Universidade Federal do Rio Grande do Norte
Centro de Ensino Superior do Seridó
Departmento de Computação e Tecnologia
Curso de Bacharelado em Sistemas de Informação

Trabalho de Estrutura de Dados

Nathan Lopes Rodrigues

Caicó-RN

14/07/2025

Nathan Lopes Rodrigues

Relatório 1

Trabalho apresentado para a disciplina de Estrutura de Dados (ED) do Departamento de Computação e Tecnologia (DCT) da Universidade Federal do Rio Grande do Norte (UFRN) como requisito para obtenção parcial de obtenção de nota.

Orientador:
Prof. Dr. João Paulo de Souza Medeiros

Caicó-RN

14/07/2025

2° Trabalho

Autor: Nathan Lopes Rodrigues
Orientador(a): Prof. Dr. João Paulo de Souza Medeiros

Resumo

O presente trabalho tem como objetivo realizar uma avaliação prática e comparativa do desempenho de diferentes algoritmos de busca, utilizando como base três estruturas de dados: a árvore binária de busca (BST), a árvore AVL (balanceada) e a tabela de dispersão (Hash Table). A proposta consiste em medir e confrontar o tempo de execução de cada estrutura ao realizar operações de busca com conjuntos de dados de tamanhos variados, permitindo observar como cada técnica se comporta diante de volumes crescentes de informação.

Para isso, foram implementados programas em linguagem C, escolhida por sua eficiência e proximidade com a arquitetura de hardware, garantindo medições mais realistas de tempo de processamento. Os testes foram realizados em ambiente controlado, considerando conjuntos de 10.000, 50.000 e 100.000 elementos, com dados gerados pseudo-aleatoriamente para simular cenários práticos. Os tempos de execução foram coletados, processados e organizados em arquivos de saída, sendo posteriormente representados em gráficos utilizando a ferramenta Gnuplot, amplamente empregada na visualização de resultados experimentais.

A análise contempla não apenas a performance observada, mas também aspectos teóricos que justificam o comportamento de cada estrutura, como a influência do balanceamento automático na árvore AVL e o tempo de acesso praticamente constante da tabela Hash. É importante destacar que fatores externos, como condição de hardware e processos em segundo plano, podem impactar as medições, mas esforços foram feitos para manter o ambiente de teste o mais estável possível.

Palavras-chave: Busca; Estruturas de Dados; Árvore Binária; Árvore AVL; Tabela Hash; Desempenho.

Abstract

This paper aims to carry out a practical evaluation and comparison of the performance of different search algorithms, using three data structures as a basis: the binary search tree (BST), the AVL tree (self-balancing) and the hash table. The proposal consists of measuring and comparing the execution time of each structure when performing search operations with datasets of varying sizes, allowing the observation of how each technique behaves as the volume of information grows.

For this purpose, programs were implemented in the C programming language, chosen for its efficiency and closeness to hardware architecture, ensuring more realistic processing time measurements. The tests were carried out in a controlled environment, using datasets of 10,000, 50,000 and 100,000 elements, with pseudo-randomly generated data to simulate practical scenarios. Execution times were collected, processed and organized into output files, and then graphically represented using the Gnuplot tool, widely used for visualizing experimental results.

The analysis covers not only the observed performance, but also theoretical aspects that justify the behavior of each structure, such as the influence of automatic balancing in the AVL tree and the practically constant access time of the hash table. It is important to highlight that external factors, such as hardware conditions and background processes, may impact the measurements, but efforts were made to keep the test environment as stable as possible.

Keywords: Search; Data Structures; Binary Tree; AVL Tree; Hash Table; Performance.

Sumário

1 Introdução

1.1 Introdução

A necessidade de manipular grandes volumes de dados é uma realidade em diversas áreas da computação, como bancos de dados, sistemas de informação, ciência de dados e aplicações de tempo real. Para atender a essa demanda, é fundamental dispor de estruturas de dados que sejam capazes de realizar operações de inserção e busca de forma rápida e eficiente, garantindo desempenho satisfatório mesmo com o aumento exponencial do volume de informações.

Neste trabalho, propõe-se uma avaliação prática e comparativa de três estruturas de dados amplamente utilizadas: a Árvore Binária de Busca (*Binary Search Tree* - BST), a Árvore AVL (versão balanceada da BST) e a Tabela de Dispersão (*Hash Table*). Cada uma dessas estruturas possui características próprias em termos de organização interna e complexidade das operações, impactando diretamente o tempo necessário para localizar ou inserir elementos.

A escolha da linguagem C para a implementação dos algoritmos justifica-se por sua proximidade com a arquitetura de hardware e sua alta eficiência na execução de operações de baixo nível, possibilitando medições de tempo mais precisas e representativas do comportamento real dos algoritmos. Além disso, a linguagem C permite maior controle sobre recursos como alocação dinâmica de memória, essencial para o gerenciamento das estruturas analisadas.

Os experimentos foram conduzidos utilizando conjuntos de dados gerados de forma pseudoaleatória, contemplando diferentes tamanhos de entrada para simular cenários práticos de utilização. Para garantir confiabilidade estatística, cada configuração foi repetida múltiplas vezes, permitindo calcular médias que minimizam variações pontuais. Por fim, os resultados foram organizados em arquivos no formato `.csv` e visualizados por meio de gráficos gerados no Gnuplot, ferramenta amplamente empregada para análise e apresen-

tação de dados experimentais.

Dessa forma, espera-se oferecer uma análise abrangente sobre o desempenho de cada estrutura de dados, fornecendo subsídios teóricos e práticos que embasem decisões de projeto em aplicações que exigem alto desempenho na manipulação de informações.

2 Análises dos dados

2.1 Árvore Binária de Busca

Binary Search Tree - BST é uma estrutura de dados hierárquica amplamente utilizada para organizar informações de forma a permitir operações de inserção, remoção e busca de elementos de forma relativamente eficiente. Sua principal característica é a propriedade de ordenação: para qualquer nó na árvore, todos os valores contidos na sua subárvore esquerda são menores que o valor do nó, enquanto todos os valores na subárvore direita são maiores ou iguais.

Essa estrutura recursiva possibilita que, ao buscar um elemento, o algoritmo descarte metade da árvore a cada comparação, desde que a árvore permaneça balanceada. Assim, em um cenário ideal — onde a árvore se encontra perfeitamente balanceada — a altura da BST é proporcional a $\log n$, sendo n o número de nós, o que garante uma complexidade de tempo média para operações de busca, inserção e remoção de $O(\log n)$.

No entanto, a BST tem como limitação principal a ausência de mecanismos automáticos de balanceamento. Caso os dados sejam inseridos em ordem crescente ou decrescente, por exemplo, a árvore degenera em uma lista encadeada simples. Nessa configuração degenerada, a altura da árvore torna-se n e, consequentemente, o tempo de execução das operações básicas passa a ter complexidade $O(n)$. Esse comportamento representa o pior caso para uma BST, afetando drasticamente o desempenho em aplicações que lidam com grandes volumes de dados sem pré-processamento ou ordenação aleatória.

Apesar dessa vulnerabilidade, a BST é frequentemente escolhida por sua implementação relativamente simples, boa eficiência em dados aleatoriamente distribuídos e por ser uma base conceitual para entender estruturas mais avançadas, como a árvore AVL e a árvore rubro-negra. Seu funcionamento intuitivo facilita o ensino de conceitos de recursão, ordenação e análise de complexidade em disciplinas de algoritmos e estruturas de dados.

2.2 Árvore AVL

Árvore AVL é uma extensão da árvore binária de busca tradicional, introduzida por Adelson-Velsky e Landis em 1962, com o objetivo de resolver uma das principais limitações da BST: a tendência ao desbalanceamento quando os dados inseridos seguem uma ordem específica. Para garantir uma altura ideal, a árvore AVL implementa um mecanismo de balanceamento automático após cada operação de inserção ou remoção.

O balanceamento é realizado monitorando o *fator de balanceamento* (*balance factor*), definido como a diferença entre as alturas das subárvores esquerda e direita de cada nó. Para que a propriedade de balanceamento seja mantida, essa diferença deve permanecer entre -1 e 1 em todos os nós. Quando essa restrição é violada, a árvore aplica operações de rotação para restaurar o equilíbrio.

Existem quatro situações possíveis que requerem rotações: rotação simples à esquerda, rotação simples à direita, rotação dupla à esquerda-direita e rotação dupla à direita-esquerda. As rotações simples corrigem desbalanceamentos em forma de linha reta, enquanto as rotações duplas são necessárias quando o desbalanceamento ocorre em forma de "L". Essas rotações reorganizam os ponteiros entre os nós sem alterar a ordenação lógica dos elementos, garantindo que a árvore permaneça uma BST válida.

Graças a esse mecanismo de autoajuste, a árvore AVL assegura que sua altura permaneça sempre proporcional a $\log n$, mesmo após um grande número de inserções e remoções. Isso garante complexidade de tempo $O(\log n)$ para operações de busca, inserção e remoção, tornando a AVL uma alternativa muito mais eficiente que uma BST pura em casos de inserções ordenadas ou padrões de dados desfavoráveis.

Apesar de exigir um pequeno custo computacional adicional para calcular fatores de balanceamento e executar rotações quando necessário, esse custo é amplamente compensado pela melhoria significativa de desempenho na maioria dos cenários práticos, especialmente em aplicações que requerem grande número de operações dinâmicas e um tempo de resposta previsível.

2.2.1 Tabela Hash

A **Tabela Hash** é uma estrutura de dados extremamente eficiente para operações de busca, inserção e remoção, principalmente quando o objetivo é alcançar desempenho próximo a tempo constante, ou seja, complexidade média de $O(1)$. Esse ganho de eficiência

decorre da utilização de uma *função hash*, responsável por mapear cada chave de entrada para um índice em um vetor de tamanho fixo, chamado de tabela.

O princípio de funcionamento baseia-se em aplicar a função hash sobre a chave, resultando em um índice onde o dado deve ser armazenado ou procurado. No entanto, colisões — situações em que duas ou mais chaves distintas geram o mesmo índice — são inevitáveis. Para lidar com isso, um dos métodos mais comuns é o *encadeamento separado* (*chaining*), que armazena múltiplos elementos na mesma posição da tabela por meio de listas encadeadas. Assim, mesmo que ocorram colisões, todos os elementos podem ser mantidos corretamente e acessados de forma eficiente.

A eficiência da tabela hash depende diretamente da qualidade da função hash e do dimensionamento adequado da tabela. Uma função hash mal projetada pode gerar agrupamentos excessivos de chaves em poucos índices, degradando o desempenho para $O(n)$ no pior caso — quando todas as chaves caem em uma única lista encadeada. Por isso, geralmente se utiliza um tamanho de tabela proporcional ao número esperado de elementos e preferencialmente um número primo para reduzir padrões repetitivos e minimizar colisões.

Em condições ideais, quando a distribuição das chaves é uniforme e a função hash é de boa qualidade, a tabela hash garante tempo de acesso, inserção e remoção praticamente constantes, sendo largamente empregada em sistemas que requerem alto desempenho para manipulação de grandes volumes de dados, como indexação de bancos de dados, compiladores, caches e dicionários.

3 Metodologia

3.0.1 Implementações

As três estruturas de dados — Árvore Binária de Busca (BST), Árvore AVL e Tabela Hash — foram implementadas em linguagem C, escolhida por oferecer maior controle de memória, desempenho próximo do hardware e precisão na medição de tempos de execução. Cada implementação foi estruturada em módulos independentes, contendo funções específicas para inserção, busca, inicialização e liberação de memória, garantindo experimentos mais justos e comparáveis entre si.

Alguns parâmetros principais foram definidos para padronizar todos os testes:

- `MAX_N`: número máximo de elementos a serem inseridos em cada estrutura, limitando o tamanho dos conjuntos de dados.
- `STEP`: incremento de elementos a cada iteração, permitindo a coleta de medições para múltiplos tamanhos de entrada de forma gradual.
- `REPETICOES`: quantidade de vezes que cada teste é repetido para o mesmo valor de `n`. No presente trabalho, foi estabelecido o valor de 100 repetições.

A opção por executar `REPETICOES` igual a 100 tem como principal objetivo suavizar a influência de ruídos externos no ambiente de execução, como variações na carga de processamento da máquina ou flutuações do sistema operacional. Dessa forma, o tempo médio obtido para cada cenário se aproxima mais do comportamento real da estrutura avaliada, fornecendo resultados mais robustos para análise comparativa.

3.0.2 Procedimentos de Medição

Para garantir a fidelidade e consistência das medições de desempenho, o processo experimental seguiu etapas rigorosas. Os dados utilizados para inserção e busca foram gerados utilizando a função padrão `rand()` da linguagem C, que produz números pseudoaleatórios dentro de um intervalo definido. Especificamente, para cada tamanho de conjunto

`n`, os números foram gerados no intervalo de `0` até aproximadamente `10 * n`, garantindo diversidade e reduzindo colisões previsíveis.

O processo de medição envolveu duas fases principais: inserção dos elementos na estrutura e subsequente busca por elementos aleatórios. Para cada iteração, a estrutura era inicialmente populada com `n` elementos gerados aleatoriamente. Em seguida, eram realizadas múltiplas buscas, também com valores aleatórios, simulando cenários práticos de consulta.

O tempo de execução para essas operações foi medido utilizando a função `clock()`, que retorna o número de ciclos de clock consumidos pelo processo até o momento da chamada. Apesar de ser uma função amplamente utilizada para medição de tempo em C, ela possui limitações, como a resolução dependente do sistema e a possível interferência de processos paralelos.

Para mitigar tais limitações, cada teste foi repetido `REPETICOES` vezes, calculando-se o tempo médio por operação a partir da soma dos tempos dividida pelo número de repetições. Além disso, o tempo médio por busca foi normalizado pelo número de buscas realizadas em cada repetição, permitindo uma comparação justa entre diferentes tamanhos e estruturas.

3.0.3 Geração e Análise dos Gráficos

Os dados obtidos das medições foram exportados em arquivos CSV com formato padrão `n,tempo(ns)`, onde `n` representa o número de elementos e `tempo(ns)` o tempo médio em nanossegundos por operação.

Para a visualização e análise dos resultados, foi utilizada a ferramenta `Gnuplot`, reconhecida pela sua capacidade de gerar gráficos de alta qualidade e facilidade de configuração. Com scripts específicos, foram plotados gráficos que ilustram as comparações de desempenho entre as três estruturas para as operações de inserção e busca.

Cada gráfico apresenta linhas e pontos que representam as medições para cada estrutura, diferenciadas por cores e estilos de linha para facilitar a interpretação visual. Por exemplo, a cor azul foi associada à árvore AVL, vermelho à BST e verde à tabela Hash. Linhas contínuas indicam o tempo de inserção, enquanto linhas tracejadas representam o tempo de busca. As legendas foram cuidadosamente configuradas para esclarecer essas distinções, permitindo ao leitor uma compreensão rápida e precisa dos resultados exibidos.

4 Resultados e Análise

4.0.1 Gráfico Geral: Inserção e Busca

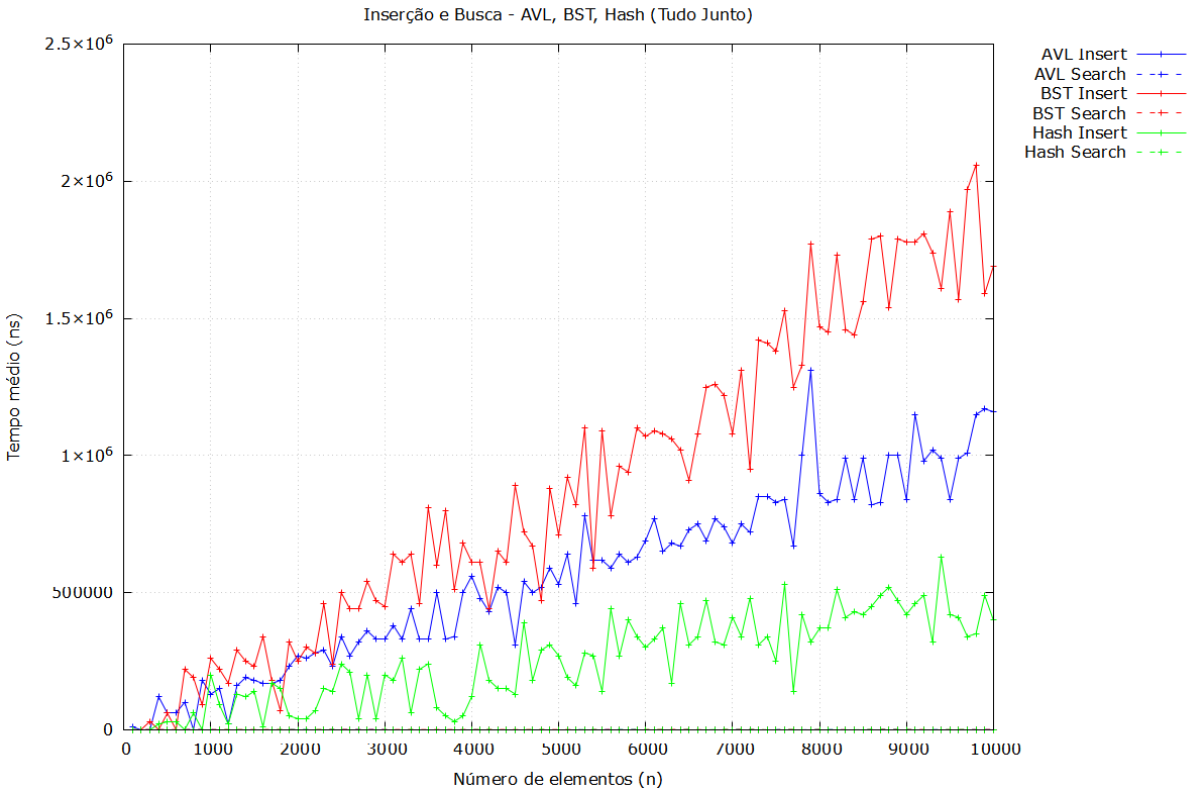

Figura 1: Comparação geral da inserção e busca para AVL, BST e Hash.

O gráfico geral apresenta a visão consolidada dos tempos médios de inserção e busca para as três estruturas avaliadas: Árvore AVL, Árvore Binária de Busca (BST) e Tabela Hash. Observa-se claramente que a Tabela Hash apresenta os menores tempos em ambas as operações, mantendo praticamente um tempo constante independentemente do tamanho do dataset. Esse comportamento confirma sua complexidade média O(1) tanto para inserção quanto para busca(será mostrado melhor mais a frente), desde que a função hash seja bem projetada e minimize colisões.

A Árvore AVL, por sua vez, demonstra tempos um pouco maiores em comparação à Hash, mas mantém uma estabilidade que reflete seu balanceamento automático, garantindo complexidade O(log n) para ambos os casos. Já a BST apresenta o pior desempenho na busca, com crescimento acentuado do tempo conforme aumenta o número de elementos, o que indica degradação significativa causada por desbalanceamento, pois em seu pior caso ela degenera em uma lista encadeada, resultando em tempo linear O(n). Essa visão geral reforça o impacto direto do balanceamento e do hashing na eficiência das operações.

4.0.2 Análise da Inserção

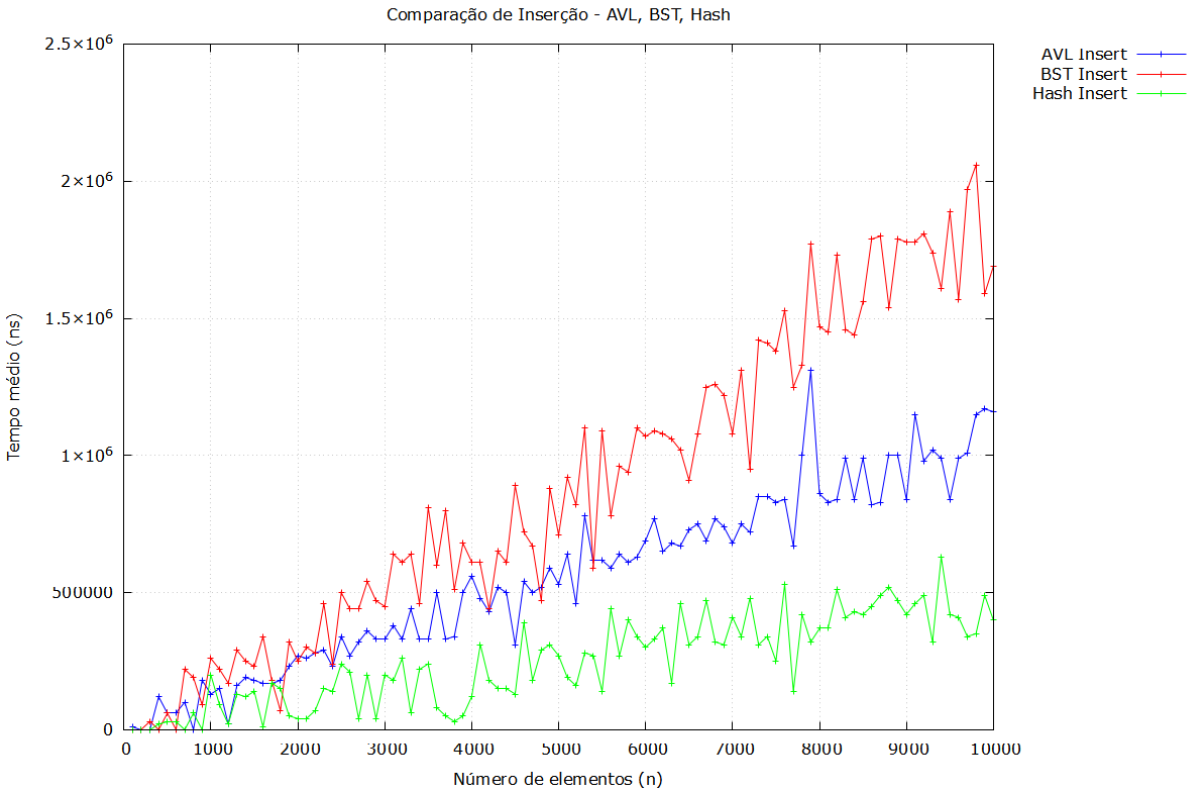

Figura 2: Comparação da inserção entre AVL, BST e Hash.

O gráfico de inserção isolada evidencia que a Tabela Hash apresenta o menor tempo em todos os tamanhos de entrada, devido à sua operação baseada diretamente no cálculo da função hash, sem necessidade de percorrer estruturas hierárquicas. A BST possui tempos relativamente baixos em conjuntos menores de dados, mas começa a apresentar crescimento mais acelerado à medida que o tamanho aumenta, novamente por conta do desbalanceamento estrutural em inserções de dados parcialmente ordenados. Já a AVL exibe um tempo de inserção maior em comparação às demais, resultado esperado devido

à necessidade de realizar rotações de balanceamento após cada inserção para manter sua altura logarítmica. Apesar desse custo adicional, esse processo garante maior eficiência nas buscas futuras.

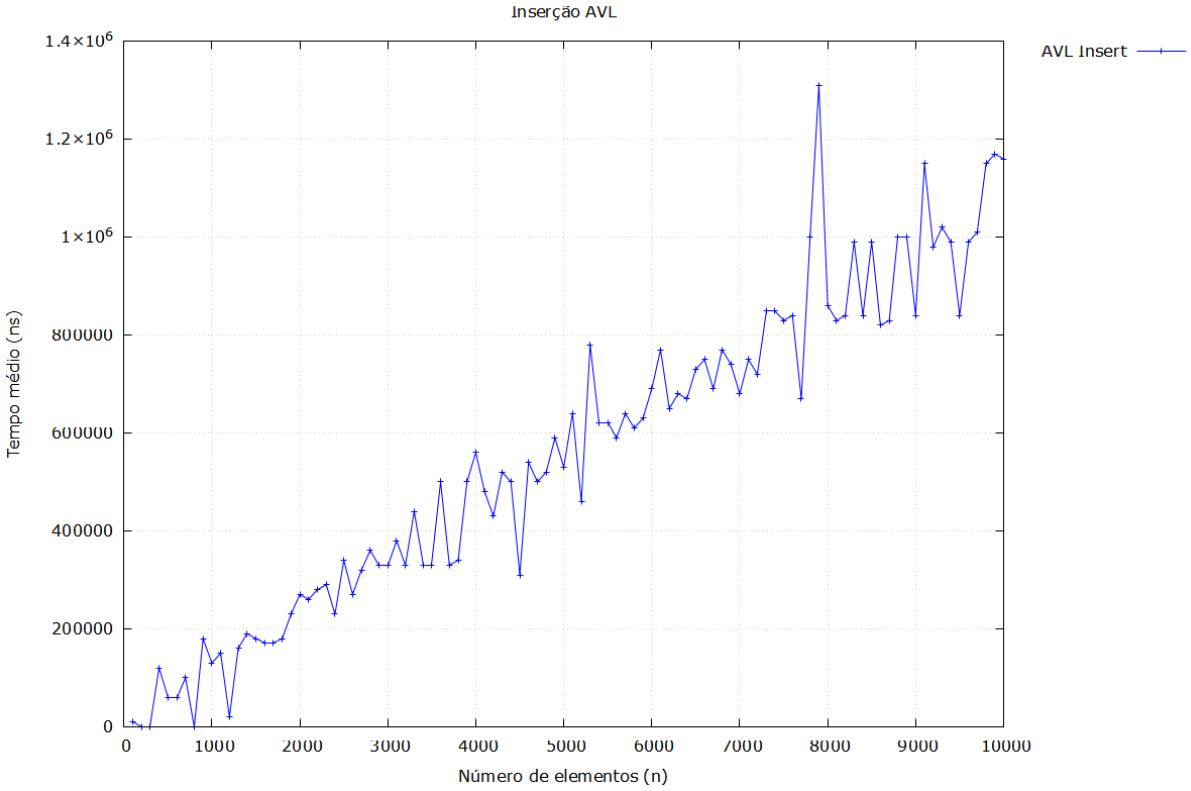

Figura 3: Inserção na Árvore AVL.

Neste gráfico específico para a Árvore AVL, observa-se um crescimento de tempo suave e estável, alinhado ao seu comportamento teórico de complexidade O(log n). Cada inserção realizada implica em possíveis cálculos de fator de balanceamento e rotações simples ou duplas, mas esses custos são compensados por manter a árvore equilibrada, garantindo eficiência consistente mesmo em datasets grandes. Essa característica torna a AVL especialmente indicada em aplicações que requerem alto volume de inserções dinâmicas com posterior necessidade de buscas rápidas e ordenadas.

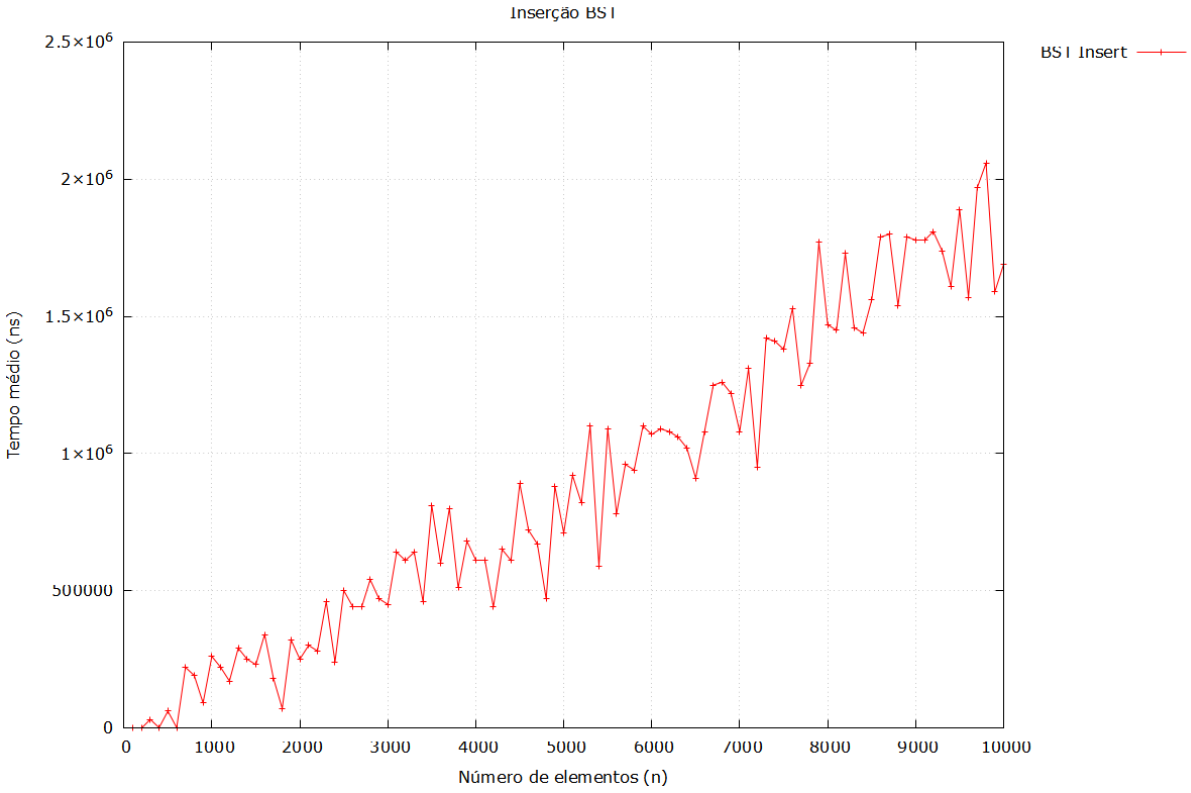

Figura 4: Inserção na Árvore BST.

O gráfico de inserção da BST demonstra um comportamento variável: em pequenos datasets, a inserção é rápida e eficiente, mas à medida que o número de elementos aumenta, principalmente em dados parcialmente ordenados, o tempo cresce de forma mais pronunciada, aproximando-se de uma curva linear. Isso ocorre porque a árvore, ao se desbalancear, passa a ter altura proporcional a n, degradando sua performance para O(n). Em situações práticas, essa limitação pode inviabilizar o uso da BST para grandes volumes sem pré-processamento ou balanceamento adicional.

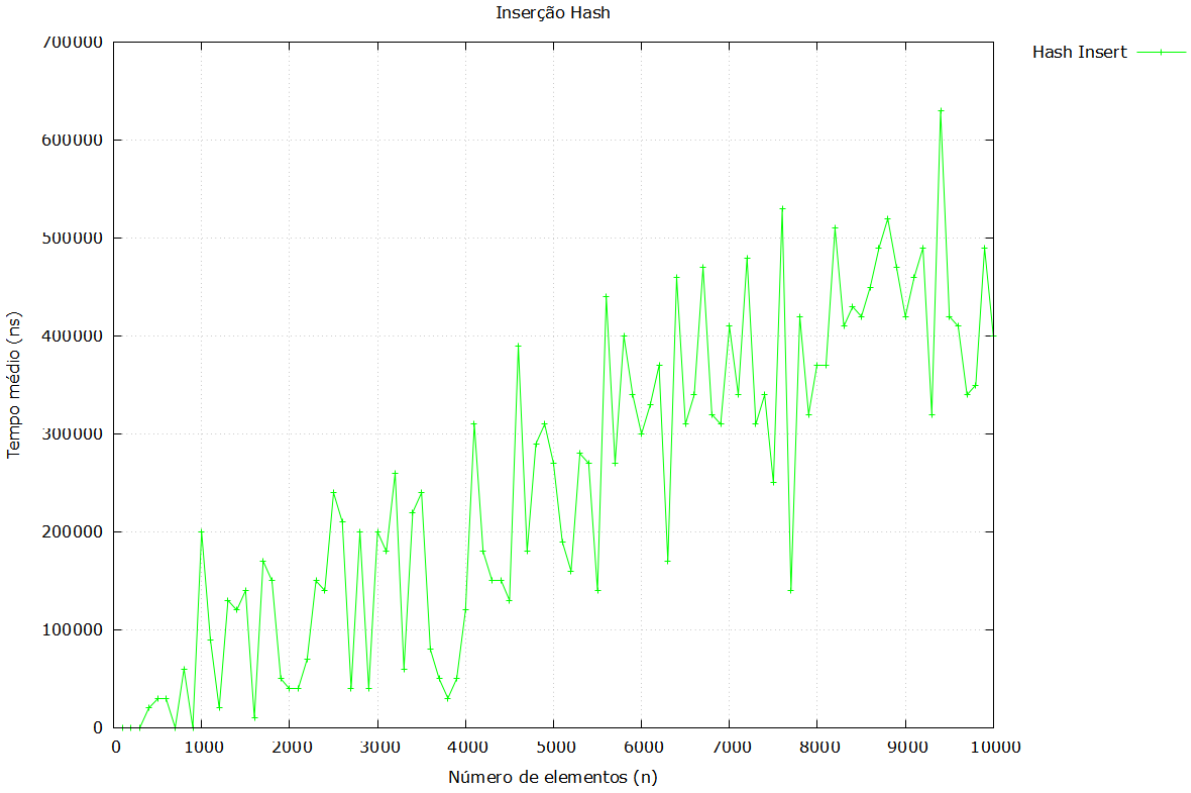

Figura 5: Inserção na Tabela Hash.

Este gráfico confirma a alta eficiência da Tabela Hash no processo de inserção. O tempo médio permanece praticamente constante mesmo com aumento expressivo no número de elementos, reforçando seu comportamento teórico de O(1) na média. Essa estrutura, portanto, se destaca como a melhor opção quando o objetivo principal é realizar inserções rápidas e sem a necessidade de ordenação, desde que sua função hash seja de qualidade e a tabela tenha tamanho adequado para reduzir colisões.

4.0.3 Análise da Busca

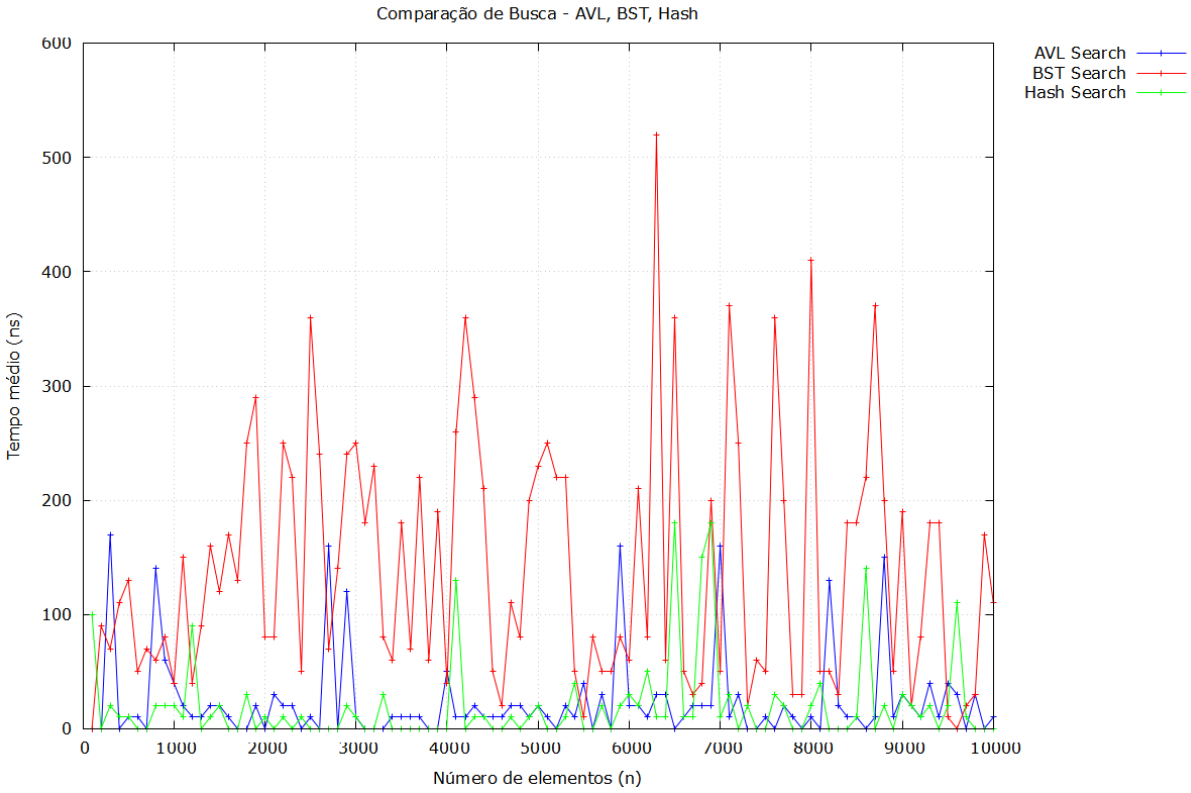

Figura 6: Comparação da busca entre AVL, BST e Hash.

Ao analisar o gráfico de busca geral, fica evidente a superioridade da Tabela Hash em relação às demais estruturas, mantendo tempos quase constantes para datasets de diferentes tamanhos. A Árvore AVL ocupa a segunda colocação em termos de eficiência, apresentando um crescimento suave e logarítmico, típico de sua estrutura balanceada. A BST, por outro lado, apresenta o pior desempenho, com aumento expressivo do tempo de busca em dados maiores, resultado direto de possíveis desbalanceamentos, que levam a buscas quase lineares no pior caso. Esse gráfico reforça a importância do balanceamento automático ou do hashing para garantir performance consistente em buscas.

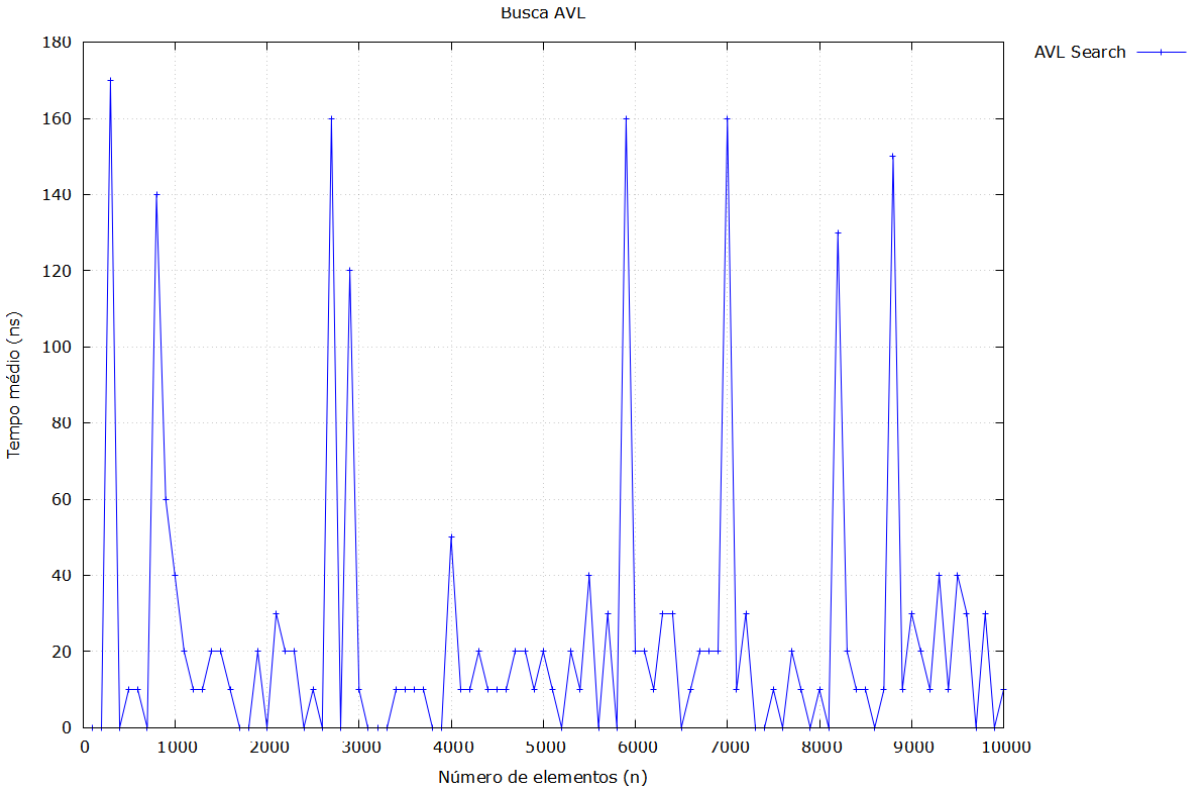

Figura 7: Busca na Árvore AVL.

A análise da busca na árvore AVL mostra que, apesar de pequenas variações, o tempo de execução se mantém baixo e com crescimento moderado conforme o número de elementos aumenta. Esse comportamento reflete sua estrutura balanceada, que garante buscas eficientes com complexidade O(log n). Por isso, a AVL é indicada em aplicações que exigem consultas rápidas em conjuntos de dados que sofrem inserções e modificações constantes, mantendo desempenho estável mesmo com grande volume de dados.

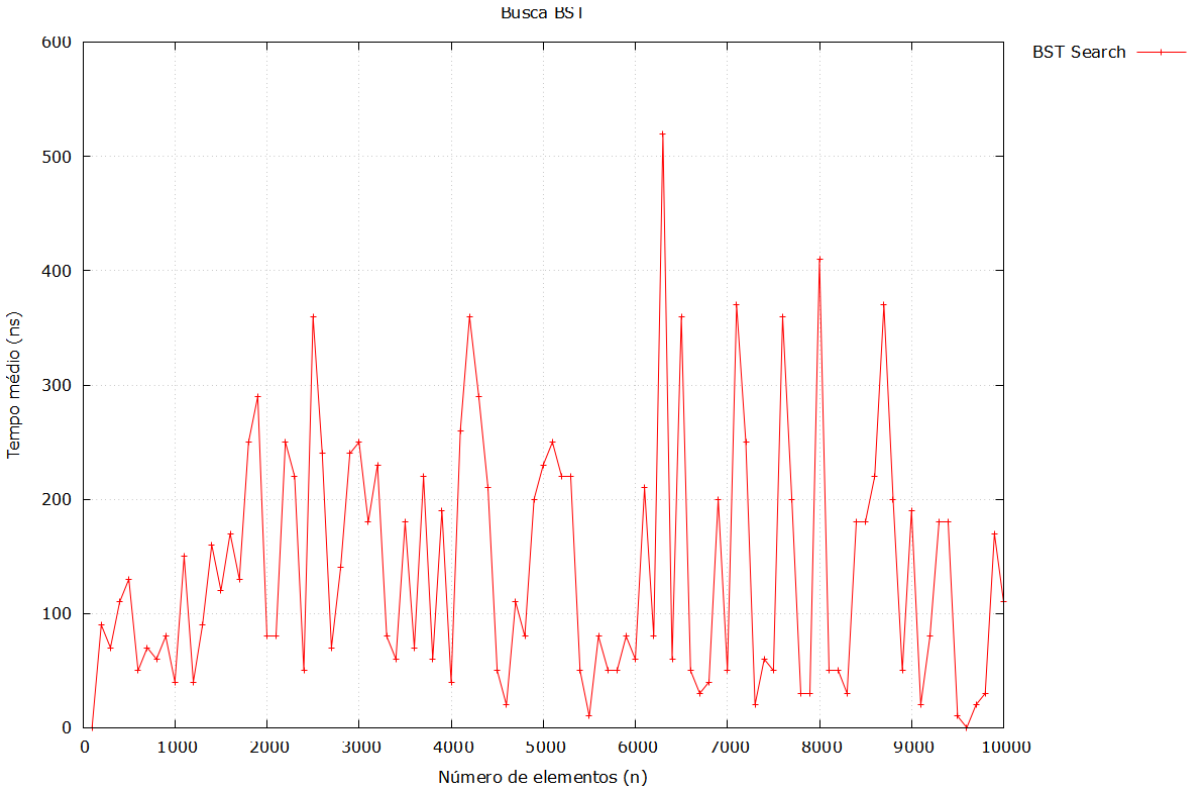

Figura 8: Busca na Árvore BST.

O gráfico de busca da BST mostra claramente sua principal limitação: os tempos de busca aumentam conforme cresce o número de elementos. Além disso, é possível observar variações significativas entre as medições, mesmo para tamanhos de dados próximos. Isso ocorre porque, dependendo da ordem de inserção dos elementos aleatórios em cada repetição, a árvore pode ficar mais ou menos desbalanceada, afetando diretamente o tempo necessário para percorrer seus nós durante a busca. Em casos onde a árvore se aproxima de uma lista encadeada, cada busca precisa percorrer quase todos os elementos até encontrar o valor desejado, resultando em complexidade linear O(n). Esse comportamento instável reduz a eficiência da BST em aplicações que exigem consultas rápidas e previsíveis em grandes volumes de dados.

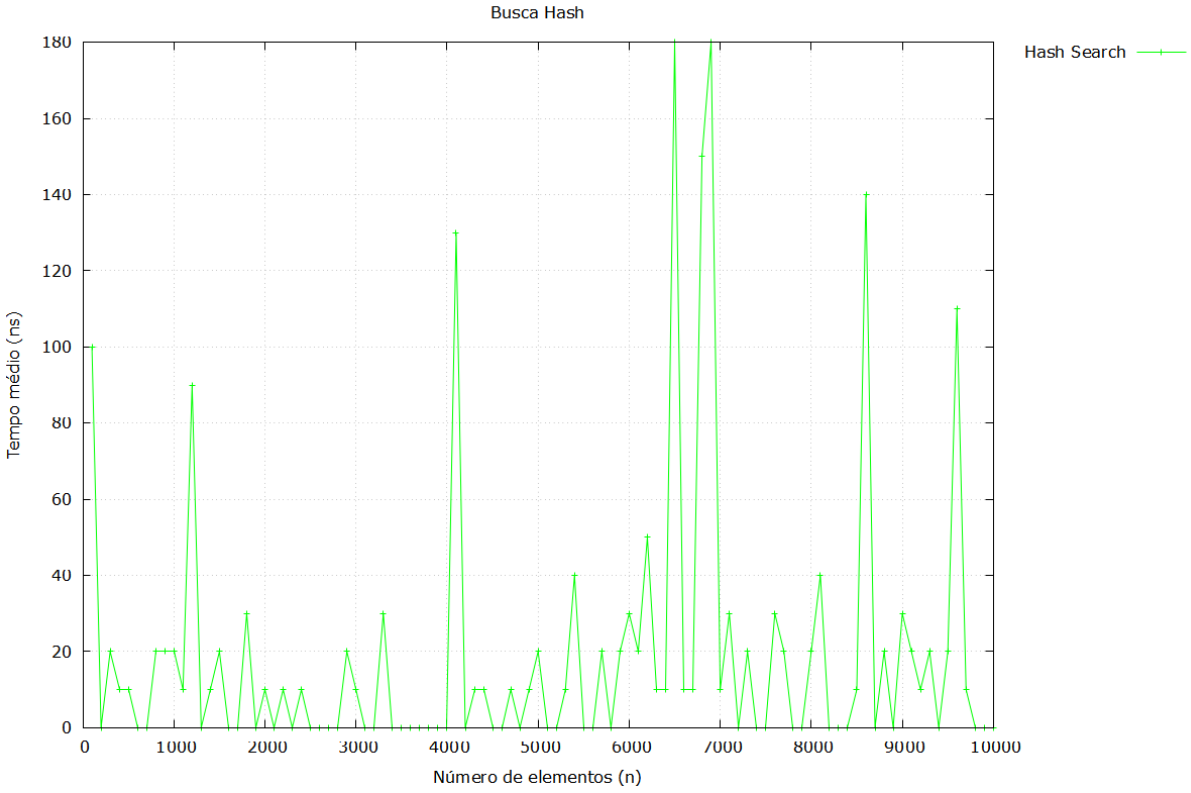

Figura 9: Busca na Tabela Hash.

Este gráfico reforça a eficiência da Tabela Hash, mostrando tempos médios de busca praticamente constantes, mesmo em datasets com 10.000 elementos. Isso ocorre porque a busca é realizada diretamente pelo cálculo do índice via função hash. Eventuais colisões são resolvidas de forma rápida pelo encadeamento separado, mantendo o tempo médio baixo. Essa característica torna a Tabela Hash indispensável em sistemas que priorizam consultas rápidas, como caches e indexadores.

4.0.4 Comparação Inserção e Busca na Mesma Estrutura

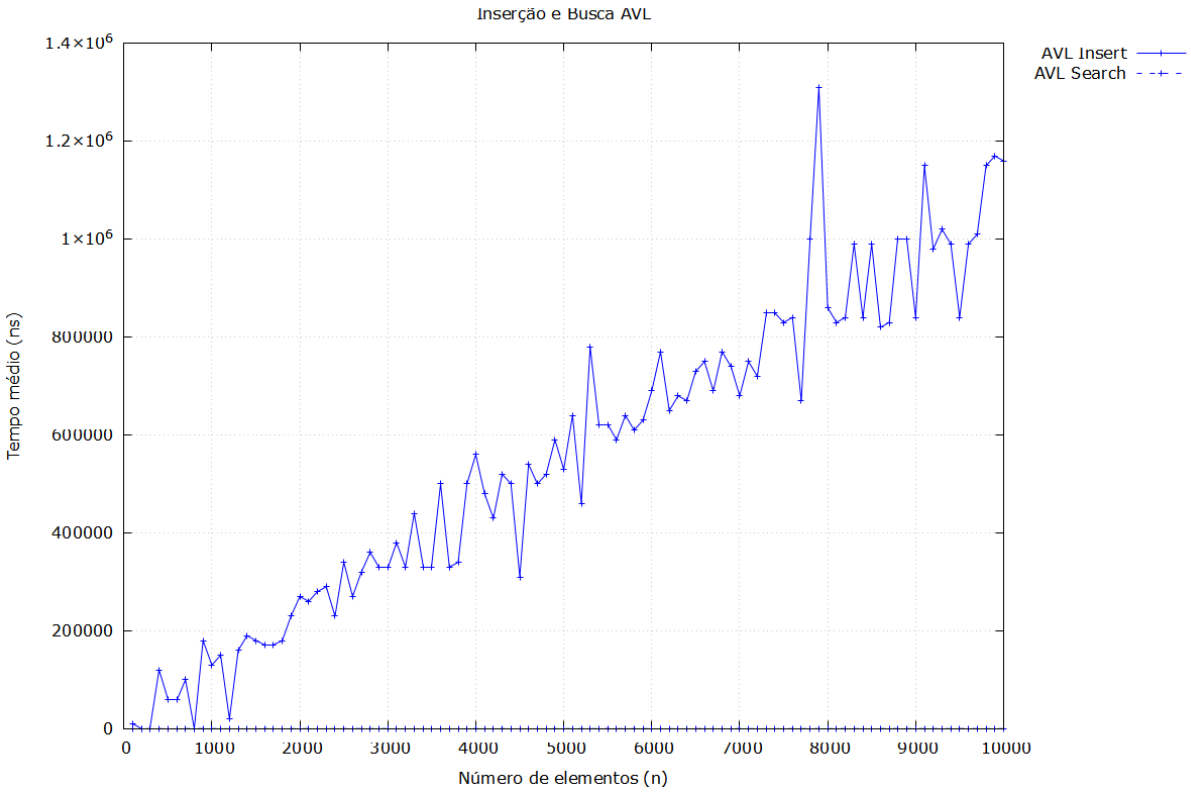

Figura 10: Inserção e busca na Árvore AVL.

Neste gráfico, observa-se que o tempo de inserção na AVL é ligeiramente superior ao tempo de busca, resultado do custo adicional das rotações de balanceamento. No entanto, ambos os tempos permanecem baixos e crescem suavemente, demonstrando a eficiência geral da AVL em manter seu equilíbrio estrutural mesmo com grande volume de inserções e buscas subsequentes.

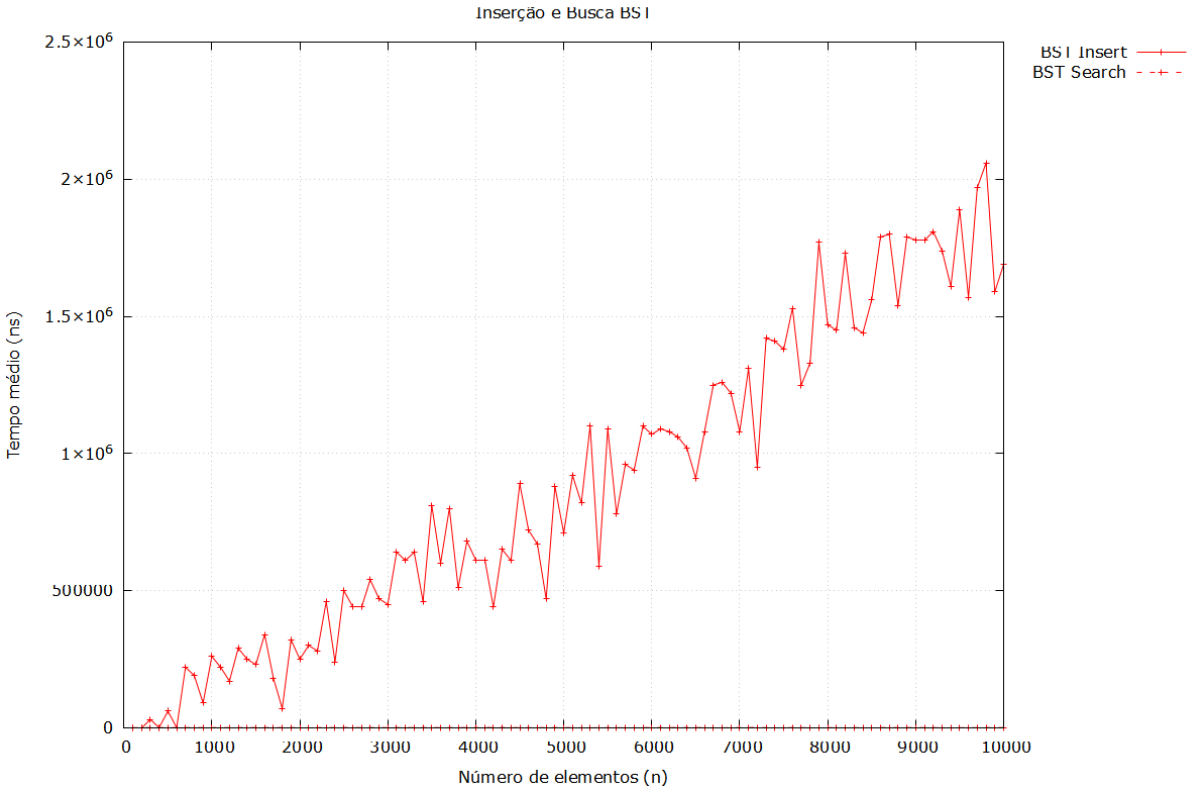

Figura 11: Inserção e busca na Árvore BST.

Na BST, o tempo de busca se mostra consideravelmente maior do que o de inserção, principalmente em datasets grandes, evidenciando o impacto do desbalanceamento nas operações de consulta. O tempo de inserção também cresce, mas em menor proporção, pois inserir em nós folha não exige percorrer todos os elementos. Ainda assim, o desempenho geral é inferior às demais estruturas analisadas.

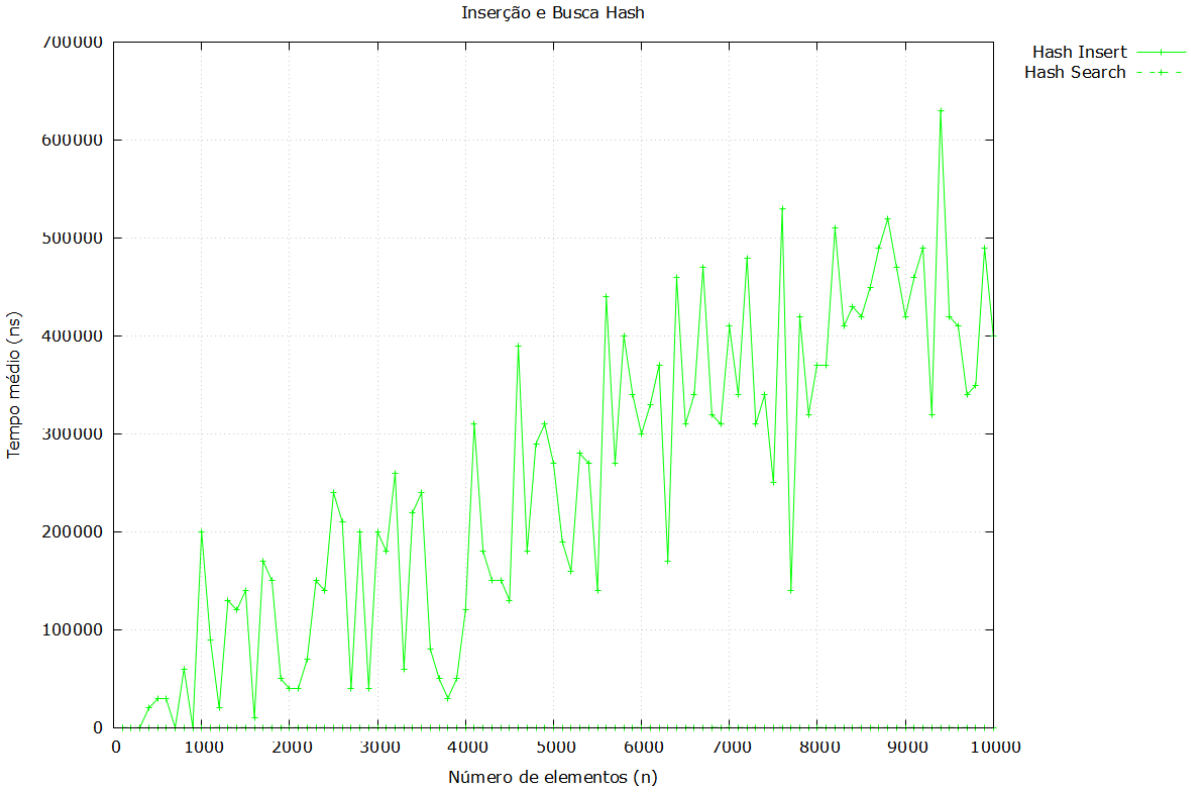

Figura 12: Inserção e busca na Tabela Hash.

Por fim, na Tabela Hash, tanto a operação de inserção quanto de busca apresentam tempos praticamente iguais e constantes, independentemente do tamanho do dataset. Isso comprova sua eficiência e a consolida como a estrutura de dados mais indicada para aplicações que priorizam rapidez em inserções e consultas, sem necessidade de ordenação interna.

5 Conclusão

Este trabalho apresentou uma análise prática e comparativa entre três estruturas de dados amplamente utilizadas — Árvore Binária de Busca (BST), Árvore AVL e Tabela Hash — com foco nas operações de inserção e busca. A partir da implementação em linguagem C e de medições realizadas com dados pseudoaleatórios, foi possível observar como cada estrutura se comporta com diferentes volumes de entrada.

Embora os testes tenham sido conduzidos com critérios de repetição e normalização para reduzir distorções, o ambiente de execução — um sistema operacional Windows comum — apresentou variações de desempenho devido à influência de processos paralelos, variações de carga e limitações do clock interno, o que é importante considerar na interpretação dos resultados. Ainda assim, os dados coletados permitiram identificar tendências claras no comportamento de cada estrutura.

A Tabela Hash destacou-se pela rapidez em ambas as operações, mantendo tempos médios quase constantes mesmo com grandes volumes de dados. A Árvore AVL, apesar de apresentar um custo adicional na inserção por conta das rotações de balanceamento, demonstrou ótimo desempenho e previsibilidade em cenários que exigem dados ordenados. Por outro lado, a BST sem balanceamento teve sua performance comprometida em datasets maiores, sobretudo quando os dados tendiam a estar ordenados, o que levou a um crescimento linear do tempo de busca.

Dessa forma, o trabalho evidencia que a escolha da estrutura de dados deve considerar não apenas a complexidade teórica, mas também o ambiente de execução e o perfil real de uso da aplicação. Além disso, destaca-se a importância da experimentação prática, mesmo com limitações, como ferramenta essencial para validar suposições e orientar decisões mais precisas em projetos que envolvem manipulação de dados em larga escala.

6 Códigos

6.1 Código Fonte - Árvore Binária de Busca (BST)

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MAX_N 10000
#define STEP 100
#define REPETICOES 100

#define MEDIR_BUSCA 0
#define MEDIR_INSERCAO 1

typedef struct node
{
    int value;
    struct node *left, *right;
} Node;

Node *initialize_node(int node)
{
    Node *w = (Node *)malloc(sizeof(Node));
    w->value = node;
    w->left = NULL;
    w->right = NULL;
    return w;
}

void insert(Node **A, Node *w)
{
    if (*A == NULL)
    {
        *A = w;
```

```
32          return;
33      }
34
35      if (w->value < (*A)->value)
36          insert(&(*A)->left, w);
37      else
38          insert(&(*A)->right, w);
39  }
40
41  Node *search(Node *A, int target)
42  {
43      if (A == NULL)
44          return NULL;
45      if (A->value == target)
46          return A;
47      if (target < A->value)
48          return search(A->left, target);
49      else
50          return search(A->right, target);
51  }
52
53  void free_tree(Node *root)
54  {
55      if (root == NULL) return;
56      free_tree(root->left);
57      free_tree(root->right);
58      free(root);
59  }
60
61  int main()
62  {
63      srand(42);
64
65      const int BUSCAS_POR_REPETICAO = 1000;
66
67  #if MEDIR_BUSCA
68      FILE *fp_search = fopen("search_bst.txt", "w");
69      if (!fp_search)
70      {
71          perror("Erro ao abrir arquivo de busca");
72          return 1;
73      }
74
```

```
75      for (int n = STEP; n <= MAX_N; n += STEP)
76      {
77          double total_search_time = 0;
78
79          for (int rep = 0; rep < REPETICOES; rep++)
80          {
81              Node *head = NULL;
82
83              for (int i = 0; i < n; i++)
84              {
85                  int numberInsert = rand() % (n * 10);
86                  Node *node = initialize_node(numberInsert);
87                  insert(&head, node);
88              }
89
90              clock_t t_start = clock();
91
92              for (int k = 0; k < BUSCAS_POR_REPETICAO; k++)
93              {
94                  int numberSearch = rand() % (n * 10);
95                  search(head, numberSearch);
96              }
97
98              clock_t t_end = clock();
99
100             total_search_time += ((double)(t_end - t_start) /
                    CLOCKS_PER_SEC) * 1e9 / BUSCAS_POR_REPETICAO;
101
102             free_tree(head);
103         }
104
105         long avg_search_time = (long)(total_search_time / REPETICOES);
106         fprintf(fp_search, "%d,%ld\n", n, avg_search_time);
107         fflush(fp_search);
108         printf("n=%d busca=%ld ns\n", n, avg_search_time);
109     }
110
111     fclose(fp_search);
112 #endif
113
114 #if MEDIR_INSERCAO
115     FILE *fp_insert = fopen("insert_bst.txt", "w");
116     if (!fp_insert)
```

```
117     {
118         perror("Erro ao abrir arquivo de insercao");
119         return 1;
120     }
121 
122     for (int n = STEP; n <= MAX_N; n += STEP)
123     {
124         double total_insert_time = 0;
125 
126         for (int rep = 0; rep < REPETICOES; rep++)
127         {
128             Node *head = NULL;
129 
130             clock_t t_start = clock();
131 
132             for (int i = 0; i < n; i++)
133             {
134                 int numberInsert = rand() % (n * 10);
135                 Node *node = initialize_node(numberInsert);
136                 insert(&head, node);
137             }
138 
139             clock_t t_end = clock();
140 
141             total_insert_time += ((double)(t_end - t_start) /
                    CLOCKS_PER_SEC) * 1e9;
142 
143             free_tree(head);
144         }
145 
146         long avg_insert_time = (long)(total_insert_time / REPETICOES);
147         fprintf(fp_insert, "%d,%ld\n", n, avg_insert_time);
148         fflush(fp_insert);
149         printf("n=%d insercao=%ld ns\n", n, avg_insert_time);
150     }
151 
152     fclose(fp_insert);
153 #endif
154 
155     return 0;
156 }
```

6.2 Código Fonte - Árvore AVL

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MAX_N 10000
#define STEP 100
#define REPETICOES 100

#define MEDIR_BUSCA 0
#define MEDIR_INSERCAO 1

typedef struct node
{
    int value;
    struct node *left, *right, *parent;
    int height;
} Node;

Node *initialize_node(int node)
{
    Node *w = (Node *)malloc(sizeof(Node));
    w->value = node;
    w->left = NULL;
    w->right = NULL;
    w->height = 0;
    return w;
}

int max(int a, int b)
{
    return (a > b) ? a : b;
}

void rLeft(Node **A)
{
    Node *x = (*A);
    Node *y = (*A)->right;

    x->right = y->left;
    if (x->right != NULL)
        x->right->parent = x;
```

```
42
43      y->left = x;
44      y->parent = x->parent;
45      x->parent = y;
46
47      int x_leftH = (x->left) ? x->left->height : -1;
48      int x_rightH = (x->right) ? x->right->height : -1;
49      x->height = max(x_leftH, x_rightH) + 1;
50
51      int y_leftH = (y->left) ? y->left->height : -1;
52      int y_rightH = (y->right) ? y->right->height : -1;
53      y->height = max(y_leftH, y_rightH) + 1;
54
55      if (y->parent == NULL)
56          (*A) = y;
57  }
58
59  void rRight(Node **A)
60  {
61      Node *x = (*A);
62      Node *y = x->left;
63
64      x->left = y->right;
65      if (x->left != NULL)
66          x->left->parent = x;
67
68      y->right = x;
69      y->parent = x->parent;
70      x->parent = y;
71
72      int x_leftH = (x->left) ? x->left->height : -1;
73      int x_rightH = (x->right) ? x->right->height : -1;
74      x->height = max(x_leftH, x_rightH) + 1;
75
76      int y_leftH = (y->left) ? y->left->height : -1;
77      int y_rightH = (y->right) ? y->right->height : -1;
78      y->height = max(y_leftH, y_rightH) + 1;
79
80      if (y->parent == NULL)
81          (*A) = y;
82  }
83
84  void update(Node **A)
```

```
85  {
86      int leftH = ((*A)->left) ? (*A)->left->height : -1;
87      int rightH = ((*A)->right) ? (*A)->right->height : -1;
88      int fatorB = leftH - rightH;
89
90      (*A)->height = max(leftH, rightH) + 1;
91
92      if (fatorB > 1)
93      {
94          int childFB = ((*A)->left)
95                              ? (((*A)->left->left ? (*A)->left->left->
                                    height : -1) -
96                                 ((*A)->left->right ? (*A)->left->right->
                                       height : -1))
97                              : 0;
98
99          if (childFB >= 0)
100         {
101             rRight(A);
102         }
103         else
104         {
105             rLeft(&(*A)->left);
106             rRight(A);
107         }
108     }
109     else if (fatorB < -1)
110     {
111         int childFB = ((*A)->right)
112                             ? (((*A)->right->left ? (*A)->right->left->
                                    height : -1) -
113                                ((*A)->right->right ? (*A)->right->right->
                                       height : -1))
114                             : 0;
115
116         if (childFB <= 0)
117         {
118             rLeft(A);
119         }
120         else
121         {
122             rRight(&(*A)->right);
123             rLeft(A);
```

```
124         }
125     }
126 }
127 
128 void insert(Node **A, Node *w, Node *parent)
129 {
130     if ((*A) != NULL)
131     {
132         if ((*A)->value >= w->value)
133             insert(&((*A)->left), w, (*A));
134         else
135             insert(&((*A)->right), w, (*A));
136         update(A);
137     }
138     else
139     {
140         w->parent = parent;
141         (*A) = w;
142     }
143 }
144 
145 Node *search(Node *A, int target)
146 {
147     if (A == NULL)
148         return NULL;
149     if (A->value == target)
150         return A;
151     if (target < A->value)
152         return search(A->left, target);
153     else
154         return search(A->right, target);
155 }
156 
157 void free_tree(Node *root)
158 {
159     if (root == NULL) return;
160     free_tree(root->left);
161     free_tree(root->right);
162     free(root);
163 }
164 
165 int main()
166 {
```

```
167     srand(42);
168
169     const int BUSCAS_POR_REPETICAO = 1000;
170
171 #if MEDIR_BUSCA
172     FILE *fp_search = fopen("search_avl.txt", "w");
173     if (!fp_search)
174     {
175         perror("Erro ao abrir arquivo de busca");
176         return 1;
177     }
178
179     for (int n = STEP; n <= MAX_N; n += STEP)
180     {
181         double total_search_time = 0;
182
183         for (int rep = 0; rep < REPETICOES; rep++)
184         {
185             Node *head = NULL;
186
187             for (int i = 0; i < n; i++)
188             {
189                 int numberInsert = rand() % (n * 10);
190                 Node *node = initialize_node(numberInsert);
191                 insert(&head, node, NULL);
192             }
193
194             clock_t t_start = clock();
195
196             for (int k = 0; k < BUSCAS_POR_REPETICAO; k++)
197             {
198                 int numberSearch = rand() % (n * 10);
199                 search(head, numberSearch);
200             }
201
202             clock_t t_end = clock();
203
204             total_search_time += ((double)(t_end - t_start) /
                    CLOCKS_PER_SEC) * 1e9 / BUSCAS_POR_REPETICAO;
205
206             free_tree(head);
207         }
208
```

```
209         long avg_search_time = (long)(total_search_time / REPETICOES);
210         fprintf(fp_search, "%d,%ld\n", n, avg_search_time);
211         fflush(fp_search);
212         printf("n=%d busca=%ld ns\n", n, avg_search_time);
213     }
214
215     fclose(fp_search);
216 #endif
217
218 #if MEDIR_INSERCAO
219     FILE *fp_insert = fopen("insert_avl.txt", "w");
220     if (!fp_insert)
221     {
222         perror("Erro ao abrir arquivo de insercao");
223         return 1;
224     }
225
226     for (int n = STEP; n <= MAX_N; n += STEP)
227     {
228         double total_insert_time = 0;
229
230         for (int rep = 0; rep < REPETICOES; rep++)
231         {
232             Node *head = NULL;
233
234             clock_t t_start = clock();
235
236             for (int i = 0; i < n; i++)
237             {
238                 int numberInsert = rand() % (n * 10);
239                 Node *node = initialize_node(numberInsert);
240                 insert(&head, node, NULL);
241             }
242
243             clock_t t_end = clock();
244
245             total_insert_time += ((double)(t_end - t_start) /
                    CLOCKS_PER_SEC) * 1e9;
246
247             free_tree(head);
248         }
249
250         long avg_insert_time = (long)(total_insert_time / REPETICOES);
```

```
251         fprintf(fp_insert, "%d,%ld\n", n, avg_insert_time);
252         fflush(fp_insert);
253         printf("n=%d insercao=%ld ns\n", n, avg_insert_time);
254     }
255
256     fclose(fp_insert);
257 #endif
258
259     return 0;
260 }
```

6.3 Código Fonte - Tabela Hash

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4
5  #define MAX_N 10000
6  #define STEP 100
7  #define REPETICOES 100
8
9  #define MEDIR_BUSCA 0
10 #define MEDIR_INSERCAO 1
11
12 #define HASH_SIZE 20011  // Um número primo grande pra evitar colisões
13
14 typedef struct node
15 {
16     int value;
17     struct node *next;
18 } Node;
19
20 int hash_function(int key)
21 {
22     return key % HASH_SIZE;
23 }
24
25 void insert(Node **table, int value)
26 {
27     int index = hash_function(value);
28     Node *new_node = (Node *)malloc(sizeof(Node));
```

```
29      new_node->value = value;
30      new_node->next = table[index];
31      table[index] = new_node;
32  }
33
34  Node *search(Node **table, int target)
35  {
36      int index = hash_function(target);
37      Node *current = table[index];
38      while (current)
39      {
40          if (current->value == target)
41              return current;
42          current = current->next;
43      }
44      return NULL;
45  }
46
47  void free_table(Node **table)
48  {
49      for (int i = 0; i < HASH_SIZE; i++)
50      {
51          Node *current = table[i];
52          while (current)
53          {
54              Node *tmp = current;
55              current = current->next;
56              free(tmp);
57          }
58      }
59  }
60
61  int main()
62  {
63      srand(42);
64
65      const int BUSCAS_POR_REPETICAO = 1000;
66
67  #if MEDIR_BUSCA
68      FILE *fp_search = fopen("search_hash.txt", "w");
69      if (!fp_search)
70      {
71          perror("Erro ao abrir arquivo de busca");
```

```
72          return 1;
73      }
74
75      for (int n = STEP; n <= MAX_N; n += STEP)
76      {
77          double total_search_time = 0;
78
79          for (int rep = 0; rep < REPETICOES; rep++)
80          {
81              Node *table[HASH_SIZE] = {0};
82
83              for (int i = 0; i < n; i++)
84              {
85                  int numberInsert = rand() % (n * 10);
86                  insert(table, numberInsert);
87              }
88
89              clock_t t_start = clock();
90
91              for (int k = 0; k < BUSCAS_POR_REPETICAO; k++)
92              {
93                  int numberSearch = rand() % (n * 10);
94                  search(table, numberSearch);
95              }
96
97              clock_t t_end = clock();
98
99              total_search_time += ((double)(t_end - t_start) /
                    CLOCKS_PER_SEC) * 1e9 / BUSCAS_POR_REPETICAO;
100
101             free_table(table);
102         }
103
104         long avg_search_time = (long)(total_search_time / REPETICOES);
105         fprintf(fp_search, "%d,%ld\n", n, avg_search_time);
106         fflush(fp_search);
107         printf("n=%d busca=%ld ns\n", n, avg_search_time);
108     }
109
110     fclose(fp_search);
111 #endif
112
113 #if MEDIR_INSERCAO
```

```
114     FILE *fp_insert = fopen("insert_hash.txt", "w");
115     if (!fp_insert)
116     {
117         perror("Erro ao abrir arquivo de insercao");
118         return 1;
119     }
120 
121     for (int n = STEP; n <= MAX_N; n += STEP)
122     {
123         double total_insert_time = 0;
124 
125         for (int rep = 0; rep < REPETICOES; rep++)
126         {
127             Node *table[HASH_SIZE] = {0};
128 
129             clock_t t_start = clock();
130 
131             for (int i = 0; i < n; i++)
132             {
133                 int numberInsert = rand() % (n * 10);
134                 insert(table, numberInsert);
135             }
136 
137             clock_t t_end = clock();
138 
139             total_insert_time += ((double)(t_end - t_start) /
                    CLOCKS_PER_SEC) * 1e9;
140 
141             free_table(table);
142         }
143 
144         long avg_insert_time = (long)(total_insert_time / REPETICOES);
145         fprintf(fp_insert, "%d,%ld\n", n, avg_insert_time);
146         fflush(fp_insert);
147         printf("n=%d insercao=%ld ns\n", n, avg_insert_time);
148     }
149 
150     fclose(fp_insert);
151 #endif
152 
153     return 0;
154 }
```

6.4 Script Gnuplot para Geração dos Gráficos

```
# Saída com boa qualidade
set datafile separator ","
set terminal pngcairo size 1200,800 enhanced font 'Verdana,12'
set grid
set xlabel "Número de elementos (n)"
set ylabel "Tempo médio (ns)"
set key outside

# 1       GERAL - TODOS JUNTOS (inserção e busca misturados)
set output "comparacao_geral.png"
set title "Inserção e Busca - AVL, BST, Hash (Tudo Junto)"
plot \
    "insert_avl.txt" using 1:2 with linespoints lt 1 lc rgb "blue" title
        "AVL Insert", \
    "search_avl.txt" using 1:2 with linespoints lt 1 lc rgb "blue" dt 2
       title "AVL Search", \
    "insert_bst.txt" using 1:2 with linespoints lt 1 lc rgb "red" title
       "BST Insert", \
    "search_bst.txt" using 1:2 with linespoints lt 1 lc rgb "red" dt 2
       title "BST Search", \
    "insert_hash.txt" using 1:2 with linespoints lt 1 lc rgb "green"
       title "Hash Insert", \
    "search_hash.txt" using 1:2 with linespoints lt 1 lc rgb "green" dt
       2 title "Hash Search"

# 2       INSER ÃO - COMPARANDO ESTRUTURAS
set output "insercao_todas.png"
set title "Comparação de Inserção - AVL, BST, Hash"
plot \
    "insert_avl.txt" using 1:2 with linespoints lt 1 lc rgb "blue" title
        "AVL Insert", \
    "insert_bst.txt" using 1:2 with linespoints lt 1 lc rgb "red" title
       "BST Insert", \
    "insert_hash.txt" using 1:2 with linespoints lt 1 lc rgb "green"
       title "Hash Insert"

# 3       BUSCA - COMPARANDO ESTRUTURAS
set output "busca_todas.png"
set title "Comparação de Busca - AVL, BST, Hash"
plot \
```

```
32      "search_avl.txt" using 1:2 with linespoints lt 1 lc rgb "blue" title
            "AVL Search", \
33      "search_bst.txt" using 1:2 with linespoints lt 1 lc rgb "red" title
            "BST Search", \
34      "search_hash.txt" using 1:2 with linespoints lt 1 lc rgb "green"
            title "Hash Search"
35
36  # 4        INSER ÃO INDIVIDUAL
37  set title "Inserção AVL"
38  set output "insercao_avl.png"
39  plot "insert_avl.txt" using 1:2 with linespoints lt 1 lc rgb "blue"
        title "AVL Insert"
40
41  set title "Inserção BST"
42  set output "insercao_bst.png"
43  plot "insert_bst.txt" using 1:2 with linespoints lt 1 lc rgb "red" title
        "BST Insert"
44
45  set title "Inserção Hash"
46  set output "insercao_hash.png"
47  plot "insert_hash.txt" using 1:2 with linespoints lt 1 lc rgb "green"
        title "Hash Insert"
48
49  # 5        BUSCA INDIVIDUAL
50  set title "Busca AVL"
51  set output "busca_avl.png"
52  plot "search_avl.txt" using 1:2 with linespoints lt 1 lc rgb "blue"
        title "AVL Search"
53
54  set title "Busca BST"
55  set output "busca_bst.png"
56  plot "search_bst.txt" using 1:2 with linespoints lt 1 lc rgb "red" title
        "BST Search"
57
58  set title "Busca Hash"
59  set output "busca_hash.png"
60  plot "search_hash.txt" using 1:2 with linespoints lt 1 lc rgb "green"
        title "Hash Search"
61
62  # 6        INSER ÃO E BUSCA JUNTOS - CADA UM
63  set title "Inserção e Busca AVL"
64  set output "avl_insercao_busca.png"
65  plot \
```

```
66      "insert_avl.txt" using 1:2 with linespoints lt 1 lc rgb "blue" title
            "AVL Insert", \
67      "search_avl.txt" using 1:2 with linespoints lt 1 lc rgb "blue" dt 2
           title "AVL Search"
68
69  set title "Inserção e Busca BST"
70  set output "bst_insercao_busca.png"
71  plot \
72      "insert_bst.txt" using 1:2 with linespoints lt 1 lc rgb "red" title
           "BST Insert", \
73      "search_bst.txt" using 1:2 with linespoints lt 1 lc rgb "red" dt 2
           title "BST Search"
74
75  set title "Inserção e Busca Hash"
76  set output "hash_insercao_busca.png"
77  plot \
78      "insert_hash.txt" using 1:2 with linespoints lt 1 lc rgb "green"
           title "Hash Insert", \
79      "search_hash.txt" using 1:2 with linespoints lt 1 lc rgb "green" dt
           2 title "Hash Search"
```