Trabalho Prático 3

Mariana Assis Ramos

2021039360

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

Introdução

Esse trabalho prático tem como objetivo a análise de dois mecanismos conhecidos de pesquisa: a tabela Hash e a árvore de pesquisa balanceada, mais especificamente a AVL.

Para tal análise, foi requisitado a construção de um dicionário de palavras com as duas estruturas. Esse dicionário deve ser capaz de fazer consultas, inserções, retiradas, impressões ordenadas e atualização de entrada. Buscando computar a carga de trabalho de cada implementação e os seus devidos custos.

Tais custos foram analisados principalmente nas funções de pesquisa, remoção e inserção, tanto a complexidade de tempo quanto a de espaço.

Método

O programa foi desenvolvido na linguagem c++, compilada pelo compilador G++ da GNU Compiler Collection.

Apesar de ser requisitado diferentes operações oque realmente é cobrado é que o diccionario armazene verbetes, imprima estes de forma ordenada, remova os verbetes que apresentem significado e depois imprima o dicionario novamente sem os verbetes de significado.

- ***VERBETE***

Verbete é uma class que foi implementada para armazenar cada palavra do dicionário. Ele é composto de um tipo, que pode ser adjetivo (a), verbo (v) ou substantivo (s). Também apresenta uma string com o nome da palavra do verbete em si. E um outro TAD chamado ListSignificados, que é uma Lista com os diversos significados de cada veberbete, podendo ser vazio também.

O Verbete é um TAD que não possui métodos, tendo sua principal função armazenamento de informações, incluindo o TAD com significados.

- ***LISTSIGNIFICADOS***

ListSignificados é um TAD Lista Encadeada que armazena os diferentes significados de cada Verbete. Apresenta 2 métodos: insereSignificados e ImprimirSignificadosParaDic.

- InsereSignificados: Esse método realiza a inserção de um novo significado. Essa inserção acontece somente quando o significado não for uma string vazia.

- ImprimirSignificadosParaDic: Essa função realiza a impressão de todos os significados que esse TAD armazena na ordem da inserção. Útil para a impressão dos verbetes sem a repetição de código.

➢ *ÁRVORE AVL*

AVL é um tipo especial da Arvore Binária de Pesquisa, sua principal característica é que para cada no da AVL a diferença entre as sub-árvores da direita e da esquerda é 1, 0 ou -1. Essa característica faz com que seja uma árvore Pseudo-Balanceada, já que se mantém balanceada depois de cada inserção/remoção. Isso faz com que a complexidade para essas operações diminua (será abordado na seção de análise de complexidade).

Para se manter balanceada usa operações de rotação de nó, que mantém as alturas das sub-árvores dentro dos valores necessários. Existem 4 operações de rotação:

- Rotação Direita: inserir na esquerda e avô com desbalanceamento negativo;
- Rotação Esquerda: inserir na direita e avô com desbalanceamento positivo;
- Rotação Esquerda-Direita: inserir na direita e avô com desbalanceamento negativo. Rotação para a Esquerda seguida de uma para a Direita. 2 rotações;
- Rotação Direita-Esquerda: inserir na esquerda e avô com desbalanceamento positivo. Rotação para a Direita seguida de uma para a Esquerda. 2 rotações;

Para conseguir realizar essas operações de rotação cada nó armazena sua própria altura.

Uma AVL realiza as mesmas operações de uma Arvore binaria não balanceada, a única diferença é que após cada operação de inserção/remoção as operações de balanceamento também são chamadas. Primeiro se atualiza a altura do no, ao pegar a altura máxima entre os dois filhos e somar 1, depois esse nó (já com a altura correta) é passado pra função balancear, que realiza as rotações caso necessário.

Os métodos para manter a árvore balanceada são:

- getAltura: Retorna a altura do no e caso seja um ponteiro nulo retorna 0, evitando erros nas operações.
- Balanceamento: Retorna o valor do fator de balanceamento, que é a diferença entre as alturas da sub-árvore da direita e da esquerda. Para esse cálculo usa a função getAltura.
- Balancear: Olha o fator do balanceamento do respectivo nó e analisa os 4 casos de rotação para balanceá-lo caso esteja desbalanceado (para ficar balanceado precisa ficar entre -1 e 1).

Já os métodos para o funcionamento do dicionário são passados para as funções recursivas que irão realizar as operações e chamados por funções de nome equivalente. Esses métodos recursivos são:

- insereDicRec: Insere um novo verbete na árvore de forma ordenada, os menores na esquerda e os maiores na direita, de forma ordenada. Essa função também confere se o verbete já foi inserido, caso já exista altera a quantidade de significados deles. Analisa também verbetes de mesmo nome, porém tipos diferentes e os considera como novas palavras. Após cada chamada checa o balanceamento.
- pesquisaDicRec: Busca um verbete específico na árvore e retorna “true” caso ache.
- imprimeDicRec: Realiza o caminhamento in-order na árvore e imprime cada nó, de forma que a impressão já aconteça em ordem alfabética.

- removeDicRec: Remove todos os verbetes que tenham significado. Faz isso por meio de um caminhamento pos-order seguido de uma checagem de quantidade de significado em cada balanceamento. Ao achar um nó que deve ser deletado segue com a remoção normal de um nó em uma Árvore de Pesquisa Binária, se preocupando com a quantidade de filhos para a remoção correta. Assim como a inserção, após cada chamada checa o balanceamento.
- Antecessor: Função auxiliar para a remoção no caso de nó com dois filhos, nesse caso para a remoção acontecer de forma correta, busca o antecessor do nó, coloca ele no lugar do nó, para conseguir remover o nó sem interferir na ordenação da árvore.

➢ *HASH*

A tabela Hash é uma estrutura de dados que se assemelha a uma tabela. É um processo que mapeia chaves e valores usando uma função (hash function) para uma tabela (hash table). Ela é bastante usada pelo fácil acesso aos elementos.

Porém sua eficiência depende de qual Hash function é usada, já que dependendo da função são geradas colisões. Baseando no paradoxo do aniversário, que diz que em um grupo de 23 ou mais pessoas a chance de 2 pessoas fazerem aniversário no mesmo dia é de mais de 50%. A Hash segue o mesmo princípio, onde dois valores podem acabar apresentando a mesma chave.

Colisões é o maior problema de uma Hash, para lidar com elas existem diferentes formas, as mais usadas são: encadeamento de listas e endereçamento aberto.

Endereçamento Aberto é uma técnica que todos os valores serão armazenados diretamente na tabela e para isso o tamanho da tabela tem que ser sempre igual ou maior ao número total de chaves. Essa solução se baseia em ação um endereço para um elemento, e caso esse endereço esteja ocupado, pule para o endereço seguinte até encontrar um vazio. O problema com essa técnica é caso o próximo endereço vazio esteja muito adiante, o que leva as operações de busca a gastarem mais tempo.

Já o encadeamento de listas é uma técnica que consiste em cada endereço da hash ser uma lista encadeada. Em caso de colisões ambos os elementos são adicionados no mesmo endereço, porém em posições diferentes da lista. Apesar de ser uma das técnicas mais usadas e de fácil implementação tem como desvantagem o gasto extra de espaço, tanto de endereços não usados na hash quanto em espaço para as listas extras.

O Enderaçamento Aberto foi a solução escolhida para lidar com colisões

Para tal implementação foi criado o TAD DicHash. Esse TAD é a tabela hash como um todo, sendo que essa tabela é alocada dinamicamente com o tamanho de linhas do arquivo de entrada, para que não ocorra erros com o endereçamento. A tabela é composta de nó's, e dentro de cada nó existe um verbete que vai armazenar a palavra, juntamente com dois valores booleanos, para indicar se esse nó está vazio ou se já existiu uma palavra lá mas foi retirada. É feito essa checagem para certificar q os verbetes fiquem nas posições corretas.

Além da hash function, a Hash também apresenta as 4 operações principais que serão analisadas nas próximas seções, são elas:

- InsereHash: Insere o elemento a partir de uma hash function. Caso o local esteja ocupado, passa para o próximo até encontrar um vazio. A hash function escolhida foi a

soma dos valores asc da palavra, atribuindo peso a eles, mod o valor total de elementos da tabela.

- pesquisaHash: Retorna “true” se achar o elemento na tabela. Acha o índice a partir da hash function e busca o elemento até achar ou encontrar um local vazio, oque significa que o elemento não existe.
- removeHashComSignificado: Remove todos os elementos da tabela que tenham significado. A remoção é não literal, já que só altera o valor do nó para “retirado”.
- imprimeHash: Após a ordenação da tabela, realiza a impressão dos verbetes que não foram retirados.

Um problema de usar Hashing é na hora de imprimir ordenado, já que o hash não armazena os elementos com base em uma ordem, mas sim com base em uma função aritmética. Para conseguir imprimir de forma ordenada foi utilizado o uma das versões estudadas de quicksort nessa disciplina.

Dessa forma, na hora de imprimir os vetores é chamado o quicksort para o vetor, e depois que ele se encontra ordenado que se imprime os verbetes.

Com essas funções é possível realizar as operações requisitadas no trabalho.

Análise de complexidade

A análise de complexidade vai se basear principalmente no tempo e espaço gasto para as operações de pesquisa, inserção e remoção.

- ***ÁRVORE AVL***

TEMPO

- **Pesquisa**

Como princípio da análise é importante compreender que a altura máxima de uma árvore balanceada com n elementos é de log n. Logo qualquer operação de busca tem um custo máximo de O(log(n))

O melhor caso é O (1), caso o elemento procurado seja a raiz da arvore e o caso médio também é de O (log n), assim como o pior caso. Logo:

Médio = Pior: O (log(n))

- **Inserção**

Para a inserção precisamos de uma série de ações. Primeiramente encontramos o local do nó, o que tem um custo médio de O (log n). Em seguida é preciso atualizar as alturas e balancear caso necessário. A atualização de alturas também tem um custo de O (log n), já que é preciso de atualizar a altura do pai do nó até a raiz. O balanceamento tem um custo constante, O (1), é uma operação que consiste somente na troca de ponteiros.

Logo, o custo total é a soma das rotações(balanceamento), atualização de altura e busca do nó:

O (1) + O (log(n)) + O (log(n))

Médio: O (log(n))

- **Remoção**

Assim como a inserção, a remoção também consiste em uma sequência de passos. Achar o antecessor e o nó: O (log(n)); realizar a retirada: O (log(n)) + O (1) (para troca de nó); operação de balanceamento, verificar balanceamento e corrigir: O (log(n)). Logo também apresenta um custo total de O (log(n)):

*O (log(n)) + O (log(n) * O (1))*

Médio: O (log(n))

Apesar da função apresentar esse custo, no programa, foi requisitado a remoção de todos os verbetes que tenham significado. Para isso foi feito o caminhamento que tem custo O (n) seguido da remoção padrão. Isso nos leva à uma complexidade maior, para o programa.

O(n)

- **Impressão Ordenada**

Por ser uma árvore ordenada, ao realizar o caminhamento in-order, já imprimimos a árvore de forma ordenada. Esse caminhamento passa por todos os nós, e tem sua complexidade de:

Médio: O (n)

ESPAÇO

A árvore AVL não ocupa espaço extra além dos próprios nós, a sua complexidade espacial é sempre igual, independentemente de ser no pior, melhor e caso médio:

O (n)

- ***HASH***

TEMPO

- **Pesquisa**

O maior inimigo da Hash table são as colisões, caso a função consiga lidar bem com as colisões, ela pode chegar a um custo total constante. Porém, caso aconteçam muitas colisões pode acontecer o agrupamento, quando muitas posições da tabela estão ocupadas e a tabela fica de forma contígua, levando uma piora no tempo das operações.

No melhor caso e caso médio a pesquisa tem custo constante, no pior caso o custo é de O (n).

Pior: O(n)

Médio = Melhor: O (1)

- **Inserção**

Apresenta os mesmos custos da pesquisa:

Pior: O(n)

Médio = Melhor: O (1)

- **Remoção**

Apresenta os mesmos custos da pesquisa:

Pior: O(n)

Médio = Melhor: O (1)

Apesar de esse ser o custo da função, é importante mencionar que o programa requer a retirada de todos os elementos que tenham significado. Isso é feito por meio de caminhar pela tabela inteira e retirar elementos que tenham significado. Assim como na implementação da AVL, no programa a função de remover vai apresentar um custo acima, aproximadamente O(n) para a remoção de valores.

O(n)

- **Impressão Ordenada**

Uma das desvantagens no uso da Hash é a impressão de dados em alguma ordem. No caso do problema desse trabalho, para conseguir realizar a impressão ordenada, foi feito a ordenação da tabela inteira usando quicksort. Essa operação tem custo de O (n log n) no caso médio. Após a ordenação, é feito o caminhamento pela tabela imprimindo os valores, esse caminhamento tem custo O (n), já que anda por todos os elementos da tabela. Logo, o custo total da impressão é o custo do algoritmo de ordenação usado:

O (n log(n)) + O (n)

Médio: O (n log(n))

ESPAÇO

Assim como a árvore, não exige espaço extra além do tamanho da própria tabela, que é do tamanho da sua entrada.

O (n)

Estratégias de Robustez

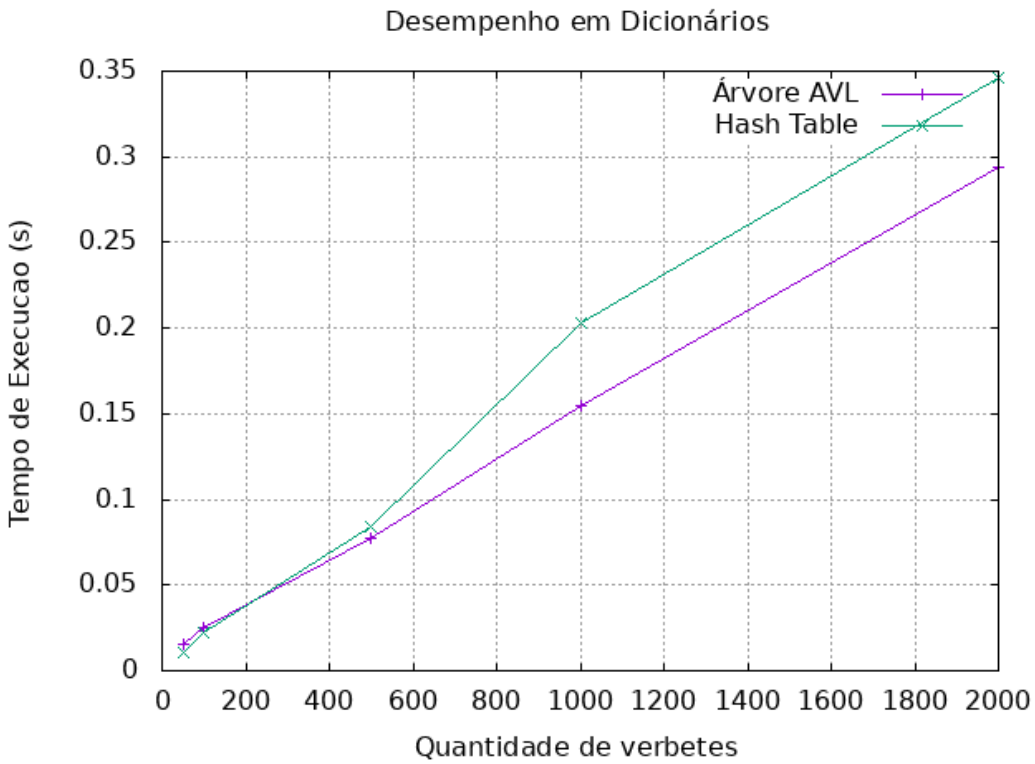
Para a estratégia de robustez foi utilizado a biblioteca “msgassert.h”, para certificar que arquivos foram abertos e que os dados passados são os corretos. Como por exemplo, certificar que o arquivo não seja vazio.

Análise experimental

Esta é a continuação da seção 3, onde contém os resultados dos testes feitos que levaram as conclusões finais do trabalho e também da análise de complexidade.

ANALISE DE DESEMPENHO

O gráfico analisa o desempenho para um ciclo completo do programa, isto é, adicionar verbetes, imprimir dicionário em ordem, remover verbetes com significado, imprimir novamente o dicionário em ordem, com diferentes quantidades de verbetes gerados aleatoriamente (usando a função rand()):



Por meio do gráfico conseguimos notar que para valores pequenos, usar a implementação de Hash é mais eficiente, porém para valores grandes de quantidade de verbetes a implementação vai perdendo desempenho. Isso se deve pela ordenação realizada antes das impressões.

- ***ÁRVORE AVL***

Registro de Memória

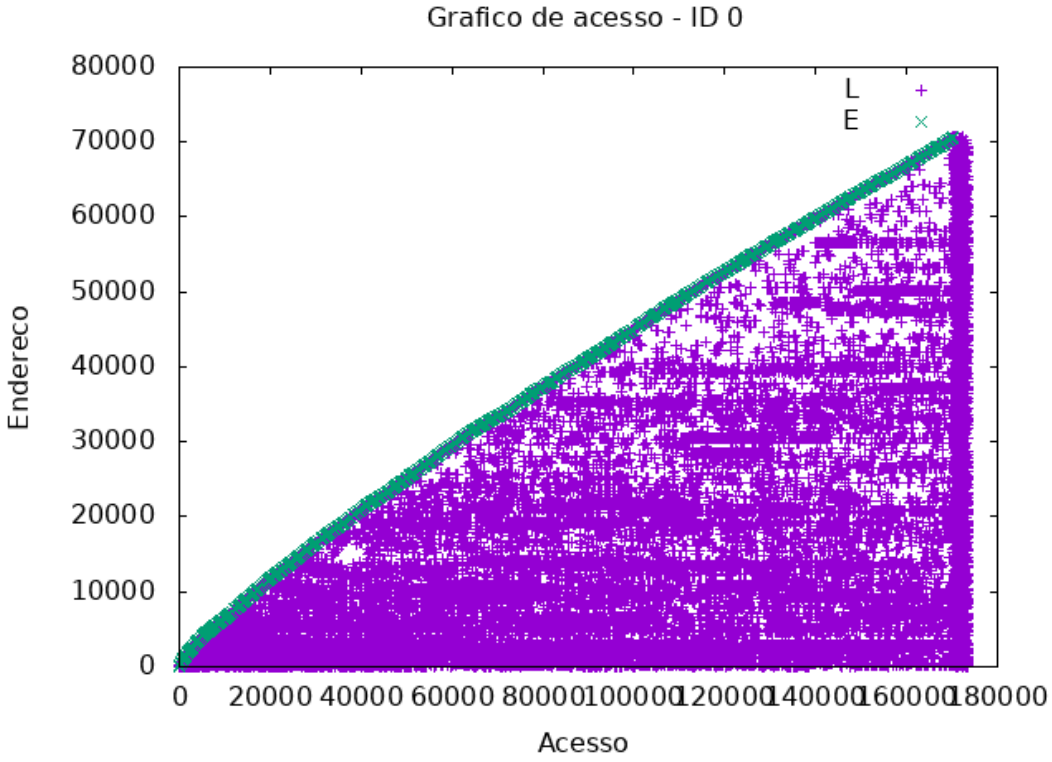
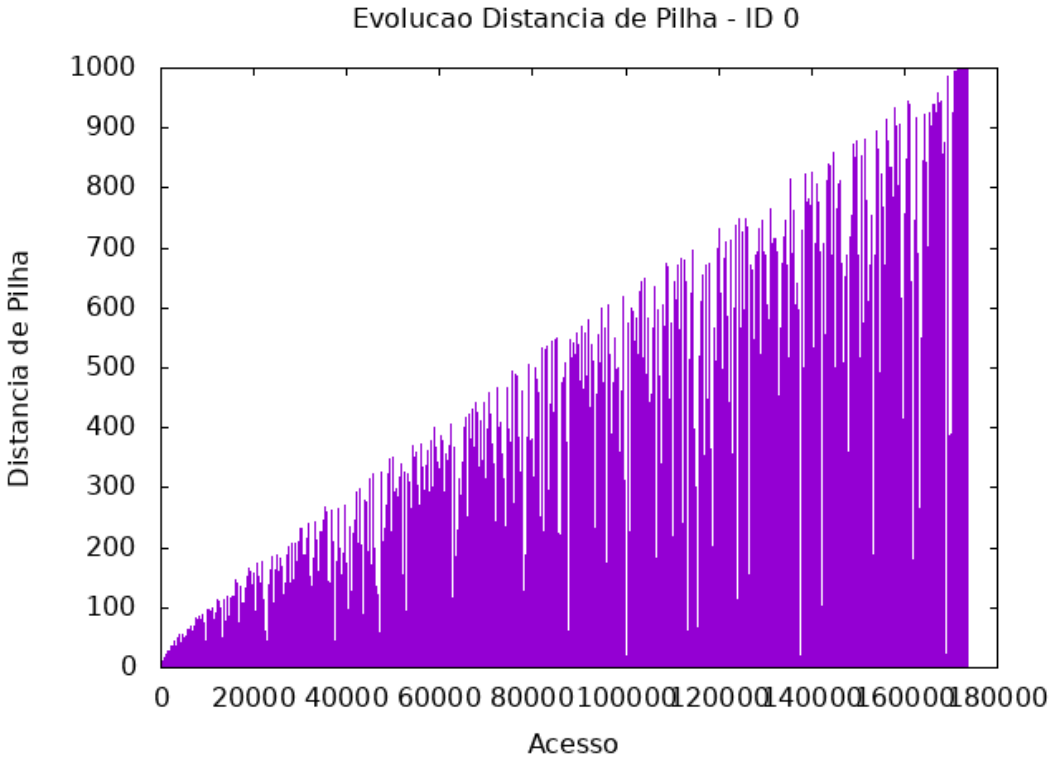
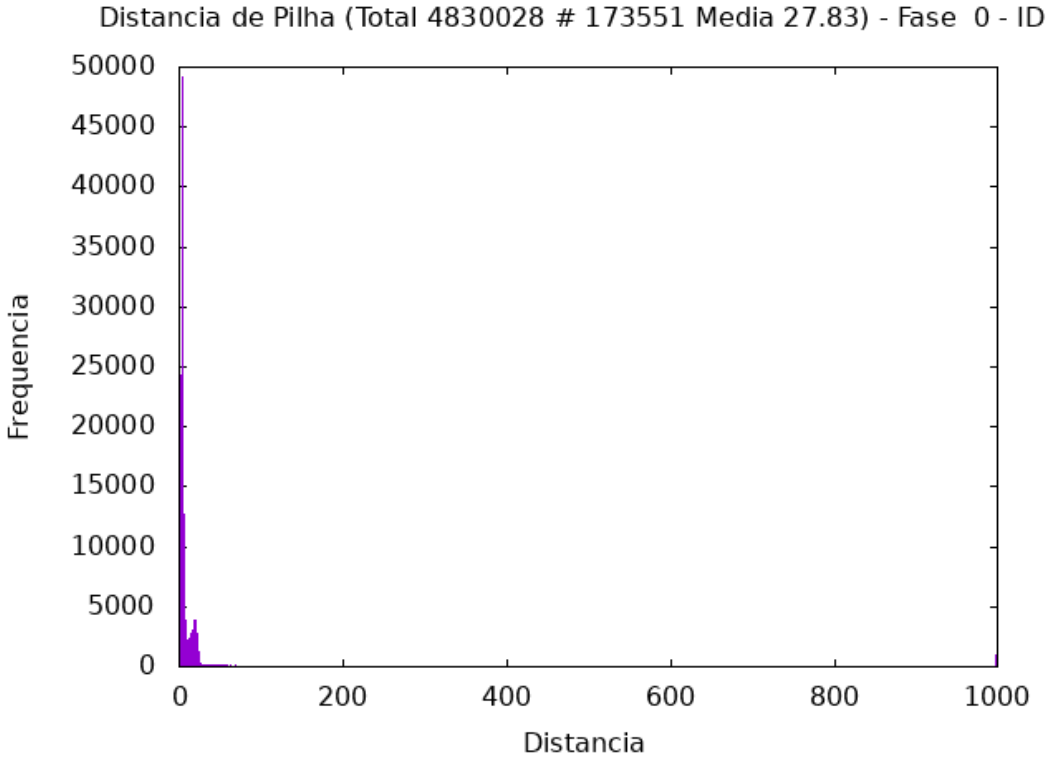


Grafico de acesso - ID 0
Endereco
Acesso
L
E
0
10000
20000
30000
40000
50000
60000
70000
80000
0
20000
40000
60000
80000
100000
120000
140000
160000
180000


Evolucao Distancia de Pilha - ID 0
Distancia de Pilha
Acesso
0
100
200
300
400
500
600
700
800
900
1000
0
20000
40000
60000
80000
100000
120000
140000
160000
180000



- ***HASH***

Registro de Memória

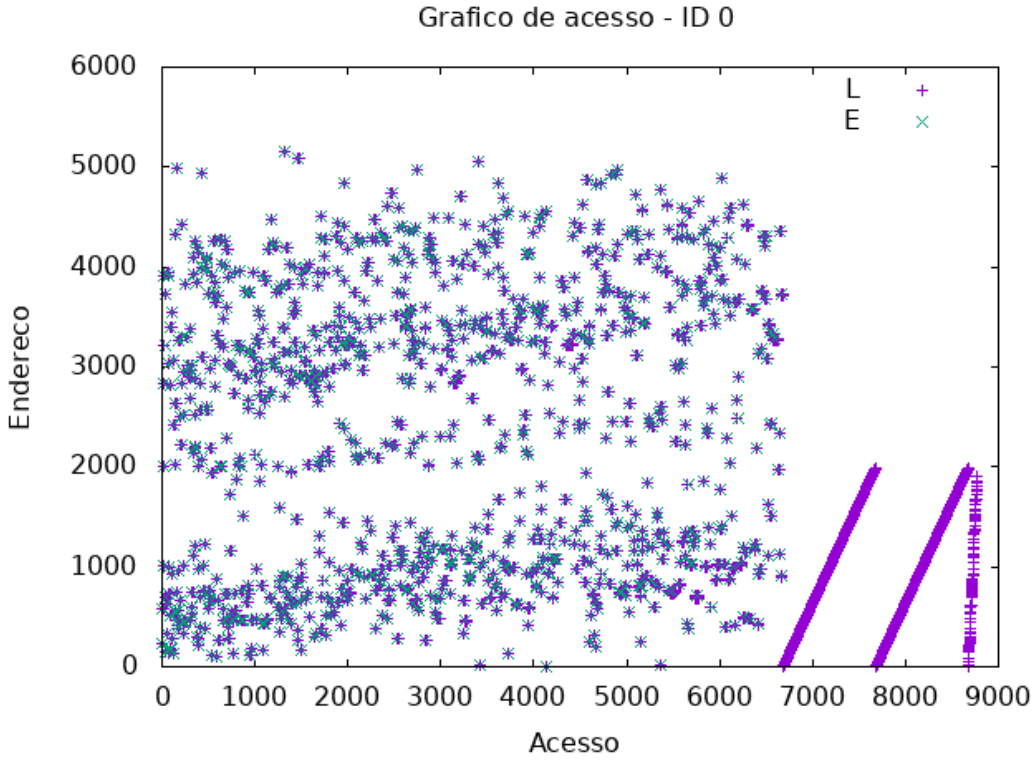
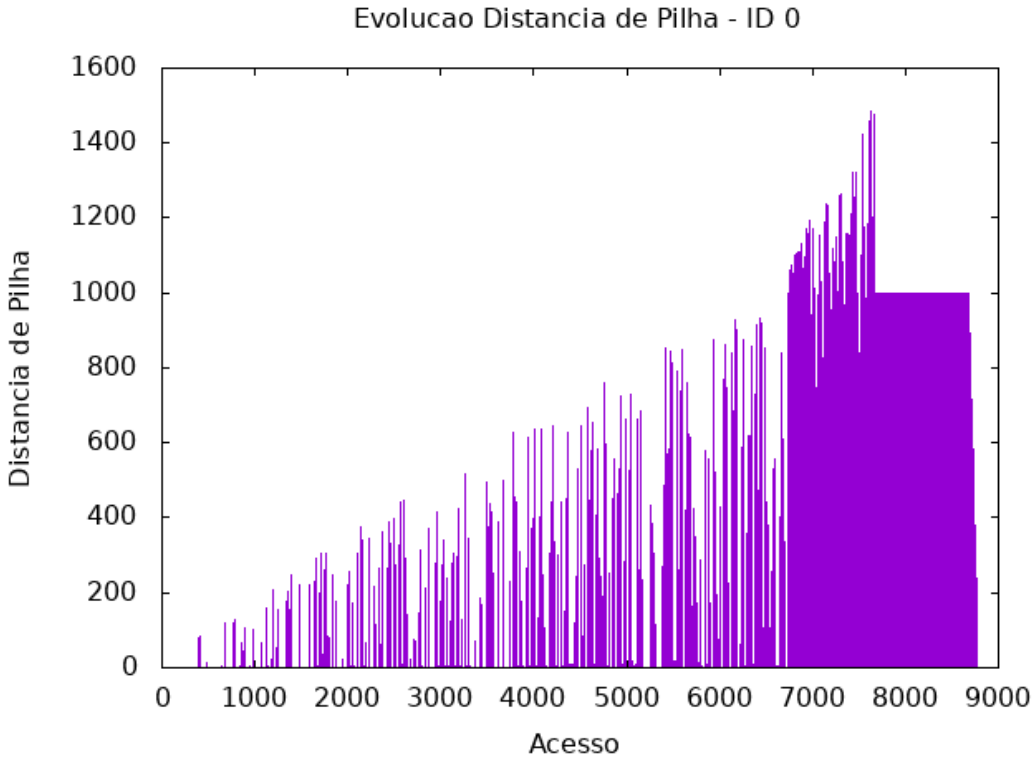
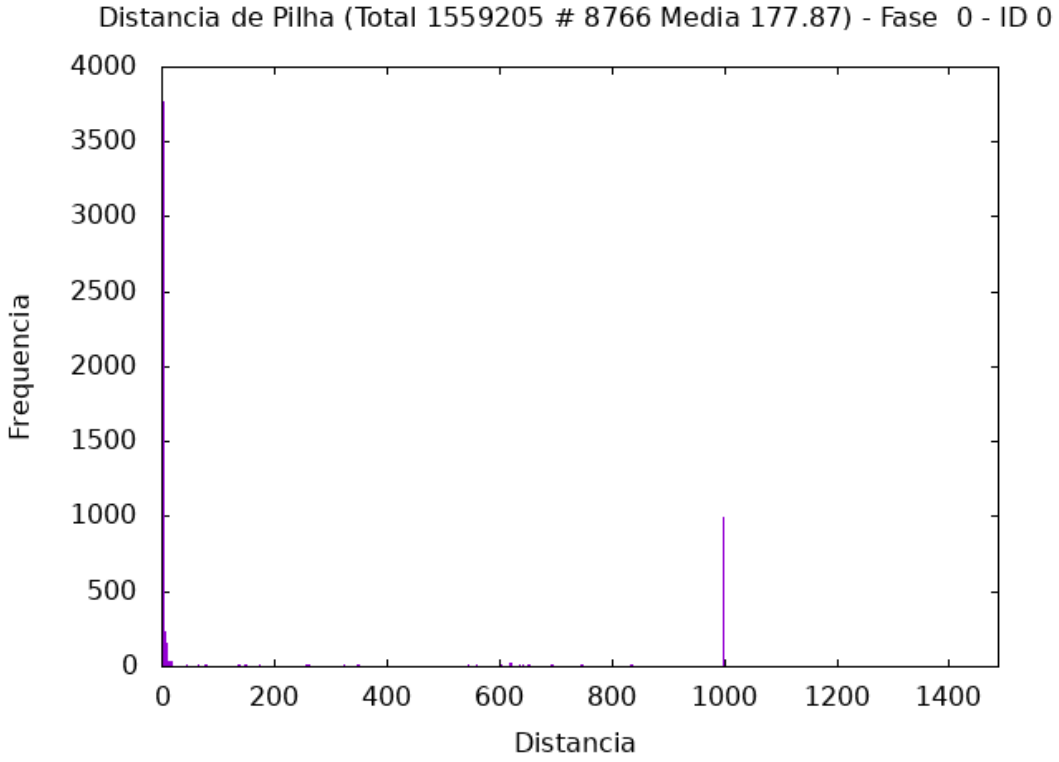


Grafico de acesso - ID 0
Endereco
Acesso
L
E
0
1000
2000
3000
4000
5000
6000
7000
8000
9000


Evolucao Distancia de Pilha - ID 0
Distancia de Pilha
Acesso
0
200
400
600
800
1000
1200
1400
1600
1000
2000
3000
4000
5000
6000
7000
8000
9000



Esses gráficos representam o registro de acesso de memória dos verbetes em cada implementação do dicionário, não levando em conta os registros de memória na hora de ordenação para impressão dos verbetes.

Vantagens e Desvantagens

Segue a tabela com as vantagens e desvantagens de cada implementação de forma geral:

Diferenças entre implementações		
Diferenças	AVL	Hash(enderaçamento aberto)
dificuldade de implementação	implementação complexa	implementação simples
custo médio operações padrões	O(log(n))	O(1)
pior caso operações padrões	O(log(n))	O(n)
custo ordenação	O(log(n))	O(n log(n))
achar maior/menor elemento	assim como ordenação, é natural. Dados armazenados de forma ordenada	requer esforço extra
saber o tamanho do input	não precisamos saber	precisamos de saber de inicio
colisões	não acontecem	precisamos lidar com elas
acessos a leitura de memoria	muito	médio

Conclusões

Em suma, ambos TAD's são ideais para diferentes problemas. Caso a ordem não seja um fator relevante a Hash Table é a melhor opção, por conta do seu código simplificado e de sua

complexidade constante, no melhor caso e caso médio, para a maioria das operações. Na hora de uma impressão ordenada depende de um algoritmo de ordenação, e estes apresentam, no geral, um custo minimo de O (n log(n)), o que faz com que a implementação perca desempenho.

No caso do dicionário requisitado, que requer a impressão ordenada, a melhor opção é a Árvore Balanceada. Apesar de perder desempenho em operações de inserção/remoção, essa diferença se compensa na impressão em ordem que requer somente um caminhamento específico.

Instruções para compilação e execução:

Acesse o diretório principal, onde se encontra o Makefile;

Compilar: make all;

Executar:

```
./tp3 -i entrada.txt -o saida1.txt -t hash
./tp3 -i entrada.txt -o saida2.txt -t arv
```

Flags:

```
-i : arquivo do input
-o : arquivo do outpu
-t : versão do dicionario (arv ou hash)
```

O “tp3” fica no diretório Bin.