Trabalho Prático 3: Servidor de Emails

Pedro de Oliveira Guedes
2021040008

DCC/UFMG

Introdução

A documentação aqui presente se refere a um programa produzido na linguagem de programação C + +, com o objetivo de implementar um servidor de e-mails através de uma tabela hash, bem como caixas de entrada através de árvores binárias.

O programa é capaz de realizar três operações:

1. **Envio de e-mails:** Para isso, o usuário deve usar o comando "ENTREGA", informando o identificador do destinatário, identificador do e-mail, quantidade de palavras do e-mail e mensagem do e-mail. As informações devem aparecer na ordem que está aqui documentada.
2. **Consulta de e-mails:** Ao realizar essa operação, o usuário deve enviar o comando "CONSULTA", seguido pelo identificador do usuário que se quer consultar o e-mail e o identificador do e-mail que se quer consultar.
3. **Exclusão de e-mails:** Para realizar essa operação, o usuário deve fornecer o comando "APAGA", informando o identificador do usuário que possui o e-mail que se quer apagar, bem como o identificador do e-mail que se quer apagar.

Mais detalhes sobre a implementação do servidor serão dados no decorrer deste documento.

Método

Nesta seção, serão dadas informações sobre a organização do programa em pastas, bem como o que cada uma abriga, formatos de entrada e saída, configurações do sistema de testes, etc..

Configurações do sistema de testes

O programa foi implementado na linguagem C + +, compilado pelo G + +, compilador da GNU Compiler Collection. Além disso, foi utilizado o sistema operacional Ubuntu 9.3.0-17 ubuntu1~20.04 em um Windows 10, através do Windows Subsystem for Linux (WSL 2), com um processador AMD Ryzen 3 3200G (3.60GHz, 4 CPUs) e 13 GB RAM.

Formato de entrada e saída

O formato de entrada dos dados é através de um arquivo de texto, do formato que seja, seguindo o padrão:

```
<Quantidade de usuários do servidor - número inteiro>
[ OPERAÇÃO <PARÂMETROS> ]
[ OPERAÇÃO <PARÂMETROS> ]
[ OPERAÇÃO <PARÂMETROS> ]
.
.
.
```

Os blocos de operação são utilizados assim como explicado na sessão introdutória dessa documentação.

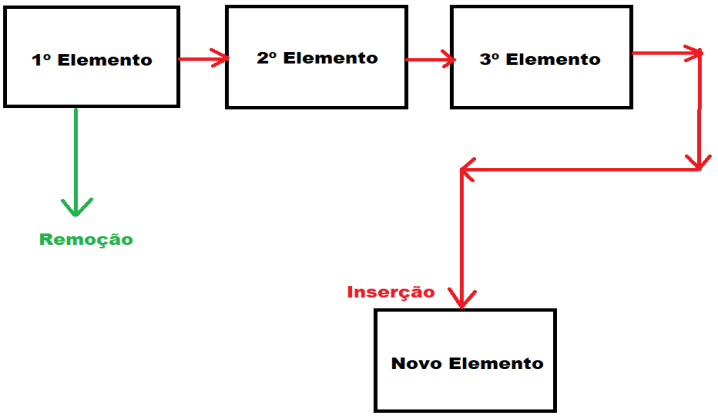
O formato de saída do programa também é através de um arquivo de texto, do formato que for informado pelo usuário, que segue o padrão:

```
[ RESULTADO DA PRIMEIRA OPERAÇÃO ]
[ RESULTADO DA SEGUNDA OPERAÇÃO ]
[ RESULTADO DA TERCEIRA OPERAÇÃO ]
.
.
.
```

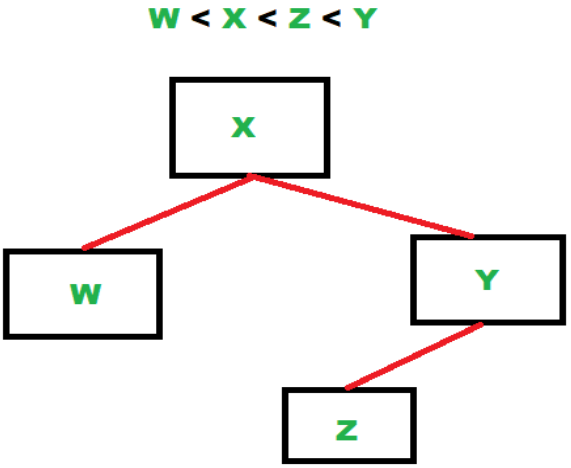
Algoritmos e Estruturas de Dados

Além das estruturas de dados específicas para armazenar dados do problema, como a classe “*Email*” e “*EmailContent*”, foram também implementadas estruturas de dados mais generalizadas para organizar e armazenar as informações.

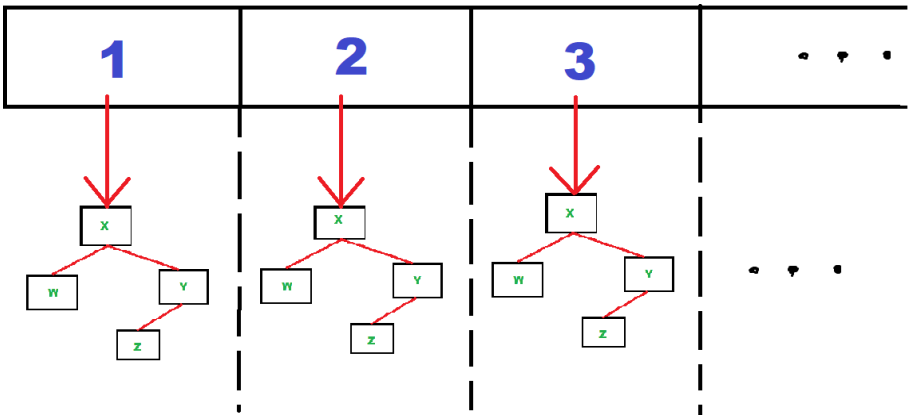
Uma das estruturas utilizadas é a de fila. Ela tem o propósito de armazenar as operações realizadas no servidor, na ordem em que foram solicitadas. Dessa forma, é possível manter um "*log*" com as informações da sessão de utilização do servidor. Abaixo temos uma representação gráfica do funcionamento, onde os elementos são as operações realizadas.



Outra estrutura importante é a caixa de entrada dos usuários. Ela foi implementada como uma árvore de pesquisa binária. Ela permite inserir e-mails de forma ordenada através do identificador dos mesmos, além de pesquisar e excluir estes e-mails pelo mesmo identificador.



Para abrigar as caixas de entrada dos usuários, é necessário um servidor. Este, por sua vez, foi implementado através de uma *tabela hash*, cujo *fator de hashing*, ou módulo da tabela, é a quantidade de usuários que utilizarão o servidor. Uma representação mais gráfica desta estrutura de dados pode ser visualizada a seguir:



Organização de pastas

A pasta raiz do projeto se chama "*TP*" e abriga, além das subpastas diretamente relacionadas ao programa, a subpasta "analisamem", que não será discutida neste documento, já que foi criada pelos professores da disciplina.

Além desta, a pasta também abriga as subpastas a seguir:

1. **bin:** Abriga todos os arquivos binários gerados pelo programa.
2. **include:** Inclui todos os arquivos de cabeçalho utilizados na implementação. Entre eles: *memlog.hpp*, *msgassert.hpp*, emailContent.hpp, email.hpp, *inbox.hpp*, *operations.hpp*, *server.hpp*, *session.hpp*.
3. **obj:** Arquivos de objeto gerados para a compilação do programa principal.
4. **src:** Todos os arquivos de código fonte para a implementação do programa. Sendo eles: *memlog.cpp*, *main.cpp*, emailContent.cpp, email.cpp, *inbox.cpp*, *operations.cpp*, *server.cpp*, *session.cpp.*

Análise de complexidade

A análise de complexidade contemplará as principais funções implementadas nos arquivos de código fonte do programa. As funções que não forem contempladas por esta análise têm, todas, complexidade constante. Para todas as funções, serão consideradas operações relevantes:

- Complexidade de tempo: leitura e atribuição de valores, comparações.
- Complexidade de espaço: alocação de memória dinâmica, recebimento de estruturas dinâmicas e chamadas de funções.

A presente análise de complexidade leva em consideração a notação de complexidade "*grande O*". A título de referência, a quantidade de e-mails armazenados em uma caixa de entrada (Árvore Binária de Pesquisa) será denotada por "*n*". Além disso, a quantidade de operações a serem realizadas em um arquivo será denotada por "*m*".

Inbox::searchEmail(int emailID, int userID):

Esta função busca um e-mail armazenado na árvore binária referente. Como a árvore não está balanceada, o programa pode ter que passar por até "*n*" e-mails antes de encontrar o buscado. Sendo assim, temos complexidade de tempo: $O(n)$. Porém, como esta função não foi implementada recursivamente, teremos complexidade de espaço: $O(1)$.

Inbox::insertEmail(int emailID, int userID, emailContent, Email &email)*:

Esta é uma função recursiva auxiliar para inserir um e-mail numa árvore binária. Ela encontra a posição que o e-mail deve ficar através do atributo "*emailID*". Dessa forma, pode precisar chamar a si mesma até "n" vezes comparando o identificador para encontrar a posição correta. Portanto, teremos complexidade de tempo: $O(n)$ e complexidade de espaço: $O(n)$.

Inbox::addEmail(int emailID, int userID, int wordsAmount, std::string msg):

Esta função define o conteúdo do e-mail e realiza a chamada da função auxiliar estudada acima "*insertEmail*". Dessa forma, como criação de estruturas é uma operação de custo constante e a complexidade da função auxiliar já foi estudada, teremos: complexidade de tempo: $O(n)$ e complexidade de espaço: $O(n)$.

*Inbox::replacePredecessor(Email *deletedEmail, Email* &substitute)*:

Esta função é auxiliar para remover e-mails na árvore que tenham 2 filhos. Ela funciona recursivamente, procurando primeiro o elemento antecessor "*substitute*" ou que se quer excluir "*deletedEmail*". Este antecessor será o elemento mais à direita do filho esquerdo do nó que se quer excluir. Dessa forma, a função poderá acessar recursivamente até "$n - 2$" (um dos excluídos é o próprio e-mail a ser excluído e o outro é o e-mail imediatamente à direita) e-mails antes de realizar a troca, que terá custo constante igual a 1. Portanto, teremos complexidade de tempo: $O(n)$ e complexidade de espaço: $O(n)$.

Inbox::removeEmail(Email &email, int emailID, int userID)*:

Este método é responsável por excluir e-mails cujo identificador foi encontrado na caixa de entrada do usuário. Ele busca recursivamente pelo e-mail, o que leva este método a ter uma complexidade de tempo e espaço originais de $O(n)$. Além disso, após encontrar o e-mail que se quer deletar, há três caminhos que podem ser tomados. Dois deles são para o caso de o e-mail a ser excluído tiver até 1 filho, o que possui complexidade $O(1)$. O outro é para quando o e-mail possui 2 filhos, nesse caso, é chamado o método "*replacePredecessor*", que foi estudado anteriormente e possui complexidade $O(n)$. Dessa forma, é possível dizer que este método possui função de complexidade $f(n) = 2n$, em notação assintótica, teremos complexidades de tempo e espaço iguais a $O(n)$.

Inbox::deleteEmail(int emailID, int userID):

Este método realiza a chamada de "*removeEmail*" que foi estudado anteriormente. Portanto, temos que a complexidade de tempo e espaço deste método é $O(n)$.

Server::sendEmail(int addressee, int emailID, int wordsAmount, std::string msg):

Este método realiza a chamada do método "*addEmail*", que já foi estudado, além de inserir a operação realizada na fila de operações, que tem custo constante igual a 1. Dessa forma, temos complexidade de tempo: $O(n)$ e complexidade de espaço: $O(n)$.

Server::searchEmail(int userID, int emailID):

Este método realiza a chamada do método "*Inbox::searchEmail*", que já foi estudado, além de inserir a operação realizada na fila de operações, que tem custo constante igual a 1. Dessa forma, temos complexidade de tempo: $O(n)$ e complexidade de espaço: $O(1)$.

Server::eraseEmail(int userID, int emailID):

Este método realiza a chamada do método "*deleteEmail*", que já foi estudado, além de inserir a operação realizada na fila de operações, que tem custo constante igual a 1. Dessa forma, temos complexidade de tempo: $O(n)$ e complexidade de espaço: $O(n)$.

Session::Session(std::string sessionInputs):

Este método faz a instanciação de um servidor e, através da leitura do arquivo com as operações a serem realizadas, itera por todas as "*m*" delas chamando uma das funções "*Server::sendEmail*", "*Server::searchEmail*" ou "*Server::eraseEmail*", cujas complexidades já foram estudadas. Dessa forma, teremos complexidade de tempo igual a $O(m * n)$ e de espaço igual a $O(n)$.

Session::finish(std::string sessionLogs):

Este método realiza a impressão dos resultados de operações no arquivo cujo nome foi passado como parâmetro. Como foram feitas "*m*" operações, teremos complexidade de tempo igual a $O(m)$ e complexidade de espaço igual a $O(1)$.

Programa completo:

O programa completo faz a chamada basicamente de duas funções, o construtor da classe *Session* e o método "*finish()*", da mesma classe. Como já foi estudado anteriormente as complexidades de cada uma, podemos afirmar que a complexidade de tempo esperada é: $O(m * n)$ e a de espaço é: $O(n)$.

Obs:

As outras funções não serão analisadas quanto à complexidade porque se tratam de métodos que não são particularmente relevantes para o programa.

Estratégias de Robustez

A robustez em um programa é de crucial importância para o bom funcionamento do mesmo. Nesse sentido, uma série de verificações de entrada do usuário se fazem necessárias, para que não haja execuções errôneas ou estouros grosseiros de erro no programa. Para isso, foi utilizada a biblioteca *msgassert.h*, também presente nos arquivos deste trabalho.

As estratégias serão listadas e explicadas com relação à relevância de utilização.

- **Falha na abertura de arquivos:** Caso algum arquivo informado pelo usuário não exista ou falhe ao ser aberto pelo programa, ele é finalizado e uma mensagem de erro explicando o ocorrido é exibida ao usuário.
- **Mensagem e palavras fora das especificações:** Ao enviar um e-mail, o usuário deve informar a quantidade de palavras que o mesmo possui. Caso a mensagem informada tenha uma quantidade de palavras diferente da que se espera, é mostrado um aviso sobre a situação no console, mas o programa segue normalmente. O mesmo vale para palavras maiores que 40 caracteres, que foi outra premissa informada para o desenvolvimento.
- **Tratamento de colisão na tabela hash:** Assim como já foi informado anteriormente na seção de "*Algoritmos e Estruturas de Dados*", a tabela hash terá, em cada uma das posições, uma árvore binária como caixa de entrada, para armazenar e-mails de usuários cujo código hash colide. Além disso, para evitar que um e-mail destinado a um usuário seja acessado por outro, por estar na mesma caixa de entrada, a estrutura de dados "*Email*" possui também um atributo "*addressee*", que indica o usuário ao qual o e-mail foi destinado originalmente.

Testes

Os testes para o programa implementado serão divididos entre testes de acesso à memória e testes de desempenho computacional. Os logs da sessão no servidor serão armazenados em arquivos texto, assim como os tempos de execução. Mapas de acesso à memória em arquivos de imagem ".png" e o teste *gprof* também em um arquivo de texto.

O teste de memória verifica a performance do programa quanto o acesso à memória alocada pelo mesmo. Será utilizada para metrificar, a ferramenta da "distância de pilha". Basicamente, quando um endereço de memória é acessado, ele é inserido na pilha, quando ele for acessado novamente, ele é retirado da posição da pilha em que estava e volta para a base da mesma. A distância da posição em que o elemento estava ao ser chamado para a primeira posição da fila será a distância de pilha daquela chamada em questão.

A distância de pilha total será calculada pela média ponderada da frequência em que um elemento é desempilhado com aquela distância, sendo que a distância em si é o peso da média. Quanto menor for a distância de pilha, melhor projetado é o código, indicando menor custo de memória.

Os testes de desempenho verificarão o tempo que leva para que uma função seja executada para um determinado tamanho de entrada. Para isso, será marcado o tempo de início da execução do programa e o tempo de finalização do mesmo. A diferença entre estes tempos é então calculada e será o tempo de execução do programa.

Será analisado não somente o tempo individual de execução para diferentes entradas, mas a evolução deste tempo e se ele comprova ou não a análise de complexidade feita em um dos tópicos anteriores.

Análise Experimental

A análise experimental foi dividida entre testes de acesso à memória e testes de desempenho computacional, que serão abordados em tópicos abaixo. Para mais informações sobre os métodos de teste utilizados para esta análise experimental, consulte o tópico "Testes" deste documento.

Desempenho Computacional

Os testes de desempenho computacional foram realizados utilizando arquivos de texto contendo operações a serem realizadas no servidor. Serão realizados 5 testes de desempenho,

sendo que a cada novo teste, será acrescentada uma quantidade fixa de novas operações. O primeiro teste consistirá de 2.000.000 operações, o segundo de 4.000.000, o terceiro de 6.000.000 operações, crescendo em 2.000.000 operações a partir do último.

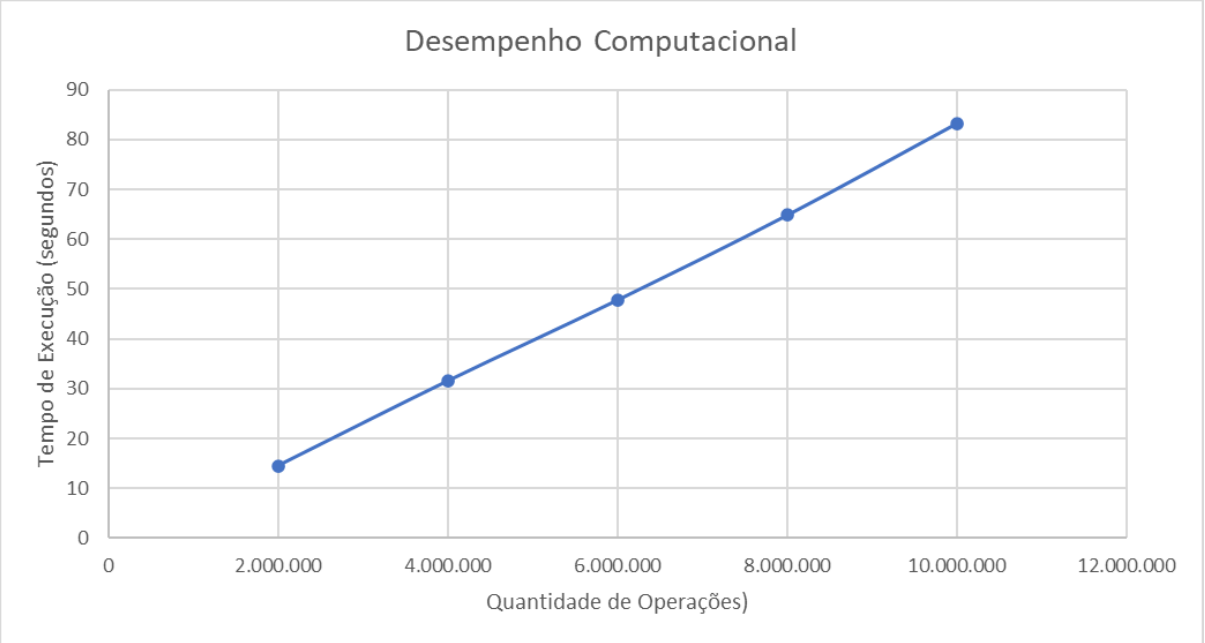
Em cada arquivo de operação, haverá sempre 5 usuários do servidor de e-mail e a quantidade de e-mails que cada um possui será sempre um décimo do número "*m*" de operações.

A seguir, será apresentado o resultado obtido para este teste.

Evolução do Desempenho Computacional

Veremos uma tabela que mostra a evolução do desempenho com o crescimento da quantidade de rodadas a serem computadas pelo programa. Será exibido também um gráfico que mostra esta evolução de forma mais visual.

Quantidade de operações e nome do arquivo	**Tempo inicial**	**Tempo final**	**Duração da execução**
2.000.000 entrada_1.out	7858.471551548	7872.993617547	14.522065999
4.000.000 entrada_2.out	7873.005934335	7904.567054664	31.561120329
6.000.000 entrada_3.out	7904.589793973	7952.384000945	47.794206972
8.000.000 entrada_4.out	7952.416531970	8017.263431502	64.846899532
10.000.000 entrada_5.out	8017.305135904	8100.603735328	83.298599424



A complexidade de tempo do programa total havia sido estimada como sendo $O(m * n)$. Porém, como foi explicado na apresentação deste teste, haverá sempre 5 usuários do servidor, além de a quantidade "n" de e-mails armazenados em cada árvore pode ser dado em função do número "m" de operações executadas. A saber, temos que: $n = \frac{m}{10}$. Como "n" é significativamente menor que "m", ele será representado como uma constante.

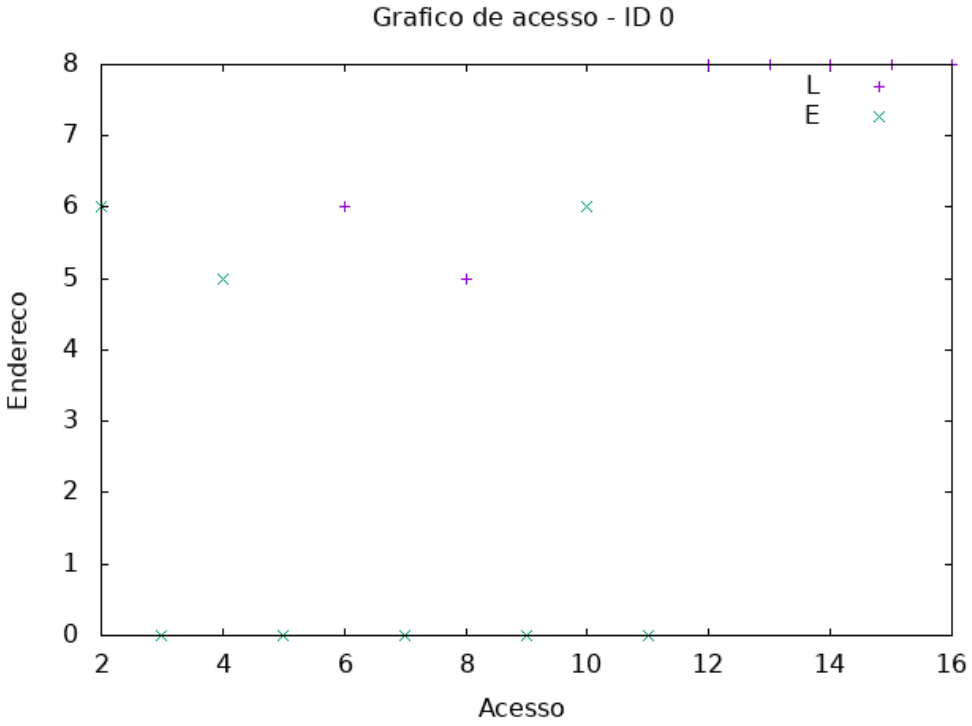
Portanto, teremos que a complexidade de tempo para o experimento realizado é $O(m)$, que é precisamente a reta representada no gráfico.

Análise de Padrão de Acesso à Memória e Localidade de Referência

Os testes de memória serão realizados com uma entrada de 5 operações, para melhor visualização de acesso. Duas delas serão de "ENTREGA", outras duas de "CONSULTA" e a última, a operação "APAGA". Não serão repetidos estes testes, já que eles são apenas para geração dos gráficos.

Mapa de acesso à memória

Este gráfico mostra como os endereços de memória do programa são acessados a cada chamada de função. Pontos em formato de "X" na cor verde representam acesso à memória para escrita, já os em formato de "+" na cor rosa, representam acesso para leitura.



Os primeiros acessos (2 a 5) são provenientes do envio dos dois e-mails iniciais, os registros de escrita na parte superior do gráfico são o armazenamento nas caixas de entrada e os que estão intercalados com os superiores na parte inferior, são o registro da operação no log do servidor. Os próximos acessos (6 a 9) representam a consulta dos e-mails no servidor e o registro da operação no log. Os registros seguintes (10 e 11), estão relacionados à deleção de um dos e-mails na caixa de entrada de um dos usuários, bem como o registro dessa operação no log.

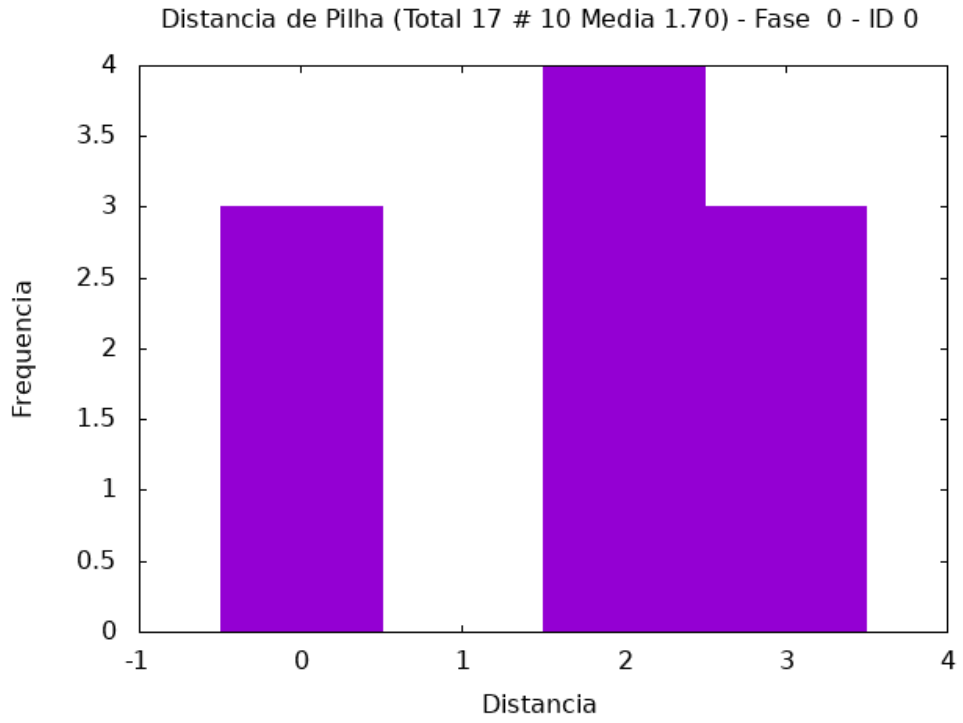
Os últimos registros (12 a 16) são todos de leitura, estando atrelados à consulta das operações registradas no log para impressão em um arquivo de texto.

É possível verificar que os acessos estão próximos uns dos outros e seguindo uma sequência lógica de acesso à memória.

Distância de pilha

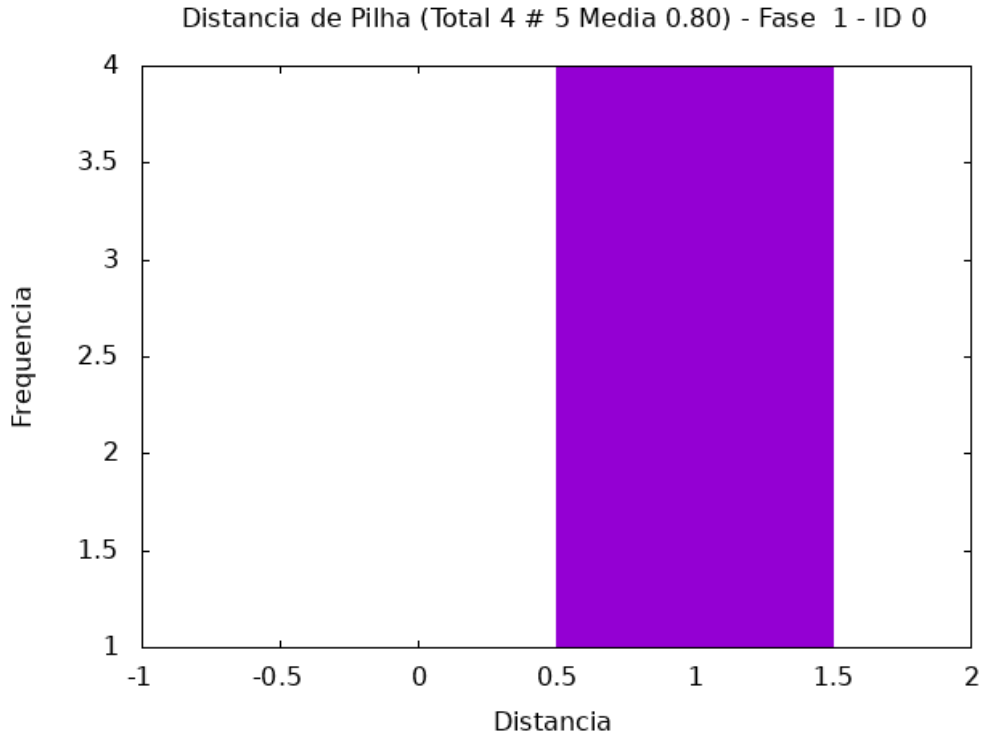
Estes gráficos são chamados de histogramas e mostram a distância de pilha dos acessos aos endereços de memória alocados para as estruturas de dados implementadas.

Durante a fase 0, o programa está responsável pela leitura dos comandos que serão executados no servidor, bem como a execução dessas operações e armazenamento das mesmas em um log. Podemos visualizar as distâncias de pilha para esses acessos a seguir:



É possível verificar que a distância de pilha das chamadas está bastante próxima e com índices baixos, o que indica bom uso da memória e bom uso da localidade de referência.

Já na fase 1, o programa está responsável pela leitura e escrita dos logs de operação no arquivo de saída, o que nos faz esperar uma grande frequência de acessos em uma única distância de pilha, já que todos os logs estão armazenados em uma mesma estrutura:

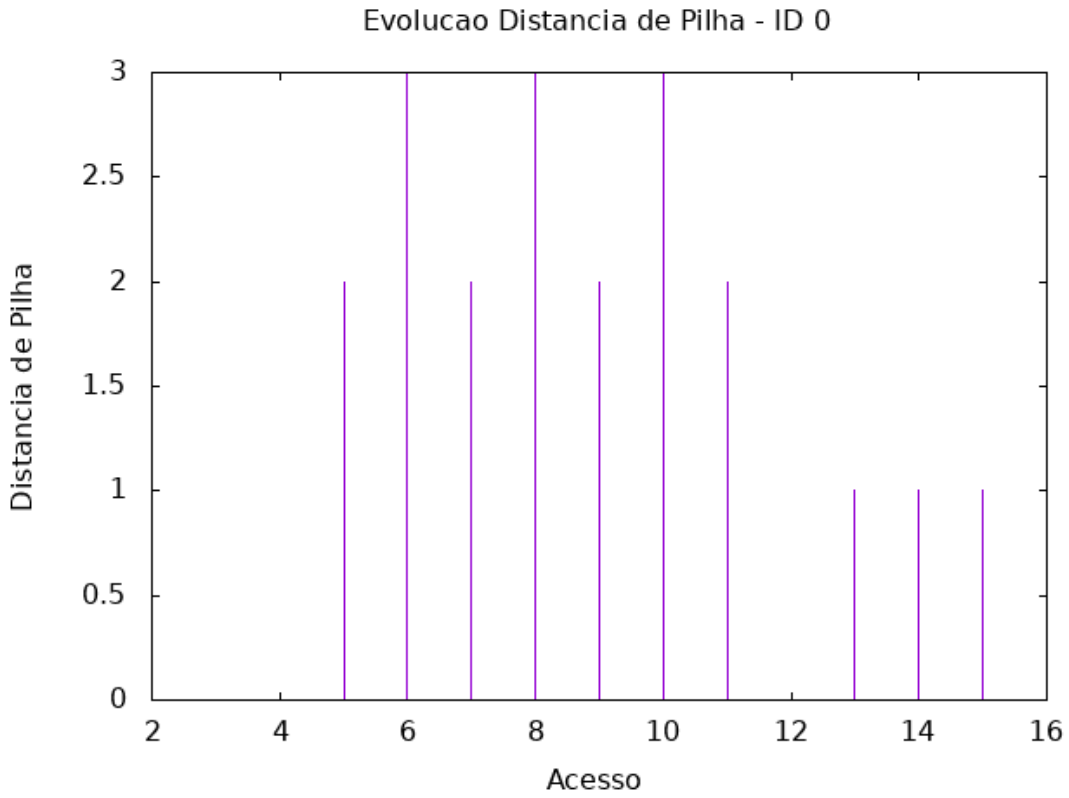


Evolução da Distância de Pilha

A distância de pilha deve apresentar valores intercalados nos primeiros acessos até a metade superior, já que há basicamente duas estruturas sendo acessadas intercaladamente por vez: a caixa de entrada e o log de operações.

Em seguida, os acessos devem ter valores iguais, um seguido do outro, já que se trata da leitura e impressão dos logs de operações do servidor.

É possível verificar essa análise pelo gráfico a seguir.



Conclusão

Neste trabalho, foram implementados alguns Tipos Abstratos de Dados (TADs) para armazenar mais informações sobre um e-mail, bem como seu conteúdo. Além de outros TADs mais conhecidos, como Árvore de Pesquisa Binária (BST), Fila Encadeada e Tabela Hash. Para esta aplicação em específico, não foi necessário utilizar nenhum algoritmo de ordenação, já que a pesquisa é facilitada pela BST.

Toda a implementação foi feita na linguagem C + +, utilizando o compilador G + +, sendo também desenvolvido em um subsistema de Linux para Windows (WSL 2). Foi

possível executar os conceitos vistos em sala de aula sobre estruturas de dados e algoritmos de pesquisa e ordenação, bem como realizar a análise de complexidade de cada um destes e exercitar as boas práticas de programação em C + +.

O trabalho contribuiu bastante para a consolidação dos conhecimentos teóricos de estruturas de dados apresentados na disciplina, sendo especialmente importante para a visualização de aplicações práticas dos conhecimentos adquiridos.

Bibliografia

Meira, W. and Pappa, G. (2021). Slides virtuais da disciplina de estruturas de dados. Disponibilizado via moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

Instruções para compilação e execução

1. Extraia o arquivo .zip presente;
2. Abra um terminal na pasta raiz do mesmo;
3. Digite “make”. Isso compilará o programa;
4. Digite “./bin/tp3.exe <argumentos>”.

    Os argumentos podem ser:

    a. -i / -I <input.txt> (arquivo com os comandos a serem executados)
    b. -o / -O <output.txt> (arquivo para registrar o log de operações executadas no servidor)
    c. -p <arquivo_de_logs.txt> (arquivo para registrar os resultados computacionais de performance e/ou memória da partida)
    d. -l Caso esta flag esteja presente, o acesso à memória é registrado no arquivo de logs de “-p”

OBS: O programa também pode ser executado sem parâmetro algum. Ele está configurado para, por padrão, acessar como input o arquivo "*entrada.txt*" e como output o arquivo "*saida.txt*”, desde que ambos estejam na mesma pasta em que o terminal foi aberto, ou seja, a pasta raiz do projeto.

É possível também executar o comando “*./bin/tp3.exe -h*”. Com isso, as instruções de utilização serão exibidas na tela.