Universidade Federal de Juiz de Fora

Instituto de Ciências Exatas

Departamento de Ciência da Computação

DCC012
ESTRUTURAS DE DADOS II

Trabalho 1 - Palavras do Momento do Twitter

Felipe Barra Knop - 201565553C

Lohan Rodrigues N. Ferreira - 201565082AC

Lucas Carvalho Ribeiro - 201565554AC

Professora Vânia de Oliveira Neves

Juiz de Fora - MG
25 de setembro de 2017

Sumário

Lista de Figuras

Lista de Tabelas

1 Introdução

Este relatório será dividido em duas partes principais:

- Parte 1 - Análise de algoritmos de ordenação e de tratamento de colisões
- Parte 2 - Análise das Estruturas de Dados para a implementação das Palavras do Momento

Na primeira parte, será tradada a análise de soluções para um problema muito comum no dia a dia de um trabalhador ou cientista na área da computação: a ordenação de um conjunto de elementos por meio de alguma informação contida dentro dos mesmos que forneça ideia de sequência. Aqui serão mostrados resultados que indicam o desempenho de algoritmos de ordenação diferentes a partir das seguintes métricas: quantidade de comparações e cópias de registros efetuadas e tempo gasto para ordenação.

Serão analisados também nesta parte o funcionamento e a eficiência de algoritmos de tratamento de colisões para funções de hashing, indicando seu desempenho a partir da quantidade de comparações efetuadas e memória gasta pela Tabela Hash resultante.

Na segunda parte, serão expostas as Estruturas de Dados utilizadas para a implementação das Palavras do Momento do Twitter.

1.1 Considerações iniciais

- Ambiente de desenvolvimento do código fonte: IntelliJ IDEA Community Edition.
- Linguagem utilizada: Java 8.
- Ambiente de desenvolvimento da documentação: ShareLaTeX - Editor online de LaTeX.

1.2 Dados utilizados

Todos os testes documentados neste relatório foram realizados com um conjunto real de Tweets públicos. A lista utilizada aqui possui 1.000.000 registros de Tweets diferentes e pode ser obtida no seguinte endereço:
`https://www.dropbox.com/s/07zflza2hj6njzj/tweets.txt?dl=0`

1.3 Testes

Todos os testes documentados neste relatório foram realizados da seguinte forma: para cada valor de N (1.000, 5.000, 10.000, 50.000, 100.000, 500.000 e 1.000.000) foram gerados 5 conjuntos de N elementos aleatórios sorteados com 5 sementes diferentes. Os resultados inclusos nas tabelas são formados pela média dos resultados de cada conjunto gerado para aquele número N de elementos.

Os resultados documentados de cada teste são como descritos na seção 1.

1.4 Realização das Atividades

Todos os membros do grupo contribuíram de alguma forma em todas as partes do trabalho, mas a divisão de tarefas foi, principalmente, a seguinte:

- Felipe: Parte 1
- Lohan: Relatório
- Lucas: Parte 2

2 Análise de algoritmos de ordenação e de tratamento de colisões

2.1 Algoritmos de Ordenação

Nesta seção poderão ser encontrados os resultados dos testes com os algoritmos de ordenação referentes a cada cenário da especificação do trabalho.

2.1.1 Cenário 1 - Impacto de diferentes estruturas de dados

Neste cenário analisamos o funcionamento do algoritmo QuickSort Recursivo com dois tipos diferentes de estruturas de dados:

- Inteiros armazenados em um vetor de tamanho N.
- Tweets armazenados em um vetor de tamanho N, ordenados pelo seu *TWEETID*.

A metodologia utilizada para os testes foi como descrita na seção 1.3.

Os resultados podem ser vistos na tabela 1:

	Tempo(ms)		Comparações		Cópias	
N	Inteiros	Tweets	Inteiros	Tweets	Inteiros	Tweets
1000	0	0	10648	10703	5695	5708
5000	4	1	72006	71985	38131	38131
10000	3	2	153868	154388	84318	85076
50000	13	20	943041	932154	491498	496513
100000	26	43	1993783	1905797	1058300	1054371
500000	176	304	11771340	1176890	6151070	6150876
1000000	396	617	24371459	24387610	12997845	12980673

Tabela 1: Resultado dos testes do Cenário 1

Como esperado, nota-se que o número de comparações e cópias cresce conforme o número de elementos dos conjuntos aumentam, porém algo interessante neste resultado é o fato de os resultados serem semelhantes tanto para inteiros quanto para Tweets, mostrando a eficiência do algoritmo independente da estrutura a ser ordenada. Uma análise mais intuitiva do tempo pode ser vista no gráfico 1:

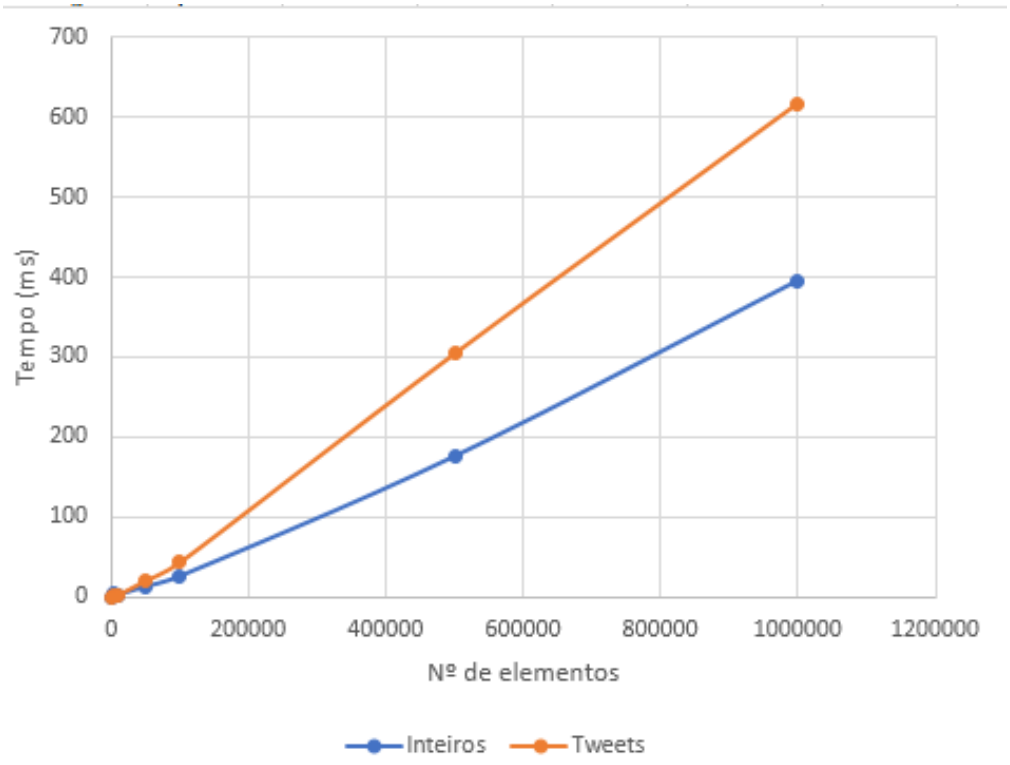

Figura 1: Tempo de execução em função do número de elementos

2.1.2 Cenário 2 - Impacto de variações do Quicksort

Neste cenário analisamos e comparamos, com testes semelhantes aos do cenário anterior, porém dessa vez utilizando somente os Tweets como elementos, a eficiência de três tipos diferentes de Quicksort: o QuickSort Recursivo já visto no cenário anterior, o QuickSort Mediana que utiliza sempre a mediana de K elementos do vetor escolhidos aleatoriamente como pivô da ordenação e o QuickSort Inserção que funciona de forma semelhante ao QuickSort Recursivo, porém quando suas divisões criam partes suficientemente pequenas (tamanho m) utilizamos o algoritmo de ordenação Insertion Sort para ordenar essas pequenas partes.

Os dados destes testes podem ser vistos nas tabelas 2, 3 e 4:

	Tempo(ms)				
		QuickSort Mediana		QuickSort Inserção	
N	QuickSort Recursivo	k = 3	k = 5	m = 10	m = 100
1000	0	1	0	0	0
5000	1	3	1	1	1
10000	2	19	3	12	2
50000	20	32	27	22	17
100000	43	39	40	38	40
500000	304	233	246	240	211
1000000	617	450	558	453	444

Tabela 2: Resultado de tempo gasto para os testes do Cenário 2

	Comparações				
		QuickSort Mediana		QuickSort Inserção	
N	QuickSort Recursivo	k = 3	k = 5	m = 10	m = 100
1000	10703	14837	14485	10575	9739
5000	71985	93567	89736	71388	67385
10000	154388	202460	193025	152556	144154
50000	932154	1250423	1159733	936465	896775
100000	1905797	2606372	2487821	1980473	1900636
500000	1176890	14684353	13787758	11697001	11302404
1000000	24387610	29845217	28342338	24191775	23409209

Tabela 3: Resultado de número de comparações para os testes do Cenário 2

	Cópias				
		QuickSort Mediana		QuickSort Inserção	
N	QuickSort Recursivo	k = 3	k = 5	m = 10	m = 100
1000	5708	8111	9110	5753	18285
5000	38131	47265	52011	38450	103889
10000	85076	101294	108978	85137	214091
50000	496513	582475	619936	493039	1139621
100000	1054371	1200752	1287278	1060786	2361826
500000	6150876	6855585	7197919	6092387	12492808
1000000	12980673	14584353	15134603	12688585	25467774

Tabela 4: Resultado de números de cópias de registros para os testes do Cenário 2

Em termos de tempo podemos notar que as outras variações do QuickSort Recursivo (Mediana e Inserção) tiveram melhor desempenho de tempo na ordenação de grandes conjuntos, sendo o QuickSort Inserção o mais rápido dentre eles.

Todos fazem um número relativamente semelhante de comparações em todos os casos, porém ao analisarmos o número de cópias notamos que o QuickSort Inserção é o que realizou mais, sendo então o que potencialmente utiliza mais memória para seu funcionamento.

A variação do K no QuickSort Mediana aparentemente não foi muito efetiva, pois o cálculo da mediana é relativamente complexo, o que tornou seu tempo de execução mais longo. Em contrapartida ao aumentar o valor de M no QuickSort Inserção podemos notar que este se tornou levemente mais rápido ao custo de aproximadamente o dobro do número de cópias.

2.1.3 Cenário 3 - Quicksort X InsertionSort X Mergesort X Heapsort X Meusort

Neste cenário analisamos o funcionamento e eficiência de outros algoritmos de ordenação de complexidades conhecidas: HeapSort, MergeSort, Insertion Sort, o QuickSort Inserção (que foi o mais bem sucedido no cénario anterior) e também o

RadixSort.

O RadixSort é um algoritmo de ordenação não comparativo, ou seja, não utiliza comparações entre elementos para ordená-los.

Funciona da seguinte forma: um vetor de mesmo tamanho daquele a ser ordenado é criado para servir de estrutura auxiliar, assim como um outro vetor de tamanho *radix*, que é a base dos elementos representados (2 para binário, 10 para decimal...), chamado de vetor contador.

A cada iteração do algoritmo, seleciona-se um mesmo dígito posicional de cada elemento (dependendo da variação do algoritmo, que pode ser LSD ou MSD, os dígitos são selecionados em ordem de menos para mais significativo ou vice-versa, respectivamente) e incrementa-se o valor na posição do vetor contador correspondente ao valor do dígito sendo analisado.

Após isso, percorre-se o vetor contador somando ao valor de cada posição o valor da posição anterior, para que não haja colisões futuramente. Em seguida, para cada elemento do vetor original, obtém-se o valor da posição do vetor contador que corresponde ao valor do dígito da iteração atual, subtrai-se 1, e insere-se tal elemento na posição do vetor auxiliar indicada pela substração que acabou de ser efetuada.

Finalmente, copia-se todos os valores do vetor auxiliar para o vetor original com a nova ordem e prossegue-se para a próxima iteração.

O algoritmo realiza w iterações, sendo w o tamanho da palavra, ou seja, a quantidade máxima de dígitos ou caracteres nos elementos do vetor.

O RadixSort possui, portanto, complexidade $O(wn)$ em tempo e $O(n + w)$ em espaço.

Uma ilustração do funcionamento do algoritmo pode ser vista no seguinte endereço:
`https://www.cs.usfca.edu/~galles/visualization/RadixSort.html`

A variação utilizada neste trabalho foi a LSD, que começa pelo dígito menos significativo, assim garantindo a estabilidade do algoritmo.

Os resultados obtidos para cada um dos algoritmos de ordenação podem ser encontrados nas tabelas 5, 6 e 7:

	Tempo(ms)				
N	QuickSort	InsertionSort	MergeSort	HeapSort	RadixSort
1000	0	0	0	0	1
5000	1	1	2	4	3
10000	2	4	5	4	4
50000	13	64	25	25	43
100000	31	230	68	56	51
500000	211	6160	426	447	293
1000000	426	24451	926	954	560

Tabela 5: Resultado de tempo gasto para os testes do Cenário 3

	Comparações				
N	QuickSort	InsertionSort	MergeSort	HeapSort	RadixSort
1000	9739	8580	10000	19154	0
5000	67385	54497	65000	119196	0
10000	144154	118979	140000	258433	0
50000	896775	711314	800000	1525225	0
100000	1900636	1522651	1700000	3250190	0
500000	11302404	8773649	9500000	18548760	0
1000000	23409209	18547549	20000000	39096894	0

Tabela 6: Resultado de número de comparações para os testes do Cenário 3

	Cópias				
N	QuickSort	InsertionSort	MergeSort	HeapSort	RadixSort
1000	18285	250847	20000	9077	15987
5000	103889	6267506	130000	57098	79801
10000	214091	25021247	280000	124216	159270
50000	1139621	623841427	1600000	737612	779955
100000	2361826	2497761410	3400000	1575095	1522156
500000	12492808	62468793816	19000000	9024380	6298044
1000000	25467774	250021454517	40000000	19048447	10116761

Tabela 7: Resultado de número de cópias de registros para os testes do Cenário 3

Existe muita coisa a ser observada aqui e podemos começar com o mais chamativo: o fato de o RadixSort não utilizar comparação alguma para seu funcionamento o torna completamente diferente dos outros algoritmos de ordenação, porém podemos notar que seu tempo de execução, apesar de bom, não supera todos os outros pelo fato de este algoritmo possuir um alto custo de memória adicional em seu funcionamento.

Outro detalhe importante é o fato de o QuickSort ter sido novamente o mais bem sucedido dentre todos os algoritmos pois na literatura encontramos informações de

que o HeapSort e MergeSort possuem menor ordem de complexidade e portanto o esperado era que estes fossem os de melhor desempenho nesse teste, porém na prática, como o QuickSort pode variar bastante dependendo da origem dos dados e da escolha do pivô, na média ele acaba sendo melhor que os outros apesar de ter complexidade maior num pior caso.

Como esperado, o InsertionSort é o menos eficiente dentre os algoritmos, sendo o mais demorado e realizando um número exorbitante de cópias de chave.

Os algoritmos HeapSort e MergeSort possuem desempenho aproximadamente semelhante conforme encontrado na literatura.

Um ultimo dado importante a ser mencionado é a relação entre o número de comparações e o número de cópias no MergeSort, onde o primeiro acaba sempre sendo exatamente a metade do segundo. Isso acontece por conta do método como este algoritmo ordena seus elementos que consiste sempre na divisão do seu vetor em subvetores que possuem metade de seu tamanho e pelo fato de as chaves, em geral, serem comparadas somente até metade do vetor e a outra metade deste vetor acaba por ser simplesmente copiada novamente.

2.2 Tratamento de Colisão

Desviando um pouco dos algoritmos de ordenação, apresentaremos agora a análise dos algoritmos de tratamento de colisão para inserção de um elemento em uma Tabela Hash.

2.2.1 Cenário 4 - Tratamento de Colisões: Endereçamento X Encadeamento

Este teste avalia o funcionamento de algoritmos de tratamento de colisão. Utilizando um código de *Hashing* simples, o Hashing Multiplicação, com valor de $\phi = 0.6180339887$, fazemos distribuição em uma tabela de espalhamento (Tabela Hash) e usamos diferentes algoritmos de tratamento de colisão para o caso em que a função Hashing tente alocar dois elementos em uma mesma posição da tabela, o que chamamos de colisão.

Os algoritmos de tratamento de colisão testados foram: Sondagem Lienar, Sondagem Quadrática, Re-Hashing, Encadeamento Separado e Encadeamento Coalescido (Sem Porão). Para cada algoritmo, armazenamos a memória gasta para sua realização e o número de comparações para análise, os dados podem ser encontrados na tabela 8 e 9:

	Memória(bytes)				
N	Sond. Lin.	Sond. Quad.	Re-Hashing	Encad. Separado	Encad. Coal.
1000	32000	32000	32000	16000	19904
5000	160000	160000	160000	80000	100374
10000	320000	320000	320000	160000	199929
50000	1600000	1600000	1600000	800000	1001996
100000	3200000	3200000	3200000	1600000	2001395
500000	16000000	16000000	16000000	8000000	10024476
1000000	32000000	32000000	32000000	16000000	20079801

Tabela 8: Resultado de memória gasta para os testes do Cenário 4

	Comparações				
N	Sond. Lin.	Sond. Quad.	Re-Hashing	Encad. Separado	Encad. Coal.
1000	547	422	688	1000	1465
5000	2593	2204	2490	5000	7429
10000	4926	4316	5110	10000	14747
50000	25409	21294	24917	50000	75658
100000	49992	42815	50002	100000	150642
500000	254840	217089	257101	500000	752879
1000000	517229	440452	531555	1000000	1517229

Tabela 9: Resultado de número de comparações para os testes do Cenário 4

Na tabela 8 podemos notar que o gasto de memória para os algoritmos de endereçamento é semelhante independente do algoritmo aplicado, o que é esperado já que neste metodo de tratamento de colisões a ideia é enviar elementos que sofreram colisões para outras posições disponíveis da tabela, não necessitando de aumento da tabela portanto. Já o consumo de memória dos algoritmos de encadeamento foi menor, o que pode implicar num processamento mais rápido que os algoritmos anteriores, porém o número de comparações se torna maior visto que os elementos tem de ser testados mais vezes durante sua alocação.

Analisando somente os dados desta tabela referentes aos algoritmos de endereçamento, podemos concluir que o melhor dentre eles foi a sondagem quadrática, sendo o que teve menor número de comparações, e portanto conseguiu tratar as colisões de maneira mais rápida, porém a diferença entre eles não é grande e vale lembrar que o algoritmo de re-hashing pode se tornar melhor ou pior baseado em quais funções de hashing estão sendo usadas para espalhamento na tabela.

Podemos concluir então, nesse teste, que os algoritmos para tratamento de colisão baseado em endereçamento tem melhor desempenho no quesito comparações enquanto os algoritmos baseados em encadeamento possuem melhor desempenho quanto ao consumo de memória.

Apesar de não haverem dados para diferentes tamanhos de Tabela Hash, é fácil perceber que uma Tabela Hash maior implicaria em uma diminuição da quantidade de comparações, pois a chance de colisão é menor, e em um aumento na quantidade de memória gasta, já que mais espaço seria reservado para os elementos na tabela, mesmo que grande parte desse espaço fique desocupado.

3 Análise das Estruturas de Dados para a implementação das Palavras do Momento

Nesta seção, serão expostas as estruturas de dados criadas para a implementação da segunda parte do trabalho, bem como os algoritmos de hashing, tratamento de colisões e ordenação utilizados, assim como a explicação para a utilização de cada um.

3.1 WordFrequency

Para tal implementação, foi criada a classe WordFrequency, que contém uma String representando uma palavra e um inteiro que representa a quantidade de vezes que essa palavra foi processada. Esta classe possui também métodos de obtenção da palavra e da quantidade, assim como para o incremento da quantidade.

Um diagrama que representa a classe WordFrequency pode ser visto na figura 2.

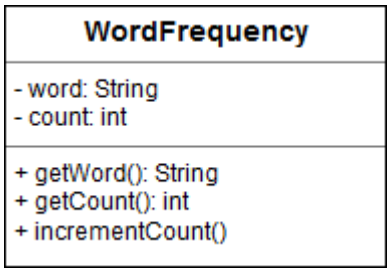

Figura 2: Diagrama da classe WordFrequency

3.2 WordFrequencyHashingMethod

A implementação desta parte do trabalho dependeu da criação da classe WordFrequencyHashingMethod, a qual continha a lógica de inserção de palavras na Tabela Hash.

A Tabela Hash foi implementada como uma lista de listas de WordFrequency, ou seja, *ArrayList<ArrayList<WordFrequency>>*, semelhante à forma da Tabela Hash do método de Encadeamento Separado, onde colisões são tratadas inserindo os elementos na lista daquela posição da tabela.

Esse formato de Tabela Hash foi escolhido por ter demonstrado bons resultados no cenário 4 (vide seção 2.2.1) em termos de memória total gasta.

A figura 3 contém um diagrama que representa um exemplo de segmento da Tabela Hash em dado momento.

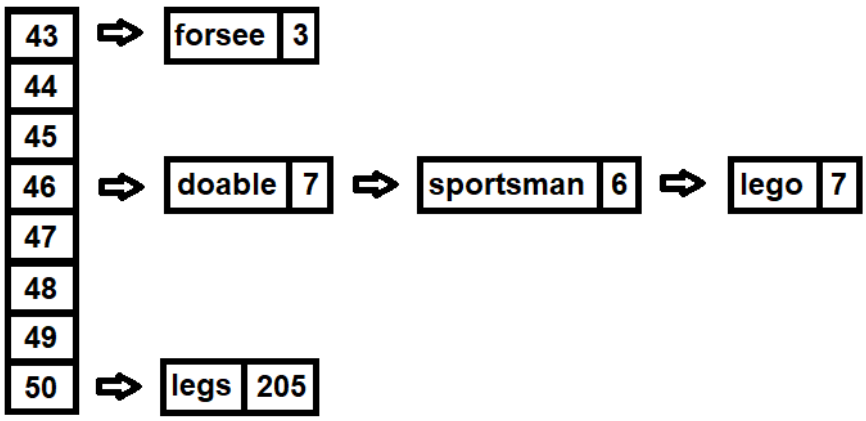

Figura 3: Tabela Hash com algumas palavras inseridas

O hash das palavras é feito utilizando o método *DJB2*, que inicia uma variável *hash* com o valor 5381 e, para cada caracter da palavra, multiplica o valor da variável *hash* por 33 e soma o valor correspondente ao caracter na tabela ASCII.

Esse método foi escolhido pois sua implementação padrão já é voltada para o hashing de Strings, é muito utilizado por sua eficiência e implementação simples e por ter demonstrado ser o mais eficiente em termos de colisão se comparado aos outros algoritmos ensinados em aula.

3.3 Ordenação das palavras por frequência

O passo final da implementação é imprimir as palavras mais frequentes com a quantidade de vezes que aparecem, portanto, após inserir todas as palavras na Tabela Hash, é necessário ordenar essas palavras.

Para isso, foi feito um método que percorre todas as *"linhas"* da Tabela Hash, inserindo todas as *"colunas"* em uma lista de WordFrequency. Finalmente, o método chama o MergeSort para ordenar esta lista em ordem decrescente da frequência de cada palavra.

O algoritmo MergeSort foi escolhido para esse passo pois, apesar de não ter obtido o melhor desempenho no cenário 3 (vide seção 2.1.3), foi implementado de forma iterativa, evitando erros do tipo StackOverflow, que poderiam ocorrer com o QuickSort se não houvesse memória suficiente, por ser recursivo.