TRABALHO PRÁTICO 3

Compactação de Arquivos de Texto

Universidade Federal de Minas Gerais - Departamento de Ciência da Computação
Estrutura de Dados - 2023/01 - Turma TN
Belo Horizonte - MG - Brasil

Nome: José Eduardo Duarte Massucato **Matrícula:** 2022043620
josedmass@ufmg.br

1. Introdução

O trabalho proposto tem por objetivo lidar com a compressão de arquivos, de forma que não haja perda de informações no processo. Diante desse contexto, por decisão de projeto, a compactação do arquivo foi feita **por caractere**, utilizando do **algoritmo de Huffman**. Este algoritmo baseia-se na construção de uma árvore binária que codifica os caracteres presentes no arquivo de entrada, atribuindo códigos menores aos caracteres mais recorrentes, fazendo com que eles então ocupem menos espaço no arquivo de saída. Nesse sentido, o programa foi desenvolvido na linguagem C++, mas utilizando de vários métodos exclusivos da linguagem C, sobretudo na parte de manipulação de arquivos, como será explicitado posteriormente.

2. Método

2.1. Estrutura de Dados

A principal estrutura de dados utilizada no programa foi a **árvore binária**. Ela foi escolhida para armazenar os caracteres que aparecem no arquivo, de forma que os mais frequentes possuam códigos menores, isto é, estejam alocados em alturas menores na árvore. Uma vez que a árvore binária consegue hierarquizar facilmente os caracteres mais recorrentes dos demais, ela foi uma escolha adequada para a aplicação do programa.

2.2. Classes

2.2.1. `class` **`Celula`**

Armazena um caractere, a partir de `char chave`, e sua respectiva frequência no arquivo, a partir de `int valor`. Além disso, possui uma variável `Celula* prox`, que aponta para a próxima célula. Possui como **funções públicas**:

- `Celula()`: constrói uma célula vazia, isto é, com `prox` apontando para `NULL` e com `valor` igual a 0. Para a aplicação do algoritmo, não importava com qual valor `chave` seria inicializada.

2.2.2. `class` **Mapa**

Possui como **variáveis privadas**: `Celula* inicio`, que é a primeira célula do Mapa e `int tam`, que armazena o número de elementos contidos no Mapa. Este TAD tem por objetivo mapear os caracteres presentes no arquivo de entrada em relação às suas frequências. Possui como **funções públicas**:

- `Mapa()`: constrói um Mapa vazio, isto é, com `inicio` apontando para `NULL` e com `tam` igual a 0.
- `Mapa(const Mapa& other)`: constrói um Mapa a partir de outro, de forma que, ao final, o Mapa `*this` fique igual ao `other`.
- `int GetTamanho()` é uma função auxiliar que retorna o tamanho atual do Mapa, enquanto que `Celula* GetInicio()` é uma função auxiliar que retorna a célula inicial do Mapa. Essas funções foram criadas com o intuito de manter o encapsulamento do algoritmo.
- `void InserePosicao(char item, int pos)`: insere uma nova célula na posição `pos` do Mapa, com `chave` igual a `item` e `valor` igual a 1.
- `void IncrementaChave(char chave)`: incrementa o `valor` de um caractere armazenado no Mapa passado como parâmetro; caso ele ainda não esteja presente no Mapa, ele é então inserido.
- `void Limpa()`: limpa, iterativamente, todos os elementos contidos no Mapa, fazendo então com que `inicio` aponte para `NULL` e `tam` seja 0.
- `~Mapa()`: chama o método `Limpa()` para destruir o espaço alocado para um Mapa.

2.2.3. `class` **No**

Possui como **variáveis privadas**: `Celula item`, que armazena os caracteres nesse nó, `No* esq` e `No* dir`, que apontam, respectivamente, para os filhos da esquerda e da direita do nó. Possui como funções públicas:

- `No()`: constrói um nó vazio, isto é, com `esq` e `dir` apontando para `NULL`.
- `No(char ch, int freq)`: constrói um nó cuja célula é inicializada com a `chave ch` e com `valor freq`. Além disso, `esq` e `dir` apontam para `NULL`.

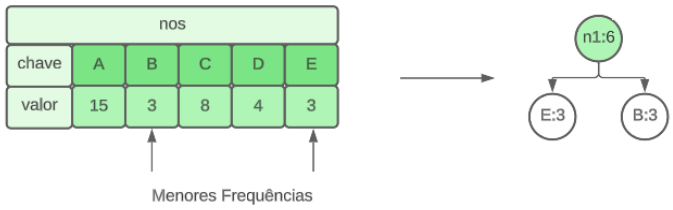
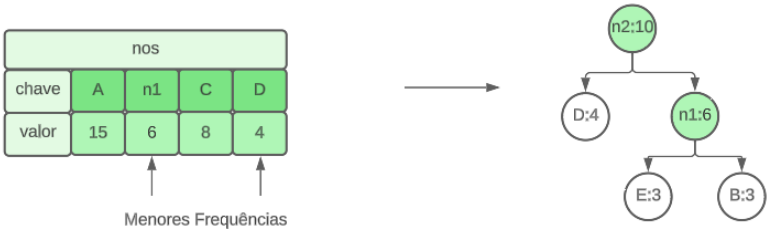
2.2.4. `class` **ArvoreHuffman**

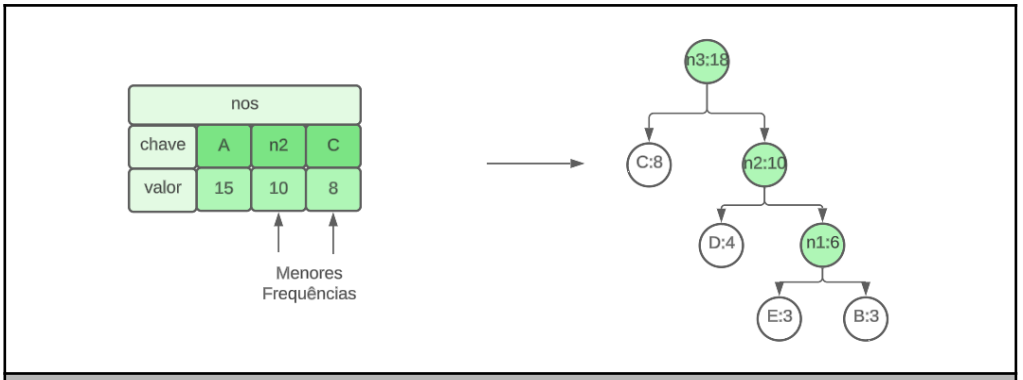
Trata-se de uma adaptação da árvore binária para a aplicação do algoritmo. Diante disso, possui como **variáveis privadas**: `No* raiz`, que aponta para a raíz da árvore, e `int tam`, que guarda o número de caracteres armazenados na árvore. Possui como **funções públicas**:

- `ArvoreHuffman()`: constrói uma árvore vazia, com `raiz` apontando para `NULL` e com `tam` igual a 0.
- `No* CopiaArvore(const No* r)`: função criada para copiar árvores, retornando o nó raiz da árvore final, igual ao `No* r`.
- `No* GetRaiz()`: função auxiliar que retorna a raiz da árvore. Essa função foi criada com o intuito de manter o encapsulamento do algoritmo.
- `void Serializa(const No* r, std::string& serie)`: transforma a árvore armazenada para o seu formato serializado, recursivamente.

  Observação: no formato serializado da árvore, as suas informações são representadas como uma *string*. Escrevemos um 0 quando trata-se de um nó interno e um 1 quando um nó folha. Além disso, quando escrevemos um 1, os próximos 8 bits serão a representação binária do caractere contido nesse nó-folha.

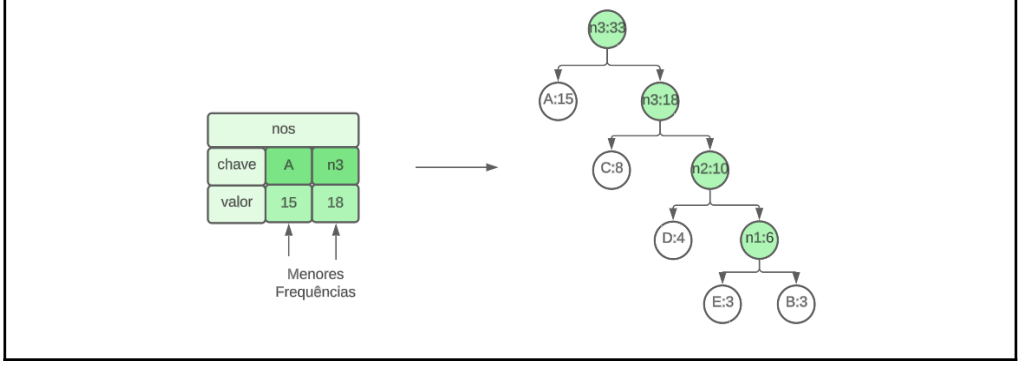
- `void Constroi(Mapa freq)`: constrói a árvore a partir de um Mapa das frequências dos caracteres do arquivo. Essa função basicamente executa o **código de Huffman** durante a fase de **compactação**. As imagens abaixo ilustram este processo:

PASSO 1
Montamos uma subárvore com os caracteres menos frequentes. A raíz dessa subárvore tem valor igual à soma das frequências dos filhos.

PASSO 2
Repetimos o processo do passo 1, mas agora temos também a subárvore n1 como um possível filho para a subárvore n2 a ser criada

PASSO 3
Repetimos o processo do passo 2



PASSO 4

Finalizamos a construção da árvore, de forma que o caractere mais frequente, no caso o ‘A’, esteja com menor altura em relação aos demais



- `void ConstroiSerie(const std::string& serie, No*& atual, int& i)`: reconstrói a árvore a partir de uma *string* que representa a serialização da mesma. Este processo é feito recursivamente, possuindo uma lógica inversa à do método `Serializa` descrito anteriormente.
- `bool BuscarCodigoHuffman(No* no, char ch, std::string codigo, std::string caminho)`: ao mesmo tempo que procura por um caractere na árvore, constrói o caminho percorrido para encontrá-lo. Se navegamos para a esquerda na árvore, temos que `caminho += “0”`, já se navegamos para a direita, então `caminho += “1”`.
- `std::string ObterCodigoHuffman(No* r, char caractere)`: retorna o código de um determinado caractere presente na árvore, a partir do método `BuscarCodigoHuffman`, explicitado anteriormente.
- `std::string ObterMensagem(std::string codigo)`: a partir de uma *string* que representa a codificação dos caracteres, reconstrói a mensagem original que fora codificada.
- `void LimpaRecursivo(No* r)`: limpa recursivamente os nós armazenados na árvore.
- `void Limpa()`: chama o método `LimpaRecursivo(raiz)` para limpar a árvore. Além disso, atribui `NULL` à `raiz` e 0 à `tam`.
- `~ArvoreHuffman()`: destrói a árvore, a partir do método `Limpa()`.

2.3. Funções

Em *TP/src/main.cpp*, os arquivos de entrada (`infile`) e saída (`outfile`) são criados. Se o usuário optar por compactar o arquivo (opção `-c`), então o arquivo de saída é aberto em modo de escrita binária (“`wb`”), garantindo que os bits sejam escritos corretamente. Por outro lado, se ele quiser descompactar (opção `-d`), então o arquivo de entrada é aberto em modo de leitura binária (“`rb`”), permitindo a leitura correta dos bits.

2.3.1. `std::string CharToBin(char caractere)`

Retorna a representação em binário de um caractere, em formato de string. Esse processo é feito iterativamente, percorrendo os bits do caractere.

2.3.2. `char BinToChar(std::string bin)`

Realiza o processo inverso da função anterior, ou seja, converte uma representação binária em formato de *string* de volta para seu caractere correspondente.

2.3.3. `Mapa frequencia(FILE* arquivo)`

Constrói o Mapa das frequências de cada caractere de `arquivo`, a partir do método `IncrementaChave`, explicitado na **seção 2.2.2.**

2.3.4. `void AjustarByte(std::string& codigo, int falta)`

Ao construir o código referente ao texto do arquivo, poderia ocorrer de ele não ter um comprimento traduzível para número de bytes, isto é, seu comprimento mod 8 ser diferente de 0. Assim, essa função garante que a *string* `codigo` tenha comprimento em bytes, adicionando 1’s e 0’s ao final da *string*. Essa função é utilizada na fase da **compactação**.

2.3.5. `int NumAdicionado(std::string codigo)`

Esta função é utilizada na fase da **descompactação**. Ela extrai quantos 1’s e 0’s foram adicionados no final do código na fase da compactação para que, posteriormente, essa quantidade fosse subtraída de `codigo`, a fim de obter o código original.

2.3.6. `void compactar(FILE* entrada, Mapa freq, FILE* saida)`

Esta função realiza a compactação do arquivo de entrada e imprime o resultado no arquivo de saída. Primeiramente, é feita a serialização da árvore e em seguida são registrados quantos 1’s e 0’s foram adicionados. Após isso, é inserida uma linha vazia. Então, o texto compactado é impresso, também acompanhado pela quantidade de 1’s e 0’s adicionados.

Observação: a escrita dos textos compactados é feita em blocos de bytes. Isso significa que, após obter a *string* com a representação dos códigos dos caracteres do texto original, basta escrevê-la no arquivo de saída em blocos de 8 bits, utilizando as funções descritas nas **seções 2.3.1** e **2.3.2**.

2.3.7. `void descompactar(FILE* entrada, FILE* saida)`

Realiza a descompactação do arquivo de entrada e imprime o resultado no arquivo de saída. Primeiramente, a função extrai a árvore serializada presente no arquivo de entrada e subtrai dela a quantidade de 1's e 0's adicionados durante a compactação. Com base nisso, a função reconstrói a árvore de Huffman e, utilizando-a, decodifica o texto compactado, recuperando o texto original byte a byte.

3. Análise de Complexidade

A análise de complexidade foi realizada com base nos dois comandos básicos que o usuário pode digitar: `-c` para compactação e `-d` para descompactação.

3.1. Compactação

No caso em que o usuário opta por compactar o arquivo, o programa primeiramente lê o arquivo de entrada para mapear as frequências de cada caractere, utilizando a função descrita na **seção 2.3.3**. Nessa etapa, há um *loop while* que percorre todo o arquivo, caractere por caractere, chamando a função `IncrementaChave`, de complexidade linear. Em seguida, a função compactar é chamada, a qual por sua vez chama várias outras funções, todas descritas anteriormente. Abaixo estão indicadas as complexidades de cada uma dessas funções, bem como a **complexidade de tempo** total final:

$$T(n) = frequencia() + compactar()$$
$$T(n) = frequencia() + Constroi() + serializa() + O(n) + n \times ObterCodigo()$$
$$T(n) = n \times O(n) + O(n) + O(n) + O(n) + n \times O(n)$$
$$T(n) = O(n^2) + O(n) + O(n^2)$$
$$T(n) = O(max(n^2, n)) = O(n^2)$$

Com relação à **complexidade de espaço**, temos que ela é $O(n)$, onde n é o número de caracteres distintos no arquivo de entrada.

3.2. Descompactação

No caso em que o usuário opta por descompactar o arquivo de entrada, que foi compactado por este programa, temos a chamada da função `descompactar`, a qual por sua vez chama várias outras funções, todas descritas anteriormente. Abaixo estão indicadas as complexidades de cada uma dessas funções, bem como a **complexidade de tempo** total final:

$$T(n) = descompactar()$$
$$T(n) = O(n) + ConstroiSerie() + ObterMensagem()$$
$$T(n) = O(n) + O(log\, n) + O(n)$$
$$T(n) = O(max(n, log\, n)) = O(n)$$

Com relação à **complexidade de espaço**, temos que ela é também $O(n)$, onde n é o número de caracteres que foram passados na representação serializada da árvore no arquivo de entrada.

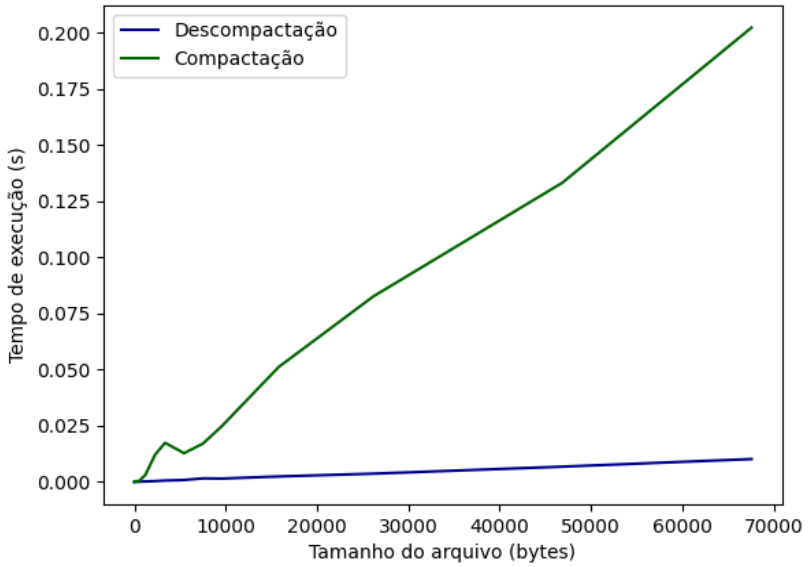
4. Estratégias de Robustez

Na aplicação deste programa, os principais problemas que podem surgir estão relacionados à manipulação dos arquivos. Especificamente, caso ocorram erros ao abrir os arquivos, mensagens de erro serão adequadamente exibidas na saída padrão para informar o usuário sobre o ocorrido.

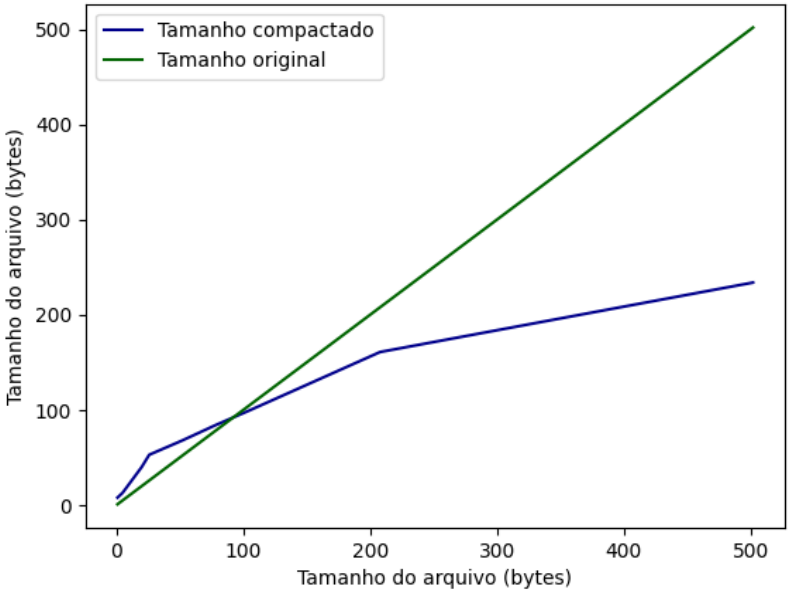
Além disso, é importante destacar que o programa possui requisitos mínimos para execução correta. Se o usuário não fornecer os nomes dos arquivos de entrada e saída, juntamente com a opção desejada (`-c` para compactar ou `-d` para descompactar) uma mensagem de erro será exibida na saída padrão. Essa abordagem visa garantir que o programa seja utilizado corretamente, evitando resultados indesejados.

5. Análise Experimental

A análise experimental do programa foi feita com base na quantidade de caracteres presentes no arquivo de entrada.



Neste gráfico, o tempo de execução é representado em segundos para diferentes tamanhos de arquivos, variando de 0 a 70000 bytes. É evidente que o algoritmo de compactação tem um tempo de execução significativamente maior do que o de descompactação. Isso ocorre porque, durante o processo de compactação, o arquivo é lido duas vezes, ambas com custo quadrático de tempo, conforme mencionado anteriormente. Por outro lado, na descompactação, além de o custo de tempo ser linear, também temos que o arquivo de entrada é compactado, ou seja, tem tamanho menor, o que justifica seu tempo menor.



Neste outro gráfico, é representado o tamanho do arquivo compactado resultante em comparação com o tamanho do arquivo original, para diferentes tamanhos de entrada. Nele, percebe-se que, para entradas de 0 a 100 bytes, o arquivo compactado apresentou tamanhos maiores do que o original. Isso se deve ao fato de que o arquivo compactado inclui a árvore serializada e o texto compactado, o que, em conjunto, resulta em um tamanho maior do que o texto original, especialmente quando o texto original é muito pequeno. Abaixo está representado um exemplo de compactação para um arquivo de 5 bytes.

ARQUIVO ORIGINAL	ARQUIVO COMPACTADO
`teste`	`YW;¥3` `Ö8`
5 bytes	9 bytes

No enunciado do Trabalho Prático, foi proposto imaginar que o algoritmo era desenvolvido para reduzir o tamanho de arquivos de jornal, que tendem a ser grandes (maiores que 1KB), então não considerei esse fato um problema para a aplicação. Além disso, para arquivos de entrada grandes, observei que o arquivo compactado tinha cerca de 50% do tamanho do arquivo original. Abaixo está representado um exemplo de compactação para um arquivo de 3KB.

ARQUIVO ORIGINAL	ARQUIVO COMPACTADO

testeteste trabalhopratico3 Estrutura de Dados testeteste trabalhopratico3 Estrutura de Dados testeteste trabalhopratico3 Estrutura de Dados testeteste trabalhopratico3 Estrutura de Dados testeteste trabalhopratico3 Estrutura de Dados testeteste trabalhopratico3 Estrutura de Dados testeteste trabalhopratico3 Estrutura de Dados testeteste trabalhopratico3 Estrutura de Dados testeteste trabalhopratico3 Estrutura de Dados testeteste trabalhopratico3 Estrutura de Dados testeteste trabalhopratico3 Estrutura de Dados testeteste trabalhopratico3 Estrutura de Dados ... 3000 bytes	▯ªgE,VÅ¢à²-›¼ºŠ$*És$▯+¥3 êúêúÏº¦sìik‹▯ C†Ì ªÊ%^éêúêúÏº¦sìik‹▯ ‹▯ C†Ì ªÊ%^éêúêúÏº¦sìik‹▯ C†Ì ªÊ%^éêúé ... 1496 bytes

6. Conclusões

Durante o desenvolvimento deste trabalho, foram criadas várias classes e funções para lidar com a compactação e descompactação de arquivos. A principal estrutura de dados utilizada e adaptada ao problema foi a árvore binária. Isso permitiu o aprendizado sobretudo de alocação dinâmica e recursão, fundamentos essenciais para o funcionamento da árvore.

No entanto, surgiu uma dificuldade ao lidar com a manipulação dos arquivos, bem como a escrita de bytes nele. Inicialmente, tentei utilizar as funções do C++ para essa finalidade, mas acabei enfrentando diversos erros, pois não compreendia completamente o funcionamento dessas funções. Como resultado, optei por trabalhar com a lógica de arquivos em C, mesmo que fosse mais complexa em alguns momentos. Essa abordagem permitiu que eu manipulasse os arquivos com mais detalhes e garantisse que o resultado fosse exatamente o esperado. Com isso, esses obstáculos foram superados.

Além disso, ao longo do trabalho, desenvolvi habilidades fundamentais da computação, como a análise de complexidade de algoritmos e a implementação de estratégias de robustez. Por outro lado, a manipulação de bits foi um fator essencial para o desenvolvimento do projeto, uma vez que ela foi muito utilizada para codificar e decodificar os caracteres, compactando-os. Essas experiências foram de grande importância para a minha formação acadêmica, ao passo que me proporcionaram uma curva de aprendizado muito boa, além de estarem adequadamente alinhadas com o que estávamos aprendendo na disciplina de Estrutura de Dados.

7. Bibliografia

[1] CORMEN, Thomas H.; LEISERSON, Charles E.; RIVEST, Ronald L.; STEIN, Clifford. Introduction to Algorithms. 3rd ed. Cambridge: MIT Press, 2009. Seção 16.3: Huffman Codes;

[2] BACKES, André. Linguagem C: Completa e Descomplicada. Rio de Janeiro:Elsevier. Capítulo 12: Arquivos.

8. Instruções para compilação e execução

8.1. Abra o terminal.

8.2. Entre na pasta `TP`

- **Comando:** `cd / <caminho para a pasta> /TP`

8.3. Adicione o(s) arquivo(s) de entrada em `./TP`

8.4. Compile e execute o programa a partir do arquivo Makefile presente em `./TP`

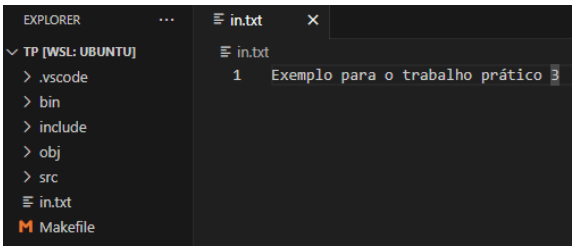
- **Comando:** `make run in=<entrada.txt> out=<saida.txt> opt=<opção>`
- *Observação 1:* este comando compila **e** executa o programa;
- *Observação 2:* o arquivo de entrada **deve** estar na pasta `TP`
- *Observação 3:* a cada nova execução, `<entrada.txt>` e `<saida.txt>` devem ser substituídos pelo nome do arquivo que se deseja operar, com extensão `.txt`.
- *Observação 4:* a cada nova execução, `<opção>` deve ser substituído por `-c`, para compactar, ou por `-d`, para descompactar.
- **Atenção:** na descompactação, o arquivo de entrada **deve** ter sido compactado pelo próprio programa, senão comportamentos indesejados podem ocorrer.

8.5. Como especificado no enunciado do TP, o resultado da execução do programa é imprimido em `<saida.txt>`.

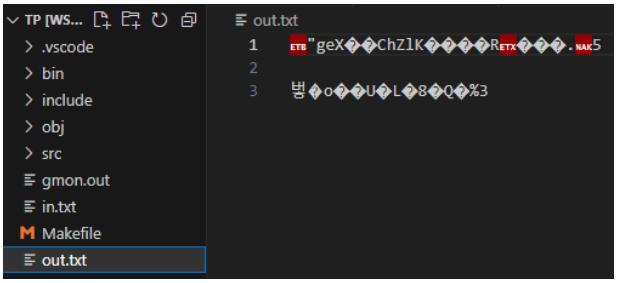
8.6. Ao final de todas as execuções, utilizar o Makefile para limpar os arquivos de objetos gerados

- **Comando:** `make clean`

Exemplo de compactação para `./TP/in.txt`




```
root@DESKTOP-O1SI1LT:~/home/josedmass/ED/TP3/TP# make run in=in.txt out=out.txt opt=-c
g++ -Wall -c -Iinclude -o obj/main.o src/main.cpp
g++ -Wall -c -Iinclude -o obj/Mapa.o src/Mapa.cpp
g++ -Wall -c -Iinclude -o obj/ArvoreHuffman.o src/ArvoreHuffman.cpp
g++ -Wall -c -Iinclude -o obj/ConversaoBinario.o src/ConversaoBinario.cpp
g++ -Wall -c -Iinclude -o obj/Compactacao.o src/Compactacao.cpp
g++ -Wall -c -Iinclude -o obj/Descompactacao.o src/Descompactacao.cpp
g++ -pg -o bin/main obj/main.o obj/Mapa.o obj/ArvoreHuffman.o obj/ConversaoBinario.o obj/Compactacao.o obj/Descompactacao.o -lm
./bin/main in.txt out.txt -c
```




```
root@DESKTOP-O1SI1LT:~/home/josedmass/ED/TP3/TP# make clean
rm -f bin/main obj/main.o obj/Mapa.o obj/ArvoreHuffman.o obj/ConversaoBinario.o obj/Compactacao.o obj/Descompactacao.o gmon.out
```