Dicionário

Marcelo Andrade

github.com/marceloanje
linkedin.com/in/marcelo-andrade-10334b160
marcelo.anje@outlook.com

1. Introdução

Uma das principais necessidades em sistemas de computação é a de armazenar e gerenciar grandes quantidades de dados. Um dicionário é uma estrutura de dados que contém como elemento como elemento suas palavras, e em cada palavra contém seu respectivo significado, o que torna muito útil para armazenar e recuperar informações de forma rápida e eficiente, quando o mesmo se encontra em ordem alfabética.

Neste trabalho, foi implementado um dicionário utilizando duas técnicas de estrutura de dados: árvore AVL e tabela hash. A árvore AVL é uma árvore de busca balanceada que garante a eficiência na inserção, exclusão e recuperação de dados. A tabela hash é uma estrutura de dados que utiliza funções hash para mapear chaves em posições de uma tabela, permitindo acesso rápido aos dados.

Utilizando essas duas técnicas, conseguimos criar um dicionário eficiente e rápido, que foi testado e comparado com ambas as implementações.

A seção dois é onde está a descrição dos métodos adotados, a seção três, a análise de complexidade, a seção quatro, as estratégias de robustez adotadas, a seção cinco, a análise experimental, e por fim, a seção seis, a conclusão.

2. Método

2.1 Implementação

O programa foi desenvolvido na linguagem C++, compilada pelo WSL da Ubuntu e os testes foram realizados em um emulador de sistemas operacionais (VirtualBox) com Linux Ubuntu 20.04.4, sendo emulado em um computador com Windows 10, Ryzen 7 2.3 GHz e 8GB de memória RAM.

2.2 Estrutura de dados

A implementação do programa teve como base algumas estruturas de dados, mas as duas principais estruturas de dados utilizadas foram, a *árvore binária de busca balanceada (AVL)* e a *tabela hash*.

Há também uma lista encadeada do tipo *Significado*, que está contida na classe *Verebete*.

A árvore binária de busca balanceada (AVL) é caracterizada por conter elementos da classe *No*, que contêm um elemento da classe *Verbete*.

A tabela hash é caracterizada por conter o tipo de tratamento de colisões por lista encadeada, os elementos que formam a tabela são do tipo *Celula*, que contém um elemento da classe *Verbete*.

2.3 Formatos de dados

O sistema foi feito de tal forma, que durante sua execução é recebido um caminho o qual deve se encontrar um arquivo de entrada no formato *.txt*, o qual contém os dados de entradas que são tratados internamente, dessa forma a realizar todo mecanismo de armazenagem, utilizando o método passado por linha de comando.

Os dados de entrada devem seguir o seguinte modelo:

a [applied] concerned with concrete problems or data

Onde:

a - indica o tipo de verbete (adjetivo, nome ou verbo);
[applied] - é verbete (sem os colchetes);
concerned with concrete problems or data - é o significado do verbete;

2.4 Classes e métodos

O programa foi dividido primordialmente em seis classes diferentes: classe ***Significado***, classe ***Verbete***, classe ***DicionarioAVL***, classe ***Celula***, classe ***Lista*** e classe ***DicionarioHash***. Além dessas três classes, há também o arquivo ***main.cpp***.

A classe ***Significado*** possui dois atributos privados, que são, *texto* do tipo *string* e um atributo ponteiro *prox* do tipo *Significado*, que é usado na manipulação da lista de *Significado*. Essa classe possui três métodos, que são eles, um construtor default *Significado*, e métodos básicos para manipulação do atributo *texto*, *getTexto* e *setTexto*. Essa classe é *friend* com a classe *Verbete*.

A classe ***Verbete*** possui cinco atributos privados, que são, *palavra* e *classe*, ambas do tipo *string*, *contador* do tipo *int*, e dois atributos ponteiro do tipo *Significado*, *primeiro* e *ultimo*, que são usados na manipulação da lista de *Significado*. Essa classe possui oito métodos, que são eles, um construtor default *Verbete*, e métodos básicos para manipulação dos atributos privados, que são eles, *getPalavra*, *getClasse*, *getContador*, *setPalavra*, *serClasse*, *adicionaSignificado*, método que adiciona um elemento do tipo *Significado* ao final da lista encadeada contida na classe e *imprimeSignificados*, método que imprime a lista de *Significado*.

A classe ***DicionarioAVL*** possui cinco atributos privados, que são, *chave* do tipo *string*, *altura* do tipo *int*, *verbete* do tipo *Verbete*, e dois atributos ponteiro do tipo *DicionarioAVL*, *esq* e *dir*, que são usados na manipulação da árvore de busca balanceada. Essa classe possui dezenove métodos, que são eles, um construtor default *DicionarioAVL*, um método *destroiAVL*, usado para destruir a árvore, métodos básicos para manipulação dos atributos privados, que são, *getAltura*, *getEsq*, *getDir*, *setAltura*, *setEsq*, *setDir*. Os outros métodos são, *max*, método que retorna o maior número entre dois passados por parâmetro, *Altura*, método que retorna o valor da altura de um nó da árvore, *novoNo*, método que cria um novo nó, *rotacaoEsq* e *rotacaoDir*, métodos que realizam a rotação de nós, quando os mesmos se encontram desbalanceados, *getFatorBalanco*, método que retorna o fator de balanço de um nó, *insereNo*, método que adiciona um novo nó a árvore, *valorMinimoNo*, método que retorno o nó com valor mínimo, *deletaNo*, método que remove um nó da árvore, *Busca*, método que retorna o nó que está sendo buscado através de sua chave, *Existe*, método

que retorna se um nó existe na árvore, *alteraNo*, método usado para alterar um nó da árvore, *Imprime*, método usado para imprimir a árvore, *removeImprime*, método usado para remover verbetes com mais de um significado e imprimir árvore.

A classe ***Celula*** possui dois atributos privados, que são, *verbete* do tipo *Verbete* e um atributo ponteiro *prox* do tipo *Celula*, que é usado na manipulação da lista de *Celula*. Essa classe possui um método, que é um construtor default *Celula*. Essa classe é *friend* com a classe *Lista*.

A classe ***Lista*** possui dois atributos privados, que são ponteiro do tipo *Celula*, *primeiro* e *ultimo*, que são usados na manipulação da lista de *Celula*. Essa classe possui nove métodos, que são eles, um construtor default *Lista* e um método destrutor *destroiLista*. Outros métodos são, *insereOrdenado*, método que adiciona elemento do tipo *Celula* a lista encadeada de forma ordenada, *removeVerbete*, método que remove um elemento específico da lista, *Pesquisa*, método retorna um elemento da lista de acordo com a chave passada, *imprimeLista*, método imprime a lista, *Existe*, método que verifica se um elemento existe na lista, *Busca*, método que busca um elemento na lista encadeada, *alteraCelula*, método que altera um elemento da lista encadeada adicionando um novo significado ao verbete contido no elemento, *excluiVerbeteSignificado*, método que remove todos os elementos que contém um Verbete não estejam com a lista de *Significado* vazia.

A classe ***DicionarioHash*** possui três atributos privados, que são, *M* do tipo *static const int*, que tem atribuído um valor de 26, um método privado *buscaLista*, que retorna em qual lista encadeada o programa deve atuar, é a função de transformação, *Tabela* do tipo *Lista*, com tamanho *M*. Essa configuração mostra que a tabela *hash* irá usar lista encadeada para tratar colisões e dessa forma terá 26 listas encadeadas, uma para cada letra do alfabeto. Essa classe possui oito métodos, que são eles, um construtor default *DicionarioHash* e um destrutor *destroiHash*. Outros métodos são, *Pesquisa*, método que busca e retorna um elemento buscado pela sua chave, *Insere*, método que insere um elemento em sua posição na tabela, *Remove*, método que remove um determinado elemento, *imprimeHash*, método que imprime toda tabela, *Existe*, método que retorna se determinado elemento existe na tabela, *adicionaSignificado*, método que altera um elemento contido na tabela, *removeVerbetes*, método que remove todos elementos da tabela cujo estejam com sua lista de *Significado* vazia.

O arquivo ***main.cpp*** possui toda as chamadas de funções e possui quatro funções próprias a ele, que são: *imprimeSugestao*, função que imprime na tela a sugestão de escrita para linha comando, *numeroPalavras*, função que retorna o número de palavras contidas em uma string e métodos *metodoAVL* e *metodoHASH* que realizam a mecânica de leitura e chamada de métodos a depender do parâmetro *-t* de entrada.

3. Análise de complexidade

A análise de complexidade do programa foi analisada os métodos relevantes de cada classe e o *main*. Dessa forma, sendo analisada no âmbito do tempo e espaço.

3.1 Complexidade temporal

Na classe ***Significado***, não há nenhum tipo de comparação, somente atribuições por meio dos métodos *set*, *get* e dos construtores, podendo ser considerados como O(1).

Na classe ***Verbete,*** será feita a análise método a método a seguir:
Verbete método construtor default, possui algumas atribuições - O(1);
getPalavra método básico, possui somente um retorno - O(1);
setPalavra método básico, possui somente uma atribuição - O(1);
getClasse método básico, possui somente um retorno - O(1);
setClasse método básico, possui somente uma atribuição - O(1);
adiconaSignificado método possui algumas atribuições e comparações, mas aciona internamente o construtor default *Significado*, que é O(1), sendo assim, suas ordens são somadas - O(1);
imprimeSignificados método possui algumas atribuições e um loop *while* - O(n);
getContador método básico, possui somente um retorno - O(1);

Para saber a complexidade da classe, basta somar a complexidade de cada método, que resulta em O(n).

Na classe ***DicionarioAVL,*** será feita a análise método a método a seguir:
DicionarioAVL método construtor default - O(1);
destroiAVL método destrutor com chamada recursiva -
O(n); *getAltura* método básico, possui somente um retorno -
O(1);
setAltura método básico, possui somente uma atribuição -
O(1); *getEsq* método básico, possui somente um retorno - O(1);
setEsq método básico, possui somente uma atribuição - O(1);
getDir método básico, possui somente um retorno - O(1);
setDir método básico, possui somente uma atribuição - O(1);
max método possui uma comparação - O(1);
Altura método possui uma comparação, mas aciona internamente o método *getAltura*, que é O(1), sendo assim, suas ordens são somadas - O(1);
novoNo método possui algumas atribuições, mas aciona internamente o construtor default *DicionarioAVL*, que é O(1), sendo assim, suas ordens são somadas - O(1);
rotacaoEsq método aciona internamente os métodos *getDir*, *getEsq*, *setEsq*, *setDir*, *setAltura*, *max* e *Altura*, que todos são O(1), sendo assim, suas ordens são somadas - O(1);
rotacaoDir método aciona internamente os métodos *getDir*, *getEsq*, *setEsq*, *setDir*, *setAltura*, *max* e *Altura*, que todos são O(1), sendo assim, suas ordens são somadas - O(1);
getFatorBalanco método possui uma comparação, mas aciona internamente os métodos *Altura*, *getEsq* e *getDir* que todos são O(1), sendo assim, suas ordens são somadas - O(1);
insereNo método possui algumas atribuições e comparações, mas aciona internamente os métodos *novoNo*, *getPalavra*, *max*, *Altura*, *getFatorBalanco*, *rotacaoDir*, *rotacaoEsq*, que todos são O(1), e também aciona recursivamente ele mesmo, sendo assim, suas ordens são somadas e feito a análise recursiva - O(log n);
valorMinimo método possui uma atribuição e comparações e um loop *while* - O(n);
deletaNo método possui algumas atribuições e comparações, mas aciona internamente os métodos *getPalavra*, *setPalavra*, *Altura*, *max*, *getFatorBalanco*, *rotacaoDir*, *rotacaoEsq*, que

todos são O(1), aciona também o método *valorMinimo*, O(n), também aciona recursivamente ele mesmo, sendo assim, suas ordens são somadas e feito a análise recursiva - O(log n);

Busca método possui algumas comparações, mas aciona internamente o método *getPalavra*, O(1), e também aciona recursivamente ele mesmo, sendo assim, suas ordens são somadas e feito a análise recursiva - O(log n);
Existe método possui uma comparação, mas aciona internamente o método *Busca*, que é O(log n), sendo assim, suas ordens são somadas - O(log n);
alteraNo método aciona internamente o método *adiconaSignificado*, O(1), da classe *Verbete* - O(1);
Imprime método aciona internamente o método *imrpimeSignificados*, O(n), da classe *Verbete*, e também aciona recursivamente ele mesmo, sendo assim, suas ordens são somadas e feito a análise recursiva - $O(n^2)$;
removeImprime método aciona internamente o método *imrpimeSignificados*, O(n), da classe *Verbete*, também aciona internamente o método *getContador*, O(1), da classe *Verbete*, e também aciona recursivamente ele mesmo, sendo assim, suas ordens são somadas e feito a análise recursiva - $O(n^2)$;

Para saber a complexidade da classe, basta somar a complexidade de cada método, que resulta em O(n).

Na classe ***Celula***, não há nenhum tipo de comparação ou atribuições, somente um construtor default, podendo ser considerados como O(1).

Na classe ***Lista,*** será feita a análise método a método a seguir:
Lista método construtor default, possui algumas atribuições - O(1);
destroiLista método destrutor, possui algumas atribuições e comparações, possui um loop *while* - O(n);
Pesquisa método possui algumas comparações e atribuições e um loop *while*, mas aciona internamente o método *getPalavra*, O(1), da classe *Verbete*, sendo assim, suas ordens são somadas - O(n);
insereOrdenado método possui algumas comparações e atribuições e um loop *while*, mas aciona internamente o método *getPalavra*, O(1), da classe *Verbete*, também aciona o método construtor default *Celula*, O(1), sendo assim, suas ordens são somadas - O(n);
removeVerbete método possui algumas comparações e atribuições e um loop *while*, mas aciona internamente o método *getPalavra*, O(1), da classe *Verbete*, sendo assim, suas ordens são somadas - O(n);
imprimeLista método possui algumas comparações e atribuições e um loop *while*, mas aciona internamente o método *getPalavra*, O(1) e *imprimeSignificados*, O(n), ambos da classe *Verbete*, sendo assim, suas ordens são multiplicadas - $O(n^2)$
Existe método possui algumas comparações e atribuições e um loop *while*, mas aciona internamente o método *getPalavra*, O(1), da classe *Verbete*, sendo assim, suas ordens são somadas - O(n);
Busca método possui algumas comparações e atribuições e um loop *while*, mas aciona internamente o método *getPalavra*, O(1), da classe *Verbete*, sendo assim, suas ordens são somadas - O(n);

alteraCelula método aciona internamente o método *Busca*, O(n) e *adiconaSignificados*, O(1), da classe *Verbete*, sendo assim, suas ordens são somadas - O(n);
excluiVerbetesSignificado método possui algumas comparações e atribuições e um loop *while*, mas aciona internamente o método *removeVerbete*, O(n), e os métodos *getPalavra*, O(1) e *getContador*, O(1), ambos da classe *Verbete*, sendo assim, suas ordens são somadas - $O(n^2)$;

Para saber a complexidade da classe, basta somar a complexidade de cada método, que resulta em $O(n^2)$.

Na classe ***DicionarioHash,*** será feita a análise método a método a seguir:
buscaLista método possui algumas comparações e atribuições - O(1);
DicionarioHash método construtor default - O(1);
destroiHash método destrutor, aciona internamente o método *destroiLista*, O(n), da classe *Lista* vinte e sete vezes - O(n);
Pesquisa método possui algumas atribuições, mas aciona internamente o método *buscaLista*, O(1), e o método *Pesquisa*, O(n), da classe *Lista*, sendo assim, suas ordens são somadas - O(n);
Insere método possui algumas atribuições, mas aciona internamente o método *buscaLista*, O(1), o método *getPalavra*, O(1), da classe *Verbete*, e o método *insereOrdenado*, O(n), da classe *Lista*, sendo assim, suas ordens são somadas - O(n);
Remove método possui algumas atribuições, mas aciona internamente o método *buscaLista*, O(1), e o método *removeVerbete*, O(n), da classe *Lista*, sendo assim, suas ordens são somadas
- O(n);
imprimeHash método possui um loop *for*, mas esse loop sempre tem o mesmo tamanho, sendo assim torna-se O(1), por ser um custo constante, também aciona internamente o método o método *imprimeLista*, $O(n^2)$, da classe *Lista*, sendo assim, suas ordens são multiplicadas - $O(n^2)$;
Existe método possui algumas atribuições, mas aciona internamente o método *buscaLista*, O(1), e o método *Existe*, O(n), da classe *Lista*, sendo assim, suas ordens são somadas - O(n);
adiconaSignificado método possui algumas atribuições, mas aciona internamente o método *buscaLista*, O(1), e o método *alteraCelula*, O(n), da classe *Lista*, sendo assim, suas ordens são somadas - O(n);
removeVerbetes método possui um loop *for*, mas esse loop sempre tem o mesmo tamanho, sendo assim torna-se O(1), por ser um custo constante, também aciona internamente o método o método *excluiVerbeteSignificado*, $O(n^2)$, da classe *Lista*, sendo assim, suas ordens são multiplicadas - $O(n^2)$;

Para saber a complexidade da classe, basta somar a complexidade de cada método, que resulta em $O(n^2)$.

O ***main*** possui um loop while, logo a complexidade é O(n). Dentro do *main* são acionados alguns métodos das classes do sistema, como a maior ordem de complexidade das classes é $O(n^2)$, dessa forma podemos considerar que o *main* como um todo possui ordem de complexidade $O(n^3)$.

Com isso, tem se as ordens de complexidade de todos os métodos e partes do programa, como a ordem de complexidade total é a soma de todas ordens, pode-se concluir que a ordem de complexidade total do sistema é $O(n^3)$.

3.2 Complexidade espacial

A análise de espaço pode se avaliar os dois principais meios de armazenamento do sistema, que são a árvore binária de busca balanceada e a tabela *hash*.

A árvore binária de busca balanceada é constituídas por vários nós do mesmo tipo, cada nó dessa estrutura possui um elemento do tipo *Verbete*, e cada *Verbete* possui uma lista encadeada de elementos do tipo *Significado*, dessa forma, podemos considerar que o espaço ocupado é $\mathbf{O(n^2)}$, pois a árvore possui somente duas dimensões. Pode se analisar também que a complexidade não irá se alterar independe do tamanho das listas, visto que, $c*O(f(n)) = O(f(n))$.

A tabela *hash* é constituída por uma lista que contém 26 listas encadeadas, e os elementos que compõem essas listas encadeada, são do *Verbete*, e cada *Verbete* possui uma lista encadeada de elementos do tipo *Significado*, dessa forma, podemos considerar que o espaço ocupado é $\mathbf{O(n^2)}$, pois a tabela *hash*, possui 26 listas com ordem espacial $O(n^2)$, mas por ser uma valor constante, não afeta no valor da ordem final avaliada. Pode se analisar também que a complexidade não irá se alterar independe do tamanho das listas, visto que, $c*O(f(n)) = O(f(n))$.

A partir dessas análises, pode-se considerar que a complexidade espacial no programa como um todo é $\mathbf{O(n^2)}$.

4. Estratégias de robustez

Uma vez que o usuário do software possa utilizar o programa e venha a passar argumentos errados ou mesmo não passar todos argumentos necessários para correr o programa, foi adicionado validação dos parâmetros passados como argumento via terminal de modo a verificar se todos os parâmetros necessários foram passados ou se a sequência não está errada

5. Análise experimental

Para a análise de desempenho do programa, foi utilizado a função *clock* da biblioteca *time.h*. A análise de desempenho foi realizada no software em dois modos, no primeiro utilizando a estrutura de dados da tabela *hash* e o segundo utilizado a estrutura de dados da árvore *AVL*, para cada um desses modos foram passados sete arquivos com um número crescente de linhas, os mesmos arquivos para cada um dos métodos. Com isso, foi possível calcular o desempenho computacional de ambos os modos.

Para a análise, foi realizado uma média entre três tempos para cada arquivo em cada modo de estrutura, com isso, chegou aos seguintes resultados.

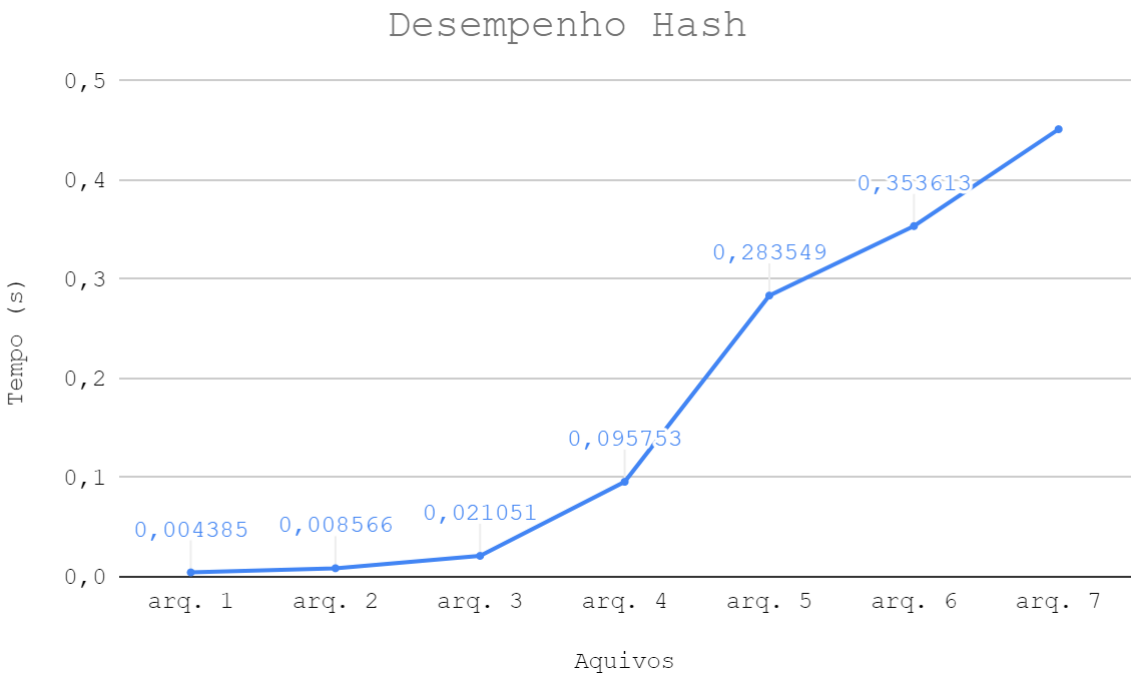

Gráfico 1 - desempenho Hash em segundos

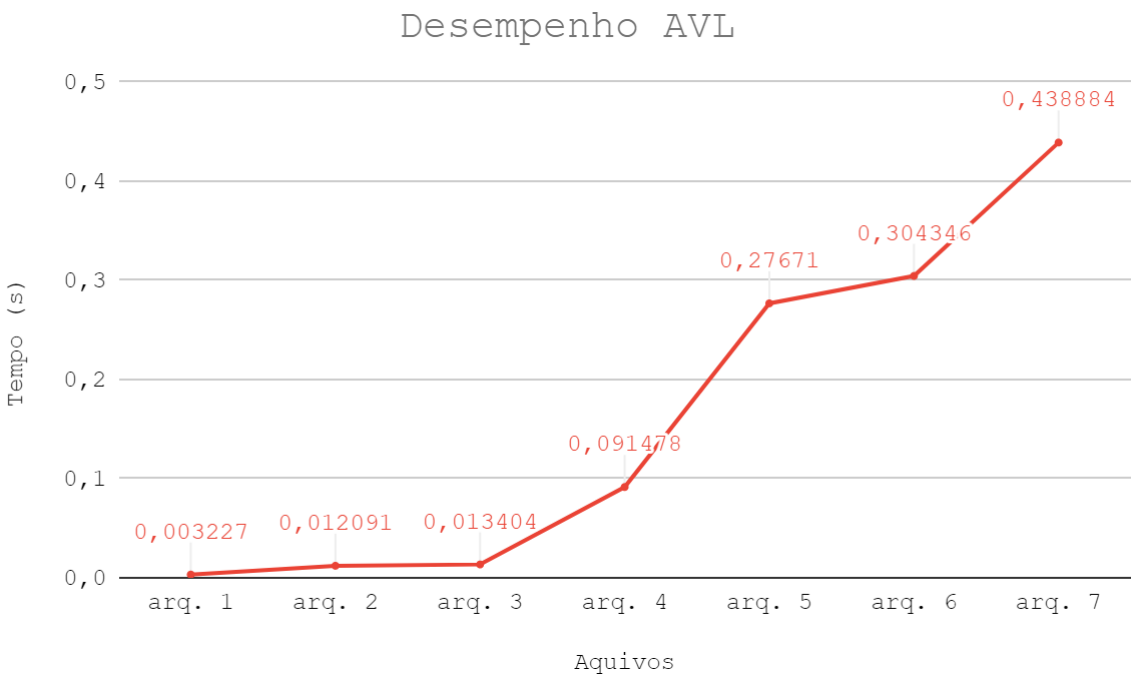

Gráfico 2 - desempenho AVL em segundos

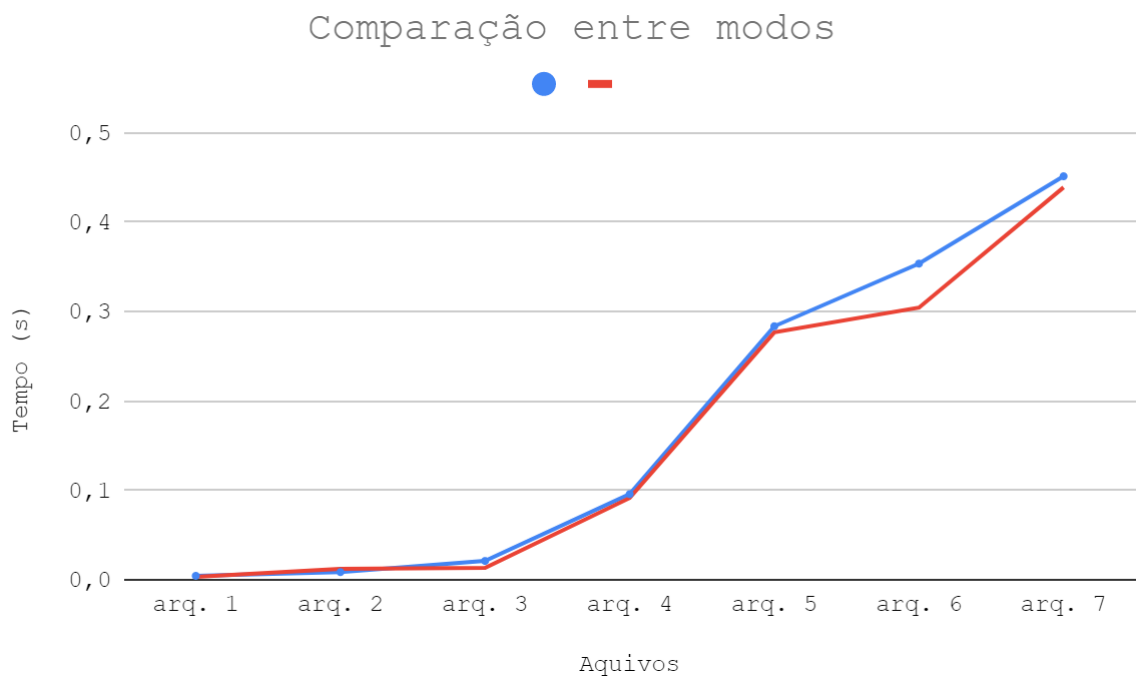

Gráfico 3 - Comparação entre modos

A partir dessa análise é possível perceber que a implementação que utiliza a árvore *AVl* é ligeiramente mais eficiente do que a que utiliza tabela *hash*. Um dos motivos para esse desempenho superior, pode ser a alocação de memória que a tabela *hash* faz de 27 listas estáticas, mesmo que não utilize todas as listas.

Ambas as implementações possuem ordem de complexidade total $O(n^2)$, no entanto, os métodos de pesquisa, inserção e remoção da árvore *AVL*, possuem ordem O(log n), enquanto na tabela *hash*, esses métodos possuem ordem O(n), essa pode ser outra causa pela qual a árvore *AVL* é mais eficiente, visto que esses métodos são os mais utilizados em uma execução.

A vantagem da implementação utilizando tabela *hash* é sua maior facilidade de implementação, outra vantagem é o maior controle dos dados, pois utilizando a função *hash* para localizar o intervalo de dados. A desvantagem vista, foi que ela possui um desempenho um pouco inferior ao da árvore *AVL*.

A vantagem da implementação utilizando árvore *AVL* é sua manutenção da altura da árvore e seu balanceamento, o que acaba resultando em operações de ordem de complexidade logaritmos, e decorre uma maior eficiência, outra vantagem é sua utilização de memória, que somente aloca o necessário. A desvantagem da árvore *AVL* é sua complexidade de implementação.

6. Conclusões

Este trabalho teve como objetivo a implementação de um dicionário que usa a árvore binária de busca balanceada e a tabela *hash*, como suas estruturas de dados, a partir disso foi desenvolvido um programa em C++ para atender a todas as especificações propostas..

Com esse desenvolvimento foi possível assimilar bem novos conceitos, como o de métodos de ordenação na inserção,estruturas de dados, como árvore *AVL* e tabela *hash*, modificando e analisando possíveis tratamentos para com essas estruturas.

Referências

Ziviani, N., **Projeto de Algoritmos com Implementações em Pascal e C**, 3ª Edição, Cengage Learning, 2011.

Instruções de compilação e execução

Para a execução do programa sigas os passos descritos abaixo, utilizando o Makefile para isso:

- Utilizando o terminal acesse o diretório;
- Execute o arquivo Makefile utilizando o seguinte comando: “make”;
- Após esse comando, na pasta raiz, utilize os seguintes comandos:

  run.out -i entrada.txt -o saida.txt -t arv

  run.out -i entrada.txt -o saida.txt -t hash
- Com esse comando deve ser gerado um arquivo com o resultado final;
- Errata 1: O nome “entrada.txt” e “saida.txt” destinam a um exemplo de nome, para a compilação é necessário especificar o caminho e o nome do arquivo, ou somente colocar o arquivo de entrada no diretório e especificar o nome do arquivo.