Trabalho Prático 1
Estruturas de Dados - Sistema de E-mails

Wallace Eduardo Pereira - 2021032013

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil

wallaceep@ufmg.br

1. Introdução

O problema proposto foi implementar a simulação de um servidor de e-mails, onde o servidor deverá realizar as operações de cadastrar um usuário, remover usuário, entregar mensagens e consultar a caixa de entrada. Para cada usuário cadastrado, era preciso criar uma caixa de entrada inicialmente vazia, onde seriam armazenados os e-mails recebidos, e a cada usuário removido do servidor, deveria-se remover também sua caixa de entrada com todos seus e-mails. As entregas e consultas aos e-mails deveriam ser feitas por ordem de prioridade, ou seja, deveria-se armazenar as mensagens na caixa de entrada de forma que ficassem ordenadas por prioridade, e, a cada consulta realizada, deveria-se ler o e-mail com maior prioridade (no caso de e-mails com mesma prioridade deveria-se ler primeiro o que foi enviado antes), e, depois de lido, deveria-se excluir para que o próximo seja lido na próxima consulta.

2. Implementação

O programa foi desenvolvido na linguagem C++ e compilado pelo compilador G++ da GNU Compiler Collection.

1. Estruturas de Dados

A implementação do programa teve como base a estrutura de dados de uma lista encadeada. Para que sejam cumpridas as especificações propostas no trabalho, foram implementadas duas listas simplesmente encadeadas, uma para armazenar os usuários cadastrados no sistema (classe ListaUsuario), e outra para armazenar os e-mails recebidos pelos usuários (classe CaixaEntrada). Cada usuário cadastrado possui uma caixa de entrada. A seguir uma ilustração da fila encadeada:

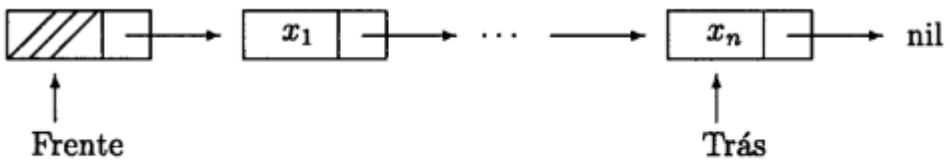
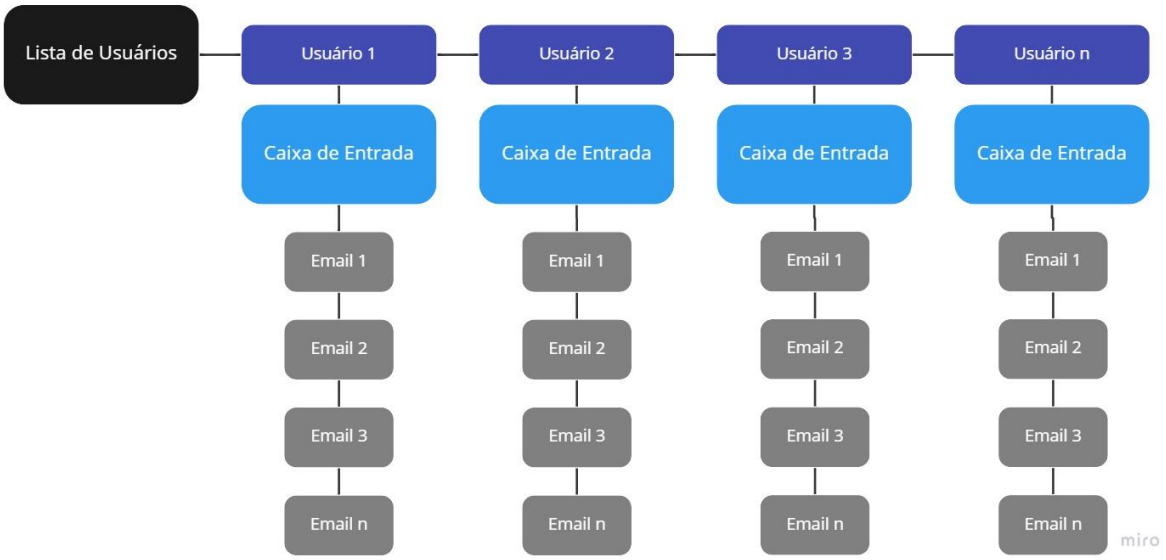


Figura 2.8: Implementação de uma fila através de apontadores

A seguir, uma ilustração de como foi a implementação do programa no contexto de servidor de e-mails, utilizando-se fila encadeada. Observa-se que há a Lista de Usuários, com n usuários cadastrados, e cada um deles possui uma Caixa de Entrada, com n e-mails cadastrados.



2. Classes

Para modularizar a implementação, foram usadas quatro classes principais, sendo elas: Usuário, que deverá conter informações de um usuário que será cadastrado no sistema, Email, que deverá conter as informações de um e-mail, e, para criar uma lista de usuários e uma lista de e-mails, respectivamente, foram criadas as classes ListaUsuario, que irá conter todos os usuários cadastrados no programa, e CaixaEntrada, que armazenará todos os e-mails enviados para determinado usuário.

As listas encadeadas possuem métodos que auxiliam em outras partes da implementação do programa, como o método retornaUser, pertencente à classe ListaUsuario, que retorna o usuário no qual a operação será realizada, e o método verificarExistencia, que verifica se o usuário está presente na lista, e caso não, bloqueia a continuação do método.

As demais classes, Email e Usuário, servem para criar, de maneira individual, os objetos que serão armazenados na lista, ou seja, eles não possuem tantos métodos porque as operações mais importantes para execução do programa estão sendo realizadas nas classes em que serão armazenados. Ademais, para classes que frequentemente deveriam ter acesso aos atributos de outra classe, foi utilizado o conceito de *friend class,* que torna os atributos privados de uma classe visíveis para aquela outra determinada.

2.1. Funções de Inserir, remover, entregar e consultar

Para que fosse cumprido com as especificações propostas para o problema, as funções de maior importância são as de inserir um novo usuário, remover um usuário, entregar uma mensagem para determinado usuário com determinada prioridade e, por fim, consultar a

mensagem de maior prioridade da caixa de entrada de um usuário específico. Essas funções foram as de maior importância porque é por meio delas que o usuário irá interagir com o sistema e realizar as devidas operações que o interessam. Ademais, foi preciso tomar alguns cuidados para evitar problemas ao usar as operações, tais como, ao inserir um usuário, verificar se ele já não está cadastrado no servidor, evitando assim uma duplicata, que poderia causar mau funcionamento do sistema e utilizar memória desnecessária. Além disso, ao realizar as demais operações (remover, entregar mensagem e consultar), é necessário verificar se o usuário realmente está cadastrado na lista de usuários para que as operações fossem realizadas corretamente e, em caso de alguma operação inválida, emitir uma mensagem de erro apontando que o usuário está realizando uma operação de forma indevida.

Toda a implementação das operações refletem na maneira como a estrutura de dados (no caso, as listas encadeadas), foram implementadas, ou seja, no cadastrar, primeiro se percorria o vetor e verificava a existência, e caso não, se insere um usuário no início da lista, e na função de armazenar e-mails, a lista encadeada se comporta como uma fila de prioridade, visto que os e-mails inseridos são posicionados sempre de acordo com sua prioridade, e o e-mail a ser lido é sempre o primeiro da lista, ou seja, aquele com maior prioridade.

3. Análise de Complexidade

3.1. Tempo

Começando a análise para as operações realizadas, e tomando como primeira operação a inserção de um novo usuário ao sistema, e, estando a lista de usuários vazia, o custo para adicionar um novo elemento na lista é constante, ou seja, $O(1)$. Se a lista já estiver com usuários cadastrados, há primeiro uma busca pela lista para verificar se o usuário já existe, tendo, no pior caso, custo $O(n)$. Se encontrar um usuário já existente, a operação é abortada, e, caso não, exige um custo $O(1)$ para adicioná-lo no começo da lista. tendo assim o custo da inserção de um usuário:

$$O(1) + O(n) + O(1) = O(n).$$

Para a operação de remoção, tem-se primeiro que verificar se o usuário está realmente cadastrado no sistema, para isso, é preciso percorrer a lista tendo, no pior caso, custo $O(n)$. Depois de encontrar o usuário, ele é removido do sistema com custo constante, ou seja, $O(1)$, sendo assim a operação de remoção com custo:

$$O(n) + O(1) = O(n).$$

Considerando agora a função de entregar uma mensagem para um usuário. primeiro se verifica a existência do mesmo, tomando o custo de $O(n)$, depois percorre-se novamente o vetor para retornar o usuário (novamente o custo $O(n)$, então, faz-se as atribuições com custo constante $O(1)$. e chama a função do inbox que armazena a mensagem na Caixa de Entrada (ou seja, a lista de e-mails). A função de armazenar e-mails na caixa de entradas possui o melhor caso, em que a caixa de entrada está vazia, inserindo o e-mail na primeira posição, e o pior caso, em que caixa de entrada contém elementos e todos com prioridade maior que o e-mail que está sendo recebido, sendo necessário inserir ao final, comparando todos os elementos $O(n)$ e inserindo $O(1)$. Assim a complexidade da entrega de mensagens fica:

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

Para a função de consultar a caixa de entrada do usuário, verifica-se a existência do mesmo, com custo, no pior caso, $O(n)$, depois retorna o usuário ao qual deseja-se consultar a lista de e-mails, de novo, no pior caso, com custo $O(n)$, depois verifica-se se a caixa de entrada está vazia, com custo $O(1)$, e chama a função da caixa de entrada que lê o e-mail, move o ponteiro para o próximo e deleta o e-mail lido, ambos com custo constante, ou seja, $O(1)$. Dessa forma, o custo para que a consulta à caixa de entrada seja realizada é de:

$$O(n) + O(n) + O(1) + O(1) = O(n).$$

3.2. Espaço

Analisando agora a complexidade de espaço, consideremos o pior caso: uma lista de usuários com n usuários cadastrados, e cada um desses usuários com uma caixa de entrada com n e-mails recebidos. Nesse caso, o custo de espaço do programa considera $O(n)$ para os usuários cadastrados, em que n é a quantidade de usuários, e $O(j)$ para a quantidade de e-mails, considerando j como a quantidade de e-mails. Considerando que cada um dos n usuários possui j e-mails, a ordem de complexidade do programa no pior caso é de $O(n*j)$, ou seja, $O(n)$.

4. Estratégias de Robustez

Para que o sistema funcione corretamente e não apresente erros inesperados, foram implementadas algumas estratégias de robustez. Elas são divididas entre as que comprometem o funcionamento do programa, e por isso deve parar a execução e as que não comprometem e podem fazer a execução seguir normalmente.

Para esse primeiro caso, existem os seguintes comportamentos:

- Se o usuário passa um número inválido de parâmetros o programa encerra execução e emite um aviso;
- Se ocorrer um erro ao abrir o arquivo, o programa encerra sua execução e emite um aviso;

Para o segundo caso, ou seja, casos em que a execução não é comprometida pelo erro, segue-se normalmente sua execução, e emite os seguintes avisos:

- Se tentar cadastrar um usuário que já existe, a ação é abortada, e é emitida uma mensagem de usuário já cadastrado.
- Ao tentar remover um usuário inexistente, a ação é abortada, e é emitida uma mensagem de usuário inexistente.
- Se tentar entregar uma mensagem para um usuário que não foi cadastrado, a ação não é concluída e é emitida uma mensagem de usuário não cadastrado.
- Ao tentar consultar uma caixa de entrada de um usuário não cadastrado, a ação não é concluída e é emitida uma mensagem de usuário não cadastrado.
- Ao tentar consultar uma caixa de entrada que não possui e-mails, é emitida uma mensagem de caixa de entrada vazia.

- Ao executar os comandos das operações, se o comando for inválido é apresentada uma mensagem de comando inválido.

Vale ressaltar que para essas últimas asserções, após a mensagem ser emitida, o programa continua normalmente, pois não compromete a execução do mesmo. Para algumas das funções foram utilizados padrões do arquivo *msgassert.h* disponibilizado pelo professor Wagner Meira.

5. Análise Experimental

5.1. Análise de Desempenho

Ao realizar testes com o arquivo *extra_input_6.txt*, disponibilizado para comparação das saídas dos casos teste, percebe-se que, embora as operações sejam sempre as mesmas. o tempo de execução varia.

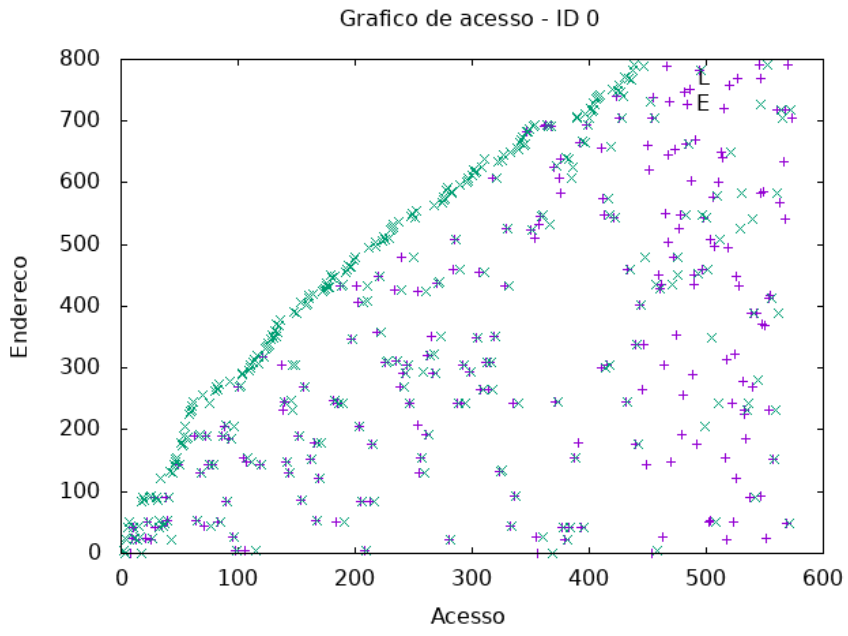
```
574    F 574 8054.183964568 0.031466921
```

```
574    F 574 8087.675545412 0.031711596
```

5.2. Análise de Localidade de Referência

Ao fazer uma análise de localidade de referência, é analisada a posição dos e-mails na memória durante a execução do programa. Para a análise feita, é realizado o teste com o arquivo *extra_input_6.txt*, disponibilizado para comparação das saídas dos casos teste.

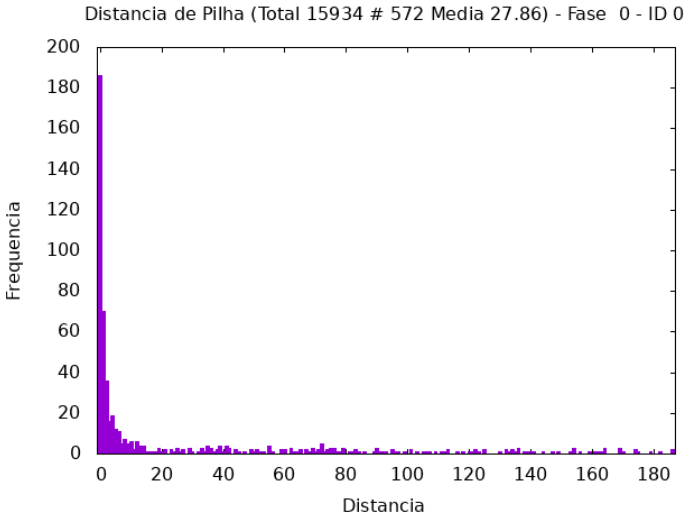
Os gráficos a seguir foram gerados após a execução do programa e o armazenamento dos seus registros na memória.

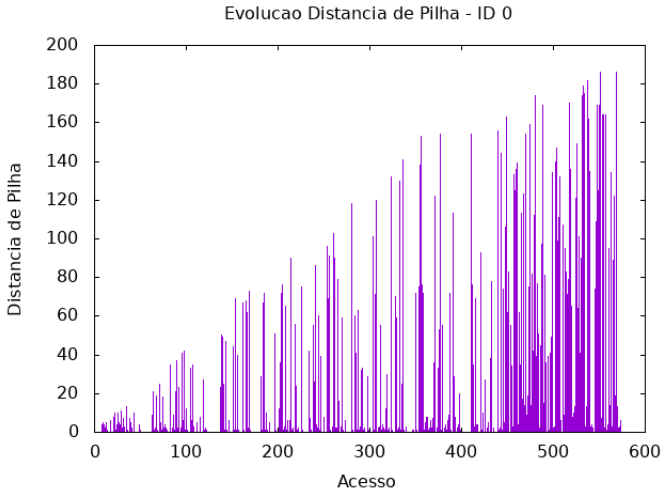


É perceptível que o gráfico ficou esparso, por conta das operações realizadas no teste, que acessaram diferentes registros na memória.

Em relação ao seu formato, é notável que cada conjunto de marcações é referente a um conjunto de operações diferentes.

Analisando os gráficos de distância de pilhas e sua evolução:





É perceptível o aumento da pilha de acordo com o aumento de acessos, seja por meio da inserção de novos e-mails na lista, que também aumenta a distância da pilha entre os registros.

6. Conclusão

Após a implementação do programa, pode-se notar que a escolha correta das estruturas de dados a serem escolhidas era de suma importância, pois fazem toda a diferença na implementação e execução do programa, e, a partir disso, a lista encadeada foi uma escolha que facilitou em várias partes. Além disso, implementar uma fila de prioridade por meio da lista encadeada também facilitou o processo, visto que a remoção era sempre do primeiro elemento e a inserção com base em sua prioridade. Também é notável que a implementação das estruturas de dados foram importantes fatores no custo das funções, visto que era algo linear.

Além disso, com essa implementação, feita com alocação dinâmica de elementos, é possível lidar com sistemas de número arbitrariamente grande de e-mails e usuários. Por meio da implementação desse trabalho prático, foi possível praticar conceitos vistos em sala de aula tanto na disciplina de Estrutura de Dados, como também em disciplinas anteriores. Foi explorado implementações que otimizam o sistema, tais como a própria lista encadeada, fila de prioridade, estratégias de robustez, alocação dinâmica de memória, etc.

Por fim, durante a implementação da solução para o problema, houveram alguns desafios importantes a serem superados, como a escolha da estrutura de dados, a manipulação dos apontadores para a realização das operações e entender a maneira como esses dados seriam armazenados para fazer as manipulações nas operações foram cruciais para a implementação do projeto.

7. Bibliografia

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

Cormen , T., Leiserson, C, Rivest R., Stein, C. Introduction to Algorithms, Third Edition, MIT Press, 2009. Versão Traduzida:Algoritmos – Teoria e Prática3a. Edição, Elsevier, 2012.

Ziviani, N., Projeto de Algoritmos com Implementações em Pascal e C, 3ª Edição, Cengage Learning, 2011.

https://github.com/Chbmleao/TP3-ED-UFMG/blob/master/Documenta%C3%A7%C3%A3o%20-%20TP3.pdf

https://www.geeksforgeeks.org/data-structures/linked-list/#singlyLinkedList

8. Instruções para compilação e execução

- Acesse o diretório TP
- Por meio de um terminal. execute o comando *make all*
    - Observe que por meio deste comando os arquivos são compilados, gerando os arquivos ".obj" e, na pasta bin, o executável *run.out*

- Em seguida, execute o comando *cd bin*
    - Assim, entrará na pasta em que o executável foi gerado.

- Por fim, digite o comando *./run.out nomeInput.txt* (onde o nomeInput deverá ser substituído pelo nome do arquivo a ser lido)
    - Também tem a opção de digitar o comando *./run.out nomeInput.txt -p* em que o makefile irá gerar na pasta bin o arquivo log.out do programa.