Trabalho Prático 3
Servidor de E-mails otimizado

Vitor Emanuel Ferreira Vital

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

vitorvital@dcc.ufmg.br

1. Introdução

Esta documentação tem como objetivo apresentar o decorrer da implementação da resolução do Trabalho Prático 3 da disciplina de Estrutura de Dados. Em tal prática, os principais temas explorados que devem ser solucionados na linguagem C/C++ são:

- Analise arquivos de texto;
- Manipulação de strings;
- Algoritmos de pesquisa;
- Desempenho do programa;

A documentação foi dividida em 6 seções principais, incluindo a introdução. A segunda seção trata da descrição das soluções implementada; já na terceira seção é apresentado a análise de complexidade de tempo e espaço dos procedimentos apresentados; a quarta seção é marcada pela descrição, justificativa e implementação dos mecanismos de programação defensiva e tolerância a falhas implementado; na quinta e penúltima seção trabalha-se a análise do desempenho computacional e localidade de referência, assim como as análises dos resultados; por último temos a conclusão geral da documentação.

2. Método

Para resolução do problema proposto o código foi dividido em seis programas distintos responsáveis por determinadas etapas da resolução. Tais divisões permitem uma maior organização do programa, pois cada etapa é responsável por uma função para resolução como um todo. Portanto, foram aglutinadas parte que empenham semelhantes papéis. A configuração dos TAD's foram feitas utilizando as seguintes funções:

- Algoritmo de pesquisa via Hashing;
- Algoritmo de pesquisa via Árvore Binária de Pesquisa;
- Definição do formato de um e-mail;
- Processador de linhas e textos;
- Analisador de desempenho;
- Mecanismos de programação defensiva e tolerância a falhas;

Precedendo a organização dos e-mails dentro da tabela e posteriormente em árvores binárias de pesquisa, o algoritmo preocupa-se, de início, em criar a tabela Hash no qual as mensagens serão endereçadas a partir da transformação aritmética sobre a chave de pesquisa. Para isso, obtém-se a informação do tamanho esperado passado no arquivo de entrada alocando uma tabela, que utiliza o Hashing, de Árvores Binárias de Pesquisa.

O funcionamento de cada inserção na tabela é realizada com a utilização de dois códigos, o InBox, referente à caixa de destino, no qual a mensagem seŕa armazenada, e o ID, que é o código da mensagem a ser armazenada. Para isso, computa-se o valor do endereço da tabela através da função de transformação, sendo essa necessária para conversão de uma chave no endereço desejado. Neste trabalho, a função utilizada foi InBox%N, no qual N é o tamanho da tabela Hash e InBox, como supracitado, o código referente ao endereço no qual a mensagem será armazenada.

Após a localização da posição no qual a mensagem será armazenada, tem-se a preocupação em definir a posição dessa mensagem na árvore binária de pesquisa contida em cada endereço da tabela Hash. O comportamento leva em conta o ID da mensagem, caso o ID seja maior que o ID da posição analisada verificamos a localização de inserção da mensagem na subárvore da direita, caso seja menor verificamos na subárvore da direita. Por fim, ao encontrar uma posição nula inserimos a nova mensagem.

Seguindo tal implementação teremos uma tabela de N posições no qual cada endereço possuirá uma árvore binária de pesquisa. Com isso, é possível armazenar as mensagens com base nos códigos definidos no programa.

Hashing

No algoritmo de pesquisa Hashing, utiliza-se a transformação aritmética sobre as chaves de pesquisa, conforme citado anteriormente. Os registros armazenados na tabela são diretamente endereçados a partir de uma transformação aritmética sobre a chave de pesquisa, que nesse caso é calcular o resto da divisão entre o código da caixa de destino e o número de partições da tabela Hash.

Entretanto, a busca por transformar uma chave em endereços de uma tabela pode ocasionar alguns problemas.

Como a função de transformação é a mesma utilizada em todas as chaves pode ocorrer de chaves diferentes possuírem um mesmo resultado ao serem convertidas a um endereço da tabela. Para isso, foram utilizadas como tratamento de colisões as árvores binárias de pesquisa, que além de auxiliar na busca por um e-mail posteriormente, permite a inserção com custo relativamente baixo em seu caso médio.

Árvore binária de pesquisa

O segundo algoritmo de pesquisa utilizado possui funcionamento simples, nesse caso, diferente do método Hashing que utiliza de transformações sobre a chave de pesquisa, temos a utilização direta dessa e efetuam-se comparações para encontrar a chave desejada.

O funcionamento da árvore de busca segue o seguinte funcionamento.

- Nó com chave maior que a chave pesquisada: prossegue-se a busca na subárvore da esquerda desse nó.
- Nó com chave menor que a chave pesquisada: prossegue-se a busca na subárvore da direita desse nó.
- Nó com chave igual: em uma árvore binária de pesquisa comum nessa etapa já foi encontrada a chave desejada, entretanto, para tratar possíveis erros, verifica-se também se o código do InBox coincide com o buscado, assim tem-se o nó buscado
- Nó com chave igual, mas com código InBox distinto: na maioria das árvores binárias de busca não há essa dupla verificação ao encontrar a chave, mas a fim de tratar erro é realizado essa comparação. Isso ocorre pois, apesar de não haver inserções com mensagens de mesmo ID pode haver pesquisas que não respeitem tal critério.

A remoção de elementos nessas árvores ocorrem da seguinte forma:

- Nó removido é uma folha: basta eliminar o nó
- Nó removido tem um filho: substitui a posição do nó pelo seu filho e elimina o nó desejado
- Nó removido tem dois filhos: troca pelo seu antecessor, ou seja, o elemento mais à direita da subárvore da direita do nó a ser removido e remove o elemento desejado.

O funcionamento da inserção de elementos segue a mesma ideia da busca, entretanto, busca-se a posição vazia onde será adicionada o novo nó, respeitando as regras da árvore binária de pesquisa.

Para toda a análise do programa implementado em linguagem C/C++ (predominantemente C++) e compilado através da ferramenta gdb. Foi utilizado um computador com sistema operacional Linux em um processador Intel® Core™ i5-10210U CPU @ 1.60GHz × 8 com 8Gb de memória RAM.

3. Análise de Complexidade

O algoritmo como um todo possui um custo de tempo e espaço devido às operações realizadas com os elementos da entrada. Portanto, é importante analisar cada função envolvida nessas execuções a fim de verificar se é uma solução ótima para tal caso ou se pode ser substituída por outra função que tornaria a resolução do problema mais eficiente, de forma a balancear os gastos e evitar utilizações de memória e/ou tempo maiores que o realmente necessário.

As funções Insert, Search e Erase chamam suas respectivas formas recursivas, portanto, apenas essa segunda parte será analisada.

Função RecursiveInsert - complexidade de tempo - essa função encontra a posição do novo nó a ser inserido na árvore. Dessa forma, tratando-se de uma árvore binária de pesquisa existem 3 cenários, O(1) se a raiz da árvore é nula, O(log(n)), caso a árvore esteja balanceada e O(n) caso esteja desbalanceada e comporte-se como uma lista encadeada. Portanto, temos uma função com ordem de complexidade que possui melhor, pior e caso médio, sendo eles, respectivamente $\Theta(1)$, $\Theta(\log(n))$ e $\Theta(n)$.

Função RecursiveInsert - complexidade de espaço - essa função realiza chamadas recursivas até encontrar a posição desejada, portanto são empilhadas chamadas de função na memória, nesse caso, conforme citado anteriormente, tem-se caso médio, pior e melhor caso, assim, complexidade assintótica de espaço dessa função é $\Theta(1)$ no melhor caso, $\Theta(\log(n))$ no caso médio e $\Theta(n)$ no pior caso.

Função RecursiveSearch - complexidade de tempo - essa função encontra a chave desejada. Dessa forma, tratando-se de uma árvore binária de pesquisa existem 3 cenários, O(1) se a raiz da árvore é a chave procurada, O(log(n)), caso a árvore esteja balanceada e O(n) caso esteja desbalanceada e comporte-se como uma lista encadeada e a chave procurada esteja no final da "lista". Portanto, temos uma função com ordem de complexidade de tempo que possui melhor, pior e caso médio, sendo eles, respectivamente $\Theta(1)$, $\Theta(\log(n))$ e $\Theta(n)$.

Função RecursiveSearch - complexidade de espaço - essa função realiza chamadas recursivas até encontrar a posição desejada, portanto são empilhadas chamadas de função na memória, nesse caso, conforme citado anteriormente, tem-se caso médio, pior e melhor caso, assim, complexidade assintótica de espaço dessa função é $\Theta(1)$ no melhor caso, $\Theta(\log(n))$ no caso médio e $\Theta(n)$ no pior caso.

Função RecursiveErase - complexidade de tempo - essa função encontra a chave a ser removida. Dessa forma, tratando-se de uma árvore binária de pesquisa existem 3 cenários, O(1) se a árvore é composta so de 1 nó, O(log(n)), caso a árvore esteja balanceada e O(n) caso esteja desbalanceada e comporte-se como uma lista encadeada e a chave a ser removida esteja no final da "lista". Portanto, temos uma função com ordem de complexidade de tempo que possui melhor, pior e caso médio, sendo eles, respectivamente $\Theta(1)$, $\Theta(\log(n))$ e $\Theta(n)$.

Função RecursiveErase - complexidade de espaço - essa função realiza chamadas recursivas até encontrar o nó, portanto, são empilhadas chamadas de função na memória, nesse caso, conforme citado anteriormente, tem-se caso médio, pior e melhor caso, assim, complexidade assintótica de espaço dessa função é $\Theta(1)$ no melhor caso, $\Theta(\log(n))$ no caso médio e $\Theta(n)$ no pior caso.

Função Predecessor - complexidade de tempo - essa função realiza chamadas recursivas até encontrar o antecessor do elemento desejado. Dessa forma, ele possui um comportamento variável, caso o antecessor seja o filho do elemento desejado temos um custo O(1), caso a subárvore da esquerda esteja balanceada e suficientemente grande e o pior caso quando o elemento antecessor esteja em uma lista gerada na subárvore à esquerda do nó a ser removido. Portanto, temos uma função com ordem de complexidade de tempo que possui melhor, pior e caso médio, sendo eles, respectivamente $\Theta(1)$, $\Theta(\log(n))$ e $\Theta(n)$.

Função Predecessor - complexidade de espaço - essa função realiza chamadas recursivas até encontrar o nó, portanto, são empilhadas chamadas de função na memória, nesse caso, conforme citado anteriormente, tem-se caso médio, pior e melhor caso, assim, complexidade assintótica de espaço dessa função é $\Theta(1)$ no melhor caso, $\Theta(\log(n))$ no caso médio e $\Theta(n)$ no pior caso.

Função CleanTree - complexidade de tempo - essa função realiza chamadas recursivas eliminando todos os nós das árvores. Portanto, temos um comportamento O(1) caso a árvore tenha apenas a raiz ou esteja vazia e O(n) caso contrário. Portanto, temos uma função com ordem de complexidade de tempo que possui melhor e pior caso, sendo eles, respectivamente $\Theta(1)$ e $\Theta(n)$.

Função CleanTree - complexidade de espaço - essa função realiza chamadas recursivas até eliminar o todos os nós da árvore, incluindo a raiz. Nesse cenário, como citado anteriormente, temos 2 situações. Logo, a complexidade assintótica de espaço dessa função é $\Theta(1)$ no melhor caso e $\Theta(n)$ no pior caso.

Função ~BinaryTree - complexidade de tempo - essa função chama a função CleanTree que realiza as atividades de destrução da árvore binária, logo o custo de tempo da ~BinaryTree depende diretamente da complexidade de tempo da CleanTree, sendo essa variável. Portanto, temos uma função com ordem de complexidade de tempo que possui melhor e pior caso, sendo eles, respectivamente $\Theta(1)$ e $\Theta(n)$, dependendo do comportamento da função chamada.

Função ~BinaryTree - complexidade de espaço - essa função em si não possui custo de espaço, pois realiza atividades constantes durante todo o processo. Porém realiza a chamada da função CleanTree que possui custo de tempo variável. Logo, a complexidade assintótica de espaço dessa função é $\Theta(1)$ no melhor caso e $\Theta(n)$ no pior caso, dependendo do comportamento da CleanTree.

Função AccessHash - complexidade de tempo - essa função percorre todos os endereços da tabela Hash acessando as árvores binárias presentes nela de forma a acessar todos os elementos existentes até então. Logo, para uma tabela Hash já inicializada, temos um custo O(n), no melhor caso, ou seja, quando não foram criadas árvores em cada partição da tabela e $\Theta(n^2)$ quando cada árvore das partições da lista já tenham sido inicializadas.

Função AccessHash - complexidade de espaço - essa função percorre os endereços da tabela Hash e para cada um deles acessa os elementos da sua respectiva árvore. A chamada da função de acesso às árvores é feita de forma recursiva, logo, no melhor caso, temos n chamadas da função, referentes a cada endereço da árvore, entretanto, no pior caso, temos n chamadas recursivas da função que acessa a árvore. Logo, a complexidade assintótica de espaço dessa função é $\Theta(n)$ no melhor caso e $\Theta(n^2)$ no pior..

Função getHashSize - complexidade de tempo - essa função realiza uma operação constante, logo a complexidade assintótica de tempo dessa função é $\Theta(1)$.

Função getHashSize - complexidade de espaço - essa função realiza uma operação constante obtendo o tamanho da tabela Hash do arquivo, logo a complexidade assintótica de espaço dessa função é $\Theta(1)$.

Função getTexts - complexidade de tempo - essa função obtém todas as ações a serem realizadas no arquivo, além de chamar as funções responsáveis por armazenar os dados. O custo, portanto, é diretamente proporcional ao maior custo das funções realizadas, sendo esse o O(n) da inserção, remoção ou pesquisa. Entretanto essas realizações podem ser feitas n vezes durante o decorrer do arquivo. Dessa forma, a função tem ordem de complexidade de tempo $\Theta(n^2)$.

Função getTexts - complexidade de espaço - essa função obtém todas as ações a serem realizadas no arquivo, além de chamar as funções responsáveis por armazenar os dados. O custo, portanto, é diretamente proporcional ao maior custo das funções realizadas, sendo esse o O(n) da inserção, remoção ou pesquisa. Entretanto essas realizações podem ser feitas n vezes durante o decorrer do arquivo. Dessa forma, a função tem ordem de complexidade de espaço $\Theta(n^2)$ que é respectiva as chamadas recursivas das funções citadas anteriormente realizadas n vezes.

Programa total - complexidade de tempo - como citado uma das ações mais relevantes realizadas no programa é a getTexts que realiza quase todas as funções do programa. Porém tem-se também os acessos realizados, que possuem custo semelhante ao getTexts. Por tal motivo, a função tem ordem de complexidade de tempo $\Theta(n^2)$.

Programa total - complexidade de espaço - como citado, o programa é quase todo "comandado" pela função getTexts, e a complexidade de tempo também segue tal critério. Portanto, a função tem ordem de complexidade de espaço $\Theta(n^2)$.

4. Estratégias de Robustez

Os mecanismos de programação defensiva e tolerância a falhas são de extrema importância na realização de programas, isso porque evita que o usuário conceda entradas incorretas, utilize dessas falhas para danificar de alguma forma o programa, ou obtenha resultados inesperados na execução das funções do programa.

Nessa etapa foram utilizadas as funções __erroassert e __avisoassert para informar ao usuário problemas durante a execução do programa ou por introdução de valores incorretos.

Nome do arquivo de entrada

Ao inicializar o programa é necessário passar o arquivo no qual os dados utilizados pelo programa estão, dessa forma, é necessário verificar se existe a passagem do nome do arquivo para sua futura abertura.

Abertura dos arquivos de entrada e saída

Na implementação desse programa é necessário obter dados de um arquivo de entrada, além de gravar os resultados obtidos em um arquivo de saída, ou seja, a utilização de arquivos é de extrema importância no decorrer do programa. Entretanto, podem haver falhas na abertura desses, o que impossibilita a execução do programa. Portanto, é de suma importância verificar se os arquivos foram abertos corretamente para utilização no programa.

Tamanho da tabela Hash

O programa usa a tabela Hash como método de pesquisa, logo manter essa tabela possível de armazenar dados é imprescindível para o decorrer do programa. Portanto, o tamanho dela deve ser maior que zero.

Fechamento dos arquivos

Durante o programa é fundamental a utilização de arquivos, porém ao findar da execução é necessário que os arquivos sejam fechados. Entretanto, a falha no fechamento não é fundamental para o prosseguimento do programa, mas algo inesperado. Portanto, é necessário alertar o usuário quanto a situação, porém não há necessidade de encerrar o programa como um todo.

Parâmetros de armazenamento

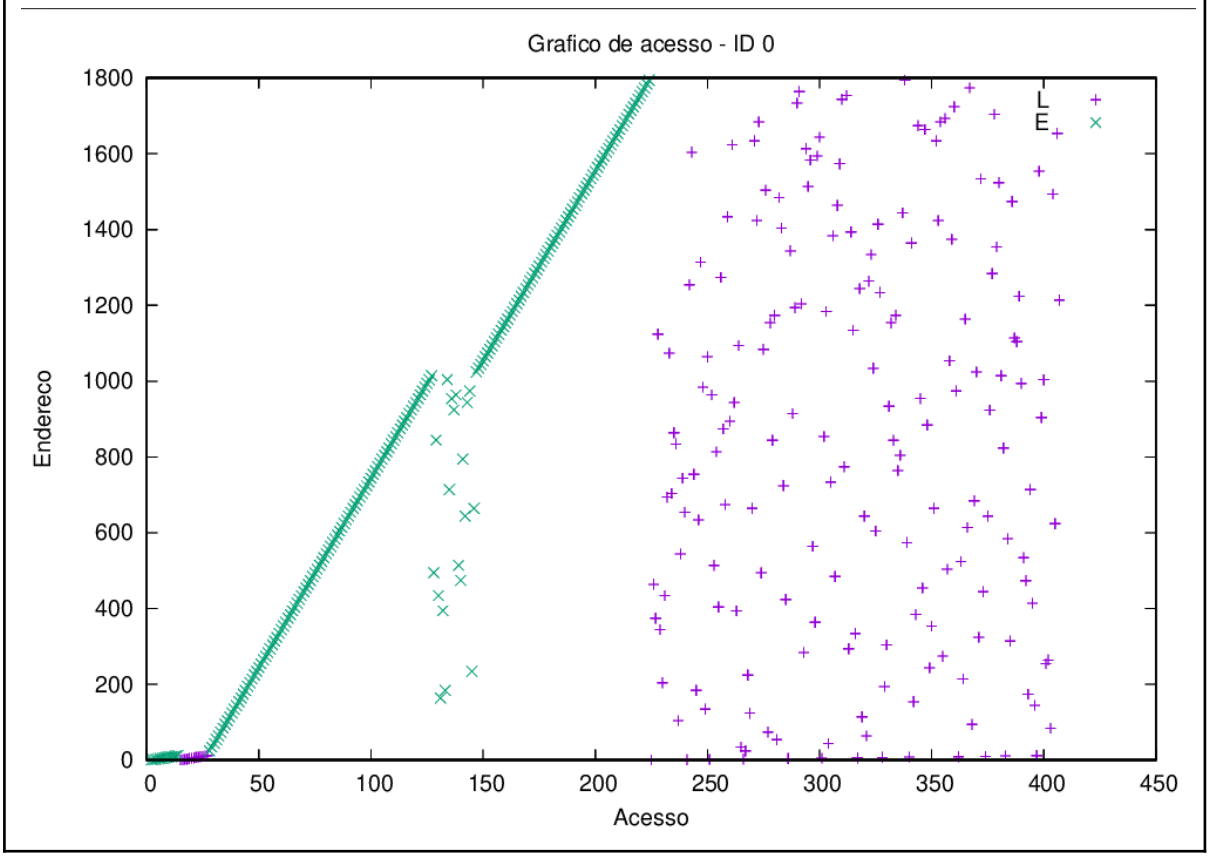
Durante o programa é fundamental os códigos que indicam a posição dos dados, por isso, é importante verificar se os códigos passados são válidos, ou seja, inteiros e maiores que zero. Entretanto, a tentativa de inserção de um dado com chaves negativas não é fundamental para o prosseguimento do programa, mas algo inesperado. Portanto, é necessário alertar o usuário quanto a situação, porém não há necessidade de encerrar o programa como um todo, apenas não inserir tal dado e alertar caso tentem buscá-lo ou removê-lo.

5. Análise Experimental

O programa possui dois mecanismos de armazenamento funcionando em conjunto. Com base nisso, será feita a análise de desempenho computacional e localidade de referência dos experimentos realizados.

O algoritmo de armazenamento e pesquisa Hashing possui um custo constante tanto em armazenamento quanto em pesquisa. Além disso, durante a inserção das mensagens na árvore binária de pesquisa, que como citado possui custos variáveis, dependendo da situação da árvore. Com base nisso, podemos analisar os mapas de acesso da função para uma entrada com 100 mensagens inseridas.

Operações com 100 Mensagens - Tabela Hash de tamanho 13
Mapa de acesso



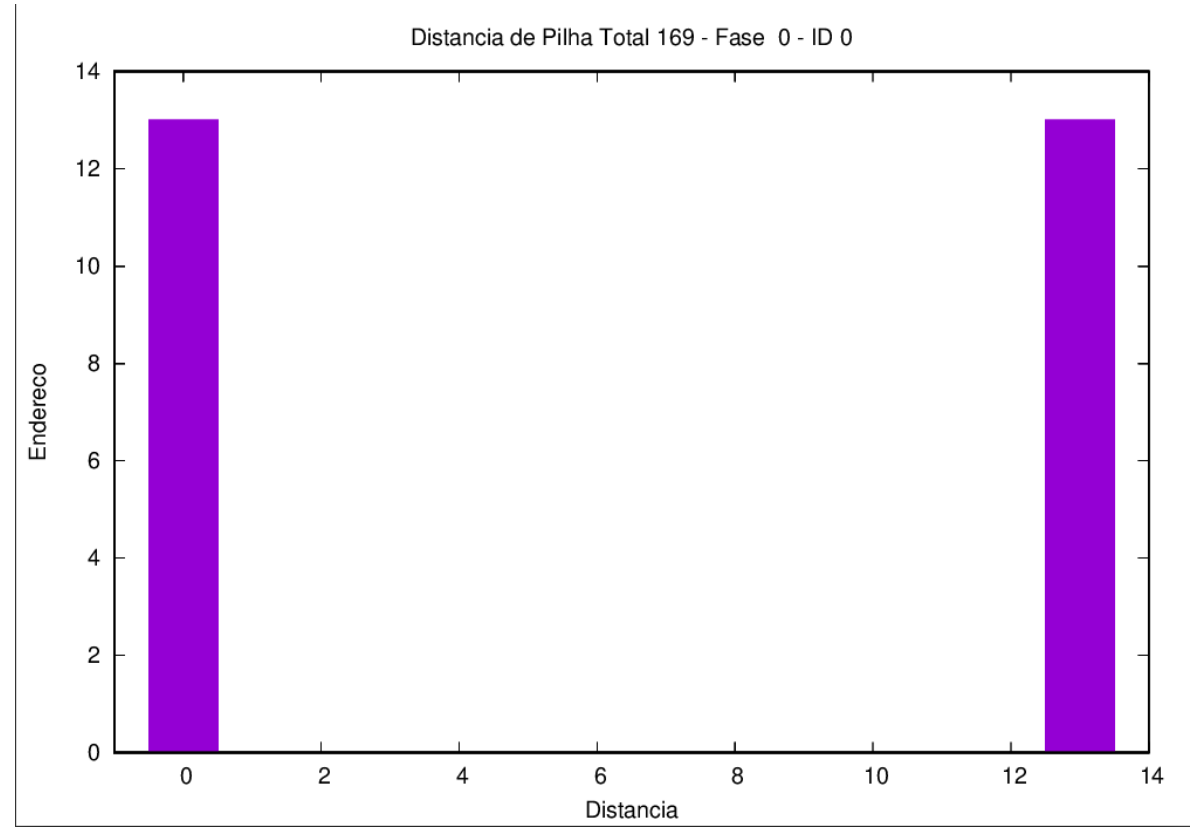
No mapa de acesso, inicialmente tem-se a criação da tabela Hash com 13 endereços e posteriormente o acesso aos seus dados. Como, de início, não há inserção de nenhuma mensagem, o comportamento limita-se aos 13 endereços iniciais.

Na segunda etapa, tem-se as 100 inserções, por basear em ponteiros elas possuem inicialmente um comportamento linear, pois são inseridas em tempos próximos. Porém posteriormente

as árvores começam a ficar preenchidas e elementos são armazenados como filhos de outros, que já tinham sido iniciados como nulo, por isso, algumas inserções variadas no meio das inserções.

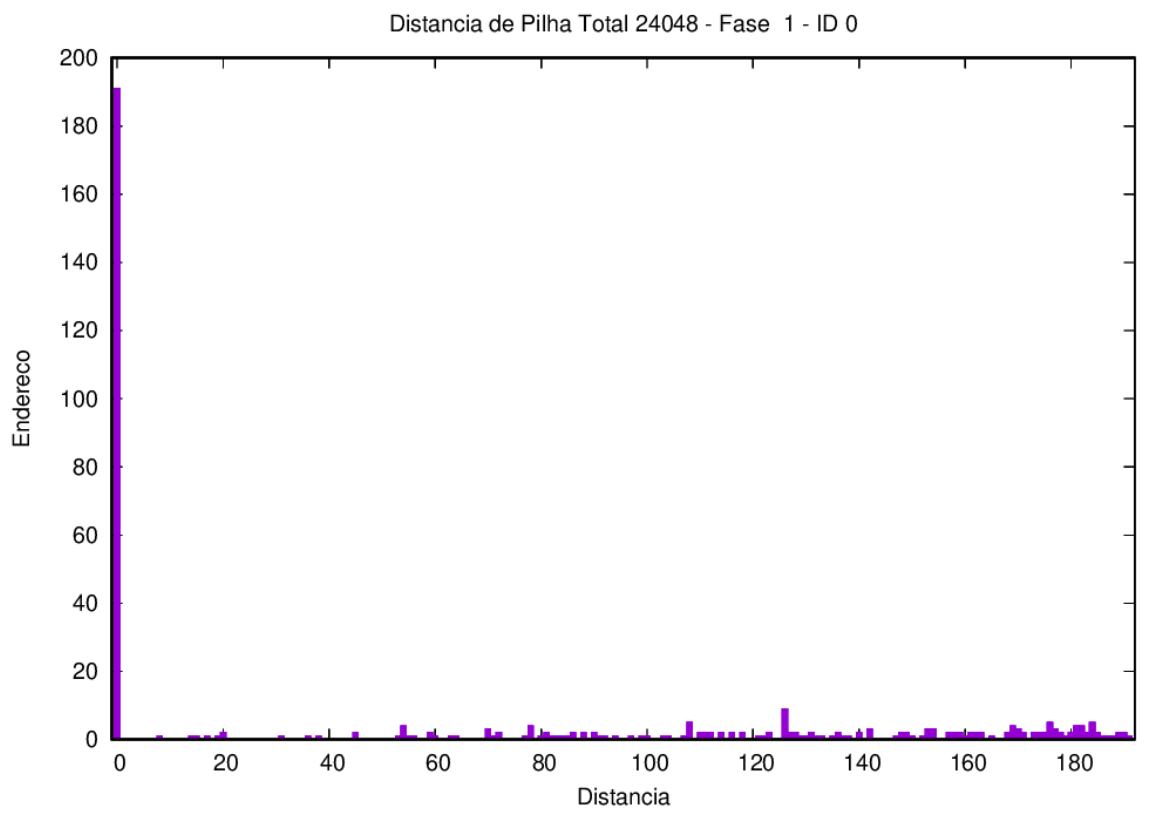
Por fim, temos o acesso a cada árvore de cada endereço da tabela, tal acesso ocorre em ordem, ou seja primeiro o elemento da esquerda é acessado, depois o nó pai e depois o elemento da direita. Como o armazenamento dos dados das árvores na memória segue a proximidade de tempo, ao acessarmos sequencialmente as tabelas temos um comportamento aleatório na memória.

Tabela Hash - Inicialização



Nessa etapa, por haver apenas a inicialização e o acesso, temos inicialmente todas as distâncias de pilha igual a 0 e posteriormente o primeiro elemento a ser inserido é, também o primeiro a ser acessado, por ser uma pilha esse elemento se encontra a maior distância do topo, tem-se, por isso, 13 acessos à distância de pilha 0 e posteriormente 13 acessos a distância de pilha 13.

Tabela Hash + ABP - Ações na Memória Primária

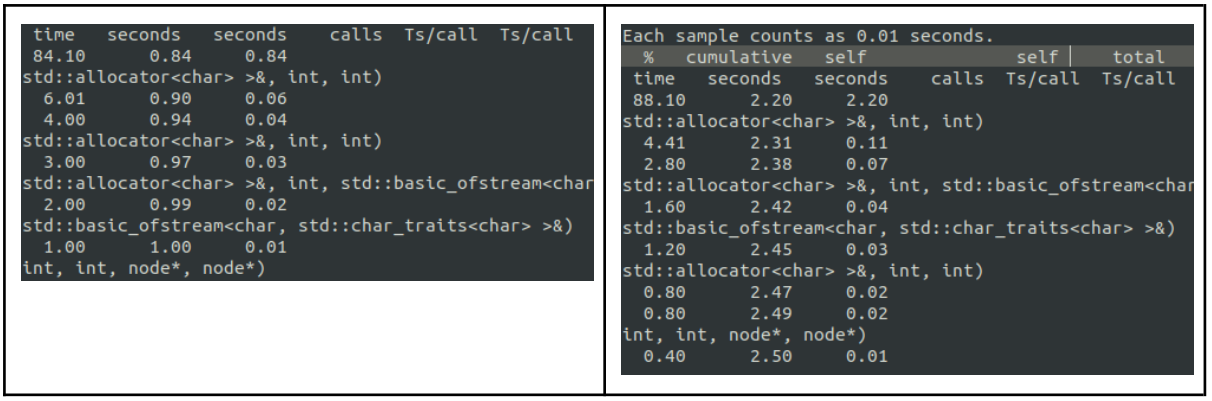


Na etapa das operações tem-se, inicialmente, o acesso a cada endereço da tabela sequencialmente e o acesso em ordem das suas respectivas ABP, empilhando os acessos à distância 0. Posteriormente temos as operações de remoção e pesquisa nas tabelas, porém tais ações não possuem uma sequência fixa, gerando acessos a elementos que estão a distâncias de pilha não bem definidas.

Tempo de execução

Comparando as execuções das operações para arquivos com grandes quantidades de texto, podemos ver que o desempenho do quicksort foi superior ao do selection sort. Podemos visualizar dois casos abaixo, para um arquivo com 40 mil linhas e outro com 120 mil linhas.

200 mil mensagens + 10 mil pesquisas + 4 mil remoções	**500 mil mensagens + 10 mil pesquisas + 4 mil remoções**
time seconds seconds calls Ts/call Ts/call nam 70.91 0.17 0.17 Bi std::allocator<char> >&, int, int) 8.34 0.19 0.02 Bi 8.34 0.21 0.02 Li std::basic_ofstream<char, std::char_traits<char> >&) 8.34 0.23 0.02 Has std::allocator<char> >&, int, std::basic_ofstream<char, s 4.17 0.24 0.01 Ema	time seconds seconds calls Ts/call Ts/call 80.09 0.32 0.32 std::allocator<char> >&, int, int) 5.01 0.34 0.02 5.01 0.36 0.02 5.01 0.38 0.02 std::allocator<char> >&, int, std::basic_ofstream<char 2.50 0.39 0.01 2.50 0.40 0.01
1 milhão de mensagens + 5 mil pesquisas + 2 mil remoções	**2 milhões de mensagens + 5 mil pesquisas + 2 mil remoções**



```
 time   seconds   seconds    calls  Ts/call  Ts/call
 84.10      0.84     0.84
std::allocator<char> >&, int, int)
  6.01      0.90     0.06
  4.00      0.94     0.04
std::allocator<char> >&, int, int)
  3.00      0.97     0.03
std::allocator<char> >&, int, std::basic_ofstream<char
  2.00      0.99     0.02
std::basic_ofstream<char, std::char_traits<char> >&)
  1.00      1.00     0.01
int, int, node*, node*)
```

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call
 88.10      2.20     2.20
std::allocator<char> >&, int, int)
  4.41      2.31     0.11
  2.80      2.38     0.07
std::allocator<char> >&, int, std::basic_ofstream<char
  1.60      2.42     0.04
std::basic_ofstream<char, std::char_traits<char> >&)
  1.20      2.45     0.03
std::allocator<char> >&, int, int)
  0.80      2.47     0.02
  0.80      2.49     0.02
int, int, node*, node*)
  0.40      2.50     0.01
```

Com base nisso, é perceptível a influência da quantidade de dados inseridos e o tempo de execução do programa e além disso, temos um comportamento quase linear, isso ocorre pois em situações temos um tempo log(n), enquanto outras podemos ter um custo de tempo $O(n^2)$, entretanto, em média tem-se, nesse programa, um comportamento linear.

6. Conclusão

Este trabalho lidou com o problema de algoritmos de pesquisa e armazenamento de dados, com tal intenção foi necessário armazenar dados com base em uma nos códigos da mensagem e da caixa de entrada desejada. Assim, a abordagem utilizada para armazenamento/pesquisa das mensagens contidas no arquivo foram o Hashing e a ABP.

Com a solução foi possível verificar a eficiência de cada um dos métodos de forma a utilizá-los quando for mais conveniente e útil. Além disso, foi de extrema importância verificar formas de tornar métodos já existentes ainda mais eficientes, seja usando árvores binárias de pesquisa como tratamento de colisões, forma como foi tratada nesse caso; seja utilizando a tabela hash e tornando as partições de armazenamento ainda menores evitando gerar uma grande árvore com vários dados, nesse sentido a busca por informações se torna mais eficiente. Dentre outras ferramentas que agregam a visão analítica de problemas e métodos já existentes.

Por meio da resolução desse trabalho, foi possível praticar conceitos relacionados a algoritmos de pesquisa e armazenamento, estruturas de dados, resoluções de problemas e análise de complexidade. Além disso, permitiu diferenciar ainda mais a perspectiva de apenas programar para resolver problemas, para programar com o objetivo de encontrar a solução mais eficaz para os problemas, entendendo a implementação e seu comportamento na memória e em custo de tempo.

Durante a implementação da solução para o problema, houveram importantes desafios a serem superados, por exemplo a utilização da árvore binária de pesquisas e sua implementação, principalmente ligado a utilização dos ponteiros. Entretanto, com o decorrer do processo e insistência na sua resolução foi possível agregar muito conhecimento no âmbito de programação de computadores.

7. Referências

www.stackoverflow.com

Ziviani, N. (2006). Projetos de Algoritmos com Implementações em Pascal e C

Cormen, T. Leiserson, C, Rivest, R., Stein, C.(2009) Introduction to Algorithms

Aulas e materiais disponibilizados no moodle UFMG.

8. Instruções para compilação e execução

Na pasta TP há um arquivo makefile responsável por compilar os testes e análises de desempenho e localidade do programa. Basta que, no terminal, seja digitado make para rodar todos os testes.

Por padrão o main.cpp abre o arquivo in.txt na pasta TP para executar o programa e sua saída estará na mesma pasta no arquivo out.txt, entretanto, todos esses dados podem ser alterados na chamada do programa. Além das opções definidas na descrição do TP, no qual define-se o nome do arquivo de entrada e saída, foram adicionadas também mais duas opções a -[p|P] e a -[l|L], nessas são possíveis definir, respectivamente, o endereço de saída do registro de desempenho e se a opção de padrão de acesso e localidade será ativada, para ativá-la basta que a opção -[l|L] seja descrita ao executar o programa.

Para o padrão de acesso e localidade a pasta analisamem disponibilizada pelo Wagner/Gisele no moodle deve estar na mesma pasta que contém a pasta TP .

Por padrão, no Makefile, todos os arquivos de saída são gerados em pastas separadas na /tmp/.

A chamada da função pode ser feita da mesma forma que foi definida na especificação do TP2 disponibilizada no moodle. Porém, tem-se por padrão do programa:

- Arquivo de entrada - “/TP/in.txt”
- Arquivo de saída - “/TP/out.txt”
- Arquivo de saída de análise do registro de desempenho - “”
- Opção de padrão de acesso e localidade - false