Trabalho Prático 03 - Sistema de Compactação

Aluno: Raphael Henrique Braga Leivas
Matrícula: 2020028101

Universidade Federal de Minas Gerais - UFMG
Belo Horizonte

1 INTRODUÇÃO

O Trabalho Prático 03 consiste essencialmente no desenvolvimento de um sistema de compactação capaz de comprimir e descomprimir arquivos de texto passados na entrada. Para isso, é usado o algoritmo de Huffman, que utiliza estrutura de dados relacionando um caractere com a quantidade de vezes que ele ocorre no arquivo de entrada, isto é, sua frequência. Assim, é possível elaborar um modo ótimo de representar cada caractere como uma cadeia binária (CORMEN, Thomas H., 2012).

2 MÉTODO

O software foi desenvolvido em linguagem C++ em ambiente Windows, sendo compilado em WSL Ubuntu 22.04 LTS com g++. O processo de compilação dos módulos é gerido por Makefile, assim mantendo a estrutura da pasta especificada em atividades da disciplina. Para verificar o acesso às chamadas de funções foi usado o gprof, ao passo que para depurar a aplicação foi usado o gdb.

Será usado o algoritmo de Huffman com compactação a nível de caractere, devido ao fato de ele ser mais simples de implementar.

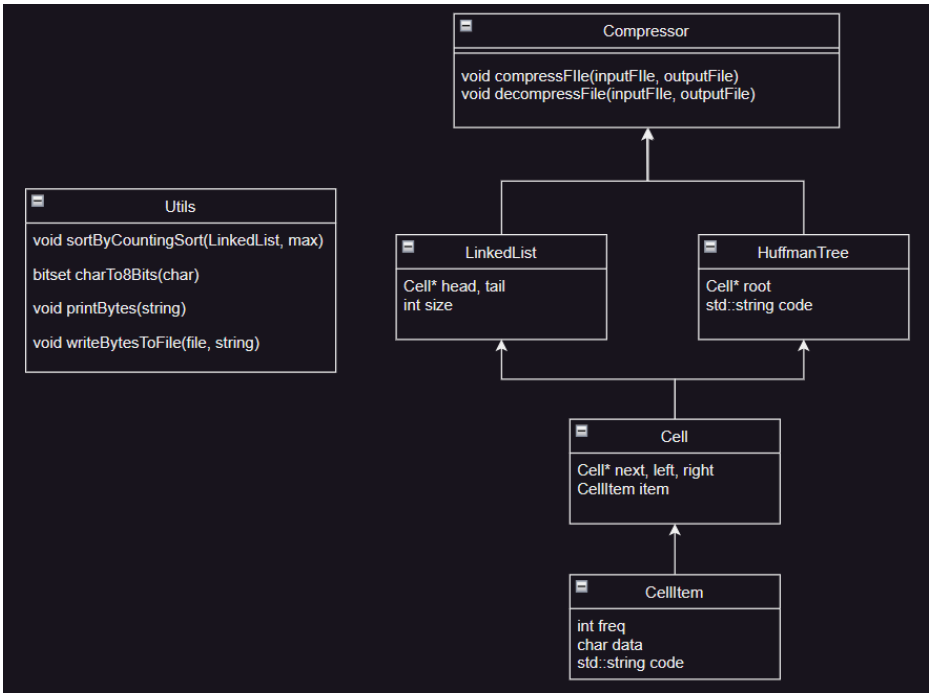
2.1 ESTRUTURAS DE DADOS

A Figura 2.1 mostra as principais estruturas de dados e classes usadas que efetuam os processos de compactação e descompactação.

Na Figura 2.1, temos uma hierarquia de classes em camadas. A função principal main chama a classe Compressor para compactar e descompactar um arquivo. Note que, para a main, não importa como as funções são implementadas ou qual algoritmo de compactação é usado, basta informar os endereços completos dos arquivos de entrada e saída que a compactação ou descompactação é feita.

A classe Compressor, por sua vez, precisa de duas classes auxiliares LinkedList e HuffmanTree para implementar os algoritmos de Huffman para compressão e descompressão.

Figura 2.1. Principais estruturas de dados e classes usadas no software.



Fonte: elaboração própria.

A classe LinkedList é essencialmente uma lista encadeada simples, com diversas funções auxiliares implementadas que são usadas durante o algoritmo (serão descritas em mais detalhes na seção 2.2). Cada elemento da lista é uma célula, classe Cell, que possui o ponteiro next usado pela lista encadeada para indicar o próximo elemento. É sobre a célula que as operações de inserção e remoção da lista encadeada são executadas. Cada célula possui um item de célula, classe CellItem (lê-se Cell Item), que contém os seguintes dados fundamentais da aplicação:

- char data: caractere UTF-8 qualquer, com código ASCII;
- int freq: inteiro que salva a frequência do caractere salvo em data;
- std::string code: código binário que representa esse caractere, determinado pelo algoritmo de Huffman. Aqui é salvo como um string, por exemplo, “1101”.

Também temos a classe HuffmanTree, que é essencialmente uma árvore binária cujos nós são do tipo Cell. É sobre essa estrutura que os algoritmos recursivos de Huffman são aplicados. Note que a classe Cell possui também ponteiros left e right além do next, de modo que ela pode ser usada tanto pela lista encadeada quanto pela árvore.

Por fim, temos a classe Utils com funções auxiliares diversas que podem ser chamadas por qualquer uma das estruturas descritas acima. Note, na Figura 2.1, que a classe Utils se localiza fora da hierarquia de classes.

2.2 IMPLEMENTAÇÃO DAS ESTRUTURAS

Nesta seção é descrita a implementação das duas principais funções do sistema: Compressor.compressFile para compactação e Compressor.decompressFile para descompactação.

2.2.1 Compactação: compressFile

A sequência de passos abaixo mostra o funcionamento do método compressFile da classe Compressor.

1. Abre o arquivo de entrada e começa a percorrer cada caractere contido nele.
2. Insere o caractere na lista encadeada. Se ele já existe, só aumenta a sua frequência em uma unidade.
3. Quando acabar de percorrer o arquivo, ordena a lista encadeada via CountingSort a partir das frequências de forma decrescente. Usamos CountingSort pois é um algoritmo de ordenação linear $O(n)$ e, durante a própria ação de percorrer o arquivo de entrada, já podemos determinar o valor máximo da frequência, e assim é possível usar o CountingSort.
4. Pega os dois últimos elementos da lista (que após a ordenação são os caracteres de menor frequência), soma a frequência deles. Salva um na esquerda e outro na direita (ponteiros left e right) de uma nova célula. Insere essa nova célula na lista de forma a manter a ordenação decrescente de frequências.
5. Repete o item 4 até a lista encadeada ter apenas um elemento. Esse elemento é a raiz da árvore de Huffman, e nesse momento efetivamente transformamos a lista encadeada em uma árvore de Huffman.
6. Percorre a árvore de Huffman a partir da raiz até chegar nas folhas (as folhas é onde estão os caracteres). Toda vez que ir para a esquerda adiciona um 0, toda vez que ir para a direita adiciona um 1, e ao chegar na folha o código desse caracter será os zeros e ums adicionados durante o percurso. Isso é feito com o método recursivo HuffmanTree.assignHuffmanCodes().
7. Feito isso, percorre o arquivo de entrada mais uma vez. Para cada caracter, salva em um buffer (string) o código correspondente a esse caractere determinado no item 6. Para achar o código desse caractere, busca recursivamente na árvore até achar a célula com esse caractere, através do método HuffmanTree.findCellByChar().
8. Depois, converte a Árvore Binária em uma representação binária com o método recursivo HuffmanTree.codifyTree(), ou seja, gera uma sequência de bits que representa a árvore atual. Isso é feito de modo semelhante ao item 6.
9. Nesse momento temos duas sequências binárias: uma representa os caracteres do arquivo de entrada convertidos para os códigos de Huffman, e outra representa a própria Árvore de Huffman em uma sequência binária. Escrevemos essas duas sequências no arquivo de saída, byte a byte. No início do arquivo reservamos 32 bits indicando o tamanho em bits da primeira sequência, seguido de mais 32 bits indicando o tamanho em bits da árvore codificada. Essa informação é fundamental para, na hora de descompactar, ser possível saber onde uma sequência termina e outra começa.
10. Fecha os arquivos de saída e entrada.

Note que o tamanho máximo do arquivo de entrada codificado é de 2^{32} bits, ou 512 Megabytes. Essa restrição existe pois reservamos 32 bits no início do arquivo comprimido

para o tamanho da sequência codificada dos dados. Acredito que esse tamanho de arquivos de entrada seja suficiente para os testes feitos durante a avaliação do TP.

2.2.2 Descompactação: decompressFile

A sequência de passos abaixo mostra o funcionamento do método decompressFile da classe Compressor.

1. Abre o arquivo comprimido e começa a percorrer byte a byte.
2. Primeiro lê o tamanho em bits dos dados N (primeiros 32 bits do arquivo) e depois o tamanho em bits da árvore codificada M (próximos 32 bits do arquivo).
3. Feito isso, lê todos os N bits de dados e depois os M bits da árvore codificada. Para de ler o arquivo quando terminar de ler o M-ésimo bit da árvore codificada.
4. Recupera a Árvore de Huffman com o método HuffmanTree.decodifyTree(), que faz o caminho inverso do método HuffmanTree.codifyTree() usado na compactação.
5. Com a Árvore de Huffman remontada, associa a cada caractere na árvore o código de Huffman com o método HuffmanTree.assignHuffmanCodes().
6. Com os códigos de cada caractere, percorre os bits dos dados codificados. A partir da raiz, quando acha um bit 0 vai para a esquerda, quando acha um bit 1 vai para a direita. Ao chegar em uma folha da árvore, recupera o caractere e salva no buffer de saída.
7. Finalmente, escreve o buffer de saída no arquivo de saída. Fecha os arquivos de entrada e saída.

Muitas manipulações de leitura e escritas que envolvem bits e bytes foram feitas usando a biblioteca std::bitset, especialmente para conversões de inteiros e caracteres em representações de 8 e 32 bits.

As descrições acima são apenas uma ideia geral do que o software está fazendo. Muitos métodos não foram elaborados em detalhes aqui por uma questão de brevidade e para evitar uma descrição literal do código-fonte.

3 ANÁLISE DE COMPLEXIDADE

3.1 COMPRESSÃO

Vamos quebrar as etapas da compressão de modo semelhante ao feito na seção 2.2 e identificar a complexidade assintótica de cada uma delas. Seja n o tamanho do arquivo de entrada.

1. Percorrer todos os n caracteres da entrada na leitura e inserir na lista encadeada: $O(n) + O(1) = O(n)$. Note que inserção e remoção em lista encadeada é $O(1)$;
2. Ordenar a lista encadeada com CountingSort: $O(n + k)$, sendo k o valor da frequência máxima de um caractere. Como o valor máximo da frequência de um caractere é $k = n$, temos $O(n + n) = O(2n) = O(n)$;

3. Montar a árvore de Huffman a partir da lista encadeada, pegando os dois últimos elementos da lista e inserindo até sobrar somente um: $O(n)$
4. Atribuir um código a todos os caracteres da árvore: $O(n)$
5. Percorrer novamente os n elementos da entrada, e para cada elemento procurar recursivamente a célula que tem esse caractere na árvore, extraindo assim seu código: $O(n) \cdot O(lg\, n) = O(n\, lg\, n)$.
6. Codificar a árvore em sequência binária: $O(n)$
7. Escrever de volta no arquivo de saída: $O(n)$

Complexidade resultante: $6 \cdot O(n) + O(n\, lg\, n) = O(n\, lg\, n)$, onde $lg\, n = log_2(n)$

3.2 DESCOMPRESSÃO

Usamos a mesma estratégia usada para a análise de complexidade de compressão:

1. Ler os n bytes do arquivo de entrada e salvar nas variáveis: $O(n)$
2. Decodificar a árvore a partir da sequência binária: $O(n)$.
3. Atribuir um código a todos os caracteres da árvore: $O(n)$
4. Decodificar o texto a partir dos códigos de Huffman atribuídos no item 3: $O(n)$

Complexidade resultante: $4 \cdot O(n) = O(n)$

4 ESTRATÉGIAS DE ROBUSTEZ

Durante a compactação, se a lista encadeada estiver vazia temos que o arquivo de entrada também era vazio, disparando uma exceção no método compressFile.

Durante a descompressão, por outro lado, a proteção é tanto de arquivos vazios quanto de falhas na leitura dos bytes. Se o número de bits lidos na sequência binária dos dados for diferente do tamanho dela informada nos primeiros 32 bits do arquivo comprimido, é disparado uma exceção. Similarmente, se o número de bits lidos na sequência binária da árvore codificada for diferente do tamanho dela informada nos próximos 32 bits do arquivo (logo depois dos 32 bits informando o tamanho dos dados), também é disparado uma exceção.

As demais estratégias de robustez consistem essencialmente em práticas de programação defensiva locais, tais como verificação de ponteiros NULL antes de acessá-los, verificação de bad_argument dentro do escopo de funções, etc.

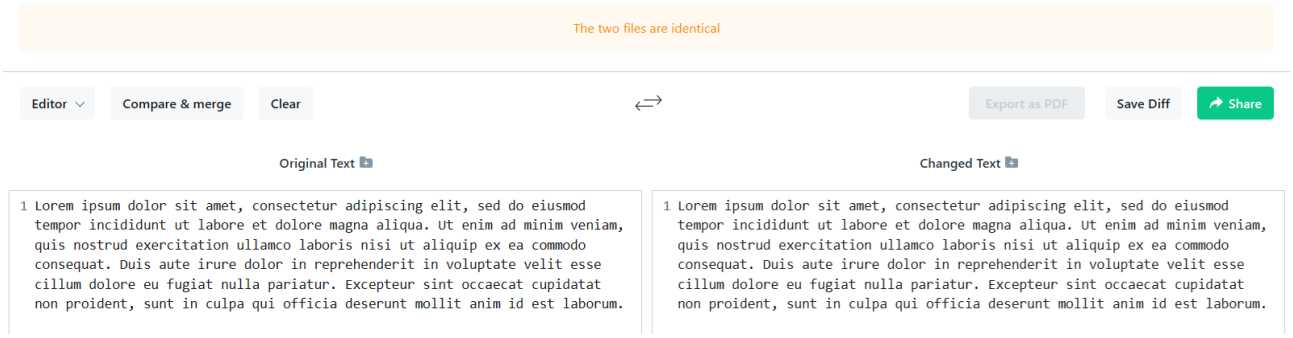
5 ANÁLISE EXPERIMENTAL

Para testar o software, usamos vários arquivos de entrada para o sistema compactar e depois descompactar, desde arquivos com poucas linhas até grandes arquivos com centenas de Kilobytes. Quando o sistema compacta um arquivo e depois

descompacta, é esperado que o arquivo descompactado seja idêntico ao arquivo original, de modo que não tenhamos perdas de dados.

Usando o site gerador de textos do tipo Lorem Ipsum https://loremipsum.io/ , e o site comparador de textos online https://www.diffchecker.com/, podemos verificar se os arquivos de entrada e saída são idênticos, como mostra a Figura 5.1.

Figura 5.1. Teste feito para verificar se os arquivos de entrada e saída gerados pelo software são idênticos.

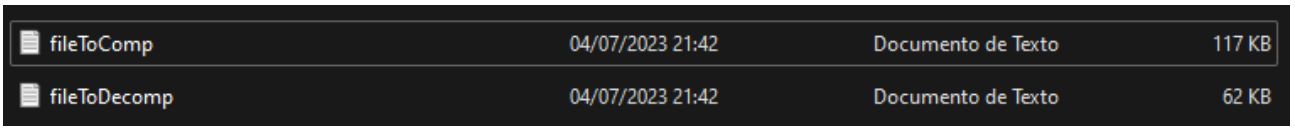


Fonte: elaboração própria.

Para verificar a taxa de compressão, usamos um teste com um grande arquivo de entrada de 117 KB. Como mostra a Figura 5.2, o arquivo compactado possui tamanho 62 KB, resultando em uma taxa de compactação de

$$taxa = \frac{tamanho\ original}{tamanho\ compactado} = \frac{117\ KB}{62\ KB} = 1.887 = 53\%$$

Figura 5.2. Taxa de compactação obtida.

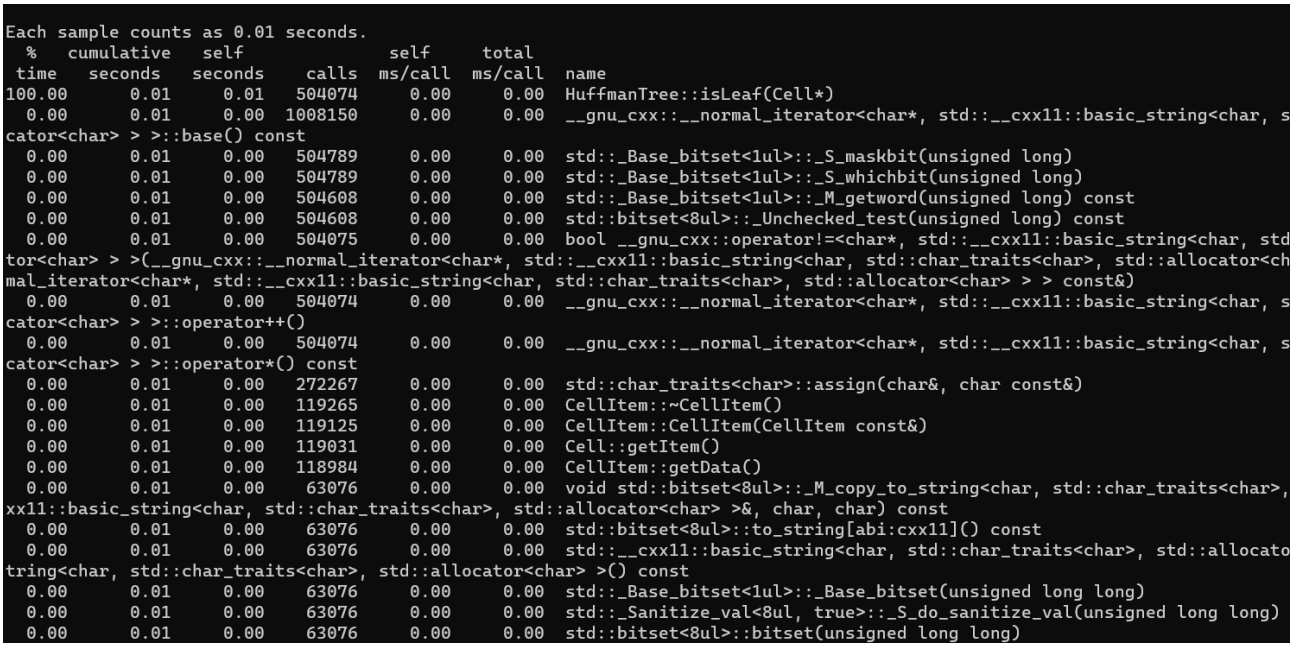


Fonte: elaboração própria.

Assim, conseguimos obter uma redução de 53% do tamanho do arquivo de entrada, sem perda de informação. Para essa mesma entrada de 117 KB, analisamos a saída com o gprof, obtendo os dados da Figura 5.3.

Analisando os dados da Figura 5.3, temos que a grande maioria das chamadas são para operadores da std::bitset e iteradores de caracteres e strings. O método huffmanTree.isLeaf(), que determina se uma célula é uma folha na árvore de Huffman, também é bastante chamado. Por fim, os métodos da célula e item de célula (Cell e CellItem) são os muito chamados em tempo de execução, conforme esperado uma vez que são a base da hierarquia de classes do sistema como mostra a Figura 2.1.

Figura 5.3. Análise do gprof para a compactação de um arquivo com 117 KB.



```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
100.00      0.01     0.01   504074     0.00     0.00  HuffmanTree::isLeaf(Cell*)
  0.00      0.01     0.00  1008150     0.00     0.00  __gnu_cxx::__normal_iterator<char*, std::__cxx11::basic_string<char, s
cator<char> > >::base() const
  0.00      0.01     0.00   504789     0.00     0.00  std::_Base_bitset<1ul>::_S_maskbit(unsigned long)
  0.00      0.01     0.00   504789     0.00     0.00  std::_Base_bitset<1ul>::_S_whichbit(unsigned long)
  0.00      0.01     0.00   504608     0.00     0.00  std::_Base_bitset<1ul>::_M_getword(unsigned long) const
  0.00      0.01     0.00   504608     0.00     0.00  std::bitset<8ul>::_Unchecked_test(unsigned long) const
  0.00      0.01     0.00   504075     0.00     0.00  bool __gnu_cxx::operator!=<char*, std::__cxx11::basic_string<char, std
tor<char> > >(__gnu_cxx::__normal_iterator<char*, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<ch
mal_iterator<char*, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > > const&)
  0.00      0.01     0.00   504074     0.00     0.00  __gnu_cxx::__normal_iterator<char*, std::__cxx11::basic_string<char, s
cator<char> > >::operator++()
  0.00      0.01     0.00   504074     0.00     0.00  __gnu_cxx::__normal_iterator<char*, std::__cxx11::basic_string<char, s
cator<char> > >::operator*() const
  0.00      0.01     0.00   272267     0.00     0.00  std::char_traits<char>::assign(char&, char const&)
  0.00      0.01     0.00   119265     0.00     0.00  CellItem::~CellItem()
  0.00      0.01     0.00   119125     0.00     0.00  CellItem::CellItem(CellItem const&)
  0.00      0.01     0.00   119031     0.00     0.00  Cell::getItem()
  0.00      0.01     0.00   118984     0.00     0.00  CellItem::getData()
  0.00      0.01     0.00    63076     0.00     0.00  void std::bitset<8ul>::_M_copy_to_string<char, std::char_traits<char>,
xx11::basic_string<char, std::char_traits<char>, std::allocator<char> >&, char, char) const
  0.00      0.01     0.00    63076     0.00     0.00  std::bitset<8ul>::to_string[abi:cxx11]() const
  0.00      0.01     0.00    63076     0.00     0.00  std::__cxx11::basic_string<char, std::char_traits<char>, std::allocato
tring<char, std::char_traits<char>, std::allocator<char> >() const
  0.00      0.01     0.00    63076     0.00     0.00  std::_Base_bitset<1ul>::_Base_bitset(unsigned long long)
  0.00      0.01     0.00    63076     0.00     0.00  std::_Sanitize_val<8ul, true>::_S_do_sanitize_val(unsigned long long)
  0.00      0.01     0.00    63076     0.00     0.00  std::bitset<8ul>::bitset(unsigned long long)
```

Fonte: elaboração própria.

Para obter resultados experimentais mais robustos da taxa de compressão, é interessante calcular a taxa de compressão para arquivos de entrada de diferentes tamanhos. A Tabela 5.1 exibe os resultados de vários testes feitos, todos com a mesma metodologia descrita acima para o teste de 117 KB de entrada.

Tabela 5.1. Resultados de testes para determinar a taxa de compressão média do sistema.

Entrada (KB)	Saída (KB)	Taxa de Compressão (%)
0.045	0.058	128.88
0.947	0.549	57.98
15	8	53.33
28	15	53.75
117	62	52.99
390	207	53.08

Fonte: elaboração própria.

Analisando os dados da Tabela 5.1, verificamos que a taxa de compressão média é de 53% para grandes arquivos. É bastante interessante observar que, para pequenos arquivos de entrada, o sistema na verdade aumenta o tamanho em bytes. Isso ocorre pois o espaço ocupado pela codificação da árvore (e os 64 bits de header para indicar os tamanhos dos dados e códigos de árvore) ocupam muito espaço relativamente ao tamanho do arquivo, resultando em baixa taxa de compressão para pequenos arquivos.

Outro ponto interessante é que o teste com entrada de 390 KB gastou bastante tempo para executar, e a maior parte do tempo foi na compactação. Isso está condizente com a análise de complexidade assintótica feita na seção 4: a complexidade de compactação $O(n\,lg\,n)$ gasta mais tempo que a de descompactação $O(n)$ para grandes valores de n.

6 CONCLUSÕES

Tendo em vista os objetivos do TP, foi possível desenvolver um sistema que compacta e descompacta arquivos de texto, com uma taxa de compressão razoável. Foi interessante observar como os TADs lista encadeada e árvore binária podem ser utilizados em conjunto, bem como a atuação de algoritmos de ordenação como o CountingSort.

A manipulação de arquivos a nível de bits e bytes também foi bastante enriquecedora, apesar de bastante desafiadora inicialmente. Por fim, o uso de algoritmos gulosos tais como o Algoritmo de Huffman integrado com os TADs vistos ao longo da disciplina foi bastante interessante.

7 REFERÊNCIAS

CORMEN, Thomas H., et al. Algoritmos. Teoria e Prática. 3 ed. Rio de Janeiro. Elsevier Editora. 2012.

GEEKS FOR GEEKS. Text FIle Compression and Decompression using Huffman Coding. Disponível em: <https://www.geeksforgeeks.org/text-file-compression-and-decompression-using-huffman-coding/>. Acesso em 09 de jun. 2023.

CHAIMOWICZ, Luiz. PRATES, Raquel. Slides Estruturas de Dados: Listas Lineares. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais.

CHAIMOWICZ, Luiz. PRATES, Raquel. Slides Estruturas de Dados: Análise de Complexidade e Complexidade Assintótica. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais.

CHAIMOWICZ, Luiz. PRATES, Raquel. Slides Estruturas de Dados: Árvores. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais.

CHAIMOWICZ, Luiz. PRATES, Raquel. Slides Estruturas de Dados: Métodos de ordenação sem comparação de chaves. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais.

8 INSTRUÇÕES DE COMPILAÇÃO E EXECUÇÃO

Siga o passo a passo abaixo para compilar e executar o TP:

1. Tenha um sistema Linux com compilador g++ e make instalado e configurado;
2. Rode o comando `make clean`
3. Rode o comando `make run`
    a. Observação: é possível que, após o make run, seja printado na saída que o arquivo de entrada não existe. Para passar o arquivo de entrada, use o passo 4 abaixo.
4. Após o make run, dentro da pasta bin estará o arquivo binário “TP03” que pode ser executado. Rode o comando

`./bin/TP03 (-c ou -d) INPUT_FILE OUTPUT_FILE`

onde INPUT_FILE é o caminho completo do arquivo txt de entrada, e OUTPUT_FILE é o caminho completo do arquivo txt de saída.

A flag -c indica que o arquivo INPUT_FILE será compactado para o arquivo OUTPUT_FILE

A flag -d indica que o arquivo INPUT_FILE será descompactado para o arquivo OUTPUT_FILE

Exemplos:

./bin/TP03 -c /mnt/c/dev/estruturas-de-dados-2023-1/TP03/fileToComp.txt /mnt/c/dev/estruturas-de-dados-2023-1/TP03/fileToDecomp.txt

./bin/TP03 -d /mnt/c/dev/estruturas-de-dados-2023-1/TP03/fileToDecomp.txt /mnt/c/dev/estruturas-de-dados-2023-1/TP03/fileToComp.txt