COMPACTADOR DE HUFFMAN
TRABALHO 02 DE ESTRUTURA DE DADOS

PROFESSORA: PATRÍCIA DOCKHORN COSTA
ALUNAS: BÁRBARA ALENCAR DE ARAUJO PEREIRA E MARIA EDUARDA NOIA MATTOS DE AZEVEDO

INTRODUÇÃO

O objetivo deste trabalho, é implementar um compactador e descompactador de arquivos baseado na codificação de Huffman.

O sistema de Huffman, se baseia em contar a frequência de caracteres de um arquivo, e criar uma hierarquia na conversão dos caracteres de tal forma que aqueles que aparecem com menos frequência no texto possuam um menor comprimento em binário, ocupando assim, menos espaço de memória e reduzindo o tamanho do arquivo.

FUNCIONAMENTO GERAL

O programa é dividido em 5 TAD's e 2 funções main, que serão compiladas separadamente para gerar o compactador e o descompactador utilizando o makefile. São eles: lista, árvore, compactador, descompactador, bitmap, mainCompacta e mainDescompacta. O único TAD importado foi o bitmap, disponibilizado pela professora para auxiliar na manipulação de bits. Dividindo os TAD's, o makefile gera dois executáveis: compacta (o compactador) e descompacta (descompactador).

O compactador funciona da seguinte maneira: em primeiro momento é feita a leitura do arquivo de texto, e é contada a frequência dos caracteres e a quantidade total de caracteres no arquivo. Em seguida, utiliza-se o TAD árvore para criar nós-folha que armazenam os caracteres, e a sua frequência no texto - que é chamada também de "peso" do nó. O papel do TAD lista é gerar uma lista de nós-folha organizados em ordem crescente pelo peso de cada nó, para que possa ser utilizada pelo TAD árvore para realizar a montagem da árvore binária pelo processo sugerido no documento disponibilizado pela professora, ilustrado pela figura 1. De acordo com essa forma de organização, os nós com maior peso - que aparecem com maior frequência no texto - tendem a ficar mais no topo da árvore, gerando um código binário menor.

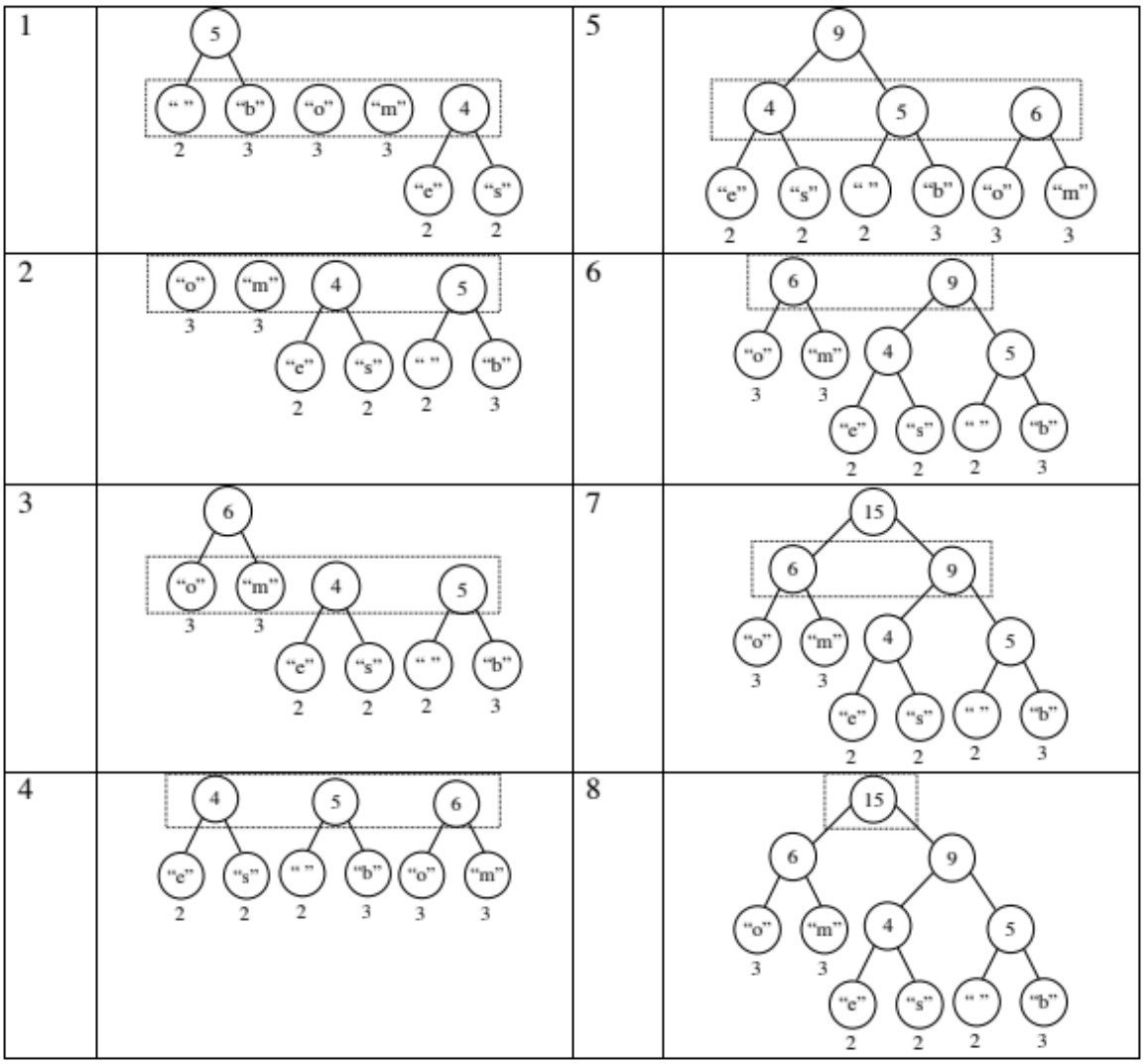

FIGURA 1

Feita a organização da árvore de compactação, o TAD árvore navega pelos ramos da árvore para criar uma tabela de conversão de caracteres, que gera uma correspondência entre o caractere lido e o seu valor em binário compactado correspondente. A regra para a montagem da tabela é simples, o 0 representa o caminho à esquerda e o 1 à direita, isso pode ser observado associando as figuras 1 e 2.

Char	Binário
b	111
o	00
m	01
e	100
s	101
Espaço	110

FIGURA 2

Em seguida, o programa abre novamente o arquivo de entrada, lê os seus caracteres e realiza a escrita no arquivo de saída com o binário codificado correspondente. Para que seja possível realizar a descompactação do arquivo, o compactador escreve toda a árvore binária no cabeçalho do arquivo, e a quantidade de caracteres total presente no texto

Já o descompactador, tem o objetivo de realizar o processo inverso da compactação. Para isso, o programa lê a árvore no cabeçalho do arquivo e utiliza do TAD árvore para remontar a árvore dentro do programa e poder decodificar o texto. Do cabeçalho, também é lida a quantidade de caracteres total presente no arquivo para saber onde termina o texto e ignorar o lixo contido no binário.

Montada a árvore, o programa inicia a leitura do binário e a escrita dos caracteres pertencentes ao arquivo original. Para a navegação na árvore de codificação, se obedece novamente a regra de que o 0 representa a esquerda e o 1 a direita. A cada caractere (nó-folha) encontrado, a árvore retorna ao seu início para ser percorrida para o próximo caractere, e como forma de controle, os caracteres são contados para assim que atingirem a sua quantidade original pararem de ser escritos, garantindo fidelidade ao original.

IMPLEMENTAÇÃO

Em uma visão geral sobre os arquivos contidos no programa, tem-se:

- **bitmap.c e bitmap.h**

TAD pronto disponibilizado pela professora, responsável pelas operações de manipulação e gerenciamento de mapas de bits.

- **lista.c e lista.h**

Responsável pela manipulação, criação e liberação das listas encadeadas utilizadas no código. Será utilizado principalmente na etapa de ordenação para a montagem da árvore.

- **arvore.c e arvore.h**

Responsável pela manipulação, criação e liberação de árvores feitas no código. Também faz a escrita e leitura de cabeçalho nos arquivos e faz a criação da tabela de codificação de caracteres.

- **compactador.c e compactador.h**

Responsável pela abertura e leitura do arquivo de entrada, contagem de caracteres, e utiliza os tads anteriores para gerar a árvore de compactação, a tabela de codificação e para gerar o arquivo de saída .comp.

- **descompactador.c e descompactador.h**

Responsável pela reversão do procedimento feito no compactador, leitura do cabeçalho do arquivo, recriação da árvore de codificação e geração do arquivo descompactado análogo ao original antes da compactação.

A descrição da implementação detalhada do programa será feita seguindo o fluxo da função main, destacando as principais funções utilizadas para a realização da compactação/descompactação do arquivo.

SEÇÃO 1: COMPACTADOR

Para o compactador, tem-se como função main:

```c
int main(int argc, char *argv[]) {

    // etapa de leitura do arquivo e criação da árvore de compactação

    if (argc < 2){
        printf("Uso: %s <nome_arquivo>\n", argv[0]);
        return 0;
    }

    char caminhoArquivo[TAM_NOME_CAMINHO];
    strncpy(caminhoArquivo, argv[1], TAM_NOME_CAMINHO - 1);
    Arv* a = criaArvoreCompactacao(caminhoArquivo);

    // etapa de criacao da tabela de compactação e realização da compactação
    do arquivo

    bitmap** tabela = iniciaTabelaCodificacao(a);

    int bytes = contaCaracteresTotal(caminhoArquivo);

    compactaArquivo(a, tabela, caminhoArquivo, bytes);

    printf("\nCompactação do arquivo %s realizada com sucesso!\n",
    caminhoArquivo);

    // liberação de memoria do programa
    liberaArvore(a);
    liberaTabela(tabela, TAM_ASCII);

    return 0;
}
```

FIGURA 3

A função main recebe o arquivo de entrada como argv[1] e copia esse caminho para a string “caminhoArquivo”. A máquina pode ser dividida em três etapas: (1) leitura do arquivo e criação da árvore de compactação, (2) criação da tabela de codificação, (3) compactação do arquivo e (4) liberação do programa. Etapa 1: leitura do arquivo e criação da árvore de compactação

ETAPA 1: LEITURA DO ARQUIVO E CRIAÇÃO DA ÁRVORE DE COMPACTAÇÃO

A partir do caminho do arquivo de entrada, é chamada a função “criaArvoreCompactacao”

```
Arv* criaArvoreCompactacao(char* caminhoEntrada){
    char caminhoArquivo[TAM_NOME_CAMINHO];
    strncpy(caminhoArquivo, caminhoEntrada, TAM_NOME_CAMINHO - 1);

    int V[TAM_ASCII] = {0};
    int qtd = contaFrequenciaCaracteres(caminhoEntrada, V, TAM_ASCII);

    Lista *l = iniciaFolhas(V, TAM_ASCII, qtd);

    Arv *a = organizaArvore(l);

    liberaLista(l);

    return a;
}
```

FIGURA 4

Nela, cria-se um vetor V de mesmo tamanho da tabela ASCII estendida, para contar a frequência dos caracteres do arquivo de entrada. Dessa forma, o index desse vetor corresponde ao caractere equivalente da tabela ASCII, e o conteúdo do vetor nesse índice seria a frequência em que o caractere aparece no arquivo de texto. Por exemplo, o caractere “@” (de índice 64 na tabela ASCII) que se aparecesse 2 vezes no texto, teria a sua frequência armazenada em V[64] = 2. Essa contagem da frequência de caracteres é feita pela função “contaFrequenciaCaracteres”, que usa um loop ‘for’ para varrer o arquivo e acrescentar +1 ao conteúdo do índice do vetor toda vez que identifica no arquivo o caractere correspondente.

```
struct arvore {
    unsigned char caracter;
    int peso;
    int tipo;
    Arv *esq;
    Arv *dir;
};
```

FIGURA 5

Em seguida, a função “iniciaFolhas” varre o vetor de frequência e para cada caractere com frequência diferente de 0, ele cria uma estrutura do tipo árvore e inicializa o nó com as informações frequência (peso) e o próprio caractere, e o tipo do nó (0 para folha ou 1 para nó interno).

Cada caractere diferente presente do texto se torna um nó-folha e é inserido em uma lista de nós, que posteriormente será utilizada para montar a árvore.

As funções de lista seguem uma lógica para favorecer a organização da árvore. A lógica da função “insereLista”, insere os elementos na lista em ordem crescente de acordo com o seu peso. E a função “retiraLista”, retira sempre o primeiro elemento da lista, e consequentemente o de menor peso.

A função “organizaArvore” que se segue recebe como parâmetro a lista de folhas inicializadas e organizadas, e realiza o procedimento sugerido na figura 1. Basicamente, os dois primeiros nós são retirados da lista, e são inseridos como subárvore esquerda e direita de um novo nó que é criado e tipado como nó-interno, cujo peso é a soma das frequências dos nós-filhos. Esse nó-pai criado, é inserido novamente na lista, e o processo se repete até que exista apenas um elemento na lista de nós — e consequentemente esse último nó será a árvore final.

ETAPA 2: CRIAÇÃO DA TABELA DE CODIFICAÇÃO

Voltando à main, inicia-se um vetor de bitmap que será a tabela de codificação. A função “inciaTabelaCodificacao” serve somente para alocar memória e fazer as devidas inicializações, e em seguida ela chama a função “criaTabelaCodificacao” para criar os códigos.

A “criaTabelaCodificacao” percorre a árvore de compactação gerada anteriormente. O fluxo é: cria-se um bitmap auxiliar inicialmente nulo para armazenar o binário a ser escrito. Durante a navegação na árvore, toda vez que se desce para o nó esquerdo, utiliza-se das funções do tad bitmap para se adicionar

um bit 0 ao final do bitmap auxiliar, e toda vez que se desce para o nó direito, se insere 1 ao final do bitmap auxiliar. Para que isso aconteça de forma recursiva foi necessário implementar uma função extra no TAD bitmap, que remove o último bit após entrar em cada lado, já que não é possível estar ao mesmo tempo na direita e na esquerda de um nível da árvore.

E esse processo se repete até que se encontre um nó-folha, o que quer dizer que se encontrou o caractere correspondente ao código do bitmap. Esse processo é exemplificado no esquema da figura 6. Assim que encontrado um caractere válido, o bitmap auxiliar é copiado para o vetor de bitmap (a tabela de codificação) no índice correspondente ao valor do caractere na tabela ASCII.

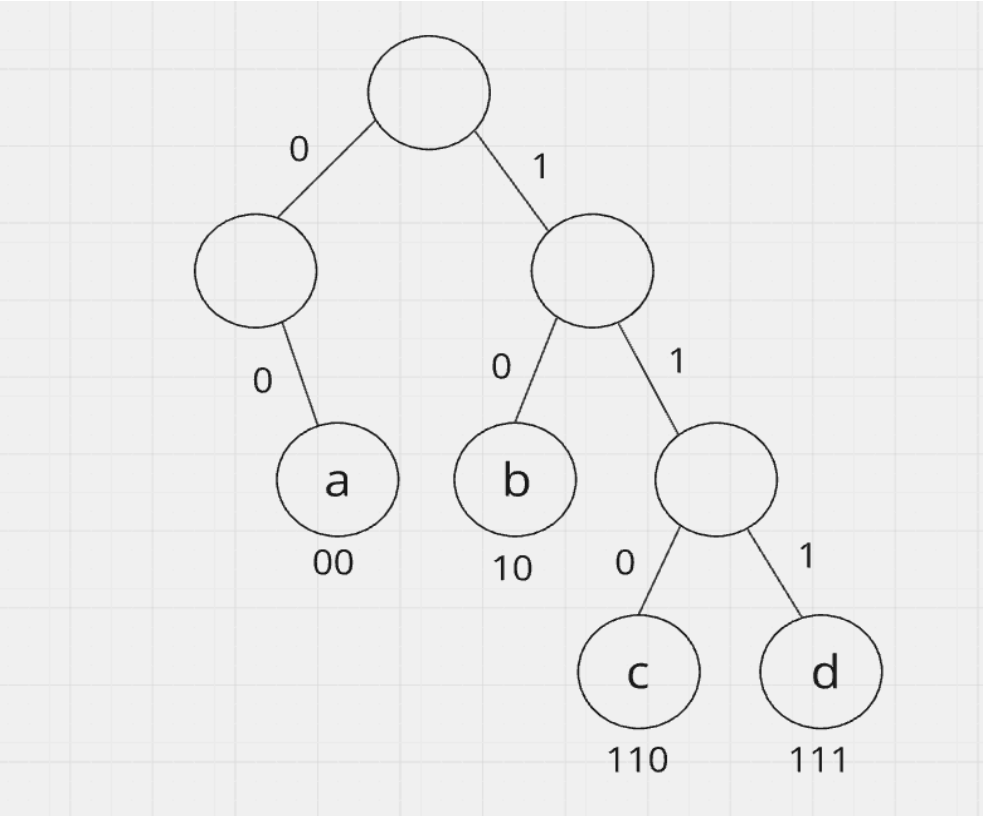

FIGURA 6

ETAPA 3: COMPACTAÇÃO DO ARQUIVO

Essa etapa é realizada dentro da função “compactaArquivo” — ela recebe como parâmetro a árvore, a tabela, o caminho do arquivo de entrada e o número total de caracteres no arquivo de entrada.

Em primeiro instante, a compactaArquivo realiza a tarefa de abrir o de entrada para que ele possa ser lido e codificado no arquivo de saída. Para que isso

aconteça, a função também gera um arquivo de saída e modifica o seu nome, para que seja uma concatenação do nome do arquivo de entrada com a extensão ".comp" no final, indicando que ele é o arquivo compactado.

```
strcpy(caminhoSaida, caminhoArquivo);
strcat(caminhoSaida, ".comp");
```

FIGURA 7

Após a abertura do arquivo de entrada e saída, é feita a escrita do cabeçalho do arquivo. Primeiramente, chama-se a função "escreveArvoreCabecalho" que realiza a tarefa de escrever a árvore no cabeçalho do arquivo. Essa função trabalha de forma recursiva navegando pela árvore e escrevendo as informações de cada nó no arquivo. O fluxo de escrita da árvore funciona da seguinte forma: a função recebe a árvore a ser escrita, e caso a árvore seja nula, se escreve no arquivo o bit 0, para indicar que não existem informações da árvore a serem lidas. Caso o nó da árvore acessado não seja nulo, se escreve no arquivo o bit 1 que indica que em seguida vão estar escritos o tipo do nó — escreve 0 se é folha ou 1 se for nó interno — e se o nó presente for folha, se escreve em binário o caractere contido nele. O processo se repete de forma recursiva até que toda a árvore tenha sido escrita no cabeçalho.

Após a escrita da árvore no cabeçalho, se escrevem também as informações sobre o bitmap utilizado para armazenar a árvore, e por último escreve a quantidade total de caracteres presente no arquivo original. A contagem do total de caracteres é feita para marcar o final do bitmap, para que o lixo não seja escrito no arquivo de saída. Essa ação se faz necessária pelo fato de que nem sempre serão lidos do arquivo um total de caracteres em uma quantidade exatamente igual ao tamanho do bitmap que vai armazená-los. Portanto, quando a quantidade de caracteres lidos é inferior ao tamanho do bitmap, o sistema o completa o bitmap com caracteres quaisquer - o chamado lixo. Nossa função então ao passar esse parâmetro é evitar a escrita desse lixo, passando o número de caracteres total.

Após a escrita do cabeçalho, declara-se duas variáveis que serão importantes (unsigned char) "bitBuffer" e (int) "bitCount", inicializadas para armazenar os bits que serão escritos no arquivo compactado e contar quantos bits foram armazenados, respectivamente.

```c
unsigned char bitBuffer = 0;
int bitCount = 0;

// loop que escreve os bits no arquivo compactado
while (1) {
    // lê 1MB de caracteres por vez do arquivo de entrada e armazena em um buffer
    size_t bytesLidos = fread(charBuffer, sizeof(unsigned char), MEGA_BYTE, entrada)

    if (bytesLidos == 0) {
        break;
    }

    // varre o buffer e busca na tabela de codificação o caractere equivalente
    for (size_t i = 0; i < bytesLidos; i++) {
        unsigned char caractere = charBuffer[i];
        bitmap *caractereCompactado = tabela[(int)caractere];

        // com o auxilio do bitmap, escreve no arquivo binario a versao compactada
        do caractere lido
        for (unsigned int j = 0; j < bitmapGetLength(caractereCompactado); j++) {
            unsigned char bit = bitmapGetBit(caractereCompactado, j);
            bitBuffer = (bitBuffer << 1) | bit;
            bitCount++;

            if (bitCount == 8) {
                fwrite(&bitBuffer, sizeof(unsigned char), 1, compactado);
                bitBuffer = 0;
                bitCount = 0;
            }
        }
    }
}

// garante que os bits restantes no bitBuffer sejam escritos no arquivo compactado
if (bitCount > 0) {
    bitBuffer <<= (8 - bitCount);
    fwrite(&bitBuffer, sizeof(unsigned char), 1, compactado);
}
```

FIGURA 8

Em seguida um *loop* 'while' é iniciado para ler o arquivo de entrada em blocos de 1 MB, armazenando os dados lidos no buffer charBuffer. Isso é feito para evitar que o tratamento dos caracteres seja feito durante a leitura do arquivo (que é feita em memória secundária) e evitar altos tempos de execução. Então, são lidos 1MB de caracteres por vez e armazenados no buffer, e trabalha-se com o conteúdo do buffer. A leitura é feita usando a função fread, que retorna o número de bytes lidos, assim o loop continua até que fread retorne 0, indicando que acabaram os caracteres a ser lidos no arquivo.

Para cada byte consumido no charBuffer, a função busca o bitmap de codificação correspondente na tabela de codificação (tabela). Este bitmap representa a codificação compactada do caractere atual. A função então percorre cada bit do bitmap, adicionando-o ao bitBuffer. Os bits são inseridos de forma que o bit mais significativo é adicionado primeiro.

Assim, a cada 8 bits adicionados ao bitBuffer, este é escrito no arquivo compactado, e após a escrita, o bitBuffer é resetado e o bitCount é reiniciado. Se o número de bits processados não for múltiplo de 8, o buffer é ajustado para garantir que todos os bits restantes sejam escritos corretamente. Isso é feito ao final do loop, antes do fechamento dos arquivos.

Após o loop de leitura e compactação, qualquer bit restante no bitBuffer que não formou um byte completo é ajustado e escrito no arquivo. A função faz isso deslocando bitBuffer para a esquerda e escreve o byte resultante no arquivo compactado.

Finalmente, a memória alocada para charBuffer é liberada, o bitmap é liberado e ambos os arquivos (entrada e compactado) são fechados. Com isso, o arquivo de entrada é compactado e salvo com a extensão ".comp".

ETAPA 4: LIBERAÇÃO DE MEMÓRIA

Esta etapa é apenas para a finalização do programa. Libera-se primeiro a memória alocada para a árvore de forma recursiva pela função “liberaArvore”, e em seguida libera-se a memória da tabela a partir da função “liberaTabela” que varre vetor de bitmap e chama a função “bitmapLibera” para cada posição.

SEÇÃO 2: DESCOMPACTADOR

Para o descompactador, tem-se como função main:

```c
int main(int argc, char *argv[]) {

    if (argc < 2){
        printf("Uso: %s <nome_arquivo>\n", argv[0]);
        return 0;
    }

    char caminhoEntrada[TAM_NOME_CAMINHO];
    strncpy(caminhoEntrada, argv[1], TAM_NOME_CAMINHO - 1);

    int tam = strlen(caminhoEntrada);
    unsigned char caminhoSaida[tam];
    strcpy(caminhoSaida, caminhoEntrada);
    caminhoSaida[tam - 5] = '\0';

    FILE *compactado = fopen(caminhoEntrada, "rb");

    if(!compactado){
        printf("Nao foi possivel abrir o arquivo compactado!\n");
        return 0;
    }

    descompactaArquivo(compactado, caminhoSaida);

    fclose(compactado);

    return 0;
}
```

FIGURA 9

A função main inicia criando uma string para o caminho do arquivo de entrada e para o caminho do arquivo de saída. Para o nome do arquivo de saída remove-se a extensão .comp para voltar ao nome do arquivo original. O arquivo compactado é então aberto e chama-se a função “descompactaArquivo” para realizar a descompactação.

Assim que o arquivo de saída é aberto, a função lê o cabeçalho do arquivo para ler o tamanho do bitmap armazenado no arquivo compactado e calcula o tamanho em bytes necessário para armazenar o bitmap que irá construir a árvore binária. A árvore de compactação é reconstruída a partir do bitmap usando a função leArvoreCabecalho, que lê a árvore do arquivo fazendo o processo reverso do “escreveArvoreCabecalho” e então retorna a árvore reconstruída. A diferença mais significativa entre a função de leitura e escrita de cabeçalho é que a função de leitura de cabeçalho precisa “andar” no bitmap lido, acrescendo o index do vetor de bitmap, para poder criar a árvore segundo as informações lidas em ordem.

Logo após a leitura e reconstrução da árvore binária, o número total de bytes originais é lido para garantir que a função descompactadora saiba quantos bytes devem ser escritos no arquivo de saída.

Encerrada a leitura do cabeçalho, inicia-se um loop para a leitura da codificação do arquivo original:

```
Arv *noAtual = a;
unsigned char byte;
int bytesEscritos = 0;
unsigned char bitBuffer = 0;

// passa pelo arquivo bit a bit
while (fread(&byte, sizeof(unsigned char), 1, entrada) ==
1) {
    for (int bit = 7; bit >= 0; bit--) {
        unsigned char mask = 1 << bit;
        unsigned char bitValue = (byte & mask) >> bit;

        // se escreveu todos os bytes, para
        if (bytesEscritos == bytes)
            break;

        // navega na arvore com base no bit
        noAtual = percorreArvore(noAtual, bitValue);

        // se achou folha escreve o caractere
        if (ehFolhaArvore(noAtual)) {
            unsigned char c = retornaCaracterArvore
            (noAtual);
            fwrite(&c, sizeof(unsigned char), 1,
            arquivoSaida);
            noAtual = a; // retorna pro inicio da arvore
            para o próximo caractere
            bytesEscritos++;
        }
    }
}
```

FIGURA 10

Inicializa-se as variáveis (Arv*) “noAtual”, utilizada para marcar nó atual que está sendo visitado na varredura pela árvore; (unsigned char) “byte”, que armazena o byte lido do arquivo compactado e (int) “bytesEscritos”, utilizada para contar os bytes escritos no arquivo de saída.

A leitura do restante do arquivo se faz por um *loop* “while” que continua a leitura até que todos os bytes do arquivo tenham sido lidos.

Em seguida, o *loop* “for” percorre cada bit do byte lido, começando pelo bit mais significativo (bit 7) até o menos significativo (bit 0). Faz-se também uma máscara “mask” para extrair o bit específico do byte atual. E a variável “bitValue” representa o valor do bit extraído (0 ou 1), obtido aplicando a máscara ao byte e deslocando o resultado para a direita.

```
for (int bit = 7; bit >= 0; bit--) {
    unsigned char mask = 1 << bit;
    unsigned char bitValue = (byte & mask) >> bit;
```

FIGURA 11

A navegação na árvore binária é feita através da função “percorreArvore”, utilizando o valor do bit (bitValue). Isto é, se bitValue é 0, o ponteiro noAtual se move para o filho esquerdo e se é 1, se move para o filho direito.

A cada percurso na árvore que é realizado, é feita a verificação do nó atual para saber se a função chegou em uma folha ou não. Caso o nó atual seja uma folha, significa que a sequência de bits lidos indica um caractere da arvore, e se escreve o mesmo no arquivo de saída. Feito isso, o nó atual volta a ser o topo da árvore e o processo se reinicia até que todos os caracteres codificados sejam lidos e transcritos.

Finalmente, após lidos todos os caracteres do arquivo de entrada, a memória alocada para o bitmap e a árvore é liberada, e os arquivos de entrada e saída são fechados.

CONSIDERAÇÕES FINAIS

A principal dificuldade encontrada no trabalho foi a compactação de arquivos de imagem. Não foi possível realizar um tratamento eficiente para esses casos e foi percebido que não houve uma compactação do arquivo, pois na maioria dos testes o tamanho aumentou ou permaneceu próximo do original. Não entendemos exatamente os motivos para esse comportamento do programa e as tentativas de conserto foram falhas.

Quanto à implementação do programa, a maior dificuldade foi entender o funcionamento do bitmap e fazer a escrita e leitura em binário, pois a manipulação de bits em C não foi ensinada na faculdade, então tivemos que descobrir como funciona por conta própria ou perguntando para colegas de classe poder concluir o trabalho.

BIBLIOGRAFIA

JOGOS & PROGRAMAÇÃO. Máscara de Bits C/C++. [S.l.: s.n.], 2 anos atrás. Disponível em: https://www.youtube.com/watch?v=RlPn-YQF1ok. Acesso em: 20 jul. 2024.

PROGRAME SEU FUTURO. Curso de Programação C | Deslocamento à esquerda, à direita, operações bit a bit | aula 288. [S.l.: s.n.], 11 min. 18 s. Disponível em: https://www.youtube.com/watch?v=zwvVd1NOzpg&list=PLqJK4Oyr5WSijcljG8mvLr-ojB1GD4OJZ. Acesso em: 22 jul. 2024.

PROGRAME SEU FUTURO. Curso de Programação C | Curso de Programação C | Operador OU | bit a bit. Como funciona a operação OR bit a bit? | aula 291. [S.l.: s.n.], 10 min. 06 s. Disponível em: https://www.youtube.com/watch?v=VL-HLPOvWZ0&list=PLqJK4Oyr5WSijCibG8mvLr-ojB1GD4OJZ&index=4. Acesso em: 22 jul. 2024.

FELIPE LOUZA (VIDEOAULAS). [C] 118. Arquivos binários: leitura e escrita com fread() e fwrite(), 16min 35s. Disponível em: https://www.youtube.com/watch?v=pEL7WOzCXE8. Acesso em 18 jul. 2024.