TAREFA 4 – EAD

SINGULAR:

Nome: Enzo Tres Mediano

Matricula: 2211731

//ENUNCIADO:

Tarefa 4 – Compactação e descompactação de arquivos utilizando o algoritmo de Huffman
Descrição: O algoritmo de Huffman permite comprimir informação sem perda através da recodificação dos seus bytes. O algoritmo consiste de uma série de passos nos quais é possível construir a codificação (fluxo de bits comprimido) que substitui o fluxo de bits original. O princípio do algoritmo é de gerar códigos mais curtos para caracteres mais frequentes e códigos maiores para os caracteres menos frequentes. O trabalho consiste em implementar um compactador e um descompactador de arquivos texto baseados no algoritmo de Huffman. Este algoritmo utiliza várias das estruturas de dados estudadas no curso, tais como array, lista encadeada/lista de prioridades e árvore binária, vistos em aula. Para teste do algoritmo, pode ser usado um arquivo contendo a seguinte frase: “As estruturas de dados são fundamentais para a organização e manipulação eficiente de informações”. Imprima a saída obtida em cada passo do algoritmo.

//PseudoCódigo e detalhes das Funções

```
void count_letters(const char* input_string, long numBytes, int letter_counts[][2]){

        forloop ate numero numBytes{

                Ver posição dentro do array.(Vai incrementando com for loop)
                Ver primero posicao disponivel.
                Se char == char dentro do posicao fequencia++
                Se vazio, inserir character e frequencia como 1

        }

}
```

Para essa função tinha que criar um array que pudesse segurar o valor de um caracter e a quantidade de vezes que aquele caracter foi encontrado. Isso foi possível porque cada vez ele ia pro proximo posição ele via um caractere e via se já tinha esse caractere no sistema. Se tinha

ele coloca frequência++, e se nao ele vai no próximo posição disponível nesse array e coloca o valor do caractere e frequência = 1.

>>>

```
Node* createNode(char c, int freq) {

        Fazer malloc para um novo node
        Inserir valores necessários.

}
```

Função simples criada para malloc baseado no struct de node.

>>>

```
Node* constructHuffmanTree(int stringList[][2], int size){

        Malloc de um array de Nodes

        Criar quantidade de nodes baseado em size e colocar dentro do array allocado


        Forloop{

                Quick sort dos nodes

                Pegar primeira e segunda posições para criar novo node de +

                Inserir valores do newNode

                Decresser o tamanho do array

        }

        Fazer root do Huffman

        Free do array

        Return root

}
```

Nessa função usando o tamanho do array criado pela função “count_letters”, o programa fazia um malloc para um array de Nodes. Com isso o programa insira em cada posição um node com o valor do caracter e a frequência. Para ordenar foi necessário usar a qsort função de C que precisava de uma função que eu criei chamado (compareFunc). Com isso, o programa colocava os Nodes em ordem de menor a maior frequência. Ele pegava os dois menores e fazia um novo node de “+” e com a frequência das duas. E a direita e esquerda deste “+” sendo os dois menores valores. Isso repete até o forloop acabar determinado pelo size. No final o array de. Nodes (não os nodes próprios) são colocados numa free para remover espaço sendo usado.

>>>

```
void freeHufftree(Node* root){
```

free do Hufftree usando recursao do esquerda e direita do node

}

Função simples recursão que dava um free no tree de Huffman.

>>>

```
int compareFunc(const void* a, const void* b){
```

criar nodes para comprarar

retornar qual tem valor maior que o outro

}

A função que foi criada para comparar os valores de frequência no qsort.

>>>

```
void assignHuffCodes(Node* root, char* code, int depth, char** huffCodes){
```

Ver se root == NULL

Situacao onde recursao para (Quando achar um no folha)

Colocar o código do Huffcode e começar retornando

Codigo para determinar se vai esquerda o direito onde o código vai ser 0 se esquerda e 1 se direito

}

Essa função pegava nossa árvore de huffman e cada vez ele ia pela esquerda node ou direita node ele colocava um valor de 0 ou 1 (0 se esquerda e 1 se direita). No final quando ele achar uma folha ele coloca um '\0' para marcar o final do código. Depois ele faz um strdup do caractere para ser inserido dentro um array chamado huffCodes.

>>>

```
void printHuffCodes(Node* root, char* spaces){
```

Ver se root == NULL

Print de letra e a frequencia dele

Criar e alocar os "Espacos"

Casos do recursao (Esquerda Direita)

Free do “Espacos criados”

}

Funcao simples que usava recursão para printar cada node e os valores dentro do node. Cada vez que ele entrava num novo recursão ele cria um espaço de “ “ para visualmente mostrar níveis da árvore que está passando, daí ficava possível ver se todos os nodes estavam no lugar correto. É essa função que criou a imagem da árvore Huffman.

>>>

```
void encodeText(const char* text, int numBytes, char** huffCodes, FILE* output){
```

Write pro arquivo binario o quantidade de bytes necesarrio quando fazendo o decode

Preparacao dos buffer

Forloop ate o quantidade de bytes

Pegar cada numero do código de Huffman

Colocando os valores dentro do buffer

Quando buffer ta cheio, write o buffer dentro do arquivo binario

Ver situação aonde chegar o final e ver que tem mas espaço no final do buffer

Esquever o buffer metade completado dentro do arquivo binario

}

Primeira coisa ele faz e escrever a quantidade de bytes que tinha dentro do arquivo com tamanho de int. Depois ele cria um buffer que vai representar o byte e bufferPos que representa os bits dentro do byte. Dentro do primeiro for loop ele vai pegar um código de Huff dentro do array de huffCodes criado mais cerdo dentro do programa. Com isso ele vai ver o tamaho do código e fazer um outro loop baseado nesse tamanho. Por cada posição do tamanaho ele vai fazer um update no buffer. Ser por algum momento o byte fica cheio ele vai esquever o byte inteiro dentro do arquivo binário. E colocar valores do buffer e bufferPos para 0. Se o loop acaba, mas ainda temos um buffer meio completado, a função vai colocar os bits que estão dentro do byte para o possível início do byte para que quando lendo a programa não vai ver 0 aleatórios dentro do byte.

>>>

```
char* createArray(char* fileName, long* numBytes){

        Criar e abrir arquivo txt

        Achar tamanho do arquivo

        Colocar dados do arquivo dentro de um array de char* (string)

        Fechar arquivo

}
```

Primeira coisa a função faz e abrir o arquivo e achar o tamanho do arquivo e colocar dentro numBytes. Depois ele faz um malloc baseado no número de bytes e faz um fread para colocar os dados do arquivo dentro de um array de char. Depois ele retorna o array de char. Foi usado no início do programa para que o programa pode manipular os dados do arquivo.

>>>

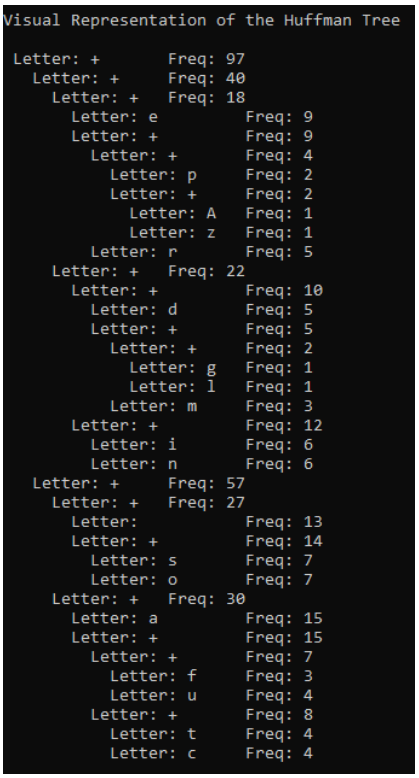
```
void decode(Node* root, int* numBytes){

        Abrir arquivo

        Achar tamanho do arquivo

        Achar quantidade de characters(achamos pq e primeio dado inserido quando fazendo o encoding do arquivo)

        Malloc e colocar dados do resto do arquivo dentro de um array

        Decode do array usando forloops vendo cada bit dentro de um byte

                Quando achar array, da um print do caracter na tela e fprintf no arquivo texto de output

}
```

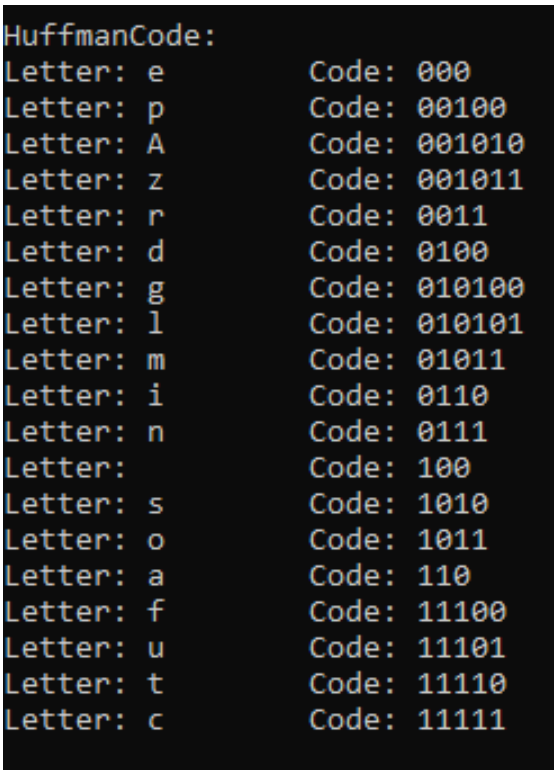
Essa função vai ser mais detalhado por causa dos estapas dele e a complexidade de cada etapa. A primeira coisa que a função faz é abrir o arquivo. O processo para abrir o arquivo binário fica bem parecido com o de “createArray”. O maior diferencia sendo que ele faz dois freads. A segunda para pegar o número de caracteres a gente tinha colocado no encoding, e o primeiro sendo para achar o tamanho do arquivo. Mas detalhes na segunda fread depois. A programa

aloca espaço baseado no tamanho do arquivo menos o tamanho de um int porque os primeiros poucos bytes do arquivo era aquele valor de quantidade de caracteres a programa tinha inserido durante o processo de encoding (lembrando que o tamanho de um int = 4 bytes então a processo aqui ta falando numBytes - 4). Depois disso ele faz um malloc baseado no número de bytes vezes o tamanho de char aonde char também fica o mesmo valor de um byte = 8 bits. Depois ele faz essa segunda fread do arquivo depois para pegar todos os valores que tinha depois da informação de número de caracteres. O arquivo fecha depois. Proximo etapa era para pegar os valores dentro desse vetor de Bytes (vBytes) que foi criado pelo malloc e inserido pelo fread. A programa cria myByte para mostrar a posição do Vetoriais vBytes, e um myBit para mostrar o posição do bit dentro do byte, com o forloop para cara myByte e um para cada bit dentro de cada byte. Dentro dessa segunda for loop de 8 ocorrências ele vai pegar o valor de um byte pela comparação do byte e 1000000 representado pelo hexadecimal de 0x80. (Pequeno detalhe de que depois de fazer isso ele faz um bitshift pela esquerda para preparar pelo próximo bit). O programa vai ver se o bit é igual a zero, se sim ele vai para a esquerda, se não ele vai para a direita (Esse fica por causa do algoritmo do Huffman). Quando ele acha uma folha no arvore de Huffman a programa pega o caractere que ta dentro da folha e dá um print na tela daquele caractere. Cada vez ele printa ele da um update no valor countChar. E o forloop para quando o countChar chega no valor de numChars que é aquele valor que o sistema tinha tirado no início do arquivo binário.

//Resultados:

- Tamanho do mensagem no arquivo txt inicial -> 97 bytes
- Foto do árvore de Huffman (esquerda) e Codigos Huffman (Direito)





- Tamanho do arquivo binaries (Tambem incluindo o int extra que tem no arquivo)
- Tamanho do arquivo txt final (incluindo int para ter numero de caracteres)-> 52

- Tamanho do mensagem dentro do arquivo binario (pos encoding) -> 48

//Conclusão

- Houve muitas dificuldades tentando criar este programa. A primeira é que não foram dadas muitas informações sobre os algoritmos de Huffman, então pesquisar o que a tarefa poderia estar pedindo resultou em muitas tentativas e erros. O enunciado não especifica o que nos pede para fazer com o algoritmo, então havia alguns aspectos da tarefa que precisavam ser esclarecidos para que os alunos pudessem entender melhor a tarefa. A grande necessidade de saber como manipular bits e métodos para contornar bytes foi necessária para a tarefa, mas quase 0 discussões foram feitas sobre esses tópicos. Embora os tópicos que pareciam úteis para a tarefa, como árvores binárias e certas alocações, fossem mais do que úteis para esta tarefa, sem esse conhecimento, seria muito mais difícil concluí-la. A tarefa parece ser como trabalhar com diferentes árvores e como trabalhar com formas de armazenar dados para testar o aluno se ele possui conhecimento prévio suficiente na forma como as informações são armazenadas por meio de bytes em bits. Consegui que tudo funcionasse no programa, então diria que a tarefa foi um sucesso durante todo o programa.