AED2 - Algoritmos e Estr. de Dados II

Aula 3: Revisão de Estruturas Lineares e Coleções Java

Prof. Aléssio Miranda Júnior
alessio@cefetmg.br

CEFET-MG - Campus Timoteo
Dep. Engenharia de Computação

Fevereiro de 2026

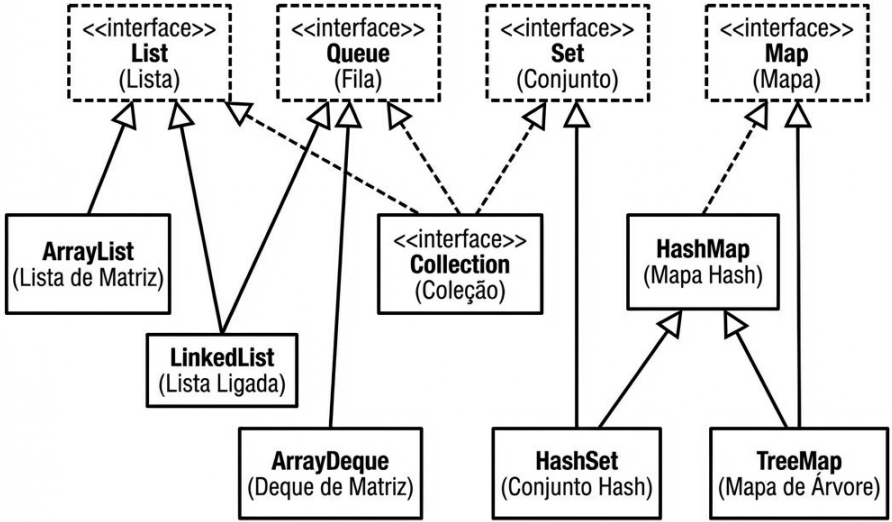
Sumário

1. O Arsenal do Maratonista: Java Collections (parte 1)

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 $O(\log N)$.

1. O Arsenal do Maratonista: Java Collections (parte 2) — Hierarquia



1.1. De AED I para Interfaces Java

Em AED I, você viu conceitos abstratos de estruturação de dados. 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). *Nota: não herda de `Collection`.*

Atenção: Você não pode instanciar uma interface diretamente (ex: `new List()`). Você precisa de uma **Classe Concreta**.

1.2. De AED I para Classes Concretas

Para cada Interface, o Java fornece **Classes Concretas** (implementações com diferentes arquiteturas por trás), baseadas nas estruturas 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>`.

O segredo não é programar a estrutura do zero, mas **escolher a classe certa** para não tomar *Time Limit Exceeded (TLE)*.

1.3. Métodos Básicos Transversais (Interface `Collection`)

Como `List`, `Set` e `Queue` herdam da super-interface `Collection`, elas **compartilham** métodos fundamentais. Se você sabe usar para um, sabe para todos:

- `boolean add(E e)`: Insere um elemento (nas listas vai pro fim, nos conjuntos é ignorado se já existir).
- `boolean remove(Object o)`: Remove uma ocorrência do objeto, se ele estiver lá.
- `boolean contains(Object o)`: Busca elemento (custo de $O(N)$ em listas a $O(1)$ em hashes).
- `int size()`: Retorna o número de elementos guardados (sempre $O(1)$).
- `boolean isEmpty()`: Verifica se a coleção está vazia.
- `void clear()`: Remove totalmente os elementos.

1.4. Métodos Específicos: Listas e Mapas

Interfaces que adicionam comportamentos específicos têm métodos próprios:

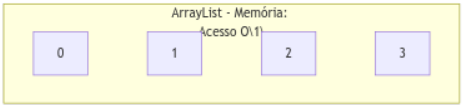
Em `List` (operações baseadas em índice):

- `E get(int index)`: Acessa o elemento na posição `index`.
- `void add(int index, E e)`: Insere na posição `index` (empurrando os demais).
- `E set(int index, E e)`: Substitui o elemento na posição `index`.

Em `Map` (operações baseadas em chave-valor):

- `V put(K key, V value)`: Insere ou atualiza o valor associado à chave.
- `V get(Object key)`: Retorna o valor associado à chave.
- `boolean containsKey(Object key)`: Verifica se a chave existe ($O(1)$ em `HashMap` ou $O(\log N)$ em `TreeMap`).

2. Listas: ArrayList vs LinkedList — Memória



2. Listas: ArrayList vs LinkedList (parte 1)

O Vetor Dinâmico: `ArrayList` (Seu melhor amigo)

É um wrapper de um array padrão `tipo[]`. Quando enche, cria um novo array com o **dobro** do tamanho e copia ($O(N)$), mas como é raramente usamos 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).

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.

Resumo: complexidade das operações

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)`	$O(N)$	$O(N)$

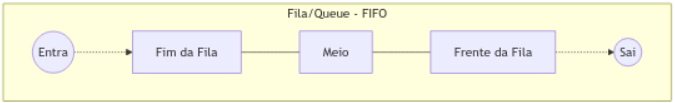
2.2. Complexidade: lembrando de AED I

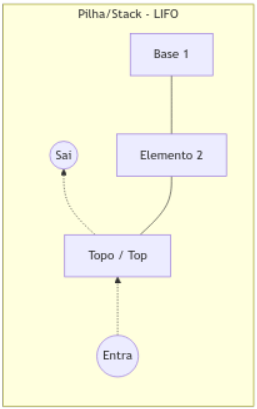
A análise que você fez em AED I continua valendo aqui:

- **ArrayList** $\approx$ vetor dinâmico:
  - Acesso por índice: $O(1)$.
  - Inserir/remover no meio: $O(N)$ (desloca elementos).
- **LinkedList** $\approx$ lista duplamente encadeada:
  - Inserir/remover em posições já conhecidas (início/fim, via iterador): $O(1)$.
  - Buscar a posição i (`get(i)`): $O(N)$, precisa percorrer.
- **HashSet / HashMap** $\approx$ tabela hash:
  - Busca/inserção/remoção: $O(1)$ médio, $O(N)$ pior caso.
  - Ordem dos elementos: não garantida.
- **TreeSet / TreeMap** $\approx$ árvore de busca balanceada:
  - Busca/inserção/remoção: $O(\log N)$.
  - Mantêm os elementos **ordenados** pela chave.

Conclusão de AED I que continua valendo: a **estrutura certa** muda a complexidade do algoritmo, e isso decide se ele passa ou toma TLE.

3. Pilhas e Filas — Comportamento (LIFO x FIFO)





Fila (Queue) - First-In First-Out

Imagine uma fila de banco. Quem chega primeiro, é atendido primeiro.

- **Aplicação Crítica:** BFS (Busca em Largura) em Grafos.
- **Implementação:** Use `LinkedList` ou `ArrayDeque`.

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

Pilha (Stack) - Last-In First-Out

Imagine uma pilha de pratos. Você só tira o do topo.

- **Aplicação Crítica:** DFS (Busca em Profundidade), recursão, validar parênteses, desfazer (Ctrl+Z).
- **Implementação:** EVITE a classe `Stack` (antiga e sincronizada/lenta). **Use** `ArrayDeque`.

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

Por que `ArrayDeque`**?** Mais rápida que `Stack`, menos memória que `LinkedList`. Híbrida (Pilha ou Fila).

5.1. Do enunciado à estrutura: Filas e Pilhas

O segredo em maratonas de programação não é codificar rápido, mas **reconhecer padrões**. Em AED I você construiu a ferramenta; agora você **aplica a ferramenta**.

- **"Preciso percorrer ou processar na ordem de chegada / O mais antigo primeiro"**
  - **Conceito de AED I:** Fila (FIFO - *First In, First Out*).
  - **Em Java:** `Queue<T>` usando `ArrayDeque` ou `LinkedList`.
  - **Exemplo Clássico:** Simulações de filas de banco, jogos de baralho (descarte pelo topo) ou em Grafos com Busca em Largura (BFS).

- **"Preciso desfazer a última operação / Validar pares perfeitamente aninhados"**
  - **Conceito de AED I:** Pilha (LIFO - *Last In, First Out*).
  - **Em Java:** `Deque<T>` usando `ArrayDeque` (via `push()` e `pop()`).
  - **Exemplo Clássico:** O problema do balanço de parênteses/chaves matemáticas '{[()]}' ou buscas em profundidade (DFS).

5.2. Do enunciado à estrutura: Mapas e Conjuntos

Muitos problemas exigem eficiência extrema de busca. Listas lineares (como `ArrayList`) forçariam uma busca $O(N)$, o que causa **Time Limit Exceeded (TLE)**.

- **"Preciso contar quantas vezes cada elemento aparece / Associar chave → valor"**
  - **Conceito de AED I:** Dicionário ou Tabela de Símbolos.
  - **Em Java:** `HashMap<K,V>` (busca em $O(1)$) ou `TreeMap<K,V>` (se precisar processar ordenado em $O(\log N)$).
  - **Exemplo Clássico:** Contar frequências de palavras em um texto ou simular o carrinho de compras associando Produto e Preço.

- **"Preciso manter uma coleção de elementos distintos (sem repetições)"**
  - **Conceito de AED I:** Conjunto matemático (Set).
  - **Em Java:** `HashSet<E>` ($O(1)$ rápido) ou `TreeSet<E>` (ordenado).
  - **Exemplo Clássico:** Dado um vetor com milhares de IDs de usuários, responder rapidamente quantos IDs *únicos* existem.

Pensamento essencial: **"Qual estrutura modela melhor o comportamento pedido?"**

4. Mapas e Conjuntos (Visão Rápida)

Embora não sejam "lineares", vamos usá-los tanto que vale citar.

- **HashSet / HashMap:** Usam Tabela Hash. Inserção e Busca em $O(1)$ médio. A ordem é aleatória.
- **TreeSet / TreeMap:** Usam Árvore Rubro-Negra. Inserção e Busca em $O(\log N)$. Mantêm os dados **ordenados**.

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<Integer>`.

Fast I/O (Leitura Rápida)

O Scanner é lento porque faz muito parsing. Para problemas com $N > 10^5$, use BufferedReader. **Setup do template:**

```
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.in));
        PrintWriter out = new PrintWriter(System.out);
        StringTokenizer st = new StringTokenizer("");
```

5. Dicas de Ouro: Fast I/O (Leitura Rápida) — parte 2

Exemplo: ler N e N inteiros

```
        // 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.
    }
}
```

6. Resumo Comparativo: Java Collections

Para **grandes volumes de dados** ou maratonas de programação, a escolha da estrutura correta é a diferença entre o **Accepted (AC)** e o **Time Limit Exceeded (TLE)**.

Classe	Ordenada?	Duplicados?	Estrutura Base	Busca
`ArrayList`	Sim (Índice)	Sim	Array Dinâmico	$O(1)$ (índice)
`LinkedList`	Sim (Índice)	Sim	Lista Encadeada	$O(n)$
`HashSet`	Não	Não	Tabela Hash	$O(1)$ (médio)
`TreeSet`	Sim (Natural)	Não	Árvore R-N	$O(\log n)$
`HashMap`	Não	Chaves Ún.	Tabela Hash	$O(1)$ (médio)
`TreeMap`	Sim (Chaves)	Chaves Ún.	Árvore R-N	$O(\log n)$

6.1. Exemplos de Uso e Decisão Técnica

Cenários em que certas estruturas são indispensáveis:

- **Histórico (Undo/Redo):** Se a aplicação exige inserções e remoções constantes apenas nas extremidades, use a `LinkedList` (ou `ArrayDeque`). Evita o *resize* em memória de um `ArrayList`.
- **Elementos Únicos:** Ex: achar quantos itens são únicos numa lista de 10^6 CPFs. Insira-os num `HashSet` (inserção $O(1)$). Algoritmo $O(N)$ global, sem sofrer com buscas $O(N^2)$.
- **Ranking em Tempo Real:** Placar atualizado a qualquer momento. O `TreeSet` / `TreeMap` os mantém ordenados ($O(\log n)$ por inserção), evitando um custoso `sort()` a cada entrada.

7. Problemas Clássicos para Treinar

Utilize problemas do site **Beecrowd** para fixar qual classe do *Java Collections* resolve melhor cada cenário:

Aplicações Práticas

- **Pilhas (Stack / Deque): Maratona 1068** – Balanço de Parênteses
  Objetivo: Verificar se uma expressão matemática abre e fecha os parênteses na ordem correta.
- **Filas (Queue): Maratona 1110** – Jogando Cartas Fora
  Objetivo: Simular o descarte e o movimento de cartas para a base da pilha até sobrar uma.
- **Mapas (HashMap / TreeMap): Maratona 1281** – Ida à Feira
  Objetivo: Associar rapidamente o nome de um produto (String) ao seu preço (Double) para somar os gastos do carrinho em $O(1)$.