Trabalho Prático 3

Filipe de Araújo Mendes

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

flipeara@ufmg.br

1. Introdução

O trabalho prático 3 consiste na implementação de um dicionário de palavras com o uso de uma árvore binária balanceada e um hash, tendo então dois tipos diferentes de dicionários.

Então, será feita uma análise das duas implementações levando em consideração o tempo de execução do programa, mostrando em quais casos o dicionário com a árvore é melhor e em quais casos é preferível utilizar o hash, assim como diferentes configurações para o mesmo.

2. Método

2.1. Main.cc

Na hora de iniciar o programa, são passados alguns argumentos, sendo responsabilidade da função main registrá-los e executá-los.

Além disso, a função main fica encarregada de criar o dicionário do tipo pedido, ler o arquivo de entrada e mandar inserir os verbetes, remover e imprimir, assim como calcular o tempo levado para a execução do programa e então destruir o dicionário criado.

2.2. MeaningList (Lista de Significados):

Levando em consideração que um Verbete precisaria poder ter mais de um Significado, foi decidido que uma lista encadeada seria uma boa solução para guardar vários deles, já que é uma estrutura com custo baixo de inserção de novos nós e, como na hora da impressão os Significados são sempre impressos sequencialmente, a lista encadeada funciona muito bem para o método de impressão.

A classe MeaningList possui uma variável do tipo inteiro que guarda o tamanho da lista e ponteiros para o primeiro Significado e para o último, que é utilizado para que a inserção seja feita com custo constante para maior eficiência.

Ela possui um construtor, que inicia os ponteiros como nulos e o tamanho como zero, um método que retorna se a lista está vazia e métodos para a inserção de novos significados e impressão dos significados já contidos.

2.3. Meaning (Significado):

O Significado é um nó da Lista de Significados, guardando uma variável do tipo string valor, que guarda, de fato, um dos significados do Verbete, e um ponteiro para o próximo Significado, que caracteriza o encadeamento da lista.

Seu construtor é simples, criando o novo Significado com o valor passado por parâmetro, inicialmente com o ponteiro para o próximo Significado nulo. Por último, é uma classe amiga da Lista de Significados, para que esta possa acessar os atributos privados dos Significados.

2.4. Entry (Verbete)

O Verbete possui uma variável do tipo string que guarda a palavra, ou valor, dele, uma variável do tipo char que guarda o tipo, se é substantivo, verbo… e uma Lista de Significados, uma vez que ele pode possuir mais de um. É importante notar que, por utilizar uma lista encadeada para os Significados, em teoria, cada Verbete pode ter uma quantidade ilimitada de Significados.

Em criação, é passado como parâmetro a palavra, ou valor, do Verbete, assim como seu tipo, que são utilizados pelo construtor para que sejam os valores iniciais desse novo Verbete, que também possui uma nova Lista de Significados inicialmente vazia. Existem métodos para incluir novos significados no Verbete, imprimi-lo ou verificar se ele já possui algum Significado. Há também os métodos getters, que retornam ou a palavra contida no Verbete ou seu tipo, necessários por possuir atributos privados, inacessíveis por fora do Verbete.

2.5. AVLDict (Dicionário AVL)

Uma árvore binária completamente balanceada seria uma implementação muito custosa e não muito eficiente, pelo alto custo de realizar balanceamentos frequentemente, assim, foi escolhida a árvore AVL, uma árvore semi-balanceada, para a implementação do dicionário em árvore.

A estrutura em si, só guarda um ponteiro para a raiz da árvore, mas possui diversos métodos. Para a inserção, a árvore inicialmente faz uma busca binária pela palavra, que pode cair em alguns casos:

1. Se a palavra existir e o tipo for o mesmo, o Verbete é atualizado, acrescentado um novo Significado a ele, se existir um novo Significado a ser adicionado.
2. Se a palavra existir e o tipo for diferente, é acrescentado um novo Verbete na ordem correta.
3. Se a palavra não existir na árvore inteira, é adicionado um novo Verbete na ordem correta.

Após cada inserção, o balanceamento é verificado e se estiver desbalanceada, a árvore é balanceada novamente. Para a remoção, como nas especificações da sequência de operações do programa, a árvore procura e remove todos os Verbetes com pelo menos um significado, para depois ser balanceada.

A árvore possui um método de impressão, que imprime todos os Verbetes contidos nela, em ordem ASCII, que é a forma que foi ordenada. Isso significa que qualquer palavra que comece com uma letra maiúscula virá antes de palavras que começam com letra minúscula, mas entre palavras que começam com o mesmo tipo, segue a ordem alfabética.

Por último, existem também vários métodos auxiliares, que são utilizados para evitar a repetição do mesmo código em vários lugares diferentes, que realizam a mesma tarefa.

2.6. AVLNode (Nó AVL)

Como a classe Verbete teve que ser criada pensando tanto na implementação em Hash quanto na implementação em Árvore, foi necessária a criação de uma classe Nó AVL, para que fosse possível a árvore AVL conter os Verbetes. Esse nó possui um ponteiro para um Verbete, uma variável do tipo inteiro que guarda a altura do nó na árvore, para o balanceamento mais fácil da estrutura, e 3 ponteiros para outros Nós, um para o filho da esquerda e um para o filho da direita, que caracterizam a árvore binária, e, então, um ponteiro para o pai, a fim de facilitar o balanceamento.

O construtor do nó pega um Verbete passado como parâmetro e já inicializa o nó com esse Verbete, mas com todos os outros ponteiros como nulos, além de inicializar a altura como 1, já que toda inserção em uma árvore binária é feita como folha da árvore. Por fim, ela possui o Dicionário AVL como classe amiga, para que essa possa manipular seus atributos privados.

2.7. HashDict (Dicionário Hash)

Pelas especificações do TP e pela necessidade de ordenar as palavras do dicionário foi escolhida a lista encadeada para resolver o

problema das colisões do Dicionário Hash. Assim, o Dicionário Hash possui uma variável do tipo inteiro M e uma tabela com M linhas, sendo que M é igual a 25 + 26^k, onde k é o número de letras que serão levadas em consideração na função hash, isso será explicado em mais detalhes posteriormente.

Na hora da inserção, o Dicionário Hash faz uma pesquisa pela palavra, caindo nos mesmos casos do Dicionário casos:

1. Se a palavra existir e o tipo for o mesmo, o Verbete é atualizado, acrescentado um novo Significado a ele, se existir um novo Significado a ser adicionado.
2. Se a palavra existir e o tipo for diferente, é acrescentado um novo Verbete na ordem correta.
3. Se a palavra não existir na árvore inteira, é adicionado um novo Verbete na ordem correta.

Caso ocorra alguma colisão com a função hash, o Verbete é então ordenado na lista encadeada correspondente no momento de inserção. Para a remoção, como no Dicionário AVL, o Dicionário Hash procura em todas as listas encadeadas e remove todos os Verbetes com pelo menos um significado. A impressão também é análoga à do Dicionário AVL.

A função hash, por sua vez, utiliza do código ASCII das letras para escolher a linha correta da tabela. Para igualar à ordenação do Dicionário AVL, se a primeira letra da palavra for maiúscula, o hash redireciona à uma das 26 primeiras linhas, de A a Z, caso seja minúscula, a transformação ocorre de maneira diferente, começando na linha 26 da tabela, e dependerá do valor de k na fórmula M = 26 + 26^k apresentada anteriormente.

Para explicar melhor, daremos valores a k. Se k for 2, significa que a tabela terá 26 + 26^2 = 702 linhas. Assim, palavras começadas em “aa” serão colocadas na linha 26 e palavras começadas com “zz” serão colocadas na linha 701, com o resto das combinações distribuídas nas linhas. Se k for 4, a tabela terá 457.002 linhas, sendo a linha 26 para todas as palavras começadas com “aaaa” e a linha 457.001 para todas as palavras começadas com “zzzz”.

Com k = 4, para encontrar a linha da tabela em que uma palavra se encontra, é pego a posição da primeira letra da palavra no alfabeto (de 0 a 25) e multiplicado por 26 elevado ao cubo, para a segunda letra, multiplica-se a posição dela no alfabeto por 26 elevado ao quadrado, por assim vai. Assim, palavras começando com “inte”, por exemplo, seriam todas enviadas para a tabela de número:

$$26 + 26^3 * 8 + 26^2 * 13 + 26^1 * 19 + 26^0 * 4 = 149920$$

Sendo que 8 é a posição da letra “i”, 13 é a posição da letra “n”, 19 é a posição da letra “t” e 4 é a posição da letra “e”.

Ao perceber que, com k = 4, a tabela possui uma linha para cada combinação entre “aaaa” e “zzzz”, é fácil notar que esta não é a função hash mais eficiente, já que existem muitas linhas que sequer terão pelo menos um Verbete. Entretanto, uma função desse tipo é simples de ser implementada e já produz bons resultados na divisão, reduzindo bastante o número de colisões e deixando as listas de Verbetes relativamente pequenas.

2.8. HashList (Lista Hash)

Como foi escolhido o método das listas encadeadas para a resolução das colisões no Dicionário Hash, foi necessário criar, de fato, a lista que seria utilizada pelo hash. Essa lista guarda, assim como a lista anterior, um tamanho, do tipo inteiro, e um ponteiro para o primeiro e o último nó e sua criação é idêntica à anterior, inicializando o tamanho como zero e os ponteiros como nulos.

Na inserção, como mencionado anteriormente, a lista compara os Verbetes de modo a sempre manter a ordem alfabética. Para remover, percorre procurando os Verbetes com significado e os remove também mantendo a ordem. Na impressão, somente percorre cada Verbete, imprimindo do início ao fim da lista.

2.9. HashNode (Nó Hash):

Assim como o Nó AVL, o Nó Hash foi criado para guardar o Verbete, possibilitando o encadeamento entre eles. Por isso, o Nó possui um ponteiro para um Verbete e um ponteiro para o próximo Nó. Em sua criação, é passado o Verbete correspondente por parâmetro, que é colocado como valor do Nó, e o ponteiro para o próximo é nulo.

Por fim, tem a Lista Hash como classe amiga para que esta possa manipular os atributos dos Nós contidos nela.

3. Análise de Complexidade

3.1. Espaço

A Lista de Significados ocupa um espaço proporcional à quantidade de significados contidos nela, dessa forma, possui ordem de complexidade O(n).

A Árvore AVL possui uma quantidade de Nós igual à quantidade de de Verbetes contidos nela e o custo de espaço é proporcional à quantidade de Nós. Dessa forma, o custo é, assim como na Lista, O(n).

O Hash, assim como a Árvore, armazena cada Verbete em um Nó, armazenando então a mesma quantidade de Verbetes e Nós e tendo o mesmo custo dos outros dois, O(n). Entretanto, o Hash possui um custo

constante bem maior que o de qualquer um dos dois, por possuir milhares de Listas Hash, as quais ocupam 24 bytes cada. Por isso, apesar da ordem de complexidade ser a mesma, o Hash é mais custoso em termos de espaço.

3.2. Tempo

O caminhamento pela Árvore AVL, por ser uma árvore binária, é feito comparando os elementos e indo ou para a direita ou para a esquerda. Dessa forma, a cada passo, o tamanho é dividido pela metade, fazendo com que, em pior caso, leve log n passos para realizar operações como inserção ou busca. Assim, para essas operações, o custo é O(log n). Para a operação específica de remoção, a Árvore é percorrida por completo para encontrar Verbetes com significados, portanto, a ordem é O(n). Por fim, para o balanceamento, se balanceado na inserção, só é necessário conferir o galho onde se encontra o elemento inserido, com altura máxima log n, portanto, no balanceamento na inserção, o custo é O(log n). Se balanceado após as remoções, a árvore inteira é percorrida para checar o balanceamento, por isso, o custo é O(n).

Já o Hash, possui custo diretamente proporcional ao tamanho das listas contidas nele, ou seja, diretamente proporcional ao número de verbetes. Por isso, sua ordem de complexidade seria O(n), mas é preciso lembrar que as listas não possuem n elementos, e sim, em funções hash com boa distribuição, aproximadamente n / m, onde m é a quantidade de listas na tabela. Dessa forma, o custo de operações de inserção e pesquisa em um Hash bem distribuído é O(1 + n / m), que é o custo da função hash, que encontra a linha correta da tabela mais o custo para a operação na lista. Apesar de o Hash implementado não ser o mais eficiente possível, o número de colisões é baixo o suficiente para que o custo computacional não seja muito afetado.

4. Estratégias de Robustez

Para evitar a interrupção desnecessária do programa, foi adicionada uma função para checar se os argumentos contendo o nome de entrada do programa contém a extensão do arquivo requerida pelo programa, não interrompendo a execução se esse não for o caso.

Outra forma foi a implementação de valores padrão para os arquivos de entrada e saída e um tipo de dicionário padrão para a operação do programa, assim como a adaptação para que o argumento tipo possa ser passado com letra maiúscula ou minúscula, sendo aceita das duas maneiras.

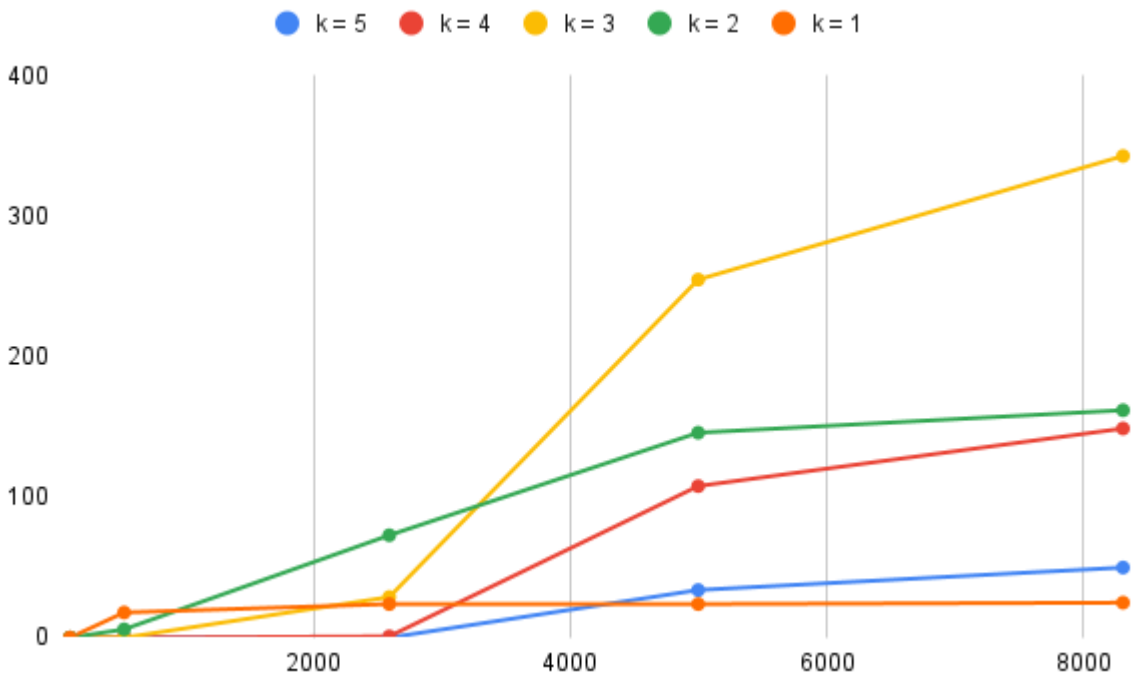
5. Análise Experimental

5.1. Hash

Inicialmente, foi feita uma análise do Hash para descobrir, para os arquivos de entrada disponibilizados, qual seria o melhor valor de k, tendo o melhor balanceamento entre distribuição das palavras e custo inicial do Hash, que produziria o menor tempo de execução do programa. Foi testado com k valendo de 1 a 5 as seguintes métricas: maior tamanho de lista na tabela, quantas listas tinham pelo menos tamanho 5, 7, 9, 12 e o tempo de execução de cada um deles.

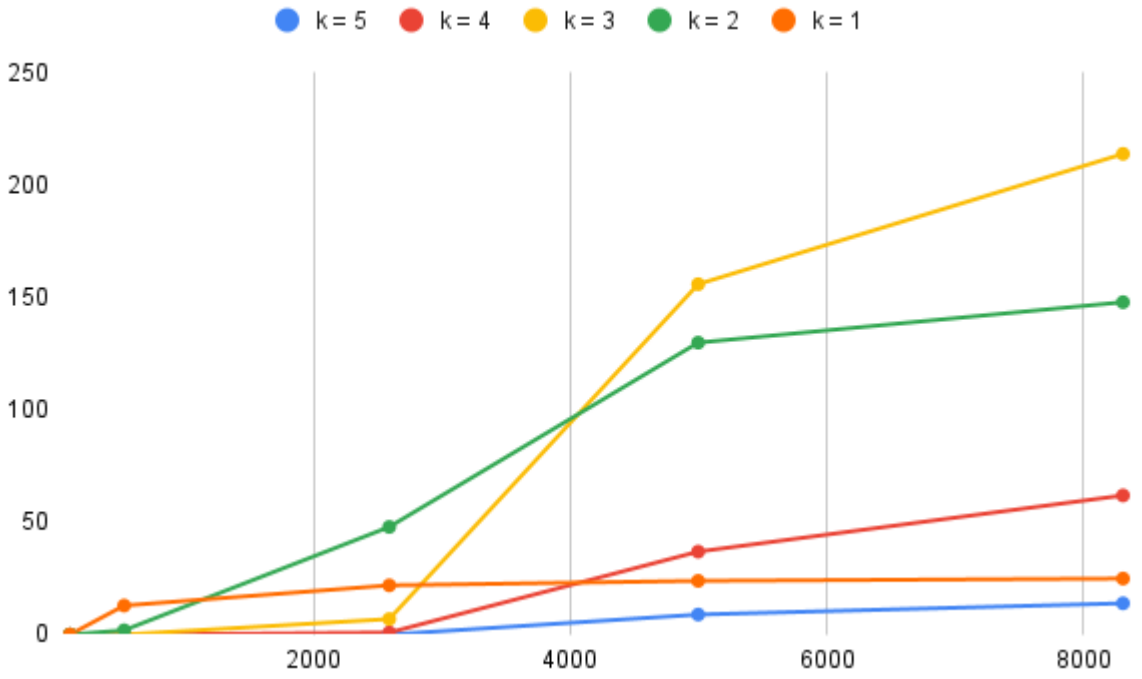
É importante ter em mente a quantidade de listas existentes no Hash com cada valor de k.

- K = 5 - 11.881.402 Listas.
- K = 4 - 457.002 Listas.
- K = 3 - 17.602 Listas.
- K = 2 - 702 Listas.
- K = 1 - 52 Listas.

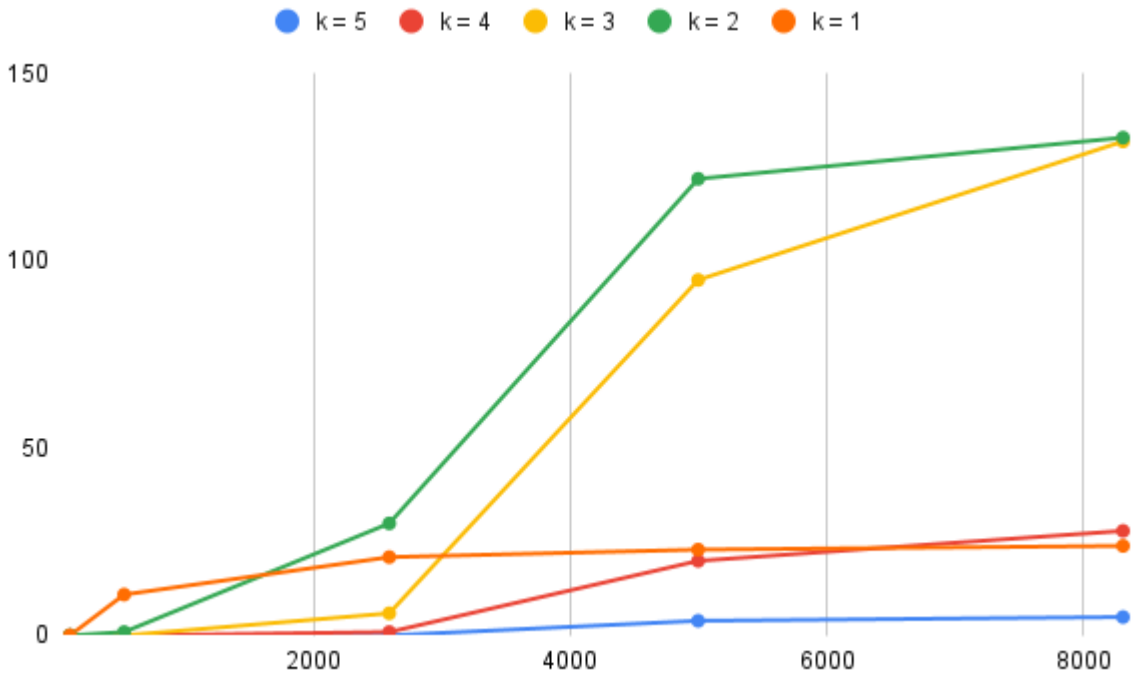


Esse gráfico mostra quantas listas tinham tamanho maior ou igual a 5 com os diferentes valores de k. Com k = 1, é possível ver que com um pequeno tamanho de entrada, quase todas as listas de letras minúsculas (26) já possuem tamanho maior que 5, por isso o valor não aumenta mais com o tamanho da entrada. Observe que acontece algo similar com k = 2, que aumenta pouco o número de listas por já ter saturado as listas de começos de palavras mais comuns, mas é necessária uma entrada de cerca de 5 mil

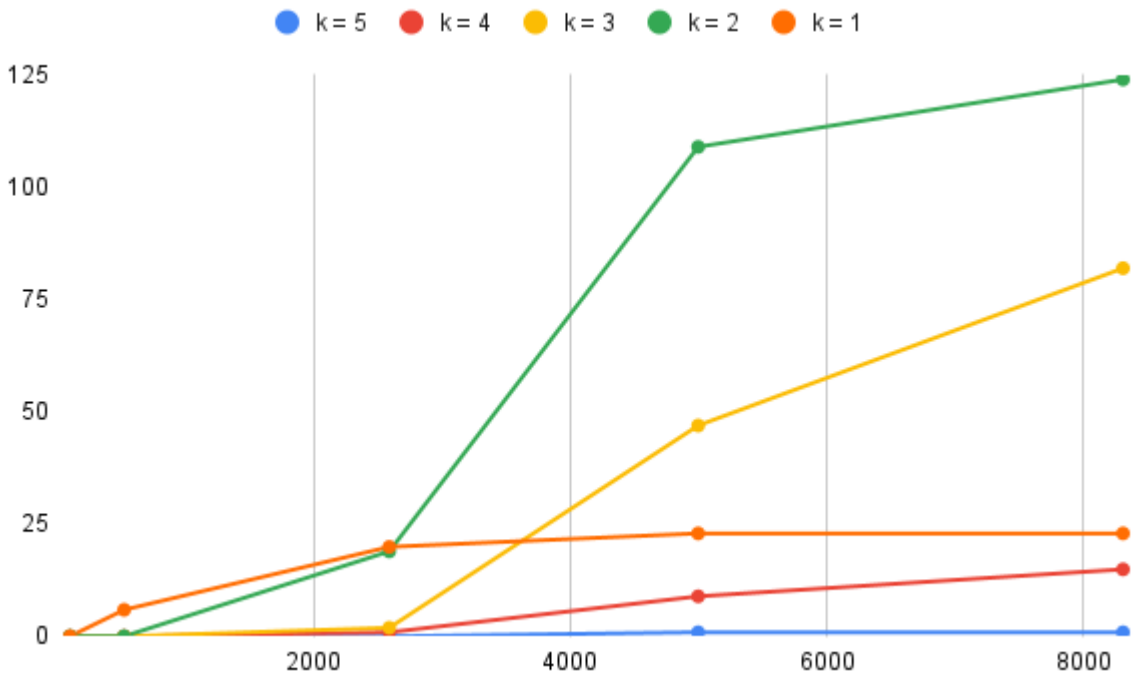
linhas para que isso aconteça. Para k = 3, na maior entrada, uma quantidade muito pequena de listas comparado ao total possuem tamanho maior ou igual a 5. Com k = 4 esse número fica abaixo da quantidade de listas em k = 2 e em k = 5, o número fica cerca do dobro da quantidade de k = 1.



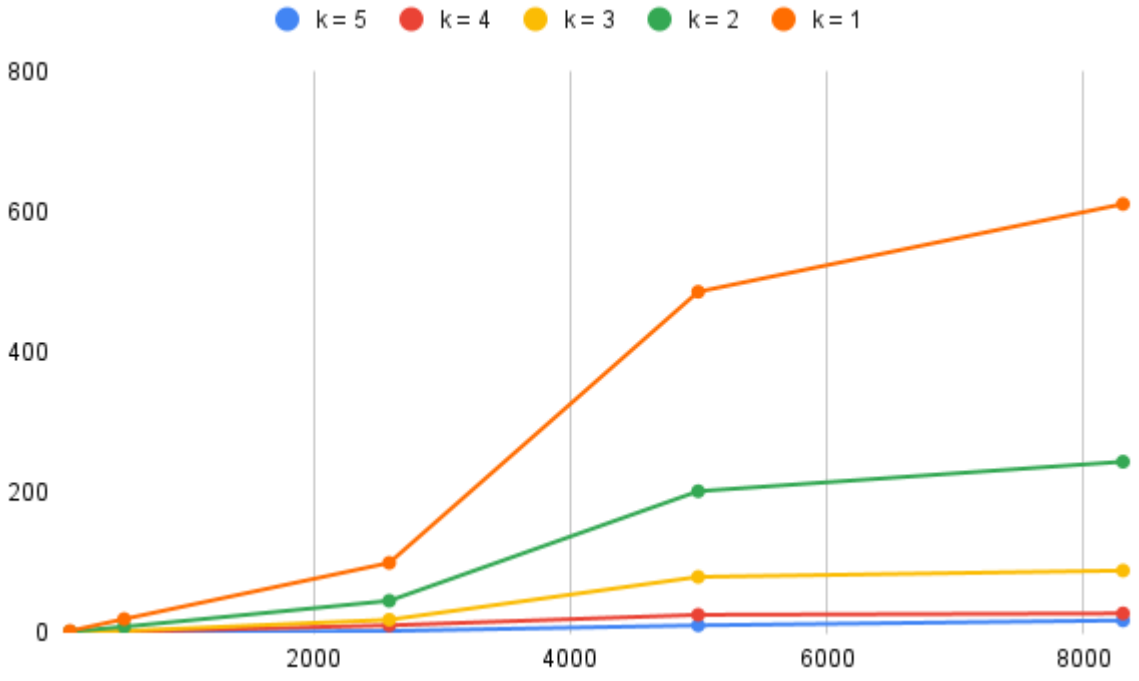
Observe que ao aumentar para tamanho maior ou igual a 7, a tendência de redução do número de listas não é seguida por k = 1 ou 2 pelo motivo da saturação explicado anteriormente. O resto dos valores de k possuem uma queda, com essa queda sendo mais expressiva para k = 4 ou 5.



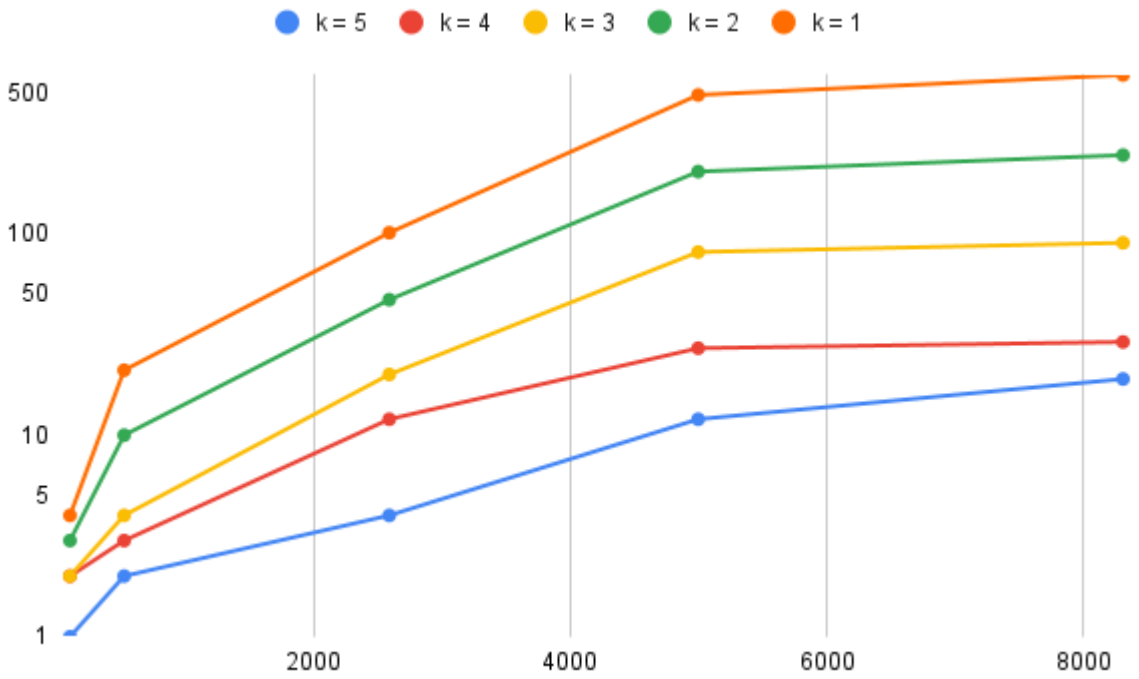
O mesmo acontece quando olhamos para tamanhos maiores ou iguais a 9. K = 1 ou 2 continuaram relativamente estáveis, enquanto os outros caíram novamente. K = 3 agora possui menos listas que k = 2 com tamanho de pelo menos 9, k = 4 está bem próximo de k = 1 e k = 5 é quase 0.



Em tamanho de pelo menos 12, k = 2 finalmente começa a cair de forma significativa, k = 1 continua estável e o resto continua caindo.



Esse último gráfico mostra o tamanho máximo das listas com cada valor de k. Como esperado, quanto maior o valor de k, menor o valor máximo de tamanho, já que há uma divisão maior. Abaixo se encontra o mesmo gráfico, mas em escala logarítmica para melhor perceber a diferença entre os tamanhos máximos de lista.



Assim, é possível perceber que com maiores valores de k, obtemos um custo cada vez mais próximo de O(1) para as operações com o Hash. Entretanto, isso não é viável por dois motivos:

- Custo constante de espaço: Com k = 4, o Hash ocupa um espaço de aproximadamente 11 MB de RAM, mas com k = 5, esse valor sobe para aproximadamente 285 MB.
- Custo constante de tempo: Quanto maior o valor de k, mais listas precisam ser criadas, adicionando ao custo constante do programa.

No gráfico abaixo, é possível perceber que, de fato, o custo das operações com k = 5 é próximo de constante e que as oscilações no tempo de execução se dão por variações normais na performance do computador, mas mesmo com isso, o tempo de execução ainda é maior do que qualquer outro, chegando a até aproximadamente 0,3 seg de execução.

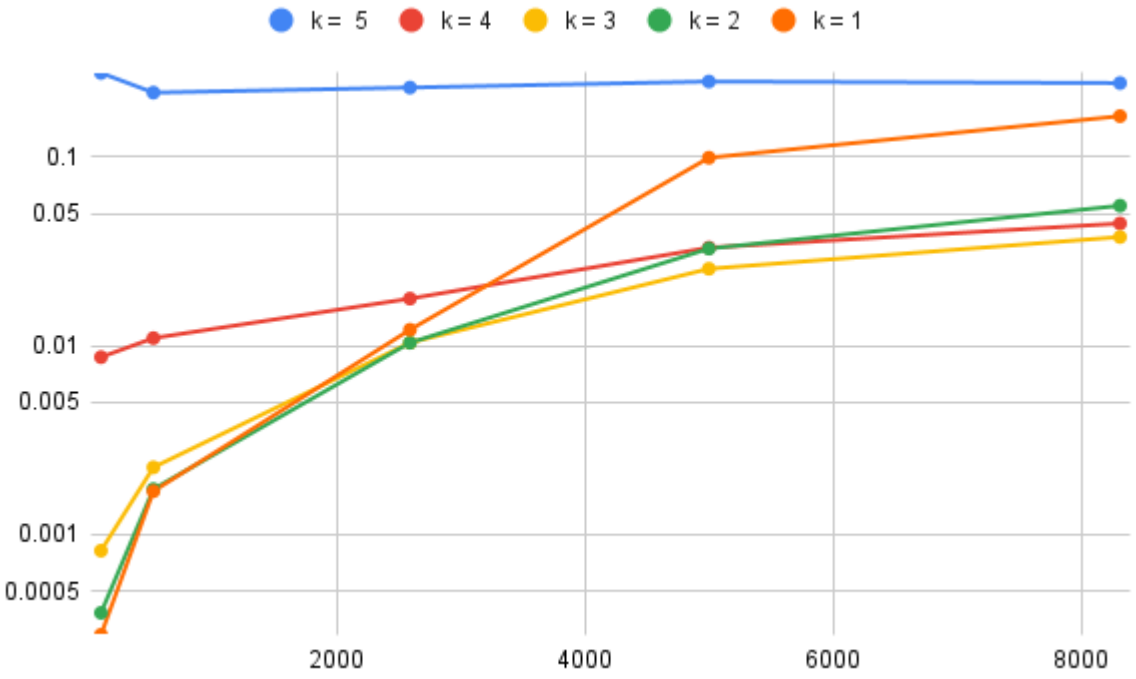
Olhando para k = 4, observa-se que o custo constante ainda está alto, com o programa levando cerca de 0.01 seg para ser executado com o menor

tamanho de entrada, enquanto com o maior, esse valor não aumenta tão significativamente.

Com k = 1, o programa tem o menor custo constante, mas como o tamanho médio das listas fica bem maior que os outros, o custo de tempo de operações nas listas se torna linear, fazendo com que na maior das entradas, o tempo de execução se assemelhe ao de k = 5.

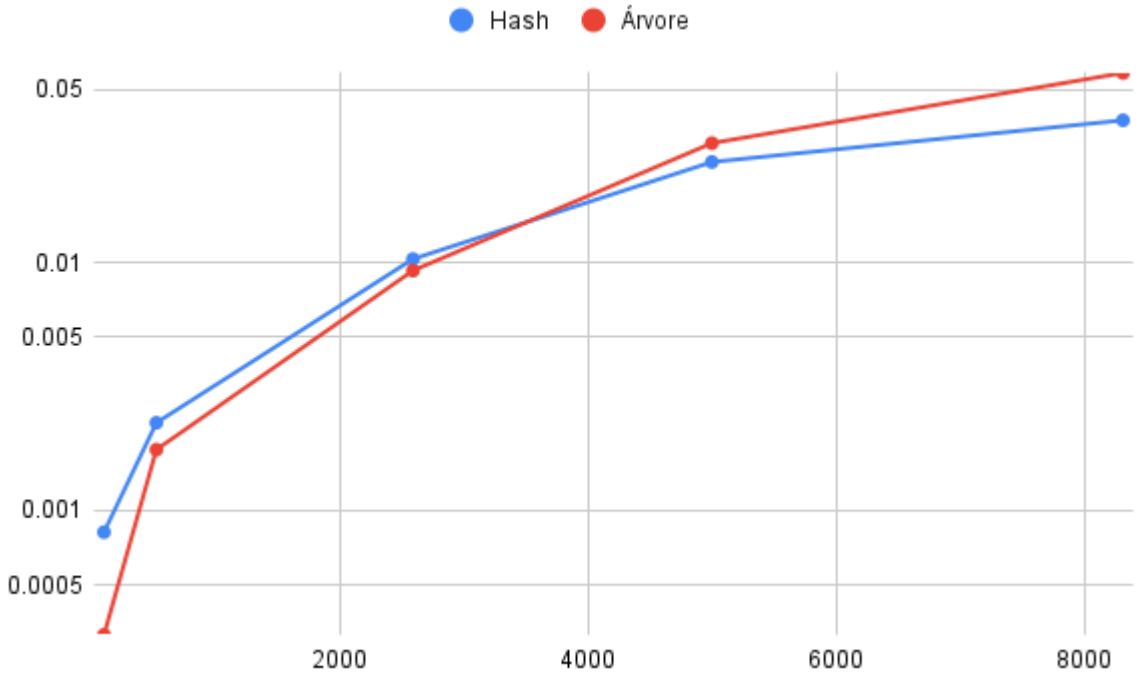
K = 2 possui um tamanho médio das listas menor que k = 1, portanto as operações são realizadas com um custo menor, mas ainda assim, em tamanhos maiores de entrada, consegue ser menos eficiente que k = 4.

O melhor valor de k para a execução nessa faixa de entrada é com certeza k = 3. Com 17.602 listas, o Hash consegue distribuir relativamente bem os Verbetes entre as várias listas, fazendo com que o custo de acesso a esses dados seja bem reduzido em comparação a outros valores de k. Além disso, em questão de custo de espaço, o custo constante do Hash com k = 3 é menos de meio MB.



5.2. Hash e Árvore

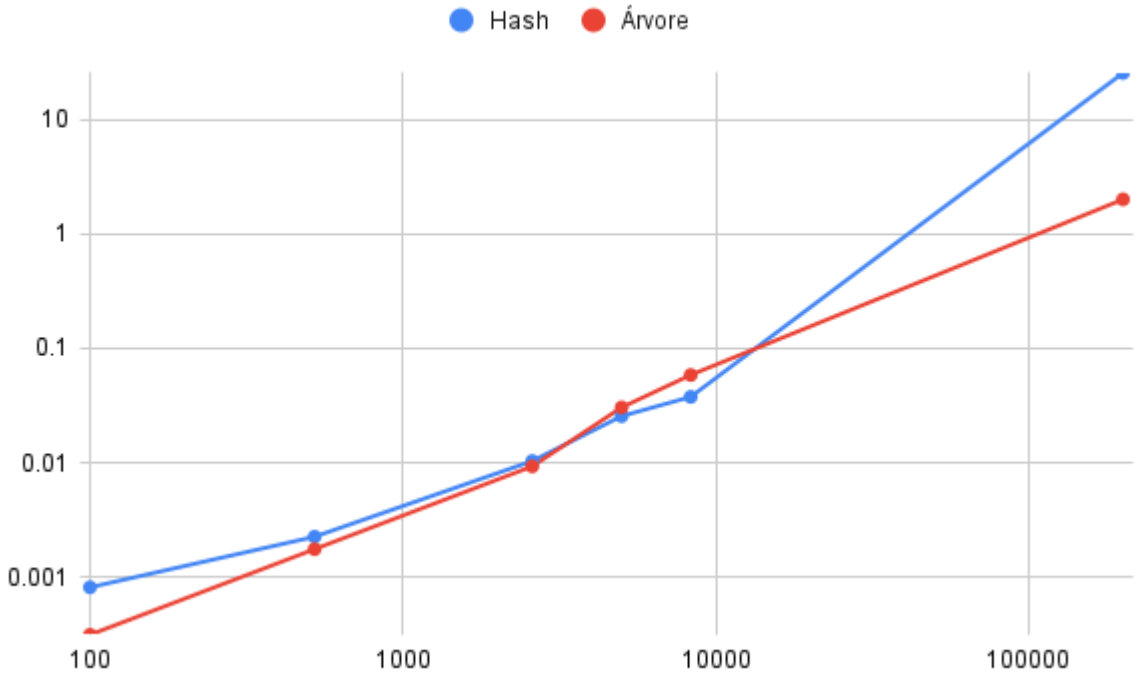
Agora, com a análise do Hash feita, é relevante comparar com o Dicionário AVL, para identificar qual dos dois é a melhor opção para esse tamanho de entrada. Nessa análise serão utilizados os mesmos arquivos de entrada e o Hash com k = 3, que foi o melhor dos 5 avaliados.



É evidente que a árvore, para menores arquivos de entrada, é mais eficiente que o Hash. Isso se dá pelo custo constante da criação das 17 mil listas no Hash, custo este que não está presente no Dicionário AVL. Entretanto, à medida que o arquivo de entrada vai crescendo, o custo inicial do Hash se torna insignificante e a altura da árvore vai aumentando, fazendo com que o custo de caminhamento na árvore, que é O(log n), e o custo de remoção, que é O(n), seja maior que o custo quase constante das operações no Hash. Assim, para arquivos acima de 4 mil linhas, o Hash se torna mais eficiente, enquanto utiliza uma quantidade ainda baixa de espaço para as listas.

Entretanto, isso não significa que o Hash é estritamente melhor que a árvore acima desse ponto, já que com o aumento das colisões, as listas vão ficando cada vez maiores e o custo de operações no Hash deixa de ser perto de constante, enquanto o custo majoritariamente O(log n) da Árvore faz com que ela seja bem mais eficiente que o Hash. Para evidenciar isso, foi feito um teste com a entrada de tamanho 200.000, como mostrado no gráfico abaixo, agora com os dois eixos em escala logarítmica.

Enquanto o Hash levou cerca de 25 segundos para ser executado, a árvore realizou a mesma tarefa em apenas 2, provando que essa função Hash não é superior à Árvore para todos os tamanhos de entrada. Com funções mais eficientes do Hash, seria possível reduzir o tempo de execução distribuindo melhor as palavras entre as listas. Por fim, foi realizados testes com o Hash com k = 5 e 6 para comparar com a árvore, que não foram colocados em gráficos.



Com k = 5, o Hash demorou cerca de 2,5 segundos para realizar as operações, ainda acima da árvore, que levou somente 2. Com k = 6, o tempo foi ainda maior, devido à criação das listas que acaba sendo uma operação muito cara, levando o tempo de execução aos 7,5 segundos. Esses dois últimos testes evidenciam ainda mais a ineficiência da função hash com valores de k maiores, já que de “zzaaa” a “zzzzz” é improvável que exista alguma palavra, o que faz com que a maioria das listas fique com tamanho zero, mas ainda afetam o tempo de execução do programa e o custo em memória.

6. Conclusões

No Trabalho Prático 3, foram criados dois tipos de dicionários diferentes, um com o uso do Hash e outro com o uso de uma Árvore AVL. Depois, foi analisado o desempenho de ambos os dicionários, comparando-os e demonstrando os positivos e negativos de cada um.

Concluímos que para valores menores de entrada, é fácil encontrar uma função hash que seja mais eficiente que a Árvore AVL na execução das tarefas. Entretanto, com o crescimento da entrada, a Árvore se mantém com um bom desempenho, enquanto são necessárias funções hash cada vez mais elaboradas para que haja uma distribuição melhor dos Verbetes em listas, coisa que requer um conhecimento mais aprofundado sobre quais inícios de palavras são mais comuns, para personalizar a função hash e remover ou agrupar várias das listas que ficam com tamanho zero.

Então, ambas as implementações possuem seus casos de uso. Fica evidente que uma boa função hash é superior à Árvore AVL quando se pensa na situação hipotética que seja encontrada uma função que deixa os tamanhos das listas para uma entrada dessas de 200.000 linhas pequenos, para que o custo das operações no hash seja quase constante. A Árvore, por sua vez, é uma boa opção, que ainda possui custos baixos, sem a necessidade de criar uma função hash extremamente complexa para a boa distribuição dos Verbetes.

7. Bibliografia

Para funções padrão de c++, foi utilizado como base o link: https://cplusplus.com/forum/

Para melhor entendimento sobre a Árvore AVL, foi utilizado o site: https://www.geeksforgeeks.org/

8. Instruções para Compilação e Execução

Para a compilação do programa em sistema Linux, primeiro é necessário abrir uma janela do terminal dentro da pasta TP. Após feito isso, basta digitar a linha de comando “make” e o programa será compilado.

Para sua execução em ambiente Linux, é necessário também abrir a janela do terminal na pasta TP, para então digitar a linha de comando “./bin/run.out ARGUMENTOS”, onde “ARGUMENTOS” são os argumentos a seguir, somente o último sendo opcional:

- “-i entrada” ou “-i entrada.txt” - O argumento “-i” é seguido do nome do arquivo de texto que se deseja ler, que deve estar no diretório TP, com ou sem o “.txt”, que indica a extensão.
  Se esse argumento for omitido, o programa tenta abrir, por padrão, um arquivo “input.txt” no diretório TP. Caso o arquivo não exista, o programa produz um erro de arquivo não encontrado.

- “-o saida” ou “-o saida.txt” - O argumento “-o” define o nome do arquivo de saída criado pelo programa, com ou sem o “.txt”, que indica a extensão.
  Se esse argumento for omitido, o nome do arquivo criado, por padrão, é “output.txt”.
- “-t tipo” - O argumento “-t” é seguido do tipo de dicionário desejado.
  Pode ser colocado “-t a” ou “-t A” para a Árvore, ou “-t h” ou “-t H” para o Hash.
  Se omitido, o programa é executado, por padrão, com o Hash.

Assim, o programa é executado e seu tempo de execução é impresso no terminal.