**Pontifícia Universidade Católica de Minas Gerais Instituto de Ciências Exatas e Informática**

**Nome:** Gabriel Henrique Vieira de Oliveira, Vinicius Cezar Pereira Menezes
**Data:** 16/10/2025
**Matéria:** Algoritmos e Estruturas de Dados 3
**Link Github:** https://github.com/GabrielHenrique20/TP_AEDS3

---

## A) Qual a estrutura usada para representar os registros?

No nosso projeto, os registros são representados por Classes localizadas no pacote **model,** como por exemplo Registro.java (Registro para a Viagem), RegistroAtividade.java, RegistroCategoria.java e RegistroUsuario.java. Essas classes guardam os dados de cada tipo de entidade (atividade, categoria, usuário e viagem), funcionando como estruturas para armazenar e manipular as informações dos registros no sistema.

Vamos exemplificar cada uma delas:

1) **Registro.java:**

   *Interface para registro de Viagens.
   *Métodos: setId(int i), getId(), toByteArray(), fromByteArray(byte[] b) e **getIdUsuario**.
   *Permite alterar o ID e transformar os dados em um formato que pode ser enviado ou recebido, além de recuperar esses dados de volta ao formato original.

2) **RegistroAtividade.java:**

   *Interface para registro de Atividades.
   *Mesmo métodos presentes no Registro.java, **(tirando o getIdUsuario).**

3) **RegistroCategoria.java:**

   *Interface simples para registro de Categorias.
   *Mesmo métodos presentes no Registro.java, **(tirando o getIdUsuario)**.

4) **RegistroUsuario.java:**

   *Interface para registro de Usuários.
   *Mesmo métodos presentes no Registro.java, **(tirando o getIdUsuario).**

Ou seja, basicamente todos seguem o mesmo padrão: definem métodos para manipular o ID e converter o registro para/desde um array de bytes, facilitando a persistência e leitura dos dados em arquivos binários.

TRABALHO2 > src > model > Registro.java > {} model

```java
package model;
import java.io.IOException;

public interface Registro {
    public void setId(int i);
    public int getId();
    public byte[] toByteArray() throws IOException;
    public void fromByteArray(byte[] b) throws IOException;
    public int getIdUsuario();
}
```

## B) Como atributos multivalorados do tipo string foram tratados?

Em nosso projeto, há a presença do atributo multivalorado **Telefone** ligado à entidade **Usuário** (tratado como um array de string, permitindo que o usuário tenha vários telefones linkados a ele). Ele é tratado da seguinte maneira:

No método **toByteArray()**, escreve-se a quantidade de telefones (this.telefone.length) e, em seguida, para cada telefone grava o tamanho e os bytes da string.

No método **fromByteArray(byte[] b)**, lê a quantidade de telefones e, em seguida, para cada telefone, lê o tamanho e os bytes, reconstruindo o array de strings.

Assim, o atributo multivalorado é armazenado e recuperado corretamente como uma lista de valores, mantendo a flexibilidade de múltiplos telefones por usuário.

```java
// Serialização
@Override
public byte[] toByteArray() throws IOException {
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    DataOutputStream dos = new DataOutputStream(baos);

    dos.writeInt(this.id);

    // nome
    byte[] bytesNome = this.nome.getBytes(charsetName:"UTF-8");
    dos.writeShort(bytesNome.length);
    dos.write(bytesNome);

    // telefone
    if (this.telefone != null) {
        dos.writeInt(this.telefone.length);
        for (String tel : this.telefone) {
            byte[] bytesTel = tel.getBytes(charsetName:"UTF-8");
            dos.writeShort(bytesTel.length);
            dos.write(bytesTel);
        }
    } else {
        dos.writeInt(v:0); // nenhum telefone
    }

    // email
    byte[] bytesT = this.email.getBytes(charsetName:"UTF-8");
    dos.writeShort(bytesT.length);
    dos.write(bytesT);

    return baos.toByteArray();
}
```

```java
@Override
public void fromByteArray(byte[] b) throws IOException {
    ByteArrayInputStream bais = new ByteArrayInputStream(b);
    DataInputStream dis = new DataInputStream(bais);

    this.id = dis.readInt();

    // nome
    short tamNome = dis.readShort();
    byte[] bytesNome = new byte[tamNome];
    dis.readFully(bytesNome);
    this.nome = new String(bytesNome, charsetName:"UTF-8");

    // telefones
    int qtd = dis.readInt();
    this.telefone = new String[qtd];
    for (int i = 0; i < qtd; i++) {
        short tamTel = dis.readShort();
        byte[] bytesTel = new byte[tamTel];
        dis.readFully(bytesTel);
        this.telefone[i] = new String(bytesTel, charsetName:"UTF-8");
    }

    // email
    short tamEmail = dis.readShort();
    byte[] bytesEmail = new byte[tamEmail];
    dis.readFully(bytesEmail);
    this.email = new String(bytesEmail, charsetName:"UTF-8");
}
```

## C) Como foi implementada a exclusão lógica?

A exclusão lógica foi implementada usando o conceito de lápide, tendo de início a seguinte estrutura nos códigos:

**Estrutura da Lápide:**

- Cada registro no arquivo possui um byte inicial que serve como lápide.
- ' ' (espaço) indica registro válido.
- '*' indica registro excluído (delete).

No código abaixo temos a função **delete(int id)**, presente dentro do pacote dao e as suas respectivas classes Arquivo., que implementa a exclusão lógica da seguinte forma:

```java
public boolean delete(int id) throws Exception {
    arquivo.seek(TAM_CABECALHO);
    while (arquivo.getFilePointer() < arquivo.length()) {
        long posicao = arquivo.getFilePointer();
        byte lapide = arquivo.readByte();
        short tamanho = arquivo.readShort();
        byte[] dados = new byte[tamanho];
        arquivo.read(dados);

        if (lapide == ' ') {
            T obj = construtor.newInstance();
            obj.fromByteArray(dados);
            if (obj.getId() == id) {
                arquivo.seek(posicao);
                arquivo.writeByte('*');
                addDeleted(tamanho, posicao);
                return true;
            }
        }
    }
    return false;
}
```

**Gerenciamento de Espaço Livre:**
O sistema mantém uma lista encadeada de espaços excluídos que podem ser reutilizados:

Cabeçalho do arquivo:

-> Primeiros 4 bytes: último ID usado.
-> Próximos 8 bytes: ponteiro para o primeiro espaço livre.

Métodos de gerenciamento:

-> **addDeleted():** adiciona um espaço livre à lista.
-> **getDeleted():** busca um espaço livre adequado para reutilização.

**Reutilização de Espaço:**
Quando um novo registro é criado no método **create(T obj)**, acontecem as seguintes situações:

-> Primeiro verifica se existe um espaço excluído adequado
-> Se encontrar, reutiliza esse espaço
-> Se não encontrar, adiciona no final do arquivo

```java
public int create(T obj) throws Exception {
    arquivo.seek(0);
    int novoID = arquivo.readInt() + 1;
    arquivo.seek(0);
    arquivo.writeInt(novoID);
    obj.setId(novoID);
    byte[] dados = obj.toByteArray();

    // inserir na árvore B+
    relviagemUsuarioArvoreBMais.create(new ParIntInt(obj.getIdUsuario(), obj.getId()));

    long endereco = getDeleted(dados.length);
    if (endereco == -1) {
        arquivo.seek(arquivo.length());
        endereco = arquivo.getFilePointer();
        arquivo.writeByte(' ');  // Lápide
        arquivo.writeShort(dados.length);
        arquivo.write(dados);
    } else {
        arquivo.seek(endereco);
        arquivo.writeByte(' ');  // Remove a lápide
        arquivo.skipBytes(2);
        arquivo.write(dados);
    }
    return obj.getId();
}
```

## D) Além das PKs, quais outras chaves foram utilizadas nesta etapa?

Além das PKs, em nosso código há a existência de outros tipos de chaves bem importantes para a implementação, sendo elas:

**1) Na classe Atividade.java:** temos a presença da FK idCategoria, que é responsável por estabelecer um relacionamento 1:N entre Categoria -> Atividade, onde uma categoria pode ter múltiplas atividades.

**2) Na classe Viagem.java:** temos a presença da FK idUsuario, que é responsável por estabelecer um relacionamento 1:N entre Usuário -> Viagem, onde um usuário pode ter múltiplas viagens.

**3) Índices secundários** baseados em Árvore B+ para materializar os **relacionamentos 1:N**. Para Usuário -> Viagem, cada viagem gera um par ParIntInt(idUsuario, idViagem) persistido em relUsuarioViagem.db; para Categoria -> Atividade, cada atividade gera ParIntInt(idCategoria, idAtividade) em relCategoriaAtividade.db. Esses índices permitem listar rapidamente todos os filhos de um pai e são mantidos pelos DAOs nas operações de inserção, atualização (se trocar o pai) e exclusão.

## E) Quais tipos de estruturas (hash, B+ Tree, extensível, etc.) foram utilizadas para cada chave de pesquisa?

Para cada chave de pesquisa, foram utilizadas as seguintes estruturas:

1) **PKs:** a estrutura utilizada foi a **Árvore B+**, onde cada entidade principal tem seu próprio arquivo principal, gerenciado por um objeto do tipo **ArvoreBMais<entidade>.**

```
// árvores B+
ArvoreBMais<ParIntInt> relviagemUsuarioArvoreBMais;
```

2) **FKs:** a estrutura utilizada também foi a **Árvore B+** (índices secundários com pares **ParIntInt** para captar as relações 1:N). Atua com os **índices secundários**, permitindo listar rapidamente todos os filhos de um elemento pai. O primeiro campo do par (idPai) é usado como chave de pesquisa na B+, e o segundo (idFilho) é usado como referência.

## F) Como foi implementado o relacionamento 1:N (explique a lógica da navegação entre registros e integridade referencial)?

O relacionamento 1:N foi implementado por meio de índices secundários baseados em Árvores B+ que associam o identificador do registro “pai” ao identificador do registro “filho”.

**Para Usuário -> Viagem**, cada viagem armazena o campo idUsuario, e um índice secundário (relUsuarioViagem.db) guarda pares ParIntInt(idUsuario, idViagem).

**Para Categoria -> Atividade**, cada atividade armazena o campo idCategoria, e o índice secundário (relCategoriaAtividade.db) guarda ParIntInt(idCategoria, idAtividade).

- **Lógica:** Ao buscar todas as viagens de um usuário, o sistema pesquisa na Árvore B+ todos os pares cujo primeiro campo (idUsuario) corresponde ao ID do usuário. Em seguida, usa o segundo campo (idViagem) para localizar cada viagem no arquivo principal (viagens.db). O mesmo ocorre com categorias e atividades. Observação importante: para vincular várias viagens a um usuário, foi criado o método

associarViagemUsuario(), responsável por estabelecer essa relação já no menu da viagem. Em seguida, as viagens associadas podem ser exibidas por meio do método listarViagensUsuario(). Já na relação entre Categoria e Atividade, a associação é realizada no momento da criação da atividade; as categorias são previamente cadastradas, seus respectivos IDs são obtidos, e, ao criar uma nova atividade, o ID da categoria correspondente é atribuído diretamente a ela. Posteriormente, o mesmo método de listagem é utilizado para exibir as atividades e suas categorias no terminal.

```java
private void associarViagemUsuario() {
    System.out.print(s:"\nID do Usuário: ");
    int userId = console.nextInt();
    console.nextLine();
    try {
        Usuario u = usuarioDAO.buscarUsuario(userId);
        if (u == null) {
            System.out.println(x:"Usuário não encontrado.");
            return;
        }

        System.out.print(s:"ID da Viagem a associar: ");
        int viagemId = console.nextInt();
        console.nextLine();

        Viagem v = viagemDAO.buscarViagem(viagemId);
        if (v == null) {
            System.out.println(x:"Viagem não encontrada.");
            return;
        }

        ParIntInt rel = new ParIntInt(userId, viagemId);
        boolean ok = relviagemUsuarioArvoreBMais.create(rel);
        if (ok) {
            System.out.println(x:"Associação usuário->viagem criada com sucesso.");
        } else {
            System.out.println(x:"Não foi possível criar a associação (talvez já exista)
        }
    } catch (Exception e) {
        System.out.println("Erro ao associar viagem: " + e.getMessage());
        e.printStackTrace();
    }
}
```

```java
private void listarViagensUsuario() {
    System.out.print(s:"\nID do Usuário: ");
    int userId = console.nextInt();
    console.nextLine();

    try {
        Usuario u = usuarioDAO.buscarUsuario(userId);
        if (u == null) {
            System.out.println(x:"Usuário não encontrado.");
            return;
        }

        ArrayList<ParIntInt> rels = relviagemUsuarioArvoreBMais.read(new ParIntInt(userId, -1));
        if (rels.isEmpty()) {
            System.out.println(x:"Nenhuma viagem associada a este usuário.");
            return;
        }

        System.out.println("\nViagens do usuário " + userId + ":");
        for (ParIntInt p : rels) {
            Viagem v = viagemDAO.buscarViagem(p.get2());
            if (v != null)
                System.out.println(v);
            else
                System.out.println("Viagem id=" + p.get2() + " (não encontrada)");
        }
    } catch (Exception e) {
        System.out.println("Erro ao listar viagens do usuário: " + e.getMessage());
        e.printStackTrace();
    }
}
```

- **Integridade Referencial:** A integridade é garantida via código nos DAOs: antes de inserir um filho, o sistema verifica se o pai existe; e ao excluir um pai, ele remove primeiro todos os filhos associados (exclusão em cascata). Assim, evita registros órfãos e mantém a consistência entre os arquivos.

## G) Como os índices são persistidos em disco? (formato, atualização, sincronização com os dados).

Os índices são armazenados em arquivos binários .db como estruturas de Árvore B+, onde as chaves e ponteiros são gravados de forma ordenada.

1) **Formato:** Cada índice (primário ou secundário) é armazenado em um arquivo separado**.** Os registros e as chaves são serializados em bytes antes de serem gravados, permitindo leitura direta (sem depender de banco relacional). Cada nó da Árvore B+ contém as chaves ordenadas e ponteiros para outros nós ou posições de registro no arquivo principal.

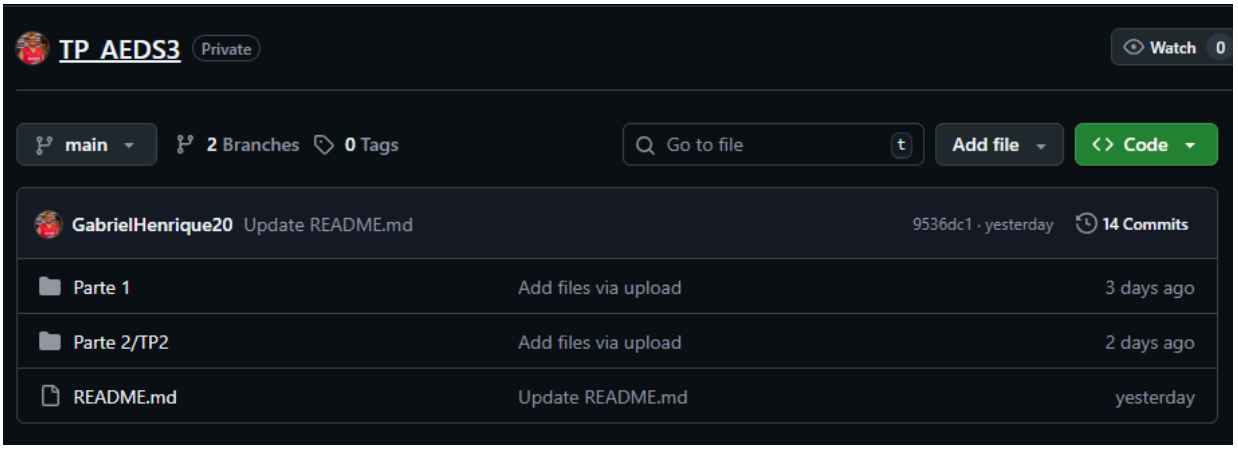
2) **Atualização:** Toda vez que ocorrer uma inserção, alteração ou exclusão, a DAO da entidade correspondente vai atualizar o arquivo principal. Em seguida, atualiza o índice associado, criando, removendo ou modificando o par na B+. O índice é mantido sincronizado com os dados, de modo que cada modificação reflete imediatamente na estrutura de busca.

3) **Sincronização:** O sistema garante consistência lógica entre arquivos. Quando inserir um filho, grava o registro e logo após o par no índice. Ao excluir um pai, remove primeiro os filhos e seus pares nos índices secundários. Então, mesmo que os dados e índices fiquem em arquivos diferentes, ambos permanecem coerentes e atualizados.

## H) Como está estruturado o projeto no GitHub (pastas, módulos, arquitetura)?

Para o projeto, foi criado um repositório central no perfil do Gabriel Henrique. A partir dele, foram estabelecidas branches distintas, permitindo que cada membro da equipe desenvolva, teste e modifique o código de forma isolada, sem impactar a estabilidade das ramificações principais.

No GitHub, o projeto do Sistema de Viagens está estruturado da seguinte maneira:




**1ª pasta:** Contém toda a parte 1 do nosso trabalho prático, ou seja, o primeiro CRUD de viagem feito além da documentação e seus respectivos diagramas.

**2ª pasta:** Pasta com aprimoramento da parte 1, contendo os seguintes pacotes:
*__O pacote aeds3__ contém as estruturas de dados implementadas pelo professor Kutova.
*__O pacote controller__ contém os menus e opções de interação com o usuário.
*__O pacote dao__ realiza a comunicação entre os modelos e os arquivos de dados.
*__O pacote model__ constitui-se das classes das entidades e seus respectivos registros.
*__O pacore views__ contém a presença da classe Principal do código.
*__As 4 classes Buscar__. são responsáveis pela busca dos registros existentes.

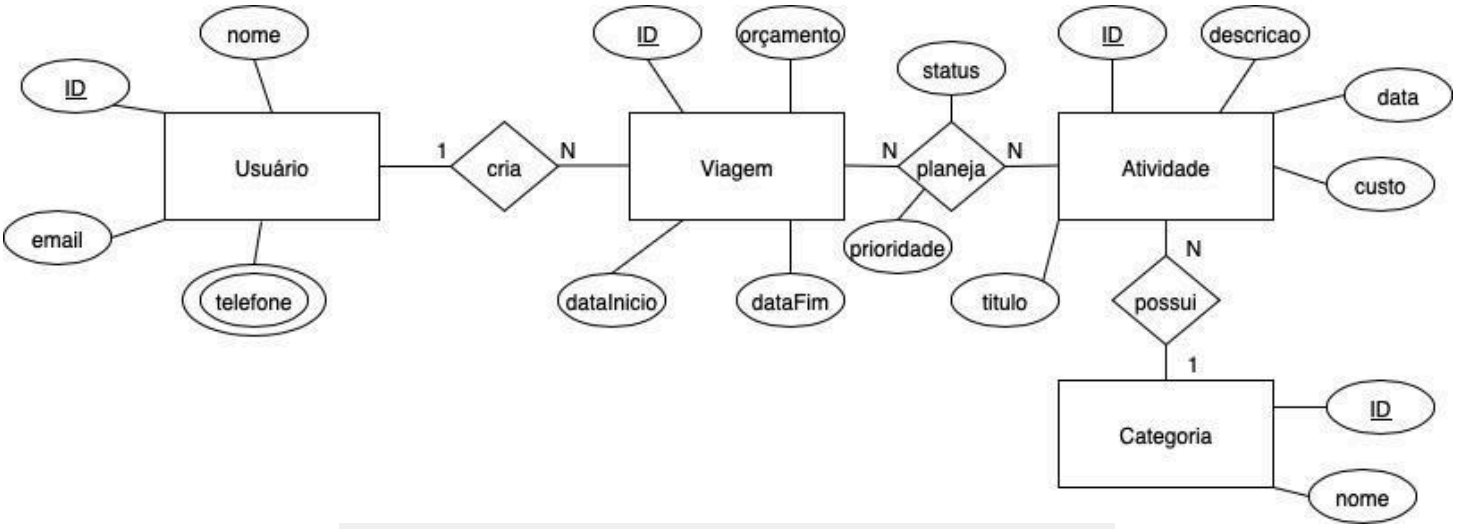
**3ª informação:** README.md contendo informações adicionais sobre nosso projeto.

## Estrutura do Projeto

1. O usuário inicia o programa.
2. É exibido um menu com opções de gerenciamento (Usuário, Viagem, Categoria, Atividade).
3. O usuário pode **adicionar**, **editar**, **buscar**, **remover** ou **listar** registros.
4. As informações são salvas e recuperadas utilizando as estruturas implementadas.

```
├── src/
│   ├── aeds3/ # Estruturas de dados (Árvore B+ e ParIntInt)
│   ├── controller/ # Menus e controle de navegação
│   ├── dao/ # Classes de persistência (DAO)
│   ├── model/ # Classes de modelo (Usuario, Viagem, Atividade, Categoria)
│   ├── views/ # Classe Principal.java
│   ├── Buscar*.java # Classes de busca de registros
│   └──
```

# DIAGRAMAS:


nome
ID
Usuário
email
telefone
1
cria
N
ID
orçamento
Viagem
dataInicio
dataFim
N
planeja
N
status
prioridade
ID
descricao
Atividade
data
custo
titulo
N
possui
1
Categoria
ID
nome

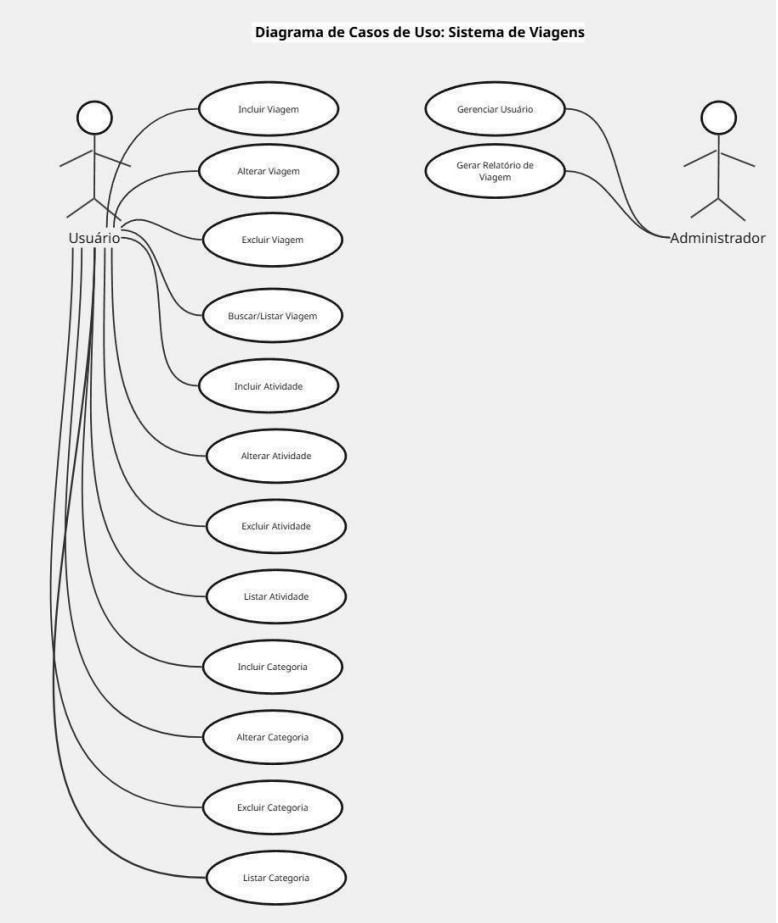


Diagrama de Casos de Uso: Sistema de Viagens
Usuário
Incluir Viagem
Alterar Viagem
Excluir Viagem
Buscar/Listar Viagem
Incluir Atividade
Alterar Atividade
Excluir Atividade
Listar Atividade
Incluir Categoria
Alterar Categoria
Excluir Categoria
Listar Categoria
Gerenciar Usuário
Gerar Relatório de Viagem
Administrador