Trabalho Prático 1

Filipe de Araújo Mendes

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

flipeara@ufmg.br

1. Introdução

O problema contido no trabalho prático 1 era a criação de um sistema de e-mails que seria capaz de criar e remover usuários, cada qual teria sua própria caixa de entrada com e-mails não lidos e poderiam enviar e-mails para outros usuários, com cada e-mail tendo sua própria prioridade, fazendo-os aparecer ou antes, se a prioridade for maior que a de outro e-mail, ou depois, se a prioridade for menor que a de outro e-mail, na caixa de entrada do usuário.

Com esse objetivo foi feito um plano de desenvolvimento, utilizando a linguagem C++, que envolvia 5 tipos abstratos de dados (TADs), cada um com seus métodos (o que o TAD faz) e seus atributos (as informações que ele guarda) próprios, visando auxiliar e possibilitar a resolução do problema.

2. Método

2.1. Main.cc

O arquivo main.cc é o responsável por iniciar o servidor, que então realiza, de fato, todas as operações descritas na entrada do programa.

Na hora de iniciar o programa, são passados alguns argumentos, sendo responsabilidade da função main registrá-los e executá-los. Além disso, a função main fica encarregada de iniciar o memLog, para a análise em tempo de execução do programa, abrir o arquivo de texto que contém as operações a serem realizadas e chamar o método manage que realmente fica encarregado da execução das operações contidas na entrada.

2.2. Estruturas de dados:

Como mencionado anteriormente, foi escolhida a divisão da resolução em 5 tipos de abstratos distintos, a classe Server e duas listas encadeadas, Inbox e UserList, cada uma com seu nó, Message e User, respectivamente.

2.2.1. Message (Mensagem)

A Message é a estrutura de dados menos abrangente criada, não incluindo em seus atributos TADs diferentes, ao contrário do resto das classes. Além disso, ela é um nó da Inbox (unidade básica de uma lista).

Cada Message possui 3 atributos. Priority, uma variável do tipo inteiro, representa a prioridade da mensagem, que definirá a ordem de e-mails na caixa de entrada. Content, do tipo string, armazena a mensagem que será lida pelo usuário a quem foi enviado o e-mail. NextMessage, por sua vez, é um ponteiro (armazena o endereço de memória onde está localizado/a outro TAD ou variável) para outra variável do tipo Message, que seria a próxima mensagem na caixa de entrada do usuário. Essa propriedade de um elemento apontar para o próximo na lista é o que define a lista encadeada. Todos os atributos são privados, o que significa que não é possível acessá-los ou modificá-los fora da estrutura Message.

Como métodos, Message tem o construtor, getters e setters. O construtor age na hora de criação do objeto, com instruções para sua inicialização. No caso da Message, são passados para o método os valores para a prioridade da Message, assim como o conteúdo do e-mail, já o atributo nextMessage é inicializado como um ponteiro nulo, para indicar que em momento de criação, a mensagem não possui próxima mensagem. Os getters e setters são necessários devido ao fato de os atributos serem privados. Por não ser possível alterar os atributos fora da estrutura Message, são criados métodos auxiliares, contidos na estrutura, que recebem como parâmetro valores para os quais alterar os atributos do Message (setters), ou que retornam os valores atuais dos atributos (getters).

Nesse caso, existe um get para cada atributo (GetPriority, GetContent e GetNextMessage), mas somente um set, o SetNextMessage, que é usado para mudar a próxima mensagem na lista.

2.2.2. Inbox (Caixa de Entrada)

A estrutura Inbox é uma lista encadeada, com a estrutura Message como nó, como mencionado anteriormente. Assim, possui poucos atributos e alguns métodos com intuito diferente dos da Message.

Assim como no tipo abstrato de dado anterior, a Inbox possui somente atributos privados, sendo eles firstMessage (primeira mensagem da lista), lastMessage (última mensagem da lista), ambos ponteiros para variáveis de tipo message, usados para declarar onde começa a lista, assim como onde ela termina., além de um atributo size, que é uma variável do tipo inteiro, a qual armazena o tamanho da lista.

A Inbox possui um construtor que inicializa firstMessage e lastMessage como ponteiros nulos, assim como o valor de size é inicializado como 0, já que, em momento de criação, ela não possui nenhuma

mensagem. Possui também um destrutor, o qual realiza a operação contrária de um construtor, realizando operações na hora de destruição do objeto, ao invés de criação. O destrutor da Inbox desaloca todas as mensagens da lista em momento de destruição desta, sendo necessário devido à alocação dinâmica..

Além desses, existe o getSize, para pegar o tamanho da lista, o addMessage, que adiciona uma nova mensagem na caixa de entrada, com a posição na lista dependendo da prioridade da mensagem e o readFirstMail, que retorna o conteúdo da primeira mensagem da Inbox, retirando-a da caixa de entrada do usuário.

2.2.3. User (Usuário)

Assim como Message, User é um nó de uma lista, mas da UserList, possuindo então atributos e métodos similares aos da Message.

Dentre os atributos, todos privados, estão ID, variável de tipo inteiro que representa o identificador de cada usuário dentro do sistema, uInbox, ponteiro para variável de tipo Inbox, que é a caixa de entrada do usuário em questão, e, por fim, o nextUser, que existe pelo mesmo motivo do atributo nextMessage, do TAD Message.

O construtor inicializa o nextUser como nulo, ID como valor passado por parâmetro e uInbox como o endereço de uma nova caixa de entrada criada. O destrutor, por sua vez, só deleta uInbox, por ser o único atributo alocado dinamicamente. O getNextUser retorna o próximo usuário na lista e o setNextUser altera o valor deste.

Existem outros métodos com funções mais específicas, as quais não existiam em Message. InboxIsEmpty retorna ou verdadeiro, se a lista estiver vazia, ou falso, se não estiver vazia. ReceiveMessage adiciona uma mensagem passada por parâmetro à caixa de entrada do usuário (uInbox) e readMail chama o método readFirstMail da Inbox e retorna o conteúdo da primeira mensagem da caixa de entrada do usuário.

2.2.4. UserList (Lista de Usuário)

Assim como Inbox, UserList é uma lista encadeada, cujo nó é User. Por isso, seus atributos e métodos são bastante similares.

Todos os atributos são privados, possuindo, igual Inbox, os atributos size, firstUser e lastUser que realizam os mesmos papéis dos atributos size, firstMessage e lastMessage, respectivamente, com uma pequena diferença nos últimos dois, que agora são ponteiros para variáveis do tipo User, não Message.

O construtor e destrutor são feitos da mesma maneira de Inbox, addUser adiciona um novo usuário à lista, sempre na ultima posição, removeUser remove o usuário de ID especificado da lista, getUser retorna o

usuário de ID especificado e userExists verifica, em toda a lista, se existe um usuário com certo ID, retornando verdadeiro caso exista, falso caso contrário.

2.2.5. Server (Servidor)

Esse é o TAD que lê o arquivo de entrada e realiza as operações contidas nele.

Só possui um atributo, o list, do tipo UserList, que vai armazenar todos os usuários existentes.

O construtor não possui nenhuma especificação sobre como será inicializado o Server, e existem mais 5 métodos:

- addUser - chama userExists para verificar a existência de um usuário e, caso esse não exista, chama addUser para criá-lo.
- removeUser - chama userExists para verificar a existência de um usuário e, caso esse exista, chama removeUser para removê-lo.
- sendMailTo - pega a prioridade e conteúdo do e-mail no arquivo de entrada, chama userExists para verificar a existência de um usuário e, caso esse exista, cria e envia a mensagem por meio do receiveMessage.
- readMailFrom - chama userExists para verificar a existência de um usuário e, caso esse exista, chama inboxIsEmpty para verificar se a caixa de entrada do usuário possui mensagens, se sim, imprime na saída a mensagem com ajuda de readMail.
- Manage - lê o arquivo de entrada, pega os comandos e o ID do usuário e chama, dependendo da operação desejada, os métodos addUser, removeUser, sendMailTo ou readMailFrom.

3. Análise de Complexidade

3.1. Espaço

A complexidade de espaço está associada à quantidade de memória utilizada pelas variáveis do programa.

O objeto abrangente, o Server, possui como atributo, 1 outro objeto do tipo UserList, o qual ocupa uma quantidade de espaço fixa por ter somente 3 atributos de tamanhos invariáveis, mas a lista em si ocupa uma quantidade de espaço diretamente proporcional à quantidade de usuários contidos nela. Cada User, por sua vez, ocupa um espaço fixo, mas cada um deles também possui uma caixa de entrada (uInbox). A Inbox, de maneira análoga à UserList, ocupa espaço fixo, mas a lista ocupa uma quantidade de espaço diretamente proporcional à quantidade de mensagens contidas nela. Parecido com o User, Message também ocupa um espaço quase fixo, porém

o atributo content possui tamanho variável, dependendo do tamanho da mensagem.

Com isso, conclui-se que a complexidade de espaço sempre será limitada superiormente pela soma dos tamanhos de todas as caixas de entrada existentes, já que todo usuário possui exatamente uma caixa de entrada (nunca usará mais espaço que a soma dos tamanhos de todas as caixas de entrada, multiplicado por uma constante), ou seja, a complexidade de espaço é O(s), sendo s a soma dos tamanhos das caixas de entrada ou, sendo n o número de usuários e m o tamanho de cada lista de mensagens, supondo que todas as listas de mensagem possuem a mesma quantidade de mensagens, O(n*m).

3.2. Tempo

Para a complexidade de tempo, analisaremos os métodos que necessitam varrer as listas, pois esses não possuem custo constante. Tais métodos são addMessage, removeUser, userExists e getUser, desconsiderando os destrutores de UserList e Inbox, que possuem custo linear.

AddMessage**:**

- Menor custo: quando a prioridade da mensagem nova é maior que a prioridade de qualquer outra mensagem na caixa de entrada ou quando a caixa de entrada está vazia. O(1).
- Maior custo: quando a prioridade da mensagem nova não é maior que a de nenhuma outra mensagem na caixa de entrada. Precisa percorrer a lista inteira para chegar na última posição. O(n).

RemoveUser**:**

- Menor custo: quando o ID informado se refere ao ID do primeiro usuário na lista. O(1).
- Maior custo: quando o ID informado se refere ao ID do último usuário na lista. O(n).

UserExists**:**

- Menor custo: quando o ID informado se refere ao ID do primeiro usuário na lista.
- Maior custo: quando o ID informado se refere ao ID do último usuário na lista ou o usuário não existe. O(n).

GetUser**:**

- Menor custo: quando o ID informado se refere ao ID do primeiro usuário na lista. O(1).
- Maior custo: quando o ID informado se refere ao ID do último usuário na lista. O(n).

Contudo, o custo do programa não é linear pela necessidade deleitura do arquivo de entrada. O método manage, do Server, possui também custo O(n), já que a operação de leitura da entrada tem custo constante, mas multiplicado pela quantidade de leituras que são necessárias, a qual aumenta com o número de linhas do arquivo. Dessa forma, o Server fica responsável por chamar métodos de diferentes custos m vezes durante a execução do programa, fazendo do pior caso aquele em que o Server vai chamar m vezes um método linear, por isso o custo de tempo do programa é O(m*n).

4. Estratégias de Robustez

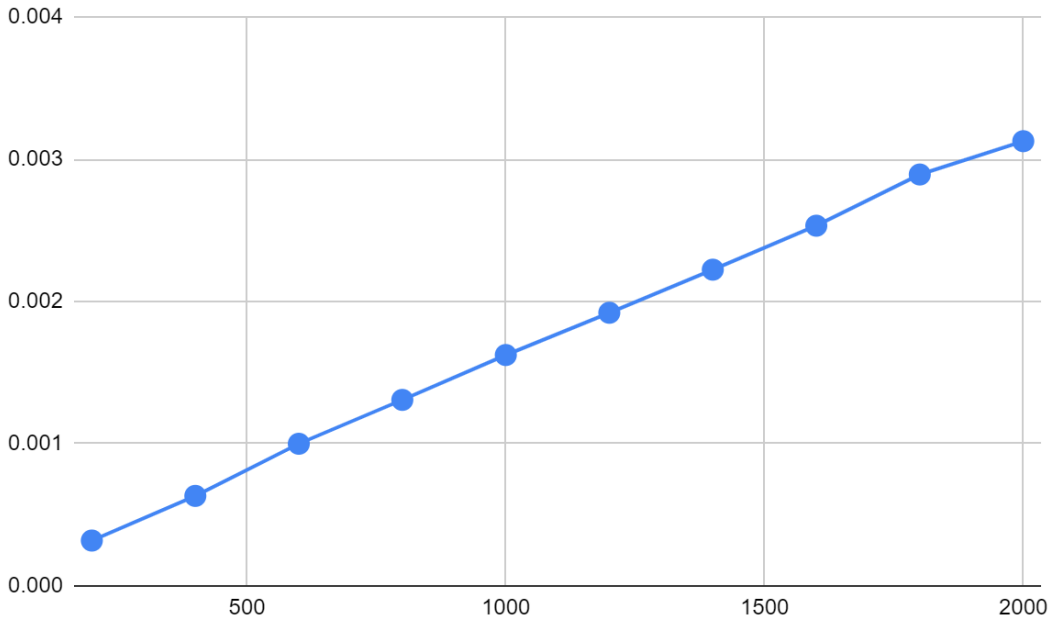
Para evitar a interrupção desnecessária do programa, foi adicionada uma função para checar se os argumentos contendo o nome de entrada do programa contém a extensão do arquivo requerida pelo programa, não interrompendo a execução se esse não for o caso.

5. Análise Experimental

Foi analisado o tempo de execução do programa com o uso da biblioteca memLog e foram realizados 2 tipos de testes, ambos testando em diferentes tamanhos do arquivo de entrada, começando em 200 linhas e aumentando em 200 até chegar nas 2000:

5.1. Teste 1

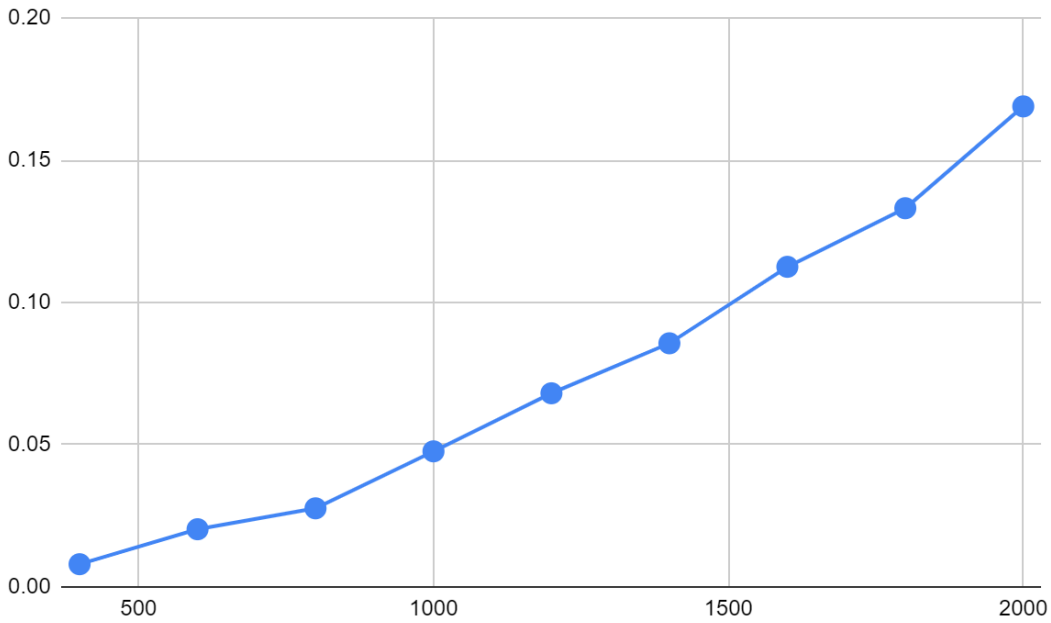
O primeiro teste foi com a entrada tentando remover um usuário com ID 1 em todas as operações, assim, o método relevante desse teste foi o userExists, sendo sempre o pior caso, mas como o tamanho da lista de usuários, além de ser 0, não é alterado, o custo dessa função é sempre mínimo, constante. Por nunca existir o usuário a ser retirado na lista, as operações não passam pelo removeUser.



Como é possível perceber, o gráfico se assemelha com uma reta, apesar do custo constante do método userExists durante toda a execução, demonstrando o custo linear da leitura da entrada, que é a única coisa sendo alterada entre os vários testes.

5.2. Teste 2

O segundo teste foi com a entrada criando um usuário e adicionando exatamente a mesma mensagem na caixa de entrada dele, assim, seria sempre o pior caso do método addMessage. O eixo X representa o número de linhas no arquivo de entrada e o Y, o tempo de execução.



É possível perceber que esse gráfico se assemelha com uma curva, a qual é causada pelo custo linear da leitura da entrada, que multiplicado pelo custo de outras funções também de custo linear, aumenta o custo do programa para O(n*m).

5.3. Observações Importantes

É necessário lembrar que os testes não foram realizados em um sistema isolado, portanto, realiza outras tarefas simultaneamente e, desse modo, não é capaz de reproduzir uma performance constante, podendo assim causar variações durante os testes.

Tal evento ocorreu durante os testes, por isso, foram repetidos algumas vezes com o intuito de obter valores intermediários que representassem, de fato, o comportamento do custo de tempo da execução dos diferentes testes, entretanto, como é possível enxergar, ainda assim

houveram oscilações, não reproduzindo, no gráfico 1, uma reta exata ou, no gráfico 2, uma curva perfeita e estável.

6. Conclusões

No trabalho, foi criado um programa para o gerenciamento de usuários e e-mails os quais os usuários podem enviar e ler de suas caixas de mensagem.

Por meio da utilização das listas encadeadas, foi possível aprender mais sobre tal estrutura de dados, tentando implementar duas listas distintas, com dados diferentes a serem guardados e fazendo manipulações que, apesar de similares, proporcionaram um entendimento maior sobre as possíveis implementações. Para citar um exemplo, os métodos de adicionar usuário ou mensagem nas listas, são diferentes já que no primeiro, o usuário sempre é adicionado no final da lista, mas no segundo, as mensagens podem ser adicionadas em lugares diferentes da lista.

Além disso, a análise de complexidade de tempo foi um desafio, uma vez que o custo da leitura da entrada do programa interferiu com o custo total de execução, fator que, por ter causado dúvidas, auxiliou no melhor aprendizado de tal análise.

7. Bibliografia

Para funções padrão de c++, foi utilizado como base o link: https://cplusplus.com/forum/

Para detalhes sobre o uso do MemLog, foi utilizado o link: https://docs.google.com/document/d/1h-yeWDz2FjBVvMgI1gfXyV_ogOmwlpneKEfq4yHihoA/edit

8. Instruções para Compilação e Execução

Para a compilação do programa em sistema Linux, primeiro é necessário abrir uma janela do terminal dentro da pasta TP. Após feito isso, basta digitar a linha de comando “make” e o programa será compilado.

Para sua execução em ambiente Linux, é necessário também abrir a janela do terminal na pasta TP, para então digitar a linha de comando “./bin/run.out ARGUMENTOS”, onde ARGUMENTOS são os argumentos a seguir:

- “-i entrada” ou “-i entrada.txt” - O argumento “-i” é seguido do nome do arquivo de texto que se deseja ler, que deve estar no diretório TP, com ou sem o “.txt”, que indica a extensão.
- “-p registro” ou “-p registro.out” - O argumento “-p” é seguido do nome do arquivo em que se deseja fazer o registro de desempenho, com ou sem o “.out”, que indica a extensão.

Dessa forma, o programa será executado, realizando as operações desejadas e imprimindo as saídas do programa no terminal.