Trabalho Prático **3**

Documentação

Estrutura de Dados 2022/01

Bernardo Reis de Almeida (2021032234)

Departamento de Ciência da Computação - Universidade Federal de Minas Gerais
Belo Horizonte - MG - Brasil
bera01@ufmg.br

1. Introdução

O presente trabalho prático tem por objetivo o desenvolvimento de habilidades relacionadas ao projeto de software, à conceituação de algoritmos e à análise computacional sob o pretexto da elaboração de um **simulador para um servidor de emails**. Em particular, um sistema de armazenamento de mensagens deve ser criado e implementado com a capacidade de executar movimentações e consultas em seus dados. Uma entrada contendo declarações de comandos - sejam eles definidos como **inserção**, **consulta** e **remoção** - será fornecida e, a partir dela, o simulador deverá proceder com cada operação de maneira conforme, simulando o funcionamento de um servidor de emails. Uma coleção de informações - as **mensagens** - deve ser armazenada em memória e cada comando deve originar uma saída correspondente. Toda a movimentação de dados entre o programa e o ambiente de execução é realizada por arquivos de entrada e de saída.

Em um primeiro momento, as decisões de projeto e detalhes de implementação serão elucidados na seção **[Método]**. Em seguida, a complexidade computacional dos algoritmos elaborados será discorrida na seção **[Análise de Complexidade]**. Uma breve discussão sobre as estratégias de robustez e de garantia de integridade adotadas será exposta na seção **[Estratégias de Robustez]**. Na sequência, os resultados de testes envolvendo o desempenho computacional e a localidade de referência da aplicação para diferentes casos de uso serão fornecidos e debatidos na seção **[Análise Experimental]**. Por fim, as conclusões sobre o processo de elaboração do presente trabalho, as fontes de pesquisa utilizadas e instruções para a compilação e para a execução do programa podem ser encontradas, respectivamente, nas seções **[Conclusões]**, **[Bibliografia]** e **[Instruções para Compilação e Execução]**.

2. Método

a. Ambiente Computacional

O programa foi desenvolvido em **linguagem C++** (versão 11), em um ambiente computacional **Linux** virtualmente simulado no sistema operacional Windows (Debian GNU/Linux 11 (bullseye) on Windows 10 x86_64) (5.10.16.3-microsoft-standard-WSL2). O compilador utilizado foi o **G++** (versão (Debian 10.2.1-6) 10.2.1 20210110). Os testes foram executados em uma máquina com um processador de 3.60 GHz (6 núcleos, 12 threads) e com 16 gigabytes de memória RAM.

b. Implementação

O problema enunciado foi abordado em duas etapas: a **elaboração de uma estrutura de dados** que simule adequadamente o servidor de emails, sendo capaz de executar cada operação exigida, e a **manipulação de informações entre o programa e arquivos externos**, os quais estabelecem a comunicação entre um suposto usuário e a aplicação.

A primeira etapa exigiu a definição de três elementos: o **servidor de emails**, o qual irá comportar **caixas de entrada**, as quais irão armazenar **mensagens**. O servidor foi modelado por meio de uma **tabela hash**. Cada uma de suas entradas representa uma caixa de mensagens e foi modelada mediante uma **árvore binária de pesquisa**. Seus **nós** são modelos das mensagens em si. O tamanho da tabela, isto é, sua quantidade de entradas, é fixo e definido pelo usuário durante a execução do programa, porém cada entrada pode conter uma quantidade arbitrária de mensagens, limitada apenas pela memória da máquina em que a aplicação está sendo executada. Em termos funcionais, essa hierarquia de estruturas de dados trabalha sobre dois tipos de **identificadores**: um para o **usuário** e um para a **mensagem**, ambos fornecidos juntamente com qualquer comando operacional - inserção, remoção e consulta. O identificador do usuário é mapeado a uma entrada da tabela por meio de uma **função hash**. O identificador da mensagem é utilizado como chave para a sua referenciação na árvore binária de pesquisa correspondente. Caso dois identificadores de usuário sejam mapeados à mesma entrada, as mensagens de ambos devem ser nela armazenadas, porém cada mensagem é unicamente caracterizada pelo identificador de seu usuário e por seu próprio identificador. Todos os elementos mencionados foram implementados como **tipos abstratos de dados**, com o uso de **classes** da linguagem C++. A seguir, tem-se os detalhes técnicos da implementação.

Classe Mail: Representa uma mensagem de email e é um nó para uma árvore binária de pesquisa. Contém seis atributos, três deles referentes ao papel de mensagem: a mensagem em si - variável do tipo string - e os identificadores desta e do usuário a que se destina - variáveis inteiras -, e três deles referentes ao papel de nó: um ponteiro para o nó pai, um para o nó filho esquerdo e um para o nó filho direito - ponteiros para o tipo Mail.

- **Mail():** Construtor da classe. Realiza a instanciação de um objeto. Recebe valores para os três atributos referentes à mensagem - como definido anteriormente - e os inicializa correspondentemente. Os ponteiros são inicializados com valores nulos.

Todos os atributos e métodos são privados. A manipulação das mensagens deve ser feita por meio da caixa de entrada, já que aquelas são apenas uma abstração interna de um dos elementos que compõem esta. A definição da classe é feita no arquivo **mailbox.hpp**.

Classe MailBox: Representa uma caixa de mensagens e é uma árvore binária de pesquisa. Contém apenas um atributo, sendo ele um ponteiro para sua raiz - ponteiro para o tipo Mail. Segue um modelo dinâmico de alocação, isto é, cada nó é progressivamente alocado e inserido na árvore, não existindo um número máximo.

- **MailBox():** Construtor da classe. Realiza a instanciação de um objeto. Apenas inicializa o único atributo, a raiz da árvore, como um ponteiro nulo.

- **~MailBox():** Destrutor da classe. Realiza a desalocação das células alocadas antes da destruição do objeto. Esse processo é realizado mediante uma chamada ao método auxiliar **clear()**, o qual recebe uma referência para um nó da árvore, verifica se o mesmo está alocado e, caso o esteja, desaloca suas subárvores, desalocando-o em seguida. O processo é repetido recursivamente sobre toda a árvore. A chamada passa a raiz como parâmetro, de modo que todos os nós sejam percorridos.

- **insert():** Realiza a inserção de uma nova mensagem (nó) na caixa de entrada (árvore). Recebe três parâmetros: o identificador do usuário a que se destina a mensagem, o identificador da mensagem -

variáveis inteiras - e a mensagem em si - variável do tipo string. A partir daí, percorre a árvore até encontrar o local correto de inserção e aloca um novo nó com as referidas informações.

- **remove():** Realiza a remoção de uma dada mensagem (nó) da caixa de entrada (árvore). Recebe dois parâmetros: o identificador do usuário ao qual a mensagem pertence e o identificador da mensagem em si - variáveis inteiras. A partir daí, busca pela mensagem usando como referência os identificadores recebidos, removendo-a e retornando um valor booleano "verdadeiro" caso a encontre. Caso contrário, retorna um valor booleano "falso". A remoção do nó em si é feita pelo método auxiliar **removeAuxiliary()**, o qual recebe uma referência para um nó - ponteiro para o tipo Mail - e realiza a manipulação dos ponteiros para sua remoção da árvore, desalocando-o em seguida.

- **search():** Realiza a pesquisa por uma dada mensagem (nó) na caixa de entrada (árvore). Recebe dois parâmetros: o identificador do usuário ao qual a mensagem pertence e o identificador da mensagem em si - variáveis inteiras. A partir daí, busca pela mensagem usando como referência os identificadores recebidos e retorna o seu conteúdo (a mensagem em si) - valor do tipo string - caso a encontre. Caso contrário, retorna um texto indicando a falha na operação - valor do tipo string.

Todos os atributos e métodos são privados. A manipulação das caixas de mensagens deve ser feita por meio do servidor de mensagens, já que aquelas são apenas uma abstração interna de um dos elementos que compõem este. A definição da classe é feita no arquivo **mailbox.hpp**.

Classe MailServer: Representa o servidor de emails e é uma tabela hash. Contém dois atributos, sendo um referente ao número de entradas da tabela - variável inteira - e o outro à tabela em si, sendo esta uma coleção de caixas de entrada (vetor de objetos do tipo MailBox) - ponteiro para o tipo MailBox. A alocação da tabela é feita de maneira dinâmica, porém fixa, isto é, o tamanho da tabela é fornecido para a alocação do vetor de objetos do tipo MailBox em tempo de execução, mas permanece constante até o término do programa.

- **MailServer():** Construtor da classe. Realiza a instanciação de um objeto. Recebe o tamanho da tabela como parâmetro - variável inteira -, atribui seu valor ao atributo correspondente e realiza a alocação do vetor de objetos do tipo MailBox com a referida quantidade de entradas.

- **~MailServer():** Destrutor da classe. Realiza a desalocação do atributo referente à tabela, isto é, do vetor de objetos do tipo MailBox, o qual foi alocado dinamicamente.

- **hash():** Função hash da classe. Realiza o mapeamento de um identificador de usuário a uma entrada da tabela. Recebe como parâmetro o identificador de um usuário - variável inteira - e retorna o índice correspondente do vetor de objetos do tipo MailBox - variável inteira. A função em si é uma operação de módulo em valor absoluto entre o identificador e o tamanho da tabela.

- **insert():** Realiza a inserção de uma nova mensagem (nó) na caixa de mensagens (árvore binária) referente a um dado usuário (entrada da tabela hash). Recebe como parâmetros o identificador do usuário, o identificador da mensagem - variáveis inteiras - e a mensagem em si - variável do tipo string. A partir daí, calcula o índice da tabela correspondente ao usuário fornecido mediante a função

hash() e invoca o método insert() sobre essa entrada (árvore binária), repassando os parâmetros recebidos. Retorna um texto indicando a inserção da mensagem no servidor - valor do tipo string.

- **remove():** Realiza a remoção de uma mensagem (nó) da caixa de mensagens (árvore binária) referente a um dado usuário (entrada da tabela hash). Recebe como parâmetros o identificador do usuário e o identificador da mensagem - variáveis inteiras. A partir daí, calcula o índice da tabela correspondente ao usuário fornecido mediante a função hash() e invoca o método remove() sobre essa entrada (árvore binária), repassando os parâmetros recebidos. Retorna um texto indicando o sucesso ou a falha na remoção - valor do tipo string.

- **search():** Realiza a pesquisa por uma mensagem (nó) na caixa de mensagens (árvore binária) referente a um dado usuário (entrada da tabela hash). Recebe como parâmetros o identificador do usuário e o identificador da mensagem - variáveis inteiras. A partir daí, calcula o índice da tabela correspondente ao usuário fornecido mediante a função hash() e invoca o método search() sobre essa entrada (árvore binária), repassando os parâmetros recebidos. Retorna um texto indicando o sucesso ou a falha na pesquisa - valor do tipo string.

Os atributos e o método hash() são classificados como privados, tendo em vista os critérios de encapsulamento da classe e a manutenção da integridade de seus objetos. Os demais são públicos e disponíveis para a manipulação e para a operacionalização do servidor de emails. A definição da classe é feita no arquivo **mailserver.hpp**.

A segunda etapa, por sua vez, é centrada na elaboração da interface de utilização da aplicação. Em particular, a abordagem escolhida envolve o uso de arquivos de texto. Um arquivo de entrada - contendo todas as operações a serem executadas - e um de saída devem ser fornecidos via linha de comando para o programa (detalhes na seção **[Instruções para Compilação e Execução]**). A partir daí, o primeiro será aberto e lido sequencialmente. Cada comando nele presente será interpretado e operacionalizado - caso aplicável - sobre o servidor de emails, gerando uma saída que será exportada ao segundo. Todos esses passos foram modularizados em procedimentos dedicados, os quais são detalhados a seguir.

- **initializeFiles():** Realiza a inicialização dos arquivos de entrada e de saída com base nos caminhos fornecidos via linha de comando, os quais são reconhecidos por meio de identificadores únicos. Recebe como parâmetros os argumentos da linha de comando - vetor de strings argv, definido pela linguagem C++ -, um arquivo de entrada e um arquivo de saída - variáveis do tipo fstream. A partir daí, identifica os respectivos caminhos e os inicializa. Dado que apenas dois identificadores são válidos, a quantidade de parâmetros que podem ser passados ao programa foi limitada a 15 elementos.

- **insertMail():** Realiza a inserção de uma nova mensagem no servidor de emails com base nas informações contidas em um arquivo de entrada. Recebe como parâmetros um arquivo de entrada, um de saída - variáveis do tipo fstream - e um servidor de emails - variável do tipo MailServer. A partir daí, faz a leitura das informações referentes à mensagem (identificador do usuário destinatário, identificador e corpo da mensagem) da entrada, insere-a no servidor por meio de uma invocação ao método insert() - parametrizado com as informações lidas - e exporta o resultado à saída.

- **removeMail():** Realiza a remoção de uma mensagem do servidor de emails com base nas informações contidas em um arquivo de entrada. Recebe como parâmetros um arquivo de entrada, um de saída - variáveis do tipo fstream - e um servidor de emails - variável do tipo MailServer. A partir daí, faz a leitura das informações referentes à mensagem (identificador do usuário destinatário e identificador da mensagem) da entrada, tenta removê-la do servidor por meio de uma invocação ao método remove() - parametrizado com as informações lidas - e exporta o resultado à saída.

- **searchMail():** Realiza a busca por mensagem no servidor de emails com base nas informações contidas em um arquivo de entrada. Recebe como parâmetros um arquivo de entrada, um de saída - variáveis do tipo fstream - e um servidor de emails - variável do tipo MailServer. A partir daí, faz a leitura das informações referentes à mensagem (identificador do usuário destinatário e identificador da mensagem) da entrada, busca por ela no servidor por meio de uma invocação ao método search() - parametrizado com as informações lidas - e exporta o resultado à saída.

Todos esses procedimentos foram definidos no arquivo **fileoperations.hpp**.

O funcionamento do programa em si, definido pela função **main()** no arquivo **main.cpp**, é apenas uma concatenação das duas etapas. Inicialmente, as variáveis que vão comportar os arquivos de entrada/saída e o servidor de emails são declaradas e inicializadas. Em seguida, o arquivo é percorrido e os comandos nele contidos são operacionalizados. A cada operação, o resultado é exportado ao arquivo de saída. Todos esses processos são mediados pelos procedimentos, classes e métodos definidos.

3. Análise de Complexidade

A complexidade do programa foi avaliada em termos de dois parâmetros de execução: o **número de entradas da tabela hash** e o **número médio de mensagens** contidas em cada uma, dado que todos os algoritmos possuem um custo escalável com alguma dessas variáveis. O número de entradas da tabela - ou, seu tamanho - é um valor **m** fornecido pelo usuário em tempo de execução. O número de mensagens em cada uma, por sua vez, será normalizado como um valor médio **n** para fins de simplificação da análise, já que a maioria dos algoritmos depende somente do tamanho de uma das entradas, aquela sobre a qual irá operar. Dessa forma, é possível estabelecer que, em média, o servidor como um todo comporta **m*n** mensagens. A seguir, tem-se a análise de complexidade daqueles algoritmos mais relevantes para a operação do programa.

- **MailBox::clear():** Percorre de maneira recursiva todas as **n** mensagens (nós) da caixa de entrada (árvore binária) sobre a qual é invocado. Dessa forma, em termos de **tempo**, apresenta uma ordem de complexidade **O (n)**. Em relação ao **espaço**, o método utiliza apenas a própria estrutura de dados na qual atua, de modo que sua ordem de complexidade também seja **O (n)**.

- **MailBox::insert():** Percorre de maneira recursiva e total um tronco da árvore binária correspondente à caixa de entrada sobre a qual é invocado, a fim de nela inserir um novo nó folha. Dessa forma, em termos de **tempo**, há duas situações de execução: um **melhor caso**, dado por uma árvore totalmente balanceada, de modo que, a cada passagem por um nó, o espaço de busca seja reduzido pela metade, originando uma complexidade da ordem **O (log n)**, e um **pior caso**, dado por uma árvore totalmente desbalanceada - ou "degenerada" -, de modo todos os seus **n** nós estejam dispostos de maneira linear, implicando uma complexidade da ordem **O (n)**. Em termos de **espaço**, o método

envolve apenas algumas variáveis auxiliares e a própria estrutura de dados em si (árvore binária), de modo que sua complexidade nesse quesito seja da ordem **O (n)**.

- **MailBox::remove():** Percorre de maneira recursiva um tronco da árvore binária correspondente à caixa de entrada sobre a qual é invocado, a fim de buscar e remover um dado nó. Em termos de **tempo**, há três situações de execução. Em um **melhor caso**, o nó a ser removido está na primeira posição (raiz) da estrutura de dados, de modo que a complexidade seja da ordem **O (1)**. Em um **caso médio**, o referido nó está em alguma posição interna ou em uma folha e a árvore está parcial ou totalmente balanceada, de modo que, a cada verificação de um nó, o espaço de busca seja aproximadamente reduzido pela metade, originando uma complexidade da ordem **O (log n)**. Em um **pior caso**, o nó a ser removido está na última posição de uma árvore degenerada, de modo que seja necessário percorrer todos os demais **n - 1** nós, implicando uma complexidade da ordem **O (n)**. Em termos de **espaço**, o método trabalha com algumas variáveis constantes e com a própria estrutura de dados (árvore binária), de modo que sua complexidade seja da ordem **O (n)**. A remoção em si é realizada pelo método **MailBox::removeAuxiliary()**, o qual apenas realiza operações constantes (movimentações de ponteiros) sobre um nó e, portanto, é da ordem de complexidade **O(1)**, tanto em termos de **tempo**, quanto de **espaço**, não afetando a análise prévia do método MailBox::remove().

- **MailBox::search():** Percorre de maneira recursiva um tronco da árvore binária correspondente à caixa de entrada sobre a qual é invocado, a fim de buscar por um dado nó. Em termos de **tempo**, há três situações de execução. Em um **melhor caso**, o nó procurado está na primeira posição (raiz) da estrutura de dados, de modo que a complexidade seja da ordem **O (1)**. Em um **caso médio**, o referido nó está em alguma posição interna ou em uma folha e a árvore está parcial ou totalmente balanceada, de modo que, a cada verificação de um nó, o espaço de busca seja aproximadamente reduzido pela metade, originando uma complexidade da ordem **O (log n)**. Em um **pior caso**, o nó procurado está na última posição de uma árvore degenerada, de modo que seja necessário percorrer todos os demais **n - 1** nós, implicando uma complexidade da ordem **O (n)**. Em termos de **espaço**, o método trabalha com algumas variáveis constantes e com a própria estrutura de dados (árvore binária), de modo que sua complexidade seja da ordem **O (n)**.

- **hash():** Realiza o mapeamento de uma dada chave de usuário a uma caixa de mensagens do servidor de emails (entrada da tabela hash). O método em si apenas realiza algumas operações constantes, com o auxílio de poucas variáveis, também constantes, de modo que sua complexidade, tanto em termos de **tempo**, quanto de **espaço**, seja da ordem **O (1)**.

- **MailServer::insert():** Realiza uma chamada ao método hash() sobre si mesmo e uma chamada ao método MailBox::insert() sobre um de seus atributos. Dessa forma, em termos de **tempo**, sua complexidade apresenta dois diferentes casos, análogos aos casos do método MailBox::insert(): um **melhor caso**, dado pela ordem O (1) (método hash()) + O (log n) (método MailBox::insert()) = **O (log n)**, e um **pior caso**, dado pela ordem O (1) (método hash()) + O (n) (método MailBox::insert()) = **O (n)**. Em termos de **espaço**, o método trabalha apenas com variáveis de tamanho constante e sobre a própria estrutura de dados (tabela hash), de modo que sua complexidade seja da ordem **O (m*n)** (**m** entradas da tabela, cada uma contendo uma árvore binária de tamanho **n**).

- **MailServer::remove():** Realiza uma chamada ao método hash() sobre si mesmo e uma chamada ao método MailBox::remove() sobre um de seus atributos. Dessa forma, em termos de **tempo**, sua complexidade apresenta três diferentes casos, análogos aos casos do método MailBox::remove(): um **melhor caso**, dado pela ordem O (1) (método hash()) + O (1) (método MailBox::remove()) = **O (1)**, um **caso médio**, dado pela ordem O (1) (método hash()) + O (log n) (método MailBox::remove()) = **O (log n)**, e um **pior caso**, dado pela ordem O (1) (método hash()) + O (n) (método MailBox::remove()) = **O (n)**. Em termos de **espaço**, o método trabalha apenas com variáveis de tamanho constante e sobre a própria estrutura de dados (tabela hash), de modo que sua complexidade seja da ordem **O (m*n)** (**m** entradas da tabela, cada uma contendo uma árvore binária de tamanho **n**).

- **MailServer::search():** Realiza uma chamada ao método hash() sobre si mesmo e uma chamada ao método MailBox::search() sobre um de seus atributos. Dessa forma, em termos de **tempo**, sua complexidade apresenta três diferentes casos, análogos aos casos do método MailBox::search(): um **melhor caso**, dado pela ordem O (1) (método hash()) + O (1) (método MailBox::search()) = **O (1)**, um **caso médio**, dado pela ordem O (1) (método hash()) + O (log n) (método MailBox::search()) = **O (log n)**, e um **pior caso**, dado pela ordem O (1) (método hash()) + O (n) (método MailBox::search()) = **O (n)**. Em termos de **espaço**, o método trabalha apenas com variáveis de tamanho constante e sobre a própria estrutura de dados (tabela hash), de modo que sua complexidade seja da ordem **O (m * n)** (**m** entradas da tabela, cada uma contendo uma árvore binária de tamanho **n**).

- **initializeFiles():** Percorre cada argumento da linha de comando, verificando se há um identificador válido e inicializando a variável correspondente, caso aplicável. Dado que o número de argumentos que podem ser fornecidos é limitado, o método apresenta um comportamento operacional aproximadamente constante, de modo que sua complexidade em termos de **tempo** seja da ordem **O (1)**. Em termos de **espaço**, há apenas o uso de algumas variáveis auxiliares e do vetor de argumentos, todos também aproximadamente constantes, de modo que sua complexidade seja da ordem **O (1)**.

- **insertMail():** Realiza a leitura das informações contidas em um arquivo de entrada - uma operação considerada constante - e invoca o método MailServer::insert() sobre um servidor de emails - objeto do tipo MailServer, fornecido como parâmetro. Dessa forma, em termos de **tempo**, sua complexidade apresenta dois diferentes casos, análogos aos casos do método MailServer::insert(): um **melhor caso**, dado pela ordem O (1) (operações constantes) + O (log n) (método MailServer::insert()) = **O (log n)**, e um **pior caso**, dado pela ordem O (1) (operações constantes) + O (n) (método MailServer::insert()) = **O (n)**. Em termos de **espaço**, apenas trabalha com variáveis constantes locais ou recebidas por referência via parâmetros, de modo que sua complexidade seja da ordem **O (1)**. Um detalhe a ser mencionado é que, nesse caso, a complexidade do método insertMail() - em termos de tempo e de espaço - ainda depende do tamanho da mensagem sendo lida, já que ele trabalha iterando sobre cada palavra e as armazenando em memória interna. Para efeito de simplificação, foi considerado que, em geral, as mensagens irão apresentar um tamanho muito próximo e, portanto, aproximadamente constante. Na prática, entretanto, reitera-se que **há uma grau linear de escalabilidade** com o referido valor variável.

- **removeMail():** Realiza a leitura das informações contidas em um arquivo de entrada - uma operação considerada constante - e invoca o método MailServer::remove() sobre um servidor de emails - objeto

do tipo MailServer, fornecido como parâmetro. Dessa forma, em termos de **tempo**, sua complexidade apresenta três diferentes casos, análogos aos casos do método MailServer::remove(): um **melhor caso**, dado pela ordem O (1) (operações constantes) + O (1) (método MailServer::remove()) = **O (1)**, um **caso médio**, dado pela ordem O (1) (operações constantes) + O (log n) (método MailServer::remove()) = **O (log n)**, e um **pior caso**, dado pela ordem O (1) (operações constantes) + O (n) (método MailServer::remove()) = **O (n)**. Em termos de **espaço**, apenas trabalha com variáveis constantes locais ou recebidas por referência via parâmetros, de modo que sua complexidade seja da ordem **O (1)**.

- **searchMail():** Realiza a leitura das informações contidas em um arquivo de entrada - uma operação considerada constante - e invoca o método MailServer::search() sobre um servidor de emails - objeto do tipo MailServer, fornecido como parâmetro. Dessa forma, em termos de **tempo**, sua complexidade apresenta três diferentes casos, análogos aos casos do método MailServer::search(): um **melhor caso**, dado pela ordem O (1) (operações constantes) + O (1) (método MailServer::search()) = **O (1)**, um **caso médio**, dado pela ordem O (1) (operações constantes) + O (log n) (método MailServer::search()) = **O (log n)**, e um **pior caso**, dado pela ordem O (1) (operações constantes) + O (n) (método MailServer::search()) = **O (n)**. Em termos de **espaço**, apenas trabalha com variáveis constantes locais ou recebidas por referência via parâmetros, de modo que sua complexidade seja da ordem **O (1)**.

A complexidade do programa como um todo, abstraída pela complexidade da função **main()**, é uma função dos diversos aspectos variáveis envolvidos em sua execução. Entretanto, nesse caso, um novo parâmetro de operação deve ser relevado, seja ele o número **k** de comandos no arquivo de entrada. Em especial, cada comando será percorrido e operado, originando uma relação de complexidade linear para com essa variável. Assim, de maneira geral, são executadas na função main() algumas operações constantes - inicialização dos arquivos, instanciação do servidor e verificações de integridade - e **k** comandos, cada um dotado de uma complexidade própria, como definido previamente. No todo, é possível estabelecer que há três principais casos, os quais levam em conta os diversos fluxos de execução existentes no espaço amostral de operacionalização da aplicação.

- **Melhor caso:** Definido por todos os comandos - inserção, remoção e busca - recaírem sobre seu melhor caso - O (log n) para a inserção, O (1) para os demais - ou sobre o caso médio - O (log n) para todos. Em termos de **tempo**, a complexidade do programa é dada por O (1) (operações constantes) + O (k * log n) = **O (k * log n)**. Em termos de **espaço**, será inicializado um servidor de emails com **m** entradas, cada qual com **n** mensagens, de modo que sua complexidade seja da ordem **O (m * n)**.

- **Pior caso:** Definido por todos os comandos - inserção, remoção e busca - recaírem sobre seu pior caso - O (n) para todos. Em termos de **tempo**, a complexidade do programa é dada por O (1) (operações constantes) + O (k * n) = **O (k * n)**. Em termos de **espaço**, será inicializado um servidor de emails com **m** entradas, cada qual com **n** mensagens, de modo que sua complexidade seja da ordem **O (m * n)**.

Um detalhe a ser mencionado é que, caso nenhum comando seja fornecido, o programa apenas executará uma série de operações constantes, de modo que sua complexidade em termos de tempo seja O (1) e, em termos de espaço, O (m) (já que nenhuma inserção será realizada, apenas um servidor contendo m

entradas e nenhuma mensagem será instanciado). Esse caso não foi abordado por não apresentar nenhuma relevância prática, dado que o objetivo do programa é justamente a simulação das operações do servidor.

4. Estratégias de Robustez

A garantia de robustez para a aplicação foi realizada em duas etapas: a **manutenção da integridade dos dados internos** e a concretização de uma **coerência lógica** na operação das classes e dos algoritmos. Para tal, foi utilizado o cabeçalho **exceptions**, definido na biblioteca padrão da linguagem C++, o qual conta com uma série de ferramentas para o tratamento de exceções - vulgo incoerências - no fluxo de execução.

A primeira etapa envolve toda e qualquer manipulação de dados dentro do programa, desde a leitura de um arquivo de entrada, à exportação a um arquivo de saída. Em um primeiro momento, é verificado - por meio de expressões condicionais - se tais arquivos foram fornecidos via linha de comando à aplicação e, caso sejam, se sua inicialização procedeu sem erros. Em seguida, cada operação de leitura e de escrita também é validada quanto à ocorrência de anormalidades - novamente, por meio de condicionais. Em qualquer um dos casos, a detecção de inconsistências implicará o lançamento de uma exceção e o encerramento da execução do programa. Tal abordagem foi adotada tendo em vista que os dados são uma parte fundamental da operação e, portanto, qualquer erro a eles relacionado originaria um comportamento inesperado/indesejado.

A segunda etapa, por sua vez, aborda a utilização das classes definidas, o que compreende a instanciação de objetos, a invocação de métodos e a passagem de parâmetros. A estratégia de encapsulamento adotada - estabelecimento de uma interface para com o usuário apenas mediante a classe MailServer, omitindo as classes MailBox e Mail - garante uma correta operacionalização das classes MailBox e Mail pela própria implementação interna do programa. A utilização da classe MailServer, por sua vez, foi mantida por meio de estratégias que direcionam um fluxo inválido a um fluxo válido de operação ou encerram a atividade da aplicação, cada qual quando aplicável. Em especial, uma situação foi relevada como potencialmente perigosa e diz respeito à passagem de um tamanho inválido à criação do servidor de emails (tabela hash), o que lança uma exceção e interrompe o programa, dado que esse tamanho não pode ser alterado e, portanto, não pode ser inicializado de maneira inválida. Todas as demais operações possíveis foram consideradas como sempre apresentando um fluxo válido de execução.

5. Análise Experimental

A análise experimental foi realizada por meio da biblioteca **memlog**, a qual contém uma série de ferramentas dedicadas à avaliação do tempo de execução e dos acessos à memória por parte do programa, e da aplicação **analisamem**, a qual permite a geração de gráficos para a visualização dos resultados da primeira.

a. Desempenho Computacional

A análise de desempenho computacional foi realizada com o objetivo de se avaliar a **performance das três operações definidas** - inserção, remoção e consulta - em diferentes cenários de execução, cada qual correspondente à variação de algum possível parâmetro que afete o comportamento do programa. Um detalhe a ser mencionado é que, para efeito de normalização, cada entrada do servidor de emails (tabela hash) receberá mensagens de um único usuário. Outro adendo diz respeito às operações de consulta e de remoção, as quais serão realizadas sempre das extremidades da árvore à sua raiz, a fim de se avaliar o pior caso de execução. No total, foram realizadas cinco baterias de testes, as quais serão detalhadas e analisadas a seguir.

1) Esta tem por objetivo avaliar o comportamento das operações para **variações no tamanho da tabela hash**. Em particular, foram realizados 5 testes, cada qual com tamanhos crescentes para a tabela. Em cada um, serão

executadas 1000000 operações de cada tipo - na ordem inserção (para a população da tabela), consulta e remoção - apenas sobre a primeira entrada. Foram utilizadas mensagens com 5 palavras, cada qual com 5 caracteres. A inserção é feita de maneira **balanceada** sobre a árvore binária correspondente a uma dada entrada. Os tempos individuais de cada uma das operações são exibidos a seguir.

Parâmetro (n° de entradas)	**Tempo (Inserção)**	**Tempo (Consulta)**	**Tempo (Remoção)**
100	2.690665657	1.511748028	1.173053404
200	2.691308698	1.514850419	1.187892754
300	2.687343505	1.521299733	1.176979803
400	2.713002562	1.521867967	1.169368465
500	2.697828416	1.511228999	1.194031870

Os resultados revelam de maneira nítida que a quantidade de entradas dotada pela tabela hash - ou, em outras palavras, a quantidade de caixas de entrada existentes no servidor de emails - não afeta o tempo de operacionalização dos comandos em uma delas. Esse comportamento, por sua vez, é compatível com a característica direta dos acessos a uma tabela hash. No caso, eles são feitos apenas por uma operação constante - o cálculo do índice com base em uma chave -, de modo que a quantidade de possíveis índices não afete o acesso a um determinado e, consequentemente, a operacionalização das mensagens, a qual irá depender apenas da configuração interna da entrada em si, seja ela uma árvore binária de pesquisa.

2) Esta tem por objetivo avaliar o comportamento das operações para **variações no tamanho da tabela hash e em sua população com dados de usuários**. De modo geral, é similar à primeira bateria, com a diferença de que a inserção de 10000000 de mensagens é realizada sobre as duas primeiras entradas - totalizando 2000000 mensagens -, enquanto as operações de consulta e remoção permanecem da mesma forma. Dessa maneira, é possível avaliar como a população de mais entradas, e não apenas sua existência, afeta as operações executadas em apenas uma delas. A inserção é feita de maneira **balanceada** sobre a árvore binária correspondente a uma dada entrada. Os tempos individuais de cada uma das operações são exibidos a seguir.

Parâmetro (n° de entradas)	**Tempo (Inserção)**	**Tempo (Consulta)**	**Tempo (Remoção)**
100	5.356210621	1.529162454	1.180001282
200	5.286507300	1.497286436	1.181100286
300	5.399408995	1.514640287	1.192009028
400	5.373502697	1.572212420	1.168646135
500	5.435636899	1.528122548	1.195936701

Novamente, reitera-se o comportamento observado na bateria de testes anterior: a variação do tamanho interno da tabela hash não afeta o tempo de operacionalização de alguma de suas entradas. Nesse caso, entretanto, um novo parâmetro foi alterado, sendo ele a quantidade de mensagens em cada entrada. Ao contrário dos testes anteriores, também foram inseridas mensagens na segunda posição da tabela. Uma das consequências disso, como é de se esperar, foi um maior tempo de inserção das mensagens, já que o dobro dessas operações estão sendo realizadas. Por outro lado, os tempos de consulta e de remoção, sendo estas realizadas apenas na primeira posição, permaneceu o mesmo, resultado que confirma mais uma vez a natureza direta dos acessos à tabela e a certa independência apresentada pelas estruturas de dados que configuram cada uma de suas entradas, já que a utilização de uma não depende do estado da outra.

3) Esta tem por objetivo avaliar o comportamento das operações para **variações no tamanho (quantidade de mensagens) da entrada (árvore binária)** sobre a qual são executadas. Foram realizados cinco testes, cada qual com quantidades crescentes de mensagens - compostas por 5 palavras com 5 caracteres cada. Todas as operações - inserção (para a população da tabela), consulta e remoção, nessa ordem - foram efetuadas sobre uma tabela hash com uma única entrada, a qual será populada de maneira **balanceada** (em termos de sua árvore binária). Nesse caso, são fornecidos dois resultados: o resultado prático, obtido no experimento, e um resultado teórico, o qual foi calculado com base em um primeiro teste e na ordem de complexidade prevista para o algoritmo. Os tempos individuais de cada uma das operações são exibidos a seguir.

Resultado Prático

Parâmetro (n° de mensagens)	**Tempo (Inserção)**	**Tempo (Consulta)**	**Tempo (Remoção)**
1000000	2.667817466	1.514859536	1.185613613
2000000	5.415433565	3.033598485	2.345962878
3000000	8.300497970	4.536497759	3.545863599
4000000	11.116363592	6.226218660	4.838114426
5000000	13.739430085	7.641477019	6.000513071

Resultado Teórico

Parâmetro (n° de mensagens)	**Tempo (Inserção)**	**Tempo (Consulta)**	**Tempo (Remoção)**
	O (n * log n)	**O (n * log n)**	**O (n * log n)**
1000000	2.667817466	1.514859536	1.185613613
2000000	5.603332625	3.181725125	2.490195646
3000000	8.639888606	4.905964449	3.839681566
4000000	11.74206064	6.667462357	5.218328133
5000000	14.89302432	8.456665494	6.618658359

A situação abordada nessa bateria de testes, como mencionado, avalia o comportamento do programa quando se considera a execução de operações sobre uma única caixa de entrada, a qual comporta uma quantidade **n** de mensagens distribuídas de maneira **balanceada** (em termos da árvore binária). Todas são inteiramente inseridas, consultadas e removidas, totalizando, portanto, **n** comandos de cada tipo. Todas essas características - como visto na seção **[Análise de Complexidade]** - implicam em uma complexidade da ordem **O (n * log n)** para cada uma das categorias de operações executadas - cada qual apresenta complexidade O (log n) e é executada **n** vezes. Os resultados obtidos confirmam de maneira positiva a análise feita, já que os tempos de execução práticos se aproximam bastante dos teóricos.

4) Esta apresenta o mesmo objetivo e as mesmas condições de amostragem da terceira bateria, com a principal diferença sendo na maneira como as mensagens são inseridas, a qual, desta vez, é **degenerada** em termos de sua árvore binária. O tamanho das entradas também é proporcionalmente menor em função dos tempos de execução. Os tempos individuais de cada uma das operações são exibidos a seguir.

Resultado Prático

Parâmetro (n° de mensagens)	**Tempo (Inserção)**	**Tempo (Consulta)**	**Tempo (Remoção)**
5000	0.422730310	0.080925410	0.069456781

10000	1.781899674	0.307300149	0.262702078
15000	4.157172082	0.700120917	0.584474270
20000	7.460334753	1.216986020	1.025412387
25000	11.478777106	1.910742950	1.567830937

Resultado Teórico

Parâmetro (n° de mensagens)	**Tempo (Inserção) O (n ^ 2)**	**Tempo (Consulta) O (n ^ 2)**	**Tempo (Remoção) O (n ^ 2)**
10000	0.422730310	0.080925410	0.069456781
20000	1.69092124	0.32370164	0.277827124
30000	3.80457279	0.72832869	0.625111029
40000	6.76368496	1.29480656	1.111308496
50000	10.56825775	2.02313525	1.736419525

Esta bateria de testes avalia o comportamento do programa para as mesmas operações em comparação à anterior, porém, desta vez, preenchida de maneira **degenerada** (em termos da árvore binária). Tal configuração, por sua vez, implica em essa estrutura de dados se manifestar de maneira semelhante a uma lista sequencial, apresentando um comportamento linear. Dessa forma, as operações que antes apresentavam complexidade da ordem O (log n), agora serão O (n), de modo que suas **n** execuções (cada) demonstrem um caráter quadrático (**O (n ^ 2)**). Novamente, os resultados práticos aproximam-se bastante dos resultados teóricos, confirmando a análise realizada.

5) Esta tem por objetivo avaliar o comportamento das operações para **diferentes tamanhos de mensagens**. Foram realizados 5 testes, em cada qual elas apresentam um número crescente de palavras. Foi adotada uma tabela hash com apenas 1 entrada, sobre a qual são executadas 10000 operações de cada tipo - inserção (para população da tabela), consulta e remoção, nessa ordem. A população é feita de maneira **balanceada** em termos da árvore binária. Os tempos individuais de cada uma das operações são exibidos a seguir.

Parâmetro (n° de palavras)	**Tempo (Inserção)**	**Tempo (Consulta)**	**Tempo (Remoção)**
1000	0.973297375	0.050665873	0.013320601
2000	1.900806264	0.077213070	0.013633256
3000	2.862660648	0.121746309	0.014003498
4000	3.968670322	0.152154614	0.013920142
5000	4.786034480	0.184870482	0.014666497

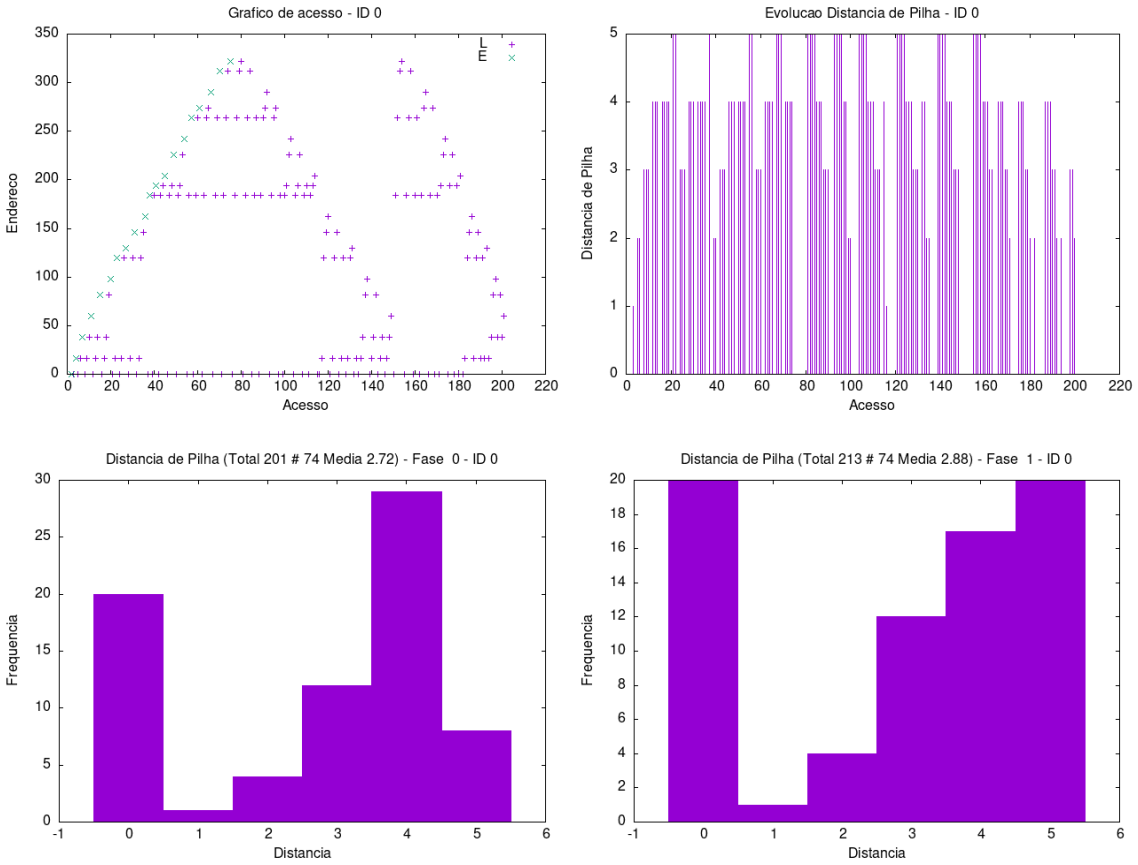
Na seção **[Análise de Complexidade]**, foi mencionado que, para fins de simplificação, o impacto da variação do tamanho das mensagens na complexidade do algoritmo seria desconsiderado. Porém, é evidente que os procedimentos que lidam diretamente com a leitura e com a escrita das mensagens dependem em algum grau de seu tamanho. Em particular, o método insertMail(), como mencionado na seção referida, apresenta uma complexidade que escala de maneira linear com a quantidade de palavras presentes na mensagem com a qual lida. Este comportamento fica nítido com os resultados práticos obtidos, nos quais o tempo de execução do comando de inserção escala proporcionalmente com o aumento linear do tamanho das mensagens. Outro detalhe a ser notado é que o tempo de execução do comando de consulta também

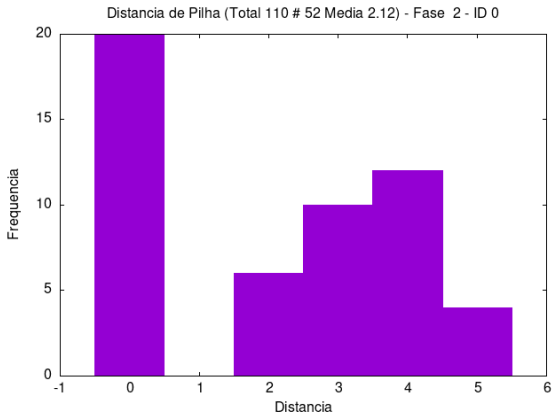
aumenta conforme o tamanho da mensagem, já que essa operação também envolve a movimentação desse elemento. O comando de remoção, porém, não é afetado, já que não lida com as mensagens em si.

b. Localidade de Referência

A análise de localidade de referência foi realizada com o objetivo de se verificar o **comportamento do programa** - em termos de **acessos para leitura/escrita em memória** - em diferentes cenários de execução, cada qual correspondente a uma configuração da entrada de dados, sendo este o principal fator na localidade da aplicação. Para tal, foi utilizado como instrumento de registro a **chave de cada mensagem** - atributo da classe Mail -, dado que ela é a referência sobre a qual manipulações na estrutura de dados são feitas e, portanto, representa com um grau de precisão satisfatório o comportamento do programa. Um detalhe a ser mencionado é que as operações de consulta e de remoção são realizadas sempre das extremidades da árvore à sua raiz, a fim de se avaliar o pior caso de execução. No total, foram realizadas quatro baterias de testes, as quais serão detalhadas e analisadas a seguir.

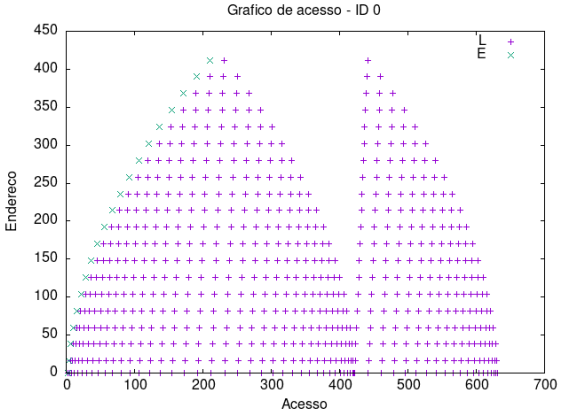
1) Avalia o comportamento do programa em termos de uma de suas entradas, isto é, de uma árvore binária, sendo esta **balanceada**. Foram realizadas 20 operações de cada tipo - inserção (para população de dados), consulta e remoção, nessa ordem -, sendo a análise de distância de pilha feita em fases referentes a cada uma. As mensagens apresentam 5 palavras com 5 caracteres cada.

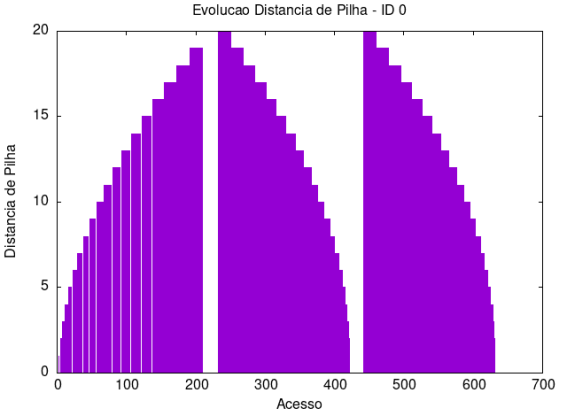


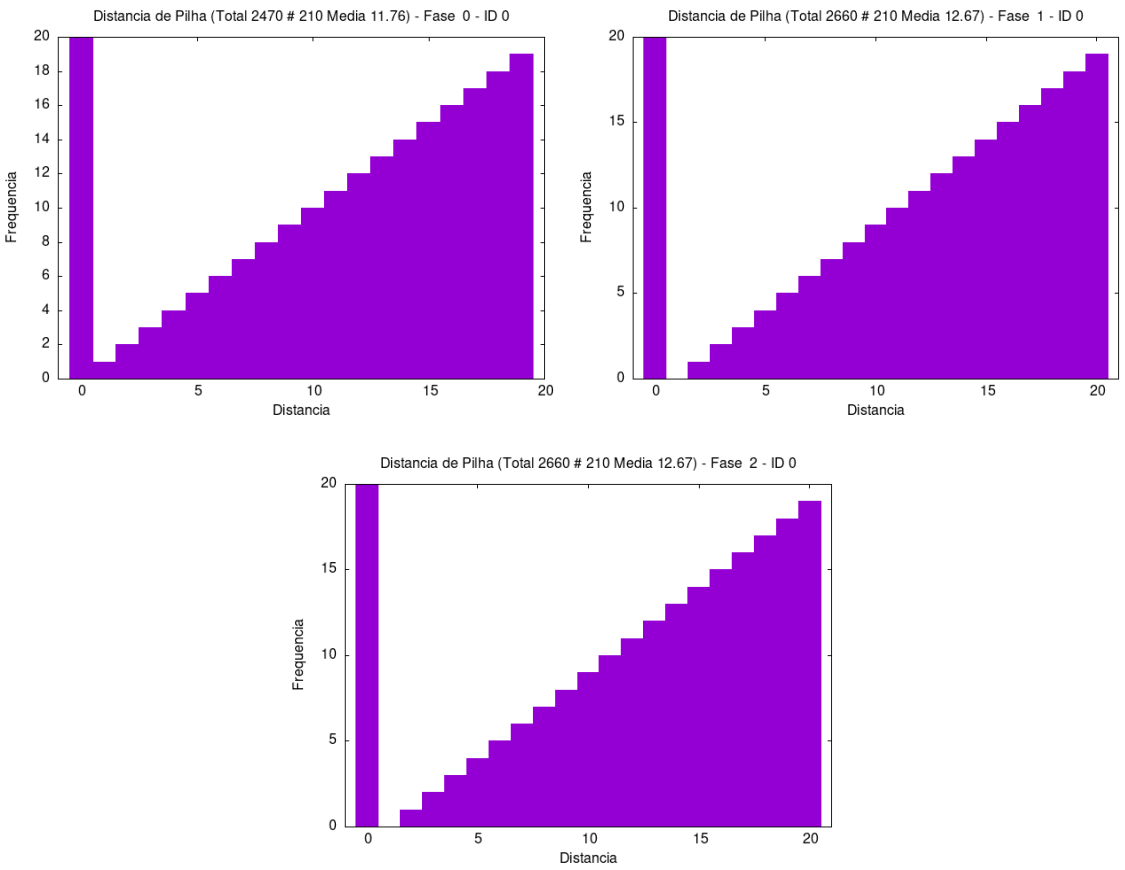


No diagrama de acessos, é possível observar de maneira nítida como a árvore binária é percorrida para a realização de uma inserção. Em particular, um tronco formado por nós alocados de maneira não sequencial é progressivamente lido até que se chegue na posição correta de inserção, na qual a mensagem a ser inserida é escrita, processo este iterado para todas as 20 inserções. O mesmo comportamento pode ser observado para as operações de consulta e de remoção, as quais, novamente, percorrem nós de maneira não sequencial - o equivalente a "andar por um tronco" - até encontrarem o desejado. Um detalhe a ser notado é que a silhueta das etapas de consulta e de remoção são idênticas. Em essência, ambas essas operações são pautadas em um mesmo processo: buscar por um determinado nó e o operar de alguma forma - retornando-o, no caso da consulta, e o removendo, no caso da remoção -, de modo que apresentem o mesmo comportamento de acessos. No que tange a distância de pilha, em todas as fases, é possível observar que a silhueta não apresenta um padrão totalmente uniforme, mas com alguma coerência, a qual reflete justamente a maneira como nós não sequenciais, porém distribuídos de maneira balanceada, são percorridos, originando blocos de distância de pilha similares, correspondentes a caminhamentos por troncos.

2) Avalia o comportamento do programa em termos de uma de suas entradas, isto é, de uma árvore binária, sendo esta **degenerada**. Foram realizadas 20 operações de cada tipo - inserção (para população de dados), consulta e remoção, nessa ordem -, sendo a análise de distância de pilha feita em fases referentes a cada uma. As mensagens apresentam 5 palavras com 5 caracteres cada.

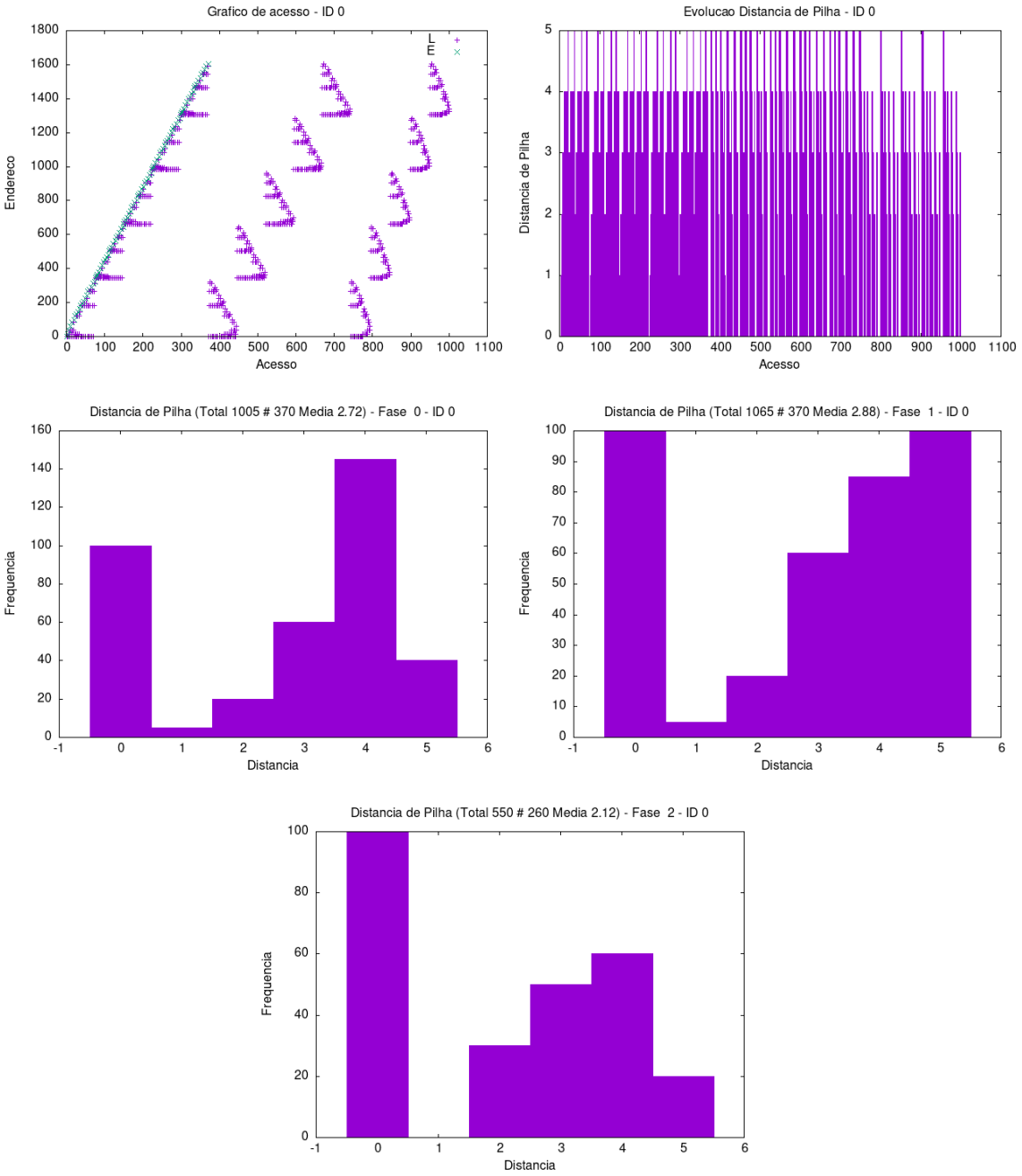




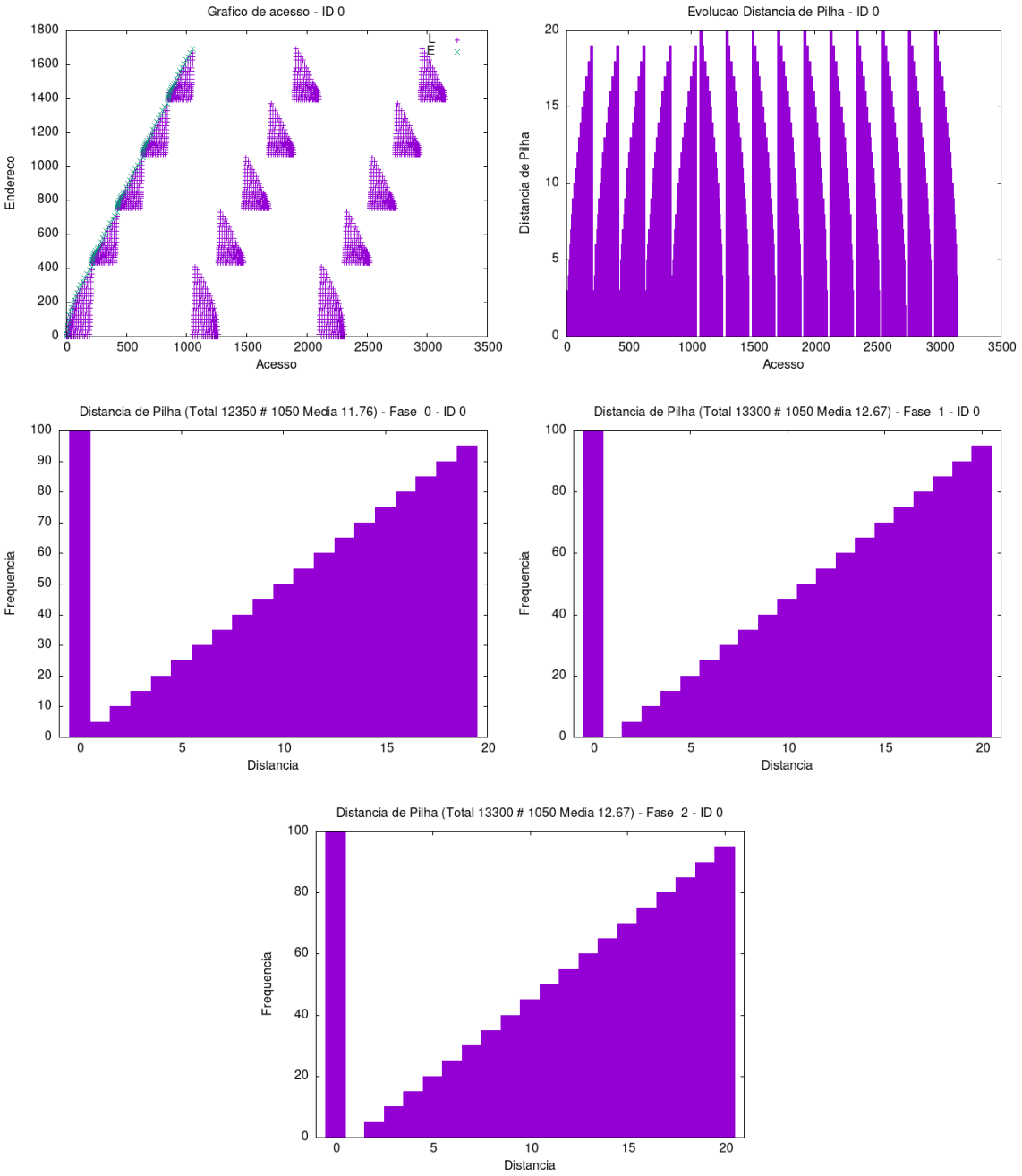


Assim como mencionado na seção **[Análise Experimental / Desempenho Computacional]**, a configuração degenerada de uma árvore binária a torna, em essência, uma lista sequencial. No diagrama de acessos, essa característica é nítida na medida em que os caminhamentos feitos para a inserção de uma nova mensagem na extremidade da estrutura de dados, antes sobre troncos, agora procedem sobre todas as suas células. O mesmo ocorre para as operações de consulta e de remoção, as quais percorrem todos os nós até encontrar o desejado, sobre o qual atuam de maneira conforme. No tangente à distância de pilha, a frequência de acessos agora apresenta uma silhueta uniforme e é organizada em blocos, cada qual referente a um caminhamento sequencial por todos os nós para determinado tamanho da estrutura, originando um aspecto de escadaria para todas as fases operacionais. Um detalhe a ser notado é a maneira como a configuração degenerada deteriora o desempenho da aplicação em termos de localidade de referência, o que pode ser observado em valores significativamente mais elevados de distância de pilha.

3) Avalia o comportamento do programa em termos de 5 entradas, cada qual sendo uma árvore binária **balanceada**. Em cada uma, foram realizadas 20 operações de cada tipo - inserção (para população de dados), consulta e remoção, nessa ordem -, sendo a análise de distância de pilha feita em fases referentes a cada categoria. As mensagens apresentam 5 palavras com 5 caracteres cada.



4) Avalia o comportamento do programa em termos de 5 entradas, cada qual sendo uma árvore binária **degenerada**. Em cada uma, foram realizadas 20 operações de cada tipo - inserção (para população de dados), consulta e remoção, nessa ordem -, sendo a análise de distância de pilha feita em fases referentes a cada categoria. As mensagens apresentam 5 palavras com 5 caracteres cada.



Estes dois últimos testes reiteram a natureza direta dos acesso às entradas de uma tabela hash, assim como o certo grau de independência que elas apresentam entre si. Em particular, é possível ver que a configuração observada nos primeiro e segundo testes para uma das entradas é apenas replicada para as demais quando operadas da mesma maneira nos terceiro e quarto testes, respectivamente.

6. Conclusões

O presente trabalho prático teve por objetivo o desenvolvimento de habilidades relacionadas ao projeto, à implementação e à análise de algoritmos e de estruturas de dados, sob o pretexto do desenvolvimento de um simulador para um servidor de emails. Em especial, três tipos abstratos de dados foram definidos: o servidor de emails, suas caixas de mensagens e a mensagem em si. Sobre eles, foram

especificadas operações de inserção, de consulta e de remoção. Cada um desses elementos foi trabalhado, implementado e testado experimentalmente.

Uma das maiores dificuldades encontradas diz respeito às análises de complexidade e experimental. Em particular, o modelo da aplicação tal como foi dado, sendo ele uma hierarquia de estruturas de dados baseadas em diferentes identificadores, implicou um notável grau de complexidade para a análise e para a testagem. Esse encadeamento de diferentes algoritmos originou uma vasta gama de fluxos de execução, cada qual dotado de um comportamento e de uma complexidade próprios, o que agregou maiores complicações às tarefas de validação e experimentação.

Entretanto, foi justamente essa dificuldade que abriu margem para os maiores ganhos. Em especial, a maneira como as estruturas de dados foram combinadas e algoritmos, adaptados ao seu funcionamento, permitiu uma ampliação das noções de uso e de implementação desses conhecimentos. Além disso, a adoção de uma contextualização verossímil também agregou benefícios no que tange a manifestação de saberes teóricos na prática computacional, novamente colaborando para uma expansão de visão e abstração.

7. Bibliografia

Stack Exchange, Inc.. *Stack Overflow* [Online]. Disponível em: https://stackoverflow.com/ (Acessado em: 19/07/2022).

CPlusPlus. *CPlusPlus* [Online]. Disponível em: https://www.cplusplus.com/ (Acessado em: 19/07/2022).

GeeksforGeeks. *GeeksforGeeks* [Online]. Disponível em: https://www.geeksforgeeks.org/ (Acessado em: 19/07/2022).

8. Instruções para Compilação e Execução

1. Extraia o conteúdo do arquivo .zip disponibilizado.

2. Acesse o diretório **</TP>** por meio do comando **<$ cd TP>**.

3. Execute no terminal o comando **<$ make all>**. Esse comando deverá gerar um executável denominado **tp3** no diretório **</TP/bin>**.

4. A execução do programa é realizada por meio do comando **<$ ./bin/tp3>** seguido dos parâmetros enumerados abaixo, em qualquer ordem:

> **-i (caminho do arquivo de entrada)** : arquivo de entrada de dados para o programa.
> **-o (caminho do arquivo de saída)** : arquivo de destino para a saída do programa.

Um exemplo de execução seria **<$ ./bin/tp3 -i entrada.txt -o saida.txt>**. Os parâmetros são **obrigatórios**.

5. O arquivo de saída contendo o resultado do programa para a entrada correspondente deve ser gerado (sobrescrito caso já exista, criado caso ainda não exista) no caminho fornecido.