Comparação da Eficiência de Algoritmos de Busca em Estruturas de Dados

Juliane Frabel N. Correia

[1]Universidade Estadual do Oeste Paranaense
Campus de Cascavel
Rua Universitária, 2069 - Bloco B (Prédio velho) - Bairro Universitário
85819-110 - Cascavel - PR

julianefrabel@gmail.com

Abstract. *This article addresses the comparison between static and dynamic arrays, linear and non-linear data structures, in terms of time and memory performance. It proposes a comparison between search algorithms across four types of data structures: ordered and unordered static arrays, linked lists, and binary search trees. The objective is to understand the behavior of each algorithm in different scenarios and quantities of elements, focusing on the structure creation time and search time. The study, conducted in Java, highlights the efficiency of binary search across various scenarios, corroborating findings from the literature.*

Resumo. *Este artigo aborda a comparação entre arranjos estáticos e dinâmicos, estruturas lineares e não lineares, em termos de desempenho de tempo e de memória. Ele propõe uma comparação entre algoritmos de busca em quatro tipos de estruturas: arranjos estáticos ordenados e não ordenados, lista encadeada e árvore binária de busca. O objetivo é compreender o comportamento de cada algoritmo em diferentes cenários e quantidades de elementos, com foco no tempo de criação da estrutura e no tempo de busca. O estudo, realizado em Java, destaca a eficiência da busca binária em diversos cenários, corroborando achados da literatura.*

1. Introdução

A escolha de arranjos estáticos ou dinâmicos quando se trata de otimizar performance pode ser um desafio, é necessário saber previamente informações como frequência de inserção, remoção de dados e frequência de buscas. Com base nessas informações pode-se tomar a decisão de manter os elementos ordenados, o que aumenta consideravelmente o custo computacional do algoritmo ou se esse custo maior é desenecessário e a melhor decisão é manter não ordenado.

Este artigo se propõe a comparar algoritmos de busca em quatro tipos de estrutura de dados, dois arranjos estáticos, um não está ordenado, e o outro está ordenado, as outras duas estruturas serão dinâmicas, uma linear, utilizando os conceitos de lista encadeada, e a última estrutura será uma árvore binária de busca.

Os casos mencionados são estruturas que vemos no dia a dia dos sistemas computacionais. Muitas vezes escolhemos o mais simples, ou até o mais complexo, esperando

que a complexidade sane os possíveis problemas futuros. Porém entender como cada estrutura de dados funciona com cada algoritmo é o melhor caminho para a escolha correta da solução para cada problema.

O objetivo deste trabalho é entender como cada algoritmo se comporta em diferentes cenários com diferentes quantidades de elementos. Com foco nos algoritmos de busca, os experimentos feitos mediram separadamente o tempo para a criação de cada estrutura de dados e o tempo de busca para o alvo selecionado.

O experimento foi conduzido na linguagem de programação Java, o que proporcionou simplicidade na codificação das estruturas, e também garantiu desempenho e confiabilidade na execução. No entanto, é importante observar que para este estudo o comportamento do *garbage collector* foi indesejado, já que atrapalhou a medição do consumo de memória uma vez que não podemos controlar quando ele é acionado.

Como resultado, notou-se grande semelhança com o descrito na literatura sobre cada método, a busca binária se mostrou muito mais eficiente mesmo nos cenários com maior número de elementos e também nos piores cenários onde forçamos um elemento alvo que não constava nas listas.

2. Fundamentação Teórica

Segundo Cormen (2002), um algoritmo é qualquer procedimento computacional bem definido que torna algum valor ou conjunto de valores como entrada e produz algum valor ou conjunto de valores como saída. Um algoritmo também pode ser considerado como uma ferramenta para resolver um problema computacional bem especificado. O enunciado especifica em termos gerais a relação desejada entre entrada e saída e o algoritmo descreve um procedimento computacional específico para se conseguir essa relação entre entrada e saída.

Nem todo problema computacional requer o algoritmo mais eficiente disponível. Algoritmos diferentes criados para resolver o mesmo problema podem ser muito diferentes em termos de eficiência. Para o propósito deste artigo, o termo eficiência está relacionado ao tempo e espaço necessário para realizar a tarefa de busca.

2.1. Arranjos Estáticos

Uma estrutura de dados do tipo lista representa um conjunto de dados organizados em ordem linear. Quando esta estrutura é representada por um arranjo, ou seja, é feita com a utilização de vetores, tem-se o uso de endereços contíguos de memória do computador e a ordem linear é determinada pelos índices do vetor. São estruturas estáticas, o que significa que possuem tamanhos fixos pré-determinados no momento da criação e que não podem ser alterados.

Para este estudo utilizou-se dois arranjos estáticos homogêneos, na busca sequencial e na busca binária, o método de inserção escolhido para a busca sequencial foi o de menor custo computacional, percorreu-se o vetor preenchendo com os elementos desejados, neste a ordenação não foi considerada. No arranjo da busca binária, antes de inserir o conjunto de dados no vetor, foi utilizado o método de ordenação *sort* da classe Java Arrays, este método executa uma ordenação do tipo Dual-Pivot Quicksort, escrito por Vladimir Yaroslavskiy, Jon Bentley e Joshua Bloch. Este algoritmo oferece uma perfor-

mance de O(n log(n)) em todos os conjuntos de dados, e é tipicamente mais rápido que o tradicional Quicksort.

2.2. Estruturas lineares e não-lineares

As estruturas de dados lineares são aquelas em que os elementos são organizados em uma sequencial linear, como listas, pilhas e filas. Já as estruturas de dados não lineares são aquelas em que os elementos são organizados de forma hierárquica, como árvores e grafos. Este trabalho buscou comparar duas estruturas muito semelhantes, onde uma é linear, a lista encadeada não ordenada, e outra é não-linear, a árvore binária de busca. Na Tabela 1. Comparação Detalhada é possível ver as diferenças entre cada estrutura.

Característica	Estruturas Lineares	Estruturas Não Lineares
Organização	Sequencial	Hierárquica
Acesso	Sequencial	Não Sequencial
Operações	Básicas	Complexas
Exemplos	Arrays, listas, filas, pilhas	Árvores, grafos, tabelas hash, conjuntos

Tabela 1. Comparação Detalhada

Acerca da lista encadeada, esta é uma estrutura de dados na qual os objetos estão organizados em ordem linear. Porém, diferente de um arranjo, onde a ordem linear é determinada pelos índices do arranjo, a ordem em uma lista encadeada é determinada por um ponteiro em cada objeto.

A estrurura de dados de lista encadeada pode ser representada em alguns tipos diferentes, cada uma delas pode ser implementada de forma dinâmica. Em uma lista simplesmente encadeada, omite-se o ponteiro anterior em cada elemento, se a lista é ordenada, a ordem linear da lista corresponde à ordem linear de chaves armazenadas, portanto o primeiro elemento da lista é o menor e o maior se encontra no final da lista. Caso a lista seja não ordenada, os elementos podem aparecer em qualquer ordem. Ainda pode-se ter listas duplamente encadeadas e listas circulares.

Para este estudo utilizou-se a estrutura lista simplesmente encadeada e não ordenada, estrutura na qual cada elemento armazena um valor e um ponteiro para o próximo elemento, que permite o encadeamento e mantém a estrutura linear. Por não ser ordenada, foi utilizado o método de inserção de menor custo computacional, onde o novo valor é inserido sempre no início da lista.

Para o contexto de estruturas não-lineares, a escolha foi do tipo árvore, onde os elementos não estão armazenados de forma sequencial e também não estão todos encadeados. Uma árvore binária é um conjunto finito de elementos, onde cada elemento é denominado nó e o primeiro é conhecido como raiz da árvore. Cada nó contém atributos esquerda, direita e p, que apontam para os nós correspondentes ao seu filho à esquerda, ao seu filho à direita e ao seu pai, respectivamente.

Para este estudo utilizou-se um método recursivo de inserção, onde a cada entrada de valor, o algoritmo verifica se o valor inserido é maior ou menor que o valor no nó atual e continua a execução até encontrar um lugar apropriado para inserção do valor.

2.3. Busca Sequencial

A técnica mais simples de busca é a busca sequencial, também chamada de busca linear. Este tipo de busca é aplicável a pequenos tamanhos de estrutura de dados em listas ou listas encadeadas. O alvo da busca é procurado de maneira sequencial. Neste método o processo de pesquisa é iniciado do início ao fim, verificando cada elemento da lista, até que o alvo da busca seja encontrado.

Este modelo de busca foi utilizado no arranjo estático e na lista encadeada, é o algoritmo mais simples para conjunto de elementos não ordenados.

2.4. Busca Binária

Outra forma simples de busca e um método mais eficiente é a busca binária. Este tipo de busca pode ser usado em tamanhos maiores de estrutura de dados lineares e não-lineares. Esta busca é baseada no algoritmo *Dividir e Conquistar*.

Dividir: a lista já ordenada é dividida em duas partes.

Conquistar: primeiro se compara o elemento alvo com o elemento do meio da lista. Se não forem iguais, checa-se se o elemento alvo é maior ou menor que o elemento do meio da lista, se for menor descarta-se a parte da lista maior e repete o algoritmo até encontrar o alvo. O mesmo ocorre se for maior, descarta-se a parte que não interessa mais e repete recursivamente até encontrar o alvo.

3. Materiais e métodos

Esta sessão descreve o ambiente utilizado no decorrer do experimento. Além do escopo utilizado para os testes.

3.1. Ambiente Computacional

3.1.1. Hardware

Para a realização dos experimentos, foi utilizado um computador com as seguintes especificações:

- Processador: 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz 2.80 GHz;
- Memória RAM: 16,0 GB (utilizável: 15,7 GB);
- Armazenamento: SSD NVMe de 240 GB.

3.1.2. Sistema Operacional

O sistema operacional utilizado durante todo o processo experimental foi:

- Windows 11 Home Single Language.

3.1.3. Linguagem de Programação e Ambiente de Desenvolvimento Integrado (IDE)

Para a implementação dos algoritmos e a condução dos experimentos, foi utilizado o seuginte ambiente de desenvolvimento integrado:

- Linguagem de Programação Java (OpenJDK 22);
- IntelliJ IDEA 2023.3.4 (Ultimate Edition).

3.1.4. Bibliotecas e Dependências

Durante a implementação dos algoritos e análise dos resultados, utilizamos algumas bibliotecas e dependencias para exportar e armazenar os dados coletados, incluindo:

- Apache POI - XSSF para documentos Excel;
- JProfiler 19.9.3 - Dados da JVM em tempo real.

3.1.5. Cenários de Testes

Para obtermos resultados padronizados e uma média bem fundamentada da análise, os testes foram estruturados da seguinte forma:

- As amostras de dados foram separadas a cada 100.000 elementos (entre 100.000 e 1.000.000 de elementos), totalizando 10 cenários;
- Os elementos não foram repetidos nos cenários (a cada cenário foi gerado um novo conjunto de elementos);
- Cada cenário manteve a mesma estrutura de dados e ordenação;
- A média de cada busca foi resultado de 100 testes em cada cenário;
- A cada volta dos 100 testes foi utilizado o mesmo alvo para os quatro tipos de busca.

4. Resultados

O experimento foi dividido em cinco etapas.

4.1. Primeira Etapa - Criar os arquivos

A primeira etapa foi criar os 10 cenários necessários, a opção escolhida foi gerar separadamente cada arquivo e armazenar em um arquivo de texto. Assim se garantiu que todas as buscas teriam o mesmo cenário.

No método criado *generateUniqueRandomArray*, foi utilizado o objeto Random, nativo do Java, este objeto pode retornar um fluxo de números pseudoaleatórios. Para garantir a unicidade de cada elemento, foi escolhido a lista Set, também nativa do Java, onde não é permitido valores duplicados. O método citado é capaz de gerar números aleatórios e garantir que eles não se repetem até preencher o número de elementos solicitado.

O resultado foi 10 arquivos com números aleatórios únicos. Variando de 100.000 elementos até 1.000.000 de elementos.

4.2. Segunda Etapa - Preencher as estruturas de dados

O objetivo principal do experimento é comparar como a busca se comporta em diferentes estruturas de dados, nesta etapa preenchemos as estruturas uma única vez para cada cenário, isso garantiu que o custo de criação e inserção não fosse contabilizado no custo da busca.

Nesta etapa também foram criados os arquivos para armazenar os dados dos testes de cada uma das quatro buscas, esses arquivos são planilhas, geradas com o auxílio da biblioteca da Apache Poi. Material importante para análise dos resultados e confiabilidade que os testes ocorreram dentro dos padrões estabelecidos. A Tabela 2. Nomes das estruturas de dados e tipos de busca trás organizado além do nome da variável, o tipo de estrutura de dados, o tipo de busca e o método que rodou o teste de busca.

Nome	Estrutura de Dados	Tipo de Busca	Método
arraySequential	Estática não ordenada	Sequencial	sequentialTest
arrayBinary	Estática ordenada	Binária	binarySearchTest
list	Lista encadeada	Sequencial	sequentialLinkedListTest
tree	Árvore binária	Binária	binarySearchBinaryTreeTest

Tabela 2. Nomes das estruturas de dados e tipos de busca

4.3. Terceira Etapa - Testes de busca

A etapa de testes é a principal para o experimento, nela foi simulado para cada um dos cenários uma busca com o mesmo alvo. O teste foi repetido 100 vezes para cada cenário. Na Figura 1. Laço de repetição de cada cenário é possível entender a estrutura dos testes citadas, onde os parâmetros de cada método de teste são: a estrutura de dados, o tamanho do cenário e o alvo da busca, respectivamente.

```
for (int i = 0; i < 100; i++) {
    System.out.println(STR."Lap: \{i + 1}");

    int target = new Random().nextInt(constant) + 1;
    sequentialTest(arraySequential, constant, target);
    binarySearchTest(arrayBinary, constant, target);
    sequentialLinkedListTest(list, constant, target);
    binarySearchBinaryTreeTest(tree, constant, target);
}
```

Figura 1. Laço de repetição de cada cenário

4.4. Quarta Etapa - Organização dos Resultados

É importante ressaltar que a primeira etapa foi realizada isolada, as segunda e terceira etapas foram realizadas em conjunto. O custo da criação dos arquivos foi desconsiderado.

Utilizamos a média dos 100 testes rodados em cada cenário para formar os dados a seguir.

A primeira comparação que temos acesso é o custo de tempo para criar as estruturas de dados. Foi considerado o tempo de criação das variáveis e a inserção dos dados.

Como pode-se verificar na Figura 2. Comparação Tempo x Quantidade, a estrutura mais custosa do experimento foi a árvore binária, o que é valido já que ela utiliza de função recursiva, é necessário armazenar o estado atual, consumindo mais recursos de tempo e memória. Conforme a quantidade de elementos foi aumentando, o custo da inserção na árvore binária também aumentou significativamente, o que leva a entender que a árvore ficou desbalanceada.

Já as demais estruturas tiveram uma eficiência semelhante entre si, os arranjos estáticos foram preenchidos sequencialmente, a ordenação do arranjo para a consulta binária está neste contexto, como pode-se verificar o custo do vetor ordenado ficou ligeiramente maior.

Por fim, a inserção da lista encadeada foi a menos custosa, já que a opção foi inserir o novo elemento no início da lista, ocorrendo em O(1).

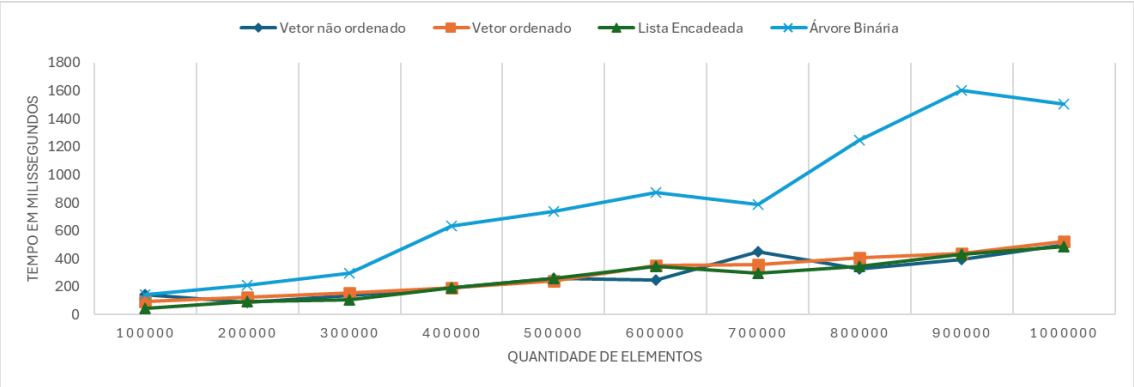

Figura 2. Comparação Tempo x Quantidade

Observou-se também em termos de eficiência, a quantidade de comparações ou trocas, feitas em cada algoritmo de busca. Ambos os algoritmos sequenciais se mostraram custosos, por necessitar percorrer toda a lista, um por um, em busca do alvo desejado.

Já as buscas binárias obtiveram uma média de 20 comparações, tanto no vetor quanto na árvore, monstrando-se econômico na quantidade de comparações, uma vez que a cada comparação metade da lista é descartada. É uma diferença bastante grande em comparação à média das buscas sequenciais que orbitou entre 45 mil e 52 mil comparações ou trocas. Visualmente destacado na Figura 3. Comparação Trocas x Quantidade.

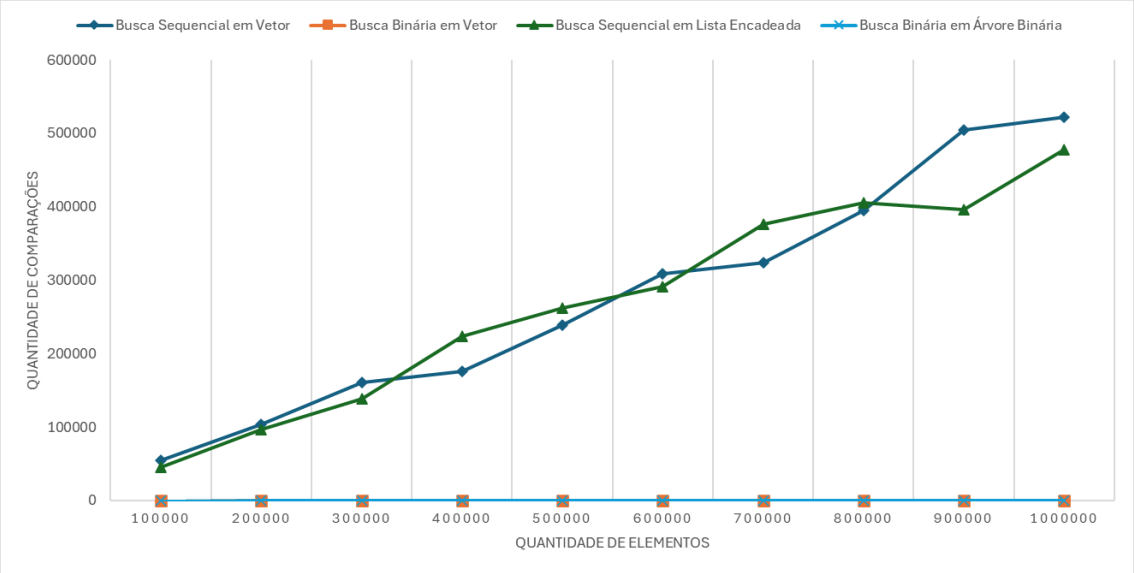

Figura 3. Comparação Trocas x Quantidade

Comparou-se também a média dos casos de testes. Aqui foi necessário diminuir a unidade de tempo, de milissegundos para nanosegundos, pois as buscas binárias se mostraram muito eficientes.

Sabe-se que a alocação contígua em memória torna a busca mais eficiente em termos de tempo, porém se não tem previamente o tamanho necessário a ser alocado corre-se o risco de ter um arranjo pequeno demais ou grande demais, ambos trazendo prejuízo ao custo final do algoritmo. Neste estudo todos os cenários foram fixos, trazendo o melhor desempenho de cada uma das estruturas.

Aqui a estrutura de lista encadeada teve o pior desempenho em tempo, nos casos médios, o custo de percorrer cada nó para encontrar o alvo mostra-se ineficiente.

Ainda sobre o custo de percorrer cada elemento para encontrar o alvo, a busca sequencial em vetor não ordenado, apesar de se mostrar bem mais rápida que a busca em lista encadeada, também se mostrou menos eficiente que ambas as buscas binárias.

Por fim, ambas as buscas binárias, tanto a feita em estrutura estática, quanto na estrutura dinâmica, tiveram médias abaixo dos zero segundos em todos os cenários. Demonstrando que idenpendentemente do tipo de estrutura, a estratégia dividir para conquistar é bastante eficiente e vale a pena o custo de inserção para poder ter a busca otimizada.

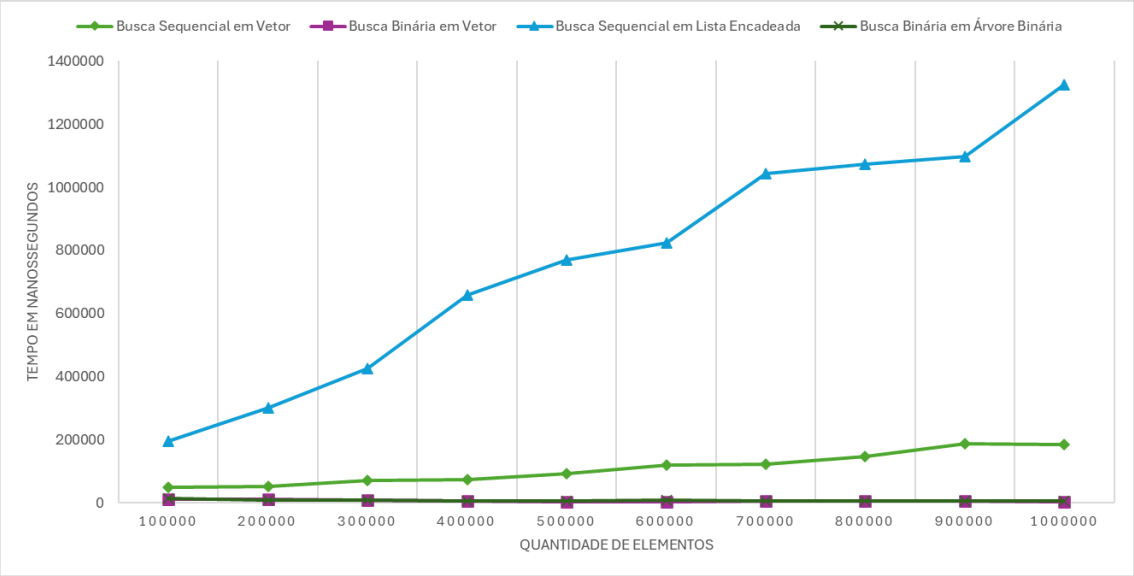

Figura 4. Comparação Tempo x Quantidade

Comparou-se também o custo em alocação de memória, esta comparação foi prejudicada, por causa da linguagem de programação escolhida, como o Java possui o *garbage collector*, o gerenciamento da memória é feita de forma automatizada e independente, ou seja, a desalocação da memória acontece quando a JVM acha necessário. O que excluiu a possibilidade de utilizar a função nativa do Runtime que exibe os dados da memória, pois ao fazer o cálculo do total de memória utilizado, o *garbage colletor* já havia passado retornando sempre um uso zero de alocação.

Optou-se por utilizar o *plugin* para a IDE JProfiler, que monitora o consumo da memória em tempo real. A consideração foi a execução completa dos algoritmos, resultando nos seguintes dados: o maior consumo foi o método para preencher os vetores, dado que foi utilizado duas vezes, tanto para o vetor não ordenado na busca sequencial quanto para o vetor ordenado na busca binária. Seguido dos outros dois métodos de inserção de elementos, o que nos mostra que em termos de eficiência de memória alocada, é mais

custosa a função de inserção do que a função de busca.

Novamente as funções relacionadas ao algoritmo de dividir para conquistar se mostram mais eficientes, consumindo o menor valor de memória nos testes, destacado na Tabela 3. Alocação de memória por método.

Método	Alocação de Memória (GB)
fillArrayFromFile	8.97
fillTreeFromFile	5.44
fillListFromFile	4.78
sequentialLinkedListTest	2.36
sequentialTest	2.35
binarySearchTreeTest	2.33
binarySearchTest	2.28

Tabela 3. Alocação de memória por método

4.5. Quinta Etapa - Piores Casos

Para encontrar os piores casos de cada cenário não é vantajoso utilizar o método randomico utilizado nas fases anteriores. Portanto a melhor opção para os algoritmos utilizados foi utilizar um número comprovadamente fora da amostra de elementos. Ainda que para os algoritmos de busca sequencial e binária nos arranjos estáticos e também na lista encadeada, utilizar um número fora da amostra funcione bem para simular um pior caso, para a árvore binária não basta, pois a árvore pode não estar balanceada, resultando em um falso pior caso.

Para simular o pior caso da árvore binária, foi necessário encontrar a altura dela, utilizou-se o método *getHeight* da classe BinaryTree, e a partir desse dado, escolheu-se um alvo que esteja nesse último nó.

A busca binária se mostra bastante eficiente, mesmo nos piores casos não é percorrido todos os elementos da lista. Mantendo um padrão de O(1).

A busca linear se mostra bem menos eficiente, percorrendo todos os elementos da lista e quanto maior a quantidade de elementos, mais demorada o algoritmo se torna, mantendo um padrão de O(n).

Na Figura 5.Comparação Trocas x Quatidade (pior caso), detalha bem como com o aumento da quantidade de elementos, as comparações das buscas sequenciais acompanham esse aumento, já as buscas binárias mantém uma média de comparações enxutas. A maior quantidade de comparações das buscas binárias foi de 25 comparações, já as buscas sequenciais neste cenário sempre foi a quantidade de elementos da lista.

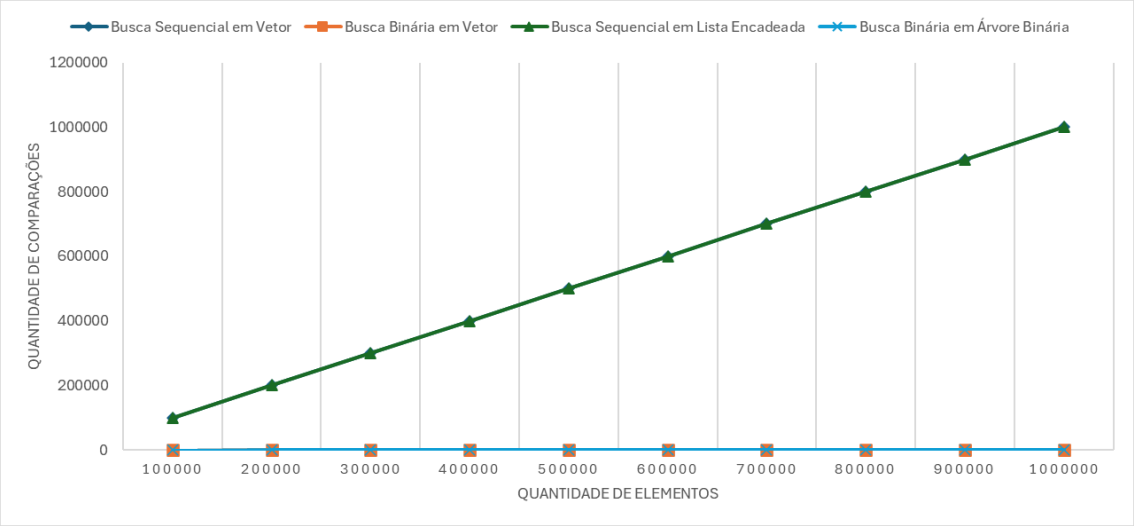

Figura 5. Comparação Trocas x Quantidade (pior caso)

Quanto ao custo de tempo, as buscas binárias continuam tendo o menor custo.

Em relação às buscas sequenciais, mesmo tendo a mesma quantidade de comparações, em termo de custo de tempo, a lista encadeada fica mais custosa. Isso se dá pela própria estrutura de dados, como cada nó possui um ponteiro para o próximo elemento, a alocação não é contígua, os elementos podem estar espalhados na memória tornando o acesso mais custoso.

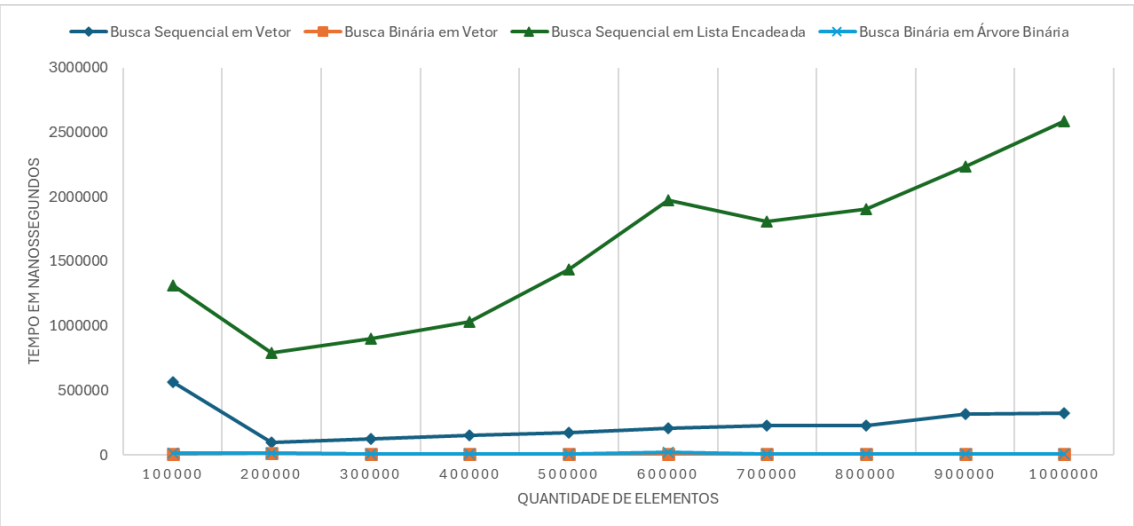

Figura 6. Comparação Tempo x Quantidade (pior caso)

5. Conclusão

Há muitos fatores que afetam a escolha de um método de busca em ciência da computação. Esta escolha é geralmente feita com base na eficiência do algoritmo. Outro fator importante é o tamanho da estrutura de dados. Este estudo mostrou que as informações não devem ser levadas em conta isoladamente, por exemplo se levar em consideração apenas o custo ao criar a estrutura de dados, a escolha será a lista encadeada, que é a menos

custosa independentemente da quantidade de elementos, porém o custo de buscar um elemento nela é o mais elevado dos quatro métodos propostos, o que pode trazer gargalos a depender do objetivo final do algoritmo.

Assim como, levando em consideração apenas o custo de busca, os algoritmos de busca binária se mostram muito mais eficientes, tanto em estruturas lineares como em não-lineares, porém o custo de criação e inserção da estrutura de dados árvore binária foi identificado como maior. Caso o ponto principal do algoritmo seja a busca, o alto custo de inserção não deve ser um problema.

Portanto, é crucial considerar a escalabilidade e a complexidade do problema em questão. Enquanto alguns algoritmos podem se destacar em cenários de pequena escala, sua eficiência pode diminuir significativamente à medida que a quantidade de dados aumenta. Portanto, ao selecionar um método de busca, é essencial avaliar não apenas o desempenho em condições ideais, mas também como ele se comporta sob diferentes cargas de trabalho e volumes de dados. Entender as necessidades do problema garantirá que a escolha do método de busca seja adequada tanto para as necessidades atuais quanto para possíveis demandas futuras, contribuindo assim para o desenvolvimento de sistemas robustos e eficientes em ciência da computação.

Referências

[1] Thomas H. Cormen (2009) *Algoritmos*, Teoria e Prática.