UFMG
UNIVERSIDADE FEDERAL DE MINAS GERAIS

Universidade Federal de Minas Gerais

Turma: Ciência da Computação **Prof.:** Gisele e Wagner

Nome: Rubia Alice Moreira de Souza

Trabalho Prático 1 - Servidor Emails

Belo Horizonte
2022

1. Introdução

Este trabalho realiza a simulação de um sistema de emails. Nele, é necessário haver uma lista flexível de usuários, aumentando e reduzindo de tamanho segundo o cadastro ou remoção deles. Além disso, cada usuário deve conter uma lista de mensagens, que são ordenadas de acordo com o nível de prioridade dela. As mais importantes devem ser lidas e removidas primeiro, em relação às outras.

Para isso, podemos utilizar uma lista como estrutura de dados. Podemos ter uma lista para armazenar os usuários cadastrados no servidor e, para cada usuário, uma lista de prioridades, que irá armazenar e ordenar automaticamente as mensagens conforme elas são recebidas.

2. Método

2.1. Classes

Inicialmente, temos o namespace Utils que contêm funções de utilidades e aplicações gerais:

Utils	
Descrição: Contém conjunto de funções de variedades diversas	
Propriedades	**Métodos**
	replace(referencia : string&, alvo : string&, conteudo : string&) : string toUpper(referencia : string&) : string trimEnd(referencia : string&)

Além disso, as seguintes estruturas de dados foram utilizadas no projeto. A Lista Encadeada foi utilizada para armazenar os usuários e a Fila de Prioridades para armazenar e ordenar as mensagens de cada um deles.

Lista	
Descrição: Representa uma lista ordenada de elementos	
Propriedades	**Métodos**
tamanho : unsigned int	limpar() getTamanho() : unsigned int estaVazia() : bool

ListaEncadeada<T> : Lista			
Descrição: É a implementação de uma lista de forma duplamente encadeada			
Propriedades	**Métodos**	**Pior Caso**	
		Tempo	**Espaço**
inicio : Celula<T> fim : Celula<T>	adicionarFim(valor : T*)	O(1)	O(1)
	adicionarComeco(valor : T*)	O(1)	O(1)
	adicionarPosicao(valor : T*, posicao unsigned int)	O(n)	O(1)
	get(posicao : unsigned int) : T*	O(n)	O(1)
	removerItem(item : T)	O(n)	O(1)
	removerFim()	O(1)	O(1)
	removerComeco()	O(1)	O(1)
	remover(posicao : unsigned int)	O(n)	O(1)
	limpar()	O(n)	O(1)

FilaPrioridade<T> : Lista			
Descrição: Representa uma implementação de uma Lista na forma de Fila, em que seus elementos são ordenados da esquerda para a direita de forma decrescente de prioridade.			
Propriedades	**Métodos**	**Pior Caso**	
		Tempo	**Espaço**
inicio : CelulaPrioridade<T> fim : CelulaPrioridade<T>	adicionar(valor : T*, prioridade : unsigned int)	O(n)	O(1)
	get(posicao : unsigned int) : T*	O(n)	O(1)
	pop() : T	O(1)	O(1)
	limpar()	O(n)	O(1)

Também temos as classes DTOs, responsáveis por carregar as informações ao longo do sistema.

Mensagem
Descrição: Representa uma mensagem que pode ser enviada para um usuário

Propriedades	Métodos
texto : string prioridade: unsigned int	setTexto(texto : string) getTexto() : string setPrioridade(prioridade : unsigned int) getPrioridade() : unsigned int

CaixaMensagens	
Descrição: Contém a lista de mensagens que um usuário recebe. Sempre retorna a primeira mensagem, removendo-a da Fila de Prioridade de Mensagens.	
Propriedades	**Métodos**
mensagens : FilaPrioridade<Mensagem>	guardarMensagem(mensagem : Mensagem) consultarMensagem() : Mensagem getMensagens() : FilaPrioridade<Mensagem>* getQuantidadeMensagens() : unsigned int

Usuario	
Descrição: Representa um usuário cadastrado no sistema	
Propriedades	**Métodos**
id : unsigned int caixaMensagens : CaixaMensagens	receberMensagem(mensagem : Mensagem) consultarMensagem() : Mensagem getMensagens() : FilaPrioridade<Mensagem> setId(id : unsigned int) getId() : unsigned int

Por fim, temos o servidor, que funciona como um orquestrador. Ele realiza a comunicação entre as classes e seleciona quais funcionalidades serão executadas.

Servidor	
Descrição: Contém as funcionalidades que podem ser executadas e realiza a distribuição de mensagens	
Propriedades	**Métodos**
usuarios : ListaEncadeada<Usuario>	cadastrarNovoUsuario(id : unsigned int)

	removerUsuario(id : unsigned int) entregarMensagem(id : unsigned int, mensagem : Mensagem) consultarEmail(id : unsigned int) : Mensagem

2.2. Fluxo execução

Toda leitura do arquivo de entrada é realizada na main.cpp. À medida que os comandos são identificados, seus parâmetros são separados e então passados para a classe Servidor, que realiza as operações requisitadas. Além disso, todos os erros lançados pela classe Servidor são tratados na main e suas respectivas mensagens de erros são exibidas.

2.3. Configuração de ambiente

O código foi desenvolvido e executado com as seguintes configurações de ambiente:

- **Processador:** Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz | 2701 Mhz, 2 Núcleo(s), 4 Processador(es) Lógico(s)
- **RAM instalada:** 8,00 GB (utilizável: 7,88 GB)
- **Tipo de sistema:** Sistema operacional de 64 bits, processador baseado em x64
- **Sistemas operacionais:** Windows 10, Linux (Ubuntu 20.4)

- **Linguagem utilizada:** C++11
- **Compilador utilizado:** g++ da GNU Compiler Collection

3. Análise de Complexidade

Iremos fazer uma Análise de Complexidade a nível de funcionalidades. Assim, serão analisados os métodos da classe Servidor, que atua como o vínculo entre todas as outras e é responsável por realizar as funções do programa.

3.1. Tempo

Para o caso de Cadastrar um Novo Usuário (cadastrarNovoUsuario()), temos uma primeira condição que depende do usuário já estar cadastrado ou não. Para o caso de um usuário que ainda não foi cadastrado, nós percorremos o vetor de usuários para garantir que seu id é único, então instanciamos o novo usuário e, por fim, adicionamos ao fim da lista. Esse processo realiza: n + 1 + O(1) operações. Assim, o custo para esse caso é O(n). Caso o usuário já esteja cadastrado, o custo irá variar de acordo com a verificação do id. Se ele for o primeiro cadastrado, o custo será de O(1). Caso ele seja o último, o custo será O(n). Após a verificação, é lançada uma exceção. Assim, o pior caso para o cadastro de um novo usuário tem custo O(n) e o melhor caso O(1).

Para Remover um Usuário (removerUsuario()), devemos verificar se a lista está vazia. Nesse caso, o custo será O(1), pois apenas é lançada uma exceção. Caso a lista esteja preenchida, devemos verificar se o usuário que será removido existe. Caso ele não

exista, nós teremos percorrido toda a lista de usuários a sua procura, gerando um custo de O(n). Caso ele já exista, a remoção irá depender da posição do usuário. Se o usuário for o último na lista, o custo será de O(n). Caso ele seja o primeiro, o custo será de O(1). Portanto, temos como pior caso O(n) e melhor caso O(1).

Para Enviar uma Mensagem para um Usuário (entregarMensagem()), devemos verificar se a lista está vazia. Caso esteja, o custo é O(1), pois apenas é retornado um erro. Caso contrário, devemos verificar se o usuário que irá receber a mensagem existe ou não. Para isso devemos percorrer toda a lista de usuários, tentando identificar o id. Caso o id não exista, toda a lista foi percorrida, gerando um custo de O(n). Caso contrário, a mensagem será armazenada na caixa de entrada do usuário. O processo de armazenar a mensagem irá variar, também, de acordo com a prioridade da mensagem. Se a caixa de mensagens está vazia, não é necessário verificar nada e a mensagem apenas será inserida, gerando um custo de O(1). Caso contrário, devemos avaliar a prioridade da mensagem. Se a mensagem for a de menor prioridade, ela será inserida diretamente no fim, com custo de O(1). Caso ela tenha a maior prioridade e seja a primeira do seu tipo, nós teríamos percorrido toda a fila para identificar que ela seria colocada na primeira posição. Logo, isso terá um custo de O(n). Portanto, para o pior caso, teríamos um custo de n + O(m), sendo “n” a quantidade de usuários e “m” a quantidade de mensagens que o usuário possui. O custo do pior caso final será O(max(n, m)). Já para o melhor caso, o custo será apenas O(1).

Por fim, temos o caso de Consultar uma Mensagem (consultarEmail()). Novamente, verificamos se a lista está vazia. Se sim, apenas retornamos um erro e temos custo O(1). Caso contrário, verificamos se o usuário existe. Se não, como verificamos toda a lista de usuários à sua procura, retornamos outro erro com custo de O(n). Se sim, a consulta irá depender da posição em que o usuário está na lista. Se ele for o primeiro, nós teremos um custo de O(1), pois apenas pegamos a primeira mensagem do usuário na caixa de mensagem por vez, isso mantém o custo constante. Para o caso que ele seja o último usuário cadastrado, então tivemos o custo de O(n) para encontrá-lo mais O(1) para pegar sua mensagem. Então, o custo final nesse caso é O(n). Portanto, o pior caso é dado por O(n) e o melhor caso por O(1).

Assim, é possível ver que, pela análise assintótica das funcionalidades do programa, ele deve apresentar um crescimento linear ao verificarmos o Desempenho Computacional na parte de Análise Experimental.

3.2. Espaço

Para a análise de espaço do servidor, devemos levar em consideração que ele, essencialmente, possui várias listas. Inicialmente, o servidor possui uma lista de usuários, que pode ter uma tamanho qualquer “n”. Além disso, para cada usuário, ele terá uma Fila de Prioridades, que contém as suas mensagens, podendo ter um tamanho qualquer “m”. A quantidade de mensagens pode ser maior ou menor dependendo do usuário.

Supondo que todos usuários teriam uma mesma quantidade de mensagens, o espaço ocupado pelo servidor seria dado por “n * m”, sendo “n” a quantidade de usuários e “m” a quantidade de mensagens que cada usuário possui. Assim, seu custo assintótico seria de O(nm).

4. Estratégias de Robustez

Para identificar e tratar possíveis erros do sistema são utilizadas exceções customizadas. Todas elas herdam da classe genérica Excecao.h, que herda de runtime_error.

As exceções são relacionadas a três classes:

- **ExcecoesLista:** Realiza verificações de exceções na busca e remoção de elementos com base em index inválidos: IndexInvalidoException.

- **ExcecoesCaixaEntrada:** Realiza a verificação de tentar acessar a caixa de entrada vazia: AcessoCaixaEntradaVaziaException.

- **ExcecoesServidor:**
    - Verifica se um usuário está sendo cadastrado com um id que já foi utilizado: UsuarioJaCadastradoException.
    - Verifica se foi realizada uma busca ou remoção de um usuário inexistente. Ou seja, com um id não cadastrado: UsuarioNaoEncontradoException.

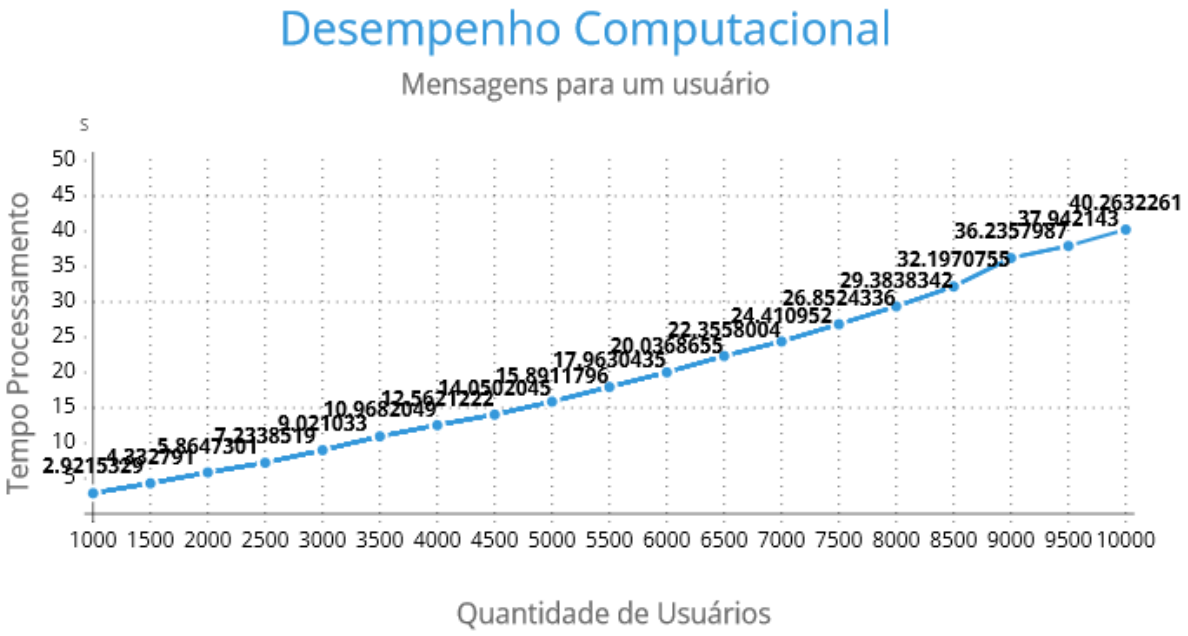
As exceções do servidor foram mapeadas de acordo com os requisitos do trabalho e elas são tratadas na main para exibir as mensagens descritas no enunciado.

Além disso, a main apenas segue a execução do programa ao receber como entrada um arquivo de texto válido. Enquanto um arquivo válido não é informado, ela mantém a execução requisitando novamente um caminho e nome válidos.

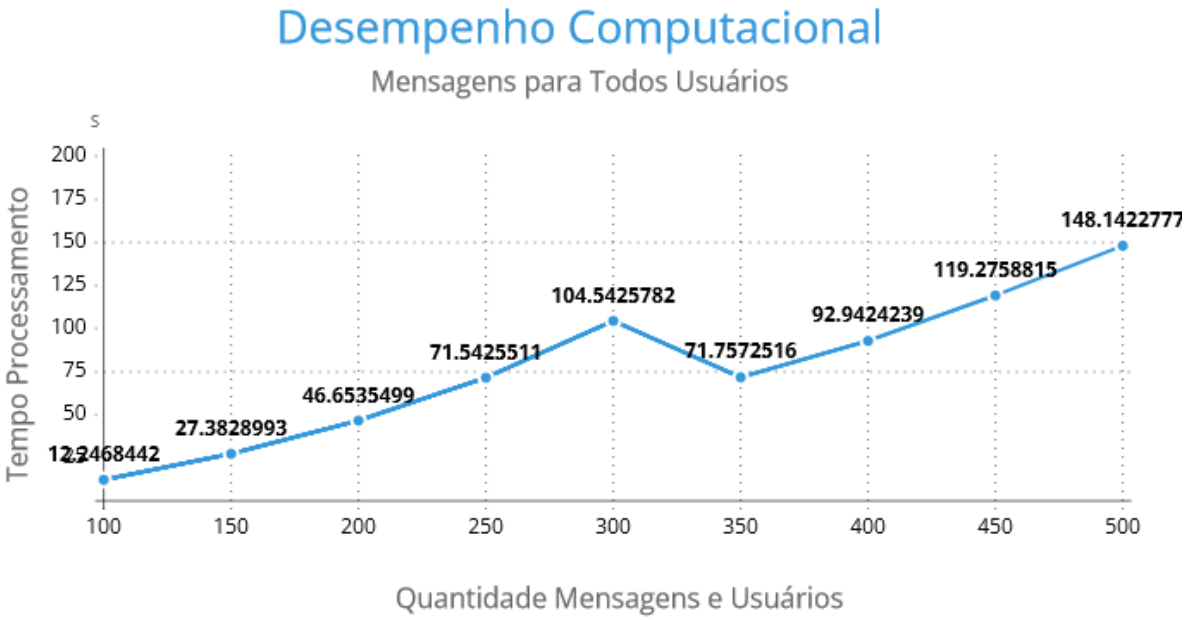
5. Análise Experimental

5.1. Desempenho computacional

Para avaliar o Desempenho Computacional, foram feitos dois tipos de testes. O primeiro tipo de teste foi realizado cadastrando uma quantidade “X” de usuários e enviando apenas para o último usuário “X” mensagens. Utilizando esse padrão, obtivemos o seguinte gráfico:



No eixo “X”, temos a Quantidade de Usuários que foram cadastrados no sistema e, também, o valor da quantidade de mensagens que foram enviadas para o último usuário e depois lidas. Como é possível notar, a linha do gráfico apresenta uma taxa de crescimento aproximadamente linear, seguindo as expectativas da Análise de Complexidade.



Já para o segundo caso de teste, em que foram enviadas uma quantidade de mensagens iguais para todos usuários e depois lidas de cada um deles, o crescimento também aparenta ser linear. Há uma quebra presente entre 300 e 350, que provavelmente deve ter ocorrido por algum uso excessivo de processamento por outro programa do sistema operacional. No entanto, a curva como um todo aparenta ter um comportamento linear.

6. Conclusões

Portanto, na criação da simulação deste sistema de emails, foi possível estudar sobre as estruturas de dados: Lista Duplamente Encadeada e da Fila Encadeada de Prioridades.

7. Bibliografia

Chaimowicz, L. and Prates, R. (2020). Slides virtuais da disciplina de estruturas de dados. Disponibilizado via moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

Apêndice1. Instruções para Compilação

Para executar o código do trabalho, disponibilizado no github, você deve realizar os seguintes passos em qualquer terminal de preferência:

1. Acesse a pasta raiz (UFMG-ED-Servidor-Email) do projeto, onde está presente o Makefile.
2. Execute o comando make.
3. Acesse a pasta bin do projeto.
4. Inicie o executável “run”.
5. Informe o caminho do arquivo de texto que contém os comandos que serão executados. [“caminho/nome_arquivo.txt”]
    - **5.1.** Caso o caminho informado esteja errado, a mensagem abaixo será exibida:
        - **5.1.1.** “Nao foi possivel abrir arquivo de leitura”.
    - **5.2.** Nesse caso, o programa solicitará novamente a entrada do caminho e nome do arquivo.
6. Após isso, o programa irá informar as respectivas saídas para as entradas fornecidas no arquivo de texto no próprio terminal.