Trabalho Prático 3
Compactação de Arquivos de Texto

Lorenzo Ventura Vagliano

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil

lorenzovagliano@dcc.ufmg.br - Matrícula: 2022035938

1 - Introdução

O problema proposto foi a implementação de um sistema de compactação de arquivos de texto baseado no **Algoritmo de Huffman** que seja capaz de compactar e descompactar arquivos.

O algoritmo de Huffman é um método de compressão de dados sem perdas. Ele busca atribuir códigos binários de tamanho variável para cada símbolo em um determinado conjunto de dados, de forma que os símbolos mais frequentes sejam representados por códigos mais curtos e os símbolos menos frequentes sejam representados por códigos mais longos. Isso permite que os símbolos frequentes sejam representados de maneira mais eficiente, resultando em uma maior taxa de compressão.

Essa codificação de símbolos pode ser feita levando em consideração letras, sílabas ou até mesmo palavras. Nessa implementação, optamos por considerar cada caractere individual como um símbolo diferente.

2 - Método

O programa foi desenvolvido em linguagem C++ e compilado pelo compilador g++ versão 11.3.0. O método de Build foi feito utilizando GNU Make.

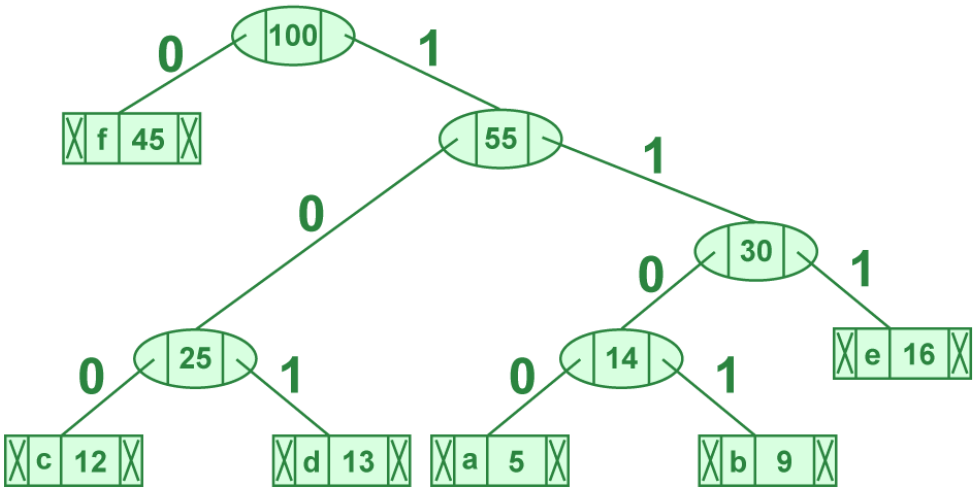
2.1 - Estruturas de Dados/Classes

Foram utilizados 3 Tipos de Dados essenciais, o Nó(ou célula), uma Lista Encadeada *especializada* e a Árvore de Huffman.

O Nó foi implementado de forma a ser utilizado tanto na Lista Encadeada, quanto na Árvore de Huffman. Por isso, ele armazena ponteiros para o nó anterior, o nó seguinte, o nó à esquerda e o nó à direita. Além disso, ele armazena um símbolo e a frequência desse mesmo símbolo.

A Lista Encadeada, caracterizada como *especializada,* busca armazenar os nós a serem utilizados para montar a árvore de Huffman. Para isso, ela armazena um símbolo em apenas um nó. Ou seja, ao se adicionar um símbolo, ela aumenta a frequência de um símbolo já existente na lista ou adiciona um novo nó. Ademais, a Lista automaticamente ordena seus nós por frequência em toda adição ou remoção, se assemelhando a uma **Fila de Prioridade**.

Por fim, a Árvore de Huffman consiste em um conjunto de nós, onde cada nó representa um símbolo ou um nó interno resultante da combinação de símbolos. Os nós são organizados de forma hierárquica, com os símbolos mais frequentes localizados mais próximos à raiz. Cada caminho da raiz até uma folha representa um código binário único atribuído a um símbolo, sendo que os caminhos para os símbolos mais frequentes são mais curtos, consequentemente gerando um código menor. Essa estrutura permite a compactação eficiente de dados, substituindo os símbolos originais por seus respectivos códigos binários.
Ilustração:



Por outro lado, a descompressão no algoritmo de Huffman consiste em reverter o processo de compactação, onde a sequência compactada de bits é percorrida bit a bit. A partir da raiz da árvore de Huffman reconstruída, cada bit é usado para seguir o caminho correspondente na árvore. Quando um caminho leva a uma folha, o símbolo correspondente é recuperado e adicionado aos dados descompactados. Esse processo é repetido até que todos os bits compactados sejam percorridos e os dados originais sejam totalmente reconstruídos, revertendo assim o processo de compressão e restaurando os dados à sua forma original.

2.2 - Funções

criarLista(string text)

A função criarLista é responsável por criar uma lista encadeada com base no texto especificado como parâmetro. Inicialmente, é criada uma nova instância da classe List chamada list. Em seguida, a função percorre o texto caractere por caractere usando um loop, e para cada caractere encontrado, o adiciona à lista por meio do método adicionar da classe List. Ao final do loop, a função retorna a lista criada, contendo todos os caracteres do texto de origem devidamente armazenados em nós individuais da lista encadeada. Essa lista é utilizada posteriormente no algoritmo de Huffman para construir a árvore de codificação.

criarArvore(List *list)

A função criarArvore é responsável por construir a árvore de Huffman com base em uma lista encadeada especificada como parâmetro. Inicialmente, a função verifica se a lista é nula ou vazia e, se

isso ocorrer, lança uma exceção. Em seguida, a função entra em um loop que se repete até que reste apenas um nó na lista. A cada iteração do loop, os dois primeiros nós da lista são selecionados como os nós iniciais. Em seguida, é calculada a frequência do novo nó pai, que é a soma das frequências dos nós iniciais. Um novo nó pai é criado com os nós iniciais como filhos e adicionado à lista. Por fim, os nós iniciais são removidos da lista. O processo continua até que reste apenas um nó na lista, que é o nó raiz da árvore de Huffman. Esse nó raiz é retornado como resultado da função. Esse algoritmo de construção da árvore de Huffman é baseado no princípio de mesclar os nós de menor frequência até que reste apenas um nó raiz que representa a árvore completa.

criarCodificacao(string* codes, Node* node, string code),
criarCodificacao(Node *raiz)

A função criarCodificacao é responsável por criar a tabela de códigos com base na árvore de Huffman. Ela recebe como parâmetro o nó raiz da árvore de Huffman e retorna a tabela de códigos.

A função inicia criando um array de strings chamado codes para armazenar os códigos de cada símbolo. Em seguida, chama a função auxiliar criarCodificacao passando como argumentos a tabela de códigos, o nó raiz e uma string vazia que será usada para construir os códigos.

A função auxiliar percorre a árvore de forma recursiva. Quando encontra uma folha (nó sem filhos), atribui o código atual à posição correspondente na tabela de códigos, utilizando o valor ASCII do símbolo como índice.

Dessa forma, a função percorre toda a árvore de Huffman e preenche a tabela de códigos com os códigos correspondentes a cada símbolo. Essa tabela será utilizada posteriormente na compressão e descompressão dos dados utilizando a codificação de Huffman.

arvoreParaString(Node* node, string saida),
arvoreParaString(Node* node)

A função arvoreParaString converte a árvore de Huffman em uma representação em string. Ela recebe como parâmetro o nó raiz da árvore. Então, ela chama a função auxiliar arvoreParaString passando como argumentos o nó raiz e uma string vazia, convertendo a árvore de Huffman em uma representação em string. Ela recebe como parâmetros o nó atual a ser processado e a string construída até o momento.
A função verifica se o nó é uma folha verificando se ele não tem filhos. Se for uma folha, adiciona o caractere '1' na string para indicar uma folha, seguido pelo símbolo contido no nó.

Caso contrário, se o nó não for uma folha, adiciona o caractere '0' na string para indicar um nó interno. Em seguida, realiza chamadas recursivas da função para processar os nós filhos esquerdo e direito, passando a string atualizada como parâmetro.

Dessa forma, a função percorre a árvore de Huffman em pré-ordem (raiz, esquerda, direita), construindo uma string que representa a estrutura da árvore. Essa representação será utilizada posteriormente na descompressão para reconstruir a árvore de Huffman.

write(string text, string filename),
write(string trie, int size, string bits, string filename)

A função write é responsável por escrever a árvore de Huffman (trie), o tamanho da string original e a sequência de bits em um arquivo. Ela recebe como parâmetros a trie convertida em formato de string (trie), o tamanho da string original (size), a sequência de bits construída (bits) e o nome do arquivo de destino (filename).

A função inicia criando uma string vazia chamada saida para armazenar o conteúdo que será escrito no arquivo. Em seguida, a trie é adicionada à saída concatenando a string trie com a variável saida. O tamanho da string original é convertido para string utilizando stringstream e concatenado com saida.

A função criarBytes é chamada para converter a sequência de bits em bytes e retornar uma string com os bytes correspondentes. Essa string de bytes é concatenada com saida. Por fim, a função write é chamada passando a string saida e o nome do arquivo filename para salvar o conteúdo no arquivo de destino.

comprimir(string filename, string outputFileName)

Essa é a função principal de compressão, comprimindo o arquivo com as funções definidas anteriormente. Para isso, ela recebe como parâmetros o nome do arquivo a ser comprimido (filename) e o nome do arquivo de saída onde o resultado da compressão será salvo (outputFileName).

A função inicia lendo o conteúdo do arquivo especificado e armazenando-o na variável text. Em seguida, uma lista de nós é criada a partir do texto, onde cada caractere do texto é representado por um nó na lista.

A árvore de Huffman é criada a partir da lista de nós usando a função criarArvore, que retorna o nó raiz da árvore. A tabela de codificação é criada a partir da árvore de Huffman usando a função criarCodificacao, que retorna um array de strings contendo os códigos correspondentes a cada símbolo.

A árvore de Huffman é convertida em uma string usando a função arvoreParaString, armazenando o resultado na variável trie.

O texto original é convertido em uma sequência de bits usando a tabela de codificação, através da função arvoreParaString que recebe o texto e a tabela de codificação como parâmetros, e armazenando o resultado na variável bits.

Por fim, a função write é chamada passando a trie, o tamanho original do texto e a sequência de bits comprimidos, juntamente com o nome do arquivo de saída, para salvar todas as informações no arquivo especificado.

criarBytes(string bits)

A função criarBytes recebe uma sequência de bits como parâmetro. Então, ela utiliza um laço de repetição para percorrer a sequência de bits. A cada iteração, verifica-se se o bit atual é '1' ou '0'. Caso seja '1', o byte é deslocado um bit para a esquerda e o bit menos significativo é definido como 1. Caso seja '0', apenas o deslocamento é feito.

Após processar cada bit, verifica-se se foi atingido o final da sequência de bits. Se sim, é verificado quantos bits faltam para completar um byte. Em seguida, são adicionados zeros à direita do byte para formar um byte completo. A cada iteração do laço, o byte resultante é adicionado à string de saída. Ao final do laço, a string de saída contém os bytes resultantes da conversão dos bits.

Por fim, a função retorna a string de saída contendo os bytes criados.

lerArvore(string text, int* index),
lerArvore(string* text)

A função lerArvore recebe uma string contendo a representação de uma árvore de Huffman. Ela então utiliza um índice index para controlar a posição de leitura na string. Inicialmente, o índice é definido como -1. Em seguida, a função chama a função auxiliar lerArvore passando a string e o ponteiro para o índice como parâmetros.

A função auxiliar começa incrementando o valor do índice para avançar para o próximo caractere na string. Em seguida, verifica se o caractere atual na posição indicada pelo índice é '1'. Se for '1', significa que o nó atual é uma folha. A função incrementa novamente o valor do índice para avançar para o próximo caractere que representa o símbolo da folha. Em seguida, cria um novo nó folha com o símbolo correspondente e retorna o nó criado.

Se o caractere atual não for '1', significa que o nó atual é intermediário. Nesse caso, a função cria um novo nó intermediário e chama recursivamente a si mesma duas vezes para construir os nós filhos esquerdo e direito do nó atual. A função passa a string e o ponteiro do índice como parâmetros para as chamadas recursivas. A cada chamada recursiva, o valor do índice é atualizado para acompanhar a posição de leitura na string. Ao final da função, o ponteiro para o nó raiz da árvore lida é retornado.

lerTamanho(string* text)

A função lerTamanho recebe uma string contendo o tamanho da string original seguido dos dados e retorna o valor do tamanho lido. A função utiliza um objeto stringstream para extrair o valor do tamanho da string convertendo-o para o tipo int. Em seguida, a função atualiza a string removendo o tamanho lido, utilizando a função replace para substituir a porção da string que representa o tamanho pelo restante dos dados. Por fim, o valor do tamanho é retornado.

arvoreParaString(string text, string* table)

A função arvoreParaString recebe uma string de texto e um ponteiro para uma tabela de codificação. A função percorre cada caractere da string de texto e utiliza a tabela de codificação para obter a representação codificada correspondente a cada caractere. A representação codificada é concatenada em uma string de saída, que é retornada no final da função. Dessa forma, a função realiza a conversão da string de texto em uma representação codificada com base na tabela de codificação fornecida.

bytesParaBits(string bytes)

A função bytesParaBits recebe uma sequência de bytes como entrada e converte essa sequência em uma representação binária de bits. A função percorre cada byte da sequência e, para cada byte, itera através dos seus 8 bits. Para cada bit, verifica se o bit mais significativo é 1 ou 0 e adiciona '1' ou '0',

respectivamente, na string de bits resultante. Em seguida, realiza um deslocamento dos bits do byte para a esquerda. Esse processo é repetido para todos os bytes da sequência, resultando na conversão completa da sequência de bytes em uma representação binária de bits. A string de bits resultante é então retornada pela função. Essa função desempenha um papel importante no processo de descompressão de Huffman, convertendo a sequência de bytes comprimidos de volta para a representação binária original.

traduzir(string bits, int size, Node* raiz)

A função traduzir recebe uma sequência de bits, o tamanho original do texto antes da compactação e a raiz da árvore de Huffman. A função realiza a tradução dos bits de volta para o texto original usando a árvore de Huffman. Para cada símbolo do texto original, a função percorre a árvore a partir da raiz, movendo-se para o nó da esquerda se o bit correspondente for '0' ou para o nó da direita se o bit for '1'. Esse processo é repetido até chegar a um nó folha, onde o símbolo correspondente é adicionado ao texto resultante. A função continua esse processo até que todos os símbolos do texto original sejam traduzidos.

descomprimir(string filename, string outputFileName)

Essa é a função principal de descompressão, comprimindo o arquivo com as funções definidas anteriormente. Primeiramente, ela lê o conteúdo do arquivo comprimido, armazenando-o na variável text. Em seguida, a árvore de Huffman é reconstruída a partir dos dados em text usando a função lerArvore. A função lerArvore é chamada para ler a árvore de Huffman da string text, atualizando o índice de leitura a cada nó processado.

Após reconstruir a árvore de Huffman, a função extrai o tamanho original do texto antes da compactação utilizando a função lerTamanho. Essa função lê o tamanho da string original a partir da string text, atualizando text para remover o tamanho lido.

Em seguida, os bytes comprimidos em text são convertidos de volta para a sequência de bits original usando a função bytesParaBits. Essa função itera sobre cada byte em text, convertendo-o em uma sequência de 8 bits (0s e 1s).

Com a sequência de bits original em mãos, a função utiliza a árvore de Huffman reconstruída para traduzir os bits de volta para o texto original. Isso é feito chamando a função traduzir.

Por fim, o texto descomprimido é salvo no arquivo de saída especificado por outputFileName usando a função write.

3 - Análise de Complexidade

Compressão:
A complexidade de tempo da função é dominada pelas seguintes etapas: criar a lista de nós para cada caractere do texto, criar a árvore de Huffman a partir da lista de nós e criar a tabela de codificação a partir da árvore. Cada uma dessas etapas possui uma complexidade de tempo de O(n log n), onde n é o número de caracteres no texto original. A criação da lista envolve percorrer o texto original e atualizar a lista de nós correspondente a cada caractere encontrado. A criação da árvore de Huffman envolve a combinação dos nós da lista até que reste apenas um nó raiz, utilizando uma fila de

prioridade. A criação da tabela de codificação envolve percorrer a árvore de Huffman e atribuir códigos binários aos caracteres.

Além das etapas mencionadas acima, a função também realiza outras operações com complexidades menores. A conversão da árvore de Huffman em uma string possui uma complexidade de tempo e memória de O(n), onde n é o número de nós na árvore. A conversão do texto original em uma sequência de bits usando a tabela de codificação possui uma complexidade de tempo e memória de O(m), onde m é o número de caracteres no texto original.

Por fim, a função salva a árvore, o tamanho original do texto e os bits comprimidos no arquivo de saída. Essa operação requer tempo e memória proporcional ao tamanho dos dados a serem salvos.

Em resumo, a complexidade de tempo total da função é **O(n log n + m)**, onde n é o número de caracteres no texto original e m é o número de caracteres no texto comprimido. A complexidade de memória é **O(n)**, onde n é o número de nós na árvore de Huffman, mais a quantidade de memória necessária para armazenar os dados no arquivo de saída.

Descompressão:
A complexidade de tempo da função é dominada pela reconstrução da árvore de Huffman, que possui uma complexidade de tempo de O(n), onde n é o número de caracteres no texto comprimido. A reconstrução da árvore envolve a leitura da string de texto comprimido e a criação dos nós da árvore com base nas informações contidas na string. A complexidade de memória da reconstrução da árvore de Huffman é O(n), onde n é o número de caracteres no texto comprimido.

Após a reconstrução da árvore, a função obtém o tamanho original do texto antes da compactação, o que consome tempo constante O(1) e não requer memória adicional. Em seguida, os bytes comprimidos são convertidos de volta para a sequência de bits original, o que resulta em uma complexidade de tempo de O(n), onde n é o número de bytes no texto comprimido. Essa conversão não consome memória adicional além da string resultante, resultando em uma complexidade de memória de O(n), onde n é o número de bytes no texto comprimido.

Após a conversão dos bytes para a sequência de bits, a função traduz a sequência de bits de volta para o texto original usando a árvore de Huffman. Essa tradução envolve percorrer os bits e navegar na árvore até chegar a um nó folha. A complexidade de tempo é proporcional ao número de bits na sequência, resultando em uma complexidade de tempo de O(m), onde m é o número de bits na sequência de bits. A complexidade de memória é proporcional ao tamanho da string resultante, resultando em uma complexidade de memória de O(m), onde m é o número de bits na sequência de bits.

Por fim, o texto descomprimido é salvo no arquivo de saída, o que requer tempo proporcional ao tamanho do texto descomprimido e memória proporcional a essa string. Portanto, a complexidade de tempo e memória depende do tamanho do texto descomprimido.

Em resumo, a complexidade de tempo total da função é **O(n + m)**, onde n é o número de caracteres no texto comprimido e m é o número de bits na sequência de bits. A complexidade de memória é **O(n + m)**, onde n é o número de caracteres no texto comprimido e m é o número de bits na sequência de bits.

4 - Estratégias de Robustez

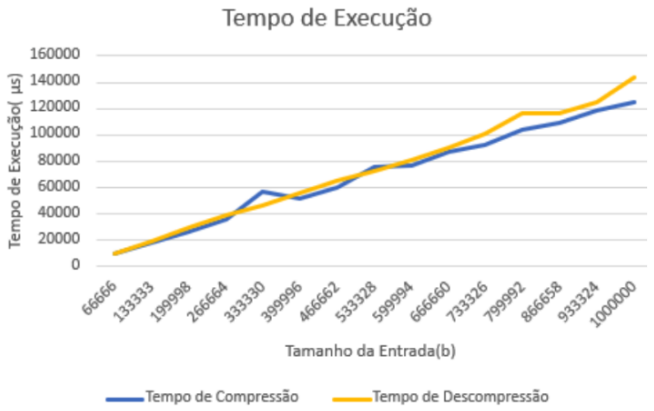
A fim de prover mais robustez ao sistema. Além de seguir boas práticas de desenvolvimento, diversos casos de erro foram tratados.

As funções referentes à manipulação de arquivos tomam precauções para garantir que o arquivo foi encontrado e que ele foi aberto corretamente. Além disso, a criação da árvore verifica se a lista utilizada para a construção está vazia ou se é representada por um valor nulo.

Por último, ao se utilizar um comando de execução errado, o usuário também é alertado.

5 - Análise Experimental

Dados Aleatórios:
Inicialmente, para avaliar a performance da compressão, foi calculada a razão entre o arquivo comprimido e o arquivo original, e o tempo de execução para arquivos de texto aleatórios progressivamente maiores.

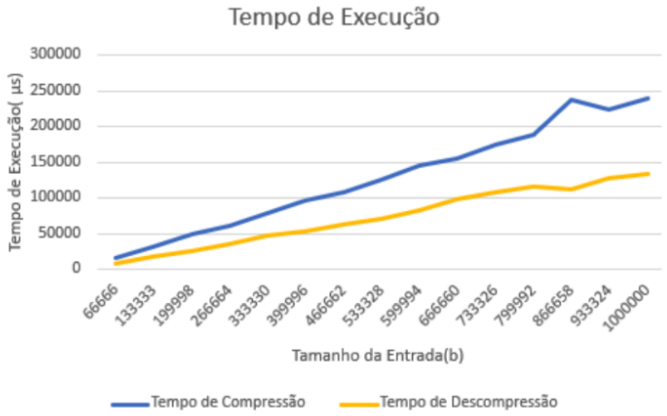


Entretanto, logo ao coletar esse resultados, foi possível perceber uma diferença extremamente pequena entre o tamanho do arquivo comprimido e do arquivo original.

Além da compressão, como comentado anteriormente, aparentar ser muito mais ineficaz do que o previsto, o tempo de execução não reflete a análise de complexidade, sendo que para dados tão grandes, já deveria ser possível visualizar a diferença entre O(n log n + m) e O(n + m).

No entanto, é importante observar que o algoritmo de Huffman não é adequado para todos os tipos de dados. Em geral, quanto mais repetições e padrões existirem nos dados originais, maior será a taxa de compressão obtida pelo algoritmo de Huffman. Por isso, iremos fazer o mesmo teste com dados literários.

Dados Literários:
Foram calculadas as mesmas medidas. Porém trechos de *An Essay Concerning Human Understanding, John Locke(Domínio Público)*, obtidos em ASCII através do recurso Textfiles, foram utilizados como entrada.



Esses resultados comprovam nossa hipótese. Houve uma melhora de cerca de 30% na compactação do arquivo em comparação com a amostra aleatória.

Além disso, agora é possível observar experimentalmente a diferença de complexidade de tempo entre a compressão e a descompressão. Em um texto literário, é comum encontrar padrões e repetições, como letras, palavras ou frases frequentes. Isso resulta em uma distribuição de frequência desigual dos símbolos e, consequentemente, em uma árvore de Huffman mais desequilibrada. A construção dessa árvore de Huffman, feita na compressão, pode exigir um tempo significativamente maior, já que os símbolos mais frequentes devem ser identificados e ordenados corretamente na árvore. Enquanto isso, a descompressão permanece igual em ambos os casos, o que também pode ser observado graficamente.

6 - Conclusões

Por intermédio desse projeto, foi possível experimentar com a implementação do Algoritmo de Huffman. Essa experiência habilitou o aprendizado sobre codificação de caracteres como ASCII e UTF-8 e estruturas de dados como a Árvore de Huffman.

A partir desses esforços, foi possível postular teorias acerca da eficiência do Huffman para datasets diferentes, indicando um uso melhor para dados com repetições, como textos literários, ao invés de dados aleatórios. Posteriormente, a partir da análise experimental, foi possível confirmar essa teoria.

7 - Bibliografia

- HUFFMAN, D. A Method for the Construction of Minimum-Redundancy Codes. **Proceedings of the IRE**, v. 40, n. 9, p. 1098–1101, set. 1952.
- SCHWARZ, K.; STEPP, M. **Huffman Encoding and Data Compression Handout**. [s.l: s.n.]. Disponível em: <https://web.stanford.edu/class/archive/cs/cs106x/cs106x.1192/resources/minibrowser2/huffman-encoding-supplement.pdf>. Acesso em 1 jul 2023.
- SEDGEWICK, R.; WAYNE, K. **Algorithms**. Upper Saddle River, Nj: Addison-Wesley, 2011.
- SCOTT, T. **How Computers Compress Text: Huffman Coding and Huffman Trees**. Disponível em: <https://www.youtube.com/watch?v=JsTptu56GM8>. Acesso em 1 jul. 2023.
- SCOTT, J. **TEXTFILES**. Disponível em: <http://textfiles.com/>. Acesso em 1 jul 2023.

8 - Instruções para compilação e execução

Compilação
Primeiramente, deve-se extrair os conteúdos do ZIP do projeto em uma pasta.

Antes de compilar o programa, é necessário ter instalado o GCC e o GNU Make.

Para compilar o programa, após abrir um terminal no diretório raiz do projeto, utilize o seguinte comando:

```
make
```

Execução
Insira os arquivos de texto a serem compactados no diretório /bin/.

Então, acesse o diretório /bin/ pelo terminal. Para isso, utilize o seguinte comando:

```
cd bin
```

Para compactar um arquivo de texto, insira um comando no seguinte formato:

```
./Huffman -c <nome do arquivo a ser compactado> <nome do arquivo de destino>
```

Para descompactar um arquivo compactado, insira um comando no seguinte formato:

```
./Huffman -d <nome do arquivo a ser descompactado> <nome do arquivo de destino>
```

Exemplos:
./Huffman -c texto.txt compacto.z
./Huffman -d compacto.z descompactado.txt