CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS

BRUNO NASCIMENTO DAMACENA
RAMON GRIFFO COSTA

Prática 01 – Implementação em JAVA do TAD Árvore Binária de Pesquisa

Trabalho apresentado à disciplina de Laboratório de Algoritmos e Estrutura de Dados II do curso de Engenharia de Computação do Departamento de Computação do CEFET-MG

Professor Thiago de Souza Rodrigues.

Belo Horizonte
Março de 2018

1. Utilizando o Netbeans, crie um projeto chamado Prática01

2. Implemente a classe Item, como especificada abaixo para ser utilizada no T.A.D.

3. Implemente uma classe chamada ArvoreBinaria para manipular uma árvore binária de pesquisa onde os nós da árvore são objetos da classe No

   Os três primeiros items foram implementados e podem ser encontrados no arquivo zip do código fonte, que foi enviado anexo a esse relatório. O código fonte também pode ser encontrado no repositório do GitHub (https://github.com/BrunoDamacena/BinarySearchTree).

4. Realizar os seguintes experimentos:

   (a) gerar árvores a partir de n elementos ORDENADOS, com n variando de 1.000 até 9.000, com intervalo de 1.000.

   (b) gerar árvores a partir de n elementos ALEATÓRIOS (long j = obj.nextInt()), com n variando de 1.000 até 9.000, com intervalo de 1.000.

   (c) Fazer um único gráfico de n x número de comparações levando em consideração as árvores geradas com inserções ordenadas e aleatórias

   (d) Fazer um único gráfico de n x tempo gasto levando em consideração as árvores geradas com inserções ordenadas e aleatórias

   As árvores foram geradas durante a execução do código. Para gerar a árvore com elementos ordenados, foi criado um ArrayList de Items ordenados de forma crescente, com tamanho *size*. Para gerar a árvore com elementos aleatórios, foi feito um *Collections.shuffle(ArrayList)* antes da inserção na árvore, de modo que os elementos foram inseridos aleatoriamente. Para avaliar a busca, foi simplesmente executada a busca pelo elemento *size + 1*.

   Os testes foram executados 10 vezes, com o resultado de cada teste disponível no arquivo .zip, no diretório *results/raw*.
   Uma vez executados os testes, os resultados foram organizados em tabelas, uma para o número de comparações, outra para o tempo de execução.

Tabela 1: Comparações por Nº de Elementos

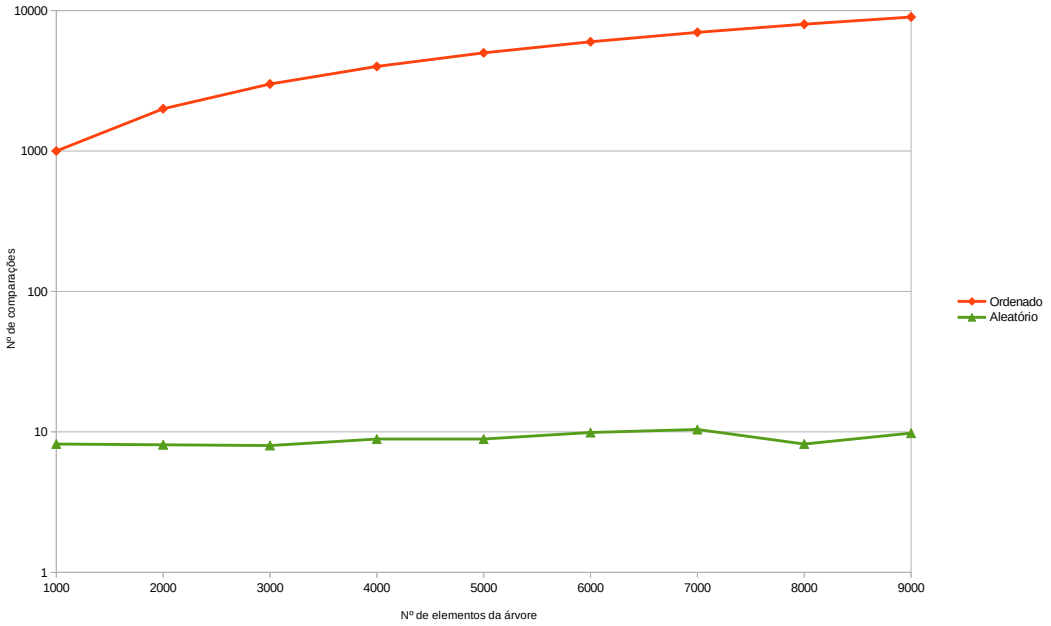
COMPARISONS MADE											
-	Output 1	Output 2	Output 3	Output 4	Output 5	Output 6	Output 7	Output 8	Output 9	Output 10	Mean
Ordered 1000	1000	1000	1000	1000	1000	1000	1000	1000	1000	1000	1000
Ordered 2000	2000	2000	2000	2000	2000	2000	2000	2000	2000	2000	2000
Ordered 3000	3000	3000	3000	3000	3000	3000	3000	3000	3000	3000	3000
Ordered 4000	4000	4000	4000	4000	4000	4000	4000	4000	4000	4000	4000
Ordered 5000	5000	5000	5000	5000	5000	5000	5000	5000	5000	5000	5000
Ordered 6000	6000	6000	6000	6000	6000	6000	6000	6000	6000	6000	6000
Ordered 7000	7000	7000	7000	7000	7000	7000	7000	7000	7000	7000	7000
Ordered 8000	8000	8000	8000	8000	8000	8000	8000	8000	8000	8000	8000
Ordered 9000	9000	9000	9000	9000	9000	9000	9000	9000	9000	9000	9000
-	-	-	-	-	-	-	-	-	-	-	-
Random 1000	5	4	7	8	12	9	9	8	7	13	8,2
Random 2000	8	6	5	11	8	6	6	11	12	8	8,1
Random 3000	7	8	10	5	6	7	8	8	11	10	8
Random 4000	10	9	9	8	7	10	9	8	7	12	8,9
Random 5000	10	10	11	11	6	8	12	8	7	6	8,9
Random 6000	10	9	11	7	10	10	15	9	9	9	9,9
Random 7000	8	8	12	13	8	12	10	7	14	12	10,4
Random 8000	8	8	6	5	5	8	12	11	11	8	8,2
Random 9000	8	6	9	11	11	6	9	8	17	13	9,8

Tabela 2: Tempo de Execução por Nº de Elementos

EXECUTION TIME (in ms)											
-	Output 1	Output 2	Output 3	Output 4	Output 5	Output 6	Output 7	Output 8	Output 9	Output 10	Mean
Ordered 1000	452	477	416	297	1197	597	398	287	289	458	486,8
Ordered 2000	420	523	530	324	627	420	335	296	463	421	435,9
Ordered 3000	432	432	517	1019	437	804	370	819	916	734	648
Ordered 4000	224	225	227	258	223	256	230	256	228	215	234,2
Ordered 5000	192	197	185	185	195	513	196	183	445	275	256,6
Ordered 6000	189	185	188	189	203	194	191	191	197	224	195,1
Ordered 7000	214	222	210	211	221	213	211	213	215	210	214
Ordered 8000	243	247	255	233	255	237	242	240	239	238	242,9
Ordered 9000	272	285	264	264	267	293	281	265	263	263	271,7
-	-	-	-	-	-	-	-	-	-	-	-
Random 1000	75	45	71	45	46	74	74	44	73	71	61,8
Random 2000	44	44	43	72	45	64	45	74	44	44	51,9
Random 3000	80	72	74	72	72	71	81	70	80	82	75,4
Random 4000	45	83	42	43	81	44	43	44	44	44	51,3
Random 5000	45	44	45	44	45	45	44	44	44	44	44,4
Random 6000	44	51	45	45	54	53	68	49	46	50	50,5
Random 7000	45	46	44	44	46	47	57	44	45	45	46,3
Random 8000	43	54	45	44	46	45	46	44	45	45	45,7
Random 9000	54	57	54	55	56	56	57	53	55	55	55,2

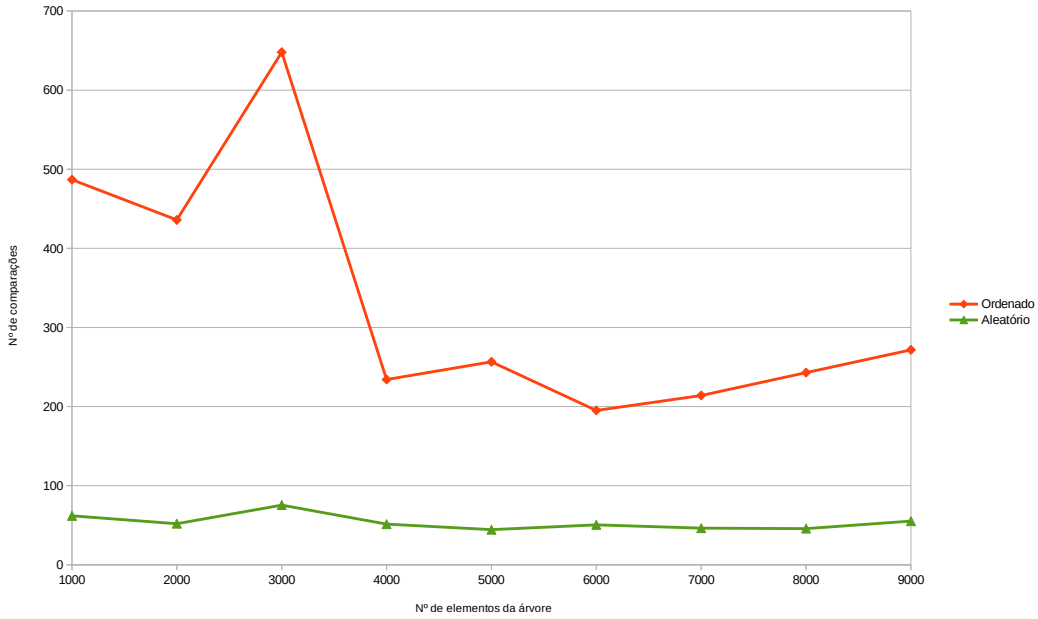
Com os dados no gráfico, calculamos a média dos testes e reproduzimos os gráficos. O eixo das abscissas representa o número de elementos da árvore, enquanto o eixo das ordenadas representa, no primeiro gráfico, o número de comparações feitas durante a busca, e no segundo gráfico, o tempo de execução da busca.

Gráfico 1: Comparações por Nº de Elementos



Obs.: O gráfico acima foi plotado em escala logarítmica, pois a diferença do número de comparações da árvore ordenada para a árvore aleatória é tão discrepante que a visualização do gráfico em escala linear é inviável.

Gráfico 2: Tempo de Execução por Nº de Elementos



5. Explique o comportamento dos gráficos gerados

No Gráfico 1, como esperado, o número de comparações na busca de um elemento inexistente em uma árvore ordenada é igual ao número de elementos da árvore em si, uma vez que ela tenha que percorrer toda a árvore para descobrir que o elemento não existe. Já na árvore aleatória, o número de comparações foi próximo de 10 comparações, o que é justificado, pois o número de comparações médio de uma árvore aleatória é de ln(size).

No Gráfico 2, fica evidente a imprecisão da utilização do tempo como fator. Principalmente na busca em árvore ordenada, graças a uma série de fatores como armazenamento em memória vs. armazenamento em cache, uso do processador do computador utilizado para os testes, imprecisão do método nanoTime(), entre outros, ficou visível que o resultado do gráfico foi inconclusivo.