Enzo Hubner
Raquel Nascimento
Kaio Marroque

Análise de Complexidade em Estruturas de Listas

Serra/Espírito Santo
2026

Enzo Hubner
Raquel Nascimento
Kaio Marroque

Análise de Complexidade em Estruturas de Listas

Relatório apresentado como requisito parcial para avaliação da disciplina de Técnicas de Programação Avançada.

IFES - Instituto Federal do Espírito Santo
Bacharelado em Sistemas de Informação
Técnicas de programação Avançada

Serra/Espírito Santo
2026

Sumário

Sumário **2**
0.1 Desenvolvimento **4**
0.1.1 Divisão de tarefas 4
0.1.2 Repositório 4
0.1.3 Descrição geral 4
0.2 Análise Matemática de Complexidade **5**
0.2.1 Considerações iniciais 5
0.2.2 Análise do método adicionar 5
0.2.3 Análise do método pesquisar 7
0.2.4 Análise do método remover 8
0.2.5 Análise do método quantidadeNos 10
0.3 Análise Empírica da Biblioteca **10**
0.3.1 Implementação 10
0.3.2 Análise de Complexidade (Resultados) 12
0.3.3 Análise dos Resultados 13
0.4 ArrayList e LinkedList **13**
0.4.1 ArrayList 13
0.4.1.1 Funcionamento Interno 13
0.4.1.2 Adições e remoções de Elementos 14
0.4.1.3 No Fim 14
0.4.1.4 No Início 14
0.4.1.5 Em uma Posição Específica (Meio) 14
0.4.1.6 Busca por um Elemento (Informação do Elemento) 15
0.4.1.7 Busca de um Elemento por Posição Específica (Índice) 15
0.4.2 LinkedList 15
0.4.2.1 Funcionamento Interno 15
0.4.2.2 Adições e remoções de Elementos 15
0.4.2.3 No Fim 15
0.4.2.4 No Início 16
0.4.2.5 Em uma Posição Específica (Meio) 16
0.4.2.6 Busca por um Elemento (Informação do Elemento) 16
0.4.2.7 Busca de um Elemento por Posição Específica (Índice) 16
0.4.3 Tabela Comparativa 17
0.5 Comparação Empírica **17**
0.5.1 Resultados Coletados 17

0.5.2 Discussão dos Resultados 18

0.1 Desenvolvimento

0.1.1 Divisão de tarefas

O desenvolvimento do trabalho foi dividido entre os integrantes do grupo da seguinte forma:

- Enzo Hubner: responsável pelas etapas 1, 2 e 3, incluindo a implementação da biblioteca de lista encadeada, desenvolvimento do programa de teste.
- Kaio Marroque: responsável pelas etapas 4 e 5, incluindo a análise matemática de complexidade e análise empírica dos resultados.
- Raquel Nascimento: responsável pelas etapas 6 e 7, incluindo o estudo teórico de ArrayList e LinkedList e a análise comparativa entre as estruturas.

0.1.2 Repositório

O código-fonte desenvolvido para este trabalho está disponível no seguinte repositório do GitHub:

`https://github.com/enzohubner/Tecnicas-de-Programacao-Avan-ada.git`

O repositório contém tanto a implementação da biblioteca de listas encadeadas quanto o programa de teste utilizado para validar as funcionalidades e realizar as medições de desempenho.

0.1.3 Descrição geral

O projeto foi desenvolvido em linguagem Java, utilizando conceitos de programação orientada a objetos e generics para permitir o armazenamento de diferentes tipos de dados em uma mesma estrutura de lista.

Estrutura do projeto:

O projeto está organizado em pacotes que separam as responsabilidades do sistema, incluindo a implementação da estrutura de dados, a interface da coleção e as classes de domínio utilizadas para testes. Essa organização facilita a manutenção e compreensão do código.

Implementação da lista:

A estrutura principal implementada foi a classe `ListaEncadeada`, que utiliza uma abordagem de lista simplesmente encadeada. Cada elemento da lista é representado por um nó que armazena o dado e uma referência para o próximo elemento.

A classe implementa a interface `IColecao`, garantindo a padronização dos métodos principais, como adicionar, remover, pesquisar e obter a quantidade de elementos.

Execução do sistema:

Foi desenvolvido um programa de teste que permite ao usuário interagir com a lista por meio de operações como inserção, remoção, busca e listagem dos elementos.

0.2 Análise Matemática de Complexidade

0.2.1 Considerações iniciais

A estrutura desenvolvida consiste em uma lista encadeada simples, na qual cada elemento é representado por um nó contendo um valor e uma referência para o próximo nó da lista. O acesso aos elementos dessa estrutura ocorre de forma sequencial, iniciando sempre pelo primeiro nó (referenciado pela variável prim). Diferente de estruturas baseadas em vetor, não há acesso direto a posições específicas da lista. Dessa forma, diversas operações dependem do percurso completo ou parcial da estrutura. Para a análise de complexidade, considera-se:

- n como o número de elementos armazenados na lista;
- a análise do pior caso, conforme solicitado;
- que operações básicas (atribuições, comparações, acessos) possuem custo constante $O(1)$.

0.2.2 Análise do método adicionar

Código analisado

```java
15      @Override
16      public void adicionar(T novoValor) {
17
18          No<T> novo = new No<>(novoValor);
19
20          if (prim == null) {
21              prim = novo;
22          } else {
23
24              No<T> aux = prim;
25
26              while (aux.prox != null) {
27                  aux = aux.prox;
28              }
29
30              aux.prox = novo;
31          }
32
33          quantidade++;
34      }
35
```

Figura 1 – Código do método adicionar numerado

Análise detalhada:

`No<T> novo = new No<>(novoValor);`
Essa operação ocorre em tempo constante, pois envolve apenas alocação de memória e atribuição → $O(1)$.

`if (prim == null)`
Essa verificação também ocorre em tempo constante → $O(1)$.

`prim = novo;`
Caso a lista esteja vazia, o novo elemento é inserido diretamente como primeiro nó → $O(1)$.

`No<T> aux = prim;`
Inicialização para localizar o último nó → $O(1)$.

`while (aux.prox != null)`
Esse laço percorre a lista nó a nó até encontrar o último elemento. Se a lista possui n elementos, o laço executa n - 1 vezes → $O(n)$.

`aux = aux.prox;`
Operação constante dentro do laço → $O(1)$ por iteração.

`aux.prox = novo;`
Inserção do novo elemento após o laço → $O(1)$.

`quantidade++;`
Incremento do contador → $O(1)$.

Conclusão: No melhor caso, quando a lista está vazia, a inserção ocorre diretamente no início: → custo $O(1)$. No entanto, no pior caso, o algoritmo precisa percorrer toda a lista até o último elemento. Ordem de complexidade no pior caso: $O(n)$

Quando ocorre o pior caso? O pior caso ocorre quando a lista já possui n elementos, e o novo elemento é inserido no final da lista. Como a implementação não mantém referência direta ao último nó, é necessário percorrer todos os elementos da lista até encontrar o último, o que torna o tempo de execução proporcional a n.

0.2.3 Análise do método pesquisar

Código analisado:

```
36      @Override
37      public T pesquisar(T valor) {
38
39          No<T> aux = prim;
40
41          while (aux != null) {
42
43              if (aux.valor.equals(valor)) {
44                  return aux.valor;
45              }
46
47              aux = aux.prox;
48          }
49
50          return null;
51      }
```

Figura 2 – Código do método pesquisar numerado

Análise detalhada:

`No<T> aux = prim;`
O método inicia apontando para o primeiro nó da lista → $O(1)$.

`while (aux != null)`
Esse laço percorre a lista desde o primeiro nó até o último. No pior caso, o laço executa n vezes $\rightarrow O(n)$.

`aux.valor.equals(valor)`
Comparação: ocorre a cada iteração $\rightarrow O(1)$.

`aux = aux.prox`
Avanço para o próximo nó $\rightarrow O(1)$.

Conclusão: Melhor caso: elemento está no primeiro nó $\rightarrow O(1)$ Pior caso: percorre toda a lista $\rightarrow O(n)$ Ordem de complexidade no pior caso: $O(n)$

Quando ocorre o pior caso? O pior caso ocorre em duas situações principais: Quando o elemento procurado está localizado no último nó da lista, exigindo a verificação de todos os elementos anteriores; Ou quando o elemento não existe na lista, obrigando o algoritmo a percorrer todos os nós até o final para concluir a busca.

0.2.4 Análise do método remover

Código analisado:

```java
@Override
public boolean remover(T valor) {

    if (prim == null) {
        return false;
    }

    if (prim.valor.equals(valor)) {
        prim = prim.prox;
        quantidade--;
        return true;
    }

    No<T> aux = prim;

    while (aux.prox != null) {

        if (aux.prox.valor.equals(valor)) {
            aux.prox = aux.prox.prox;
            quantidade--;
            return true;
        }

        aux = aux.prox;
    }

    return false;
}
```

Figura 3 – Código do método remover numerado

Análise detalhada:

`if (prim == null)`
O método começa verificando se a lista está vazia → $O(1)$.

`if (prim.valor.equals(valor))`
Depois, verifica se o elemento a ser removido está no primeiro nó → $O(1)$.

`prim = prim.prox`
Se for o caso, a remoção é imediata → $O(1)$.

`No<T> aux = prim;`
Caso contrário, o algoritmo inicia a variável auxiliar → $O(1)$.

`while (aux.prox != null)`
Percorre a lista buscando o elemento a ser removido. O laço pode executar até n - 1 vezes

$\rightarrow O(n)$.

`Ajuste de ponteiros e comparações`
Dentro do laço, as operações de comparação e ajuste de ponteiros são constantes $\rightarrow O(1)$.

Conclusão: Melhor caso: remoção do primeiro elemento $\rightarrow O(1)$ Pior caso: percorre toda a lista $\rightarrow O(n)$ Ordem de complexidade no pior caso: $O(n)$

Quando ocorre o pior caso? O pior caso ocorre quando: O elemento a ser removido está localizado no final da lista, exigindo a verificação de todos os elementos anteriores; Ou o elemento não existe na lista, sendo necessário percorrer toda a estrutura para confirmar sua ausência.

0.2.5 Análise do método quantidadeNos

Código analisado:

```
82      @Override
83      public int quantidadeNos() {
84          return quantidade;
85      }
86
```

Figura 4 – Código do método quantidadeNos numerado

Análise detalhada:

`return quantidade;`
O método apenas retorna o valor de uma variável que armazena a quantidade de elementos da lista. Não há laços $\rightarrow O(1)$.

Conclusão: Ordem de complexidade no pior caso: $O(1)$

Quando ocorre o pior caso? Neste método, o pior caso ocorre em qualquer situação, pois o tempo de execução é constante e independe do número de elementos da lista. Isso é possível porque a variável quantidade é atualizada durante as operações de inserção e remoção.

0.3 Análise Empírica da Biblioteca

0.3.1 Implementação

Para a realização da análise empírica, foi desenvolvido um programa em Java responsável por ler arquivos contendo registros de objetos do tipo Aluno, inseri-los em uma lista

encadeada e medir o tempo de execução das principais operações da estrutura. Os arquivos de entrada foram gerados por meio de um programa próprio, que cria registros no formato:

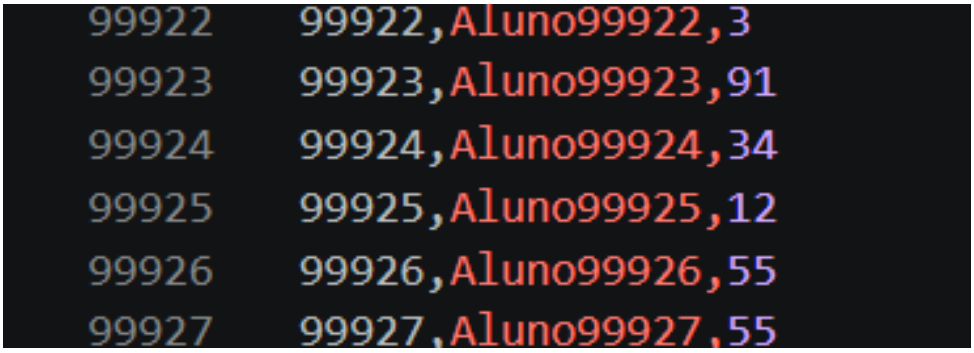


Figura 5 – Exemplo do formato do arquivo de entrada

Cada arquivo contém uma quantidade específica de elementos, sendo utilizados três tamanhos distintos para os experimentos: 100.000, 200.000 e 400.000 registros. A geração automatizada dos arquivos garante padronização dos dados e controle sobre o tamanho das entradas. Para medir o tempo de execução das operações, foi utilizado o método `System.currentTimeMillis()`, que retorna o tempo atual em milissegundos. O tempo de cada operação foi obtido calculando a diferença entre o instante antes e após sua execução.

O programa implementado realiza as seguintes operações:

- Leitura do arquivo e inserção de todos os elementos na lista;
- Busca pelo último elemento inserido;
- Busca por um elemento inexistente;
- Remoção do último elemento da lista;
- Obtenção da quantidade de elementos da lista.

A seguir, apresenta-se um trecho do código responsável pela leitura do arquivo e inserção dos elementos:

```java
try {
    BufferedReader br = new BufferedReader(new FileReader(arquivo));

    String linha;
    Aluno ultimo = null;

    inicio = System.currentTimeMillis();

    while ((linha = br.readLine()) != null) {
        String[] partes = linha.split(regex: ",");

        int matricula = Integer.parseInt(partes[0]);
        String nome = partes[1];
        int nota = Integer.parseInt(partes[2]);

        Aluno aluno = new Aluno(matricula, nome, nota);
        lista.adicionar(aluno);
        ultimo = aluno;
    }
```

Figura 6 – Trecho de código para leitura e inserção

Também foi implementada a medição de tempo para cada operação, conforme ilustrado abaixo:

```java
    fim = System.currentTimeMillis();
    System.out.println("Tempo inserção: " + (fim - inicio) + " ms");

    br.close();

    inicio = System.currentTimeMillis();
    lista.pesquisar(ultimo);
    fim = System.currentTimeMillis();
    System.out.println("Busca último: " + (fim - inicio) + " ms");

    Aluno inexistente = new Aluno(matricula: 0, nome: "NAO_EXISTE", nota: 0);

    inicio = System.currentTimeMillis();
    lista.pesquisar(inexistente);
    fim = System.currentTimeMillis();
    System.out.println("Busca inexistente: " + (fim - inicio) + " ms");

    inicio = System.currentTimeMillis();
    lista.remover(ultimo);
    fim = System.currentTimeMillis();
    System.out.println("Remoção último: " + (fim - inicio) + " ms");
```

Figura 7 – Exemplo da lógica de medição de tempo

Durante a leitura do arquivo, o último elemento inserido foi armazenado em uma variável auxiliar, permitindo a realização da busca e remoção desse elemento posteriormente.

0.3.2 Análise de Complexidade (Resultados)

Os testes foram executados utilizando arquivos com diferentes quantidades de elementos, sendo realizadas múltiplas execuções para maior confiabilidade dos resultados.

Tabela 1 – Tabela de Resultados Empíricos

Elementos	Inserção (ms)	Busca Últo (ms)	Busca Inex. (ms)	Remoção (ms)	Qtd (ms)
100.000	52215 / 50421	2 / 4	5 / 3	4 / 6	0 / 0
200.000	211539 / 213783	5 / 5	5 / 4	5 / 7	0 / 0
400.000	1066957	14	11	12	0

Fonte: Elaboração própria.

0.3.3 Análise dos Resultados

A operação de inserção apresentou crescimento significativo no tempo de execução à medida que o número de elementos aumentou. Observa-se que ao dobrar o tamanho da entrada de 100.000 para 200.000 elementos, o tempo mais que quadruplicou, e ao aumentar para 400.000 elementos, o crescimento foi ainda mais acentuado. Esse comportamento indica que a inserção não possui complexidade linear no contexto do experimento, mas sim quadrática. Isso ocorre porque, na implementação da lista encadeada, cada inserção exige a travessia completa da lista até o último elemento. Dessa forma, o custo de cada inserção é proporcional ao tamanho atual da lista, resultando em um custo total aproximado de $O(n^2)$ para inserir n elementos. As operações de busca, tanto para o último elemento quanto para um elemento inexistente, apresentaram crescimento proporcional ao tamanho da lista. Isso ocorre porque, em ambos os casos, o algoritmo percorre todos os elementos até encontrar o valor desejado ou até o final da lista. Assim, os resultados experimentais confirmam a complexidade linear $O(n)$ para essas operações. A operação de remoção do último elemento também apresentou comportamento linear. Isso se deve ao fato de que, para remover o último elemento, é necessário percorrer toda a lista até o penúltimo nó, o que implica custo proporcional ao número de elementos. Já a operação de obtenção da quantidade de elementos apresentou tempo de execução igual a 0 ms em todos os testes realizados. Esse resultado ocorre devido à limitação da função de medição utilizada (`System.currentTimeMillis()`), que possui precisão em milissegundos. Como essa operação consiste apenas no retorno de uma variável previamente armazenada, seu tempo de execução é extremamente pequeno, sendo inferior a 1 ms. Dessa forma, conclui-se que sua complexidade é constante, ou seja, $O(1)$.

0.4 ArrayList e LinkedList

0.4.1 ArrayList

0.4.1.1 Funcionamento Interno

O ArrayList funciona internamente utilizando um array dinâmico para armazenar seus elementos em locais de memória contíguos (sequenciais). Ele possui uma capacidade inicial

e, à medida que novos elementos foram adicionados além dessa capacidade, o array cresce automaticamente nos bastidores. Essa natureza de array significa que, embora forneça uma estrutura previsível, a manipulação estrutural (como adicionar ou remover no meio do array) exige o deslocamento na memória de vários elementos.

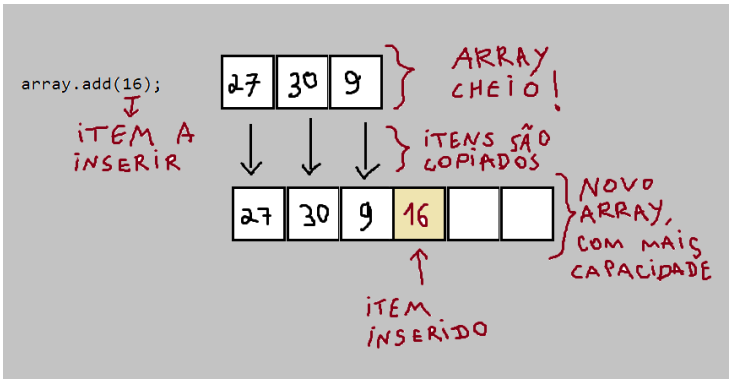


Figura 8 – Funcionamento Interno Array List

0.4.1.2 Adições e remoções de Elementos

0.4.1.3 No Fim

A adição no final ocorre em tempo constante amortizado, O(1). Como o array possui espaço pré-alocado, basta inserir no último índice vazio, tornando a operação rápida na maioria dos casos. A remoção do último elemento também é O(1), pois não exige deslocamento dos demais — apenas a exclusão do índice final e redução do tamanho. Ambas as operações ocupam tempo linear, O(n), apenas em inserções pontuais quando o array enche e precisa ser redimensionado e copiado para um novo espaço na memória.

0.4.1.4 No Início

Complexidade linear, O(n). É lenta para inserir ou remover um item na primeira posição, porque será preciso deslocar todos os elementos existentes para a direita (no caso da inserção) ou para a esquerda (na remoção). Se houver muitos elementos, essa operação vai demorar.

0.4.1.5 Em uma Posição Específica (Meio)

Complexidade linear, O(n). Tanto inserir quanto remover no meio exige que todos os elementos subsequentes a partir daquele índice sejam deslocados — para a direita na inserção, e para a esquerda na remoção — a fim de manter a estrutura contígua do array.

0.4.1.6 Busca por um Elemento (Informação do Elemento)

Complexidade linear, O(n). O método indexOf ou contains tem que varrer os elementos a partir do início, um por um, fazendo comparações de valores (via equals), até encontrar o alvo procurado.

0.4.1.7 Busca de um Elemento por Posição Específica (Índice)

ArrayList: Complexidade constante, O(1). Como os elementos já estão indexados, armazenados um ao lado do outro na memória. Para localizar o décimo elemento de um array, por exemplo, basta usar o método get(), passando como argumento o número do índice (no caso do acesso ao décimo item, seria o índice 9) e rapidamente conseguimos obtê-lo.

0.4.2 LinkedList

0.4.2.1 Funcionamento Interno

Por outro lado, o LinkedList utiliza internamente uma estrutura de lista duplamente encadeada (doubly linked list). Neste modelo, os elementos não são armazenados em locais contíguos de memória. Em vez disso, cada elemento é encapsulado em um "nó"(node), e cada nó armazena três informações: o próprio dado e duas referências (ponteiros) — uma apontando para o próximo elemento e outra para o elemento anterior na lista. há também um atributo para armazenar o endereço de memória do primeiro item da lista, chamado de head, e outro que aponta para o endereço do último elemento, ou tail.

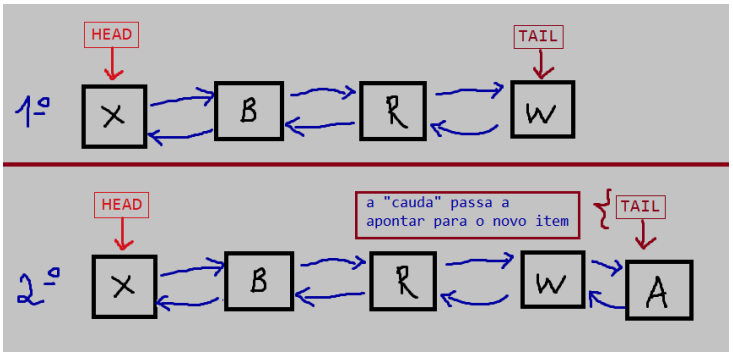


Figura 9 – Funcionamento Interno Linked List

0.4.2.2 Adições e remoções de Elementos

0.4.2.3 No Fim

Tanto a adição quanto a remoção do último elemento ocorrem em tempo constante, O(1). A estrutura mantém um ponteiro direto para o nó final, de modo que adicionar

apenas cria um novo nó e atualiza esse ponteiro, enquanto remover simplesmente reatribui o ponteiro final para o nó anterior.

0.4.2.4 No Início

Complexidade constante, O(1). Esse deslocamento todo não acontece na LinkedList: a operação de adicionar ou remover elementos na primeira posição só mexe nas referências da head e dos elementos anterior e posterior ao novo item inserido.

0.4.2.5 Em uma Posição Específica (Meio)

Complexidade linear, O(n). Embora a manipulação dos ponteiros em si seja feita de forma instantânea, a lista precisa ser percorrida partindo do início ou do fim até alcançar o índice exato. Só após essa travessia — que é a parte cara da operação — os vizinhos do nó têm seus ponteiros ajustados para concluir a inserção ou remoção.

0.4.2.6 Busca por um Elemento (Informação do Elemento)

Complexidade linear, O(n). Igualmente à busca no ArrayList, ele precisa percorrer nó por nó comparando os valores na memória.

0.4.2.7 Busca de um Elemento por Posição Específica (Índice)

Complexidade linear, O(n). Como a memória não é sequencial, ele não consegue saltar diretamente para um índice. Ao invés disso, o programa iniciará a navegação a partir do começo ou do fim da lista (o que estiver mais próximo ao índice buscado) percorrendo cada nó até alcançar a posição solicitada.

Operation	Array (Time Complexity)	Linked List (Time Complexity)
Access Element	O(1)	O(n)
Insert at Beginning	O(n)	O(1)
Insert at End	O(1) (if space is available)	O(n) (for singly linked list)
Delete Element	O(n)	O(1) (if pointer is available)

Figura 10 – Busca por um elemento

0.4.3 Tabela Comparativa

Tabela 2 – Comparação de Complexidade entre ArrayList e LinkedList

Operação	ArrayList	LinkedList
Inserção no fim	O(1)*	O(1)
Inserção no início	O(n)	O(1)
Inserção no meio	O(n)	O(n)
Remoção no fim	O(1)	O(1)
Remoção no início	O(n)	O(1)
Remoção no meio	O(n)	O(n)
Busca por valor	O(n)	O(n)
Acesso por índice	O(1)	O(n)

Fonte: Elaboração própria.

*O(1) amortizado.

0.5 Comparação Empírica

0.5.1 Resultados Coletados

As tabelas a seguir consolidam os tempos de execução (em milissegundos) para as operações de inserção total dos dados e para operações individuais no final da estrutura.

Tabela 3 – Comparação de Desempenho: Inserção Total (ms)

Tamanho (n)	Lista Encadeada (Própria)	ArrayList	LinkedList
100.000	46.348 ms	116 ms	249 ms
200.000	266.515 ms	165 ms	328 ms
400.000	1.200.176 ms	271 ms	541 ms

Fonte: Elaboração própria.

Tabela 4 – Comparação de Desempenho: Busca e Remoção (ms)

Estrutura	Tamanho	Busca (Último)	Busca (Inexistente)	Remoção (Último)
Lista Encadeada	100.000	4 ms	3 ms	3 ms
	200.000	8 ms	6 ms	12 ms
	400.000	24 ms	14 ms	24 ms
ArrayList	100.000	6 ms	5 ms	7 ms
	200.000	7 ms	5 ms	5 ms
	400.000	9 ms	4 ms	9 ms
LinkedList	100.000	8 ms	4 ms	7 ms
	200.000	10 ms	5 ms	7 ms
	400.000	18 ms	11 ms	19 ms

Fonte: Elaboração própria.

0.5.2 Discussão dos Resultados

A análise dos dados revela contrastes significativos entre a implementação própria e as estruturas nativas do Java, especialmente no que tange à escalabilidade.

O Gargalo da Inserção: A `ListaEncadeada` desenvolvida apresentou um crescimento de tempo de inserção que sugere uma complexidade de $O(n^2)$ para o processo total de carga. Enquanto o `ArrayList` e a `LinkedList` nativas levaram menos de um segundo para processar 400.000 elementos, a nossa implementação levou aproximadamente 20 minutos (1.200.176 ms). Isso indica que, para cada inserção, o algoritmo provavelmente percorre a lista inteira até o fim para adicionar o novo nó, caracterizando uma operação $O(n)$ por elemento inserido.

Em contrapartida, o `ArrayList` demonstra a eficiência do tempo constante $O(1)$ amortizado, e a `LinkedList` do Java utiliza um ponteiro `tail` (cauda), o que permite inserções no fim em $O(1)$, justificando a rapidez extrema frente à nossa implementação manual.

Busca e Remoção: Nas operações de busca pelo último elemento e busca inexistente, todas as estruturas apresentaram tempos baixos, mas com distribuições importantes:

- A `ListaEncadeada` e a `LinkedList` mostraram tempos que crescem linearmente conforme n aumenta, o que valida a teoria de que a busca em listas encadeadas exige a travessia de n nós ($O(n)$).
- O `ArrayList` manteve tempos de busca e remoção no fim quase constantes e muito baixos, confirmando a vantagem do acesso direto via índice e a ausência de necessidade de deslocamento de memória quando a alteração ocorre na última posição.

Conclusão da Análise Empírica: Os testes comprovam que a escolha da estrutura de dados impacta drasticamente o tempo de resposta do sistema. Para grandes volumes de dados, como o cenário de 400.000 registros, o uso de uma lista simplesmente encadeada

sem otimização de ponteiro de cauda torna-se inviável, enquanto o `ArrayList` se provou a opção mais eficiente para inserções e operações ao final da lista devido ao seu gerenciamento otimizado de memória contígua.