

# Trabalho Prático 1

Henrique Soares Assumpção e Silva

henriquesoares@dcc.ufmg.br

Matrícula: 2020006620

Universidade Federal de Minas Gerais

Julho 2021

## 1 Introdução

A empresa Reallocator CO. fez o contrato para o desenvolvimento de um sistema de segurança, detecção de anomalias e implementação de planos de ação para auxiliar nas atividades de transferência de consciência da mega-net. Dado um conjunto de comandos especificados por um usuário, o sistema deve realizar as ações determinadas por estes comandos de acordo com as definições estabelecidas no enunciado. Essas ações configuram desde operações simples, como inserção de dados de um determinado indivíduo em um buffer, até ações mais complexas, como o envio das consciências armazenadas nesses buffers para a mega-net. Além disso, o sistema também deve ser capaz de lidar com potenciais erros internos e ataques maliciosos que podem ser realizados por terceiros.

Essa documentação tem como objetivo especificar a implementação desse sistema de forma detalhada, explicitando os aspectos mais relevantes da implementação, bem como concluindo que o sistema implementado de fato segue as especificações requisitadas.

## 2 Implementação

O programa foi desenvolvido na linguagem C++, compilada pelo compilador G++ da GNU Compiler Collection, no sistema operacional Windows 10. O programa foi inicialmente implementado e compilado em uma máquina com 16GB de RAM em um processador Intel Core I5-9400F.

## 2.1 Estrutura de Dados

A estrutura de dados principal do programa é a estrutura **Buffer**, que especifica um buffer para um servidor específico.

A estrutura **Buffer** é essencialmente uma Lista Encadeada [1] modificada e simplificada para atender as especificações da Realocator CO. Ela foi implementada numa classe e colocada como um template, que permite que o usuário escolha qual será o tipo de dado de interesse a ser armazenado pelo **Buffer**, no caso do programa, os dados são do tipo `std::string`.

A implementação do **Buffer** segue os passos básicos de uma Lista Encadeada vistos em aula [2]. Foram criados duas estruturas auxiliares para a construção do **Buffer**: a estrutura **Item**, implementada numa STRUCT e colocada como um template, que possui um atributo do tipo especificado pelo template — que no caso do programa é `std::string` — e um atributo do tipo `int` que atua como uma chave identificadora<sup>1</sup>, e a estrutura **BufferCell**, também implementada numa STRUCT e colocada como um template, que armazena uma variável do tipo `Item<T>` e um ponteiro para outra variável do tipo **BufferCell**, que irá fazer o encadeamento da lista.

A estrutura **Buffer** possui quatro atributos privados, bem como um conjunto de métodos públicos que serão discutidos na seção 2.2 desta documentação. Os atributos seguem o padrão discutido em aula, ou seja, são dois atributos do tipo ponteiro para **BufferCell** chamados FRONT e BACK, que irão agir como o início e o fim do **Buffer**, respectivamente, além de dois atributos do tipo `int` chamados SIZE e KEY\_COUNTER, um armazenando o tamanho do **Buffer** e outro agindo como um contador<sup>2</sup> para as chaves identificadoras de um **Buffer**, respectivamente. Optou-se por utilizar uma célula cabeça para a auxiliação da implementação da classe **Buffer**, célula essa que é intuitivamente inicializada no método construtor. Além disso, a estrutura **Buffer** também declara uma classe interna **EmptyBufferError**, que será utilizada no manejo de exceções durante a implementação dos métodos da classe.

Em relação à estrutura de dados escolhida para representar um **Buffer** do sistema, vale ressaltar que, de acordo com as instruções enunciadas, ela se comporta de forma bastante semelhante à uma Fila Encadeada [1]. No entanto, como existem alguns casos especiais que quebram com a estrutura padrão da Fila - como por exemplo o comando WARN que deve transferir um item de uma dada posição para o início do Buffer - optou-se por utilizar uma implementação baseada em uma Lista Encadeada, mesmo que muitos dos comandos sigam uma ordem FIFO(First in First Out).

---

<sup>1</sup>Vale observar que, dentro do contexto do trabalho, o uso de chaves identificadoras não é necessário, pois nenhum dos comandos possíveis do programa requer uma busca pela lista. Apesar disso, optou-se por manter um atributo chave em **Item**, para caso se queira implementar métodos e funções que necessitem dessa funcionalidade no futuro.

<sup>2</sup>Esse atributo foi adicionado com o objetivo de realizar um controle interno das chaves identificadores para cada **Item** armazenado no **Buffer**, de modo a tornar o uso do programa mais fácil e menos trabalhoso para o usuário. Mais informações sobre esse atributo serão fornecidas na seção 2.2.

## 2.2 Especificação de classes e métodos

Como as estruturas **BufferCell** e **Item** tratam-se de **STRUCTS**, não é necessária a implementação de métodos **get** e **set** para ter acesso às suas variáveis. Por esse motivo, optou-se por discutir em detalhes apenas a implementação da classe **Buffer**, tendo em vista a simplicidade das outras estruturas utilizadas.

A classe **Buffer** possui os seguintes métodos:

**Buffer()**: método construtor, que inicializa o encadeamento do buffer, criando uma célula cabeça e armazenando seu endereço na variável **front**, bem como apontando a variável **back** para essa célula;

**bool is\_empty()**: método que retorna o valor **TRUE** de tipo **bool** caso o atributo **size** do **Buffer** seja igual à zero, caso contrário retorna **FALSE**;

**int get\_size()**: método que retorna o atributo **size** do **Buffer**;

**void push\_back(Item<T> item)**: método que recebe como entrada um objeto de tipo **Item<T>** e o adiciona ao final do **Buffer**. Inicialmente, o atributo **key** da entrada recebe o valor do atributo **key\_counter** do **Buffer**, e então é criada uma nova célula que armazena a entrada, e os atributos **key\_counter** e **size** são incrementados em uma unidade. Além disso, o atributo **back** do **Buffer** aponta para a nova célula criada. O processo descrito com o atributo **key\_counter** garante a criação de uma chave única para cada um dos itens armazenados no **Buffer**, além de poupar o manejo de chaves<sup>3</sup> pelo usuário;

**void push\_back(T content)**: método sobrecarregado<sup>4</sup> que recebe como entrada uma variável do tipo **T** definido no template, instancia um objeto de tipo **Item<T>** que armazena **content** e finalmente chama o método **void push\_back(Item<T> item)** previamente descrito;

**void push\_front(Item<T> item)**: método que recebe como entrada um objeto de tipo **Item<T>** e o adiciona ao início do **Buffer**. Realiza as mesmas operações com os atributos **key\_counter** e **size** do **Buffer** descritas em **void push\_back(Item<T> item)**. Como está se utilizando uma implementação com célula cabeça, uma nova célula é criada armazenando o objeto de tipo **Item<T>** da entrada, que então é armazenada no atributo **next** da célula cabeça do **Buffer**;

**void push\_front(T content)**: método sobrecarregado<sup>4</sup> que recebe como entrada uma variável do tipo **T** definido no template, instancia um objeto

---

<sup>3</sup>Essa escolha de implementação foi realizada tendo em vista a estrutura do programa desejado. Como não é necessário a definição explícita de chaves pelo usuário em nenhum momento do programa, acredita-se que transferir essa responsabilidade por completo às estruturas da classe torna o programa principal mais intuitivo e de fácil uso.

<sup>4</sup>Essa escolha de implementação foi feita com o objetivo de facilitar a inserção de elementos no Buffer pelo usuário, tendo em vista que desta forma o mesmo não precisa se preocupar em instanciar um objeto de tipo **Item<T>** para adicionar valores do tipo **T** ao **Buffer**.

de tipo `Item<T>` que armazena `content` e finalmente chama o método `void push_front(Item<T> item)` previamente descrito;

`Item<T> pop_front()`: método que, caso o **Buffer** não esteja vazio, remove a primeira célula do **Buffer**, i.e., a célula seguinte à célula cabeça, e retorna o item armazenado nesta. Além disso, o atributo `next` da célula cabeça agora armazena o valor de `next` da célula removida. Caso o **Buffer** esteja vazio, este método joga um objeto do tipo **EmptyBufferError** e encerra;

`Item<T> pop_pos(int pos)`: método que recebe como entrada uma variável do tipo `int` e, caso o **Buffer** não esteja vazio, remove a célula na posição `pos` do **Buffer** e retorna o item armazenado nesta. Caso o **Buffer** esteja vazio, este método joga um objeto do tipo **EmptyBufferError** e encerra;

`void print()`: método que imprime os elementos do **Buffer**, seguindo uma ordem FIFO (First in First Out) estabelecida no enunciado do problema, ou seja, serão imprimidos os conteúdos dos atributos de tipo `Item<T>` de cada célula do **Buffer** na ordem em que estes foram adicionados ao **Buffer**. Esse método é vazio dentro da classe **Buffer** e é especificado fora da classe somente para o tipo `std::string`;

`void clear()`: método que deleta todas as células do **Buffer**, com exceção da cabeça. Esse método realiza chamadas sucessivas do método `Item<T> pop_front()` até que o **Buffer** fique vazio;

`~Buffer()`: método destrutor do **Buffer**, que chama o método `void clear()` e deleta o valor armazenado pela célula cabeça;

## 2.3 Programa principal

Inicialmente o programa principal, implementado no arquivo `main.cpp`, lê da linha de comando um argumento de entrada que representa o nome do arquivo que contém os comandos a serem executados. Após isso, o programa lê do arquivo com os comandos o número de buffers necessários e os armazena em uma variável do tipo `int`, variável essa que **caso receba um valor menor ou igual à zero resultará no encerramento do programa**. Caso contrário, é criado um array de objetos do tipo `Buffer<std::string>`, que são armazenados na variável `servers` de tipo `Buffer<std::string>*`. Além disso, outra variável chamada `hist` de tipo `Buffer<std::string>` é inicializada, para armazenar o histórico de consciências enviadas para a rede.

Após esses procedimentos de inicialização, o programa entra em um loop `while`, que lê cada uma das linhas do arquivo contendo os comandos até encontrar um EOF. Os possíveis comandos são: INFO, WARN, TRAN, ERRO, SEND e FLUSH. Para cada linha do arquivo de comandos, o programa identi-

fica o comando em questão, recebe os parâmetros necessários para sua execução, checa se a operação em questão é válida<sup>5</sup> dentro do contexto dos buffers, e finalmente executa o comando. Os comandos podem executar ações variadas, e como uma descrição detalhada de cada um deles já foi fornecida no enunciado do trabalho, optou-se por omiti-la nesta documentação. Durante a execução dos comandos ERRO e FLUSH, utiliza-se o método `void print()` da classe **Buffer** para fornecer a saída no formato padrão ‘`stdout`’ exigido pelo enunciado. Ao final do loop, o programa fecha o arquivo contendo os comandos, deleta o conteúdo da variável `servers` e encerra.

### 3 Instruções de compilação e execução

Os arquivos foram estruturados de acordo com o exigido pelo enunciado, ou seja, para executar o programa principal basta seguir os seguintes passos:

- Acesse o diretório TP/
- Utilizando um terminal, execute o arquivo **Makefile**, através do comando `make`
- Após isso, execute os seguintes passos de acordo com o sistema operacional utilizado:
  - No terminal Linux: execute o comando `./bin/run.out <TARGET_FILE>`.
  - No prompt de comando do Windows: execute o comando `.\bin\run.out <TARGET_FILE>`.

Onde `<TARGET_FILE>` indica o nome do arquivo com os comandos.

- Por fim, execute o comando `make clean` para remover os arquivos `.o` e `.out` gerados. Por limitações do próprio **Make**, esse comando pode não funcionar corretamente em sistemas Windows. Nesse caso, recomenda-se que se delete os arquivos compilados manualmente.

### 4 Análise de complexidade

Nesta seção, será realizada uma análise de complexidade de tempo e espaço para métodos apresentados na seção 2, bem como uma análise do programa principal.

---

<sup>5</sup>Como o enunciado não especifica qual ação deve ser tomada em caso de uma entrada inválida, como por exemplo a execução do comando `WARN 5 10` quando o buffer 5 tem menos de 11 elementos, ou seja, quando se quer acessar uma posição inexistente de um buffer, ou a execução de `ERRO -1` - pois o buffer de índice `-1` não existe -, optou-se por ignorar comandos inválidos. Para isso, é feita uma checagem nos atributos de entrada de comandos que manipulam algum buffer, para garantir que esse buffer existe e que o acesso a um determinado elemento do buffer é válido. Caso o comando seja determinado como inválido, ele é ignorado e o programa passa para a próxima linha do arquivo contendo os comandos.

## 4.1 Análise de Tempo

### 4.1.1 Métodos da classe Buffer

**Buffer():** o método construtor faz duas operações de ordem  $\mathcal{O}(1)$ , portanto sua ordem de complexidade também é  $\mathcal{O}(1)$ ;

**bool is\_empty():** o método faz uma operação de comparação de ordem  $\mathcal{O}(1)$ , portanto sua ordem de complexidade também é  $\mathcal{O}(1)$ ;

**int get\_size():** o método apenas retorna um atributo da classe, portanto sua ordem de complexidade é  $\mathcal{O}(1)$ ;

**void push\_back(Item<T> item):** o método faz 5 operações de ordem  $\mathcal{O}(1)$ , portanto sua ordem de complexidade também é  $\mathcal{O}(1)$ . Esse resultado também segue do fato que sabe-se que a inserção de elementos no fim de uma lista encadeada tem ordem de complexidade de tempo constante [1];

**void push\_back(T content):** como se trata de um método sobrecarregado que chama o método descrito anteriormente, sua ordem de complexidade também é  $\mathcal{O}(1)$ ;

**void push\_front(Item<T> item):** o método faz 5 operações de ordem  $\mathcal{O}(1)$ , portanto sua ordem de complexidade também é  $\mathcal{O}(1)$ . Esse resultado também segue do fato que sabe-se que a inserção de elementos no início de uma lista encadeada tem ordem de complexidade de tempo constante [1];

**void push\_front(T content):** como se trata de um método sobrecarregado que chama o método descrito anteriormente, sua ordem de complexidade também é  $\mathcal{O}(1)$ ;

**Item<T> pop\_front():** o método faz 4 operações de ordem  $\mathcal{O}(1)$ , portanto sua ordem de complexidade também é  $\mathcal{O}(1)$ . Esse resultado também segue do fato que sabe-se que deletar elementos no início de uma lista encadeada tem ordem de complexidade de tempo constante [1];

**Item<T> pop\_pos(int pos):** o método deve percorrer o buffer até uma determinada posição **pos**, portanto sua complexidade de tempo depende da entrada **int pos**. No melhor caso, **int pos** = 0 e o método executa 9 operações de ordem  $\mathcal{O}(1)$ , logo sua ordem de complexidade no melhor caso também é  $\mathcal{O}(1)$ . No pior caso, **int pos** =  $n$ , sendo  $n$  o tamanho do Buffer, ou seja, o método deve percorrer todo o buffer para realizar sua operação e portanto ele executa 8 operações de ordem  $\mathcal{O}(1)$  e uma loop de ordem  $\mathcal{O}(n)$ , logo sua ordem de complexidade é  $\mathcal{O}(n)$ ;

**void print():** o método deve percorrer o buffer e imprimir cada um de seus

itens, fazendo duas operações de ordem  $\mathcal{O}(1)$  e um loop de ordem  $\mathcal{O}(n)$ , onde  $n$  é o tamanho do buffer, portanto sua ordem de complexidade no pior caso é  $\mathcal{O}(n)$ ;

`void clear()`: o método deve percorrer o buffer e deletar cada uma de suas células, fazendo um loop de ordem  $\mathcal{O}(n)$ , onde  $n$  é o tamanho do buffer, portanto sua ordem de complexidade é  $\mathcal{O}(n)$ ;

`~Buffer()`: o método destrutor chama o método `void clear()`: e executa uma operação de ordem  $\mathcal{O}(1)$ , logo sua ordem de complexidade é  $\mathcal{O}(n)$ ;

#### 4.1.2 Programa principal

Para obter a ordem de complexidade de tempo do programa principal, é necessário analisar tanto a estrutura do loop principal do programa quanto as estruturas internas de determinados comandos.

Primeiramente, define-se  $N > 0$  como o número total de linhas contidas no arquivo com os comandos, com exceção da primeira. Também define-se  $0 < K \leq N$  como o número de linhas contidas no arquivo com os comandos que executam o comando INFO - o único comando que adiciona um novo item a algum buffer, todos os outros consistem de manipulações com dados já existentes ou de remoções de dados. O caso em que  $N = 0$  é trivial e resulta em uma ordem de complexidade de tempo constante. Sendo assim, a loop `while` que itera sobre todas as linhas realiza  $N$  iterações.

Agora, define-se  $n > 0$  como sendo o número de buffers inicializados pelo programa, fornecidos pela primeira linha do arquivo contendo os comandos. O caso onde  $n = 0$  caracteriza o melhor caso do programa, e é trivial ver que ele resulta em uma ordem de complexidade igual a  $\mathcal{O}(1)$ , pois o programa encerra caso  $n \leq 0$ .

O comando INFO realiza operações de ordem constante, já o restante dos comandos realiza operações que dependem do tamanho de um buffer específico. Como o maior tamanho possível para um buffer é  $K$  - pois existem no máximo esse número de itens no programa -, temos que a ordem de complexidade de tempo no pior caso de todos estes comandos é  $\mathcal{O}(K)$ . No pior cenário, um dentre os  $n$  buffers possui o número máximo de elementos e todos os comandos operam sobre esse buffer. Além disso, nesse contexto de pior caso, o comando ERRO só pode ser chamado para o buffer que contém todos os elementos se ele for o último a ser executado, tendo em vista que esse comando apaga os elementos do buffer.

O comando INFO é executado  $K$  vezes, portanto os demais comandos serão executados  $N - K$  vezes. Logo, o loop `while` principal realizará  $N - K$  iterações de ordem  $\mathcal{O}(K)$  e  $K$  iterações de ordem  $\mathcal{O}(1)$ , portanto a ordem de complexidade de tempo final do programa no pior caso será:

$$(N - K) \cdot \mathcal{O}(K) + K \cdot \mathcal{O}(1) = \mathcal{O}(NK - K^2) + \mathcal{O}(K) \quad (1)$$

logo, a ordem de complexidade de tempo é  $\mathcal{O}(\max(NK - K^2, K))$ .

Como  $K$  é constante e depende de  $N$ , percebe-se que a complexidade de tempo de pior caso do programa principal pode ser simplificada e é dada por  $\mathcal{O}(N)$ .

## 4.2 Análise de Espaço

### 4.2.1 Métodos da classe Buffer

Nenhum dos métodos da classe Buffer instancia objetos que dependam de alguma entrada, ou seja, são feitas operações de complexidade de tempo constante para todos os métodos. Portanto, a ordem de complexidade de espaço de cada um deles é  $\mathcal{O}(1)$ .

### 4.2.2 Programa principal

O programa principal instancia um objeto de tipo `Buffer<std::string>*` na variável `servers` e um objeto de tipo `Buffer<std::string>` na variável `hist`. Apenas as operações realizadas nessas variáveis apresentam complexidade de espaço não constante, tendo em vista que o restante das variáveis não são conjuntos de elementos, e.g. `int`, `std::string`, etc, ou seja, para definir a ordem de complexidade de espaço de todo o programa, basta obter a ordem de complexidade da variável `servers`, de todos os buffers armazenados nesta e também da variável `hist`.

Sendo  $n > 0$  o número de Buffers a serem criados, definido na primeira linha do arquivo contendo os comandos, temos que a variável `servers` cria um array com  $n$  objetos do tipo `Buffer<std::string>`, e portanto realizando uma operação com ordem de complexidade de espaço de  $\mathcal{O}(n)$ . Considera-se  $n$  estritamente maior que zero pois o oposto leva ao caso trivial onde não são criados buffers no programa, e portanto sua complexidade de espaço é dada por  $\mathcal{O}(1)$ .

Sendo  $N > 0$  o número de linhas do arquivo que executam o comando INFO - o único comando que adiciona um novo item a algum buffer, todos os outros consistem de manipulações com dados já existentes ou de remoções de dados. Define-se  $N$  como um valor estritamente maior que zero pois, caso  $N = 0$ , a complexidade de espaço do programa dependeria somente de  $n$  e seria trivialmente da ordem de  $\mathcal{O}(n)$ . Nota-se que, durante toda a execução do programa, existem no máximo  $N$  dados armazenados no conjunto de todos os buffers, ou seja, a ordem de complexidade de espaço do conjunto de todos os buffers é  $\mathcal{O}(N)$ . Para a variável `hist`, percebe-se que, por definição, ela pode armazenar no máximo  $N$ , portanto sua complexidade de espaço de pior caso também é  $\mathcal{O}(N)$ .

Assim sendo, obtemos as três ordens de complexidade de interesse e agora podemos calcular a ordem de complexidade de espaço final do programa<sup>6</sup>, que será:

$$\mathcal{O}(n) + \mathcal{O}(N) + \mathcal{O}(N) = \mathcal{O}(\max(n, N)) \quad (2)$$



### 4.3 Disponibilidade do código

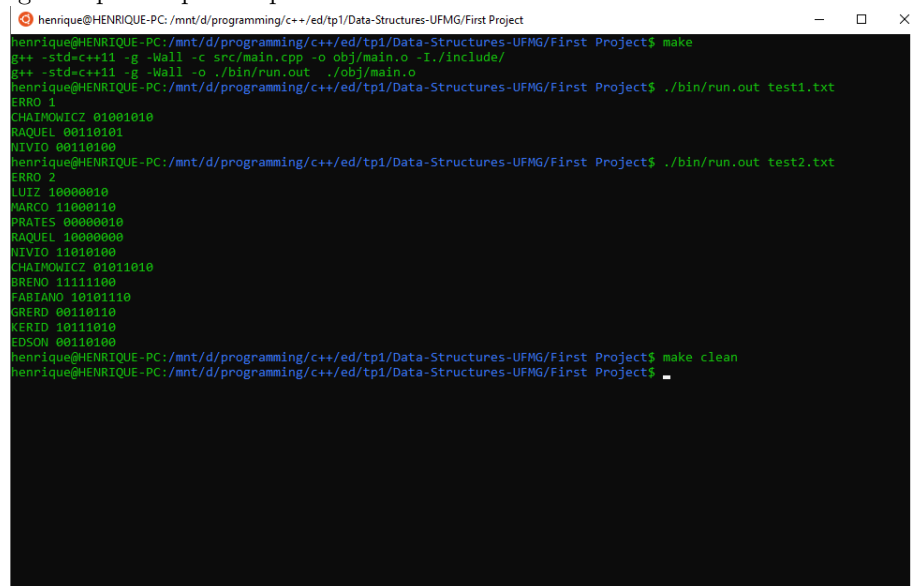
Todo o código deste projeto, bem como esta documentação, serão disponibilizados para o livre acesso e manipulação em um repositório do GitHub<sup>7</sup>, sob a licença THE UNLICENSE.

### 4.4 Testes

Em conjunto ao código supracitado, também foi realizado uma série de testes de correteude sobre a classe **Buffer**, para garantir que todos os métodos estão realizando as suas funcionalidades de forma correta. Com o objetivo de explicitar os testes realizados, um arquivo `tests.cpp` contendo alguns testes estruturais básicos utilizando o framework doctest será disponibilizado no repositório do GitHub deste projeto. A compilação do arquivo de testes não é feita automaticamente pelo arquivo `Makefile`, ou seja, deve ser feita uma compilação manual por meio de um compilador de C++ compatível.

O programa também foi testado em ambos os casos de teste fornecidos no enunciado, e a figura 1 ilustra as saídas obtidas.

Figure 1: Resultado da execução do arquivo `Makefile` no terminal Linux(Ubuntu 20.04 LTS). O arquivo `test1.txt` contém o caso de teste fornecido na seção 3 do enunciado. Já o arquivo `test2.txt` contém o caso de teste fornecido no apêndice A do enunciado. Nota-se que a saída é exatamente igual àquela esperada pelo enunciado.



```
henrique@HENRIQUE-PC: /mnt/d/programming/c++/ed/tp1/Data-Structures-UFMG/First Project
henrique@HENRIQUE-PC: /mnt/d/programming/c++/ed/tp1/Data-Structures-UFMG/First Project$ make
g++ -std=c++11 -g -Wall -c src/main.cpp -o obj/main.o -I./include/
g++ -std=c++11 -g -Wall -o ./bin/run.out ./obj/main.o
henrique@HENRIQUE-PC: /mnt/d/programming/c++/ed/tp1/Data-Structures-UFMG/First Project$ ./bin/run.out test1.txt
ERRO 1
CHAIMOWICZ 01001010
RAQUEL 00110101
NIVIO 00110100
henrique@HENRIQUE-PC: /mnt/d/programming/c++/ed/tp1/Data-Structures-UFMG/First Project$ ./bin/run.out test2.txt
ERRO 2
LUIZ 10000010
MARCO 11000110
PRATES 00000010
RAQUEL 10000000
NIVIO 11010100
CHAIMOWICZ 01011010
BRENO 11111100
FABIANO 10101110
GERERD 00110110
KERID 10111010
EDSON 00110100
henrique@HENRIQUE-PC: /mnt/d/programming/c++/ed/tp1/Data-Structures-UFMG/First Project$ make clean
henrique@HENRIQUE-PC: /mnt/d/programming/c++/ed/tp1/Data-Structures-UFMG/First Project$
```

<sup>6</sup>Caso o comando SEND - o único comando que insere elementos na variável `hist` - não seja chamado, sua ordem de complexidade de espaço será constante, porém o cálculo da ordem final do programa não será alterado, pois  $\mathcal{O}(N) + \mathcal{O}(1) = \mathcal{O}(N)$ .

## 5 Conclusão

Este trabalho abordou o problema de se criar um sistema para auxiliar no gerenciamento de consciências para a empresa Reallocator CO., no qual utilizou-se de uma estrutura **Buffer**, inspirada em uma Lista Encadeada, para realizar possíveis operações requisitadas por um usuário do sistema.

A seção 4.4 ilustra a corretude do sistema criado, que atende as demandas da empresa e executa com êxito todas as atividades previstas. Verifica-se também que a solução apresentada não é só correta mas também eficiente, como explicitado na seção 4.

Por meio deste projeto, foi possível exercitar os conhecimentos sobre estruturas de dados vistos em sala, bem como aplica-los em um contexto empresarial e que se aproxima mais do cotidiano de um futuro programador. Não foram encontrados grandes desafios durante a implementação dos programas em si, no entanto, a criação de uma documentação clara e bem executada foi sem dúvida um desafio que ilustra a importância desse tipo de atividade para o entendimento e a propagação de código em larga escala.

## References

- [1] Ziviani, N., Projeto de Algoritmos com Implementações em Pascal e C, 2<sup>a</sup> Edição, Editora Thomson, 2004.
- [2] Chaimowicz, L. and Prates, R. (2020). Slides virtuais da disciplina de estruturas de dados. Disponibilizado via moodle. Departamento de Ciencia da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

---

<sup>7</sup><https://github.com/HenrySilvaCS/Data-Structures-UFMG/tree/main/First%20Project>