UFMG

Estruturas de Dados – Trabalho Prático 2

Otimização dos Servidores

Nome: Arthur Pontes Nader

Matrícula: 2019022294

Universidade Federal de Minas Gerais

Belo Horizonte - MG - Brasil

arthurpn@ufmg.br

1. Introdução

Após implementação do sistema capaz de fazer as transferências de consciências pela meganet (Trabalho Prático 1), surge um novo desafio para o sistema de Realocação Interplanetária: implementar uma nova medida de segurança que seja capaz de ordenar as consciências em seus servidores de tal forma a reduzir o tempo total de envio. Essa redução fará com que o tempo de visibilidade das consciências fique menor, dificultando assim os ataques realizados por hackers.

Uma das principais habilidades exigidas de um cientista da computação é a capacidade de ordenar um vetor de acordo com os requisitos exigidos de uma determinada aplicação, considerando assim diversos fatores, como tempo de execução e estabilidade. Para se fazer isso, existem diversos algoritmos conhecidos, tais como: Inserção, Bolha, Mergesort, Quicksort, Radixsort, Buckesort, etc. Cada um deles possui suas vantagens e suas desvantagens.

Assim, o principal objetivo desse trabalho foi a implementação de funções de ordenação de tal modo que organizasse as consciências a serem realocadas a partir de seus dados e de seus nomes, otimizando dessa forma os servidores do sistema de Realocação Interplanetária. Para implementação do programa, utilizou-se a linguagem C++.

A seção 2 desse relatório mostram como o programa foi estruturado e implementado. Já a seção 3 se refere à a complexidade de cada função criada, enquanto a 4 descreve as configurações experimentais usadas e os objetivos delas. A seção 5 apresenta os resultados obtidos, analisando a

estabilidade e mostrando o tempo de execução de cada uma das configurações utilizadas. Por fim, a seção 6 apresenta a conclusão do trabalho e a seção 7 enumera as referências bibliográficas consultadas. Ao final, há um apêndice explicando como devem ser feitas a compilação e a execução do programa desenvolvido.

2. Descrição da implementação

Para implementação do programa, criou-se um tipo abstrato de dado que representasse as consciências a serem realocadas. Os métodos de ordenação desenvolvidos foram adaptados dos algoritmos vistos em aula. São eles: Quicksort, Mergesort, Heapsort e Radixsort. A seguir, há uma breve descrição dos métodos implementados para o TAD desenvolvido, algumas observações sobre cada um das funções de ordenação, informações sobre a "main" e sobre a maneira como o código lida com entradas inválidas.

- **Classe Consciencia:**

  A classe Consciencia tem como principal papel o armazenamento dos dados e a instanciação das consciências a serem ordenadas. Cada uma dessas consciências possuirão duas strings: uma representará o nome da consciência e outra os dados binários dela. A classe possui os seguintes métodos:

  Consciencia(): construtor padrão de objetos da classe.

  Consciencia(std::string nomeCons, std::string dadosCons): construtor que cria uma consciência que tem como nome e dados binários os parâmetros passados na chamada do método.

  std::string getNome(): retorna o nome da consciência.

  std::string getDados(): retorna os dados binários da consciência.

  void mostrarConsciencia(): imprimi na saída padrão o nome e os dados binários da consciência.

- **Métodos do Quicksort:**
  void ParticionarQuick(int limEsq, int limDir, int* i, int* j, Consciencia* vet): essa função escolhe o elemento do meio do vetor como sendo o pivô. Em seguida, percorre o vetor com um índice que vai do começo ao fim e outro do final ao início. Isso é feito até que se ache um elemento menor e um maior que o pivô, sendo que quando isso acontecer eles serão trocados de posição. O processo para quando esses dois índices se cruzarem.

void QuickSort(Consciencia* vetor, int indiceEsq, int indiceDir): essa parte do método é responsável por chamar o método anterior novamente, só que dessa vez passando como limites os subvetores que ainda não foram ordenados.

- **Métodos do Mergesort:**

  void Merge(Consciencia* vetor, int indiceEsquerda, int indiceMeio, int indiceDireita): essa função combina dois vetores ordenados adicionando-se o menor de cada um deles a um novo vetor até que não reste nenhum elemento em um dos vetores. Quando isso acontecer, o restante do vetor não vazio é copiado para completar as posições que sobraram. Ao final, o novo vetor criado estará devidamente ordenado.

  void MergeSort(Consciencia* vet, int esq, int dir): a função é responsável por dividir o problema de ordenação até que reste apenas um elemento no vetor. Quando isso ocorrer, a função anterior será chamada para combinar os subvetores ordenados.

- **Métodos do Heapsort:**

  void RefazerHeap(int indiceEs, int indiceDir, Consciencia* vetor): essa função é responsável por transformar o vetor a ser ordenado em uma representação de arvore binária em que cada nó localizado em um índice x, será maior que seus filhos localizados nas posições 2x e 2x+1. Ao final do processo, percebe-se que o maior elemento estará na primeira posição do vetor.

  void HeapSort(Consciencia* vet, int tamanho): depois que o vetor estiver organizado na forma de um heap, troca-se o primeiro elemento com o último. Em seguida, a função para refazer o heap é chamada novamente para o vetor desconsiderando a última posição. Assim, na próxima iteração, o maior elemento dessa parte do vetor estará novamente na primeira posição.

- **Métodos do Radixsort**

  void RadixSort(Consciencia* vetor, int indiceEsquerda, int indiceDireita, int totalBits, int bitAnalisado): esse método de ordenação é similar ao Quicksort, entretanto, a comparação é feita entre os bits das cadeias, analisando-os da esquerda para a direita. Na chamada recursiva, passa-se os índices dos subvetores ainda não

ordenados aumentando-se o índice do bit que deve ser analisado em uma unidade. Isso é feito até que se atinja o total de bits.

- **Main:**

A abertura do arquivo a ser lido ocorrerá na “main” do programa. A instanciação das consciências a serem ordenadas ocorre por meio de um laço “for” que lê e formata as linhas do arquivo até que se atinja a quantidade passada como parâmetro. Essa formatação é responsável por gerar os parâmetros de instanciação de um objeto da classe Consciencia. Em seguida, é executada as funções de ordenação referentes à configuração passada como parâmetro. Utiliza-se um laço “for” para impressão do vetor devidamente ordenado. Por fim, o vetor é desalocado, fecha-se o arquivo e o programa é encerrado.

- **Entradas imprevistas:**

Parâmetros passados que fazem referência a configurações experimentais não implementadas simplesmente farão com que o vetor não seja ordenado. Entradas de dados da consciência que não sigam o padrão estabelecido e tentativas de acesso a posições não inicializadas do vetor poderão ocasionar erros de execução. O arquivo de leitura deve conter pelo menos a quantidade de consciências passadas como parâmetro para inicialização do vetor.

3. Complexidade de espaço e de tempo

Para realização da análise assintótica de cada método, define-se uma variável n que representará o número de consciências instanciadas a partir da leitura do arquivo, ou seja, o tamanho do vetor a ser ordenado.

- **Métodos da Classe Consciencia:**

Consciencia():

Complexidade de tempo: construtor padrão apenas cria um objeto do tipo Consciencia, portanto, O(1).

Complexidade de espaço: criar um objeto dessa classe tem custo de espaço constante, ou seja, O(1).

Consciencia(std::string nomeDaConsciencia, std::string DadosDaConsciencia):

Complexidade de tempo: esse construtor somente atribui os parâmetros passados ao objeto criado, por isso, esse método é O(1).

Complexidade de espaço: O(1), já que é criado apenas um objeto da classe Consciencia.

std::string getNome():

Complexidade de tempo: apenas retorna o atributo “nomes”, portanto, esse método é O(1).

Complexidade de espaço: nenhuma espaço de memória extra é necessário, portanto, o método é O(1).

std::string getDados():

Complexidade de tempo: como anteriormente, retornar uma string tem complexidade de tempo O(1).

Complexidade de espaço: O(1), uma vez que nenhuma memória extra é necessária para execução.

void mostrarConsciencia()

Complexidade de tempo: esse método apenas imprimi duas strings separadas por um espaço, logo, ele é O(1).

Complexidade de espaço: nenhum espaço de memória a mais é necessário durante a execução, por isso, esse procedimento é O(1).

- **Métodos do Quicksort:**

void ParticionarQuick(int limEsq, int limDir, int* i, int* j, Consciencia* vet):

Complexidade de tempo: as operações de comparação, em uma execução dessa função, percorrem o vetor inteiro, por isso essa função é O(n) para a complexidade de tempo.

Complexidade de espaço: só é necessário espaço extra para guardar as informações do pivô e para auxiliar na troca de posições, portanto, a função é O(1).

void QuickSort(Consciencia* vetor, int indiceEsq, int indiceDir):

Complexidade de tempo: essa função divide sucessivamente o vetor após cada execução, por isso essa parte é O(log n). Como cada uma dessas partes chama a função de partição, a função será O(n log n) no caso médio. O pior caso possível é $O(n^2)$, entretanto, ele dificilmente ocorre.

Complexidade de espaço: a pilha de execução guarda cada instanciação da função em que se avaliará a necessidade de ordenar um vetor menor. Por isso, a complexidade de espaço é O(log n).

- **Métodos do Mergesort:**

void Merge(Consciencia* vetor, int indiceEsquerda, int indiceMeio, int indiceDireita):

Complexidade de tempo: a função percorre cada um dos subvetores passados selecionando o menor elemento de cada um deles. Como isso é feito até que não reste mais elementos, a função é O(n).

Complexidade de espaço: um vetor auxiliar é necessário para guardar a ordenação gerada pela combinação das partes do vetor. Por isso, esse método é O(n) para a complexidade de espaço.

void MergeSort(Consciencia* vet, int esq, int dir

Complexidade de tempo: esse algoritmo é do tipo divisão e conquista, já que ele divide o vetor sucessivamente até atingir o caso base. São feitas “log n” divisões, sendo que cada uma delas irá chamar a função Merge para ordenação. Portanto, Mergesort é O(n log n).

Complexidade de espaço: similar à complexidade de tempo, temos “log n” chamadas da função Merge. Assim, o método é O(n log n) para a complexidade de espaço.

- **Métodos do Heapsort:**

void RefazerHeap(int indiceEs, int indiceDir, Consciencia* vetor):

Complexidade de tempo: para refazer o heap, temos que o pior caso ocorrerá quanto for necessário percorrer todo um galho da árvore. Portanto, temos que a função é O(log n).

Complexidade de espaço: é necessário espaço extra somente para auxiliar na troca de posições, assim, tem-se que a função é O(1) para complexidade de espaço.

void HeapSort(Consciencia* vet, int tamanho):

Complexidade de tempo: para realizar a ordenação, a função refaz o heap para cada elemento presente no vetor. Assim, ela é O(n log n).

Complexidade de espaço: como o algoritmo não é do tipo divisão e conquista e a chamada para refazer o heap é O(1) para a complexidade de espaço, tem-se que Heapsort também será O(1).

- **Métodos do Radixsort:**

void RadixSort(Consciencia* vetor, int indiceEsquerda, int indiceDireita, int totalBits, int bitAnalisado):

Complexidade de tempo: o custo para percorrer todo o vetor realizando as trocas necessárias é O(n). Como isso é executado “log n” vezes, tem-se que a complexidade de tempo do RadixSort será O(n log n).

Complexidade de espaço: a pilha de execução guarda cada instanciação da função para posterior ordenação da segunda metade do vetor ou subvetor. Por isso, a complexidade de espaço é O(log n).

- **Main:**

Complexidade de tempo: o programa principal pode chamar cada uma das configurações de ordenação criadas. Portanto, sua complexidade de tempo será limitada pela configuração de maior custo computacional. Assim, o programa principal é O(n log n), dado que o pior caso do Quicksort, que é $O(n^2)$, dificilmente ocorre.

Complexidade de espaço: o programa principal pode chamar o método Mergesort, que possui a maior complexidade de espaço. Por isso, ele será O(n log n).

4. Configurações experimentais

A seguir, há uma breve descrição de cada uma das configurações usadas para realizar a ordenação desejada.

- **Configuração 1:**
  Heapsort para ordenar “Dados” e Quicksort para ordenar “Nome”.

- **Configuração 2:**
  Radixsort para ordenar “Dados” e Quicksort para ordenar “Nome”

- **Configuração 3:**
  Heapsort para ordenar “Dados” e Mergesort para ordenar “Nome”.

- **Configuração 4:**
  Radixsort para ordenar “Dados” e Mergesort para ordenar “Nome”.

Os experimentos realizados consistiram em aplicar cada uma dessas configurações de ordenação a um determinado valor de entrada que indicaria o tamanho do vetor a ser ordenado, sendo que esse valor poderia ser 1000, 10000, 50000, 100000 ou 200000.

Assim, o principal objetivo de cada uma dessas combinações de configuração e tamanho da entrada foi determinar e analisar o tempo gasto para ordenação do vetor. Outro fator que deveria ser observado foi o modo como a estabilidade ou não de cada um dos métodos que compõem a configuração afetam a maneira como os dados são organizados ao final do processo.

5. Resultados

- **Estabilidade**

A implementação das funções de ordenação ocorreu de tal forma que Quicksort, Heapsort e Radixsort não são estáveis. Assim, o único método estável implementado foi o Mergesort.

Então, é possível perceber que as configurações 1 e 2 fazem com que as consciências fiquem ordenadas somente pela ordem alfabética devido ao método Quicksort, que ordena pelo atributo “nome”, não ser estável. Portanto, consciências que possuem o mesmo nome ficam desordenadas quando se considera o atributo “dados”.

Como o atributo “dados” é ordenado primeiro que “nome”, isso faz com que, apesar de possuírem métodos não estáveis para ordenação de “dados” (Heapsort e Radixsort), as configurações 3 e 4 produzem um vetor em que, além das consciências estarem em ordem alfabética, consciências com mesmo nome fiquem ordenadas devidamente quando se considera o atributo “dados”. Isso ocorre devido ao fato de que, por ser estável, o Mergesort não altera a primeira ordenação realizada para os dados das consciências.

O único problema possível no caso das ordenações geradas pelas configurações 3 e 4 ocorreria se houvessem duas consciências com mesmo “nome” e “dados binários”. Assim, a execução poderia fazer com que a consciência que inicialmente aparece mais ao final do vetor apareça à frente da que estava mais ao início depois da ordenação.

- **Tempo de execução**

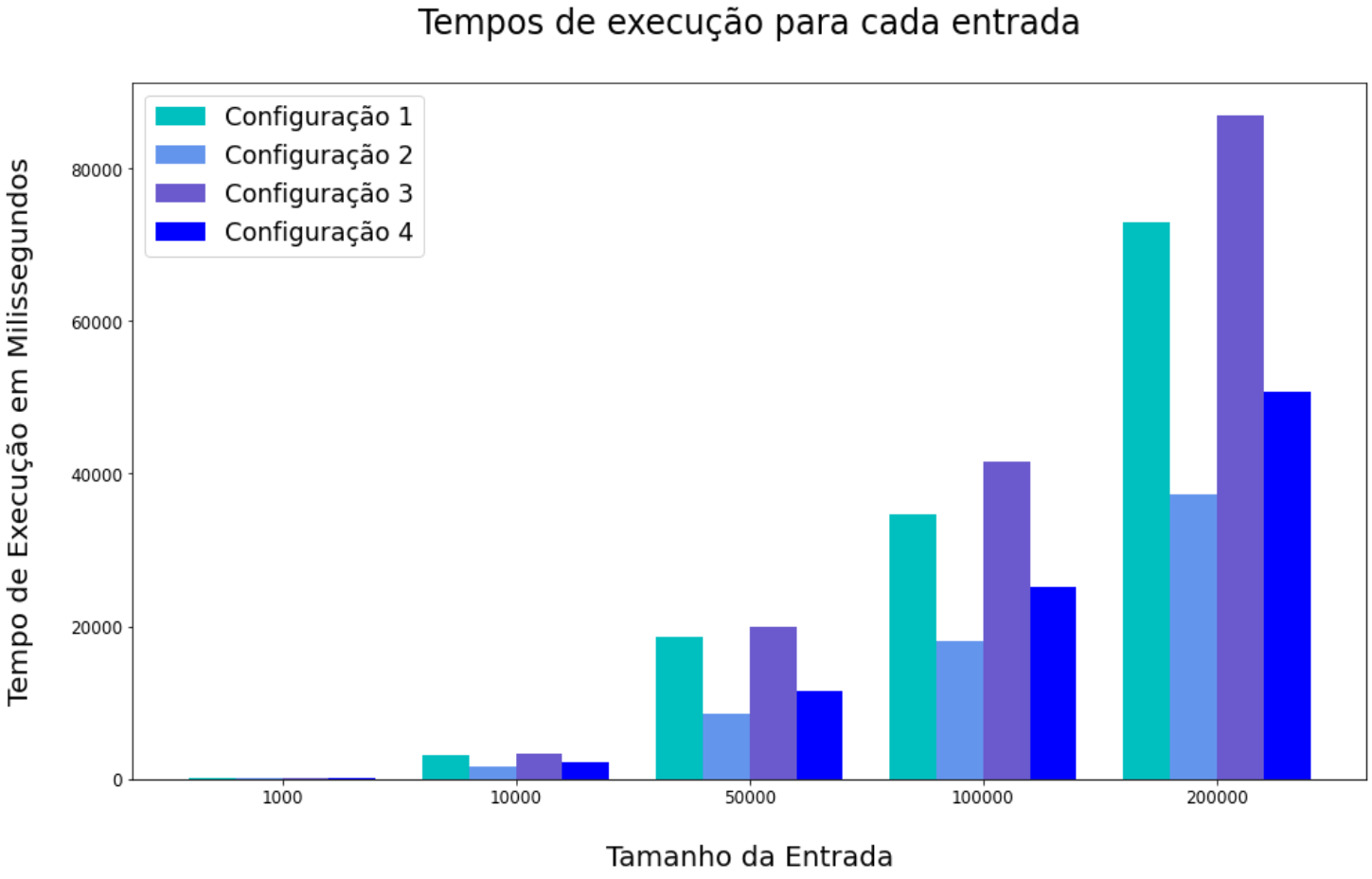
Para obtenção dos tempos de execução, utilizou-se um computador com um processador Ryzen 3750H e que possui 8GB de memória RAM.

A medição de tempo ocorreu usando-se a biblioteca “chrono” da própria linguagem C++. Apesar disso, o arquivo enviado para correção não inclui essa biblioteca para evitar possíveis erros de compilação e execução na avaliação dos casos de teste. A unidade de medida usada foi milissegundos, devido ao fato de que entradas pequenas executam em menos de 1 segundo.

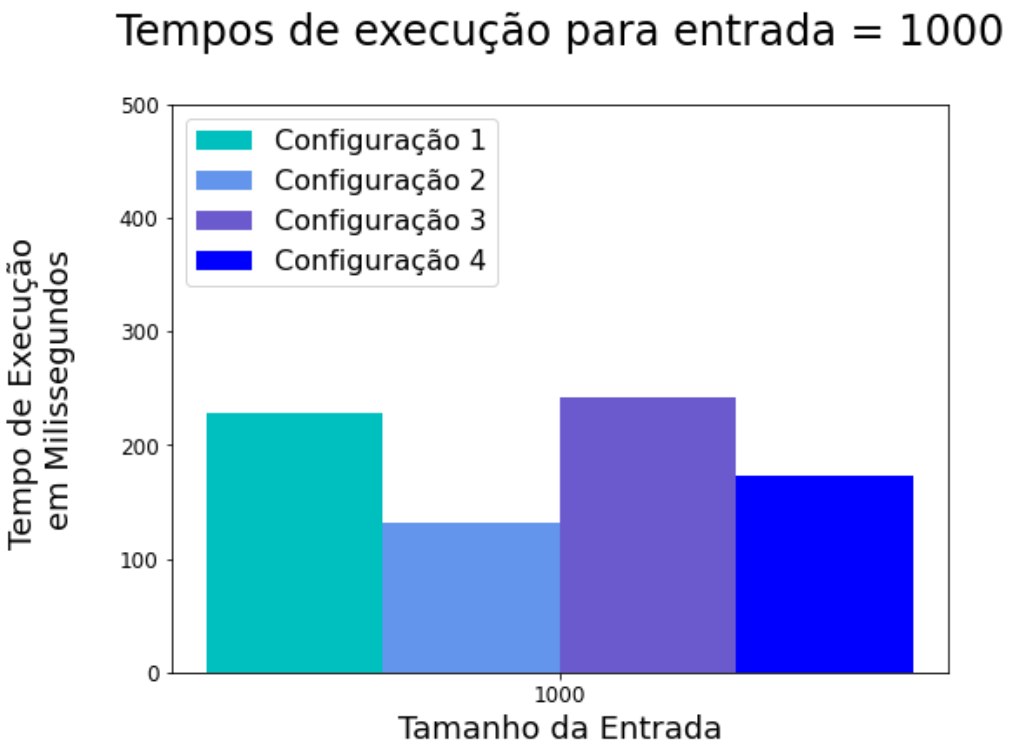
A tabela a seguir apresenta os valores de tempo em milissegundos obtidos para cada entrada e configuração:

Tamanho da entrada	Configuração 1	Configuração 2	Configuração 3	Configuração 4
1000	228	131	242	173
10000	3168	1695	3351	2201
50000	18600	8502	19946	11583
100000	34627	17997	41571	25076
200000	72857	37256	86957	50807

Utilizando a linguagem de programação Python (e algumas de suas bibliotecas, tais como NumPy e Matplotlib), gerou-se um gráfico em barras a partir dos valores obtidos.



Como os valores para entrada igual a 1000 ficaram pouco visíveis nesse gráfico, fez-se um gráfico mostrando apenas os tempos para esse valor de entrada. Tal gráfico pode ser observado a seguir:



6. Conclusão

O trabalho prático realizado consistiu na implementação de funções de ordenação com intuito de otimizar os servidores do sistema de Realocação Interplanetária.

Por meio da análise dos resultados obtidos, percebe-se que a configuração de ordenação que teve melhor tempo de execução foi a configuração 2. Entretanto, por conter uma função não estável para ordenação dos nomes, essa configuração acaba por deixar consciências de mesmo nome desordenadas quanto ao atributo “dados”.

Considerando-se a organização final do vetor, a configuração 4 é a mais adequada, pois preserva a ordenação do atributo “dados” para consciências de mesmo nome, o que ocorre devido ao método Mergesort implementado ser estável. Além disso, os resultados obtidos indicam que ela foi mais rápida que a configuração 3 independentemente do valor do tamanho da entrada.

O desenvolvimento de cada uma das funções de ordenação ocorreu sem muitos problemas de compilação e execução, sendo que os casos de testes

realizados, considerando a estabilidade de cada configuração de ordenação, produziram as saídas esperadas quando executados pelo código produzido.

O maior desafio encontrado foi na implementação do Radixsort, pois o algoritmo visto em aula tinha um caso de parada diferente do que se parecia ser necessário para solução do problema especificado. Entretanto, isso foi resolvido adicionando-se ao método mais um parâmetro que indicará o número total de bits a serem considerados pela ordenação, sendo possível assim executar o código até que esse último bit seja alcançado.

Por fim, a realização do trabalho prático possibilitou uma boa assimilação dos conceitos de ordenação vistos em aula. Entender como funciona cada método implementado e suas principais vantagens e desvantagens foi essencial na hora de analisar os resultados obtidos.

7. Bibliografia

- PRATES, R., CHAIMOWICZ, L. **Ordenação: MergeSort.** Departamento de Ciência da Computação, UFMG, 2020. Apresentação em slide. 38 slides.
- PRATES, R., CHAIMOWICZ, L. **Ordenação: Quicksort.** Departamento de Ciência da Computação, UFMG, 2020. Apresentação em slide. 58 slides.
- PRATES, R., CHAIMOWICZ, L. **Ordenação: Heapsort.** Departamento de Ciência da Computação, UFMG, 2020. Apresentação em slide. 60 slides.
- PRATES, R., CHAIMOWICZ, L. **Métodos de Ordenação sem comparação de chaves.** Departamento de Ciência da Computação, UFMG, 2020. Apresentação em slide. 50 slides.
- ZIVIANI, N**. Projeto de Algoritmos com Implementações em Pascal e C**. 3ª edição, Cengage Learning, 2011.
- CORMEN, T., LEISERSON, C., RIVEST, R., STEIN, C**. Introduction to Algorithms**. Third Edition, MIT Press, 2009.
- DROZDEK, A. **Data Structures and Algorithms in C++.** Fourth Edition, Cengage Learning, 2013.

Apêndice - Compilação e execução

Para compilação e execução do programa desenvolvido, utilizou-se um computador com 4 GB de memória RAM e que possui como processador um Intel i3 de terceira geração. O sistema operacional usado durante o processo foi o Linux Mint.

Inicialmente, deve-se adicionar os arquivos “.txt” a serem lidos pelo programa na raiz do projeto. Para compilar o programa desenvolvido, é preciso acessar o diretório do programa e digitar “make” no terminal de comando. Aparecerá no terminal que os arquivos do projeto foram compilados com sucesso.

Para execução do código, deve-se digitar no terminal “./bin/run.out ” seguido do nome do arquivo, a configuração de ordenação desejada e o número de linhas do arquivo que deverão ser consideradas para a ordenação. Esses parâmetros devem estar separados um do outro por meio de espaços (“ “). O arquivo lido deve conter os dados das consciências a serem colocadas em ordem pelo programa. Ao final disso, será mostrado no terminal a ordem final das consciências de acordo com uma configuração escolhida pelo usuário.

Para finalizar o processo de modo correto, digite “make clean” no terminal para limpar os arquivos executáveis gerados.