Trabalho Prático 3 – Servidor de e-mails otimizado

Carlos Henrique Brito Malta Leão

Matrícula: 2021039794

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

chbmleao@ufmg.br

1. Introdução

O objetivo deste trabalho é a implementação do simulador de um servidor de e-mails, em que, além do gerenciamento adequado da memória do sistema, há foco na otimização da pesquisa por usuários e mensagens. Dessa forma, o sistema implementado apresenta suporte à entrega, consulta e remoção de mensagens para usuários, em que o sistema funciona corretamente ao executar as diversas operações do servidor em diferentes situações.

Esta documentação tem como objetivo explicitar como foi realizada a implementação desse sistema, além de realizar uma análise explicativa sobre como o simulador do servidor de e-mails funciona. Ademais, também serão explicitados a robustez, abstração e desempenho do programa, possibilitando um melhor entendimento de como o sistema funciona na prática.

Para a resolução do problema supracitado, foi criado um programa na linguagem C++, que utiliza quatro principais classes: *Email, Node, BinaryTree* e *Hash_BT*. Essas quatro classes representam em conjunto uma tabela hash de árvores binárias. Dessa forma, a classe *Email* representa um e-mail, que são armazenados em nós, representados pela classe *Node*. Esses nós, fazem parte de uma árvore binária, *BinaryTree*, que são armazenadas em um arranjo na classe *Hash_BT,* que é utilizada para a realização de pesquisas de e-mails. Por fim, temos a estrutura *memlog,* que auxilia no processo de análise de desempenho do programa e as funções *msgassert,* que são utilizadas de forma a aumentar a robustez do programa.

Por fim, ao decorrer dessa documentação, alguns aspectos sobre o trabalho serão melhor explicados, como a descrição da implementação na seção 2 e a análise de complexidade na seção 3. Em seguida, nas seções 4 e 5 serão explicitadas as estratégias de robustez e a análise experimental. Por fim, teremos a conclusão do trabalho, sumarizando o que foi aprendido durante seu desenvolvimento.

2. Método

Durante essa seção, será realizada uma descrição da implementação do programa estruturado na linguagem C++. Dessa forma, serão detalhadas as estruturas de dados, classes e funções implementadas.

2.1 Email

Primeiramente, a classe mais simples do programa é a classe *Email*, que simula um objeto de e-mail, armazenando três atributos, sendo a mensagem, o identificador do usuário destinatário e o identificador do e-mail em si. A mensagem é armazenada em um atributo de tipo *string* e os identificadores são armazenadas em inteiros.

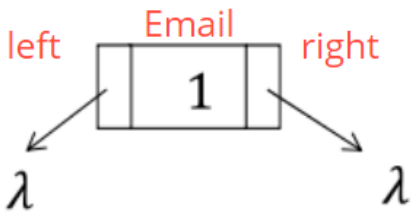
Nesse sentido, essa classe é muito simples e apresenta apenas algumas funções:

- *Email():* Inicializa um e-mail com mensagem vazia e identificadores com -1.
- *Email(emailKey, userKey, message):* Inicializa um e-mail com os atributos recebidos como parâmetro.
- *getEmailKey():* Retorna o identificador de e-mail.
- *getUserKey():* Retorna o identificador do destinatário.
- *getMessage():* Retorna a mensagem do e-mail.
- *print():* Função para depuração que imprime os atributos do e-mail.

2.2 Node

A classe *Node* também é muito simples, ela representa uma célula, nó, de uma árvore binária, que será melhor explicada em seguida. Dessa forma, essa classe apresenta apenas três atributos, o e-mail que será armazenado na célula e os endereços das células localizadas na esquerda e na direita, no formato de um nó de uma árvore binária.

Um diagrama esquemático da classe nó pode ser visto na figura a seguir. Nesse sentido, o número 1 representa o identificador de e-mail armazenado e 'left' e 'right' representam os endereços das células da esquerda e direita respectivamente.



Por fim, a função apresenta apenas duas funções, que são seus construtores:

- *Node():* Inicializa um nó com um e-mail vazio e endereços da esquerda e direita nulos.
- *Node(email):* Inicializa um nó com um e-mail recebido como parâmetro e endereços da esquerda e direita nulos.

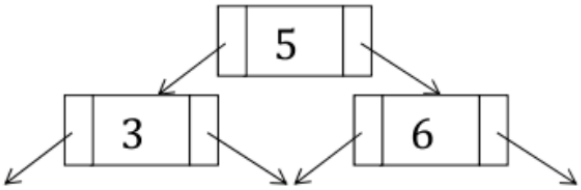
2.3 BinaryTree

A classe *BinaryTree* é mais complexa, ela representa uma árvore binária formada por nós que armazenam e-mails. Nessa árvore binária, cada nó pode ter no máximo dois filhos, que são subárvores da esquerda e da direita. Nesse sentido, considerando um nó em específico, todos os nós que forem armazenados na sua subárvore da esquerda, devem possuir um identificador de e-mail menor que o que ele possui, enquanto a subárvore da direita deve apresentar apenas identificadores de e-mail maiores. Essa lógica serve para todos os nós da árvore, o que garante melhor eficiência para métodos de inserção, remoção e pesquisa de e-mails.

Ademais, a classe apresenta um atributo que armazena a raiz da árvore, a partir desse nó, é possível chegar em qualquer outro nó da árvore binária. Além disso, a árvore não apresenta elementos repetidos, já que cada e-mail enviado e recebido apresenta identificadores distintos.

Um diagrama esquemático da árvore binária implementada pode ser visto na figura a seguir. Nesse sentido, os números representam o identificador de e-mail

armazenada em cada célula, enquanto as setas representam os endereços da esquerda e direita armazenados.



Por fim, essa classe apresenta algumas funções específicas, que abstraem e aumentam a robustez do código. Nesse sentido, percebemos que as funções de inserção, remoção e pesquisa são muito semelhantes, devido ao funcionamento da árvore. Nesse sentido, essa funções são recursivas, em que o caso base ocorre quando a função chega em um nó com endereço nulo, isso significa que a árvore chegou em sua extremidade, uma folha, e que não há mais nós para percorrer.

- *BinaryTree():* Construtor da classe que atribui endereço nulo para a raiz.
- *~BinaryTree():* Destrutor da classe que chama a função *clean*.
- *insert(email):* Insere um e-mail na árvore binária chamando a função *insertRecursive* e passando o e-mail e a raiz da árvore como parâmetros.
- *insertRecursive(currentNode, email):* Essa é uma função recursiva para a inserção de e-mails na árvore. O caso base da função ocorre quando o nó atual apresenta endereço nulo, dessa forma é possível inserir um novo nó na árvore. Caso contrário, se a chave do e-mail a ser inserido é menor que a chave do nó atual, a função começa a tratar o nó da esquerda, caso contrário a função chama o nó da direita. Dessa forma, a função percorre os elementos da árvore até chegar em um nó com endereço nulo.
- *printInOrder():* Imprime os e-mails da árvore binária em ordem crescente dos identificadores de e-mail chamando a função *printInOrderRecursive*.
- *printInOrderRecursive(currentNode):* Função recursiva para a impressão dos e-mails da árvore em ordem. O caso base da função ocorre se o nó atual tiver endereço nulo. Caso contrário, a função chama o nó a esquerda, imprime o e-mail, e em seguida chama o nó a direita. Dessa forma, os nós da árvore binária são impressos em ordem.
- *clean():* Libera a memória alocada para os nós da árvore chamando a função *cleanRecursive* e atribuindo o endereço nulo para a raiz.
- *cleanRecursive(currentNode):* Função recursiva que libera a memória alocada para os nós da árvore. O caso base da função ocorre se o nó atual tiver o endereço nulo. Caso contrário, a função percorre a árvore até os nós da extremidade, os excluindo primeiro até chegar a raiz.
- *search(emailKey, userKey):* Pesquisa na árvore um e-mail com os identificadores de e-mail e usuário recebidos como parâmetros chamando a função *searchRecursive*.
- *searchRecursive(currentNode, emailKey, userKey):* Função recursiva que pesquisa um e-mail específico na árvore. O caso base da função ocorre se o nó atual tiver o endereço nulo. Caso contrário, se o identificador do e-mail for menor que o e-mail atual, o nó da esquerda é chamado, caso contrário, se o identificador for maior que o atual, o nó da direita é chamado, caso contrário, se o identificador do usuário for igual ao do e-mail atual, a função retorna o e-mail atual, caso contrário, a função retorna um e-mail vazio.

- *remove(emailKey, userKey):* Remove um nó da árvore com os identificadores de e-mail e usuário recebidos como parâmetros, chamando a função *removeRecursive*.
- *removeRecursive(currentNode, emailKey, userKey):* Função recursiva que remove um nó específico da árvore. O caso base da função ocorre se o nó atual tiver o endereço nulo. Caso contrário, o processo é muito semelhante ao procedimento da função de pesquisa, mas ao invés de retornar o e-mail procurado, o nó procurado tem sua memória liberada. Ademais, ao liberar a memória, se o nó da direita for nulo, o nó atual recebe os valores do nó da direita, caso contrário, se o nó da esquerda for nulo, o nó atual recebe os valores do nó da esquerda. Caso contrário, se os nós da esquerda e direita não sejam nulos, é preciso chamar a função auxiliar *previous*, passando o nó atual e o nó da esquerda. Por fim, caso o nó procurado exista e seja deletado, a função retorna o booleano verdadeiro, caso contrário retorna falso.
- *previous(Node q, Node r):* Função recursiva que, caso necessário, reestrutura a árvore após uma remoção. O caso base da função ocorre quando o nó atual tiver o endereço nulo. Caso contrário, a função é chamada novamente com o endereço do nó a ser retirado e o nó a direita do antecessor. Além disso, a função atribui ao antecessor os valores do nó a esquerda do antecessor, e libera a memória desse antecessor.

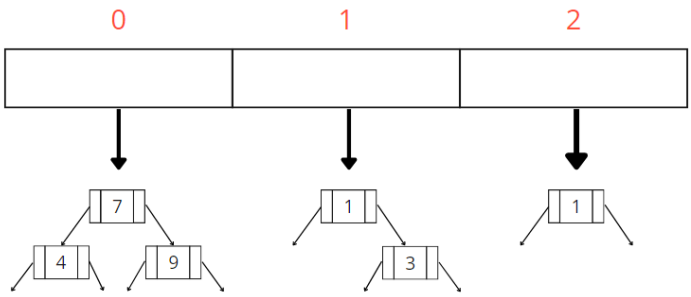
2.4 Hash_BT

A classe *Hash_BT* implementa um algoritmo de pesquisa do tipo Hashing, que são algoritmos que efetuam uma transformação aritmética sobre a chave de pesquisa para encontrá-la. Nesse sentido, os registros armazenados em uma tabela são diretamente endereçados a partir de uma transformação aritmética sobre a chave de pesquisa. Além disso, a busca é feita por meio de operações aritméticas que transformam a chave em endereços em uma tabela. A utilização de árvores binárias deve-se ao tratamento de colisões.

Dessa forma, na implementação utilizada, é criada uma tabela de m árvores binárias, em que o valor de m é especificado no início do arquivo de entrada. Nesse sentido, a operação aritmética utilizada para encontrar o endereço da tabela é o resto da divisão do identificado do usuário pelo tamanho da tabela de árvores, esse resto é a posição da tabela de árvores binárias que o e-mail deve ser inserido.

Porém, considerando que duas ou mais chaves podem ser transformadas em um mesmo endereço da tabela, ou seja, o e-mail de mais de um usuário pode ser armazenado em uma mesma árvore binária. Dessa forma, ao realizar buscas nas árvores binárias, é preciso considerar tanto o identificador do e-mail quanto o do usuário, para não realizar operações equivocadas.

Um diagrama esquemático do Hash de árvores binárias implementado pode ser visto na figura a seguir.



Por fim, essa classe apresenta algumas funções específicas, que abstraem e aumentam a robustez do código. Nesse sentido, a maioria delas segue o mesmo procedimento, primeiro se utiliza a operação aritmética supracitada para encontrar a posição da tabela de árvores binárias a ser utilizada, e em seguida utiliza uma função dessa árvore.

- *Hash_BT(tableSize):* Construtor da classe que atribui o valor do tamanho da tabela e aloca o espaço na memória para as árvores binárias.
- *hash(key):* Função aritmética do Hashing, retorna o resto da divisão do identificador do usuário pelo tamanho da tabela, ou seja, retorna a posição que deve ser inserida na tabela.
- *search(userKey, emailKey):* Pesquisa um e-mail com os identificadores recebidos como parâmetro. Primeiramente encontra a posição chamando a função *hash*, e, em seguida, chama a função *search* da árvore binária na posição encontrada.
- *insert(userKey, email):* Insere um e-mail na tabela hash. Primeiramente encontra a posição que deve ser inserido o e-mail na tabela chamando a função hash, e, em seguida, chama a função *insert* da árvore binária na posição encontrada.
- *remove(userKey, emailKey):* Remove um e-mail da tabela hash. Primeiramente encontra a posição que deve ser removido o e-mail na tabela chamando a função hash, e, em seguida, chama a função *remove* da árvore binária na posição encontrada.
- *print():* Função utilizada para depuração de código, que imprime todos os e-mails armazenados na tabela hash.

3. Análise de complexidade

Esta seção apresenta a análise de complexidade de tempo e espaço para a execução completa do programa, desde a leitura do arquivo até a impressão de todos os comandos. Nesse sentido, realizaremos uma comparação, descrevendo as vantagens e desvantagens de se utilizar métodos de pesquisa e ordenação mais sofisticados. Além disso, realizaremos a análise de complexidade de tempo primeiro e, em seguida, a análise de espaço.

3.1 Comparação dos métodos

Como já foi supracitado, foram utilizadas duas estruturas de dados, a tabela hash e a árvore binária. Nesse sentido, ambas apresentam métodos de pesquisa e ordenação mais sofisticados que possuem vantagens e desvantagens. A seguir, iremos comparar os lados positivos e negativos dessas implementações.

Primeiramente a árvore binária apresenta uma série de vantagens, em relação as suas três principais funções de pesquisa, remoção, e consulta, em que ambas apresentam custo $O(\log n)$ para o caso médio. Porém, esse custo depende muito de como foi realizada a inserção da árvore, que, se os e-mails forem inseridos com suas chaves em ordem crescente, a árvore formará uma lista encadeada, e, dessa forma, essas funções terão custo $O(n)$, representado seu pior caso.

Para resolver esse problema do desbalanceamento, seria possível implementar árvores pseudo-balanceadas, como a árvore AVL e a árvore B. Mas nesses casos, após operações de inserção e remoção é preciso realizar rebalanceamentos na árvore, o que também gera alguns custos.

Por outro lado, a tabela hash apresenta uma vantagem muito interessante que é sua simplicidade de implementação e interpretação. Além disso, apresenta alta eficiência no custo de pesquisa, que é $O(1)$ para o caso médio, dependendo apenas da consulta na árvore binária. Porém, esse mecanismo de pesquisa também apresenta algumas desvantagens, como o custo para recuperar os registros na ordem crescente das chaves é alto, sendo necessário ordenar o arquivo. Além disso, quando o número de entradas é muito alto o número de registros da tabela hash é pequeno, seu pior caso se aproxima de $O(n)$, tendo custo linear, semelhante à uma pesquisa sequencial.

3.2 Análise de complexidade de tempo

Para entender melhor a complexidade do programa, primeiramente realizaremos a análise de complexidade de cada função de forma individual, para, em seguida, analisar a complexidade do programa como um todo. Além disso, para uma melhor organização, analisaremos uma classe implementada de cada vez.

3.2.1 Email

A classe *Email*, como já foi supracitado, é uma classe muito simples, que apresenta apenas alguns atributos e funções. Nesse sentido, essas funções lidam apenas com operações constantes, como os construtores que atribuem valores, ou as funções do tipo *get* que retornam atributos da classe. Além disso também existe a função *print*, que foi utilizada exclusivamente para a depuração de código, mas também só realiza operações constantes. Por fim, todas as funções da classe *Email*, apresentam complexidade assintótica $O(1)$.

3.2.2 Node

A classe *Node* representa uma célula da árvore binária implementada. Dessa forma, esta função também apresenta um comportamento muito simples, assim como suas funções. Nesse sentido, a classe apresenta apenas duas funções, sendo elas o construtor padrão e o construtor que recebe um e-mail como parâmetro, ambas realizam apenas operações de custo constante, apresentando complexidade assintótica $O(1)$.

3.2.3 BinaryTree

A classe BinaryTree representa uma árvore binária de n elementos, que no caso são os e-mails enviados e altura h. Dessa forma, utilizaremos a quantidade de e-mails n para realizar a análise de complexidade, juntamente com a altura h da árvore. A seguir, analisaremos a complexidade de cada função da classe de forma separada.

- *BinaryTree:* Construtor da classe, inicializa o atributo com operações constantes com tempo $O(1)$.
- *~BinaryTree:* Chama a função clean da classe com tempo $O(n)$.
- *clean:* Chama a função *cleanRecursive*, com tempo $O(n)$. Além disso, realiza operações constantes com tempo $O(1)$. Dessa forma, a complexidade assintótica da função será $O(1) + O(n) = O(max(1, n)) = O(n)$.
- *cleanRecursive:* Realiza operações constantes com tempo $O(1)$. Além disso, acessa todos os nós da árvore de forma recursiva, com tempo $O(n)$. Dessa forma, a complexidade assintótica da função será $O(1) + O(n) = O(max(1, n)) = O(n)$.

- *search:* Chama a função *searchRecursive*, com tempo médio $O(log\, n)$.

- *searchRecursive:* Realiza operações constantes com tempo $O(1)$. Além disso, o tempo de execução dos algoritmos para árvores binárias de pesquisa dependem muito do formato das árvores. Dessa forma, o pior caso dessa função ocorre quando a árvore binária está completamente desbalanceada, se tornando praticamente uma lista encadeada, e o e-mail pesquisado se encontra na folha da árvore ou não existe, dessa forma a pesquisa tem tempo $O(n)$. Já o melhor caso, ocorre quando o elemento pesquisado está na raiz da árvore, com tempo $O(1)$. Por fim, podemos tomar como base o caso médio da pesquisa, que é $O(h)$, em que h representa a altura da árvore, aproximadamente $O(\log n)$.

- *remove:* Chama a função *removeRecursive*, com tempo médio $O(\log n)$.

- *removeRecursive:* Realiza operações constantes com tempo $O(1)$. Além disso, o algoritmo de remoção é muito semelhante ao algoritmo de pesquisa, assim, o pior caso dessa função ocorre quando a altura h é igual à quantidade de elementos n, portanto tem complexidade $O(n)$. Já o caso médio é muito dependente do valor da altura da árvore e do seu balanceamento, dessa forma, o caso médio dessa função tem tempo, aproximadamente, $O(h) = O(\log n)$.

- *insert:* Chama a função *insertRecursive*, com tempo médio $O(\log n)$.

- *insertRecursive:* Realiza operações constantes com tempo $O(1)$. Além disso, o algoritmo de inserção é muito semelhante aos algoritmos de pesquisa e remoção, assim, o pior caso dessa função ocorre quando a altura h é igual à quantidade de elementos n, sendo uma árvore muito desbalanceada, portanto tem complexidade $O(n)$. Já o melhor caso ocorre quando a árvore está vazia, com tempo $O(1)$. Por fim, o caso médio é muito dependente do valor da altura da árvore e de seu balanceamento, dessa forma, o caso médio dessa função tem tempo, aproximadamente, $O(h) = O(\log n)$.

- *printInOrder:* Chama a função *printInOrderRecursive,* com tempo $O(n)$.

- *printInOrder:* Realiza operações constantes com tempo $O(1)$. Além disso, itera recursivamente todos os elementos da árvore, com tempo $O(n)$. Dessa forma, a complexidade assintótica da função é $O(1) + O(n) = O(\max(1, n)) = O(n)$.]

3.2.4 Hash_BT

A classe *Hash_BT* representa uma tabela Hash de árvores binárias de tamanho m e que possui registrado n e-mails no total, contabilizando os e-mails de todos os usuários. Nesse sentido, a análise de complexidade torna-se um pouco mais complexa, portanto, para generalizar e simplificar a análise, vamos supor que cada árvore binária da tabela possui, em média, $\frac{n}{m}$ elementos, o que tornará essa seção mais objetiva e consistente.

- *Hash_BT:* Realiza operações constantes de atribuição de valores, com tempo $O(1)$.
- *hash:* Realiza operações constantes, com tempo $O(1)$.
- *search*: Realiza operações constantes com tempo $O(1)$. Além disso, chama a função hash, tempo $O(1)$ e a função *search* de uma árvore binária de $\frac{n}{m}$ elementos, com tempo $O\left(\log \frac{n}{m}\right)$. Dessa forma, a complexidade assintótica é $O(1) + O(1) + O\left(\log \frac{n}{m}\right) = O\left(\max\left(1, \log \frac{n}{m}\right)\right) = O\left(\log \frac{n}{m}\right)$.

- *insert*: Realiza operações constantes com tempo $O(1)$. Além disso, chama a função hash, tempo $O(1)$ e a função *search* da classe com tempo $O\left(\log\frac{n}{m}\right)$. Por fim, chama a função *insert* de uma árvore binária de $\frac{n}{m}$ elementos, com tempo $O\left(\log\frac{n}{m}\right)$. Dessa forma, a complexidade assintótica é $O(1) + O\left(\log\frac{n}{m}\right) + O\left(\log\frac{n}{m}\right) = O\left(\max\left(1, \log\frac{n}{m}\right)\right) = O\left(\log\frac{n}{m}\right)$.
- *remove:* Realiza operações constantes com tempo $O(1)$. Além disso, chama a função hash, tempo $O(1)$ e a função *remove* de uma árvore binária de $\frac{n}{m}$ elementos, com tempo $O\left(\log\frac{n}{m}\right)$. Dessa forma, a complexidade assintótica é $O(1) + O(1) + O\left(\log\frac{n}{m}\right) = O\left(\max\left(1, \log\frac{n}{m}\right)\right) = O\left(\log\frac{n}{m}\right)$.
- *print*: Realiza operações constantes com tempo $O(1)$. Além disso, itera todos os elementos da tabela Hash, com tempo $O(n)$. Dessa forma, a complexidade é $O(n)$.

3.2.5 Main

O arquivo Main lida com apenas uma função que orquestra outras três, essa função é chamada *readInputFile*. Essa função lê todo o arquivo de entrada, executa as operações requisitadas e imprime o arquivo de saída. As três funções orquestradas são as seguintes:

- *sendEmail:* Realiza operações constantes com tempo $O(1)$. Além disso, realiza uma iteração por todas as p palavras da mensagem passada no arquivo de entrada, com complexidade $O(p)$. Além disso, chama a função *insert* da classe Hash, com tempo $O\left(\log\frac{n}{m}\right)$. Dessa forma, como cada mensagem apresenta no máximo 200 palavras, podemos considerar m como 200, assim a complexidade assintótica da função é $O(200) + O\left(\log\frac{n}{m}\right) = O\left(\max\left(1, log\frac{n}{m}\right)\right) = O\left(\log\frac{n}{m}\right)$.
- *consultEmail:* Realiza operações constantes com tempo $O(1)$. Além disso, chama a função *search* da classe Hash, com tempo $O\left(\log\frac{n}{m}\right)$.
- *removeEmail:* Realiza operações constantes com tempo $O(1)$. Além disso, chama a função *remove* da classe Hash, com tempo $O\left(\log\frac{n}{m}\right)$.

Dessa forma, a complexidade assintótica de todo o programa, depende de três variáveis, todas dependentes do arquivo de entrada. São elas, o número m de árvores binárias na tabela Hash, o número n de e-mails enviados e armazenados no sistema, e o número x de operações requisitadas, dessa forma, o programa como um todo apresenta tempo médio $O\left(x\log\frac{n}{m}\right)$.

A complexidade encontrada, descarta uma série de complicações de casos de entrada, para tornar a análise mais simples, já que se considerássemos todas as possibilidades de entrada, a análise de complexidade se tornaria muito mais complexa. Um exemplo, é a inserção, que muda o valor de n e-mails no sistema, ou seja, a primeira inserção apresenta complexidade $O\left(\log\frac{1}{m}\right)$, enquanto a última apresenta complexidade $O\left(\log\frac{n}{m}\right)$. Um caso semelhante é a remoção, que diminui o número n de e-mails, sendo assim, a primeira remoção apresenta complexidade $O\left(\log\frac{n}{m}\right)$, enquanto a última tem tempo $O(1)$. Por fim, concluímos que o tempo assintótico do programa é muito inconstante, e depende muito da natureza do arquivo de entrada.

3.3 Análise de complexidade de espaço

Para realizar a análise de complexidade de espaço, precisamos analisar a função *Main* como um todo, já que ela lida com toda a execução do programa. Nesse sentido, a função *Main* apresenta três principais funções, *sendEmail*, *consultEmail* e *removeEmail*. Além disso, é preciso lembrar que a quantidade de e-mails utilizada durante a execução do programa depende exclusivamente do arquivo de entrada.

Primeiramente, temos que a tabela hash armazena m árvores binárias armazenadas. Dessa forma, se tivermos lidando com n e-mails, cada árvore apresentará $\frac{n}{m}$ e-mails. Sabendo disso, podemos realiza a análise do restante do programa.

Em segundo lugar, a função *sendEmail* trata da inserção de um e-mail na tabela hash e na sua respectiva árvore binária. Dessa forma, é preciso percorrer, no máximo, $\frac{n}{m}$ elementos. Dessa forma, a complexidade de espaço dessa função pode ser definida como $O\left(\frac{n}{m}\right)$, o que faz sentido, já que a árvore binária apresenta $\frac{n}{m}$ elementos. Porém, devemos considerar que a cada vez que essa função for chamada, a quantidade de elementos aumentará, incrementando também o valor de n.

Em sequência, a função *removeEmail* trata da remoção de um e-mail da tabela hash e na sua respectiva árvore binária. Dessa forma, é preciso percorrer, no máximo, $\frac{n}{m}$ elementos. Dessa forma, a complexidade de espaço dessa função também pode ser definida como $O\left(\frac{n}{m}\right)$. Porém, devemos considerar que a cada vez que essa função for chamada, a quantidade de elementos diminuirá, decrementando também o valor de n.

Por fim, temos a função *consultEmail* trata da consulta de um e-mail da tabela hash na sua respectiva árvore binária. Dessa forma, é preciso percorrer, no máximo $\frac{n}{m}$ elementos. Dessa forma, a complexidade de espaço dessa função também pode ser definida como $O\left(\frac{n}{m}\right)$.

Em conclusão, podemos sumarizar todas as ordens de complexidade encontrada de cada um dos registros da tabela hash. Dessa forma, teremos a seguinte complexidade espacial do programa como um todo:

$$O\left(m * \frac{n}{m}\right)$$
$$= \boldsymbol{O(n)}$$

4. Estratégias de Robustez

Robustez é a capacidade de um sistema funcionar mesmo em condições anormais. Nesse sentido, as funções implementadas no programa criado apresentam verificações de entradas inadequadas e mal funcionamento do programa. Para tornar o código mais robusto e protegido, forma utilizadas funções da biblioteca *msgassert.h*, disponibilizada no *moodle* da matéria Estrutura de Dados.

Primeiramente, analisaremos a robustez das funções das duas estruturas de dados implementadas, árvore binária e tabela hash. Em seguida, poderemos observar a robustez da função *main* como um todo.

4.1 Email

A classe *Email* é uma classe muito simples. Dessa forma, a sua robustez também é simples, existindo apenas no construtor da classe. Nesse sentido, a função verifica se o identificador do e-mail e usuário recebidos como parâmetros são números inteiros positivos.

4.2 BinaryTree

A classe *BinaryTree* é um pouco mais complexa, dessa forma precisa de um tratamento de robustez mais detalhado. Nesse sentido, como as funções apresentam comportamentos semelhantes, elas também apresentam verificações de robustez semelhantes, em que na maioria delas é preciso verificar a consistência dos identificadores de e-mail e usuário recebidos como parâmetros. Assim, as funções *insert*, *search* e *remove* verificam se os identificadores recebidos como parâmetro são positivos, ou seja, maiores ou iguais a zero.

Além disso, duas funções que iteram todos os nós da árvore precisam verificar se a árvore não está vazia. Essas são as funções *printInOrder* e *clean*, que no início da execução verificam se o primeiro nó da árvore, sua raiz, aponta para um endereço nulo.

4.3 Hash

A classe *Hash* também necessita de algumas verificações, já que ela lida com a maior parte do programa. Primeiramente, seu construtor precisa verificar se o tamanho da tabela hash recebido como parâmetro é maior ou igual a zero, caso contrário a execução do programa é interrompida. Além disso, ainda no construtor, depois de realizar a alocação da tabela de árvores binárias é conferido se a alocação foi realizada de forma correta, verificando se o endereço da tabela não é nulo.

Por fim, assim como na classe BinaryTree, também é preciso verificar a consistência dos identificadores recebidos como parâmetro. Dessa forma, nas funções *search*, *insert* e *remove* é verificado se os identificadores de usuário e e-mail recebidos como parâmetros são maiores ou iguais a zero. Da mesma forma, a função *hash* também verifica se a chave recebida é positiva.

4.4 Main

A classe *Main* lida com a execução do programa como um todo, dessa forma, é preciso lidar desde a leitura dos argumentos da execução do programa até a leitura dos arquivos. Nesse sentido, primeiramente, durante a leitura dos argumentos, é verificado se o nome do arquivo passado como parâmetro não está vazio, caso contrário, o nome “entrada.txt” é atribuído ao nome do arquivo. Da mesma forma, um processo semelhante é executado para a verificação do arquivo de saída e para o arquivo de registro, em que é atribuído “saida.txt” e “/tmp/tp3log.out” respectivamente.

Partindo para a leitura do arquivo de entrada e execução do programa, primeiramente, é verificado se foi possível acessar os arquivos de entrada e saída, caso contrário, a execução do programa é interrompida. Por fim, a função *sendEmail* verifica se o número de palavras da mensagem de e-mail é menor ou igual a 200, caso contrário, a execução do programa é interrompida.

5. Análise Experimental

Esta seção apresenta os experimentos realizados em termos de desempenho computacional e localidade de referência. Os resultados desses experimentos foram gerados utilizando as bibliotecas disponibilizadas no *Moodle* da disciplina de estrutura de dados, *analisamem* e *memlog*. Por fim, essa seção também apresenta uma análise desses resultados, para um melhor entendimento sobre a utilização de memória durante a execução do programa.

5.1 Análise de Desempenho

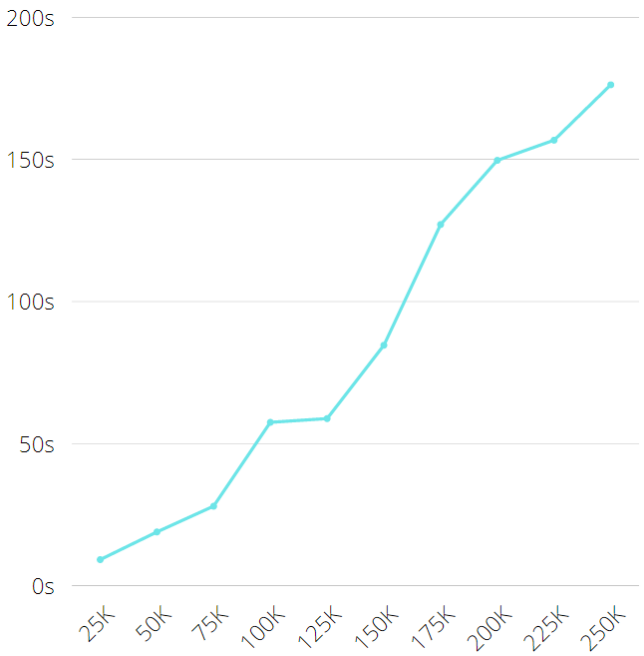
Primeiramente, realizaremos a análise de desempenho. Para gerar os grandes arquivos de entrada, foi desenvolvido dois programas simples na linguagem Python, que é possível definir o número de usuários que utilizarão o sistema de e-mail e a quantidade de mensagens por usuário. Além disso, o programa define o tamanho da tabela hash como 1.3 vezes o valor do número de usuários. Dos dois programas desenvolvidos, um gera uma entrada regular para o programa, enquanto o outro gera uma entrada mais aleatória, que será melhor explicado a seguir.

Para realizar essa análise, consideraremos quatro dimensões do programa: número de usuários, número de mensagens, tamanho das mensagens e distribuição de frequência de operações. Para isso, realizaremos a análise de cada uma dessas dimensões de forma separada nas seções seguintes.

5.1.1 Número de usuários

Primeiramente, para realizar a análise de desempenho variando apenas o número de usuários, utilizaremos o programa que gera uma entrada padrão. Nesse sentido, essa entrada insere todos os e-mails, com seus identificadores em ordem aleatória, em que cada usuário receberá 50 e-mails. Dessa forma, fixaremos o número de e-mails em 50, o tamanho das mensagens será 15 palavras de, aproximadamente, 5 caracteres cada, totalizando, aproximadamente, 75 caracteres. Além disso, depois de inserir todos os e-mails, a entrada pesquisa metade dos e-mails e, por último, remove metade, de forma completamente aleatória.

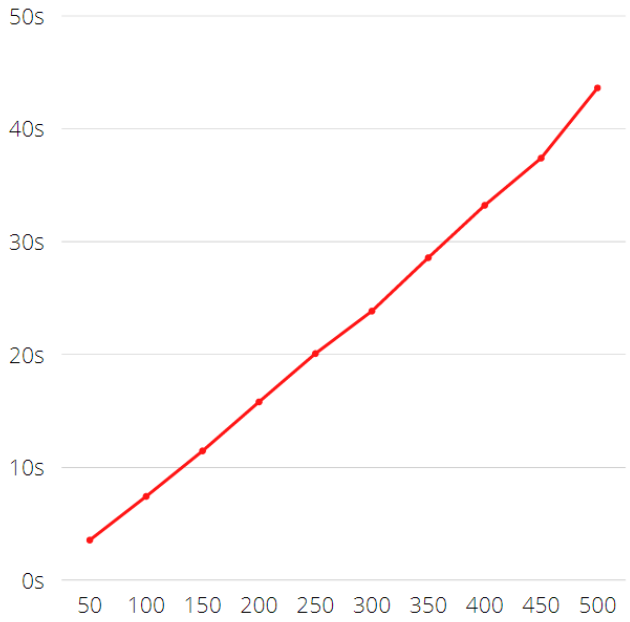
Assim, realizamos a execução do programa com o número de usuários variando de 25 mil usuários até 250 mil, variando de 25 em 25 mil, gerando 10 casos de entrada. Ao executar o programa com as funções da biblioteca *memlog*, obtemos os seguintes tempos de execução, representados no gráfico ilustrativo a seguir. Como a quantidade de usuários não é um fator significante na complexidade assintótica do programa, além da entrada ser gerada de forma completamente aleatória, faz com que a evolução dos tempos de execução apresente certa inconstância, mas o crescimento é aproximadamente linear.



5.1.2 Número de mensagens

Para realizar a análise de desempenho variando apenas o número de mensagens, também utilizaremos o programa que gera uma entrada padronizada. Dessa forma, fixaremos o número de usuários em 10 mil, o tamanho das mensagens será 15 palavras de, aproximadamente, 5 caracteres cada, totalizando, aproximadamente, 75 caracteres. Além disso, depois de inserir todos os e-mails, a entrada pesquisa metade dos e-mails e, por último, remove metade, de forma completamente aleatória.

Assim, realizaremos a execução do programa com o número de mensagens variando de 50 até 500, variando de 50 em 50, totalizando 10 entradas. Ao executar o programa com as funções da biblioteca *memlog*, obtemos os seguintes tempos de execução, representados no gráfico ilustrativo a seguir. O gráfico apresenta crescimento perceptivelmente linear, com apenas algumas inconstâncias, o que condiz com a complexidade assintótica encontrada na seção 3, $O\left(x\frac{n}{m}\right)$, em que x é o número de operações, m o tamanho da tabela hash e n é o número de e-mails. Como o número de operações e o tamanho da tabela hash permanecem os mesmos durante os testes, o crescimento do tempo de execução torna-se linear.

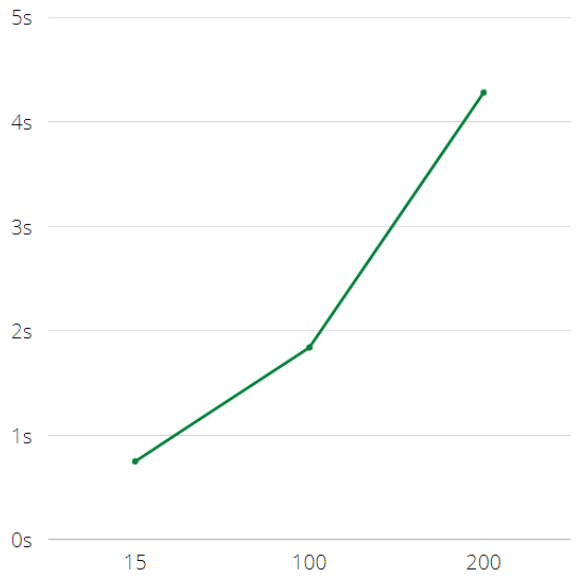


5.1.3 Tamanho das mensagens

Para realizar a análise de desempenho variando apenas o tamanho das mensagens, também utilizaremos o programa que gera uma entrada padronizada. Dessa forma, fixaremos o número de usuários em 10 mil, e o número de mensagens em 100. Além disso, depois de inserir todos os e-mails, a entrada pesquisa metade dos e-mails e, por último, remove metade, de forma completamente aleatória.

Assim, realizaremos a execução do programa com três tamanhos de mensagens diferentes, pequeno, médio e grande. A mensagem pequena contém apenas 15 palavras, totalizando cerca de 75 caracteres. Já as mensagens médias apresentam 100 palavras, cada uma com 20 caracteres, totalizando 2000. Por fim, as mensagens grandes possuem 200 palavras, cada uma com 40 caracteres, totalizando 8000. Ao executar o programa com as funções da biblioteca *memlog*, obtemos os seguintes

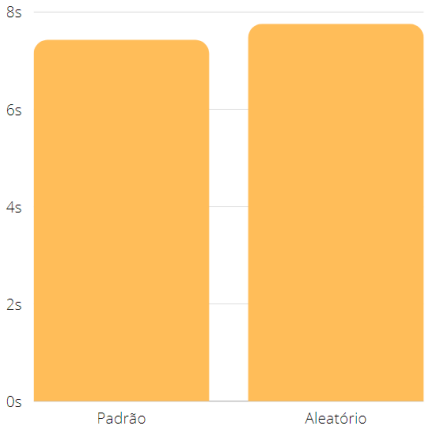
tempos de execução, representados no gráfico ilustrativo a seguir, em que varia o tempo em relação ao número de palavras da mensagem.



5.1.4 Frequência de operações

Durante as análises anteriores sempre foi utilizado um arquivo de entrada padrão, ou seja, primeiro eram inseridos os n e-mails de cada usuário, ou seja, são inseridos todos os e-mails de cada usuário de forma separada e, só ao final, é realizada a busca e a remoção de $\frac{n}{2}$ e-mails. Outra entrada possível, é uma entrada mais aleatória, que durante o processo das inserções é realizado buscas e remoções de forma aleatória, em qualquer usuário que já teve seus e-mails enviados. Dessa forma, podemos comparar o tempo de execução de cada uma dessas entradas.

Assim, realizaremos duas execuções do programa, ambas com 10000 usuários, 100 mensagens por usuário e apenas mensagens pequenas. Ao executar o programa com as funções da biblioteca *memlog*, obtemos tempos de execução semelhantes, mas não iguais, como podemos ver no gráfico a direita. A entrada padronizada apresentou tempo de execução de 7.42 segundos, enquanto a entrada aleatória tem tempo 7.75 segundos.

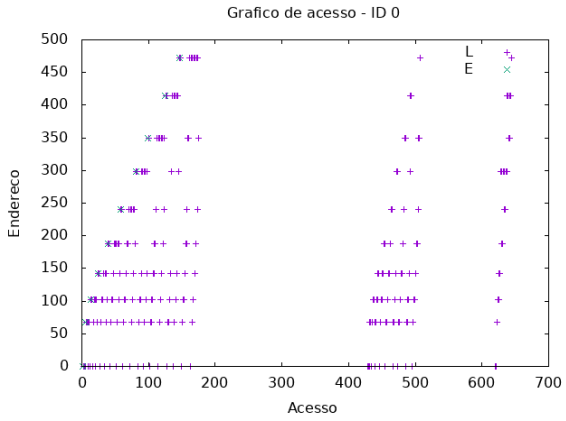


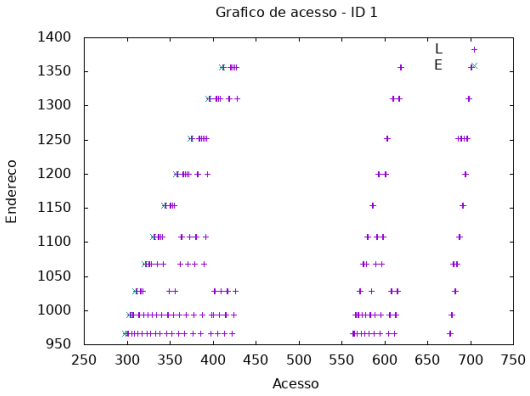
Dessa forma, concluímos que a entrada aleatória apresenta um resultado de desempenho inferior que ao resultado da entrada padrão. Isso ocorre, principalmente, devido à distância de pilha, que na entrada padrão, a cada 100 operações lida com um usuário diferente, ou seja, a cada 100 operações estão relacionadas a apenas uma árvore binária, o que faz a distância de pilha ser muito pequena. Isso não ocorre na entrada aleatória, em que cada operação pode incluir qualquer usuário já inserido na tabela hash, o que deixa a distância de pilha maior, e, consequentemente, o tempo de execução maior também.

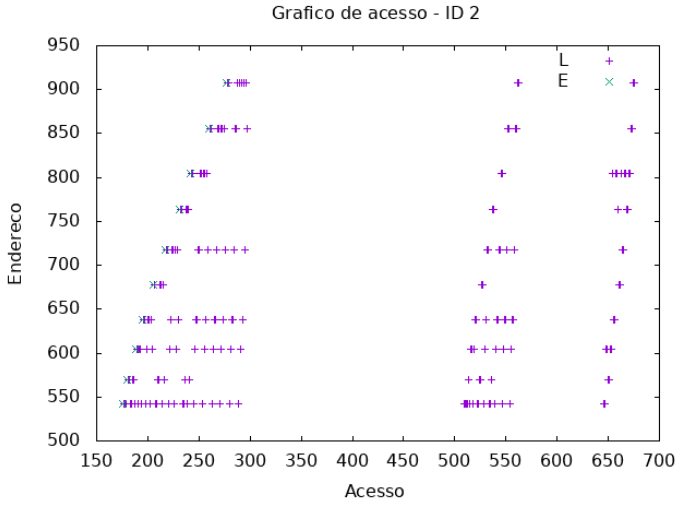
5.2 Análise de Localidade de Referência

Em seguida, para realizar a análise de localidade de referência, analisaremos especificamente a posição de cada e-mail durante a execução do programa na memória. Nesse sentido, o arquivo de entrada utilizado para realizar essa análise contém uma tabela hash de tamanho três, e também apenas três usuários. Além disso, o arquivo de entrada realiza noventa operações, sendo elas dez inserções, consultas e remoções para cada usuário, totalizando trinta operações por usuário. Nesse sentido, primeiro são realizadas as trinta inserções, depois as trinta consultas, e, por fim, o restante das operações de remoção.

Os gráficos a seguir, foram gerados após a execução do programa e o armazenamento de seus registros de memória. Nesses gráficos, o 'ID' representa o identificador de cada um usuário, ou seja, o gráfico de acesso de 'ID 0' apresenta as consultas de e-mail específicas do usuário que possui o identificador igual a zero. Primeiramente, analisaremos os gráficos de acesso, que apresentam o endereço de cada e-mail em relação ao número de seu acesso.





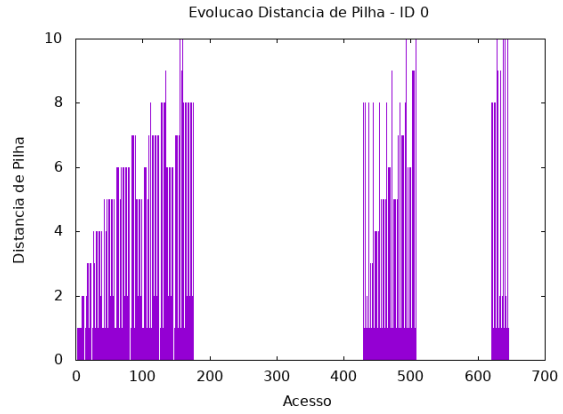


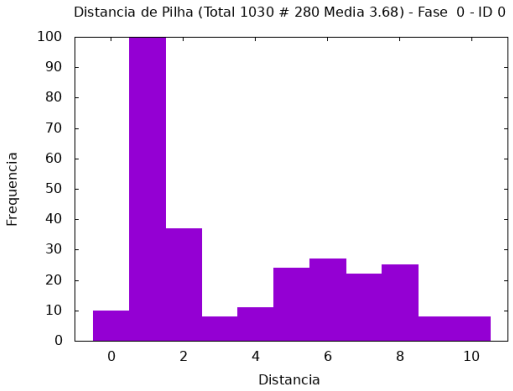
É perceptível que os três gráficos apresentam um comportamento muito semelhante. Isso ocorre porque cada gráfico representa as operações realizadas em relação a cada usuário, como os três usuários realizam as mesmas operações com a mesma quantidade de e-mails, os gráficos ficam muito parecidos.

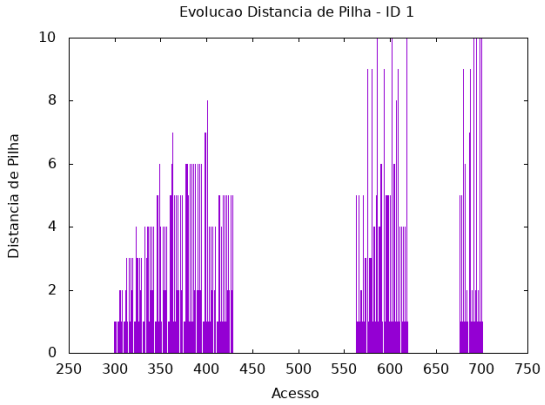
Em relação ao formato do gráfico, percebemos que cada conjunto de marcações representa um conjunto de operações diferente, em que o primeiro conjunto são as 10

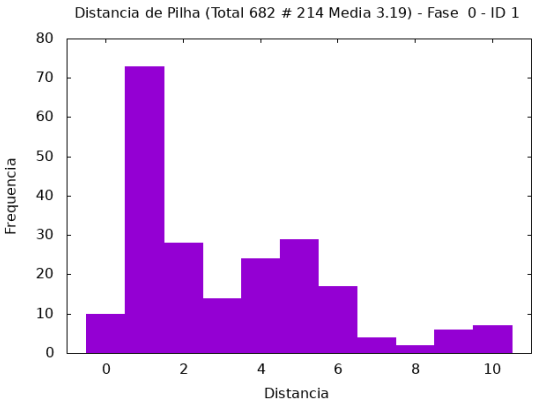
operações de soma, o seguinte são as 10 de consulta e o último são as remoções. Percebemos que, no início de toda operação o primeiro endereço é acessado, que representa a raiz da árvore binária. Assim, quando as funções recursivas caminham na árvore, acessam endereços distintos, o que deixa o gráfico mais esparso em endereços maiores.

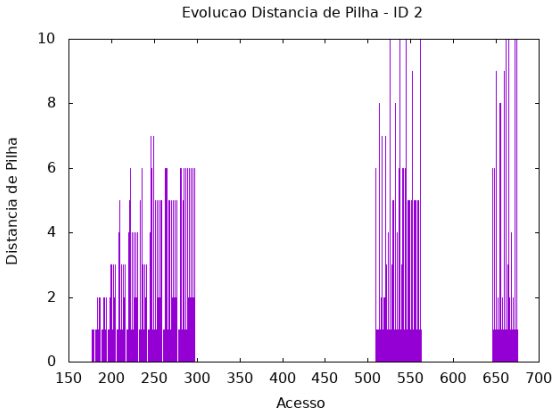
Em segundo lugar podemos analisar os gráficos de distância de pilha e sua evolução.

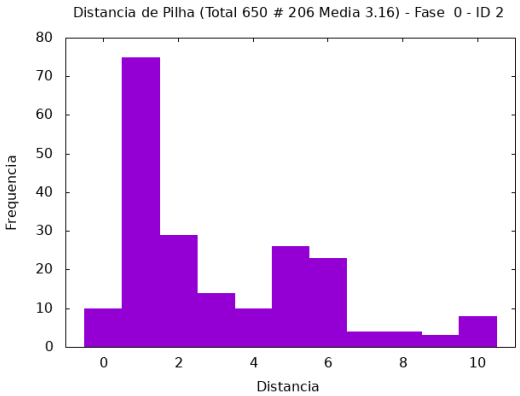












Assim como os gráficos de acesso, os gráficos de distância e evolução de pilha dos três usuários são muito semelhantes, já que realizam as mesmas operações. Porém, também possuem pequenas diferenças, que ocorrem devido à organização da árvore, que depende muito de como foi realizada sua inserção.

Além disso, é perceptível o aumento da distância de pilha com o aumento dos acessos. Isso ocorre por causa da inserção de novos e-mails nas árvores, o que também aumenta a distância de pilha entre os registros. Por fim, percebemos que a distância de pilha mais frequente tem valor 1, isso ocorre por conta da raiz, e nós mais superficiais da árvore binária, que são frequentemente acessados durante as operações.

Em conclusão, a utilização de árvores binárias valoriza muito a distância de pilha da memória, já que toda operação precisa passar pela raiz da árvore. Além disso, essa distância de pilha poderia ser melhorada caso a árvore implementada tivesse pseudo-balanceamento, como uma árvore AVL ou uma árvore B.

6. Conclusões

Este trabalho lidou com o problema da implementação de um simulador de servidor de e-mails, com gerenciamento adequado da memória do sistema e otimização da pesquisa por usuários e mensagens. Nesse sentido, o sistema apresenta suporte à entrega, consulta e remoção de mensagens para usuários, utilizando estruturas de dados e algoritmos de pesquisa aprendidos durante o curso de Estrutura de Dados. Nesse sentido, a abordagem utilizada foi a criação de um programa na linguagem C++, que utiliza duas principais estruturas de dados, que juntas formam uma tabela hash de árvores binárias, além de algoritmos de pesquisa estudados em sala de aula, como a pesquisa binária em árvores.

Com a solução adotada, pode-se verificar que é possível lidar com sistemas com um número arbitrariamente grande de e-mails, mensagens e usuários, como foi exemplificado durante os testes de performance na seção 5. Essa solução permitiu realizar uma análise sucinta de como a execução do programa ocorre em relação à memória do computador, em relação tanto ao tempo quanto ao espaço, observando os gráficos e resultados de desempenho gerados pela execução do Makefile juntamente com as bibliotecas *analisamem* e *gprof*.

Por meio da resolução desse trabalho, foi possível praticar os conceitos relacionados às estruturas de dados trabalhados durante o curso, além de um entendimento mais profundo sobre o funcionamento dos algoritmos de pesquisa binária em árvores de forma recursiva, em que a utilização da recursividade foi um aspecto muito desafiador durante o desenvolvimento do programa. Em conclusão, todas essas desafiadoras implementações auxiliaram no entendimento aprofundado da tabela hash, árvores binárias e pesquisa binária.

7. Bibliografia

Slides virtuais e códigos da disciplina de estruturas de dados disponibilizados via moodle.

Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

8. Instruções para compilação e execução

- Primeiramente, utilizando um terminal, acesse o diretório TP.
- Ao digitar o comando *make all*, o comando *build* do arquivo Makefile é executado no terminal, e, dessa forma, os arquivos objetos serão gerados na pasta *obj*, enquanto o executável do programa será criado com nome 'tp3.exe' na pasta *bin*.
- Além disso, outros comandos também podem ser executados:

- ***perf:*** Lê todos os arquivos de entrada utilizados durante a seção 5, na análise de desempenho do programa. São arquivos muito grandes, dessa forma, não é possível enviá-los no sistema de entrega do *Moodle*. Esses arquivos são retirados da pasta *assets*. Além disso, os resultados de desempenho são armazenados, retornando o tempo de execução de cada entrada. Os resultados de desempenho são armazenados na pasta *tmp*, para ver esses resultados, basta acessar a pasta *tmp* no terminal e escrever comandos no seguinte formato: 'more perf_user_200k.out'.
    - ***mem:*** Executa o arquivo "entrada.txt" localizado na raiz do diretório, de preferência um arquivo com poucos usuários e mensagens, armazenando no arquivo de saída "saida.txt". Além disso, os acessos à memória são registrados e podem ser utilizados para plotagem de gráficos utilizando a biblioteca disponibilizada *analisamem*, juntamente com o *gnuplot*.
    - ***run:*** Executa o arquivo 'bin/tp3.exe', dessa forma, as operações contidas no arquivo "entrada.txt", que deve estar no diretório raiz, serão executadas. Além disso, o registro de memória do programa será armazenado no arquivo '/tmp/runlog.out'

- Caso seja da preferência do usuário, é possível executar um arquivo específico de jogo de pôquer. Para isso, estando acessando o diretório raiz do programa, primeiramente é necessário executar o comando 'make build'. Em seguida, é preciso executar uma linha semelhante a seguinte, substituindo o nome do arquivo txt pelo arquivo desejado. Além disso, é preciso inserir o destino do arquivo de acesso de memória após o '-p'.
    - 'bin/tp3.exe -i entrada.txt -o saida.txt –p /tmp/runlog.out'
- Em caso de dúvidas, é possível executar o comando 'bin/tp3.exe -h' para visualizar a mensagem de uso do programa.