TRABALHO PRÁTICO 01

ARTHUR LINHARES MADUREIRA
2021031599

1) INTRODUÇÃO

O objetivo do trabalho foi criar um servidor de e-mails que contém diferentes usuários. Os e-mails possuem prioridades distintas (variando de 0 a 9), sendo este o fator que determina a ordem de e-mails na caixa de entrada: os e-mails com prioridade mais alta se situam no início dessa caixa. Além disso, deve ser possível ler a caixa de entrada de um usuário específico usando sua ID.
Para resolver esse desafio, foram utilizadas duas listas encadeadas. Uma lista é responsável por armazenar os e-mails de cada usuário e a outra aloca os diferentes usuários do sistema.

2) MÉTODO

❾ CLASSE EMAIL

int prioridade: inteiro que define a prioridade do e-mail;

string msg: mensagem contida no e-mail;

Email (int_prioridade, std::string _msg): método construtor;

Email* proximoEmail: ponteiro para o próximo e-mail.

- OBS: essa classe é definida como *friend* da classe caixaEntrada que será apresentada a seguir.

❾ CLASSE CAIXAENTRADA

Email* primeiroEmail: define a cabeça da lista encadeada;

Email* ultimoEmail: define a calda da lista encadeada;

~caixaEntrada(): método destrutor;

std::string consultarEmail (int _id): consulta o primeiro e-mail da lista encadeada;

void entregarEmail (int _prioridade, std::string _msg): entrega um e-mail e define a sua prioridade.

❾ CLASSE USUARIO

int id: inteiro que define o ID do usuário;

Usuario* proximoUsuario: ponteiro para o próximo usuário;

caixaEntrada* caixaUsuario: define caixa de entrada do usuário;

Usuario (int _id): método construtor.

- OBS: essa classe é definida como *friend* da classe listaUsuarios que será apresentada a seguir.

❾ CLASSE LISTAUSUARIOS

Usuario* primeiroUsuario: define a cabeça da lista;

Usuario* ultimoUsuario: define a cauda da lista;

~listaUsuarios (): método destrutor;

std::string cadastraUsuario (int _id): insere um novo elemento do tipo *Usuario na lista encadeada;*

std::string removeUsuario (int _id): remove um elemento da lista encadeada;

std::string selecionaRecebedor (int _id, int prioridade, std::string msg): por meio da ID, seleciona o usuário que vai receber a mensagem;

std::string mostraMsg (int _id): mostra a primeira mensagem do usuário com a ID fornecida.

3) ANÁLISE DE COMPLEXIDADE

3.1) TEMPO

Para fins práticos, vamos considerar que o número de usuários e o número de elementos são iguais a *n*.

Para cadastrar um usuário, deve-se analisar se já existe um primeiro usuário. Essa operação é de complexidade O(1). No melhor caso, ele não existe a complexidade permanece em O(1). Caso não exista, deve-se checar se a ID já está em uso percorrendo toda lista encadeada de *n* elementos, com complexidade, portanto, de O(n). Logo,

$$\mathbf{O(1) + O(n) = O(n)}$$

Para remover um usuário, deve-se analisar se a fila está (ou não) vazia. Essa operação tem complexidade O(1). Caso não esteja vazia, é preciso consultar se o usuário a ser removido é o último, o que também é O(1). Se não for último usuário, precisa-se percorrer a lista até o elemento anterior a ser removido, demandando complexidade O(n). Após isso, é necessário ver se o elemento anterior ao que foi deletado se tornou o último elemento (O(1)). Logo,

$$\mathbf{O(1)+ O(1) + O(n) + O(1)= O(n)}$$

Para entregar um e-mail, deve-se selecionar o recebedor percorrendo, no pior dos casos, todos os n elementos da lista encadeada de usuários. Então, também no pior dos casos, precisa-se adicionar o e-mail na caixa do usuário percorrendo os n elementos da lista encadeada de e-mails.

O(n) + O(n) = O(n)

Para consultar a primeira mensagem de um usuário, deve-se checar se a lista de usuários está (ou não) vazia – O(1). Caso esteja preenchida, é necessário percorrer, no pior dos casos, todos o n usuários: O(n). Na situação em que o usuário com a ID fornecida exista, deve-se checar se a caixa de entrada usuários possui e-mails, demandando O(1).

O(1) +O(n) + O(1) = O(n)

O método destrutor de *listaUsuarios* deve percorrer todos os *n* elementos.

O(n)

O método destrutor de *caixaEntrada* deve percorrer todos os *n* elementos.

O(n)

3.2) ESPAÇO

Vamos considerar que o usuário e o e-mail ocupem uma unidade de espaço. Logo, todos juntos ocupam um espaço equivalente a n + n = 2n.

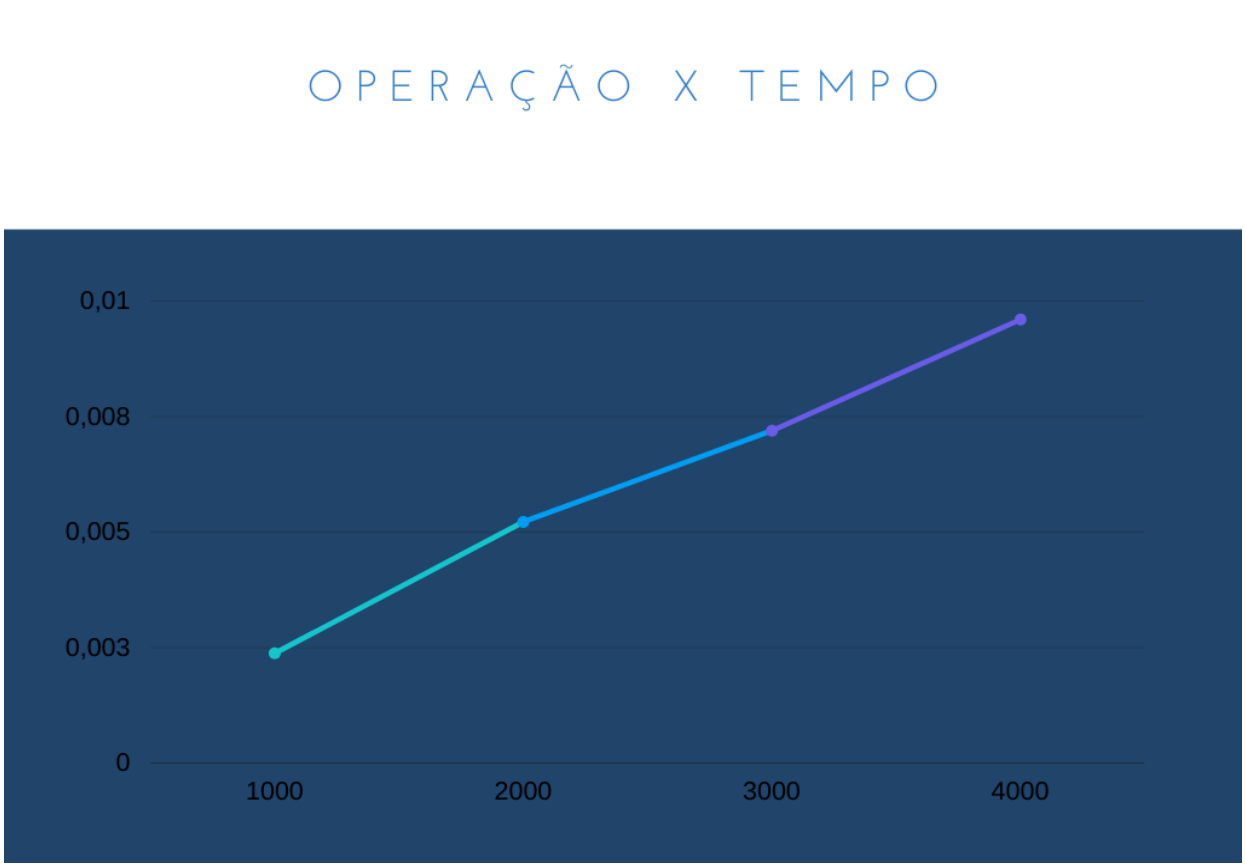
O(2n) = O(n)

4) ESTRATÉGIAS DE ROBUSTEZ

- **CHECAGEM DOS ARQUIVOS:** aciona um *erroAssert* caso não sejam fornecidos os nomes arquivos de entrada e saída ou caso o arquivo de entrada não seja aberto corretamente.
- **ANÁLISE DO ID:** caso o ID fornecido esteja fora do intervalo permitido ($0 \leq ID \leq 10^6$), é acionado um *erroAssert.*
- **ANÁLISE DA PRIORIDADE:** caso a prioridade fornecida esteja fora do intervalo permitido ($0 \leq$ PRIORIDADE ≤ 9), é acionado um *erroAssert.*

5) ANÁLISE EXPERIMENTAL

Como analisado anteriormente, a complexidade de todos os métodos do programa é O(n). Logo, o tempo de execução esperado deve aumentar seguindo proporcionalmente o número de operações. Isso é comprovado pelo gráfico abaixo, no qual o tempo acompanha linearmente o crescimento do número de operações (obviamente, com leves variações que são normais dada a inconstância do hardware).



6) CONCLUSÕES

No trabalho, foram feitas duas listas encadeadas para armazenar os e-mails e os usuários. Com isso, houve muito aprendizado sobre como criar esse tipo de estrutura de dados, além de como realizar as operações básicas (remover e inserir). Nessas duas operações, foi utilizado a prioridade do e-mail para definir em qual local o novo elemento da lista encadeada deve ser alocado. Além disso, foi necessário aprender fazer um método destrutor para a lista encadeada, fato que foi concretizado por meio de um laço de repetição.

7) BIBLIOGRAFIA

Ziviani, N. (2006). Projetos de Algoritmos com Implementações em Java e C++ . Editora Cengage.

INSTRUÇÕES PARA COMPILAÇÃO E EXECUÇÃO

- Para compilar o código, escreva *make* no terminal do diretório;
- Após compilado, execute o código com *./bin/run.out input.txt output.txt.* Você deve escolher o nome dos arquivos de entrada e saída - sempre respeitando a ordem (nome da entrada antes do nome da saída).