UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO

CARLOS DANIEL ALBERTINO VIEIRA

WinRAR

Vitória

2021

CARLOS DANIEL ALBERTINO VIEIRA

WinRAR

Trabalho prático apresentado a disciplina de Estrutura de Dados 1 como requisito para obtenção de nota.

Professora: Patrícia Dockhorn Costa

Matricula: 2020100867

Vitória

2021

Sumário

1. INTRODUÇÃO

O armazenamento de arquivos sempre consistiu numa grande preocupação desde o início da computação nos anos 40. Com efeito, atualmente inúmeras pessoas, empresas e instituições contém um altíssimo volume de informações armazenadas na forma de dados (em HDs, SSDs, nuvens) e a demanda por formas de aumentar a quantidade de informação salva é crescente, em vista das possibilidades lançadas nas áreas de segurança, manipulação e análise de dados por tal economia.

Sob a lente da busca por tecnologias capazes de aumentar o poder de armazenamento de arquivos, o trabalho presente apresenta um programa construído na linguagem C que oferece operações de compactação e descompactação de arquivos textos, de modo a reduzir o espaço consumido em memória por esses, semelhante ao programa winRAR.

1.1 OBJETIVOS

O programa Compactador propõe compactar arquivos textos na forma binária, de modo a possibilitar um possível ganho em memória decorrente da redução do tamanho total de armazenamento do arquivo. Apresenta, ainda, um programa de descompactação denominado Descompactador, que visa recuperar os arquivos originais compactados em binário na etapa de compactação.

1.2 VISÃO GERAL DO FUNCIONAMENTO

O programa Compactador inicia-se com a leitura do arquivo texto (formado apenas por caracteres ASCII padrão), cujo conteúdo será submetido ao algoritmo de Huffman para a montagem de um mapa/alfabeto de caracteres mais frequentes no arquivo entrado. Feito o mapeamento, gera-se um binário, que contém a árvore de codificação de Huffmann seguido do conteúdo do arquivo texto (ambos em bits), cujos caracteres foram gravados segundo o código (de tamanho variável) previsto na árvore de codificação. O binário gerado pode, então, ser submetido a descompactação por meio do programa Descompactador, que reconstroi a árvore de Huffman posicionada no começo do arquivo objeto, para então reconstruir a mensagem original ao ler os códigos dispostos no restante do binário e analisar a qual caracter ASCII esses mapeam.

2. IMPLEMENTAÇÃO

2.1 ALGORITMO DE HUFFMAN

Coração da compactação feita pelo programa compactador, o algoritmo de Huffman consiste na técnica por trás da codificação dos caracteres ASCII padrão gravados no arquivo texto. Ao performar uma rotina de criação sequencial de árvore binária, o algoritmo de Huffman retorna ao final de sua operação uma árvore binária, cujos nós folhas contém os caracteres que compõem a mensagem do arquivo texto e suas respectivas frequências.

A identificação dos caracteres nessa nova estrutura se dá por meio de uma orientação binária convencionada: 0 corresponde a árvore filha esquerda e 1 corresponde a árvore filha direita. Com essa orientação em mente e partindo da árvore mãe/geratriz de toda a estrutura, é possível visitar qualquer nó folha dessa estrutura utilizando um código binário sequencial, que, ao ser lido digito a digito e da esquerda para a direita, localizará com sucesso um nó onde reside em seu conteúdo a informação de um caracter ASCII presente no texto, bem como também sua frequência. Para os demais nós não-folha, esses armazenam a frequência total de todas as demais árvores que derivam deles, ou seja, que estão hierarquicamente abaixo e que dependem deles para que sejam acessados.

Ao final do algoritmo, espera-se que a codificação dos caracteres mais frequentes utilize menos bits para serem gravados no arquivo binário do que aqueles menos frequentes. Seja esse o caso, conclui-se logicamente que os caracteres mais frequentes devem estar o mais próximo possível da árvore mãe/geratriz da estrutura, para que não seja necessário um código binário sequencial longo para visita-los.

Antes de efetuado, no entanto, o algoritmo de Huffman solicita alguns preparativos para que possa operar. Nesse sentido, é necessário momentaneamente armazenar os caracteres lidos no arquivo-texto em um buffer, contabilizar suas ocorrências e ordenar os caracteres em ordem decrescente de frequência. Para efetuar isso, pensou-se em utilizar uma lista para armazenar os caracteres da mensagem, o que proveria uma forma de alocar gradativamente memória quanto mais caracteres ASCII aparecessem no arquivo texto, de forma a otimizar o uso de memória de acordo com a demanda do programa.

2.1.1 Lista escolhida

Escolheu-se aplicar o conceito de lista circular duplamente encadeada para preparar as células contendo as árvores (recém-nascidas e não interligadas) a serem utilizadas pelo algoritmo de Huffman no mapeamento dos caracteres mais frequentes do arquivo texto. A razão por trás dessa escolha foi a flexibilidade que esse tipo de lista traria ao facilitar a escolha do elemento inicial da árvore em formação, o que poderia se tornar uma vantagem na rotina do algoritmo de Huffman. Posteriormente, percebeu-se que a hipótese inicial de um possível ganho de praticidade no código não foi como se imaginou, no entanto obteve-se um ganho colateral na dinâmica do algoritmo: a identificação da condição de encerramento do algoritmo (quando resta apenas uma única árvore (sendo essa a árvore de codificação) na lista) tornaria-se mais simples, já que essa lista normalmente é construída com uma variável que armazena o número atual de elementos contidos.

2.2 ESTRUTURAS DE DADOS

Optou-se pela estratégia de criar arquivos gerais/coringa para a estrutura de listas circulares duplamente encadeadas sem sentinela e para a estrutura de árvore binária, as quais poderiam ser aproveitadas quantas vezes fosse necessário para criar novas estruturas desses tipos. Nesse sentido, foram criadas duas structs: tree e listaCircular. Os arquivos tree.h e listaCircular.h cumpriram com o papel de fornecer a declaração dos métodos necessários para a construção das funções básicas e comuns que envolvem as structs, respectivamente, enquanto seus arquivos fontes parelhos (tree.c e listaCircular.c) apresentam a definição de fato desses.

Criou-se, ainda, um arquivo específico para a estrutura caracter, que serve para armazenar o código ASCII e a frequência do caracter lido no arquivo texto. Nesse caso, o arquivo contendo os cabeçalhos dos métodos desse tipo se encontram no arquivo conteudo-mapa.h, enquanto suas definições no arquivo conteudo-mapa.c

Assim, ao todo foram criadas três structs: tree, listaCircular e caracter. Além dessas, também utilizou-se a estrutura bitmap, a qual foi cedida pela doscência para armazenamento e manipulação de bits.

2.3 PANORAMA GERAL DOS ARQUIVOS FEITOS

Foram criados 11 arquivos headers e 11 arquivos fontes, totalizando 22 arquivos. É possível sintetizar as motivações dos 11 arquivos headers (cada um apresentando um arquivo fonte parelho, com exceção o arquivo convencoes.h) como segue:

1. tree.h: estrutura e funções relacionadas ao tipo genérico de árvore binária
2. mapa.h: funções específicas relacionadas a árvore do tipo mapa
3. listaTree.h: funções específicas relacionadas a uma lista circular de árvores binárias
4. listaCircular.h: estrutura e funções relacionadas ao tipo genérico de lista circular
5. descompactador.h: funções relacionadas às tarefas de descompactação do arquivo binário gerado
6. convencoes.h: macros relacionados a convenção de orientação de deslocamento binário
7. conteudo-mapa.h: estrutura e funções ligadas ao conteúdo do mapa (código ASCII, frequência e bitmap dos caracteres do arquivo texto entrado)
8. compactador.h: funções relacionadas às tarefas de compactação do arquivo texto entrado
9. cliente.h: arquivo que disponibiliza as funções de compactação e descompactação disponíveis ao cliente
10. bitmapPLUS.h: funções complementares ao TAD bitmap
11. analisar-compactado.h: arquivo de depuração, onde estão funções capazes de levantar algumas informações inteligíveis sobre o arquivo compactado gerado.

Obs: também utilizou-se o arquivo bitmap.h e bitmap.c disponibilizado pela doscência para manipulação do TAD que envolve o mapa de bits.

2.4 PRINCIPAIS FUNÇÕES

As principais funções do trabalho se encontram no arquivo cliente.c, compactador.c e descompactador.c, onde estão implementadas as funções que são utilizadas para resolver os problemas principais propostos pelo trabalho. A ideia por trás da implementação de muitas dessas funções é a de dividir a tarefa principal em tarefas menores e resolver tais pormenores por meio da chamada de funções auxiliares construídas em arquivos fonte separados. Visto isso, o nome das principais funções e de suas dinâmicas segue abaixo:

2.4.1 cliente.c

1) compactar:

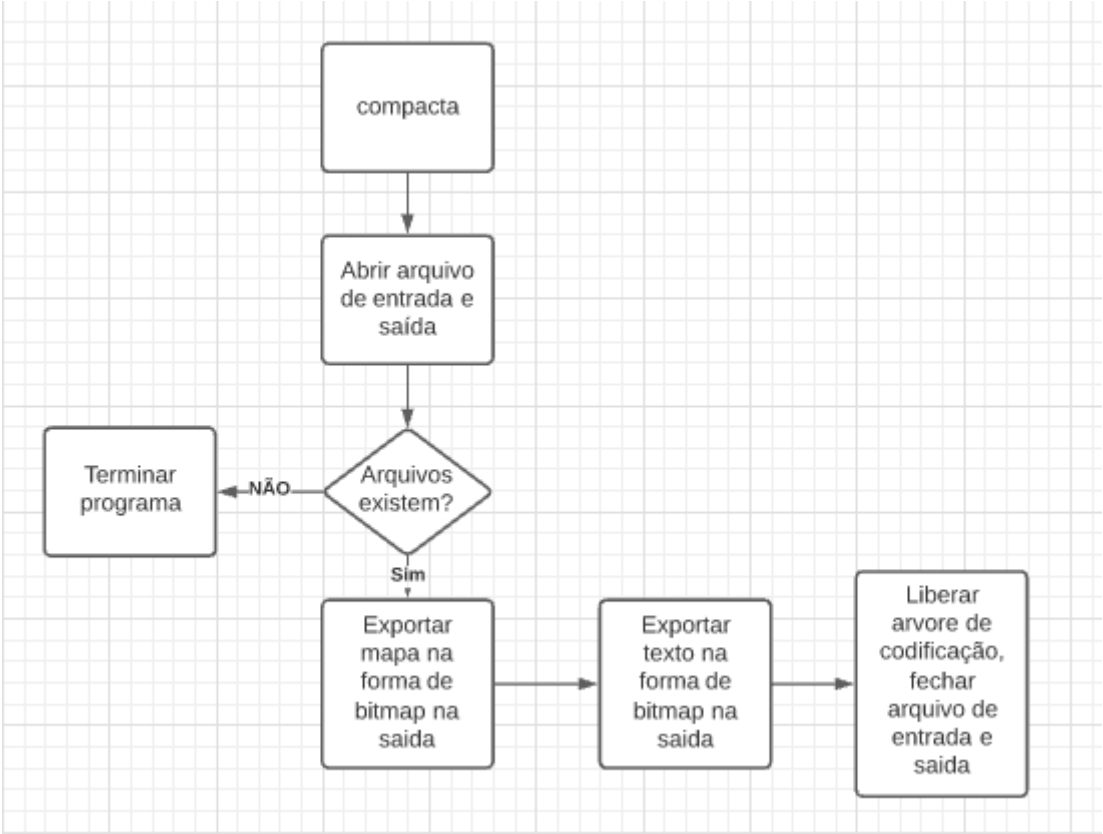

Figura 1: Funcionamento da função compactar

2) descompactar:

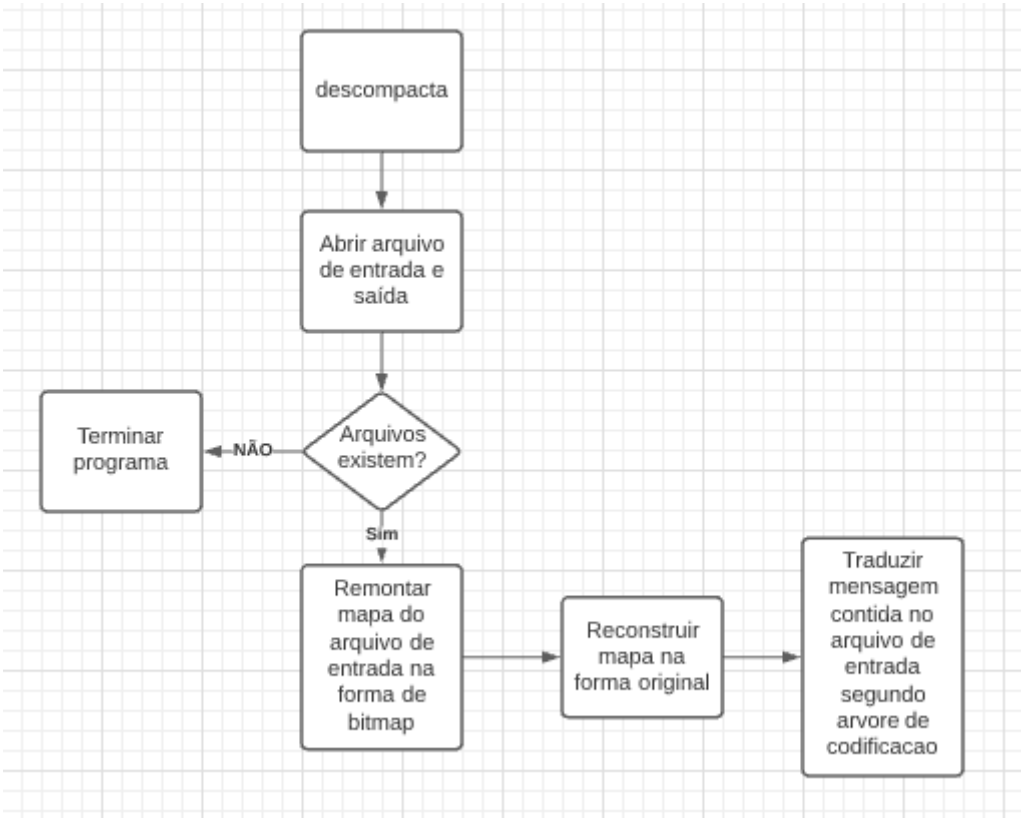

Figura 2: Funcionamento da função descompactar

2.4.2 compactador.c

3) montar_mapa:

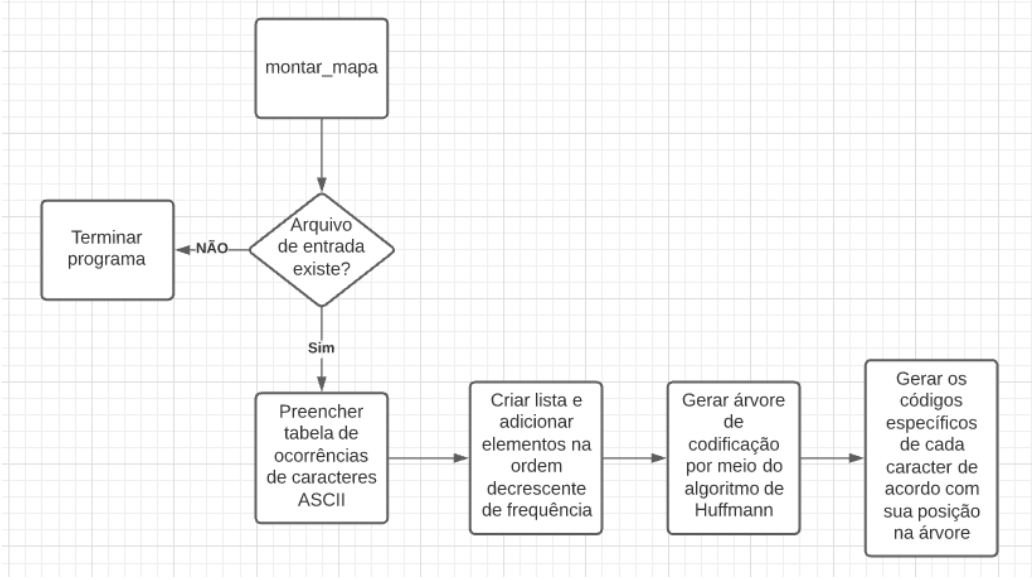

Figura 3: Funcionamento da função montar mapa

4) gravar_codigos_mapa:

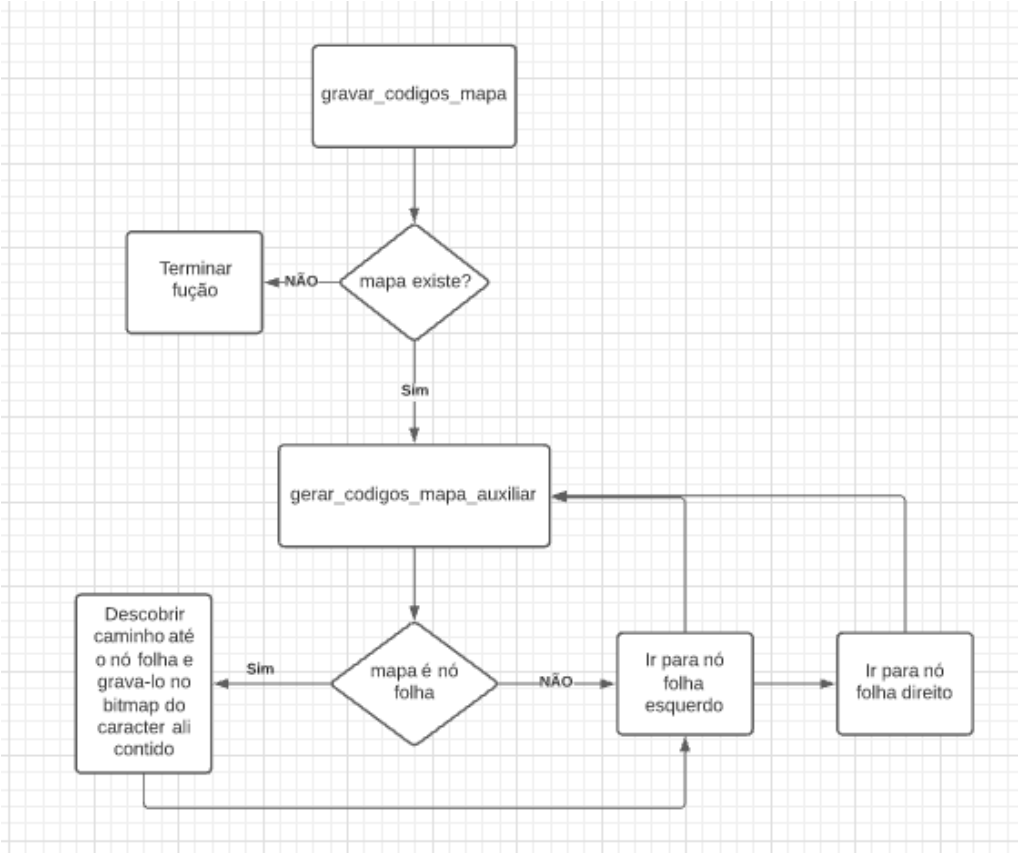

Figura 4: Funcionamento da função gravar_codigos_mapa

5) exportar_mapa_formato_bitmap:

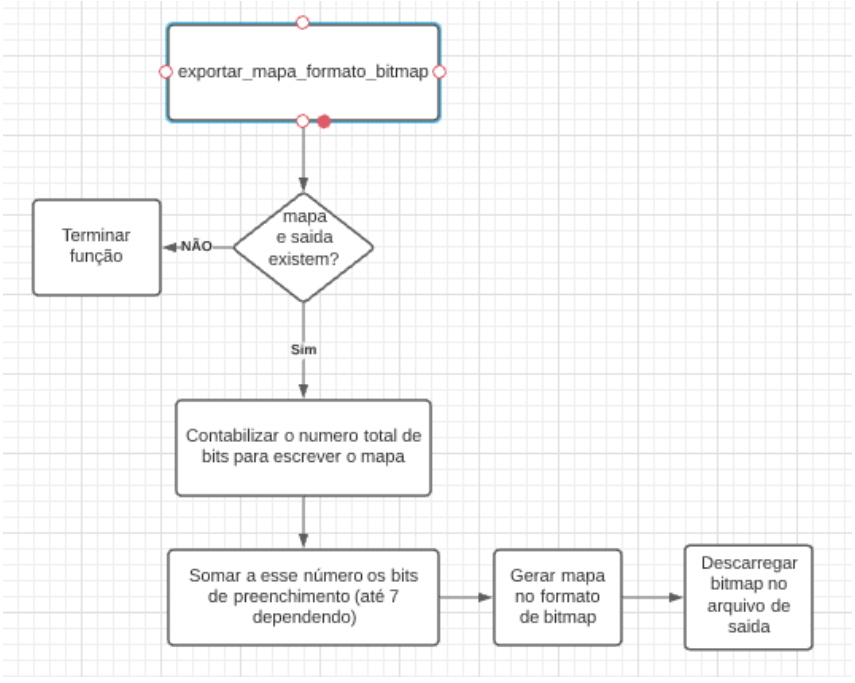

Figura 5: Funcionamento da função exportar_mapa_formato_bitmap

6) gerar_mapa_formato_bitmap:

Figura 6: Funcionamento da função gerar_mapa_formato_bitmap

7) exportar_texto_formato_bitmap:

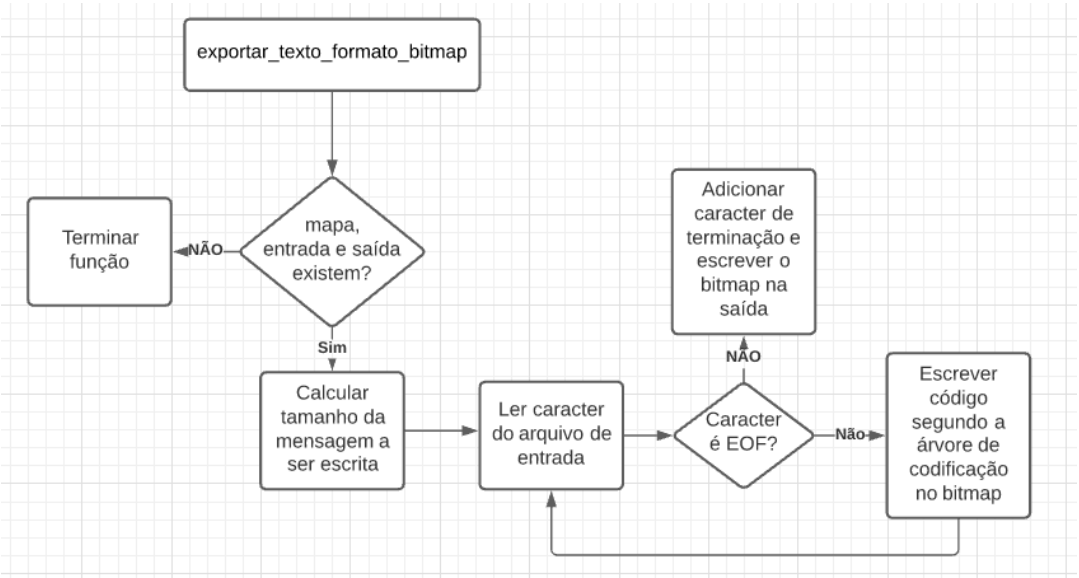

Figura 7: Funcionamento da função exportar_texto_formato_bitmap

2.4.3 descompactador.c

8) remontar_mapa_forma_bitmap:

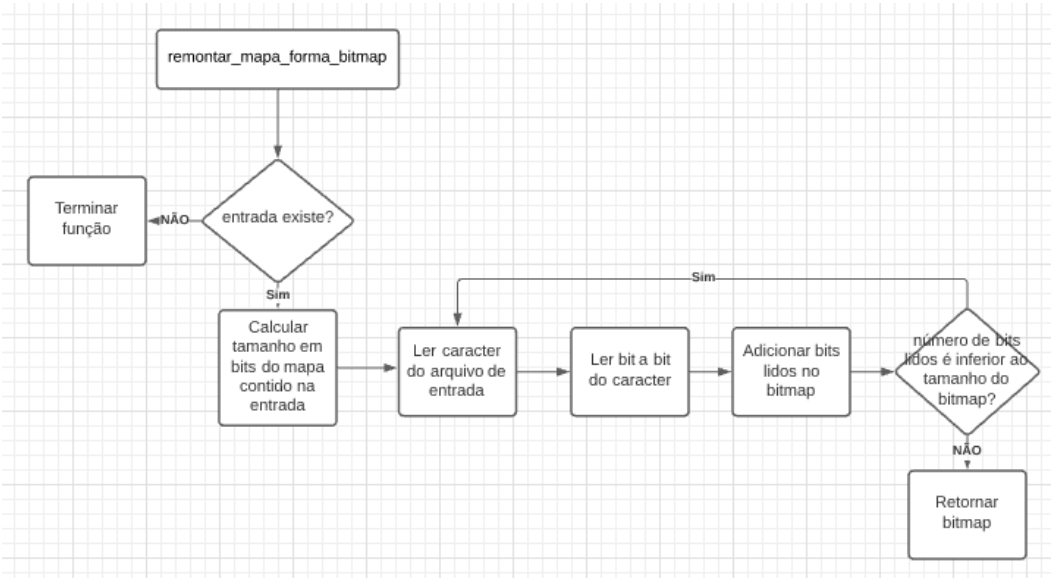

Figura 8: Funcionamento da função remontar_mapa_forma_bitmap

9) reconstruir_mapa_forma_original:

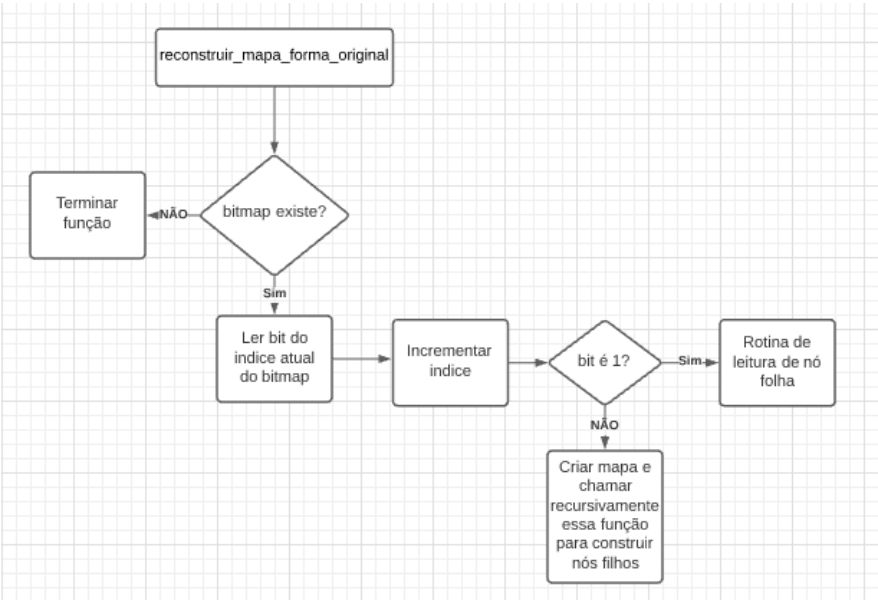

Figura 9: Funcionamento da função reconstruir_mapa_forma_original

10) traduzir_mensagem:

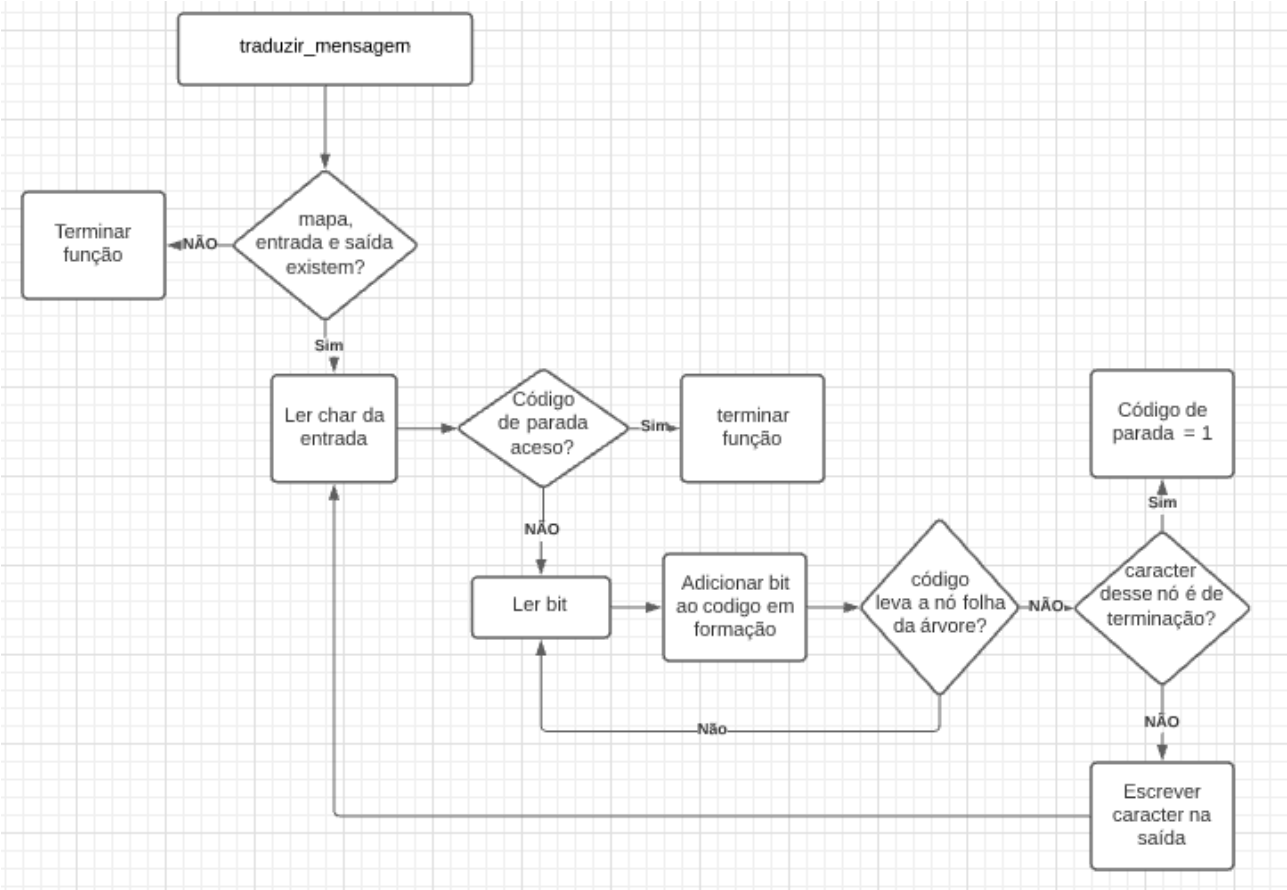

Figura 10: Funcionamento da função traduzir_mensagem

2.5 TOMADA DE DECISÕES

Uma das decisões primordiais para o funcionamento do programa foi entender o endianness a ser utilizado na gravação dos bits no arquivo compactado. Optou-se por utilizar a orientação de bit mais significativo primeiro para gravar os bitmaps utilizados no arquivo compactado. Essa se tratou em uma decisão importante por conta de um erro obtido durante a fase de criação das funções de descompactação do arquivo binário produzido. O erro consista na leitura do bit de maior significância e atribuição desse como bit de menor significância ao utilizar a função bitmapAppendLeastSignificant. Desse modo, gerava-se uma variável unsigned char espelhada em relação ao original (two's complement), o que interrompeu o fluxo normal de produção do trabalho por algum tempo.

Outra decisão tomada foi a de não implementar uma função de sorting para o buffer que armazena a frequência dos caracteres ASCII presentes no conteúdo do arquivo texto de entrada. O problema tratava-se que, ao se efetuar o sort na ordem decrescente (isso também ocorreria para ordem crescente), perdia-se a referência dos caracteres e suas respectivas frequências no arquivo de entrada, o que impossibilitava a continuidade do trabalho. A solução encontrada foi a de criar uma função que encontra o indice do bufer que apresenta o menor elemento dentre todos os presentes nesse. Desse modo, foi possível criar uma lista circular encadeada ordenada, sem que houvesse a necessidade de reorganizar o buffer que armazena as frequências dos caracteres lidos, o que demandou, no entanto, sacrificar a eficiência do algoritmo, que apresenta complexidade superior.

Há também o que se comentar sobre uma decisão tomada no formato de gravação da árvore de codificação de caracteres e da mensagem codificada no arquivo compactado. Durante a fase de implementação desse recurso, tropeçou-se numa dificuldade relacionada ao armazenamento em bits da árvore de codificação e da mensagem, que como se tratam de informações de tamanho váriavel, poderiam ocorrer erros na diferenciação de quando é finalizada a transmissão de bits da árvore e de quando é iniciada a transmissão de bits da mensagem codificada. Para resolver isso, foram estabelecidas as seguintes convenções: a transmissão do mapa em bits sempre será um número múltiplo de 8 e a mensagem codificada consistirá em todo o conteúdo restante do arquivo binário. Essas idéias simples facilitaram a decodificação do arquivo, pois otimizaram a identificação do divisor entre a transmissão do mapa e da mensagem feita pelas funções de reconstrução/tradução, que operam com mais segurança. Isso necessitou, contudo, em aumentar em alguns bits o tamanho total do arquivo compactado, que é de no máximo 7 bits, o que parece ser uma troca justa ao se considerar o ganho em segurança de código sobre a diferença no tamanho total.

3. DISCUSSÕES

A principal dificuldade encontrada no trabalho foi a de estender o conjunto de caracteres válidos na compactação. Nesse sentido, dispendeu-se um grande esforço em tentar ler caracteres do alfabeto romano, codificados pela tabela ASCII estendida, para que fosse possível codificar e decodificar imagens e outros tipos de arquivo, além dos arquivos de texto utilizados nos testes.

Inicialmente a implementação desse tweak parecia simples: bastaria modificar o valor do macro char_ascii_set para o valor 256 em vez de 128, dessa maneira imaginava-se que todos os caracteres da tabela ASCII estendida seriam lidos e armazenados no buffer de frequência dos caracteres disponíveis no arquivo texto e que o programa funcionaria normalmente para toda a rotina eventual de compactação e descompactação. No entanto, a representação de caracteres acentuados leva 4 bytes em memória (UNICODE), enquanto que os caracteres padrão ASCII requerem apenas 1 byte. Essa diferenciação na forma de armazenar caracteres inviabilizou a generalização dos arquivos de leitura, pois não seria possível alternar entre a leitura de um caracter ASCII e de um UNICODE em tempo de execução. No entanto, caso fossem fornecidos arquivos cujos caracteres romanos utilizassem a representação do ASCII estendido (1 byte), então o programa funcionaria normalmente para codificar e decodificar esse tipo de arquivo.

Além disso, notou-se que a estratégia de compactação escolhida para gerar o arquivo binário não é boa o suficiente para arquivos ou muito pequenos ou com poucas repetições de caracteres, por vezes até mesmo aumentando o tamanho do arquivo original. Com efeito, a mensagem codificada sempre será menor em tamanho de bits, no entanto o problema está no mapa de Huffmann que abre os arquivos binários compactados. A gravação desse mapa pelo programa Compactador no arquivo compactado pode consumir muitos bits, a ponto de se tornar desvantajoso para o processo de compactação quando o arquivo texto consome pouco espaço de armazenamento ou quando apresenta pouca repetição de caracteres.

Em vista disso, cabe a orientação/alerta de utilizar esse programa para arquivos preferencialmente com mais de 50 bytes e que apresentem caracteres que se repetem. Logo, nada de tentar representar o alfabeto! Esse caso teste, aliás, foi o que apresentou pior desempenho: o arquivo binário gerado apresenta o dobro do tamanho do arquivo original. Por isso, vale se atentar ao conteúdo do arquivo texto a ser compactado pela ferramenta.

4. CONCLUSÃO

Em conclusão, o trabalho serviu como uma ótimo oportunidade para aprender mais sobre listas encadeadas e estruturas de dados de forma a exercitar o pensamento profundo e extenso sobre o modo de produzir um código modularizado e estruturado, segundo a direção de listas, tipos estruturados e funções genéricas.

REFERÊNCIAS

MAHER, R.C. Command-line arguments in the C language. *In*: **Command-line arguments in the C language**. [*S. l.*], 2004. Disponível em: https://www.montana.edu/rmaher/ee475_fl04/Command_line.pdf. Acesso em: 18 set. 2021.

BITWISE Operators in C/C++. [*S. l.*], 2021. Disponível em: https://www.geeksforgeeks.org/bitwise-operators-in-c-cpp/. Acesso em: 19 set. 2021.

WHAT is binary mode of file operation and how to read/write in binary mode?. [*S. l.*], 2021. Disponível em: https://www.equestionanswers.com/c/c-binary-mode.php. Acesso em: 19 set. 2021.

READ and write to binary files in C?. [*S. l.*], 2013. Disponível em: https://stackoverflow.com/questions/17598572/read-and-write-to-binary-files-in-c. Acesso em: 20 set. 2021.

WHAT'S a binary file and how do I create one?. [*S. l.*], 2009. Disponível em: https://stackoverflow.com/questions/979816/whats-a-binary-file-and-how-do-i-create-one. Acesso em: 20 set. 2021.

Unsigned char in C with Examples. [*S. l.*], 2020. Disponível em: https://www.geeksforgeeks.org/unsigned-char-in-c-with-examples/. Acesso em: 19 set. 2021.

FWRITE. [*S. l.*], [2019?]. Disponível em: http://www.cmaismais.com.br/referencia/cstdio/fwrite/. Acesso em: 26 set. 2021.