Trabalho Prático 1

Servidor de emails

Matheus Guimarães Couto de Melo Afonso
Matrícula: 2021039450

Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

matheusgcma@ufmg.br

1. Introdução

O trabalho prático 1 consistiu em implementar uma simulação de um servidor de emails. No sistema, é possível realizar 4 operações: cadastro de usuários, remoção de usuário, envio de email e consulta de email. Cada email tem uma prioridade diferente, e eles são organizados em uma fila do de maior prioridade até o de menor prioridade.

A seção 2 contém uma explicação mais profunda sobre as estruturas de dados e métodos implementados, bem como informações sobre o ambiente de desenvolvimento. A seção 3 apresenta uma análise de complexidade do código, enquanto a seção 4 se aprofunda nas estratégias de robustez aplicadas no desenvolvimento. Na seção 5, mostro os resultados dos testes de desempenho do programa, e por fim a seção 6 conclui a documentação do trabalho.

2. Método

2.1. Organização do projeto

O projeto é estruturado no modelo sugerido pelos professores: a pasta **src** armazena os arquivos de código (*.cpp) e a pasta **include** têm os arquivos de cabeçalho (*.h). Ao compilar o código por meio do **Makefile**, que está na pasta raiz do projeto, os arquivos *.o são gerados no diretório **out** e o arquivo executável **run.out** é gerado na pasta **bin**.

2.2. Estruturas de dados

2.2.1. Email

A classe Email implementa um objeto que tem como atributos uma mensagem de texto e um valor entre 0 e 9 que representa a prioridade daquela mensagem, com

valores mais altos representando uma prioridade maior. Além disso, cada objeto da classe tem um apontador do tipo `Email *next`, que é responsável por encadear todos os elementos de uma caixa de entrada em uma fila de prioridades.

```
class Email {
  private:
    std::string msg;
    int priority;
    Email *next;

  public:
    Email();
    Email(std::string msg, int prio);
    ~Email();
    void setMessage(std::string msg);
    std::string getMessage();
    void setPrio(int prio);
    int getPrio();

    friend class CaixaDeEntrada;
};
```

Os métodos da classe Email são apenas setters e getters.

2.2.2. Caixa de Entrada

A classe CaixaDeEntrada implementa uma caixa de entrada de um serviço de email. Cada instância do objeto tem um id, de forma que dois objetos diferentes no servidor não podem ter o mesmo id, além de um ponteiro do tipo `Email *email_head`, que aponta para o email de maior prioridade recebido pela caixa de entrada.

```
class CaixaDeEntrada {
  private:
    int id;
    Email *email_head;

  public:
    CaixaDeEntrada();
    ~CaixaDeEntrada();
    void setId(int id);
    int getId();
    std::string consultaEmail();
    void recebeEmail(std::string msg, int prio);
    void limpaCaixa();
};
```

A CaixaDeEntrada implementa, além do setter e getter do *id*, três métodos:

- `consultaEmail()`: retorna a mensagem do `*email_head`, que corresponde ao primeiro elemento da fila, e exclui esse email da caixa de entrada.
- `recebeEmail(string msg, int prio)`: recebe como parâmetros uma string que corresponde à mensagem de um email e um inteiro que corresponde à prioridade do mesmo. Em seguida, cria um objeto Email com esses atributos e o adiciona na fila de acordo com a sua prioridade.

- `limpaCaixa()`: exclui todos os emails da caixa de entrada.

2.2.3. Servidor

A classe Servidor centraliza a implementação completa do servidor de emails: todas as interações entre o arquivo main e o servidor são feitas por meio de um objeto desta classe. O único atributo da classe é um ponteiro para uma struct do tipo `User`. Cada objeto do tipo `User` tem uma caixa de entrada e um apontador para outro objeto `User *next`, permitindo assim que o servidor armazene as diferentes instâncias de usuários dinamicamente em uma lista encadeada.

```
// Cada usuário tem uma caixa de entrada e aponta para um outro usuário
struct User {
    CaixaDeEntrada *caixa;
    User *next;
};

class Servidor {
  private:
    User *head;
    User *encontrarUsuario(int id);
    void printaCaixas();

  public:
    Servidor();
    ~Servidor();
    void criarUsuario(int id);
    void excluirUsuario(int id);
    void limpaServidor();
    void enviarEmail(int id, std::string msg, int prio);
    void consultarEmail(int id);
};
```

O Servidor implementa cinco métodos públicos:

- `criarUsuario(int id)`: verifica se existe um usuário com o id na lista, e caso contrário cria uma nova caixa de entrada vazia.
- `excluirUsuario(int id)`: verifica se existe um usuário com o id na lista, e caso exista desaloca a caixa de entrada dele e remove o usuário da lista.
- `limpaServidor()`: exclui todos os usuários do servidor.
- `enviarEmail(int id, string msg, int prio)`: verifica se existe um usuário com o id na lista, e caso sim chama a função `usuario->caixa->recebeEmail(msg, prio)`.
- `consultarEmail(int id)`: verifica se existe um usuário com o id na lista, e caso sim chama a função `usuario->caixa->consultaEmail()`.

Além desses, há a implementação do método privado `encontrarUsuario(int id)`, que é a chave para o funcionamento do servidor, uma vez que todos os métodos

públicos o utilizam. Esse método retorna um ponteiro para o usuário cuja caixa de entrada tenha o *id* passado por parâmetro. Quando não existe uma caixa com esse *id*, retorna `NULL`.

2.3. Ambiente de desenvolvimento

- Sistema operacional: WSL Ubuntu 20.4
- Linguagem de programação: C++
- Compilador: g++ Ubuntu 9.4.0-1ubuntu1
- Processador: Intel Core i5-9400F @2.90GHz
- Memória RAM: 2 x 8GB

3. Análise de Complexidade

A complexidade das funções do programa pode depender da quantidade de usuários cadastrados no servidor (m), da quantidade de email de um usuário (n) ou de ambos. Visto que existem 4 operações possíveis no servidor (cadastro de usuário, remoção de usuário, envio de email e consulta de email), a análise de complexidade será feita sobre cada uma dessas operações individualmente, para melhor entendimento. Além disso, uma vez que as 4 operações utilizam o método `encontrarUsuario`, encontrar a complexidade dele é de suma importância para analisar a complexidade das operações.

3.1. Complexidade de tempo

- `encontrarUsuario`: os usuários são armazenados em uma lista encadeada, e a função percorre a lista de forma linear. No pior caso (quando o usuário está na última posição da lista, ou quando não existe o usuário buscado), a função tem complexidade $O(m)$. Uma maneira de reduzir a complexidade dessa função seria armazenando os usuários em uma árvore binária ao invés de uma lista, mas não me considerei apto a realizar essa implementação.

- Cadastro de usuário: verifica, por meio do método `encontrarUsuario`, se já existe um usuário cadastrado com o *id* desejado ($O(m)$). Caso contrário, insere um novo usuário no início da lista ($O(1)$). Complexidade total:

$$O(m) + O(1) = O(max(m, 1)) = O(m)$$

- Remoção de usuário: verifica, por meio do método `encontrarUsuario`, se já existe um usuário cadastrado com o *id* desejado ($O(m)$). Caso exista, percorre a lista novamente buscando o usuário que precede o que queremos remover ($O(m)$) e desaloca a memória correspondente a ele, relacionando o anterior com o próximo para manter a lista encadeada. Complexidade total:

$$O(m) + O(m) = O(max(m, m)) = O(m)$$

- Envio de email: verifica, por meio do método `encontrarUsuario`, se já existe um usuário cadastrado com o *id* desejado ($O(m)$). Caso exista, insere o email na lista de emails, de acordo com a prioridade dele ($O(n)$ no pior caso, quando o email tem prioridade 0). Complexidade total:

$$O(m) + O(n) = O(max(m, n))$$

- Consulta de email: verifica, por meio do método `encontrarUsuario`, se já existe um usuário cadastrado com o *id* desejado ($O(m)$). Caso exista, remove o primeiro email da lista, que é o de maior prioridade ($O(1)$). Complexidade total:

$$O(m) + O(1) = O(max(m, 1)) = O(m)$$

A complexidade de tempo total do programa depende do número k de operações que o usuário deseja realizar, crescendo de forma linear de acordo com o aumento de k. Há também uma dependência do tipo das operações, visto que elas têm complexidades diferentes. Apesar disso, podemos dizer que, de maneira geral, a complexidade do programa é $O(k)$.

3.2. Complexidade de espaço

- `encontrarUsuario`: realiza apenas busca de um usuário na memória, sem escrever nada. Portanto, a complexidade de espaço é $O(0)$.

- Cadastro de usuário: verifica, por meio do método `encontrarUsuario`, se já existe um usuário cadastrado com o *id* desejado ($O(m)$). Caso contrário, insere um novo usuário no início da lista ($O(1)$). Complexidade total:

$$O(m) + O(1) = O(max(m, 1)) = O(m)$$

- Remoção de usuário: verifica, por meio do método `encontrarUsuario`, se já existe um usuário cadastrado com o *id* desejado ($O(m)$). Caso exista, percorre a lista novamente buscando o usuário que precede o que queremos remover ($O(m)$) e desaloca a memória correspondente a ele, relacionando o anterior com o próximo para manter a lista encadeada. Complexidade total:

$$O(m) + O(m) = O(max(m, m)) = O(m)$$

- Envio de email: verifica, por meio do método `encontrarUsuario`, se já existe um usuário cadastrado com o *id* desejado ($O(m)$). Caso exista, insere o email na lista de emails, de acordo com a prioridade dele ($O(n)$ no pior caso, quando o email tem prioridade 0). Complexidade total:

$$O(m) + O(n) = O(max(m, n))$$

- Consulta de email: verifica, por meio do método `encontrarUsuario`, se já existe um usuário cadastrado com o *id* desejado ($O(m)$). Caso exista, remove o primeiro email da lista, que é o de maior prioridade ($O(1)$). Complexidade total:

$$O(m) + O(1) = O(max(m, 1)) = O(m)$$

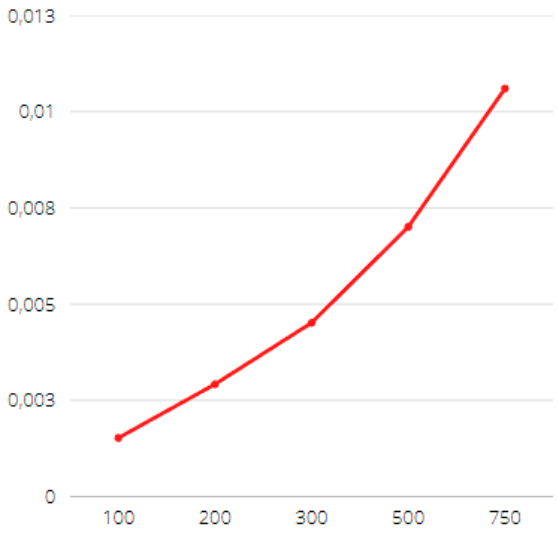
4. Estratégias de Robustez

A robustez de um código diz respeito à sua capacidade de funcionar corretamente mesmo em condições anormais, por meio de mecanismos de programação defensiva e tolerância a falhas. A robustez do programa foi implementada por meio da biblioteca **msgassert.h**, disponibilizada pelos professores de Estrutura de Dados.

- Durante a leitura dos comandos é lançado um *erroAssert* e a execução é abortada caso seja passada uma operação desconhecida pelo programa.
- Caso o usuário tente enviar um email com prioridade < 0 ou > 9, uma função auxiliar é utilizada para limitar o valor da prioridade.
- É lançado um *erroAssert* quando o id de usuário está fora do intervalo $[0,\ 1.000.000]$.

5. Análise Experimental

Na seção de análise experimental, são apresentados os resultados de testes feitos em cima do programa, que verificam fatores como tempo de execução, localidade de referência e distância de pilha. São utilizadas como ferramentas de análise as bibliotecas **memlog** e **analisamem**, disponibilizadas pelos professores da disciplina.



O gráfico acima ilustra o tempo de execução do programa conforme o número k de operações executadas aumenta. A curva que resulta da análise desses tempos de

execução representa uma função semelhante à uma função linear, o que foi previsto anteriormente pela análise de complexidade. A curva não é perfeitamente linear porque, como dito antes, o tempo de execução depende da natureza das operações.

6. Conclusões

O trabalho prático 1 tratou da implementação de um programa em C++ que simula um servidor de emails, capaz de realizar operações de cadastro de usuário, remoção de usuário, envio de email e leitura de email.

A solução trabalhada utilizou os conceitos de listas encadeadas e alocação dinâmica, apresentados pelos professores no início da disciplina de Estrutura de Dados. As listas encadeadas foram utilizadas de diferentes maneiras de acordo com a necessidade de armazenamento dos diferentes tipos abstratos de dados: os Emails foram organizados em uma lista de prioridades, com inserção em qualquer lugar, de acordo com a prioridade dos mesmos, e remoção do início da lista; já os usuários foram encadeados com inserção no início da lista, e remoção em qualquer posição. A alocação dinâmica dessas estruturas permitiu que o servidor fosse utilizado com um número arbitrariamente grande de usuários ou emails, bem como que a memória seja usada de maneira eficiente.

7. Bibliografia

Slides e códigos da disciplina Estrutura de Dados. Disponibilizados via Moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

8. Instruções para compilação e execução

- Acesse o diretório /tp do projeto por meio de um terminal
- Digite o comando **make** para compilar o programa e gerar o arquivo executável **run.out** na pasta /bin
- Ao executar o comando **bin/run.out**, o programa é executado com as entradas e saídas padrão (teclado do usuário e terminal, respectivamente)
- Para utilizar um arquivo de entrada (*.txt, por exemplo), a execução deve ser feita com **bin/run.out < input.txt**