Universidade Federal de Minas Gerais

Departamento de Ciência da Computação

Trabalho Prático 3 – **Estrutura de Dados**

Daniel Nogueira Junqueira – 2021072244

danijnog@ufmg.br

1. Introdução

O seguinte programa tem por objetivo realizar a compressão e a descompressão de arquivos utilizando para isso o Algoritmo de Huffman, implementado e desenvolvido no projeto.

2. Método

O programa foi desenvolvido e feito no WSL 2, com a distribuição Linux Ubuntu, utilizando a linguagem C++. As especificações do computador são:

- Sistema Operacional em que está ativo o WSL: Windows 11.
- Processador: Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz 2.59 GHz
- RAM instalada: 16,0 GB.
- Tipo de sistema: 64 bits.

2.1 Estrutura de Dados e implementação

A estrutura do algoritmo de Huffman foi implementado da seguinte forma: foi começado implementando a tabela de frequência dos caracteres, depois disso é feito a lista encadeada ordenada e o preenchimento da lista encadeada de acordo

com a tabela de frequência gerada e, com isso, é feito a árvore de Huffman a partir da lista encadeada de acordo com os nós e os caracteres em cada nó da lista.

Com a árvore montada, se torna possível desenvolver o dicionário de caracteres de acordo com a frequência de cada caracter, e com isso montar a codificação total do texto do arquivo passado como parâmetro e armazenar isso no programa. Porém, até o momento, essa codificação ainda não fica armazenada em binário, mas sim como um ponteiro de char, e então uma outra função no programa no arquivo *compress.cpp* é responsável por armazenar esse texto passado como parâmetro de forma binária através de deslocamento de bits.

Dessa forma, o arquivo de texto é comprimido da forma que está definida na regra de negócio do TP para uma forma binária.

Além disso, depois dessa compressão é possível realizar a descompressão do arquivo também do jeito que está definido no trabalho.

2.2 Arquivos e Funções

Os arquivos do programa estão divididos nas pastas “src” e “include”, sendo que na pasta “include” estão os arquivos de cabeçalho como esperado e na pasta “src” a sua implementação de fato.

Os arquivos na pasta “include” são as seguintes:

Compress.h

Dictionary.h

Entry.h

Exceptions.h

frequenceTable

Huffman.h

No.h

Todos os métodos de cada arquivo foram desenvolvidos a partir de classes, com exceção do arquivo ‘entry.h’ que serve para ler e validar a entrada e o ‘exceptions.h’ que tem como objetivo tratar as exceções do programa.

Logo, todos os arquivos com exceção dos dois mencionados no parágrafo acima contém seu construtor, os quais não abordarei aqui para não ficar repetitivo e por ser uma coisa mais simples, portanto, vou focar nos métodos das classes.

Descrição dos métodos de cada **arquivo**:

compress.h:

- Método **compress:** Realiza a compressão de Huffman através de um ponteiro de char codificado já gerado através da árvore de Huffman. Realiza essa compressão em um arquivo binário passado como parâmetro.
- Método **isBit1:** Verifica se um char passado como parâmetro é bit 1, retornando qualquer valor de 0 caso verdaderio.

dictionary.h:

- Método **alocateDictionary:** Aloca dinâmicamente a matriz de caracteres de acordo com a quantidade de colunas, que é definida pela altura da árvore.
- Método **buildDictionary:** Constroi o dicionário de caracteres com os valores 0 e 1 de acordo com a árvore de Huffman.
- Método **printDctionary:** Printa o dicionário de dados, com o caracter e o seu código definido.

entry.h*:*

- Método **countCharactersOfFile:** Retorna a quantidade de caracteres de um arquivo.
- Método **readFile:** Lê o arquivo passado como parâmetro e atribui ao ponteiro de char passado também como parâmetro os caracteres do arquivo lido.
- Método **validateEntry:** Lê a entrada do programa e valida a mesma de acordo com as regras de negócio definidas na entrada do programa no enunciado do TP.
- Método **writeEntryOnAuxFile:** Escreve o texto do arquivo de entrada em um arquivo auxiliar durante a compactação com o objetivo de utilizar esse arquivo auxiliar para remontar a tabela de frequência e a árvore de Huffman para realizar a descompactação.
- Método **getFileSize:** Retorna o tamanho de um arquivo em Bytes.

exceptions.h:

- Contém todas as classes de exceções que vão ser utilizadas e lançadas no programa.

frequenceTable.h:

- Método ***initializeFrequenceTableWith0: c***omo diz o próprio nome, inicializa toda a tabela de frequência com o valor 0.
- Método ***fillfrequenceTable***: preenche a tabela de frequência com o texto passado como parâmetro no método.

- Método ***printFrequenceTable***: printa a tabela de frequência, com o caracter em seu formato normal, em seu código na tabela ASC e sua frequência também.

huffman.h:

- Método **sortedInsert:** Insere na lista encadeada de forma ordenada.
- Método **fillList:** Preenche a lista ordenada de acordo com a tabela de frequência gerada.
- Método **printList:** Printa a lista encadeada ordenada.
- Método **popFromList:** Remove um elemento do início da lista encadeada.
- Método **buildTree:** Constrói a árvore de Huffman de acordo com a lista encadeada criada.
- Método **printTree:** Printa a árvore de Huffman construída.

no.h:

- Método **treeHeight**: retorna a altura da árvore.
- Método **getFrequence**: retorna a frequência de um nó.
- Método **getCharacter**: retorna o caracter de um nó.
- Método **getNextNo**: retorna o próximo nó.
- Método **getLeftNo**: retorna o nó a esquerda.
- Método **getRightNo**: retorna o nó a direita.
- Método **setFrequence**: “seta” a frequência de um nó.
- Método **setCharacter**: “seta” o caracter de um nó.
- Método **setNextNo**: “seta” o próximo nó de um nó.
- Método **setLeftNo:** “seta” o nó a esquerda de um nó.
- Método **setRightNo**: “seta” o nó a direita de um nó.

3. Análise de complexidade

Arquivo ***compress.h***:

Método **compress:** recebe como parâmetro um array de char codificado e um arquivo, em que percorre todo o array de char de 0 e 1 até o seu final e vai escrevendo no arquivo binário de acordo com as operações de deslocamento de bit em C++. Complexidade de espaço e tempo O(n), onde ‘n’ é o tamanho do array de entrada.

Método **isBit1**: recebe como parâmetro um char e um inteiro ‘pos’, em que verifica se esse char passado como parâmetro é bit 1 através do deslocamento de ‘pos’ bits da representação em binário do número ‘1’ e fazendo a comparação ‘and’ bit a bit

do char passado como parâmetro e da representação do número 1 deslocado. Complexidade de tempo e espaço constante O(1).

Método **decompress**: recebe como parâmetro o nó raiz, o nome do arquivo compactado e o nome do arquivo que será gerado como output do arquivo compactado. Abre o arquivo compactado passado como parâmetro e realiza a leitura do arquivo verificando se os bits são 1 ou 0 e realizando a operação de escrita no arquivo de texto output analisando se já foi atingido as folhas da árvore ou não, gerando assim o arquivo descompactado. Complexidade de tempo e espaço O(n), onde 'n'.

Arquivo ***dictionary.h*****:**

Método **alocateDictionary**: recebe como parâmetro a quantidade de colunas e aloca memória para uma matriz de tamanho fixo TAM linhas e COLUMS colunas, onde TAM é 256 para alcançar todos os caracteres da Tabela ASC e columns é a altura da árvore. Complexidade de tempo e espaço O(TAM * COLUNS).

Método **buildDictionary**: recebe como parâmetro o ponteiro de ponteiro de dictionary, o nó raiz da árvore, o caminho e a quantidade de colunas. Possui chamadas recursivas, em que a complexidade de tempo depende do tamanho da árvore. Portanto, complexidade de tempo O(n), onde 'n' é o número de caracteres diferentes do texto. A complexidade de espaço fica O(n * colums), devido a chamada recursiva enquanto ainda não foi alcançado a folha da árvore.

Método **printDictionary**: recebe como parâmetro o ponteiro de ponteiro de dictionary e faz uma iteração fixa TAM vezes sobre o dictionary. Complexidade de tempo O(TAM) e complexidade de espaço constante O(1).

Método **calculateSizeOfString**: calcula o tamanho da codificação dos caracteres que estão no dicionário, iterando sobre cada caracter do texto passado como parâmetro na função. Complexidade de tempo O(n), onde 'n' é o tamanho do texto passado como parâmetro e complexidade de espaço O(1).

Método **encode**: recebe como parâmetro o ponteiro de ponteiro de dictionary e um texto e percorre cada caracter do texto no dicionário, concatenando em uma variável a codificação de cada caracter no dicionário, retornando essa variável com o dicionário inteiro codificado. Complexidade de tempo O(n), onde 'n' é o tamanho do texto passado como parâmetro e complexidade de espaço O(1).

Método **decode**: recebe como parâmetro o nó raiz da árvore e o texto codificado (com valores 0 e 1), percorrendo esse texto e fazendo o devido caminhamento na árvore de acordo com os valores 0 e 1 do texto codificado, e concatenando em um array de char o caracter encontrado quando se atinge uma folha da árvore, retornando esse array concatenado que conterá o texto decodificado. Complexidade de tempo O(n), onde 'n' é o tamanho do texto codificado e complexidade de espaço O(1).

Arquivo ***entry.h***:

Método **countCharacterOfFile**: conta a quantidade de caracteres de um arquivo passado como parâmetro. Complexidade de tempo O(n), onde ‘n’ é o tamanho do arquivo e complexidade de espaço constante O(1).

Método **readFile**: lê um arquivo passado como parâmetro e atribui cada caracter do arquivo ao array de char passdo como parâmetro também no método. Complexidade de tempo O(n), onde ‘n’ é o tamanho do arquivo e complexidade de espaço constante O(1).

Métodos **validateEntry**, **writeEntryOnAuxFile** e **getFileSize**: Complexidade de tempo e espaço constantes O(1).

Arquivo ***frequenceTable.h***:

Métodos **initializeFrequenceTableWith0** e **printFrequenceTable**: Complexidade de tempo O(TAM) e espaço constante O(1).

Método **fillFrequenceTable**: Realiza um loop até o fim do array de char passado como parâmetro na função, para preencher o valor do mesmo na tabela de frequência. Complexidade de tempo O(n), onde ‘n’ é o tamanho do array de char e complexidade de espaço constante O(1).

Arquivo ***huffman.h***:

Métodos **sortedInsert** e **printList**: A inserção na lista encadeada ordenada e a impressão da mesma depende da quantidade de nós. Complexidade de tempo O(n), onde ‘n’ é a quantiade de nós e complexidade de espaço constante O(1).

Método **fillList**: Preenche a lista encadeada de acordo com a tabela de frequência. Complexidade de tempo O(n), onde ‘n’ é o tamanho da tabela de frequência e complexidade de espaço. Por criar um novo nó para a inserção cada vez que o valor na tabela de frequência do caracter for maior que 0, possui complexidade de espaço O(n) também.

Método **popFromList**: complexidade de tempo e espaço constantes O(1).

Método **buildTree**: constroi a árvore a partir da lista encadeada, logo depende da quantidade de nós. Complexidade de tempo e espaço O(n), onde ‘n’ é a quantidade de nós.

Método **printTree**: implementada de forma recursiva, imprimindo os caracteres da árvore se foi chegado em nas folhas da árvore ou não. Sua complexidade de tempo depende da quantidade de nós, O(n), e sua complexidade de espaço depende da

altura da árvore que é passado como parâmetro e vai sendo incrementado a cada chamada recursiva, O(n).

Arquivo ***no.h***:

Método **treeHeight**: retorna a altura da árvore. Envolve chamadas recursivas, então o método depende da quantidade de nós da árvore. Complexidade de tempo e espaço O(n), onde ‘n’ é a quantidade de nós da árvore.

Os demais métodos do arquivo (citados mais acima nesse documento) são todos constantes O(1).

4. Estratégias de robustez

Tratamento de erros: No programa, o tratamento de erros é feito a partir da definição de classes de exceções que serão lançadas no programa no arquivo “exceptions.h”, já que o C++ nos permite lançar exceções, ao contrário da linguagem C. Dessa forma, o programa se torna mais robusto e mais organizado em relação ao tratamento de erros e exceções.

Memória: no programa ocorre vazamento de memória, que vai aumentando a medida que o arquivo de entrada aumenta, porém isso é originário do código de Huffman e é necessário para que o algoritmo seja executado, tornando difícil ou quase impossível ter zero vazamento de memória.

5. Análise experimental

Ao rodar o gprof para uma entrada de um texto pequeno (200 linhas), realizando a compactação e depois a descompactação, obtemos a seguinte quantidade de chamada de métodos:

% time	cumulative seconds	self seconds	calls	self Ts/call	total Ts/call	name
0.00	0.00	0.00	78075	0.00	0.00	No::getLeftNo()
0.00	0.00	0.00	40486	0.00	0.00	No::getRightNo()
0.00	0.00	0.00	11777	0.00	0.00	No::getCharacter()
0.00	0.00	0.00	9018	0.00	0.00	No::getNextNo()
0.00	0.00	0.00	6370	0.00	0.00	No::getFrequence()
0.00	0.00	0.00	637	0.00	0.00	No::setNextNo(No*)
0.00	0.00	0.00	161	0.00	0.00	No::setRightNo(No*)
0.00	0.00	0.00	161	0.00	0.00	No::setCharacter(unsigned char)
0.00	0.00	0.00	161	0.00	0.00	No::setFrequence(int)
0.00	0.00	0.00	161	0.00	0.00	No::setLeftNo(No*)
0.00	0.00	0.00	161	0.00	0.00	Huffman::sortedInsert(No*)
0.00	0.00	0.00	81	0.00	0.00	No::No()
0.00	0.00	0.00	1	0.00	0.00	_GLOBAL__sub_I__Z21countCharactersOfFilePKc
0.00	0.00	0.00	1	0.00	0.00	_GLOBAL__sub_I__ZN10DictionaryC2Ev
0.00	0.00	0.00	1	0.00	0.00	_GLOBAL__sub_I__ZN14frequenceTableC2Ev
0.00	0.00	0.00	1	0.00	0.00	_GLOBAL__sub_I__ZN7HuffmanC2Ev
0.00	0.00	0.00	1	0.00	0.00	_GLOBAL__sub_I__ZN8CompressC2Ev
0.00	0.00	0.00	1	0.00	0.00	_GLOBAL__sub_I_main

É notável a quantidade de chamada das funções relacionadas a árvore, mesmo com uma entrada relativamente pequena.

Nesse caso, o arquivo original era de 11KB e foi reduzido para 4KB no arquivo compactado.

Fazendo o mesmo teste, porém agora com uma entrada maior de 2065 linhas de texto de aproximadamente mesmo tamanho por linha igual o teste anterior:

% time	cumulative seconds	self seconds	calls	self Ts/call	total Ts/call	name
0.00	0.00	0.00	778817	0.00	0.00	No::getLeftNo()
0.00	0.00	0.00	406312	0.00	0.00	No::getRightNo()
0.00	0.00	0.00	117769	0.00	0.00	No::getCharacter()
0.00	0.00	0.00	13118	0.00	0.00	No::getNextNo()
0.00	0.00	0.00	9192	0.00	0.00	No::getFrequence()
0.00	0.00	0.00	788	0.00	0.00	No::setNextNo(No*)
0.00	0.00	0.00	199	0.00	0.00	No::setRightNo(No*)
0.00	0.00	0.00	199	0.00	0.00	No::setCharacter(unsigned char)
0.00	0.00	0.00	199	0.00	0.00	No::setFrequence(int)
0.00	0.00	0.00	199	0.00	0.00	No::setLeftNo(No*)
0.00	0.00	0.00	199	0.00	0.00	Huffman::sortedInsert(No*)
0.00	0.00	0.00	100	0.00	0.00	No::No()
0.00	0.00	0.00	1	0.00	0.00	_GLOBAL__sub_I__Z21countCharactersOfFilePKc
0.00	0.00	0.00	1	0.00	0.00	_GLOBAL__sub_I__ZN10DictionaryC2Ev
0.00	0.00	0.00	1	0.00	0.00	_GLOBAL__sub_I__ZN14frequenceTableC2Ev
0.00	0.00	0.00	1	0.00	0.00	_GLOBAL__sub_I__ZN7HuffmanC2Ev
0.00	0.00	0.00	1	0.00	0.00	_GLOBAL__sub_I__ZN8CompressC2Ev
0.00	0.00	0.00	1	0.00	0.00	_GLOBAL__sub_I_main

A quantidade de chamadas das funções é aumentada significativamente, devido a principalmente a chamada recursiva que acontece em diversos métodos.

Nesse teste, o arquivo original era de 117KB e foi reduzido para 65KB no arquivo compactado, mostrando realmente a eficácia do algoritmo que reduziu/reduz, geralmente, quase pela metade o tamanho do arquivo.

6. Conclusões

Portanto, o objetivo do programa era explorar o algoritmo de Huffman e as estruturas de dados utilizadas pelo mesmo, que no caso mostrou-se um algoritmo realmente bem eficaz na compressão de arquivos, em que pode-se observar isso na prática pela compactação e descompactação de arquivos no programa.

Achei interessante a implementação desse algoritmo, apesar de um pouco complexo com muitos métodos com várias chamadas recursivas, o que contribui fortemente para causar pequenos deslizes no código, ele se mostrou eficiente de fato e também pude estudar melhor a implementação das estruturas de dados utilizadas por ele também, como árvores e listas encadeadas.

7. Bibliografia

Conteúdos e vídeos do canal *Programa seu Futuro* sobre o Código de Huffman:
Algoritmo de Huffman em C

Apêndice

Instruções para compilação e execução

No diretório do programa, de o comando ‘make’ para compilar todos os arquivos e gerar o executável ‘main’ na pasta ‘bin’.

Com isso, agora o programa segue a regra de negócio definida no TP:

Para a **compactação**, dê o comando:

- bin/main -c arquivo_entrada.txt arquivo_compactado.wg

onde ‘*arquivo_entrada.txt*’ deve ser substituído pelo seu arquivo de entrada, e o *‘arquivo_compactado.wg*’ deve ser substituído pelo nome que você deseja dar ao arquivo compactado gerado pelo programa.

A descompactação só funciona com o arquivo compactado anteriormente.
Para a **descompactação**, dê o comando:

- bin/main -d arquivo_compactado.wg output.txt

Onde ‘*arquivo_compactado.wg*’ deve ser substituído pelo nome do arquivo compactado que você definiu durante a compactação, e o ‘*output.txt*’ deve ser substituído pelo nome desejado do arquivo de saída descompactado que será gerado.

Atualmente, no diretório do programa já existe um arquivo nomeado ‘*entrada.txt*’ que pode ser usado como teste, caso deseje.

Além disso, caso deseje, é possível passar o seguinte comando no terminal para obter o Menu de Ajuda: ‘bin/main -h’.