# Algoritmos e Estruturas de Dados II

Capítulo da Aula 3: Revisão de Estruturas Lineares e Coleções Java

Prof. Aléssio Miranda Júnior
alessio@cefetmg.br
CEFET-MG - Campus Timóteo

Fevereiro de 2026

## 0.1 1. O Arsenal do Maratonista: Java Collections

para AED II e competições, você não vai re-implementar uma Lista Encadeada do zero a cada problema. Você usará o **Java Collections Framework**. Saber qual ferramenta usar pode reduzir seu tempo de execução de $O(N)$ para $O(1)$ ou $O(\log N)$.

### 0.1.1 1.1. Interfaces × Classes Concretas

A grande sacada do Java Collections é separar o **conceito** da **implementação**.

Em AED I, você viu conceitos abstratos. Em Java, essas abstrações puras são representadas por **Interfaces**:

- `List<E>`: sequência *indexada* (lista linear). Permite elementos duplicados.
- `Set<E>`: coleção sem elementos repetidos (conjunto matemático).
- `Queue<E>` e `Deque<E>`: coleções com disciplina de inserção e remoção (Fila FIFO e Pilha LIFO).
- `Map<K, V>`: coleção de pares chave-valor (dicionário ou tabela de símbolos). *Atenção: não herda da interface generalista* `Collection`.

O detalhe é que você não pode instanciar uma interface diretamente (nada de `new List()`). Para isso, existem as **Classes Concretas**, baseadas nas arquiteturas de AED I:

- **Arranjos / Vetores dinâmicos** → `ArrayList<T>` e `ArrayDeque<T>`.
- **Listas ligadas (ponteiros)** → `LinkedList<T>`.
- **Tabelas Hash (espalhamento)** → `HashSet<E>` e `HashMap<K,V>`.
- **Árvores de Busca (balanceadas)** → `TreeSet<E>` e `TreeMap<K,V>`.
- **Heaps (filas de prioridade)** → `PriorityQueue<E>`.

### 0.1.2 1.2. Métodos Essenciais

A hierarquia garante que métodos básicos funcionem em (quase) todas as coleções:

- **Operações transversais (Interface `Collection`):** `add(e)`, `remove(o)`, `contains(o)`, `size()`, `isEmpty()`, `clear()`.
- **Específicos de `List`:** baseado em índices como `get(index)`, `set(index, e)`, `add(index, e)`.
- **Específicos de `Map`:** chave-valor como `put(k, v)`, `get(k)`, `containsKey(k)`.

```
classDiagram
    direction BT
    class Collection { <<interface>> }
    class List { <<interface>> }
    class Queue { <<interface>> }
    class Deque { <<interface>> }
    class Set { <<interface>> }
    class Map { <<interface>> }

    List --|> Collection
    Queue --|> Collection
    Set --|> Collection
    Deque --|> Queue

    class ArrayList
    class LinkedList
    class ArrayDeque
    class PriorityQueue
    class HashSet
    class HashMap
    class TreeMap

    ArrayList ..|> List
    LinkedList ..|> List
    LinkedList ..|> Deque
    ArrayDeque ..|> Deque
    PriorityQueue ..|> Queue
    HashSet ..|> Set
    HashMap ..|> Map
    TreeMap ..|> Map
```

## 0.2 2. Listas: A Batalha Contínua vs Encadeada

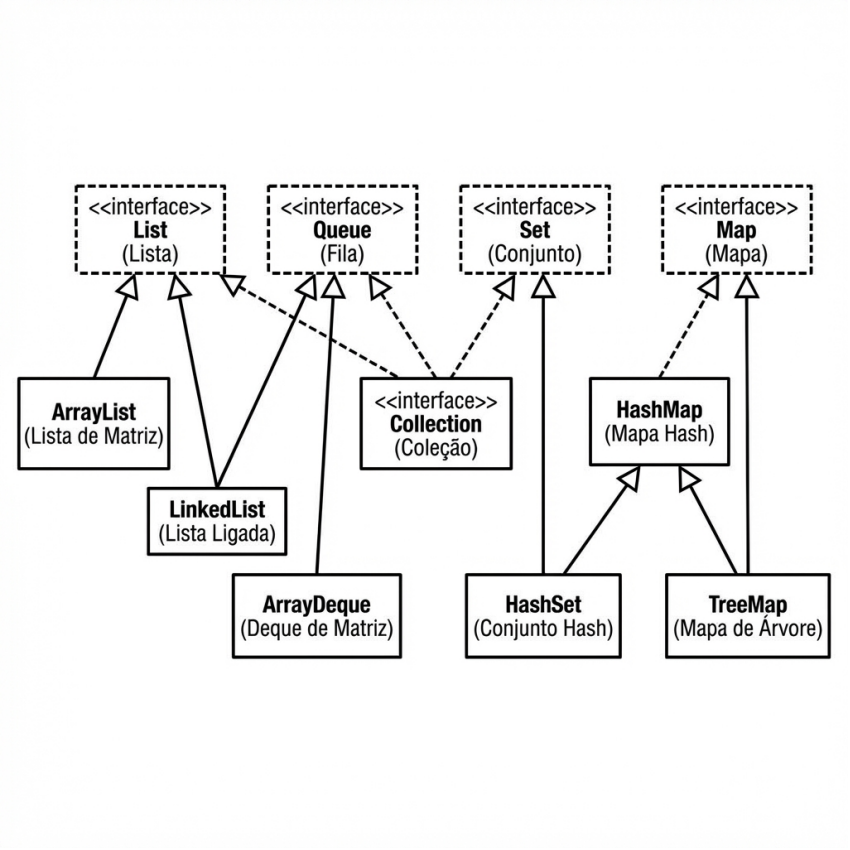


Figure 1: Hierarquia das interfaces e classes do Java Collections (List, Queue, Set, Map).

### 0.2.1 O Vetor Dinâmico: 'ArrayList' (Seu melhor amigo)

É um wrapper sobre um array padrão `tipo[]`. Quando enche, ele cria um novo array com o **dobro** do tamanho e copia tudo ($O(N)$), mas isso acontece tão raramente que a complexidade amortizada é $O(1)$.

- **Vantagem:** Acesso aleatório (`get(i)`) em $O(1)$.
- **Desvantagem:** Remover ou inserir no meio é $O(N)$ (tem que empurrar todo mundo).
- **Segredo:** Muito amigável ao Cache da CPU (dados contíguos na RAM).

### 0.2.2 A Lista Encadeada: 'LinkedList' (O incompreendido)

Cada elemento é um objeto `Node` espalhado na memória, ligado por ponteiros.

- **Vantagem:** Inserir/Remover no início ou fim é $O(1)$.
- **Desvantagem:** Acesso aleatório (`get(i)`) é $O(N)$ – péssimo!
- **Segredo:** Só use se você precisa remover muito no meio da lista **usando um Iterator**. Se for para usar índice, esqueça.

| Operação | ArrayList | LinkedList | | :— | :—: | :—: | | `get(i)` | $O(1)$ | $O(N)$ | | `add(e)` (final) | $O(1)$ | $O(1)$ | | `add(0, e)` (início) | $O(N)$ | $O(1)$ | | `remove(i)` | $O(N)$ | $O(N)$ (mas $O(1)$ com Iterator) |


Figure 2: ArrayList: blocos contíguos na memória. LinkedList: nós espalhados com ponteiros.

## 0.3 3. Pilhas e Filas: As Ferramentas de Trabalho

O segredo em maratonas de programação não é escrever código rápido mecanicamente, mas sim **reconhecer padrões**. Em AED I você construiu a ferramenta; agora você **aplica a ferramenta** aos problemas.

### 0.3.1 Fila (Queue) - FIFO (First-In First-Out)

O padrão é claro: **"Preciso percorrer ou processar na ordem de chegada / O mais antigo primeiro"**. Imagine uma fila de banco. Quem chega primeiro, é atendido primeiro.

- **Aplicação Clássica:** Simulações de filas de atendimento, jogos de baralho (descarte e pesca pelo topo) ou percursos em Grafos via Busca em Largura (BFS).
- **Implementação recomendada:** Use `Queue<T>` instanciando como `LinkedList` ou `ArrayDeque`.

```
Queue<String> fila = new LinkedList<>();
fila.add("Cliente A"); // Enfileira
fila.add("Cliente B");
String prox = fila.poll(); // Desenfileira ("Cliente A")
```

### 0.3.2 Pilha (Stack) - LIFO (Last-In First-Out)

O padrão clássico aqui é: **"Preciso desfazer a última operação / Validar pares perfeitamente aninhados"**. Imagine uma pilha de pratos: você só tira ou coloca o do topo.

- **Aplicação Clássica:** O problema do balanço de parênteses e chaves matemáticas '{[()]}', navegação de histórico (Ctrl+Z), recursão ou buscas em profundidade em Grafos (DFS).
- **Implementação recomendada:** EVITE a classe nativa `Stack` da linguagem (ela é antiga e fortemente sincronizada, logo, muito lenta). **Use `Deque<T>`** instanciando `ArrayDeque`.

```
Deque<Integer> pilha = new ArrayDeque<>();
pilha.push(10); // Empilha
pilha.push(20);
int topo = pilha.pop(); // Desempilha (20)
```

**Por que `ArrayDeque`?** É mais rápida que `Stack` e consome menos memória que `LinkedList`. É híbrida (pode ser Pilha ou Fila).


Figure 3: Pilha (LIFO): último a entrar, primeiro a sair. Fila (FIFO): primeiro a entrar, primeiro a sair.

## 0.4 4. Mapas e Conjuntos (Visão Rápida)

Muitos problemas exigem eficiência extrema de busca. Listas lineares (como `ArrayList`) forçariam uma verificação sequencial $O(N)$, o que é a receita certa para falhar com **Time Limit Exceeded (TLE)** nos juízes online.

### 0.4.1 Dicionários: A Interface `Map`

Reconhecimento de padrão: **"Preciso contar quantas vezes cada elemento aparece / Associar chave → valor"**.

- **HashMap<K,V>:** Usa Tabela Hash. Inserção e busca em tempo quase $O(1)$ médio. A ordem dos elementos inseridos parecerá totalmente aleatória.
- **TreeMap<K,V>:** Usa Árvore Rubro-Negra (Red-Black Tree balanceada). Inserção e busca custam $O(\log N)$, levemente mais lentas, mas os dados sempre ficam rigorosamente **ordenados** e prontos para percurso sequencial lógico.
- **Exemplo Prático:** Contar frequências de certas palavras em um texto enorme ou simular um carrinho de compras veloz associando ID de Produto e seu Preço.

### 0.4.2 Conjuntos: A Interface `Set`

Reconhecimento de padrão: **"Preciso manter uma coleção de elementos distintos (sem repetições)"**.

- **HashSet<E> e TreeSet<E>:** Funcionam com o mesmo mecanismo interno abstrato dos mapas, mas aqui elas se preocupam em ser apenas um saco de objetos únicos ($O(1)$ para hash, $O(\log N)$ para árvore).
- **Exemplo Prático:** Dado um vetor com centenas de milhares de IDs de usuários (vários deles duplicados e sujos), responder rapidamente *quantos IDs únicos* de fato acessaram o sistema ontem.

## 0.5 5. Dicas de Ouro para Java

### 0.5.1 Wrapper Classes e Autoboxing

Java não permite `ArrayList<int>`. Temos que usar `ArrayList<Integer>`. Isso tem um custo de memória absurdo.

- `int`: 4 bytes.
- `Integer`: 24 bytes (Cabeçalho de objeto + referência + valor).

**Dica:** Se o limite de memória for apertado (ex: 256MB) e $N = 10^7$, use arrays primitivos `int[]` em vez de `ArrayList`.

### 0.5.2 Fast I/O (Leitura Rápida)

O `Scanner` é lento porque faz muito parsing. Para problemas com $N > 10^5$, use `BufferedReader`.

**Template para Maratonas:**

```
import java.io.*;
import java.util.StringTokenizer;

public class Main {
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System
        PrintWriter out = new PrintWriter(System.out);

        StringTokenizer st = new StringTokenizer("");

        // Exemplo: Ler N
        String linha = br.readLine();
        if (linha == null) return;
        int N = Integer.parseInt(linha);

        // Ler N inteiros na mesma linha
        st = new StringTokenizer(br.readLine());
        for(int i=0; i<N; i++) {
            int x = Integer.parseInt(st.nextToken());
            out.println(x * 2); // Bufferiza a saída
        }

        out.flush(); // Importante! Despeja o buffer no final.
    }
}
```

## 0.6 6. Problemas Clássicos para Treinar

Utilize problemas do site **Maratona** para fixar não apenas as propriedades da Estrutura de Dados, mas qual classe concreta do *Java Collections* resolve melhor o gargalo temporal das restrições impostas.

- **Pilhas (Stack / Deque): Maratona 1068** – Balanço de Parênteses
  *Objetivo*: Analisar uma string contendo uma expressão matemática e verificar se todos os parênteses abrem e fecham corretamente aninhados.
- **Filas (Queue): Maratona 1110** – Jogando Cartas Fora
  *Objetivo*: Dado um monte de $N$ cartas numeradas de 1 a $N$ no topo de uma estrutura, simular o descarte sucessivo do topo, enviando sempre a carta adjacente à nova base do monte.
- **Mapas (HashMap / TreeMap): Maratona 1281** – Ida à Feira
  *Objetivo*: Uma lista de produtos é apresentada com o respectivo preço. Em seguida um pedido é feito com os nomes dos produtos. Você deve associar o preço ao seu nome perfeitamente num mapeamento em memória veloz ($O(1)$) para acelerar a resposta de simulações com milhões de compras.

## 0.7 7. Exercícios

### 7.1. Exercícios Conceituais

1 Explique em que situações práticas um `ArrayList` é claramente superior a um `LinkedList`, e vice-versa, levando em conta complexidade de tempo e comportamento de cache.

2 Discuta as diferenças de uso entre `Queue`, `Deque` e `Stack` em aplicações de sistemas (escalonamento, algoritmos de busca, undo/redo em editores).

3 Compare `HashMap` e `TreeMap` em termos de complexidade, ordenação e consumo de memória. Dê exemplos de problemas onde cada um é mais indicado.

### 7.2. Exercícios Analíticos

1 Preencha uma tabela comparando `ArrayList`, `LinkedList`, `ArrayDeque`, `HashSet` e `TreeSet` para as operações:

- Inserção no fim/início.
- Remoção no meio.
- Busca por valor.
- Iteração sequencial.

Use complexidades assintóticas ($O(1)$, $O(\log N)$, $O(N)$) e faça comentários sobre o impacto prático dessas diferenças.

2 Analise o impacto de autoboxing (`int` vs `Integer`) na memória ocupada quando se armazenam $10^7$ elementos em um `ArrayList`. Estime a memória usada em cada caso e discuta se isso é viável em ambientes com 256MB de limite.

### 7.3. Exercícios de Programação

1 Implemente uma pequena biblioteca em Java que ofereça uma interface unificada para:

- Fila com duas implementações: `LinkedList` e `ArrayDeque`.
- Pilha com duas implementações: `ArrayDeque` e `ArrayList`.

Faça um programa de teste que compare o tempo de execução das operações básicas para grandes volumes de dados.

2 Escolha um problema de simulação (por exemplo, fila de banco, processamento de requisições HTTP, sistema de impressão) e modele-o usando `Queue` e/ou `Deque`. Meça o impacto de trocar as implementações internas (por exemplo, `LinkedList` vs `ArrayDeque`).

3 Implemente uma solução para um problema de juiz online que requeira mapeamento de chaves para valores (por exemplo, contagem de frequências de palavras, dicionário de produtos). Faça duas versões: uma com `HashMap` e outra com `TreeMap`, e compare o tempo de execução em cenários com dados ordenados e desordenados.