Documentação - Trabalho Prático 3

Disciplina: Estrutura de Dados 2022/1
Aluno: Gustavo Tavares Corrêa

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

1. Introdução

O trabalho consistiu na implementação de um servidor de e-mails simulado, com funções de entrega, consulta e remoção de mensagens. Tal projeto foi efetuado utilizando tipos abstratos de dados e o uma tabela de hash, de modo a praticar os conteúdos vistos na disciplina.

2. Implementação

O programa foi desenvolvido na linguagem C++, utilizando o compilador g++. O código apresenta três arquivos .hpp na pasta include, message.hpp, BST.hpp e hashTable.hpp, todos arquivos de definição das classes utilizadas no programa. A pasta src possui os arquivos .cpp: message.cpp, BST.cpp, hashTable.cpp e main.cpp. Os três arquivos .hpp estão associados com a implementação dos respectivos arquivos de mesmo nome, porém extensão .cpp, e o arquivo main.cpp lida com a entrada e saída do projeto por meio do uso das estruturas implementadas. A classe BST corresponde a Binary Search Tree, e as árvores compostas de mensagens formam a tabela de hash. As pastas bin e obj ficam inicialmente vazias, mas após o uso do make, guardam o executável e os objetos utilizados para o criar, respectivamente. Por fim, há também um arquivo makefile na pasta do TP.

A classe mensagem serve tanto como nó para a árvore binária de busca quanto para guardar os elementos do email que se quer associar ao nó. Logo, ela possui um identificador único, o destinatário, o número de palavras do texto enviado e o próprio texto, além de dois ponteiros para os nós esquerdo e direito associados a ela, outras mensagens.

A classe BST corresponde à implementação básica de uma árvore binária de busca com funções de inserção, remoção e pesquisa e usando elementos da classe message como nós com os e-mails. A função de remoção tem uma modificação em relação ao modelo clássico por não ser do tipo void, e sim do tipo int, ela retorna 1 caso a operação tenha sido um sucesso e 0 caso não tenha, tendo em vista que é importante diferenciar os casos no sistema de e-mails. Para verificar a adequação das buscas e da remoção, ao achar o identificador requisitado, é feita uma checagem adicional para ver se o usuário na chamada de função também é o pedido, no caso da busca, dentro da main, e na remoção, dentro da própria chamada de função. Além disso, a busca retorna uma mensagem padrão, com identificador inválido -1, caso não encontre um e-mail requisitado, o que é checado ao executar a impressão dos resultados.

A classe hashTable implementa uma tabela de hash em que cada entrada da tabela está associada a uma árvore binária de busca, a qual corresponde à uma caixa de entrada e a tabela completa é o servidor de e-mails. A tabela é composta de um vetor estático de árvores binárias e possui funções de inserção, consulta e remoção, as quais calculam a caixa de entrada correspondente ao usuário e chamam a função equivalente para a árvore binária dessa caixa.

O arquivo main.cpp lê os nomes dos arquivos de entrada e saída da linha de comando, os abre no programa, cria um servidor de e-mails (tabela hash) e faz os ajustes necessários nas entradas lidas para que possam ser passadas e funcionarem nas funções, seja criando um e-mail a ser entregue no servidor, seja pegando o usuário e o identificador de um e-mail a ser removido ou consultado, de acordo com o comando lido. Além disso, ele imprime no arquivo de saída os resultados correspondentes às chamadas.

3. Análise de Complexidade

A complexidade do programa pode ser dividida em duas análises, tempo e espaço, e serão utilizados o parâmetro n para representar o número de e-mails presentes em um determinado momento dentro do servidor da tabela hash e o parâmetro k para representar o número de caixas de entrada.

Partindo do pressuposto de que a distribuição dos elementos na tabela seja boa, igualitária, com a probabilidade de um elemento cair em uma caixa ser aproximadamente a mesma para cada uma das caixas, a complexidade irá diminuir. Há um compromisso entre o espaço disponibilizado para a tabela e o tempo necessário para realizar as operações de consulta, entrega e remoção, quanto maior o espaço disponibilizado, menor o tempo necessário, pois cada caixa passa a corresponder a uma árvore de tamanho menor. Entretanto, pode-se também acabar possuindo várias caixas vazias e inutilizadas, desperdiçando espaço.

A complexidade temporal corresponde a O($log(n/k)$) operações no caso médio de entrega, remoção ou pesquisa nas árvores binárias, dada uma distribuição boa dos elementos. Caso a distribuição não seja boa, e de forma mais geral, a complexidade é O(log(n)) no caso médio, O(1) no melhor caso, em que há somente um elemento na árvore, e O(n) no pior caso, em que a árvore é degenerada e torna-se uma lista encadeada.

A complexidade espacial do programa corresponde a O(n) espaços, correspondentes aos n e-mails entregues no servidor. Cada e-mail é composto de três inteiros, uma string e dois ponteiros para outros e-mails, logo há uma complexidade implícita no tamanho dos elementos da classe e-mail.

Em termos de acesso de memória, uma análise teórica sugere que eles são feitos de maneira linear no uso das estruturas, pois deve haver uma proximidade de localidade de referência entre caixas de e-mail posicionadas próximas no servidor e também entre os nós presentes em cada uma delas.

4. Estratégias de Robustez

A robustez do código foi implementada majoritariamente para lidar com possíveis problemas nos parâmetros de entrada, para evitar acessos de posições inválidas, como a posição 1 em uma string i ao invés de -i, ou caso um -i ou -o seja passado no final do arquivo sem um próximo elemento, e também para evitar que um dos arquivos de entrada ou saída não seja inserido. Além disso, foi implementada uma estratégia para lidar com comandos de entrada que não sejam reconhecidos, distintos de “ENTREGA”, “CONSULTA” ou “APAGA”.

5. Testes

Os testes gerais foram feitos utilizando entradas de 1000, 2000, 5000 e 10000 comandos. Além disso, há testes analisando as mudanças nos parâmetros número de usuários, número de mensagens, tamanho das mensagens e distribuição de frequência de operações. O gerador de carga utilizado para isso foi o feito por mim e disponibilizado no fórum do Moodle. No caso específico do parâmetro número de usuários, o que é alterado é o número máximo deles, tendo em vista que o gerador utiliza números aleatórios ao definir quais usuários estão associados aos comandos. O número de palavras em cada mensagem também é aleatório, mas nunca passa de 200, de acordo com as especificações do TP, por mais que o tamanho individual de cada palavra seja fixo.

6. Análise Experimental

Resultados de tempo:

Os resultados de tempo foram obtidos a partir da análise dos arquivos de log.txt gerados pelo uso do memlog na aplicação do TP3 sem o uso da função ativaMemLog, usando apenas iniciaMemLog e finalizaMemLog.

Primeiro caso de testes - comandos aleatórios, máximo de 10000 usuários, palavras de tamanho 1:

Os resultados apontam para uma progressão linear quando o número de comandos aumenta e o tamanho da tabela de hash se mantém. Já o aumento no tamanho da tabela de hash não apresenta mudanças significativas nos resultados de tempo.

(Tempo em segundos)		Número de comandos			
		1000	2000	5000	10000
Tamanho da tabela de hash	10	0.1668	0.3359	0.8133	1.5807
	100	0.1795	0.3775	0.8274	1.5654
	500	0.1706	0.3358	0.7937	1.6114
	1000	0.1728	0.3282	0.8602	1.5978

Segundo caso de testes - comandos específicos, máximo de 10000 usuários, palavras de tamanho 1:

Para essa rodada de casos testes, preferi utilizar dois extremos e um caso médio nos valores dos parâmetros.

Apenas inserções:

Os resultados foram extremamente semelhantes aos com comandos variados.

(Tempo em segundos)		Número de comandos		
		1000	5000	10000
Tamanho da tabela de hash	10	0.1805	0.8575	1.6842
	100	0.1749	0.8930	1.6462
	1000	0.1783	0.8240	1.5049

Apenas remoções / apenas exclusões:

Ambos podem ser analisados simultaneamente, pois como não há elementos inseridos, quaisquer chamadas de ambos comandos imediatamente retornam erro. Os resultados corresponderam aos de comandos aleatórios, o que diverge da minha teoria original de que seria mais rápido, por não ter que andar nas árvores.

(Tempo em segundos)		Número de comandos		
		1000	5000	10000
Tamanho da tabela de hash	10	0.1663	0.7620	1.6336
	100	0.1587	0.8271	1.6042
	1000	0.1727	0.7785	1.5808

Terceiro caso de testes - 100 servidores, 1000 comandos variados:

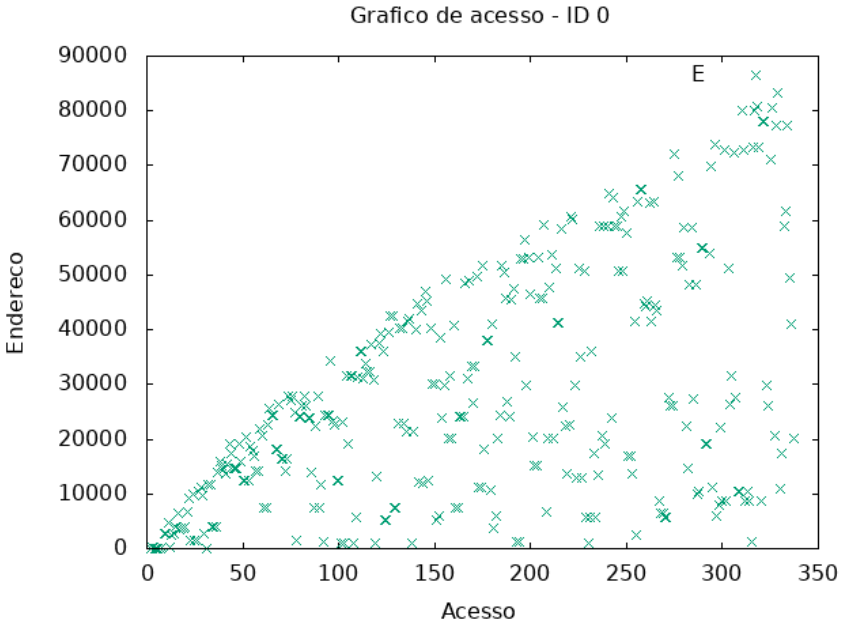
Aumentar o tamanho máximo de usuários gerou uma pequena redução nos tempos necessários, especialmente quando o tamanho das palavras é maior, e aumentar o tamanho das palavras aumentou significativamente os tempos, contudo, com uma proporção menor que linear.

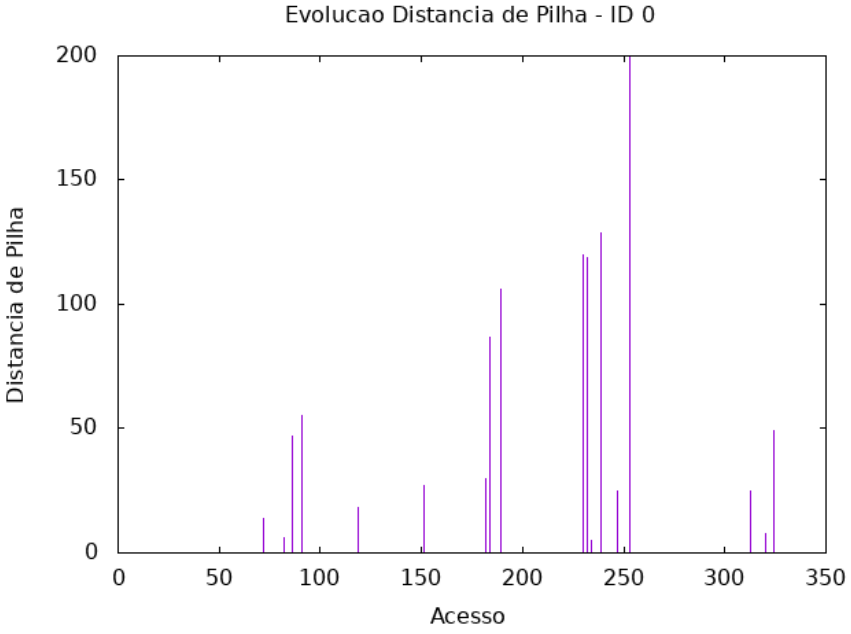
(Tempo em segundos)		Tamanho das palavras		
		10	100	1000
Número máximo de usuários	10	0.1956	0.2598	0.9696
	100	0.1816	0.2304	0.8991
	1000	0.1778	0.2294	0.8232

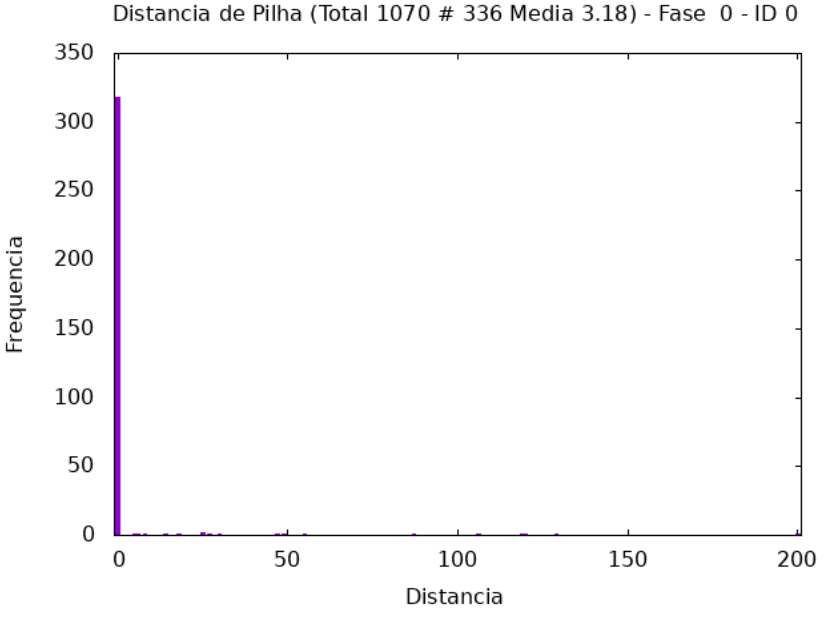
Resultados de espaço:

Os resultados de espaço foram obtidos a partir da análise dos arquivos de log.out gerados pelo uso do memlog na aplicação do TP2 com o uso da função ativaMemLog. A entrada considerada foi tamanho de tabela 10, 1000 comandos, tamanho de palavras 10 e máximo de 50 usuários. A escolha de números pequenos de usuários e tamanho de tabela, em contraste com o grande número de comandos, foi feita para que mais elementos fossem inseridos na mesma árvore, gerando gráficos mais interessantes para análise.

Gráficos de inserção:







Os gráficos permitem concluir que a inserção é bem variada, e certamente não apresenta linearidade, assim como era de se esperar, tendo em vista que são operações em nós de árvores binárias e com exclusões no meio.

7. Conclusões

O trabalho consistiu na criação de um servidor de e-mails com funções de entrega, exclusão e consulta de e-mails. Foram praticados os usos dos algoritmos de pesquisa árvore binária de pesquisa e tabela de hash e analisou-se como alterações nos parâmetros associados a esses algoritmos podem afetar no desempenho final do programa.

8. Bibliografia

Não foi utilizado nenhum recurso além dos ensinos da faculdade.

9. Instruções para compilação e execução

Acessar a pasta TP e utilizar o comando make para gerar o executável “tp3” do trabalho prático no diretório bin e os arquivos *.o no diretório “obj”. Inserir o arquivo de entrada na pasta TP e, se de interesse, um arquivo de saída (esse segundo é gerado automaticamente pelo programa caso não esteja presente) no diretório raiz, TP. A partir disso, o comando ./bin/tp3 -i entrada.txt -o saida.txt, caso se esteja no diretório raiz (pasta TP), deve executar adequadamente os comandos com a entrada solicitada e imprimir os resultados no arquivo de saída. Os arquivos entrada.txt e saida.txt também podem ser alterados de acordo com os interesses de quem executa o código, seja em nome ou no tipo do arquivo.