Análise de Algoritmos de Busca Sequencial e Binária em Estruturas de dados lineares e não lineares

Gabriel Ranulfo Branquinho Cirillo

[1]Mestrado em Ciência da Computação
Universidade Estadual do Oeste do Paraná (Unioeste) – Cascavel, PR – Brasil

gabriel.cirillo@unioeste.br

Abstract. *The objetive this paper is to compare and identify the advantages and disadvantages of linear and non-linear data structures, as well as their efficiency in sequential and binary search methods. The tests consider the worst-case scenarios for each scenario, along with random search cases, with the aim of diversifying the tests for reproducibility in other experiments. Finally, we conclude with the identified results, as well as the algorithm best suggested for each situation.*

Resumo. *O objetivo deste trabalho é comparar e identificar vantagens e desvantagens de estruturas de dados lineares e não-lineares e sua eficiência em métodos de busca sequencial e binária. Os testes consideram os piores casos para cada cenário, além de casos aleatórios de busca, com o interesse de diversificar os testes para que possam ser reproduzidos em outros experimentos. Por fim, concluímos com os resultados identificados, bem como o algoritmo melhor sugerido para cada situação.*

1. Introdução

A eficiência na manipulação e processamento de dados é fundamental em diversas aplicações computacionais. A escolha entre arranjos estáticos e estruturas de dados lineares e não-lineares pode influenciar significativamente o desempenho de algoritmos, especialmente em operações como busca e ordenação. Este projeto visa investigar e comparar a eficiência de diferentes métodos de busca em arranjos estáticos e estruturas de dados, considerando valores inteiros positivos.

Ao analisar os arranjos estáticos, dois métodos de busca são estudados: busca sequencial padrão e busca binária. Por outro lado, nas estruturas lineares e não-lineares, concentramo-nos na busca sequencial em lista ligada e busca em árvores binárias de busca. A comparação desses métodos permitirá uma compreensão mais aprofundada de suas vantagens e desvantagens em diferentes contextos de problema.

Para avaliar a eficiência dos algoritmos de busca, serão gerados casos de teste variando o tamanho do arranjo de 100 mil a 1 milhão de elementos únicos, este elementos variando entres os valores de 1 a 120 milhões, por sua vez, separados em arquivos de intervalos de 100 mil. Todos os algoritmos serão testados nos mesmos cenários para garantir uma comparação justa. As métricas de desempenho consideradas incluem a média e desvio padrão para o número de comparações, tempo de execução e consumo de memória.

Diversos cenários serão explorados, incluindo o pior caso, com 3 execuções de cada algoritmo, e casos aleatórios com 100 buscas para cada cenário. É importante ressaltar que o custo de criação das estruturas será descartado, focando apenas no desempenho das operações de busca, nenhum dos nossos testes irá considerar listas ordenadas.

Com essas considerações em mente, este projeto busca fornecer *insights* valiosos sobre a escolha e aplicação adequadas de estruturas de dados e algoritmos de busca em diferentes contextos computacionais.

2. Fundamentação Teórica

2.1. Estruturas de Dados Lineares e Não Lineares

As estruturas de dados são formas de organizar e armazenar dados de maneira eficiente para facilitar a manipulação e o acesso a eles. Existem dois tipos principais de estruturas de dados: lineares e não lineares.

2.1.1. Estruturas de Dados Lineares

As estruturas de dados lineares são aquelas em que os elementos são organizados de forma sequencial, ou seja, cada elemento tem um sucessor e, opcionalmente, um antecessor. Exemplos comuns de estruturas de dados lineares incluem:

- Listas
- Pilhas
- Filas
- Deques

Nessas estruturas, a ordem dos elementos é importante e as operações de inserção e remoção geralmente ocorrem em uma extremidade da estrutura.

2.1.2. Estruturas de Dados Não Lineares

As estruturas de dados não lineares são aquelas em que os elementos não são organizados de forma sequencial. Em vez disso, os elementos são organizados de maneira hierárquica, formando relações complexas entre si. Exemplos comuns de estruturas de dados não lineares incluem:

- Árvores
- Grafos
- Tabelas de espalhamento (*Hash Tables*)

Nessas estruturas, a ordem dos elementos pode não ser tão clara, e as operações de inserção e remoção podem ocorrer em diferentes partes da estrutura, dependendo da implementação específica.

Em resumo, as estruturas de dados lineares são aquelas em que os elementos são organizados de forma sequencial, enquanto as estruturas de dados não lineares são aquelas em que os elementos são organizados de forma hierárquica e não sequencial. Ambos os tipos de estruturas têm suas próprias vantagens e aplicações, e a escolha entre eles depende das necessidades específicas do problema em questão.

2.2. A notação Big O

A notação Big O informa quão eficiente é um algoritmo em termos de escalabilidade. Por exemplo, se temos uma lista de tamanho n, e o tempo de execução do algoritmo é $O(n)$, isso significa que o tempo de execução do algoritmo cresce linearmente com o tamanho da lista. No entanto, a notação Big O não fornece uma medida direta do tempo em segundos. Em vez disso, ela permite comparar o número de operações realizadas pelo algoritmo para diferentes tamanhos de entrada. Essa notação nos ajuda a entender como o desempenho do algoritmo se comporta à medida que o tamanho da entrada aumenta [Cormen et al. 2012].

2.3. Arranjos Estáticos

Arranjos estáticos, também conhecidos como *arrays* estáticos, são estruturas de dados que representam uma coleção de elementos de tamanho fixo e contíguos na memória. Eles são usados para armazenar dados do mesmo tipo em uma sequência ordenada, onde cada elemento pode ser acessado por meio de um índice específico.

Uma característica fundamental dos arranjos estáticos é que o tamanho do arranjo é definido em tempo de compilação e não pode ser alterado durante a execução do programa. Isso significa que o número máximo de elementos que o arranjo pode armazenar é conhecido antecipadamente e não pode ser excedido sem alterar o código-fonte e recompilar o programa.

Os elementos de um arranjo estático são armazenados de forma contígua na memória, o que significa que ocupam um bloco contínuo de endereços de memória. Isso facilita o acesso aleatório aos elementos do arranjo, pois cada elemento pode ser acessado diretamente por meio de seu índice, sem a necessidade de percorrer a estrutura de dados.

Embora os arranjos estáticos ofereçam acesso eficiente aos elementos e operações de busca rápida, sua principal limitação é a incapacidade de redimensionar dinamicamente o tamanho do arranjo durante a execução do programa. Isso pode ser problemático em situações onde o número de elementos a serem armazenados não é conhecido antecipadamente ou pode variar ao longo do tempo.

Em resumo, os arranjos estáticos são estruturas de dados simples e eficientes para armazenar uma coleção de elementos de tamanho fixo, oferecendo acesso rápido aos elementos por meio de índices. No entanto, sua inflexibilidade em relação ao tamanho pode limitar sua aplicabilidade em certos cenários.

2.4. Listas Ligadas

Uma lista ligada é uma estrutura de dados que consiste em uma sequência de elementos, onde cada elemento é armazenado em um nó e cada nó contém um campo de dados e um ou mais ponteiros que apontam para o próximo nó na sequência. Ao contrário dos arranjos estáticos, onde os elementos são armazenados de forma contígua na memória, os nós de uma lista ligada podem estar dispersos pela memória e são conectados por meio de ponteiros.

Existem vários tipos de listas ligadas, sendo as mais comuns as listas ligadas simples, onde cada nó possui apenas um ponteiro que aponta para o próximo nó na sequência,

e as listas ligadas duplamente encadeadas, onde cada nó possui um ponteiro que aponta para o próximo nó e outro ponteiro que aponta para o nó anterior na sequência.

Uma das principais vantagens das listas ligadas é a capacidade de inserir e remover elementos em qualquer posição da lista com complexidade de tempo O(1), desde que seja conhecido o ponteiro para o nó anterior ou posterior ao nó a ser inserido ou removido. Isso torna as listas ligadas uma escolha popular para implementar estruturas de dados dinâmicas, como pilhas, filas e listas encadeadas.

No entanto, as listas ligadas também têm algumas desvantagens. O acesso aleatório aos elementos de uma lista ligada é menos eficiente do que em arranjos estáticos, pois requer percorrer a lista a partir do início ou de um nó conhecido até o nó desejado. Além disso, o uso de ponteiros adicionais pode aumentar o consumo de memória em comparação com arranjos estáticos.

Em resumo, as listas ligadas são estruturas de dados flexíveis e eficientes para armazenar uma sequência de elementos, permitindo inserções e remoções rápidas em qualquer posição da lista. No entanto, seu acesso aleatório menos eficiente e o consumo potencialmente maior de memória podem limitar sua aplicabilidade em certos contextos.

2.5. Busca Sequencial Padrão

A busca sequencial padrão é um método simples para encontrar um elemento em uma lista. Ele percorre cada elemento da lista, um por um, até encontrar o elemento desejado ou até o final da lista ser alcançado. A complexidade desse método é linear, o que significa que o tempo de execução cresce proporcionalmente ao tamanho da lista. A complexidade deste algoritmo é $O(n)$, onde n é o número de elementos na lista. Isso ocorre porque, no pior caso, pode ser necessário percorrer todos os n elementos da lista para encontrar o elemento desejado [Bhargava 2016].

2.6. Busca Binária

A busca binária é um algoritmo eficiente para encontrar um elemento específico em um arranjo ordenado. Ele opera dividindo repetidamente o arranjo ao meio e comparando o elemento alvo com o elemento central da sub lista atual. Dependendo do resultado da comparação, o algoritmo decide em qual metade do arranjo continuar a busca. Esse processo é repetido até que o elemento desejado seja encontrado ou a sub lista se torne vazia. A complexidade da busca binária é $O(\log n)$, onde n é o número de elementos no arranjo [Dasgupta et al. 2006].

2.7. Busca Sequencial em Lista Ligada

A busca sequencial em uma lista ligada é um método para encontrar um elemento específico em uma lista encadeada, que é uma estrutura de dados na qual cada elemento (nó) contém um valor e uma referência (ou ponteiro) para o próximo elemento na lista.

Na busca sequencial, percorremos a lista ligada começando pelo primeiro nó e avançando para o próximo nó até encontrar o elemento desejado ou chegar ao final da lista (quando o próximo nó é nulo). Durante o processo de busca, cada nó é examinado para verificar se seu valor corresponde ao valor que está sendo buscado [Cormen et al. 2012].

A complexidade da busca sequencial em uma lista ligada é linear, ou seja, $O(n)$, onde n é o número de elementos na lista. Isso ocorre porque, em caso de pior cenário,

precisamos percorrer todos os elementos da lista até encontrar o elemento desejado ou chegar ao final da lista.

Embora a busca sequencial em lista ligada seja simples de implementar, ela pode ser menos eficiente do que outros métodos de busca, especialmente em listas longas, devido à necessidade de percorrer todos os elementos da lista. Em contrapartida, oferece a vantagem de não exigir que a lista esteja ordenada.

2.8. Árvore Binária de Busca

Uma árvore binária de busca *(BST - Binary Search Tree)* é uma estrutura de dados em árvore na qual cada nó tem no máximo dois filhos, com a seguinte propriedade: para cada nó, todos os elementos na subárvore à esquerda desse nó são menores que o valor armazenado no nó e todos os elementos na subárvore à direita são maiores.

Essa propriedade permite a rápida busca, inserção e exclusão de elementos na árvore, tornando as árvores binárias de busca uma estrutura de dados eficiente para muitas operações. Por exemplo, ao buscar um elemento em uma árvore binária de busca, 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, em casos extremos, a árvore pode degenerar em uma lista vinculada, resultando em uma complexidade de busca de $O(n)$ [Cormen et al. 2012].

2.9. Árvores Balanceadas

Uma árvore balanceada é uma estrutura de dados que mantém suas subárvores relativamente equilibradas em termos de altura ou profundidade. Essa característica de balanceamento tem como objetivo otimizar o desempenho de operações como busca, inserção e remoção na árvore.

Em uma árvore balanceada, a diferença de altura entre as subárvores esquerda e direita de cada nó é limitada por uma constante específica, garantindo que a altura da árvore permaneça próxima do mínimo possível para o número de elementos armazenados. Isso evita casos extremos de desbalanceamento, nos quais a árvore poderia degenerar em uma estrutura linear, reduzindo assim a eficiência das operações.

O balanceamento das árvores é fundamental para garantir um desempenho previsível e eficiente das operações realizadas sobre elas. Ao manter a altura da árvore em um nível próximo ao mínimo possível, as operações de busca, inserção e remoção têm complexidade temporal controlada, o que é crucial em uma ampla variedade de aplicações computacionais.

2.10. Árvores AVL

As árvores AVL são uma forma de árvore binária de busca balanceada, introduzida por Adelson-Velsky e Landis em 1962. Elas são denominadas AVL em homenagem aos seus inventores. O principal objetivo por trás da criação das árvores AVL foi garantir um tempo de pesquisa eficiente e evitar a deterioração do desempenho das operações de busca em árvores binárias de busca não balanceadas.

Uma árvore AVL é uma árvore binária de busca em que a diferença de altura entre as subárvores esquerda e direita de cada nó, chamada de fator de balanceamento,

é mantida dentro de uma faixa específica. Mais formalmente, em uma árvore AVL, para cada nó, a diferença entre as alturas da subárvore esquerda e da subárvore direita é no máximo 1. Se essa diferença exceder 1, a árvore é rebalanceada por meio de rotações.

A propriedade de balanceamento das árvores AVL garante que as operações de inserção, exclusão e pesquisa possam ser realizadas em tempo $O(\log n)$, onde n é o número de elementos na árvore. Isso é possível porque a altura da árvore é mantida em um nível controlado, o que impede que a árvore se torne desequilibrada e degenerada em uma lista encadeada, o que resultaria em operações de busca com complexidade $O(n)$.

Um ponto importante a ser destacado é que a garantia de balanceamento das árvores AVL implica em um custo adicional durante as operações de inserção e exclusão, pois é necessário verificar e, se necessário, corrigir o balanceamento da árvore após essas operações. No entanto, esse custo adicional é compensado pela garantia de um desempenho consistente das operações de busca ao longo do tempo.

Portanto, as árvores AVL são uma ferramenta fundamental em computação, especialmente em estruturas de dados onde a eficiência das operações de busca é crítica. Elas fornecem uma solução elegante e eficaz para manter o balanceamento em árvores binárias de busca, garantindo um desempenho previsível e eficiente em uma ampla gama de cenários de aplicação.

3. Materiais e Métodos

Neste capítulo, serão detalhados os materiais utilizados e os métodos empregados na implementação e avaliação dos diferentes métodos de busca em arranjos estáticos e estruturas de dados lineares e não-lineares.

3.1. Materiais Utilizados

Para a realização deste projeto, foram utilizados os seguintes materiais:

- *Notebook Dell Inspiron 5406 2-in-1,* executando *Microsoft Windows 11 Home Single Language (x64),* Versão *23H2, Build 22631.3527,* equipado com um processador *Intel(R) Core(TM) i3-1115G4* de *11ª* geração *@ 3.00GHz e 16 GBytes* de memória RAM DDR4.
- Ambiente de desenvolvimento integrado (IDE): *Visual Studio Code* (versão 1.89.12).
- Linguagem de programação: *Python* 3.10.9.
- Conjunto de dados gerados para os testes, variando o tamanho do arranjo de 100 mil a 1 milhão de elementos únicos.
- Bibliotecas utilizadas: *csv, random, os, psutil, tracemalloc* e *time.*
- Estruturas de dados específicas e métodos de busca implementados em estruturas lineares e não-lineares, como listas ligadas e árvores binárias de busca.

O código fonte e os materiais utilizados neste projeto podem ser encontrados no seguinte repositório do GitHub `https://github.com/gabrielranulfo/Avaliacao-EDAA`.

3.2. Geração dos Casos de Teste

Para avaliar a eficiência dos algoritmos de busca, os casos gerados de teste variam o tamanho do arranjo de 100 mil a 1 milhão de elementos únicos, este elementos estão entres os valores de 1 a 120 milhões, por sua vez, separados em arquivos de intervalos de 100 mil. Todos os algoritmos foram testados nos mesmos cenários para garantir uma comparação justa.

3.3. Coleta de Dados

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 microsegundos (µs).
- Consumo de memória durante a execução em *bytes*.

Os dados foram registrados para análise posterior, incluindo o cálculo da média e do desvio padrão para as métricas coletadas.

3.4. Considerações Adicionais

Além dos métodos de busca propriamente ditos, foram descartados custo de criação das estruturas, custo de criação dos arquivos e custo de ordenação.

Este capítulo detalhou os materiais utilizados e os métodos empregados na implementação e avaliação dos métodos de busca estudados neste projeto. Os próximos capítulos abordarão os resultados obtidos e sua análise.

4. Resultados

Os resultados dos testes de pior cenário para listas ligadas são apresentados na Tabela 1. Observa-se que a média e o desvio padrão da memória permanecem em torno de 353 e 211 *bytes*, respectivamente. Isso pode ser atribuído a diversas razões, como otimizações do compilador, gerenciamento de memória do sistema operacional, ou mesmo da linguagem de programação utilizada.

É possível notar que, neste teste de pior cenário para a busca sequencial, a quantidade de comparações é extremamente grande, pois o algoritmo percorre cada elemento da lista até o final, validando sua complexidade de $O(n)$, onde n é a quantidade de elementos de cada lista.

O tempo necessário para percorrer a lista é linear em relação à quantidade de comparações realizadas neste cenário, e, por conseguinte, também é elevado.

Arranjo	Média Tempo (µs)	Std. Tempo (µs)	Média Comparações	Std. Comparações	Média Memória (*bytes*)	Std. Memória (*bytes*)
100000	86508	884	100000	0	1049	1195
200000	258075	43884	200000	0	353	211
300000	385067	38774	300000	0	353	211
400000	501006	51902	400000	0	353	211
500000	574761	58635	500000	0	353	211
600000	670253	26537	600000	0	353	211
700000	789230	41023	700000	0	353	211
800000	1016517	135664	800000	0	353	211
900000	1015882	30018	900000	0	353	211
1000000	1002419	125873	1000000	0	353	211

Tabela 1. Busca Sequencial em Lista Ligada para Pior Cenário

Os testes de busca em listas ligadas para 100 casos aleatórios são apresentados na Tabela 2. Observa-se que, seguindo o padrão dos testes de busca sequencial, a média e o desvio padrão da memória permanecem estáveis, com 246 e 147 *bytes*, respectivamente, devido ao número reduzido de comparações.

É possível observar que, neste teste de busca aleatória, a quantidade de comparações apresenta alta variação e desvio padrão, pois o algoritmo percorre cada elemento da lista até identificar e encontrar o elemento procurado.

O tempo tem comportamento semelhante ao teste anterior, porém, é razoavelmente mais baixo, uma vez que, neste cenário, os números foram identificados, finalizando a busca antes de ler todos os elementos.

Arranjo	Média Tempo (µs)	Std. Tempo (µs)	Média Comparações	Std. Comparações	Média Memória (*bytes*)	Std. Memória (*bytes*)
100000	52472	38750	52403	27933	246	147
200000	115676	64462	99001	55125	246	147
300000	239854	163339	168166	82716	246	147
400000	244306	127676	218675	109161	249	148
500000	263613	173555	221442	145631	246	147
600000	302448	175346	278723	161065	246	147
700000	366048	219928	326814	193767	246	147
800000	430472	246137	388756	226297	246	147
900000	484793	280708	445701	258298	249	149
1000000	590916	360580	502061	279101	246	147

Tabela 2. Busca Sequencial em Lista Ligada para 100 Números Aleatórios

A Tabela 3 mostra os resultados para o pior cenário em arranjos estáticos. Apesar de seguir o mesmo padrão de comportamento observado nos testes com listas ligadas, neste cenário, destaca-se uma redução significativa no desvio padrão da memória, que diminui para 15 *bytes*.

A quantidade de comparações e o tempo de execução mantêm-se consistentes, seguindo o padrão observado nos testes anteriores.

Arranjo	Média Tempo (µs)	Std. Tempo (µs)	Média Comparações	Std. Comparações	Média Memória (*bytes*)	Std. Memória (*bytes*)
100000	75222	1293	100000	0	492	407
200000	194223	9222	200000	0	215	15
300000	341951	63221	300000	0	215	15
400000	412654	75092	400000	0	215	15
500000	554573	87263	500000	0	215	15
600000	651441	58738	600000	0	215	15
700000	751246	24103	700000	0	215	15
800000	902571	48820	800000	0	215	15
900000	1114220	74737	900000	0	215	15
1000000	830447	46981	1000000	0	215	15

Tabela 3. Busca Sequencial em Arranjos Estáticos para Pior Cenário

A Tabela 4 apresenta os resultados dos testes de busca de 100 números aleatórios em arranjos estáticos. Observa-se que, assim como nos testes em listas ligadas, a média e o desvio padrão da memória não sofrem alterações significativas. No entanto, neste cenário, são ligeiramente mais baixos em comparação.

A quantidade de comparações exibe uma alta variação e desvio padrão, uma vez que o algoritmo percorre cada elemento do vetor até identificar o elemento buscado.

Quanto ao tempo, seu comportamento é semelhante ao teste anterior, porém, razoavelmente mais baixo. Isso ocorre porque, neste cenário, os números são identificados, finalizando a busca antes de percorrer todos os números.

Arranjo	Média Tempo (µs)	Std. Tempo (µs)	Média Comparações	Std. Comparações	Média Memória (*bytes*)	Std. Memória (*bytes*)
100000	36259	23422	46639	29679	246	147
200000	108318	57915	102417	53660	245	147
300000	147893	84703	146589	82036	245	147
400000	207495	117753	208268	114551	247	148
500000	251446	142753	257351	143924	245	147
600000	301827	174897	302355	168425	245	147
700000	526437	533529	335551	212528	245	147
800000	455872	259248	415447	231372	245	147
900000	433782	275646	433408	269678	248	148
1000000	420341	242030	508841	269554	245	147

Tabela 4. Busca Sequencial em Arranjos Estáticos para 100 Números Aleatórios

A Tabela 5 apresenta os resultados dos testes de busca binária em arranjos estáticos para casos de pior cenário. Neste tipo de busca, observamos um comportamento notavelmente diferente. No cenário de pior caso, podemos perceber uma qualidade e velocidade na busca extremamente mais ágil e rápida do que em situações onde utilizamos busca sequencial, devido à complexidade da busca binária que é $O(\log n)$, onde n é o número de elementos no arranjo.

Arranjo	Média Tempo (µs)	Std. Tempo (µs)	Média Comparações	Std. Comparações	Média Memória (*bytes*)	Std. Memória (*bytes*)
100000	49	19	16	0	467	411
200000	40	12	17	0	187	15
300000	93	10	18	0	187	15
400000	38	15	18	0	187	15
500000	38	12	18	0	187	15
600000	39	16	19	0	187	15
700000	43	20	19	0	187	15
800000	37	13	19	0	187	15
900000	42	19	19	0	187	15
1000000	46	23	19	0	187	15

Tabela 5. Busca Binária em Arranjos Estáticos para Pior Cenário

A Tabela 6 apresenta os resultados dos testes de busca binária em arranjos estáticos para 100 números aleatórios. A média e o desvio padrão de memória variam entre 218 e 147 *bytes*, respectivamente.

Comparadas às pesquisas simples, as comparações são significativamente reduzidas devido à eliminação de parte da lista nesse cenário. Isso resulta em um tempo médio de execução e identificação do número também menor, variando entre 31 a 43 µs, o que é extremamente rápido.

Arranjo	Média Tempo (µs)	Std. Tempo (µs)	Média Comparações	Std. Comparações	Média Memória (*bytes*)	Std. Memória (*bytes*)
100000	31	4	17	0	218	147
200000	43	22	18	0	217	147
300000	40	14	18	0	217	147
400000	37	10	19	0	219	148
500000	34	6	19	0	217	147
600000	35	7	19	0	217	147
700000	35	11	19	0	217	147
800000	35	2	19	0	217	147
900000	37	19	20	0	220	148
1000000	35	8	20	0	217	147

Tabela 6. Busca Binária em Arranjos Estáticos para 100 Números Aleatórios

A Tabela 7 apresenta os resultados dos testes de busca em árvores binárias de busca. Utilizamos uma árvore AVL balanceada, e os resultados, assim como nos arranjos de busca binária, demonstram extrema eficiência com baixo consumo de memória e poucas comparações.

Embora o consumo médio de memória tenha sido ligeiramente maior do que nos arranjos, esse comportamento está provavelmente associado aos tratamentos e ajustes realizados para o balanceamento da árvore.

Arranjo	Média Tempo (µs)	Std. Tempo (µs)	Média Comparações	Std. Comparações	Média Memória (*bytes*)	Std. Memória (*bytes*)
100000	41	21	18	0	2917	2127
200000	46	26	17	0	299	173
300000	36	19	20	0	299	173
400000	37	24	20	0	299	173
500000	42	18	20	0	299	173
600000	60	45	18	0	299	173
700000	117	101	20	0	299	173
800000	32	17	19	0	299	173
900000	43	28	22	0	299	173
1000000	44	31	21	0	299	173

Tabela 7. Busca em Árvore Binária de Busca para Pior Cenário

A tabela 8 apresenta os resultados dos testes de busca em árvores binárias de buscas para 100 números aleatórios, utilizamos aqui uma árvore AVL balanceada, os resultados assim como nos arranjos de busca binária são extremamente eficientes com baixo uso de memória e pouquíssimas comparações.

O uso médio de memória foi um pouco maior que nos arranjos, comportamento este provavelmente ligado a questão dos tratamentos e ajustes realizados para balanceamento da árvore.

Arranjo	Média Tempo (µs)	Std. Tempo (µs)	Média Comparações	Std. Comparações	Média Memória (*bytes*)	Std. Memória (*bytes*)
100000	18	3	8	2	255	283
200000	25	5	9	2	218	147
300000	29	9	9	2	218	147
400000	23	4	9	2	220	148
500000	24	7	10	2	218	147
600000	31	9	9	2	218	147
700000	29	7	11	2	218	147
800000	23	4	10	3	218	147
900000	27	6	10	2	219	147
1000000	26	8	10	2	218	147

Tabela 8. Busca em Árvore Binária de Busca para 100 Números Aleatórios

Podemos observar em um gráfico na figura 1 com todos os testes, que os algoritmos de busca sequencial aumentam linearmente o número de comparações conforme o tamanho do vetor, enquanto que os testes com buscas binárias por sua vez são extremamente menores.

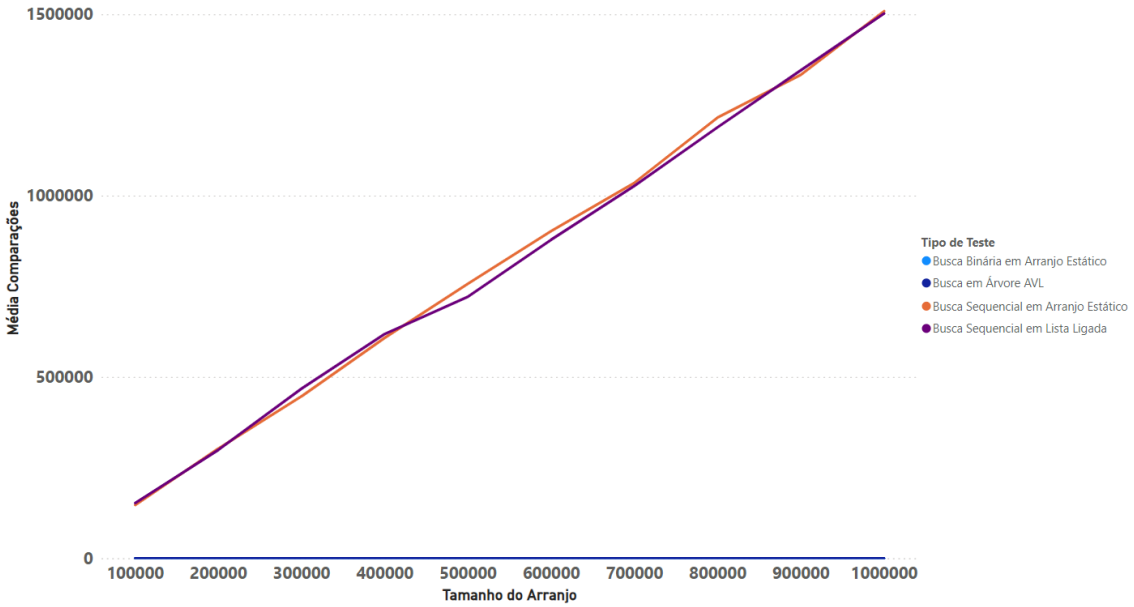

Figura 1. Número de Comparações por Arranjo

Comportamento semelhante ocorre com o tempo dos algoritmos conforme observamos na figura 2 onde podemos observar que o tempo cresce de maneira linear com o tamanho do vetor assim como com a quantidade de comparações.

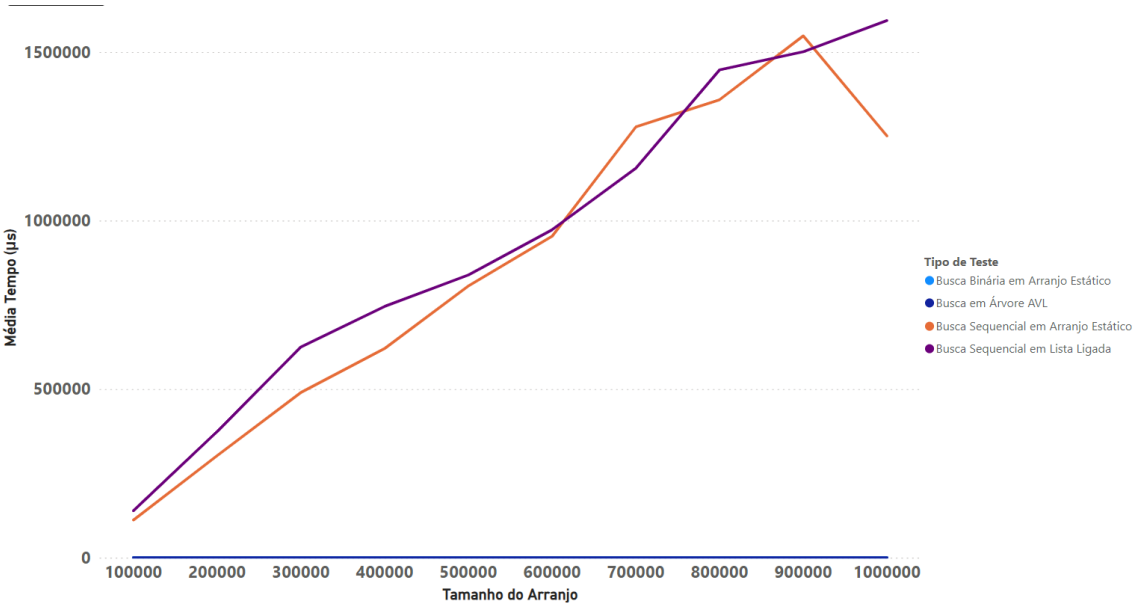

Figura 2. Tempo médio de Comparações por Arranjo

5. Conclusão

Com base nos resultados dos testes realizados para diferentes métodos de busca em arranjos estáticos e estruturas de dados lineares e não-lineares, podemos inferir como os algoritmos operam. No caso da busca sequencial, os testes corroboram a notação de Big

O, demonstrando que a quantidade de comparações aumenta linearmente com o tamanho da lista, o que confirma a complexidade $O(n)$.

Os testes também confirmam que a mesma relação de Big O se aplica a listas de complexidade $O(\log n)$, onde a quantidade de comparações varia em média de acordo com o logaritmo do número total de elementos. Isso evidencia que tais algoritmos são adequados para cenários onde se busca minimizar tanto o tempo quanto a quantidade de comparações necessárias.

A seleção do método ideal para uma aplicação específica depende da implementação. Em conjuntos pequenos ou quando a ordenação prévia é impraticável, a busca sequencial pode ser suficiente. No entanto, para conjuntos de dados grandes, ordenados ou não, a busca binária é, indubitavelmente, a melhor escolha.

Conclui-se, portanto, que a escolha da estrutura de dados influencia diretamente no método de busca. As listas ligadas são destacadas em situações que demandam flexibilidade, enquanto as árvores são mais adequadas para cenários de busca.

É importante ressaltar que as análises e conclusões apresentadas neste relatório são derivadas dos resultados obtidos nos testes conduzidos. Contudo, é fundamental reconhecer que tais conclusões podem não ser generalizadas para todas as situações, uma vez que o desempenho e a eficácia da estrutura de dados podem variar conforme as características específicas do problema ou do ambiente de implementação.

Referências

Bhargava, A. Y. (2016). *Entendendo Algoritmos, um guia ilustrado para programadores e outros curiosos*. Novatec Editora, São Paulo, Brasil.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2012). *Algoritmos - Teoria e Prática*. Elsevier, Rio de Janeiro, 3 edition.

Dasgupta, S., Papadimitriou, C. H., and Vazirani, U. V. (2006). *Algorithms*. McGraw-Hill, New York, NY.