

Algoritmos e Estrutura de Dados: análise da complexidade de árvores binárias

Gabriele S. Araújo¹, Adrielson F. Justino¹, , Bruno R.C. Alves¹

¹ Departamento de Engenharia da Computação
Universidade Estadual do Maranhão – São Luís – MA – Brazil

`gabimitusa@gmail.com, adrielferreira28@gmail.com, bruno-rca@hotmail.com`

Abstract. *This meta-paper describes the style to be used in articles and short papers for SBC conferences. For papers in English, you should add just an abstract while for the papers in Portuguese, we also ask for an abstract in Portuguese (“resumo”). In both cases, abstracts should not have more than 10 lines and must be in the first page of the paper.*

Resumo. *Este meta-artigo descreve o estilo a ser usado na confecção de artigos e resumos de artigos para publicação nos anais das conferências organizadas pela SBC. É solicitada a escrita de resumo e abstract apenas para os artigos escritos em português. Artigos em inglês deverão apresentar apenas abstract. Nos dois casos, o autor deve tomar cuidado para que o resumo (e o abstract) não ultrapassem 10 linhas cada, sendo que ambos devem estar na primeira página do artigo.*

1. Introdução

A análise de complexidade de algoritmos é fundamental para o desenvolvimento eficiente de *softwares* [Bournez et al. 2020]. Com novos algoritmos, a análise de complexidade de tempo e espaço é imperativa para demonstrar melhorias sobre as técnicas existentes e estabelecer sua utilidade [Tarek 2007]. Independentemente da velocidade dos computadores ou do custo da memória, a eficiência sempre será um fator importante na decisão [Neapolitan and Naimipour 2004].

Para indicar a complexidade de um algoritmo, é utilizada a notação Big O, que fornece uma estimativa do limite superior dos recursos computacionais necessários para implementar um algoritmo [Tarek 2007]. Através da análise assintótica, é possível analisar o comportamento de um algoritmo em termos de tempo e espaço, proporcionando uma maneira padronizada de comparar diferentes abordagens [Rubinstein-Salzedo 2018].

No contexto da programação, engenharia de software e ciência da computação, a árvore é uma das mais importantes estruturas de dados não lineares, possuindo uma disposição hierárquica dos dados, ao contrário das listas, onde os dados são sequenciais. Tal hierarquia é refletida na sua estrutura, onde cada elemento ou nó pode ter conexões denominadas ramos, ou filhos, formando subárvores [Goodrich and Tamassia 2013]. Devido à sua versatilidade, diversas variações das árvores foram desenvolvidas, entre elas a árvore binária, onde cada nó possui no máximo dois filhos, facilitando operações de busca e ordenação [Goodrich and Tamassia 2013].

A eficiência das operações básicas como inserção, busca e exclusão em árvores binárias é diretamente influenciada pela estrutura da árvore, tornando a análise de complexidade um aspecto vital do seu estudo. Desta forma, o objetivo deste estudo é desenvolver uma análise detalhada da complexidade de algoritmos relacionados a árvores binárias, utilizando a notação Big O. Serão exploradas as diferenças de desempenho entre

árvores binárias, como árvores de busca binária (BST) e árvores AVL, além de fornecer uma compreensão clara de como diferentes estruturas de árvore afetam a eficiência das operações.

Para alcançar esse objetivo, será adotada uma abordagem empírica e teórica, detalhando o processo de implementação dos algoritmos, desenvolvendo testes para avaliar seu desempenho e utilizando ferramentas específicas para medir e analisar os resultados. Por meio de gráficos e tabelas, os resultados obtidos serão apresentados, permitindo uma comparação visual e quantitativa das diferentes abordagens. Por fim, será realizada uma discussão sobre como os resultados experimentais se comparam com a teoria, abordando os desafios encontrados e as descobertas feitas ao longo do estudo.

2. Fundamentação Teórica

2.1. Árvores Binárias

Uma árvore binária é uma estrutura de dados hierárquica onde cada nó tem, no máximo, dois filhos (filho esquerdo e filho direito), podendo ou não ser ordenada. No caso de ser ordenada, as subárvores de cada nó são identificadas por sua posição, sendo uma denominada de subárvore esquerda e a outra de subárvore direita [Edelweiss and Galante 2009].

Dentro das árvores binárias, a árvore busca binária (BST) se destaca como uma variação com a propriedade adicional de que, para cada nó, todos os valores na subárvore esquerda são menores e todos os valores na subárvore direita são maiores [CORMEN et al. 2012]. Esta propriedade facilita operações como busca, inserção e exclusão de elementos contendo uma complexidade $O(h)$, onde h é a altura da árvore, com um tempo médio de $O(\log n)$ para uma árvore balanceada [Lin 2019].

No entanto, no pior caso quando a árvore se torna desbalanceada, a altura pode chegar a ser igual ao número de nós ($h = n$), degradando a complexidade de tempo das operações para $O(n)$ e se comportando exatamente como uma lista (duplamente) encadeada [Goodrich and Tamassia 2013]. Essa degeneração ocorre quando os dados são inseridos em ordem crescente ou decrescente, fazendo com que a árvore cresça apenas em uma direção, tornando-se linear [Jindal et al. 2010].

Para superar essa limitação, Adelson-Velsky e Landis introduziram as árvores AVL, uma estrutura de dados que garante o balanceamento da árvore após cada operação de inserção ou remoção [Goodrich and Tamassia 2013]. As árvores AVL mantêm a diferença de altura entre subárvores de qualquer nó em no máximo uma unidade, garantindo um balanço ideal para que as operações de inserção, busca e exclusão possam ser realizadas em tempo $O(\log n)$ mesmo no pior caso.

O balanceamento em árvores AVL é alcançado através do cálculo e manutenção do “fator de balanceamento” de cada nó, que é a diferença entre a altura da subárvore direita e a altura da subárvore esquerda. Se o fator de balanceamento de um nó ultrapassar o valor absoluto de 1, a árvore realiza rotações (simples ou duplas) para restaurar o equilíbrio [Edelweiss and Galante 2009]. Essas rotações são operações locais que envolvem a reorganização de alguns nós, mantendo a ordem dos elementos e garantindo que a árvore permaneça balanceada.

Dessa forma, as árvores AVL oferecem um desempenho consistente e eficiente em todas as operações, independentemente da ordem dos dados inseridos.

2.2. Notação Big O

A notação Big O fornece uma forma de expressar o comportamento assintótico de um algoritmo, ou seja, como o tempo de execução ou o uso de memória cresce com o aumento

do tamanho da entrada [Mala and Ali 2022]. Por exemplo:

O(1): Se uma função f é $O(1)$, isso significa que o valor de f é constante e não cresce com o tamanho da entrada. Um exemplo é $f(n) = e^{-n}$, onde $n \in N$.

O(n): Se uma função f é $O(n)$, isso significa que f cresce linearmente com o tamanho da entrada. Percorrer uma lista é $O(n)$ porque cada item deve ser visitado.

O(n²): Se uma função f é $O(n^2)$, isso significa que f cresce de forma quadrática com o tamanho da entrada.

O(2ⁿ): Se uma função f é $O(2^n)$, isso significa que f cresce de forma exponencial com o tamanho da entrada.

O(log n): Se uma função f é $O(\log n)$, isso significa que f cresce de forma logarítmica com o tamanho da entrada. Um exemplo é o algoritmo de busca binária.

O(n log n): Se uma função f é $O(n \log n)$, isso significa que f cresce de forma linear multiplicada pelo logaritmo do tamanho da entrada.

3. Metodologia

Para realizar a análise de complexidade de Árvores Binárias, foram seguidos os seguintes passos:

3.1. Implementação da Estrutura de Dados

A implementação das estruturas de dados envolveu a definição das classes `No` e `ArvoreBinariaDeBusca` para a Árvore Binária de Busca (BST) e das classes `NoAVL` e `ArvoreAVL` para a Árvore AVL. A classe `No` define os nós da árvore binária, enquanto a classe `ArvoreBinariaDeBusca` implementa as operações fundamentais como inserção, busca e remoção. Da mesma forma, a classe `NoAVL` define os nós da árvore AVL, e a classe `ArvoreAVL` implementa as operações básicas juntamente com o balanceamento da árvore.

3.2. Desenvolvimento de Testes

Para avaliar o desempenho das operações nas árvores binárias, foram criados cenários de teste com diferentes tamanhos de dados (2000, 4000, 6000, 8000 e 10000 elementos). Utilizou-se uma abordagem empírica para testar a eficiência das operações de inserção, busca e remoção. Foram gerados dados aleatórios e casos extremos (dados em ordem decrescente) para avaliar o desempenho em situações variadas. As funções de medição de tempo foram implementadas para calcular a duração das operações em ambas as estruturas de árvore.

3.3. Ferramentas Utilizadas

A linguagem de programação Java foi escolhida para a implementação devido ao seu desempenho e ao forte suporte a estruturas de dados. Utilizou-se a biblioteca `java.util`¹ para manipulação de dados e geração de valores aleatórios. Os resultados dos testes foram exportados para arquivos *Comma-separated values* (CSV) utilizando a classe `FileWriter`², permitindo a análise incremental dos dados. O desenvolvimento foi realizado no ambiente de desenvolvimento integrado (IDE) IntelliJ³. Todo código com suas respectivas funções e materiais adicionais estão disponíveis em material suplementar no seguinte repositório do *Github*: https://github.com/complexidade_arvores.

¹<https://docs.oracle.com/javase/8/docs/api/java/util/package-summary.html>

²<https://docs.oracle.com/javase/8/docs/api/java/io/FileWriter.html>

³<https://www.jetbrains.com/idea/>

3.4. Procedimentos de Implementação

A implementação iniciou com a definição das classes para representar os nós e as árvores. Na `ArvoreBinariaDeBusca`, os métodos `inserir`, `buscar` e `remover` foram desenvolvidos para realizar as operações básicas. A classe `ArvoreAVL` incluiu, além dessas operações, métodos para rotacionar os nós e manter a árvore balanceada. As funções `measureTime` e `exportToCSV` foram desenvolvidas para medir o tempo das operações e exportar os resultados para um arquivo CSV.

3.5. Execução dos Testes

Os testes foram executados em diferentes tamanhos de dados: 2000, 4000, 6000, 8000 e 10000 elementos. Para cada tamanho de dado, foram gerados valores aleatórios, e as operações de inserção, busca e remoção foram realizadas nas árvores BST e AVL. O tempo de execução de cada operação foi medido e registrado. Os resultados foram exportados incrementalmente para um arquivo CSV, permitindo uma análise contínua do desempenho.

3.6. Análise dos Resultados

Os resultados obtidos foram visualizados por meio de gráficos gerados com ferramentas de análise de dados. Esses gráficos permitem uma comparação visual e quantitativa das diferentes abordagens, facilitando a interpretação do desempenho das operações nas árvores BST e AVL. A análise comparativa ajudou a identificar as vantagens e desvantagens de cada estrutura de árvore em termos de complexidade de tempo.

3.7. Análise de Complexidade

A análise da complexidade dos algoritmos foi realizada para entender o comportamento assintótico de cada um. A complexidade de tempo e espaço foi determinada utilizando a notação Big O, que descreve o crescimento do tempo de execução ou do uso de memória em função do tamanho da entrada.

As complexidades das operações para as Árvores Binárias de Busca (BST) e Árvores AVL podem ser observadas na Tabela 1.

Tabela 1. Complexidade das Operações nas Árvores Binárias de Busca (BST) e Árvores AVL

Operação	Árvore AVL		Árvore Binária de Busca (BST)	
	Médio	Pior	Médio	Pior
Inserção	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
Remoção	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
Busca	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
Travessia	$O(n)$	$O(n)$	$O(n)$	$O(n)$

Para a operação de inserção na classe `ArvoreBinariaDeBusca`, a altura da árvore determina a complexidade. No caso médio, quando a árvore está balanceada, a altura é aproximadamente $\log n$, resultando em uma complexidade de $O(\log n)$. No pior caso, se a árvore estiver desbalanceada (semelhante a uma lista encadeada), a altura pode ser n , levando a uma complexidade de $O(n)$. A implementação da operação de inserção na `ArvoreBinariaDeBusca` começa na raiz e percorre a árvore para encontrar o local apropriado para o novo nó. A função `inserir` e sua recursiva `inserirRecursivo` são responsáveis por essa operação.

A operação de busca na `ArvoreBinariaDeBusca` é similar à inserção. No caso médio, a complexidade é $O(\log n)$ devido à altura balanceada da árvore. No entanto, no pior caso, a busca pode percorrer toda a árvore, resultando em uma complexidade de $O(n)$. A busca percorre a árvore a partir da raiz para encontrar o nó desejado. A função `buscar` e sua recursiva `buscarRecursivo` implementam essa operação.

A remoção na `ArvoreBinariaDeBusca` também depende da altura da árvore. No caso médio, a complexidade é $O(\log n)$, pois a árvore está balanceada. No pior caso, a remoção pode ter uma complexidade de $O(n)$ se a árvore estiver completamente desbalanceada. A remoção na `ArvoreBinariaDeBusca` envolve encontrar o nó a ser removido e reestruturar a árvore conforme necessário. As funções `remover` e `removerRecursivo` são utilizadas para essa operação.

Para a classe `ArvoreAVL`, a operação de inserção é mais complexa devido ao balanceamento automático da árvore. No entanto, isso garante que a altura da árvore permaneça $O(\log n)$, resultando em uma complexidade de inserção de $O(\log n)$. A inserção na `ArvoreAVL` é similar à inserção na `ArvoreBinariaDeBusca`, mas inclui etapas adicionais para manter o balanceamento. A função `inserir` e sua recursiva `inserirRecursivo`, junto com as funções de rotação `rotacaoEsquerda` e `rotacaoDireita`, implementam essa operação.

A busca na `ArvoreAVL` também se beneficia do balanceamento, mantendo uma complexidade de $O(\log n)$. A busca na `ArvoreAVL` é igual à busca na `ArvoreBinariaDeBusca`, mas a altura balanceada garante a complexidade $O(\log n)$. As funções `buscar` e `buscarRecursivo` implementam essa operação.

A remoção na `ArvoreAVL` envolve reestruturar a árvore e garantir que ela permaneça balanceada. Assim como a inserção e a busca, a remoção na `ArvoreAVL` tem complexidade $O(\log n)$ devido ao balanceamento automático. A remoção na `ArvoreAVL` envolve não só a remoção do nó, mas também a reestruturação e o balanceamento da árvore. As funções `remover` e `removerRecursivo`, junto com as funções de rotação `rotacaoEsquerda` e `rotacaoDireita`, são responsáveis por essa operação.

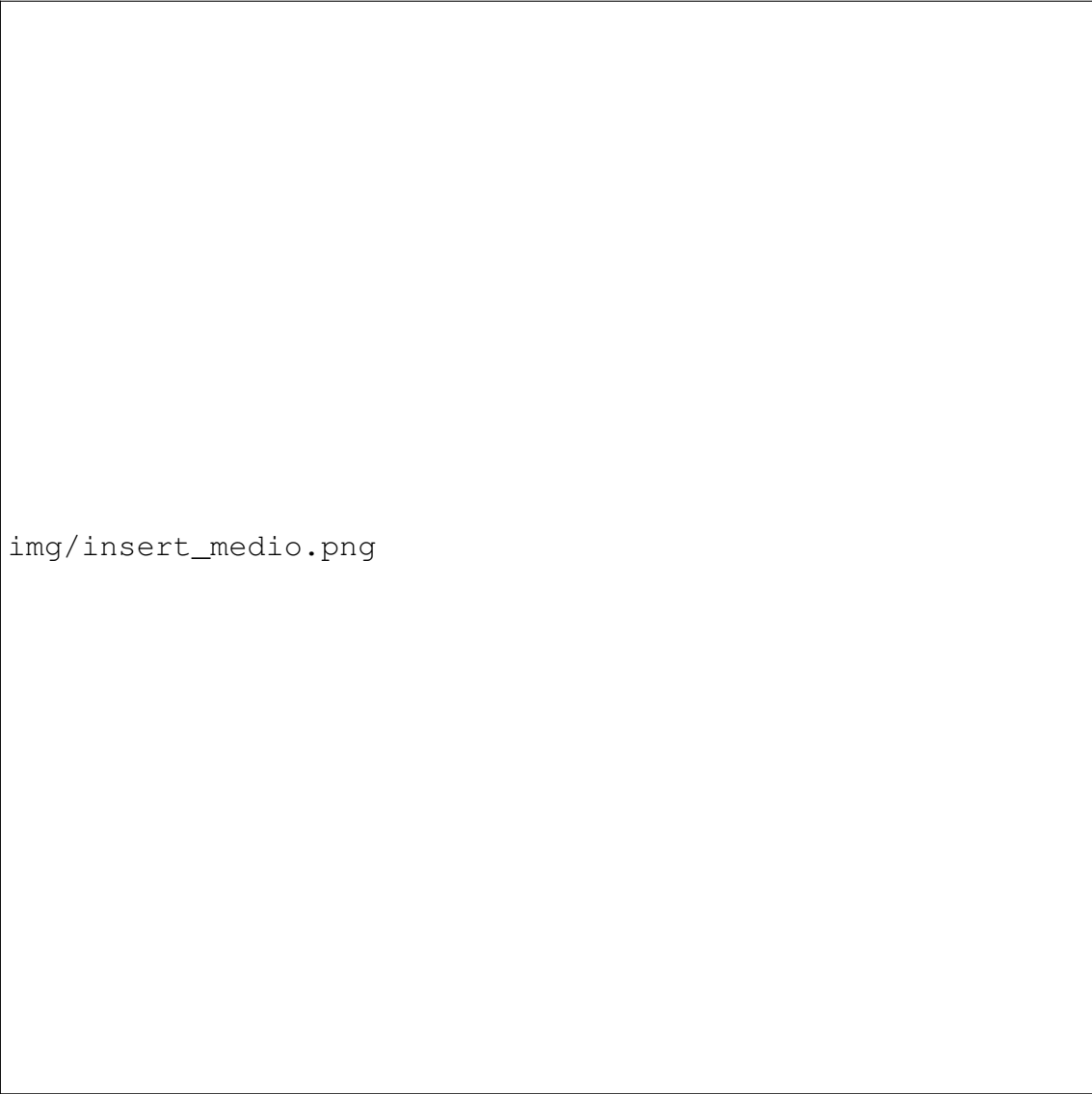
A complexidade espacial para ambas as estruturas de dados, `ArvoreBinariaDeBusca` e `ArvoreAVL`, é $O(n)$, pois ambas precisam armazenar todos os n nós. A `ArvoreAVL`, com seu balanceamento automático, garante que todas as operações principais (inserção, busca e remoção) mantenham uma complexidade de $O(\log n)$, enquanto a `ArvoreBinariaDeBusca` pode degradar para $O(n)$ no pior caso se não estiver balanceada.

4. Resultados e Discussões

Nesta seção, são apresentados os resultados das execuções das operações de inserção, busca e remoção nas Árvore Binárias de Busca (BST) e Árvore AVL, abordando o tempo de execução para cada tamanho de dados nos cenários médio e pior.

4.1. Inserção

A Figura 1 mostra o tempo de execução para a operação de inserção no caso médio para as árvores BST e AVL. Observa-se que, no caso médio, a complexidade da operação de inserção em ambas as árvores é $O(\log n)$. No entanto, a árvore AVL apresenta um tempo de execução ligeiramente maior devido ao tempo adicional necessário para manter a árvore balanceada.



img/insert_medio.png

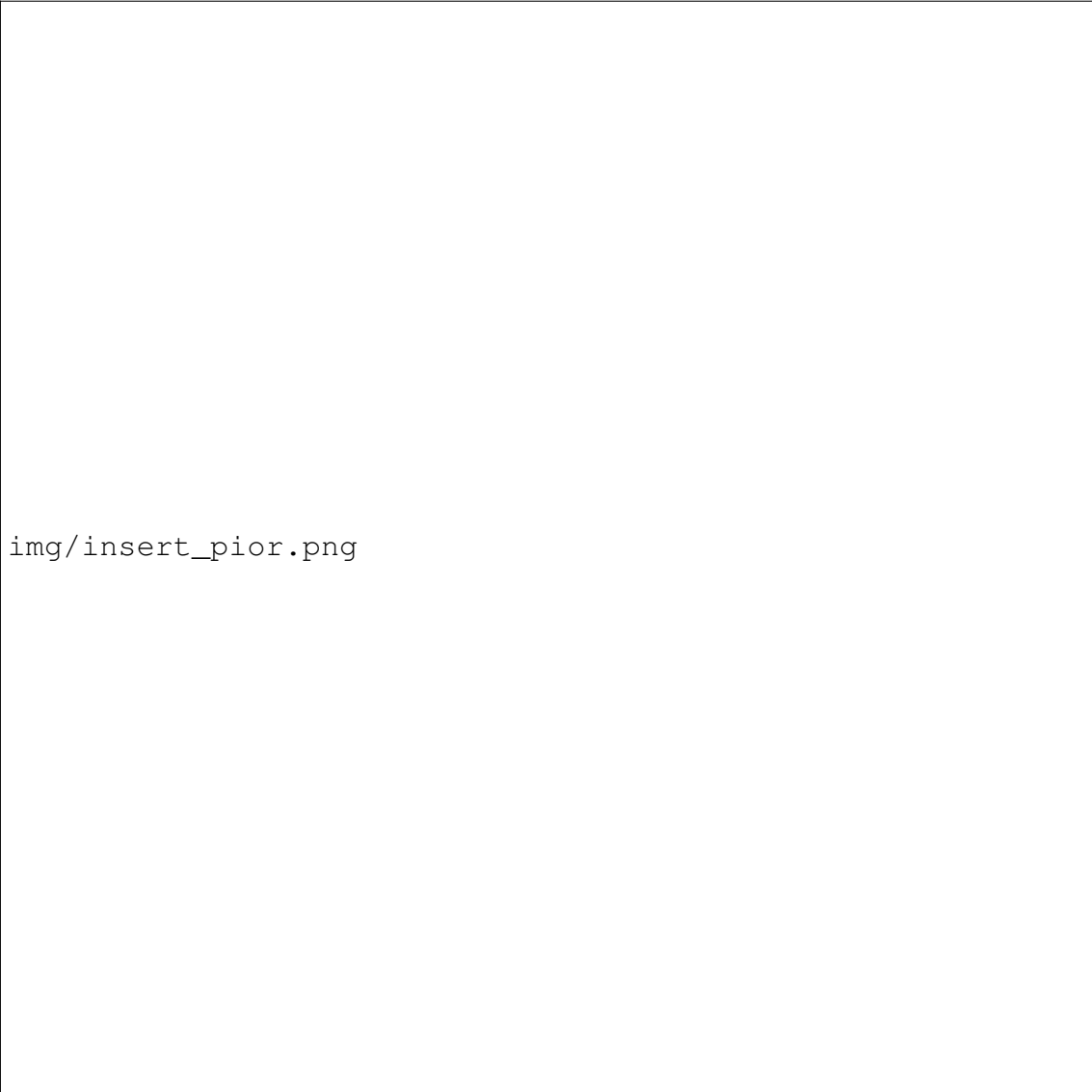
Figura 1. Tempo de execução da operação de inserção no caso médio (BST e AVL)

A Figura 2 ilustra o tempo de execução da operação de inserção no pior caso. Para a BST, o tempo de inserção aumenta drasticamente, alcançando $O(n)$ devido à árvore desbalanceada, que se assemelha a uma lista encadeada. Em contraste, a árvore AVL mantém a complexidade $O(\log n)$, mostrando sua eficiência no balanceamento automático e confirmando que a utilização de uma árvore AVL resolve o problema do pior caso da BST.

4.2. Busca

A Figura 3 apresenta o tempo de execução da operação de busca no caso médio para as árvores BST e AVL. Ambos os algoritmos mostram uma complexidade de $O(\log n)$, mas a árvore AVL possui um tempo de busca ligeiramente maior devido ao balanceamento.

Na Figura 4, o tempo de execução da operação de busca no pior caso é demonstrado. A BST apresenta uma complexidade $O(n)$ quando a árvore está desbalanceada, enquanto a árvore AVL mantém a complexidade $O(\log n)$ devido ao seu balanceamento.



img/insert_pior.png

Figura 2. Tempo de execução da operação de inserção no pior caso (BST e AVL)

automático, demonstrando sua eficiência em situações extremas.

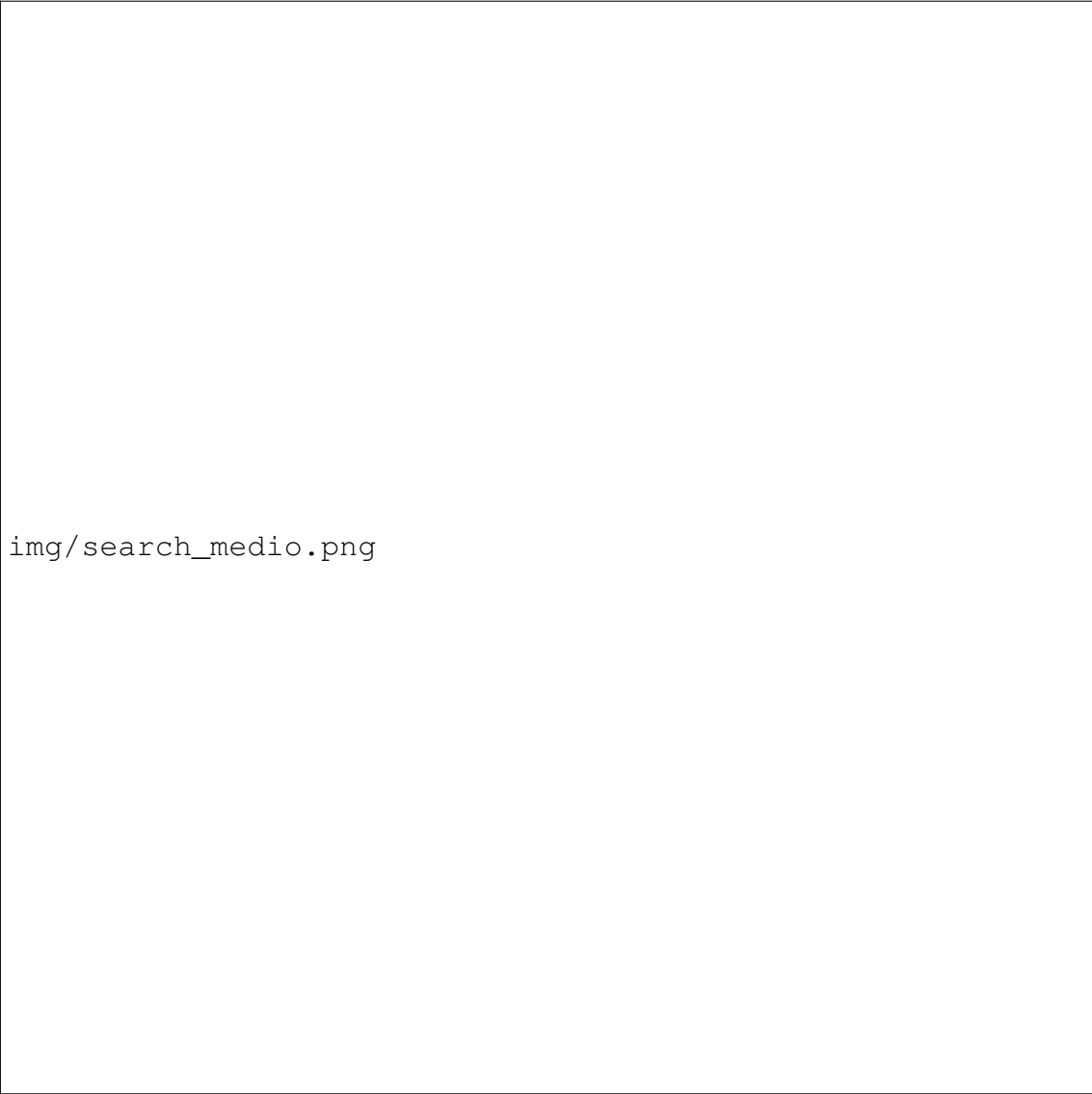
4.3. Remoção

A Figura 5 ilustra o tempo de execução para a operação de remoção no caso médio. Ambos os algoritmos possuem uma complexidade de $O(\log n)$, com a árvore AVL apresentando um tempo ligeiramente maior devido ao balanceamento.

A Figura 6 mostra o tempo de execução da operação de remoção no pior caso. A BST apresenta uma complexidade $O(n)$, enquanto a árvore AVL mantém uma complexidade $O(\log n)$ graças ao seu balanceamento automático, novamente destacando a eficiência da árvore AVL em manter um desempenho consistente.

4.4. Discussão

Os resultados obtidos confirmam a teoria de que a árvore AVL oferece um desempenho mais consistente em comparação à BST, especialmente no pior caso. Enquanto a BST



img/search_medio.png

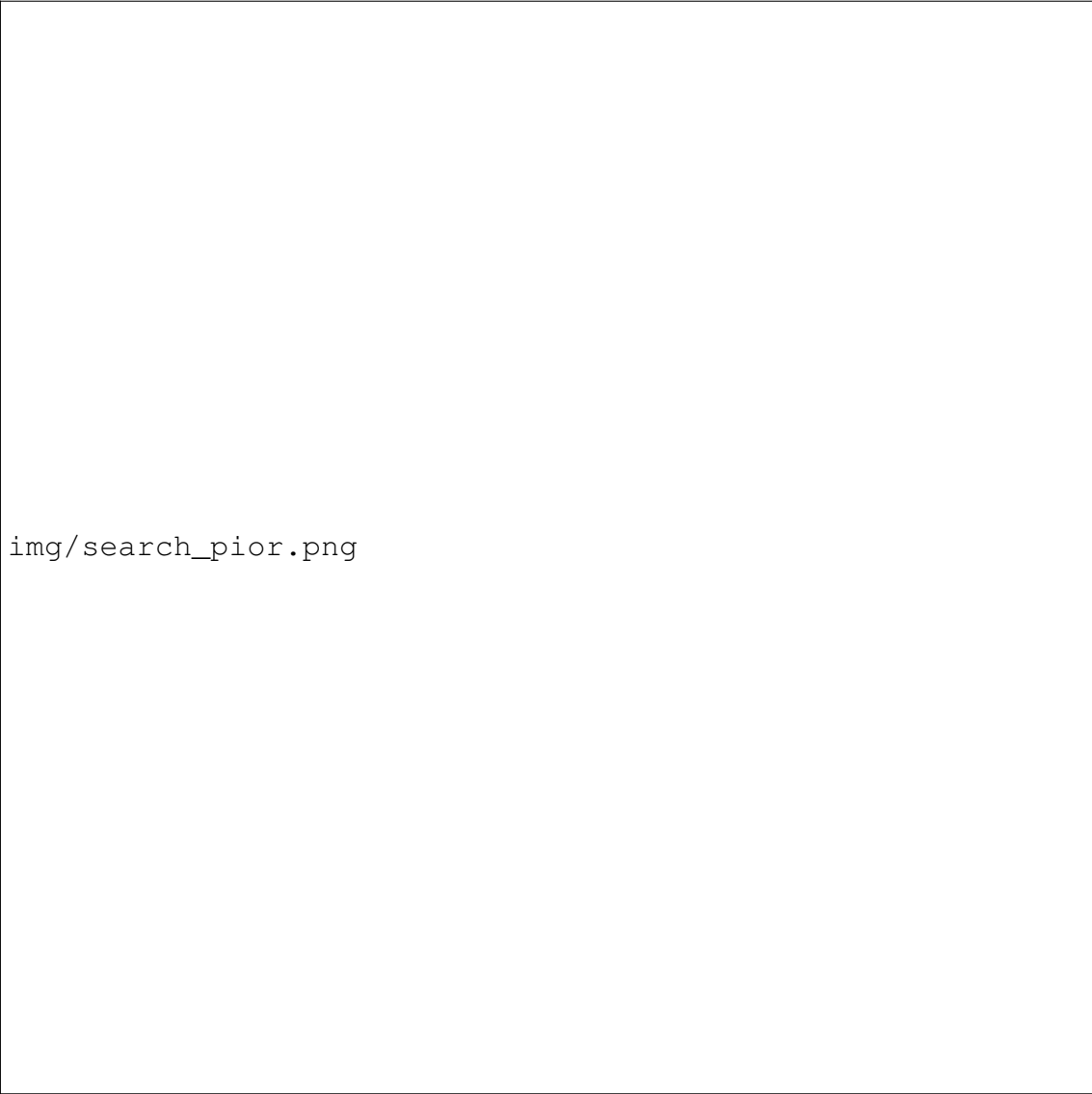
Figura 3. Tempo de execução da operação de busca no caso médio (BST e AVL)

pode sofrer degradação para uma complexidade linear $O(n)$ quando desbalanceada, a AVL mantém todas as operações principais com complexidade $O(\log n)$ devido ao balanceamento automático.

A utilização de uma árvore AVL soluciona os problemas de desempenho enfrentados pela BST no pior caso. Como mostrado nos gráficos, a AVL apresenta tempos de execução muito mais baixos em situações extremas, enquanto a BST sofre um aumento exponencial no tempo de execução conforme o tamanho dos dados aumenta.

Conforme descrito no site *Codedeposit*⁴, o balanceamento automático da árvore AVL garante que a árvore permaneça eficiente, independentemente da ordem de inserção ou remoção dos elementos. Este estudo corrobora essa afirmação, demonstrando que a árvore AVL é uma escolha robusta para cenários onde o pior caso da BST pode ocorrer.

⁴<https://codedeposit.wordpress.com/2015/10/07/red-black-vs-avl/>



img/search_pior.png

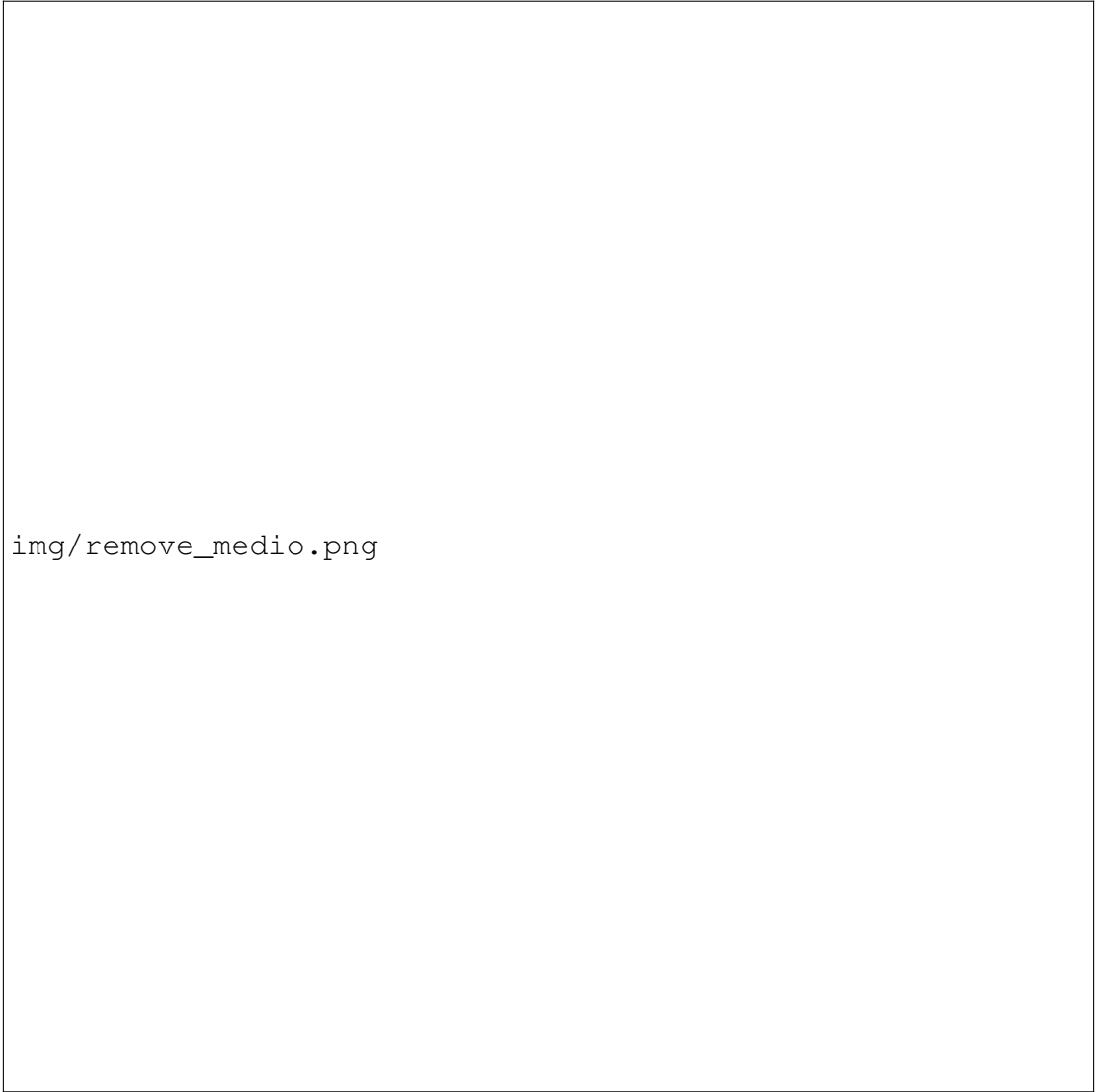
Figura 4. Tempo de execução da operação de busca no pior caso (BST e AVL)

Em resumo, a análise comparativa das árvores BST e AVL mostrou que, embora a AVL tenha um custo adicional de balanceamento, ela oferece um desempenho superior e mais previsível em termos de tempo de execução, especialmente em casos onde a BST pode se desbalancear.

5. Conclusão

A análise de complexidade de Árvores Binárias de Busca demonstra a importância do balanceamento para manter a eficiência das operações fundamentais. Os testes confirmam que, enquanto a complexidade média é $O(\log n)$, cenários desbalanceados podem levar a complexidades de $O(n)$. Recomenda-se a utilização de árvores balanceadas em aplicações que exigem alta performance consistente.

A metodologia adotada permitiu uma avaliação detalhada da complexidade dos algoritmos relacionados a árvores binárias. A combinação de abordagens empíricas e




img/remove_medio.png

Figura 5. Tempo de execução da operação de remoção no caso médio (BST e AVL)

teóricas, juntamente com o uso de ferramentas de visualização e análise de dados, forneceu uma compreensão abrangente do desempenho das operações em diferentes estruturas de árvore. Os resultados contribuem para a escolha de estruturas de dados eficientes, auxiliando no desenvolvimento de soluções computacionais mais rápidas e eficazes.

Referências

- [Bournez et al. 2020] Bournez, O., Dowek, G., Gilleron, R., Grigorieff, S., Marion, J., Perdrix, S., and Tison, S. (2020). Theoretical computer science: Computational complexity. *A Guided Tour of Artificial Intelligence Research*.
- [CORMEN et al. 2012] CORMEN, T. H., LEISESON, C. E., RIVEST, R. L., and STEIN, C. (2012). Algoritmos teoria e pratica. tradução arlete simille marques.



img/remove_pior.png

Figura 6. Tempo de execução da operação de remoção no pior caso (BST e AVL)

- [Edelweiss and Galante 2009] Edelweiss, N. and Galante, R. (2009). *Estruturas de Dados: Volume 18*. Bookman Editora.
- [Goodrich and Tamassia 2013] Goodrich, M. T. and Tamassia, R. (2013). *Estruturas de dados & algoritmos em Java*. Bookman Editora.
- [Jindal et al. 2010] Jindal, P., Kumar, A., and Kumar, S. (2010). Analysis of time complexity in binary search tree. In *Proceedings of the 4th National Conference; INDIACom*.
- [Lin 2019] Lin, A. (2019). Binary search algorithm. *WikiJournal of Science*, 2(1):1–13.
- [Mala and Ali 2022] Mala, F. A. and Ali, R. (2022). The big-o of mathematics and computer science. *Journal of Applied Mathematics and Computation*.
- [Neapolitan and Naimipour 2004] Neapolitan, R. E. and Naimipour, K. (2004). *Foundations of algorithms using C++ pseudocode*. Jones & Bartlett Learning.

- [Rubinstein-Salzedo 2018] Rubinstein-Salzedo, S. (2018). Big o notation and algorithm efficiency. In *Cryptography*, pages 75–83. Springer.
- [Tarek 2007] Tarek, A. (2007). A new paradigm for the computational complexity analysis of algorithms and functions. *Int. J. Appl. Math. Informat.*, 1(1):5–12.