Trabalho Prático 1

Gabriela Tavares Barreto

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

gbarreto@ufmg.br

1 Introdução

A proposta desse trabalho prático foi de implementar um simulador de servidor de e-mails. A simulação tinha que se comportar tal qual um servidor real, no qual é possível adicionar e remover usuários, criar caixas de entrada para eles e permitir a entrega de mensagens. Além disso, o sistema deveria poder adicionar e remover usuários, e emails, sem limitações de quantidades pré-definidas.

Para cumprir essa proposta foram empregadas duas estruturas de dados do tipo lista encadeada, ou seja, que usam alocação dinâmica de memória, podendo assim atender as demandas do trabalho. Além disso, essas estruturas permitem a remoção de itens individuais da lista sem a necessidade de realocar ou mudar os outros itens que a compõem, se provando como escolha eficaz de estrutura para o servidor. Mais detalhes sobre essa implementação serão dados a seguir.

2 Método

O programa foi desenvolvido na linguagem C++, e compilado pelo G++ da GNU Compiler Collection, usando o WSL. O computador utilizado tem como sistema operacional o Windows 11, com 12 GB de RAM.

2.1 Implementação

Foram criadas seis classes: **Email, CélulaEmail, Inbox, Usuário, CélulaUsuário e Sistema.** Agora, serão abordadas com mais detalhamento cada uma.

2.1.1 Email

Essa classe armazena uma string **texto**, que guarda a mensagem do email, e um inteiro **pri**, referente à prioridade do e-mail.

Suas funções são um construtor da classe, que inicializa o o texto como uma string vazia e a prioridade como -1. Além disso, essa classe tem funções get e set para ambos atributos.

2.1.2 CelulaEmail

Essa classe armazena um item **email** do tipo Email, e um ponteiro **prox**, que aponta para outra CélulaEmail. Sua única função é um construtor que inicializa prox como NULL. Ela é uma classe amiga de Inbox.

2.1.3 Inbox

Essa classe representa a caixa de entrada, e é uma lista encadeada composta por células do tipo CélulaEmail. Seus atributos são dois ponteiros para o tipo CelulaEmail. O ponteiro **primeiro** aponta para a cabeça da lista, que será sempre uma célula vazia. Já o atributo **último** aponta para a última célula da lista. Ela tem também o atributo do tipo inteiro chamado **tamanho**, que contabiliza o número de itens da lista. As funções dessa classe são as seguintes:

- **Inbox():** construtor da classe, aloca memória para o atributo primeiro e aponta o atributo último para a cabeça da lista. Inicializa tamanho como zero.
- **void limpar():** é auxiliar ao destrutor e percorre a lista, a partir de **primeiro→prox**, e desaloca a memória usada por cada célula.
- **∼Inbox():** destrutor da classe, chama a função limpar e desaloca a memória de primeiro.
- **void inserir_começo(Email email):** cria uma nova CelulaEmail que tem como conteúdo o email passado como parâmetro e a insere no começo da lista.
- **void inserir_prioridade(Email email):** se a lista estiver vazia chama função inserir_começo para alocação da célula, senão, percorre a lista verificando a prioridade de cada email. Ao identificar a posição correta para o email(emails são ordenados por prioridade decrescente no Inbox), cria uma nova CelulaEmail, que tem como conteúdo o email passado como parâmetro, e a insere nessa posição.
- **Email remover_começo():** apaga o primeiro email da lista, e a faz o ponteiro primeiro apontar para a célula seguinte. Retorna uma cópia do item email armazenado na célula apagada.
- **int get_tamanho():** retorna o tamanho da lista.

2.1.4 Usuario

Essa classe tem como atributos um inteiro **id** e um ponteiro **inbox** para o tipo Inbox. Suas funções são um construtor, que inicializa id como -1 e aloca memória para o ponteiro inbox, um destrutor, que ao ser chamado desaloca memória de inbox, e, por fim, funções get e set para o id, e uma função get para inbox.

2.1.5 CelulaUsuario

Essa classe armazena um item **usuario** do tipo Usuario, e um ponteiro **prox**, que aponta para outra CelulaUsuario. Sua única função é um construtor que inicializa seu ponteiro como NULL. Ela é uma classe amiga de Sistema.

2.1.6 Sistema

Essa classe representa o servidor de e-mails, e é uma lista encadeada composta por células do tipo CelulaUsuario. Seus atributos são dois ponteiros para o tipo CelulaUsuario. O ponteiro **primeiro** aponta para a cabeça da lista, que será sempre uma célula com um item vazio. Já o atributo **último** aponta para a última célula da lista. Ela tem também o atributo do tipo inteiro chamado **tamanho**, que contabiliza o número de itens da lista. As funções dessa classe são as seguintes:

- **Sistema():** construtor da classe, aloca memória para o atributo primeiro e aponta o atributo último para a cabeça da lista. Inicializa tamanho como zero.

- **void limpar():** é auxiliar ao destrutor e percorre a lista, a partir de **primeiro→prox**, e desaloca a memória usada por cada célula.
- **∼Sistema():** destrutor da classe, chama a função limpar e desaloca a memória de primeiro.
- **CelulaUsuario* pesquisar_id(int id):** percorre a lista, verificando se id do usuário da célula corresponde ao id passado como parâmetro, retornando endereço da célula no final. Ao não encontrar célula com id correspondente, retorna NULL.
- **void imprimir_erro(int id):** imprime a seguinte mensagem de erro ao ser chamada "ERRO: CONTA ID NAO EXISTE".
- **void adicionar_usuario(int id):** verifica se já existe alguma célula que tenha um usuário com o ID passado como parâmetro. Caso tenha, encerra a função com a mensagem de aviso "ERRO: CONTA ID JA EXISTENTE". Caso contrário, cria uma nova CelulaUsuario, atribui id ao usuario dela, e a insere no final da lista. Por fim, adiciona um ao tamanho da lista, imprimindo uma mensagem de confirmação de criação do usuário no terminal.
- **void remover_usuario(int id):** procura pela célula com ID de valor id. Ao encontrá-la, faz com que a célula anterior a ela aponte para sua célula posterior, e depois a deleta. Por fim, diminui um do tamanho da lista imprimindo uma mensagem de confirmação de remoção do usuário no terminal. Caso não encontre a célula procurada, chama a função imprimir_erro, avisando programa que ao usuário não existe.
- **void acessar_inbox(int id):** verifica se a célula com ID passado como parâmetro existe ou não, caso não exista, manda chama imprimir_erro e encerra a função. Caso exista, ela usa a função get_inbox para acessar a caixa de entrada do usuário da célula. Depois, ela verifica o tamanho da caixa de entrada, para saber se ela possui mensagens ou não. Se a caixa de entrada estiver vazia, imprime a mensagem "CAIXA DE ENTRADA VAZIA", caso contrário, ela chama a função remover_começo para obter a mensagem do primeiro email na caixa entrada, e depois a imprime no terminal.
- **void entregar_mensagem(CelulaUsuario* celula, Email email):** a partir da célula passada como parâmetro, obtém endereço da caixa de entrada de seu usuário. Chama a função inserir_prioridade, passando o email como parâmetro. Por fim, confirma a entrega da mensagem imprimindo "OK: MENSAGEM PARA ID ENTREGUE", no terminal.

2.2 Programa principal - Main

O programa principal faz a leitura do arquivo passado como parâmetro junto ao executável. Se verificar que argv[1] for nulo, aborta o programa.

Caso contrário, ele cria o objeto **sistema** do tipo Sistema, e abre o arquivo, fazendo sua leitura palavra por palavra, usando o operador >>. Assim ele verfica se a palavra lida coincide com alguma das palavras de comando **"CADASTRA", "CONSULTA", "ENTREGA"** ou **"REMOVE"**. Cada palavra indica um roteiro de ações diferente. Assim, caso coincida com alguma delas, esse roteiro é seguido. A leitura do arquivo se dá até atingir EOF(end-of-file).

Para fazer essa leitura são usadas a seguintes variáveis auxiliares: string aux, string mensagem, CelulaUsuario* celula, int id, int pri e Email email.

2.2.1 CADASTRA

Lê o valor de ID do arquivo e adiciona o usuário ao sistema chamando a função adicionar_usuario.

2.2.2 CONSULTA

Lê o valor de ID do arquivo e acessa a caixa de entrada do usuário chamando a função acessar_inbox.

2.2.3 ENTREGA

Lê o ID do usuário e a prioridade PRI da mensagem. Depois, verifica se existe um usuário com o valor de ID informado usando a função pesquisar_id, salvando o endereço da célula na variável célula. Caso exista, armazena o texto da mensagem na variável de tipo string mensagem. Depois, associa a mensagem e prioridade informados à variável email. Por fim, chama a função entregar_mensagem, passando a célula e email como parâmetros.

2.2.4 REMOVE

Lê o valor de ID do arquivo e chama a função remover_usuario, passando id como parâmentro.

3 Análise de Complexidade

Nesta seção, será apresentada a análise de complexidade de tempo para os métodos apresentados na seção 2. O valor n aqui significa o tamanho da lista encadeada.

- Todas as funções construtoras, funções get e funções set do trabalho tem complexidade $O(1)$, pois realizam sempre o mesmo número de operações ao serem chamadas.
- **void limpar() - presente na classe Inbox e na classe Sistema:** essa função é idêntica em ambas classes, e tem complexidade $O(n)$, pois precisa percorrer toda a lista para deletá-la.
- **∼Inbox e ∼Sistema:** essas funções são idênticas, e tem complexidade $O(n)$, pois chamam a função limpar e executam além disso somente a ação de deletar a cabeça da lista.
- **void inserir_prioridade(Email email) - da classe Inbox:** tem complexidade assintótica $O(n)$, pois em seu pior caso, terá de percorrer a lista toda para verificar a posição correta do email a ser inserido.

As seguinte funções a serem analisadas são da classe Sistema:

- **CelulaUsuario* pesquisar_id(int id):** tem complexidade assintótica $O(n)$, pois em seu pior caso, terá de percorrer a lista toda para obter o endereço da célula do usuário que está sendo procurado.
- **void imprimir_erro():** tem complexidade $O(1)$, pois sempre executa uma única ação, que é imprimir a mesma mensagem no terminal.
- **void adicionar_usuario(int id):** tem complexidade $O(n)$, pois precisa sempre chamar a função pesquisar_id, que tem essa complexidade, para verificar se pode ou não adicionar o novo usuário ao sistema. Se puder adiconá-lo, irá executar sempre o mesmo numero de passos adicionais.
- **void remover_usuario(int id):** tem complexidade $O(n)$, pois em seu pior caso tem que percorrer a lista totalmente para encontrar usuário a ser deletado.
- **void acessar_inbox(int id):** tem complexidade $O(n)$, pois precisa sempre chamar a função pesquisar_id, que tem essa complexidade, para verificar se existe ou não um usuário do id passado, antes de fazer uma consulta a sua caixa de entrada. Se ele assistir, as funções

adicionais que são chamadas para acessar e imprimir a primeira mensagem da sua caixa de entrada tem complexidade $O(1)$.

- **entregar_mensagem(CelulaUsuario *celula, Email email):** complexidade $O(n)$ pois além de chamar a função inserir_prioridade, de complexidade $O(n)$, executa somente funções de complexidade constante $O(1)$.

  Agora, falando especificamente do programa principal:

- **int main(int argc, char ** argv):** a função main lê um arquivo de texto, com n linhas. Essa leitura é feita dentro de um while, e nele temos ifs que conferem se a palavra inicial da linha do arquivo coincide com as palavras de comando "CADASTRA", "REMOVE", "ENTREGA" ou "CONSULTA". Dessa forma, para cada linha ele terá, no pior caso, de fazer quatro verificações de if. Assim a função tem complexidade $O(n)$. Sendo n aqui a quantidade de linhas do arquivo.

  Vale ressaltar que, dentro de cada if há uma chamada de função. As funções que podem ser chamadas são da classe Sistema, como adicionar_usuario, remover_usuario, entre outras. Essas funções tem complexidade dependente do tamanho lista encadeada. A análise de complexidade delas já foi feita acima.

4 Estratégias de Robustez

O programa feito depende de entradas do usuário, para saber que comandos seguir. Logo, foi necessário pensar em que erros ele poderia cometer, e em como lidar com eles. Os erros possíveis são:

- usuário do programa solicitar a criação de um usuário com um ID que já está sendo usado;
- usuário do programa solicitar a remoção ou consulta de uma conta que não existe;

Como estratégia de programação defensiva, foram estabelecidos parâmetros de verificação, que retornavam mensagens de erro quando o usuário pedia algo impossível, como nos exemplos acima.

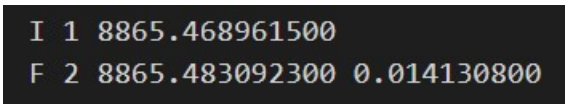
Como foi estabelecido que o programa deveria apenas advertir o usuário do erro, e continuar executando, as estratégias de robustez foram pensadas apenas com o objetivo de avisar o usuário somente, e não de corrigir entradas, ou seja, entradas erradas fazem com que o programa gere um aviso, e são simplesmente desconsideradas.

Por isso foi criada a função imprimir_erro, que informa que a conta com ID passado pelo usuário não existe. Ela é uma função auxiliar nas funções acessar_inbox e remover_usuario da classe Sistema.

5 Análise experimental

Para a análise experimental foi utilizado o memlog para registrar o tempo de execução do programa. O registro de tempo do programa sempre é feito ao rodar o programa, e cria-se o arquivo "log.out" na pasta bin, onde essas informações são salvas.

Abaixo, encontra-se um print do registro de tempo do programa ao ser executado com um arquivo de 1000 linhas fornecido pelos monitores, o extra_input_6.txt.



6 Conclusão

Por fim, verifica-se que a escolha de listas encadeadas para realizar esse trabalho foi correta, visto que essa estrutura de dados pôde atender perfeitamente as exigências do trabalho.

Além disso, gostaria de dizer que achei muito interessante fazer uma aplicação usando a estrutura de lista. Primeiro achei que eu precisaria de criar o TAD com todas as funções mostradas em aula, mas não foi bem assim. Apesar de algumas funções serem obrigatórias para o bom funcionamento e caracterização correta da lista encadadeada, tive o trabalho de pensar e selecionar as funções que seriam necessárias nesse programa, além de modificá-las de modo a atender as exigências específicas do trabalho.

Foi interessante criar uma lista encadeada que continha itens que tinham outra lista encadeada (a lista Sistema contém Usuários, e Usuários contém a lista Inbox). A classe Sistema foi usada para gerenciar todos os usuários. Essa centralização permitiu que o main ficasse simples e conciso. Na classe usuário, a lista Inbox tinha uma função mais especializada, de gerenciar as mensagens específicas de determinado usuário. Aplicando a lista para esses dois usos diferentes, aprendi de forma mais prática como essa estrutura de dados é uma base, e que cabe ao cientista da computação explorar seu uso de formas diferentes e criativas.

7 Instruções de Compilação e execução

Para compilar o programa, deve-se seguir os seguintes passos:

- Acessar o diretório TP1;
- Utilizando um terminal, digitar make, de forma a gerar o arquivo executável **run.out**;
- Chamar o executável ./run.out, junto a um arquivo.