Trabalho Prático 2

Victor Augusto Hon Fonseca- número de matrícula: 2022035814

1.0 Introdução

Sob o contexto passado pelo enunciado do trabalho que estou trabalhando em um jornal, que o departamento financeiro reduziu o número de discos rígidos e eu, no entanto, gostaria de manter as informações presentes em todas as edições do jornal, eu devo implementar um sistema de compactação baseado no algoritmo de Huffman. Para resolver esse problema, o programa receberá como argumentos dois nomes de arquivos e uma flag “-c” (ela vem antes dos nomes dos arquivos, para indicar que deseja compactar a mensagem do primeiro arquivo e colocá-la no segundo), ou dois nomes de arquivos e uma flag “-d” (ela vem antes dos nomes dos arquivos, para indicar que deseja descompactar a mensagem do primeiro arquivo e colocá-la no segundo), ou então uma combinação das duas usando a flag “-c” com dois nomes de arquivos depois e a flag “-d” com dois nomes de arquivos depois e, a partir disso, usando estruturas de dados, classes e funções determinadas realizará essa funcionalidade

2.0 Método

O programa foi desenvolvido utilizando a linguagem de programação C++, as bibliotecas “iostream”, “string”, “limits”, “stdexcept” dessa linguagem para os arquivos, de modo geral, e, foram utilizadas tamas bibliotecas “fstream”, para a leitura das linhas de arquivos utilizados, a biblioteca “cctype”, para identificar se um caractere lido seria um número ou uma letra, “bitset” para armazenar bytes e fazer operações neles e a biblioteca “algorithm”. Além disso, vale relatar que o programa foi compilado utilizando o compilador G++, da GNU Compiler Collection, na IDE Visual Studio Code, utilizando o recurso Subsistema do Windows para Linux (WSL), do sistema operacional Windows, na IDE Visual Studio Code.

2.1 Estruturas de Dados

É importante destacar que, durante a implementação do programa, além de classes, também foi necessário utilizar estruturas de dados auxiliares, para armazenar, por exemplo, objetos do tipo TipoNo*. No meu programa, após a leitura dos caracteres do arquivo, elas são usadas para criar um objeto do tipo TipoNo* (uma estrutura de dados auxiliar criada para servir como os nós de uma árvore), o qual é armazenado dentro de uma estrutura de dados implementada por mim, chamada de “vector4”, criada utilizando de templates, o que permite escolher o tipo de elemento armazenado dentro dela. Ademais, elas também são usadas em outros momentos do programa, como, por exemplo, para armazenar objetos do tipo Par (estrutura de dados auxiliar criada que possui como atributos uma “string” chamada caractere e uma outra chamada codificação), o que constribui para a associação caractere-codificação binária, em seu interior há a classe iterador, o qual retorna um ponteiro para determinada posição do “vector4”, sendo essa composta de atributos como capacidade, uma constante, índice para o último elemento e um array, outra estrutura de dados, mais simples, usada para armazenar elementos passados, e a classe “iterador” composta por um ponteiro como um atributo. Ambas possuem métodos já esperados para essas estruturas de dados, entre eles: construtor e destrutor. Vale destacar que a classe “vector” possui métodos para adicionar elementos, retirar elementos, retornar o tamanho do vector, construir

um vector a partir de iteradores e um iterador que aponta para sua primeira posição e outro para depois da última posição. Já a classe “iterator” possui métodos para permitir que operações de: adição aconteçam com os iteradores, fazendo com que apontem para posições posteriores, substração aconteçam com os iteradores, fazendo com que apontem para posições anteriores e comparação aconteçam com os iteradores (retornando falso ou verdadeiro).

Além disso, é utilizada a estrutura de dados bitset, a qual é possível ser acessada utilizando a biblioteca também chamada de “bitset”, para criar bytes, iterar sobre os bits dele e realizar operações neles.

2.2 Classes

O programa utiliza apenas de uma classe, a classe Arvore, a qual possui como atributos principais vários vectors4, um sendo responsável por armazenar os caracteres do arquivo, chamado de “folhas”, um vector4 chamado “copia”, que possui o mesmo conteúdo que “folhas”, mas são realizadas várias operações com seus elementos, um vector4 chamado “nós”, o qual armazena nós resultantes da combinação de outros nós, seguindo o algoritmo de Huffman, um vector4 chamado “tamanhos”, que armazena o tamanho da codificação em binário de cada caractere lido no arquivo e um vector4 chamado “dicionário”, o qual armazena objetos da estrutura de dados auxiliar Par, cada objeto tendo caractere e codificação em binário (em formato string) como atributos. Como alguns métodos menos importantes ela possui as funções “Heapsort”, “Refaz” e “Constroi”, as quais juntas ordenam um vetor de objetos TipoNo* recebido, com base em seu atributo frequência e a função “gerarNumeroUnico”, a qual serve para gerar identificadores para cada nó durante a construção da árvore de Huffman, “achaDescompactado”, a qual a partir de uma codificação em binário acha o caractere correspondente no vector4 “dicionário” e a função “acha”, a qual a partir de uma string caractere acha a codificação em binário correspondente no vector4 “dicionário” e, por fim, a função “Huff”, a qual caso haja só um tipo de caractere já cria o vector4 “dicionário” da árvore e, caso não haja, segue o procedimento normal de chamar a função “descobreMaior”.

2.3 Funções

O programa conta com 7 funções principais, parte sendo métodos da classe “Arvore” e parte são funções do arquivo “main.cpp”, as quais são as responsáveis, em conjunto com funções auxiliares, para atender diretamente aos requisitos do trabalho:

-“countCharacterFrequency”: Função que transformar cada caractere lido diferente dos que já foram lidos anteriormente em um TipoNo* e armazena em um vector4, alterando apenas seu atributo frequência caso um mesmo caractere apareça mais de uma vez.

-“ajustarNumeros”: Função que identifica nós cujo item seja um número e com o intuito de diferenciá-los de nós que tenham itens números também, na hora de montar a árvore de Huffman, acrescenta um caractere c, antes de cada número, no item deles.

-“achaarqEscondido”: Sem preciar de acessar nenhum método da classe Arvore, essa função já lê o arquivo binário passado como parâmetro e identifica qual o nome do arquivo original estava escrito em seu início e retorna o nome dele, possibilitando a reconstrução da árvore que foi usada para compacta-lo, no primeiro lugar.

-“descobreMaior”: Função que forma a árvore de Huffman a partir do vector4 “copia”, o qual possui nós com cada um dos caracteres do arquivo passado como parâmetro. Ela cria novos nós resultantes da junção das frequências de nós anteriores, seguindo o algoritmo de Huffman, até se chegar na raiz da árvore, o qual é salvo como atributo da classe Arvore pelo nome “maior”.

-“codificacao”: Função que junto com a função auxiliar “isFilho” (gera o código em binário de cada caractere, só que, ao contrário, o que é revertido depois) gera a codificação em binário de todos os caracteres lidos do arquivo passado como parâmetro e os associa em um objeto Par que armazena a string caractere e a codificação e coloca todos os objeto Par no vector4 “dicionário”.

-“compactarArquivo”: Função que abre o arquivo original para leitura e para cada caractere lido escreve sua codificação no arquivo em que foi passado como parâmetro para ser onde a mensagem codificada estaria. No fim, resulta em um arquivo com a mensagem do arquivo original compactada.

-“descompactarArquivo”: Função que abre o arquivo compactado, realiza a descompactação do arquivo compactado, lendo bit a bit por byte, a partir do bit correspondente ao tamanho de cada codificação binária, o que é acessado pelo vector4 tamanho e reescrevendo o caractere correspondente a cada codificação em um caractere que foi passado como parâmetro para conter a mensagem descompactada.

3.0 Análise de Complexidade

3.1 Complexidade quanto ao tempo

-“countCharacterFrequency”: Possui complexidade em relação ao tempo de O(n^2) pois realiza um loop que lê o arquivo char por char e, além disso, a cada iteração, verifica por meio de um loop se o char já havia sido detectado

-“ajustarNumeros”: Possui complexidade em relação ao tempo de O(n), pois realiza um loop que itera elemento a elemento de um vector4.

-“achaarqEscondido”: Possui complexidade de O(1), pois usa apenas a função “read”, mas o mesmo número de vezes para todos os casos.

-“descobreMaior”: Possui complexidade O(n^((log n)^2), pois o algoritmo em si possui apenas complexidade O(n^logn) pois a cada chamada retira-se dois elementos e adiciona-se um no lugar até sobrar só um elemento no vector4, mas isso também deve ser multiplicado pela complexidade do Heapsort, que é utilizado a cada chamada para reordenar os elementos do vector4.

-“codificacao”: Possui complexidade O(n), pois realiza dois loops, mas ambos separados.

-“compactarArquivo”: Possui complexidade O(n), pois lê o arquivo original char a char.

-“descompactarArquivo”: Possui também complexidade O(n), pois lê o arquivo compactado byte a byte.

3.2 Complexidade quanto ao espaço

Considerando como forma de medida para a complexidade quanto ao espaço, o número de alocações dinâmicas realizadas

-“countCharacterFrequency”: Possui complexidade em relação ao espaço O(n), com n sendo o número de caracteres diferentes presentes no texto, pois para cada caractere diferente cria-se um novo TipoNo*.

-“ajustarNumeros”: Não faz alocações dinâmicas, então O(1).

-“achaarqEscondido”: Possui complexidade de O(1), pois para todos os casos apenas uma alocação dinâmica é realizada.

-“descobreMaior”: Possui complexidade de O(n-1) aproximadamente, com esse sendo o número de nós criados , aproximadamente, por alocação dinâmica toda vez que se chama a função.

-“codificacao”: Não realiza alocações dinâmicas, então a complexidade em relação ao espaço é O(n).

-“compactarArquivo”: Não realiza alocações dinâmicas, então a complexidade em relação ao espaço é O(n).

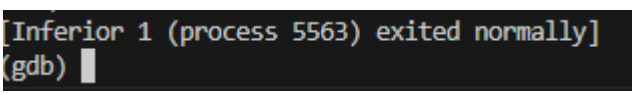
-“descompactarArquivo”: Realiza o mesmo número de alocações dinâmicas sempre, então complexidade O(1).

4.0 Estratégias de Robustez

Como uma das funções básicas do programa corresponde à leitura de arquivos, uma das principais estratégias de robustez aplicadas foi que, em relação à compactação, caso o arquivo que terá sua mensagem compactada não possa ser aberto, seja porque ele não existe no diretório atual, ou por qualquer outra razão, lança-se uma exceção que interrompe a execução do programa imediatamente para evitar resultados inesperados, indicando que houve um erro na abertura do arquivo. Isso também é percebido, em relação a descompactação, caso o arquivo que contém a mensagem compactada não exista ou não esteja no diretório atual, lança-se uma exceção que interrompe a execução do programa imediatamente para evitar resultados inesperados, indicando que houve um erro na abertura do arquivo. Ademais, no caso da compactação, caso o arquivo no qual deseja-se escrever a mensagem compactada não exista, é criado um arquivo com o nome passado como parâmetro. Isso também ocorre no caso da descompactação, só que em relação ao arquivo em que deseja-se escrever a mensagem descompactada.

Além disso, dentro da classe “vector” (estrutura de dados criada), há várias exceções implementadas, como antes de inserir um elemento em um vector conferir se ele não está cheio, antes de remover um elemento de um vector conferir se ele não está vazio, ao tentar acessar um elemento do vector usando os operadores “[]”, conferir se a posição existe mesmo e, por exemplo, na função erase conferir se o iterador passado não é inválido.

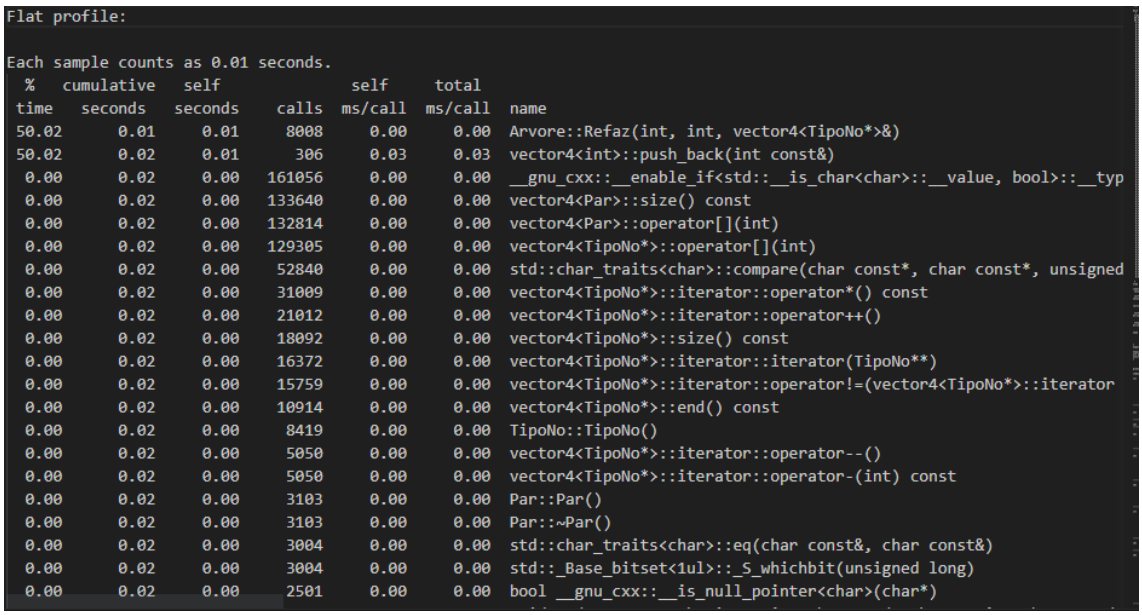
Ademais, a ferramenta de depuração gdb foi usada para conferir se não havia erros no programa:



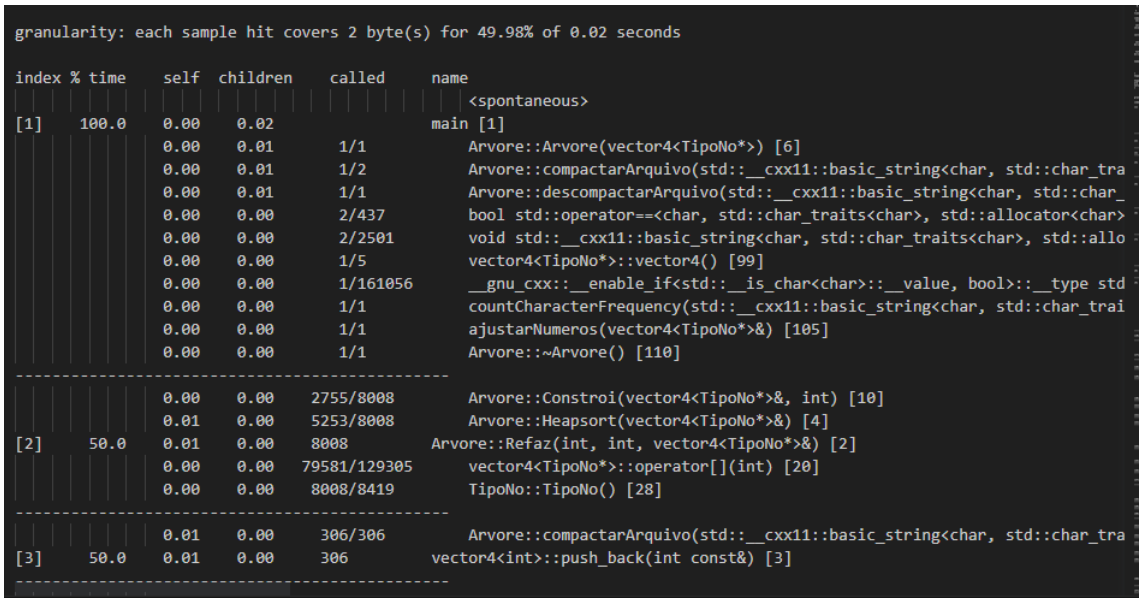
O que como pode-se perceber, indicou que não havia erros no programa que poderiam levar a algum segmentation fault ou algo do tipo, o que poderia prejudicar o programa, contribuindo para sua robustez.

5.0 Análise Experimental

Para a minha análise experimental, uma das ferramentas utilizadas, foi o “gprof”, e o relatório “flat profile”, gerado a partir dessa ferramenta.

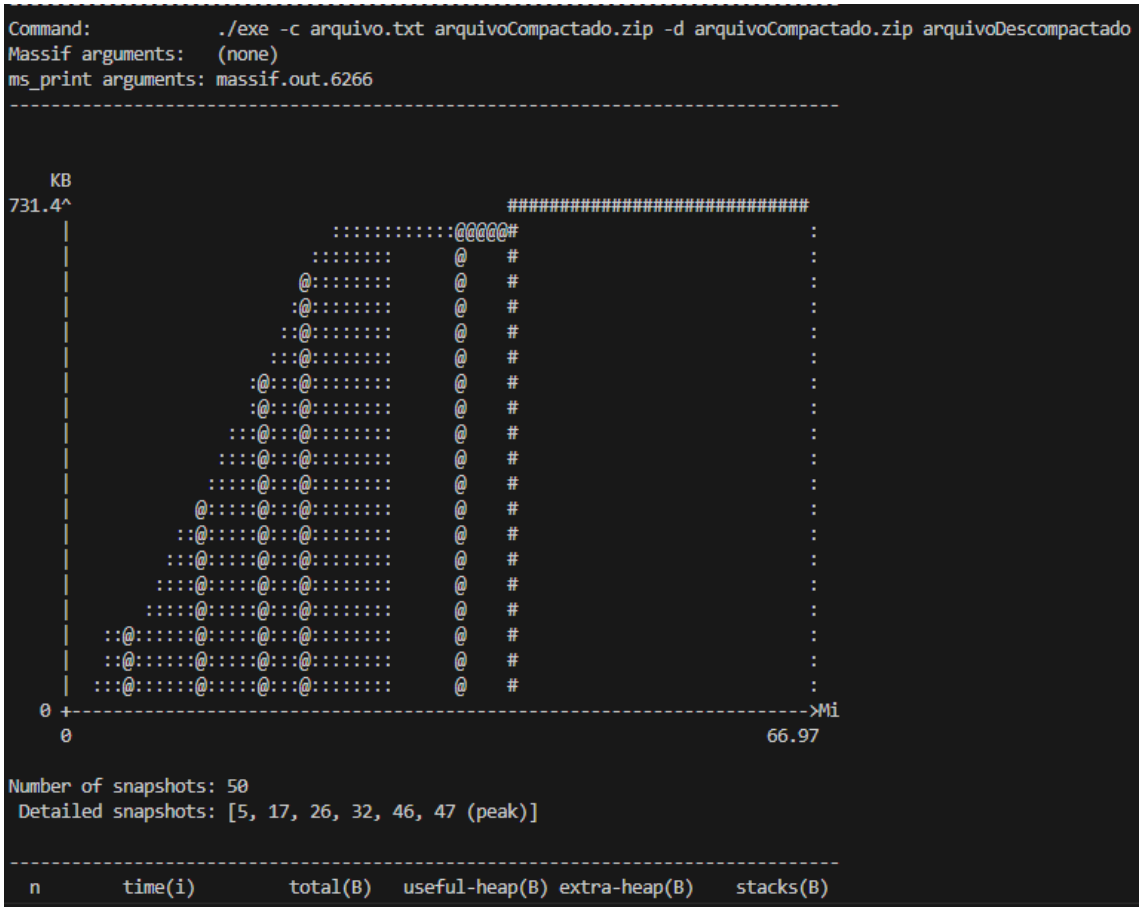


```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 50.02      0.01     0.01     8008     0.00     0.00  Arvore::Refaz(int, int, vector4<TipoNo*>&)
 50.02      0.02     0.01      306     0.03     0.03  vector4<int>::push_back(int const&)
  0.00      0.02     0.00   161056     0.00     0.00  __gnu_cxx::__enable_if<std::__is_char<char>::__value, bool>::__typ
  0.00      0.02     0.00   133640     0.00     0.00  vector4<Par>::size() const
  0.00      0.02     0.00   132814     0.00     0.00  vector4<Par>::operator[](int)
  0.00      0.02     0.00   129305     0.00     0.00  vector4<TipoNo*>::operator[](int)
  0.00      0.02     0.00    52840     0.00     0.00  std::char_traits<char>::compare(char const*, char const*, unsigned
  0.00      0.02     0.00    31009     0.00     0.00  vector4<TipoNo*>::iterator::operator*() const
  0.00      0.02     0.00    21012     0.00     0.00  vector4<TipoNo*>::iterator::operator++()
  0.00      0.02     0.00    18092     0.00     0.00  vector4<TipoNo*>::size() const
  0.00      0.02     0.00    16372     0.00     0.00  vector4<TipoNo*>::iterator::iterator(TipoNo**)
  0.00      0.02     0.00    15759     0.00     0.00  vector4<TipoNo*>::iterator::operator!=(vector4<TipoNo*>::iterator
  0.00      0.02     0.00    10914     0.00     0.00  vector4<TipoNo*>::end() const
  0.00      0.02     0.00     8419     0.00     0.00  TipoNo::TipoNo()
  0.00      0.02     0.00     5050     0.00     0.00  vector4<TipoNo*>::iterator::operator--()
  0.00      0.02     0.00     5050     0.00     0.00  vector4<TipoNo*>::iterator::operator-(int) const
  0.00      0.02     0.00     3103     0.00     0.00  Par::Par()
  0.00      0.02     0.00     3103     0.00     0.00  Par::~Par()
  0.00      0.02     0.00     3004     0.00     0.00  std::char_traits<char>::eq(char const&, char const&)
  0.00      0.02     0.00     3004     0.00     0.00  std::_Base_bitset<1ul>::_S_whichbit(unsigned long)
  0.00      0.02     0.00     2501     0.00     0.00  bool __gnu_cxx::__is_null_pointer<char>(char*)
```



```
granularity: each sample hit covers 2 byte(s) for 49.98% of 0.02 seconds

index % time    self  children    called     name
                                             <spontaneous>
[1]    100.0    0.00    0.02                 main [1]
                0.00    0.01       1/1           Arvore::Arvore(vector4<TipoNo*>) [6]
                0.00    0.01       1/2           Arvore::compactarArquivo(std::__cxx11::basic_string<char, std::char_tra
                0.00    0.01       1/1           Arvore::descompactarArquivo(std::__cxx11::basic_string<char, std::char_
                0.00    0.00       2/437         bool std::operator==<char, std::char_traits<char>, std::allocator<char>
                0.00    0.00       2/2501        void std::__cxx11::basic_string<char, std::char_traits<char>, std::allo
                0.00    0.00       1/5           vector4<TipoNo*>::vector4() [99]
                0.00    0.00       1/161056      __gnu_cxx::__enable_if<std::__is_char<char>::__value, bool>::__type std
                0.00    0.00       1/1           countCharacterFrequency(std::__cxx11::basic_string<char, std::char_trai
                0.00    0.00       1/1           ajustarNumeros(vector4<TipoNo*>&) [105]
                0.00    0.00       1/1           Arvore::~Arvore() [110]
-----------------------------------------------
                0.00    0.00    2755/8008        Arvore::Constroi(vector4<TipoNo*>&, int) [10]
                0.01    0.00    5253/8008        Arvore::Heapsort(vector4<TipoNo*>&) [4]
[2]     50.0    0.01    0.00    8008         Arvore::Refaz(int, int, vector4<TipoNo*>&) [2]
                0.00    0.00   79581/129305      vector4<TipoNo*>::operator[](int) [20]
                0.00    0.00    8008/8419        TipoNo::TipoNo() [28]
-----------------------------------------------
                0.01    0.00     306/306         Arvore::compactarArquivo(std::__cxx11::basic_string<char, std::char_tra
[3]     50.0    0.01    0.00     306         vector4<int>::push_back(int const&) [3]
-----------------------------------------------
```

Essas imagens mostram apenas trechos do relatório “flat profile” gerado pelo gprof (o relatório era muito grande e todos os tempos variavam eram ou 0.01 ou 0.04, mas o relatório completo vai estar no diretório raiz do programa). Esse resultado curiosamente mostra todos os tempos como entre 0.01 e 0.04, como a ferramenta foi utilizada corretamente, compilando-se o código utilizando a flag “-pg” e a geração do relatório também, uma das possíveis causas que explica esse curioso comportamento, que contrasta com o elevado número de chamadas de cada operação, presente na primeira foto é que as estruturas de dados usadas não possuem um grande espaço de armazenamento (3000) e que embora haja loops em algumas funções, a complexidade delas não ultrapassa O(n^2).



n	time(i)	total(B)	useful-heap(B)	extra-heap(B)	stacks(B)
0	0	0	0	0	0
1	2,376,490	42,584	42,570	14	0
2	2,940,750	127,536	127,050	486	0
3	3,433,643	129,264	128,586	678	0
4	4,110,021	130,920	130,058	862	0
5	5,015,363	131,640	130,698	942	0

```
99.28% (130,698B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->55.23% (72,704B) 0x48F8A99: ??? (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.28)
| ->55.23% (72,704B) 0x4011B99: call_init.part.0 (dl-init.c:72)
|   ->55.23% (72,704B) 0x4011CA0: call_init (dl-init.c:30)
|     ->55.23% (72,704B) 0x4011CA0: _dl_init (dl-init.c:119)
|       ->55.23% (72,704B) 0x4001139: ??? (in /usr/lib/x86_64-linux-gnu/ld-2.31.so)
|         ->55.23% (72,704B) 0x6: ???
|           ->55.23% (72,704B) 0x1FFEFFFF96: ???
|             ->55.23% (72,704B) 0x1FFEFFFF9C: ???
|               ->55.23% (72,704B) 0x1FFEFFFF9F: ???
|                 ->55.23% (72,704B) 0x1FFEFFFFAB: ???
|                   ->55.23% (72,704B) 0x1FFEFFFFC1: ???
|                     ->55.23% (72,704B) 0x1FFEFFFFC4: ???
|                       ->55.23% (72,704B) 0x1FFEFFFFDA: ???
|
```

Ademais, utilizou-se ferramentas de profiling, como o Valgrind Massif, para visualizar o uso da memória ao longo do tempo, como se vê acima, permitindo, por exemplo a visualização de número de acessos e acesso a endereços. Como pode-se perceber alguns dos destaques de número de acessos foram chamadas de alocação de heap

como “malloc”, “new” e “new[]” no heap e, grande parte do que acontece dentro do heap em funções não identificada ligadas à “Linux-gnu”, o que indica que essa alocação de memória está relacionada ao funcionamento interno da biblioteca padrão do c++.

6.0 Conclusões

Nesse trabalho, foi requisitado que fosse implementado um sistema de compactação baseado no algoritmo de Huffman. Essa proposta foi um excelente exercício para o treinamento na parte do raciocínio lógico e abstrato, pois não havia sido diretamente ensinado na aula como compactar e descompactar arquivos, o que fez com que esse assunto tivesse que ser pesquisado e aprendido. Além disso, foi muito útil para a revisão de conteúdos mais antigos do curso como alocação dinâmica de memória e elaboração de classes, possibilitando com que estruturas de dados mais avançadas como “vector” utilizado fossem implementadas manualmente.

Entretanto, é importante frisar que um dos aprendizados mais importantes deste trabalho, foi sobre a reafirmação da importância do raciocínio lógico e abstrato para a solução de problemas, tendo sido necessário pensar em soluções criativas para realizar certas operações matemáticas. Um dos exemplos mais perceptíveis foi quando estava buscando implementar a classe vector, quando tive que implementar uma estrutura de dados até então bem mais avançada do que as que estava acostumado (pilha, fila circular, etc), devido, principalmente aos iteradores e na implementação da estrura de dados árvore, a qual em geral é construída do nó raiz até as folhas, mas, nesse caso, teve que ser construída na direção constrária indo das folhas até a raiz.

Referências bibliográficas:

- (Thomas H. Cormen - Algoritmos: Teoria e Prática)
- Material disponibilizado em forma de slides pelo professor Wagner Meira Jr

Instruções para Compilação e Execução

1. Deve-se abrir o terminal.

2. Entrar na pasta TP, usando o comando “cd TP”.

3.Transferir os arquivos que deseja-se ou compactar ou descompactar para o diretório TP. (**Atenção! As operações de compactação e descompactação podem ser realizadas quando e na ordem que for desejado, porém caso queira realizar qualquer operação que seja em um arquivo, seja compactá-lo ou descompactar o arquivo com a mensagem compactada de outro arquivo, não delete o arquivo original, ele deve sempre estar no mesmo diretório que os demais para qualquer operação. Caso isso não seja obedecido, o programa retornará uma exceção**)

4. Utilizar o comando “make run ARGS=**“argumentos”**” para compilar e executar o código, o comando deve ser exatamente esse, com apenas nome, sendo substituído pelo conjunto de flags e nomes de arquivos desejados. **Exemplos do que substituir o termo argumentos:** “-c arquivo arquivoCompactado” (vai ler a mensagem de “arquivo” e colocar ela compactada no arquivo chamado “arquivoCompactado”), “-d arquivoCompactado arquivoDescompactado” (vai ler a mensagem de “arquivoCompactado” e colocar ela descompactada no arquivo chamado “arquivoDescompactado”) ou, por fim, “-c arquivo.txt arquivoCompactado.zip -d arquivoCompactado.zip arquivoDescompactado” (vai ler a mensagem de “arquivo.txt” e colocar ela compactada no arquivo chamado “arquivoCompactado.zip” e vai ler a mensagem de “arquivoCompactado.zip” e colocar ela descompactada no arquivo chamado “arquivoDescompactado”).

Atenção! O nome passado como parâmetro deve ser exatamente o nome do arquivo, incluindo a extensão, por exemplo, “entrada.txt” é algo que poderia ser colocado para substituir o termo nome no comando dito. Entretanto certifique-se de que o arquivo desejado esteja no diretório atual (exemplo: se a pasta se chama TP e tenha subpastas bin, obj, src e include, o arquivo deve estar na pasta TP, mas não nas subpastas) do projeto, mas nem em “bin”, nem em “include”, nem em “src” e nem em “obj”.