Trabalho Prático 3

Luiz Felipe de Sousa Faria – 2020027148

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil

lutilipe@ufmg.br

1. Introdução

O problema proposto foi implementar um sistema de servidor de *email*. Esse sistema é responsável pelo gerenciamento de emails, tanto da entrega, quanto consulta e exclusão. O objetivo da documentação é analisar o desempenho, a robustez, a localidade de referência e distância de pilha das principais estruturas, funções e métodos utilizados para o desenvolvimento do servidor, bem como refletir sobre algumas decisões e estruturas implementadas no projeto. Para implementar o servidor, foram utilizadas as estruturas de dados de árvore binária e tabela hash.

2. Método

2.1. Descrição

O programa principal se encontra no diretório ***TP***. Esse diretório é dividido em:

- ***include***: contém o arquivo *header* das classes *Email*, *EmailServer* (servidor de email) e *EmailBox* (caixa de email), do *memlog e* de funções de assert;
- ***src***: contém a implementação das funções do include. A função *main* está em *main.cpp*;
- ***obj***: contém os .o gerados pelo programa (vazio inicialmente)
- ***bin***: contém o binário do programa (vazio inicialmente)
- ***Makefile***: gera os arquivos binários e *.o* do programa e executa os testes de desempenho e análise de localidade.

O programa foi escrito em *C++* e compilado utilizando o *G++*.

O programa tem como entrada um arquivo que contém o tamanho da tabela hash e uma lista de comandos a serem executados. Essa lista contém o comando, que pode ser “ENTREGA”, “CONSULTA” ou “APAGA”, juntamente com alguns parâmetros, que dependem do comando que está sendo executado. No caso de “ENTREGA”, existem os parâmetros (na ordem): id do usuário que recebeu a mensagem, id da mensagem, número de palavras da mensagem e mensagem a ser

enviada. Para os comandos “CONSULTA” e “APAGA”, temos: id do usuário que recebeu a mensagem e id da mensagem.

Como saída, o programa gera um arquivo contendo a resposta do servidor para cada um dos comandos. Para o comando de “ENTREGA”, contém o id da mensagem enviada, o id do usuário que recebeu a mensagem e a localização na tabela hash da árvore que contém essa mensagem. Para “CONSULTA”, temos o id da mensagem e do usuário, mais a mensagem consultada. Caso não exista, será exibida uma mensagem de erro. Por fim, para o “APAGA”, também é exibido o id do usuário e da mensagem. E para caso não exista, exibe-se uma mensagem de erro.

As principais classes do programa são:

- ***Email***: responsável por guardar as informações do email (id, usuário e mensagem);
- ***EmailBox***: árvore binária que armazena os emails;
- ***EmailServer***: tabela hash responsável por armazenar vários *EmailBox* em cada posição.

2.2. Estrutura de dados utilizadas

No projeto, foram utilizadas duas estruturas de dados principais: árvores binárias e tabela hash.

As árvores binárias (figura 1) são estruturas que, assim como o nome diz, são semelhantes à uma árvore. Cada nó da árvore possui um valor e dois apontadores, um para a "esquerda" e outro para a “direita”. A disposição e localização de um valor na árvore depende dos outros valores já existentes na estrutura. De forma geral, ao inserir um valor, o mesmo é comparado com o valor “raíz” (se ele existir). Caso o valor a ser inserido for menor, repetimos o processo de comparação com o valor do nó à esquerda da raíz. Caso for maior, repetimos com o da direita. E assim sucessivamente até achar a posição para o valor ser inserido. O sistema de busca nessa estrutura também segue esse processo de comparação.

Já o processo de deleção ocorre de uma maneira diferente. Após encontrar o elemento a ser removido na árvore (pelo processo de busca), trocamos seu valor pelo maior valor localizado à esquerda desse nó e ajustamos as folhas (nós sem filhos), caso precise.

No caso do sistema do servidor de emails, a árvore binária foi utilizada para armazenar os emails de vários usuários, já que essa estrutura é bem eficiente em termos de busca, deleção e inserção (ver seção 3). A comparação no projeto foi feita em relação ao id do usuário a ser buscado e da chave da mensagem.

A outra estrutura de dados utilizada foi a tabela hash (figura 2). Essa estrutura de dados associa chaves a valores de pesquisa, através de uma função de *hashing*. Com isso, consegue localizar rapidamente um determinado valor a partir de entradas complexas (como *strings*).

No projeto, a função de hashing utilizada foi gerar um id a partir do resto da divisão entre o id do usuário e o tamanho da tabela. E cada posição da tabela é ocupada por uma árvore binária.

Figura 1: Árvore Binária

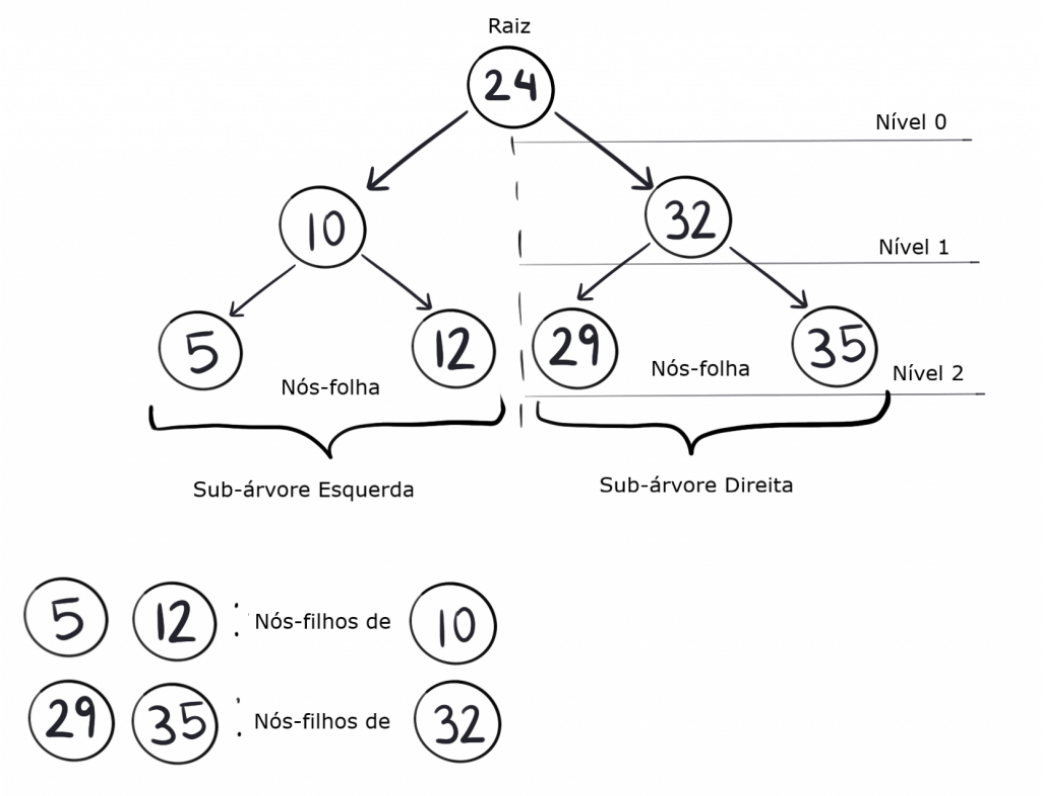
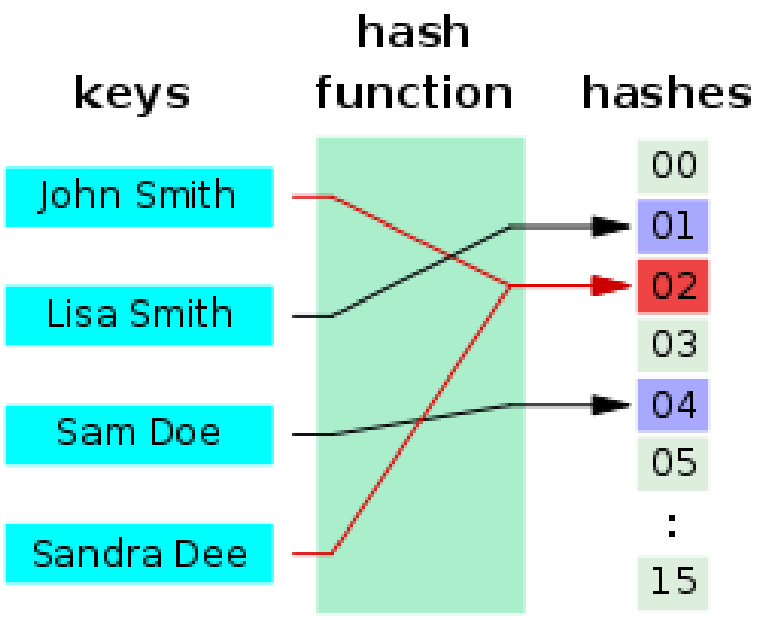


Figura 2: Tabela hash



3. Análise de Complexidade e Espaço

A complexidade de tempo e espaço do programa reflete àquelas referentes às estruturas de árvore binária e tabela hash, bem como o tamanho da mensagem de entrada. Considera-se **n** a quantidade de mensagens inseridas, **m** o tamanho das mensagens (para as análises, considera-se que toda mensagem tem tamanho **m**) e **c** a quantidade de comandos. Como a tabela hash possui uma função de hashing com complexidade **O(1)**, a complexidade de tempo do programa é igual a complexidade da árvore binária e leitura das palavras da mensagem. Nessa estrutura, o melhor caso se dá em **O(1)** (quando o valor está na raiz da árvore), o caso médio é **O(log n)** e o pior caso é **O(n)**. Esse pior caso ocorre quando a árvore está desbalanceada de uma forma que sua estrutura é semelhante a de uma lista encadeada. A leitura das mensagens e dos comandos será, respectivamente, **O(m)** e **O(c)**. Considerando um caso em que o comando executado será de envio da mensagem (único comando na qual a leitura da mensagem e realizada), a complexidade do programa se dá por:

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

O pior caso de busca na árvore poderia ser contornado caso utilizássemos uma árvore balanceada, como por exemplo a árvore AVL, que garante que os ramos da esquerda e da direita da árvore (para cada nó), não tenham uma diferença de altura muito grande. Com isso, o pior caso seria reduzido para **O(log n),** já que o número de operação para buscar um determinado elemento seria menor.

Para complexidade de espaço, o programa utiliza memória extra de tamanho **O(n)**.

4. Estratégias de Robustez

Para garantir a robustez do programa, foram implementadas algumas estratégias para impedir valores indevidos de variáveis ou ações que o programa não suporta.

Para o parâmetro que define o tamanho da tabela hash, é checado se ele não é nulo (tamanho 0), para evitar problemas na criação e alocação da memória para a tabela. Também é checado se o comando passado para o ṕrograma é válido (se está dentro da lista de comandos aceitos). Caso algum dessas condições não sejam cumpridas, o programa retorna um erro com uma mensagem explicando o motivo da interrupção da execução.

5. Análise Experimental

Para a análise experimental, foram realizados alguns experimentos a fim de medir a distância de pilha do programa, bem como essa distância de pilha varia com o tempo; calcular o tempo de execução das principais funções. Para tal, variou-se os seguintes parâmetros de entrada:

- Número de usuários
- Número de mensagens
- Tamanho das mensagens
- Distribuição de frequência de operações

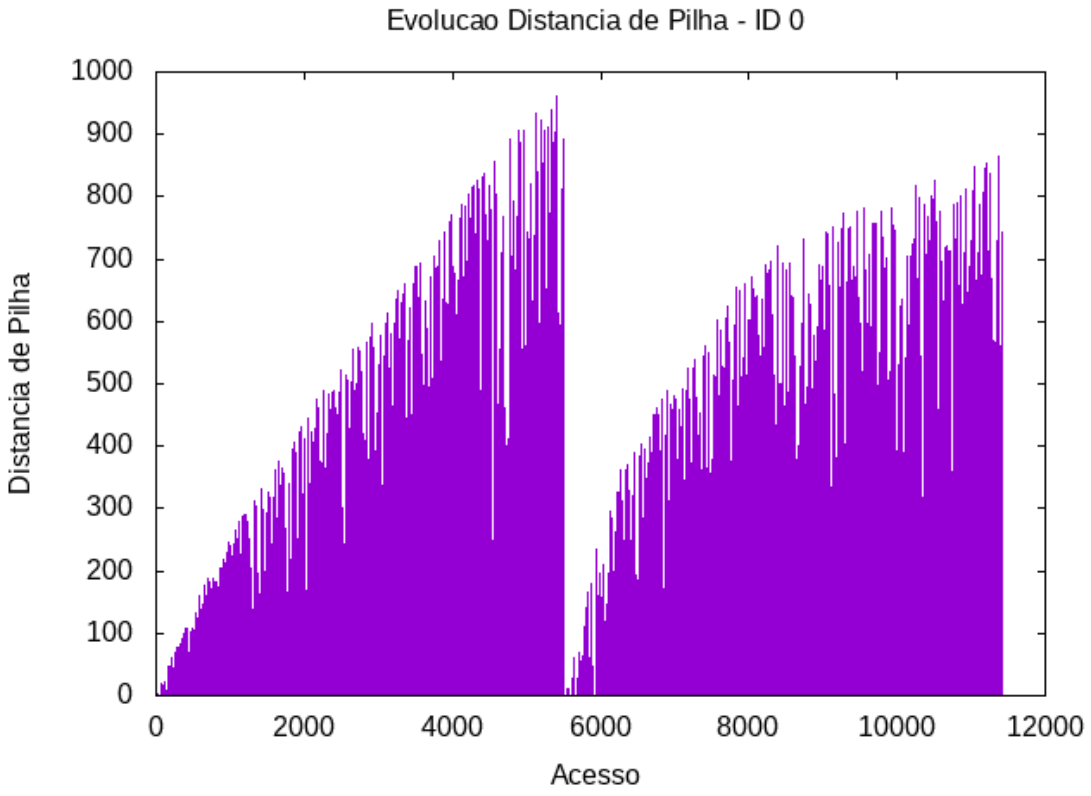
Tais parâmetros e seus respectivos valores são descritos em cada experimento realizado. O tamanho da tabela hash ficou fixo em 100 em todos os experimentos.

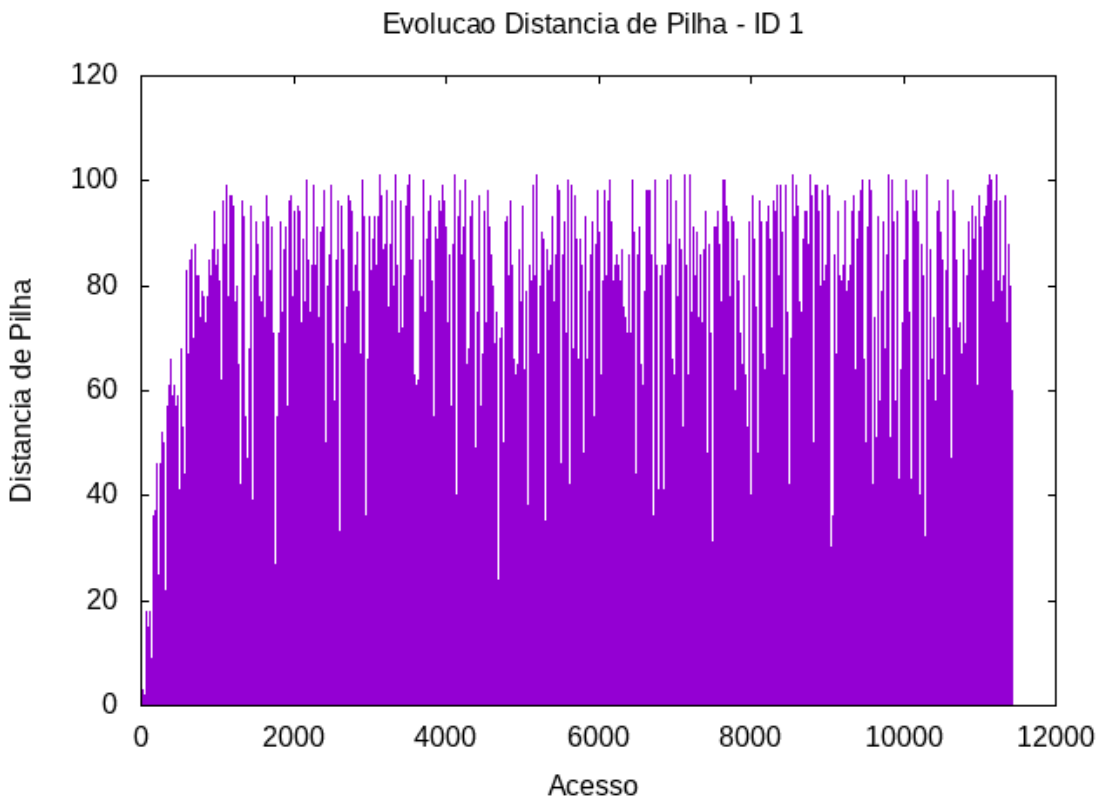
5.1. Localidade de referência e acesso à memória

Para o teste de localidade de referência e acesso à memória, foi considerada uma entrada de 3000 comandos, na ordem: Inserção > Consulta > Deleção. Cada um desses comandos foram executados 1000 vezes. O número de usuários foi aleatório, e o tamanho das mensagens foi fixo de 100. O tamanho da tabela hash também foi 200.

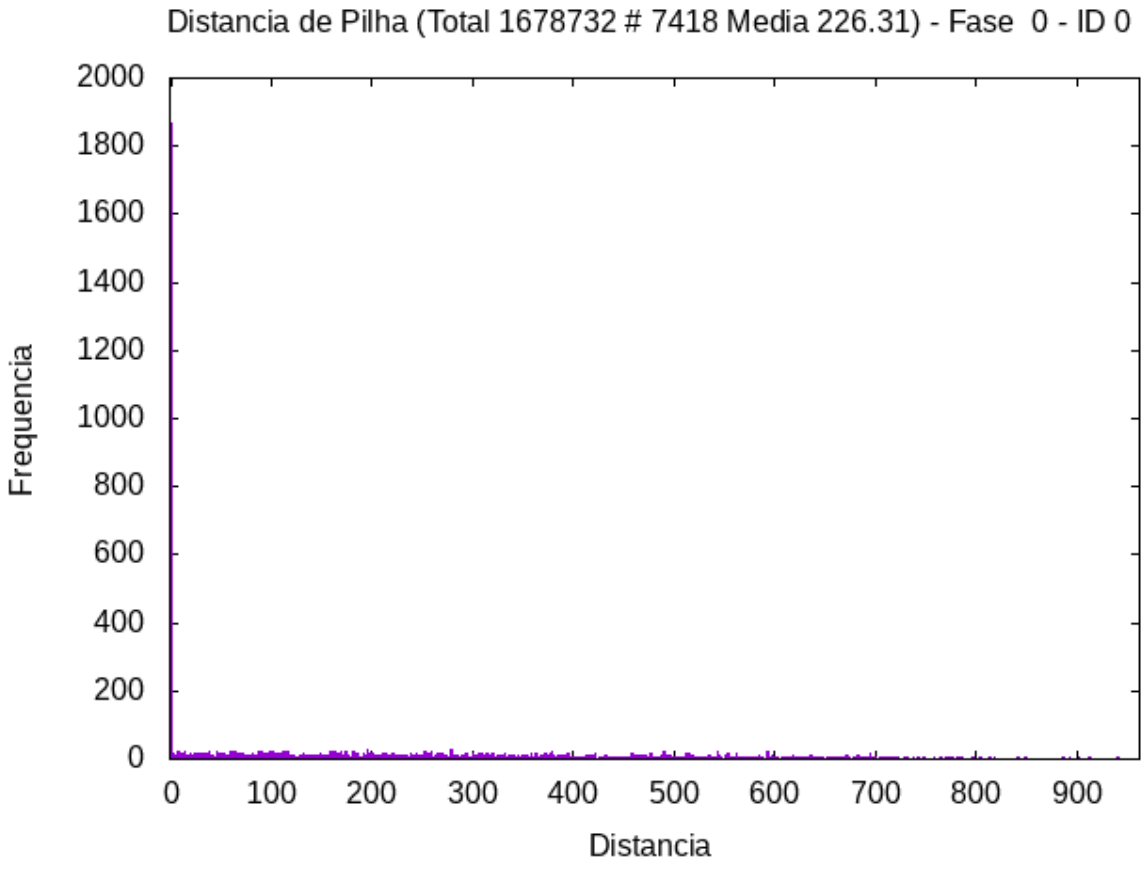
Os gráficos e mapas com ID 0 referem-se aos acessos na árvore binária, enquanto os com ID 1 referem-se aos acessos na tabela hash.

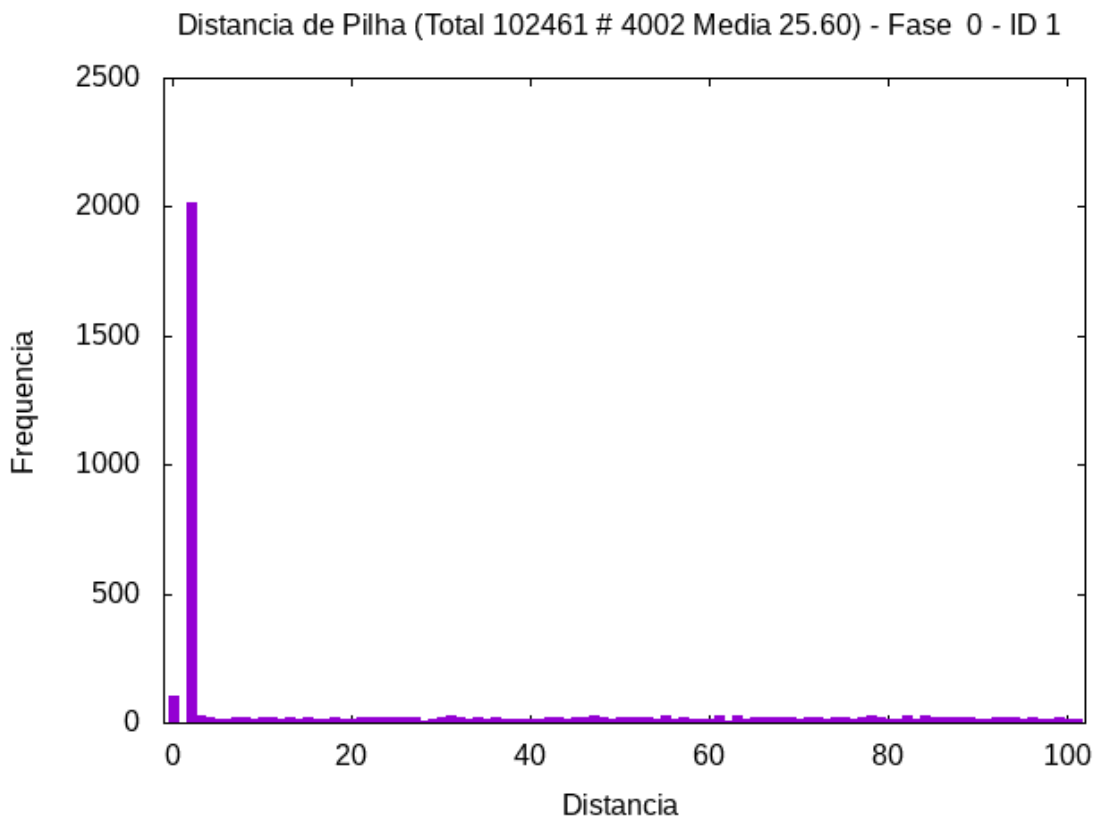
- **Distâncias de pilha**



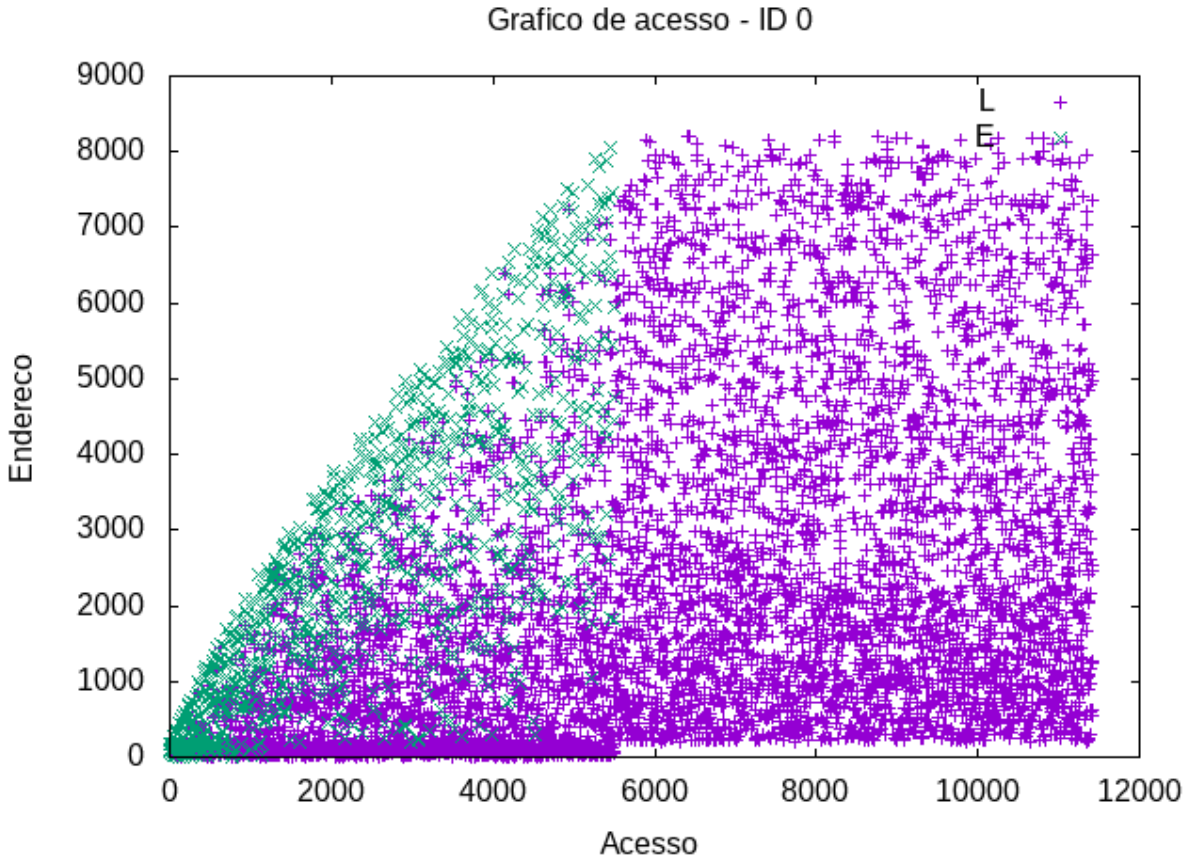


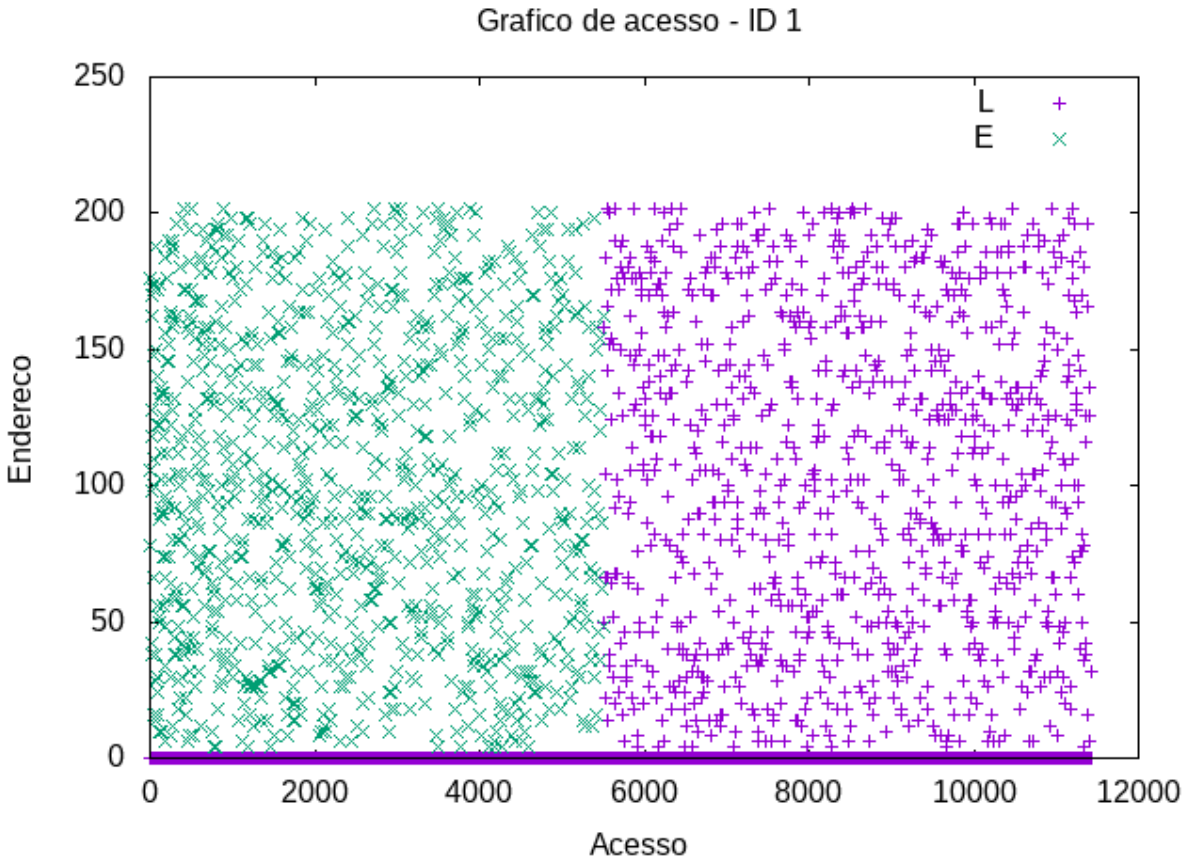
- **Histogramas**





- **Mapa de acesso**





Percebe-se pelo mapa e pela evolução da distância de pilha da árvore binárias que, pelo fato da árvore não estar balanceada, são realizados muitos acessos para consulta e inserção na árvore. Além disso, o próprio mapa revela que a árvore, em certos momentos, possui uma estrutura desbalanceada, se assemelhando em alguns ramos à uma lista encadeada. A evolução de pilha também separa os acessos de escrita e leitura, tendo em vista que esses comandos são realizados um após o outros, seguindo a ordem da entrada.

Para o histograma, percebe-se uma distância de pilha pequena em muitos casos. Em outros, percebe-se que essa distância aumenta, também reflexo do desbalanceamento da árvore.

Já para a tabela hash, percebe-se que o acesso se dá, em sua maioria de forma aleatória, mas dentro do range definido. Os acessos localizados em baixo, com endereço menor, formam uma linha. Eles representam o acesso à variável que armazena o tamanho da tabela. O histograma revela acessos de distância de pilha baixa, mas em grande quantidade

5.2. Desempenho computacional

1)

- 10000 comandos aleatórios
- Tamanho das palavras: 100
- Número de usuários: aleatório
- Número de mensagens: aleatório

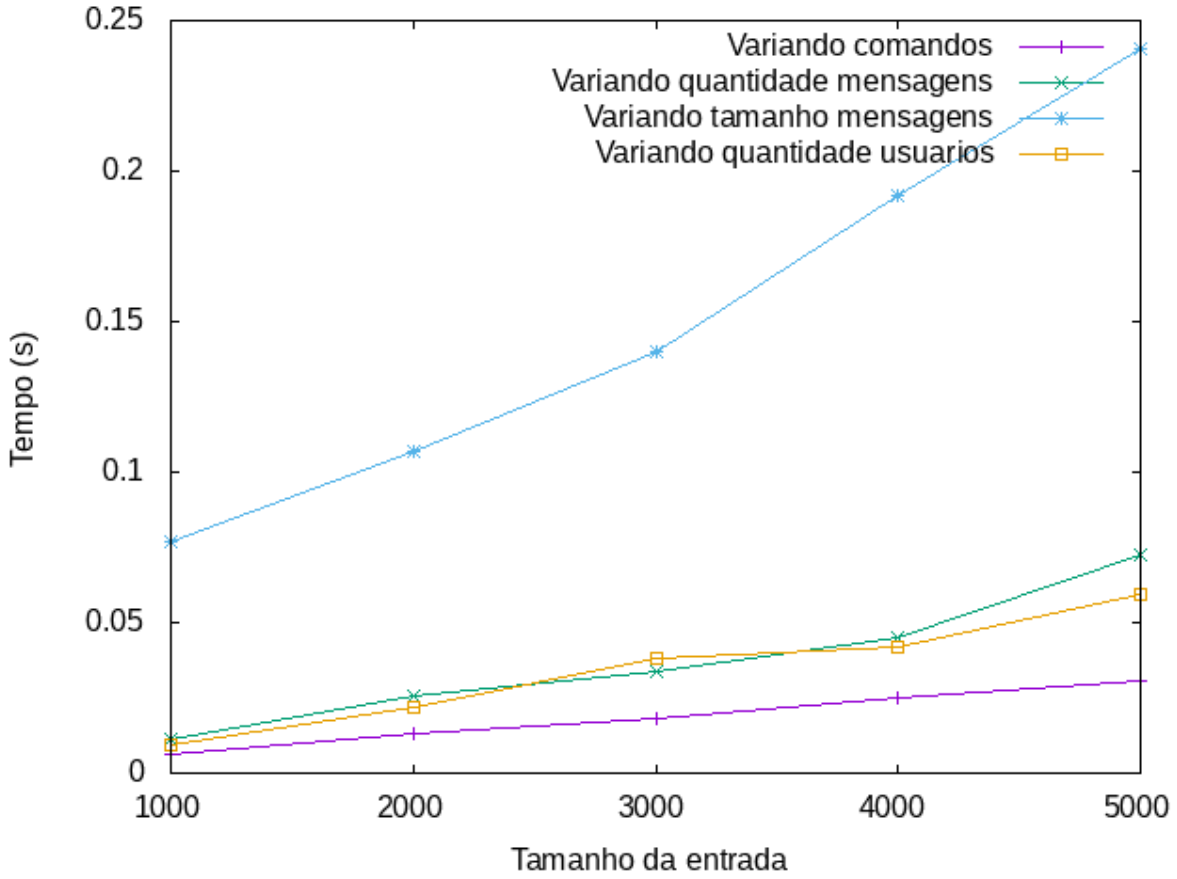
% time	cumulative seconds	self seconds	calls	self us/calls	total us/calls	name
50.03	0.02	0.02	1470522	10.21	10.21	Email::getKey()
33.35	0.03	0.01	33618	297.63	478.10	EmailBox::add()
0.00	0.03	0.00	33736	0.00	0.00	Email::getUserId()
0.00	0.03	0.00	33618	0.00	0.00	Email::operator=()
0.00	0.03	0.00	33518	0.00	160.60	EmailBox::addHelper()
0.00	0.03	0.00	33315	0.00	134.64	EmailBox::get()
0.00	0.03	0.00	33315	0.00	134.64	EmailBox::dfs()
0.00	0.03	0.00	33067	0.00	134.74	EmailBox::removeHelper()
0.00	0.03	0.00	33067	0.00	0.00	EmailBox::remove()
0.00	0.03	0.00	10000	0.00	0.00	EmailServer::handleCommands()
0.00	0.03	0.00	100	0.00	0.00	EmailBox::~EmailBox()

2)

- 10000 comandos. Apenas inserção
- Tamanho das palavras: 10000
- Número de usuários: aleatório
- Número de mensagens: 10000

Utilizou-se apenas inserção, pois nesse comando deve-se ler a mensagem do usuário e inserir na árvore de busca binária. Com isso, o tempo de execução será maior.

% time	cumulative seconds	self seconds	calls	self us/calls	total us/calls	name
100.06	0.15	0.15				EmailServer::handleDeliveryMessage()
0.00	0.15	0.00	131248	0.00	0.00	Email::getKey()
0.00	0.15	0.00	10000	0.00	0.00	Email::getUserId()
0.00	0.15	0.00	10000	0.00	0.00	Email::Email()
0.00	0.15	0.00	10000	0.00	0.00	Email::operator=()
0.00	0.15	0.00	10000	0.00	0.00	EmailBox::add()
0.00	0.15	0.00	9900	0.00	0.00	EmailBox::addHelper()
0.00	0.15	0.00	1453	0.00	0.00	EmailServer::handleCommands()
0.00	0.15	0.00	100	0.00	0.00	EmailBox::~EmailBox()



Para o gráfico, utilizou-se valores de entrada variando no intervalo 1000-5000. Cada linha representa um valor que variou desse intervalo (descrito na legenda). Os valores que não variaram seguiram o seguinte valor padrão

- Número de usuários: 100
- Número de mensagens: 100
- Tamanho da mensagem: 100
- Variação dos comandos: comandos aleatórios. A quantidade padrão de comandos executados foi: 1000.

Para o teste variando o número de mensagens e usuários, considerou-se apenas o comando de inserção (visto que ele faz tanto a busca da posição para inserir a mensagem, quanto a leitura da mensagem).

Nota-se que aumentando o tamanho da mensagem, o tempo de execução aumenta substancialmente. Isso se deve a função utilizada para ler as palavras do arquivo, que possui um custo alto. Para os outros valores, percebe-se que não há uma variação muito grande. Nota-se também que, ao variar os comandos ao invés de manter todos como inserção, o tempo de execução diminui, visto que terá menos registros nas árvores e um menor tempo de busca nelas.

6. Conclusão

O presente trabalho propôs-se a implementar e descrever o funcionamento de um servidor de gerenciamento de emails e algumas operações realizadas por ele, bem como fazer a análise computacional de desempenho e acesso à memória dos métodos implementados.

A análise feita sobre o programa acaba se tornando mais complexa, devido à complexidade das estruturas de dados implementados. No entanto, com a geração de gráficos e tabelas feitas durante as análises, é possível visualizar o comportamento da estrutura com diferentes tamanhos de entradas (número de usuários, número de mensagens, tamanho das mensagens e variação nas operações realizadas) ao longo da execução do programa, tornando a análise mais simples.

7. Bibliografia

CHORMEN, Thomasb H. *et al*. **Algoritmos - Teoria e Prática**. 3ª edição.
Capítulo 11: Tabelas de espelhamento. Editora Elsevier, 10 de abril de 2012.

CHORMEN, Thomasb H. *et al*. **Algoritmos - Teoria e Prática**. 3ª edição.
Capítulo 12: Árvores de busca binária. Editora Elsevier, 10 de abril de 2012.

APÊNDICE A - INSTRUÇÕES PARA COMPILAÇÃO E EXECUÇÃO

A compilação do programa pode acontecer apenas rodando o comando **make**.

Para executá-lo, utilize o comando:

bin/tp3 -i <arq> **-o** <arq>

- **i:** representa o caminho para o arquivo contendo a entrada do programa. É necessário que o arquivo exista.
- **o:** representa o caminho para o arquivo na qual será escrita a saída do programa.