Trabalho Prático 3 - Dicionário

Lucas Almeida Santos de Souza - 2021092563[1]

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

luscaxalmeidass@ufmg.br

1 Introdução

A proposta desse trabalho é comparar a eficiência das estruturas de dados Hash e Árvore AVL, através da implementação de um dicionário que armazena palavras e suas respectivas definições em ordem alfabética. Esse dicionário é feito pensando na utilização comum por alguma empresa, por exemplo, e contém funções para inserir, buscar, remover, atualizar e imprimir verbetes em ambas as implementações. O programa principal construído para testar as funções segue o seguinte roteiro:

1. O usuário envia na linha de comando os arquivos de entrada e saída, junto da opção de qual estrutura de dados deseja utilizar para armazenar os verbetes;
2. O programa lê o arquivo de entrada contendo as palavras e suas definições e insere no dicionário utilizando a estrutura de dados desejada;
3. O programa imprime o dicionário montado em ordem alfabética e em ordem de inserção de seus significados;
4. O programa então remove todos os verbetes do dicionário que possuem pelo menos um significado registrado, pois a empresa que está construindo o dicionário deseja saber quais verbetes ainda devem ser complementados com seus significados;
5. O programa imprime o dicionário atualizado, apenas com os verbetes que não têm nenhum significado registrado.

Para a análise, será considerado o tempo de execução de diferentes testes, com alterações no número de verbetes inseridos e no número de significados de cada verbete. Isso será visto mais adiante no relatório, na seção de Análise Experimental.

2 Método

2.1 Especificações

O código foi desenvolvido na linguagem C++, compilada pelo compilador GCC da GNU Compiler Collection. O Computador utilizado tem as seguintes especificações:

- Sistema operacional Linux Ubuntu 22.04
- Processador intel core i5 8ª geração
- 8GB de memória RAM

2.2 Execução

A implementação das estruturas de Hash e Árvore AVL foi feita com base nas aulas de Estruturas de Dados e nos slides da matéria. Também foram utilizados conceitos de Lista Encadeada e Fila de Prioridade, também vistos nas aulas e nos slides.

Além disso, foi criado uma biblioteca somente para declaração das variáveis globais utilizadas para receber os argumentos da linha de execução do código.

Variáveis Globais

Namespace args: Namespace criado para agrupar as variáveis globais utilizadas para receber os argumentos na linha de execução do programa, que são usadas durante o programa em suas respectivas funções.

Classes

- **Significado e ListaSignificado:** A classe Significado armazena o significado de um verbete, com um atributo string para armazenar o texto e um ponteiro para o próximo significado da lista. A classe ListaSignificado é uma lista encadeada de significados, com ponteiro para o primeiro e último elementos e um inteiro representando seu tamanho.
- **Verbete:** Essa classe armazena os verbetes do dicionário, com os atributos:

  `tipo` - caractere que armazena o tipo do verbete, podendo ser:

  * `n` para substantivo (noun);
  * `v` para verbo (verb);
  * `a` para adjetivo (adjective).

  `palavra` - string que armazena a palavra do verbete;

  `significados` - ponteiro para a lista de significados do verbete;

  `prox` - ponteiro para o próximo verbete da lista.
- **ListaVerbete:** Lista encadeada de verbetes, com ponteiro para o primeiro e último elementos e um inteiro representando seu tamanho.

- **Dicionario:** Classe abstrata que define as funções em comum que serão implementadas nas classes de Hash e Árvore AVL.
- **Hash:** Classe que implementa a estrutura de Hash, com as funções de inserção, pesquisa e remoção. Cada elemento da tabela contém uma lista de verbetes ordenados alfabeticamente para o caso de haver colisões. Como a estrutura deve ser impressa em ordem alfabética, foi optado por já armazenar de um modo que a impressão não precisasse se ser odenada. Por isso, o método de hashing utiliza as três primeiras letras da palavra inserida (caso não tenha a segunda e/ou terceira letra, essas são consideradas como **a**) para calcular a posição na tabela. Assim, a tabela do hash tem 26^3 posições, e a função de hashing é: $h(palavra) = (palavra[0] - a) \cdot 26^2 + (palavra[1] - a) \cdot 26 + (palavra[2] - a)$. A subtração por 'a' é feita para que a tabela comece a ser preenchida a partir da posição 0, e não da posição 97, que é o código ASCII da letra 'a'.
- **No e ArvAVL:** Classe que implementa a estrutura de Árvore AVL, com as funções de inserção, pesquisa e remoção, além das funções próprias dessa estrutura de dados (balanceamento, rotação, cálculo de altura, etc). Os nós da árvore, representados pela classe No, contém um verbete, e um ponteiro para os filhos da esquerda e da direita, além de um inteiro indicando sua altura na árvore.

Funções utilizadas para a execução do programa

- **main( ):** Função principal do programa, que chama as funções de leitura do arquivo de entrada, execução do algoritmo de ordenação escolhido e escrita do arquivo de saída.
- **parseArgs( ):** Recebe os argumentos da linha de execução e os armazena nas respectivas variáveis globais.

3 Análise de Complexidade

Essa seção se trata de analisar a complexidade de tempo e espaço de cada uma das funções utilizadas no programa. Para analisar a complexidade de espaço, será feito também um cálculo aproximado da memória interna utilizada por cada estrutura. Através da função `sizeof()`, temos que cada elemento da classe Verbete ocupa 56 bytes (incluindo o ponteiro para a lista de significados), e cada elemento da classe Significado ocupa 40 bytes.

3.1 Hash

Nessa estrutura, o melhor caso acontece quando não há colisões, enquanto o pior caso acontece quando todas as entradas são inseridas na mesma posição da tabela (por exemplo, um dicionário somente com palavras que começam com "int"). Nesses casos, temos as seguintes complexidades:

Hashing

Para todos os casos, a função de hashing é a mesma e sua complexidade é $O(1)$, pois ela é feita apenas com operações aritméticas.

Inserção

- **Melhor caso:** $O(1)$, pois a única operação feita é o cálculo do hashing.
- **Pior caso:** $O(n)$, pois além do cáclulo do hashing, é feita a inserção ordenada na lista, onde o pior caso também depende da palavra ser alfabeticamente a última começando com as mesmas três letras.

Pesquisa

- **Melhor caso:** $O(1)$, pois a única operação feita é o cálculo do hashing.
- **Pior caso:** $O(n)$, pois além do cáclulo do hashing, é feita a pesquisa na lista, onde o pior caso também depende da palavra buscada ser alfabeticamente a última começando com as mesmas três letras.

Impressão

A complexidade da função de impressão é $O(n)$ para todos os casos, pois ela percorre todas as posições da tabela de hashing, e para cada uma delas, percorre a lista de verbetes, imprimindo cada um deles.

Remoção de um elemento

- **Melhor caso:** $O(1)$, pois a única operação feita é o cálculo do hashing e a exclusão do elemento.
- **Pior caso:** $O(n)$, pois além do cáclulo do hashing, é feita uma busca na lista pelo elemento que será removido.

Remoção dos verbetes com significados

Nessa função, todos os casos têm complexidade $O(n)$, pois independentemente da quantidade de verbetes com significados que existir no dicionário, a função percorrerá todo o hash, pois todos os elementos devem ser checados.

Complexidade de espaço

A estrutura de hash contém uma tabela de tamanho 26^3 contendo uma lista de verbetes em cada posição. Esse tamanho é constante, independente da quantidade de entradas. Cada entrada do arquivo porém, corresponde ou a um novo verbete ou a um significado que será inserido em um verbete existente. Assim, se a entrada contém n verbetes distintos sem significados e m verbetes repetidos (com a função de adicionar um novo significado), a complexidade de espaço do hash será de $O(n + m)$.

Em questão de memória, um hash vazio ocupa aproximadamente 421.824 bytes, ou 412kB de memória, pois é um vetor de 26^3 posições de 24 bytes (tamanho da estrutura ListaVerbete) cada. Após ser preenchido com n verbetes distintos sem significados, a memória ocupada se torna de $421.824 + 56n$ bytes. Se, no total, m significados forem adicionados a esses verbetes, temos

uma memória ocupada de $421.824 + 56n + 40m$ bytes. Isso confirma a afirmação anterior, pois o crescimento da memória ocupada é linearmente proporcional ao número de verbetes e significados.

3.2 Árvore AVL

Nessa estrutura, as funções têm complexidades similares independente dos casos, pois a árvore realiza rotações para manter seu balanceamento.

Rotação

As funções do rotação relaizam apenas redefinições de valores de ponteiros, tendo assim complexidades de tempo $O(1)$.

Inserção

Essa função realiza uma busca binária para encontrar a posição em que o nó de ver inserido, com custo $O(\log n)$. Após inserir, as alturas de cada nó presente na busca são atualizados em tempo $O(\log n)$ e, ao final da operação, é realizado o balanceamento da árvore em complexidade $O(\log n)$ e operações de rotação para cada desbalanceamento. Assim, a complexidade total da inserção é dada por $O(\log n) + O(\log n) + (O(\log n) \cdot O(1)) = O(\log n)$.

Impressão

A complexidade da função de impressão é $O(n)$ para todos os casos, pois ela percorre todas as posições da árvore, imprimindo cada nó.

Remoção de um elemento

- **Melhor caso:** Ocorre quando o elemento que deseja ser retirado está na raiz da árvore. Nesse caso, o programa realiza apenas as seguintes operações: busca a maior folha menor que o elemento ($O(\log n)$); troca o elemento por essa folha ($O(1)$); chama a função recursivamente para retirar a folha ($O(\log n)$); atualiza as alturas dos nós ($O(\log n)$); realiza o balanceamento da árvore ($O(\log n)$), com operações de rotação $O(1)$ para cada desbalanceamento. Assim, a complexidade total é $O(\log n) + O(1) + O(\log n) + O(\log n) + (O(\log n) \cdot O(1)) = O(\log n)$.
- **Pior caso:** Ocorre quando o elemento que deseja buscar é o maior elemento, presente na folha mais à direita da árvore. Nesse caso, além de todas as operações acima, a busca é feita em tempo $O(n)$, tornando toda a complexidade $O(n)$.

Remoção dos verbetes com significados

Essa função percorre toda a árvore em formato pós-ordem e, ao encontrar um verbete que contém significados, chama a função `RemoveRecursivo()` com o endereço do nó e sua chave. Assim, a função de remoção não realiza a busca, tendo apenas a complexidade $O(\log n)$. Ao final do processo, a árvore é balanceada, com complexidade $O(\log n)$. Ao todo, a complexidade da função é $O(n) \cdot O(\log n) + O(\log n) = O(n \log n)$, pois percorre toda a árvore e realiza a remoção

de cada nó que contém significados. Se a árvore não tiver nenhum verbete com significados, a complexidade da função se torna somente $O(n)$.

Complexidade de espaço

A complexidade de espaço da árvore AVL ocupada por n verbetes e m significados é $O(n+m)$, pois ela armazena todos os verbetes com seus significados em seus nós. Cada elemento da classe No ocupa 32 bytes, tornando a memória total da árvore $32(56n) + 40m$ bytes, pois cada nó tem necessariamente um único verbete.

4 Estratégias de Robustez

Para tratar da robustez do código, foram usadas as macros da biblioteca `msgassert.h`. O programa confere se foram passados arquivos de entrada, saída e memlog válidos. Caso não tenha sido informado um arquivo de entrada, o programa é terminado, mas caso não seja informado um arquivo de saída ou de memlog, o programa simplesmente cria um arquivo padrão na pasta bin e informa ao usuário o local e nome do arquivo. Também há um `erroAssert()` para impedir que o usuário insira uma estrutura que não seja árvore nem hash.

Tambem há uma condicional para impedir que sejam adicionados ao dicionário verbetes que não sejam de nenhum dos três tipos especificados (verbo, substantivo e adjetivo). É enviado um aviso ao usuário e aquele verbete é simplesmente ignorado na inserção

Na fórmula do hashing, mesmo que todas as entradas alfabéticas sejam calculadas para alguma posição dentro da tabela, há um `% tamanho` ao final, pois se o usuário inserir uma palavra contendo caracteres especiais, essa palavra ainda será inserida dentro do alcance da tabela, impedindo um *segmentation fault*. Essa foi a solução mais prática encontrada pois, apesar de não ser desejável ter palavras com caracteres especiais no dicionário, se o programa for interrompido ou alguma etapa for alterada durante o cálculo do hash, é possível que ocorram vazamentos de memória.

5 Análise Experimental

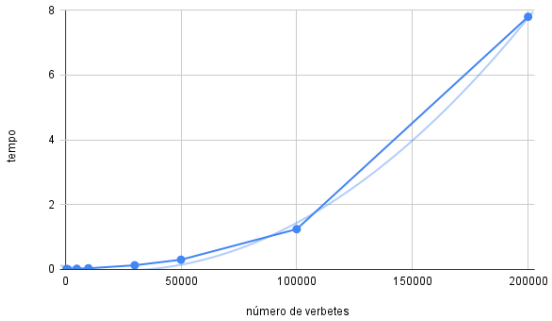
5.1 Método

Para analisar as estruturas, foi criado um script em c que recebe um total de 400 mil palavras da língua portuguesa e gera os arquivos no formato de entrada desejado. Assim, foram criados dois conjuntos de testes: no primeiro, foram criados apenas verbetes sem significados, para analisar como a quantidade de verbetes afeta o tempo do programa. No segundo, a quantidade de verbetes criados foi fixada em 20 e foram sendo adicionados cada vez mais significados a eles. Para medir os tempos, foram utilizadas as funções da biblioteca memlog.

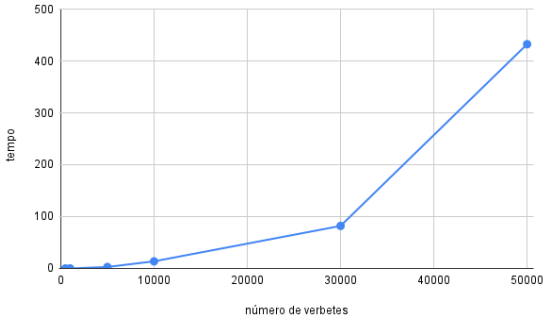
5.2 Resultados

Primeiro teste

No primeiro teste, foram criados arquivos com 500, 1 mil, 5 mil, 10 mil, 30 mil, 50 mil, 100 mil e 200 mil verbetes. Os resultados podem ser vistos nos gráficos abaixo:



(a) Hash

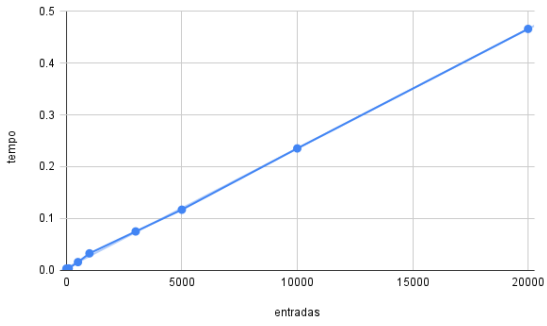


(b) Árvore AVL

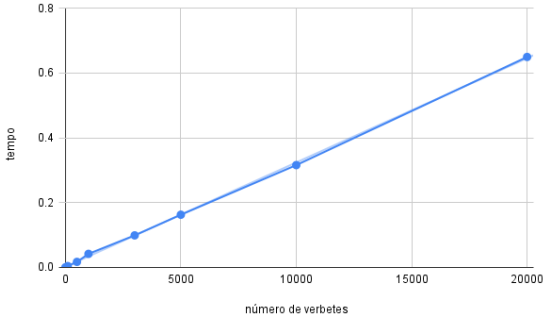
O gráfico mostra o tempo geral do programa para cada número de verbetes. A complexidade da inserção e remoção no hash é aproximadamente $O(n)$, por causa da lista encadeada utilizada nas colisões, enquanto o da árvore é $O(\log n)$. Porém, como o número de entradas também aumenta, a função representada nos gráficos se aproxima de $O(n^2)$ para o hash e $O(n \log n)$ para a árvore.

Segundo teste

No segundo teste, foram criados arquivos com 20 verbetes e 1, 100, 500, 1 mil, 3 mil, 5 mil, 10 mil e 20 mil significados para cada verbete. Os números foram menores que o teste anterior, pois o número de entradas já seria multiplicado por 20. Os resultados podem ser vistos nos gráficos abaixo:



(a) Hash



(b) Árvore AVL

Podemos perceber que o crescimento dessa análise é linear nos dois casos, pois em ambos dos casos a inserção do significado na lista de significados do verbete tem complexidade $O(n)$

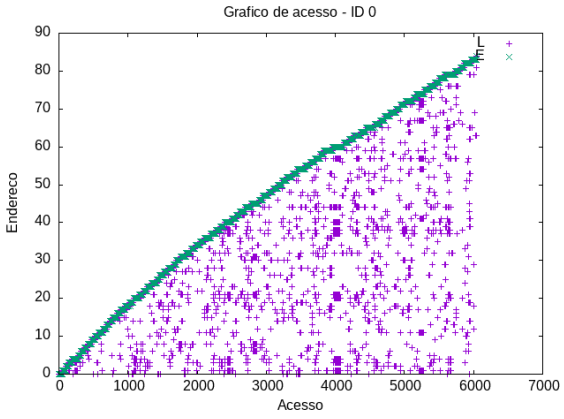
5.3 Analisamem

Para gerar os gráficos de acesso à memória e histogramas de pilha, foram utilizadas as funções e macros das bibliotecas `memlog` e `analisamem`. Paa os testes, foi utilizado o arquivo `input_02.txt`, disponibilizado pelos monitores da disciplina. Esse arquivo foi escolhido pois contém uma variedade de palavras, mas não contém muitas entradas, impedindo que o analisamem seja sobrecarregado.

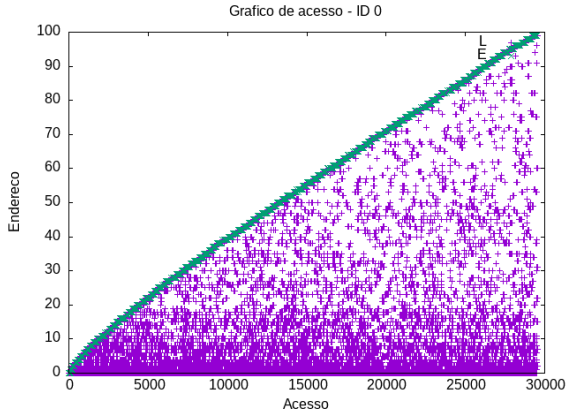
Inserção

Para o hash, foi inserido um `LEMEMLOG()` na pesquisa dentro da lista de verbetes do hash. Assim é possível ver a quantidade de verbetes que o sistema percorre para inserir cada entrada. Foi inserido também um `ESCREVEMEMLOG()` no momento onde o elemento é inserido.

Para a árvore, também foi inserido um `LEMEMLOG()` a cada vez que o programa não encontrou o verbete ainda e chama a função para um filho, ou quando o verbete já existe. O `ESCREVEMEMLOG()` representa o momento em que o elemento é inserido.



(a) Hash



(b) Árvore AVL

No gráfico do hash, podemos ver que nos momentos em que há colisão, o gráfico apresenta uma densidade levemente maior, pois realiza mais acessos em busca do elemento na lista para ser inserido. No gráfico da árvore, há uma densidade maior de leituras na parte debaixo do gráfico, identificando um maior número de chamadas da função próximo às folhas da árvore.

Os histogramas de pilha e distância de pilha total podem ser vistos abaixo:

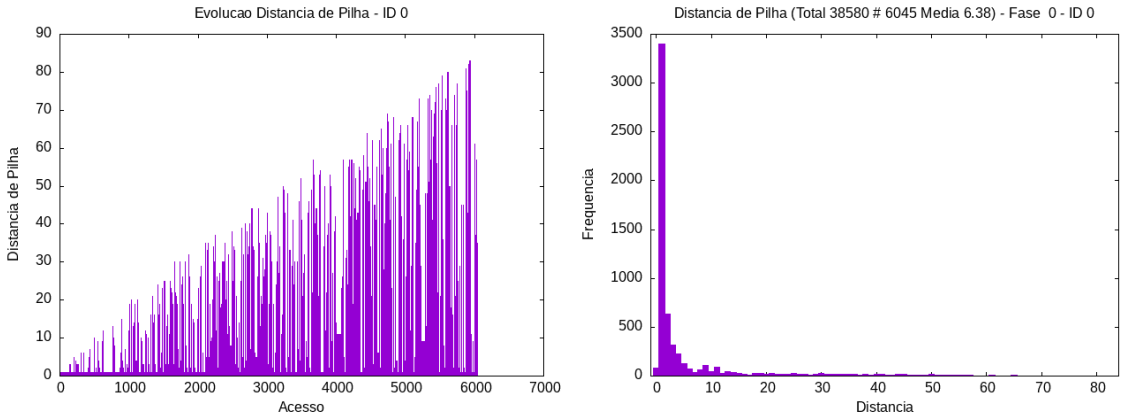


Figura 4: Histogramas do Hash

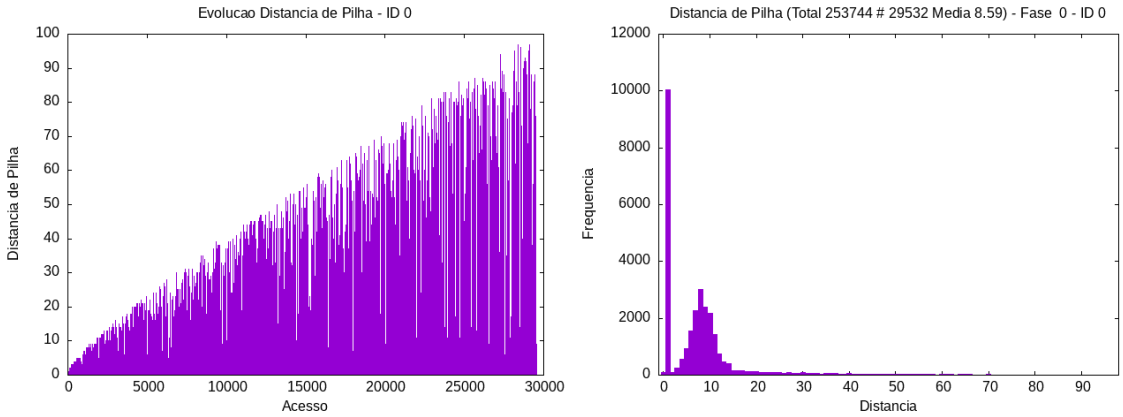
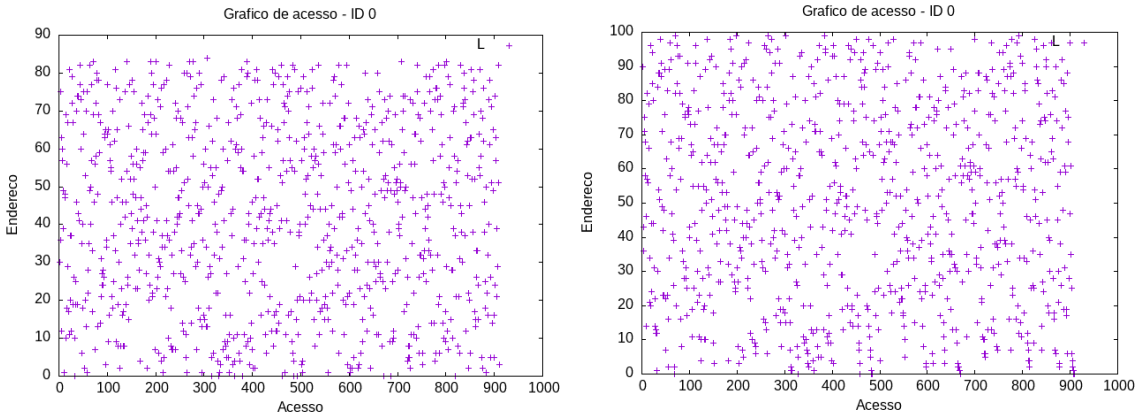


Figura 5: Histogramas da Árvore AVL

Remoção

Para a remoção dos verbetes com significado, nas duas estruturas foi inserido um `LEMEMLOG()` no momento em que um verbete com significado é encontrado.



(a) Hash (b) Árvore AVL

Os gráficos não são muito concisos devido ao acesso à memória não ser sequencial, pois os ponteiros que armazenam as posições na tabela do hash e os nós e seus filhos na árvore não são endereçados sequencialmente.

Os histogramas de pilha e distância de pilha total podem ser vistos abaixo:

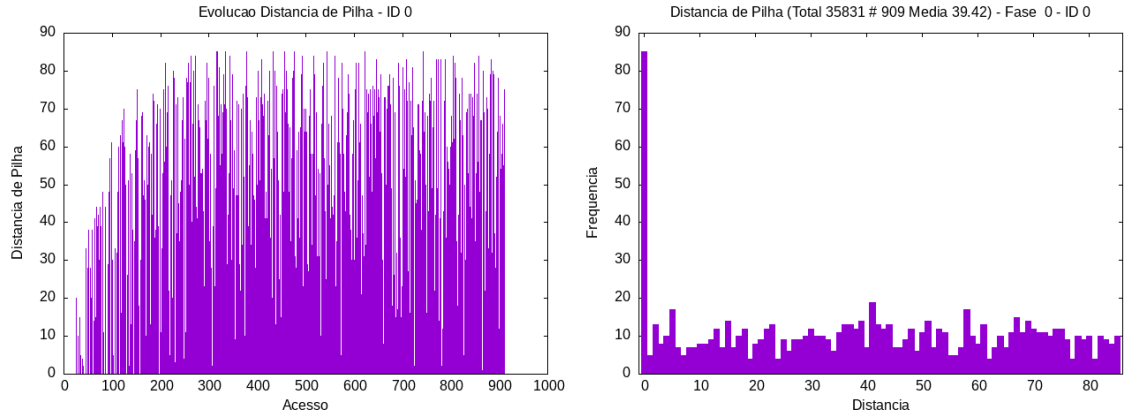


Figura 7: Histogramas do Hash

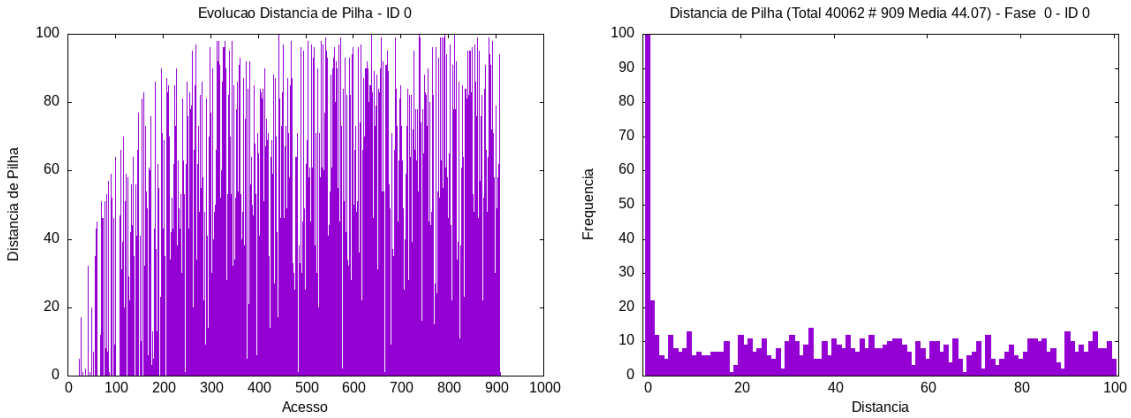
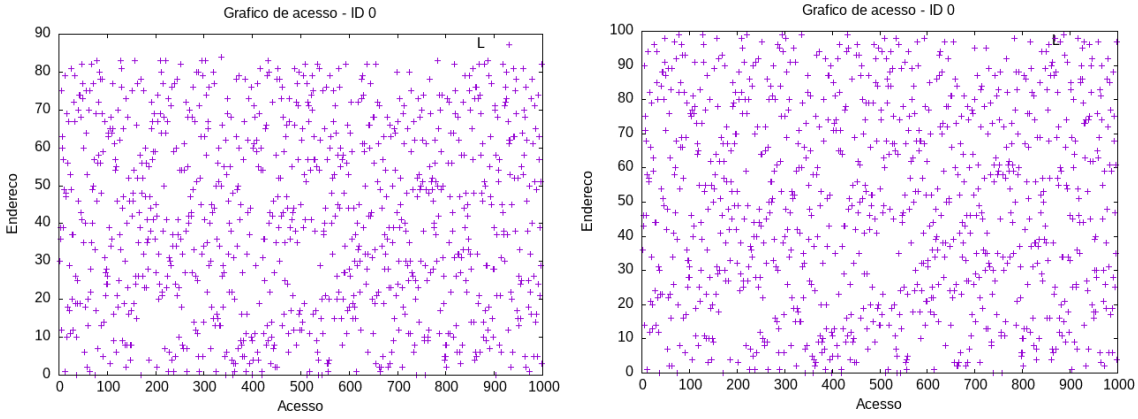


Figura 8: Histogramas da Árvore AVL

Impressao

Para a impressão, foi simplesmente inserido um `LEMEMLOG()` no momento em que o verbete é impresso, tanto no hash quanto na árvore.



(a) Hash

(b) Árvore AVL

Novamente, os gráficos não são muito concisos devido ao acesso à memória não ser sequencial.

Os histogramas de pilha e distância de pilha total podem ser vistos abaixo:

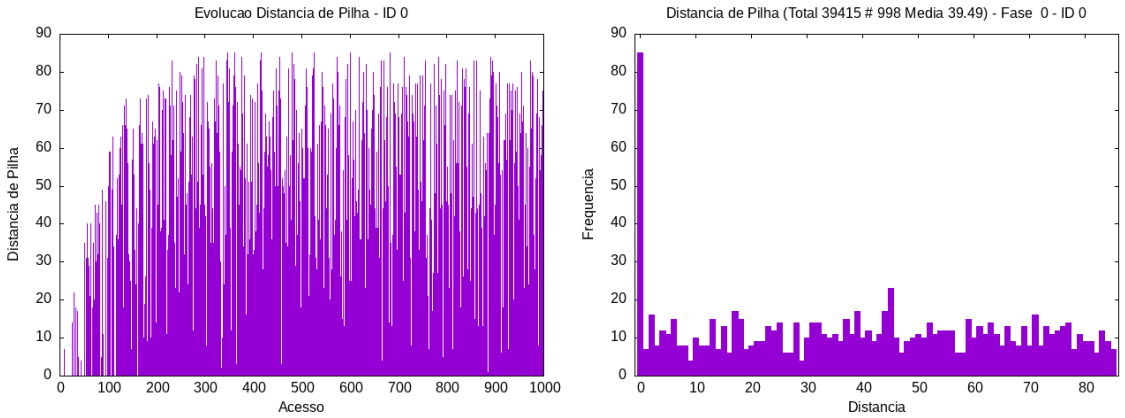


Figura 10: Histogramas do Hash

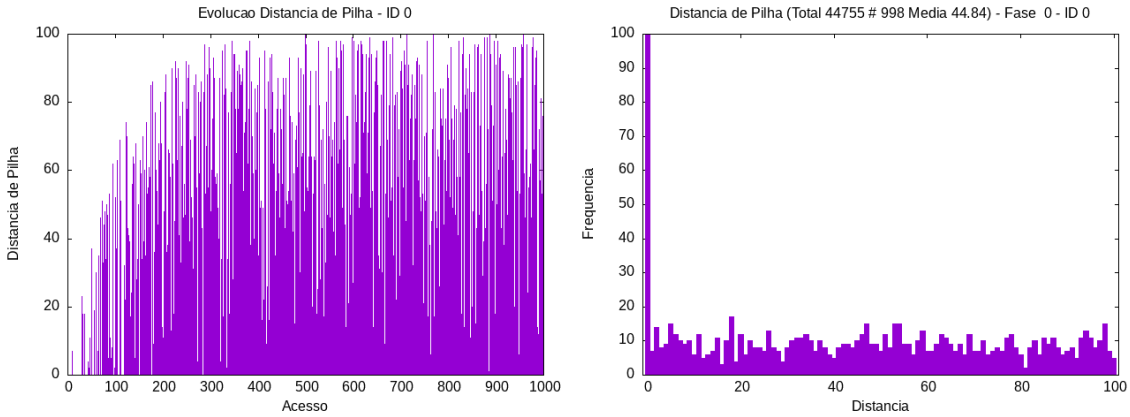


Figura 11: Histogramas da Árvore AVL

6 Conclusões

Durante esse trabalho, foram postos em prática os conhecimentos de Pesquisa Binária, Árvore AVL e Hash adquiridos na disciplina de **Estruturas de Dados**. Além disso, alguns conceitos utilizados em trabalhos anteriores também foi reaproveitado, como a Lista Encadeada e a Fila de Prioridade.

A estrutura mais desafiadora de ser desenvolvida foi a Árvore AVL, principalmente no entendimento sobre balanceamento e rotações. Outra etapa desafiadora foi pensar em um algoritmo para o hashing que fosse eficiente, e ajudasse no momento da impressão ordenada. O algoritmo pensado poderia ser melhor, pois posições da tabela que correspondem às palavras iniciadas em zzz ou yyy, por exemplo, raramente conterão alguma palavra. Porém, com o tempo escasso e recursos limitados, foi optado por um algoritmo que fosse simples e funcionasse.

Durante a implementação do sistema, também foi utilizado o **gdb**, depurador incorporado do VSCode, para encontrar erros de execução e lógica que não poderiam ser facilmente diagnosticados ao rodar o programa normalmente. Foi utilizado também o software **Valgrind** com testes de

tamanhos menores para identificar problemas de alocação e liberação incorretas de memória e para rastrear possíveis *memory leaks* que possam ter passado despercebidos.

Após a realização dos testes e da análise experimental, podemos compreender que as duas estruturas de dados utilizadas, **Hash** e **Árvore AVL**, são eficientes para o armazenamento e busca de palavras, sendo que a Árvore AVL é mais simples nesse caso, pois ela naturalmente armazena os verbetes em ordem alfabética, por ser uma árvore de busca binária, enquanto para o Hash é necessário um algoritmo de hashing que garanta a ordenação. Porém, a Árvore AVL é mais complexa de ser implementada, pois é necessário se preocupar com o balanceamento da árvore, enquanto o Hash é mais simples, pois não é necessário se preocupar com o balanceamento.

Por fim, a formulação do relatório se mostrou novamente como uma oportunidade para utilizar os conhecimentos do sistema de preparação de documentos LaTeX, uma ferramenta muito útil durante o percurso acadêmico, na escrita de artigos e documentos científicos, pois tem um foco no conteúdo escrito sem se preocupar com a formatação final.

Referências

D'AQUINO, P. Right way to conditionally initialize a C++ member variable?
Disponível em: https://stackoverflow.com/questions/1014518/. Acesso em: 13 dez. 2022.

std::Hash
Disponível em: https://cplusplus.com/reference/functional/hash/. Acesso em: 13 dez. 2022.

Using getopt() in C++ to handle arguments.
Disponível em: https://stackoverflow.com/questions/52467531/. Acesso em: 13 dez. 2022.

Instruções de Compilação e execução

Para executar o programa, siga os passos:

1. Acesse o diretório raiz `tp`;
2. Utilizando um terminal, utilize o comando `make` para compilar o código;
3. Com esse comando, um arquivo `run.out` será criado no diretório `bin`;
4. Execute usando `./bin/run.out` e os argumentos desejados em seguida:
    - `-i <nome do arquivo>` (arquivo de entrada)
    - `-o <nome do arquivo>` (arquivo de saída)
    - `-t <nome da estrutura>` (estrutura utilizada para armazenar o dicionário (`hash` ou `arv`))
    - `-l <nome do arquivo>` (arquivo de registro do memlog - opcional)

Exemplos de comandos de entrada

* Os comandos que não inserirem o argumento `-l` terão a saída do memlog escrita no arquivo padrão `/bin/memlog.out`

- `./bin/run.out -i entrada.txt -o saida.txt -t arv`
- `./bin/run.out -i entrada.txt -o saida.txt -t hash`
- `./bin/run.out -i entrada.txt -o saida.txt -t hash -l bin/arquivoMemLog.out`
- `./bin/run.out -v 1 -s 10 -i entrada.txt -t arv`

* Esse comando terá sua saída escrita no arquivo padrão `/bin/saida.txt` pois não inseriu a flag `-o`

Formato do arquivo de entrada

O arquivo de entrada deve ser um arquivo de texto (em formato `.txt`, por exemplo) contendo os verbetes na seguinte forma:

`a [applied] concerned with concrete problems or data`

onde:

a indica o tipo do verbete (adjetivo, nome ou verbo);

applied é o verbete;

concerned with concrete problems or data é o significado do verbete.

Exemplo de formato de arquivo de entrada

entrada.txt

a [bad] immoral, evil
a [bad] below average in quality or performance, defective
a [bad] spoiled, spoilt, capable of harming
a [artistic]

a [ashamed]
a [asleep]

Formato do arquivo de saída

O arquivo de saída contém a impressão do dicionário após a inserção, com cada verbete e seus significados. Ao funal dessa impressão, é impresso novamente o dicionário, sem os verbetes que contém significados.

Exemplo de formato de arquivo de saída

saida.txt

artistic (a)
ashamed (a)
asleep (a)
bad (a)
1. immoral, evil
2. below average in quality or performance, defective
3. spoiled, spoilt, capable of harming
artistic (a)
ashamed (a)
asleep (a)