Trabalho Prático 3

DCC214 - Estrutura de Dados

Igor Joaquim da Silva Costa

1. Introdução

O problema proposto foi implementar um simulador de um servidor de emails, usando conceitos de tabela hash e árvores binárias.

Para resolver o problema citado, foi implementado uma tabela hash simples, onde o id do usuário seria a chave para armazenar um email na tabela. Além disso, a fim de melhorar o tempo de execução, todo email na mesma posição da tabela é armazenado dentro de uma árvore binária AVL, onde o id da mensagem é a chave.

Diante do exposto, a documentação presente possui como objetivo detalhar como o sistema foi implementado (Seção 2), o quão eficiente ele pode ser em termos teóricos(Seção 3) e aplicados (Seção 5), além de explicar como o programa lida com possíveis inconsistências durante sua execução (Seção 4). Por fim, o projeto é sumarizado junto com os aprendizados gerados durante a produção do trabalho(Seção 6).

2. Método

Esta seção tem como objetivo discutir as decisões que levaram à atual implementação do programa.

2.1 Fluxo principal

Antes de detalhar como o programa opera sobre um texto, vale discorrer de maneira textual e intuitiva o que o programa faz a cada passo de execução. Primeiramente, o sistema é capaz de efetuar 3 operações, inserção, busca e remoção. Diante disso, para cada linha do arquivo de entrada, é avaliado qual operação deve ser executada. Após isso, é feita a operação corresponde e o resultado é armazenado no arquivo de saida.

2.2 Classes

A fim de suportar as operações descritas acima, são necessárias estruturas robustas o suficiente para guardar emails, ids, árvores binárias e tabelas hash. Nesse sentido, foi implementado uma classe para cada elemento do mini-mundo de interesse. A seguir, segue a definição das mesmas:

2.2.1 Email

Um email é composto por seus ids de usuário e de mensagem, e da mensagem propriamente dita. Cada email possui uma lista de N palavras chamada mensagem, representando o texto lido.

2.2.2 AVL

Como são necessárias diversas consultas e remoções durante o programa, tornou-se válida a implementação de uma árvore AVL. Nesse sentido, uma AVL é um conjunto de nós (node_t), onde cada nó possui uma referência aos seus filhos, seu pai e ao conteúdo armazenado, que nesse caso é o email. Ainda assim, um nó é responsável por calcular a sua própria altura e a altura dos seus antepassados, além de conhecer o seu fator de balanceamento.

Dito isso, uma AVL é uma árvore binária balanceada, onde, a cada operação, uma operação extra de balanceamento é executada, a fim de manter a propriedade descrita. Sendo assim, a AVL possui 4 métodos principais: Inserção, remoção, pesquisa e balanceamento. Após cada inserção e remoção, a função balanceamento é chamada. Para executar o balanceamento, são usadas 2 funções auxiliares de rotação, que são performadas segundo o caso que o balanceamento da AVL se encontra.

No que tange a inserção e remoção, os possíveis casos de desbalanceamento estão descritos no código fonte da árvore.

2.2.3 Tabela Hash

Para representar a tabela Hash, foi implementada uma lista dinâmica de AVLS, de tamanho N. Sendo assim, cada id de usuário possui uma posição na tabela, dada pela função de hash id%N.

3. Análise de complexidade

3.1 Espaço

Cada email possui 3 inteiros de controle e uma lista de palavras de tamanho máximo 200. Cada nó possui uma referência ao seu conteudo e à seus vizinhos, uma árvore é um conjunto de M nós, uma tabela possui B árvores. Considerando que toda posição da tabela hash tenha o mesmo número M de nós, o espaço gasto é $O(M * B)$. Ainda assim se N for o tamanho da entrada e cada árvore tiver M nós, M = N/B, ou seja, $O(N)$ no tamanho da entrada.

3.1.2 Chamadas recursivas

Como são feitas algumas chamadas recursivas durante a execução do programa, suas execuções podem ocupar grande espaço na memória, se o N for suficientemente grande. Para o balanceamento e a operação de atualizar altura de um nó, o nó atual segue fazendo chamadas recursivas até alcançar a raiz, sabe-se que a distância entre um nó folha de uma árvore balanceada até a raiz é de $O(log(N))$. Nesse caso, cada inserção/remoção preenche cerca de $O(log(N))$ de espaço, durante a execução do progrma.

3.2 Tempo

Para análise de tempo, considere N como o número de elementos na árvore.

3.2.1 Inserção

Para se realizar a inserção, é feita uma busca binária pela posição que o novo elemento deve ser inserido. Considerando a comparação como operação mais significativa, é conhecido que tal busca efetua $O(log(N))$ comparações. Entretanto, após inserir um nó na árvore AVL, deve-se recalcular as alturas do novo nó até a raiz, além de rebalancear a árvore. Para atualizar a altura de um único nó, ou executar seu balanceamento, gasta-se $O(1)$, já que essas operações só acessam elementos que já estão armazenados dentro da estrutura nó. Para executar essas operações do nó até a raiz, são gastos mais $O(log(N))$ passos. Ainda assim, a complexidade geral da inserção se mantém $O(log(N))$.

3.2.2 Busca

Considerando a comparação como a operação mais significativa, que toda buscar um elemento performa o mesmo passo a passo da inserção. Dessa forma, a busca é $O(log(N))$.

3.2.3 Remoção

Para se realizar a remoção, primeiro é necessário buscar o elemento a ser removido. Considerando a comparação como operação mais significativa, é conhecido que tal busca efetua $O(log(N))$ comparações, dado o tópico anterior. Nesse sentido, se o elemento não foi encontrado, a remoção termina aqui. Caso o elemento seja encontrado, existem 2 casos a se considerar. Se o elemento for nó folha, ou tiver só um filho, a remoção executa mais O(1) operações, já que só são executadas operações que acessam elementos que já estão armazenados dentro da estrutura nó. No caso do nó com 2 filhos, a estratégia usada foi substituí-lo pelo seu antecessor, ou seja, o maior elemento da subárvore à esquerda do nó. Encontrar esse elemento também ocorre $O(log(N))$, já que estamos buscando o maior elemento de uma árvore binária balanceada. Ainda assim, a complexidade do pior caso, caso médio e melhor caso da remoção se mantém $O(log(N))$.

3.2.4 Complexidade Geral.

Dito isso, a complexidade geral da ordenação corresponde a quantidade de operações executadas, vezes um fator de log(N), que corresponde ao custo de executar tal operação. Sendo B o tamanho de operações presentes no arquivo de entrada, o programa executa em

$$O(B * log(N))$$

Onde N é a média dos tamanhos das árvores.

Por fim, as operações de inserção, remoção e consulta na tabela hash ocorrem na mesma complexidade que as mesmas operações na árvore AVL.

4. Estratégias de Robustez

Um problema encontrado durante a implementação do sistema foi lidar com arquivos de entrada grandes. Dessa forma, arquivos com mais de 50000 linhas podem apresentar erros de leitura durante a execução do programa.

5. Análise Experimental

A análise experimental a seguir tem como objetivo medir o quão eficiente é o sistema implementado usando duas métricas, o desempenho computacional - quão rápido o programa é executado com entradas grandes - e análises de acesso em memória.

5.1 Desempenho computacional

5.1.1 Perfil de execução

Para testar o desempenho computacional, primeiramente, o programa foi compilado em estado de "profiling", a fim de analisar quais funções consomem relativamente mais tempo durante a execução do programa. Diante disso, foram feitas baterias de testes a partir de entradas geradas aleatoriamente por um programa python implementado pelo aluno. O teste a seguir foi feito por uma entrada composta por 1000 inserções, 100 remoções e 1500 buscas. Com o profile das execuções pronto, cada execução é processada pelo programa “gprof”, uma ferramenta que auxilia na análise do desempenho computacional. Dito isso, segue a análise das chamadas de funções.

Tempo de execução (%)	Calls	
45.51	15051	arvore::pesquisa(int, int)
18.20	172965	int std::max<int>(int const&, int const&)
18.20	*10001*	*arvore::insere(email*)*
9.10	*10001*	*tabelaH::insere(email*)*
9.10	*1*	*main*
0.00	*235449*	*node_t::fator_balanceamento()*
0.00	*19301*	*node_t::atualiza_altura()*
0.00	*10101*	*arvore::balanceia(node_t*)*

Para evitar redundância, os outros perfis de execução apresentam o mesmo comportamento do teste apresentado acima. No que tange a análise dos experimentos, fica evidente que a função que mais demanda tempo, em porcentagem, é a função para pesquisar o elemento na árvore, visto que ela foi a operação mais abundante no arquivo de entrada, além

de ser necessária para executar a remoção. Para além disso, a função apresenta complexidade logarítmica no tamanho médio da árvore, gastando cerca de $1500 * log(N)$ em tempo de execução durante o programa, usando a fórmula desenvolvida na seção 3.2.2. Dessa forma, a função de pesquisa estar em primeiro lugar não é de grande surpresa, embora existam formas de diminuir esse tempo de execução usando de técnicas como hashing.

A função que ocupa a segunda posição é a função que seleciona o máximo entre dois valores inteiros, usado durante a atualização da altura da árvore. Tal função gastar mais tempo de execução que a própria função de atualizar altura revela que talvez existam formas mais eficientes de recuperar o máximo entre dois valores, melhorando a performance do programa.

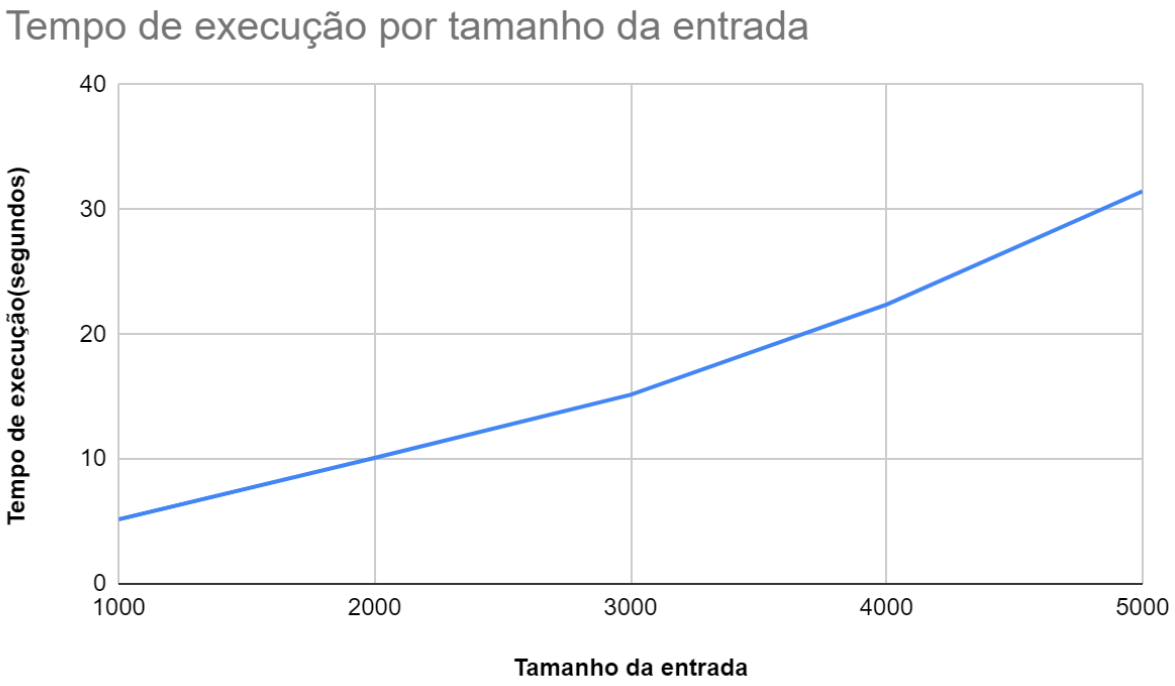
O terceiro e quarto lugar se tratam da inserção de elementos na tabela. Como esperado, elas acontecem todas as vezes que existe a inserção, sendo justo elas ficarem abaixo da operação de busca no perfil de execução, visto que acontecem 50% mais buscas do que inserções no teste.

Embora as funções de atualizar altura e balanceamento sejam chamadas várias vezes durante o programa, elas representam um tempo de execução quase insignificante. Tal evento pode ser explicado pelo fato de ser raro acontecer uma inserção ou remoção que desbalanceie a árvore de forma a ser custoso ocorrer o balanceamento. Ainda assim, a função de calcular o fator de desbalanceamento foi executada o maior número de vezes, o que também não é estranho.

Nesse sentido, conclui-se que ainda existem margens para a melhora no desempenho do programa, a nível de chamadas de funções, visto que muitas delas poderiam ser feitas de outras formas, a fim de melhor utilizar tempo de execução do programa.

5.1.2 Desempenho em termos da entrada

Além do tempo gasto pelas chamadas de funções, deve-se medir a relação entre o aumento do volume da entrada do programa com o seu tempo de execução. Para realizar tal objetivo, foi utilizada a ferramenta Memlog do professor Wagner Meira e ferramentas para a geração de gráficos. A seguir, o gráfico com os resultados.



Como visto acima, o tempo de execução cresce muito próximo a N, onde N é a quantidade de operações feitas assim como foi discutido na seção de análise de complexidade.

5.2 Análise de localidade de referência e acesso de memória

Como explicado em aula, programas eficientes em uso de memória são aqueles que tendem a acessar regiões de memória contíguas, já que essa prática é otimizada pelos sistemas operacionais onde o programa é executado. Tal propriedade é chamada de localidade de referência e é uma ótima métrica de como o programa acessa a memória por ele utilizada.

Nesse sentido, munido das ferramentas Analisamem e Memlog, disponibilizadas pelo professor Wagner Meira, é possível gerar gráficos que demonstram o acesso de memória do programa em suas diversas fases, e, assim, analisar se o acesso de memória do programa segue as boas práticas desejadas. Dito isso, segue a análise de acesso de memória e localidade de referência, onde o id 0 representa a inserção dos elementos na árvore, o id 1 os acessos durante o balanceamento, o id 2 como os acessos durante pesquisas e o id 3 os acessos feitos durante a remoção.

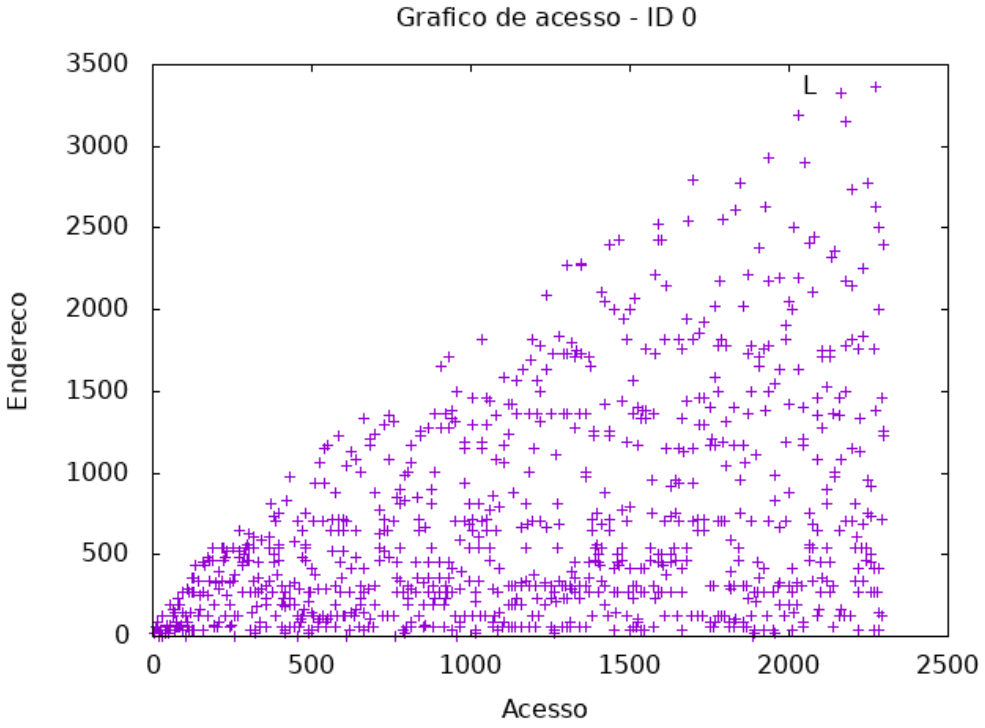


Grafico de acesso - ID 0
Endereco
Acesso
L
0
500
1000
1500
2000
2500
3000
3500

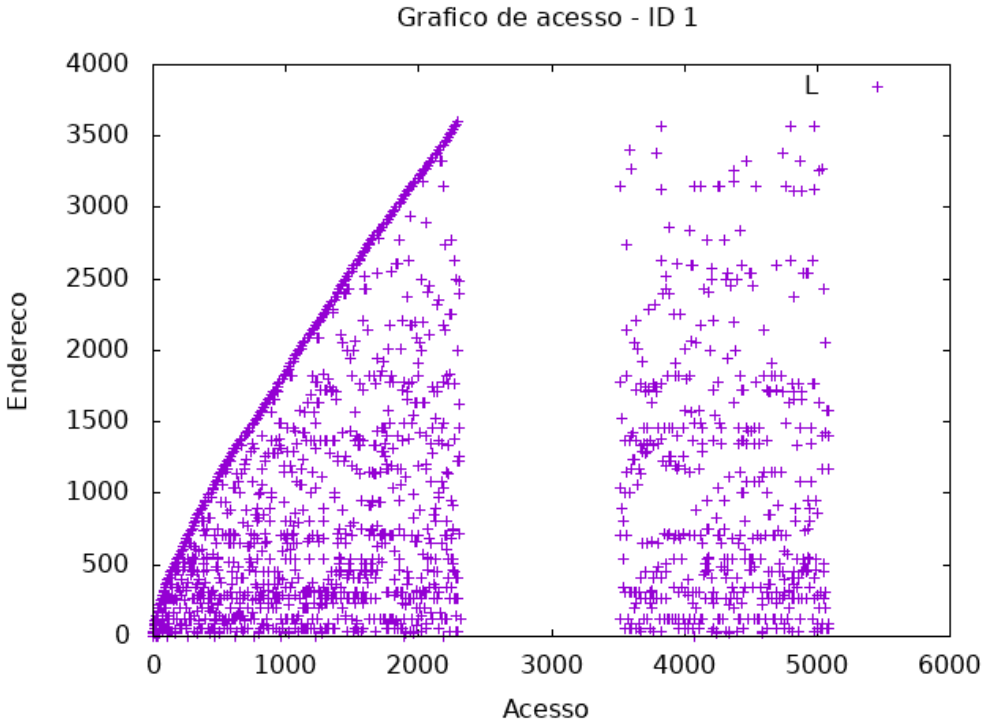


Grafico de acesso - ID 1
Endereco
Acesso
L
0
500
1000
1500
2000
2500
3000
3500
4000
0
1000
2000
3000
4000
5000
6000

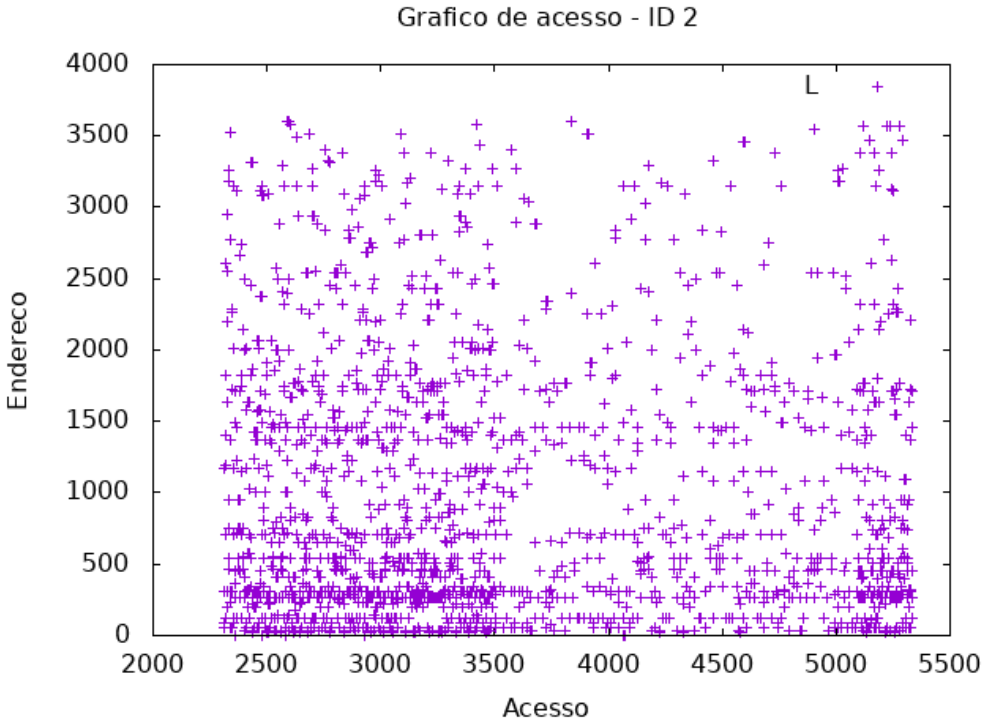
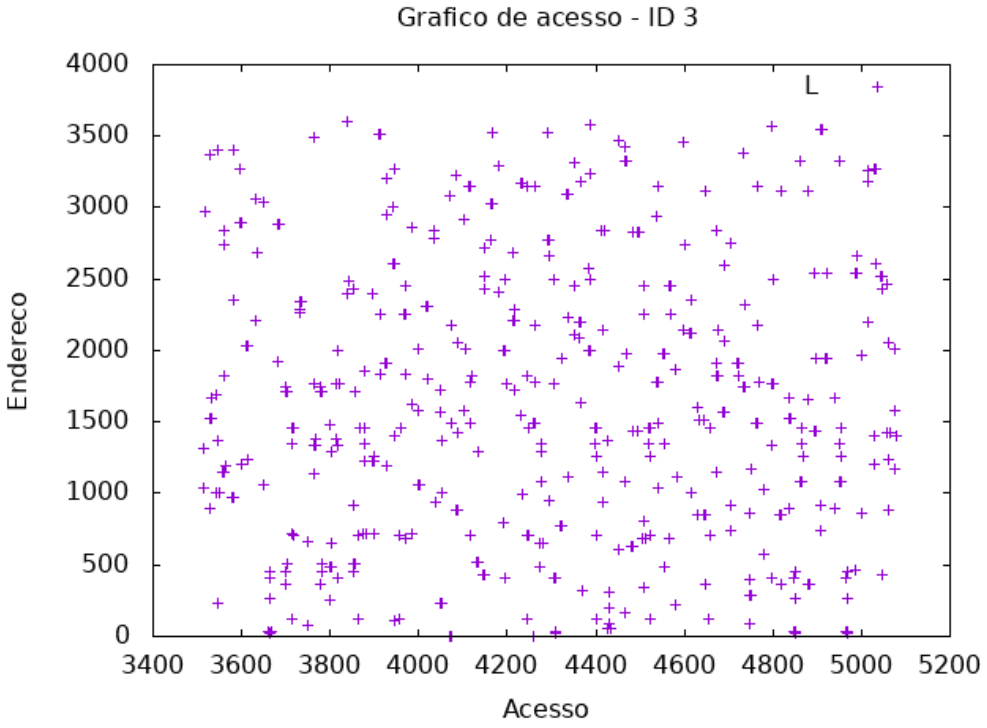
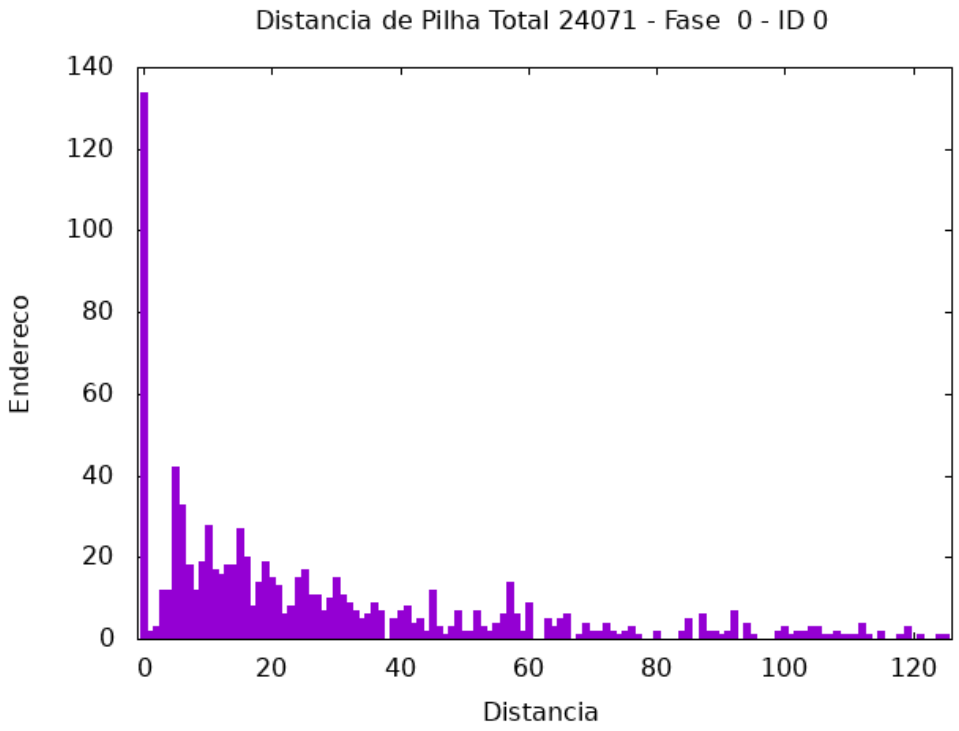


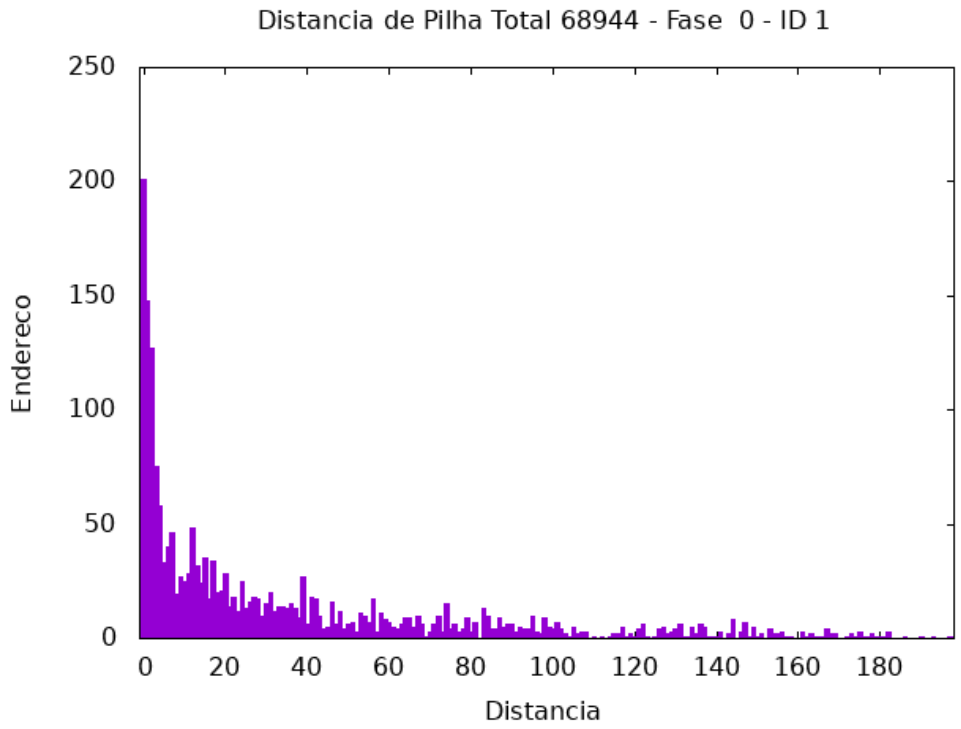
Grafico de acesso - ID 2
Endereco
Acesso
L
0
500
1000
1500
2000
2500
3000
3500
4000
2000
2500
3000
3500
4000
4500
5000
5500

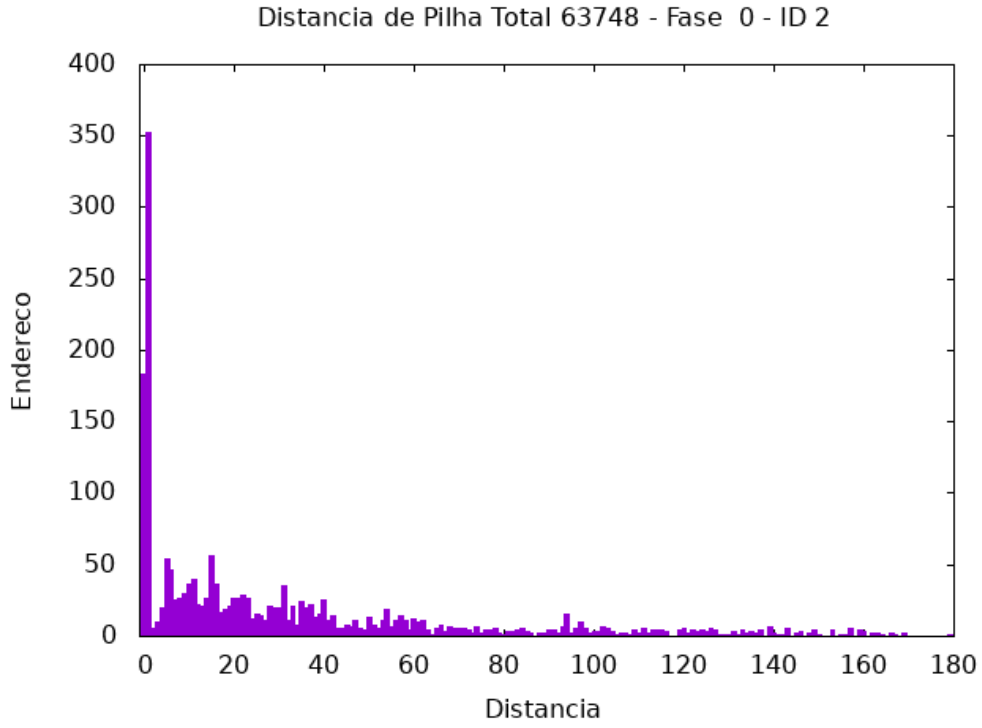


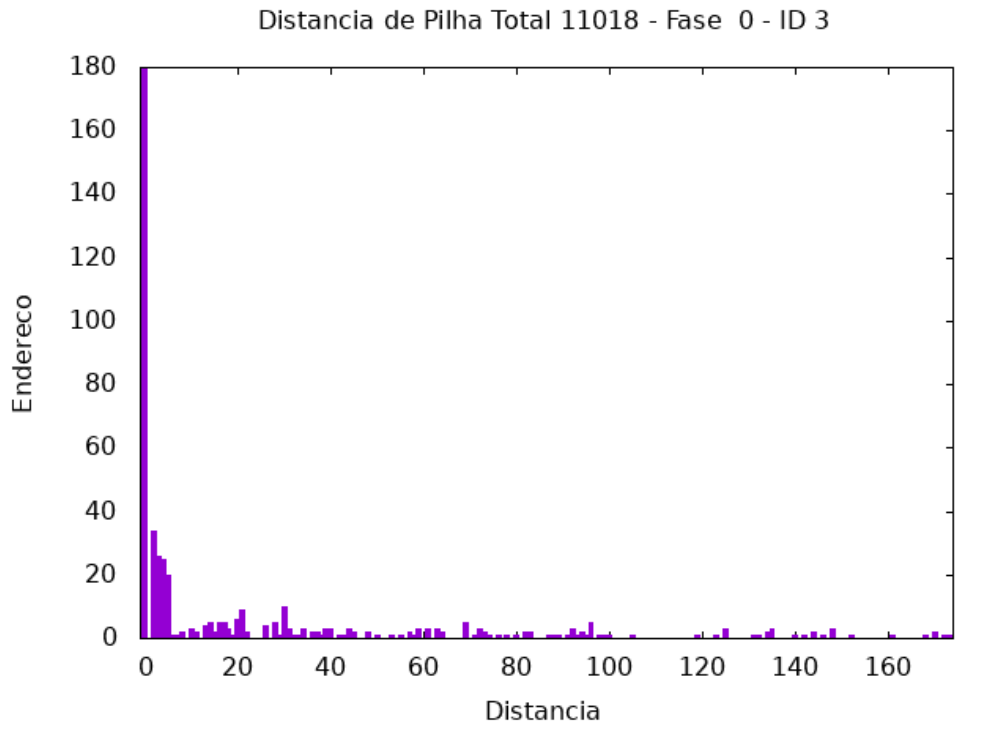
Como visto acima, o programa acessa diversas vezes posições de memória distantes, isso acontece porque os elementos são armazenados de forma dinâmica, desfavorecendo a localidade de referência. Nos ids 0 e 1, podemos enxergar uma forma de triângulo, que corresponde a busca binária durante as primeiras inserções feitas, visto que, durante a primeira população dos dados, existe muita memória livre, podendo-se armazenar dados próximos em posições contíguas na memória. No caso dos ids 2, 4 e 1 durante a segunda metade da execução, o gráfico mostra o efeito da busca binária na memória. Como muitos dados já foram rotacionados e removidos, elementos filhos de um nó da árvore tendem a ficar distantes na memória, desfavorecendo a localidade de referência quando se precisa realizar uma busca. Dessa forma, é esperado que a distância de pilha seja alta, visto que elementos distantes na memória são acessados e trocados de posição durante toda a execução do programa.

A seguir, estão os histogramas de distância de pilha do programa:


Distancia de Pilha Total 24071 - Fase 0 - ID 0
Endereco
Distancia
0
20
40
60
80
100
120
140


Distancia de Pilha Total 68944 - Fase 0 - ID 1
Endereco
Distancia
0
50
100
150
200
250
0
20
40
60
80
100
120
140
160
180





Dessa forma, conclui-se que o sistema não mantém uma baixa distância de pilha durante a execução, configurando um ponto negativo no que tange ao método usado para armazenar os dados do trabalho.

6. Conclusões

Com o intuito de simular um sistema de troca de emails entre usuários, foi implementado um programa que utiliza estruturas de dados que mantém a ordenação de forma não trivial para resolver o problema.

Durante o projeto do sistema foram levadas em consideração não só aspectos práticos do processamento de texto, mas também como a linguagem de programação escolhida poderia ser uma ferramenta útil para chegar no objetivo esperado. Toda a questão de mapear um mini-mundo de interesse em um modelo computacional robusto se mostrou bastante produtiva, levando o aluno a pensar em formas criativas de se resolver e entender o problema, tendo como resultado uma extensa prática técnica, principalmente ao se implementar uma árvore AVL completa e robusta. Por fim, o tempo extra usado para projetar o sistema trouxe várias recompensas no sentido da implementação, sendo um aspecto a ser levado para trabalhos futuros.

Além disso, os testes e a análise de complexidade se mostraram muito úteis para entender como as entradas influenciam a execução do programa e como alguns detalhes de implementação podem ser significativos em termos de desempenho computacional e localidade de referência.

Nesse sentido, todo o fluxo de trabalho foi essencial para a consolidação de conteúdos aprendidos em sala, além de apresentar, de forma prática, como softwares maiores, mais consistentes e robustos são projetados e implementados.

7. Bibliografia:

Ziviani, N. (2006). Projetos de Algoritmos com Implementações em Java e C++: Capítulo 3: Estruturas de Dados Básicas . Editora Cengage.

Paulo Feofiloff, Notas de Aula da disciplina Projeto de Algoritmos (2018). DCC-IME-USP. disponível em: https://www.ime.usp.br/~pf/algoritmos/aulas/

8. Instruções para compilação e execução:

8.1 Compilação

Existem partes do programa que são compatíveis apenas às versões mais recentes da linguagem c++, dito isso, deve-se seguir as seguintes configurações para a compilação:

Linguagem: C++
Compilador: Gnu g++
Flags de compilação: -std=c++11 -g
Versão da linguagem: standard C++11
Sistema operacional (preferência): distribuições baseadas no kernel Linux 5.15.

O comando para compilar o programa automaticamente está presente no arquivo **"Makefile"** e sua execução é chamada pelo comando **"make all"**. Ainda assim, seguem as instruções para compilar manualmente:

Para gerar o executável do programa, é necessário, primeiro, gerar o objeto para cada arquivo presente na pasta "/src". Tal objetivo pode ser alcançado seguindo os seguintes comandos em ordem:

```
g++ -std=c++11 -Wall -pg -c src/memlog.cpp -o obj/memlog.o -I./include/
g++ -std=c++11 -Wall -pg -c src/arvore.cpp -o obj/arvore.o -I./include/
g++ -std=c++11 -Wall -pg -c src/node.cpp -o obj/node.o -I./include/
g++ -std=c++11 -Wall -pg -c src/main.cpp -o obj/main.o -I./include/
g++ -std=c++11 -Wall -pg -c src/email.cpp -o obj/email.o -I./include/
g++ -std=c++11 -Wall -pg -c src/hash.cpp -o obj/hash.o -I./include/
```

Após esse passo, deve-se juntar todos os objetos em um único arquivo executável, seguindo o comando:

```
g++ -std=c++11 -Wall -pg -o ./bin/tp3.out ./obj/memlog.o ./obj/arvore.o ./obj/node.o ./obj/main.o ./obj/email.o ./obj/hash.o
```

Deste modo, o executável "/bin/tp3.out" estará compilado e pronto para ser utilizado.

8.2 Execução

Seguem as instruções para a execução manual:

1. Certifique-se que o compilável foi gerado de maneira correta, se algum problema ocorrer, execute o comando “make all” presente no “Makefile”.
2. Dado que o compilável foi gerado de maneira correta, certifique-se que o arquivo de entrada existe. Se ele não existir, crie-o.
3. Certifique-se que o arquivo de entrada está na formatação esperada, onde existe um inteiro na primeira linha do arquivo, seguido pelas operações especificadas.
4. Uma vez que os passos anteriores foram cumpridos, execute o programa com o comando: ./bin/tp3.out -i (entrada) -o (saida)
5. A saída estará guardada no arquivo de saida.