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

A eficiência na manipulação e recuperação de informações é um aspecto central no desenvolvimento de sistemas computacionais. Estruturas de dados bem definidas e algoritmos de busca otimizados são fundamentais para garantir desempenho satisfatório em aplicações que exigem consultas rápidas, como bancos de dados, sistemas de arquivos, mecanismos de indexação e soluções de armazenamento em grande escala.

Neste contexto, o presente trabalho tem como objetivo analisar, na prática, o comportamento de três estruturas de dados clássicas amplamente utilizadas em operações de busca: a árvore binária de busca (BST), a árvore AVL (auto-balanceada) e a tabela de dispersão (Hash Table). Para isso, foram desenvolvidos programas em linguagem C, responsáveis por realizar a inserção de dados pseudo-aleatórios, executar operações de busca e medir o tempo de processamento dessas operações em diferentes volumes de dados.

A metodologia adotada incluiu a implementação individual de cada estrutura, com geração de arquivos contendo os tempos médios de execução das buscas. Esses dados foram posteriormente organizados em gráficos utilizando o Gnuplot, ferramenta escolhida pela sua capacidade de representar resultados experimentais de forma clara e precisa.

Além da parte prática de implementação e testes, este relatório descreve a lógica de funcionamento de cada estrutura, os aspectos teóricos que impactam seu desempenho e uma análise comparativa dos resultados obtidos. A organização do texto segue uma divisão clara: os capítulos seguintes abordam, individualmente, a árvore binária, a árvore AVL e a tabela Hash, detalhando suas características e resultados obtidos em cada experimento. Em seguida, apresenta-se uma comparação geral entre as três técnicas, acompanhada dos gráficos gerados e, por fim, uma conclusão que resume as observações mais relevantes.

A proposta, portanto, visa consolidar o entendimento das diferenças práticas entre as estruturas analisadas, reforçando a importância da escolha criteriosa de algoritmos e estruturas de dados para aplicações que dependem de operações de busca eficientes e escaláveis.

2 Análises dos dados

2.1 Estruturas de Dados Analisadas

Neste capítulo são descritas individualmente as três estruturas de dados analisadas: a árvore binária de busca (BST), a árvore AVL e a tabela de dispersão (Hash Table). Cada subseção apresenta uma breve fundamentação teórica, os principais aspectos de implementação, o procedimento de medição do tempo de execução e considerações sobre os resultados obtidos.

2.1.1 Árvore Binária de Busca (BST)

A árvore binária de busca (Binary Search Tree) é uma estrutura hierárquica em que cada nó possui no máximo dois filhos, organizados de forma que o valor do nó à esquerda é menor que o valor do nó pai, e o valor do nó à direita é maior. Essa propriedade permite que operações de inserção, remoção e busca sejam realizadas de forma relativamente eficiente, com complexidade média de $O(\log n)$ em árvores balanceadas. No entanto, caso os dados sejam inseridos em ordem crescente ou decrescente, a árvore pode se degenerar em uma lista ligada, com pior caso de complexidade $O(n)$.

No experimento, a árvore binária foi implementada em linguagem C, com funções recursivas para inserção e busca de elementos. Foram inseridos dados pseudo-aleatórios para reduzir o risco de degeneração da árvore. Para medir o tempo de busca, o mesmo valor foi procurado repetidas vezes, garantindo um tempo total suficientemente grande para que a medição fosse precisa. Os tempos foram armazenados no arquivo `tempos_binaria.dat` para posterior plotagem. Aqui está o código que usei:

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

typedef struct Node
{
    int valor;
    struct Node *esq, *dir;
```

```
} Node;

Node *inserir(Node *raiz, int valor)
{
    if (raiz == NULL)
    {
        Node *novo = malloc(sizeof(Node));
        novo->valor = valor;
        novo->esq = novo->dir = NULL;
        return novo;
    }
    if (valor < raiz->valor)
        raiz->esq = inserir(raiz->esq, valor);
    else
        raiz->dir = inserir(raiz->dir, valor);
    return raiz;
}

int buscar(Node *raiz, int valor)
{
    if (raiz == NULL)
        return 0;
    if (raiz->valor == valor)
        return 1;
    if (valor < raiz->valor)
        return buscar(raiz->esq, valor);
    else
        return buscar(raiz->dir, valor);
}

int main()
{
    Node *raiz = NULL;
    int N[] = {10000, 50000, 100000};
    FILE *f = fopen("graficos/tempos_binaria.dat", "w");

    for (int i = 0; i < 3; i++)
    {
        srand(42);
        raiz = NULL;

        for (int j = 0; j < N[i]; j++)
            raiz = inserir(raiz, rand());
```

```
52
53          srand(time(NULL));
54          int chave = rand();
55
56          int repeticoes = 10000000;
57
58          clock_t ini = clock();
59          for (int k = 0; k < repeticoes; k++)
60          {
61              buscar(raiz, chave);
62          }
63          clock_t fim = clock();
64
65          double tempo = (double)(fim - ini) / CLOCKS_PER_SEC; // tempo
                total
66
67          fprintf(f, "%d\t%lf\n", N[i], tempo);
68          printf("BST: N=%d Tempo=%lf\n", N[i], tempo);
69      }
70      fclose(f);
71      return 0;
72  }
```

2.1.2 Árvore AVL

A árvore AVL é uma árvore binária de busca auto-balanceada, criada por Adelson-Velsky e Landis, que garante que a diferença de altura entre as subárvores esquerda e direita de qualquer nó seja no máximo um. Para manter essa propriedade, são aplicadas rotações simples ou duplas durante a inserção de novos elementos, evitando a formação de árvores degeneradas.

No projeto, a árvore AVL foi desenvolvida em C, com funções específicas para inserção balanceada, cálculo de altura, rotações e busca. Assim como na BST, os dados foram gerados de forma pseudo-aleatória e as buscas foram repetidas múltiplas vezes. Os tempos médios foram gravados no arquivo `tempos_avl.dat`. Em comparação com a árvore binária simples, a AVL tende a manter o desempenho de busca mais estável, mesmo com grandes quantidades de dados. Aqui está o código:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
```

```
4
5  typedef struct Node
6  {
7      int valor, altura;
8      struct Node *esq, *dir;
9  } Node;
10
11 int altura(Node *n)
12 {
13     return n ? n->altura : 0;
14 }
15
16 int max(int a, int b)
17 {
18     return a > b ? a : b;
19 }
20
21 Node *rot_dir(Node *y)
22 {
23     Node *x = y->esq;
24     Node *T2 = x->dir;
25     x->dir = y;
26     y->esq = T2;
27     y->altura = max(altura(y->esq), altura(y->dir)) + 1;
28     x->altura = max(altura(x->esq), altura(x->dir)) + 1;
29     return x;
30 }
31
32 Node *rot_esq(Node *x)
33 {
34     Node *y = x->dir;
35     Node *T2 = y->esq;
36     y->esq = x;
37     x->dir = T2;
38     x->altura = max(altura(x->esq), altura(x->dir)) + 1;
39     y->altura = max(altura(y->esq), altura(y->dir)) + 1;
40     return y;
41 }
42
43 int get_balance(Node *n)
44 {
45     return n ? altura(n->esq) - altura(n->dir) : 0;
46 }
```

```
47
48  Node *inserir(Node *node, int valor)
49  {
50      if (!node)
51      {
52          Node *novo = malloc(sizeof(Node));
53          novo->valor = valor;
54          novo->esq = novo->dir = NULL;
55          novo->altura = 1;
56          return novo;
57      }
58
59      if (valor < node->valor)
60          node->esq = inserir(node->esq, valor);
61      else if (valor > node->valor)
62          node->dir = inserir(node->dir, valor);
63      else
64          return node;
65
66      node->altura = 1 + max(altura(node->esq), altura(node->dir));
67      int balance = get_balance(node);
68
69      if (balance > 1 && valor < node->esq->valor)
70          return rot_dir(node);
71
72      if (balance < -1 && valor > node->dir->valor)
73          return rot_esq(node);
74
75      if (balance > 1 && valor > node->esq->valor)
76      {
77          node->esq = rot_esq(node->esq);
78          return rot_dir(node);
79      }
80
81      if (balance < -1 && valor < node->dir->valor)
82      {
83          node->dir = rot_dir(node->dir);
84          return rot_esq(node);
85      }
86
87      return node;
88  }
89
```

```
90  int buscar(Node *raiz, int valor)
91  {
92      if (raiz == NULL)
93          return 0;
94      if (raiz->valor == valor)
95          return 1;
96      if (valor < raiz->valor)
97          return buscar(raiz->esq, valor);
98      else
99          return buscar(raiz->dir, valor);
100 }
101
102 int main()
103 {
104     Node *raiz = NULL;
105     int N[] = {10000, 50000, 100000};
106     FILE *f = fopen("graficos/tempos_avl.dat", "w");
107
108     for (int i = 0; i < 3; i++)
109     {
110         srand(42);
111         raiz = NULL;
112
113         for (int j = 0; j < N[i]; j++)
114             raiz = inserir(raiz, rand());
115
116         srand(time(NULL));
117         int chave = rand();
118
119         int repeticoes = 10000000;
120
121         clock_t ini = clock();
122         for (int k = 0; k < repeticoes; k++)
123         {
124             buscar(raiz, chave);
125         }
126         clock_t fim = clock();
127
128         double tempo = (double)(fim - ini) / CLOCKS_PER_SEC; // tempo
                total
129
130         fprintf(f, "%d\t%lf\n", N[i], tempo);
131         printf("AVL: N=%d Tempo=%lf\n", N[i], tempo);
```

```
    }
    fclose(f);
    return 0;
}
```

2.1.3 Tabela de Dispersão (Hash Table)

A tabela de dispersão, ou Hash Table, é uma estrutura de dados que mapeia chaves a índices de forma que a busca por um elemento possa ser feita, idealmente, em tempo constante $O(1)$. Para isso, é usada uma função de dispersão (hash function) que distribui uniformemente os valores pela tabela. Para tratar colisões — quando duas chaves diferentes geram o mesmo índice — foi adotado o método de encadeamento, no qual cada posição do vetor de hash aponta para uma lista encadeada de elementos.

A implementação foi feita em linguagem C, utilizando um vetor de listas encadeadas. Foram inseridos valores pseudo-aleatórios e o tempo de busca foi medido repetindo a operação várias vezes, garantindo precisão na coleta dos dados. O resultado foi armazenado no arquivo `tempos_hash.dat`. Conforme esperado, os tempos de busca mantiveram-se praticamente constantes, validando a eficiência da tabela de dispersão para operações de consulta em grandes volumes de dados. Aqui está o código utilizado:

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

#define TAM 200003

typedef struct Node
{
    int valor;
    struct Node *prox;
} Node;

Node *tabela[TAM];

void inserir(int valor)
{
    int idx = valor % TAM;
    Node *novo = malloc(sizeof(Node));
    novo->valor = valor;
    novo->prox = tabela[idx];
```

```
21      tabela[idx] = novo;
22  }
23
24  int buscar(int valor)
25  {
26      int idx = valor % TAM;
27      Node *atual = tabela[idx];
28      while (atual)
29      {
30          if (atual->valor == valor)
31              return 1;
32          atual = atual->prox;
33      }
34      return 0;
35  }
36
37  int main()
38  {
39      int N[] = {10000, 50000, 100000};
40      FILE *f = fopen("graficos/tempos_hash.dat", "w");
41
42      for (int i = 0; i < 3; i++)
43      {
44          srand(42);
45          for (int j = 0; j < TAM; j++)
46              tabela[j] = NULL;
47
48          for (int j = 0; j < N[i]; j++)
49              inserir(rand());
50
51          srand(time(NULL));
52          int chave = rand();
53
54          int repeticoes = 10000000;
55
56          clock_t ini = clock();
57          for (int k = 0; k < repeticoes; k++)
58          {
59              buscar(chave);
60          }
61          clock_t fim = clock();
62
63          double tempo = (double)(fim - ini) / CLOCKS_PER_SEC;
```

```
64
65          fprintf(f, "%d\t%lf\n", N[i], tempo);
66          printf("HASH: N=%d Tempo=%lf\n", N[i], tempo);
67      }
68      fclose(f);
69      return 0;
70  }
```

3 Comparação dos Resultados

Após a execução dos testes individuais para cada estrutura de dados, foi possível comparar de forma prática o desempenho de busca entre a árvore binária de busca (BST), a árvore AVL e a tabela de dispersão (Hash Table). Essa comparação permite visualizar claramente as diferenças de comportamento em relação ao tempo de execução, conforme o volume de dados aumenta.

A Figura 1 apresenta o gráfico gerado com os dados coletados, onde é possível observar o tempo médio de busca para os conjuntos de 10.000, 50.000 e 100.000 elementos em cada estrutura. O gráfico foi produzido com a ferramenta Gnuplot, utilizando os arquivos de saída `tempos_binaria.dat`, `tempos_avl.dat` e `tempos_hash.dat`.

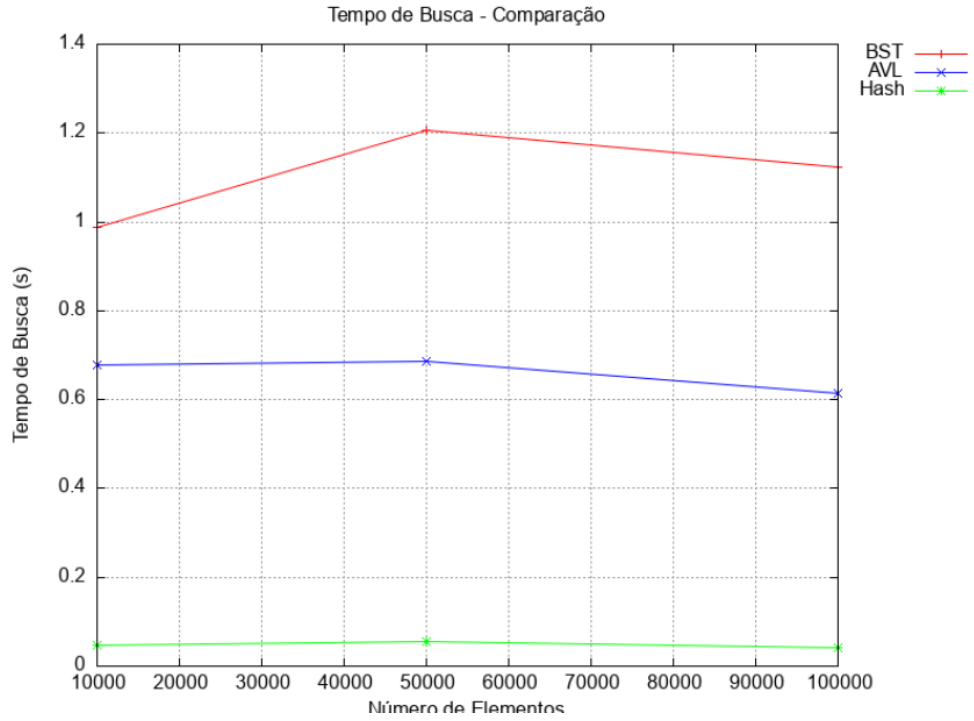

Figura 1: Comparação do tempo de busca: Árvore Binária, Árvore AVL e Tabela Hash

A análise dos dados evidencia que a árvore binária de busca apresenta desempenho inferior quando comparada às demais, principalmente em volumes maiores, devido à possibilidade de desbalanceamento, o que impacta negativamente o tempo de busca. A árvore AVL, por sua vez, demonstra maior eficiência por manter-se balanceada auto-

maticamente, garantindo uma altura logarítmica que otimiza as operações de busca. Já a tabela de dispersão se destaca por apresentar tempo de busca praticamente constante, validando a eficiência do uso de uma função de hash bem definida e de uma estratégia eficaz de tratamento de colisões.

Esses resultados estão em conformidade com a teoria, reforçando a importância de selecionar a estrutura de dados mais adequada de acordo com as características do problema a ser resolvido. Enquanto árvores são recomendadas em cenários que exigem ordenação dinâmica dos dados, a tabela Hash é a opção preferencial quando o foco é acelerar operações de culta sem a necessidade de manter elementos ordenados.

4 Conclusão

A análise prática desenvolvida neste trabalho permitiu comprovar, na prática, como diferentes estruturas de dados se comportam em operações de busca, especialmente quando expostas a conjuntos de dados de tamanhos variados. Foi possível observar que a árvore binária de busca, apesar de simples e didática, pode apresentar desempenho inferior em casos onde não há controle de balanceamento, tornando-se inadequada para aplicações que exigem alta eficiência.

A árvore AVL demonstrou resultados mais consistentes, mantendo tempos de busca baixos mesmo em volumes de dados maiores, graças ao balanceamento automático que preserva a altura logarítmica da árvore. Já a tabela de dispersão destacou-se por apresentar o menor tempo médio de busca entre as três estruturas, confirmando seu potencial para aplicações que exigem consultas rápidas e em larga escala, desde que a função de hash seja eficiente e as colisões sejam bem tratadas.

Dessa forma, os resultados reforçam a importância de uma escolha criteriosa da estrutura de dados, considerando não apenas o volume de informações, mas também as características das operações que serão realizadas sobre os dados. Além disso, o trabalho evidenciou a relevância da prática experimental para complementar o estudo teórico, proporcionando uma compreensão mais concreta do impacto que detalhes de implementação e estratégias de balanceamento têm no desempenho final de um sistema.

Referências