TRABALHO PRÁTICO 3
SERVIDOR DE E-MAILS

Tarcizio Augusto Santos Lafaiete

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

tarcizio-augusto@hotmail.com

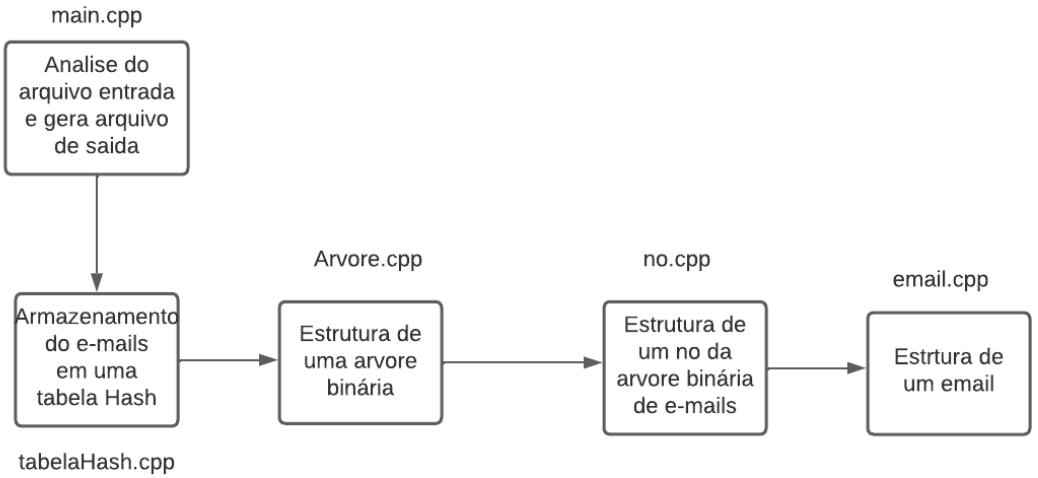
1.Introdução

O problema proposto para implementação do trabalho prático 3 foi o desenvolvimento de um programa para gerenciamento de um novo servidor de e-mails para o Google que seja capaz de efetuar as operações de entrega, consulta e exclusão de e-mail. O programa que realiza estas operações recebe um arquivo de entrada no qual em cada linha há um comando de alguma destas operações para ser realizada no servidor e então ele gera como saída um arquivo com os logs de cada operação. Para o armazenamento das mensagens foi desenvolvida uma tabela hash em que cada posição possui uma árvore binária.

Esta documentação inicia na seção 2 explicando as decisões de implementação tomadas pelo aluno e explicações do funcionamento básico das classes implementadas. Em seguida na seção 3 as informações da seção 2 são complementadas trazendo as estratégias de robustez usadas pelo desenvolvedor do programa para tratar exceções. Na seção 4 tratamos da complexidade espacial e temporal de diversos métodos implementados. Assim, após finalizar a explicação do código, na seção 5 é falado sobre as análises experimentais feitas em relação a este. Por fim, na seção 6 conclui a documentação com as considerações finais do autor e no apêndice é explicado o processo de compilação do código.

2.Implementação

Para a elaboração do servidor de e-mail solicitado neste trabalho prático, primeiro foi-se planejado como seria o fluxo de informações partindo do arquivo de texto com as operações a serem executadas até a saída com o log das operações:



Tendo em mente este diagrama, além do programa principal foram criadas quatro classes,sendo elas a estrutura base de um e-mail, uma árvore binária e seus nós para guardar os e-mails e uma classe com uma tabela Hash para o armazenamento das mensagens do servidor.

2.1 Estruturas de dados

2.1.1 Árvore Binária

Uma árvore binária é uma estrutura de dados baseado no processo de ramificação das árvores na vida real. Estas árvores têm como parte mais fundamental os nós, eles se caracterizam por guardar um elemento e possuir dois ponteiros apontando para outros dois nós, estes ponteiros são comumente denominados como nó à esquerda e nó à direita e por padrão o nó à esquerda é sempre menor que o nó principal e o nó à direita é sempre maior. O primeiro nó da árvore é denominado como raiz e a partir deste nó são feitos os caminhamentos e outras operações dentro da árvore.

Os métodos de inserção, exclusão e pesquisa implementados nesta TAD foram elaborados seguindo a um princípio fundamental do caminhamento em árvores que é ir a partir da raiz verificando se a chave inserida no método é maior ou menor que a chave do nó atualmente comparado caso seja maior a chave é enviada para ser comparada com o nó à direita do nó comparado e caso seja menor é enviada para a comparação com o nó à esquerda, isso ocorre até que não haja mais nós para serem comparados, ou o objetivo do método tenha sido realizado.

2.2 Estrutura de Busca

2.2.1 Tabela Hash

Uma tabela Hash é uma estrutura de dados com enfoque para busca, seu princípio é o uso de uma função Hash que mapeia entradas de valores variados para valores contidos dentro de uma faixa fixa conhecida m. Estes valores fixos se tornam posições em um vetor de m posições em que o elemento é armazenado na posição equivalente ao resultado da sua função Hash.

Geralmente dentro de uma tabela é comum que a função Hash gerar como resultado uma mesma posição para dois valores diferentes, este fenômeno é denominado colisão e para o seu tratamento foi adotado a técnica de encadeamento na qual cada posição da tabela Hash possui uma árvore binária para armazenar os diferentes elementos que podem ser alocados dentro desta posição.

2.3 TAD E-mail e Main

2.3.1 Classe e-mail

Esta classe é bem simples e é utilizada para representar um e-mail, nela além de uma string contendo a mensagem enviada, temos inteiros para identificar o destinatário do e-mail, o identificador único do e-mail e o número de palavras da mensagem. Dentro desta classe, além do construtor, foram desenvolvidos métodos de get e set para as variáveis anteriormente descritas.

2.3.2 Main

Finalizando este subtópico, temos no nosso programa principal a análise dos argumentos passados no momento da execução do programa, a partir desta análise é criada uma tabela hash conforme o tamanho especificado no arquivo de entrada passado como argumento em seguida são lidas as linhas contendo as operações exigidas para o servidor, e a partir da palavra de comando de cada operação é realizado a requisição especificada para o servidor. Como resultado de cada operação executada no servidor é escrito uma linha de log,conforme as especificações do trabalho prático, em um arquivo de saída passado como argumento.

2.3 Dados do ambiente

Este programa foi desenvolvido em C++, compilado pelo G++ da GNU Compiler Collection em computador com o processador Intel Core i3-6006 U, com 4GB de memória RAM e o Linux Mint 20.2 como sistema operacional.

3.Estratégias de Robustez

As estratégias de Robustez implementadas no código tiveram como foco a análise dos argumentos passados no momento da execução do programa e sobre os limites estabelecidos para o identificador de e-mail e usuário e os limites de palavras por mensagem.

No primeiro caso, no momento da passagem de parâmetro caso o argumento -i ou -o não for passado o programa irá fechar instantaneamente e irá avisar a falta do parâmetro do arquivo de entrada ou de saída por mensagem no terminal. Já nos casos em que os identificadores ou número de palavras da operação extrapolam o limite estabelecido, a operação em questão é ignorada pelo programa.

4.Análise de Complexidade

As análises de Complexidade do programa serão dividas em partes, sendo então analisada a complexidade de cada método das classes até chegarmos na análise de complexidade da main.

4.1 Classe Email

Na classe email os únicos métodos são métodos de get e set, sendo assim todos os seu métodos possuem complexidade temporal O(1) e complexidade espacial O(1).

4.2 Classe ÁrvoreEmail

insereEmailRecursivo - complexidade de tempo: Este método tem complexidade temporal O(log n) no melhor caso quando a árvore está balanceada, já quando a árvore está completamente desbalanceada ele é O(n).

insereEmailRecursivo - complexidade espacial: Ele possui complexidade espacial O(log n) no melhor caso devido ao espaço alocado na pilha recursiva indo da raiz até uma das folhas da árvore, já no pior caso é O(n), pois a árvore irá se assemelhar a uma lista encadeada sendo assim será necessário alocar n elementos na pilha recursiva para chegar a folha da árvore.

insereEmail - complexidade de tempo: Este método tem complexidade temporal O(log n) no melhor caso e O(n) no pior caso, devido a chamada da função insereEmailRecursivo.

insereEmail - complexidade espacial: Ele possui complexidade espacial O(n) no pior caso e O(log n) no melhor caso, pois o espaço alocado por este método é o espaço alocado pelo método insereEmailRecursivo.

consultaEmailRecursivo -complexidade de tempo: Este método possui complexidade O(log n) no melhor caso, já que ele consulta um item de cada um dos log n níveis da árvore até achar o elemento pesquisado, e terá pior caso O(n) quando a árvore estiver completamente desbalanceada.

consultaEmailRecursivo - complexidade espacial: Este método possui complexidade espacial O(log n) devido a alocar um elemento na pilha recursiva a cada uma das log n chamadas da função.

consultaEmail - complexidade de tempo: Este método possui complexidade temporal O(log n) no melhor caso e O(n) no pior caso, uma vez que ele tem como operação mais relevante a chamada da função consultaEmailRecursivo.

consultaEmail - complexidade espacial: Este método tem complexidade espacial O(log n), já que ele aloca como memória auxiliar espaço para a pilha recursiva do método consultaEmailRecursivo.

removeEmailRecursivo - complexidade de tempo: Este método tem complexidade temporal O(log n), pois realiza um caminhamento pelos log n níveis da árvore até encontrar o item a ser removido e terá pior caso O(n) com a árvore desbalanceada.

removeEmailRecursivo - complexidade espacial: Ele possui complexidade espacial O(log n) devido a alocação de um elemento na pilha recursiva a cada uma das log n recursões realizadas.

removeEmail - complexidade de tempo: Este método possui complexidade temporal igual a O(log n) no melhor caso e O(n) no pior caso devido a chamada da função removeEmailRecursivo.

removeEmail - complexidade espacial: A alocação de mémoria realizada por este método é devido ao uso do método removeEmailRecursivo, sendo assim sua complexidade espacial é log n.
antecessor - complexidade de tempo: Este método tem complexidade temporal O(log n) devido a realizar um caminhamento dentro de uma das subárvores da árvore binária principal e caso a árvore esteja totalmente desbalanceada sua complexidade se torna O(n) no pior caso.
antecessor - complexidade espacial: Ela possui complexidade espacial O(log n), pois aloca na pilha de recursão espaço para cada uma das chamadas recursivas feitas, sendo então um método que realiza o caminhamento a recursão ocorrerá log n vezes.

4.3 Classe TableHash

pesquisa - complexidade de tempo: Este método tem complexidade de tempo O(log n) no melhor caso e O(n) no pior caso, uma vez que a tabela hash tem como operação mais custosa a chama da função consultaEmail.
pesquisa - complexidade espacial: Ela possui complexidade de espaço O(log n), já que ele aloca espaço para a execução da função recursiva consultaEmailRecursivo.
remove - complexidade de tempo: Este método tem complexidade temporal O(log n) no melhor caso e O(n) no pior caso devido a chamada da função removeEmail que possui está complexidade.
remove - complexidade espacial: Ele possui a complexidade de espaço O(log n), pois ela precisa alocar espaço para o método removeEmailRecursivo que aloca log n espaços na pilha recursiva.
insere - complexidade de tempo: O método insere tem complexidade de tempo O(log n) no melhor caso e O(n) no pior caso, uma vez que ele tem como operação mais custosa a chamada do método insereEmail.
insere - complexidade espacial: Ele possui a complexidade de espaço O(log n), devido a alocação de memória auxiliar para o método recursivo insereEmailRecursivo.
hash - complexidade de tempo: O método de hash tem complexidade de tempo O(1), já que ele possui apenas operações de valor constante.
hash - complexidade espacial: O método de hash tem complexidade de espaço O(1), pois ele sempre aloca na memória auxiliar o mesmo espaço.

4.4 Main.cpp

main - complexidade de tempo: No programa principal temos uma complexidade O(n log n), uma vez que ele possui a chamada o parseArgs de complexidade O(1), em seguida é iniciado um loop lendo cada uma das n operações descritas no arquivo de entrada, cada operação requisitada ao servidor de emails tem um custo de O(log n), sendo assim temos n operações de custo log n ficando então a complexidade da main como O(1) + O(n) O(log n) = O(n log n).
main - complexidade espacial: O programa principal tem complexidade espacial O(log n), pois os métodos da classe tabelaHash que são chamados dentro do

programa principal possuem todas O(log n) como complexidade espacial e fora elas todas as outras operações possuem custo constante .
parseArgs - complexidade de tempo: Esta função tem complexidade temporal O(1), uma vez que o loop presente nela repete constantemente 2 vezes que é o valor passado no argumento argc dentro da função main.
parseArgs - complexidade espacial: Sua complexidade espacial é O(1), pois o espaço alocado pela função é constante e independente da entrada.

5.Análise de Experimentos

5.1 Desempenho computacional de tempo

Os experimentos realizados para medir o desempenho computacional do analisador de texto foram feitos utilizando a biblioteca de c, time.h, com ela criamos duas variáveis start e end do tipo clock_t. O clock_t é uma estrutura de dados implementada pela biblioteca time.h, no qual as variáveis deste tipo são capazes de guardar o número de sinais de clock utilizados pelo processador até o momento pelo programa. Então a start foi chamada logo antes do início do algoritmo implementado na main e a end no final do código.

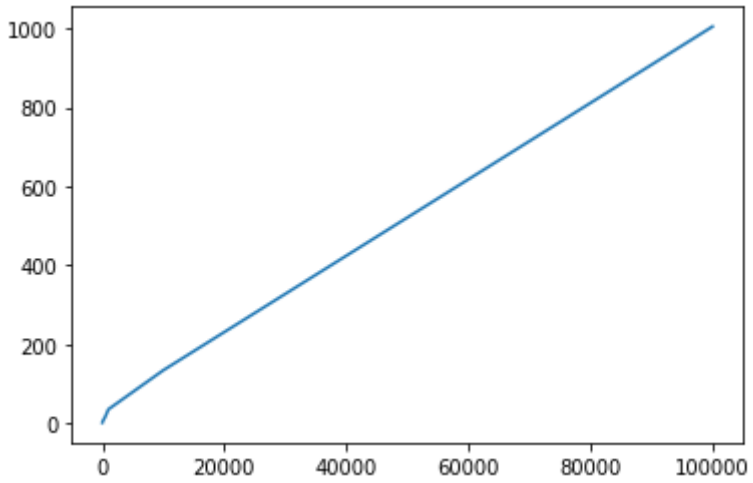
Com esses valores armazenados, fazemos a diferença deles e dividimos por CLOCKS_PER_SEC que é um define da biblioteca time.h que representa o valor do número de clocks que o processador gera por segundo. Dessa divisão então conseguimos o tempo de execução do programa.

Explicado como foram feitas as medições de tempo, agora pode-se detalhar como e em quais condições foram feitas as medidas. Todas as medidas tiveram como padrão um tamanho de mensagem igual a 15 caracteres e uma frequência de de um comando APAGA e um comando CONSULTA a cada dois comandos ENTREGA.

Deste experimento retiramos duas tabelas uma em que fixou-se apenas uma mensagem por usuário e outra que fixamos 1 usuário para n mensagens. Assim teve-se o seguinte resultado :

Usuários	Medida 1	Medida 2	Medida 3	Medida 4	Medida 5	Média
10	1,762 ms	1,301 ms	1,501 ms	1,332 ms	1,517 ms	1,4826 ms
100	5,917 ms	7,082 ms	6,354 ms	4,464 ms	6,171 ms	5,9976 ms
500	7,911 ms	18,79 ms	19,582 ms	18,897 ms	20,12 ms	17,06 ms
1000	47,122 ms	30,425 ms	44,384 ms	33,729 ms	20,762 ms	35,2844 ms
10000	118,5 ms	138,747 ms	142,976 ms	140,592 ms	130,528 ms	134,2686 ms
100000	756,485 ms	1,13398 s	1,13645 s	910,305 ms	1,080107 s	1,004654 s

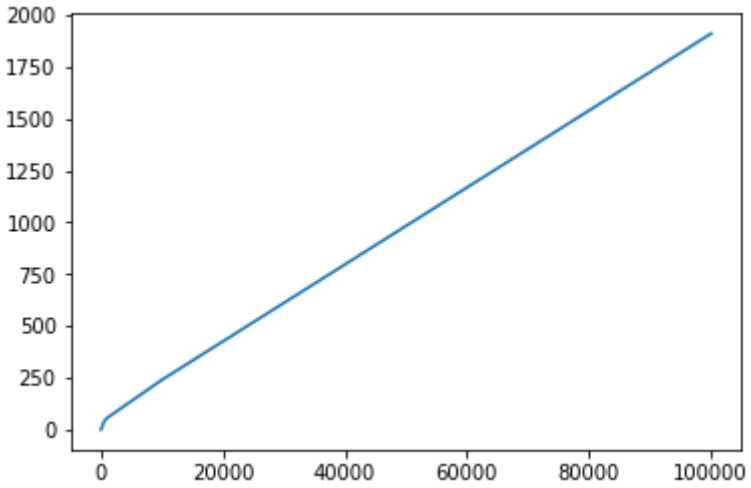
Utilizando então os dados apresentados na tabela e as funções da biblioteca matplot da linguagem python, criamos um gráfico com o número de usuários como eixo x e a média de tempo obtida como eixo y. Assim tivemos este gráfico:



Observando este gráfico percebemos um comportamento linear na resposta do algoritmo em relação a variação do número de usuários, isso pode ser explicado pelo fato de que as mensagens possuem sempre o mesmo tamanho, ou seja sempre tem o mesmo custo computacional e que além disso, como há apenas uma mensagem por usuário não haverá o custo do caminhamento pelas árvores binárias da tabela Hash tornando então as suas operações com custo O(1), logo a úncia coisa que irá variar o tempo de execução do programa será o próprio processo de computação do arquivo de entrada que é O(n), ou seja, tem custo linear.

Mensagens	Medida 1	Medida 2	Medida 3	Medida 4	Medida 5	Média
10	487 us	1,482 ms	1,700 ms	1,031 ms	368 us	1,0136 ms
100	2,687 ms	2,268 ms	2,392 ms	7,187 ms	7,102 ms	4,3272 ms
500	33,323 ms	45,005 ms	31,26 ms	37,79 ms	46,492 ms	38,774 ms
1000	38,593 ms	70,24 ms	73,753 ms	76,826 ms	25,089 ms	56,9002 ms
10000	179,837 ms	179,19 ms	267,859 ms	291,018 ms	286,663 ms	240,9134 ms
100000	1,763 s	2,248 s	1,853 s	1,93 s	1,76 s	1,9108 s

Novamente usando o matplot do python criamos um gráfico como o número de mensagens como eixo x e a média do tempo de execução como eixo y assim obtivemos o seguinte gráfico:

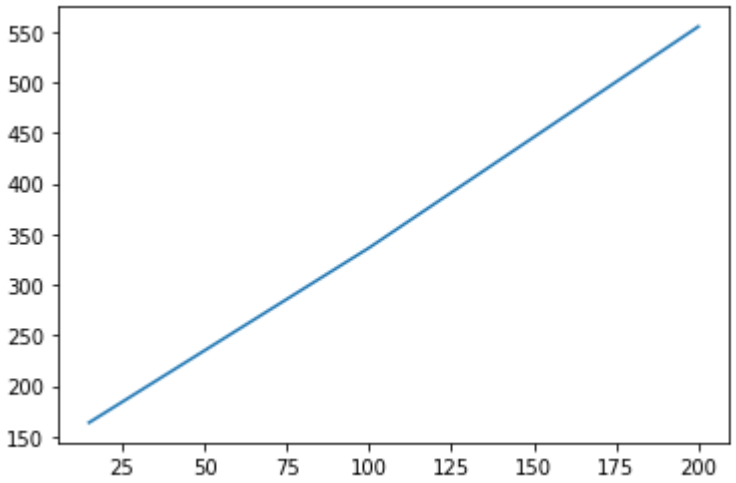


Observando este gráfico também nota-se um comportamento semelhante ao do primeiro gráfico, contudo pode-se perceber que os valores de média obtidos nesta segunda tabela destoam dos valores da primeira, isso se deve pelo fato de que agora como há apenas um usuário recebendo n mensagens, o custo das operações da tabela Hash deverão considerar o custo dos processos de caminhamento dentro da árvore binária, logo tendo custo O(log n). Então, combinando o custo log n da árvore com o custo da computação do arquivo de entrada, que é O(n) como dito anteriormente, teremos O(n log n), que apesar de possuir um comportamento próximo ao de uma progressão linear ainda tem um custo maior.

Além dessas análises foram feitas análises em relação ao efeito gerado pelo número de palavras da mensagem dentro do programa, para isso testamos a operação ENTREGA de 100 e-mails para 100 usuários e tivemos o seguinte resultado:

Palavras	Medida 1	Medida 2	Medida 3	Medida 4	Medida 5	Média
15	140,718 ms	129,891 ms	170,128 ms	188,908 ms	188,648 ms	163,6586 ms
100	327,02 ms	299,049 ms	360,929 ms	357,845 ms	337,026 ms	336,3738 ms
200	534,363 ms	582,409 ms	558,035 ms	545,464 ms	555,018 ms	555,0578 ms

Utilizando mais uma vez o matplot do python com as médias de tempo como eixo y e o número de palavras como eixo x gerou-se o seguinte gráfico:



Com este gráfico conseguimos perceber uma progressão linear no tempo de execução a medida em que se aumenta o número de palavras em uma mensagem, o que é esperado já é esperado uma vez que o processo de leitura da mensagem dentro do arquivo de entrada é feito através de um loop que percorre as n palavras do texto, assim sendo esta componente tem ordem de complexidade O(n), ou seja é linear.

Por último, foi feita uma análise sobre o efeito da frequência das operações APAGA e CONSULTA para o tempo de execução do programa, para isso foram feitas cinco medições para cada proporção analisada e assim foi gerada esta tabela:

Proporçao	Consulta	Apaga
1/2	226,372 ms	205,285 ms
3/4	229,136 ms	232,49 ms
1	269,922 ms	241,034 ms

Nota-se então que na média o tempo de execução do programa não sofre grandes efeitos devido a frequência das operações CONSULTA e APAGA, isso se dá pelo fato de que ambos os métodos possuem ordem de complexidade log n, ou seja, o seu tempo de execução para proporções semelhantes deve ser relativamente parecido.

7.Conclusão

Neste trabalho foi desenvolvido um software que analisa um arquivo de entrada, na qual contém um conjunto de operações que devem ser executadas em um servidor de emails e a partir deste documento deve-se executar as operações e gerar um arquivo de saída com os logs das operações. Durante a execução do trabalho grandes aprendizados foram adquiridos tanto pelos erros, quanto pelos acertos durante o processo. Além do óbvio ganho de prática com o desenvolvimento de programas do zero.

Com o trabalho o autor pode adquirir mais prática e conhecimento sobre o uso dos algoritmos de pesquisa. Importante ressaltar as dificuldades iniciais com o entendimento do método de remoção dentro da árvore binária, uma vez que este método talvez tenha o algoritmo mais complexo dentre os métodos da árvore binária implementada, além disso o processo de mescla de um hashing com um árvore binária de pesquisa foi extremamente interessante e assim como o trabalho anterior trouxe ao ao autor uma visão mais completa sobre as estruturas e algoritmos estudados durante o semestre.

Apêndice

Instruções de compilação

- Abra o dirétoria TP no terminal linux ou prompt comando do Windows
- Realize o comando make all
- Entre na pasta bin
- Coloque o arquivo de entrada dentro desta pasta
- Realize a linha de comando para execução do programa conforme o padrão de passagem de argumentos descrito nas especificações deste trabalho prático.