# Relatório Trabalho 1 de Estrutura de Dados

*Professora: Patrícia Dockhorn Costa*

*Período: 2022/1*

*EDCare: Ed Cuidando de Idosos!*

*Alunos: Rhuan Garcia 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:**

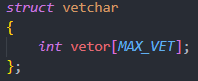
Para começar, fizemos uma implementação sem seguir nenhum padrão, reutilizamos as árvores que implementamos nas aulas para fazer a árvore de Huffman.

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).

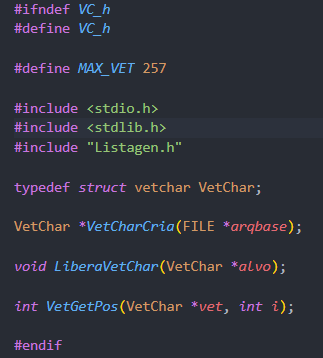
Começamos primeiro pelo compactador e seus TADs para depois seguir para o Descompactador.

***TAD VetChar:***

Nesse TAD foi implementado o vetor de frequência de caracteres. Seu *struct* comporta todos os 256 valores que um byte pode ter utilizando um vetor de inteiros.



Segue seu .h:



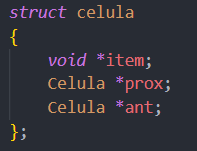
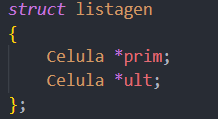
Segue abaixo seu .c com as suas funções devidamente implementadas:



Vemos aqui que ele lê todo o arquivo em *PreencheVetChar* e já vai contando quantas vezes cada caractere apareceu.

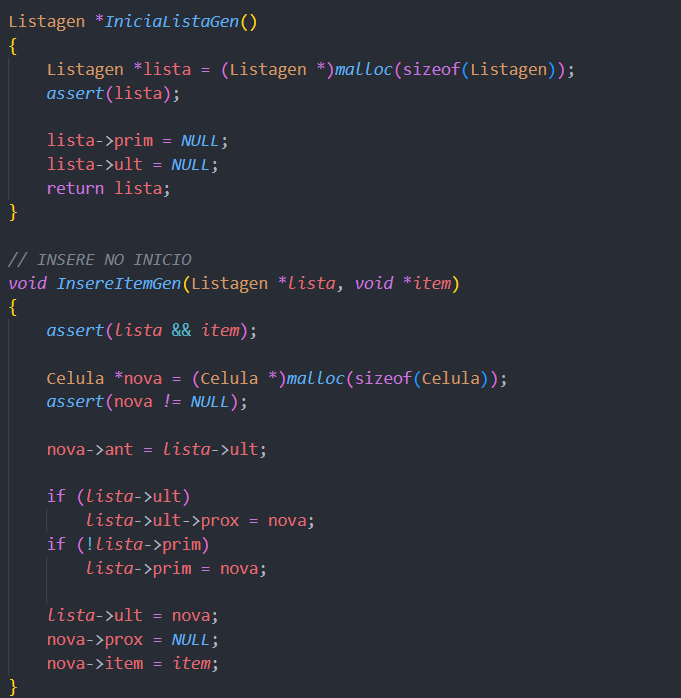
***TAD ListaGen e ListaArv:***

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:

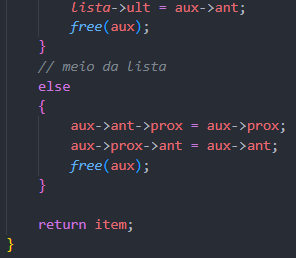
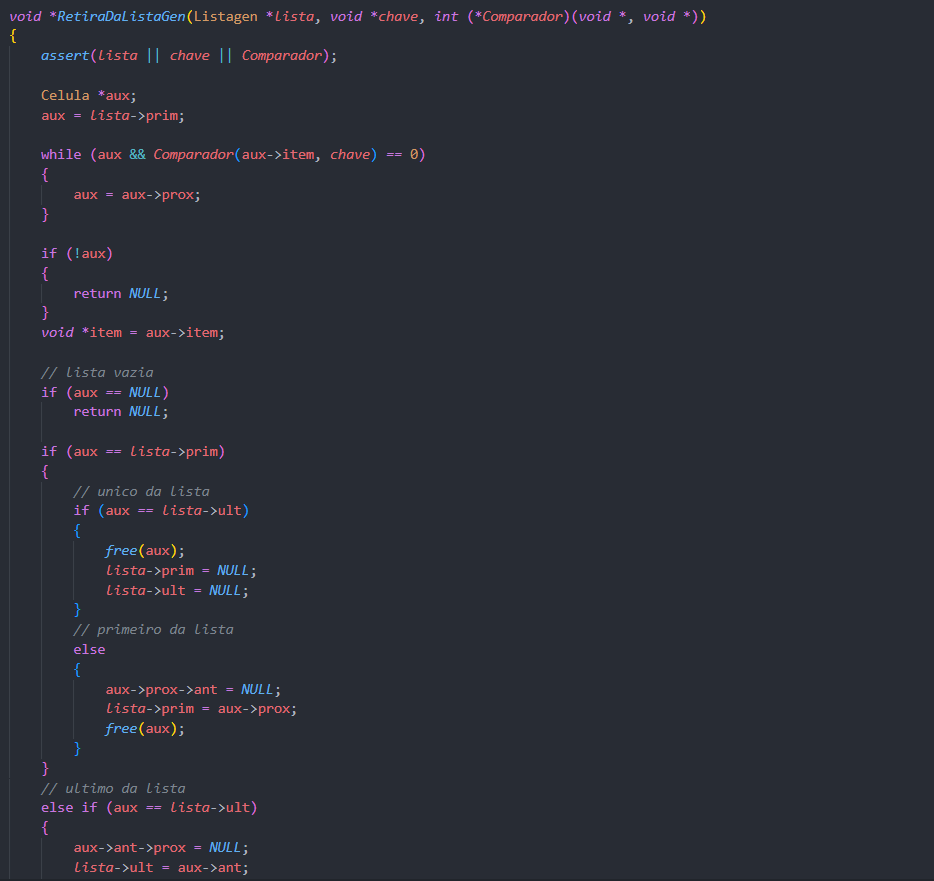


Irei simplesmente mostrar as funções da lista que são básicas, elas todas estão comentadas no .h, explicarei a fundo as mais especificas e complexas.

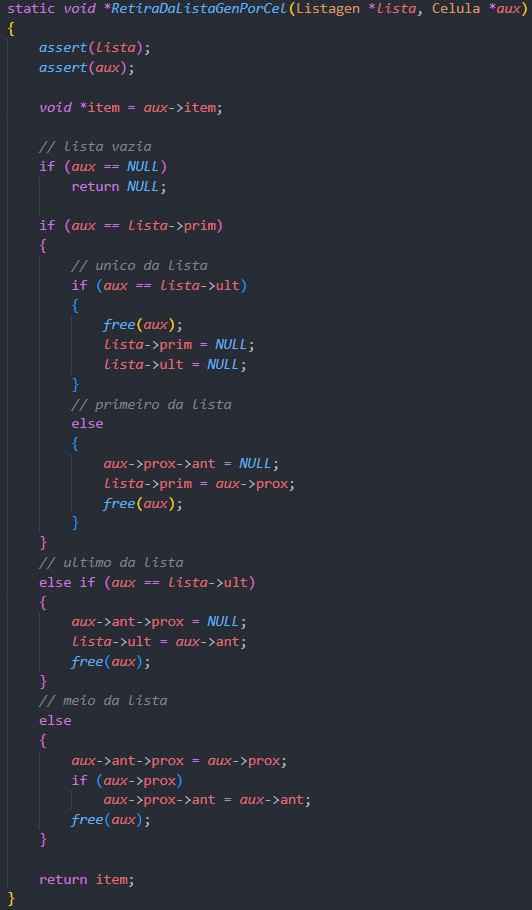
Inicia (cria) e Insere no Inicio:



Retira (Normal) comparador definido pelo tipo ListaArv:

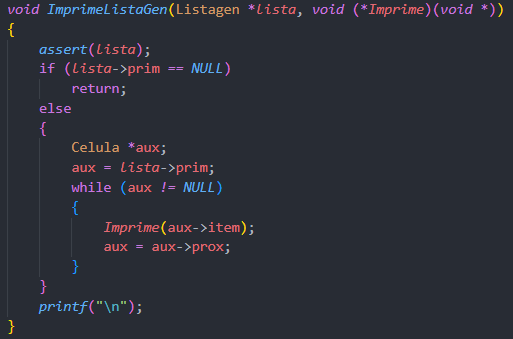


Retira por Célula ( função privada de uso especifico para reorganizar a lista):

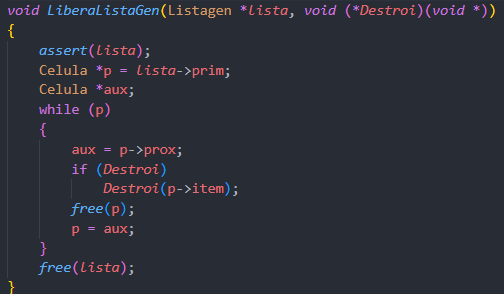


A função acima é usada para organizar a lista de forma que ela sempre recebera uma célula contida na lista assim evitando a busca que gasta tempo.

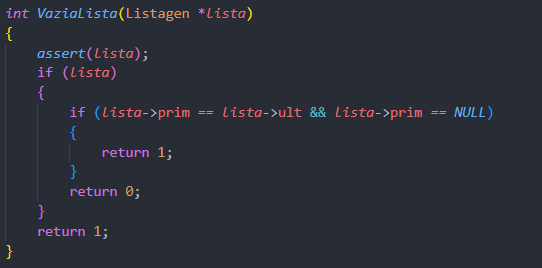
Imprime :



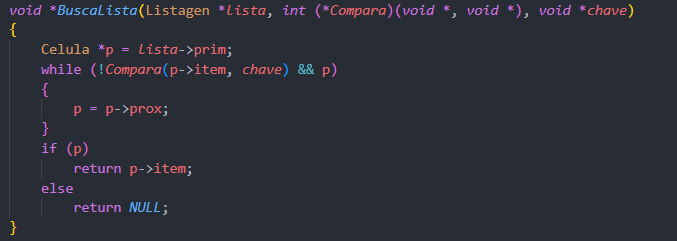
Libera:



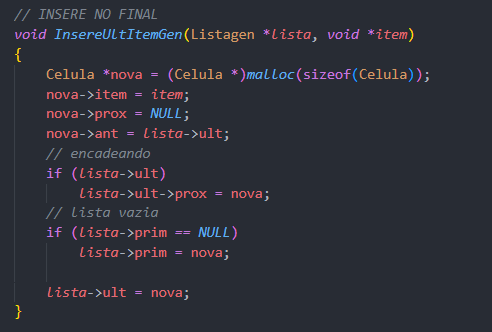
ListaVazia (1 se lista estiver vazia):



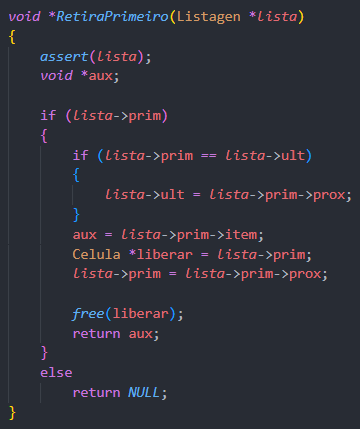
Busca Lista (retorna item):



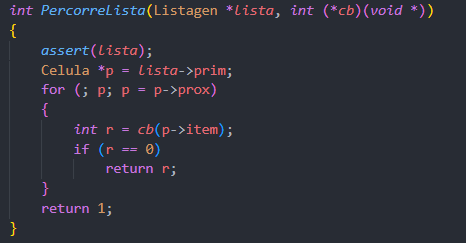
Outro insere porém nesse caso é no final:



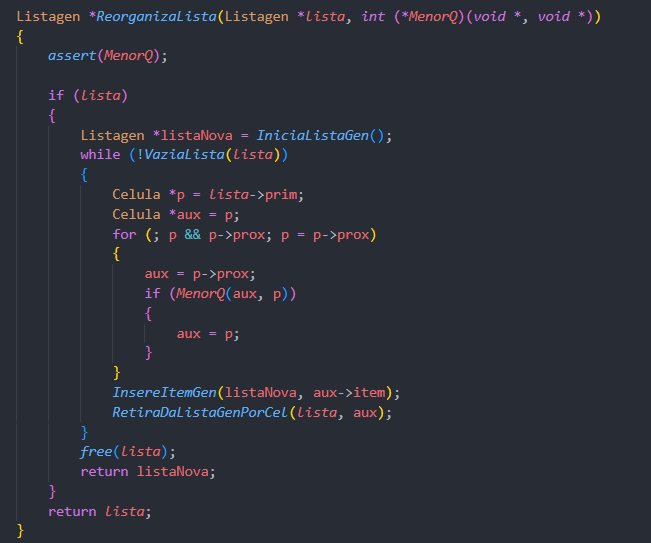
Outro Retira porém agora da primeira posição direto:



Uma percorre Lista:

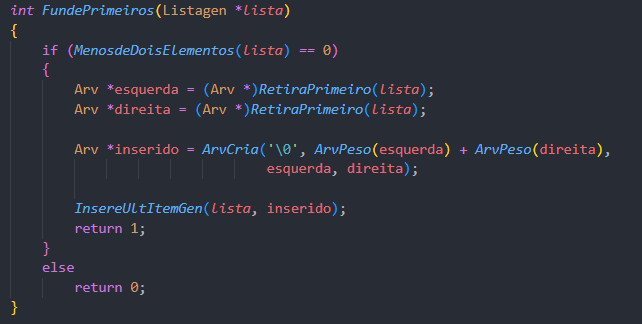


Agora chegamos na função Reorganiza Lista, aqui estamos implementando uma função para ajudar na hora de fazer a arvore de Huffman deixando os caracteres de menos peso mais ao final da lista e os de maior peso mais no inicio, usando uma função de callback, que no caso pode organizar a lista de qualquer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



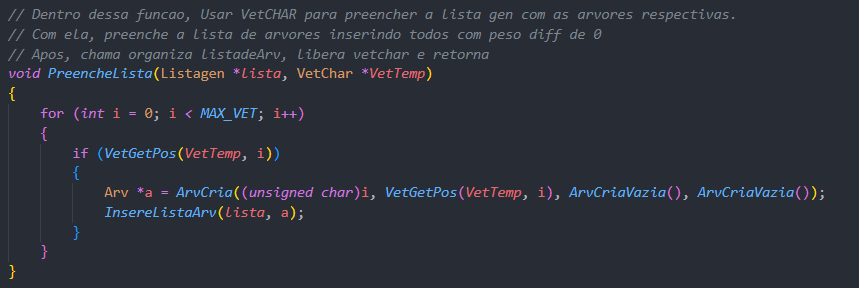
Vemos que uma nova lista é criada e para cada elemento da lista nós fazemos uma exaustiva comparação entre eles para inserir na nova lista da forma estabelecida pela função MenorQ passada como parâmetro. Depois a lista antiga é liberada e retornamos a nova.

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:

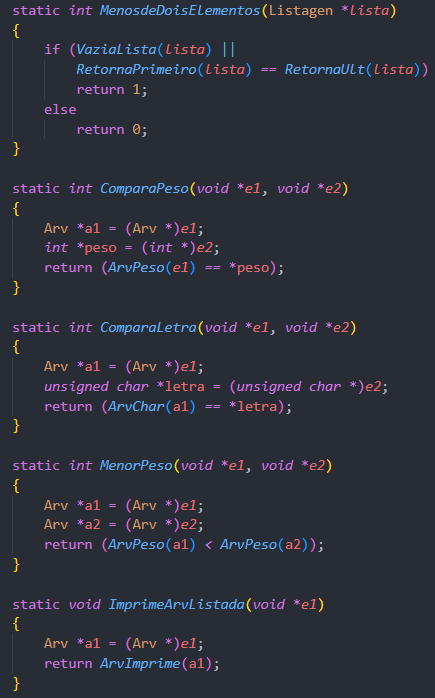


Se a lista tiver menos de 2 elementos a função acima para e retorna 2.

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:

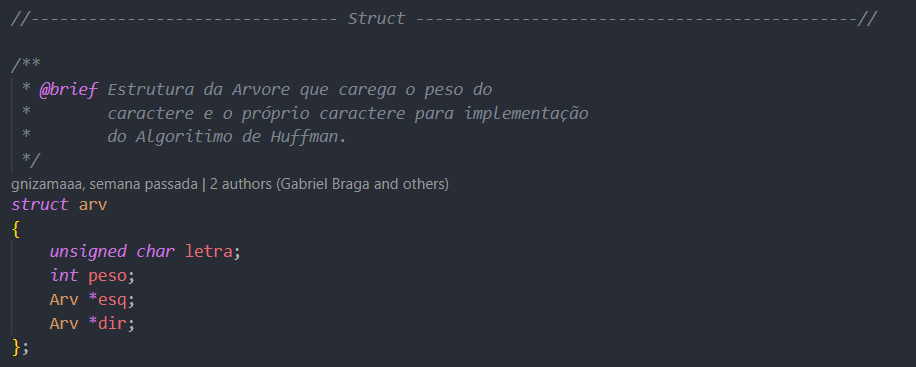


Abaixo seguem funções utilizadas como callback nas funções da lista genérica:



***TAD 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, além da recursividade, onde temos árvore esquerda e direita. Segue a estrutura arv:



No *Arvore.h* temos o *typedef* que define o tipo *Arv* como essa *struct* acima.

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:

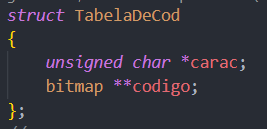


Pulando as funções normais de arvore, que não fogem do padrão, seguimos para as mais complexas. Começando pela

***TAD 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:



Essa Tabela têm suas funções de Montar e Liberar: