Documentação Trabalho Prático 1 da Disciplina de Estrutura de Dados

Daniel Oliveira Barbosa
Matrícula: 2020006450

Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brazil

danielolibar@gmail.com

1. Introdução

Esta documentação lida com a implementação de um simulador de servidor de emails que suporte as operações de cadastrar um usuário, remover um usuário, entregar um email (de um usuário para outro) e consultar um email. Para implementar esse simulador, será necessário:

- Criar classes que modelam um email, uma entrada, uma caixa de entradas, uma conta de usuário, um servidor de emails e um leitor de arquivos de comandos.
- Criar estruturas de dados que armazene dinamicamente as entradas dentro da classe que modela a caixa de entrada e as contas de usuários dentro da classe que modela o servidor de emails.
- Criar comandos no servidor para cadastrar e remover usuários, entregar um email para uma caixa de entrada e consultar um email de uma caixa de entrada.

2. Implementação

Modelamos o simulador da seguinte maneira: um servidor de emails contém uma lista encadeada de contas de usuários (células da lista). Uma conta de usuário contém um ID, uma caixa de entrada e um ponteiro para a próxima conta de usuário. Uma caixa de entrada contém uma fila de prioridade encadeada de entradas (células da fila). Uma entrada contém um email e um ponteiro para a próxima entrada.
Escolhemos a estrutura de lista encadeada para armazenar contas de usuário devido a um número variável de contas e menor custo de remoção quando comparado a um vetor alocado dinamicamente. Abreviaremos o nome “Lista Encadeada de Contas de Usuários" como LECU daqui pra frente. Similarmente, escolhemos a estrutura de fila de prioridade encadeada para armazenar as entradas devido a um número variável de entradas a natureza “First In First Out” de prioridade da caixa de entradas (inserções no final da fila, respeitando a ordem prioridade, e remoções no início da fila). Abreviaremos o nome “Fila de Prioridade Encadeada de Entradas” como FPEE daqui pra frente. Apresentaremos nos tópicos a seguir as classes utilizadas no projeto, juntamente com seus atributos e métodos.

- **Email:**
    a. **Atributos:**
        i. **prioridade** : int (armazena a prioridade do email)
        ii. **mensagem** : string (armazena a mensagem do email)
    b. **Métodos:**

i. **get_____()** (retorna o atributo _____)
ii. **Email()** (constrói um email)
iii. OBS.: Há dois construtores, um que recebe parâmetros e outro que não recebe.

- **Entrada** (Classe amiga da classe CaixaDeEntrada):
  a. **Atributos:**
    i. **email** : Email (armazena a o email da entrada)
    ii. **proximaEntrada**: Entrada* (armazena um ponteiro para a próxima entrada da caixa de entradas)
  b. **Métodos:**
    i. **Entrada()** (constrói uma entrada)
- **CaixaDeEntrada:**
  a. **Atributos:**
    i. **primeiraEntrada** : Entrada* (armazena um ponteiro para a primeira entrada da caixa de entradas - *i.e.* FPEE)
    ii. **ultimaEntrada** : Entrada* (armazena um ponteiro para a última entrada da caixa de entradas - *i.e.* FPEE)
  b. **Métodos:**
    i. **CaixaDeEntrada()** (constrói uma caixa de entrada)
    ii. **~CaixaDeEntrada()** (destrói uma caixa de entrada)
    iii. **adicionarEmail()** (adiciona uma entrada que contém um email, na caixa de entradas segundo a ordem de prioridade, ou seja, enfileira uma entrada, que contém um email, na FPEE segundo a ordem de prioridade. Imprime mensagem de sucesso caso consiga, caso contrário imprime mensagem de erro)
    iv. **removerEmail()** (remove a primeira entrada, que contém um email, na caixa de entradas, ou seja, desenfileira a primeira entrada, que contém um email, na FPEE. Imprime mensagem de sucesso e retorna true caso consiga, caso contrário imprime mensagem de erro e retorna false)
    v. **limpar()** (remove todas as entradas da caixa de entrada)
    vi. **getMensagemPrimeiroEmail()** (retorna a mensagem do email da primeira entrada)
    vii. **isVazia()** (retorna true se a caixa de entrada estiver vazia ou false caso contrário)
    viii. **inserirEmailNoInicio()** (método privado que auxilia o método **adicionarEmail()** a adicionar uma entrada que contém um email com uma prioridade maior que todos os outros emails no início da caixa de entrada - *i.e.* no início da FPEE)
    ix. **inserirEmailNoFinal()** (método privado que auxilia o método **adicionarEmail()** a adicionar uma entrada que contém um email com uma

prioridade menor que todos os outros emails no final da caixa de entrada - *i.e.* no final da FPEE)

x. **inserirEmailNoMeio()** (método privado que auxilia o método **adicionarEmail()** a adicionar uma entrada que contém um email com uma prioridade menor que todos os outros emails em algum lugar no meio da caixa de entrada segundo a ordem de prioridade - *i.e.* em algum lugar da FPEE segundo a ordem de prioridade)

xi. **inserirEmailEmCaixaVazia()** (método privado que auxilia o método **adicionarEmail()** a adicionar uma entrada que contém um email em uma caixa de entrada vazia - *i.e.* uma FPEE vazia)

- **ContaDeUsuário** (Classe amiga da classe ServidorDeEmails):
  a. **Atributos:**
    i. **ID**: int (armazena o ID do usuário)
    ii. **caixaDeEntrada**: CaixaDeEntrada (armazena a caixa de entrada da conta - *i.e.* a FPEE da conta)
    iii. **proximaContaDeUsuario**: ContaDeUsuario* (armazena um ponteiro para a próxima conta de usuário do servidor de emails - *i.e.* da LECU)
  b. **Métodos:**
    i. **ContaDeUsuario()** (constrói uma conta de usuário)
- **ServidorDeEmails:**
  a. **Atributos:**
    i. **primeiraContaDeUsuario** : ContaDeUsuario* (armazena um ponteiro para a primeira conta de usuário do servidor de emails - *i.e.* da LECU)
    ii. **ultimaContaDeUsuario** : ContaDeUsuario* (armazena um ponteiro para a última conta de usuário do servidor de emails - *i.e.* da LECU)
  b. **Métodos:**
    i. **ServidorDeEmails()** (constrói servidor de emails)
    ii. ~**ServidorDeEmails()** (destrói servidor de emails)
    iii. **isContaExistente()** (verifica se existe uma conta no servidor de emails - *i.e.* LECU - com ID igual o ID passado como parâmetro. Se existir, retorna um ponteiro para essa conta, caso contrário, retorna nullptr)
    iv. **cadastrarConta()** (adiciona uma nova conta de usuário no servidor de emails - *i.e.* LECU -, identificando essa conta de usuário com o ID passado como parâmetro. Caso consiga cadastrar corretamente, imprime mensagem de sucesso, caso contrário, imprime mensagem de erro)
    v. **removerConta()** (remove a conta de usuário identificada pelo ID passado como parâmetro do servidor de emails - *i.e.* LECU - juntamente com todas as entradas - e consequentemente emails - da caixa de entrada do usuário dessa conta. Caso consiga remover corretamente, imprime mensagem de sucesso, caso contrário, imprime mensagem de erro)

vi. **entregarEmail()** (adiciona uma entrada com um novo email na caixa de entrada - *i.e.* FPEE - da conta identificada pelo ID passado como parâmetro. Caso consiga entregar corretamente, imprime mensagem de sucesso, caso contrário, imprime mensagem de erro)

vii. **consultarCaixa()** (imprime a mensagem do email da primeira entrada da caixa de entrada - *i.e.* FPEE - da conta identificada pelo ID passado como parâmetro. Caso consiga consultar corretamente, imprime mensagem de sucesso, caso contrário, imprime mensagem de erro)

- **ArquivoDeComandos:**
  a. **Atributos:**
     i. **arquivoDeComandos**: fstream (armazena um arquivo com comandos para o simulador de servidor de emails)
  b. **Métodos:**
     i. **ler()** (abre o arquivo de comandos, inicializa o servidor e lê os comandos do arquivo, executando esses comandos no servidor simultaneamente)
     ii. **lerMensagem()** (método privado que auxilia o método **ler()** a ler a mensagem de um comando de entrega de email, retornando um string com a mensagem sem a palavra reservada "FIM")
     iii. **isStringUmNumero()** (método privado que auxilia o método **ler()** a verificar se o ID e a prioridade do email passados no arquivo de comandos são realmente números inteiros, retornando true caso seja ou false caso contrário)

Devido a uma boa modularização, o programa principal conta com apenas

- A captura do arquivo de comandos do terminal por argv[]
- A instanciação desse arquivo de comandos com o construtor **ArquivoDeComandos(argv[1])**
- A leitura do arquivo de comandos instanciado no passo anterior pela chamada do método **ler()**.

O programa foi desenvolvido utilizando a linguagem C++, compilador G++ da GNU Compiler Collection e sistema operacional Ubuntu (versão 2204.1.6.0) através do Windows Subsystem for Linux 2 (a máquina utiliza como sistema operacional padrão o Windows 10 Pro 10.0.19043 Build 19043).

3. Análise de complexidade

As operações relevantes do programa são as de inserção, remoção e caminhamento das estruturas de dados escolhidas (listas encadeadas e filas de prioridade encadeadas) e serão elas que usaremos para analisar a complexidade de tempo.
OBS-1: os métodos: **isVazia()** da classe **ServidorDeEmails**, **lerMensagem()** e **isStringUmNumero()** da classe **ArquivoDeComandos** e os **get___()** e construtores das classes em geral, por não contemplarem as operações relevantes (inserção, remoção e caminhamento das estruturas de dados escolhidas), não serão contemplados na tabela abaixo.

Nome do Método	O que o Método Faz	Complexidade de Tempo	Complexidade de Espaço
inserirEmailNoInicio(), inserirEmailNoFinal() e inserirEmailEmCaixaVazia( ) da classe CaixaDeEntrada	Métodos auxiliares que fazem a inserção de uma nova entrada (que contém um email) na caixa de entrada (que é uma FPEE) de acordo com a ordem de prioridade no início da caixa, no final da caixa, ou em uma caixa vazia, respectivamente.	O(1), pois a inserção de uma célula (entrada) em uma fila de prioridade encadeada (caixa de entrada) no início, no fim ou em uma fila de prioridade encadeada vazia é sempre O(1).	O(**n**), sendo **n** o número de células (entradas) da fila de prioridade encadeada (caixa de entrada).
inserirEmailNoMeio() da classe CaixaDeEntrada	Método auxiliar que faz a inserção de uma nova entrada (que contém um email) na caixa de entrada (que é uma FPEE) de acordo com a ordem de prioridade no meio da caixa de entrada não vazia.	O(**n**), sendo **n** o número de entradas da caixa de entradas. Isso se dá pois o pior caso de inserção de uma célula (entrada) no meio de uma fila de prioridade encadeada (caixa de entrada) é a inserção feita uma posição antes da última célula. Para ”caminhar” até essa posição há complexidade O(**n**).	O(**n**), sendo **n** o número de células (entradas) da fila de prioridade encadeada (caixa de entrada).
adicionarEmail() da classe CaixaDeEntrada	Faz a inserção de uma nova entrada (que contém um email) na caixa de entrada (que é uma FPEE) de acordo com a ordem de prioridade com o auxílio dos métodos inserirEmailNoInicio() , inserirEmailNoFinal(), inserirEmailEmCaixa Vazia() e inserirEmailNoMeio().	O(**n**), sendo **n** o número de entradas da caixa de entradas. Isso se dá pois o pior caso de inserção de uma célula (entrada) em uma fila de prioridade encadeada (caixa de entrada) é a inserção feita uma posição antes da última célula, que é feito através do método auxiliar inserirEmailNoMeio() a qual já havíamos determinado que possui complexidade O(**n**). Todos	O(**n**), sendo **n** o número de células (entradas) da fila de prioridade encadeada (caixa de entrada).

		os outros casos de inserção (início, fim e em caixa vazia) contemplam o melhor caso, que tem complexidade O(1).	
removerEmail() da classe CaixaDeEntrada	Faz a remoção da primeira entrada (que contém um email) da caixa de entrada (que é uma FPEE).	O(1), pois a remoção da primeira célula (entrada) em uma fila de prioridade encadeada é sempre O(1).	O(**n**), sendo **n** o número de células (entradas) da fila de prioridade encadeada (caixa de entrada).
limpar() da classe CaixaDeEntrada	Faz a remoção de todas as entradas (as quais contém um email) da caixa de entrada (que é uma FPEE).	O(**n**), sendo **n** o número de entradas da caixa de entradas. Isso se dá pois o método limpar() chama o método removerEmail(), que tem complexidade O(1), repetidamente até que todas as entradas tenham sido deletadas. Ou seja, removerEmail() é chamado **n** vezes. Logo limpar() possui complexidade O(**n**).	O(**n**), sendo **n** o número de células (entradas) da fila de prioridade encadeada (caixa de entrada).
isContaExistente() da classe ServidorDeEmails	Método auxiliar que busca uma conta de usuário com ID igual ao ID passado como parâmetro e retorna um ponteiro para essa conta.	O(**n**), sendo **n** o número de contas de usuários do servidor de emails. Isso se dá pois o pior caso de busca de uma célula (conta) em uma lista encadeada (servidor de emails) é se essa célula for a penúltima célula da lista. Para ”caminhar” até essa célula há complexidade O(**n**).	O(**n**), sendo **n** o número de células (contas de usuários) da lista encadeada (servidor de Emails).
cadastrarConta() da classe ServidorDeEmails	Faz a inserção de uma nova conta de usuário, com um ID igual ao ID passado por parâmetro, na primeira posição (no início) do	Se formos considerar a inserção como a operação relevante, a complexidade será O(**1**), pois a inserção de uma célula (conta de usuário) no início de uma	O(**n**), sendo **n** o número de células (contas de usuários) da lista encadeada (servidor de Emails).

	servidor de emails (que é uma LECU). Isso só acontece **caso a conta não exista** (que é verificado com o método auxiliar isContaExistente()).	lista encadeada (servidor de emails) é sempre O(1). Se formos considerar o "caminhamento" pela lista encadeada para verificar se a conta de usuário existe como a operação relevante, a complexidade será O(**n**). Isso se dá pois o caminhamento é feito através do método auxiliar isContaExistente(), que já vimos que tem complexidade O(**n**).	
removerConta() da classe ServidorDeEmails	Faz a remoção de uma conta de usuário, com um ID igual ao ID passado por parâmetro, do servidor de emails (que é uma LECU). Isso só acontece **caso a conta não exista**.	O(**n**), sendo **n** o número de contas de usuários do servidor de emails. Isso se dá pois a remoção de uma célula (conta de usuário) de uma lista encadeada (servidor de emails) tem complexidade O(**n**) no pior caso.	O(**n**), sendo **n** o número de células (contas de usuários) da lista encadeada (servidor de Emails).
entregarEmail() da classe ServidorDeEmails	Faz a inserção de uma nova entrada (que contém um email) segundo a ordem de prioridade na caixa de entrada de uma conta de usuário cujo ID foi passado como parâmetro. Isso só acontece **caso a conta não exista**.	Se formos considerar a inserção como a operação relevante, a complexidade será O(**n**). Isso se dá pois a inserção de uma nova entrada é feita através do método adicionarEmail(), a qual já vimos que possui complexidade O(**n**). Se formos considerar o "caminhamento" pela lista encadeada para verificar se a conta de usuário existe como a operação relevante, a complexidade será também O(**n**). Isso se dá pois o caminhamento é feito através do método	O(**n*m**), sendo **n** o número de células (contas de usuários) da lista encadeada (servidor de Emails) e **m** o número de células (entradas) da fila de prioridade encadeada (caixa de entrada).

		auxiliar isContaExistente(), que já vimos que tem complexidade O(**n**).	
consultarCaixa() da classe ServidorDeEmails	Faz a remoção da primeira entrada (que contém um email) da caixa de entrada de uma conta de usuário cujo ID foi passado como parâmetro. Isso só acontece **caso a conta não exista**.	Se formos considerar a remoção como a operação relevante, a complexidade será O(**1**), pois a remoção de uma entrada (que contém um email) é feita através do método removerEmail(), a qual já vimos que possui complexidade O(1). Se formos considerar o "caminhamento" pela lista encadeada para verificar se a conta de usuário existe como a operação relevante, a complexidade será também O(**n**). Isso se dá pois o caminhamento é feito através do método auxiliar isContaExistente(), que já vimos que tem complexidade O(**n**).	O(**n*m**), sendo **n** o número de células (contas de usuários) da lista encadeada (servidor de Emails) e **m** o número de células (entradas) da fila de prioridade encadeada (caixa de entrada).

4. Estratégias de robustez

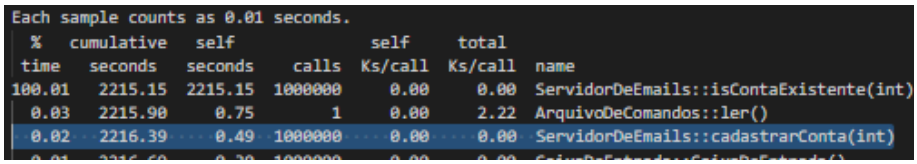
As estratégias de robustez criadas para evitar erros de input de usuários pelo arquivo de comandos foram implementadas através de uma combinação de "erroAsserts" e impressões de mensagens de erros. Aqui estão as estratégias utilizadas:

- ErroAssert utilizado para verificar se o arquivo de comandos foi aberto corretamente.
- ErroAsserts utilizados para verificar se o comando lido é válido ("CADASTRA", "REMOVE", "ENTREGA" ou "CONSULTA").
- ErroAsserts utilizados para verificar se o ID e prioridade do email estão no formato correto (ambos inteiros, ID estando no intervalo $[0, 10^6]$ e prioridade estando no intervalo [0, 9]).
- ErroAsserts utilizados para verificar se a mensagem lido do arquivo de comandos é não-vazia.
- Impressão de mensagem de erro caso haja um comando para cadastrar uma conta já existente.

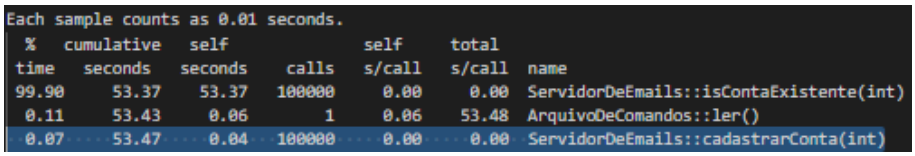
- Impressão de mensagem de erro caso haja um comando para remover, consultar a caixa de entrada ou entregar um email para uma conta que não existe.
- Impressão de mensagem de erro caso haja um comando para consultar uma caixa de entrada vazia.

5. Análise Experimental

Usando o gprof, fizemos dois testes: um teste com um arquivo de comandos com 10^5 comandos e um outro com 10^6 comandos "CADASTRA" com números de ID crescentes 1, 2, 3, … (garantindo que teríamos o pior caso possível para o método isContaExistente()). Veja as imagens do "flat profile" do gprof abaixo. (a primeira imagem é o resultado do teste com 10^5 comandos e a segunda é o resultado do teste 10^6 comandos)

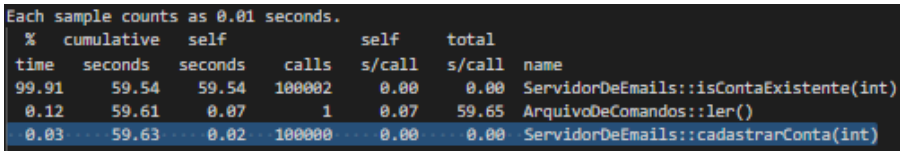


```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ks/call  Ks/call  name
100.01   2215.15  2215.15  1000000     0.00     0.00  ServidorDeEmails::isContaExistente(int)
  0.03   2215.90     0.75        1     0.00     2.22  ArquivoDeComandos::ler()
  0.02   2216.39     0.49  1000000     0.00     0.00  ServidorDeEmails::cadastrarConta(int)
```

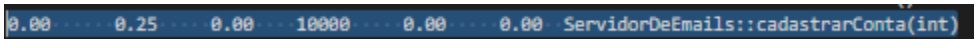


```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 99.90     53.37    53.37   100000     0.00     0.00  ServidorDeEmails::isContaExistente(int)
  0.11     53.43     0.06        1     0.06    53.48  ArquivoDeComandos::ler()
  0.07     53.47     0.04   100000     0.00     0.00  ServidorDeEmails::cadastrarConta(int)
```

Veja que, no primeiro teste (10^5 comandos), o método **cadastrarConta()** demorou 0.4 segundos para ser executado. Já no segundo teste (10^6 comandos), o programa demorou 0.49 segundos para ser executado. Devido a precisão do gprof ser até 0.01 segundos, podemos considerar que houve um aumento de 10 vezes no tempo de execução de um teste para outro, que condiz com o aumento em 10 vezes o número de comandos e, portanto, um aumento em 10 vezes do tamanho da lista encadeada onde as contas são cadastradas. Assim, foi possível atestar empiricamente que a complexidade de tempo do método **cadastrarConta()** é de fato de O(**n**), quando consideramos a operação relevante ser o caminhamento pela lista encadeada. Isso se dá pois o caminhamento é feito pelo método **isContaExistente()**, que já havíamos determinado ter complexidade O(**n**) no pior caso. Se o **cadastrarConta()** não chamasse o **isContaExistente()** e aa chamada ao **isContaExistente()** fosse feito antes da chamada do método **cadastrarConta()**, vemos que o aumento de tempo de execução do método **cadastrarConta()** é inexpressivo. Isso se dá pois **cadastrarConta()** é uma inserção no início da lista encadeada que tem complexidade O(**1**). Fizemos esse teste e obtivemos um resultado que comprova essa hipótese, como vemos nas imagens abaixo. (a primeira imagem é o resultado do teste com 10^4 comandos e a segunda é o resultado do teste 10^5 comandos)



```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 99.91     59.54    59.54   100002     0.00     0.00  ServidorDeEmails::isContaExistente(int)
  0.12     59.61     0.07        1     0.07    59.65  ArquivoDeComandos::ler()
  0.03     59.63     0.02   100000     0.00     0.00  ServidorDeEmails::cadastrarConta(int)
```



```
0.00      0.25      0.00    10000     0.00     0.00  ServidorDeEmails::cadastrarConta(int)
```

O exato mesmo fenômeno ocorre com o método **consultarConta()**: O(**n**) se levarmos em consideração o **isContaExistente()**, e O(**1**) se chamarmos **isContaExistente()** antes de chamar **consultarConta().**
No caso dos comandos "ENTREGA" e "REMOVE", há uma complexidade O(**n**) independente de se chamarmos **isContaExistente()** antes de chamar **entregarEmail()** ou **removerConta()** respectivamente. Isso se dá pois **entregarEmail()** é uma inserção em uma fila de prioridade encadeada (que ocorre na penúltima posição no pior caso) e **removerConta()** é uma remoção em uma lista encadeada (que ocorre na penúltima posição no pior caso). Ambos esses métodos possuem complexidade O(**n**) por conta desse "caminhamento" até a posição de inserção ou remoção. Portanto o aumento em 10 vezes o número da lista/fila gera um aumento em 10 vezes o tempo de execução desses métodos.

6. Conclusão

O problema a ser solucionado neste trabalho prático era de desenvolver um simulador de servidor de emails. O maior desafio do trabalho foi conseguir modelar o servidor de emails e a caixa de entrada como listas e filas encadeadas, e a partir disso criar as classes e métodos necessários para inserir, remover e caminhar sobre essas estruturas de dados.
Com todo esse processo, pude revisar e entender mais a fundo como funcionam listas e filas encadeadas e em quais situações elas melhor se aplicam. Além disso, pude dar continuidade no aprendizado de C++ em geral e conceitos de programação orientada a objetos.
Em suma, conclui-se que o trabalho teve sucesso no seu objetivo de ensino sobre estruturas de dados de listas e filas.

7. Bibliografia

- 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. Versão Traduzida:Algoritmos – Teoria e Prática 3a. Edição, Elsevier, 2012
- https://www.youtube.com/watch?v=imMoeV2vj8U&ab_channel=Programeseufuturo
- Vídeo Aulas e Slides disponibilizadas na Metaturma da disciplina de Estrutura de Dados

8. Instruções de compilação e execução

- Vá para o diretório raiz do projeto (/TP)
- Digite na linha de comando (para limpar arquivos desnecessários do diretório e compilar o programa): **make**
- Digite na linha de comando (para executar o programa): **./bin/programa.exe <nome do arquivo de comandos>**