Comparação de métodos de busca em diferentes estruturas de dados

Mateus Tenorio dos Santos[1]

[1]PPGCOMP – Universidade Estadual do Oeste do Paraná (Unioeste)
Rua Universitária, 2069 – Bloco B (Prédio velho) – Bairro Universitário - 85819-110
Cascavel - PR - Brazil

mateus.santos28@unioeste.br

Abstract. *This paper aims to describe some data structures and mainly compare its performance on searching for an element inside of it. This paper presents array, linked list, binary search tree, Avl Tree and RB tree, comparing them both in performance and implementation. With the results presented in this paper is easy to conclude that the algorithms that performed better are related to trees, mainly AVL and RB Tree, because of its balanced characteristic (the tree tries to always be in a balanced state).*

Resumo. *Este artigo tem como objetivo descrever algumas estruturas de dados e principalmente comparar seu desempenho na busca para um elemento dentro dele. Este artigo apresenta array, lista vinculada, árvore de busca binária, árvore Avl e árvore RB, comparando-os em desempenho e implementação. Com os resultados apresentados neste artigo é fácil concluir que os algoritmos que tiveram melhor desempenho estão relacionados às árvores, principalmente AVL e RB Tree, devido à sua característica balanceada (a árvore tenta estar sempre em estado balanceado).*

1. Introdução

Quando estamos no mundo da computação, somos introduzidos a diversas estruturas de dados e algoritmos de busca diferentes. Aprendemos que existem algoritmos que são mais rápidos que outros, estruturas de dados que possibilitam algoritmos de busca extremamente rápido em troca de uso a mais de memória para alocação da estrutura de dados, e algoritmos que conseguem performar tão rápido em seu pior caso quanto no seu caso médio.

Entretanto, isso não é tão visual para nós, uma vez que, com processadores cada vez mais capazes, eficientes e rápidos, esse processo de busca costuma ter um custo em tempo ínfimo, na maioria dos casos na casa dos milésimos de segundo. Isso faz com que, para o desenvolvedor comum, não seja interessante ou relevante aprender e entender a diferença entre as diferentes estruturas de dados e seus possíveis casos de uso. Afinal, porque conhecer e saber as diferenças entre diferentes estruturas de dados se o computador é capaz de resolver tarefas de maneira rápida?

Para isso, primeiro devemos entender o que é uma estrutura de dados. **Estrutura de dados** é o ramo da computação que estuda os diversos mecanismos de organização de dados para atender aos diferentes requisitos de processamento. São responsáveis por definir a organização, métodos de acesso e opções de processamento para a informação

manipulada pelo programa. A definição da organização interna de uma estrutura de dados é tarefa do projetista da estrutura, que define também qual a *API*[1] para a estrutura, ou seja, qual o conjunto de procedimentos que podem ser usados para manipular os dados na estrutura. É esta *API* que determina a visão funcional da estrutura de dados, que é a única informação relevante para um programador que vá utilizar uma estrutura de dados pré-definida.

Tendo esse conhecimento, podemos partir do pressuposto que diferentes estruturas de dados possuíram diferentes aplicações, de modo que umas poderão obter um desempenho maior que outras em determinada aplicação, podendo ou não ter custos a mais pela forma como foi estruturada. Portanto, é correto dizer que não existe uma estrutura de dados que é a melhor ou a correta a se utilizar para um problema, o correto é buscar a estrutura que permite obter o maior desempenho, dadas a necessidade e a capacidade computacional disponível para resolver x ou y problema. O arquiteto/projetista da solução deve compreender e ser capaz de fazer a melhor escolha, levando essas variáveis em consideração.

Portanto, esse artigo tem como objetivo principal abordar algumas estruturas de dados, entre elas o arranjo, a lista ligada, a árvore de busca binária, a árvore AVL e a árvore rubro-negra. O objetivo é apresentá-las, apresentar suas principais características e comparar o desempenho dessas estruturas numa situação simples: Busca de elemento. Para isso, abordaremos o problema de forma simples, realizando 100 testes para o caso médio, com 100.000 elementos inicialmente, que serão incrementados de 100.000 em 100.000 até o valor de 1.000.000 . Os mesmos testes serão refeitos, porém apenas 3 vezes, para o pior caso. Desses testes, serão retiradas as informações de média e desvio padrão para tempo de execução, consumo de memória e número de comparações que o algoritmo de busca realiza. Os métodos de busca serão: Busca sequencial e busca binária para arranjo, busca sequencial para lista ligada e busca em árvore para as três árvores apresentadas.

2. Fundamentação Teórica

Conforme mencionado anteriormente, o termo estrutura de dados refere-se ao ramo da computação que estuda os diversos mecanismos de organização de dados para atender aos diferentes requisitos de processamento. Sendo eles responsáveis por definir a organização, métodos de acesso e opções de processamento para a informação manipulado pelo programa. As estruturas de dados podem ser classificadas de diversas formas, mas uma delas é pela forma em que os dados são organizados. Quando classificamos dessa maneira, obtemos dois tipos de estruturas de dados: Estruturas de dados lineares e estruturas de dados não lineares.

2.1. Estruturas de dados lineares

Para entendermos e conhecermos melhor quem são, precisamos primeiro entender do que se trata a linearidade mencionada no tipo dessas estruturas. A linearidade aqui refere-se a

[1] *Em ciência da computação, a interface de programação de aplicações (do inglês application programming interface, abreviado API) é um conjunto de serviços/funções que foram implementadas em um programa de computador que são disponibilizados para que outros programas/aplicativos possam utiliza-los diretamente de forma simplificada*

maneira com a qual os dados estão organizados dentro dessa estrutura. Isso significa que, no caso das estruturas de dados lineares, cada elemento possui um sucessor e opcionalmente um antecessor. Estruturas de dados comuns que se enquadram nessa categoria são: Arranjos, listas, pilhas, filas e deques. Nesse artigo serão apresentados os arranjos e as listas.

2.2. Estruturas de dados não lineares

Diferentemente das estruturas de dados lineares, nas estruturas de dados não lineares os elementos são organizados de uma forma diferente. Aqui, os elementos são estruturados de forma hierárquica, podendo formar relações complexas entre si. Nessas estruturas, não existe uma ordem clara entre os elementos, ou seja, não existe um sucessor ou antecessor. O que existe é um relacionamento entre os elementos dada uma determinada regra, como é por exemplo nas árvores de busca binária, onde os elementos que são menores são incluídos de um lado, enquanto que os maiores são incluídos do outro. Não existe uma relação de sucessão ou antecessão, e sim uma relação entre maior e menor. Exemplos de estruturas de dados não lineares são: Árvores, grafos, Hash tables. Nesse artigo, focaremos na árvore e seus subtipos.

De forma geral, as estruturas de dados lineares são aquelas em que os elementos são organizados de forma sequencial, onde seu sucessor e antecessor (caso exista) são bem definidos, enquanto que as estruturas de dados não lineares são aquelas em que os dados são organizados de forma hierárquica e não sequencial, mas que possuem uma relação entre si. Ambos os tipos possuem vantagens e desvantagens em suas aplicações, e como mencionado anteriormente, a escolha da estrutura de dados depende exclusivamente das necessidades específicas do problema e fica à cargo do projetista escolher qual melhor se aplica, baseando-se na criticidade e nos recursos disponíveis.

2.3. Notação Big O

[Bae 2019] menciona que antes de aprendermos a implementarmos algoritmos, deveríamos aprender como analisar a efetividade dele e sua performance. [Danziger 2010] explica que para analisarmos a eficiência de um algoritmo devemos sempre olhar para o pior caso. O problema do circuito hamiltoniano pode ser resolvido em aproximadamente $n!$ etapas, considerando todos os circuitos possíveis. Nós podemos ter sorte e descobrir um circuito hamiltoniano na primeira tentativa, mas devemos assumir o pior caso.

A notação Big O tem como objetivo nos informar a eficiência de um algoritmo através da análise do mesmo. Portanto, a notação Big O não nos trás uma medida em tempo, ou seja, ela não vai nos informar o tempo de execução de determinado algoritmo, e sim, uma espécie de aproximação pra quantidade de vezes que aquele algoritmo será executado. Por exemplo, um algoritmo de busca sequencial em um arranjo de tamanho n possui a notação $O(n)$ para tempo de execução, o que significa que para achar um elemento dentro desse arranjo serão feitas n comparações. Entretanto, podemos ter sorte e achar o elemento na primeira posição do arranjo, o que nos dá um tempo de acesso quase que imediato. Porém, como mencionado anteriormente, devemos sempre olhar para o pior caso, e nesse caso o pior caso é que o elemento está na posição n, onde n é o tamanho do arranjo, por isso a notação para esse algorítmo seria $O(n)$.

[Danziger 2010] trás uma definição formal do que seria a notação Big O, e ela é dada pelo seguinte (resumidamente) : A notação Big O é uma notação matemática que

descreve o limite superior do comportamento assintótico de uma função, em termos de outra função. Formalmente, seja $f(n)$ uma função que descreve o tempo de execução de um algoritmo em termos do tamanho da entrada n. $f(n)$ é $O(g(n))$ (lido como "f de n é big O de g de n") se existem constantes positivas c e n_0 tais que, para todo n maior ou igual a n_0, $f(n)$ é menor ou igual a $c \times g(n)$. Em outras palavras:

$$f(n) \leq c \times g(n)$$

para todo $n \geq n_0$.

2.4. Arranjos estáticos

Arranjos, também conhecidos como *arrays*, são estruturas de dados que representam uma coleção de elementos de tamanho e tipo fixos. São armazenados de forma contígua na memória, facilitando o acesso aleatório ou através de um índice específico. Para os arranjos estáticos, a característica fundamental é de que o tamanho do arranjo é conhecido previamente, de modo que o mesmo é definido em tempo de compilação, impossibilitando a sua alteração durante a execução do programa.

Como mencionado anteriormente, os arranjos são armazenados de forma contígua na memória, o que significa que ocupam um bloco contínuo de endereços de memória. O que, como mencionado anteriormente, é o facilitador para o acesso aleatório de elementos do arranjo. Embora possua a vantagem de oferecer um acesso eficiente aos elementos através da utilização dos indices, suas desvantagens se dão pela impossibilidade de redimensionamento durante a execução do programa e pelo custo da busca por um valor que não se conhece o indice. Para arranjos pequenos, isso pode não ser um problema, mas caso seja necessário armazenar um grande número de dados, o arranjo acaba sendo uma das estruturas menos otimizadas.

2.4.1. Algoritmos de busca

Nesta sub seção, serão discutidos os algoritmos de busca implementados nesse artigo para o arranjo, sendo eles: Busca sequencial e busca binária.

Busca sequencial

A busca sequencial em arranjo trata-se da forma mais simples de busca. Nesse algoritmo, realiza-se um *loop* que percorre todo o arranjo e realiza comparações entre o valor que deseja ser buscado e o valor que se encontra no índice x do arranjo. Pela descrição do mesmo, pode-se imaginar que para casos em que o arranjo é pequeno não teremos muitos problemas com o tempo de execução, mas se escalarmos isso para um grande número de elementos, com certeza o tempo de execução será grande. Esse algoritmo possui complexidade de tempo de *O(n)* e complexidade de espaço auxiliar de *O(1)*, pois não é necessário utilizar nenhuma outra estrutura de dados ou arranjo auxiliar para encontrarmos o elemento desejado. O algoritmo possui complexidade de tempo de *O(n)* uma vez que, o pior caso é definido como o caso em que o elemento a ser buscado não existe ou está na última posição do arranjo, que é *n* para um arranjo de tamanho *n*. Em ambos os casos o algorítmo deve percorrer o arranjo inteiro.

Busca Binária

O algoritmo de busca binária apresenta um método diferente da busca sequencial. Ele consiste em comparar inicialmente o elemento buscado com o elemento central do conjunto, de forma que uma das metades possa ser desprezada, minimizando o escopo de busca. Nessa comparação, verifica-se se o elemento buscado é maior ou menor que o elemento central, para que a busca parta para uma das metades. Para que isso aconteça, o conjunto de elementos a ser analisado deve estar em ordem crescente. Sua complexidade é sempre O(log n)[Ziviani et al. 2004]. Para este trabalho, o algoritmo de ordenação implementado foi o *quick sort*, um dos melhores algoritmos de ordenação em arranjo.

Quick sort

O Quick Sort, como o nome sugere, trata-se de um algoritmo de ordenação que utiliza a estratégia de dividir e conquistar. No entanto, antes da divisão, é escolhido um pivô para que tudo antes dele seja menor e tudo depois dele seja maior. A cada chamada recursiva é escolhido um novo pivô e os elementos são ordenados. Ao contrário do Merge Sort, que também utiliza da estratgeia de dividir e conquistar, não é necessário combinar os subarranjos ordenados, pois estes já estão ordenados corretamente. Geralmente, o Quick Sort é a melhor opção para ordenar elementos num conjunto devido a sua velocidade[Cormen et al. 2002].

2.5. Listas ligadas

As listas ligadas tratam-se de estruturas de dados que consistem em uma sequência de elementos, onde cada elemento é armazenado em um nó, e cada nó contem um campo de valor e outro que armazena o ponteiro para o próximo nó na sequência. Por isso o nome lista ligada. Dessa forma, sua principal vantagem em comparação com o arranjo estático, é que, ao contrário do arranjo que é armazenado em um bloco específico da memória, a lista ligada pode ser armazenada em vários locais da memória, já que um elemento da lista guarda um ponteiro para o próximo elemento. Dessa forma, seu tamanho é ilimitado (obedecendo as limitações óbvias de recurso disponível) e pode ser alterado durante o tempo de execução do programa.

Existem diversos tipos de listas ligadas, onde as mais comuns são as listas simplesmente encadeada e a duplamente encadeada, onde a diferença é o ponteiro a mais guardado pela segunda, que guarda o ponteiro para o seu antecessor na lista.

Sua principal vantagem, além do tamanho que é variável, se dá na facilidade de inserção ou remoção de elementos em qualquer posição, já que é uma simples operação com ponteiros (desde que conheçamos os ponteiros de sucessor e antecessor caso exista). Isso as torna uma escolha popular para a implementação de estruturas de dados dinâmicas, como pilhas, filas e deques (deques costumam utilizar listas duplamente encadeadas).

Entretanto, ela também possui suas desvantagens. A primeira e mais óbvia se dá pelo uso a mais de memória para armazenar os ponteiros necessários para o tipo de lista implementado. A segunda trata-se da menor eficiência de acesso a elementos aleatórios da lista, uma vez que não existem indices, para encontrar um elemento aleatório devemos percorrer parte da lista ou toda ela para encontrar o elemento. Um fato curioso, é que a segunda desvantagem impossibilita a implementação de um quick sort otimizado para listas, devido ao complicado acesso aleatório de elementos. Entretanto, existem implementações

de merge sort para lista que são tão eficazes quanto.

Nesse trabalho, o algoritmo de busca utilizado é uma busca sequencial simples, similar a mencionada anteriormente para arranjos estáticos, onde ao invés de percorrermos as posições do arranjo, iremos percorrer os nós da lista ligada.

2.6. Árvores

Nesta seção serão apresentados os diferentes tipos de estruturas de dados de Árvores que foram implementadas neste trabalho.

2.6.1. Árvore de busca binária (BST - Binary Search Tree)

A árvore binária de busca, também conhecida como *Binary Search Tree (BST)*, é uma estrutura de dados em árvore[2] na qual cada nó possui no máximo dois filhos (direito e esquerdo), e todos obedecem a seguinte propriedade: para cada nó, todos os elementos que constituem a subárvore à esquerda desse nó possuem valores menores que o do nó pai, e todos os elementos da subárvore à direita possuem valores maiores que o do nó pai.

Essa propriedade da árvore binária de busca permitem uma rápida busca, inserção e exclusão de elementos, tornando as árvores binária de busca uma estrutura de dados eficientes para muitas operações. Por exemplo, ao realizarmos a busca por um elemento na árvore binária de busca, assim como na busca binária em arranjo, é possível eliminar metade dos elementos restantes a cada passo, resultando em uma busca eficiente em tempo médio de *O(log n)*, onde n é o número de elementos na árvore. No entanto, caso a árvore seja montada incorretamente, de modo que cada elemento sempre é maior ou menor que seu pai, pode gerar uma árvore degenerada, se tornando uma lista ligada. Nesse caso extremo a complexidade de busca se torna *O(n)*[Cormen et al. 2002].

2.6.2. Árvores balanceadas

A árvore balanceada é um tipo de árvore na qual as suas subárvores são mantidas relativamente equilibradas em termos de altura ou profundidade. Nesse tipo de árvore, casos como o citado anteriormente no qual a árvore possui somente uma ramificação (todos os nós possuem somente ou filhos para a esquerda ou para a direita) são impossíveis, devida à característica de balanceamento dessas árvores. As operações de balanceamento tem como objetivo a otimização do desempenho de operações como busca, inserção e remoção na árvore.

Em uma árvore balanceada, a diferença de altura entre as subárvores de cada nó é limitada por uma constante específica, garantindo que a altura permaneça próxima do mínimo possível para o número de elementos armazenados. O balanceamento dessas árvores é garantido através de operações de rotação nos nós, que serão discutidas futuramente. A seguir, discutiremos duas árvores balanceadas implementadas nesse trabalho, as árvores AVL e as árvores Rubro-negras.

[2]Estruturas de dados em árvore obedecem uma hierárquia, onde cada nó possui 2 (árvores binárias) ou mais filhos (árvores n-árias).

Árvores AVL

As árvores AVL são uma forma de árvore binária de busca balanceada, introduzida por Adelson, Velsky e Landis em 1962 [3]. O principal objetivo por trás de sua criação foi garantir um tempo de busca eficiente e evitar a deterioração do desempenho das operações de busca em arvores binárias de busca não balanceadas.

A árvore AVL é basicamente um árvore binária de busca, tendo como diferença o fato de que a diferença de altura entre as subárvores de cada nó é mantida dentro de uma faixa específica, denominada fator de balanceamento. Dessa forma, garante-se uma árvore balanceada e garante-se também que as operações de inserção, exclusão e busca sejam realizadas em tempo *O(log n)*, onde *n* é o número de elementos na árvore.

Entretanto, para garantir o balanceamento das árvores há um custo adicional durante as operações de inserção e remoção, pois é necessário verificar o balanceamento das subárvores de cada nó e se necessário, corrigir o balanceamento da árvore após essas operações. Para corrigir o balanceamento são realizadas operações que são chamadas de rotações, e podem ser: Esquerda, direita, esquerda direita e direita esquerda. A rotação na árvore AVL ocorre devido ao seu desbalanceamento, uma rotação simples ocorre quando um nó está desbalanceado e seu filho estiver no mesmo sentido da inclinação, formando uma linha reta. Uma rotação-dupla ocorre quando um nó estiver desbalanceado e seu filho estiver inclinado no sentido inverso ao pai, formando um "joelho"[4]. Apesar de aumentar o custo para inserção ou remoção, compensa isso com o fato de a árvore quando balanceada mantém o tempo de busca em *O(log n)*.

Árvores Rubro-negras (RB Tree)

As árvores rubro-negras[5] são árvores binária de busca com o diferencial no que é armazenado em cada nó. Nas árvores rubro-negras cada nó é constituído dos seguintes campos: cor (podendo ser vermelho ou preto), valor, ponteiro para filho esquerdo, ponteiro para filho direito, ponteiro para o nó pai. A necessidade do ponteiro para o nó pai se dá para facilitar a operação da árvore rubro-negra.

Uma árvore rubro-negra é uma árvore binária de busca, com algumas propriedades adicionais. Quando um nó não possui um filho (esquerdo ou direito) então vamos supor que ao invés de apontar para nil, ele aponta para um nó fictício, que será uma folha da árvore. Assim, todos os nós internos contêm chaves e todas as folhas são nós fictícios. Suas principais propriedades são:

- Todo nó da árvore ou é vermelho ou é preto;
- A raiz é preta;
- Toda folha (nil) é preta;
- Se um nó é vermelho, então ambos os filhos são pretos;
- Para todo nó, todos os caminhos do nó até as folhas descendentes contêm o mesmo número de nós pretos.

Assim como nas árvores AVL, necessita da realização de rotações (nesse caso, rotação esquerda ou rotação direita) para manter a árvore balanceada no caso de operações

[3] O nome desse tipo de árvore é dado em homenagem a seus autores, por isso AVL

[4] Wikipedia, Árvore AVL

[5] IME-USP: Árvore rubro-negra: `https://www.ime.usp.br/~song/mac5710/slides/08rb.pdf`

de inserção ou remoção. Em comparação com a árvore AVL, as operações de balanceamento na árvore rubro-negra são mais complexas. Entretanto, o número de rotações é menor. Ambas são excelentes para pesquisa, pois sempre mantém a complexidade de busca em *O(log n)*· Um fato curioso é que as árvores rubro-negra são de uso mais geral do que as árvores AVL, sendo utilizada em diversas aplicações e bibliotecas em linguagens de programação, como por exemplo:

- Java: java.util.TreeMap, java.util.TreeSet
- C++ STL: Map, multimap, multiset
- Linux kernel: Completely fair scheduler

3. Materiais e Métodos

Nesta seção serão detalhados os materiais utilizados e os métodos empregados na implementação e avaliação dos diferentes métodos de busca em diferentes estruturas de dados.

3.1. Materiais utilizados

Para a realização desse projeto, foi utilizado um computador de mesa com as seguintes especificações:

- Sistema operacional: Elementary OS versão 7, kernel similar ao ubuntu 23.04;
- Processador: Intel core i5 11400F, *undervolt* de 150mv
- Memória: 32 GBytes DDR4 2666Mhz *overclockadas* para 3200Mhz com timings customizados.
- Ambiente de desenvolvimento integrado (IDE): Visual studio code
- Linguagem de programação: *Python* 3.12
- Conjunto de dados: Números inteiros que variam o tamanho de 100 mil a 1 milhão de elementos únicos.
- Bibliotecas utilizadas: random, os, psutil, time, numpy

O código fonte e os materiais utilizados nesse projeto podem ser encontrados no seguinte repositório do GitHub[6]

3.2. Geração dos casos de teste

Para avaliar a eficiência dos algoritmos de busca, os casos de teste gerados variam o tamanho do arranjo de números de 100 mil a 1 milhão de elementos, e estes elementos variam também conforme o intervalo determinado dos números. Por exemplo, no primeiro caso de teste, os elementos vão variar de 0 à 100 mil, para o segundo, de 0 à 200 mil, e assim por diante. Todos os algoritmos foram testados nos mesmos cenários, para garantir uma comparação justa.

3.3. Dados coletados

Para cada método de busca e para cada cenário de teste, foram coletados os seguintes dados:

- Número de comparações realizadas durante a busca.
- Tempo de execução do algoritmo em milisegundos (ms).
- Consumo de memória durante a execução em Mbytes.

Esses dados foram coletados para análise posterior, incluindo o cálculo da média e o desvio padrão para as métricas coletadas.

[6]Github Repository: `https://github.com/hisck/Trabalho-EDAA`

3.4. Considerações adicionais

Nos testes realizados, foram descartados custos de criação das estruturas, custo de inserção de dados nas mesmas além do custo de ordenação para o caso da busca binária em arranjo. Adicionalmente, para que a medida em tempo ficasse mais ”palpável”, foi adicionado um sleep em cada iteração de cada algoritmo de busca de 0.000001 segundos, ou 1 micro segundo.

4. Resultados

Nesta seção serão discutidos os resultados obtidos pelos testes. O pior cenário para todos os casos foi adotado como um valor que não existe nos elementos inseridos nas estruturas, entretanto, para os casos de árvores, ou algoritmos que geram árvores, como a busca binária, veremos que não foi essencialmente o pior caso, pois ocorreu um desvio padrão. De qualquer forma, foi possível averiguar uma piora no tempo e no número de comparações realizados nos piores casos.

As tabelas 1 e 2 apresentam os resultados para os casos de arranjo simples com busca sequencial. Valores com o * são valores estimados, porque por algum motivo especial para esses valores o sistema não terminou a simulação.

Tam. Teste	Média tempo (ms)	Std. Tempo (ms)	Média Comparações	Std. Comparações	Média Mem. (Mbytes)	Std. Mem. (Mbytes)
100 mil	2.774	1.50	52187.96	28382.34	0.0	0.0
200 mil	4.156	2.92	78128.83	55054.43	0.0	0.0
300 mil	5.41	4.09	101735.72	76865.85	0.0	0.0
400 mil	6.71	5.15	126078.35	96851.57	0.0	0.0
500 mil	8.16	6.5	153384.328	122981.38	0.0	0.0
600 mil	9.42	7.63	176944.43	143235.35	0.0	0.0
700 mil	11.21	9.10	210417.76	170714.63	0.0	0.0
800 mil	12.50	10.13	234614.53	190243.24	0.0	0.0
900 mil	13.84	11.19	259532.67	209592.70	0.0	0.0
1 mi.	15.00*	12.00*	300000.00*	230000.00*	0.0*	0.0*

Tabela 1. Tabela 1: Resultados para o caso médio da busca simples em arranjo

Tam. Teste	Média tempo (ms)	Std. Tempo (ms)	Média Comparações	Std. Comparações	Média Mem. (Mbytes)	Std. Mem. (Mbytes)
100 mil	5.32	0.01	100000.0	0	0.0	0.0
200 mil	7.98	2.66	150000.0	50000.0	0.0	0.0
300 mil	10.66	4.35	200000.0	81649.658	0.0	0.0
400 mil	13.32	5.95	250000.0	111803.398	0.0	0.0
500 mil	15.99	7.54	300000.0	141421.356	0.0	0.0
600 mil	18.66	9.11	350000.0	170782.512	0.0	0.0
700 mil	21.32	10.66	400000.0	200000.0	0.0	0.0
800 mil	24.00	12.24	450000.0	229128.78	0.0	0.0
900 mil	26.69	13.82	500000.0	258198.88	0.0	0.0
1 mi.	30.00*	15.00*	550000.00*	300000.00*	0.0*	0.0*

Tabela 2. Tabela 2: Resultados para o pior caso da busca simples em arranjo

As tabelas 3 e 4 apresentam os resultados para os casos de arranjo simples com busca binária. É importante destacar esse método, pois entre os métodos para estruturas de dados lineares, a busca binária em arranjo simples possuiu a melhor performance geral.

As tabelas 5 e 6 apresentam os resultados para os casos de busca sequencial em uma lista ligada. Valores com o * são valores estimados, porque por algum motivo especial para esses valores o sistema não terminou a simulação.

A tabela 7 apresenta o resultado para o caso médio da árvore binária de busca. Não foi possível apresentar o pior caso para a árvore binária de busca pois o mesmo consiste

Tam. Teste	Média tempo (ms)	Std. Tempo (ms)	Média Comparações	Std. Comparações	Média Mem. (Mbytes)	Std. Mem. (Mbytes)
100 mil	0.9	0.16	15.71	1.81	0.03	0.154
200 mil	0.9	0.12	16.26	1.69	0.026	0.16
300 mil	0.9	0.11	16.55	1.65	0.027	0.19
400 mil	0.96	0.11	16.85	1.66	0.027	0.22
500 mil	0.98	0.11	17	1.69	0.03	0.31
600 mil	1.0	0.15	17.2	1.72	0.031	0.28
700 mil	1.0	0.14	17.44	1.74	0.033	0.32
800 mil	1.0	0.14	17.62	1.75	0.044	0.428
900 mil	1.0	0.136	17.77	1.76	0.039	0.404
1 mi.	1.0	0.136	17.89	1.77	0.05	0.600

Tabela 3. Tabela 3: Resultados para o caso médio da busca binária em arranjo

Tam. Teste	Média tempo (ms)	Std. Tempo (ms)	Média Comparações	Std. Comparações	Média Mem. (Mbytes)	Std. Mem. (Mbytes)
100 mil	0.9	0.16	17	0	0.0	0.0
200 mil	0.9	0.12	17.5	0.5	0.0	0.0
300 mil	1.0	0.03	18.0	0.816	0.0	0.0
400 mil	1.0	0.03	18.25	0.82	0.0	0.0
500 mil	1.0	0.03	18.4	0.79	0.0	0.0
600 mil	1.0	0.05	18.66	0.94	0.0	0.0
700 mil	1.0	0.06	18.85	0.98	0.0	0.0
800 mil	1.0	0.06	19.0	1.0	0.0	0.0
900 mil	1.0	0.06	19.1	0.99	0.0	0.0
1 mi.	1.0	0.06	19.2	0.97	0.0	0.0

Tabela 4. Tabela 4: Resultados para o pior caso da busca binária em arranjo

Tam. Teste	Média tempo (ms)	Std. Tempo (ms)	Média Comparações	Std. Comparações	Média Mem. (Mbytes)	Std. Mem. (Mbytes)
100 mil	2.27	1.37	42922.62	25833.79	0.16	1.523
200 mil	3.64	2.71	68486.58	51028.14	0.14	1.387
300 mil	5.19	4.06	97563.53	76293.33	0.12	1.248
400 mil	6.65	5.34	125105.91	100432.76	0.11	1.127
500 mil	8.01	6.52	150634.436	122532.41	0.09	1.018
600 mil	9.65	7.84	181251.76	147264.91	0.08	0.930
700 mil	11.14	9.24	209237.62	173428.94	0.067	0.872
800 mil	12.51	10.25	234966.55	192482.90	0.051	0.845
900 mil	13.75	11.22	258063.24	210186.48	0.0347	0.849
1 mi.	15.00*	12.00*	300000.00*	230000.00*	0.04*	0.899*

Tabela 5. Tabela 5: Resultados para o caso médio da busca simples em lista

Tam. Teste	Média tempo (ms)	Std. Tempo (ms)	Média Comparações	Std. Comparações	Média Mem. (Mbytes)	Std. Mem. (Mbytes)
100 mil	5.31	0.00	100000.0	0.00	0.0	0.0
200 mil	7.99	2.67	150000.0	50000.0	0.0	0.0
300 mil	10.65	4.348	200000.0	81649.65	0.0	0.0
400 mil	13.31	5.95	250000.0	111803.39	0.0	0.0
500 mil	15.97	7.52	300000.0	141421.35	0.0	0.0
600 mil	18.63	9.09	350000.0	170782.51	0.0	0.0
700 mil	21.30	10.66	400000.0	200000.0	0.0	0.0
800 mil	24.011	12.27	450000.0	229128.78	0.0	0.0
900 mil	26.697	13.84	500000.0	258198.88	0.0	0.0
1 mi.	30.00*	15.00*	550000.00*	300000.00*	0.0*	0.0*

Tabela 6. Tabela 6: Resultados para o pior caso da busca simples em lista

essencialmente em gerar uma lista ligada através da utilização de uma árvore binária de busca. Para realizar isso, seria necessário um grande número de recursão, o que causaria um estouro da pilha por alcançar o limite máximo de recursão da linguagem utilizada *Python*.

As tabelas 8 e 9 apresentam os resultados para os casos de busca em uma árvore rubro-negra.

As tabelas 10 e 11 apresentam os resultados para os casos de busca em uma árvore AVL. Interessante notar como os resultados de ambas as árvores são similares, com pe-

Tam. Teste	Média tempo (ms)	Std. Tempo (ms)	Média Comparações	Std. Comparações	Média Mem. (Mbytes)	Std. Mem. (Mbytes)
100 mil	1.20	0.2	49.77	12.61	0.16	1.523
200 mil	1.31	0.3	54.71	13.72	0.14	1.387
300 mil	1.33	0.3	55.1	12.86	0.12	1.248
400 mil	1.35	0.3	56.1	13.11	0.11	1.127
500 mil	1.37	0.3	57	13.38	0.09	1.018
600 mil	1.39	0.3	57.4	13.30	0.08	0.930
700 mil	1.39	0.3	57.8	13.26	0.067	0.872
800 mil	1.40	0.3	58.3	13.12	0.051	0.845
900 mil	1.41	0.3	59	13.29	0.0359	0.855
1 mi.	1.43	0.3	60	13.88	0.0203	0.899

Tabela 7. Tabela 7: Resultados para o caso médio da árvore binária de busca

Tam. Teste	Média tempo (ms)	Std. Tempo (ms)	Média Comparações	Std. Comparações	Média Mem. (Mbytes)	Std. Mem. (Mbytes)
100 mil	0.96	0.1	24.83	3.49	0.16	3.77
200 mil	0.98	0.1	24.875	3.819	0.18	3.81
300 mil	1.0	0.1	25.62	3.78	0.16	4.29
400 mil	1.0	0.1	26.05	3.611	0.166	4.66
500 mil	1.0	0.1	26.39	3.50	0.14	5.72
600 mil	1.05	0.1	26.82	3.62	0.12	6.92
700 mil	1.06	0.1	27.10	3.71	0.109	7.83
800 mil	1.07	0.1	27.4	3.73	0.108	8.85
900 mil	1.07	0.1	27.58	3.750	0.087	11.47
1 mi.	1.08	0.1	27.79	3.77	0.099	12.50

Tabela 8. Tabela 8: Resultados para o caso médio da árvore Rubro-negra

Tam. Teste	Média tempo (ms)	Std. Tempo (ms)	Média Comparações	Std. Comparações	Média Mem. (Mbytes)	Std. Mem. (Mbytes)
100 mil	1.06	0.08	35.0	0	1.33	5.24
200 mil	1.04	0.02	34.0	1	0.48	5.50
300 mil	1.03	0.02	33.66	0.94	0.130	4.5
400 mil	1.05	0.04	34.5	1.65	0.647	5.07
500 mil	1.1	0.09	36.2	3.70	0.579	4.54
600 mil	1.12	0.1	37.0	3.8	0.481	6.05
700 mil	1.12	0.09	37.28	3.61	0.0169	9.353
800 mil	1.14	0.09	37.75	3.59	0.611	11.36
900 mil	1.14	0.09	37.88	3.413	0.043	10.94
1 mi.	1.14	0.08	37.8	3.24	0.979	12.55

Tabela 9. Tabela 9: Resultados para o pior caso da árvore Rubro-negra

quenas vantagens para a árvore AVL, uma vez que a mesma possui regras mais restritas, balanceando a árvore um número maior de vezes em comparação com a árvore rubro-negra. Mesmo assim, apesar dos pequenos ganhos, um fator interessante que não é apresentado nesses valores é que a criação de uma árvore AVL possui um custo muito maior que uma árvore rubro-negro, de modo que o tempo total de execução dos testes foi consideravelmente maior quando considerado com os testes da árvore rubro-negro.

Tam. Teste	Média tempo (ms)	Std. Tempo (ms)	Média Comparações	Std. Comparações	Média Mem. (Mbytes)	Std. Mem. (Mbytes)
100 mil	0.96	0.08	24.41	3.02	0.16	1.5
200 mil	0.98	0.09	24.985	3.16	0.14	1.394
300 mil	1.0	0.09	25.55	3.28	0.127	1.25
400 mil	1.0	0.09	25.87	3.3	0.111	1.12
500 mil	1.0	0.09	26.26	3.34	0.09	1.01
600 mil	1.0	0.1	26.67	3.44	0.08	0.930
700 mil	1.05	0.1	26.89	3.49	0.06	0.87
800 mil	1.06	0.1	27.06	3.51	0.05	0.84
900 mil	1.06	0.1	27.33	3.52	0.03	0.85
1 mi.	1.07	0.1	27.48	3.52	0.019	0.899

Tabela 10. Tabela 10: Resultados para o caso médio da árvore AVL

5. Conclusão

Com base nos resultados dos testes realizados para diferentes métodos de buscas em diferentes estruturas de dados, podemos inferir como os algoritmos operam. Todos os casos corroboram com a notação de Big O, demonstrando que a quantidade de comparações aumenta na medida que o tamanho da entrada aumenta. Para os casos de busca sequencial, as comparações aumentaram linearmente, enquanto que nos casos que o algoritmo gera uma espécie de árvore, ou é uma árvore, obedece o logaritmo.

Os testes nos confirmam que a escolha do método ideal para uma aplicação específica depende da implementação e da criticidade da aplicação. Obviamente que não é o ideal utilizar a busca sequencial, mas dependendo do caso, podemos optar por uma busca binária, caso seja possível perder um pequeno tempo com a ordenação desse arranjo. Podemos também observar como os algoritmos que geram árvores ou são árvores possuem tempos similares, onde a árvore AVL possuiu o menor tempo, mas a custa de um maior tempo para geração da árvore, inclusão e remoção de elementos.

Podemos portanto concluir que a escolha da estrutura de dados influencia diretamente no método de busca. Entretanto, devemos ressaltar que as análises e conclusões apresentadas nesse relatório são derivadas dos resultados obtidos nos testes conduzidos. É fundamental reconhecer que essas conclusões podem não ser generalizadas para todas as situações, uma vez que o desempenho e a eficácia dessas estruturas podem variar conforme as características do ambiente de implementação e do problema a ser resolvido.

Referências

Bae, S. (2019). *Big-O Notation*, pages 1–11. Apress, Berkeley, CA.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2002). Algoritmos: teoria e prática. *Editora Campus*, 2:296.

Danziger, P. (2010). Big o notation. *Source internet: http://www. scs. ryerson. ca/mth110/Handouts/PD/bigO. pdf, Retrieve: April*, 1(1):6.

Ziviani, N. et al. (2004). *Projeto de algoritmos: com implementações em Pascal e C*, volume 2. Thomson Luton.

Tam. Teste	Média tempo (ms)	Std. Tempo (ms)	Média Comparações	Std. Comparações	Média Mem. (Mbytes)	Std. Mem. (Mbytes)
100 mil	0.94	0.01	31	3.02	0.0	0.0
200 mil	1.0	0.06	33	2.0	0.0	0.0
300 mil	1.05	0.09	35	3.26	0.0	0.0
400 mil	1.06	0.08	35	2.82	0.0	0.0
500 mil	1.08	0.08	35.4	2.65	0.0	0.0
600 mil	1.1	0.1	36.66	3.72	0.0	0.0
700 mil	1.1	0.1	37.28	3.76	0.0	0.0
800 mil	1.14	0.1	38.0	4.0	0.0	0.0
900 mil	1.15	0.1	38.33	3.88	0.0	0.0
1 mi.	1.16	0.1	38.6	3.77	0.0	0.0

Tabela 11. Tabela 11: Resultados para o pior caso da árvore AVL