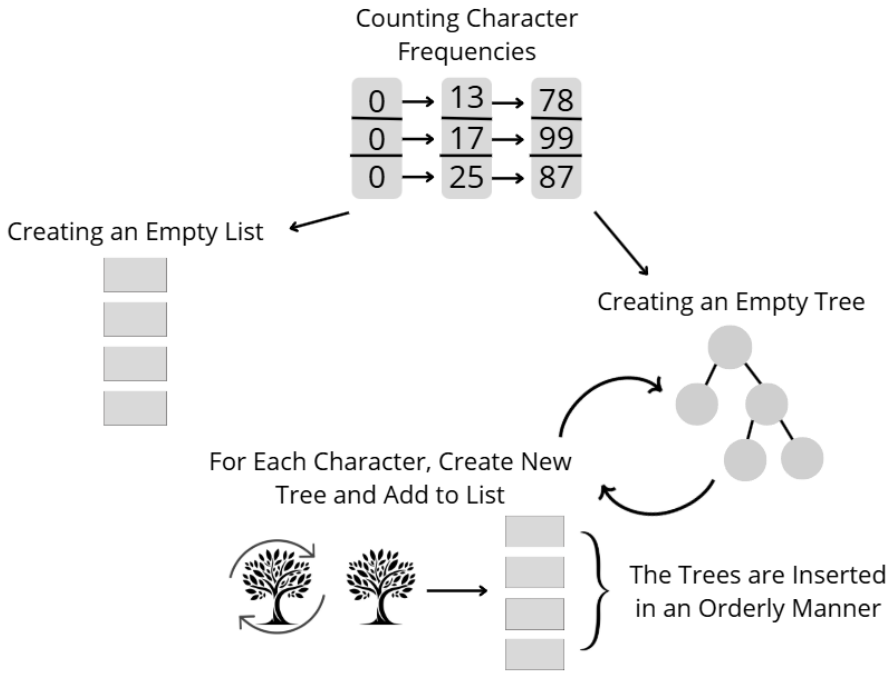
Compactador de Huffman

Introdução: A ideia geral do projeto é criar um código capaz de compactar e descompactar arquivos – .png, .txt, etc – utilizando a codificação de Huffman. O programa “compactador.c” recebe como parâmetro, pelo terminal, o nome do arquivo a ser compactado e realiza a compactação do mesmo, criando um arquivo com o mesmo nome do título passado, adicionando, ao final, um “.comp”. O “descompactador.c”, ao ser executado, obtém, também, pelo terminal, o nome do arquivo com final “.comp”, descompactando-o e recriando o arquivo original com o mesmo apelido passado como parâmetro no “compactador.c”.

Implementação:

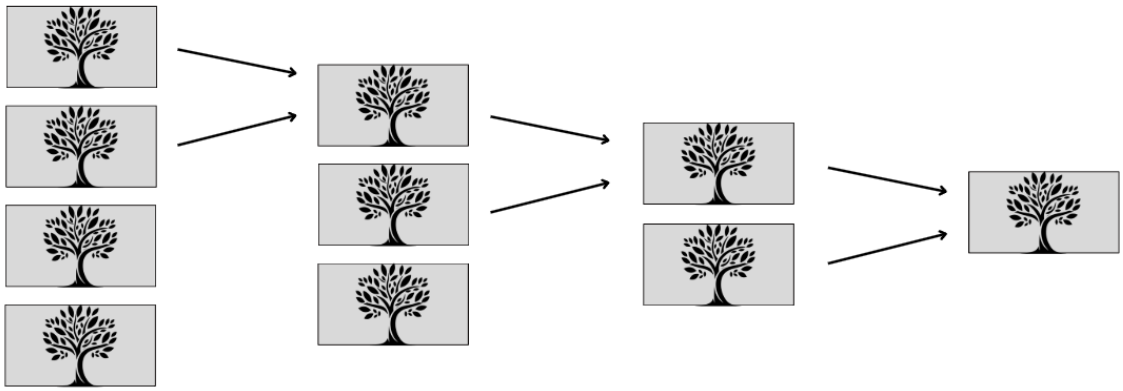
Compactador.c: O programa abre o arquivo para leitura binária e, para cada *byte* lido, conta sua frequência no vetor de frequências. Válido ressaltar que cada índice do vetor é identificado pelo respectivo código ASCII da informação binária presente na leitura da entrada.

Cada *byte* é armazenado em uma estrutura de dados do tipo “árvore” e adicionada ordenadamente, de acordo com sua frequência, em uma lista encadeada como mostra o diagrama a seguir:



Funções envolvidas: *“criaArv_vazia”, “insere_Arv”, “insereNaLista”.*

Em seguida, a cada *loop*, duas árvores de menor frequência são juntas, somando suas frequências e formando uma outra árvore, que agora contém duas sub-árvores, sendo adicionada na lista encadeada de forma ordenada. Esse processo se repetirá até que reste apenas uma árvore, conhecida como árvore de Huffman. Importante destacar que, a cada *loop*, essas duas árvores são retiradas da lista para serem adicionadas em outro "nó".



Funções envolvidas: *" juncao_arvs_lista", "junta_arvs", "retira_duasArv_Da_Lista".*

A codificação de Huffman é utilizada para atribuir um código binário a cada "folha" dessa árvore, o tamanho de *bits* deste código, bem como a sequência codificadora, dependem, respectivamente, da sua altura em relação ao "nó-raiz", e da direção que a função de busca seguiu a partir do início – "0" para a esquerda, e "1" para a direita. Com isso em mente, para comentar a respeito das principais funções utilizadas nessa etapa do programa, é necessário entender o arquivo "codificacao.c".

Contendo um *struct* "Cod", que guarda a informação de uma variável do tipo *unsigned char* e um vetor do tipo *char,* esse arquivo é responsável pela parte de codificação do programa.

Funções envolvidas:

" vetor_codificacao" - Cria um vetor de ponteiros para "Cod" com o tamanho especificado.

" inicializa_vetor_COD" - Inicializa o vetor de ponteiros com valores padrão.

“gera_codigo” - Percorre a árvore de Huffman para construir o código binário (string de '0's e '1's) para cada caractere (folha da árvore). Guarda no vetor de ponteiros “Cod” essa informação.

Por fim, o programa utiliza as funções da biblioteca “bitmap.c” para guardar, de forma compactada, o conteúdo do arquivo original no arquivo “.comp”. Interessante ressaltar que, a árvore de Huffman, também precisa ser guardada no arquivo compactado. Então, é necessária a serialização desta árvore (“nós não-folha” são considerados como “0”, e “nós-folha” são considerados como “1”), importante serializar o *byte* que esse “nó-folha” contém também.

Descompactador.c: O programa abre o arquivo compactado – “.comp” – passado como parâmetro no terminal para leitura binária. Em seguida, descompacta as informações presentes: número de *bits* da árvore e seu respectivo *bitmap*; Número de *bits* da informação compactada e o próprio conteúdo presente em seu *bitmap.*

Após isso, o programa segue seu funcionamento:

1 - Interpreta a serialização da árvore de Huffman compactada e reconstrói suas informações. Funções envolvidas: *“preenche_bits_array”, “huffmanTree_descompacta”*

2 - Reconstrói a tabela de códigos a partir da árvore descompactada. Funções envolvidas: *“tabelaCod_reconstruct”.*

3 - Percorre a sequência de bits presente no *bitmap* que contém a informação do arquivo original e decodifica a mesma. Recriando o arquivo originário. Funções envolvidas: *“file_reconstruct”, “file_reconstruct_aux”.*

Casos Especiais:

1 - Arquivo Vazio: No início da execução, o compactador, por meio do “*fseek”*, move o ponteiro do “*FILE *input_file”* para o final do arquivo. Posteriormente, a função “*ftell”* é chamada para determinar a posição do ponteiro “*FILE*”* , caso essa posição seja igual à zero, o arquivo está vazio. Se, de fato, estiver vazio, o programa cria uma compactação do arquivo original contendo o valor zero que, posteriormente, será lido pelo descompactador para identificar que o *“.comp”* não possui informação. O descompactador, por sua vez, identifica que o arquivo original estava vazio, recriando o mesmo e finalizando a execução.

2 - Arquivo com um tipo de informação única:

Ex: caractere único – “X”.

Nessa situação, o compactador verifica, inicialmente, se a lista de árvores contém apenas um elemento. Caso, de fato, possua apenas um elemento, o programa identifica isso e aciona uma *flag*, que será utilizada nas funções posteriores para o funcionamento normal do código. Quando ocorre esse cenário, o programa, ao gerar o respectivo código de Huffman desse caractere único, sempre o identifica como ‘1’, bem como a serialização da árvore, que é identificada como ‘1’ também, para não prejudicar a execução do código.

O descompactador, em sua execução, reconstrói a árvore de Huffman e identifica se ela possui apenas a raiz, ou seja, se contém apenas um nó. Caso identifique esse tipo de ocasião, o programa gera uma *flag* a ser utilizada nas funções a seguir, que resultam no pleno comportamento do código. Aqui, na hora de reconstruir a tabela de códigos, o programa também tem o número ‘1’ como identificador do caractere único.

Informações do arquivo compactado:

Contém a quantidade de *bits* serializados da árvore de Huffman e a própria árvore. Ademais, possui, também, a quantidade de *bits* utilizados para compactar a informação do arquivo original, além do próprio conteúdo.

Comentários Individuais:

Davi: Minha principal dificuldade durante o projeto foi entender e aprender como utilizar o *bitmap*. Demorei diversas horas para compreender como a criação e implementação do mapa de *bits* guardaria a árvore de Huffman no arquivo compactado junto ao código binário de cada caractere. Tendo isso em vista, achei o projeto de fato interessante e útil, mesmo sendo trabalhoso e bem complexo.

Bruno: O trabalho parecia muito mais complexo no início, então essa parte é bem demorada. Mas, depois que se começa, o funcionamento do código vai fazendo sentido e tudo se encaixa. O maior desafio foi chegar em um consenso sobre a abordagem do problema (eu queria fazer uma pilha ordenada, Davi preferiu tentar uma lista que insere ordenadamente). No demais, foi gratificante terminar e ver o código funcionar.

Materiais de Apoio:

https://youtu.be/sXY_V_HPfyA?si=pwutqEyi3k4v_Snc

https://youtu.be/JgEF_kkhLjQ?si=x7Z9sN6_Wk5O5ihO