UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE ENSINO SUPERIOR DO SERIDÓ
DEPARTAMENTO DE COMPUTAÇÃO E TECNOLOGIA
BACHARELADO EM SISTEMAS DE INFORMAÇÃO

Trabalho Avaliativo para disciplina de Estrutura de Dados.

Felipe Augusto Araújo da Cunha

Caicó - RN
Julho de 2024

Felipe Augusto Araújo da Cunha

Trabalho Avaliativo para disciplina de Estrutura de Dados.

Trabalho apresentado a disciplina Estrutura de Dados do Departamento de Computação e Tecnologia da Universidade Federal do Rio Grande do Norte como requisito parcial para a obtenção da nota da avaliação II.

Orientador(a): Prof. Dr. João Paulo de Souza Medeiros.

Caicó - RN
Julho de 2024

RESUMO

O trabalho apresentado se baseia na implementação dos algoritmos solicitados pelo professor João Paulo. Este documento inclui os códigos dos algoritmos, juntamente com comentários e explicações detalhadas sobre cada um deles. Além disso, são apresentados os gráficos gerados para cada algoritmo, seguidos por uma comparação e análise dos tempos de execução.

Palavras-chave: Questões, Limites, Cálculo, Resoluções.

ABSTRACT

The work presented is based on the implementation of the algorithms requested by Professor João Paulo. This document includes the codes of the algorithms, along with detailed comments and explanations about each of them. In addition, the graphs generated for each algorithm are presented, followed by a comparison and analysis of the execution times.

Keywords: Questions, Limits, Calculation, Resolutions.

SUMÁRIO

1 INTRODUÇÃO

Este trabalho tem como objetivo analisar e comparar diferentes algoritmos de consulta, especificamente: Busca em árvore binária, árvore balanceada (AVL) e tabela de dispersão (Hash). Os códigos dos algoritmos foram desenvolvidos utilizando a linguagem de programação C, escolhida por sua eficiência e controle de baixo nível, o que facilita a análise de desempenho.

Para a visualização dos resultados, foram gerados gráficos utilizando a ferramenta Gnuplot. Esses gráficos mostram as estimativas práticas dos tempos de execução de cada algoritmo.

Além da geração dos gráficos, este trabalho também inclui uma análise do tempo de execução de cada algoritmo, bem como a comparação entre alguns deles. Por fim, um relatório detalhado foi preparado, consolidando todas as observações e conclusões obtidas durante o estudo.

2 RESOLUÇÃO DOS CÓDIGOS

2.1 MELHOR CASO $O(1)$

No melhor caso, as operações (busca, inserção e remoção) nas três estruturas de dados são realizadas em tempo constante, $O(1)$. Isso acontece sob a condição de encontrar de cara o elemento na primeira posição que é verificada.

Os casos particularmente:

- **Árvore Binária de Busca (BST):**No melhor caso, o elemento que você deseja buscar, inserir ou remover está na raiz da árvore. Como a raiz é o primeiro nó verificado, a operação é realizada em tempo constante, sem a necessidade de percorrer outros nós.
- **Árvore AVL:** Semelhante à BST, o melhor caso na árvore AVL ocorre quando o elemento está na raiz. Graças ao balanceamento automático da árvore, o elemento pode ser encontrado, inserido ou removido na raiz em tempo constante, sem precisar navegar por outras subárvores.
- **Tabela Hash:** O melhor caso para uma tabela hash é quando a função hash distribui as chaves de maneira uniforme, sem colisões. Nesse cenário, ao realizar uma busca, inserção ou remoção, o elemento é acessado diretamente na posição calculada pela função hash, resultando em tempo constante, $O(1)$.

2.2 ÁRVORE BINÁRIA DE BUSCA (BST) - CASO MÉDIO

A busca em uma Árvore Binária de Busca (BST) é um algoritmo que explora a propriedade fundamental da BST, onde todos os nós à esquerda de um nó possuem valores menores, e todos os nós à direita possuem valores maiores. Isso permite que a busca seja realizada de maneira eficiente ao reduzir o espaço de busca pela metade a cada passo.

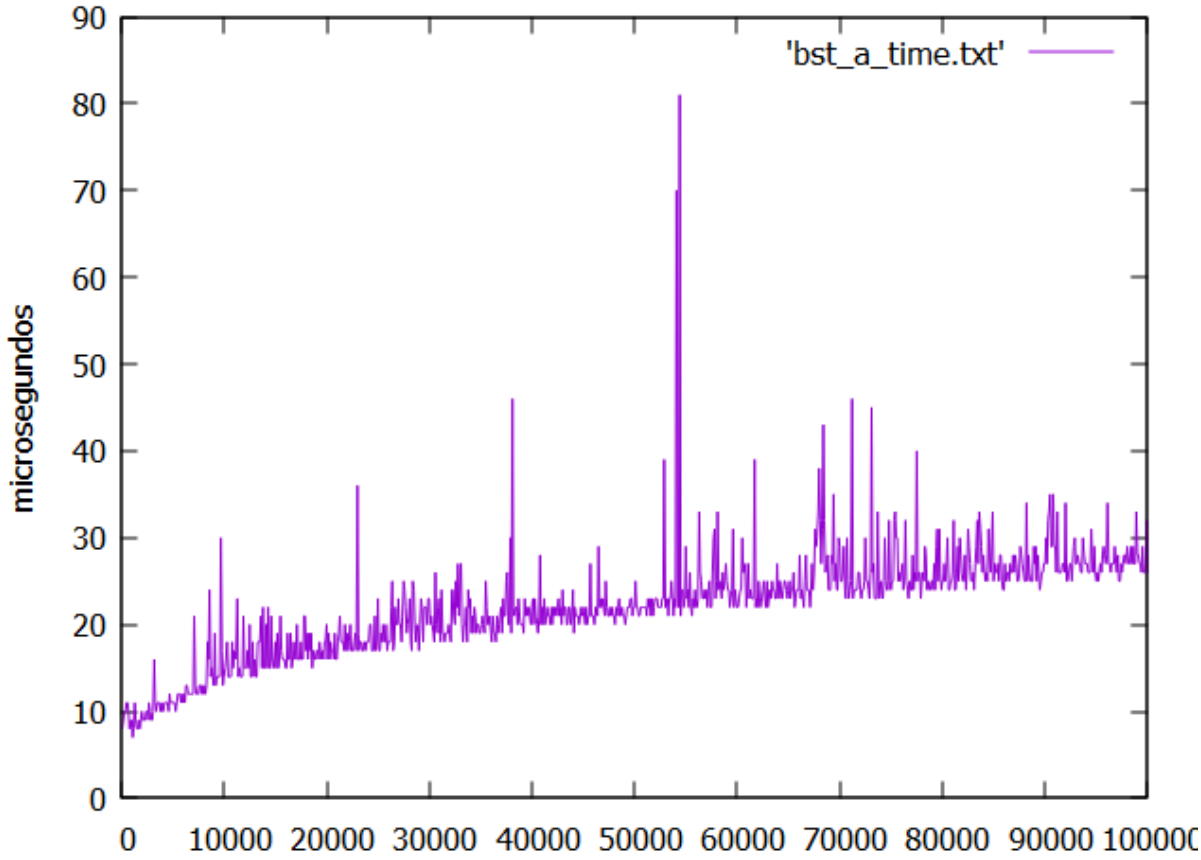

Figura 1 – Gráfico de tempo de execução para busca em BST

ANÁLISE

O algoritmo de busca em uma BST opera comparando o valor buscado com o valor do nó atual e, em seguida, decide qual subárvore deve ser explorada.

O processo de busca em uma BST funciona da seguinte maneira:

- Comparação com a Raiz: A busca começa na raiz da árvore. O valor buscado é comparado com o valor do nó atual.
- **Decisão de Direção:** Se o valor buscado for menor que o valor do nó atual, a busca continua na subárvore esquerda. Se for maior, continua na subárvore direita.

- **Recursão ou Iteração:** Esse processo de comparação e decisão continua recursivamente ou iterativamente, descendo pela árvore, até que o valor seja encontrado ou até que um nó folha seja alcançado (indicando que o valor não está presente).

- **Complexidade de Tempo:** No caso médio, a complexidade da busca é $O(\log n)$, pois a altura da árvore em uma BST razoavelmente balanceada é logarítmica em relação ao número de nós.

- **Eficiência Espacial:** A busca em uma BST requer espaço adicional mínimo, basicamente constante, para armazenar as informações necessárias para a recursão ou para a iteração.

A busca em uma BST é eficiente para conjuntos de dados onde os elementos são inseridos de forma a manter a árvore balanceada. Em cenários ideais, ela permite localizar elementos rapidamente, o que a torna uma escolha popular para operações de busca em estruturas de dados hierárquicas.

2.3 ÁRVORE BINÁRIA DE BUSCA (BST) - PIOR CASO

A busca em uma Árvore Binária de Busca (BST) no pior caso ocorre quando a árvore está completamente desbalanceada, assumindo a forma de uma lista encadeada. Isso acontece quando os elementos são inseridos em ordem crescente ou decrescente, resultando em uma altura da árvore igual ao número de nós, $O(n)$. Neste cenário, o algoritmo de busca perde a eficiência típica de uma BST e precisa percorrer todos os nós até encontrar o valor desejado ou concluir que ele não está presente.

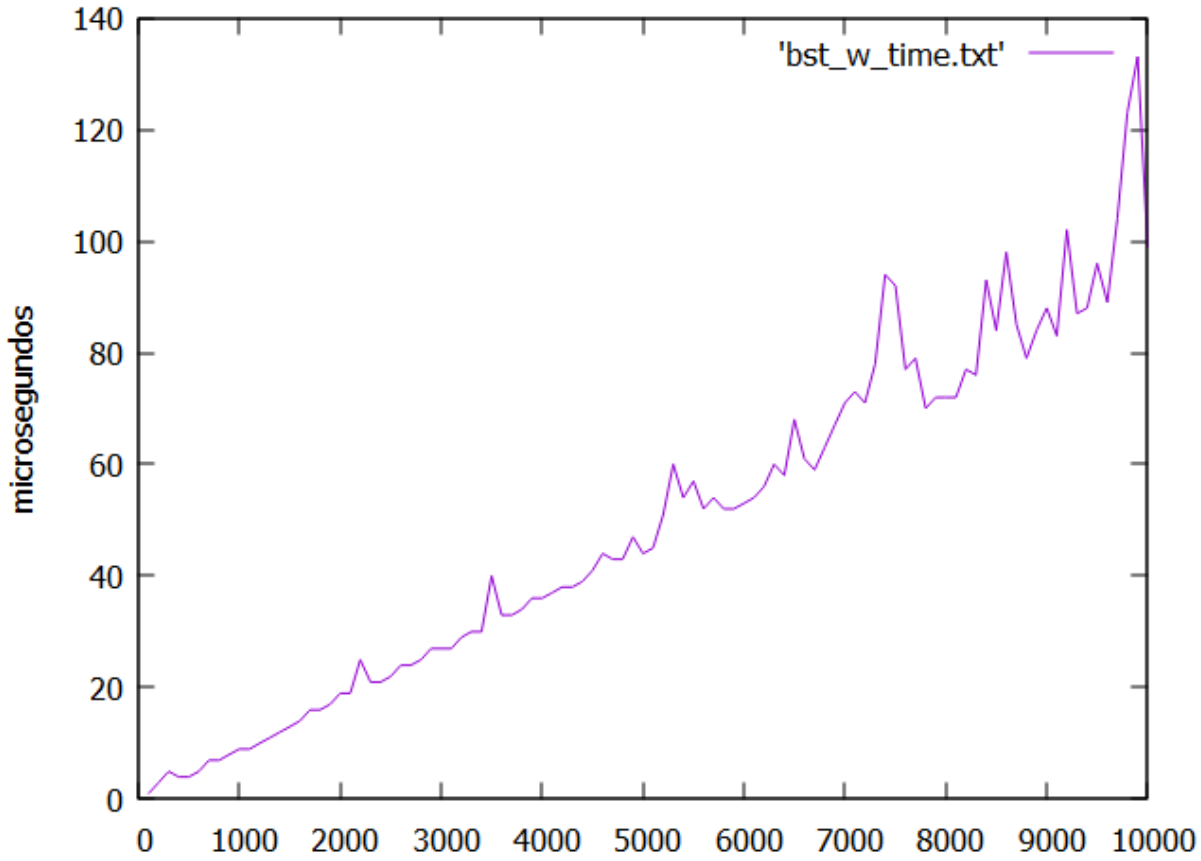

Figura 2 – Gráfico de tempo de execução para busca em BST no pior caso

ANÁLISE

No pior caso, a busca em uma BST se assemelha a uma busca sequencial em uma lista, o que compromete o desempenho esperado.

O processo de busca no pior caso funciona da seguinte maneira:

- **Percurso Linear:** Como a árvore está desbalanceada, se o valor não estiver na raiz, a busca continua em apenas uma subárvore (ou à esquerda ou à direita), de forma linear.

- **Recursão ou Iteração:** O processo continua recursivamente ou iterativamente, percorrendo cada nó sequencialmente, de cima para baixo, até encontrar o valor ou alcançar o último nó da "lista encadeada"formada pelos nós da árvore.
- **Complexidade de Tempo:** No pior caso, a complexidade da busca é $O(n)$, pois é necessário percorrer todos os nós da árvore, o que equivale a uma busca linear.
- **Eficiência Espacial:** Mesmo no pior caso, a busca em uma BST continua a utilizar espaço constante para a recursão ou iteração, mas o tempo de execução se torna o fator limitante.

No pior caso, a busca em uma BST perde a principal vantagem de eficiência logarítmica, tornando-se ineficiente, especialmente em grandes conjuntos de dados. Para evitar esse cenário, técnicas de balanceamento, como aquelas usadas em Árvores AVL ou Red-Black Trees, são frequentemente empregadas.

2.4 ÁRVORE AVL - CASO MÉDIO

A busca em uma Árvore AVL no caso médio é eficiente devido ao balanceamento automático da árvore. As Árvores AVL mantêm-se balanceadas após cada inserção ou remoção, garantindo que a altura da árvore seja logarítmica em relação ao número de nós, n. Isso permite que as operações de busca sejam realizadas de forma consistente em tempo $O(n \log n)$, independentemente da ordem de inserção dos elementos.

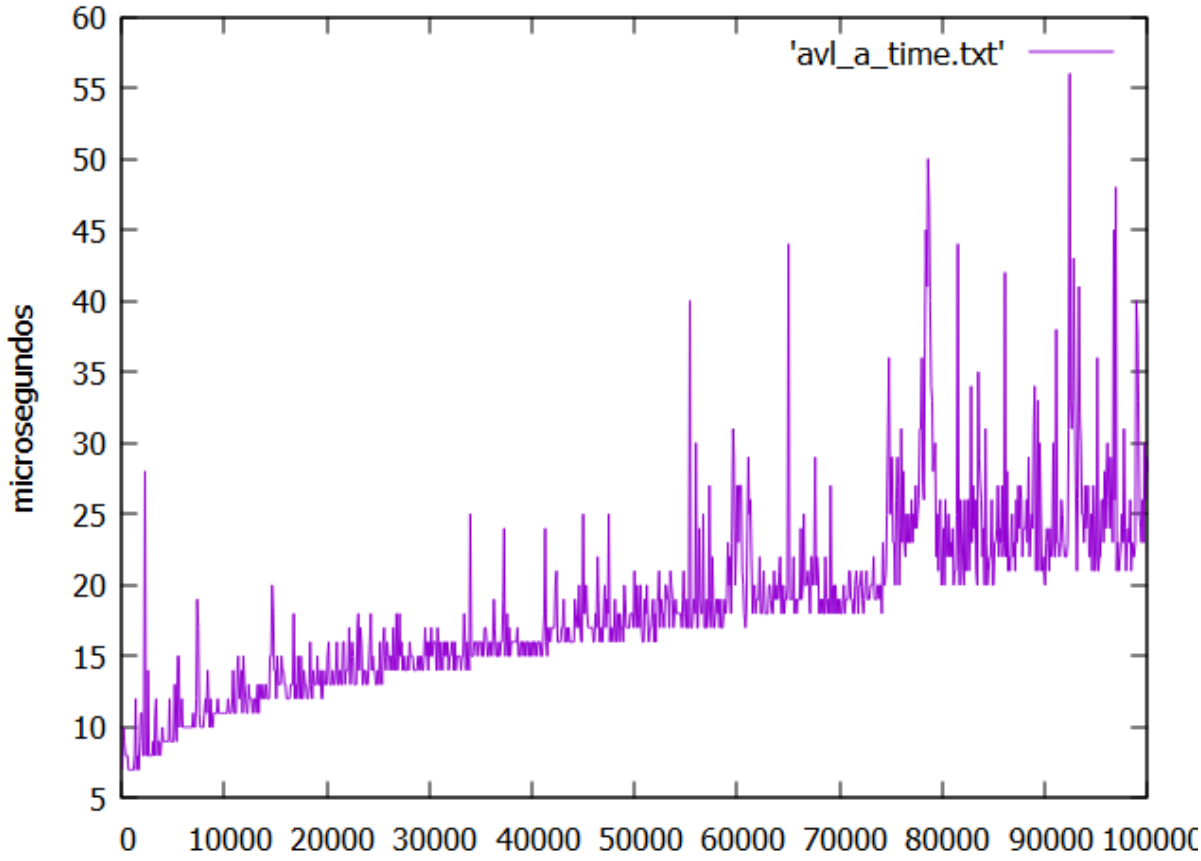

Figura 3 – Gráfico de tempo de execução para busca em AVL no caso médio

ANÁLISE

O algoritmo de busca em uma árvore AVL no caso médio mantém sua eficiência por causa do balanceamento rigoroso que a árvore mantém após cada modificação.

O processo de busca na árvore AVL funciona da seguinte maneira:

- Comparação com a Raiz: A busca começa na raiz, onde o valor a ser encontrado é comparado com o valor do nó atual.
- **Decisão de Direção:** Dependendo da comparação, a busca continua na subárvore esquerda (se o valor for menor) ou na subárvore direita (se o valor for maior).

- **Recursão ou Iteração:** Esse processo de comparação e decisão continua ao longo da árvore, descendo por um caminho que tem, em média, logarítmica profundidade ($O(\log n)$).
- **Complexidade de Tempo:** Graças ao balanceamento automático, a árvore AVL garante que a profundidade máxima seja mantida em $O(\log n)$, mesmo no caso médio. Isso assegura que a busca seja eficiente, independentemente do conjunto de dados.
- **Eficiência Espacial:** A busca em uma árvore AVL requer espaço adicional mínimo, principalmente para a recursão ou iteração, semelhante ao espaço necessário em outras árvores binárias.

No caso médio, a busca em uma árvore AVL é eficiente e consistente, garantindo tempos de busca rápidos devido ao balanceamento automático que minimiza a profundidade da árvore. Isso faz com que a AVL seja uma excelente escolha para cenários onde a eficiência da busca é crítica, mesmo quando os dados são inseridos em ordem não ideal.

2.5 ÁRVORE AVL - PIOR CASO

A busca em uma Árvore AVL no pior caso ainda é altamente eficiente devido à natureza balanceada da árvore. Garantindo que a diferença de altura entre as subárvores de qualquer nó seja no máximo 1. Isso significa que, mesmo no pior caso, a altura da árvore é mantida em $O(\log n)$, assegurando que a busca permaneça rápida e eficiente.

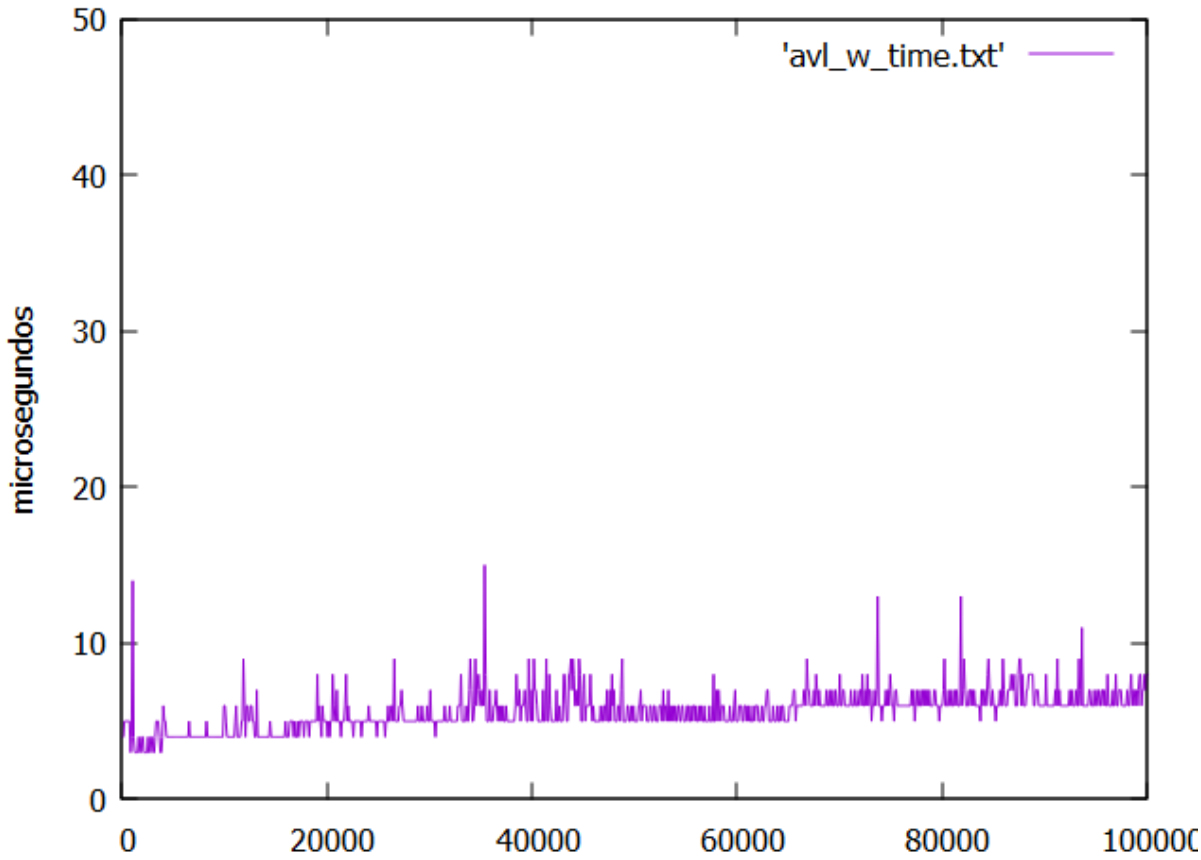

Figura 4 – Gráfico de tempo de execução para busca em AVL no pior caso

ANÁLISE

Mesmo no pior cenário possível, que é o elemento buscado não estar na árvore, a Árvore AVL garante que a busca seja realizada de maneira eficiente, preservando sua estrutura balanceada.

O processo de busca na árvore AVL no pior caso funciona da seguinte maneira:

- Comparação com a Raiz: A busca começa na raiz, onde o valor a ser encontrado é comparado com o valor do nó atual.
- **Decisão de Direção:** Baseado na comparação, a busca continua na subárvore esquerda ou direita, dependendo se o valor buscado é menor ou maior que o valor do

nó atual.

- **Recursão ou Iteração:** Esse processo continua até que a busca atinja um nó folha, e o elemento não seja encontrado, ou seja, não está presente na árvore. Mesmo no pior caso, a profundidade máxima que a busca precisa percorrer é logarítimo.
- **Complexidade de Tempo:** A árvore AVL garante que a complexidade da busca permaneça $O(\log n)$ no pior caso, porque o balanceamento rigoroso impede que a árvore se torne desbalanceada, diferentemente de uma árvore binária de busca comum (BST).
- **Eficiência Espacial:** Assim como no caso médio, a busca em uma árvore AVL no pior caso requer espaço constante para a execução da recursão ou iteração.

No pior caso, a busca em uma árvore AVL continua a ser altamente eficiente, graças ao seu balanceamento automático, que previne cenários onde a busca se tornaria linear. Isso faz com que a AVL seja uma escolha robusta para operações de busca, garantindo consistência de desempenho independentemente da ordem de inserção dos dados.

2.6 TABELA HASH - CASO MÉDIO

A busca em uma tabela hash no caso médio é extremamente eficiente, com uma complexidade de tempo $O(1)$. As tabelas hash utilizam uma função de dispersão (hash function) para mapear chaves a posições específicas em uma tabela. No caso médio, a função de hash distribui as chaves uniformemente, minimizando colisões e permitindo o acesso direto ao elemento desejado.

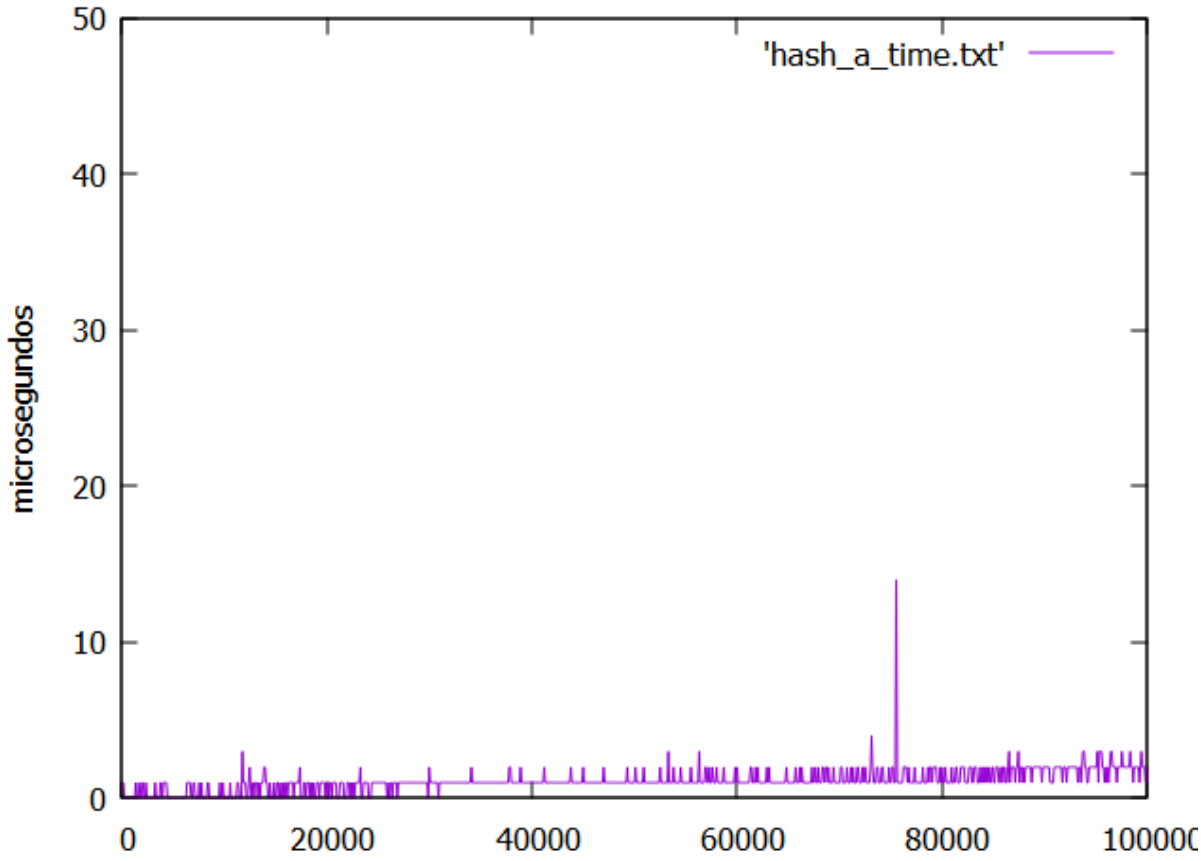

Figura 5 – Gráfico de tempo de execução para busca em Tabela Hash no caso médio

ANÁLISE

O algoritmo de busca em uma tabela hash no caso médio é simples e extremamente rápido devido à eficiência da função de hash em distribuir uniformemente as chaves.

O processo de busca em uma tabela hash no caso médio funciona da seguinte maneira:

- Cálculo do Índice: A chave do item a ser buscado é passada através de uma função de hash, que calcula um índice na tabela onde o item deve estar armazenado.

- **Acesso Direto:** O índice calculado é utilizado para acessar diretamente a posição na tabela, onde o item pode ser encontrado.
- **Verificação de Colisão:** No caso médio, as colisões (quando duas chaves diferentes geram o mesmo índice) são minimizadas pela função de hash. Se ocorrer uma colisão, o algoritmo verifica a próxima posição usando um método de resolução de colisão (como encadeamento ou sondagem).
- **Complexidade de Tempo:** No caso médio, a complexidade da busca é $O(1)$, pois o acesso ao elemento ocorre diretamente no índice calculado pela função de hash, com colisões sendo raras.
- **Eficiência Espacial:** As tabelas hash são eficientes em termos de espaço, embora o uso de espaço possa aumentar para evitar colisões. Um bom ajuste da função de hash e do tamanho da tabela (evitando altas taxas de ocupação) garante que o caso médio continue eficiente.

No caso médio, a busca em uma tabela hash é extremamente eficiente e rápida, tornando-a uma escolha popular para operações de busca onde a velocidade é crucial e o espaço adicional é disponível para garantir uma baixa taxa de colisões.

2.7 TABELA HASH - PIOR CASO

A busca em uma tabela hash no pior caso pode ser ineficiente, com uma complexidade de tempo $O(n)$. Esse cenário ocorre quando a função de hash não distribui as chaves uniformemente, resultando em um grande número de colisões, onde várias chaves diferentes são mapeadas para o mesmo índice ou para um pequeno conjunto de índices. Isso transforma a busca em uma operação linear, semelhante à busca em uma lista encadeada.

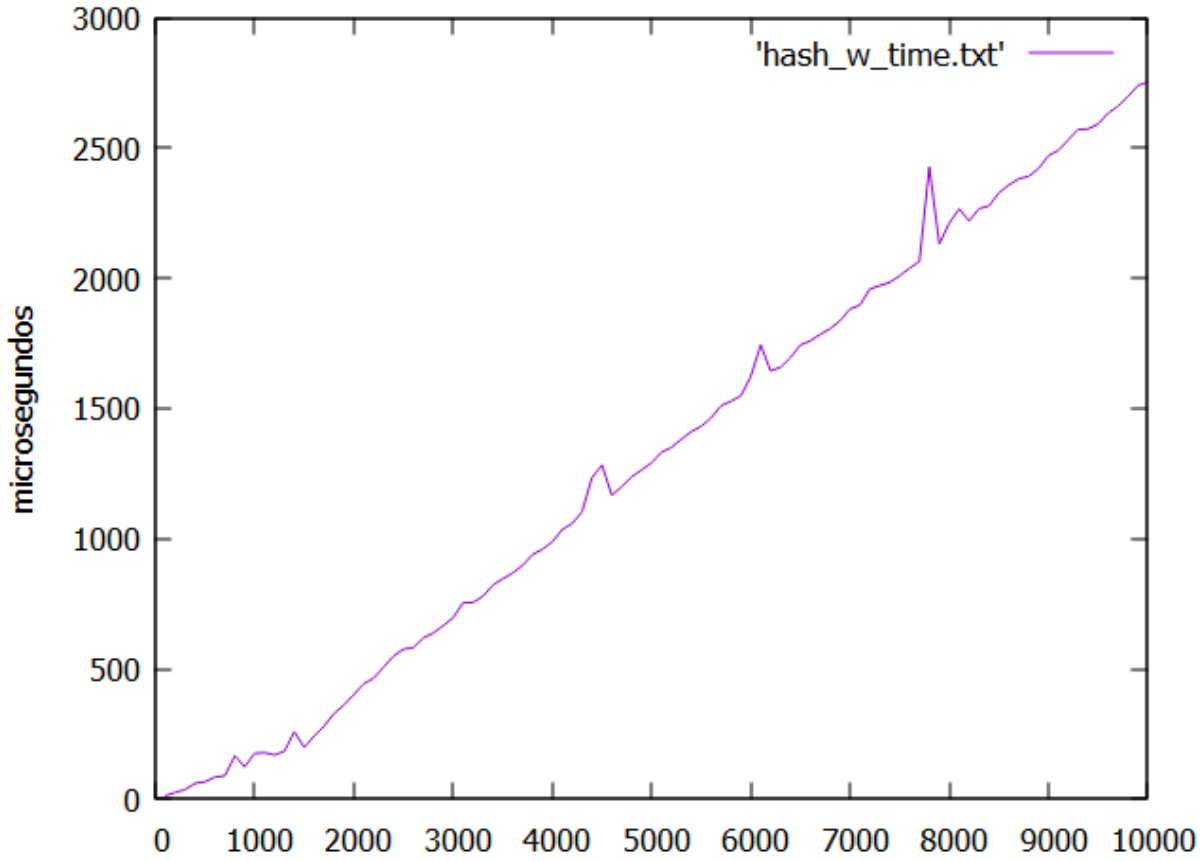

Figura 6 – Gráfico de tempo de execução para busca em Tabela Hash no pior caso

ANÁLISE

No pior caso, a eficiência da tabela hash é severamente comprometida, e a busca pode se tornar muito lenta.

O processo de busca em uma tabela hash no pior caso funciona da seguinte maneira:

- Cálculo do Índice: A chave do item a ser buscado é passada através de uma função de hash, que calcula um índice na tabela onde o item deveria estar armazenado.
- **Resolução de Colisão:** Devido a uma má distribuição das chaves pela função de hash, várias chaves podem ser mapeadas para o mesmo índice. Quando isso ocorre,

um método de resolução de colisão, como encadeamento (listas ligadas) ou sondagem linear, é utilizado.

- **Busca Sequencial:** No pior caso, todas as chaves são mapeadas para o mesmo índice, criando uma lista encadeada de elementos. A busca então se degrada para uma busca linear, onde cada elemento na lista encadeada precisa ser verificado até que o valor desejado seja encontrado ou todos os elementos sejam percorridos.

- **Complexidade de Tempo:** No pior caso, a complexidade da busca se torna $O(n)$, pois é necessário percorrer uma lista de tamanho n, onde n é o número de elementos inseridos na tabela.

- **Eficiência Espacial:** Embora as tabelas hash sejam geralmente eficientes em termos de espaço, no pior caso, o espaço adicional necessário para armazenar listas encadeadas pode aumentar, especialmente se a função de hash for ineficaz.

No pior caso, a busca em uma tabela hash perde sua principal vantagem de tempo constante $O(1)$, tornando-se tão ineficiente quanto uma busca linear em uma lista não ordenada. Para mitigar esse cenário, é crucial escolher uma função de hash apropriada e ajustar o tamanho da tabela para minimizar colisões, como visto nas aulas, podendo-se utilizar de funções rehash e resize para evitar colisões, assim evitando que o algorítimo se torne linear.

2.8 COMPARAÇÃO DE LINEARES

Nesta seção, serão comparados os casos de complexidade $O(n)$: Pior caso da Árvore Binária (BST) e pior caso da Tabela Hash. O gráfico abaixo mostra uma severa distinção entre os casos apesar de ambos serem lineares. O tempo de execução em relação ao tamanho do vetor.

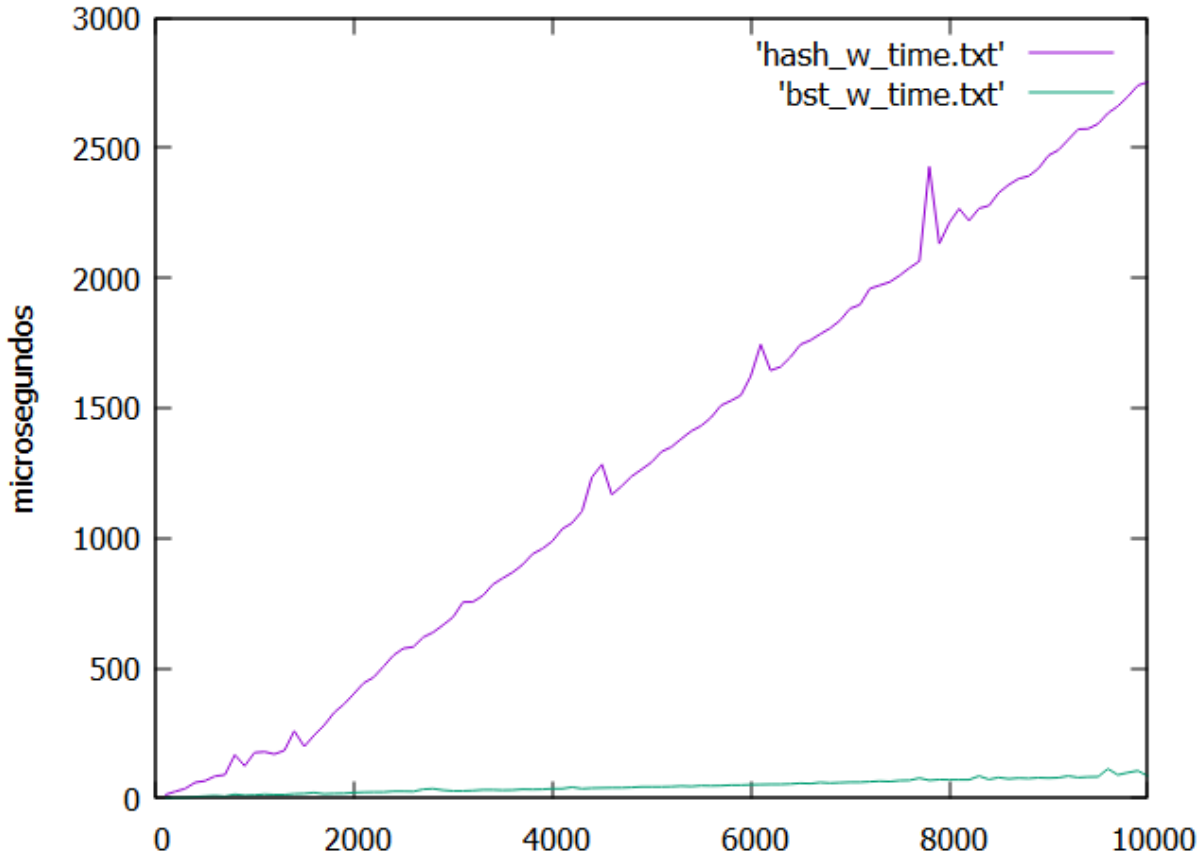

Figura 7 – Comparação de desempenho entre os casos lineares

A comparação entre busca linear, o pior caso de uma tabela hash, e o pior caso de uma árvore binária de busca (BST) destaca "diferentes desempenhos"conforme o tamanho dos dados aumenta:

- Tabela Hash (Pior Caso): Idealmente, busca em tempo constante O(1), mas no pior caso, devido a colisões e má distribuição dos elementos, o tempo pode se aproximar de $O(n)$. Se a tabela hash não for bem dimensionada ou a função hash não for eficaz, o desempenho pode se deteriorar drasticamente.
- BST (Pior Caso): Se a BST estiver desbalanceada, seu desempenho pode ser $O(n)$, semelhante à busca linear. Porém, em uma árvore balanceada ou moderadamente

desbalanceada, o tempo de execução é $O(logn)$, mais eficiente que a busca linear no geral.

O gráfico mostra que, à medida que o vetor cresce, o tempo de execução da tabela hash é muito pior do que o da BST. Isso sugere que a tabela hash sofre de problemas como alta colisão ou má distribuição de elementos, fazendo com que seu desempenho seja inferior ao da BST, que consegue manter uma organização dos dados que a favorece, mesmo no pior caso.

2.9 COMPARAÇÃO DE LOGARÍTIMOS

Nesta seção, serão comparados os casos de complexidade $O(logn)$: Caso médio caso da Árvore Binária (BST), caso médio da Árvore AVL e pior caso da Árvore AVL. O gráfico abaixo mostra o tempo de execução em relação ao tamanho do vetor.

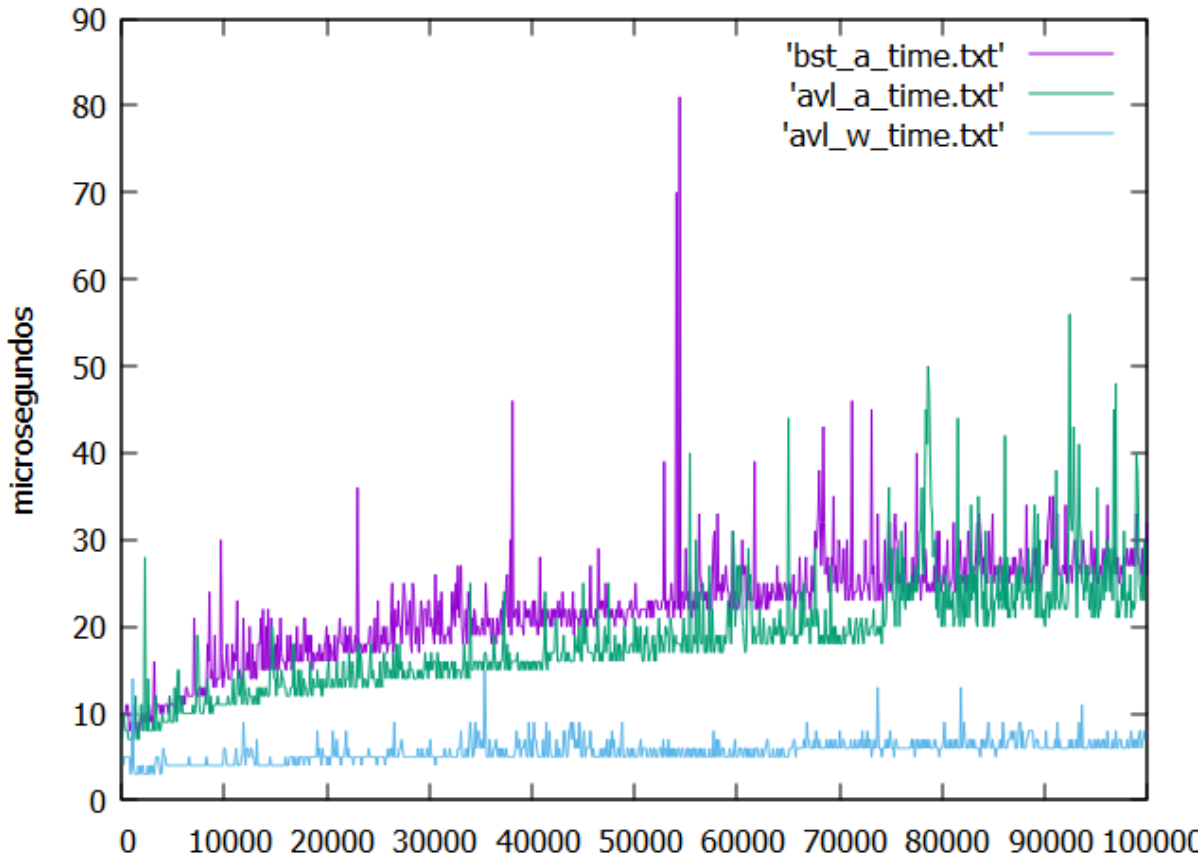

Figura 8 – Comparação de desempenho entre os casos logarítimos

Apesar de um pouco feio, o gráfico mostra que a árvore AVL apresenta tempos de busca mais consistentes e eficientes tanto no caso médio quanto no pior caso, sempre em $O(logn)$. Em contraste, a BST pode ter desempenho logarítmico no caso médio, mas é suscetível a se desbalancear, resultando em um tempo de execução potencialmente muito pior ($O(n)$) no pior caso. Assim, o gráfico destaca a AVL como a estrutura com o desempenho mais estável e rápido, enquanto a BST pode apresentar variação significativa dependendo do balanceamento da árvore.

Além disso, é notável que o pior caso da Árvore AVL está melhor do que o caso médio da própria. Isso ocorre devido problemas técnicos relacionado a máquina e sistema operacional em que o código foi rodado. O gráfico ideal seria o pior caso da Árvore AVL apresentar o crescimento logarítimo entre os dois casos médios.

3 CONCLUSÃO

Na análise dos códigos de busca para Árvore Binária de Busca (BST), Árvore AVL e Tabela Hash, cada estrutura de dados oferece vantagens e desvantagens específicas em termos de desempenho e complexidade:

Árvore Binária de Busca (BST): O código de busca em uma BST é relativamente simples e segue uma abordagem recursiva ou iterativa para localizar um nó. No entanto, o desempenho da busca depende fortemente do balanceamento da árvore. No caso médio, a busca é eficiente, com tempo O(log n), mas no pior caso, quando a árvore está desbalanceada, o desempenho degrada-se para O(n).

Árvore AVL: A busca em uma árvore AVL, embora semelhante ao código de busca em uma BST, é mais eficiente no geral, pois a AVL é autobalanceada. Isso significa que, mesmo no pior caso, o tempo de busca permanece O(log n). O código de inserção e manutenção da árvore AVL é mais complexo devido às rotações necessárias para manter o balanceamento, mas o código de busca se beneficia diretamente desse balanceamento automático, garantindo desempenho consistente.

Tabela Hash: A busca em uma tabela hash é muito eficiente no caso ideal, com tempo O(1). No entanto, o código de busca pode se complicar no pior caso, onde colisões frequentes resultam em listas encadeadas longas dentro dos buckets. Se a tabela hash não for bem implementada (com uma função hash ineficiente ou sem redimensionamento adequado), o desempenho pode se aproximar de O(n), tornando-a menos eficiente que a AVL e potencialmente até que uma BST.

Cada uma dessas estruturas de dados tem seu lugar dependendo do contexto de uso. A BST é simples e eficaz quando os dados são inseridos aleatoriamente, mas pode sofrer de problemas de desempenho quando desbalanceada. A AVL garante um desempenho de busca consistente, embora com uma implementação mais complexa. Já a tabela hash é ideal para buscas rápidas, mas exige cuidado na escolha da função hash e no gerenciamento de colisões. A escolha da estrutura deve, portanto, considerar tanto as características dos dados quanto as exigências de desempenho do sistema.