UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA
ESTRUTURAS DE DADOS I

Gabrielly Barcelos Cariman
Lucas Manfioletti

COMPACTADOR/DESCOMPACTADOR DE HUFFMAN

Vitória-ES

Set./2021

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

Este trabalho tem como objetivo aprender de forma prática fundamentos ensinados na matéria de Estrutura de Dados I, ministrada pela Professora Dr. Patrícia Dockhorn Costa.

Nele são utilizados conceitos como Tipos Abstratos de Dados (TADs), Lista Duplamente Encadeada com Sentinela, leitura e escrita em arquivos, alocação dinâmica de memória, recursão e árvores, tudo utilizando a linguagem de programação C.

O problema apresentado para que fosse resolvido é implementar de forma simplificada um programa compactador e descompactador de arquivos. Nele, o usuário poderia inserir um arquivo, de forma a gerar um novo arquivo compactado, com base na codificação de Huffman. A codificação, visa reduzir o espaço ocupado por cada caractere, de forma a estabelecer a própria codificação com os caracteres utilizados com maior frequência, ocupando respetivamente os valores de menos espaço no arquivo. De forma análoga, o programa também seria capaz de restaurar o arquivo original, a partir de um arquivo compactado pelo mesmo.

Para isso, foi utilizado principalmente TADs para organizar as informações de Arquivos, Codificações, e Bitmap. Além de que alguns desses TADs eram organizados em Listas Duplamente Encadeadas com Sentinela, que também são TADs, uma para cada tipo diferente.

Cada um desses TADs possui dois arquivos, um onde ele é sinalizado que existe juntamente com suas funções específicas e documentação (extensão .h) e outro onde ele é de fato declarado e implementado (extensão .c), ou seja, onde está o código que possui a lógica de cada função.

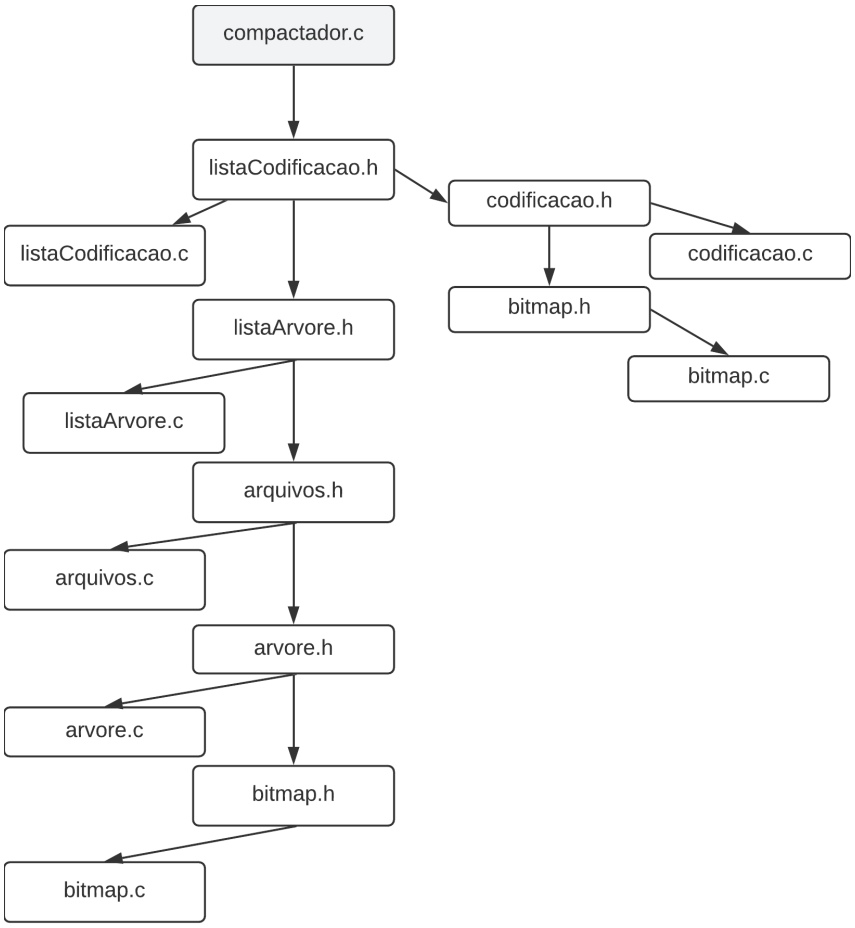
2 METODOLOGIA

2.1 Organização dos arquivos

Todos os diagramas apresentados neste trabalho foram feitos na plataforma Lucidchart ().

Os arquivos criados para a organização de todo o código do compactador estão na Figura 1 e a organização dos códigos do descompactador estão na Figura 2.

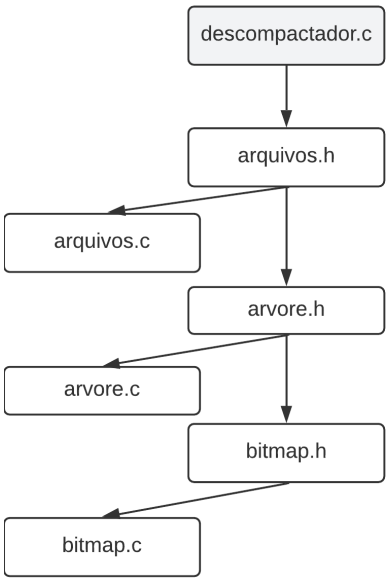
Figura 1 – Gráfico de Dependência do Compactador



Fonte: Produção do próprio autor.

Note que para cada TAD existe um arquivo '.h' e outro '.c', e como foi abordado na introdução, o primeiro contém a declaração dos seus métodos e sua documentação, e o segundo a sua definição com a implementação da lógica das funções.

Figura 2 – Gráfico de Dependência do Descompactador

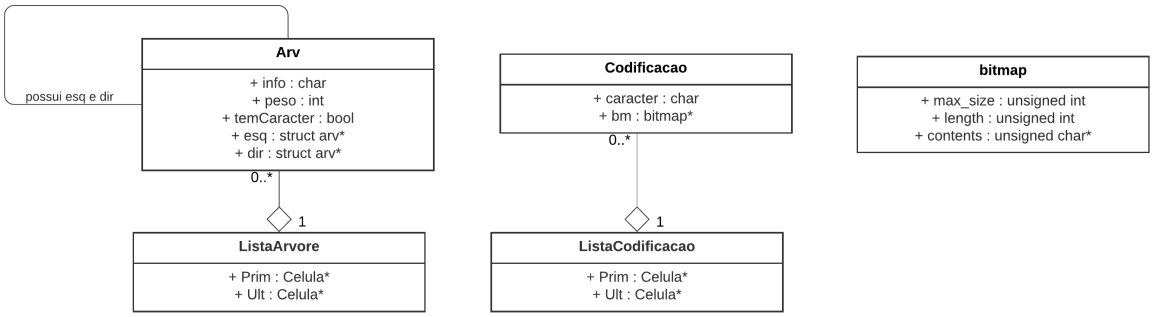


Fonte: Produção do próprio autor.

2.2 Modelagem

A modelagem dos TADs, o que cada um possui de atributos, métodos e os seus relacionamentos, pode ser observado no diagrama de structs na Figura 3.

Figura 3 – Diagrama de Structs



Fonte: Produção do próprio autor.

Analisando a Figura 3, percebe-se que os TADs possuem a seguinte estrutura (note que Celula não possui uma estrutura própria, isso será explicado posteriormente):

1. Arv - TAD que faz a organização de como um nó de uma árvore é armazenada na memória.

- info : char - armazena o caractere da árvore;
- peso : int - armazena a quantidade de vezes que o caractere se repetiu;
- temCaracter : bool - armazena se o nó possui um caractere válido lido ou não;
- esq : struct arv* - armazena o ponteiro para a estrutura do nó esquerdo da árvore;
- dir : struct arv* - armazena o ponteiro para a estrutura do nó direito da árvore.

2. ListaArvore - TAD que faz a organização de como uma lista de árvores é armazenada na memória.
   - Prim : Celula* - armazena o ponteiro para a primeira Celula da lista;
   - Ult : Celula* - armazena o ponteiro para a última Celula da lista.
3. Codificacao - TAD que faz a organização de como uma codificação é armazenada na memória.
   - caracter : char - armazena o caractere da codificação;
   - bm : bitmap* - armazena o ponteiro para o bitmap que possui a codificação do caractere.
4. bitmap - TAD que faz a organização de como um bitmap é armazenada na memória.
   - max_size : unsigned int - armazena o tamanho máximo em bits do bitmap;
   - length : unsigned int - armazena o tamanho atual em bits do bitmap;
   - contents : unsigned char* - armazena o conteúdo do mapa de bits do bitmap.

Todas as listas implementadas no trabalho são listas com sentinelas. Isso porque, essa estrutura apresenta maior segurança ao ser manipulado, por manter sempre o mesmo enderenço de memoria do primeiro elemento, a sentinela. Ademais, foi escolhido também que elas fossem duplamente encadeadas em vez de simplesmente, isso por que há necessidade de percorrer as listas no sentido contrário para que na inserção nas listas seja feita uma busca da posição certa para as Celulas já fiquem organizadas em ordem crescente de peso e de tamanho do bitmap. Dessa forma, pode ser mais econômico fazer essas tipo de inserção pelo final ou percorrer a lista ao contrário.

As listas com sentinelas possuem a TAD Celula com a seguinte estrutura:

1. Celula
   - dado: TipoDado*

- prox: Celula*

Cada lista possui sua própria implementação do TAD Celula, mas todos seguem a mesma lógica. Por exemplo: Na lista de Codificacao, ao invés de Celula possuir o atributo 'dado' do tipo 'TipoDado*', ela possui o atributo 'codificacao' (o nome do atributo não faz diferença, apenas é mais intuitivo colocar 'codificacao' nesse caso) do tipo 'Codificacao*'. E como dito, esse padrão se repete para as outras listas existentes.

Além dos structs apresentados, o trabalho conta ainda com os arquivos 'arquivos.h' e arquivos.c'. Esses arquivos não apresentam nenhum tipo de struct em sua definição, sua função é a manipulação de arquivos binários.

2.3 Funções Importantes

Como foi dito anteriormente, cada TAD possui uma gama de funções diferentes, que realizam diversas operações que são essenciais para o funcionamento do programa como um todo. Além disso, existe também o 'compactador.c' e o 'descompactador.c' que utilizam as funções dos TADs para compactar e descompactar os arquivos. A manipulação de arquivos binários foi visto em Raimundo () e a de entradas por meio do argv e argc foi visto em Unicamp ().

Simplificando, cada TAD tem suas respectivas funções de inicialização, retornar algum valor, liberar espaço na memória, e no caso das listas, inserção, remoção e retorno depois de uma busca.

É importante destacar que as funções de liberar espaço na memória e remoção acontecem da seguinte forma: Quando o cliente utiliza a função de liberar uma lista de codificações, por exemplo, além de liberar o espaço da lista em si, também é liberado cada uma das codificações. E ao utilizar a função de remover uma codificação da lista de codificações, por exemplo, a codificação removida não tem seu espaço de memória liberado, apenas a Celula tem.

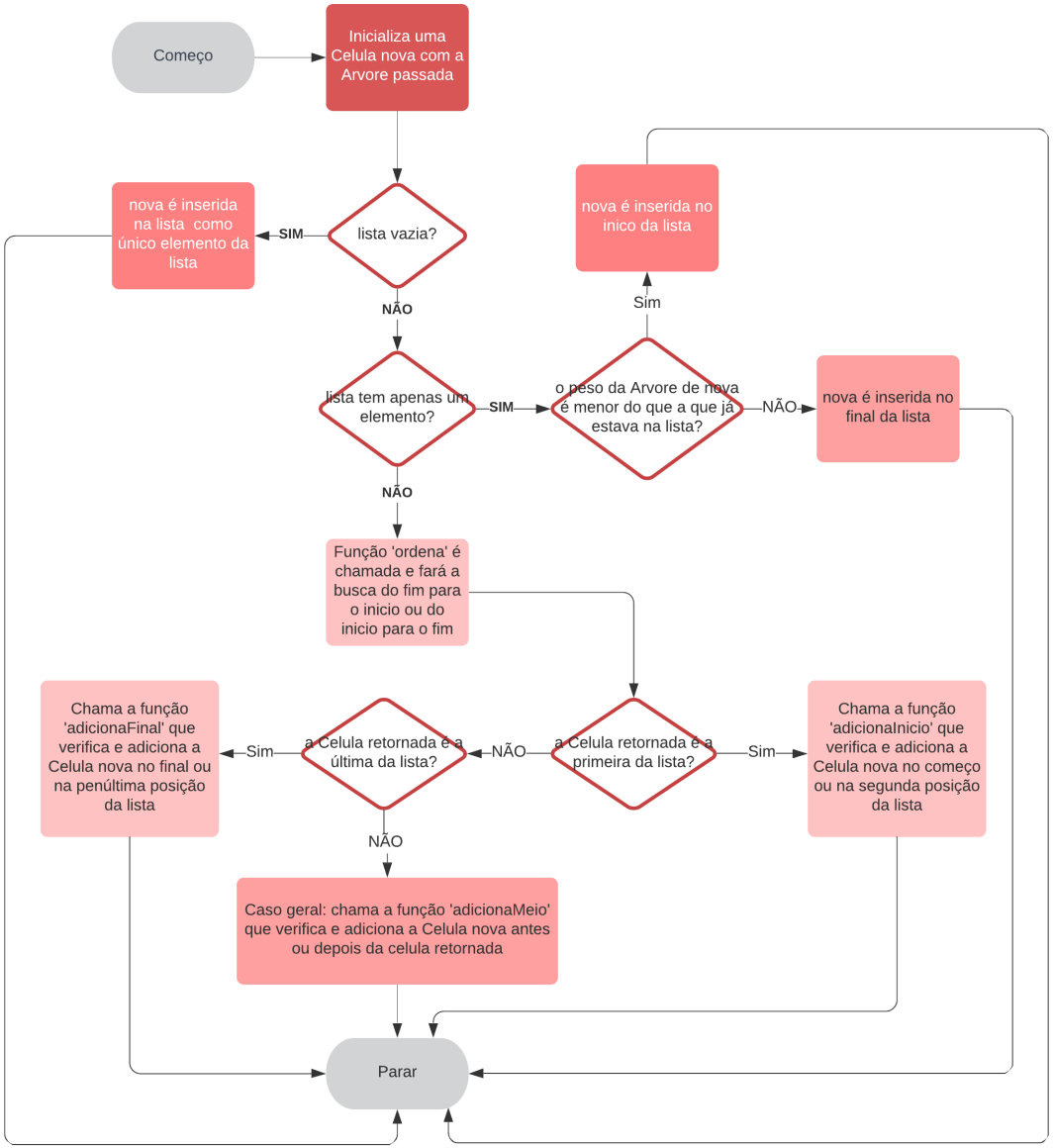
2.3.1 Compactador

As funções mais complexas e interessantes do compactador serão explicadas a seguir:

1. Função par definir a lista de árvores, de forma crescente:

    - **TAD ListaArvore: void defineListaArvore(ListaArvore *lista, int *repeticoes)** - a partir do vetor de inteiro com 256 elementos já definidor, lendo o arquivo passado em modo binário em que o índice do vetor é o código ASCII do caractere e o conteúdo é o número de vezes em que ele se repete, se o elemento desse vetor fosse diferente de zero, era criada uma árvore com o caractere sendo o índice do vetor, o peso sendo o conteúdo do elemento e os campos para os nós esq e dir da árvore eram nulos. Após isso, a função 'insereArvoreOrdenada' era chamada passando a lista, a árvore criada. Essa função vai fazer a inserção da árvore na lista de forma ordenada crescente de acordo com o peso da árvore como é apresentado no Flowchart desse algoritmo na Figura 4.

Figura 4 – Flowchart de insereArvoreOrdenada



Fonte: Produção do próprio autor.

2. Funções para definir toda a tabela de codificações do arquivo:

   - **TAD ListaCodificacao: void tabelaCodificacao(ListaCodificacao *listaCod, ListaArvore *listaArv)** - essa função calcula a altura da primeira árvore da lista passada e cria um bitmap com o tamanho da altura dessa árvore. A partir disso, a função estática 'defineTabelaCodificacao' é chamada passando a lista de codificações, o bitmap e a árvore.
   - **TAD ListaCodificacao: static void defineTabelaCodificacao(ListaCodificacao *listaCod, bitmap *bm, Arv *a)** - essa função recursiva tem como caso base a árvore apresentar um caractere válido, se ela tiver, um novo bitmap é criado com o mesmo conteúdo do outro bitmap. Após isso, uma codificação é criada com o novo bitmap e com o caractere da árvore. Essa codificação é inserida de forma ordenada na lista, seguindo o mesmo Flowchart da Figura 4.

     Caso a árvore não tenha um caractere válido, antes da função ser chamada recursivamente para o nó da esquerda, o bit 0 é adicionado ao final do bitmap e após o seu retorno o último bit é retirado pela função 'bitmapDeleteLeastSignificantBit'. O mesmo ocorre para o nó da direita, porém o bit 1 que é adicionado ao final do bitmap.

3. Funções para imprimir a quantidade de folhas e a árvore no arquivo binário:

   - **TAD ListaArvore: void imprimeArvoreBinario(ListaArvore *lista, char *nomeArquivo)** - primeiro, essa função exclui o arquivo compactado se ele já existir. Após isso, chama a função 'dec2bin', adaptada de Lacobus (2017), que recebe o número de folhas da árvore e retorna um bitmap com 8 bits com o conteúdo sendo o número de folhas em binário. Posteriormente, chama a função 'escreveCompactado' passando o bitmap e o nome do arquivo para ser adicionado ao final do arquivo o conteúdo do bitmap. Assim, o número de folhas da arvore ficará no inicio do arquivo depois dele a arvore será inserida no arquivo. Para isso, um bitmap será criado com o número máximo de bits de oito vezes a quantidade de folhas menos 1, para considerar 0 como uma folha e respeitar o máximo de 255 folhas, e o número máximos de nós que a árvore pode ter. Para fazer o bitmap com o conteúdo da árvore a função 'arvBinario' é chamada passando a árvore e o bitmap. Com o conteúdo do bitmap definido, a função 'escreveCompactado' é chamada mais uma vez e o o conteúdo do bitmap é adicionado ao final do arquivo compactado.
   - **TAD Arv: void arvBinario(Arv *a, bitmap *bm)** - essa função recursiva tem como caso base o nó da árvore ter um caractere válido. Caso ele tenha, o bit 1 é adicionado ao final do bitmap e valor em decimal na tabela ASCII do caractere da árvore é convertido em binário pela função 'dec2bin' e adicionado

ao final do bitmap. Caso o nó não seja folha, isto é, não possua um caractere válido, é apenas adicionado 0 ao final do bitmap.

4. Funções para imprimir a quantidade de caracteres, e a codificação no arquivo binário:

   - **TAD Arquivos: void imprimeArvoreBinario(ListaArvore *lista, char *nomeArquivo)** - essa função recebe os nomes dos arquivos, de leitura(antigo), e de escrita(novo), faz o calculo da quantidade de caracteres que o arquivo tem, através da função 'calculaQuantidadeDeCaracteres', que retorna o tamanho do arquivos e bytes, e por conseguinte, o a quantidade de caracteres. Por fim, esse valor é convertindo em binário pela função 'dec2bin', e gravado no arquivo através da função 'escreveCompactado'. Ademais, um marcador de encerramento é gravado para auxiliar na leitura posterior do arquivo binário.
   - **TAD ListaCodificação: void imprimeCodificacaoBinario(listaCod, argv[1], nomeArquivoNovo)** - a função cria um bitmap, chamando a função 'bitmapCompacta', que faz a comparação entre o caractere do arquivo, e a tabela de codificação, retornando o bitmap de cada caractere em um único bitmap "concatenado". Por fim, a função 'escreveCompactado' é chamada novamente, e grava o bitmap da codificação ao final do novo arquivo.

2.3.2 Descompactador

As funções mais complexas e interessantes do descompactador serão explicadas a seguir:

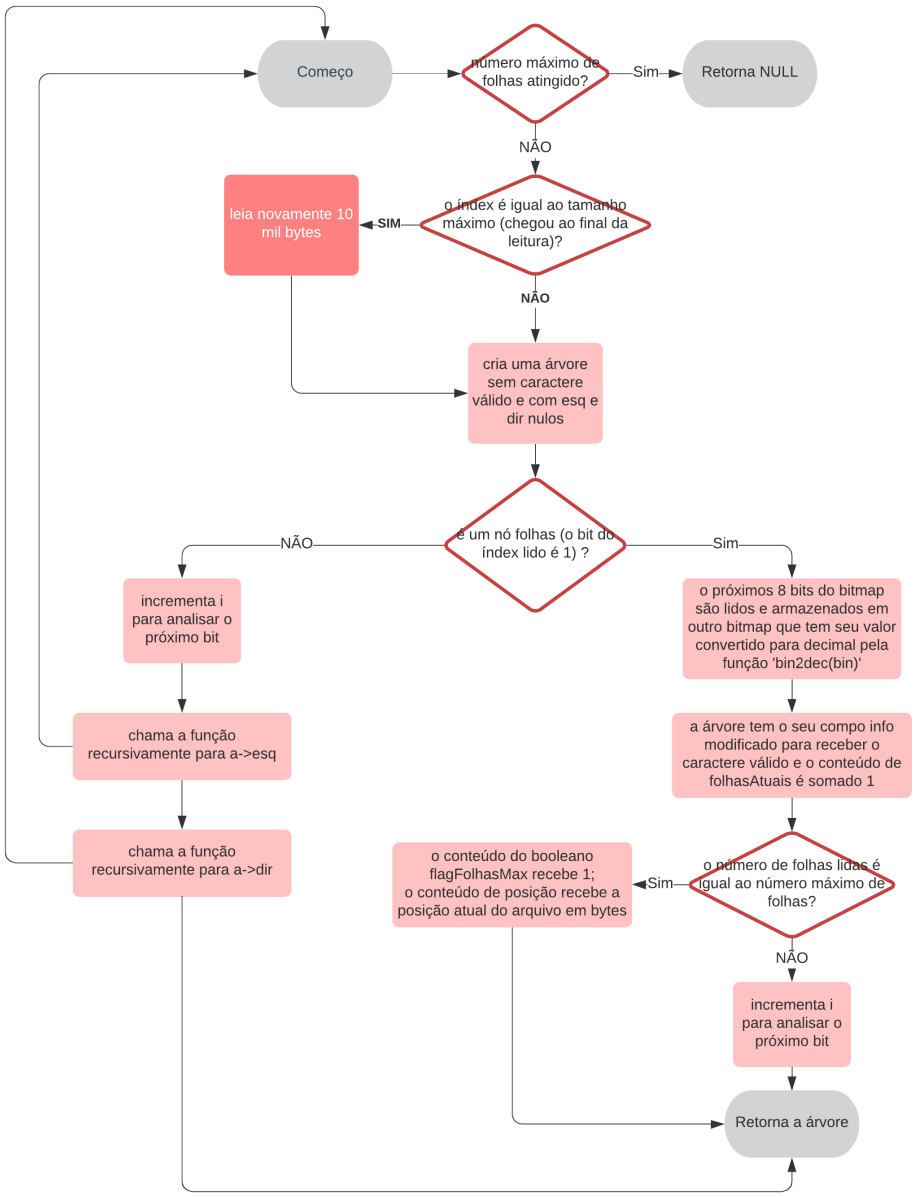
1. Funções para ler quantidade de folhas e armazenar a árvore do arquivo binário em uma estrutura árvore:

   - **Arquivos: Arv *leituraArvore(char *nomeArquivo, long *posicaoFimArv)** - essa função lê um byte do arquivo binário e chama a função 'bin2dec', adaptada de Bacco (2016), para converter em decimal e conseguir o número de folhas da árvore somado a 1, porque para imprimir no arquivo binário foi necessário retirar 1. Com o número total de folhas, a função 'redefineArvore' é chamada passando uma árvore, um bitmap com o tamanho máximo de 10 mil bytes lidos, o total de folhas, um ponteiro para a quantidade de folhas atuais, um ponteiro para um booleano flagMaxFolhas que verifica se a quantidade de folhas lidas já chegou ao máximo, um ponteiro para o índex do bit do bitmap que está sendo verificado no momento, ponteiro para o arquivo que está aberto, ponteiro para a quantidade de leituras feitas e ponteiro para a posição final da leitura da árvore no arquivo binário em bytes. É de suma importância que

as variáveis que terão os seus valores modificados dentro da função, como ser incrementada, sejam passadas como ponteiros. Isso porque, como se trata de uma função recursiva, ela irá apresentar vários escopos diferente. Então, caso as variáveis não fosse passadas como parâmetros, elas iriam apresentar diferentes valores em diferentes momentos da função. Isso ria prejudicar o seguimento da função.

- **TAD Arv: Arv *redefineArvore(Arv *a, bitmap *bm, int totalFolhas, int *folhasAtuais, bool *flagFolhasMax, int *i, FILE *arq, int *leituras, long *posicao)** - uma das funções mais complexas do programa, o Flowchart desse algoritmo é apresentado na Figura 5.

Figura 5 – Flowchart de redefineArvore



Fonte: Produção do próprio autor.

Essa função recursiva apresenta como primeiro caso base o número de folhas lido ser igual ao número máximo de folhas, retornando nulo, caso isso aconteça. A partir disso, se o conteúdo do índex lido for igual ao valor máximo do bitmap, isto é, se a leitura chegou ao final, é feita novamente uma leitura 10 mil bytes. Após isso, uma árvore sem um caractere válido e com os campos 'esq' e 'dir' nulos é criada.

Posteriormente, uma verificação é feita se o bit analisado é 1, isto é, se é um nó folha. Se ele for, os próximos 8 bits são lidos e convertido em decimal para modificar o campo 'info' da árvore e adicionar um caractere válido. O número de folhas atuais é incrementado e é verificado se essa valor é igual ao número total de folhas, se ele for, 'flagFolhasMax' recebe 1 e posição recebe a posição atual do arquivo em bytes, se não for, o índex é incrementado. Por fim, a árvore é retornada. Caso não seja um nó folha, i é incrementado e a função é chamada recursivamente para a->esq e para a->dir. Finalmente, a árvore é retornada.

2. Funções para ler a quantidade de caracteres, e restaurar o arquivo original:

   - **Arquivos: void imprimeArquivoOrignal(argv[1], nomeArquivoNovo, arv, posicaoFimArv)** - essa função recebe 4 variáveis, o nome do arquivo a ser lido, o nome do arquivo a ser restaurado(novo), a árvore criada pela codificação de Huffman e a posição de leitura no arquivo. Os arquivos são abertos, e a posição de leitura no arquivo é ajustada para continuação, após leitura da árvore na função anterior a mesma. O número de caracteres é lido e convertido para decimal através da função 'bin2dec', até encontrar o caractere de separação do cabeçalho. Em seguida, a função avança para leitura da codificação, usando a função 'retornaCaracter' para percorrer a árvore de acordo com os valores lidos no arquivo binário de forma recursiva, que faz a busca pela árvore até encontrar o nó correspondente ao caminho, que contenha um caracter, ao final chamando a função 'escreveBinario' para restaurar aquele determinado caracter no arquivo original. O processo é realizado até que se alcance a quantidade original de caracteres.

3 CONCLUSÃO

O trabalho com certeza foi de grande importância para fixar todos os conceitos abordados pela disciplina. Além de servir como fixação, foi preciso uma pesquisa por conteúdo adicional, como a parte de manipulação de binários e arquivos em C.

Os maiores desafios presente no trabalho foram conseguir pensar em como implementar a função 'insereArvoreOrdenada' e a 'redefineArvore', assim como quais passos seguir para isso. Além disso, a manipulação binários, arquivos binários e funções recursivas para manipular estruturas do tipo árvore levaram algum tempo para serem pensados, pesquisados, implementados e testados até que estivessem finalizados.

Trabalhos assim são de grande relevância durante a graduação, pois aproxima a teoria aos problemas encontrados no mundo e gera um sentimento de que a implementação poderia ser utilizado para resolver um problema real.

REFERÊNCIAS

BACCO. Como converter binário em decimal? 2016. Disponível em: <https://pt.stackoverflow.com/questions/152947/como-converter-bin%C3%A1rio-em-decimal>. Acesso em: 04 out. 2021. Citado na página 11.

LACOBUS. Conversão decimal em binário em linguagem C. 2017. Disponível em: <https://pt.stackoverflow.com/questions/216128/convers%C3%A3o-decimal-em-bin%C3%A1rio-em-linguagem-c>. Acesso em: 04 out. 2021. Citado na página 10.

LUCIDCHART. Disponível em: <https://www.lucidchart.com/pages/pt/landing?utm_source=google&utm_medium=cpc&utm_campaign=_chart_pt_allcountries_mixed_search_brand_exact_&km_CPC_CampaignId=1500131167&km_CPC_AdGroupID=59412157138&km_CPC_Keyword=lucidchart&km_CPC_MatchType=e&km_CPC_ExtensionID=&km_CPC_Network=g&km_CPC_AdPosition=&km_CPC_Creative=294337318298&km_CPC_TargetID=kwd-33511936169&km_CPC_Country=1031797&km_CPC_Device=c&km_CPC_placement=&km_CPC_target=&mkwid=seFG43w3S_pcrid_294337318298_pkw_lucidchart_pmt_e_pdv_c_slid___pgrid_59412157138_ptaid_kwd-33511936169_&gclid=CjwKCAjw9uKIBhA8EiwAYPUS3Ea5dWA8cWLzUHVOTkbkMxw5RMQwUEH2ItcKpVxdm18U7hTBwE>. Acesso em: 15 ago. 2021. Citado na página 4.

RAIMUNDO, M. C. Manipulação de Arquivos Binários em C. Disponível em: <http://wiki.icmc.usp.br/images/1/1c/SCC060702219Matheus.pdf>. Acesso em: 04 out. 2021. Citado na página 7.

UNICAMP, I. de C. Arquivos em C e Parâmetros do Programa. Disponível em: <https://www.ic.unicamp.br/~lucas/teaching/mc102/2017-1/aula27.pdf>. Acesso em: 02 set. 2021. Citado na página 7.