Projeto 2 de Algoritmos e Estruturas de Dados

Neste trabalho, foram implementadas 6 estruturas de dados no intuito de compara-las em diferentes situações de inserção, remoção e busca afim de observar como elas se comportam em cada situação.

As estruturas as quais o projeto aborda foram:

- Vetor para realização de busca binária
- Lista Ordenada
- Lista Ordenada com Sentinela
- Árvore Binária de Busca
- Árvore AVL
- Lista Circular Dinâmica Duplamente Encadeada, organizada pela Frequência de Busca

Para cada uma das estruturas foi implementado um TAD em linguagem C referente a ela, contendo, no geral, funções de criação e alocação da estrutura, inserção, remoção e busca, além de funções auxiliares necessárias para o seu funcionamento, esses TADS são referentes aos arquivos: busca_bin.c/.h, lo.c/.h, los.c/.h, abb.c/.h, avl.c/.h, lfreq.c/.h.

De modo geral, para os TADS de Lista Ordenada, Lista Ordenada com Sentinela, Árvore Binária de Busca e Árvore AVL (com exceção da remoção) foram usados os códigos implementados em sala de aula durante o curso.

Para o restante, os TADS foram criados pelos integrantes do grupo

Em cada um dos TADS foi definido um tipo elemento referente ao tipo de informação que cada uma das estruturas guardas, que no caso são números inteiros. Ainda para evitar quaisquer problemas de compilação no início de cada arquivo.h foi utilizada a(s) macro(s) #ifndef ... #define...

Por fim foi criada uma main (main.c) com o objetivo de obter dados para vetores de tamanho n (com n variando entre 100,1000,10000,100000) as seguintes tabelas:

- Tabela 1: Tempo de insercao crescente
- Tabela 2: Tempo de insercao decrescente
- Tabela 3: Tempo de insercao aleatória
- Tabela 4: Tempo de remocao crescente (apos insercao crescente)
- Tabela 5: Tempo de remocao decrescente (apos insercao crescente)
- Tabela 6: Tempo de remocao aleatoria (apos insercao aleatoria)
- Tabela 7: Tempo de busca (apos insercao crescente)
- Tabela 8: Tempo de busca (apos insercao decrescente)
- Tabela 9: Tempo de busca (apos insercao aleatoria)

Após a implementação do projeto ele foi rodado em uma máquina virtual com o sistema operacional Linux e a distribuição Ubuntu devido a problemas que a plataforma Windows (onde o projeto foi desenvolvido, através da IDE CODEBLOCKS) apresentava ao rodar o algoritmo da Árvore Binária de Busca provavelmente devido ao número muito grande de

chamadas recursivas que esse algoritmo faz em determinada situação. Além disso, a saída foi testada e estava de acordo com a fornecida pelo arquivo testaFormato.c, como se pode ver na imagem abaixo.

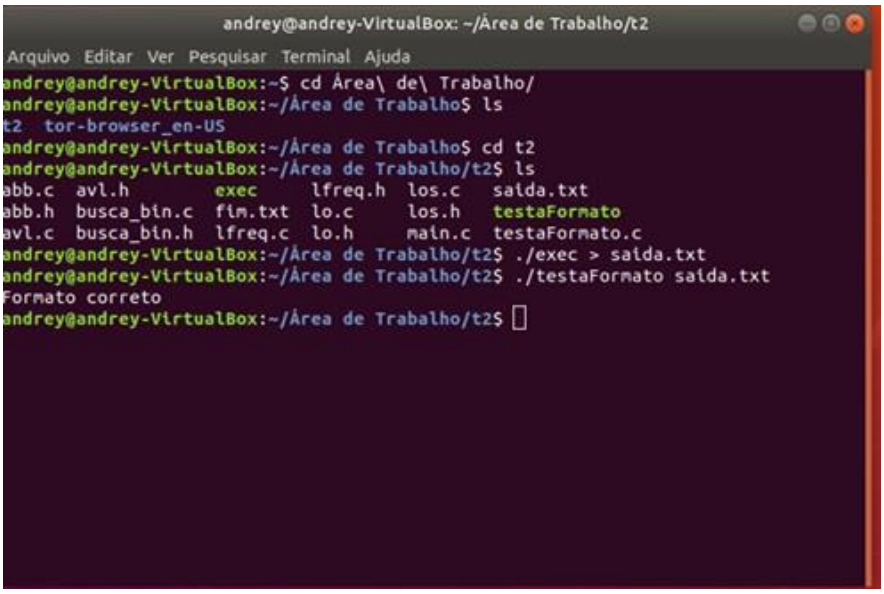

Figura 01 – Confirmação de Formato Correto na Saída do Programa

Dados coletados:

As médias temporais coletadas na saída do programa estão mostradas nas imagens abaixo, e serão exibidas na forma de gráficos logo após.

INSERÇÃO CRESCENTE:

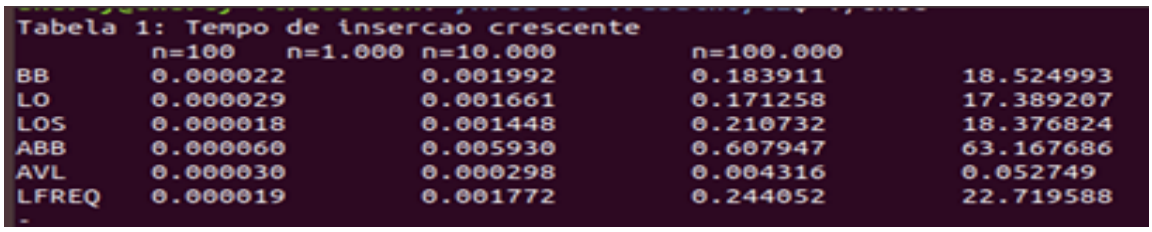

Figura 02 – Tabela de Tempo de Inserção Crescente

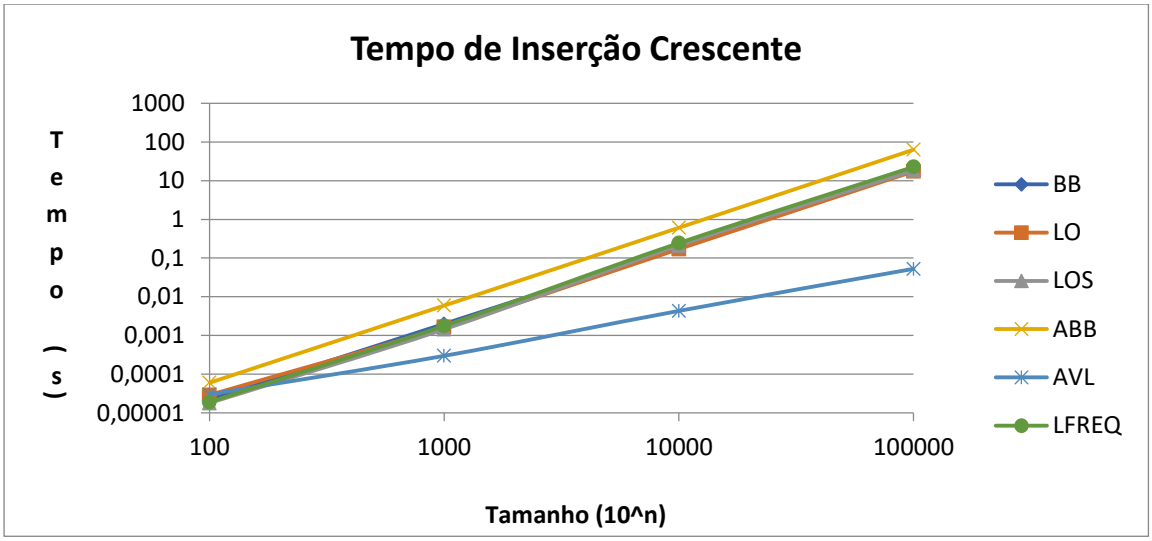

Gráfico 01 – Tempo de Inserção Crescente

Para o caso de Inserção Crescente, como se pode reparar no Gráfico 01 e Figura 02, para n = 100 o tempo de execução para cada estrutura é aproximadamente igual, porém quanto mais aumentamos o n, maior a diferença entre algumas. Com exceção de AVL e ABB (as estruturas em árvore), as demais estruturas apresentam resultados muito próximos para inserção crescente, uma vez que elas são estruturas lineares e a inserção ocorre de maneira sequencial (ordenada), mesmo com altos valores de n. A estrutura AVL se apresenta inicialmente como "mais uma", porém ao crescermos o n, repara-se que esta estrutura se destaca como a melhor, isto devido a facilidade nas comparações de menor ou maior. A estrutura ABB acaba se apresentando a pior uma vez que uma Inserção Crescente a transforma em uma Lista Ordenada, tendo complexidade O(n), seu pior caso, necessitando ainda de mais passos que suas concorrentes próximas.

INSERÇÃO DECRESCENTE:

Tabela 2: Tempo de insercao decrescente

	n=100	n=1.000	n=10.000	n=100.000	
BB	0.000034		0.003087	0.306531	31.314643
LO	0.000013		0.000052	0.000558	0.005290
LOS	0.000012		0.000050	0.000538	0.005209
ABB	0.000072		0.006219	0.623558	65.186100
AVL	0.000029		0.000294	0.003862	0.048500
LFREQ	0.000023		0.001521	0.160900	17.069830

Figura 03 - Tabela de Tempo de Inserção Decrescente

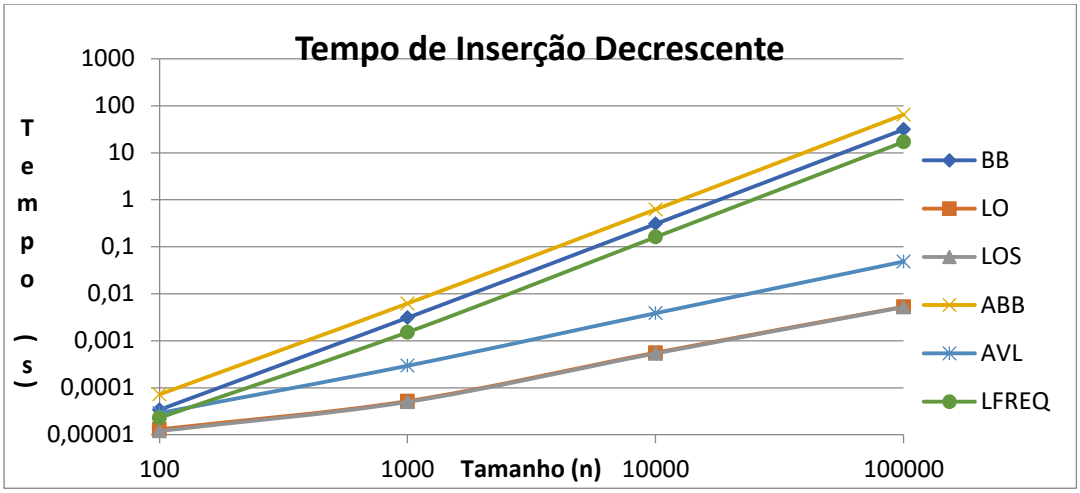

Gráfico 02 – Tempo de Inserção Decrescente

O segundo caso, Inserção Decrescente, é de certa forma parecido com o primeiro, porém os valores são adicionados do maior para o menor, em sequencia. Novamente a estrutura ABB se apresenta como pior caso, pois novamente teremos uma Lista Ordenada com complexidade O(n). LFREQ e BB não apresentam grandes diferenças de tempo de execução, sendo de valor médio para este caso, o mesmo podendo ser dito da AVL. Quem se destaca desta vem são as Lista Ordenada (LO) e Lista Ordenada com Sentinela (LOS). A principal diferença é que neste caso, a estrutura vai ordenando o vetor após cada inserção, ou seja, vai adicionando diretamente na posição correta, uma vez que é necessário apenas ir adicionando no inicio da lista.

INSERÇÃO ALEATÓRIA:

Tabela 3: Tempo de insercao aleatoria

	n=100	n=1.000	n=10.000	n=100.000	
BB	0.000030		0.002476	0.274295	24.891724
LO	0.000023		0.000673	0.104497	26.138034
LOS	0.000014		0.001389	0.241083	28.067633
ABB	0.000015		0.000187	0.002994	0.045601
AVL	0.000027		0.000324	0.005450	0.067429
LFREQ	0.000029		0.003455	0.712316	40.675276

Figura 04 - Tabela de Tempo de Inserção Aleatório

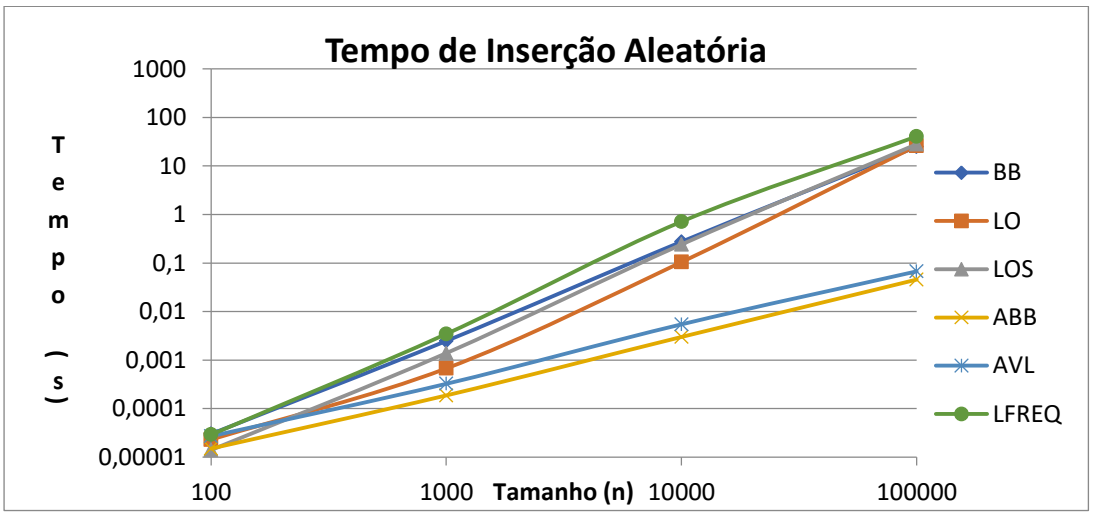

Gráfico 03 – Tempo de Inserção Aleatório

Por fim, a Inserção Aleatória. Para as estruturas de listas/que trabalham sequencialmente, este é o pior caso. A cada inserção é necessário que uma ordenação por todos os valores seja feita, precisando percorrer o vetor todo para encontrar a posição ideal. As estruturas em árvore se destacam novamente, desta vez com a ABB sendo a melhor opção. Como a inserção é aleatória, a chance de obtermos uma sequencia é menor, o que facilita com que a última espalhe os valores por sua estrutura. A AVL continua apresentando bons resultados, pois assim como sua companheira, precisa passar por menos valores para saber onde adicionar.

REMOÇÃO CRESCENTE (APÓS INSERÇÃO CRESCENTE):

Tabela 4: Tempo de remocao crescente (apos insercao crescente)

	n=100	n=1.000	n=10.000	n=100.000	
BB	0.000014		0.001034	0.093271	9.254958
LO	0.000023		0.000886	0.083665	8.398805
LOS	0.000010		0.000699	0.109127	7.065056
ABB	0.000031		0.002960	0.311603	33.040234
AVL	0.000007		0.000080	0.001078	0.014676
LFREQ	0.000001		0.000007	0.000058	0.000623

Figura 05 - Tabela de Tempo de Remoção Crescente

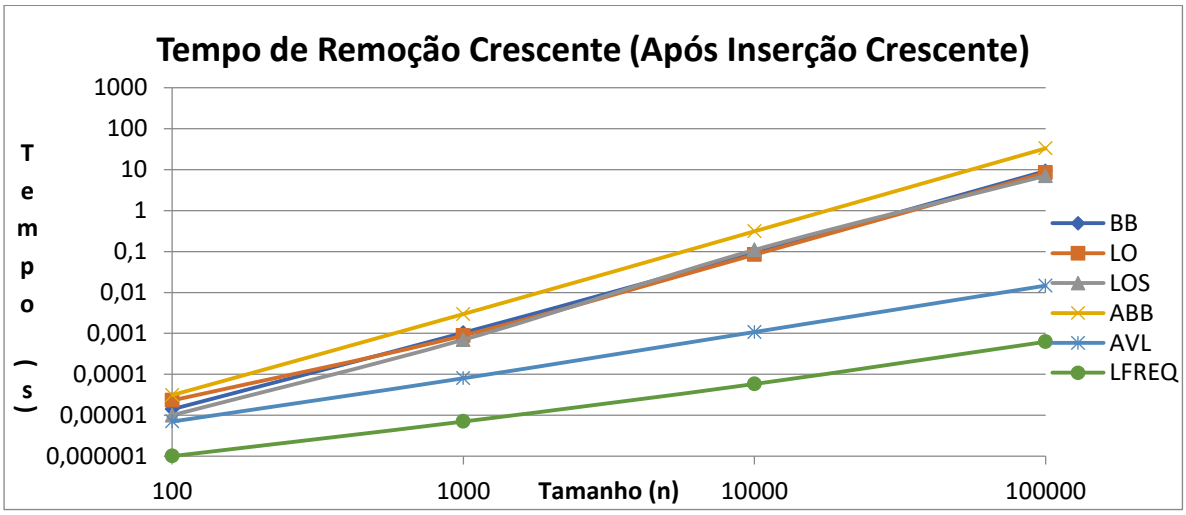

Gráfico 04 – Tempo de Remoção Crescente (Após inserção Crescente)

Começando as remoções, começando com a Crescente após inserção crescente. Para os casos de BB, LO e LOS o mesmo problema anteriormente citado ocorre. Eles precisarão passar por todos o s valores, um por um, removendo em ordem. Para a ABB é ainda pior, uma vez que após inserção crescente ela se torna uma lista ordenada com mais passos para remoção. A AVL continua mantendo seus bons resultados, removendo do topo da árvore para baixo, realizando algumas rotações quando necessário. A estrutura que se destaca desta vez é a LFREQ, que se trata de uma Lista Circular Dinâmica Duplamente Encadeada organizada pela frequência de busca.

REMOÇÃO DECRESCENTE(APÓS INSERÇÃO CRESCENTE):

Tabela 5: Tempo de remocao decrescente (apos insercao crescente)

	n=100	n=1.000	n=10.000	n=100.000	
BB	0.000012		0.000700	0.061652	6.244371
LO	0.000032		0.004974	0.957376	27.225298
LOS	0.000020		0.001692	0.239800	24.608010
ABB	0.000073		0.007516	0.797629	92.302291
AVL	0.000006		0.000064	0.000768	0.008060
LFREQ	0.000001		0.000006	0.000059	0.000568

Figura 06 - Tabela de Tempo de Remoção Decrescente

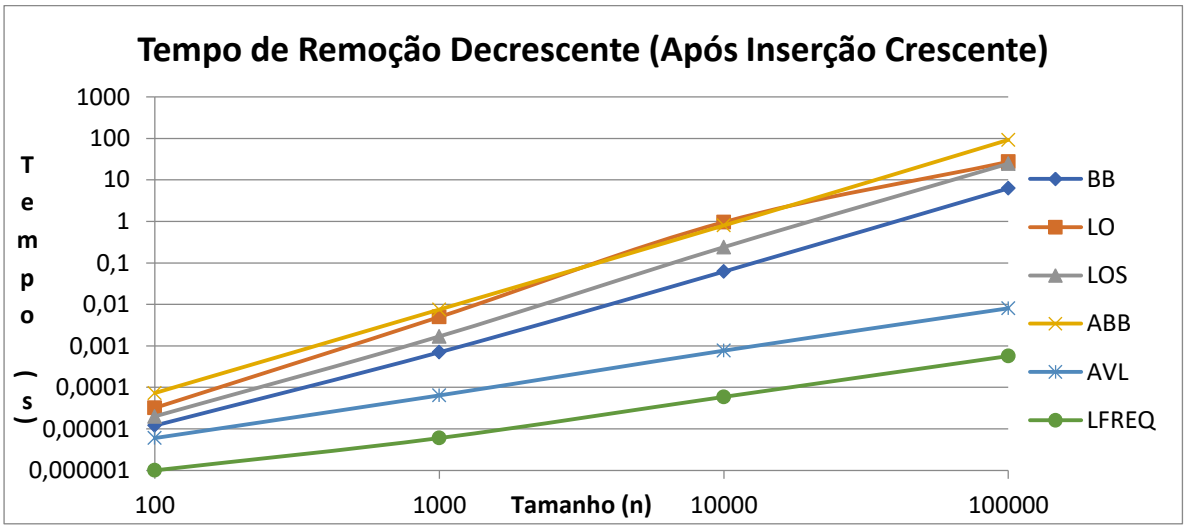

Gráfico 05 – Tempo de Remoção Decrescente (Após inserção Crescente)

Neste caso, remover decrescente após inserção também decrescente não nos traria muitas diferenças do caso anterior, portanto foi feita após inserção crescente. Novamente temos BB, LO e LOS com tempos intermediários, uma vez que para este caso devem percorrer até o fim da estrutura para encontrar o valor a ser retirado. O mesmo vale para a ABB, pois inserção crescente a transforma em uma lista ordenada. A AVL ainda apresenta resultados parecidos para os outros casos, mostrando sua consistência como estrutura de dados tanto para inserção como remoção. Já a LFREQ apresenta o melhor resultado novamente, principalmente pelo fato de ser uma lista duplamente encadeada, tendo fácil aceso as duas pontas da estrutura encadeada.

REMOÇÃO ALEATÓRIA (APÓS INSERÇÃO ALEATÓRIA):

Tabela 6: Tempo de remocao aleatoria (apos insercao aleatoria)

	n=100	n=1.000	n=10.000	n=100.000	
BB	0.000012		0.000380	0.029773	2.963915
LO	0.000015		0.001073	0.189994	47.332388
LOS	0.000015		0.002473	0.421462	49.401516
ABB	0.000031		0.000151	0.002098	0.029220
AVL	0.000007		0.000072	0.000920	0.012901
LFREQ	0.000004		0.000031	0.000339	0.003152

Figura 07 - Tabela de Tempo Remoção Aleatório

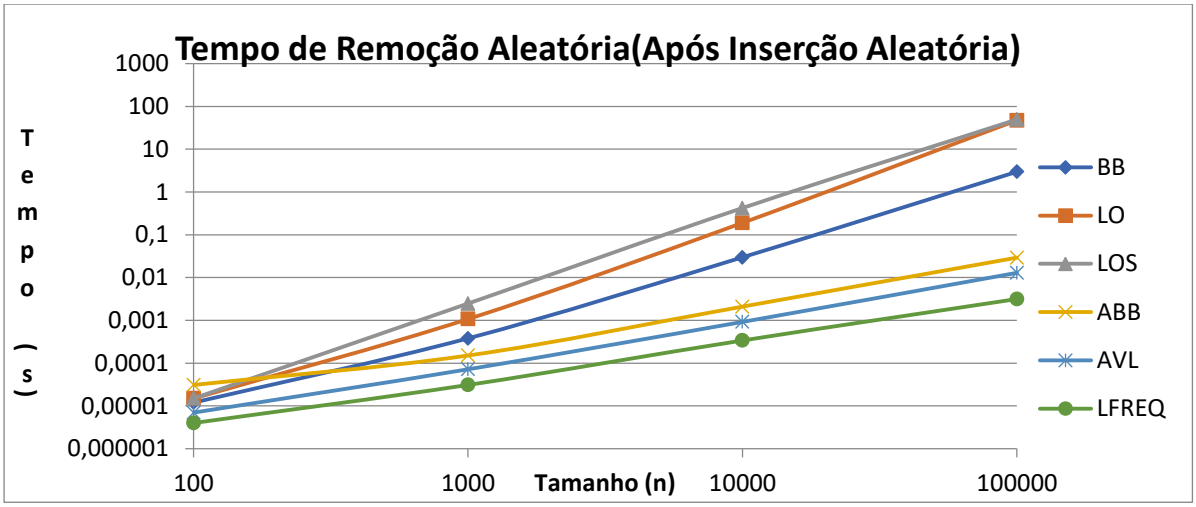

Gráfico 06 – Tempo de Remoção Aleatória(Após inserção Crescente)

As LO e LOS apresentam os piores resultados, pois devem buscar valores aleatórios no vetor até os encontrar ou decidir que não estão presentes, podendo chegar a ser O(n). A BB apresenta um resultado um pouco melhor devido ao fato de ir limitando a busca pelo elemento a cada tentativa, sendo mais fácil do que nos casos anteriores. Novamente as estruturas em árvore apresentam bons resultados (a ABB pela parte aleatória) e novamente a LFREQ apresentando o melhor resultado, diferentemente do esperado para uma lista encadeada. O resultado talvez possa ser explicado devido a buscas aleatórias que ocorreram previamente à remoção, alterando a ordem em que os valores eram dispostos, podendo facilitar ou dificultar a vida da remoção, também de valores aleatórios. Vale lembrar que para todos os casos de remoção (crescente, decrescente e aleatório) foram removidos n/2 elementos existentes e n/2 não existentes (tentativas de remoção).

BUSCA ALEATÓRIA (APÓS INSERÇÃO CRESCENTE):

Tabela 7: Tempo de busca (apos insercao crescente)

	n=100	n=1.000	n=10.000	n=100.000
BB	0.000010	0.000106	0.001403	0.015422
LO	0.000051	0.002468	0.243776	24.595930
LOS	0.000025	0.002295	0.328720	26.000413
ABB	0.000077	0.008462	0.871962	90.320982
AVL	0.000010	0.000119	0.001611	0.029637
LFREQ	0.000028	0.003105	0.376265	31.981602

Figura 08 - Tabela de Tempo de Busca Aleatória

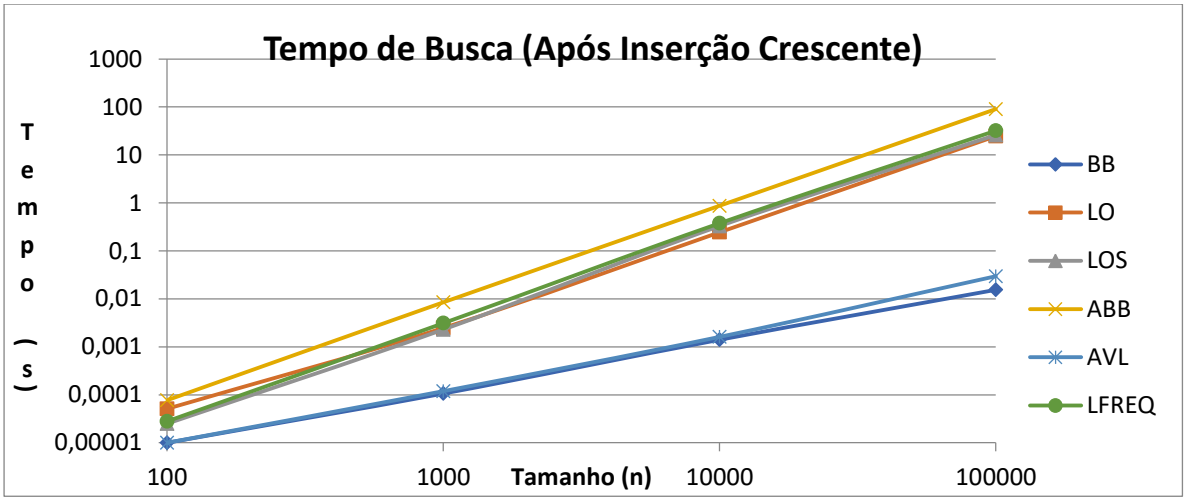

Gráfico 07 – Tempo de Busca (Após inserção Crescente)

As buscas foram realizadas sempre aleatoriamente com n/2 elementos sabidos existentes e n/2 sabidos não existentes. Como era de se esperar, LO, LOS, LFREQ e ABB (inserção crescente = lista ordenada) foram os piores casos devido a valores aleatórios sendo buscados em listas ordenadas. Caso a busca fosse crescente, seus resultados seriam melhores (mas ainda precisaria percorrer toda a lista para encontrar os últimos valores). A AVL não surpreende e continua com bons resultados, como esperado. Para todos casos de busca a AVL apresenta um caso especial onde a cada comparação se reduz pela metade o número de elementos comparáveis. Assim para 1 bilhão de elementos, são necessárias apenas 30 comparações até se encontrar (ou decidir que não se encontra), sendo O(log n). Incrivelmente temos a BB com bons resultados. Isto se deve ao fato de ela também reduzir o número de elementos comparáveis durante a busca, tendo resultados muito próximos da AVL.

BUSCA ALEATÓRIA (APÓS INSERÇÃO DECRESCENTE):

Tabela 8: Tempo de busca (apos insercao decrescente)

	n=100	n=1.000	n=10.000	n=100.000	
BB	0.000008		0.000095	0.001240	0.015470
LO	0.000028		0.002425	0.246778	25.518864
LOS	0.000025		0.002193	0.216402	23.078178
ABB	0.000029		0.002786	0.279800	30.745944
AVL	0.000009		0.000117	0.001618	0.028838
LFREQ	0.000027		0.002434	0.249342	26.895573

Figura 09 - Tabela de Tempo de Busca Aleatória

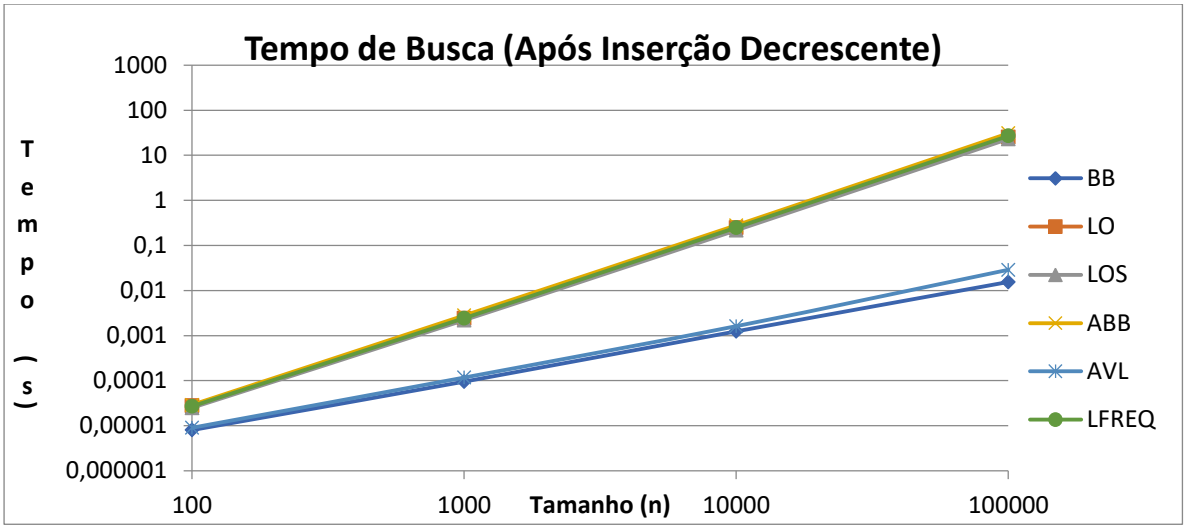

Gráfico 08 – Tempo de Busca(Após Inserção Decrescente)

O resultado é muito parecido com o do caso anterior. Os problemas apresentados são os mesmos com a diferença que os valores estão em ordem decrescente desta vez. Mesmo nos casos em que se pararia a busca de elementos não existentes logo de cara ao comparar que ele é maior que já o primeiro elemento das listas (se tornando mais rápido), para encontrar os elementos existentes (randomizados entre os n/2 primeiros), teria que se percorre boa parte das listas, os tornando ineficientes da mesma forma. Novamente, AVL e BB se apresentam como melhores opções.

BUSCA ALEATÓRIA (APÓS INSERÇÃO ALEATÓRIA):

Tabela 9: Tempo de busca (apos insercao aleatoria)

	n=100	n=1.000	n=10.000	n=100.000	
BB	0.000009		0.000095	0.001218	0.015587
LO	0.000016		0.001567	0.308947	74.972418
LOS	0.000024		0.004046	0.702895	77.365274
ABB	0.000010		0.000133	0.002074	0.027508
AVL	0.000009		0.000110	0.001608	0.023997
LFREQ	0.000052		0.007029	1.355377	58.983653

Figura 10 – Tabela de Tempo de Busca Aleatória

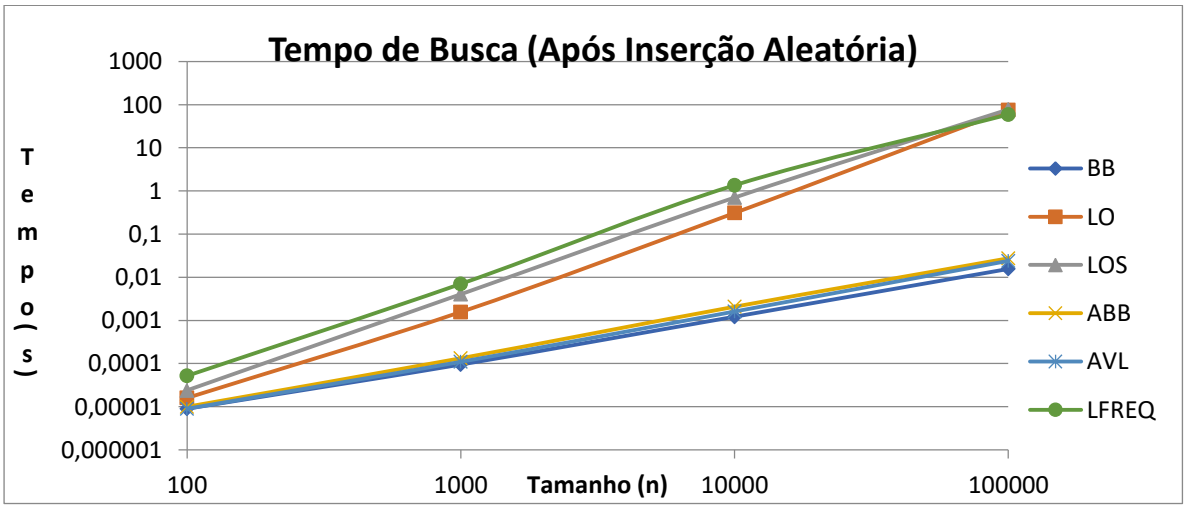

Gráfico 09 – Tempo de Busca (Após Inserção Aleatória)

Último caso averiguado no trabalho, a busca aleatória após inserção aleatória nos dá resultados interessantes. Assim como nos 2 casos anteriores, AVL e BB apresentam ótimos resultados. Temos a presença da ABB com bons resultados novamente, devido à inserção aleatória já explicada anteriormente. As LO, LOS e LFREQ apresentam novamente os piores resultados, como se era esperado. A LFREQ começa a apresentar resultado melhores que suas concorrentes quanto maior o valor de n, isso pode ter acontecido devido a busca ser aleatória com chances de buscar o mesmo elemento e uma vez que os elementos mais buscados acabam ficando no inicia da lista, seu tempo de busca pode ter diminuído.

Assim, após compara cada estrutura em cada um dos 9 casos, para um caso geral a melhor estrutura seria a AVL. Em todos os casos este tipo apresentou resultados consistentes e próximos de constantes, não sendo muito afetada pelas diferenças dos casos de estudo. As demais estruturas se apresentaram boas em alguns casos e muito ruins em outros, não podendo ser indicadas para casos gerais. Assim, apesar de uma implementação mais complexa/complicada, a AVL é nossa recomendação.

CONCLUSÃO

O desenvolvimento deste trabalho teve intuito de entender diferentes estruturas de dados e como cada uma responde, temporalmente, a diferentes tipos de processo de inserção, remoção e busca. Foi possível distinguir alguns pontos já durante as implementações onde começamos a imaginar como aquela estrutura iria se sair em cada caso. Assim foi possível encontrar erros mais facilmente após cada teste, uma vez que o comportamento esperado não era apresentado. Após o fim da coleta de dados, pudemos confirmar tais teorias e realmente ver a diferença que cada estrutura tem em possíveis situações cotidianas.

Por fim, pudemos afirmar com clareza que para casos gerais, a estrutura AVL é recomendada, obtendo o melhor caso em todos os testes realizados.