Trabalho Prático 3

Brisa do Nascimento Barbosa
Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil
brisabn@ufmg.br

1. Introdução

Neste presente trabalho, teve-se como objetivo a implementação de estruturas de dados eficientes para projetar um dicionário, em que suas operações básicas deveriam ser a consulta, inserção apropriada de significados, impressão ordenada dos verbetes, e remoção. A remoção era especialmente útil, visto que, após imprimir todo o dicionário, era necessário reimprimir considerando apenas os verbetes sem significado. Para isso, foram consideradas a tabela hash e a árvore de pesquisa balanceada AVL, e, posteriormente, foram avaliadas as vantagens e desvantagens de cada uma na projeção de um dicionário. Os procedimentos utilizados serão detalhados adiante, com a especificação do uso de bibliotecas, métodos e conceitos de desenvolvimento em C++.

2. Métodos

O programa foi desenvolvido na linguagem C++, compilado em G++ da GNU Compiler Collection. O computador utilizado possui o processador i7 e memória RAM de 8GM. O sistema operacional instalado é o Windows 10, portanto, foi utilizada uma máquina virtual WSL (Subsistema Windows para Linux) para a compilação em Linux.

2.1. Estruturas de dados

As estruturas de dados utilizadas foram a tabela hash e a árvore de pesquisa balanceada. Para lidar com as colisões em cada chave do hash, foi implementada uma lista encadeada de verbetes, e para estruturar os significados deles (tanto no hash quanto na árvore), foi utilizada uma lista simplesmente encadeada. Além disso, classes foram utilizadas para organização e abstração dos dados.

2.2. Classes

Em prol da modularização do programa, foram montados dois hpp principais: *hash* e *avl*, cada um tem como classe única a própria estrutura (Hash e AVL). O primeiro representa um dicionário projetado a partir de uma tabela hash, e o segundo, a partir de uma árvore de pesquisa balanceada. Ambas utilizam do hpp *significados* para armazenar os significados de cada verbete em uma lista

simplesmente encadeada. Além disso, especificamente para o hash, é utilizado também um hpp *ListaEncadeada* que armazena os verbetes de cada chave em uma lista duplamente encadeada.

Na *ListaEncadeada*, tem-se como classe principal “ListaEncadeada”, a qual possui como atributo a primeira e a última célula da lista e a classe “Celula_Verbete”, que detém um Verbete e seu próximo. A classe Verbete está presente no hpp *significados,* ela armazena uma lista de significados da palavra, a palavra, seu tipo, e uma chave hash. Além disso, nesse mesmo hpp, é definida uma classe ListaFrases, que é uma lista simplesmente encadeada de frases (significados), e as Frases, que armazena a string do significado e o próximo.

2.3. Funções

Apesar do Hash e da AVL possuírem funções afins, a estrutura de cada uma é bem diferente. Ao inserir um novo verbete na árvore balanceada, por exemplo, precisa-se atribuir uma nova raiz à árvore. Além disso, cada uma possui funções específicas que se adequam às necessidades de cada TAD.

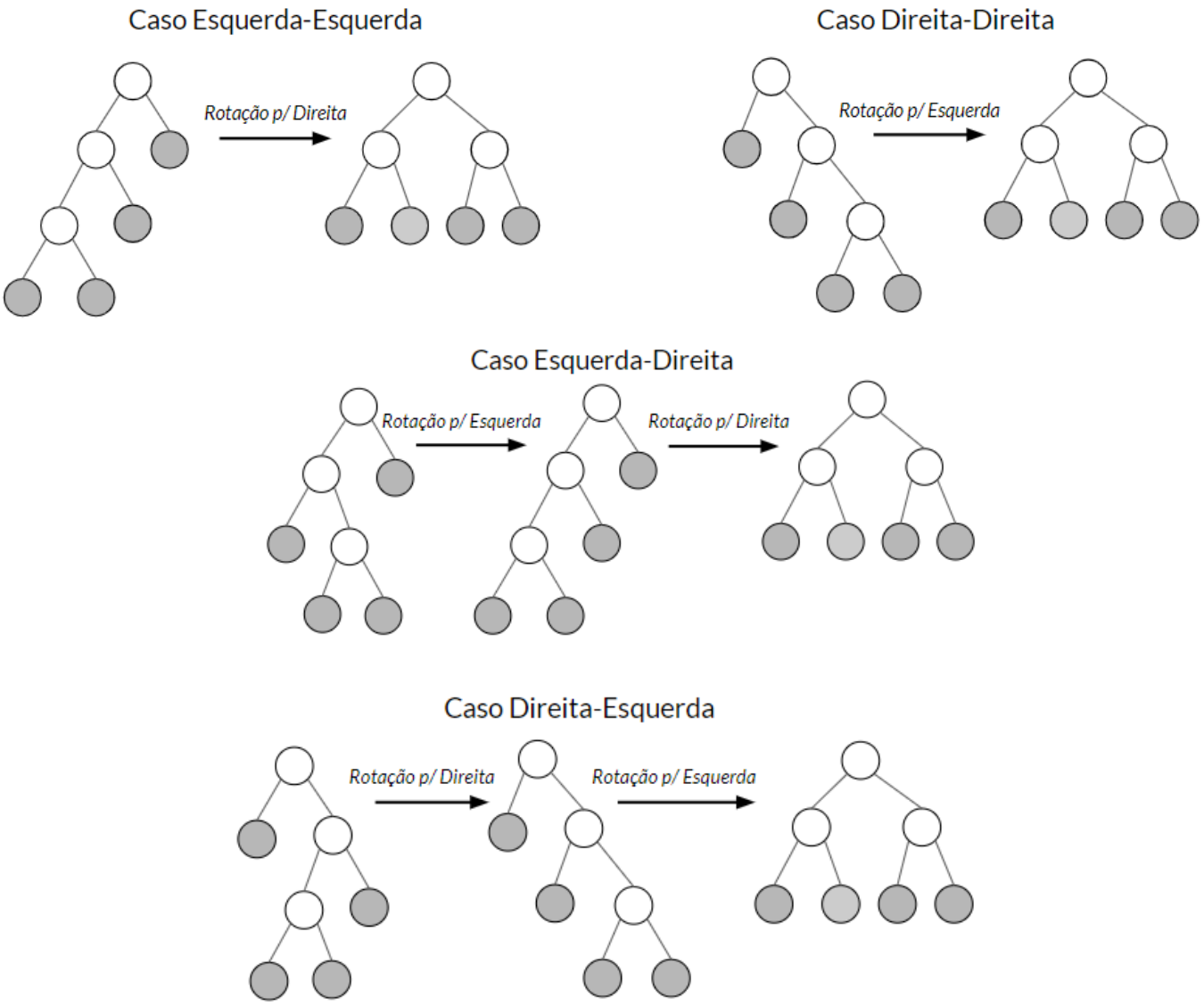
Funções da classe AVL:

Possui como funções principais inserir, remover, atualizar e imprimir verbetes. Atrelada à função de remover, há uma função “Remove_verbs” que itera a árvore procurando os verbetes com pelo menos um significado para removê-los. Já atrelada à função de pesquisa, há uma função que pesquisa pelo nódulo a ser removido.

Além disso, ao remover e inserir verbetes, é necessário rebalancear a árvore, para isso, a classe conta com as funções de rotacionar, medir o desbalanceamento e de encontrar o nódulo mínimo. A seguir, serão explicadas as funções principais com mais detalhes:

Insere: inicialmente, realiza-se uma inserção padrão de uma árvore binária, isto é, se a palavra for ‘menor’ lexicograficamente da palavra da raiz analisada, então recursivamente chama a raiz da esquerda, caso contrário, da direita. Uma observação é que no caso da árvore implementada, não há chances das palavras serem iguais, já que toda vez que a palavra ressurge no input, ela é a atualizada no mesmo nódulo da que já foi inserida.

Posteriormente a isso, a árvore está possivelmente desbalanceada, então primeiro atualiza a altura dela e em seguida mede-se o balanceamento entre o lado direito e esquerdo, que nada mais é do que a diferença de altura entre os dois. Caso o valor obtido seja maior que 1, será necessário rotacionar para direita, caso seja menor que 1, rotacionarmos para esquerda. Há quatro casos de rotação, segue-se eles (imagens de autoria própria):



Remove: realiza-se uma remoção padrão de uma árvore binária. Similarmente à inserção, chama-se a função recursivamente dependendo da ordem lexicográfica da palavra. Em seguida, ao encontrar a palavra na árvore,

Atualiza: adiciona um significado na lista simplesmente encadeada de significados do verbete. Logo, muda o nódulo antigo para ser esse verbete atualizado.

Pesquisa: procura recursivamente pelo nódulo que tem a mesma palavra do verbete pesquisado. Caso seja, então retorna o verbete do nódulo. A função Pesquisa_raiz é similar, mas ao invés de retornar um verbete, retorna o nódulo desse verbete.

Imprime (Display): imprime a árvore no pelo método em ordem, ou seja, do menor ao maior na ordem lexicográfica. Para isso, chama a função de impressão da classe verbete.

Destruir: percorre a árvore pelo método pós-ordem removendo os verbetes até então alocados na árvore.

Funções da classe Hash:

Possui como funções principais inserir, remover, atualizar e imprimir verbetes. Atrelada à função de remover, há uma função “Remove_verbs” que itera o hash procurando os verbetes com pelo menos um significado para removê-los. Com isso, é chamada a função "VerbetesVazios" da *ListaEncadeada*. Além dessas, a função “hash_chave” define a chave hash. A seguir, serão explicadas cada uma com mais detalhes:

hash_chave: define a chave baseada na primeira letra da palavra, isso é definido a partir do ASCII da letra. Quanto às letras maiúsculas, elas ficam antes na ordem lexicográfica, como sugerido pelos testes enviados no Moodle. Assim, há 52 chaves disponíveis, do ‘A’ ao ‘Z’ (alfabeto maiúsculo) e do ‘a’ ao ‘z’, alfabeto minúsculo.

Insere: calcula-se a posição da tabela baseada na chave hash. Após isso, insere-se o verbete na *ListaEncadeada* daquela determinada chave, logo, chama-se a função InsereItem da Lista. Essa inserção é realizada de acordo com a ordem lexicográfica.

Remove: calcula-se a posição da tabela baseada na chave hash. Após isso, retira-se o verbete da *ListaEncadeada* daquela determinada chave, logo, chama-se a função RemoveItem da Lista.

Atualiza: calcula-se a posição da tabela baseada na chave hash. Após isso, atualiza-se o verbete da *ListaEncadeada* daquela determinada chave, logo, chama-se a função Replace da Lista. Nessa função, adiciona o novo significado ao verbete e chama a função SetItem para redefinir o item na posição do verbete na lista.

Pesquisa: tal como as anteriores, a chave hash também é calculada para definir qual posição da tabela. Logo, chama-se a função de pesquisa da *ListaEncadeada*, que procura a palavra e retorna o verbete que for igual a procurada.

Imprime (Display): realiza-se uma iteração das listas encadeadas da tabela e imprime cada uma delas a partir da função de imprimir da *ListaEncadeada.*

Destruir: percorre cada chave do hash, a cada lista encadeada que encontra, chama a função de limpar da lista, dessa forma, deletando cada verbete até então alocados no hash.

3. Análise de complexidade

3.1. Hash

3.1.1 Tempo

Numa tabela hash padrão, tanto para inserir, para retirar, e para pesquisa, a complexidade é constante no melhor caso, O(1), algo impressionante. No entanto, como são utilizadas listas duplamente encadeadas para lidar com as colisões em cada letra, a complexidade vai ser somada com as respectivas complexidades de inserção, retirada e pesquisa de uma lista encadeada. No caso médio, será O(1) + O(n/m) para inserção e retirada. Em relação à função de Atualiza, realiza uma pesquisa para descobrir a posição do verbete, logo, O(n/m) no pior caso e O(1) no melhor. Portanto, o custo total médio será:

$$O(n/m) + O(1) = O(n/m)$$, que é linear

3.1.2 Espaço

Ao lidar com as colisões, foram utilizadas listas encadeadas, o que requer um armazenamento extra de espaço. Logo, sendo n o número de itens inseridos e m o tamanho da tabela hash, a complexidade de espaço será $O(n+m)$, que é linear.

3.2. AVL

3.2.1 Tempo

Considerando ‘n’ o número de nódulos da árvore, para inserir um verbete na avl, no pior caso, a complexidade é O(log n), que é a altura da árvore. Já para balancear a árvore, as rotações realizam operações em O(1), logo, a complexidade permanece O(log n). Essa mesma complexidade aparece na retirada de elementos. Para pesquisar um verbete, no melhor caso ele já está na raiz (complexidade O(1), constante), e no pior, está no final da árvore (complexidade O(log n), altura da árvore).

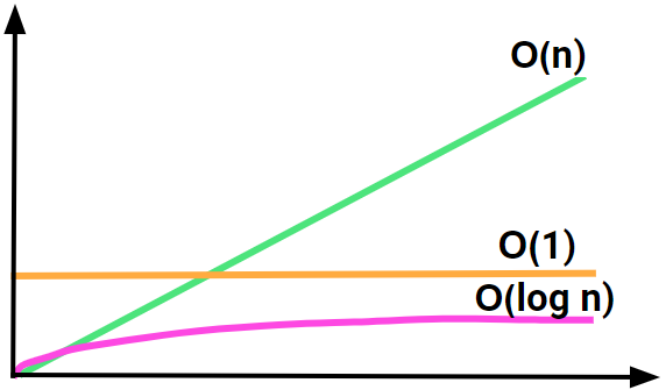
3.2.2 Espaço

Tal qual a árvore binária de pesquisa, a complexidade de espaço será linear, O(n), tanto no pior caso quanto no melhor caso.

3.3. Comparação

A partir das complexidades obtidas, pôde-se perceber que o algoritmo hash possui melhor complexidade do que a árvore avl no melhor caso, já que a lista encadeada fornece O(1) no melhor caso + O(1) do hash, apesar da complexidade logarítmica da avl também ser ótima. Já quando se analisa o pior caso, pôde-se notar que a avl pode ser melhor do que o hash, já que sempre será O(log n), enquanto o hash, por conta da implementação de listas encadeadas, será O(n).

Segue-se abaixo uma comparação entre as complexidades abordadas



Assim, é notável que, como a melhor complexidade é a constante, o hash será superior em alguns casos, mas a avl, em média, será O(log n). Com isso, percebe-se o impacto do uso de listas encadeadas no hash.

4. Estratégias de Robustez

A fim de firmar a robustez do código, foi utilizada a biblioteca <msgassert.h>, a qual define macros para verificar a precisão das funções.

No **Hash**, são definidos os seguintes asserts:

- Aviso se o primeiro caracter da palavra não for válido (não estiver no alfabeto);
- Aviso se a posição na tabela não for válida, o que poderia causar segmentation fault

Na **ListaEncadeada**, são definidos os seguintes asserts

- Erro de posição inválida se o tamanho da lista for menor que a posição ou a posição for menor que zero, ao posicionar um item;
- Erro se a lista estiver vazia ao remover um item;
- Erro se a lista estiver vazia ao pesquisar;

Na **AVL**, são definidos os seguintes asserts

- Aviso se o primeiro caracter da palavra não for válido (não estiver no alfabeto);

Na **main**, é definido os seguintes asserts

- Erro a implementação do dicionário (hash ou avl) não seja definida;
- Erro caso o tipo seja diferente de n (nome), a (adjetivo) ou v (verbo);
- Erro caso haja erro na abertura do arquivo de entrada;
- Erro caso haja erro na abertura do arquivo de saída;
- Erro caso o arquivo de entrada não seja definido;
- Erro caso o arquivo de saída não seja definido;
- Aviso casa a entrada não esteja no padrão adequado (baseado nos testes fornecidos no moodle);

5. Análise Experimental

A fim de comparar o desempenho de cada algoritmo, foram consideradas duas métricas: tempo e memória. Para isso, inicialmente, foram realizados testes considerando o tempo depreendido para a execução de cada algoritmo.

Por conta da própria máquina, há uma diferença no tempo de execução a cada vez que se roda o programa, assim, essa métrica foi considerada apenas para ter uma noção quanto à rapidez dos algoritmos, mas não para tomar esses valores como apuração definitiva. Ainda, para abranger um maior espectro dessas variações, cada teste foi rodado 20 vezes, e o tempo na tabela é a média deles em milissegundos.

Segue-se a tabela com diferentes valores de número de verbetes e de número máximo de significados por verbete:

N° de Verbetes	N° Máximo de Significados	HASH Encadeado	Árvore AVL
200	5	5.63	7.43
1000	2	20.67	27.34
1000	5	22.72	30.91
2000	5	43.75	55.59
2000	7	50.32	65.69
3000	5	65.86	85.25
5000	7	154.92	197.87

Analisando a tabela, fica notável a vantagem do hash em relação à árvore balanceada no quesito tempo. De fato, realizando uma porcentagem entre os valores, a melhora é de, respectivamente, 31%, 32%, 36%, 27%, 30%, 29% e 27%. Ainda que, evidentemente, apresenta uma variância entre os porcentuais, a melhora fica na faixa de 30%.

Ademais, a partir desses resultados, é possível perceber o impacto do desempenho quando se aumenta o número máximo de verbetes para uma mesma quantidade de verbetes. Isso é, quanto mais significados tem um verbete, o programa tende a ser mais lento. Por exemplo, quando a entrada fornece 1000 verbetes, ao aumentar o número de significados de 2 para 5, tanto o hash quanto a árvore ficaram cerca de 10% mais lento. Isso, provavelmente, se deve pelo fato de adicionar um significado ser mais custoso do que inserir um novo verbete, já que depende de substituir itens de uma árvore ou uma tabela encadeada, ao invés de só inserir.

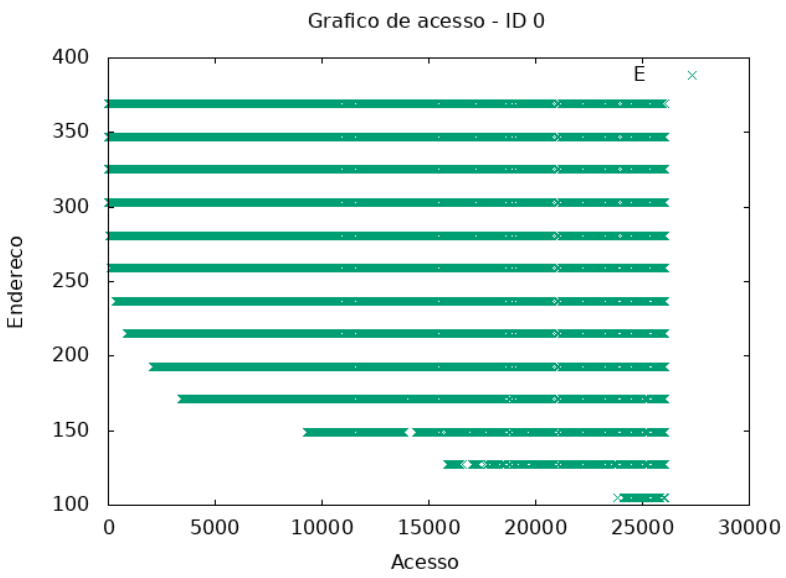
Já em termos de funcionalidade, para todos os casos o programa executou perfeitamente, fornecendo a saída esperada. Esses testes foram realizados a partir dos testes disponibilizados pelo moodle e por entradas modificadas a partir desses mesmos arquivos.

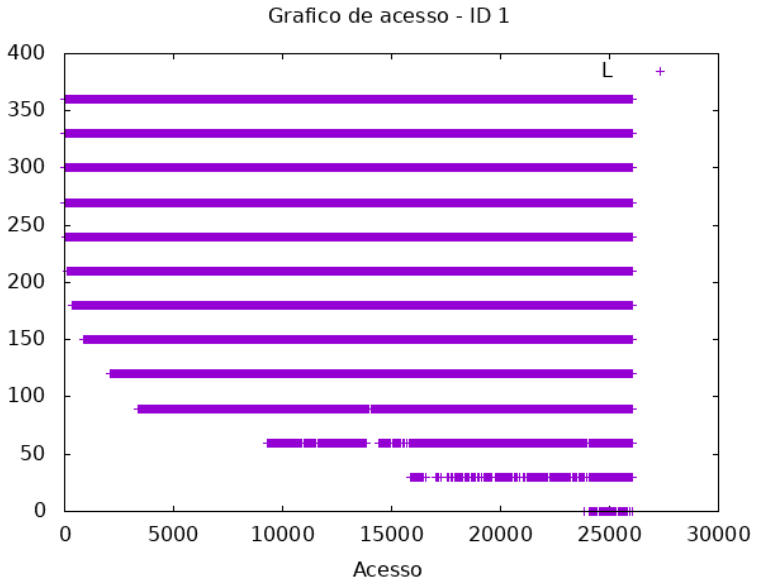
5.1. Gráficos

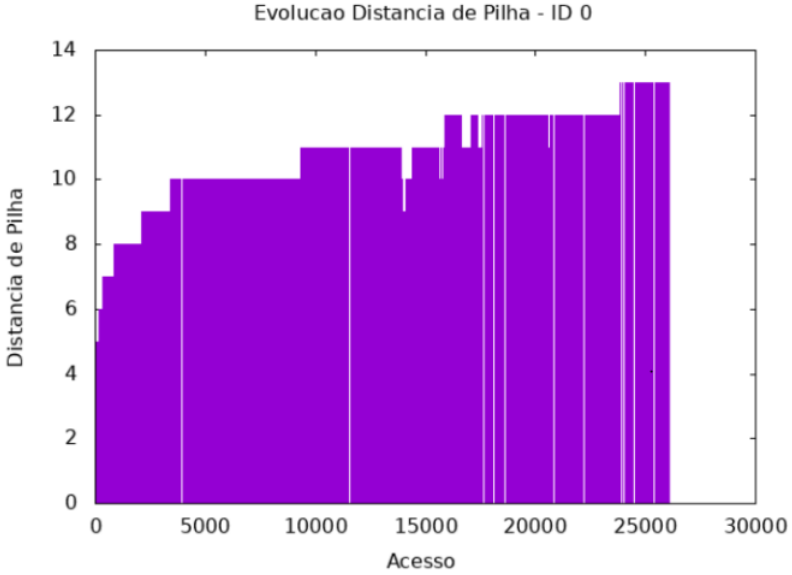
Os seguintes gráficos foram gerados pelos logs com arquivos de entrada de 1000 palavras com no máximo 2 significados. Para melhor visualização, a leitura está no ID 1 e a escrita no ID 0.

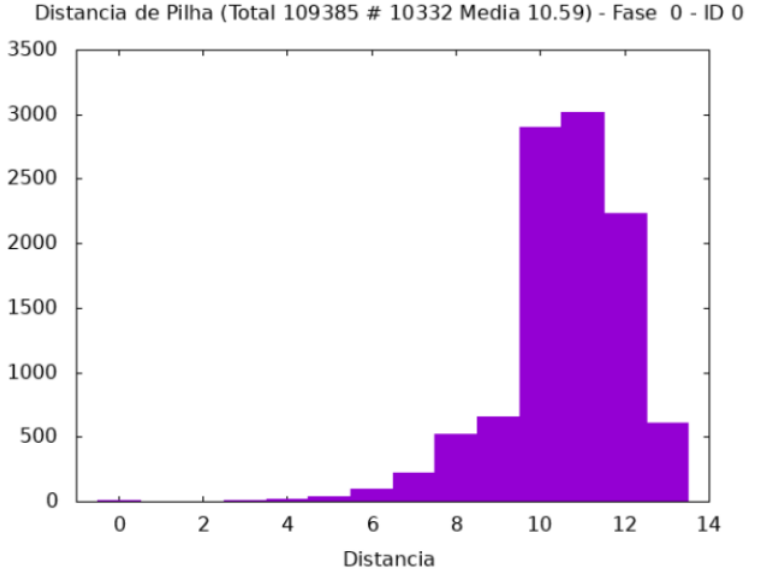
5.1.2. AVL

Abaixo estão gráficos de acesso, evolução de distância de pilha e distância de pilha obtidos ao rodar o dicionário com a estrutura AVL.



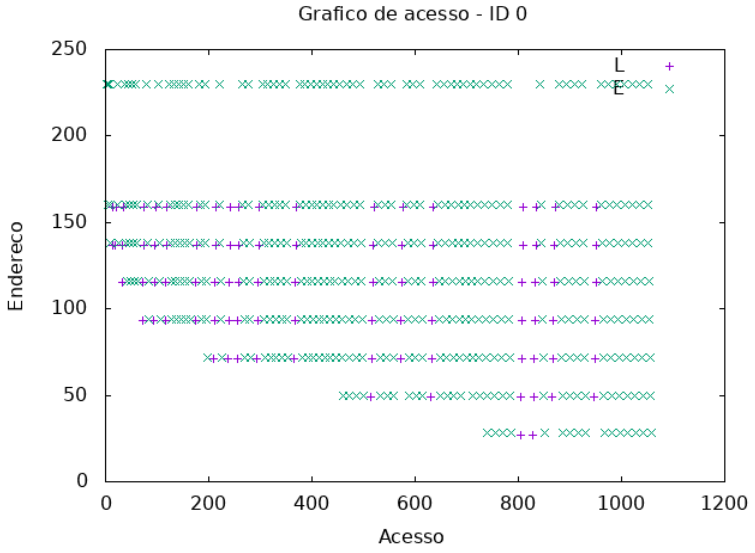






Os gráficos da árvore AVL gerados pelo gnuplot expõem com clareza o comportamento da memória ao se acessar e ler itens da árvore. O aspecto de ser salteado se deve pelo acesso à árvore não ser sequencial, e sim, a cada nódulo ir “pulando” e criando auxiliares de acesso. Agora, em se tratando dos gráficos de distância de pilha, esses também estão condizentes com o esperado, já que tem apresenta “alturas” diferentes de acordo com o acesso e a distância. Isso pode ser explicado pois a pilha varia de acordo com os nódulos acessados.

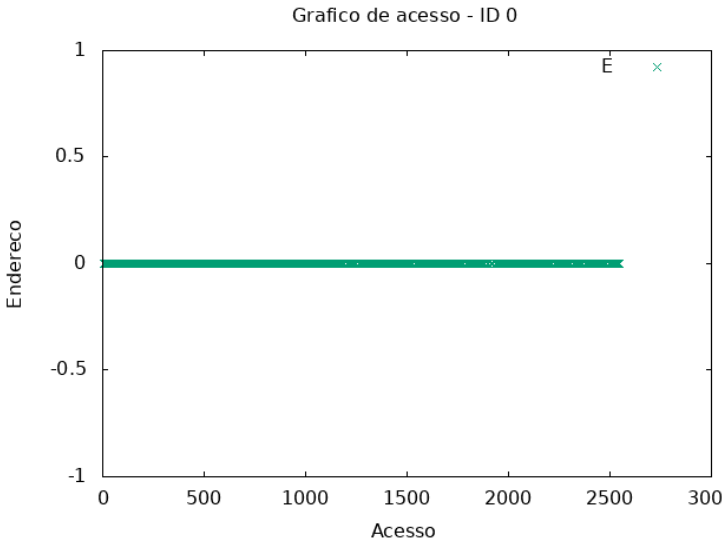
É válido analisar também a relação da leitura e escrita de acesso, pois quando se realiza leitura e já existe o verbete, ao invés de inserir na árvore, atualiza o já existente com um novo significado. Observe os gráficos juntos para uma entrada de 65 palavras, com no máximo 2 significados por verbete

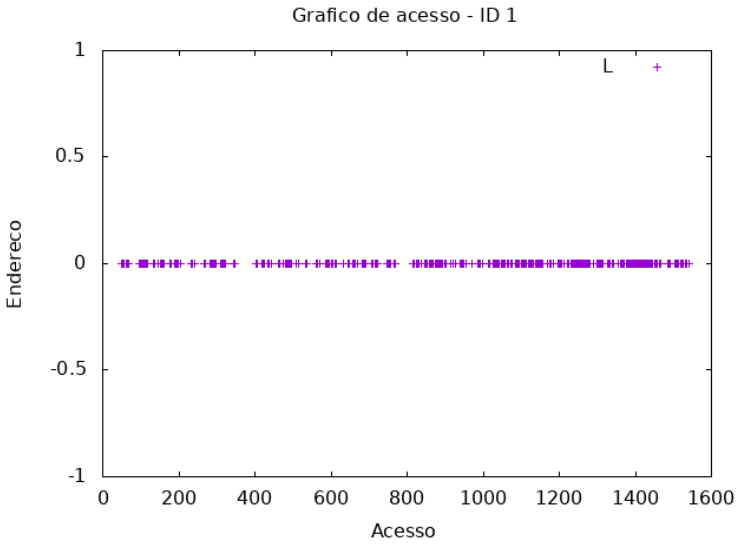


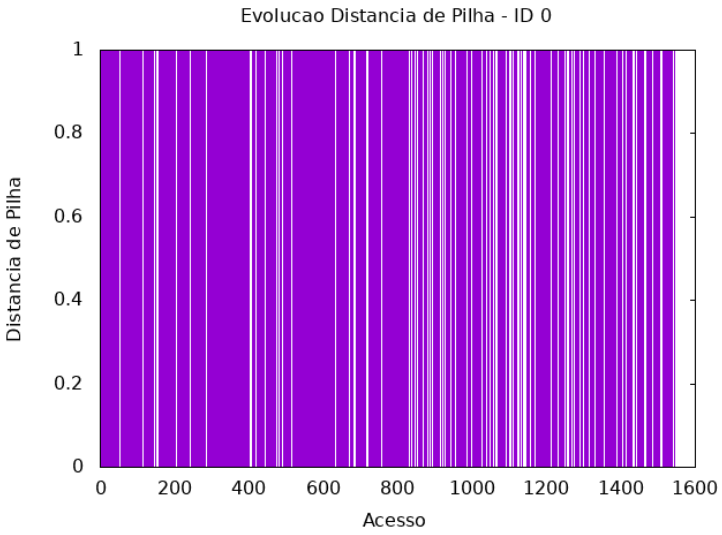
Esse gráfico é bem interessante, pois mostra que, quando realiza leitura e já existe um verbete, ao invés de inserir na árvore, atualiza o nódulo já existente.

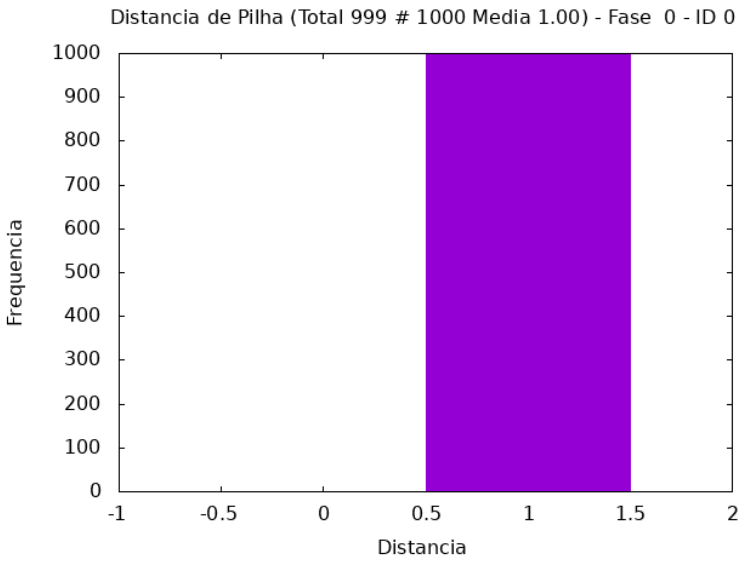
5.2. HASH

Abaixo estão os gráficos de acesso, evolução de distância de pilha e distância de pilha obtidos ao rodar o dicionário com a estrutura hash.



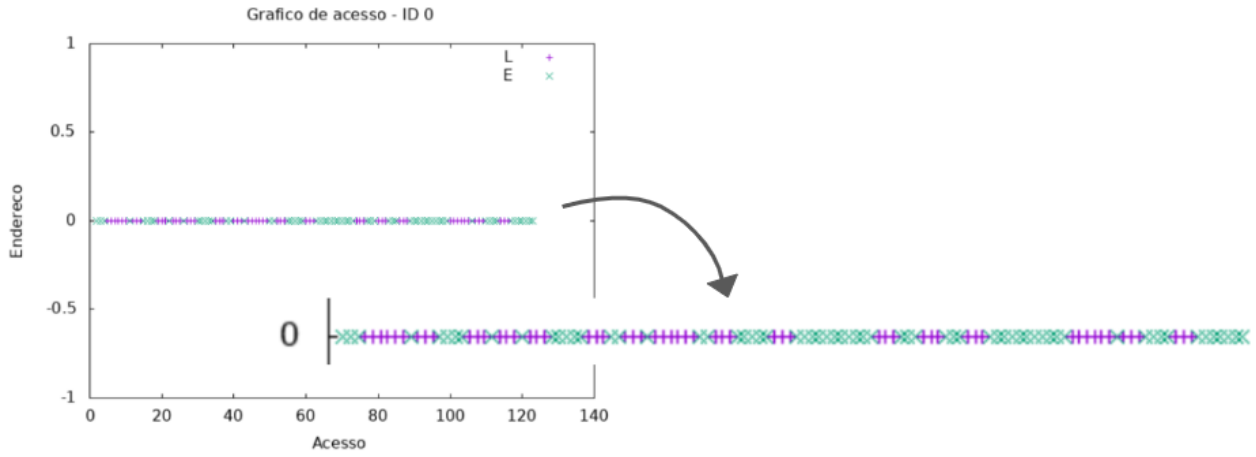






Os gráficos gerados pelo gnuplot permitem verificar que o acesso à memória é bem definido e constante, ou seja, o desempenho computacional do hash é bastante eficiente. Dessa forma, aspecto de rapidez e constância das operações do hash podem ser confirmadas, transparecendo a superioridade do hash como um excelente método de pesquisa.

Além disso, percebe-se que existem “buracos” no gráfico de escrita. Isso pode ser melhor visualizado com menores quantidades de palavras, observe para 65 palavras:



Esse comportamento demonstra claramente que, ao fazer a leitura para conferir se ele já existe, se existir, então o verbete não é inserido, e sim é feita uma atualização.

6. Conclusão

A partir do desenvolvimento do programa, pode-se compreender na prática o funcionamento de árvores AVL e tabelas hash, além de poder explorar a adequação dessas estruturas às necessidades de diferentes propósitos, nesse caso, um dicionário. Para isso, foi necessário ponderar a respeito das vantagens e desvantagens de cada estrutura, pois, apesar de funções relativamente parecidas, essas possuem diferentes complexidades. E além do potencial computacional, a tabela hash pareceu mais "simples" de ser implementada, já que utiliza conceitos mais intuitivos do que uma árvore, a qual demanda mais reflexão em termos de recursão. Nesse sentido, implementar essas estruturas foi de demasiado proveito, já que agora, ficou muito mais claro com funcionamento prático do que o mero conhecimento teórico delas.

No entanto, a maior dificuldade no desenvolvimento deste trabalho consistiu na estruturação do próprio programa, já que são duas implementações complexas para uma mesma finalidade. Então, inicialmente, não soube como realizar essa combinação. Posteriormente, com os fundamentos agregados de modularização em C++, isso foi entendido melhor. Além disso, houve um desafio considerável quanto à análise experimental. Apesar da geração de gráficos com o <*memlog*> ter sido mais tranquila, depois de ter aprendido e melhorado a cada tp realizado, considerar as dimensões (número de verbetes e significados) não foi intuitivo de analisar, além que gerar arquivos de entrada diferentes para serem testadas não foi uma tarefa fácil. Contudo, apesar dos desafios, houveram mais aprendizados do que infortúnios.

Portanto, com a finalização do programa e subsequentes análises computacionais, pode-se verificar a relevância da compreensão de estruturas de dados, nesse caso de hash e árvores balanceadas de pesquisa, para um entendimento correto da manipulação e gerenciamento de dados. Principalmente no tocante à análise de complexidade, pois tanto o hash quanto o avl se mostraram realmente bem efetivos. Além disso, percebeu-se que práticas anteriores realizadas ao longo do semestre foram bastante válidas para proceder outros elementos do presente trabalho, como o uso da biblioteca <*memlog*>, a implementação da lista encadeada, análise de custos de complexidade, dentre outros.

Bibliografia

- Cormen , T., Leiserson, C, Rivest R., Stein, C. *Introduction to Algorithms, Third Edition, MIT Press, 2009.*
- Chaimowicz, L. and Prates, R. (2020). *Slides virtuais da disciplina de estruturas de dados*. Disponibilizado via moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

Instruções de compilação e execução

O makefile possui três comandos principais: make (compila arquivos recentemente modificados), make all (compila todos os arquivos) e make all (limpa os arquivos compilados). Para rodar corretamente, deve-se utilizar um terminal e digitar make para compilar.

Em seguida, há duas opções para organizar o dicionário: *hash e árvore avl*. Se for um hash, coloque *-t hash,* se for avl, coloque *-t arv*. Para informar o arquivo de entrada, o argumento é *-i [nome do arquivo]*; para produzir um arquivo de saída, o argumento é -o *[nome do arquivo].* Exemplos de linha de comando após compilado o programa são:

Hashtable: ./bin/run.out **-i** input.txt **-o** output.txt **-t** hash

Árvore AVL: ./bin/run.out **-i** input.txt **-o** output.txt **-t** arv

Onde ./bin/run.out é o executável e input é o arquivo de texto que contém os comandos da simulação.