UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO
CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

BERNARDO SUNDERHUS
THIAGO BORGES

PROJETO DE PESQUISA:

TRABALHO DE ESTRUTURA DE DADOS

VITÓRIA
2013

BERNARDO SUNDERHUS

THIAGO BORGES

PROJETO DE PESQUISA:

TRABALHO DE ESTRUTURA DE DADOS

Trabalho científico apresentado ao curso de Ciência da Computação da Universidade Federal do Espírito Santo – UFES, como requisito parcial para aprovação na disciplina de Estrutura de Dados, ministrada pela professora Dra. Patrícia Dockhorn Costa.

VITÓRIA

2013

1 INTRODUÇÃO

1.1 PROBLEMA GERAL

O presente trabalho aborda a problemática da criação de dois programas que realizam a compactação e a descompactação de arquivos através da implementação do código de Huffman. O objetivo do programa de compactação é gerar um arquivo de saída, que seja mais leve que o arquivo original e que ao mesmo tempo represente tudo aquilo que está no arquivo original ao mesmo tempo em que contem um “mapa” de decodificação para o programa de descompactação. O programa de descompactação por sua vez, tem que ser capaz de ler o arquivo criado pelo compactador e produzir o arquivo que foi originalmente introduzido na compactação, para isso ele usa o “mapa” criado pelo compactador para conseguir ler o arquivo binário.

1.2 A CODIFICAÇÃO DE HUFFMAN

A codificação de Huffman é o método de compactação que se baseia na quantidade de um determinado símbolo em um arquivo. O objetivo da codificação é criar uma linguagem que sirva como tradução, tal que essa linguagem represente de forma mais eficiente os símbolos.

É partido do principio em que cada símbolo computacional é expresso na tabela ASCII com uma determinada ordem de 1 byte, como por exemplo: tem-se a letra 'A' = 01000001. O objetivo da compactação de Huffman é fazer com que exista uma “simplificação”, uma compactação, melhor dizendo, neste símbolo, para que então possamos escrevê-lo sem precisar ocupar um total de 1 byte.

Usando o principio de árvore-binária a codificação de Huffman cria uma árvore que representa todos os símbolos da tabela ASCII como folhas desta árvore, e também informa quantas vezes este mesmo símbolo aparece. Então a árvore é montada em ordem decrescente (do o que mais aparece para o que menos aparece).

1.3 O PROGRAMA DE COMPACTAÇÃO

1.3.1 OBJETIVO

O compactador, como dito anteriormente, tem como objetivo receber um arquivo de entrada, um arquvio texto por exemplo, e gerar um arquivo que represente a informação contida no arquivo recebido ao mesmo tempo em que é mais leve que o arquivo recebido.

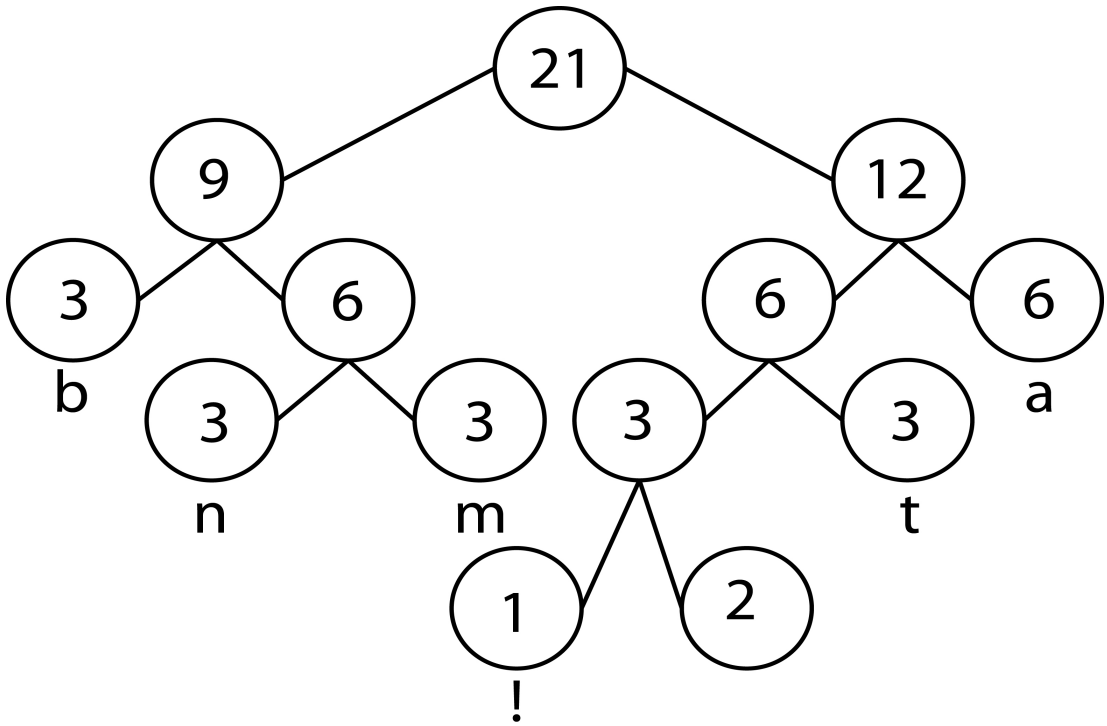
O segredo então é saber como escrever algo com menos “letras” e ainda assim representar a mesma coisa.

A partir da tabela ASCII sabemos que todo símbolo é na verdade uma sequencia de 1 byte que o representa e é utilizada para sua leitura. A ideia da codificação de Huffman é achar um jeito de escrever a informação em menos do que 1 byte. Melhor do que isso até, a ideia é escrever em o menor número possível de bits os símbolos que mais aparecem no arquivo! Pois então geraríamos uma compactação significativa.

Tendo em mãos a árvore que contem os símbolos do arquivo de entrada e a quantidade de vezes que cada símbolo aparece em cada folha, temos a ideia de quais serão os símbolos que precisarão ser mais compactados para gerar maior significância na redução do arquivo.

Tenha como exemplo um arquivo texto escrito: batman batman batman!
Este arquivo é reescrito pela tabela ASCII por: 01100010 01100001 01110100 01101101 01100001 01101110 00100000 01100010 01100001 01110100 01101101 01100001 01101110 00100000 01100010 01100001 01110100 01101101 01100001 01101110 00100001

Como podemos ver, são necessários 21 bytes (168 bits) para a escrita do arquivo texto. Agora, se reescrevermos o arquivo texto usando a ideia da codificação de Huffman teríamos uma árvore-binária deste jeito:



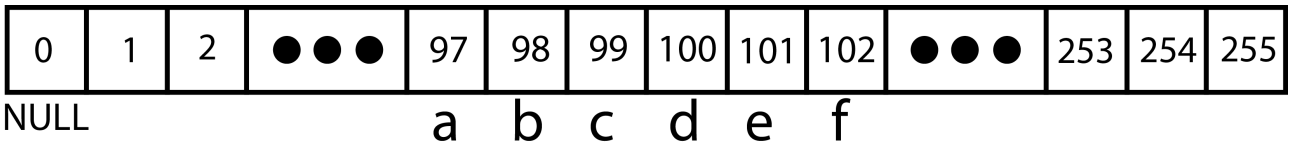
Então poderíamos adotar a ideia de que andar na árvore representaria o carácter, logo por questão de direção, direita é 1 e esquerda é 0. Poderíamos reescrever o texto como:
00 11 101 011 11 010 1001 00 11 101 011 11 010 1001 00 11 101 011 11 010 1000

Como podemos ver, são necessários 57 bits! Que poderiam ser expressos dentro de apenas 8 bytes para a escrita do arquivo texto! Claramente a codificação teve sucesso em sua compactação.

1.3.2 O PROCESSO DE COMPACTAÇÃO

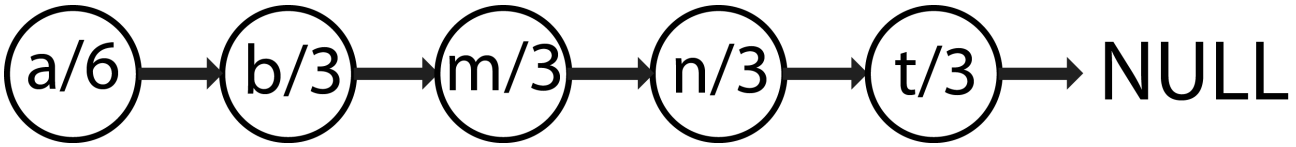
Inicialmente é aberto o arquivo de entrada e então é capturada o nome da extensão do arquivo (.txt, por exemplo) e então é criado um arquivo de saída. A extensão é escrita no arquivo de saída, já que essa terá que ser passada para o descompactador de alguma maneira.

Para poder atingir a ideia que tanto queremos de escrever o arquivo original em função de uma árvore-binária, tivemos a ideia inicial de ler o arquivo “capturando” símbolo a símbolo em um vetor de inteiros de 256 posições onde cada posição representaria um símbolo diferente da tabela ASCII. Assim, teríamos todos os símbolos utilizados para gerar o arquivo e o número de vezes que cada símbolo apareceu.



É necessária fazer com que esse vetor de inteiros seja transformado em árvore para montar-se a árvore de Huffman.

Cria-se então uma lista encadeada que contem dentro dela uma raiz da árvore, cada raiz contendo o símbolo representante e o número de repetições do símbolo.



Concatena-se as árvores em ordem dos números de vezes (as raízes com o maior número de repetição de símbolo em primeiro lugar) criando assim a árvore da codificação de Huffman.

Com a árvore em mãos o arquivo de entrada não será mais utilizado por um tempo então o fecharemos por enquanto.

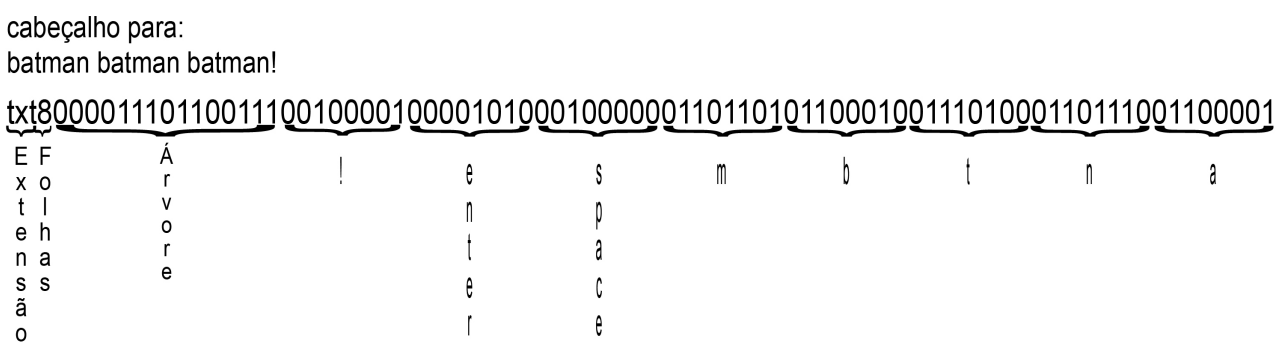
É feita então a contagem do número de folhas da árvore e logo em seguida esse número é escrito no cabeçalho do arquivo de saída. O número de folhas é usado como forma de parada de leitura da árvore na descompactação.

Agora é necessário passar ao arquivo de saída a árvore da codificação de Huffman. Adotamos então duas verdades na escrita e, futuramente, na leitura da árvore: a árvore terá que ser escrita da direita para esquerda, 0 representa uma raiz e 1 representa uma folha. Desse modo faremos a escrita da árvore de Huffman no arquivo de saída.

Mais um detalhe é acrescido na escrita da árvore no arquivo de saída: nós queremos economizar o maior número possível de bytes ao escrever a árvore, então não usaremos a tabela ASCII para escreve-la (0 em ASCII é 00110000 e 1 é 00110001), ao invés disso representaremos 0 e 1 com apenas 1 bit.

Apos imprimir com sucesso a árvore de Huffman no arquivo de saída, precisamos imprimir também a sequencia de símbolos em ordem Decrescente de quantidade no formato ASCII para que o descompactador saiba o que é o que.

Abaixo a imagem de exemplo de saída (a imagem está na ordem crescente).



Finalmente é aberto novamente o arquivo de entrada com intuito de escreve-lo no arquivo de saída, e então é feita a leitura símbolo a símbolo do arquivo, porem dessa vez não é usada a tabela ASCII para escrever o arquivo de saída, ao invés disso é utilizada a árvore de Huffman partindo do pretexto de que iremos “andar” na árvore em busca do símbolo procurado, sendo 1 direita e 0 esquerda teremos então a nova representação do símbolo pela árvore de Huffman.

Ao final deste processo, haverá um momento em que não haverá mais símbolos a serem escritos no bitmap e o bitmap ainda não estará completamente cheio. Chamamos esse espaço vazio do bitmap de lixo de informações. Esse lixo é devidamente tratado, porem tendo uma pequena quantidade de lixo que não há como tratar no compactador. Esse lixo então é passado pelo compactador e é acrescido no cabeçalho a informação de quantos são os bits de lixo. Assim finalizando a compactação.

1.4 O PROGRAMA DE DESCOMPACTAÇÃO

1.4.1 O OBJETIVO

O descompactador, como dito anteriormente, tem como objetivo receber um arquivo de entrada do tipo .comp que esteja devidamente compactado e a partir desse arquivo conseguir então gerar o arquivo original. O objetivo chave então é saber entender o “mapa” fornecido pelo compactador para conseguir então fazer o caminho inverso.

O descompactador tem que criar a partir das informações de cabeçalho do arquivo compactado a árvore usada para a compactação, e então usa-la para conseguir completar com sucesso a descompactação. Isso é feito lembrando-se que na compactação algumas verdades foram adotadas, e é em cima dessas verdades que iremos descompactar:

1. O arquivo compactado tem no inicio do cabeçalho o número de bits lixo.
2. A segunda informação é a extensão do arquivo que foi compactado.

3. A terceira é o número de folhas que é usado como chave de parada da árvore.
4. A quarta informação é a árvore escrita no formato 0 = raiz, 1 = folha e sabendo também que é feita da esquerda para direita (precisa-se de um caminho de referencia).
5. A última informação contida no cabeçalho do arquivo compactado são todos os símbolos usados no arquivo original em ordem decrescente, escrito no formato da tabela ASCII.

Com isso em mão podemos realizar a descompactação do arquivo.

1.4.2 O PROCESSO DE DESCOMPACTAÇÃO

Inicialmente é feita a abertura do arquivo compactado e logo é feita a leitura das primeiras informações do cabeçalho. Primeiramente o número de bits-lixo do arquivo é pego e guardado em uma variável, logo em seguida é feita a captura da extensão do arquivo original e então é gerado o novo arquivo de saída (o arquivo descompactado) e também é feita a leitura do número de folhas que é guardada em uma variável.

Agora, sabendo o número de folhas da árvore, é feito o processo de leitura da árvore sabendo que a árvore é escrita no formato 0 = raiz, 1 = folha e sabendo também que é feita da esquerda para direita (precisa-se de um caminho de referencia), logo o número de folhas é usado para saber quando parar (se tiver 10 folhas na árvore, a leitura acaba após o 10º digito 1).

Com isso nós conseguimos montar a estrutura da árvore, só que a árvore ainda está “oca”, sem informação alguma. O próximo passo é preencher a nova árvore com as informações que virão logo em seguida no cabeçalho, os símbolos em ordem decrescente de ocorrência.

Os símbolos estão escritos no formato dado pela tabela ASCII, mas eles estão escritos em bit, não em byte para evitar a geração de lixo no cabeçalho e então é feito um tratamento para descobrir os símbolos e eles são então preenchidos na nova árvore.
Com a nova árvore preenchida o próximo passo é ler os bits do arquivo de entrada para descobrir o caminho a ser percorrido na árvore e assim saber os simbolos a serem escritos no arquivo de saída.

Para isso é criado um vetor de caracteres que será o vetor de escrita. Assim então o símbolo encontrado pelo caminho é posto no vetor de escrita até o vetor de escrita ser preenchido completamente. Então é feita a impressão do vetor de escrita no arquivo de saída e então o vetor é reinicializado e o processo continua. No final do arquivo haverá algum momento em que o vetor de escrito não será totalmente preenchido e os símbolos então terminarão. Juntamente com os símbolos teremos o lixo passado pelo compactador, onde é feito o devido tratamento e remoção do lixo, finalizando assim a descompactação.

2 IMPLEMENTAÇÃO

2.1 COMPACTADOR

Como já demos a descrição geral de como foi implementado o compactador, aqui faremos alguns aprofundamentos e falaremos mais a respeito das estruturas de dados usadas.
A inicialização do compactador é basicamente a abertura do arquivo de saída e a leitura da entrada, só atendo ao pequeno fato de ter que se pegar a extensão do arquivo de entrada para passar para o descompactador. É usada a função **extensão** que tem como entrada uma string e o arquivo de saída. A função faz com que a string receba o nome completo do arquivo de entrada e ande na string até achar o símbolo '.', dai em diante a função retorna tudo que está a frente do ponto.

Temos também o inicialização do bitmap através da função **bitmapInit.** O bitmap é um vetor pré-definido que tem como função a escrita bit a bit no arquivo de saída.
Assim que a inicialização é concluída e o programa puder continuar é feita então a **leitura de todos os símbolos** pertencentes ao arquivo de entrada.

Para isso usamos um vetor de inteiro de 256 posições denotado de **caracteres**. A leitura é realizada pela função **contaCaracter** que tem como entrada o arquivo de entrada e o vetor de inteiros. Não há nenhuma pré-condição a se satisfazer para entrar nessa função e a pós-condição é que nada seja alterado. A saída da função é precisamente o vetor de inteiros preenchido conforme o arquivo de entrada.

Agora seguindo a descrição geral do compactador, **é criado uma lista de árvores** onde cada nó contem apena uma raiz informando um símbolo e a quantidade de vezes que ele apareceu. A função **inicializaLista** faz a inicialização devida da lista e a função **criaLista**

faz a criação da lista a partir do vetor de inteiros que foi preenchido com as informações do arquivo de entrada. A função **criaLista** tem como entrada a lista devidamente inicializada e o vetor de inteiros denominado caracteres. Tem como saida a lista de árvores.

Então agora que temos a lista de árvores nós temos o necessário para **criar a árvore da codificação de Huffman**. Para isso chamamos a função **Huffman** que em si, na verdade é a chamada em cadeia de duas funções internas que fazem o trabalho de arrumar a lista em ordem decrescente e de concatenar os dois primeiros nós da lista.

A função **Huffman** tem como entrada a lista de árvores e tem como saída uma nova lista, contendo somente uma árvore-binária condizente com a codificação de Huffman. Como pré-condição a lista deve estar devidamente inicializada e como pós-condição a nova lista deve condizer com a árvore-binária da codificação de Huffman.

Com a árvore formada podemos então chamar a função fclose para fechar o arquivo de entrada e então é chamada a função **contaLista** que tem como entrada a lista contendo a árvore (a partir de agora chamada de árvore de Huffman) e como saída o número de folhas que a árvore tem.

É feita a escrita do número de folhas diretamente no cabeçalho do arquivo de saída.
Agora que temos o número de folhas é feita a impressão da árvore de Huffman no arquivo de saída, chama-se então a função **listaToHuff** que tem como entrada o endereço do bitmap, a árvore de Huffman e o arquivo de saída. Não há saída na função de impressão.
Agora no arquivo de saida temos o número de folhas e a árvore de Huffman escrita no formato indicado na descrição geral. O próximo passo é escrever em ordem decrescente os símbolos da árvore. Para isso é chamada a função **convCharlista** que tem como entrada o endereço do bitmap, a árvore de Huffman e o arquivo de saída. Como é uma função de impressão, não há saída.

As principais estruturas de dados apresentadas no compactador até o momento são: o vetor de inteiros chamado de caracteres, a lista de árvores que contem em cada nó uma árvore e um ponteiro pra próxima árvore e a estrutura da árvore em si que contem um simbolo, um valor (número de vezes que o símbolo se repete no arquivo de entrada) e dois ponteiros para os filhos da árvore.

//diagrama da árvore, da lista de árvores, do vetor de inteiros.
Agora temos montado e escrito no arquivo de saída a: extensão, o numero de folhas, a árvore codificada em formato 0 = raiz, 1 = folha, e todos os caracteres escritos em ordem decrescente de repetição no formato da tabela ASCII.

O arquivo de entrada então é fechado e reaberto para começar a codificação do texto.
Para isso é criado um vetor de inteiro denominado **caminho** que tem como tamanho a altura da árvore + 1 e será o vetor que recebera o caminho necessário para chegar em um determinado símbolo. Na impressão nós imprimiremos caminho para a representação do símbolo.

Então pegamos cada caracter do arquivo e percorremos a árvore procurando o caminho, fazendo com que o vetor preencha as casas com 0 para a esquerda e 1 para a direita. Ao mesmo tempo pomos a informação de caminho dentro do bitmap até enche-lo. Com o bitmap cheio, é feita então a impressão dos caminhos e então o bitmap é reinicializado e a operação continua.

Finalmente após ter sido impresso todo o caminho de cada símbolo até a ultima chamada é feito a passagem do número de lixo em bits.

O PROBLEMA DO LIXO:

O arquivo é escrito no bitmap de forma binária, conforme desce os caminhos da árvore da codificação de Huffman e então cada vez que o bitmap é preenchido tem-se a escrita do conteúdo do bitmap no arquivo de saída por meio da função fwrite. O problema é que a função fwrite faz a escrita em bytes, enquanto que o bitmap é feito em bits.

Na maioria dos casos o número de bits utilizados para expressar o arquivo de entrada não é um múltiplo de 8, logo, sobraria lixo. Dessa forma é passado ao descompactador o número exato de lixo em bits que o arquivo terá e então no descompactador é feito o devido tratamento.

O calculo do lixo é bem simples: se pega o tamanho final do bitmap (bits) e divide-se por 8. Este número representa o número de bits q não caberiam, logo o número de lixo é o complemento para 8 deste número.

Usamos a função **rewind** para poder voltar para o inicio do arquivo de saída e escrevemos então no cabeçalho o número de bits que representa o lixo, finalizando assim o processo de compactação.

2.2 DESCOMPACTADOR

Como já demos a descrição geral de como foi implementado o descompactado, aqui faremos alguns aprofundamentos e falaremos mais a respeito das estruturas de dados usadas.

A inicialização do descompactador é dada pela abertura do arquivo compactado e é feita então já a leitura direta do cabeçalho. É feita a leitura do primeiro byte, que representa a quantidade de lixo no final do arquivo. Usamos para isso a função **fread** e então armazenamos esse número em uma variável inteira.

Logo em seguida é feita a coleção da extensão do arquivo original que está no cabeçalho. Para isso criamos um vetor de caracteres que recebe a função **extensaof** que tem como entrada o arquivo compactado e o vetor de caracteres que receberá a extensão. A função realiza um processo de leitura através da função fread em um loop que pega toda a extensão. É retornado então o vetor de caracteres contendo a extensão.

Logo em seguida é pego dos argumentos de entrada o nome que será posto para o arquivo de saída, isso é realizado pela função **nomeComp** que tem como entrada uma string, que recebera o nome, a extenção e o vetor do argumento de entrada.
Com o nome do arquivo e a extensão, é feita a abertura do arquivo de saída.
Pegamos então o número de folhas com a função **folha**, que é basicamente um fread.

Com o número de folhas em mão é chamada a função **recriaArv**, que como o nome indica faz o papel de recriar a árvore da codificação de Huffman. A função recriaArv Pega os bytes do arquivo de entrada e converte para um vetor onde cada casa representa um bit do caractere encontrado.

Verifica se o vetor está na ultima casa para precisar pegar o próximo símbolo e preencher novamente o vetor com os bits. Depois disso percorremos o vetor procurando o valor dos bits, caso seja 0 a função cria uma raiz e entra novamente nela com os valores alterados

(para a próxima casa do vetor de bits) caso seja 1 é criada a folha, o número do verificador de folhas é incrementado e retorna para a função anterior (se estiver na primeira chamada, volta para a main), retornando a arvore criada, o processo é feito até o verificador de folhas chegar ao mesmo valor do total de folhas da árvore original.

Lembrando que nesse momento temos em mão a nova árvore só que “oca”.
É feito então a inicialização de um bitmap do tamanho das folhas vezes o multiplicador 8, para poder caber o devido caminho total de todos os símbolos que serão pegos logo em seguida.

As funções **charToBitmap** e **putChar** fazem o trabalho de pegar o caminho do símbolo converte-lo para um símbolo e então adiciona-lo na árvore, respeitando toda a ordem de decrescência e de direção da esquerda para direita.

Agora temos a nova árvore montada e preenchida devidamente. Nesse instante do programa de descompactação é feito a chamada da função **caminhoLixo** que é simplesmente a função que retorna o número necessário para o tratamento adequado do lixo, que será explicado mais a frente.

Logo em seguida é feita a chamada da função **umadirecao**. Os próximos bits a serem lidos são os que vão percorrer a árvore e colocar os caracteres das folhas nos devidos espaços do arquivo de saída, restaurando o arquivo original. Lemos os bits do arquivo de entrada e percorremos ele verificando o caminho, caso o bit seja 0 equivale a ir para a esquerda, se for 1 equivale ir para a direita.

Fazemos esse processo até encontrarmos uma folha. O caractere dessa folha é preenchido no vetor que aloca o texto antes de ser escrito no arquivo de saída. Quando o vetor fica cheio, escrevemos o conteúdo dele no arquivo de saída e voltamos para a primeira casa do vetor, fazemos isso até o fim do arquivo de entrada.

A função uma direção, como dito a cima, faz a conversão de todo o texto compactado de volta para o seu estado original e o deposita em um vetor de caracteres que quando cheio é impresso no arquivo de saída e então reinicializado para continuar o processo. Porém no final do processamento desta função, haverá um vetor de caracteres com conteúdo e não haverá mais conteúdo a ser escrito.

Nessa hora a função termina e então é feita a impressão do ultimo vetor de caracteres, lembrando-se do tratamento de lixo que é necessário.

O TRATAMENTO DO LIXO:

A problemática é que temos nas ultimas casas do vetor de caracteres a ser impresso, uma quantidade calculável de lixo que foi gerada pelo programa de compactação. Temos pelo cabeçalho o número em bits do lixo, mas devido a tratamentos necessários no processo de descompactação esses bits de lixos foram transformados em bytes de lixo, sendo que esse número de byte não é algo fixo, ele na verdade depende da estrutura da árvore.

A linha de raciocínio parte do pressuposto que em algum momento esse lixo (em bits) será lido pela função umadirecao que irá converter esse lixo para algo legível partindo da ideia de que se temos um número determinado de bits-lixo então eu terei o mesmo número de passos na árvore, já que cada bit representa um caminho na árvore. Logo, se temos 6 bits podemos andar na árvore até 6 passos, para esquerda ou para direita. Por recursividade sabemos que o caminho da árvore é feita dando preferencia a esquerda. Logo o lixo irá sempre caminhar a esquerda atrás do menor caminho e quando ele encontra uma folha ele recomeça o mesmo processo graças a recursividade.

Assim a função umadirecao fará o erro de escrever no final do vetor de caracteres um número de bytes equivalente aos bits do lixo divido pelo caminho para chegar ao símbolo de maior ocorrência (o menor caminho). Isso nos dá uma formula: o número de bytes a ser eliminado no final do ultimo vetor de caracteres é: o número de bits-lixo dividido pelo caminho até o símbolo de maior ocorrência (caminholixo).

Armazenamos esse número em uma variável inteira chamada **menos** que será então subtraída do tamanho do vetor de caracteres para então eliminarmos da impressão os bytes que contem lixo.
Finalmente, é realizado o fechamento do arquivo, o programa e o arquivo descompactado estão então completos!

3 CONCLUSÃO

Mesmo com um bom conhecimento na linguagem C adquiridos nas matérias de Programação II e Estruturas de Dados I, tivemos muitos desafios para a montagem deste trabalho e ao finalizá-lo, entendemos melhor a linguagem e adquirimos mais conhecimento sobre C.

Inicialmente parecia uma tarefa bem simples. Apenas contar os caracteres, arrumar em pesos e concatenando para formar uma árvore binária, depois então ler os caminhos de cada símbolo e imprimir no arquivo de saída um cabeçalho e os caminhos. Depois percebemos que era bem mais complexo que isso, pois precisávamos escrever o arquivo bit a bit, e não byte a byte, como é tratado usualmente em C. Isso contando com o detalhe de que o tratamento de bit a bit gera lixo no arquivo que irá precisar de um tratamento.

Formulamos diversas ideias, pesquisamos fontes e descobrimos ferramentas que nos ajudaram a implementar as funções necessárias para o término do trabalho.

Apesar dos desafios encontrados, e os diversos problemas para conseguir compactar e descompactar perfeitamente pela codificação de Huffman, o trabalho foi uma grande experiência para botar os nossos conhecimentos adquiridos na matéria em prática e também para nos ensinar a buscar por nós mesmos o necessário para a implementação.

REFERÊNCIAS

STACKOVERFLOW. Convert char to int in c and c++. Disponível em: http://stackoverflow.com/questions/5029840/convert-char-to-int-in-c-and-c. Acessado em: 09/08/2013.

STACKOVERFLOW. How to convert int to char. Disponível em : http://stackoverflow.com/questions/1114741/how-to-convert-int-to-char-c. Acessado em: 09/08/2013.

TUTORIALS POINT. C function fwrite. Disponível em: http://www.tutorialspoint.com/c_standard_library/c_function_fwrite.htm. Acessado em: 26/08/2013.

STACKOVERFLOW. How to convert a char to binary. Disponível em: http://stackoverflow.com/questions/4892579/how-to-convert-a-char-to-binary. Acessado em: 31/08/2013.