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 desenvolvida em Java com interface gráfica utilizando a biblioteca Swing. Ele simula o processo de coleta, transporte e descarte de resíduos urbanos na cidade de Teresina, considerando cinco zonas urbanas, caminhões pequenos para coleta, estações de transferência e caminhões grandes para transporte ao aterro sanitário.

Este documento detalha a implementação do sistema, com ênfase nas estruturas de dados personalizadas (`ListaEncadeada<T>` e `Fila<T>`), suas aplicações no contexto da simulação e os algoritmos utilizados. Além disso, são descritas as classes principais, a interface gráfica e os cálculos realizados para otimização de recursos.

2 Visão Geral do Sistema

O sistema modela os seguintes elementos:

- **Zonas Urbanas**: Cinco áreas da cidade (Sul, Norte, Centro, Leste, Sudeste), cada uma gerando quantidades variáveis de lixo em intervalos de tempo simulados.
- **Caminhões Pequenos**: Veículos com capacidades de 2, 4, 8 ou 10 toneladas, responsáveis pela coleta de lixo nas zonas urbanas.
- **Estações de Transferência**: Locais onde os caminhões pequenos descarregam o lixo, que é então transferido para caminhões grandes.
- **Caminhões Grandes**: Veículos com capacidade fixa de 20 toneladas, que transportam o lixo das estações para o aterro sanitário.
- **Interface Gráfica**: Uma interface Swing que exibe um mapa da cidade, informações sobre zonas, caminhões, estações e estatísticas em tempo real.

A simulação opera em minutos simulados, com geração periódica de lixo, coleta, transporte e descarte. O sistema calcula métricas como eficiência da coleta, tempo médio de espera e número mínimo de caminhões grandes necessários.

3 TADs Implementadas

Para atender aos requisitos do projeto, foram implementadas duas estruturas de dados genéricas: `ListaEncadeada<T>` e `Fila<T>`. Essas estruturas foram escolhidas em vez de coleções padrão do Java para proporcionar maior controle e demonstrar a implementação de estruturas encadeadas.

3.1 TAD ListaEncadeada (`ListaEncadeada<T>`)

3.1.1 Descrição

A classe `ListaEncadeada<T>` implementa uma lista simplesmente encadeada genérica, permitindo a adição, obtenção e remoção de elementos. É utilizada para armazenar coleções de zonas, caminhões, estações de transferência e dados estatísticos.

3.1.2 Implementação

```java
package Estruturas;

public class ListaEncadeada<T> {
    private No<T> inicio;
    private int tamanho;

    private static class No<T> {
        T dado;
        No<T> proximo;

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

    public ListaEncadeada() {
        this.inicio = null;
        this.tamanho = 0;
    }

    public void adicionar(T elemento) {
        No<T> novoNo = new No<>(elemento);
        if (inicio == null) {
            inicio = novoNo;
        } else {
            No<T> atual = inicio;
            while (atual.proximo != null) {
                atual = atual.proximo;
            }
            atual.proximo = novoNo;
        }
        tamanho++;
    }

    public T obter(int indice) {
        if (indice < 0 || indice >= tamanho) {
            return null;
        }
        No<T> atual = inicio;
        for (int i = 0; i < indice; i++) {
            atual = atual.proximo;
        }
        return atual.dado;
    }

    public int tamanho() {
        return tamanho;
    }

```

```
51      public boolean estaVazia() {
52          return tamanho == 0;
53      }
54
55      public void limpar() {
56          inicio = null;
57          tamanho = 0;
58      }
59  }
```

3.1.3 Características Principais

- **Genérica**: Suporta qualquer tipo de dado.
- **Estrutura Encadeada**: Baseada em nós ligados, sem limite fixo de tamanho.
- **Operações Simples**: Inclui adição no final, acesso por índice, verificação de tamanho e limpeza.

3.1.4 Operações e Complexidade

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

Table 1: Operações da TAD `ListaEncadeada<T>`

3.2 TAD Fila (`Fila<T>`)

3.2.1 Descrição

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

3.2.2 Implementação

```
1   package Estruturas;
2
3   public class Fila<T> {
4       private No<T> inicio;
5       private No<T> fim;
6       private int tamanho;
7
8       private static class No<T> {
9           T dado;
10          No<T> proximo;
11
```

```
12          No(T dado) {
13              this.dado = dado;
14              this.proximo = null;
15          }
16      }
17
18      public Fila() {
19          this.inicio = null;
20          this.fim = null;
21          this.tamanho = 0;
22      }
23
24      public void enfileirar(T elemento) {
25          No<T> novoNo = new No<>(elemento);
26          if (estaVazia()) {
27              inicio = novoNo;
28              fim = novoNo;
29          } else {
30              fim.proximo = novoNo;
31              fim = novoNo;
32          }
33          tamanho++;
34      }
35
36      public T desenfileirar() {
37          if (estaVazia()) return null;
38          T elemento = inicio.dado;
39          inicio = inicio.proximo;
40          tamanho--;
41          if (estaVazia()) fim = null;
42          return elemento;
43      }
44
45      public T frente() {
46          if (estaVazia()) return null;
47          return inicio.dado;
48      }
49
50      public boolean estaVazia() {
51          return tamanho == 0;
52      }
53
54      public int tamanho() {
55          return tamanho;
56      }
57  }
```

3.2.3 Características Principais

- **Estrutura Encadeada**: Baseada em nós ligados, sem limite fixo de tamanho.

- **FIFO**: Garante que o primeiro elemento inserido é o primeiro a ser removido.
- **Genérica**: Suporta qualquer tipo de dado.

3.2.4 Operações e Complexidade

Operação	Método	Complexidade	Descrição
Enfileirar	`enfileirar(T elemento)`	O(1)	Adiciona um elemento ao final da fila.
Desenfileirar	`desenfileirar()`	O(1)	Remove e retorna o primeiro elemento.
Frente	`frente()`	O(1)	Retorna o primeiro elemento sem remo
Tamanho	`tamanho()`	O(1)	Retorna a quantidade de elementos.
Vazio	`estaVazia()`	O(1)	Verifica se a fila está vazia.

Table 2: Operações da TAD `Fila<T>`

4 Uso das TADs no Sistema

4.1 Uso da TAD ListaEncadeada

A classe `ListaEncadeada<T>` é utilizada para gerenciar coleções de entidades no sistema.

4.1.1 No Simulador

Na classe `Simulador`, a `ListaEncadeada` é usada para armazenar:

- Zonas (`ListaEncadeada<Zona> zonas`).
- Caminhões pequenos (`ListaEncadeada<CaminhaoPequeno> caminhoesPequenos`).
- Estações de transferência (`ListaEncadeada<EstacaoTransferencia> estacoesTransferencia`)
- Quantidades de lixo transportadas ao aterro (`ListaEncadeada<Double> lixoParaAterro`).
- Tempos de espera nas estações (`ListaEncadeada<Integer> temposDeEspera`).

```
1 private final ListaEncadeada<Zona> zonas;
2 private final ListaEncadeada<CaminhaoPequeno> caminhoesPequenos;
3 private final ListaEncadeada<EstacaoTransferencia>
     estacoesTransferencia;
4 private final ListaEncadeada<Double> lixoParaAterro;
5 private final ListaEncadeada<Integer> temposDeEspera;
```

4.1.2 Nas Estatísticas

A classe `EstacaoTransferencia` utiliza `ListaEncadeada` para armazenar tempos de espera:

```
1 private ListaEncadeada<Integer> temposEspera;
```

Isso permite calcular o tempo médio de espera dos caminhões pequenos.

4.1.3 Na Interface Gráfica

A classe `VisualizacaoSimulacaoSwing` usa `ListaEncadeada` para acessar e exibir informações sobre zonas, caminhões e estações em tempo real.

4.2 Uso da TAD Fila

A classe `Fila<T>` é utilizada nas estações de transferência para gerenciar a ordem de chegada dos caminhões pequenos.

4.2.1 Na EstacaoTransferencia

```
private Fila<CaminhaoPequeno> filaCaminhoesPequenos;

public EstacaoTransferencia(int id, int
    tempoMaxEsperaCaminhaoPequeno, int capacidadeCaminhaoGrande, int
    toleranciaEsperaCaminhaoGrande) {
     this.filaCaminhoesPequenos = new Fila<>();
     // Resto da inicialização
}

public void adicionarCaminhaoPequeno(CaminhaoPequeno caminhao, int
    tempoChegada) {
     caminhao.setTempoChegada(tempoChegada);
     filaCaminhoesPequenos.enfileirar(caminhao);
}
```

A fila mantém a ordem FIFO, garantindo que o primeiro caminhão a chegar seja o primeiro a descarregar.

5 Algoritmos Relevantes Utilizando as TADs

5.1 Processamento de Fila nas Estações

O método `processar` na classe `EstacaoTransferencia` gerencia a fila de caminhões pequenos e a transferência de lixo para caminhões grandes:

```
public boolean processar(int tempoAtual, ListaEncadeada<Double>
    lixoParaAterro) {
     boolean novoCaminhaoGrandeNecessario = false;

     if (caminhaoGrandeAtual != null) {
         if (tempoEsperaAtualGrande > toleranciaEsperaCaminhaoGrande
            && caminhaoGrandeAtual.getCargaAtual() > 0) {
             lixoParaAterro.adicionar(caminhaoGrandeAtual.getCargaAtual());
             caminhaoGrandeAtual = new CaminhaoGrande(id,
                capacidadeCaminhaoGrande, 0);
             tempoEsperaAtualGrande = 0;
             novoCaminhaoGrandeNecessario = true;
         } else if (caminhaoGrandeAtual.getCargaAtual() >=
            capacidadeCaminhaoGrande) {
```

```
11              lixoParaAterro.adicionar(caminhaoGrandeAtual.getCargaAtual());
12              caminhaoGrandeAtual = new CaminhaoGrande(id + 1,
                    capacidadeCaminhaoGrande, 0);
13              tempoEsperaAtualGrande = 0;
14              novoCaminhaoGrandeNecessario = true;
15          }
16      }
17
18      if (!filaCaminhoesPequenos.estaVazia() && caminhaoGrandeAtual
            != null) {
19          CaminhaoPequeno caminhao = filaCaminhoesPequenos.frente();
20          if (caminhao != null) {
21              int tempoEspera = tempoAtual -
                    caminhao.getTempoChegada();
22              if (tempoEspera >= 0) {
23                  temposEspera.adicionar(tempoEspera);
24                  caminhao = filaCaminhoesPequenos.desenfileirar();
25                  double lixoDescarregado = caminhao.descarregar();
26                  lixoArmazenado += lixoDescarregado;
27                  double lixoParaCaminhaoGrande =
                        Math.min(lixoArmazenado,
                        capacidadeCaminhaoGrande -
                        caminhaoGrandeAtual.getCargaAtual());
28                  caminhaoGrandeAtual.adicionarCarga(lixoParaCaminhaoGrande);
29                  lixoArmazenado -= lixoParaCaminhaoGrande;
30                  if (tempoEspera > tempoMaxEsperaCaminhaoPequeno) {
31                      precisaNovoCaminhaoGrande = true;
32                  }
33              }
34          }
35      }
36
37      if (caminhaoGrandeAtual != null &&
            caminhaoGrandeAtual.getCargaAtual() <
            capacidadeCaminhaoGrande) {
38          tempoEsperaAtualGrande++;
39      }
40
41      return novoCaminhaoGrandeNecessario ||
            precisaNovoCaminhaoGrande;
42  }
```

Este algoritmo:

- Usa a `Fila` para processar caminhões pequenos na ordem de chegada.
- Armazena tempos de espera em uma `ListaEncadeada` para estatísticas.
- Gerencia a carga do caminhão grande e decide quando substituí-lo.

5.2 Simulação Principal

O método `executarSimulacaoInterna` na classe `Simulador` coordena a simulação:

```
private void executarSimulacaoInterna(int duracao) {
    while (tempoSimulacao < duracao) {
        if (!simulacaoAtiva) {
            try { Thread.sleep(100); } catch (InterruptedException
               e) { e.printStackTrace(); }
            continue;
        }

        synchronized (zonas) {
            for (int i = 0; i < zonas.tamanho(); i++) {
                Zona zona = zonas.obter(i);
                if (zona != null) {
                    double lixoAntes = zona.getLixoAcumulado();
                    zona.gerarLixo(1);
                    lixoGeradoTotal += (zona.getLixoAcumulado() -
                       lixoAntes);
                }
            }
        }

        synchronized (caminhoesPequenos) {
            for (int i = 0; i < caminhoesPequenos.tamanho(); i++) {
                CaminhaoPequeno caminhao =
                   caminhoesPequenos.obter(i);
                if (caminhao != null &&
                   !caminhao.estaNaEstacaoTransferencia()) {
                    caminhao.coletarLixo();
                    if (caminhao.getCargaAtual() >=
                       caminhao.getCapacidade() * 0.8) {
                        EstacaoTransferencia estacao =
                           estacoesTransferencia.obter(new
                           Random().nextInt(estacoesTransferencia.tamanho()));
                        int tempoViagem =
                           caminhao.calcularTempoViagem(tempoSimulacao);
                        estacao.adicionarCaminhaoPequeno(caminhao,
                           tempoSimulacao + tempoViagem);
                    }
                }
            }
        }

        synchronized (estacoesTransferencia) {
            for (int i = 0; i < estacoesTransferencia.tamanho();
               i++) {
                EstacaoTransferencia estacao =
                   estacoesTransferencia.obter(i);
                if (estacao != null &&
                   estacao.processar(tempoSimulacao,
```

```
                    lixoParaAterro)) {
37                      caminhoesGrandesUsados++;
38                  }
39              }
40          }
41
42          tempoSimulacao++;
43      }
44      atualizarLixoAcumulado();
45      exibirEstatisticas();
46  }
```

Este algoritmo:

- Usa `ListaEncadeada` para iterar sobre zonas, caminhões e estações.
- Sincroniza o acesso para evitar condições de corrida.
- Calcula tempos de viagem e atualiza o estado da simulação.

6 Análise Crítica das TADs Implementadas

6.1 Pontos Fortes

1. **Encapsulamento**: As TADs oferecem interfaces claras, escondendo detalhes de implementação.
2. **Reuso**: Estruturas genéricas são usadas em todo o sistema.
3. **Controle**: A implementação própria permite ajustes específicos às necessidades do sistema.
4. **Simplicidade**: As TADs são fáceis de entender e adequadas ao escopo do projeto.

6.2 Limitações e Possíveis Melhorias

1. **Falta de Iterador**: A ausência de suporte ao `Iterable<T>` impede o uso de `for-each`, exigindo iterações manuais com `obter(int)`.
2. **Operações Limitadas**: Faltam métodos como `contem(T)` ou `indiceDe(T)`.
3. **Ineficiência em Acesso**: O acesso por índice em `ListaEncadeada` tem complexidade O(n).
4. **Sem Ordenação**: Não há algoritmos de ordenação integrados, o que seria útil para priorizar zonas ou caminhões.

6.3 Sugestões de Melhorias

6.3.1 Implementar Interface `Iterable<T>`

```
1 public class ListaEncadeada<T> implements Iterable<T> {
2     @Override
3     public Iterator<T> iterator() {
```

```
        return new Iterator<T>() {
            private No<T> atual = inicio;
            @Override
            public boolean hasNext() { return atual != null; }
            @Override
            public T next() {
                T dado = atual.dado;
                atual = atual.proximo;
                return dado;
            }
        };
    }
}
```

Isso permitiria iterações mais simples com `for-each`.

6.3.2 Adicionar Métodos Utilitários

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

public int indiceDe(T elemento) {
    No<T> atual = inicio;
    int indice = 0;
    while (atual != null) {
        if (atual.dado.equals(elemento)) return indice;
        atual = atual.proximo;
        indice++;
    }
    return -1;
}
```

Esses métodos facilitariam a busca e manipulação de elementos.

6.3.3 Algoritmo de Ordenação

Implementar um algoritmo como QuickSort para `ListaEncadeada` poderia melhorar a eficiência em tarefas como priorização de zonas.

7 Interface Gráfica

A classe `VisualizacaoSimulacaoSwing` implementa uma interface gráfica com:

- **Mapa**: Exibe zonas, estações e caminhões em posições geográficas.

- **Abas**: Mostram informações detalhadas sobre zonas, caminhões e estações.
- **Controles**: Botões para iniciar, pausar, reiniciar, calcular o mínimo de caminhões grandes e gerar relatórios.
- **Estatísticas**: Exibe tempo médio de espera, lixo coletado, lixo acumulado e número de caminhões grandes.

A interface usa `ListaEncadeada` para atualizar dinamicamente as informações exibidas.

8 Conclusão

As TADs `ListaEncadeada<T>` e `Fila<T>` são fundamentais para o sistema, suportando o gerenciamento de entidades e o processamento de filas. Apesar de limitações, como a falta de métodos utilitários e iteração simplificada, as estruturas atendem às necessidades do projeto e demonstram um bom domínio de conceitos de estruturas de dados.

O sistema oferece uma simulação robusta e visualmente rica, com potencial para análises logísticas e otimizações no gerenciamento de resíduos urbanos.

9 Apêndices

9.1 Termos e Definições

Termo	**Definição**
TAD	Tipo Abstrato de Dados - Modelo matemático definido por suas operações.
ListaEncadeada	Estrutura linear baseada em nós ligados, permitindo acesso sequencial.
Fila	Estrutura linear que segue o princípio FIFO (First-In-First-Out).
FIFO	Primeiro a entrar, primeiro a sair.
Complexidade	Medida do crescimento de tempo ou espaço em relação ao tamanho da entrada.
O(1)	Complexidade constante, independente do tamanho da entrada.
O(n)	Complexidade linear, proporcional ao tamanho da entrada.

Table 3: Termos e Definições