Trabalho Pratico 1

Servidor de e-mails

João Victor Evangelista Cruz

Universidade Federal de Minas Gerais (UFMG)

Belo Horizonte – MG – Brasil

joaovecruz@ufmg.br

1. Introdução

O problema proposto foi implementar um simulador de servidor de e-mails para a Google de forma que milhões de pessoas possam utilizá-lo. O servidor deve dar suporte ao cadastro de novas contas de usuários, assim como a remoção de contas já existentes. Também deve possibilitar o envio e consulta de mensagens de e-mail de usuários já cadastrados.

2. Método

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

2.1. Estrutura de Dados

A implementação do programa teve como base a estrutura de dados de uma lista encadeada. Tem-se dois tipos de listas diferentes, uma para organizar as contas criadas e outra para organizar os e-mails de cada conta. Ambas armazenam TADs células, que estes por sua vez, armazenam as contas e e-mails, respectivamente.

Como visto nas aulas e nos materiais disponibilizados, as listas possuem elementos chamados “primeiro”, que não armazena nada e serve para apontar para o início da lista, e “ultimo”, que aponta para o último elemento da lista.

A maioria das funções das listas, como construtores, destrutores, assim como funções de inserção e remoção, de um ou todos os elementos da lista foram adaptados dos algoritmos vistos nos materiais disponibilizados

em aula e no moodle. Além desses, há na lista de contas, a função “Verifica”, que tem como parâmetro um inteiro “ID”, e a função itera sobre a lista verificando se a lista possui algum elemento já existente com esse ID. Além disso, há duas funções específicas dessa implementação. A primeira é uma função responsável por enviar as mensagens de e-mail para os usuários, armazenando-as em suas caixas de e-mail, organizando-as de acordo com a prioridade e a ordem de envio de cada e-mail. Já a segunda faz a consulta do próximo e-mail da caixa. Além dessas, há também funções de Get para retornar variáveis das listas.

2.2. Classes

Para a modularização da implementação, foram criadas seis classes. Duas dessas foram as listas que foram abordadas acima. Outras duas foram as células das respectivas listas, que possuem um ponteiro com o nome de “prox”, que aponta para a próxima célula da lista. As células de conta também armazenam um elemento do tipo Conta, assim como as células de e-mail armazenam um elemento do tipo Email. As duas últimas classes são essas que acabaram de ser citadas: a classe Conta, que possui uma variável do tipo inteiro representando o ID da conta, e um ponteiro “caixa” que aponta para a caixa de e-mails dessa conta. A classe Email possui variáveis “conteúdo”, do tipo string que armazena a mensagem do e-mail, e “prioridade”, do tipo inteiro, que representa a prioridade do e-mail. Além disso, ambas possuem funções que auxiliam outras funções das listas.

2.3. Ordem de consulta dos e-mails

Há no programa a implementação de um sistema de organização dos e-mails que chegam nas caixas de e-mail. Tal implementação segue tais regras:

- E-mails com prioridade mais alta devem ser mostrados antes do que e-mails com prioridade menor. Essa prioridade é um número que varia de 0 a 9, sendo 0 a menor prioridade e 9 a maior.

- Quando há e-mails com prioridades iguais, deve-se respeitar a ordem de chegada. E-mails que chegaram antes devem ser mostrados antes do que e-mails que chegaram depois.

2.4. Funções

ListaContas:

bool Verifica(int ID): Função auxiliar da classe da lista de contas que recebe como parâmetro o inteiro ID e itera sobre os elementos da lista.

Retorna true se tiver alguma conta já existente na lista com esse ID e retorna false caso contrário.

void InsereID(int ID): Função da classe da lista de contas que recebe como parâmetro o inteiro ID e utiliza a função de verificação para ver se há alguma conta existente com esse ID. Se não houver, cria-se uma nova conta com esse ID e adiciona essa conta no início da lista, incrementa o tamanho da lista e escreve na tela que a operação teve sucesso. Se já houver uma conta com o ID na lista, não faz nada além de escrever na tela avisando que a operação foi malsucedida.

void RemoveID(int ID): Função que recebe o inteiro ID da conta que se deseja retirar da lista. Utiliza a função auxiliar para verificar se a determinada conta existe. Se sim, retira a conta da lista e a deleta e mostra na tela que a conta foi removida. Se não, só mostra na tela que a conta não existe.

void Limpa(): Função que remove e deleta todos os elementos da lista, além de definir a variável “tamanho” como 0.

void RecebeEmailID(int ID, std::string cont, int prior): Função que recebe como parâmetro o ID da conta que se quer mandar o e-mail, o conteúdo do e-mail e a prioridade. Verifica se existe uma conta com esse ID e caso positivo, cria um Email e insere na caixa de emails dessa conta, além de mostrar na tela que a mensagem foi entregue. Caso não exista a conta, mostra na tela que a conta é inexistente.

void LeEmailID(int ID): Função que recebe como parâmetro o ID da conta que se quer consultar o e-mail. Verifica se a conta existe e caso exista, verifica se há algum e-mail na caixa de e-mails da conta. Se houver, imprime na tela a mensagem e remove o e-mail da caixa utilizando a função LeEmail() da lista de emails. Se a caixa estiver vazia, imprime na tela a situação. Se a conta não existir, também imprime na tela que a conta é inexistente.

CaixaDeEmail:

void RecebeEmail(Email e): Função da lista de e-mails que recebe como parâmetro o Email “e” e o insere na lista de acordo com as regras de prioridade e ordem de chegada já explicados, além de incrementar a variável de tamanho da lista.

void LeEmail(): Função que exibe o próximo email da lista e o remove, decrementando também a variável de tamanho da lista.

void Limpa(): Função análoga a função de mesmo nome da lista de contas.

Essas foram as principais funções. As demais funções somente auxiliam as funções principais, retornam valores das variáveis das classes ou são os construtores e destrutores de cada classe.

3. Análise de Complexidade

3.1. Tempo

Vamos começar a análise pela situação inicial de cadastrar uma conta. A função Verifica itera sobre todos os elementos da lista, então possui custo O(n). Para inserir um elemento na lista encadeada, o custo é O(1). Logo todo o processo custa O(n) + O(1) = O(n).

Para remover uma conta, utiliza-se também a função Verifica e o custo de remoção é o mesmo de inserção na lista encadeada. Além disso, tem que iterar na lista para achar a conta que possui o ID desejado para então removê-lo. O custo fica O(n) + O(n) + O(1) = O(n).

Na entrega dos e-mails, utiliza-se as mesmas duas iterações da remoção. Além disso, chama outra função que itera pela caixa de e-mails e adiciona o e-mail à caixa. Logo o custo é 3 O(n) + O(1) = O(n).

Na consulta dos e-mails, há também as duas interações citadas acima, para então imprimir o e-mail. Então também tem custo O(n).

Por último, a função de limpar a lista também itera sobre a lista deletando os itens, então também tem custo O(n).

Portanto, o programa tem custo O(n).

3.2. Espaço

Temos uma lista podendo ter várias contas e cada conta tem uma outra lista, podendo ter vários e-mails armazenados. Em um caso extremo em que se tenha n contas e cada conta tenha n e-mails em sua caixa, temos $O(n*n) = O(n^2)$.

4. Estratégias de Robustez

Cada conta criada tem seu ID e ele deve ser diferente para ser possível diferenciar uma conta da outra na hora de enviar um e-mail para ela, deletar a conta ou acessar os e-mails dela. Por isso tiveram no código os seguintes mecanismos de defesa:

Na criação de uma nova conta, você indica um ID para a nova conta. Por isso, verifica-se antes se já existe uma conta com esse ID, para não haver

duas contas com a mesma identificação. Há um if que chama a função de verificar, e se a função retornar false, quer dizer que não há contas com aquele ID, logo pode-se proceder com a criação da conta. Porém, se retornar true, não se pode criar a conta, então gera uma indicação ao usuário do programa de que a conta não poderia ser criada.

Já na remoção de uma conta, deve-se verificar antes se a conta que se quer remover está inserida na lista assim como se há algum elemento na lista, por isso tem um if com a função de verificação junto com a condicional “tamanho > 0”. Ou seja, se houver algo na lista e se tiver a conta que se quer remover, então pode-se remover a conta. Se não, ocorre um erro, pois a remoção não pode ocorrer.

A mesma coisa acontece na função de entregar uma mensagem a uma conta. Deve-se existir a conta que é o alvo da mensagem. Por isso há também um if verificando se a conta existe. Se existir, procede-se para o envio do e-mail. Se não existir, gera um erro informando ao usuário que não a conta é inexistente.

Já na função de consulta de e-mail de uma conta, além de verificar que a conta alvo existe, ainda tem que verificar se há algum e-mail na caixa de mensagens dessa conta. Por isso tem inicialmente um if para verificar se a conta existe. Se não existir, gera um erro de conta inexistente. Se a conta existir, há ainda outro if verificando quantos e-mails há na caixa dessa conta. Se houver um ou mais e-mails na caixa, procede-se para a consulta da mensagem. Se não, informa ao usuário que a caixa está vazia.

5. Conclusão

Para o trabalho, foi necessário tanto a escolha e adaptação da estrutura de dados correta, que foi usada de duas maneiras diferentes, criando duas listas de elementos diferentes que se complementavam como também foi necessário desenvolver formas de organizar os elementos dessas listas de acordo com o que foi proposto, além de fazer a simulação desse servidor de e-mail, fazendo criações e remoções de contas, enviando e lendo mensagens.

Ambas as partes foram desafiadoras, pois, por mais que tivéssemos o material disponibilizado sobre estruturas de dados e listas, tivemos que adaptar estes para o determinado uso no trabalho, além de que tinha uma lista dentro de outra.

Ainda assim, a segunda parte foi especialmente difícil, pois foi preciso ter criatividade para desenvolver soluções relacionadas ao armazenamento e leitura corretos dos e-mails, assim como no cadastro e remoção de novas contas.

Com esse trabalho, fica nítido a importância da organização do código através da modularização, principalmente quando se tem códigos grandes e com várias classes, como esse e também a utilização do makefile, para poupar tempo e esforço na hora de compilar o programa e fazer testes. Além disso, percebe-se a importância de saber utilizar as estruturas de dados, principalmente a lista, nesse trabalho, pois sem isso, seria quase impossível desenvolver uma solução para o problema apresentado. O conhecimento das diferentes estruturas de dados é vital para programas de qualidade.

6. Bibliografia

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

- Cormen, T., Leiserson, C, Rivest R., Stein, C. Algoritmos – Teoria e Prática, 3a. Edição, Elsevier, 2012.

7. Instruções para compilação e execução

Para a compilação do programa, deve-se utilizar o comando “make all”, no terminal, estando na raiz da pasta TP.

Para a execução, deve se colocar o arquivo de teste na pasta TP, na raiz do projeto e digitar o comando estando na raiz do projeto, no terminal:

“./bin//main nome_arquivo.txt”, onde nome_arquivo.txt representa o nome do arquivo de testes.

Por exemplo, se o nome do arquivo for “testes.txt”, o comando deverá ser:

```
./bin//main testes.txt
```