INSTITUTO FEDERAL CATARINENSE – CAMPUS RIO DO SUL

NOME: Pedro Henrique Korb 3ª Fase / BCC

Sistema de Atendimento Hospitalar com Estrutura de Listas Lineares

1. Introdução

O sistema modelado é voltado para a gestão de atendimentos hospitalares, focando nas fases de triagem, encaminhamento para fila de prioridade e consulta de pacientes. Utiliza-se estruturas de dados como pilhas e filas encadeadas. Foi dividido em três pastas para estruturação do projeto, sendo: Classes (representam a DTO das classes principais, contendo os atributos, construtor e métodos para manipulação de dados destas classes), Listas Lineares (contém as classes da estrutura das filas lineares – fila e pilha – no qual utiliza a classe Nodo com tipo genérico para que possa utilizar estas listas para todo o tipo de dado) e View (Contém o Main do projeto, onde há a instanciação dos objetos, organização dos métodos das classes em métodos estáticos para serem chamados de dentro de um while true e acompanhar a execução do sistema de atendimento).

2. Estrutura de Classes

2.1 Classe Paciente

Representa os pacientes do sistema, cada um possui: idPaciente, nomePaciente e uma lista de sintomas, o construtor define apenas o IdPaciente e nomePaciente, porque os sintomas devem ser adicionados através do método 'setSintomas' que pode receber inúmeras strings de sintomas, que serão todos adicionados dentro de uma ListaEncadeada (Explicação no item 3), também contém o método 'toString' para visualização dos dados do Paciente.

```
public Paciente(int idPaciente, String nomePaciente) {
    super();
    this.idPaciente = idPaciente;
    this.nomePaciente = nomePaciente;
    this.sintomas = new ListaEncadeada<>();
}
```

Imagem 1 – Construtor da classe

```java
public void setSintomas(String... sintomas) {
    for (String sintoma : sintomas) {
        this.sintomas.inserir(sintoma);
    }
}
```

Imagem 2 – Método 'setSintomas'

2.2 Classe Enfermeiro

Será responsável por realizar a triagem dos pacientes, possui os atributos: 'idEnfermeiro', 'nomeEnfermeiro' e 'disponivel', contém o construtor com os atributos das classes, métodos 'getters', 'setters' e 'toString' para visualização dos dados.

O atributo 'disponível' deve ser alterado para falso quando o enfermeiro realizar uma triagem.

```java
public Enfermeiro(int idEnfermeiro, String nomeEnfermeiro, boolean disponivel) {
    super();
    this.idEnfermeiro = idEnfermeiro;
    this.nomeEnfermeiro = nomeEnfermeiro;
    this.setDisponivel(disponivel);
}
```

Imagem 3 – Construtor da classe Enfermeiro

2.3 Classe Medico

Atua no atendimento dos pacientes após a triagem. Contém os atributos: 'idMedico', 'nomeMedico', 'especialidadeMedico' e 'disponivel', e seus métodos são semelhantes da classe 'Enfermeiro'.

```java
public Medico(int idMedico, String nomeMedico, String especialidadeMedico, boolean disponivel) {
    super();
    this.idMedico = idMedico;
    this.nomeMedico = nomeMedico;
    this.especialidadeMedico = especialidadeMedico;
    this.disponivel = disponivel;
}
```

Imagem 4 – Construtor da classe Medico

2.4 Classe SalaAtendimento

Representa uma sala física onde as consultas acontecem. Cada sala tem: 'idSala', 'numeroSala', 'especialidadeSala' e 'disponivel'. Métodos permitem definir e obter os dados do objeto.

```
public SalaAtendimento(int idSala, int numeracaoSala, String especialidadeSala, boolean disponivel) {
    super();
    this.idSala = idSala;
    this.numeracaoSala = numeracaoSala;
    this.especialidadeSala = especialidadeSala;
    this.disponivel = disponivel;
}
```

Imagem 5 – Contrutor da classe SalaAtendimento

2.5 Classe FilaChegada

Contém uma estrutura de lista encadeada que armazena pacientes recém-chegados ao hospital. Possui métodos para inserir pacientes na lista e mostrar todos os pacientes aguardando triagem, estes métodos são derivados da classe 'ListaEncadeada'.

Com a instanciação do objeto 'FilaChegada' sem parâmetros, já é criado uma fila automaticamente que será responsável por armazenar os dados dos pacientes.

```
public FilaChegada() {
    super();
    this.listaPaciente = new ListaEncadeada<>();
}
```

Imagem 6 – Construtor da classe FilaChegada

2.6 Classe FilaPrioridade

Gerencia os pacientes após a triagem, considerando a prioridade de atendimento (de 1 a 5). Utiliza um HashMap que associa níveis de prioridade (inteiros) a listas de pacientes. Permite adicionar pacientes, consultar listas por prioridade, verificar se a fila está vazia e mostrar todos os pacientes.

O construtor inicializa o HashMap (estrutura importada) com cinco listas de pacientes, não é necessário parâmetros.

```
public FilaPrioridade() {
    filasPorPrioridade = new HashMap<>();
    for (int i = 1; i <= 5; i++) {
        filasPorPrioridade.put(i, new ListaEncadeada<>());
    }
}
```

Imagem 7 – Construtor da classe FilaPrioridade

2.7 Classe PilhaObservacao

Representa uma pilha de pacientes em observação médica. Permite adicionar pacientes e retirá-los do topo da pilha. Esta classe utiliza a estrutura de pilhas, criada dentro deste mesmo projeto na pasta 'ListasLineares'. O construtor da classe apenas inicializa um objeto 'PilhaEncadeada<Paciente> pilhaPaciente'.

2.8 Classe Triagem

Coordena o processo de triagem: Recebe pacientes da FilaChegada, associa um Enfermeiro para realizar a triagem e envia o paciente para a FilaPrioridade, definindo seu nível de prioridade. O construtor recebe a lista de chegada e a fila de prioridade, e as triagens são realizadas através do método 'realizaTriagem', recebendo primeiro paciente da fila de chegada, o enfermeiro disponível e o nível de risco do paciente, para que seja adicionado corretamente na fila de prioridade.

```
public Triagem(FilaChegada listaChegada, FilaPrioridade listaPrioridade) {
    super();
    this.listaChegada = listaChegada;
    this.listaPrioridade = listaPrioridade;
}
```

Imagem 8 – Construtor da classe Triagem

```
public void realizaTriagem(Paciente paciente, Enfermeiro enfermeiro, int prioridade) {
    this.listaChegada.getListaPaciente().removerPorPosicao(0);
    this.listaPrioridade.adicionarPaciente(prioridade, paciente);
}
```

Imagem 9 – Método realizaTriagem

2.9 Classe Consulta

Gerencia o processo de atendimento médico: Associa pacientes triados a médicos e salas disponíveis. Se necessário, move pacientes para a PilhaObservacao. Utiliza listas e pilhas para administrar o fluxo de atendimento.

Nesta classe, a consulta é realizada através do próprio construtor da classe, sendo o método principal. Seus parâmetros são: 'pilhaObservacao' (no qual o paciente pode ou não ser enviado para aguardar um leito no hospital), listaPrioridade (recebe a instancia do HashMap de prioridade), 'salaAtendimento' (recebe uma sala de atendimento disponível), 'medico' (recebe um médico disponível para realizar a consulta) e o parâmetro boolean 'observacao' que indica se o paciente vai para a pilha de observação ou vai ser liberado após a consulta.

O paciente da consulta é extraído automaticamente através do método ‘this.listaPrioridade.retirarPaciente()’, no qual retira o paciente de mais alta prioridade, não sendo necessário informar o paciente no método construtor.

```
public Consulta(PilhaObservacao pilhaObservacao, FilaPrioridade listaPrioridade, SalaAtendimento salaAtendimento,
        Medico medico, boolean observacao) {
    super();
    this.pilhaObservacao = pilhaObservacao;
    this.listaPrioridade = listaPrioridade;
    this.paciente = this.listaPrioridade.retirarPaciente();
    this.salaAtendimento = salaAtendimento;
    this.medico = medico;
    this.observacao = observacao;
    if (this.paciente == null) {
        return;
    }

    if (observacao) {
        this.pilhaObservacao.adicionarPaciente(this.paciente);
    }

    this.medico.setDisponibilidade(false);
    this.salaAtendimento.setDisponivel(false);
}
```

Imagem 10 – Construtor da classe Consulta

3. Estrutura de Dados – Pilha e Fila

O sistema utiliza estruturas de dados encadeadas personalizadas para manipular listas e pilhas de forma eficiente. Essas estruturas são genéricas (<T>) e permitem trabalhar com qualquer tipo de objeto, como pacientes, médicos ou atendimentos.

3.1 Classe Nodo<T>

Elemento básico das estruturas encadeadas (pilha e fila). Contém os atributos: dado (Armazena o valor do tipo genérico <T>), próximo (faz referência para o próximo Nodo na estrutura). Essa classe é a base para a construção tanto da lista encadeada quanto da pilha encadeada.

O construtor inicializa o Nodo com um dado e define que o próximo é null, a definição de quais são os próximos Nodos é feita nas classes das listas.

```
T dado;
Nodo<T> proximo;

public Nodo(T dado) {
    this.dado = dado;
    this.proximo = null;
}
```

Imagem 11 – Atributos e construtor da classe Nodo

3.2 Classe ListaEncadeada<T>

Estrutura de lista linear que permite a manipulação de dados genéricos. Quando o construtor é inicializado, os seguintes atributos são definidos: inicio (faz referência para o primeiro nodo da lista) e tamanho (indica o número de elementos na lista).

Métodos principais:

- inserir(valor: T): Adiciona um novo elemento ao final da lista (imagem 12);

```
public void inserir(T valor) {
    Nodo<T> novo = new Nodo<>(valor);
    if (inicio == null) {
        inicio = novo;
    } else {
        Nodo<T> atual = inicio;
        while (atual.proximo != null) {
            atual = atual.proximo;
        }
        atual.proximo = novo;
    }
    tamanho++;
}
```

Imagem 12

- mostrarTela(): Imprime os elementos da lista.

```
public void mostrarTela() {
    Nodo<T> atual = inicio;
    while (atual != null) {
        System.out.println(atual.dado);
        atual = atual.proximo;
    }
}
```

Imagem 13

- getTodos(): Retorna todos os elementos da lista em um ArrayList.

```java
public ArrayList<T> getTodos() {
    ArrayList<T> lista = new ArrayList<>();
    Nodo<T> atual = inicio;
    while (atual != null) {
        lista.add(atual.dado);
        atual = atual.proximo;
    }
    return lista;
}
```

Imagem 14

- getProximo(valor: T) / getAnterior(valor: T): Retorna o elemento seguinte ou anterior ao informado.

```java
public T getProximo(T valor) {
    Nodo<T> atual = inicio;
    while (atual != null && atual.proximo != null) {
        if (atual.dado.equals(valor)) {
            return atual.proximo.dado;
        }
        atual = atual.proximo;
    }
    return null;
}
```

Imagem 15

```java
public T getAnterior(T valor) {
    Nodo<T> anterior = null;
    Nodo<T> atual = inicio;

    while (atual != null) {
        if (atual.dado.equals(valor)) {
            return anterior != null ? anterior.dado : null;
        }
        anterior = atual;
        atual = atual.proximo;
    }
    return null;
}
```

Imagem 16

- removerPorValor(valor: T) Remove um elemento específico da lista.

```java
public boolean removerPorValor(T valor) {
    if (inicio == null) return false;

    if (inicio.dado.equals(valor)) {
        inicio = inicio.proximo;
        tamanho--;
        return true;
    }

    Nodo<T> anterior = inicio;
    Nodo<T> atual = inicio.proximo;

    while (atual != null) {
        if (atual.dado.equals(valor)) {
            anterior.proximo = atual.proximo;
            tamanho--;
            return true;
        }
        anterior = atual;
        atual = atual.proximo;
    }
    return false;
}
```

Imagem 17

- ordenar(comparator): Permite ordenar a lista (utiliza a classe Comparator)

```java
public void ordenar(Comparator<T> comparator) {
    if (inicio == null || inicio.proximo == null) return;

    ArrayList<T> lista = getTodos();
    lista.sort(comparator);

    inicio = null;
    tamanho = 0;

    for (T item : lista) {
        inserir(item);
    }
}
```

Imagem 18

- removerUltimo(): Remove o último elemento da lista.

```java
public boolean removerUltimo() {
    if (inicio == null) return false;

    if (inicio.proximo == null) {
        inicio = null;
        tamanho--;
        return true;
    }

    Nodo<T> atual = inicio;
    while (atual.proximo.proximo != null) {
        atual = atual.proximo;
    }

    atual.proximo = null;
    tamanho--;
    return true;
}
```

Imagem 19

- tamanho(), estaVazia(): Retornam o tamanho da lista ou se ela está vazia.
- getPrimeiro(): Retorna o primeiro elemento da lista.
- contem(paciente: T): Verifica se um valor está presente na lista.

```java
public int tamanho() {
    return tamanho;
}

public boolean estaVazia() {
    return inicio == null;
}

public T getPrimeiro() {
    if (estaVazia()) {
        return null;
    }
    return inicio.getDado();
}

public boolean contem(T paciente) {
    Nodo<T> atual = inicio;
    while (atual != null) {
        if (atual.getDado().equals(paciente)) {
            return true;
        }
        atual = atual.proximo;
    }
    return false;
}
```

Imagem 20 – Métodos 'tamanho()', 'estaVazia()', 'getPrimeiro()' e 'contem()'

3.3 Classe PilhaEncadeada<T>

Implementação de uma pilha (estrutura LIFO - Last In, First Out) usando nodos encadeados. Na instanciação da pilha, o atributo topo (referência para o elemento que está no topo da pilha) é definido como null.

Métodos principais:

- empilhar(dado: T): Adiciona um novo elemento no topo da pilha.

```
public void empilhar(T dado) {
    Nodo<T> novo = new Nodo<>(dado);
    novo.setProximo(topo);
    topo = novo;
}
```

Imagem 21

- desempilhar(): Remove e retorna o elemento do topo.

```
public T desempilhar() {
    if (estaVazia()) {
        return null;
    }
    T dado = topo.getDado();
    topo = topo.getProximo();
    return dado;
}
```

Imagem 22

- verTopo(): Retorna o elemento no topo sem removê-lo.

- estaVazia(): Verifica se a pilha está vazia.
- mostrarTela(): Exibe os elementos da pilha.
- tamanho(): Retorna a quantidade de elementos na pilha.

```java
public T verTopo() {
    if (estaVazia()) {
        return null;
    }
    return topo.getDado();
}

public boolean estaVazia() {
    return topo == null;
}

public void mostrarTela() {
    Nodo<T> atual = topo;
    while (atual != null) {
        System.out.println(atual.getDado());
        atual = atual.getProximo();
    }
}

public int tamanho() {
    int contador = 0;
    Nodo<T> atual = topo;
    while (atual != null) {
        contador++;
        atual = atual.getProximo();
    }
    return contador;
}
```

Imagem 23

4. Classe View

A classe Main, localizada na pasta View, representa a camada de visualização e controle da simulação do sistema. Essa classe é responsável por integrar e simular o funcionamento de todas as classes do sistema, e interage com o usuário através do console.

No início é criado instâncias das filas (FilaChegada, FilaPrioridade), pilha (PilhaObservacao), e objetos envolvidos: Paciente, Enfermeiro, Medico, SalaAtendimento eTriagem.

Simulação por tempo: Utiliza um while true para simular o avanço de tempo em ciclos de 15 minutos, onde cada ciclo representa uma nova etapa no fluxo de atendimento dos pacientes.

A cada ciclo, tem a probabilidade – randomicamente - de acontecer as seguintes situações: Inserção de um novo paciente na fila de chegada, realização de triagem com um enfermeiro disponível ou realização de consulta com um médico disponível em uma sala sorteada.

A saída no console é feita utilizando métodos ‘echo’ (write), ‘wbold’ (write bold)’ e ‘wred’ (write red), para visualizar facilmente o estado atual do sistema, mostrando quais pacientes estão em cada estrutura e qual o progresso de cada um.

4.1. Teste de Implementação

- Ao executar a classe main, é mostrado que todas as filas estão vazias, e que está pronta para receber dados ao apertar a tecla ENTER

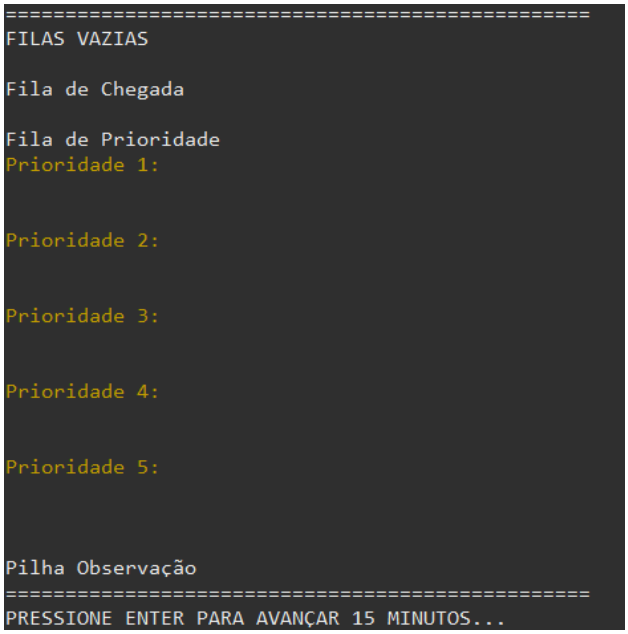

Imagem 24

- Ao clicar ENTER, há a possibilidade de acontecer alguns processos, após 60 minutos, um paciente entrou na fila:

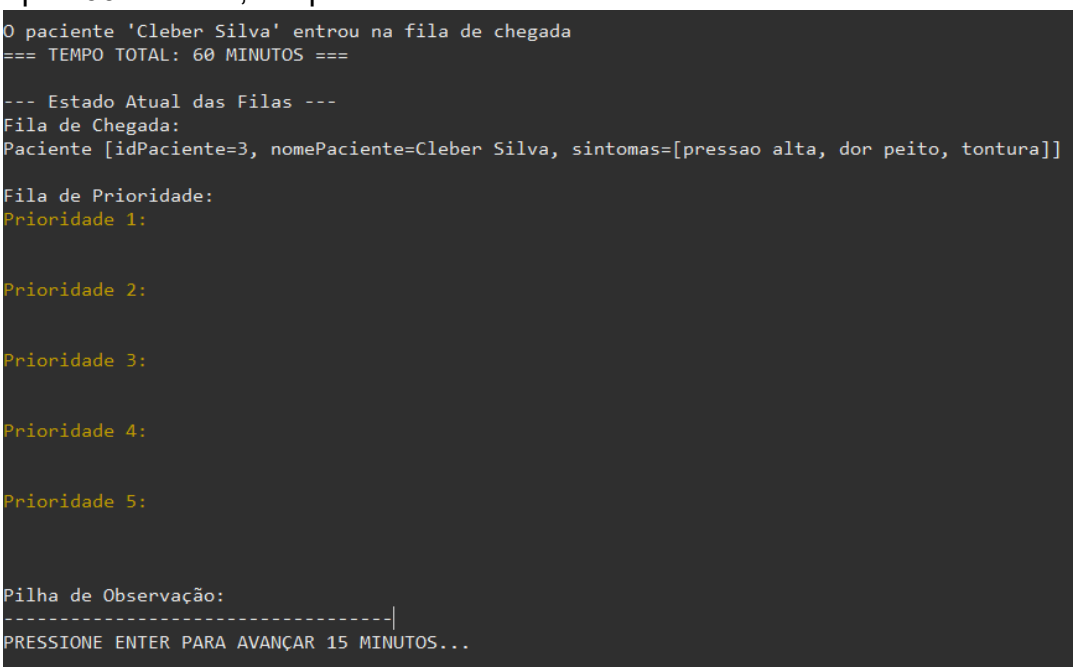

Imagem 25

- O paciente ‘Cleber Silva’ foi triado e foi para fila de prioridade com prioridade 5.

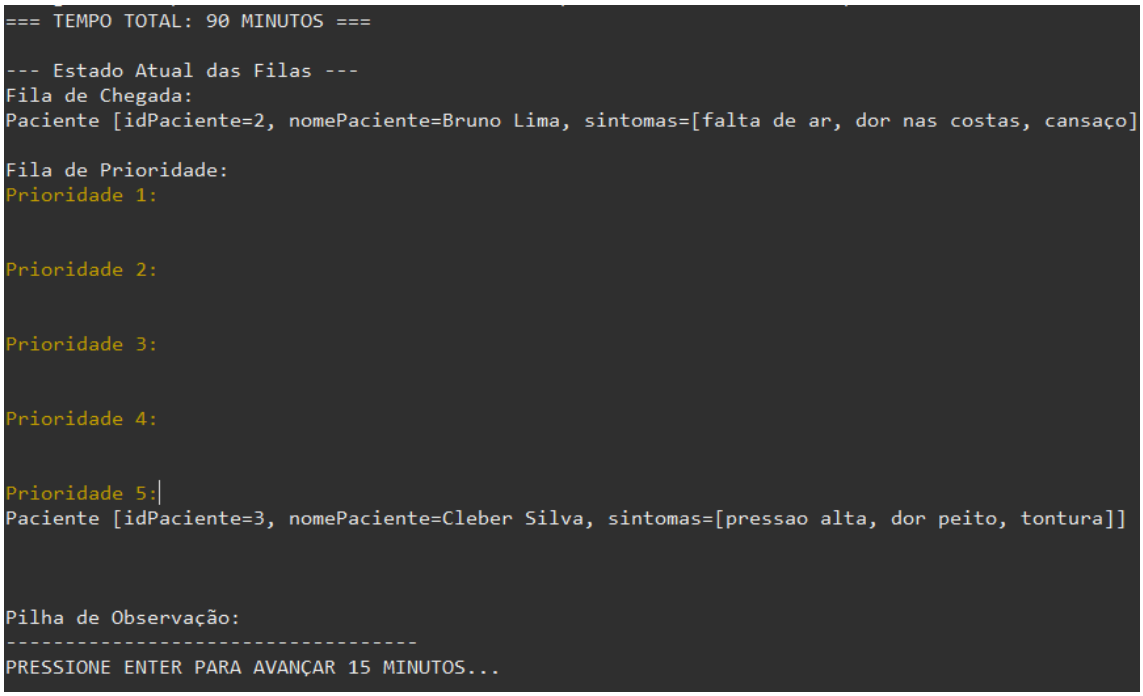

Imagem 26

- Após 150 minutos o paciente ‘Gustavo Nunes’ foi consultado com Dr. Carlos na sala 100 e após a consulta foi movido para a pilha de observação. Nesse período o paciente ‘Bruno Lima’ foi triado e enviado para fila de prioridade 4

```
Paciente consulta: Gustavo Nunes
Consulta realizada com Dr. Carlos na sala 100
Paciente movido para Pilha de observação para aguardo de leito
=== TEMPO TOTAL: 150 MINUTOS ===

--- Estado Atual das Filas ---
Fila de Chegada:

Fila de Prioridade:
Prioridade 1:


Prioridade 2:


Prioridade 3:


Prioridade 4:
Paciente [idPaciente=2, nomePaciente=Bruno Lima, sintomas=[falta de ar, dor nas costas, cansaço]]


Prioridade 5:
Paciente [idPaciente=3, nomePaciente=Cleber Silva, sintomas=[pressao alta, dor peito, tontura]]



Pilha de Observação:
Paciente [idPaciente=7, nomePaciente=Gustavo Nunes, sintomas=[visão embaçada, dores nos olhos, dor de cabeça]]
-----------------------------------
PRESSIONE ENTER PARA AVANÇAR 15 MINUTOS...
```

Imagem 27

- O sistema continua dessa maneira em um loop até todos os pacientes pré-definidos acabarem.

5. Conclusão

A maioria das funcionalidades exigidas nos requisitos do projeto foi implementada com sucesso, incluindo os itens 1, 2, 3, 4 e 6. No entanto, o item 5, referente a *"Relatórios e Estatísticas"*, que incluía métricas como total de pacientes atendidos por prioridade, tempo médio de espera, número de pacientes em observação e desistências por espera excessiva, não foi implementado devido à limitação de tempo por parte do autor. Apesar disso, o projeto cumpre satisfatoriamente os demais requisitos propostos.

O código-fonte completo pode ser acessado no seguinte link: https://github.com/Pedro-Korb/sistemaHospitalar.git