Trabalho Prático 3

Servidor de e-mails otimizado

João Vitor Ferreira

Universidade Federal de Minas Gerais (UFMG)

Belo Horizonte — MG — Brasil

joaovitorferreira@ufmg.br

1. Introdução

Este arquivo visa documentar e analisar o funcionamento do programa “simulador”. Implementar um código em que realiza a simulação de um servidor de e-mails. Com suporte a 3 operações a Entrega de e-mails, ou seja, o envio, consulta de e-mail e a exclusão de e-mails, sendo que os dados são lidos em um arquivo especificado como argumento da linha de comando e salvando os resultados em outro arquivo.

2. Método

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

2.1. Estrutura de Dados

A implementação teve como base o uso de 3 classes que armazenam os dados da Mensagem, a implementação da tabela Hash e da Árvore Binaria.

Os métodos e dados guardados nessas classes são:

Mensagem:

Mensagem() — construtor para do TAD.

void SetIdMensagem(int _id_mensagem), void SetConteudo(string _msg_conteudo), void SetIdDestinatario(int _id_destinatario) — Guardam os valores de id da mensagem, o conteúdo do e-mail e o id do destinatário.

int GetIdMensagem(), int GetIdDestinatario(), string GetConteudo() — Quando chamadas retorna o dado guardado no id da mensagem, o conteúdo do e-mail e o id do destinatário

int id_mensagem — id do e-mail

int id_destinatario — id do destinatário do e-mail

string conteudo — conteúdo do e-mail

string arquivo_saida — nome do arquivo de saída passado como parâmetro “-o” na linha de comando na execução do programa

int indice_analisamem — índice artificial que determina qual tipo de operação está sendo realizada. Usado apenas para fins de análise de desempenho usando a biblioteca analisamem.

Arvore_Binaria:

Métodos públicos:

ArvoreBinaria() — construtor para do TAD Arvore Binaria

~ArvoreBinaria() — destrutor para do TAD Arvore Binaria

void Insere(Mensagem email) — Método para procura e inserção do e-mail em seu respectivo nó na árvore, é feito no método InsereRecursivo.

void Limpa() — Método para limpar todos os nós da Árvore.

Mensagem Pesquisa(Mensagem email) — Método para realizar a pesquisa de dado e-mail, feita através do método PesquisaRecursivo.

void Remove(Mensagem email) — Método para realiza a procura e caso esteja presente realiza a remoção no e-mail e consequentemente seu nó, a remoção é feita pelo RemoveRecursivo.

Métodos privados:

void InsereRecursivo(TipoNo *&p, Mensagem email) — Recebe um nó e um e-mail, percorre a Árvore até chegar numa folha e realiza a inserção do e-mail.

void ApagaRecursivo(TipoNo *p) — Percorre todos os nós da árvore e define eles como Nulo, dessa forma apagando os dados da Árvore.

Mensagem PesquisaRecursivo(TipoNo *p, Mensagem email) — Percorre todos os nós da árvore até que o nó procurado seja encontrado e retornado. Caso ele não seja encontrado retorna o id -1 que representa um e-mail não presente na árvore.

void RemoveRecursivo(TipoNo *&p, Mensagem email) — Percorre a árvore até encontrar o nó desejado e realiza a remoção dele, depois faz a troca dele com um de seus filhos, caso ele possua.

void Antecessor(TipoNo *q, TipoNo *&r) — Realiza a rotação do elemento da árvore que acabou de ser removido.

TipoNo *raiz — Classe amiga da Árvore que trata os dados do nó como seu conteúdo e carrega os apontadores de filhos a direita e esquerda.

Hash_AB:

Métodos públicos:

Hash_AB(int M) — Construtor do Hash, que inicializa a árvore binaria de tamanho ‘M’

Mensagem Pesquisa(Mensagem email, int M, int Tipo) — Método que trata a pesquisa, realiza o Hash para encontrar a posição do e-mail na árvore e faz a pesquisa dele, na Árvore. Além de realizar a escrita se a operação de pesquisa foi bem sucedida ou não, resultando na Saída: “CONSULTA U E: ”, sendo ‘U’ o id do destinatário e ‘E’ o id da mensagem e após essa mensagem é impresso o e-mail armazenado se houver ou “MENSAGEM INEXISTENTE”, no caso dele não ser encontrado.

void Insere(Mensagem email, int M) — Método que realiza a inserção, realiza o Hash para encontrar a posição do e-mail na árvore e faz a inserção dele, na Árvore. Além de realizar o apontamento da execução da ação no arquivo de saída. Que

segue o molde “OK: MENSAGEM E PARA U ARMAZENADA EM b”, sendo ‘E’, ‘U’ e ‘B’, respectivamente, o id da mensagem, o destinatário e a posição dessa mensagem na árvore.

void Remove(Mensagem email, int M) — Método que realiza a remoção, realiza o Hash para encontrar a posição do e-mail na árvore após ele realiza a pesquisa para que retorna se o elemento foi encontrado ou não, caso ele não seja encontrado é escrito no arquivo de saída, a mensagem “ERRO: MENSAGEM INEXISTENTE”, caso seja encontrado sua remoção é realizada e a mensagem “OK: MENSAGEM APAGADA” é guardada no arquivo de saída.

Métodos privados:

int Hash(Mensagem email, int M) — Realiza o Hash do id da mensagem com o tamanho da tabela, sendo essa operação o mod.

ArvoreBinaria *Tabela — Criação de uma ArvoreBinaria denominada Tabela.

Outras funções foram implementadas elas são:

void Executa(char *arquivo_entrada, char *arquivo_saida) —Função que realiza a execução do programa e faz a chamada das funções do simulador. Nela que a tabela e a mensagem são inicializadas sendo feita a leitura dos dados no arquivo de entrada, além da destruição da tabela.

Simulador contém somente os comandos necessários para que as operações de Inserção(void Entregar_email(Hash_AB *server, Mensagem email, int U, string conteudo, int M, int E)). Consulta (void Consultar_email(Hash_AB *server, Mensagem email, int U, int M, int E)) e Exclusão (void Apagar_email(Hash_AB *server, Mensagem email, int U, int M, int E)) seja executada. Essas foram distribuídas em 3 funções uma para cada operação, os parâmetros delas são os necessários para ela ser implementada, logo, é passado a tabela Hash, o e-mail, o id do destinatário e o id da mensagem para as 3, somente a função de inserção que possui uma string conteúdo que a mensagem no e-mail a ser armazenado.

int main(int argc, char **argv) — No main do programa nele é iniciado e finalizado a análise de Complexidade e chamado a função Executa, além de analisar os comandos passos pela linha de comando e convertidos para variáveis usadas no programa.

2.2. Classes

A modularização foi implementada ao longo do código, por tanto foi necessário a criação de 4 TADs, estas das quais foram denominadas, Mensagem, armazena o conteúdo do e-mail, seu id e o id do destinatário, Nó que armazena os dados de cada no da Árvore Binaria, Arvore Binaria, que contém os dados para a implementação dela e Hash_AB, que implementa uma tabela Hash de Árvore Binaria.

2.3. Formato de Entrada e Saída

O formato de entrada é a inserção de uma pasta denominada entradas a qual conterá os arquivos com as entradas para o programa, o nome deste arquivo deve ser passado. A saída é padronizada e estará contida no arquivo de nome a ser determinado e está na pasta denominada saídas.

3. Análise de Complexidade

3.1. Tempo

Inicialmente seguiremos algumas preposições as estruturas auxiliares mais básicas consideraremos com o,$O(1)$ além de que as funções de definição e retorno de dados, construtores simples e demais funções que possuem custo $O(1)$ não serão analisadas, desde que essas funções não desempenhem um papel principal na execução do código. Dessa forma o custo das funções são:

void Executa(char *arquivo_entrada, char *arquivo_saida) – $O(n)$

void Entregar_email(Hash_AB *server, Mensagem email, int U, string conteudo, int M, int E) – $O(1)$

void Consultar_email(Hash_AB *server, Mensagem email, int U, int M, int E) – $O(1)$

void Apagar_email(Hash_AB *server, Mensagem email, int U, int M, int E) – $O(1)$

void ArvoreBinaria::InsereRecursivo(TipoNo *&p, Mensagem email) – $O(\log n)$

Mensagem ArvoreBinaria::PesquisaRecursivo(TipoNo *no, Mensagem email) – $O(\log n)$

void ArvoreBinaria::RemoveRecursivo(TipoNo *&no, Mensagem email) – $O(\log n)$

void ArvoreBinaria::ApagaRecursivo(TipoNo *p) – $O(n)$

void ArvoreBinaria::Antecessor(TipoNo *q, TipoNo *&r) – $O(n)$

Mensagem Hash_AB::Pesquisa(Mensagem email, int M, int Tipo) – $O(1)$

void Hash_AB::Insere(Mensagem email, int M) – $O(1)$

void Hash_AB::Remove(Mensagem email, int M) – $O(1)$

3.2. Espaço

O espaço utilizado é variável e dependera da quantidade de comandos que estão no arquivo, considerando um valor x de linhas da tabela Hash sera alocado uma quantidade x de posições, no entanto cada linha possui y e-mails, dado que o tamanho da tabela já está previamente definido. Considerando as funções são feitas usando ponteiros temos não haver grande adicional de memória sendo alocada ao longo da execução, usando assim de um espaço menor. Desta forma desconsiderarei o espaço usado por manipulações que utilizem de espaço que não foram previamente alocados e que minhas funções somente utilizem do espaço alocado inicialmente, então desta forma temos que o espaço é de,$O(\log n)$ em suma o espaço considerado sera,$\log n$ pois como estamos tratando de e-mails somente a inserção que aumentara o uso de memória as consultas não alteram a memória e a exclusão remove da memória. Portanto como o tamanho não é constante e é flexível o uso do espaço não

é fixo, para aumentarmos a precisão da quantidade de espaço utilizada poderíamos considerar como, sendo$O(\log_x n)$x representado pelas remoções, já que a remoção libera a memória alocada e a cada remoção bem sucedida o espaço é reduzido.

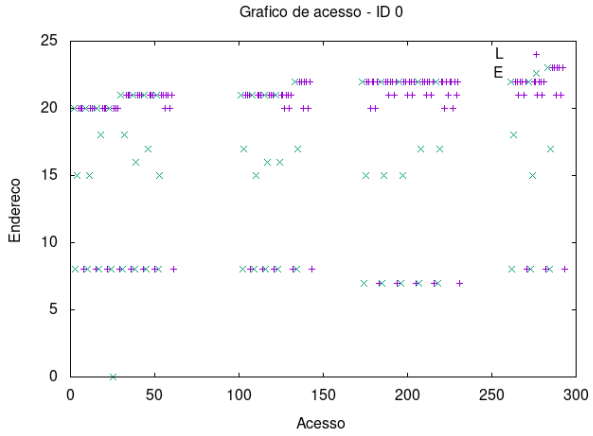
4. Estratégias de Robustez

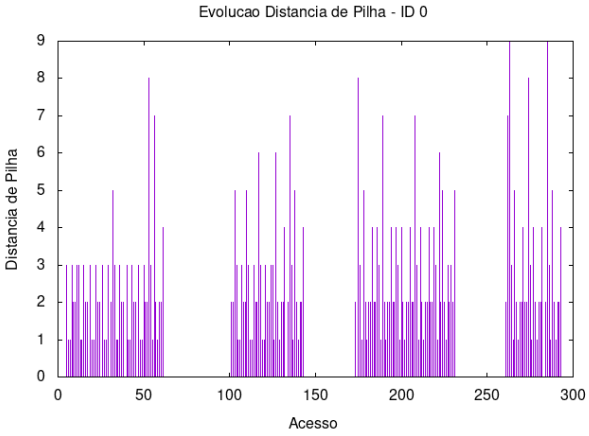
Para garantir o correto funcionamento do código foram adicionados errosAssert, dos quais param o código toda vez que um erro é localizado. Os erros adicionados estão relacionados com a abertura dos. No caso da abertura do arquivo caso ele está vazio um erro retornara avisando não haver nada no arquivo ou que ele não pode ser aberto.

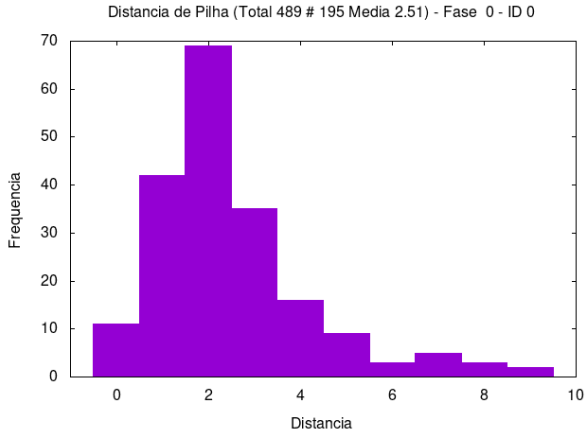
5. Análise Experimental

Para realizar a análise realizei a criação de um arquivo de entrada em que possuía 60 comando distribuídos entre ENTREGA, CONSULTA e APAGA. Para fazermos uma análise mais detalhada de como é a execução do programa, realizei a geração dos gráficos sendo cada um deles para um tipo de operação ou seja, há 1 gráfico de acesso, distância de pilha e evolução de distância de pilha para cada uma das operações. As operações são representadas pelos ids 0, 1 e 2, que representam, respectivamente, ENTREGA, CONSULTA e APAGA.

5.1 ENTREGA



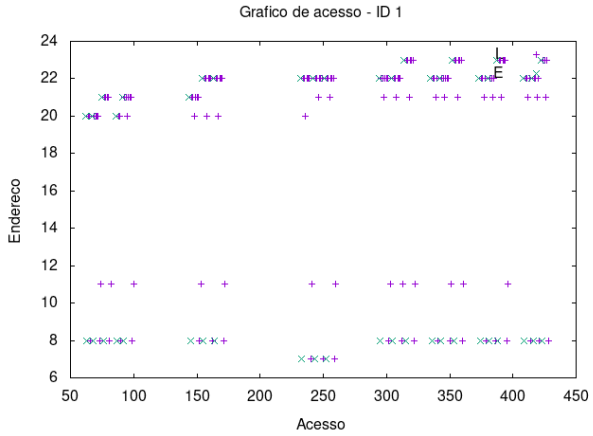


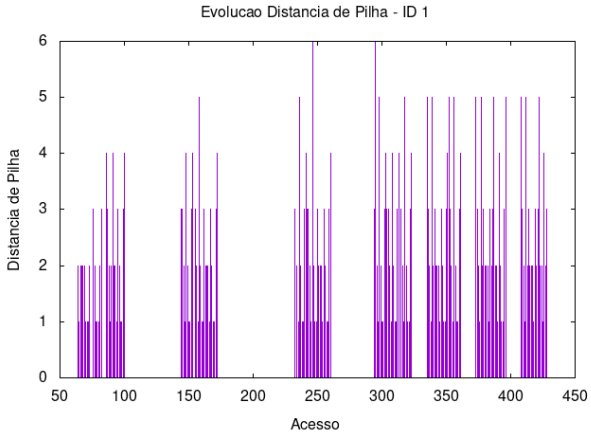


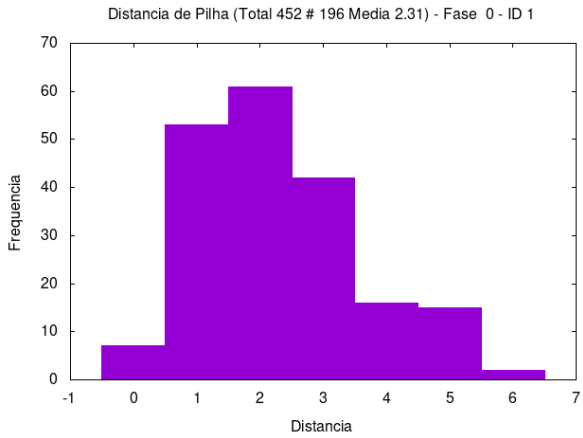
Analisando os gráficos de uma maneira mais geral é notório a forma onde a memória é acessada para leitura e escrita deriva exatamente da forma onde o programa foi desenvolvido. Nota-se que os acessos são em geral consecutivos, especialmente a escrita que em praticamente todos os casos é sempre feita em endereços contínuos, e a leitura segue a mesma lógica. Também é notável ver que a leitura e escrita dos dados é feita consecutivamente, logo todos os dados estão alinhados, também notamos não haver leitura em uma área do gráfico somente escrita, esta parte representa o índice virtual para essa análise que não é lido pelo código somente para esta análise de memória. Na evolução da distância de pilha notamos que ela segue certa consistência na maioria dessa análise, isso pode indicar que o caminhamento das árvores chegou em seu caso médio ou as arvoes está balanceado, ou seja, a distância a ser percorrida para realiza a inserção de determinado dado na árvore é “constante”, já nas partes onde se temos um acesso

maior, deriva do pior caso de da árvore em que ela se assemelha a uma lista. Por fim temos a distância de pilha, que esclarece o que foi inferido anteriormente, grande parte dos dados está a somente 2 acessos em média um do outros, o que é resultante da implementação da árvore em que balanceia e agiliza a forma em que inserimos os dados nela.

5.2 CONSULTA



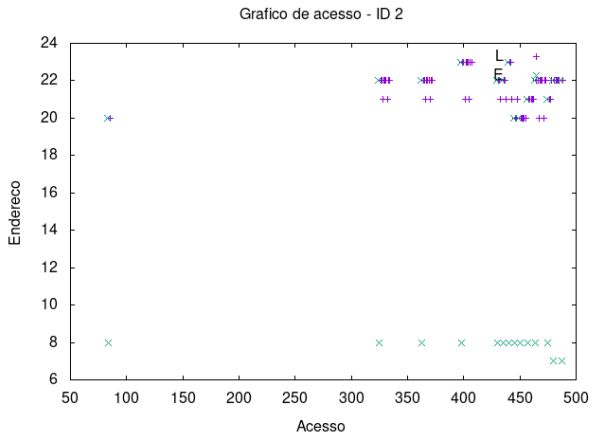


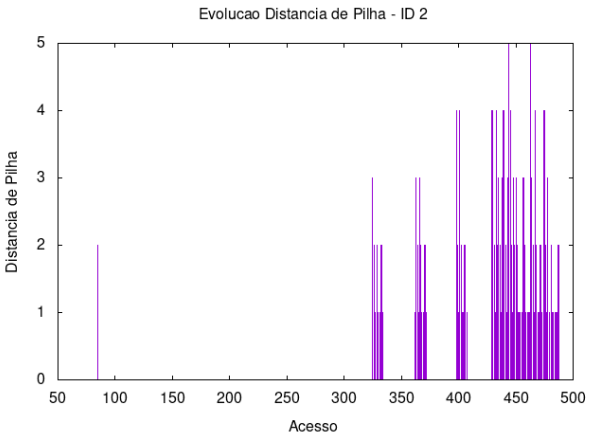


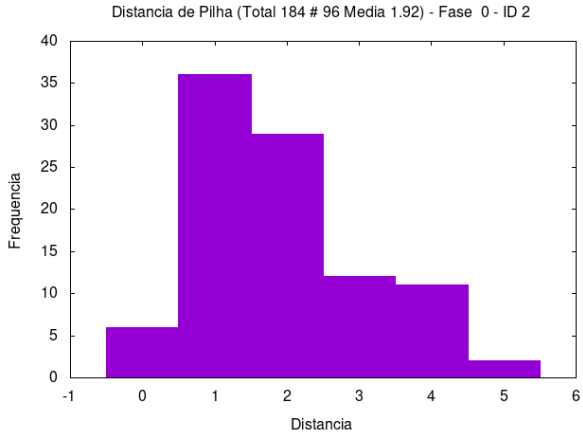
Na análise de acesso vemos nitidamente momento em que temos a definição da mensagem a ser procurada e analise consecutiva dos dados para encontrá-la, estas posições onde são procurados os dados coincidem exatamente, nas mostradas na inserção, o que já era esperado, dado que estamos realizando as pesquisas nos mesmo locais onde foram feitas as inserções. Outro fator interessante de se analisar neste gráfico é que há diversas leituras únicas, este resultado pode ser dar de somente

quando o dado pesquisado está na primeira posição da árvore, raiz, sendo desnecessário realizar o caminhamento na árvore. Já no caso em que temos diversas leituras seguidas temos 2 casos em que esse ponto pode ocorrer, quando o dado é encontras ele pode percorrer um alto número de nós para encontrá-lo, ou quando o dado não é encontrado sendo realizado o caminhamento até uma folha. A análise da evolução de distância de pilha reforça ainda mais as conclusões pois fazemos muitas consultas e a distância de pilha tende a não ser superior a 2 na grande maioria dos casos, e é extremamente alta em possíveis casos em que é realizada uma consulta de dado mensagem que não está contida no servidor. Essa neutralidade dos dados é comprovada no gráfico de distância de pilha em que vemos que a frequência de dado em 1, 2 ou 3 nos de distância da raiz é extremamente maior do que os outros. Demonstrando um dos grandes trunfos da árvore binária de pesquisa.

5.3 APAGA

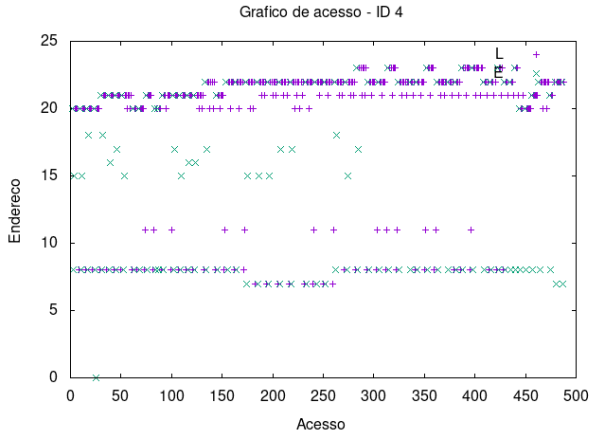


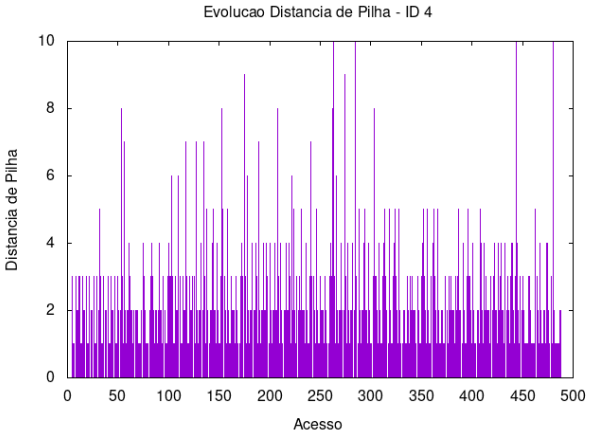


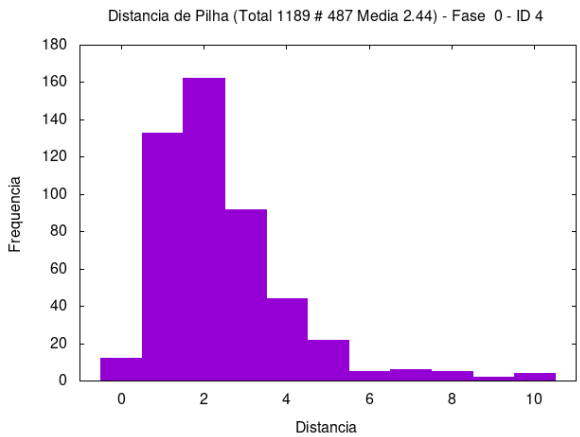


No gráfico de exclusão notamos a mesma característica das outras operações. A leitura nos mesmo locais e a procura dos dados para realizar a exclusão dos mesmo. Outro ponto interessante é que neste gráfico realizamos escritas sem leitura nenhuma realizada por elas, isso ocorre pois estamos deletando o conteúdo da mensagem, explicando não necessidade de leitura visto que esses dados estão interligados ao id. O qual foi pesquisado anteriormente. No de evolução da distância de pilha vemos que temos uma alto acesso em conteúdo de distância 1 e se deve ao fato do roteiro não realizar exclusão de dados muito profundos na árvore, caso esse fosse o caso teríamos um acesso maiores as posições mais profundas da árvore. Vemos também que temos alto caminhamento que se assemelham aos da consulta, julgo que sejam de dados que não foram inseridos e estão sendo consultados ou excluídos. Por fim temos o gráfico de distância e pilha que mostra que temos sim uma alta frequência em deletar mensagem nos primeiros nos dá arvore.

5.4 COMPLETO







Por fim realizei uma análise com esta mesma saída porém com somente 1 id para o programa inteiro. Analisando cada gráfico e juntando todos temos os gráficos gerados nessa análise. Aqui concluímos que todas as inferências feitas anteriormente são fatos que demonstram a eficiência de execução. Notamos que as escritas e leituras seguem exatamente o a mesma lógica no gráfico de acesso, também temos que a distância de pilha efetiva as suposições de que os acessos que passam muito da média são aqueles que não foram inseridos. Logo diante de todos esses dados temos que o desempenho do programa não configura o melhor caso de Hash com árvore binaria, mas está bem próximo, notamos um resultado que flutua bem entre melhor caso e caso médio, dado que possuímos acessos altos, pessimistamente assumiremos que nessa analise temos o caso médio.

5.5 GPROF

Ao tentar realizar a análise gprof não consegui gerar o arquivo necessário. Diferentemente dos TP anteriores não foi possível gerar arquivo nenhum desta vez, mesmo usando o comando -pg na execução. O arquivo gprof não foi gerado. Desta vez desconheço os motivos e não possuo nenhuma suposição para o motivo deste problema ter ocorrido.

Apesar das inúmeras pesquisas em fóruns não consegui encontrar uma solução que contornasse meu problema, portanto devido aos fatores explicitados acima, minha análise de desempenho foi realizada sem considerar os dados gerado pelo gprof. Então da análise gprof não é possível inferir nada.

5.6 USO DE MÉTODOS SOFISTICADOS

Para a realização deste trabalho forem implementados somente métodos de ordenação e pesquisa sofisticados. Foram usados a pesquisa em Hash com tratamento de colisões com árvore binaria de pesquisa e para ordenação foi usada a árvore binaria de pesquisa também, dado que ela é derivada da árvore binaria tradicional.

Mesmo que tenhamos feito uso destes métodos era possível realizar o mesmo trabalho só que recorrendo a outros métodos de ordenação e pesquisa sofisticados. Realizarei uma breve comparação dos métodos, analisando suas possíveis vantagens e desvantagens. Considerarei os métodos de QuickSort, HeapSort e MergeSort como métodos de ordenação e o Hash e pesquisa em árvore binaria balanceada.

Iniciarei falando dos métodos de ordenação. A maior vantagem de usá-los é que eles possuem o caso médio com um custo relativamente baixo em média O(n log n), o que em comparação aos outros métodos de ordenação é extremamente ágil e eficiente visto que eles em média têm custo O(n) ou $O(n^2)$. Além de que os métodos de ordenação seguem os princípios de boas práticas de programação visto que há uma alta modularização de código. As desvantagens que ele pode possuir são duas, alguns desses métodos possuem o pior caso similar ao funcionamento dos métodos mais simples e a implementação deles é complexa e delicada, o que faz com que pequenos erros na programação resultem em efeitos inesperados.

Os métodos de ordenação sofisticados. Suas vantagens são similares aos de ordenação tendo um baixo custo computacional. O caso médio de ambos é O(n log n), muito menor que o de pesquisa sequencial que O(n), outro grande benefício é a simplicidade na implementação, manutenção e alteração deles. A desvantagem desses métodos está na tabela Hash, o tratamento de colisões, estas que ocorrem com frequência e necessitam de um tratamento eficiente e que não altere drasticamente o custo, tratamento este que implementamos neste trabalho, onde as colisões foram adicionadas a nós de uma árvore binaria.

Portanto diante, desses pontos é indubitável que o uso destes métodos sofisticados é essencial para projetos complexos. Seu uso se faz quase que indispensável e necessário, pois com o avanço da computação a necessidade de ordenação e busca possui um alto escalonamento e ficar preso a métodos pouco eficiente e simples é um passo na direção oposta a evolução. As desvantagens deles são meros empecilhos na eficiência que eles proporcionam.

6. Conclusão

Após a implementação do programa foi notório que ele pode ser visto de 3 momentos diferentes. A inserção dos e-mails nas posições corretas da tabela e na árvore, a consulta na posição correta da tabela e o apagamento do e-mail.

O grande ponto central deste trabalho era o de conseguir implementar funções e realizar estes procedimentos de forma simples e objetiva, e que ao final da execução resultasse numa saída correta. Com um enfoque maior na implementação da tabela Hash com tratamento de colisões usando uma árvore binaria, esta arvore que deve armazenar os e-mails 1 por nó sendo usada no tratamento de colisões, a pesquisa nela e realizada pelo id do destinatário e mensagem. O uso de técnicas de estruturas de dados foi imprescindível para a execução deste trabalho, pois sem ela a implementação seria catastroficamente complexa e resultaria num programa pouco funcional, dado a complexidade dos métodos usados para a criação dele. Pode-se notar que o custo para executar o programa não foi tão alto, parte deste custo computacional não ser elevado se deve ao fato que por usarmos uma árvore binaria somente teremos custo alto se inserimos os dados crescentemente sendo extremamente difícil de se ocorrer por estarmos usando um método de Hashing sendo o impacta no percorrimento e usabilidade, já que não precisamos percorrer todas as árvores para se encontrar o elemento, basta somente realiza a pesquisa na árvore guarda na posição da tabela que o resultado do mod gerou.

Portando é notório que a boa implementação e domínio das estruturas de dados, e a análise de custo são essenciais para obtermos um programa o mais próximo do ideal, que realize o que é necessário de maneira requisitada, possua um custo computacional o mais baixo possível e seja simples de se entender. Requisitos os quais foram implementados no TP3. Além dele explicitar explicitamente que o uso do Hash deixa a pesquisa extremamente mais eficiente por isso é deliberadamente usado, pois é eficiente e com um alto volume de dados se distancia por magnitudes de medida de desempenho de outras formas de pesquisa, por não possuir uma entrada de alto tamanho não pude realizar este teste de eficiência com dezenas de milhares de comando de inserção, consulta e exclusão.

Portanto inferimos que as estruturas de dados são de extrema importância assim como os métodos de ordenação, e que a implementação dos mesmos precisa ser

realizada cuidadosamente, pois a qualquer erro podemos ter um programa onde tem uma péssima eficiência e alto custo, o que não é desejado.

7. Bibliografia

Ziviani, N. (2006). *Projetos de Algoritmos com Implementações em Java e C++: ˜ Capítulo 3: Estruturas de Dados Básicas'.*Editora Cengage.

8. Instruções para compilação e execução

A compilação do código é feita através do Makefile, para a complicação usa-se o make que compila o programa e o executa, o comando make clean exclui todos os arquivos da extensão “.o”, executáveis e saídas geradas pelo programa.

Para se poder executar o programa é necessário haver uma pasta no mesmo diretório onde o programa se encontra nomeadas entradas e saídas, sendo que inicialmente deve haver arquivos somente na primeira. Arquivos estes na extensão “txt”, que contenham os dados de entrada no formato especificado. Logo após adicioná-los a essa pasta basta alterar a linha de comando responsável por ele no Makefile e executá-lo que as saídas serão geradas na pasta saída.

Para executar a análise de desempenho basta apenas adicionar o comando “-l” no momento da execução.