Documentação do Sistema de Simulação de Coleta de Lixo de Teresina

Maio de 2025

Contents

1 Introdução

O sistema de simulação de coleta de lixo de Teresina é uma aplicação Java modular que modela a logística de coleta, transferência e descarte de resíduos urbanos na cidade de Teresina. Utiliza uma interface de linha de comando (CLI) implementada na classe `InterfaceSimulador`, avançando minuto a minuto para simular realisticamente a geração de lixo, alocação de caminhões, operações em estações de transferência e transporte ao aterro sanitário. A classe `Estatisticas` gerencia métricas detalhadas, como lixo coletado, transportado, tempos de espera e uso de caminhões, enquanto `DistribuicaoCaminhoes` otimiza a alocação de caminhões pequenos às zonas urbanas com base em lixo acumulado, distância e horários de pico. O sistema é organizado em pacotes (`caminhoes`, `estacoes`, `estruturas`, `simulacao`, `zonas`), usa tipos abstratos de dados (TADs) personalizados (`Lista<T>`, `Fila<T>`, `MapaEventos`, `No<T>`) e inclui um sistema de logging (`LoggerSimulacao`) com modos Normal e Debug.

Este documento detalha a implementação, incluindo TADs, classes principais, algoritmos, gerenciamento de estatísticas e logs, e a interface CLI. São abordados o uso das TADs, a lógica de simulação, e cálculos para otimização logística, com foco na modularidade e precisão das métricas.

2 Visão Geral do Sistema

O sistema modela a logística de resíduos sólidos em Teresina, considerando cinco zonas urbanas (Norte, Sul, Leste, Sudeste, Centro), caminhões pequenos (2t, 4t, 8t, 10t) para coleta, duas estações de transferência (A e B), caminhões grandes (20t) para transporte ao aterro sanitário, e uma interface CLI para configuração e controle. A simulação opera em minutos simulados, com geração diária de lixo, alocação dinâmica de caminhões, coleta, transferência, transporte e descarte. A classe `Estatisticas` registra métricas detalhadas, incluindo:

- Quantidade total de lixo gerado e coletado, geral e por zona.
- Quantidade de lixo transportado ao aterro.
- Tempo médio de espera dos caminhões pequenos nas filas.
- Número total e máximo simultâneo de caminhões grandes utilizados.
- Número de descarregamentos nas estações.
- Indicadores de desempenho operacional (eficiência de alocação e tempos de espera).

Os principais componentes são:

- **Zonas Urbanas** (`ZonaUrbana`): Representam regiões que geram lixo diariamente, com intervalos mínimo e máximo configuráveis. Rastreiam lixo acumulado, caminhões ativos e tempos de viagem (normais ou de pico). A classe `ZonaEstatistica` agrega dados estatísticos por zona, como lixo gerado e coletado.
- **Caminhões Pequenos** (`CaminhaoPequeno`): Coletam lixo nas zonas, com capacidades fixas e limite de viagens diárias. Operam em estados como `COLETANDO`, `EM_TRÂNSITO`, `FILA_ESTAÇÃO`, com coleta incremental.

- **Caminhões Grandes** (`CaminhaoGrande`): Transportam lixo das estações ao aterro, com capacidade de 20t. São liberados quando cheios ou após exceder a tolerância de espera configurável.
- **Estações de Transferência** (`EstacaoTransferencia`): Gerenciam filas de caminhões pequenos (`Fila<CaminhaoPequeno>`), listas de caminhões grandes esperando (`Lista<CaminhaoGrande>`), e descarregamentos ativos (`Lista<Descarregamento>`). Controlam descarregamento progressivo e tempos de espera.
- **Aterro Sanitário**: Localizado em uma zona urbana, é o destino final do lixo.
- **Distribuição de Caminhões** (`DistribuicaoCaminhoes`): Aloca caminhões pequenos dinamicamente, usando critérios como lixo acumulado, distância e horários de pico.
- **Geração de Placas** (`Placa`): Gera e valida placas no padrão Mercosul (LLLNLNN) para caminhões.
- **Interface CLI** (`InterfaceSimulador`): Permite configurar parâmetros, iniciar/-pausar/encerrar a simulação, exibir e salvar relatórios.
- **Logging** (`LoggerSimulacao`): Registra eventos com formatação colorida, modos Normal e Debug, e exporta logs para arquivo.
- **Auxiliares**: `ResultadoProcessamentoFila` indica resultados de processamento de filas; `Descarregamento` (interna a `EstacaoTransferencia`) rastreia operações de descarregamento.

2.1 Organização por Pacotes

O sistema é estruturado em cinco pacotes para modularidade:

Pacote	Conteúdo Principal
`caminhoes`	Classes relacionadas a caminhões: `CaminhaoPequeno`, `CaminhaoGrande`, `DistribuicaoCaminhoes`, `Placa`.
`estacoes`	Lógica das estações: `EstacaoTransferencia`, `ResultadoProcessamentoFila`, `Descarregamento`.
`estruturas`	TADs personalizados: `Lista<T>`, `Fila<T>`, `MapaEventos`, `No<T>`.
`simulacao`	Lógica principal: `Simulador`, `Estatisticas`, `LoggerSimulacao`, `InterfaceSimulador`.
`zonas`	Zonas urbanas e estatísticas: `ZonaUrbana`, `ZonaEstatistica`.

Table 1: Organização por Pacotes

3 TADs Implementadas

O sistema utiliza quatro TADs personalizados (`No<T>`, `Lista<T>`, `Fila<T>`, `MapaEventos`) implementados no pacote `estruturas`, em vez de coleções padrão do Java, para maior controle e demonstração de estruturas encadeadas e de mapeamento.

3.1 TAD No (`No<T>`)

3.1.1 Descrição

A classe `No<T>` é uma estrutura auxiliar que representa um nó genérico em listas encadeadas, usada como base para `Lista<T>` e `Fila<T>`.

3.1.2 Implementação

```
package estruturas;

class No<T> {
    T dado;
    No<T> prox;

    No(T dado) {
        this.dado = dado;
        this.prox = null;
    }
}
```

3.1.3 Características Principais

- **Genérica**: Suporta qualquer tipo de dado.
- **Estrutura Encadeada**: Contém um dado e um ponteiro para o próximo nó.

3.2 TAD Lista (`Lista<T>`)

3.2.1 Descrição

A classe `Lista<T>` implementa uma lista simplesmente encadeada genérica, permitindo adição, obtenção e remoção de elementos por índice. É usada para armazenar coleções dinâmicas de zonas, caminhões, estações, descarregamentos ativos e estatísticas.

3.2.2 Implementação

```
package estruturas;

public class Lista<T> {
    private No<T> head;
    private int tamanho;

    public Lista() {
```

```
8           this.head = null;
9           this.tamanho = 0;
10      }
11
12      public void adicionar(T dado) {
13          No<T> novoNo = new No<>(dado);
14          if (head == null) {
15              head = novoNo;
16          } else {
17              No<T> atual = head;
18              while (atual.prox != null) {
19                  atual = atual.prox;
20              }
21              atual.prox = novoNo;
22          }
23          tamanho++;
24      }
25
26      public T obter(int indice) {
27          if (indice < 0 || indice >= tamanho) {
28              throw new IndexOutOfBoundsException("ndice invlido: " + indice);
29          }
30          No<T> atual = head;
31          for (int i = 0; i < indice; i++) {
32              atual = atual.prox;
33          }
34          return atual.dado;
35      }
36
37      public T remover(int indice) {
38          if (indice < 0 || indice >= tamanho) {
39              throw new IndexOutOfBoundsException("ndice invlido: " + indice);
40          }
41          T removido;
42          if (indice == 0) {
43              removido = head.dado;
44              head = head.prox;
45          } else {
46              No<T> anterior = head;
47              for (int i = 0; i < indice - 1; i++) {
48                  anterior = anterior.prox;
49              }
50              removido = anterior.prox.dado;
51              anterior.prox = anterior.prox.prox;
52          }
53          tamanho--;
54          return removido;
55      }
56
57      public void limpar() {
58          head = null;
```

```
59          tamanho = 0;
60      }
61
62      public int getTamanho() {
63          return tamanho;
64      }
65
66      public boolean estaVazia() {
67          return tamanho == 0;
68      }
69  }
```

3.2.3 Características Principais

- **Genérica**: Suporta qualquer tipo de dado.
- **Estrutura Encadeada**: Baseada em nós `No<T>`, sem limite fixo de tamanho.
- **Operações Básicas**: Adição no final, acesso/remoção por índice, verificação de tamanho, limpeza.

3.2.4 Operações e Complexidade

Operação	Método	Complexidade	Descrição
Inserção	`adicionar(T dado)`	$\mathcal{O}(n)$	Adiciona um elemento ao final da lista.
Acesso	`obter(int indice)`	$\mathcal{O}(n)$	Acessa um elemento pelo índice.
Remoção	`remover(int indice)`	$\mathcal{O}(n)$	Remove um elemento pelo índice.
Tamanho	`getTamanho()`	$\mathcal{O}(1)$	Retorna a quantidade de elementos.
Vazio	`estaVazia()`	$\mathcal{O}(1)$	Verifica se a lista está vazia.
Limpeza	`limpar()`	$\mathcal{O}(1)$	Remove todos os elementos.

Table 2: Operações da TAD `Lista<T>`

3.3 TAD Fila (`Fila<T>`)

3.3.1 Descrição

A classe `Fila<T>` implementa uma fila genérica baseada em nós encadeados, seguindo o princípio FIFO (First-In-First-Out). Utiliza uma estrutura circular (o último nó aponta para o primeiro) para gerenciar a ordem de chegada dos caminhões pequenos nas estações de transferência.

3.3.2 Implementação

```
package estruturas;

public class Fila<T> {
    private No<T> head;
    private No<T> tail;
    private int tamanho;

    public Fila() {
        this.head = null;
        this.tail = null;
        this.tamanho = 0;
    }

    public void enfileirar(T dado) {
        No<T> novoNo = new No<>(dado);
        if (estaVazia()) {
            head = novoNo;
            tail = novoNo;
            tail.prox = head;
        } else {
            tail.prox = novoNo;
            tail = novoNo;
            tail.prox = head;
        }
        tamanho++;
    }

    public T remover() {
        if (estaVazia()) {
            throw new RuntimeException("Fila vazia!");
        }
        T dadoRemovido = head.dado;
        head = head.prox;
        tamanho--;
        if (head == null) {
            tail = null;
        } else {
            tail.prox = head;
        }
        return dadoRemovido;
    }

    public T primeiroDaFila() {
        if (estaVazia()) {
            throw new NoSuchElementException("A fila est vazia!");
        }
        return head.dado;
    }

    public T obter(int indice) {
```

```
51          if (estaVazia()) {
52              throw new NoSuchElementException("A fila est vazia!");
53          }
54          if (indice < 0 || indice >= tamanho) {
55              throw new IndexOutOfBoundsException("ndice invlido: " + indice);
56          }
57          No<T> atual = head;
58          for (int i = 0; i < indice; i++) {
59              atual = atual.prox;
60          }
61          return atual.dado;
62      }
63
64      public boolean estaVazia() {
65          return tamanho == 0;
66      }
67
68      public int getTamanho() {
69          return tamanho;
70      }
71  }
```

3.3.3 Características Principais

- **Estrutura Encadeada**: Baseada em nós `No<T>`, sem limite fixo.
-
- **FIFO**: Garante que o primeiro elemento inserido é o primeiro removido.
- **Genérica**: Suporta qualquer tipo de dado.
- **Estrutura Circular**: O último nó aponta para o primeiro, otimizando enfileiramento.

3.3.4 Operações e Complexidade

Operação	Método	Complexidade	Descrição
Enfileirar	`enfileirar(T dado)`	$\mathcal{O}(1)$	Adiciona um elemento ao final da fila.
Desenfileirar	`remover()`	$\mathcal{O}(1)$	Remove e retorna o primeiro elemento.
Frente	`primeiroDaFila()`	$\mathcal{O}(1)$	Retorna o primeiro elemento sem removê-lo.
Acesso	`obter(int indice)`	$\mathcal{O}(n)$	Acessa um elemento por índice.
Tamanho	`getTamanho()`	$\mathcal{O}(1)$	Retorna a quantidade de elementos.
Vazio	`estaVazia()`	$\mathcal{O}(1)$	Verifica se a fila está livre.

Table 3: Operações da TAD `Fila<T>`

3.4 TAD MapaEventos (`MapaEventos`)

3.4.1 Descrição

A classe `MapaEventos` implementa um mapeamento chave-valor, onde chaves são strings (tipos de eventos, e.g., "COLETA", "ERRO") e valores são códigos de cores). É usado no sistema de logging para formatação colorida. A busca linear tem complexidade $\mathcal{O}(n)$, sugerindo otimização (ver Seção 6).

Implementação

```
package estruturas;

public class MapaEventos {
    private static class ParEventoCor {
        String evento;
        String cor;

        ParEventoCor(String evento, String cor) {
            this.evento = evento;
            this.cor = cor;
        }
    }

    private final Lista<ParEventoCor> pares;

    public MapaEventos() {
        this.pares = new Lista<>();
    }

    public void put(String evento, String cor) {
        for (int i = 0; i < pares.getTamanho(); i++) {
            if (pares.obter(i).evento.equals(evento)) {
                pares.remover(i);
                break;
            }
        }
        pares.adicionar(new ParEventoCor(evento, cor));
    }

    public String get(String evento) {
        for (int i = 0; i < pares.getTamanho(); i++) {
            if (pares.obter(i).evento.equals(evento)) {
                return pares.obter(i).cor;
            }
        }
        return null;
    }
}
```

Características Principais

- **Chave-Valor**: Associa eventos a cores.
- **Estrutura Baseada em Lista**: Usa `Lista<T>` para pares evento-cor.
- **Substituição de Chave**: Remove chaves existentes antes de adicionar novo par.

Operações e Complexidade

Operação	Método	Complexidade	Descrição
Inserção	`put(String, String)`	$\mathcal{O}(n)$	Adiciona ou substitui um par evento-cor.
Busca	`get(String)`	$\mathcal{O}(n)$	Recupera a cor associada a um evento.

Table 4: Operações da TAD `MapaEventos`

4 Uso das TADs no Sistema

4.1 Uso da TAD Lista

A `Lista<T>` é amplamente utilizada para gerenciar coleções dinâmicas no sistema.
No Simulador Na classe `Simulador`, a `Lista<T>` armazena:

- Zonas urbanas (`Lista<ZonaUrbana> zonas`).
- Caminhões pequenos (`Lista<CaminhaoPequeno> caminhoesPequenos`).
- Estações de transferência (`Lista<EstacaoTransferencia> estacoes`).
- Caminhões grandes ocupados (`Lista<CaminhaoGrande> caminhoesGrandesOcupados`).
- Todos os caminhões grandes (`Lista<CaminhaoGrande> todosCaminhoesGrandes`).

```
1 private static Lista<CaminhaoPequeno> caminhoesPequenos;
2 private static Lista<ZonaUrbana> zonas;
3 private static Lista<EstacaoTransferencia> estacoes;
4 private static Lista<CaminhaoGrande> caminhoesGrandesOcupados = new Lista<>();
5 private static Lista<CaminhaoGrande> todosCaminhoesGrandes = new Lista<>();
```

Nas Estações de Transferência A classe `EstacaoTransferencia` utiliza `Lista<T>` para:

- Armazenar caminhões grandes esperando (`Lista<CaminhaoGrande> caminhoesGrandesEsperand`
- Gerenciar descarregamentos ativos (`Lista<Descarregamento> descarregamentosAtivos`).
- Listar caminhões grandes atribuídos (`Lista<CaminhaoGrande> listaGrandes`).

```
1 private final Lista<CaminhaoGrande> listaGrandes;
2 private final Lista<CaminhaoGrande> caminhoesGrandesEsperando;
3 private final Lista<Descarregamento> descarregamentosAtivos;
```

Nas Estatísticas A classe `Estatisticas` usa `Lista<ZonaEstatistica>` para rastrear estatísticas por zona:

```
private final Lista<ZonaEstatistica> lixoPorZona;
```

No Logger A `MapaEventos` utiliza `Lista<ParEventoCor>` para mapear eventos a cores:

```
private final Lista<ParEventoCor> pares;
```

Uso da TAD Fila A `Fila<T>` é usada nas estações de transferência para gerenciar a ordem de chegada dos caminhões pequenos.

Na EstacaoTransferencia

```
private final Fila<CaminhaoPequeno> filaPequenos;
public EstacaoTransferencia(String nome, int esperaMaxPequenos, ZonaUrbana
    zonaDaEstacao) {
    this.filaPequenos = new Fila<>();
    // Inicializao restante
}
public void receberCaminhaoPequeno(CaminhaoPequeno caminhao, int tempoAtual) {
    filaPequenos.enfileirar(caminhao);
    caminhao.incrementarTempoEspera();
    LoggerSimulacao.log("CHEGADA", String.format("%s: Caminho %s chegou  estao
        e entrou na fila. Tamanho da fila: %d",
            nome, caminhao.getPlaca(), filaPequenos.getTamanho()));
}
```

A fila garante a ordem FIFO, processando caminhões na sequência de chegada.

Uso da TAD MapaEventos A `MapaEventos` é usada na classe `LoggerSimulacao` para mapear tipos de eventos a cores ANSI, permitindo formatação colorida no console.

```
private static final MapaEventos CORES_EVENTO = new MapaEventos();
static {
    CORES_EVENTO.put("COLETA", VERDE);
    CORES_EVENTO.put("CHEGADA", AZUL_CLARO);
    CORES_EVENTO.put("DESCARGA", CIANO_CLARO);
    CORES_EVENTO.put("ERRO", VERMELHO);
    CORES_EVENTO.put("INFO", AMARELO_CLARO);
    CORES_EVENTO.put("CONFIG", RESET);
    CORES_EVENTO.put("ESTATISTICA", RESET);
    CORES_EVENTO.put("VIAGEM", AZUL);
    CORES_EVENTO.put("ATRIBUICAO", MAGENTA_CLARO);
    CORES_EVENTO.put("ADIO", VERDE_CLARO);
}
```

5 Algoritmos Relevantes Utilizando as TADs

Processamento de Fila nas Estações O método `processarFila` na classe `EstacaoTransferencia` gerencia a fila de caminhões pequenos e a transferência de lixo para caminhões grandes, registrando tempos de espera em `Estatisticas`:

```
public ResultadoProcessamentoFila processarFila(int tempoAtual) {
    int esperaAcumulada = atualizarEsperaCaminhoesPequenos();
```

```
    esperaTotalPequenos += esperaAcumulada;
    processarDescarregamentosAtivos();
    if (!filaPequenos.estaVazia() && !caminhoesGrandesEsperando.estaVazia()) {
        return descarregarCaminhaoPequeno(tempoAtual);
    }
    return new ResultadoProcessamentoFila(null, 0, false);
}
```

Este algoritmo:

- Usa `Fila<CaminhaoPequeno>` para processar caminhões na ordem de chegada.
- Usa `Lista<Descarregamento>` para rastrear descarregamentos em andamento.
- Usa `Lista<CaminhaoGrande>` para gerenciar caminhões grandes disponíveis.
- Calcula tempos de espera, contribuindo para `tempoTotalEsperaPequenos` em `Estatisticas`.
- Decide quando liberar caminhões grandes com base na tolerância de espera.

Distribuição de Caminhões Pequenos O método `distribuirCaminhoes` na classe `DistribuicaoCaminhoes` aloca caminhões pequenos para zonas com lixo acumulado, usando um sistema de pontuação baseado em lixo acumulado, distância e número de caminhões ativos:

```
public int distribuirCaminhoes(Lista<CaminhaoPequeno> caminhoes, Lista<
    ZonaUrbana> zonas) {
    int distribuidos = 0;
    for (int i = 0; i < caminhoes.getTamanho(); i++) {
        CaminhaoPequeno caminhao = caminhoes.obter(i);
        if (isCaminhaoDisponivel(caminhao)) {
            ZonaUrbana melhorZona = encontrarMelhorZona(caminhao, zonas);
            ZonaUrbana zonaAtual = caminhao.getZonaAtual();
            if (melhorZona == null) {
                caminhao.setEstado(6); // ENCERRADO
            } else if (melhorZona != zonaAtual) {
                int tempoViagem = calcularTempoViagem(caminhao.getZonaAtual(),
                    melhorZona);
                caminhao.definirTempoViagem(tempoViagem);
                caminhao.setEstado(3); // EM_TRNSITO
                caminhao.setZonaDestino(melhorZona);
                distribuidos++;
            } else {
                caminhao.setEstado(2); // COLETANDO
            }
        }
    }
    return distribuidos;
}
```

Este algoritmo:

- Usa `Lista<CaminhaoPequeno>` e `Lista<ZonaUrbana>` para iterar sobre caminhões e zonas.

- Calcula pontuações em `encontrarMelhorZona` com base em lixo acumulado, distância e caminhões ativos.
- Usa `calcularTempoViagem` para estimar tempos de viagem, considerando horários de pico.

Simulação Principal O método `executarSimulacao` na classe `Simulador` coordena a simulação, gerando relatórios horários via `Estatisticas`:

```
private void executarSimulacao() {
    while (rodando) {
        if (!pausado) {
            atualizarSimulacao();
        }
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            LoggerSimulacao.log("ERRO", "Erro na simulao: " + e.getMessage());
        }
    }
}
private void atualizarSimulacao() {
    tempoSimulado++;
    if (tempoSimulado % TEMPO_MINUTOS_POR_DIA == 0 && tempoSimulado != 1) {
        concluirDia();
    }
    if (temLixoOuTrabalhoPendente() && tempoSimulado % 10 == 0) {
        distribuirCaminhoesDisponiveis();
    }
    processarCaminhoesPequenos();
    processarEstacoes();
    processarCaminhoesGrandesOcupados();
    if (tempoSimulado % 60 == 0) {
        estatisticas.setTempoSimulado(tempoSimulado);
        estatisticas.imprimirRelatorio();
    }
}
```

Este algoritmo:

- Usa `Lista<T>` para gerenciar zonas, caminhões e estações.
- Sincroniza a simulação com pausas e relatórios a cada hora simulada.
- Coordena geração de lixo, coleta, transporte e descarregamento.

Fluxo da Simulação O sistema opera em um ciclo minuto a minuto, controlado pela classe `Simulador`:

1. **Configuração**: O usuário define parâmetros via `InterfaceSimulador`, incluindo tolerância de espera dos caminhões grandes, tempo máximo de espera dos pequenos, limite de viagens diárias, intervalos de geração de lixo, localizações de estações e aterro, e quantidades de caminhões pequenos por capacidade.

2. **Início do Ciclo**: O `Simulador` inicia a simulação, avançando `tempoSimulado`.

3. **Geração de Lixo**: Cada `ZonaUrbana` gera lixo diário com base em intervalos configurados, registrado em `Estatisticas`.

4. **Alocação de Caminhões**: `DistribuicaoCaminhoes` aloca `CaminhaoPequeno` às zonas com lixo acumulado, usando pontuações baseadas em lixo, distância e horários de pico.

5. **Coleta**: Caminhões pequenos coletam lixo incrementalmente, atualizando `lixoColetado` em `ZonaEstatistica`.

6. **Descarregamento**: `EstacaoTransferencia` processa `Fila<CaminhaoPequeno>`, transferindo lixo para `CaminhaoGrande` via `Descarregamento`, com tempos de espera registrados.

7. **Transporte ao Aterro**: Caminhões grandes viajam ao aterro, descarregam e retornam, atualizando `totalLixoAterro`.

8. **Logging**: `LoggerSimulacao` registra eventos (e.g., "COLETA", "CHEGADA") com cores, salvando logs em arquivo.

9. **Estatísticas**: `Estatisticas` consolida métricas (lixo gerado/coletado, tempos de espera, caminhões usados) e gera relatórios horários ou sob demanda.

10. **Controle**: O usuário pode pausar, continuar ou encerrar a simulação, salvando relatórios via `InterfaceSimulador`.

6 Análise Crítica das TADs Implementadas

Pontos Fortes

1. **Encapsulamento**: As TADs oferecem interfaces claras, escondendo detalhes de implementação.

2. **Reuso**: Estruturas genéricas são amplamente reutilizadas em diferentes contextos.

3. **Controle**: Implementações personalizadas permitem ajustes específicos, como a estrutura circular da `Fila<T>`.

4. **Simplicidade**: As TADs são adequadas ao escopo do projeto, com operações suficientes para a simulação.

Limitações e Possíveis Melhorias

1. **Falta de Iterador**: A ausência de suporte ao `Iterable<T>` exige iterações manuais com `obter(int)`.

2. **Operações Limitadas**: Métodos como `contem(T)` ou `indiceDe(T)` não estão presentes.

3. **Ineficiência em Acesso**: Acesso por índice em `Lista<T>` e `Fila<T>` tem complexidade $\mathcal{O}(n)$.

4. **MapaEventos Ineficiente**: A busca linear em `MapaEventos` pode ser lenta para muitos eventos.

Sugestões de Melhorias Implementar Interface `Iterable<T>`

```
public class Lista<T> implements Iterable<T> {
    @Override
    public Iterator<T> iterator() {
        return new Iterator<T>() {
            private No<T> atual = head;
            @Override
            public boolean hasNext() { return atual != null; }
            @Override
            public T next() {
                T dado = atual.dado;
                atual = atual.prox;
                return dado;
            }
        };
    }
}
```

Adicionar Métodos Utilitários

```
public boolean contem(T elemento) {
    No<T> atual = head;
    while (atual != null) {
        if (atual.dado.equals(elemento)) return true;
        atual = atual.prox;
    }
    return false;
}
```

Otimizar MapaEventos

```
public class MapaEventos {
    private final HashMap<String, String> pares = new HashMap<>();
    public void put(String evento, String cor) {
        pares.put(evento, cor);
    }
    public String get(String evento) {
        return pares.get(evento);
    }
}
```

7 Interface de Linha de Comando

A classe `InterfaceSimulador` implementa uma interface CLI com as seguintes funcionalidades:

- **Configuração**: Permite definir parâmetros como tolerância de espera dos caminhões grandes (em minutos), tempo máximo de espera dos caminhões pequenos

nas estações de transferência, limite de viagens diárias por caminhão pequeno, intervalos de geração de lixo por zona, localizações das estações de transferência e do aterro sanitário, e quantidades de caminhões pequenos por capacidade (2t, 4t, 8t, 10t).

- **Controles**: Oferece comandos para iniciar, pausar, continuar ou encerrar a simulação, além de opções para exibir relatórios em tempo real ou salvá-los em arquivo.
- **Estatísticas**: Permite ao usuário solicitar a exibição ou salvamento de relatórios estatísticos, que são gerenciados pela classe `Estatisticas`. As métricas incluem:
  - Quantidade total de lixo coletado por zona e no geral.
  - Quantidade total de lixo transportado ao aterro sanitário.
  - Tempo médio de espera dos caminhões pequenos nas filas das estações de transferência.
  - Número total de caminhões grandes (20 toneladas) utilizados.
  - Número máximo de caminhões grandes em uso simultaneamente, indicando o mínimo necessário para atender à demanda.
  - Quantidade total de lixo gerado por zona.
  - Número de descarregamentos realizados nas estações.
  - Indicadores de desempenho operacional, como eficiência na alocação de caminhões (baseada no número de descarregamentos e no uso de caminhões grandes) e tempos de espera.

```
1  public class InterfaceSimulador {
2      private final Simulador simulador;
3      private final Scanner scanner;
4
5      public InterfaceSimulador(Simulador simulador) {
6          this.simulador = simulador;
7          this.scanner = new Scanner(System.in);
8      }
9
10     public void iniciar() {
11         configurarSimulador();
12         mostrarMenuCompleto();
13         while (true) {
14             int opcao = lerOpcao();
15             processarOpcao(opcao);
16         }
17     }
18 }
```

8 Logging de Sistema de Logística

A classe `LoggerSimulacao` gerencia logs com dois modos (`NORMAL` e `DEBUG`), usando `MapaEventos` para formatação colorida e salvando dados em arquivo:

```java
public class LoggerSimulacao {
    public enum ModoLog { NORMAL, DEBUG }
    private static ModoLog modoAtual = ModoLog.NORMAL;
    private static PrintWriter escritorArquivoLog;

    public static void log(String tipoEvento, String mensagem) {
        String cor = CORES_EVENTO.get(tipoEvento);
        String timestamp = String.formatarTempo(Simulador.getTempoSimulado());
        String mensagemFormatada = String.format("%s[S%s] %s%s%s", cor,
            timestamp, mensagem, RESET);
        // Exibe no console e salva no sistema
    }
}
```

9 Conclusão

O sistema de simulação de coleta de lixo de Teresina é uma solução robusta e modular implementada em Java, modelando realisticamente a logística de resíduos urbanos. As TADs `Lista<T>`, `Fila<T>`, `MapaEventos` e `No<T>` suportam o gerenciamento eficiente de coleções, filas de espera, e logs estilizados. A classe `Estatisticas` fornece métricas detalhadas, incluindo o número mínimo de caminhões grandes necessários e indicadores de desempenho operacional, essenciais para otimização logística. A estrutura por pacotes (`caminhoes`, `estacoes`, `estruturas`, `simulacao`, `zonas`) e a interface CLI (`InterfaceSimulador`) garantem modularidade e usabilidade. Apesar de limitações, como a busca linear em `MapaEventos` e a ausência de iteradores, o sistema atende aos requisitos com precisão, oferecendo flexibilidade para análises logísticas detalhadas.

10 Apêndices

Termos e Definições

Termo	**Definição**
`TAD`	Tipo Abstrato de Dados - Modelo matemático definido por suas operações.
`Lista`	Estrutura linear baseada em nós ligados, permitindo acesso sequencial.
`Fila`	Estrutura linear que segue o princípio FIFO (First-In-First-Out).
`MapaEventos`	Estrutura de mapeamento chave-valor para associar eventos a cores.
`FIFO`	Primeiro a entrar, primeiro a sair.
$\mathcal{O}(1)$	Complexidade constante, independente do tamanho da entrada.
$\mathcal{O}(n)$	Complexidade linear, proporcional ao tamanho da entrada.

Table 5: Termos e Definições