Trabalho Prático 3
MailingSistem

Daniel Alves Resende
2020026834

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

danielresende2000@ufmg.br

1. Introdução

Proposto para os alunos de Estruturas de Dados da turma do primeiro semestre de 2022, o trabalho prático nº 3 tem como objetivo apresentar um desafio para que os alunos coloquem em prática os conhecimentos adquiridos até então em aula. Os estudantes receberam a tarefa de criar um programa que simule um sistema de registro de e-mails. Na entrada, comandos de inserção, remoção e consulta de elementos podem ser utilizados, e um arquivo textual será gerado como saída.

O programa deve ser um sistema de e-mails otimizado. Por conta disso, uma especificação do programa é que o mesmo deve ser estruturado com uma tabela hash, com cada elemento possuindo uma árvore binária para casos de colisão.

Foram disponibilizados para os alunos: um enunciado com todas as descrições do trabalho; as ferramentas *MEMLOG*, *analisamem* e *msgassert*; ambos modelo e exemplo de documentação.

2. Implementação

O programa foi escrito em C++, compilado pelo compilador GCC da GNU compiler collection. A entrada de dados deve ser fornecida em forma de um arquivo de entrada, cujo especifidades serão esclarecidas na última seção desse documento.

2.1. Classes

Para a implementação do sistema, foram utilizadas as seguintes classes: ***MsgContainer***, que armazena todas as informações de cada registro de email, ***TreeNode*** e ***AVL***, que compõem a árvore AVL, e ***Hash***, como a estrutura de dados superior do sistema. A classe *MsgContainer* possui *overloads* das funções de comparação, então as árvore *AVL* é implementada sem alterações à sua versão generalizada.

A classe *MsgContainer* possui dois construtores não-vazios. O primeiro recebe o identificador da mensagem, o identificador do destinatário, o número de palavras da mensagem e a mensagem em si. O segundo recebe apenas os identificadores de mensagem e destinatário, já que nos comandos de remoção e consulta de elementos essas são as únicas informações fornecidas.

2.2. Estrutura de Dados

Como informado anteriormente, as estruturas de dados utilizadas foram árvores *AVL* e uma tabela *Hash*. Como queremos otimizar as operações de inserção, consulta e remoção de elementos, a estratégia hash é a melhor a ser implementada. As árvores AVL são utilizadas para tratamento de colisões. Como a variedade AVL das árvores binárias normaliza os melhores e piores casos, foi decidido que a mesma seria utilizada.

2.3. Métodos e Análise de Complexidade

Pela simplicidade da classe MsgContainer, todos seus métodos são O(1). Usaremos para a análise de complexidade final **n** igual ao número de mensagens de email inseridas. Além disso, antes precisamos expor que a altura da árvore AVL é, por definição, O(log n). Isso será útil para a análise de alguns métodos.

Os métodos relevantes para a análise são os seguintes:

TreeNode* AVLTree::insert_recursive

Recebe um nodo cabeça e um elemento a ser inserido, e chama outras iterações dependendo da comparação do novo elemento ao atual. Quando a cabeça é nula, insere o novo nodo e o retorna. Por fim balanceia a árvore caso necessário. No melhor caso, o balanceamento não é necessário e a inserção leva complexidade O(log n). O pior caso exige balanceamento, mas como as funções utilizadas possuem complexidade O(1), esse também é O(log n).

void AVLTree::insertNewNode

Chama a função utilidade insert_recursive. O(log n).

TreeNode* AVLTree::search_recursive

Possui lógica similar à inserção recursiva, exceto pelas alterações e balanceamento. É uma pesquisa binária padrão, com melhor caso O(1), pior caso O(log n) e caso médio O(log n).

TreeNode* AVLTree::searchNode

Chama a função utilidade search_recursive. O(log n).

TreeNode* AVLTree::delete_recursive

Função quase idêntica à inserção recursiva, exceto pelas operações de remoção quando o nodo é identificado. A diferença mais notável é a pesquisa do menor/maior nodo de uma das folhas do nodo recebido pela função, que possui complexidade O(log n). Isso não altera a complexidade final da operação, porém, mantendo-a O(log n).

void AVLTree::deleteNode

Chama a função utilidade delete_recursive. O(log n). Chama também um search para garantir a presença do elemento antes da exclusão.

void Hash::insert

Calcula o fator hash do elemento a inserir, e chama insertNewNode na árvore equivalente. Melhor, pior e caso médio todos são O(log n).

void Hash::remove

Calcula o fator hash do elemento a inserir, e chama deleteNode na árvore equivalente. Melhor, pior e caso médio todos são O(log n).

MsgContainer Hash::consult

Calcula o fator hash do elemento a inserir, e chama searchNode na árvore equivalente. Melhor caso O(1), pior e médio casos são O(log n).

void readMailLogFile

Função responsável por executar o programa. De acordo com o número de mensagens armazenada, não muda o tempo de execução. O(1). Nuances serão discutidas abaixo.

Como cada comando é executado separadamente de acordo com a entrada, calcular uma complexidade final para o programa é infrutífero. Podemos, em vez disso, afirmar que as operações de inserção, consulta e remoção possuem todas uma complexidade média O(log n), e a consulta possui melhores casos O(1).

O programa como um todo possui uma complexidade mais difícil de se entender. A função *readMailLogFile* itera pelo arquivo de entrada em forma de loop, então a complexidade final dela é maior que O(1), sendo considerados parâmetros diferentes do número de mensagens armazenadas. Portanto, consideraremos:

A = Número de comandos de inserção
B = Número de comandos de consulta
C = Número de comandos de remoção
P = Número de comandos totais = A + B + C
m = Tamanho médio das mensagens inseridas em palavras
n = Número de mensagens já inseridas no tempo de execução do comando

Primeiramente, sabemos que o programa itera pelos comandos e lê as mensagens em caso de inserções. Assim, já encontramos uma complexidade inicial O(Am + B + C). Sabemos também que em todo comando o programa encontra o hash equivalente da entrada e itera pela árvore procurando uma posição ou o elemento informado. Sendo uma árvore AVL, o caso médio de todas as operações é O(log n), então a complexidade até então é O(A(m + log n) + B(log n) + C(log n)). Como P é igual à soma de A, B e C, escrevemos isso como **O(P*log n + A*m)**.

Podemos perceber que essa complexidade é **linear** com relação ao **número de comandos**, independentemente de qual dos três seja usado, e também com relação ao **número de palavras das mensagens**. Com relação aos elementos já inseridos, ele mantém sua natureza logarítmica.

4. Estratégias de Robustez

A ferramenta *msgassert* foi usada ao decorrer do programa para garantir que casos em que o usuário não tenha seguido as especificações de uso não causem problemas na execução. Sempre que algum comportamento fosse determinado como previsível e indesejado, seria feita uma chamada da função *erroAssert* para impedir esse cenário. Caso o usuário tente fazer uma operação julgada como fora dos limites do programa, uma mensagem de erro o avisa da inconsistência com objetivo de instruí-lo a utilizar o programa corretamente. A função *avisoAssert* também foi utilizada quando a funcionalidade do programa não fosse comprometida, reportando ao usuário a solução utilizada.

Alguns exemplos são:

- Se o usuário começar uma linha do arquivo de entrada com um comando não especificado, o programa parará a execução e retornará uma mensagem de erro;
- Se o usuário não informar os nomes dos arquivos de entrada e saída corretamente, o programa não executará e retornará uma mensagem de erro;
- Caso uma operação de inserção seja chamada com dados chave já existentes na estrutura, o programa ignorará a chamada desse comando e continuará a execução normalmente, retornando uma mensagem de erro breve;
- Se o arquivo de entrada não for encontrado, o programa retornará uma mensagem de erro.

5. Testes

Pela natureza complexa da análise de memória do programa, diversos testes diferentes foram executados. Todos as entradas utilizadas possuiam parâmetros variantes, e cada caso de teste variava parâmetros específicos. Esses parâmetros são: número de entradas; número de consultas; número de remoções; número de palavras máximas por mensagem; tamanho do hash; número máximo dos usuários.

Os testes são:

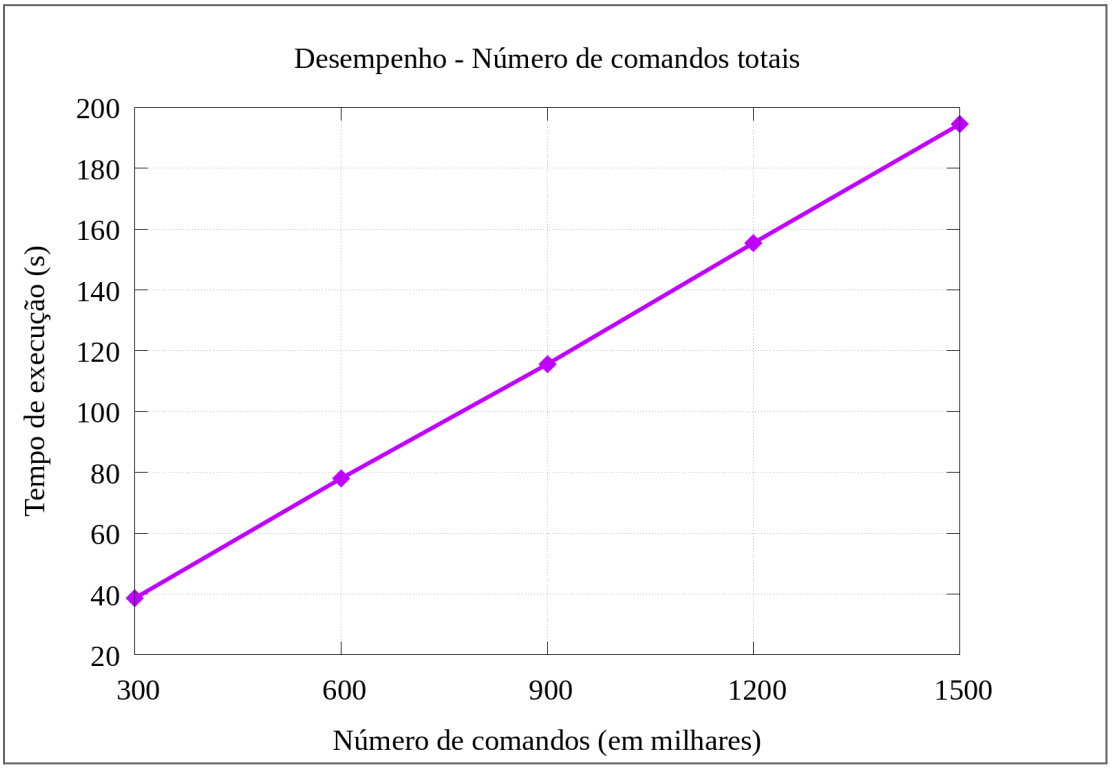
- **Teste de qualidade:** Os testes de qualidade da saída do programa foram executados utilizando os arquivos de entrada e saída disponibilizados pelos professores. O programa é executado diversas vezes com as entradas recebidas, e as saídas são comparadas com os arquivos de saída recebidos. A comparação foi feita rapidamente com um programa de comparação de arquivos.
- **Teste de desempenho 1 (variando número de comandos):** Variam os números de comandos, sendo esses inserção, consulta e remoção. Eles variam em conjunto, de cem mil a quinhentos mil. Nenhum outro parâmetro é alterado.
- **Teste de desempenho 2 (variando tamanho das mensagens):** Varia-se o número máximo de palavras por mensagem em progressão logarítmica, de dez palavras a cem mil. O número de inserções é dez mil, e não foram feitas consultas e remoções.
- **Teste de desempenho 3 (variando número de usuários):** Varia-se o número máximo do ID dos usuários, mantendo o número de inserções, consultas e remoções constantes e iguais a dez mil. O tamanho do hash é 500, e os usuários variam de 200 a 1000 em incrementos de 200.
- **Teste de desempenho 4 (variância de comandos):** Variam o número de consultas e remoções com relação ao número de entradas, e o número de entradas em si. As consultas e remoções tomam porcentagens de zero, dez e cem porcento no número de entradas, e o mesmo varia de dez a cinquenta mil.

Estes testes foram então executados, e os dados foram compilados em gráficos usando *GNUplot*. O teste de desempenho de variância de comandos possui três casos de teste incluidos em si.

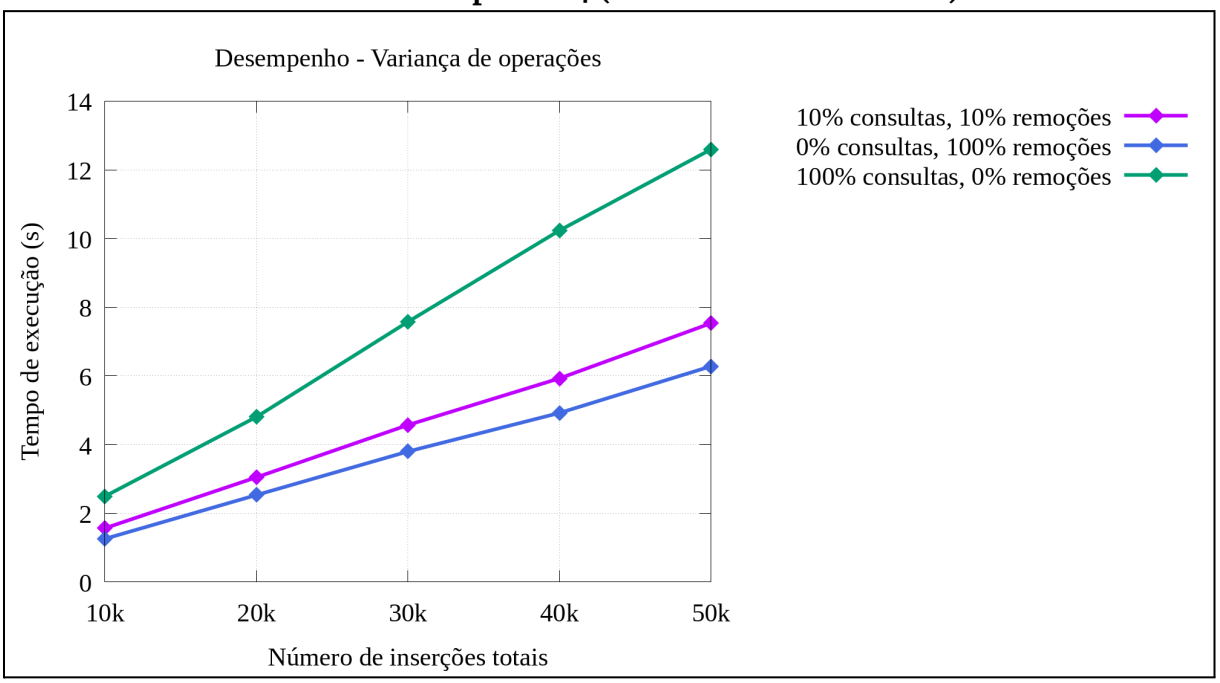
6. Análise Experimental

6.1 Desempenho Computacional

Teste de desempenho 1 (variando número de comandos):

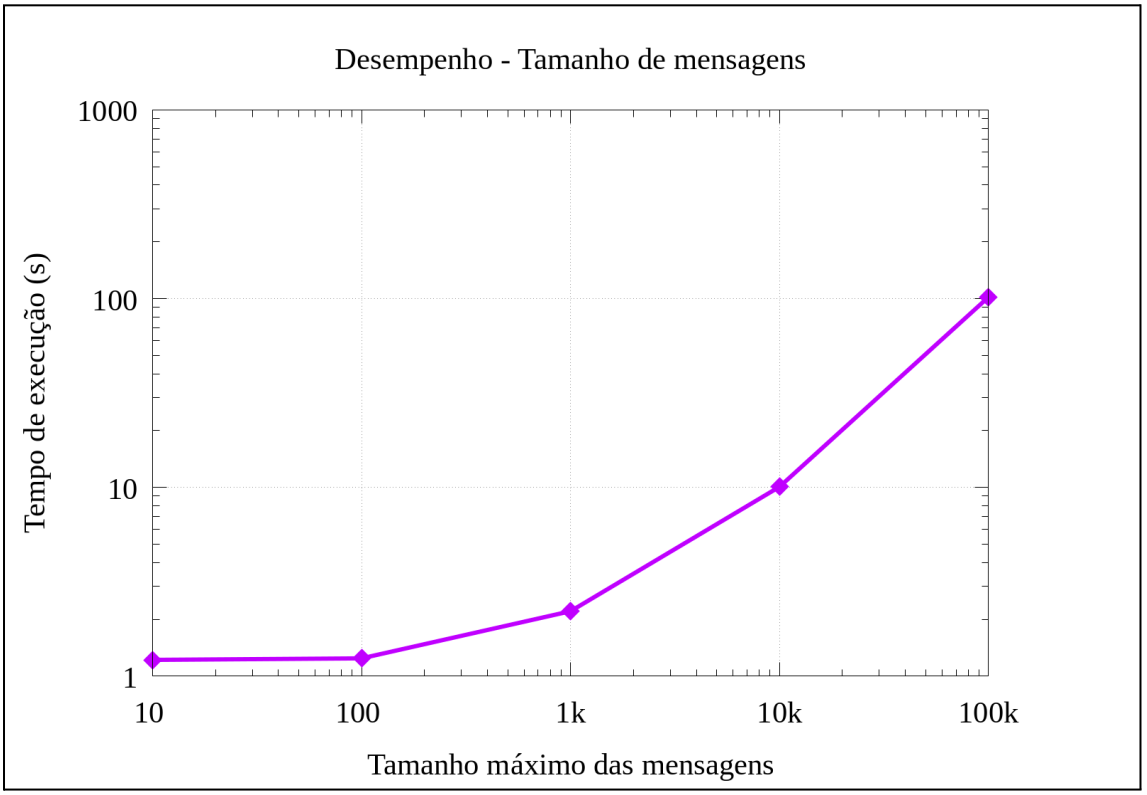


Teste de desempenho 4 (variância de comandos):

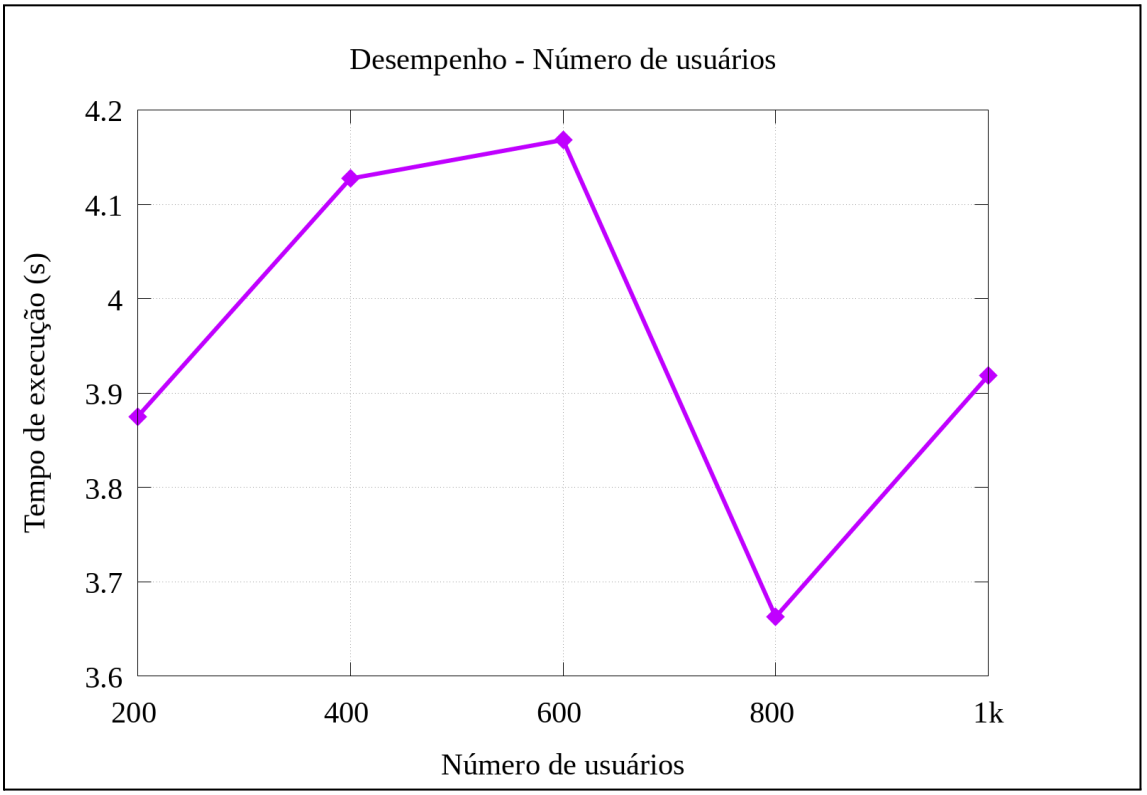


O gráficos nos confirmam a natureza de complexidade linear com relação ao número de comandos. Existe uma variação de acordo com o número de elementos inseridos, mas a mesma não acompanha o crescimento da complexidade dos comandos. Assim, as curvas finais são quase retas quando o número de comandos é significativamente elevado. Os casos com mais remoções tem tempos de execução menores por conta da diminuição de **n** conforme os comandos de remoção são feitos.

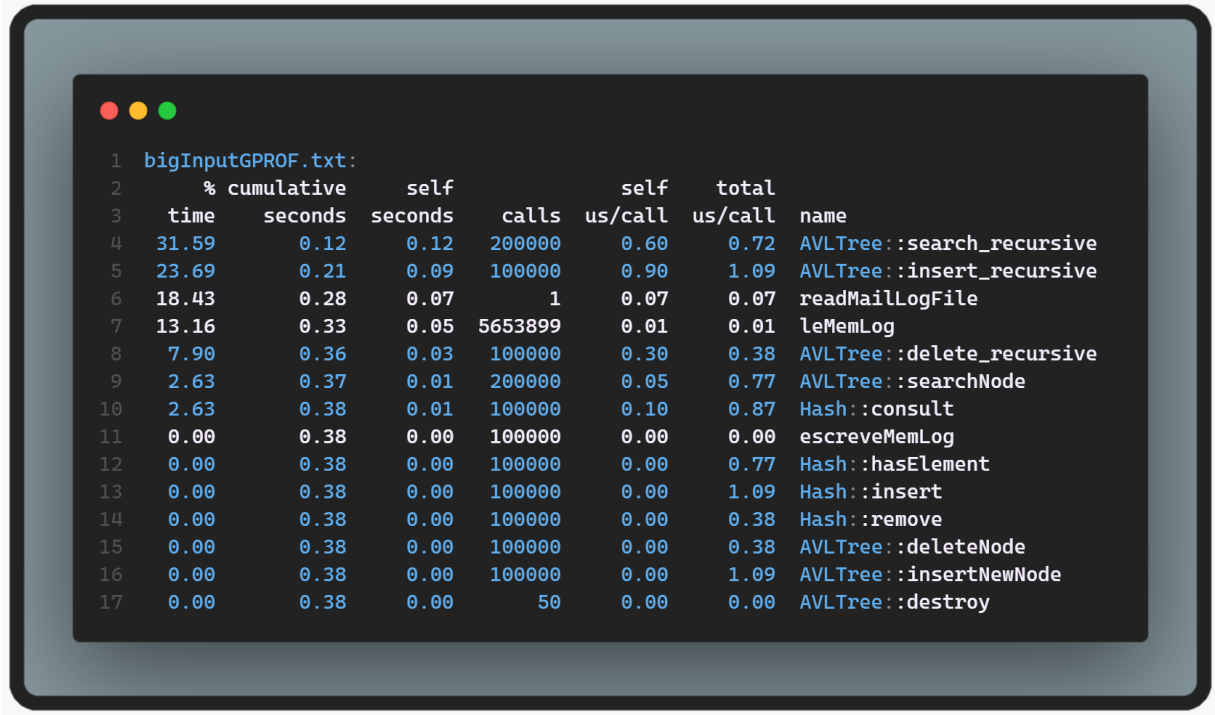
Teste de desempenho 2 (variando tamanho das mensagens):



Teste de desempenho 3 (variando número de usuários):



Por fim observamos que, como previsto, o tamanho das mensagens possui complexidade de tempo linear (importante notar que o gráfico é logarítmico). Também percebemos que o número de usuários é insignificante para o tempo de execução do programa. Isso ocorre porque as colisões acontecem aleatoriamente independentemente desse número, desde que ele seja de uma ordem maior que o tamanho da tabela *Hash*.

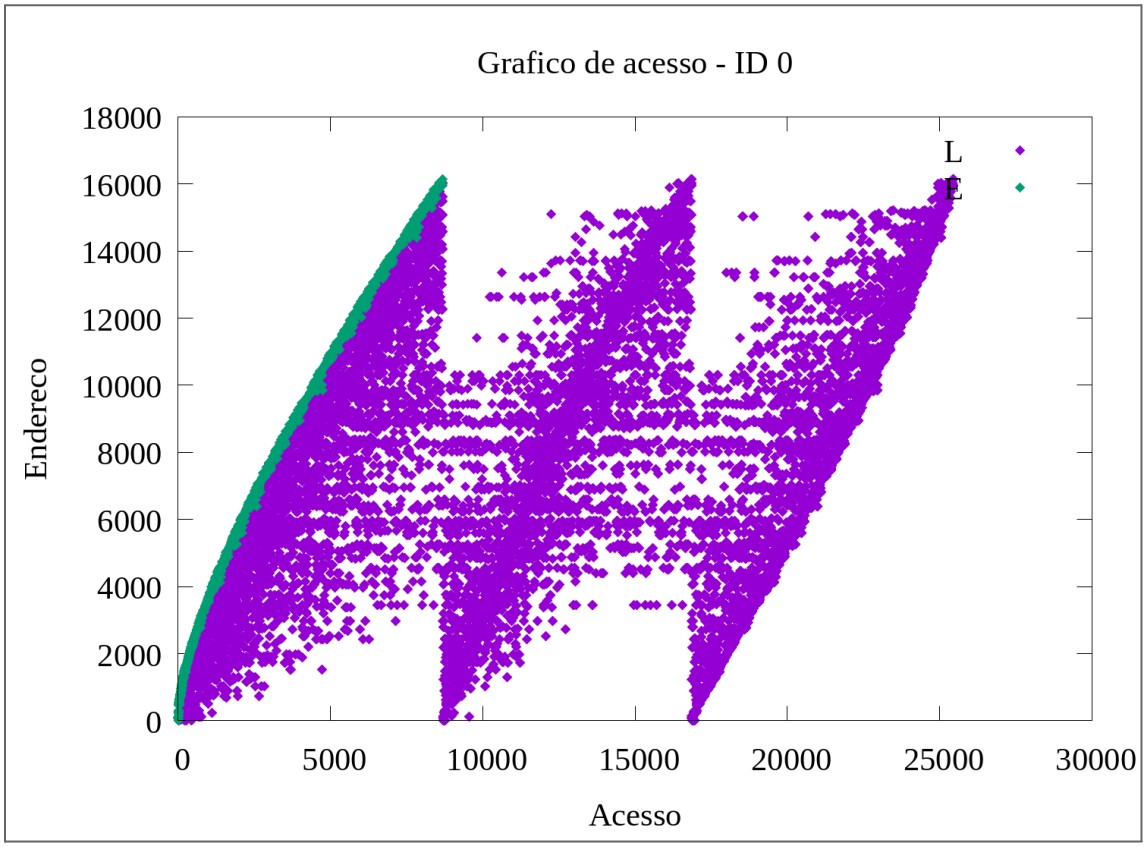


```
bigInputGPROF.txt:
     % cumulative     self              self     total
    time    seconds  seconds    calls  us/call  us/call  name
   31.59       0.12     0.12   200000     0.60     0.72  AVLTree::search_recursive
   23.69       0.21     0.09   100000     0.90     1.09  AVLTree::insert_recursive
   18.43       0.28     0.07        1     0.07     0.07  readMailLogFile
   13.16       0.33     0.05  5653899     0.01     0.01  leMemLog
    7.90       0.36     0.03   100000     0.30     0.38  AVLTree::delete_recursive
    2.63       0.37     0.01   200000     0.05     0.77  AVLTree::searchNode
    2.63       0.38     0.01   100000     0.10     0.87  Hash::consult
    0.00       0.38     0.00   100000     0.00     0.00  escreveMemLog
    0.00       0.38     0.00   100000     0.00     0.77  Hash::hasElement
    0.00       0.38     0.00   100000     0.00     1.09  Hash::insert
    0.00       0.38     0.00   100000     0.00     0.38  Hash::remove
    0.00       0.38     0.00   100000     0.00     0.38  AVLTree::deleteNode
    0.00       0.38     0.00   100000     0.00     1.09  AVLTree::insertNewNode
    0.00       0.38     0.00       50     0.00     0.00  AVLTree::destroy
```

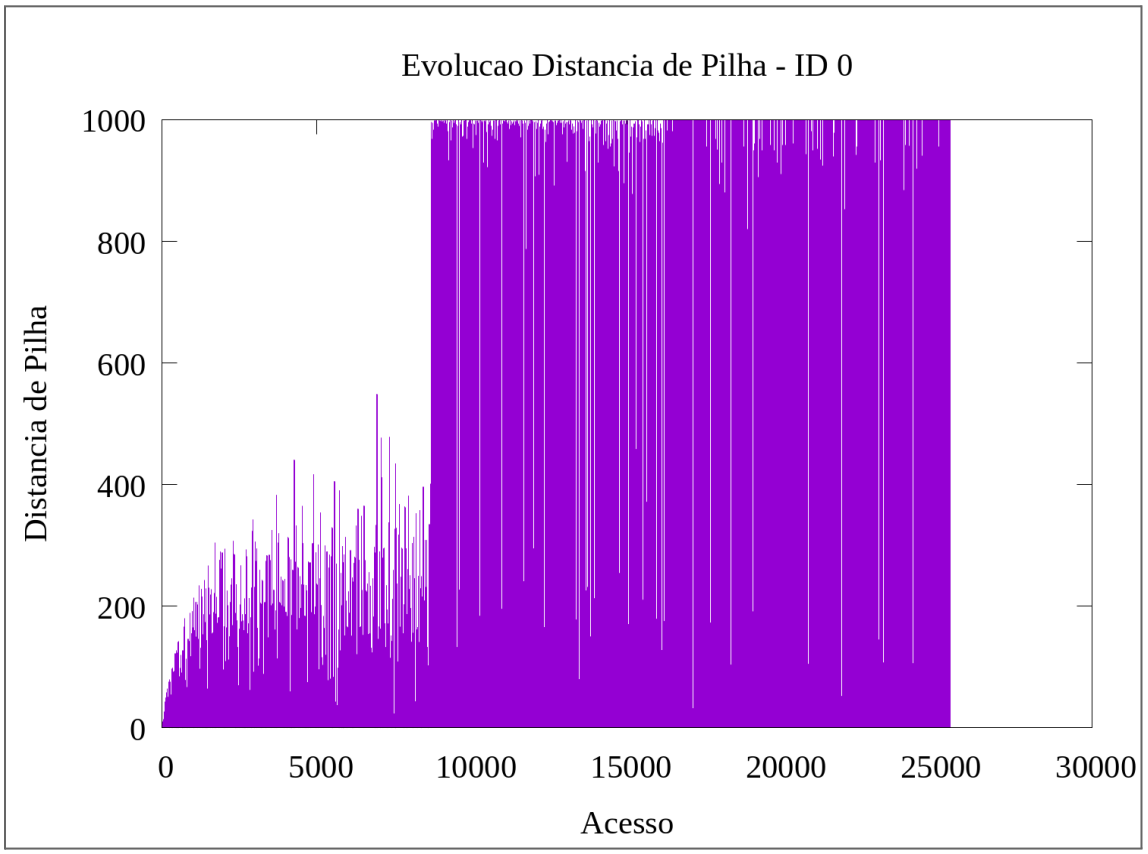
Podemos observar que as funções que mais despendem o tempo do programa são as funções de inserção e leitura de entradas. A pesquisa e remoção também possuem tempos de execução consideráveis, mas não impactam tanto quanto as outras duas.

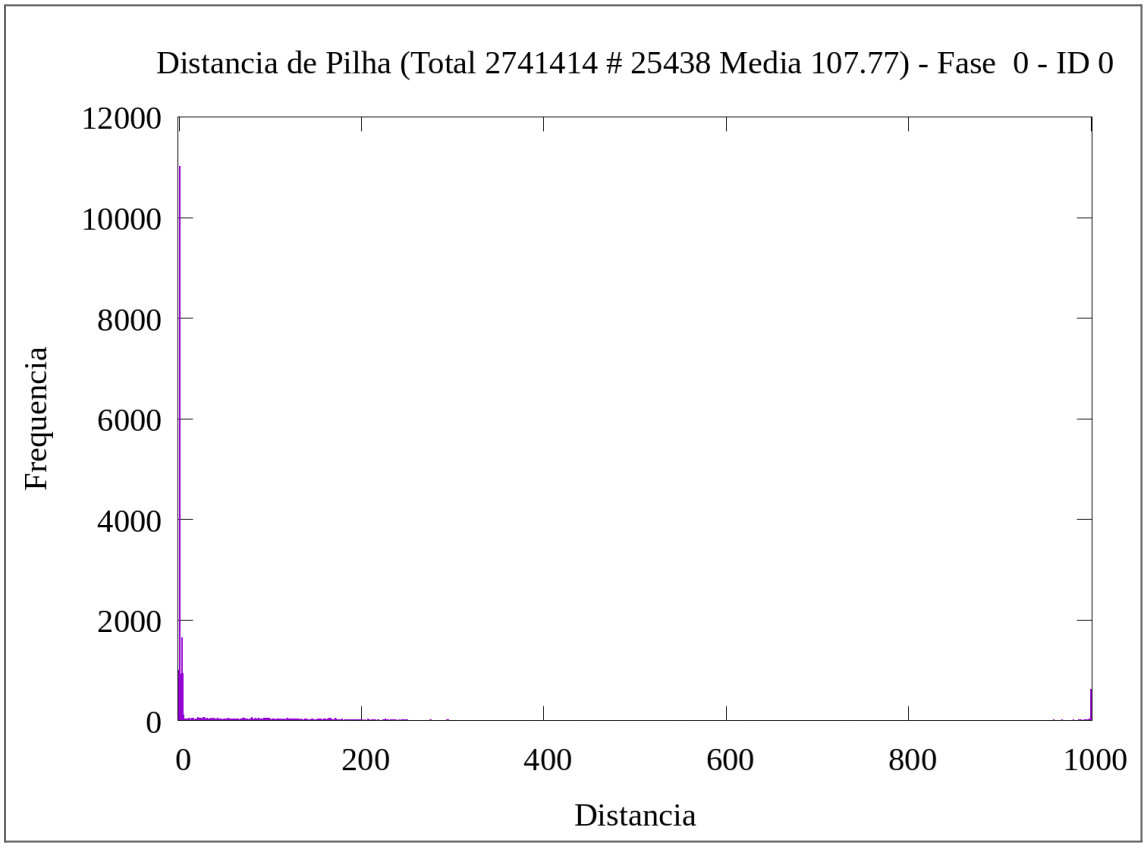
6.2 Padrão de Acesso à Memória e Localidade de referência

6.2.1 Gráfico de acesso



6.2.2 Histogramas e evolução de localidade de referência





6.2.4 Análise

GRÁFICO DE ACESSO: Este gráfico representa um teste que progressivamente faz inserções, consultas e remoções com comandos aleatorizados. Eles ocorrem, porém, com a mesma ordem de elementos, isto é, o primeiro elemento inserido é o primeiro elemento a ser consultado e também o primeiro elemento a ser removido. O teste foi feito desta forma pelo gráfico gerado ser conveniente para leitura.

Pela natureza do teste, podemos perceber uma progressão linear dos processos de inserção, consulta e remoção. Conforme as inserções ocorrem, cada comando se torna mais custoso. Isso é refletido no gráfico na densidade de comparações abaixo de cada inserção subsequente. Da mesma forma, o inverso acontece nas remoções, tornando todo comando subsequente mais leve.

LOCALIDADE DE REFERÊNCIA: Uma característica importante da árvore binária AVL é o quão erráticos são os acessos de memória. Isso ocorre por conta do rebalanceamento da árvore, que move os nodos, fazendo com que dois nodos conectados tenham endereços mais afastados que o normal. Além disso, todo comando itera sobre a árvore e gera distância do nodo inicial, fazendo com que o início do próximo comando gere um registro de distância de pilha de tamanho considerável. Assim, a inserção possui pilhas crescentes, e tanto a consulta quanto a remoção geram registros altíssimos.

7. Conclusão

Esse trabalho foi uma ótima oportunidade para exercitar o conhecimento dos alunos com relação às escolhas importantes a serem feitas durante o processo de criação de um sistema complexo de dados. Embora o mesmo tenha sido o trabalho prático mais restritivo em questão aos requisitos da estrutura de dados, isso possibilitou aos alunos um entendimento mais aprofundado de decisões que normalmente são secundárias à escolha da estrutura em si. Assim, em conjunto com os outros trabalhos práticos, todos os passos do processo de desenvolvimento aprendidos em sala de aula foram colocados em prática na disciplina. Como trabalho prático final, o mesmo ata nossos conhecimentos em um pacote sucinto, fazendo com que os mesmos se afixem eficientemente na memória do aluno.

Bibliografia

CHAIMOWICZ, L. **PRATES, R.** Slides virtuais da disciplina de estruturas de dados (2020), disponibilizado via moodle. Departamento de Ciencia da Computação, Universidade Federal de Minas Gerais, Belo Horizonte. Acesso em Junho de 2022.

Instruções para compilação e execução

Para a compilação do programa final, é necessário antes compilar os objetos que serão usados pelo mesmo. Assim, a melhor forma de compilar o programa é com o arquivo *Makefile*. Executar o comando '*make*' no terminal enquanto na pasta do projeto compilará o executável '*tp2.exe*' na pasta *bin*. Caso seja necessário, o código a seguir deve ser utilizado para criar outro arquivo *Makefile*.

```
CXX = g++
SRC = src
OBJ = obj
INC = include
BIN = bin
OBJS = $(OBJ)/memlog.o $(OBJ)/mailingSistem.o $(OBJ)/Hash.o $(OBJ)/AVL.o
HDRS = $(INC)/msgassert.hpp $(INC)/memlog.hpp $(INC)/MsgContainer.hpp
$(INC)/TreeNode.hpp $(INC)/AVL.hpp $(INC)/Hash.hpp
LDFLAGS =  -pg -O3
CXXFLAGS = -Wall -c -I$(INC) $(LDFLAGS)
ANALISAMEM = ../analisamem/bin/analisamem
EXE = $(BIN)/tp3.exe

$(EXE): $(OBJS)
    $(CXX) $(LDFLAGS) -o $(EXE) $(OBJS)

$(OBJ)/mailingSistem.o: $(HDRS) $(SRC)/mailingSistem.cpp
    $(CXX) $(CXXFLAGS) -o $(OBJ)/mailingSistem.o $(SRC)/mailingSistem.cpp

$(OBJ)/memlog.o: $(HDRS) $(SRC)/memlog.cpp
    $(CXX) $(CXXFLAGS) -o $(OBJ)/memlog.o $(SRC)/memlog.cpp

$(OBJ)/Hash.o: $(HDRS) $(SRC)/Hash.cpp
    $(CXX) $(CXXFLAGS) -o $(OBJ)/Hash.o $(SRC)/Hash.cpp

$(OBJ)/AVL.o: $(HDRS) $(SRC)/AVL.cpp
    $(CXX) $(CXXFLAGS) -o $(OBJ)/AVL.o $(SRC)/AVL.cpp
```

Para a execução do programa *tp3.exe*, deve-se, no terminal, ser utilizado a seguinte estrutura de comandos:

bin/tp3.exe

-i/I	<arg string>	(arquivo de entrada)
-o/O	<arg string>	(arquivo de saída)
-p	<arg string>	(arquivo de registro de desempenho)
-l		(padrão de acesso e localidade)

Importante: o programa **não funcionará** se nomes de arquivos de entrada e/ou saída não forem informados, e retornará um *erroAssert*.

Um exemplo de comando é válido é:

```
bin/tp3.exe -i entrada.in -o saida.out
```