Documentação - Trabalho Prático 1

Gabriel Medeiros Teixeira
Matrícula: 2020054420
Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil
Sistemas de Informação
Estruturas de Dados - 2021/1

gabrielmt09@ufmg.com

1. Introdução

O problema proposto foi implementar um sistema de servidores de "transferência de consciência" para a empresa Rellocator CO. em um universo fictício futurista. Na prática, esses servidores atuam utilizando buffers para armazenar dados. Além disso, o sistema também possui um histórico com propriedades bem semelhantes aos buffers citados anteriormente, tendo a movimentação de dados implementada usando a política First In First Out. Ademais, uma série de comandos devem poder ser usados para manipular o sistema, sendo eles "**info**", que adiciona informações ao buffer de um servidor, "**warn**", que transfere um dado x de uma posição qualquer do buffer escolhido para a primeira, "**tran**", que transfere as informações de um servidor para outro, além de "**erro**", que declara um erro ocorrido em um servidor, imprime todo o seu conteúdo e depois o deleta, "**send**", que envia o primeiro dado de todos os servidores e o insere no histórico, seguindo a política "FIFO", e, por fim "**flush**", que imprime todos os dados do histórico e, logo depois, dos servidores em ordem. O programa em questão tem como entrada um arquivo que descreve quantos servidores serão utilizados, seguido dos comandos, na ordem em que devem ser executados, e dados que devem ser manipulados nos servidores.

2. Implementação

O programa foi desenvolvido na linguagem C++ (C++11) e compilado pelo compilador G++ (Ubuntu 9.3.0-17ubuntu1~20.04) da GNU Compiler Collection. Ele deve ser compilado com o Makefile incluído na pasta raiz do programa, que foi testado em um sistema operacional Windows 10, a partir de um terminal WSL (v1). Assim, para rodar o programa, deve ser executado, no diretório raiz, o comando “make all”, que compilará os programas, para então executar o arquivo “run.out”, na pasta bin, com o arquivo de entrada desejado. Seguem alguns detalhes de hardware do ambiente de teste:

- Processador: Intel® Core™ i5-9300H CPU @ 2.40GHz
- Memória RAM: 16 GB

2.1. Estrutura de Dados

A implementação do programa teve como base o uso de filas encadeadas. Elas foram usadas para desenvolver a principal estrutura utilizada, os buffers dos servidores citados na premissa do problema. A preferência por elas no lugar das filas sequenciais deu-se pela primeira não ser limitada por um tamanho máximo previamente estabelecido (possui tamanho dinâmico), ou seja, o número de buffers/servidores que serão utilizados depende integralmente da entrada do sistema. Para executar todas as tarefas propostas, foram construídas 3 classes, que serão apresentadas a seguir e, mais futuramente, seus métodos.

2.1.1. Celula

Para representar as células que constituem a fila encadeada, como ensinado nas aulas, foi construída a classe *Celula*, declarada como um template, ferramenta que possibilita o polimorfismo de tipos em C++. Ela tem como função armazenar o um dado (em *item*, do tipo T) e possui um atributo (*prox*) que armazena o ponteiro

para outra instância da classe, sendo assim um componente essencial da fila encadeada. Vale lembrar que todos os atributos dessa classe são privados. Essa estrutura também está marcada como friend das classes Buffer e Sistema, que ainda citarei, ou seja, essas classes podem acessar seus atributos privados.

2.1.2. Buffer

A fila encadeada que representa um buffer foi construída como a classe *Buffer*. Vale adiantar que todos os atributos dessa estrutura são marcados como privados. Ela possui um ponteiro para uma “célula-cabeça”, armazenado em *frente*, sendo que o ponteiro para a primeira célula que armazena um dado é guardado em *frente->prox*, assim como visto em aula, a fim de facilitar algumas operações. Também é armazenado o ponteiro para a última célula da fila (*tras*) e o tamanho inteiro dela (*tamanho*), a fim de evitar o custo de iterar pela estrutura toda vez que se quiser saber seu tamanho.

Já tratando-se dos métodos dessa estrutura (tendo em mente que todos são públicos), a maioria foi adaptada daqueles ensinados nas aulas sobre essa matéria, com exceção de *furaFila,* que passa um dos elementos de um buffer para sua primeira posição, e *imprime*, que imprime na tela o valor do dado armazenado em cada célula do buffer em questão. Todos os métodos referentes a essa e outras classes serão mais detalhados futuramente.

2.1.3. Sistema

A classe Sistema representa o ‘sistema de servidores’ cuja implementação é proposta pelo problema, ou seja, ele possui, como atributos privados, um ponteiro de *Buffer* para armazenar um *array* de *Buffer*, *servidores*, sendo que cada posição desse *array* representa um dos servidores propostos no problema em questão, além de *historico*, do tipo *Buffer*, e, por fim, o inteiro *numeroDeServidores*, que armazena o número de servidores que essa instância da classe irá manejar (além de *historico*). Em outras palavras, *numeroDeServidores* delimita o tamanho do *array* guardado por *servidores*.

Além disso, essa estrutura também possui os métodos públicos *info*, *warn*, *tran*, *erro*, *send* e *flush*, além de seu destrutor e construtor.

2.2. Função main e Métodos de Classe

Nessa seção serão detalhados todos os métodos usados em cada classe e a função *main* do programa implementado. Vale ressaltar de antemão que todos os métodos de cada classe são públicos.

2.2.1. Celula

O único método dessa classe é seu construtor padrão, que não recebe parâmetros. Ele chama o construtor padrão do tipo 'T' e atribui 'nullptr' a 'prox'.

2.2.2. Buffer

De início, tem-se o construtor padrão da classe, que atribui a *tamanho* o valor 0, inicializa *frente* com o endereço de memória de instância de *Celula* alocada dinamicamente e atribui a *tras* o valor de *frente*. Tem-se então o destrutor da classe, que chama o método *limpa* e utiliza a função *delete* para deletar *frente*. Já *getTamanho* retorna o valor inteiro do atributo *tamanho*. e*nfileira* adiciona uma célula, contendo a string passada através do parâmetro *item*, no final do buffer e incrementa o valor de *tamanho* em 1. *furaFila* posiciona a célula referente à posição passada como parâmetro (tendo 0 como índice da primeira célula que armazena um dado) na primeira posição do buffer (indicada por *frente->prox*). *desenfileira* remove a primeira célula do buffer (*frente->prox*), retorna a string que ela armazena e reduz em 1 o valor de *tamanho*. *imprime* imprime a string armazenada por todas as células do buffer (sendo *frente->prox* a primeira), seguindo a política FIFO. Por fim, *limpa* utiliza a função *delete* para deletar todas as células do buffer (a partir de *frente->prox*), então define *tamanho* como 0 e atribui *frente* a *tras*.

O único tratamento de erro feito nessa classe é a verificação de que os índices são válidos para aqueles métodos que os recebem como parâmetros.

2.2.3. Sistema

Inicialmente, tem-se o construtor da classe, que inicializa *numeroDeServidores* como o valor inteiro do parâmetro *numeroDeServidores*,

instancia um array de *Buffer* (com o tamanho equivalente a *numeroDeServidores*) dinamicamente e atribui seu endereço de memória ao ponteiro de *Buffer servidores*. Já o destrutor utiliza da função *delete* para deletar o array de *Buffer* apontado por *servidores*.

info adiciona uma célula no final do Buffer armazenado na posição referente ao parâmetro inteiro *indiceServidor* do array de Buffer apontado por *servidores*. Essa nova célula armazena o valor da variável de string *dados*, passada como parâmetro para o método.

warn recebe como parâmetros os inteiros *indiceServidor* e *posicaoItem*, acessa o buffer armazenado na posição *indiceServidor* do array de Buffer apontado por *servidores* e passa para a frente do buffer (*frente->prox*) a célula na posição referente ao valor *posicaoItem*.

tran recebe como parâmetros os inteiros *indiceServidor1* e *indiceServidor2* e acessa os buffers nas posições *indiceServidor1* e *indiceServidor2,* inserindo uma célula nova ao final do segundo para cada uma do primeiro. Essas novas células do buffer referente a *indiceServidor2* armazenam o mesmo valor das correspondentes do buffer indicado por *indiceServidor1*. Tanto a inserção de novas células em um quanto a remoção no outro são feitas seguindo a política FIFO.

erro recebe como parâmetro o inteiro *indiceServidor*, acessa o buffer de posição *indiceServidor* do array de Buffer apontado por *servidores*, remove cada célula que ele contém, seguindo a política FIFO, e imprime seu valor.

send remove a célula da primeira posição (*frente->prox*) de cada buffer armazenado no array de Buffer apontado por *servidores* e adiciona novas no Buffer *historico*, uma para cada removida (os valores armazenados nas novas células de *historico* são os mesmos que eram armazenados por aquelas removidas dos buffers de *servidores*).

Por fim, *flush* imprime o valor armazenado por cada célula de *historico*, para depois imprimir também os valores armazenados nas células de cada buffer armazenado no array de Buffer apontado por *servidores*, um buffer por vez. As impressões das células seguem a política FIFO. O método *limpa* é chamado após a impressão do conteúdo de cada buffer (contando com *historico*), a fim de esvaziá-lo.

O único tratamento de erro feito nessa classe é a verificação de que os índices são válidos para aqueles métodos que os recebem como parâmetros.

2.2.4. Função *main*

É a principal função do programa, de forma que recebe como um dos parâmetros o arquivo de entrada. Ela utiliza a função *getline* da biblioteca *string* para ler cada linha do arquivo. Desse modo, o valor da primeira linha é convertido para inteiro e usado para inicializar uma instância da classe *Sistema*, *sistema*.

A partir desse ponto, a função itera por cada linha da entrada, a fim de realizar os comandos que elas indicam. Vale ressaltar que, no começo de cada iteração, é inicializada uma variável (*tamanho*) que tem como objetivo armazenar o valor inteiro do tamanho da linha (*texto*). É realizado um tratamento referente a *tamanho* para o caso de possuir o caractere *\r* no final (esse caractere é inserido no Windows mas não em outros sistemas operacionais e, portanto, se essa potencial ocorrência não for levada em consideração para calcular *tamanho*, essa variável terá valores diferentes dependendo do sistema operacional em que a entrada de teste foi criada).

Em seguida, a função *substr*, também da biblioteca *string*, é usada para identificar o comando indicado naquela linha específica. Tendo esse valor, é checado a partir de uma série de *if*s, um para identificar cada comando que o sistema deve ser capaz de executar de acordo com a entrada, como deve-se prosseguir. Logo após, outras informações de *texto* referentes ao comando da entrada são armazenadas em variáveis por meio da utilização de *substr* para, finalmente, serem passadas como parâmetro para o método de *sistema* equivalente ao comando indicado. Vale ressaltar que os erros na função main são exibidos por um catch, caso ocorram.

3. Análise de Complexidade

Nesta seção serão feitas as análises de complexidade, referentes a tempo e espaço, de todos os 17 métodos de classe do programa e da função main.

3.1. Buffer

3.1.1. Construtor

Tempo:

Esse método somente atribui valores aos seus atributos. Vale ressaltar que é chamado o construtor padrão da classe *Celula* para instanciá-la dinamicamente e atribuir essa instância à *frente*. Como todas essas operações possuem custo assintótico de tempo O(1), essa também é a complexidade de tempo do método.

Espaço:

Essa classe não inicializa variáveis internamente. Além disso, como o construtor de Celula chamado possui complexidade de espaço O(1), é possível concluir que a complexidade de espaço geral do método também é O(1).

3.1.2. Destrutor

Tempo:

Esse método chama inicialmente o *limpa()*, que possui complexidade de tempo O(n), para em seguida utilizar o operador *delete* para deletar uma instância de *Celula*, que possui complexidade O(1). Sendo assim, a complexidade de tempo geral do método é O(n).

Espaço:

Tendo em vista que o método não possui variáveis internas e tanto a complexidade de espaço de *limpa()* quanto a do operador *delete* (quando utilizado para deletar uma única instância de *Celula*) são O(1), conclui-se que a complexidade de espaço geral do método é O(1).

3.1.3. getTamanho

Tempo:

O método apenas retorna o valor do atributo *tamanho*, possuindo assim complexidade de tempo O(1).

Espaço:

O método não precisa de espaço para declarar variáveis internas, possuindo assim complexidade de espaço O(1).

3.1.4. limpa

Tempo:

Esse método realiza um loop e três atribuições fora dele. Essas atribuições possuem complexidade O(1), já o loop realiza três operações de complexidade de tempo O(1) (duas atribuições e uma chamada do operador *delete* para uma instância de *Celula*) em cada iteração. Essas operações são realizadas *n* vezes, sendo *n* o tamanho do buffer a partir do qual o método é chamado. Partindo desse pressuposto, o melhor caso acontece quando o tamanho do buffer é 0 no momento em que *limpa* é chamado, pois não será executada nenhuma iteração, já o pior caso ocorre para quando o tamanho do buffer é maior que 1 (O(n)). Levando em consideração todas as informações citadas, conclui-se que a complexidade de tempo do pior caso do método é O(*n*) e do melhor é O(1).

Espaço:

O método precisa de espaço para declaração da variável *celula* fora do loop, mas não requer espaço adicional para as operações realizadas no interior do *while*. sendo assim, a complexidade de espaço do método é constante, O(1).

3.1.5. enfileira

Tempo:

Esse método somente realiza atribuições e chama o construtor padrão do tipo *Celula*. Tendo em vista que a complexidade de tempo de todas essas operações é O(1), esta também é a complexidade de tempo do método.

Espaço:

Todas as operações, como atribuições e inicializações de variáveis possuem custo constante, portanto, a complexidade de espaço do método é O(1).

3.1.6. furaFila

Tempo:

O método possui um *if* que verifica se o parâmetro *posicao* é válido. Caso essa condicional seja cumprida, é utilizado o operador *throw* e o método é encerrado, possuindo complexidade de tempo O(1), nesse caso. Se a condicional não for cumprida, são realizadas operações de atribuição (complexidade O(1)) e é iniciado um loop, que realiza uma atribuição *n* vezes (sendo *n* equivalente ao inteiro *posicao*) e faz uma verificação uma vez. Se essa verificação tiver sua condição cumprida, é realizada uma atribuição (O(1)). Após a conclusão do loop também é realizado um *if* para verificar se a célula colocada no começo da fila era a que estava na última posição. Caso essa condicional seja cumprida, uma atribuição é feita (*this->tras = anterior*). Assim, o melhor caso do método ocorre quando *posicao* tem o valor 0, já que não serão feitas iterações no loop e a complexidade de tempo do método será O(1). O pior caso ocorre quando *posicao* possui um valor maior que 1, quando o método terá complexidade de tempo O(n).

Espaço:

O método precisa de espaço constante para declarar as variáveis *celula* e *anterior*, bem como para o resto das operações realizadas fora do loop (O(1)). Como o *for* não declara nenhuma variável internamente, ele não precisa de espaço extra, de forma que requer espaço constante (O(1)). Assim, a complexidade de espaço do método é O(1).

3.1.7. desenfileira

Tempo:

O método possui um *if* e, caso sua condicional seja cumprida, o operador *throw* é utilizado antes de qualquer outra operação, encerrando o método com complexidade de tempo O(1). Caso essa condicional não seja cumprida, são realizadas somente atribuições, além do uso dos operadores *delete* e *return*. Como todas essas operações possuem complexidade *O(1)*, essa também é a complexidade de tempo geral do método.

Espaço:

O método realiza apenas operações de custo de espaço constante (como a declaração de variáveis fora de um loop, verificação de condicional, uso dos operadores delete e return), possuindo assim complexidade de espaço O(1).

3.1.8. imprime

Tempo:

O método possui um *if* e, caso sua condicional seja cumprida (*tamanho = 0*), o operador *return* é utilizado antes de qualquer outra operação, encerrando o método com complexidade de tempo O(1). Caso essa condicional não seja cumprida, é realizada a declaração de uma variável fora de um loop e um *for* é iniciado. Esse *for* itera *n* vezes (sendo *n* o tamanho do buffer naquele momento), em que cada iteração realiza uma atribuição e imprime um valor na tela. Partindo desse

pressuposto, o melhor caso do método ocorre quando o tamanho do buffer, representado pelo atributo inteiro *tamanho*, é igual a 0, pois o *if* no início do método será chamado, com *imprime* possuindo, assim, complexidade de tempo O(1). Já o pior caso ocorre quando *tamanho* é maior que 1, pois as operações no interior do loop serão realizadas *n* vezes. Conclui-se assim que, no pior caso, o método possui complexidade de tempo O(n).

Espaço:

O método precisa de espaço constante para declarar a variável *celula* (O(1)). Como o *for* não declara nenhuma variável internamente, ele não precisa de espaço extra, de forma que requer espaço constante (O(1)). Assim, a complexidade de espaço do método é O(1).

3.2. Sistema

3.2.1. Construtor

Tempo:

O método realiza uma operação de atribuição (O(1)) e inicializa *n* instâncias de *Buffer* dinamicamente (vale ressaltar que a complexidade de tempo do construtor de *Buffer* para qualquer situação usada no programa é O(1)), sendo *n* o parâmetro inteiro *numeroDeServidores*. Sendo assim, o melhor caso ocorre quando *numeroDeServidores* é 1 e a última operação citada ocorre uma única vez, de modo que a complexidade de tempo é constante (O(1)). Já o pior caso ocorre quando *numeroDeServidores* é maior que 1. Nessa situação, a complexidade de tempo do método será O(n).

Espaço:

O método precisa de espaço constante para realizar uma atribuição ao atributo *numeroDeServidores* (O(1)). Entretanto, ele também aloca espaço para *n* instâncias da classe Buffer, sendo *n* o parâmetro inteiro *numeroDeServidores*. Dessa forma, o melhor caso do método ocorre quando *numeroDeServidores* é 1 e

somente uma instância de *Buffer* é alocada no heap, de modo que a complexidade de espaço do método é constante, O(1). Já o pior caso ocorre quando *numeroDeServidores* é maior que 1. Nessa situação, a complexidade de espaço do método será O(n)

3.2.2. Destrutor

Tempo:

O método utiliza o operador *delete[]* para deletar o array de buffers *servidores*, de modo que precisa chamar o destrutor do tipo *Buffer* (O(1)) para cada instância armazenada no array. Tendo em conta que essa operação será chamada *m* vezes, sendo *m* equivalente ao atributo variável *numeroDeServidores*, o melhor caso do método ocorre quando *numeroDeServidores* é igual a 1 e o tamanho desse único buffer é 0, já que nesse caso a complexidade de tempo do método é O(1). O pior caso ocorre quando *numeroDeServidores* é maior que 1, quando o destrutor de Buffer será chamado *n* vezes, com a complexidade de tempo do método sendo O(nm), em é o tamanho do maior buffer no array *servidores*.

Espaço:

O método realiza apenas uma operação, o uso do operador *delete[]* para deletar *servidores*, e , portanto, tem complexidade de espaço O(1).

3.2.3. info

Tempo:

O método apenas acessa uma posição do array *servidores* e chama o método *enfileira* do tipo *Buffer* (que tem complexidade de tempo O(1)). Conclui-se, portanto, que a complexidade de tempo de *info* é O(1).

Espaço:

O método requer espaço constante para realizar suas operações, já que não declara nenhuma variável e *enfileira* também possui complexidade de espaço O(1). Conclui-se, portanto, que a complexidade de espaço do método é O(1).

3.2.4. warn

Tempo:

O método apenas acessa uma posição do array *servidores* e chama o método *furaFila* do tipo *Buffer* (que em tem complexidade de tempo O(1) quando o parâmetro inteiro *posicaoItem* é igual a 1 ou 0 e O(n) quando *posicaoItem* é maior que 1). Conclui-se, portanto, que a complexidade de tempo de *warn*, em seu melhor caso, é O(1), em contrapartida ao pior caso, em que é O(n).

Espaço:

O método requer espaço constante para realizar suas operações, já que não declara nenhuma variável e *furaFila* também possui complexidade de espaço O(1). Conclui-se, portanto, que a complexidade de espaço do método é O(1).

3.2.5. tran

Tempo:

O método executa um loop que realiza *n* iterações, sendo *n* equivalente ao atributo *tamanho* do buffer referente à posição indicada pelo parâmetro inteiro *indiceServidor1* no array *servidores* (pertencente à classe *Sistema*). Esse loop *while* realiza 4 operações com complexidade de tempo O(1) (acessa duas posições de *servidores* e chama os métodos *enfileira* e *desenfileira*, da classe *Buffer*). Assim, o melhor caso do método ocorre quando o tamanho do buffer indicado por *indiceServidor1* é igual a 0, em que o método terá complexidade de tempo O(1). Já o pior caso ocorre quando o tamanho desse mesmo buffer é maior que 1, pois nessa situação o método terá um custo linear O(n).

Espaço:

O método requer espaço constante para realizar suas operações, já que não declara nenhuma variável e tanto o método *enfileira* quanto o *desenfileira* possuem complexidade de espaço O(1). Conclui-se, portanto, que a complexidade de espaço do método é O(1).

3.2.6. erro

Tempo:

O método executa um *while* que realiza *n* iterações, sendo *n* equivalente ao atributo *tamanho* do buffer referente à posição indicada pelo parâmetro inteiro *indiceServidor* no array *servidores*. Cada iteração realiza somente operações com complexidade de tempo O(1): imprimir na tela uma string, acessar uma posição de *servidores* e executar o método *desenfileira*. Assim, conclui-se que, o melhor caso do método ocorre quando o tamanho do buffer indicado por *indiceServidor* é igual a 0, em que o método terá complexidade de tempo O(1). Já o pior caso ocorre quando o tamanho desse mesmo buffer é maior que 1, pois nessa situação o método terá um custo linear O(n).

Espaço:

O método requer espaço constante para realizar suas operações, já que não declara nenhuma variável e *desenfileira* também possui complexidade de espaço O(1). Conclui-se, portanto, que a complexidade de espaço do método é O(1).

3.2.7. send

Tempo:

O método executa um *while* que realiza *n* iterações, sendo *n* equivalente ao atributo *numeroDeServidores* da instância de *Sistema* pela na qual o método é

chamado. Cada iteração realiza somente operações com complexidade de tempo O(1): verificar uma condicional, executar o método *getTamanho* para uma instância de *Buffer*, acessar uma posição de *servidores* e executar o método *desenfileira*. Assim, conclui-se que, o melhor caso do método ocorre quando *numeroDeServidores* é igual a 1, em que o método terá complexidade de tempo O(1). Já o pior caso ocorre quando o tamanho desse atributo é maior que 1, pois nessa situação o método terá um custo linear O(n).

Espaço:

O método requer espaço constante para realizar suas operações, já que não declara nenhuma variável e nenhuma das funções que executa possui complexidade de espaço maior que O(1). Conclui-se, portanto, que a complexidade de espaço do método é O(1).

3.2.8. flush

Tempo:

O método realiza um loop *for* que executa as operações em seu interior *m* vezes, sendo *m* equivalente ao atributo *numeroDeServidores* da instância de *Sistema* na qual o método é chamado. As operações realizadas dentro do escopo do *for* para cada repetição são: uma verificação de condicional (O(1)), uma chamada do método *getTamanho* (O(1)) e, caso a condicional seja cumprida, uma chamada do método *imprime* e uma de *limpa*, para a mesma instância de *Buffer*. Além disso, fora do *for*, o método executa *imprime* e *limpa* mais uma vez, chamados a partir do atributo *historico* da instância de *Sistema* em questão. Conclui-se, portanto, que a complexidade de tempo de *flush* é, em seu pior caso, O(nm), em que *n* é o tamanho da maior instância de *buffer* do sistema (incluindo *historico)*. Já o melhor caso ocorre quando o tamanho de *historico* é 0 e *numeroDeServidores* possui valor 1, situação em que a complexidade de tempo do método será O(1).

Espaço:

O método requer espaço constante para realizar suas operações, já que não declara nenhuma variável e nenhuma das funções que executa possui complexidade de espaço maior que O(1). Conclui-se, portanto, que a complexidade de espaço do método é O(1).

3.3. Celula

3.3.1. Construtor

Tempo:

Esse método somente realiza uma atribuição (O(1)) e chama o construtor padrão do atributo de tipo *T item*, que, tendo em vista que é usado somente para armazenar strings, tem custo constante O(1). Assim, a complexidade de tempo do método é O(1).

Espaço:

Tendo em vista que o método não possui variáveis internas e o construtor padrão do tipo *string* (único tipo de valor armazenado em *item* no programa) também não, a complexidade de espaço do método é O(1).

3.4. Função main

Tempo:

Essa função realiza algumas atribuições e operações com complexidade O(1), executa uma vez a função *getLine* da biblioteca *string*, que tem complexidade de tempo O(t) (sendo *t* o tamanho da maior linha lida do arquivo de entrada) e então inicia um loop *while* que faz *p - 1* iterações, sendo *p* o número de linhas do arquivo de entrada. Para cada uma dessas iterações, a função *substr* (com complexidade de tempo O(t)) é executada e são realizadas 6 verificações de condicionais (O(1)).

Para facilitar a análise, veremos a complexidade de tempo do pior caso do método *flush* como O(nm), sendo *n* o tamanho da maior instância de *Buffer* do sistema e *m* equivalente ao atributo *numeroDeServidores* de *sistema*. Sendo assim, conclui-se que o pior caso da função *main* tem complexidade de tempo O(p * max(t, nm)), já que se *t* for maior que *nm*, essa será a complexidade de tempo da condicional mais custosa e caso *nm* seja maior que *t*, a condicional mais custosa terá tal complexidade de tempo. Em contrapartida, o melhor caso da função *main* ocorre quando o arquivo de entrada (presumindo que tenha pelo menos a primeira linha descrevendo o número de servidores como, no mínimo, 1, além de uma segunda linha com um comando) estabelece o número de servidores com 1 e execute o comando menos custoso, já que nessa situação a complexidade de tempo da função será O(t).

Espaço:

A função requer espaço constante para declarar algumas variáveis e atualizá-las dentro do loop *while*. Além disso, requer espaço para armazenar *m* buffers em *sistema*, sendo *m* delimitado pelo inteiro na primeira linha do arquivo de entrada, além das células armazenadas em cada buffer. Conclui-se assim que o pior caso da função *main* tem complexidade de espaço O(nm), sendo *n* o tamanho do buffer com mais células (maior que 1). Já o melhor caso tem complexidade de espaço O(1), quando o número de buffers de *sistema é 1 e o tamanho desse único buffer é 0;*

4. Conclusão

Ao desenvolver o trabalho prático, foi possível perceber que ele é dividido em duas grandes partes, sendo a primeira aquela referente a implementar as estruturas de dados principais, como aquela referente à fila encadeada, além de suas células. Já a segunda trata-se de implementar efetivamente os comandos que compõem o sistema de gerenciamento de servidores da “Rellocator CO”.

A parte mais complicada foi a construção correta da segunda parte, levando em consideração as exceções possíveis e como tratá-las. Isso se estende desde a implementação dos métodos da classe *Sistema* que representam os comandos que

devem ser realizados pelo sistema até o processamento de entrada no arquivo principal do programa, *Main*. Tal procedimento levou várias etapas de refatoração e correção de bugs até a conclusão.

Conclui-se, portanto, que por meio desse trabalho prático foi possível ter noção prática da importância do planejamento do desenvolvimento, atenção às exceções e o impacto que mudanças aparentemente pequenas podem ter no comportamento de um algoritmo. Por fim, pode-se perceber a distância entre a implementação e a efetiva utilização de um programa, já que a primeira pode ser completamente alterada sem modificar a segunda.

5. Referências

Ziviani, N. (2006). Projetos de Algoritmos com Implementações em Java e C++

Chaimowicz, L. and Prates, R. (2020). Slides virtuais da disciplina de estruturas de dados. Disponibilizado via moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

www.cplusplus.com/reference/string/string