Trabalho Prático 1

Brisa do Nascimento Barbosa

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil
brisabn@ufmg.br

1. Introdução

Neste presente trabalho, teve-se como objetivo a implementação de estruturas de dados eficientes para simular um servidor de e-mail, em que suas operações básicas deveriam ser o gerenciamento de usuários, entrega de mensagens e consulta da caixa de entrada. A fim de realizar tal aplicação, foi necessário considerar também a ordem de prioridade dos emails, além de efetuar o uso de boas práticas quanto ao armazenamento dinâmico de memória. Por isso, a estratégia foi baseada na implementação e adaptação da estrutura de dados lista encadeada. Os procedimentos utilizados serão detalhados adiante, com a especificação do uso de bibliotecas, métodos e conceitos de desenvolvimento em C++.

2. Métodos

O programa foi desenvolvido na linguagem C++, compilado em G++ da GNU Compiler Collection. O computador utilizado possui o processador i7 e memória RAM de 8GM. O sistema operacional instalado é o Windows 10, portanto, foi utilizada uma máquina virtual WSL (Subsistema Windows para Linux) para a compilação em Linux.

2.1. Estruturas de dados

A principal estrutura de dados utilizada foi a lista encadeada, que foi implementada tanto para a lista de usuários quanto para a lista de e-mail de cada usuário. Além disso, classes foram utilizadas para organização e abstração dos dados.

2.2. Classes

Em prol da modularização do programa, foram montados dois hpp principais: usuarios e caixa. O primeiro representa a lista de usuários cadastrados nesse servidor de e-mail e tem como classes Lista_Usuarios, Celula_User e User, enquanto a segunda representa a lista de e-mails de cada usuário cadastrado e tem como classes Caixa, Celula_Email e Email.

Para a lista encadeada de usuários, tem-se como classe principal “Lista_Usuarios”, a qual possui como atributo a primeira e a última célula da lista e o tamanho, que é atualizado à medida que adiciona ou retira itens. Já na classe de “Celula_user”, há o item e o próximo da fila como atributos. Esse item é um TAD de User, que representa cada usuário e tem como atributo o seu número de identificação e uma caixa de entrada (que é uma lista encadeada de emails).

Já a lista encadeada de emails tem a classe principal “Caixa” e possui, similarmente à lista de usuários, a primeira e a última célula da lista. Essas são representadas pela classe “Celulas_Email”, que tem um e-mail e a próxima célula como atributo. Por fim, a classe de “Email” possui como atributo a sua prioridade e a mensagem.

Abaixo, apresenta-se uma figura da estruturação desse sistema,

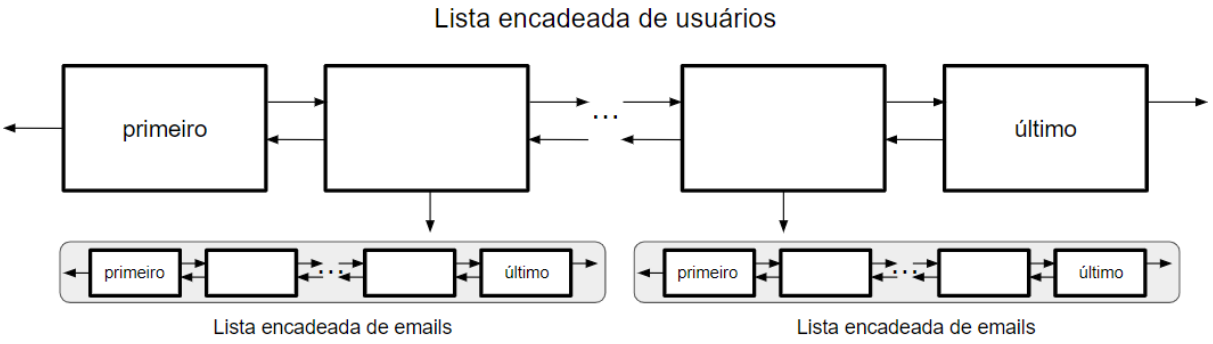


Figura 1.

2.3. Funções

Cada lista encadeada implementada possui funções específicas que se adequam às necessidades de cada TAD. Assim, a Caixa, por exemplo, precisa de uma função específica para armazenar os emails na ordem de prioridade, enquanto Lista_Usuarios pode-se armazenar cada item recebido no início da lista.

Funções da lista encadeada de usuários: possui como funções principais inserir e remover (ambas com o ID do usuário como parâmetro) e funções auxiliares para pesquisar pelo ID e sua posição na fila, útil para a função de posicionar um usuário.

Funções da lista encadeada da caixa de entrada: possui como funções principais inserir de acordo com a prioridade (com um email e a prioridade como parâmetro) e remover, que remove do início, assim como a função de imprimir, que chama a função de imprimir mensagem da classe Email.

Assim, no funcionamento do programa, ao receber os comandos lidos a partir de um arquivo fornecido na linha de comando, realizam-se as funções da seguinte maneira:

CADASTRA: procura na lista se existe o ID fornecido, se não, insere o usuário no início da lista.

REMOVE: procura na lista se existe o ID fornecido, se sim, remove pela posição desse.

ENTREGA: procura na lista se existe o ID fornecido, se sim, pesquisa por ele e a partir da caixa desse usuário, insere-se o email de acordo com a prioridade.

CONSULTA: procura na lista se existe o ID fornecido, se sim, verifica se a lista de emails desse usuário está vazia, se não está, imprime o início da lista e depois o remove.

3. Análise de complexidade

3.1. Tempo

Todos os comandos primeiro analisam se já existe o ID fornecido na lista de usuários cadastros, isso demanda, no melhor caso, O(1) se ele já estiver na primeira posição da lista. Mas caso esteja na última posição ou não existir, encaramos o pior caso, em que a complexidade será O(n).

Ao cadastrar um usuário, ou seja, inserir uma célula no início da lista, o custo é constante, O(1). Já o comando de remover um usuário possui a complexidade O(n), já que remove o ID de acordo com sua posição, e, para saber a posição daquele usuário, procura-se do primeiro item da lista até aquele ID, ou seja, O(n) no pior caso e O(1) no melhor caso.

Para entregar um email para um determinado usuário, insere-se de acordo com a prioridade, então é realizada uma pesquisa de comparação dele com os outros itens na lista, ou seja, uma operação com custo O(n) na pior das hipóteses. Por fim, para consultar uma caixa de entrada de determinado usuário, a complexidade será constante, já que será retirado e impresso o primeiro item da caixa de entrada do usuário.

Assim, a complexidade de tempo será

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

3.2. Espaço

As listas encadeadas possuem, além dos itens de usuários e emails cadastrados, espaço extra para armazenar a primeira e última posição (que apontam para vazio), o que equivale a O(1) em termos de complexidade de espaço. E quanto ao armazenamento dos itens das listas, a complexidade é a quantidade máxima m de emails e n de usuários, logo, O(nm).

Portanto, a complexidade de tempo será

$$O(nm) + O(1) = O(nm)$$

4. Estratégias de Robustez

A fim de firmar a robustez do código, foi utilizada a biblioteca <msgassert.h>, a qual define macros para verificar a precisão das funções.

No TAD de **caixa de entrada**, são definidos os seguintes asserts:

- Erro a prioridade está fora do intervalo [1,9];
- Erro se a lista estiver vazia ao remover o início;
- Erro se a lista estiver vazia ao pesquisar sequencialmente;
- Erro se a lista estiver vazia ao imprimir o início;

No TAD de **usuários cadastrados**, são definidos os seguintes asserts

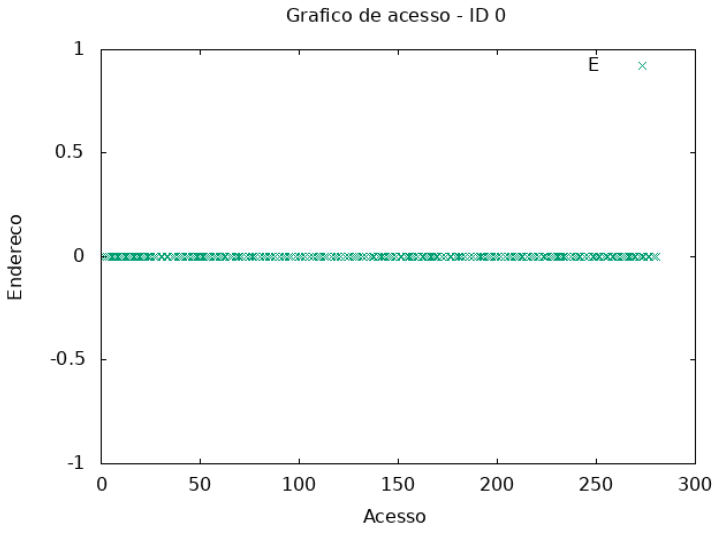
- Erro o ID está fora o intervalo [0, 10^6]
- Erro de posição inválida se o tamanho da lista for menor que a posição ou a posição for menor que zero;
- Erro se a lista estiver vazia ao remover uma posição de um ID;
- Erro se a lista estiver vazia ao pesquisar sequencialmente;

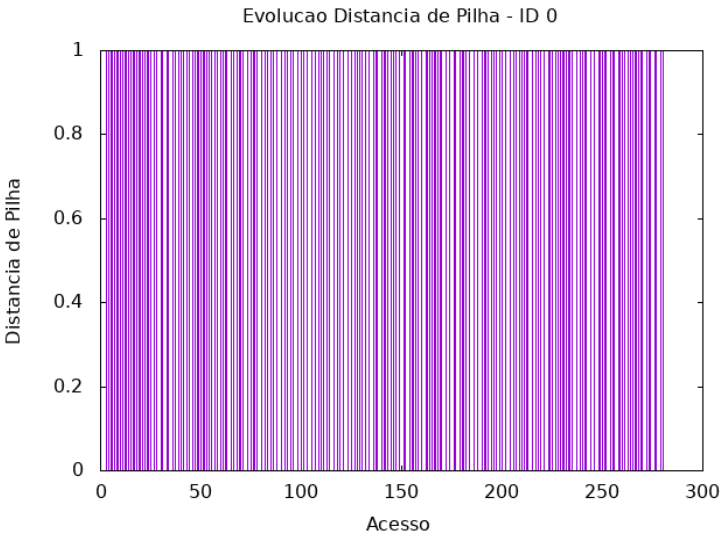
Na **main**, é definido os seguintes asserts

- Erro quando o arquivo de com os comandos não foi informado;
- Erro na abertura do arquivo;
- Erro no fechamento do arquivo;

5. Análise Experimental

A partir do uso da biblioteca memlog, foram obtidos logs de testes de vários tamanhos de lista, segue-se o teste realizado com o cadastro de 30 usuários, em que é mostrada uma linearidade esperada de acesso da memória:





Os gráficos gerados pelo gnuplot permitem verificar que o acesso à memória é bem definido e constante, ou seja, o desempenho computacional é eficiente para as demandas do projeto.

Além disso, com os testes realizados, observou-se o funcionamento adequado do programa para vários usuários e emails entregues, sendo que quanto maior as listas encadeadas, o custo de tempo adicionado era imperceptível. Por isso, entende-se que os resultados são positivos para efetivação de um servidor de email com essa dinâmica de memória.

6. Conclusão

A partir do desenvolvimento do programa, pode-se compreender o funcionamento de listas e manipulações da mesma para adequação às necessidades de um contrato. Para tal, foi necessário ponderar a respeito de estruturas de dados que seriam adequadas para usuários cadastrados e suas respectivas caixas de entradas, e que essas pudessem ser dinâmicas e de tamanho indefinido. Principalmente porque, com diferentes prioridades, houve necessidade de checar a prioridade de cada item da caixa de entrada para determinar onde inserir um novo email. Dessa forma, foi escolhida a lista encadeada, que permite adicionar e retirar elementos em qualquer posição, além de possuir uma complexidade satisfatória para o requerido.

Contudo, ao longo da análise de memória do servidor de emails, houve um desafio considerável quanto à implementação de um log da memória do programa a partir da biblioteca <*memlog*>, já que, para gerar gráficos, era necessário um arquivo log.out apurado. No início, não foi claro identificar locais do código para escrever e ler a memória, mas, posteriormente, empreendendo mais tempo para estudar a estrutura de um memlog, essa dúvida foi sanada.

Por fim, com a finalização do programa e subsequentes análises computacionais, pode-se verificar a relevância da compreensão de estruturas de dados, nesse caso de listas encadeadas, para um entendimento correto da manipulação e gerenciamento de dados. Além disso, a prática de modularização e boas práticas de C++ foi bastante válida para compreender como colocar uma lista encadeada “dentro” de outra lista encadeada, já que foi essencial separar tipos abstratos para usuário e respectivas caixas de entradas.

Bibliografia

- Cormen , T., Leiserson, C, Rivest R., Stein, C. *Introduction to Algorithms, Third Edition, MIT Press, 2009.*
  Chapter 1: Foundations.
- 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.

Instruções de compilação e execução

O makefile possui dois comandos principais: make (compila arquivos recentemente modificados) e make all (compila todos os arquivos). Para rodar corretamente, deve-se utilizar um terminal e digitar make para compilar.

Para informar o arquivo .txt que deve ser lido, basta digitá-lo após o argumento do executável.

Um exemplo de linha de comando após compilado o programa é:

./bin/run.out input.txt

Onde ./bin/run.out é o executável e input é o arquivo de texto que contém os comandos da simulação.

.