Trabalho Prático 1 - Servidor de Emails

Lucas Almeida Santos de Souza - 2021092563[1]

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

luscaxalmeidass@ufmg.br

1 Introdução

Este trabalho consiste na construção de um servidor de emails, supondo que o aluno seja um funcionário da Google e queira fazer um sistema mais eficiente e com melhor gerenciamento de memória utilizando os conceitos de tipos abstratos de dados e alocação dinâmica.

O servidor deve gerenciar os usuários, podendo criar e remover contas com `ids` únicos que podem variar de 0 a 10^6. Também deve ser possível enviar emails para um usuário e acessar a caixa de entrada do mesmo para ler as mensagens recebidas. Cada email tem, além da mensagem, uma prioridade estabelecidada no intervalo de 0 a 9, que define sua importância. A caixa de entrada deve ordenar os emails recebidos em ordem decrescente de prioridade e, sempre que consultada, imprimir o primeiro email da lista e remover o email lido.

2 Método

2.1 Especificações

O código foi desenvolvido na linguagem C++, compilada pelo compilador GCC da GNU Compiler Collection. O Computador utilizado tem as seguintes especificações:

- Sistema operacional Linux Ubuntu 22.04
- Processador intel core i5 8ª geração
- 8GB de memória RAM

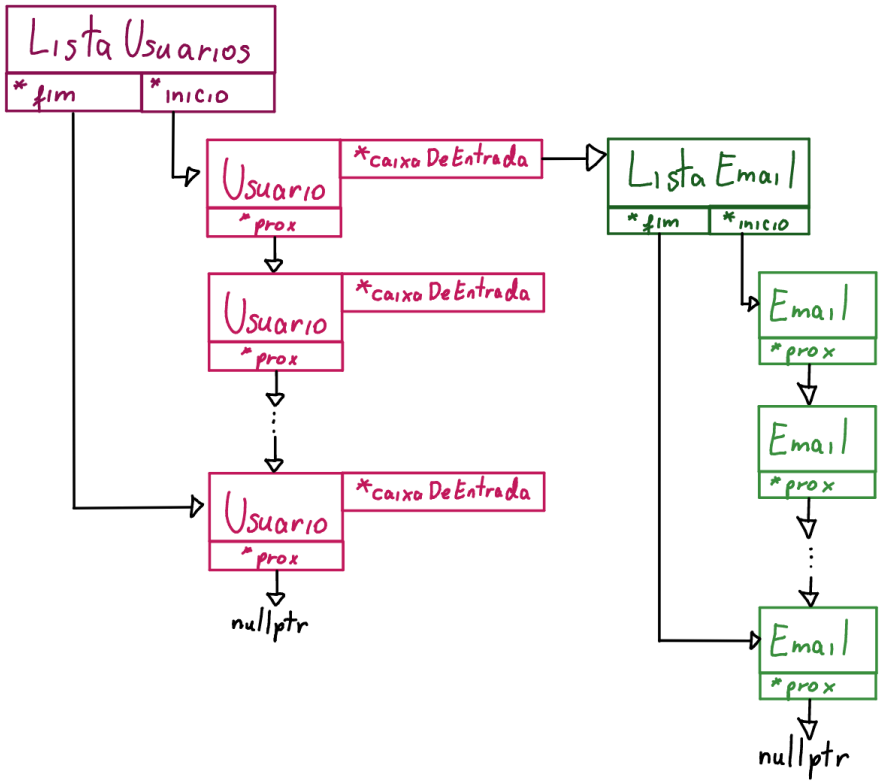
2.2 Execução

O trabalho foi realizado utilizando os conceitos de lista e fila encadeadas vistos em sala. No total foram implementadas cinco classes representando os objetos necessários para a construção do servidor de emails.

As classes Email e ListaEmail são baseadas no conceito de Lista Encadeada, porém com

algumas alterações nas funções de inserção e remoção. A classe Email representa os nós[1] da lista, e a ListaEmail é responsável por gerenciá-los[2]. Já as classes Usuario e ListaUsuario representam uma lista encadeada simples, sem alterações nas suas funções, e cada nó contém como dado um objeto da classe ListaEmail. A imagem abaixo ilustra o funcionamento da estrutura de dados descrita:

Figura 1: Esquema de funcionamento das listas encadeadas



2.2.1 Lista de Emails

A lista de emails é composta de duas classes que se comportam como os nós e o cabeçalho de uma lista encadeada.

A primeira classe a ser implementada é o **Email**, que são os nós da lista. Cada Email contém o corpo da mensagem em formato `std::string`, um número inteiro indicando sua prioridade e um ponteiro para o próximo Email da lista.

[1] Um nó geralmente é um elemento da lista, composto de um item com seus dados e um ponteiro ou qualquer tipo de referência que o conecte a outro nó.

[2] Nesse modo de implementação de listas encadeadas, a classe Lista contém somente ponteiros para a primeira e última posição da lista, e os usa para realizar as operações.

A segunda classe é a **ListaEmail**, que contem os ponteiros para o início e fim da lista e um inteiro indicando seu tamanho atual. Ela é responsável por gerenciar os nós da lista e realizar operações, mas conta com algumas alterações nas funções de inserção e remoção de elementos.

A função `void insereOrdenado(Email* e)` recebe um email a ser inserido e percorre a lista até que a prioridade do próximo email seja menor ou igual à do recebido, inserindo o email naquela posição.

Já a função `bool removePrimeiro(Email& e)` se comporta como uma função remoção em uma fila encadeada, ou seja, o elemento a ser removido é sempre o primeiro da lista. Ela recebe uma referência para um Email e, se a lista estiver vazia, a função retorna `false`, indicando que a operação não pode ser realizada. Caso contrário, ela armazena o email a ser removido no endereço recebido, exclui a mensagem e retrona `true`.

2.2.2 Lista de Usuários

A lista de usuários também consiste em duas classes, uma para os nós e a outra para o cabeçalho.

A classe **Usuario** contém o `id` único do usuário, uma lista de emails e um ponteiro para o próximo usuário. Ele conta com as funções `void recebeEmail(Email *e)`, que recebe um endereço de email e o insere em sua lista de emails e `bool lerEmail(string &mensagem)`, que recebe uma referência para uma string e remove o primeiro email da lista, armazenando sua mensagem no endereço recebido e retornando `true` caso a operação de remoção tenha sido concluída com sucesso e `false` caso contrário.

A classe **ListaUsuario** é composta das funções padrão[3] de uma lista encadeada, visto que não há nenhum comportamento específico para essa lista.

2.2.3 Servidor

A classe **Servidor** gerencia todo o sistema, com as funções `adicionaUsuario(int id)`, que adiciona um usuário com o `id` recebido caso não exista nenhum usuário com o mesmo id; `removeUsuario(int id)` que busca o usuário por `id` e, caso encontre, remove o usuário; `consultaUsuario(int id)` que busca o usuário por `id` e lê o primeiro email de sua caixa de entrada e `enviaMensagem(int id, int prioridade, string mensagem)` para adicionar uma mensagem à caixa de entrada do usuário. Esses procedimentos chamam as funções das listas e utilizam os booleanos retornados para imprimir as mensagens de sucesso ou erro para o usuário.

2.2.4 Main

Por fim, a função **Main** é responsável por ler os comandos enviados em um loop que se encerra quando o valor lido for EOF. A cada instrução lida, o programa confere o comando recebido, chamando as funções de Servidor. Para as funções CADASTRA, CONSULTA e REMOVE, ele apenas recebe o id e chama o respectivo método. Porém na função ENTREGA, além de receber o id e a prioridade, antes de chamar a função de Servidor o programa utiliza um loop para

[3]Inserção: insere um elemento ao final da fila; Remoção: remove um elemento pesquisando seu id; Limpeza: deleta todos os elementos da lista e coloca o tamanho como 0

receber cada palavra da mensagem e concatenar até encontrar a palavra FIM, que indica o fim da mensagem.

3 Análise de Complexidade

Nesta seção, analisaremos a complexidade das quatro funções que a classe Servidor executa. No geral, as funções individuais têm complexidade de tempo linear, porém é importante ressaltar que, como na função **Main** há um loop para receber os comandos de cada linha enviada, a complexidade de tempo geral do programa não é linear, pois temos x instruções e, para cada instrução, temos uma função de complexidade linear $O(n)$, tornando a complexidade de tempo geral do programa $O(x) \times O(n) = O(xn)$ e, no pior caso $x = n$, temos $O(n^2)$.

3.1 Complexidade de tempo

3.1.1 função adicionaUsuarios( )

Essa função recebe o `id` do usuário a ser **cadastrado**, percorre a lista de caixas de entrada e, caso o `id` informado não esteja presente na lista, cria uma caixa de entrada com aquele `id`. No funcionamento normal da função (quando um usuário é adicionado), sua complexidade é de $\Theta(n)$, sendo n o tamanho da lista no momento da inserção. Isso se dá pois nesse caso, o programa percorrerá toda a lista para garantir que nenhum elemento tem o mesmo `id`. Os melhores casos, de complexidade 1, são:

1. se o elemento que se deseja adicionar tem o mesmo `id` que o primeiro elemento da lista e
2. se o elemento a ser adicionado for o primeiro, pois ainda não há lista para ser percorrida.

Assim, a complexidade geral da função é $O(n)$.

3.1.2 função removeUsuario( )

Essa função recebe o `id` do usuário a ser **removido**, percorre a lista de caixas de entrada e, ao encontrar a caixa de entrada com o `id` informado, remove essa caixa da lista. O pior caso dessa função é quando a caixa a ser removida não existe ou é a última, pois nesse caso o programa percorrerá toda a lista. Assim, temos que a complexidade dessa função é $O(n)$.

3.1.3 função consultaUsuario( )

Essa função recebe um `id` e busca o usuário para ler o primeiro email de sua caixa de entrada. Ao percorrer a lista em busca do usuário, o pior caso também é quando a caixa a ser consultada não existe ou é a última, tendo a complexidade da função em $O(n)$, pois a função que retira o primeiro email da lista sempre tem complexidade 1.

3.1.4 função enviaMensagem( )

Essa função recebe um `id`, o texto de uma mensagem e sua prioridade, busca o usuário pelo `id` e adiciona em sua caixa de entrada o email ordenado pela prioridade atribuída. Nesse processo

existem dois loops, porém não são completamente aninhados, pois há uma condicional que os separa. O primeiro loop percorre a lista de usuários buscando o `id` informado e, **se o usuário for encontrado**, ele chama a função `recebeEmail()` de Usuario, onde entra em outro loop para percorrer a lista de emails até que a prioridade do email a ser inserido seja maior que o email atual, inserindo-o naquela posição. Sendo n = número de usuários cadastrados e m = número de emails presentes na caixa de entrada desse usuário, a complexidade de tempo da função pode ser definida como $O(n + m)$. No pior caso, temos $n = m$, tornando a complexidade de tempo da função $O(2n)$ ou simplesmente $O(n)$.

3.2 Complexidade de espaço

O programa é composto de uma lista de n usuários únicos e cada um desses usuários tem uma caixa de entrada própria. Considerando m como o número total de emails enviados para os n usuários, temos que a complexidade de espaço do programa $O(n + m)$, ou seja, complexidade linear. No pior caso, temos uma quantidade de usuários igual à quantidade de emails, tornando a complexidade $O(2n)$, ou simplesmente $O(n)$.

4 Estratégias de Robustez

Para tratar da robustez do código, foram usadas as macros da biblioteca `msgassert.h`. No construtor de `Email()`, foi usado um `avisoAssert()` para avisar se a prioridade definida está fora dos limites estabelecidos (entre 0 e 9). Caso esteja fora dos limites, o programa reajusta a prioridade para o limite mais próximo (se for abaixo de 0 reajusta para 0 e se for acima de 9 reajusta para 9). No construtor de `Usuario()` também há uma função `avisoAssert()` para informar caso o `id` esteja fora dos limites estabelecidos (entre 0 e 10^6). Nessa parte não foi reajustado o `id` como feito com no `Email()` pois há o risco de já que exista um usuário com o `id` 0 ou 10^6.

5 Análise Experimental

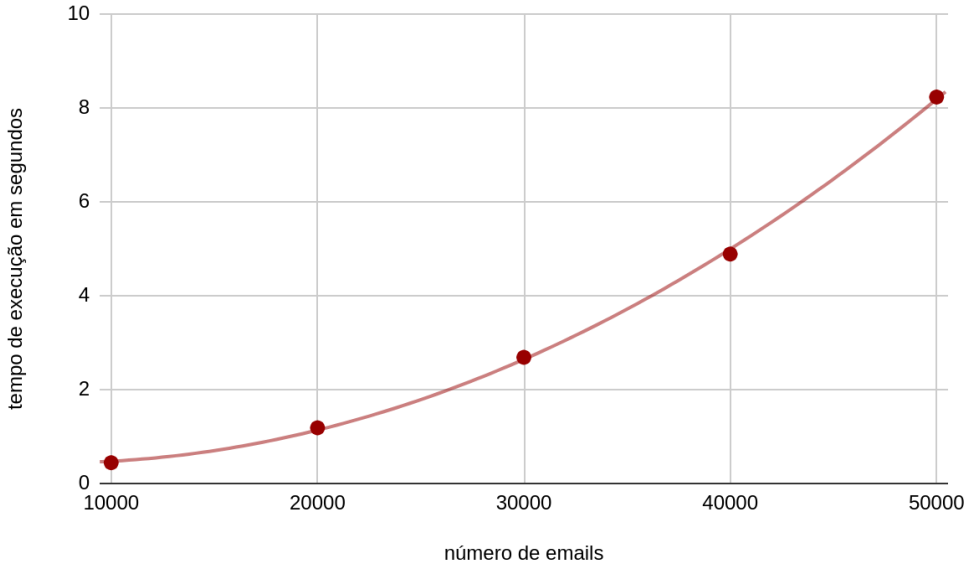
Para a análise de tempo de execução, foram utilizadas as funções da biblioteca `memlog.h`. As funções `iniciaMemLog()` e `finalizaMemLog()` englobam todo o programa, marcando o tempo de execução total. Como não foi feita uma análise dos acessos de memória dentro das funções, o arquivo log.out registrou somente os tempos de início e fim do programa.

Para as entradas de teste, foram construídos arquivos de entrada `.txt` que criam um usuário com `id` $= 1$ e enviam uma quantidade x de emails de tamanho constante e mesma prioridade[4] para cada usuário. Para gerar os arquivos, foram usados scripts *python* que imprimissem os comandos repetindo cada entrada x vezes. Antes de executar os testes, o computador foi completamente reiniciado e somente o terminal foi aberto durante a execução.

[4] A prioridade de todos os emails enviados nos testes foi definida como 0 para poder simular o pior caso possível, pois ao inserir o n-ésimo email, o programa irá percorrer todos os $n - 1$ emails anteriores para enfim adicioná-lo ao final da lista.

No total, foram feitos 5 testes com x igual a 10 mil, 20 mil, 30 mil, 40 mil e 50 mil emails. Podemos observar o tempo de execução de cada teste no gráfico abaixo:

Figura 2: Gráfico de tempo de execução dos testes



A curva gerada representa a função $y = 4,22^{-9}x^2 - 6,01^{-5}x + 0,652$

Ao analisarmos o tempo de execução desses cinco exemplos, podemos confirmar a afirmação feita durante a Análise de Complexidade de que, apesar das funções do programa serem de complexidade linear, a complexidade geral da função $F(n)$ testada tem crescimento representado como

$$\Theta(n) \preceq F(n) \preceq \Theta(n^2).$$

Cada ponto do gráfico representa um dos testes feitos e a linha em vermelho claro representa uma aproximação calculada de uma função quadrática que melhor aproximasse os pontos. Como podemos ver, a linha atravessa todos os pontos com um erro mínimo. Essa variação se dá pois, como dito anteriormente, a complexidade do programa não é linear, mas também não é quadrática em todos os casos.

6 Conclusões

Nesse trabalho utilizamos os conhecimentos de estruturas de C++ adiquiridas nas disciplinas de **Estruturas de Dados** e **Programação e Desenvolvimento de Software II** para criar um sistema que simule um servidor de emails, desde seu planejamento até sua execução e relatório.

A execução da tarefa foi uma oportunidade para colocar em prática os conhecimentos de planejamento de sistemas, com o objetivo de organizar as etapas da criação do sistema e evitar problemas de dependência entre classes. Todas a implementação foi planejada previamente e, durante sua execução, a ordem de criação das classes também foi pensada para que houvesse

a menor dependência possível entre elas, ou seja, evitar precisar de uma outra classe ainda não existente durante a implementação de uma classe.

Durante a implementação, foram utilizados conceitos das duas disciplinas mencionadas para construir uma Lista Encadeada sem utilizar a classe `List` da biblioteca padrão(STL) de C++. O modo escolhido para implementar a Lista foi visto na disciplina de **Programação e Desenvolvimento de Software II** (MACHARET, 2022) e, apesar de ter a mesma eficiência que a Lista vista em **Estruturas de Dados** (PRATES e CHAIMOWICZ, 2020)[5], foi escolhido para a realização do trabalho devido à maior familiaridade. Além disso, também foram feitas algumas alterações nas funções padrão de uma Lista para cumprir os requisitos do trabalho, utilizando conceitos de Filas Encadeadas e Listas Ordenadas.

Depois de implementado todo o programa, foi possível utilizar os conhecimentos de depuração de código para encontrar erros de lógica que, apesar de permitir que o programa compile, geram saídas diferentes do esperado. Para isso, foi utilizado o depurador **gdb** incorporado do VSCode, que permitiu percorrer o programa linha por linha para encontrar exatamente onde se encontravam esses problemas e repará-los mais eficientemente.

Quando o código rodou corretamente e gerou as saídas esperadas, foi utilizado o software **Valgrind** para descobrir possíveis *memory leaks*. O software informa as funções que causam o vazamento de memória e, com isso, podem ser feitas correções para impedir que esse problema continue acontecendo, pois mesmo que o programa esteja funcionando, é importante que ele seja eficiente e tenha um bom gerenciamento de memória.

Por fim, a formulação do relatório se mostrou como uma oportunidade para utilizar os conhecimentos do sistema de preparação de documentos LaTeX. É importante que se tenha o conhecimento e a prática com a ferramenta, pois durante o percurso acadêmico há necessidade constante de escrever artigos e documentos científicos e o LaTeX contribui para que o aluno possa se concentrar no conteúdo escrito sem se preocupar com a formatação final.

Referências

Initializing strings as null vs empty string. Disponível em: <https://stackoverflow.com/questions/11556394/>. Acesso em: 18 out. 2022.

MACHARET, Douglas G. Aula 05 - Listas encadeadas e Árvores binárias. 12 abr. 2022. Apresentação em formato PDF. Disponível em: <https://virtual.ufmg.br/20221/>. Acesso em: 18 out. 2022.

PRATES, Raquel; CHAIMOWICZ, Luiz. Aula 05 - Listas Lineares. 24 ago. 2020. Apresentação em formato PDF. Disponível em: <https://virtual.ufmg.br/20222/>. Acesso em: 18 out. 2022.

What is Linked List - GeeksforGeeks. Disponível em: <https://www.geeksforgeeks.org/what-is-linked-list/>. Acesso em: 15 out. 2022.

[5] A implementação de Lista vista na disciplina de Estruturas de Dados consiste de uma classe Nó contendo somente o dado e uma classe Lista, que contém um nó e uma referência para outro objeto da classe Lista

Instruções de Compilação e execução

Para executar o programa, siga os passos:

1. Acesse o diretório raiz `tp`;
2. Utilizando um terminal, utilize o comando `make` para compilar o código;
3. Com esse comando, um arquivo `run.out` será criado no diretório `bin`;
4. Execute usando `make run`;
5. O programa iniciará e já é possível inserir os comandos manualmente.

OBS: Caso se queira utilizar um arquivo de entrada, basta utilizar o comando:
`./bin/run.out < nomeDoArquivo`

Exemplos de comandos de entrada

- `./bin/run.out < entrada.txt`
- `./bin/run.out < diretorio/entrada.txt`
- `./bin/run.out < diretorio/entrada.txt > saida.txt`

Formato do arquivo de entrada

O arquivo de entrada deve ser um arquivo de texto (em formato `.txt`, por exemplo) contendo apenas os comandos que serão interpretados pelo programa. Os comandos disponíveis são:

- **CADASTRA <id>:** cadastra um usuário com o id informado e cria uma caixa de entrada vazia para ele.
- **REMOVE <id>:** remove o usuário através do id informado e esvazia sua caixa de entrada.
- **ENTREGA <id> <prioridade> <mensagem> FIM:** entrega uma mensagem para o usuário cujo id é informado, adicionando o email à sua caixa de entrada ordenada por prioridade em ordem decrescente. A mensagem é composta de palavras separadas por espaços em branco e termina com a palavra reservada FIM.
- **CONSULTA <id>:** imprime a primeira mensagem da caixa de entrada do usuário cujo id é informado e exclui a mensagem de sua caixa de entrada.

Exemplo de formato de arquivo de entrada

entrada.txt

CADASTRA 3
ENTREGA 3 5 mensagem para usuário 3 com prioridade 5 FIM
CADASTRA 13
ENTREGA 13 7 mensagem para usuário 13 com prioridade 7 FIM
CONSULTA 3
REMOVE 13