Trabalho Prático 3

Gabriel Franco Jallais

Departamento de Ciência da Computação – Universidade Federal de Minas Gerais(UFMG)

Belo Horizonte – MG – Brasil

gfjallais@ufmg.br

Entrega: 04/07/2022

1. Introdução

O problema proposto para este trabalho foi elaborar um sistema de gerenciamento de e-mails, um servidor, que suportasse as seguintes operações: entrega, consulta e remoção de e-mails. Para o desenvolvimento de tal servidor, nos foi solicitado a elaboração de três TADs, um para o servidor, outro para as caixas de entrada, e outro para os e-mails, sendo o servidor uma tabela hash e as caixas de entrada árvores binárias. Para realizar as operações deve ser passado um arquivo contendo o tamanho da tabela hash que o servidor deverá utilizar na primeira linha, e nas linhas seguintes os comandos desejados.

Para tal tarefa, foram criados apenas classes referentes aos TADs solicitados, visto que não são necessárias estruturaras adicionais.

Nas próximas seções serão dados mais detalhes quanto a implementação, a análise de complexidade das operações, além de quais foram as estratégias de robustez adotadas, quais testes foram feitos, junto a análise experimental, por fim, é apresenta a conclusão, a bibliografia utilizada e instruções para compilação e execução.

2. Implementação

O programa foi desenvolvido na linguagem C++, compilada pelo compilador G++ da GNU Compiler Collection.

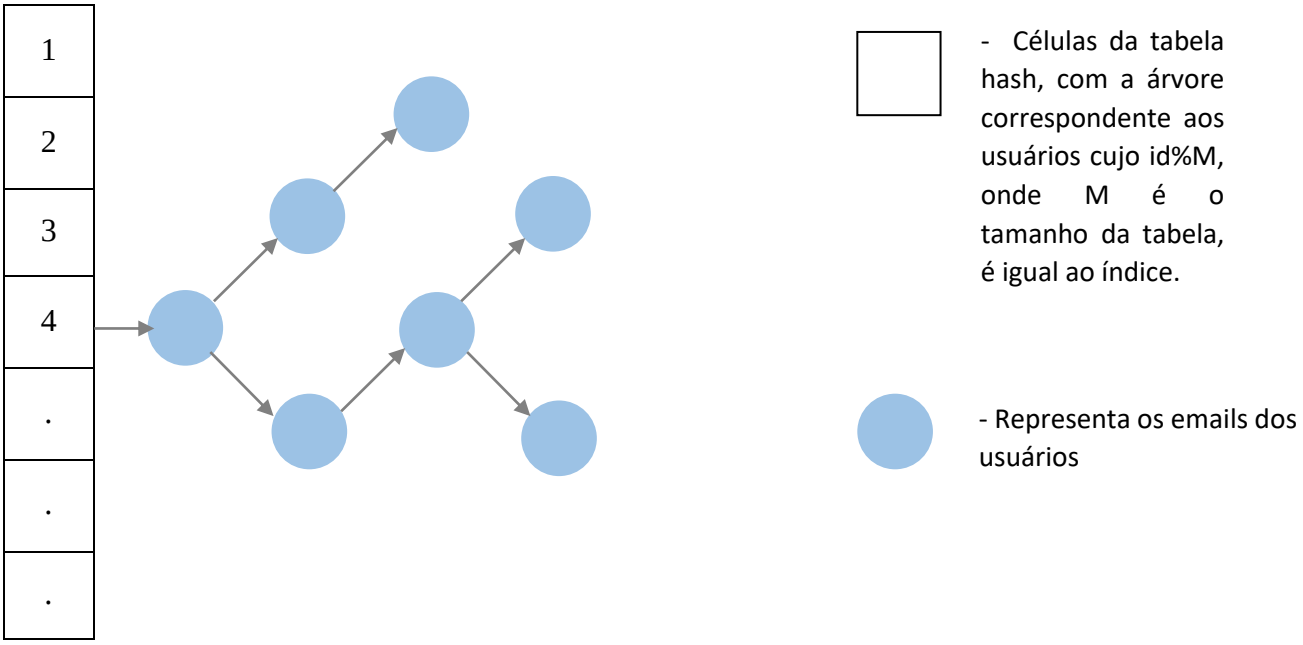
Foi compilado e executado no sistema operacional Ubuntu 20.04 dentro do WSL2 no Windows 11, com um processador i7-1165G7 com 16 GB de memória RAM.

2.1 Estrutura de Dados

A implementação do programa teve como base a uma tabela hash (servidor) e diversas arvores binárias uma para cada index da tabela, armazenando cada e-mail em um nó da arvore correspondente ao destinatário.

Para tornar mais simples a implementação não foi criada uma classe específica para um node da árvore, os próprios e-mails possuem essa função.

Segue um esquema da estrutura de dados:



2.1.1 Implementação da Tabela Hash (Servidor)

Servidor: Possui como atributos privados um vetor de ***caixa_de_entrada***, um inteiro ***tam***, que armazena o tamanho da tabela e uma *fstream* saída, que armazena o arquivo de saída do servidor, no qual ele imprimirá os resultados das operações solicitadas.

A classe servidor possui como métodos:

servidor(int *tam*, ***std::string*** *saida)**:*** Aloca o vetor de caixa_de_entrada de tamanho *tam*, abre o arquivo com nome *saída*, checa se este foi aberto corretamente e armazena *tam*.

~servidor(): Fecha o arquivo de saída.

void entrega_email(int *id_usuario*, ***email*** *em)**:*** Realiza a inserção do email passado na árvore correspondente a *id_usuario*, e imprime as mensagens de confirmação da inserção ou erro.

void consulta_email(int *id_usuario*, ***int*** *id_email)**:*** Busca pelo e-mail cujo id_email e id_usuario correspondam com os parâmentros passados, e caso o e-mail seja encontrado uma mensagem com as suas informações é impressa, caso contrário é informado que a mensagem não existe.

void apaga_email(int *id_usuario*, ***int*** *id_email)**:*** Busca pelo e-mail cujo id_email e id_usuario correspondam com os parâmentros passados, caso o e-mail seja encontrado ele é apagado da árvore, em caso de dois filhos, o sucessor é

escolhido, nos demais casos segue-se o padrão. Também são impressas mensagens exibindo o sucesso ou não da remoção.

void insert(T** item, **bool** count_duplicates**): Caso *count_duplicates* seja false, insere o item passado na lista, caso seja true, antes de inserir, busca o item na lista, se encontrado apenas incrementa a quantidade deste e não insere nada na lista, caso contrário insere o item na lista.

void processa_opcao(std::string** str**): Recebe a string referente a operação desejada, "ENTREGA", "CONSULTA" e "APAGA", faz as devidas leituras, dos ids e da mensagem, e chama os outros métodos da classe referentes as operações passadas.

A implementação desta classe se encontra no arquivo servidor.cpp.

2.1.2 Implementação da Árvore Binária (Caixa de entrada e e-mail)

email: Possui como atributos privados: inteiros ***id_email***¸ que armazena o id do email, e ***id_usuario***, que armazena o id do destinatário, uma std::string ***mensagem***, que armazena a mensagem do e-mail, e dois ponteiros para e-mail, ***left*** e ***right***.

A classe email possui como métodos:

email(): Define valores padrão para cada atributo, no caso, -1 para os inteiros, "" para a mensagem e ***nullptr*** para os ponteiros.

email(int** id_email**, int** id_usuario, **std::string** msg**): Atribui os valores passados para os atributos correspondentes e ***nullptr*** para os ponteiros.

int get_id_email(): Retorna o id do email.

int get_id_usuario(): Retorna o id do destinatário do email.

std::string get_msg(): Retorna o a mensagem do email.

void set_email(int** id_e**, int** id_u**, std::string** msg**): Atribui os valores passados aos atributos correspondentes.

caixa_de_entrada: A classe possui como atributo privado apenas um ponteiro de email, ***root***, para armazenar a raíz da árvore.

A classe email possui como métodos:

caixa_de_entrada(): Apenas atribui ***nullptr*** para o atributo ***root***.

~caixa_de_entrada(): Chama o método ***limpa*** da classe.

void limpa(email n**):*** Desaloca cada email/node da árvore.

email* insere(email e**):*** Chama o método ***insere_recursivo*** da classe.

email*insere_recursivo(email e, **email* &**atual**):*** Insere o email de forma recursiva na árvore.

email* consulta(int** id_email**): Retorna a chamada para o método ***pesquisa***.

email* pesquisa(int** id_email, **bool** pai**):** Busca pelo e-mail com o id passado como parâmetro, caso *pai* seja ***false, o retorna, caso contrário, retorna o pai.

email* get_sucessor(email *atual***):** Busca pelo sucessor do e-mail passado, esse método é utilizado para ajudar na remoção de e-mails da árvore.

email* min_val(email *atual***):** Busca o filho mais a esquerda de um email, método auxiliar a ***get_sucessor***.

***email* apaga(int** id_email, **int** id_usuario**):** Busca pelo e-mail desejado e o apaga, cobrindo os casos em que ele é folha, quando possui apenas um filho e quando possui dois filhos, utilizando de outros métodos da classe para isso.

A implementação destas classes se encontra nos arquivos caixa_de_entrada.cpp e email.cpp.

2.2 Funções

Foi utilizada a função ***parseArgs(int** argc, **char*** argv**)***, passada no primeiro trabalho, para realizar o processamento dos argumentos passados na execução do programa.

3. Análise de Complexidade

Para a análise de complexidade do programa, serão levadas em conta apenas as funções de pesquisa e inserção na caixa de entrada, visto que são as únicas partes do programa que dependem da variação da entrada, como os e-mails dos usuários cujo id%m, onde m é o tamanho da tabela hash são armazenados na mesma árvore o custo para encontrar a respectiva árvore é constante, ***O(1)***.

Análise de complexidade de espaço:

Inserção na árvore binária: Como são feitas chamadas recursivas para a inserção o custo da stack de chamadas é linear ***O(n)***.

Consulta na árvore binária: Como não são utilizadas estruturas auxiliares nem chamadas recursivas para nenhuma operação o custo de espaço da pesquisa/consulta é ***O(1)***.

Remoção na árvore binária: Como são feitas chamadas recursivas para a remoção, quando o e-mail que se quer apagar possui dois filhos, o custo da stack de chamadas é linear ***O(n)***.

Análise de complexidade de tempo:

Inserção na árvore binária: Como não foram feitas otimizações quanto o balanceamento da árvore, como a utilização de uma árvore AVL, o custo de inserção de um novo e-mail na caixa de entrada é ***O(n)*** e não ***O(logn)***¸ visto que a árvore pode estar degenerada. Onde n é o número de e-mails na árvore.

Consulta na árvore binária: Assim como a inserção, a consulta/pesquisa na árvore binária elaborada possui custo linear, ***O(n)***, e da mesma forma utilizando-se de métodos de balanceamento o custo cairia para ***O(logn)***.

Remoção na árvore binária: Assim como nos demais casos o custo é linear e poderia, mas poderia ser logarítmico caso fosse efetuada alguma forma de balanceamento na árvore para evitar o pior caso.

- *A execução do arquivo main.cpp*

Como todas as operações envolvidas no programa foram elaboradas com custo linear o custo geral do programa é um múltiplo escalar desse custo o que resulta em ***O(n)*** em relação ao tempo e ao espaço.

4. Estratégias de Robustez

Para garantir a boa operação do programa, isto é, que ele seja robusto para tolerar erros do usuário e se defender de tentativas de "ataque" que visam achar falhas no programa para explorá-las, foram tomadas as seguintes estratégias.

Primeiramente, é feita a leitura dos argumentos para a execução e checa se eles são válidos para que os arquivos abram corretamente e as operações sejam executadas da maneira certa, para isso, são usadas as funções *parseArgs(int argc,char ** argv)*, o método open da classe fstream e a função *erroAssert(e,m)* da biblioteca msgassert.h.

Ademais, para cada função/operação descrita na seção 2, são feitas checagens para que as entradas cumpram os requisitos para as operações requisitadas, isto é, para que não sejam acessados endereços de memória indesejados nem se tente efetuar operações em ponteiros nulos.

5. Testes

Para analisar desempenho computacional do programa, além de fazer a análise do padrão de acesso à memória e localidade de referência, foram feitos testes utilizando um gerador de texto lorem ipsum (https://www.lipsum.com/):

Para a análise de desempenho, a análise de memória não foi ativada, para melhor fidelidade e representação do desempenho do programa, e foram dadas entradas de 1000 a 10000 palavras indo de 500 em 500, com m = 5 e s =10.

Já para a análise de acesso e localidade de memória, foi utilizada uma entrada pequena, com apenas 65 palavras, o arquivo 1.tst.i, passado como exemplo de input junto ao enunciado do trabalho.

6. Análise Experimental

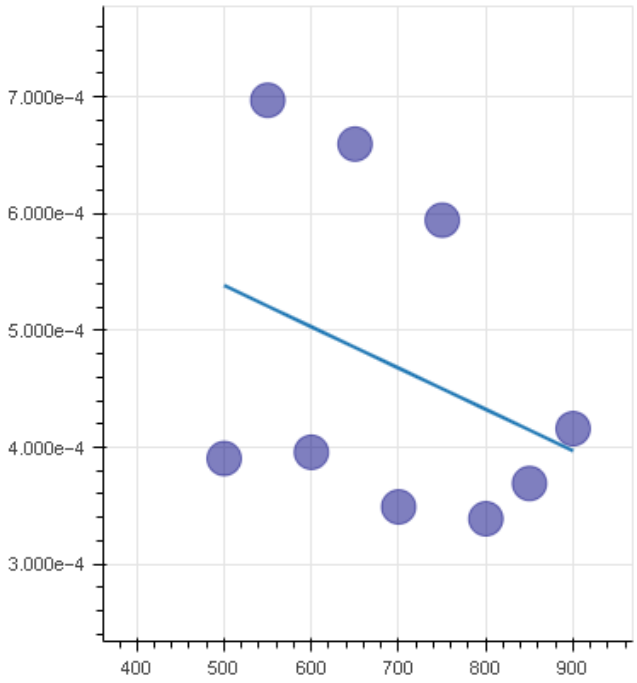
Para efetuar a análise do programa em questão foram utilizados os softwares memlog e analisamem, que foram responsáveis por mapear o tempo de execução e o acesso à memória.

6.1 Desempenho Computacional

Para verificar a análise de complexidade feita previamente, foi feitas sequências de testes e curvas foram traçadas para capturar o comportamento do programa em função do aumento da entrada.

- Análise com variação no número de usuários:

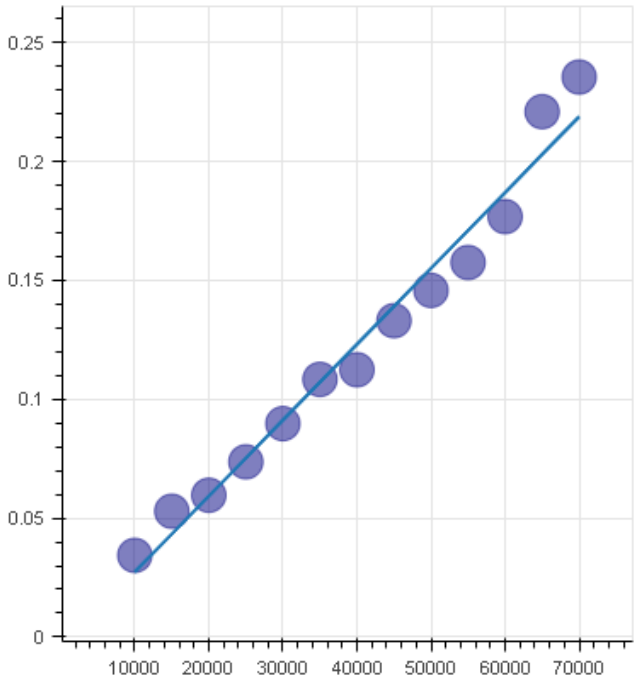
Foram levadas em conta entradas tamanho de tabela 100, 100 entregas, 50 consultas, 50 remoções e mensagens de tamanho entre 1 e 30. O n variou de 50 em 50 a partir de 500 até 900.



A variação no número de usuários traz diferenças insignificantes na variação de tempo, e como a grandeza do tempo é muito pequena a oscilação do sistema impede a formação de um gráfico que demonstre a complexidade. Portanto, sem a variação no número de operações o número de usuários é insignificante para a performance do programa.

- Análise com variação no número de mensagens:

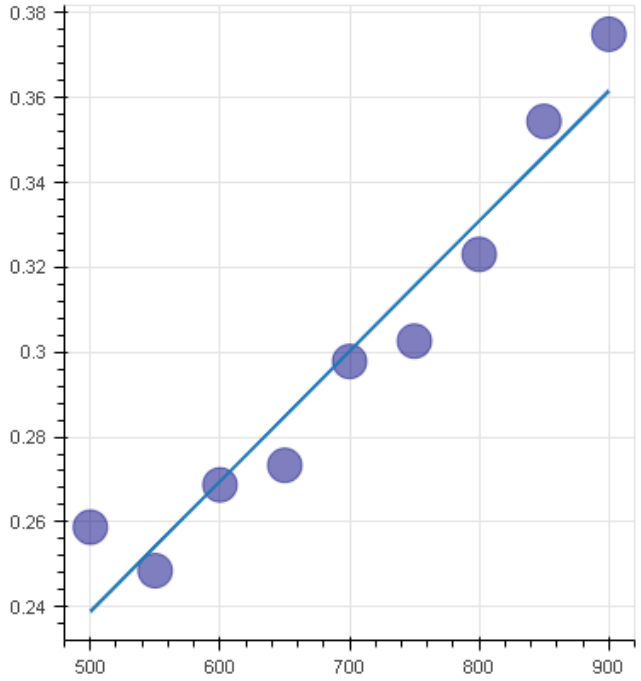
Foram levadas em conta entradas tamanho de tabela 100, 0 consultas, 0 remoções, 150 usuários e mensagens de tamanho entre 1 e 30. O n variou de 5000 em 5000 a partir de 10000 até 70000.



Como o número de mensagens implica no número de entregas, pode-se observar como o comportamento das experimentações foi coerente com a análise de complexidade feita na seção 3.

- Análise com variação no tamanho das mensagens:

Foram levadas em conta entradas tamanho de tabela 100, 0 consultas, 0 remoções, 10000 entregas, 150 usuários. O n variou de 50 em 50 a partir de 500 até 900.



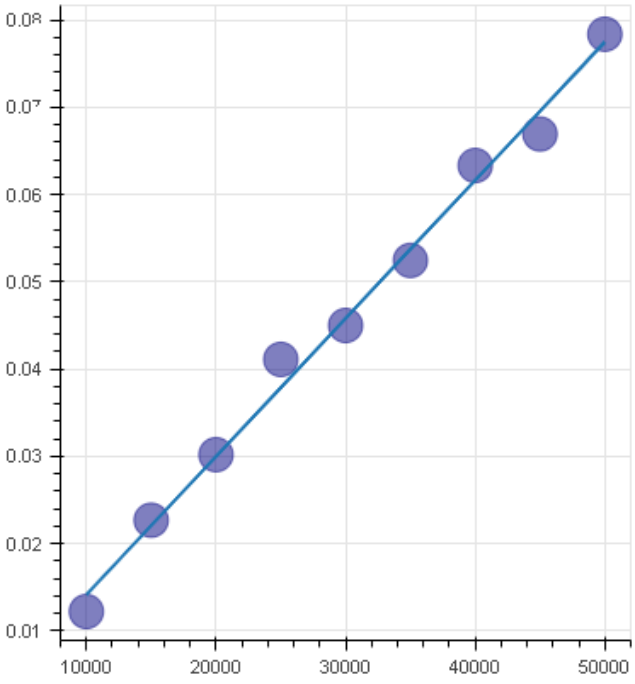
A partir dos experimentos feitos, pode-se observar que a variação no tamanho da mensagem implica em uma complexidade ***O(n)*** possivelmente pela alocação e cópia das strings.

- Análise com variação na distribuição de operações:

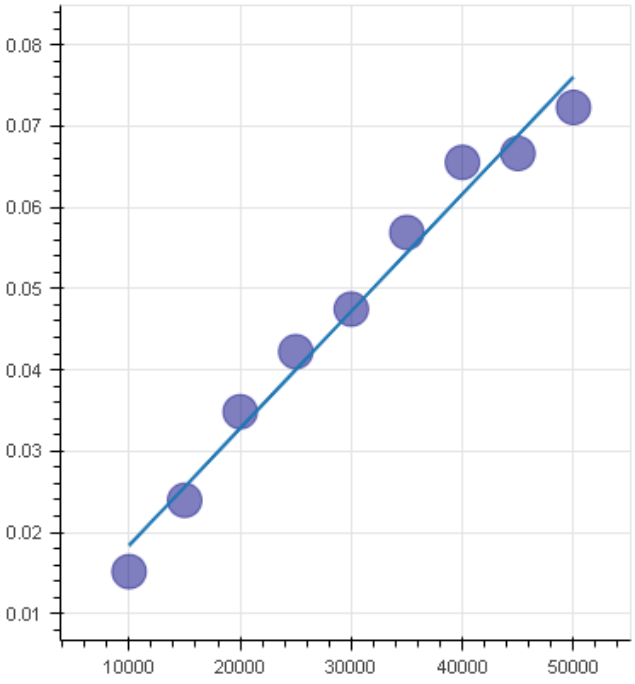
Para realizar a análise do comportamento do programa com a variação na distribuição de operações, foram feitas 3 análises, onde em cada uma as operações foram distribuídas em (3/20)*n, (3/20)*n, (14/20)*n, havendo permutação em cada análise, e com n variando de 10000 a 50000 de 5000 em 5000.

A tabela hash foi alocada com tamanho 100, utilizou-se de 150 usuários e mensagens de tamanho entre 1 e 30.

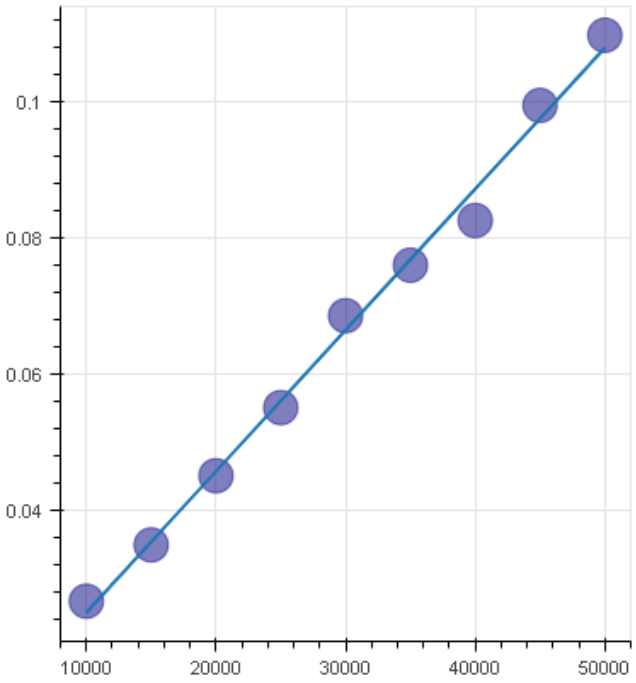
Análise 1: (3/20) Entregas, (3/20) Consultas, (14/20) Remoções



Análise 2: (3/20) Entregas, (14/20) Consultas, (3/20) Remoções



Análise 3: (14/20) Entregas, (3/20) Consultas, (3/20) Remoções



Como pode-se observar, a performance do programa se comporta de maneira linear conforme a entrada. Com base nas amostras obtidas, observa-se que quando as entregas possuem a maior parcela o tempo de execução foi relativamente maior que nas demais análises, seguida das consultas e das remoções. Talvez pelo fato de o gerador de carga gerar um número razoável de consultas e de remoções com ids que não estão presentes na árvore.

Vê-se nas análises feitas acima que a análise feita na seção 3 se confirma, visto que em todos os casos o programa apresentou custo linear conforme o aumento dos n em cada caso, com exceção do número de usuários, mas isso já era esperado.

Os resultados para cada entrada são oscilantes devido ao hardware e às outras atividades que estavam sendo executadas em paralelo no computador, levando isso em consideração foram feitos alguns testes para tentar melhorar a precisão da análise.

6.1.1 Resultados do gprof

Para analisar os resultados do gprof, utilizaremos uma entrada com 500000 entregas, consultas e remoções, tabela hash de tamanho 100, 150 usuários, e mensagens com tamanhos entre 1 e 100, para haver uma visualização mais direta do funcionamento do programa e para que o tempo de execução seja grande o suficiente para gerar resultados visíveis.

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self Ts/call	total Ts/call	name
59.75	0.56	0.56				email::get_id_email()
9.69	0.65	0.09				servidor::processa_opcao(std::string)
8.07	0.72	0.08				caixa_de_entrada::pesquisa(int, bool)
5.92	0.78	0.06				caixa_de_entrada::insere_recursivo(email*, email*&)
4.31	0.82	0.04				std::string<> std::operator+< >std::string
2.15	0.84	0.02				hashing(int, int)
2.15	0.86	0.02				email::email(email const&)
2.15	0.88	0.02				servidor::consulta_email(int, int)
2.15	0.90	0.02				main
1.08	0.91	0.01				caixa_de_entrada::consulta(int)
1.08	0.92	0.01				bool std::operator==< >std::string<>
1.08	0.93	0.01				std::operator|(std::_Ios_Openmode, std::_Ios_Openmode)
0.54	0.93	0.01				email::email(int, int, std::string)

***Alguns dados tiveram seus nomes reduzidos no resultado para fins de apresentação.**

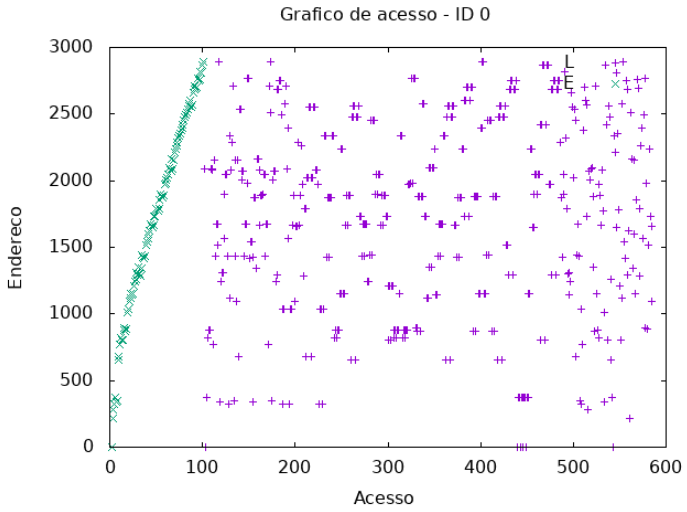
Vê-se como o método get_id_email() é a função predominante no programa, o que faz sentido, visto que ela está presente em todas as três operações, seguida do método processa_opcao que é chamada para todas as 1500000 linhas. Em suma as funções que consomem mais tempo são aquelas que estão mais presentes nas diferentes partes do programa.

6.2 Análise de Padrão de Acesso à Memória e Localidade de Referência

Agora, para entender melhor como o programa acessa memória e a sua localidade de referência, observaremos os gráficos gerados por meio do software gnuplot, e cujos dados foram fornecidos pelo programa analisamem, utilizaremos uma entrada com 100 entregas, consultas e remoções, tabela hash de tamanho 100, 150 usuários, e mensagens com tamanhos entre 1 e 30, para haver uma visualização mais direta do funcionamento do programa e para que o tempo de execução seja grande o suficiente para gerar resultados visíveis.

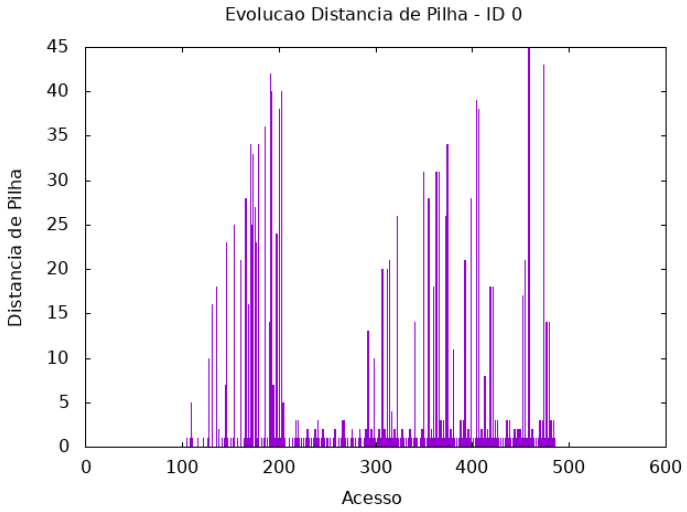
Primeiramente, cabe dizer que, o id 0 é referente as árvores binárias de cada usuário. Os gráficos mostram como foi feito o acesso à memória ao decorrer do tempo, pontos (+) próximos no eixo x indicam endereços que foram acessados em intervalos próximos de tempo.

O programa main.cpp segue os seguintes passos: leitura do arquivo de entrada, em seguida, a criação do servidor, e leitura dos comandos.

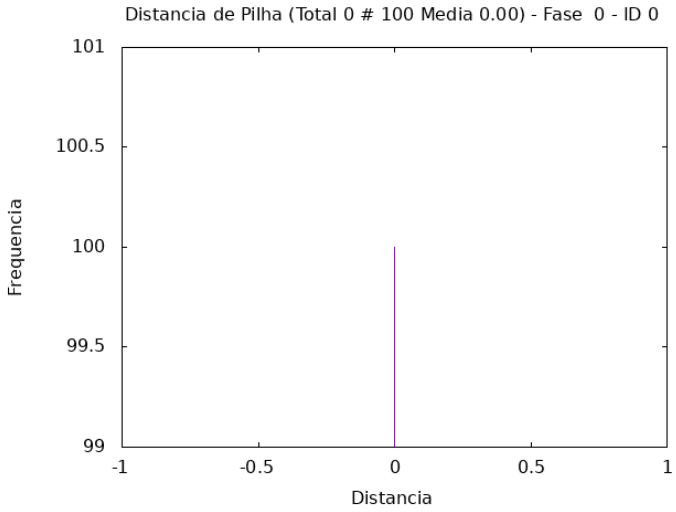


Portanto, vê-se, na faixa verde, mais à esquerda a alocação dos nós/e-mails da árvore, seguido de consultas, remoções e por fim a liberação da memória. Após a inserção o acesso a memória se mostra mais caótico, visto que são diversos ponteiros para árvores armazenadas de forma contígua, mas cada nó é armazenado de forma não sequencial, isto é, por meio de ponteiros, o que justifica a esparsidade no gráfico.

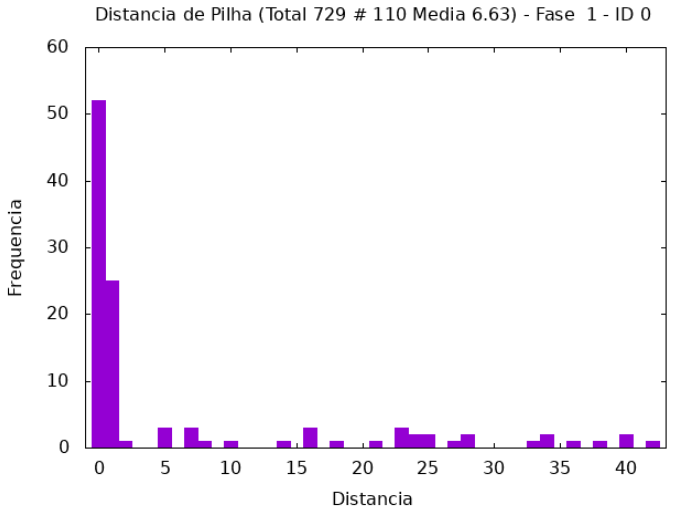
Da mesma forma observa-se o impacto do acesso não contíguo da memória no gráfico de evolução da distância de pilha abaixo, as partes onde a distância de pilha é maior provavelmente são as árvores com mais mensagens ou mais acessadas.



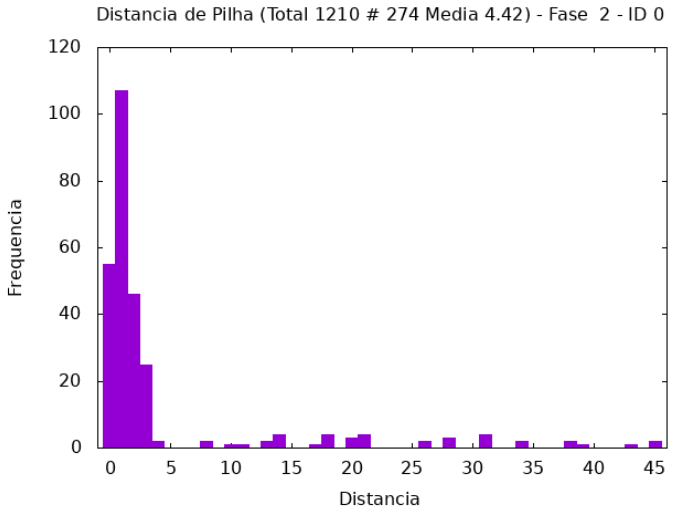
A primeira fase, fase 0, engloba apenas a criação de cada árvore, por isso o histograma é mais uniforme.



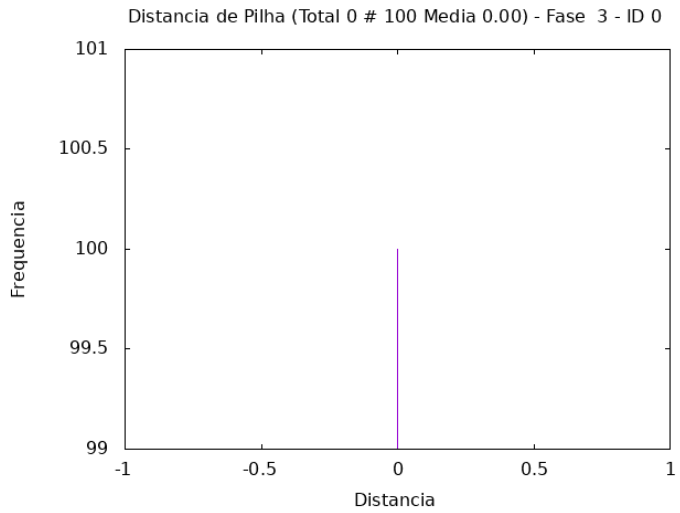
Já a 1, captura o comportamento das consultas, é aí que vemos o comportamento caótico visto nos gráficos de acesso e de evolução da distância de pilha, conforme os ids de usuários e e-mails e o tamanho de cada árvore, devido ao caminhamento na árvore, as distâncias de pilha variam.



Na 2, o comportamento é bem similar, por envolver as remoções e, consequentemente, a pesquisa pelos e-mail na árvore.



Por fim, na terceira fase, a memória de cada árvore é desalocada da tabela é desalocada e observa-se um comportamento mais uniforme.



7. Conclusões

O trabalho consistiu em simular o funcionamento de um servidor de e-mails que suporte as operações de entrega, consulta e remoção de e-mails, utilizando-se de uma tabela hash para otimizar a busca por usuários e seus respectivos e-mails, além de árvores binárias para representar cada caixa de entrada, permitindo com que, dependendo do nível de balanceamento da árvore, a busca seja mais otimizada que outros métodos, possuindo custo próximo ou igual a ***O(logn)***.

Por se tratar de uma tarefa relativamente simples, não houve grandes dificuldades para a sua realização, sendo a tarefa de análise a mais laboriosas. Pôde-se observar como as funções seguem a análise assintótica feita na seção 2, e como o acesso a memória e a distância de pilha foram coerentes com a implementação feita para as operações e coerentes com as entradas fornecidas.

8. Bibliografia

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Algoritmos: Teoria e Prática. 3a edição. Elsevier, 2012. ISBN 9788535236996

Nivio Ziviani. Projeto de Algoritmos com implementações em Pascal e C. 3ª edição. Cengage Learning, 2011. ISBN 9788522110506

Apêndice

Instruções para compilação e execução

Para facilitar a compilação e execução do programa main.cpp, foi disponibilizado um arquivo Makefile com comandos mais simples.

Para compilar o programa, basta possuir o arquivo Makefile no mesmo diretório que os outros programas, estruturados de forma:

|- src
|- bin
|- obj
|- include
Makefile
entrada.txt

Onde src deve possuir os arquivos .cpp e include deve possuir os arquivos .h e executar o comando “make comp”.

Para executar o programa basta executar o comando na linha de comando no seguinte formato:

(EXE) (COMANDO OPICIONAL -p) (SAÍDA DA ANÁLISE DE DESEMPENHO) (COMANDO OPICIONAL -l) -[i|I] (ARQUIVO DE ENTRADA) -[o|O] (ARQUIVO DE SAÍDA)

EXE = tp3
COMANDO OPICIONAL 1 = -p, para a geração de arquivo .out com dados de desempenho computacional
COMANDO OPICIONAL 2 = -l, para a geração de mapas de acesso a memória
ARQUIVOS DE ENTREDA = entrada.txt
ARQUIVO DE SAÍDA = saída.txt

Para realizar as experimentações vistas na documentação basta executar o comando “make mem”, para gerar os mapas de acesso e histogramas, o comando “make perf”, para gerar arquivos .out para a análise de desempenho, e o comando “make gprof”, para gerar arquivos .gprof para a análise de desempenho. ***É necessário possuir o programa analisamen.c**

Por fim, o comando “make clean”, pode ser utilizado para apagar os arquivos .o e gmon.out gerados