Análise Experimental da Eficiência de Algoritmos de Busca em Vetores e Listas Ligadas Ordenadas e Não Ordenadas Papers and Abstracts

[1]Ciência da Computação – Universidade Estadual do Oeste do Paraná
UNIOESTE
Cascavel, Paraná – Brasil

Abstract. *This paper presents a comparative analysis of the efficiency of different search algorithms applied to arrays (static structures) and linked lists (dynamic structures), in both ordered and unordered contexts. The goal was to evaluate how data structure and ordering affect execution time, number of comparisons, and memory usage. Search algorithms implemented in C include sequential search (standard and optimized) and binary search for arrays, and sequential search for linked lists. The structures were tested with datasets ranging from 100,000 to 1 million unique elements, and performance metrics were collected using custom instrumentation and system commands, automated via shell scripts. Results were stored in a PostgreSQL database and analyzed using Python, with visual outputs including graphs and tables. The findings indicate that arrays performed better in terms of time and memory, particularly when ordered, while linked lists incurred higher costs in creation and access, with scalability limitations.*

Resumo. *Este trabalho apresenta uma análise comparativa da eficiência de diferentes métodos de busca aplicados a vetores (estruturas estáticas) e listas ligadas (estruturas dinâmicas), em contextos ordenados e não ordenados. O objetivo foi avaliar o impacto da estrutura de dados e da ordenação sobre o tempo de execução, número de comparações e uso de memória. Para isso, foram implementados, em linguagem C, algoritmos de busca sequencial (comum e otimizada) e busca binária para vetores, além da busca sequencial para listas ligadas. As estruturas foram testadas com conjuntos variando de 100 mil a 1 milhão de elementos, e os dados de desempenho foram coletados por meio de instrumentação de código e comandos do sistema operacional, organizados via scripts de automação. Os resultados foram armazenados em banco de dados PostgreSQL e analisados com Python, gerando gráficos e tabelas comparativas. Constatou-se que vetores apresentaram melhor desempenho em tempo e memória, especialmente quando ordenados, enquanto listas ligadas mostraram maior custo na criação e acesso, com impacto significativo na escalabilidade.*

1. Introdução

A escolha adequada das estruturas de dados é um fator decisivo para a eficiência de algoritmos em aplicações computacionais. Dentre as operações fundamentais, a busca por elementos em conjuntos de dados é uma das mais recorrentes, e seu desempenho depende diretamente da estrutura subjacente e do contexto de uso. Em particular, estruturas como

vetores (arranjos estáticos) e listas ligadas oferecem diferentes vantagens e limitações a depender dos critérios analisados, como tempo de execução, número de comparações e uso de memória. Segundo [Cormen et al. 2012], a escolha da estrutura de dados correta pode significar a diferença entre um programa que funciona eficientemente e outro que se torna impraticável. Operações como busca, inserção e remoção variam consideravelmente em desempenho conforme a estrutura utilizada.

Diante disso, este trabalho parte da seguinte questão: *em cenários de grande volume de dados, qual estrutura de dados linear — vetores ou listas ligadas — apresenta melhor desempenho em operações de criação e busca, considerando algoritmos distintos e diferentes configurações de ordenação?* A investigação busca responder a essa pergunta por meio da análise comparativa entre métodos de busca sequencial padrão, sequencial otimizada e binária aplicados a vetores, e buscas sequenciais em listas ligadas ordenadas e não ordenadas.

A comparação considera diferentes cenários de entrada com dados inteiros únicos, variando de 100 mil a 1 milhão de elementos, com análise separada dos custos de criação e busca. A motivação para essa análise está na necessidade de fundamentar, por meio de experimentação, quais estruturas e algoritmos oferecem melhor desempenho sob diferentes condições. Em sistemas reais, a decisão sobre a estrutura de dados a ser utilizada pode impactar significativamente a escalabilidade e o tempo de resposta das aplicações, especialmente em ambientes com recursos limitados.

Foram implementadas as estruturas e algoritmos em linguagem C, com medições detalhadas de tempo de execução, número de comparações e uso de memória. Os testes foram conduzidos com controle dos fatores variáveis e reprodutibilidade, utilizando uma mesma base de dados gerada aleatoriamente para todos os algoritmos. Além disso, foram avaliados tanto cenários de pior caso quanto buscas aleatórias.

Este trabalho está organizado da seguinte forma; A seção de Fundamentação Teórica descreve os conceitos de vetores e listas ligadas, suas diferenças quanto à manipulação e eficiência, bem como os algoritmos de busca utilizados. Em seguida, a seção de Materiais e Métodos detalha o ambiente de testes, a organização do código e os parâmetros das execuções. Na seção de Resultados são apresentados e discutidos os dados obtidos. Por fim, a Conclusão retoma os principais achados do estudo e sugere possibilidades para trabalhos futuros.

2. Fundamentação Teórica

A análise da eficiência de algoritmos de busca está diretamente relacionada à estrutura de dados sobre a qual esses algoritmos operam. Neste trabalho, foram adotadas duas representações clássicas de coleções lineares: vetores e listas ligadas. Ambas são estruturas lineares, mas com características distintas em termos de alocação de memória, complexidade de acesso e desempenho em diferentes cenários.

De acordo com [Cormen et al. 2012], um vetor, ou *array*, é uma estrutura de dados que armazena elementos contiguamente em memória, permitindo acesso direto a qualquer posição via índice. Essa característica torna a operação de acesso extremamente eficiente, com complexidade constante $\mathcal{O}(1)$. No entanto, a inserção e a remoção de elementos podem se tornar custosas, especialmente em vetores ordenados, onde é necessário deslocar elementos subsequentes, levando a um custo de até $\mathcal{O}(n)$ em casos desfavoráveis.

Em contrapartida, uma lista ligada (ou *linked list*) é composta por células (ou nós), onde cada nó armazena um valor e um ponteiro para o próximo elemento na sequência. Segundo [Goodrich and Tamassia 2011], listas ligadas são estruturas dinâmicas que não exigem alocação contígua em memória e permitem inserções e remoções em tempo constante $\mathcal{O}(1)$, desde que a posição seja previamente conhecida. No entanto, o acesso sequencial aos elementos impõe um custo linear $\mathcal{O}(n)$ para buscas, pois pode ser necessário percorrer toda a lista.

Neste estudo, foram consideradas duas variações estruturais em vetores e listas: ordenada e não ordenada. No caso dos vetores, a versão ordenada foi construída após o carregamento inicial dos dados, com a aplicação do algoritmo de ordenação *bubble sort*. Esse método consiste em realizar comparações sucessivas entre pares de elementos adjacentes e trocá-los de posição quando estão fora da ordem desejada. A cada iteração, o maior (ou menor) valor da porção não ordenada é movido para a extremidade correta do vetor, simulando o comportamento de "bolhas"subindo à superfície. Apesar de seu apelo didático, o *bubble sort* possui complexidade quadrática no caso médio e no pior caso, $\mathcal{O}(n^2)$, o que o torna ineficiente para grandes volumes de dados [Backes 2023]. Ainda assim, foi adotado neste trabalho por sua simplicidade e facilidade no processo de ordenação. Já os vetores não ordenados foram preenchidos de forma sequencial utilizando iteração simples sobre o conjunto de entrada, alocando cada elemento diretamente em posições sucessivas do *array*, sem qualquer critério de ordenação ou necessidade de reorganização posterior. Esse processo explora a eficiência da indexação direta característica dos arranjos estáticos, resultando em inserções com custo constante $\mathcal{O}(1)$ por operação [Cormen et al. 2012].

Com relação as listas ligadas, a versão não ordenada foi construída por inserções no início da lista — uma prática comum devido à sua eficiência, já que não requer percorrer os elementos previamente inseridos [Cormen et al. 2012]. A lista ordenada, por sua vez, foi construída de forma incremental, com inserção de cada novo elemento na posição correta para manter a ordenação crescente. Esse processo exige a varredura sequencial da lista para encontrar o ponto de inserção adequado, resultando em um custo total de construção proporcional a $\mathcal{O}(n^2)$. Importante destacar que a implementação não utilizou um ponteiro adicional para o último nó (*tail*), apenas para a cabeça (*head*), o que teria permitido inserções diretas no final em tempo constante. Dessa forma, todas as inserções foram realizadas a partir do início da lista, o que acentuou o custo de construção da versão ordenada [Backes 2023].

A eficiência de modelo computacional depende não apenas da estrutura de dados, mas também do próprio método de busca utilizado [Backes 2023]. No projeto, foram implementadas três variações principais para vetores: busca sequencial padrão, busca sequencial otimizada (com sentinela) e busca binária. Para as listas ligadas, foi utilizada a busca sequencial clássica.

A busca sequencial (ou linear) consiste em percorrer a estrutura elemento por elemento, comparando o valor buscado com os valores armazenados. Esta é a abordagem mais simples e direta, com complexidade $\mathcal{O}(n)$, sendo indicada para listas ou vetores não ordenados. Segundo [Cormen et al. 2012], essa técnica é eficaz para pequenas coleções ou quando a ordenação não está garantida. A implementação em C foi feita com um simples laço *for*, incrementando o índice até encontrar o valor desejado ou atingir o fim da

estrutura. A contagem de comparações é uma métrica central para avaliar o desempenho em diferentes cenários — especialmente no pior caso, quando o elemento está ausente e todas as posições são verificadas.

Uma variação da busca sequencial é a busca com sentinela, mencionada por [Cormen et al. 2012], que define que ela consiste em colocar temporariamente o valor buscado na última posição do vetor (como uma "sentinela"), garantindo que o laço de busca termine sem verificar explicitamente o fim do vetor a cada iteração. Isso reduz o número de comparações, especialmente úteis em buscas frequentes. A lógica implementada em C primeiro copia o último valor do vetor, insere a chave como sentinela, e inicia o laço de comparação sem teste de limites. Ao final, verifica se a posição onde o valor foi encontrado está dentro do vetor origina.

A busca binária é adequada exclusivamente para coleções ordenadas. Ela consiste em dividir recursivamente a estrutura ao meio, comparando o elemento central com a chave buscada [Cormen et al. 2012]. Ainda, segundo [Sedgewick and Wayne 2011], esse algoritmo apresenta complexidade $\mathcal{O}(\log n)$ no pior caso, sendo significativamente mais eficiente em vetores grandes e ordenados. O algoritmo foi implementado iterativamente. A cada passo, o meio do intervalo atual é avaliado. Se o valor for igual à chave, o algoritmo termina; caso contrário, reduz o espaço de busca pela metade, ajustando os ponteiros esquerda e direita.

Segundo [Knuth 1998], a busca binária foi mencionada pela primeira vez por John Mauchly em 1946, nas *Moore School Lectures*, sendo considerada uma das primeiras discussões publicadas sobre métodos de programação não numéricos. Posteriormente, em 1960, Derrick Henry Lehmer publicou uma versão do algoritmo que funcionava para listas de qualquer tamanho, superando limitações anteriores que exigiam listas com tamanho de $2^n - 1$ elementos [Lehmer 1960].

Para as listas ligadas, o método utilizado foi a busca sequencial. Como não há acesso direto a elementos via índice (característica intrínseca à estrutura), é necessário percorrer os nós da lista um a um. A operação apresenta complexidade $\mathcal{O}(n)$, com custo proporcional ao número de nós. A implementação percorre a lista com um ponteiro auxiliar, comparando o valor de cada nó com a chave [Manzano 2014].

3. Materiais e Métodos

A avaliação comparativa dos algoritmos de busca foi realizada em um ambiente de desenvolvimento controlado e versionada *Git*. A Infraestrutura computacional foi um computador pessoal com processador *Intel Core* i7 9700k[1] operando em 4.9Ghz, tendo 12mb Cache, tendo 32 GB de memória RAM (4x8GB DDR4 DRAM 3200 MHz) e unidade de armazenamento SSD de 512gb (Leitura: 2300MB/s, Gravação: 900MB/s). O sistema operacional utilizado foi o Ubuntu 22.04 LTS, operando por meio do *Windows Subsystem for Linux* (WSL[2]) executando dentro do Windows 11 (v24H2, c26100.3915). As implementações dos algoritmos foram feitas na linguagem C, escolhida por seu alinhamento com os exemplos clássicos da literatura e por permitir controle detalhado so-

[1]Detalhes do processador disponível em: https://www.intel.com.br/content/www/br/pt/products/sku/186604/intel-core-i79700k-processor-12m-cache-up-to-4-90-ghz/specifications.html

[2]Documentação disponível em: https://learn.microsoft.com/pt-br/windows/wsl/

bre operações de baixo nível, como manipulação de memória, estruturação de dados e medição de desempenho.

Para os testes, foram criados conjuntos de dados em arquivos *TXT* com tamanhos variando de 100 mil a 1 milhão de elementos, em intervalos de 100 mil utilizando um algoritmo em Python em sua versão 3.9, utilizando a biblioteca *Random*[3]. Cada conjunto foi processado em duas representações distintas: vetor (estrutura estática com acesso direto) e lista ligada (estrutura dinâmica com alocação distribuída), cada uma em duas variantes: ordenada e não ordenada. As buscas foram realizadas com 1000 chaves existentes e 10 chaves inexistentes por conjunto, para capturar tanto o comportamento médio quanto o pior caso.

A coleta de métricas foi realizada por meio de funções implementadas em C para contagem de comparações e medição do tempo de execução com *gettimeofday()*, além da leitura direta do uso de memória via */proc/self/status*[4], a coleta foi realizada por meio do campo VmRSS, que representa a memória residente em RAM utilizada pelo processo no momento da medição. Essa leitura foi implementada programaticamente durante a execução dos testes, permitindo registrar o consumo de memória de forma precisa e sem dependência de ferramentas externas. Complementarmente, *scripts* em *shell* (*rodar_cria_vetores.sh* e *rodar_cria_encadeadas.sh*) automatizaram a execução dos testes em lote trabalhando com criação e execussão de todos os testes, utilizando o comando */usr/bin/time -v* para extrair tempo e memória máxima, redirecionando os resultados para arquivos de saída processados posteriormente por *scripts* em Python.

Os dados resultantes foram estruturados e armazenados em um banco de dados PostgreSQL, permitindo sua consulta e análise posterior. As análises foram conduzidas com o auxílio de *scripts* em Python, utilizando as bibliotecas *pandas*[5]*, matplotlib*[6] *e seaborn*[7] para geração de estatísticas descritivas e visualizações gráficas, distribuídos em tabelas comparativas, gráficos de linha, barras e dispersão sobre o comportamento das estruturas e algoritmos em diferentes cenários. Todos os dados da pesquisa, arquivos de testes e documentos utilizados estão disponível em repositório público no *GitHub*[8].

4. Resultados

A análise dos resultados obtidos neste estudo é dividida em duas etapas principais: a criação das estruturas de dados e o desempenho dos algoritmos de busca sobre essas estruturas. Primeiramente, serão apresentados os resultados referentes à construção das quatro variações analisadas: vetores ordenados, vetores não ordenados, listas ligadas ordenadas e listas ligadas não ordenadas. Essa etapa teve como objetivo compreender o impacto do tipo de estrutura e da ordenação sobre o custo de inicialização dos dados.

Na segunda etapa, a atenção se voltou para o comportamento dos algoritmos de busca aplicados às estruturas já construídas. Foram consideradas três técnicas de busca

[3]Documentação disponível em: https://docs.python.org/pt-br/3/library/random.html

[4]É um pseudarquivo do sistema Linux que fornece informações detalhadas sobre o processo em execução, incluindo uso de memória, estado, identificadores e permissões, permitindo monitoramento direto em tempo real sem ferramentas externas.

[5]Documentação disponível em: https://pandas.pydata.org/docs/

[6]Documentação disponível em: https://matplotlib.org/stable/index.html

[7]Documentação disponível em: https://seaborn.pydata.org/whatsnew/

[8]Disponível em: https://github.com/**********/******

para vetores (sequencial simples, sequencial otimizada e binária) e busca sequencial para listas ligadas, tanto ordenadas quanto não ordenadas. Para cada caso, foram coletadas métricas de tempo de execução, número de comparações realizadas e memória utilizada.

4.1. Criação dos Vetores e Listas

A primeira etapa dos experimentos consistiu na análise das métricas de criação das estruturas vetoriais, avaliando separadamente os vetores não ordenados e os vetores ordenados. Os vetores não ordenados foram preenchidos diretamente com os elementos em sequência, o que permite inserções rápidas e sem necessidade de realocação ou ordenação. Como resultado, os tempos de criação para esses vetores permaneceram consistentemente baixos, mesmo em cenários com até 1 milhão de elementos, com valores inferiores a 0,2 segundos em todos os casos. A alocação contígua de memória e o processo de cópia sequencial dos dados explicam esse desempenho.

Em contraste, os vetores ordenados foram construídos utilizando o algoritmo de ordenação *Bubble Sort* logo após a inserção inicial dos elementos. Esse algoritmo possui complexidade quadrática no pior e no caso médio, $\mathcal{O}(n^2)$, o que resultou em tempos de criação significativamente superiores. Por exemplo, para o vetor de 800 mil elementos, o tempo de criação superou os 13 minutos ($\approx 784, s$), chegando a quase 16 minutos ($\approx 993, s$) para 900 mil elementos e ultrapassando 20 minutos ($\approx 1229, s$) no caso de 1 milhão. Ainda que não seja o algoritmo de ordenação mais eficiente, sua utilização foi intencionalmente didática, com o objetivo de evidenciar as implicações de se manter um vetor ordenado com estratégias simples.

Quanto ao consumo de memória, os vetores ordenados e não ordenados apresentaram comportamentos semelhantes, com uma leve tendência de aumento nos vetores ordenados. Esse acréscimo não decorre do uso de estruturas auxiliares pelo algoritmo de ordenação — já que o *Bubble Sort* opera *in-place* —, mas sim da sobrecarga computacional associada à sua execução prolongada [Backes 2023]. Ainda assim, os valores permaneceram abaixo de 10 MB em todos os experimentos, confirmando a natureza compacta dos vetores e sua previsibilidade em termos de alocação de memória.

A segunda etapa dos experimentos concentrou-se na análise das métricas de criação das listas ligadas. As listas não ordenadas apresentaram desempenho extremamente eficiente: mesmo com 1 milhão de elementos, o tempo de construção permaneceu inferior a 0,03 segundos. Esse comportamento estável e de baixa latência reflete diretamente a natureza da operação de inserção adotada, que evita percursos adicionais e realocações.

Por outro lado, as listas ordenadas registraram tempos de criação significativamente superiores, com crescimento exponencial à medida que o volume de dados aumentava. Para 800 mil elementos, o tempo ultrapassou 140 minutos ($\approx 8430, s$), chegando a 180 minutos ($\approx 10834, s$) para 900 mil e atingindo 234 minutos ($\approx 14065, s$) no caso de 1 milhão de registros. Esse crescimento acentuado é coerente com a complexidade quadrática observada em inserções ordenadas sem uso de ponteiro auxiliar para o fim da lista, o que exige múltiplas varreduras completas para posicionar corretamente cada novo elemento.

Quanto ao uso de memória, as listas ordenadas consumiram mais espaço que suas equivalentes não ordenadas em todos os cenários, ultrapassando 66 MB nos mai-

ores casos. Essa diferença reflete tanto o tempo de vida prolongado dos nós quanto o impacto do gerenciamento dinâmico de memória em estruturas altamente fragmentadas. A comparação geral entre as abordagens evidencia que manter uma lista ligada ordenada sem estratégias de otimização pode tornar o tempo de construção proibitivo em grandes volumes de dados, mesmo que os benefícios para buscas ordenadas sejam posteriormente explorados. Na Figura 1 podemos observar os valores de criação das estruturas quanto a tempo (s) e memória (MB).

	M_LNO	M_LO	M_VNO	M_VO	T_LNO	T_LO	T_VNO	T_VO
100	5.15	8.36	3.20	4.54	0.004017	27.299703	0.016604	11.993436
200	8.57	14.75	2.42	3.42	0.005236	140.253383	0.031061	48.408307
300	14.18	23.19	4.67	5.70	0.007700	600.419646	0.047635	109.451685
400	17.27	29.37	4.22	6.00	0.010646	1491.131559	0.063409	194.498081
500	20.92	36.07	4.06	5.92	0.012714	2718.436103	0.079125	304.815260
600	24.00	42.29	5.25	7.54	0.015024	4047.750093	0.093145	446.462121
700	26.32	47.68	4.62	7.52	0.017573	6065.198683	0.109876	599.378837
800	29.41	53.76	5.17	8.25	0.019859	8430.430791	0.127907	784.143006
900	32.76	60.33	4.99	8.76	0.025558	10834.548501	0.134108	993.540382
1000	36.02	66.76	5.66	9.52	0.025482	14065.202890	0.157750	1229.069183

Legenda: T = Tempo (s), M = Memória (MB), VO = Vetor Ordenado, VNO = Vetor Não Ordenado, LO = Lista Ordenada, LNO = Lista Não Ordenada

Figura 1. Tempo (s) e memória (MB) por estrutura e tamanho

Esses resultados reforçam que vetores e listas ligadas, embora resolvam problemas semelhantes, possuem características operacionais e implicações de desempenho bastante distintas. Nas Figuras 2 e 3, mostramos um gráfico de barras e linhas onde comparamos tempo de criação das estrutura de dados, nas Figuras 4 e 5, mostramos um gráfico de barras e linhas onde comparamos a memória de criação.

Figura 2. Tempo (s) por estrutura e tamanho

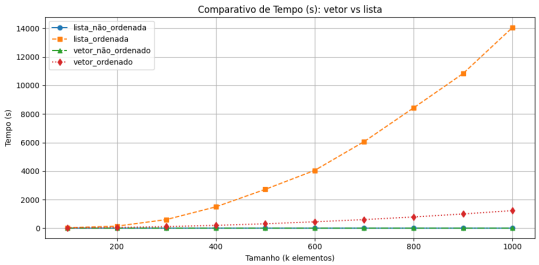

Figura 3. Tempo (s) por estrutura e tamanho

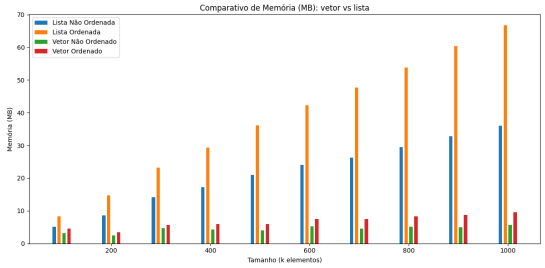

Figura 4. Tamanho (MB) por estrutura e tamanho

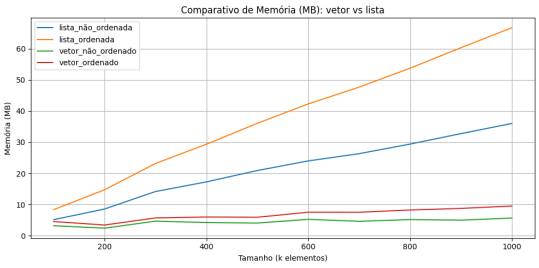

Figura 5. Tamanho (MB) por estrutura e tamanho

Os testes de criação foram executados em dois conjuntos independentes, com

variações controladas. Para fins de padronização e clareza, os gráficos apresentados utilizam os resultados mais estáveis entre as execuções. Por esse motivo, não foram incluídos os desvios padrão da etapa de criação, visto que a variabilidade entre as medições foi desprezível, além disto os tempos de criação e de busca foram medidos de forma separada. Após a criação de cada estrutura, ela foi mantida em memória para as buscas.

4.2. Algoritmos de Busca

A etapa final da análise centrou-se na eficiência dos algoritmos de busca aplicados sobre as estruturas previamente construídas, considerando dois cenários distintos: 1.000 buscas com chaves existentes e 10 buscas com chaves inexistentes, em vetores ordenados e não ordenados. Os algoritmos avaliados foram: busca sequencial simples, busca sequencial com sentinela, busca sequencial otimizada (aplicada a vetores ordenados) e busca binária. As métricas analisadas foram: tempo médio de execução, número médio de comparações e uso de memória.

Na Figura 6, observa-se que a busca binária se destaca amplamente em termos de desempenho. Mesmo com vetores contendo até 1 milhão de elementos, seu tempo médio de execução permaneceu praticamente constante, evidenciando sua complexidade assintótica $\mathcal{O}(\log n)$. Além disso, o desvio padrão associado a esse algoritmo é praticamente nulo, o que demonstra alta estabilidade entre execuções.

Em contraste, os algoritmos de busca sequencial exibiram crescimento linear no tempo, conforme o aumento do volume de dados. Dentre estes, a versão com sentinela apresentou os melhores resultados em tempo médio, mantendo valores inferiores aos da busca simples e da otimizada, especialmente em vetores maiores. Ainda assim, os desvios padrão dessas versões são visivelmente superiores ao da busca binária, indicando uma maior variabilidade nas execuções.

Essa tendência se manteve no cenário com chaves inexistentes (Figura 7), onde a busca binária novamente demonstrou estabilidade. Os algoritmos lineares, por outro lado, apresentaram aumento expressivo no tempo médio, com destaque para a busca sequencial simples, que foi a mais lenta. A versão com sentinela manteve sua vantagem relativa, reforçando seu benefício prático em cenários de busca exaustiva.

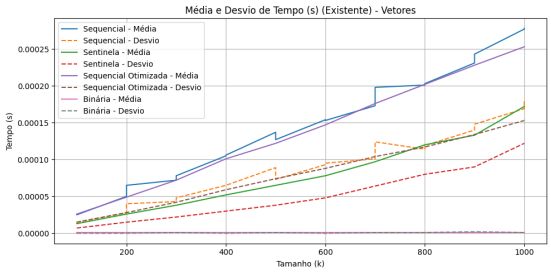

Figura 6. Tempo médio e desvio padrão com chaves existentes (vetores)

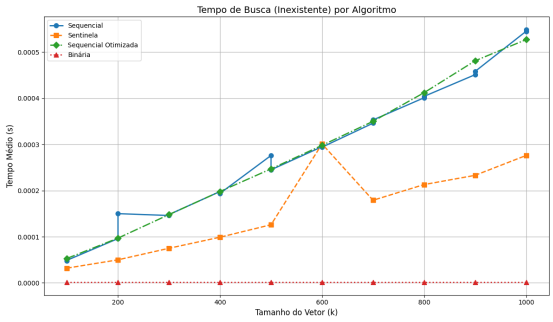

Figura 7. Tempo médio de busca com chaves inexistentes

Os gráficos de comparações médias (Figuras 8 e 9) corroboram esses achados. Nos testes com elementos existentes, os algoritmos lineares apresentaram número

de comparações próximo de $n/2$, com crescimento proporcional ao tamanho da estrutura. A Figura 8 mostra que, além desse crescimento esperado, os desvios padrão nas comparações são relativamente baixos em relação às médias — o que indica alta consistência nos resultados.

A busca binária, mais uma vez, manteve-se praticamente inalterada, com cerca de 18 a 20 comparações em média e desvio praticamente nulo, mesmo nos maiores vetores. Esse comportamento reafirma sua complexidade $\mathcal{O}(\log n)$ e sua eficiência em cenários com ordenação prévia.

Em testes com elementos inexistentes (Figura 9), o número de comparações nos algoritmos lineares atingiu seu máximo teórico, ou seja, n comparações — comportamento característico de algoritmos com complexidade $\mathcal{O}(n)$ no pior caso.

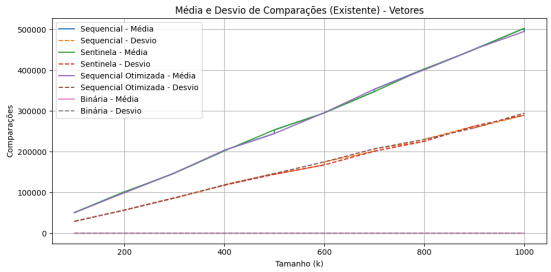

Figura 8. Comparações médias e desvio padrão com chaves existentes

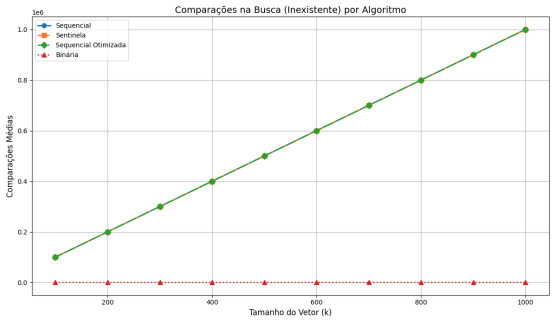

Figura 9. Comparações médias com chaves inexistentes

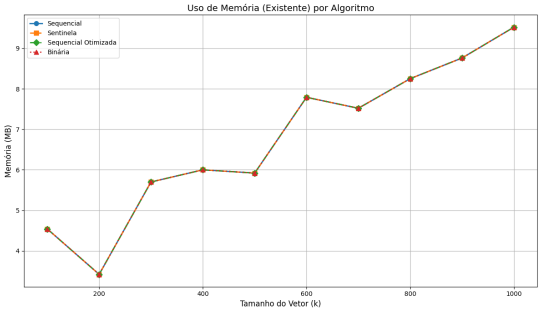

Figura 10. Uso de memória com chaves existentes

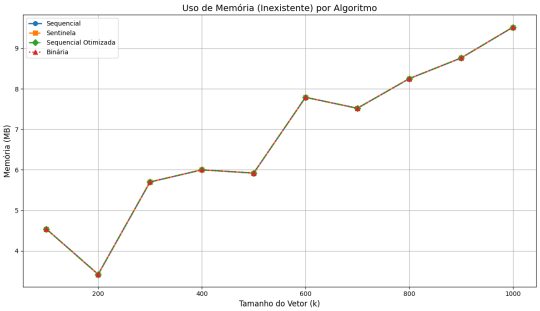

Figura 11. Uso de memória com chaves inexistentes

Com relação ao uso de memória (Figuras 10 e 11), todos os algoritmos demonstraram consumo praticamente idêntico, com variações mínimas entre si. O uso de memória cresceu de forma proporcional ao vetor, girando entre 4,5MB e 9,5MB, refletindo o custo da estrutura base e não do algoritmo de busca em si. Isso ocorre porque todos os métodos operam diretamente sobre os mesmos vetores já alocados, sem demandar estruturas auxiliares relevantes.

A análise conjunta das métricas reforça que algoritmos com menor número de comparações tendem a apresentar melhor desempenho em tempo de execução — desde que o custo de ordenação inicial dos dados seja justificado. Assim, embora a busca binária

se mostre superior em eficiência, sua viabilidade depende da existência de ordenação prévia, o que, como demonstrado na seção anterior, pode ter um custo proibitivo para grandes volumes de dados quando realizado com algoritmos ingênuos.

Portanto, a escolha entre busca linear ou binária deve considerar o volume de dados, a frequência de atualizações, o custo de construção da estrutura e o padrão de acesso. Vetores ordenados favorecem buscas rápidas, mas exigem maior esforço de construção. Já vetores não ordenados são mais rápidos de inicializar, porém menos eficientes em acesso. Dentre os métodos lineares, a busca com sentinela representa uma alternativa eficaz e simples, reduzindo comparações sem custos adicionais.

No segundo caso, referente a análise dos algoritmos de busca aplicados às listas encadeadas considerou duas variações estruturais: listas ordenadas e listas não ordenadas, ambas testadas com 1.000 chaves existentes e 10 inexistentes. Em ambos os casos, foi utilizado o algoritmo de busca sequencial, uma vez que a estrutura da lista ligada não permite acesso direto a posições como ocorre com vetores, inviabilizando a aplicação de busca binária.

Na Figura 12, observa-se que, no cenário com chaves existentes, as listas ordenadas apresentaram desempenho inferior às não ordenadas. Isso ocorre porque, durante a construção ordenada, os elementos são inseridos na posição correta, resultando em distribuição mais uniforme e um percurso mais longo até encontrar a maioria das chaves. Por outro lado, as listas não ordenadas, construídas com inserções no início, concentram elementos mais recentemente inseridos no começo, o que favorece buscas em dados aleatórios.

Além disso, os desvios padrão de tempo, também representados na Figura 12, evidenciam que a busca em listas não ordenadas é não apenas mais rápida em média, como também mais estável. A variação temporal nas buscas ordenadas é significativamente maior, refletindo o custo adicional e a imprevisibilidade associada ao posicionamento dos elementos. Esse padrão se confirma nos maiores volumes de dados testados.

O mesmo padrão foi observado no cenário com chaves inexistentes. Como mostrado na Figura 13, as listas ordenadas demandaram mais tempo de busca, uma vez que a ausência da chave obriga o algoritmo a percorrer todos os nós até o fim. Isso confirma a complexidade linear $\mathcal{O}(n)$ da busca sequencial em listas ligadas, independentemente da ordenação.

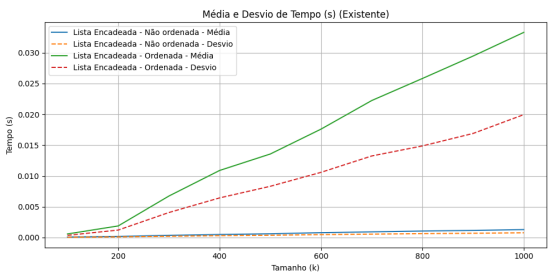

Figura 12. Tempo médio e desvio padrão com chaves existentes (listas)

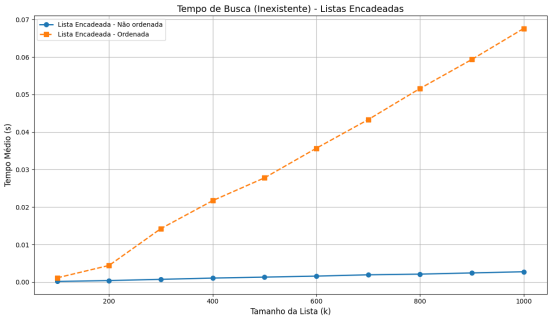

Figura 13. Tempo médio de busca com chaves inexistentes (listas)

As Figuras 14 e 15 ilustram o número médio de comparações por tipo de lista. Para buscas existentes, o número de comparações em listas não ordenadas foi ligeiramente inferior, novamente devido à inserção no início, que favorece a localização antecipada de elementos. Além disso, a Figura 14 mostra que os desvios padrão nas comparações são baixos em relação aos valores médios, indicando estabilidade na execução dos algoritmos mesmo com grandes volumes de dados.

A ordenação da lista tende a dispersar os elementos de forma mais uniforme, o que pode resultar em maior número de comparações em buscas aleatórias. No entanto, o comportamento se manteve estável e previsível, com desvios compatíveis com o padrão observado nas versões não ordenadas.

Para chaves inexistentes (Figura 15), ambas as listas atingiram o número máximo de comparações, equivalente ao tamanho total da estrutura, comportamento esperado para algoritmos com complexidade $\mathcal{O}(n)$ no pior caso. Neste cenário, o desvio padrão é praticamente nulo, uma vez que todas as execuções percorrem toda a lista sem encontrar o elemento.

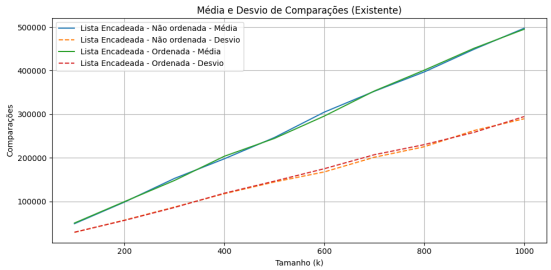

Figura 14. Comparações médias e desvio padrão com chaves existentes (listas)

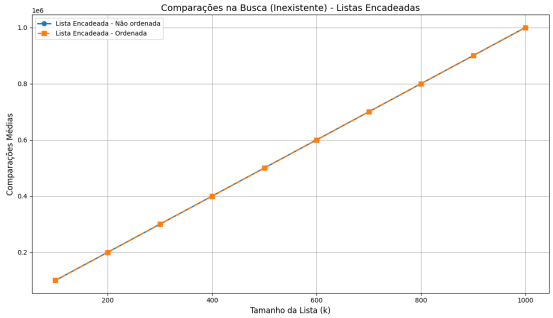

Figura 15. Comparações médias com chaves inexistentes (listas)

Em relação ao uso de memória, as Figuras 16 e 17 demonstram um crescimento proporcional ao tamanho da lista. As listas ordenadas apresentaram ligeiro aumento no consumo de memória em relação às não ordenadas. Isso pode ser atribuído ao maior tempo de execução e ao consequente tempo de vida dos nós e ponteiros na memória RAM, além do maior número de operações realizadas durante a execução do algoritmo.

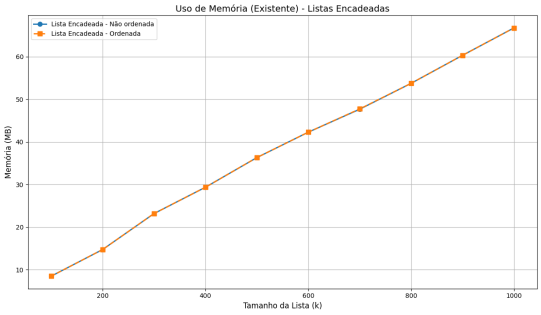

Figura 16. Uso de memória com chaves existentes (listas)

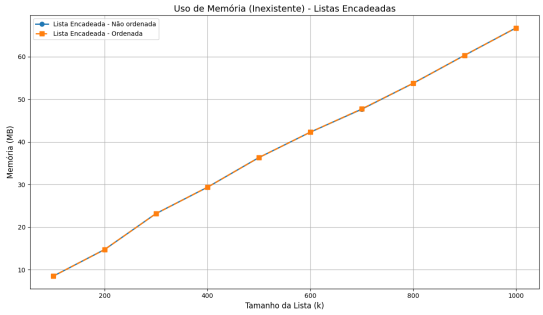

Figura 17. Uso de memória com chaves inexistentes (listas)

Observa-se então que, em listas ligadas, a ordenação impõe um custo adicional não apenas na criação da estrutura (como discutido anteriormente), mas também na busca, principalmente quando a posição dos dados na memória não favorece a localização rápida. Assim, em contextos em que a ordenação não é necessária para outras operações, a lista não ordenada representa uma alternativa mais eficiente tanto em tempo de execução quanto em número de comparações.

Além disso, o experimento evidenciou que os algoritmos lineares, embora simples, não são ideais para cenários com grandes volumes de dados, a menos que a ordenação dos dados seja inviável ou o contexto não permita reestruturações. A busca binária, quando viável, foi amplamente superior, mantendo desempenho estável mesmo diante do crescimento exponencial do tamanho das estruturas.

Esses achados reforçam a importância de decisões fundamentadas no desenho de sistemas e aplicações, em especial aquelas voltadas a grandes volumes de dados, onde eficiência e escalabilidade são requisitos críticos.

5. Conclusão

Este trabalho apresentou uma análise experimental da eficiência de diferentes algoritmos de busca aplicados a vetores e listas ligadas, em versões ordenadas e não ordenadas. Os experimentos envolveram testes com até 1 milhão de elementos, analisando separadamente os custos de criação das estruturas e o desempenho das buscas sob dois cenários: chaves existentes e inexistentes.

A partir dos resultados, foi possível constatar que vetores oferecem desempenho significativamente superior em termos de tempo e uso de memória, sobretudo quando ordenados. A busca binária demonstrou ser a técnica mais eficiente, com número constante de comparações e tempo de execução extremamente reduzido, validando sua recomendação em contextos onde a ordenação é viável.

Por outro lado, listas ligadas apresentaram maior flexibilidade estrutural, mas com custo computacional elevado, especialmente quando ordenadas. A ausência de acesso direto aos elementos limita sua eficiência em operações de busca, tornando-as menos indicadas para aplicações onde a recuperação frequente de dados é um fator crítico.

A análise empírica reforça o que a literatura já consolida: a escolha da estrutura de dados deve considerar não apenas a operação dominante, mas também o perfil de uso da aplicação. Em cenários que exigem busca rápida e acesso frequente, vetores ordenados com busca binária são preferíveis. Já em contextos onde inserções dinâmicas são prioritárias e a ordenação é dispensável, listas não ordenadas podem ser uma alternativa adequada.

Como trabalho futuro, sugere-se a ampliação dos testes para incluir outras estruturas, como árvores balanceadas ou tabelas de dispersão, além da avaliação de algoritmos em arquiteturas paralelas ou distribuídas, visando ampliar a aplicabilidade dos resultados em contextos de alto desempenho e larga escala.

Referências

Backes, A. R. (2023). *Algoritmos e Estruturas de Dados em Linguagem C*. LTC, Rio de Janeiro. E-book. p.29. ISBN 9788521638315. Dis-

ponível em: https://app.minhabiblioteca.com.br/reader/books/9788521638315/. Acesso em: 03 mai. 2025.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2012). *Algoritmos*. Elsevier, Rio de Janeiro. Tradução de: Introduction to Algorithms, 3rd ed. Tradução de Arlete Simille Marques.

Goodrich, M. T. and Tamassia, R. (2011). *Estruturas de dados e algoritmos em C++: uma abordagem orientada a objetos*. Bookman, São Paulo, 2 edition.

Knuth, D. E. (1998). *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley.

Lehmer, D. H. (1960). Teaching combinatorial tricks to a computer. In *Proceedings of Symposia in Applied Mathematics*, volume 10, pages 179–193. American Mathematical Society.

Manzano, J. A. N. G. (2014). *Programação de Computadores com C/C++*. Érica, Rio de Janeiro. E-book. Disponível em: https://app.minhabiblioteca.com.br/reader/books/9788536519487/. Acesso em: 03 mai. 2025.

Sedgewick, R. and Wayne, K. (2011). *Algorithms*. Addison-Wesley, Boston, 4 edition.