificaChar pega e gera um código em bitmap (de acordo com a arvore de *Huffman*) para o caractere i caso seu peso seja diferente de 0, temos assim uma tabela que para cada caractere no vetor de *char* seu respectivo código corresponde *(mesmo index)*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

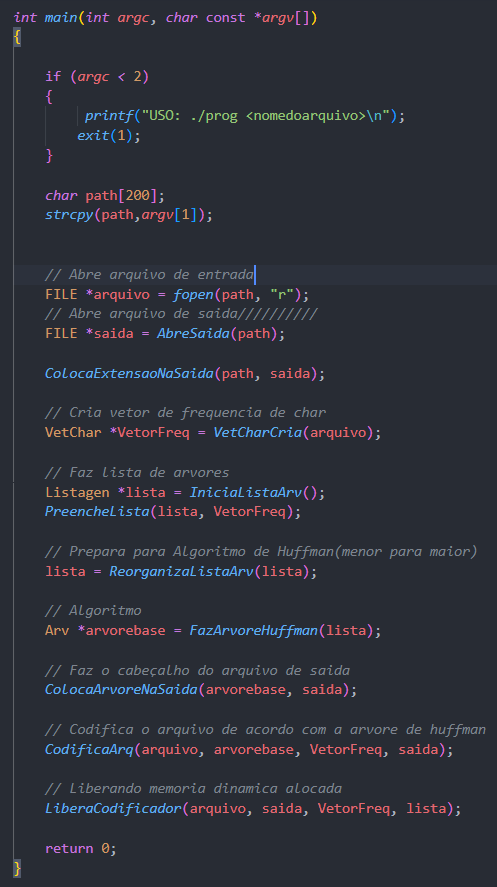
Voltando ao compactador em si temos essas funções privadas:



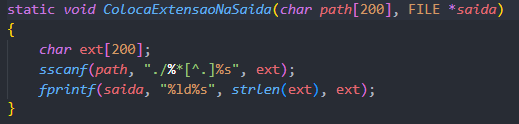
A *main* segue esse fluxograma:



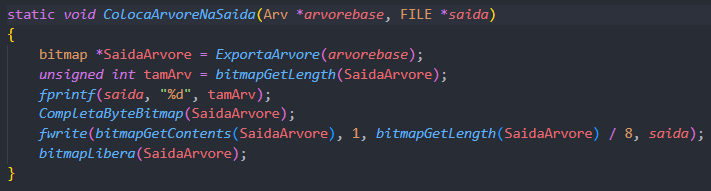
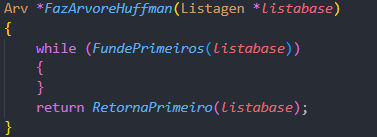
Olhando o código:



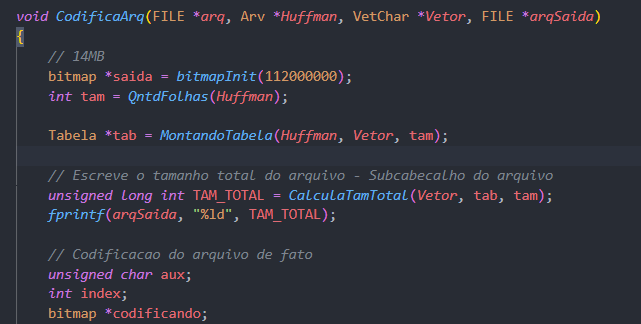
Vemos que a *main* começa verificando caso o programa foi chamado corretamente, depois passamos o caminho para o arquivo salvando no path, abrimos o arquivo de entrada e a saida, depois usamos uma função para colocar a extensão do arquivo e seu tamanho em bytes (4(tamanho) .txt por exemplo) na saída,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



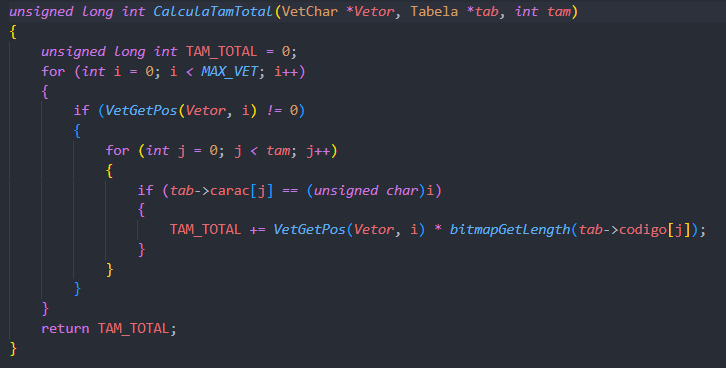
Após isso criamos o vetor de frequência de caracteres para formar a arvore de codificação, mas antes colocamos os caracteres presentes em arvores e inserimos na lista, que logo após é organizada para usar o algorítimo de *Huffman.* Com isso pronto montamos a arvore e colocamos ela na saída, colocamos 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:



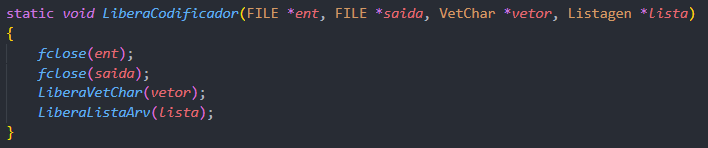
A função acima simplesmente calcula o tamanho do código gasto por cada caractere da tabela de codificação vezes seu peso para saber quantos bits no total serão usados.



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*.

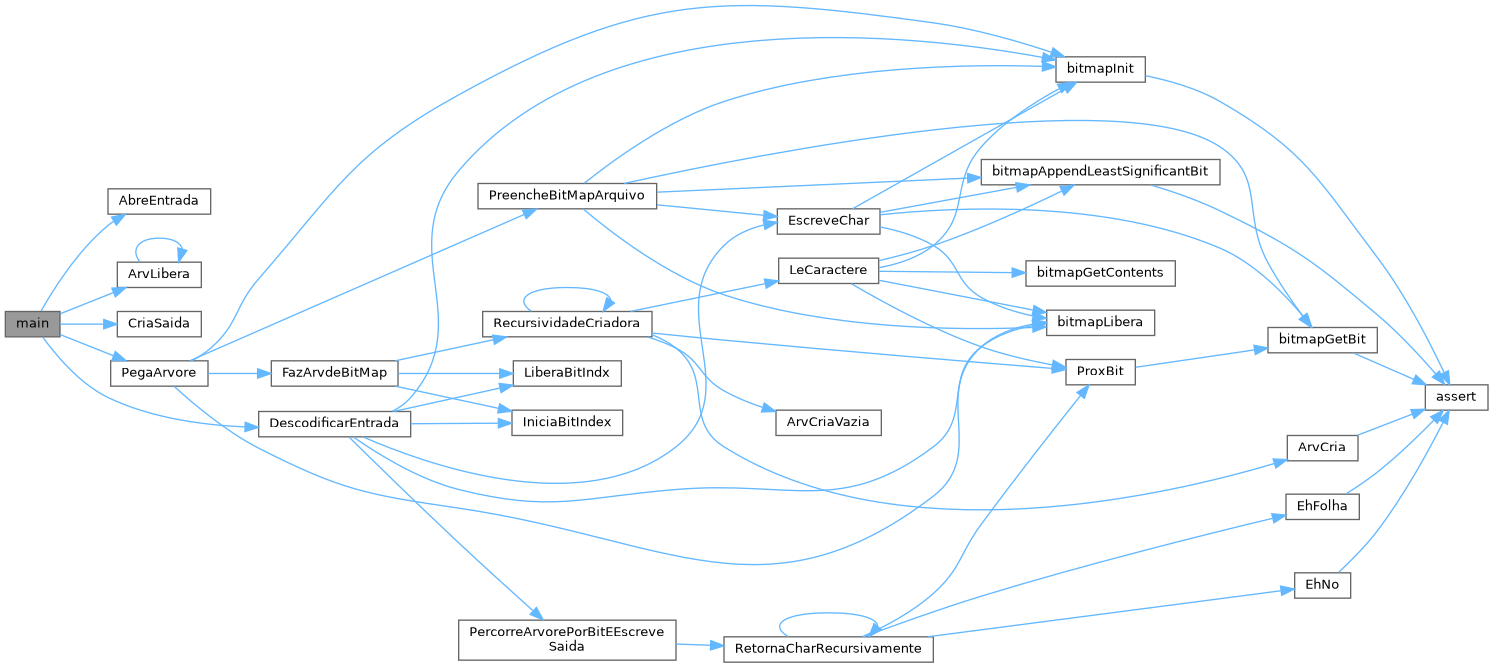
Voltando a *main* o próximo passo é liberar tudo:



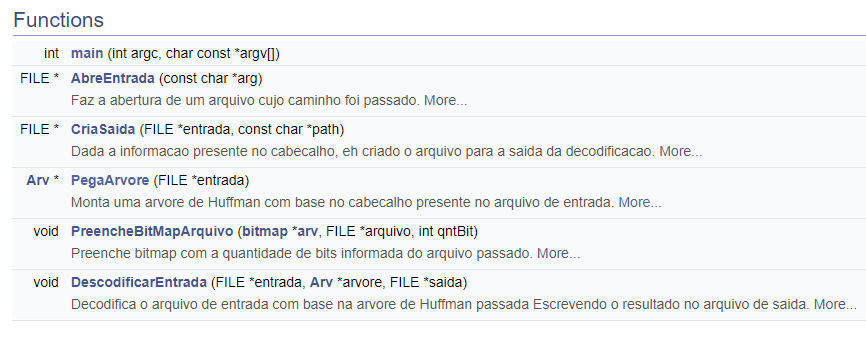
Obs.: Como a lista ainda tem a raiz da arvore de *Huffman* liberando a lista nós liberamos a Arvore e assim termina o Compactador.

TAD Decodificador:

O nome é decodificador, porém este TAD é responsável pela descompactação do arquivo. Começando pela *main* temos o seguinte fluxograma:



Como podemos ver o TAD BitIndex é utilizado aqui para que possamos fazer manipulação mais tranquila de bits.Segue as funções utilizadas e o que elas fazem em um curto resumo (a frente explicaremos melhor):



Vamos olhar o código da main:



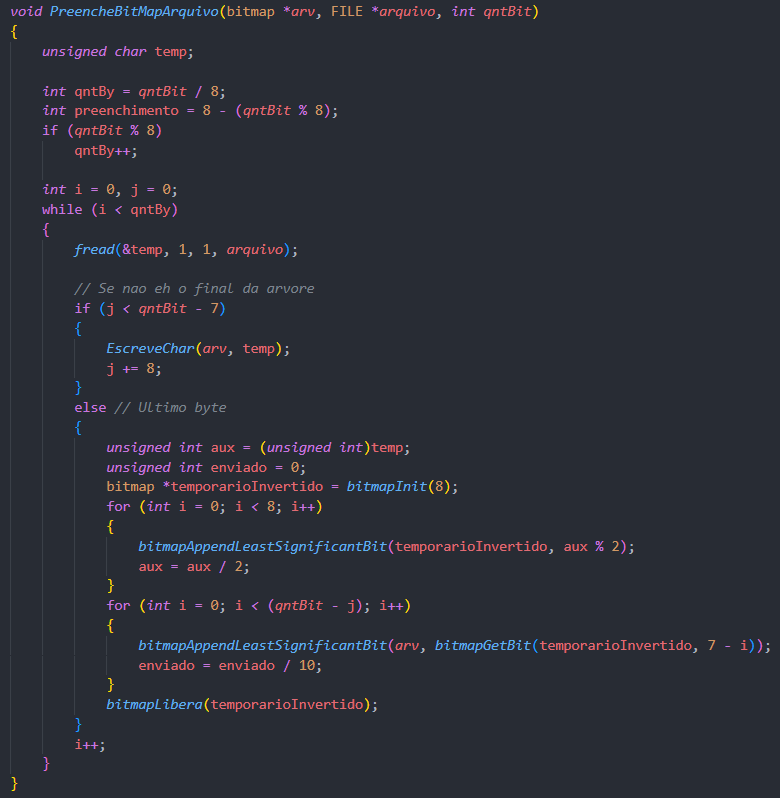
Podemos ver que o programa começa da mesma forma que o compactador, fazemos verificações e abrimos os arquivos de entrada e saída, para fazer isso lemos quantos bytes vai ser gasto para a extensão e logo depois lemos a extensão da entrada, com ela em mãos a saída é aberta.



Logo após já conseguimos pegar a arvore de codificação pelo cabeçalho com a função PegaArvore utilizando o a entrada. Segue a função abaixo:



Conseguimos ver claramente os passos da função aqui acima, primeiro lemos a quantidade de bits gastos pela arvore, iniciamos um bitmap e preenchemos o bitmap com a arvore usando a função EscreveChar já explicada anteriormente. Segue a função:



Depois usamos a função do TAD Arvore FazArvdeBitaMap para fazer a arvore em si usando seu *bitmap,* logo após o *bitmap* é liberado e a arvore montada é retornada. Voltando a *main* depois de ter a arvore em mãos o próximo passo é decodificar o arquivo, usando a função DecodificarEntrada abaixo:



Como podemos ver na função acima após ler o tamanho total de bits gastados pelo arquivo em si iniciamos um bitmap com esse tamanho e transformamos ele em BitIndex, depois lemos até o final do arquivo e gravamos no bitmap.

Logo após usamos a função PercorreArvorePorBitEEscreveSaida que roda a arvore de acordo com o bitmap (1 para direita e 0 para a esquerda), para encontrar os devidos caracteres de acordo com o que está na entrada. Depois escrever na saída o resultado final, que é no caso o arquivo descompactado.

Depois toda a memória é liberada.

Conclusão:

Um dos trabalhos que mexe com o mais baixo nível em C foi bem interessante, com certeza aprendemos muito fazendo ele. Uma das partes mais difícil foi fazer algoritmos que trabalhassem com o binário em si, mesmo já tendo a biblioteca do bitmap disponível ainda foi necessário criar a BitIndex para descompactar os arquivos. Uma experiência que com certeza iremos carregar durante nossas jornadas como estudantes de Ciência da Computação.

Grande parte do aprendizado que extraímos desse trabalho se deve a essa complexidade de 0’s e 1’s, que normalmente é abstraída.